Everything a BAPI Programmer Needs to Know About the Business Object Repository

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 CA925 and CA926 classes. Prior to founding ARAsoft in 1993, he worked with SAP AG and SAP America for seven years.

(complete bio appears on page 32)

Trying to develop or use SAP's Business Application Programming Interfaces (BAPIs) without a solid understanding of what they are, how they are defined in the Business Object Repository (BOR), and what the difference between a BAPI and the ABAP Function Module underneath is, leads to errors and frustration. Beginners are often confounded as to how to approach the BAPIs. Even experienced developers do not always appreciate all relevant details of the BOR metadata and spend more time than necessary trying to solve a problem. This article aims to provide a solid foundation for any BAPI-related activity. Whether you are a manager in charge of a BAPI development project or a developer commissioned to write new BAPIs or use existing ones, reading this article should make your life easier.

Introducing the BAPIs

In order to make interoperability between SAP and non-SAP components easier in general, and to specifically address the new challenges posed by e-commerce, SAP — in release 3.1H — decided to introduce a new set of interfaces to the SAP components (of which R/3 is one[1]), the BAPIs. As a first approximation, BAPIs can be described as official, documented, upward-compatible, GUI-free interfaces. In order to provide the maximum degree of openness the BAPIs needed to be:

- Defined independently from any middleware technology (DCOM, CORBA, etc.)

[1] R/3 is not the only SAP product that has BAPIs. They can also be found in APO, BW, etc.

- Usable from non-object-oriented languages like C and classical ABAP
- Usable from object-oriented (ABAP Objects, C++, Java, etc.) and somewhat object-oriented (Visual Basic before 7.0) languages

To support non-object-oriented languages, SAP developed the BAPIs as regular Remote Function Call (RFC)-enabled ABAP Function Modules.[2]

To provide object-oriented access to the BAPIs, they had to be defined as methods of object types[3]. They needed an object-oriented home, as it were. Fortunately, this home did already exist. In 3.0A, SAP had introduced its Business Object Repository. The main motivation behind this development was Business Workflow. When you model a process in your organization you want to be able to think in terms of:

> When the purchase requisition for a PC has been approved by the appropriate line manager, send it to the IT department to validate the configuration.

instead of:

> When field "XXXX" in table "YYYY" contains value "1234", call function module "ZCHKCFG" and pass value "=CHKPC" into parameter "FUNC".

In other words, you need an object model with object types, attributes, events, and methods. Underneath this clean, understandable layer, we still have tables and reports and transactions and dialog modules and function modules, but fortunately we can ignore them when we model our workflows.

So when the BAPIs needed an object-oriented repository, it was an obvious decision to reuse the BOR (remember that one of the reasons for object-orientation is better support for reuse).

[2] Henceforth called RFMs.

[3] If your background is object-oriented programming languages: This is totally synonymous with the term "classes" for our purposes.

As a consequence, when we look at the BOR today, we not only find the BAPIs, but lots of workflow-related stuff, as well. This can be confusing if you are looking at the BOR for the first time and try to distinguish between BAPI-related and other information. Let me be specific:

- The BOR contains object types that (currently) have no BAPIs whatsoever.
- The BOR contains methods that are not BAPIs.
- The BOR contains metadata (like events) that is not relevant for the BAPIs.

Accessing the BOR

SAP offers different ways of accessing the information in the BOR. Depending on your requirements, you will need to use one or more of them. The information in the BOR can be accessed:

- Through the *SAP Interface Repository* (**http://ifr.sap.com**).
- Through the *Business Object Repository API* (BOR API). This consists of a set of function modules that can be used to retrieve the metadata via RFC. The documentation for this API is available from **www.sap.com/csp/scenarios/validation/docs/bor_api.pdf**.
- Through the *Business Object Builder, Business Object Browser,* and *BAPI Explorer* SAPGUI transactions (transaction codes SWO1, SWO2, and BAPI, respectively).

Mostly, you will use the SAPGUI transactions, but I will deal with the two other possibilities first, because the majority of this article is devoted to understanding the BOR metadata and learning how to use the SAPGUI transactions.

The Interface Repository

This is an exciting new effort by SAP to provide all relevant interfaces (BAPIs and IDoc types) on the

Internet. SAP has written extraction programs that generate XML documents, which are then used to generate HTML output that you can view in your browser. At the time of writing, all BAPIs and IDoc types for releases 3.1I, 4.0B, 4.5B, 4.6B, and 4.6C can be studied in the Interface Repository. Currently, not all details required by developers are available so we will still have to use the SAPGUI transactions if we need the complete picture. But this effort is definitely headed in the right direction and SAP has plans to extend the Interface Repository in its second release. You should check the URL given above for news in this area once in a while.

Using the BOR API

Why would you want to access the BOR through an API instead of simply looking at it using SAPGUI transactions? Mostly, that is required for software companies that want to build tools around the BAPIs, for example:

- A BAPI browser integrated into their development environment
- A code generator to facilitate BAPI access in a specific programming language
- A metadata extractor in order to store the metadata in a non-SAP repository, for example, a case tool
- An interface tool (probably consisting of a mapping tool and a runtime environment) that allows users to create interfaces between SAP and non-SAP systems without writing any code[4]

To some extent, though, accessing the metadata is also useful in slightly advanced BAPI-enabled applications. If a BAPI parameter has to be entered by a user within the GUI of your application, and this parameter has a default value, it would be nice if you showed the default in the data entry field to let the user know which value will be used if he enters no data into this field.

From extensive personal experience, I can tell you that using the BOR API is not a trivial exercise. Although there is documentation, you still need to spend some time experimenting with the various function modules until you really understand what is going on. You should know how to use the *Function Builder* test environment (discussed in my article "Need to Understand a BAPI's Parameters? Test-Drive the BAPI Inside SAP!" in the November/ December 2000 issue of this publication) in order to find out what is really going on. Fortunately, some third parties have built components that encapsulate the BOR API for various environments.[5]

The SAPGUI Tools

The various transaction codes used in SAP to explore the BOR will be discussed in the last part of this article.

First, I want to introduce an object model that should facilitate your task of understanding what metadata exists and how it is related.

A BAPI-Centric Object Model for the BOR

My belief is that once you understand the various types of metadata in the BOR and the relationships between them, it will be simpler to use the SAPGUI transaction codes in order to retrieve the desired information relevant for BAPI programming. When teaching SAP's BAPI training courses[6], I noticed that although I gave an overview of the concepts employed in the BOR first, some participants struggled in the BOR-related exercise.

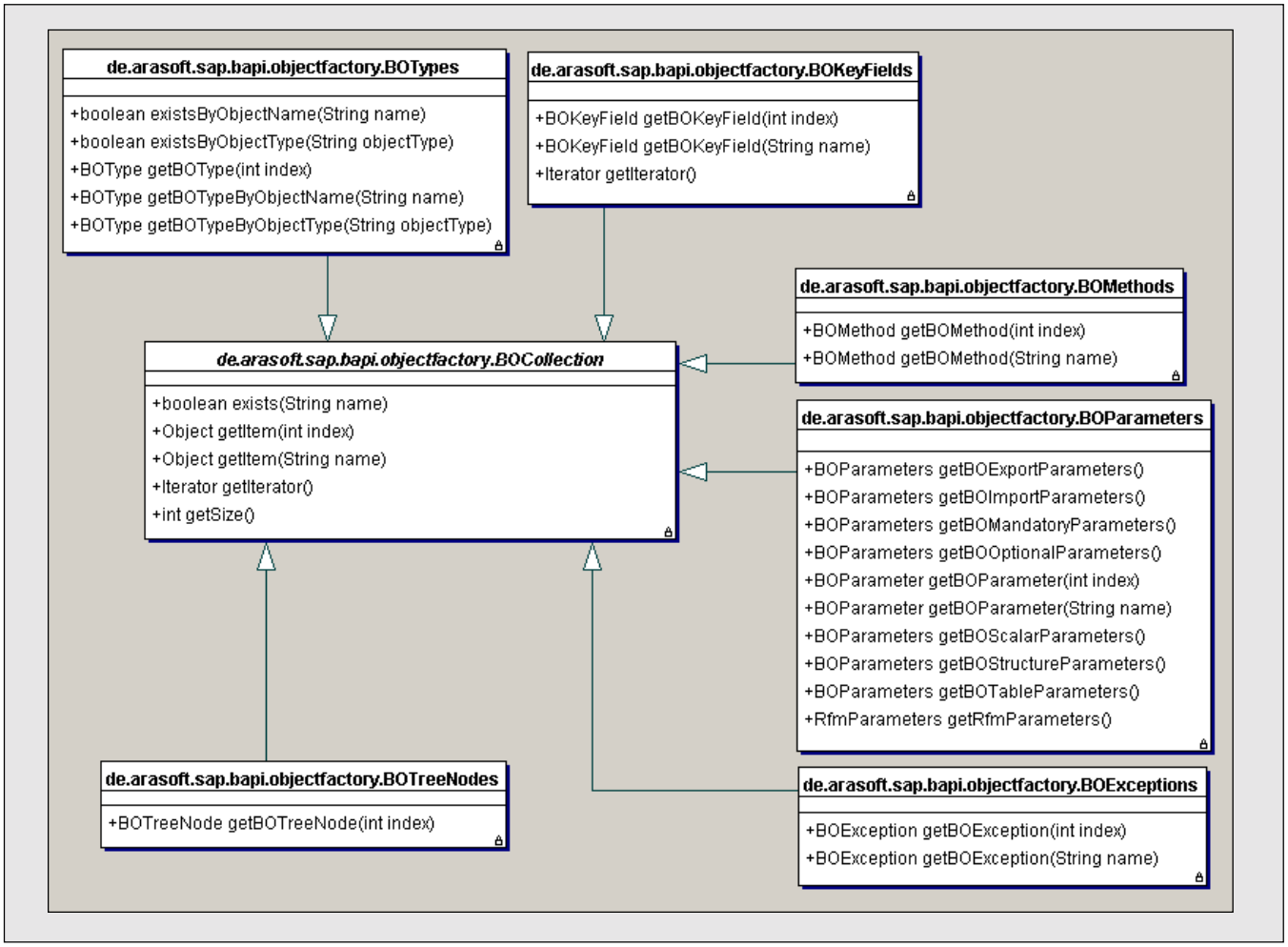
---

[4] See, for example, **www.scribesoft.com/Uploads/Intg_4_SAP.pdf**.

[5] Contact the author to find out more.

[6] CA925 and CA926, for Visual Basic and Java programmers, respectively.

Figure 1 *The BOR Collections*



Being able to review a graphical representation of the various entities, their properties, and the relationship between the entities, should facilitate the understanding of the wealth of metadata contained in the BOR. Therefore, I have generated UML diagrams depicting the relevant metadata. In order to avoid the effort required to enter all the information manually, I have used the reengineering capabilities of the *Together Control Center* product by TogetherSoft (**www.togethersoft.com**) to create the diagrams based on some Java classes that I have written.[7]

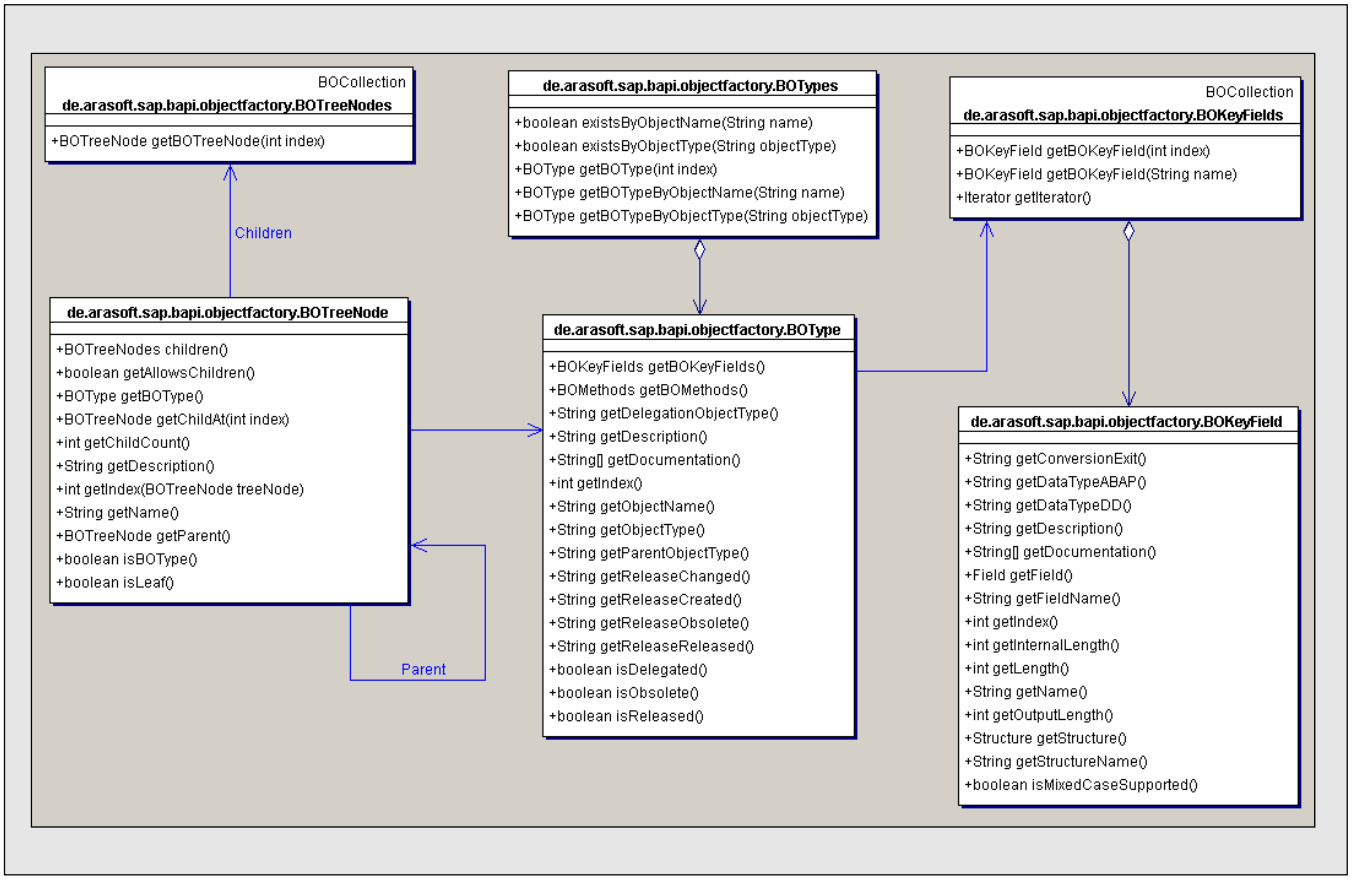
[7] That is why you see the package name "de.arasoft.sap.bapi.objectfactory" before each class name.

This model only deals with object types that have BAPIs and ignores everything in the BOR that is only relevant for Workflow.

Have Collections Everywhere

We already know that the BOR contains multiple object types, each potentially with several BAPIs. A BAPI has one or more parameters. Expressed differently, we are dealing with collections a lot. The BOR has a collection of object types, an object type a collection of BAPIs, a BAPI a collection of parameters. Hence it makes sense to have a common class that deals with the base functionality of a

Figure 2 The BOR Tree and the BOR Object Types



collection for BOR metadata. In **Figure 1**, you see the BOCollection class, which is used as the superclass of BOTreeNodes, BOTypes, BOKeyFields, BOMethods, BOParameters, and BOExceptions. BOCollection offers the basic behavior required by any collection in the BOR, the ability to:

- Check for the existence of an item (exists())
- Find out how many items there are (getSize())
- Access items by their name or index (getItem(...))
- Get an iterator (sometimes also called an enumerator) that allows access to each item in succession without specifying an explicit index

Note that BOCollection does not define any method to remove an item because I wanted to represent the static view of the BOR relevant for writing BAPI client applications. Of course, the SAP system offers the ability to add object types, BAPIs, etc.

The BOR Tree

The object types in the BOR are organized according to the SAP application hierarchy, i.e., they are part of a tree. Classes BOTreeNodes and BOTreeNode in the object model (see **Figure 2**) describe the relevant relationships. Each branch (or folder) in the tree is represented by BOTreeNodes, a collection of zero or more BOTreeNode objects. A tree node is either an object type (a leaf) or another branch with zero or more child nodes. As for any tree, this one also has a root (the top level of the hierarchy). Any node in the tree has a parent, with the exception of the root.

The SAP application hierarchy is not cast in stone. Each new release usually brings about some (minor) changes. Other than making it a little harder to find an object type that was moved, this has no adverse consequences since the position of an object type in the application hierarchy has no implications for the user (or even the developer) of BAPIs. Since not everybody is familiar with the way in which SAP structures its applications, it is also possible to access all object types directly. This is represented by the BOTypes class.

The BOR Object Type

Normally, any item in a BOR collection has one name identifying it and a description. This is not all for a BOR object type, though. It has an internal name (getObjectType(), e.g., "BUS0002") and an external, official name (getObjectName(), e.g.,"CompanyCode"). Both these names must be unique in order for the BAPIs to work.

All BOR metadata entities allow the developer to store documentation. This is true not only for object types (getDocumentation()), but also for key fields, BAPIs, parameters, etc.

An object type can be marked as a Business Object Type. This is not reflected in our object model since it has no consequences whatsoever for BAPI programming. SAP's definition of what makes something a Business Object Type is quite different from what one would intuitively assume. For most people a Business Object Type is one that is related to a business application. In SAP, an object type is a Business Object Type only if a data model exists for it. While having a data model to look at may be an added value in your study of an object type, it is totally unrelated to the question of whether the object type has any BAPIs that we want to call in an application. For our purposes, we can totally ignore this property.

Inheritance

The BOR allows you to subclass an existing object type. This is necessary if you want to modify or enhance an object type. Making changes directly to an object type that was delivered by SAP is not a good idea because of the extra effort that would be required during an upgrade. The new object type inherits all metadata of the superclass, including its key fields and methods. You can now add new BAPIs or extend existing ones.

Each object type can have a superclass (getParentObjectType()), but does not need to have one. You can also create new object types that do not inherit from anything.

Delegation

In object-oriented languages, the developer decides which object type to instantiate in his program. This leads to a problem with customer extensions, though. Since we should not modify an SAP object type, the question arises of how to ensure that everybody uses our new enhanced/modified object type. This is important because otherwise we would have to rely on the good will of the programmer who uses an object type to actually use the new subclassed one. Imagine that you have modified a BAPI of your subclassed object type in a way that is important for your company. The developer of the client application could now use the new object type with the modified BAPI, or the old one with the old behavior. While choice is often a good thing, here it is definitely bad. We need a mechanism that guarantees that our new object type is used, no matter what.

SAP has introduced delegation to solve this issue. In a superclass you can specify which of its subclasses it should be delegated to. Any call to the superclass's methods will now result in a call of the appropriate method of the subclass.

Release Information

An object type was created first in a certain release of SAP (getReleaseCreated()). The release in which the last change was made (getReleaseChanged()) and when

Figure 3 *Obsolete Object Types in 4.6B*

Object type	Object name	Description	Obsolete since
BUS1008	Creditor	Vendor	4.6A
BUS1007	Debtor	Customer	4.6A
BUS1057	InvestmentProgram	Investment program	4.6A

Be careful if you invoke BAPIs through a middleware that uses generated proxy classes. Both SAP DCOM Connector and the SAP Java Connector (and probably most third party generators, as well) generate the name of the underlying RFM that implements the BAPI into their proxies. If you have modified a BAPI in your subclass, the underlying function module has a different name than the original SAP RFM. At generation time, the middleware generator tool should check whether the object type is delegated and use the names of the RFMs defined for the subclass for any BAPI that was modified. SAP's tools actually do this, but when you use any third party middleware you should double-check with your vendor.

What happens, though, when the proxies were generated before we defined the delegation? If a third party sold us an application using proxy-based middleware with pre-generated proxies? In spite of our delegation, the standard SAP BAPIs will be invoked instead of our modified versions.

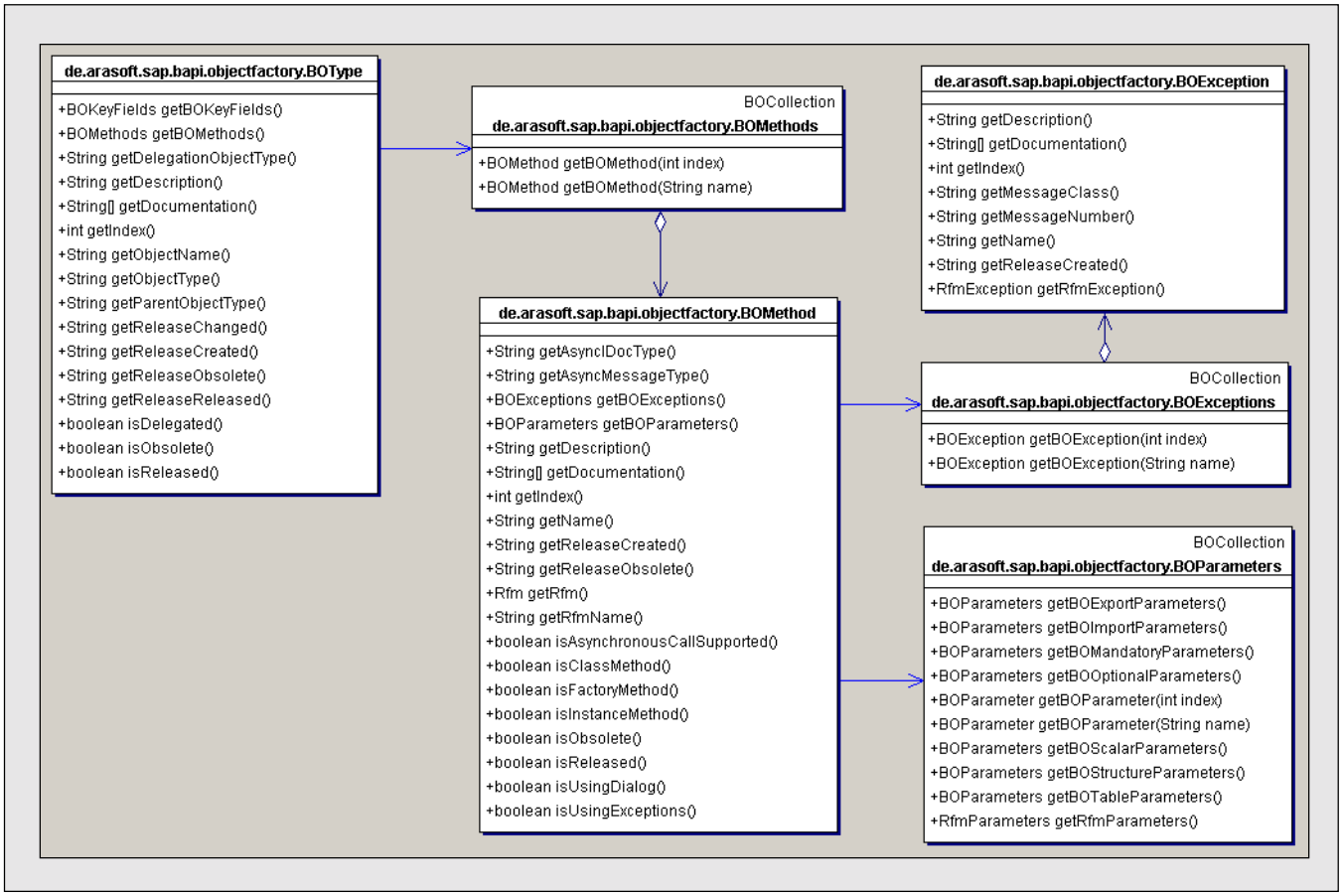
If you are using delegation in your SAP system and proxy-based middleware in your own software or a third party application, you need to make sure that the proxy generation takes place after the definition of the delegation! In the case of a third party product, that will require you to have access to the generator in your company, which is not a problem if standard SAP middleware is used.

it was officially released for customer use (getReleaseReleased()) are also available. Only rarely does a whole object type become obsolete. This is usually due to a major change in the application object model. Obsoleteness is not a Boolean attribute, instead the BOR specifies the release in which the object type became obsolete (getReleaseObsolete()). If you start to build a new application it is a good idea to stay away from obsolete object types. **Figure 3** contains a list of all object types in 4.6B that have BAPIs and are obsolete. (See below for a discussion of the obsoleteness of individual BAPIs.)

Key Fields

The SAP object types are an object-oriented representation of an application using a relational database. The persistent data of an object type is stored in one or more tables in the database. Each table has one or more (primary) key fields uniquely identifying each record. There is always one base table that contains the basic data for an entity. Data for customers (object type Customer), for example, is stored in multiple tables, the base table is KNA1. It uses one key field (the customer number) to identify a particular customer. In order for the BAPIs (and Workflow methods) to identify an instance of an object type (retrieve a particular record in its base table) SAP defines key fields for each object type (getKeyFields()). Most SAP object types have one key field, some need multiple key fields (because the base table has multiple fields in the primary key), some object types of a more technical nature (helper object types like Helpvalues) have none.

Figure 4 The BAPIs of an Object Type



Each key field has a name (getName()) and a description (getDescription()). Each key field is based on the associated field of the basic database table for the object type. The name of the table (getStructureName()) and the name of the field (getFieldName()) can be used to look up additional properties in the SAP Data Dictionary. All these properties will be discussed later when we deal with BAPI parameters (which are also based on dictionary references).

Finally: The BAPIs

The relevant metadata for the BAPIs is depicted in **Figure 4**. Each object type we are interested in has one or more BAPIs (getBOMethods()). A BAPI has a name (getName()), a description (getDescription()), and an underlying RFM (getRfmName()).

Instance and Class Methods

Some BAPIs do not require us to set the key fields of their object type in order to work. SalesOrder.GetList, for example, can be invoked without us specifying the key of an existing sales order. SAP calls methods like this *instance-independent*, but we will use the more common term *class method* (isClassMethod()). Other BAPIs need the key fields in order to select a particular record from the database, SalesOrder.GetStatus, for example. It will only work if we set the one key field that the SalesOrder object type has to the key of an existing

sales order record in the database. (We could use SalesOrder.GetList to retrieve the keys of all sales orders for a customer and then use each key in turn to find out the status of this order.) BAPIs of this type are called *instance-dependent* in SAP, we will use the term *instance method* (isInstanceMethod()). Each BAPI is either a class or an instance method.

Factory Methods

Some BAPIs create new instances (i.e., records in the database). SalesOrder.CreateFromDat1, for example, allows us to create a new sales order. Such BAPIs are called *instance-creating* in SAP, we will use the term *factory method* (isFactoryMethod()). Here the key fields are set by the BAPI, the document number of the newly created sales order for SalesOrder.CreateFromDat1, for example. All factory BAPIs up to 4.6B are class methods, but it would be possible to have an instance method that is a factory method as well. This would require us to set the key fields of an existing instance before the BAPI call, and then, after the call, the key fields would contain the new values set by the BAPI.

Obsoleteness

If SAP needs to enhance an existing BAPI, they try to do it in an upward-compatible fashion by adding optional parameters. Sometimes a more drastic change is required though in order to provide additional functionality. In that case, a new BAPI is created. The name of the new BAPI is usually the same as the name of the old one plus a number. SalesOrder.CreateFromDat1 is the successor of SalesOrder.CreateFromData (there was no more room for the "a" in the new name, hence the "Dat1" instead of "Data1"). The old BAPI is marked obsolete (getReleaseObsolete()) for the release in which the new BAPI is introduced. An obsolete BAPI is still guaranteed to work in the release in which it became obsolete and at least one subsequent functional release. (SAP distinguishes between functional and maintenance releases, 4.0A was functional, 4.0B maintenance, and so on.) Since there is no more than one functional release per year, this gives you at least two years (more if you do not switch to the next functional release the day it becomes available, something that hardly anyone does) in which your BAPI-enabled application will run without any change.

The key fields that are used by instance and factory methods are not defined as parameters of the BAPI, but they are defined as parameters of the underlying RFM.

If you invoke an instance or factory BAPI in an object-oriented way (with, e.g., SAP DCOM Connector, SAP ActiveX Controls, SAP Java Connector), the key fields are set and retrieved separately (the details vary slightly depending on the chosen middleware). The middleware takes care of mapping the key fields to the appropriate parameters (import for instance methods, export for factory methods) of the underlying RFM. This mapping requires each key field-related parameter of the RFM to have exactly the same name as the key field in the object type definition.

If you invoke a BAPI in a non-object-oriented fashion, i.e., you invoke the RFM directly (in classical ABAP or C, for example), you treat the key fields as normal parameters of the RFM because you do not deal with an object type that has key fields defined.

Developers who are not yet very familiar with object-orientation usually find the latter approach much simpler, but as always, the initial extra effort required to learn object-oriented programming is rewarded manifold by increased productivity and more enjoyment of the software development effort.

Figure 5 contains a list of all obsolete BAPIs of non-obsolete, released object types, up to 4.6B. (In addition, all BAPIs of the object types listed in Figure 3 are obsolete because the whole object type is obsolete.)

In case you are confused by some BAPIs being obsolete since 4.0C: Release 4.0C never existed, this is just another way of saying 4.5A (the next functional release after 4.0A).

Figure 5 Obsolete BAPIs in 4.6B

Object name	BAPI name	Description	Obsolete since
BapiService	DataConversionExt2Int	Convert data from external format into internal format	4.6A
BapiService	DataConversionInt2Ext	Convert data from internal format into external format	4.6A
ControllingDocument	GetDetail	Use find details: Do not use this method any longer	4.6A
CostCenter	GetDetail	Detailed Information About Cost Center For Key Date	4.6A
CostCenter	GetList	List Of All Cost Centers According To Selection Criteria	4.6A
Customer	ChangeFromData	Change	4.0C
Customer	CheckExistence	Check existence	4.6A
Customer	CheckPassword	Check password	4.6A
Customer	CreateFromData	Create	4.0C
Customer	GetDetail	Read	4.0C
Customer	Search	Find	4.0C
EmployeeAbsence	Approve	Unlock absence	4.6A
EmployeeAbsence	Change	Change absence	4.6A
EmployeeAbsence	Create	Create absence	4.6A
EmployeeAbsence	Delete	Delete absence	4.6A
EmployeeAbsence	Request	Create locked absence	4.6A
EmployeeBenefit	GetEventList	Determine the currently valid events of an employee	4.0C

(continued on next page)

Figure 5 (continued)

Object name	BAPI name	Description	Obsolete since
EmployeeBenefit	GetOffer	Define benefits offer	4.0C
EmployeeBenefit	GetOpenEnrollmentPeriod	Check if open enrollment period exists and period is indicated	4.0C
FixedAsset	CreateFromData	Create asset	4.6A
ItCustBillingDoc	CancelFromData	Cancel/reverse	4.6A
ItCustBillingDoc	CreateFromData	Create using external document	4.6A
ItCustBillingDoc	IsCancelled	Has the billing document been cancelled?	4.6A
ItCustBillingDoc	Simulate	Simulate using external document	4.6A
Kanban	GetListForSupplier	Provide Kanban data for vendors	4.0A
Material	GetInternalNumber	Assign new internal numbers	4.6A
MaterialPhysInv	Create	Create Physical Inventory Document	4.6A
Promotion	GetSitePlanning	Read plant planning data for a promotion	4.6A
PurchaseOrder	GetList	List purchase orders - only up to 4.0A	4.0A
PurchaseReqItem	CreateFromData	Create requirement coverage request	4.0A
PurchaseReqItem	GetList	Read requirement coverage request	4.0A
PurchaseReqItem	SingleReleaseNoDialg	Release purchase requisition	4.6A
SiteLayoutModule	GetItem	Material data for a layout module	4.6A
USER	Create	Create a user	4.6A

BAPIs with Dialog, Really?

The main raison d'être for the whole BAPI endeavor is to enable you (and also SAP) to invoke functionality from outside an SAP system, using your own or no GUI. This obviously requires the BAPIs to not use SAPGUI dialogs. But in some integration scenarios between multiple SAP systems it makes sense for one SAP system to invoke a BAPI in another system and for that BAPI to have a dialog with the SAPGUI user. Therefore, SAP introduced BAPIs with dialog (isUsingDialog()) in 4.6A. They will mainly be used by SAP applications, but if you want to use them in your own (desktop) applications, that is possible as well, as long as the user of your application has SAPGUI installed and does not get too confused by it popping up in the middle of your application.

Figure 6 BAPIs with Dialog in 4.6B

Object name	BAPI name	Description
ACCDocument	Display	Display
AcctngServices	DocumentDisplay	Accounting: Display method for follow-on document display
BusPartnerContact	CreateFromDataWithDialog	Create Data Container with Dialog
BusPartnerEmployee	Create	Create
BusPartnerEmployee	Display	Display
BusPartnerEmployee	Edit	Edit
Customer	Create	Create online
Customer	Delete	Delete customer master data online
Customer	Display	Display online
Customer	Edit	Change online
EmployeeTrip	ChangeInteractive	Change a trip (interactively)
EmployeeTrip	CreateInteractive	Create trip online (receipt entry)
ItCustBillingDoc	Confirm	Confirm object
ItCustBillingDoc	Create	Create object
ItCustBillingDoc	Display	Display object
ItCustBillingDoc	Edit	Change object
Material	Delete	Delete

(continued on next page)

Figure 6 contains a list of all BAPIs with dialog for non-obsolete, released object types, up to 4.6B.

Asynchronicity

Normally, we want to call BAPIs synchronously in order to get feedback (return code, result data) immediately. Of course, a synchronous call will only work if the system we want to access is up and running. When SAP started to use BAPIs for the communication between different SAP systems, they also wanted a mechanism to invoke a BAPI asynchronously, so that the calling application could proceed even if the remote system was currently unavailable. Luckily, they already had an

Figure 6 (continued)

Object name	BAPI name	Description
Material	Display	Display object
Material	Edit	Change object
PayrollAccDocument	Display_Acc	Display_Acc
PTManagerExtTimeSpec	Display	Display external data in infotype
RetailMaterial	Create	Create material
RetailMaterial	Delete	Discontinue material
RetailMaterial	Display	Display material
RetailMaterial	Edit	Change material
StandardMaterial	Create	Create
StandardMaterial	Delete	Delete
StandardMaterial	Display	Display object
StandardMaterial	Edit	Change object
TravelAccDocument	Display_Acc	Display_Acc
USER	Display	Display object
Vendor	Create	Create online
Vendor	Delete	Set deletion indicator online for vendor
Vendor	Display	Display online
Vendor	Edit	Change online

asynchronous message passing mechanism with guaranteed delivery: ALE (Application Link Enabling). ALE uses IDocs (Intermediate Documents) as the containers for messages. IDocs are instances of IDoc types, which in turn are associated with ALE Message Types. How can you turn a BAPI into an IDoc? SAP developed a generator that takes a BAPI and creates:

- An appropriate ALE Message Type with an associated IDoc Type. The IDoc Type has fields for all the data in the BAPI's import parameters.
- One function module for outbound processing with the same parameters as the original BAPI. It is invoked by the client ABAP code instead of the original BAPI that would have been called

synchronously. The generated function module takes the parameters and puts them into an IDoc. This IDoc is then sent to the target system by the ALE communication layer. This may happen immediately or much later, depending on the availability of the target system and the ALE configuration.

- One function module for inbound processing that is used in the target system to call the BAPI. The ALE inbound processing layer calls this function module, which takes the data from the IDoc and invokes the original BAPI with all passed parameters.

The BOR can tell you whether asynchronous calls are supported for a BAPI (isAsynchronousCallSupported()) and — if they are — the names of the generated ALE Message (getAsyncMessageType()) and IDoc (getAsyncIDocType()) types.

This whole mechanism can only be used if the calling application does not need any data back from the BAPI. The calling application in the source system may have completed long before the IDoc is processed in the target system. Since no information is returned, the calling application does not even know whether the resulting BAPI call was successful. So error handling must happen in the target system, and a workflow item is created in the target system whenever an error occurs. Also, this mechanism only makes sense for BAPIs that update the database. Calling a BAPI used to retrieve information asynchronously is nonsensical since no data is returned to the caller.

For SAP, calling BAPIs asynchronously via IDocs was a suitable and easy-to-use (for the ABAP developer) way of updating applications running in different systems. Does this apply to non-SAP applications? Not as much:

- If you want to send a message asynchronously, you have to create the IDoc yourself because you do not have a generated function module that takes care of that. Learning how to do this is not trivial.
- Normally you want some feedback in your application (success/failure of the call, key fields of created sales order, etc.).

In most cases, it is probably simpler to try a synchronous call, and if that fails, save the data in your own database, and try another synchronous call later.

Exceptions Are an Exception

ABAP function modules can define exceptions that are "thrown" in case of an error. One of the many rules that BAPIs should follow is that they do not use exceptions, and have a Return parameter instead. Unfortunately, not everybody plays by the rules. An exception has a name (getName()), which is the string defined by the ABAP programmer in the RFM, a description (getDescription()), and an associated text message from table T100 (getMessageClass() and getMessageNumber()).

Figure 7 contains a list of all non-obsolete BAPIs with exceptions for released object types, up to 4.6B.

> ### ✔ *Tip*
>
> *If you are using BAPIs with exceptions in your applications, make sure that you understand how such an exception will be communicated by the middleware you use and write code to handle the exception (again, details vary depending on the particular middleware).*

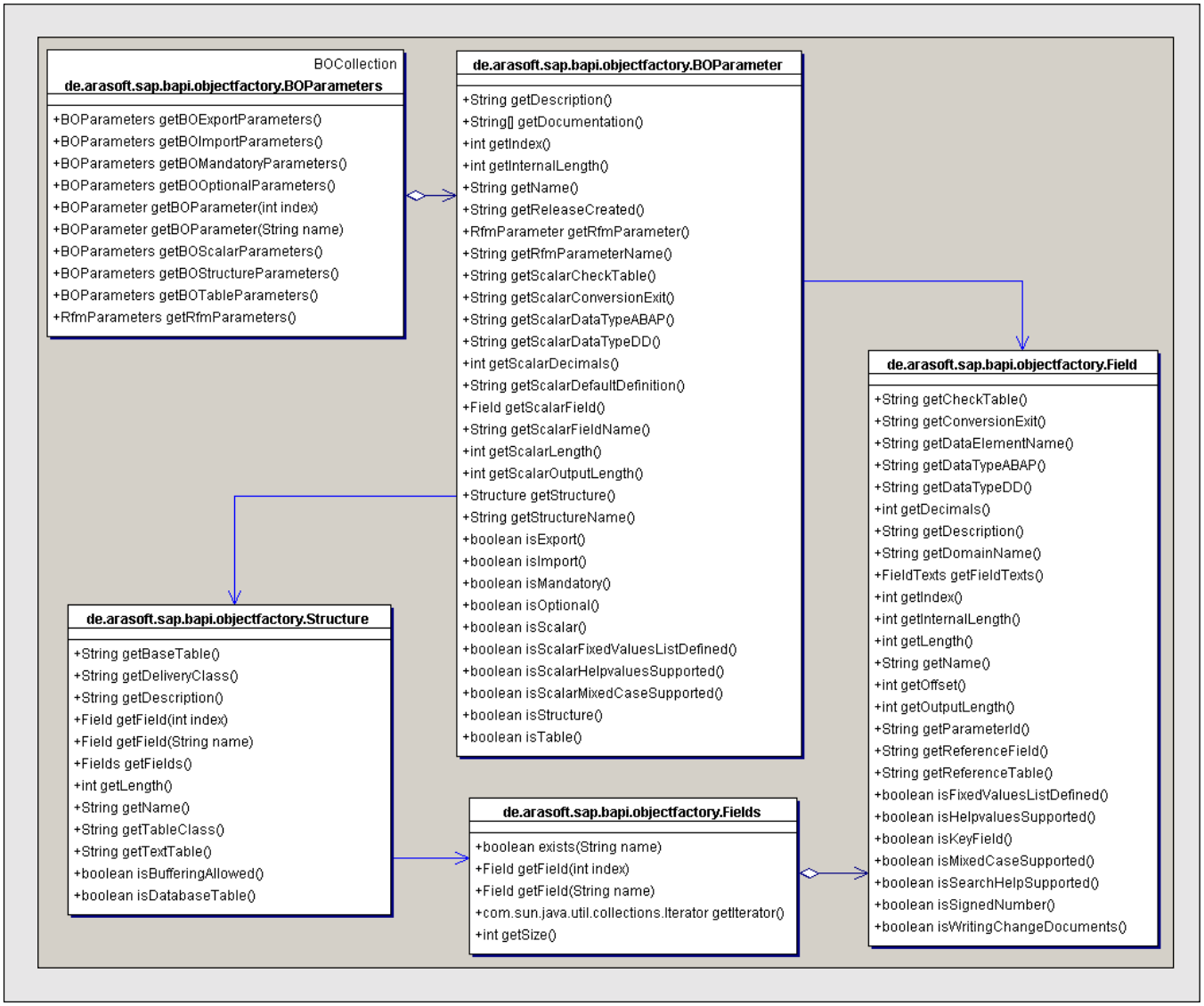
BAPI Parameters

The most important metadata for a BAPI is its

Figure 7 BAPIs with Exceptions in 4.6B

Object name	BAPI name	Description
Attendee	ChangePassword	Change attendee password
Attendee	CheckExistence	Check existence of attendee
Attendee	CheckPassword	Check attendee password
Attendee	GetBookList	Attendee Bookings
Attendee	GetCompanyBookList	Bookings of group attendee
Attendee	GetCompanyPrebookList	Group attendee's prebookings
Attendee	GetPrebookList	Attendee Prebookings
Attendee	GetTypeList	Internet-relevant attendee types
BusinessEvent	GetInfo	Business event information
BusinessEvent	GetLanguage	Business event languages
BusinessEvent	GetSchedule	Time schedule of business event
BusinessEvent	Init	Default values for standard parameters
BusinessEventGroup	GetEventtypeList	Business event types in business event group
BusinessEventGroup	GetList	Read Business Event Group Hierarchy
BusinessEventtype	GetEventList	Dates of business event type
BusinessEventtype	GetInfo	Information on business event type
Location	GetListAll	List business event locations
PurchaseOrder	Release	Release purchase orders
PurchaseOrder	ResetRelease	Cancel release of purchase orders
PurchaseReqItem	Release	Release purchase requisition
PurchaseReqItem	ResetRelease	Cancel release of purchase requisitions

Figure 8 The Parameters of a BAPI



parameters (cf. **Figure 8**). Parameters can be categorized in different ways.

Firstly, there are import (isImport()) and export (isExport()) parameters. Import parameters are passed to, export parameters returned from the BAPI. A parameter can be both import and export.

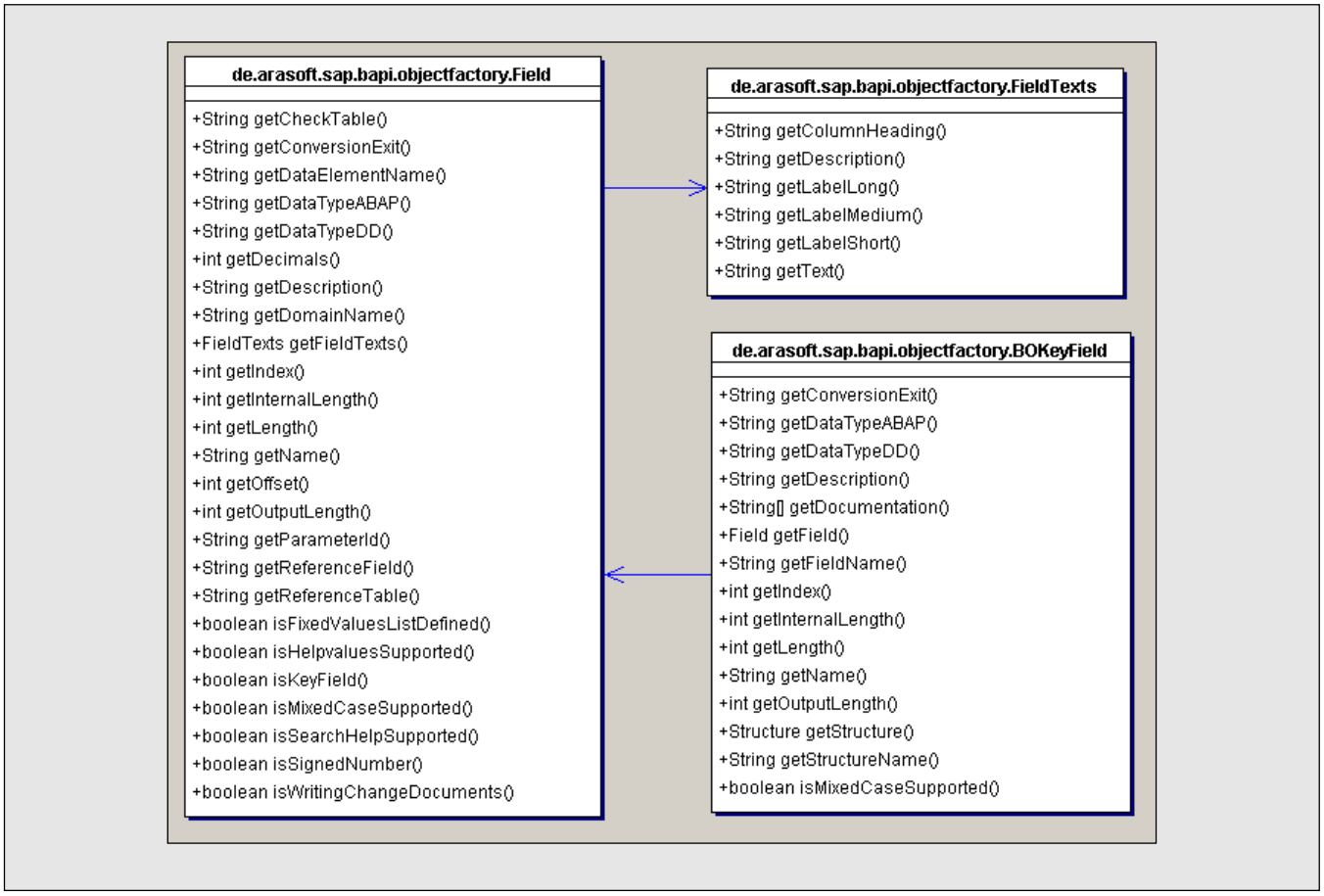
Secondly, parameters can be mandatory (isMandatory()) or optional (isOptional()). All parameters that are only import are optional.

Thirdly, parameters can be scalar (isScalar()) or structure (isStructure()) or table (isTable()) parameters.

A scalar parameter is based on a field in a dictionary structure or table (getStructureName() and getScalarFieldName()). An optional scalar import parameter can have a default value (getScalarDefaultDefinition()).

A structure parameter is based on a complete

Figure 9 *A Dictionary Field*



structure or table in the dictionary (getStructureName()). The structure consists of one or more fields.

A table parameter is based on a complete structure or table in the dictionary (getStructureName()). The table consists of zero or more rows, each with the fields defined in the dictionary structure. On the level of the RFM, table parameters are always import and export (they are passed by reference in RFC). On the BOR level, a table parameter can be defined as import, or export, or both. This allows the developer of the BAPI to tell us whether we might want to put data into a table parameter before the BAPI call or whether the table is (also) used to return data to the client program. The attribute is not always maintained correctly in SAP, though. Some table parameters are defined as import and export, although the BAPIs they are used in never look at any data submitted by us, and just use the parameter to return data to us. The documentation usually gives us an indication that this is the case. CompanyCode.GetList, for example, has a table parameter called CompanyCodeList, marked as import *and* export, that in reality is only export.

Dictionary Fields

Dictionary fields (cf. **Figure 9**) are behind the following entities:

- Key fields

- Scalar parameters
- Fields in a structure parameter
- Fields in a row of a table parameter

The SAP Data Dictionary contains a plethora of information about each field. Each field has a name (getName()). The field name in the dictionary is not related to the name of a key field or scalar parameter based on the dictionary field. The key field for the Customer object type is called "CustomerNo". It is based on the field "KUNNR" in dictionary table "KNA1". Fields in structure or table parameters have only their own name, though.

Each field has an ABAP (getDataTypeABAP()) and a dictionary (getDataTypeDD()) data type. Three different length values are defined:

- The internal length (getInternalLength()) is the number of bytes used in memory.
- The length (getLength()) is the number of positions required on a GUI to display the largest possible content of a field without extra formatting.
- The output length (getOutputLength()) is the number of positions required on a GUI to display the largest possible content of a field with extra formatting.

Let me give you an example: A packed (or as some say, Binary-Coded Decimal) number with an internal length of 6 bytes, would have a length of 11 positions and an output length of 16. Internally, each byte of the packed number contains two digits (with the exception of the last half-byte that represents the sign).

The largest number of digits supported is 6 times 2 minus 1 equals 11. A packed number can have decimals and a minus sign. If we assume that we use all digits and that our number has one decimal (getDecimals()) and is negative, a nicely formatted version might look like "-1,234,567,890.1".

Each field has a maximum of five texts (getFieldTexts()) associated with it:

- A description (getDescription())
- A column heading for reports (getColumnHeading())
- A long screen label (getLabelLong())
- A medium-sized screen label (getLabelMedium())
- A short screen label (getLabelShort())

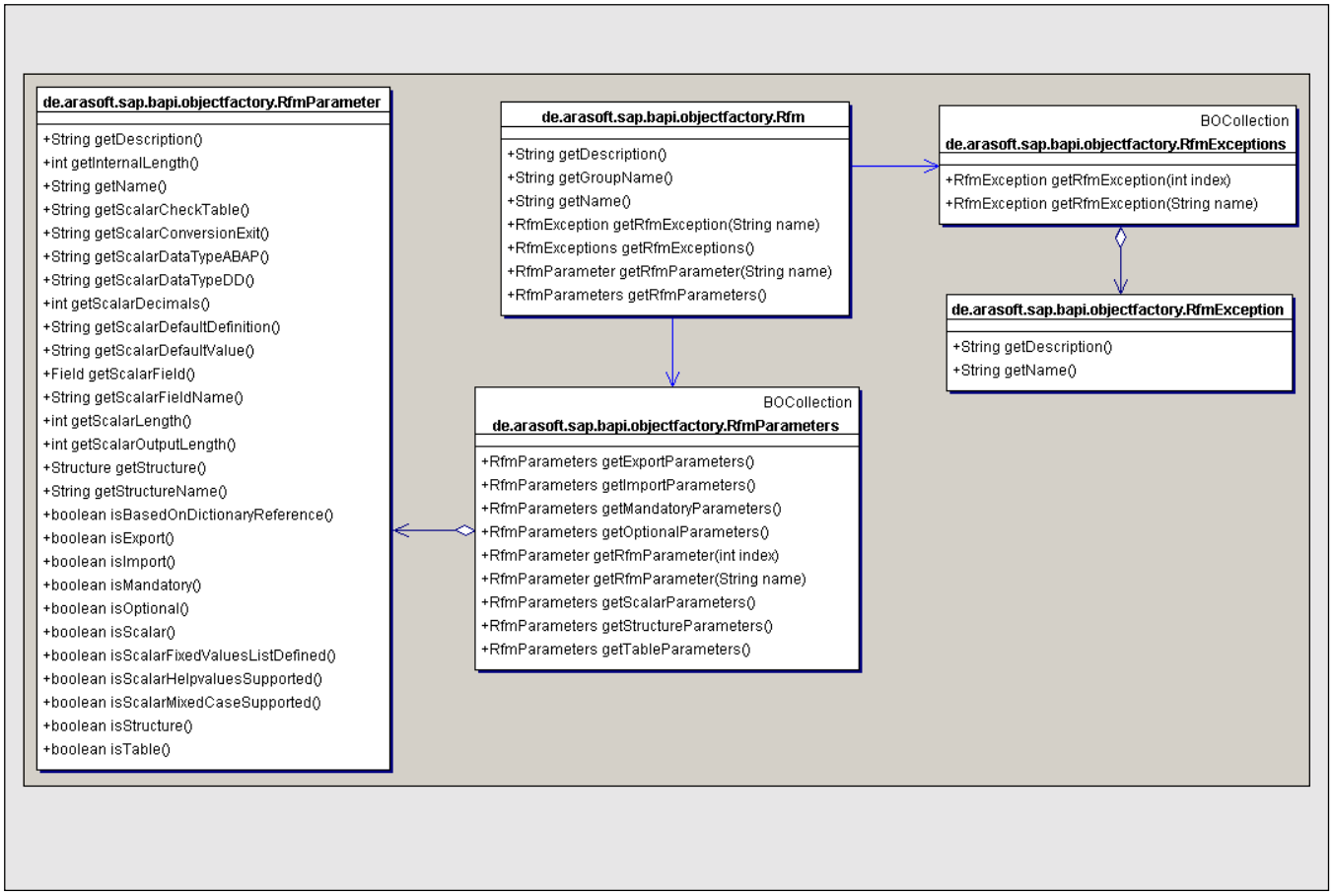
Not all of these texts are necessarily maintained in all languages.

Fields can be based on a check table (getCheckTable()) or a fixed values list (isFixedValuesListDefined()). Under certain conditions (see my article "Enabling Point-and-Click Data Entry Assistance for Your BAPI Applications" in the September/October 2000 issue of this publication) additional data can be retrieved for a field using the Helpvalues.GetList BAPI (isHelpvaluesSupported()). In SAPGUI, all fields based on check tables or fixed values lists (and in some other special cases) offer search help (isSearchHelpSupported()).

Some fields use conversion exists (getConversionExit()). In that case, there is an internal value (used in the database) and an external value (used on a GUI). The conversion BAPIs of the BapiService object type can be used to convert between the two formats.

Some character fields store only uppercase characters, some allow mixed case (isMixedCaseSupported()).

Figure 10 A Function Module with Parameters and Exceptions



The Function Module's View

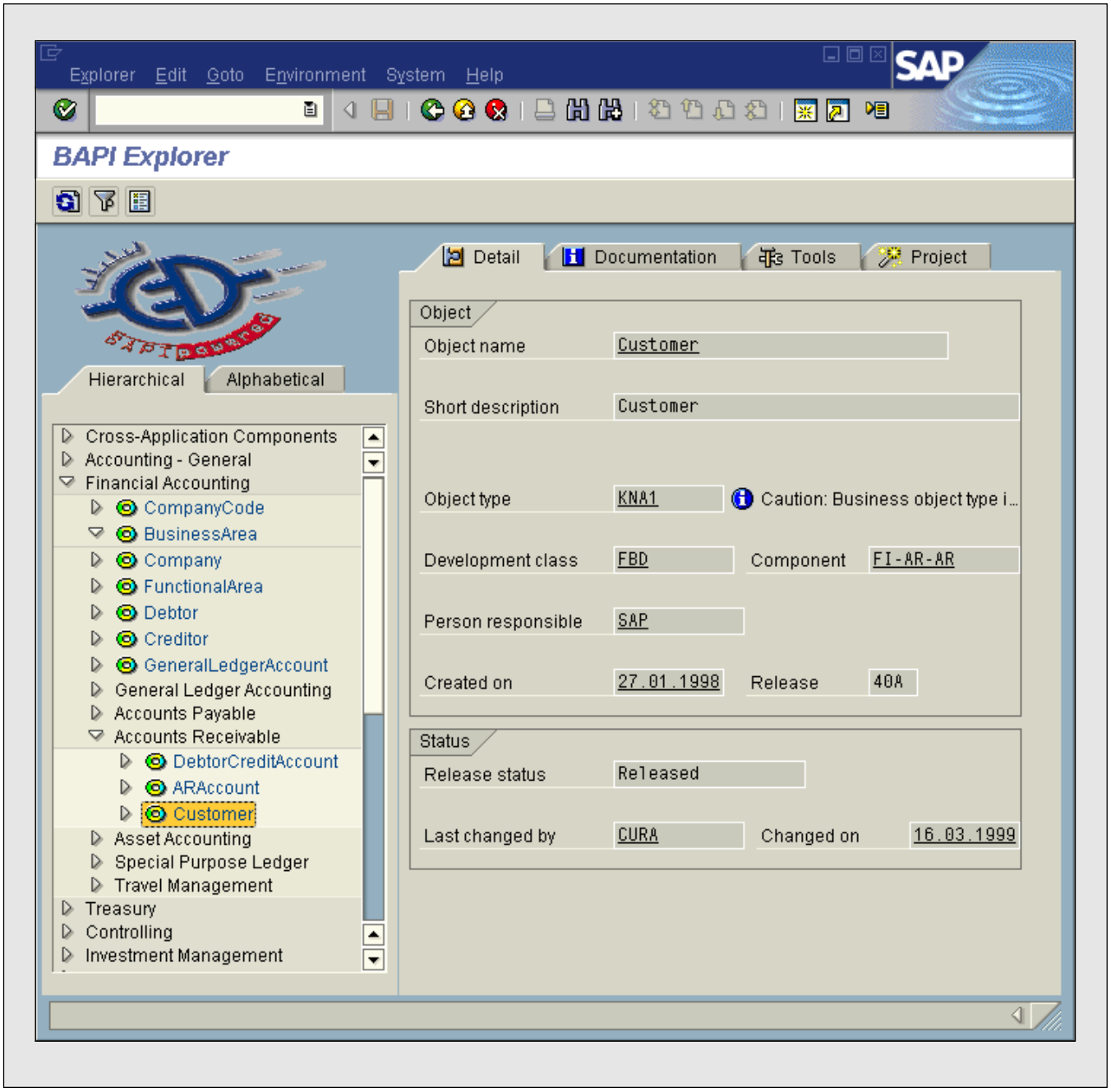
We have discussed all important metadata from the BOR's viewpoint now. **Figure 10** shows an RFM (one that implements a BAPI or one that is not part of an object type) with its parameters and exceptions. The similarity between an RFM's metadata and a BAPI's is big. That should be no surprise because BAPIs are implemented by RFMs. By now you should have no problem interpreting the information in Figure 10.

Exploring BAPIs in SAPGUI

Now that we understand what information is available, it should be no big deal to actually look it up in the SAP system. The following transaction codes are commonly used to research BAPIs:

- The BAPI Explorer (transaction code BAPI)
- The Business Object Builder (transaction code SWO1)
- The Business Object Browser (transaction code SWO2)
- The Function Builder (transaction code SE37)
- The Dictionary (transaction code SE11)

Figure 11 *The BAPI Explorer*



The BAPI Explorer (see **Figure 11**) is the natural starting point. All other transactions can be reached from it, usually with a double-click or the pressing of a button. SAPGUI has amazing navigational capabilities.

Figure 11 shows the hierarchical view of the BOR, with the Customer object type selected. Alternatively, you can switch to an alphabetic view.

By default, the BAPI Explorer shows only

released BAPIs. If you want to see the unreleased ones as well, you can click on the filter icon (the one that looks like a funnel) and select the "All" radio button (see **Figure 12**).

Look at the "i" icon in Figure 11. If you rest your cursor over the text to the right of it, the complete text is shown in a tooltip: "Caution: Business object type is being delegated". This is SAP English for "Caution: Business object type has been delegated". Later you will see how we can find out which object type it has been delegated to.

In order to understand the icons used in the tree portion of the BAPI Explorer, you can display the legend pop-up (see **Figure 13**).

Figure 12 Selecting all BAPIs

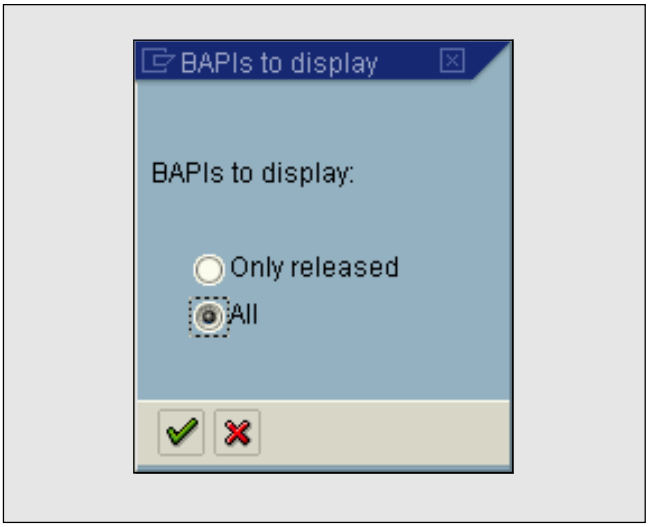


Figure 13 The BAPI Explorer Legend

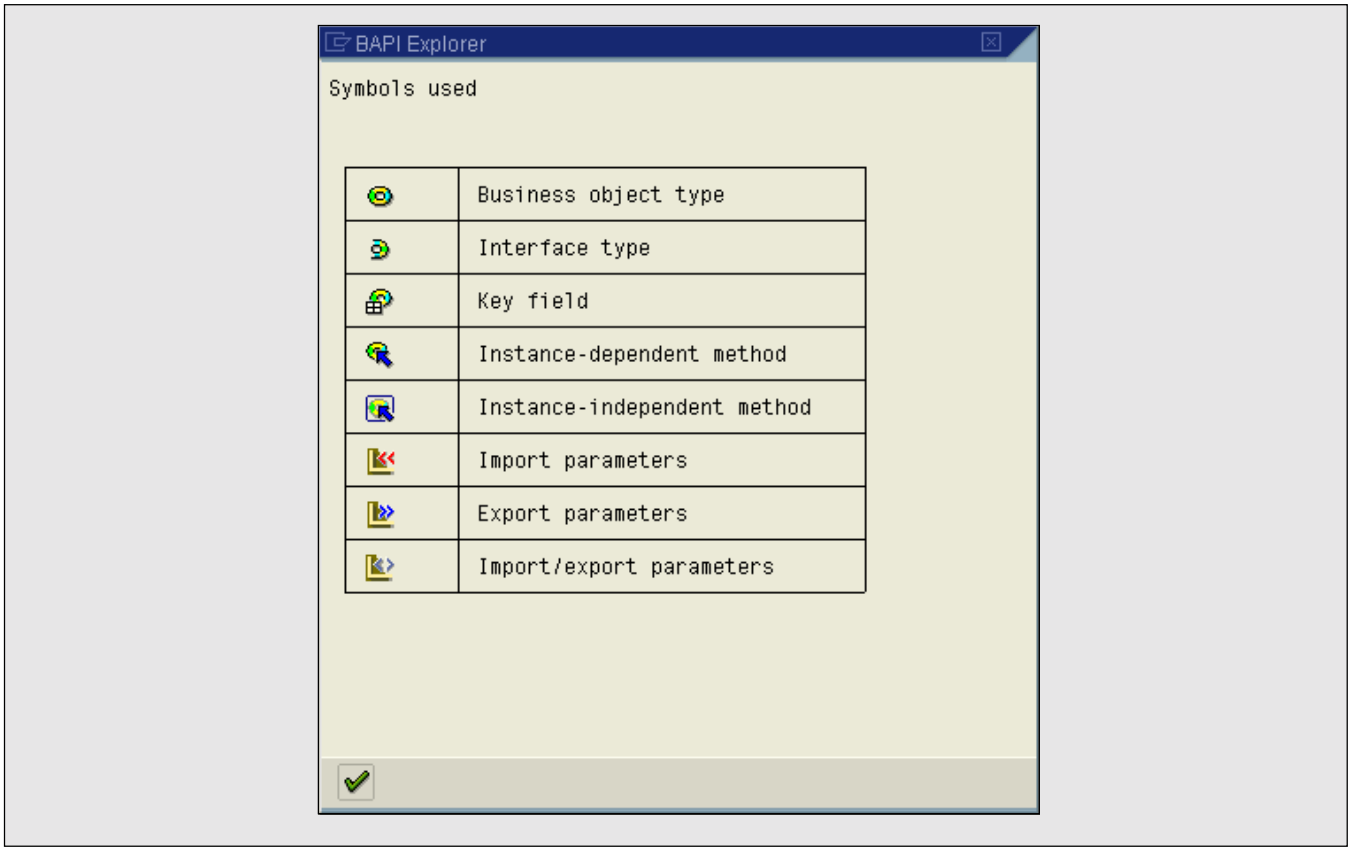
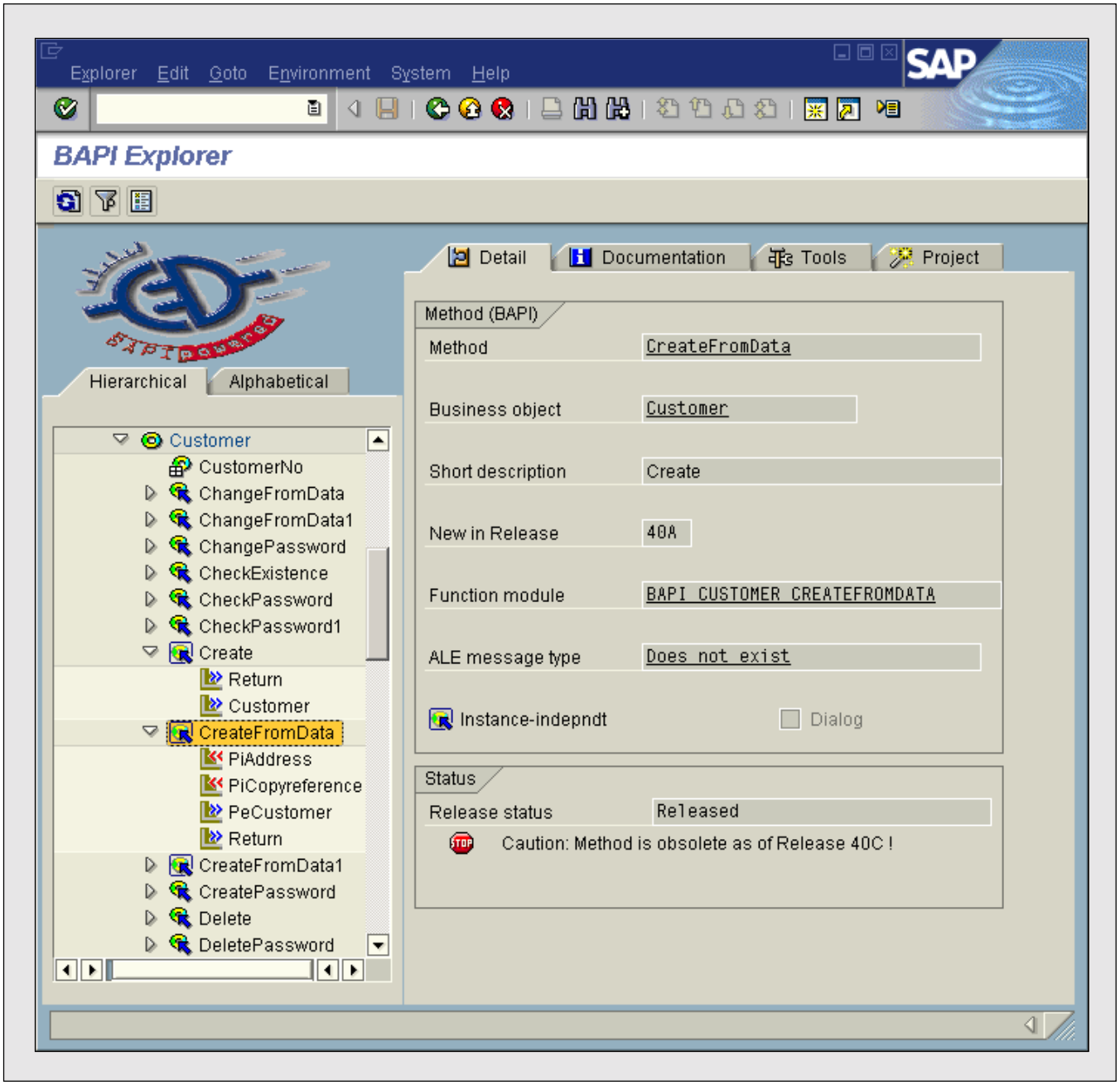
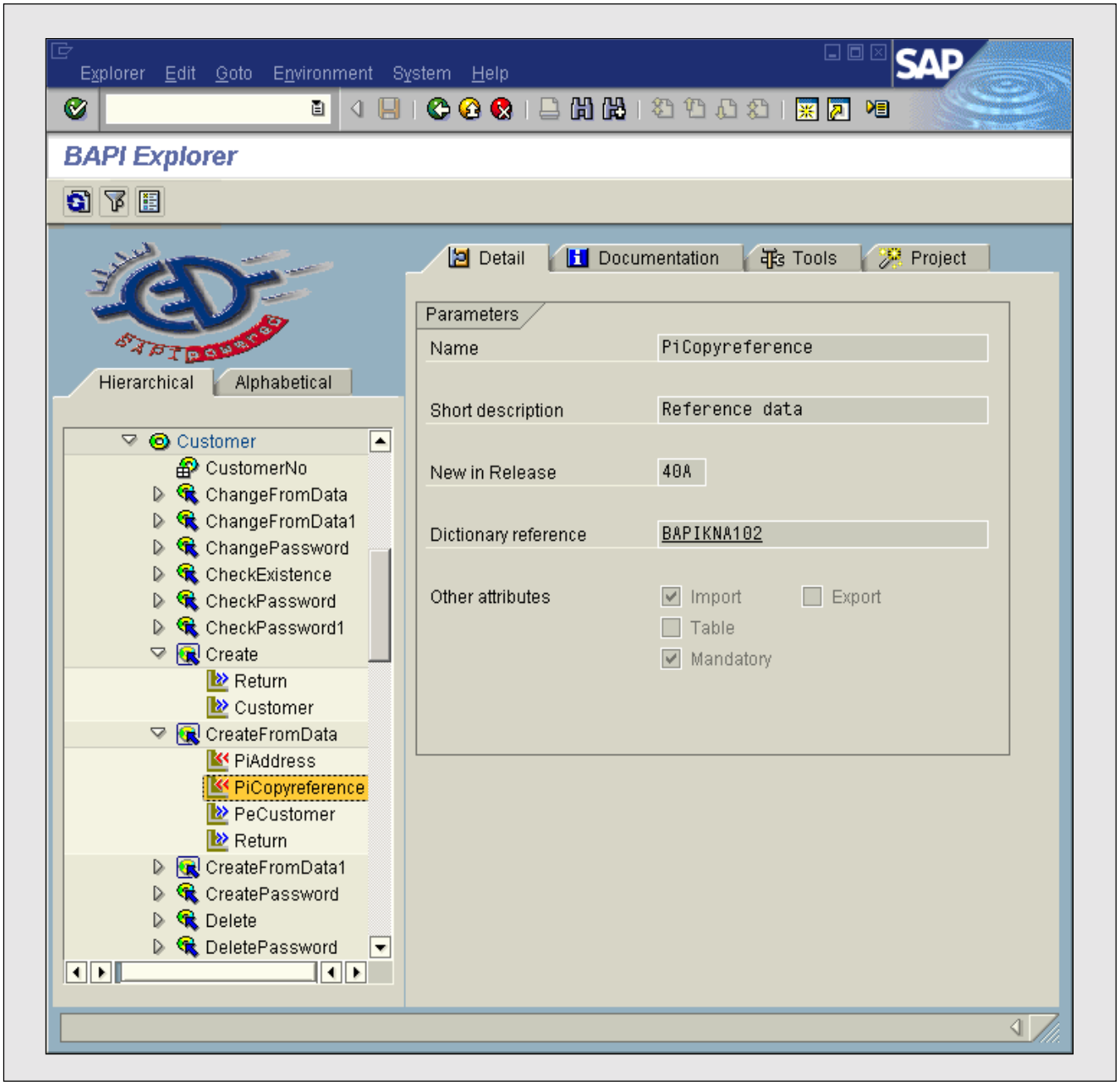


Figure 14 One BAPI of an Object Type



After opening up an object type in the tree (or the alphabetical list), you can see the key fields, BAPIs, and parameters. In **Figure 14**, I have selected the CreateFromData BAPI. It is a class method (instance-independent), and also a factory method, which the BAPI Explorer in its 4.6B incarnation does not tell you, but the name of the BAPI is a dead giveaway, and of course we could always read the documentation to double-check. As you can see, this BAPI is obsolete, and we should use its successor (CreateFromData1) in new projects. No asynchronous calls are possible for this BAPI, because no ALE

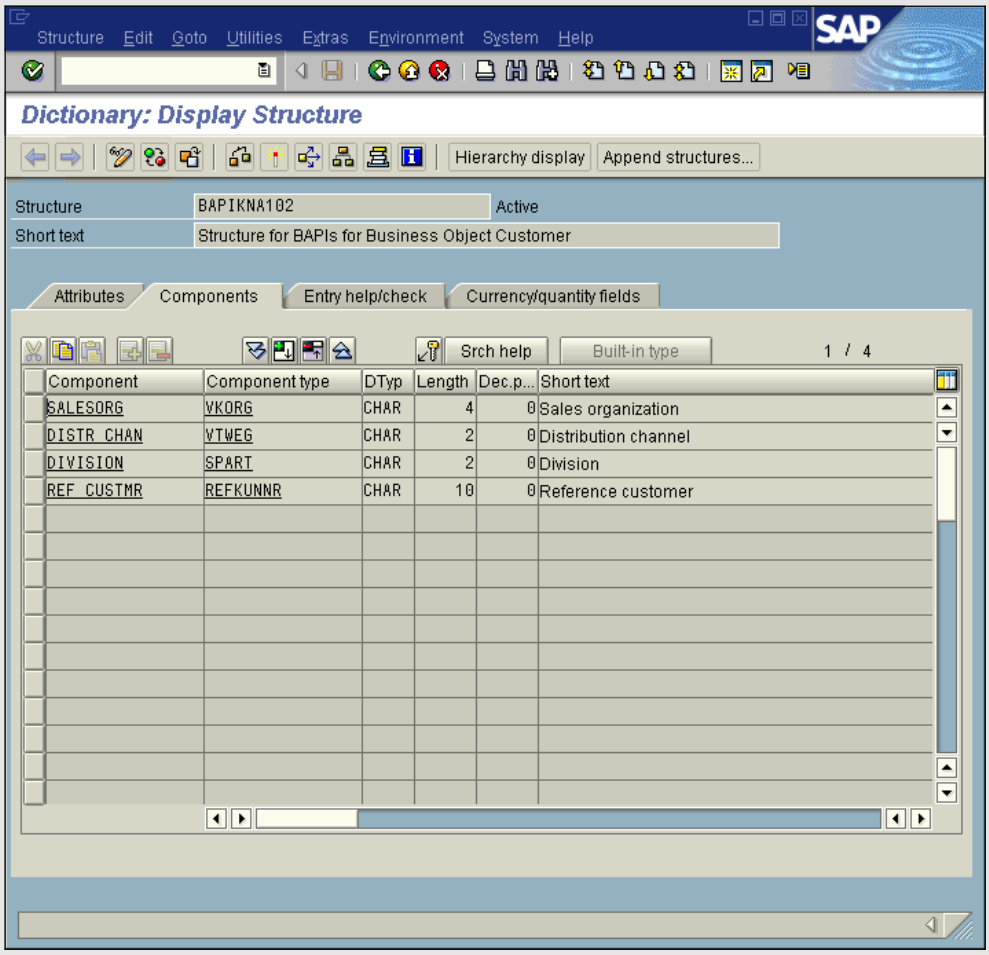
Figure 15 *One Parameter of a BAPI*



message type has been generated yet. Double-clicking on the "Does not exist" text would take you to a wizard that allows you to generate one.

If you select one of the parameters, you get to a screen like the one shown in **Figure 15**. This particular parameter is a mandatory import structure parameter. You can tell that it is not a table parameter since the appropriate check box is unchecked. And you can deduce that it is a structure parameter, because the "Dictionary reference" field contains no hyphen, which means that the parameter

Figure 16 The Dictionary Structure for a Parameter



is based on the complete structure. Scalar parameters show up in the format "STRUCTURENAME-FIELDNAME".

Double-clicking on the text in the "Dictionary reference" field takes you directly to the SAP Data Dictionary (**Figure 16**). Here you see all fields in this structure with their data types, lengths, and descriptions.

Clicking on the other available tabs allows you to look at more information for all the fields. **Figure 17** shows the Search Help/Helpvalues-related information of each field.

To navigate to the RFM implementing a BAPI you select the "Tools" tab in the BAPI Explorer (cf. **Figure 18**). Click on the "Display" button to go to the Function Builder.

Figure 17 The Search Help View of a Structure

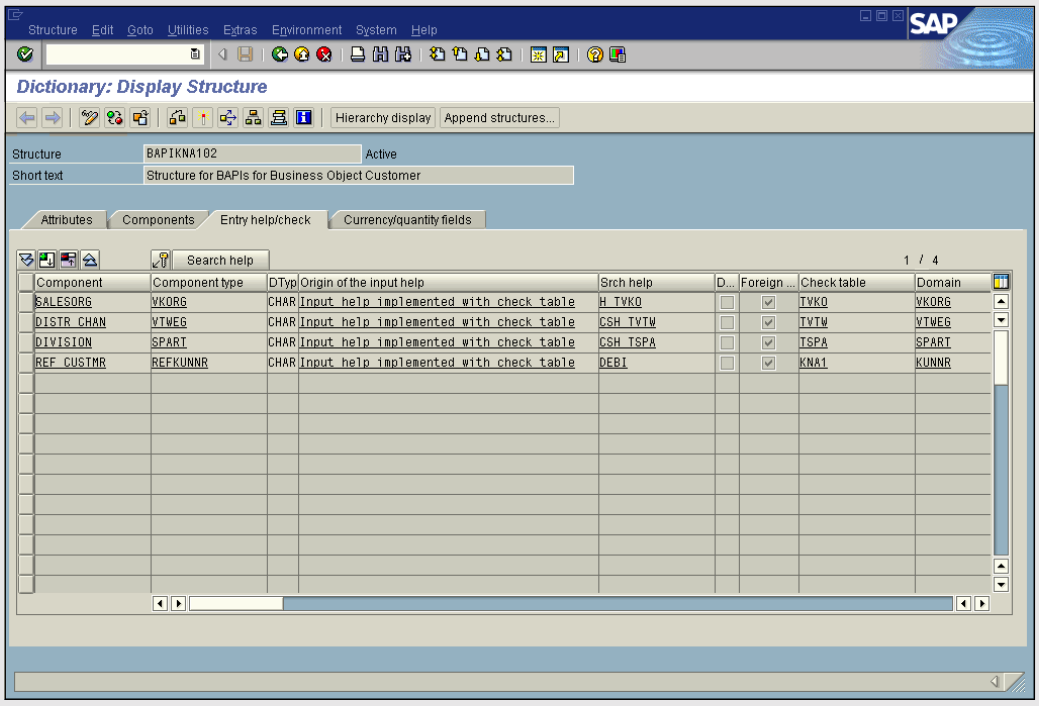


Figure 18 Selecting the Function Builder

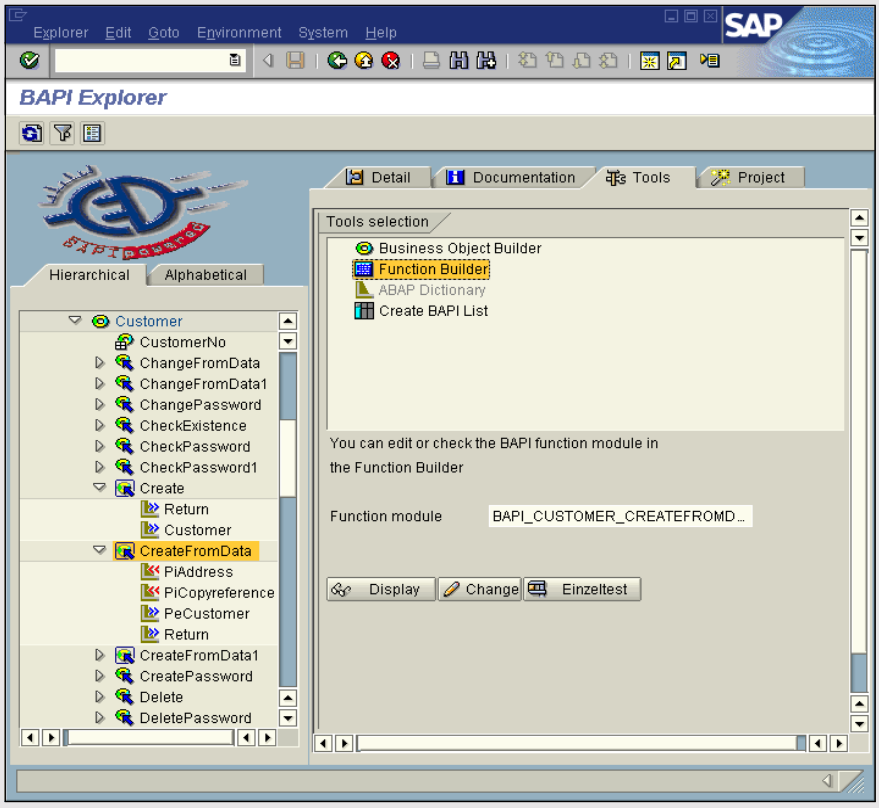


Figure 19 The RFM Behind Customer.CreateFromData

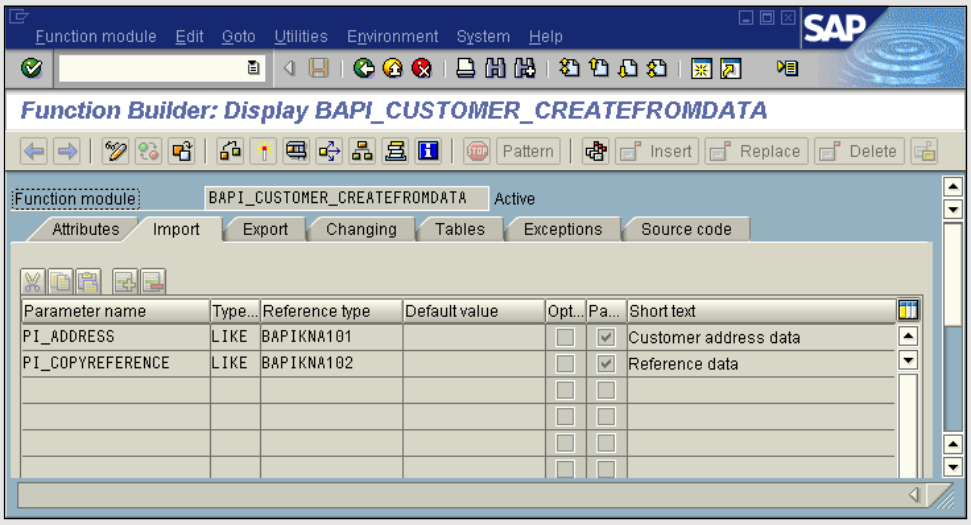


Figure 20 Selecting the Business Object Builder

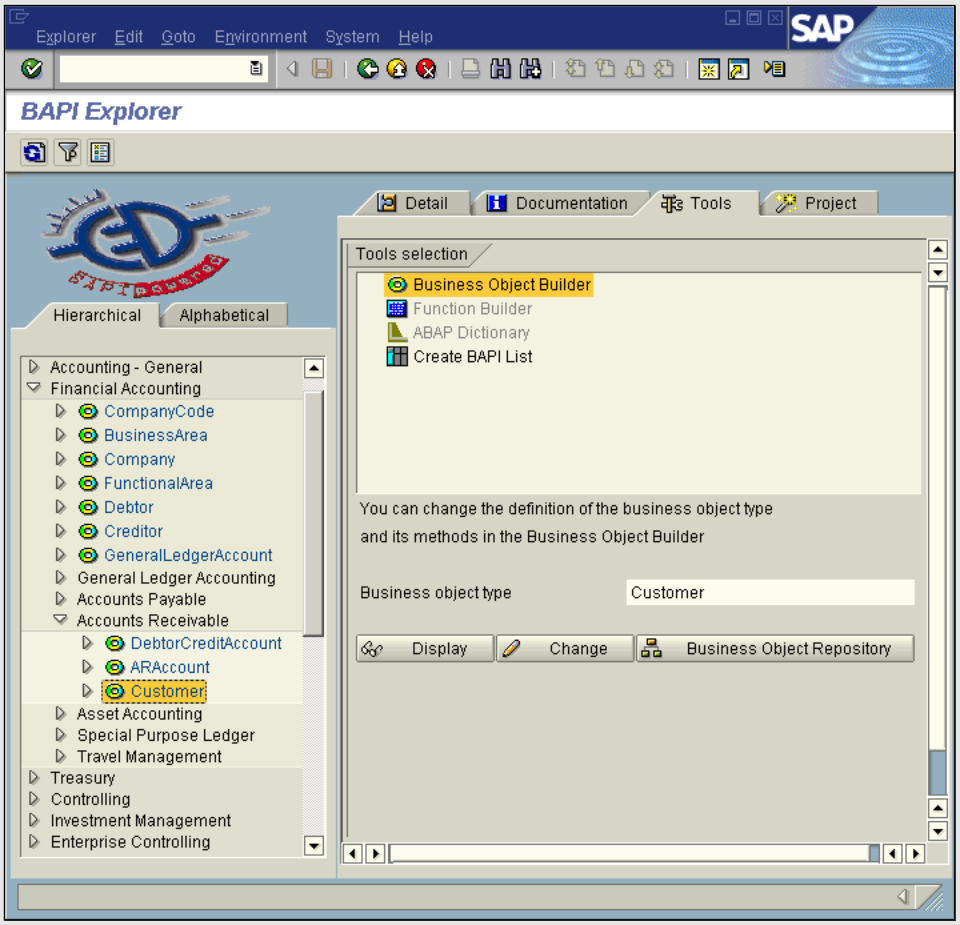
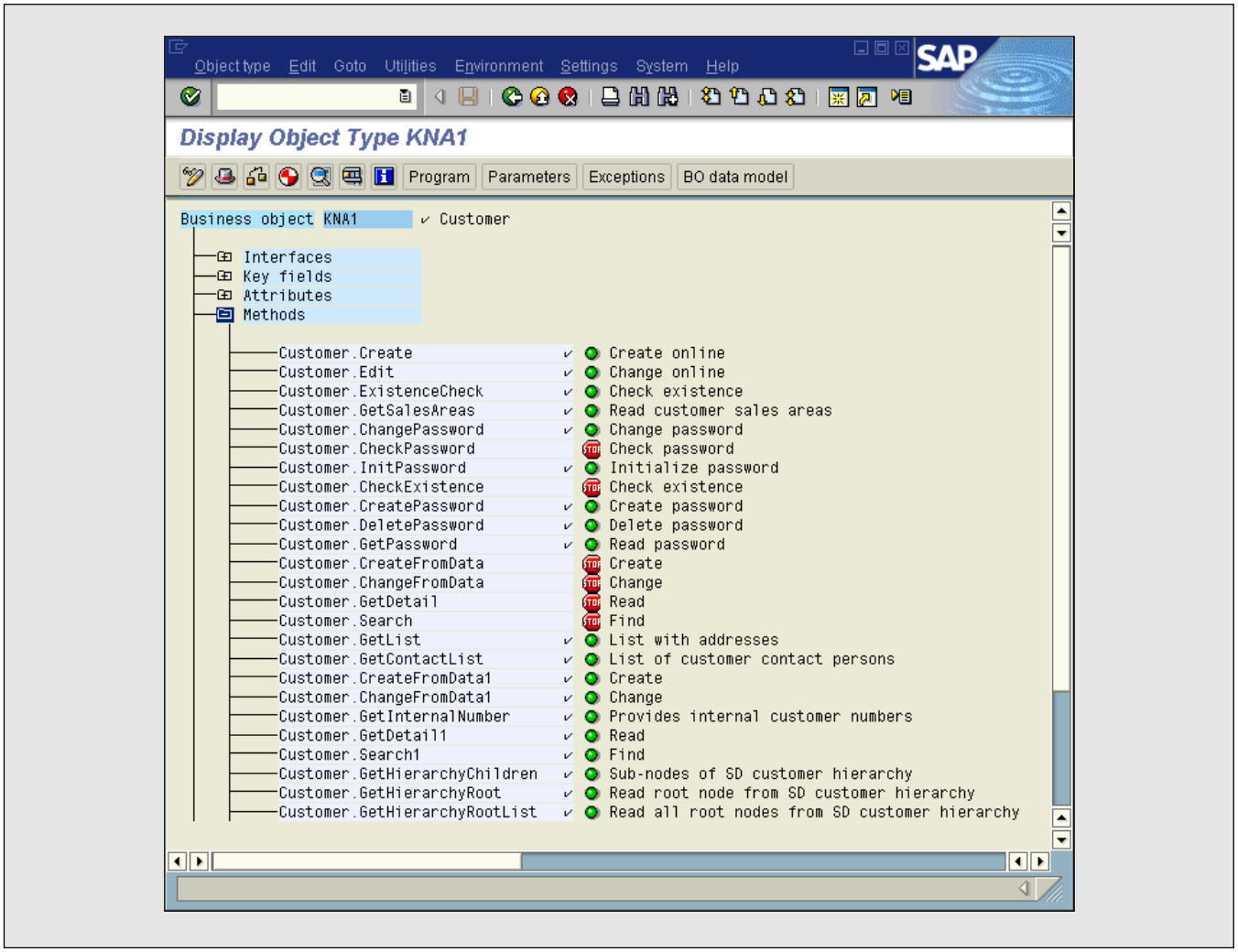


Figure 21 *Selecting the Business Object Builder*



Here you can look at all relevant metadata from the RFM's perspective, including import (cf. **Figure 19)**, export, and tables parameters, as well as exceptions and the ABAP source code. Normally, it will suffice to study a BAPI from the BOR perspective.

If you need more information about an object type than is available directly in the BAPI Explorer, you can always switch to the BOR tools. In **Figure 20**, you can press the "Display" or "Change" button to navigate to the selected object type displayed in the Business Object Browser or Builder, respectively. The "Business Object Repository" button takes you to the Business Object Repository Browser (transaction code SWO3), which shows the same types of information as the Business Object Browser, but has more options selected in its filter by default. (IDocs are included in the hierarchy, for example.)

Figure 21 shows the Customer object type (internal name KNA1) in the Business Object Browser with not only BAPI- but also Workflow-

related information. Interfaces, attributes, and events (scroll down in Figure 21) are of no relevance for BAPI programming.

Again, we can find out the meaning of the icons in the legend (cf. **Figure 22**).

Double-clicking on the object type ("KNA1") in Figure 21 displays the basic data for the object type (cf. **Figure 23**). If you select the "Customizing" tab, you can see that this object type is delegated to "ZZKNA1", which only exists (unless you create one yourself) in IDES systems (used for training and demos by SAP and customers).

In order to see the subclassing required for delegation, you can start another session and use transaction code SWO2 to bring up the Business Object Browser Hierarchy. Drilling down to the

Figure 22 The Business Object Browser Legend

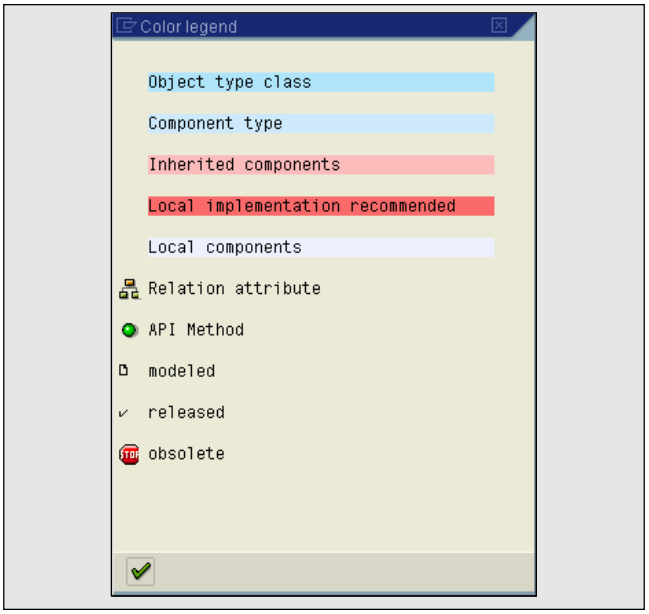


Figure 23 The Basic Data of KNA1

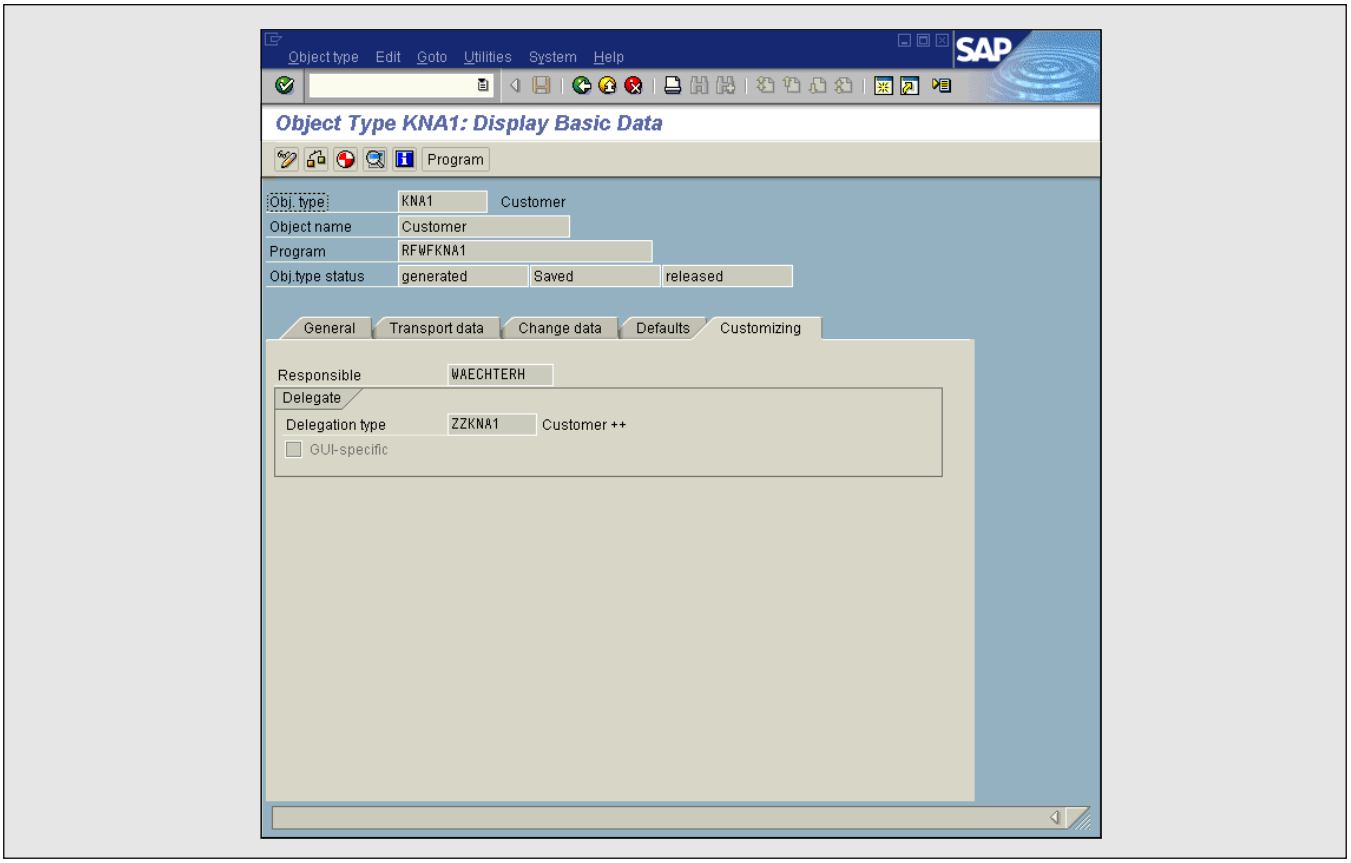


Figure 24 The Customer Object Type in the BOR Hierarchy

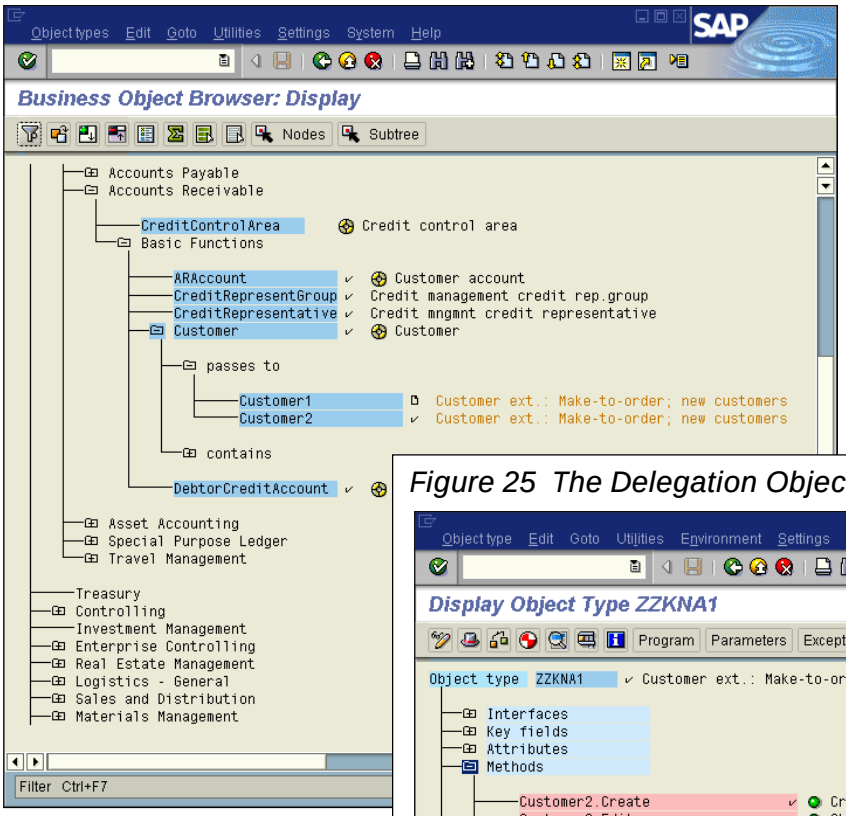
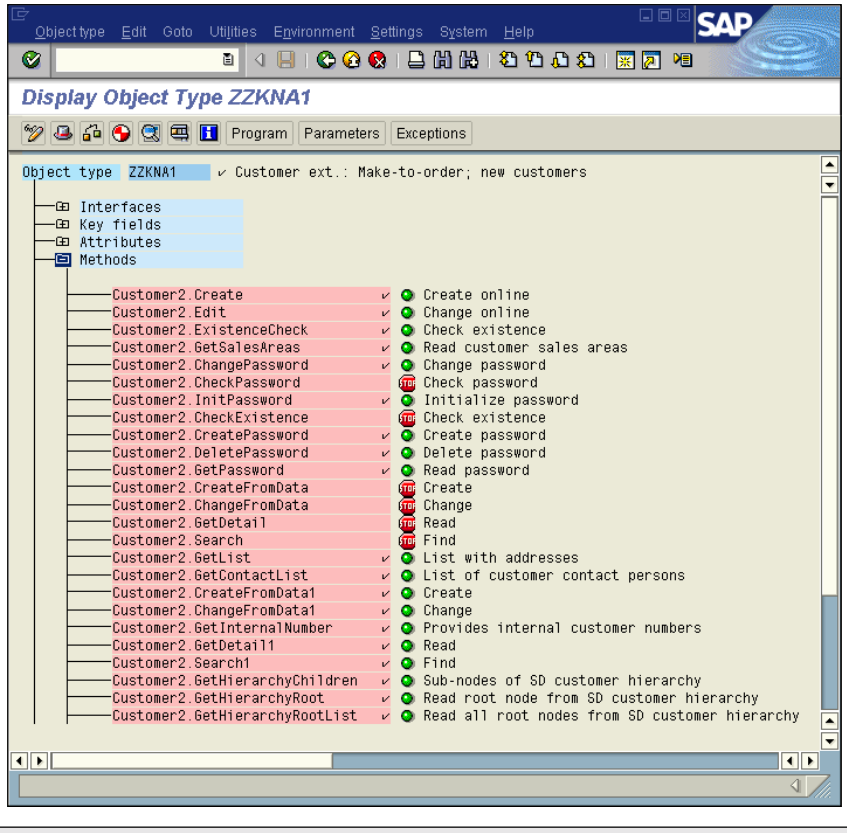


Figure 25 The Delegation Object Type for Customer Hierarchy



Customer object type shows the screen depicted in **Figure 24**. Two subclasses are defined for Customer. The second one (Customer2) is the one that Customer is delegated to. Double-click it to bring up **Figure 25**, which confirms that its object type is "ZZKNA1", the delegation type shown in Figure 23.

Conclusion

By now you should have a firm understanding of the BAPI metadata available in the BOR and how to access it inside the SAP system. This will help you in using BAPIs in your own applications as well as in implementing your own BAPIs.

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 CA925 "Developing BAPI-enabled Web applications with Visual Basic" and CA926 "Developing BAPI-enabled Web applications with Java" classes, which he teaches in Germany and in English-speaking countries. His book on the same subject, "The BAPI Bible for SAP Programmers: The Comprehensive Guide to Integrating SAP Products with Web, Desktop, and Mobile Applications Using Java, Visual Basic, and ABAP", will be published soon by the SAP Professional Journal. 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.

