

# Micro Focus Net Express® and Microsoft.NET

*Using Net Express today with Microsoft.NET*

**WHITE PAPER**





---

## Abstract

According to Microsoft, .NET is a "revolutionary new platform, built on open Internet protocols and standards, with tools and services that meld computing and communications in new ways". When talking about developing applications for .NET, the focus has been on the products that support the Common Language Runtime and Microsoft Intermediate Language. However, Microsoft.NET provides excellent support for interoperability with existing code. This paper describes how Micro Focus Net Express can be used today with Microsoft.NET technologies to maximize your investment in COBOL while developing applications that will either be deployed as Web Services or as new applications that target the Windows platform.



## Table of Contents

<b>Introduction.....</b>	<b>1</b>
Web Services .....	1
The .NET Framework .....	1
MERANT's Position on the .NET Framework.....	2
What's Possible with Net Express today? .....	3
Example Provided with the White Paper .....	4
<b>Using COBOL Components from Managed Code.....</b>	<b>6</b>
Overview of creating a COBOL COM Component .....	6
Making a COM Component Usable by Managed Code .....	7
Example .....	7
Creating the COM Component .....	7
Creating the Metadata for the Account Component.....	14
Using the Account Component from Managed Code .....	15
Raising Exceptions.....	16
<b>Accessing Managed Code from COBOL .....</b>	<b>19</b>
Making Managed Code Visible to COBOL Applications .....	19
Example. ....	19
Handling Exceptions.....	22
Visibility of Managed Types .....	23
<b>Creating a Web Service in COBOL .....</b>	<b>26</b>
Web Services Description Language .....	26
Web Services Meta Language.....	26
Creating WSDL and WSML Files .....	27
Example .....	27
<b>Conclusion .....</b>	<b>30</b>



---

## Introduction

There has been a lot of focus recently on the .NET initiative from Microsoft. However, coming up with a simple description of .NET is difficult as it covers a number of different areas and is at the very core of Microsoft's future direction.

According to Microsoft, .NET is a "revolutionary new platform, built on open Internet protocols and standards, with tools and services that meld computing and communications in new ways".

However, it is important to note that the .NET initiative does not just affect the development of applications for the web. It also has a major impact on the development of applications for the Windows platform.

## Web Services

At the heart of .NET is the concept of disparate systems communicating and sharing data seamlessly. This is the goal of Web Services. A Web Service is a unit of application logic accessible using standard Internet protocols. Looking at it another way, this means that Web-supported standards such as HTTP, XML and SOAP are used for transparent communication between applications running on different systems.

Microsoft .NET shifts focus from individual Web sites and devices to constellations of computers, devices, and services that work together to deliver broader, richer solutions. Internet users will have control over how, when, and what information is delivered to them. Businesses will be able to offer that information, plus products and services, on a range of devices including cell phones, handheld computers, and desktop computers.

Microsoft recently released the SOAP Toolkit V2.0. SOAP (Simple Object Access Protocol) is a lightweight and simple XML-based protocol that is designed to exchange structured and typed information on the Web. The purpose of SOAP is to enable powerful automated Web services. One of the most powerful features of the SOAP Toolkit is a server-side component that maps invoked Web service operations to COM object method calls. This means that the functionality inside a COM object, created using COBOL, can easily be made accessible over the Internet.

## The .NET Framework

The .NET Framework is the infrastructure for the overall .NET Platform. It is a new environment for developing and running software applications, featuring ease of development of web-based services, rich standard run-time services available to components written in a variety of programming languages, and inter-language and inter-machine interoperability. In many ways, it has been likened to as big a change to move from programming for Windows today to programming for .NET as it was to move from programming for DOS to programming for Windows.

The .NET Framework includes the Common Language Runtime (CLR) and the Microsoft Intermediate Language (MSIL). The CLR provides features such as cross-language integration, cross-language exception handling, enhanced security, versioning and deployment support, a simplified model for component interaction, and debugging and profiling services. Code developed for the CLR and compiled to MSIL is called *managed* code. Code that does not target the CLR is called *unmanaged* code. This includes all code compiled using compilers available today, including the code produced by Net Express.

To enable managed code and unmanaged code to communicate, Microsoft has provided extensive facilities for interoperability in the .NET Framework. The focus of the interoperability layer is on enabling existing COM components to be callable from managed code and to enable managed code to be visible to unmanaged code as COM objects.

It is important to note that the .NET Framework and Visual Studio.NET are still in beta and changes may occur between now and their final release. All examples in this white paper have been tested using beta one of the .NET Framework and Visual Studio.NET and may need to be changed to work with the final release of .NET.

## **MERANT's Position on the .NET Framework**

In order to take advantage of many of the new features provided in the .NET Framework, you do not need to wait for a COBOL compiler that generates Microsoft Intermediate Language (MSIL). The .NET Framework provides complete interoperability between managed code and unmanaged COM components. Because of this support, Net Express provides a high level of interoperability with the .NET Framework right now, especially in the areas that are most important to COBOL programmers.

An important strength of the MERANT Micro Focus product range is its cross-platform support. MERANT fully believes that the need for cross-platform support overrides any platform-specific architecture. In addition, MERANT intends to ensure that any solution is effective both for new and existing customers, in particular the large number of existing customers who have made significant investments in Micro Focus technology and COBOL over many years.

As a COBOL development environment Net Express provides a key strength of enabling programmers to meet business needs by targeting a wide range of technologies. MERANT offers a .NET Interoperability solution that can be adopted now.

MERANT has not yet made a decision about targeting the Common Language Runtime (CLR) with Net Express or its other COBOL products. In particular, there are two key areas that require more detailed investigation:

- It is not yet clear to MERANT that full support for all aspects of the COBOL language and the additional features provided by the Micro Focus runtime system can be provided by the CLR and MSIL to provide provides full backward compatibility with earlier versions of Micro Focus products
- The performance of code compiled to MSIL needs to be assessed, especially given the lack of underlying support for common COBOL data types such as packed decimal, usage display numeric and numeric-edited fields and non-intel byte ordering of COMP data items which can be very processor intensive.

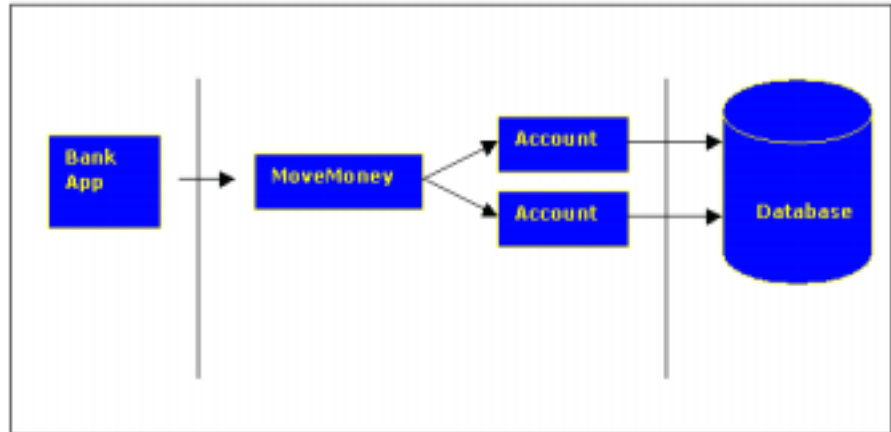
### **What's Possible with Net Express today?**

MERANT first introduced support for COM five years ago in Micro Focus Object COBOL™. Since then, the level of support has steadily increased and, today, Net Express® 3.1 provides more extensive support than any other COBOL compiler for both using COM services from COBOL applications and wrapping COBOL code up as COM objects. This paper introduces how to use the existing support to:

- Create COM components in COBOL and use them from managed code
- Access managed code components from COBOL
- Use the Microsoft SOAP toolkit to turn a COBOL COM component into a Web Service.

## Example Provided with the White Paper

The concepts described here are demonstrated in detail in the example supplied with this white paper. The example is a modified version of a sample provided by Microsoft with the .NET Framework SDK to demonstrate interoperability between COM and .NET. The example also shows how the COBOL components can make use of COM+ transaction services. The original sample can be found in the `samples\interop\comservices` sub-directory of the .NET Framework SDK.



The Bank Sample demonstrates a simple business problem of performing database operations and preserving the integrity of the database through distributed transactions.

The BankApp Client uses the MoveMoney component to transfer money between two accounts and uses two Account components to perform the transfer. The SQL Server database holds the account rows and also keeps track of receipts. The Account components perform the actual work of updating the database table. MoveMoney and Account are marked to require a transaction. When MoveMoney is created, a transaction is started by COM+ Services. When the Account components are created, this transaction is flowed to them, so that when the transfer is completed and MoveMoney returns, the transaction is committed and the database table reflects the transfer.

The components are provided in multiple languages:

- Bank App client is written in Visual Basic 6.0
- MoveMoney is provided in multiple languages: Visual Basic 6.0, Visual Basic.NET, C# and COBOL
- Account is also provided in multiple languages: Visual Basic 6.0, Visual Basic.NET, C# and COBOL

The COBOL version of MoveMoney invokes the C# version of Account. This demonstrates how managed code components can be called from COBOL.

The Visual Basic.NET and C# versions of the MoveMoney component invoke the COBOL version of Account, showing how a COBOL component can be called from managed code.

---

## Using COBOL Components from Managed Code

To be callable from managed code, a component must expose *metadata*. This is simply the term used for the description of the interfaces supported by the component and the types of the parameters used in individual method calls. A similar idea has been available for COM components for some time in the form of type libraries. The .NET Framework SDK includes a tool, *TLBIMP*, which converts the information found in a type library for a COM component into the equivalent definitions in metadata format. Using TLBIMP is the easiest way to make your COM components accessible to managed code. Although it is possible to use a COM component that does not have a type library available, it is considerably more complex and beyond the scope of this white paper. Luckily, if you use the Net Express class and method wizards to create your COM components, the wizards will create and maintain the type library for you.

### Overview of creating a COBOL COM Component

As mentioned earlier, the key to calling COBOL code from managed code is to create a COM component out of your COBOL code. The easiest way to do this is to use the Net Express class and method wizards.

If the terms ‘class’ and ‘method’ make you uneasy, don’t worry. When MERANT introduced the support for creating COM components in COBOL, it decided to adopt the use of the emerging object orientation (OO) extensions to COBOL. When you are working with COM components, everything is described using object terminology, so the use of the OO extensions makes sense. MERANT has used the same technique to introduce support for making COBOL components deployable as Enterprise Java Beans and, in fact, it is possible to develop one set of source code that can be deployed as a COM component or an Enterprise Java Bean.

Even though MERANT uses the OO extensions, you do not have to learn OO programming techniques to create COM components in Cobol. Net Express provides wizards that generate skeleton code for the component—you just insert the code that will perform the functionality required. In most cases, you can use standard procedural Cobol statements.

The key point to note when you create a COM component using the class wizard is that you should select the option to create a type library for the component. This is not the default, so you should ensure that you select the option on the correct screen. This will be seen in the example later in this section.

## Making a COM Component Usable by Managed Code

Once you have created your COM component, you need to create a file containing the .NET metadata for your component. This is done using a utility provided with the .NET Framework SDK called TLBIMP. TLBIMP is run using the following command:

```
TLBIMP <input-file> /out:<output-file>
```

where:

<input-file> is the name of the file containing your type library. In most cases, the type library is built into the dynamic link library (DLL) for your component, so you would specify the name of the dll here.

<output-file> is the name of the file that will contain the metadata. This is built as a dll and the resulting file must be supplied with your component.

## Example

Before looking at specific issues involved in calling COBOL from managed code, it is beneficial to look at a simple example. Here, we will create the Account object in the Bank sample mentioned earlier. This will be developed as a COM component, so we will be using the Net Express Class and Method wizards to create the skeleton of the component for us.

### Creating the COM Component

To create a new class, select **New...** on the Net Express File menu and select **Class** from the dialog box (Figure 1).

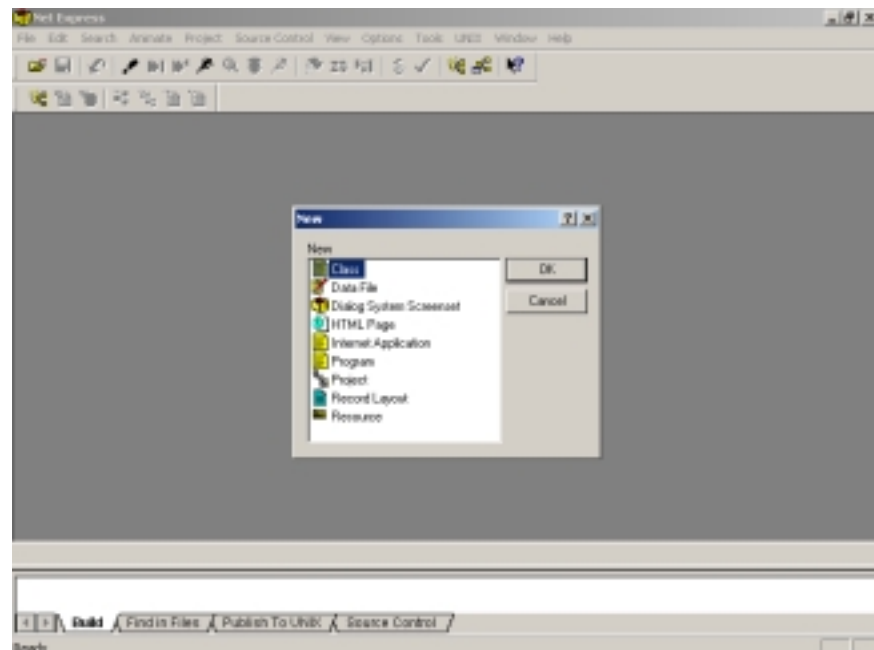


Figure 1 - The New Project Dialog Box

This will bring up the Class Wizard (Figure 2)

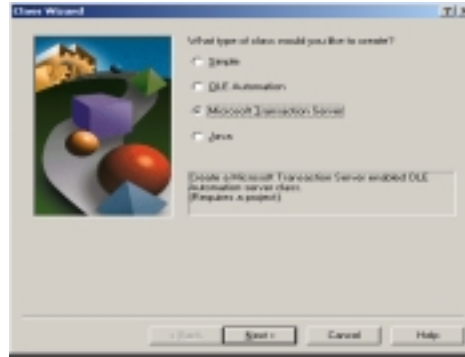


Figure 2 - The Class Wizard Dialog

There are two options on this screen that could be used to create a COM component - **OLE Automation** or **Microsoft Transaction Server**. The skeleton class is similar for each of these. The difference is that if the Microsoft Transaction Server option is selected, additional code is added to retrieve transactional context information and to complete or abort transactions.

---

*Note. The terminology used for these technologies changes rapidly. The text used in these dialog boxes refers to older names for the technologies. However, the code generated is not affected by the changes in terminology.*

---

Because we are creating a transactional component in this example, select the option **Microsoft Transaction Server**. If not creating a transactional component, you would select **OLE Automation**.

If you do not already have a Net Express project open, the Class Wizard will next ask you to provide a name for the project and a location for the new project. In this case, we have called the project **COBOLBank**.

Figure 3 shows the next dialog that prompts for the name of the class and a filename.

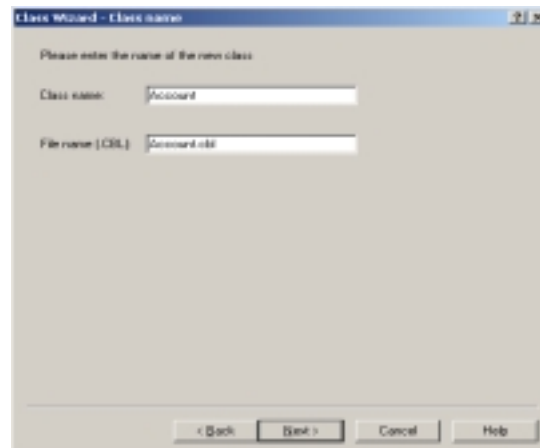


Figure 3 - Providing a name for the Class

The Class name is the name of the component you will be creating. In this case, we are creating an Account component, so we use **Account** as the name of the class. It also makes sense to use **Account.cbl** as the name of the source file.

Figure 4 shows the next dialog that asks you to provide a name for the dll that will be created. The default is to use the same name as the source file. In this case, this is **Account.dll**, so just select **Next** to move to the next dialog in the wizard.

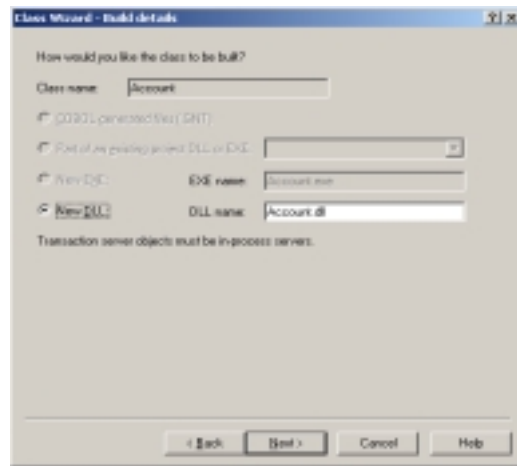


Figure 4 - Build Details

---

*If you chose OLE Automation in the first dialog of the Wizard instead of Microsoft Transaction Server, this dialog would also have an option to create an EXE rather than a DLL. In most cases, you will select DLL to create an in-process server. If you select EXE, the result is an out-of-process server. The differences between in-process and out-of-process servers are outside the scope of this white paper, but more information can be found in the Net Express documentation. Transactional components must be in-process servers, so you are not given an option to create an EXE in this example.*

---

The next dialog in the Wizard is the key dialog involved in setting the options that are most important for your COM component. Figure 5 shows the OLE Details dialog.

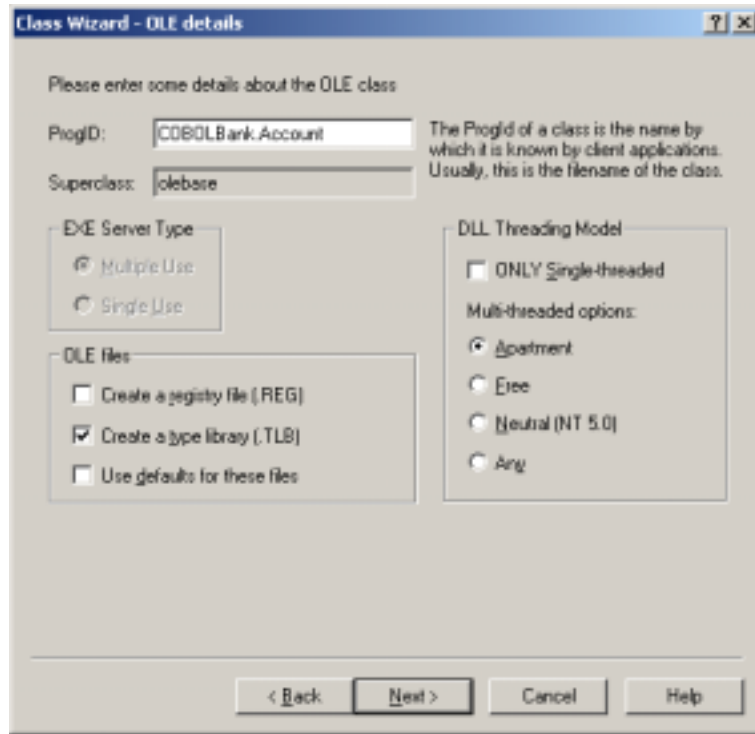


Figure 5 – The OLE Details Dialog

If you wish to make your COM component visible from managed code, it is important to ensure that you check the **Create a type library (.TLB)** box. By default, this box is NOT checked. It is not necessary to create a registry file since your component will be self-registering, so you can uncheck the **Create a registry file (.REG)** box.

The ProgID is the name by which applications will access your component. In the Bank example, there are multiple versions of Account, created in different languages. To distinguish between them, we give the ProgID for this component the name **COBOLBank.Account** to indicate that it is the COBOL version of the object. You could enter virtually any name in the ProgID field, but it makes sense to use a name that easily distinguishes your component from others.

Finally, by default, the Wizard creates a single-threaded component. Since multiple clients could call our component, we want to create a multi-threaded component. For transactional components, you would normally select **Apartment** threading. This threading model guarantees that any one instance of the component will always be called from the same thread.

---

*Note. You will notice that the box marked **Use defaults for these files** has been unchecked (checked is the default). If you uncheck this box, the wizard will prompt you for additional information that will be placed in the type library such as descriptions for the classes and interfaces and the names of interfaces. This is done in this example so that we can see the names that are generated. This will be useful later when we look at using the component from managed code.*

---

Pressing **Next** takes you to the Trigger Details dialog. The trigger program is a simple program with a standard entry point that loads the server class. It is used when the server is registered and unregistered. In most cases, you would just accept the default name (in this case, **AccountTrigger.cbl**).

The next dialog is the OLE Server Details dialog. Here you can change the description of the class. This description will be placed in the type library and in the registry when the object is registered. Pressing **Next** on this dialog takes you to the Type Information Details dialog seen in Figure 6.

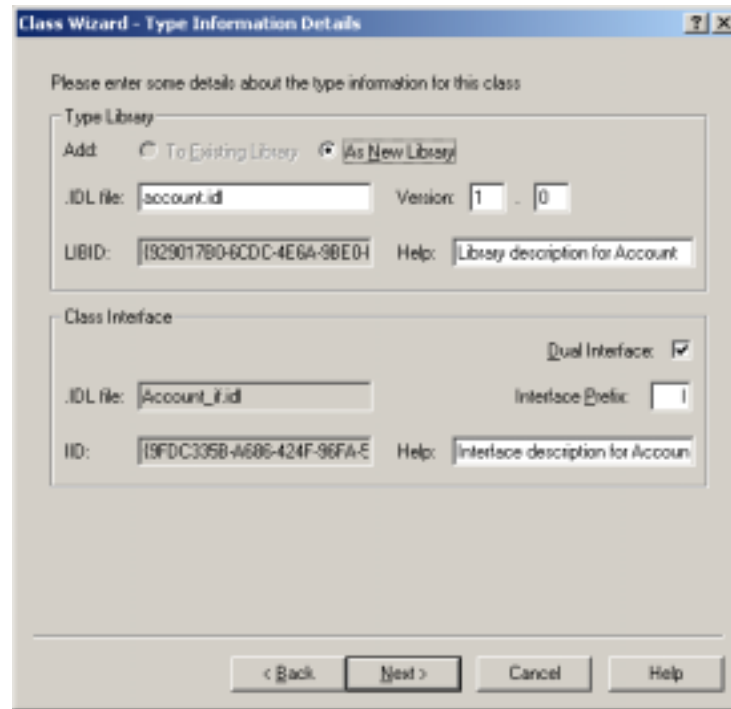


Figure 6 - The Type Information Details Dialog

There are two key items to note on this dialog. The first is the box labeled **Interface Prefix**. As you can see, this contains the letter 'I'. The interface for a component is the underlying mechanism by which a program actually calls the methods in the class. The standard naming convention for interfaces is to prefix the class name with 'I'. In this example, the interface name will be **IAccount**. We will see this name used later when the component is called from managed code.

The other key item in this dialog is the check box labeled **Dual Interface**. Leaving this box checked is key to making the component easy to use from managed code. This means that the component can be accessed via *late binding* or *early binding* code. These terms will be described later, but for now, it is important to note that the Dual Interface box must be left checked.

The final screen in the Class Wizard is the Summary screen. This enables you to review the options you selected. Pressing **Finish** will create the class files and all of the other files needed to create the project. Figure 7 shows the resulting project.

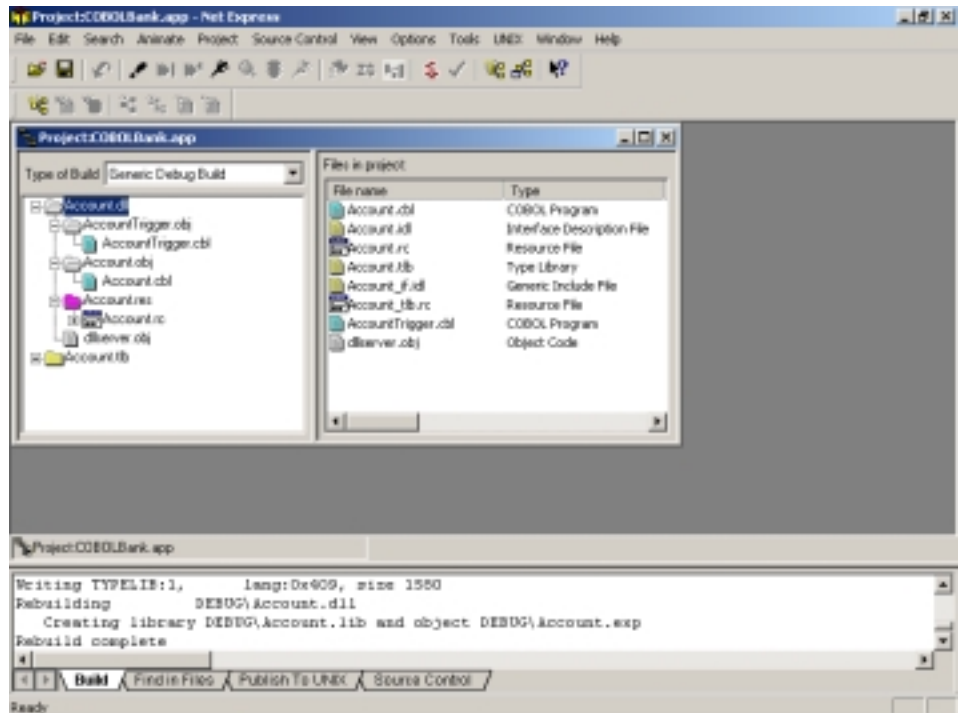


Figure 7 – The Project for the COM Component

As you can see, the Class Wizard has created eight source files. The only one you will normally change is `Account.cbl`, the source file for your class.

Although we now have a COM component that can be registered, it does not actually do anything useful yet. To add functionality to the component, we need to use the Method Wizard. We will add one method to the `Account` object to post an amount to an account. This will take two parameters, an account number and an amount, and will return a string containing the result.

Open `Account.cbl` and click on the **Add Method** button on the Object COBOL Toolbar (Figure 8)

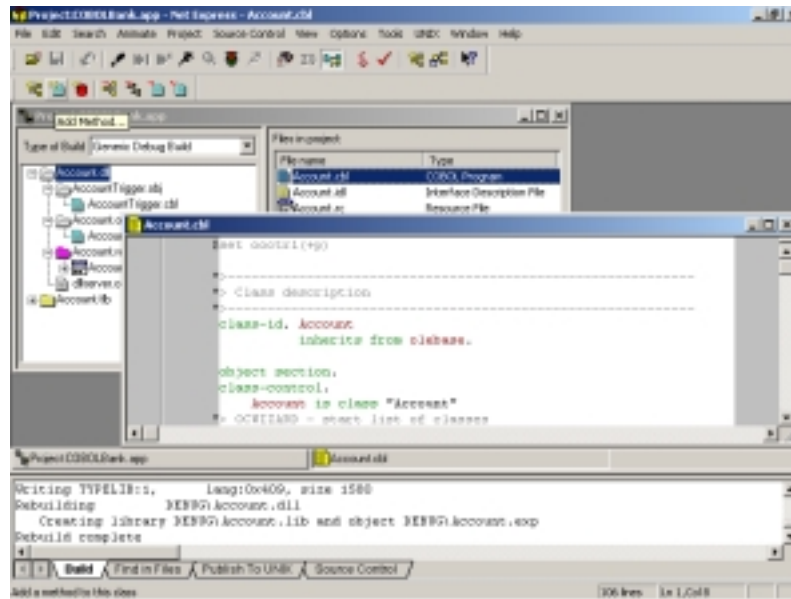


Figure 8 – Selecting the Method Wizard

---

If the Object COBOL toolbar is not visible, you can make it visible by checking the Object COBOL box on the View.Toolbars menu).

---

The first dialog in the Method Wizard allows you to select a class to add the method to. In this case, we only have one class, so simply press **Next** to go to the next dialog.

Figure 9 shows the OLE Details dialog.

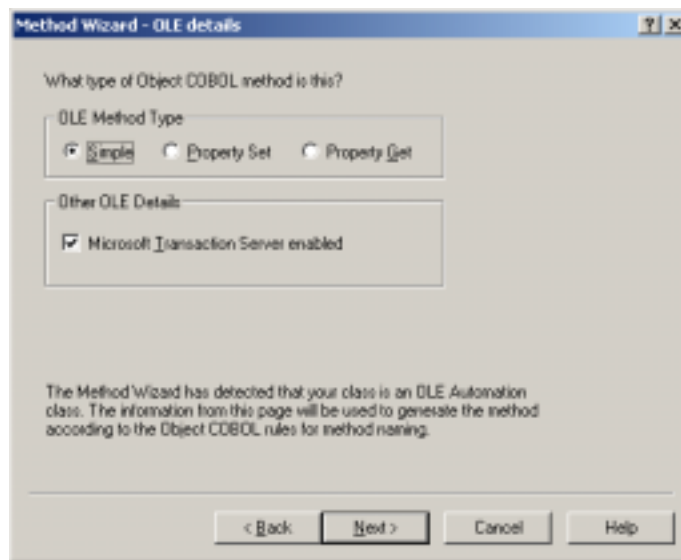


Figure 9 – OLE Details Dialog

In most cases, you will select the Simple method type. Simple methods are methods that perform actions. The Set and Get property methods are special methods used if you have particular properties in your component that you want to be able to set and retrieve.

The next dialog in the wizard prompts you to name the method and provide a *Dispatch ID*. In this example, we will call the method **Post**. The dispatch ID is a number used to identify the method in your class. Each method in a particular class will have a different dispatch ID. The Method Wizard will automatically generate one for you and you are unlikely to need to change it.

Figure 10 shows the next dialog in the Method Wizard. This is where you specify the parameters to the method. In this example, the parameters are simple and consist of signed 4-byte integers (in real life, more complex data types would most likely be used, especially for the amount).

The following dialog enables you to specify a return value. In this case, we are passing back a string, so select **pic x(32)** from the type selection box. This can be modified later to the exact length to be used.

The final dialog in the Method Wizard displays a summary of the options selected. Pressing **Finish** on this dialog creates the skeleton for the method.

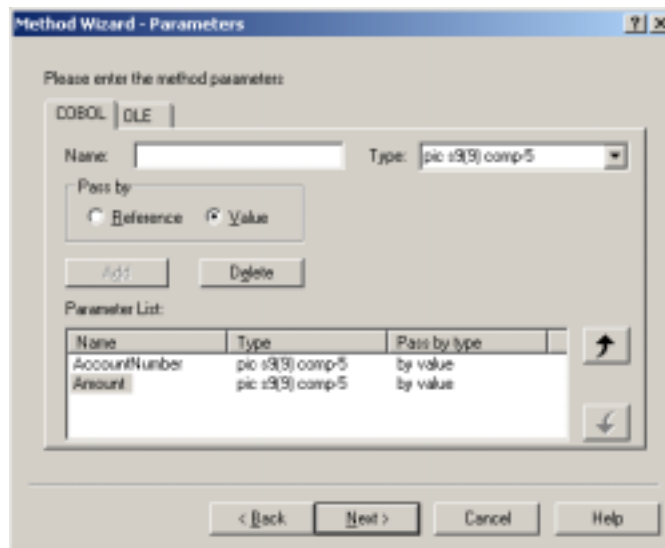


Figure 10 – Method Parameters Dialog

All that remains is for the COBOL business logic to be added to the COM component. For a full listing of Account.cbl, see the accompanying example source code.

When the project is built, the end result will be Account.dll. This contains the executable code for the component, as well as the type library. This is registered with Windows using the *regsvr32* utility (supplied with Windows). For example:

```
regsvr32 account.dll
```

### Creating the Metadata for the Account Component

We now need to create the metadata for the component to enable it to be used from managed code. As mentioned earlier, we use the TLBIMP utility from the .NET Framework SDK for this. From the Windows command line run:

```
Tlbimp account.dll /out:cobolaccount.dll
```

This creates the file `cobolaccount.dll` that contains the metadata for your component. This dll must be copied to the same directory as any managed code that uses your component.

### Using the Account Component from Managed Code

If you are using Visual Basic.NET or C#, the first thing that needs to be done is to make the metadata visible to the compiler. This is done by specifying `cobolaccount.dll` on the command line for these compilers using the */reference* compiler option (*/r* for short). For example, `/r:cobolaccount.dll`.

You also need to make the interfaces defined in the type library visible to the managed code. This is done in C# via the *using* statement and in Visual Basic.NET via the *imports* statement. These statements should reference the library name defined in the type library. In this case, it is *account*.

For example:

#### C#

```
using account;
```

#### VB.NET

```
imports account
```

---

*In the example program included with this white paper, four components are built into one dll and one type library is used to describe all of the classes. So, you will see a different library name referenced in the example, but the concept is the same.*

*Also, this white paper only provides examples in C# and Visual Basic.NET. Of course, Visual C++ could be used instead. However, Visual C++ can be used to create managed or unmanaged code, so to avoid any confusion about the type of code being discussed, the examples have been limited to C# and Visual Basic.NET.*

---

There are two ways that the COBOL component can be called from managed code. The easiest mechanism is via *early binding*. The alternative is *late binding* which is straightforward in Visual Basic.NET, but more complex in C#. *Binding* refers to the mechanism used to access code in another application.

Late binding is the older mechanism used to access COM components from languages such as Visual Basic and, until recently, was the only mechanism available for accessing COBOL COM components. Late binding occurs at run-time and is slower than early binding. In late-bound code, the underlying code that handles the call to a component must look up an object and its methods and properties each time it executes a line of code that includes that component.

Support for early binding to COBOL COM components was introduced in Net Express 3.1. With early binding, all of the information needed to access the component is available to the calling application and the component at compile time. This means that the information about the call can be verified when the calling application is compiled and calls to the component's methods and properties execute much quicker than if late binding is used.

This is why it was important to ensure that the **Dual Interface** box was checked in the Type Information Details dialog earlier. If this is not checked, the COBOL component will only be accessible via late binding. This check box ensures that the component can be called via early binding, as well as via late binding.

Accessing COM components via late binding from managed code requires use of the *reflection* classes in the .NET Framework and is beyond the scope of this white paper. The rest of this section will look at using early binding.

To create a new instance of a COBOL component from Visual Basic.NET or C#, all you need to do is declare an object of the type of the interface you want to use and create a new instance of that object, specifying the library name, followed by the class name of the component. In this case, this would be `CobolBank.Account()`. For example:

#### C#

```
IAccount objAccount;
objAccount = (IAccount) new CobolBank.Account();
```

#### VB.NET

```
dim objAccount as IAccount
objAccount = new CobolBank.Account()
```

Now it is a simple matter to use the object you have created to execute the methods in the COBOL component. The following code will execute the 'Post' method in the Account component.

#### C#

```
result = objAccount.Post (lngPrimeAccount, lngAmount);
```

#### VB.NET

```
strResult = objAccount.Post(lngPrimeAccount, lngAmount)
```

## Raising Exceptions

In all but the simplest components, it is likely that your component will need to indicate to the calling application that an application-generated error occurred. For example, if a customer exceeds their overdraft limit and a debit from an account is not allowed. One way of handling errors is to define a parameter that is used to pass back status information. However, languages such as Visual Basic.NET and C# contain extensive support for exception handling and it would be useful if the COBOL component could raise an exception that would be trapped by the calling application's exception handler. Support recently added to Net Express enables you to do this.

A new method, *RaiseException*, has been added to the support class *olesup*. The format of the call is:

```
Invoke olesup "RaiseException" using
  by reference ExceptionText
  by reference HelpFile
  by value HelpContext
```

where:

- ExceptionText is defined as PIC X(n) and is a null-terminated string that specifies the exception error text. The value of n can be between 1 and 512.
- HelpFile is defined as PIC X(n) and is a null-terminated string that specifies a help file that provides more information about the error. If this parameter is NULL, then no help file is specified.
- HelpContext is defined as PIC X(4) COMP-5 and specifies the context identifier in the help file. This parameter is only referenced if a help file is specified.

### Examples

The following would raise an exception with the message "Exception occurred in COBOL class". No help file is specified.

```
Invoke olesup "RaiseException" using
  by reference z"Exception occurred in COBOL class"
  by value 0 size 4
  by value 0 size 4
```

Note the use of the 'z' prefix on the exception string to force null termination. Also, the use of 'by value 0 size 4' to indicate a null pointer for the help file parameter.

The following code would raise the same exception, but the message is held in a working storage item.

```
01 ExceptionText pic x(50).

Move z"Exception occurred in COBOL class"
  to ExceptionText
Invoke olesup "RaiseException" using
  by reference ExceptionText
  by value 0 size 4
  by value 0 size 4
```

The following would raise the same exception, but specifies a help file and help context that provides more information about the error. It is up to the calling application to determine whether to use this information to display more information to the user or not.

```
Invoke olesup "RaiseException" using
  by reference z"Exception occurred in COBOL class"
  by reference z"ErrorFile.Hlp"
  by value 12 size 4.
```

In the managed code that calls the component, this exception could be trapped in a 'try ... catch' block. For example:

```
try
    dim objAccount as IAccount
    objAccount = new COBOLBank.Account()
    strResult = objAccount.Post(lngPrimeAccount, lngAmount)
    ContextUtil.SetComplete()    ' we are finished and happy
    exit function

catch e as Exception
    ContextUtil.SetAbort()      ' we are unhappy
```

---

*Note. This mechanism for raising exceptions is also useful if you are using Visual Basic 6.0 to create a graphical front-end for an application today. Any exception raised by the COBOL component will cause any Visual Basic ON ERROR statement to occur and the ERR structure will be setup to contain the values specified in the call to RaiseException.*

---

---

## Accessing Managed Code from COBOL

Just as you can call COBOL components from managed code, you can also call managed code objects from COBOL applications. This section looks at how you make managed code visible to your COBOL applications.

### Making Managed Code Visible to COBOL Applications

Before a managed code object can be called from a COBOL application, the metadata for the object must be read and the appropriate entries added to the Windows registry. Once this has been done, the COBOL application can access the managed code object as though it was a COM component. One way to do this is to use *REGASM*. *REGASM.EXE* is provided with the Microsoft.NET Framework SDK.

The simplest use of *REGASM* is simply:

```
regasm managedcode.dll
```

where *managedcode.dll* is the name of the dynamic link library that contains the managed code object. For example:

```
regasm charpbank.dll
```

Another utility provided with the .NET Framework SDK is *REGSVCS*. Like *REGASM*, *REGSVCS.EXE* loads the information from the metadata into the registry. However, *REGSVCS* can also install the component as part of a COM+ 1.0 application and configure services in the component. The example supplied with this white paper uses *REGSVCS* since the components are packaged into a COM+ application.

For more information on *REGASM* and *REGSVCS*, refer to the documentation that comes with the Microsoft.NET Framework SDK.

### Example.

In the previous section, you saw how an account object written in COBOL could be called from managed code. In this section, we look at how an account object written in C# or Visual Basic.NET can be called from COBOL.

---

*Note. The example programs included with this white paper demonstrate an account object written in C# being called from the COBOL MoveMoney object. The code in the example is slightly more complex than that shown here because the example uses COM+ Transaction Services. However, the concepts are the same.*

---

The examples create a dll called `csharpbank.dll` that contains the classes that implement the account object as well as other objects. Before the classes in this dll can be called from COBOL, the information needs to be loaded into the registry using REGASM or REGSVCS as described previously. This would be done using the command lines:

```
regasm chsarpbank.dll
```

or:

```
regsvcs csharpbank.dll
```

It is worthwhile taking a brief look at what information is placed in the registry. COM objects are accessed via a ProgID (as you saw in the previous section). However, managed code objects do not have the concept of a ProgID. So, when REGASM or REGSVCS place information in the registry, the information is synthesized from the metadata for the managed code object. To see how this is done, we can use a utility called *ILDASM* (provided with the .NET Framework SDK) to look at the metadata for `csharpbank.dll`. The command line would be:

```
ildasm charpbank.dll
```

Figure 11 shows the window that is displayed.

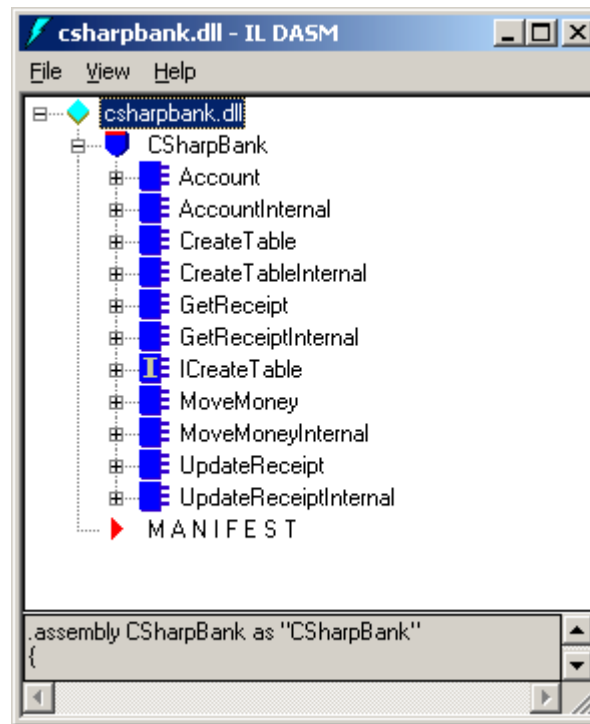


Figure 11 - ILDASM

In the tree display in the window, you can see the name **CSharpBank** under the name of the dll. This is the *namespace* of the component. Below that, you see the names of the classes in this namespace. One of these is the class we are interested in – **Account**. The ProgID that is placed in the registry is synthesized from the namespace, followed by the class name. The two parts of the ProgID are separated by a period. In this case, the ProgID will be ‘CSharpBank.Account’.

---

*For more information on namespaces, refer to the documentation that comes with the .NET Framework SDK.*

---

Now that the information is in the registry, we can access the C# code just as if it was a COM component. The steps needed to do this are documented in the Net Express documentation, but they can be summarized as follows:

- Use the compiler directive `ooctrl(+P)` when compiling the component.
- Define a Class-Control section that includes the components you will be accessing. The names of the components should be prefixed with '\$OLE\$' to indicate that they are COM components.
- Declare at least one variable of type object reference to be used when calling the component. The number of object reference variables you will need will depend on the complexity of the component you are accessing.
- Use the 'New' method to create a new instance of the component. This returns a reference to the component.
- Invoke the methods in the component using the object reference returned by the 'New' method.
- Finalize all objects before terminating the application using the 'Finalize' method.

In this case, the code would be as follows:

```
$set ooctrl(+p)
class-control.
    CSharpAccount is class "$OLE$CSharpBank.Account"

working-storage section.

01 AccountObj          object reference.
01 ReturnString        pic x(160).
01 Account             pic s9(9) comp-5.
01 Amount              pic s9(9) comp-5.

procedure division.
    invoke CSharpAccount "new" returning AccountObj
    invoke AccountObj "Post" using Account
                                Amount
                                returning ResultString
    invoke AccountObj "finalize" returning AccountObj
stop run.
```

## Handling Exceptions

A common mechanism for reporting errors in managed code is by raising exceptions. These will be seen by the COBOL code as a COM exception. In order to trap exceptions, you will need to install an exception handler for your code. If you are calling the managed code from a standard procedural COBOL program, then this is relatively straightforward and you should refer to the Net Express documentation for information on how to do this (refer to the section “OLE Automation Exceptions” in the Distributed Computing manual).

However, if you are calling the managed code from a COBOL COM component, then it gets a bit more complicated. Exceptions are handled using classes in the COBOL class library. However, these classes do not handle the case where the exception handler is a method in a COM component (as seen in the MoveMoney component in the example included with this white paper). Therefore, the exception handler must be a method in a standard COBOL class. To make this easier, the included sample includes a class called COMHelper that provides methods to register an exception handler that can be in your COM component. Refer to COMHelper.cbl and MoveMoney.cbl for information on how this class is used.

## Visibility of Managed Types

Not all classes and methods declared in managed code are visible via COM interfaces. In general, *public* classes are visible, but classes declared as *private* are not. In addition, public methods that are declared as *static* are not visible via COM.

In order to make static methods visible to your COBOL code, a simple managed code wrapper needs to be created that defines a method that is not static.

An example of this is if you wish to make calls to the WinForms classes from your COBOL application. The WinForms classes are a set of classes provided by the .NET Framework that enable the creation and manipulation of graphical user interfaces objects. Because the WinForms classes are managed code, they can be accessed just like any other managed code.

The first thing to be done is to add the information into the registry. This is done running regasm against the file that contains the WinForms classes. This is system.winforms.dll that is installed by the .NET Framework SDK. For example:

```
regasm system.winforms.dll
```

Now, you can create a COBOL application that creates Windows forms and controls. For example, the following application creates a simple form with one button on it.

```
$set ooctrl(+P)
class-control.
    WinForm is class "$OLE$System.WinForms.Form"
    WinButton is class "$OLE$System.WinForms.Button"

data division.
working-storage section.

01 Form                object reference.
01 Button              object reference.
01 FormControls        object reference.

procedure division.
Main section.
    invoke WinForm "New" returning Form
    invoke Form "setText"
        using z"This is a System.WinForms.Form"
    invoke Form "Show"
    invoke WinButton "New" returning Button
    invoke Button "setText" using z"Button"
    invoke Form "getControls" returning FormControls
    invoke FormControls "Add" using Button.
```

---

*For details on the methods used here, refer to the documentation that is included with the .NET Framework SDK*

---

This is not too useful since, in most cases, you will want to pass control to the form so that it can process messages. To do that, an application must invoke the **Run** method in the **Application** class. However, the Application class is declared as **public static**, meaning that it is not visible to COM. To overcome this, a simple class can be created in C# that makes the run method visible to COBOL. An example would be as follows:

```
// Wrapper class for the Application class. This makes
// the relevant static methods in the Application class
// visible to a COBOL application built using
// Net Express.

using System;
using System.Windows.Forms;
using Microsoft.Win32.Interop;

// Specifying the namespace 'COBOL' will ensure that
// all ProgIDs get prefixed with 'COBOL.', i.e.
// 'COBOL.Application'.

namespace COBOL
{
    public class Application
    {
        public void Run(Form formobject)
        {
            System.Windows.Forms.Application.Run(formobject);
        }
    }
}
```

This is compiled and registered as *COBOL.Application* using the following commands:

```
csc /target:library /r:System.dll /r:System.Windows.Forms.dll
    /r:Microsoft.Win32.Interop.dll appclass.cs
regasm appclass.dll
```

Now, it is a simple matter to update the application created earlier to use this class to pass control to the form we have created.

```

$set ooctrl(+P)

class-control.
  WinForm is class "$OLE$System.WinForms.Form"
  WinButton is class "$OLE$System.WinForms.Button"
  COBOLApplication is class "$OLE$COBOL.Application".

data division.
working-storage section.

01 Form                object reference.
01 Button              object reference.
01 FormControls        object reference.
01 Application         object reference.

procedure division.
Main section.
  invoke WinForm "New" returning Form
  invoke Form "setText"
    using z"This is a System.WinForms.Form"
  invoke Form "Show"
  invoke WinButton "New" returning Button
  invoke Button "setText" using z"Button"
  invoke Form "getControls" returning FormControls
  invoke FormControls "Add" using Button
*> Use our new C# class to enable us to access the
*> Application.Run method
  invoke COBOLApplication "New" returning Applon
  invoke Application "Run" using Form
*> Ensure we clean up the objects we have created
  invoke Application "Finalize" using Application
  invoke Form "Close"
  invoke Button "Finalize" returning Button
  invoke FormControls "Finalize"
    returning FormControls
  invoke Form "Finalize" returning Form
stop run.

```

---

## Creating a Web Service in COBOL

A Web Service is a unit of application logic providing data and services to other applications. Applications access Web Services via Web protocols and data formats such as HTTP, XML, and SOAP, with no need to worry about how each Web Service is implemented.

The key to creating Web Services in COBOL is the use of the Microsoft SOAP Toolkit. This provides Microsoft's implementation of SOAP, as well as the tools needed to create Web Services. SOAP stands for the Simple Object Access Protocol. The actual specification can be found at [www.w3.org/tr/soap](http://www.w3.org/tr/soap). SOAP is based in XML and describes a messaging format for machine-to-machine communication.

The SOAP Toolkit provides two items needed to turn a COM component into a Web Service that runs under Microsoft Internet Information Server:

- A server-side component that maps invoked Web service operations to COM object method calls as described by the Web Services Description Language (WSDL) and Web Services Meta Language (WSML) files.
- A WSDL/WSML Generator tool that generates the WSDL and WSML files for you, relieving you of the process of manually creating such files.

### Web Services Description Language

The Web Services Description Language (WSDL) is an XML format for describing the network services offered by your Web Service. For each of the services, the WSDL file also describes the format that the client must follow in requesting that service. Since the WSDL file sets up requirements for both the server and the client, this file is like a contract between the two. The server agrees to provide certain services only if the client sends a properly formatted SOAP request.

For example, suppose a WSDL file defines a service called *BankAccount*. This service describes a operations such as *PostToAccount* and *DebitFromAccount*. You place this file on the server. A client who wishes to send a SOAP request to the server first obtains a copy of this WSDL file from the server. The client then uses the information in this file to format a SOAP request. The client sends this request to the server. The server executes the requested operation and sends the resulting balance back to the client as a SOAP response.

### Web Services Meta Language

A Web Services Meta Language (WSML) file provides information that maps the operations of a service (as described in the WSDL file) to specific methods in the COM object. The WSML file determines which COM component to load to service the request for each operation.

## Creating WSDL and WSML Files

Writing your own WSDL and WSML files can be tedious. However, the Microsoft SOAP Toolkit 2.0 provides a tool, the WSDL/WSML Generator that generates these files for you. All you need to do is provide your COM component as input, specify the classes and methods you want to make visible as part of your Web Service and the WSDL/WSML generator takes care of generating the necessary WSDL and WSML files.

## Example

In this example, we use the COM component created in the sample supplied with this white paper. The component is COBOLBank.dll. This contains four classes, one of which is the account class we created earlier in this paper (We have simply built four classes into one dll rather than deploying them as four separate dlls).

First, you will need to create a virtual directory for the Internet Information Server that points to the location of the COM component. This is done using the Internet Services Manager. In this example, we create a virtual directory called **COBOLBankSample** that points to the directory containing COBOLBank.dll (e:\dotnet\soap).

Next, you need to install the Microsoft SOAP Toolkit 2.0 (downloadable from [www.microsoft.com](http://www.microsoft.com)). This adds a new program group to the Windows Start menu called 'Microsoft SOAP Toolkit'. From this menu, run the 'WSDL Generator'. Figure 12 shows the first dialog that enables you to select the COM component that you will be deploying as a Web Service.

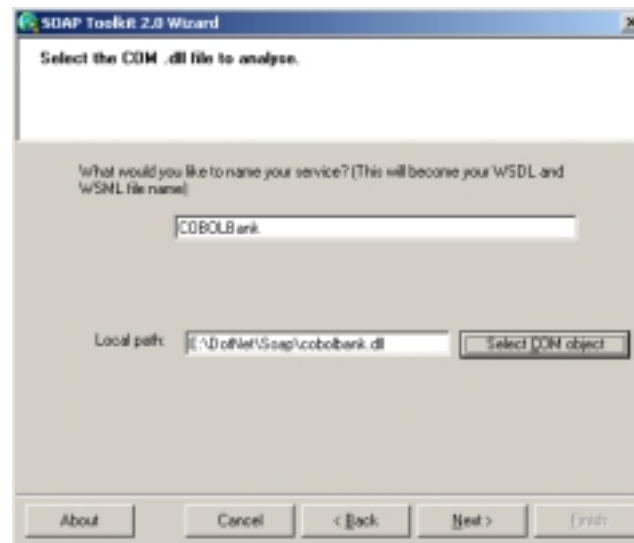


Figure 12 – Selection of the COM Component

Here, we are going to give the service the name **COBOLBank**. This is the name that will be used by clients to access the service. We also specify the location of the COM component.

The next dialog (Figure 13) displays the classes and methods implemented by the component.

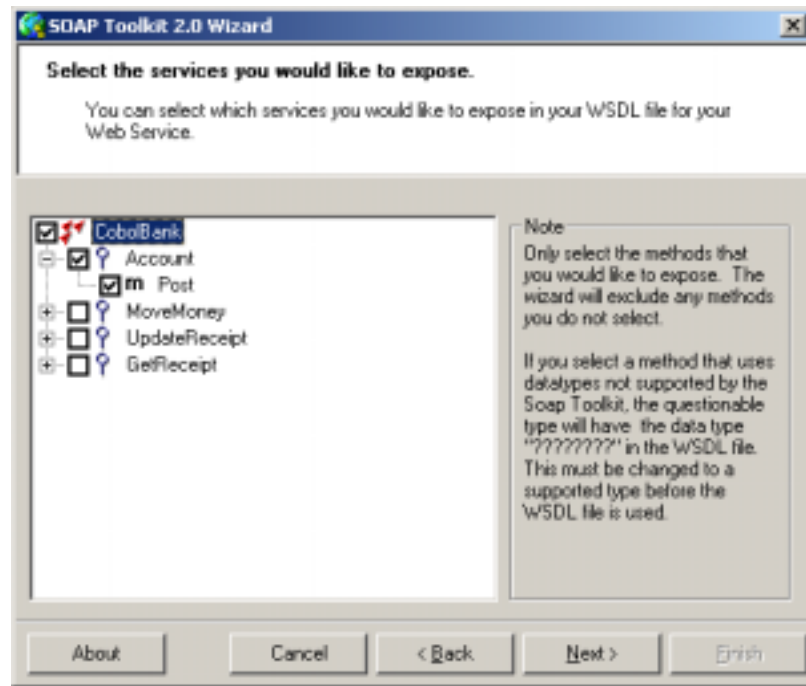


Figure 13 – Selection of Classes and Methods

We select the **Account** class and the **Post** method to be visible via the Web Service. The other classes and methods in the component will not be visible to SOAP clients.

The next dialog in the WSDL/WSML Generator prompts us for the URL for the service. In this case, we enter <http://localhost/COBOLBankSample> since we will be testing on the same server (note that COBOLBankSample is the name of the virtual directory we set up earlier).

The final dialog prompts us for a location for the output WSDL and WSML files. We simply specify the name of the directory that contains the COBOL COM component. The WSDL/WSML Generator will then create two files - COBOLBank.WSDL and COBOLBank.WSML.

Now, we can create a client that accesses this Web Service. The client machine must also have the SOAP client (supplied with the SOAP Toolkit) installed. In this example, we will use VBScript to create a simple command-line driven client, but any language that can call the SOAP APIs could be used.

We create a script called **post.vbs** that contains the following:

```
set soapclient = CreateObject("MSSOAP.SoapClient")
Call soapclient.mssoapinit
    ("http://localhost/COBOLBankSample/COBOLBank.wsdl",
    "COBOLBank", "AccountSoapPort")
wscript.echo soapclient.Post(1, 1000)
```

As can be seen, the client simply creates a new instance of object **MSSOAP.SoapClient** and calls the method **mssoapinit**. After that, it can invoke any of the methods in the COBOL COM component that have been exposed via the

Web Service. In this case, we invoke the **Post** method with account number 1 and an amount of \$1000. The script displays the returned result in a message box.

It is worthwhile looking at the parameters to `mssoapinit`. This method takes three parameters, as follows:

- The name and location of the WSDL file for the service. In this case, it is <http://localhost/COBOLBankSample/COBOLBank.wsdl>.
- The name of the service we want to access. In some cases, the WSDL file might publish details about many different services. In this case, we are only publishing one service – COBOLBank.
- The name of the port we want to access. This name was generated by the WSDL Generator and is based on the class name. If we look in the file COBOLBank.WSDL, we will see the following section:

```
<portType name='AccountSoapPort'>  
  <operation name='Post' parameterOrder='p0 p1'>  
    <input message='wsdlns:Account.Post' />  
    <output message='wsdlns:Account.PostResponse' />  
  </operation>
```

We can see that the name **AccountSoapPort** has been used.

---

## Conclusion

Hopefully, this white paper has demonstrated to you that you do not need to wait for new tools in order to maximize your investment in COBOL when developing applications for the new .NET platform. The level of support for COM in Net Express and the excellent support for interoperability provided by the .NET Framework means that Net Express can be used today to deploy your COBOL code under .NET, either as Web Services or as part of an application developed using the new Microsoft compilers that target the .NET framework.

More information on Net Express can be found on the MERANT web site at <http://www.merant.com>.

For more detailed information on .NET, the .NET Framework and managed code, refer to <http://www.microsoft.com/net>.

More information on Web Services and the SOAP Toolkit can be found at <http://msdn.microsoft.com/webservices>.

\*Note: Needs Run-Time Fixpack (version 3.1.005 or later) which can be found at <http://support.merant.com/websupport/websync/netx31.asp>

## FOR MORE INFORMATION

MERANT is the leading provider of software solutions for enterprise change management; development, transformation and integration of legacy applications; and data connectivity. More than 5 million professionals use MERANT technology at 60,000 customer sites, including the entire Fortune 100 and the majority of the Global 500. Founded in 1976, MERANT has 1,500 employees and more than 600 technology partners.

### MERANT Worldwide Sales

Corporate headquarters. 301-838-5000

**Asia/Pacific** ..... (+65) 834 9880

#### **Australia**

Melbourne ..... (+61) 3 9522 4466

Sydney ..... (+61) 2 9904 6111

**Belgium** ..... (+32) 15 30 77 00

**Brazil** ..... (+55) 11 3048 8304

**Canada** ..... (+1) 905-306-7280

**Denmark** ..... (+45) 33 37 72 90

**Finland** ..... (+35) 8 962 27 25 00

**France** ..... (+33) 1 70 92 94 94

#### **Germany**

Dortmund ..... (+49) 231 75 850

Munich ..... (+49) 89 962 7 10

#### **Italy**

**Milan** ..... (+39) 02 694 34 01

**Rome** ..... (+39) 06 51 53 93 1

**Japan** ..... (+81) 3 5401 9600

**Netherlands** ..... (+31) 33 450 20 70

**Norway** ..... (+47) 22 91 07 20

**Portugal** ..... (+35) 11 384 5010

**South Africa** ..... (+27) 11 237 7800

#### **Spain**

Barcelona ..... (+34) 93 435 7001

Madrid ..... (+34) 91 302 82 26

#### **Sweden**

Gothenburg ..... (+46) 0 31 727 78 40

Stockholm ..... (+46) 0 8 54 51 33 90

#### **United Kingdom**

Newbury ..... (+44) 1635 32 646

St. Albans ..... (+44) 1727 812 812

**United States** ..... 800 547 4000

Also available at authorized resellers.

**877-772-4450**

[www.merant.com](http://www.merant.com)