BAPI Programming Made Easy

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 134)

I have written a number of articles for this publication explaining most of the advanced issues of BAPI programming.[1] Many readers have asked for an introduction suitable for beginners, though. And BAPIs are becoming more and more useful. On one hand because the number of BAPIs keeps increasing with every release, on the other hand because new ways of utilizing BAPIs evolve, like web services and other XML-based communication technologies and the SAP Web Application Server. This article aims to address the demand for a BAPI introduction, but contains enough information that even readers who have already worked with BAPIs should benefit. While the sample program is written in Java, most of the presented information applies to all programming languages.

I will introduce the BAPI concept, show you how to investigate the features of individual BAPIs in the SAP system, and discuss a full-blown sample program that creates a sales order.

For this article, I had to make several choices, which I will enumerate and justify first:

- **Middleware/programming language:** Several choices (from SAP and third parties) exist as far as middleware is concerned. Since all middleware choices are more or less prejudging the language decision, these two issues should be addressed together. To completely justify my choice here would require a big article in itself, so I will just give a brief statement instead. I consider the

[1] Cf. the "Past Issues" section of **www.SAPpro.com** for a list of these articles. (See footnote 7 for a partial list.)

SAP Java Connector (JCo) the best middleware for BAPI programming. JCo is mature, fast, and provides complete coverage of all Remote Function Call (RFC[2]) features. The fact that JCo is — in my view — the best middleware is convenient because it automatically leads to the choice of Java as the programming language. Java is object-oriented. Java is more vendor-independent than any of its relevant alternatives. Java provides platform-independence. Java has become the most popular choice for creating browser-based Internet and intranet applications. In other words, JCo and Java[3] are a good solution for the largest number of customers.

- **Inclusion of non-BAPI technologies:** In addition to BAPIs, there are also non-BAPI RFMs (RFC-enabled Function Modules) and IDocs (Intermediate Documents). I have chosen to exclude both technologies for several reasons.

  There are not very many useful non-BAPI RFMs. Most of them are not well documented. Most of them are not officially released for customer use. This makes non-BAPI RFMs more difficult to use from an application standpoint and more expensive to use from a maintenance standpoint[4].

  Technically, using non-BAPI RFMs is not very much different from using BAPIs, so readers of this article should have little difficulty using them in their applications — once they have figured out how they work from an application standpoint.[5]

  IDocs are used for asynchronous communication with an SAP system. They are the major alternative to BAPIs. Sending and/or receiving IDocs is very different from calling BAPIs, though, and another large article would be required to deal with them.

- **Complexity of the sample project:** Most real-world applications need to update the SAP database, so I have chosen to use sales order creation as my example. This is a good choice because many customers actually want to build their own sales order front-ends and also because the business knowledge required is limited compared to, for instance, a Production Planning and Control (PPC) scenario. Finally, most readers will have ordered something on the Internet and therefore have some familiarity with the associated business concepts.

  The sample project does not come with a GUI, though. Building desktop or browser GUIs adds a lot of complexity and code and also has nothing whatsoever to do with BAPIs.

- **Use of proxies and additional software:** In order to keep things simple for beginners — without limiting the usefulness of the sample project — I have decided to use generated proxies and additional support software. That does not imply that you have to use these components in your own projects, but it is the only way to avoid the study of various comparatively complex issues and spend significant time[6] writing your own components to deal with these issues.[7]

---

[2] RFC is the protocol used for system-to-system communication in the SAP world.

[3] Java is not the best available programming language, though. In my opinion, Eiffel (cf. **www.eiffel.com**) is the best language, but Java is the best (more or less) vendor-independent and commonly used language.

[4] SAP can always change or even delete non-released RFMs.

[5] If you are using non-released SAP interfaces, always encapsulate access to these interfaces in a component, so that changes to the interface necessitate maintenance only to this one component, and not many application programs.

[6] At least several weeks, but months more realistically.

[7] You would need to read at least the following articles: "Simplifying BAPI Programming with Components" (November/December 1999), "Transaction Handling in SAP R/3 — What Every Programmer Needs to Know" (July/August 2000), "Enabling Point-and-Click Data Entry Assistance for Your BAPI Applications" (September/October 2000), "Everything a BAPI Programmer Needs to Know About the Business Object Repository" (January/February 2001), "Currencies and Currency Conversions in BAPI Programming" (September/October 2001), "Why You Need and How You Build BAPI Components" (March/April 2002), "BAPI Return Messages Made Easy" (May/June 2002). To order the issues in which these articles appear, see the form at the end of this issue or contact customer@WISpubs.com. In addition, there are other relevant articles available online at **www.SAPinsider.com** and the author's JCo tutorial, available on request from tgs@arasoft.de.

- **SAP release:** The number of BAPIs has grown significantly since they were first introduced in 3.1H. From less than 200 the number has now increased to about 2,000 in 4.6C. Hence I will use 4.6C in my sample project. Using a previous release for BAPI-enabled solutions is also possible, but may require more effort because of missing functionality in earlier releases.

Introduction to BAPIs

SAP's Business Application Programming Interfaces (BAPIs) are the primary technology for interfacing external systems with an SAP system. These external systems can be other SAP systems, web servers, non-SAP inhouse applications, desktop applications to facilitate user interaction with the SAP system, business partner applications, applications for hand-held devices like PDAs and mobile phones, etc.

BAPIs are basically externally callable functions that are written in ABAP and defined as methods of object types in the SAP Business Object Repository.

BAPIs offer the following features and advantages:

- **Official interfaces:** BAPIs are usually released for customer use. Once a BAPI has been released it is guaranteed to be upward-compatible, which reduces the maintenance cost of an application using BAPIs because your application will not require changes when you upgrade your SAP system[8]. The exception to this rule are new BAPIs that are sometimes introduced in a non-released state so that customers can test them and SAP can make required changes based on customer feedback. You should only use released BAPIs, unless you absolutely need the new functionality and are willing to change your application if SAP makes changes to the BAPI. I will show you later how to distinguish between released and non-released BAPIs.

[8] See below for details on upward-compatibility.

- **Synchronicity:** BAPIs are invoked synchronously. The client program sends parameters to SAP, and the BAPI executes and then sends parameters back. One of the parameters coming back from a BAPI call (the *Return* parameter) tells the client program whether the BAPI executed successfully or why it failed.

✓ *Asynchronous BAPIs*

If you want to interface two SAP systems through BAPIs it is sometimes desirable to use asynchronous communication for updates of remote systems and synchronous communication for everything else. SAP supports this by allowing you to generate an IDoc message type for an update BAPI. An example of this strategy is the decoupling of the HR component and the rest of the R/3 system. See the SAP online documentation for details.

- **Flexibility:** BAPIs are not limited in what they can accomplish. A BAPI can do simple data retrieval or invoke complex processes like sales order creation. Customers and SAP partners can even develop their own BAPIs.

- **Commonality:** BAPIs should follow the rules defined in SAP's *BAPI Programming Guide*[9]. This makes it easy to use additional BAPIs once you have learned how to use BAPIs at all.[10]

- **Documentation:** BAPIs should contain sufficient documentation so that a developer who understands the application requirements of a project should be able to use the relevant BAPIs

[9] This text is part of the SAP online documentation.

[10] Some of the complexities of using BAPIs in the real world originate from the fact that not all BAPIs actually follow the rules. See the articles listed in footnote 7 for details. The software components used in the sample project solve most of the issues introduced by the non-conforming BAPIs.

successfully. In most projects, though, the application that invokes the BAPIs is not written by application specialists, but, for instance, by web developers in Java. It is a good idea to have access to one or more experienced SAP application consultants if you do not have the necessary application skills yourself. If a BAPI is not documented well, you should let SAP know about this. Bad documentation should be considered a software bug.

- **Object-orientation:** The BAPIs are defined in the SAP Business Object Repository (BOR) as methods of object types[11]. This means that all BAPIs for an entity like sales order are available as methods of an object type, in this case, the SAP business object type *SalesOrder*. This fact allows us to generate proxy classes for Java so that we use the object-oriented approach in our complete application, including the access to the SAP functionality.

- **Easy to find:** As opposed to non-BAPI RFMs, BAPIs can easily be found using the online BAPI Explorer (transaction code BAPI) or the SAP Interface Repository at **http://ifr.sap.com**. I will introduce the BAPI Explorer and how to use it to look up the important features of a BAPI in the subsequent section.

- **Upward-compatible:** A released BAPI should not be changed[12] by SAP in any way that would prevent it from being upward-compatible. The following changes to a BAPI's interface are allowed since they do not necessitate changes to a client application: new export parameters, new optional import parameters, and new fields at the end of existing structure or table parameters.[13]

  If SAP needs to make incompatible changes, this is handled in the following manner: A new BAPI is created, the name of which is derived from the old BAPI's name by adding a new number or incrementing an existing one. The successor to *SalesOrder.CreateFromData*, for instance, was called *SalesOrder.CreateFromDat1*, then came *SalesOrder.CreateFromDat2*. The old BAPI is declared *obsolete*[14]. Obsolete BAPIs are guaranteed to exist and work from the release in which they were declared obsolete up to (but not including) the next functional release plus one. I know this sounds complicated, so here is an example:

  > A BAPI declared obsolete in 4.0A will at least work up to (and including) 4.5B. The next functional release after 4.0A was 4.5A, the next after that 4.6A. 4.5B was a maintenance release for 4.5A, hence the BAPI was still valid.

  BAPIs may continue to exist and work past their minimum guaranteed validity (and most obsolete BAPIs are actually still around in 4.6C), but there is no guarantee for that. Since SAP publishes only about one functional release per year, developers of BAPI-enabled applications have at least two years during which they do not have to worry about maintenance.

- **Authority checks:** BAPIs contain the required authorization checks from an SAP application standpoint. Only users with appropriate authorizations will be able to successfully invoke a BAPI. The client program does not have to worry about dealing with these complex issues.

- **Commit handling:** Most BAPIs do not cause any changes to the database. If we invoke *Customer.GetDetail*, for example, the database is accessed to retrieve information, but no change takes place. BAPIs like *SalesOrder.CreateFromDat1*, on the other hand, will update the database. All update BAPIs before 4.0A had to contain their own COMMIT

---

[11] An *object type* in SAP is equivalent to a *class* in Java.

[12] I would be much more comfortable if I could say *will not ever be changed* but there are some rare occurrences when SAP has broken this rule.

[13] See below for more details on BAPI parameters.

[14] Java programmers know this as *deprecated*.

✓ *How to Find the Required Authorizations*

When you deploy a new BAPI-based application, you need to make sure that the future users of the application have the required authorizations to invoke the BAPIs used in your application. These authorizations should be listed in the BAPIs' documentation. Often this is not the case, though. Should this situation arise, the best approach is to contact your security administrator and ask for the authorizations needed for the online transaction equivalent to each BAPI. If you want to call SalesOrder.CreateFromDat1, for example, you need exactly the same authorizations as you would for creating a sales order in SAPGUI, using transaction code VA01.

WORK statement, thus making BAPI programming stateless. In 4.0A, the rule was changed. All BAPIs since 4.0A are not supposed to contain their own COMMIT WORK statement, but instead allow the client program to decide when all updates executed so far should be written to the database in the asynchronous update task of the SAP system. You can use the *BapiService.TransactionCommit* BAPI to commit your changes.

- **Middleware-independent:** BAPIs can be invoked within an SAP system, from a different SAP system, and from many external platforms, including various Unix and Win32 flavors. Many programming languages can be utilized, including Java, C, C#, C++, and Visual Basic.

As you could see in this section, there are several good reasons to consider BAPIs first in your integration projects. But if you would prefer asynchronous communication with SAP, you should take a closer look at IDocs. In many projects, you will be able to use both BAPIs and IDocs. A sample scenario would be an Internet sales application that has its own customer and product database tables that are updated from SAP through IDocs, but uses BAPIs for the actual sales order creation, status lookups, etc.

✓ *Are You Committed?*

Not all BAPIs in 4.0A and later follow the new rule. Some BAPIs will invoke COMMIT WORK. Others have a parameter that allows you to choose whether they should commit or not. There is no formal definition of the commit behavior of any given BAPI. If you are lucky, the documentation will tell you. Otherwise you will simply have to test it yourself. Invoke the BAPI without issuing a commit call yourself and then check the SAP system to find out whether the appropriate change has happened in the database.

In theory, being in control of the commit yourself allows you to combine multiple updates into one Logical Unit of Work (LUW). There is an exception to this, though. It is not possible, for example, to add a new customer and create a sales order for this customer in the same LUW. In addition, calling the customer create BAPI, then the commit BAPI, and then the sales order create BAPI, all in rapid succession, will often not work. The reason is that the asynchronous update task started by the commit BAPI in order to add the new customer to the database may not have finished by the time you invoke the sales order create BAPI. To avoid situations like this, the BapiService.TransactionCommit BAPI since 4.5A has a Wait parameter that allows you to specify that the commit BAPI should wait for the completion of the update task (COMMIT WORK AND WAIT). This option slows down the system to some extent, though, and should only be used when this is really required as in the example given.

Finally, there is also a BAPI called BapiService.TransactionRollback, which can be used to roll back any update activities before the commit has happened.

The SAP BAPI Explorer

To find BAPIs in the SAP system and to study their parameters, it is a good idea to turn to the SAP system and use the BAPI Explorer (transaction code BAPI). **Figure 1** shows the initial output that you will see when you invoke the BAPI Explorer. Note the message displayed at the bottom of the screen telling you that by default only released BAPIs are displayed. As explained above, sticking to released BAPIs is a good idea unless you absolutely need the new functionality offered by an as-yet unreleased version of a BAPI. You can click the filter icon to switch on the display of all BAPIs, including unreleased ones. **Figure 2** shows the pop-up window that lets you make the change.

Figure 1 displays the application hierarchy tree in the left pane. If you are familiar with this hierarchy, you can drill down to the object types and look at their associated BAPIs. **Figure 3** is a screenshot of the BAPI Explorer listing the BAPIs of object type *SalesOrder*. The *CreateFromDat1* BAPI has been selected. In the right pane, a lot of useful information

Figure 1 The SAP BAPI Explorer

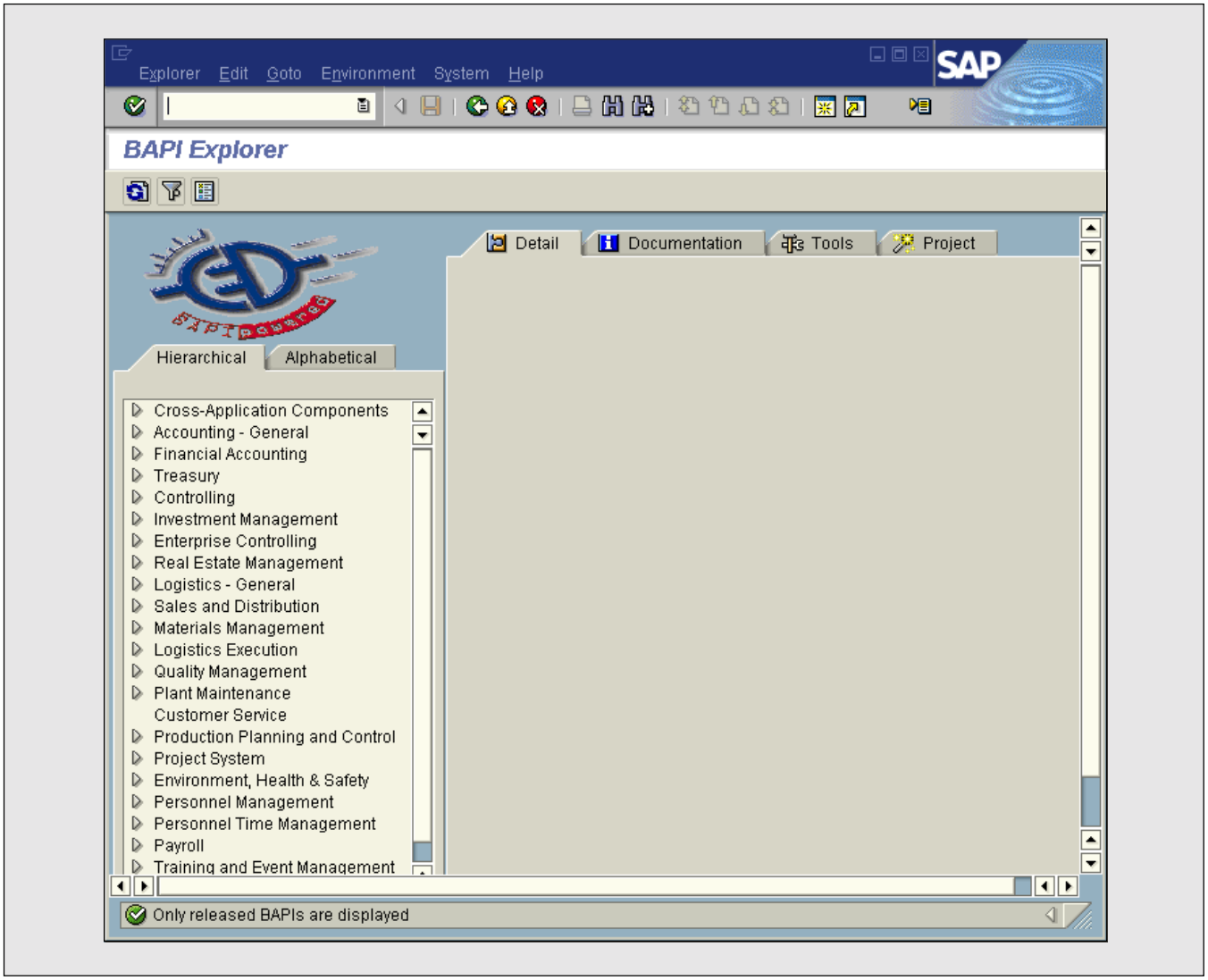
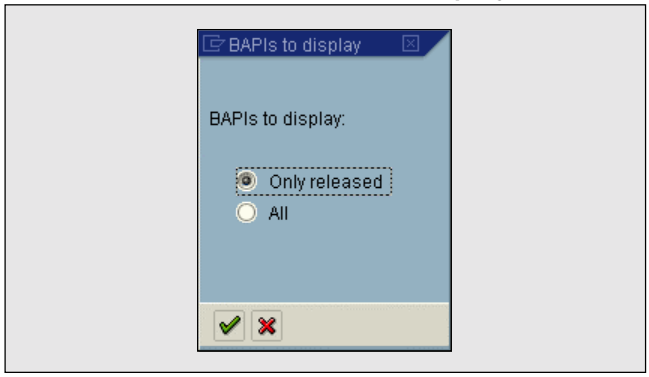


Figure 2 Pop-Up to Select Whether All BAPIs Should Be Displayed



can be found. Let me discuss the most important fields:

- This BAPI was added in release 4.0A (field "New in Release").
- The BAPI is released (field "Release status").
- The BAPI is obsolete since release 4.6C[15] (indicated by the stop sign and the text "Caution: Method is obsolete as of Release 46C !"). We will nevertheless use it in the sample program

[15] All screenshots were taken on a 4.6C system.

Figure 3 The BAPIs for Object Type SalesOrder

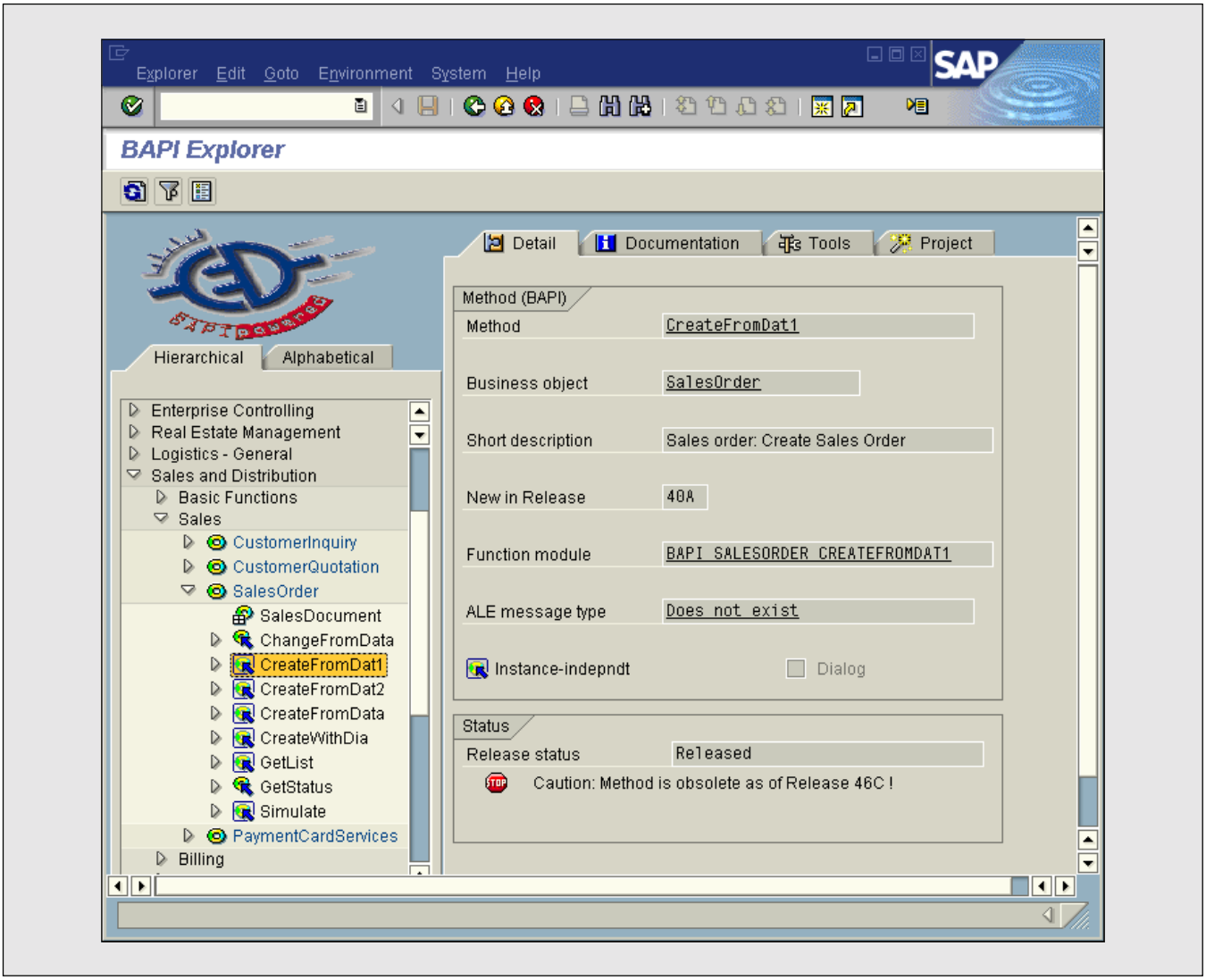
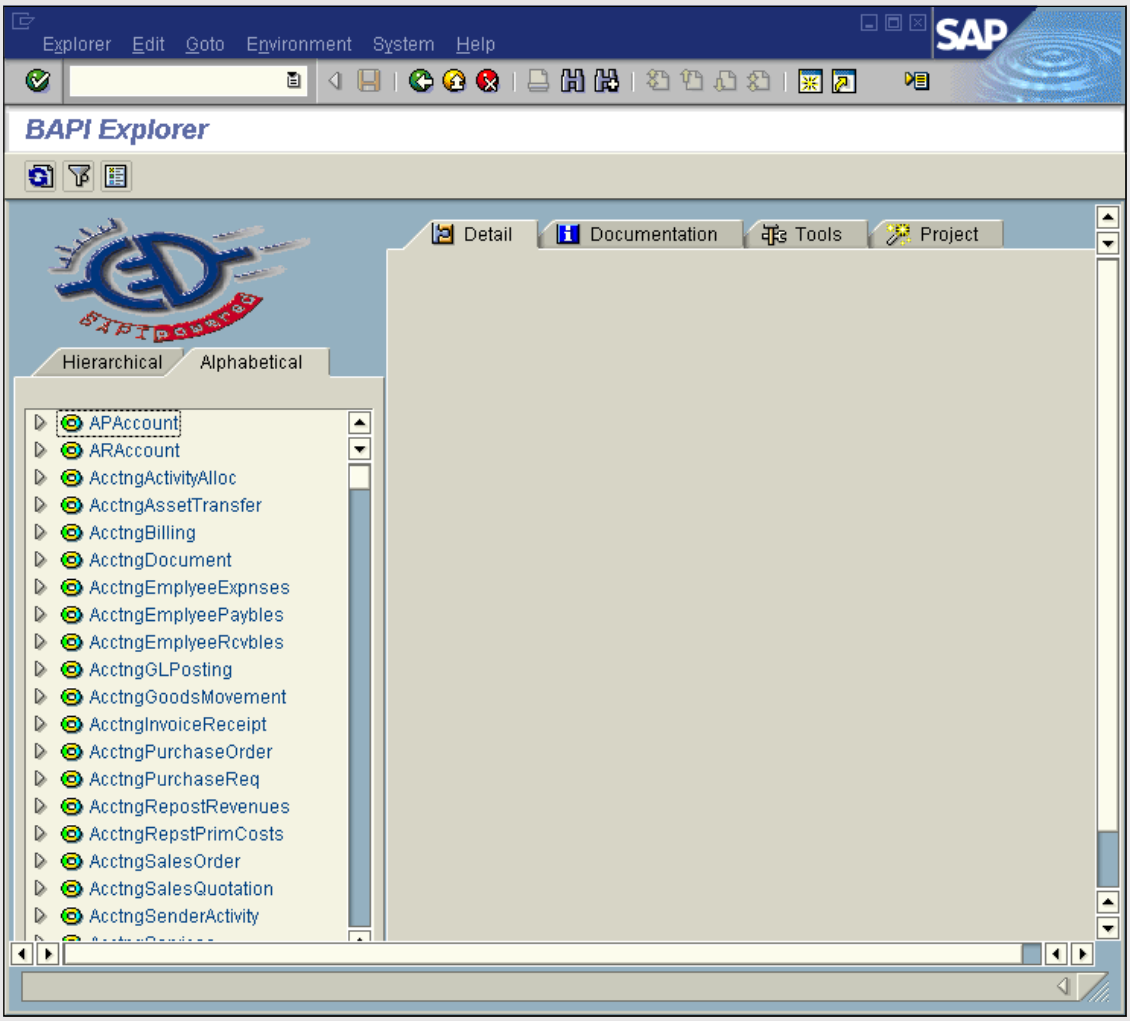


Figure 4 Listing the Object Types with BAPIs Alphabetically



introduced later since it is much easier to use than *CreateFromDat2* and will be supported for quite some time.[16]

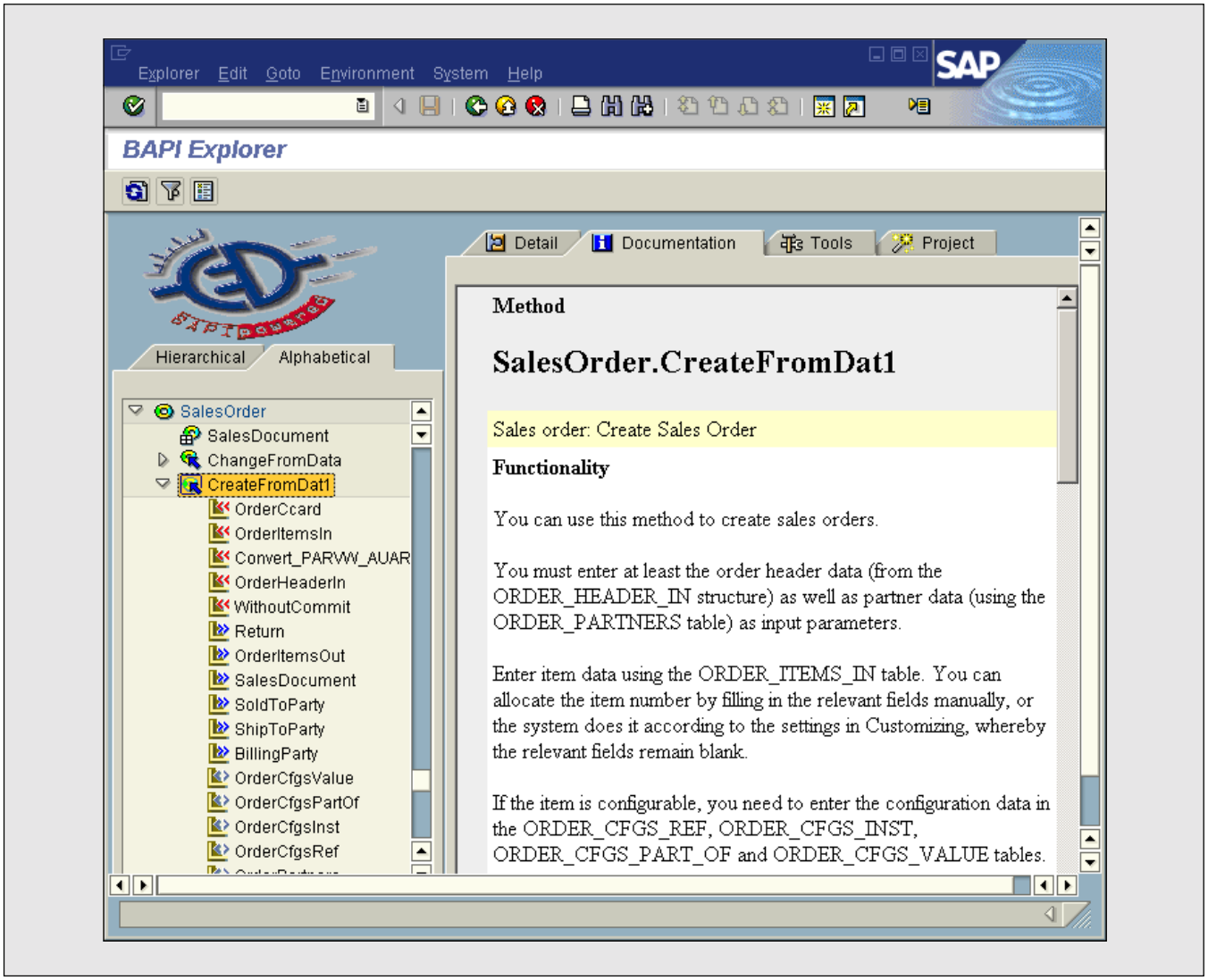
- The BAPI is *instance-independent* ("Instance-indepndt"). In most books on object-orientation, the term *class method* would have been used instead. If a BAPI is instance-independent, we do not have to set the key fields before calling it.

[16] The 4.7 version of R/3 has not even been released yet.

✓ *Tired of Hierarchies?*

You can look at the BAPIs without having to navigate through the application hierarchy tree. If you click on the "Alphabetical" tab in the left pane of the BAPI Explorer, a simple, alphabetical list of all object types that have BAPIs is displayed (see ***Figure 4****). Since there are just over 100 such object types, most people prefer to use the alphabetical list instead of having to search through the hierarchy.*

Figure 5 *The Documentation of a BAPI*



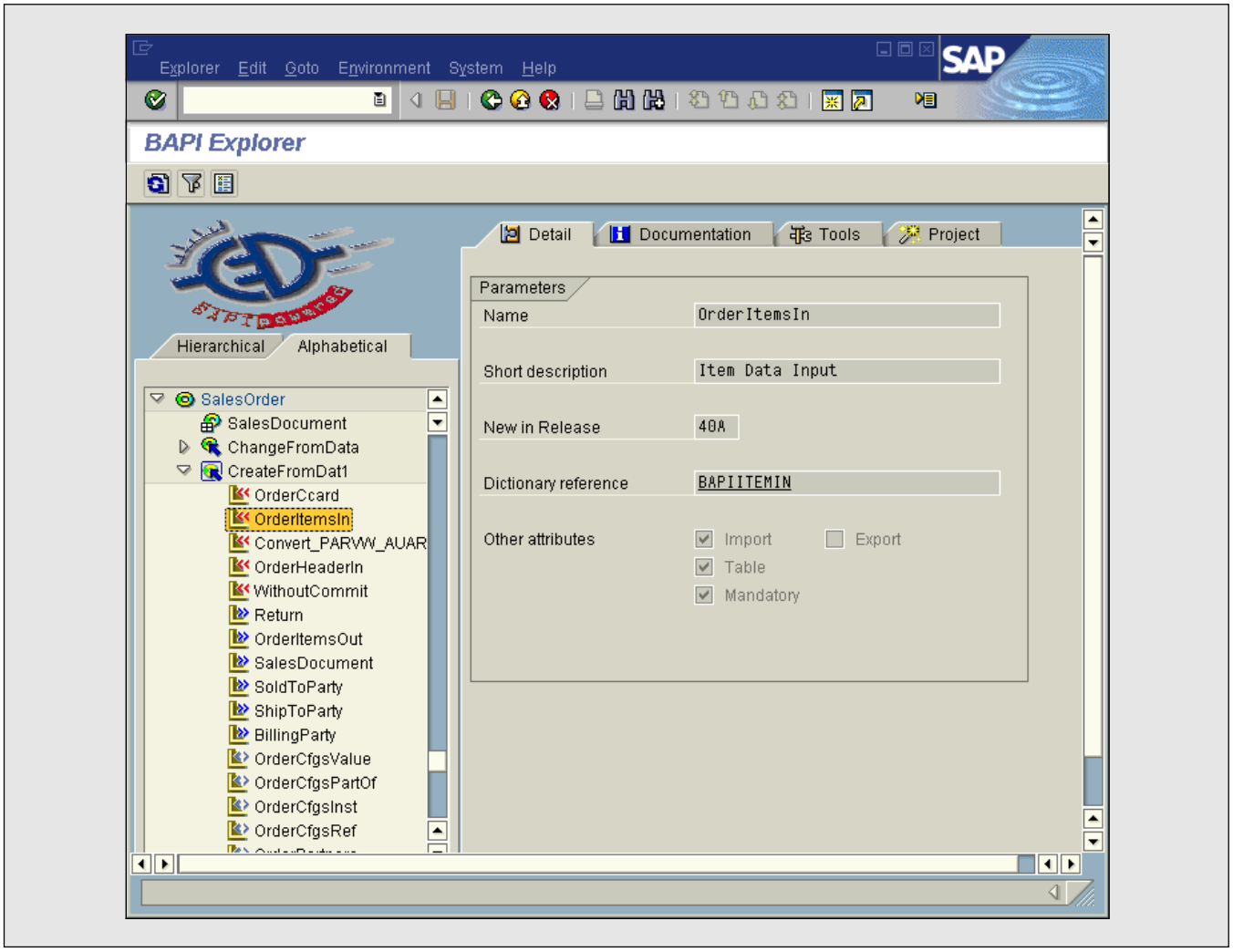
The key fields of an object type are shown in the left pane of the BAPI Explorer, directly underneath the name of the object type. The icon associated with key fields looks like a table symbol[17] in front of a lifesaver and denotes the link between the relational database world (organized in tables) and the object world. The *SalesOrder* object type has one key field, by the name of *SalesDocument*.

[17] If you can read Chinese characters, the table symbol will remind you of the character for (rice) field or paddy.

More BAPI Details

Whenever you want to take a closer look at a BAPI, for example, to establish whether it satisfies the requirements of your application, you should first study its documentation. **Figure 5** displays the documentation for *SalesOrder.CreateFromDat1*. In most cases, all available documentation for a BAPI is shown when you select the BAPI in the left pane and click on the "Documentation" tab in the right pane. Sometimes, though, additional documentation is available for individual parameters, so you should

Figure 6 *Parameter Information*



always check whether that is the case by selecting the individual parameters in the left pane while the documentation tab is displayed in the right pane. Making use of all available information will save you a lot of effort when you write and test your application later. Several examples will be discussed throughout the remainder of this article.

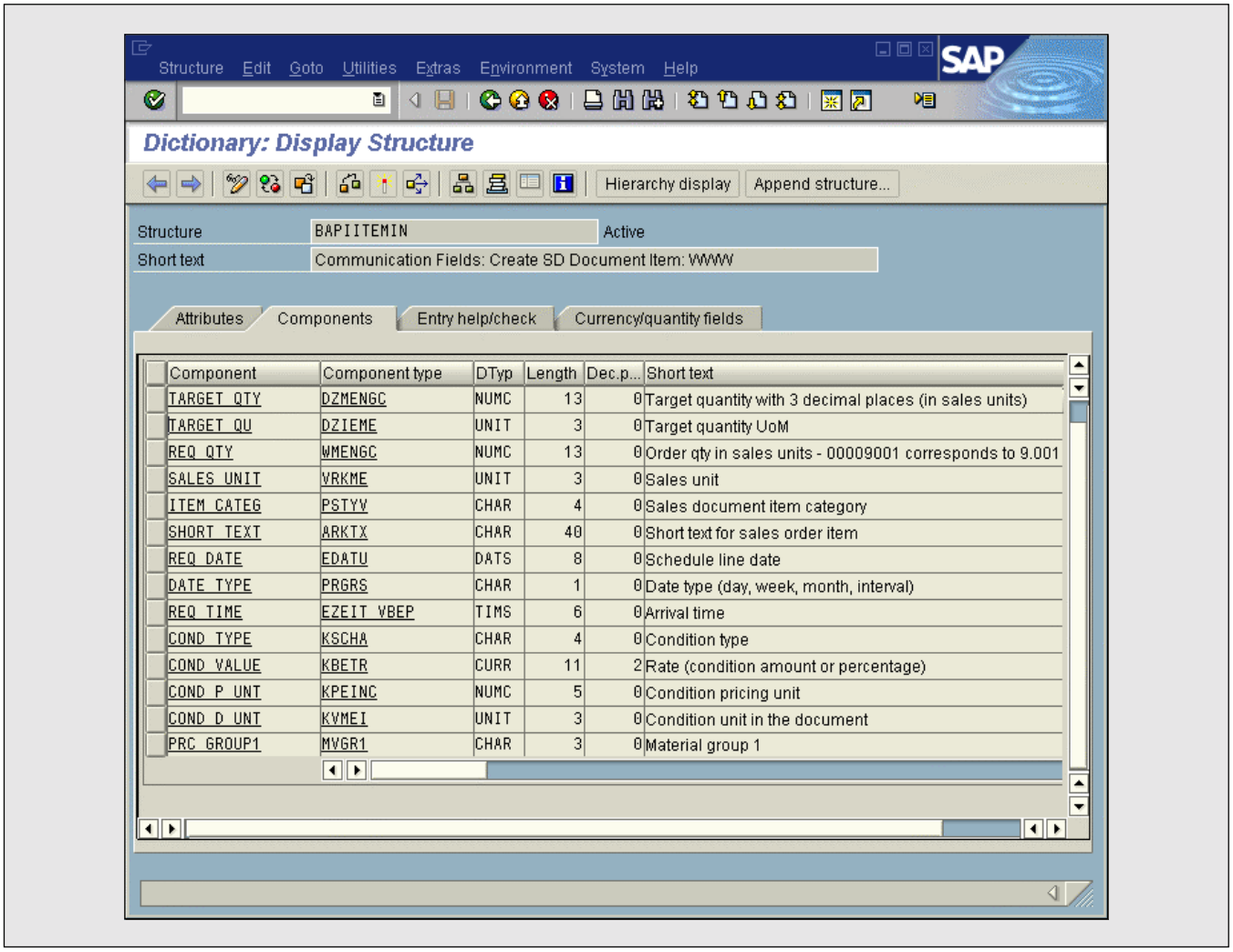
Let us now look at the parameters of a BAPI (see **Figure 6**). Parameters can be classified in different ways:

- A parameter can be a *simple field*, a *structure*, or a *table*. A structure parameter is a group of simple fields. A table contains zero or more rows, each containing one or more simple fields.[18] To find out whether a parameter is a table or not, you look at the "Table" checkbox in the right pane of Figure 6. But how do we distinguish between simple field and structure parameters? Field "Dictionary reference" contains the answer, albeit in a non-obvious way: If the field contains a hyphen, the parameter is a simple field, otherwise it is a structure.

To display more information about a parameter, you simply double-click the contents of the

[18] A structure is therefore like a table with exactly one row.

Figure 7 *Dictionary Information for a Parameter*



"Dictionary reference" field ("BAPIITEMIN"). **Figure 7** contains the details for the *OrderItemsIn* parameter selected in Figure 6. We will discuss this later.

- There are import and export parameters. Import parameters are sent by the client program to SAP, export parameters are sent back to the client program. Table parameters, but not structure or simple field parameters, can be import *and* export parameters. Look at the "Import" and "Export" checkboxes in Figure 6 to find out whether a parameter is sent to the SAP system, from the SAP system, or both.

- A parameter can be mandatory (checkbox "Mandatory") or, otherwise, optional. This category is only important for import parameters, all export-only parameters are always optional. Depending on the requirements of your client application you may have to use some of the optional import parameters, in other words a parameter may be technically optional, but still required in a given application. An example would be *SalesOrder.CreateFromDat1*, which has optional parameters for configurable materials that must be used if your sales order contains such materials.

Parameters in the Dictionary

The information available in the SAP Data Dictionary[19] for each parameter is very important. When using structure and table parameters, we need to know which fields they contain. Figure 7 lists some of the fields of the *OrderItemsIn* parameter. I have scrolled down the field list because I want to discuss the "REQ_QTY" field (the third field shown in Figure 7) as an example. First, we can find out its data type ("NUMC", which means numerical character), length (13), and number of decimals (zero). For a complete list of the dictionary data types, please see the SAP online documentation. The length specifies the internal length, i.e., how many bytes this field occupies in computer memory, not necessarily how many character positions are needed to display it. For some data types (e.g., "CHAR", which means character and is a fixed-length string) the internal length and the display length are identical. For most data types, this is not the case. A "DATS" (date) field, for instance, has an internal length of eight bytes, but is usually displayed with additional formatting characters ("/" in the United States, for example), requiring a total of 10 display characters.

In addition to looking up this information, you should also read the description of the field ("Short text"). And for the "REQ_QTY" field, we notice something peculiar: This field is supposed to contain a quantity, yet it uses data type "NUMC", which is usually used for information that consists of digits, but whose value will never be used in any computation. Employee numbers in SAP, for example, use this data type. The description for the "REQ_QTY" field tells us how we need to use it. The number we put into it will be interpreted assuming that the last three digits of the value contain the decimals of the quantity, in other words, if we want to order one widget, we have to specify the quantity "1000". This is definitely vital information if we want to use the *SalesOrder.CreateFromDat1* BAPI successfully!

[19] Sometimes called the *ABAP Dictionary*.

While most descriptions do not contain surprises of that magnitude, it is always a good idea to read the description texts.

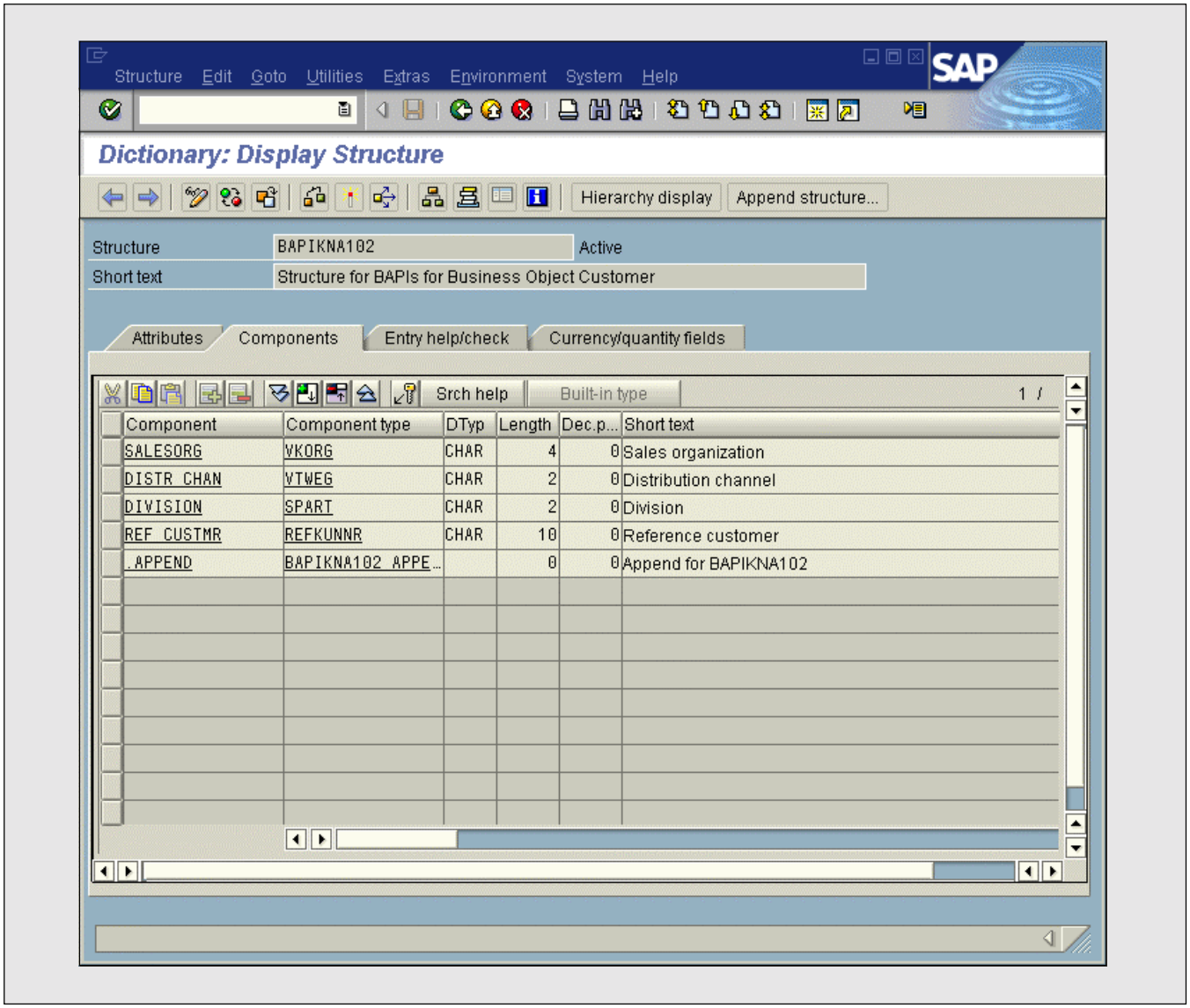
Which brings us to another important issue. The "BAPIITEMIN" structure in the dictionary (which defines the fields of parameter *OrderItemsIn*) contains quite a few fields, 87 to be exact. How do we find out which of these fields have to be populated in our application? If the BAPI follows the rules, its documentation should contain a list of the fields that are absolutely required. What if that list does not exist or we need additional fields, but do not know in which combination they have to be used? Here you will need help from an application specialist. In any BAPI project you should have the pertinent application know-how!

Figure 7 shows the fields of a table parameter, so we are interested — at least potentially — in all the fields of this parameter. The same would apply to a structure parameter. For a simple field parameter, on the other hand, we would expect to see only information for this particular field. What we will see in reality, though, is the complete dictionary structure in which the field is defined. **Figure 8** shows the dictionary information displayed when we double-click on the "Dictionary reference" of the *Division* parameter of the *Customer.GetDetail* BAPI. If the dictionary structure contained just one field (which is the case for the simple field parameters of *SalesOrder.CreateFromDat1*, check it out in the BAPI Explorer yourself), we would have no problem. But Figure 8 lists four fields (plus an empty append structure). The one we are interested in ("DIVISION") is highlighted by the fact that the cursor is in this field, but the screenshot does not show that very clearly. You will have no difficulty in the online system, though.

To Convert or Not to Convert

There is one last technicality that you need to know about, the issue of field conversions. When you use

Figure 8 Dictionary Information for a Simple Field Parameter



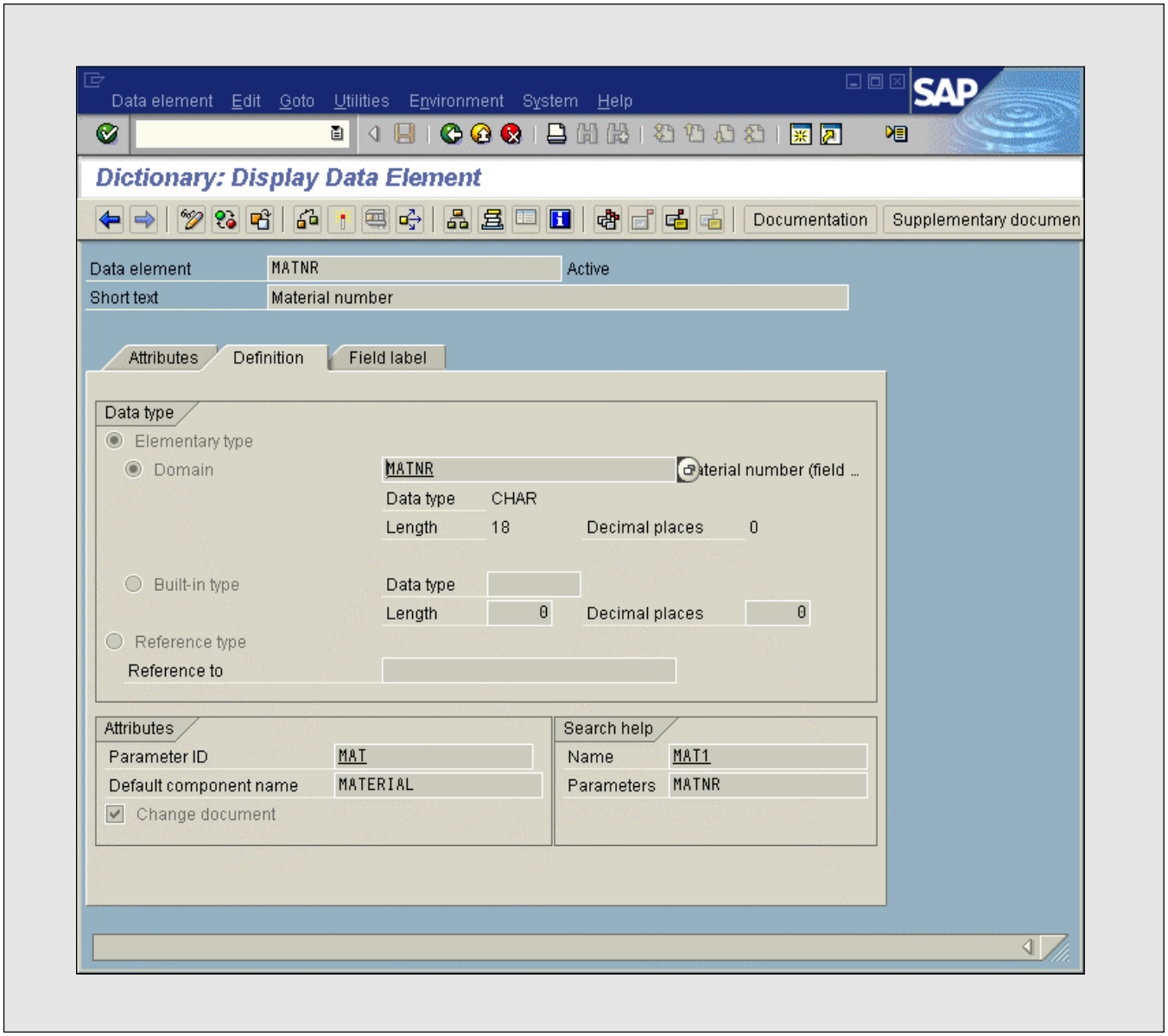
SAPGUI, many fields are converted between their external (user display) and internal (database) representations. Examples are customer and material numbers, order types, units of measure, and many other codes. A customer number that the user enters (and wants displayed) as "1400" is internally stored as "0000001400". A unit of measure that an English-speaking user enters (and wants displayed) as "PC" (for piece) is stored in its German translation ("ST") in the database.

As opposed to SAPGUI, the BAPIs do not perform these conversions, so we need to do it for them.[20]

How do we figure out whether a field uses conversion? In Figures 7 and 8 you see a column called "Component type". When you double-click in that column for a particular field, its data element

[20] For more details, see my article "BAPI Conversion Issues" at **www.SAPinsider.com** (*Article Archives → 2001 → January-March 2001*).

Figure 9 *Data Element Information*

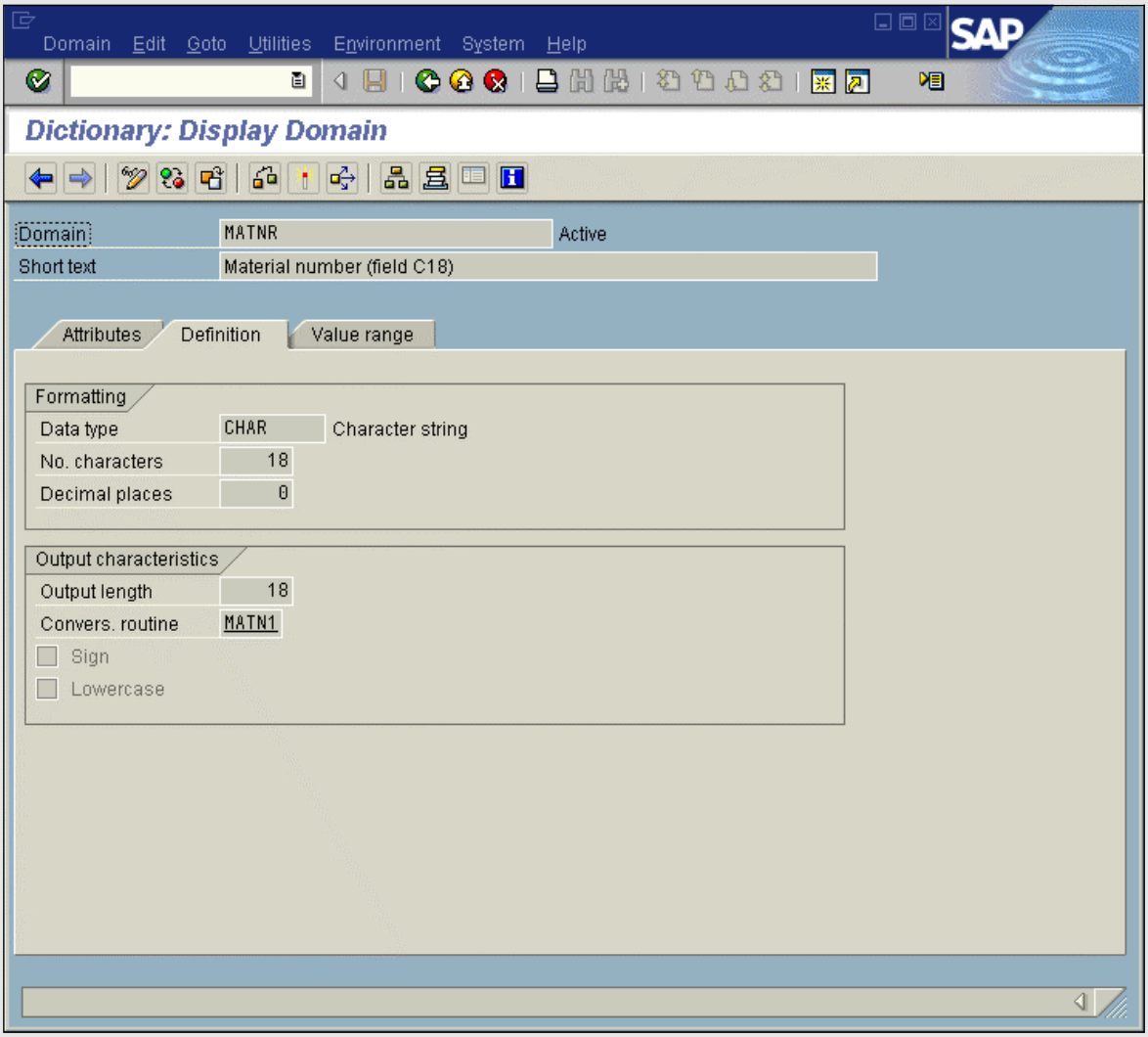


information is displayed. **Figure 9** shows the "MATNR" data element upon which the "MATERIAL" field in the *OrderItemsIn* parameter of the *SalesOrder.CreateFromDat1* BAPI is based. Double-clicking the "Domain" field takes you to **Figure 10**, the domain information. Here field "Convers. routine" is empty if no conversion takes place and otherwise contains the name of the conversion routine, in this case "MATN1".

A Sales Order Creation Sample Program

We now know enough to build a program to actually create a sales order in SAP. The program shown in **Listing 1** should work with any system containing an IDES client (of course the system and user information needs to be adjusted).

Figure 10 *Domain Information*



Listing 1: Creating a Sales Order

```
001 public class DemoSalesOrder {
002
003  JCO.Client mConnection;
004  JCoRepository mRepository;
005  JCoContext context;
006
```

(continued on next page)

(continued from previous page)

```
007  public DemoSalesOrder() {
008    try {
009 // Change the logon information to your own system/user
010      mConnection =
011         JCO.createClient(
012           "001",                // SAP client
013           "<userid>",           // userid
014           "*****",              // password
015           "EN",                 // language
016           "<hostname>",         // application server host name
017           "00");                // system number
018       mConnection.connect();
019       mRepository = new JCoRepository(mConnection);
020
021       context = new JCoContext(mConnection, mRepository);
022       context.startSapTransaction();
023
024       SAPSalesOrder salesOrder = new SAPSalesOrder(context);
025       SAPSalesOrder.CreateFromDat1Request request =
026         salesOrder.getRequests().createCreateFromDat1Request();
027
028       request.getWithoutCommit().setValue("X");
029
030       SAPSalesOrder.CreateFromDat1Request.OrderHeaderIn orderHeader
031         = request.getOrderHeaderIn();
032       orderHeader.setSALES_ORG("3000");
033       orderHeader.setDISTR_CHAN("10");
034       orderHeader.setDIVISION("00");
035
036       String userEnteredOrderType = "OR";
037       orderHeader.setDOC_TYPE(userEnteredOrderType);
038       orderHeader.setDOC_TYPE
039         (mRepository.getInternalValue
040         (orderHeader.getJCoFields().getDOC_TYPE()));
```

The subsequent discussion concentrates on the generic BAPI programming issues, not on the Java programming language or the technicalities of JCo.[21]

SalesOrder.CreateFromDat1 has quite a few parameters. Fortunately, for a simple sales order only a few are required: the sales order header (parameter *OrderHeaderIn*), the line items (parameter *OrderItemsIn*), and the customer information (parameter *OrderPartners*).

Let us take a look at the most important parts of the sample program. To facilitate this, I have added line numbers to the source code.

[21] If you would like a free JCo tutorial, please send me an email.


```
041
042       orderHeader.setPURCH_NO("Purchase Order 1");
043
044       SAPSalesOrder.CreateFromDat1Request.OrderPartners
045         orderPartners = request.getOrderPartners();
046       orderPartners.appendRow();
047       orderPartners.setPARTN_ROLE("AG");
048       String userEnteredCustomerNumber = "30001";
049       orderPartners.setPARTN_NUMB(userEnteredCustomerNumber);
050       orderPartners.setPARTN_NUMB
051         (mRepository.getInternalValue
052         (orderPartners.getJCoFields().getPARTN_NUMB()));
053
054       SAPSalesOrder.CreateFromDat1Request.OrderItemsIn orderItems
055         = request.getOrderItemsIn();
056       orderItems.appendRow();
057       orderItems.setMATERIAL("M-08");
058       orderItems.setMATERIAL(
059         mRepository.getInternalValue
060         (orderItems.getJCoFields().getMATERIAL()));
061       orderItems.setREQ_QTY("1000");   // Really: 1.000
062
063       orderItems.appendRow();
064       orderItems.setMATERIAL("DCC-12");
065       orderItems.setMATERIAL
066         (mRepository.getInternalValue
067         (orderItems.getJCoFields().getMATERIAL()));
068       orderItems.setREQ_QTY("15000");  // Really: 15.000
069
070       SAPSalesOrder.CreateFromDat1Response response =
071         salesOrder.createFromDat1(request);
072
073       BapiMessageInfo returnMessage =
074         new BapiMessageInfo
```

(continued on next page)

In lines 10-17 we create a *JCO.Client* object to hold a connection with SAP and in line 18 we connect to the system.

In line 19 we create a JCo repository that will contain the metadata for the BAPIs we are going to invoke. JCo retrieves the required metadata dynamically from SAP.

In lines 21-22 we create a *JCoContext* object that facilitates the commit handling of our program (see the discussion of lines 77 and 88 below).

We are now ready for the specific application logic. In line 24 a proxy object is instantiated and lines 25-26 show the creation of the request

(continued from previous page)

```
075         (response.getReturn().getJCoStructure());
076       if ( ! returnMessage.isBapiReturnCodeOkay() ) {
077         context.rollbackSapTransaction();
078         System.out.println
079           (returnMessage.getFormattedMessage());
080         System.out.println
081           ("--- Documentation for error message: ---");
082         String[] documentation =
083           mRepository.getMessageDocumentation(returnMessage);
084         for (int j = 0; j < documentation.length; j++) {
085           System.out.println(documentation[j]);
086         }
087       } else {
088         context.commitSapTransaction();
089         System.out.println
090           ("Sales order number of new sales order is: " +
091           salesOrder.getKeyFields().
092           getSalesDocument().getString());
093       }
094     }
095     catch (Exception ex) {
096       ex.printStackTrace();
097       System.exit(1);
098     }
099     finally {
100       mConnection.disconnect();
101     }
102   }
103
104   public static void main (String args[]) {
105     DemoSalesOrder app = new DemoSalesOrder();
106   }
107 }
```

container. The request container allows easy access to all import parameters.

While update BAPIs since 4.0A are not supposed to contain any COMMIT WORK statements, the *SalesOrder.CreateFromDat1* BAPI allows the client program to decide whether a COMMIT WORK should take place automatically or not. Since I want to control the commit myself, I set the *WithoutCommit* parameter to true (line 28). Since ABAP does not have a Boolean data type, this is accomplished by putting "X" into the one-byte character field (an empty string would indicate false).

The *OrderHeaderIn* parameter is a mandatory structure parameter. Lines 30-31 access this parameter in the request container, and in lines 32-34 the fields "SALES_ORG", "DISTR_CHAN", and "DIVISION" are filled. We also need to specify the order type (field "DOC_TYPE"). If we look up the details for this field in the dictionary, we see that a conversion routine is defined for this field. Our user (if the sample program had a GUI) would expect to be able to use the order types as defined for the language in which he is logged on to SAP. But the BAPI requires the German code. In line 36 we simulate the order type "OR" (in English) having been entered by the user. Lines 37-40 convert this value into the internal (German) equivalent ("TA").

In line 42 an arbitrary purchase order number is entered.

These are the only required fields in the *OrderHeaderIn* parameter.

The customer for the sales order is specified next. This happens via the *OrderPartners* table parameter, which is accessed in the request container in lines 44-45. Line 46 adds a new row to the table. Line 47 hard-codes a partner role (field "PARTN_ROLE"). This field also uses a conversion routine, but we can omit the conversion here since we assume that this code would never show up in a user interface. "AG" denotes the sold-to party.

The other field in this table parameter ("PARTN_NUMB") contains the customer number, another field with a conversion routine. Line 48 simulates our presumed user entering "30001". This value is then converted appropriately in lines 49-52.

One more parameter to go, the *OrderItemsIn* table parameter. It is accessed in the request container in lines 54-55. Line 56 adds a row to the table. Two fields ("MATERIAL", the material number, and "REQ_QTY", the desired quantity) must be filled. In lines 57-60 the material number is set and converted (another case with a conversion routine). Line 61 fills in the quantity with a value of one (expressed as "1000"[22]).

Lines 63-68 add a second item to our sales order.

We are now ready to call the BAPI, which we do in lines 70-71. The BAPI method offered by the proxy class returns a response container that allows easy access to all the export parameters. Every self-respecting client program will always check the standard *Return* parameter. To make this as simple as possible, we create a *BapiMessageInfo*[23] object (lines 73-75).

We check the return code of the BAPI in line 76, and if it is bad, do the following:

- Issue a rollback call (line 77). This is not absolutely required, but I like to be on the safe side.
- Print out the BAPI error message (lines 78-79).
- Retrieve the detailed documentation for this error message (lines 82-83).
- Print this documentation (lines 84-86).

In case the BAPI call succeeded, the following will take place:

- Commit the sales order (line 88).
- Print a message indicating the success of our endeavor, including the number of the new sales order just created in SAP (lines 89-92).

Eventually (line 100) we disconnect from SAP.

---

[22] We discussed this peculiar behavior earlier.

[23] This class was discussed in my article "BAPI Return Messages Made Easy" in the May/June 2002 issue of this publication.

Conclusion

While there is still more to learn about BAPIs[24], you should by now be familiar with what BAPIs are and how to investigate them. You know the most important issues that can occur in BAPI programming and how to deal with them. Your fingers should now be itching to get to the keyboard and write a BAPI-enabled application yourself. Given the ever-increasing number of BAPIs and the growing demand for integration of SAP systems both with your own non-SAP and your business partner applications, you should have no difficulty finding a suitable project.

If you program in Java and you want the source code from Listing 1, please send me an email.

[24] The articles listed in footnote 7 contain a lot, but not all of it.

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.

