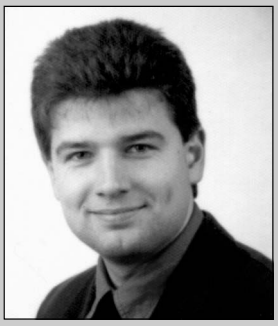
BAPI Programming with Java — Beyond the Basics

Michael Czerniak



Michael Czerniak is a Senior Consultant/Software Architect at SerCon, IBM Global Services, where he specializes in the SAP Business Framework and Middleware. He also teaches a "Programming with BAPIs in Java" training class and is involved in the ongoing development of BAPI technology.

(complete bio appears on page 50)

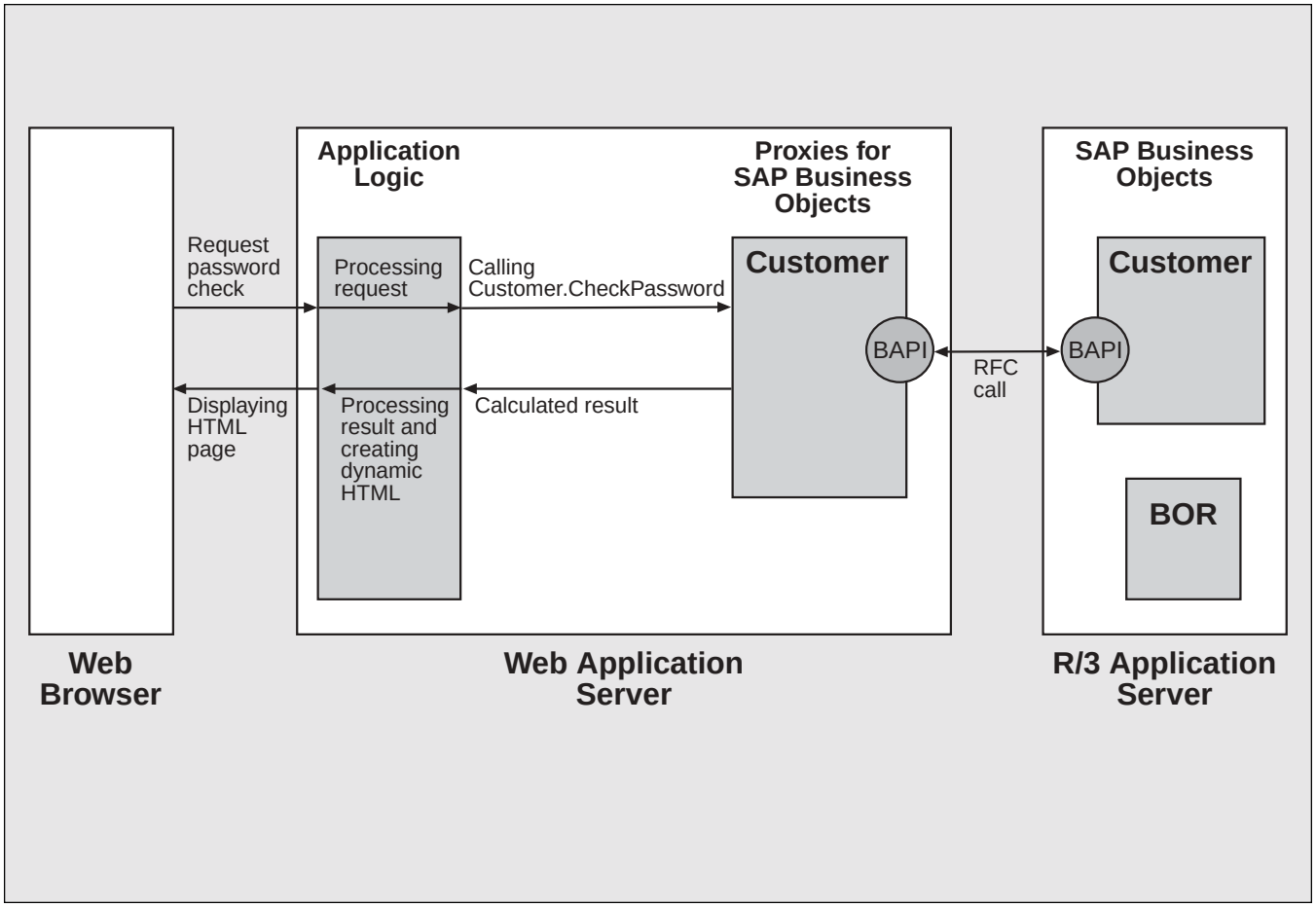
SAP's Common RFC Interface for Java has made it easier for customers to adopt the "outside-in" approach, whereby Java programs running on non-SAP systems can make BAPI calls into an R/3 system to leverage R/3 functionality. The Common RFC Interface, jointly developed by SAP and IBM, is a Java encapsulation of the RFC library that enables non-R/3 applications to communicate with the SAP R/3 application server and utilize the protocol of the RFC library.

Both SAP and IBM have developed middleware solutions around this common interface standard. SAP offers *Java RFC*. IBM offers *Access Builder for SAP R/3*. Both solutions support proxy generation for Business Objects.[1] Recently a third middleware solution was released: *Java eConnector for SAP R/3*, an enterprise connector for HAHT Software's application server, HAHTsite.

How do these middleware solutions support an outside-in R/3 integration effort? E-business is the rage these days, so let's suppose we want to build a multilingual online store in Java, and that this application needs to interoperate with an R/3 system. (Granted, you could buy a ready-made solution for this purpose. I offer this example because it's a relatively easy-to-understand scenario.) A user connects to our online store, fills out a logon/registration screen, browses through various catalogs, which are retrieved from R/3, and clicks on the items he wants. The user can inspect a picture of the item (product description and price) and can check the availability of the selected item. When he has completed his shopping spree, he checks out and the online store automatically creates an R/3 sales order. The selected items will be shipped

[1] Access Builder for SAP R/3 also generates proxies for RFC-enabled function modules.

Figure 1 E-Business Scenario with Java: Performing a Password Check



according to the "normal" R/3 processes. The customer can check the status of his placed sales order(s) at any time.

There's a lot more here than meets the eye. As developers, you need to understand:

- How data will be retrieved from R/3 and then displayed in the user's browser. The customer authentication process and presentation of the product catalog download, for example, both depend on this.
- How a new customer will be created inside R/3. This will be the case every time a user who is logging on for the very first time visits our online store.
- How an availability check can be performed prior to creating a new sales order.
- How a new sales order will be generated.
- How generated sales orders can be tracked.

Figure 1 offers an architectural rendering of a customer password check. **Figure 2** lists all the Business Objects and BAPIs that are involved in this simple scenario.

As you zero in on the specific list of BAPI calls that you will need to facilitate your integration efforts, and then start to work with those BAPIs, you may find a dearth of documentation where you

Figure 2 Business Objects and BAPIs for Our Simple E-Business Scenario

Business Object	BAPI	Description
Customer	Search1	Checks whether or not a (potential) customer exists in R/3
	CheckPassword	Verifies the password of a customer
	CreateFromDat1	Creates a new customer
	CreatePassword	Creates a password for a new customer
ProductCatalog	GetList	Retrieves a list of existing product catalogs
	GetLayoutParam	Retrieves the catalog
SalesOrder	CreateFromDat1	Creates a new sales order
	Simulate	Checks availability of the ordered material and calculates the appropriate prices
	GetList	Retrieves a list of sales orders and their status — for example, “Delivered” or “Not delivered”
BapiService	DataConversionInt2Ext	Converts the R/3 internal representation of a measure unit into the external (language-dependent) representation
	DataConversionExt2Int	Converts the external (language-dependent) representation of a measure unit into the internal representation
Helpvalues	GetList	Retrieves extended metadata for relevant GUI values
RFM	DDIF_FIELDINFO_GET	Retrieves language-dependent field descriptions for the GUI

need it most.[2] That was certainly my experience. Working with Java middleware for SAP R/3 is a somewhat complex affair. Hopefully, this article will help. It’s designed to give you fair warning about some of the nasty details you will encounter when programming with BAPIs in Java, and how best to deal with them. As we delve into the nuts and bolts of the BAPI calls you will need to include in your Java code, I will show you how to:

- Retrieve existing instances (data) from within R/3
- Create a new instance within R/3
- Support transaction handling
- Make dynamic BAPI calls without proxies
- Work with metadata

But First, A Quick Review of What Every Java Programmer Needs to Know About SAP R/3 Middleware

Java applications cannot talk directly to an R/3 application server. A middleware solution, like the ones

[2] SAP’s Interface Advisor is one of the few helpful resources I’ve been able to locate. It represents a centralized pool of information for planning, designing, and implementing interfaces between SAP R/3 and external systems. You can order the Interface Advisor via the SAP Online Store at **http://www.sap.com/solutions/technology/int_adv.htm**.

you see listed in **Figure 3**, is needed to translate Java-based BAPI calls and data types into ABAP, and vice versa. (Figure 3 also lists each solution's Integrated Proxy Generator — IPG.) This intermediary step is what makes it possible to transparently incorporate R/3 functionality into a Java application.[3]

*Figure 3 Middleware Implementations: An Overview**

Name:	Access Builder for SAP R/3
Vendor:	IBM Corp.
IPG:	Access Builder for SAP R/3
More info:	www.software.ibm.com/ad/vajava
Availability:	Integrated with VisualAge for Java Enterprise Edition, which has been available in version 3 since November 1999
Name:	Java eConnector for SAP R/3
Vendor:	HAHT Software Inc.
IPG:	Java eConnector Proxy Generator
More info:	www.haht.com/jec
Availability:	Integrated with HAHTsite 5.0
Name:	Java RFC
Vendor:	SAP Labs
IPG:	SAP Assistant
More info:	www.saptech.com/usa
Availability:	Packaged with SAP Automation 4.5A and higher

* There are other middleware implementations available for Java. These, however, are proprietary solutions that do not offer support for the Common RFC Interface.

[3] Michael Czerniak's last article, "BAPI Basics When Programming with Java" (*SAP Professional Journal*, Premiere Issue), shows Java programmers how to start using the Common RFC Interface for Java with either the SAP or IBM middleware implementation. It describes how you initialize the middleware, the steps to set up a connection to R/3, and the way to call a BAPI that returns data from R/3. These are the basic building blocks for building Java applications that can talk to SAP Business Objects.

Beneath these middleware solutions — the portion of the architectural diagram shown in **Figure 4** that appears below the dotted line — sits the Common RFC Interface for Java.

The Common RFC Interface defines generic rules for data mapping and for retrieving metadata for Business Objects and BAPIs. Through this access layer, a Java programmer can build applications that access R/3, yet are not dependent on any one vendor's middleware solution. You can actually mix and match the different middleware implementations you see beneath the dotted line, which represents the Common RFC Interface for Java. You can generate, for example, your Business Object proxies with HAHT's Java eConnector for R/3 and run your Java code with IBM's Websphere application server.[4]

Now take a look at **Figure 5**. Notice how the different vendors have defined different object models for constructing Business Object proxies on top of the Common RFC Interface. It's not just that the names of the generated proxies are different; the way you instantiate a proxy object (e.g., the number and order of the parameters required for a BAPI call) is different too.

The way you define your BAPI calls depends on which vendor's object model you are using to map an R/3 Business Object to a Java class. Java eConnector, for example, generates a Java class for each Business Object with integrated parameter container and wrapper classes as nested top-level classes. This dramatically reduces the number of generated proxy classes and eases navigation to locate the parameter containers and further parameters.

[4] This flexibility provides a new dimension of openness for ISVs, who can now integrate their products with R/3 by either choosing an existing middleware solution or building one of their own. The Common RFC Interface for Java and its flexibility provides the foundation for building R/3-related Java components and component libraries for different middleware implementations. These components could be built to encapsulate complex logic in R/3 integration scenarios and serve as connectors to various industry standards like EJB, CORBA, and DCOM.

Figure 4 *SAP Middleware: Architectural Overview*

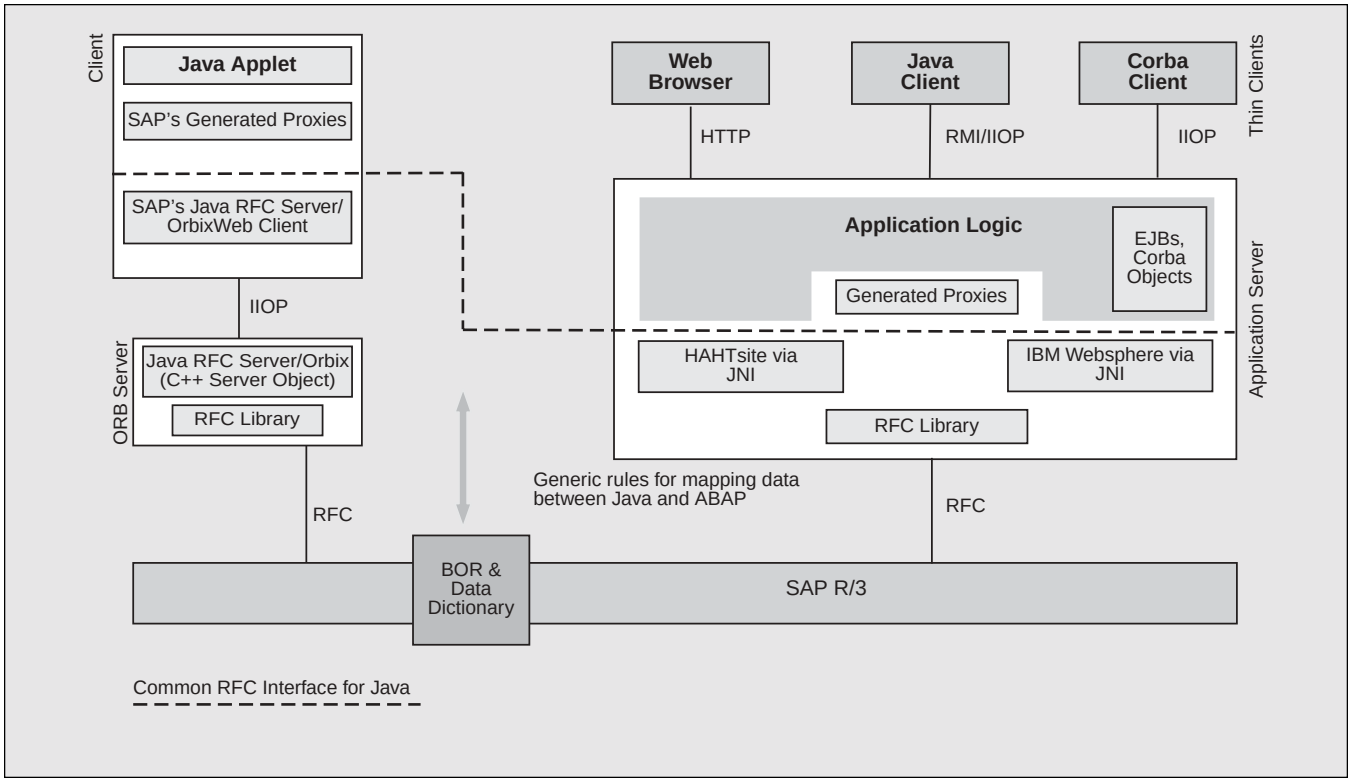
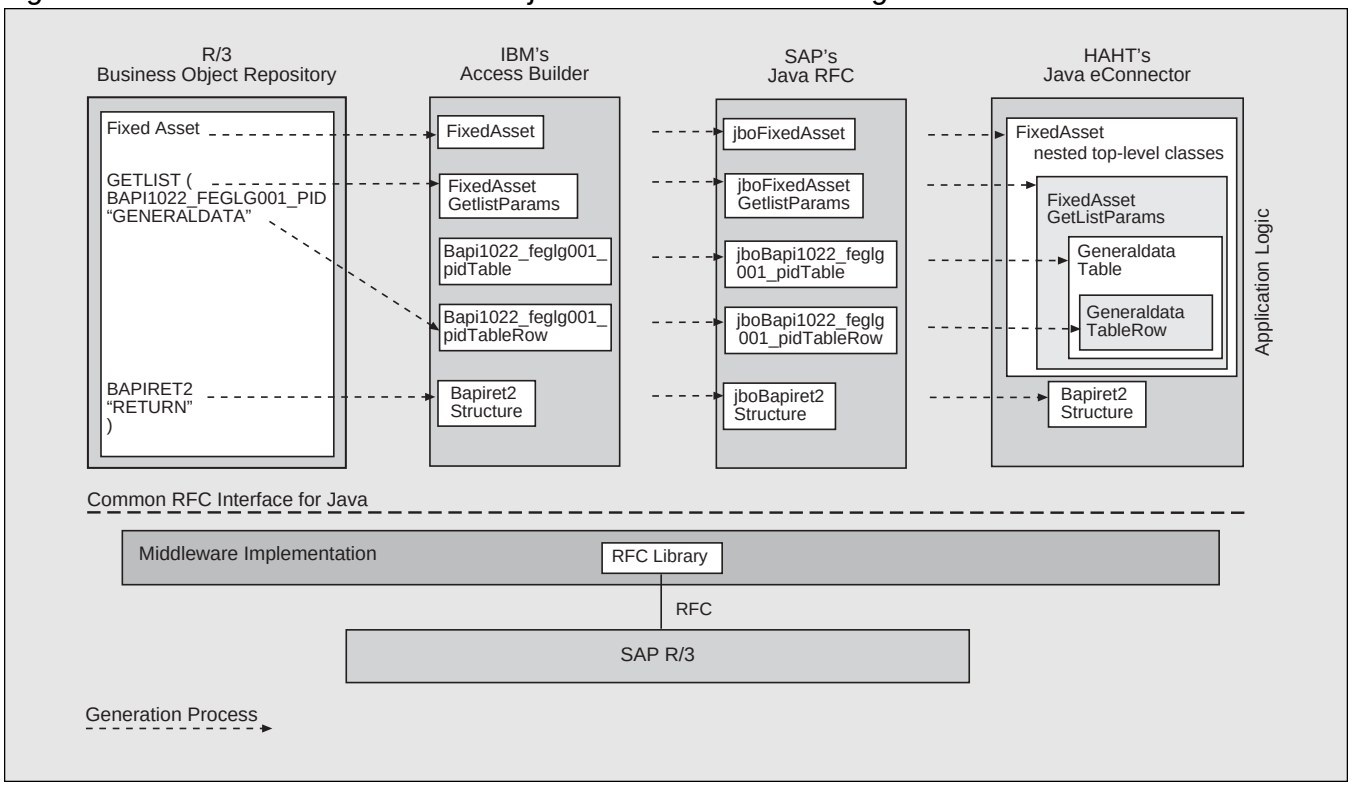


Figure 5 *Different Object Models for Generating Proxies*



In my last article, I described the different proxy classes and wrapper classes generated for a specific Business Object. These classes can be generated with the proxy generator tools that come packaged with the middleware implementations listed back in Figure 3. In the sections that follow, I will show you how you retrieve a list of instances of Business Objects, even if the Business Object has no GetList BAPI. A number of other real-world challenges will be highlighted, too!

Retrieving Instances from R/3

A BAPI is defined as a method of a Business Object. After generation of a proxy for the Business Object, the BAPI will be represented as a Java method.

There are two different types of BAPIs defined in R/3:

- **Instance-independent BAPIs**, which can be called directly on the generated proxy object type.
- **Instance-dependent BAPIs**, which cannot — you must first instantiate an object instance before calling the BAPI.

Both types of BAPIs can also be **instance-creating** BAPIs, which create instances of Business Objects in R/3 (e.g., the creation of a new sales order would be done by an instance-independent BAPI, which also has instance-creating behavior). These BAPIs are often called "factory BAPIs."

Instance-independent BAPIs are mapped by the tool that generates the proxies to *static* Java methods (unless they are also factory BAPIs and you use SAP or IBM's proxy generators). These BAPIs are not related to a specific instance of a Business Object in R/3. The GetList BAPI[5] (and, as you will see a bit later on in this article, the Create BAPI as well) is an instance-independent BAPI. The GetList BAPI can be called directly as a Java method on the proxy class, as shown in **Listing 1**.

The call you see in Listing 1 is the one you would use in your Java code if you were working with the HAHT middleware solution. It retrieves a list of existing CompanyCode instances from R/3. Here, it is not necessary to specify a parameter container, because the parameter container was generated as a nested top-level class.

[5] As long as I refer to the BAPIs in R/3, I'll give the BAPI names in mixed-case notation — i.e., GetList. Java naming conventions restrict method names to beginning with a lowercase letter, therefore I refer to a Java method representing a BAPI on the proxy object using the Java (lowercase) notation — i.e., getlist.

Listing 1: Calling CompanyCode.GetList with HAHT Java eConnector

```
1  // The connection has to be opened up before the call
2  CompanyCode.getList(aConnection);
```

Listing 2: Calling CompanyCode.GetList with IBM Access Builder

```
1  // The connection has to be opened up before the call
2  CompanyCodeGetlistParams ccGetListParams = new CompanyCodeGetlistParams();
3  CompanyCode.getlist(aConnection, ccGetlistParams);
```

Listing 3: Calling CompanyCode.GetList with SAP Java RFC

```
1  // The connection has to be opened up before the call
2  jboCompanyCodeGetlistParams ccGetlistParams = new jboCompanyCodeGetlistParams();
3  jboCompanyCode.getlist(ccGetlistParams, aConnection);
```


If you were working with the IBM middleware solution, you would define the GetList BAPI call a bit differently, adding a parameter container, as is shown in **Listing 2**. In line 2, the parameter container for the BAPI call is explicitly instantiated. No explicit instantiation of an instance of type CompanyCode, however, is required (line 3) because getlist is a static method. The same call with SAP's proxies would look like the code shown in **Listing 3**. Notice that the parameters are switched (compared to IBM's Access Builder): the parameter container must be specified first, then the connection object.

> ✔ **Tip**
> *Before defining a BAPI call, you should inspect the generated proxy objects. These objects tell you which parameters are required to define the BAPI call, and the order in which they must be given to the BAPI. The documentation and the sample code that come with the middleware implementation might be helpful as well. If you use a development environment with an integrated "Coding-Help-While-Writing-Code" feature (i.e., CodeInsight or CodeClue[6]), you won't have a problem identifying the correct parameters for a BAPI call, because the Help feature tells you everything you need to know.*

GetList in HR — A BAPI for Time-Dependent Business Objects

Considering how many developers work with HR Business Objects, I felt that this discussion would not be complete without mentioning that the GetList BAPIs in SAP's Human Resources (HR) component behave differently than the standard GetList BAPI. (If you don't have occasion to work with these BAPIs, just skip to the next section.)

Unlike CompanyCode.GetList, which returns a list of existing object instances, the HR GetList BAPIs retrieve a list of existing instances of a specific *infotype*. These infotypes, some of which are shown in **Figure 6**, represent HR-specific entities.

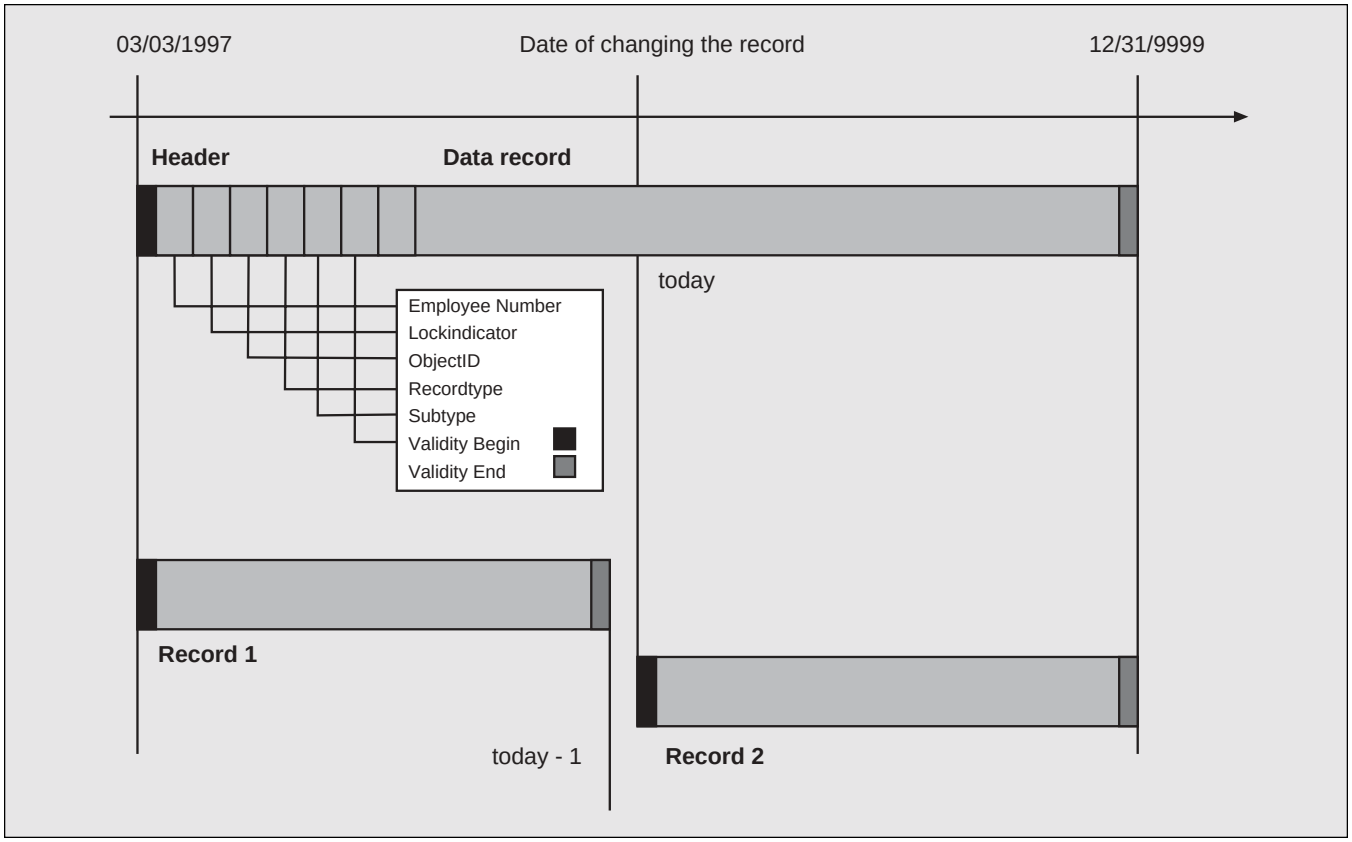
An instance of an infotype describes specific, time-dependent attributes of an employee's dataset. Consider the situation where an employee gets married and assumes her husband's last name. R/3 must still store her maiden name. Her married name is valid from the day of the marriage until the "ultimate-end-of-the-world-day," which is defined as 12/31/9999 in HR. A new record to contain the new

[6] The term for this integrated help feature depends on the development product you use. Inprise, for example, calls this feature "CodeInsight" and IBM refers to it as "CodeClue."

Figure 6 Infotypes Related to the Employee Business Object in HR

Infotype (Business Object)	Description
EmployeePersonalData	General description of employee data — e.g., last name, first name, gender, nationality, etc.
EmployeePrivAddress	Detail information regarding street, city, postal code, phone number, etc.
EmployeeBankDetail	Banking account details — e.g., account number, currency, country code of the bank, etc.
EmployeeBasicpay	Information about an employee's salary details
EmployeeFamilyMember	Family member-related fields
EmployeeIntControl	Employee's location inside the organization — e.g., building number, room number, extension, etc.

Figure 7 Changing Employee Data in HR



name is created for this employee for the infotype EmployeePersonalData, while the old record remains available, as shown in **Figure 7**.

In **Listing 4**, I present some sample Java code for retrieving a list of valid records for an EmployeePersonalData infotype with employee number 1000 in the interval [01/01/1980, 12/31/9999].[7] The BAPI GetList will return a list of records of a specific infotype in a defined time interval for the employee, and the code writes the result to standard-out (stdout).

There is an intentional error in this code listing. Did you catch it? Take a look at line 8. The date specified there is November 31, 9999. (Thirty days has September, April, June, and November…!) The Java Gregorian Calendar implementation behaves strangely at this point, but this behavior is documented, so don't be too concerned by it. This command returns the ultimate HR end date, 12/31/9999.

In summary, the HR GetList BAPIs are used to retrieve valid records of an infotype of a specific employee. However, we are interested in real employee data, like first name, last name, address, etc. The method that can be called to retrieve this data is GetDetail. But GetDetail is an instance-dependent method — you have to instantiate the Business Object before you can call GetDetail. Therefore a couple of key fields like employee number and time interval must be specified. If you compare the key fields of the Business Object with the fields of the table returned by GetList, you will find that they are identical. You can use the GetList "output" as "input" for the instantiation of the Business Object and the following GetDetail call. After all, there must be a mechanism (in this case, GetList) to

[7] This code sample utilizes proxies generated with SAP Assistant, but can easily be adapted to utilize proxies generated by either the IBM or HAHT proxy generators.

Listing 4: Retrieving Valid Records for EmployeePersonalData

```
01  jboEmployeePersonalDataGetlistParams aEPDGetlistParams = new
    jboEmployeePersonalDataGetlistParams();
02  // Set import parameters for getlist call
03  aEPDGetlistParams.setEmployeenumber(new java.math.BigInteger("1000"));
04  java.util.Calendar cal =
    java.util.GregorianCalendar.getInstance();
05  cal.clear();
06  cal.set(1980,1,1);     // 01/01/1980
07  aEPDGetlistParams.setTimeintervallow(cal.getTime());
08  cal.set(9999,11,31);   // 12/31/9999
09  aEPDGetlistParams.setTimeintervalhigh(cal.getTime());
10  // The connection has to be opened up before the call
11  jboEmployeePersonalData.getlist(aEPDGetlistParams, aConnection);
12  jboBapipakeyTable aPA = aEPDGetlistParams.getPersonaldatakey();;
13  jboBapipakeyTableRow cRow;
14  for (int i = 0; i < aPA.getRowCount(); i++) {
15     cRow = aPA.getRow(i);
16     System.out.println("EmpNo.:\t" + cRow.getEmployeeno());
17     System.out.println("LockInd.:\t" + cRow.getLockindic());
18     System.out.println("ObjectID:\t" + cRow.getObjectid());
19     System.out.println("RecordNo.:\t" + cRow.getRecordnr());
20     System.out.println("Subtype:\t" + cRow.getSubtype());
21     System.out.println("Validbegin:\t:" + cRow.getValidbegin());
22     System.out.println("Validend:\t:" + cRow.getValidend());
23     System.out.println("—End of record—");
24  }
```

provide you with all the information that is required to instantiate an infotype proxy, otherwise it wouldn't be possible to call instance-dependent BAPIs for that infotype. All this is a result of the time-dependent behavior of HR Business Objects.

What Happens When Business Objects Do Not Have a Defined GetList BAPI?

Those of you who already have R/3 Release 4.6 can be happy. This release introduces support for matchcode searches for customers and suppliers. The next section describes how you deal with BAPIs without the matchcode search capability (before Release 4.6). The following Business Objects currently do not have a GetList BAPI defined[8]:

- Debtor
- Creditor
- Customer
- PurchaseRequisition
- MaterialReservation

So, if you thought that a GetList method would provide you with all existing instances of Debtor and their key fields, think again. There is no GetList BAPI defined for Debtor! To retrieve a list of all debtors identified by their debtor ID (or customer ID) from R/3, in the absence of a GetList BAPI, you have to utilize another BAPI, like Find, and perform a "normal" search. (Remember, before Release 4.6 there is no matchcode search capablity.) The result of the Find call will return the required information (DebtorId or CustomerId). If you want to retrieve

[8] Please note that this is not a complete listing and that this listing may be subject to change.

Figure 8 The "Find" BAPI Parameters

Type	Name	Data Element	Description
Import	MaxCnt	SYTMAXL	Maximum count of records
Export	Return	BAPIRETURN 1	Return code
Import/Export	ResultTab	BAPI1007_8	Contains the result of matching records
Import/Export	**SeloptTab**	**BAPI1007_7**	**Contains the search criteria**

Figure 9 Fields of the SeloptTab Table: The "Find" BAPI's Search Criteria

Field Name	Data Element	Value Table	Description
COMP_CODE	BUKRS	T001	Company code
TABNAME	TABNAME	DD02L	Table name
FIELDNAME	FIELDNAME	DD03L	Field name
FIELDVALUE	FIELDVALUE	—	The value to be searching for

information like address, account details, etc., for a debtor, you must call Debtor.GetDetail. The Debtor.GetDetail BAPI is an *instance-dependent* method. This means you must know the key field for this Business Object, which is DebtorId (or CustomerNo). (I will provide more detail about instance-dependent methods in the section that follows.)

One note of caution — figuring out how a BAPI like Find or Search works can be quite difficult. How, for example, can you know which set of import parameters is required? I recommend SAP transaction SE37 for this purpose and executing the BAPI with SAP GUI before using it within your Java program.

The Find BAPI is characterized by the import and export parameters shown in **Figure 8**. To really understand this BAPI's search mechanism, however, you must look at **SeloptTab** (**Figure 9**), the table that holds the search criteria. Note that the underlying data element is **BAPI1007_7**.

The **Value table** defines a list of possible entries for table name and field name (DD02L and DD03L). To make sure that the Find call will work successfully, you have to specify proper values for the fields TABNAME and FIELDNAME. One nice little detail here is that only tables that contain the CustomerId (data element KUNNR) can be used for the search. Examples of tables that fit this criterion are KNA1, KNB1, etc. All valid table names are listed in the value table DD03L. You can use SAP transaction SE16 and inspect the table DD03L with selection KUNNR as field name. There's no getting around this procedure. You have to contend with DD02L and DD03L to figure out which values are allowed for TABNAME and FIELDNAME. Otherwise, the BAPI call will abort with an error message.

Figure 10 shows the result after calling the Remote Function Module BAPI_DEBTOR_FIND, which represents the BAPI Debtor.Find. The list of debtor IDs shown in **Figure 11** is the result of specifying the following values for the fields in SELOPT_TAB[9]:

- COMP_CODE := 1000
- TABNAME := KNB1
- FIELDNAME := BUKRS
- FIELDVALUE := 1000

[9] It makes no sense to specify BUKRS as a field name, because this field is already specified by COMP_CODE. But since wildcards are not working properly here, it's the only way to get a full list. You can, of course, figure out other selection criteria that will result in the return of a complete list of debtors.

Figure 10 Performing the Find Call with Transaction SE37

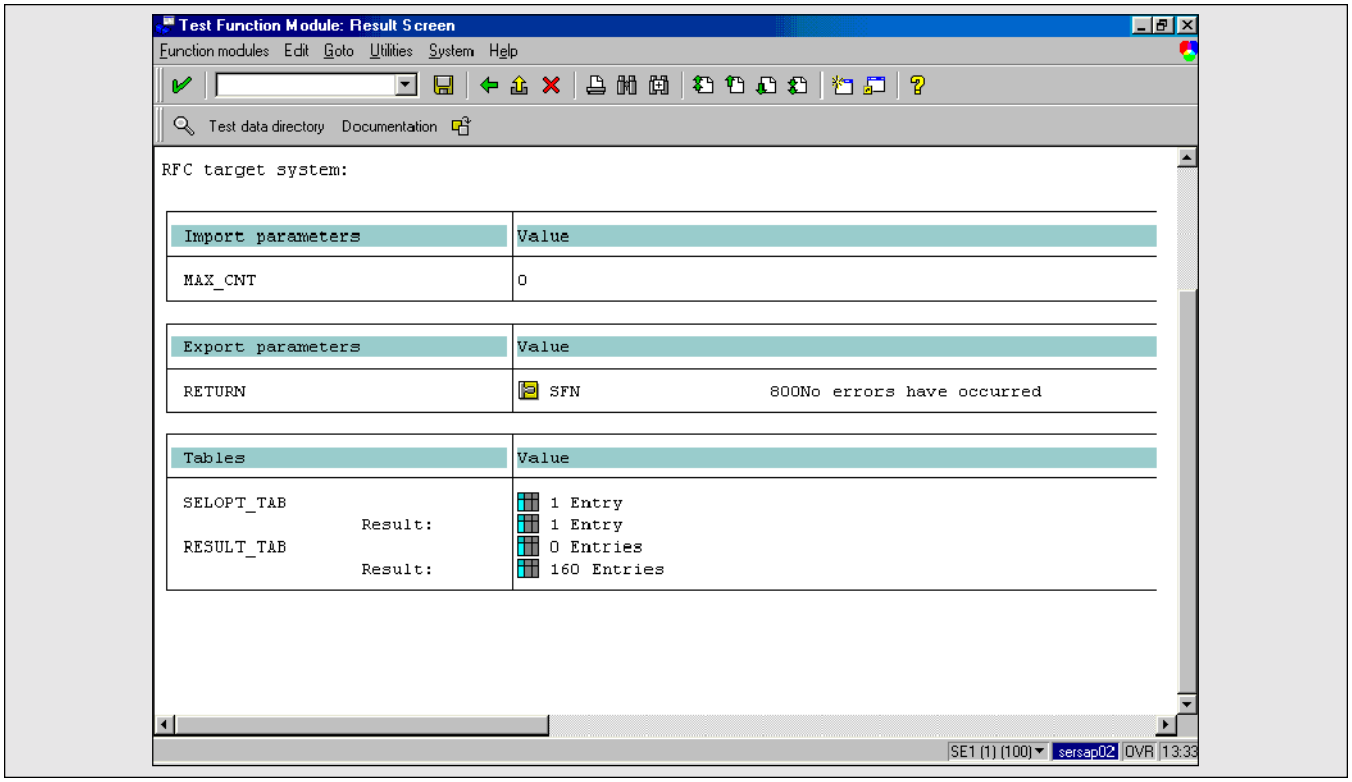
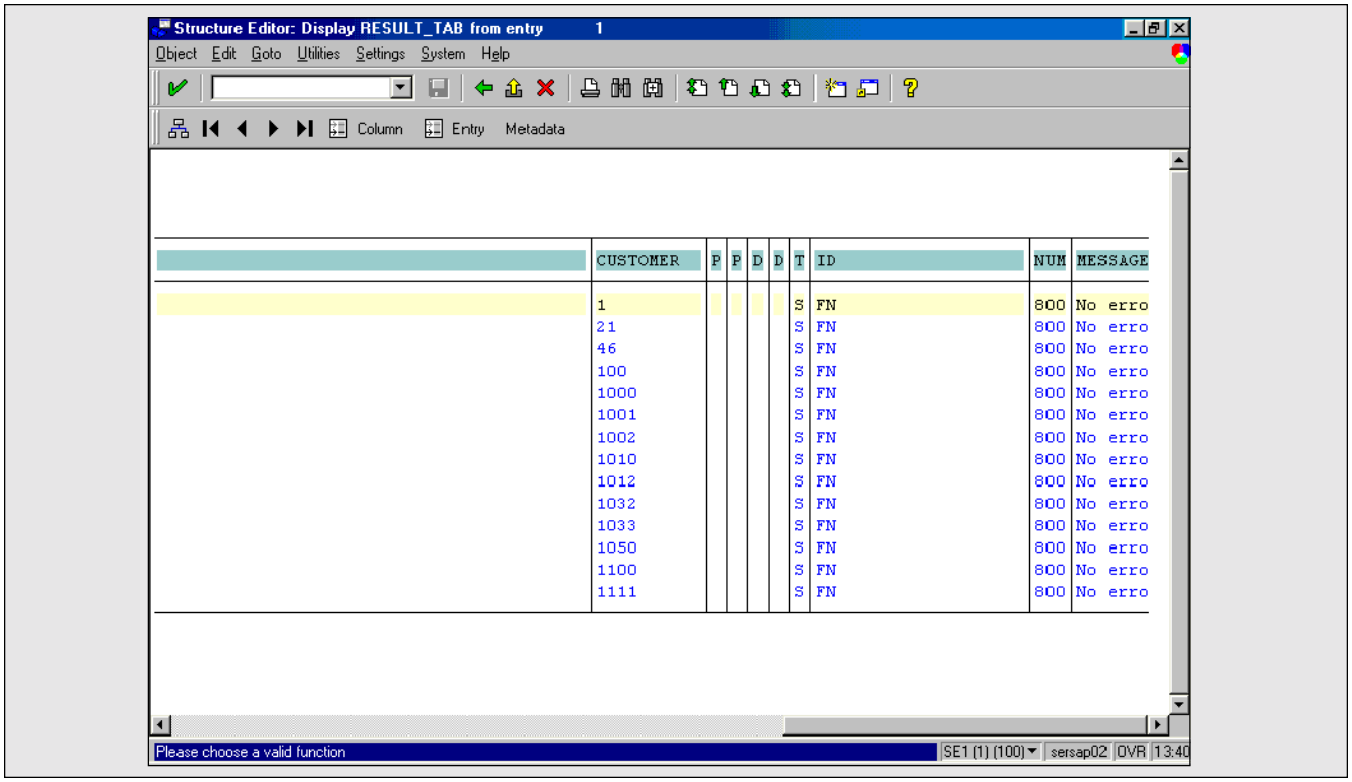


Figure 11 Result of a Debtor.Find Call: Customer Number for Instantiating the Debtor Object



> ✔ **Tip**
> *Often the Business Objects without a GetList BAPI do provide other BAPIs that enable you to instantiate the proxy. Always analyze the existing BAPIs of the current Business Object and do not forget to analyze related Business Objects.*

The column we are initially interested in is the customer number (CUSTOMER) column. By the way, customer number and debtor number (DebtorId) are identical because the underlying data element is KUNNR. You need the DebtorId prior to instantiation of the Debtor Business Object — therefore, the DebtorId is required, which you can take from the column CUSTOMER.

Having gone through these search gyrations, which admittedly are no fun, we are finally in a position where we can now instantiate the proxy for debtor with one of the DebtorIds (again, the column CUSTOMER) shown in Figure 11. You can, for example, retrieve detail information like address or bank details for all listed debtors. I will show you the steps required for instantiation of a proxy and calling GetDetail (instance-dependent BAPI) in the next section.

Instance-Dependent BAPIs

Instance-dependent BAPIs are related to a specific instance of a Business Object. The instance can be exactly identified by a set of key fields. For CompanyCode, the one and only key field is CompanyCodeId. A proxy that will be instantiated with a CompanyCodeId 1000 relates to exactly the dataset of this specific CompanyCode in R/3. You can see this in **Listing 5**, where line 2 instantiates a new instance of a proxy related to the CompanyCode with CompanyCodeId := 1000, and in line 3, the BAPI call is performed. These two lines of code only work with the HAHT Java eConnector middleware implementation.

The different middleware solutions may require different parameters, like a valid connection or a parameter container. The code shown in **Listing 6**, for example, works with the IBM Access Builder. Here, a parameter container must be instantiated. This is done in line 3. Two parameters are required: a valid connection object and a parameter container for the GetDetail call. In line 4 the BAPI call is performed.

Listing 5: Calling CompanyCode.GetDetail with HAHT Java eConnector

```
1 // The connection has to be opened up before the call
2 CompanyCode aCC = new CompanyCode("1000");
3 aCC.getDetail(aConnection);
```

Listing 6: Calling CompanyCode.GetDetail with IBM Access Builder

```
1 // The connection has to be opened up before the call
2 CompanyCode aCC = new CompanyCode("1000");
3 CompanyCodeGetdetailParams aCCGetdetailParams = new
  CompanyCodeGetdetailParams();
4 aCC.getdetail(aConnection, aCCGetdetailParams);
```

Listing 7: Calling CompanyCode.GetDetail with SAP Java RFC

```
1 // The connection has to be opened up before the call
2 CustomerCreatefromdataParams custCreateParams = new
  CustomerCreatefromdataParams();
3 // Define all parameters for the call using custCreateParams
4 // [...]
5 // Call the BAPI and create a new Customer instance in R/3
6 Customer aCustomer = new Customer(aConnection, custCreateParams);
```


There is no consistent key field handling across the different object models. You can, for example, utilize a constructor to pass the key fields directly as shown above, or construct a key field container, set the appropriate key fields (by the way, key fields are always mandatory!), and instantiate the proxy instance using the key field container. The constructors are overloaded, which means that a Business Object proxy has several constructors with different parameters. They all are generated by the proxy generator tool. (To learn more about the key handling of a specific middleware implementation, refer to the documentation from the vendor.)

> ✔ **Tip**
> *When talking about instance-dependent BAPI calls, make sure to instantiate the proxy first, before calling the BAPI. This creates an instance in Java related to the Business Object instance you want to talk to in R/3. The relationship between Java and R/3 object instances are defined by key fields, which can be specified by using a constructor or a key field container.*

Creating Instances in R/3

Factory BAPIs, otherwise known as instance-creating BAPIs, are what you call to create new data inside R/3. A new customer, for example, must be created in R/3 via the factory BAPI CreateFromDat1.

For the moment, all factory BAPIs are implemented as instance-independent.[10] This means it is not necessary to call the CreateFromDat1 BAPI on a specific instance of a Business Object. You can call it as a static method on the object type Customer, in which case you have to pass in all the data that is required to construct a new instance for the Business Object Customer in R/3. In Java, you utilize the appropriate parameter container — i.e., CustomerCreatefromdat1Params — to do this. The code shown in **Listing 7** (which works only with the IBM Access Builder's object model) creates a new Customer instance in R/3 by utilizing a constructor.

IBM's object model defines a constructor with a valid connection object and a parameter container for the factory BAPI. In fact, you instantiate a new instance in R/3 (yes, you write to R/3 databases through this BAPI!) using the *new* keyword in Java. HAHT's Java eConnector for SAP R/3 maps a factory BAPI in the same way it maps other BAPIs — that is, as Java methods. Customer.CreateFromDat1 will appear as method createFromDat1() on the proxy class representing the Business Object Customer. **Figure 12** shows the differences between Access Builder and Java RFC, and Java eConnector.

[10] In theory, a factory BAPI could be both instance-dependent and instance-independent.

Figure 12 Different Object Models, Different Ways to Create Instances in R/3

Access Builder and Java RFC	Java eConnector
Customer newCustomer = new Customer(aConnection, custCreateParams) Uses a constructor to create a new instance (createfromdat1 will be called under the hood).	Customer newCustomer = Customer.createFromDat1(aConnection) Maps the createfromdat1 BAPI to a Java method.
Pros: Instantiating an object (new Business Object in R/3) via constructor is the proper object-oriented way to create a new instance in R/3.	**Pros:** Factory BAPIs are handled the same way other BAPIs are handled.

After successfully creating a new instance in R/3, one or more key fields are generated by R/3 or taken from the passed parameter container. Sometimes R/3 can internally create unique IDs. After creating a sales order, R/3 will automatically create an order number (SalesOrderId) for the new sales order. This order number can be retrieved using the key field SALESDOCUMENT. The code presented in **Listing 8** creates a new SalesOrder instance (with Access Builder proxies) in R/3 and prints the resulting order number to standard-out (stdout).

But there are also Business Objects, like BusinessPartner, that can take an external number or ID via the parameter container. This number must be unique, otherwise the creation of the new instance inside R/3 will fail. Why should you use your own ID for a new instance? Maybe your organization had specific patterns for customer numbers or sales order IDs before you rolled out R/3 and you want to transfer these patterns to R/3 processes. Or maybe you want to refer to an entity defined in another software system. You can often specify, via the import parameter, whether or not R/3's internal ID creation algorithm should be used.

The documentation of the BAPI tells you whether a BAPI provides support for external IDs or not. All features of BAPIs must be documented properly. Otherwise, you can complain — it's a bug!

You've just seen how to create instances of Business Objects in R/3 and change data in R/3's database tables. In complex situations it is often necessary to call more than one instance-creating BAPI and combine the calls to a transaction. Suppose, for example, you want to change the address and bank account information for an employee who moves from one city to another. Therefore two BAPI calls are used to define the transaction. The first call, "change address information," changes the address information of the employee via EmployeePrivAddress.Create. The second call, EmployeeBankDetail.Create, must only be performed after successfully changing the employee's address.

In the next section, I'll show you how to define transactions with BAPIs and then execute those BAPIs from Java.

Transaction Handling

When I talk about a *transaction* in this section, I use the word to denote a series of operations that follow the "ACID" principle, whereby each transaction can be described as **a**tomic, **c**onsistent, **i**solated, and **d**urable. By this definition, a BAPI call is a transaction. It is:

- *Atomic* because a BAPI call can either be successful or not.

Listing 8: Create a New Sales Order in R/3

```
1 // The connection has to be opened up before the call
2 SalesOrderCreatefromdataParams salesCreateParams = new
  SalesOrderCreatefromdataParams();
3 // Define all parameters for the call using salesCreateParams
4 // [...]
5 // Call the BAPI and create a new SalesOrder instance in R/3
6 SalesOrder aSalesOrder = new SalesOrder(aConnection, salesCreateParams);
7 System.out.println("No. of created SalesOrder: " +
  aSalesOrder.getObjectId().getKeyField ("SALESDOCUMENT");
```


- *Consistent* because the consistency layer of the BAPI checks for inconsistent operations that are not allowed.

- *Isolated* because other BAPIs cannot rely on temporary data in the middle of a transaction.

- *Durable* because a transaction can change data in R/3's database tables.

Any RFM call, in fact, should follow the ACID principle, too.

If a single BAPI call is already a transaction, how can you combine BAPI and RFM calls to form a *logical unit of work* (LUW), which is itself a transaction? You must tell the BAPIs to forget about their atomic nature and be part of the LUW (which has to be atomic, of course). In other words, a BAPI may not be allowed to commit the changes it makes to R/3 data until *all* BAPI calls within the LUW have been executed successfully. Because BAPIs that are only reading data from R/3 do not impact the consistency of R/3 data, we are only interested in *instance-modifying BAPIs*: instance-creating or factory BAPIs and BAPIs that change data in R/3's database tables.

If a single BAPI call is already a transaction, how can you combine BAPI and RFM calls to form a logical unit of work (LUW), which is itself a transaction? You must tell the BAPIs to forget about their atomic nature and be part of the LUW (which has to be atomic, of course). In other words, a BAPI may not be allowed to commit the changes it makes to R/3 data until all BAPI calls within the LUW have been executed successfully.

Release 3.1 does not support this behavior. Each BAPI call within this release is a transaction and always has an atomic nature, which means that you cannot define LUWs, because each BAPI commits the changes it makes itself right after the call. Instance-modifying BAPIs automatically call the ABAP statement COMMIT WORK to commit the changes they've made to R/3 data. Clearly, BAPIs that manipulate R/3 data and call COMMIT WORK automatically to confirm their changes via R/3's update task cannot be combined to a LUW.

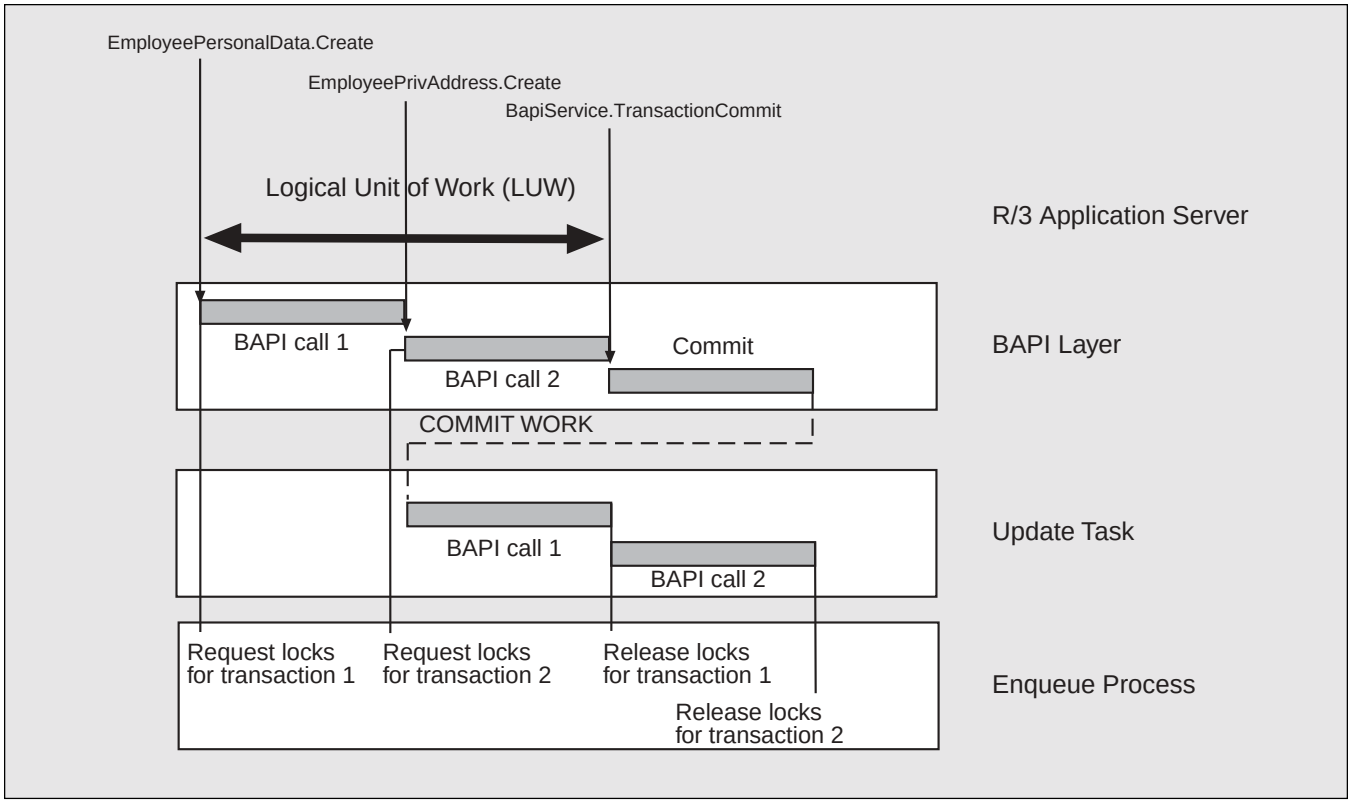
Support for transaction handling (or better, LUW handling) was introduced in R/3 Release 4.0. As of this release, an instance-modifying BAPI can change data and *not* automatically call COMMIT WORK.

You invoke the commit statement manually by calling the RFM BAPI_TRANSACTION_COMMIT if you work with Release 4.0. For Release 4.5, SAP introduced two new BAPIs on the Business Object BapiService: TransactionCommit and TransactionRollback. These BAPIs replace the RFMs BAPI_TRANSACTION_COMMIT and BAPI_TRANSACTION_ROLLBACK used in Release 4.0. If something goes wrong with a BAPI call of the LUW, you are able to roll back to the last consistent state of the data in R/3 and terminate the LUW without committing the changes by calling the RFM BAPI_TRANSACTION_ROLLBACK. Then, of course, you have to take appropriate action in your Java program — i.e., inform the user that something is wrong with his input data, etc.

To call the Release 4.0 COMMIT and ROLLBACK RFMs from Java, you use either Access Builder or Java eConnector to build proxies and invoke the appropriate methods on the generated proxy classes. There's even a way to call these RFMs, which works for all three middleware implementations: you call the RFMs dynamically using the autoCreate statement of the Common RFC Interface.[11] I will introduce this feature in the next section.

[11] Release 4.5 introduced two new BAPIs on the Business Object BapiService to replace the BAPI_TRANSACTION_COMMIT and BAPI_TRANSACTION_ROLLBACK RFMs. They are, respectively, TransactionCommit and TransactionRollback.

Figure 13 Defining a Logical Unit of Work with Two BAPI Calls



Now look at **Figure 13**. The BAPI call EmployeePersonalData.Create and the call EmployeePrivAddress.Create define an LUW. The LUW automatically starts with the first instance-modifying BAPI call (EmployeePersonalData.Create). After successfully calling the BAPI EmployeePrivAddress.Create, both calls will be committed by calling BapiService.TransactionCommit. This call usually does not require any parameters. However, in Release 4.5 you can use a parameter to tell R/3 to wait until the update task has successfully executed the whole LUW. (Thomas G. Schuessler discusses the transactional behavior of BAPIs in the upcoming July/August issue of the *SAP Professional Journal*.) After callingBapiService.TransactionCommit, the next LUW will start automatically with the next instance-modifying BAPI call.

In **Listing 9**, I provide you with a piece of code that calls these two BAPIs to change employee data in R/3:

- EmployeePersonalData.Create
- EmployeePrivAddress.Create

These calls change the personal data and the private address information of an employee. Unfortunately, not all BAPIs support this transaction-handling (or LUW-handling) behavior properly. Some BAPIs can be triggered whether they commit their changes on their own or wait for an external commit call via BapiService.TransactionCommit or BAPI_TRANSACTION_COMMIT. For this purpose, the BAPI offers a flag called *NoCommit* as import parameter. If you set this flag to SAP's definition of *boolean true,* represented by "X," the BAPI will not commit the call automatically via the update task. So every time you want to define an LUW, you *must* set this flag to true if it exists.

Some other BAPIs also provide a *NoCommit* flag to trigger transaction behavior. Sometimes this flag is

Listing 9: Calling BapiService.Commit After Changing Data of Two Infotype Instances

```
01 // The connection object aConnection must be already opened.
02
03 BapiServiceTransactioncommitParams commitParams = new
   BapiServiceTransactioncommitParams();
04 EmployeePersonalDataParams empPDParams = new EmployeePersonalDataParams();
05 EmployeePrivAddressParams empPAParams = new EmployeePrivAddressParams();
06
07 // Set import parameter for both parameter containers
08 // [...]
09 empPDParams.setNocommit("X"); // Force the BAPI to support transactions
10 empPAParams.setNocommit("X"); // Force the BAPI to support transactions
11
12 // Enqueue employee for allowing to make changes by calling enqueue()
13 // on EmployeeAbstract
14 // [...]
15
16 // Create new record for EmployeePersonalData and EmployeePrivAddress
17 EmployeePersonalData newEmp = new EmployeePersonalData(aConnection,
   empPDParams);
18 EmployeePrivAddress newAdr = new EmployeePrivAddress(aConnection,
   empPAParams);
19
20 // Commit both calls.
21 BapiService.transactioncommit(aConnection, commitParams);
```

called differently — i.e., *WithoutCommit* on the Business Object SalesOrder. Only the documentation can tell you exactly how a specific BAPI behaves in a transactional context. And while we're on the subject of context, currently it is not possible to tell R/3 something about the state of a proxy or any other "outside" information.

The code in Listing 9 was written with Access Builder proxies. To successfully change data of HR infotypes, the employee instance must be locked prior to modifying operations. Normally this will be done automatically by the BAPI layer, but in HR you have to call a specific BAPI for this operation: EmployeeAbstract.Enqueue. After modifying data, the employee object (and the lock) must be released by calling EmployeeAbstract.Dequeue.

When you issue a BAPI call that performs some sort of transaction handling (and all version 4.x BAPIs do), you've got to do the following:

- Check whether a flag triggers the BAPI to call COMMIT WORK automatically. Be sure to set the flag when dealing with an LUW.
- Often (in HR, for example) it is required to lock an instance in R/3 by manually calling the Enqueue Server. Make sure to release the lock (EmployeeAbstract.Dequeue) when your code throws an exception! Otherwise, the instance of the Business Object you worked with will remain locked.
- Call BapiService.TransactionCommit after successfully calling all BAPIs of your LUW.
- Make sure to call BapiService.TransactionRollback if an exception occurs in your Java code.

Figure 14 Calling RFMs Utilizing a Command Object

RFM	Access Builder Call	Java eConnector Call
RFC_PING	RFC_PING.execute()	RFMObject.RFC_PING()
BAPI_TRANSACTION_ COMMIT	BAPI_TRANSACTION_ COMMIT.execute()	RFMObject.BAPI_TRANSACTION_ COMMIT()

Otherwise, the LUW and the data it changed have an inconsistent state if you call other BAPIs afterward.

In this section, you were introduced to just the basics of transaction support. There's still a lot of work to do on both the BAPI and Java side. It would be nice if SAP enabled the BapiService Business Object, for example, to provide developers with more information about the transactional context. And on the Java side, I'd like to see developers and ISVs build more sophisticated components for monitoring and manipulating transactions.

Dynamic BAPI Calls Without Proxies

Until now, you have seen how to utilize proxy objects to perform BAPI calls. These proxies were generated at design time by a proxy builder that creates proxy classes for Business Objects, a parameter container for each BAPI, and wrapper classes for a BAPI's parameters, respectively. These proxies describe the structure of the BAPI parameters at design time and are required for effective coding. For example, you are able to use the code help features (i.e., CodeInsight) of your IDE, which gives you all available methods of a proxy or wrapper class in a pop-up dialog.

With IBM's and HAHT's middleware implementations, you can also generate proxies for Remote Function Modules (RFMs).[13] Generating proxies for RFMs enables you to execute linear pieces of ABAP code from Java. These proxies know everything about BAPIs and RFMs and their parameters at design time. Access Builder generates a command object for each RFM. You can run the RFM by calling the execute() method on the appropriate command object. Java eConnector defines RFMs as methods on one or more proxy objects, which can be configured. Here, one proxy can hold multiple RFMs, which can be executed by calling the appropriate method of this object. This method has the same name as the original RFM it represents. **Figure 14** shows the difference between Access Builder and Java eConnector in handling RFMs. Since SAP's proxy generator (SAP Assistant) does not support RFM proxies, SAP Assistant does not appear in Figure 14.

What if we could build a proxy generator that could enable a developer to "customize" the proxies just before code generation? Say, for example, that you are only interested in SalesOrder.GetStatus. Without customization, the proxy builder would generate code that represents the whole object, not only the getstatus method. The result consumes a huge amount of code overhead. You must carry around all the useless parameter containers and wrapper classes. And, even worse, you have to package this useless information with your application.

With a customized proxy, however, developers could choose, at design time, just the BAPIs they are interested in and the generator would only generate the required code. This saves lots of proxy code! This way, you could specify one particular BAPI call without generating the proxy class first. You could call the underlying RFM, which implements the BAPI. Now, the name and the parameters for the RFM are known at design time, but what about all the parameters the RFM may consume? You could look

[13] The SAP middleware solution does not support this.

up all parameters using SAP transaction SE37 and construct them manually in Java. But this is a tedious job. You have to define each field of a structure, table, and so forth with data type, length, and number of decimals (if applicable). Alternatively you can use the autoCreate feature to initialize RFMs, structures, and tables with all fields required.

The autoCreate methods are defined in the Common RFC Interface for Java for the following factory methods:

- RfcModuleFactory
- StructureFactory
- TableFactory

These methods connect to R/3 and retrieve metadata information at runtime about RFMs, their structures, and tables. When RFMs, structures, and tables are changed in a new R/3 release (remember, only BAPIs should provide a stable interface!), autoCreate can make your programming life much easier because you obtain metadata at runtime. This means it does not matter which version of a structure or table you use in your code, since the metadata is retrieved at runtime.[14] You can be sure to instantiate the most current version of an RFM (compared to an RFM proxy that can be subject to change between R/3 releases), structure, or table. However, dynamic coding isn't all that intuitive, and using proxies at design time makes your code faster when compared to using dynamic calls, because they must retrieve metadata for parameters "just-in-time."

How exactly does autoCreate work? It calls the following two RFMs under the hood to retrieve metadata about the RFM and its scalars, structures, and tables from the Business Object Repository and the Data Dictionary:

- RFC_GET_FUNCTION_INTERFACE
- RFC_GET_STRUCTURE_DEFINITION

The first call retrieves detailed information about the parameter names of an RFM. The second call is able to get all required information on field names and field data. If you are interested in building your own middleware solution, I recommend you take a close look at these two RFMs and their parameters. To inspect these two RFMs, you can use SAP transaction SE37.

An instance of an RFM can be called by using RfcModuleFactory.autoCreateRfcModule(IRfcConnection, String). This method needs two parameters: a valid connection object and the name of the RFM that should be called. After "autoCreating" an instance of the RFM and defining all required import parameters, the RFM can be called with the method callReceive(). You can directly operate with scalars, structures, and tables on the RFM instance you created. You must, of course, be aware of the parameter names you want to deal with. **Listing 10** illustrates how you can manipulate parameters — scalar (line 17); structure (line 15); and table (line 27).

[14] This assumes that the field names of the fields you access remain unchanged.

Listing 10: Call BAPI_FIXEDASSET_GETLIST Using autoCreateRfcModule

```
01 /**
02 * Call BAPI_FIXEDASSET_GETLIST which represents the BAPI call
03 * FixedAsset.getlist() - retrieveFAList
04 * This method calls the RFM without proxy classes and displays a list of
     FixedAsset
05 * instances on System.out
06 * @param aConnection com.sap.rfc.IRfcConnection
```

(continued on next page)

(continued from previous page)

```
07 * @param aCC java.lang.String - representing the CompanyCodeId
08 */
09 public void retrieveFAList(IRfcConnection aConnection, String aCC) {
10
11    try {
12          IRfcModule getlistRFM =
              aFactoryManager.getRfcModuleFactory().autoCreateRfcModule
              (aConnection,"BAPI_FIXEDASSET_GETLIST");
13
14          // Set import parameter
15          IStructure aReqTabs =
              getlistRFM.getStructImportParam("REQUESTEDTABLESX");
16
17          aReqTabs.getSimpleField("GENERALDATA").setString("X");
18          getlistRFM.getSimpleImportParam("COMPANYCODE").setString(aCC);
19
20          getlistRFM.getSimpleImportParam("DEPRECIATIONAREA").
              setBigInteger(new java.math.BigInteger("01"));
21
22          getlistRFM.getSimpleImportParam("MAXENTRIES").setInt(100);
23
24          getlistRFM.callReceive(); // call the RFM
25
26          // Extract table param and show result
27          ITable aTable = getlistRFM.getTableParam("GENERALDATA");
28          IRow aTableRow = null;
29
30          for (int i = 0; i < aTable.getRowCount(); i++) {
31              aTableRow = aTable.getRow(i);
32              System.out.println("\t" + aTableRow.getSimpleField("ASSET") +
                  "\t" + aTableRow.getSimpleField("DESCRIPT")); // console
                  output
33          }
34    }
35    catch (Exception e) {
36          // Add your exception handling here!
37    }
38 }
```

When dealing with Business Object proxies and BAPIs, it is more complicated to manipulate parameters directly, because all parameters are encapsulated by a parameter container for each BAPI. You can do it, but it's more work and your code may become more difficult to read.

After performing the call via callReceive(), results (export and table parameters) can be retrieved via get methods.

Let's say, for example, you want to call an RFM with export and table parameters that change from one R/3 release to another. To make sure that your code works with both releases, it is useful to create structure and table objects for parameters

dynamically without using generated proxies (parameter wrapper).

There are two autoCreate methods available for this purpose:

- StructureFactory.autoCreateStructure(String, IRfcConnection, String)
- TableFactory.autoCreateTable(String, IRfcConnection, String)

These two calls return fully initialized structure (IStructure) and table (ITable) objects, but these objects are still empty after creation. The first parameter is a string specifying the parameter name of the structure or table, the second parameter must be a valid connection object, and the third parameter again is a string defining the name of the underlying data element.

Scalars represent single values in BAPI import and export parameters, and are defined as SimpleFields in the Common RFC Interface. They can't be initialized via autoCreate. If scalar import or export parameters are required, you need to define the related metadata object (SimpleInfo) manually.

Often, it is not necessary (or you simply are not able) to generate proxies for Business Objects or RFMs at development time. Then the autoCreate features of the Common RFC Interface offer you help to define RFMs, structure, and table objects at runtime by retrieving all relevant metadata information from R/3's BOR.

Working with Metadata

When we are talking about parameters, we are using metadata provided by the proxy objects or (dynamically) by autoCreate methods, as you saw in the last section. This metadata describes scalars, structures, and tables.

Generic Mapping of ABAP Data to Java: A Quick Overview

The Common RFC Interface for Java knows about three different basic data types:

- Scalar (just single fields or variables)
- Structure (collection of scalars)
- Table (collection of structures)

These three basic data types are mapped to their Java relatives. For scalars, the mapping process is quite simple. They are mapped to Java variables of a specific data type depending on their ABAP data type — i.e., an ABAP character will be mapped to java.lang.String, etc.

For structures and tables, the Common RFC Interface provides two interfaces — IStructure and ITable — which must be implemented by the appropriate classes of the middleware implementation. The object representing a structure (which implements IStructure) is described by a specific metadata object (ComplexInfo) — i.e., number of fields, field name for each field, data type for each field, etc. The same applies to tables, but table objects are implementing the ITable interface.

Listing 10 shows sample code that retrieves a list of FixedAsset instances with dynamic calls only. For a description of import, export, and table parameters for FixedAsset, where I also described a business scenario using FixedAsset, please refer to my last article, "BAPI Basics When Programming with Java," which appeared in the Premiere Issue of the *SAP Professional Journal*.

The Common RFC Interface provides only basic functionality in the metadata area. The SimpleInfo object basically describes a scalar and offers:

- Parameter name
- Data type
- Length in bytes
- Number of decimals (if applicable)

The Common RFC Interface does not provide the ability to get extended metadata. For example, while you can get the length of a field, you cannot get an indicator of whether or not a field supports only lowercase letters. In real-world situations, extended metadata is very important. You may want to check a user's input prior to calling a BAPI to avoid errors. In this case, you need to know about the data type, length, and if check tables or fixed value lists are available. If you deal with units of measure in your BAPIs, you must be aware of conversions between your Java program and the BAPI parameter. This is because units of measure are language-dependent but stored in the R/3 database in the German format. In English-speaking countries, for example, the abbreviation for "piece" is PC, in French-speaking countries it is UN for "unité," and in German it is ST for "Stück." Your program has to convert from the internal to the external format and vice versa. But before you can perform any conversion operation, you have to know if the field you are dealing with needs to be converted. The SimpleInfo object does not provide you with this information.

For situations where extended metadata is necessary, I recommend you either consult Thomas G. Schuessler's article, "Simplifying BAPI Programming with Components,"[15] to learn how you can get extended metadata regarding R/3 structures and tables through the use of a component, or consider using HAHT's Java eConnector object model for proxies. This object model takes care of extended metadata. It defines a *SimpleMetadata* object that provides a developer with additional metadata for a *Simple* object. **Figure 15** shows you all attributes dealing with extended metadata for the key field CompanyCodeId of the Business Object CompanyCode.

The attribute *helpValuesSupported*, for instance, can be accessed via the method isHelpValuesSupported(), which tells you whether additional information via Helpvalues.GetList can be retrieved for this field or not. Another method returns the name of the check table — however, currently it isn't possible to retrieve the check table values (as name/value pairs, for example).

The lack of extended metadata in the Common RFC Interface gives you a clue as to just how complex a real-world integration project can be. In the next section, I'll raise some more issues to consider at the starting point of an integration project.

If you deal with units of measure in your BAPIs, you must be aware of conversions between your Java program and the BAPI parameter. This is because units of measure are language-dependent but stored in the R/3 database in the German format. Your program has to convert from the internal to the external format and vice versa.

Real-World Considerations for Real-World R/3 Integration Projects

In the last couple of years, many large companies have come to us with plans to launch mission-critical R/3 integration projects that focus on bringing not only R/3 data, but R/3 processes to the Web and combining them with external systems. This task wasn't at all easy — we built e-business solutions

[15] *SAP Professional Journal*, November/December 1999.

Figure 15 All Attributes Dealing with Extended Metadata for the CompanyCodeId Key Field

Data Type	Field Name	Description	Example for CompanyCodeId
Boolean	MixedCaseSupported	This flag indicates whether or not names in mixed case are supported	True
Boolean	HelpValuesSupported	Support for help values via Helpvalues.GetList	True
java.lang.String	ReferenceTable	Not of interest to BAPI developers	—
java.lang.String	ReferenceField	Not of interest to BAPI developers	—
Int	Length	The length of the field; also available as property of the SimpleInfo object	4
Int	Decimals	Number of decimals; also available as property of the SimpleInfo object	0
Int	InternalLength	The internal length can be different from the Length field; not of interest to BAPI developers	4
Boolean	HasValueList	Fixed value list exists for this field	False
java.lang.String	DataTypeDD	Data type in Data Dictionary	C
java.lang.String	DataTypeABAP	ABAP Data type	CHAR
java.lang.String	ConversionExit	Points to a conversion exit that tells you whether or not a field must be converted prior to call a BAPI	—
java.lang.String	CheckTable	Name of a check table	—
java.lang.String	FieldTable	Name of the data element	BUKRS
java.lang.String	StructureName	Name of the structure containing this field	T001

and discovered that our clients' sales processes were not optimized for R/3-related Internet sales. Monumental programming efforts in the middle-tier were required. Fortunately, the Common RFC Interface supports building complex logic that uses R/3 functionality.

Listed below are considerations to keep in mind for your own R/3 integration projects. These are tips that I've gained through real-world experiences like the example above. My hope is that this listing will help you to avoid some of the common pitfalls many companies fall into. And, don't forget, we are discussing Java technologies here, which means that you are able to leverage the full benefit of object-oriented technology!

Figure 16 Skill Matrix for Assigning Project Resources

Skill Level	Description	Skills
1	Basics	Internet technologies, Web design, operating systems, databases, security
2	Application development	Java or C++ or Visual Basic and/or ABAP, SAP Business Framework and BAPI technology, Common RFC Interface for Java, and different object models of the different middleware vendors; this knowledge can be obtained by attending SAP's CA926 "Programming with BAPIs in Java" training class
3	Application server	For example: HAHTsite, Websphere
4	Design, modeling, and architecture	OOA/OOD, RMI, EJB, CORBA, DCOM

✔ If you are going to build an alternative Java-based frontend to R/3, be prepared to:

- Analyze your processes and determine which BAPIs fit your needs — if you cannot find any, search for RFMs, or, worst case, write your own RFMs or BAPIs.
- Wrestle with multilingual field descriptions
- Wrestle with conversion problems
- Perform multiple BAPI and RFM calls in a transactional context
- Visualize complex and hierarchical data

If you are going to build an alternative Java-based frontend to R/3, be prepared to, analyze your processes and determine which BAPIs fit your needs wrestle with multilingual field descriptions, wrestle with conversion problems, perform multiple BAPI and RFM calls in a transactional context, and visualize complex and hierarchical data.

✔ If you are going to build a cross-platform Java integration layer that will connect existing (legacy) systems to your R/3 system, you will need to have answers to the following questions:

- Does platform independence play a role?
- How easy is it to modify processes inside R/3, and do you have the appropriate resources to do it?
- Does it make sense to transport or connect processes inside and outside R/3?
- Is it required to maintain an open standard for connecting other systems in the future?
- Is support for a transactional context necessary?
- What skills are available in your company (see the skill matrix in **Figure 16**)?

As you can see, some questions are relevant for both alternative frontend projects and legacy integration projects.

✔ When starting an integration project, the integration scenario must be described with all its business processes. This description is the basis for

a decision matrix summarizing the pros and cons of the "outside-in" Java approach. In this article, I've assumed the decision was already made in favor of Java technology.

✔ Then start your architecture discussions. Determine all important project parameters and how best to use a distributed environment, across which you will need:

- Applets
- Servlets
- Java Server Pages (JSPs)
- Remote Method Invocation (RMI)
- Enterprise JavaBeans (EJBs)
- CORBA

✔ When talking about Enterprise Java applications (servlets and EJBs), you also have to consider an application server. It ensures that your application will scale and that you have a transactional environment.

✔ We discussed lots of technology issues that are important to successfully complete an integration project. The skill matrix in Figure 16 can help you to assign resources to the different phases of a project:

- Start
- Design
- Implementation
- Production

Of course, it would be great if these skills could be cumulated in one person, but this would mean an application architect with skill level 4 should also have in-depth application server and application development know-how, and these people belong to a rare species of IT professionals.

When starting an integration project, the integration scenario must be described with all its business processes. This description is the basis for a decision matrix summarizing the pros and cons of the "outside-in" Java approach. In this article, I've assumed the decision was already made in favor of Java technology.

✔ The following steps have also proved very useful in real-world scenarios:

- Describe the integration scenario in detail. You should be able to make a meta-description of the scenario that is independent of the underlying technology.
- Use industry standards as much as possible to build an open, flexible system.
- Determine the skill requirements and allocate resources.
- Complex tasks should be encapsulated into components to hide complex logic and make sure they can be maintained in a central place. Components also enable you to reuse already implemented functionality.
- Strictly separate visual logic from application logic and make your visual components as thin as possible.
- In performance-critical situations, retrieve static R/3 data (i.e., currency, countries, company codes, etc.) just once and make it persistent on the application server.

✔ A layered architecture makes it relatively easy to build and maintain complex components: let's call them "eScenario" components (or components to power your e-business applications), which are highly

specialized and process-dependent. They can encapsulate Business Object proxies. These eScenario components can be extended by generic helper and/or communication components like an RFC server component for Outbound RFC calls, CORBA support, or conversion/translator components.

✔ Server-side Java implementation projects use three-tier architectures. Your R/3 connector architecture should be totally independent of the underlying application architecture.

Conclusion

The Common RFC Interface for Java is the basis to leverage SAP and Java technologies. But the "write once, run anywhere" paradigm of Java, which allows you to develop, reuse, and deploy your application logic for client-side as well as server-side enterprise applications running on an application server, will only take you so far. To be successful with an integration project, the process modeling and architecture must be defined properly. This is the key to building e-business applications for this new millennium.

Michael Czerniak joined SerCon, an IBM Global Services company, in February 1997 as an OO specialist and developer for C++ and Java. Before SerCon, he owned a small software engineering company specializing in OO development, databases, groupware, and Internet technology. After starting with SerCon as a member of a development team working on the SAP GUI, focusing in detail on the SAP-JavaGUI project, he and his team began the Access Builder project at IBM. He was then relocated to SAP's basis development site to lead the AB team there.

These days Michael is a Senior Consultant/ Software Architect at SerCon, IBM Global Services, where he leads a group of highly specialized IT consultants whose expertise is in the SAP Business Framework and Middleware — the first to provide consulting services around the SAP Business Framework and Java worldwide. He is also teaching a "Programming with BAPIs in Java" training class, and is involved in future development regarding BAPI technology in Java. He has collaborated with Thomas G. Schuessler on various projects regarding BAPI middleware implementations concerning the Common RFC Interface for Java. Additionally, he is a regular speaker at SAP TechEd conferences in the area of BAPI Programming with Java. Michael can be reached at michael.czerniak@sercon.de.

