



C# Programming in Depth

Prof. Dr. Bertrand Meyer

March 2007 - May 2007

Lecture 7: .NET Assemblies, Type Reflection
and Attribute-Based Programming

Lisa (Ling) Liu

How to create and deploy a .NET assembly library?



- Build a Vehicle library, which includes type Car, SportsCar and MiniVan, UFO and Helicopter as one binary file or several binary files?
- Application-wide or machine-wide deployment?
- Deploy the library on local machine or web?

How to build extendable applications?



- Provide some input vehicle to allow the user to specify the module to plug in
- Determine if the module supports the correct functionality
- Obtain a reference to the required infrastructure and invoke the members to trigger the underlying functionality

Overview



- Single-file and mutifile assemblies
- Private and shared assemblies
- Assembly configuration
- Type reflection and late binding
- Attribute-based programming

Assembly's definition



- .NET applications are constructed by piecing together any number of assemblies.
- An assembly (*.exe or *.dll) is a versioned, self-describing binary file hosted by the CLR.
- An assembly can be composed of multiple modules.
- A module is a generic term for a valid .NET binary file.

The role of .NET assemblies



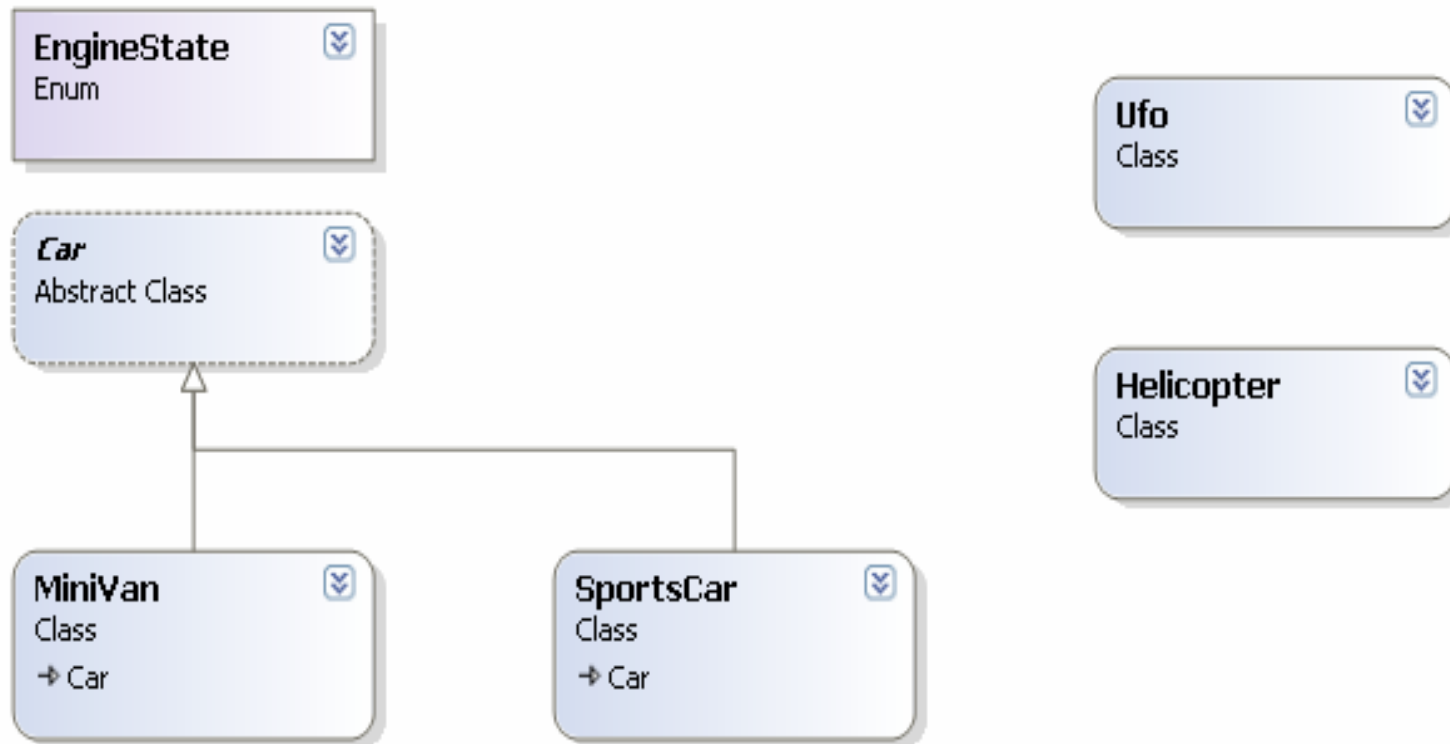
- Promote code reuse
 - reuse types in a language-independent manner
- Establish a type boundary
- Form versionable units
- Provide type and composition information (self-describing)
- Facilitate deployment through configuration files (configurable)

Format of a .NET assembly



- A Win32 file header
- A CLR file header
- CIL code
- Type metadata
- An assembly manifest
- Optional embedded resources

Class diagram of the Vehicle.dll assembly



Some questions you may ask



- Do you want to use the same programming language to implement the library?
- Do you want to allow client applications to download the part of the library on demand?
- Do you want to build the assembly as one binary file or multiple binary files?

answer:

- same language, no, one binary file => Single file assembly
- different languages, yes, multiple binary files => Multifile assembly

Single-file and multifile assemblies



- A single file assembly is exactly composed of a single module
 - Contains all of the necessary elements in a single *.exe or *.dll
- A multifile assembly is a set of .NET *.dlls that are deployed and versioned as a single logic unit
 - One of these *.dlls is termed the primary module and contains the assembly-level manifest.

Single-file assemble



Manifest
Type Metadata
CIL Code
(Optional) Resources

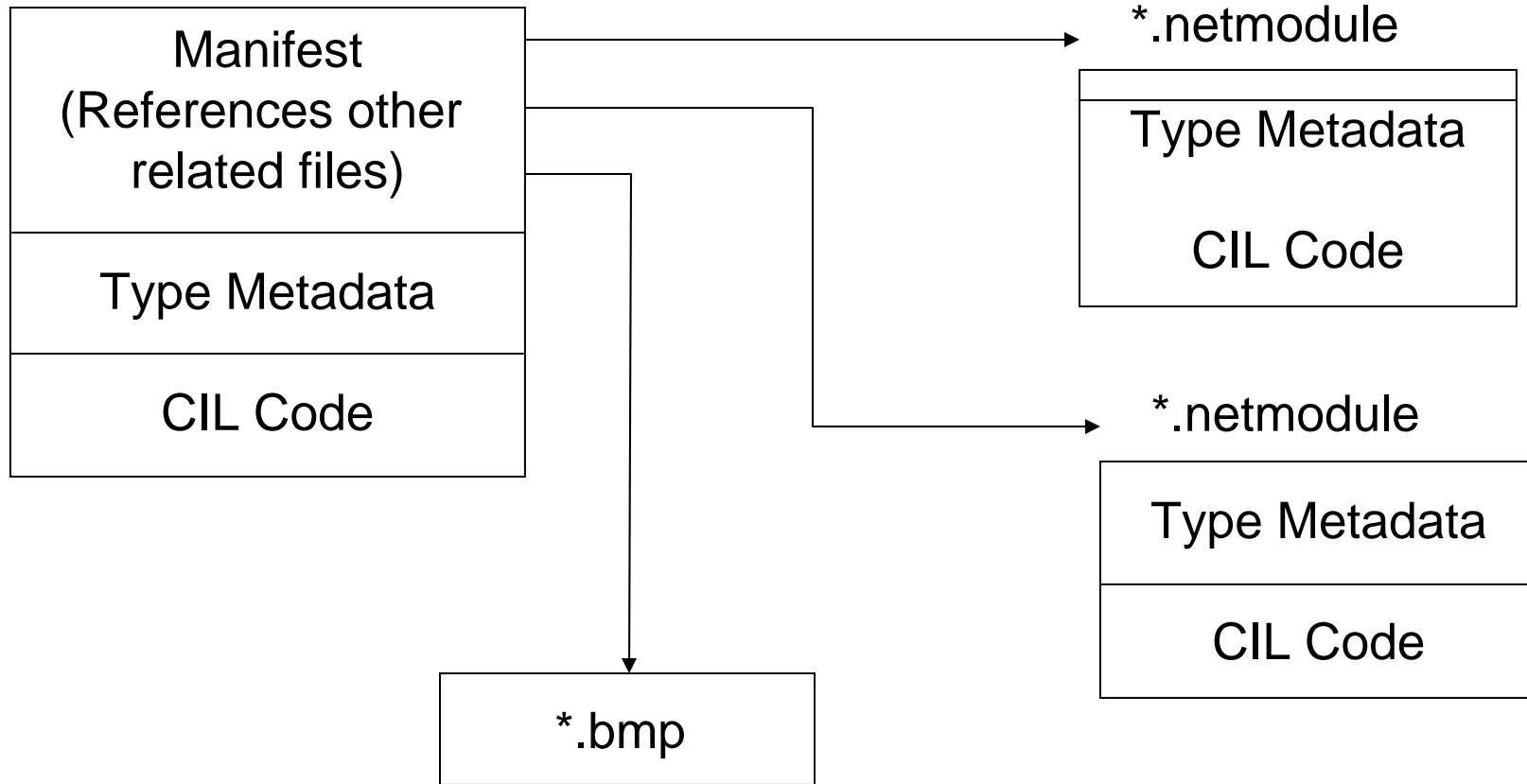
Building and consuming a single-file assembly

- You can use command-line compilers or Visual Studio 2005 to create a single-file assembly. By default, the compiler creates an assembly file with an .exe extension
- Visual studio automatically places a copy of the library assembly into the directory of the client application

Multifile assembly



Primary module (*.dll)



Building and consuming a mutifile assembly



- Build a mutifile assembly
 1. Create secondary modules (*.netmodules)
`csc.exe /t:module *.cs`
 2. Create primary module (*.dll)
`csc.exe /t:library /addmodule:*.netmodule /out:*.dll
*.cs`
- Consume a mutifile assemble
 - Supply onle the primary module to the compiler
`csc /r: primary.dll client.cs`

Benefits of multifile assemblies



- Provide a very efficient way to download content
- Enable modules to be authored using multiple .NET programming languages

Deploy .NET assemblies



- Private assemblies (Xcopy deployment)
 - Private assemblies are required to be located in the same directory as the client application
 - Used to create an application-wide class library

- Shared Assemblies
 - A single copy of a shared assembly can be used by several applications on a single machine
 - Used to create machine-wide class library

Understanding private assemblies



- Install a private assembly
 - The identity of a private assembly
 - The probing process
 - Configuration of private assemblies

- Uninstall a private assembly
 - Delete the application folder

Identify a private assembly



- The full identity of a private assembly is composed of
 - The friendly name of the assembly
 - The numerical version of the assembly
- The assembly manifest records the identity of the private assembly

```
.assembly Vehicle
{
    ...
    .ver 1:0:0:0}
```

Probe a private assemble



- Probing is the process of mapping an external assembly request to the location of the requested binary file
- Implicit probing
- Explicit probing

Configure private assemblies



*.exe.config

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <runtime>
    <assemblyBinding xmlns="urn:schemas-microsoft-com:asm.v1">
      <probing privatePath="lib"/>
    </assemblyBinding>
  </runtime>
</configuration>
```

- Instruct CLR to probe the subdirectories within client application directory
- Has the same name as the launching application and take a *.config file extension
- Must be deployed in the application directory
- Use the `privatePath` attribute to set the probing subdirectory

Understanding shared assemblies



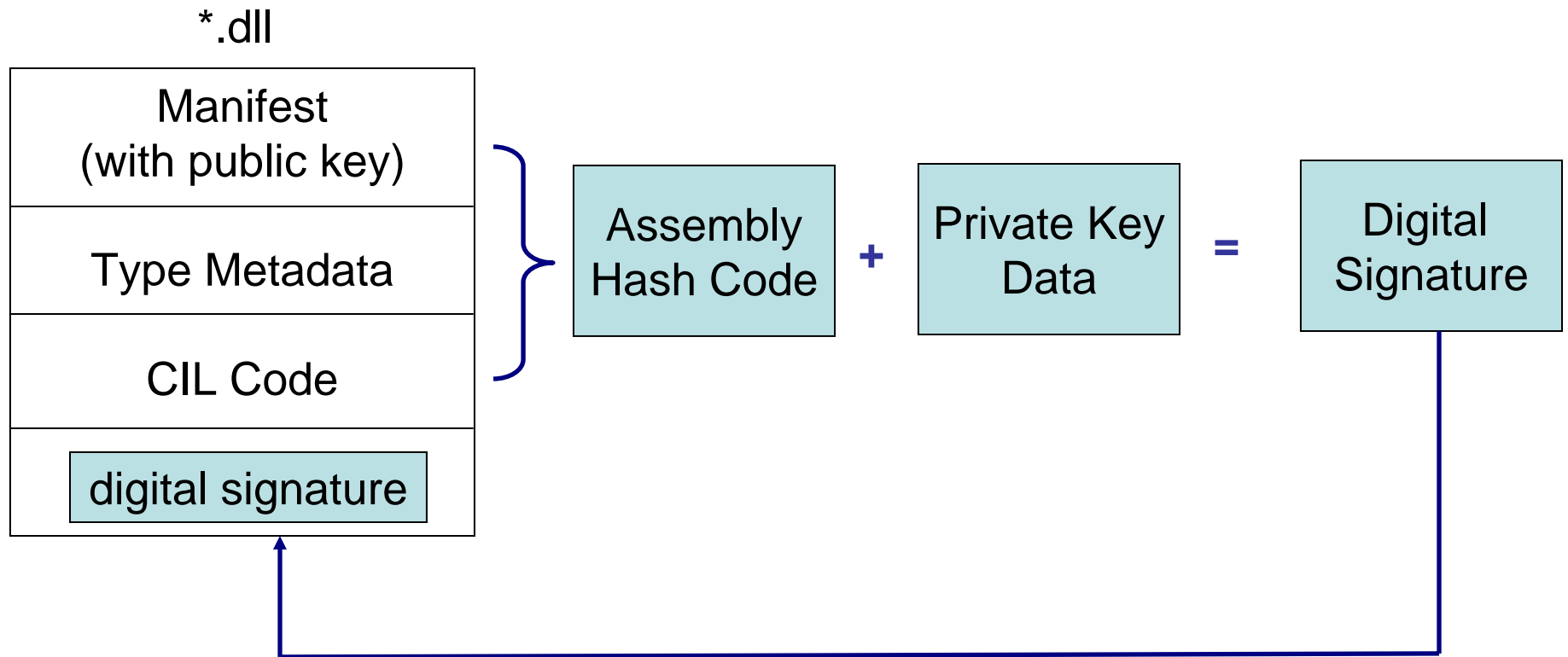
- Install a shared assembly
 - Strongly name a shared assembly
 - Install the shared assembly into the Global Assembly Cache (GAC)
- Uninstall a shared assembly
 - Use command-line utility `gacutil.exe` to uninstall a shared assembly

Strong name



- A strong name is a unique identifier of an assembly
- A strong name is based on two cryptographically related keys (public key and private key).
- A strong name consists of following data:
 - 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 identifier value for localization purpose (assigned using the [AssemblyCulture] attribute)
 - An embedded digital signature created using a hash of the assembly's contents and the private key value

Generate digital signature at compile-time



Strongly naming an assembly



1. Create the required key data using `sn.exe`
 - `sn -k MyTestKey.snk`
2. Inform the C# compiler where the `MyTestKey.snk` is located
 - `[Assembly: AssemblyKeyFile(@"C:\MyTestKey\MyTestKey.snk")]`
3. Specify the version of a shared assembly
 - A .NET version is composed of the four parts:
<major>.<minor>.<build>.<version>
 - `[Assembly: AssemblyVersion("1.0.*")]`

Install a shared assembly to the GAC



- Global Assembly Cache (GAC) is located under the **Assembly** subdirectory (C:\Windows\Assembly) under your windows directory
- Install a shared assembly to the GAC by dragging or dropping the assembly to C:\Windows\Assembly

gacutil.exe



- /i Install a strongly name assembly into the *GAC*
- /u Uninstall an assembly from the *GAC*
- /l Displays the assemblies (or a specific assembly) in the *GAC*

Delayed signing



Install an assembly into the *GAC* for testing purpose

- The administrator or some trusted individual that holds the *.snk file extracts the public key value from the *.snk file

```
sn.exe -p myKey.snk testPublicKey.snk
```

- Distribute the extracted testPublicKey.snk to developers
- The developer specifies the delayed signing attribute in the assembly

```
[Assembly: AssemblyDelaySign(true)]
```

```
[Assembly:
```

```
AssemblyKeyFile(@"C:\MyKey\testPublicKey.snk")]
```

- Disable the signature verification process when deploying an assembly to *GAC*

```
sn.exe -Vr MyAssembly.dll
```

After delayed signing ...



- Ship the assembly to the trusted individual who holds the "true" public/private key file
- The trusted individual Resigns the assembly to provide the correct digital signature
`sn.exe -R MyAssembly.dll C:\MyKey\myKey.snk`
- Enable the signature verification procedure
`sn.exe -Vu MyAssembly.dll`

Consuming a shared assembly



- In Visual Studio 2005, you must reference shared assemblies by navigating to the project's `\Bin\Debug` directory
- The **Copy Local** property can force a copy of a strongly named code library

Configure shared assemblies



- Build `ClientApplication.exe.config` file to instruct CLR to bind to the assembly with required version number
- Set `bindingRedirect` attribute of dependent `Assembly` element to the required version number
- Dynamic updating library

Publisher policy assemblies



- With **Publisher policy** assemblies, client applications need not to set specific *.config files
- CLR performs the request redirection at the level of *GAC*
- Create publisher policy assemblies using **al.exe** command
 - `al.exe /link: CarLibraryPolicy.xml`
 - `/out:policy.1.0.CarLibrary.dll`
 - `/keyf:C:\MyKey\MyKey.snk /v:1.0.0.0`
- Disable publisher policy
 - Build specific client application configuration file and set the apply attribute of element publisherPolicy as "no"

<codeBase> element



- The <codeBase> element is used to instruct the CLR to probe for dependent assemblies located at arbitrary locations
- It can be used to download assemblies from a given URL

The System.Configuration namespace



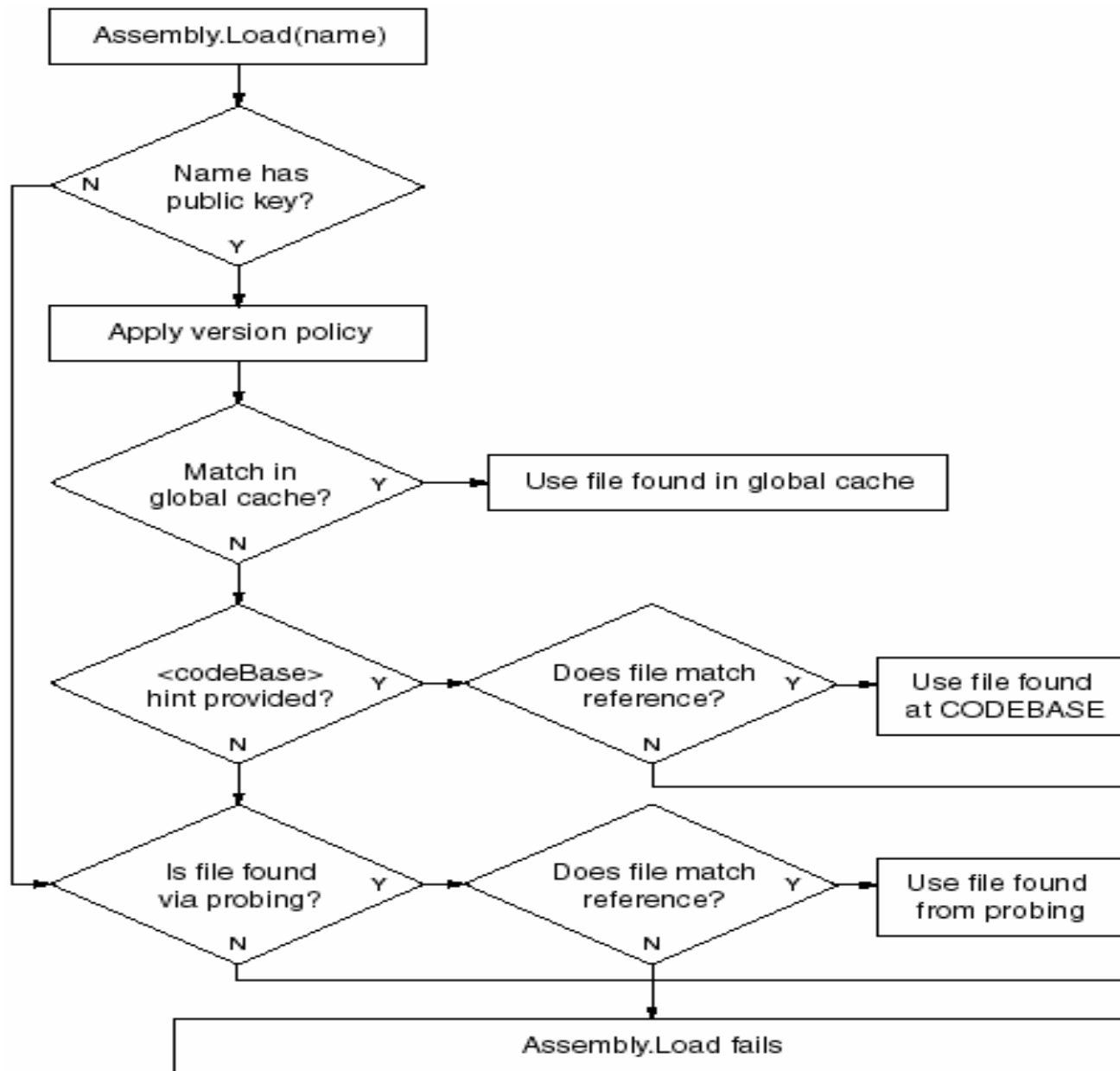
- The System.Configuration namespace allows programmers to programmatically read the data within a client configuration file

The machine configuration file



machine.config file

- It contains machinewide application settings (via an `<appsetting>` element)
- .NET platform maintains a separate *.config file for each version of the framework installed on the local machine



How to build extendable applications?



- Provide some input vehicle to allow the user to specify the module to plug in (Dynamic loading)
- Determine if the module supports the correct functionality (Type reflection)
- Obtain a reference to the required infrastructure and invoke the members to trigger the underlying functionality (Late binding)

Type reflection



- In the .NET universe, reflection is the process of runtime type discovery
- System.Reflection namespace
 - Assembly, AssemblyName, EventInfo, FieldInfo, MethodInfo, ParameterInfo and PropertyInfo
- System.Type class
 - GetConstructors(), GetEvents(), GetFields(), GetInterfaces(), GetMembers(), GetMethods(), GetProperties(), InvokeMember()



Reflect the type information

- Obtain a type reference using `System.Type.GetType()`
`Type t = Type.GetType (typeName);`
- Reflecting on methods
`MethodInfo[] mi = t.GetMethods();`
- Reflecting on Fields and properties
`FieldInfo[] fi = t.GetFields();`
`PropertyInfo[] pi = t.GetProperties();`

How to reflect the type defined in an assembly not known at compile time?

Dynamically loading assemblies



- Dynamic load
 - The act of loading external assemblies on demand
- Use methods `Load()` or `LoadFrom()` defined in class `Assembly` to dynamically load an assembly (private or shared assembly)

Reflect on shared assemblies



- To load a shared assembly from the *GAC*, the `Assembly.Load()` must specify a `publickeytoken` value
- Two ways of specifying a `publickeytoken` when loading a shared assembly

```
Assembly a = Assembly.Load(@"System.Windows.Forms,  
    PublicKeyToken=b77a5c561934e089")
```

```
AssemblyName asmName;  
asmName = new AssemblyName();  
asmName.SetPublicKeyToken(byte[]);
```

Late binding



- Late binding is a technique in which you are able to create an instance of a given type and invoke its members at runtime without having compile-time knowledge of its existence

```
Assembly a = Assembly.Load("CarLibrary");  
Type miniVan = a.GetType("CarLibrary.MiniVan");  
object obj = Activator.CreateInstance(miniVan);  
MethodInfo mi = miniVan.GetMethod("TurboBoost");  
mi.Invoke(obj, null);
```

Attributed programming



- Using attributes, programmers can embed additional metadata into an assembly
- .NET attributes are class types that extend the abstract `System.Attribute` base class
- Attribute consumers
 - C# compiler
 - Numerous methods in the .NET base class libraries, for example, `Serialize()`
- An attribute is only applied to the “very next” item

Predefined attributes in C#



[CLSCompliant]

[DllImport]

[Obsolete]

[Serializable]

[NonSerialized]

[WebMethod]

Building custom attributes



```
public sealed class VehicleDescriptionAttribute: System.Attribute
{
    private string msgData;
    public VehicleDescriptionAttribute (string description)
    {msgData = description;}
    public vehicleDescriptionAttribute() {}
    public string Description
    {
        get {return msgData;}
        set {msgData = value;}
    }
}
```

Assembly-level attributes



- Apply attributes on all types within a given module or all modules within a given assembly using the [module:] and [assembly:]
[assembly: System.CLSCompliantAttribute(true)]
- Visual studio 2005 AssemblyInfo.cs file
A handy place to put attributes that are to be applied at the assembly level

Questions?

