A Beginner's Guide to Accessing BAPIs with the SAP DCOM Connector

Thomas G. Schuessler

Thomas G. Schuessler is the founder of ARAsoft, which offers products, consulting, custom development, and training in the area of integration with R/3 world-wide. He is the author of SAP's CA925 "Programming with BAPIs in Visual Basic" class, and he has spoken at many SAP TechEd and SAPPHIRE events. Prior to founding ARAsoft, he worked with SAP AG and SAP America for seven years.

(complete bio appears on page 66)

SAP's Business Application Programming Interfaces (BAPIs) can be called from a variety of platforms. Among the most popular are the Microsoft Windows (NT, 95, 98, 2000) platforms. Many of you, no doubt, have worked with the SAP ActiveX controls to invoke BAPIs (or other ABAP functions) on a Microsoft platform.

The problem with the SAP ActiveX controls, however, is that they are not optimized to support server-based application scenarios. They are best suited for single-user desktop applications and for intranet scenarios where ActiveX components are allowed to run on the client.

For server-based applications, you will find the SAP DCOM Connector (SDC) to be a better runtime environment. This new middleware was jointly developed by SAP and Microsoft to support high-volume server applications as well as single-user desktop applications. With it, you can build both fat client and thin(ner) client Internet and intranet applications in any programming language that supports Microsoft's Component Object Model (COM). (In this article, I will use Visual Basic for the sample code, but you should have no problem applying the concepts that I will be presenting to any development tool/programming language that has COM support.)

The first version of SDC had two shortcomings, though:

- SDC required Microsoft Transaction Server (MTS), and not all customers want to use this product.
- Your program had to separately tell each BAPI object which R/3 system to talk to. Each object would have its own session with R/3.

This used additional resources in R/3 and made it impossible to use BAPIs that require an explicit call to the BapiService.TransactionCommit BAPI.

The new version of the SDC (4.6A) fixes these shortcomings.[1] It does not require MTS, but of course still supports it. And it introduces a session object that allows all objects in an application to share one connection to R/3.

This article will introduce you to the basic design principles behind SDC, show you how to build BAPI-enabled applications with SDC in Visual Basic, and discuss some advanced concepts for SDC-based applications. In order to keep things simple, we will assume that our applications are built without MTS. So I will not be covering MTS-specific issues here.

SAP DCOM Connector Design Principles

SAP and Microsoft developed the SAP DCOM Connector to accomplish the following:

- Enable developers to generate lightweight components ("proxies") that contain the metadata for BAPIs and other RFC-enabled Function Modules (RFMs).[2] This eliminates the need to perform runtime retrieval of a business object's metadata (as is the case with the SAP BAPI ActiveX control). It also provides support for IntelliSense[3] in tools like Visual Basic, since these proxies contain the required Type Library information.

- Allow scripting languages like VBScript to directly access the BAPIs and other RFMs by using the Variant data type for most parameters.

- Support ActiveX Data Objects (ADO) Recordsets for structure and table parameters, thus allowing programmers to benefit from ADO-aware visual controls.

- Support Microsoft Transaction Server for customers who do want to use MTS as their standard transaction monitor. (Be aware, however, that SDC does not currently support MTS transactions, since R/3 does not support two-phase commit. This may change in the future.)

- Support connection pooling for server applications where many users use the same userID. Connection pooling works on a per-userID basis. You would use this technology in situations where you have many users logging on to SAP using the same userID, something you typically find in Internet applications.

- Enable client programs to invoke BAPIs and other RFMs without having to install SAP components on the client. This implies support for a multi-tiered architecture.

As I take you through the following aspects of programming with SDC, you will see the effects of these principles on your development:

- **Destination administration:** A destination describes one R/3 system to which you want to connect.

- **Generating proxies:** All BAPIs and non-BAPI

[1] Even when running an older version of R/3, you should always use the latest version of the frontend components. They are backward compatible, and each new version usually offers significant improvements over older versions.

[2] RFC stands for Remote Function Call, SAP's protocol for calls between SAP and non-SAP or other SAP systems.

[3] IntelliSense is much easier to use than to explain. When you have typed the name of an object variable and a dot in Visual Basic, for example, a list of all available methods and properties is automatically displayed. Once you select one of the methods, VB will show you its parameters.

RFMs need to be represented by proxies that SDC can generate for you.

- **Building your first application:** Here, I will show you the steps you need to take in order to build a simple BAPI-enabled application:
  - Logging on to R/3
  - Instantiating the proxy classes
  - Calling their BAPIs

  In addition, we will call a non-BAPI RFM.

- **Adding advanced features to your application:** I will conclude this article with some tutorials that describe:
  - How to log on to SAP without using predefined destinations
  - How to use the RFC trace
  - How to use a Visual Logon Component
  - How to use a component that facilitates metadata retrieval

Destination Administration

After installing SDC from the SAP Presentation CD (4.6A for Windows, order # 51007640) on your PC, you should add destinations that describe the R/3 system(s) to which you want your application to connect.[4] All SDC administration is done using the SDC's DHTML frontend, which requires Microsoft Internet Explorer 4.0 or later. You will find a shortcut for the "DCOM Connector" in the "4.6 GUI" folder of the "Programs" section of the Windows "Start" menu.

[4] As we will see later, this is not absolutely required, but normally you would do it.

SDC Architecture

All components of SDC are installed on the Windows PC. The R/3 system itself does not need to be updated at all. SDC supports BAPI calls to R/3 starting with Release 3.1G, and supports non-BAPI RFM calls starting with 3.0C.

The main components of SDC are:

☑ **Destination Manager**

The Destination Manager allows you to define, change, and delete destinations. (Each destination describes one R/3 system.)

☑ **Object Builder**

The Object Builder allows you to generate proxy classes for BAPIs and non-BAPI RFMs.

☑ **Monitor**

The Monitor shows you your current SDC-based R/3 connections and their status.

☑ **Components Viewer**

The Components Viewer shows you all SDC-generated proxy components registered on your PC.

☑ **Runtime Environment**

The only SAP files needed at runtime are librfc32.dll and ccadmin.dll.

Figure 1 *The Main Screen of SDC*

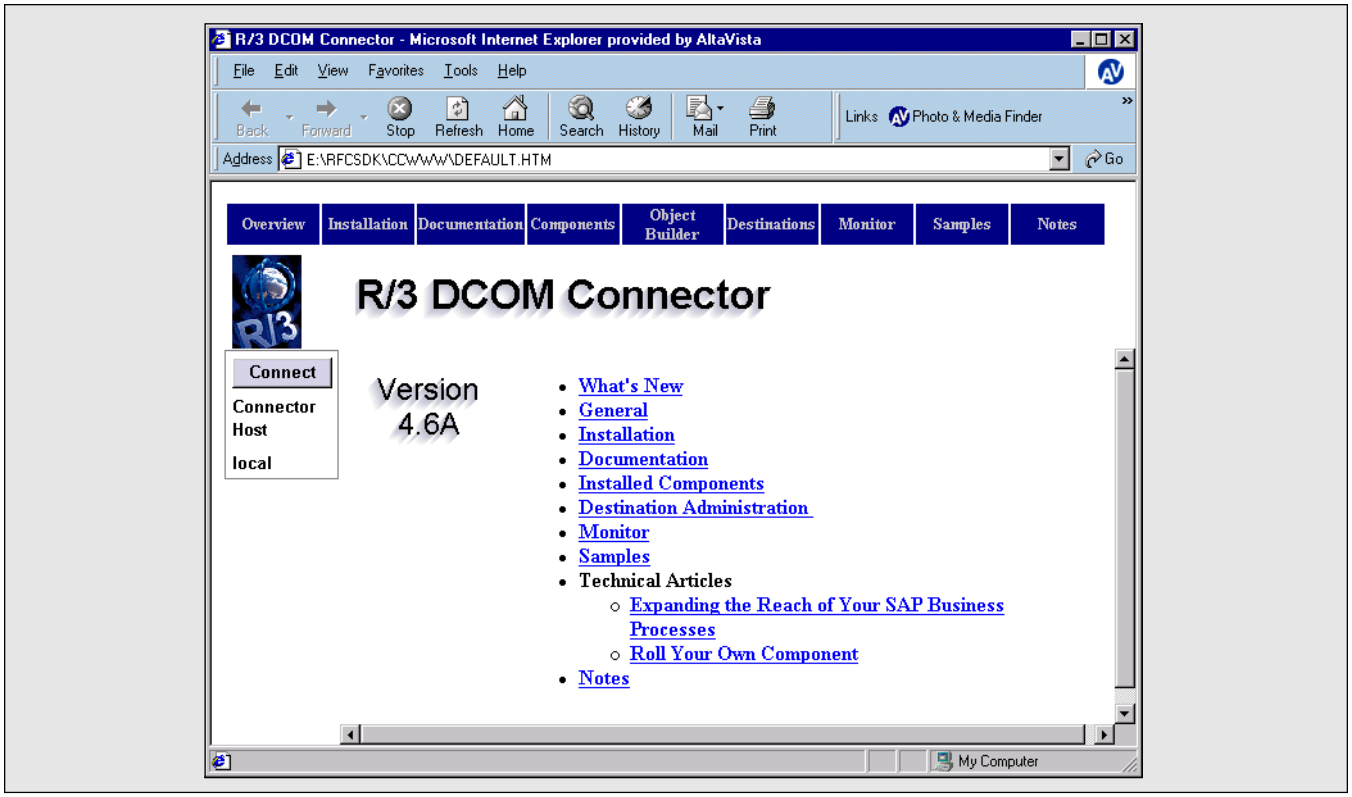


Figure 2 *The List of Destinations*

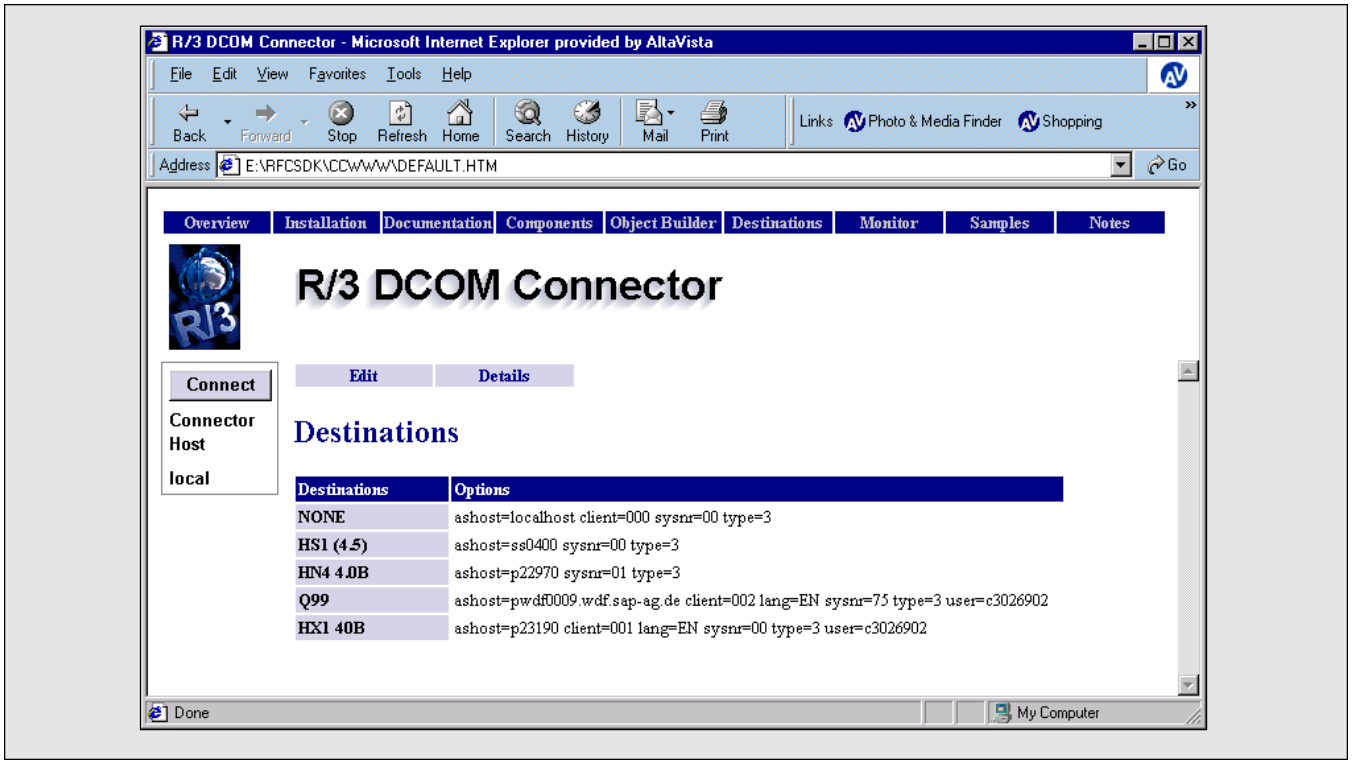
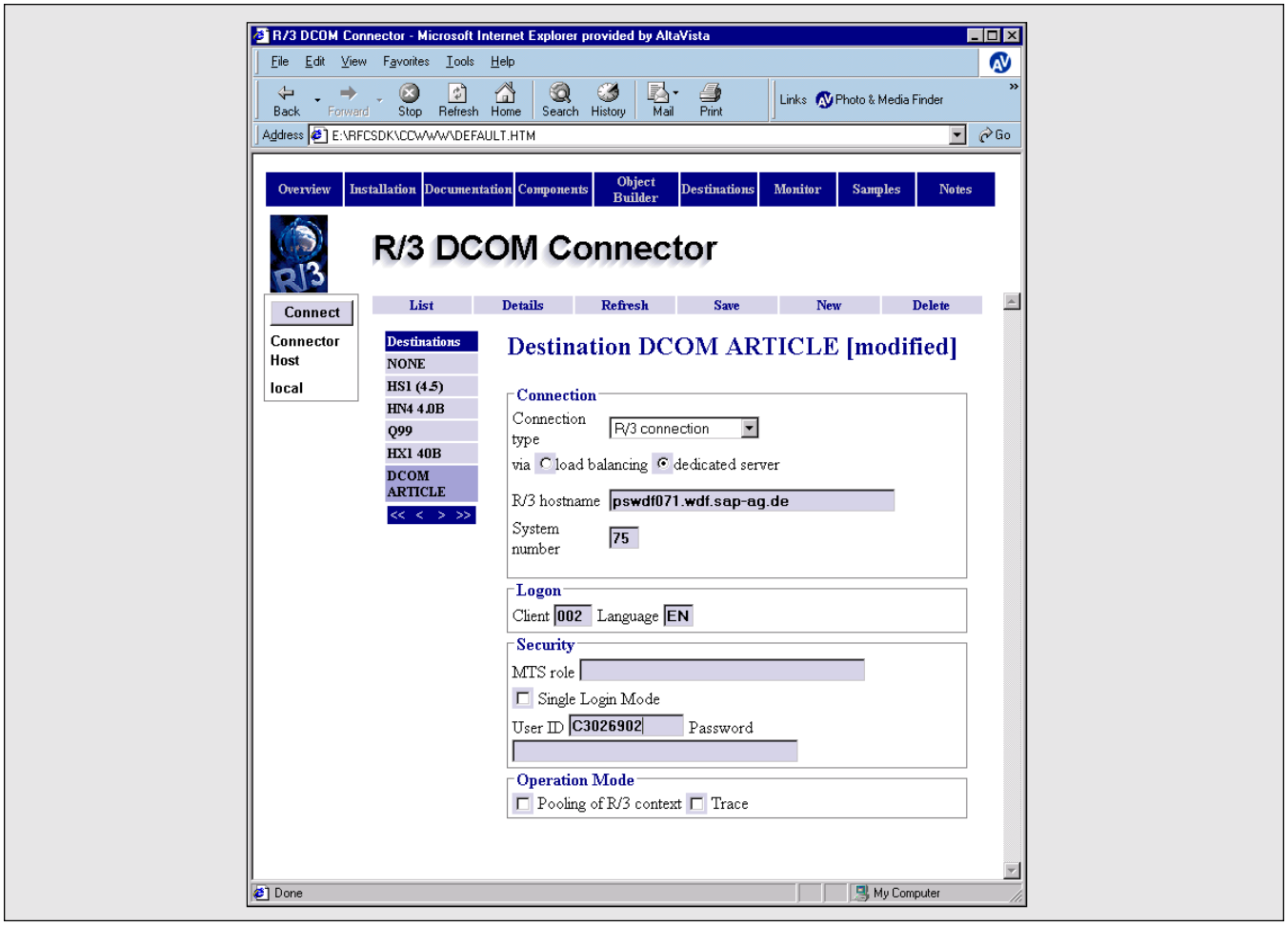


Figure 3 Input Screen for a New Destination



SDC's main screen is shown in **Figure 1**.[5] Our focus here will be on the "Destinations" and "Object Builder" screens. (The other functions are beyond the scope of this article.) To establish a destination, click on the "Destinations" button or the "Destination Administration" link. This will take you to the screen shown in **Figure 2**, which shows you the current list of destinations, along with their defining characteristics.

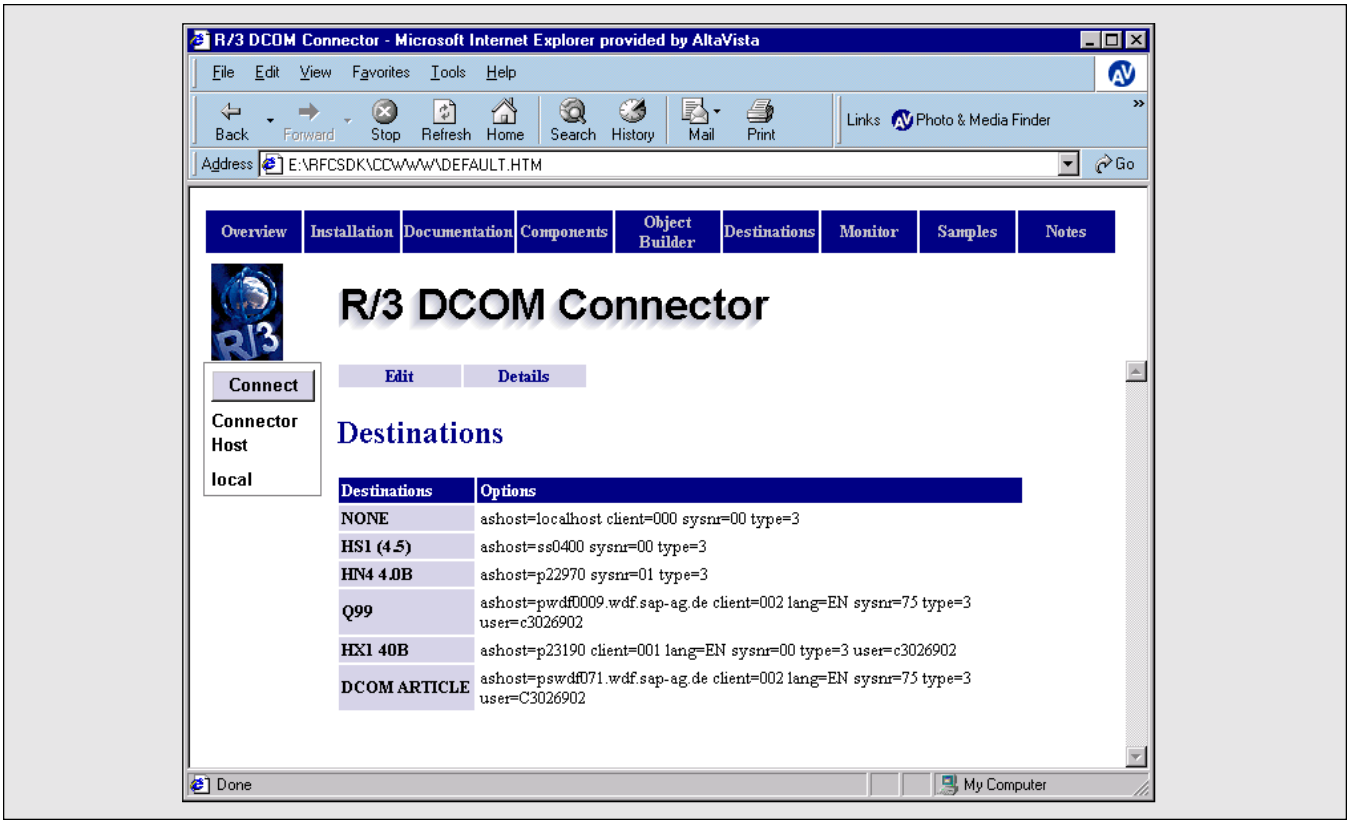
To add a new destination, simply click on the "Edit" button, which takes you to another screen on which you will find a "New" button. Click on that button to get to the input screen shown in **Figure 3**.

[5] As you can see, the online documentation for SDC can be easily reached from this screen. Reading the documentation after reading this article is a good idea.

In this screen, I am defining a new destination called "DCOM ARTICLE". At a minimum, you need to populate the fields in the "Connection" frame information. You can specify an individual application server or a load-balancing group. I chose a dedicated server. If you are undecided which approach to use or do not know the specific information for your R/3 system(s), ask your SAP system administrator for help.

In addition to specifying this minimum system information, you can also specify defaults for the client, language, and userID to be used for this destination. Although you could also enter your password, I would strongly advise you not to do so, unless you have secured your PC so that no one else can ever use it. If you enter a password, it is

Figure 4 The New List of Destinations



encrypted before it is stored. Even so, anyone capable of using your PC could now run any BAPI-enabled program via SDC using your R/3 authorizations.

I will cover the "Trace" option later. For now, I just press the "Save" button and the "OK" button on the subsequent confirmation pop-up window to save the changes. Pressing the "List" button will bring you back to the overview of all defined destinations, which is shown in **Figure 4**. SDC stores the destinations in the Windows Registry.

Generating Proxies

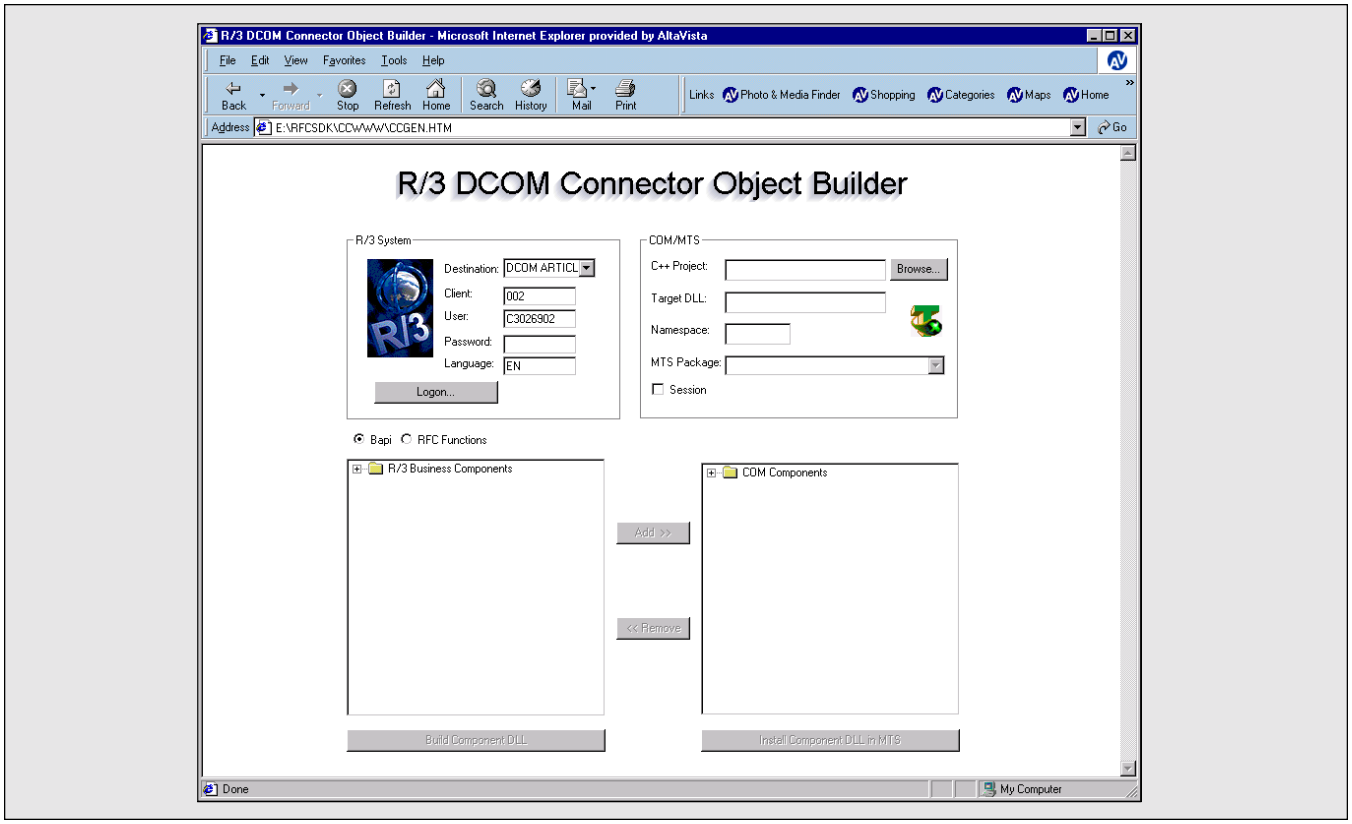
The next step is the generation of proxies for the BAPIs and other RFMs that you want to use in your project(s). These proxies are required so that the BAPIs and non-BAPI RFMs can be accessed in your application like any other COM component.

The proxies generated by SDC follow the SDC architecture design principles listed previously:

- The proxies are lightweight, because SDC generates C++ source code based on the Microsoft Active Template Library (ATL), not on the Microsoft Foundation Classes (MFC) upon which the SAP ActiveX controls are based. (MFC has a massive footprint on the client machine!) The client program talks to R/3 through these proxies. There is no need to know anything about the underlying RFC protocol or to learn the object model of the SAP ActiveX controls. All required metadata is generated into the proxies, eliminating the need for runtime metadata retrieval, and thus improving performance.

- IntelliSense is supported since, as I mentioned earlier, the proxies contain the required Type Library information. The class for a business

Figure 5 The SDC Object Builder Before Logging On



object knows which BAPIs it supports and which parameters each BAPI has. The class for one or more non-BAPI RFMs knows all the functions that are supported and their respective parameters. This makes it much easier to write client programs. Also, the Early Binding supported by the Type Library information improves the performance of your application.

- Most parameters are defined as type Variant. The only exception is mandatory scalar import parameters.[6] Making almost everything a Variant is required so that scripting languages like VBScript can use the proxies.[7] This compromise makes the proxies less convenient in full-featured development environments like Visual Basic, because you cannot tell which data type a specific parameter uses without looking it up in R/3.

- All structure and table parameters are passed as ActiveX Data Objects (ADO) Recordsets. This makes them easy to use, but also has some drawbacks, which we will discuss a bit later.

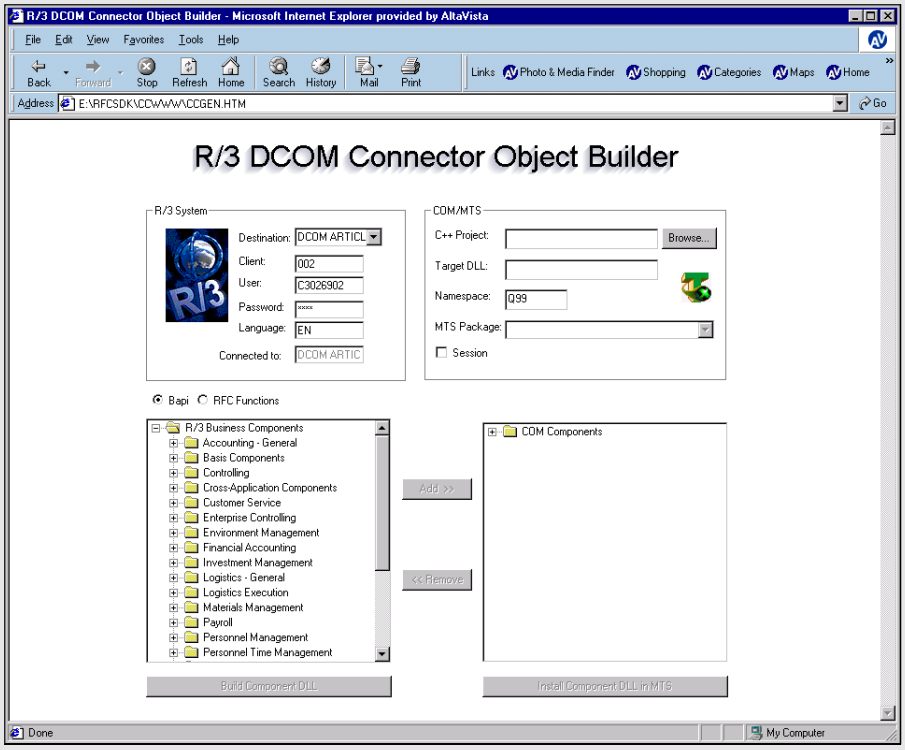
To *generate* proxies, you need Visual C++ installed on at least one computer.[8] To *use* the generated proxies, this is not required. Select the "Object Builder" button in SDC to bring up the screen shown in **Figure 5**. (Depending on your Internet Explorer security settings, you may see a pop-up window that warns you about potentially

[6] Scalar parameters are simple data types like strings, integers, and so on. Import parameters are the parameters that are sent to the BAPI or RFM.

[7] VBScript only uses the Variant data type. Because VBScript is free of charge, Microsoft needed to put some artificial limitations into the product so that people would still pay money for Visual Basic. If you do not like decisions like this, you should use Linux and Java.

[8] I recommend that you use the latest version of all discussed tools (at the time of writing: Visual Studio 6.0 with Service Pack 3 installed). Although you can get some earlier versions to work with SDC, it is not worth the effort.

Figure 6 Displaying the SAP Application Hierarchy



unsafe components. Press the "Yes" button to continue.[9])

Now select a destination, complete the logon information (in this case, it was just a matter of entering my password) and press the "Logon..." button. SDC will then retrieve all the object types containing BAPIs from the destination system and will show the screen in **Figure 6**. (We will discuss non-BAPI RFMs a little bit later.)

All business objects that have BAPIs are displayed, organized according to the application areas to which they belong. You can open the individual application areas and select the business objects you want to put into your proxy component by marking them and then pressing the "Add >>" button. **Figure 7** shows the result of selecting three business objects: CompanyCode, BapiService, and Helpvalues.

[9] Would SAP ever give us unsafe components?

The next step is to specify the names you want to use and whether or not you want a session object included in your proxy component. Which naming standard you want to use is up to you. The consequences for your client code will become obvious later on, when we discuss the details of writing client applications.

In this case, I selected "CompanyCodeProxies" as the C++ project name in the "C:\Generated\" directory and "SAP46B" as the name space. The name of the target DLL is set to the name of the C++ project automatically. The "Target DLL" textbox cannot be edited. (You can later manually rename this DLL if you do not like the default name.) I checked the "Session" option, and changed the name of the class to "SessionComponent". Finally, press the "Build Component DLL" button to generate the C++ source code and automatically invoke the C++ compiler to build the DLL. This is shown in **Figure 8**.

Figure 7 *Selecting Business Objects*

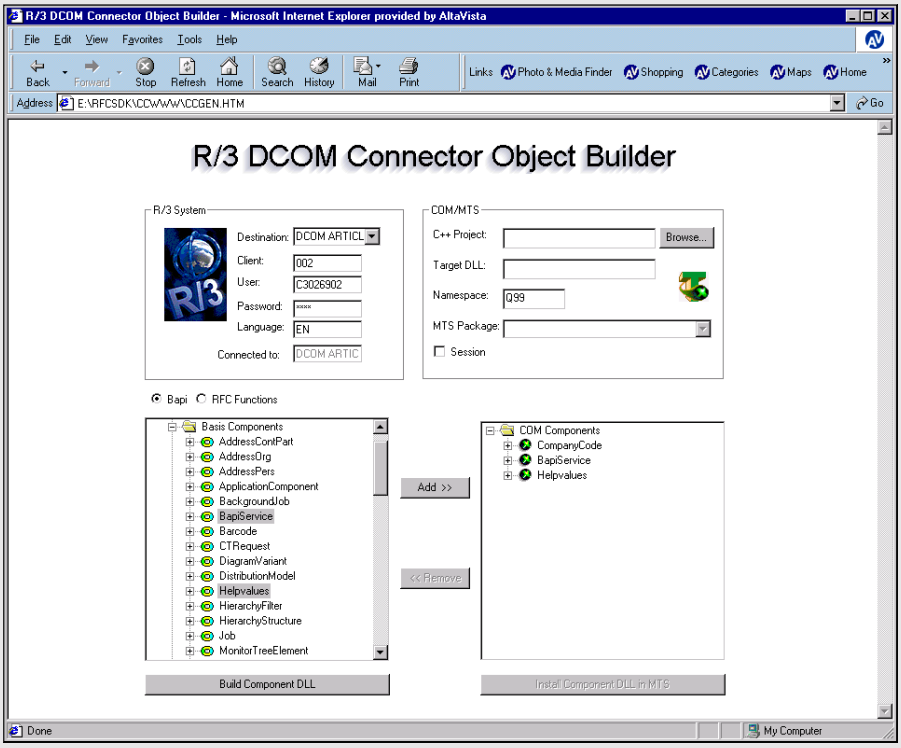


Figure 8 *Generating the BAPI Proxy Component*

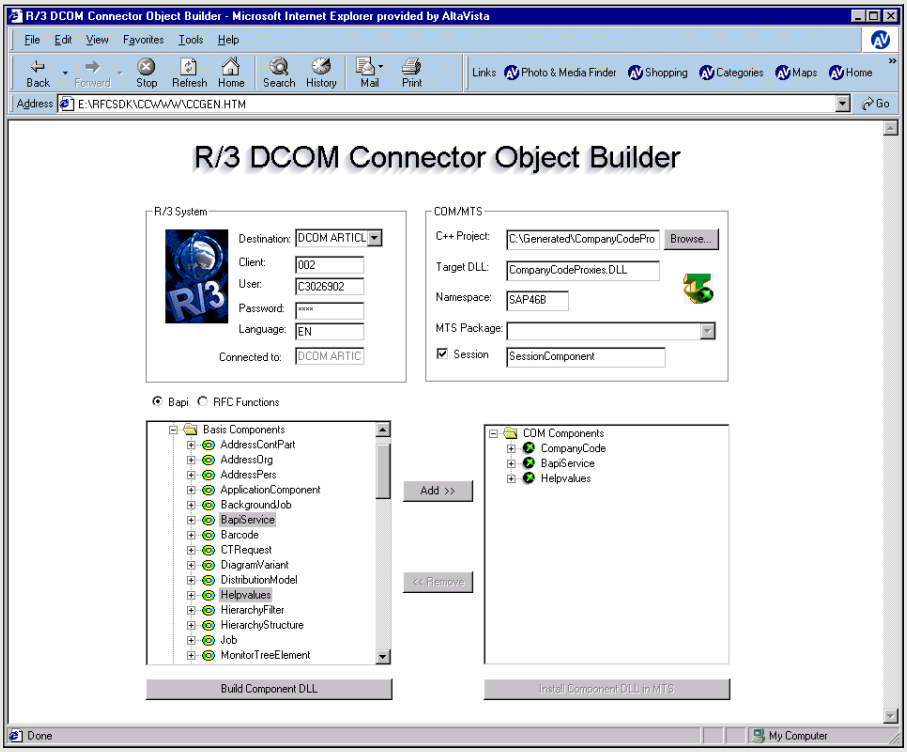
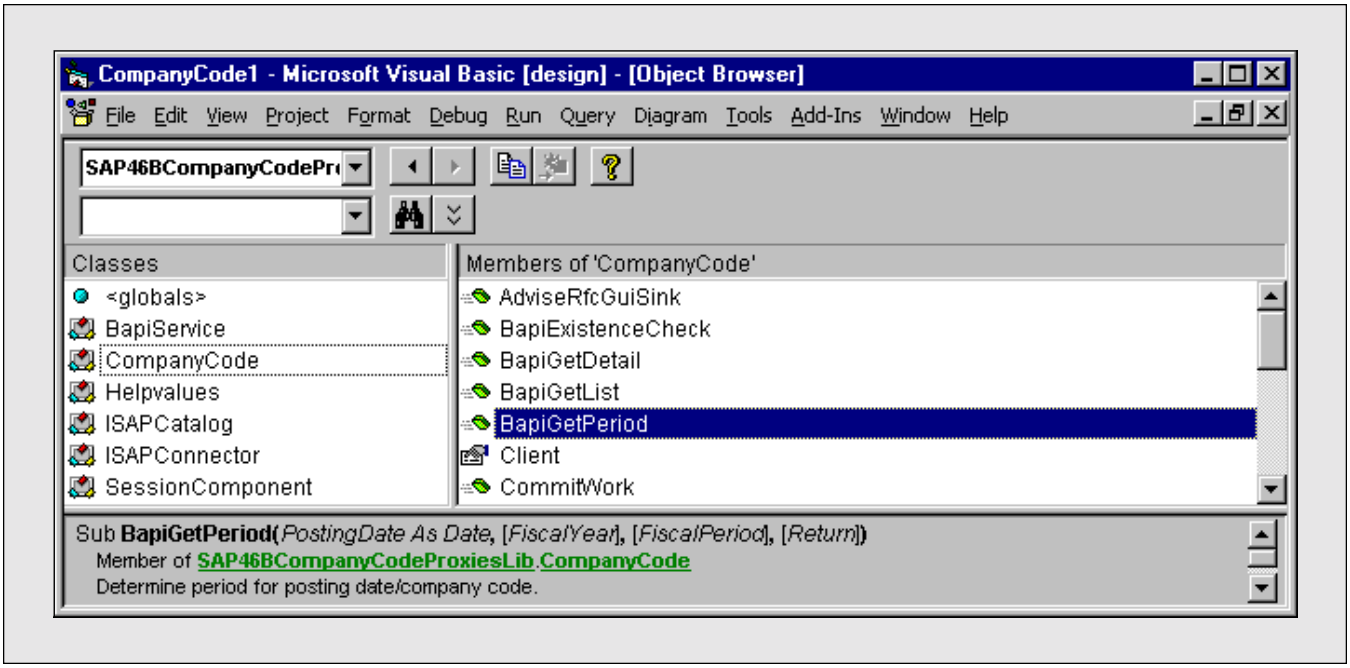


Figure 9 The Proxy Component in the Visual Basic Object Browser



The generated DLL (CompanyCodeProxies.dll) contains the following classes:

- BapiService
- CompanyCode
- Helpvalues
- SessionComponent

The first three classes were included because I selected these three R/3 business objects. The last class, SessionComponent, was included because I checked the "Session" checkbox.

The generated component contains more than just the information used at runtime. It also contains texts that are useful for developers at design time. **Figure 9** shows the Visual Basic Object Browser displaying the GetPeriod BAPI of the CompanyCode object.[10] Note the description of the BAPI at the bottom of the screen.

[10] The fact that the name of the BAPI is prefixed with "Bapi" will be explained later.

In order to avoid potential confusion about the relationship between the R/3 system that is used to retrieve the metadata and the generated component, you need to know that:

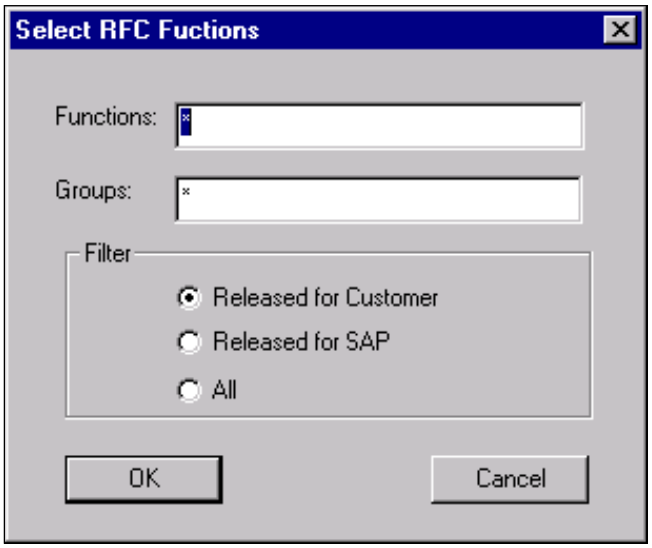
- The metadata of the system to which your application is connected in the Object Builder determines the generated code.
- The generated component can be used to access any R/3 system with the same or a higher release, provided that you use standard SAP BAPIs.[11] You do not have to generate separate components for each R/3 system!
- In cases where you plan to use custom business objects or custom BAPIs, you obviously have to make sure that the R/3 system employed at runtime contains those custom objects and BAPIs.

[11] As long as the BAPIs you use are not obsolete in the higher release. An obsolete BAPI will be supported in at least two functional releases, starting with the one in which it was declared obsolete. For a BAPI that is obsolete since 4.0A, this means that it will be supported until at least 4.5B, but may have disappeared in 4.6A.

Generating Remote Function Modules (RFMs) Components

If you want to use non-BAPI RFMs in your project, you have to build a separate component. After selecting the "RFC Functions" radio button in Figure 8, you will see the RFM selection dialog shown in **Figure 10**.

Figure 10 The RFM Selection Screen



You can limit the selection of RFMs by function name and/or group name. This dialog is not meant to be used as a query tool to find useful RFMs. Use transaction "SE37" in R/3 to find the RFMs you are interested in. This transaction allows you to search by various additional attributes.

Very few non-BAPI RFMs are categorized as "Released for Customer." You should be aware that other RFMs could change in future R/3 releases.[12]

[12] Client code used in production should never call non-BAPI RFMs directly, but rather through a component that encapsulates the various non-BAPI RFMs that a company or a project uses. Only this component will then have to be tested and potentially changed for each new version of R/3. No client code will have to be touched!

✔Tip

Since all BAPIs *are* RFMs, you could theoretically ignore the object-orientation offered by the Business Object Repository (BOR) and reflected in the business object proxy components, and use the BAPIs as normal RFMs.

The only advantage would be that the BAPIs could be combined in one component with all the non-BAPI RFMs you need in a project. However, I believe that the disadvantages of this approach are overwhelming:

- You lose what little object-orientation is supported by Visual Basic.
- You have to pass the key fields of an instance to each RFM that implements an instance BAPI.
- You have to use the parameter names of the RFMs (which sometimes differ from the ones used on the BOR level).
- If you use positional instead of named arguments — as you have to in VBScript, for example — additional optional parameters introduced in a higher release of R/3 could prevent your program from functioning correctly anymore. For a BAPI (used as a method of a business object), the sequence of the parameters will remain the same for all releases (additional parameters will always be inserted at the end). For an RFM, this cannot be guaranteed since the ABAP Workbench does not maintain the required information on the function level, but only on the BAPI level.

Figure 11 Searching for Specific RFMs

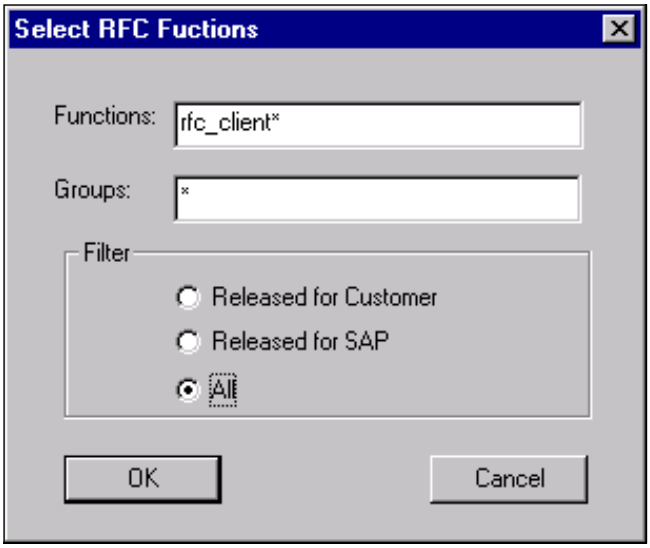
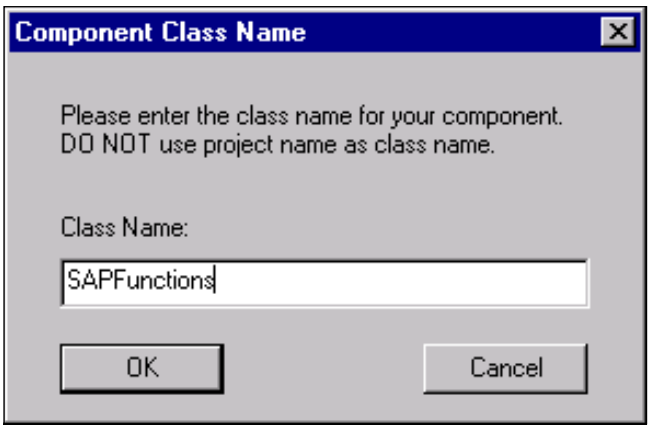


Figure 12 Naming the RFM Proxy Class



The RFM I want to use, "RFC_CLIENT_INFO," is not released, so I have checked "All" in the "Filter" frame shown in **Figure 11**.

Exactly one RFM was found that fulfills the selection criteria. When I mark it and add it to the list of RFMs to be included in the proxy component, the dialog shown in **Figure 12** appears. In this dialog, you enter the name of the class of which the selected RFMs should be methods. You can use any name you want, but I have chosen to call this class "SAPFunctions" in this case.

Back in the main SDC Object Builder screen, you enter a C++ project name and name space for the new component. No session class is needed, since the BAPI component that was generated earlier already contains one. **Figure 13** shows the SDC Object Builder with all required information ready to build the component. Press the "Build Component DLL" button as before to generate the C++ source code and compile it into the target DLL.

Building Your First Application

We are now ready to use the generated components in a simple application. This application will display a list of all company codes and associated texts and allow the user to double-click one company code to see detail information for the selected company code. While this application does not have any socially redeeming value, it is able to demonstrate the important principles of calling BAPIs via SDC.

After starting Visual Basic, we have to add references to several components. At a minimum, we will need the Microsoft ActiveX Data Objects Recordset Library and all proxy components that we want to use in our program. **Figure 14** shows the References dialog in Visual Basic with those references set.

The first programming task facing us now is to connect to R/3. For that purpose, we have to define and instantiate an object of the SessionComponent class that SDC generated into our first component. To define the variable, we can just use the class name, but this might not be unique in our project. In cases where there are multiple classes with the same name, Visual Basic instantiates the class SessionComponent in the component highest in its priority list. In order to avoid potential side-effects of a name ambiguity, we can alternatively define the variable using its complete name, including the Type Library name.

Figure 13 Generating the RFM Proxy Component

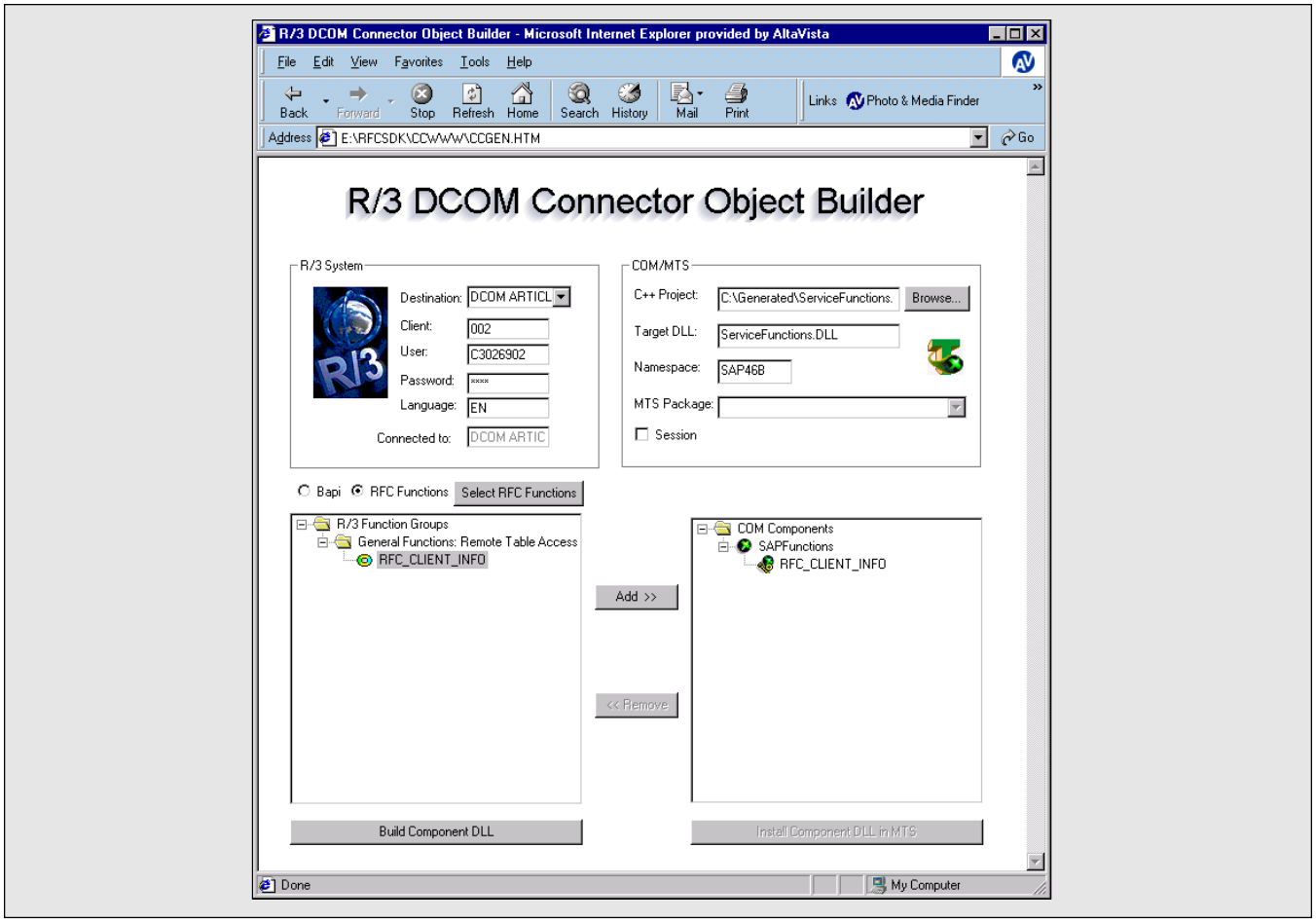
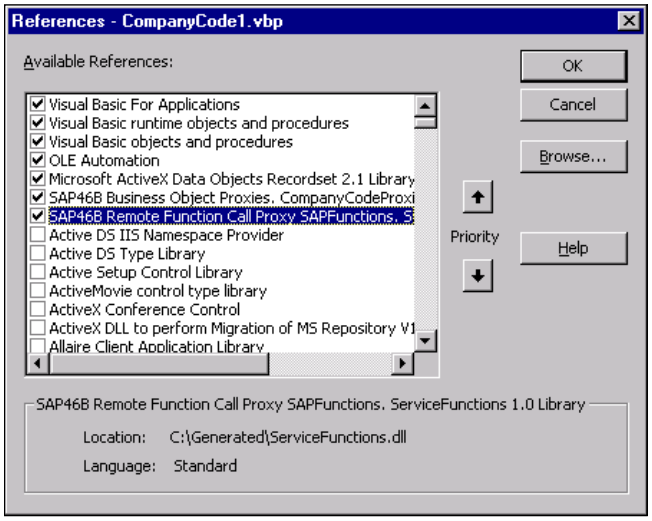


Figure 14 The References Dialog



The choice is yours — here are the pros and cons:

- Using just the class name means less code changes when the proxy component name changes.

- Using the complete name means that you do not depend on the position of a component in Visual Basic's priority list.

If you are sure that your component (Type Library) name will never change, using the complete name seems to be preferable. In this case, you will have to know the name of the Type Library of your proxy component. This name consists of the name space we specified (in our case: "SAP46B"),

followed by the C++ project name (in our case: "CompanyCodeProxies"), followed by "Lib". Our BAPI component's Type Library is called "SAP46BCompanyCodeProxiesLib".[13]

Let us now discuss the other consequence of our earlier naming decisions: The ProgID (Programmatic IDentifier) of any generated proxy class consists of the name space, followed by the period sign, followed by the class name. The ProgID of our CompanyCode proxy class is therefore:

```
SAP46B.CompanyCode
```

We will see later that we have to use the ProgID when we instantiate objects for the business object proxy classes.

If you write software that will have to run with various releases of R/3, it is a good idea to be able to tell which release is the minimum that a generated component will support. I like this information to be visible in two places, the name of the DLL on disk and the ProgID.

- The name of the DLL should reflect the minimum supported release level so that I can easily determine if a given R/3 release is supported (remember: BAPIs are upward, but not necessarily downward compatible).

- The ProgID should contain the release so that no one can install a component containing the same business object for a different release, but with the same ProgID, which would lead to all sorts of strange effects.

You can satisfy the two requirements by making the release a part of the name space specified when you generate the proxy component and manually change the name of the generated DLL to include the release. I, for instance, changed the name of the BAPI proxy DLL from "CompanyCodeProxies.dll" to "46BCompanyCodeProxies.dll". Had I included the release in the C++ project name, the Type Library name would have been "SAP46B46BCompanyCodeProxiesLib", which is uglier than even I can bear.

Let us now start to write the code that allows us to connect to R/3. We will need a SessionComponent object, which we declare and instantiate as shown in **Listing 1**.

Listing 1: The Session Object

```
Private oSession As SessionComponent
Set oSession = New SessionComponent
```

In this case, I have decided to use just the class name for the SessionComponent, since we only use one component that contains this class name in our project. The next step is to specify which system we want to talk to. This is accomplished by using the SessionComponent's PutSessionInfo method. This method has the parameters shown in **Figure 15**.

Figure 15 PutSessionInfo Parameters

Parameter Name	Data Type	Optional
Destination	String	Yes
UserID	String	Yes
Password	String	Yes
Language	String	Yes
Client	String	Yes

While all the parameters are defined as being optional, in reality, you need to pass the Destination parameter. Depending on how much information you put in your destination when you defined it, you may or may not have to pass any of the other parameters. If you followed my advice and left at least the Password empty, you will need to specify the Destination and Password parameters. Of course, you can specify any of the other parameters as well, thus overriding the information stored for the destination. In our program, we will just pass the

[13] If you do not remember the naming decisions you made, you can always look up the name in Visual Basic's Object Browser.

Destination and Password parameters, as shown in **Listing 2**.

Instead of the name of the destination, we could also have passed an explicit string containing all the information required to identify the system and the user. (This will be discussed later in this article.)

The Password will be entered by the user at runtime. Unfortunately, the InputBox function does not allow us to specify that we do not want the entered text to be readable, so in your programs you should use a better version of InputBox or some other dialog to enter the password. (Later in this article, we will discuss a component that encapsulates the whole business of letting a user select a destination and provide user information.)

Now we are ready to actually log on to R/3. We use the Logon method as shown in **Listing 3**.

The Logon method has an optional string parameter that contains an error text if Logon fails. Logon returns a Boolean value indicating whether it succeeded. If not, we show the error text in a message box and exit our subroutine.

Why are we using a variable to store the result of the Logon call? And why is the variable of type Variant, and not Boolean? Why not use the code shown in **Listing 4**?

Well, the 4.6A version of SDC has a bug in the Logon method. Visual Basic expects an integer value of -1 for True, whereas Logon returns positive 1. The result is that the code in Listing 4 will display a message box with no text if Logon succeeds, since

Listing 2: Setting the Session Information

```
Dim sPassword As String
  sPassword = InputBox("Please enter your password.", _
                       App.Title)
  oSession.PutSessionInfo Destination:="DCOM Article", Password:=sPassword
```

Listing 3: Logging on to R/3

```
Dim vLogonSuccessful As Variant
Dim sLogonMessage As String
vLogonSuccessful = oSession.Logon(sLogonMessage)
  If Not vLogonSuccessful Then
    MsgBox sLogonMessage
    Exit Sub
  End If
```

Listing 4: This Does Not Work!

```
Dim sLogonMessage As String
  If Not oSession.Logon(sLogonMessage) Then
    MsgBox sLogonMessage
    Exit Sub
  End If
```

VB interprets the 1 to denote False instead of the intended True. The same problem would occur if we had defined vLogonSuccessful as Boolean in Listing 3. If we use a Variant instead, VB happily interprets the positive 1 as True.[14] SAP is aware of this bug, and it has been fixed in Release 4.6B. For now, we have to live with the work-around shown in Listing 3.

After having logged on successfully, we are ready to call BAPIs. In our sample program, we want to display a list of all company codes. Hence, we will call the CompanyCode.GetList BAPI. The required source code is shown in **Listing 5**.

We use a NAMESPACE constant so that we only have to change one line of code if we ever decide to use a different name space for our generated proxy component. First, we create an instance of the CompanyCode object. This is accomplished by using the SessionComponent's CreateInstance method. Why don't we use the Visual Basic New operator instead? Well, we could do that, but then we would need to tell each proxy object which SAP system it should talk to and each object would run in its own R/3 session. This is a waste of resources and also prevents us from being able to use any BAPI that requires a call to BapiService.TransactionCommit to commit its updates.

The parameter needed for the CreateInstance method is the ProgID of our generated proxy class, which consists of the name space, a period sign, and the business object name in R/3.

Next, we actually call the BAPI. Note that the BAPI name that you see in R/3 (GetList) is prefixed with "Bapi" here. The SDC Object Builder prefixes all BAPIs in this way. This ensures that there can never be a conflict with the restricted keywords in Visual Basic or the names reserved by COM. The parameters are passed as named arguments. Alternatively, you could use positional arguments, but your source code would be harder to read. All structure and table parameters of a BAPI are treated as ActiveX Data Objects (ADO) Recordsets by SDC. Therefore, we defined both the CompanyCodeList (table) and Return (structure) parameters as Recordsets.

Now would be a good time to check the return code of the BAPI, as shown in **Listing 6**.

The TYPE field in the return structure is empty, or it contains "S" to indicate a successful call, or "I" if an informational message is returned. For our purposes, we will treat informational messages as an indication of a successful execution of the BAPI. In any other case, we will display the error type, error code, and error message text in a message box.

What remains to be done is to display the list of company codes. Since it is returned as a Recordset, we can take advantage of the ADO-aware controls delivered with Visual Basic 6. We have put a DataGrid control on our main form, and we specify its data source, as shown in **Listing 7**.

The last line is the most important one. It connects the Recordset containing the company codes and texts with the DataGrid1 variable. This will cause the Recordset's contents (all rows and all columns) to be displayed in the DataGrid. Any changes to the Recordset will automatically be reflected in the GUI. Any changes made by the user will automatically be reflected in the connected Recordset.[15] ADO really is a cool concept!

[14] This is probably an unintended inconsistency in the conversion functions used in VB.

[15] We prevent the user from making changes in this case by setting AllowUpdate to False.

Listing 5: CompanyCode.GetList

```
Private Const NAMESPACE As String = "SAP46B"
Private oCompanyCode As CompanyCode
Private rsList As Recordset
Private rsReturn As Recordset
  Set oCompanyCode = _
    oSession.CreateInstance(NAMESPACE & ".CompanyCode")
  oCompanyCode.BapiGetList CompanyCodeList:=rsList, _
                           Return:=rsReturn
```

Listing 6: Checking the Return Code

```
If Not (rsReturn("TYPE") = "" Or _
        rsReturn("TYPE") = "S" Or _
        rsReturn("TYPE") = "I") Then
  MsgBox rsReturn("TYPE") & rsReturn("CODE") & ": " & _
         rsReturn("MESSAGE")
  Exit Sub
End If
```

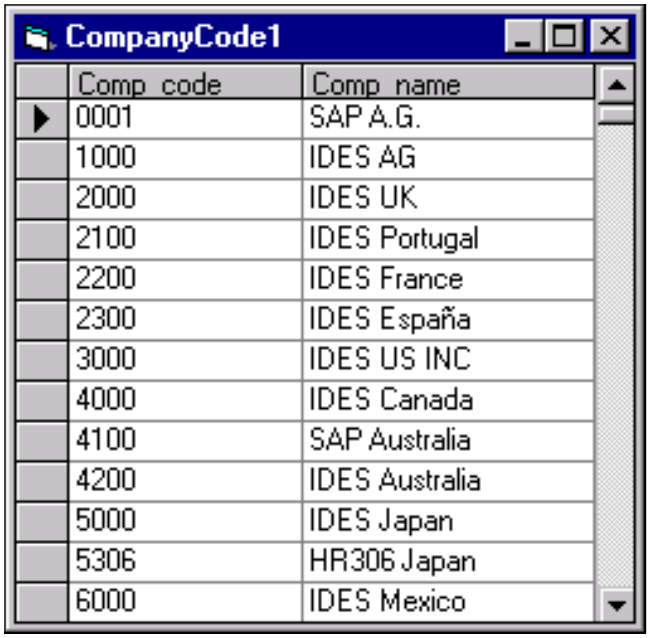
Listing 7: Displaying the Company Codes

```
DataGrid1.AllowUpdate = False
If rsList Is Nothing Then
  MsgBox "No company codes defined."
  Exit Sub
End If
Set DataGrid1.DataSource = rsList
```

The first line in Listing 7 turns DataGrid1 into a read-only display. We do not want the user to change the displayed information in this application. The If statement that follows may surprise you. Why do we check whether rsList is Nothing? We checked the BAPI's return code before so we should assume that all is well?! Unfortunately, SAP decided that empty tables are not returned as empty Recordsets, but as Nothing instead. Therefore, unless we are absolutely sure that a table can never be empty, we always have to check the Recordset variable representing a table first.

This concludes the part of our application that displays all company codes. A screen shot of this

Figure 16 The Result of CompanyCode.GetList Displayed in a DataGrid



program in action is shown in **Figure 16**.

Note that the column headers contain the R/3 field names that are used in the structure defined for the CompanyCodeList parameter. This is not the best solution. Instead, you can either set the header texts in some hard-coded fashion or retrieve suitable texts from R/3, as discussed below.

Calling an Instance Method

We now want to add the capability to display more information about an individual company code when the user double-clicks a row of the grid. The source code for this is shown in **Listing 8**.

CompanyCode.GetDetail is a so-called instance method. It requires us to identify a specific object in the R/3 relational database by supplying all key fields completely. The CompanyCode object has just one key field, called CompanyCodeId. This key field (prefixed by "Key" by the SDC Object Builder)

Listing 8: Displaying Detail Information

```
Private Sub DataGrid1_DblClick()
Dim rsDetail As Recordset
  oCompanyCode.InitKeys rsList("COMP_CODE")
  oCompanyCode.BapiGetDetail CompanycodeDetail:=rsDetail, _
                             Return:=rsReturn
  If Not (rsReturn("TYPE") = "" Or _
          rsReturn("TYPE") = "S" Or _
          rsReturn("TYPE") = "I" Or _
          (rsReturn("TYPE") = "W" And _
           rsReturn("CODE") = "FN021")) Then
    MsgBox rsReturn("TYPE") & rsReturn("CODE") & ": " & _
           rsReturn("MESSAGE")
    Exit Sub
  End If
  MsgBox "The language code for company code " & _
         oCompanyCode.KeyCompanyCodeId & " is " & _
         rsDetail("LANGU") & "."
End Sub
```


is a read-only property of the proxy class for CompanyCode. To set the key fields for an object we use the proxy class's InitKeys method. The InitKeys method has as many parameters as the business object has key fields. In our example, there is only one key field, hence only one parameter. We pass the COMP_CODE field of the current (double-clicked) record in our Recordset.

Now we can call the GetDetail BAPI. Note that the name of the BAPI needs to be prefixed with "Bapi" again.

The error checking is a little more work this time: CompanyCode.GetDetail has three parameters, two of which are of interest to our application. The third one is called CompanycodeAddress. Once in a while (especially in development or test systems) you will find a company code where GetDetail returns the warning (message type "W") FN021. This message indicates that the address data is not completely maintained for this company code. In our application, we are not concerned with this, since we do not even use this parameter. The BAPI cannot know this and therefore warns us. We decide to ignore this particular warning, since we do not need the address data structure in our application.

If all went well, we now display the company code (available through the KeyCompanyCodeId property) and the associated language code (a field in the CompanycodeDetail structure) in a message box.

Creating SAP Recordsets

The BAPIs we have used so far (CompanyCode.GetList and CompanyCode.GetDetail) did not require us to fill in any structure or table parameters before calling the BAPI. R/3 and SDC cooperated to create the tables and structures that were returned to us. How do you create a Recordset for a table or structure parameter if you need to provide data in it? You could, of course, create a Recordset manually in your program adding all the fields with their attributes yourself. This would be a lot of work and would make your program release-dependent, since additional fields added at the end of the structure in a future release would not be created in your program.

The better answer is the DimAs method that every business object proxy class supports. The parameters for this method are shown in **Figure 17**.

Figure 17 DimAs Parameters

Parameter Name	Data Type	Optional
Method	String	No
Parameter	String	No
pRS	Variant	No

The Method parameter needs to be set to the name of the BAPI that contains the structure or table parameter we want to create, including the "Bapi" prefix SDC uses. Parameter contains the name of the parameter for which the structure or table is to be used. Both of these parameters are case-sensitive.

The pRS parameter is the variable name of the newly created Recordset. **Listing 9** contains the source code to create the Recordset for the CompanyCodeList table parameter of the CompanyCode.GetList BAPI.

In our application, this call is not required, since this table is returned by R/3, and it does not make

Listing 9: Creating a Recordset

```
Call oCompanyCode.DimAs("BapiGetList", "CompanyCodeList", rsList)
```

sense to fill in a table that we call a BAPI to retrieve in the first place. But now you know how to create Recordsets in applications where you need them.

Calling a Non-BAPI RFM

We will now learn how to call non-BAPI RFMs. A little earlier we created a proxy component that includes support for the RFC_CLIENT_INFO RFM. This function has the three parameters shown in **Figure 18**.

Figure 18 Parameters for RFC_CLIENT_INFO

Parameter Name	Data Type	Optional
Hostid	Variant	Yes
Hostname	Variant	Yes
Sysid	Variant	Yes

Hostid is the system number for the R/3 system. *Hostname* is the name of the application server. *Sysid* is the identification of the R/3 system.

Listing 10 shows the source code to call the function and display the returned host name as part of the form's caption.

The call to the CreateInstance method must specify the correct ProgID. For a BAPI, we used the name space, a period sign, and the name of the business object. Since non-BAPI RFMs do not belong to business objects, a pseudo-object is used. The name of this pseudo-object ("SAPFunctions") is the one we entered when we generated the RFM proxy component back in Figure 12).

The SDC Object Builder does not prefix non-BAPI RFMs with any string to prevent a potential name conflict.[16] Therefore, we just use "Rfc_Client_Info" instead of "Rfm_Rfc_Client_Info", for example.

Summary of the CompanyCode Sample Program

We have done it! Our modest application is ready. We have logged on to R/3, called class and instance BAPIs, and called a non-BAPI RFM. To facilitate a review of the source code, **Listing 11** contains the complete code.

Earlier in this article, I talked about the fact that the Type Library generated into the proxy components allows us to view the BAPIs and their parameters in the VB Object Browser, and to use IntelliSense when writing source code. This does not apply to the fields in the Recordsets, though. The field names are not part of the Type Library and must be typed correctly (from memory or using Copy/Paste from R/3). No IntelliSense is available for the field names. This is a consequence of SAP's decision to

[16] This may change in a future release.

Listing 10: Calling an RFM

```
Dim oFunctions As SAPFunctions
Dim sHostName As String
  Set oFunctions = oSession.CreateInstance(NAMESPACE & ".SAPFunctions")
  oFunctions.Rfc_Client_Info HostName:=sHostName
  Caption = App.Title & ", Host Name: " & sHostName
```


Listing 11: The Complete Application Code Listing

```
Option Explicit

Private Const NAMESPACE As String = "SAP46B"

Private oCompanyCode As CompanyCode
Private oSession As SessionComponent
Private oFunctions As SAPFunctions

Private rsList As Recordset
Private rsReturn As Recordset

Private Sub DataGrid1_DblClick()
Dim rsDetail As Recordset
  oCompanyCode.InitKeys rsList("COMP_CODE")
  oCompanyCode.BapiGetDetail CompanycodeDetail:=rsDetail, _
                             Return:=rsReturn
  If Not (rsReturn("TYPE") = "" Or _
          rsReturn("TYPE") = "S" Or _
          rsReturn("TYPE") = "I" Or _
          (rsReturn("TYPE") = "W" And _
           rsReturn("CODE") = "FN021")) Then
    MsgBox rsReturn("TYPE") & rsReturn("CODE") & ": " & _
           rsReturn("MESSAGE")
    Exit Sub
  End If
  MsgBox "The language code for company code " & _
         oCompanyCode.KeyCompanyCodeId & " is " & _
         rsDetail("LANGU") & "."
End Sub

Private Sub Form_Load()
Dim vLogonSuccessful As Variant
Dim sLogonMessage As String
Dim sPassword As String
Dim sHostName As String
  DataGrid1.AllowUpdate = False
  Set oSession = New SessionComponent
  sPassword = InputBox("Please enter your password.", _
                       App.Title)
  oSession.PutSessionInfo Destination:="DCOM Article", _
                          Password:=sPassword
  vLogonSuccessful = oSession.Logon(sLogonMessage)
  If Not vLogonSuccessful Then
    MsgBox sLogonMessage
    Exit Sub
  End If
  Set oFunctions = oSession.CreateInstance(NAMESPACE & ".SAPFunctions")
  oFunctions.Rfc_Client_Info HostName:=sHostName
  Caption = App.Title & ", Host Name: " & sHostName
  Set oCompanyCode = _
    oSession.CreateInstance(NAMESPACE & ".CompanyCode")
  Call oCompanyCode.DimAs("BapiGetList", "CompanyCodeList", rsList)
```

```
  oCompanyCode.BapiGetList CompanyCodeList:=rsList, _
                           Return:=rsReturn
  If Not (rsReturn("TYPE") = "" Or _
          rsReturn("TYPE") = "S" Or _
          rsReturn("TYPE") = "I") Then
    MsgBox rsReturn("TYPE") & rsReturn("CODE") & ": " & _
           rsReturn("MESSAGE")
    Exit Sub
  End If
  If rsList Is Nothing Then
    MsgBox "No company codes defined."
    Exit Sub
  End If
  Set DataGrid1.DataSource = rsList
End Sub

Private Sub Form_Resize()
  DataGrid1.Move 0, 0, ScaleWidth, ScaleHeight
End Sub
```

represent all structure and table parameters as standard ADO Recordsets.[17]

Do you think that something is still missing from this program? When do we ever log off?

[17] As a little experiment, I wrote a generator that builds a component containing Type Library information for all structure and table parameters used by any BAPI. Using the generated component, one can now use IntelliSense during development with very little runtime overhead. Here is some sample code showing the use of this component:

```
Private StructureFactory As StructureFactory
Private rsList As Recordset
Private rsReturn As Recordset
Private oList As BAPI0002_1
Private oReturn As BAPIRETURN
  Set StructureFactory = New StructureFactory
  Set oCompanyCode = _
    oSession.CreateInstance(NAMESPACE & _
           ".CompanyCode")
  oCompanyCode.BapiGetList CompanyCodeList:=rsList, _
                           Return:=rsReturn
  Set oReturn = _
    StructureFactory.GetBAPIRETURN(rsReturn)
  If Not (oReturn.TTYPE = "" Or _
          oReturn.TTYPE = "S" Or _
          oReturn.TTYPE = "I") Then
    MsgBox oReturn.TTYPE & oReturn.CODE & ": " & _
           oReturn.MESSAGE
    Exit Sub
  End If
```

Given enough positive feedback, I might be tempted to make this generator publicly available!

Well, we do not explicitly log off from R/3. SDC's SessionComponent class does not even have a Logoff method. The session with R/3 is automatically terminated when:

- The session object we use has no more references to it. We would accomplish this by setting it to Nothing in our code.
- R/3 closes the connection because it has been inactive longer than the system parameter that controls this type of activity allows.

Advanced Topics

This concludes the discussion of the common tasks involved in writing BAPI-enabled applications for the SDC using Visual Basic. The remainder of this article will cover a few advanced topics.

Logon Without Predefined Destinations

Earlier I mentioned that you did not have to define destinations. How can we log on to a system that is

not defined as a destination? In our sample program, we set the Destination parameter of the PutSessionInfo method to the name of a destination that we had defined previously.

As an alternative, we could pass a string containing all required information surrounded by curly braces. The format of this string is the same as that which is shown in Figure 4. **Figure 19** shows the list of keywords that can be used in the connection string.

Figure 19 Connection String Keywords

Keyword	Description
ashost	Application server name
sysnr	System number
mshost	Message server name
group	Load balancing group
r3name	R/3 system name
type	System type (2 = R/2[18], 3 = R/3)
client	Client number
lang	Language code
user	UserID
passwd	Password
trace	Trace option (0 = trace off, 1= trace on)

[18] Yes, you can even use SDC to communicate with SAP's mainframe product, R/2. There are no BAPIs in R/2, but quite a few RFMs.

A system can be identified by either using "ashost" and "sysnr" (for a specific application server) or "mshost", "group", and "r3name" (for a load-balancing group). These two sets of keywords cannot be combined. The "trace" and "lang" keywords do not have to be specified. Trace defaults to off. Language defaults to the standard language set for the particular R/3 system.

Listing 12 shows the code to set the session information using an explicit connect string.

We can still use the other parameters of the PutSessionInfo method, so an alternative could be the code in **Listing 13**.

Using the Trace

In both Listing 12 and 13 we have turned the trace on. This will cause the RFC layer to write a complete trace of all our activities to a file called rfcxxxxx_yyyyy.trc, where xxxxx and yyyyy are dynamically generated numbers to guarantee that each trace is written to a different file. Most problems in our applications will not require us to use a trace, but at least you should know how it is turned on in case you ever need it.

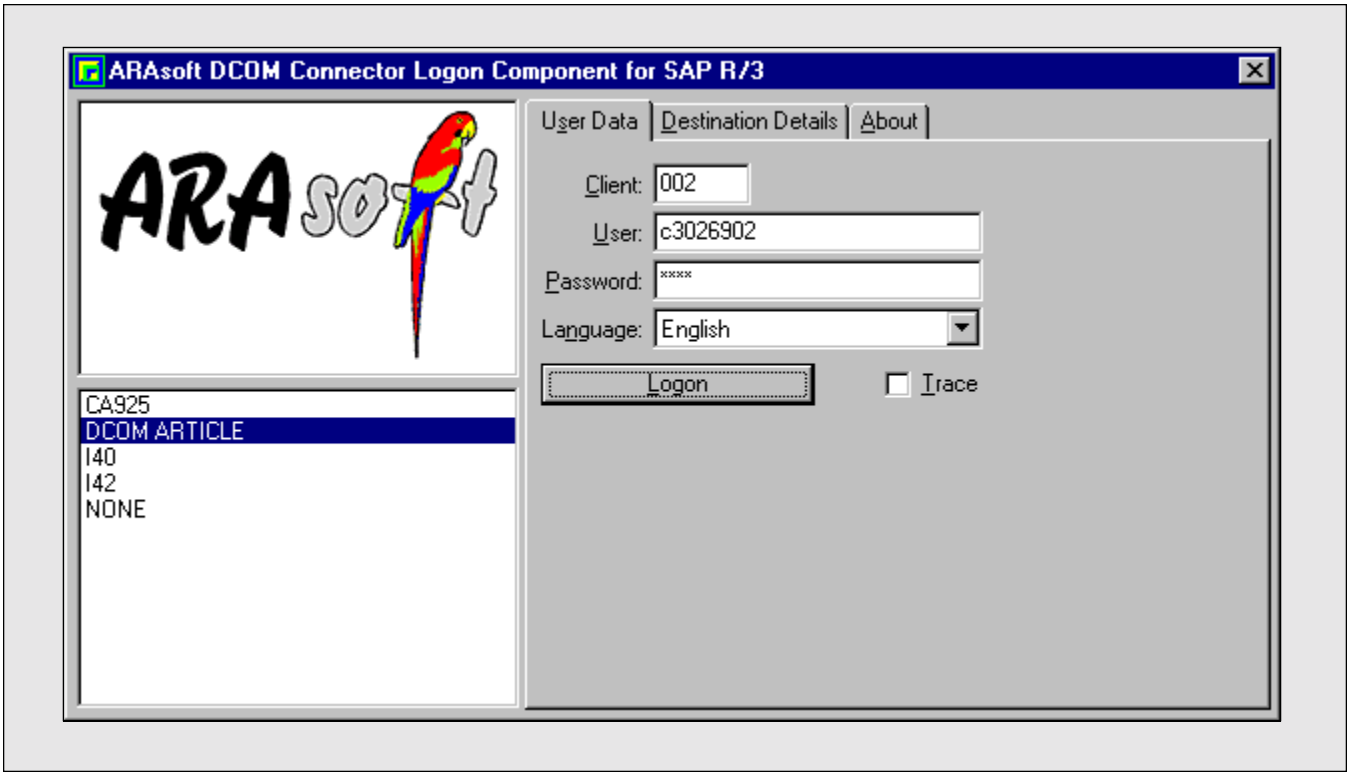
Listing 12: Setting Session Information Without Using a Predefined Destination

```
Dim sConnectString As String
  sConnectString = _
    "{ ashost=pswdf071.wdf.sap-ag.de client=002 lang=EN sysnr=75 type=3
user=c3026902 passwd=*** trace=1 }"
  oSession.PutSessionInfo Destination:=sConnectString
```

Listing 13: Alternative Version for Listing 12

```
Dim sConnectString As String
  sConnectString = _
    "{ ashost pswdf071.wdf.sap-ag.de sysnr=75 type=3 trace=1 }"
  oSession.PutSessionInfo Destination:=sConnectString, _
                          UserID:="c3026902", _
                          Password:="***", _
                          Language:="EN", _
                          Client:="002"
```

Figure 20 The ARAsoft DCOM Connector Logon Component



The trace can be turned on for a specific program as shown in Listings 12 and 13. To turn it on for all programs that access a destination, you can check the Trace option of the particular destination on the destination administration screen of SDC (Figure 3).

Unfortunately, the PutSessionInfo method has no Trace parameter. Therefore, we cannot turn on the trace for a specific program execution unless we use the explicit connection string shown in Listings 12 and 13. This will hopefully be amended in a future release.

Using a Visual Logon Component

Hard-coding the destination into our application, as we did so far, is clearly not adequate for most programs. Often we want to allow our user to select a system and language and force him to type at least the password, even if all the other user information is stored in the destination entry. Since Walldorf does not currently offer a Visual Logon Component for SDC, I decided to build my own, which you can download from **http://www.sap.com/products/techno/bapis/com/ara_dcom_2.zip**

This component is freeware. A screen shot of the component in action is shown in **Figure 20**.

The user does not have to remember the various language codes. Instead, there is a dropdown list which contains the names of the languages in the languages themselves, so anyone can easily find their own language in the list, instead of having to know the English translation, or worse yet, the ISO code for the language.

Listing 14 contains sample code exemplifying the use of the component in an application.

Listing 14: Using the Logon Component

```
Dim sErrorMessage As String
Dim ldSuccess As ARAsoftDLC.LOGONDIALOGRESULT
  ldSuccess = ARAsoftDLC.LogonDialog(oSession, sErrorMessage)
  If ldSuccess = Success Then
    MsgBox "Okay"
  ElseIf ldSuccess = Failure Then
    MsgBox sErrorMessage
  ElseIf ldSuccess = Cancel Then
    ' User cancelled the logon dialog
  End If
```

Listing 15: Using Meaningful Column Headers

```
Private oBof As ARAsoftBOF.ObjectFactory
Set oBof = New ARAsoftBOF.ObjectFactory
  Set oBof.Session = oSession
  DataGrid1.Columns("COMP_CODE").Caption = _
    oBof.GetStructures("BAPI0002_1").Fields("COMP_CODE").LabelLong
  DataGrid1.Columns("COMP_NAME").Caption = _
    oBof.GetStructures("BAPI0002_1").Fields("COMP_NAME").LabelLong
```

Metadata Retrieval

As you saw in Figure 16, the default header text used for a column of a Recordset is the SAP field name. This is not a good solution in real applications, where the users expect useful header texts in their own language. One way to handle this challenge would be to hard-code texts for the headers in our application, but this would not be very flexible.

Instead, we can take advantage of all the texts stored in R/3. Using the DDIF_FIELDINFO_GET function, we can retrieve the texts and other metadata for any structure or table defined in R/3.

I have built a component that encapsulates the access to this RFM. **Listing 15** shows the source code that allows the application to dynamically retrieve the texts for the two columns of the table in the user's logon language.

We access the Structures collection offered by the ObjectFactory using the name of the structure behind the CompanyCodeList parameter ("BAPI0002_1"), then use the long label text of the two fields.

Figure 21 is a screen shot of the enhanced company code program, with French used as the logon language.

Figure 21 Enhanced Company Code Program

Summary

The SAP DCOM Connector is the recommended middleware for building BAPI-enabled applications for Microsoft platforms. The latest release of SDC should be used even if the R/3 system itself uses an earlier release.

The proxy classes generated by SDC facilitate the developer's job and improve runtime performance. Building components to be able to reuse functionality is always a good idea. Some suggestions for SDC-related components were discussed in the Advanced Topics section of this article. Many more ideas will present themselves to you when you start to build your own solutions. Enjoy!

Thomas G. Schuessler is the founder of ARAsoft, a Germany-based company offering products, consulting, custom development, and training in the area of integration with R/3 worldwide. Thomas is the author of SAP's CA925 "Programming with BAPIs in Visual Basic" class, which he teaches in Germany and in English-speaking countries. His book on the same subject will be published in English and German by Addison Wesley Longman in early 2000. Thomas has spoken at many SAP TechEd and SAPPHIRE events. Prior to founding ARAsoft in 1993, he worked with SAP AG and SAP America for seven years. Thomas can be contacted at thomas.schuessler@sap.com or at arasoft@t-online.de.

