The Inside Track
"COM See, COM Saw"
January 2001

If my titles get to be too cute, blast me some e-mail. That said I really like the title this month. You see, I remember only about a half dozen phrases from the French I learned 30 years ago and this phrase has stuck in my head. "Comme ci, comme ça" which is a response to the question "How are you?" The answer means "so, so". I've mangled the spelling for this month's title to reflect the topic COM (Component Object Model).

Inside ~~Great Plains~~Microsoft?

I'm betting that you've noticed that while my column is called "The Inside Track," almost all of the technologies I've been talking about are straight out of the Microsoft playbook. So, you say, why didn't I just call this column "The Outside Track"? By now you've heard that Microsoft has signed an intent to acquire Great Plains so I could just argue that the outside is becoming the inside. Wouldn't that be a copout? And the truth is that I heard about the acquisition on December 21st just like everyone else. Oh, I heard one person with a rumor a week or two before, but I dismissed it. I'd heard that one before!

No the real reason that I've been focusing on Microsoft technologies in this column is just what I said in my first column. At Great Plains, we are building our next generation products on Microsoft's family of technologies called .NET. Nonetheless, I've started to feel that I need to bring some of what I've been writing about together with what we are doing in development at Great Plains.

COM in Common

When two programs talk to each other its called Inter-application Communication (IAC). All of the .NET languages run in the Common Language Runtime and use the .NET Framework class library. I haven't delved into the IAC part of .NET myself, so I don't know much about the specifics except one. When the two programs are on different machines, the preferred IAC mechanism is XML over HTTP using a defined format called SOAP (Simple Object Access Protocol).

Today, we can use COM for the same task. COM defines how two chunks of code communicate. If one chunk is in an .EXE and the other in a .DLL, this is call in-process COM. If both chunks are in .EXE's then this is called out-of-process COM. If both chunks are in .EXE's but on different machines, its called Distributed COM or DCOM. You probably knew all that.

So DCOM and SOAP are intended to accomplish pretty much the same thing. While I think we can assume that Microsoft would have liked COM to be ubiquitous, the Internet has dictated that HTTP and XML will be the IAC mechanism of choice.

Just like any technology, there will be a time while COM continues to be the dominant IAC mechanism. So we'll need a transitional technology. What is it? It's a part of .NET that Microsoft calls COM Interoperability or COM Interop for short. So thinking about COM still makes sense today.

COM has existed in Dexterity for sometime now. Continuum is based on COM. Of course Continuum is about an external application reaching into Dexterity using COM. Dexterity has been able to use outgoing COM via VBA (Visual Basic for Applications). And we know how useful that is!

But SanScript programmers have been left out of the picture. You could always figure out a way to pump data from Dynamics or eEnterprise into a mail merge in Word or an Excel spreadsheet,

but it hasn't been as clean as you (or we) might like. Well take heart because here comes SanScript based COM. That's right boys and girls, COM right from within native Dexterity scripting.

Now you may see why I had a lot of fun with this month's title! COM is the tool we *see* today, but it's not the tool of choice in a .NET/Internet world of tomorrow. While COM will be with us for a while, in the future COM will be the technology we *saw* in the past. In other words a "so, so" technology. I know, way too cute.

Now that you've heard the good news about COM in Dexterity, let's look at how it's done.

Binding and Type Libraries

There are two basic ways of talking to a COM component (or ActiveX if you prefer): early bound and late bound. In Visual Basic you use early (compile time) binding by adding a reference (menu Project->References…) to your project and then using the name of the component to declare a variable:

```
Dim rs As ADODB.Recordset
```

In this example, ADODB refers to a type library that contains a description of the RecordSet component. A type library tells your development environment which components are available and the components properties, methods and events. You may remember last month when we talked about how the new .NET language C# handles these issues. A Visual C++ programmer creates a type library by compiling a .MDL (short for MIDL the Microsoft Interface Definition Language) file into a .TLB (type library) file. If you write a component in VB (ActiveX .EXE or ActiveX .DLL project type), VB builds the type library for you. Sometimes the type library is a separate file and sometimes its included right in the .EXE or .DLL (or .OCX). Wherever it comes from, if you're using early binding, the type library is used by the VB compiler to construct the correct call to the COM component. Because this work is carried out at compile time, its fast at runtime.

On the other hand, if you declare a variable as:

```
Dim rs As Object
```

Then the type of the component won't be known at compile time, this is known as late (at runtime) binding. Because constructing the call to the COM component must now take place at runtime it's slower. If you've used a scripting language (VBScript or JavaScript), you've used late binding, it's the only choice.

COM in Dexterity

Dexterity handles referencing a type library with the new "import" statement on the first line of a script.

```
import "<filename>";
```

or:

```
import "<GUID>[,version]";
```

For example:

```
import "shell32.dll";
```

or:

```
import "{00000205-0000-0010-8000-00AA006D2EA4}";
```

In Dexterity you can also use either early or late binding. If you use the import statement, you can declare a variable of a specific type and create an object of that type for early binding:

```
local GPData.Customer Cust;
Cust = new GPData.Customer();
```

Similarly in VB, you would write:

```
Dim Cust as GPData.Customer
Set Cust = New GPData.Customer
```

To use late binding in Dexterity no import statement is needed and you declare the variable to be of type “reference”:

```
local reference Cust;
Cust = COM_CreateObject("GPData.Customer");
```

In VB no reference is needed and the variable type is “Object”:

```
Dim Cust as Object
Set Cust = CreateObject(“GPData.Customer”)
```

Calling a method in Dexterity is easy:

```
b = Cust.Load('(L) ID');
```

In VB:

```
b = Cust.Load(txtID.Text)
```

Last is reading from and writing to a property. In Dexterity:

```
'(L) Name' = Cust.Name;
Cust.Name = '(L) Name';
```

In VB the terms are “get” and “let”, but the syntax is very similar:

```
txtName.Text = Cust.Name
Cust.Name = txtName.Text
```

For the advanced student, there is a COM_GetObject() function in Dexterity that is similar to VB’s GetObject() and you can specify the server in an optional second argument to COM_CreateObject() for DCOM.

Unfortunately, COM in Dexterity is not in the current product so you can’t try my examples. And my example is purposely simple to avoid some of the holes in the current partial implementation. To give you a little more context for the examples, I’m going to include the code for the VB COM component I used with Dexterity, the VB test program I used to test the COM component and then the SanScript for exactly the same program in Dexterity.

The VB COM Component

In VB, create a new ActiveX DLL project, name it GPData, paste the following code into the class module and name the class module Customer. NOTE: you will need to modify the ConnStr to refer to an appropriate ODBC data source name and you will need to add a reference to ADODB.

ADODB is named something like “Microsoft ActiveX Data Objects 2.5 Library” when you are in the References window in VB.

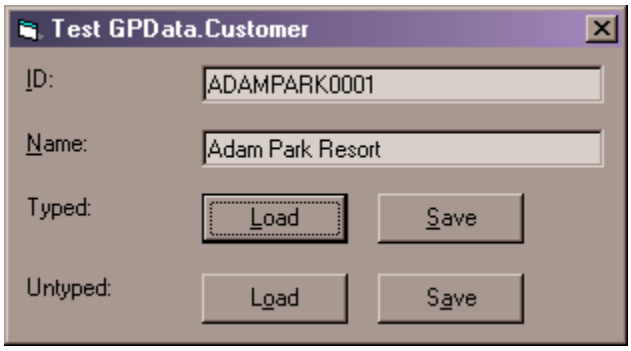
```
Option Explicit

Private Const ConnStr = "DSN=TWO;UID=sa;"

Private m_ID As String
Private m_Name As String

Public Property Get ID() As String
       ID = m_ID
End Property

Public Property Let ID(ByVal NewID As String)
       m_ID = NewID
End Property

Public Property Get Name() As String
       Name = m_Name
End Property

Public Property Let Name(ByVal NewName As String)
       m_Name = NewName
End Property

Public Function Load(ByVal LoadID As String) As Boolean
       Dim rs As ADODB.Recordset

       ID = LoadID
       Set rs = New ADODB.Recordset
       Call rs.Open("select CUSTNAME from RM00101 where CUSTNMBR = '" + ID +
"'", ConnStr, adOpenStatic, adLockReadOnly, -1)
       If rs.BOF Or rs.EOF Then
              Load = False
       Else
              m_Name = rs.Fields("CUSTNAME")
              Load = True
       End If
       rs.Close
       Set rs = Nothing
End Function

Public Function Save() As Boolean
       Dim cmd As ADODB.Command
       Dim recs As Integer

       Set cmd = New ADODB.Command
       cmd.CommandText = "update RM00101 set CUSTNAME = '" + Name + "' where
CUSTNMBR = '" + ID + "'"
       cmd.CommandType = adCmdText
       cmd.ActiveConnection = ConnStr
       cmd.Execute recs
       Save = recs = 1
End Function
```

The VB Test Program

Create a Standard EXE project, add the controls and set their names as given below. If you add the COM component and the test program to the same Project Group in VB, it’s a lot easier to

debug and you don't need to "make" either program to run it. You will also need to add a reference in the test program to the COM component as "GPData" in the References window.

My version of the VB test program has 6 controls and 4 labels. The 6 controls are named: txtID, txtName, cmdLoad, cmdSave, cmdLoadUntyped and cmdSaveUntyped. Here is what my form looks like at run time:



If you have used exactly the same names for the controls, you can open a code window for the project and paste the following code in directly.

```
Option Explicit

Private Sub cmdLoad_Click()
	Dim Cust As GPData.Customer

	Set Cust = New GPData.Customer
	If Cust.Load(txtID.Text) Then
		txtName.Text = Cust.Name
	Else
		txtName.Text = ""
		MsgBox "Customer ID not found"
	End If
	Set Cust = Nothing
End Sub

Private Sub cmdLoadUntyped_Click()
	Dim Cust As Object

	Set Cust = CreateObject("GPData.Customer")
	If Cust.Load(txtID.Text) Then
		txtName.Text = Cust.Name
	Else
		txtName.Text = ""
		MsgBox "Customer ID not found"
	End If
	Set Cust = Nothing
End Sub

Private Sub cmdSave_Click()
	Dim Cust As GPData.Customer

	Set Cust = New GPData.Customer
	Cust.ID = txtID.Text
	Cust.Name = txtName.Text
	If Cust.Save Then
		txtID.Text = ""
		txtName.Text = ""
		MsgBox "Customer Saved"
```

```
	Else
		MsgBox "Customer ID not found"
	End If
	Set Cust = Nothing
End Sub

Private Sub cmdSaveUntyped_Click()
	Dim Cust As Object

	Set Cust = CreateObject("GPData.Customer")
	Cust.ID = txtID.Text
	Cust.Name = txtName.Text
	If Cust.Save Then
		txtID.Text = ""
		txtName.Text = ""
		MsgBox "Customer Saved"
	Else
		MsgBox "Customer ID not found"
	End If
	Set Cust = Nothing
End Sub
```

The Dexterity Test Program

The Dexterity version looks almost the same as the VB version.

There are 4 scripts on the form, one for each button. If you were actually going to run this program, you would need to know the GUID created by GPData.dll replacing mine (hint: use OLE View in Visual Studio Tools) or put the path (either relative or absolute) to GPData.dll in the import statements.

First, the “Typed Load” button.

```
import "{0E0B0D9E-A85D-4248-85BD-ECC2B3C0A8F3}";

local GPData.Customer Cust;
local boolean b;

Cust = new GPData.Customer();
Cust.ID = '(L) ID';
b = Cust.Load('(L) ID');
```

```
if b then
       '(L) Name' = Cust.Name;
else
       '(L) Name' = "";
       warning "Customer ID not found";
end if;
Cust = null;
```

The “Typed Save” button:

```
import "{0E0B0D9E-A85D-4248-85BD-ECC2B3C0A8F3}";

local GPData.Customer Cust;
local boolean b;

Cust = new GPData.Customer();
Cust.ID = '(L) ID';
Cust.Name = '(L) Name';
b = Cust.Save();
if b then
       '(L) ID' = "";
       '(L) Name' = "";
       warning "Customer Saved";
else
       warning "Customer ID not found";
end if;
Cust = null;
```

The “Untyped Load” button:

```
local reference Cust;
local boolean b;

Cust = COM_CreateObject("GPData.Customer");
Cust.ID = '(L) ID';
b = Cust.Load('(L) ID');
if b then
       '(L) Name' = Cust.Name;
else
       '(L) Name' = "";
       warning "Customer ID not found";
end if;
Cust = null;
```

And last, the “Untyped Save” button:

```
local reference Cust;
local boolean b;

Cust = COM_CreateObject("GPData.Customer");
Cust.ID = '(L) ID';
Cust.Name = '(L) Name';
b = Cust.Save();
if b then
       '(L) ID' = "";
       '(L) Name' = "";
       warning "Customer Saved";
else
       warning "Customer ID not found";
end if;
Cust = null;
```

Talk Again Later

There you have it. While we build the new road with .NET, we are building bridges that will bring us all forward into that wonderful new Internet based world. I hope you're feeling better than just "so, so" about .NET. It will be an exciting and challenging trip, but even in software development, the fun is in the trip not the arrival.

Until next month, signing off,
Karl Gunderson
Technical Evangelist