What is Microsoft .NET?

Microsoft began its Internet development efforts in 1995. Prior to this, Microsoft's focus had, for several years, been on moving desktop and server operating systems to 32-bit, GUI-based technologies, but once Microsoft realized the importance of the Internet, it made a dramatic shift. The company became focused on integrating its Windows platform with the Internet and it succeeded in making Windows a serious platform for the development of Internet applications.

However, it had been necessary for Microsoft to make some compromises in order to quickly produce Internet-based tools and technologies. The most glaring example was **Active Server Pages (ASP)**. While ASP was simple in concept and easily accessible to new developers, it did not encourage structured or object-oriented development. Creating user interfaces with interpreted script and limited visual elements was a real step back from the form-based user interfaces of **Visual Basic (VB)**. Many applications were written with a vast amount of interpreted script, which lead to problems of debugging and maintenance.

Visual Basic (and other languages) has continued to be used in Internet applications on Microsoft platforms, but mostly to create components that were accessed in ASP. Before Microsoft .NET, Microsoft tools were lacking in their integration and ease-of-use for web development. The few attempts that were made to place a web interface on traditional languages, such as WebClasses in VB, were compromises that never gained widespread acceptance. The result was that developing a large Internet application required the use of a large number of loosely integrated tools and technologies.

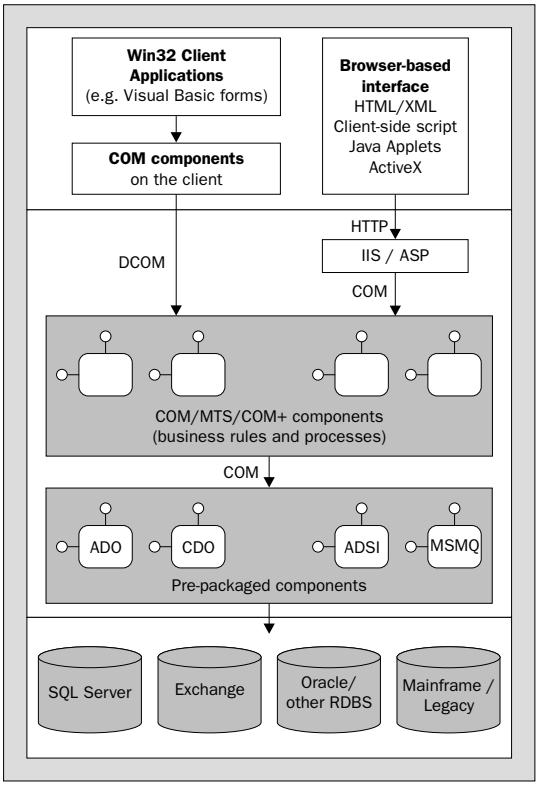
Microsoft .NET is the first software development platform to be designed from the ground up with the Internet in mind – although .NET is not exclusively for Internet development; rather it provides a consistent programming model that can be used for many types of applications. However, when an application needs Internet capabilities, access to those capabilities is almost transparent, unlike tools currently used for Internet-enabled applications.

To understand what the importance of .NET is, it's helpful to understand how current tools such as COM limit us in a development model based on the Internet. In this chapter, we'll look at what's wrong with COM and the DNA architectural model, and then examine how .NET corrects the drawbacks in these technologies.

Although we'll discuss the drawbacks of current tools in the context of the Microsoft platform, almost all apply in some form to all the platforms that are currently available for Internet development. Moreover, many of these other platforms have unique drawbacks of their own.

The DNA Programming Model

In the late 1990s, Microsoft attempted to bring some order to Internet development with the concept of **Windows DNA applications**. DNA consists of a standard three-tier development based on COM, with ASP (as well as Win32 clients) in the presentation layer, business objects in a middle layer, and a relational data store and engine in the bottom layer. The following diagram shows a generic Windows DNA application:



Presentation Tier

In Windows DNA, there are two types of user interfaces – **Win32 clients** and **browser-based clients**.

Win32 clients are most often produced with a visual development tool such as Visual Basic. They are simple to create, and offer a rich user interface. The drawback of such software is that it is difficult to deploy and maintain – it must be installed on every client and every installation must be altered whenever an upgrade to the software is made. In addition to these logistical difficulties, DLL conflicts frequently occur on the client because of variations in the version of the operating system and other software installed on the client. This is known as **DLL Hell**, which we'll discuss later on.

Browser-based clients are far easier to deploy than Win32 clients as the client only needs a compatible browser and a network (Internet or intranet) connection. However browser-based clients are often more difficult to create than Win32 clients, and offer a more limited user interface that has fewer controls and permits only limited control over the layout of the screen and the handling of screen events.

There are some in-between options. If clients are restricted to certain browsers, **Dynamic HTML (DHTML)** can be used to add functionality to the interface. If clients are further restricted to Internet Explorer (IE), **ActiveX controls** can be used to create an interface that is close to that available in a Win32 client. However, ActiveX controls have deployment issues of their own. VB can be used to create ActiveX controls, but then deploying the controls requires lots of supporting VB DLLs to be present on the client. Consequently, ActiveX controls are typically written in C++ to make the installation as lightweight as possible, but this adds to development time and requires a higher level of development expertise.

One important factor that is often overlooked in the DNA model is that there may be a need to implement both Win32-based and Internet-based user interfaces. Alternatively, there may be a need to have different types of user interface, perhaps one for novice or occasional users, and one for advanced users. This is practical only if the design of the system keeps the user interface layer as thin as possible. Normally, it should only contain logic that manages the user interface and performs basic validation of user data. (Such validation of data in the client layer is important as it minimizes round trips to the server.)

Middle Tier

The middle tier in a DNA application should encapsulate as much of the business logic processing as possible. Apart from those rules that are needed to validate data on the client, all the **business rules** should be in the middle layer.

The middle tier is often broken down into sub-tiers. One tier may handle the interface to the client, another tier the business rules, and another tier the interface to the data repositories.

Visual Basic is the language most commonly used to write middle-tier components. This is a more sophisticated type of development than for forms-based Visual Basic programs; it requires a greater level of expertise in COM and object-oriented programming. Understanding COM is important in constructing a middle tier because the components in this layer must work together, which means that all the components must be versioned properly so that they can understand each other's interfaces. It's also important to create components that scale well, which often means developing components that are implemented using Microsoft Transaction Server (MTS) or COM+ Services. Such components typically use stateless designs, which can look very different from the stateful designs commonly used in client-based components.

Components in the middle tier may use a variety of protocols and components to communicate data to the data tier. The diagram shows examples such as HTTP, ADO (ActiveX Data Objects), ADSI (Active Directory Service Interfaces), and CDO (Collaboration Data Objects), but that list is by no means exhaustive.

Data Tier

Most business applications must store information for long-term use. The nature of the storage mechanism varies with the installation but a relational database management system (RDBMS) is often required, with the most common options being Microsoft SQL Server and Oracle. However, if the information is based around documents and messages, a messaging data store such as Exchange may be required, and many installations will depend on legacy mainframe systems.

Besides holding the data, the data tier may also contain logic that processes, retrieves, and validates data. Stored procedures, written in some variation of SQL (Structured Query Language), can be used with RDBMS databases to do this.

Issues with the DNA Model

The concept behind DNA is sound, but actually getting it to work well is overly complicated. A DNA application will often require:

- Visual Basic code in forms, as well as in components on both the client and the server
- ASP scripting code as well as client-side scripting
- HTML, DHTML, CSS (Cascading Style Sheets)
- XML, XSL
- C++ in ActiveX components
- Stored procedures (Transact-SQL in SQL Server or PL-SQL in Oracle)

With so many options, it's all too easy to make inappropriate design decisions, such as putting logic on the client that belongs on the server, or creating VBScript for formatting when CSS would work better. Designing and constructing a complex DNA-based application requires a high level of expertise in a number of different technologies.

The Limitations of COM

While COM is a viable platform for enterprise-level Internet applications, it does have some serious limitations. Let's cover some of the major ones.

DLL Hell

COM-based applications are subject to major deployment and configuration issues. Small changes in COM interfaces can render entire applications inoperable. This problem, in which small problems cascade through an entire component-based tier is often referred to as **DLL Hell** – experienced COM developers will attest to the appropriateness of the term. Getting a large set of DLLs to a compatible state of versioning requires skill and a well-controlled deployment process.

While DLL Hell is most common in the middle tier, there are also deployment issues in the client tier that are caused by COM. Any forms-based interface will depend on COM components in order to function. Some of these components are from the tool used to create the interface (such as Visual Basic); others may be custom-written DLLs. All will need to be installed on the client.

The class IDs (which are GUID-based identifiers) of all the COM-based components must be placed in the local client's Windows Registry. Complex installation programs typically do this. Getting all the necessary components registered and properly versioned on the client is a variant of DLL Hell, and makes deploying client applications to large numbers of desktop machines an expensive process. This has driven many application designers to use browser-based interfaces whenever possible in order to avoid such deployment costs, even though the browser user interface is not as flexible.

Lack of Interoperability with Other Platforms

COM works well on pure Microsoft platforms but it doesn't provide the ability to activate or interoperate with components on other platforms such as UNIX. For enterprise-level applications, this is a significant shortcoming as large organizations often have a variety of operating platforms, and require interoperability between them.

Lack of Built-In Inheritance

One of the most important ways in which functionality can be reused is for a software component to be inherited by another component, and then extended with new functionality. (Chapter 6 covers this issue in detail.) Inheritance is crucial in developing complex application frameworks, but COM does *not* support inheritance natively.

Inheritance has been possible on Microsoft platforms at the source language level, using languages such as C++ and Delphi. However, since inheritance is not built into the basic structure of COM, many languages (such as VB6) don't support it, and there was no capability on Microsoft platforms before .NET to allow languages to inherit from components written in another language.

Limitations of VB6 for DNA Application Development

Visual Basic 6 is easily the most popular language for developing applications with the DNA model. It is used in two major roles: forms-based VB clients and COM components (either on 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.

However, although it's popular, VB6 isn't without its limitations, which include:

- **No capability for multithreading** – which implies, for example, that VB6 can't be used to write an NT-type service. There are also situations in which the apartment threading used by components created in VB6 limits performance.
- **A lack of implementation inheritance and other object-oriented features** – this makes VB6 unsuitable for the development of object-based frameworks.

- **Poor error-handling ability** – VB6's archaic error handling becomes especially annoying in a multi-tier environment. It's difficult in VB6 to track and pass errors through a stack of component interfaces.
- **Poor integration with other languages such as C++** – VB6's implementation of COM, although easy to use, causes problems with such integration. Class parameters (object interfaces) in VB6 are "variant compliant", forcing C++ developers who want to integrate with VB to convert parameters to less appropriate types. These varying data structures and interface conventions must be resolved before components in VB can be integrated into a multiple language project. Besides requiring extra code, these conversions may also result in a performance hit.
- **No effective user interface for Internet-based applications** – perhaps the biggest drawback to using VB6 became apparent when developing for the Internet. While VB6 forms for a Win32 client were state-of-the-art, for applications with a browser interface VB6 was mostly relegated to use in components.

Microsoft tried to address this last problem in VB6 with **WebClasses** and **DHTML Pages** but neither caught on:

- WebClasses offered an obscure programming model, and limited control over visual layout.
- DHTML Pages in VB6 had to send a (usually large) DLL to the client, and so needed a high-bandwidth connection to be practical. This limited their use mostly to intranet applications. DHTML Pages were also restricted to Internet Explorer.

Limitations of DNA Internet Development

There are a few additional areas in which previous Microsoft tools and technologies fell short of the ideal for Internet application development.

Different Programming Models

With DNA-based software development, creating software that is accessed by a user locally is done very differently from development for the Internet. The starkest example of this is the use of VB forms for client-server user interfaces versus the use of ASP for Internet user interfaces. Even though both situations involve designing and implementing GUI-based user interfaces, the tools and programming techniques used are quite different.

Having very different programming models for these similar types of development causes several problems:

- Developers have to learn multiple programming models.
- Code developed for one type of interface typically cannot be used for the other type of interface.
- It is uncommon to have both local and web-based user interfaces for an application, even though this could result in a better user experience for local users. Usually, it's simply too expensive to implement two interface tiers.

No Automatic State Management

Developers using VB6 forms and local components are accustomed to making the user interface more convenient by creating forms that remember things for the user – the interface maintains **state**. If a piece of information is placed in a text box, it stays there until it is explicitly changed or removed by the developer or user.

ASP, however, has no such capability. Every time a page is rendered, we must make sure that all the visual controls have their information loaded. It is the programmer's responsibility to manage the state in the user interface, and to transfer state information between pages.

This means that developers have to write a lot of code for Internet user interfaces that is not relevant to the business problem the application is designed to solve. In addition, if an Internet application is going to run on a group of web servers (often called a web farm), then considerable additional work is necessary to design a state management system that is independent of a particular server.

Weak User Interfaces over the Web

It is possible to produce sophisticated user interfaces for the Web by using DHTML and writing a lot of JavaScript. However, most web-based applications actually offer fairly primitive user interfaces because it takes too much time and expertise to write a sophisticated one. (Including a lot of nice graphics doesn't make a user interface sophisticated – it just makes it pretty.)

Developers who cut their teeth on producing state-of-the-art interactive user interfaces in VB during the mid-1990s were never satisfied with the compromises necessary for web interfaces. Better user interfaces on the Web would be an enormous boost for user productivity.

The Need to Abstract the Operating System

Today's applications need to use the Windows API for a variety of purposes. VB6 developers use the API to monitor Windows messages, manipulate controls, read, and write INI files, and a variety of other tasks.

This is some of the fussiest programming VB6 developers ever have to do. The Windows API is hard to program to for a variety of reasons. It isn't object-based, which means we must learn complex calls to functions with long lists of arguments. The naming scheme for the functions is inconsistent and since the whole API is written in C/C++, getting calling conventions right on data types such as strings is very messy.

There's a larger issue here as well. As hardware platforms proliferate, it's no longer enough for software just to run on desktop clients and servers. There are handheld and wireless devices of various kinds, kiosks, and other types of systems, many of which run on different processors and don't use standard Windows as an operating system. Any software written with calls to the Windows API won't be portable to any of these systems without major changes. The only way that software produced with Microsoft tools can become more portable is to abstract away the Windows API, so that application software does not write directly to it. This actually creates the possibility of an equivalent layer of abstraction on other platforms that could allow Microsoft-based software to run on them.

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

The Solution – Microsoft .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 integration of software over the Internet, and a set of pre-built web-based services called **.NET My Services** (formerly codenamed Hailstorm).

Microsoft also released several products early in 2001, which were described as being part of the **.NET Enterprise Server** family: SQL Server 2000, Commerce Server 2000, BizTalk Server, Exchange 2000, Host Integration Server (the successor to SNA Server), and Internet Security and Administration (ISA) Server (the successor to Proxy Server).

Some of the marketing literature for these products emphasizes that they are part of Microsoft's .NET strategy. However, it is important to understand the difference between these products and the .NET Framework. The .NET Enterprise Servers are *not* based on the .NET Framework. Most of them are successors to previous server-based products, and they use the same COM/COM+ technologies as their predecessors.

These .NET Enterprise Servers still have a major role to play in future software development projects. When actual .NET Framework projects are developed, most will depend on the technologies in the .NET Enterprise Servers for functions like data storage and messaging.

The General Goals of .NET

Many of the goals Microsoft had in mind when designing .NET reflect the limitations we identified for their earlier tools and technologies.

Creating Highly Distributed Applications

The trend in business applications is towards a more highly distributed model. The next generation of applications may have their elements distributed among various organizations. This contrasts with today's dominant model in which all the elements of an application (except possibly a browser-based client) are located solely within a single organization.

Simplifying Software Development

Developers need to be able to concentrate on the business logic in their applications, and to stop writing logic for things like state management and scalability. Writing software for the Internet should not require expertise in a long list of Internet-specific technologies.

A related goal is to have development for the Internet look very much like development for other platforms. A component accessed over a local network or over the Internet should be manipulated with code very much like that for a component accessed on the local machine. The software platform should be able to take care of the details in transmitting information to and from the component.

Better User Interfaces over the Web

User interface development also needs to be as similar as possible for the Internet compared to local access. While using local, platform-specific interfaces will always offer more flexibility than a browser-based interface, Microsoft .NET aims to make those two types of interfaces as similar to develop as possible.

By making web-based user interfaces richer and more flexible than they are now, bringing them as close as possible to the richness of local, forms-based interfaces.

Simplifying Deployment

The problems of DLL Hell, and the need for large installs of forms-based applications, are just two examples of current deployment issues. Microsoft .NET aims to make deployment as simple as it was for DOS – just copy a compiled module over to a system and run it. No registration, no GUIDs, no special installation procedure.

Support for a Variety of Languages

While the idea of one grand, unifying language sounds good in theory, in the real world, different types of developers need different tools. Microsoft .NET is designed to support a multitude of languages, from Microsoft and third-parties. This will allow the development community to evolve languages that best fit various development needs.

An Extendable Platform for the Future

A new platform needs the capability to adapt to changing conditions through extensions and variations. .NET is designed with greater extendibility and flexibility than any previous software development platform.

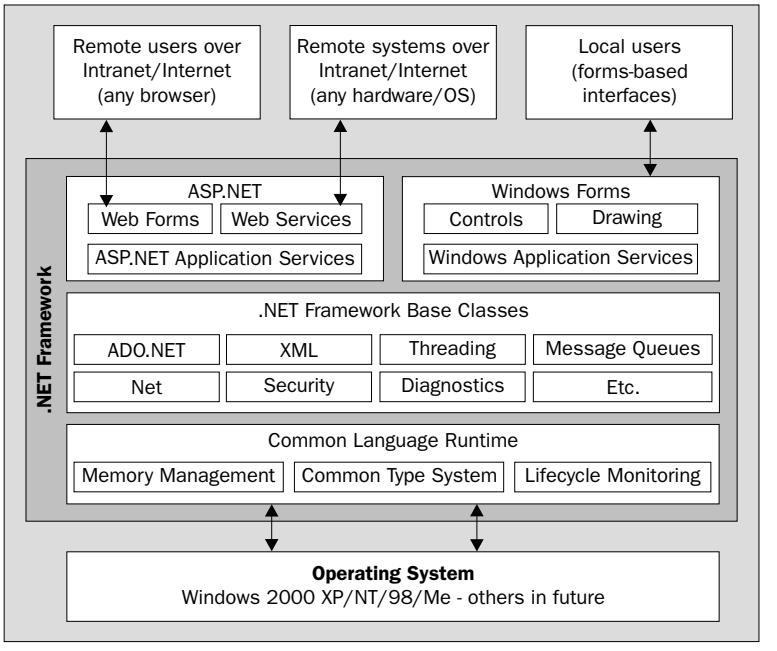
Future Portability of Compiled Applications

Operating systems will make major changes and perhaps entirely new ones will be introduced in the future. Investments in software development need to be carried forward to those platforms. The goal of .NET is to allow applications to move from current platforms to future platforms, such as 64-bit operating systems with a simple copy, and no recompilation.

The Structure of Microsoft .NET

These are ambitious goals. To understand how they are accomplished, we need to understand the general structure of Microsoft .NET.

One way to look at .NET is to see how it fits into the rest of the computing world. Here is a diagram of the major layers of .NET, showing how they sit on top of an operating system, and provide various ways to interface to the outside world. Note how the entire architecture has been created to make it as easy to develop Internet applications, as it is to develop for the desktop:



The first point of this diagram is that .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.

The .NET Framework wraps the operating system, insulating software developed with .NET from most operating system specifics such as file handling and memory allocation.

The .NET Framework itself starts with the execution engine, memory management, and component loading, 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.

The Common Language Runtime

The **Common Language Runtime (CLR)** is at the heart of the .NET Framework. The core of the CLR is an execution engine that loads, executes, and manages code that has been compiled into an intermediate byte-code format called **Microsoft Intermediate Language** (**MSIL** and often referred to as just **IL**). This code is not interpreted – it is compiled to native binary code before execution by just-in-time compilers built into the CLR.

That means there are two levels of compilers in .NET. The language compiler takes the source code and creates MSIL. This MSIL byte code is portable to any .NET platform. At execution time, this code is then compiled by the just-in-time compilers into the native binary code of the machine the code is executed on.

> **Chapter 3 will cover the capabilities of the CLR in detail. Do not skip that chapter. Understanding the CLR is a vital step in understanding .NET as a whole.**

The .NET Framework Class Library

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 Framework class library**, sometimes referred to as the **.NET base classes**. .NET includes functionality that is, in many cases, a duplication of existing class libraries. There are several reasons for this:

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

The .NET Framework class library contains thousands of classes and interfaces. Here is just some of the functionality it contains:

- Database access and manipulation
- Creation and management of threads
- Interfaces from .NET to the outside world – Windows Forms, Web Forms, Web Services, and console applications
- Definition, management, and enforcement of application security
- Application configuration
- Working with Directory Services, Event Logs, Processes, Message Queues and Timers
- Creating and working with Windows Services
- Encrypting and decrypting files
- Parsing and manipulating data in XML files
- Sending and receiving data with a variety of network protocols
- Accessing metadata information stored in assemblies, which are the execution units of .NET (think of them as DLLs and EXEs)

Much of the functionality that you might think of as being part of a language has been moved to the .NET Framework classes. For example, the `System.Math.Sqrt` method in the Framework Classes replaces the Visual Basic keyword `Sqr` for extracting a square root.

All .NET languages have these Framework classes available. That means that C#, for example, can use the same function mentioned above for getting a square root. This makes accessing base functionality highly consistent across languages. All calls to `Sqrt` look essentially the same (apart from syntactical differences between 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)
```

```
// Same example in C#
Double dblNumber = 200;
Double dblSquareRoot;
dblSquareRoot = System.Math.Sqrt(dblNumber);
```

User and Program Interfaces

In a sense, the top layer of the .NET Framework is an extension of the .NET Framework Base Classes layer immediately underneath it. It comprises highly innovative user and program interfaces that allow .NET to work with the outside world. These interfacing technologies are all highly innovative:

- **Windows Forms** is a language-independent forms engine that brings the drag-and-drop design features of Visual Basic to all .NET-enabled languages, and also enables developers to develop forms-based interfaces with little or no access to the Win32 API. They are discussed in Chapters 12 and 13.
- **Web Forms** brings drag-and-drop design and an event-driven architecture to Web-based interfaces, implementing a programming model that is much like standard VB6 forms-based development. User interfaces created with Web Forms also have built-in browser independence and state management. They are discussed in Chapters 14 and 15.
- **Web Services** allow remote components, possibly running on completely different operating systems, to be invoked and used. This capability for communications and interoperability with remote components over the Internet serves as the mechanism by which highly-distributed applications can be built, going far beyond what is feasible with existing technologies like DCOM. Web Services are discussed in Chapter 22.

XML as the .NET Meta-Language

Much of the underlying integration of .NET is accomplished with XML:

- Web Services depend completely on XML for interfacing with remote objects.
- The information about execution modules, called **assemblies**, can be exported as XML.
- 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 data store by ADO.NET when remote manipulation is finished. (Chapter 11 discusses ADO.NET in more detail.)

XML, and its relationship to VB.NET, is discussed further in Chapter 10.

How Microsoft .NET Attains Its Goals

Now that we've had a short introduction to the structure of .NET, we can better understand how it meets the goals Microsoft set out for it.

Simplified Software Development

.NET simplifies the development of business application software through:

- **Pre-Written Functionality** – The .NET Framework Base Classes make it unnecessary to write system-level code. These classes furnish a wide array of functionality, and can be extended via inheritance if additional functionality is needed. It is no longer necessary to start over from scratch if a particular pre-built component does not do exactly what we need.
- **Transparent Integration of Internet Technologies** – In .NET, the protocols and mechanisms for accessing Internet resources are built into the platform in such a way that we do not need to handle the details. For example, Web Services are created by simply marking a function with a `<WebMethod>` attribute (more on this in Chapter 22). Creating simple web interfaces with Web Forms doesn't require an extensive knowledge of HTML, or how to handle information from an HTTP post operation. The controls used in Web Forms automatically produce JavaScript (if the browser in use can run it) to handle data validation on the client. If the browser does not support JavaScript, the data validation is run transparently on the server.

  This integration of web technologies reduces the expertise barriers to web development. While it's still helpful to know a lot about HTML, DHTML, and so on, traditional VB developers will find that developing web software with VB.NET is much easier than with ASP.
- **Unified Programming Models for All Types of Development** – Web Services are just regular functions with a `<WebMethod>` attribute attached, which means they are created and consumed much like local components. Once the Web Service's location is referenced, Web Service classes are instantiated the same way as local classes, and their interface looks like a typical object interface. Web Services even have IntelliSense in the development environment, just like local components.

  Likewise, developing user interfaces in Windows Forms is very similar to developing them in Web Forms. Both contain commonly used controls, such as labels and text boxes, which have similar properties and methods. Of course, not everything can be the same, because the disconnected model for Web Forms means it is impractical to have as many events as in Windows Forms. (For example, the mouse-moving events are mostly missing from Web Forms.) However, there's enough commonality between the two to make it easy to move between the two types of development, and for traditional VB developers to start using Web Forms.

Highly Distributed Systems

The vision of Microsoft .NET is of globally distributed systems that use XML as a universal glue in order to allow functions running on different computers across organizations to form a single application. In this vision, systems from servers to wireless palmtops (and everything in between) will share the same general platform, with versions of .NET available for all of them, and with each able to integrate transparently with the others.

Web Services are the mechanism for reaching this vision. In Web Services, software functionality is exposed as a service that doesn't care what the consumer of the service is (apart from security considerations). Web Services allow developers to build applications by combining local and remote resources to create an integrated, distributed solution.

Web Services have enormous potential. For example, a commercial software company could produce a Web Service that calculates sales tax for every jurisdiction in the nation. A subscription to that Web Service could then be sold to any company that needs to calculate sales tax. The customer has no need to deploy the sales tax calculator because is it just called over the Web. The company producing the sales tax calculator can dynamically update it to include new rates and rules for various jurisdictions and their customers that use the Web Service don't have to do anything to get these updates.

Better User Interfaces over the Web

Web Forms are a giant step towards much richer web-based user interfaces. Their built-in intelligence allows rich, browser-independent screens to be developed quickly, and to be easily integrated with compiled code.

Simplified Deployment

Executable modules in .NET are self-describing. Once the CLR knows where a module resides, it can find out everything else it needs to know to run the module, such as the module's object interface and security requirements, from the module itself. That means that a module can be copied to a new system and immediately executed.

The CLR is capable of loading multiple versions of a single DLL that can be executed side-by-side. Each executable module identifies the particular DLL it needs, and the CLR runs it against the correct one. This means that versioning difficulties are dramatically reduced with .NET.

This is a huge leap from the complex deployment of before. While advanced applications still need an installation program to accomplish tasks such as setting up database connections and other configuration information, such programs are much simpler than before. Simple applications do not need an installation program at all.

But perhaps the most radical improvement for deployment is the ability of .NET to deploy over the Internet. The components and forms for an application can be placed on a web server, and a simple launch program on the client can automatically cause elements of the application to be copied from the web server as needed, and for new versions of the application's components to be automatically updated on the client. The only requirement for the client to use this capability is to have the .NET Framework installed on it, and to have Internet connectivity. This option promises to revitalize the use of smart client interfaces, because the deployment costs associated with COM-based client applications are virtually eliminated.

Support for a Variety of Languages

The CLR executes binary code in MSIL, and that code looks the same regardless of the original source language. All .NET-enabled languages use the same data types and the same interfacing conventions. This makes it possible for all .NET languages to interoperate transparently. One language can call another easily, and languages can even inherit classes written in another language and extend them. No other platform has anywhere near this level of language interoperability.

This makes choosing a language mostly a matter of taste. .NET-enabled languages will typically have the same performance characteristics, the same overall functionality, and will interoperate with other languages the same.

One of the most important aspects of meeting this goal is that Visual Basic becomes a first-class language. It has almost exactly the same capabilities as C#. It has inheritance, structured error handling, and other advanced features. With the large number of developers who already know Visual Basic, VB.NET should set the stage for Visual Basic to continue to be the most popular programming language in the world.

Extendibility of the Platform

The completely object-based approach of .NET is designed to allow base functionality to be extended through inheritance (unlike COM), and the platform's functionality is appropriately partitioned to allow various parts (such as the just-in-time compilers discussed in Chapter 3) to be replaced as new versions are needed.

It is likely that, in the future, new ways of interfacing to the outside world will be added to the current trio of Windows Form, Web Forms, and Web Services. The architecture of .NET makes such additions quite practical.

Future Portability

By abstracting away the underlying platform as much as possible .NET makes possible a future in which software is moved to new hardware and operating system platforms. The core elements of .NET have been submitted to standards bodies, with the intent of standardizing the core of .NET on different systems. The ultimate goal is that code compiled on one implementation of .NET (such as Windows) could be moved to another implementation of .NET on a different operating system and executed without change.

The Role of COM

.NET integrates very well with COM-based software, which is fortunate because COM is not going to disappear for a while. Any COM component can be treated as a .NET component by other .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 COM-based software.

Going in the other direction, the .NET Framework can expose .NET components with a COM interface. This allows COM components to use .NET-based components as if they were developed using COM. (COM interoperability is discussed in more detail in Chapter 17).

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

In the long term, the fact that .NET does not use COM internally may lead to the decline of COM – but that is for the very long term. In the short to medium term, COM is definitely still important.

The Role of DNA

Earlier in the chapter, we discussed the limitations of the current DNA programming model. These limitations are mostly inherent in the technologies used to implement DNA today, not in the overall structure or philosophy. There is nothing fundamentally wrong with the multi-tiered approach to development specified by the DNA model. Many design issues, such as the need to encapsulate business rules, or to provide for multiple user interface access points to a system, apply to .NET.

In many cases, applications developed in the .NET Framework will still use a DNA model to design the appropriate tiers. However, the tiers will be a lot easier to produce in .NET:

- The presentation tier benefits from the new interface technologies, particularly Web Forms for Internet development
- The middle tier requires far less COM-related headaches to develop and implement
- Richer, more distributed middle tier designs are possible by using Web Services

The architectural skills that experienced developers have learned in the DNA world are still valuable in the .NET world.

Additional Benefits

In addition to the advantages conferred by meeting the goals we discussed previously, .NET offers a number of additional benefits. These include:

- Faster development – we have less to do as the system handles more
- More reuse because of inheritance
- Greater scalability – many capabilities to help applications scale are built into .NET
- Easier to build sophisticated development tools – debuggers and profilers can target the Common Language Runtime, and thus become accessible to all .NET-enabled languages
- Fewer bugs – whole classes of bugs should be unknown in .NET; for example, with the CLR handling memory management, memory leaks should be a thing of the past
- Potentially better performance – Microsoft's heavy investment in system level code for memory management, garbage collection, and the like have yielded an architecture that should meet or exceed performance of typical COM-based applications today

Impact on Visual Basic

Since VB.NET is built on top of the .NET framework, the shortcomings in VB6 that we discussed earlier have been eliminated. In fact, VB gets the most extensive changes of any existing language in the Visual Studio suite. These changes pull VB in line with other languages in terms of data types, calling conventions, error handling, and – most importantly – object-orientation. These changes will be covered in Chapters 5, 6, and 7.

Microsoft includes a migration tool in Visual Studio .NET, and it can assist in porting VB6 projects to .NET, but it will not do everything required. There are some areas, including unsupported, obsolete syntax such as `GOSUB`, where the tool merely places a note that indicates something needs to be done. You can find more information about compatibility between VB6 and VB.NET in Appendix A.

Summary

This chapter explained the importance of .NET and just how much it changes the way that applications are developed. Understanding these concepts is essential in order to use VB.NET in the most effective manner.

It is possible to use VB.NET merely to write the same kinds of software as were written in VB6, only faster and more cleanly. However, this would be failing to use much of the value of VB.NET. The real opportunities are in creating entirely new types of applications such as Web Services, and in implementing application frameworks that promote reuse of code. The rest of this book explains the concepts and technologies you'll need to do that.

In the next chapter we'll get started creating VB.NET applications, as well as take a first look at the new development environment provided by Visual Studio .NET.