Unit 8

Understanding .NET Assemblies

8.1 Problems with Classic COM Binaries :

1. **COM Versioning :** COM runtime offers no intrinsic support to ensure that the correct version of a binary server is loaded for calling client. It is true that a COM programmer can modify the version of the type library, update the registry to reflect these changes, and even reengineer the client's code base to reference a particular library. But the fact remains that these are the tasks delegated to the programmer & typically require rebuilding the code base and redeploying the software.
2. **COM Deployment :** For the COM runtime to locate & load a binary, the COM server must be configured correctly on the target machine. But COM server requires a vast number of registration entries to be made. Typically, every COM class, interface, type library & application must be documented within the system registry.

8.2 An Overview of .NET Assemblies :

An assembly is a versioned, self-describing binary file hosted by the CLR. Now, despite the fact that .NET assemblies have exactly the same file extensions (*.exe or *.dll) as previous Win32 binaries (including legacy COM servers).

A .NET assembly (*.dll or *.exe) consists of the following elements:

1. A Win32 file header
2. A CLR file header
3. CIL code
4. Type metadata
5. An assembly manifest
6. Optional embedded resources

- The Win32 file header establishes the fact that the assembly can be loaded and manipulated by the Windows family of operating systems. This header data also identifies the kind of application (console based, GUI-based, or *.dll code library) to be hosted by the Windows operating system.
- The CLR header is a block of data that all .NET files must support (and do support, courtesy of the C# compiler) in order to be hosted by the CLR. In a nutshell, this header defines numerous flags that enable the runtime to understand the layout of the managed file.
- At its core, an assembly contains CIL code, which as you recall is a platform- and CPU-agnostic intermediate language. At runtime, the internal CIL is compiled on the fly (using a just-in-time [JIT] compiler) to platform- and CPU-specific instructions.
- An assembly also contains metadata that completely describes the format of the contained types as well as the format of external types referenced by this assembly. The .NET runtime uses this metadata to resolve the location of types (and their members) within the binary, lay out types in memory, and facilitate remote method invocations.

- An assembly must also contain an associated *manifest* (also referred to as *assembly metadata*). The manifest documents each module within the assembly, establishes the version of the assembly, and also documents any *external* assemblies referenced by the current assembly.
- Finally, a .NET assembly may contain any number of embedded resources such as application icons, image files, sound clips, or string tables. The .NET platform supports *satellite assemblies* that contain nothing but localized resources.

8.3 Single-File & Multifile Assemblies :

8.3.1 Single File Assemblies :

- If an assembly is composed of a single *.dll or *.exe module, then it is called as Single File Assembly.
- Single-file assemblies contain all of the necessary elements (header information, CIL code, type metadata, manifest, and required resources) in a single *.exe or *.dll package.
- Figure 11-3 illustrates the composition of a single-file assembly.

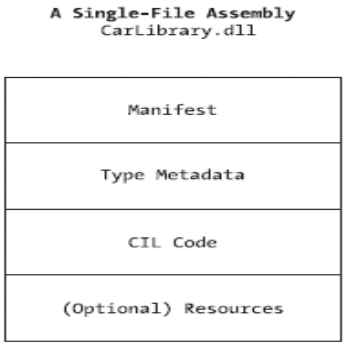


Figure 11-3. *A single-file assembly*

8.3.2 Multifile Assemblies :

- Multifile Assemblies are composed of numerous .NET binaries, each of which is termed a module.
- One of these *.dlls is termed the *primary module* and contains the assembly-level manifest (as well as any necessary CIL code, metadata, header information, and optional resources).
- The other related modules contain a module level manifest, CIL and type metadata.
- The major benefit of constructing multifile assemblies is that they provide a very efficient way to download content.
- Another benefit of multifile assemblies is that they enable modules to be authored using multiple .NET programming languages.
- Figure 11-4 illustrates a multifile assembly composed of three modules, each authored using a unique .NET programming language.

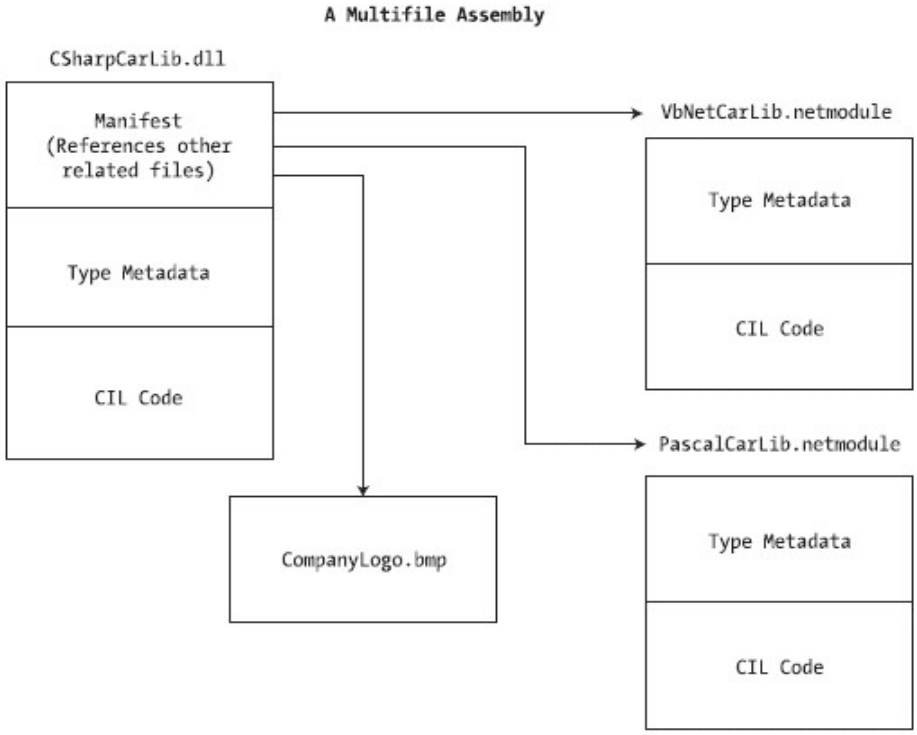
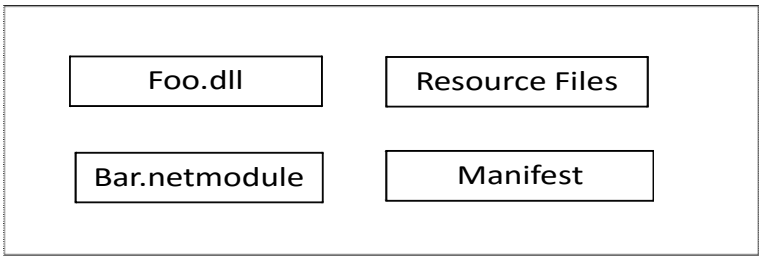


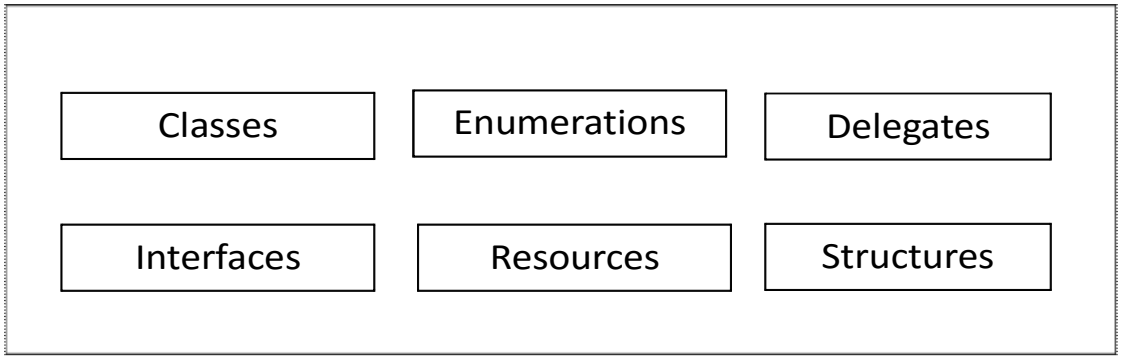
Figure 11-4. *The primary module records secondary modules in the assembly manifest*

8.4 Two Views of Assembly :

8.4.1 Physical View : In this case, the assembly can be realized as some number of files that contain your custom types & resources.



8.4.2 Logical View : In this case, you can understand an assembly as a versioned collection of public types that you can use in your current application.



8.4.3 Benefits of Assemblies :

1. **Assemblies Promote Code Reuse : A** *code library* (also termed a *class library*) is a *.dll that contains types intended to be used by external applications. When you are creating executable assemblies, you will be leveraging numerous system-supplied and custom code libraries as you create the application at hand. The .NET platform allows you to reuse types in a language-independent manner. For example, you could create a code library in C# and reuse that library in any other .NET programming language. It is possible to not only allocate types across languages, but derive from them as well.
2. **Assemblies Establish a Type Boundary:** A type's *fully qualified name* is composed by prefixing the type's namespace (e.g., System) to its name (e.g., Console). The assembly in which a type resides further establishes a type's identity.
3. **Assemblies Are Versionable Units :** .NET assemblies are assigned a four-part numerical version number of the form <*major*>.<*minor*>. <*build*>.<*revision*>. This number, in conjunction with an optional *public key value,* allows multiple versions of the same assembly to coexist in harmony on a single machine.
4. **Assemblies Are Self-Describing :** Assemblies are regarded as *self-describing* in part because they record every external assembly it must have access to in order to function correctly. Thus, if your assembly requires System.Windows.Forms.dll and System.Drawing.dll, they will be documented in the assembly's *manifest*. In addition to manifest data, an assembly contains metadata that describes the composition (member names, implemented interfaces, base classes, constructors and so forth) of every contained type.
5. **Assemblies Are Configurable :** Assemblies can be deployed as "private" or "shared." Private assemblies reside in the same directory (or possibly a subdirectory) as the client application making use of them. Shared assemblies, on the other hand, are libraries intended to be consumed by numerous applications on a single machine and are deployed to a specific directory termed the *Global Assembly Cache* (*GAC*).
6. **Assemblies define a security Context :** An assembly may also contain security details. The security constraints defined by an assembly are explicitly listed within its manifest.
7. **Assemblies Enable Side-by-Side Execution :** The biggest advantage of .NET assembly is the ability to install & load multiple versions of the same assembly on a single machine. It is possible to control which version of a assembly should be loaded using application configuration files.

8.5 Building and Consuming a Single-File Assembly :

- To begin the process of comprehending the world of .NET assemblies, we'll first create a single-file *.dll assembly (named CarLibrary) that contains a small set of public types.
- The design of your automobile library begins with an abstract base class named Car that defines a number of protected data members exposed through custom properties. This class has a single abstract method named TurboBoost(), which makes use of a custom enumeration (EngineState) representing the current condition of the car's engine:

```
using System;
namespace CarLibrary
{
    // Represents the state of the engine.
    public enum EngineState
    { engineAlive, engineDead }
    // The abstract base class in the hierarchy.
    public abstract class Car
    {
        protected string petName;
        protected short currSpeed;
        protected short maxSpeed;
        protected EngineState egnState = EngineState.engineAlive;
        public abstract void TurboBoost();
        public Car(){ }
        public Car(string name, short max, short curr)
        {
        petName = name; maxSpeed = max; currSpeed = curr;
        }
        public string PetName
        {
        get { return petName; }
        set { petName = value; }
        }
        public short CurrSpeed
        {
        get { return currSpeed; }
        set { currSpeed = value; }
        }
        public short MaxSpeed
        { get { return maxSpeed; } }
        public EngineState EngineState
        { get { return egnState; } }
    }
}
```

- Now assume that you have two direct descendents of the Car type named MiniVan and SportsCar.
- Each overrides the abstract TurboBoost() method in an appropriate manner.

```
using System;
using System.Windows.Forms;
namespace CarLibrary
{
public class SportsCar : Car
{
        public SportsCar(){ }
        public SportsCar(string name, short max, short curr) : base (name, max, curr){ }
        public override void TurboBoost()
        {
        MessageBox.Show("Ramming speed!", "Faster is better...");
        }
}
public class MiniVan : Car
{
        public MiniVan(){ }
        public MiniVan(string name, short max, short curr) : base (name, max, curr){ }
        public override void TurboBoost()
        {
        // Minivans have poor turbo capabilities!
        egnState = EngineState.engineDead;
        MessageBox.Show("Time to call AAA", "Your car is dead");
        }
}
}
```

8.6 Building a C# Client Application :

- Because each of the CarLibrary types has been declared using the public keyword, other assemblies are able to make use of them.
- To consume these types, create a new C# console application project (CsharpCarClient).
- At this point you can build your client application to make use of the external types. Update your initial C# file as so:

```
using System;
// Don't forget to 'use' the CarLibrary namespace!
using CarLibrary;
namespace CSharpCarClient
{
public class CarClient
{
```

```
        static void Main(string[] args)
        {
        // Make a sports car.
        SportsCar viper = new SportsCar("Viper", 240, 40);
        viper.TurboBoost();
        // Make a minivan.
        MiniVan mv = new MiniVan();
        mv.TurboBoost();
        Console.ReadLine();
        }
    }
    }
```

8.7 Building a Visual Basic .NET Client Application :

- To illustrate the language-agnostic attitude of the .NET platform, let's create another console application (VbNetCarClient), this time using Visual Basic .NET.
- Like C#, Visual Basic .NET requires you to list each namespace used within the current file.
- However, Visual Basic .NET offers the Imports keyword rather than the C# using keyword.

```
Imports CarLibrary
Module Module1
        Sub Main( )
        Console.WriteLine("***** Fun with Visual Basic .NET *****")
        Dim myMiniVan As New MiniVan()
        myMiniVan.TurboBoost()
        Dim mySportsCar As New SportsCar()
        mySportsCar.TurboBoost()
        Console.ReadLine()
        End Sub
End Module
```

8.8 Cross-Language Inheritance:

- A very enticing aspect of .NET development is the notion of cross-language inheritance.
- To illustrate, let's create a new Visual Basic .NET class that derives from SportsCar (which was authored using C#).
- First, add a new class file to your current Visual Basic .NET application named PerformanceCar.vb.
- Update the initial class definition by deriving from the SportsCar type using the Inherits keyword.
- Furthermore, override the abstract TurboBoost() method using the Overrides keyword:

```
// Code is in C#
using System;
using System.Windows.Forms;
namespace CarLibrary
{
public class SportsCar : Car
{
public SportsCar(){ }
public SportsCar(string name, short max, short curr) : base (name, max, curr){ }
public override void TurboBoost()
{
MessageBox.Show("Ramming speed!", "Faster is better...");
}
}
```

```
// Code in VB.NET
Imports CarLibrary
' This VB type is deriving from the C# SportsCar.
Public Class PerformanceCar
Inherits SportsCar
Public Overrides Sub TurboBoost()
Console.WriteLine("Zero to 60 in a cool 4.8 seconds...")
End Sub
End Class
```

To test this new class type, update the Module's Main() method as so:

```
Sub Main()
...
Dim dreamCar As New PerformanceCar()
' Inherited property.
dreamCar.PetName = "Hank"
dreamCar.TurboBoost()
Console.ReadLine()
End Sub
```

8.9 Building and Consuming a Multifile Assembly:

- A multifile assembly is simply a collection of related modules that is deployed and versioned as a single unit.
- To illustrate the process, you will build a multifile assembly named AirVehicles.
- The primary module (airvehicles.dll) will contain a single class type named Helicopter.

- The related manifest (also contained in airvehicles.dll) catalogues an additional *.netmodule file named ufo.netmodule, which contains another class type named (of course) Ufo.
- Although both class types are physically contained in separate binaries, you will group them into a single namespace named AirVehicles.
- Finally, both classes are created using C#.
- To begin, open a simple text editor (such as Notepad) and create the following Ufo class definition saved to a file named ufo.cs:

```
using System;
namespace AirVehicles
{
public class Ufo
{
public void AbductHuman()
{
Console.WriteLine("Resistance is futile");
}
}
}
```

- To compile this class into a .NET module, navigate to the folder containing ufo.cs and issue the following command to the C# compiler :

  csc.exe /t:module ufo.cs

- Next, create a new file named helicopter.cs that contains the following class definition:

```
using System;
namespace AirVehicles
{
public class Helicopter
{
public void TakeOff()
{
Console.WriteLine("Helicopter taking off!");
}
}
}
```

- Given that airvehicles.dll is the intended name of the primary module of this multifile assembly, you will need to compile helicopter.cs using the /t:library and /out: options. The following command does the trick:

  csc /t:library /addmodule:ufo.netmodule /out:airvehicles.dll helicopter.cs

- At this point, your directory should contain the primary airvehicles.dll module as well as the secondary ufo.netmodule binaries.

8.10 Consuming a Multifile Assembly:

- The consumers of a multifile assembly couldn't care less that the assembly they are referencing is composed of numerous modules.
- Let's create a new Visual Basic .NET client application at the command line.
- Create a new file named Client.vb with the following Module definition.
- When you are done, save it in the same location as your multifile assembly.

```
Imports AirVehicles
Module Module1
Sub Main()
Dim h As New AirVehicles.Helicopter()
h.TakeOff()
' This will load the *.netmodule on demand.
Dim u As New UFO()
u.AbductHuman()
End Sub
End Module
```

- To compile this executable assembly at the command line, you will make use of the Visual Basic.NET command-line compiler, vbc.exe, with the following command set:

vbc /r:airvehicles.dll *.vb

8.11 Understanding Private Assemblies :

- Private Assemblies are a collection of modules that is only used by the application with which it has been deployed.
- These are required to be located within the same directory as the client application (termed the application directory) or a subdirectory thereof.
- When a client program uses the types defined within this external assembly, the CLR simply loads the local copy of CarLibrary.dll.

8.11.1 The Identity of a Private Assembly :

- The full identity of a private assembly consists of the friendly name and numerical version, both of which are recorded in the assembly manifest.
- The friendly name simply is the name of the module that contains the assembly's manifest minus the file extension.
- For example, if you examine the manifest of the CarLibrary.dll assembly, you find the following (your version will no doubt differ):

```
.assembly CarLibrary
{
...
.ver 1:0:454:30104
}
```

8.11.2 Understanding the Probing Process :

- The .NET runtime resolves the location of a private assembly using a technique termed probing.
- Probing is the process of mapping an external assembly request to the location of the requested binary file.
- Strictly speaking, a request to load an assembly may be either implicit or explicit.
- An implicit load request occurs when the CLR consults the manifest in order to resolve the location of an assembly defined using the .assembly extern tokens:

```
// An implicit load request.
.assembly extern CarLibrary
{ ...}
```

- An explicit load request occurs programmatically using the Load() or LoadFrom() method of the System.Reflection.Assembly class type, typically for the purposes of late binding and dynamic invocation of type members.
- An example of an explicit load request in the following code:

```
// An explicit load request.
Assembly asm = Assembly.Load("CarLibrary");
```

- In either case, the CLR extracts the friendly name of the assembly and begins probing the client's application directory for a file named CarLibrary.dll.
- If neither of these files can be located in the application directory, the runtime gives up and throws a FileNotFound exception at runtime.

8.12 Configuring Private Assemblies :

- While it is possible to deploy a .NET application by simply copying all required assemblies to a single folder on the user's hard drive, you will most likely wish to define a number of subdirectories to group related content.
- To illustrate the process, create a new directory on your C drive named MyApp using Windows Explorer.
- Next, copy CSharpCarClient.exe and CarLibrary.dll to this new folder, and run the program by double-clicking the executable.
- Your program should run successfully at this point.
- Next, create a new subdirectory under C:\MyApp named MyLibraries and move CarLibrary.dll to this location.
- Create a new configuration file named CsharpCarClient.exe.config and save it in the *same* folder containing the CSharpCarClient.exe application, which in this example would be C:\MyApp. Open this file and enter the following content exactly as shown (be aware that XML is case sensitive!):

```
<configuration>
<runtime>
```

```
<assemblyBinding xmlns="urn:schemas-microsoft-com:asm.v1">
<probing privatePath="MyLibraries"/>
</assemblyBinding>
</runtime>
</configuration>
```

- The <probing> element simply instructs the CLR to investigate all specified subdirectories for the requested assembly until the first match is encountered.
- Once you've finished creating CSharpCarClient.exe.config, run the client by double-clicking the executable in Windows Explorer

8.13 Understanding Shared Assemblies :

- A shared assembly is a collection of types and (optional) resources contained within some number of modules.
- These can be used by several clients on a single machine.
- The shared assemblies are installed into the machine-wide Global Assembly Cache (GAC).
- The GAC is located under a subdirectory of your Windows directory named Assembly (e.g., C:\Windows\Assembly).

8.13.1 Understanding Strong Names :

- Before you can deploy an assembly to the GAC, you must assign it a *strong name*, which is used to uniquely identify the publisher of a given .NET binary.
- A strong name is composed of a set of related data, much of which is specified using assembly-level attributes:
  - The friendly name of the assembly
  - The version number of the assembly (assigned using the [AssemblyVersion] attribute)
  - The public key value (assigned using the [AssemblyKeyFile] attribute)
  - An optional culture identity value for localization purposes (assigned using the [AssemblyCulture] attribute)
  - An embedded *digital signature* created using a hash of the assembly's contents and the private key value
- To provide a strong name for an assembly, your first step is to generate public/private key data using the .NET Framework 2.0 SDK's sn.exe utility. The sn.exe utility responds by generating a file (typically ending with the *.snk [Strong Name Key] file extension) that contains data for two distinct but mathematically related keys, the "public" key and the "private" key.

8.14 Building a Shared Assembly:

- To illustrate the process of assigning a strong name to the an assembly, let's walk through a complete example.
- Assume you have created a new C# class library named SharedAssembly, which contains the following class definition:

public class VWMiniVan

```
{
        private bool isBust = false;
        public VWMiniVan( )
        {
                MessageBox.Show("Using version 1.0.0.0!", "Shared car");
        }
        public void Play60sTunes( )
        {
                MessageBox.Show("What a long, strange trip it's been.....");
        }
        public bool Busted
        {
                get{ return isBust;}
                set{isBust = Value;}
        }
}
```

- To generate the key file, you need to make use of the sn.exe utility.
- Although this tool has numerous command-line options, all you need to concern yourself with for the moment is the -k flag, which instructs the tool to generate a new file containing the public/private key information.
- Now, issue the following command to generate a file named MyTestKeyPair.snk:

  sn -k MyTestKeyPair.snk
- The next step is to inform the C# compiler exactly where MyTestKeyPair.snk is located.
- The AssemblyKeyFile assembly-level attribute can be used to inform the compiler of the location of a valid *.snk file.
- Simply specify the path as a string parameter, for example:

  [assembly: AssemblyKeyFile(@"C:\MyTestKeyPair\MyTestKeyPair.snk")]
- Next is to specify a specific version number for an assembly. In the AssemblyInfo.cs file, you will find another attribute named AssemblyVersion. Initially the value is set to 1.0.*:

  [assembly: AssemblyVersion("1.0.0.0")]

8.14.1 The Role of Delayed Signing :

- When you are building your own custom .NET assemblies, you are able to assign a strong name using your own personal *.snk file.
- Delayed signing begins by the trusted individual holding the *.snk file extracting the public key value from the public/private *.snk file using the -p command-line flag of sn.exe, to produce a new file that only contains the public key value:

  sn -p myKey.snk testPublicKey.snk
- To inform the C# compiler that the assembly in question is making use of delayed signing, the developer must make sure to set the value of the AssemblyDelaySign attribute to true in addition to specifying the pseudo-key file as the parameter to the AssemblyKeyFile attribute. Here are the relevant updates to the project's AssemblyInfo.cs file:

  [assembly: **AssemblyDelaySign**(true)]

  [assembly: AssemblyKeyFile(@"C:\MyKey**testPublicKey.snk**)]

- Once an assembly has enabled delayed signing, the next step is to disable the signature verification process that happens automatically when an assembly is deployed to the GAC.
- To do so, specify the -vr flag (using sn.exe) to skip the verification process on the current machine:

  sn.exe -vr MyAssembly.dll
- Once all testing has been performed, the assembly in question can be shipped to the trusted individual who holds the "true" public/private key file to resign the binary to provide the correct digital signature.
- Again, sn.exe provides the necessary behavior, this time using the -r flag:

  sn.exe -r MyAssembly.dll C:\MyKey\myKey.snk
- To enable the signature verification process, the final step is to apply the -vu flag:

  sn.exe -vu MyAssembly.dll

8.14.2 Using or Consuming a Shared Assembly:

- Create a new C# console application named SharedAsmClient and exercise your types as you wish:

```
public class SharedAsmClient
{
        static void Main(string[ ] args)
        {
        VWMiniVan v = new VWMiniVan( );
        v.Play60sTunes( );
        Console.ReadLine( );
        }
}
```

- Once you have compiled your client application, navigate to the directory that contains SharedAsmClient.exe using Windows Explorer and notice that Visual Studio 2005 has *not* copied CarLibrary.dll to the client's application directory.

8.15 Key Elements & CIL Tokens for Assembly Manifest :

8.15.1 Key Elements of Assembly Manifest :

Manifest-Centric Information	Meaning
Assembly Name	A text string specifying the assembly's name.
Version Number	A major & minor version number and a revision & build number.
Strong Name information	In part, the strong name of an assembly consists of a public key maintained by the publisher of the assembly.
List of all modules in the assembly	A hash of each module contained in the assembly.
Information on Referenced Assemblies	A list of other assemblies that are statically referenced by the assembly.

8.15.2 CIL Tokens of Assembly Manifest:

Manifest Tag	Meaning
.assembly	Marks the assembly declaration, indicating that the file is an assembly.
.file	Marks additional files in a multifile assembly.
.class extern	Classes exported by the assembly but declared in another module.
.manifestres	Indicates the manifest resources.
.module	Module declaration, indicating that the file is a module & not the primary assembly.
.assembly extern	The assembly reference indicates another assembly containing items referenced by this module.
.publickey	Contains the actual bytes of the public key.
.publickeytoken	Contains a token of the actual public key.