1

What Is Microsoft .NET?

New technologies force change, nowhere more so than in computers and software. Occasionally, a new technology is so innovative that it forces us to challenge our most fundamental assumptions. In the computing industry, the latest such technology is the Internet. It has forced us to rethink how software should be created, deployed, and used.

However, that process takes time. Usually, when a powerful new technology comes along, it is first simply strapped onto existing platforms. So it has been for the Internet. Before the advent of Microsoft .NET, we used older platforms with new Internet capabilities "strapped on". The resulting systems worked, but they were expensive and difficult to produce, hard to use, and difficult to maintain.

Realizing this several years ago, Microsoft decided it was time to design a new platform from the ground up specifically for the post-Internet world. The result is called *.NET*. It represents a turning point in the world of Windows software for Microsoft platforms. Microsoft has staked their future on .NET, and publicly stated that henceforth almost all their research and development will be done on this platform. It is expected that, eventually, almost all Microsoft products will be ported to the .NET platform. (However, the name ".NET" will evolve, as we will see at the end of the chapter.)

What Is .NET?

Microsoft's .NET initiative is broad-based and very ambitious. It includes the *.NET Framework*, which encompasses the languages and execution platform, plus extensive class libraries providing rich built-in functionality. Besides the core .NET Framework, the .NET initiative includes protocols (such as the *Simple Object Access Protocol*, commonly known as *SOAP*) to provide a new level of software integration over the Internet, via a standard known as Web services.

Although Web services are important (and are discussed in detail in Chapter 22), the foundation of all .NET-based systems is the .NET Framework. This chapter will look at the .NET Framework from

the viewpoint of a Visual Basic developer. Unless you are quite familiar with the Framework already, you should consider this introduction an essential first step in assimilating the information about Visual Basic .NET that will be presented in the rest of this book.

The first released product based on the .NET Framework was *Visual Studio .NET 2002*, which was publicly launched in February of 2002, and included version 1.0 of the .NET Framework. The current version is *Visual Studio .NET 2003*, which was introduced a year later, and included version 1.1 of the .NET Framework. This book assumes you are using VS.NET 2003, but almost all of the examples will work transparently with VS.NET 2002 because the differences in the two versions are minor.

A Broad and Deep Platform for the Future

Calling the .NET Framework a *platform* doesn't begin to describe how broad and deep it is. It encompasses a virtual machine that abstracts away much of the Windows API from development. It includes a class library with more functionality than any yet created. It makes available a development environment that spans multiple languages, and it exposes an architecture that makes multiple language integration simple and straightforward.

At first glance, some aspects of .NET appear similar to previous architectures, such as UCSD Pascal and Java. No doubt some of the ideas for .NET were inspired by these past efforts, but there are also many brand new architectural ideas in .NET. Overall, the result is a radically new approach to software development.

The vision of Microsoft .NET is globally distributed systems, using XML as the universal glue to allow functions running on different computers across an organization or across the world to come together in a single application. In this vision, systems from servers to wireless palmtops, with everything in between, will share the same general platform, with versions of .NET available for all of them, and with each of them able to integrate transparently with the others.

This does not leave out classic applications as we have always known them, though. Microsoft .NET also aims to make traditional business applications much easier to develop and deploy. Some of the technologies of the .NET Framework, such as Windows Forms, demonstrate that Microsoft has not forgotten the traditional business developer. In fact, such developers will find it possible to Internet-enable their applications more easily than with any previous platform.

What's Wrong with DNA and COM?

The pre-.NET technologies used for development on Microsoft platforms encompassed the *COM (Component Object Model) standard* for creation of components, and the *DNA model* for multitier software architectures. As these technologies were extended into larger, more enterprise-level settings, and as integration with the Internet began to be important, several major drawbacks became apparent. These included:

- Difficulty in integrating Internet technologies:
  - Hard to produce Internet-based user interfaces
  - No standard way for systems and processes to communicate over the Internet

- Expensive, difficult, and undependable deployment
- Poor cross-language integration

- Weaknesses in the most popular Microsoft tool — Visual Basic:
  - Lack of full object orientation, which made it impossible to produce frameworks in Visual Basic
  - One threading model that did not work in some contexts
  - Poor integration with the Internet
- Other weaknesses such as poor error handling capabilities

It is important to note that all pre-.NET platforms, such as Java, also have some of these drawbacks, as well as unique ones of their own. The drawbacks related to the Internet are particularly ubiquitous.

Let's take a brief look at these drawbacks to pre-.NET Microsoft technologies before taking up how .NET addresses them.

Difficulty in Integrating Internet Technologies

Starting in late 1995, Microsoft made a dramatic shift toward the Internet. They had to make some serious compromises to quickly produce Internet-based tools and technologies. The main result, *Active Server Pages (ASP)*, was a tool that was not oriented around structured and object-oriented development. Having to design, debug, and maintain such unstructured ASP code is also a headache. While many viable systems were produced with ASP pages, these obvious flaws needed to be addressed.

Later in the evolution of the Internet, it became apparent that communicating with the user via HTTP and HTML was limiting. To get, for example, a stock quote from an Internet server, it was often necessary for a program to pretend to be a user, get an HTML page, and then take it apart to get the information needed. This was fussy development, and the result was quite brittle because of the possibility that the format of the page might change and, thus, need new parsing logic.

Developers needed a standard way for *processes* to communicate over the Internet, rather than the communication being directed only at *users*. DNA and COM lacked any such standard.

Deployment Issues

Microsoft's COM standard was developed for use on small systems with limited memory running Microsoft Windows. The design trade-offs for COM were oriented around sharing memory, and quick performance on hardware we would now consider slow.

This meant that *Dynamic Link Libraries (DLLs)* were shared between applications to save memory, and a binary interface standard was used to ensure good performance. To quickly find the components needed to run an application, DLLs had to register their class IDs to the local Windows Registry.

Besides the registration logistics needed to make DLLs work at all, COM components could be rendered inoperable by versioning issues. The resulting morass of problems related to versioning was colloquially known as "DLL Hell."

The need to register components locally also resulted in other limitations. It was not possible for a COM application to be placed on a CD-ROM or a network drive, and then run from that location without an installation procedure.

Poor Cross-Language Integration

COM/DNA typically required the use of three separate development models. Business components were most often written in Visual Basic, and Visual Basic could also be used for local Win32 user interfaces. Browser-based user interfaces required ASP. System components sometimes required the use of C++.

Each of these languages had difficulties integrating with the others. Getting VB strings properly transferred to and from C++ routines is a challenge. For example, ASP pages required a COM interface with only `Variant`s for data, which negated the strong typing available in Visual Basic and C++. Getting all three languages to work together required several arcane skills.

Weaknesses in Visual Basic in COM/DNA Applications

Visual Basic 6 (VB6) (and earlier versions) was easily the most popular language for developing applications with the DNA model. As noted above, it can be used in two major roles — forms-based VB clients and COM components (on either the client or the server).

There are other options, of course, including C++, J++, and various third-party languages such as Delphi and Perl, but the number of VB developers outnumbers them all put together.

Despite its popularity, VB6 suffered from a number of limitations in the COM/DNA environment. Some of the most serious limitations include:

- No capability for multithreading
- Lack of implementation inheritance and other object-oriented features
- Poor error handling ability
- Poor integration with other languages such as C++ (as discussed above)
- No effective user interface for Internet-based applications

Lack of multithreading implies, for example, that VB6 can't be used "out of the box" to write an NT-type service. There are also situations in which the apartment threading used by components created in Visual Basic limits performance.

VB6's limited object-oriented features, in particular the lack of inheritance, made it unsuitable for development of object-based frameworks, and denied design options to VB6 developers that were available to C++ or Java developers.

VB6's archaic error handling becomes especially annoying in a multitier environment. It is difficult in VB6 to track and pass errors through a stack of component interfaces.

Perhaps the biggest drawback to using VB6 became apparent when many developers moved to the Internet. While VB forms for a Win32 client were state-of-the-art, for applications with a browser interface, VB6 was relegated mostly to use in components because it did not have an effective way to do user interfaces for the Web.

Microsoft tried to address this problem in VB6 with Web classes and DHTML pages. Neither caught on because of their inherent limitations.

All of these limitations needed to be addressed, but Microsoft decided to look beyond just Visual Basic and solve these problems on a more global level. All of these limitations are solved in VB.NET through the use of technology in the .NET Framework.

An Overview of the .NET Framework

First and foremost, .NET is a framework that covers all the layers of software development above the operating system. It provides the richest level of integration among presentation technologies, component technologies, and data technologies ever seen on a Microsoft, or perhaps any, platform. Secondly, the entire architecture has been created to make it as easy to develop Internet applications, as it is to develop for the desktop.

The .NET Framework actually "wraps" the operating system, insulating software developed with .NET from most operating system specifics such as file handling and memory allocation. This prepares for a possible future in which the software developed for .NET is portable to a wide variety of hardware and operating system foundations.

VS.NET supports Windows 2003, Windows XP, and all versions of Windows 2000. Programs created for .NET can also run under Windows NT, Windows 98, and Windows Me, though VS.NET does not run on these systems.

The major components of the Microsoft .NET Framework are shown in Figure 1-1.

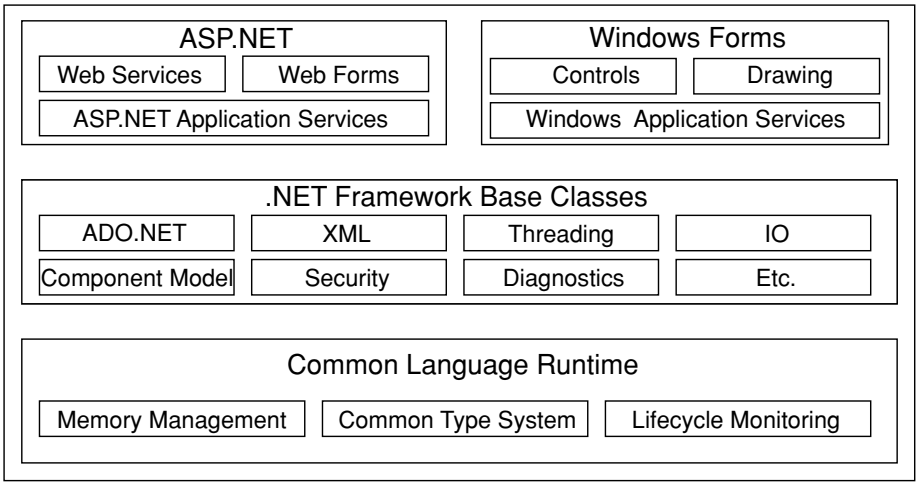

Figure 1-1

The framework starts all the way down at the memory management and component loading level, and goes all the way up to multiple ways of rendering user and program interfaces. In between, there are layers that provide just about any system-level capability that a developer would need.

At the base is the *common language runtime*, often abbreviated to *CLR*. This is the heart of the .NET Framework — it is the engine that drives key functionality. It includes, for example, a common system of datatypes. These common types, plus a standard interface convention, make cross-language inheritance possible. In addition to allocation and management of memory, the CLR also does reference tracking for objects, and handles garbage collection.

The middle layer includes the next generation of standard system services such as classes that manage data and XML. These services are brought under control of the framework, making them universally available and making their usage consistent across languages.

The top layer includes user and program interfaces. *Windows Forms* is a new and more advanced way to do standard Win32 screens (often referred to as "'smart clients"). *Web Forms* provides a new Web-based user interface. Perhaps the most revolutionary is *Web services*, which provides a mechanism for programs to communicate over the Internet, using SOAP. Web services provide an analog of COM and DCOM for object brokering and interfacing, but based on Internet technologies so that allowance is made even for integration to non-Microsoft platforms. Web Forms and Web services, which comprise the Internet interface portion of .NET, are implemented by a part of the .NET Framework referred to as *ASP.NET*.

All of these capabilities are available to any language that is based on the .NET platform, including, of course, VB.NET.

The Common Language Runtime

We are all familiar with runtimes — they go back further than DOS languages. However, the *common language runtime* (CLR) is as advanced over traditional runtimes as a machine gun is over a musket. Figure 1-2 shows a quick diagrammatic summary of the major pieces of the CLR.

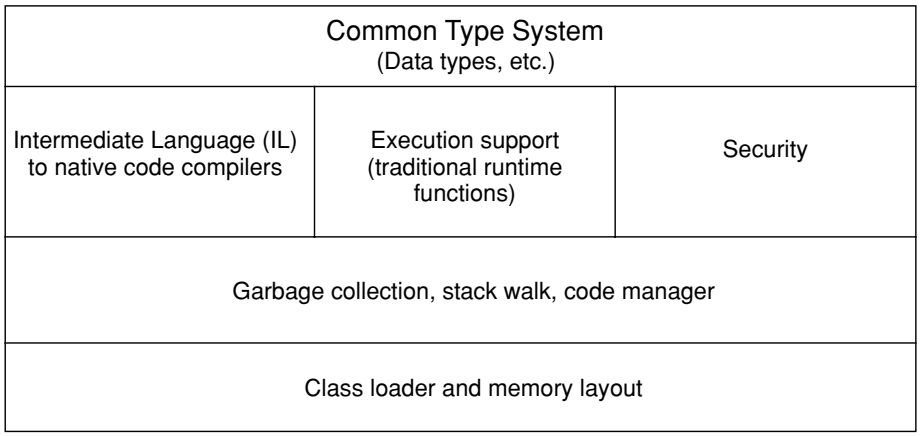

Figure 1-2

That small part in the middle of Figure 1-2 called Execution support contains most of the capabilities normally associated with a language runtime (such as the `VBRUNxxx.DLL` runtime used with Visual Basic). The rest is new, at least for Microsoft platforms.

Key Design Goals

The design of the CLR is based on the following primary goals:

- Simpler, faster development
- Automatic handling of system-level tasks such as memory management and process communication
- Excellent tool support
- Simpler, safer deployment
- Scalability

Notice that many of these design goals directly address the limitations of COM/DNA. Let's look at some of these in detail.

Simpler, Faster Development

A broad, consistent framework allows developers to write less code, and reuse code more. Less code is possible because the system provides a rich set of underlying functionality. Programs in .NET access this functionality in a standard, consistent way, requiring less "hardwiring" and customization logic to interface with the functionality than is typically needed today.

Programming is also simpler in .NET because of the standardization of datatypes and interface conventions. As will be discussed later, .NET makes knowledge of the intricacies of COM much less important.

The net result is that programs written in VB.NET that take proper advantage of the full capabilities of the .NET Framework typically have significantly less code than equivalent programs written in earlier versions of Visual Basic. Less code means faster development, fewer bugs, and easier maintenance.

Excellent Tool Support

Although much of what the CLR does is similar to operating system functionality, it is very much designed to support development languages. It furnishes a rich set of object models that are useful to tools like designers, wizards, debuggers, and profilers, and since the object models are at the runtime-level, such tools can be designed to work across all languages that use the CLR. It is expected that third parties will produce a host of such tools.

Simpler, Safer Deployment

It is hard for an experienced Windows component developer to see how anything can work without registration, GUIDs, and the like, but the CLR does. Applications produced in the .NET Framework can be designed to install with a simple XCOPY. That's right—just copy the files onto the disk and run the application. This hasn't been seen in the Microsoft world since the days of DOS (and some of us really miss it).

This works because compilers in the .NET Framework embed identifiers (in the form of *metadata*, to be discussed later) into compiled modules, and the CLR manages those identifiers automatically. The identifiers provide all the information needed to load and run modules, and to locate related modules.

As a great by-product, the CLR can manage multiple versions of the same component (even a shared component), and have them run side by side. The identifiers tell the CLR which version is needed for a

particular compiled module because such information is captured at compile time. The runtime policy can be set in a module to use the exact version of a component that was available at compile time, to use the latest compatible version, or to specify an exact version. The bottom line is that .NET is intended to eradicate DLL Hell once and for all.

This has implications that might not be apparent at first. For example, if a program needs to run directly from a CD or a shared network drive (without first running an installation program), that was not feasible in Visual Basic after version 3. That capability reappears with VB.NET. This dramatically reduces the cost of deployment in many common scenarios.

Another significant deployment benefit in .NET is that applications only need to install their own core logic. An application produced in .NET does not need to install a runtime, for example, or modules for ADO or XML. Such base functionality is part of the .NET Framework, which is installed separately and only once for each system. The .NET Framework will eventually be included with the operating system and probably with various applications. Those four-disk installs for a VB "Hello world" program will be a thing of the past.

The .NET Framework, which includes the CLR and the Framework base classes, is required on every machine where you want to run .NET applications and code. For Windows 2003 and above, the .NET Framework is installed automatically as part of the operating systems. For older operating systems, the .NET Framework is a separate installation. Deployment of .NET applications is discussed in Chapter 25.

Scalability

Since most of the system-level execution functions are concentrated in the CLR, they can be optimized and architected to allow a wide range of scalability for applications produced in the .NET Framework. As with most of the other advantages of the CLR, this one comes to all applications with little or no effort.

Memory and process management is one area where scalability can be built in. The memory management in the CLR is self-configuring and tunes itself automatically. Garbage collection (reclaiming memory that is no longer being actively used) is highly optimized, and the CLR supports many of the component management capabilities of MTS/COM+ (such as object pooling). The result is that components can run faster, and thus support more users.

This has some interesting side effects. For example, the performance and scalability differences among languages become smaller. All languages compile to a standard bytecode called *Microsoft Intermediate Language (MSIL)*, often referred to simply as *IL*, and there is a discussion later on how the CLR executes IL. With all languages compiling down to similar bytecode, it becomes unnecessary in most cases to look to other languages when performance is an issue. The difference in performance among .NET languages is minor — Visual Basic, for example, gives about the same performance as any of the other .NET languages.

Versions of the CLR are available on a wide range of devices. The vision is for .NET to be running at all levels, from smart palmtop devices all the way up to Web farms. The same development tools work across the entire range — news that will be appreciated by those who have tried to use older Windows CE development kits.

Metadata

The .NET Framework needs lots of information about an application to carry out several automatic functions. The design of .NET requires applications to carry that information within them. That is, applications are *self-describing*. The collected information that describes an application is called *metadata*.

The concept of metadata is not new. For example, COM components use a form of it called a type library, which contains metadata describing the classes exposed by the component and is used to facilitate OLE Automation. A component's type library, however, is stored in a separate file. In contrast, the metadata in .NET is stored in one place — *inside* the component it describes. Metadata in .NET also contains more information about the component, and is better organized.

Chapter 3 on the CLR goes into more information about metadata. For now, the most important point for you to internalize is that metadata is key to the easy deployment in .NET. When a component is upgraded or moved, the necessary information about the component cannot be left behind. Metadata can never get out of sync with a .NET component because it is not in a separate file. Everything the CLR needs to know to run a component is supplied with the component.

Multiple Language Integration and Support

The CLR is designed to support multiple languages and allow unprecedented levels of integration among those languages. By enforcing a common type system, and by having complete control over interface calls, the CLR allows languages to work together more transparently than ever before. The cross-language integration issues of COM simply don't exist in .NET.

It is straightforward in the .NET Framework to use one language to subclass a class implemented in another. A class written in Visual Basic can inherit from a base class written in C#, or in COBOL for that matter. The VB program doesn't even need to know the language used for the base class. .NET offers full implementation inheritance with no problems requiring recompilation when the base class changes.

Chapter 3 also includes more information on the multiple language integration features of .NET.

A Common Type System

A key piece of functionality that enables multiple language support is a *common type system*, in which all commonly used datatypes, even base types such as `Long` and `Boolean`, are actually implemented as objects. Coercion among types can now be done at a lower level for more consistency between languages. Also, since all languages are using the same library of types, calling one language from another doesn't require type conversion or weird calling conventions.

This results in the need for some readjustment, particularly for VB developers. For example, what we called an `Integer` in VB6 and earlier, is now known as a `Short` in VB.NET. The adjustment is worth it to bring Visual Basic in line with everything else, though, and, as a by-product, other languages get the same support for strings that Visual Basic has always had.

The CLR enforces the requirement that all datatypes satisfy the common type system. This has important implications. For example, it is not possible with the common type system to get the problem known in COM as a buffer overrun, which is the source of many security vulnerabilities. Programs written on .NET

should therefore have fewer such vulnerabilities, because .NET is not dependent on the programmer to constantly check passed parameters for appropriate type and length. Such checking is done by default.

Chapter 4 goes into detail about the new type system in .NET.

Namespaces

One of the most important concepts in Microsoft .NET is *namespaces*. Namespaces help organize object libraries and hierarchies, simplify object references, prevent ambiguity when referring to objects, and control the scope of object identifiers. The namespace for a class allows the CLR to unambiguously identify that class in the available .NET libraries that it can load.

Namespaces are discussed briefly in Chapter 2 and in more detail in Chapter 8. Understanding the concept of a namespace is essential for your progress in .NET, so do not skip those sections if you are unfamiliar with namespaces.

The Next Layer — The .NET Class Framework

The next layer up in the framework provides the services and object models for data, input/output, security, and so forth. It is called the *.NET Class Framework*, sometimes referred to as the *.NET base classes*. For example, the next generation of ADO, called ADO.NET, resides here. Some of the additional functionality in the .NET Class Framework is listed below.

You might be wondering why .NET includes functionality that is, in many cases, duplication of existing class libraries. There are several good reasons:

- The .NET Class Framework libraries are implemented in the .NET Framework, making them easier to integrate with .NET-developed programs.
- The .NET Class Framework brings together most of the system class libraries needed into one location, which increases consistency and convenience.
- The class libraries in the .NET Class Framework are much easier to extend than older class libraries, using the inheritance capabilities in .NET.
- Having the libraries as part of the .NET Framework simplifies deployment of .NET applications. Once the .NET Framework is on a system, individual applications don't need to install base class libraries for functions like data access.

What Is in the .NET Class Framework?

The .NET Class Framework contains literally thousands of classes and interfaces. Here are just some of the functions of various libraries in the .NET Class Framework:

- Data access and manipulation
- Creation and management of threads of execution
- Interfaces from .NET to the outside world — Windows Forms, Web Forms, Web services, and console applications
- Definition, management, and enforcement of application security

- Encryption, disk file I/O, network I/O, serialization of objects, and other system-level functions
- Application configuration
- Working with directory services, event logs, performance counters, message queues, and timers
- Sending and receiving data with a variety of network protocols
- Accessing metadata information stored in assemblies

Much of the functionality that a programmer might think of as being part of a language has been moved to the base classes. For example, the VB keyword `Sqr` for extracting a square root is no longer available in .NET. It has been replaced by the `System.Math.Sqrt()` method in the framework classes.

It's important to emphasize that all languages based on the .NET Framework have these framework classes available. That means that COBOL, for example, can use the same function mentioned above for getting a square root. This makes such base functionality widely available and highly consistent across languages. All calls to `Sqrt` look essentially the same (allowing for syntactical differences among languages) and access the same underlying code. Here are examples in VB.NET and C#:

```
' Example using Sqrt in Visual Basic .NET
Dim dblNumber As Double = 200
Dim dblSquareRoot As Double
dblSquareRoot = System.Math.Sqrt(dblNumber)
Label1.Text = dblSquareRoot.ToString

' Same example in C#
Double dblNumber = 200;
Double dblSquareRoot = System.Math.Sqrt(dblNumber);
dblSquareRoot = System.Math.Sqrt(dblNumber);
label1.Text = dblSquareRoot.ToString;
```

Notice that the line using the `Sqrt()` function is exactly the same in both languages.

> *As a side note, a programming shop can create its own classes for core functionality, such as globally available, already compiled functions. This custom functionality can then be referenced in code the same way as built-in .NET functionality.*

Much of the functionality in the base framework classes resides in a vast namespace called `System`. The `System.Math.Sqrt()` method was just mentioned. Here are a few other examples of the subsections of the `System` namespace, which actually contains dozens of such subcategories:

This list merely begins to hint at the capabilities in the `System` namespace. Some of these namespaces are used in later examples in other chapters throughout the book.

User and Program Interfaces

At the top layer, .NET provides three ways to render and manage user interfaces:

- *Windows Forms*
- *Web Forms*
- *Console applications*

Namespace	What It Contains	Example Classes and Sub-namespaces
`System.Collections`	Creation and management of various types of collections	`Arraylist`, `Hashtable`, `SortedList`
`System.Data`	Classes and types related to basic database management (see Chapter 11 for details)	`DataSet`, `DataTable`, `DataColumn`,
`System.Diagnostics`	Classes to debug an application and to trace the execution of code	`Debug`, `Trace`
`System.IO`	Types which allow reading and writing to files and other data streams	`File`, `FileStream`, `Path`, `StreamReader`, `StreamWriter`
`System.Math`	Members to calculate common mathematical quantities, such as trigonometric and logarithmic functions	`Sqrt` (square root), `Cos` (cosine), `Log` (logarithm), `Min` (minimum)
`System.Reflection`	Capability to inspect metadata	`Assembly`, `Module`
`System.Security`	Types that enable security capabilities (see Chapter 24 for details)	`Cryptography`, `Permissions`, `Policy`

and one way to handle interfaces with remote components:

- *Web services*

Windows Forms

Windows Forms is a more advanced and integrated way to do standard Win32 screens. All languages that work on the .NET Framework, including new versions of Visual Studio languages, use the Windows Forms engine, which duplicates the functionality of the VB forms engine. It provides a rich, unified set of controls and drawing functions for all languages, as well as a standard API for underlying Windows services for graphics and drawing. It effectively replaces the Windows graphical API, wrapping it in such a way that the developer normally has no need to go directly to the Windows API for any graphical or screen functions.

In Chapter 12, we will look at Windows Forms in more detail and note significant changes in Windows Forms versus older VB forms. Chapter 13 continues discussing Windows Forms technologies by describing in detail the various methods for creating a Windows Forms control.

Client Applications versus Browser-Based Applications

Before .NET, many internal corporate applications were made browser-based simply because of the cost of installing and maintaining a client application on hundreds or thousands of workstations. Windows Forms and the .NET Framework change the economics of these decisions. A Windows Forms

application is much easier to install and update than an equivalent VB6 desktop application. With a simple XCOPY deployment and no registration issues, installation and updating become much easier.

That means "smart client" applications with a rich user interface are more practical under .NET, even for a large number of users. It may not be necessary to resort to browser-based applications just to save installation and deployment costs. .NET even has the capability to deploy these "smart client" applications over the Internet from a Web server, with automatic updating of changed modules on the client.

This means you should not dismiss Windows Forms applications as merely replacements for earlier VB6 desktop applications. Instead, you should examine applications in .NET and explicitly decide what kind of interface makes sense in a given case. In some cases, applications that you might have assumed should be browser-based simply because of a large number of users and wide geographic deployment instead can be smart-client-based, which can improve usability, security, and productivity.

Web Forms

The part of .NET that handles communications with the Internet is called ASP.NET. It includes a forms engine called Web Forms, which can be used to create browser-based user interfaces.

Divorcing layout from logic, Web Forms consist of two parts:

- A *template*, which contains HTML-based layout information for all user interface elements
- A *component*, which contains all logic to be hooked to the user interface

It is as if a standard Visual Basic form were split into two parts, one containing information on controls and their properties and layout, and the other containing the code. Just as in Visual Basic, the code operates "behind" the controls, with events in the controls activating event routines in the code.

As with Windows Forms, Web Forms will be available to all languages. The component handling logic for a form can be in any language that supports .NET. This brings complete, flexible Web interface capability to a wide variety of languages. Chapters 14 and 15 go into detail on Web Forms and the controls that are used on them.

Console Applications

Although Microsoft doesn't emphasize the ability to write character-based applications, the .NET Framework does include an interface for such console applications. Batch processes, for example, can now have components integrated into them that are written to a console interface.

As with Windows Forms and Web Forms, this console interface is available for applications written in any .NET language. Writing character-based applications in previous versions of Visual Basic, for example, has always been a struggle because it was completely oriented around a GUI. VB.NET can be used for true console applications.

Web Services

Application development is moving into the next stage of decentralization. The oldest idea of an application is a piece of software that accesses basic operating system services, such as the file system and

graphics system. Then we moved to applications that used lots of base functionality from other system-level applications, such as a database — this type of application added value by applying generic functionality to specific problems. The developer's job was to focus on adding business value, not on building the foundation.

Web services represent the next step in this direction. In Web services, software functionality becomes exposed as a service that doesn't care what the consumer of the service is (unless there are security considerations). Web services allow developers to build applications by combining local and remote resources for an overall integrated and distributed solution.

In .NET, Web services are implemented as part of ASP.NET (see Figure 1-1), which handles all Web interfaces. It allows programs to talk to each other directly over the Web, using the SOAP standard. This has the capacity to dramatically change the architecture of Web applications, allowing services running all over the Web to be integrated into a local application.

Chapter 22 contains a detailed discussion of Web services.

XML as the .NET "Meta-Language"

Much of the underlying integration of .NET is accomplished with XML. For example, Web services depend completely on XML for interfacing with remote objects. Looking at metadata usually means looking at an XML version of it.

ADO.NET, the successor to ADO, is heavily dependent on XML for remote representation of data. Essentially, when ADO.NET creates what it calls a *dataset* (a more complex successor to a recordset), the data is converted to XML for manipulation by ADO.NET. Then, the changes to that XML are posted back to the datastore by ADO.NET when remote manipulation is finished.

> **Chapter 10 discusses XML in .NET in more detail, and, as previously mentioned, Chapter 8 contains a discussion of ADO.NET.**

With XML as an "entry point" into so many areas of .NET, integration opportunities are multiplied. Using XML to expose interfaces to .NET functions allows developers to tie components and functions together in new, unexpected ways. XML can be the glue that ties pieces together in ways that were never anticipated, both to Microsoft and non-Microsoft platforms.

The Role of COM

When the .NET Framework was introduced, some uninformed journalists interpreted it as the death of COM. That is completely incorrect. COM is not going anywhere for a while. In fact, Windows will not boot without COM.

.NET integrates very well with COM-based software. Any COM component can be treated as a .NET component by native .NET components. The .NET Framework wraps COM components and exposes an

interface that .NET components can work with. This is absolutely essential to the quick acceptance of .NET, because it makes .NET interoperable with a tremendous amount of older COM-based software.

Going in the other direction, the .NET Framework can expose .NET components with a COM interface. This allows older COM components to use .NET-based components as if they were developed using COM.

Chapter 17 discusses COM interoperability in more detail.

No Internal Use of COM

It is important, however, to understand that native .NET components *do not* interface using COM. The CLR implements a new way for components to interface, one that is not COM-based. Use of COM is only necessary when interfacing with COM components produced by non-.NET tools.

Over a long span of time, the fact that .NET does not use COM internally may lead to the decline of COM, but for any immediate purposes, COM is definitely important.

Some Things Never Change . . .

Earlier in the chapter, we discussed the limitations of the pre-.NET programming models. However, those models have many aspects that still apply to .NET development. Tiered layers in software architecture, for example, were specifically developed to deal with the challenges in design and development of complex applications, and are still appropriate. Many persistent design issues, such as the need to encapsulate business rules, or to provide for multiple user-interface access points to a system, do not go away with .NET.

Applications developed in the .NET Framework will still, in many cases, use a tiered architecture. However, the tiers will be a lot easier to produce in .NET. The presentation tier will benefit from the new interface technologies, especially Web Forms for Internet development. The middle tier will require far less COM-related headaches to develop and implement. And richer, more distributed middle tier designs will be possible by using Web services.

The architectural skills that experienced developers have learned in earlier models are definitely still important and valuable in the .NET world.

.NET Drives Changes in Visual Basic

We previously covered the limitations of Visual Basic in earlier versions. To recap, they are:

- No capability for multithreading
- Lack of implementation inheritance and other object features
- Poor error handling ability
- Poor integration with other languages such as C++
- No effective user interface for Internet-based applications

Since VB.NET is built on top of the .NET Framework, all of these shortcomings have been eliminated. In fact, Visual Basic gets the most extensive changes of any existing language in the VS.NET suite. These changes pull Visual Basic in line with other languages in terms of datatypes, calling conventions, error handling, and, most importantly, object orientation. Chapters 5, 6, and 7 go into detail about object-oriented concepts in VB.NET, and Chapter 9 discusses error handling.

"How Does .NET Affect Me?"

One of the reasons you are probably reading this book is because you want to know how VB.NET will affect you as an existing Visual Basic developer. Here are some of the most important implications.

A Spectrum of Programming Models

In previous Microsoft-based development tools, there were a couple of quantum leaps required to move from simple to complex. A developer could start simply with ASP pages and VBScript, but when those became cumbersome, it was a big leap to learn component-based, three-tier development in Visual Basic. And it was another quantum leap to become proficient in C++, ATL, and related technologies for system-level work.

A key benefit of VB.NET and the .NET Framework is that there exists a more gradual transition in programming models from simple to full power. ASP.NET pages are far more structured than ASP pages, and code used in them is often identical to equivalent code used in a Windows Forms application. Internet development can now be done using real Visual Basic code instead of VBScript.

Visual Basic itself becomes a tool with wider applicability, as it becomes easy to do a Web interface with Web Forms, and it also becomes possible to do advanced object-oriented designs. Even system-level capabilities, such as Windows services can be done with VB.NET (see Chapter 21). Old reasons for using another language, such as lack of performance or flexibility, are mostly gone. Visual Basic will do almost anything that other .NET languages can do.

This increases the range of applicability of Visual Basic. It can be used all the way from "scripts" (which are actually compiled on the fly) written with a text editor, up through sophisticated component and Web programming in one of the most advanced development environments available.

Reducing Barriers to Internet Development

With older tools, programming for the Internet requires a completely different programming model than programming systems that will be run locally. The differences are most apparent in user-interface construction, but that's not the only area of difference. Objects constructed for access by ASP pages, for example, must support `Variant` parameters, but objects constructed for access by Visual Basic forms can have parameters of any datatype. Accessing databases over the Internet requires using technologies like RDS instead of the ADO connections that local programming typically uses.

The .NET Framework erases many of these differences. Programming for the Internet and programming for local systems are much more alike in .NET than with today's systems. Differences remain — Web Forms still have significant differences from Windows Forms, for example, but many other differences, such as the way data is handled, are much more unified under .NET.

A big result of this similarity of programming models is to make Internet programming more practical and accessible. With functionality for the Internet designed in from the start, developers don't have to know as much or do as much to produce Internet systems with the .NET Framework.

Libraries of Prewritten Functionality

The evolution of Windows development languages, including Visual Basic, has been in the direction of providing more and more built-in functionality so that developers can ignore the foundations and concentrate on solving business problems. The .NET Framework continues this trend.

One particularly important implication is that the .NET Framework extends the trend of developers spending less time writing code and more time discovering how to do something with prewritten functionality. Mainframe COBOL programmers could learn everything they ever needed to know about COBOL in a year or two, and very seldom need to consult reference materials after that. In contrast, today's Visual Basic developers already spend a significant portion of their time digging through reference material to figure out how to do something that they may never do again. The sheer expanse of available functionality, plus the rapidly changing pace, makes it imperative for an effective developer to be a researcher also. .NET accelerates this trend, and will probably increase the ratio of research time to coding time for a typical developer.

Easier Deployment

A major design goal in Microsoft .NET is to simplify installation and configuration of software. With "DLL Hell" mostly gone, and with installation of compiled modules a matter of a simple file copy, developers should be able to spend less time worrying about deployment of their applications, and more time concentrating on the functionality of their systems. The budget for the deployment technology needed by a typical application will be significantly smaller.

The Future of .NET

At the Professional Developer's Conference (PDC) in Los Angeles in October of 2003, Microsoft gave the first public look at their next generation operating system, code-named Longhorn. It was clear from even this early glimpse that .NET is at the heart of Microsoft's operating system strategy going forward.

However, the naming of what we now know as .NET is going to change. While Web services and related technologies may still carry the .NET label going forward, the .NET Framework is called WinFX in Longhorn. This may cause some confusion in names going forward, but be assured that what you learn today about the .NET Framework and VB.NET will be important for years to come in the world of Microsoft applications.

Summary

VB.NET is not like other versions of Visual Basic. It is built with completely different assumptions, on a new platform that is central to Microsoft's entire product strategy. In this chapter, we have discussed the reasons Microsoft has created this platform, and how challenges in earlier, pre-Internet technologies have been met by .NET.

This chapter has also discussed in particular how this will affect VB developers. .NET presents many new challenges for developers, but simultaneously provides them with greatly enhanced functionality. In particular, Visual Basic developers now have the ability to develop object-oriented and Web-based applications far more easily and cheaply.

In the next chapter, we move on to take a closer look at the VS.NET IDE, and discuss the basics of doing applications in VB.NET.