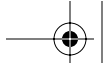O   N   E

Microsoft .NET Framework

Microsoft .NET has been considered a paradigm shift in the world of Web application development and deployment. Web services are being envisioned as the next big step in Internet and Internet technology. Microsoft .NET simplifies the development of Web services; in fact, development of Web services has been one of the major design goals behind .NET. Let's understand what Microsoft .NET Framework is and what benefits it gives to the application developer in terms of development efforts and platforms. This chapter gives an introduction to the .NET Framework, and it gives the reader an overview of the Common Language Runtime (CLR), Common Type System (CTS), and Common Language Specification (CLS). The chapter also gives an overview of the .NET Framework class library.

There are many books that highlight the importance of .NET over earlier technologies. Hence, in this chapter rather than listing all the features of .NET in detail, we emphasize those key aspects of .NET that need to be considered while planning for migration of existing applications to .NET. In this book we have included as many code snippets as possible, highlighting the syntactical differences between the languages being discussed and then supplying workarounds wherever feasible.

One of the main objectives of this chapter is to make the reader aware of the steps that must be followed while migrating any application from one technology to another.

Need for .NET Framework

Before understanding the Microsoft .NET Framework, it is important to understand the current limitations of Microsoft technologies as well other technologies such as Java.

1

In the current Internet scenario, various applications run on multiple networks. These applications could have been written using different languages provided by various vendors. Microsoft provides different programming languages such as Visual Basic and Visual C++ for application development. For example, a manufacturing organization would have different systems, such as an inventory management system, a bill of material systems, and a general ledger system, all implemented using various technologies available for application development. These systems need to be integrated to form a higher-level enterprise information system for an organization. To do so, application developers had to use technologies such as Microsoft Distributed Component Object Model (DCOM), Component Object Request Broker Architecture (CORBA), Java Remote Method Invocation (RMI), and so on. However, these distributed technologies are very tightly coupled to the programming languages in which these applications have been developed. This leads to problems in application integration if the applications have been developed using disparate technologies.

The other limitation with earlier technologies was that cross-language interoperability was very limited. For example, if classes had been developed in Visual C++, it was not possible to develop new classes in Visual Basic that extended from those developed in Visual C++. Therefore, developers would have to rewrite the same logic for the classes in all the languages that were supported in their organization. Functional reusability was supported, but true code reusability was not available in earlier technologies; therefore, a developer was forced to learn all languages that were being used for application development in the organization.

Developers who have developed applications using Microsoft technologies have extensively used Component Object Model (COM) technology to develop components. These components were used mainly to encapsulate the business logic of the application. COM technology suffered from the following limitations:

- **Registration of COM components.** COM components had to be registered on the target machine before they could be used by the application. The application had to look up the Windows registry to locate and load the COM components.
- **Unloading COM components.** COM objects also required a special logic for freeing up the objects from memory. This method is known as reference counting of COM objects. It is used to keep track of the number of active references. When an object's reference count reaches zero, the object is removed from memory. The major problem that arises out of this situation is that of circular reference. If circular references exist between two COM components, they would not be freed from memory.
- **Data link library (DLL) hell.** Whenever applications that use COM components were installed on a machine, the installation process would update the registry with the COM components information. Thus, there was a chance that these DLLs would be overwritten
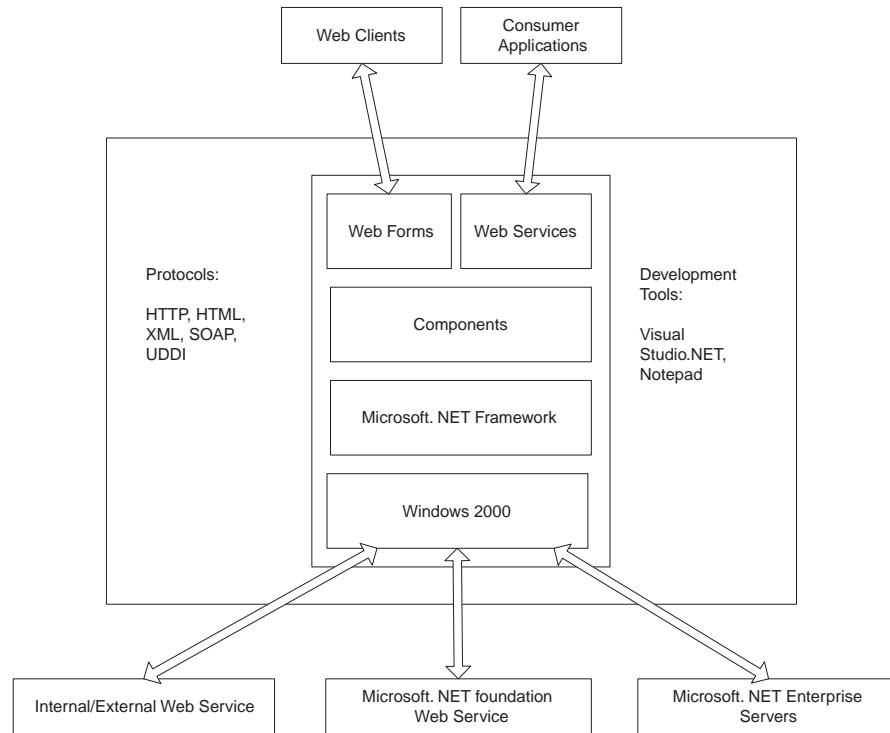
Figure 1-1   *Building blocks of Microsoft .NET platform.*

when some other applications were installed on the same computer. Therefore, an application that had been referring to one particular DLL would refer to the wrong DLL. This caused a major problem when an application was referring to particular version of a DLL.

Microsoft .NET has been developed keeping the above limitations in mind and making Web services development one of its major goals.

Building Blocks of .NET

Figure 1-1 shows building blocks of Microsoft .NET platform. A brief explanation of each of these building blocks is presented below.

The .NET Framework Features

Microsoft .NET Framework is a computing platform for developing distributed applications for the Internet. Following are the design goals of Microsoft .NET Framework:
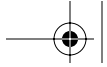
1. To provide a very high degree of language interoperability
2. To provide a runtime environment that completely manages code execution
3. To provide a very simple software deployment and versioning model
4. To provide high-level code security through code access security and strong type checking
5. To provide a consistent object-oriented programming model
6. To facilitate application communication by using industry standards such as SOAP and XML.
7. To simplify Web application development

Visual Studio .NET

Visual Studio .NET is a new development environment and a rapid application development (RAD) tool that is fully supported by the MSDN developer service and Windows DNA 2000 servers. It is an integrated development environment (IDE) that is common to all the Microsoft programming languages in the .NET Framework. Visual Studio .NET is based on XML and it uses XML for data storage and formatting.

Visual Studio .NET enables the easy deployment of highly distributed, programmable services that run across stand-alone machines, in corporate data centers, and across the Internet. The following types of applications can be created using it:

- Console applications
- Rich Windows graphical user interface (GUI) applications
- Class library
- Web applications
- ASP .NET Web services
- Windows as well as Web control library
- Windows services

Apart from good debugging features, Visual Studio .NET has features such as source code version control through Visual Source Safe (VSS) and true IntelliSense.

NET Enterprise Servers

The server infrastructure for .NET Framework, including Windows and the .NET Enterprise Servers, is a suite of infrastructure applications for building, deploying, and operating XML Web services. Key technologies include support for XML, scale-out, and business process orchestration across applications and services. These servers include

- Application Center 2000 to enable scale-out solutions
- BizTalk Server 2000 to create and manage XML-based business process orchestration across applications and services

- Host Integration Server 2000 for accessing data and applications on mainframes
- Mobile Information 2001 Server to enable use of applications by mobile devices like cell phones
- SQL Server 2000 to store and retrieve structured XML data

Microsoft .NET Building Block Services

The building block services are a set of XML Web services that are used to move control of user data between various applications and users. They enable personalized simplicity and consistency across the various applications, services, and devices while ensuring that user consent is the basis for all transactions.

The building block services include Passport (for user identification) and services for message delivery, file storage, user-preference management, calendar management, and other functions. Microsoft will offer a few building block services in areas that are critical to the infrastructure of .NET Framework; a wide range of partners and developers are expected to significantly expand the set of building block services. You'll also see corporate and vertical building block services built on the .NET platform.

Introduction to Microsoft .NET Framework

Microsoft .NET Framework is a new programming model that simplifies application development for the highly distributed environment of the Internet. The major components of the .NET Framework are the CLR and the .NET Framework class library. Figure 1-2 shows the architecture of the Microsoft .NET Framework and illustrates features such as ADO.NET, Windows Forms, Web Forms, and Web services. These are new features that have been introduced in the Microsoft .NET Framework.

Windows Forms are used in developing rich GUI desktop .NET applications. Web Forms are used to develop Web applications. With the introduction of Web Forms, the .NET Framework brings the ease of drag and drop designing into Web application development.

ADO.NET brings with it the concept of disconnected data access. This vastly improves the performance of ASP.NET applications. ADO.NET components have been logically grouped into data access and data manipulation components. The data access components first access the data. Then a connection with the data source is no longer required. Once data has been retrieved, the data manipulation components manipulate the data according to the business logic. The ADO.NET classes are found in the **System.Data** namespace. The concept of namespaces is covered in a later section in this chapter.
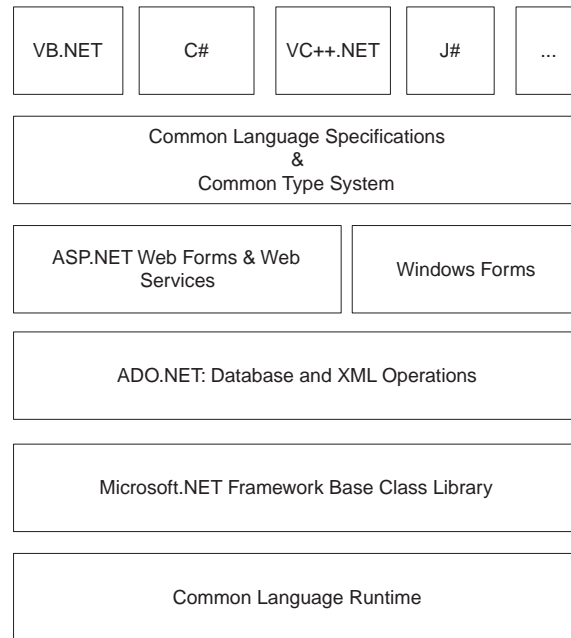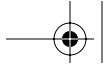
VB.NET	C#	VC++.NET	J#	...

Common Language Specifications & Common Type System

ASP.NET Web Forms & Web Services	Windows Forms

ADO.NET: Database and XML Operations

Microsoft.NET Framework Base Class Library

Common Language Runtime

Figure 1–2   *Microsoft .NET Framework.*

The CLR, CTS, and the CLS are explained in more detail in the next subsections.

Introduction to the CLR

The CLR is the execution environment provided by the Microsoft .NET Framework. It provides many services such as

- Automatic garbage collection
- Code access security
- Simplified versioning
- Simple and reliable deployment
- Deep cross-language interoperability
- Debugging across different languages
- Performance
- Scalability

Because the CLR manages the code execution, all the code that is targeted for the CLR is known as managed code. Managed code emits metadata along with the executable. This metadata is used to describe the types (classes) and members used in the code, along with all the external references used in executing the code. The CLR uses this meta-

data to load the classes during execution and resolve method invocations during runtime.

The CLR provides automatic garbage collection of the objects that have been loaded into memory. All objects that are created via the *new* operator are allocated memory on the heap. A program can allocate as many objects as are required by the program logic. However, when an object is no longer required, there must be some mechanism to free up the memory that was occupied by the object.

This is accomplished in the CLR via a program called garbage collector, which collects all objects in memory that have no references. This program runs as a low-priority thread in the background process and collects all unreferenced objects. Because memory management is automatic, the chances for memory leaks in the program are minimized. However, the time when garbage collector would actually release the objects from the memory is not known. This concept is known as nondeterministic garbage collection because it cannot be determined in advance when the objects would be released from memory.
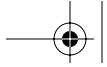
If sufficient memory is not available for creating new objects, the CLR throws an exception that can be caught and gracefully handled by the application.

Code Access Security (CAS), as the name suggests, is used to control the access that the code has to system resources. The CLR has a runtime security system. Administrators can configure policy settings by specifying the resources that can be accessed by the code.

A call stack is created that represents the order in which the assemblies get called. The CLR's security system walks the stack to determine whether the code is authorized to access the system resources or perform certain operations. If any caller in the call stack does not have the requisite permission to access the specific system resources, a security exception is thrown by the CLR.

Simplified versioning is another feature provided in the .NET Framework. It supports versioning and also provides for side-by-side execution of different versions of the same component. The specific versions of the assembly and the dependent assemblies are stored in the assembly's manifest. The copies of the same assembly that differ only in version numbers are considered to be different assemblies by the CLR. Assemblies are explained in more detail in the later sections.

Simplified deployment is one of the features provided in the .NET Framework. The most important point to mention is that .NET components do not need to be registered in the Windows registry. All code generated in the .NET Framework is self-describing because assemblies contain the manifest and metadata information. This information contains all the data about the dependencies of the assembly and the specific versions of the components that these assemblies would use at execution time; therefore, multiple

versions of the same components can coexist. The CLR enforces the versioning policy.

Cross-language interoperability is an important feature, and it was one of the design goals of the .NET Framework. This feature is possible because of the CTS and CLS. The CTS is explained in more detail in the next subsection.

Visual Studio .NET allows for debugging across an application consisting of different languages targeted for the CLR. In fact, the IDE also allows for debugging an application in which managed code interacts with unmanaged code.

CLR ensures that performance of the code execution is optimized. Compiled code is stored in cache. When the same code is called next time, this code is loaded into memory from cache. This advantage stands out more in the case of ASP.NET applications than for ASP applications. ASP code was interpreted every time an ASP page was requested. In ASP.NET, the code is compiled only once when the page is requested for the first time. This ensures that performance is optimized.

The .NET Framework also provides some classes for tracking the performance of the .NET applications. These classes are known as performance counters. The .NET Framework provides performance counters for getting information on exception handling, interoperation with unmanaged code, loading and unloading code into memory, locking and threading, memory, networking operations, and so on. These performance counters help to fine-tune the performance of the .NET applications.

Introduction to the CTS

The CTS defines how types are declared, used, and managed in the runtime environment. The CTS is the key element for the CLR's support of cross-language integration. The common type system is used to

- Enable cross-language integration, type safety, and high-performance code execution.
- Define rules that languages should follow. This helps to ensure that objects written in different languages can interact with each other.

The Microsoft .NET Framework supports two categories of types, reference and value. As the name suggests, reference types contain a reference to memory address of a value stored in memory. If any changes are made to the value using the reference address, the original value is changed in memory. Reference type variables are allocated in the heap memory.

Value types, on the other hand, contain the actual value. If the value of one variable (of value type) is assigned to another variable (of value type), the contents of the first variable are copied into the second variable. If any changes are made in the second variable, the contents of the first variable are not changed. Value type variables are allocated on the stack. Value types are stored more efficiently as primitive types. Value types are derived from the
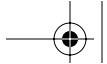
System.ValueType class. Because of this, the value type variables can have fields, properties, and events, just as reference type of variables.

Microsoft .NET has introduced a concept called boxing and unboxing. Boxing involves the process of converting a value type variable into a reference type variable. Boxing a variable of a value type allocates an object instance on the heap and copies the value of the value type variable into the heap. Unboxing is the explicit conversion from the object type to a value type. During unboxing an **InvalidCastException** might be thrown if the source argument is null or is a reference to an incompatible type.

A type definition includes the type name, visibility, base type, interfaces implemented by the new type, and members of the new type. A type needs to be identified by a name. It can have global access; that is, all other assemblies can access the type if the accessibility of the type is public. If the accessibility of the type is assembly, the type can be accessed only within the assembly in which the type is defined. A type can inherit from other types and extend the behavior of the base types. A type can inherit only from a single type. It can also implement any number of interfaces. In addition, attributes can be used with the types to provide more information about the types.

Value types are built-in data types provided by the programming languages supported in the Microsoft .NET Framework. **Integer**, **Float**, and **Double** are some examples of built-in data types and are value types. User-defined value types can be defined. **Structure** is a common example of a user-defined value type that is supported in Visual Basic .NET.

Some of the reference types found in the Microsoft .NET Framework are the classes, arrays, pointers, delegates, and so on.

Introduction to the CLS

The CLS rules define the basic language features required by many applications and promoting language interoperability. The CLS defines a subset of the CTS (explained in the previous section). All languages targeted for the Microsoft .NET Framework comply with the CLS. This ensures language interoperability between the various languages. The CLS also establishes requirements for CLS compliance; these help developers determine whether their managed code conforms to the CLS.

The CLS rules are normally used from the perspective of the high-level source code and tools, such as compilers, that are used in the process of generating assemblies in the Microsoft .NET Framework.

The CLS rules apply only to externally visible items. Within a single assembly there are no restrictions to the programming techniques that are used. Code can be marked as CLS compliant or otherwise with the help of custom attributes.

An assembly can contain many types. If an assembly is marked as CLS compliant with a custom attribute, all types within that assembly are

automatically CLS compliant. However, one can also mark individual types with custom attributes to make them non-CLS compliant. Similarly, if a type is marked as CLS compliant, all the members of that type are automatically CLS compliant unless marked as non-CLS compliant with the help of attributes.

Most of the classes found in the .NET Framework class library are CLS compliant. Thus, these classes can be used from all languages that are targeted for the .NET Framework.

Understanding Compilation in .NET Framework

Compilation is a two-step process in the .NET Framework, as shown in Figure 1-3. In the first step, the language compiler generates Microsoft Intermediate Language (MSIL) Generally all compilers complying with the CLR will generate MSIL. Therefore, the CLR is presented with MSIL, no matter what language has been used to develop the code. MSIL is commonly referred as the language of the CLR.

In the second step of compilation, the just-in-time (JIT) compiler compiles MSIL into native code, which can be executed on specific hardware and operating systems. The JIT compiler is a part of the runtime execution environment.
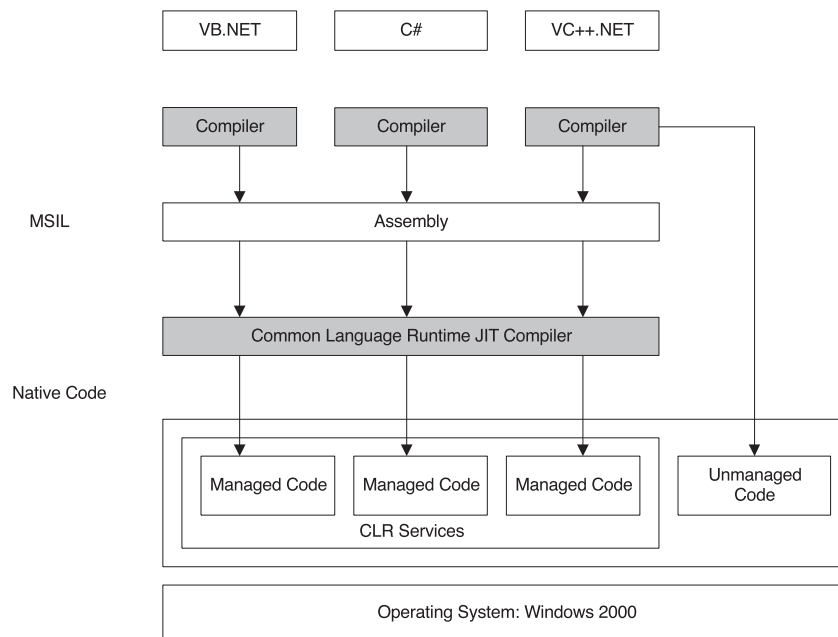
Figure 1-3   *Compilation in Microsoft .NET.*
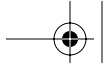
In Microsoft .NET there are three types of JIT compilers:

- **Pre-JIT.** Pre-JIT compiles complete source code into native code in a single compilation cycle. This is done at the time of deployment of the application.
- **Econo-JIT.** Econo-JIT compiles only those methods that are called at runtime. However, these compiled methods are removed when they are not required.
- **Normal-JIT.** Normal-JIT compiles only those methods that are called at runtime. These methods are compiled the first time they are called, and then they are stored in cache. When the same methods are called again, the compiled code from cache is used for execution.

The security policy settings are referred at the compilation stage. If the code is not type-safe, the JIT process is aborted and an exception is raised. This type safety is ensured during compilation using JIT.

Overall the role of a JIT compiler is to bring higher performance by placing the compiled code in cache so that when the next call is made to the same method or procedure, it gets executed at a faster speed.

Additional Concepts in .NET Framework

This section will cover the some of the namespaces found in the .NET Framework class library. It will also give an introduction to the concept of assemblies in the .NET Framework and explain the terms managed and unmanaged code.

### Introduction to Namespaces

Microsoft .NET Framework includes a rich base class library that contains classes, interfaces, and value types that accelerate the development process. This class library provides access to system functionality. To ensure interoperability between languages, the classes in the .NET Framework class library are CLS compliant. Thus, they can be used from any programming language targeted for the Microsoft .NET Framework that complies with the CLS specifications.

Microsoft .NET introduces a concept of namespaces that is similar to packages in Java. Namespaces are logical groupings of functionally related classes. They also help avoid collisions when referring to classes with same name.

The .NET Framework class library contains classes, which perform the following functions:

- Representing base data types
- Abstracting common exceptions
- File I/O handling
- XML handling
- Database handling
- Accessing information about loaded types through reflection
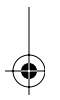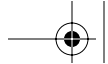
It also contains security related classes, Windows Forms and Web Forms, rich Windows controls, and rich server-side controls for ASP and provides a rich set of interfaces. There are many abstract and concrete classes provided in the class library. User-defined classes can implement any number of the interfaces provided by the .NET Framework class library.

Figure 1-4 shows the Framework class library provided by Microsoft .NET Framework.

The **System** namespace is the root namespace for built-in types provided by the Microsoft .NET Framework. This namespace includes classes that represent the base data types. Any language supported in the .NET Framework can use these base data types. The class **Object** is the root class in the inheritance hierarchy of the **System** namespace. **Int16**, **Int32**, **Byte**, **Boolean**, **Single**,

Figure 1-4   *Microsoft .NET Framework class library.*

Double, **Char**, **Array**, and **String** are some examples of members in the **System** namespace. It is important to note that many of these types correspond to the primitive data types, which are used in programming languages such as Visual Basic, Visual C++, and others.

Functionalities offered by some of the commonly used and important namespaces, which can be used by developer and would be helpful during migration, are as follows:

- **System.Runtime.InteropServices** contains classes for interoperability with COM and other unmanaged code. The classes perform all the data marshalling and all other plumbing work required for interoperating with the unmanaged code. This namespace is important during migration because only the main application needs to be migrated to Visual Basic .NET. The COM components can be used as they are through the interoperability mechanism offered by the classes in this namespace.

- **Microsoft.VisualBasic.Compatibility** contains classes that are not used in Visual Basic .NET but are preserved so that elements of Visual Basic 6.0 can still be used in Visual Basic .NET during the migration process.

- **DirListBox**, **DriveListBox**, and **FileListBox** are some of the commonly used controls in Visual Basic 6.0 forms that get upgraded to Visual Basic .NET with the help of this namespace.

- **System.Runtime.Remoting** contains classes for creating and configuring distributed applications.

- **System.IO** contains classes for basic data stream access and management, including file and memory I/O.

- **System.Reflection** contains functionality to access type metadata and dynamic creation and invocation of types.

- **System.Net** contains classes for sending and receiving data over a network for commonly used network protocols.

- **System.Threading** contains classes for multithreaded programming support, including locking and synchronization.

- **System.Security** contains classes for accessing the .NET Framework security system, including policy resolution, stack walks, and permissions.

- **System.Collections** contains classes for collections of objects, such as lists, queues, arrays, hash tables, and dictionaries.

- **System.Drawing** contains classes for rich two-dimensional graphics functionality and access to GDI+ functionality. Classes provided in this namespace replace graphics functionality offered by individual controls in Visual Basic. More advanced functionality is provided in the **System.Drawing.Drawing2D**, **System.Drawing.Imaging**, and **System.Drawing.Text** namespaces.

- **System.Windows.Forms** provides classes for rich user-interface features for Windows-based applications. Classes in this namespace replace the Visual Basic Form object and other control objects.
- **System.Web** contains classes for developing Web applications. Classes in this namespace provide core infrastructure for ASP .NET, including Web Forms support.
- **System.Web.Services** contains classes for developing SOAP-based Web services and clients for consuming these Web services.
- **System.Data** offers a complete set of functionality for all database-related operations. It contains nested namespaces. One namespace, **System.Data.SqlClient**, provides database-related operations for SQL Server. The other namespace, **System.Data.OleDb**, deals with other kinds of databases. The **System.Data.OleDb** namespace provides classes for native data types within SQL Server. These classes provide a safer, faster alternative to other data types.
- **System.XML** contains classes for creating and processing XML documents.

Understanding Assemblies in the .NET Framework

An assembly is the unit of deployment in the .NET Framework. It contains code that can be executed in the CLR. Figure 1-5 shows the contents of a typical assembly in the .NET Framework. As seen in the figure, an assembly con-
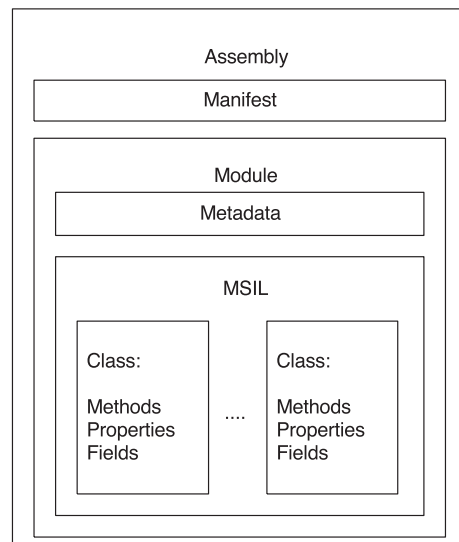
**Figure 1-5**   *Assembly in Microsoft .NET.*

tains manifest data and one or more modules. Manifest data contains information about the assembly and other lists of assemblies that it depends on. It also contains all the publicly exposed types and resources. An assembly contains various modules. Each module contains metadata and Intermediate Language (IL). In Microsoft .NET Module can either be an EXE file or a DLL. Manifest in an assembly contains list of dependent assemblies and types and resources exposed by assembly.

Assemblies can be static as well as dynamic. Static assemblies are stored on the disk as portable executable (PE) files. Dynamic assemblies are created at runtime and can be executed directly from the memory. They need not be saved to the hard disk.

The scope of an assembly can be made specific to a particular application by copying the assembly in the application's directory structure. The assembly can also be made global so that all other applications can make use of this assembly. To make the assembly global, it has to be put into the global assembly cache. This is achieved with the help of a global assembly cache tool (**gacutil.exe**) provided by the .NET Framework.

Microsoft .NET Framework SDK provides a tool called MSIL Disassembler (**ILDasm**.**exe**) to view the MSIL. The Intermediate Language Disassembler (**ILDasm**.**exe**) allows the developer to load any Microsoft .NET assembly (EXE or DLL) and investigate its contents (including the associated manifest, IL instruction set and type metadata).

The folder **AddClass** for this chapter contains a small application using Visual Basic .NET.

Code
Example
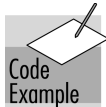
```
Public Class AddClass

    Public Shared Sub Main()
       Console.WriteLine("Addition of {0} and {1} is _
          {2}",10, 20, Add(10, 20))
    End Sub

    Private Shared Function Add(ByVal iOperand1 As _
       Integer, ByVal iOperand2 As Integer) As Integer
       Return (iOperand1 + iOperand2)
    End Function
End Class
```

In this source code a class named **AddClass** has been defined. There are two elements in this class: public method **Main** and private function **Add**. This code adds two integers: **iOperand1** and **iOperand2**. The source code file named **AddClass.vb** has been created and compiled in Visual Studio .NET using the Visual Basic .NET compiler. On the command line **AddClass.vb** can be compiled using **vbc AddClass.vb** command. This compilation process
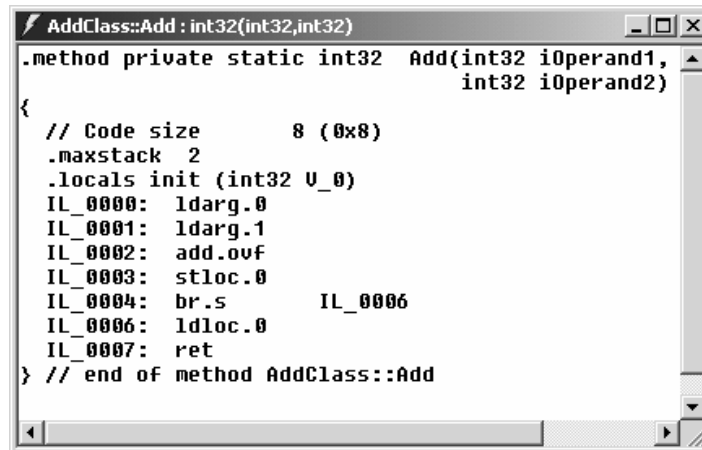
16   Chapter 1 • **Microsoft .NET Framework**

Figure 1-6   **ILDasm** *utility for* **AddClass.exe** *file.*

generates an assembly. To view the contents of an assembly, the **ILDasm.exe** utility is used as shown in Figure 1-6.

As seen in Figure 1-6, the structure of the assembly is presented in a familiar tree view format. Methods, properties, and so on for a given type are identified by a specific icon. The screen in Figure 1-6 shows that the assembly **AddClass.exe** contains a default constructor and two static methods named **Main** and **Add**. To check the MSIL for the method **Add**, double-click the **Add** method. This opens up a separate window displaying the MSIL code as shown in Figure 1-7.
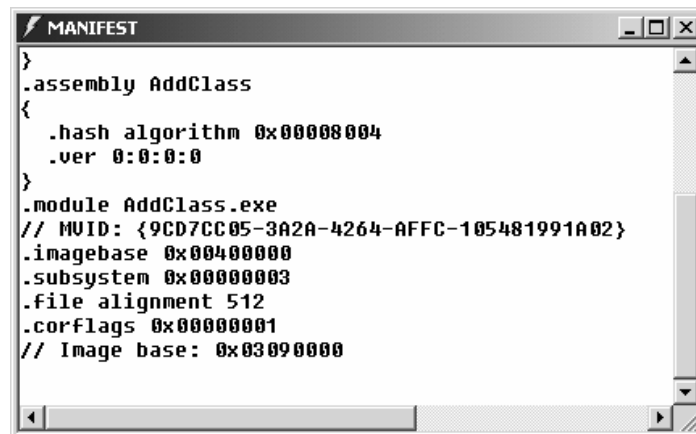
It is important to note that the same MSIL code is generated irrespective of the language that has been used to develop the application. Thus all Microsoft .NET languages have the same capabilities, and the language used for development depends on the preferences of the developer. In addition, the manifest information for the assembly can be viewed by double-clicking on the Manifest link in the main **ILDasm** utility. Figure 1-8 shows the information for the **AddClass.exe** assembly that was created in the previous step.

As shown in Figure 1.8, the assembly **AddClass.exe** references the assembly **mscorlib.dll**, which contains the base class library DLL. Although the entire base class library is broken into several assemblies, the primary binary is **mscorlib.dll**. This assembly contains a large number of core types, and it provides common programming tasks needed to build .NET applications.

```
AddClass::Add : int32(int32,int32)                    _|□|×|
.method private static int32   Add(int32 iOperand1,   ▲
                                   int32 iOperand2)

{
  // Code size       8 (0x8)
  .maxstack  2
  .locals init (int32 V_0)
  IL_0000:  ldarg.0
  IL_0001:  ldarg.1
  IL_0002:  add.ovf
  IL_0003:  stloc.0
  IL_0004:  br.s         IL_0006
  IL_0006:  ldloc.0
  IL_0007:  ret
} // end of method AddClass::Add                      ▼
◄|                                            ►|
```

Figure 1-7   *MSIL code for **Add** method in **AddClass**.*

```
MANIFEST                                              _|□|×|
}                                                     ▲
.assembly AddClass

{
  .hash algorithm 0x00008004
  .ver 0:0:0:0
}
.module AddClass.exe
// MVID: {9CD7CC05-3A2A-4264-AFFC-105481991A02}
.imagebase 0x00400000
.subsystem 0x00000003
.file alignment 512
.corflags 0x00000001
// Image base: 0x03090000                             ▼
◄|                                            ►|
```
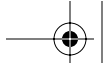
Figure 1-8   *Manifest Information for **AddClass**.*

MSIL is a language in its own right. The tool developers guide provided with the Microsoft .NET SDK provides information on the Common Language Infrastructure (CLI). The documentation also provides comprehensive reference documentation for the IL instruction set.

The MSIL code can be stored in file with a .IL extension. This file can be converted into a portable executable file by using a utility called **ILASM.EXE**. This utility has been provided with Microsoft .NET SDK, and it generates a PE file from MSIL assembly language.

Introduction to Managed and Unmanaged Code

One of the most important concepts in the .NET Framework is managed code, which is code that runs under the supervision of the CLR. The advantage of managed code is that the CLR provides services to managed code such as automatic memory management through a garbage collection mechanism, easy versioning and deployment capabilities, and better security options.

Unmanaged code does not run under the supervision of the CLR. It has to implement memory allocation and deallocation logic. Unmanaged code can be generated in Microsoft .NET by compiling code with the /unsafe option during the compilation process.

During execution of a .NET application, managed code can interact with send and receive data from unmanaged code. This process of passing data between managed and unmanaged code is known as data marshalling. For Microsoft .NET applications that use classic COM components, the classes provided in the **System.Runtime.Interop** namespace do the data marshalling. The application developer does not have to write code explicitly for the data marshalling in this case.

To call into unmanaged code from managed code, a runtime callable wrapper (**RCW**) is created. This, however, adds some performance penalty. Similarly, calling into managed code from unmanaged code requires a COM callable wrapper (**CCW**) to be generated.

Consideration for Migration

Microsoft .NET Framework interoperability features encourage developers and organizations to continue using existing COM applications, Win32 DLL's, and ASP applications. Further, the interoperability features also allow the use of existing unmanaged code, which usually is in the form of COM components and Win32 DLLs to be used by the .NET applications.

In many of the scenarios where application code is packed into COM components, the existing code base is interoperable with the .NET applications. It means that developers can enhance applications using .NET while making use of the existing code base by utilizing the COM interoperability options provided by .NET Framework. For new projects, they should consider developing the entire project using .NET Framework. On the other hand, if an existing application addresses the business needs of an organization and is stable, it's better not to consider those applications for migration. Developers often have a tough time opting between a migration and interoperability and more importantly when to choose interoperability options and when to go in for migration.

We have seen in earlier sections of this chapter that Microsoft .NET is definitely a decent improvement over COM technology and is considered more compact. It brings us nearer to true code reusability. However, many

organizations have probably invested huge amounts in building COM compo-
nents in the last decade. COM has been in the market for almost 8 years now.
Also, the last decade saw a substantial growth of the information technology
(IT) industry in various segments, which means that there has been a lot of
code packed in the form of components. So, it is impracticable to rewrite the
entire code into .NET components, however good the components may be
for designing more robust applications. Existing applications using COM com-
ponents have already been been tested, and a shift to .NET components
would mean that this testing would have to be performed against the newly
developed applications again. So, there is a great need for interoperability
between the COM and .NET components.

However, there are scenarios wherein we need to consider existing
applications to be moved to .NET. Let's presume that a method call in our
applications only sets a value of a certain property or does reasonably a small
amount of work; then the overhead of interoperability would be bit heavy.
Usually, the overhead of calling from managed code to unmanaged code via
.NET COM interoperability options is insignificant. In cases where there is an
intensive usage of **Get** and **Set** methods it's better considering rewriting of the
application using any of the .NET compliant languages rather than choosing
interoperability; otherwise there would heavy performance penalties.

With Visual Studio .NET the development, maintenance, and deploy-
ment of applications becomes much easier and faster. As such, the .NET
development environment is a notable improvement over the COM-based
development model for writing distributed applications. If most clients of
your existing components will be written in managed code, you should
consider either migrating your component to managed code or writing a
managed wrapper around it.

There are certain other things that you need to take into consideration
while migrating applications to .NET. Most of these things vary from case to case.
However, there are obvious things that can be discussed in general. One is
choosing an operating system. The choice of an operating system will affect the
migration strategy to be followed. Choosing an operating system like Win 2000,
Win XP, or Windows .NET can reap maximum benefits of the .NET Framework.
Although the .NET Framework works on Windows 98, Windows NT, and Win-
dows ME, these platforms don't offer complete access .NET features. So the
choice of operating systems should go along with the migration plan.

The object-oriented features available in .NET also make it a favorite
choice of many designers to consider moving their existing applications to
.NET. The reliance of .NET on open standards such as CLI, XML, and SOAP
and rich enhancement to the ASP model also makes it a good candidate for
migrating existing applications to .NET. Knowing about code reusability and
COM interoperability for migration to .NET is of special importance because it
not only helps developers decide how to migrate, but it also helps designers
and architects deliver better extensible, integral, and interoperable systems.
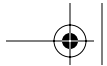
Roadmap to Migration

Because this book deals with technical aspects of migration to Visual Basic
.NET, ASP.NET, and Visual C++ .NET, not all of the steps presented in this sec-
tion are discussed in detail in subsequent chapters; this section deals with the
generic process approach rather than the technical approach. To have proper
management of the migration project, the migration team should follow the
basic steps presented in this section.

Migration from one technology to another involves four major steps,
which ensure smooth migration to the new technology. This section takes a
look at the steps involved in any migration process. They are applicable to all
three migrations covered in this book: Visual Basic to Visual Basic .NET, ASP
to ASP.NET, and Visual C++ to Visual C++ .NET.

Subsequent sections contain brief descriptions of various phases of the
methodology in addition to the tasks and activities for each phase.

Phases Involved in Migration

The methodology presented here supports the full spectrum of activities that
constitute a migration project. It provides guidelines for various phases.

Depending upon the project type, some of the activities and tasks can
be executed simultaneously. Sequencing and timings of these tasks and activ-
ities should be defined during the planning stage of the migration project.

TABLE 1–1	*Phases of Migration*
Phase	**Activities**
Phase I: Assessment	Study and analyze existing system Establish design goals for the new system Do a cost-benefit/risk analysis
Phase II: Reverse Engineering	Recover software design artifacts Define functional specification for the new system
Phase III: Forward Engineering	Design the new system Apply migration changes to the existing application Construct the new system Test the new system
Phase IV: Installation	Develop necessary operation manuals and procedures Install new hardware and software environment Execute production trial run

It is important to note that the migration methodology can be custom-
ized according to

- Existing system type
- Migration goals

- Required deliverables
- Staff experience
- Size of the project
- Complexity of the project

This methodology should not be viewed as a set of rigid procedures. Each phase, task, and activity need not be executed for all the projects. The process of customization requires selecting, deleting, adding, combining, and sequencing various phases, tasks, and activities. It is necessary to update migration methodology as a result of ongoing project experience. We will look at each of the features in the following sections.

Phase I—Assessment

Assessment is the first phase of a migration project. It involves developing an understanding of the existing system. This can be carried out through various activities such as interviews, application demonstration, meeting with system experts, and evaluation of proposed (new) system architecture. This will help in deciding the migration path from an existing system to a new system based on the methodology provided in this document. It involves preparing a detailed migration plan.

Assessment activities can be classified into three categories:

1. Understanding the existing system and its development environment. This is achieved through

   - Review of existing system documentation
   - Interviews, meetings, and application demonstration with system experts (such as business leaders, developers and maintainers, users, etc.)
   - Review of system history records (change log, error log, maintenance records)

2. Analysis and decision making. Based on the analysis of the existing system, business goals and objectives, customer needs, and migration recommendations, the following decisions are made:

   - Migration goals and objectives
   - Scope and extent of migration efforts
   - Migration strategies and technical approach
   - Development environment and architecture of new system
   - Critical success factors for the migration effort

3. Planning. Once the decision to migrate the existing system is affirmed, the following plans for the migration efforts are prepared:

   - Reverse-engineering plan
   - Forward-engineering (development) plan
   - Test plan
   - Configuration management plan

- Data conversion plan
- Installation and cutover plan
- Training plan

Phase II—Reverse Engineering

The purpose of reverse engineering is to recover and reconstruct software design artifacts such as DFDs, business and validation rules, and key data elements of the existing system. Types of the artifacts to be recovered and the effort involved depend upon the goals and objectives of a migration project and also upon the gap between existing system documentation (such as functional specification, design document, etc.) and the running system.

The following steps will form a major part of this phase:

1. Carry a program-level code walk-through and detailed analysis. The following factors would be evaluated during this phase:

   - Module-level functional description
   - Functional hierarchy
   - Program control flow diagrams
   - Data flow diagrams at each level
   - Business rules
   - Cross-references
   - Entity relationship diagram
   - Handling of locks in each module
   - Error and help message handling
   - Identifying common libraries and functions
   - Identifying components and Active X controls
   - Data source details
   - Mapping between application domain entities and data source
   - Help files and lookups
   - Informal comments and observations.

2. Based on information recovered from the program-level code walk-through, regenerate functional specification document and other higher level software artifacts of the current system.

3. Create the functional specification for the new system by

   - Adding new requirements to recovered functional specification
   - Remove obsolete functions

Phase III—Forward Engineering

The aim of the forward engineering phase is to design, develop, and test the new system and migrate the existing system. The design artifacts of the existing system recovered in the reverse engineering phase and the functional specification of the new system form the major inputs to this phase.

The selection of a software development life-cycle model (Waterfall, RAD, Prototype, etc.), which depends upon the type of the application, will affect the activities involved in this phase.

Activities given below are quite generalized and should be tuned accordingly:
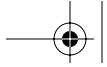
1. Design the new system; it will include following activities:

    - Prototype design
    - GUI design
    - Database design
    - Module-level design
    - Program design
    - Unit test cases design
    - Integration test cases design
    - System test cases design

2. Migrate the existing application code to newer programming language. For Visual Basic this will involve migrating Visual Basic 6.0 code to Visual Basic .NET and will include following activities:

    - Pre-migration changes in original source code as recommended in this book
    - A run through of the upgrade wizard (for Visual Basic code)
    - Post-migration changes as presented in this book

    For Visual C++ and ASP applications there is no concept of an upgrade wizard and hence there is no clear demarcation such as pre- and post-migration.

3. Construct the new system. It will include following activities:

    - Coding
    - Other QA activities such as code walk-through

4. Test the new system, including

    - Unit and module testing
    - Integration testing
    - System and acceptance testing

Phase IV—Installation and Release

The installation and release phase identifies all the installation-related activities and procedures, and sequences and schedules these activities in an installation plan. The installation plan should be prepared much earlier in a migration project and installation progress should be reviewed periodically.

Release involves putting the new system into operation. Before moving into operation, it is important to have a trial run and evaluate the results. If required, the trial run should be repeated until the desired results are achieved. The new system should become operational only after it is acceptable to all the users.

This phase will involve following steps:

1. Procure and install the new operating hardware and software
2. Review results of UAT trial run and initiate corrective action
3. Release new applications

Summary

The Microsoft .NET development platform is a new programming environment. This chapter gives a brief introduction to the Microsoft .NET Framework. The .NET Framework can be used to develop console applications, rich Windows GUI applications, Web applications, Web services, and Windows services applications.

Major limitations of previous Microsoft technologies, such as COM registration, lack of language interoperability, and so on, have been eliminated in Microsoft .NET Framework. Cross-language interoperability has been achieved with the help of the CLR, CTS, and CLS.

The CLR is an execution environment for applications written in .NET Framework. It offers various runtime services such as automatic garbage collection, code security, and scalability. Now Microsoft .NET Framework contains a rich base class library. The classes in this library are accessible by all languages targeted for the CLR. The .NET Framework class library contains classes that offer commonly used functionalities such as Windows and Web Forms, database handling, XML-related operations, file I/O operations, network-related operations, and security related operations.

The code targeted for Microsoft .NET platform produces MSIL, which is just-in-time compiled to generate native code. MSIL is known as the language of the CLR; it is a language in its own right. A developer can develop applications using MSIL. Language vendors can make their language .NET compliant by providing a compiler that converts the source code into MSIL.

Various considerations for migration have been discussed in this chapter. However, the decision to migrate or use the interoperability mechanism varies on case-to-case basis depending on architecture of the application.

The methodology presented in this chapter can be applied to any migration. Migration from Visual Basic to Visual Basic .NET, ASP to ASP.NET, and Visual C++ to Visual C++ .NET will have four stages: assessment, reverse engineering, forward engineering, and installation. However, individual steps involved in each of these stages will differ for these three technologies.