2 Components

THE CLR HAS its own set of concepts and techniques for packaging, deploying, and discovering component code. These concepts and techniques are fundamentally different from those used by technologies such as COM, Java, or Win32. The difference is best understood by looking closely at the CLR loader, but first one must look at how code and metadata are actually packaged.

Modules Defined

Programs written for the CLR reside in **modules**. A CLR module is a byte stream, typically stored as a file in the local file system or on a Web server.

As shown in Figure 2.1, a CLR module uses an extended version of the PE/COFF executable file format used by Windows NT. By extending the PE/COFF format rather than starting from scratch, CLR modules are also valid Win32 modules that can be loaded using the `LoadLibrary` system call. However, a CLR module uses very little PE/COFF functionality. Rather, the majority of a CLR module's contents are stored as opaque data in the `.text` section of the PE/COFF file.

CLR modules contain code, metadata, and resources. The code is typically stored in common intermediate language (CIL) format, although it may also be stored as processor-specific machine instructions. The module's metadata describes the types defined in the module, including names, inheritance relationships, method signatures, and dependency

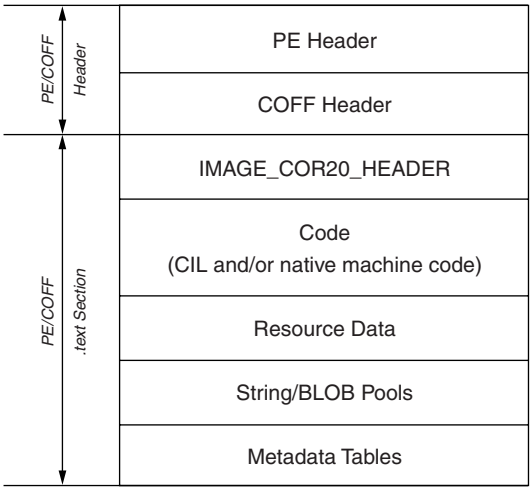

Figure 2.1: *CLR Module Format*

information. The module's resources consist of static read-only data such as strings, bitmaps, and other aspects of the program that are not stored as executable code.

The file format used by CLR modules is fairly well documented; however, few developers will ever encounter the format in the raw. Even developers who need to generate programs on-the-fly will typically use one of the two facilities provided by the CLR for programmatically generating modules. The `IMetaDataEmit` interface is a low-level COM interface that can be used to generate module metadata programmatically from classic C++. The `System.Reflection.Emit` namespace is a higher-level library that can be used to generate metadata and CIL programmatically from any CLR-friendly language (e.g., C#, VB.NET). The CodeDOM works at an even higher layer of abstraction, removing the need to know or understand CIL. However, for the vast majority of developers, who simply need to generate code during development and not at runtime, a CLR-friendly compiler will suffice.

The C# compiler (`CSC.EXE`), the VB.NET compiler (`VBC.EXE`), and the C++ compiler (`CL.EXE`) all translate source code into CLR modules. Each of the compilers uses command-line switches to control which kind of module to produce. As shown in Table 2.1, there are four possible options. In C# and VB.NET, one uses the `/target` command-line switch (or its

TABLE 2.1 Module Output Options

C#/VB.NET	C++	Directly Loadable?	Runnable from Shell?	Access to Console?
`/t:exe`	`/CLR`	Yes	Yes	Always
`/t:winexe`	`/CLR /link /subsystem:windows`	Yes	Yes	Never
`/t:library`	`/CLR /LD`	Yes	No	Host-dependent
`/t:module`	`/CLR:NOASSEMBLY /LD`	No	No	Host-dependent

shortcut, `/t`) to select which option to use. The C++ compiler uses a combination of several switches; however, one always uses the `/CLR` switch to force the C++ compiler to generate CLR-compliant modules. The remainder of this discussion will refer to the C# and VB.NET switches, given their somewhat simpler format.

The `/t:module` option produces a "raw" module that by default will use the `.netmodule` file extension. Modules in this format cannot be deployed by themselves as stand-alone code, nor can the CLR load them directly. Rather, developers must associate raw modules with a full-fledged component (called an assembly) prior to deployment. In contrast, compiling with the `/t:library` option produces a module that contains additional metadata that allows developers to deploy it as stand-alone code. A module produced by compiling with `/t:library` will have a `.DLL` file extension by default.

Modules compiled with `/t:library` can be loaded directly by the CLR but cannot be launched as an executable program from a command shell or the Windows Explorer. To produce this kind of module, you must compile using either the `/t:exe` or the `/t:winexe` option. Both options produce a file whose extension is `.EXE`. The only difference between these two options is that the former assumes the use of the console UI subsystem; the latter option assumes the GUI subsystem. If no `/t` option is specified, the default is `/t:exe`.

Modules produced using either the `/t:exe` or the `/t:winexe` option must have an **initial entry point** defined. The initial entry point is the method that the CLR will execute automatically when the program is launched. Programmers must declare this method `static`, and, in C# or VB.NET, they must name it `Main`. Programmers can declare the entry point method to return no value or to return an `int` as its exit code. They can also declare it to accept no parameters or to accept an array of strings, which will contain the parsed command-line arguments from the shell. The following are four legal implementations for the `Main` method in C#:

```
static void Main() { }
static void Main(string[] argv) { }
static int Main() { return 0; }
static int Main(string[] argv) { return 0; }
```

These correspond to the following in VB.NET:

```
shared sub Main() : end sub
shared sub Main(argv as string()) : end sub
shared function Main() : return 0 : end function
shared function Main(argv as string())
  return 0
end function
```

Note that these methods do not need to be declared `public`. Programmers must, however, declare the `Main` method inside a type definition, although the name of the type is immaterial.

The following is a minimal C# program that does nothing but print the string `Hello, World` to the console:

```
class myapp {
  static void Main() {
    System.Console.WriteLine("Hello, World");
  }
}
```

In this example, there is exactly one class that has a static method called `Main`. It would be ambiguous (and therefore an error) to present the C# or VB.NET compiler with source files containing more than one type having

a static method called `Main`. To resolve this ambiguity, programmers can use the `/main` command-line switch to tell the C# or VB.NET compiler which type to use for the program's initial entry point.

Assemblies Defined

In order to deploy a CLR module, developers must first affiliate it with an **assembly**. An assembly is a logical collection of one or more modules. As just described, modules are physical constructs that exist as byte streams, typically in the file system. Assemblies are logical constructs and are referenced by location-independent names that must be translated to physical paths either in the file system or on the Internet. Those physical paths ultimately point to one or more modules that contain the type definitions, code, and resources that make up the assembly.

The CLR allows developers to compose assemblies from more than one module primarily to support deferred loading of infrequently accessed code without forming separate encapsulation boundaries. This feature is especially useful when developers are using code download because they can download the initial module first and download secondary modules only on an as-needed basis. The ability to build multimodule assemblies also enables mixed-language assemblies. This allows developers to work in a high-productivity language (e.g., Logo.NET) for the majority of their work but to write low-level grunge code in a more flexible language (e.g., C++). By conjoining the two modules into a single assembly, developers reference, deploy, and version the C++ and Logo.NET code as an atomic unit.

> Parenthetically, though an assembly may consist of more than one module, a module is generally affiliated with only one assembly. As a point of interest, if two assemblies happen to reference a common module, the CLR will treat this as if there are two distinct modules, something that results in two distinct copies of every type in the common module. For that reason, the remainder of this chapter assumes that a module is affiliated with exactly one assembly.

Assemblies are the "atom" of deployment in the CLR and are used to package, load, distribute, and version CLR modules. Although an assembly may consist of multiple modules and auxiliary files, the assembly is named and versioned as an atomic unit. If one of the modules in an assembly must be versioned, then the entire assembly must be redeployed because the version number is part of the assembly name and not the underlying module name.

Modules typically rely on types from other assemblies. At the very least, every module relies on the types defined in the `mscorlib` assembly, which is where types such as `System.Object` and `System.String` are defined. Every CLR module contains a list of assembly names that identifies which assemblies are used by this module. These external assembly references use the logical name of the assembly, which contains no remnants of the underlying module names or locations. It is the job of the CLR to convert these logical assembly names into module pathnames at runtime, as is discussed later in this chapter.

To assist the CLR in finding the various pieces of an assembly, every assembly has exactly one module whose metadata contains the **assembly manifest**. The assembly manifest is an additional chunk of CLR metadata that acts as a directory of adjunct files that contain additional type definitions and code. The CLR can directly load modules that contain an assembly manifest. For modules that lack an assembly manifest, the CLR can load them only indirectly, by first loading a module whose assembly manifest refers to the manifest-less module. Figure 2.2 shows two modules: one with an assembly manifest and one without one. Note that of the four `/t` compiler options, only `/t:module` produces a module with no assembly manifest.

Figure 2.3 shows an application that uses a multimodule assembly, and Listing 2.1 shows the MAKEFILE that would produce it. In this example, `code.netmodule` is a module that does not contain an assembly manifest. To make it useful, one needs a second module (in this case, `component.dll`) that provides an assembly manifest that references `code.netmodule` as a subordinate module. One achieves this using the `/addmodule` switch when compiling the containing assembly. After this assembly is produced, all the types defined in `component.dll` and

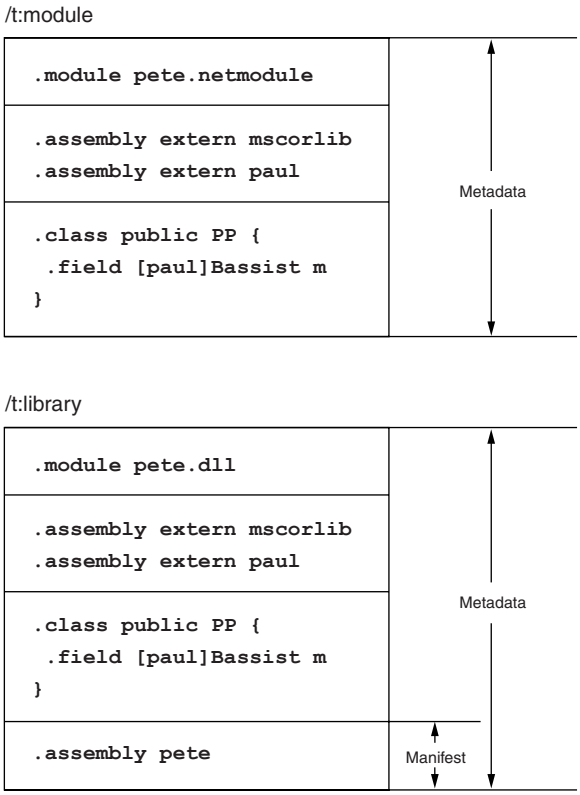

Figure 2.2: *Modules and Assemblies*

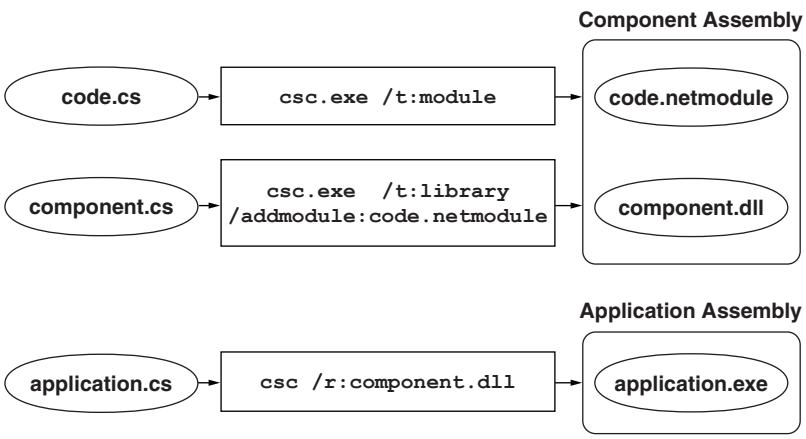

Figure 2.3: *Multimodule Assemblies Using CSC.EXE*

`code.netmodule` are scoped by the name of the assembly (`component`). Programs such as `application.exe` use the `/r` compiler switch to reference the module containing the assembly manifest. This makes the types in both modules available to the referencing program.

Listing 2.1: *Multimodule Assemblies Using CSC.EXE an1d NMAKE*

```
# code.netmodule cannot be loaded as is until an assembly
# is created
code.netmodule : code.cs
  csc /t:module code.cs

# types in component.cs can see internal and public members
# and types defined in code.cs
component.dll : component.cs code.netmodule
  csc /t:library /addmodule:code.netmodule component.cs

# types in application.cs cannot see internal members and
# types defined in code.cs (or component.cs)
application.exe : application.cs component.dll
  csc /t:exe /r:component.dll application.cs
```

The assembly manifest resides in exactly one module and contains all of the information needed to locate types and resources defined as part of the assembly. Figure 2.4 shows a set of modules composed into a single assembly, as well as the `CSC.EXE` switches required to build them. Notice that in this example, the assembly manifest contains a list of file references to the subordinate modules `pete.netmodule` and `george.netmodule`. In addition to these file references, each of the public types in these subordinate modules is listed using the `.class extern` directive, which allows the complete list of public types to be discovered without traversing the metadata for each of the modules in the assembly. Each entry in this list specifies both the file name that contains the type as well as the numeric metadata token that uniquely identifies the type within its module. Finally, the module containing the assembly manifest will contain the master list of externally referenced assemblies. This list consists of the dependencies of every module in the assembly, not just the dependencies of the current module. This allows all of the assembly's dependencies to be discovered by loading a single file.

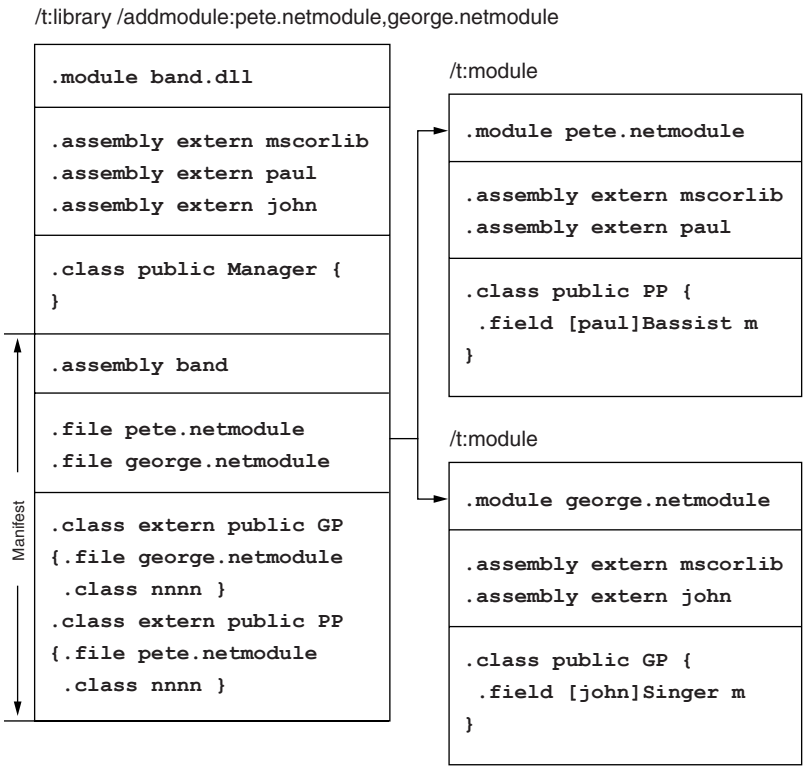

Figure 2.4: *A Multimodule Assembly*

Finally, the module containing the assembly manifest will contain the master list of externally referenced assemblies. This list consists of the dependencies of every module in the assembly not only the dependencies of the current module. This allows all of the assembly's dependencies to be discovered by loading a single file.

Assemblies form an **encapsulation boundary** to protect internal implementation details from interassembly access. Programmers can apply this protection to members of a type (e.g., fields, methods, constructors) or to a type as a whole. Marking a member or type as `internal` causes it to be available only to modules that are part of the same assembly. Marking a type or member as `public` causes it to be available to all code (both inside and outside the current assembly). Individual members of a type (e.g., methods, fields, constructors) can also be marked as `private`, which restricts access to only methods and constructors of the declaring type. This supports classic C++-style programming, in which intracomponent encap-

sulation is desired. In a similar vein, programmers can mark members of a type as `protected`, which broadens the access allowed by `private` to include methods and constructors of derived types. The `protected` and `internal` access modifiers can be combined, something that provides access to types that are either derived from the current type or are in the same assembly as the current type. Table 2.2 shows the language-specific modifiers as they apply both to types and to individual members. Note that members marked `protected internal` in C# require only that the accessor be in the same assembly or in a derived type. The CLR also supports an access modifier that requires the accessor to be both in the same assembly and in a derived type (marked `famandassem` in the metadata). However, VB.NET and C# do not allow programmers to specify this access modifier.

Assemblies scope the type definitions of a component. CLR types are uniquely identified by their assembly name/type name pair. This allows two definitions of the type `Customer` to coexist inside the runtime without ambiguity, provided that each one is affiliated with a different assembly. Although it is possible for multiple assemblies to define the type `Customer` without confusing the runtime, it does not help the programmer who

TABLE 2.2 Access Modifiers

	C#	VB.NET	Meaning
Type	`public`	`Public`	Type is visible everywhere.
	`internal`	`Friend`	Type is visible only inside assembly.
Member	`public`	`Public*`	Member is visible everywhere.
	`internal`	`Friend`	Member is visible only inside assembly.
	`protected`	`Protected`	Member is visible only inside declaring type and its subtypes.
	`protected internal`	`Protected Friend`	Member is visible only inside declaring type and its subtypes or other types inside assembly.
	`private`	`Private*`	Member is visible only inside declaring type.

* VB.NET defaults to `Public` for methods and `Private` for fields declared using the Dim keyword.

wants to use two or more definitions of the same type name in a single program because the symbolic type name is always `Customer` no matter which assembly defines it. To address this limitation of most programming languages, CLR type names can have a **namespace prefix**. This prefix is a string that typically begins with either the organization name of the developer (e.g., `Microsoft`, `AcmeCorp`) or `System` if the type is part of the .NET framework. An emerging convention is to name the assembly based on the namespace prefix. For example, the .NET XML stack is deployed in the `System.Xml` assembly, and all of the contained types use the `System.Xml` namespace prefix. This is simply a convention and not a rule. For example, the type `System.Object` resides in an assembly called `mscorlib` and not in the assembly called `System`, even though there actually is an assembly called `System`.

Assembly Names

Each assembly has a four-part name that uniquely identifies it. This four-part name consists of the friendly name, culture, developer, and version of the component. These names are stored in the assembly manifest of the assembly itself as well as all assemblies that reference it. The CLR uses the four-part assembly name to find the correct component at load time. The CLR provides programmatic access to assembly names via the `System.Reflection.AssemblyName` type, which is easily accessed via the `System.Reflection.Assembly.GetName` method.

The `Name` property of the assembly name typically corresponds to the underlying file name of the assembly manifest sans any file extension that may be in use. This is the only part of the assembly name that is not optional. In simple scenarios, the `Name` property is all that the CLR needs to locate the correct component at load time. When one builds an assembly, this part of the name is automatically selected by your compiler based on the target file name.

All assembly names have a four-part version number (`Version`) of the form `Major.Minor.Build.Revision`. If you do not set this version number explicitly, its default value will be 0.0.0.0. The version number is set at build time, typically using a custom attribute in the source code. The `System.Reflection.AssemblyVersion` attribute accepts a variety of string

TABLE 2.3 Inside the AssemblyVersion Attribute

Attribute Parameter	Actual Value
1	1.0.0.0
1.2	1.2.0.0
1.2.3	1.2.3.0
1.2.3.4	1.2.3.4
1.2.*	1.2.d.s
1.2.3.*	1.2.3.s
‹absent›	0.0.0.0

* Where *d* is the number of days since Feb. 1, 2000, and *s* is the number of seconds since midnight /2

formats, as shown in Table 2.3. When you specify the version number, the `Major` version number is mandatory. Any missing parts are assumed to be zero. At build time, the `Revision` can be specified as `*` (asterisk), and that causes the compiler to use the wall clock to produce a monotonically increasing revision number for each compilation. If an `*` is specified for the `Build` number, the number emitted into the assembly manifest is based on the number of days that have elapsed since February 1, 2000, ensuring that each day has its own unique build number but that a given build number will be applied only for a given 24-hour period. You cannot specify an `*` for the `Major` or `Minor` part of the version number. Later, this chapter discusses how the assembly loader and resolver use the `Version` of the assembly.

Assembly names can contain a `CultureInfo` attribute that identifies the spoken language and country code that the component has been developed for. Developers specify `CultureInfo` using the `System.Reflection.AssemblyCulture` attribute, which accepts a two-part string as specified by Internet Engineering Task Force (IETF) Request for Comments (RFC) 1766. The first part of the string identifies the spoken language using

a two-character lowercase code. The (optional) second part of the string identifies the geographic region using a two-character uppercase code. The string `"en-US"` identifies U.S. English. Assemblies that contain a `CultureInfo` cannot contain code; rather, they must be **resource-only** assemblies (also known as **satellite** assemblies) that can contain only localized strings and other user-interface elements. Satellite assemblies allow a single DLL containing code to selectively load (and download) localized resources based on where they are deployed. Assemblies containing code (that is, the vast majority of assemblies) are said to be **culture-neutral** and have no culture identifier.

Finally, an assembly name can contain a **public key** that identifies the developer of the component. An assembly reference can use either the full 128-byte public key or the 8-byte public key token. The public key (token) is used to resolve file name collisions between organizations, allowing multiple `utilities.dll` components to coexist in memory and on disk provided that each one originates from a different organization, each of which is guaranteed to have a unique public key. The next section discusses public key management in detail.

Because assembly references occasionally must be entered by hand (for example, for use in configuration files), the CLR defines a standard format for writing four-part assembly names as strings. This format is known as the **display name** of the assembly. The display name of the assembly always begins with the simple `Name` of the assembly and is followed by an optional list of comma-delimited properties that correspond to the other three properties of the assembly name. If all four parts of the name are specified, the corresponding assembly reference is called a **fully qualified reference**. If one or more of the properties is missing, the reference is called a **partially qualified reference**.

Figure 2.5 shows a display name and the corresponding CLR attributes used to control each property. Note that if an assembly with no culture is desired, the display name must indicate this using `Culture=neutral`. Also, if an assembly with no public key is desired, the display name must indicate this using `PublicKeyToken=null`. Both of these are substantially different from a display name with no `Culture` or `PublicKeyToken` property. Simply omitting these properties from the display name results

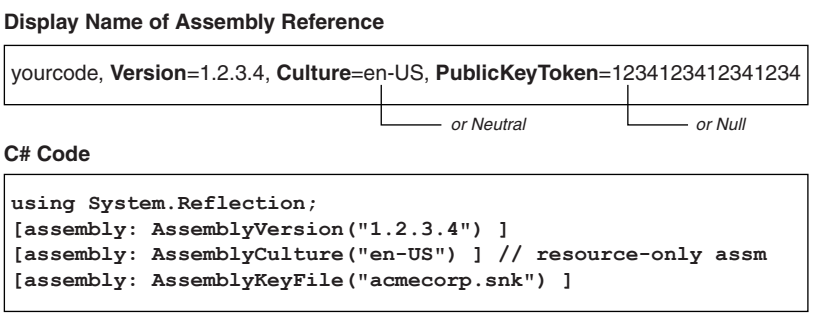

Figure 2.5: *Fully Specified Assembly Names*

in a partially specified name that allows any `Culture` or `PublicKeyToken` to be matched.

In general, you should avoid using partially specified assembly names; otherwise, various parts of the CLR will work in unexpected (and unpleasant) ways. However, to deal with code that does not heed this warning, the CLR allows partial assembly names to be fully qualified in configuration files. For example, consider the following application configuration file:

```
<configuration>
 <runtime>
   <asm:assemblyBinding
    xmlns:asm="urn:schemas-microsoft-com:asm.v1"
   >
     <asm:qualifyAssembly partialName="AcmeCorp.Code"
         fullName="AcmeCorp.Code,version=1.0.0.0,
         publicKeyToken=a1690a5ea44bab32,culture=neutral"
     />
    </asm:assemblyBinding>
  </runtime>
</configuration>
```

This configuration allows the following call to `Assembly.Load`:

```
Assembly assm = Assembly.Load("AcmeCorp.Code");
```

The preceding call behaves identically to a call such as this one:

```
Assembly assm = Assembly.Load("AcmeCorp.Code,"+
  "version=1.0.0.0,publicKeyToken=a1690a5ea44bab32,"+
  "culture=neutral");
```

The `partialName` attribute must match the parameter to `Assembly.Load` completely; that is, each property specified in the call to `Assembly.Load` must also be present in the `partialName` attribute in the configuration file. Also, each property specified in the `partialName` attribute must be present in the call to `Assembly.Load`. Later, this chapter discusses how configuration files are located.

Public Keys and Assemblies

The CLR uses public key technology both to uniquely identify the developer of a component and to protect the component from being tampered with once it is out of the original developer's hands. Each assembly can have a public key embedded in its manifest that identifies the developer. Assemblies with public keys also have a **digital signature** that is generated before the assembly is first shipped that provides a secure hash of the assembly manifest, which itself contains hashes of all subordinate modules. This ensures that once the assembly ships, no one can modify the code or other resources contained in the assembly. This digital signature can be verified using only the public key; however, the signature can be generated only with the corresponding **private key**, which organizations must guard more closely than their source code. The current builds of the CLR use RSA public/private keys and Secure Hash Algorithm (SHA) hashing to produce the digital signature. Although the private key used to sign the assembly is a unique fingerprint for each organization, it does not provide the same level of **nonrepudiation** that digital certificates provide. For example, there is no way to look up the developer's identity based solely on an assembly's public key. The CLR does provide support for embedding digital certificates into assemblies, but that is outside the scope of this chapter (for more information, see Chapter 9).

The .NET SDK ships with a tool (`SN.EXE`) that simplifies working with public and private keys during development and deployment. Running `SN.EXE` with the `-k` option creates a new file that contains a newly generated public/private key pair. This file contains your private key, so it is critical that you practice safe computing and do not leave this file in an unsecured location. Because the private key is so critical, most organizations

postpone the actual signing of the assembly until just before shipping, a practice called **delay signing**. To allow all developers in an organization to access the public key without having access to the private key, `SN.EXE` supports removing the private key portion using the `-p` option. This option creates a new file that contains only the public key. The conventional file extension for both public/private and public-only key files is `.SNK`.

The public key produced by `SN.EXE` is a 128-byte opaque algorithm-specific structure with an additional 32 bytes of header information. To keep the size of assembly references (and their display names) compact, an assembly reference can use a **public key token**, which is an 8-byte hash of the full public key. The assembly references emitted by most compilers use this token in lieu of the full public key to keep the overall size of the manifest small. You can calculate the token for a public key by using `SN.EXE`'s `-t` or `-T` options. The former calculates the token based on an `.SNK` file containing only a public key. The latter calculates the token based on a public key stored in an assembly's manifest. Figure 2.6 shows the `SN.EXE` tool in action.

Development tools that support the CLR must provide some mechanism for developers to sign their assemblies, either via custom attributes or

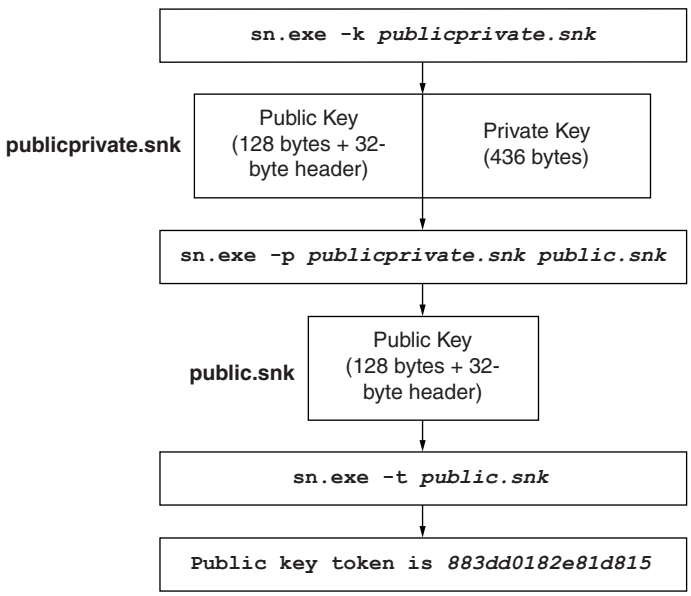

Figure 2.6: *Managing Public/Private Keys Using SN.EXE*

command-line switches. The `System.Reflection.AssemblyKeyFile` attribute tells the compiler where to find the `.SNK` file that contains the developer's public key. This attribute will work with either the public/private key pair or the public-only key, something that allows developers to build, test, and debug their components without access to the organization's private key. In order to build an assembly using only a public key, you must also use the `System.Reflection.AssemblyDelaySign` attribute to inform the compiler that no private key is present and that no meaningful digital signature can be produced. When delay signing is used, space is reserved for the digital signature so that a trusted member of the organization can re-sign the assembly without having to replicate the original developer's build environment. In general, assemblies that have a public key but do not have a valid signature cannot be loaded or executed. To allow delay-signed assemblies to be used during development, this policy can be disabled for a particular assembly or public key using the `-Vr` option to `SN.EXE`. Figure 2.7 shows the `AssemblyKeyFile` attribute used from C#. This figure also shows the resultant assembly as well as another assembly that references it. Note that the 128-byte public key is stored in

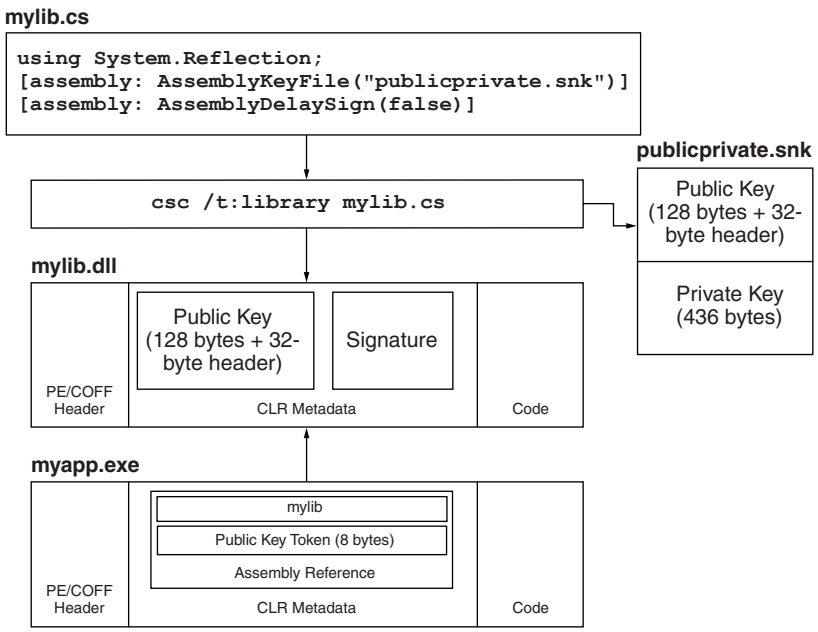

Figure 2.7: *Strong Assembly References*

the target's assembly manifest along with a digital signature to protect the assembly from tampering. Also note that the second assembly, which references the target, contains only the 8-byte public key token. Because the target assembly was built with delay signing turned off, the assembly can now be deployed and loaded in secured environments. In contrast, the target assembly produced by the C# compiler shown in Figure 2.8 is not suitable for deployment because it is built with delay signing turned on. However, after a trusted individual signs the assembly with the private key, the assembly is ready to be deployed. Note that in this example, the `SN.EXE` tool is used with the `-R` option, which overwrites the digital signature in the target assembly with one based on the public/private key provided on the command line. To manually verify that an assembly has been signed, you can use `SN.EXE` with the `-v` or `-vf` option. The latter overrides any configured settings that might disable signature verification.

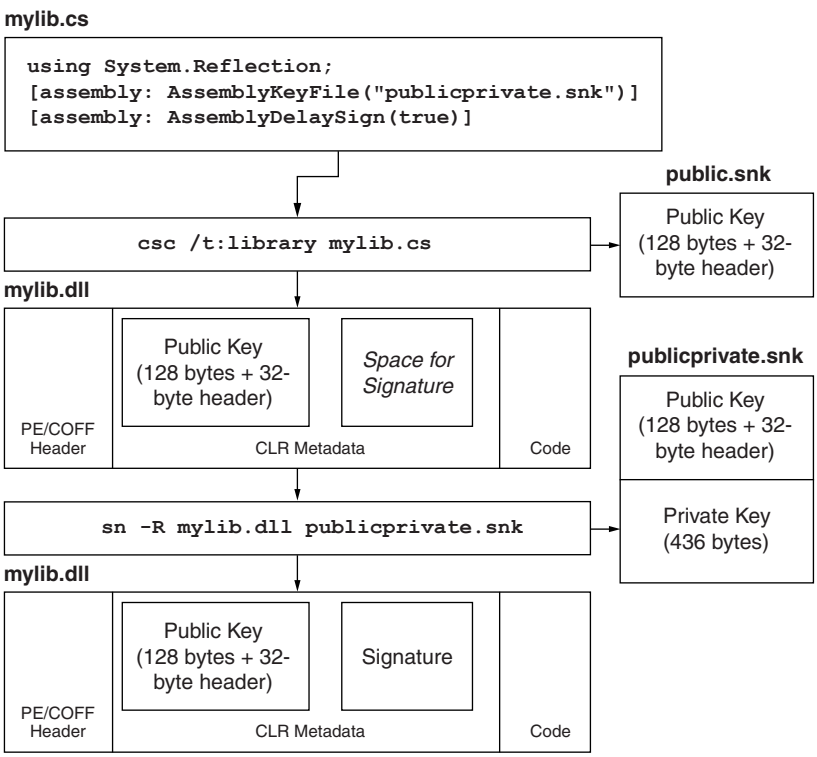

Figure 2.8: *Delay Signing an Assembly*

The CLR Loader

The **CLR loader** is responsible for loading and initializing assemblies, modules, resources, and types. The CLR loader loads and initializes as little as it can get away with. Unlike the Win32 loader, the CLR loader does not resolve and automatically load the subordinate modules (or assemblies). Rather, the subordinate pieces are loaded on demand only if they are actually needed (as with Visual C++ 6.0's delay-load feature). This not only speeds up program initialization time but also reduces the amount of resources consumed by a running program.

In the CLR, loading typically is triggered by the just in time (JIT) compiler based on types. When the JIT compiler tries to convert a method body from CIL to machine code, it needs access to the type definition of the declaring type as well as the type definitions for the type's fields. Moreover, the JIT compiler also needs access to the type definitions used by any local variables or parameters of the method being JIT-compiled. Loading a type implies loading both the assembly and the module that contain the type definition.

This policy of loading types (and assemblies and modules) on demand means that parts of a program that are not used are never brought into memory. It also means that a running application will often see new assemblies and modules loaded over time as the types contained in those files are needed during execution. If this is not the behavior you want, you have two options. One is to simply declare hidden static fields of the types you want to guarantee are loaded when your type is loaded. The other is to interact with the loader explicitly.

The loader typically does its work implicitly on your behalf. Developers can interact with the loader explicitly via the **assembly loader**. The assembly loader is exposed to developers via the `LoadFrom` static method on the `System.Reflection.Assembly` class. This method accepts a CODEBASE string, which can be either a file system path or a **uniform resource locator (URL)** that identifies the module containing the assembly manifest. If the specified file cannot be found, the loader will throw a `System.FileNotFoundException` exception. If the specified file can be found but is not a CLR module containing an assembly manifest, the loader will throw a `System.BadImageFormatException` exception. Finally, if the

CODEBASE is a URL that uses a scheme other than `file:`, the caller must have `WebPermission` access rights or else a `System.SecurityException` exception is thrown. Additionally, assemblies at URLs with protocols other than `file:` are first downloaded to the download cache prior to being loaded.

Listing 2.2 shows a simple C# program that loads an assembly located at `file://C:/usr/bin/xyzzy.dll` and then creates an instance of the contained type named `AcmeCorp.LOB.Customer`. In this example, all that is provided by the caller is the physical location of the assembly. When a program uses the assembly loader in this fashion, the CLR ignores the four-part name of the assembly, including its version number.

Listing 2.2: *Loading an Assembly with an Explicit CODEBASE*

```
using System;
using System.Reflection;
public class Utilities {
  public static Object LoadCustomerType() {
    Assembly a = Assembly.LoadFrom(
                    "file://C:/usr/bin/xyzzy.dll");
    return a.CreateInstance("AcmeCorp.LOB.Customer");
  }
}
```

Although loading assemblies by location is somewhat interesting, most assemblies are loaded by name using the **assembly resolver**. The assembly resolver uses the four-part assembly name to determine which underlying file to load into memory using the assembly loader. As shown in Figure 2.9, this name-to-location resolution process takes into account a variety of factors, including the directory the application is hosted in, versioning policies, and other configuration details (all of which are discussed later in this chapter).

The assembly resolver is exposed to developers via the `Load` method of the `System.Reflection.Assembly` class. As shown in Listing 2.3, this method accepts a four-part assembly name (either as a string or as an `AssemblyName` reference) and superficially appears to be similar to the `LoadFrom` method exposed by the assembly loader. The similarity is only skin deep because the `Load` method first uses the assembly resolver to find

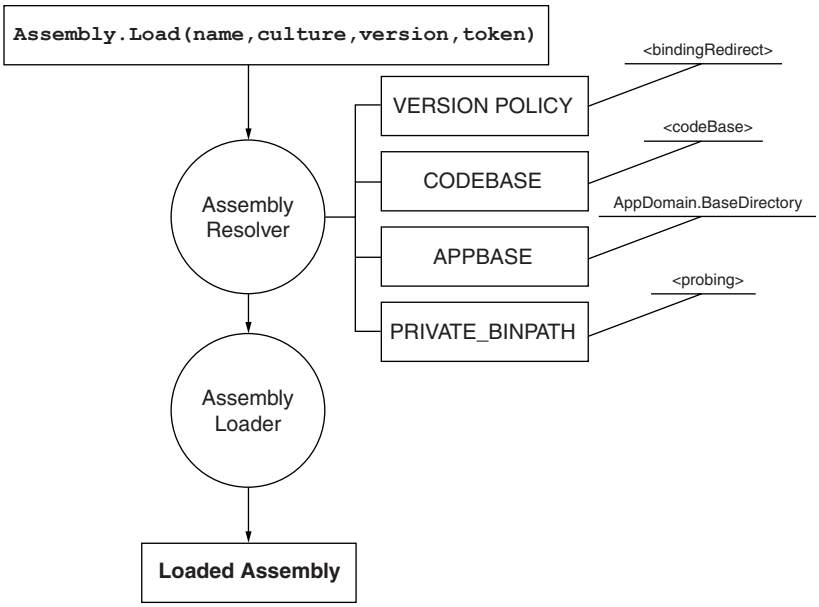

Figure 2.9: *Assembly Resolution and Loading*

a suitable file using a fairly complex series of operations. The first of these operations is to apply a version policy to determine exactly which version of the desired assembly should be loaded.

Listing 2.3: *Loading an Assembly Using the Assembly Resolver*

```
using System;
using System.Reflection;

public class Utilities {
  public static Object LoadCustomerType() {
    Assembly a = Assembly.Load(
      "xyzzy, Version=1.2.3.4, " +
      "Culture=neutral, PublicKeyToken=9a33f27632997fcc");
    return a.CreateInstance("AcmeCorp.LOB.Customer");
  }
}
```

The assembly resolver begins its work by applying any **version policies** that may be in effect. Version policies are used to redirect the assembly resolver to load an alternate version of the requested assembly. A version policy can map one or more versions of a given assembly to a different

version; however, a version policy cannot redirect the resolver to an assembly whose name differs by any facet other than version number (i.e., an assembly named `Acme.HealthCare` cannot be redirected to an assembly named `Acme.Mortuary`). It is critical to note that version policies are applied only to assemblies that are fully specified by their four-part assembly name. If the assembly name is only partially specified (e.g., the public key token, version, or culture is missing), then no version policy will be applied. Also, no version policies are applied if the assembly resolver is bypassed by a direct call to `Assembly.LoadFrom` because you are specifying only a physical path and not an assembly name.

Version policies are specified via configuration files. These include a machine-wide configuration file and an application-specific configuration file. The **machine-wide** configuration file is always named `machine.config` and is located in the `%SystemRoot%\Microsoft.Net\Framework\V1.0.nnnn\CONFIG` directory. The **application-specific** configuration file is always located at the APPBASE for the application. For CLR-based `.EXE` programs, the APPBASE is the base URI (or directory) for the location the main executable was loaded from. For ASP.NET applications, the APPBASE is the root of the Web application's virtual directory. The name of the configuration file for CLR-based `.EXE` programs is the same as the executable name with an additional `".config"` suffix. For example, if the launching CLR program is in `C:\myapp\app.exe`, the corresponding configuration file would be `C:\myapp\app.exe.config`. For ASP.NET applications, the configuration file is always named `web.config`.

Configuration files are based on the Extensible Markup Language (XML) and always have a root element named `configuration`. Configuration files are used by the assembly resolver, the remoting infrastructure, and by ASP.NET. Figure 2.10 shows the basic schema for the elements used to configure the assembly resolver. All relevant elements are under the `assemblyBinding` element in the `urn:schemas-microsoft-com:asm.v1` namespace. There are application-wide settings to control probe paths and publisher version policy mode (both of which are described later in this chapter). Additionally, the `dependentAssembly` elements are used to specify version and location settings for each dependent assembly.

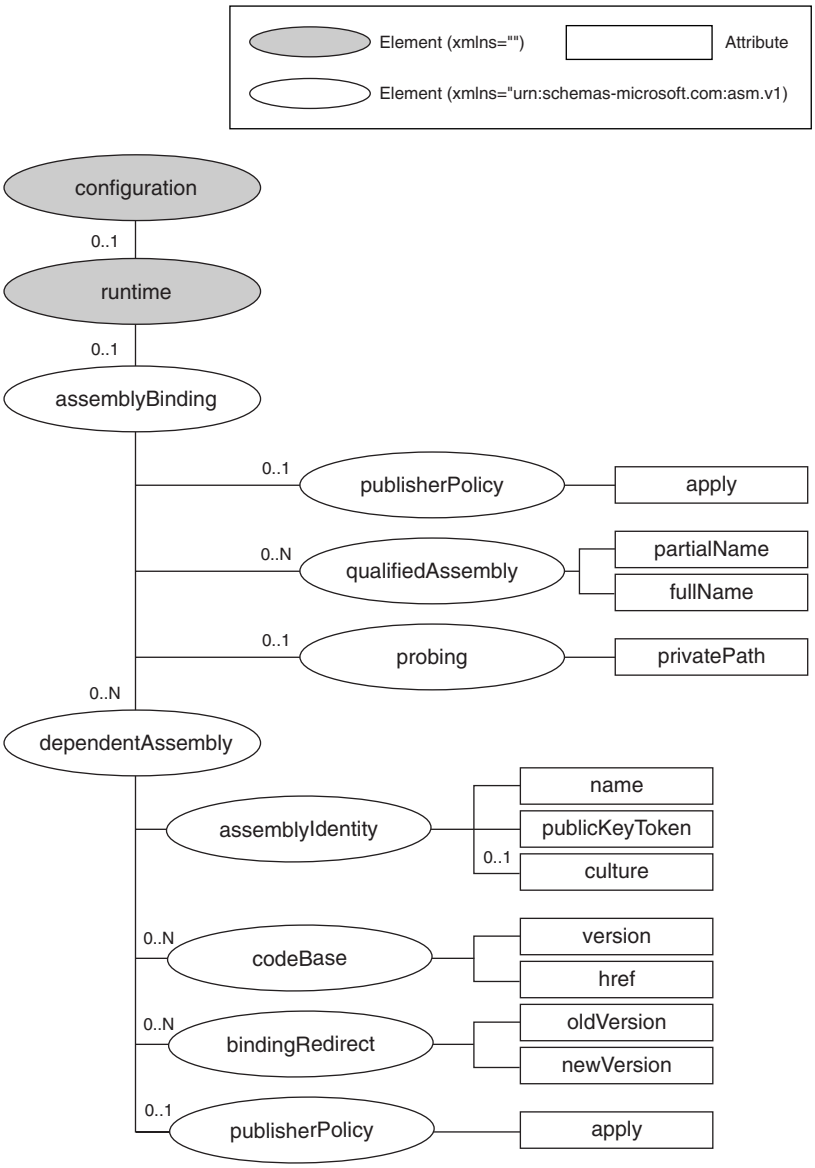

Figure 2.10: *Assembly Resolver Configuration File Format*

Listing 2.4 shows a simple configuration file containing two version policies for one assembly. The first policy redirects version 1.2.3.4 of the specified assembly (`Acme.HealthCare`) to version 1.3.0.0. The second policy redirects versions 1.0.0.0 through 1.2.3.399 of that assembly to version 1.2.3.7.

Listing 2.4: *Setting the Version Policy*

```
<?xml version="1.0" ?>

<configuration
    xmlns:asm="urn:schemas-microsoft-com:asm.v1"
>
  <runtime>
    <asm:assemblyBinding>
<!-- one dependentAssembly per unique assembly name -->
      <asm:dependentAssembly>
        <asm:assemblyIdentity
           name="Acme.HealthCare"
           publicKeyToken="38218fe715288aac" />
<!-- one bindingRedirect per redirection -->
        <asm:bindingRedirect oldVersion="1.2.3.4"
                     newVersion="1.3.0.0" />
        <asm:bindingRedirect oldVersion="1-1.2.3.399"
                     newVersion="1.2.3.7" />
      </asm:dependentAssembly>
    </asm:assemblyBinding>
  </runtime>
</configuration>
```

Version policy can be specified at three levels: per application, per component, and per machine. Each of these levels gets an opportunity to process the version number, with the results of one level acting as input to

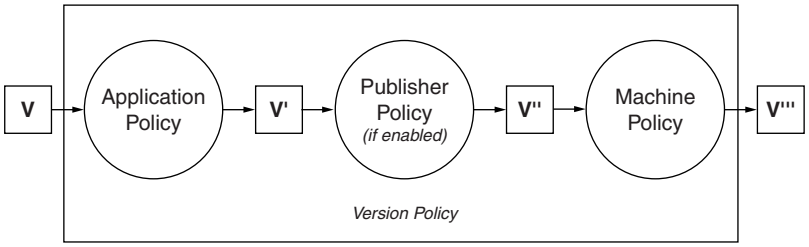

Figure 2.11: *Version Policy*

the level below it. This is illustrated in Figure 2.11. Note that if both the application's and the machine's configuration files have a version policy for a given assembly, the application's policy is run first, and the resultant version number is then run through the machine-wide policy to get the actual version number used to locate the assembly. In this example, if the machine-wide configuration file redirected version 1.3.0.0 of `Acme.HealthCare` to version 2.0.0.0, the assembly resolver would use version 2.0.0.0 when version 1.2.3.4 was requested because the application's version policy maps version 1.2.3.4 to 1.3.0.0.

In addition to application-specific and machine-wide configuration settings, a given assembly can also have a **publisher policy**. A publisher policy is a statement from the component developer indicating which versions of a given component are compatible with one another.

Publisher policies are stored as configuration files in the machine-wide global assembly cache. The structure of these files is identical to that of the application and machine configuration files. However, to be installed on the user's machine, the publisher policy configuration file must be wrapped in a surrounding assembly DLL as a custom resource. Assuming that the file `foo.config` contains the publisher's configuration policy, the following command line would invoke the assembly linker (`AL.EXE`) and create a suitable publisher policy assembly for `AcmeCorp.Code` version 2.0:

```
al.exe /link:foo.config
       /out:policy.2.0.AcmeCorp.Code.dll
       /keyf:pubpriv.snk
       /v:2.0.0.0
```

The name of the publisher policy file follows the form `policy.major.minor.assmname.dll`. Because of this naming convention, a given assembly can have only one publisher policy file per major.minor version. In this example, all requests for `AcmeCorp.Code` whose major.minor version is 2.0 will be routed through the policy file linked with `policy.2.0.AcmeCorp.Code.DLL`. If no such assembly exists in the **global assembly cache (GAC)**, then there is no publisher policy. As shown in Figure 2.11, publisher policies are applied after the application-specific

version policy but before the machine-wide version policy stored in `machine.config`.

Given the fragility inherent in versioning component software, the CLR allows programmers to turn off publisher version policies on an application-wide basis. To do this, programmers use the `publisherPolicy` element in the application's configuration file. Listing 2.5 shows this element in a simple configuration file. When this element has the attribute `apply="no"`, the publisher policies will be ignored for this application. When this attribute is set to `apply="yes"` (or is not specified at all), the publisher policies will be used as just described. As shown in Figure 2.10, the `publisherPolicy` element can enable or disable publisher policy on an application-wide or an assembly-by-assembly basis.

Listing 2.5: *Setting the Application to Safe Mode*

```
<?xml version="1.0" ?>

<configuration xmlns:rt="urn:schemas-microsoft-com:asm.v1">
  <runtime>
    <rt:assemblyBinding>
      <rt:publisherPolicy apply="no" />
    </rt:assemblyBinding>
  </runtime>
</configuration>
```

Resolving Names to Locations

After the assembly resolver decides which version of the assembly to load, it must locate a suitable file to pass to the underlying assembly loader. The CLR looks first in the directory specified by the `DEVPATH` operating system (OS) environment variable. This environment variable is typically not set on the deployment machine. Rather, it is intended for developer use only and exists to allow delay-signed assemblies to be loaded from a shared file-system directory. Moreover, the `DEVPATH` environment variable is considered only if the following XML configuration file element is present in the `machine.config` file:

```
<configuration>
  <runtime>
    <developmentMode developerInstallation="true" />
  </runtime>
</configuration>
```

Because the `DEVPATH` environment variable is not intended for deployment, the remainder of the chapter will ignore its existence.

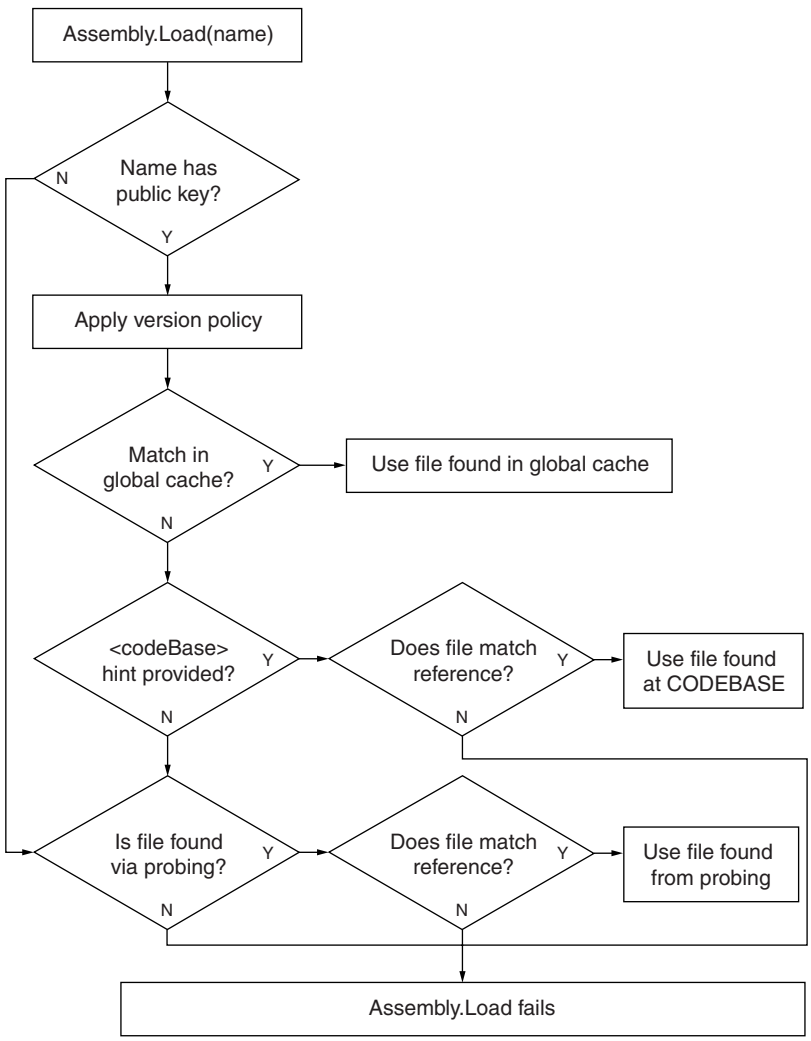

Figure 2.12: *Assembly Resolution*

Figure 2.12 shows the entire process the assembly resolver goes through in order to find an appropriate assembly file. In normal deployment scenarios, the first location that the assembly resolver uses to find an assembly is the global assembly cache (GAC). The GAC is a machine-wide code cache that contains assemblies that have been installed for machine-wide use. The GAC allows administrators to install assemblies once per machine for all applications to use. To avoid system corruption, the GAC accepts only assemblies that have valid digital signatures and public keys. Additionally, entries in the GAC can be deleted only by administrators, something that prevents non-admin users from deleting or moving critical system-level components.

To avoid ambiguity, the assembly resolver will look in the GAC only if the requested assembly name contains a public key. This prevents requests for generic names such as `utilities` from being satisfied by the wrong implementation. The public key can be provided either explicitly as part of an assembly reference or parameter to `Assembly.Load` or implicitly via the `qualifyAssembly` configuration file element.

The GAC is controlled by a system-level component (`FUSION.DLL`) that keeps a cache of DLLs under the `%WINNT%\Assembly` directory. `FUSION.DLL` manages this directory hierarchy for you and provides access to the stored files based on the four-part assembly name, as shown in Table 2.4. Although one can traverse the underlying directories, the scheme used by FUSION to store cached DLLs is an implementation detail that is guaranteed to change as the CLR evolves. Instead, you must interact with the GAC via the `GACUTIL.EXE` tool or some other facade over the FUSION

TABLE 2.4 Global Assembly Cache

Name	Version	Culture	Public Key Token	Mangled Path
yourcode	1.0.1.3	de	89abcde...	t3s\e4\yourcode.dll
yourcode	1.0.1.3	en	89abcde...	a1x\bb\yourcode.dll
yourcode	1.0.1.8	en	89abcde...	vv\a0\yourcode.dll
libzero	1.1.0.0	en	89abcde...	ig\u\libzero.dll

application programming interface (API). One such facade is `SHFUSION.DLL`, a Windows Explorer shell extension that provides a user-friendly interface to the GAC.

If the assembly resolver cannot find the requested assembly in the GAC, the assembly resolver then tries to use a **CODEBASE hint** to access the assembly. A CODEBASE hint simply maps an assembly name to a file name or URL where the module containing the assembly manifest is located. Like version policies, CODEBASE hints are located in both application- and machine-wide configuration files. Listing 2.6 shows an example configuration file that contains two CODEBASE hints. The first hint maps version 1.2.3.4 of the `Acme.HealthCare` assembly to the file `C:\acmestuff\Acme.HealthCare.DLL`. The second hint maps version 1.3.0.0 of the same assembly to the file located at `http://www.acme.com/bin/Acme.HealthCare.DLL`.

Assuming that a CODEBASE hint is provided, the assembly resolver can simply load the corresponding assembly file, and the loading of the assembly proceeds as if the assembly were loaded by an explicit CODEBASE a la `Assembly.LoadFrom`. However, if no CODEBASE hint is provided, the assembly resolver must begin a potentially expensive procedure for finding an assembly file that matches the request.

Listing 2.6: *Specifying the CODEBASE Using Configuration Files*

```
<?xml version="1.0" ?>

<configuration
    xmlns:asm="urn:schemas-microsoft-com:asm.v1"
>
  <runtime>
    <asm:assemblyBinding>
<!-- one dependentAssembly per unique assembly name -->
      <asm:dependentAssembly>
        <asm:assemblyIdentity
           name="Acme.HealthCare"
           publicKeyToken="38218fe715288aac" />
<!-- one codeBase per version -->
        <asm:codeBase
           version="1.2.3.4"
           href="file://C:/acmestuff/Acme.HealthCare.DLL"/>
        <asm:codeBase
           version="1.3.0.0"
```

```
            href="http://www.acme.com/Acme.HealthCare.DLL"/>
      </asm:dependentAssembly>
    </asm:assemblyBinding>
  </runtime>
</configuration>
```

If the assembly resolver cannot locate the assembly using the GAC or a CODEBASE hint, it performs a search through a series of directories relative to the root directory of the application. This search is known as **probing**. Probing will search only in directories that are at or below the APPBASE directory (recall that the APPBASE directory is the directory that contains the application's configuration file). For example, given the directory hierarchy shown in Figure 2.13, only directories `m`, `common`, `shared`, and `q` are eligible for probing. That stated, the assembly resolver will probe only into subdirectories that are explicitly listed in the application's configuration file. Listing 2.7 shows a sample configuration file that sets the relative search path to the directories `shared` and `common`. All subdirectories of APPBASE that are not listed in the configuration file will be pruned from the search.

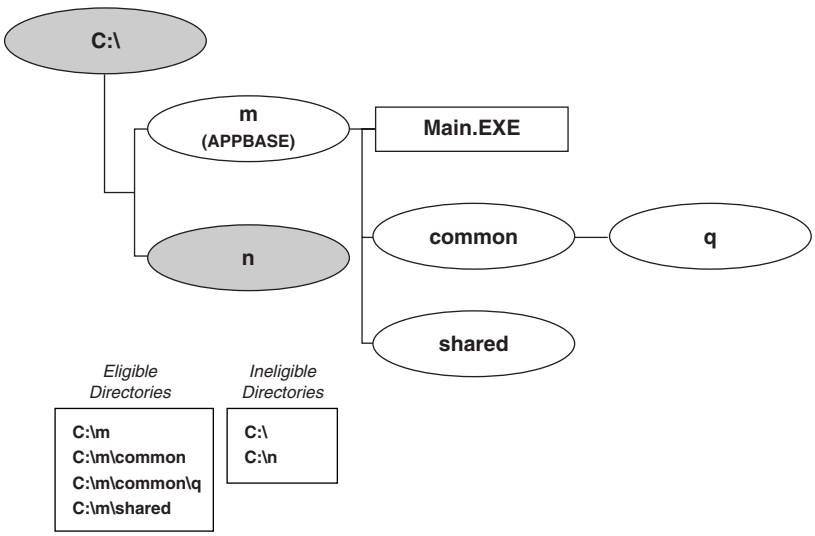

Figure 2.13: *APPBASE and the Relative Search Path*

Listing 2.7: *Setting the Relative Search Path*

```
<?xml version="1.0" ?>
<configuration
    xmlns:asm="urn:schemas-microsoft-com:asm.v1"
>
  <runtime>
    <asm:assemblyBinding>
      <asm:probing privatePath="shared;common"  />
    </asm:assemblyBinding>
  </runtime>
</configuration>
```

When probing for an assembly, the assembly resolver constructs CODEBASE URLs based on the simple name of the assembly, the relative search path just described, and the requested culture of the assembly (if present in the assembly reference). Figure 2.14 shows an example of the CODEBASE URLs that will be used to resolve an assembly reference with no culture specified. In this example, the simple name of the assembly is `yourcode` and the relative search path is the `shared` and `common` directories. The assembly resolver first looks for a file named `yourcode.dll` in the APPBASE directory. If there is no such file, the resolver then assumes that the assembly is in a directory with the same name and looks for a file with that name under the `yourcode` directory. If the file is still not found, this process is repeated for each of the entries in the relative search path until a file named `yourcode.dll` is found. If the file is found, then probing stops. Otherwise, the probe process is repeated, this time looking for the file named `yourcode.exe` in the same locations as before. Assuming that a file is found, the assembly resolver verifies that the file matches all properties of the assembly name specified in the assembly reference and then loads the assembly. If one of the properties of the file's assembly name does not match all of the (post-version policy) assembly reference's properties, the `Assembly.Load` call fails. Otherwise, the assembly is loaded and ready for use.

Probing is somewhat more complex when the assembly reference contains a culture identifier. As shown in Figure 2.15, the preceding algorithm is augmented by looking in subdirectories whose names match the

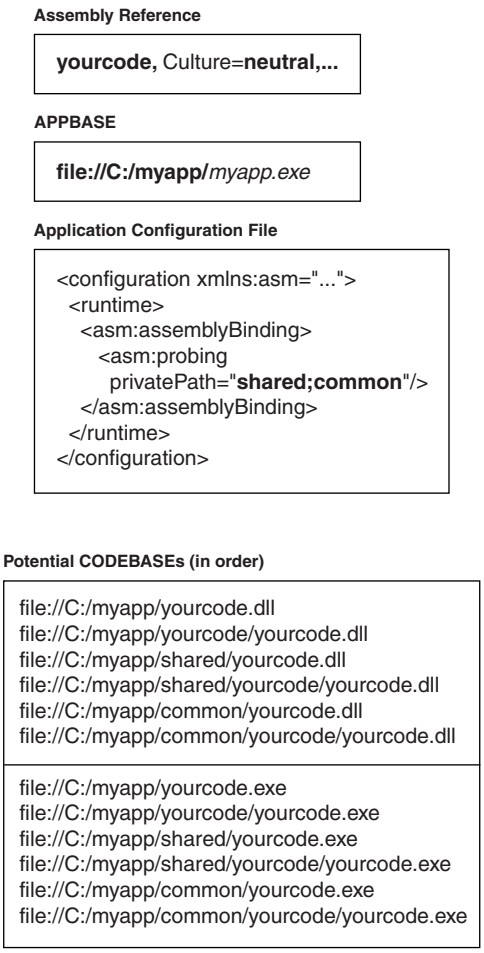

Figure 2.14: *Culture-Neutral Probing*

requested culture. In general, applications should keep relative search paths small to avoid excessive load-time delays.

Versioning Hazards

The preceding discussion of how the assembly resolver determines which version of an assembly to load focuses primarily on the mechanism used by the CLR. What was not discussed is what policies a developer should use to determine when, how, and why to version an assembly. Given that the platform being described has not actually shipped at the time of this writing, it

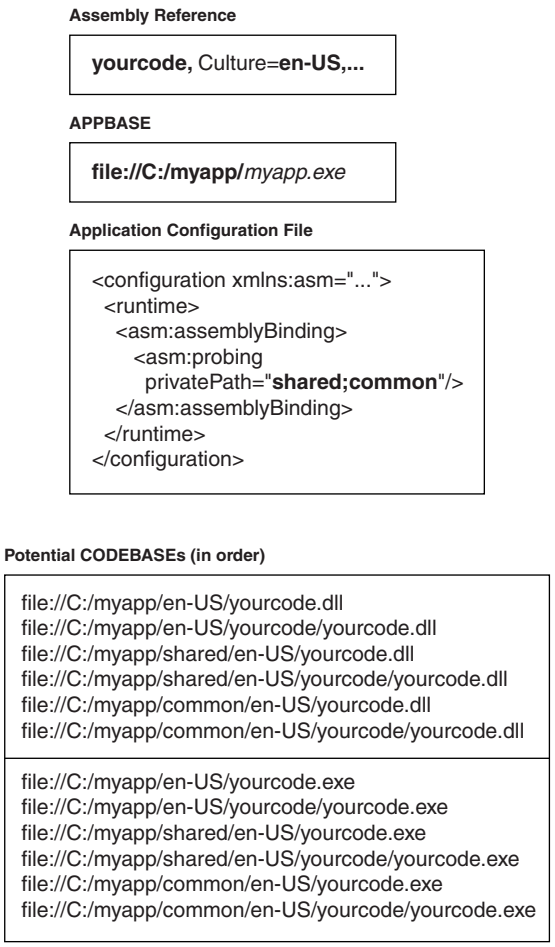

Figure 2.15: *Culture-Dependent Probing*

is somewhat difficult to list a set of "best practices" that are known to be valid based on hard-won experiences. However, it is reasonable to look at the known state of the CLR and extrapolate a set of guidelines.

It is important to note that assemblies are versioned as a unit. Trying to replace a subset of the files in an assembly without changing the version number will certainly lead to unpredictability. To that end, the remainder of this section looks at versioning with respect to an assembly as a whole rather than versioning individual files in an assembly.

The question of when to change version numbers is an interesting one. Obviously, if the public contract of a type changes, the type's assembly

must be given a new version number. Otherwise, programs that depend on one version of the type signature will get runtime errors when a type with a different signature is loaded. This means that if you add or remove a `public` or `protected` member of a public type, you must change the version number of the type's assembly. If you change the signature of a public or protected member of a public type (e.g., adding a method parameter, changing a field's type), you also need a new assembly version number. These are absolute rules. Violating them will result in unpredictability.

The more difficult question to answer relates to modifications that do not impact the public signature of the assembly's types. For example, changes to a member that is marked as `private` or `internal` are considered nonbreaking changes, at least as far as signature matching is concerned. Because no code outside of your assembly can rely upon `private` or `internal` members, having signature mismatches occur at runtime is a nonissue because it doesn't happen. Unfortunately, signature mismatches are only the tip of the iceberg.

There is a reasonable argument to be made for changing the version number for every build of an assembly, even if no publicly visible signatures have changed. This approach is supported by the fact that even a seemingly innocuous change to a single method body may have a subtle but very real rippling effect on the behavior of programs that use the assembly. If the developer gives each build of an assembly a unique version number, code that is tested against a particular build won't be surprised at deployment time.

The argument against giving each build of an assembly a unique version number is that "safe" fixes to the code won't be picked up by programs that are not rebuilt against the new version. This argument doesn't hold water in the face of publisher policy files. Developers who use unique version numbers for every build are expected to provide publisher policy files that state which versions of their assembly are backward-compatible. By default, this gives consumers of the down-level version an automatic upgrade to the newer (and hopefully faster or less buggy) assembly. For times when the assembly's developer guesses wrong, each application can use the `publisherPolicy` element in its configuration file to disable the automatic upgrade, in essence running the application in "safe mode."

As discussed earlier, the CLR assembly resolver supports side-by-side installation of multiple versions of an assembly via CODEBASE hints, private probe paths, and the GAC. This allows several versions of a given assembly to peacefully coexist in the file system. However, things become somewhat unpredictable if more than one of these assemblies is actually loaded into memory at any one time, either by independent programs or by a single program. Side-by-side execution is much harder to deal with than side-by-side installation.

The primary problem with supporting multiple versions in memory at once is that, to the runtime, the types contained in those assemblies are distinct. That is, if an assembly contains a type called `Customer`, then when two different versions of the assembly are loaded, there are two distinct types in memory, each with its own unique identity. This has several serious downsides. For one, each type has its own copy of any static fields. If the type needed to keep track of some shared state is independent of how many versions of the type had been loaded, it could not use the obvious solution of using a static field. Rather, developers would need to rewrite the code with versioning in mind and store the shared state in a location that is not version-sensitive. One approach would be to store the shared state in some runtime-provided place such as the ASP.NET application object. Another approach would be to define a separate type that contained only the shared state as static fields. Developers could deploy this type in a separate assembly that would never be versioned, thus ensuring that only one copy of the static fields would be in memory for a given application.

Another problem related to side-by-side execution arises when versioned types are passed as method parameters. If the caller and callee of a method have differing views on which version of an assembly will be loaded, the caller will pass a parameter whose type is unknown to the callee. Developers can avoid this problem by always defining parameter types using version-invariant types for all public (cross-assembly) methods. More importantly, these shared types need to be deployed in a separate assembly that itself will not be versioned.

The metadata for an assembly has three distinguished attributes that allows the developer to specify whether multiple versions of the assembly can be loaded at the same time. If none of these attributes is present, the assembly is assumed safe for side-by-side execution in all scenarios. The `nonsidebysideappdomain` attribute indicates that only one version of the assembly can be loaded per AppDomain. The `nonsidebyside-process` attribute indicates that only one version of the assembly can be loaded per process. Finally, the `nonsidebysidemachine` attribute indicates that only one version of the assembly can be loaded at a time for the entire machine. At the time of this writing, these metadata bits are ignored by the assembly resolver and loader. However, they do serve as a hint that hopefully will be enforced in future versions of the CLR.

Where Are We?

Modules and assemblies are the component building blocks of the CLR. Each CLR type resides in exactly one physical file (called a module), which contains the code and metadata that make that type real. To be deployed, a module must be associated with a logical assembly that gives the module's types a fully qualified name. The CLR loader works primarily in terms of assemblies, with modules (and types) being loaded only as they are needed. The CLR loader typically works in terms of location-independent assembly names that are resolved to physical file paths or URLs prior to loading. This not only allows more flexible deployment and versioning, but it also ensures that the component's origin cannot be spoofed through the use of public keys and digital signatures.