Overview of the .NET Framework

Rockford Lhotka, Magenic Technologies

Introduction

Microsoft .NET is an exciting new development platform that brings true programmability to the world of the Internet. There are many facets to the .NET technologies, and this session is an overview of the major technologies and the roles they play.

Rockford Lhotka

Rockford Lhotka is the author of "Visual Basic 6 Distributed Objects" and "Visual Basic 6 Business Objects", and is a contributing author for Visual Basic Programmers Journal and DevX. He has presented at major conferences in the US and Europe. Rockford is the Principal Technology Evangelist for Magenic Technologies, one of the nation's premiere Microsoft Certified Solution Providers.

.NET Architecture Overview

The Internet and World Wide Web have transformed our industry in many ways - energizing users around computing and offering many new challenges to developers.

Interestingly enough, many of the challenges we face when developing in the web environment stem from the relative immaturity of the environment itself and the tools we use in development. This is not surprising, since it took many years for the client/server environment to stabilize, for client/server tools to exist - these advancements were required for developers to be fully productive.

The tools and environment for web development have been in a constant state of flux because of the web's meteoric rise in popularity. However, few if any of the tools have provided a development environment similar or comparable to Visual Basic (VB) or its ilk.

Recognizing this, Microsoft has announced Microsoft .NET – a leap forward in web development tools and environment.

Backoffice Servers

Before we get too far, it is important to recognize that this paper will only address *part* of .NET. .NET includes an entire set of backoffice servers that will be released over the next few months, including:

- SQL Server
- BizTalk Server
- Exchange Server
- Commerce Server
- Host Integration Server

While these are exciting releases, each offering new capabilities and productivity – they are largely evolutionary. With the exception of Biztalk Server, they build on previous versions of the products, increasing performance and features.

> *Originally these servers were the core of Windows DNA 2000. While .NET is not just DNA 2000 renamed, it is true that these servers now form a large part of the .NET strategy, with Windows DNA 2000 now a defunct term.*

The .NET Framework

.NET also includes some more revolutionary components – in the form of the .NET Framework. The .NET Framework includes:

- Common Language Runtime
- .NET System Class library
- ASP+ (WebForms)
- ADO+
- C# (pronounced *C sharp*)

In this paper we'll focus on the .NET Framework and its constituent parts. From a developer's perspective this is truly exciting - these are the things that can change the way web development is done!

By way of disclaimer – .NET is huge and this paper is small. This paper is merely a high-level overview of some of the key features of .NET. For more details, download the .NET SDK from Microsoft's web site at http://msdn.microsoft.com.

Abstracting the Operating System

One of the ideals that computer science has been working toward for many years is the concept that a program can be written to run on many different hardware or operating systems.

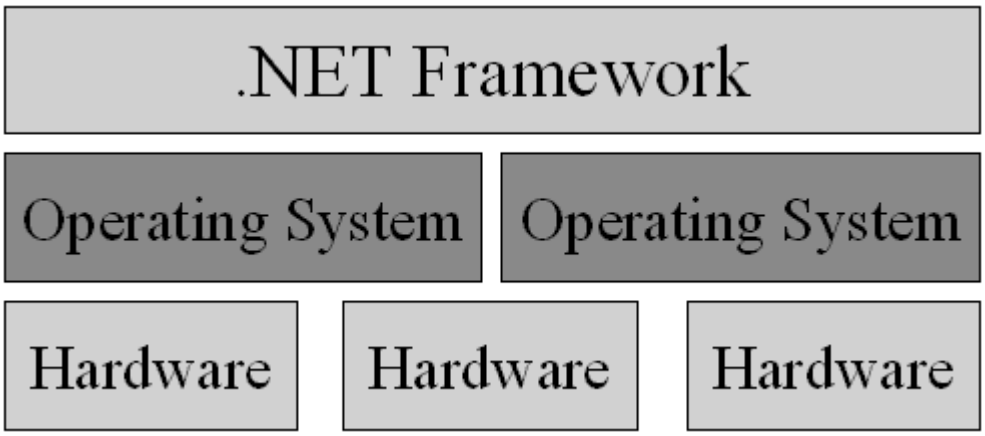
To some degree, we've addressed this issue at the hardware level by abstracting the hardware through the use of an operating system. At its core, any operating system's purpose is to abstract the underlying hardware so a programmer doesn't have to worry about registers, memory, and other hardware level issues.

The problem is, however, more complex at an operating system level. To date, the typical strategy has been for operating system vendors to strive for market share - enabling users to run software on most computers because they tend to use one of a few common operating systems. Unfortunately, even with a single vendor (such as Microsoft) there are variations on the operating system - Win95, Win98, Win98SE, etc. Our programs don't always run the same with these variations, much less across both Win32 and OSs along the lines of Solaris.

> *A common Q and A within computer science goes something like this:*
>
> *Q: "How do you solve a tough computer problem?"*
>
> *A: "Add a layer of abstraction."*

That's certainly what we've done with hardware, by layering operating systems on top to abstract all that complexity.

And in many ways that is exactly what Microsoft is doing with .NET. The Common Language Runtime (CLR), combined with the system class library, form a comprehensive development environment that *abstracts the operating system*.



.NET abstracts operating systems, which are abstracting hardware.

This means that we can write software to target the CLR – and that software can run on any operating system or platform which implements the CLR.

Let's take a look at the CLR along with the other key parts of the .NET Framework.

Common Language Runtime

The CLR lives at the heart of the .NET Framework. The CLR provides an environment in which our programs can run. This includes concepts such as compilation, registration, and even deployment issues.

As the name implies, the CLR is designed to support many programming languages in a common manner. This is exciting because it means that developers don't have to learn a new language syntax to program in the .NET environment. Microsoft will be releasing some core languages with Visual Studio.NET, including:

- VB
- C#
- C++
- Jscript

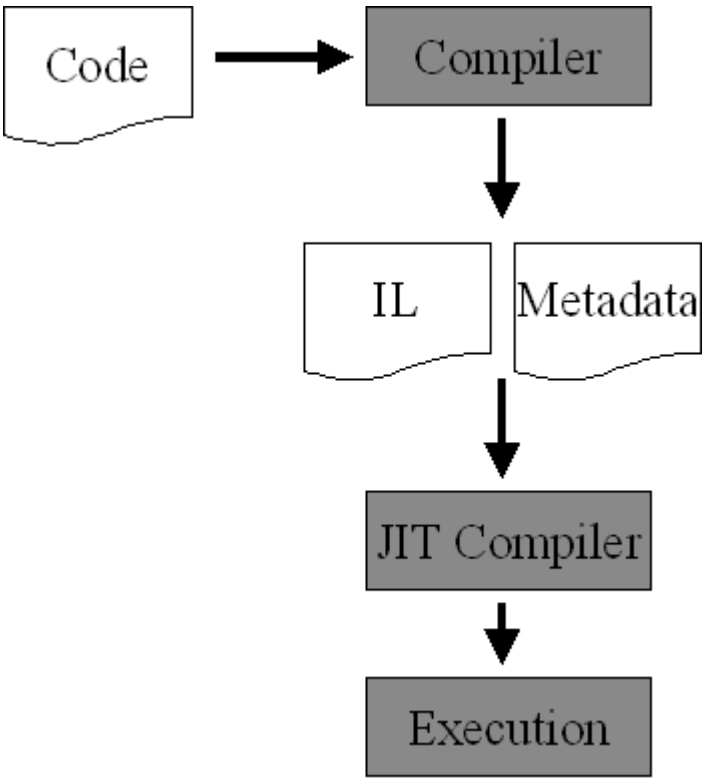
Other vendors will undoubtedly develop compilers for other languages, including COBOL and perhaps even Java.

Compilation

An important part of the CLR is its compilation process. This process provides a great deal of flexibility and is at the root of many of .NET's capabilities overall. All code in .NET is natively compiled, but that compilation typically occurs just in time to run the code rather than way back before deployment as is typical today.

To understand the compilation process we need to realize that .NET introduces not one new language, but two. C# has received a lot of attention from the press and developers, but .NET also introduces MSIL - the Microsoft Intermediate Language. All languages that work with .NET have compilers that generate MSIL rather than machine code.

MSIL code (and some metadata we'll discuss later) can be deployed as needed, and will be compiled into machine code on the user's computer, as is illustrated by the following diagram:



Languages compile to IL, and IL is compiled to machine code.

To summarize, a developer can build an application in VB.NET and compile it. That compilation process generates IL code rather than binary machine code. The application can then be deployed as needed. When a user actually runs the application, the CLR invokes a just-in-time (JIT) compiler on the user's computer to compile the IL into machine code.

Metadata

As illustrated by the diagram, not only does each language compiler generate IL, it also generates metadata. This metadata is key to .NET as it provides an exact description of the components that were compiled to IL.

In the world of COM, components were described by MIDL - the Microsoft Interface Definition Language. MIDL was unable to completely and accurately describe components, which could be a limiting factor in application design, and most definitely caused problems when VB code tried to call a C++ component or visa versa.

The metadata scheme used by .NET is far more robust. The data is kept in XML so it is easily accessible and understandable. More importantly, the metadata for the component completely and accurately describes the component's interfaces. The result is that components can be created in and called by any language - avoiding nearly all associated problems with COM.

In fact, because the IL and metadata are fully accessible to the .NET runtime we gain a number of key cross-language benefits, including:

- Seamless cross-language method calls
- Cross-language inheritance
- Code can be checked to see if it is "unsafe" before being run
- .NET can automatically serialize objects

Additionally, metadata eliminates any reliance on the Windows registry - something looked at more deeply when deployment is subsequently discussed.

Managed vs. Unmanaged Code

A common term that comes up when discussing .NET and the CLR is the concept of **managed code**. This is a pretty straightforward concept.

Any code that runs within the context of the CLR is managed code. Any code that runs within the native environment of the underlying operating system (within the Win32 environment for instance) is unmanaged code.

Managed code is just code that is running, and thus is being managed by, the .NET Common Language Runtime.

Unsafe Code

When discussing .NET, the concept of "safe" vs. "unsafe" code will arise. The term "unsafe" can be confusing, because unsafe code isn't *necessarily* unsafe – it just *could be* unsafe.

Unsafe code is code that works with pointers - meaning that it could directly alter the call stack, or take other steps that might circumvent predictable behavior. That doesn't mean our code actually *does* any of these things, just that it could.

Essentially, unsafe code is code that the CLR can't ensure will always run properly or safely.

Because the CLR compiles the IL code right before it is actually run, it can easily check through the IL as it is being compiled to determine if it is safe. Security settings on a machine can prohibit the running of unsafe code, helping to prevent malicious code from running within the .NET environment.

Deployment

One of the biggest issues Windows developers have been grappling with for years is **DLL hell**. This problem has many faces, including (but certainly not limited to):

- Incompatible DLLs on a client system
- Installing a program can break existing programs
- Inability to run a component that isn't registered
- Requirement to reregister a component when it is updated
- Inflexibility in component interfaces

.NET addresses DLL hell and the general complexity of deploying COM applications by entirely changing the way components interact with each other.

Applications in .NET are composed of one or more **assemblies**. An assembly contains IL and its associated metadata - thus containing both code and a complete descriptor of that code all in one convenient package.

Since an assembly is totally self-describing (via its metadata), there's no need to register anything within the registry. Because of this fact, deployment is no more complicated than copying the assembly to a directory and running it.

If code in an assembly requires code from another assembly, they access each other via a directory path. We might build a complex application with various assemblies in a set of

subdirectories under our main application's directory. To deploy our application to a new machine, we could simply use XCOPY to copy all the directories and files - nothing else is required.

The term "XCOPY deployment" is a reference to this level of simplicity.

.NET System Class Library

The CLR is an important part of .NET. However, merely having the ability to run code on many different hardware or operating systems is just a part of the overall puzzle. To be useful, applications need to interact with their environment - its files, data, processes, fonts, graphical components, etc.

Without a consistent way to interact with the environment across various operating systems, there's no way to create a complex business application. The CLR ensures that our code will compile in many places - but it is the .NET system class library that provides our code with a consistent way to interact with the environment.

The system class library is quite comprehensive. It has to be. We are used to programming within the Win32 environment - which is a pretty comprehensive "runtime'. To be successful, .NET has to provide a runtime with comparable features and functionality, essentially providing operating system capabilities while being operating system neutral.

> **If there's one thing that all .NET programmers must learn, it is the system class library.**

The system class library is far too large and complex to cover in this paper. However, what follows is a list of the top-level namespaces provided by the library:

Code DOM	Globalization	Security
Collections	IO	Service Process
Components	Messaging	System
Configuration	Microsoft.Win32	Text
Core	Net	Threading
Data	NewXML	Timers
Diagnostics	Reflection	Web
Directory services	Resources	WinForms
Drawing	Runtime	XML

The CLR and system class library compose the environment in which all .NET applications will run. A comprehensive understanding of the system class library and its features is critical for success regardless of the specific programming language being used by the developer.

As we'll see in the remainder of this paper, everything from a GUI to a web interface, and everything from data access to queuing, are accessed via the system class library.

Web Forms and ASP+

Active Server Pages (ASP) is one of the most popular and widely used programming tools for the Web. Despite its popularity, however, ASP is a relatively primitive environment, providing minimal services or support for the developer.

.NET addresses this by providing a much more comprehensive and cohesive programming environment for web developers. Rather than thinking of web pages as HTML sprinkled with

script code, we can now think of web pages as programmable forms - very similar in concept to the way we've thought of VB forms for years.

In fact, creating a web page using ASP+ is different from ASP in several ways, perhaps the most notable being the extensive and easy use of server-side controls for most rendering.

Rather than using the old-fashioned <INPUT> tag to get some text from the user, we can use the more descriptive and powerful ASP+ construct:

```
<asp:TextBox id=TextBox1 runat="server"></asp:TextBox>
```

This control is quite intelligent. It renders itself to the browser using an appropriate <INPUT> tag, so any browser can display its information. However, within our application code it appears as an object with which we can interact.

For instance, if we have the TextBox1 control shown above, along with a label control on a page, we can write code as follows:

```
Protected Sub TextBox1_TextChanged(ByVal sender _
    As System.Object, ByVal e As System.EventArgs)

  label1.Text = textbox1.Text

End Sub
```

When the user enters some text and presses *Enter*, the page will be submitted back to the server, where our code will run - setting the label text to the text entered by the user. The resulting page will be returned to the browser for display to the user.

No more messing around with the Session object or any of that other ugly state management - the ASP+ environment handles it all for us.

In fact, ASP+ uses highly scalable state management techniques for this type of thing, enabling us to use web farms for load balancing without resorting to sticky IP numbers or IP forwarding technologies. None of the state is held on the web server, providing a scalable mechanism for state management.

While designing and developing web pages may never be as easy as working with a traditional VB form, .NET comes incredibly close to providing exactly that type of functionality.

Win Forms

As has probably become apparent, .NET is not merely a whole new way to do web programming - it is a whole new way of doing Windows programming as well.

Historically, each Windows programming language has had its own way of handling the user interface. VB provides its type of forms, C++ its dialogs, VBA delivers a different form designer, and so on.

As with ASP+ and Web Forms, .NET provides a single, unified way to create user interfaces outside of a browser environment - **Win Forms**.

Win Forms provide a language-neutral way to create forms, add controls, and respond to the user's actions. For VB programmers there's not a lot of difference - the model continues to work the same. Drag and drop to build the form, double-click to open the code window, and write code to respond to the events.

Behind the scenes things are different, however. Win Forms are not based on ActiveX, so the controls we're working with are not ActiveX controls. Rather, they are instances of classes in the .NET system class library. As with ActiveX, we can create our own controls - though now this is done by subclassing existing classes, using inheritance.

Though the capabilities of Windows forms have changed little, the underlying technologies have been replaced with more powerful and integrated components and classes that fit smoothly into .NET.

Web Services

One of the hottest topics recently has been the Simple Object Access Protocol (SOAP). SOAP is a standard by which programs can call methods on objects running on another computer across the Internet.

SOAP utilizes XML to package up the data being transferred, and current implementations handle the method call via HTTP - going right through most firewalls and enabling a truly programmable Internet.

Web Services are an implementation of this technology. Based on SOAP, Web Services are methods that can be called by other programs, on other machines, across the Internet. SOAP enables programs written in C on Unix to communicate seamlessly with objects written in VB under Windows, and vice versa. SOAP is platform and language independent, enabling unprecedented interoperability for our applications.

SOAP is likely to forever alter the way we view Internet programming. Web Services are particularly exciting because this .NET technology abstracts and hides all the complexity of SOAP - making it as simple to call a method on an object across the Internet as it is to call a method of an object running right within our own application.

ADO+

Every 2-3 years Microsoft develops a new and better data access technology. First we had DAO, then RDO, and most recently ADO. ADO was pretty good – but not particularly ideal in the highly scalable and often disconnected world of web development.

ADO+ is a new data access technology that is optimized for disconnected operations. It is XML based, and offers a great deal of flexibility over its predecessors. Some of its new features include:

- Integrated XML support
- Enhanced support for disconnected data sets
- Easier data sharing
- Strongly typed data
- Increased performance

ADO+ allows us to establish a connection to a data source, retrieve that data, map the data into an XML-based DataSet, then distribute that DataSet object as needed. The DataSet object is designed for disconnected operation.

Of course, data binding remains available, both in Web Forms and Win Forms. From a programmer's perspective, data binding is not much different to the way it has been since VB6. We still set properties on our controls to indicate where to get the data and which data field or fields are to be displayed.

As with virtually all the technologies we're discussing here, ADO+ is accessed via the .NET system class libraries - in this case via the System.Data.ADO namespace.

Languages

Microsoft .NET will include Visual Studio.NET at some point in the future. VS.NET is the next step in the evolution of Microsoft's development tools, and includes:

- Visual C++
- C#
- VB.NET
- Jscript
- Visual FoxPro

Visual C++ will allow programming in both native Win32 and within .NET. In the Win32 environment, C++ allows developers to take advantage of MFC and ATL technologies - much as they do today. Within the .NET environment, a C++ program is managed by the .NET runtime just like any other code. The new C++ has language extensions to enable programmers to write code more easily within the managed environment.

C# is a new language designed by Microsoft for the .NET Framework. C# has a C-like syntax, but is designed to fit directly into the .NET environment. All C# code is managed code, running within the context of .NET. However, C# can be used to create *unsafe* code - code that uses pointers and could potentially alter the call stack, etc.

Visual Basic.NET (VB.NET) is an enhanced version of VB designed to work within the .NET Framework. VB.NET adds important new features to the language, including inheritance, constructors, and method overloading. It also gains all of the capabilities provided by the .NET runtime, including multithreading, object serialization, the ability to create Web Forms and Win Forms, etc.

Though originally a script language, Jscript is part of the .NET environment. Within .NET however, Jscript is a compiled language, (like all .NET languages) offering increased performance as well as increased features and functionality.

Visual FoxPro remains part of VS.NET even though it is not designed to create managed code within the .NET runtime. VFP continues to create native Win32 applications.

Other vendors are working on languages for the .NET runtime, including Fujitsu COBOL. There are several other languages in the works as well, including Perl and perhaps even an implementation of Java for .NET.

Summary

The Microsoft .NET Framework, consisting of the Common Language Runtime along with the .NET System Class library, provides us with revolutionary new programming capabilities.

This new environment supports both web and Windows developers in a consistent manner. Microsoft .NET is language neutral and is designed to provide a rich programming environment independent of any specific programming language.

The .NET Framework is, to a large degree, an abstraction layer sitting on top of the operating system. It abstracts the operating system in much the same way as an operating system abstracts computer hardware - enabling programs to run not only across various hardware configurations, but also across various operating systems.

The .NET system class library offers a very rich set of functionality that can be easily tapped by applications regardless of programming language. In many ways, the biggest leap for developers entering .NET will be to familiarize themselves with this class library and its capabilities.

Microsoft.NET is exciting and powerful - enabling a truly programmable Internet.