BAPI Return Messages Made Easy

Thomas G. Schuessler

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

(complete bio appears on page 92)

the herrors I could have expected, all, let them all come, they are my villains
James Joyce: Finnegans Wake, p. 545.12

As a developer, you do not need to be told that proper error handling is one of the keys to a stable application with low maintenance cost. If you develop BAPI-enabled components and applications this means that you need to know how to deal with BAPI Return messages. And as a developer of BAPIs, you should know the rules regarding BAPI Return messages that were defined by SAP. Otherwise, you may end up confusing the client programmers who use your BAPIs and this in turn may lead to incorrect error handling in their applications.

So the target audience for this article includes anybody who develops or uses BAPIs.

The article will describe in detail how BAPIs inform us about success and failure by using a standardized[1] `Return` parameter. We will discuss the intricacies of interpreting the data in this parameter correctly in order to avoid pitfalls.

As you will see, handling BAPI messages correctly is a little more complicated than you might expect[2]. Hence I decided to build a Java class that makes dealing with BAPI messages much easier for developers who use Java, and therefore the SAP Java Connector (JCo).

[1] As you will see later, the term is used here *cum grano salis*.

[2] And, if you ask me, than it should be.

Figure 1 List of Relevant BAPIs in 4.6C Without a Return Parameter in the BOR

Object Name	BAPI Name	RFM Name	RFM Has RETURN?
Appraisal	Change	BAPI_APPRAISAL_CHANGE	true
Attendee	GetTypeList	BAPI_ATTENDEE_TYPE_LIST	true
Batch	GetLevel	BAPI_BATCH_GET_LEVEL	false
BBPIncomingInvoice	BbpInvoiceCreate	BBP_INVOICE_CREATE	true
Location	GetListAll	BAPI_BUS_EVENT_LOCATION	true
ProdOrdConfirmation	ExistenceCheck	BAPI_PRODORDCONF_EXIST_CHK	true

At the end of this article, I will briefly introduce this class so that you can build your own solution, in Java or even other programming languages.

The Return Parameter

How do we find out whether a BAPI call accomplished what we intended? BAPIs are supposed[3] to have a `Return` parameter that tells the client program whether the BAPI call succeeded or what went wrong.

Most BAPIs obey this rule, but let us quickly talk about those that don't. **Figure 1** contains a list of those BAPIs in 4.6C that — according to the Business Object Repository (BOR) — do not have a `Return` parameter. In most cases[4], the underlying RFC-enabled Function Module (RFM) actually has a `Return` parameter, but somehow it is missing from the BOR. When you research a BAPI in the SAP BAPI Explorer (transaction code `BAPI`), you should always look at the underlying RFM in the SAP Function Builder (transaction code `SE37`) to double-check that the information in the BOR is really correct. In cases where the two tools disagree, the metadata in the Function Builder takes precedence over what the BOR says!

The header text for Figure 1 states that the relevant BAPIs are listed. How is "relevant" defined here? I have only included BAPIs that — in release 4.6C — fulfill the following conditions[5]:

- The BAPI is released.
- The BAPI is not obsolete.
- The BAPI does not pop up SAPGUI dialogs.[6]

If you want to create your own list, for a different release or with different conditions, take a look at the code in **Listing 1**, which I used to produce the data for Figure 1.

Exceptions Are the Exception

So how do BAPIs that have no `Return` parameter inform us about problems? They usually throw

[3] Cf. the *SAP BAPI Programming Guide*, which is part of the standard SAP documentation. Be careful with your assumptions, though. Not all BAPIs (as we will also see in this article) follow all the rules.

[4] Cf. the column called "RFM Has RETURN?".

[5] These conditions exclude BAPIs that the majority of all customers will never use — and for good reasons, too.

[6] BAPIs that have SAPGUI dialogs are mainly for use by SAP and therefore not really relevant for our discussion.

Listing 1: Source Code to Generate Figure 1

```
private void getBapisWithoutReturn() {
  try {
    BOTypes bos = mRepository.getObjectFactory().getBOTypes();
    for (int i = 0; i < bos.getSize(); i++) {
      BOType bo = bos.getBOType(i);
      BOMethods bapis = bo.getBOMethods();
      for (int j = 0; j < bapis.getSize(); j++) {
        BOMethod bapi = bapis.getBOMethod(j);
        if ( bapi.isReleased() &&
             ! bapi.getBOParameters().exists("Return") &&
             ! bapi.isObsolete() &&
             ! bapi.isUsingDialog() ) {
          boolean rfmReturn =
            bapi.getRfm().getRfmParameters().exists("RETURN");
          System.out.println(bo.getObjectName() + '\t' +
                             bapi.getName() + '\t' +
                             bapi.getRfmName() + '\t' +
                             rfmReturn
          );
        }
      }
    }
  }
  catch (Exception ex) { ex.printStackTrace(); }
}
```

Figure 2 List of Relevant BAPIs in 4.6C with Exceptions

Object Name	BAPI Name	RFM Name	RFM Has RETURN?
Attendee	ChangePassword	BAPI_ATTENDEE_CHANGEPASSWORD	true
Attendee	CheckExistence	BAPI_ATTENDEE_CHECKEXISTENCE	true
Attendee	CheckPassword	BAPI_ATTENDEE_CHECKPASSWORD	true
Attendee	GetBookList	BAPI_ATTENDEE_BOOK_LIST	true
Attendee	GetCompanyBookList	BAPI_COMPANY_BOOK_LIST	true
Attendee	GetCompanyPrebookList	BAPI_COMPANY_PREBOOK_LIST	true
Attendee	GetPrebookList	BAPI_ATTENDEE_PREBOOK_LIST	true
Attendee	GetTypeList	BAPI_ATTENDEE_TYPE_LIST	true
BusinessEvent	GetInfo	BAPI_BUS_EVENT_INFO	true
BusinessEvent	GetLanguage	BAPI_BUS_EVENT_LANGUAGE	true
BusinessEvent	GetSchedule	BAPI_BUS_EVENT_SCHEDULE	true
BusinessEvent	Init	BAPI_BUS_EVENT_INIT	true
BusinessEventGroup	GetEventtypeList	BAPI_BUS_EVENTTYPE_LIST	true
BusinessEventGroup	GetList	BAPI_BUS_EVENTGROUP_LIST	true
BusinessEventtype	GetEventList	BAPI_BUS_EVENT_LIST	true
BusinessEventtype	GetInfo	BAPI_BUS_EVENTTYPE_INFO	true
FinancialTransaction	CreateFromData	BAPI_FTR_CREATEFROMDATA	true
Location	GetListAll	BAPI_BUS_EVENT_LOCATION	true
PurchaseOrder	Release	BAPI_PO_RELEASE	true
PurchaseOrder	ResetRelease	BAPI_PO_RESET_RELEASE	true
PurchaseReqItem	Release	BAPI_REQUISITION_RELEASE	true
PurchaseReqItem	ResetRelease	BAPI_REQUISITION_RESET_RELEASE	true

ABAP exceptions. This is also against the rules, but rules are just rules, not guarantees. You need to know which BAPIs throw ABAP exceptions, since your error handling has to be coded differently.[7] **Figure 2** lists the relevant[8] BAPIs with ABAP exceptions in 4.6C and **Listing 2** shows the source code that generated the data for Figure 2.

We would have expected that *Batch.GetLevel* from Figure 1 (the one with absolutely no `Return` parameter) would show up in Figure 2, but it does not. This BAPI actually has neither a `Return` parameter nor ABAP exceptions, which probably implies that it cannot fail under any condition. Please join me in congratulating the developer of this BAPI for his boundless optimism!

How do we deal with BAPIs that have ABAP exceptions and a `Return` parameter? Some of them had just ABAP exceptions in previous releases and a `Return` parameter was added later. Unless the documentation of the BAPI clearly states that the ABAP exceptions are not used anymore, your code should assume that an ABAP exception could still be thrown, which means that you need to evaluate the `Return` parameter and also check for ABAP exceptions.

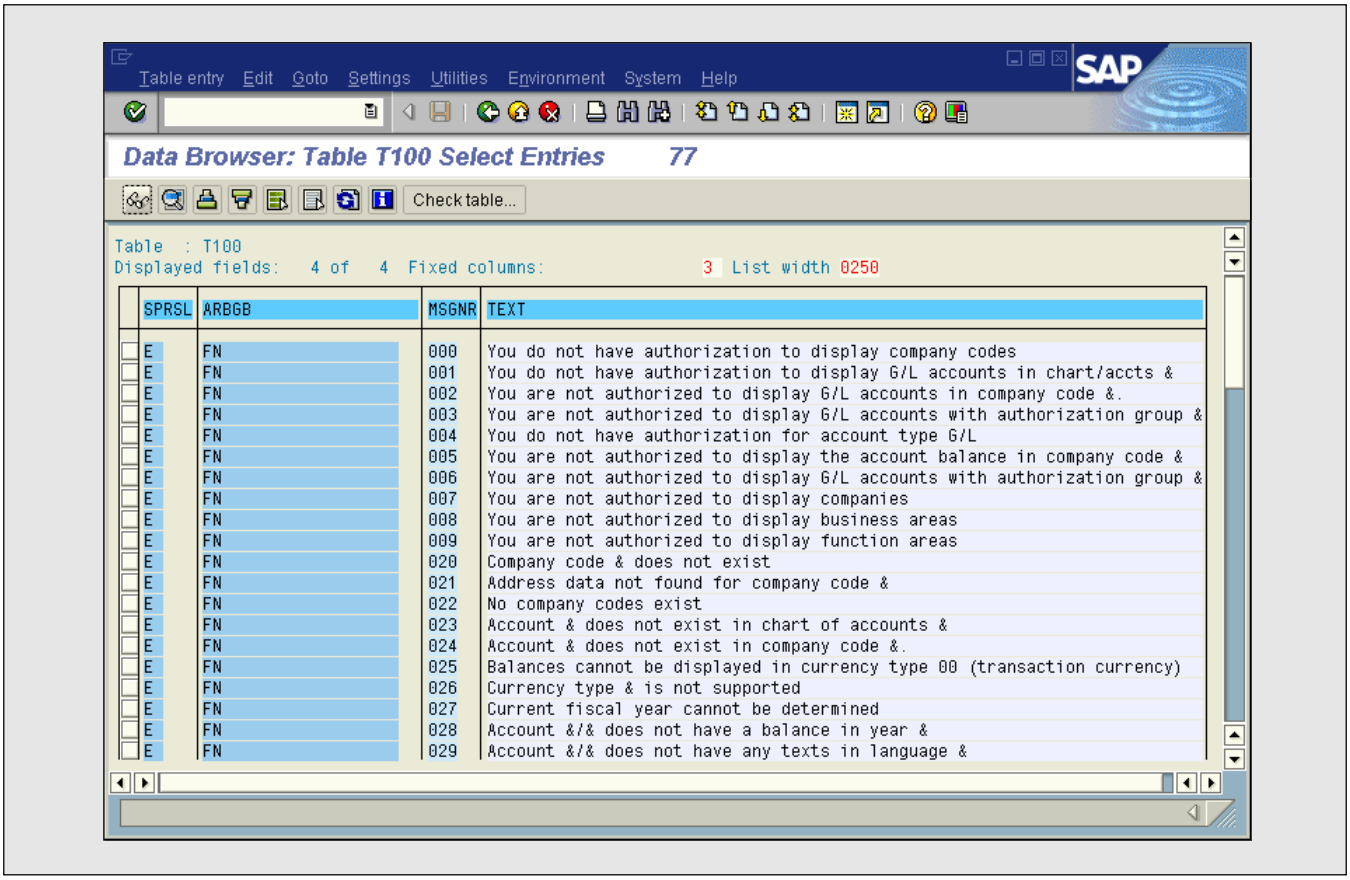
[7] The details depend on the middleware you use.

[8] Using the same criteria as for Figure 1.

Listing 2: Source Code to Generate Figure 2

```
private void getBapisWithExceptions() {
  try {
    BOTypes bos = mRepository.getObjectFactory().getBOTypes();
    for (int i = 0; i < bos.getSize(); i++) {
      BOType bo = bos.getBOType(i);
      BOMethods bapis = bo.getBOMethods();
      for (int j = 0; j < bapis.getSize(); j++) {
        BOMethod bapi = bapis.getBOMethod(j);
        if ( bapi.isReleased() &&
             ! bapi.isObsolete() &&
             ! bapi.isUsingDialog() &&
             bapi.getRfm().getRfmExceptions().getSize() > 0 ) {
          boolean rfmReturn =
            bapi.getRfm().getRfmParameters().exists("RETURN");
          System.out.println(bo.getObjectName() + '\t' +
                             bapi.getName() + '\t' +
                             bapi.getRfmName() + '\t' +
                             rfmReturn
          );
        }
      }
    }
  }
  catch (Exception ex) { ex.printStackTrace(); }
}
```

Figure 3 *Some Entries from Table T100*



SPRSL	ARBGB	MSGNR	TEXT
E	FN	000	You do not have authorization to display company codes
E	FN	001	You do not have authorization to display G/L accounts in chart/accts &
E	FN	002	You are not authorized to display G/L accounts in company code &.
E	FN	003	You are not authorized to display G/L accounts with authorization group &
E	FN	004	You do not have authorization for account type G/L
E	FN	005	You are not authorized to display the account balance in company code &
E	FN	006	You are not authorized to display G/L accounts with authorization group &
E	FN	007	You are not authorized to display companies
E	FN	008	You are not authorized to display business areas
E	FN	009	You are not authorized to display function areas
E	FN	020	Company code & does not exist
E	FN	021	Address data not found for company code &
E	FN	022	No company codes exist
E	FN	023	Account & does not exist in chart of accounts &
E	FN	024	Account & does not exist in company code &.
E	FN	025	Balances cannot be displayed in currency type 00 (transaction currency)
E	FN	026	Currency type & is not supported
E	FN	027	Current fiscal year cannot be determined
E	FN	028	Account &/& does not have a balance in year &
E	FN	029	Account &/& does not have any texts in language &

Since not all customers are on 4.6C yet, let me give you a breakdown of the different situations you might encounter in the real world:

- BAPIs with a `Return` parameter and no ABAP exceptions. These are the rule-abiding majority of all BAPIs.
- BAPIs with no `Return` parameter, but ABAP exceptions.
- BAPIs with a `Return` parameter and ABAP exceptions. Some of them may no longer throw the exceptions. (Again, check the documentation[9] for a statement to that effect, otherwise assume that an ABAP exception might still be thrown.)
- BAPIs with no `Return` parameter and no ABAP exceptions. They are (hopefully) infallible.

[9] Or study the ABAP source code if you are into masochism.

Checking whether an ABAP exception occurred is easy for a client application. Correctly interpreting the `Return` parameter proves to be somewhat more intricate. Time to study this parameter in more detail.

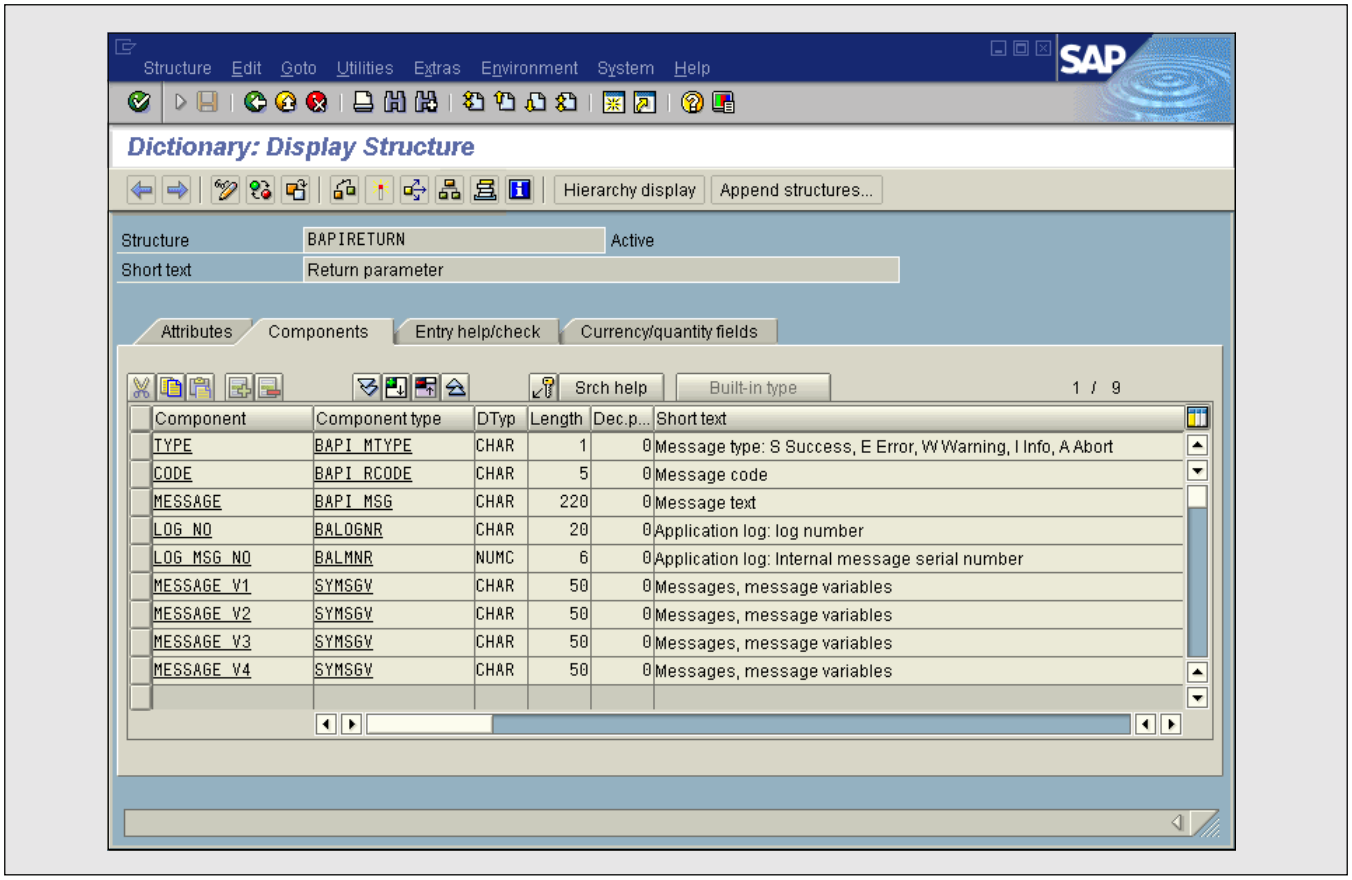
Return Structures

The messages that a BAPI returns in its `Return` parameter are similar to the ones used in the online transactions (SAPGUI), they are even stored in the same table (T100). **Figure 3** is a screenshot that shows you some messages from this table.

Let us look at the four columns:

- SPRSL: Messages are maintained in multiple languages. This column contains the language code.

Figure 4 *The BAPIRETURN Structure*



- ARBGB: Each message is assigned to an application area (nowadays usually called a message class).
- MSGNR: The message number.
- TEXT: The message text. It may include up to four ampersands, which will be replaced with relevant information when the actual message is created in the application. Newer messages use numbered ampersands ("&1" and so on) so that the translators can change their relative positions, which is a good idea in languages with a different word order.

The `Return` parameter of a BAPI will contain the message class, the message number, and the message text, with the variable portions (the ampersands) filled in. But SAP uses four different Data Dictionary structures to implement the `Return` parameter. BAPIRET2 is the standard one since 4.6, but the others are used as well, especially in older BAPIs. **Figures 4-6** show the fields used in the various structures. BAPIRETURN1 and BAPIRET1 actually contain the same field list, so I included only one screenshot.[10]

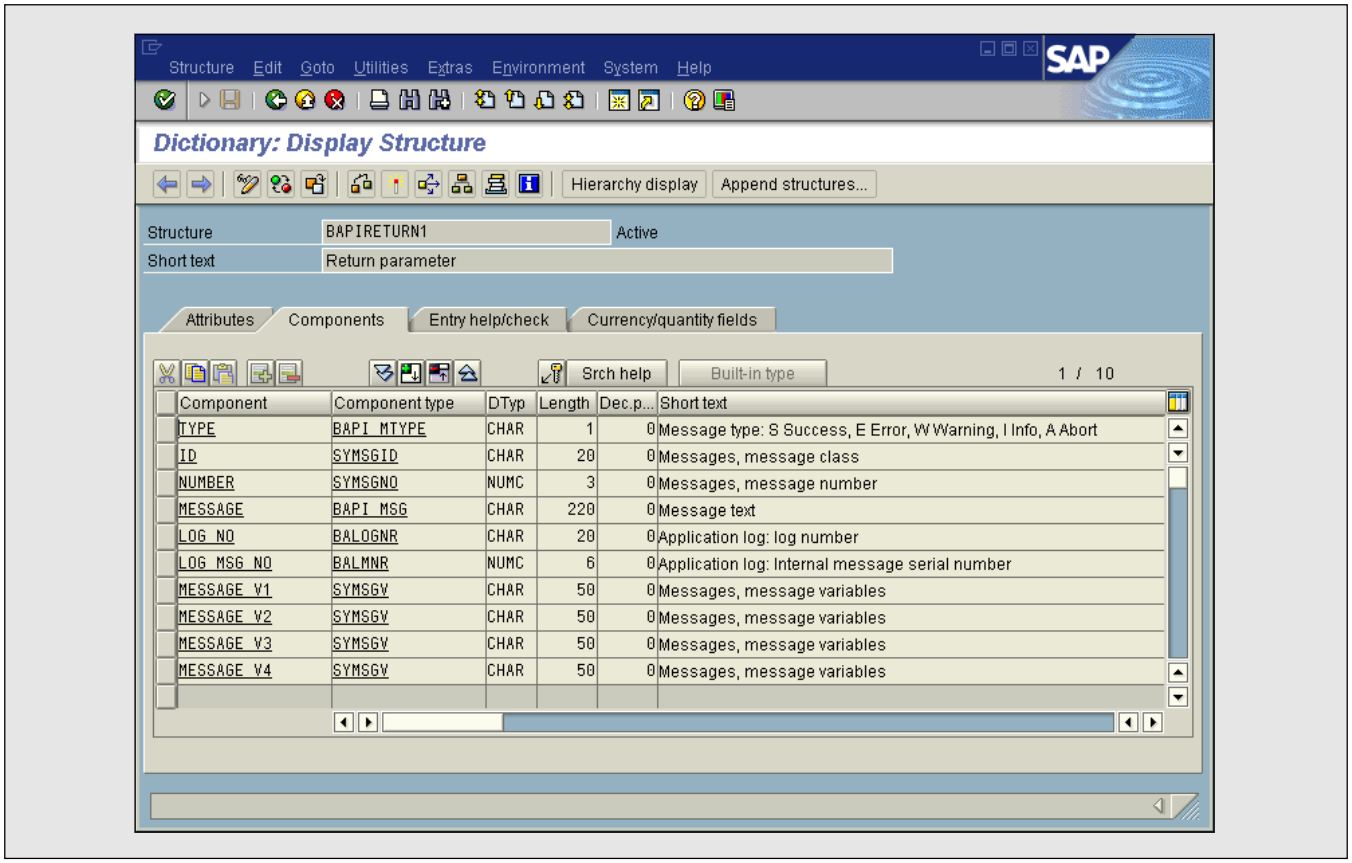
Message Types

Let us discuss the TYPE field first. This field tells you about the severity of the message. Here is a list of the possible contents:

- "S" or space (" ") indicate success. The BAPI call worked fine.
- "A" says that there was a severe problem in SAP. This usually means that the problem is not caused

[10] And no, I will not tell you why there are two structures with exactly the same fields. Some secrets must remain secrets. (Hence the word.)

Figure 5 The BAPIRETURN1 and BAPIRET1 Structures



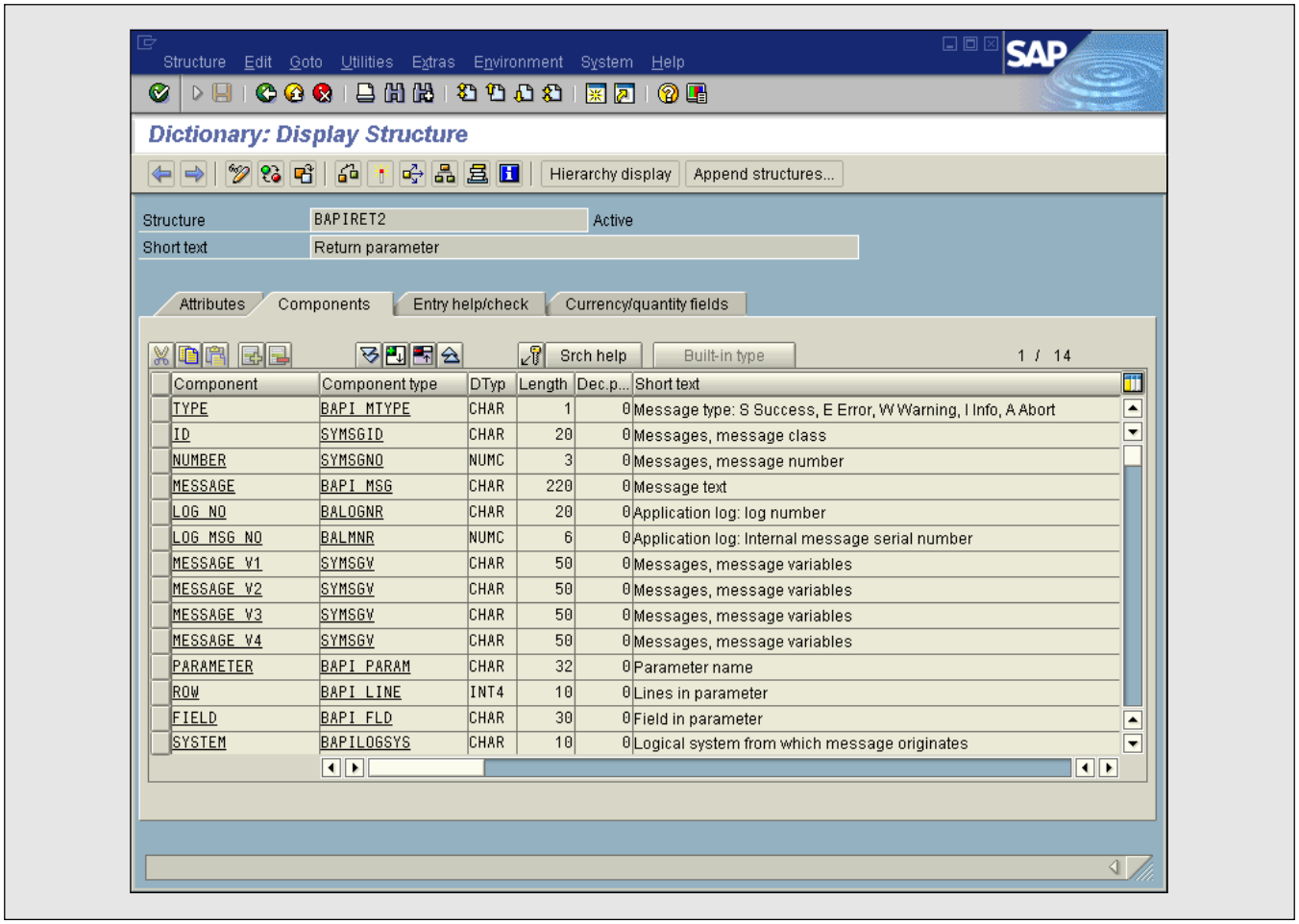
by your parameters but by a misconfiguration[11] of SAP, a full table space in the database, or similar reasons. Someone will have to look at the SAP System Log and rectify the situation.

- "E" is a normal error. Probably something is wrong with the parameters you passed to the BAPI.
- "W" indicates that the BAPI worked at least partially, but something was not perfect. An example of a warning message coming back from a BAPI is FN021, issued by *CompanyCode.GetDetail* when the address data for a company code has not been maintained.
- "I" denotes an information message. Our assumption would be that the BAPI worked fine, but has some interesting extra information it would like us to know. See below about the validity of this assumption.

How should your application react to the different error types? "S", " " (space), "A", and "E" do not present much of a challenge. The call succeeded or failed, so the application should have no difficulty reacting accordingly. "W" and "I" are trickier. A warning can sometimes be ignored. In the aforementioned case of FN021 when calling *CompanyCode.GetDetail*, for example, we may not need the address and therefore can safely ignore the warning. But that presupposes that we know the specific warning and its implications when developing our application. Therefore, the *SAP BAPI Programming Guide* has a rule stating that all Return messages that can potentially be issued by a BAPI must be listed in the BAPI's documentation. If the BAPI follows this rule (and of course not all BAPIs

[11] This word does not exist according to Bartleby and my spell checker. Since www.google.com finds about 55,000 occurrences, I think it is about time they update their dictionaries.

Figure 6 *The BAPIRET2 Structure*



do!), the developer of a client application can check all warnings while developing the application and decide what to do in each case.

What about the BAPIs not obeying the rule?[12] Extensive testing of your application will uncover some of the warnings, but most likely not all of them. If an application encounters an unforeseen warning message, it can react in different ways, but if you were to ask me, I would suggest the following response:

- If you have a user whom you trust to be able to react intelligently to a warning message, then you should present that user with the warning returned by SAP and ask the user to decide whether this warning should be ignored or treated as an error. In addition to the message text (field MESSAGE) you could also display the documentation for the message (see below).
- Otherwise, it is safest to assume that the BAPI call was not successful.

Information messages seem to be just a special case of success messages, but unfortunately that is not always true. I have seen at least one BAPI, the name of which unfortunately eludes me, that issued an information message in a situation that I would have called an error. So my personal opinion is that information messages should be treated just like warnings.

[12] It would not be totally wrong for customers to report those to SAP so that corrective action will be taken!

A final warning: Some BAPIs, e.g., the conversion BAPIs of the *BapiService* object type, sometimes return a type of "2" for an error.[13] So an application should test explicitly for the types it wants to accept as successes, not for the negation of types it sees as errors.

The Other Fields

The TYPE field exists in all the structures for the `Return` parameter. But why are there different structures at all? There are two reasons:

- Before 4.0, the length of the message class was two bytes, and the message class and message number were combined into field CODE. In 4.0, the maximum length of the message class was increased to 20 bytes. Instead of extending the CODE field of BAPIRETURN accordingly (which would have been an incompatible change and therefore against the rules), a new structure (or actually two, as mentioned above) was introduced, where the message class and the message number are represented by two distinct fields, ID and NUMBER. This is unfortunate since an application now has to be aware which Return structure a BAPI employs, and use the appropriate field names.[14]
- The second reason is that sometimes the `Return` parameter is a table, and not a structure. This will be discussed below.

The MESSAGE field contains the text from table T100 in the logon language, with the ampersands replaced by the appropriate variable information.

Fields MESSAGE_V1 to MESSAGE_V4 contain the variable text elements used to fill in the ampersands. This information can be used to construct your own message texts, utilizing the variable parts if you want to.

Fields LOG_NO and LOG_MSG_NO return the key to an Application Log entry, if the BAPI has indeed written one. Not many BAPIs take advantage of the Application Log, yet. But if you are using a BAPI that does, you can retrieve the contents of an Application Log entry using the *BapiService.ApplicationLogGetDetail* BAPI available since 4.5A.

Return Tables

Usually, the `Return` parameter is a structure, but sometimes it is a table instead. This is the case if the BAPI

- can perform multiple unrelated activities in one call, or
- wants to be able to report multiple errors for one activity (very rare).

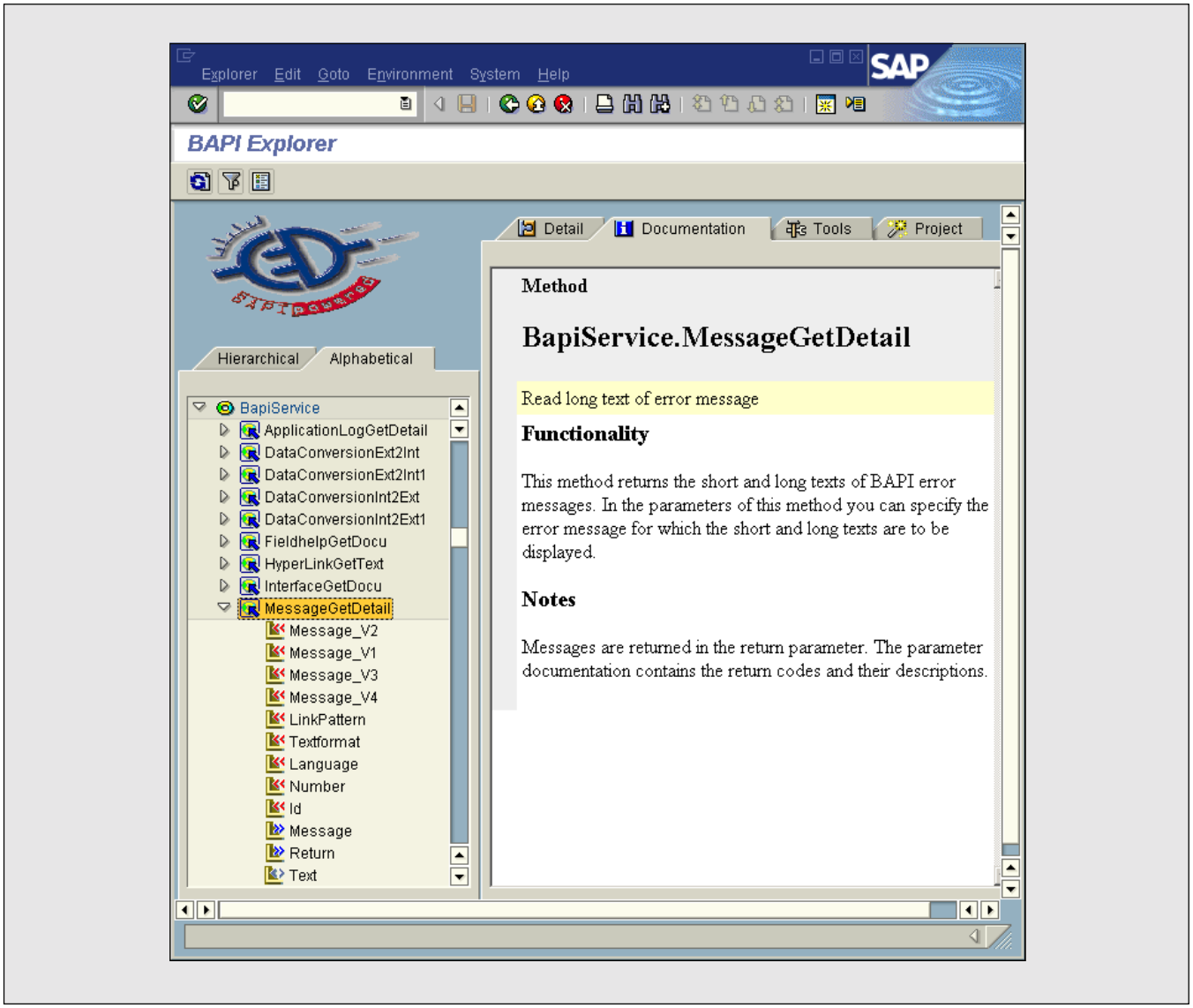
I do not know an example for the latter case, so let us discuss the former. The *BapiService* conversion BAPIs are capable of multiple conversions in one call. This is good because it saves roundtrips and therefore improves performance. Each of the conversions can fail or succeed. Normally, such a BAPI (and the conversion BAPIs are normal in that respect) will return an empty table in case everything succeeded. For each failed conversion, one row would be added to the `Return` table parameter. The application must now be able to find out which specific activity the error message relates to. Therefore, for all `Return` tables, structure BAPIRET2 is used, which has a ROW field containing the row number in the input data, for which the activity (in our example, the conversion) failed.

Again, you need to be careful. While most BAPIs using a table for the `Return` parameter

[13] See previous footnote.

[14] This is one of several reasons why I recommend to use a class that encapsulates dealing with Return messages. My own solution is discussed later in this article.

Figure 7 *BapiService.MessageGetDetail*



will return the table empty if everything worked fine, some others will actually send you back one row with a success message for every distinct activity they performed. Your error handling code should not assume that only an empty table denotes success.

The remaining new fields in BAPIRET2 — viz., PARAMETER, FIELD, and SYSTEM — are only used in very rare cases. See the *SAP BAPI Programming Guide* for additional information.

Message Documentation

In SAPGUI, a user seeking more enlightenment about a message simply double-clicks on the message, and a window with the complete documentation for the message will be shown. Since 4.5A, developers of BAPI-enabled applications can offer the same service to their users. *BapiService.MessageGetDetail* can retrieve the message documentation for any message in SAP (see **Figure 7** for a screenshot of the BAPI with its parameters and documentation).

This concludes our discussion about how to handle BAPI Return messages in a client application. Now a few words for developers of BAPIs…

Guidelines for Developers of BAPIs

These can be summed up easily: Follow the rules defined in the *SAP BAPI Programming Guide*. In my interpretation, these are:

- Use structure BAPIRET2.
- Document all messages that could ever be returned in the BAPI's documentation.
- Try to avoid "W" and "I" messages, because many applications do not handle them very well.

Of course, all the normal rules about error messages apply to BAPIs as well: Use language the addressee will understand. Avoid vagueness ("The document could not be added because the customer does not exist *or* the database is full"). Avoid messages like "This should never have happened" and "Internal error".

The BapiMessageInfo Class

After I had — little by little — discovered all the information presented so far in this article, I decided to build a Java class (which I called *BapiMessageInfo*) that encapsulates the processing of BAPI Return messages. Before introducing my implementation, I first want to discuss the main design guidelines for this class so that you can build your own solution in Java. Developers not using Java should also be able to create a similar component in other programming languages.

- The different field names in the different Return structures used by SAP should be hidden from a client developer. (My class has accessor methods that are independent of the specific structure used. See *getMessageClass()*, *getMessageNumber()*, and *getMessageKey()*.)
- There should be a simple way to find out whether a BAPI call succeeded. (Method *isBapiReturnCodeOkay()* accomplishes this.)
- There should be a way for the client programmers to accept all or selected warning and/or information messages. (Overloaded versions of method *isBapiReturnCodeOkay()* allow you to do this.)
- There should be a simple method to produce a printable message string with all relevant information. (See *getFormattedMessage()*.)
- There should be an easy way to access a message's documentation. (Method *getDocumentation()* takes care of this.)

Listings 3-4 show sample application code that utilize class *BapiMessageInfo*.

This is about as much as I believe a normal application should have to know about BAPI Return messages. Building the class to make this possible is actually not very hard once you know what you want to accomplish. The source code in the appendix to this article should speak pretty much for itself. Note that the class deals with individual Return messages (from a structure or one row in a table). Support for dealing with the complete table in a meaningful way is left as an exercise for the reader.

Listing 3: A Simple Check of a BAPI Return Message

```
JCO.Structure returnStruct =
  function.getExportParameterList().getStructure("RETURN");
BapiMessageInfo bapiMessage = new BapiMessageInfo(returnStruct);
if ( ! bapiMessage.isBapiReturnCodeOkay() ) {
  System.out.println(bapiMessage.getFormattedMessage());
}
```

Listing 4: Allowing One Warning Message and Retrieving Documentation

```
JCO.Structure returnStructure =
  function.getExportParameterList().getStructure("RETURN");
BapiMessageInfo bapiMessage = new BapiMessageInfo(returnStructure);
// Warning FN021 can be ignored in our case
if ( ! bapiMessage.isBapiReturnCodeOkay(false, false, null, "FN021") ){
  System.out.println(bapiMessage.getFormattedMessage());
  String[] documentation =
    bapiMessage.getDocumentation(mRepository);
  for (int j = 0; j < documentation.length; j++) {
    System.out.println(documentation[j]);
  }
}
```

Conclusion

As is often the case for BAPI programming, the proverbial devil lurks in the details[15]. Developers of BAPI-enabled components and applications should use a general purpose class (like *BapiMessageInfo*) to deal with the idiosyncrasies of BAPI Return messages. Create your own class using the ideas presented in this article or send me an email if you want to try my solution.

[15] And, as you saw, it has ample space to lurk.

Thomas G. Schuessler is the founder of ARAsoft (www.arasoft.de), a company offering products, consulting, custom development, and training to a worldwide base of customers. The company specializes in integration between SAP and non-SAP components and applications. ARAsoft offers various products for BAPI-enabled programs on the Windows and Java platforms. These products facilitate the development of desktop and Internet applications that communicate with R/3. Thomas is the author of SAP's BIT525 "Developing BAPI-enabled Web Applications with Visual Basic" and BIT526 "Developing BAPI-enabled Web Applications with Java" classes, which he teaches in Germany and in English-speaking countries. Thomas is a regularly featured speaker at SAP TechEd and SAPPHIRE conferences. Prior to founding ARAsoft in 1993, he worked with SAP AG and SAP America for seven years. Thomas can be contacted at thomas.schuessler@sap.com or at tgs@arasoft.de.

Appendix: Source Code for Class BapiMessageInfo

```
package de.arasoft.sap.jco;


import com.sap.mw.jco.*;
import de.arasoft.java.*;


/*
 * Copyright (c) 2001 ARAsoft GmbH
 * All Rights Reserved.
 */


/**
 * Information for a BAPI return message.
 * Helps to interpret the message and
 * can retrieve the message's additional documentation.
 *
 * @author ARAsoft GmbH
 * @version 2.0
 * @since 1.0
 */


public class BapiMessageInfo {
  private String[] mDocumentation = new String[0];
  private String mMessageType = "";
```

(continued on next page)

(continued from previous page)

```
  private String mMessageText = "";
  private String mMessageNumber = null;
  private String mMessageClass = null;
  private String mMessageVariable1 = "";
  private String mMessageVariable2 = "";
  private String mMessageVariable3 = "";
  private String mMessageVariable4 = "";
  private String mAppLogNo = null;
  private String mAppLogMsgNo = null;
  private boolean initialized = false;


/**
 * Constructor to be used if all information about a message
 * is already known.
 * @param messageClass     Message class (application area).
 * @param messageNumber    Message number.
 * @param messageVariable1 The first message variable.
 * @param messageVariable1 The second message variable.
 * @param messageVariable1 The third message variable.
 * @param messageVariable1 The fourth message variable.
 * @param messageText      The message text.
 * @param documentation    The documentation.
 */
  public BapiMessageInfo(String messageClass, String messageNumber,
                         String messageVariable1,
                         String messageVariable2,
                         String messageVariable3,
                         String messageVariable4,
```



```
                          String messageText, String[] documentation
                         ) {
    mMessageClass = messageClass;
    mMessageNumber = messageNumber;
    mMessageVariable1 = messageVariable1;
    mMessageVariable2 = messageVariable2;
    mMessageVariable3 = messageVariable3;
    mMessageVariable4 = messageVariable4;
    mMessageText = messageText;
    mDocumentation = documentation;
    initialized = true;
  }

/**
 * Constructor to be used if you know the message class and number
 * and have four message variables.
 * Use <code>getMessageText(JCoComponentConnector connector)</code>
 * and <code>getDocumentation(JCoComponentConnector connector)</code>
 * to find out additional information.
 * @param messageClass     Message class (application area).
 * @param messageNumber    Message number.
 * @param messageVariable1 The first message variable.
 * @param messageVariable2 The second message variable.
 * @param messageVariable3 The third message variable.
 * @param messageVariable4 The fourth message variable.
 * @see #getMessageText(JCoComponentConnector)
 * @see #getDocumentation(JCoComponentConnector)
 */
```

(continued on next page)

(continued from previous page)

```
  public BapiMessageInfo(String messageClass, String messageNumber,
                         String messageVariable1,
                         String messageVariable2,
                         String messageVariable3,
                         String messageVariable4
                        ) {
    mMessageClass = messageClass;
    mMessageNumber = messageNumber;
    mMessageVariable1 = messageVariable1;
    mMessageVariable2 = messageVariable2;
    mMessageVariable3 = messageVariable3;
    mMessageVariable4 = messageVariable4;
  }

// ... Similar constructors for one through three message variables
//     omitted to save space ...

/**
 * Constructor to be used if you know the message class and number.
 * Use <code>getMessageText(JCoComponentConnector connector)</code>
 * and <code>getDocumentation(JCoComponentConnector connector)</code>
 * to find out additional information.
 * @param messageClass     Message class (application area).
 * @param messageNumber    Message number.
 * @see #getMessageText(JCoComponentConnector)
 * @see #getDocumentation(JCoComponentConnector)
 */
  public BapiMessageInfo(String messageClass, String messageNumber) {
```



```
    mMessageClass = messageClass;
    mMessageNumber = messageNumber;
  }

/**
 * Constructor to be used if you have an actual return message from
 * a BAPI call. If you pass a JCo table, the current row will be
 * interpreted.
 * @param record The JCo structure or JCo table.
 */
  public BapiMessageInfo(JCO.Record record) throws ARAsoftException {
    try {
      mMessageType = record.getString("TYPE");
      mMessageText = record.getString("MESSAGE");
      mMessageVariable1 = record.getString("MESSAGE_V1");
      mMessageVariable2 = record.getString("MESSAGE_V2");
      mMessageVariable3 = record.getString("MESSAGE_V3");
      mMessageVariable4 = record.getString("MESSAGE_V4");
      mAppLogNo = record.getString("LOG_NO");
      mAppLogMsgNo = record.getString("LOG_MSG_NO");
      if (record.hasField("CODE")) {
        String s = record.getString("CODE");
        if (s.length() < 3) {
          mMessageClass = s;
          mMessageNumber = "";
        }
        else {
          mMessageClass = s.substring(0, 2);
```

(continued on next page)

(continued from previous page)

```
            mMessageNumber = s.substring(2);
          }
        }
        else {
          mMessageClass = record.getString("ID");
          mMessageNumber = record.getString("NUMBER");
        }
      }
      catch (Exception ex) {
        throw new ARAsoftException("Not a valid Return record.");
      }
    }

  /**
   * Returns the application log number.
   * @return The application log number
   */
    public String getApplicationLogNumber() {
      return mAppLogNo;
    }

  /**
   * Returns the application log message number.
   * @return The application log message number
   */
    public String getApplicationLogMessageNumber() {
      return mAppLogMsgNo;
    }
```



```
/**
 * Returns the message type.
 * @return The message type
 */
  public String getMessageType() {
    return mMessageType;
  }

/**
 * Returns the message class and message number concatenated.
 * @return The message class and message number concatenated.
 */
  public String getMessageKey() {
    return mMessageClass + mMessageNumber;
  }

/**
 * Returns the message class (application area).
 * @return The message class
 */
  public String getMessageClass() {
    return mMessageClass;
  }

/**
 * Returns the message number.
 * @return The message number
 */
```

(continued on next page)

(continued from previous page)

```
  public String getMessageNumber() {
    return mMessageNumber;
  }

/**
 * Returns message variable 1.
 * @return The message variable 1
 */
  public String getMessageVariable1() {
    return mMessageVariable1;
  }

/**
 * Returns message variable 2.
 * @return The message variable 2
 */
  public String getMessageVariable2() {
    return mMessageVariable2;
  }

/**
 * Returns message variable 3.
 * @return The message variable 3
 */
  public String getMessageVariable3() {
    return mMessageVariable3;
  }
```



```
/**
 * Returns message variable 4.
 * @return The message variable 4
 */
  public String getMessageVariable4() {
    return mMessageVariable4;
  }

/**
 * Returns the message text.
 * @return The message text
 */
  public String getMessageText() {
    return mMessageText;
  }

/**
 * Returns the message text. Useful if you have used one
 * of the constructors without the message text.
 * @return The message text.
 * @param connector The connector to SAP
 */
  public String getMessageText(JCoComponentConnector connector)
         throws ARAsoftException {
    if ( ! initialized )
      init(connector);
    return mMessageText;
  }
```

(continued on next page)

(continued from previous page)

```
/**
 * Returns the message documentation. This will only work properly if
 * you have either used the constructor
 * that includes the documentation or have used
 * <code>getDocumentation(JCoComponentConnector connector)
 * </code> before.
 * @return The message documentation
 */
  public String[] getDocumentation() {
    return mDocumentation;
  }

/**
 * Returns the message documentation.
 * @return The message documentation
 * @param connector The connector to SAP
 */
  public synchronized String[] getDocumentation
        (JCoComponentConnector connector)
        throws ARAsoftException {
    if ( ! initialized )
      init(connector);
    return mDocumentation;
  }

/**
 * Returns the message documentation.
 * @return The message documentation
```



```
 * @param repository The repository
 */
  public synchronized String[] getDocumentation
        (JCoRepository repository)
        throws ARAsoftException {
    return
      this.getDocumentation(repository.getJCoComponentConnector());
  }

/**
 * Returns a formatted version of the message, e.g.,
 * "E XX-007: Some error has occurred".
 * @return The formatted message
 */
  public String getFormattedMessage() {
    return getMessageType() + " " + getMessageClass() +
           "-" + getMessageNumber() + ": " + getMessageText();
  }

/**
 * Check whether the BAPI executed correctly. "S" or an empty string
 * in field TYPE are interpreted as indicating success.
 * @return Is the return code okay?
 */
  public boolean isBapiReturnCodeOkay() {
    return getMessageType().equals("") || getMessageType().equals("S");
  }
```

(continued on next page)

(continued from previous page)

```
/**
 * Check whether the BAPI executed correctly. "S" or an empty string
 * in field TYPE are always interpreted as indicating success, you can
 * control whether Information messages ("I") and/or
 * Warning messages ("W") are accepted as well.
 * @return Is the return code okay?
 * @param allowInformationMessages Should "I" messages be acceptable?
 * @param allowWarningMessages Should "W" messages be acceptable?
 */
  public boolean isBapiReturnCodeOkay(boolean allowInformationMessages,
                                      boolean allowWarningMessages) {
    return getMessageType().equals("") ||
           getMessageType().equals("S") ||
           (getMessageType().equals("I") &&
            allowInformationMessages) ||
           (getMessageType().equals("W") &&
            allowWarningMessages) ;
  }

/**
 * Check whether the BAPI executed correctly. "S" or an empty string
 * in field TYPE are always interpreted as indicating success, you can
 * control whether Information messages ("I") and/or
 * Warning messages ("W") are accepted as well.
 * In addition, you can specify which specific information
 * and/or warning messages should be interpreted as success.
 * @return Is the return code okay?
 * @param allowInformationMessages Should "I" messages be acceptable?
```



```
* @param allowWarningMessages Should "W" messages be acceptable?
* @param informationMessageKeys Array of acceptable "I" message keys?
* @param warningMessageKeys Array of acceptable "W" message keys?
* @see #getMessageKey
*/
 public boolean isBapiReturnCodeOkay(boolean allowInformationMessages,
                                     boolean allowWarningMessages,
                                     String[] informationMessageKeys,
                                     String[] warningMessageKeys) {
   boolean b = getMessageType().equals("") ||
               getMessageType().equals("S") ||
               (getMessageType().equals("I") &&
                allowInformationMessages) ||
               (getMessageType().equals("W") &&
                allowWarningMessages) ;
   if (b) return true;
   String s;
   if (informationMessageKeys != null &&
       getMessageType().equals("I")) {
     for (int i = 0; i < informationMessageKeys.length; i++) {
       s = informationMessageKeys[i];
       if (s != null && s.equals(getMessageKey()))
         return true;
     }
   }
   if (warningMessageKeys != null && getMessageType().equals("W")) {
     for (int i = 0; i < warningMessageKeys.length; i++) {
       s = warningMessageKeys[i];
```

(continued on next page)

(continued from previous page)

```
        if (s != null && s.equals(getMessageKey()))
          return true;
      }
    }
    return false;
  }

/**
 * Check whether the BAPI executed correctly. "S" or an empty string
 * in field TYPE are always interpreted as indicating success, you can
 * control whether Information messages ("I") and/or
 * Warning messages ("W") are accepted as well.
 * In addition, you can specify which specific information
 * and/or warning messages should be interpreted as success.
 * @return Is the return code okay?
 * @param allowInformationMessages Should "I" messages be acceptable?
 * @param allowWarningMessages Should "W" messages be acceptable?
 * @param informationMessageKey Key of one acceptable "I" message?
 * @param warningMessageKey Key of one acceptable "W" message?
 * @see #getMessageKey
 */
  public boolean isBapiReturnCodeOkay(boolean allowInformationMessages,
                                      boolean allowWarningMessages,
                                      String informationMessageKey,
                                      String warningMessageKey) {
    return isBapiReturnCodeOkay(allowInformationMessages,
                                allowWarningMessages,
                                new String[] { informationMessageKey },
```



```
                                new String[] { warningMessageKey }
                                );
}

private void init (JCoComponentConnector connector)
                  throws ARAsoftException {
  JCO.Function function = null;
  try {
    function = connector.createFunction("BAPI_MESSAGE_GETDETAIL");
    if (function == null)
      throw new ARAsoftException
        ("BAPI_MESSAGE_GETDETAIL not found in SAP.");
    function.getImportParameterList().
      setValue(getMessageClass(), "ID");
    function.getImportParameterList().
      setValue(getMessageNumber(), "NUMBER");
    function.getImportParameterList().
      setValue(connector.getUserLanguage(), "LANGUAGE");
    function.getImportParameterList().
      setValue("ASC" , "TEXTFORMAT");
    function.getImportParameterList().
      setValue(getMessageVariable1() , "MESSAGE_V1");
    function.getImportParameterList().
      setValue(getMessageVariable2() , "MESSAGE_V2");
    function.getImportParameterList().
      setValue(getMessageVariable3() , "MESSAGE_V3");
    function.getImportParameterList().
      setValue(getMessageVariable4() , "MESSAGE_V4");
```

(continued on next page)

(continued from previous page)

```
        connector.executeStateless(function);
        JCO.Structure retStruct =
          function.getExportParameterList().getStructure("RETURN");
        BapiMessageInfo bmi = new BapiMessageInfo(retStruct);
        if ( ! bmi.isBapiReturnCodeOkay() )
          if ( ! bmi.getMessageNumber().equals("104") )
            throw new ARAsoftException(bmi.getFormattedMessage());
        mMessageText =
          function.getExportParameterList().getString("MESSAGE");
        JCO.Table table =
          function.getTableParameterList().getTable("TEXT");
        String[] strings = new String[table.getNumRows()];
        for (int i = 0; i < table.getNumRows(); i++, table.nextRow()) {
          strings[i] = table.getString(0);
        }
        mDocumentation = strings;
        initialized = true;
      }
      catch (JCO.Exception ex) {
        throw new ARAsoftException(ex);
      }
      catch (ARAsoftException ex) { throw ex; }
      catch (Exception ex) {
        throw new ARAsoftException(ex);
      }
    }
  }
```

