Assemblies

Source	http://www.akadia.com/services/dotnet_assemblies.html

Introduction

Summary

In any programming environment it is of particular importance to be able to encapsulate dedicated functionality in libraries. We like to build modules and must be in the position to use the same code in different locations. Additionally, we want to save resources and have binaries loaded only if required by the functionality a program is using at a given time (dynamic linking). Libraries can be part of a single application or shared for the use by other programs, implemented by our own or even by other companies.

In this context we need answers to the two questions:

How libraries are located ?	How they are identified ?

In the past Dynamic Link Libraries (DLL) were located by the system environment's PATH setting and the Windows Registry. Globally and shared DLLs were stored in the Windows system folder. The identification was specified by the DLL's file name. These mechanisms do have certain drawbacks. One is that we can't move applications in our file system because the Windows Registry does have paths stored. This is especially on bigger systems a significant problem, if more disk storage has to be added, data has to be moved, disk layouts reorganised and so on. Another thing is that DLLs can be overwritten by mistake or on purpose, for instance by other applications' installation programs. These newer DLLs may not be compatible anymore and other programs using them may crash.

With Microsoft.NET the Windows Registry is not used anymore and the libraries are stored either together with an application as Private Assemblies or in a Global Assembly Cache (GAC) as Global Assemblies to be shared among applications. Assemblies are signed and verified to recognise content modifications. They are identified by name and version to resolve incompatibility issues.

To conclude, the former DLL problems could be improved with the introduction of assemblies and all the related concepts. Developers are now in the position to use DLLs safely as libraries for single programs or to share functionality among other applications.

Scope

This documents describes the basic concepts of DLLs, Private and Global Assemblies. A sample application is used to explain an implementation in detail. Key aspects like signing and version control are worked out in different chapters. Version control with configuration

files is not part of this document. The command line debugger is introduced as a tool to verify program code and to learn more about DLLs and assemblies.

Setup of the .NET Framework

The Microsoft® .NET Framework is the infrastructure for the overall .NET Platform. The common language runtime and class libraries (including Microsoft Windows® Forms, ADO.NET, and ASP.NET) combine to provide services and solutions that can be easily integrated within and across a variety of systems.

The .NET Framework provides a fully managed, protected, and feature-rich application execution environment, simplified development and deployment, and seamless integration with a wide variety of languages.

Assemblies

The new basic entity is called an Assembly. This is a collection of class modules presented as a single DLL or EXE file. Even though EXE files can also be called assemblies, most of the time we talk about libraries if we use this word. Assemblies are very similar to Java Archives (JAR).

Physically, assemblies are files located somewhere on disk and are per definition so-called Portable Executable (PE). Assemblies are loaded on demand. They are not loaded if not needed.

Assemblies have Metadata stored to provide version information along with a complete description of methods and types. Part of this metadata is a Manifest. The manifest includes identification information, public types and a list of other used assemblies.

We distinguish between the typical DLLs, we've already used in the times before .NET, Private Assemblies, used for single programs, and Global Assemblies shared among several applications.

Assemblies are still DLLs even if they differ from the former DLLs. In our context, there is no difference between the direct use of DLLs and Private Assemblies. Thus the direct use of DLLs is still an issue and not outdated.

Direct Use of DLLs

Source files may be compiled into DLLs instead of EXEs. Even if we link them later on to assemblies, we first compile them to normal DLLs. During EXE file compilation DLLs may be added directly. There is no obligation to create assemblies first.

- Location is specified at compile time. Usually in the same folder like the application's EXE file or in any of the sub folders.
- PATH is not checked while looking up files, neither set by Control Panel 'System' configuration nor set in a Console Window.
- Identified by name only.
- Get smaller EXE files.
- Dynamic linking, i.e. loading on demand.

Private Assemblies

Intended use by single applications. Building modules to group common functionality.

- **Location is specified at compile time. Usually in the same folder like the application's EXE file or in any of the sub folders.**
- **PATH is not checked while looking up files, neither set by Control Panel 'System' configuration nor set in a Console Window.**
- **Identified by name and version if required. But only one version at a time.**
- **Digital signature possible to ensure that it can't be tampered.**
- **Get smaller EXE files.**
- **Dynamic linking, i.e. loading on demand.**

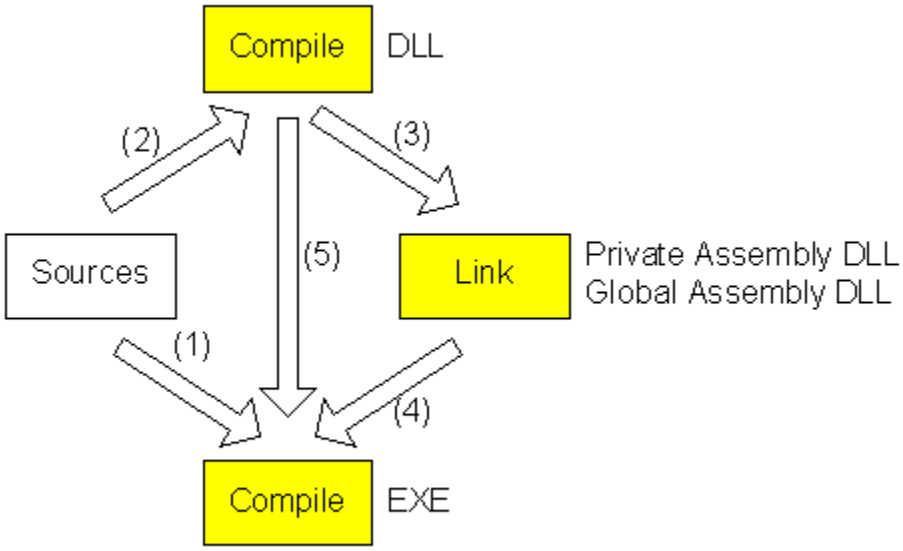
Global Assemblies

Publicly sharing functionality among different application.

- **Located in Global Assembly Cache (GAC).**
- **Identified by globally unique name and version.**
- **Digital signature to ensure that it can't be tampered.**
- **Get smaller EXE files.**
- **Dynamic linking, i.e. loading on demand.**

Compile / Link Cycle

To create an application we have to compile our source files and to end up in an Executable (EXE). There are different ways to do this:



View Assemblies - The Intermediate Language Disassembler (ILDASM)

Display metadata of one of your .NET programs or libraries by the use of the ILDASM tool:

```
zahn@ARKUM> ildasm app1.exe
```

We can see the assembly's metadata with all the methods and types in a tree representation. If we click on ' M A N I F E S T ' we get a window, which shows the manifest information. Even if there are a lot of non well readable lines, we are able to identify version information and all the referenced assemblies used to run the displayed executable or library.

Sample Application

Our sample application is somehow similar to the typical 'Hello world' programs everywhere around. The program meets somebody and says 'Hello', 'How do you do' and 'Good bye'. All the three greetings printed by another library.

Steps

1. **Create the Private Assembly** `GreetAssembly.dll` **from** `Hello.dll` **and** `GoodBye.dll`
2. **Create the Global Assembly** `HowDoYouDoSharedAssembly.dll` **from** `HowDoYouDo.dll`
3. **Reference Private and Global Assemblies within** `App1.exe`
4. **Call DLLs directly and reference Global Assemblies within** `App2.exe`

After we have all the three libraries ready, we may compile two EXE main programs. The first one with a reference to `GreetAssembly.dll` **and the second adding the** `Hello.dll` **and** `GoodBye.dll` **files directly.**

App.cs

The main application `App.cs` **program looks like this:**

```
using csharp.test.app.greet;
namespace csharp.test.app {
  public class Application {
    public static void Main() {
      Application a = new Application();

      Hello h = new Hello();
      h.SayHello();

      HowDoYouDo d = new HowDoYouDo();
      d.SayHowDoYouDo();

      a.Leave();
    }
    private void Leave() {
      GoodBye g = new GoodBye();
      g.SayGoodBye();
    }
  }
}
```

The `Main()` **method creates an Application object to invoke the** `Leave`**() method at the end to say 'Good bye'. Inside** `Main`**(), just after the Application object creation, the Hello and** `HowDoYouDo` **classes are instantiated to say 'Hello' and 'How do you do'.**

The Leave() method is created to verify that, while the program runs, not all the DLLs are loaded. Instead, the DLL with GoodBye stays apart until we enter the `Leave()` **method's scope. We are going to investigate this later on in details.**

We need a 'using' declaration because `Hello` **and** `GoodBye` **are in another namespace as the main program.**

Hello.cs

There is a simple class providing a method to print out a 'Hello':

```
namespace csharp.test.app.greet {
  public class Hello {
    public void SayHello() {
      System.Console.WriteLine("Hello my friend, I am a DLL");
    }
  }
}
```

The namespace is declared and equals to `GoodBye`***.cs*****.** `Hello`***.cs*** **and** `GoodBye`***.cs*** **will be put into a single Private Assembly. They must be in the same namespace.**

GoodBye.cs

Similar to `Hello`***.cs*** **but prints a 'Good bye':**

```
namespace csharp.test.app.greet {
  public class GoodBye {
```

```
    public void SayGoodBye() {
      System.Console.WriteLine("Good bye, I am a DLL too");
    }
  }
}
```

Uses the same namespace like `Hello`***.cs*****.**

HowDoYouDo.cs

We are going to implement this source file as a Global Assembly:

```
using System.Reflection;
[assembly:AssemblyKeyFile("app.snk")]
[assembly:AssemblyVersion("1.0.0.0")]
namespace csharp.test.app {
  public class HowDoYouDo {
    public void SayHowDoYouDo() {
      System.Console.WriteLine("How do you do, I am a Global Assembly");
    }
  }
}
```

With the Attributes at the top we specify the key file used to generate a hash code and to declare the version. For what this hash code is used and how the versioning exactly works will be discussed later on. Like in the other classes there is a method to print out a text.

Compile Classes to DLLs - The CSharp Compiler (CSC)

To compile our source files we use the C# Compiler (csc):

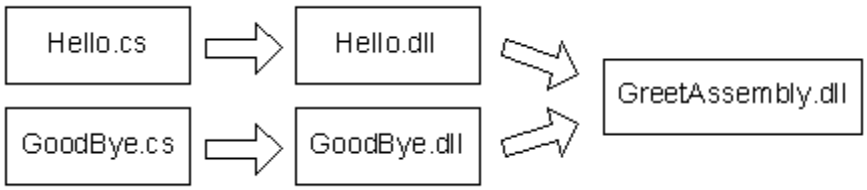
```
DotNet> csc /debug /t:module /out:bin\Hello.dll Hello.cs
DotNet> csc /debug /t:module /out:bin\GoodBye.dll GoodBye.cs
DotNet> csc /debug /t:module /out:bin\HowDoYouDo.dll HowDoYouDo.cs
```

While /debug includes debug information, we are going to use later on, the /t (target) switch lets us create a DLL. We are writing all our DLLs into a bin folder.

Group DLLs in a Private Assembly - The Assembly Linker (AL)

We like to combine `Hello.dll` **with** `GoodBye.dll` **and put them into a Private Assembly we call** `GreetAssembly.dll`**, this is Step 1 in our Sample Application.**

```
DotNet> al /t:library /out:bin\GreetAssembly.dll bin\Hello.dll
bin\GoodBye.dll
```



For this purpose we use the Assembly Linker. As /t (target) we generate here a library referencing the two other DLLs. This is also called a Multi-Module Assembly. Again, we store all the binaries in a bin folder.

MANIFEST

```
.assembly extern mscorlib
{
  .publickeytoken = (B7 7A 5C 56 19 34 E0 89 )
  .ver 1:0:3300:0
}
.assembly extern GreetAssembly
{
  .ver 0:0:0:0
}
.assembly extern HowDoYouDoSharedAssembly
{
  .publickeytoken = (06 FE 90 DA 7F E1 C8 9A )
  .ver 1:0:0:0
}
.assembly App1
{
  // --- The following custom attribute is added automatica
  //  .custom instance void [mscorlib]System.Diagnostics.Del
  //
  .hash algorithm 0x00008004
  .ver 0:0:0:0
}
.module App1.exe
// MVID: {30ED43E7-40D3-4399-9957-238AA009423F}
.imagebase 0x00400000
.subsystem 0x00000003
.file alignment 512
.corflags 0x00000001
// Image base: 0x031b0000
```

Why does the tool we have used right now call ' Intermediate Language Disassembler ' ? Just bring back into our minds that if we compile sources, for instance with the CSharp Compiler (CSC) or any other .NET language compiler, we won't get machine code executable on a computer. Instead we get Microsoft Intermediate Language (MSIL) or shortened to Intermediate Language (IL) stored as an EXE file. To get machine code and

to run the EXE we need the Common Language Runtime (CLR) part of the .NET installation somewhere on our computer.

~~~ End of Article ~~~