SAP .NET Connector for C# Programmers

Thomas G. Schuessler

Thomas G. Schuessler is the founder of ARAsoft, a company offering products, consulting, custom development, and training to customers worldwide, specializing in integration between SAP and non-SAP components and applications. Thomas is the author of SAP's BIT525 and BIT526 classes. Prior to founding ARAsoft in 1993, he worked with SAP AG and SAP America for seven years.

(complete bio appears on page 104)

"Would you tell me, please, which way I ought to go from here?"
"That depends a good deal on where you want to get to"
– Lewis Carroll, Alice's Adventures in Wonderland

In the past, there were four main alternatives for writing applications that communicate with ABAP-based SAP components like R/3: the low-level Remote Function Call (RFC) Library; the SAP Java Connector (JCo) for platform-independent development; and, for Windows-based development, the SAP ActiveX Controls and the SAP DCOM Connector. Now SAP has released a new product for the Microsoft platform, the SAP .NET Connector.

There are several reasons why developers who have used the SAP ActiveX Controls and the SAP DCOM Connector should learn about the SAP .NET Connector:

- .NET is a strategic Microsoft technology.
- The SAP .NET Connector is fully integrated with Microsoft Visual Studio .NET, making it easier than ever before to build Microsoft-specific solutions that communicate with SAP.
- The SAP ActiveX Controls and the SAP DCOM Connector have several limitations that do not apply to the SAP .NET Connector.
- The SAP DCOM Connector will only be supported until December 31st, 2004 (see SAP Note 533055 for details).

This article will introduce you to the SAP .NET Connector using C#[1] as the programming language for the sample code, but you can use any .NET-compliant language for your own applications. A working knowledge of Visual Studio .NET is assumed.

Before we look at the SAP .NET Connector proper, let me remind you of a few basics.

The RFC Library

RFC is a bidirectional protocol for communication between SAP and other SAP or non-SAP components. Non-SAP components use the RFC Library, which is available for all relevant SAP-supported platforms, to "speak" the protocol.

While it is possible to use this library directly in your application (which would then probably be written in C or C++), it is much easier to use a higher-level middleware like the SAP .NET Connector or JCo.

RFMs and BAPIs

There are more than 10,000 RFC-enabled Function Modules (RFMs) in R/3 4.6C, about 2,000 of which are BAPIs.

An RFM can have parameters and exceptions. The parameters can be subdivided as follows:

- **Import parameters:** These can be simple fields (scalars) or structures (groups of fields). Very recent releases of SAP allow even tables as import parameters. Import parameters can be mandatory or optional. An optional scalar import parameter can have a default value defined.
- **Export parameters:** Like import parameters, except that they are all optional (but do not have default values).
- **Table parameters:** These are intelligent arrays known as internal tables in ABAP.
- **Changing parameters:** This is similar to using the same parameter as an import and export parameter.

BAPIs are RFMs with an object-oriented twist. They are defined as methods of business object types in the Business Object Repository (BOR). BAPIs are invoked like other RFMs at runtime.

The SAP ActiveX Controls

The SAP ActiveX Controls represent the oldest higher-level RFC middleware for Windows. There are two controls used for calling BAPIs and other RFMs, the SAP Functions Control and the SAP BAPI Control. The latter uses an object-oriented approach: You instantiate a business object type and call its methods (the BAPIs). In spite of the advantages of this object-orientation, it is recommended that you use the SAP Functions Control instead if — for whatever reason — you do not want to migrate to .NET yet. The main rationale behind this recommendation is that the SAP Functions Control provides slightly better performance, but I also suspect that SAP is more likely to fix problems in the SAP Functions Control than in the SAP BAPI Control.

The SAP DCOM Connector

The SAP DCOM Connector is based on the idea of generating proxy classes for business object types with their BAPIs as well as for non-BAPI RFMs. The application developers then use these proxy classes in their applications, which makes it easier to

[1] If you do not know C#, but at least a little bit of Java, take a look at my article "C# for Java Programmers" in the May/June 2003 issue of this publication.

write code by providing support for Code Completion (Intellisense) in the IDE. Unfortunately, no classes are generated for structure and table parameters, so the developer still needs to do a lot of copy and paste for the field names inside these.

While there is also support for server programming (where ABAP calls your component), it is offered as a separate component (COM4ABAP), which uses a totally different programming paradigm and also has some idiosyncrasies in its GUI.

Upgrading to the SAP .NET Connector

There are many advantages that the SAP .NET Connector has over the SAP ActiveX Controls and the SAP DCOM Connector. Some were mentioned in the introduction and others will become obvious in the article. So sooner or later everybody interested in Microsoft-specific SAP-enabled applications is likely to upgrade. The only bad news is that there are no migration tools, so you will have to make the necessary changes yourself.

SAP .NET Connector Overview

The SAP .NET Connector is fully integrated with Visual Studio .NET and supports all relevant features of RFC:

- You can build client (your component calls an RFM) and server (ABAP calls your component) applications.
- You can use synchronous RFC (sRFC), transactional RFC (tRFC), and queued RFC (qRFC).
- In addition to calling BAPIs and other RFMs, you can send and receive Intermediate Documents (IDocs).
- Single Sign-On is fully supported.

If your SAP component is based on the SAP Web Application Server 6.20 or later[2], you can use the SOAP protocol instead of RFC.

This article is limited to sRFC client applications (the most frequently used type of SAP-enabled application), but future articles will discuss other aspects of the SAP .NET Connector.

The .NET applications that you build can be of any type supported by Visual Studio .NET: Windows Applications (Windows Forms), ASP.NET Web Applications (Web Forms), and Console Applications.

The SAP .NET Connector generates proxy classes for RFMs (including BAPIs) with complete support for Code Completion, including structure and table parameters. Unfortunately, SAP has decided to treat BAPIs the same as other RFMs, i.e., there is no notion of a BOR business object type with the BAPIs as its methods, as there was for the SAP BAPI Control and the SAP DCOM Connector. You will see later in this article how using the proxy customization offered by the SAP .NET Connector allows you to achieve the same level of (BOR) object-orientation if you are willing to invest a little bit of effort.

In order to generate the proxy classes, the SAP .NET Connector needs access to the relevant metadata. The metadata can be retrieved from your SAP system (the normal case), the SAP Interface Repository (IFR), or a standard WSDL[3] file. The SAP .NET Connector generates a WSDL file (with extension “sapwsdl”) and C# source code based on this file. The generated classes can be part of your application assembly or built into a separate assembly.

You can combine as many RFMs as you like into one WSDL file. One main class is generated that contains the RFMs as methods.

[2] For example, R/3 Enterprise (4.7).

[3] Web Service Definition Language, an XML-based standard to describe Web Services.

In addition, one class each is generated for each SAP Data Dictionary structure used by the RFMs as structure or table parameters. These structure classes are derived from class *SAP.Connector.SAPStructure*, which is part of the SAP .NET Connector. Each field in a structure is represented by a C# read/write property.

For each table parameter, an additional class with suffix "Table" is generated. This class is basically a collection of rows. Table classes are derived from class *SAP.Connector.SAPTable*. The rows in the table object are of the type of the associated structure class, e.g., a table of class *BAPI0002_1Table* contains rows of type *BAPI0002_1*. The table class contains methods to add and remove rows. Table classes support data binding with suitable .NET controls like the data grid.

Connecting to SAP

The SAP .NET Connector contains support for connection pooling (through class *SAP.Connector.SAPConnectionPool*) as well as individual connections. Class *SAP.Connector.Destination* is the base class used to set the user and system information required to connect to SAP. Class *SAP.Connector.SAPLogonDestination* can be used to access the SAP systems defined in the "saplogon.ini" file of SAPGUI. The *SAP.Connector.SAPLoginProvider* class and the SAP Login Form are used in ASP.NET applications to facilitate connecting to SAP.

Downloading and Installing the SAP .NET Connector

You download the SAP .NET Connector from **http://service.sap.com/connectors**. You will need to know your SAP OSS userid to do this. There are three main files that you should download:

- **SAP .NET Connector Release Notes ("Release Notes.txt")**
  This file contains the release notes for the current release (1.0). You should definitely read this yourself, but some of the information contained therein will be discussed below.

- **SAP .NET Connector Installation ("SAP_Dotnet_Connector.zip")**
  This file contains the Windows Installer file "SAP.NET_Connector.msi".

- **SAP .NET Connector Documentation ("dotnetconnector10.pdf")**
  This is the documentation for the SAP .NET Connector in PDF format.

In addition there is a tutorial:

- **SAP .NET Connector Tutorial ("SAPDOTNETCONNECTOR.SIM")**
  This requires the SAPTutor Player ("SAP Tutor Player.EXE"), which can be downloaded from the same page.

There are three SAP-related prerequisites for using the SAP .NET Connector:

☑ Your SAP R/3 release must be 4.0 or later. R/3 3.1 is not supported and according to the release notes there seems to be no intention to remedy this limitation in the future. In the unlikely event that you are still running R/3 3.1 systems, you have to use either the SAP Java Connector (JCo) or the abovementioned precursors to the SAP .NET Connector, the SAP ActiveX Controls or the SAP DCOM Connector.

☑ To generate proxy classes from the metadata in your SAP system (and only for that purpose), you need to have a Java Virtual Machine installed on your system since the SAP .NET Connector uses JCo to retrieve metadata from SAP. If you want to use a different VM than the one from Sun, you need to make some registry changes (details are given in the release notes). The runtime of JCo is

Figure 1 Data Types in ABAP and the SAP .NET Connector

ABAP Data Type	Description	C# Data Type
C	Fixed-length string	System.String
g	Variable-length string	System.String
X	Fixed-length raw data	byte[]
y	Variable-length raw data	byte[]
N	Numeric character (0-9)	System.String
P	Packed number (BCD)	System.Decimal
F	8-byte floating point number	System.Double
I	4-byte integer	System.Int32
s	2-byte integer	System.Int16
b	1-byte integer	System.Int32
D	8-byte date string	System.String
T	6-byte time string	System.String

automatically installed when you install the SAP .NET Connector.

☑ A copy of the RFC Library ("librfc32.dll", release 6.20 or later) must reside on your machine. This library is automatically installed when you install SAPGUI or other SAP components like the SAP Java Connector.

For runtime deployment of a solution, you only need the "SAP.Connector.dll" and "librfc32.dll" files.

To commence the installation of the SAP .NET Connector, simply double-click on the "SAP.NET_Connector.msi" file. The installation worked flawlessly on my PC. By default, the installation target is "C:\Program Files\SAP\SAP .NET Connector". In this directory there is a "Samples" directory with quite a few sample programs for different types of applications. Studying the sample programs is highly recommended.

The next time you start Visual Studio .NET, the SAP .NET Connector is ready to be used!

Data Type Conversions

ABAP has its own set of data types that have to be mapped to data types available in .NET. **Figure 1** contains a table that lists all scalar ABAP data types that can be used for RFC, a description, and the respective C# data type.[4]

Most of these data type mappings are straightforward. So we only need to discuss some of them in detail:

- **Data types C and g:** Trailing blanks of the ABAP string are automatically removed by the SAP .NET Connector.
- **Data type N:** In ABAP, this is a string that can contain only the decimal digits 0 through 9. Space characters (blanks) are not permitted. When sending a value of type N to SAP, make sure that the string you use is padded with leading "0" characters if it is shorter than the length defined for the field in SAP.

[4] This table is more complete than the one on page 78 of the SAP .NET Connector documentation.

The SAP .NET Connector documentation says in its discussion of this data type (page 78): "Some type *N* fields like invoice number or customer number require you to enter the string with leading zeroes." This is only half true. Customer numbers and invoice numbers are alphanumerical and stored in fixed-length character strings (data type C, not N) using the ALPHA conversion exit in SAP. If they contain only digits (0-9), then leading zeroes must be supplied by the external application.

- **Data type b:** Figure 1 indicates that this data type is mapped to a 4-byte integer, although one would expect a 2-byte integer, like for data type s. It turns out, though, that while the proxy generation for an RFM with a parameter of type b works fine, at runtime you will receive an exception of type *SAP.Connector.RfcMarshalException* with the following additional information: "System exception thrown while marshaling .NET type 0 to RFCTYPE_INT1". It appears that the SAP .NET Connector currently does not support this data type, which is not a big issue since I know of no BAPI where it is used. There may be some non-BAPI RFMs with this data type, though.

- **Data types D and T:** ABAP dates and times are mapped to strings, using the ABAP standard formats of YYYYMMDD and HHMMSS, respectively. The SAP .NET Connector provides some useful methods in class *SAP.Connector.RfcConvert* that allow you to convert from and to the appropriate C# data types.

Date and Time Conversions

Class *SAP.Connector.RfcConvert* offers the following conversion methods:

- *DateTimeToRfcDate()* converts the date in a *System.DateTime* object to the string required by SAP.

- *DateTimeToRfcTime()* converts the time in a *System.DateTime* object to the string required by SAP.

- *RfcDateToDateTime()* creates a *System.DateTime* object from the SAP date string.

- *RfcTimeToTimeSpan()* creates a *System.TimeSpan* object from the SAP time string.

- *TimeSpanToRfcTime()* converts the time in a *System.TimeSpan* object to the string required by SAP.

One could argue about the choice of the *TimeSpan* class here, but there is a much bigger issue: ABAP is quite liberal about the values that are allowed in type D and T fields. A date of "99999999" is used in some RFMs to denote the end of the universe, but it is definitely not a legal date. Trying to use the *RfcDateToDateTime()* on such a value will cause an exception of type *System.ArgumentOutOfRangeException* to be thrown. ABAP also allows a time value of "240000", which is not a legal time as such. Also, some RFMs return a field of type D and one of type T, which together are equivalent to one *DateTime* object in .NET. Hence I have written a method *CombineDateAndTime()* for just these cases (**Listing 1**). The specific decisions about which values to allow and how to handle slightly illegal values like "240000" are obviously very subjective, but the supplied code should be a good starting point for your own implementation. Based on these ideas, you can then also write additional conversion methods to supplement the ones offered by SAP.

RFM Parameters in the SAP .NET Connector

Each method representing an RFM in a proxy class has all the parameters defined in SAP (unless you remove some parameters, see the discussion of proxy customization later). Many RFMs, especially the BAPIs, have quite a few parameters. A normal

Listing 1: Combining a Date and a Time

```
public static DateTime CombineDateAndTime(string date, string time) {
  if (date == null || time == null) {
    throw new ArgumentOutOfRangeException(
      "The date and time parameters may not be null.");
  }
  if (time.Length == 0) time = "000000";
  if (date.Length != 8 || time.Length != 6) {
    throw new ArgumentOutOfRangeException(
      "The date must be 8 bytes, and the time 6 bytes long.");
  }
  int year;
  int month;
  int day;
  int hour;
  int min;
  int sec;
  try {
    year = Convert.ToInt32(date.Substring(0, 4));
    month = Convert.ToInt32(date.Substring(4, 2));
    day = Convert.ToInt32(date.Substring(6, 2));
  }
  catch (Exception ex) {
    throw new ArgumentException("Date value is not numeric.", ex);
  }
  try {
    hour = Convert.ToInt32(time.Substring(0, 2));
    min = Convert.ToInt32(time.Substring(2, 2));
    sec = Convert.ToInt32(time.Substring(4, 2));
  }
  catch (Exception ex) {
    throw new ArgumentException("Time value is not numeric.", ex);
  }
  if (month > 12) month = 12;
  if (day > 31) day = 31;
  if (hour == 24 && min == 0 && sec == 0) {
    hour = 23;
    min = 59;
    sec = 59;
  }
  try {
    return new DateTime(year, month, day, hour, min, sec);
  }
  catch (Exception ex) {
    throw new ArgumentException(
      "Could not create a DateTime object.", ex);
  }
}
```

application uses only a small subset of these parameters. Do you have to pass all the parameters, even the ones you are not interested in? Let us look at how RFM parameters are dealt with in the proxies:

- Import parameters are standard C# parameters (i.e., they are not defined as `out` or `ref`). If the parameter is a reference type (e.g., a structure, a table, or a string) you can pass `null` if you do not want to use the parameter. If the parameter is a value type (e.g., an integer), you have to pass a variable or literal of the proper type. This leads to an interesting challenge if the parameter is optional, has a default value in SAP, and you would like to use that default. Since you have to pass a value, you have to know the default value. The simple solution is to look it up in the SAP Function Builder and hard-code it. The good solution would be to retrieve it at runtime, but that would require you to build a component to do this, since — to the best of my knowledge — the SAP .NET Connector does not provide the default value for you.

- Export parameters are `out` parameters in C#. You must always pass a variable of the proper type for each of these parameters, although, from SAP's standpoint, export parameters are always optional! This can be quite inconvenient, and the only way to avoid this is to remove the parameters that you do not need from the proxy class.

- Table parameters are `ref` parameters in C#. You must always provide a variable of the proper type for each of these parameters. If the variable is `null`, the SAP .NET Connector assumes that you are not interested in this table and will not use this parameter. Hence it is extremely important to instantiate a table variable before invoking an RFM if you are interested in the data in this parameter after the call.

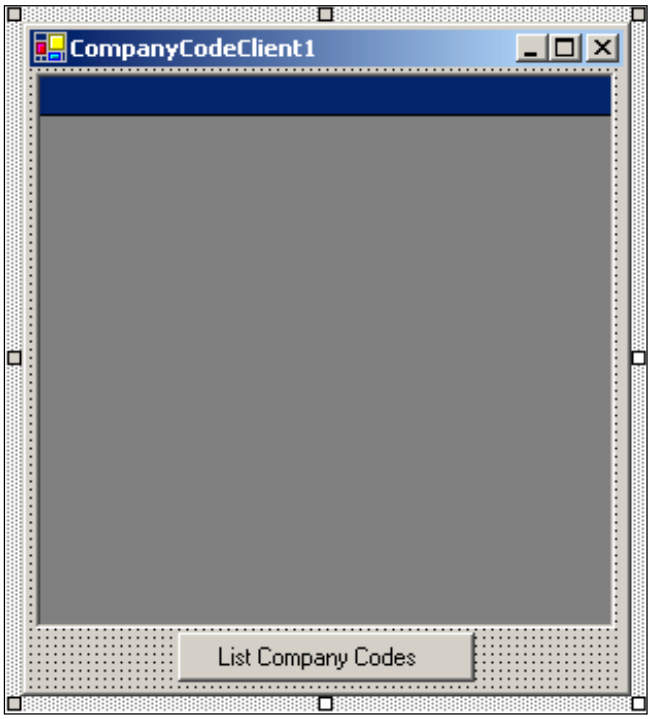
- Changing parameters are `ref` parameters in C#. You must always provide a variable of the proper type for each of these parameters.

A Very Simple Client Program

For our first sample program, I will build a simple GUI (with one Windows Form) that calls one BAPI (*CompanyCode.GetList*[5]) and displays the returned list in a data grid. The required SAP proxy classes will be part of the project.

First, I create a new project of type *Windows Application*, called `CompanyCodeClient1`. Then I add a data grid and a push button to obtain the GUI shown in **Figure 2**. Nothing SAP-related has taken place so far.

Figure 2 The CompanyCodeClient1 GUI



To add the required SAP proxy classes, I add a new item to the project (see **Figure 3**), calling the proxy `SAPCompanyCode1`.

[5] This BAPI has become the *Hello World* of the SAP BAPI community. It is very simple to call and while it does not do anything terribly interesting, it is a proper BAPI and thus a good starting point for studying an SAP connector.

Figure 3 *Adding the SAP Proxy Classes*

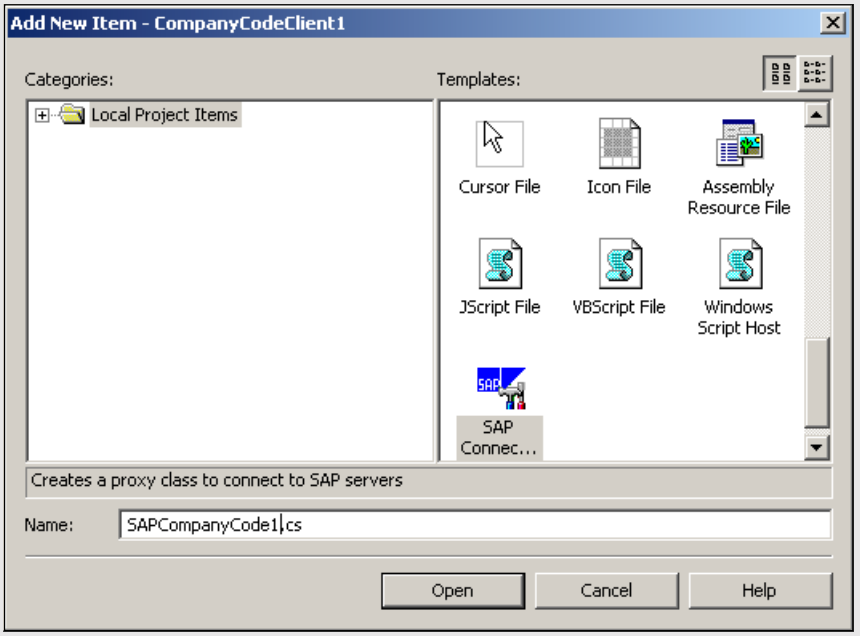
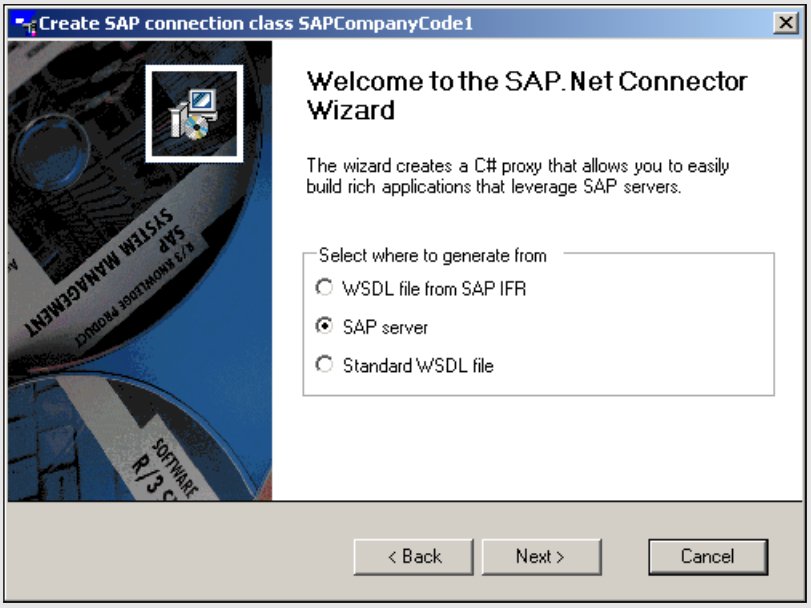


Figure 4 *The SAP Proxy Generation Wizard, Step 1*



Clicking on the *Open* button causes the SAP proxy generation wizard to appear (**Figure 4**).

I want to retrieve the SAP metadata for the proxy generation from an actual SAP system, so I keep the

Figure 5 The SAP Proxy Generation Wizard, Step 2

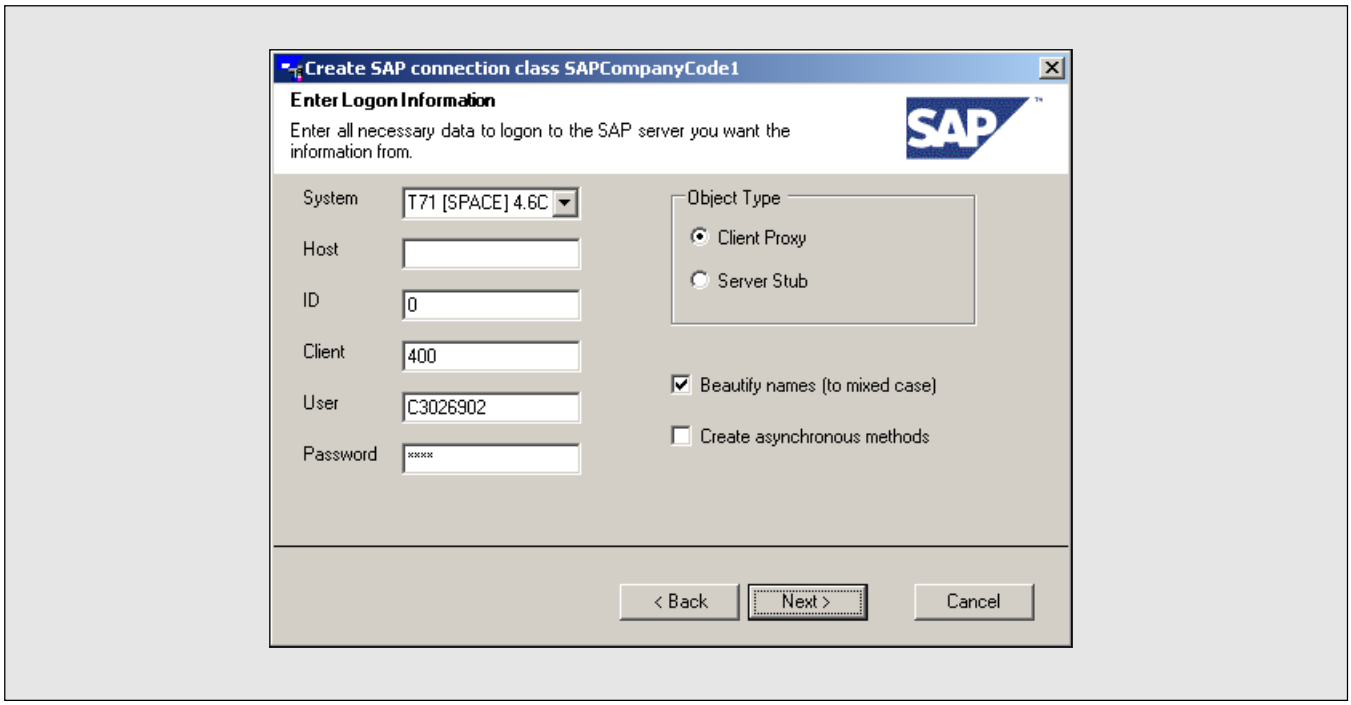
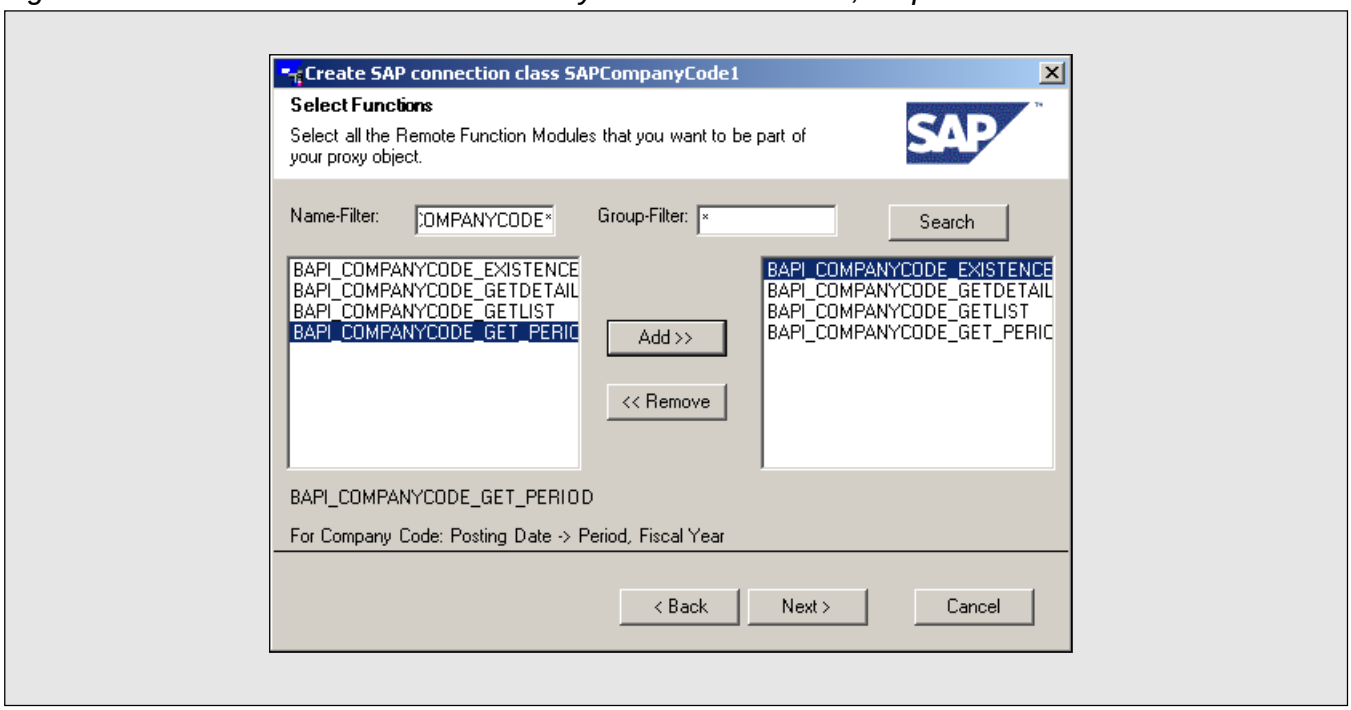


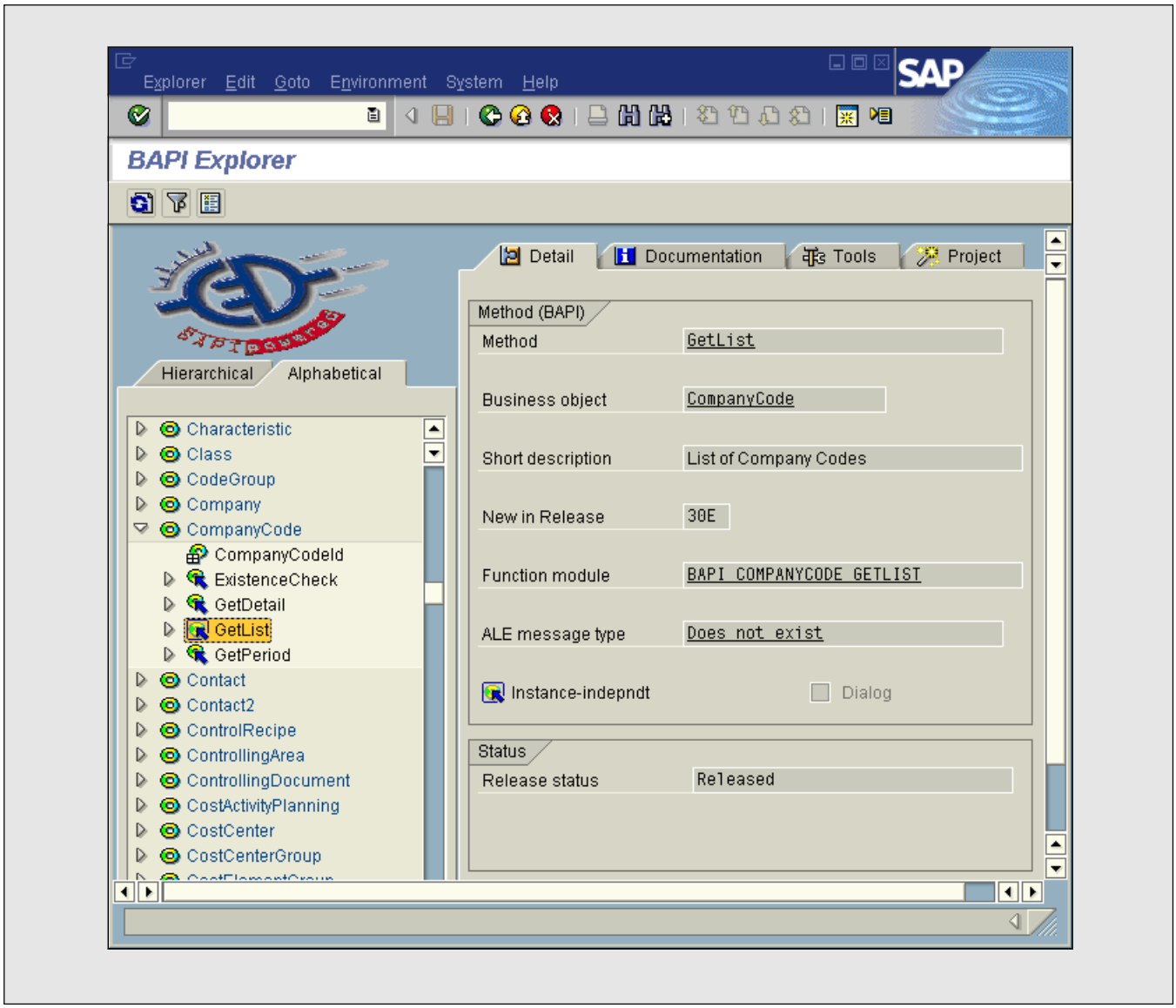
Figure 6 The SAP Proxy Generation Wizard, Step 3



default setting (SAP server) and click on the *Next* button, which takes me to **Figure 5**.

Here I select an SAP system from the list defined for my SAPGUI and enter the Client, User,

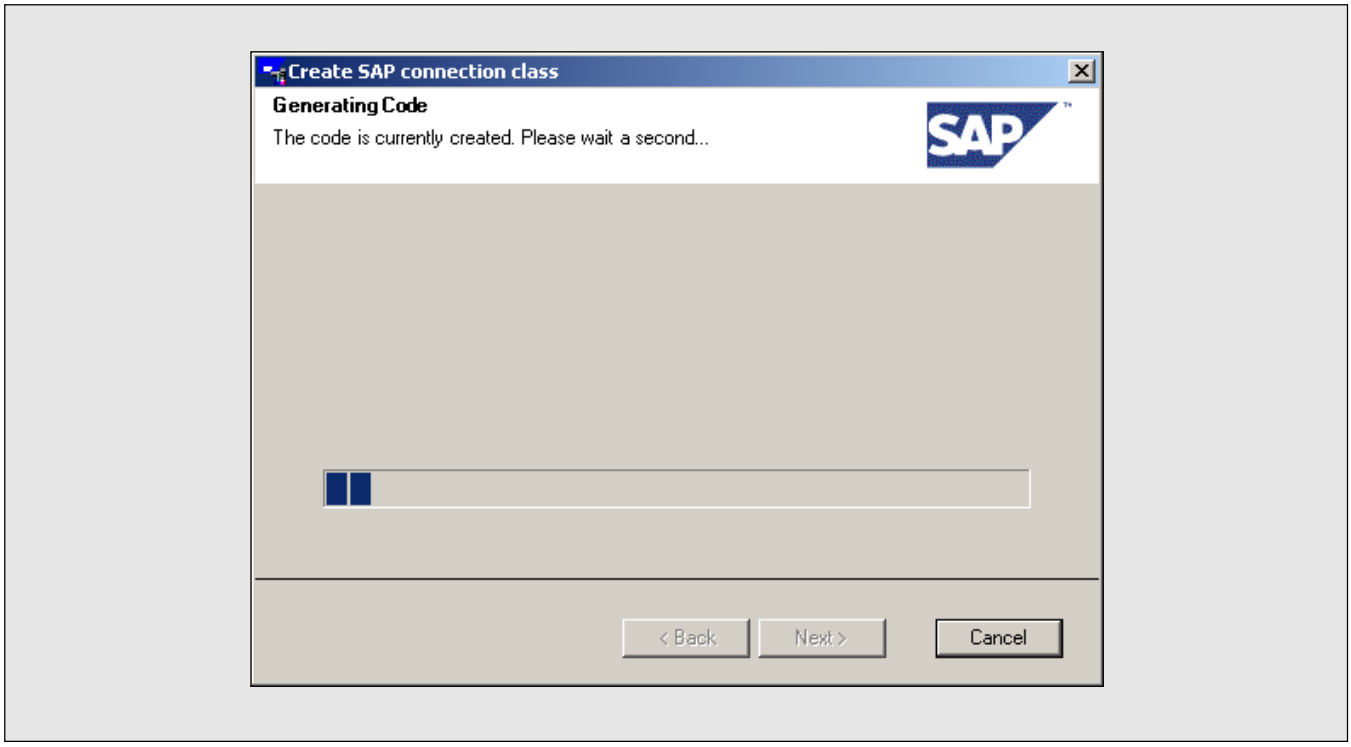
Figure 7 Looking up RFM Names in the SAP BAPI Explorer



and Password information. Clicking *Next* again lets me proceed to the RFM selection screen (**Figure 6**).

As discussed above, the SAP Proxy Generator does not support the notion of SAP business object types, so you can only enter the names of the RFMs implementing the BAPIs. **Figure 7** is a screenshot of the SAP BAPI Explorer, showing the RFM name for *CompanyCode.GetList* (BAPI_COMPANYCODE_GETLIST). For the current scope of the sample program it would have been sufficient to select just this one RFM in Figure 6, but in order to be able to extend the program later, I looked up the RFM names of the other BAPIs, discovered that they all started with the string BAPI_COMPANYCODE, and entered BAPI_COMPANYCODE* in the field Name-Filter to retrieve them all. Clicking on the *Search* button filled the left list box in Figure 6. I then selected each RFM and pressed the *Add* button to reach Figure 6 as shown.

Figure 8 The SAP Proxy Generation Wizard, Step 4

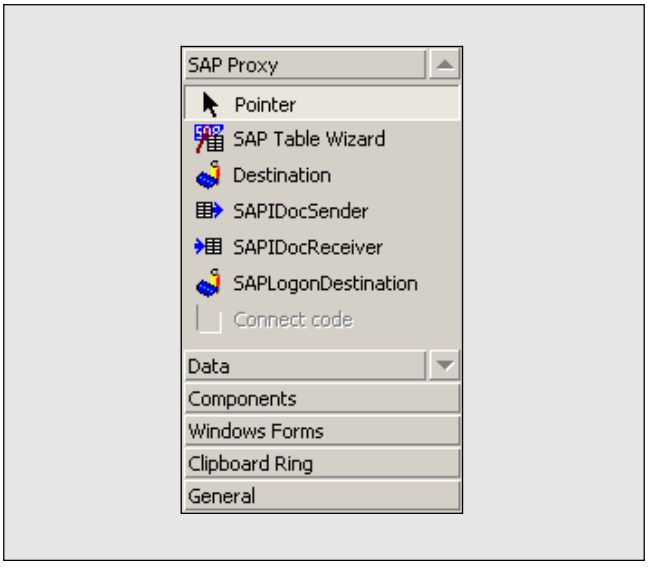


> ✓ ***Note!***
>
> *Clicking on the Search button in Figure 6 does not cause an hourglass cursor to appear. Since there is no error message when you enter an invalid RFM name in the Name-Filter field (the left list box simply stays as before), it is difficult to determine whether the wizard is still retrieving metadata or the entered RFM name was wrong. If you want to look up just one RFM make sure that its name is spelled correctly. If you want to retrieve multiple RFMs starting with the same string make sure that you do not forget the * character to indicate a generic search.*

Clicking *Next* in Figure 6 starts the generation process. **Figure 8** is displayed while the code generation is running.

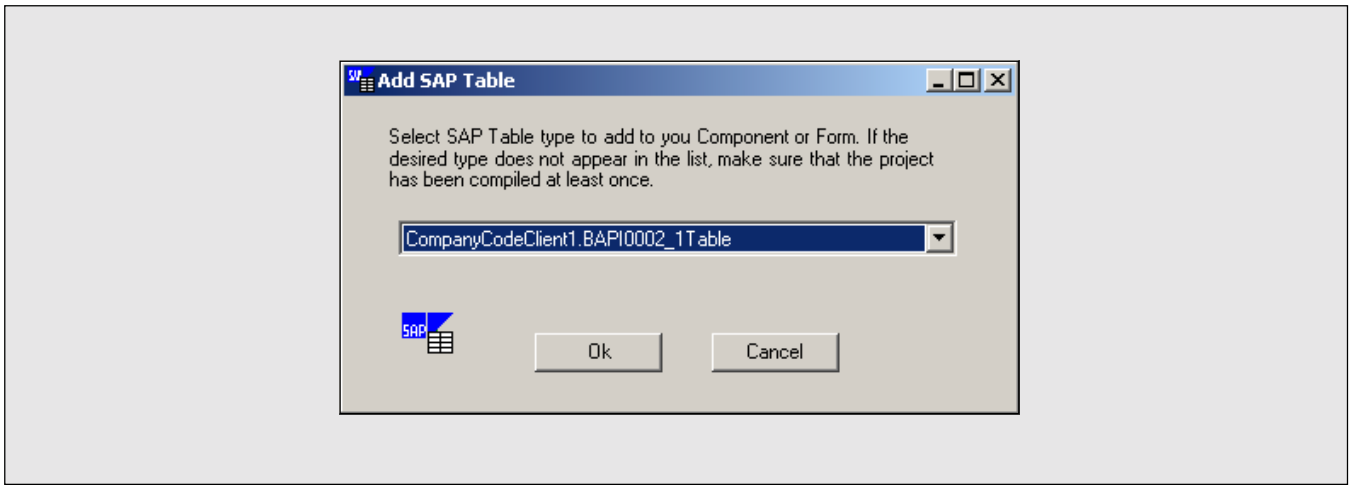
The easiest way to display the table returned by

Figure 9 The SAP Proxy Toolbox



CompanyCode.GetList is to use the SAP Table Wizard, which is part of the SAP Proxy Toolbox (**Figure 9**). This toolbox is automatically added to

Figure 10 *The SAP Proxy Toolbox Table Wizard*



Visual Studio .NET when the SAP .NET Connector is installed. Double-clicking the SAP Table Wizard in the toolbox brings up the wizard, as shown in **Figure 10**.

Since there is only one table parameter used by the BAPIs of business object type *CompanyCode*, it is selected by the wizard and I simply click *Ok* to generate the appropriate code for my project.

If there are multiple table parameters for the RFMs you have selected during the proxy generation, then you have to look up the names of the underlying SAP Data Dictionary structure to obtain the correct name. Take a look at **Figure 11**. This is the Visual Studio .NET Solution Explorer. I have selected the "Show All Files" mode (the middle icon at the top of Figure 11 allows you to toggle the mode) in order to see all the classes generated by the proxy generator.

Class *SAPCompanyCode1* contains all the RFMs that I selected as methods. In addition there is one class for each table or structure parameter, named after the underlying SAP Dictionary Structure. For each table parameter, an additional class with the suffix "Table" is generated.

By looking up the detailed parameter information for the *CompanyCodeList* parameter of

Figure 11 *Looking up a Parameter in the Solution Explorer*

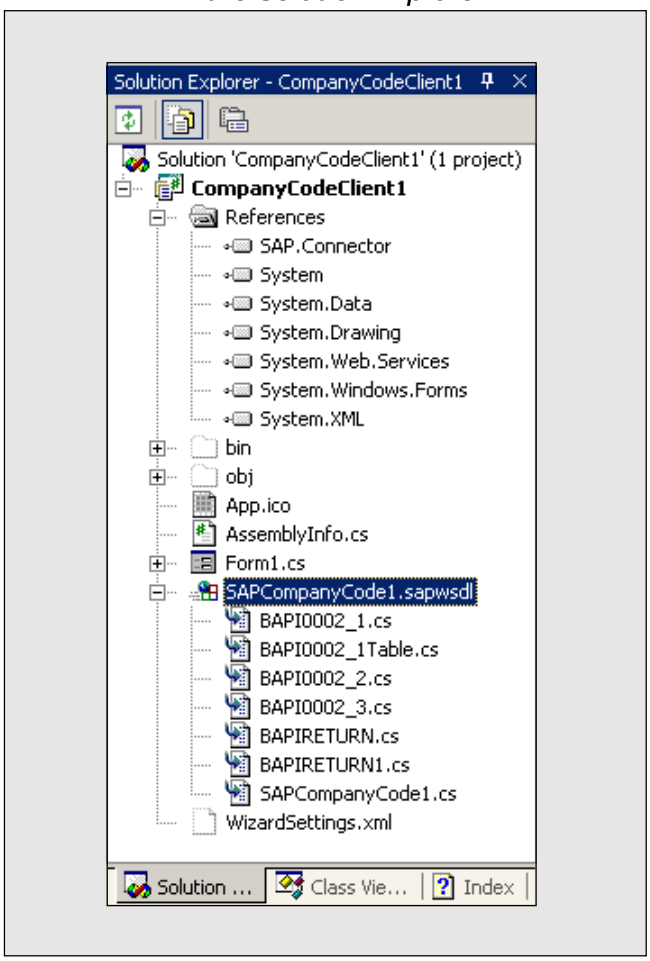
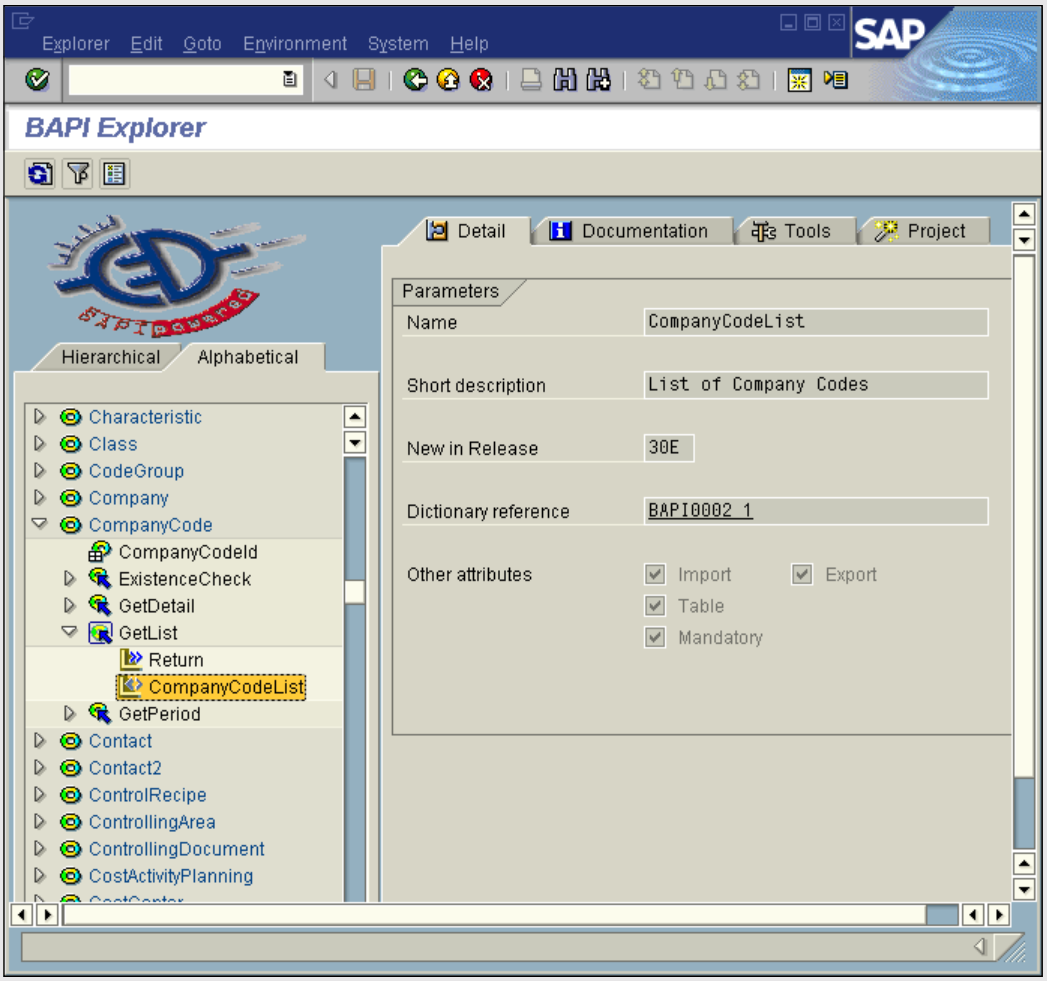


Figure 12 Looking up a Parameter in the BAPI Explorer



CompanyCode.GetList in the BAPI Explorer (**Figure 12**), you can find out that the name of the dictionary structure for this parameter is BAPI0002_1, hence *BAPI0002_1Table* is the name for the generated class representing this table parameter.

If you think that this is not a very user-friendly name, just be patient for a little bit. The SAP .NET Connector allows you to customize your proxy classes quite extensively, enabling you to use more meaningful names. This feature will be discussed later in this article.

In order to get automatic data binding between the SAP table parameter returned by *CompanyCode.GetList* and the data grid, I set its *DataSource* property appropriately. This is easy, because due to the code generated by the SAP Table Wizard, I can simply select the proper value from the drop-down list shown in **Figure 13**.

Since I will also need to connect to SAP later, I add a *Destination* object by double-clicking the Destination entry in the SAP Proxy Toolbox.

The result of these activities is shown in

Figure 13 Setting the DataSource Property

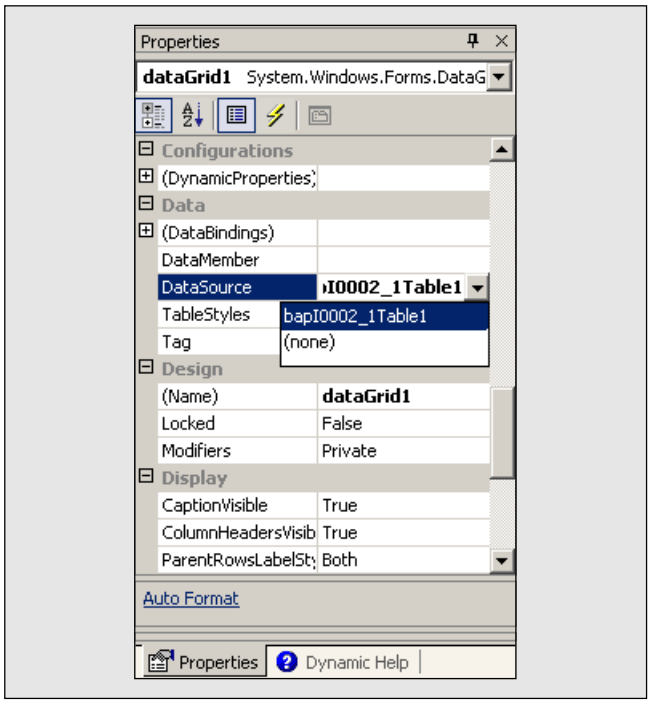
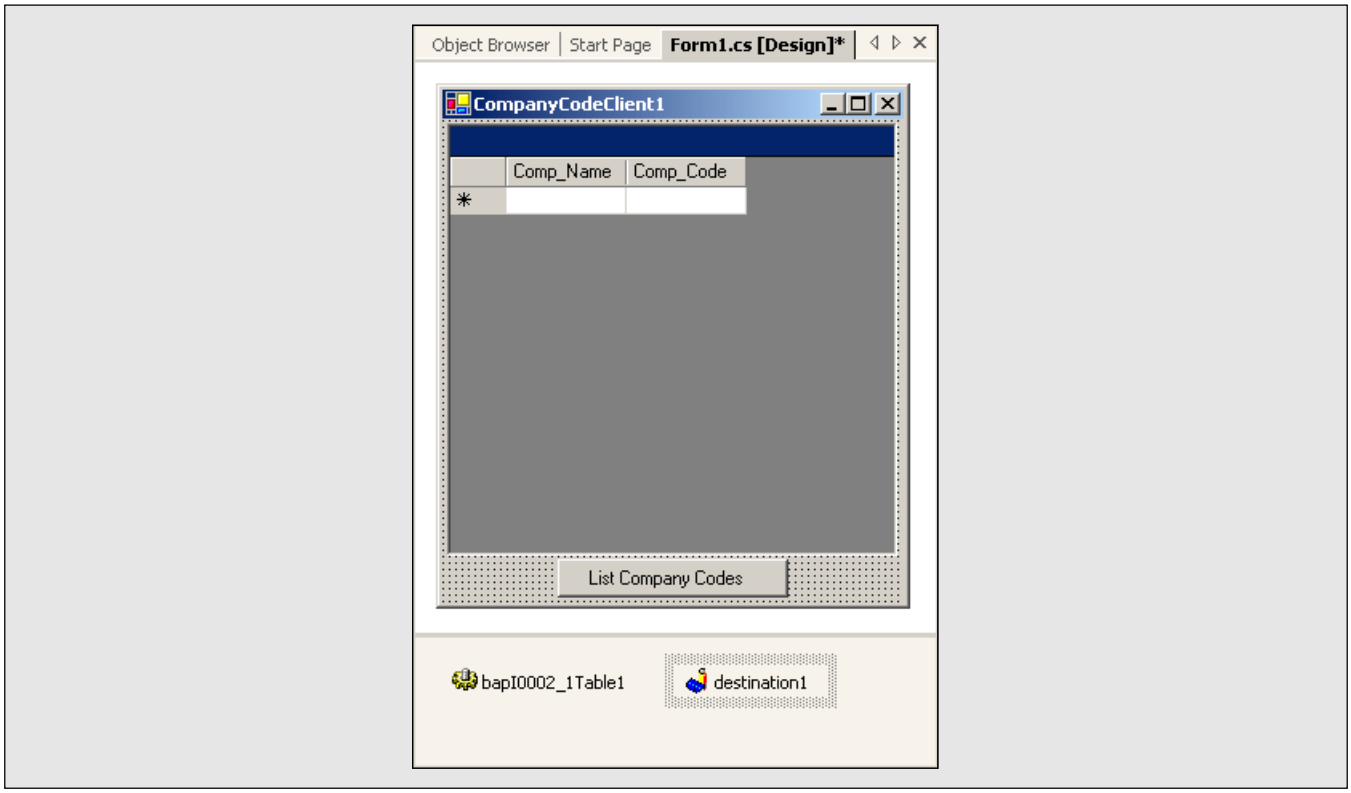


Figure 14. Not only do you see the two variables added from the SAP Proxy Toolbox, you also see the columns of the table parameter in the data grid. This magic is accomplished by the design-time data binding capabilities of the SAP .NET Connector. The column headers for the data grid are the field names of the BAPI0002_1 dictionary structure, beautified somewhat to use mixed case (the result of leaving the beautification option in Figure 5 checked). These field names are obviously not appropriate for end users. One way to obtain better names is to customize the generated proxy classes (discussed later). Other alternatives, like retrieving the appropriate texts from SAP at runtime, are beyond the scope of this article.

Note that the sequence of the fields is different from what is defined in the SAP Data Dictionary.

We are almost there. Most of the required code has been generated for us. Just a few lines of code

Figure 14 Design-Time Data Binding



Listing 2: Handwritten Code for CompanyCodeClient1

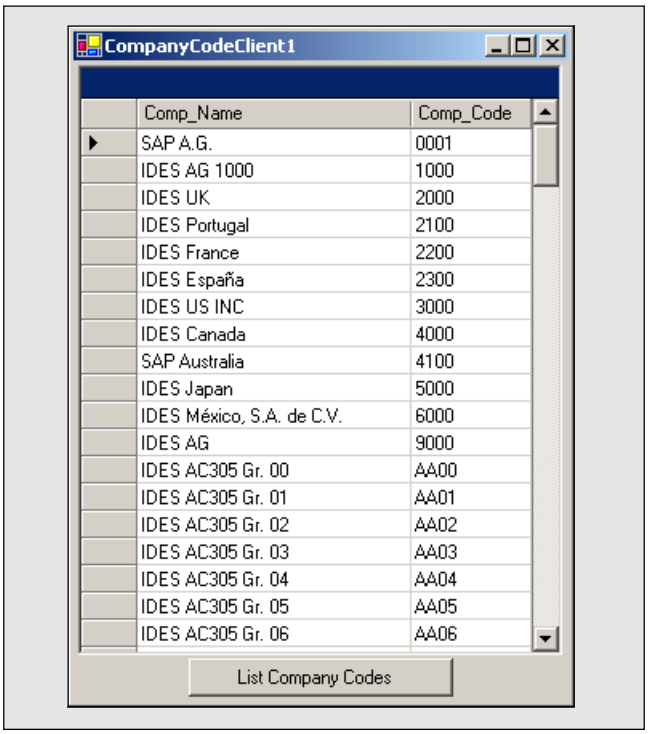
```
private SAPCompanyCode1 sapCompanyCode;
private BAPIRETURN bapiReturn;

  private void button1_Click(object sender, System.EventArgs e) {
    sapCompanyCode = new SAPCompanyCode1();
    destination1.AppServerHost = "hostname";
    destination1.SystemNumber = 0;
    destination1.Client = 400;
    destination1.Username = "userid";
    destination1.Password = "password";
    sapCompanyCode.Connection = new SAPConnection(destination1);
    sapCompanyCode.Bapi_Companycode_Getlist(out bapiReturn,
                                            ref bapI0002_1Table1);
  }
```

are needed to complete the application. **Listing 2** contains the required code. There are two variable declarations, one for the main proxy class that contains the RFMs as methods, the other one for the *Return* parameter of *CompanyCode.GetList*.

We also need some way to connect to SAP and call the BAPI. This is done when the user presses the only button in our GUI, and the code is shown in the *button1_Click* method in Listing 2. First, we initialize the sapCompanyCode variable. Then we set the system and user properties of the destination1 variable created from the SAP Proxy Toolbox earlier. In the last statement but one, we initialize the *Connection* property of sapCompanyCode to a new SAPConnection object initialized with the *Destination* object. Finally, the BAPI itself is executed.

Running this program (shown in Appendix A in its entirety) yielded the screenshot shown in **Figure 15**.

Figure 15 Running the Sample Program



Checking the Return Parameter

Obviously, this sample program leaves a lot to be desired. Let us look at the BAPI *Return* parameter first. While we pass it into the method call (see Listing 2), we do not ever check it. We could add specific code to check this parameter whenever we call a BAPI, but this would assume that every application programmer knows how to do this properly. Hence it is better to provide a generic way

of checking a BAPI *Return* parameter. Writing a generic method for this is not totally straightforward with the SAP .NET Connector since it uses proxy classes and provides no generic way to access RFMs. While the proxy classes make life easier for a normal application programmer, it is easier to build generic routines without them. But where there is a will…

Our generic BAPI Return code checker can rely only on a few things:

- The *Return* parameter is either a structure or a table.
- The *Return* parameter is based on one of the following SAP Data Dictionary structures: BAPIRETURN, BAPIRETURN1, BAPIRET1, BAPIRET2.
- If the *Return* parameter is a structure, then success is signified by field TYPE containing either “S” or an empty string.
- If the *Return* parameter is a table, then success is signified by a table with no rows or each row containing “S” or an empty string in field TYPE.

What the generic method cannot rely upon is the class name for the proxy class used to represent the parameter or the names of the individual fields used in the proxy. It took me a while to come up with the solution shown in **Listing 3**.

Listing 3: Method IsBapiReturnCodeOkay()

```
public static bool IsBapiReturnCodeOkay(Object returnParameter) {
  string type;
  try {
    SAPTable table = returnParameter as SAPTable;
    if (table != null) {
      if (table.Count == 0) return true;
      System.Data.DataTable adoTable = table.ToADODataTable();
      foreach (System.Data.DataRow row in adoTable.Rows) {
        type = (string)row[0];
        if (type != "S" && type != "")
          return false;
      }
      return true;
    }
    SAPStructure structure = returnParameter as SAPStructure;
    if (structure != null) {
      type = (string)structure[0];
      if (type != "S" && type != "")
        return false;
      else
        return true;
    }
  }
  catch (Exception ex) {
    throw new ArgumentException
      ("Parameter must be a BAPI return table or structure.", ex);
  }
  throw new ArgumentException
    ("Parameter must be a BAPI return table or structure.");
}
```

Listing 4: Testing the IsBapiReturnCodeOkay() Method

```
bool result =
  De.ARAsoft.SAPdotNet.StaticMethods.IsBapiReturnCodeOkay(bapiReturn);
MessageBox.Show("The BAPI call was successful: " + result);
```

Listing 5: Converting a Sales Order Type

```
private SAPProxy1 sapProxy;
private BAPICONVRSTable conversionTable;
private BAPIRET2Table bapiReturn;
private Destination destination;
  private void button1_Click(object sender, System.EventArgs e) {
    destination = new Destination();
    destination.AppServerHost = "hostname";
    destination.SystemNumber = 0;
    destination.Client = 400;
    destination.Username = "userid";
    destination.Password = "password";
    sapProxy = new SAPProxy1();
    sapProxy.Connection = new SAPConnection(destination);

    conversionTable = new BAPICONVRSTable();

    BAPICONVRS conversionRow;
    conversionRow = new BAPICONVRS();
    conversionRow.Objname = "SalesOrder";
    conversionRow.Method = "CreateFromDat1";
    conversionRow.Parameter = "OrderHeaderIn";
    conversionRow.Field = "DOC_TYPE";
    conversionRow.Ext_Format = tbExternal.Text.Trim();
    conversionTable.Add(conversionRow);

    bapiReturn = new BAPIRET2Table();

    sapProxy.Bapi_Conversion_Ext2int1(ref conversionTable,
                                      ref bapiReturn);
    bool result =
      De.ARAsoft.SAPdotNet.StaticMethods.
      IsBapiReturnCodeOkay(bapiReturn);
    MessageBox.Show("The BAPI call was successful: " + result);

    tbInternal.Text = conversionTable[0].Int_Format;
  }
```


To test the *IsBapiReturnCodeOkay()* method for a structure *Return* parameter, I added the code in **Listing 4** to the sample program.

Table Return Parameters

To test the *IsBapiReturnCodeOkay()* method for a table *Return* parameter, I wrote the code shown in **Listing 5**. This program (you can see it in action in **Figure 16**) allows a user to convert a sales order type from the external (i.e., language-dependent) format to the internal (i.e., SAP) format. The program has the added value of showing you how to deal with tables in your programs.

This time I used none of the SAP Proxy Toolbox facilities, therefore the destination, conversion Table, and bapiReturn variables are instantiated by handwritten code. Why is bapiReturn instantiated before the BAPI is called? The SAP BAPI Explorer (**Figure 17**) indicates that the *Return* parameter of *BapiService.DataConversionExt2Int1* (the conversion

Figure 16 Running the Sales Order Type Conversion Program

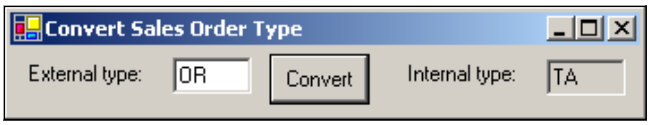
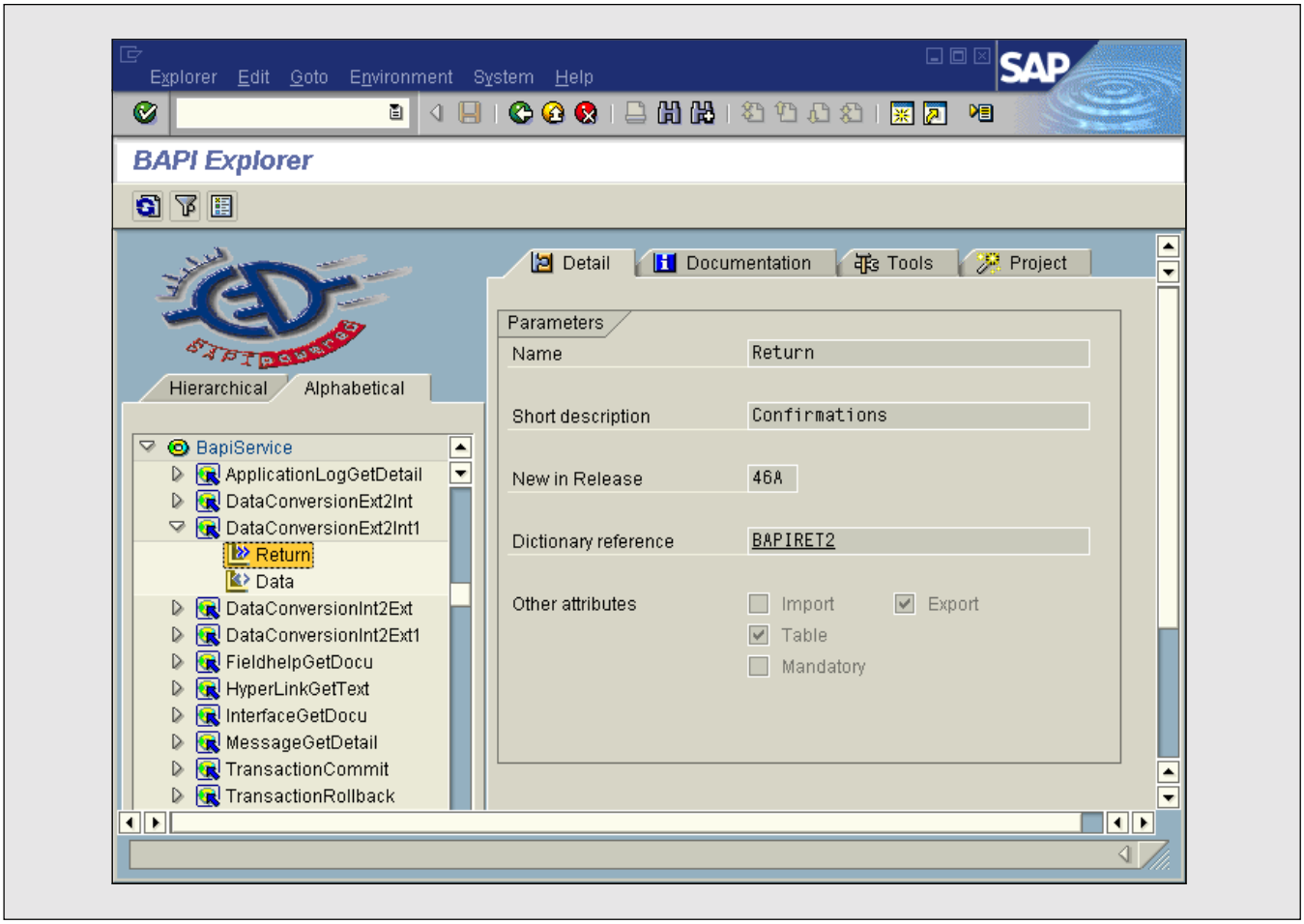


Figure 17 Object Type BapiService in the SAP BAPI Explorer



BAPI we are calling) is export only. But you need to remember a few things here:

- The import/export attributes of a table parameter in the SAP BAPI Explorer are totally ignored by the underlying RFM. They help a developer to understand a BAPI, but do not affect its runtime behavior.
- If a table parameter passed by reference is not instantiated, the SAP .NET Connector will totally ignore it. Variable bapiReturn would still be `null` after the BAPI call if we had not instantiated it.

To add a new row to the table parameter represented by variable conversionTable, I instantiate an object of type *BAPICONVRS* (variable conversionRow). I then assign values to the fields required by the BAPI, and add the row to the table using method *Add()*.

To access the converted value (last statement in Listing 5), I simply use the indexer on conversionTable (row zero, which indicates the first row) and then retrieve the required field (property *Int_Format*).

Appendix B has the complete source code for the sales order type conversion program.

Customizing Proxies

So far, we have added the SAP .NET Connector proxy classes directly to the individual applications, but you can also put them into separate assemblies, shared by multiple applications. This makes sense especially if you customize the proxies to make them easier to use. Other developers can then benefit from the effort you put into the customization. Some of the names we had to use so far, like *BAPI0002_1Table*, are far from easy to remember. Fortunately, you do not have to. The SAP .NET Connector allows extensive customization of the proxies, enabling you to change names, remove parameters, and define default values for fields.

To create a separate assembly for the proxies, create a new project of type "SAP Connector". You will then go through exactly the same steps as discussed for our first sample program. The only change I made was to disable the "Beautify names" option in Figure 5 so that the SAP .NET Connector will accept and apply the names I assign during

Figure 18 The Solution Explorer

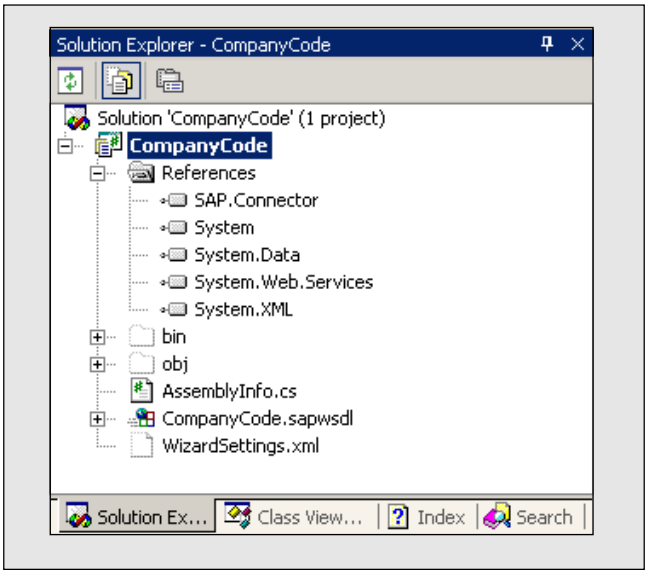


Figure 19 Changing the Name Space

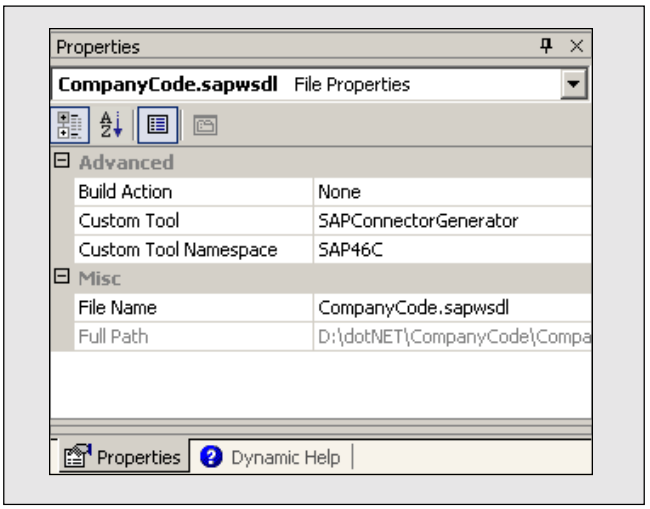
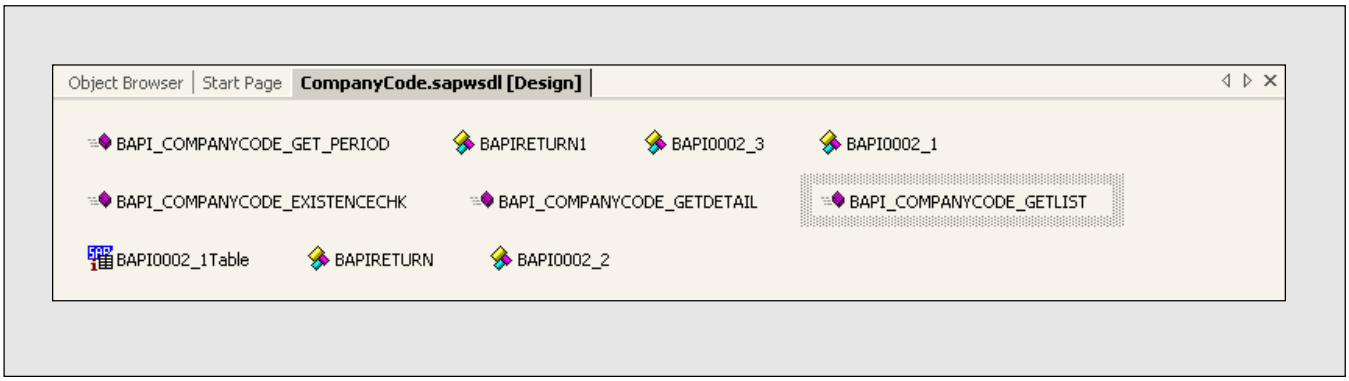


Figure 20 Methods and Structure/Table Classes in the Designer



customization. For my new proxy assembly I have selected the name CompanyCode. To apply customization to the proxy classes, you start the SAP Designer by double-clicking the .sapwsdl file in the Visual Studio .NET Solution Explorer (cf. **Figure 18**).

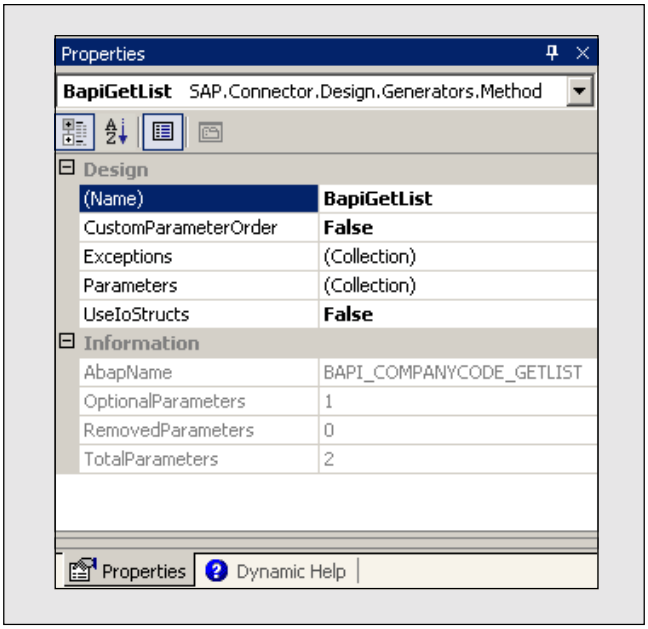
The first change I want to apply is a new name space for the proxy classes. Single-clicking on the .sapwsdl file allows me to do this (see **Figure 19**). I have changed the name space to SAP46C to indicate that this is the release for which the proxies were generated.

Now I want to change individual methods of the main proxy class, *CompanyCode*. All methods as well as the classes generated for structure and table parameters are visible in the middle pane of Visual Studio .NET, as seen in **Figure 20**.

Selecting BAPI_COMPANYCODE_GETLIST in Figure 20 lets me edit this method as shown in **Figure 21**. All decisions about name spaces, method names, and so on should follow some kind of standard that your company should define. Let me tell you my current standard for shared proxies for BAPIs:

- I create one assembly per SAP business object type. This allows me to get the same level of object-orientation for my proxies that were offered by the SAP BAPI Control and the SAP DCOM Connector.

Figure 21 Changing the Method Name



- The name of the project is identical to the name of the SAP business object type (*CompanyCode* in this example).
- The name space is “SAP” plus the release for which the proxies were generated (“46C” in this example).
- The method names are changed to the ones shown in the SAP BAPI Explorer, prefixed by “Bapi” to make all BAPIs show up together when using Intellisense in Visual Studio .NET.

Figure 22 New Method Names

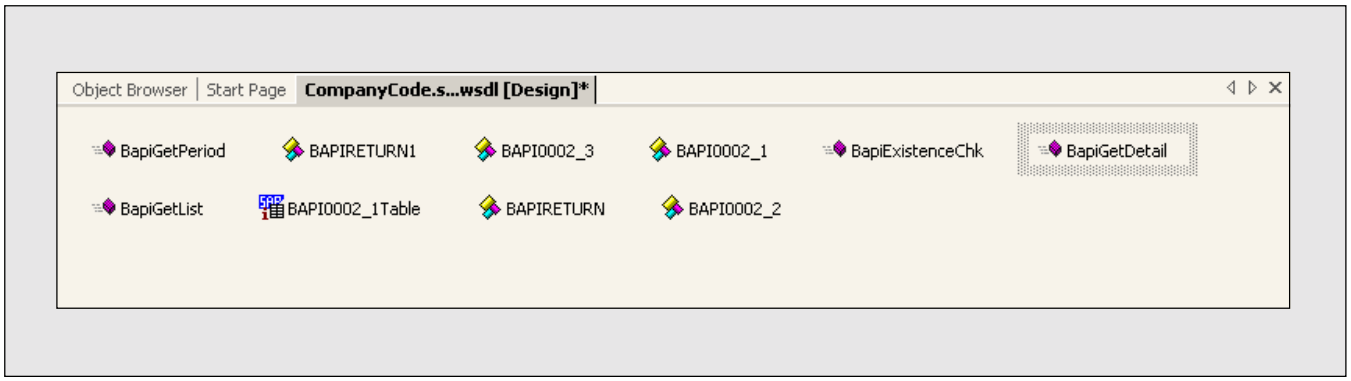


Figure 23 After Renaming BAPI0002_1

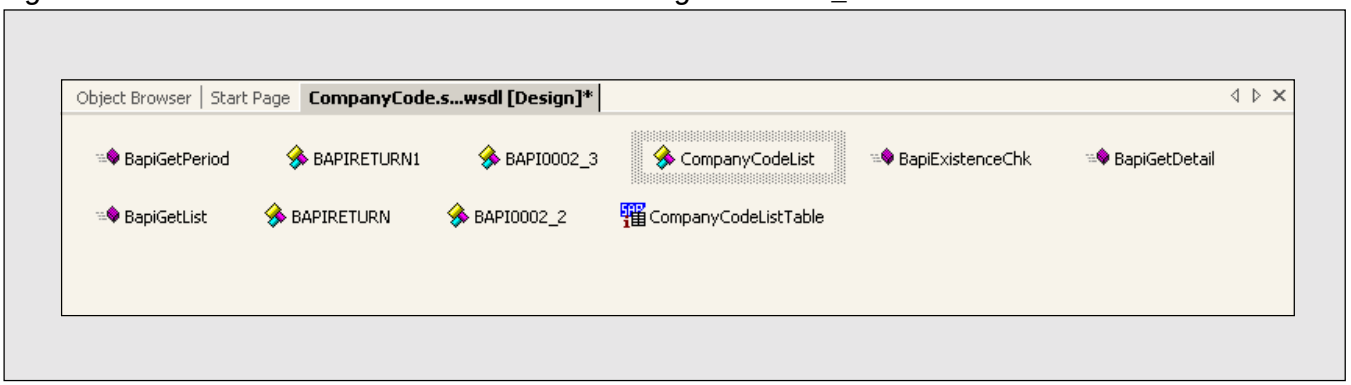
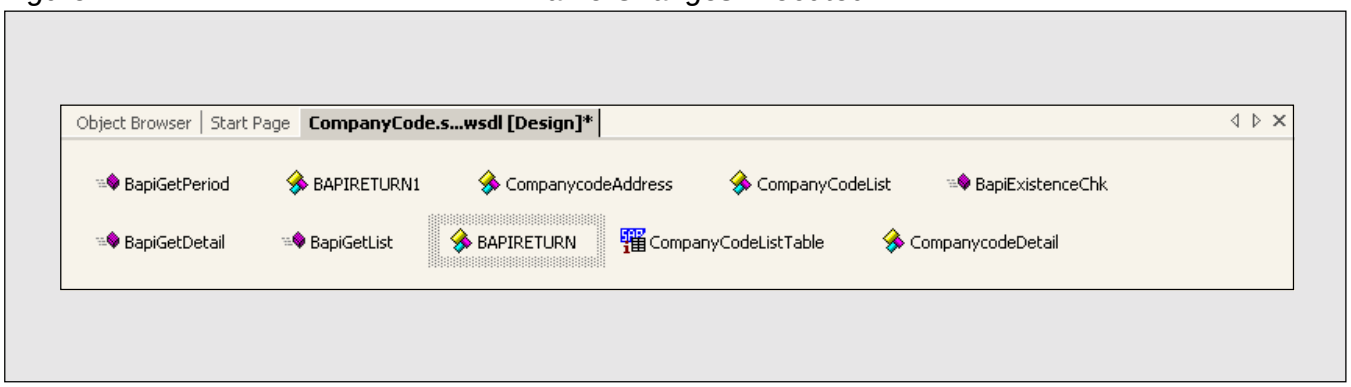


Figure 24 All Name Changes Executed



- Use meaningful names for the structure/table classes. If in doubt use the parameter names of the BAPIs.

So, according to my standard, I have changed the method name to "BapiGetList" in Figure 21. Applying the appropriate changes to the other three BAPIs yields **Figure 22**.

Next are the structure names. I change BAPI0002_1 to CompanyCodeList, which automatically changes BAPI0002_1Table to CompanyCodeListTable (**Figure 23**).

Figure 24 shows the names after applying similar changes to BAPI0002_2 and BAPI0002_3. BAPIRETURN and BAPIRETURN1 are okay, as far

as I am concerned (and remember, this is just my standard, you can do whatever you like).

You can also change the parameter names of the methods. To do this, select the appropriate method in the designer window and click on the parameters collection in the property window (see **Figure 25**). This will open the Parameter Collection Editor. **Figure 26** shows how I changed the parameter names. Optional parameters that are not relevant for your company can be removed altogether so that they no longer show up in the method signature.

Similarly, you can change the field names of the structure classes, but I have decided to keep the names from the SAP Data Dictionary.

When you are done with your changes, save the Designer window and rebuild the assembly.

Figure 25 Properties of a Method

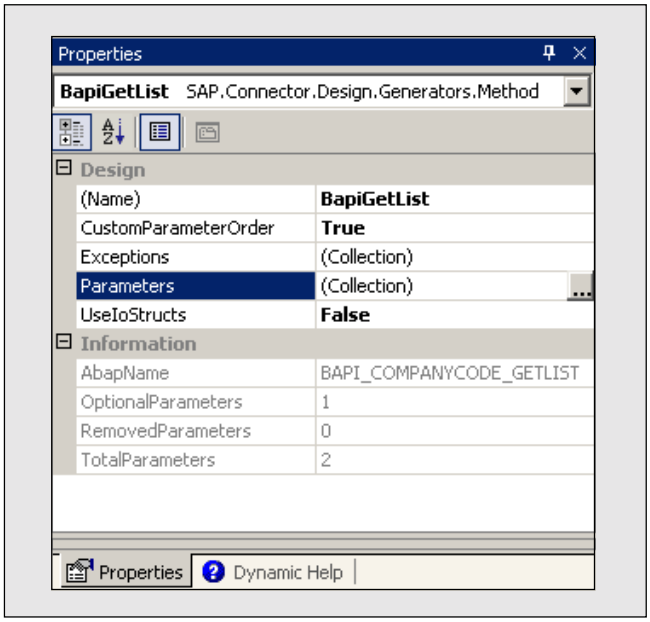
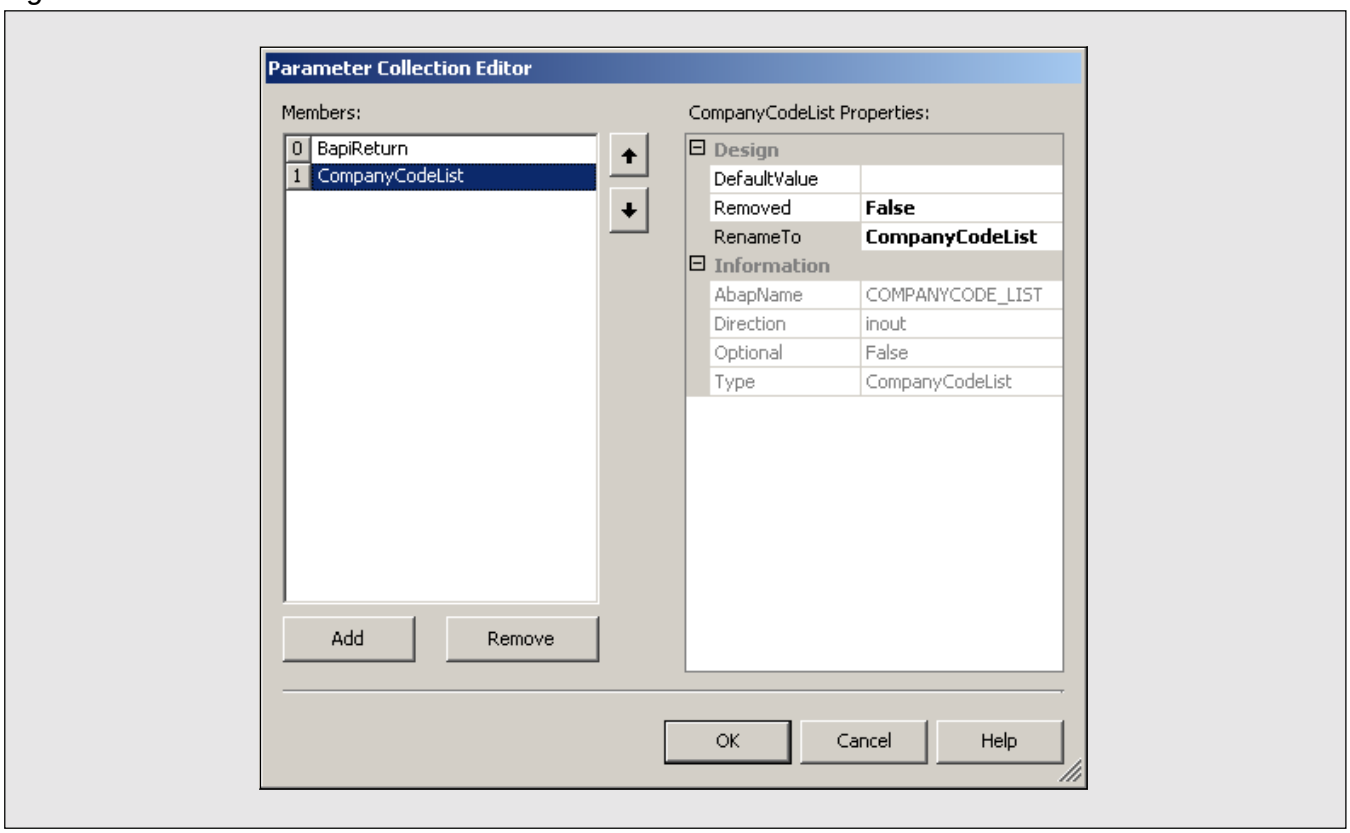


Figure 26 Parameter Collection Editor



Changing the client program to use the new proxy classes (in their own assembly now instead of part of the client program) is simple and left as an exercise to the reader.

Conclusion

The SAP .NET Connector is clearly superior to the SAP ActiveX Controls and the SAP DCOM Connector. Microsoft .NET is clearly the best software Microsoft ever produced. C# is one of the best programming languages ever invented.

So the conclusion is easy: if you build an SAP-enabled solution for the Microsoft platform only, use the SAP .NET Connector.

Thomas G. Schuessler is the founder of ARAsoft (www.arasoft.de), a company offering products, consulting, custom development, and training to a worldwide base of customers. The company specializes in integration between SAP and non-SAP components and applications. ARAsoft offers various products for BAPI-enabled programs on the Windows and Java platforms. These products facilitate the development of desktop and Internet applications that communicate with R/3. Thomas is the author of SAP's BIT525 "Developing BAPI-enabled Web Applications with Visual Basic" and BIT526 "Developing BAPI-enabled Web Applications with Java" classes, which he teaches in Germany and in English-speaking countries. Thomas is a regularly featured speaker at SAP TechEd and SAPPHIRE conferences. 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 tgs@arasoft.de.

Appendix A: The First Sample Program

```
using System;
using System.Drawing;
using System.Collections;
using System.ComponentModel;
using System.Windows.Forms;
using System.Data;

using SAP.Connector;

namespace CompanyCodeClient1 {
  public class Form1 : System.Windows.Forms.Form {

    private SAPCompanyCode1 sapCompanyCode;
    private BAPIRETURN bapiReturn;

    private System.Windows.Forms.Button button1;
    private System.Windows.Forms.DataGrid dataGrid1;
    private CompanyCodeClient1.BAPI0002_1Table bapI0002_1Table1;
    private SAP.Connector.Destination destination1;
    private System.ComponentModel.IContainer components;

    public Form1() {
    //
    // Required for Windows Form Designer support
    //
      InitializeComponent();
    }

    /// <summary>
    /// Clean up any resources being used.
    /// </summary>
    protected override void Dispose( bool disposing ) {
      if( disposing ) {
        if (components != null) {
          components.Dispose();
```

```
    }
  }
  base.Dispose( disposing );
}

#region Windows Form Designer generated code
/// <summary>
/// Required method for Designer support - do not modify
/// the contents of this method with the code editor.
/// </summary>
private void InitializeComponent() {
  this.components = new System.ComponentModel.Container();
  this.button1 = new System.Windows.Forms.Button();
  this.dataGrid1 = new System.Windows.Forms.DataGrid();
  this.bapI0002_1Table1 = new CompanyCodeClient1.BAPI0002_1Table();
  this.destination1 =
    new SAP.Connector.Destination(this.components);
  ((System.ComponentModel.ISupportInitialize)(this.dataGrid1))
    .BeginInit();
  ((System.ComponentModel.ISupportInitialize)
    (this.bapI0002_1Table1)).BeginInit();
  this.SuspendLayout();
  //
  // button1
  //
  this.button1.Anchor = ((System.Windows.Forms.AnchorStyles.Bottom
    | System.Windows.Forms.AnchorStyles.Left)
    | System.Windows.Forms.AnchorStyles.Right);
  this.button1.Location = new System.Drawing.Point(72, 274);
  this.button1.Name = "button1";
  this.button1.Size = new System.Drawing.Size(144, 24);
  this.button1.TabIndex = 0;
  this.button1.Text = "List Company Codes";
  this.button1.Click +=
    new System.EventHandler(this.button1_Click);
  //
  // dataGrid1
  //
  this.dataGrid1.Anchor = (((System.Windows.Forms.AnchorStyles.Top
    | System.Windows.Forms.AnchorStyles.Bottom)
    | System.Windows.Forms.AnchorStyles.Left)
    | System.Windows.Forms.AnchorStyles.Right);
  this.dataGrid1.DataMember = "";
  this.dataGrid1.DataSource = this.bapI0002_1Table1;
  this.dataGrid1.HeaderForeColor =
    System.Drawing.SystemColors.ControlText;
```



```
    this.dataGrid1.Location = new System.Drawing.Point(3, 2);
    this.dataGrid1.Name = "dataGrid1";
    this.dataGrid1.Size = new System.Drawing.Size(281, 270);
    this.dataGrid1.TabIndex = 1;
    //
    // bapI0002_1Table1
    //
    this.bapI0002_1Table1.AllowEdit = true;
    this.bapI0002_1Table1.AllowNew = true;
    this.bapI0002_1Table1.AllowRemove = true;
    this.bapI0002_1Table1.SupportsChangeNotification = true;
    //
    // Form1
    //
    this.AutoScaleBaseSize = new System.Drawing.Size(5, 13);
    this.ClientSize = new System.Drawing.Size(288, 301);
    this.Controls.AddRange(new System.Windows.Forms.Control[] {
      this.dataGrid1,
      this.button1});
    this.Name = "Form1";
    this.Text = "CompanyCodeClient1";
    this.Load += new System.EventHandler(this.Form1_Load);
    ((System.ComponentModel.ISupportInitialize)(this.dataGrid1))
      .EndInit();
    ((System.ComponentModel.ISupportInitialize)
      (this.bapI0002_1Table1)).EndInit();
    this.ResumeLayout(false);
  }
  #endregion

  /// <summary>
  /// The main entry point for the application.
  /// </summary>
   [STAThread]
  static void Main() {
    Application.Run(new Form1());
  }

  private void Form1_Load(object sender, System.EventArgs e) {
  }

  private void button1_Click(object sender, System.EventArgs e) {
    sapCompanyCode = new SAPCompanyCode1();
    destination1.AppServerHost = "hostname";
    destination1.SystemNumber = 0;
    destination1.Client = 400;
```

```
        destination1.Username = "userid";
        destination1.Password = "password";
        sapCompanyCode.Connection = new SAPConnection(destination1);
        sapCompanyCode.Bapi_Companycode_Getlist
          (out bapiReturn, ref bapI0002_1Table1);
      }
    }
  }
```


Appendix B: The Sales Order Type Conversion Program

```
using System;
using System.Drawing;
using System.Collections;
using System.ComponentModel;
using System.Windows.Forms;
using System.Data;

using SAP.Connector;

using De.ARAsoft.SAPdotNet;

namespace BapiConversion {
  public class Form1 : System.Windows.Forms.Form {
    private SAPProxy1 sapProxy;
    private BAPICONVRSTable conversionTable;
    private BAPIRET2Table bapiReturn;
    private Destination destination;
    private System.Windows.Forms.Button button1;
    private System.Windows.Forms.TextBox tbExternal;
    private System.Windows.Forms.Label label1;
    private System.Windows.Forms.Label label3;
    private System.Windows.Forms.TextBox tbInternal;

    /// <summary>
    /// Required designer variable.
    /// </summary>
    private System.ComponentModel.Container components = null;

    public Form1() {
      //
      // Required for Windows Form Designer support
      //
      InitializeComponent();
    }

    /// <summary>
```

```
/// Clean up any resources being used.
/// </summary>
protected override void Dispose( bool disposing )
{
  if( disposing ) {
    if (components != null) {
      components.Dispose();
    }
  }
  base.Dispose( disposing );
}

#region Windows Form Designer generated code
/// <summary>
/// Required method for Designer support - do not modify
/// the contents of this method with the code editor.
/// </summary>
private void InitializeComponent() {
  this.button1 = new System.Windows.Forms.Button();
  this.tbExternal = new System.Windows.Forms.TextBox();
  this.label1 = new System.Windows.Forms.Label();
  this.label3 = new System.Windows.Forms.Label();
  this.tbInternal = new System.Windows.Forms.TextBox();
  this.SuspendLayout();
  //
  // button1
  //
  this.button1.Location = new System.Drawing.Point(145, 4);
  this.button1.Name = "button1";
  this.button1.Size = new System.Drawing.Size(56, 28);
  this.button1.TabIndex = 0;
  this.button1.Text = "Convert";
  this.button1.Click +=
    new System.EventHandler(this.button1_Click);
  //
  // tbExternal
  //
  this.tbExternal.Location = new System.Drawing.Point(91, 6);
  this.tbExternal.Name = "tbExternal";
  this.tbExternal.Size = new System.Drawing.Size(44, 20);
  this.tbExternal.TabIndex = 1;
  this.tbExternal.Text = "";
  this.tbExternal.WordWrap = false;
  //
  // label1
  //
  this.label1.Location = new System.Drawing.Point(8, 8);
```



```
    this.label1.Name = "label1";
    this.label1.Size = new System.Drawing.Size(76, 20);
    this.label1.TabIndex = 2;
    this.label1.Text = "External type:"
    //
    // label3
    //
    this.label3.Location = new System.Drawing.Point(220, 8);
    this.label3.Name = "label3";
    this.label3.Size = new System.Drawing.Size(70, 20);
    this.label3.TabIndex = 4;
    this.label3.Text = "Internal type:";
    //
    // tbInternal
    //
    this.tbInternal.Location = new System.Drawing.Point(300, 7);
    this.tbInternal.Name = "tbInternal";
    this.tbInternal.ReadOnly = true;
    this.tbInternal.Size = new System.Drawing.Size(44, 20);
    this.tbInternal.TabIndex = 5;
    this.tbInternal.Text = "";
    this.tbInternal.WordWrap = false;
    //
    // Form1
    //
    this.AutoScaleBaseSize = new System.Drawing.Size(5, 13);
    this.ClientSize = new System.Drawing.Size(352, 37);
    this.Controls.AddRange(new System.Windows.Forms.Control[] {
      this.tbInternal,
      this.label3,
      this.label1,
      this.tbExternal,
      this.button1});
    this.Name = "Form1";
    this.Text = "Convert Sales Order Type";
    this.Load += new System.EventHandler(this.Form1_Load);
    this.ResumeLayout(false);
  }
  #endregion

  /// <summary>
  /// The main entry point for the application.
  /// </summary>
  [STAThread]
  static void Main() {
    Application.Run(new Form1());
  }
```

```
    private void Form1_Load(object sender, System.EventArgs e) {}

    private void button1_Click(object sender, System.EventArgs e) {
      destination = new Destination();
      destination.AppServerHost = "hostname";
      destination.SystemNumber = 0;
      destination.Client = 400;
      destination.Username = "userid";
      destination.Password = "password";
      sapProxy = new SAPProxy1();
      sapProxy.Connection = new SAPConnection(destination);

      conversionTable = new BAPICONVRSTable();

      BAPICONVRS conversionRow;
      conversionRow = new BAPICONVRS();
      conversionRow.Objname = "SalesOrder";
      conversionRow.Method = "CreateFromDat1";
      conversionRow.Parameter = "OrderHeaderIn";
      conversionRow.Field = "DOC_TYPE";
      conversionRow.Ext_Format = tbExternal.Text.Trim();
      conversionTable.Add(conversionRow);

      bapiReturn = new BAPIRET2Table();

      sapProxy.Bapi_Conversion_Ext2int1(ref conversionTable,
                                        ref bapiReturn);
      bool result =
        De.ARAsoft.SAPdotNet.StaticMethods.
        IsBapiReturnCodeOkay(bapiReturn);
      MessageBox.Show("The BAPI call was successful: " + result);

      tbInternal.Text = conversionTable[0].Int_Format;
    }
  }
}
```

