STUDENT ACTIVITY 2.1: UNDERSTANDING .NET CLASS HIERARCHIES

MTA Course: 10754 Microsoft .NET Fundamentals
Topic: Understanding .NET class hierarchies
File name: 10754_Msft.NET_SA_2.1

Lesson Objective

2.1: Understand .NET class hierarchies. *This objective may include but is not limited to:* understanding system classes, classifications of classes, and logical organization of classes

Works Cited

Gunderloy, Mike. (2002). Understanding and Using Assemblies and Namespaces in .NET. Retrieved April 4, 2011, from MSDN® Library: *http://msdn.microsoft.com/en-us/library/ms973231.aspx*.

Understanding and Using Assemblies and Namespaces in .NET

(Mike Gunderloy, 2002)

Summary: Creating assemblies and namespaces in Microsoft® Visual Basic® .NET.

Directions to the student:

Read the following article and complete the hands-on activities described. Ask your instructor to further explain concepts that you do not fully understand. The instructor will verify your work.

Objectives

- Understand assemblies in Microsoft .NET.
- Understand namespaces in .NET.
- Use Microsoft Visual Basic .NET to create and customize an assembly.
- Use Visual Basic .NET to create a namespace.

Assumptions

The following should be true for you to get the most out of this document:

- You are familiar with Visual Basic programming.
- You understand the basic concept of object-oriented programming (OOP).
- You have access to Visual Basic .NET.

Contents

Introduction

Microsoft .NET provides several ways to think of your code as more than just a bunch of disconnected lines. As a Visual Basic programmer, you're already familiar with the concept of a *class*, a section of code that defines an object and its behavior. But two of the higher-level groupings may be unfamiliar to you:

- An assembly provides a fundamental unit of physical code grouping.
- A namespace provides a fundamental unit of logical code grouping.

As you'll see in this document, you can use Visual Basic .NET to create both assemblies and namespaces. You'll need to understand both of these concepts to be a productive Visual Basic .NET developer.

Assemblies

An assembly is a collection of types and resources that forms a logical unit of functionality. All types in the Microsoft .NET Framework must exist in assemblies; the common language runtime does not support types outside of assemblies. Each time you create a Microsoft Windows® Application, Windows Service, Class Library, or other application with Visual Basic .NET, you're building a single assembly. Each assembly is stored as an .exe or .dll file.

Note Although it's technically possible to create assemblies that span multiple files, you're not likely to use this technology in most situations.

The .NET Framework uses assemblies as the fundamental unit for several purposes:

- Security
- Type identity
- Reference scope
- Versioning
- Deployment

Security

An assembly is the unit at which security permissions are requested and granted. Assemblies are also the level at which you establish identity and trust. The .NET Framework provides two mechanisms for this level of assembly security: strong names and Signcode.exe. You can also manage security by specifying the level of trust for code from a particular site or zone.

Signing an assembly with a strong name adds public key encryption to the assembly. This ensures name uniqueness and prevents substituting another assembly with the same name for the assembly that you provided.

The Signcode.exe tool embeds a digital certificate in the assembly. This allows users of the assembly to verify the identity of the assembly's developer by using a public or private trust hierarchy.

You can choose to use either strong names, Signcode.exe, or both, to strengthen the identity of your assembly.

The common language runtime also uses internal hashing information, in conjunction with strong names and signcode, to verify that the assembly being loaded has not been altered after it was built.

Type Identity

The identity of a type depends on the assembly where that type is defined. That is, if you define a type named *DataStore* in one assembly, and a type named *DataStore* in another assembly, the .NET Framework can tell them apart because they are in two different assemblies. Of course, you can't define two different types with the same name in the same assembly.

Reference Scope

The assembly is also the location of reference information in general. Each assembly contains information on references in two directions:

- The assembly contains metadata that specifies the types and resources within the assembly that are exposed to code outside of the assembly. For example, a particular assembly could expose a public type named Customer with a public property named *AccountBalance.*
- The assembly contains metadata specifying the other assemblies on which it depends. For example, a particular assembly might specify that it depends on the System.Windows.Forms.dll assembly.

Versioning

Each assembly has a 128-bit version number that is presented as a set of four decimal pieces: *Major.Minor.Build.Revision*

For example, an assembly might have the version number 3.5.0.126.

By default, an assembly will only use types from the exact same assembly (name and version number) that it was built and tested with. That is, if you have an assembly that uses a type from version 1.0.0.2 of another assembly, it will (by default) not use the same type from version 1.0.0.4 of the other assembly. This use of both name and version to identify referenced assemblies helps avoid the "DLL Hell" problem of upgrades to one application breaking other applications.

Tip An administrator or developer can use configuration files to relax this strict version checking. Look for information on publisher policy in the .NET Framework Developer's Guide.

Deployment

Assemblies are the natural unit of deployment. The Windows Installer Service 2.0 can install individual assemblies as part of a larger setup program. You can also deploy assemblies in other ways, including by a simple xcopy to the target system or via code download from a website. When you start an application, it loads other assemblies as a unit as types and resources from those assemblies are needed.

The Assembly Manifest

Every assembly contains an assembly manifest, a set of metadata with information about the assembly. The assembly manifest contains these items:

- The assembly name and version
- The culture or language the assembly supports (not required in all assemblies)
- The public key for any strong name assigned to the assembly (not required in all assemblies)
- A list of files in the assembly with hash information
- Information on exported types
- Information on referenced assemblies

In addition, you can add other information to the manifest by using assembly attributes. Assembly attributes are declared inside of a file in an assembly, and are text strings that describe the assembly. For example, you can set a friendly name for an assembly with the *AssemblyTitle* attribute:

```
Copy

<Assembly: AssemblyTitle("Test Project")>
```

Table 1. Standard Assembly Attributes

Attribute	Meaning
AssemblyCompany	Company shipping the assembly
AssemblyCopyright	Copyright information
AssemblyCulture	Enumeration indicating the target culture for the assembly
AssemblyDelaySign	True to indicate that delayed signing is being used
AssemblyDescription	Short description of the assembly
AssemblyFileVersion	String specifying the Win32 file version. Defaults to the *AssemblyVersion* value.
AssemblyInformationalVersion	Human-readable version; not used by the common language runtime
AssemblyKeyFile	Name of the file containing keys for signing the assembly
AssemblyKeyName	Key container containing a key pair to use for signing
AssemblyProduct	Product name
AssemblyTitle	Friendly name for the assembly
AssemblyTrademark	Trademark information
AssemblyVersion	Version number expressed as a string

You can also define your own custom attributes by inheriting from the *System.Attribute* class. These attributes will be available in the assembly manifest just like the attributes listed above.

The Global Assembly Cache

Assemblies can be either private or shared. By default, assemblies are private, and types contained within those assemblies are only available to applications in the same directory as the assembly. But every computer with the .NET Framework installed also has a global assembly cache (GAC) containing assemblies that are designed to be shared by multiple applications. There are three ways to add an assembly to the GAC:

- Install them with the Windows Installer 2.0
- Use the Gacutil.exe tool
- Drag the assemblies to the cache with Windows Explorer

Note that in most cases, you should plan to install assemblies to the GAC on user computers by using the Windows Installer. The Gacutil.exe tool and the drag method exist for use during the development cycle. You can view the contents of your GAC by using Windows Explorer to navigate to the WINNT\assembly folder, as shown in Figure 1.

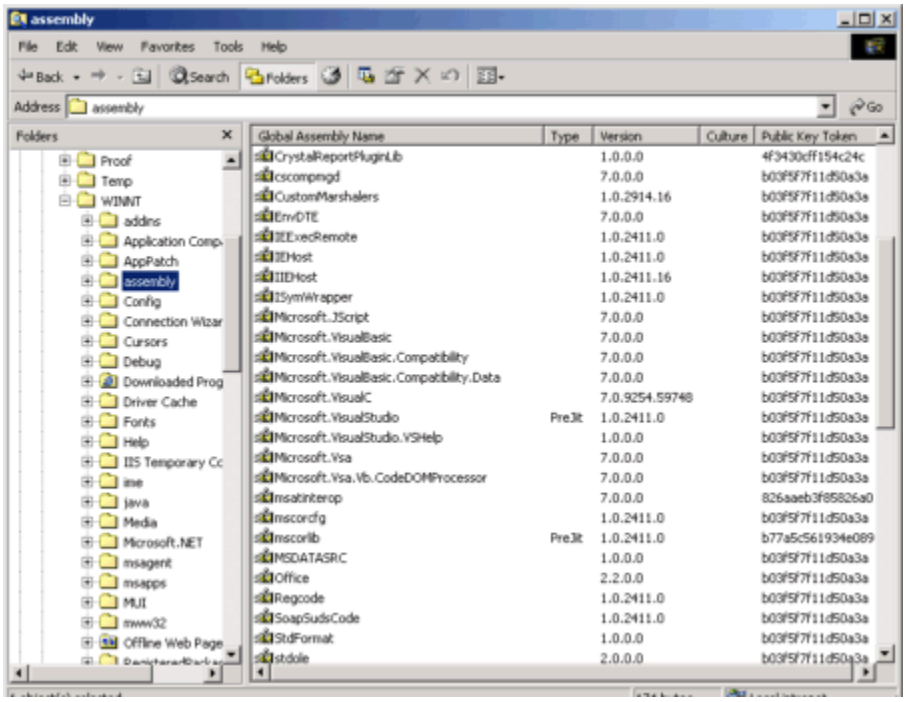

Figure 1. Viewing the GAC in Windows Explorer

Assembly Info.vb

When you create a new project using Visual Basic .NET, one of the components of the project will be a file named AssemblyInfo.vb. This file contains all of the assembly attributes that describe the assembly. To characterize your assembly, you should customize this information. You can edit the AssemblyInfo.vb file just like any other Visual Basic .NET source file. The default contents of this file look like this:

```
Copy

Imports System.Reflection

Imports System.Runtime.InteropServices


' General Information about an assembly is controlled through

    the following

' set of attributes. Change these attribute values to modify

    the information

' associated with an assembly.


' Review the values of the assembly attributes


<Assembly: AssemblyTitle("")>

<Assembly: AssemblyDescription("")>
```

```
<Assembly: AssemblyCompany("")>
<Assembly: AssemblyProduct("")>
<Assembly: AssemblyCopyright("")>
<Assembly: AssemblyTrademark("")>
<Assembly: CLSCompliant(True)>

'The following GUID is for the ID of the typelib if this project
   is exposed to COM
<Assembly: Guid("0C23E636-A54A-4F44-9432-E4ED4BD3017D")>

' Version information for an assembly consists of the following
    four values:
'
'      Major Version
'      Minor Version
'      Build Number
'      Revision
'
' You can specify all the values or you can default the Build
    and Revision Numbers
' by using the '*' as shown below:

<Assembly: AssemblyVersion("1.0.*")>
```

Namespaces

Another way to organize your Visual Basic .NET code is through the use of namespaces. Namespaces are not a replacement for assemblies, but a second organizational method that complements assemblies. Namespaces are a way of grouping type names and reducing the chance of name collisions. A namespace can contain both other namespaces and types. The full name of a type includes the combination of namespaces that contain that type.

The Namespace Hierarchy and Fully Qualified Names

You're probably already familiar with namespaces from the .NET Framework Class Library. For example, the *Button* type is contained in the *System.Windows.Forms* namespace. That's actually shorthand for the situation shown in Figure 2, which shows that the *Button* class is contained in the *Forms* namespace that is contained in the *Windows* namespace that is contained in the root *System* namespace.

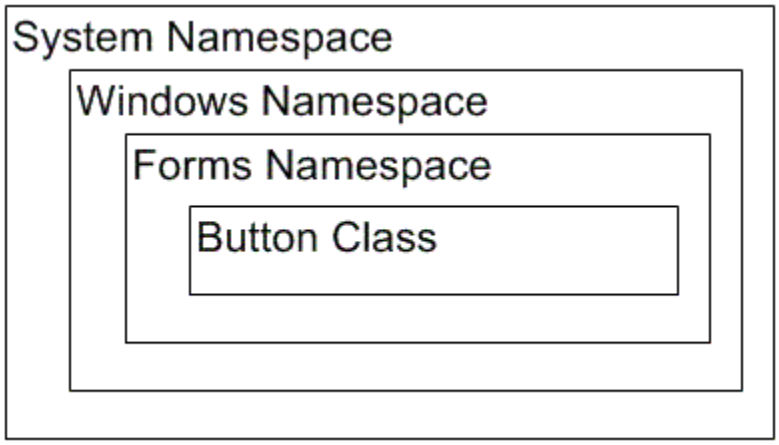

Figure 2. A namespace and class hierarchy

The fully qualified name of a class is constructed by concatenating the names of all the namespaces that contain the type. For example, the fully qualified name of the *Button* class is *System.Windows.Forms.Button.* The namespace hierarchy helps distinguish types with the same name from one another. For example, you might define your own class named *Button,* but it might be contained in the *ControlPanel* namespace within the *PowerPlant* namespace, making its fully qualified name *PowerPlant.ControlPanel.Button.*

Tip There's no need to use fully qualified names in your code unless you need to resolve an ambiguity between two types with the same type name used in the same project.

Declaring Namespaces

You can use the *Namespace* statement to declare a namespace in your own code. Namespace statements can be nested. For example, a Visual Basic .NET module could contain these lines of code:

```
Copy

Namespace PowerPlant

    Namespace ControlPanel

        Public Class Button

            ' Statements to implement Button

        End Class

    End Namespace

End Namespace
```

An alternative way to express this same hierarchy is to combine the *Namespace* statements:

```
Copy

Namespace PowerPlant.ControlPanel

    Public Class Button

        ' Statements to implement Button

    End Class

End Namespace
```

By default, a Visual Basic .NET project declares a *root namespace* that has the same name as the project. If the above declaration was in a project called PowerLib, then the fully qualified name of the *Button* class would be *PowerLib.PowerPlant.ControlPanel.Button.* You can change the name of the root namespace by following these steps:

1. In Project Explorer, right-click the project and select Properties.
2. Click Common Properties.
3. Enter a new name for the root namespace. It's good practice to use a name such as **CompanyName.Technology** for the root namespace, to avoid conflicts with namespaces defined by other developers.
4. Click OK.

   Note Strictly speaking, assemblies and namespaces are orthogonal. That is, you can declare members of a single namespace across multiple assemblies, or declare multiple namespaces in a single assembly. Unless you have a good reason for such an arrangement, though, it's best to keep things simple, with one namespace per assembly and vice versa.

Practice Creating an Assembly

In the following example, you'll create a class library containing a class that exposes a single method and a single event. Then you'll use assembly attributes to customize the assembly information.

Create the Class Library

Follow these steps to create the class library that will raise the events:

1. Open Microsoft Visual Studio® .NET, click Start, and then click New Project.
2. In the left pane tree view, select Visual Basic Project.
3. Select Class Library as the project template.
4. Set the name of the application to **PowerLib** and click OK.
5. In Solution Explorer, highlight the class called Class1.vb and rename it to **Button.vb**.

6. Select the code for *Class1* in Button.vb (this be an empty class definition) and replace it with the following code:

```
Copy
Public Class Button


    Private mfState As Boolean


    Public Sub Toggle()
        mfState = Not mfState
    End Sub


    Public Property State() As Boolean
        Get
            State = mfState
        End Get
        Set(ByVal Value As Boolean)
            mfState = Value
        End Set
    End Property


End Class
```

Customize the Assembly

Follow these steps to customize the assembly attributes:

1. In Solution Explorer, double-click the AssemblyInfo.vb file to open it in the code editor.
2. Select the first block of assembly attributes and modify them as follows:

```
Copy
<Assembly: AssemblyTitle("PowerPlant")>
<Assembly: AssemblyDescription("Power plant user interface")>
<Assembly: AssemblyCompany("Your Company")>
<Assembly: AssemblyProduct("PowerLib")>
<Assembly: AssemblyCopyright("Copyright 2002")>
<Assembly: AssemblyTrademark("")>
<Assembly: CLSCompliant(True)>
```

3. Select the version attribute and modify it as follows:

```
Copy

<Assembly: AssemblyVersion("1.0.0.0")>
```

Practice Creating a Namespace

In the following example, you'll add namespace information to the PowerLib project. Follow these steps to create the namespace information:

1. In the code editor, open the Button.vb file and add this line to the top of the module, above the declaration for the *Button* class:

```
Copy

Namespace ControlPanel
```

2. Visual Basic .NET automatically creates a corresponding *End Namespace* statement. Move this statement below the *End Class* statement that terminates the *Button* class.
3. In Solution Explorer, right-click the PowerLib project node and choose Properties.
4. Change the *Root* namespace property to **PowerPlant**.
5. On the File menu, click Save All.

Try It Out

On the Build menu, click Build Solution (or press Ctrl+Shift+B) to build the project. This will create the assembly and the namespaces that it contains. To view the assembly manifest, you can use the Ildasm.exe utility that ships with the .NET Framework. Follow these steps:

1. Click Start, click Programs, click Microsoft Visual Studio .NET 7.0 (or newer), click Visual Studio .NET Tools, and then click Visual Studio .NET Command Prompt.
2. At the command prompt, type **ildasm** and press Enter.
3. In the Ildasm interface, on the File menu, click Open. Navigate to My Documents\Visual Studio Projects\PowerLib\bin\PowerLib.dll and click Open.
4. In the ildasm window, expand the *PowerPlant.ControlPanel* namespace and you'll find the *Button* class. Expand the class and you'll see the members of its interface.
5. In the tree view, double-click the Manifest node and Ildasm will open the Manifest viewer. Scroll down and to the right, and you'll find the values of your custom assembly attributes, as shown in Figure 3.

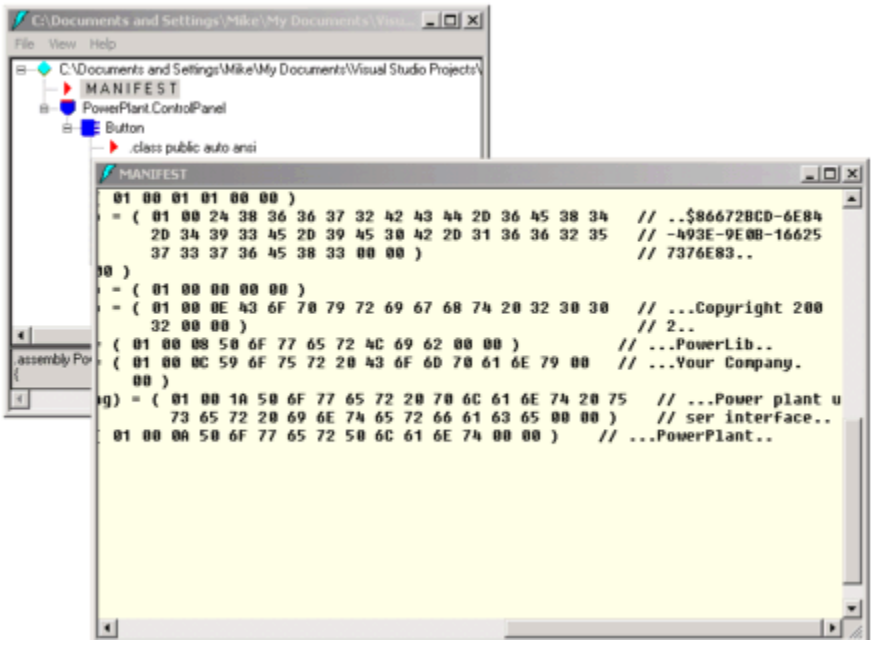

Figure 3. Viewing assembly manifest information

What's New Since Visual Basic 6.0?

The concepts of assemblies and namespaces are completely new to Visual Studio .NET. Visual Basic 6.0 can create class libraries, but those class libraries lack the versioning, deployment, security, and other special features of assemblies. Visual Basic 6.0 also provides no way to create a hierarchy of namespaces.

Summary

Visual Basic .NET allows you to group your code into logical units in several ways. First, by grouping code in assemblies, you can establish security, version, reference, and deployment boundaries. Second, by grouping classes into namespaces, you can create a hierarchy in which it's easy to identify classes by their fully qualified names. These two methods of organization complement one another, and you'll need to use both of them in your own Visual Basic .NET development.

About the Author

Mike Gunderloy writes about software and raises chickens in eastern Washington State. He's the coauthor of *Access 2002 Developer's Handbook* and author of *SQL Server Developer's Guide to OLAP with Analysis Services*, both from Sybex. He's been writing code for Microsoft products since the prehistoric pre-Windows era and has no intention of stopping any time soon.

About Informant Communications Group

Informant Communications Group, Inc. (*www.informant.com*) is a diversified media company focused on the information technology sector. Specializing in software development publications, conferences, catalog publishing, and websites, ICG was founded in 1990. With offices in the United States and the United Kingdom, ICG has served as a respected media and marketing content integrator, satisfying the burgeoning appetite of IT professionals for quality technical information.