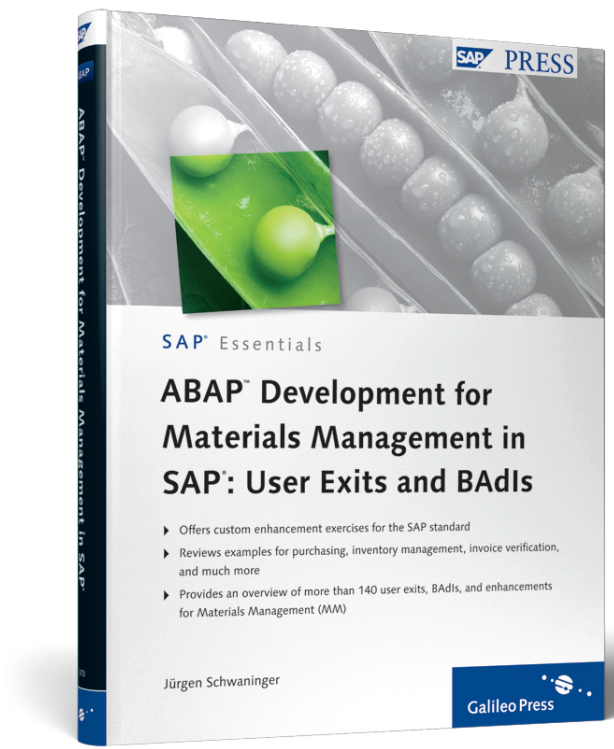
SAP PRESS

Jürgen Schwaninger

ABAP™ Development for Materials Management in SAP®: User Exits and BAdIs



www.sap-press.com

Galileo Press

Bonn • Boston

Contents at a Glance

Contents

Appendices 193

5 User Exits and BAdIs in Inventory Management

The concept of Inventory Management in SAP Materials Management (MM) basically involves summarizing the management of all warehouse stocks with regard to value and quantity as well as the associated goods movements. Because the conceivable processes in this core area are very comprehensive, there are also many setting options in the system. This is also reflected in the number of possible program-related enhancements. In this chapter, you will learn about the most important enhancements for Inventory Management.

5.1 Custom Fields in Transaction MIGO

With the implementation of the central Transaction MIGO for all goods movements, a powerful BAdI is provided: `MB_MIGO_BADI`. With this BAdI, you can integrate and post custom fields as custom tabstrips at the header and item level. You can also carry out input checks in your custom fields or populate many standard fields with default values.

Display of Custom Data

If you execute the following example, you may find that the subsequent display of the data of a posted material document at item level doesn't work. This is the case when SAP Note 1029951 has been manually imported into your system or support packages have been imported.

As a result of the changes from this note, the method `LINE_MODIFY`, which is actually provided for reading the item data from the database, will no longer run when material documents are displayed. Unfortunately, it isn't possible to use another method for this because the associated standard item data (structure `GOITEM`) isn't provided in all other relevant methods, and you therefore cannot assign your data to a specific document line.

With SAP Note 1477221, the method `LINE_MODIFY` will run again. If this note hasn't yet been released for customers when you read these lines, you can modify the program location relevant for the call of the method, as described in SAP Note 1136344. The example given here will be fully functional in both cases.

5.1.1 Custom Fields: An Overview

The capability of the BAdI `MB_MIGO_BADI` results in a certain amount of complexity. There are 17 methods that are called at many different times. Nevertheless, to make the next example as clear as possible, you'll only view the implementation of custom fields at the header and item level (see Figure 5.1). This reduces the BAdI to 11 methods with which the minimum requirements can be fulfilled. In a second step, you'll then learn about the function of the remaining methods in a brief overview.

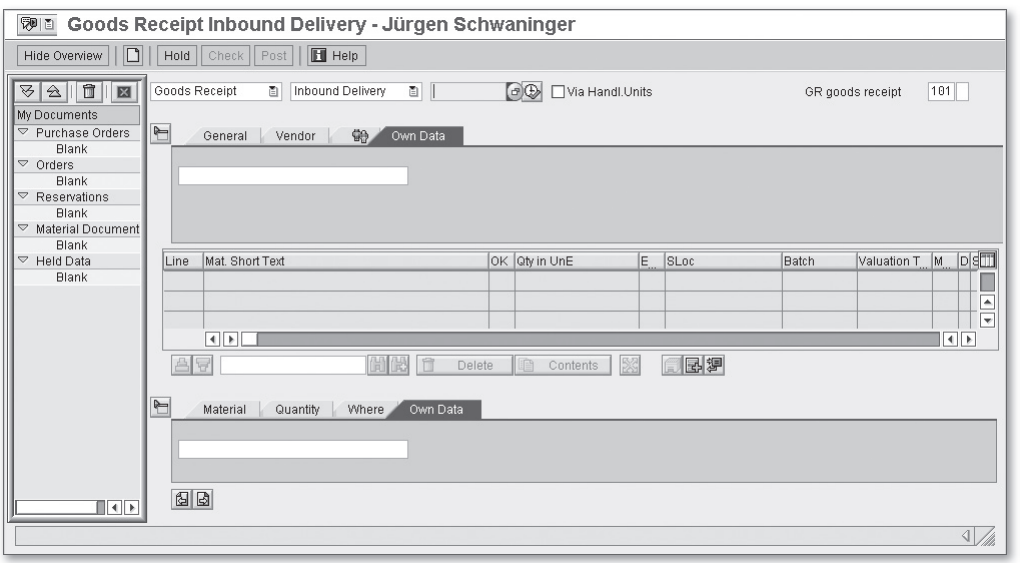


Figure 5.1 Custom Fields in Transaction MIGO

Function Group for the Management of Dynpros

Just as with the custom fields in the purchase order (see Section 3.1, Custom Fields in Purchase Orders, in Chapter 3), you first need a function group to include the dynpros and some function modules for the exchange of data and for updating custom fields. However, this function group can be implemented for the BAdI `MB_MIGO_BADI` much more easily and must assume fewer tasks.

You have to work with two BAdIs in the purchase order to activate the custom fields. The function group therefore not only has to assume the communication with the dynpros but also must map the communication between the BAdIs. You can, however, execute all the necessary steps directly in the BAdI `MB_MIGO_BADI` for

Transaction MIGO, so additional communication with another BAdI isn't needed. You can create a dynpro in the function group for the header and item data and create two function modules for each dynpro to store and retrieve the data in the dynpro. You also need an update module to save your custom data. To post your custom data, you need to define custom tables in the ABAP Dictionary.

Usage of the BAdI MB_MIGO_BADI

As soon as you've created the function group, you can begin with the implementation of the BAdI. You keep the data on the custom fields for runtime in the attributes of the BAdI. First, these areas need to be defined.

`MB_MIGO_BADI` can be implemented many times. However, there are only five custom tabstrips for the header and item levels. For this reason, precisely five active implementations are allowed. The method `INIT` is activated when Transaction MIGO is started, and it's used to register an implementation and populate a tabstrip.

With the initialization of a document (new document or display of an existing document), the methods will run from Table 5.1. The methods `MODE_SET` and `STATUS_AND_HEADER` are also triggered when the document status changes.

Method	Description
`RESET`	This method is used to initialize all custom data in the attributes of the BAdI implementation.
`MODE_SET`	In this method, you obtain information on the action (goods receipt, goods issue, etc.) and the chosen reference (purchase order, reservation, etc.) chosen by the user; these are the fields that are always available in Transaction MIGO in the top-left side. You can evaluate this information, and store it in the attributes of the implementation for subsequent use.
`STATUS_AND_HEADER`	This method is mainly used to fill the custom header data when an existing document is read.

Table 5.1 Initialization of a Document and Status Change

The methods are also executed with each dialog step for header data from Table 5.2 and with each dialog step on item data from Table 5.3. The method `LINE_MODIFY` is also executed when a line is added either when the user makes an entry or when an existing material document is read from the database.

Method	Description
PBO_HEADER	This method is called before the header data are displayed. You must transfer the data to be displayed to the function group and specify the dynpro to be displayed. The tabstrip header is also specified here.
PAI_HEADER	This method runs as soon as the user has entered the data. You need to retrieve the possibly changed data from the function group.

Table 5.2 Methods in the Dialog Step on Header Data

Method	Description
PBO_DETAIL	This method is called before the item data is displayed. In the parameter I_LINE_ID, you obtain the current item and must transfer the associated data in the function group. You also need to specify the dynpro to be displayed as well as the header of the tabstrip here.
PAI_DETAIL	You retrieve the data from the dynpro after the user has entered it. First, check whether data has changed, and if applicable, set the parameter E_FORCE_CHANGE. The method LINE_MODIFY is activated when the parameter has been set.
LINE_MODIFY	This method is activated when a line has been changed (see PAI_DETAIL) or when a new line has been inserted. When adding new lines, you must initialize your custom fields, or if the document is being read in the display mode, you must read the data from the database. If required, you have the option here of writing standard fields that you've also possibly integrated in your dynpro back to the standard items.
LINE_DELETE	This method runs when the user has deleted a line of the material document. In this case, you also need to delete the associated custom data.

Table 5.3 Methods in the Dialog Step for Item Data

Finally, you must use the method POST_DOCUMENT to format your custom data and call the update module from your function group. The final line number for each document line is set only in this method, as contained in Table MSEG — that is, in

the table in which the standard item data is stored. You should convert your custom data accordingly so that you can assign it more easily later on.

5.1.2 Preparations in the ABAP Dictionary

At this point, it makes sense to refer again to the general data definitions in the ABAP Dictionary. Both header and item data are implemented in this example, and two suitable database tables are required to store the data. The following example again only uses a simple variant for this, and only one text field is used as a custom field.

1. Switch to the ABAP Dictionary (Transaction SE11), and create the table for the header data. Besides the actual text field `ZK_FELD1`, you also need the key fields; that is, the material document number and posting year. The table name `ZMB_MIGOHEAD` has been used in the example. You can view the associated fields in Table 5.4.

Field Name	Key Field	Data Element
MANDT	Yes	MANDT
MBLNR	Yes	MBLNR
MJAHR	Yes	MJAHR
ZK_FELD1		CHAR32

Table 5.4 Fields of the Table ZMB_MIGOHEAD

2. Create the table for the item data. A line number is necessary besides the previously used key fields. The custom field is called `ZP_FELD1` here. The table name `ZMB_MIGOITEM` is used. The associated fields can be viewed in Table 5.5.

Field Name	Key Field	Data Element
MANDT	Yes	MANDT
MBLNR	Yes	MBLNR
MJAHR	Yes	MJAHR
ZEILE	Yes	MBLPO
ZP_FELD1		CHAR32

Table 5.5 Fields of the Table ZMB_MIGOITEM

5.1.3 Preparation of the Function Group

You next need to deal with the function group and the dynpro. In this example, both the header and the item data are implemented. A dynpro is also required here. Moreover, two function modules for the data communication and one update module are necessary in each case. Another module informs the function group on the current status of Transaction MIGO. If this isn't in the display mode, the fields in the dynpros aren't ready for input.

1. Switch to Transaction SE80. Select the Function Group option in the Repository Browser, and specify ZMB_MIGO as the name. Press Enter. Confirm in the dialog that this function group should now be generated.

Dynpro Variations

In this example, a dynpro is defined for the header and item data in each case. You certainly have the option, however, of preparing different situations as well as different dynpros. Your requirement can, for example, be different in goods receipt than in goods issue. You decide which dynpro is actually displayed at runtime via your programming.

In the method MODE_SET of the BAdI MB_MIGO_BADI, you learn which action (displays, goods receipt, goods issue, etc.) has just been selected. You can keep this information in an attribute of the class and then dynamically set the dynpro in method PBO_HEADER or PBO_DETAIL based on this information.

2. Create the dynpro for the header data. Now right-click ZMB_MIGO in the object browser, and select Create • Dynpro from the context menu. The dynpro number should be 0100. In the properties, you must set Subscreen as the Dynpro Type because other types cannot be integrated with tabstrips, and you would get a short dump upon implementation.
3. Click the Layout button to switch to the Screen Painter. Here you choose Dictionary/Program Fields in the GoTo • Secondary Window menu, or press F6. In the window displayed, specify ZMB_MIGOHEAD as the table name and press Enter. Mark the field ZK_FELD1, and press Enter again (see Figure 5.2). Using the mouse, now position the field in the top-left corner of the dynpro. Save and activate the dynpro, and then exit the Screen Painter.
4. Create the dynpro 0200 for the item data. Choose the field ZP_FELD1 from Table ZMB_MIGOITEM, and then proceed as in the previous step.

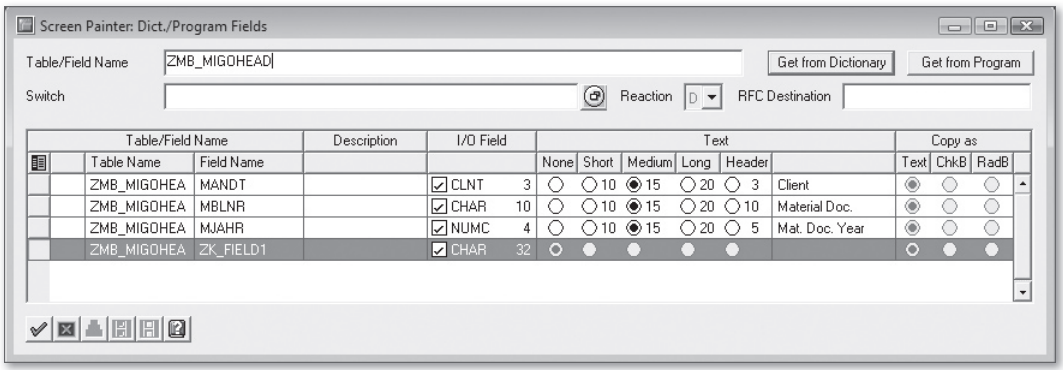


Figure 5.2 Dict/Program Fields Window

5. So that you are able to subsequently fill the fields in these dynpros with data, fields with an identical name must be available in the global data. Create these with the keyword `TABLES`. Furthermore, the fields are then ready for input only if the document isn't in a display mode. The mode is to be subsequently set via a function module. You can create it now, however, as a flag (`gv_outputonly`, data type `C`) in the global data.

   You switch to the global data via the Object Browser by navigating to the include `LZMB_MIGOTOP` in the INCLUDES section. An example is given in Listing 5.1.

```
FUNCTION-POOL ZMB_MIGO.                    "MESSAGE-ID ..
* Work structures for dynpros
TABLES: zmb_migohead,
        zmb_migoitem.

* Display mode?
DATA gv_outputonly TYPE c.
```

Listing 5.1 Global Data in LZMB_MIGOTOP

6. If the flag `gv_outputonly` is set, then the fields in the dynpro aren't ready for input. Therefore, change to the flow logic of dynpro 0100. Remove the comments in the `PROCESS BEFORE OUTPUT` time from the `STATUS_0100` module. Double-click the module name. To generate the module, use the suggested include name.

   Set the value of `SCREEN-INPUT` based on the field `gv_outputonly` with a `LOOP AT SCREEN`. An example is given in Listing 5.2.

```
*---------------------------------------------------------------*
***INCLUDE LZMB_MIGO001 .
*---------------------------------------------------------------*
MODULE status_0100 OUTPUT.
  LOOP AT SCREEN.
    IF gv_outputonly IS INITIAL.
      screen-input = 1.
    ELSE.
      screen-input = 0.
    ENDIF.

    MODIFY SCREEN.
  ENDLOOP.
ENDMODULE.                 " STATUS_0100  OUTPUT
```

Listing 5.2 Flow Logic on Dynpro 0100

7. Dynpro 0200 is also not ready for input in the display mode. Switch to dynpro 0200, and proceed as in the previous step.

Next you can begin with the function modules:

1. An overview of the required modules is given in Table 5.6. Begin with the module `ZMB_MIGO_SETSTATUS`. You only need a flag as an input parameter; from this, you overwrite the global field `gv_outputonly` (see Listing 5.3).

Function Module	Description
ZMB_MIGO_SETSTATUS	Sets the flag `gv_outputonly`
ZMB_MIGO_HEAD_SET	Sets data for dynpro 100
ZMB_MIGO_HEAD_GET	Returns data from dynpro 100
ZMB_MIGO_ITEM_SET	Sets data for dynpro 200
ZMB_MIGO_ITEM_GET	Returns data from dynpro 200
ZMB_MIGO_POST	Updates header and item data

Table 5.6 Function Modules for Data Exchange

```
FUNCTION ZMB_MIGO_SETSTATUS.
*"----------------------------------------------------------------
*"*"Local interface:
*"  IMPORTING
*"     REFERENCE(I_OUTPUTONLY) TYPE  C
```

```
*"----------------------------------------------------------------
* Set input status
  gv_outputonly = i_outputonly.

ENDFUNCTION.
```

Listing 5.3 Coding on ZMB_MIGO_SETSTATUS

2. Create the modules ZMB_MIGO_HEAD_SET and ZMB_MIGO_HEAD_GET. You need a structure with Table ZMB_MIGOHEAD as a data type to use it as an import or export parameter. Simply copy the data in the respective direction between the parameters and the work structure previously defined with TABLES (see Listing 5.4 and Listing 5.5).

```
FUNCTION ZMB_MIGO_HEAD_SET.
*"----------------------------------------------------------------
*"*"Local interface:
*"  IMPORTING
*"     REFERENCE(I_HEAD) TYPE  ZMB_MIGOHEAD
*"----------------------------------------------------------------
* Prepare data for dynpro
  zmb_migohead = i_head.

ENDFUNCTION.
```

Listing 5.4 Coding on ZMB_MIGO_HEAD_SET

```
FUNCTION zmb_migo_head_get.
*"----------------------------------------------------------------
*"*"Local interface:
*"  EXPORTING
*"     REFERENCE(E_HEAD) TYPE  ZMB_MIGOHEAD
*"----------------------------------------------------------------
* Return data from dynpro
  e_head = zmb_migohead.

ENDFUNCTION.
```

Listing 5.5 Coding on ZMB_MIGO_HEAD_GET

3. For the function modules ZMB_MIGO_ITEM_SET and ZMB_MIGO_ITEM_GET, proceed exactly as before. Only use ZMB_MIGOITEM as a reference (see Listing 5.6 and Listing 5.7).

```
FUNCTION zmb_migo_item_set.
*"----------------------------------------------------------------------
*"*"Local interface:
*"  IMPORTING
*"     REFERENCE(I_ITEM) TYPE  ZMB_MIGOITEM
*"----------------------------------------------------------------------
* Prepare data for dynpro
  zmb_migoitem = i_item.

ENDFUNCTION.
```

Listing 5.6 Coding on ZMB_MIGO_ITEM_SET

```
FUNCTION ZMB_MIGO_ITEM_GET.
*"----------------------------------------------------------------------
*"*"Local interface:
*"  EXPORTING
*"     REFERENCE(E_ITEM) TYPE  ZMB_MIGOITEM
*"----------------------------------------------------------------------
* Return data from dynpro
  e_item = zmb_migoitem.

ENDFUNCTION.
```

Listing 5.7 Coding on ZMB_MIGO_ITEM_GET

4. Create the function module ZMB_MIGO_POST. The module must be marked in the Processing Type section as an Update Module (Start immed.) (see Figure 5.3).

   Create an import parameter (suggestion I_HEAD) as a structure for Table ZMB_MIGOHEAD. Activate the Pass Value option for this parameter because no reference parameters are allowed in update modules. Then define a table parameter (suggestion T_ITEMS) with reference to Table ZMB_MIGOITEM.

5. With regard to update modules, you should always integrate several logical tests to prevent erroneous updates as much as possible. You need to check whether all key fields of the header and item data are filled. Otherwise, an exception will be triggered. To do this, create the DATA_ERROR exception in the Exceptions tabstrip.

   If the writing of the data fails, however, an exception will also be triggered. You create this exception as INSERT_ERROR. Triggering an exception results in an

update termination, and if there's an error, the complete material document is not posted.

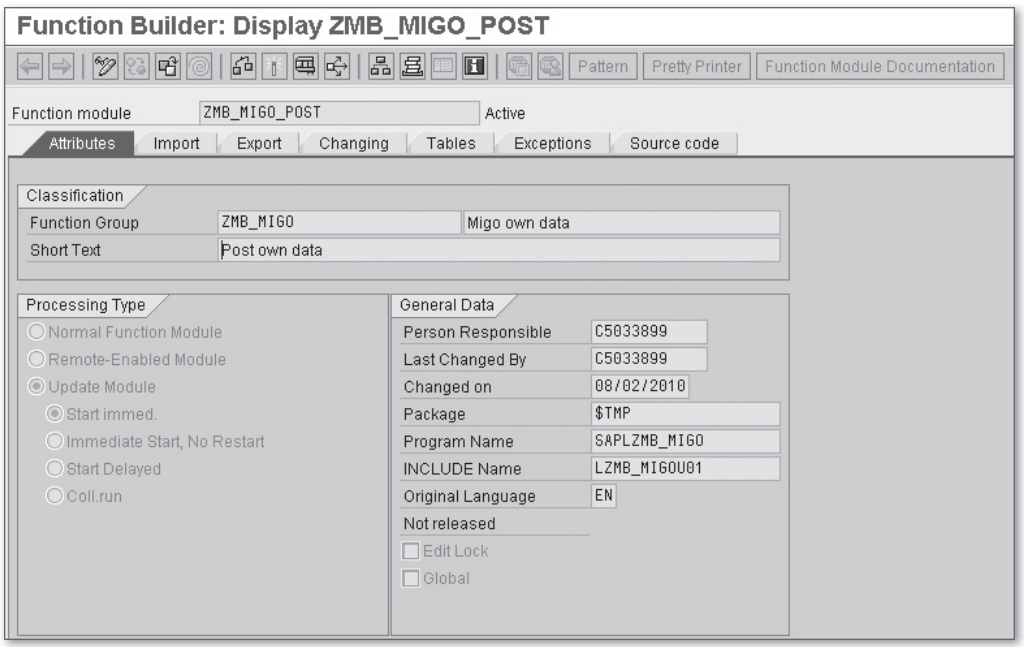


Figure 5.3 Properties of the Update Module

6. Switch to the Source Text, and begin with the programming. Check whether the key fields of the header data are filled (parameter `I_HEAD`, fields `MBLNR` and `MJAHR`). Proceed exactly as before in a loop via the item data (Table `T_ITEMS`). Check also the line number (field `ZEILE`). If an error occurs, trigger the exception `DATA_ERROR` by using the command `RAISE`.
7. If no error occurs, update the data using the command `INSERT`. If an error occurs here (`SY-SUBRC <> 0`), trigger the exception `INSERT_ERROR` (see Listing 5.8).

```
FUNCTION zmb_migo_post.
*"----------------------------------------------------------------------
*"*"Update module:
*"
*"*"Local interface:
*"  IMPORTING
*"     VALUE(I_HEAD) TYPE  ZMB_MIGOHEAD
*"  TABLES
```

```
*"      T_ITEMS STRUCTURE  ZMB_MIGOITEM
*"  EXCEPTIONS
*"      INSERT_ERROR
*"      DATA_ERROR
*"----------------------------------------------------------------------

* Check transferred data
  IF i_head-mblnr IS INITIAL OR
     i_head-mjahr IS INITIAL.
    RAISE data_error.
  ENDIF.

  LOOP AT t_items.
    IF t_items-mblnr IS INITIAL OR
       t_items-mjahr IS INITIAL OR
       t_items-zeile IS INITIAL.
      RAISE data_error.
    ENDIF.
  ENDLOOP.

* Write header data
  INSERT zmb_migohead FROM i_head.
  IF sy-subrc <> 0.
    RAISE insert_error.
  ENDIF.

* Write item data
  INSERT zmb_migoitem FROM TABLE t_items.
  IF sy-subrc <> 0.
    RAISE insert_error.
  ENDIF.
ENDFUNCTION.
```

Listing 5.8 Coding on ZMB_MIGO_POST

8. Save all changes, and check whether all components of the function group have been activated.

5.1.4 Preparation and Status Management in MB_MIGO_BADI

From now on, you can completely focus on the BAdI. All preparations have been made. As described earlier, all necessary data are kept in the attributes of the class. These are therefore created first.

1. Switch to Transaction SE19, and create a new implementation for the BAdI MB_MIGO_BADI. Double-click the name of the implementing class ZCL_IM_MB_MIGO_BADI, and then switch to the Attributes tabstrip.
2. Create the attributes according to Table 5.7. Consider the following notes:
   - GC_CLASS
     Each implementation of the BAdI MB_MIGO_BADI must clearly be identified toward Transaction MIGO. This happens via a constant of the type MIGO_CLASS_ID, which you must define in the attributes because you use these in several areas. In the example given, the constant has been specified with the initial value 'MIGO_OWN'. Note that the single quotation marks are necessary here.
   - GV_LINEID
     In this line number, always note the item transferred last in the dynpro.
   - GV_ACTION
     This attribute contains the currently chosen action (see Table 5.8).
   - GV_REFDOC
     This attribute contains the currently chosen reference document type (see Table 5.9).
   - GS_HEADER
     This structure contains the current header data on your tabstrip.
   - GT_ITEM
     This internal table contains all items on the current document. You can only define internal tables in the attributes when you refer to a table type defined in the ABAP Dictionary or you create a local type. In the following example, a local type (TT_MIGOITEM) has been used that is still currently unknown.

Attribute	Type	Visibility	Reference Type
GC_CLASS	Constant	Private	MIGO_CLASS_ID
GV_LINEID	Instance Attribute	Private	GOITEM-GLOBAL_COUNTER
GV_ACTION	Instance Attribute	Private	GOACTION
GV_REFDOC	Instance Attribute	Private	REFDOC
GS_HEADER	Instance Attribute	Private	ZMB_MIGOHEAD
GT_ITEM	Instance Attribute	Private	TT_MIGOITEM

Table 5.7 Attributes of the Implementing Class

Action/Operation	Description
A01	Goods receipt
A02	Return delivery
A03	Cancellation
A04	Display
A05	Release Goods receipt (GR) blocked stock
A06	Subsequent delivery
A07	Goods issue
A08	Transfer posting
A09	Remove from storage
A10	Place in storage
A11	Subsequent adjustment

Table 5.8 List of Actions in Transaction MIGO

Reference Document	Description
R01	Purchase order
R02	Material document
R03	Delivery note
R04	Inbound delivery
R05	Outbound delivery
R06	Transport
R07	Transport ID code
R08	Order
R09	Reservation
R10	Other

Table 5.9 List of the Possible Reference Documents in Transaction MIGO

3. Switch to the Types tabstrip to create the data type `TT_MIGOITEM`. To define a table, you need the direct type input (see Figure 5.4).

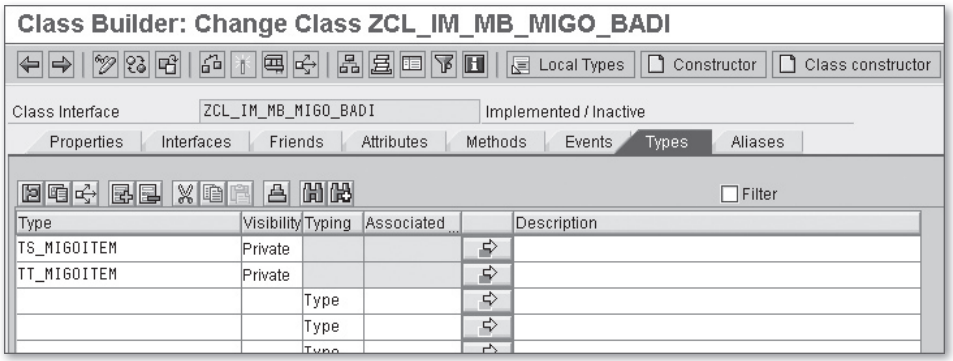


Figure 5.4 Enhanced Type Definition in Classes

4. Enter the name of the data type: `TT_MIGOITEM`.
5. Choose the PRIVATE entry in the VISIBILITY column because this type is only used within the class.
6. Click the yellow arrow icon on the right next to the reference type (direct type input). You're now in the ABAP Editor and can define the type as required.
7. Besides the actual data from table `ZMB_MIGOITEM`, the internal table requires another field `LINE_ID` that displays the internal number during the entry. Create a type that is made up of the field `LINE_ID` (type `MB_LINE_ID`) and `ZMB_MIGOITEM`. Name this type `TS_MIGOITEM`.
8. Define the type `TT_MIGOITEM` as an internal table for `TS_MIGOITEM`. The complete section should appear as follows:

```
PRIVATE SECTION.
  TYPES BEGIN OF ts_migoitem.
   TYPES line_id TYPE  mb_line_id.
   INCLUDE TYPE zmb_migoitem.
  TYPES END OF ts_migoitem.

  TYPES tt_migoitem TYPE TABLE OF ts_migoitem.
```

9. Save and activate your changes.

The first methods can now be programmed. Start with the method `INIT`. As described earlier, this method is used to announce the implementation in Transaction MIGO. A maximum of five implementations may be active at the same time.

You can now prepare the methods RESET and MODE_SET (refer to Table 5.1). You'll focus on the method STATUS_AND_HEADER later.

1. Switch to the Methods tabstrip of the class ZCL_IM_MB_MIGO_BADI, and then navigate to method INIT. Here you must only transfer defined constants GC_CLASS to the internal Table CT_INIT in the attributes. This table may already contain other implementations. Therefore, don't overwrite the table; instead, attach your constant via APPEND. Because Table CT_INIT only consists of one field, you can attach the constant directly. You don't need a local structure (see Listing 5.9).

```
METHOD if_ex_mb_migo_badi~init.
* Register implementation
  APPEND gc_class TO ct_init.
ENDMETHOD.
```

Listing 5.9 Coding on the Method INIT

2. Edit the method RESET. This method is called when a new document is entered or loaded. Initialize the instance attributes of your class (see Listing 5.10).

```
METHOD if_ex_mb_migo_badi~reset.
* Initialize instance attributes
  CLEAR:  gv_lineid,
          gv_action,
          gv_refdoc,
          gs_header.
  REFRESH gt_item.
ENDMETHOD.
```

Listing 5.10 Coding on the Method RESET

3. Switch to the method MODE_SET. In this method, you obtain the chosen action (parameter I_ACTION; refer to Table 5.8) and the chosen reference document type (parameter I_REFDOC; refer to Table 5.9).

   Your custom fields are only integrated when this allows the current action. You've already prepared the function module ZMB_MIGO_SETSTATUS, which you now call. Set the parameter I_OUTPUTONLY when the action A04 (Display) or A03 (Cancellation) is selected. Also note the current action in the attribute GV_ACTION and the reference document type in the attribute GV_REFDOC to be used later (see Listing 5.11).

```
METHOD if_ex_mb_migo_badi~mode_set.
* Local data declarations
  DATA lv_outputonly TYPE c.

* No input for action A04 or A03
  IF    i_action = 'A04' " Anzeige
     OR i_action = 'A03'. "Storno
    lv_outputonly = 'X'.
  ENDIF.

* Set current status
  CALL FUNCTION 'ZMB_MIGO_SETSTATUS'
    EXPORTING
      i_outputonly = lv_outputonly.

* Hold action
  gv_action = i_action.
* Hold reference document type
  gv_refdoc = i_refdoc.

ENDMETHOD.
```

Listing 5.11 Coding on the Method MODE_SET

5.1.5 Activation of Custom Header Data

Now let's move on to the relatively simple header data. At this point, your dynpro is displayed as a tabstrip, and data is exchanged between the BAdI and your dynpro. The reading of custom data is already implemented here when a posted document is open. You post the header data later, together with the item data.

1. Switch to the method `PBO_HEADER`. Start with a mandatory check: For technical reasons, the methods from all implementations are always called in an undefined sequence when one BAdI method is called to a method with several active implementations. At the same time, the method `PBO_HEADER` is called once for each of the five possible tabstrips. All implemented methods are therefore started each time. For this reason, you're informed of the class ID in the input parameter `I_CLASS_ID` (you've registered this in the method `INIT`; see Figure 5.5) that is associated with this call. You must make sure that you check whether the content of `I_CLASS_ID` corresponds to the class ID used in this implementation from constant `GC_CLASS`. Otherwise, unforeseeable side effects may arise.

2. The dynpro is registered next by transferring E_CPROG to the parameters and E_DYNNR to the program name and the dynpro number. In the parameter E_HEADING, specify the name of the tabstrip. In the example given, the dynpro 0100 is used in the program SAPLZMB_MIGO.

   As an example, the tabstrip is only displayed when the operation refers to a purchase order document. You can check via the attribute GV_REFDOC whether you have filled this in method MODE_SET. The reference key for purchase orders is R01 (refer to Table 5.9).

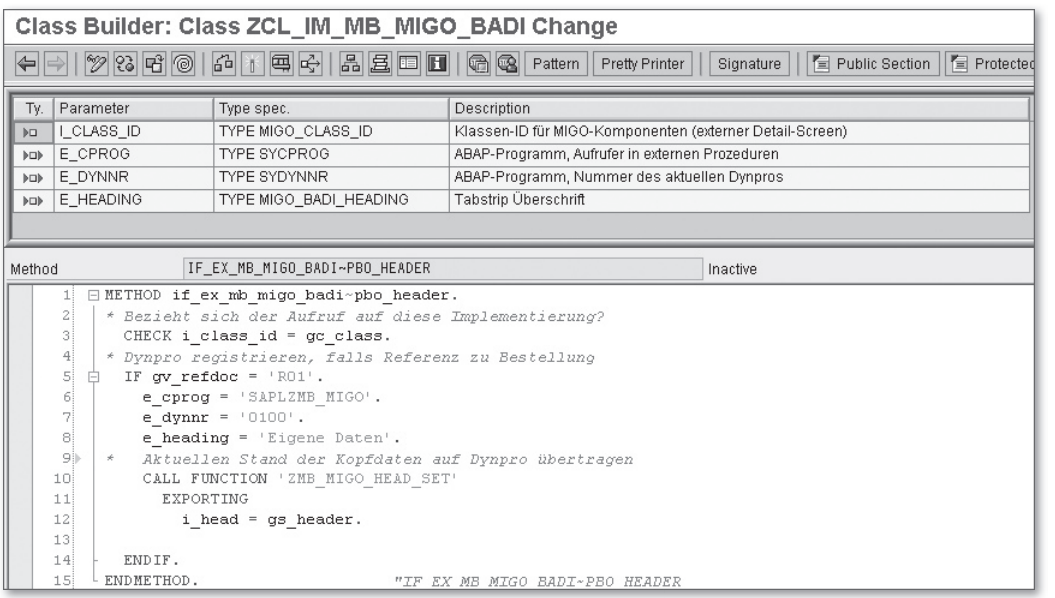


Figure 5.5 Interface of the Method PBO_HEADER

3. Finally, you still need to call the function module ZMB_MIGO_HEAD_SET and transfer the current content of the header data to the dynpro. The current status can always be found in the attribute GS_HEADER (see Listing 5.12).

```
METHOD if_ex_mb_migo_badi~pbo_header.
* Does the call refer to this implmentation?
  CHECK i_class_id = gc_class.

* Register dynpro if reference to purchase order
  IF gv_refdoc = 'R01'.
    e_cprog   = 'SAPLZMB_MIGO'.
```

```
    e_dynnr   = '0100'.
    e_heading = 'Eigene Daten'.

*   Transfer current status of the header data to dynpro
    CALL FUNCTION 'ZMB_MIGO_HEAD_SET'
      EXPORTING
        i_head = gs_header.

  ENDIF.
ENDMETHOD.
```

Listing 5.12 Coding on the Method PBO_HEADER

4. Now process the method PAI_HEADER. You only need to write the data from the dynpro back to the attribute GS_HEADER at this point. You do this by calling the function module ZMB_MIGO_HEAD_GET (see Listing 5.13).

```
METHOD if_ex_mb_migo_badi~pai_header.
* Retrieve data from the dynpro
  CALL FUNCTION 'ZMB_MIGO_HEAD_GET'
    IMPORTING
      e_head = gs_header.

ENDMETHOD.
```

Listing 5.13 Coding on the Method PAI_HEADER

5. You must also read the custom data from the database if required, and transfer it to the attribute GS_HEADER. You carry this out in method STATUS_AND_HEADER. Rereading the data is only necessary when an already-posted document is displayed or canceled. Therefore, you can check the action (attribute GV_ACTION) again here.

   You read the data from the table ZMB_MIGOHEAD by using key fields MBLNR (material document number) and MJAHR (posting year) that are available in the IS_GOHEAD parameter (see Listing 5.14).

```
METHOD if_ex_mb_migo_badi~status_and_header.
* If an already posted document is to be displayed
* or canceled, read custom fields
* also from database.

  IF   gv_action = 'A04'  " Anzeige
    OR gv_action = 'A03'. " Storno
```

```
*   Read data from database
    SELECT SINGLE *
       FROM ZMB_MIGOHEAD
       INTO gs_header
       WHERE mblnr = is_gohead-mblnr
       AND   mjahr = is_gohead-mjahr.
  ENDIF.
ENDMETHOD.
```

Listing 5.14 Coding on the Method STATUS_AND_HEADER

Note Data

You can also set the parameter E_HOLD_DATA_DISABLE as an option in the method STATUS_AND_HEADER to prohibit the function Hold. If you continue to allow the noting of material documents, you should program the methods HOLD_DATA_SAVE, HOLD_DATA_LOAD, and HOLD_DATA_DELETE so that your custom data can also be noted. You can obtain further information on this in Section 5.2.1, Noting Custom Data.

5.1.6 Activation of Custom Item Data

The actual activation of the item tabstrip works in a similar way to that of the header tabstrip. The communication with the dynpro and the management of the item data is slightly more time-consuming. Ultimately, there are several items in a document that must always be correctly assigned.

1. Start with the method PBO_DETAIL to register the item tabstrip and prepare the data. It's also important to check the class ID just as you did before with the header data.
2. In the parameter I_LINE_ID, you can find the line number that is being processed. All subsequent actions are only carried out when the parameter has a content unequal to zero.

   You need to also note this line number in the attribute GV_LINEID so that you can check later which line is actually being displayed in the dynpro.
3. There are also three parameters here, E_CPROG, E_DYNNR, and E_HEADING, to specify the program name, the dynpro number, and the caption of the tabstrip. The item tabstrip always appears in the example regardless of the reference document type.

4. You must then read the current line according to parameter I_LINE_ID from your internal table defined in the attributes GT_ITEM and transfer it to your function group by calling the function module ZMB_MIGO_ITEM_SET. Make sure that you copy the data via MOVECORRESPONDING into a structure suitable for the function module because your table with additional line ID has been defined in the attributes (see Listing 5.15).

```
METHOD if_ex_mb_migo_badi~pbo_detail.
* Local declarations
  DATA ls_item TYPE ts_migoitem.
  DATA ls_dynpro TYPE zmb_migoitem.

* Does the call refer to this implementation?
  CHECK i_class_id = gc_class.

* Has a line been set?
  CHECK i_line_id <> 0.

* Hold line
  gv_lineid = i_line_id.

* Register dynpro
  e_cprog   = 'SAPLZMB_MIGO'.
  e_dynnr   = '0200'.
  e_heading = 'Eigene Daten'.

* Read line from internal table
  READ TABLE gt_item
       INTO ls_item
       WITH KEY line_id = i_line_id.

* Copy necessary fields
  MOVE-CORRESPONDING ls_item TO ls_dynpro.
* Prepare line in dynpro
  CALL FUNCTION ZMB_MIGO_ITEM_SET'
    EXPORTING
      i_item = ls_dynpro.

ENDMETHOD.
```

Listing 5.15 Coding on the Method PBO_DETAIL

5. Switch to the method PAI_DETAIL in which you retrieve your data from the dynpro, and check whether the content has changed. If it has, set the flag E_FORCE_CHANGE through which the method LINE_MODIFY is triggered. Then carry out the actual handling of the data. The advantage to this is that the complete item management can be found in one central position (see Listing 5.16).

```
METHOD if_ex_mb_migo_badi~pai_detail.
* Local declarations
  DATA: ls_olddata TYPE zmb_migoitem,
        ls_newdata TYPE zmb_migoitem,
        ls_item TYPE ts_migoitem.


* Has a line been set?
  CHECK i_line_id <> 0.

* Retrieve old status from internal table
  READ TABLE gt_item
       INTO ls_item
       WITH KEY line_id = i_line_id.
  MOVE-CORRESPONDING ls_item TO ls_olddata.

* Retrieve new status from dynpro
  CALL FUNCTION 'ZMB_MIGO_ITEM_GET'
    IMPORTING
      e_item = ls_newdata.

* Check whether the data has been changed
  IF ls_olddata <> ls_newdata.
*   Trigger execution of LINE_MODIFY
    e_force_change = 'X'.
  ENDIF.

ENDMETHOD.
```

Listing 5.16 Coding on the Method PAI_DETAIL

6. Now it's time for the method LINE_MODIFY. As mentioned earlier, in this method, you manage not only possible changes to existing lines but also the insertion of new document lines in one central position.

As a parameter, you obtain the number of the current line in I_LINE_ID. With this line number, you must first of all check which case is actually present. Read the table GT_ITEM using this key. If the line already exists, possible changes must be transferred. If this line is not yet contained in GT_ITEM, you need to initialize the line and add the table. If the document is currently being read, at this point also read your custom data from the database.

Take a look at the case of a change. To ensure that the line being handled in the BAdI is also the line that is currently in the dynpro, compare the parameter I_LINE_ID with the previously noted line in the attribute GV_LINEID. You carry out the following steps only when the comparison is successful:

1. Retrieve the data from the dynpro by calling the function module ZMB_MIGO_ITEM_GET again. Copy the data received into a work structure that fits Table GT_ITEM (data type TS_MIGOITEM), and enhance the line at the current LINE_ID. Then write the changes back to Table GT_ITEM.
2. If the line doesn't yet exist, you must first check whether it's a line that has already been posted. In this case, the fields CS_GOITEM-MBLNR (material document number), CS_GOITEM-MJAHR (posting year), and CS_GOITEM-ZEILE (document item) are filled. Read your data from Table ZMB_MIGOITEM using these fields as a key, and copy these again into a suitable work structure for Table GT_ITEM.

   If the line doesn't yet exist in the database, leave the work structure empty, apart from the line LINE_ID, which you still need to enhance in both cases. Then write the new line in Table GT_ITEM (see Listing 5.17).

```
METHOD if_ex_mb_migo_badi~line_modify.
* Local declarations
  DATA lv_tabix TYPE sy-tabix.
  DATA ls_item TYPE ts_migoitem.
  DATA ls_newdata TYPE zmb_migoitem.

* Does the line already exist?
  READ TABLE gt_item INTO ls_item
    WITH KEY line_id = i_line_id.
  lv_tabix = sy-tabix.

  IF sy-subrc = 0.
*   Line exists already, does change
*   correspond to line in the BAdI of the dynpro line?
    CHECK i_line_id = gv_lineid.
```

```
*   Retrieve data from dynpro
    CALL FUNCTION 'ZMB_MIGO_ITEM_GET'
      IMPORTING
        e_item = ls_newdata.

*   Format changes
    MOVE-CORRESPONDING ls_newdata TO ls_item.
    ls_item-line_id = i_line_id.

*   Write back in table gt_item
    MODIFY gt_item FROM ls_item INDEX lv_tabix.
  ELSE.
*   Line doesn't yet exist, insert
    IF cs_goitem-mblnr IS NOT INITIAL AND
       cs_goitem-mjahr IS NOT INITIAL AND
       cs_goitem-zeile IS NOT INITIAL.
*     Line refers to an existing
*     material document, retrieve data from database
      SELECT SINGLE * FROM zmb_migoitem
                      INTO ls_newdata
                      WHERE mblnr = cs_goitem-mblnr AND
                            mjahr = cs_goitem-mjahr AND
                            zeile = cs_goitem-zeile.
      IF sy-subrc = 0.
*       Copy data
        MOVE-CORRESPONDING ls_newdata TO ls_item.
      ENDIF.
    ENDIF.
*   Build line with line ID and copy
*   to internal table
    ls_item-line_id = i_line_id.
    APPEND ls_item TO gt_item.
  ENDIF.

ENDMETHOD.
```

Listing 5.17 Coding on the Method LINE_MODIFY

3. Now you need to program the method `LINE_DELETE`. The method is called when a document item is deleted. In this case, you must also delete the associated custom data from Table `GT_ITEM`. Parameter `I_LINE_ID` is provided for you to identify the suitable line (see Listing 5.18).

```
METHOD if_ex_mb_migo_badi~line_delete.
* Delete line
  DELETE gt_item WHERE line_id = i_line_id.
ENDMETHOD.
```

Listing 5.18 Coding on the Method LINE_DELETE

5.1.7 Updating the Data

Now that you've fully programmed the internal handling of the data, you must ensure that your data is also posted when the material document is updated. To accomplish this, the method POST_DOCUMENT runs while the data is being posted.

1. Switch to the method POST_DOCUMENT. Before you can call your update modules, you must prepare the data. The structure GS_HEADER, which has been defined in the attributes, must still be enhanced in the document number. You obtain this structure via the parameter IS_MKPF. You can copy the values via a simple MOVE-CORRESPONDING.

2. Regarding item data, so far you have managed the data via the LINE_ID, that is, the internal line number. As you know, only the items for which the user has selected the OK field in Transaction MIGO are updated while being posted. Therefore, the table to be updated, MSEG, which is transferred to you as the parameter IT_MSEG, can have a different line numbering. The item number here is specified via the field ZEILE, and you must now convert your data to this line number. However, because Table IT_MSEG also contains the original LINE_ID besides the line number, this isn't a problem.

   Simply process all your custom item data from the internal Table GT_ITEM in a loop, and read the associated line from Table IT_MSEG using the field LINE_ID. You can then fill a structure for Table ZMB_MIGOITEM from both data structures. From the structure for Table IT_MSEG, copy the key fields, and from the structure on GT_ITEM, copy the custom fields. Add the result of a local internal table that also uses the type ZMB_MIGOITEM.

3. You now only need to call the function module ZMB_MIGO_POST and transfer the formatted header data and item data. Don't forget the addition IN UPDATE TASK, so that the function module in the updating is called (see Listing 5.19).

```
METHOD if_ex_mb_migo_badi~post_document.
* Local data declarations
  DATA: ls_item TYPE ts_migoitem,
```

```
        ls_mseg TYPE mseg,
        ls_migoitem TYPE zmb_migoitem,
        lt_migoitem TYPE TABLE OF zmb_migoitem.

* Prepare header data
* The key fields (material document number, posting year)
* are copied from the standard data (IS_MKPF)
  MOVE-CORRESPONDING is_mkpf TO gs_header.

* Prepare item data
  LOOP AT gt_item INTO ls_item.
*   Determine corresponding line in IT_MSEG.
*   Conversion between  LINE_ID and ZEILE
*   LINE_ID: internal line ID during entry
*   ZEILE:   Line number in table MSEG
    READ TABLE it_mseg
         INTO ls_mseg
         WITH KEY line_id = ls_item-line_id.
    IF sy-subrc = 0.
      MOVE-CORRESPONDING ls_item TO ls_migoitem.
      MOVE-CORRESPONDING ls_mseg TO ls_migoitem.
      APPEND ls_migoitem TO lt_migoitem.
    ENDIF.
  ENDLOOP.

* Updating of the data
  CALL FUNCTION 'ZMB_MIGO_POST'
    IN UPDATE TASK
    EXPORTING
      i_head  = gs_header
    TABLES
      t_items = lt_migoitem.

ENDMETHOD.
```

Listing 5.19 Coding on the Method POST_DOCUMENT

5.2 Other Functions of the BAdI MB_MIGO_BADI

After you've implemented custom fields via the BAdI `MB_MIGO_BADI` in the previous chapter, you already know all the fundamentals on this BAdI. Using other

methods, you can still implement additional functions, particularly to further extend the functionality behind your custom fields.

5.2.1 Noting Custom Data

If you haven't prohibited the function `Note` in the method `STATUS_AND_HEADER` (see Section 5.1.5, Activation of Custom Header Data), you should ensure that your custom fields are also noted. Because a noted document isn't actually posted yet, it still doesn't have any material document number under which it can be stored. The data are instead clearly identified by a 22-digit GUID (Global Unique Identifier). Standard data are stored under this key in Table `MMIM_PRED`.

Three methods are available for storing your custom data: `HOLD_DATA_SAVE`, `HOLD_DATA_LOAD`, and `HOLD_DATA_DELETE`. The GUID, which has also been used for the standard fields, will transfer each of these methods as input parameters. Whether you now store your data, that is, the header data that are in the structure `GS_HEADER` and the item data in internal Table `GT_ITEM`, is up to you.

You have several options for storing the data:

- You can copy Tables `ZMB_MIGOHEAD` and `ZMB_MIGOITEM` and provide them with a GUID field as the key field. This makes saving somewhat more time-consuming, however, and when you add another custom field later to the original tables, you must also implement such enhancements in the `Hold` function.
- One alternative allows the comfortable, dynamic storage of both objects without any additional customization effort. You have the option in ABAP to serialize complex variables such as structures or internal tables; that is, to convert them into a character string. This string must have a very special data type. Use the type `XSTRING`in ABAP and type `RAWSTRING` in the ABAP Dictionary. You then convert the data via the command `EXPORT ... TO DATA BUFFER`. You can store the result of the conversion in a field of a database line, regardless of whether or not this object has displayed an internal table with many lines beforehand. If the data is subsequently retrieved, using the command `IMPORT ... FROM DATA BUFFER` performs a reconversion in the actual object.

Based on the example of custom fields in Section 5.1, Custom Fields in Transaction MIGO, the following example shows you how simply you can temporarily store your data:

1. Your header and item data must be stored in a custom table. The data is saved under a GUID, which you obtain in the methods. You also need a field of the type RAWSTRING for the header data and the item data.

   Navigate to the ABAP Dictionary (Transaction SE11), and create a new table. Name this table ZMB_MIGOHOLD, for example.

2. Maintain the fields according to Table 5.10. The data element ZMB_RAW doesn't yet exist. You can create this by double-clicking. Don't use any domains to generate the data element, but choose the Integrated Type option in the Data Type tabstrip, and enter RAWSTRING here. Activate your data element, and then navigate back to the table.

Field Name	Key Field	Data Element
MANDT	Yes	MANDT
GUID	Yes	GUID
HHEAD		ZMB_RAW (Definition: Integrated type RAWSTRING)
HITEMS		ZMB_RAW

Table 5.10 Structure of the Table ZMB_MIGOHOLD

3. Maintain the technical settings of the table, and then activate the table.
4. Switch to your BAdI implementation, and navigate to the method HOLD_DATA_SAVE. Create a local structure for Table ZMB_MIGOHOLD, and fill the GUID field from the input parameter I_GUID.
5. Fill the fields HHEAD and HITEMS from your attributes S_HEADER and GT_ITEM via the command EXPORT, which has the following structure:

```
EXPORT <id> FROM <variable> TO DATA BUFFER
  <ziel> COMPRESSION ON.
```

   <id> is any ID, <variable> is the source object that is to be converted, and <target> is the target variable of the type RAWSTRING. The addition, COMPRESSION ON, is optional and compresses the data. This saves space in the database. The subsequent IMPORT automatically recognizes whether the data has been compressed, and extracts the data accordingly.

6. Finally, save your local structure via INSERT in the database (see Listing 5.20).

```
METHOD if_ex_mb_migo_badi~hold_data_save.
* Local work structure for ZMB_MIGOHOLD
  DATA ls_migohold TYPE zmb_migohold.

* Copy GUID
  ls_migohold-guid = i_guid.

* Convert header data in RAWSTRING/XSTRING
  EXPORT header FROM gs_header TO DATA BUFFER
     ls_migohold-hhead COMPRESSION ON.

* Convert item data in RAWSTRING/XSTRING
  EXPORT item FROM gt_item TO DATA BUFFER
      ls_migohold-hitems  COMPRESSION ON.

* Store data in database
  INSERT into zmb_migohold values ls_migohold.

ENDMETHOD.
```

Listing 5.20 Coding on the Method HOLD_DATA_SAVE

Now focus on the method HOLD_DATA_LOAD, which is called when a held document is loaded back.

1. Here you must load back the data from Table ZMB_MIGOHOLD using the retransferred GUID, and fill the attributes GS_HEADER and GT_ITEM from this.
2. Create a new local work structure for Table ZB_MIGOHOLD, and read the data via SELECT using the key in parameter I_GUID from your table.
3. You perform the reconversion from the fields HHEAD and HITEMS via the IMPORT function with the following structure:

   ```
   IMPORT <id> TO <variable> FROM DATA BUFFER <quelle>.
   ```

   <id> stands for the ID that you've used with regard to the EXPORT; <variable> is the target variable in which the result is to be written, that is, GS_HEADER or GT_ITEM; and <source> is the RAWSTRING field from your work structure for ZMB_MIGOHOLD. You can find an example on this method in Listing 5.21.

```
METHOD if_ex_mb_migo_badi~hold_data_load.
* Local work structure for ZMB_MIGOHOLD
  DATA ls_migohold TYPE zmb_migohold.

* Read held data using I_GUID
```

```
  SELECT SINGLE * FROM zmb_migohold
         INTO ls_migohold
         WHERE guid = i_guid.

  IF sy-subrc = 0.
*   Convert header data in original object
    IMPORT header TO gs_header FROM DATA BUFFER
       ls_migohold-hhead.

*   Convert item data in original object
    IMPORT item TO gt_item FROM DATA BUFFER
       ls_migohold-hitems.
  ENDIF.
ENDMETHOD.
```

Listing 5.21 Coding on the Method HOLD_DATA_LOAD

4. If you delete the marked data, you must also remove your noted data. To do this, the method `HOLD_DATA_DELETE` is used. You also get the GUID as an input parameter again. Simply delete the associated record in Table `ZMB_MIGOHOLD` (see Listing 5.22).

```
METHOD if_ex_mb_migo_badi~hold_data_delete.
* Delete held data
  DELETE FROM zmb_migohold WHERE guid = i_guid.
ENDMETHOD.
```

Listing 5.22 Coding on the Method HOLD_DATA_DELETE

Changes to the Data Structure

As you've seen, you can easily convert ABAP structures in a `RAWSTRING` using the commands `EXPORT` and `IMPORT`, and copy these back to a structure. However, what happens when you change such a packed structure later on?

For example, if you add another field to Table `ZMB_MIGOHEAD` for the header data, data could still exist in the `HHEAD` field of the `ZMB_MIGOHOLD` table, which does not contain this new field. However, this generally doesn't constitute any problem. The command `IMPORT` copies the suitable part back to the work structure while reading the noted data, and the new field simply remains empty.

Vice versa is a bit more difficult. If you delete the second field again at a later stage and still contain noted data in the `RAWSTRING HHEAD` of this, you obtain a short dump when you `IMPORT`. You can avoid this dump by using the `ACCEPTING TRUNCATION` addition:

```
IMPORT <id> TO <variable> FROM DATA BUFFER <source> ACCEPTING TRUNCATION.
```

For major structure changes, you should delete the content of Table ZMB_MIGOHOLD as soon as you transport your changes to the production system.

5.2.2 Input Checks in Transaction MIGO

Two more methods are provided for checking your custom data: CHECK_HEADER and CHECK_ITEM. These methods are only carried out when the user chooses the Check or Post function. Follow the example below to activate the checks.

You need to carry out the checks on the attributes of your class in both cases. To do this, you obtain the input parameter I_LINE_ID for the items. If a warning or error message is displayed, this isn't allowed to happen directly via the MESSAGE command. Instead, you need to fill a return-table. You might know about this from using the BAPI function modules. The name of this return-table is in both methods: ET_BAPIRET2. Your messages are therefore displayed in the standard error log (see Figure 5.6) of Transaction MIGO, and the item number for item-related messages, to which the message refers, is displayed by default.

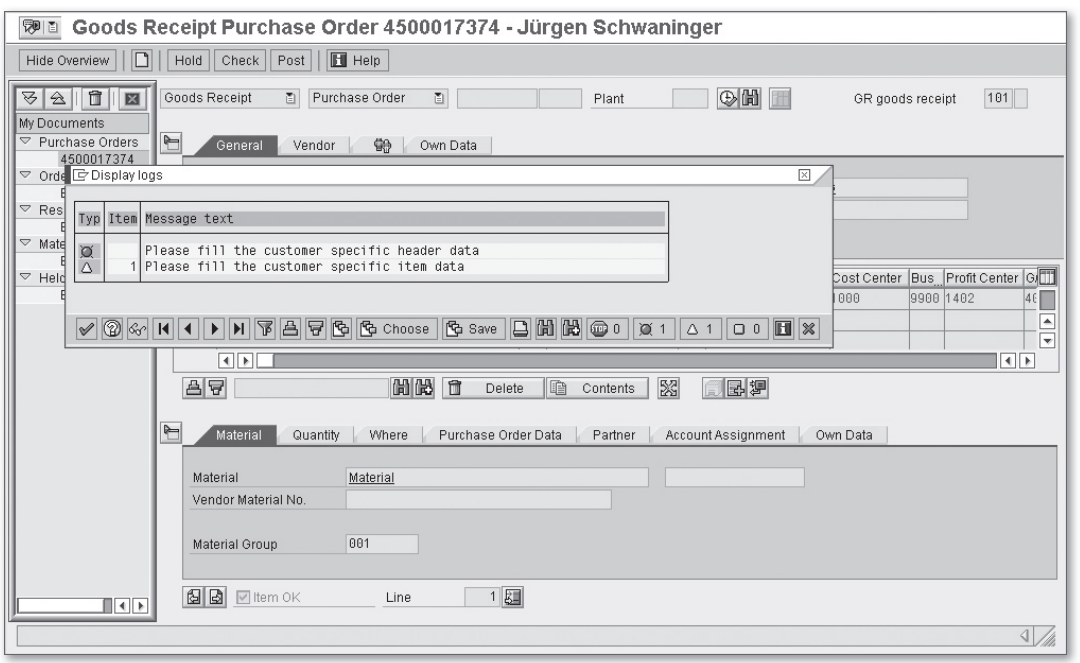


Figure 5.6 Custom Messages in the Error Log

1. Switch to the method CHECK_HEADER. For the return of messages, you must first define a work structure for the return-table by using the type BAPIRET2. This type is the line type that has been used to define internal Table ET_BAPIRET2. This type therefore contains the fields required for a message (see Table 5.11).
2. Now check whether the ZK_FELD1 field is filled from the attribute GS_HEADER. If this isn't the case, an error message appears.
3. To display the message, fill the fields of your work structure according to Table 5.11. Then copy the structure to internal Table ET_BAPIRET2, which is defined in the interface of the method (see Listing 5.23).

Field	Description
TYPE	Type of message, for example, 'E' for error messages or 'W' for warning messages
ID	Message class
NUMBER	Message number
MESSAGE_V1 – MESSAGE_V4	Optional message variables 1-4 that can be included

Table 5.11 Fields of the Return-Table

```
METHOD if_ex_mb_migo_badi~check_header.
* Local declaration on the return table
  DATA: ls_bapiret TYPE bapiret2.

* Header field must be filled!
  IF gs_header-zk_feld1 IS INITIAL.
*   Configure error message
    ls_bapiret-type       = 'E'.
    ls_bapiret-id         = 'ZMB'.
    ls_bapiret-number     = '050'.
    APPEND ls_bapiret TO et_bapiret2.
  ENDIF.

ENDMETHOD.
```

Listing 5.23 Coding on the Method CHECK_HEADER

4. Switch to the method CHECK_ITEM. You basically proceed here as you did with checking the header data. However, you must also define a local work structure

for your items, and read the suitable item from Table GT_ITEM using the input parameter I_LINE_ID.

5. You can then carry out your check again. The user should also fill the field ZP_FELD1. However, only a warning message is displayed if the field is empty (see Listing 5.24).

```
METHOD if_ex_mb_migo_badi~check_item.
* Local declarations
  DATA: ls_item TYPE ts_migoitem,
        ls_bapiret TYPE bapiret2.

* Read item from GT_ITEM
  READ TABLE gt_item INTO ls_item
       WITH KEY line_id = i_line_id.

  IF ls_item-zp_feld1 IS INITIAL.
*   Configure warning message
    ls_bapiret-type       = 'W'.
    ls_bapiret-id         = 'ZMB'.
    ls_bapiret-number     = '051'.
    APPEND ls_bapiret TO et_bapiret2.
  ENDIF.
ENDMETHOD.
```

Listing 5.24 Coding on the Method CHECK_ITEM

5.3 Checking and Prepopulating Standard Fields

Standard fields aren't provided in the check methods of the BAdI MB_MIGO_BADI. However, you can use the BAdI MB_MIGO_ITEM_BADI to also check these fields. Furthermore, in this BAdI, you can simply prepopulate the storage location or the item text. If other fields are prepopulated, you can take another look at the method LINE_MODIFY in the BAdI MB_MIGO_BADI, which provides more options in this context.

5.3.1 Prepopulation of Storage Location and Text

The only method of the BAdI MB_MIGO_ITEM_BADI, ITEM_MODIFY is called when new items are added or the user chooses the Check or Post function. When you fill the

export parameter E_STGE_LOC (storage location) or E_ITEM_TEXT (item text) in the method, these values are set for all new additional items.

5.3.2 Checking the Standard Fields

If the user chooses the Check or Post functions, you can carry out custom checks and prevent posting if applicable. For this, the header data in parameter IS_GOHEAD and the item data table IS_GOITEM are provided.

If a message is displayed, you also need a work structure for a return-table with the type BAPIRET2. Fill the work structure according to Table 5.11 in Section 5.2.2, Input Checks in Transaction MIGO, and then append the message to internal Table ET_RETURN, which is also defined in the interface. Because the method ITEM_MODIFY could be called repeatedly, and Table ET_RETURN is already filled by the previous call, you should first delete the content; otherwise, the same message could possibly appear twice in the message log (see Listing 5.25).

```
METHOD if_ex_mb_migo_item_badi~item_modify.
* Local work structure for the return- table
  DATA: ls_bapiret TYPE bapiret2.

* For plant 1000 storage location 0001 is always
* to be suggested.
  IF is_goitem-werks = '1000'.
    e_stge_loc = '0001'.
  ENDIF.

* Reset return table
  REFRESH et_return.
* Header text must be populated
  IF is_gohead-bktxt IS INITIAL.
*   Configure message, warning
    ls_bapiret-type       = 'W'.
    ls_bapiret-id         = 'ZMB'.
    ls_bapiret-number     = '060'.
    APPEND ls_bapiret TO et_return.
  ENDIF.

* Item text must be filled
  IF is_goitem-sgtxt IS INITIAL.
*   Configure message, error
```

```
      ls_bapiret-type       = 'E'.
      ls_bapiret-id         = 'ZMB'.
      ls_bapiret-number     = '061'.
      APPEND ls_bapiret TO et_return.
    ENDIF.
  ENDMETHOD.
```

Listing 5.25 Coding on the Method ITEM_MODIFY

5.4 Check of the Earliest Delivery Date

You have the option in the purchase order to specify a delivery date. However, vendors often don't keep to the delivery date and deliver goods too early. When this is better controlled, you can set the message M7 254 ("Earliest Possible Delivery Date is &" as shown in Figure 5.7) as an error message in Customizing. To do this, choose the Materials Management • Inventory Management and Physical Inventory • DEFINE Attributes Of System Messages • Settings for System Messages setting in Transaction SPRO, and define the message M7 254 as type 'E'.

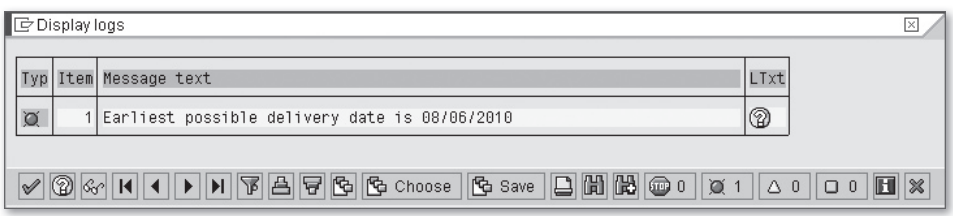


Figure 5.7 Error Message M7 254

As a result, all goods receipts that are supposed to be posted at an earlier date are rejected with the error message. However, you can post the goods receipt in the GR blocked stock (transaction type 103).

In certain circumstances, the settings for such an error message are too strict. With the enhancement MEFLD004 (EXIT_SAPLEINR_004), you can specify the earliest delivery date. Let's assume you have activated the error message. Vendor 1002 is allowed to deliver before the delivery date specified in the purchase order, but not before the purchase order date. The following applies to all other vendors: For all items with material group 00, a delivery of up to seven days prior to the delivery

date is possible. For other goods deliveries, a delivery of up to three days prior to the delivery date is permitted.

1. Create a new project in Transaction CMOD, and include the enhancement MEFLD004.
2. Switch to the user exit EXIT_SAPLEINR_004, and create the include ZXM06U54.
3. In the interface of the exit, the purchase order header (EKKO) and the purchase order item to be checked (EKPO) are transferred. Based on this data, you can overwrite the earliest delivery date (parameter FRLFD). The field FRLFD already contains the delivery date from the purchase order. You can therefore leave the content unchanged, if no special rule applies.

   Check whether the vendor (EKKO-LIFNR) has the number 1002 (look out for leading zeros); if yes, the purchase order date (EKKO-BEDAT) will apply as the earliest delivery date.
4. Check whether the purchase order item has the material group 003; if yes, a goods receipt of up to seven days prior to the delivery date is expected to be possible. Because the field FRLFD already contains the delivery date from the purchase order, you can simply deduct seven days.

   For all other items, the goods receipt can take place three days earlier. Also carry out the respective calculation (see Listing 5.26). Don't forget to activate the project so that your user exit will run.

```
*&---------------------------------------------------------------*
*&  Include           ZXM06U54
*&---------------------------------------------------------------*
*"*"Local interface:
*"  IMPORTING
*"     VALUE(EKKO) TYPE  EKKO
*"     VALUE(EKPO) TYPE  EKPO
*"  CHANGING
*"     VALUE(FRLFD) LIKE  EBEFU-FRLFD
*"---------------------------------------------------------------
* Determine earliest delivery date dynamically

IF ekko-lifnr = '0000001002'.
* Vendor 1002 is allowed to deliver earlier indefinitely,
* however, not before the purchase order date
  frlfd = ekko-bedat.
```

```
ELSE.
* Item for material group 003?
  IF ekpo-matkl = '003'.
*   Goods receipt can take place seven days earlier
    frlfd = frlfd - 7.
  ELSE.
*   Goods receipt can take place three days earlier
    frlfd = frlfd - 3.
  ENDIF.
ENDIF.
```

Listing 5.26 Coding on User-Exit EXIT_SAPLEINR_004

5.5 Tolerance Limits for Scheduling Agreements

Using the current date, the default quantity and tolerance check quantity are determined from the schedule lines when goods receipts are posted according to scheduling agreements. If no tolerances are permitted in the scheduling agreement, and the vendor occasionally delivers a few days too early, you receive an error message in the goods receipt (see Figure 5.8).

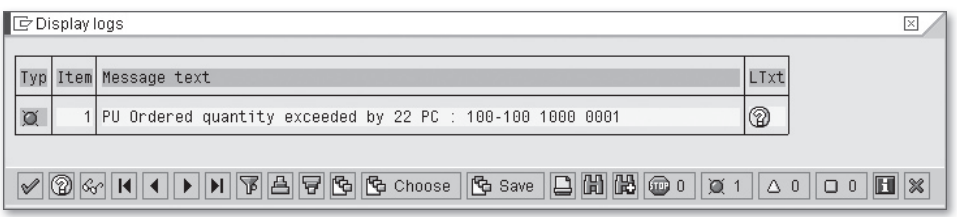


Figure 5.8 Exceeding the Tolerance Levels

If your vendor is allowed to deliver the goods, for example, up to three days earlier, you may use the user exit `EXIT_SAPLEINR_001` of the enhancement `MEVME001` to determine the default quantity and tolerance levels dynamically.

5.5.1 Overwriting Overdelivery Quantity

Let's first take a look at the scheduling agreement schedule in Figure 5.9. Items 1 and 2 have each a scheduled quantity of 10 pieces. A goods receipt of more than 12 pieces has already been posted. Item 1 is therefore complete; for item 2, there is still an open quantity of 8 pieces.

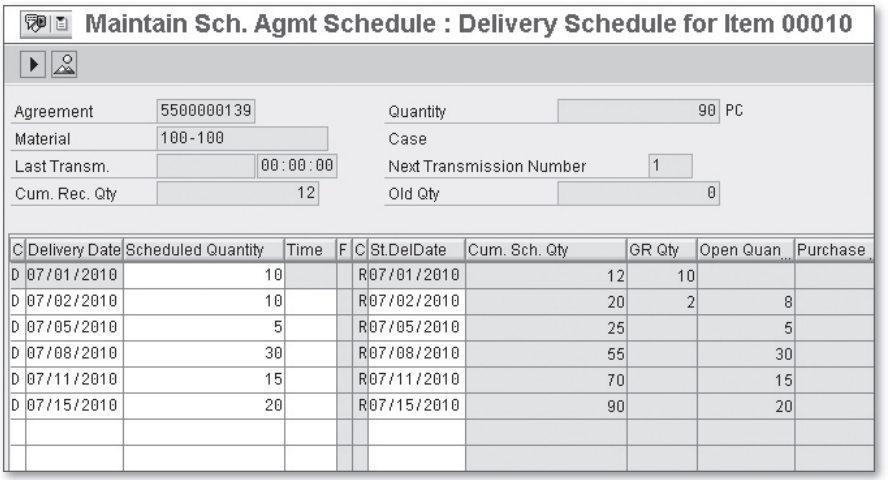


Figure 5.9 Scheduling Agreement Schedule

On July 6, 2010, another goods receipt of more than 43 pieces is now supposed to be posted. The open quantity consists of the 5 pieces scheduled for delivery on July 5, 2010, and also of the 8 pieces still open from the previous items. A total of 13 pieces are open. The goods receipt is therefore rejected due to an overbooking of 30 pieces (43 pc – 13 pc). The vendor has already added the 30 pieces, which are expected on July 8, 2010 — that is, in the future. Because this is within the 3 days for which you allow an earlier delivery, the goods receipt is expected to go through.

1. To resolve this situation, you must program the aforementioned user exit `EXIT_SAPLEINR_001`. Create a project in Transaction CMOD, and add the enhancement `MEVME001`.
2. Switch to the user exit `EXIT_SAPLEINR_001`, and create the include `ZXM06U28`.
3. You get two parameters in the interface: The parameter `POT` contains the item data from the scheduling agreement; the internal table `CETT` contains all schedule lines. Based on this data, you can perform your own calculations and fill the parameters according to Table 5.12. Keep in mind that these parameters in the documentation on the enhancement `MEVME001` are not completely and uniquely defined. The parameters `F3` and `F4` have only been added with SAP Note 737495; the documentation has not been customized, however.

Parameter	Description
F1	WE default quantity
F2	Open quantity on key date
F3	Underdelivery quantity
F4	Overdelivery quantity

Table 5.12 Output Parameter EXIT_SAPLEINR_001

4. Because the overdelivery quantity is customized, you need to determine a new value for parameter F4. The calculation for this works very simply. The parameter contains the complete scheduled quantity, starting from the first item of the scheduling to the last scheduling prior to the key data in total. The system automatically deducts goods receipts already carried out, so you don't need to take this into consideration.

   Calculate in a loop via Table CETT the total of all schedule quantities with one vendor smaller than the date "today + 3 days". You write the result of the calculation in the field F4 (see Listing 5.27).

```
*&---------------------------------------------------------------------*
*&  Include           ZXM06U28
*&---------------------------------------------------------------------*
*"*"Local interface:
*"  IMPORTING
*"     VALUE(POT)
*"  TABLES
*"      CETT STRUCTURE  EKET
*"  CHANGING
*"     VALUE(F1)
*"     VALUE(F2)
*"     VALUE(F3) OPTIONAL
*"     VALUE(F4) OPTIONAL
*"----------------------------------------------------------------------
* Local declarations
DATA lv_menge TYPE eket-menge.
DATA lv_checkdate TYPE eket-eindt.

* Consider schedules until today + 3 days
lv_checkdate = sy-datlo + 3.

* Total all schedule quantities
```

```
LOOP AT cett WHERE eindt <= lv_checkdate.
  lv_menge = lv_menge + cett-menge.
ENDLOOP.

* Transfer of the calculated value as maximum limit
f4 = lv_menge.
```

Listing 5.27 Coding for EXIT_SAPLEINR_001

5.5.2 Overwriting Default Quantity

After the goods receipt that includes 43 pieces has been posted, the new situation in the scheduling agreement is shown in Figure 5.10. As you can see, if there is no longer any quantity open until and including July 8, 2010, the next goods receipt is not expected again until July 11, 2010.

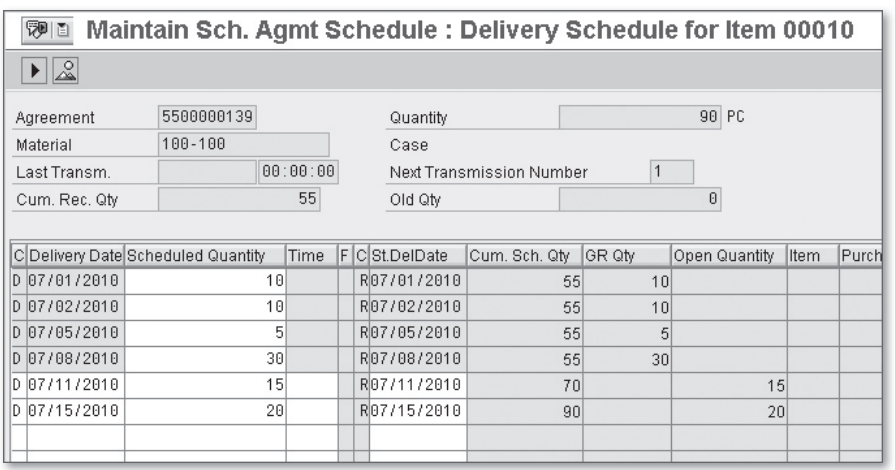


Figure 5.10 Scheduling Agreement Schedule following Goods Receipt

If you now receive a new delivery of more than 15 pieces on July 11, 2010, and these are supposed to be posted as goods receipt, you receive the message "Document contains no selectable item" if the user hasn't activated the PROPOSE ALL ITEMS option. You can react using the user exit: If an open scheduling exists within the next three days, the first open quantity found is suggested. In the example given, this quantity refers to the 15 pieces from July 11, 2010.

1. Navigate again to the user exit `EXIT_SAPLEINR_001`, and enhance the coding for the current case. The field `F2` contains the quantity of all schedulings up to and

including today, minus the goods receipts already carried out. If this value is zero, the line is not suggested. Check the value F2. If no quantity is suggested here, you must calculate a new value.

2. Because the system has already checked all schedulings up to and including the current date, you must check Table CETT again in a loop only from the day "today + 1" until "today + 3". The schedule quantity, as before in the field MENGE, was carried out in the goods receipts in field WEMNG. Calculate the open quantity from this. As soon as you find an open quantity, you can leave the loop.
3. You don't need to assign the result of the calculation to the field F2. In fact, the field F1 again contains the complete schedule quantity from the first day of the scheduling agreement until today. The system deducts the goods receipts already carried out only later from this figure. Simply add the result of your calculation to the field F1. The line with your determined quantity is therefore suggested (see Listing 5.28).

```
...
* Additional determination of the default quantity
* Further data declarations
DATA lv_morgen TYPE eket-eindt.

IF F2 = 0.
* No default quantity available, check whether an open
* schedule exists in the period today + 1 until today + 3
  lv_morgen = sy-datlo + 1.
  clear lv_menge.

  LOOP AT cett WHERE eindt <= lv_checkdate AND
                     eindt >= lv_morgen.
    lv_menge = cett-menge - cett-wemng.
    IF lv_menge > 0.
*     Open schedule available, exit loop
      EXIT.
    ENDIF.
  ENDLOOP.

* Increase expected quantity for found value
  F1 = F1 + lv_menge.

ENDIF.
```

Listing 5.28 Further Coding for EXIT_SAPLEINR_001

4. Based on the situation from Figure 5.10, if you post a goods receipt on July 10, 2010, an item with a quantity of 15 pieces is suggested to you, which is the value of the schedule of July 11, 2010.

5.6 Enhancement of Reservations

For reservations, you can't use Transaction MIGO. As previously stated, reservations are created or changed in Transactions MB21 and MB22. Therefore, you can't use any enhancements for reservations, for example, the BAdI `MB_MIGO_BADI`, to prepopulate fields or to carry out input checks. Even here there is some assistance — the BAdI `MB_RESERVATION_BADI`.

This BAdI is essentially suitable for two situations. First, there is the method `DATA_MODIFY`, which is called when an item is entered before the detail screen view (see Figure 5.11) is displayed. You can use this method to prepopulate individual fields with values. The method `DATA_CHECK` is then called. Here you have the option to carry out custom checks and display, if applicable, a warning or error message.

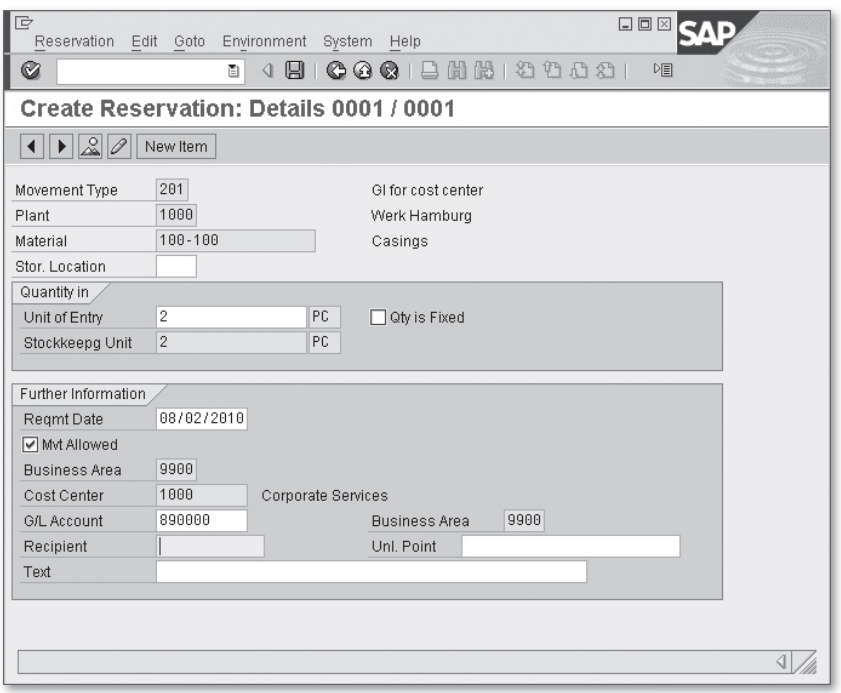


Figure 5.11 Detailed Entry in the Reservation

For both cases, a small example will help you implement these enhancements.

5.6.1 Prepopulating Fields

The prepopulation of fields via the method DATA_MODIFY works relatively simply. However, you can't overwrite all of the fields of the document. All fields of Table 5.13 can basically be overwritten. However, this list is further restricted by the current field status. Fields that haven't been integrated may also not be overwritten by the BAdI method. Therefore, internal Table IT_CHANGEABLE is transferred as a parameter that contains all changeable fields. All changes you make in the BAdI to fields not contained there are subsequently not copied.

Field	Description
RESB-MATNR	Material number
RESB-WERKS	Plant
RESB-LGORT	Storage location
RESB-CHARG	Batch
RESB-SAKNR	GL account number
RESB-ERFMG	Quantity in unit of entry
RESB-ERFMG	Unit of entry
RESB-BDMNG	Requirement quantity
RESB-FMENG	Characteristic: Fixed quantity
RESB-KZEAR	Characteristic: Final issue of the reservation
RESB-XLOEK	Item deleted
RESB-WEMPF	Ship-to party
RESB-ABLAD	Unloading point
RESB-SGTXT	Item text
RESB-BDMNG	Requirement date of the components
RESB-XWAOK	Characteristic: Goods movement to the reservation permitted

Table 5.13 Fields of the Reservation that Can Be Overwritten

Furthermore, the parameter I_NEW_ITEM exists, which means that this item will run the first time in the method. If your prepopulation is only supposed to be a

suggestion that can be overwritten by the user at any time via other values, then it makes sense to check this parameter and to overwrite the field content only when it is run for the first time.

You can find the complete parameter interface of the method DATA_MODIFY in Table 5.14.

Parameter	Description
IT_CHANGEABLE	Table of the changeable fields.
I_NEW_ITEM	Current item is new and is handled the first time in the method.
CS_RKPF	Header data of the reservation. The fields of this structure cannot be changed.
CS_RESB	Structure on an item of the reservation; the fields contained may be overwritten if the field is contained in Table IT_CHANGEABLE.

Table 5.14 Parameters of the Method DATA_MODIFY

In a small example, the field GL account (RESB-SAKNR) is populated for all reservations in plant 1000 with the GL account number 400000.

1. Create an implementation for the BAdI MB_RESERVATION_BADI in Transaction SE19. You can choose, for example, ZMB_RESERVATION_BADI as the name of the implementation.
2. Now navigate to the method DATA_MODIFY. The GL account is only a default value that can be overwritten by the user at any time. For this reason, the method is executed for an item only on the first run (i.e., before the detail screen for an item may have been displayed the first time). Therefore, check the content of I_NEW_ITEM, and exit the method if necessary.
3. Next, you must check whether the field may at all be changed. For this, check whether the field is available in Table IT_CHANGEABLE.

   This internal table is defined via a table type TDTAB_C132 in the ABAP Dictionary; however, this is done without using a structure but instead via an integrated data type. For this reason, the only field of this internal table doesn't have any defined name. However, you can always access the internal table via the field name TABLE_LINE .

4. When the checks are successful, you can overwrite the field value in the structure CS_RESB. A suitable example is given in Listing 5.29.

```
METHOD if_ex_mb_reservation_badi~data_modify .
* Prepopulate GL account
* Prepopulation is only to be a default value, therefore
* only execute item on first run
  CHECK i_new_item IS NOT INITIAL.

* Check whether GL account is changeable
  READ TABLE it_changeable WITH KEY table_line = 'RESB-SAKNR'
  TRANSPORTING NO FIELDS.

* Suggest GL account for plant 1000
  IF sy-subrc = 0 AND
     cs_resb-werks = '1000'.
    cs_resb-saknr = '400000'.
  ENDIF.
ENDMETHOD.
```

Listing 5.29 Coding on the Method DATA_MODIFY

5.6.2 Checking Entries

Custom input checks are also simple to implement. To implement your checks, the header and item data are provided. To trigger a warning or error message, you can work perfectly normally with the MESSAGE command, but you must trigger the exception EXTERNAL_MESSAGE.

You can find the parameter interface of the method DATA_CHECK in Table 5.15.

Parameter	Description
IT_CHANGEABLE	This internal table again contains the list of current changeable fields. This doesn't usually play any major role for this method.
IS_RKPF	The header data of the reservation.
IS_RESB	Structure on an item of the reservation.
I_NEW_ITEM	Current item is new and is handled the first time in the method.

Table 5.15 Parameters of the Method DATA_CHECK

As an example, the fields are supposed to be checked for the ship-to party (RESB-WEMPF) and the unloading point (RESB-ABLAD). The specification of a ship-to party is mandatory. If this specification is missing, it must be confirmed by an error message. If there is no entry in the unloading point field, this is indicated by a warning message.

1. Navigate to the implementation of the BAdIs that you created in the previous example on method DATA_MODIFY. Then, switch to the method DATA_CHECK.
2. The method is only called for an item when the item is new or has been changed. Therefore, the evaluation of the parameter I_NEW_ITEM isn't absolutely necessary. Check whether the ship-to party is filled, and display an error message. To trigger the exception, you need the addition RAISING EXTERNAL_MESSAGE.
3. So that the message only appears for the unloading point, which is just a warning when the item is entered, but not in the case of each subsequent change, you can at this point also check the parameter I_NEW_ITEM. If the parameter is set, and the unloading point hasn't been completed, a warning message is displayed. You must also use the addition RAISING EXTERNAL_MESSAGE with regard to warning messages.

   By displaying the warning message, you can always navigate to the detail screen in which the unloading point can be entered. This is usually only shown when an important piece of information is missing. You can find the relevant example in Listing 5.30.

```
METHOD if_ex_mb_reservation_badi~data_check .
* The ship-to party must be filled.
  IF is_resb-wempf is initial.
*   Message: “Please specify a ship-to party“
    MESSAGE  e100(ZMM) RAISING external_message.
  ENDIF.

* The unloading point should be filled
* Check only on initial run of item
  IF i_new_item is not initial AND
     is_resb-ablad IS INITIAL.
*   Message: “Unloading point was not entered!“
    MESSAGE  w101(ZMM) RAISING external_message.
  ENDIF.
ENDMETHOD.
```

Listing 5.30 Coding on the Method DATA_CHECK

Index

A

B

C

D

E

F

G

H

I

L

M

N

O

P

Q

R

S

T

U

V

W

X

Y