Issue	Fall/Winter 2002
Section	features
Main file name	**nm0101jut11.rtf**
Listing file name	
Sidebar file name	
Table file name	
Screen capture file names	**nm0101juf1.bmp, nm0101juf2.bmp**
Infographic/illustration file names	
Photos or book scans	
Special instructions for Art dept.	
Editor	DD
Status	tech review by BN; content edited by DD (t3); reviewed by author (t4); revised by DD (t5); submitted to ME (t6); final tech review(t7); NF copyedit (t8); RMA for online version (9o); NP changed to make online version the print version and take out listings (t11)
Spellchecked (set Language to English U.S.)	Yes
EN review	approved raw
Character count	39,373
Package length	10 pages
ToC blurb	.NET was designed from the ground up to simplify component development and deployment while providing unprecedented programming language interoperability.

Section:

.NET Essentials

Overline:

.NET Overview

Byline:

by Juval Lowy

Head:

Set a New Course With .NET

Deck:

.NET provides frameworks, a new runtime and assemblies, a component technology, development tools, and programming language interoperability on an unprecedented scale.

Microsoft .NET is a revolutionary new component technology, available with the next release of Visual Studio. .NET offers new application frameworks such as Web Services, ASP.NET, Windows Forms, and ADO.NET. As Microsoft's next generation component technology, .NET is designed from the ground up to simplify component development and deployment while providing programming language interoperability on an unprecedented scale.

This overview describes the core concepts of .NET, such as its runtime and assemblies, and it provides a brief overview of .NET's various frameworks. It also portrays .NET as a component technology, using COM as a reference model. My aim is to provide you with distilled information on .NET, not to show you how to use it. I'll present enough information so you know what it is and where to look for more information so you can make educated decisions and start aligning your product roadmap with this new technology. I will not cover all of .NET's technologies because some of them are covered in some detail already in other articles in this issue.

.NET is based on a Common Language Runtime (CLR) environment that manages every runtime aspect of the 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 that surrounds the 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 an object's behavior, code that targets 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 a developer seamlessly derive 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 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, the high-level code is compiled 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, the native code is used 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, a developer must use numerous compiler directives and extensions—called managed extensions for C++—resulting in less readable code.

Similarly, Visual Basic .NET bears only some resemblance to its Visual Studio 6.0 ancestor, which requires a developer to unlearn VB6 practices. Only C#, the fresh .NET language, has no legacy. C# was derived from C++, combining the power of C++ with the ease of VB6, and offering 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 a developer to write understandable and reusable OO code.

I'll use C# in my code samples, but bear in mind it's possible to 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 that can be used as-is, or derived and extended them for a specific need. An exception thrown from one language can be caught and handled in another language.

The good news: .NET makes building binary components easy. For example, .NET doesn't require an ATL-like framework; a developer simply declares a class, and it becomes a component (download Listing 1 from the *.NET Magazine* Web site; see the Go Online box for details).

One the most important principles of component-oriented development is separating interfaces from implementation. COM enforces this separation by making the developer separate the definitions of interfaces from the classes implementing them. .NET doesn't force a developer to make his or her class's public methods part of any interface, but doing so is imperative to allow polymorphism between different implementations of the same interface.

For example, Listing 1 has an interface definition as part of the code, but .NET doesn't require IDL files. The reserved C# word *interface* allows a developer 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.

Implement a Class

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 shortly—as well as any COM client once the component is exported to COM. A developer 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 finalization. I'll say more on this subject in the section on object life cycle. A developer can implement other methods to do object cleanup. Finally, the implementation of ShowMessage() uses the static Show() method of the MessageBox class. As in C++, a developer can call a class static method without instantiating an object first.

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:

```
using MyNamespace;

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

A developer doesn't usually use pointers in C#. Instead, everything is referenced using the "." (dot) operator. In addition, the client casts the newly created object to the IMessage interface, which is the .NET equivalent of QueryInterface(). 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 value of the returned type.

Because .NET doesn't enforce separation of interface from implementation, the client can create the object type directly:

```
using MyNamespace;

//Avoid doing this:
MyComponent myObj;
myObj = new MyComponent();
myObj.ShowMessage();
```

A developer should avoid writing client code this way, however, because it means the client code isn't polymorphic with other implementations of the same interface. That kind of client code also reduces interoperability between modules. Imagine one module creating the object and the other using it, and only the interface type being passed around and not the actual implementing type. Microsoft wants to make building components easier for developers who understand object-oriented programming but have a hard time with components, so to bridge this gap, .NET doesn't enforce the separation of interfaces from implementation. Any seasoned C++ or VB6 COM developer should know better and should enforce the separation manually—both on the class side by implementing interfaces instead of just public methods, and on the client side by using the interface instead of the implementing type.

The basic code-packaging unit in .NET is the *assembly*. An assembly is a logical library, which means it can combine more than one physical DLL—groupings called *modules*—into a single deployment, versioning, and security unit. However, an assembly usually contains just one DLL—the default in VS.NET—and the developer must use command-line compiler switches to incorporate more than one module in an assembly. An assembly is not limited to containing only DLLs; it can contain an EXE as well. A component developer usually develops components that reside in a single- or multiple-DLL assembly, to be consumed by a client application residing in an assembly that has an EXE. 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.

Drill Down on Assemblies

An assembly contains more than just 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.

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, a developer can simply drag and drop it into the GAC folder, or use the .NET Administration tool. Once in the GAC, the assembly is accessible by multiple applications, managed and unmanaged alike.

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 can't be generated by a party other than the original publisher. The strong name is generated using private/public encryption keys, created using the SN.exe command line utility. Future versions of VS.NET might allow doing that from within the visual environment.

Each assembly must contain metadata. Metadata is the .NET equivalent of COM's type libraries, except that the metadata resembles a COM type library on steroids. The metadata contains descriptions of all the types defined in the assembly—such as interfaces, classes and their base classes, method signatures, properties, events, member variables, and custom attributes. The compiler generates the metadata automatically when it compiles the source files of a project. The ILDASM utility lets a developer view the metadata of an assembly.

Just as the metadata describes the type in an 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. The ILDASM utility also lets a developer view the manifest of an assembly.

.NET Differs From COM

.NET attempts to improve on COM's deficiencies while maintaining core COM concepts—concepts that have proved themselves as important principles of component-oriented development. For example, .NET provides binary compatibility between client and component, separation of interface from implementation, object location transparency, concurrency management, security, and language independence. I won't attempt a comprehensive discussion of .NET as a component technology, but it's important you understand the main characteristics of .NET as a component technology.

.NET appears to be missing many the things you might take for granted as part of developing components when compared to COM. However, the missing elements are actually present in .NET, albeit implemented differently:

For example, NET includes no canonical base interface, such as IUnknown, that all components derive from. Instead, all components derive from the class System.Object. This means every .NET object is polymorphic with System.Object. Similarly, .NET lacks class factories. Rather, the runtime resolves a type declaration to the assembly containing it and the exact class or value type (such as a struct) in the assembly. .NET also lacks object reference counting, relying instead on a sophisticated garbage-collection mechanism that detects when clients no longer use an object and then destroys the object.

The differences don't end there. .NET lacks IDL files or type libraries describing interfaces and custom types. Instead, a developer must put those definitions in the source code. The compiler is responsible for embedding the type definitions in a special format in an assembly's metadata. Also, .NET foregoes Globally Unique Identifiers (GUIDs), providing uniqueness of type (class or interface) by scoping the types with the namespace and assembly name. This means the assembly must contain a strong name when sharing an assembly between clients. In other words, .NET has identifiers that are globally unique, but they don't have to be managed anymore by the developer, and they are not GUIDs.

You need to keep in mind a couple other differences, as well. First, .NET doesn't have apartments. By default, every .NET component executes in a free-threaded environment, and it's up to the developer to synchronize access to components. A developer provides synchronization by relying on the .NET synchronization infrastructure. Second, .NET has no connection point protocol to subscribe and notify the developer of events. Like any other types, events are part of the CLR type system with clear, type-safe semantics for establishing the connection and publishing events. Finally, a developer doesn't need the Registry to keep track of components' locations and marshaling information. Components either are used privately by other assemblies or registered in the GAC. Marshaling type information is kept in every assembly's metadata.

.NET Improves Development Environment

The changes in .NET are extensive. One aspect of this is that .NET provides a superb development environment whose semantics are the product of years of observing the way developers use COM and the hurdles they face. Key new aspects of this development environment include the .NET Base Classes, component inheritance, component visibility, attribute-based programming, component-oriented security, simplified component development, simplified object lifecycle, and nondeterministic finalization:

The .NET Base Classes make .NET's component technology much 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, developers 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 them. It also provides more than 3,500 base classes, available in similar form to all languages. These base classes help a developer implement business logic faster. Even better, the base classes are easy to learn and apply. This means the base classes can be used as-is, or derived from and extended to specialize their behavior.

.NET enforces strict inheritance semantics and inheritance conflict resolution. For example, .NET doesn't allow multiple implementation inheritance. This means a developer can derive components from only one concrete class. It's possible to derive from as many interfaces as desired, however. When overriding a method in a base class, a developer must declare his or her intent explicitly. If a developer wants to override the base class method, he or she uses the *override* reserved word; if a developer wants to hide it, he or she uses the *new* reserved word.

Another factor that contributes to better development environment: component visibility. While developing a set of interoperating components, a developer often has components that are intended only for private use and should not be shared with clients. Under COM, they had no easy way of keeping components private. The client could always hunt through the registry, find the Class Identifier (CLSID) of the private component, and use it. In .NET, a developer can use the *internal* keyword on the class definition instead of public. When they use the internal keyword, the runtime denies access to the component for any caller outside the component's assembly.

.NET's attribute-based programming lets a developer use attributes to declare a component's needs instead of coding them. This is analogous to COM developers today who declare the threading model attribute of their components. .NET provides numerous attributes that let a developer focus on the domain problem at hand. For example, component services are accessed through attributes. This means a developer can also define new attributes or extend existing ones.

.NET component-oriented security is a departure from the classic Windows NT security model, which is based on what a given user is allowed to do. This model has evolved during a time when COM was in its infancy and applications were usually standalone, monolithic chunks of code. In today's highly distributed, component-oriented environment, a security model should be based on what a given piece of code—a component—is allowed to do, not only what its caller is allowed to do.

.NET allows a developer to configure permissions for a piece of code and provide an *evidence* to prove that the code has the right credentials to access a resource or perform some sensitive work. Evidence-based security is related tightly to the component's origin. System administrators can decide they trust all code that comes from a particular vendor but distrust everything else, from downloaded components to malicious attacks. A component can also demand that a permission check be performed to verify that all callers in its call chain have the right permissions before it proceeds to do its work.

This model complements COM+ role-based security and call authentication, providing the application administrator with granular control over not only what the users are allowed to do, but also what the components are allowed to do.

It Just Works

.NET's features don't just give a developer power and flexibility. They make things simpler for a developer. For example, .NET simplifies the process of component deployment. Specifically, .NET doesn't rely on the Registry for anything that has to do with components. 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 (see Figure 1). The result is zero-impact installation. By default, installing a new application can't harm another application. This ends the predicament known as DLL Hell. The .NET motto is: "It just works." If a developer wants to install components to be shared by multiple applications, he or she can install them in the GAC. If the GAC contains a previous version of an assembly already, it keeps it to be used by clients built against the old version. A developer can purge old versions as well, but that's not the default.

Similarly, .NET doesn't use a reference count to manage an object life cycle—creating less work for the developer. Instead, .NET keeps track of accessible paths in the code to an 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. One of the crucial CLR entities is the garbage collector. Periodically, the garbage collector traverses the list of existing objects, and 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.

A side effect of garbage collection is non-deterministic finalization of objects. In COM, the object knows it's no longer required by its clients when its reference count goes down to zero. The object then does its cleanup and destroys itself. The ATL framework even calls a method on an object called FinalRelease(), letting the developer handle the object cleanup.

In .NET, unlike COM, the object itself is never told when it's deemed garbage. If the object has specific cleanup to do, it should implement a method called Finalize(). The garbage collector calls Finalize() just before destroying the object. Finalize() is .NET's component destructor. In fact, even a developer implements a destructor, the compiler converts it to a Finalize() method.

Simplifying the object lifecycle, however, comes with a cost in system scalability and throughput. If the object is holding on to expensive resources such as files or database connections, these resources are released only when Finalize() is called, which is done at an undetermined point in the future—usually when the process hosting the component runs out of memory.

In theory, .NET might never release the expensive resources the object holds, which hampers system scalability and throughput severely.

.NET offers two solutions to problems arising from nondeterministic finalization. The first is to implement methods on an object that allow the client to order an explicit cleanup of the expensive resources the object holds. If the object holds on to resources that can be reallocated, the object should expose methods like Open() and Close(). An object encapsulating a file is a good example. The client calls Close() on the object, which allows the object to release the file for other objects to use. If the client wants to access the file again, it can call Open() without re-creating the object. The more common case is when disposing of resources amounts to destroying the object. In this case, the object should implement a method called Dispose(). When a client calls Dispose(), the object should dispose of all its expensive recourses and the client should not try to access the object again. The problem with both Close() and Dispose() is they make sharing the object between clients much more complex than COM's reference counts. The

clients must coordinate which one of them is responsible for calling Close() or Dispose() and when Dispose() should be called, so the clients are coupled to each other.

The second solution to the problems presented by nondeterministic finalization: Use .NET Enterprise Services Just-In Time Activation (JITA). If a developer provides a destructor, it will be called as soon as the object deactivates itself, allowing it to dispose of the expensive resources it holds then and there, without waiting for the garbage collector. JITA gives managed components deterministic finalization, a service nothing else in .NET can provide out of the box.

COM and .NET Interoperate Well

COM and .NET are fully interoperable with each other. Any COM client can call managed objects, and any COM object is accessible to a managed client. To export.NET components to COM, use the TlbExp.exe utility. The TlbExp utility generates a type library that COM clients use to create managed types and interfaces. A developer can run regasm.exe to get the .NET components registered as COM components. He or she can also use various attributes on a managed class to direct the export process, such as providing a CLSID and IID.

To import an existing COM object to .NET—by far the more common scenario—a developer uses the TlbImp.exe utility. The TlbImp utility generates a managed wrapper class, which a managed client uses. The wrapper class manages the reference count on the actual COM object. When the wrapper class is garbage collected, the wrapper releases the COM object it wraps. A developer can also import a COM object from within the VS.NET environment by selecting the COM object from the project reference dialog—making VS.NET call TlbImp for him or her.

.NET supports invoking native Win32 API calls, or any DLL-exported functions, by importing the method signatures to the managed environment. The developer is usually required to provide marshaling directives to direct the import process. It's likely that a tool will be available in the future to automate the generation of import directives.

With all its innovations and advanced concepts, .NET is only a component technology; it provides a developer with the means to build binary components rapidly. COM, too, is a component technology, and Microsoft intends .NET to succeed COM over time as the mainstream, most pervasive component technology ever developed. But that's all .NET is—a component technology. It does not have component services. .NET relies on COM+ to provide it with component services such as instance management, transactions, activity-based synchronization, granular role-based security, disconnected asynchronous queued components, and loosely coupled events. In fact, Microsoft renamed COM+ in .NET as *Enterprise Services*, which better reflects its pivotal role in .NET.

A .NET component that uses COM+ services is called a *serviced component*, and it must derive from the .NET base class ServicedComponent. A developercan configure a serviced component to use COM+ services in two ways. The first is COM-like use: he or she can derive the component from ServicedComponent, add the component to the COM+ Explorer, and configure it there. The second way is to apply special attributes to the component, configuring it at the source-code level. When a developer registers the component with COM+ it is configured automatically according to the values of those attributes.

.NET makes it possible to apply the serviced component attributes with great flexibility. If a developer doesn't configure a service with attributes, it's configured usually according to the default COM+ settings when that component is registered with COM+. A developer can apply as many attributes as he or she likes, although some COM+ services can be applied only through the COM+ Explorer. These are mostly deployment-specific configurations, such as persistent

subscriptions to COM+ Events and allocation of users to roles. In general, almost everything a developer can do with the COM+ Explorer can be done with attributes. The recommended usage pattern for serviced components is to enter as many design-level attributes into the code as possible—such as transaction support and object pooling—and use the COM+ Explorer to configure deployment-specific details. For example, to configure a serviced component to use COM+ object pooling, use the ObjectPooling attribute. A developer can provide optional parameters for pool parameters:

```
[ObjectPooling(MinPoolSize = 3,MaxPoolSize = 10)]
public class MyComponent :ServicedComponent
{…}
```

From a configuration management point of view, the .NET integration with COM+ is better than COM under VS6 because .NET permits capturing design decisions in code instead of storing them separately in the COM+ Explorer.

Web Services: Behind the Hype

Web Services are the most exciting piece of technology in the .NET platform. Web Services allow a middle-tier component at one site to invoke methods on a middle-tier component at another site with the same ease as if the remote component were in a local assembly. The underlying technology facilitating Web Services is a serialization of the calls into XML packages using protocols such as Simple Object Access Protocol (SOAP) or HTTP GET/POST. XML-based calls are plain text, so they can be made across firewalls. This makes them an ideal transport mechanism for Web Services calls. .NET hides the required details successfully from the client and the server developer. All a Web Service developer must do is use the WebMethod attribute on the public methods exposed as Web Services (download Listing 2).

The class can derive from the WebService base class optionally, defined in the System.Web.Services namespace. Doing so provides easy access to application and session state objects.

Although .NET's marketing effort concentrates heavily on demonstrating how to build Web applications in .NET, Microsoft is not abandoning the desktop market. .NET includes a comprehensive framework for developing rich UI client applications for Windows called *Windows Forms*. Windows Forms projects are somewhere between MFC applications and VB6. Windows Forms projects provide VB6's productivity-oriented environment and ease of development—such as the properties window—but the code layout is much more exposed, as in an MFC application where you can see controls binding to handling methods.

Using Windows Forms, it's possible build almost every kind of Windows application, including ActiveX controls, SDI and MDI applications, and custom controls, but the default is a dialog-based application. VS.NET enables a developer to build Windows Forms using only C# or VB.NET, without managed C++. A Windows Forms project lets a developer drag and drop controls such as buttons and edit boxes to a form layout, which generates the binding code for the developer. Interestingly enough, unlike a classic Windows application where user events such as a button click are mapped to messages that are handled in code, a Windows Forms control handles events the .NET way—the event is mapped to a .NET delegate that's bound to a method in a class.

As in MFC, .NET provides a multitude of base classes in the System.Windows.Forms namespace—such as Button, DataGrid, and CheckBox—that can be used as-is or derived and

extended, depending on what's needed. Finally, Windows Forms provide a new development service called *Visual Inheritance*. It's possible to derive a form from an already existing component in a binary assembly. The user interface layout associated with the base component, such as buttons and controls on the form, is displayed in the visual editor in a special way. They can't be changed them, but it's possible to see where they are and how best to add the controls.

Another critical piece of .NET technology: ASP.NET, which completely overhauls classic ASP. ASP.NET is a new .NET framework used to develop dynamic HTML pages. Classic ASP has many limitations and deficiencies. It's a blend of static HTML and script code, executing either on the server or on the browser side. The resulting programming model is messy because it has no clear separation of user interface code from business logic, and it has many design limitations. ASP projects often result in unmaintainable spaghetti code that doesn't scale, both performance-wise and management-wise. In addition, Interdev (ASP's development tool) leaves much to be desired compared with the Visual Studio tools; it has limited debugging capabilities and requires a develop to master HTML for even simple rendering.

Simplify Web Development

ASP.NET's goal is to make it as easy to develop Web applications as it is to create desktop applications. A developer simply drops controls on a form—called a Web Form—and binds the control properties to a class members and event handlers. The controls execute on the server, (also called server-side controls) are smart enough to know how to render themselves in HTML, and even accommodate different browsers—all without the developer writing a single line of code.

User input on the browser side is collected and posted back to the server where it's stored as the form class data members and properties. The resulting programming model is manageable, extensible, and object-oriented, much like Windows Forms. An ASP.NET developer no longer needs to know HTML because rendering is done by the controls, and they have a clear separation of the user interface—the form layout in the visual designer—from the business logic, which is the class behind the form. The class associated with the Web form is written in a .NET language such as C#, and it's compiled at run time to native code, which provides a significant performance boost compared with the interpreted classic ASP script.

ASP.NET developers are first-class VS.NET citizens and can use the rich set of base classes available with the .NET framework, call and step into other components in other assemblies, and take advantage of the VS.NET Integrated Development Environment (including the debugger). ASP.NET provides many new features: automatic validation controls that execute on the client side to validate input and save round trips to the server; extensive tracing; control values caching, automating data binding of controls—such as a grid—to a data source; numerous new controls, such as the Calendar; and an easy way to define custom server controls.

Microsoft has released it's share of data access technologies through the years. It's about to release another. ADO.NET is a set of classes used to access data sources such as databases. ADO.NET is based on a new object model compared with classic ADO because it's oriented and optimized for disconnected remote access to the data sources over the Internet. ADO.NET decouples the data consumer from the data source and the platform it resides on by introducing a level of indirection using two classes—DataSet and DataAdapter.

DataSet is the basic data-container object, containing structured information on a set of tables. Each DataSet object is associated with a particular subclass of DataAdapter. That subclass is tailored to interact with a particular data source, such as SQLDataAdapter. This design pattern

provides the data source indirection because the DataSet interacts only with the generic interface of the DataAdapter base class.

To change a source type, all the data consumer must do is associate the DataSet with a different DataAdapter. The data set caters to doing disconnected work over the Internet because it can cache whole portions of a database and perform the synchronization once changes are committed. While in transit, the data is represented in XML and can pass through firewalls over a regular HTTP port. .NET Web Services methods can return DataSets as parameters, enabling remote data access for Web Services consumers. Another benefit of the DataSet class is that lets a developer access tables as a property of the DataSet. This means a developer doesn't have to know—or at least handcraft—SQL queries.

That said, it's still possible to use ADO.NET for whatever classic ADO was used for—namely, same-machine or Intranet data access and manipulation. VS.NET provides visual designer support for setting up a connection with a data source; all a developer must do is drag and drop a data source to a project from the Server Explorer, and VS.NET creates the template code for him or her to connect and bind to that data source.

IDE Adds Many RAD Features

The VS.NET Integrated Development Environment (IDE) looks overwhelming at first. The IDE is customizable, however, and a developer can select from several available IDE templates such as Visual C++ 6.0 to ease the transition. Many of the changes from the VS6 IDE center around increasing application development speed and improved productivity tools. The overall user ergonomics are improved, including toolbars that appear only when required and remain minimized when not in use. The overall IDE concept is integration. The same development environment is used by all .NET languages, such as C# and VB.NET, and the IDE can display almost any development-related information such as system processes, threads, message queues, performance counters, system services, Web Services, control panel applets, database tables, and query results.

The new IDE has improvements for existing elements. The class view is much improved, and it now shows namespaces, classes, base classes, methods and properties, interfaces, and available overrideables—virtual methods from the base class. A new object browser—similar to the one available in VB6—can display information about your project or any other assembly, often saving you the trouble of using Help. A developer edits most things using enhanced VB6-like a properties window; instead of just one item in the clipboard in VS6, it's now possible to store as many clips as you like in the Clipboard Ring (see Figure 2).

Perhaps the most impressive and important improvement is the debugger. It includes all the old windows, such as call stack and threads, plus new windows, and it lets you tab and dock all of them together.

But the IDE has many new features too. The code editor can collapse blocks of code like collapsing folders in the file Explorer and hide them from view, enabling a developer to focus only on the method he or she wants; the code editor can present line numbers, as well. The IDE checks what is typed constantly. If it finds a compilation problem, it underlines that code the way Word's spellchecker does when you type misspelled text. The specific warning or error can be learned from a tool tip by hovering over the underlined text.

A special help window keeps track of a developer's activities and suggests relevant help links dynamically. A task list window lists remaining tasks such as complication errors or any other entries the developer puts in. C# (but not VB.NET) also includes built-in support for

documentation. Entering inline comments in code using special XML tags enables VS.NET to auto-generate well-formatted Web page documentation of a project.

Author bio:
Juval Lowy is a software architect and the principal of IDesign, a consulting company focused on COM/.NET design. 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:

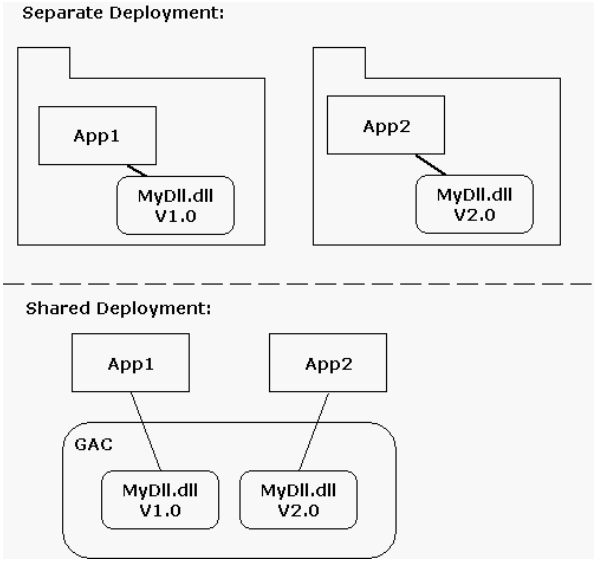
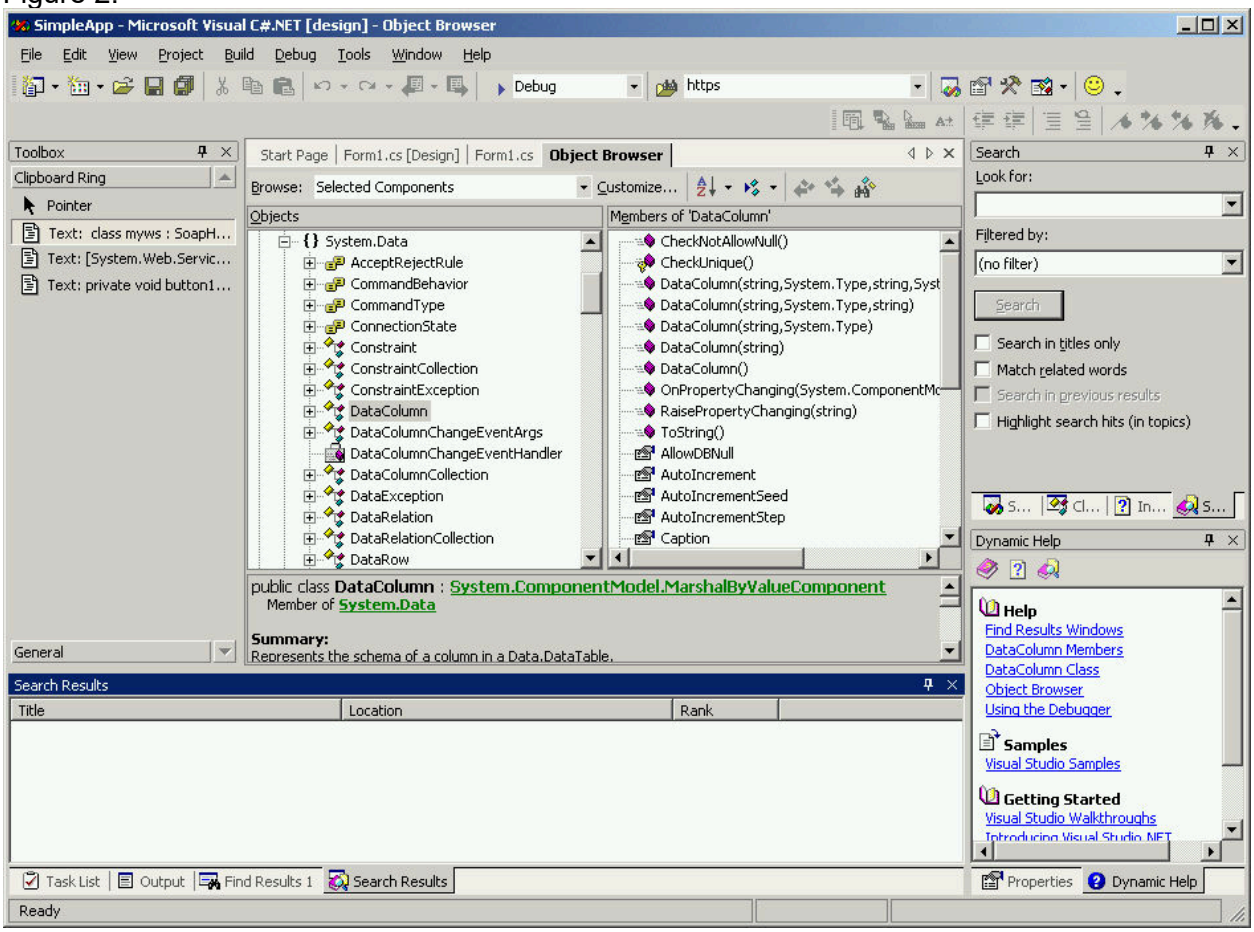


Figure 1.
Side-By-Side Deployment. .NET eliminates DLL Hell by allowing side-by-side deployment of .NET assemblies. Assemblies can be deployed locally to the application that uses them, and you can have multiple versions of a given assembly running simultaneously by different applications. Or you can 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.

Figure 2.



The Visual Studio .NET IDE. The IDE contains everything you need—and then some. Shown here are a few of the support windows that can be displayed in any Visual Studio project, including the Object Browser, Clipboard ring, integrated Dynamic Help topics and search capabilities. Even though the IDE presents an overwhelming amount of information in one place, it is completely manageable through an intuitive tabbed, retracting window management scheme.

Listing 1:

Build a Component. Building a component in .NET is straightforward: Just declare a class, and it becomes a binary component. You usually scope a component in a namespace and declare the namespaces you use. The interface definition is an integral part of you code, so you don't need a separate IDL file. Here a class implements the IMessage interface, displaying a message box with "Hello" in it when the ShowMessage() method is called.

```
namespace MyNamespace
{
  using System;
  using System.Windows.Forms  //For the MessageBox class

  public interface IMessage
  {
    void ShowMessage();
  }

  public class MyComponent : IMessage
  {
    public MyComponent(){}//constructor
    ~ MyComponent(){}//destructor
    public void ShowMessage()
    {
      MessageBox.Show("Hello!","MyComponent");
    }
  }
}
```

Listing 2:

Provide a Web Service. The MyWebService Web service provides the MyMessage service, which returns the string "Hello" to the caller. To qualify as a Web service, use the WebMethod attribute on the exposed Web services, and optionally derive from the WebService base class.

```
using System.Web.Services;

public class MyWebService : WebService
{
      public MyWebService(){}

   [WebMethod]
   public string MyMessage()
   {
     return "Hello";
   }
}
```