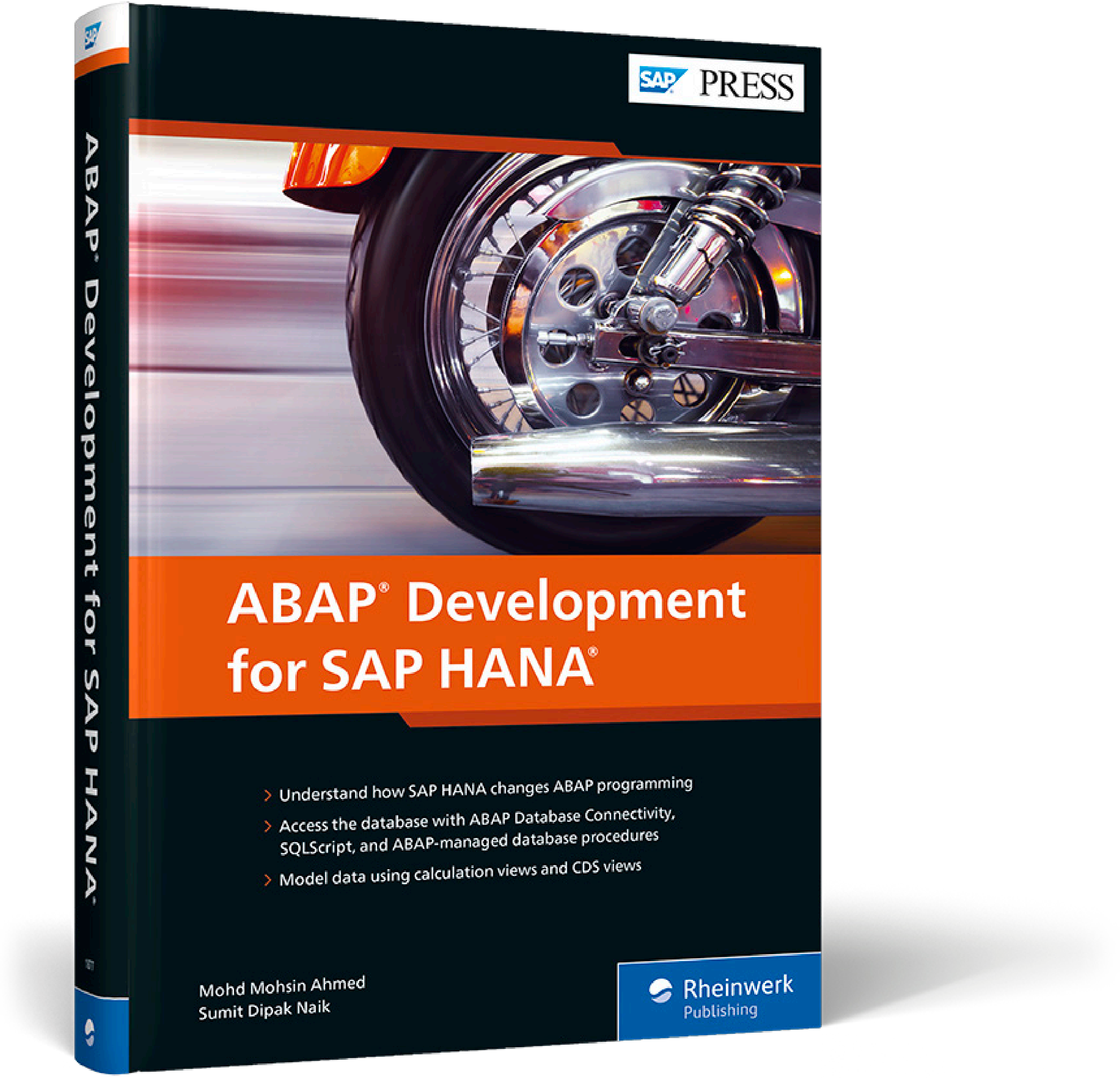

SAP PRESS
ABAP® Development for SAP HANA®
› Understand how SAP HANA changes ABAP programming
› Access the database with ABAP Database Connectivity, SQLScript, and ABAP-managed database procedures
› Model data using calculation views and CDS views
Mohd Mohsin Ahmed
Sumit Dipak Naik
Rheinwerk Publishing
ABAP® Development for SAP HANA®

Browse the Book

ABAP-managed database procedures can be leveraged to optimize nonnative, ABAP-based applications such as programs, forms, and interfaces by leveraging the code pushdown features supported by SAP HANA database. In this chapter you will learn to define and implement AMDP methods.

"ABAP-Managed Database Procedures"

Contents

Index

The Authors

Mohd Mohsin Ahmed, Sumit Dipak Naik

ABAP Development for SAP HANA

643 Pages, 2021, $89.95
ISBN 978-1-4932-1877-6

www.sap-press.com/4954

Chapter 6
ABAP-Managed Database Procedures

ABAP-managed database procedures can be leveraged to optimize non-native, ABAP-based applications such as programs, forms, and interfaces by leveraging the code pushdown features supported by SAP HANA database. In this chapter you will learn to define and implement AMDP methods.

In Chapter 2, we briefly introduced you to the code pushdown paradigm, which helps developers optimize business applications on the SAP HANA database. This chapter will include an emphasis on ABAP-managed database procedures (AMDPs), one of the recommended approaches to achieve code pushdown functionalities.

In this chapter, we'll start by covering essential aspects like the motivation behind introducing AMDPs, including how they are created and consumed in business applications in Section 6.1. We'll then show you how to create AMDP classes in Section 6.2 and also explore the concepts behind enhancements using AMDPs in Section 6.3. Next, we'll turn to exception handing in Section 6.4 and explore different debugging techniques to analyze your procedures in Section 6.5. We'll conclude this chapter on AMDPs tools in Section 6.6.

6.1 Introduction

Let's briefly recap some information you learned earlier in this book about code pushdown. In the classic style of coding, as shown in Figure 6.1, developers used to design applications by retrieving all the data at once using database array operations such as `FOR ALL ENTRIES`, `JOIN`, or ABAP Dictionary views. This approach reduced loads on database servers by limiting data transfer requests between the application and the database server.

Subsequent data processing operations are performed on the application server's internal table to achieve the desired results. However, this approach's drawback was the formation of complex SQL queries to retrieve data. Performance issues arose because unnecessary data was retrieved, filtered, and processed in the application layer.

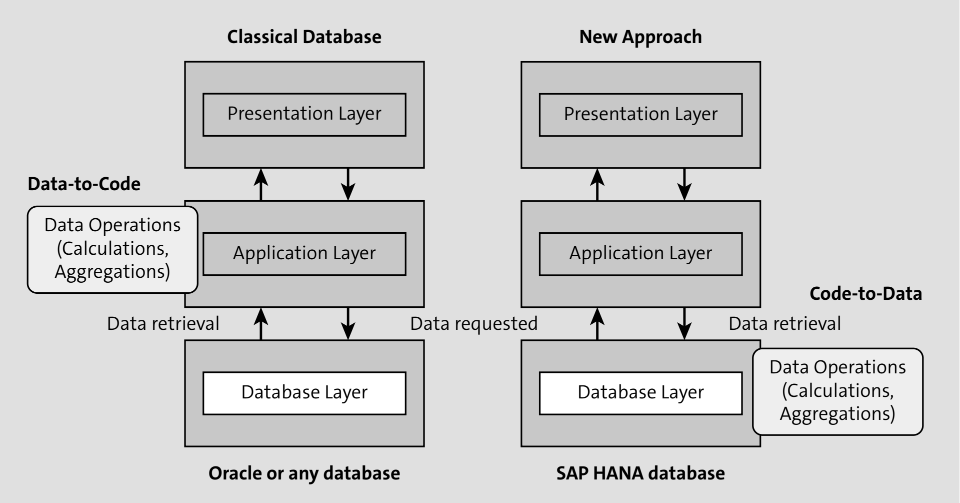


Figure 6.1 New Programming Approach

This fundamental change to the ABAP programming model in favor of pushing code and data processing to the database is called the *code-to-data paradigm*, shown in Figure 6.1, instead of the classic *data-to-code approach*.

In SAP HANA, several code-to-data techniques are available for implementing data-intensive calculations in the database layer. SAP HANA performs data-intensive calculations in the database layer using SAP HANA views or procedures. These artifacts are later utilized in ABAP applications using several code-to-data techniques. Whether or not you should use these techniques depends on the SAP NetWeaver Application Server for ABAP (AS ABAP) release used in your landscape.

As shown in Figure 6.2, if you are using SAP NetWeaver AS ABAP 7.4 SP02 or lower, SAP HANA repository objects, such as SAP HANA views or procedures, are directly accessed in ABAP applications using *native SQL*. With SAP NetWeaver AS ABAP 7.4 SP02 or higher, these objects are accessed in ABAP applications using ABAP proxy objects such as *external views* or *database procedure proxies* to overcome the limitations of using native SQL.

The techniques used before SAP NetWeaver AS ABAP 7.4 SP05 are known as bottom-up approaches because SAP HANA views or procedures are created in the database layer. The views are created using a database user in the **Modeler** perspective and later consumed in the ABAP layer using Native SQL or proxy objects. Although these techniques offer the benefit of performing the data processing in the database layer, one limitation of using a bottom-up approach is its complex lifecycle management requirements. For example, handling SAP HANA repository objects must occur separately through delivery units and ABAP proxy objects by using the SAP HANA transport containers to import objects to other systems. Additionally, developers must ensure proper synchronization of all SAP HANA artifacts in all the environments whenever any procedure or view changed. These shortcomings were later handled with the release of SAP NetWeaver AS ABAP 7.4 SP05 by introducing progressive code pushdown techniques like ABAP-managed database procedures (AMDPs) and core data services (CDS). These techniques are referred to as top-down approaches because the entire lifecycle management is conducted by the ABAP layer. SAP HANA artifacts such as views and procedures are automatically created in the database.

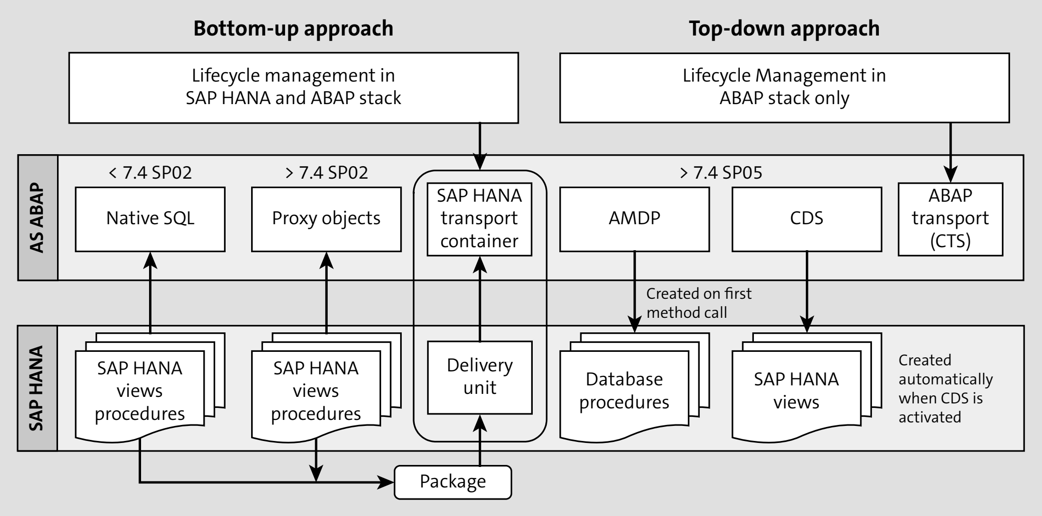

Bottom-up approach
Top-down approach
Lifecycle management in SAP HANA and ABAP stack
Lifecycle Management in ABAP stack only
AS ABAP
< 7.4 SP02
> 7.4 SP02
> 7.4 SP05
Native SQL
Proxy objects
SAP HANA transport container
AMDP
CDS
ABAP transport (CTS)
Created on first method call
SAP HANA
SAP HANA views procedures
SAP HANA views procedures
Delivery unit
Database procedures
SAP HANA views
Created automatically when CDS is activated
Package

Figure 6.2 Code-to-Data Approach

For example, when an AMDP class and method are defined, a procedure is created in the database before an AMDP method is called for the first time in the calling program and updated on the subsequent calls if it has been changed. In contrast, in the case of a CDS view, the SAP HANA view is created when the CDS view is activated.

6.1.1 ABAP-Managed Database Procedure Framework

AMDPs are a recommended technique for achieving code pushdown functionality if your underlying database is SAP HANA. The framework uses a top-down approach to create and manage database procedures in the ABAP environment. The AMDP framework guarantees that the complete lifecycle of the AMDP procedure—from creating, changing, activating, and transporting the procedure—occurs in the application layer by the ABAP runtime environment.

As a developer, you'll write procedures in an AMDP method implementation of an AMDP class. In contrast to the traditional ABAP method, an AMDP method is a unique method and implements database-specific programming languages, such as SQLScript, native SQL, and L (used internally by SAP). The keyword `LANGUAGE` specifies the database-specific language for implementing the procedure.

As shown in Listing 6.1, the usage of the keyword `BY DATABASE PROCEDURE` in the implementation section of the AMDP method helps differentiates whether the method uses ABAP or any other language to implement the procedure.

```
AMDP Class Definition
CLASS zcl_amdp_example DEFINITION
  PUBLIC
  FINAL
  CREATE PUBLIC.

  PUBLIC SECTION.

AMDP Marker Interface
    INTERFACES if_amdp_marker_hdb.

AMDP Method
    CLASS-METHODS amdp_method
Importing parameter defined using Dictionary type
      IMPORTING VALUE(im_input)  TYPE matnr
Importing parameter defined using ABAP type
      EXPORTING VALUE(ex_output) TYPE i
Importing parameter defined using TABLE type
      CHANGING  VALUE(ch_param)  TYPE ttyp_d.

  PROTECTED SECTION.
  PRIVATE SECTION.
ENDCLASS.

AMDP Class Implementation
CLASS zcl_amdp_example IMPLEMENTATION.

AMDP Method Implementation
  METHOD amdp_method BY DATABASE PROCEDURE
                   FOR HDB
                   LANGUAGE SQLSCRIPT.
--Implement SQLScript Code ( Database specific code )
  ENDMETHOD.
ENDCLASS.
```

Listing 6.1 AMDP Framework Definition

In general, you should consider the following points before creating or consuming an AMDP:

- Currently, the AMDP framework only supports database procedures for SAP HANA databases. However, SAP has designed the framework to support stored procedures for other databases.
- You can only create or edit an AMDP using the Eclipse-based ABAP Development Tools (ADT). Thus, the classic SAP GUI-based class builder (Transaction SE24) is not suitable for managing an AMDP class and its methods, as only the display function is supported in Transaction SE24.
- Developers classified as standard ABAP users with appropriate authorizations can manage database procedures using an AMDP class, and Transaction SICK can detect missing permissions.

Benefits of Using AMDPs

Let's discuss a few benefits of using AMDPs:

- A standard ABAP user can manage an AMDP, unlike in bottom-up techniques like SAP HANA views and procedures, where both ABAP and SAP HANA database users are required.
- The AMDP framework is responsible for communicating with the database and automatically creating the database procedures as SAP HANA repository catalog objects.
- The entire lifecycle management to synchronize, create, change, activate, and transport procedures is performed in the ABAP layer.
- The **ABAP** perspective within ADT serves as a development environment for writing and managing your SQL scripts.
- The framework supports full integration of SQLScript syntax check and debugging into the ABAP environment.
- Even though an AMDP might be implemented using a database-specific language, such as native SQL or SQLScript, the ABAP environment still evaluates source code for any syntax errors.
- Procedures are automatically created in the SAP HANA database by the ABAP runtime environment before the first AMDP method call.
- You can extend an AMDP using Business Add-Ins (BAdIs) if it has an extension provisioned by the software provider.
- An AMDP is not a replacement for database procedure proxies, which are still considered in sidecar scenarios with secondary database connections to access SQLScript procedures in a different SAP HANA database.

6.1.2 Development Environment for AMDP

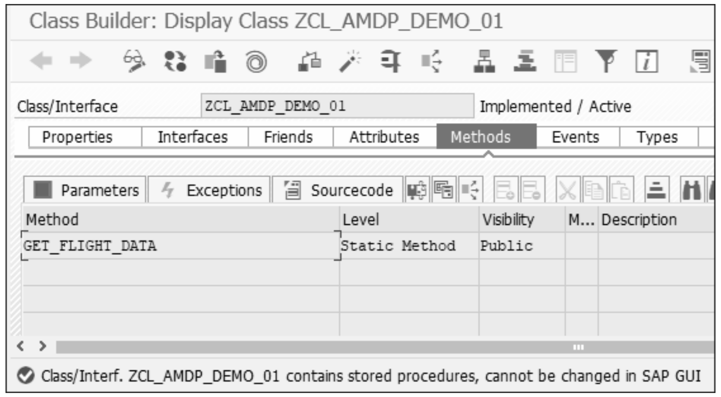
Using ABAP Development Tools (ADT) is mandatory for creating and changing AMDPs, as the classic SAP GUI-based Transaction SE24 only supports the display function. ADT also delivers additional features for developers to work effectively with AMDPs and improve developer productivity and efficiency.

Some features delivered by ADT for managing ADMPs include the following:

- Code completion functionality for ABAP, accessed by pressing `Ctrl` + `Space`
- SQLScript syntax check available in the ABAP environment
- Highlighting of syntax errors in SQLScript
- Analysis of AMDP methods via the debugging functionality
- Highlighted usage of embedded language to distinguish between database-specific language and ABAP code

Let's take a look at some of the features supported by ADT next.

As shown in Figure 6.3, the SAP GUI-based class builder (Transaction SE24) does not allow you to edit an AMDP class and only supports display function; thus, ADT is the preferred development environment for AMDPs.


Class Builder: Display Class ZCL_AMDP_DEMO_01

Class/Interface ZCL_AMDP_DEMO_01 Implemented / Active

Method	Level	Visibility	M...	Description
GET_FLIGHT_DATA	Static Method	Public		

Class/Interf. ZCL_AMDP_DEMO_01 contains stored procedures, cannot be changed in SAP GUI

Figure 6.3 SAP GUI-Based Class Builder (Transaction SE24)

In ADT, the appearance of the form-based editor can be changed by the developer to highlight syntax errors and to differentiate between the embedded (database-specific) language and the ABAP language. To enable syntax highlighting, navigate to **Windows • Preferences,** as shown in Figure 6.4.

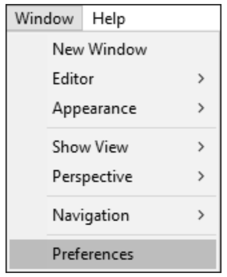
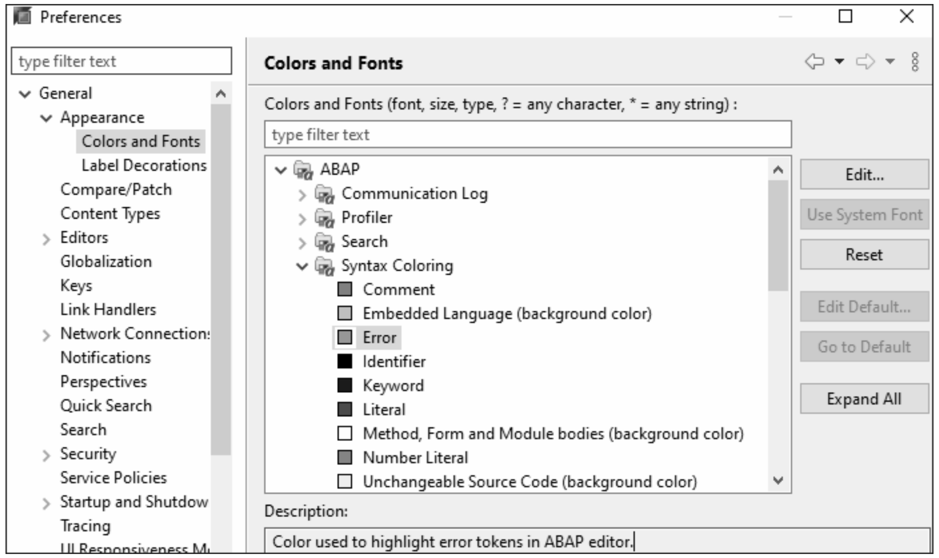


Figure 6.4 Preferences for the Form-Based Editor

Select **General • Appearance • Colors and Fonts • ABAP**, as shown in Figure 6.5. Then, select **Embedded language** under **Syntax Coloring** and click **Edit...** to modify the color according to your preferences.

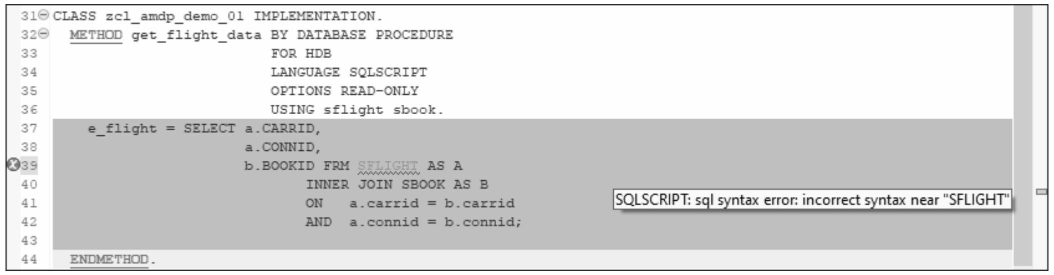


Description:

Color used to highlight error tokens in ABAP editor.

Figure 6.5 Modifying Colors and Fonts of the Form-Based Editor

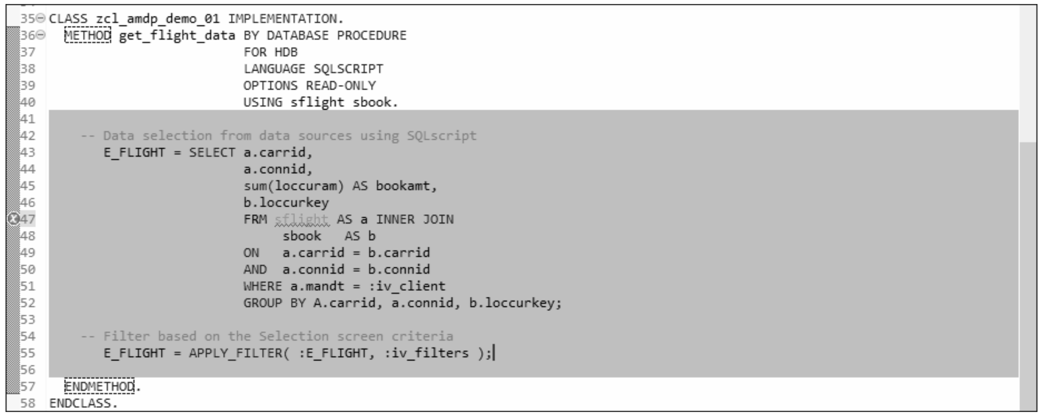
In ADT, SQLScript syntax error is fully supported and integrated into ABAP. This can be seen by toggling the cursor on the error marker on the right-hand side of the form-based editor. The detailed SQLScript syntax errors can be seen by toggling the cursor on the error, as shown in Figure 6.6.


```
CLASS zcl_amdp_demo_01 IMPLEMENTATION.
  METHOD get_flight_data BY DATABASE PROCEDURE
                         FOR HDB
                         LANGUAGE SQLSCRIPT
                         OPTIONS READ-ONLY
                         USING sflight sbook.
    e_flight = SELECT a.CARRID,
                      a.CONNID,
                      b.BOOKID FRM SFLIGHT AS A
                           INNER JOIN SBOOK AS B
                           ON   a.carrid = b.carrid
                           AND  a.connid = b.connid;

  ENDMETHOD.
```

SQLSCRIPT: sql syntax error: incorrect syntax near "SFLIGHT"

Figure 6.6 SQLScript Syntax Errors

In ADT, you can also highlight the SQLScript syntax errors at the point where they occur. As shown in Figure 6.7, the syntax error statement `SFLIGHT` is emphasized in amber color. Additionally the detailed syntax error description can also be seen on the right-hand side of the source code editor.



```
CLASS zcl_amdp_demo_01 IMPLEMENTATION.
  METHOD get_flight_data BY DATABASE PROCEDURE
                         FOR HDB
                         LANGUAGE SQLSCRIPT
                         OPTIONS READ-ONLY
                         USING sflight sbook.

    -- Data selection from data sources using SQLscript
       E_FLIGHT = SELECT a.carrid,
                         a.connid,
                         sum(loccuram) AS bookamt,
                         b.loccurkey
                         FRM sflight AS a INNER JOIN
                              sbook   AS b
                         ON   a.carrid = b.carrid
                         AND  a.connid = b.connid
                         WHERE a.mandt = :iv_client
                         GROUP BY A.carrid, a.connid, b.loccurkey;

    -- Filter based on the Selection screen criteria
       E_FLIGHT = APPLY_FILTER( :E_FLIGHT, :iv_filters );

  ENDMETHOD.
ENDCLASS.
```

Figure 6.7 Emphasizing SQLScript Errors and Differentiating between ABAP and Database Specific Code

In the AMDP method, you can also set the background color of the embedded language to differentiate between ABAP and database specific code such as SQLScript. Set the color using the same process we discussed earlier in Figure 6.4: navigate to **Windows • Preferences • General • Appearance • Colors and Fonts • ABAP • Embedded language**. Figure 6.7 you can see that the background color for database-specific syntax is emphasized in gray color.

6.2 Creating AMDP Classes

A global class must be defined in the class library using ADT to create an AMDP procedure. A class is categorized as an AMDP class if its definition contains one or more tag interfaces. The tag interfaces are prefixed with `IF_AMDP_MARKER` and end in a suffix indicating database system for which the procedure is implemented.

In the following sections, we will understand several prerequisites that you should consider while defining and implementing an AMDP method. You will also learn to consume AMDP and check if current database (or a database specified using a database connection) supports the AMDP features in the ABAP applications.

> **Example**
>
> The marker interface `IF_AMDP_MARKER_HDB` is relevant for SAP HANA database, where `HDB` indicates that the procedure is intended for an SAP HANA database.

6.2.1 Prerequisites

An AMDP class can be comprised of one or more traditional and AMDP methods. It can also contain AMDPs for each database system specified by a tag interface. The source code shown in Listing 6.2 illustrates a simple AMDP definition implementing all the required prerequisites. These prerequisites are as follows:

- An AMDP class definition should contain a marker interface `IF_AMDP_MARKER_HDB`, as it implements an AMDP method for the SAP HANA database.
- In the class definition, the AMDP method parameter types should be a dictionary, ABAP (for example, integer or character), or table types. For parameters with table types, the line types should contain elementary components because nested tables are not supported.
- An AMDP method can only contain importing, exporting, and changing parameters. An AMDP method cannot have return parameters.
- Similar to remote function call (RFC) parameters, all method parameters should be defined as *pass by value*. *Pass by reference* in the method definition is not permitted.
- An AMDP method can be defined in the `PUBLIC SECTION`, `PRIVATE SECTION`, or `PROTECTED SECTION` of the class. However, if the AMDP methods of other classes do not call the method, you must declare the method as `PRIVATE`.

```
CLASS zcl_amdp_demo_01 DEFINITION
  PUBLIC
  FINAL
  CREATE PUBLIC.
  PUBLIC SECTION.

    AMDP Marker Interface
    INTERFACES if_amdp_marker_hdb.

    Type Declaration
    TYPES: BEGIN OF d_flight,
             carrid    TYPE s_carr_id,
             connid    TYPE s_conn_id,
             bookamt   TYPE s_f_cur_pr,
             loccurkey TYPE s_currcode,
           END OF d_flight.
    TYPES: tt_flight TYPE STANDARD TABLE OF d_flight.

    AMDP Method Definition
    CLASS-METHODS get_flight_data
      IMPORTING
                VALUE(iv_filters) TYPE string
                VALUE(iv_client)  TYPE sy-mandt
      EXPORTING
                VALUE(e_flight)   TYPE tt_flight
      CHANGING
                VALUE(c_return)   TYPE i

      RAISING   cx_amdp_no_connection
                cx_amdp_execution_error.
  PROTECTED SECTION.
  PRIVATE SECTION.
ENDCLASS.
```

Listing 6.2 AMDP Class Definition

6.2.2 Implementing AMDP Methods

An AMDP method is a unique method that optimizes ABAP applications by implementing code pushdown from the application server layer to the database layer. This method is wrapped in a global class and can be defined as either a static method or instance method. Even though you can define an AMDP method as an instance method, it will always be executed as a static method call.

Two types of AMDP methods exist:

- An AMDP procedure without a return code is defined by a method using the addition `BY DATABASE PROCEDURE`.
- An AMDP function with a return code is defined by a method using the addition `BY DATABASE FUNCTION`.

Any regular method within an AMDP class can be transformed into an AMDP method by using either the `BY DATABASE PROCEDURE` or `BY DATABASE FUNCTION` addition at the start of the method statement in the method implementation part, followed by the database system for which the procedure is implemented, the language in which the business logic is written, and the mandatory ABAP objects (which may include transparent tables, views, and other AMDPs that are used as data sources).

Additionally, you can mark an AMDP method as `READ ONLY` using the addition `OPTIONS`, which is optional.

The body of an AMDP method, shown in Listing 6.3 uses database-specific language such as SQLScript or native SQL. The source code shown in Listing 6.3 illustrates an AMDP class and method with database-specific logic using SQLScript to summarize sales by airline and flight code. The procedure also filters records based on the selection screen using the `FILTER` keyword. To implement business logic, you can use the full SQLScript reference, except for calculation engine (CE) functions such as the following:

- `CE_LEFT_OUTER_JOIN`
- `CE_COLUMN_TABLE`
- `CE_UNION_ALL`

```
CLASS zcl_amdp_demo_01 IMPLEMENTATION.
  METHOD get_flight_data BY DATABASE PROCEDURE
                         FOR HDB
                         LANGUAGE SQLSCRIPT
                         OPTIONS READ-ONLY
                         USING sflight sbook.

   Data selection from data sources using SQLScript
       E_FLIGHT = SELECT a.carrid,
                         a.connid,
                         sum(loccuram) AS bookamt,
                         b.loccurkey
                         FROM sflight AS a INNER JOIN
                              sbook   AS b
                         ON   a.carrid = b.carrid
                         AND  a.connid = b.connid
                         WHERE a.mandt = :iv_client
                   GROUP BY A.carrid, a.connid, b.loccurkey;

   Filter based on the selection screen criteria
       E_FLIGHT = APPLY_FILTER( :E_FLIGHT, :iv_filters );

  ENDMETHOD.
ENDCLASS.
```

Listing 6.3 AMDP Method Implementation Example

Listing 6.3 contains the following elements:

- A global AMDP class implementation contains the AMDP method implementation. In our example, our AMDP method will summarize total flight sales by airline and flight code.
- The method `GET_FLIGHT_DATA` is implemented as an AMDP method since the method is defined with the addition `BY DATABASE PROCEDURE`.
- What follows is the database addition `HDB` to implement the procedure for the SAP HANA database. The AMDP framework only supports SAP HANA database; however, the framework is designed to work with other databases.
- Further, the database-specific language to be used is specified. In this case, SQLScript will be used in the AMDP method to implement the business logic.
- All database objects, such as dictionary tables, views, and other AMDP methods used as data sources within the method body, must be declared explicitly with the keyword `USING`. These objects can be accessed directly, that is, without the need to prefix these objects with *SAP<SID> (Schema)*. However, for nested AMDP calls, that is, for

AMDP methods called inside an AMDP body, you should specify objects by their full names, that is, with the class they belong to and the method name in uppercase and closed in double quotation marks.

- You must ensure that any objects that are not part of *SAP<SID>* schema are available at runtime since these objects are not managed and therefore are not included in the `USING` clause.
- Finally, the SQLScript language is used within the AMDP body to write the business logic that is executed in the database layer.
- The procedure results are filtered using the SQLScript function `APPLY_FILTER` based on the selection criteria provided as an importing parameter to the AMDP method.
- The procedure written in the AMDP body is highlighted in gray to differentiate between ABAP-specific language and database-specific language. Refer to Figure 6.4 to see how to set the color for the editor's background.

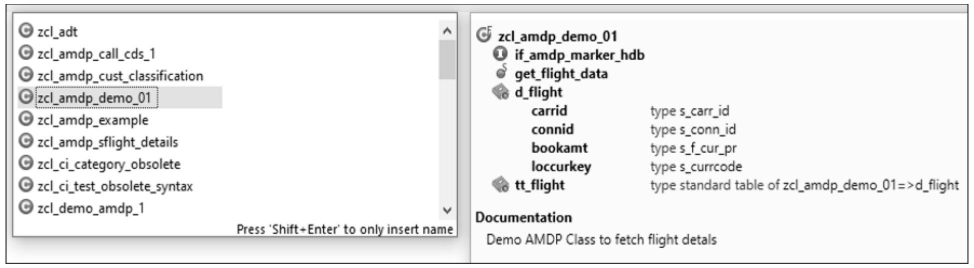
You should consider the following restrictions when implementing an AMDP method:

- Data definition language (DDL) such as `Create`, `Alter`, or `Delete` are not allowed to create, change, or delete any database objects.
- You cannot access local temporary data objects, such as internal tables, or variables defined in the class definition in the method implementation.
- Statements like database commits and rollbacks are not allowed in the method body. Also, to avoid data inconsistencies between procedures, you should handle logical units of work separately in the ABAP program.
- You cannot extend AMDP methods using implicit enhancement options, as these methods are directly executed on the database, and implicit options are not available within an AMDP method.
- While using data manipulation language (DML), such as `INSERT`, `UPDATE`, `MODIFY`, `DELETE`, etc., write access to buffered tables is not allowed.

6.2.3 Calling AMDP Methods in Applications

An AMDP method is called in an ABAP application similar to any other regular method, using an Eclipse-based form editor in ADT or through SAP GUI-based transactions. These methods are always executed as static method calls, even if defined as instance methods.

You can call an AMDP method in an ABAP application in several ways. In the Eclipse-based **ABAP** perspective (see Figure 6.8), you can use the code completion template by pressing Ctrl + Space to call an AMDP method. In SAP GUI-based editors (see Figure 6.9), you can use ABAP-based patterns by pressing Ctrl + F6.



```
zcl_adt
zcl_amdp_call_cds_1
zcl_amdp_cust_classification
zcl_amdp_demo_01
zcl_amdp_example
zcl_amdp_sflight_details
zcl_ci_category_obsolete
zcl_ci_test_obsolete_syntax
zcl_demo_amdp_1
```

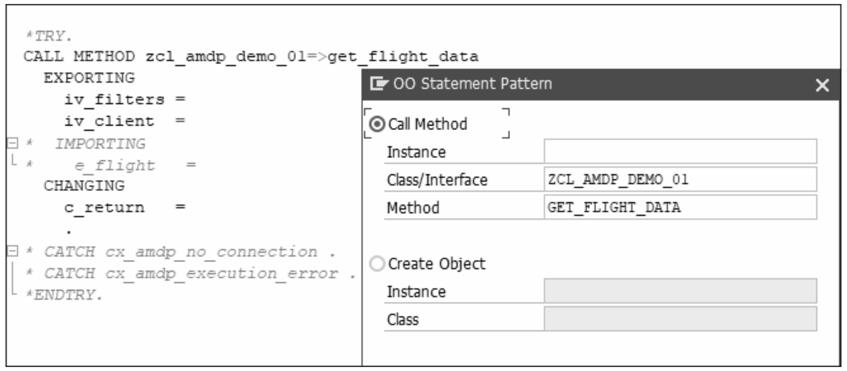
Press 'Shift+Enter' to only insert name

```
zcl_amdp_demo_01
  if_amdp_marker_hdb
  get_flight_data
  d_flight
    carrid       type s_carr_id
    connid       type s_conn_id
    bookamt      type s_f_cur_pr
    loccurkey    type s_currcode
  tt_flight      type standard table of zcl_amdp_demo_01=>d_flight
```

Documentation

Demo AMDP Class to fetch flight detals

Figure 6.8 Consuming AMDP Method in ABAP (Eclipse)



```
*TRY.
CALL METHOD zcl_amdp_demo_01=>get_flight_data
  EXPORTING
    iv_filters =
    iv_client  =
*  IMPORTING
*    e_flight   =
  CHANGING
    c_return   =
    .
* CATCH cx_amdp_no_connection .
* CATCH cx_amdp_execution_error .
*ENDTRY.
```

OO Statement Pattern

● Call Method

Instance	
Class/Interface	ZCL_AMDP_DEMO_01
Method	GET_FLIGHT_DATA

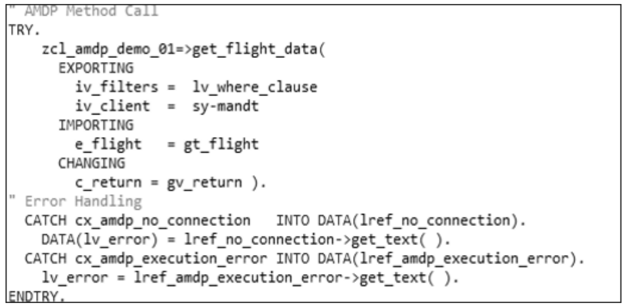
○ Create Object

Instance	
Class	

Figure 6.9 Consuming AMDP Method in ABAP (SAP GUI)

However, to consume an AMDP, the SAP NetWeaver AS ABAP's central database should be managed by the database system for which the AMDP method is implemented. If not the case, then the procedure call results in a runtime error.

As shown in Figure 6.10, before the first method call, the ABAP runtime environment creates the procedure implemented in the AMDP method in the database system or updates any existing database procedure if the AMDP has changed, as shown in Figure 6.11. Once the method is called, the execution is performed in the database system. Parameters of the interface are passed from the native SQL interface to the database system or are applied by the database system.



```
" AMDP Method Call
TRY.
    zcl_amdp_demo_01=>get_flight_data(
      EXPORTING
        iv_filters =  lv_where_clause
        iv_client  =  sy-mandt
      IMPORTING
        e_flight   = gt_flight
      CHANGING
        c_return = gv_return ).
" Error Handling
  CATCH cx_amdp_no_connection   INTO DATA(lref_no_connection).
    DATA(lv_error) = lref_no_connection->get_text( ).
  CATCH cx_amdp_execution_error INTO DATA(lref_amdp_execution_error).
    lv_error = lref_amdp_execution_error->get_text( ).
ENDTRY.
```

Figure 6.10 AMDP Method Called in Application

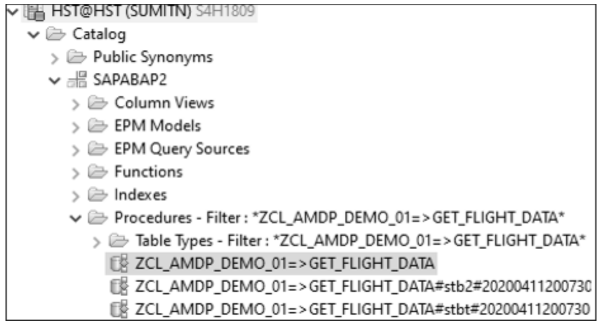

HST@HST (SUMITN) S4H1809
Catalog
Public Synonyms
SAPABAP2
Column Views
EPM Models
EPM Query Sources
Functions
Indexes
Procedures - Filter : *ZCL_AMDP_DEMO_01=>GET_FLIGHT_DATA*
Table Types - Filter : *ZCL_AMDP_DEMO_01=>GET_FLIGHT_DATA*
ZCL_AMDP_DEMO_01=>GET_FLIGHT_DATA
ZCL_AMDP_DEMO_01=>GET_FLIGHT_DATA#stb2#20200411200730
ZCL_AMDP_DEMO_01=>GET_FLIGHT_DATA#stbt#20200411200730

Figure 6.11 AMDP Procedure Created on the Database

Once a database procedure managed using AMDP has been created (ZCL_AMDP_DEMO_01=>GET_FLIGHT_DATA) on the database schema, SAPABAP2, as show in Figure 6.11, this procedure can be called from other database procedures using the database syntax, provided that the database permits this access, including AMDP procedures (or database procedures) that are not managed by AMDP. If an AMDP procedure calls another procedure, this procedure must be specified in the calling method with the addition USING.

In general, we recommend that AMDP procedure implementations that are not called from AMDP methods of other classes be created as private methods of an AMDP class and that they be called in regular ABAP methods.

> **Note**
>
> In database systems that do not support AMDP, a traditional method can be created using an alternative implementation in Open SQL or native SQL.

As shown in Listing 6.4, an ABAP application can call an AMDP procedure to display flight booking information based on a user's selection. The example also illustrates the use of SELECT-OPTIONS to filter data records.

```
REPORT zcl_amdp_demo_call_01.

" Data declaration
DATA: gwa_sflight TYPE sflight.

" Select Options
SELECT-OPTIONS: s_carrid FOR gwa_sflight-carrid,
                s_connid FOR gwa_sflight-connid.

" Types Declaration
TYPES: BEGIN OF d_flight,
         carrid    TYPE s_carr_id,
         connid    TYPE s_conn_id,
```

```
         bookamt   TYPE s_f_cur_pr,
         loccurkey TYPE s_currcode,
       END OF d_flight.

" Internal table
DATA: gt_flight TYPE STANDARD TABLE OF d_flight.

" Variable
DATA: gv_return TYPE i.

" Build dynamic where clause, and pass it to the AMDP method
TRY.
    DATA(lv_where_clause) = cl_shdb_seltab=>combine_seltabs
                  (it_named_seltabs = VALUE #(
                  ( name = 'CARRID' dref = REF #( s_carrid[] ) )
                  ( name = 'CONNID' dref = REF #( s_connid[] ) ) ) ).
  CATCH cx_shdb_exception INTO DATA(lref_shdb_exception).
    DATA(lv_meesage) = lref_shdb_exception->get_text( ).
ENDTRY.


" AMDP Method Call
TRY.
    zcl_amdp_demo_01=>get_flight_data(
      EXPORTING
        iv_filters =  lv_where_clause
        iv_client  =  sy-mandt
      IMPORTING
        e_flight   = gt_flight
      CHANGING
        c_return = gv_return ).
" Error Handling
  CATCH cx_amdp_no_connection   INTO DATA(lref_no_connection).
    DATA(lv_error) = lref_no_connection->get_text( ).
  CATCH cx_amdp_execution_error INTO DATA(lref_amdp_execution_error).
    lv_error = lref_amdp_execution_error->get_text( ).
ENDTRY.

" Display results
IF lv_error IS INITIAL.
cl_demo_output=>display_data(
 EXPORTING
   value =  gt_flight
   name  =  'Flight Booking information').
```

```
" Error Handling
ELSE.
  WRITE: lv_error.
ENDIF.
```

Listing 6.4 AMDP Method Call in ABAP Application

6.2.4 Using Multiple Selection Criteria

In ABAP reports, defining a selection screen is essential to empowering business users so they can filter data based on the desired elements. Selection criteria ensure that applications can process data faster by filtering out unwanted data in the database layer. But, to filter the data, you must define selection criteria using parameters, SELECT-OPTIONS, or a combination of both.

The purpose of parameters is to filter the records based on a single value, whereas with SELECT-OPTIONS, you can define complex selection criteria to filter out records. Developers can then use these selection screen elements directly in a WHERE clause of an Open SQL statement to filter the data. These selection criteria are then converted into the SQL WHERE conditions by the ABAP application server.

However, suppose you want to use these selection screen elements in an AMDP procedure. In this case, you can use parameters directly in the AMDP method, but this approach is not valid with SELECT-OPTIONS.

Because you cannot pass SELECT-OPTIONS directly to an AMDP method, this limitation of using an AMDP must be kept in mind. To pass SELECT-OPTIONS to an AMDP method, you must first transform the selection criteria into a filter string and then pass the string as an IMPORTING parameter to the AMDP method. To convert the SELECT-OPTIONS (selection tables or range tables) into a dynamic SQL WHERE clause, you can use the static method COMBINE_SELTABS( ) of the new class CL_SHDB_SELTAB.

This generated condition can then be used in SQLScript to filter the data source using the SQLScript function APPLY_FILTER in the AMDP method implementation. This function can be applied to database tables, views, and SAP HANA views, however this function cannot be used with analytical or table variables.

The APPLY_FILTER function expects two parameters. The first parameter is the data source to which you want to apply the filter, and the second parameter is the generated WHERE clause, which is passed as a string argument.

The CL_SHDB_SELTAB class is not available with SAP NetWeaver AS ABAP 7.4 and should be imported by following the steps described in SAP Note 2124672. SAP NetWeaver AS ABAP 7.4 SP08 or higher is required to apply this SAP Note.

Note

The class CL_LIB_SELTAB and its methods are obsolete.

An ABAP report can consume an AMDP method, as shown in Listing 6.5, filtering data based on the user selection via parameters or based on SELECT-OPTIONS to display bookings for all airline codes by a specific date and customer category.

```
REPORT zamdp_sflight_details.

* Data declaration
DATA: gwa_sflight TYPE sflight.

* Selection screen
PARAMETERS: p_date TYPE s_date.

SELECT-OPTIONS: s_carrid FOR gwa_sflight-carrid,
                s_connid FOR gwa_sflight-connid.

* Build dynamic where clause
TRY.
    DATA(lv_where_clause) = cl_shdb_seltab=>combine_seltabs(
     it_named_seltabs = VALUE #(
            ( name = 'CARRID' dref = REF #( s_carrid[] ) )
            ( name = 'CONNID' dref = REF #( s_connid[] ) ) )
              iv_client_field  = 'MANDT' ).
  CATCH cx_shdb_exception INTO DATA(lref_shdb_exception).
    DATA(lv_meesage) = lref_shdb_exception->get_text( ).
ENDTRY.

* AMDP Method call to summarize booking amount by flight, airline code, date,
and customer type
zcl_amdp_sflight_details=>get_data(
  EXPORTING
    iv_client  = sy-mandt
    iv_date    = p_date
    iv_filters = lv_where_clause
  IMPORTING
    et_results = DATA(gt_results) ).

* Display results
cl_demo_output=>display_data(
 EXPORTING
   value =  gt_results
   name  =  'Flight Booking information').
```

Listing 6.5 ABAP Report: Handling SELECT-OPTIONS

The AMDP method shown in Listing 6.6 is filtering the data based on parameters and SELECT-OPTIONS passed from the application program shown earlier in Listing 6.5 using the SQLScript function APPLY_FILTER.

```
CLASS zcl_amdp_sflight_details DEFINITION
  PUBLIC
  FINAL
  CREATE PUBLIC.

  PUBLIC SECTION.
* AMDP Marker Interface
    INTERFACES: if_amdp_marker_hdb.
* Data declaration
    TYPES: BEGIN OF d_sflight,
             carrid TYPE s_carr_id,
             connid TYPE s_conn_id,
             fldate TYPE s_date,
             type   TYPE string,
             total  TYPE s_l_cur_pr,
           END OF d_sflight,

           tty_sflight TYPE STANDARD TABLE OF d_sflight.
* AMDP Method
    CLASS-METHODS get_data
      IMPORTING
        VALUE(iv_client)  TYPE sy-mandt
        VALUE(iv_date)    TYPE s_date
        VALUE(iv_filters) TYPE string
      EXPORTING
        VALUE(et_results) TYPE  tty_sflight.
  PROTECTED SECTION.
  PRIVATE SECTION.
ENDCLASS.

CLASS zcl_amdp_sflight_details IMPLEMENTATION.
  METHOD get_data BY DATABASE PROCEDURE
                  FOR HDB
                  LANGUAGE SQLSCRIPT
                  OPTIONS READ-ONLY
                  USING sflight sbook.

    ET_RESULTS = SELECT a.carrid, a.connid, b.fldate,
                        CASE b.custtype
                        WHEN 'B' then 'Business Customer'
```

```
                        WHEN 'P' then 'Private Customer'
                        ELSE 'Others'
                        END AS "TYPE",
                        SUM(b.loccuram) AS TOTAL
                       from sflight as  a INNER JOIN
                        sbook   as  b on a.carrid = b.carrid
                      and a.connid = b.connid
               WHERE a.mandt   = :iv_client  --Parameters
               AND   b.fldate  = :iv_date    --Parameters
           GROUP BY a.carrid, a.connid, b.fldate, b.custtype ;

* Filter based on Selection screen (Select options)
    ET_RESULTS = APPLY_FILTER( :ET_RESULTS, :iv_filters );
  ENDMETHOD.
ENDCLASS.
```

Listing 6.6 AMDP: Filtering Using SELECT-OPTIONS and Parameters

6.2.5 Feature Support Check Using Global Classes

We recommend checking whether the current database or a database specified using a database connection supports AMDP features and if it can be used at runtime in the ABAP applications.

The method USE_FEATURE of the global class CL_ABAP_DBFEATURES can be used to check support for the database-specific feature. Several constants are provided to check database-specific features and can be passed to the USE_FEATURE method in an internal table. The method returns the value of ABAP_TRUE if the feature is supported by the database, whereas unsupported values raise an exception from the class CX_ABAP_INVALID_PARAM_VALUE and can be handled within the application program to avoid runtime errors. The database-specific features listed in Table 6.1 can be validated using the global class CL_ABAP_DBFEATURES.

Database Feature	Constant Name	Value
External views	EXTERNAL_VIEWS	2
Maximum number of key fields > 16 (120)	TABLE_KEYCNT_MAX1	3
Maximum width of key fields > 900 bytes (up to 2000)	TABLE_KEYLEN_MAX1	4
Maximum width of table or view > 4030 bytes (up to 16293)	TABLE_LEN_MAX1	5
AMDP table functions	AMDP_TABLE_FUNCTION	6

Table 6.1 Database-Specific Features

Database Feature	Constant Name	Value
AMDP methods are supported	`CALL_AMDP_METHOD`	8
CALL DATABASE PROCEDURE is supported	`CALL_DATABASE_PROCEDURE`	7
Internal table as the source in the FROM clause (from release 7.52)	`ITABS_IN_FROM_CLAUSE`	9
Limit/offset in subselect or common table expressions (CTEs)	`LIMIT_IN_SUBSELECT_OR_CTE`	10
CTE used in a correlated subquery	`CTE_IN_CORRELATED_SUBQUERIES`	11
MODIFY FROM SELECT	`MODIFY_FROM_SELECT`	12
Hierarchies	`HIERARCHIES`	13
GROUPING SETS	`GROUPING_SETS`	14

Table 6.1 Database-Specific Features (Cont.)

The AMDP-specific constants `CALL_AMDP_METHOD` and `AMDP_TABLE_FUNCTION` can be passed to the importing parameters of the method `USE_FEATURES` to validate if the underlying database supports the AMDP procedure, as shown in Listing 6.7.

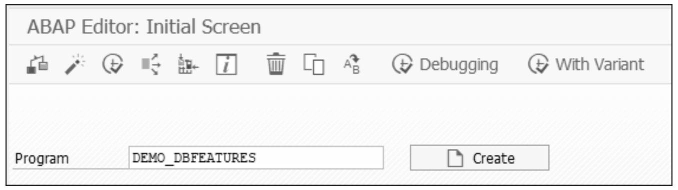
```
TRY.
    DATA(lv_supported) = cl_abap_dbfeatures=>use_
features(   EXPORTING requested_features =
VALUE #( ( cl_abap_dbfeatures=>call_amdp_method  )
         ( cl_abap_dbfeatures=>amdp_table_function ) ) ).
CATCH cx_abap_invalid_param_value INTO DATA(lref_invalid_value).
    DATA(lv_message) = lref_invalid_value->get_text( ).
ENDTRY.
IF lv_supported IS NOT INITIAL."ABAP_TRUE
  WRITE:'Database specific feature', cl_abap_dbfeatures=>call_amdp_
method, 'is supported'.
ELSE.
  WRITE lv_message.
ENDIF.
```

Listing 6.7 Database Specific Feature Check

You can also use the standard program `DEMO_DBFEATURES` to validate if the current database supports any database features before using them.

To validate database-specific features, follow these steps:

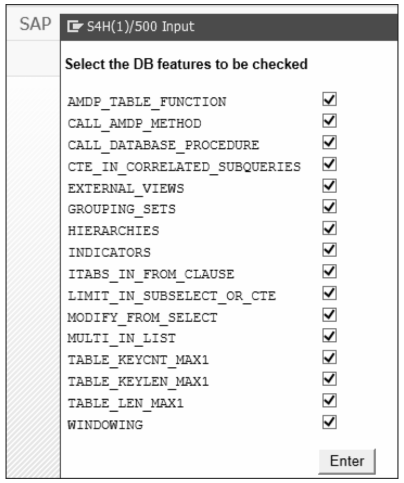
1. Execute the standard SAP program `DEMO_DBFEATURES` using Transaction SE38 (see Figure 6.12).



Program DEMO_DBFEATURES

Figure 6.12 Execute DEMO_DBFEATURES in Transaction SE38

2. You can choose the database features to validate for the SAP HANA database version (see Figure 6.13). Click on **Enter** to view the results.

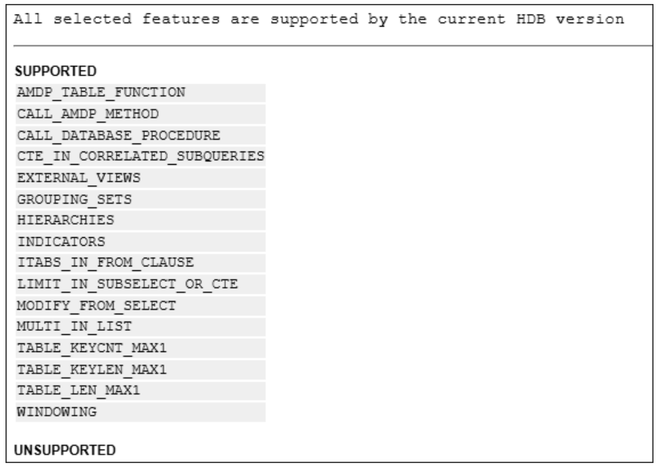


Select the DB features to be checked

```
AMDP_TABLE_FUNCTION            ☑
CALL_AMDP_METHOD               ☑
CALL_DATABASE_PROCEDURE        ☑
CTE_IN_CORRELATED_SUBQUERIES   ☑
EXTERNAL_VIEWS                 ☑
GROUPING_SETS                  ☑
HIERARCHIES                    ☑
INDICATORS                     ☑
ITABS_IN_FROM_CLAUSE           ☑
LIMIT_IN_SUBSELECT_OR_CTE      ☑
MODIFY_FROM_SELECT             ☑
MULTI_IN_LIST                  ☑
TABLE_KEYCNT_MAX1              ☑
TABLE_KEYLEN_MAX1              ☑
TABLE_LEN_MAX1                 ☑
WINDOWING                      ☑
```

Figure 6.13 Select Database Features for AMBP Supportability Check

3. The report displays the list of supported and unsupported features for the underlying database (see Figure 6.14).



```
All selected features are supported by the current HDB version

SUPPORTED
AMDP_TABLE_FUNCTION
CALL_AMDP_METHOD
CALL_DATABASE_PROCEDURE
CTE_IN_CORRELATED_SUBQUERIES
EXTERNAL_VIEWS
GROUPING_SETS
HIERARCHIES
INDICATORS
ITABS_IN_FROM_CLAUSE
LIMIT_IN_SUBSELECT_OR_CTE
MODIFY_FROM_SELECT
MULTI_IN_LIST
TABLE_KEYCNT_MAX1
TABLE_KEYLEN_MAX1
TABLE_LEN_MAX1
WINDOWING

UNSUPPORTED
```

Figure 6.14 Resulting List of Supported and Unsupported AMDP Features

6.3 Enhancements

Similar to classic ABAP extensions, where several enhancement techniques like user exits, customer exits, business transaction events (BTE), and business add-ins (BAdI), enhancement frameworks are available to perform modification-free extensions to SAP applications. These enhancements frameworks include implicit and explicit enhancements or are BAdI-managed using enhancements spots.

In the following sections, you'll learn how to define, implement, and invoke AMDP BAdI calls within other AMDP methods to extend standard business functionality.

6.3.1 AMDP BAdI Overview

You can also extend an AMDP procedure if the software or extension provides for this extensibility. As described in Table 6.2, AMDP BAdIs were introduced with SAP NetWeaver AS ABAP 7.4 SP08 to allow for modification-free extensions. As shown in Figure 6.15, you could then consume these extensions to add or modify a business requirement in the procedure.

Software or Extension Provider	Customer or Partner Extension Consumer
■ Responsible for creating an enhancement spot in Transaction SE20 ■ Responsible for creating a BAdI definition categorized as an AMDP BAdI, defines the BAdI interface, and implements the fallback class ■ Responsible for integrating the enhancement spot with the application	■ Responsible for providing an implementation class for the AMDP BAdI ■ Responsible for creating an active BAdI implementation with SQLScript code to extend or add a business requirement

Table 6.2 Modification-Free Extensions

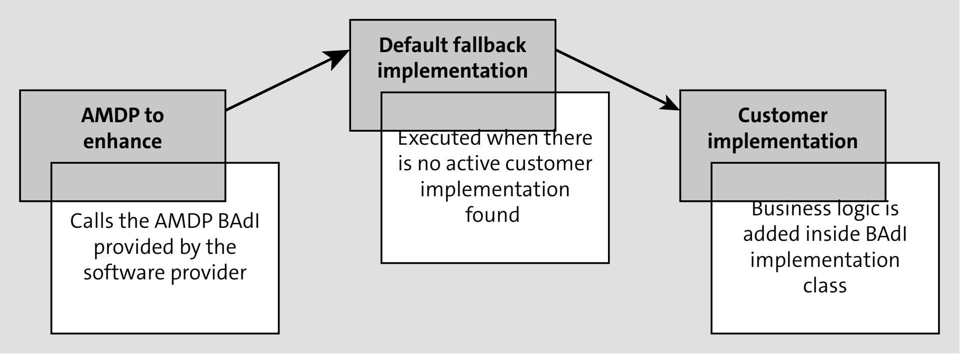
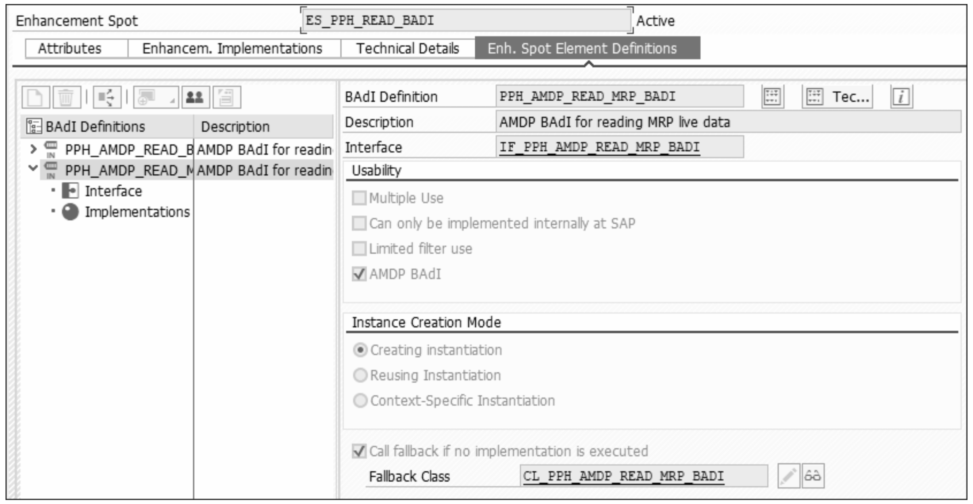

AMDP to enhance
Calls the AMDP BAdI provided by the software provider
Default fallback implementation
Executed when there is no active customer implementation found
Customer implementation
Business logic is added inside BAdI implementation class

Figure 6.15 AMDP BAdI Framework

In addition to implementing the methods of a normal BAdI as AMDP methods and making these methods callable using `CALL BADI`, you can also create a special AMDP BAdI.

An AMDP BAdI is created in Transaction SE20 and is later called within an AMDP implementation, similar to other AMDPs. An AMDP BAdI is a BAdI that is categorized accordingly in the BAdI Builder and meets the following prerequisites, which are shown in Figure 6.16:

1. SAP has provided an **Enhancement Spot** (`ES_PPH_READ_BADI`, in our example) and a **BAdI Definition** (`PPH_AMDP_READ_MRP_BADI`, in our example).
2. An AMDP BAdI does not currently have any filters, as they are not supported, indicated by the unchecked **Limited filter use** box in the **Usability** section.
3. In its definition, the BAdI is categorized as an **AMDP BAdI**, as you can see in the **Usability** section.
4. The software provider has provided the mandatory **Fallback Class**. In this example, the fallback class is `CL_PPH_AMDP_READ_MRP_BADI`. Only an AMDP class can be provided as a fallback class or implementation class.



Enhancement Spot ES_PPH_READ_BADI Active

BAdI Definitions	Description
PPH_AMDP_READ_B	AMDP BAdI for readin
PPH_AMDP_READ_M	AMDP BAdI for readin
Interface	
Implementations	

BAdI Definition PPH_AMDP_READ_MRP_BADI

Description AMDP BAdI for reading MRP live data

Interface IF_PPH_AMDP_READ_MRP_BADI

Usability

☐ Multiple Use

☐ Can only be implemented internally at SAP

☐ Limited filter use

☑ AMDP BAdI

Instance Creation Mode

● Creating instantiation

○ Reusing Instantiation

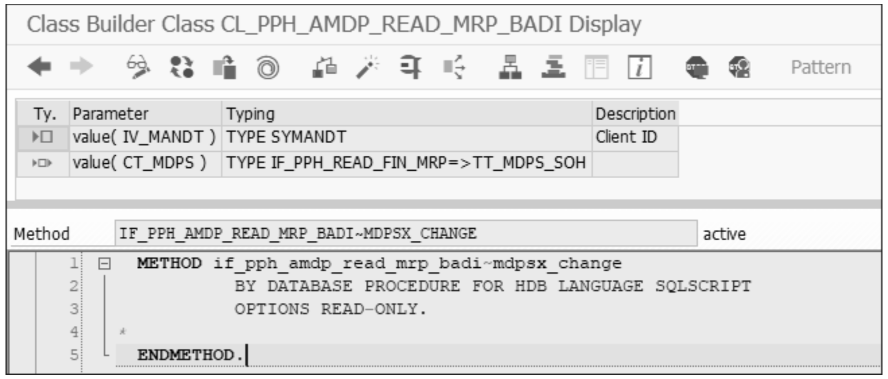
○ Context-Specific Instantiation

☑ Call fallback if no implementation is executed

Fallback Class CL_PPH_AMDP_READ_MRP_BADI

Figure 6.16 AMDP BAdI Prerequisites

5. Additionally, every BAdI method of an AMDP BAdI must be an AMDP method and must be implemented for the same database platform (only SAP HANA currently supported). This is shown in Figure 6.17 where the addition `FOR DATABASE PROCEDURE FOR HDB` is used to specify the database platform in the AMDP method `IF_PPH_AMDP_READ_MRP_BAID~MDPSX_CHANGE`. The addition `HDB` indicates this procedure is only relevant for the SAP HANA database.



Class Builder Class CL_PPH_AMDP_READ_MRP_BADI Display

Ty.	Parameter	Typing	Description
▸□	value(IV_MANDT)	TYPE SYMANDT	Client ID
▸□▸	value(CT_MDPS)	TYPE IF_PPH_READ_FIN_MRP=>TT_MDPS_SOH	

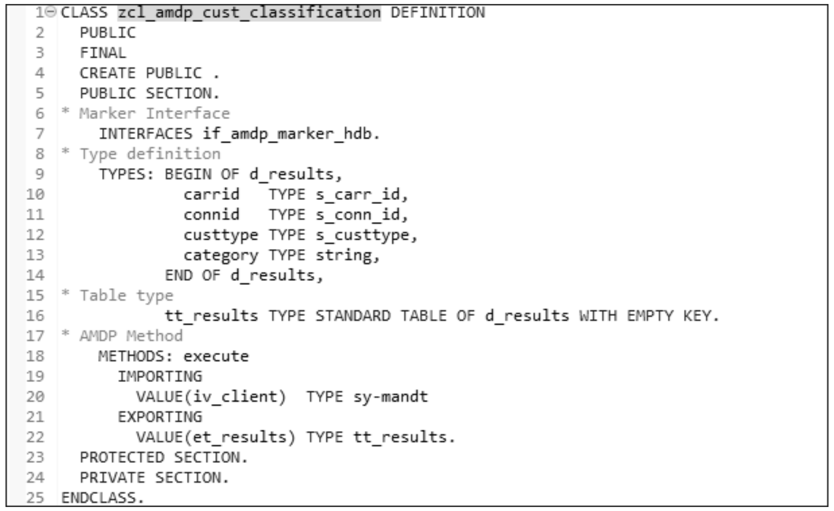
Method IF_PPH_AMDP_READ_MRP_BADI~MDPSX_CHANGE active

```
METHOD if_pph_amdp_read_mrp_badi~mdpsx_change
          BY DATABASE PROCEDURE FOR HDB LANGUAGE SQLSCRIPT
          OPTIONS READ-ONLY.
*
ENDMETHOD.
```

Figure 6.17 BAdI Method Declared as AMDP Method

Let's consider an example of an AMDP BAdI. Let's say we want an AMDP class that determines a customer's category based on the customer type, which requires an extension to achieve a customer-specific business requirement. The class's AMDP method will be consumed in an ABAP application to display the customer's category classification by following these steps:

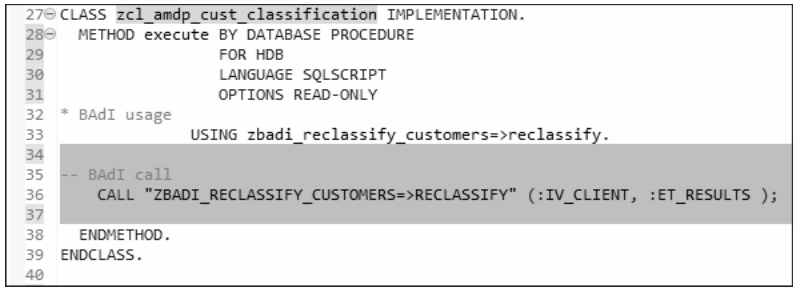
1. The AMDP class definition `ZCL_AMDP_CUST_CLASSIFICATION`, as shown in Figure 6.18, determines the customer category based on the customer type. The business logic is encapsulated in the AMDP BAdI definition in the AMDP method `ZIF_RECLASSIFY_CUSTOMERS~RECLASSIFY` of the fallback class and is called in the `EXECUTE` method implementation of the AMDP class `ZCL_AMDP_CUST_CLASSIFICATION`.



```
CLASS zcl_amdp_cust_classification DEFINITION
  PUBLIC
  FINAL
  CREATE PUBLIC .
  PUBLIC SECTION.
* Marker Interface
    INTERFACES if_amdp_marker_hdb.
* Type definition
    TYPES: BEGIN OF d_results,
             carrid   TYPE s_carr_id,
             connid   TYPE s_conn_id,
             custtype TYPE s_custtype,
             category TYPE string,
           END OF d_results,
* Table type
           tt_results TYPE STANDARD TABLE OF d_results WITH EMPTY KEY.
* AMDP Method
    METHODS: execute
      IMPORTING
        VALUE(iv_client)  TYPE sy-mandt
      EXPORTING
        VALUE(et_results) TYPE tt_results.
  PROTECTED SECTION.
  PRIVATE SECTION.
ENDCLASS.
```

Figure 6.18 AMDP Class Definition

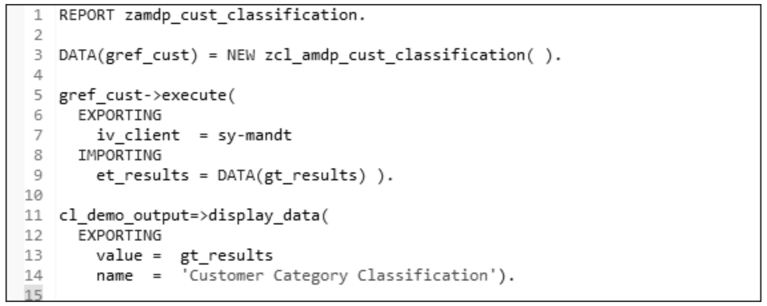
2. In the AMDP method implementation `EXECUTE`, the AMDP BAdI method `RECLASSIFY` is called to determine the customer category, as shown in Figure 6.19. The method executes the default fallback class `ZCL_RECLASSIFY_CUSTOMER_DEF`, if no active implementation exists for the AMDP BAdI.



```
CLASS zcl_amdp_cust_classification IMPLEMENTATION.
  METHOD execute BY DATABASE PROCEDURE
                 FOR HDB
                 LANGUAGE SQLSCRIPT
                 OPTIONS READ-ONLY
* BAdI usage
              USING zbadi_reclassify_customers=>reclassify.

-- BAdI call
    CALL "ZBADI_RECLASSIFY_CUSTOMERS=>RECLASSIFY" (:IV_CLIENT, :ET_RESULTS );

  ENDMETHOD.
ENDCLASS.

```

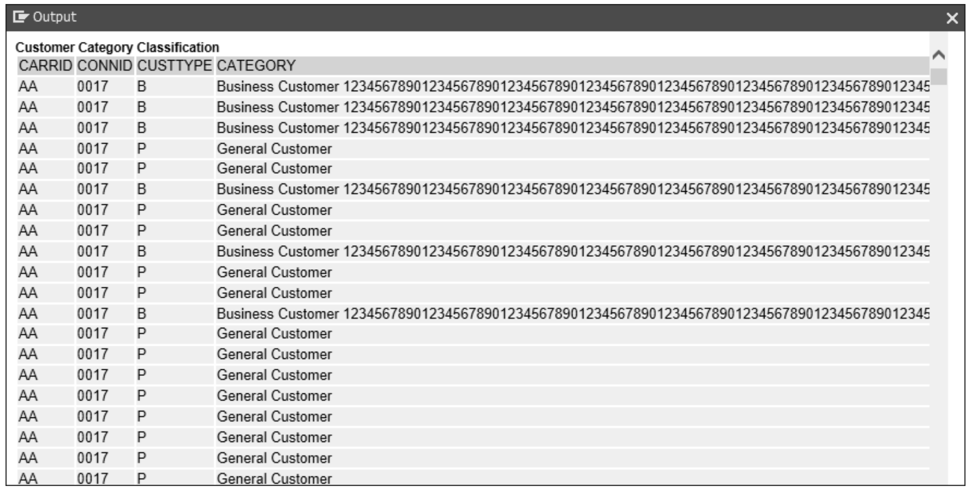
Figure 6.19 AMDP Method Implementation

3. The AMDP method `EXECUTE` is consumed in an ABAP application, as shown in Figure 6.20, to display the custom category classification (see Figure 6.21) before extending the AMDP BAdI definition.



```
REPORT zamdp_cust_classification.

DATA(gref_cust) = NEW zcl_amdp_cust_classification( ).

gref_cust->execute(
  EXPORTING
    iv_client  = sy-mandt
  IMPORTING
    et_results = DATA(gt_results) ).

cl_demo_output=>display_data(
  EXPORTING
    value =  gt_results
    name  =  'Customer Category Classification').

```

Figure 6.20 AMDP Method Consumed in an ABAP Application



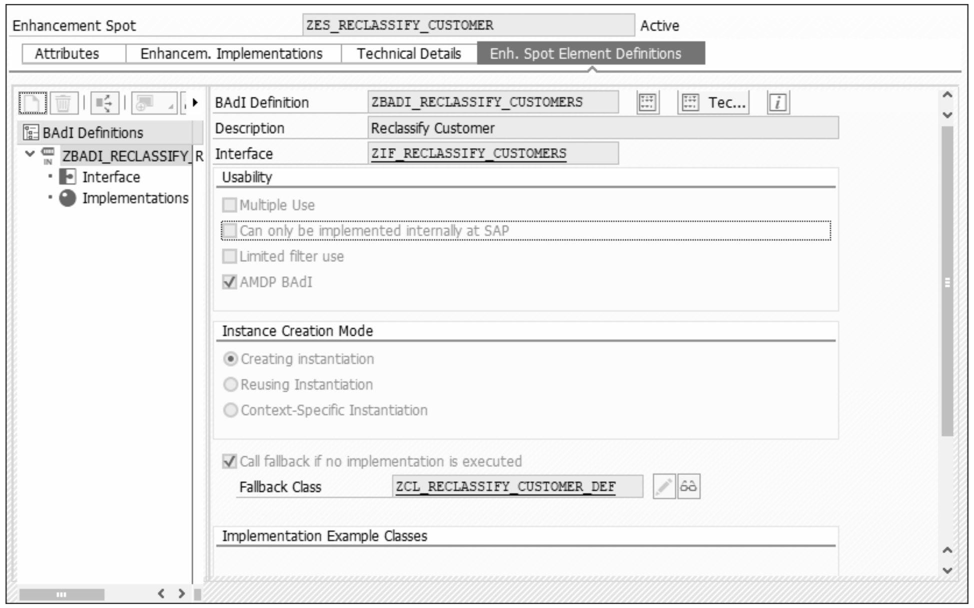
Output

```
Customer Category Classification
CARRID CONNID CUSTTYPE CATEGORY
AA     0017   B        Business Customer 123456789012345678901234567890123456789012345678901234567890123456789012345
AA     0017   B        Business Customer 123456789012345678901234567890123456789012345678901234567890123456789012345
AA     0017   B        Business Customer 123456789012345678901234567890123456789012345678901234567890123456789012345
AA     0017   P        General Customer
AA     0017   P        General Customer
AA     0017   B        Business Customer 123456789012345678901234567890123456789012345678901234567890123456789012345
AA     0017   P        General Customer
AA     0017   P        General Customer
AA     0017   B        Business Customer 123456789012345678901234567890123456789012345678901234567890123456789012345
AA     0017   P        General Customer
AA     0017   P        General Customer
AA     0017   B        Business Customer 123456789012345678901234567890123456789012345678901234567890123456789012345
AA     0017   P        General Customer
AA     0017   P        General Customer
AA     0017   P        General Customer
AA     0017   P        General Customer
AA     0017   P        General Customer
AA     0017   P        General Customer
AA     0017   P        General Customer
AA     0017   P        General Customer
```

Figure 6.21 ABAP Application Output

To extend the ADMP method shown in Figure 6.20, the software provider needs to provision an AMDP BAdI. In this case, BAdI definition `ZBADI_RECLASSIFY_CUSTOMERS` is available and encapsulated in the enhancement spot `ZES_RECLASSIFY_CUSTOMER`. You can view the AMDP BAdI definition in Transaction SE18. Let's look at the definition more deeply, especially the following aspects:

1. An AMDP BAdI definition `ZBADI_RECLASSIFY_CUSTOMERS` is provided by SAP to extend the procedure and is encapsulated in the **Enhancement Spot** `ZES_RECLASSIFY_CUSTOMER`, as shown in Figure 6.22.


Enhancement Spot: ZES_RECLASSIFY_CUSTOMER Active

BAdI Definition: ZBADI_RECLASSIFY_CUSTOMERS
Description: Reclassify Customer
Interface: ZIF_RECLASSIFY_CUSTOMERS

Usability
□ Multiple Use
□ Can only be implemented internally at SAP
□ Limited filter use
☑ AMDP BAdI

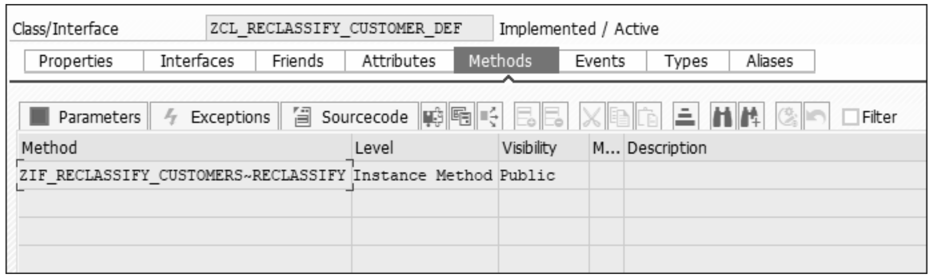
Instance Creation Mode
● Creating instantiation
○ Reusing Instantiation
○ Context-Specific Instantiation

☑ Call fallback if no implementation is executed
Fallback Class: ZCL_RECLASSIFY_CUSTOMER_DEF

Implementation Example Classes

Figure 6.22 AMDP BAdI Definition

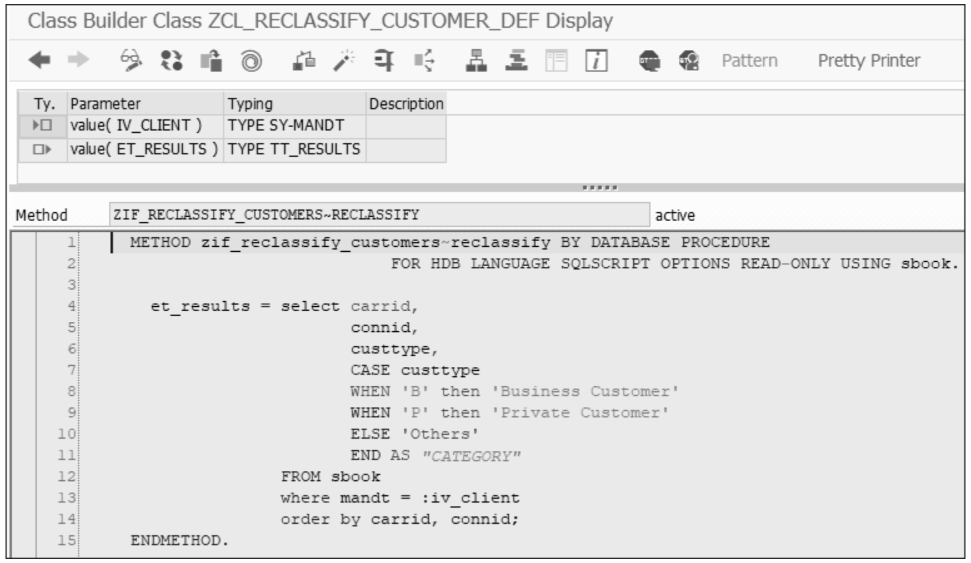
2. If no active implementation exists for the AMDP BAdI under **Implementations** section, the default implementation of the fallback class `ZCL_RECLASSIFY_CUSTOMER_DEF` will be executed, as shown in Figure 6.23.


Class/Interface: ZCL_RECLASSIFY_CUSTOMER_DEF Implemented / Active

Method	Level	Visibility	M...	Description
ZIF_RECLASSIFY_CUSTOMERS~RECLASSIFY	Instance Method	Public		

Figure 6.23 AMDP BAdI Method

3. Our AMDP BAdI method `ZIF_RECLASSIFY_CUSTOMERS~RECLASSIFY` will encapsulate the business logic written in SQLScript language, as shown in Figure 6.24, to classify and determine the customer category.


Class Builder Class ZCL_RECLASSIFY_CUSTOMER_DEF Display

Ty.	Parameter	Typing	Description
▸□	value(IV_CLIENT)	TYPE SY-MANDT	
□▸	value(ET_RESULTS)	TYPE TT_RESULTS	

Method: ZIF_RECLASSIFY_CUSTOMERS~RECLASSIFY active

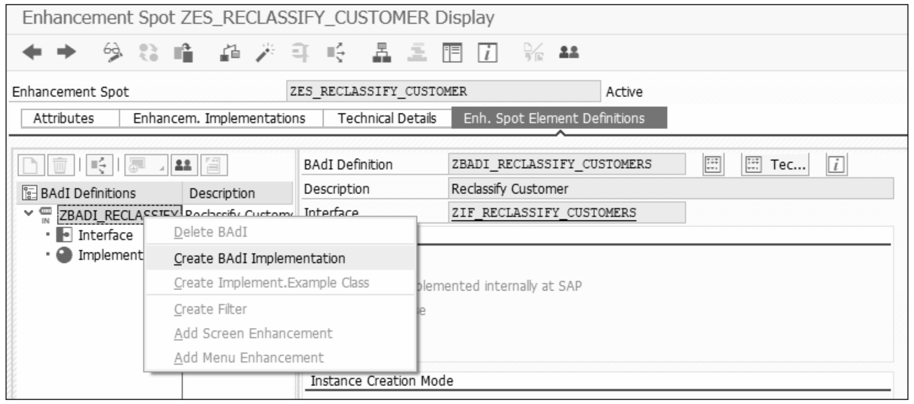
```
 METHOD zif_reclassify_customers~reclassify BY DATABASE PROCEDURE
                              FOR HDB LANGUAGE SQLSCRIPT OPTIONS READ-ONLY USING sbook.

   et_results = select carrid,
                       connid,
                       custtype,
                       CASE custtype
                       WHEN 'B' then 'Business Customer'
                       WHEN 'P' then 'Private Customer'
                       ELSE 'Others'
                       END AS "CATEGORY"
                FROM sbook
                where mandt = :iv_client
                order by carrid, connid;
 ENDMETHOD.
```

Figure 6.24 AMDP BAdI Method with SQLScript Logic

6.3.2 AMDP BAdI Implementation

To extend an AMDP method `ZIF_RECLASSIFY_CUSTOMERS~RECLASSIFY` shown in Figure 6.24, open the BAdI definition in Transaction SE18 and follow these steps to implement the BAdI definition:

1. After opening the BAdI definition in Transaction SE18, right-click on the BAdI definition name `ZBADI_RECLASSIFY_CUSTOMERS` and select on **Create BAdI Implementation**, as shown in Figure 6.25.

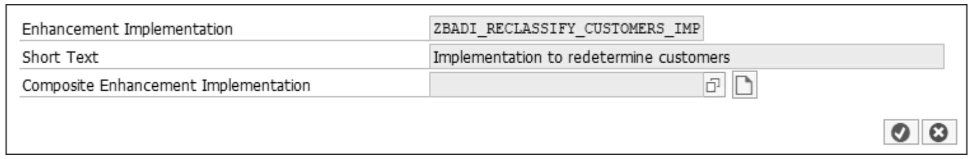

Enhancement Spot ZES_RECLASSIFY_CUSTOMER Display

Enhancement Spot: ZES_RECLASSIFY_CUSTOMER Active

BAdI Definition: ZBADI_RECLASSIFY_CUSTOMERS
Description: Reclassify Customer
Interface: ZIF_RECLASSIFY_CUSTOMERS

Figure 6.25 Create BAdI Implementation

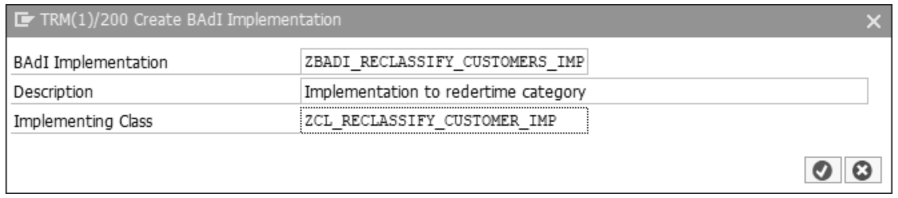
2. Create the enhancement implementation by providing a name and a short text and then clicking on **OK**, as shown in Figure 6.26.



Enhancement Implementation	ZBADI_RECLASSIFY_CUSTOMERS_IMP
Short Text	Implementation to redetermine customers
Composite Enhancement Implementation	

Figure 6.26 Create Enhancement Implementation

3. Now, create a BAdI implementation by providing the BAdI implementation a name and specifying the class to extend the AMDP method, as shown in Figure 6.27. Then, click on **OK**.

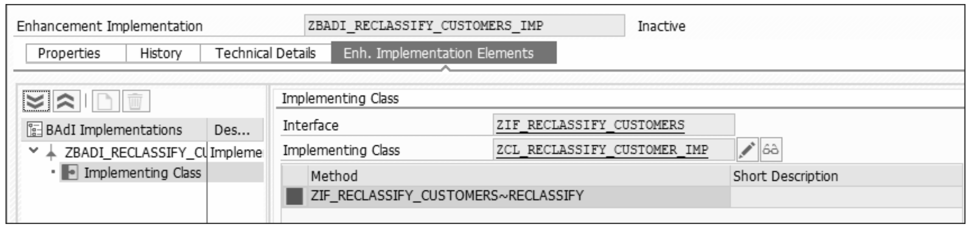


TRM(1)/200 Create BAdI Implementation

BAdI Implementation	ZBADI_RECLASSIFY_CUSTOMERS_IMP
Description	Implementation to redertime category
Implementing Class	ZCL_RECLASSIFY_CUSTOMER_IMP

Figure 6.27 Name and Describe BAdI Implementation and Specify Implementing Class

4. Click on **Save** or use the shortcut (**Ctrl + S**) to save the BAdI implementation.
5. Click on the **Implementing Class** (see Figure 6.28) to extend the AMDP method `RECLASSIFY` and incorporate the customer requirement using database-specific language (SQLScript). The customer category will be determined in this step, with `'Privilege Customer'` for customer type B and `'General Customer'` for customer type P, as shown in Figure 6.29.
6. Click on **Save** and **Activate** to activate the implementation. The active implementation is shown in Figure 6.30.



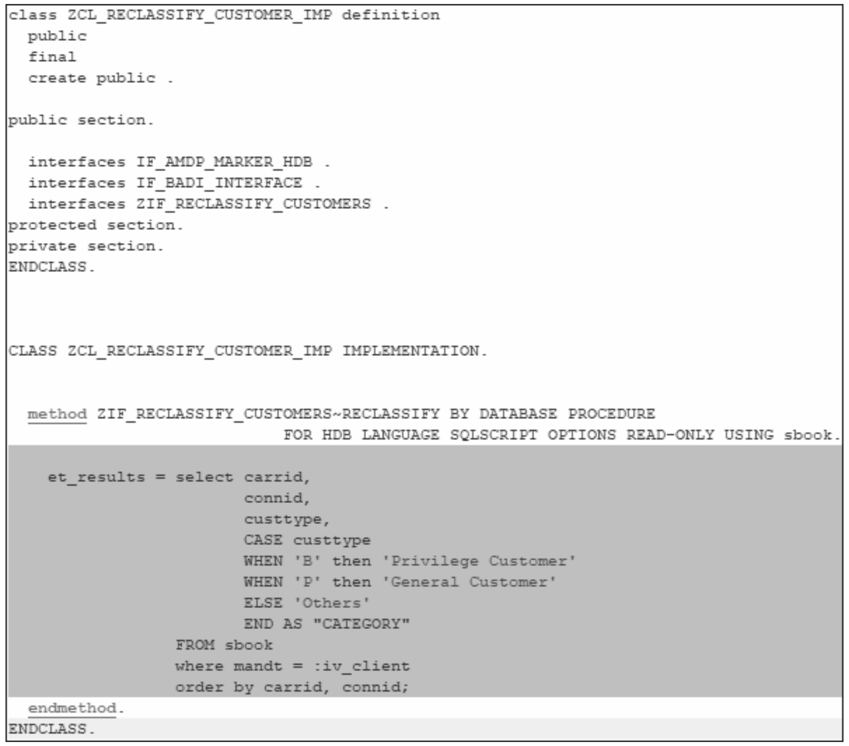
Enhancement Implementation: ZBADI_RECLASSIFY_CUSTOMERS_IMP Inactive

Implementing Class

Interface	ZIF_RECLASSIFY_CUSTOMERS
Implementing Class	ZCL_RECLASSIFY_CUSTOMER_IMP

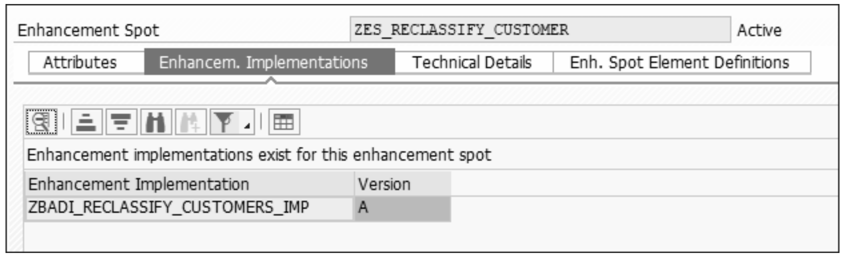
Method	Short Description
ZIF_RECLASSIFY_CUSTOMERS~RECLASSIFY	

Figure 6.28 Select the Implementing Class to Extend the AMDP Method



```
class ZCL_RECLASSIFY_CUSTOMER_IMP definition
  public
  final
  create public .

public section.

  interfaces IF_AMDP_MARKER_HDB .
  interfaces IF_BADI_INTERFACE .
  interfaces ZIF_RECLASSIFY_CUSTOMERS .
protected section.
private section.
ENDCLASS.



CLASS ZCL_RECLASSIFY_CUSTOMER_IMP IMPLEMENTATION.


  method ZIF_RECLASSIFY_CUSTOMERS~RECLASSIFY BY DATABASE PROCEDURE
                           FOR HDB LANGUAGE SQLSCRIPT OPTIONS READ-ONLY USING sbook.

    et_results = select carrid,
                        connid,
                        custtype,
                        CASE custtype
                        WHEN 'B' then 'Privilege Customer'
                        WHEN 'P' then 'General Customer'
                        ELSE 'Others'
                        END AS "CATEGORY"
                 FROM sbook
                 where mandt = :iv_client
                 order by carrid, connid;
  endmethod.
ENDCLASS.
```

Figure 6.29 Extend AMDP Method



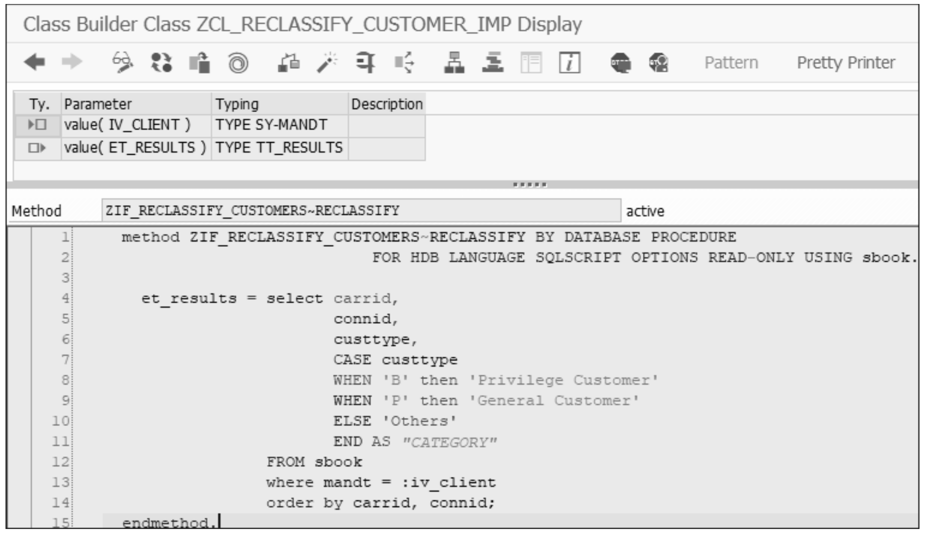
Enhancement Spot: ZES_RECLASSIFY_CUSTOMER Active

Enhancement implementations exist for this enhancement spot

Enhancement Implementation	Version
ZBADI_RECLASSIFY_CUSTOMERS_IMP	A

Figure 6.30 Active Enhancement Implementation

7. After implementing the BAdI definition, the business requirement is extended in the implementing class, to redetermine the business category as `'Privilege Customer'` or `'General Customer'` based on the customer type, as shown in Figure 6.31.
8. Execute F8 the application that calls the AMDP BAdI (see Figure 6.32) to display the customer classification data, as shown in Figure 6.33. The application program reclassifies the customer category to `'Privilege Customer'` or `'General Customer'` category using BAdI extensions.

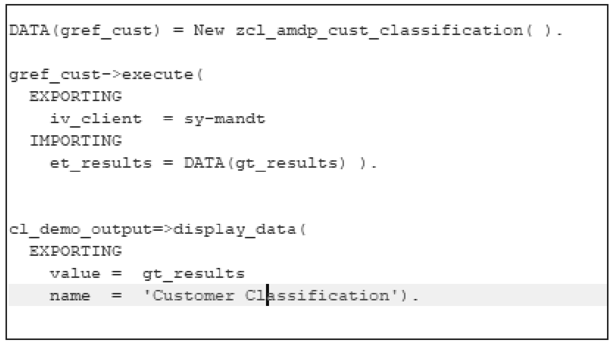


Ty.	Parameter	Typing	Description
	value(IV_CLIENT)	TYPE SY-MANDT	
	value(ET_RESULTS)	TYPE TT_RESULTS	

Method ZIF_RECLASSIFY_CUSTOMERS~RECLASSIFY active

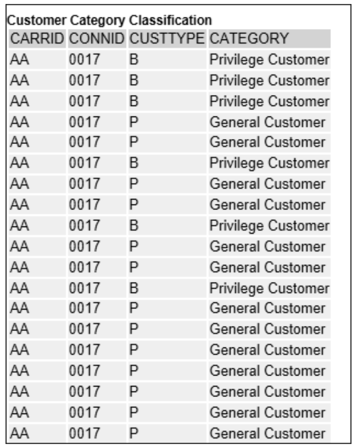
```
  method ZIF_RECLASSIFY_CUSTOMERS~RECLASSIFY BY DATABASE PROCEDURE
                              FOR HDB LANGUAGE SQLSCRIPT OPTIONS READ-ONLY USING sbook.

    et_results = select carrid,
                        connid,
                        custtype,
                        CASE custtype
                        WHEN 'B' then 'Privilege Customer'
                        WHEN 'P' then 'General Customer'
                        ELSE 'Others'
                        END AS "CATEGORY"
                 FROM sbook
                 where mandt = :iv_client
                 order by carrid, connid;
  endmethod.
```

Figure 6.31 Implementing Class to Reclassify the Customer Category



```
DATA(gref_cust) = New zcl_amdp_cust_classification( ).

gref_cust->execute(
  EXPORTING
    iv_client  = sy-mandt
  IMPORTING
    et_results = DATA(gt_results) ).


cl_demo_output=>display_data(
  EXPORTING
    value =  gt_results
    name  =  'Customer Classification').
```

Figure 6.32 Create BAdI Implementation

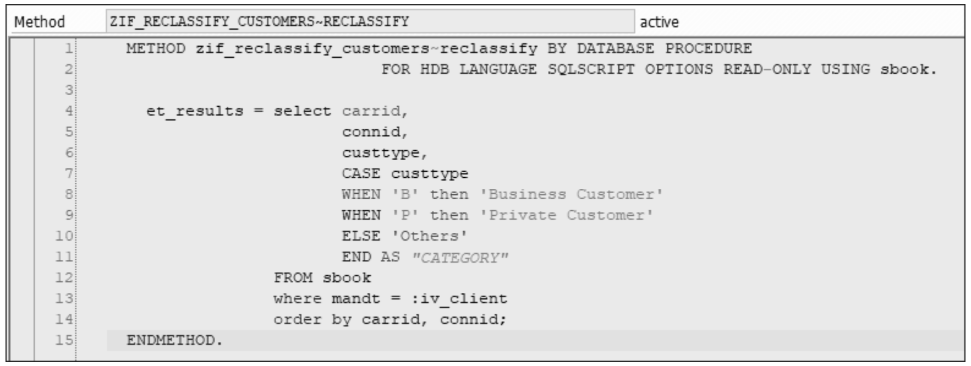


Customer Category Classification

CARRID	CONNID	CUSTTYPE	CATEGORY
AA	0017	B	Privilege Customer
AA	0017	B	Privilege Customer
AA	0017	B	Privilege Customer
AA	0017	P	General Customer
AA	0017	P	General Customer
AA	0017	B	Privilege Customer
AA	0017	P	General Customer
AA	0017	P	General Customer
AA	0017	B	Privilege Customer
AA	0017	P	General Customer
AA	0017	P	General Customer
AA	0017	B	Privilege Customer
AA	0017	P	General Customer
AA	0017	P	General Customer
AA	0017	P	General Customer
AA	0017	P	General Customer
AA	0017	P	General Customer
AA	0017	P	General Customer
AA	0017	P	General Customer

Figure 6.33 Resulting Customer Category Classification

9. If the implementation as shown in Figure 6.30 is deactivated or no active customer implementation exists for the BAdI definition. Then the application program calls the default fallback implementation as shown in Figure 6.34. This implementation categorizes customers as either `'Business Customer'` or `'Private Customer'` based on customer type, as shown in Figure 6.34.



Method ZIF_RECLASSIFY_CUSTOMERS~RECLASSIFY active

```
  METHOD zif_reclassify_customers~reclassify BY DATABASE PROCEDURE
                              FOR HDB LANGUAGE SQLSCRIPT OPTIONS READ-ONLY USING sbook.

    et_results = select carrid,
                        connid,
                        custtype,
                        CASE custtype
                        WHEN 'B' then 'Business Customer'
                        WHEN 'P' then 'Private Customer'
                        ELSE 'Others'
                        END AS "CATEGORY"
                 FROM sbook
                 where mandt = :iv_client
                 order by carrid, connid;
  ENDMETHOD.
```

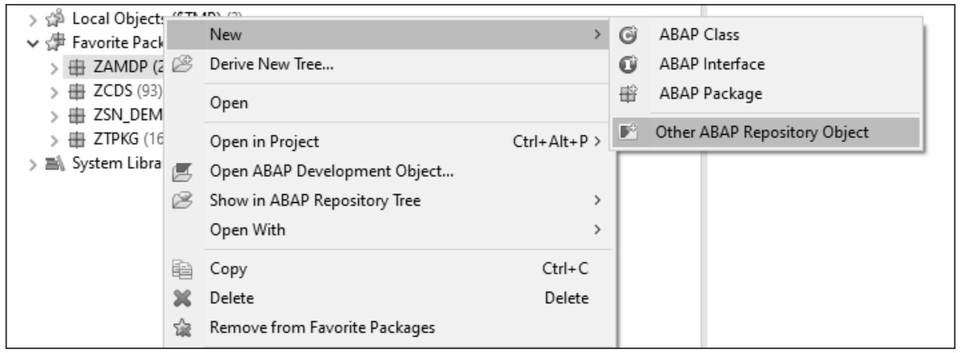
Figure 6.34 Default Fallback Implementation (No Active Customer Implementation)

6.3.3 AMDP BAdI Definition

In general, a software provider will anticipate extension points for extending AMDP procedures. SAP also allows you to create AMDP BAdIs for enhancement spots for your applications using the Eclipse-based editor in ADT or through the Enhancement Builder (Transaction SE20). The BAdI definition should be categorized as an AMDP BAdI and should conform to the prerequisites shown earlier in Figure 6.16 and Figure 6.17.

Let's walk through the steps for creating a BAdI definition to classify a customer into a category based on the customer type. Follow these steps:

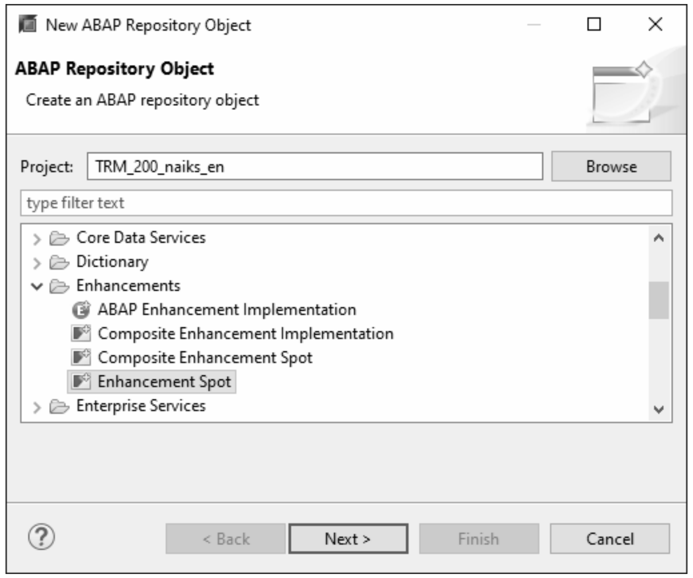
1. First, we'll create an enhancement spot by right-clicking on the package name and selecting **New • Other ABAP Repository Object**, as shown in Figure 6.35.



Local Objects (\$TMP) (2)
Favorite Pack
ZAMDP (2
ZCDS (93)
ZSN_DEM
ZTPKG (16
System Libra
New
Derive New Tree...
Open
Open in Project Ctrl+Alt+P
Open ABAP Development Object...
Show in ABAP Repository Tree
Open With
Copy Ctrl+C
Delete Delete
Remove from Favorite Packages
ABAP Class
ABAP Interface
ABAP Package
Other ABAP Repository Object

Figure 6.35 Select Other ABAP Repository Object

2. Choose the object type **Enhancement Spot** and click on **Next** to create the enhancement spot, as shown in Figure 6.36.



New ABAP Repository Object

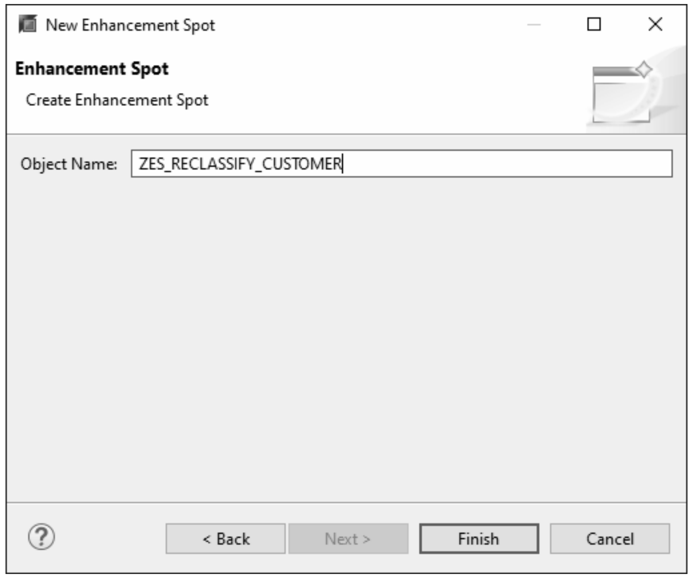
ABAP Repository Object

Create an ABAP repository object

Project: TRM_200_naiks_en

Figure 6.36 Create an Enhancement Spot using ADT in SAP HANA Studio

3. To create the enhancement spot, maintain the **Object Name** field and click on **Next**, as shown in Figure 6.37.



New Enhancement Spot

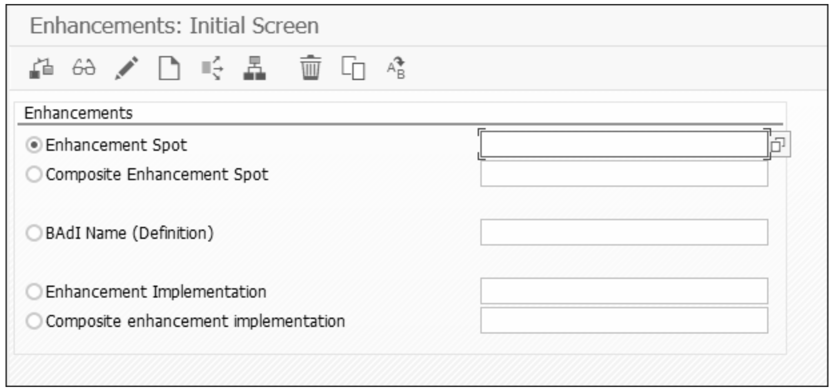
Enhancement Spot

Create Enhancement Spot

Object Name: ZES_RECLASSIFY_CUSTOMER

Figure 6.37 Specify Object Name for the Enhancement Spot using ADT in SAP HANA Studio

4. Alternatively, you can also use Transaction SE20 to create the BAdI definition (see Figure 6.38).



Enhancements: Initial Screen

Enhancements

● Enhancement Spot

○ Composite Enhancement Spot

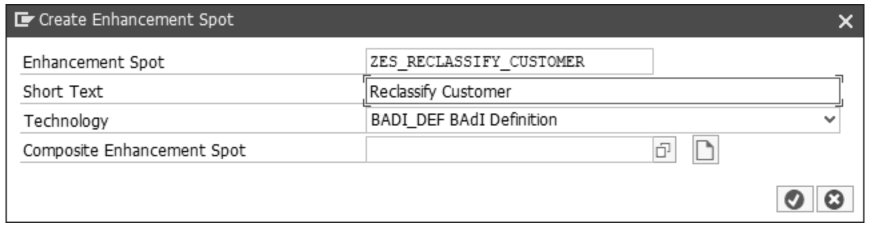
○ BAdI Name (Definition)

○ Enhancement Implementation

○ Composite enhancement implementation

Figure 6.38 Create an Enhancement Spot using Transaction SE20

5. Next, specify a name for the enhancement spot and maintain the **Short Text** and **Technology** fields, as shown in Figure 6.39. To create the enhancement spot, click on **OK**.

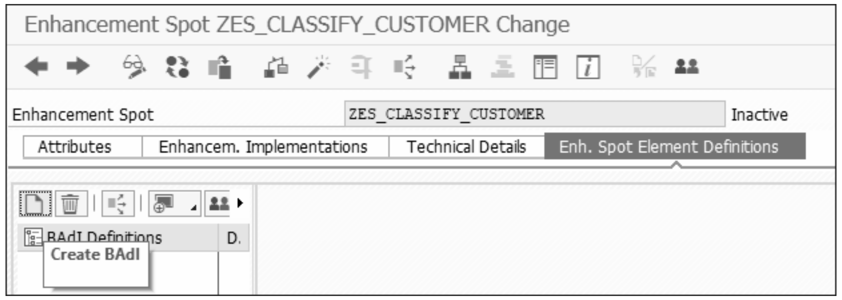


Create Enhancement Spot

Enhancement Spot	ZES_RECLASSIFY_CUSTOMER
Short Text	Reclassify Customer
Technology	BADI_DEF BAdI Definition
Composite Enhancement Spot	

Figure 6.39 Specify Name and Short Text for the Enhancement Spot Using Transaction SE20

6. Under the **Enh. Spot Element Definitions** tab, as shown in Figure 6.40, click on **Create BAdI** to create the definition.

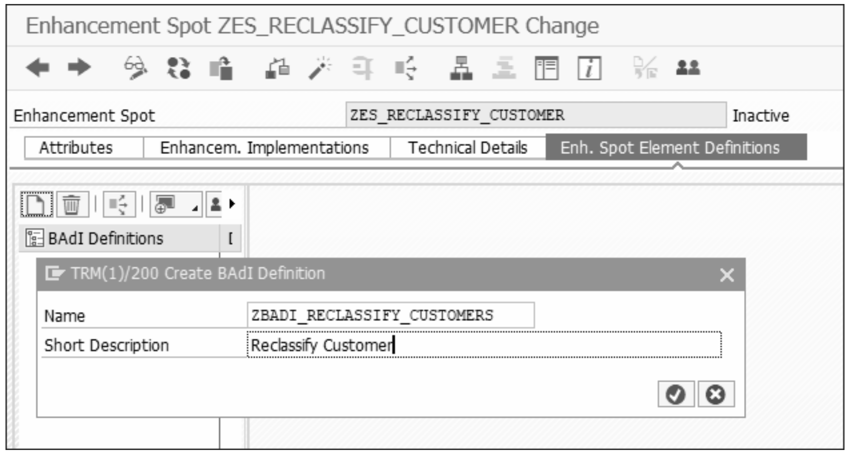


Enhancement Spot ZES_CLASSIFY_CUSTOMER Change

Enhancement Spot ZES_CLASSIFY_CUSTOMER Inactive

Figure 6.40 Create a BAdI Definition

7. Provide the BAdI a name and a short description and click on **OK** to create the definition, as shown in Figure 6.41.



Enhancement Spot ZES_RECLASSIFY_CUSTOMER Change

Enhancement Spot ZES_RECLASSIFY_CUSTOMER Inactive

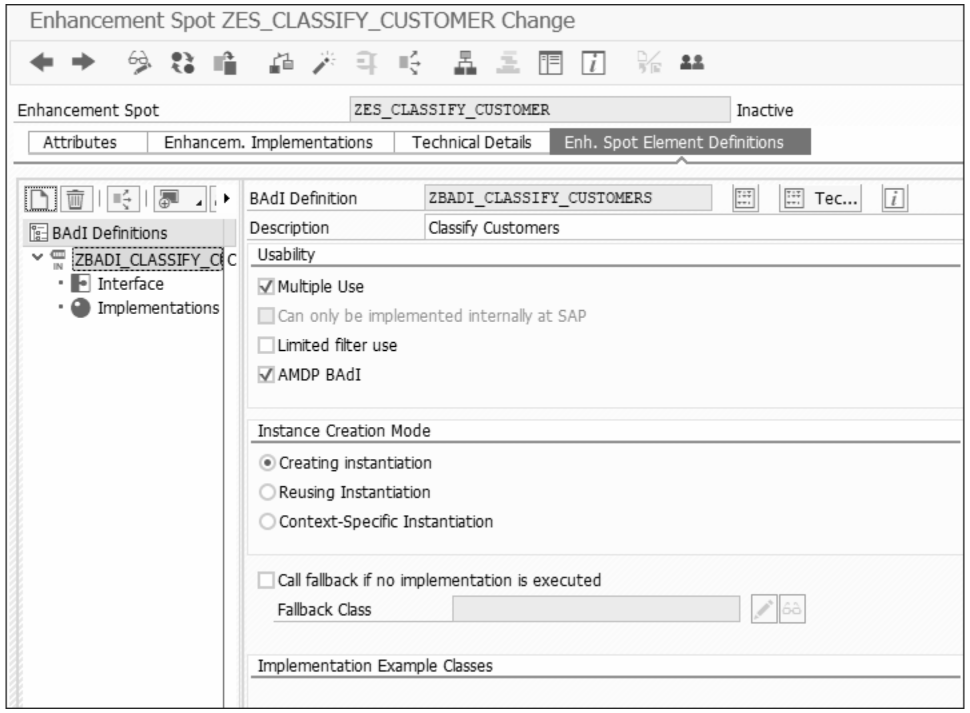
TRM(1)/200 Create BAdI Definition

Name ZBADI_RECLASSIFY_CUSTOMERS

Short Description Reclassify Customer

Figure 6.41 Specify BAdI Name and Short Description

8. Under the **Usability** section, categorize the BAdI definition as an AMDP BAdI by selecting the **AMDP BAdI** checkbox, as shown in Figure 6.42. You cannot classify the BAdI as filter dependent as the AMDP BAdI does not support filter functionality.



Enhancement Spot ZES_CLASSIFY_CUSTOMER Change

Enhancement Spot ZES_CLASSIFY_CUSTOMER Inactive

BAdI Definition ZBADI_CLASSIFY_CUSTOMERS

Description Classify Customers

Usability

☑ Multiple Use
☐ Can only be implemented internally at SAP
☐ Limited filter use
☑ AMDP BAdI

Instance Creation Mode

● Creating instantiation
○ Reusing Instantiation
○ Context-Specific Instantiation

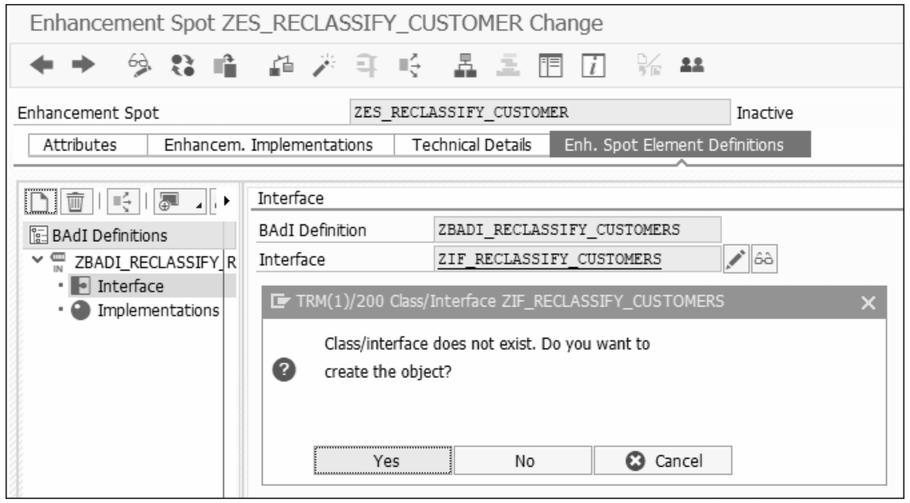
☐ Call fallback if no implementation is executed

Fallback Class

Implementation Example Classes

Figure 6.42 Categorize BAdI Definition as AMDP BAdI

9. Click on the **Interface** node under **BAdI Definitions** and specify the BAdI interface name ZIF_RECLASSIFY_CUSTOMERS and then click on **Yes** to create the BAdI interface, as shown in Figure 6.43.



Enhancement Spot ZES_RECLASSIFY_CUSTOMER Change

Enhancement Spot ZES_RECLASSIFY_CUSTOMER Inactive

Interface

BAdI Definition ZBADI_RECLASSIFY_CUSTOMERS

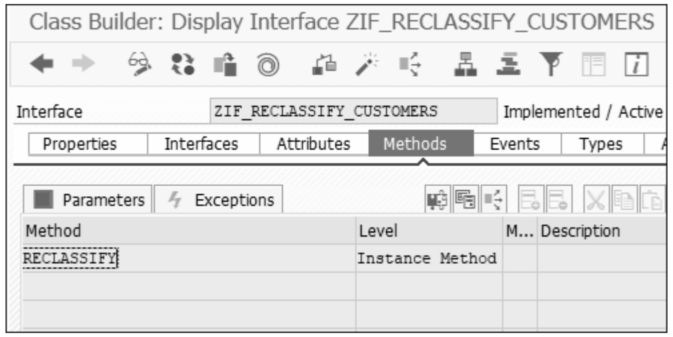
Interface ZIF_RECLASSIFY_CUSTOMERS

TRM(1)/200 Class/Interface ZIF_RECLASSIFY_CUSTOMERS

Class/interface does not exist. Do you want to create the object?

Figure 6.43 Create a BAdI Interface

10. Click on the interface name ZIF_RECLASAIFY_CUSTOMERS to define the AMDP method RECLASSIFY in the BAdI interface, as shown in Figure 6.44.



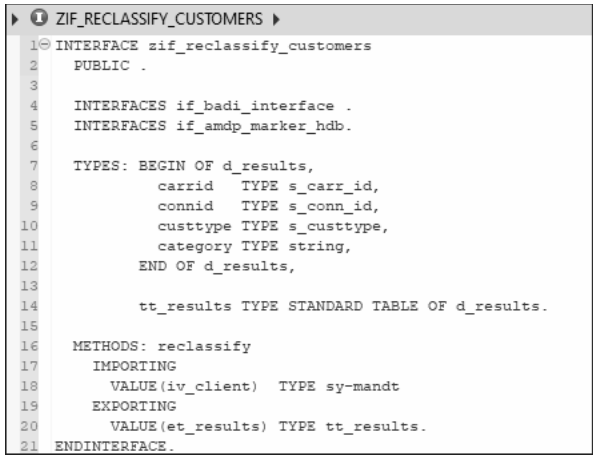
Class Builder: Display Interface ZIF_RECLASSIFY_CUSTOMERS

Interface ZIF_RECLASSIFY_CUSTOMERS Implemented / Active

Method	Level	M...	Description
RECLASSIFY	Instance Method		

Figure 6.44 Define AMDP Method in BAdI Interface

11. In the BAdI interface, as shown in Figure 6.45, include the AMDP marker interface IF_AMDP_MARKER_HDB for the database for which the procedure is to be created, In our case, this is specific to the SAP HANA database.

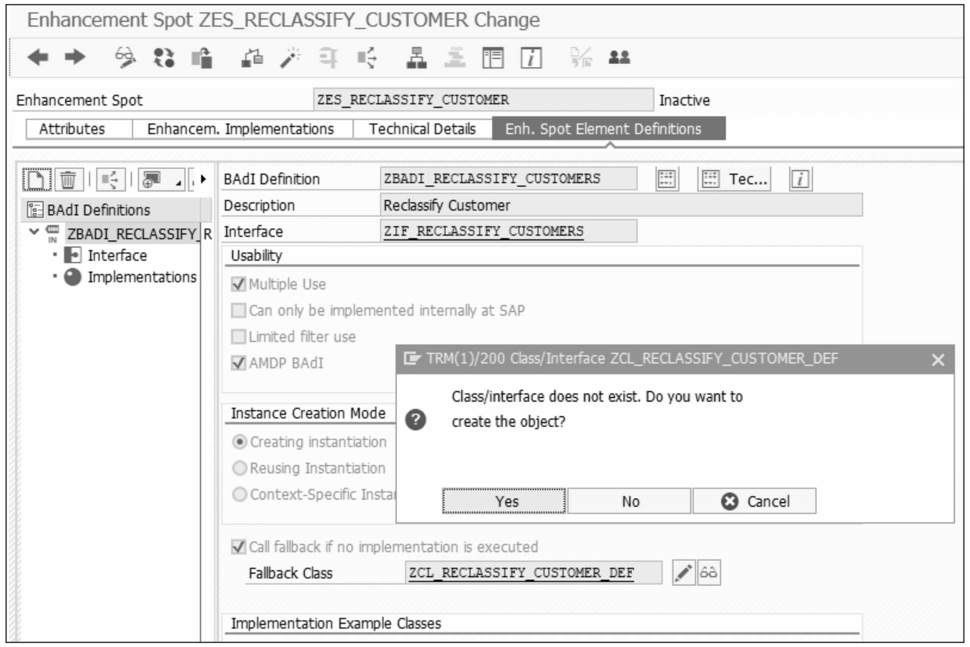


ZIF_RECLASSIFY_CUSTOMERS

```
INTERFACE zif_reclassify_customers
  PUBLIC .

  INTERFACES if_badi_interface .
  INTERFACES if_amdp_marker_hdb.

  TYPES: BEGIN OF d_results,
           carrid   TYPE s_carr_id,
           connid   TYPE s_conn_id,
           custtype TYPE s_custtype,
           category TYPE string,
         END OF d_results,

         tt_results TYPE STANDARD TABLE OF d_results.

  METHODS: reclassify
    IMPORTING
      VALUE(iv_client)  TYPE sy-mandt
    EXPORTING
      VALUE(et_results) TYPE tt_results.
ENDINTERFACE.
```

Figure 6.45 AMDP Method Definition

12. Finally, click on the fallback class `ZCL_RECLASSIFY_CUSTOMER_DEF`. Click **Yes** to create the fallback class, as shown in Figure 6.46. Then, encapsulate the business logic in the AMDP method `ZIF_RECLASSIFY_CUSTOMERS~RECLASSIFY`, as shown in Figure 6.47. If no active implementation is created for the BAdI definition, then the runtime environment calls the active implementation in the fallback class when the AMDP BAdI is called.



Enhancement Spot ZES_RECLASSIFY_CUSTOMER Change

Enhancement Spot: ZES_RECLASSIFY_CUSTOMER Inactive

BAdI Definition: ZBADI_RECLASSIFY_CUSTOMERS

Description: Reclassify Customer

Interface: ZIF_RECLASSIFY_CUSTOMERS

Usability

☑ Multiple Use

□ Can only be implemented internally at SAP

□ Limited filter use

☑ AMDP BAdI

Instance Creation Mode

● Creating instantiation

○ Reusing Instantiation

○ Context-Specific Insta

TRM(1)/200 Class/Interface ZCL_RECLASSIFY_CUSTOMER_DEF

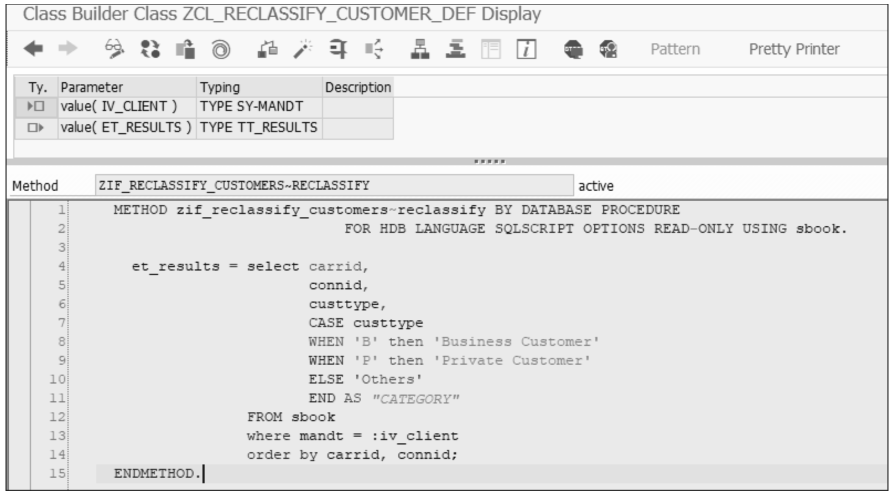
Class/interface does not exist. Do you want to create the object?

☑ Call fallback if no implementation is executed

Fallback Class: ZCL_RECLASSIFY_CUSTOMER_DEF

Implementation Example Classes

Figure 6.46 Create a Fallback Class



Class Builder Class ZCL_RECLASSIFY_CUSTOMER_DEF Display

Ty.	Parameter	Typing	Description
▸□	value(IV_CLIENT)	TYPE SY-MANDT	
□▸	value(ET_RESULTS)	TYPE TT_RESULTS	

Method: ZIF_RECLASSIFY_CUSTOMERS~RECLASSIFY active

```
  METHOD zif_reclassify_customers~reclassify BY DATABASE PROCEDURE
                             FOR HDB LANGUAGE SQLSCRIPT OPTIONS READ-ONLY USING sbook.

    et_results = select carrid,
                        connid,
                        custtype,
                        CASE custtype
                        WHEN 'B' then 'Business Customer'
                        WHEN 'P' then 'Private Customer'
                        ELSE 'Others'
                        END AS "CATEGORY"
                 FROM sbook
                 where mandt = :iv_client
                 order by carrid, connid;
  ENDMETHOD.
```

Figure 6.47 Encapsulate Business Logic in AMDP Method

Listing 6.8 illustrates an AMDP interface `ZIF_RECLASSIFY_CUSTOMERS` definition where an AMDP Method `RECLASSIFY` has been defined, the AMDP method `RECLASSIFY` is implemented in the fallback class `ZCL_RECLASSIFY_CUSTOMER_DEF` to encapsulate the business logic written in database specific SQL language to reclassify customers.

```
*AMDP Interface Definition
INTERFACE zif_reclassify_customers
  PUBLIC .
  INTERFACES if_badi_interface .
  INTERFACES if_amdp_marker_hdb.

  TYPES: BEGIN OF d_results,
           carrid   TYPE s_carr_id,
           connid   TYPE s_conn_id,
           custtype TYPE s_custtype,
           category TYPE string,
         END OF d_results,

         tt_results TYPE STANDARD TABLE OF d_results WITH EMPTY KEY.

  METHODS: reclassify
    IMPORTING
      VALUE(iv_client)  TYPE sy-mandt
    EXPORTING
      VALUE(et_results) TYPE tt_results.
ENDINTERFACE.
*AMDP Method Implementation in the fallback Class ZCL_RECLASSIFY_CUSTOMER_DEF
  METHOD zif_reclassify_customers~reclassify BY DATABASE PROCEDURE
```

```
                            FOR HDB LANGUAGE SQLSCRIPT OPTIONS READ-ONLY
USING sbook.

    et_results = select carrid,
                        connid,
                        custtype,
                        CASE custtype
                        WHEN 'B' then 'Business Customer'
                        WHEN 'P' then 'Private Customer'
                        ELSE 'Others'
                        END AS "CATEGORY"
                 FROM sbook
                 where mandt = :iv_client
                 order by carrid, connid;
  ENDMETHOD.
```

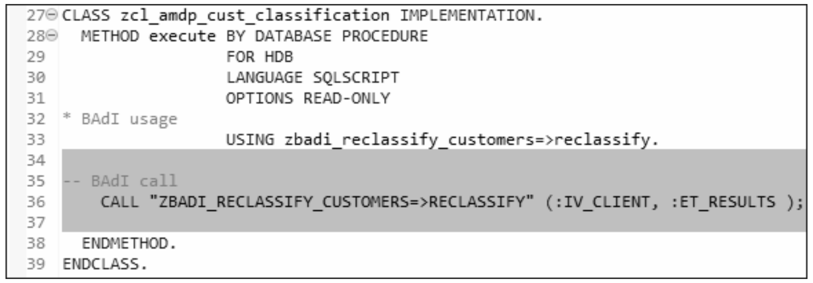
Listing 6.8 BADI Interface and AMDP Method Implementation

6.3.4 AMDP BAdI Calls

Software or extension providers usually call the AMDP method of an AMDP BAdI interface in another AMDP method or an application program to support modification-free extensions to AMDPs. The AMDP framework then generates a database procedure for each AMDP BAdI. An AMDP BAdI's name consists of the BAdI name and the interface method with the `=>` separator.

If you want to call an AMDP BAdI in an AMDP method, its usage must be first defined in the AMDP method implementation with the `USING` clause. Inside the AMDP method body, the BAdI call is invoked by specifying the BAdI definition and method name in uppercase and with the `=>` separator. The BAdI call is enclosed in double quotation marks, and furthermore, importing, exporting, and changing parameters are passed to the interface method. To define the usage of an AMDP BAdI, follow these steps:

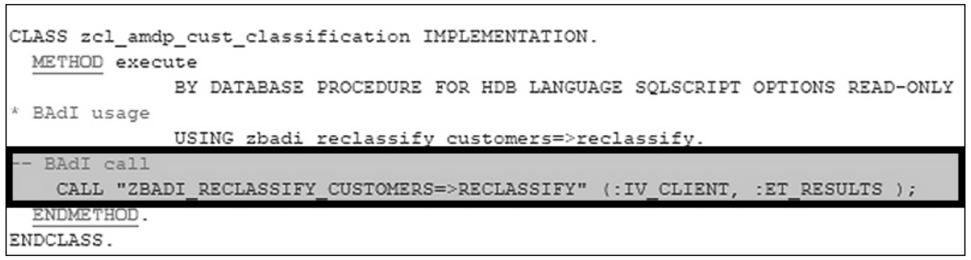
1. In the AMDP method implementation, the usage of the BAdI definition is specified in the `USING` clause, as shown in Figure 6.48.



```
CLASS zcl_amdp_cust_classification IMPLEMENTATION.
  METHOD execute BY DATABASE PROCEDURE
                 FOR HDB
                 LANGUAGE SQLSCRIPT
                 OPTIONS READ-ONLY
* BAdI usage
                 USING zbadi_reclassify_customers=>reclassify.

-- BAdI call
    CALL "ZBADI_RECLASSIFY_CUSTOMERS=>RECLASSIFY" (:IV_CLIENT, :ET_RESULTS );

  ENDMETHOD.
ENDCLASS.
```

Figure 6.48 BAdI Definition Usage in an AMDP Method

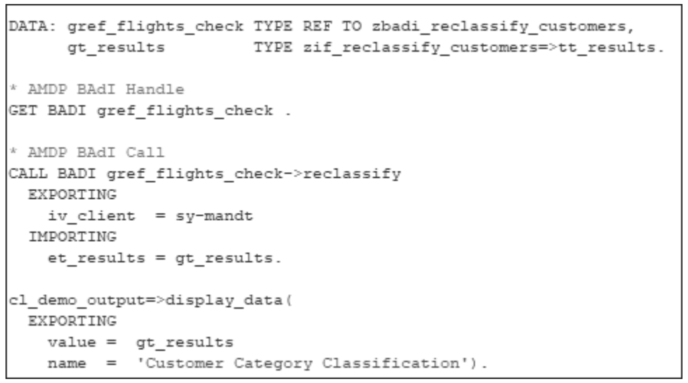
2. The BAdI call is invoked by specifying both the BAdI definition and the method name in uppercase and with the => separator. As shown in Figure 6.49, the BAdI call is enclosed in double quotation marks, and furthermore, importing, exporting, and changing parameters are passed to the interface method.



```
CLASS zcl_amdp_cust_classification IMPLEMENTATION.
  METHOD execute
            BY DATABASE PROCEDURE FOR HDB LANGUAGE SQLSCRIPT OPTIONS READ-ONLY
* BAdI usage
            USING zbadi_reclassify_customers=>reclassify.
-- BAdI call
    CALL "ZBADI_RECLASSIFY_CUSTOMERS=>RECLASSIFY" (:IV_CLIENT, :ET_RESULTS );
  ENDMETHOD.
ENDCLASS.
```

Figure 6.49 BAdI Call in an AMDP Method

3. Finally, as shown in Figure 6.50, the AMDP BAdI call is invoked in an ABAP application to derive the customer category classification.



```
DATA: gref_flights_check TYPE REF TO zbadi_reclassify_customers,
      gt_results         TYPE zif_reclassify_customers=>tt_results.

* AMDP BAdI Handle
GET BADI gref_flights_check .

* AMDP BAdI Call
CALL BADI gref_flights_check->reclassify
  EXPORTING
    iv_client  = sy-mandt
  IMPORTING
    et_results = gt_results.

cl_demo_output=>display_data(
  EXPORTING
    value =  gt_results
    name  =  'Customer Category Classification').
```

Figure 6.50 AMDP BAdI Call Invoked in ABAP Application

The source code shown in Listing 6.9 illustrates a BAdI AMDP method call in another AMDP method.

```
CLASS zcl_amdp_cust_classification DEFINITION
  PUBLIC
  FINAL
  CREATE PUBLIC.
  PUBLIC SECTION.
* Marker Interface
    INTERFACES if_amdp_marker_hdb.

* Type Definition
    TYPES: BEGIN OF d_results,
             carrid   TYPE s_carr_id,
             connid   TYPE s_conn_id,
             custtype TYPE s_custtype,
             category TYPE string,
           END OF d_results,
* Table Type
           tt_results TYPE STANDARD TABLE OF d_results WITH EMPTY KEY.
* AMDP Method
    METHODS: execute
      IMPORTING
        VALUE(iv_client)  TYPE sy-mandt
      EXPORTING
        VALUE(et_results) TYPE tt_results.
  PROTECTED SECTION.
  PRIVATE SECTION.
ENDCLASS.

CLASS zcl_amdp_cust_classification IMPLEMENTATION.
  METHOD execute BY DATABASE PROCEDURE
                    FOR HDB
                    LANGUAGE SQLSCRIPT
                    OPTIONS READ-ONLY
* Specify BAdI Usage before Call
                    USING zbadi_reclassify_customers=>reclassify.
-- BAdI Call
    CALL "ZBADI_RECLASSIFY_CUSTOMERS=>RECLASSIFY"
         (:IV_CLIENT, :ET_RESULTS );
  ENDMETHOD.
```

Listing 6.9 BAdI Method Call in an AMDP Method

As shown in Listing 6.10, an AMDP BAdI method call in any application is similar to a kernel BAdI call using a `CALL BAdI` or `GET BAdI` statement.

```
REPORT zamdp_cust_classification.

DATA: gref_flights_check TYPE REF TO zbadi_reclassify_customers,
      gt_results         TYPE zif_reclassify_customers=>tt_results.

* AMDP BAdI Handle
GET BADI gref_flights_check.

* AMDP BAdI Call
CALL BADI gref_flights_check->reclassify
  EXPORTING
    iv_client  = sy-mandt
```

```
  IMPORTING
    et_results = gt_results.

cl_demo_output=>display_data(
  EXPORTING
    value =  gt_results
    name  =  'Customer Category Classification').
```

Listing 6.10 BAdI Method Call in an Application Program

6.4 Exception Handling

Like any other traditional method call, an AMDP method call may produce ABAP runtime errors. Even though these calls are executed directly in the SAP HANA database, runtime errors can be traced and analyzed in Transaction ST22. This transaction provides additional AMDP-relevant information in the **Database Procedure (AMDP) Information** area under **ABAP Developer View**.

These errors can occur for several reasons, such as a version mismatch with the stored procedure, while creating or executing the method. Other reasons include missing authorizations or database connectivity errors.

Exceptions are raised when any such error occurs either by the runtime environment or by the method definition's `RAISE` exception statement. Therefore, exceptions should be handled in the calling program to avoid runtime errors during program execution.

Figure 6.51 shows the exception classes that can be handled within an AMDP method to avoid runtime errors.

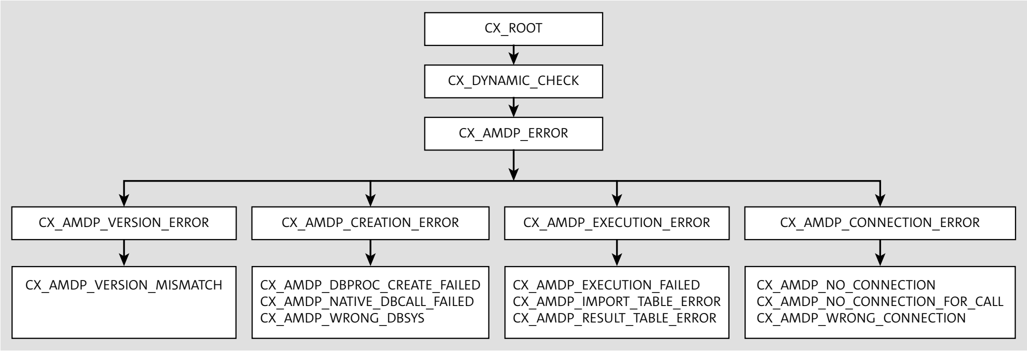


Figure 6.51 AMDP Exception Classes

The methods are prefixed with `CX_AMDP`, indicating these classes are exception classes for handling errors during AMDP calls. These exception classes belong to the `CX_DYNAMIC_CHECK` category and must be declared explicitly using the `RAISING` addition in the definition of an AMDP method and handled when called in the application program.

Contents

3 Development Environments 51

4 Native SQL 103

5 SQLScript Programming 123

6 ABAP-Managed Database Procedures 159

7 Modeling 219

8 Core Data Services 291

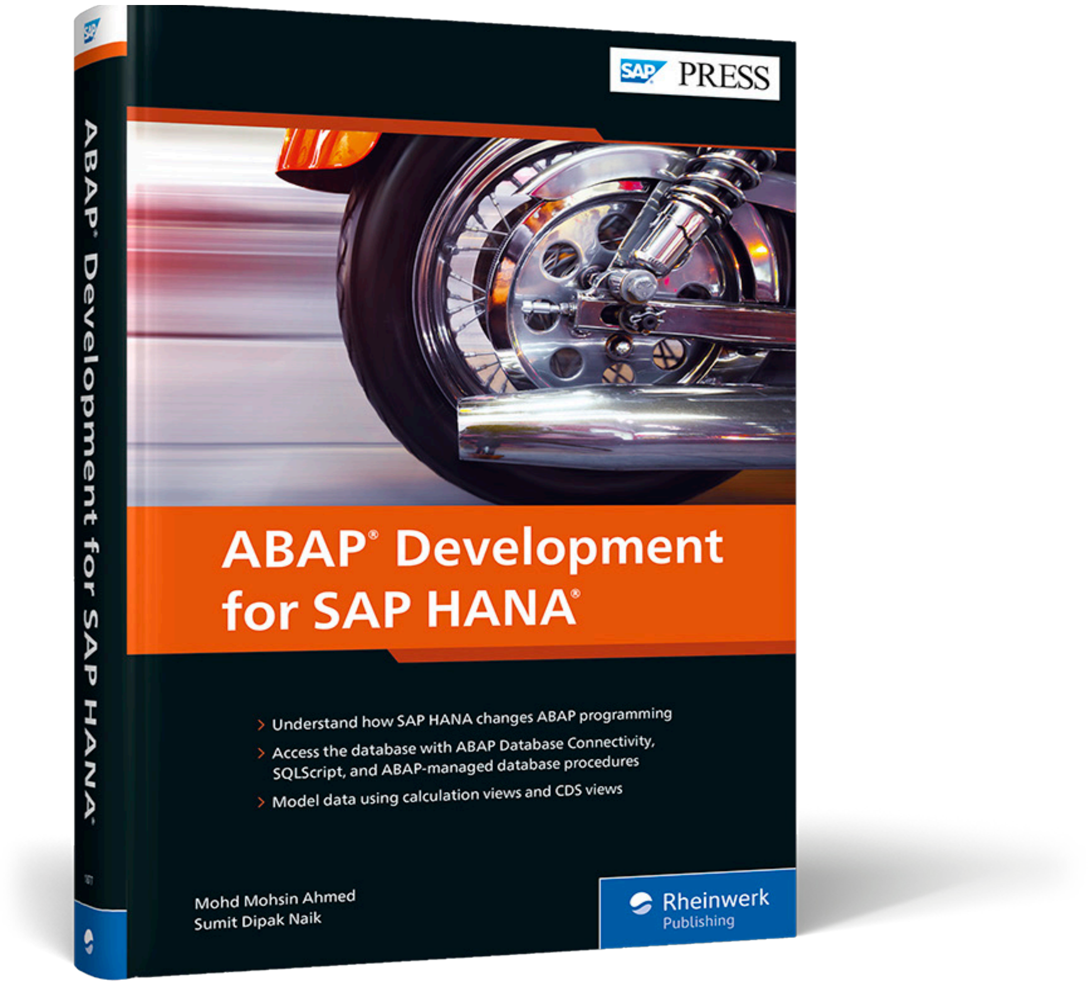
9 Open SQL and ABAP SQL 401

10 Business Object Processing Framework 435

Index

SAP PRESS First-hand knowledge.

Rheinwerk Publishing


SAP PRESS
ABAP® Development for SAP HANA®
Understand how SAP HANA changes ABAP programming
Access the database with ABAP Database Connectivity, SQLScript, and ABAP-managed database procedures
Model data using calculation views and CDS views
Mohd Mohsin Ahmed
Sumit Dipak Naik
Rheinwerk Publishing
ABAP® Development for SAP HANA®

Mohd Mohsin Ahmed is an SAP HANA certified professional with more than 14 years of experience as an SAP technical consultant. He currently specializes in implementing SAP S/4HANA solutions with a variety of methodologies, across various industry sectors, including manufacturing, retail, food and beverage, life science, energy and utilities, public sector, and IT. He holds a degree in has a degree in computer science engineering from Jawaharlal Nehru Technological University in Hyderabad, India.

Sumit Dipak Naik is an experienced ABAP and SAP HANA certified professional. He has more than 16 years of technical consulting, solutioning, and management experience. His focus is in implementing SAP ERP and SAP S/4HANA solutions, across various industry sectors, including telecommunications, food and beverages, manufacturing, retail, life sciences, energy and utilities, public sector, and IT. He has extensive experience with different implementation methodologies, approaches, and accelerators.

Mohd Mohsin Ahmed, Sumit Dipak Naik

ABAP Development for SAP HANA

643 Pages, 2021, $89.95
ISBN 978-1-4932-1877-6

www.sap-press.com/4954