Link Files Across Projects

VS.NET makes it easy for multiple-project solutions to share solution-wide attributes.

by Juval Löwy and Fabio Claudio Ferracchiati

Technology Toolbox

- ☑ VB.NET
- ☑ C#
- ☐ SQL Server 2000
- ☑ ASP.NET
- ☐ XML
- ☐ VB6

Go Online!

Use these Locator+ codes at www.visualstudiomagazine.com to go directly to these resources.

Download

VS0309QA Download the code for this article, which includes a linked-files demo, a WinForms custom-control icon, and an About box demo.

Discuss

VS0309QA_D Discuss this article in the .NET Framework / IDE forum.

Read More

VS0309QA_T Read this article online.

VS0307QA_T Q&A, "Implement an ASP.NET Back Control," by Juval Löwy

VS0210BB_T Black Belt, "Generate Assemblies With Reflection Emit," by Randy Holloway

VS0206QA_T Q&A, "Develop Rich-UI Apps," by Karl E. Peterson and Juval Löwy

Q: Link Files Across Projects

I have a solution with many projects, and I want all the projects to share the same version number and strong-name keys file. How can I do this without duplicating the AssemblyInfo file?

A:

Every .NET project you create with VS.NET contains a file called AssemblyInfo.cs (or AssemblyInfo.vb), which includes assembly-wide attributes, such as version number, strong-name keys filename, assembly title, and culture. Although you can place these attributes anywhere in your source files, the convention is to put them all in the AssemblyInfo file, so you have a centralized, known place for them. The convention is also to avoid placing any code in the AssemblyInfo file. This works fine for a single project, but you often want all projects in a solution that has multiple projects to share the same version number and strong name, as well as other solution-wide attributes. If you have one AssemblyInfo per project, you must update all these files manually on every change. Fortunately, you have an easy solution to this problem: VS.NET's little-known ability to link files across projects.

For example, consider the sample MyApp solution, which contains two class libraries (ClassLibrary1 and ClassLibrary2) and a client project (MyClient) that uses these libraries (download the source code from the *VSM* Web site; see the Go Online box for details). You can factor the solution-wide attribute into a SolutionInfo.cs file (see Listing 1). The SolutionInfo.cs file contains the solution-wide version number, the company's name and copyright notice, and the strong-name keys filename. Place

C# • Link All Projects

```
using System.Reflection;
using System.Runtime.CompilerServices;

[assembly: AssemblyCompany("My Company")]
[assembly: AssemblyProduct("My Product")]
[assembly: AssemblyCopyright("(c) My Company")]
[assembly: AssemblyVersion("1.0.*")]
[assembly: AssemblyDelaySign(false)]
[assembly: AssemblyKeyFile(@"..\..\..\MyApp.snk")]
[assembly: AssemblyKeyName("")]
```

Listing 1 The SolutionInfo.cs file contains solution-wide information, such as the name of the strong-name keys file, information about the company and products, and version number. All projects in the solution must link to this file.

C# • Edit Assembly Information

```
using System.Reflection;
using System.Runtime.CompilerServices;

[assembly: AssemblyTitle("")]
[assembly: AssemblyDescription("")]
[assembly: AssemblyConfiguration("")]
[assembly: AssemblyTrademark("")]
[assembly: AssemblyCulture("")]
```

Listing 2 The AssemblyInfo.cs file contains project-specific information, such as the assembly name, description, and locale. You provide version and strong-name information by linking to the SolutionInfo.cs file.

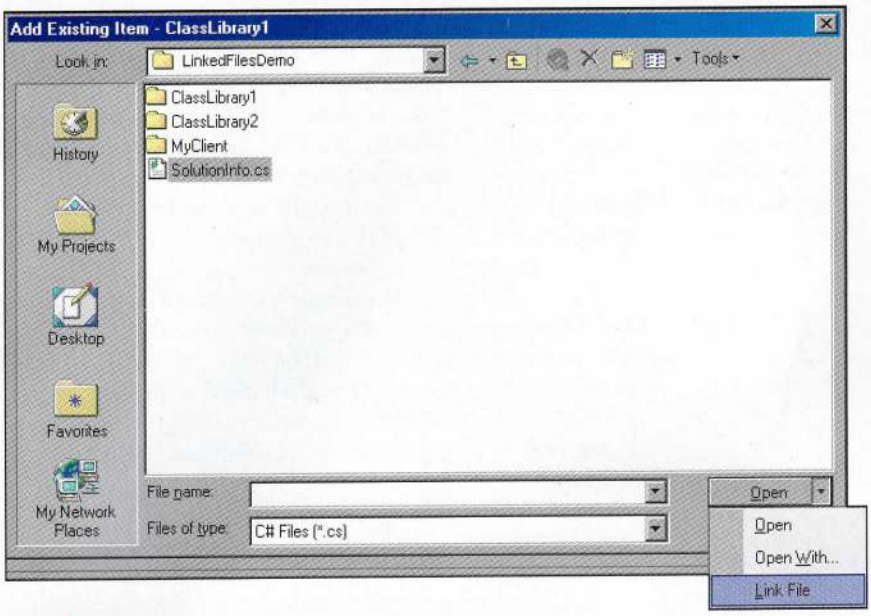

Figure 1 Add a Link to a File. When you link to a file, VS.NET merely adds a link to that file from the project and avoids copying the file itself. VS.NET still takes the information in the linked file into account when you build the project.

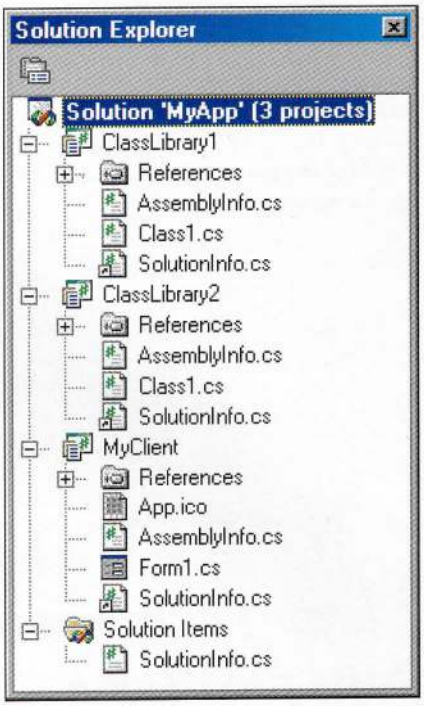

Figure 2 Link Multiple Projects. All projects in this solution link to the SolutionInfo.cs file, which contains solution-wide attributes and resides in the Solution Items folder. Every project has a dedicated AssemblyInfo.cs file, with project-specific attributes.

this file in the solution's root. Add it to the solution by right-clicking on the solution, selecting Add | Add Existing Item... from the context menu, then selecting SolutionInfo.cs and Open.

VS.NET creates a new folder in the solution called Solution Items and places the SolutionInfo.cs file in it. Next, edit each project's AssemblyInfo.cs file so that it contains only project-specific attributes, such as assembly title, description, and culture (see Listing 2). Finally, you must link each project to SolutionInfo.cs. Select Add | Add Existing Item... from one project's popup context menu. Browse to the project's root and highlight the SolutionInfo.cs file. Don't double-click on it, because doing so simply makes a copy of the file and adds it to the project. Instead, click on the dropdown arrow to the right of the Open button and select Link File (see Figure 1). This adds a link to the SolutionInfo.cs file from the project. The code in a linked file is part of the project, but the file itself isn't. Repeat this process for the remaining projects (see Figure 2 for the end result). Note the shortcut icon in the three projects to the linked SolutionInfo.cs file. —*J.L.*

Q: Display a Custom Control Icon

How do I add an icon to a custom control and make the toolbox display it when I add the control to the toolbox?

A:

WinForms and ASP.NET both allow you to develop custom controls that you can add to the toolbox. VS.NET assigns an icon to a custom control if you don't provide it yourself. However, you should provide your own icons, because the default icon is generic and meaningless. Create the icon first; it must be a BMP file, 16-by-16 pixels in size. The color of the lower-left pixel is the transparent background color; for example, if you set this pixel to white, all white pixels in the icon are transparent and show the toolbox's gray underneath. Add the icon to the project, then bring up its properties. Set the Build Action to Embedded Resource, so that the control's class library contains the icon, rather than shipping it in a separate file (see Figure 3).

You can indicate to VS.NET which icon to associate with a control in two ways: implicitly and explicitly. You assign the icon implicitly by naming it exactly the same as the control and putting it in the project's root. For example, if the custom control is called MyControl, name the icon MyControl.bmp (download the sample code). Now, when you add the control to the toolbox, VS.NET displays MyControl.bmp as the control's icon. Implicit assignment also requires the custom control to be in the project's default namespace.

Explicit assignment relies on the ToolboxBitmap attribute. ToolboxBitmap has a number of overloaded constructors, and the one you choose depends on the way you want to bind to the icon. You can indicate which assembly the icon resides in, and you can provide the icon filename. For example, this version of the ToolBoxBitmap constructor accepts the name of the file containing the icon, and the assembly where the icon is embedded as a resource:

```
[ToolboxBitmap(typeof(MyControl),
   "MyIcon.bmp")]
public class MyControl : UserControl
{...}
```

You indicate the assembly by providing any type from it. If the embedded resource is in the same assembly as the control (as is often the case), you can simply provide the type of the custom control itself. If you provide only a type (indicating the assembly where the icon is embedded), then VS.NET looks in that assembly for an icon named the same as the control:

```
[ToolboxBitmap(typeof(MyControl))]
public class MyControl : UserControl
{...}
```

Be aware that VS.NET suffixes the embedded resource with the assembly's default namespace. As long as the custom control is in the default namespace, the icon file's name can be any name of your choosing. For example, if the default namespace is CustomControls, then this definition of the custom control can use the name MyIcon.bmp as-is:

```
namespace CustomControls
{
   [ToolboxBitmap(typeof(MyControl),
      "MyIcon.bmp")]
   public class MyControl : UserControl
   {...}
}
```

The code in a linked file is part of the project, but the file itself isn't.

However, if you place the control in a different namespace in the assembly, such as the nested namespace MoreControls, then you must name the icon file MoreControls.MyIcon.bmp:

```
namespace CustomControls
{
   namespace MoreControls
   {
      [ToolboxBitmap(typeof(MyControl),
         "MyIcon.bmp")]
      public class MyControl :
         UserControl
      {...}
   }
}
```

This is the case even if you reference the icon as MyIcon.bmp in the ToolboxBitmap attribute. —*J.L.*

Q: Create a Dynamic About Box

I'd like to develop an About box for my .NET application that retrieves information directly from the AssemblyInfo file. How can I do this?

A:

The .NET Framework provides some classes within the Reflection namespace that you can use to analyze assembly characteristics during application run time (download the sample code). You should understand first what happens when you execute a .NET application. The Common Language Runtime (CLR) module uses the manifest data within the assembly to find out which other assemblies must be executed. The CLR checks for the references to external assemblies you added during the development phase and loads the related libraries in memory. The CLR checks for each library's version and allows so-called "side-by-side" assembly execution—the possibility of having more than one version of the same assembly running both in memory and on the hard disk. You can use the Reflection namespace's classes to emulate the CLR's behavior and inspect an assembly at run time.

The compiler injects the information you add with the <Assembly> attribute into the assembly's manifest. (If you use VS.NET to build your application, you can find this information in the AssemblyInfo file.) You should use the Reflection namespace classes to read the information within the manifest. The first class to use is the Assembly class. It provides many useful shared methods, such as GetExecutingAssembly:

```
Dim asm As [Assembly] = [Assembly].GetExecutingAssembly()
```

The GetExecutingAssembly method retrieves a reference to the assembly that the current code—the application—is running from. The Assembly class contains many methods you can use to inspect the assembly at run time. I'll show you two methods for retrieving information from the manifest: GetCustomAttributes and GetName. GetCustomAttributes retrieves an array of Object data types where the attributes are stored in the assembly. You can go through each object the method retrieves to search for the attribute you need. GetName retrieves a reference to the AssemblyName class, which contains useful properties such as the assembly version.

This code retrieves an array of Object objects filled with the attributes specified in the manifest:

```
attributes = asm.GetCustomAttributes(True)
```

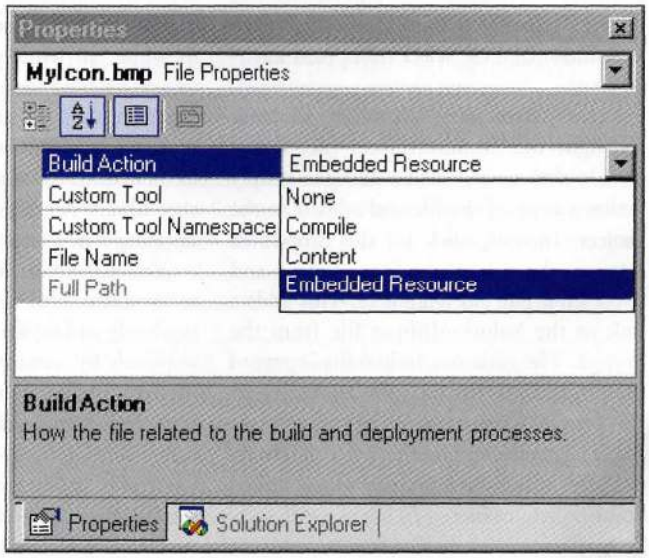

Figure 3 Embed a Resource. When you assign an icon to a custom control, you can set the Build Action on the bitmap file to Embedded Resource. This causes VS.NET to embed the file in the assembly, so you don't need to ship the icon along with the binary file that contains the control.

You can use a Select Case statement to retrieve only some of them:

```
For Each attr In attributes
   Select Case attr.GetType().ToString()
   Case "System.Reflection." & _
      "AssemblyTitleAttribute"
      dlgAbout.Text = CType(attr, _
         AssemblyTitleAttribute).Title
   Case "System.Reflection." & _
      "AssemblyDescriptionAttribute"
      dlgAbout.lbDescription.Text = _
         CType(attr, _
         AssemblyDescriptionAttribute _
         ).Description
   End Select
Next
```

The preceding code uses the GetType method to retrieve the current Object's type. You can use the ToString method to check the Object's type by comparing it to another string containing the attribute name and its namespace. After you find the right attribute, you use the CType function to convert the generic Object type into the specific attribute type. Once you've retrieved the right type, you can access the properties to retrieve the Title and the Description attributes and set the label text in the About box.

The AssemblyName class contains the Version property, which retrieves the AssemblyVersion attribute specified in the assembly's manifest:

```
dlgAbout.lbVersion.Text += " " & asm. _
   GetName().Version.ToString()
```

Thanks to the Assembly class's GetName method, you can obtain a reference to an object of the AssemblyName class and retrieve the assembly version. —*F.F.* VSM

Juval Löwy is a software architect and the principal of IDesign, a consulting and training company focused on .NET design and .NET migration. Juval is Microsoft's regional director for the Silicon Valley, working with Microsoft on helping the industry adopt .NET. His latest book is *Programming .NET Components* (O'Reilly & Associates). Juval speaks frequently at software-development conferences. Contact him at www.idesign.net.

Fabio Claudio Ferracchiati has 10 years of experience using Microsoft technologies. He's been focusing attention recently on the new .NET Framework architecture and languages and has written books for Wrox Press about this technology. He works in Rome for the CPI Progetti SpA company (www.cpiprogetti.it). Contact him at ferracchiati@rocketmail.com.

Additional Resources

Programming .NET Components by Juval Löwy [O'Reilly & Associates, 2003, ISBN: 0596003471]