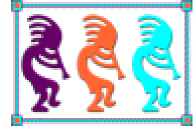

Creating ActiveX Controls for VFP Using .Net

Doug Hennig
Stonefield Software Inc.
2323 Broad Street
Regina, SK Canada S4P 1Y9
Email: dhennig@stonefield.com
Web site: www.stonefieldquery.com
Blog: DougHennig.BlogSpot.com
Twitter: DougHennig

ActiveX controls provide a way to add both functionality and a modern appearance to your VFP applications. Unfortunately, there are few new ActiveX controls these days, especially ones that take advantage of features in Windows Vista and Windows 7. On the other hand, the .Net framework comes with many attractive looking controls, and there's an entire industry devoted to providing even more. This document discusses techniques for creating ActiveX controls from .Net components so we can take advantage of these controls without having to convert our applications to .Net.

Introduction

One of the ways to extend the life of your VFP applications is to make them look more modern. Microsoft Office is the standard bearer for Windows applications, so updating your user interface to resemble parts of Office will give your applications a fresh and yet comfortable appearance. UI elements you should target include menus, toolbars, open and save dialogs (which have a new appearance and more functionality in Windows Vista and later), datetime pickers, progress bars, and so on.

There are several ways you can add more modern controls to your VFP applications. One is to use the various projects on VFPX (http://vfpx.codeplex.com), such as the ThemedControls project or the controls in the Ctl32 library. Another is to use ActiveX controls, which usually provide features native controls can't. Unfortunately, the ActiveX controls that come with VFP look fairly dated these days and few developers are creating ActiveX controls anymore as they've moved on to creating .Net controls.

Speaking of .Net, it comes with quite a few attractive looking native controls, such as:

- ProgressBar, which presents a Vista/Windows 7-like appearance
- ToolStrip, which provides an Office-like toolbar
- MenuStrip, which displays an Office-like menu system
- ContextMenuStrip, which presents an Office-like shortcut menu

In addition to the native controls, many companies, such as Telerik and Infragistics, sell Windows Forms controls, and there are thousands of free ones available. If we could find some way to leverage .Net controls in VFP applications, we'd have an almost limitless source of new, modern-looking controls with functionality difficult or impossible to create natively in VFP.

Fortunately, there is a way to do that. It's possible to create ActiveX versions of .Net controls that can be used in VFP or any development environment that can host ActiveX controls. You don't need any expensive add-ons for Visual Studio (VS). In fact, you don't even need to purchase Visual Studio; you can download the free Visual Studio Express (http://www.microsoft.com/express/Windows) and use it to create all the ActiveX controls you wish (although it's not nearly as easy as VS Express is missing some of the tools that we'll use in VS).

Creating ActiveX versions of .Net controls has several benefits:

- You get the functionality and modern UI of these controls without much work.
- You don't have to rewrite your VFP applications to take advantage of .Net.
- You can use these controls as a migration path if you do intend to one day rewrite your applications.

This document provides a cookbook-like approach to creating ActiveX controls using .Net. Most of the samples are presented in C#, but I don't expect you to be familiar with that language, as I'm certainly no expert myself.

Acknowledgment

Before we get started, I'd like to thank Craig Boyd for a couple of blog posts (http://tinyurl.com/29yk4fk and http://tinyurl.com/2dvs58g) that got me started and did most of the hard work in figuring out this stuff works. I don't know about you, but most of the time when I search the Internet for how to do something cool in VFP, I end up at Craig's blog because he's done it first and shared it with our community.

Our first ActiveX control

Start VS as an administrator; if you don't, you'll get an error later when VS tries to register the ActiveX control we'll build as a COM object.

Create a new project and choose the "Windows Form Control Library" template from the C# templates. Let's call the project "FirstActiveX." See **Figure 1**.

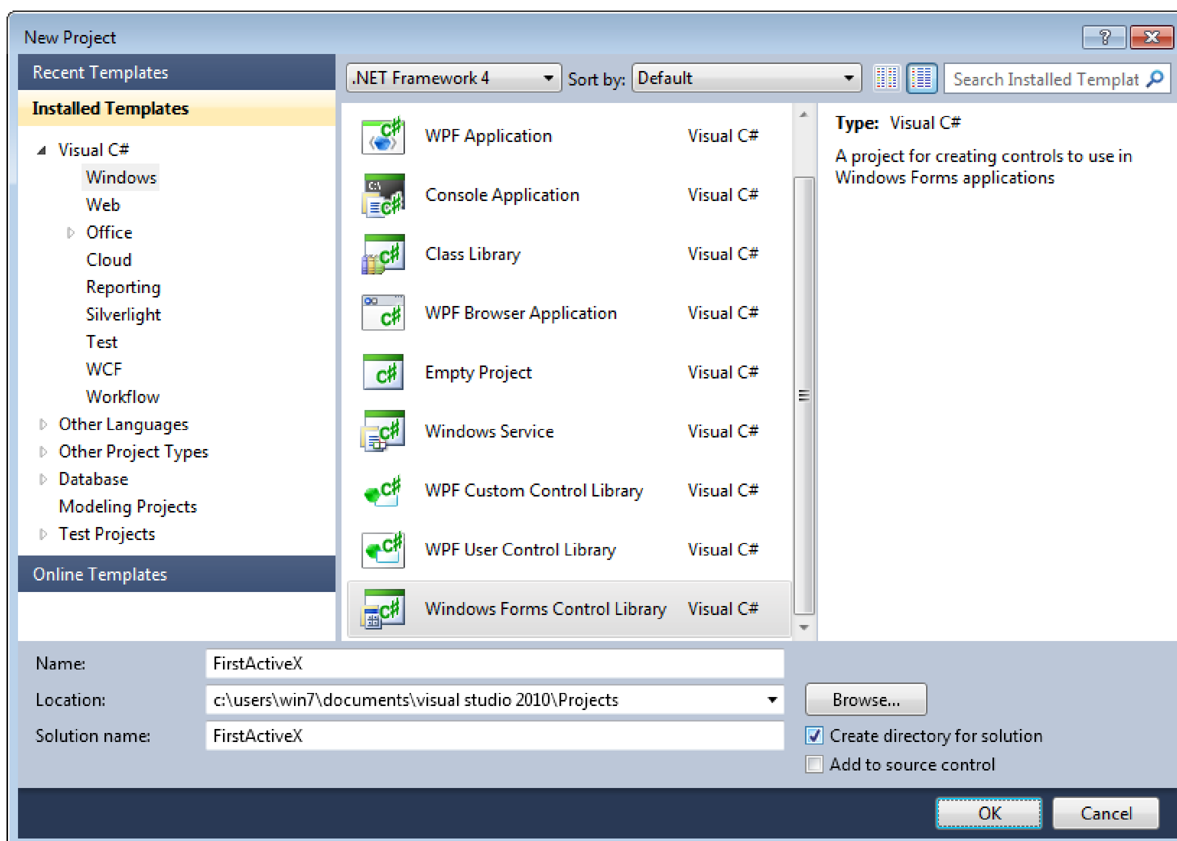

Figure 1. Select the Windows Form Controls Library when creating our first project.

VS creates a file named UserControl1.cs and opens a visual designer for it. Although you could rename it to something more appropriate, we'll leave the name alone for this project. Select the Toolbox and drag a ToolStrip from the "Menus & Toolbars" section to the user control designer. Size the ToolStrip as desired and size the user control so ToolStrip just fills it.

Although we could add our own buttons and other controls to the ToolStrip, let's just populate it with "standard" buttons. Select the ToolStrip and click the little arrow in the upper right corner. In the ToolStrip Tasks dialog, click the Insert Standard Items link. See **Figure 2** to see what the ToolStrip looks like in the designer.

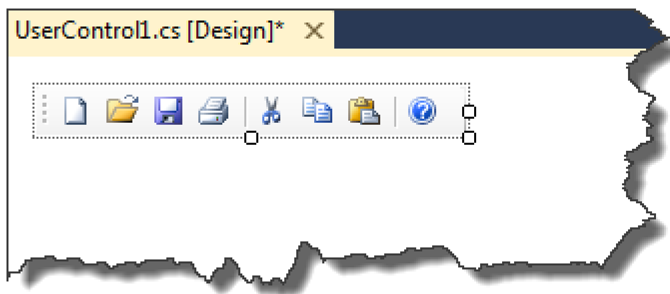

Figure 2. The ToolStrip control as it appears in the designer.

To try out your toolbar, choose Start Debugging from the Debug menu or press F5. The toolbar appears in a test container (**Figure 3**). Close the test container window.

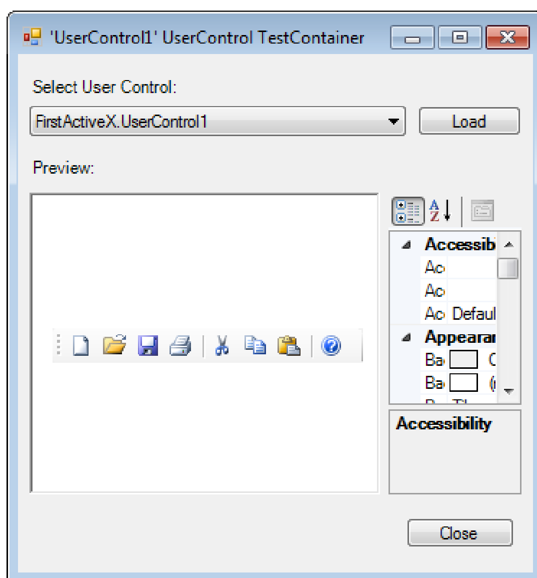

Figure 3. Testing the ToolStrip.

This is fine, but it's still a .Net control, not an ActiveX control. Now the fun starts.

Select UserControl1.cs in Solution Explorer, right-click, and choose View Code. Add the following to the list of using statements at the top of the file:

```
using System.Runtime.InteropServices;
using System.Reflection;
using Microsoft.Win32;
```

Add the following lines just before the class statement:

```
[Guid("A2FC55A1-D5EB-413C-8F95-729AF0E88ACB")]
[ProgId("FirstActiveX.UserControl1")]
[ClassInterface(ClassInterfaceType.AutoDual)]
```

The square brackets around these statements indicate that they're attributes rather than code. The first attribute specifies the GUID for the control. This GUID is from my sample project. You'll need to create your own: from the Tools menu, select Create GUID, choose 5 for the GUID format, click Copy to copy the GUID to the clipboard, and then click Exit. Select the existing GUID statement and paste to replace it with the new one.

The second statement specifies the COM ProgID for the ActiveX control. As you likely know, a COM object is known by its ProgID, typically in the format MyCOMServer.MyClass. This ProgID is used for what's known as "late binding." In VFP, early binding means dropping the control on a form or container, in which case it knows the COM object by GUID, and late binding means you instantiate it using CREATEOBJECT('MyCOMServer.MyClass'). VS automatically assigns a ProgID of *Namespace.Class*, where *Namespace* is the namespace the class is in and *Class* is the class name. The ProgId attribute allows you to change the ProgID from that default to something else. However, I don't think that's a good idea because it causes confusion: the early binding name, the one you see in the VFP Insert Object dialog which appears when you drop an OLE object on a container, is always *Namespace.Class*, so if you specify a different ProgId, you have to use a different name for late binding. Although I included it here as a discussion point, we won't be using it again for other projects.

The third statement specifies that VS should generate a COM interface for the class, and that it should be available for both early and late binding (the "AutoDual" enumeration of ClassInterfaceType). Although the .Net documentation recommends against using AutoDual, it's needed for the ActiveX control to work in VFP. However, we'll see an alternative way of generating the interface later.

Find the closing curly brace ending the class definition (the second last one in the file; the last one closes the namespace) and place the following code before it:

```
[ComRegisterFunction()]
public static void RegisterClass(string key)
{
    // Strip off HKEY_CLASSES_ROOT\ from the passed key as I don't need it
    StringBuilder sb = new StringBuilder(key);
    sb.Replace(@"HKEY_CLASSES_ROOT\", "");

    // Open the CLSID\{guid} key for write access
    RegistryKey k = Registry.ClassesRoot.OpenSubKey(sb.ToString(), true);
```

```
    // And create the 'Control' key - this allows it to show up in
    // the ActiveX control container
    RegistryKey ctrl = k.CreateSubKey("Control");
    ctrl.Close();

    // Next create the CodeBase entry - needed if not string named and GACced.
    RegistryKey inprocServer32 = k.OpenSubKey("InprocServer32", true);
    inprocServer32.SetValue("CodeBase", Assembly.GetExecutingAssembly().CodeBase);
    inprocServer32.Close();

    // Finally close the main key
    k.Close();
}

[ComUnregisterFunction()]
public static void UnregisterClass(string key)
{
    StringBuilder sb = new StringBuilder(key);
    sb.Replace(@"HKEY_CLASSES_ROOT\", "");

    // Open HKCR\CLSID\{guid} for write access
    RegistryKey k = Registry.ClassesRoot.OpenSubKey(sb.ToString(), true);

    // Delete the 'Control' key, but don't throw an exception if it does not exist
    k.DeleteSubKey("Control", false);

    // Next open up InprocServer32
    RegistryKey inprocServer32 = k.OpenSubKey("InprocServer32", true);

    // And delete the CodeBase key, again not throwing if missing
    k.DeleteSubKey("CodeBase", false);

    // Finally close the main key
    k.Close();
}
```

This code was published by Morgan Skinner in an article titled “Exposing Windows Form Controls as ActiveX controls” (http://tinyurl.com/24yk3xd). The two methods are automatically called when the ActiveX control is registered or unregistered as a COM object. They ensure the proper Registry entries are created so the control works as an ActiveX control.

We need to tell VS that this project should be a COM object so we can use it as an ActiveX control. Select the project in the Solution Explorer, right-click, and choose Properties. In the Application page, click Assembly Information and in the Assembly Information dialog, turn on “Make assembly COM-Visible” (**Figure 4**).You can also set some of the other properties if you wish; these are similar to the build properties of a VFP EXE or DLL.

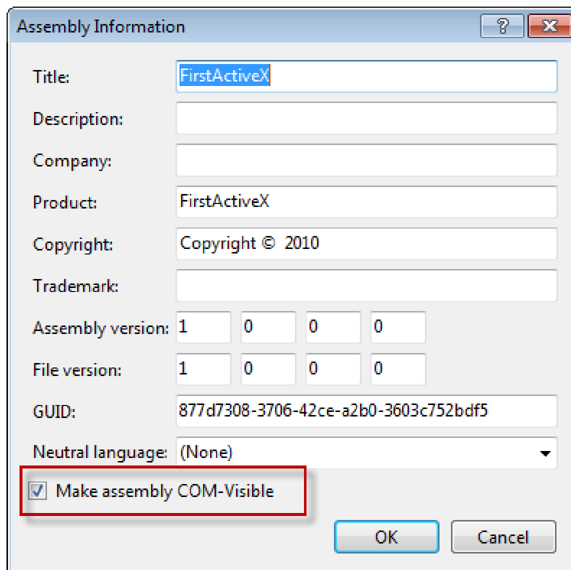

Figure 4. Making our project into a COM control.

One other change you may have to make on the Application page is to set the Target Framework to the version installed on the systems the ActiveX control is to be deployed to. For example, if you're using VS 2010, the default target framework is 4.0. However, even someone using Windows 7 doesn't likely have that version installed by default, so unless you want to install it, change Target Framework to something else. In the Deploying section later in this document, I'll discuss how to ensure the .Net framework is installed on the user's system.

In the Build page, turn on "Register for COM interop" (**Figure 5**); this is really only needed so that when you build the project, VS automatically registers the project for COM on your system, saving you from doing it manually. We'll see later what you have to do on your user's system to register it. Close the Properties pane.

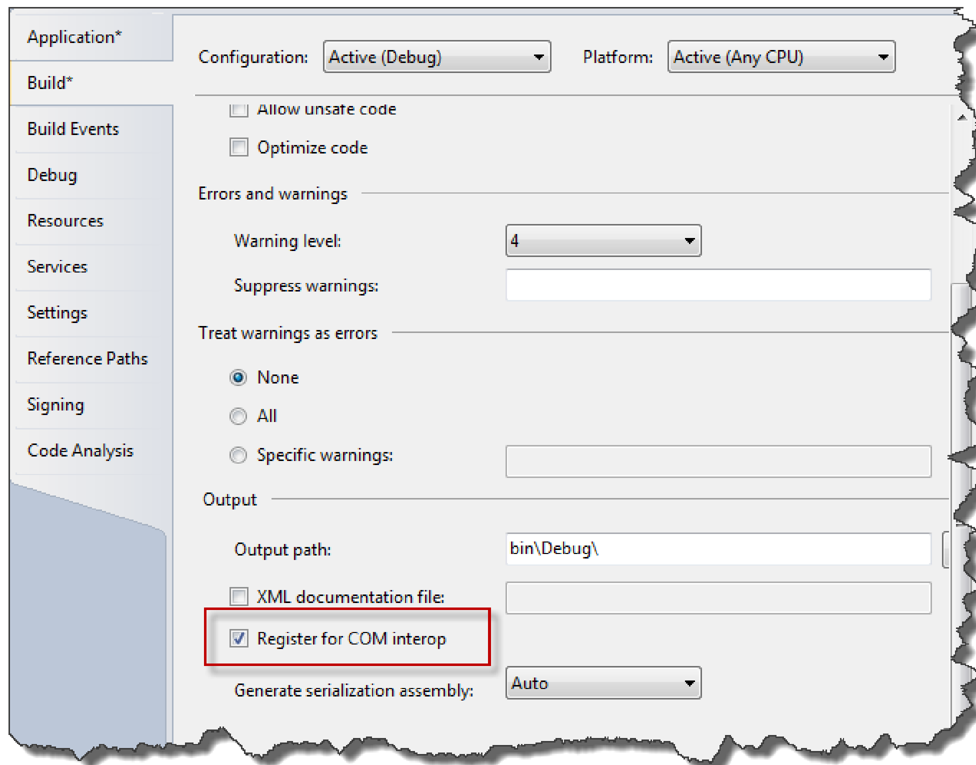

Figure 5. Turn on "Register for COM interop" to automatically register the control on your system.

Let's try it out. Choose Build FirstActiveX from the Build menu or press Shift-F6. You'll get a whole bunch of warnings, like "Type library exporter warning processing 'FirstActiveX.UserControl1.DoDragDrop(#0), FirstActiveX'. Warning: Non COM visible value type 'System.Windows.Forms.DragDropEffects' is being referenced either from the type currently being exported or from one of its base types." You can ignore these.

Start VFP, create a form, add an OLE control to it, and notice FirstActiveX.UserControl1, our ActiveX control, is in the list. Woohoo, it worked! Select the control and click OK. Set the Anchor property to 10 so it resizes horizontally when the form resizes. Finally, run the form. As you move the mouse over each button, you should see a tooltip. If you size the form so the control is smaller than it needs to be to display all buttons, it displays a drop-down button at the right end; click that button to display the rest of the buttons. See **Figure 6**.

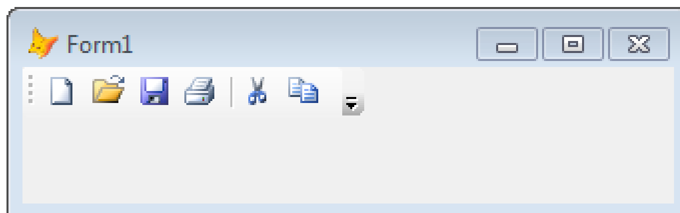

Figure 6. A VFP form hosting our first ActiveX control.

Using Windows Explorer, navigate to the folder where your project is located and in the Bin\Debug folder, take a look at the size of the DLL. On my system, it's only 17K. That's incredibly tiny considering that we now have an attractive toolbar available for our VFP apps. Of course, the reason it's so small is that most of the code is in the .Net framework.

Accessing components in the ActiveX control

Open the form in the Form Designer and try to select one of the buttons in the ActiveX control. You can't. Maybe we have to drill down into the control first. Right-click; there's no Edit button in the shortcut menu. Look in the Properties window; notice there's nothing under the OLE control. In fact, notice that we don't even see the ToolStrip as a component; we just have the UserControl itself. Obviously we don't have access to the components inside the control. However, that's just because we didn't specify that we wanted to expose them. Each object in our control has a Modifier property you can see in the VS Properties window that's like the visibility of a member in VFP. By default, Modifier is Private, so the components are only accessible to the class. Changing it to Public makes them accessible to anything.

Close VFP; we need to do that when we make changes and rebuild our .Net project because VFP holds a reference to the COM object which will prevent the build from succeeding.

Change Modifier for the ToolStrip and the New button to Public, rebuild the project, run VFP, and open the form you added the Toolbar to. You still can't drill down from the ActiveX control to see the ToolStrip and New button. That's because Windows Forms in .Net doesn't have the concept of visual containership like VFP does. Instead, references to those components are stored as properties of the control. These properties don't show up in the Properties window, but you can see them in IntelliSense: when you type "This." in a method of the control, you'll see both toolStrip1 and newToolStripButton properties.

Unfortunately, having access to these components doesn't do you much good. First, you don't get IntelliSense on the components, so typing "This.toolStrip1." doesn't display anything. More seriously, however, trying to access many of the properties of the components causes OLE errors. For example, this works:

```
This.toolStrip1.Enabled = .F.
```

but this:

```
This.newToolStripButton.Enabled = .T.
```

gives "OLE error code 0x80131509: Unknown COM status code", which is an InvalidOperationException in .Net. Most of the errors seem to occur when you access properties that contain .Net objects, such as Font, which is an instance of the System.Drawing.Font object. However, as you can see from the second command, sometimes even accessing simple properties causes problems too.

The solution is to create methods of the ActiveX control that access or set properties of the components. This is better from an OOP perspective anyway because we shouldn't know the inner details, such as the names of the components, of the ActiveX control.

Handling events in the ActiveX control

Because we didn't define any behavior for the buttons, clicking them does nothing. Let's add some code to the Click event of each button to handle that. Oops, as we saw in the previous section, we don't have access to the individual buttons or even the ToolStrip. Perhaps there's a Click event for the UserControl. You can see a lot of properties and methods in the Properties window, but the only events are those for the VFP wrapper of the ActiveX control, like Init, Destroy, and so on. The events for the .Net control aren't exposed in our ActiveX control because events in COM are exposed through interfaces, and those interfaces aren't created when we build our control.

Coding events in .Net

Close VFP. Double-click the New button in the ToolStrip to create a Click event handler. You'll find this code was added to UserControl1.cs:

```
private void newToolStripButton_Click(object sender, EventArgs e)
{

}
```

Sidebar: Handling events

Events in .Net aren't treated exactly like events in VFP. In VFP, when you open a code window for the Click event of a button, we think we're putting code into the Click event. However, that's not really the case. Windows raises an event when you click the button, and VFP has an event handler called Click that's called when the event is raised. VFP hides the internals from us so we only see the event handler, not the wiring between the event and its handler.

.Net makes this process more visible: you wire up the event to a handler that you specify. By default, VS names the handler *ObjectName_EventName* but you can specify a different name if you wish. The Click event is specifically wired to its handler through generated code like this in the "Component Designer generated code" section of UserControl1.Designer.cs:

```
this.newToolStripButton.Click += new System.EventHandler(this.newToolStripButton_Click);
```

This code adds the newToolStripButton_Click method to the list of handlers for the Click event for the newToolStripButton object. (In case you're wondering, yes, an event can have more than one handler, something that we need to use BINDEVENT() for in VFP to accomplish.)

This distinction will be more important when we define our own events rather than letting VS generate the code for us. For more information on how events in .Net work from a VFP perspective, see Rick Strahl's article "Handling .NET Events in Visual FoxPro via COM Interop" at http://tinyurl.com/2crxznl.

Change the scope of the event handler method from "private" to "public" and add the following code inside the curly braces to display a message box when you click the button.

```
MessageBox.Show("You clicked New");
```

Rebuild the project, start VFP, run the form, and click the New button. As expected, the message box appears. That's fine, but we probably want to add code that fires in VFP rather than in .Net. Open the form, double-click the ActiveX control to open a code window, find the newToolStripButton_Click method, and add a MESSAGEBOX() statement like:

```
*** ActiveX Control Method ***
LPARAMETERS sender, e
messagebox('Fired VFP code')
```

Run the form and click the New button. Hmm, we still see the message box from the .Net code. Close VFP, remove the statement in newToolStripButton_Click, rebuild, start VFP, run the form, and click New. Now nothing appears even though there's VFP code in the method. The problem is that .Net is handling the event itself and not raising it as an event VFP can handle.

Sidebar: Raising events

There are three components to raising an event in .Net. First, you have to define a delegate. A delegate is a pointer to a function. It specifies what the signature of the function is: what parameters it accepts and what data type it returns. Here's an example of a delegate definition:

```
public delegate void MyEventHandler();
```

The second thing you need is an event. In .Net, you can define your own events. We can sort of do the same thing in VFP using the RAISEEVENT() function but it's a little more sophisticated in .Net. An event is a special type of delegate, so you specify what delegate it's based on. The following creates an event that uses the delegate we just defined:

```
event MyEventHandler MyEvent;
```

Finally, to raise the event, you call it like you would any method.

Defining a delegate and an event

Let's define a delegate for button click events. Although we could define one for each button, there's no need to do that; we'll just create one for all of them because they'll all

have the same signature. Add the following to UserControl1.cs just after the COM registration code and before the curly brace ending the class (the second last one):

```
public delegate void ButtonClickEventHandler();
```

Let's define an event we'll raise when you click the New button. The event uses the delegate we just created. Add this just below the delegate statement:

```
event ButtonClickEventHandler NewButtonClick;
```

Finally, let's raise the NewButtonClick event from the Click event handler for the new Button: add this new statement in newToolStripButton_Click:

```
public void newToolStripButton_Click(object sender, EventArgs e)
{
    if (NewButtonClick != null)
        NewButtonClick();
}
```

newToolStripButton_Click is already wired up to the Click event, so **Figure 7** shows the sequence of things that happen when you click the New button.

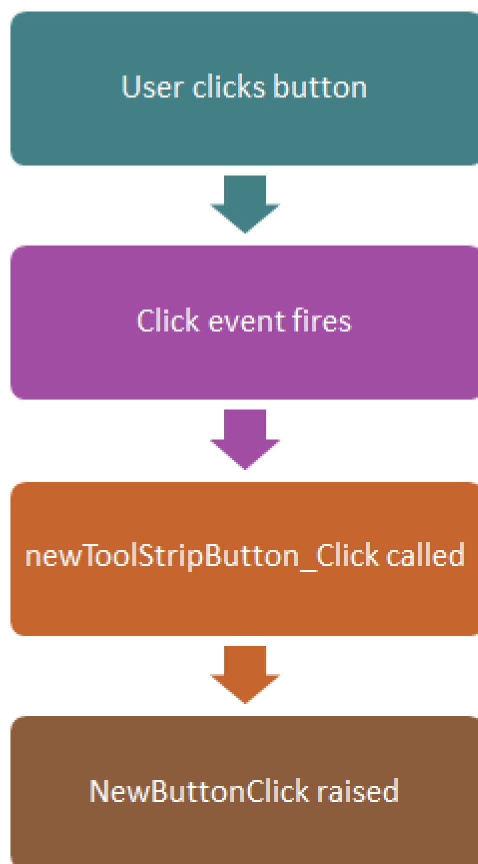

Figure 7. The flow of events when the user clicks the New button.

Adding the event to the COM interface

Now that we have an event, we have to add it to the COM interface for the ActiveX control. We'll define an interface called ControlEvents that specifies our event and add COM attributes to the interface so it's visible in VFP.

Add the following between the namespace and class definition in UserControl1.cs:

```
[Guid("5760720E-8F36-4BBD-8B0D-88582BC91A7F")]
[InterfaceType(ComInterfaceType.InterfaceIsIDispatch)]
public interface ControlEvents
{
    [DispId(1)]
    void NewButtonClick();
}
```

Remember to generate your own GUID. Add a ComSourceInterfaces attribute to the class, specifying the new interface:

```
[ComSourceInterfaces(typeof(ControlEvents))]
```

Build the project, start VFP, open the form, double-click the ActiveX control, and notice we now have a NewButtonClick event. Add the following code:

```
*** ActiveX Control Event ***
messagebox('Fired from VFP')
```

Run the form and click the New button. Woohoo #2, it works.

You can now go back and create Click event handlers for each of the buttons, create events for each of those, and wire up the event handler to the event just as we did with the New button.

Creating a base ActiveX class

What we've built so far works great but has a couple of issues:

- If you create several ActiveX controls, you have to copy and paste the RegisterClass and UnregisterClass methods in each one.
- The interface exposed by the ActiveX control includes a lot of members we don't care about, as you can see in **Figure 8**.

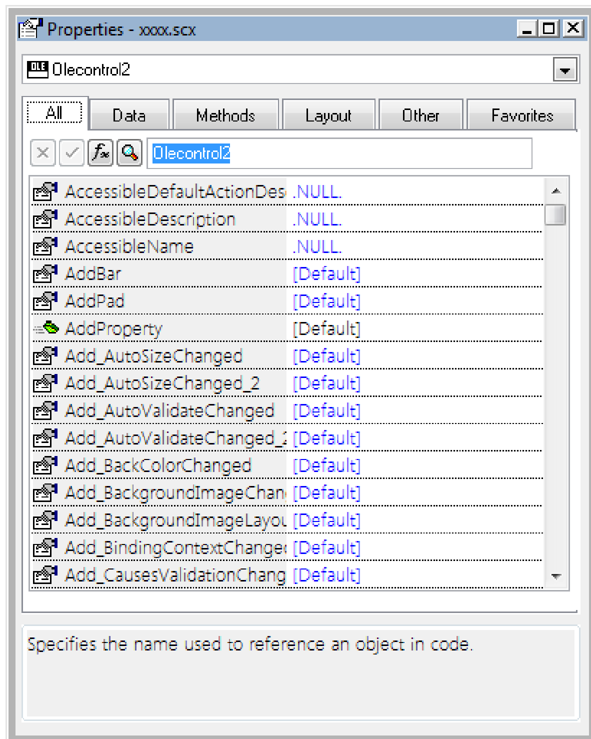

Figure 8. There are more members exposed for the ActiveX control than we want.

We'll fix the first problem by creating a base class we'll use for all of our ActiveX controls. Create a new project using the "UserControl" template called DotNetControls. Rename UserControl1.cs to COMUserControl.cs. Right-click COMUserControl.cs and choose View Code. Replace the code in the class with the following:

```
using System.Text;
using System.Windows.Forms;
using System.Reflection;
using Microsoft.Win32;
using System.Runtime.InteropServices;

namespace DotNetControls
{
    [ComVisible(false)]
    public partial class COMUserControl : UserControl
    {
        public COMUserControl()
        {
            InitializeComponent();
        }

        [ComRegisterFunction()]
        public static void RegisterClass(string key)
        {
            // Strip off HKEY_CLASSES_ROOT\ from the passed key as I don't need it
            StringBuilder sb = new StringBuilder(key);
            sb.Replace(@"HKEY_CLASSES_ROOT\", "");
```

```
            // Open the CLSID\{guid} key for write access
            RegistryKey k = Registry.ClassesRoot.OpenSubKey(sb.ToString(), true);

            // And create the 'Control' key - this allows it to show up in
            // the ActiveX control container
            RegistryKey ctrl = k.CreateSubKey("Control");
            ctrl.Close();

            // Next create the CodeBase entry - needed if not string named and GACced.
            RegistryKey inprocServer32 = k.OpenSubKey("InprocServer32", true);
            inprocServer32.SetValue("CodeBase",
                Assembly.GetExecutingAssembly().CodeBase);
            inprocServer32.Close();

            // Finally close the main key
            k.Close();
        }

        [ComUnregisterFunction()]
        public static void UnregisterClass(string key)
        {
            StringBuilder sb = new StringBuilder(key);
            sb.Replace(@"HKEY_CLASSES_ROOT\", "");

            // Open HKCR\CLSID\{guid} for write access
            RegistryKey k = Registry.ClassesRoot.OpenSubKey(sb.ToString(), true);

            // Delete the 'Control' key, but don't throw an exception if it does not exist
            k.DeleteSubKey("Control", false);

            // Next open up InprocServer32
            RegistryKey inprocServer32 = k.OpenSubKey("InprocServer32", true);

            // And delete the CodeBase key, again not throwing if missing
            k.DeleteSubKey("CodeBase", false);

            // Finally close the main key
            k.Close();
        }
    }
}
```

This code declares the COMUserControl class as non-visible to COM; we won't be using it directly so we don't want it showing up in COM. It also has the RegisterClass and UnregisterClass methods so we don't have to add those methods to our classes anymore.

To use this base class, our ActiveX controls will inherit from COMUserControl rather than UserControl; that way, we automatically get the COM register/unregister functionality.

The fix for the second problem is changing the COM interface generated by VS so we only see the members we want. We'll change the ClassInterface attribute for our classes from AutoDual to None. That tells VS to not generate a COM interface from the UserControl class, so we won't see any members we don't want. However, if we then build the project and try to use the ActiveX control in VFP, we'll get an "interface not supported" error. So, we need

to add one other thing to our ActiveX control class: an interface that specifies the members we want exposed through COM. We'll then have our class inherit from that interface.

Sidebar: Multiple Inheritance and Interfaces

VFP doesn't really have the concept of an interface. An interface is the definition (but not implementation) of the members of a class. It lists the properties and their data types and the methods and their signatures without providing any behavioral code. In essence, it's a contract that guarantees anything inheriting from the interface has the specified properties and methods. The closest we can come to an interface in VFP is to define an abstract class with properties and methods and no code, and then subclass from it to implement behavior.

One interesting thing about interfaces is that a class can inherit from more than one. Like VFP, .Net doesn't support multiple inheritance; a class can derive from only one parent class. However, .Net does allow a class in implement multiple interfaces. For example, support one interface specifies an Execute method and another has an IsValid property. A class can inherit from both interfaces as long as it has an Execute method with the correct signature and an IsValid property of the correct data type.

Interfaces have several uses including allowing you to treat different objects that implement the same interface as if they were the same. However, we're going to make use of only one aspect of an interface: telling COM what members our controls expose.

Let's look at an example.

Creating a generic toolbar

The toolbar we built for our first ActiveX control works great if you want a static ToolStrip with pre-defined buttons. However, what we really want is a generic toolbar we can add buttons to in VFP. Follow these steps to create the control:

- Build the project so the COMUserControl is available in the next step.
- Right-click the DotNetControls project, choose Add, New Item. Enter "Toolbar.cs" as the name. From the Windows Form category, double-click "Inherited User Control." Choose "COMUserControl" to inherit from, and click OK.
- Add a ToolStrip control to the UserControl and change its name to "toolStrip." Size the user control as desired.
- Right-click Toolbar.cs in the Solution Explorer and choose View Code.
- Add the following to the using section:

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

- Add the following lines above the class definition, and replace the two GUIDs with new ones:

```
[Guid("11DFA2BB-4E7B-43CB-8201-2E74983DE6D4")]
[InterfaceType(ComInterfaceType.InterfaceIsIDispatch)]
public interface ToolbarEvents
{
    [DispId(1)]
    void ButtonClick();
}

[Guid("A2FC55A1-D5EB-413C-8F95-729AF0E88ACB")]
[ClassInterface(ClassInterfaceType.None)]
[ComSourceInterfaces(typeof(ToolbarEvents))]
```

- Add the button click event handling:

```
public delegate void ButtonClickEventHandler();
event ButtonClickEventHandler ButtonClick;
private void ToolStripButton_Click(object sender, EventArgs e)
{
    if (ButtonClick != null)
    {
        ButtonClick();
    }
}
```

- Add a method to allow programmatically adding buttons. It accepts the name of the button, the tooltip text for it, and the name of the image file to use. It also registers ToolStripButton_Click as the event handler for the button.

```
public ToolStripButton AddButton(string name, string text, string imageFile)
{
    ToolStripButton newButton = new ToolStripButton();
    this.toolStrip.Items.Add(newButton);
    newButton.DisplayStyle = ToolStripItemDisplayStyle.Image;
    newButton.Image = System.Drawing.Image.FromFile(imageFile);
    newButton.ImageTransparentColor = System.Drawing.Color.White;
    newButton.Name = name;
    newButton.AutoSize = true;
    newButton.Text = text;
    newButton.Click += new System.EventHandler(this.ToolStripButton_Click);
    return newButton;
}
```

- As I mentioned in the previous section, we need to define an interface that specifies the COM interface we want exposed and have our class inherit from that interface. Add the following code after the class definition (before the final curly brace which ends the namespace):

```
public interface ToolbarInterface
{
    ToolStripButton AddButton(string name, string text, string imageFile);
}
```

This interface defines the signature for the AddButton method. Note that we now have two interfaces defined: one for the events we'll raise that's only there so those events are exported to COM, and one for the members of the class.

- To specify that we want our class to inherit from this interface, change the class definition to:

```
public partial class Toolbar : COMUserControl, ToolbarInterface
```

- Turn on the two COM registration settings in the Properties dialog for the project discussed earlier.
- Build the project.
- Start VFP, create a new form, add DotNetControls.Toolbar, and name it oToolbar. Set Anchor to 10.
- Add the following code to the Init method of the form to programmatically create buttons in the toolbar (This code assumes the specified images exist in the current folder. They're included with the sample files accompanying this document.)

```
with This.oToolbar
   .AddButton('OpenButton', 'Open', 'Open.bmp')
   .AddButton('NewButton',  'New',  'New.bmp')
   .AddButton('SaveButton', 'Save', 'Save.bmp')
endwith
```

- Add this code to oToolbar.ButtonClick:

```
messagebox('Button clicked')
```

- Run the form and click one of the buttons. You should see the message box appear. See **Figure 9**.

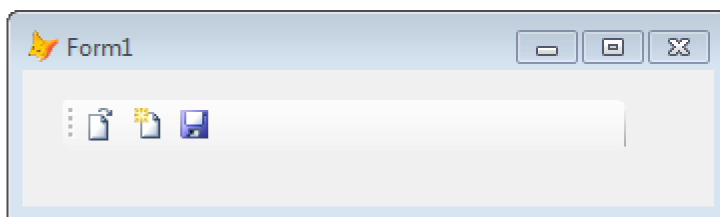

Figure 9. A VFP form hosting our generic toolbar ActiveX control.

The only thing missing now is that we can't tell which button the user clicked. We're going to have to pass a parameter to ButtonClick. Close VFP and switch back to VS.

- Change the signature for the delegate to accept the name of the button:

```
public delegate void ButtonClickEventHandler(string buttonName);
```

- Similarly, change the ButtonClick method signature in the ToolbarControlEvents interface:

```
void ButtonClick(string buttonName);
```

- Change the code in ToolStripButton_Click to pass the name of the clicked button to the ButtonClick event:

```
private void ToolStripButton_Click(object sender, EventArgs e)
{
    if (ButtonClick != null)
    {
        ToolStripButton button = (ToolStripButton)sender;
        ButtonClick(button.Name);
    }
}
```

- Build the project.
- Start VFP, remove the code in oToolbar.ButtonClick, then close the method and open it again. You'll now see a LPARAMETERS buttonname statement. Add this code:

```
messagebox(buttonname)
```

- Run the form and click the button. The message box now shows the name of the button. In a real form, you'd use a CASE statement to decide what to do based on which button was clicked.

As you can see in **Figure 10**, thanks to the interfaces we defined, we now only see the members we're interested in: AddToolStripButton (thanks to ToolbarInterface) and ButtonClick (thanks to ToolbarControlEvents).

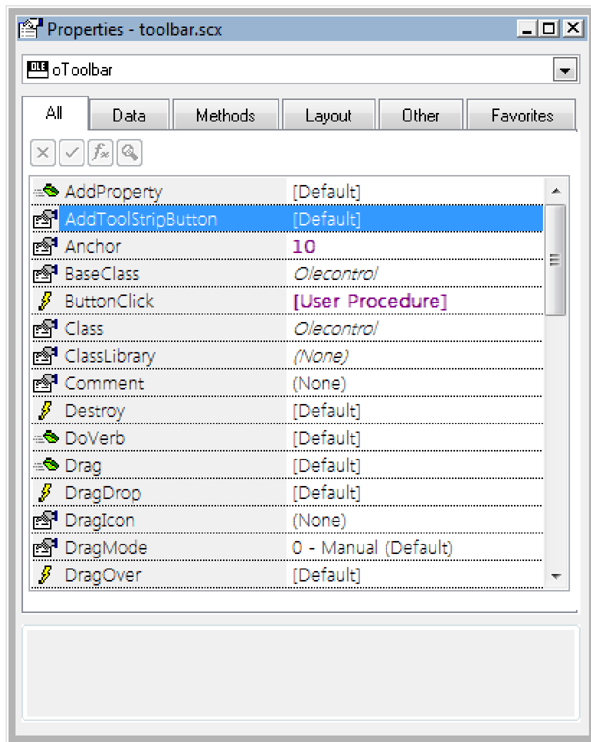

Figure 10. By specifying the interface for our class, we only expose the members we want.

The Toolbar class still needs some work. For example, you need a way to modify a button dynamically, such as changing the tooltip, the image, or enabled status. Also, other types of controls can be used in ToolStrips, such as labels, comboboxes, and separators. However, this is a good start towards a complete .Net ActiveX control we can use in VFP.

Sidebar: Adding descriptions to properties and methods

To make your ActiveX class easier to use, you might want to add descriptions to the properties and methods. VFP IntelliSense displays your descriptions as tooltips in the list of properties and methods that appear when you type the object name and a period.

Add descriptions using the [Description] attribute. You'll need to add a "using System.ComponentModel" line in your code to use the attribute. Here's an example of how this attribute is used:

```
[Description("This is my method")]
```

Note a few things about how this works. First, descriptions on the properties and methods of a class are ignored; only those on the interface for the class are used. Second, the description for a property is ignored; you have to put the description on the "getter" of the property. Here's an example of an interface that has descriptions for its members:

```
public interface MyInterface
{
```

```
    string Name
    {
        [Description("The name of the item")]
        get;
        set;
    }
    [Description("Some method")]
    void SomeMethod();
}
```

Add more classes

Let's add a menubar class to the project. Because we're not creating a new project, this will give us one DLL containing several controls.

- Close VFP.
- Right-click the DotNetControls project, choose Add, New Item. Enter "MenuBar.cs" as the name. From the Windows Form category, double-click "Inherited User Control." Choose "COMUserControl" to inherit from, and click OK.
- From the Toolbox, drag a MenuStrip to the UserControl and rename it to menuStrip.
- Open MenuBar.cs and add this to the using section:

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

- Add the following lines above the class definition, and replace the two GUIDs with new ones:

```
[Guid("13278C09-7E50-4C1F-ACAA-2AD7BEB3C17D")]
[InterfaceType(ComInterfaceType.InterfaceIsIDispatch)]
public interface MenuBarEvents
{
    [DispId(1)]
    void ItemClick(string ItemName);
}

[Guid("EF4B01BE-578F-493C-88EE-FE1CA165D2B6")]
[ClassInterface(ClassInterfaceType.None)]
[ComSourceInterfaces(typeof(MenuBarEvents))]
```

- Add the event handling code:

```
public delegate void ItemClickEventHandler(string itemName);
event ItemClickEventHandler ItemClick;
private void MenuItem_Click(object sender, EventArgs e)
{
    if (ItemClick != null)
    {
        ToolStripMenuItem item = (ToolStripMenuItem)sender;
        ItemClick(item.Name);
    }
}
```

- Add methods to add a pad to the menu and add a menu item to a pad:

```
public ToolStripMenuItem AddPad(string name, string text, string shortcutKey)
{
    ToolStripMenuItem newItem = new ToolStripMenuItem();
    this.menuStrip.Items.Add(newItem);
    newItem.Name = name;
    newItem.AutoSize = true;
    newItem.Text = text;
    if (shortcutKey != "")
    {
        char key = shortcutKey.ToUpper().ToCharArray()[0];
        newItem.ShortcutKeys = ((Keys)((Keys.Alt | (Keys)(byte)key)));
    }
    return newItem;
}

public ToolStripMenuItem AddBar(ToolStripMenuItem menuPad, string name, string
       text, string shortcutKey, string imageFile)
{
    ToolStripMenuItem newBar = new ToolStripMenuItem();
    menuPad.DropDownItems.Add(newBar);
    newBar.Name = name;
    newBar.AutoSize = true;
    newBar.Text = text;
    newBar.Click += new System.EventHandler(this.MenuItem_Click);
    if (imageFile != "")
    {
        newBar.Image = System.Drawing.Image.FromFile(imageFile);
        newBar.ImageTransparentColor = System.Drawing.Color.White;
    }
    if (shortcutKey != "")
    {
        char key = shortcutKey.ToUpper().ToCharArray()[0];
        newBar.ShortcutKeys = ((Keys)((Keys.Control | (Keys)(byte)key)));
    }
    return newBar;
}
```

- Create the COM interface and add it to the class definition for MenuBar:

```
public interface MenuBarInterface
{
    ToolStripMenuItem AddPad(string name, string text, string shortcutKey);
    ToolStripMenuItem AddBar(ToolStripMenuItem menuPad, string name, string text,
       string shortcutKey, string imageFile);
}
```

- Build the project.
- Start VFP, create a new form, add DotNetControls.MenuBar, and name it oMenu.
- Add the following code to the Init method of the form to programmatically fill the menu:

```
with This.oMenu
```

```
   loPad = .AddPad('FilePad', 'File', 'F')
   .AddBar(loPad, 'FileOpenBar', '&Open', 'O', 'Open.bmp')
   .AddBar(loPad, 'FileNewBar',  '&New',  'N', 'New.bmp')

   loPad = .AddPad('EditPad', 'Edit', 'E')
   .AddBar(loPad, 'EditCut',   '&Cut',   'X', 'Cut.bmp')
   .AddBar(loPad, 'EditCopy',  '&Copy',  'C', 'Copy.bmp')
   .AddBar(loPad, 'EditPaste', '&Paste', 'V', 'Paste.bmp')
endwith
```

- Add the following code to oMenu.ItemClick:

```
LPARAMETERS itemname
messagebox(itemname)
```

- Run the form and select a menu item; the message box displays the name of the item you chose (**Figure 11**).

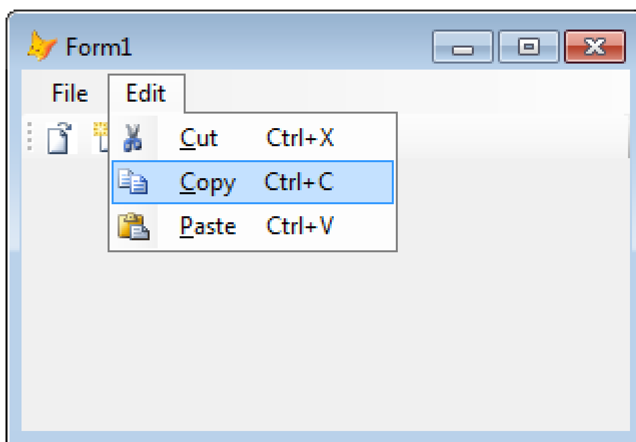

Figure 11. This VFP form hosts both MenuBar and Toolbar ActiveX controls.

Non-visual controls

There's nothing saying the ActiveX control you create has to be a visual one. For example, the .Net framework has several dialog classes that use the latest version of your operating system's common dialogs. Compare the dialogs displayed by the VFP GETFILE() function (**Figure 12**) and the .Net OpenFileDialog (**Figure 13**), for example. The VFP dialog looks old and doesn't support features added in Windows Vista and Windows 7 such as search and libraries. The .Net dialog, on the other hand, looks like a typical dialog you see in other modern applications.

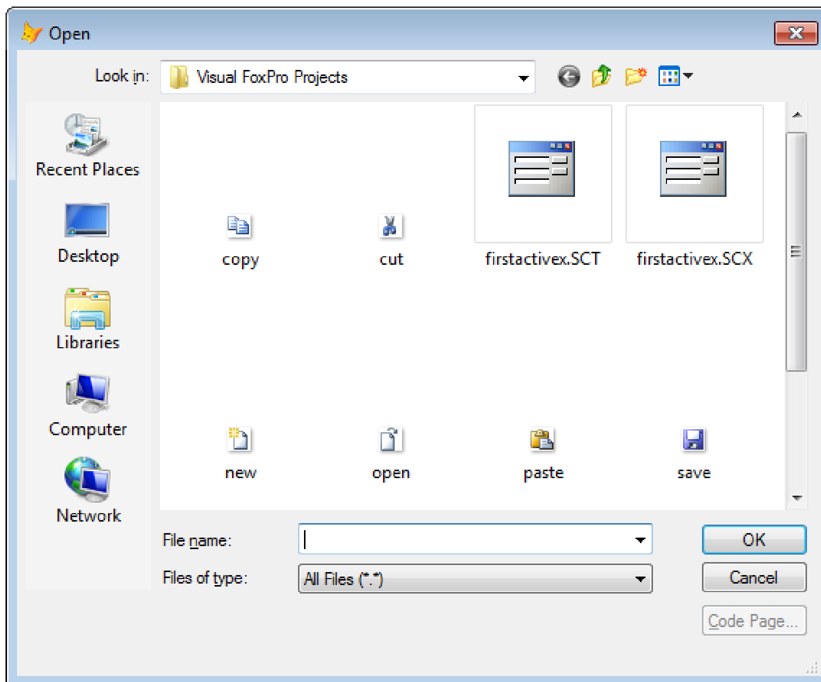

Figure 12. The VFP GETFILE() dialog looks old and doesn't support Windows Vista or 7 features.

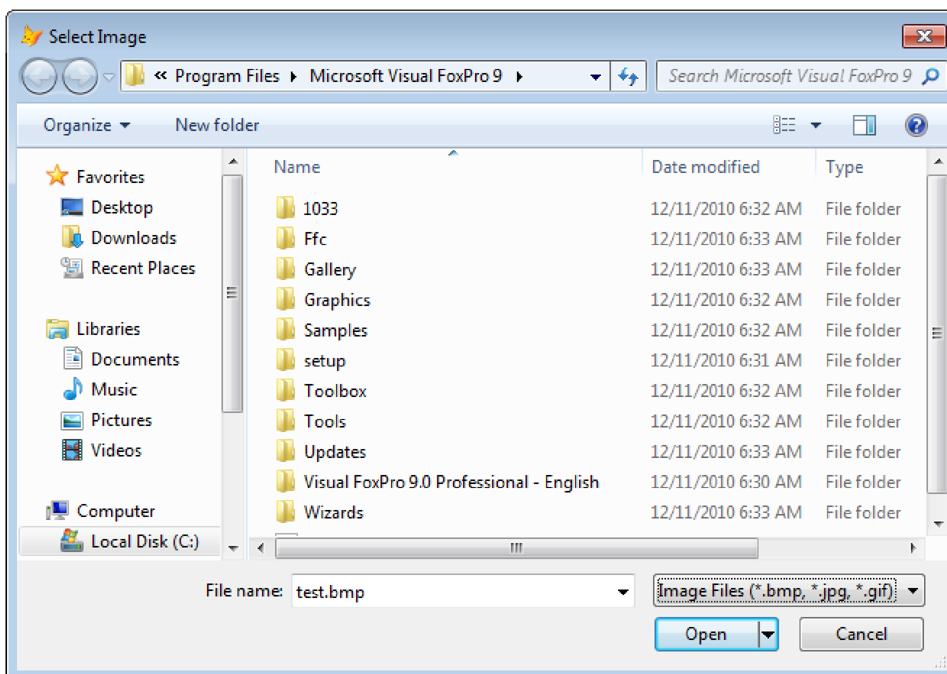

Figure 13. The .Net open file dialog supports all of the new features in Windows Vista and 7.

Let’s create an ActiveX control that exposes several .Net dialog classes. Create a new class using the “Class Library” template; we’ll create an ActiveX control we instantiate with CREATEOBJECT() rather than a visual control we drop on a form. Name the class Dialogs.

Here's the code for the class. Remember to change the GUID.

```
using System.Windows.Forms;
using System.Runtime.InteropServices;

namespace DotNetControls
{
    [Guid("A4E548B7-D31E-43C0-8B25-A77CDE2C383C")]
    [ClassInterface(ClassInterfaceType.None)]
    public class Dialogs : COMUserControl, DialogsInterface
    {
        public string DefaultExt { get; set; }
        public string FileName { get; set; }
        public string InitialDir { get; set; }
        public string Title { get; set; }
        public string Filter { get; set; }
        public int FilterIndex { get; set; }

        public string ShowOpenDialog()
        {
            string fileName;
            OpenFileDialog dialog = new OpenFileDialog();
            dialog.FileName = FileName;
            dialog.DefaultExt = DefaultExt;
            dialog.InitialDirectory = InitialDir;
            dialog.Title = Title;
            dialog.Filter = Filter;
            dialog.FilterIndex = FilterIndex;

            if (dialog.ShowDialog() == DialogResult.OK)
                fileName = dialog.FileName;
            else
                fileName = "";
            return fileName;
        }

        public string ShowSaveDialog()
        {
            string fileName;
            SaveFileDialog dialog = new SaveFileDialog();
            dialog.FileName = FileName;
            dialog.DefaultExt = DefaultExt;
            dialog.InitialDirectory = InitialDir;
            dialog.Title = Title;
            dialog.Filter = Filter;
            dialog.FilterIndex = FilterIndex;

            if (dialog.ShowDialog() == DialogResult.OK)
                fileName = dialog.FileName;
            else
                fileName = "";
            return fileName;
        }
    }

    public interface DialogsInterface
    {
```

```
        string DefaultExt { get; set; }
        string FileName { get; set; }
        string InitialDir { get; set; }
        string Title { get; set; }
        string Filter { get; set; }
        int FilterIndex { get; set; }

        string ShowOpenDialog();
        string ShowSaveDialog();
    }
}
```

This code creates properties for the default extension of the file, the default filename, the initial directory to display, the title of the dialog, the filter (file types displayed in the "Files of type" combobox), and the index for the default filter. Filter is similar to the cFileExtensions parameter of the VFP GETFILE() function, but is formatted a little differently: specify each file type as the description of the filter, followed by a vertical bar (|) and the filter pattern. You can specify multiple extensions for a given file type by separating them with semicolons. Separate additional file types with a vertical bar. For example, use this to specify image files with three possible extensions or all files:

Image Files (*.bmp, *.jpg, *.gif)|*.bmp;*.jpg;*.gif|All files (*.*)|*.*

There are other properties of the OpenFileDialog and SaveFileDialog classes we could set, such as Multiselect and ShowReadOnly; I'll leave that as an exercise for you. See http://tinyurl.com/2weng6a and http://tinyurl.com/36n3fsp for documentation on the properties of these classes.

Build the project, then start VFP. Set up IntelliSense for the Dialogs class: select IntelliSense Manager from the Tools menu, select the Types page, click the Type Libraries button, click the checkbox for DotNetControls, click Done, and click OK (**Figure 14**).

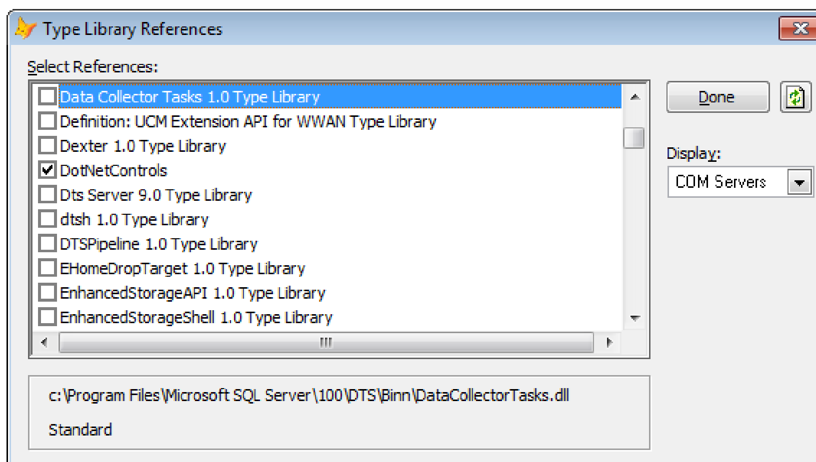

Figure 14. Adding the DotNetControls library to the IntelliSense Manager provides IntelliSense on the Dialogs class.

Create a form, drop a CommandButton on it, and enter the following code in its Click method:

```
local loDialog as DotNetControls.Dialogs, ;
    lcFile
loDialog = createobject('DotNetControls.Dialogs')
loDialog.FileName = 'test.bmp'
loDialog.InitialDir = home()
loDialog.Filter = 'Image Files (*.bmp, *.jpg, *.gif)|*.bmp;*.jpg;' + ;
    '*.gif|All files (*.*)|*.*'
loDialog.Title = 'Select Image'
lcFile = loDialog.ShowOpenDialog()
if not empty(lcFile)
    messagebox(lcFile)
endif not empty(lcFile)
```

Add another CommandButton with similar code but call ShowSaveDialog() instead. Run the form and click each button to see how the dialogs look. They're a lot nicer than GETFILE() and PUTFILE() and you have more control over the dialogs as well.

Subclassing controls directly

So far, we've been using a UserControl container to host .Net controls. What about subclassing a control directly? That's possible too. The benefit of this approach is that you don't have to expose properties of the control; that's already taken care of. You do have to expose events, however.

The following class named MyButton is a subclass of Button:

```
using System;
using System.Windows.Forms;
using System.Runtime.InteropServices;

namespace MyNamespace
{
    [Guid("DD26B13D-F956-4898-8786-AA8D30C86DF0")]
    [InterfaceType(ComInterfaceType.InterfaceIsIDispatch)]
    public interface ButtonControlEvents
    {
        [DispId(1)]
        void ButtonClick();
    }

    [Guid("D32AFCD9-EA53-4AA5-AE75-308D618C450D")]
    [ClassInterface(ClassInterfaceType.AutoDual)]
    [ComSourceInterfaces(typeof(ButtonControlEvents))]
    public class MyButton : Button
    {
        public delegate void ButtonClickEventHandler();
        event ButtonClickEventHandler ButtonClick;
        private void Button_Click(object sender, EventArgs e)
        {
            ButtonClick();
        }
```

```
        public MyButton()
        {
            this.Click += new System.EventHandler(this.Button_Click);
        }

        [ComRegisterFunction()]
        public static void RegisterClass(string key)
        {
            COMRegistration.RegisterClass(key);
        }

        [ComUnregisterFunction()]
        public static void UnregisterClass(string key)
        {
            COMRegistration.UnregisterClass(key);
        }
    }
}
```

Most of this code should be clear to you by now. However, you're likely wondering why we don't just expose the Click event itself directly in the ButtonControlEvents interface but instead have Click handled by Button_Click and have Button_Click raise the ButtonClick event. Unfortunately, that doesn't work; if you try that and put code in the Click event of the ActiveX control in VFP, you'll get an error. The Click event needs an event handler.

Unfortunately, this approach has some limitations, the biggest of which is that, like we saw earlier when accessing components, some properties are difficult to work with. For example, although you can't see Font in the Properties window, it does show up in IntelliSense. However, Font is an object (System.Drawing.Font) with Name and Size properties, and code like this doesn't work (it doesn't cause an error, but the font isn't changed):

```
This.Font.Name = 'Segoe UI'
This.Font.Size = 24
```

The same is true for other properties that are objects, such as Image and Dock (in fact, Dock doesn't even show up in IntelliSense). If you want to expose those properties, you have to create methods or other properties that get or set their values.

Other controls

The DotNetControls project that accompanies this document has several other ActiveX controls.

The .Net DateTimePicker control is a very nice date/time picker with several formats available and a cool dropdown calendar. I've exposed it as DotNetControls.DateTimePicker with Value, Format (1 = long, which displays the day as well as date, 2 = date only, 4 = time, and 8 = custom), CustomFormat (see http://tinyurl.com/34nyy9u for the possible values), FontName, and FontSize properties. See **Figure 15** for an example.

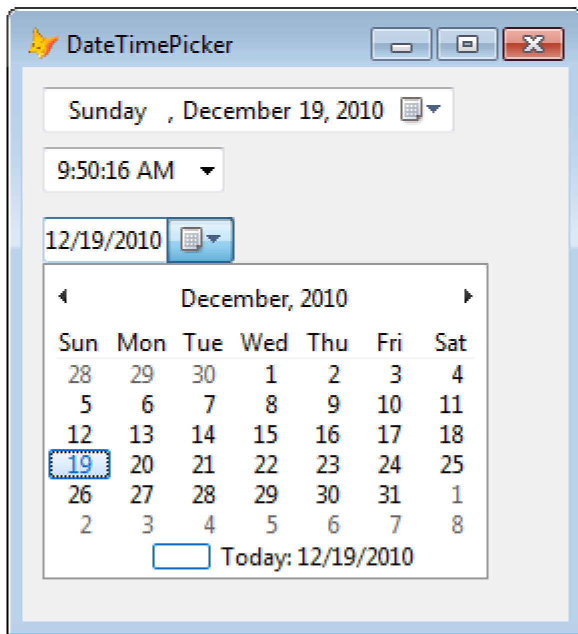

Figure 15. DotNetControls.DateTimePicker is a nice datetime picker control.

DotNetControls.ProgressBar is a progress bar that displays a modern Vista/Windows 7 progress bar rather than the older XP-style bar the ActiveX progress bar control that comes with VFP uses. Set its Value property to the desired value (the range is 0 to 100). Set Marquee to .T. to display a continuously moving bar.

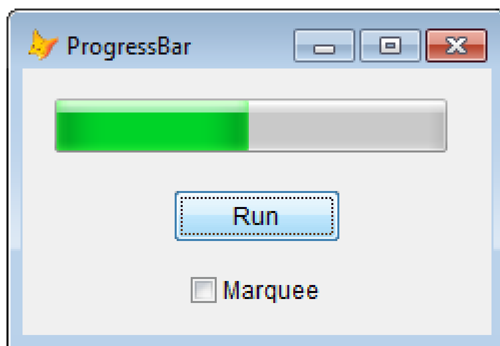

Figure 16. DotNetControls.ProgressBar is a modern-looking progress bar.

DotNetControls.ContextMenu is similar to MenuBar but for shortcut menus (**Figure 17**). Since it doesn't have pads, it doesn't have an AddPad method, just AddBar, to which you pass the name, caption, shortcut key, and image file. To display the menu, call ShowMenu, usually from the RightClick method of an object. Call Clear to remove all menu items. Like MenuBar, the ItemClick method receives the name of the clicked menu item. Because it doesn't have a visual representation like the other controls, ContextMenu displays as a small menu icon at design time. You have to set Visible to .F. so it doesn't display at run time.

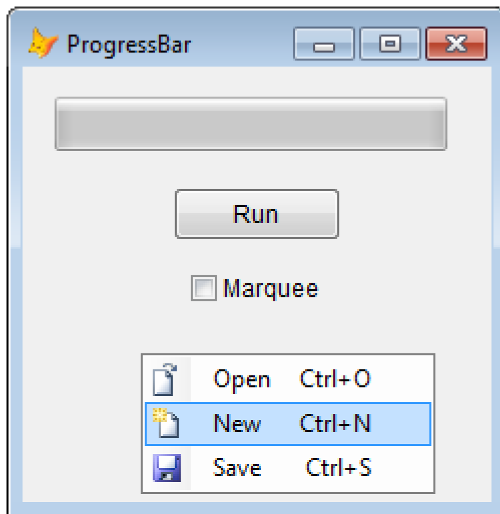

Figure 17. DotNetControls.ContextMenu displays a shortcut menu.

Of course, you don't have to use just native .Net controls. Many companies, such as Telerik and Infragistics, sell Windows Forms controls, and there are thousands of free ones available. For example, a CodePlex project, VistaControls (http://tinyurl.com/29mjbqe), has a TreeView control that uses the Vista/Windows 7 style rather than the older style used by the ActiveX controls that comes with VFP and even the native .Net version. I created DotNetControls.TreeView that uses this control. It has the following members:

- CheckBoxes: set this property to .T. if you want node checkboxes.
- LoadImage(Key, ImageFile): adds the specified image to the contained ImageList control so nodes can have images.
- AddNode(Key, Text, ImageKey): adds a top-level node to the TreeView using the specified key, text, and image key from the ImageList, and returns a reference to the node so you can set additional properties such as Checked or ToolTipText.
- AddChildNode(ParentKey, Key, Text, ImageKey): adds a node as a child of the specified node. I originally created a second AddNode method with a different signature that accepts ParentKey but since COM doesn't allow two methods with the same name, its name appeared as AddNode_2, which isn't very intuitive.
- Clear: removes all nodes.
- NodeClick(Key): this event is raised when the user clicks a node. It's passed the key of the node.

Of course, this TreeView isn't fully functional yet because it doesn't expose all of the members of the TreeView or raise all of the events, but it's a start. **Figure 18** shows a sample form with a DotNetControls.TreeView on it.

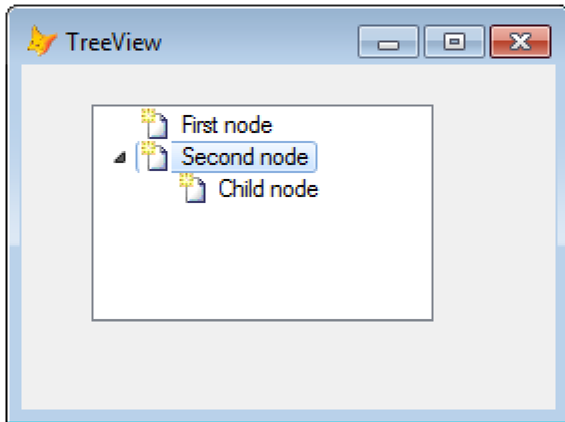

Figure 18. DotNetControls.TreeView displays a Windows Vista/7 style TreeView control.

Interop Forms Toolkit

Everything we've done so far is manual: adding the COM attributes to our classes, creating GUIDs, turning on project properties to create the ActiveX control, and so on. However, several years ago, Microsoft released a free VS add-in that automates this process: the Interop Forms Toolkit. Although its name implies it's for creating forms, it can also create UserControl-based ActiveX controls. You can download and install the Interop Forms Toolkit from http://tinyurl.com/5pa5gg.

One potential downside of the Interop Forms Toolkit is that its templates are in Visual Basic rather than C#. Of course, that isn't a problem if you'd rather work in Visual Basic or don't mind working in two languages. Although I haven't tried it, someone has created a C# version at http://tinyurl.com/3yf7uc3. Another downside is that it doesn't work in the free VS Express versions.

Let's try it out. Create a new project, but select Visual Basic as the language and "VB6 Interop UserControl" as the template. Name the project "FirstInteropControl" (see **Figure 19**).

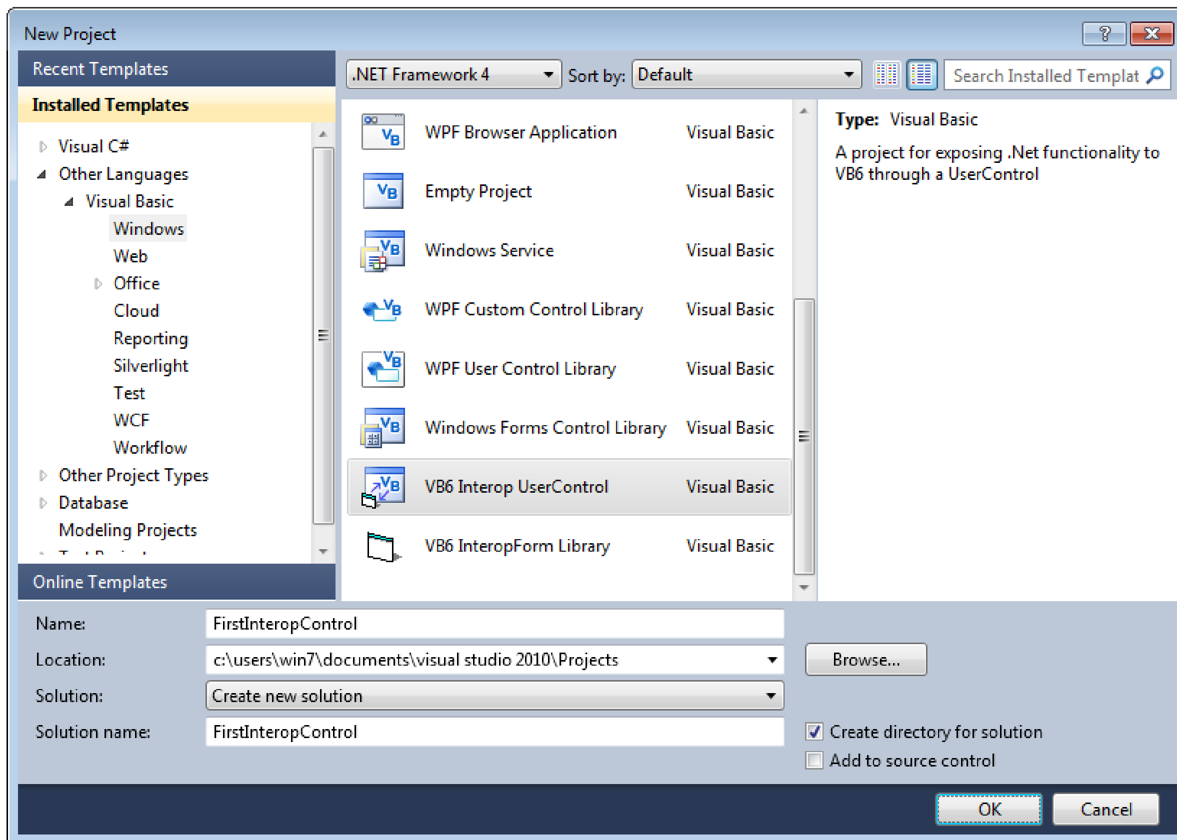

Figure 19. Use the "VB6 Interop UserControl" template to create an ActiveX control using the Interop Forms Toolkit.

After project creation is complete, open InteropUserControl.vb. This is essentially the same as a Windows UserControl, so drag a CheckedListBox to it and name it listBox. Size the user control to fit the listbox, then set the listbox's Anchor property to Top, Bottom, Left, and Right.

Right-click InteropUserControl.vb in the Solution Explorer and choose View Code. Notice the ComClass attribute at the top of the file. The syntax is a little different than the ClassInterface attribute we used in C# but the purpose is the same. Expand the "VB6 Interop Code" region and then expand the "COM Registration" region. Again, the code is different but its purpose is obvious: registering and unregistering the ActiveX control just like we did with the C# classes. Right-click the InteropLibrary project and choose Properties. On the Application page, click Assembly Information and notice that "Make assembly COM-visible" is already turned on. On the Compile page, you'll see that "Register for COM interop" is already turned on. In other words, the Interop Forms Toolkit is doing the same thing we were doing manually in our C# project; it just does most of the work for us.

Let's expose a few things so we make the class useful. Add the following code to the class:

```
Public Sub AddItem(ByVal Text As String, ByVal Checked As Boolean)
    ListBox.Items.Add(Text, Checked)
```

```
End Sub

Public ReadOnly Property CheckedItems As CheckedListBox.CheckedItemCollection
    Get
        Return ListBox.CheckedItems
    End Get
End Property

Public Property FontName As String
    Get
        Return ListBox.Font.Name
    End Get
    Set(ByVal value As String)
        ListBox.Font = New System.Drawing.Font(value, ListBox.Font.Size)
    End Set
End Property

Public Property FontSize As Single
    Get
        Return ListBox.Font.Size
    End Get
    Set(ByVal value As Single)
        ListBox.Font = New System.Drawing.Font(ListBox.Font.Name, value)
    End Set
End Property
```

The AddItem method adds an item to the listbox and accepts two parameters: the text for the item and whether it's initially checked or not. The CheckItems property returns a collection of those items in the listbox that are checked. The FontName and FontSize properties allow you to set the font for the listbox.

Sidebar: Properties in .Net

Properties in .Net aren't really like properties in VFP. For one thing, they don't actually store values. Instead, they're a special type of method with "get" and "set" components to read from and write to something. Typically, a property is the public interface to private underlying variables known as "fields." So used in that way, a .Net property is similar to a VFP property with Access and Assign methods. You can code any behavior you wish in those methods, even choosing to not store a value but instead calculate it on the fly. In our code above, the CheckedItems, FontName, and FontSize properties are really wrappers for other, more difficult to access properties.

Notice that CheckedItem doesn't have a "set" component, so it's a read-only property. You can also create write-only properties by having a set without a get.

To try it out, build the project, start VFP, create a form, and add FirstInteropControl.InteropUserControl. Name it oList, set Anchor to 15, and put the following code into Init:

```
This.AddItem('Apples', .F.)
This.AddItem('Bananas', .T.)
This.AddItem('Oranges', .F.)
```

```
This.AddItem('Peaches', .F.)
This.AddItem('Plums', .T.)

This.FontName = 'Segoe UI'
This.FontSize = 9
```

Add a command button, set Anchor to 4, and add this code to Click:

```
lcItems = ''
for each lcItem in Thisform.oList.Object.CheckedItems
	lcItems = lcItems + iif(empty(lcItems), '', chr(13)) + lcItem
next lcItem
messagebox('You selected: ' + chr(13) + chr(13) + lcItems)
```

Run the form, check some items, and click the command button to see which items you checked. See **Figure 20** for the result.

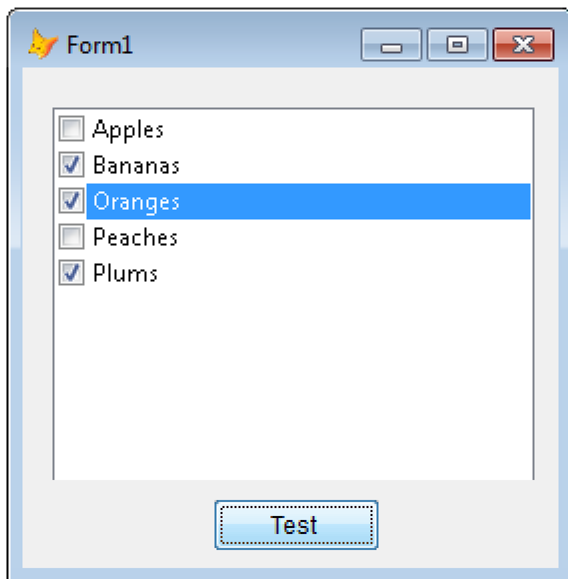

Figure 20. This VFP form hosts a .Net CheckedListBox control.

If you're interested in reading more about the Interop Forms Toolkit, Bernard Bout created a six-entry blog series about his experiences at http://tinyurl.com/39m5opf.

Deploying .Net ActiveX controls

So far, we've been using the .Net ActiveX controls on the same machine as VFP. To deploy on another system, you need to do two things: copy the control's files to the system and register the control for COM access.

In the project folder on your development system, you'll find a "bin" folder. That folder has a Debug folder (although if you change the Configuration in the Build page of the project properties dialog to Release, you'll find a Release folder) that contains three files: your project's DLL, PLB (program database file which contains debugging information), and TLB

(type library) files. We don't need the PLB file and even the TLB is optional since we can regenerate it on the user's system as you'll see in a moment. So, copy either just the DLL or both the DLL and TLB files to the user's system. Normally, you'll do this as part of your application's installation process.

Because we configured the project so VS automatically registers our control for COM access, we didn't have to do anything special to get it to work with VFP. However, on another machine, you need to register the control before it can be used. You do that using the utility RegAsm.exe that comes with the .Net framework. RegAsm is like RegSvr32, which you may know is used to register COM objects, but is used to register .Net assemblies for COM.

RegAsm.EXE is located in one of these directories:

- C:\Windows\Microsoft.NET\Framework\v2.0.50727 to register an ActiveX control targeted for the .Net framework 2.0, 3.0, or 3.5.
- C:\Windows\Microsoft.NET\Framework\v4.0.30319 for assemblies targeting .Net 4.0.

To manually register the control, use the following syntax:

```
RegAsm <path and name of DLL> /codebase
```

If you didn't copy the TLB file, use this syntax instead to both register the controls and regenerate that file:

```
RegAsm <path and name of DLL> /codebase /tlb
```

Note that if you're running on Windows Vista or Windows 7 or on an earlier version of Windows under a limited account, you need to run RegAsm as administrator or it'll fail. In that case (assuming you're doing that manually and not part of an installer), create a BAT file with the appropriate command and run it as administrator. If you run the BAT file several times, you may wish to deregister the control before registering it, so use this syntax:

```
RegAsm <path and name of DLL> /u
```

For example, here's the content of a BAT file to register DotNetControls.DLL which is copied to my Visual FoxPro Projects folder:

```
rem Unregister
C:\Windows\Microsoft.NET\Framework\v2.0.50727\regasm "C:\Users\Win7\Documents\Visual
FoxPro Projects\DotNetControls.dll" /u

rem Register
C:\Windows\Microsoft.NET\Framework\v2.0.50727\regasm "C:\Users\Win7\Documents\Visual
FoxPro Projects\DotNetControls.dll" /codebase /tlb

pause
```

(The PAUSE command is just there so I can see the results of the RegAsm call.)

RegAsm displays a message that you should sign an assembly using a strong name if you want to use the /CODEBASE parameter. In my experience, this isn't necessary, but isn't a bad idea and is easy to do. See http://tinyurl.com/2dmfzsl for information.

This may be fine for testing on another system, but for a user's system, you'll want to do this as part of your application's installer. I use Inno Setup for my installations, so I use something like this to install and register the ActiveX control:

```
[Files]
Source: "MyDotNetControl.dll";  DestDir: "{app}";  Flags: ignoreversion

[Run]
Filename: "{dotnet20}\Regasm.exe";  Parameters: "MyDotNetControl.dll /codebase /tlb";
WorkingDir: "{app}";  Flags: runhidden
```

The "runhidden" flag means the user won't see the DOS window and the results of the register process. "{dotnet20}" is an Inno Setup constant that points to the .Net 2.0 framework folder. Use "{dotnet40}" if you want .Net 4.0 instead.

If you're not sure whether the .Net framework is installed on the user's system, you can tell Inno Setup to download and install it if necessary. Add this to your script:

```
#include "DotNet2Install.iss"
```

This tells Inno Setup to load another install script file, DotNet2Install.iss, and use its contents as well. DotNet2Install.iss was modified from a script written by Priyank Bolia and posted at http://tinyurl.com/3yf59sa. This script requires ISXDL.DLL, which along with this script is included in the sample files accompanying this document. During the installation process, if the .Net 2.0 framework isn't found on the user's system, it's downloaded from Microsoft's web site and installed. The nice thing about this script is that you don't have to include the 22 MB .Net installer as part of your installer; it does, however, require that the user have an Internet connection.

References

In addition to the references given throughout this document, here are some others you may find useful:

- "Build and Deploy a .NET COM Assembly" by Phil Wilson, http://tinyurl.com/25qk2uo
- "Creating Multi-threaded .NET Components for COM Interop with Visual FoxPro" by Rick Strahl, http://tinyurl.com/3xozrhe
- "Interop Forms Toolkit - Your New Best Friend" by Beth Massie, http://tinyurl.com/3amtvtb

Summary

As you hopefully seen by now, creating ActiveX controls using .Net is fairly easy. It doesn't require a deep knowledge of .Net languages; what little you need to know is easy to pick up. The benefit is that you can leverage the thousands of .Net controls that are available to not only freshen the UI of your applications but also add functionality that would be difficult to write using just VFP code.

I'm interested in hearing about the types of controls you'd like to see, so please contact me and let me know about your ideas.

Biography

Doug Hennig is a partner with Stonefield Software Inc. He is the author of the award-winning Stonefield Database Toolkit (SDT); the award-winning Stonefield Query; the MemberData Editor, Anchor Editor, and CursorAdapter and DataEnvironment builders that come with Microsoft Visual FoxPro; and the My namespace and updated Upsizing Wizard in Sedna.

Doug is co-author of "Making Sense of Sedna and SP2," the "What's New in Visual FoxPro" series (the latest being "What's New in Nine"), "Visual FoxPro Best Practices For The Next Ten Years," and "The Hacker's Guide to Visual FoxPro 7.0." He was the technical editor of "The Hacker's Guide to Visual FoxPro 6.0" and "The Fundamentals." All of these books are from Hentzenwerke Publishing (http://www.hentzenwerke.com). He wrote over 100 articles in 10 years for FoxTalk and has written numerous articles in FoxPro Advisor, Advisor Guide to Visual FoxPro, and CoDe. He currently writes for FoxRockX (http://www.foxrockx.com).

Doug spoke at every Microsoft FoxPro Developers Conference (DevCon) starting in 1997 and at user groups and developer conferences all over the world. He is one of the organizers of the annual Southwest Fox conference (http://www.swfox.net). He is one of the administrators for the VFPX VFP community extensions Web site (http://vfpx.codeplex.com). He has been a Microsoft Most Valuable Professional (MVP) since 1996. Doug was awarded the 2006 FoxPro Community Lifetime Achievement Award (http://tinyurl.com/ygnk73h).

Copyright, 2011 Doug Hennig.