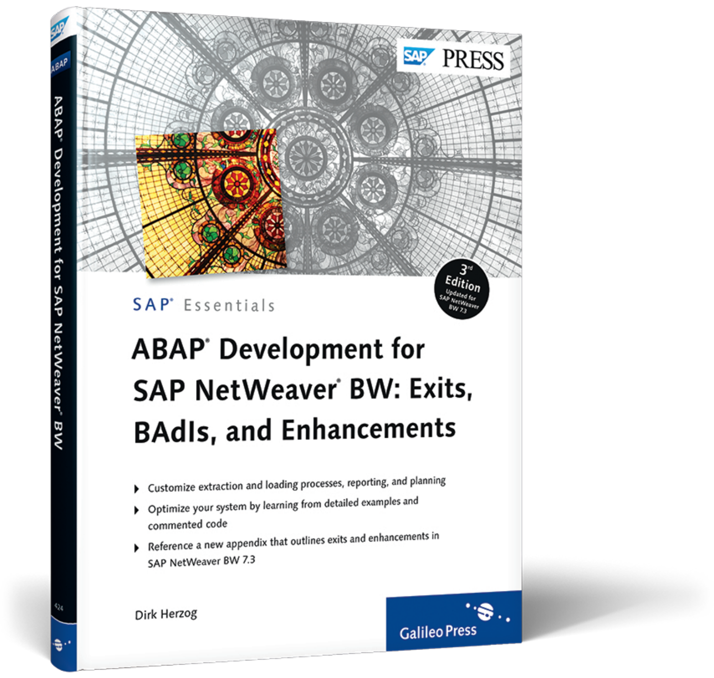
Dirk Herzog

ABAP® Development for SAP NetWeaver® BW

Exits, BAdIs, and Enhancements


SAP PRESS
3rd Edition Updated for SAP NetWeaver BW 7.3
SAP Essentials
ABAP Development for SAP NetWeaver BW: Exits, BAdIs, and Enhancements
Customize extraction and loading processes, reporting, and planning
Optimize your system by learning from detailed examples and commented code
Reference a new appendix that outlines exits and enhancements in SAP NetWeaver BW 7.3
Dirk Herzog
Galileo Press

Galileo Press

Bonn • Boston

Contents at a Glance

Contents

4 User Exits and BAdIs in Reporting

The exits described so far are very efficient and enable you to format the data so that it meets many typical reporting requirements. However, it's often necessary to make the reports more flexible by using variables. For example, no SAP NetWeaver BW consultant would recommend that you define separate reports for each cost center within your company. Instead, he would ask you to define the cost center as an input variable.

On the other hand, this flexibility should not be attained at the cost of the users by overloading them with a multitude of variables. For this reason, it's useful for the user to get suggestions from the program regarding the cost center(s) he will probably want to see in the report, depending on his position and area of responsibility.

The following sections describe how you can derive variable values and set default values by employing the right programming technique. You then will be introduced to virtual characteristics and key figures as well as to the topic of VirtualProviders. Whereas the variables make the value selection at the time that a report is analyzed more flexible, both virtual InfoObjects and VirtualProviders go one step further and enable you to customize the data at the time of reporting. As you might guess, the implementation of InfoObjects and VirtualProviders is more complex. Finally, the chapter provides a brief description of all other techniques that employ ABAP to customize the reporting process.

4.1 Variable Exit RSR00001

At its core, the user exit `RSR00001` contains function module `EXIT_SAPLRRS0_001`, which performs several different functions related to the input of variables. This exit poses the same problem as the user exit for extractor extension (`RSAP0001`), described in Section 2.3. In a typical BW system, the user exit grows steadily. For this reason, you should make sure from the start that you don't carry out the implementation in include `ZXRSRU01`. Instead, you should separate the implementation from the include (for instance, by deriving the name of a function module from the variable name).

The interface of the user exit contains two central parameters. In addition to the variable name, `I_VAR`, the call step `I_STEP` is very important. This call step is a counter because the user exit is called several times at up to four different places. You can implement different functions at those different points in time. For this reason, we describe the `I_STEP` parameter before we continue explaining the actual implementation.

- `I_STEP = 1`
  The first call of the user exit is used to calculate default values (that is, the values that the system displays as suggestions for users in the variables that are ready for input). For this purpose, the exit is called once per customer-exit-type variable. This is actually one of the biggest weaknesses of the exit. If you want to assign a default value to an existing input variable, you can do so only by creating a new variable of the customer exit type and then making it ready for input. It would be very useful if in this step you were able to assign default values to other variables that are ready for input.

  Even though this step is used for determining default values, you can also fill variables that aren't ready for input. That makes sense, however, only if you need the values later in Step 2 to derive other variables.
- `I_STEP = 2`
  This step isn't called for all variables of the user exit type, only for those not ready for input. In this step, you define the final variable values. To do that, you must call the user exit once for each variable that isn't ready for input. So, you can't overwrite any user input in this step. That's generally impossible because it often leads to misunderstandings by the user. For this reason, you should either forbid any user input or reject the user entry by returning an error message.
- `I_STEP = 3`
  In contrast to the first two steps, Step 3 isn't called for each variable but only once. When it's called, all variable content is transferred for validation purposes. So you can check whether "Period from" is smaller or equal to "Period to," provided that the two periods aren't entered as an interval but in two different variables. This technique is indispensable for a flexible validation, that is, to check whether the input values correspond to each other. However, it obstructs

a clear separation by queries—or at least by projects—because variables should be used across several projects to keep the total number of variables as small as possible. For this reason, you should use a different field to obtain a separation in the implementation.

- `I_STEP = 0`
  Step 0 is used very rarely. It's used for all calls that aren't related to the variables. SAP NetWeaver 7.x and SAP BW 3.x contain three applications in this context:
  - The step is used when you use variables in the InfoPackage for selection purposes.
  - It's used to determine the filter values in the navigation block of the query.
  - Possibly most importantly, it's used to fill variables that are used in authorizations. This is particularly useful if authorizations are to be filled from tables that have been imported into a DataStore object or if the authorizations are to be derived from master data.

  This book focuses on only the last case because all other cases occur only rarely in practice.

4.1.1 Interface of Function Module EXIT_SAPLRSR0_001

Now that we have clarified the meaning of the `I_STEP` parameter, this section describes the complete interface. The header of the function module has the structure shown in Listing 4.1.

```
FUNCTION EXIT_SAPLRRS0_001.
*"-----------------------------------------
*"*"Local interface:
*" IMPORTING
*"  VALUE(I_VNAM)   LIKE  RSZGLOBV-VNAM
*"  VALUE(I_VARTYP) LIKE  RSZGLOBV-VARTYP
*"  VALUE(I_IOBJNM) LIKE  RSZGLOBV-IOBJNM
*"  VALUE(I_S_COB_PRO) TYPE
*"                   RSD_S_COB_PRO
*"  VALUE(I_S_RKB1D)   TYPE
*"                   RSR_S_RKB1D
*"  VALUE(I_PERIV) TYPE
*"                 RRO01_S_RKB1F-PERIV
*"  VALUE(I_T_VAR_RANGE) TYPE
*"                     RRS0_T_VAR_RANGE
*"  VALUE(I_STEP) TYPE  I DEFAULT 0
*" EXPORTING
*"  VALUE(E_T_RANGE) TYPE  RSR_T_RANGESID
*"  VALUE(E_MEEHT) LIKE  RSZGLOBV-MEEHT
*"  VALUE(E_MEFAC) LIKE  RSZGLOBV-MEFAC
*"  VALUE(E_WAERS) LIKE  RSZGLOBV-WAERS
*"  VALUE(E_WHFAC) LIKE  RSZGLOBV-WHFAC
*" CHANGING
*"  VALUE(C_S_CUSTOMER) TYPE
*"              RRO04_S_CUSTOMER OPTIONAL
*"--------------------------------------
```

Listing 4.1 Interface of Function Module EXIT_SAPLRSR0_001

The various parameters have the following meanings:

- `I_VNAM`
  In Steps 0, 1, and 2, this parameter contains the variable name to be calculated.
- `I_VARTYP`
  This parameter specifies which variable type is being used in Steps 1 and 2; that is, it determines whether the return value to be expected is a characteristic value, a text, a formula value, a hierarchy node, or a hierarchy.
- `I_IOBJNM`
  This parameter specifies the InfoObject to which the variable refers.
- `I_S_COB_PRO`
  This parameter contains various pieces of information about the InfoObject to which the variable refers. `ATRNAVFL`, for instance, specifies whether the variable is a navigation attribute, whereas `ATRTIMFL` tells you whether the navigation attribute is time-dependent. Furthermore, the parameter provides information that is needed only in special situations because that information is usually fixed at the time of programming, and it isn't changed afterward. For this reason, the parameter isn't used frequently.
- `I_S_RKB1D`
  This parameter contains information such as the query name, the calling program, and so on. It's particularly important in Step 3 to determine which validations are to be carried out. Important fields for this are `INFOCUBE` (this is the name of the InfoProvider on which the report is run) and `COMPID` (the name of the query that's currently executed).

- `I_PERIV`
  This parameter contains the fiscal year variant, provided it can be uniquely determined in the query. This is important whenever you must determine a period based on a date, for instance, if you want to present the current posting period as a default value. However, because it's rare for several different fiscal year periods to exist in the same system, the parameter isn't used very often.
- `I_T_VAR_RANGE`
  This parameter contains a table, which in turn contains all other variable values. This is particularly useful for Steps 2 and 3.
- `I_STEP`
  The step for variable determination has already been described in detail.
- `E_T_RANGE`
  In this table, the return values in Steps 0, 1, and 2 must be returned. The table is structured like a ranges table; all other fields can be ignored. Depending on the type of variable, you must take into account the following constraints.
  - For characteristic values, the field `LOW` contains the value or the lower value limit (for intervals). For text variables, it contains the text; for hierarchy variables, the hierarchy; for node variables, the node; and for formula variables, the calculation value.
  - The field `HIGH` contains the upper limit of the interval for characteristic value variables for intervals or selection options. For hierarchy-node variables, this field contains the InfoObject of the hierarchy node. This can be omitted if the node is a leaf of the hierarchy. For other variables, it's empty.
  - The field `SIGN` usually contains an `I` (include). The only exception can be selection options, which also allow an `E` (exclude). SAP NetWeaver Business Warehouse also allows an `E` for intervals. However, because this doesn't correspond to the logic of intervals, you should implement such variables as selection options.
  - The `OPT` field is usually filled with `EQ` (equal). For intervals, you also can use `BT` (between) or `NB` (not between). For selection options, you can use all operators that are allowed in ranges tables. Those operators can, for instance, be found in the ABAP documentation for the key word `IF`.
- `E_MEEHT`, `E_MEFAC`, `E_WAERS`, `E_WHFAC`, `C_S_CUSTOMER`
  These parameters are specified in the interface, but they aren't queried.

Based on the different meanings of the individual steps, you can also define the implementation of the exit accordingly. If you want to make a generic definition that calls individual function modules, you can use the implementation shown in Listing 4.2.

```
DATA: l_d_fname(30) TYPE c.
CASE i_step.
  WHEN 1.
    CONCATENATE ‘Z_VAR_PRE_POPUP_’
      i_vnam INTO l_d_fname.
  WHEN 2.
    CONCATENATE ‘Z_VAR_POST_POPUP_’
      i_vnam INTO l_d_fname.
  WHEN 0.
    CONCATENATE ‘Z_VAR_AUTHORITY_’
      i_vnam INTO l_d_fname.
  WHEN 3.
    l_d_fname = ‘Z_VAR_CHECK_VALIDITY’.
ENDCASE.
TRY.
  CALL FUNCTION l_d_fname
    EXPORTING
      I_VNAM        = i_vnam
      I_VARTYP      = i_vartyp
      I_IOBJNM      = i_iobjnm
      I_S_COB_PRO   = i_s_cob_pro
      I_S_RKB1D     = i_s_rkb1d
      I_PERIV       = i_periv
      I_T_VAR_RANGE = i_t_var_range
      I_STEP        = i_step
    IMPORTING
      E_T_RANGE     = e_t_range
      E_MEEHT       = e_meeht
      E_MEFAC       = e_mefac
      E_WAERS       = e_waers
      E_WHFAC       = e_whfac
    CHANGING
      C_S_CUSTOMER = c_s_customer
    EXCEPTIONS
      OTHERS = 1.
  IF SY-SUBRC <> 0.
    MESSAGE ID SY-MSGID TYPE SY-MSGTY
```

```
            NUMBER SY-MSGNO
            WITH
              SY-MSGV1 SY-MSGV2
              SY-MSGV3 SY-MSGV4
            RAISING ERROR_IN_VARIABLE.
  ENDIF.
CATCH CX_SY_DYN_CALL_ILLEGAL_FUNCTION.
* Don't do anything, because no exit was implemented,
* for example, in Step 1 if only Step 2 was implemented.
ENDTRY.
```

Listing 4.2 Sample Implementation for Variable Exit ZXRSRU01

If you take a closer look at the RSVAREXIT_* function modules, you can see that the SAP development implements variables with an SAP exit in almost the same way as shown in Listing 4.2, the only difference being that SAP development doesn't make any distinction based on I_STEP. However, you should make a distinction here because you then can tell the purpose of a function module by its name. If you have already worked in larger BW systems, you may have noticed that the exit ZXRSRU01 is one of the biggest exits in the entire system. All changes to this exit are critical because those changes can affect all reports. You'll know what we're talking about if you've ever transported into a live system an exit that wasn't free of syntax errors. For this reason, you should use the sample implementation right from the start to avoid the annoying problem of a transport separated by content to enable different variable implementations. This frequently doesn't occur until several projects are running concurrently in the live system. Naturally, the question of who will be responsible for function module Z_VAR_CHECK_VALIDITY will arise. But, in case of doubt, you can call a central function module from any project in this function module and collect the check results there.

4.1.2 Implementation for I_STEP = 1

Let's now discuss the actual implementation of the variable logic. Because I_STEP = 1 sets the default values, you can implement only algorithms that don't require any input by the user. There are two main types of algorithms available:

- Date-dependent default value determination
- Table-controlled default value determination

Example 1: Date-Dependent Default Value Assignment

You want to implement a variable called WEEK6F for InfoObject 0CALWEEK. This variable is a characteristic value variable of the interval type, and you want to assign it the interval that starts in the current week and lasts for six weeks. The code could then appear as shown in Listing 4.3.

```
  DATA: l_d_datum TYPE d,
        l_d_woche TYPE /bi0/oicalweek.
  DATA: l_s_range TYPE rsr_s_rangesid.
CASE i_vnam.
  WHEN ‘WEEK6F’.
    IF i_step = 1. “Prior to popup
*     determine current week
      CALL FUNCTION ‘DATE_GET_WEEK’
        EXPORTING
          date          = sy-datum
        IMPORTING
         week           = l_d_woche.
*     Include week in return
      l_s_range-low = l_d_woche.
      l_s_range-sign = ‘I’.
      l_s_range-option = ‘EQ’.
*     Now determine six weeks later
      l_d_datum = sy-datum + 42.
                      “42 days = 6 weeks
      CALL FUNCTION ‘DATE_GET_WEEK’
        EXPORTING
          date          = l_d_datum
        IMPORTING
         week           = l_d_woche.
*    Fill in end of return interval
      l_s_range-high = l_d_woche.
      APPEND l_s_range TO l_t_range.
```

Listing 4.3 Example of a Date-Dependent Variable Pre-Assignment

Example 2: Table-Dependent Default Value Assignment

In the second example, the variable LASTCPER is supposed to contain the last closed period from the source system. To avoid having to call the source system via Remote Function Calls (RFCs) for every query call, a table Z_BEX_VAR was created that is maintained manually during the month-end closing and consists of the fields KEY (CHAR 20) and VALUE (CHAR 60). KEY is the only key field in the table, value the data field.

The last closed period is maintained in the entry that contains the string `KEY = 'LASTCPER'`. The code that's used to query the table is shown in Listing 4.4.

```
DATA: l_s_bex_var TYPE z_bex_var,
      l_s_range TYPE rsr_s_rangesid.
CASE i_vnam.
  WHEN 'LASTCPER'.
    IF i_step = 1. "Prior to variable popup
*     read table entry
      SELECT SINGLE * FROM z_bex_var
        INTO l_s_bex_var
        WHERE key = 'LASTCPER'.
      IF SY-SUBRC = 0.
*       Include in return
        l_s_range-low = l_s_bex_var-value.
      ELSE.
*       Here, an error could be output
        clear l_s_range-low.
      ENDIF.
      l_s_range-sign = 'I'.
      l_s_range-opt  = 'EQ'.
    ENDIF.
ENDCASE.
```

Listing 4.4 Example of a Table-Dependent Pre-Assignment

The type of pre-assignment you choose determines the type of implementation to be used. Often, the table-dependent default value pre-assignment is used to make the assignment dependent on the user name or even on specific user rights. Because it isn't easy to handle reporting authorizations, the following example demonstrates how to read RSR class reporting authorizations or the new analysis authorizations without having to perform an `AUTHORITY-CHECK` for each master record.

Example 3: Pre-Assignment Based on Authorizations

A customer who has cross-company business areas wants to implement a two-dimensional authorization object that merges the external company structure (InfoObject `0COMPANY`) and the internal business area (InfoObject `0BUS_AREA`) into one authorization object. This is intended to facilitate the differentiation between the external and internal view. Depending on the report that is chosen, the respective owners should see either all costs incurred within the entire company or all costs incurred within their business area.

[+]

Two-Dimensional Authorization Object

The fact that two InfoObjects are used in one reporting authorization is rather unusual, and many developers don't know that this is possible. But both the reporting process and planning run without any problem in the system, and they check the authorizations correctly.

For this reason, many employees have two values for their authorization objects, such as:

- `0COMPANY = 100` and `0BUS_AREA = *`
- `0COMPANY = *` and `0BUS_AREA = 1000`

There's a big problem with this concept. If an employee calls a report in the external view, the default value for his company, `100`, should be automatically displayed. However, if the variable is filled with information from the authorization, the system always displays the extended value, `*`, which represents an empty selection in this case.

The implementation shown in Listing 4.5 reads the authorizations and populates the default value accordingly. First, the existing authorizations for the `0BUS_AREA` InfoObject are read. Then, they are checked. If it is a general authorization (`*-authorization`), it is ignored. If it is a concrete business area, it is returned to the variable pre-assignment.

```
DATA: v_tsx_auth_values_user  TYPE
                              rssb_tsx_auth_values_user,
      w_sx_auth_values_user   TYPE  line of
                              rssb_tsx_auth_values_user,
      w_sx_auth_values_auth   TYPE
                              rssb_sx_auth_values_user_auth,
      w_sx_auth_values_iobjnm TYPE
                              rssb_sx_auth_values_user_iobj,
      w_sx_auth_values_range  LIKE  rrrange,
      v_ts_authnode           TYPE  rssbr_ts_authnode,
      w_s_authnode            TYPE  line of
                              rssbr_ts_authnode,
      v_authhieruid           TYPE  rssauthhieruid.

CONSTANTS: c_zcomp_bus TYPE rssb_sx_auth_values_user-object
             VALUE ‘ZCOMP_BUS’,
           c_attrinm_company TYPE
```

```
                      rssb_sx_auth_values_user_iobj-iobjnm
             VALUE ‘0COMPANY’,
           c_attrinm_busarea TYPE
                      rssb_sx_auth_values_user_iobj-iobjnm
             VALUE ‘0BUS_AREA’.

  IF i_step  = 1.

* Read all reporting authorizations of user
* (only InfoObject 0BUS_AREA)
  CALL FUNCTION ‘RSSB_AUTHORIZATIONS_OF_USER’
    EXPORTING
      I_IOBJNM                     = ‘0BUS_AREA’
      I_INFOPROV                   = ‘DEMOCUBE’
      I_UNAME                      = sy-uname
      I_HIENM                      = ‘DEMOHIER’
      I_DATETO                     = sy-datum
      I_VERSION                    = ‘000’
    IMPORTING
*     E_T_RANGESID                 =
      E_TSX_AUTH_VALUES_USER       = v_tsx_auth_values_user
*     E_NIOBJNM                    =
*     E_NODE                       =
*     E_TS_NODE                    =
*     E_TS_AUTH_VALUES_HIERARCHY   =
*     E_T_MSG                      =
    EXCEPTIONS
      NOT_AUTHORIZED               = 1
      INTERNAL_ERROR               = 2
      USER_DOESNT_EXIST            = 3
      X_MESSAGE                    = 4
      OTHERS                       = 5.
  IF sy-subrc <> 0.
    MESSAGE ID SY-MSGID TYPE SY-MSGTY
            NUMBER SY-MSGNO
            WITH SY-MSGV1 SY-MSGV2 SY-MSGV3 SY-MSGV4
            RAISING not_authorized.
  ENDIF.

* LOOP across all reporting authorizations of user
  LOOP AT v_tsx_auth_values_user
       INTO w_sx_auth_values_user.
*   Which authorization object?
```

```
    CHECK w_sx_auth_values_user-object = c_zcomp_bus.
*   Double authorization object LOOP across all values for
*   authorization object
    LOOP AT w_sx_auth_values_user-auth
         INTO w_sx_auth_values_auth.
*     Read company value
      READ TABLE w_sx_auth_values_auth-values_iobjnm
           INTO  w_sx_auth_values_iobjnm
           WITH KEY iobjnm = c_attrinm_company.
      IF sy-subrc = 0.
*       Company found
        LOOP AT w_sx_auth_values_iobjnm-ranges
             INTO w_sx_auth_values_range.
          IF  ( w_sx_auth_values_range-sign = 'I' )
          AND ( w_sx_auth_values_range-opt = 'CP' )
          AND ( w_sx_auth_values_range-low = '*' ).
*  Does an authorization exist for 0COMPANY?
*  Determine business area.
            READ TABLE w_sx_auth_values_auth-values_iobjnm
                 INTO  w_sx_auth_values_iobjnm
                 WITH KEY iobjnm = c_attrinm_busarea.
            IF sy-subrc = 0.
*       Authorization for business area
              LOOP AT w_sx_auth_values_iobjnm-ranges
                INTO w_sx_auth_values_range.
                MOVE-CORRESPONDING w_sx_auth_values_range
                                   TO l_s_range.
                APPEND l_s_range TO e_t_range.
              ENDLOOP. "w_sx_auth_values_iobjnm_range
            ENDIF.  "sy-subrc = 0
          ENDIF.  "* authorization for 0COMPANY
        ENDLOOP.  " w_sx_auth_values_iobjnm-ranges
      ENDIF.  "sy-subrc = 0
    ENDLOOP.  "w_sx_auth_values_user-auth
  ENDLOOP.    "v_tsx_auth_values_user
```

Listing 4.5 Derivation of Authorizations from Specific Authorization Objects

4.1.3 Implementation for I_STEP = 2

The implementation of Step 2 is essentially the same as that of Step 1. However, there's one significant difference: In Step 2, you can use the values that have previously been entered in the dialog box. This is often necessary if the user enters a specific time, and derivations are made based on that point in time. This scenario is explained in the following example.

Example 4: Time-Dependent Versions

The user wants to enter a reporting period in the variable, REP_PER. Depending on the time of variable REP_PER, the variable REP_VERS is to get the values listed in Table 4.1.

REP_PER Time	REP_VERS Value
Period is closed.	ACT (actual version)
Period is in the current year, but not closed	PRE (preview)
Period is in a subsequent year.	PLN (planned version)

Table 4.1 Version Implementation

The implementation for this is shown in Listing 4.6. First, the reporting period, REP_PER, is read. Then, the last day of the reporting period is determined. According to the specifications in Table 4.1, the return value is determined.

```
DATA: l_s_range TYPE rsr_s_rangesid.
DATA: l_s_var_range LIKE rrrangeexit,
      l_d_datum LIKE sy-datum,
      l_d_year  TYPE /bi0/oifiscyear,
      l_d_per3  TYPE /bi0/oifiscper3.
CASE i_vnam.
  WHEN ‘CUMMONTH’.
    CLEAR l_s_range.
    IF i_step = 2.  “Read entries
*     of REP_PER variable after dialog popup
      READ TABLE i_t_var_range INTO l_s_var_range
           WITH KEY vnam = ‘REP_PER’.
      IF sy-subrc = 0.
        l_d_year = l_s_var_range-low(4).
        l_d_per3 = l_s_var_range-low+4(3).
        CALL_FUNCTION ‘LAST_DAY_IN_PERIOD_GET’
          EXPORTING
            I_GJAHR   = l_d_year
            I_PERIV   = i_periv
            I_POPER  = l_d_per3
          IMPORTING
```

```
            E_DATUM  = l_d_datum
          EXCEPTIONS
            OTHERS   = 1.
        IF SY-SUBRC <> 0.
*         Cannot be determined (fiscal year
*         variant missing?) => set default
          l_s_range-low = 'ACT'.
        ELSE.
          IF l_d_datum <= sy-datum.
*           Period closed
            l_s_range-low = 'ACT'.
          ELSE.
            CALL_FUNCTION 'FIRST_DAY_IN_YEAR_GET'
              EXPORTING
                I_GJAHR   = l_d_year
                I_PERIV   = i_periv
              IMPORTING
                E_DATUM  = l_d_datum
              EXCEPTIONS
                OTHERS   = 1.
            IF sy-subrc <> 0.
*             Cannot be determined (but could period
*             be determined?) => set default
              l_s_range-low = 'ACT'.
            ELSE.
              IF l_d_datum > sy-datum.
*               Fiscal year has not yet started
                l_s_range-low = 'PLN'.
              ELSE.
                l_s_range-low = 'PRE'.
              ENDIF.
            ENDIF.
          ENDIF.
        ENDIF.
      ELSE.
*       Variable was used in error, set useful
*       default (e.g., ACT)
        l_s_range-low = 'ACT'.
      ENDIF.
      l_s_range-sign = 'I'.
      l_s_range-opt  = 'EQ'.
      APPEND l_s_range TO e_t_range.
```

Listing 4.6 Sample Variable Determination Based on Input Variables

Here are typical examples of variable determinations that depend on input variables.

- Derivation of a formula variable from an input period to extrapolate annual forecasts or to distribute annual values.
- Derivation of a comparison group (for example, a hierarchy node variable) for an input characteristic such as a profit center.
- Derivation of upper and lower limits of a time interval that is defined not by a fixed number of periods but by other constraints, such as the beginning or end of a quarter.
- Determination of an authorization-dependent formula variable to display periods that have not yet been released. Based on the specified period, a check is carried out as to whether the period has already been released. If not, an authorization check is performed. If the authorization check fails, 0 is returned; otherwise, 1. The query then multiplies the actual value by the result of the formula variable.
- Derivation of variables for MultiProviders. For example, you can query a variable for a cost center. You then derive the profit center that's assigned to the cost center because one of the involved InfoProviders doesn't contain any cost center, only the profit center.

4.1.4 Implementation for I_STEP = 0

Step 0 can be used, among other things, to derive variables in authorizations. In a reporting authorization, you can simply use a variable by inserting a variable name that begins with a dollar sign (`$`). This variable must be defined up front in the Query Designer. When running the report, the exit is called, and you can implement the authorization in ABAP.

When you determine authorizations, the return table is treated like an authorization so that all restrictions for authorizations must also be accounted for with regard to the return table. Accordingly, `SIGN` must always have the value `I`, and the value of `OPT` must either be `EQ` or `BT`. Excluding authorizations such as "all salaries except those of the executive board" aren't allowed.

Possible and typical applications for these authorizations are authorization derivations from tables or master data. You also can use excluding authorizations to assign an authorization dependent on the current date. Consider this typical example.

Example 5: Changing Authorizations in the Quiet Period

In this example, we want to implement the time-dependent control of an authorization for a regional hierarchy. All employees have authorizations for all regions except during the so-called *quiet period*, which lasts from the 25th of a quarter-end month until the 15th of the following month (in January, until the 25th of January). During that time, employees are allowed to view only those regions that are stored in a table. For this purpose, their authorization contains the variable QREGION. We'll now fill this variable in Listing 4.7. First, the current day is determined. Then, it is checked as to whether the day is in the period defined. If it is, the user can view specific regions only; otherwise, the user can view everything.

```
DATA: L_S_RANGE TYPE RSR_S_RANGESID.
CASE i_vnam.
  WHEN 'QREGION'.
*   Date in quiet period?
    DATA: l_d_tag(4) TYPE n,
          l_s_region TYPE z_user_region.
    l_d_tag = sy-datum+4(4).
    IF   ( l_d_tag <= '0125' )   OR
       ( ( l_d_tag >= '0325' ) AND
         ( l_d_tag <= '0415' ) ) OR
       ( ( l_d_tag >= '0625' ) AND
         ( l_d_tag <= '0715' ) ) OR
       ( ( l_d_tag >= '0925' ) AND
         ( l_d_tag <= '1015' ) ) OR
         ( l_d_tag >= '1225' ).
      l_s_range-sign = 'I'.
      l_s_range-opt  = 'EQ'.
      SELECT * FROM z_user_region
        INTO  l_s_region
        WHERE uname = sy-uname.
        l_s_range-low =
                      l_s_region-region.
        APPEND l_s_range TO e_t_range.
      ENDSELECT.
    ELSE.
      l_s_range-sign = 'I'.
      l_s_range-opt  = 'CP'.
      l_s_range-low  = '*'.
      APPEND l_s_range TO e_t_range.
    ENDIF.
```

Listing 4.7 Implementing an Authorization Variable

The `IF` query in Listing 4.7 normally would not be written directly in the variable but stored in a separate check function or even in a customizing table. Otherwise, those lines would have to be changed regularly in practice.

4.1.5 Implementation for I_STEP = 3

The implementation for `I_STEP = 3` is different and a little more complicated than the implementation of the other steps. This is because you must make sure that not every query uses the same variables and also because not all checks are restricted to one variable.

Nevertheless, some patterns recur in variable checks. Before these patterns can be examined, you should try to minimize the problems by using a clever variable-handling strategy.

[+]

Variable Handling in SAP NetWeaver Business Warehouse

The definition of variables in SAP NetWeaver BW always causes problems because you must take into account two opposing trends. In an ideal scenario, you want to reuse the variables, but you also want to use individual texts and default values for the individual reports. Moreover, the definition of variables is very easy, so many report developers prefer to define a new variable instead of searching through the existing ones to check whether a variable exists that meets the requirements of a specific report. This attitude is perfectly understandable to those of you who have seen what happens if a variable is redefined, and then users complain about sudden occurrences of incorrect default values in live reports.

Because there's no way to solve this problem, you should come up with a strategy that specifies how you want to use the variables. There are as many possible strategies as there are data warehouses, so the following suggestion is particularly intended for those readers who don't yet have any strategy.

Regarding variables, there is a simple rule of thumb: You must assign an owner to each variable. This is an easy way to resolve all usage problems. The rule could be implemented as follows:

- The owner of the system is responsible for all business content variables and defines central variables that can be used by everyone, such as time variables and variables for central company concepts.

- Each project[1] is assigned a prefix and can define any variable within this namespace. Furthermore, each project can use all central variables without restriction. If a project tries to use variables from other projects, the use must be announced. In addition, the project must inform those users who have announced the use about changes to variables.

If this strategy is adhered to, various central variables will be widely used, while an even greater number of project-specific variables will be used outside of the respective projects in exceptional cases. Those who don't comply with this strategy will probably do so when they see that one of their variables has been changed several times without prior announcement.

A consistent variable strategy has the following invaluable advantages for Step 3.

- It's immediately clear who is responsible for the implementation of the exit. If the validation is centrally defined, it's centrally implemented. If not, it's implemented in the project.
- Each project knows where its variables are being used and can adjust the implementation accordingly. This holds true especially for combination checks because, in those checks, it's clear where and in which combinations the variables are used.
- It's clear who must be informed about changes to the exit.

If you use the example shown in Listing 4.2 for implementing the variable exit, you'll probably ask yourself how you can implement the function module `Z_VAR_CHECK_VALIDITY` to meet these requirements. At this point, you can call a central function module and a function module for each project to carry out the validation. This way, you can make sure that each project can validate its own variables so that identical validations don't need to be implemented several times. On the other hand, if you do implement the identical validations, you can transport the validations separately.

In general, you must distinguish between two different cases:

- You can validate individual variables. The variant should not be query-dependent.
- You can validate combinations of variables. The second variant is often query-dependent because you can't be sure that other queries don't use the same variables.

1 By "project," we mean an entity that defines one or several InfoProviders that are separated based on their contents, including their reports. In large BW projects, this corresponds to the sub-project level.

4.1.6 Validating an Individual Variable

The validation of an individual variable is relatively simple. Step 3 of the variable exit determines the variable input and checks whether the value entered by the user is permitted.

Probably the most frequent use you'll find in practice is the validation of time entries. In this context, the check must determine whether specific intervals must have a minimum or maximum size or whether specific reports can be executed only once per quarter.

Example 6: Specifying a Quarter

A quarterly report is defined in an InfoProvider that contains the InfoObjects `0FISCPER` and `0FISCPER3` as its only time characteristics. Because certain postings are made only at the end of the quarter, the report would provide incorrect results if it wasn't run at the end of the quarter. For this reason, we define a variable, `FP_QUART`, which is intended to output an error if the user enters a value other than 3, 6, 9, or 12 for the period. Listing 4.8 describes the implementation.

```
DATA: l_s_var_range LIKE rrrangeexit.
READ TABLE i_t_var_range
     INTO l_s_var_range
     WITH KEY vnam = ‘FP_QUART’.
IF sy-subrc = 0.
  IF ( l_s_var_range-low = ‘003’ ) OR
     ( l_s_var_range-low = ‘006’ ) OR
     ( l_s_var_range-low = ‘009’ ) OR
     ( l_s_var_range-low = ‘012’ ).
*    Everything OK, don’t do anything
  ELSE.
*   The actual exception can be freely selected here.
    RAISE wrong_entry.
  ENDIF.
ELSE.
* If variable does not exist, do not perform any action
ENDIF.
```

Listing 4.8 Sample Validation of a Variable

At this point, you can see that the validation you want to implement should be error-tolerant. Each line you implement here is run through for all queries irrespective of the variables that are actually used in the query, provided that at least one variable is used. For this reason, you shouldn't constantly output error messages; instead you should react only if you're sure the overall combination is wrong.

You can't easily check whether the query contains certain characteristics. You can obtain the name of the query, but the exit doesn't contain any definition. Therefore, it's usually impossible to check at this point whether a variable was used for a specific characteristic or whether the characteristic exists without any constraint in the query.

4.1.7 Checking Characteristic Combinations in Step 3

The validation of multiple variables in the exit is a much more specific case. The reasons for a combination check are often as specific as the algorithms those checks are based on. Examples are characteristic values that are valid only in specific periods or variable inputs for compounded characteristics.

A frequent scenario is the validation of time intervals if they aren't entered as intervals but as individual variables. This scenario is described in the following example.

Example 7: Validating Period Entries

In the following example, four variables are queried: two for "Period from and to," and two for "Fiscal year from and to." The exit checks whether "Fiscal year from" is smaller than "Fiscal year to" or whether "Fiscal year from" and "Fiscal year to" are equal, and whether "Period from" is smaller than or equal to "Period to." This is implemented as shown in Listing 4.9.

```
DATA: l_s_var_range TYPE rrrangeexit,
      l_d_year_from TYPE /bi0/oifiscyear,
      l_d_year_to   TYPE /bi0/oifiscyear,
      l_d_per3_from TYPE /bi0/oifiscper3,
      l_d_per3_to   TYPE /bi0/oifiscper3.
IF i_step = 3.
  READ TABLE i_t_var_range INTO l_s_var_range
       WITH KEY vnam = ‘YEARFROM’.
  IF sy-subrc = 0.
    l_d_year_from = l_s_var_range-low(4).
  ELSE.
*   If the variable does not exist, the value is set to
*   0000 to make the final validation always valid.
```

```
    l_d_year_from = ‘0000’.
  ENDIF.
  READ TABLE i_t_var_range INTO l_s_var_range
       WITH KEY vnam = ‘YEARTO’.
  IF sy-subrc = 0.
    l_d_year_to   = l_s_var_range-low(4).
  ELSE.
*   If the variable does not exist, the value is set to
*   9999 to make the final validation always valid.
    l_d_year_to   = ‘9999’.
  ENDIF.
  READ TABLE i_t_var_range INTO l_s_var_range
       WITH KEY vnam = ‘PER3FROM’.
  IF sy-subrc = 0.
    l_d_per3_from = l_s_var_range-low(4).
  ELSE.
*   If the variable does not exist, the value is set to 000
*   to make the final validation always valid.
    l_d_per3_from = ‘000’.
  ENDIF.
  READ TABLE i_t_var_range INTO l_s_var_range
       WITH KEY vnam = ‘PER3TO’.
  IF sy-subrc = 0.
    l_d_year_from = l_s_var_range-low(4).
  ELSE.
*   If the variable does not exist, the value is set to 999
*   to make the final validation always valid.
    l_d_year_from = ‘999’.
  ENDIF.
  IF l_d_year_from > l_d_year_to.
    RAISE wrong_value.
  ELSEIF l_d_year_from = l_d_year_to.
    IF l_d_per3_from > l_d_per3_to.
      RAISE wrong_value.
    ENDIF.
  ENDIF.
ENDIF.
```

Listing 4.9 Sample Validation of Characteristic Combinations

The example in Listing 4.9 shows how conscientiously you must be about ensuring that an error message is returned only if an error really occurs. Here, the validation check is carried out only if the query uses all four variables. Otherwise, setting the default values to 9999 for the fiscal year or 000 for the period makes sure that the lower IF queries never cause an error.

You can also see the problems involved in those queries. If only one query developer uses the variables YEARFROM, PER3FROM, and PER3TO and expects the system to automatically check whether PER3FROM is smaller than or equal to PER3TO, the developer will see that this isn't the case.

For this reason, you must either modify the query in such a way that different conditions are validated depending on the variables being used, or you must limit the query to specific queries. To do this, you must query the field, I_S_RKB1D-COMPID. The disadvantage of this variant is that new queries must be released via a corresponding specific customizing for the validation to work properly.

[+]

Returning Error Messages

The validation types described so far all work, but they have one essential drawback: The user receives only a vague error message saying that the values of the variables are invalid. However, the user doesn't obtain any information on the variables that actually caused the error. This is why, if possible, you should store a corresponding message in the message collector in addition to the actual RAISE EXCEPTION statement.

The exit can't return error messages by itself unlike the routines for data staging. Nor is it possible to use a MESSAGE...RAISING... call to return an adequate message because that message would simply be ignored. For this reason, you must use the same message collector used by the OLAP processor: the function module RRMS_MESSAGE_HANDLING. This function module contains the typical interface for a message collector (see Listing 4.10).

```
FUNCTION rrms_message_handling.
*"----------------------------------------
*"*"Local interface:
*"  IMPORTING
*"     VALUE(I_CLASS)  LIKE  SMESG-ARBGB
*"                     DEFAULT 'BRAIN'
*"     VALUE(I_TYPE)   LIKE  SMESG-MSGTY
*"                     DEFAULT 'S'
*"     VALUE(I_NUMBER) LIKE  SMESG-TXTNR
*"                     DEFAULT '000'
*"     VALUE(I_MSGV1)  OPTIONAL
*"     VALUE(I_MSGV2)  OPTIONAL
*"     VALUE(I_MSGV3)  OPTIONAL
*"     VALUE(I_MSGV4)  OPTIONAL
*"     VALUE(I_INTERRUPT_SEVERITY)
*"       LIKE SY-SUBRC DEFAULT 16
*"     VALUE(I_LANGU)  LIKE SY-LANGU
*"                     DEFAULT SY-LANGU
*"     VALUE(I_SAPGUI_FLAG) DEFAULT SPACE
*"     VALUE(I_SUPPRMESS)
*"       TYPE RSRSUPPRMESS OPTIONAL
*"  EXCEPTIONS
*"      DUMMY
```

Listing 4.10 Interface of Function Module RRMS_MESSAGE_HANDLING

If you want to output only plain text, you can use message 000 of message class RSBBS. However, if possible, you should create your own messages in Transaction SE91 to be able to specify a long text.

If you use this function module, the final part of the example in Listing 4.9 can then be implemented as shown in Listing 4.11.

```
IF l_d_year_from > l_d_year_to.
  CALL FUNCTION 'RRMS_MESSAGE_HANDLING'
    EXPORTING
      I_CLASS  = 'RSBBS'
      I_TYPE   = 'I'
      I_NUMBER = '000'
      I_MSGV1  = 'Year from bigger than year to'.
  RAISE WRONG_VALUE.
ELSEIF l_d_year_from = l_d_year_to.
  IF l_d_per3_from > l_d_per3_to.
  CALL FUNCTION 'RRMS_MESSAGE_HANDLING'
     EXPORTING
      I_CLASS  = 'RSBBS'
      I_TYPE   = 'I'
      I_NUMBER = '000'
      I_MSGV1  = 'Month from bigger than month to'.
    RAISE WRONG_VALUE.
  ENDIF.
ENDIF.
```

Listing 4.11 Example of Using Messages

Of course, you can also use the function module RRMS_MESSAGE_HANDLING to return error messages in the other steps of the exit, which always makes sense.

4.2 Virtual Key Figures and Characteristics

Another exit frequently used in SAP NetWeaver BW is the exit for virtual key figures and characteristics. Virtual key figures and characteristics are InfoObjects that are contained in the InfoProvider, but whose values aren't read from the InfoProvider. Instead, the values are determined at the time the query is run. There are many reasons for this; these are the most frequent:

- You want to carry out calculations that are too complex for query formulas and for which you don't have all necessary information at the time of loading.
- You want to integrate data from several InfoProviders, but the results provided by the MultiProvider are insufficient.
- You must carry out calculations at a level that is more detailed than the query. This happens, for example, if costs are calculated as a product of quantity and price at purchase-order level, while the query presents the results at sales department level.
- You want to carry out complex calculations based on user input.
- You want to display specific values based on the time the query is executed or hide those values (during the quiet period, see Example 5 in Section 4.1.4).

In all these cases, you'll have to use virtual key figures and characteristics.

The advantage of virtual key figures and characteristics is that they make reporting much more flexible. The ability to generate values at the time a query is run provides simple solutions to various problems that otherwise would have to be calculated manually.

However, this method also has one drawback: Each calculation that's carried out at this point affects the performance during the query execution. Therefore, you must consider two factors. First, you should try to avoid any kind of database access. Second, the exit enables you to read additional characteristics from the database. This means that existing queries no longer access existing aggregates.

This can have a substantial impact, particularly because the exit is called once per data record, and the data record in turn is read from the database. If the records are read from an aggregate, the number of records read from the database is significantly smaller. For this reason, it's important to always test the exit by using (almost) live data, particularly with regard to the number of data records, when you use virtual key figures and characteristics.

[Ex]

Testing the Use of Virtual Key Figures

In a customer system, the exit for virtual key figures was run in a test system with approximately 500 data records without any error and then transported into the live system. When the respective report was called in the live system, it took two hours before the report returned the required data. In this case, the data was wrong because not all combinations had been tested in the test system.

4.2.1 Implementation

Those of you who have experience with virtual key figures and characteristics may remember the user exit `RSR00002`, which allowed you to implement virtual InfoObjects before BW release 3.0. This user exit was very difficult to implement because it required the creation of specific form routines with fixed naming conventions in certain includes. Consequently, it was as easy to make mistakes as it was difficult to find those mistakes.

This is why we want to focus on describing how to implement virtual key figures and characteristics using the BAdI `RSR_OLAP_BADI`. Compared to the old implementation, this BAdI has the following three advantages:

- You can create different implementations that can even be restricted to different InfoProviders.
- You need to implement only two methods. A relatively simple pattern is available for both of these. This pattern facilitates the implementation. A third method that is available in the BAdI can be copied unmodified from the sample implementation.
- There are no large confusing includes.

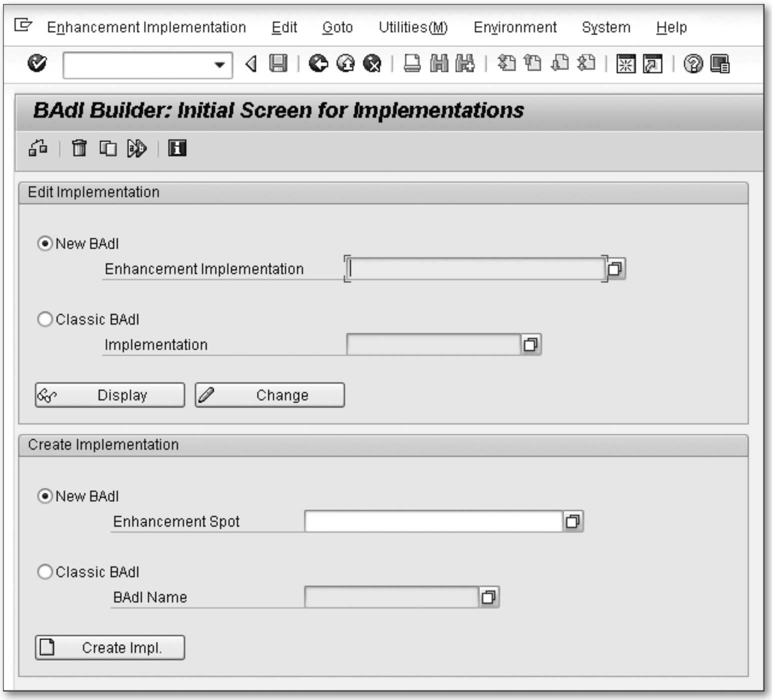
These three factors make it easier to implement virtual key figures and characteristics in such a way that after you have implemented the BAdI once or twice, you'll be able to quickly create more.

The following describes an example in which you would want to implement the display of customer contacts. An InfoProvider contains purchase orders that include customer numbers and sales offices. A DataStore object, `CCONTACT` (current contact), contains one contact (`CONTACT`) per sales office (`SALESOFF`) and customer number (`0CUSTOMER`). The sales office and customer number aren't compounded. You want to create a report that displays the current contact depending on the sales office.

To do this, the DataStore must be read at report runtime because it's impossible to reload the complete purchase order InfoProvider if the contacts change.

Step 1: Creating the BAdI Implementation

1. Call Transaction SE19 to create a BAdI implementation. This transaction was completely revised for SAP NetWeaver 7.0.
2. In SAP NetWeaver 7.x, go to the Create Implementation section, and choose the Classic BAdI radio button (see Figure 4.1).


Enhancement Implementation
Edit
Goto
Utilities(M)
Environment
System
Help
BAdI Builder: Initial Screen for Implementations
Display
Change
Create Impl.

Edit Implementation

(●) New BAdI
Enhancement Implementation

( ) Classic BAdI
Implementation

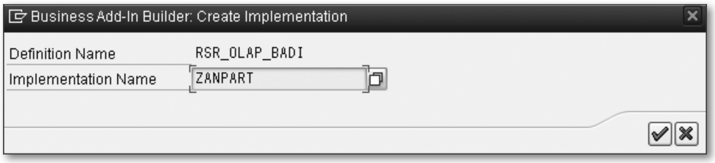
Create Implementation

(●) New BAdI
Enhancement Spot

( ) Classic BAdI
BAdI Name

Figure 4.1 Creating a BAdI in SAP NetWeaver 7.x

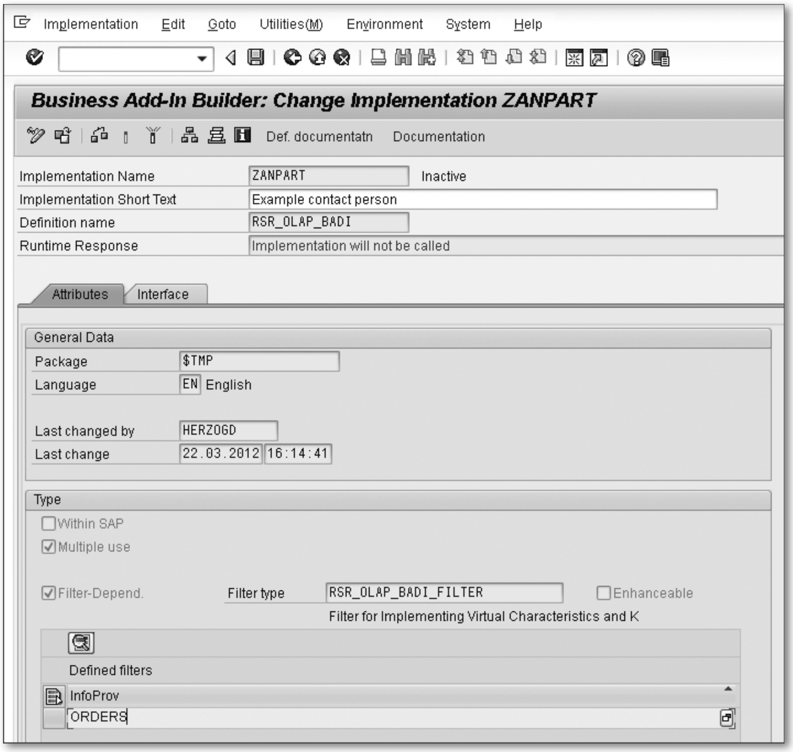
3. Enter the name of the BAdI definition. For virtual key figures and characteristics, that's "RSR_OLAP_BADI". Click on the Create Impl. button.
4. Enter a corresponding name in the dialog box that follows, for example, "ZAN-PART" (see Figure 4.2).


Business Add-In Builder: Create Implementation

Definition Name	RSR_OLAP_BADI
Implementation Name	ZANPART

Figure 4.2 Dialog Box for Querying the Implementation

5. The dialog box that follows prompts you for the BAdI definition. The BAdI definition for virtual key figures and characteristics is called "RSR_OLAP_BADI".
6. After you have confirmed your entry by pressing `Enter`, the attributes of the BAdI implementation are displayed. Assign an Implementation Short Text, and, in the lower part of the screen, select the InfoProvider for which you want to create the implementation. Masked entries are also allowed. In our example, the InfoProvider is called `ORDERS` (see Figure 4.3).



Business Add-In Builder: Change Implementation ZANPART

Implementation Name	ZANPART Inactive
Implementation Short Text	Example contact person
Definition name	RSR_OLAP_BADI
Runtime Response	Implementation will not be called

General Data

Package	$TMP
Language	EN English
Last changed by	HERZOGD
Last change	22.03.2012 16:14:41

Type

☐ Within SAP
☑ Multiple use
☑ Filter-Depend. Filter type RSR_OLAP_BADI_FILTER ☐ Enhanceable
Filter for Implementing Virtual Characteristics and K

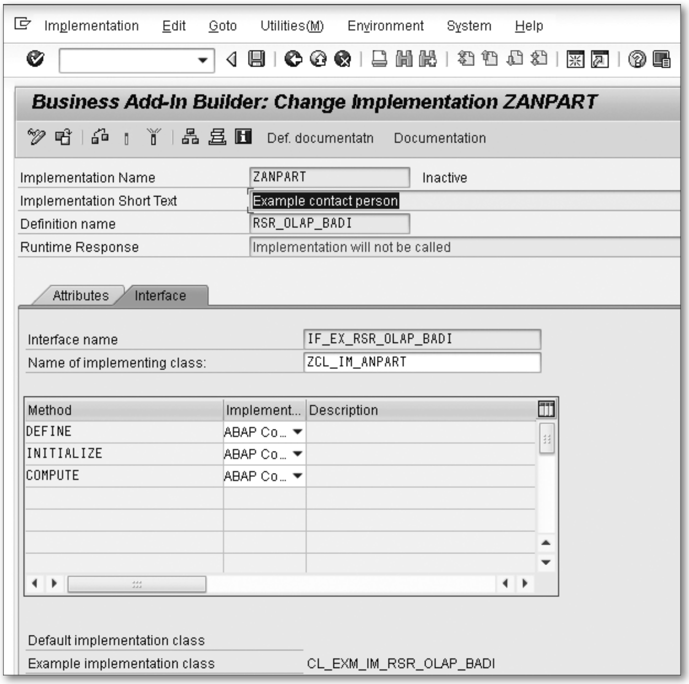
Defined filters

InfoProv
ORDERS

Figure 4.3 Properties of the BAdI

7. Save the BAdI implementation prior to copying the sample code provided by SAP via GOTO • SAMPLE CODING • COPY. This step is important because the sample code contains the complete `INITIALIZATION` method, which means you don't need to change that.
8. Go to the INTERFACE tab (see Figure 4.4). You'll see that the BAdI contains three methods for implementation: `DEFINE`, `INITIALIZATION`, and `COMPUTE`. You need to implement only the first and the last one.

   Moreover, at this point, you can still change the name of the class to be implemented. But you should do that only in exceptional cases. The class name suggested by the system can be clearly derived from the name of the implementation so that the class name clearly indicates that the implementation is a BAdI implementation.



Implementation Name	ZANPART	Inactive
Implementation Short Text	Example contact person	
Definition name	RSR_OLAP_BADI	
Runtime Response	Implementation will not be called	

Interface name	IF_EX_RSR_OLAP_BADI
Name of implementing class:	ZCL_IM_ANPART

Method	Implement...	Description
DEFINE	ABAP Co...	
INITIALIZE	ABAP Co...	
COMPUTE	ABAP Co...	

Default implementation class	
Example implementation class	CL_EXM_IM_RSR_OLAP_BADI

Figure 4.4 Interface of the BAdI

Step 2: Implementing the DEFINE Method

In the next step, you implement the `DEFINE` method. The `DEFINE` method is called at the time the query is generated, and it tells the system which characteristics and key figures are virtual and which ones are required for calculating the virtual characteristics.

The method contains the interface shown in Table 4.2.

Type	Parameter	Type	Description
I	FLT_VAL	RSR_OLAP_BADI_FLT	FLT_VAL specifies the filter that is defined in the interface.
I	I_S_RKB1D	RSR_S_RKB1D	I_S_RKB1D is the same parameter that is used in the interface for user exit EXIT_SAPLRRS0_001. It contains information on the query being used, particularly on the InfoProvider that is used.
I	I_TH_CHANM_USED	RRKE_TH_CHANM	I_TH_CHANM_USED contains all characteristics used in the query. Characteristics that don't exist in the query don't need to be calculated.
I	I_TH_KYFNM_USED	RRKE_TH_KYFNM	I_TH_KYFNM_USED contains all key figures used in the query. This is important to ensuring that key figures that don't exist in the query don't need to be calculated.
C	C_T_CHANM	RRKE_T_CHANM	C_T_CHANM and C_T_KYFNM must be filled in the method. That's where the characteristics and key figures that are to be filled and are required as a precondition are included. In addition to the InfoObject, C_T_CHANM contains the read mode in the MODE field. This field has two values: RRKE_C_MODE-READ makes sure that the InfoObject is read from the database, while RRKE_C_MODE-NO_SELECTION makes sure that the InfoObject isn't read from the database and that it can be filled in the COMPUTE method instead.
C	C_T_KYFNM	RSD_T_KYFNM	
X	CX_RS_ERROR		In case of a severe error, the exception CX_RS_ERROR can be triggered.
I = Importing, E = Exporting, C = Changing, X = Exception			

Table 4.2 Signature of the DEFINE Method

1. Double-click on the method name `DEFINE` to go to the ABAP Editor where you want to implement the method. For this example, you should enter the code shown in Listing 4.12.

```
METHOD IF_EX_RSR_OLAP_BADI~DEFINE .
  DATA: l_s_chanm      TYPE rrke_s_chanm,
        l_s_chanm_used TYPE rschanm,
        l_kyfnm        TYPE rsd_kyfnm.
  CASE i_s_rkb1d-infocube.
    WHEN 'ORDER'.
      LOOP AT i_th_chanm_used INTO l_s_chanm_used.
        CASE l_s_chanm_used.
*         Here the required InfoObjects are queried
          WHEN 'CONTACT'.
            l_s_chanm-chanm = 'CONTACT'.
            l_s_chanm-mode  =  rrke_c_mode-no_selection.
            APPEND l_s_chanm TO c_t_chanm.
            l_s_chanm-chanm = 'SALESOFF'.
            l_s_chanm-mode  = rrke_c_mode-read.
            APPEND l_s_chanm TO c_t_chanm.
            l_s_chanm-chanm = '0CUSTOMER'.
            l_s_chanm-mode  = rrke_c_mode-read.
            APPEND l_s_chanm TO c_t_chanm.
        ENDCASE.
      ENDLOOP.
*     LOOP AT i_t_kyfnm_used into l_kyfnm.
*       CASE l_kyfnm.
*         WHEN '...'.
*           APPEND l_kyfnm TO c_t_kyfnm.
*       ENDCASE.
*     ENDLOOP.
  ENDCASE.
```

Listing 4.12 Implementing the DEFINE Method

It usually makes sense to have the `DEFINE` method begin with a `CASE` query on InfoProvider `I_S_RKB1D-INFOCUBE`, even though you can use only one Info-Provider. If you do that, you can already see in the code that InfoProvider the definition runs on.

2. The required characteristics must be inserted in table `C_T_CHANM`. A decisive factor regarding functionality is the read mode, `L_S_CHANM-MODE`.

Two read modes are available:

- ▶ `RRKE_C_MODE-READ` reads the characteristic from the database in any case, even if it isn't used in the query. For this reason, you may no longer be able to use existing aggregates.
- ▶ `RRKE_C_MODE-NO_SELECTION` doesn't read the characteristic from the database. Instead, it marks the characteristic as virtual. This means it can be modified in the BAdI.

3. If the contact is contained in the query, the sales office and the customer number must also be read so that the contact can be derived. So the query uses only aggregates that contain `SALESOFF` and `0CUSTOMER`. After that, the key figures follow. The key figures don't require a read mode.
4. Finally, you must save and activate the method.

[+]

Object-Oriented Programming

At this point, it's inevitable that you will take a quick look at object-oriented programming because you will have to create attributes for a class. But we'll focus on the required technique without going into the theoretical details. You should regard the instance attributes as a kind of global variable.

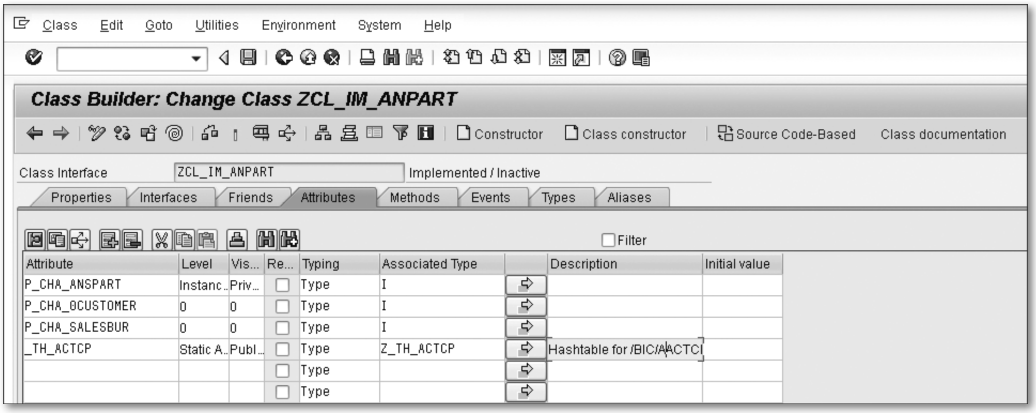
Step 3: Creating the Instance Attributes

1. If you double-click on the `INITIALIZATION` method to take a look at it, you see a complete implementation that doesn't need to be changed. This method is used to facilitate the implementation of the `COMPUTE` method. However, you must first do some preparatory work.
2. Double-click on the name of the implementing class in the INTERFACE tab of the BAdI to go to the class definition (see Figure 4.4). Then select the ATTRIBUTES tab (see Figure 4.5).
3. Now you must include an attribute for each InfoObject that you inserted into the `C_T_CHANM` and `C_T_KFNM` tables in the `DEFINE` method. For characteristics, this attribute is called `P_CHA_<InfoObject>`; for key figures, `P_KYF_<InfoObject>`. All attributes are instance attributes that have the visibility PUBLIC of type `I`.

   For this case, you must create three public instance attributes: `P_CHA_CONTACT`, `P_CHA_0CUSTOMER`, and `P_CHA_SALESOFF`. These attributes are necessary because the structure that is transferred to the `COMPUTE` method doesn't contain any typical field names such as /BIC/CONTACT. Instead, it contains field names such as K_022.

To access the correct fields, the INITIALIZATION method fills the attributes as follows. If the InfoObject is contained in the query, the attribute P_CHA_<InfoObject> is assigned a value greater than 0. This value indicates the position of the InfoObject within the structure. If the query doesn't contain the InfoObject, the value is 0. These attributes are used in the COMPUTE method to access the appropriate field via an ASSIGN COMPONENT.

4. In addition, you need another attribute that buffers the data of the DataStore object. For this purpose, you need a Z_TH_CCONTACT table type that references a hashed table with the structure /BIC/ACCONTACT00.
5. Create a static attribute, _TH_CCONTACT, that has the corresponding table type. The result is displayed in Figure 4.5.



Class Builder: Change Class ZCL_IM_ANPART

Class Interface ZCL_IM_ANPART Implemented / Inactive

Attribute	Level	Vis...	Re...	Typing	Associated Type		Description	Initial value
P_CHA_ANSPART	Instanc...	Priv...	☐	Type	I	⇨		
P_CHA_0CUSTOMER	0	0	☐	Type	I	⇨		
P_CHA_SALESBUR	0	0	☐	Type	I	⇨		
_TH_ACTCP	Static A...	Publ...	☐	Type	Z_TH_ACTCP	⇨	Hashtable for /BIC/AACTCI	
			☐	Type		⇨		
			☐	Type		⇨		

Figure 4.5 Attributes of the ZCL_IM_CONTACT Class

Step 4: Implementing the COMPUTE Method

Now you can implement the COMPUTE method:

1. Double-click on the method name COMPUTE in the INTERFACE tab of the BAdI implementation to go to the implementation of the method that contains the interface from Table 4.3.

Type	Parameter	Type	Description
I	FLT_VAL	RSR_OLAP_BADI_FILTER	FLT_VAL specifies the filter that is stored in the BAdI.
I	I_PARTCUBE	RSD_INFOCUBE	I_PARTCUBE specifies the subprovider of the MultiProvider from which the current data record originates.
I	I_S_RKB1D	RSR_S_RKB1D	I_S_RKB1D is the same parameter that is used in the interface for user exit EXIT_SAPLRRS0_001. It contains information on the query being used, particularly on the InfoProvider that is used.
I	I_NCUM	RS_BOOL	I_NCUM specifies whether the InfoProvider contains inventory values.
C	C_S_DATA	ANY	C_S_DATA is the structure that contains the values from the database in the structure that's necessary for the query. This structure also contains the fields that were included in the C_T_CHANM and C_T_KYFNM tables in the DEFINE method. However, the field names don't correspond to the usual field names in database tables.
I = Importing, C = Changing			

Table 4.3 Signature of the COMPUTE Method

2. In this method, you will implement the actual calculation of the virtual characteristics. The implementation of the method consists of two parts. The first part is standardized and is used to assign input and output values to field symbols. The second part contains the actual calculation. The first part is shown in Listing 4.13.

```
METHOD IF_EX_RSR_OLAP_BADI~COMPUTE .
FIELD-SYMBOLS: <l_contact>  TYPE /bic/oicontact,
               <l_customer> TYPE /bi0/oicustomer,
               <l_salesoff>   TYPE /bic/oisalesoff.
* Assign field symbols using instance attributes
  IF p_cha_contact > 0.
     ASSIGN COMPONENT p_cha_contact
```

```
        OF STRUCTURE c_s_data
        TO <l_contact>.
  ELSE.
*   If virtual characteristic does not exist, exit routine
    EXIT.
  ENDIF.
  IF p_cha_customer > 0.
     ASSIGN COMPONENT p_cha_customer
       OF STRUCTURE c_s_data
       TO <l_customer>.
  ELSE.
*   If source characteristic does not exist, initialize
*   CONTACT and exit routine
    CLEAR <l_contact>.
    EXIT.
  ENDIF.
  IF p_cha_salesoff > 0.
     ASSIGN COMPONENT p_cha_salesoff
       OF STRUCTURE c_s_data
       TO <l_salesoff>.
  ELSE.
*   If source characteristic does not exist, initialize
*   CONTACT and exit routine
    CLEAR <l_contact>.
    EXIT.
  ENDIF.
```

Listing 4.13 First Part of the Implementation of the COMPUTE Method

The decisive command is always `ASSIGN COMPONENT p_....` This command must be entered for each InfoObject that is required for the calculation. Because the exit is usually executed for each query on the InfoProvider, you must always be able to react to the fact that certain InfoObjects aren't contained in the query; sometimes you may even skip the entire calculation.

3. After all field symbols have been assigned in this way, the actual calculation can begin (see Listing 4.14).

```
  IF _th_contact[] IS INITIAL.
    SELECT * FROM /bic/accontact00
      INTO TABLE _th_ccontact.
  ENDIF.
  DATA: l_s_ccontact TYPE /bic/acontact00.
  READ TABLE _th_ccontact INTO l_s_ccontact
```

```
    WITH TABLE KEY
      customer      = <l_customer>
      /bic/salesoff = <l_salesoff>.
  IF sy-subrc = 0.
    <l_contact> = l_s_ccontact-/bic/contact.
  ELSE.
*   No records exists, hence empty return value
    CLEAR <l_contact>.
  ENDIF.
```

Listing 4.14 Second Part of the Implementation of the COMPUTE Method

If necessary, the contents of the DataStore are read first. Usually, this is done in a `CONSTRUCTOR` method. However, we do that here to keep the example simple. After the field symbols have been assigned, the table entry can be read.

Here it's important that you keep the logic as efficient as possible. The `ASSIGN` calls per data record are already time-consuming enough. For this reason, you should use hashed tables whenever possible, fill these tables in the `CONSTRUCTOR`, and implement as few database accesses as possible. For the database selection in the `CONSTRUCTOR`, you must make sure that all `SELECT` calls in the `CONSTRUCTOR` are executed even when the table isn't needed in the `COMPUTE` method; if, for instance, the virtual characteristic isn't used at all in the query. Otherwise, the performance of those queries would be affected without any reason. Large loops usually also have a negative influence on the query performance, but they are rarely used because the `COMPUTE` method only processes individual records.

4.2.2 Other Useful Information

The procedure described in the preceding example is typical of a BAdI implementation. However, you must take some things into account for more complex applications:

- You can't make any statement on the sequence in which the data records arrive. For example, if you try to determine the material range of coverage by first entering the inventory and then running the open orders against the inventory in the exit, you will be faced with the problem that the orders will probably not be received in the correct time sequence.
- Because only individual processing is possible, dependencies between individual records can't be evaluated. Therefore, standard deviations and variations can't be calculated in this BAdI.

- Moreover, you can't create or insert any additional records in this BAdI. If not all characteristic combinations are available in the InfoProvider, you can't generate any additional ones.
- In individual cases, the relationship between virtual characteristics and the elimination of intercompany sales can be important. The elimination of intercompany sales doesn't consider the virtual characteristics; the elimination is based on the values that are originally stored in the InfoCube. This fact prevents you from creating some nice solutions that would use the elimination of intercompany sales to adjust the hierarchy aggregation.

4.2.3 Transferring Variable Values to the BAdI

Another option to make the BAdI more flexible is the transfer of variables. For this purpose, a static public attribute is created in the implementing class for each variable to be transferred to the BAdI.

The actual transfer of the variable occurs in Step 3 of exit `EXIT_SAPLRRS0_001`. You can use the code shown in Listing 4.15, and in the BAdI, you can simply access the value `ZCL_IM_CONTACT=>MY_VAR`. If you want to transfer intervals and selection options, the query must be modified accordingly.

```
DATA: l_s_var_range LIKE rrrangeexit.
READ TABLE i_t_var_range
     INTO l_s_var_range
     WITH KEY vnam = 'MY_VAR'.
IF SY-SUBRC = 0.
  zcl_im_anpart=>my_var = l_s_var_range-low.
ENDIF.
```

Listing 4.15 Sample Variable Transfer to a BAdI

Variables can be particularly useful in restricting a possible selection right from the start. Instead of reading the entire contents of a DataStore object, only the required records are read. It can also be useful to transfer formula variables or posting periods to the exit if you want to implement specific nonlinear extrapolations.

The virtual key figures and characteristics represent a very efficient way to extend the reporting functionality. Because you can access variables and also add more database tables to the calculation, you can bypass many restrictions of SAP NetWeaver BW.

However, because the BAdI is called at a sensitive point, namely after the data has been read from the database, the implementation must be very clean to avoid problems in reporting performance.

Furthermore, you must always take into account that the BAdI is called for all queries that run on the corresponding InfoProviders even if the virtual characteristics and key figures aren't used. Here you must ensure that calculations and especially database selections are carried out only if they are really required.

4.3 VirtualProvider

If you think that the exits discussed so far aren't flexible enough to provide the required results, your last resort is the *VirtualProvider*, or rather the VirtualProvider that's based on a function module (in SAP BW 3.x it was called *RemoteCube with services*). By definition, a VirtualProvider is an InfoProvider that doesn't store the data by itself but reads the data from other data sources that aren't InfoProviders (as opposed to MultiProviders). In this context, three DataSources are relevant:

- The data is read from a DataSource (for example, directly from the source system)
- The data is read from a table using a BAPI
- The data is calculated using a function module

The third method, in particular, provides a lot of options for designing a report, but its implementation is complex and exceeds the scope of this book. The following sections describe how you can create a VirtualProvider and which interfaces you must implement; however, we don't provide any extensive sample implementations.

[+]

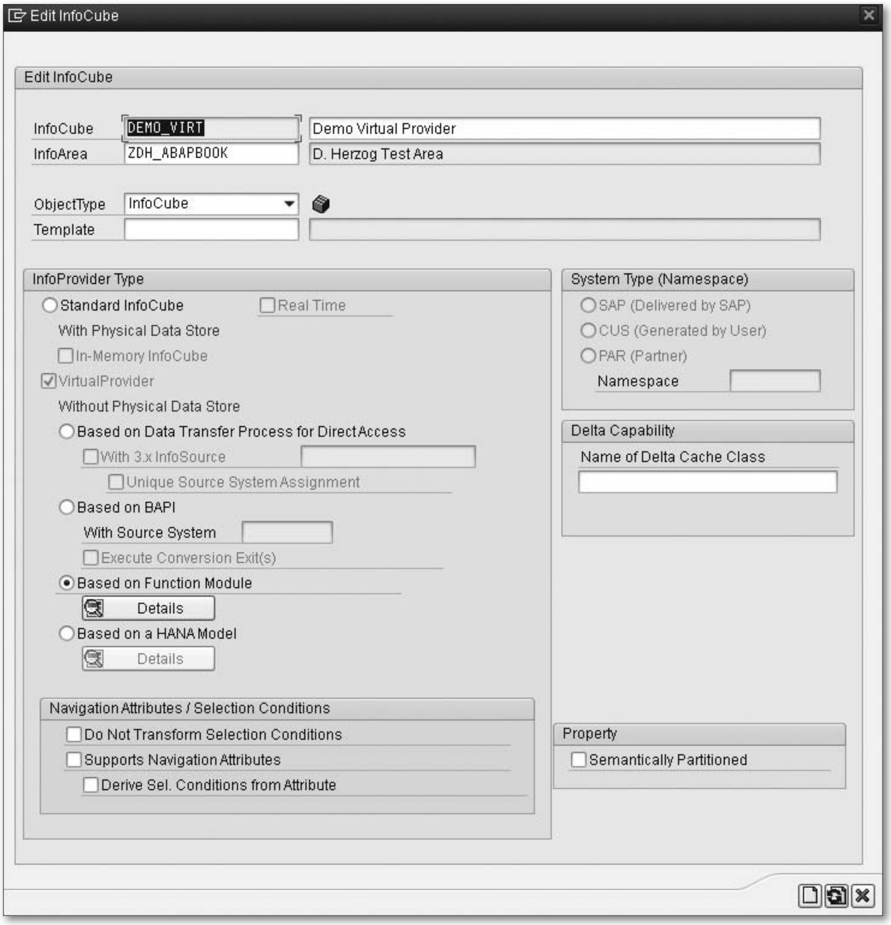
Avoiding Rounding Differences

If, in an annual report, certain overview reports are presented in terms of thousands, rounding differences are very likely to occur (for example, \$500,000 + \$500,000 = \$1,000,000 would be represented as 1+1=1 in terms of thousands). If you try to round off the results during the loading process, the totals will no longer match. For this reason, you could use a VirtualProvider to read the basic data and convert it to thousands, then aggregate both values on the basis of the transferred variables, calculate the difference, and distribute the difference downward. Some developers prefer to correct the rounding differences manually.

Interestingly, it's much easier to include an additional line for the rounding difference in the program code and show the rounding difference there.

4.3.1 Creating a VirtualProvider

You can create a VirtualProvider via the context menu of an InfoProvider. You have three options for a VirtualProvider (see Figure 4.6).


Edit InfoCube

Edit InfoCube

InfoCube: DEMO_VIRT — Demo Virtual Provider
InfoArea: ZDH_ABAPBOOK — D. Herzog Test Area
ObjectType: InfoCube
Template:

InfoProvider Type
- ☐ Standard InfoCube ☐ Real Time
- With Physical Data Store
- ☐ In-Memory InfoCube
- ☑ VirtualProvider
- Without Physical Data Store
- ○ Based on Data Transfer Process for Direct Access
  - ☐ With 3.x InfoSource
  - ☐ Unique Source System Assignment
- ○ Based on BAPI
  - With Source System
  - ☐ Execute Conversion Exit(s)
- ● Based on Function Module
  - Details
- ○ Based on a HANA Model
  - Details

Navigation Attributes / Selection Conditions
- ☐ Do Not Transform Selection Conditions
- ☐ Supports Navigation Attributes
- ☐ Derive Sel. Conditions from Attribute

System Type (Namespace)
- ○ SAP (Delivered by SAP)
- ○ CUS (Generated by User)
- ○ PAR (Partner)
- Namespace

Delta Capability
- Name of Delta Cache Class

Property
- ☐ Semantically Partitioned

Figure 4.6 Creating a VirtualProvider

From the point of view of an ABAP developer, the most interesting and most flexible VirtualProvider is the VIRTUALPROVIDER <EURO2>N BASED ON FUNCTION MODULE path. If you select this VirtualProvider and click on the DETAILS button, various control checkboxes are displayed. For the implementation, the checkboxes RFC PACKING and SID SUPPORT are of particular importance because the interface of the function module to be implemented changes depending on these switches. The following scenarios are possible.

Case 1: RFC Packing Is Set

If RFC Packing is set for the VirtualProvider, the interface is structured as shown in Listing 4.16.

```
IMPORTING
  infocube LIKE bapi6200-infocube
  keydate LIKE bapi6200-keydate
EXPORTING
  return LIKE bapiret2
TABLES
  selection STRUCTURE bapi6200sl
  characteristics STRUCTURE bapi6200fd
  keyfigures STRUCTURE bapi6200fd
  data STRUCTURE bapi6100da
EXCEPTIONS
  communication_failure
  system_failure
```

Listing 4.16 Interface of the VirtualProvider (RFC Packing)

This is the standard case, which should be used whenever possible. The interface is implemented in function module `BAPI_INFOCUBE_READ_REMOTE_DATA`, which you can use as a template module. It also contains a comprehensive documentation.

Because the parameters can't be derived from the interface without difficulty, we briefly describe them here.

- `INFOCUBE` is the name of the VirtualProvider.
- `KEYDATE` is the key date on which the query is run and on which the master data should be read. However, you'll see further on that you have to read the navigation attributes by yourself. In doing so, you don't have to adhere to the key date, but you should do so to avoid confusing the reader. You can implement different key dates for different navigation attributes.
- `RETURN` is the standard return structure of each BAPI. BAPIs are Business Application Interfaces and contain the SAP Business Logic encapsulated in function modules. You should make sure that the function module to be implemented should behave like a BAPI. In particular, it should not end with an X message,

which causes the program to abort. Instead, the function module should always provide a clean return and, if necessary, an abort message. Otherwise, the user will have to wait in front of the Business Explorer (BEx) screen because the system isn't able to detect that the data-retrieval process in SAP NetWeaver BW has terminated.

- `SELECTION` contains the selections that are defined in the query. Variables have already been replaced with input or replacements. The structure of the table is `EXPRESSION`, `INFOOBJECT`, `SIGN`, `OPTION`, `LOW`, `HIGH`.

  The most important field here is the `EXPRESSION` field: If the value of this field is 0, the corresponding restriction is a global filter. If the value isn't 0, the restriction refers to a row, column, or cell reference. This is of decisive importance for data selections that must be implemented. It makes a big difference whether in a query a column is restricted to the country "USA" and the sales channel "Internet," or a query contains two columns and one of them is restricted to the country "USA" while the other is restricted to the sales channel "Internet."

  If several values exist for `EXPRESSION`, the logic to be implemented requires that all those records from the database are read to which the restriction `E0` and at least one of the restrictions `E1` through `En` apply. The documentation for function module `BAPI_INFOCUBE_READ_REMOTE_DATA` describes this logic in the following way:

  ```
  E0 AND ( E1 OR E2 OR E3 ... OR En )
  ```

  A better way of presenting this logic is to display it as follows:

  ```
  ( E0 AND E1 ) OR
  ( E0 AND E2 ) OR
  ( E0 AND E3 ) ... OR
  ( E0 AND En ).
  ```

  This way, you can regard each statement as a separate line or column of a query.

- `CHARACTERISTICS` and `KEYFIGURES` are two tables that contain the characteristics and key figures used in the query. Both tables have the same structure: `INFOOBJECT`, `DATATYPE`, `DECIMALS`, `SIGN`, `LENGTH`, `OFFSET`, `LOWERCASE`. The `LENGTH` and `OFFSET` fields are particularly important here. These fields indicate in which position and at what length the InfoObject can be found in the return structure.

  As already mentioned, not only are the characteristics transferred in the `CHARACTERISTICS` table but also the navigation attributes used in the query.

- DATA is the table that contains the return values for the OLAP processor. This table consists of only the CONTINUATION and DATA fields. Because it's a BAPI, the length of the DATA field can't be variable; it always consists of 250 characters.

  If the return structure contains more than 250 characters, it must be distributed to several rows in the DATA table. To do this, you can use the CONTINUATION field. Each line of a data record except for the last one is assigned the value X in this field. The field remains empty in the last record. Therefore, it must be interpreted as "Data record will continue in the next row."

The class CL_RSDRV_REMOTE_IPROV_SRV contains some useful methods and services for implementing this BAPI. This class is predominantly used to implement the BAPI for reading data from a database table.

Case 2: RFC Packing and SID Support Are Not Set

If neither RFC Packing nor SID Support is set, the interface is structured as shown in Listing 4.17.

```
IMPORTING
  i_infoprov TYPE rsinfoprov
  i_keydate TYPE rrsrdate
  i_th_sfc TYPE rsdri_th_sfc
  i_th_sfk TYPE rsdri_th_sfk
  i_t_range TYPE rsdri_t_range
  i_tx_rangetab TYPE rsdri_tx_rangetab
  i_first_call TYPE rs_bool
  i_packagesize TYPE i
EXPORTING
  e_t_data TYPE standard table
  e_end_of_data TYPE rs_bool
  e_t_msg TYPE rs_t_msg
```

Listing 4.17 Interface of the VirtualProvider (Neither RFC Packing nor SID Support Are Set)

Instead of the BAPI structures, this interface contains the structures that are used internally by SAP NetWeaver BW. Whereas in Case 1, the function module is called only once, the generated function module is called several times in this implementation; that is, it's called until the E_END_OF_DATA parameter is set to RS_TRUE. For each call, I_PACKAGESIZE records are expected. However, this default setting comes from system customizing and can be ignored. If so many data records must be returned that the DATA table described in Listing 4.16 would significantly affect system performance or cause the main memory to overflow, you should consider implementing the variant described in Listing 4.17.

Case 3: RFC Packing Is Not Set; SID Support Is Set

In all other cases, that is, if RFC PACKING isn't set and SID SUPPORT is set, the variant is referred to as "internal," and it indicates that the interface can be modified without a warning. For this reason, you should use Case 1. For the intrepid among you, however, Listing 4.18 contains the interface for Case 3.

```
IMPORTING
  i_infoprov TYPE rsinfoprov
  i_keydate TYPE rrsrdate
  i_th_sfc TYPE rsdd_th_sfc
  i_th_sfk TYPE rsdd_th_sfk
  i_tsx_seldr TYPE rsdd_tsx_seldr
  i_first_call TYPE rs_bool
  i_packagesize TYPE i
EXPORTING
  e_t_data TYPE standard table
  e_end_of_data TYPE rs_bool
  e_t_msg TYPE rs_t_msg
```

Listing 4.18 Interface of the VirtualProvider (RFC Packing Is Not Set, SID Support Is Set)

When implementing this, you should note that SID SUPPORT doesn't mean that you will be supported in defining the SIDs. It means that in addition to the characteristic values, you must also return the SIDs for the data records. This variant is used by the SAP development team, for instance, to implement additional logic in SAP SEM-BCS. For each InfoProvider that stores data, SAP SEM-BCS creates a VirtualProvider in which reporting takes place. This VirtualProvider also uses a corresponding service that reads the InfoProvider and the SIDs at the same time. This means the OLAP processor doesn't need to further determine the SIDs afterward.

4.3.2 Do's and Don'ts for the Implementation of the Service

Similar to the implementation of virtual key figures and characteristics, the implementation of a VirtualProvider is critical because it's called for each query. For this reason, you should absolutely adhere to the following restrictions:

- If your DataSource isn't a small table containing some 1,000 data records, you should read only what has been transferred in the query.
- You should read only the requested characteristic values and, if possible, have those records aggregated by the database.
- Similarly, you should read only the required key figures from the database. Especially when the query is run, every additional byte is inconvenient, particularly if the query is used by many users. Nothing is more annoying in the BW system than having users wait too long for a query to be run. This immediately decreases the degree of acceptance of the system as a whole.

It's relatively easy to read data from master data tables and DataStore objects because this data is stored in flat tables, and every developer can implement the `SELECT` statements by himself, regardless of optimization aspects. But what can you do if the data is stored not in a DataStore object but in an InfoCube? You have two options: loading the data into a DataStore object or into a standard SAP function module.

If you don't need the entire InfoCube but only aggregated data that refers to a relatively small number of data records (in SAP NetWeaver BW, this means approximately 100,000 records, depending on the design of the system and the structure of the data), you can create a DataStore object with the required structure, load the data into the DataStore via an export DataSource, and obtain the required aggregation. To do this, however, you need additional hard disk space. But why should you store the data once again when it's already stored in a form that's optimized for database access, namely the star schema of the InfoCube? That's why you must find an efficient way to access the contents of the InfoCube.

One option is to use function module `RSDRI_INFOPROV_READ`. Although this function module has not been officially released by SAP, it has been used successfully in the past, even for larger quantities of data. This function module has several advantages. You can assign the required characteristics and key figures to it, and the function module automatically carries out a data aggregation. Further, it uses the parameters assigned to it to automatically select an aggregate, which it then uses for data selection. It's this feature, in particular, that enables you to build VirtualProviders that access InfoProviders containing millions of data records and still return a query result within a couple of seconds.

The interface of the function module is described in Listing 4.19.

```
FUNCTION rsdri_infoprov_read .
*"----------------------------------------
*"*"Local interface:
*"  IMPORTING
*"     REFERENCE(I_INFOPROV)
*"               TYPE  RSINFOPROV
*"     REFERENCE(I_TH_SFC)
*"               TYPE  RSDRI_TH_SFC
*"     REFERENCE(I_TH_SFK)
*"              TYPE  RSDRI_TH_SFK
*"     REFERENCE(I_T_RANGE)
*"        TYPE  RSDRI_T_RANGE OPTIONAL
*"     REFERENCE(I_TH_TABLESEL)
*"        TYPE  RSDRI_TH_SELT OPTIONAL
*"     REFERENCE(I_T_RTIME)
*"        TYPE  RSDRI_T_RTIME OPTIONAL
*"     VALUE(I_REFERENCE_DATE)
*"           TYPE  RSDRI_REFDATE
*"           DEFAULT SY-DATUM
*"     VALUE(I_ROLLUP_ONLY)
*"        TYPE  RS_BOOL DEFAULT RS_C_TRUE
*"     REFERENCE(I_T_REQUID)
*"          TYPE  RSDR0_T_REQUID OPTIONAL
*"     VALUE(I_SAVE_IN_TABLE)
*"          TYPE  RSDRI_SAVE_IN_TABLE
*"          DEFAULT SPACE
*"     VALUE(I_TABLENAME)
*"        TYPE  RSDRI_TABLENAME OPTIONAL
*"     VALUE(I_SAVE_IN_FILE)
*"          TYPE  RSDRI_SAVE_IN_FILE
*"          DEFAULT SPACE
*"     VALUE(I_FILENAME)
*"          TYPE  RSDRI_FILENAME OPTIONAL
*"     VALUE(I_PACKAGESIZE)
*"          TYPE  I DEFAULT 1000
*"     VALUE(I_MAXROWS) TYPE  I DEFAULT 0
*"     VALUE(I_AUTHORITY_CHECK)
*"          TYPE  RSDRI_AUTHCHK DEFAULT
*"          DEFAULT RSDRC_C_AUTHCHK-READ
*"     VALUE(I_CURRENCY_CONVERSION)
*"          TYPE  RSDR0_CURR_CONV
*"          DEFAULT 'X'
*"     VALUE(I_USE_DB_AGGREGATION)
*"          TYPE  RS_BOOL
*"          DEFAULT RS_C_TRUE
*"     VALUE(I_USE_AGGREGATES)
*"          TYPE  RS_BOOL
*"          DEFAULT RS_C_TRUE
*"     VALUE(I_READ_ODS_DELTA)
*"        TYPE RSDRI_CHANGELOG_EXTRACTION
*"        DEFAULT  RS_C_FALSE
*"     VALUE(I_CALLER)
*"          TYPE  RSDRS_CALLER
*"          DEFAULT RSDRS_C_CALLER-RSDRI
*"     VALUE(I_DEBUG)
*"          TYPE  RS_BOOL
*"          DEFAULT RS_C_FALSE
*"     VALUE(I_CLEAR)
*"          TYPE  RS_BOOL
*"          DEFAULT RS_C_FALSE
*"  EXPORTING
*"     REFERENCE(E_T_DATA)
*"              TYPE  STANDARD TABLE
*"     VALUE(E_END_OF_DATA)
*"          TYPE  RS_BOOL
*"     VALUE(E_AGGREGATE)
*"          TYPE  RSINFOCUBE
*"     VALUE(E_SPLIT_OCCURRED)
*"          TYPE  RSDR0_SPLIT_OCCURRED
*"     REFERENCE(E_T_MSG) TYPE  RS_T_MSG
*"  CHANGING
*"     REFERENCE(C_FIRST_CALL)
*"              TYPE  RS_BOOL
*"  EXCEPTIONS
*"      ILLEGAL_INPUT
*"      ILLEGAL_INPUT_SFC
*"      ILLEGAL_INPUT_SFK
*"      ILLEGAL_INPUT_RANGE
*"      ILLEGAL_INPUT_TABLESEL
*"      NO_AUTHORIZATION
*"      ILLEGAL_DOWNLOAD
*"      ILLEGAL_TABLENAME
*"      TRANS_NO_WRITE_MODE
*"      INHERITED_ERROR
*"      X_MESSAGE
```

Listing 4.19 Interface of Function Module RSDRI_INFOPROV_READ

At first glance, this table appears shocking. What is important in this function module is the IMPORTING parameter I_INFOPROV that contains the name of the InfoProvider plus I_TH_SFC and I_TH_SFK. These two tables contain the characteristics (I_TH_SFC) and key figures (I_TH_SFK) that are to be read from the InfoProvider. Most of the other parameters are self-explanatory and can be ignored. It's also advisable to set a breakpoint in the function module and to call Transaction LISTCUBE. Overall, the function module is very powerful; try it for yourself.

If you want to use the function module in the VirtualProvider, you will certainly receive an error message telling you that the function module must not be called recursively. The reason for this is that the same function module is used to read data from the VirtualProvider. In that case, you should create an RFC-enabled function module that wraps the part of the interface you need. Then you can call the newly created function module via RFC.

Nevertheless, it's clearly preferable that your users are flexible enough to avoid using VirtualProviders with services. Usually these InfoProviders require more maintenance work, and it's more time-consuming to extend queries than to use normal InfoCubes or MultiProviders. If possible, you should always try to convince your customer of a solution that meets his requirements by using virtual key figures and characteristics. Always remember that it's possible to carry out some of the calculations in a Microsoft Excel worksheet.

4.4 BAdI SMOD_RSR00004

Finally, let's look at BAdI SMOD_RSR00004, which originated from user exit RSR00004 and was automatically migrated from it. This BAdI is called in the report-report interface, where it's used to carry out automatic adjustments. You can use this function, for example, to modify an InfoObject.

For instance, if you display those cost centers in a report that services a specific purchase order, and if you want to display a cost report for the cost center, the cost center will be contained in InfoObject 0PCOST_CTR (Partner Cost Center) in the first report. In the second report, it will be located in InfoObject 0COSTCENTER. This kind of mapping can't be stored as such in the report-report interface. For this reason, you can use the BAdI at this point to carry out the jump anyway. Jumps from a current monthly report to a report that shows the cumulative values from the beginning of the year require this BAdI.

The BAdI contains two methods: EXIT_SAPLRSBBS_001 and EXIT_SAPLRSBBS_002. Both methods can be used to individually customize the field mapping, which is defined in Transaction RSBBS in the report-report interface. The only difference between them is the time at which the jump is called. The EXIT_SAPLRSBBS_001 method is called for normal jumps within SAP NetWeaver BW, whereas the EXIT_SAPLRSBBS_002 method is used for jumps within SAP ERP.

The EXIT_SAPLRSBBS_001 method has the signature shown in Table 4.4. The table contains several old friends of ours, such as the fiscal year variant I_PERIV, the query information for sender and recipient, I_S_RKB1D_SENDER and I_S_RKB1D, and the structure E_S_RETURN that is used to return status information.

Type	Parameter	Type
I	I_PERIV	RRO01_S_RKB1F-PERIV
I	I_S_RKB1D	RSR_S_RKB1D
I	I_S_RKB1D_SENDER	RSR_S_RKB1D
I	I_THX_RECEIVER	RSBBS_THX_MAPPING
I	I_THX_SENDER	RSBBS_THX_MAPPING
E	E_S_RETURN	BAPIRET2
E	E_THX_MAPPING	RSBBS_THX_MAP_BY_EXIT
I = Importing, E = Exporting		

Table 4.4 Signature of the EXIT_SAPLRSBBS_001 Method

However, the most important tables are I_THX_RECEIVER and I_THX_SENDER, which contain the mapping from the definition of the report-report interface and E_THX_MAPPING that is supposed to contain the new mapping.

As an example, suppose you now want to implement the jump from a profitability data report (SALES_COPA) into a report that contains the corresponding sales orders (SALESORDERS). You want to map the posting periods (0FISCPER) to the corresponding calendar days (0CALDAY). This method then appears as shown in Listing 4.20. First, it is checked whether the correct reports are used. Then, the selected periods of the result data report are read and converted into the corresponding calendar days, which are then inserted into the mapping table.

```
METHOD IF_EX_SMOD_RSR00004~EXIT_SAPLRSBBS_001 .
DATA: l_s_thx_sender
      TYPE LINE OF rsbbs_thx_mapping,
      l_s_thx_mapping TYPE LINE OF rsbbs_thx_map_by_exit,
      l_s_range       TYPE rrrangesid,
      l_d_year        TYPE /bi0/oifiscyear,
      l_d_per3        TYPE /bi0/oifiscper3,
      l_d_date_from   LIKE SY-DATUM,
      l_d_date_to     LIKE SY-DATUM.
* Check queries
  IF i_s_rkb1d_sender-compid <> ‘SALES_COPA’.
    EXIT.
  ENDIF.
  IF i_s_rkb1d-compid <> ‘SALESORDERS’.
    EXIT.
  ENDIF.
* Read period in sender
  READ TABLE i_thx_sender
       INTO l_s_thx_sender
       WITH KEY fieldnm = ‘0FISCPER’.
  IF SY-SUBRC <> 0.
*   No period found, hence no restriction
    EXIT.
  ENDIF.
* Start of actual mapping: define InfoObject
* in recipient report
  l_s_thx_mapping-fieldnm_to = ‘0CALDAY’.
  l_s_thx_mapping-fieldtp_to = RSBBS_C_FIELDTP-INFOOBJECT.
  l_s_thx_mapping-dtelnm     = ‘/BI0/OICALDAY’.
  l_s_thx_mapping-domanm     = ‘/BI0/OCALDAY’.
* Transfer data
  LOOP AT l_s_thx_sender-range
       INTO l_s_range.
    IF l_s_range-opt = ‘EQ’ OR
       l_s_range-opt = ‘NE’ OR
       l_s_range-opt = ‘LT’ OR
       l_s_range-opt = ‘LE’.
      l_s_range-high = l_s_range-low.
    ENDIF.
*   Convert periods
    l_d_year = l_s_range-low(4).
    l_d_per3 = l_s_range-low+4(3).
    CALL FUNCTION ‘FIRST_DAY_IN_PERIOD_GET’
```

```
      EXPORTING
        GJAHR  = l_d_year
        PERIV  = i_periv
        POPER  = l_d_per3
      IMPORTING
        DATUM  = l_d_date_from.
    IF NOT ( l_s_range-high IS INITIAL ).
      l_d_year = l_s_range-high(4).
      l_d_per3 = l_s_range-high+4(3).
      CALL FUNCTION 'FIRST_DAY_IN_PERIOD_GET'
        EXPORTING
          GJAHR  = l_d_year
          PERIV  = i_periv
          POPER  = l_d_per3
        IMPORTING
          DATUM  = l_d_date_to.
    ENDIF.
*   Compose correct interval
    CASE l_s_range-opt.
      WHEN 'EQ' OR 'BT'.
        l_s_range-opt = 'BT'.
        l_s_range-low = l_d_date_from.
        l_s_range-high = l_d_date_to.
      WHEN 'NE' OR 'NB'.
        l_s_range-opt = 'NB'.
        l_s_range-low = l_d_date_from.
        l_s_range-high = l_d_date_to.
      WHEN 'LE' OR 'GT'.
*       <= includes the period, thus last day
*       of period analogous >
        l_s_range-low = l_d_date_to.
      WHEN 'LT' OR 'GE'.
*       < does not include period, thus first day
*       of period analogous >=
        l_s_range-low = l_d_date_from.
    ENDCASE.
    APPEND l_s_range TO l_s_thx_mapping-range.
  ENDLOOP.
  INSERT l_s_thx_mapping INTO TABLE e_thx_mapping.
ENDMETHOD.
```

Listing 4.20 Example of Method EXIT_SAPLRSBBS_001

The interface of the second method is slightly different, but it works in a way similar to the first one. Table 4.5 contains this interface.

Type	Parameter	Type
I	I_COMPID	RSZCOMPID
I	I_INFOCUBE	RSD_INFOCUBE
I	I_S_RECEIVER	RSTIREC
I	I_THX_MAPPING	RSBBS_THX_MAPPING
E	E_THX_MAPPING	RSBBS_THX_MAP_BY_EXIT
I = Importing, E = Exporting		

Table 4.5 Signature of the EXIT_SAPLRSBBS_002 Method

Contrary to the first method, the mapping here is exported only on the side of the sender. Instead of the entire query information, only the query name (`I_COMPID`) and the InfoProvider (`I_INFOCUBE`) are transferred. The information on the source system is contained in structure `I_S_RECEIVER`, which contains all necessary information such as the field `I_S_RECEIVER-RONAM`. This contains a transaction name in case it was selected as a jump target. The actual tables used for the original mapping and the return are identical. This means that the example in Listing 4.20 can be used virtually unmodified for a jump into a source system. You should make sure, however, that you adapt the accesses to `i_s_rkb1d_sender` here.

In general, these methods are very useful if you need to resolve specific problems when jumping from one report into another. However, because this subject usually doesn't attract as much attention as it deserves, this BAdI is probably abandoned in most BW systems. Even though the actual implementation isn't difficult, you should nevertheless ask yourself—as you should with all exits—whether it's worth the implementation effort and, above all, the future maintenance work.

4.5 Implementing Own Read Routines for Master Data

SAP NetWeaver 7.0 also includes a new exit. With this release, you can store your own master data read routines in InfoObjects. This is useful if you have a lot of master data for which you want to maintain the changes in texts and attributes per upload because this data is already contained in other tables, or because you can read the data from a table, domain, and so on without loading it again. Depending on the user authorizations, you can also display or delete specific attributes.

To implement a master data read routine, you need to enter a master data read class in the InfoObject on the MASTER DATA/TEXTS tab. A master data read class is a class that implements interface `IF_RSMD_RS_ACCESS`. If interface `IF_RSMD_RS_GENERIC` is integrated additionally, you can transfer parameters to the master data read class. This option is very useful, for example, for the master data read class `CL_RSMD_RS_GENERIC_DOMAIN`, which is delivered by SAP and which you can use to populate the values and texts of an InfoObject automatically with the fixed values of a domain.

The following sections describe how you can use such a class. For this purpose, you implement a class that reads the master data of the existing InfoObject `0COSTCENTER` and hides all cost centers that don't contain `'X'` in the attribute `IST_HKST`. You can then use this class in InfoObject `HAUPT_KST`, for example, to display only those cost centers in the input help during the query execution and creation that are selected as primary cost centers. The benefit of this method is that the data is maintained in duplicate, and you don't require any additional load processes to transfer the data from one InfoObject to the other.

The master data read classes are used at different locations since Release 7.0, in particular to provide master data for InfoObjects that are defined by SAP-specific tables, for example, the master data for the client (`0CLIENT`) or the fiscal year variant (`0FISCVARNT`). Moreover, there are numerous InfoObjects that access fixed values of domains or contents of tables via generic master data read classes, for example, `0INM_MFNL` or `0RSPL_LOCK`. The master data read classes used here, `CL_RSMD_RS_GENERIC_DOMAIN` or `CL_RSMD_RS_GENERIC_TABLE`, can also be used for your own InfoObjects, in particular if the contents of the InfoObjects originate from your own tables or other custom developments. You can thus avoid loading data into InfoObjects or double maintenance of data.

4.5.1 Creating a Master Data Read Class

You can create a master data read class in the Object Navigator (Transaction SE80). For this purpose, go to the INTERFACES tab in the class interface and enter interface `IF_RSMD_RS_ACCESS` in the table. Save the class. In the object list, you now see six methods you can implement (see Table 4.6).

Con. No.	Method	Description
1	CREATE	Static method; returns the reference to the object
2	GET_ATTRIBUTES	Reads the master data attributes
3	GET_CAPABILITIES	Returns information about the texts and time dependencies to be delivered
4	GET_TEXT	Reads texts for master data
5	GET_VALUES	Returns the existing values
6	SET_KEY_DATE	Sets the key data for time-dependent texts and master data

Table 4.6 Methods of the IF_RSMD_RS_ACCESS Interface

The following sections describe the individual methods without any examples and then discuss the complete implementation of a class.

CREATE Method

The `CREATE` method is a static method that is called by the calling program to retrieve a handler to the object. Consequently, the method must generate and return an object of the class. Additionally, you can fill buffer tables if these have not been populated yet in order to avoid repeated identical database accesses. It's usually not necessary to overwrite the standard implementation, which is described in Listing 4.21.

```
METHOD if_rsmd_rs_access~create.
  DATA l_r_rs_bw_spec TYPE REF TO cl_rsmd_rs_bw_spec.
  IF i_clnm IS NOT INITIAL.
    CREATE OBJECT r_r_rs_access TYPE (i_clnm)
      EXPORTING i_chanm = i_chanm
                i_infoprov = i_infoprov
                i_langu = i_langu.
  ELSE.
    CREATE OBJECT l_r_rs_bw_spec
        EXPORTING i_chanm = i_chanm
                  i_infoprov = i_infoprov
                  i_langu = i_langu.
    r_r_rs_access ?= l_r_rs_bw_spec.
  ENDIF.
ENDMETHOD.
```

Listing 4.21 Example of the IF_RSMD_RS_ACCESS~CREATE Method

As you can see here, an object of the `CL_RSMD_RS_BW_SPEC` class is created in case of doubt, that is, for all InfoObjects for which no implementation is provided. This class is delivered by SAP and is used as the standard for all InfoObjects that don't implement their own master data read routines. If required, you can check the implementation of this class to see how the reading from the master data tables is done.

SET_KEY_DATE Method

The `SET_KEY_DATE` method is called during the execution of a query to transfer the query key date. Here, an instance attribute, `P_KEY_DATE`, is populated from the called parameter. In principle, a redefinition isn't required; the only exception is that you want to read different characteristics at different points in time. However, no application is known for which this is required.

GET_CAPABILITIES Method

The only parameter of the `GET_CAPABILITIES` method is a returning parameter `R_S_CAPABILITIES`. It contains seven individual flags (see Table 4.7) that control the properties of the InfoObject. These must be filled completely.

Name of the Flag	Description	FALSE	TRUE
`NOVALFL`	Characteristic doesn't contain a check table.	‘ ’	‘X’
`TXTTABFL`	Text table exists.	‘0’	‘1’
`TXTTIMFL`	Text table is time-dependent.	‘0’	‘1’
`NOLANGU`	Texts are language-independent.	‘ ’	‘X’
`TXTSHFL`	Short text exists.	‘ ’	‘X’
`TXTMDFL`	Medium text exists.	‘ ’	‘X’
`TXTLGFL`	Long text exists.	‘ ’	‘X’

Table 4.7 Fields of Parameter R_S_CAPABILITIES

For InfoObject 0FISCPER, the method is implemented as shown in Listing 4.22.

```
METHOD IF_RSMD_RS_ACCESS~GET_CAPABILITIES.
  r_s_capabilities-novalfl   = rs_c_false.
  r_s_capabilities-txttabfl  = rsd_c_cnvfl-true.
  r_s_capabilities-txttimfl  = rsd_c_cnvfl-false.
  r_s_capabilities-nolangu   = rs_c_false.
  r_s_capabilities-txtshfl   = rs_c_true.
  r_s_capabilities-txtmdfl   = rs_c_true.
  r_s_capabilities-txtlgfl   = rs_c_false.
ENDMETHOD.
```

Listing 4.22 Example of the IF_RSMD_RS_ACCESS~GET_CAPABILITIES Method

You can see the difference between the various flags.

If you don't want to implement the flags directly, you can also fill the corresponding fields from table RSDCHABAS which lists all InfoObjects. The implementation could look like the one shown in Listing 4.23.

```
 METHOD IF_RSMD_RS_ACCESS~GET_CAPABILITIES.
  SELECT SINGLE * FROM RSDCHABAS
         INTO CORRESPONDING FIEDLS OF r_s_capabilities
         WHERE CHABASNM = ‘HAUPT_KST’
         AND    OBJVERS = ‘A’.
ENDMETHOD.
```

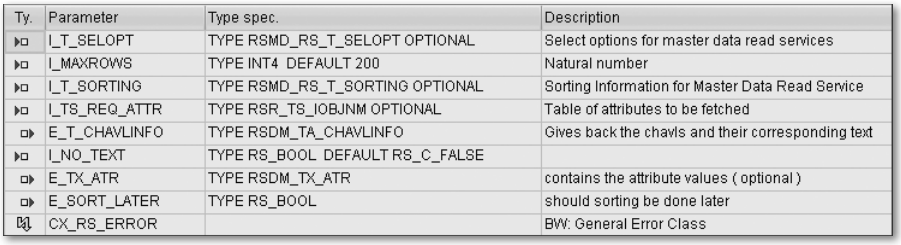
Listing 4.23 Generic Implementation of the IF_RSMD_RS_ACCESS~GET_CAPABILITIES Method

The benefit here is that you can change the individual settings directly in the InfoObject. For this purpose, the following methods must be implemented generically so that the correct values are returned depending on the set switches.

GET_VALUES Method

The GET_VALUES method is the central method of the interface. In this method, values, texts, and, if necessary, the required attributes are returned. The method's signature is shown in Figure 4.7.

In this method, you must implement three things. First, you must collect all allowed values. For this purpose, SAP usually creates a _VALUES_GENERATE method, which returns the desired values in table E_T_CHAVLINFO. If you implement the class yourself, you should use the same name for the method.



Ty.	Parameter	Type spec.	Description
▶□	I_T_SELOPT	TYPE RSMD_RS_T_SELOPT OPTIONAL	Select options for master data read services
▶□	I_MAXROWS	TYPE INT4 DEFAULT 200	Natural number
▶□	I_T_SORTING	TYPE RSMD_RS_T_SORTING OPTIONAL	Sorting Information for Master Data Read Service
▶□	I_TS_REQ_ATTR	TYPE RSR_TS_IOBJNM OPTIONAL	Table of attributes to be fetched
□▶	E_T_CHAVLINFO	TYPE RSDM_TA_CHAVLINFO	Gives back the chavls and their corresponding text
▶□	I_NO_TEXT	TYPE RS_BOOL DEFAULT RS_C_FALSE	
□▶	E_TX_ATR	TYPE RSDM_TX_ATR	contains the attribute values (optional)
□▶	E_SORT_LATER	TYPE RS_BOOL	should sorting be done later
⚡	CX_RS_ERROR		BW: General Error Class

Figure 4.7 Signature of the GET_VALUES Method

The tables, which must be filled in the method, are an important aspect in the method's interface. E_T_CHAVLINFO contains the characteristic values and the corresponding texts, while E_TX_ATR comprises the corresponding attributes. To maintain performance you should note the following:

- Table I_T_SELOPT contains selection options. Only characteristic values meeting the conditions should be returned.
- Parameter I_MAXROWS contains the number of values returned. If possible, this number of values should not be exceeded.
- Table I_TS_REQ_ATR contains the requested attributes. No additional attributes should be returned.

When the values have been determined, you must enrich them with texts and attributes. To do so, you can call methods GET_ATTRIBUTES and GET_TEXT, which are described in the following. A possible implementation for this is shown in Listing 4.24. First, characteristic values are read, then texts, and finally attributes.

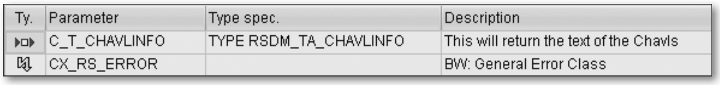
```
METHOD if_rsmd_rs_access~get_values.

* First save the transferred parameters
* to make them available for other methods.
  o_t_selopt         = i_t_selopt.
  o_maxrows          = i_maxrows.
  o_t_sorting        = i_t_sorting.

  TRY.
      CALL METHOD _values_generate
* You must implement this method yourself. You
* must only fill the fields, C_CHAVL and I_READ_MODE.
        IMPORTING
          e_t_chavlinfo = e_t_chavlinfo.
```

```
    CATCH cx_rs_error .
      x_message('IF_RSMD_RS_ACCESS~GET_VALUES-01').
  ENDTRY.

***********************************************************
* Now, the texts are read.
***********************************************************

  TRY.
      CALL METHOD if_rsmd_rs_access~get_text
* This method must be implemented/overwritten.
* Create an empty implementation in case of doubt.
        CHANGING
          c_t_chavlinfo = e_t_chavlinfo.
    CATCH cx_rs_error .
      x_message('IF_RSMD_RS_ACCESS~GET_VALUES-02').
  ENDTRY.

***********************************************************
**  Now, the attributes are read, if they have been
*   requested.
***********************************************************

  IF i_ts_req_attr IS NOT INITIAL.
* Have attributes been requested?
    TRY.
        CALL METHOD if_rsmd_rs_access~get_attributes
* This method must be implemented/overwritten.
* Create an empty implementation in case of doubt.
          EXPORTING
            i_ts_req_attr = i_ts_req_attr
            i_t_chavlinfo = e_t_chavlinfo
          RECEIVING
            r_tx_atr      = e_tx_atr.
      CATCH cx_rs_error .
        x_message('IF_RSMD_RS_ACCESS~GET_VALUES-03').
    ENDTRY.
  ENDIF.
ENDMETHOD.
```

Listing 4.24 Example of the IF_RSMD_RS_ACCESS~GET_VALUES Method

GET_TEXT Method

In the GET_TEXT method, you must enrich the characteristic attributes provided in the table with corresponding texts. Hence, the signature of this method is simple (see Figure 4.8).



Ty.	Parameter	Type spec.	Description
	C_T_CHAVLINFO	TYPE RSDM_TA_CHAVLINFO	This will return the text of the Chavls
	CX_RS_ERROR		BW: General Error Class

Figure 4.8 Signature of the GET_TEXT Method

The only parameter of this method is table C_T_CHAVLINFO. This table contains all characteristic values, texts, and other information. The corresponding structure is shown in Table 4.8.

Con. No.	Field Name	Type	Description
1	I_SID	RSD_SID	SID of the characteristic value
2	C_CHAVL	RSD_CHAVL	Characteristic value
3	C_NIOBJNM	RSD_IOBJNM	Name of the InfoObject
4	C_HIEID	RSHIEID	Hierarchy ID
5	I_TABIX	I	Row index
6	E_EXIST	RS_BOOL	Indicates whether value exists
7	E_CHATEXTS	RS_S_TXTSML	Short, medium, or long text
8	C_RC	SYSUBRC	Return value
9	I_READ_MODE	RSDM_READ_ MODE	Read mode

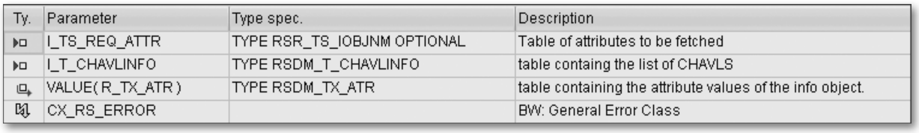
Table 4.8 Fields of Table C_T_CHAVLINFO

In the GET_TEXT method, it's important to fill structure E_CHATEXTS with the fields TXTSH (short text), TXTMD (medium-length text), and TXTLG (long text). You must fill the fields that were indicated in method GET_CAPABILITIES.

GET_ATTRIBUTES Method

The GET_ATTRIBUTES method fills table R_TX_ATTR. This table contains the corresponding attributes for each characteristic value that is transferred in table I_T_CHAVLINFO.

The requested attributes are transferred to the method in table I_TS_REQ_ATTR. The method's signature is shown in Figure 4.9.



Ty.	Parameter	Type spec.	Description
▶□	I_TS_REQ_ATTR	TYPE RSR_TS_IOBJNM OPTIONAL	Table of attributes to be fetched
▶□	I_T_CHAVLINFO	TYPE RSDM_T_CHAVLINFO	table containg the list of CHAVLS
	VALUE(R_TX_ATR)	TYPE RSDM_TX_ATR	table containing the attribute values of the info object.
	CX_RS_ERROR		BW: General Error Class

Figure 4.9 Signature of the GET_ATTRIBUTES Method

You already know the structure of table I_T_CHAVLINFO from the GET_TEXT method. Table I_TS_REQ_ATTR is a simple table that contains an InfoObject for each row. Table R_TX_ATTR contains the characteristic value, CHANM, in each row; the SID and the table of the attribute value, T_ATR, which comprises the attribute name ATTRINM; and the attribute value ATTRIVL in each row.

_VALUES_GENERATE Method

The _VALUES_GENERATE (or _VALUE_GENERATE) method isn't contained in interface IF_RSMD_RS_ACCESS. Nevertheless, it's contained in most classes delivered by SAP; for this reason, it's described below.

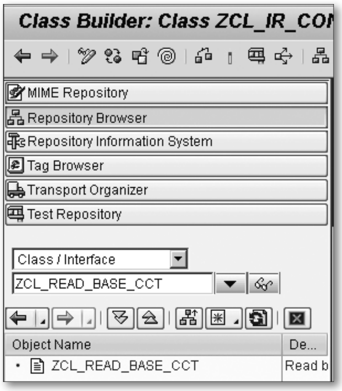
This method is used to initially fill table E_T_CHAVLINFO, which is the only parameter of the method. C_CHAVL is the only field of the structure that must be filled. The corresponding attributes of method GET_VALUES indicate which characteristic values must be delivered and how many.

4.5.2 Sample Implementation of a Master Data Read Class

The following sections describe in detail how you can implement a class for InfoObject HAUPT_KST that automatically contains all master data of the cost centers 0COSTCENTER for which a specific flag (IST_HKST) was set in their master data. Texts and attributes are automatically read from the existing texts and attributes of 0COSTCENTER.

Creating Class ZCL_READ_HAUPT_KST

1. Call Transaction SE80.
2. Select Class/Interface, and enter the class name, "ZCL_READ_BASE_CCT" (see Figure 4.10).

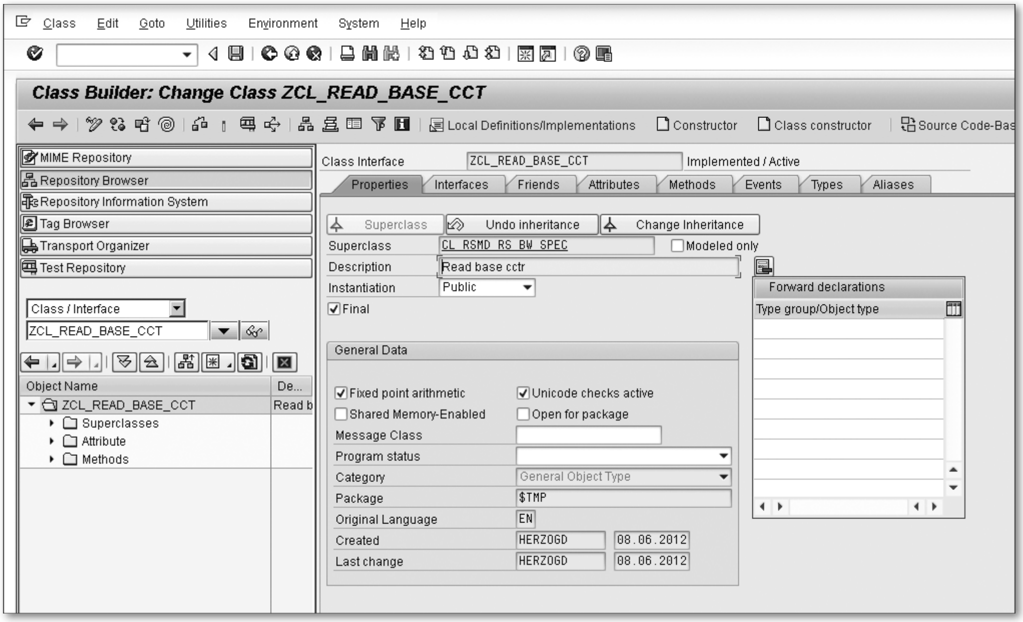


Class / Interface
ZCL_READ_BASE_CCT

Object Name	De...
ZCL_READ_BASE_CCT	Read b

Figure 4.10 Creating Class ZCL_READ_BASE_CCT

3. Confirm that you want to create a new class.
4. The simplest way to receive all required attributes is to inherit class `CL_RSMD_RS_BW_SPEC`. For this purpose, right-click on the class name, `ZCL_READ_BASE_CCT`, select CHANGE, and go to the PROPERTIES tab.
5. Click the SUPERCLASS button, and enter the "CL_RSMD_RS_BW_SPEC" class in the field below (see Figure 4.11).



Class / Interface
ZCL_READ_BASE_CCT

Object Name	De...
ZCL_READ_BASE_CCT	Read b
Superclasses	
Attribute	
Methods	

Class Interface ZCL_READ_BASE_CCT Implemented / Active

Superclass	CL_RSMD_RS_BW_SPEC ☐ Modeled only
Description	Read base cctr
Instantiation	Public
☑ Final	

General Data

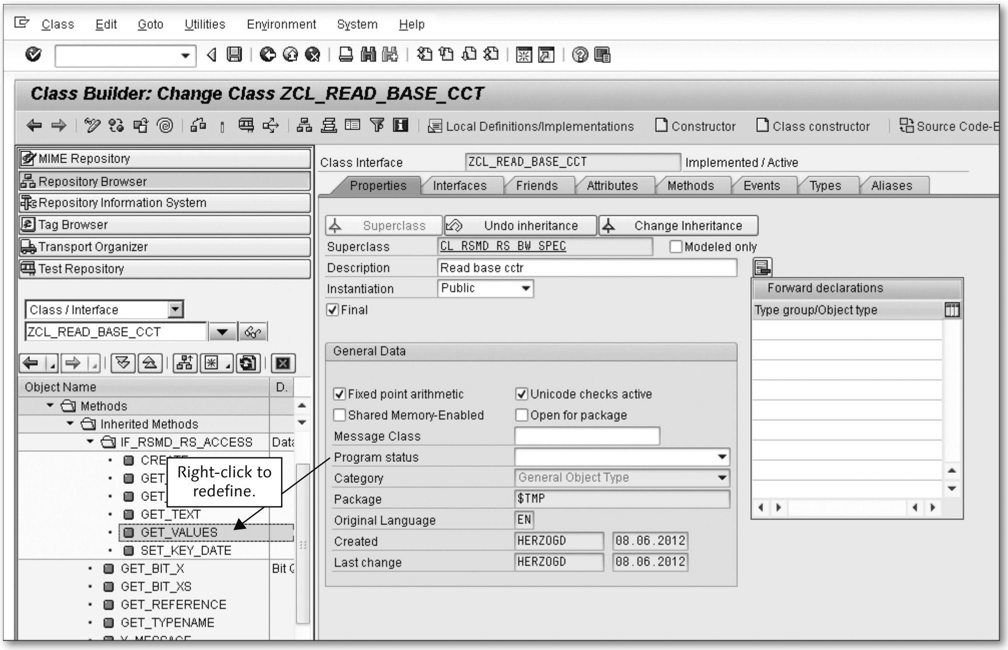
☑ Fixed point arithmetic	☑ Unicode checks active
☐ Shared Memory-Enabled	☐ Open for package
Message Class	
Program status	
Category	General Object Type
Package	$TMP
Original Language	EN
Created	HERZOGD 08.06.2012
Last change	HERZOGD 08.06.2012

Figure 4.11 Specifying the Inheritance in the Class Properties

Creating GET_VALUES Method

To create the `GET_VALUES` method, proceed as follows.

1. Search the `GET_VALUES` method in the left-hand object list under ZCL_READ_BASE_CCT • METHODS • INHERITED METHODS • IF_RSMD_RS_ACCESS.
2. Right-click on the `GET_VALUES` method and select REDEFINE (see Figure 4.12).
3. Select the `IF_RSMD_RS_ACCESS~GET_VALUES` method under ZCL_READ_BASE_CCT • METHODS • REDEFINITIONS, and insert the following code (see Listing 4.25).



Class / Interface
ZCL_READ_BASE_CCT

Class Interface: ZCL_READ_BASE_CCT Implemented / Active

Superclass: CL_RSMD_RS_BW_SPEC ☐ Modeled only

Description: Read base cctr

Instantiation: Public

☑ Final

☑ Fixed point arithmetic ☑ Unicode checks active

☐ Shared Memory-Enabled ☐ Open for package

Message Class:

Program status:

Category: General Object Type

Package: $TMP

Original Language: EN

Created: HERZOGD 08.06.2012

Last change: HERZOGD 08.06.2012

Figure 4.12 Redefining a Method

```
METHOD IF_RSMD_RS_ACCESS~GET_VALUES.

  o_t_selopt          = i_t_selopt.
  o_maxrows           = i_maxrows.
  o_t_sorting         = i_t_sorting.

  TRY.
*********************************************************
* First, the characteristic values of the InfoObject
* HAUPT_KST are determined.
```

```
********************************************************

      CALL METHOD _value_generate
        IMPORTING
          e_t_chavlinfo = e_t_chavlinfo.
    CATCH cx_rs_error .
      x_message('IF_RSMD_RS_ACCESS~GET_VALUES-01').
  ENDTRY.

********************************************************
* Now, the texts for the characteristic values are determined.
********************************************************

  TRY.
      CALL METHOD if_rsmd_rs_access~get_text
        CHANGING
          c_t_chavlinfo = e_t_chavlinfo.
    CATCH cx_rs_error .
      x_message('IF_RSMD_RS_ACCESS~GET_VALUES-02').
  ENDTRY.


*********************************************************
*  Finally, the attributes of the InfoObject
*  HAUPT_KST are determined.
*********************************************************

  IF i_ts_req_attr IS NOT INITIAL.

    TRY.
        CALL METHOD if_rsmd_rs_access~get_attributes
          EXPORTING
            i_ts_req_attr = i_ts_req_attr
            i_t_chavlinfo = e_t_chavlinfo
          RECEIVING
            r_tx_atr      = e_tx_atr.
      CATCH cx_rs_error .
        x_message('IF_RSMD_RS_ACCESS~GET_VALUES-03').
    ENDTRY.

  ENDIF.
ENDMETHOD.
```

Listing 4.25 Comprehensive Example of the IF_RSMD_RS_ACCESS~GET_VALUES Method

Creating _VALUE_GENERATE Method

The _VALUE_GENERATE method is copied from class CL_RSMD_RS_BW_SPEC and corresponds to the _VALUES_GENERATE method used in other classes. It is redefined in a manner analogous to the GET_VALUES method and is created as described in Listing 4.26.

```
METHOD _VALUE_GENERATE.
  DATA: l_s_costcenter TYPE /bi0/mcostcenter,
        l_s_chavlinfo   TYPE rsdm_s_chavlinfo.
* Read data in buffers. In a real implementation
* Table l_s_costcenter should be created as a
* static attribute to the class.
  SELECT * FROM /BI0/MCOSTCENTER INTO l_s_costcenter
        UP TO o_maxrows ROWS
    WHERE DATEFROM <= sy-datum
      AND DATETO   >= sy-datum
      AND /BIC/IST_HKST = ‘X’.
* Set read mode now
    l_s_chavlinfo-i_read_mode = rsdm_c_read_mode-text.
* Because HAUPT_KST is compounded to 0CO_AREA  (analogous to
* 0COSTCENTER), the key must be composed here.
    CONCATENATE l_s_costcenter-co_area
                l_s_costcenter-costcenter
                INTO l_s_chavlinfo-c_chavl.
    INSERT l_s_chavlinfo INTO TABLE e_t_chavlinfo.
  ENDSELECT.
  IF sy-subrc <> 0.
    RAISE EXCEPTION TYPE cx_rs_error.
  ENDIF.
ENDMETHOD.
```

Listing 4.26 Comprehensive Example of the _VALUE_GENERATE Method

Creating GET_ATTRIBUTES Method

Analogous to the previous methods, the GET_ATTRIBUTES method is implemented as follows (see Listing 4.27). Here, the master data of the cost centers is read from the view to attribute table /BI0/MCOSTCENTER. For the sake of simplicity, time-dependency is ignored here, and only the currently valid value is read. Then, the attributes, 0PROFIT_CTR and 0RESP_PERS, are transferred to the return table, R_TX_ATR.

```
METHOD IF_RSMD_RS_ACCESS~GET_ATTRIBUTES.
**TRY.
```

```
*CALL METHOD SUPER->IF_RSMD_RS_ACCESS~GET_ATTRIBUTES
*  EXPORTING
**    i_ts_req_attr =
*    I_T_CHAVLINFO =
*  RECEIVING
*    R_TX_ATR      =
*    .
** CATCH cx_rs_error .
**ENDTRY.

  FIELD-SYMBOLS:
    <f_chavl>  TYPE RSDM_S_CHAVLINFO,
    <f_mcctr>  TYPE /bi0/mcostcenter.

  DATA: l_t_mcctr  TYPE HASHED TABLE
            OF /bi0/mcostcenter
            WITH UNIQUE KEY co_area costcenter
            INITIAL SIZE 0.
  DATA: l_sx_atr TYPE rsdm_sx_atr,
        l_s_atr  TYPE rsdm_s_atr.

* Here as well,  Table l_t_mcctr should be outsourced in a
* static attribute of the class and filled in the
* CREATE routine.
  SELECT * FROM /bi0/mcostcenter INTO TABLE l_t_mcctr
         WHERE dateto   >= sy-datum
           AND datefrom <= sy-datum.

* Now, all characteristics to be filled are read
  LOOP AT i_t_chavlinfo ASSIGNING <f_chavl>.
    READ TABLE l_t_mcctr ASSIGNING <f_mcctr>
       WITH TABLE KEY co_area    = <f_chavl>-c_chavl(4)
                      costcenter = <f_chavl>-c_chavl+4(10).
    IF sy-subrc = 0.
*     Now, Return Table R_TX_ATR is filled
      l_sx_atr-chavl = <f_chavl>-c_chavl.
      REFRESH l_sx_atr-t_atr.
*     Fill profit center
      l_s_atr-attrinm = ‘0PROFIT_CTR’.
      l_s_atr-attrivl = <f_mcctr>-profit_ctr.
      INSERT l_s_atr INTO TABLE l_sx_atr-t_atr.
*     Fill person responsible
      l_s_atr-attrinm = ‘0RESP_PERS’.
      l_s_atr-attrivl = <f_mcctr>-resp_pers.
      INSERT l_s_atr INTO TABLE l_sx_atr-t_atr.
      INSERT l_sx_atr INTO TABLE r_tx_atr.

    ENDIF.
  ENDLOOP.
ENDMETHOD.
```

Listing 4.27 Comprehensive Example of the IF_RSMD_RS_ACCESS~GET_CAPABILITIES Method

Creating GET_TEXT Method

The `GET_TEXT` method is implemented as shown in Listing 4.28. For the determined cost centers, the texts from text table `/BI0/TCOSTCENTER` are read and filled into the fields of structure `E_CHATEXTS`.

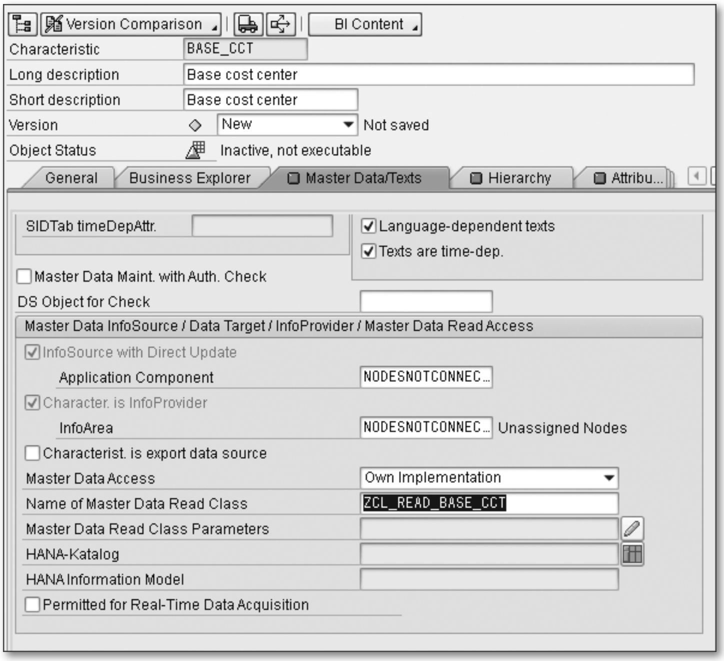
```
METHOD IF_RSMD_RS_ACCESS~GET_TEXT.
**TRY.
*CALL METHOD SUPER->IF_RSMD_RS_ACCESS~GET_TEXT
*  CHANGING
*    C_T_CHAVLINFO =
*    .
** CATCH cx_rs_error .
**ENDTRY.
  FIELD-SYMBOLS:
    <f_chavl>  TYPE RSDM_S_CHAVLINFO,
    <f_tcctr>  TYPE /bi0/tcostcenter.

  DATA: l_t_tcctr  TYPE HASHED TABLE
          OF /bi0/tcostcenter
          WITH UNIQUE KEY co_area costcenter
          INITIAL SIZE 0.

  SELECT * FROM /bi0/tcostcenter INTO TABLE l_t_tcctr
         WHERE dateto   >= sy-datum
           AND datefrom <= sy-datum
           AND langu     = sy-langu.

  LOOP AT c_t_chavlinfo ASSIGNING <f_chavl>.
    READ TABLE l_t_tcctr ASSIGNING <f_tcctr>
       WITH TABLE KEY co_area    = <f_chavl>-c_chavl(4)
                      costcenter = <f_chavl>-c_chavl+4(10).
    IF sy-subrc = 0.
      <f_chavl>-e_chatexts-txtsh  = <f_tcctr>-txtsh.
      <f_chavl>-e_chatexts-txtmd  = <f_tcctr>-txtmd.
    ENDIF.
  ENDLOOP.
ENDMETHOD.
```

Listing 4.28 Comprehensive Example of the IF_RSMD_RS_ACCESS~GET_TEXT Method

4.5.3 Entering the Class in the InfoObject

You must change the InfoObject `BASE_CCT`.

1. Call InfoObject `BASE_CCT`.
2. The MASTER DATA/TEXTS tab contains the field MASTER DATA ACCESS. In this field, select OWN IMPLEMENTATION (see Figure 4.13).



Characteristic	BASE_CCT
Long description	Base cost center
Short description	Base cost center
Version	New — Not saved
Object Status	Inactive, not executable

SIDTab timeDepAttr.	
✓ Language-dependent texts	
✓ Texts are time-dep.	
☐ Master Data Maint. with Auth. Check	
DS Object for Check	

Master Data InfoSource / Data Target / InfoProvider / Master Data Read Access

✓ InfoSource with Direct Update	
Application Component	NODESNOTCONNEC...
✓ Character. is InfoProvider	
InfoArea	NODESNOTCONNEC... Unassigned Nodes
☐ Characterist. is export data source	
Master Data Access	Own Implementation
Name of Master Data Read Class	ZCL_READ_BASE_CCT
Master Data Read Class Parameters	
HANA-Katalog	
HANA Information Model	
☐ Permitted for Real-Time Data Acquisition	

Figure 4.13 Entering the Master Data Read Class in the InfoObject

3. Select the name of the master data read class, `ZCL_READ_BASE_CCT`.

[!]

InfoObjects with Master Data Class

InfoObjects with master data read classes can't have *any* navigation attributes. Moreover, the master data maintenance doesn't read the results of the master data read class.

4. After you've activated the InfoObject, you can test the implementation. The simplest way to do this is to call the master data maintenance and use F4 in the InfoObject.

   At this point, you can set a clean breakpoint to debug any possible errors.

In general, creating your own master data read class is a nice alternative to implementing effects that would not be possible otherwise (for example, automatic filtering or user-dependent texts). However, you should always check whether it would make more sense to load the InfoObject conventionally.

It's recommended that you implement the master data read classes only in exceptional cases. Particularly, the restriction that InfoObjects mustn't contain navigation attributes doesn't allow for their widespread use.

Index