© 2002 Visual Studio Magazine
Fawcette Technical Publications

Issue	June 2002 VSM
Section	Online features
Main file name	Vs0206JLt4.rtf
Listing file name	Na
Sidebar file name	Na
Table file name	Na
Screen capture file names	Vs0206JLf2,3.bmp
Infographic/illustration file names	Vs0206JLf1.bmp
Photos or book scans	--
Special instructions for Art dept.	Don't use Figure 3. That's why the article doesn't refer to it.
Editor	LT
Status	TE'd
Spellchecked (set Language to English U.S.)	*
PM review	
Character count	10,444
Package length	2 ½ pages
ToC blurb	Component-oriented programming requires client/server binary compatibility; .NET lets you achieve this with metadata compatibility, which decouples client code from server code.

ONLINE SLUGS	
Magazine home page	VSM
[Features/ Columns/ Departments] [Author last name/ Column name/ Department name]	Online Feature
DART tag	vsm default
Title tag	<title>Visual Studio Magazine –Exploit Binary Compatibility in .NET</title>
Metatags	<!-- Start META Tags --> <meta name="Keywords" content=".NET, C#, binary compatibility, COM, interfaces, changing interfaces"> <meta name="DESCRIPTION" content=" Component-oriented programming requires client/server binary compatibility; .NET lets you achieve this with metadata compatibility, which decouples client code from server code "> <meta name="Author" content="Juval Lowy"> <meta name="Issue" content="Visual Studio Magazine, June 2002">

	<!-- End META Tags -->
180-character blurb	Component-oriented programming requires client/server binary compatibility; .NET lets you achieve this with metadata compatibility, which decouples client code from server code, and even lets you change interfaces.
90-character blurb	Get binary compatibility without having to change client code when you change server code.
90-character blurb describing download	Code includes a COM server and client projects and a .NET class library assembly, and Windows Forms client.
Locator+ code	VS0206
Locator+ text-only code	VS0206JL_T
Accompanying ZIP file name	vs0206JL.zip
Photo (for columnists) filename	
Photo (for columnists) pathname	SERVER_D/ EDIT/ 2002 Edit/ xxxxxxxx

RESOURCES ONLINE SLUGS	
DART tag	vsm default
Title tag	<title>Visual Studio Magazine Exploit Binary Compatibility in .NET</title>
Metatags	<!-- Start META Tags --> <meta name="Keywords" content=".NET, C#, binary compatibility, COM, interfaces, changing interfaces "> <meta name="DESCRIPTION" content=" Component-oriented programming requires client/server binary compatibility; .NET lets you achieve this with metadata compatibility, which decouples client code from server code "> <meta name="Author" content="Juval Lowy"> <meta name="Issue" content="Visual Studio Magazine, June 2002"> <!-- End META Tags -->

Overline:

Client/Server Development

Byline:
By Juval Lowy

Exploit Binary Compatibility With Metadata in .NET

Deck:

In .NET you can change server code and interfaces without having to make corresponding changes to client code. Here's how.

Component-oriented programming requires binary compatibility between client and server; .NET lets you achieve this without having to change your client code every time you tweak the server code. This is a big advance, because traditional object-oriented programming required all your client and server objects to belong to one monolithic application. During compilation of such OOP apps the compiler would bake the server entry point addresses into the client code. This methodology was inflexible.

Component-oriented programming models like COM and later .NET, revolve around packaging code into components—binary building blocks. By breaking a monolithic application into binary building blocks, you gain extensibility, reusability, and easier maintenance. Changes to server code are contained in the binary unit hosting it; you don't need to recompile and redeploy clients.

But the ability to replace and plug in new binary versions of the server implies binary compatibility between client and server. At run time client code must interact with exactly what it expects to find regarding the server entry points and binary layout in memory.

This binary compatibility forms the contract between server and client. As long as the new version of a server abides with this contract, clients aren't affected. However, early attempts at binary compatibility using DLLs with exported functions failed. The problems involved with DLLs versioning (commonly known as DLL Hell) outweighed the benefits of component-oriented programming.

COM was the first component technology to support binary commutability without DLL Hell, but COM is nothing if not a challenge to master. So as Microsoft's latest version of component technology, .NET seeks to maintain all of COM's core component-oriented principals with a fraction of the complexity and learning curve.

.NET's way of providing binary compatibility differs from that of COM. I'll first compare the COM and .NET binary compatibility models, then walk you through a demo application so you can experience the differences first hand.

COM provides binary compatibility by means of interface pointers and virtual tables. The client has an interface pointer, pointing to another pointer called the virtual table pointer. The virtual table pointer points to a table of function pointers. Each entry in the table points to where the corresponding interface's method code resides (see Figure 1). When the client uses the interface pointer to call a method (such as the second method of the interface), the compiler writes a jump command in the client's code. This command says to go to the address pointed by the second entry in the virtual table—in effect, just the offset from the table's address.

At runtime the loader patches this jump command to the actual address, because it already knows the table memory location. So COM's binary compatibility is based on offsets from the virtual table start address. Any server is considered binary compatible if it provides the same table layout and methods with exactly the same signature. Of course, this is nothing new—it's the exact definition of implementing a COM interface.

This model yields the famous COM Prime Directive: "Thou Shalt Not Change a Published Interface." Because any change will most likely to break existing client code. So, when you change the interface definition you must recompile and redeploy the clients as well.

"Leave Those Clients Alone"

.NET lightens the load by providing binary compatibility through metadata. High level code (such as C# or VB.NET) compiles to Intermediate Language (IL). At run time, the Just In Time (JIT) compiler compiles the IL to machine code and links it. The IL only contains requests to invoke or access a method or a field of an object, based on that type's metadata. Any type that provides these entry points in its metadata is binary compatible. All the client code contains is the name of the entry point based on metadata (a token uniquely identifying this point). This combines the flexibility of late binding with the safety of early binding; you'll never jump to the wrong address, yet you never bake actual entry point addresses or offsets into client code.

I prepared a sample project you can download (see Go Online box for details). My BinaryCopm solution contains an ATL 7 .0 COM project, an MFC 7.0 client for the COM object, a .NET class library assembly, and a .NET Windows Forms client (see Figure 2). Note how easy it is to mix all these project types within the same Visual Studio.NET solution. This will probably be the reality for many companies for the foreseeable future.

The COM ATL server defines this IDL interface:

```
interface IMyInterface : IDispatch
{
    [id(1)] _
        HRESULT Method1(void);
    [id(2)] _
        HRESULT Method2(void);
};
```

The server implementation simply pops up a message box indicating the method name:

```
STDMETHODIMP CMyServer::Method1()
{
    AfxMessageBox("Method1");

    return S_OK;
}
```

```
STDMETHODIMP CMyServer::Method2()
{
	AfxMessageBox("Method2");

	return S_OK;
}
```

The MFC client is a dialog application that invokes these methods.

Now build the solution and run the MFC client. Click on the buttons to make sure all is well. Next, open the client \Debug folder, and make a copy of the COMClient.exe. Name that copy OldCOMClient.exe. Change the order of the methods in the interface definition to:

```
interface IMyInterface : IDispatch
{
	[id(1)] _
	HRESULT Method2(void);
	[id(2)] _
	HRESULT Method1(void);
};
```

Run the old client. When you click the Method1 button you will actually invoke method 2 and vice versa. This is because COM binary compatibility is based on the virtual table offset, compiled into the old client code.
Remove Method1 from the interface definition:

```
interface IMyInterface : IDispatch
{
	[id(2), helpstring("method Method2")] _
		HRESULT Method2(void);
};
```

Remove Method1 from the server implementation. Build the COM server and run the old client again. When you click the Method2 button you'll crash the client, because the offset to Method2 points to a bogus address—the virtual table doesn't have a corresponding entry at all.

Now repeat these experiments on the .NET side. The class library assembly defines this C# interface:

```
public interface IMyInterface
{
	void Method1();
	void Method2();
}
```

Here too the implementation simply pops up a message box indicating the method name:

```
public void Method1()
{
	MessageBox.Show("Method 1",".NET Server");
}
public void Method2()
{
	MessageBox.Show("Method 2",".NET Server");
}
```

The .NET client is a Windows Forms form that invokes these methods. Run the .NET client and click the buttons to make sure all is well here. Next, open the client bin\Debug folder, and make a copy of the NETClient.exe. Name that copy OldNETClient.exe. Change the order of interface definition methods to:

```
public interface IMyInterface
{
      void Method2();
      void Method1();
}
```

Run the old client. When you click Method1 you invoke method 1, because the only thing hard-coded into the client code is the method name. Binding to the virtual table offset occurs at runtime during JIT compilation. Remove Method1 from the interface definition:

```
public interface IMyInterface
{
      void Method2();
}
```

Build the .NET class library assembly and run the old client. When you click Method2 you'll invoke method 2. And you won't crash, as you would have with the COM client.

The benefits of .NET binary compatibility and advantages over the COM model are clear. You can remove unused methods and add new methods (one of the common reasons for defining new interfaces in COM is adding new methods); you can change order of methods; and in general you gain late binding flexibility.

However, you can't change method parameters. Nor can you remove methods that clients expect to use. And if you choose to use the pre-JIT compiler (Ngen.exe) and generate a native image of your client, then you're back to square one: COM rigidity, because you now must hard-code virtual table offsets in the client code.

Subclass with Metadata

An interesting side effect of metadata-based binary compatibility is that it allows binary inheritance of implementation. You can have the base class in one assembly and the subclass in another. The subclass only has access to the base class in a binary format. In fact, this is how you use subclassing and extend classes in the .NET application frameworks. In traditional object-oriented programming, the subclass developer had to have access to a source file describing the base class in order to derive a class from it.

In .NET, you describe types using metadata, so the subclass developer need only access the metadata of the base class. The compiler reads the metadata from the binary file containing it. After that the compiler knows which methods and fields are available in the base type. If fact, even when both the base class and the subclass are defined in the same project, the compiler still uses the metadata in the project to compile the subclass. This is why in .NET the order in which classes are defined doesn't matter, unlike in C++, which often required forward type declarations, or a particular order of listing the included header files.

About the Author:

Juval Löwy is a software architect, conference speaker, and principal of IDesign, a consulting and training company focused on .NET design and migration. This article is based on excerpts from his upcoming book

on programming .NET components (by O'Reilly). Juval is an officer of the .NET California Bay Area User Group. Reach him at www.idesign.net

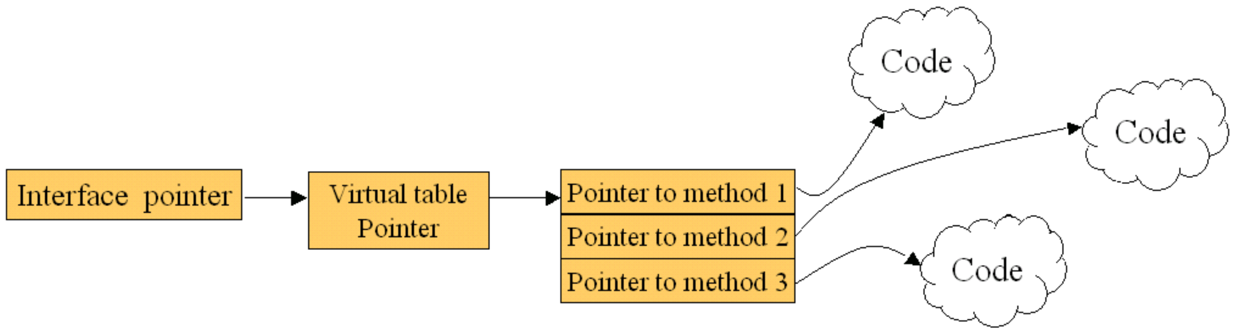

Figure 1:
COM Binary Compatibility: Effective But Rigid.
In COM, binary compatibility is based on virtual tables. The client code is compiled against known offset in the virtual table, and the interface pointer shields the client from the exact table location.

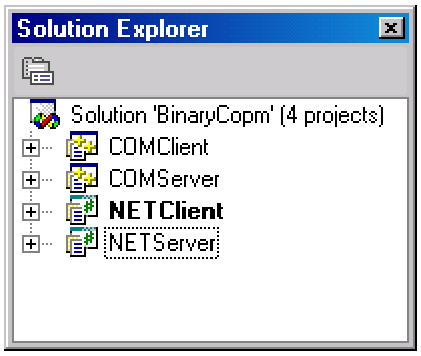

Figure 2:

.NET Binary Compatibility Solution: Unified development Environment
The BinaryCopm solution contains an ATL 7 .0 COM project, an MFC 7.0 client for the COM object, A a .NET class library assembly, and a .NET Windows Forms client. Note how Visual Studio.NET solutions can contain any project type, such as ATL, MFC, .NET class library assembly and Windows Forms.

FIGURE 1 ONLINE SLUGS	
DART tag	vsm default
Title tag	<title>**Visual Studio Magazine Exploit Binary Compatibility in .NET**</title>
Metatags	<!-- Start META Tags --> <meta name="Keywords" content="**.NET, C#, binary compatibility COM, interfaces, changing interfaces">** <meta name="DESCRIPTION" content=" In COM, binary compatibility is based on virtual tables.

	The client code is compiled against known offset in the virtual table, and the interface pointer shields the client from the exact table location. **">** <meta name="Author" content="**Juval Lowy**"> <meta name="Issue" content="**Visual Studio Magazine, June 2002**"> <!-- End META Tags -->

FIGURE 2 ONLINE SLUGS	
DART tag	vsm default
Title tag	<title>**Visual Studio Magazine – Exploit Binary Compatibility in .NET**</title>
Metatags	<!-- Start META Tags --> <meta name="Keywords" content="**.NET, C#, binary compatibility, COM, interfaces**, **changing interfaces**"> <meta name="DESCRIPTION" content="**.** The BinaryCopm solution contains an ATL 7 .0 COM project, an MFC 7.0 client for the COM object, a .NET class library assembly, and a .NET Windows Forms client. Note how Visual Studio.NET solutions can contain any project type, such as ATL, MFC, .NET class library assembly and Windows Forms. "> <meta name="Author" content="**Juval Lowy**"> <meta name="Issue" content="**Visual Studio Magazine, June 2002**"> <!-- End META Tags -->

Go Online

Use these Locator+ codes at www.visualstudiomagazine.com to go directly to these related resources.

VS0206 Download all the code for this issue of *VSM*.

VS0206JLO Download the code for this article separately. This article's code includes an ATL 7 .0 COM project, an MFC 7.0 client for the COM object, a .NET class library assembly, and a .NET Windows Forms client.

VS0206JLO_T Read this article online.