Issue	
Section	
Main file name	
Listing file name	
Sidebar file name	
Table file name	
Screen capture file names	
Infographic/illustration file names	
Photos or book scans	
Special instructions for Art dept.	
Editor	
Status	
Spellchecked (set Language to English U.S.)	
PM review	
Character count	
Package length	

Overline:

Byline:
By Juval Lowy

What you need:
Visual Studio.NET Beta 2 or better

Head:

Learn the Ropes of .NET

Deck:

This revolutionary product is the real deal.

How many times have you heard hype about "revolutionary" technology, "groundbreaking" advancements, and so on? More than once, I suspect. When you hear vegetable peelers described as "cutting edge," you know something's got to give. Well, in the case of Microsoft's .NET, the clamor is justified. As Microsoft's next-generation component technology, the company has designed .NET from the ground up to simplify component development and deployment, while providing unprecedented programming language interoperability.

This article describes the core concepts of .NET, such as its runtime and assemblies. I'll portray .NET as a component technology, using COM as a reference model. I'll describe NET, but I won't show you how to use it.

The .NET Common Language Runtime

.NET is based on a Common Language Runtime (CLR) environment that manages every runtime aspect of your code. The "common" in this name means all .NET components, regardless of the language they were developed in, execute in the same runtime. The CLR is like a warm blanket surrounding your code, providing it with memory management, a secure environment to run in, object location transparency, concurrency management, and access to underlying operating system services. Because the CLR manages these aspects of your object's behavior, code targeting the CLR is called *managed code*.

The CLR provides an unprecedented level of language interoperability, allowing component reuse on a scale not possible with COM technology. COM also provides language independence, allowing two binary components developed in two different languages—such as Visual Basic and C++—to call each other's methods. But COM language interoperability works only at run time. At development time, .NET lets you derive seamlessly a component developed in one language from a component developed in another language. .NET can do this because the CLR is based on a strict type system. To qualify as a .NET language, all constructs in a language—such as class, struct, and primitive types—must compile to CLR-compatible types.

This language interoperability gain comes at the expense of existing languages and compilers. Existing compilers produce CLR-ignorant code—code that doesn't comply

with the CLR type system and isn't managed by the CLR. To address this issue, Visual Studio.NET includes four CLR-compliant languages: C#, Visual Basic.NET, JScript.NET, and Managed C++. Third-party compiler vendors are also targeting the CLR with more than 20 other languages, from COBOL to Eiffel.

Learn the .NET Programming Languages

All .NET programming languages use the same set of base classes, development environment, CLR design constraints, and CLR types. Compiling code in .NET is a two-phase process. First, you compile the high-level code to a generic language called Microsoft Intermediate Language (IL), similar to machine code. At run time, on first call into the IL code, the .NET CLR compiles the IL to native code and executes it as native code. Once compiled, you use the native code until the program terminates.

The IL is the common denominator of all .NET programming languages, so in theory, equivalent constructs in two different languages should produce an identical IL. As a result, all .NET programming languages are equal in performance and ease of development. The differences between the languages are aesthetic and based on personal preference. For example, to make C++ CLR-compliant, you must use numerous compiler directives and extensions—called managed extensions for C++—resulting in less readable code.

Similarly, Visual Basic.NET only vaguely resembles its Visual Studio 6.0 ancestor, which requires you to unlearn your VB6 practices. Only C#, the fresh .NET language, has no legacy. Microsoft derived C# from C++, combining the power of C++ with the ease of VB6, and offering you readable, CLR-compliant C++-like code. In fact, C# resembles normal C++ more than managed C++ resembles conventional C++. C# is also a pure object-oriented (OO) language, enabling you to write understandable, reusable OO code. I'll use C# in my code samples, but bear in mind you can also do all the code samples in VB.NET and managed C++.

Other features of .NET languages include common error handling based on exceptions and viewing every entity—including primitive types—as an object, resulting in a smoother programming model. The CLR has a rich, predefined set of exception classes you can use as is, or you can derive and extend them for a specific need. You can catch and handle an exception thrown from one language in another language.

Develop .NET Components

.NET makes building binary components easy. You don't need an ATL-like framework. You simply declare a class and it becomes a component (see Listing 1). One the most important principles of component-oriented development is separating interfaces from implementation. COM enforces this separation by having you separate the definitions of interfaces from the classes implementing them. .NET doesn't force you to make your class's public methods part of any interface, but doing so allows polymorphism between different implementations of the same interface.

Listing 1 has an interface definition as part of the code. You don't need IDL files. The reserved C# word *interface* allows you to define a type that is pure virtual, has no implementation, and can't be instantiated by clients, similar to a C++ pure virtual or abstract class. The interface methods don't have to return HRESULT or any other error handling type. In case of an error, the method implementation should throw an exception.

Class Implementation

Implementing classes in .NET is straightforward. The class MyComponent in Listing 1 is defined as public, which makes it accessible by any managed client in other assemblies—a term I'll define soon—as well as any COM client once the component is exported to COM. You can define a class constructor to do object initialization, but the destructor has different semantics than a classic C++ destructor because .NET uses nondeterministic object. The destructor is called when garbage collection takes place, and because of that its execution time is nondeterministic. You can also implement other methods to do object cleanup, called explicitly by the object client. Finally, the implementation of ShowMessage() uses the static Show() method of the MessageBox class. As in C++, you can call a class static method without instantiating an object first.

Write Client-Side Code

All a managed client must do to use a component is add a reference in its project settings to the assembly that contains the component, create an object, and use it. Here is the client code corresponding to the component in Listing 1:

```
using MyNamespace;

//Interface-based programming:
IMessage myObj;
myObj = new MyComponent();
myObj.ShowMessage();
```

Usually, you don't use pointers in C#. Instead, you reference everything using the "." (dot) operator. In addition, the client casts the newly created object to the IMessage interface. If the object doesn't support this interface, .NET throws an exception. You can also use the "as" operator to cast without an exception and check the returned type's value. The client can also call the method directly, but that negates many of the benefits of separating the interface from the type implementing it.

The *assembly* is the basic code-packaging unit in .NET. An assembly is a logical library, meaning the assembly can combine more than one physical DLL—groupings called *modules*—into a single deployment, versioning, and security unit. However, an assembly usually contains one DLL—the default in VS.NET—and you must use command-line compiler switches to incorporate more than one module in your assembly. An assembly isn't limited to containing only DLLs; it can contain an EXE as well. As a component developer, you usually develop components residing in a single- or multiple-DLL assembly, to be consumed by a client application residing in an EXE assembly. The code in the assembly—in the DLLs or the EXE—is only the IL code, and at run time the IL is compiled to native code.

An assembly contains more than the IL code. Every assembly contains *metadata,* a description of all the types declared in the assembly, and a *manifest,* a description of the assembly and all other required assemblies. The manifest contains various assembly-wide information, such as the assembly version information. The version information is the product of a version number, build, and revision number. All modules in the assembly share the same version number and are deployed together as one unit.

The assembly boundary serves as a .NET security boundary, and security permission is granted at the assembly level. All the components in an assembly share the same set of permissions.

Sharing Assemblies

Assemblies can be private or shared. A *private* assembly resides in the same directory of the application that uses it or is in its path. A *shared* assembly is in a known location, called the *Global Assembly Cache* (GAC). To add an assembly to the GAC, you simply drag and drop it into the GAC folder, or you use the .NET Administration tool. Once in the GAC, multiple applications, managed and unmanaged, can access the assembly.

To avoid conflicts in the GAC between different assemblies with the same name, a shared assembly must have a *strong name*, also known as signing the assembly. The strong name verifies the assembly's origin and identity, and only the original publisher can generate it. You generate the strong name using private/public encryption keys, created with the SN.EXE command-line utility. Future versions of VS.NET might allow you to do that from within the visual environment.

Assembly's Metadata

Each assembly must contain metadata. The metadata contains descriptions of all the types defined in the assembly, including interfaces, classes and their base classes, method signatures, properties, events, member variables, and custom attributes. The compiler generates the metadata automatically when it compiles your project's source files. You can view the assembly's metadata using the ILDASM utility.

Assembly's Manifest

Just as the metadata describes the type in your assembly, the manifest describes the assembly itself. The manifest contains the assembly version information, the locale information, and the assembly's strong name. The manifest also contains the assembly's types visibility—which types are public and can be accessed by other assemblies, and which types are internal and can be accessed only from within the assembly. Finally, the manifest contains the security permission checks to run on behalf of the assembly. As with the metadata, the compiler generates the manifest automatically during the assembly compilation. You can view your assembly's manifest using the ILDASM utility.

Base Classes

A component technology is more than a set of rules and guidelines on how to build components. A successful component technology must provide developers with a development environment and tools that allow them to develop components rapidly. For example, COM needed tools such as ATL and VB for its success.

As demonstrated in Listing 1, you don't need a hard-to-learn component development framework such as ATL to build binary managed components. .NET takes care of all the underlying plumbing for you. In addition, to help you develop your business logic faster, .NET provides more than 6,600 base classes, available in similar form to all languages. The base classes are easy to learn and apply. You can use the base classes as they are, or you can derive from them to extend and specialize their behavior.

.NET enforces strict inheritance semantics and inheritance conflict resolution. .NET doesn't allow multiple implementation inheritance. You can derive components from

only one concrete class. You can, however, derive from as many interfaces as you like. When you override a method in a base class, you must declare your intent explicitly. For example, if you want to override it, you use the *override* reserved word; if you want to hide it, you use the *new* reserved word.

Component Visibility

While developing a set of interoperating components, you often have components intended only for private use that shouldn't be shared with your clients. Under COM you had no easy way of keeping your components private. The client could always hunt through the registry, find your private component's Class Identifier (CLSID), and use it. In .NET, you can use the *internal* keyword on the class definition instead of public, as in Listing 1. When you use internal, the runtime denies access to your component for any caller outside your assembly.

Attribute-Based Programming

When developing components, you can use attributes to declare your component needs, instead of coding them. This ability is analogous to COM developers today declaring the threading model attribute of their components. .NET provides numerous attributes, allowing you to focus on your domain problem at hand. For example, you access component services through attributes. You can also define your own attributes or extend existing ones.

Simplified Component Deployment

.NET doesn't rely on the Registry for anything to do with your components. In fact, installing a .NET component is as simple as copying it to the directory of the application using it. .NET maintains tight version control, enabling side-by-side execution of new and old versions of the same component on the same machine. The result is zero-impact installation. By default, you can't harm another application by installing yours, which ends the predicament known as DLL Hell. The .NET motto is: "It just works." If you want to install components to be shared by multiple applications, you can install them in the GAC. If the GAC contains a previous version of your assembly, it keeps the previous version for clients built against the old version (see Figure 1). You can purge old versions as well, but that's not the default.

Simplified Object Lifecycle

.NET doesn't use a reference count to manage an object life cycle. Instead, it keeps track of accessible paths in your code to the object. As long as any client has a reference to an object, it's considered *reachable*. Reachable objects are kept alive; unreachable objects are considered garbage, so destroying them harms no one. A crucial CLR entity is the garbage collector. Periodically, the garbage collector traverses the list of existing objects. It detects unreachable objects using a sophisticated pointing schema and releases the memory allocated to those objects. As a result, clients don't have to increment or decrement a reference count on objects they create.

About the author:

Juval Lowy is a software architect and the principal of IDesign, a consulting company focused on .NET design and .NET migration. Juval also conducts training classes and conference talks on component-oriented design and development processes. He wrote "COM and .NET Component Services—Mastering COM+" (O'Reilly). Reach him at www.componentware.net.

Captions:

Figure 1.

Eliminate DLL Hell .NET allows side-by-side deployment of .NET assemblies, sparing you major grief. You can deploy assemblies locally to the application using them, and you can have multiple versions of a given assembly running simultaneously by different applications. Or, you can deploy the assemblies to the Global Assembly Cache (GAC), and the runtime will figure out the correct version to use based on the version numbers and strong name assigned to the assembly.

Pullquotes:

NET is based on a CLR environment that manages every runtime aspect of your code.

The CLR provides an unprecedented level of language interoperability

The IL is the common denominator of all .NET programming languages,