Assemblies

In visual studio 6.0, application deployment was based on DLL files and this DLL based deployment has the following drawbacks that are called as DLL hell.

1. Side by side execution of two different versions of same component on a system is not possible. Because DLL file doesn't contain version information and when you install a new version, it automatically overwrites the old version.
2. To use a DLL based component, you must first register in windows registry and it is not possible to use a DLL based component directly from a CD or network resource. Hence it is not possible to use DLL based components on other operating systems.

To eliminate the drawbacks of DLL based deployment, in .net **Assembly** based deployment was introduced. An assembly contains version information. Hence during the installation of a new version, it will not overwrite the old version and hence you can run two different versions of a component on a system side by side. You can use an assembly directly from a CD or network resource without registering it in windows registry. Another advantage of assemblies is they will be loaded in to memory as and when required and not every time the .net application was run. Hence memory resources can be efficiently managed with assemblies.

The fundamental building block of a .net application is an assembly. An assembly will contain metadata i.e. the data about the assembly, application's MSIL code and object types available within the assembly. When we are creating a .net application using either vb.net or C#.net, assembly will be created in the form of a single portable executable (PE) file, specifically an exe or dll.

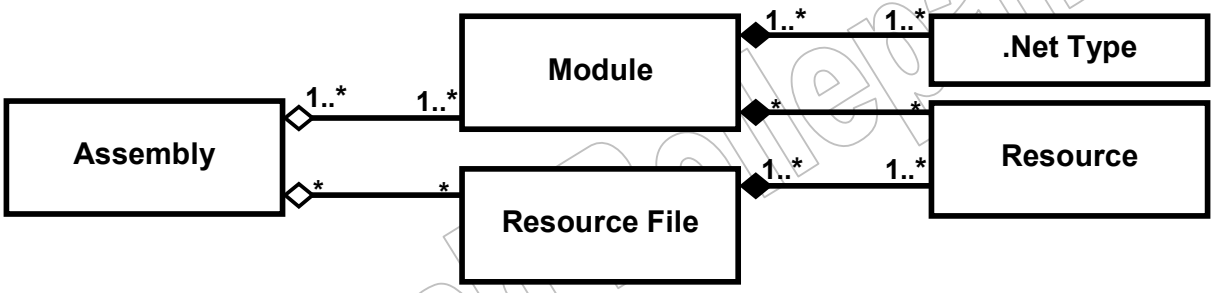
An assembly may be a single module assembly or multi module assembly. A project may be divided in to modules and each module may be developed by a separate developer and finally all these modules are combined in to a single assembly. By default VS.net will create single module assemblies and it is not possible to create multi module assemblies from VS.net. To create a multi module assembly, command line tools **vbc.exe, csc.exe or AL.exe** is used. Advantages of multi module assemblies are

1. A big portion of unused code can be separated in to a module so that it will never be loaded in to memory.
2. When a project is divided in to modules, developer of one module may be interested in VB.net while another developer in C#.net. This is possible only with multi module assemblies.

In a multi module assembly, there will be one main module, which will be taken in to consideration by the CLR. This main module will contain references to other modules and resource files in the assembly. Main module will be saved with the extension .exe or .dll and other modules in the assembly will be saved with the extension .netmodule.

Resource Files

Along with the modules, an assembly can also contain resources like bitmaps and xml documents. These resources can be directly embedded in to assembly or can be stored in a file and can be referenced by the assembly. Relationship between assembly, module and resource files is represented using UML diagram as follows.



Creating Multi Module Assemblies

The following example creates one module in vb.net and one module in c#.net and both are combined to form a single module assembly.

1. Open notepad and type the following code in vb.net syntax and save it with the name **Main.vb** in **D:**.

```
Namespace Test
    Module Module1
        Sub Main()
            Console.WriteLine("From Main Module")
            Dim C as new Class1
            Console.WriteLine(C.ToString())
        End Sub
    End Module
End Namespace
```

2. Within notepad write the following code in c# syntax and save it with the name **CsModule.Cs** in **D:**.

```
Namespace Test
{
        Class Class1
        {
                Public override string ToString()
                {
                        Return "From CS Module";
                }
        }
}
```

3. Compile the module created in c# **CsModule.Cs** to **CsModule.Netmodule** with the help of the command line tool **CSC.exe** as follows at the command prompt. Command prompt can be opened by selecting **Visual Studio 2008 Command Prompt** in **Visual Studio Tools** in **Visual Studio 2008** in **Programs** in **Start menu.**

```
CSC /target : module D:\CsModule.Cs
```

4. Copy the **CsModule.Netmodule** to **D:**
5. Compile the **Main.vb** file by adding **CsModule.Netmodule** to it with the help of **VBC.exe** command line tool as follows at command prompt to generate **Main.exe** file.

```
VBC /addmodule : D:\CsModule.Netmodule D:\Main.vb
```

The **Main.exe** is now a multi module assembly that contains two modules, one created in vb.net and another created in c#.

Structure of Modules in An Assembly

Main Module

PE Header	CLR Header	Manifest	Metadata	MSIL code	Resources

Sub Module

PE Header	CLR Header	Metadata	MSIL code	Resources

PE Header : Contains smallest windows version on which the assembly can be executed.

CLR Header : Contains the version of CLR for which assembly was targeted.

Manifest : This is available only in main module of the assembly and it Contains references to sub modules and resource files available in the assembly.

Metadata : Contains classes, structures, enumerations, interfaces and delegates created in the assembly and it doesn't contain any code of these members.

MSIL Code : Contains MSIL code for the members of classes, structures, enumerations, interfaces and delegates that are available in assembly.

Resources : Contains any resources that are embedded in to the module and this section is optional.

Observing the Contents of An Assembly

To open and see the contents of an assembly, you can use the command line tool **ildasm.exe (Intermediate Language Disassembler).** Type the command **ildasm** at the command prompt that opens ildasm window. In this window in file menu choose open, select the exe or dll file in open file dialog box and click on open button, which will open the assembly in ildasm.

Types of Assemblies

Assemblies are classified in to **private** assemblies, **shared** assemblies and **satellite** assemblies. Another classification of the assemblies is **static** and **dynamic** assemblies. Static assembly is an assembly whose reference is added to the project at design time it self and dynamic assembly is an assembly whose reference is added to the project at runtime i.e. during execution of the project.

Private Assemblies

An assembly that is private to one particular application is called as private assembly. When a private assembly was referred by three different .net applications and all these applications are run at a time then three different copies the private assembly will be loaded in to memory, one for each application even they are referring to the same assembly. This wastes the memory resources.

Shared Assemblies

An assembly that is shared by more than one application is called as shared assembly. When a shared assembly was referred by three different .net applications and all these applications are run at a time then only one copy of the assembly is loaded in to memory and is shared by all three applications. This saves the memory resources. By default every assembly created in .net is a private assembly. To convert a private assembly to shared assembly, you have to perform the following steps.

1. Signing the assembly
2. Placing assembly in Global Assembly Cache (GAC)

Signing the Assembly

When an assembly was provided with a strong name then that assembly is called as signed assembly. To sign the assembly, follow the following two steps.

1. Create a strong name key file
2. Associate strong name key file to the assembly

Creating Strong Name Key File

To create a strong name key file, use the command line tool **Sn.exe** that has the following syntax.

Sn –k MyKey.snk

This will create a strong name and writes it to the **MyKey.snk** file.

Associating Strong Name Key File To Assembly

To associate the strong name key file generated using **Sn** command line tool, first open the project, which you want to convert to shared assembly in VS.net and open solution explorer. Within the solution explorer click on **show all files** button and then double click and open **AssemblyInfo.vb** or **AssemblyInfo.cs** file and write the following entry in that file.

<Assembly : AssemblyKeyFile("KeyFilePath")> // C#.Net
[Assembly : AssemblyKeyFile("KeyFilePath")] // VB.Net

After creating and associating strong name key file to the assembly build the solution with shortcut **Ctrl + Shift + B** to make the generated assembly as signed assembly.

Placing Assembly In GAC

To place an assembly in to GAC, use the command line tool **GACUTIL.exe** that has the following syntax.

Gacutil /i Pathoftheassembly

GAC is nothing but a folder with the name **assembly** in windows folder. Once the assembly is placed in GAC, it will become a shared assembly. If you want to remove the assembly from GAC, use the following syntax of **Gacutil** command line tool.

Gacutil /u Pathoftheassembly

Signing the Assembly In .net 2005

In .net 2005 you can sign the assembly from within VS.net without using any command line tools and has the following steps.

1. Open the project which you want to convert to shared assembly in VS.net and then open the properties of project.
2. Within the properties of the project select **Signing** tab and then check the check box **Sign the assembly** that activates a combo box.
3. Within the combo box select **new** to create a new **strong name key file**. This will open a dialog box asking you to enter a **name** for the strong name key file and **password** to provide security. Provide a name and password and click on ok button.
4. Close properties window, save the project and build the solution to generate a signed assembly for the project.

After signing the assembly, place it in GAC in order to convert it to shared assembly.

Secret Behind Strong Naming

Strong name is not a name and is a pair of keys called **private key** and **public key**. Strong name is used by the CLR to uniquely identify the assembly and provide security for the assembly from tampering by others. When a strong name is associated to the assembly, then a **hash algorithm** is applied on current content of the assembly to generate a **hash value** and this hash value is **encrypted** with the help of **private key** to get **digital signature** and will be stored in **manifest** of the main module and **public key** is stored in **metadata** of the main module.

While executing the assembly, the digital signature available in manifest will be decrypted with the help of public key available in metadata to get original hash value of the assembly and at the same time hash algorithm is applied on the assembly to get hash value based on current content of the assembly. If both of these hash values are matched then assembly was not tampered and will be executed. Otherwise causes error and will not be executed.

Delayed Sigining Of The Assembly

Any one who knows the hash value and private key of an assembly can tamper the assembly. The developers of the project will know these details of the assembly and hence they have a chance for tampering the assembly. To protect the assembly from this situation, delayed signing was provided. In delayed signing, first assembly will be signed only with public key and just before the assembly was released it will be resigned with both public and private keys only by the trusted person. The process of delayed signing will be as follows.

1. Create a strong name key file using sn command line tool as follows.

   Sn –k MyKey.Snk

2. Extract the public key from .snk file created in first step and store it in another strong name key file using sn command line tool as follows.

   Sn –p MyKey.Snk MyPublicKey.Snk

3. Sign the assembly using the strong name key file that contains only public key.
4. When assembly was signed with strong name key file that contains only public key, then hash value will not be encrypted as private key was not available. But space will be reserved for digital signature in the manifest and will be empty. In this case if you try to use the assembly, CLR will try to verify whether assembly was tampered by decrypting the digital signature that was not available and will cause an error. To skip the verification process, write verification skip entries to the assembly with the help of sn command line tool as follows.

   Sn –Vr E:\MyLibrary\MyLibrary\Bin\Release\MyLibrary.Dll

5. Just before releasing the component, resign the assembly with strong name key file that contains both private and public keys with the help of sn command line tool as follows.

   Sn –R MyKey.Snk E:\MyLibrary\MyLibrary\Bin\Release\MyLibrary.Dll

6. After resigning the assembly remove the verification skip entries written to the assembly to skip the process of verifying whether assembly was tampered with the help of sn command line tool as follows.

   Sn –Vx E:\MyLibrary\MyLibrary\Bin\Release\MyLibrary.Dll