COM and .NET Interoperability

ANDREW TROELSEN

COM and .NET Interoperability
Copyright © 2002 by Andrew Troelsen

ISBN (pbk): 1-59059-011-2
Printed and bound in the United States of America 12345678910

Technical Reviewers: Habib Heydarian, Eric Gunnerson
Editorial Directors: Dan Appleman, Peter Blackburn, Gary Cornell, Jason Gilmore, Karen Watterson, John Zukowski
Managing Editor: Grace Wong
Copy Editors: Anne Friedman, Ami Knox
Proofreaders: Nicole LeClerc, Sofia Marchant
Compositor: Diana Van Winkle, Van Winkle Design
Artist: Kurt Krames
Indexer: Valerie Robbins
Cover Designer: Tom Debolski
Marketing Manager: Stephanie Rodriguez

Distributed to the book trade in the United States by Springer-Verlag New York, Inc., 175 Fifth Avenue, New York, NY, 10010 and outside the United States by Springer-Verlag GmbH & Co. KG, Tiergartenstr. 17, 69112 Heidelberg, Germany.

In the United States, phone 1-800-SPRINGER, email `orders@springer-ny.com`, or visit `http://www.springer-ny.com`.
Outside the United States, fax +49 6221 345229, email `orders@springer.de`, or visit `http://www.springer.de`.

For information on translations, please contact Apress directly at 2560 Ninth Street, Suite 219, Berkeley, CA 94710.
Phone: 510-549-5930, Fax: 510-549-5939, Email: `info@apress.com`, Web site: `http://www.apress.com`.

The source code for this book is available to readers at `http://www.apress.com` in the Downloads section. You will need to answer questions pertaining to this book in order to successfully download the code.

CHAPTER 12

COM-to-.NET Interoperability—Advanced Topics

The point of this chapter is to round out your knowledge of exposing .NET types to COM applications by examining a number of advanced techniques. The first major topic is to examine how .NET types can implement COM interfaces to achieve binary compatibility with other like-minded COM objects (a topic first broached in Chapter 7). Closely related to this topic is the process of defining COM types directly using managed code. Using this technique, it is possible to build a binary-compatible .NET type that does not directly reference a related interop assembly (and is therefore a bit more lightweight). As for the next major topic, you examine the process of building a customized version of tlbexp.exe while also addressing how to programmatically register interop assemblies at runtime. Finally, you wrap up by taking a deeper look at the .NET runtime environment and checking out how a COM client can be used to build a custom host for .NET types. In addition to being a very interesting point of discussion, you will see that a custom CLR host can simplify COM-to-.NET registration issues.

Changing Type Marshaling Using MarshalAsAttribute

Before digging into the real meat of this chapter, let's examine yet another interop-centric attribute. As you have seen, one nice thing about the tlbexp.exe utility is that it will always ensure the generated type information is [oleautomation] compatible. When you build COM interfaces that are indeed [oleautomation] compatible, you are able to ensure that all COM-aware languages can interact with the .NET type (as well as receive a free stub/proxy layer courtesy of the universal marshaler). Typically, if you have created a COM interface that is not [oleautomation] compatible, you have either (a) made a mistake, (b) are building a COM server you only intend to use from C++, or (c) wish to define a custom stub and proxy DLL for performance reasons.

Nevertheless, if you wish to create a managed method that is exposed to COM as a non–oleautomation-compatible entity, you are able to apply the MarshalAsAttribute type. The MarshalAs attribute can also be helpful when a single .NET type has the ability to be represented by multiple COM types. For example, a System.String could be marshaled to unmanaged code as a LPSTR, LPWSTR, LPTSTR, or BSTR. While the default behavior (System.String to COM BSTRs) is typically exactly what you want, the MarshalAsAttribute type can be used to expose System.String in alternative formats.

This attribute may be applied to a method return type, type member, and a particular member parameter. Applying this attribute is simple enough; however, the argument that is specified as a constructor parameter (UnmanagedType) is a .NET enumeration that defines a *ton* of possibilities. To fully understand the scope of the MarshalAs attribute, let's check out some core values of this marshal-centric enumeration. First up, Table 12-1 documents the key values of UnmanagedType that allow you to expose a System.String in various formats.

Table 12-1. String-Centric Values of UnmanagedType

String-Centric UnmanagedType Member Name	Meaning in Life
AnsiBStr	ANSI character string that is a length-prefixed single byte.
BStr	Unicode character string that is a length-prefixed double byte.
LPStr	A single-byte ANSI character string.
LPTStr	A platform-dependent character string, ANSI on Windows 98, Unicode on Windows NT. This value is only supported for Platform Invoke, and not COM interop, because exporting a string of type LPTStr is not supported.
LPWStr	A double-byte Unicode character string.

To illustrate, assume you have a small set of .NET members that are defined as follows:

```
[ClassInterface(ClassInterfaceType.AutoDual)]
public class MyMarshalAsClass
{
    public MyMarshalAsClass(){}

    // String marshaling.
    public void ExposeAsLPStr
        ([MarshalAs(UnmanagedType.LPStr)]string s){}
    public void ExposeAsLPWStr
        ([MarshalAs(UnmanagedType.LPWStr)]string s){}
}
```

Once processed by tlbexp.exe, you find the following COM IDL:

```
interface _MyMarshalAsClass : IDispatch
{
    [id(0x60020004)]
    HRESULT ExposeAsLPStr([in] LPSTR s);
    [id(0x60020005)]
    HRESULT ExposeAsLPWStr([in] LPWSTR s);
};
```

Table 12-2 documents the key values of UnmanagedType that are used to expose System.Object types as various flavors of COM types.

Table 12-2. System.Object-Centric Values of UnmanagedType

Object-Centric UnmanagedType Member Name	Meaning in Life
IDispatch	A COM IDispatch pointer
IUnknown	A COM IUnknown pointer

If you extend the MyMarshalAsClass type to support the following members:

```
// Object marshaling.
public void ExposeAsIUnk
    ([MarshalAs(UnmanagedType.IUnknown)]object o){}
public void ExposeAsIDisp
    ([MarshalAs(UnmanagedType.IDispatch)]object o){}
```

you find the following COM type information:

```
 [id(0x60020006)]
HRESULT ExposeAsIUnk([in] IUnknown* o);
[id(0x60020007)]
HRESULT ExposeAsIDisp([in] IDispatch* o);
```

UnmanagedType also provides a number of values that are used to alter how a .NET array is exposed to classic COM. Again, remember that by default, .NET arrays are exposed as COM SAFEARRAY types, which is typically what you require. For the sake of knowledge, however, Table 12-3 documents the key array-centric member of UnmanagedType.

Table 12-3. Array-Centric Value of UnmanagedType

Array-Centric UnmanagedType Member Name	Meaning in Life
LPArray	A C-style array

As you would guess, the following C# member definition:

```
// Array marshaling.
public void ExposeAsCArray
    ([MarshalAs(UnmanagedType.LPArray)]int[] myInts){}
```

results in the following IDL:

```
 [id(0x60020008)]
HRESULT ExposeAsCArray([in] long* myInts);
```

Finally, UnmanagedType defines a number of members that allow you to expose intrinsic .NET data types in various COM mappings. While many of these values are used for generic whole numbers, floating-point numbers, and whatnot, one item of interest is UnmanagedType.Currency. As you recall, the COM CURRENCY type is not supported under .NET and has been replaced by System.Decimal. Table 12-4 documents the key data-centric types.

Table 12-4. Data-Centric Values of UnmanagedType

Data Type-Centric UnmanagedType Member Name	Meaning in Life
AsAny	Dynamic type that determines the Type of an object at runtime and marshals the object as that Type.
Bool	4-byte Boolean value (true != 0, false = 0).
Currency	Used on a System.Decimal to marshal the decimal value as a COM currency type instead of as a Decimal.
I1	1-byte signed integer.
I2	2-byte signed integer.
I4	4-byte signed integer.
I8	8-byte signed integer.
R4	4-byte floating-point number.
R8	8-byte floating-point number.

Table 12-4. Data-Centric Values of UnmanagedType (continued)

Data Type-Centric UnmanagedType Member Name	**Meaning in Life**
SysInt	A platform-dependent signed integer. 4 bytes on 32-bit Windows, 8 bytes on 64-bit Windows.
SysUInt	Hardware natural-size unsigned integer.
U1	1-byte unsigned integer.
U2	2-byte unsigned integer.
U4	4-byte unsigned integer.
U8	8-byte unsigned integer.
VariantBool	2-byte OLE-defined Boolean value (true = -1, false = 0).

Again, the most useful of these data type-centric members of the UnmanagedType enumeration is the UnmanagedType.Currency value, given that .NET no longer supports the COM CURRENCY type. However, given that a System.Decimal provides the same storage, you can apply MarshalAs as follows:

```
// Exposing Decimal and Currency.
public void ExposeAsCURRENCY
    ([MarshalAs(UnmanagedType.Currency)]Decimal d){}
```

This results in the following IDL:

```
 [id(0x60020008)]
HRESULT ExposeAsCURRENCY([in] CURRENCY d);
```

So, now that you have seen the various ways that the MarshalAsAttribute type can be configured, you may be wondering exactly when (or why) you may wish to alter the default interop marshaler. In reality, you typically won't need to alter the default marshaling behavior. The only time it might be beneficial on a somewhat regular basis is when you wish to expose .NET System.Objects as a specific COM interface type (IUnknown or IDispatch) or expose a System.Decimal as a legacy COM CURRENCY type.

CODE *The MyMarshalAsLibrary project is included under the Chapter 12 subdirectory.*

.NET Types Implementing COM Interfaces

Recall from Chapter 9 that if a COM coclass implements a COM-visible .NET interface, the coclass in question is able to achieve type compatibility with other like-minded .NET objects. The converse of this scenario is also true: .NET types can implement COM interfaces to achieve binary compatibility with other like-minded COM types. When a .NET programmer chooses to account for COM interfaces in his or her type implementations, there are two possible choices:

- Implement a *custom* COM interface.
- Implement a *standard* COM interface.

As you recall from Chapter 2, although a COM interface always boils down to the same physical form (a collection of pure virtual functions identified by a GUID), standard interfaces are predefined types (published by Microsoft). Furthermore, standard interfaces are already defined in terms of COM IDL, have a predefined GUID, and are recorded in the system registry. Custom interfaces, on the other hand, are authored by a COM developer during the course of a software development cycle. In this case, the programmer is the one in charge of describing the item in terms of COM IDL and registering the resulting type library (all of which is done automatically when using VB 6.0). When a .NET type implements a custom COM interface, the result is that a given COM client is able to interact with the .NET type as if it were a coclass adhering to a specific binary format.

On the other hand, if a .NET type implements a standard interface (such as IDispatch, IConnectionPointContainer, or ITypeInfo), it will be used as a customized *replacement* for the equivalent interface implemented by the CCW. To be sure, the chances that you will need to provide a customized implementation of an interface supported by the CCW are slim to none. Given this likelihood, I focus solely on the process of defining managed versions of custom COM interfaces.

Defining Custom COM Interfaces

Before you can examine how to implement custom COM interfaces on a .NET type, you first need the IDL descriptions of the interfaces themselves. As you will see later in this chapter, it is possible to build a binary-compatible .NET type without a formal COM type description; however, for this example, assume you have created an ATL in-proc COM server (AnotherAtlCarServer). This COM server defines a coclass (CoTruck) by implementing two simple interfaces named

IStartable and IStoppable. Here is the relevant IDL (if you need a refresher on building COM servers with ATL, see Chapter 3):

```
[object,
uuid(7FE41805-124B-44AE-BEAE-C3491E35372B),
oleautomation,
helpstring("IStartable Interface"),
pointer_default(unique)]
interface IStartable : IUnknown
{ HRESULT Start(); };

[object,
uuid(B001A308-8D66-4d23-84A4-B67615646ABB),
oleautomation,
helpstring("IStartable Interface"),
pointer_default(unique)]
interface IStoppable : IUnknown
{ HRESULT Break();};

[uuid(7B69AEB6-F0B7-46BB-8AD4-1CACD1EA5AE9),
version(1.0),
helpstring("AnotherAtlCarServer 1.0 Type Library")]
library ANOTHERATLCARSERVERLib
{
    importlib("stdole32.tlb");
    importlib("stdole2.tlb");

    [uuid(862C5338-8AD7-43A3-A9A7-F21B145D61D0),
    helpstring("CoTruck Class")]
    coclass CoTruck
    {
        [default] interface IStartable;
        interface IStoppable;
    };
};
```

The implementation of the CoTruck::Start() and CoTruck::Break() methods simply triggers a Win32 MessageBox() API to inform the caller which object has been told to do what:

```
STDMETHODIMP CCoTruck::Start()
{
    MessageBox(NULL, "The truck as started.",
        "CoTruck::Start() Says:", MB_OK);
    return S_OK;
}
```

```
STDMETHODIMP CCoTruck::Break()
{
    MessageBox(NULL, "The truck as stopped.",
        "CoTruck::Start() Says:", MB_OK);
    return S_OK;
}
```

That's it. Go ahead and compile this ATL project to ensure that this COM server is properly recorded in the system registry.

> **CODE** *The AnotherAtlCarServer project can be found under the Chapter 12 subdirectory.*

Building and Deploying the Interop Assembly

Now that you have a COM server defining a set of custom interfaces, you need to transform the COM type information into terms of .NET metadata. Thus, assuming you have a valid *.snk file, configure a strongly named interop assembly using tlbimp.exe as follows:

```
tlbimp AnotherAtlCarServer.dll /out:interop.AnotherAtlCarServer.dll
/keyfile:theKey.snk
```

Finally, deploy this interop assembly into the GAC (Figure 12-1).

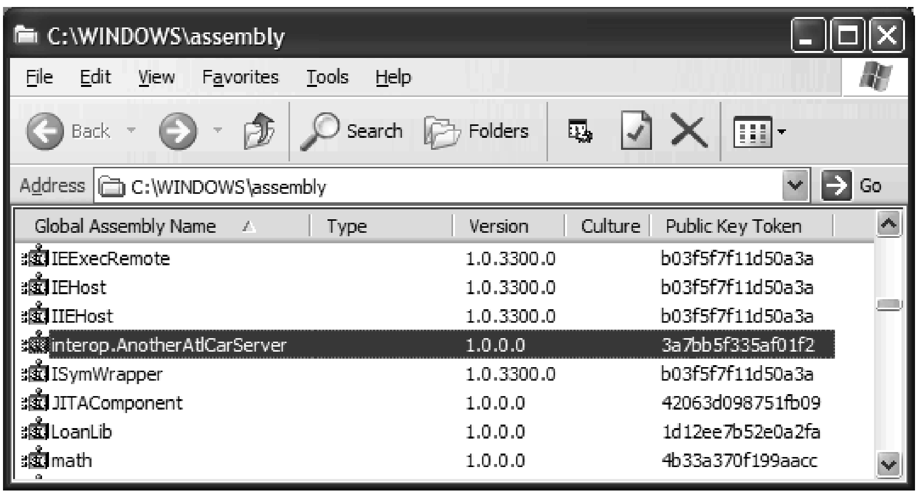

Figure 12-1. Another machine-wide interop assembly

> **CODE** *The interop assembly for AnotherAltCarServer.dll is included under the Chapter 12 subdirectory.*

Building a Binary-Compatible C# Type

To illustrate building a binary-compatible .NET type, let's create a new C# Code Library that defines a simple class (DotNetLawnMower) that supports both interfaces. First, add a reference to interop.AnotherAtlCarServer.dll, and for simplicity, configure this type to be exposed to COM as an AutoDual class interface:

```
namespace BinaryCompatibleDotNetTypeServer
{
    // This .NET class supports two COM interfaces.
    [ClassInterface(ClassInterfaceType.AutoDual)]
    public class DotNetLawnMower: IStartable, IStoppable
    {
        public DotNetLawnMower(){}
    }
}
```

Now that DotNetLawnMower has defined support for IStartable and IStoppable, and you are obligated to flesh out the details of the Start() and Break() methods. While you could manually type the definitions of each inherited member, you do have a shortcut. The Visual Studio .NET IDE supports an integrated wizard that automatically generates stub code for an implemented interface. However, the manner in which you interact with this tool depends on your language of choice. Here, in your C# project, you activate this tool by right-clicking a supported interface using Class View (Figure 12-2).

Again, the implementation of each member is irrelevant for this example, so just set a reference to System.Windows.Forms.dll and call MessageBox.Show() in an appropriate manner:

```
public void Start()
{
    MessageBox.Show("Lawn Mower starting..." ,
        "DotNetLawnMower says:");
}
public void Break()
{
    MessageBox.Show("Lawn Mower stopping..." ,
        "DotNetLawnMower says:");
}
```

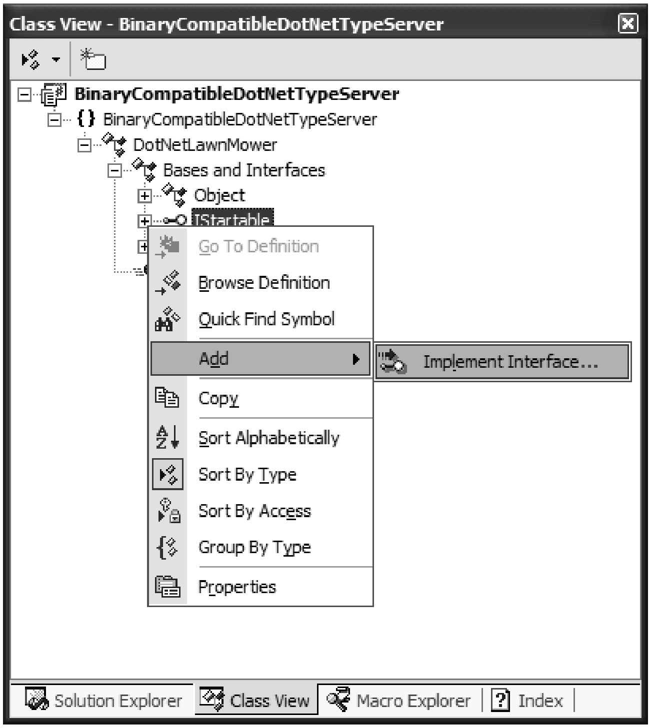

Figure 12-2. The C# IDE Implement Interface Wizard

Because this .NET class library is to be used by a classic COM client, you will want to deploy this binary as a shared assembly. Thus, be sure to set the assembly's version (1.0.0.0 will do) and specify a valid *.snk file. Once you have done so, deploy this assembly to the GAC.

CODE *The BinaryCompatibleDotNetTypeServer project is included under the Chapter 12 subdirectory.*

Building a Binary-Compatible VB .NET Type

Any managed language has the ability to implement COM interfaces, provided they have access to the interface descriptions. To further highlight the process, assume you have a VB .NET Code Library that defines a type named UFO. The UFO type is able to be started and stopped (presumably) and thus wishes to implement the COM interfaces defined in the ATL server. Once you set a reference to the interop assembly and define support for each interface (via the Implements keyword), the VB .NET IDE provides a simple shortcut to automatically build stubs for each method. Simply select the name of the supported interface from the left drop-down list and the name of the method from the right drop-down list (Figure 12-3).

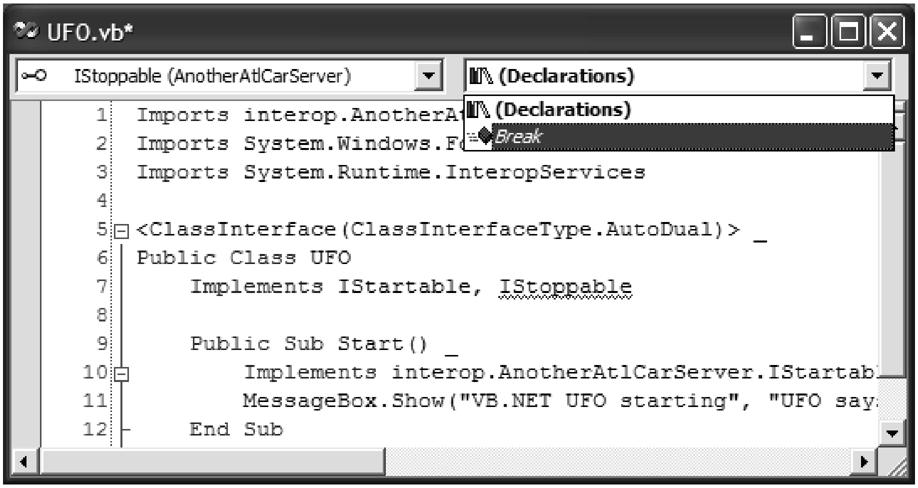

Figure 12-3. The VB .NET IDE Implement Interface Wizard

Here is the complete VB .NET definition of UFO, which also makes use of an AutoDual class interface (again, be sure to assign a strong name to the assembly and deploy this assembly to the GAC):

```
<ClassInterface(ClassInterfaceType.AutoDual)> _
Public Class UFO
    Implements IStartable, IStoppable
    Public Sub Start() _
        Implements ANOTHERATLCARSERVERLib.IStartable.Start
        MessageBox.Show("VB.NET UFO starting", "UFO says:")
    End Sub
    Public Sub Break() Implements ANOTHERATLCARSERVERLib.IStoppable.Break
        MessageBox.Show("VB.NET UFO stopping", "UFO says:")
    End Sub
End Class
```

CODE *The BinaryCompatibleVbNetTypeServer project is included under the Chapter 12 directory.*

Registering the .NET Assemblies with COM

So, to recap the story thus far, at this point you have three objects (CoTruck, LawnMower, and UFO). Each has been created in a specific language (C++, C#, or VB .NET) using two different architectures (COM and .NET) that implement the same two COM interfaces. Furthermore, the interop assembly for the AnotherAtlCarServer.dll COM server and the strongly named .NET assemblies have been deployed to the GAC. Like any COM-to-.NET interaction, however, you must generate COM type information (and register the contents) for each native .NET assembly using regasm.exe. Thus, from the command line, run regasm.exe against both of your .NET assemblies. For example:

```
regasm BinaryCompatibleVbNetTypeServer.dll /tlb
```

Building a VB 6.0 COM Client

Now that each .NET assembly has been configured to be reachable by a COM client, the final step of this example is to build an application that interacts with each object in a binary-compatible manner. While you are free to use any COM-aware programming language, I'll make use of a VB 6.0 Standard EXE project that interacts with each type. The big picture is illustrated in Figure 12-4.

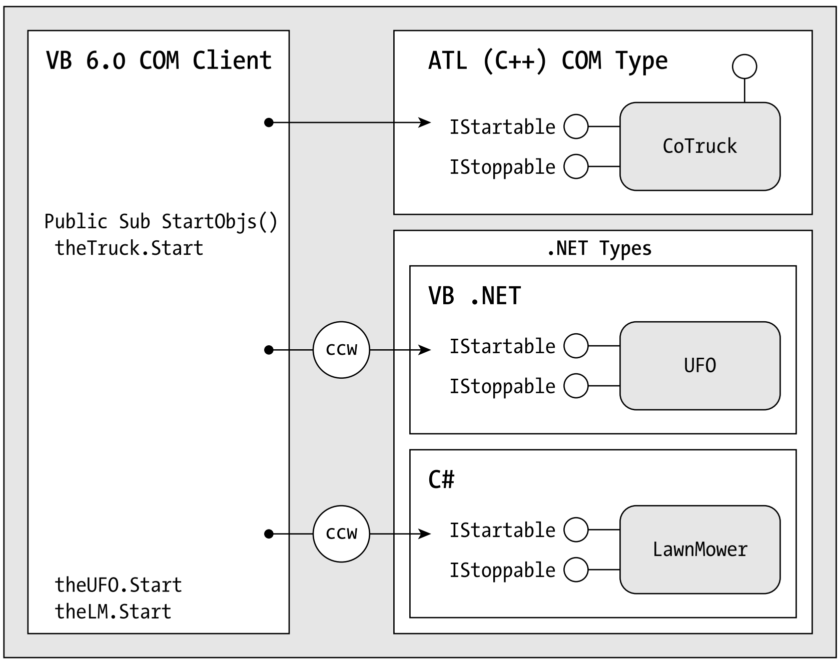

Figure 12-4. Behold, the power of interface-based programming.

The first step (of course) is to set a reference to each type library (Figure 12-5).

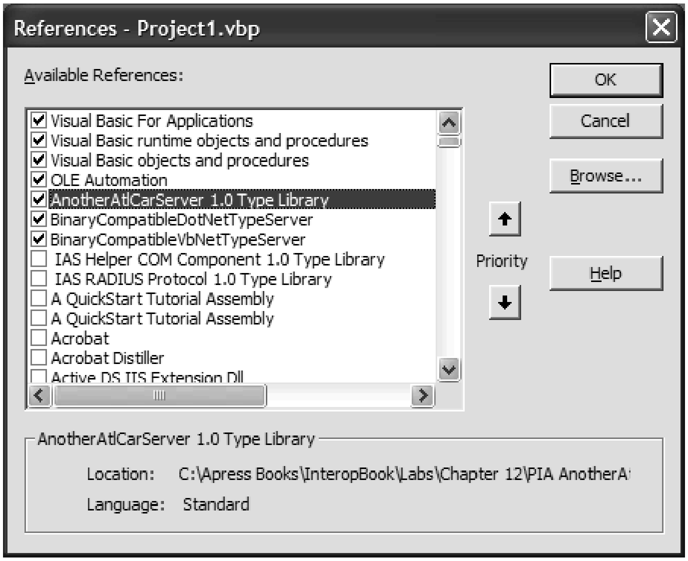

Figure 12-5. Referencing the COM type information

Just to keep things interesting, you will add one additional refinement to the scenario suggested by Figure 12-4. Rather than declaring three Form-level member variables of type UFO, LawnMower, and CoTruck, let's make use of a VB 6.0 Collection type to contain each item (as this will better illustrate the interface-based polymorphism of semantically gluing the types together). Thus, if the main Form has two Button types that start and stop each item in the collection, you are able to author the following VB 6.0 code:

```
Option Explicit
Private theObjs As Collection

' Loop through the collection
' and start everything using IStartable.
Private Sub btnStartObjs_Click()
    Dim temp As IStartable
    Dim i As Integer
    For i = 0 To theObjs.Count - 1
        Set temp = theObjs(i + 1)
        temp.Start
    Next
End Sub
```

```
' Loop through the collection and
' stop everything using IStoppable.
Private Sub btnStopObjs_Click()
    Dim temp As IStoppable
    Dim i As Integer
    For i = 0 To theObjs.Count - 1
        Set temp = theObjs(i + 1)
        temp.Break
    Next
End Sub

' Fill the collection with some
' binary compatible types.
Private Sub Form_Load()
    Set theObjs = New Collection
    theObjs.Add New CoTruck                ' ATL type.
    theObjs.Add New UFO                    ' VB .NET type.
    theObjs.Add New DotNetLawnMower        ' C# type
End Sub
```

Notice that you are able to communicate with each type using the custom COM interfaces defined in the original ATL server (thus the binary compatibility nature of the example). If you were to run the client application, you would see a series of message boxes pop up as the types in the collection were manipulated.

> **CODE** *The Vb6COMCompatibleClient project is included under the Chapter 12 subdirectory.*

Defining COM Interfaces Using Managed Code

Although the previous example did indeed allow the .NET types to implement existing COM interfaces, you had to jump through a few undesirable hoops during the process. First, each .NET code library was required to obtain the type information of IStoppable and IStartable via tlbimp.exe. This of course results in an [.assembly extern] listing in each assembly manifest. Given this, each .NET assembly now depends on the presence of the interop assembly on the target machine. If the interop assembly is not present and accounted for, the .NET consumer is unable to find the correct metadata and it becomes woefully binary-*incompatible* with other like-minded COM types.

When you think about it, the C# LawnMower and VB .NET UFO types never needed to directly interact with the CoTruck. All these projects required were the managed definitions of the raw COM interfaces. To simplify the process, you could have defined IStartable and IStoppable (using managed code) directly within the .NET assemblies. In this way, your .NET assemblies are no longer tied to an interop assembly and are still binary compatible!

To illustrate, let's see a simple example. Assume you have yet another C# Code Library (ManagedComDefs) that contains a simple class named DvdPlayer. Given that DVD players are also startable and stoppable, our goal is to achieve binary compatibility with the CoTruck, UFO, and LawnMower types, *without* referencing the interop.AnotherAltCarServer.dll assembly.

When you define COM interfaces directly within managed code, each and every interface must be attributed with the ComImportAttribute, GuidAttribute, and InterfaceTypeAttribute types. Therefore, all your managed interfaces look something like the following:

```
// Some binary compatible COM interface
// defined in managed code.
[ComImport, Guid("<IID>"),
InterfaceType(ComInterfaceType.<type of COM interface>)]
public interface SomeBinaryCompatibleInterface
{ // Members…}
```

The ComImportAttribute type is simply used to identify this type as a COM entity when exposed to a COM client. Obviously, the value of the GuidAttribute type must be identical to the original IDL IID. As for the InterfaceTypeAttribute, you are provided with the following related enumeration to mark the representation of the COM interface you are describing:

```
public enum System.Runtime.InteropServices.ComInterfaceType
{
    InterfaceIsDual,
    InterfaceIsIDispatch,
    InterfaceIsIUnknown
}
```

The ComInterfaceType value passed into the InterfaceTypeAttribute is used by the .NET runtime to determine how to build the correct vtable for the unmanaged COM interface (more on this tidbit in just a moment). Recall that the IStartable and IStoppable interfaces were defined in IDL as follows:

```
[object,
uuid(7FE41805-124B-44AE-BEAE-C3491E35372B),
oleautomation,
helpstring("IStartable Interface"),
pointer_default(unique)]
interface IStartable : IUnknown
{ HRESULT Start(); };
```

```
[object,
uuid(B001A308-8D66-4d23-84A4-B67615646ABB),
oleautomation,
helpstring("IStartable Interface"),
pointer_default(unique)]
interface IStoppable : IUnknown
{ HRESULT Break();};
```

Looking at these interface types, it should be clear that the COM-to-.NET data type, type, and type member conversion rules still apply (for example, System.String becomes BSTR and whatnot). In this case, you are happy to find that Start() and Break() take no parameters, and therefore can be defined in terms of C# in a rather straightforward manner. Here is the complete code behind the binary-compatible DvdPlayer:

```
using System;
using System.Runtime.InteropServices;
using System.Windows.Forms;

namespace ManuallyInterfaceDefsServer
{
    // Managed definition of IStartable.
    [ComImport,
    Guid("7FE41805-124B-44AE-BEAE-C3491E35372B"),
    InterfaceType(ComInterfaceType.InterfaceIsIUnknown)]
    interface IStartable { void Start(); };

    // Managed definition of IStoppable.
    [ComImport,
    Guid("B001A308-8D66-4d23-84A4-B67615646ABB"),
    InterfaceType(ComInterfaceType.InterfaceIsIUnknown)]
    interface IStoppable { void Break();};

    // A binary compatible DVD player!
    [ClassInterface(ClassInterfaceType.AutoDual)]
    public class DvdPlayer : IStartable, IStoppable
    {
        public DvdPlayer(){}
        public void Start()
        { MessageBox.Show("Staring movie...", "DvdPlayer");}

        public void Break()
        {MessageBox.Show("Stopping movie...", "DvdPlayer");}
    }
}
```

Once you compile this .NET assembly, if you (a) deploy this assembly into the GAC and (b) export the metadata to a COM *.tlb file via regasm.exe, you would be able to set a reference to the exported *.tlb file and update the VB 6.0 COM client as follows:

```
' Add a DVD player into the mix.
Private Sub Form_Load()
    Set theObjs = New Collection
    theObjs.Add New CoTruck
    theObjs.Add New UFO
    theObjs.Add New DotNetLawnMower
    theObjs.Add New DvdPlayer
End Sub
```

Sure enough, you are able to make use of IStartable and IStoppable of the DvdPlayer as expected (Figure 12-6).

Figure 12-6. Using the binary-compatible DvdPlayer

CODE *The ManagedComDefs project is included under the Chapter 12 subdirectory.*

Selected Notes on Manually Defining COM Interfaces Using Managed Code

The previous example was quite straightforward, given that the interfaces you defined were IUnknown-derived entities (thus no DISPIDs) and contained methods with no parameters (thus no [in], [out], or [out, retval] attributes to worry about). As you might expect, if you attempt to manually pound out the details of more complex COM interfaces, you need to apply additional .NET attributes. Furthermore, it is possible (although not altogether likely) that you might need to define other COM types (enums, structures, coclasses) in terms of managed code. To be sure, if the COM type you are attempting to become binary-compatible with has been defined in terms of COM IDL, you will never need to manually define COM types other than the occasional interface. Even then, if the dependency on a related interop assembly is acceptable, you will not need to bother to do this much.

However, there may be some (hopefully) rare cases in which you will need to manually define COM interfaces via managed code. For example, in C++, it is possible to build a COM class supporting a set of COM interfaces without the use of IDL. Given that the midl.exe compiler simply regards IDL interfaces as a collection of C++ pure virtual functions, a C++ developer could choose to define the pure virtual functions directly in terms of C++. The obvious downfall to this approach is that the programmer has effectively created a COM server that can only be used by other C++ clients. If a .NET programmer wished to build a binary-compatible type using an interface described in raw C++, it would demand creating a managed definition of the COM type, given that the COM type library (and thus the interop assembly) doesn't exist!

The process of manually defining a COM type in terms of managed code can be very helpful if you require only a subset of items defined in the type library, or if you need to somehow modify the COM type to work better from a managed environment. As you may recall from Chapter 9, it is possible to crack open an interop assembly and tweak the internal metadata. The same result can often be achieved by directly implementing the COM types using managed code (not to mention, it can be achieved in a much simpler manner). Given these possibilities, let's walk through an extended example.

Manually Defining COM Atoms: An Extended Example

The next COM server you examine (AtlShapesServer) defines a coclass (CoHexagon) that supports a single [dual] interface (IDrawable). IDrawable defines a small set of methods, one of which makes use of a custom COM enumeration. Here is the complete IDL:

```
typedef enum SHAPECOLOR
{
    RED, PINK, RUST
}SHAPECOLOR;

[object,
uuid(B1691C03-7EA8-4DAB-86CC-7D6CD859671A),
dual,
pointer_default(unique)]
interface IDrawable : IDispatch
{
    [id(1), helpstring("method Draw")]
    HRESULT Draw([in] int top, [in] int left, [in] int bottom, [in] int right);
```

```
    [id(2), helpstring("method SetColor")]
    HRESULT SetColor([in] SHAPECOLOR c);
};

[uuid(95FBF6E3-1B03-4904-A5D3-C77A02785F9A),
version(1.0)]
library ATLSHAPESSERVERLib
{
    importlib("stdole32.tlb");
    importlib("stdole2.tlb");
    [uuid(204F9A4B-4D22-451B-BE2F-338F2917E7F5)]
    coclass CoHexagon
    {
         [default] interface IDrawable;
    };
};
```

Defining the Dual Interface (and SHAPECOLOR Enum) Using C#

When you describe a [dual] interface in terms of managed code, you obviously need to supply ComInterfaceType.InterfaceIsDual to the InterfaceTypeAttribute constructor (given the IDL definition). Additionally, you are required to supply the correct DISPID values for each member. This alone is not too earth-shattering. However, recall that the IDrawable interface defines two members:

```
interface IDrawable : IDispatch
{
    [id(1)]
    HRESULT Draw([in] int top, [in] int left, [in] int bottom, [in] int right );
    [id(2)] HRESULT SetColor([in] SHAPECOLOR c);
};
```

Now, as you are aware, COM interfaces are used to construct a vtable for the implementing coclass. A vtable is little more than a listing of addresses that point to the correct function implementation. Given that COM is so dependent on a valid vtable, you must understand that it is *critical* that you define the methods of a managed COM interface in the same order as found in the original IDL (or C++ header file). If you do not, you are most certainly not binary-compatible. Given this, here is the definition of IDrawable (and the related SHAPECOLOR enum) in terms of C#:

```
// Defining COM enums in managed
// code is painless.
public enum SHAPECOLOR
{ RED, PINK, RUST };
```

```
// The managed version of IDrawable.
[ComImport,
Guid("B1691C03-7EA8-4DAB-86CC-7D6CD859671A"),
InterfaceType(ComInterfaceType.InterfaceIsDual)]
interface IDrawable
{
    [DispId(1)]
    void Draw([In] int top, [In] int left,
              [In] int bottom, [In] int right);
    [DispId(2)]
    void SetColor([In] SHAPECOLOR c);
};
```

Here, you are making use of the DispIdAttribute type to define the DISPIDs of each interface. As you are most likely able to figure out, it is critical that the values supplied to each DispIdAttribute match the values of the original COM IDL. If you build a .NET type that is binary compatible with the IDrawable interface, you might author the following:

```
[ClassInterface(ClassInterfaceType.AutoDual)]
public class Circle: IDrawable
{
    public Circle(){}
    public void Draw(int top, int left, int bottom, int right)
    {
        MessageBox.Show(String.Format("Top:{0} Left:{1} Bottom:{2} Right{3}",
            top, left, bottom, right));
    }

    public void SetColor(SHAPECOLOR c)
    {
        MessageBox.Show(String.Format("Shape color is {0}", c.ToString()));
    }
}
```

If you view the .NET metadata descriptions of the IDrawable interface using ILDasm.exe, you find that the ComImportAttribute type is not listed directly with the GuidAttribute and InterfaceType values. The essence of the ComImport attribute is cataloged, however, using the [import] tag on the interface definition:

```
.class interface private abstract auto ansi import IDrawable
{
…
} // end of class IDrawable
```

Assuming you have processed this .NET assembly using regasm.exe, you would now be able to build an unmanaged COM client that interacts with the ATL CoHexagon and C# Circle type in a binary-compatible manner (using either early or late binding).

So, to wrap up the topic of building binary-compatible .NET types, understand that just because you can define COM interfaces in managed code does not mean you have to. Typically speaking, you simply set a reference to the correct interop assembly. However, if you are building a managed application that needs to communicate to a COM class using an interface for which there is no interop assembly, it is often necessary to manually define the type in terms of managed code (recall, for example, your C# COM type library viewer in Chapter 4).

Interacting with Interop Assembly Registration

As you recall from Chapter 2, a COM in-process server defines two function exports that are called by various installation utilities (regsvr32.exe) to register or unregister the necessary registry entries. As well, when a .NET assembly is to be used by COM, the system registry must be updated using regasm.exe to effectively fool the COM runtime. As you have seen, regasm.exe catalogs the correct entries automatically. What happens, however, if you want to insert custom bits of information into the registry during the default process performed by regasm.exe?

The System.Runtime.InteropServices namespace defines two attributes for this very reason. To illustrate, assume you have a new C# code library (CustomRegAsm) that defines some number of types. When you want to allow regasm.exe to trigger a custom method during the registration process, simply define a static (or Shared in VB .NET) method that is adorned with the ComRegisterFunctionAttribute. Likewise, if you wish to provide a hook for the unregistration process, define a second static member that supports the ComUnregisterFunctionAttribute. For example:

```
public class SomeClass
{
    public SomeClass(){}

    // This method will be called when
    // regasm.exe is run against this assembly.
    [ComRegisterFunction()]
    private static void CustomReg(Type t)
    {
        MessageBox.Show(String.Format("Registering {0}",
            t.ToString()));
    }
```

```
    // This method will be called when
    // regasm.exe is run against this
    // assembly using the /u flag.
    [ComUnregisterFunction()]
    private static void CustomUnReg(Type t)
    {
        MessageBox.Show(String.Format("Registering {0}",
             t.ToString()));
    }
}
```

As you can see, the target methods must provide a single argument of type System.Type, which represents the current type in the assembly being registered for use by COM. As you might guess, regasm.exe passes in this parameter automatically.

Inserting Custom Registration Information

So, when might you need to interact with the assembly's registration process? Assume that you wish to record the date and time on which a given .NET assembly has been registered on a given user's machine. To do this, you can make use of the Microsoft.Win32 namespace, which contains a small number of types that allow you to programmatically read from and write to the system registry. For example, the CustomReg() and CustomUnReg() methods could be retrofitted as follows:

```
[ComRegisterFunction()]
private static void CustomReg(Type t)
{
    RegistryKey k =
        Registry.CurrentUser.CreateSubKey(@"Software\Intertech\CustomRegAsm");
    k.SetValue("InstallTime", DateTime.Now.ToShortTimeString());
    k.SetValue("InstallDate", DateTime.Now.ToShortDateString());
    k.Close();
}

[ComUnregisterFunction()]
private static void CustomUnReg(Type t)
{
    Registry.CurrentUser.DeleteSubKey(@"Software\Intertech\CustomRegAsm");
}
```

When you register this .NET assembly via regasm.exe,
you find the following information inserted under
HKEY_CURRENT_USER\Software\Intertech\CustomRegAsm (Figure 12-7).

Figure 12-7. Getting involved with assembly registration

If you specify the /u flag, the information is correctly removed from the same subkey.

CODE *The CustomRegAsm project is included under the Chapter 12 subdirectory.*

Programmatically Converting Assemblies to COM Type Information

Recall from Chapter 9 that the System.Runtime.InteropServices.TypeLibConverter type allows you to programmatically convert COM *.tlb files into .NET interop assemblies. As mentioned at that time, this same class provides the ability to convert .NET assemblies into COM type information programmatically. Given this, let's examine the process of building a customized version of the tlbexp.exe command line utility (which as you will see looks much like the customized tlbimp.exe utility).

To begin, assume that you have a new C# console application named MyTypeLibExporter. The goal here is to allow the user to enter the path to a given .NET assembly and, using TypeLibConverter, to build a corresponding COM type library. The application's Main() method prompts for the assembly to export and passes this string into a static helper function named GenerateTLBFromAsm().

Once the *.tlb file has been generated (and stored in the application directory), the user is again prompted to determine if the .NET assembly should be registered for use by COM. If the user wishes to do so, make use of the System.Runtime.InteropServices.RegistrationServices type. Here then, is the complete implementation behind Main():

```
static void Main(string[] args)
{
    // Get the path to the assembly.
    Console.WriteLine("Please enter the path to the .NET binary");
    Console.WriteLine(@"Example: C:\MyStuff\Blah\myDotNetServer.dll");
    Console.Write("Path: ");
    string pathToAssembly = Console.ReadLine();

    // Generate type lib for this assembly.
    UCOMITypeLib i = GenerateTLBFromAsm(pathToAssembly);

    // Ask if user wants to register this server with COM.
    int regValue;
    Console.WriteLine("Would you like to register this .NET library with COM?");
    Console.Write("1 = yes or 0 = no ");
    regValue = Console.Read();

    if(regValue == 1)
    {
        RegistrationServices rs = new RegistrationServices();
        Assembly asm = Assembly.LoadFrom(pathToAssembly);
        rs.RegisterAssembly(asm, AssemblyRegistrationFlags.None);
        Console.WriteLine(".NET assembly registered with COM!");
    }
}
```

As you can see, the real workhorse of this application is the GenerateTLBFromAsm() helper function. Like the custom tlbimp.exe application you created earlier in this text, the TypeLibConverter.ConvertAssemblyToTypeLib() method requires you to pass in an instance of a class that will be called by the TypeLibConverter type to resolve references to additional assemblies as well as general reporting information. In this case, however, the class type is required to adhere to the behavior defined by ITypeLibExporterNotifySink:

```
public interface ITypeLibExporterNotifySink
{
    void ReportEvent(ExporterEventKind eventKind,
        int eventCode, string eventMsg);
    object ResolveRef(System.Reflection.Assembly assembly);
}
```

Much like the ITypeLibImporterNotifySink interface seen in Chapter 9, the implementation of ITypeLibExporterNotifySink delegates the work of resolving the referenced assembly to the static MyTypeLibExporter.GenerateTLBFromAsm() helper function:

```
// The callback object.
internal class ExporterNotiferSink : ITypeLibExporterNotifySink
{
    public void ReportEvent(ExporterEventKind eventKind,
        int eventCode, string eventMsg)
    {
         Console.WriteLine("Event reported: {0}", eventMsg);
    }

    public object ResolveRef(System.Reflection.Assembly assembly)
    {
        // If the assembly we are converting references another assembly,
        // we need to generate a *tlb for it as well.
        string pathToAsm;
        Console.WriteLine("MyTypeLibExporter encountered an assembly");
        Console.WriteLine("which referenced another assembly...");
        Console.WriteLine("Please enter the location to {0}", assembly.FullName);
        pathToAsm = Console.ReadLine();
        return MyTypeLibExporter.GenerateTLBFromAsm(pathToAsm);
    }
}
```

Before you see the details behind MyTypeLibExporter.GenerateTLBFromAsm(), you need to define some low-level COM types in terms of managed code. As you may recall from Chapter 4, when you create a custom COM type library generation tool, you need to call ICreateTypeLib.SaveAllChanges() to commit the type information to file. The trouble, however, is that the System.Runtime.InteropServices namespace does not define a managed equivalent of this method. Thus, using the tricks presented in this chapter, here is a makeshift version. It is makeshift in that I am representing the ICreateTypeInfo interface returned from the CreateTypeInfo() method (also recall from Chapter 4 that the ICreateTypeInfo interface is *huge*).

```
[ComImport,
GuidAttribute("00020406-0000-0000-C000-000000000046"),
InterfaceTypeAttribute(ComInterfaceType.InterfaceIsIUnknown),
ComVisible(false)]
internal interface UCOMICreateTypeLib
{
    // IntPtr is a hack to avoid having
    // to define ICreateTypeInfo (which is HUGE).
    IntPtr CreateTypeInfo(string name, TYPEKIND kind);
    void SetName(string name);
    void SetVersion(short major, short minor);
    void SetGuid(ref Guid theGuid);
    void SetDocString(string doc);
```

```
    void SetHelpFileName(string helpFile);
    void SetHelpContext(int helpCtx);
    void SetLcid(int lcid);
    void SetLibFlags(uint flags);
    void SaveAllChanges();
}
```

Now that you have a managed definition for use by the GenerateTLBFromAsm() method, you can flesh out the details as follows:

```
public static UCOMITypeLib GenerateTLBFromAsm(string pathToAssmebly)
{
    UCOMITypeLib managedITypeLib = null;
    ExporterNotiferSink sink = new ExporterNotiferSink();

    // Load the assembly to convert.
    Assembly asm = Assembly.LoadFrom(pathToAssmebly);
    if (asm != null)
    {
        try
        {
            // Create name of type library based on .NET assembly.
            string tlbname = asm.GetName().Name + ".tlb";

            // Convert the assembly.
            ITypeLibConverter TLBConv = new TypeLibConverter();
            managedITypeLib = (UCOMITypeLib)
                TLBConv.ConvertAssemblyToTypeLib(asm, tlbname, 0, sink);

            // Save the type library to file.
            try
            {
                UCOMICreateTypeLib managedICreateITypeLib =
                    (UCOMICreateTypeLib)managedITypeLib;
                managedICreateITypeLib.SaveAllChanges();
            }
            catch (COMException e)
            {
                throw new Exception("Error saving the type lib : "
                    + e.ErrorCode.ToString("x"));
            }
        }
        catch (Exception e)
        {
            throw new Exception("Error Converting assembly" + e);
        }
    }
    return managedITypeLib;
}
```

I'd bet the details of this method are not too shocking by this point in the text. Basically, you load the assembly based on the incoming string parameter and define a name for the type library you are creating using the assembly's name as a base. Once you have an Assembly reference, you call ConvertAssemblyToTypeLib() and specify the reference to the loaded assembly, the name of the type library to create, any additional flags (or in our case, a lack thereof), and an instance of the sink implementing ITypeLibExporterNotifySink.

The System.Object that is returned from ConvertAssemblyToTypeLib() actually represents a reference to the in-memory representation of the COM type information, which is to say, an UCOMITypeLib interface. Once you cast this type into your version of the unmanaged ICreateTypeLib type, you are able to call SaveAllChanges() to commit the information to file.

Do note that your GenerateTLBFromAsm() helper function returns the UCOMITypeLib interface to the caller. You really don't need to do so. Using this type, however, you could interact with the internal COM types defined by this type library (as illustrated in Chapter 4). In any case, this wraps up the implementation of your custom tlbexp.exe utility. Figure 12-8 shows a test drive by importing the CSharpCarLibrary.dll assembly created in Chapter 6.

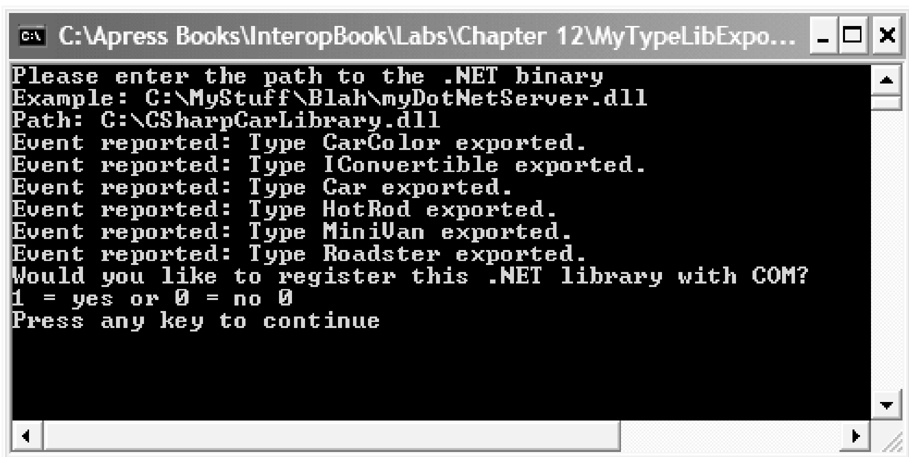

Figure 12-8. Exporting CSharpCarLibrary.dll

If you opened the generated *.tlb file using oleview.exe, you would find the COM definitions for each .NET type (Figure 12-9).

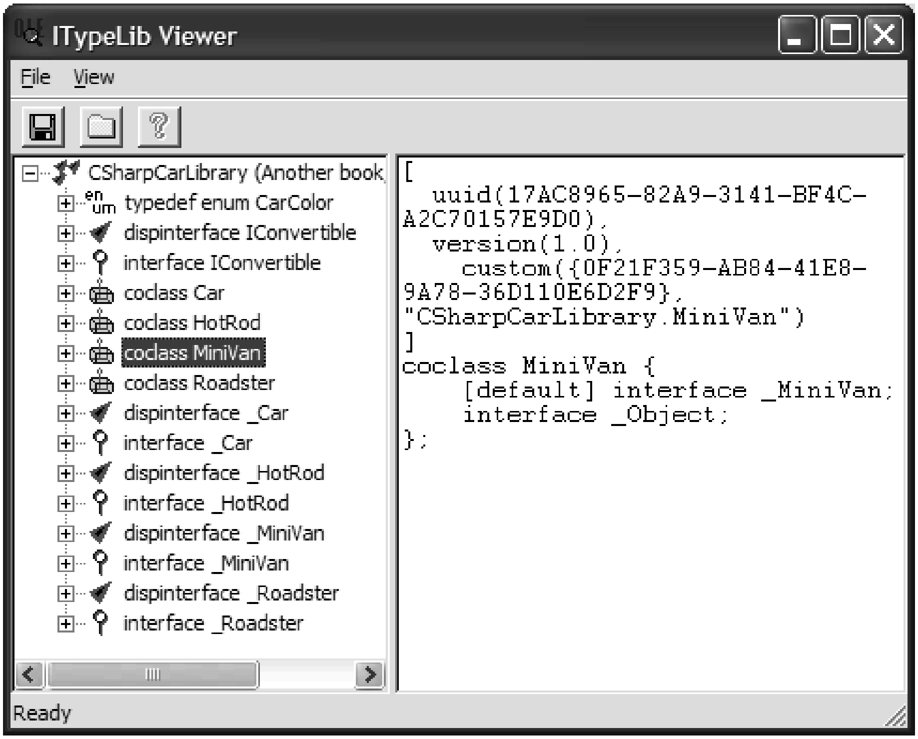

*Figure 12-9. The exported *.tlb file*

> **CODE** *The MyTypeLibExporter project is included under the Chapter 12 subdirectory.*

Hosting the .NET Runtime from an Unmanaged Environment

The final topic of this chapter is a rather intriguing one: building a custom host for the .NET runtime (aka the CLR). Like all things under the .NET platform, the runtime engine is accessible using a set of managed types. In this case, the assembly in question is mscoree.dll (where "ee" stands for execution engine). It may surprise you to know that when you install the .NET platform, you receive a corresponding *.tlb file for mscoree.dll (mscoree.tlb) that has been properly configured in the system registry.

Because the content of mscoree.dll has been expressed in terms of COM metadata, it is possible to build a custom host using any COM-aware programming language (within the realm of the language's limitations). Do understand that regardless of which COM language you choose, when you make use of mscoree.tlb, you are also required to reference the related mscorlib.tlb file. For the example that follows, assume that you have created a new Standard EXE application using VB 6.0. This assumption aside, set a reference to each *.tlb file using the IDE's Project | References menu option (see Figure 12-10).

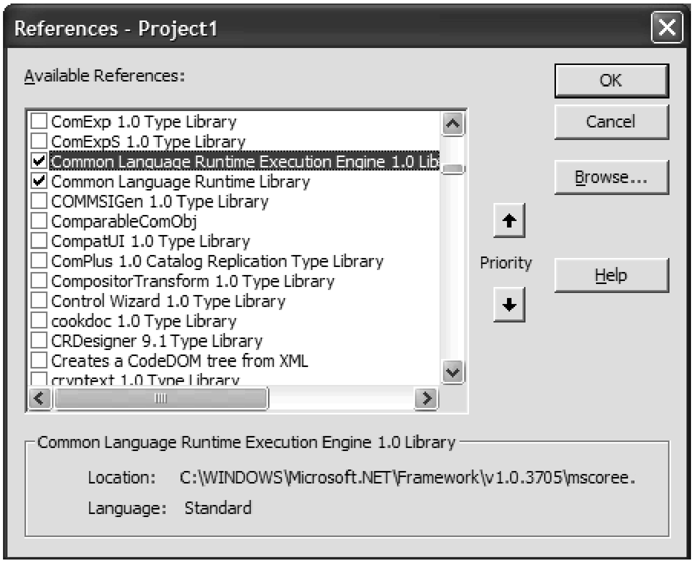

Figure 12-10. Referencing mscoree.tlb/mscorelib.tlb

Various chapters of this text have already examined some types contained within mscorlib.tlb, but what of mscoree.tlb? Like any loaded type library, the VB 6.0 Object Browser allows you to view the contained types. As you can see from Figure 12-11, despite the exotic nature of this exported assembly, mscoree.tlb defines a surprisingly small number of items.

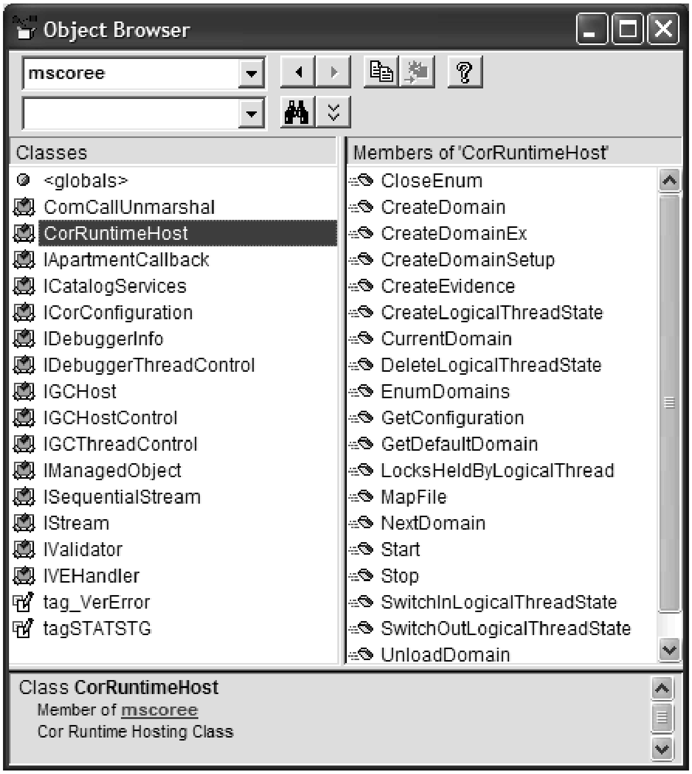

Figure 12-11. The crux of the CLR in terms of COM type information

A full treatment of each and every type defined in mscoree.dll is beyond the scope of this text. Luckily, you are able to build a custom CLR host using a single type: CorRuntimeHost. This single .NET class type implements a set of interfaces (also defined within mscoree.tlb) that provide the following functionality:

- The ability to load and unload .NET application domains
- The ability to manipulate the .NET garbage collector
- The ability to validate code within a given .NET assembly
- The ability to interact with a given debugger attached to the current process

So, given that mscoree.tlb defines the types you need to build a custom CLR host, the next logical question is when you might want to do this. Besides the fact that building a custom host is extremely interesting in its own right, there is a practical reason to do so. When you build a custom host from unmanaged code, you are able to dynamically load .NET assemblies for use by COM, without having to register the assembly using regasm.exe.

Building a Custom Host

The first detail of your VB 6.0 host is to establish a valid application domain to host the loaded assemblies. As you may know, under the .NET platform an application domain is a unit of isolated execution within a Win32 process (similar in function to the apartment architecture of classic COM). Just as a process may contain numerous application domains, a given application domain may contain numerous .NET assemblies. You are able to represent a given application domain using the System.AppDomain type.

Given this, the Form_Load() event handler creates an instance of CorRuntimeHost. Once the host has started, obtain a valid AppDomain via CorRuntimeHost.GetDefaultDomain(). The Form_Unload() event handler shuts down the CLR via the aptly named CorRuntimeHost.Stop(). Here is the story thus far:

```
' The types we need to host the CLR.
Private myAppDomain As AppDomain
Private myCLRHost As CorRuntimeHost

' Load the CLR and set app domain.
Private Sub Form_Load()
    Set myCLRHost = New CorRuntimeHost
    myCLRHost.Start
    myCLRHost.GetDefaultDomain myAppDomain
End Sub

' Unload the CLR.
Private Sub Form_Unload(Cancel As Integer)
    myCLRHost.Stop
End Sub
```

Now assume that the main Form has three VB 6.0 Button types. The Click event handler of the first button (btnListLoadedAsms_Click()) obtains and displays the list of each assembly currently hosted by the default application domain. To do this, you are able to obtain an array of Assembly types from the GetAssemblies() method of the AppDomain type. To display the name of each assembly, you are able to simply make use of the Assembly.FullName property:

```
' List all the loaded assemblies.
Private Sub btnListLoadedAsms_Click()
    Dim loadedAsms() As Assembly
    loadedAsms = myAppDomain.GetAssemblies()
```

```
    Dim theAsms As String
    Dim i As Integer
    For i = 0 To UBound(loadedAsms)
        theAsms = theAsms + loadedAsms(i).FullName + vbLf
    Next
    MsgBox theAsms
End Sub
```

The next Button type is responsible for loading the System.Collections.dll assembly from the GAC to exercise the ArrayList type. Note how the CreateInstance() method requires you to send in (a) the friendly name of the assembly containing the type and (b) the fully qualified name of the type itself. What is returned from AppDomain.CreateInstance() is an ObjectHandle type, which provides the ability to obtain the underlying type using the Unwrap() method:

```
' Load a type from the GAC.
Private Sub btnLoadFromGAC_Click()
    Dim arLst As ArrayList
    Dim obj As ObjectHandle
    Set obj = myAppDomain.CreateInstance("mscorlib",
        "System.Collections.ArrayList")
    Set arLst = obj.Unwrap

    arLst.Add "Hello there!"
    arLst.Add 12
    arLst.Add True

    Dim items As String
    items = items + arLst(0) + vbLf
    items = items + CStr(arLst(1)) + vbLf
    items = items + CStr(arLst(2)) + vbLf

    MsgBox items
End Sub
```

If you run the application at this point, once you load System.Collection.dll, you find the message displayed in Figure 12-12.

Figure 12-12. Interacting with System.Collections.dll

The final button of your VB 6.0 Form type is responsible for loading a private, and unregistered, .NET assembly. To ensure that this example illustrates the point of loading unregistered .NET binaries, assume you have the following trivial C# class definition, defined in an assembly named (of course) UnregisteredAssembly:

```
using System;
using System.Runtime.InteropServices;

namespace UnregisteredAssembly
{
    [ClassInterface(ClassInterfaceType.AutoDual)]
    public class AnotherAdder
    {
        public AnotherAdder(){}
        public int Add(int x, int y)
        { return x + y;}
    }
}
```

Now, although you do not need to register this assembly, you still need to generate type information for your VB 6.0 client. Thus, run tlbexp.exe against this binary, and place the *.tlb and UnregisteredAssembly.dll files in the same directory as the current VB 6.0 project (Figure 12-13).

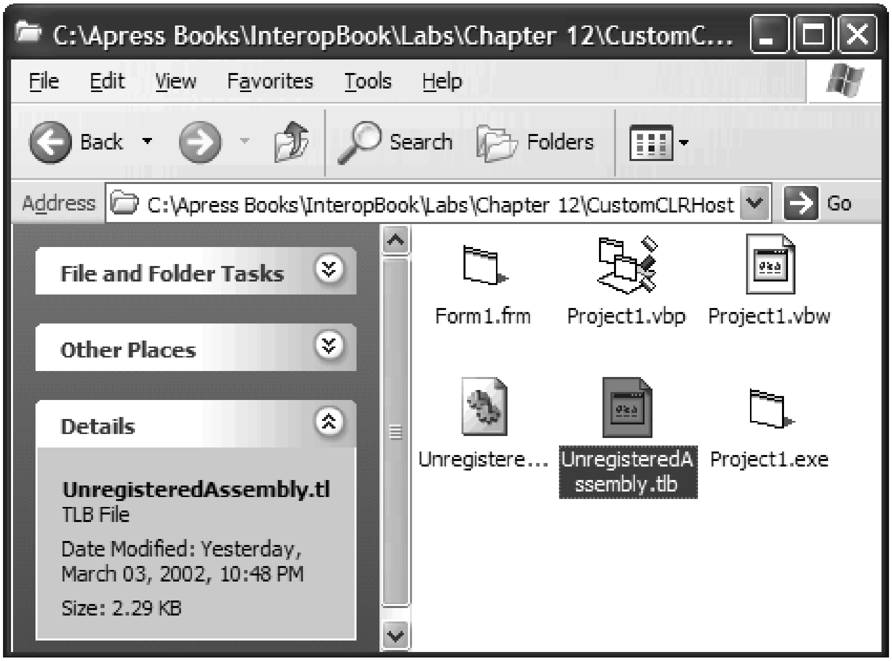

*Figure 12-13. Configuring the unregistered assembly and related *.tlb file*

Now that you have a private assembly, you are able to write the following event handler for the Form's final Button type:

```
' NOTE!!!  Because VB projects do not directly
' run from the application directory within
' the IDE, you will
' need to run the EXE to use this function.
Private Sub btnLoadFromPrivateAsm_Click()
    Dim adder As AnotherAdder
    Dim obj As ObjectHandle
    Set obj = myAppDomain.CreateInstance("UnregisteredAssembly", _
        "UnregisteredAssembly.AnotherAdder")
    Set adder = obj.Unwrap
    MsgBox adder.Add(99, 3)
End Sub
```

As you can gather from the lengthy code comment, before you can test this final bit of functionality, you need to build the VB 6.0 application (File | Make) and run the application outside the VB IDE. Once you have built the EXE, simply double-click the executable file. If you loaded UnmanagedAssembly.dll and System.Collections.dll via the correct Button types, you would now find the results shown in Figure 12-14 when you click on the "list all loaded assemblies" Button type.

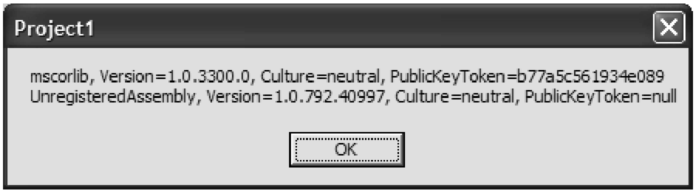

Figure 12-14. Documenting loaded assemblies

With your custom host complete, you come to the end of Chapter 12. As illustrated by this example, when you build a custom host for the CLR, you are able to avoid the process of registering .NET assemblies prior to building COM clients that consume them. If you want to dive into further details of the functionality of mscoree.dll, be sure to check out the tool-builders documents included with the .NET SDK (installed by default under C:\Program Files\Microsoft Visual Studio .NET\FrameworkSDK\Tool Developers Guide\docs).

CODE *The UnmanagedAssembly and CustomCLRHost projects are located in the Chapter 12 subdirectory.*

Summary

The chapter wraps up your investigation of COM-to-.NET interoperability issues. As you have seen, just as a COM type can implement .NET interfaces to achieve type compatibility, a .NET type can implement COM interfaces to achieve binary compatibility with related coclasses. Using managed code, you are able to build managed representations of COM types to avoid creating a dependency with a related interop assembly.

Another key aspect of this chapter illustrated how you are able to build a customized version of tlbexp.exe. While you may never be in the position of needing to do so, this should solidify your understanding of what this tool does on your behalf. The final major topic presented here illustrated how you can interact with the CLR via mscoree.tlb to build a custom host from unmanaged code.

At this point in the text, you have drilled quite deeply into the COM and .NET type systems, and you have seen numerous aspects of the interoperability layer. Before I wrap things up, the next (and final) chapter addresses the topic of building COM+ types (i.e., configured components) using managed code.

Index

B

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W