Cookbook for enhancing SAP Business Partner with additional customer/vendor fields

Created by Claus Fretz and Alfred Dewald (both AIS FS-BP)
Version 1.8 – July 13th, 2021

Get the latest version of this cookbook here:
https://wiki.wdf.sap.corp/wiki/pages/viewpage.action?pageId=1802800692

Change log

Version 1.8 – July 13th, 2021
- Chapter 4.5.5 – BDT Field Groups – BUS2
  --> Correct description how to enhance existing FMOD2 function modules with own logic

Version 1.7 – February 13th, 2018
- 4.10 Field modification for blocked customer/vendor

Version 1.6 – May 04th, 2017
- 1.4 Hyperlinks of document references updated
- 4.5.2 Necessary section to screen assignments in BDT customizing
- 4.1 Typo: component name CA-GTF-BTD corrected to CA-GTF-BDT

Version 1.5 – September 16th, 2016
- All Textual review
- 1.4 Added document reference table

Version 1.4 – September 01st, 2016
- All Textual review

Version 1.3 – August 19th, 2016
- New title of the cookbook
- 1. Added scenario C
- 1.3 Updated contacts and stakeholders
- 6. Add a comment to emphasize that database changes must be prevented when application is not active
- 6.2.2 Contact information if append fields need to be added to CVIS_EI_EXTERN
- 6.2.3 New section about foreign key checks for append fields
- 7 New chapter for mass processing

Version 1.2 – June 13th, 2016
- 5.2.1 Details on calling validation methods in PAI modules
- 6.1. Improved example implementation for determining current KUNNR / LIFNR
- 6.2.2 Details on supporting table like data (e.g. KNB1) added
  Setting update indicator properly

Version 1.1 – February 04th, 2016
- 5.2.1 Improved coding for Validation Objects, read arbitrary table data

Version 1.0 – November 24th, 2015

1 General Information

In SAP Business Suite (ERP 600 and Enhancement Packages), customer master data and vendor master data transactions such as FD01, FD02, FD03, XD01, XD02, XD03, FK01, FK02, FK03, XK01, XK02 and XK03 have been enhanced by extension-specific fields using the Business Add-In (BAdI) technology. In the customer and vendor master dialog transactions, these fields were integrated by adding additional sub-screens to the existing screens.

Moving to SAP S/4 HANA release, traditional customer/vendor master transactions are made obsolete and replaced by the business partner transaction BP. Because of this, all extension-specific fields have to be integrated into the business partner.

Important
There are three different scenarios, which have to be considered differently:

Scenario A	Integration of existing application-owned customer/vendor dependent tables
Scenario B	Integration of existing appends to customer/vendor core tables (e.g. appends to tables KNA1 or LFA1)
Scenario C*	*Integration of a new development into the SAP Business Partner**

**Scenario C is not in scope of this document. New requirements with the goal to enhance master data with application owned fields have to be development completely within the SAP Business Partner in all S4 OnPremise and S4 Cloud Edition releases.*

Please refer to the guidelines for enhancing the SAP Business Partner:

- *BDT (Business Data Toolset) – Developer's Manual*
- *Help Portal – Business Partner Extensibility*
- *Easy Enhancement Workbench (EEW)*

Please find links to these topics in reference table in chapter 1.4.
If you need support with enhancing the business partner, open an internal BCP message on component AP-MD-BP.

1.1 Used tools and frameworks

Enhancements to the business partner dialog (transaction BP) can be made by using the frameworks / tools mentioned below:

a. BDT (Business Data Tool Set) – Enhancing dialog screens in transaction BP
b. XO Framework (Extensible Objects) – Validating and storing screen data in memory
c. CVI Synchronization – Mapping of the data from BP to customer/vendor and saving to database

1.2 Idea in a nutshell

The BDT will be used to enhance the existing screens in transaction BP with the required additional fields, tables or checkboxes needed by your application. The XO framework will be used to validate and to store the data in the memory. BAdI implementations within the CVI synchronization will collect the data from XO memory and save it to the database, in case additional data is not part of the complex interface structure.

For mass processing the “Central API” is available. This interface provides the functionality for creating and updating master data including both core and application data.

1.3 Stakeholders and contacts

Name	Team	Topics
Alfred Dewald, D033094	AIS FS-BP	Dialog enhancement - BDT Screen data to memory - XO
Claus Fretz, D047682		
Alain Bacchi, I023145	AIS LO-MD-BP	CVI
Dinesh Ravindran, I036567	DEV LO-MD-BP	Customer/Vendor enhancements Central API
Vinutha Varadarajalyengar, I065383		

1.4 Document references

Topic	Reference
BDT (Business Data Tool Set)	BDT Developer’s Manual Link to SAP User Assistance
XO Framework (Extensible Objects)	SAP note “1623809 - Developer documentation for the XO framework” Link to SAP note 1623809
CVI (Customer-Vendor-Integration)	SAP note “956054 - BP_CVI: Customer/vendor integration as of ERP 6.00” Link to SAP note 956054 Help Portal Link to SAP Documentation
Easy Enhancement Workbench (EEW)	Help Portal Link to SAP Documentation
Business Partner Extensibility	Help Portal Link to SAP Documentation → Functions → Extensibility

Table of Contents

2 General integration overview

For details about the Extensible Objects (XO) tool see note 1623809 (the note contains a developer documentation as attachment). The complete XO customizing is maintained in transaction XO80. Only the XO business object type BUSINESS_PARTNER is relevant for the CVI enhancement.

The purpose of this chapter is to provide an overview over the necessary activities using the tools/frameworks mentioned in chapter 1.1. In chapters 4, 5 and 6 the detailed steps to be executed per tool/framework are described. The relation between the overview chapter and the detail chapters is as follows:

- chapter 2.1 (BDT integration) → details see chapter 4
- chapter 2.2 (XO integration) → details see chapter 5
- chapter 2.3 (Saving data to database) → details see chapter 6

2.1 BDT Integration

This sub-chapter provides the overview of the BDT integration of the CVI. You can find the detailed activities in Chapter 4 below.

The integration of XO into BDT has been implemented in a generic way. The major part of the integration is implemented generically. All further new datasets or core table appends that are integrated into BDT using XO do not need to care about this part. If the integration is done as described below, the framework will take care of the generic logic automatically. Especially most of the BDT events are treated generically by the so-called BDT-adapter. The generic implementation is provided by class

- XO_BDT_ADAPTER (generic class for BDT integration).

On top of this, there are the classes

- FSBP_BDT_ADAPTER_INTERN (inheriting from XO_BDT_ADAPTER, several methods redefined): Specification for Financial Services Business Partner datasets
- Class CVI_BDT_ADAPTER_INTERN (inheriting from FSBP_BDT_ADAPTER, several methods redefined): Specification for CVI Business Partner datasets

Within the BDT adapter classes you find methods (generic_*; e.g. GENERIC_ISDAT or GENERIC_ISSTA), that are called by the BDT event function modules (XO_GENERIC_EVENT_* in transaction BUS7). These methods are generically responsible for processing the corresponding events for all Financial Services and CVI datasets. For these events, no individual implementation is necessary.

In addition, there are BDT settings that need to be implemented individually. For example in the corresponding view in BDT a PBO and a PAI module need to be registered for every view. Within these two function modules the selection of data for this dataset from XO memory as well as the saving of the changed data into XO memory later on are implemented. Further individual BDT settings that need to be considered are for example a new BDT application per extension (separate for customer and vendor enhancements), and of course the complete screen construction (screens, sections, views, field groups, datasets).

In addition dynpros (SE80-screens) need to be implemented in transaction SE80 including PBO and PAI logic within the BDT function group that is responsible for the corresponding BDT application. The already mentioned PBO- and PAI-function modules need to be implemented there as well.

You can find many more details including a precise step-by-step description about the BDT implementation in chapter 4 below.

2.2 XO overview

In this sub-chapter a rough overview of the XO implementation is provided. For more details have a look into chapter 5.

2.2.1 Scenario A – Integration of application owned tables

For every dataset (database table) of customer or vendor to be supported in transaction BP a memory object needs to be assigned in XO customizing. Basic elements of a memory object are the database table that shall be integrated into CVI (which usually already exists) and the class in which the memory of this database table is kept during runtime. This class will not yet be available and has to be implemented inheriting from one of the classes CVI_MO_CUSTOMER or CVI_MO_VENDOR.

Within every memory object class you need to make sure that

- for every field (for which at least one check is necessary) a separate method VALIDATE_<FIELD_NAME> is created (you can also create validation methods with combined field checks like: If field1 is not initial -> verify field2 is not initial)
- method VALIDATE_INTERN is redefined in a way that it calls SUPER->VALIDATE_INTERN at the beginning and afterwards all individual validation methods that were added in the current inheritance level.

Additionally a persistence object is needed to retrieve data from the database including a reference class representing the persistence object. A persistence object needs to be assigned to every memory object according to the following logic:

- If your table uses KUNNR or LIFNR as key field you can refer to one of the existing persistence objects CUSTOMER for table key KUNNR or VENDOR for table key LIFNR.
- If the key field in your table is a different one you have to create a new persistence object referring to a new class, which has to inherit from either CVI_PO_CUSTOMER or CVI_PO_VENDOR.

2.2.2 Scenario B – Integration of core table appends

Appends to customer/vendor core tables (KNA1, KNB1, LFA1, LFB1, …) are treated differently. For these fields the enhanced memory object class already exists and it would lead to technical problems if a new enhancement was created. Because of this for the integration of core table appends instead a validation object (VO) represented by a VO-class needs to be implemented in XO. The VO has the purpose of providing static validation methods, which can be used to validate the data of the append fields (e.g. in the corresponding PAI modules). Additionally the validation methods will automatically be called during the overall business partner validation.

2.3 Saving data to the database

For fields that are part of the complex customer/vendor interface (complex structure containing all core fields of customer/vendor) the database persistence is provided generically. There is no additional implementation effort. For saving data to the database that are not part of the complex interface the CVI synchronization provides several BAdIs, which have to be implemented in this case. Within these BAdI implementations, the data has to be retrieved from the XO memory and be either added to the corresponding core table exporting parameter or directly written to the database.

For more details about the BAdI implementations see chapter 6.

3 Switching of the enhancements

The standard process of enhancing the BP dialog with former customer/vendor fields is to include them into the standard roles FLCU00/01 or FLVN00/01. For performance reasons and to make sure that not all customers are confused with many UI changes they might not need, it is necessary to make the extensions switchable on (BDT-) application level.

A very simple approach to switch-off a BDT application and all connected BDT objects during runtime is the usage of BDT event APPLC. In a new APPLC function module, you need to implement a check if your extension is active. If this is not the case the value XAKTV for your BDT application should be set to space in the changing table CT_TBZ0A. As a consequence your UI extensions will not be visible to the user and all your BDT-related code will not be processed. This will help keeping business partner maintenance as lean as possible.

Naming convention:
<APPL_NAME>_BUPA_EVENT_APPLC (example: ISOC_BUPA_EVENT_APPLC)
Example – APPLC Function Module: In system ER9 have a look at FM FS05_BUPA_EVENT_APPLC.

Every APPLC function module needs to be registered in transaction BUS7 for event APPLC, assigning the corresponding application (to which the extension fields belong) and value ‘X’ in field ‘Call’. It is recommended to create a separate APPLC function module per BDT application. However, it is also possible to create one APPLC module for several applications. In this case, it is advisable to assign application BUP to the “collective” APPLC module.

4 Settings in BDT – Business Data Toolset

The Business Data Toolset (BDT) is used to integrate new fields into the business partner UI (transaction BP). The BDT integration involves the creation of various field groups, screens, views and sections so that the corresponding tabs and the screens appear in transaction BP. Using BDT you can create own objects or enhance existing standard objects.

The following aspects need to be considered when implementing application-specific enhancements in BDT (general transaction for BDT business partner control menu is /NBUPT).

4.1 General information about naming conventions in BDT

The central part in the naming for all BDT objects is the application name (<APPL_NAME>) defined in transaction BUS1 (example: ISOC).

All datasets, screens, sections and views are named with <APPL_NAME>number (example: ISOC01…ISOC99).

All event function modules need to have the following naming:
APPL_NAME>_BUPA_EVENT_<EVENT_NAME> (example: ISOC_BUPA_EVENT_FCODE).

All PBO/PAI function modules need to have the following naming:
<APPL_NAME>_BUPA_PBO_<VIEW_NAME> (example: ISOC_BUPA_PBO_ISOC01)
<APPL_NAME>_BUPA_PAI_<VIEW_NAME> (example: ISOC_BUPA_PAI_ISOC01).

Rules for the use of position numbers (e.g. when assigning function modules to BDT events)

- The objects assigned by help of position numbers will be processed in the order of the numbering (e.g. the views assigned to a section will be constructed in a way that the view with the lowest number is displayed first and all higher numbers below / same for the event function modules the module with the lowest number assigned to an event is processed first and all others later according to their number)
- Especially for the event function modules conflicts can occur because function modules from different software components are delivered in the same table. For screen elements this conflict does normally not occur because higher extensions normally create their own screen elements rather than assigning a SAP_APPL screen element to an SAP_ABA element.
- In order to make sure that the same number is not used twice in different software components the following rules apply:
  SAP_ABA: use position number with last 5 digits equal to 0 (e.g. 3.600.000 or 200.000)
  SAP_APPL: use position number with last 4 digits equal to 0, 5th digit from end needs to be unequal to 0 (e.g. 550.000 or 3.760.000)
  Industry extensions like EA-FINSERV or IS-MEDIA: Use position number with last 3 digits equal to 0, 4th digit needs to be unequal to 0 (e.g. 3.601.000 or 402.000)
  Further rules apply to SAP Partners (3rd digit unequal to 0) or customers (last two digits unequal 0).
- Field group numbers: In order to get a field group number range to be used for the field groups of your development please contact the BDT team (component CA-GTF-BDT, contact is Gurumoorthy H (manager of BDT team).

4.2 Own BDT application – BUS1 or BUPT

You need to create your own BDT application(s) and assign them to your objects (views, bp views, …). Do not add the generic CVI applications CVIC or CVIV. You can also decide to split your enhancement into different BDT applications. The BDT application is used to hide your enhancements in the UI in case your extension is not active. It necessary to use different BDT applications for customer and vendor enhancements.

4.3 Separate BDT datasets – BUS23 or BUPT

You need to create your own datasets. Do not use the generic CVI datasets CVIC*/CVIV*. A separate dataset is needed at least for the differentiation between general data, company code dependent data and sales area dependent data for each customer and vendor. However, you should split your enhancement into different datasets according to the business context. In SAP standard usually there

is one dataset per table (although in special cases deviations are possible). All data belonging to one view can only be assigned to the same dataset (because the assignment is made by the dataset assignment in the view).

4.4 Registering tables – BUSG or BUPT

Every table that is added to CVI needs to be registered in this transaction with the corresponding attributes (change document object, function module to get data, function module to collect data). If for the enhanced table (or set of tables) a change document object exists, this needs to be added in the corresponding field. In case no change document object exists, a new one needs to be created and integrated.

4.5 BDT Assignments of screen->section->view->field group->field

You should always create a new section with its own frame title. If fields fit to an already existing screen (tab), because they are to be seen in the same business context, the new section can be assigned to an existing screen. In this case you only need to create views and field groups in addition and assign everything (see below). In case you have fields that do not belong to the already existing tabs you can create new tabs (screens) and assign them to the corresponding screen sequence (transaction BUS6 – screen sequence BUP001 for general data, screen sequence FS0001 for company code data customer and vendor, screen sequence CVIC01 for sales data customer, screen sequence CVIV01 for sales data vendor).

4.5.1 BDT screen sequences – BUS6 or BUPT

In case you have a completely separate set of screens (tabs), you can create your own subheader-id (button in the subheader line in transaction BP). For this, you need a new screen sequence, which can be defined here. In addition, the relevant screens need to be assigned to the screen sequence here.

4.5.2 BDT screens – BUS5 or BUPT

In case you have fields that belong to a screen sequence, but do not fit into any of the existing tabs, you can create a new tab (screen) and assign it to an existing screen sequence. Assign your sections to the corresponding screens also in transaction BUS5.

When creating a new screen, it is necessary to assign the correct header section to this screen. The header section must be the first section in the screen, which means it must have the lowest item number. There are different header sections for general, company code, sales area and purchasing organization data which must be assigned:

Data Segment	**Header Section**	
Customer General	BUP009	Header Data (General Screens)
Customer Company Code	FI0201	Header Data (Company Code-Dependent Screens)
Customer Sales Area	CVIC00	Header Data (Sales Area-Dependent Screens)
Vendor General	BUP009	Header Data (General Screens)
Vendor Company Code	FI0201	Header Data (Company Code-Dependent Screens)
Vendor Purchasing Org.	CVIV00	Vendor: Header Data (Purchasing Organization)

4.5.3 BDT sections – BUS4 or BUPT

For any new field or set of fields, you need to create a section of your own with a speaking frame title (see column “Title” in transaction BUS4). Assign only those views to a section that fit together. If the business meaning is different, create a new section. Assign the views to the sections here as well.

4.5.4 BDT views – BUS3

The view is the central object in BDT. It contains the connection to the SE80-screen (in a function group) where the input fields are defined. For differentiation between the BDT elements and the SE80 elements, the German name for the SE80-screen, **dynpro**, is used here. The view also contains the PBO/PAI-

logic as well as an assignment of application and dataset. In addition, the fields are assigned to the views via the field groups here.

General rule should be: Do not assign too many field groups to a view. Only assign more than one field group into one view if the fields belong to the same context. If in doubt, create different views for different field groups. This will help you very much later in maintenance!

In addition, you need to add a PBO and PAI function module here in which the PBO and PAI logic is processed. The coding in the PBO and PAI-modules can be mainly copied from the standard modules. Compare for example:

- CVIC_BUPA_PBO_CVIC01 / CVIC_BUPA_PAI_CVIC01 for general customer data
- CVIC_BUPA_PBO_CVIC15 / CVIC_BUPA_PAI_CVIC15 for sales area customer data
- CVIC_BUPA_PBO_CVIC30 / CVIC_BUPA_PAI_CVIC30 for company code customer data
- CVIV_BUPA_PBO_CVIV01 / CVIV_BUPA_PAI_CVIV01 for general vendor data
- CVIV_BUPA_PBO_CVIV30 / CVIV_BUPA_PAI_CVIV30 for company code vendor data
- CVIV_BUPA_PBO_CVIV71 / CVIV_BUPA_PAI_CVIV71 for purchasing vendor data

All these modules are constructed in the following way: In PBO first of all the data is taken from XO memory and transferred to the dynpro structure. After that in PAI module, the changed data is saved back to the local table and integrated in the current memory. After that, it is transferred back to XO memory for later saving. In addition, in PBO modules, the texts for F4-fields are also determined and in PAI modules at the end, the validation methods from the XO memory or validation object are called to validate the changes.

In addition to the PBO and PAI function modules there are – of course – also the PBO and PAI modules in the dynpro flow logic. Here you need to call the standard function modules provided by BDT:
BUS_PBO – to be called in a perform PBO, which in turn is called in the PBO module
BUS_PAI – to be called in a perform PAI, which in turn is called in the PAI module.
For an example, check the dynpros in function group CVI_FS_UI_CUSTOMER.

Attention: When you miss to call function modules BUS_PBO and BUS_PAI in the dynpro flow logic, there will be problems with field modification and the behavior of the fields in dialog (e.g. fields available for input in display mode or similar problems).

4.5.5 BDT field groups – BUS2

On field group level the fields from the dynpro are assigned.

General rule: only one field per field group! Only in exceptional cases, you should assign more than one field to one field group. Field modifications are done on the basis of field groups and two fields that belong to the same field group can only be set to “required“, “display“, “hidden“, “change“ or “unspecified“ together.

In addition, you need to add a function module for FMOD2 event in the field group. If you do not have any specific field modification logic for your field groups you can add the standard FMOD-function modules here:

- CVIC_BUPA_EVENT_FMOD2/CVIV_BUPA_EVENT_FMOD2 for general data
- CVIC_BUPA_EVENT_FMOD2_SALES/CVIV_BUPA_EVENT_FMOD2_PORG for sales data/purchasing org data
- CVIC_BUPA_EVENT_FMOD2_CC / CVIV_BUPA_EVENT_FMOD2_CC for company code data.

In case you have an additional requirement for field modification setting create an own FMOD2 function module in your BDT function group and – as a first step within your code – call the current core FMOD2 function module at the beginning. After that enhance it by your logic. When doing so you can rely on the incoming field status (IN_STATUS) for all general field groups and only need to add logic in case you would like your own field groups to deviate from this.

4.6 BDT events – BUS7

The big majority of BDT events is handled automatically by class XO_BDT_ADAPTER and corresponding enhancements (classes inheriting from XO_BDT_ADAPTER, see chapter 2.1), provided you have implemented your XO-customizing correctly.

You only need to care about BDT events in case you do anything special like for example adding some function codes (buttons) for navigating to a popup or similar. In this case you need a separate event function module (in above example an FCODE-function module) and assign it to the corresponding event (e.g. FCODE).

In addition, it is necessary to implement additional event function modules for displaying the change documents for your table(s) in transaction BP. Please have a look into our standard CHGD* function modules and implement an analogous logic for your tables

- CVIC_BUPA_EVENT_CHGD1 / CVIV_BUPA_EVENT_CHGD1
- CVIC_BUPA_EVENT_CHGD3 / CVIV_BUPA_EVENT_CHGD3
- CVIC_BUPA_EVENT_CHGD4 / CVIV_BUPA_EVENT_CHGD4

For event CHGD2 no function module needs to be assigned. If your tables are correctly integrated in XO, function module XO_GENERIC_EVENT_CHGD2 will process the necessary logic generically.

Further event function modules might be needed once you start supporting the archiving/deletion of a business partner.

4.7 Additional functions – BUS9

You need this transaction in case you have buttons (function codes) in your application. In order to integrate a function code in the application the following steps are necessary:

1. Enter the function on your dynpro. Naming convention: The name of the push button needs to have the prefix “PUSH_”. Example: PUSH_BUTTON1.
2. In the screen painter details in field “FctCode”, you need to enter the name of the push button again.
3. In order to react to the selection of the button during runtime by a user you need to implement an FCODE function module (naming convention <APPL_NAME>_BUPA_EVENT_FCODE) and implement the following coding here

```
"set fcode as processed per default
e_xhandle = true.
case i_fcode.
  when 'BUTTON1'.
    "implement application-specific logic here
  when others.
    clear e_xhandle.
endcase.
```

   Please note that you need to remove the prefix “PUSH_” when catching the function code in your FCODE function module (==> WHEN ‘BUTTON1’ and **not** WHEN ‘PUSH_BUTTON1’).
4. In order to make sure that the push button can be influenced using the field modification adjustments, create a new field group (transaction BUS2) and assign the button to the field group using the “Field Group -> Fields” sub-menu. The name of the button (including the prefix) needs to be entered in column “field name”. Column “table” remains empty. In addition, you should assign an FMOD2-function module according to the rules described in chapter “BDT field groups – BUS2”.
5. In addition, you have to create a separate view for the button according to the rules above in chapter “BDT views – BUS3”. Maintain the dynpro in which the button has been implemented and assign the field group created in “4” to the view.
6. Finally, the function code needs to be made known to the BDT. For this, it has to be defined in transaction BUS9. The following steps need to be executed for this:
   a. Press button “New entries”.
   b. In field “Function code”, enter the name of your function code **without** the prefix (“PUSH_”).
   c. In field “Function text”, enter a text for your function code. You can leave field “screen sequence cat.” empty.
   d. After that, you have to decide if you would like to rely the visibility of the function code on either field group or view. Recommendation: use function “Activate using Status of

Field Group" and maintain visibility in section "Active per Status of Field Group" at the end of the page.

To have a reference to copy from you can have a look at the push button PUSH_CVIS_SEPA (BUS9), which is assigned to field group 309 and view CVIS01. The corresponding FCODE function module is CVIS_BUPA_EVENT_FCODE.

4.8 Assignments Dynpro field->DB field and DI field->DB field – BUSB and BUSB_DI

You need this transaction in case you have dynpro fields that have a different name than the corresponding database fields. BDT needs to be able to assign the fields, otherwise for example the required-fields check will not work.

4.9 BP_Views – BUSD

This is the technical view on the business partner roles (see views V_TB003 / V_TB003A in transaction SM30). You may think about creating your own roles instead of enhancing the standard roles. Then you would not need to switch your functionality.

In order to make sure that your enhanced fields are available in the corresponding roles add your datasets and applications to the BP views. Differentiate between customer and vendor roles here.

4.10 Field modification status for blocked customer/vendor master

S/4 releases support the blocking of customer and vendor master without blocking the assigned business partner. In such a case where an assigned customer/vendor is blocked while the business partner is not, all users with standard authorization will not have any access to exclusive customer/vendor based data in transaction BP. General data like names or addresses are still accessible and editable in the business partner, but any changes to these data will no longer be synchronized with the blocked customer/vendor master.

A user with "Data Privacy Officer" (DPO) authorizations, who opens a business partner which is not blocked itself but assigned to a blocked customer or vendor in transaction BP, has access to all customer/vendor data fields and sections. When the DPO-user switches transaction BP to change mode, all fields you have added to transaction BP in context of customer-vendor-integration will be editable (please note that changes to these fields will not be transferred to the blocked customer/vendor when saving and will therefore not be saved!). Nevertheless, this is not a consistent UI behavior and you should take the below measures to prevent the fields to be open for change. You need to create a function module which sets the field status of your fields to "display" in that case and assign this function module in BDT customizing, so that it is processed in FMOD1 event.

The function module should have the following naming convention:

<APPL_NAME>_BUPA_EVENT_FMOD1_DPP

Execute the following steps:

1. Make sure that SAP notes 2592806 and 2590430 are implemented in your system.
2. Create a new function module taking function module CVIS_BUPA_EVENT_FMOD1_DPP as a reference. If you copy the code and adjust replace the name of the BDT application to your own BDT application, your field groups will also be switched to display mode in case an assigned customer/vendor is blocked.
3. Afterwards register your function module in BDT:
    a. Open maintenance view V_TBZ3Q in SM30 and select application object BUPA
    b. Add a new entry with
        i. FGroupCrit: <APPL_NAME>000
        ii. Description: CustVEnd Blocked -> Display
        iii. Read function module: <Name of your function module>
4. Save this entry.

5 XO Framework – Extensible Objects

This chapter provides detailed instructions about the integration of your application into XO. If you need any details about XO, please see note 1623809 (contains a developer documentation as attachment).

XO customizing
For maintaining, the XO customizing transaction SE80 is used. The relevant business object type for all customer/vendor data is BUSINESS_PARTNER.

5.1 Scenario A – Integration of application owned tables

This chapter describes the XO-related steps that are necessary to integrate additional (new / application-owned) tables into transaction BP.

5.1.1 Create Memory Object (MO) class

For each database table that shall be integrated into transaction BP a memory object class is needed. The class has to inherit from class CVI_MO_CUSTOMER for customer data or CVI_MO_VENDOR for vendor data.
Naming convention: CVI_MO_<TABLE_NAME>
Example: CVI_MO_KNAS

Validation

In the new class, it is necessary to create separate validation methods (static methods with public visibility) for every table field that needs to be validated. You can also create methods to perform combined validations using multiple fields. However, it is strongly recommended to have a low granularity when creating the methods.

In addition, method VALIDATE_INTERN must be redefined. Within the redefinition first of all method VALIDATE_INTERN of the super-class has to be called, before all new validation methods within this redefinition are processed. You can compare for example the implementation in method CVI_MO_KNA1->VALIDATE_INTERN. Method VALIDATE_INTERN is called in an overall check on saving the business partner or when pressing button “check” in business partner maintenance.

In addition, the created validation methods must be called in the corresponding PAI function modules for dialog input validation.

Please also have look at SAP note 2293713 if want to integrate foreign key checks provided by classes CMD_EI_API_CHECK and VMD_EI_API_CHECK.

Class examples

CVI_MO_KNA1, CVI_MO_KNB1, CVI_MO_LFA1, CVI_MO_LFB1

5.1.2 Create Persistence Object (PO) Class

If your application table uses KUNNR or LIFNR as key field, no separate PO-class is needed. You only need to assign the existing persistence object CUSTOMER or VENDOR to your memory object in the next section.

If the key field of your table is not KUNNR or LIFNR, you have to create a separate persistence object class. The new class has to inherit from CVI_PO_CUSTOMER or CVI_PO_VENDOR. Now execute the following steps:

1. Redefine method IF_XO_PERSISTENCE_OBJECT~READ_DATA
   Copy the coding from the inherited method (superclass method)
   In the copied coding change the select statement at the end according to your table key.
2. Redefine method IF_XO_PERSISTENCE_OBJECT~SORT_DATA_BY_KEY
   Copy the coding from the inherited method (superclass method)
   Exchange the name of the assigned component ‘KUNNR’ / ‘LIFNR’ with the key field of your table at the end of the method.

5.1.3 XO Customizing

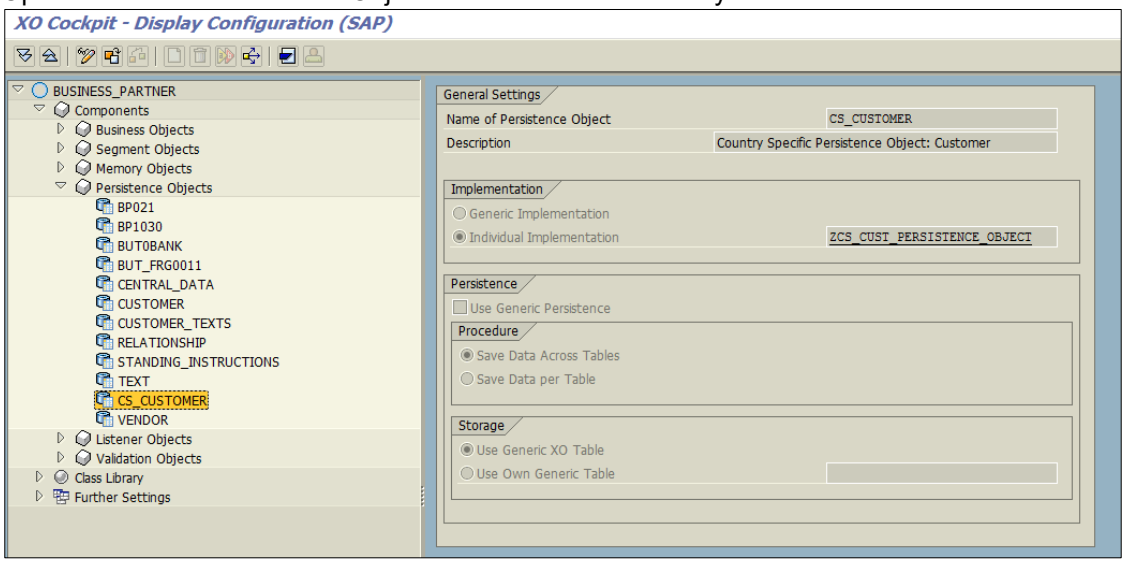
Execute the following steps to maintain the XO customizing for your objects:

- Start transaction XO80.
- Select Business Object Type BUSINESS_PARTNER from the initial popup.
- Switch to change mode.
- Make sure that you are in “SAP standard version” (ctrl+F3, 10th button in button line)

a) Persistence Object

This step is only needed when you have created an own persistence object in chapter 5.1.2. If you use one of the standard persistence objects CUSTOMER or VENDOR, you can skip this step.

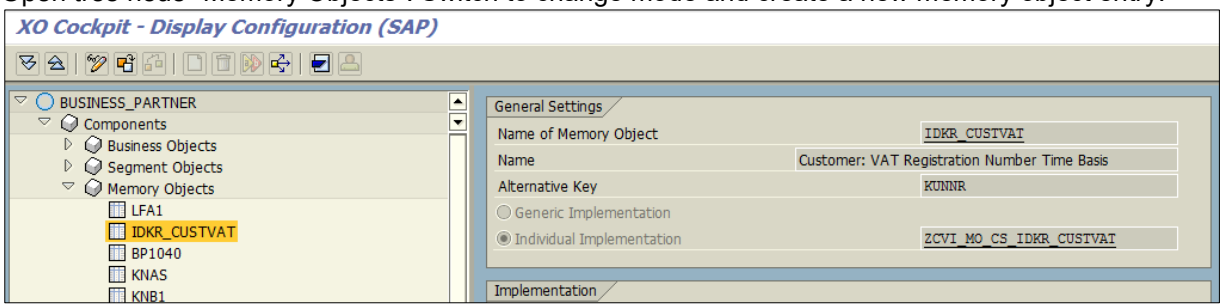
Open tree node “Persistence Objects” and create a new entry.



Name of Persistence Object = can be chosen arbitrarily
Description = can be chosen arbitrarily
Individual Implementation = Name of persistence object class created in step 5.1.2

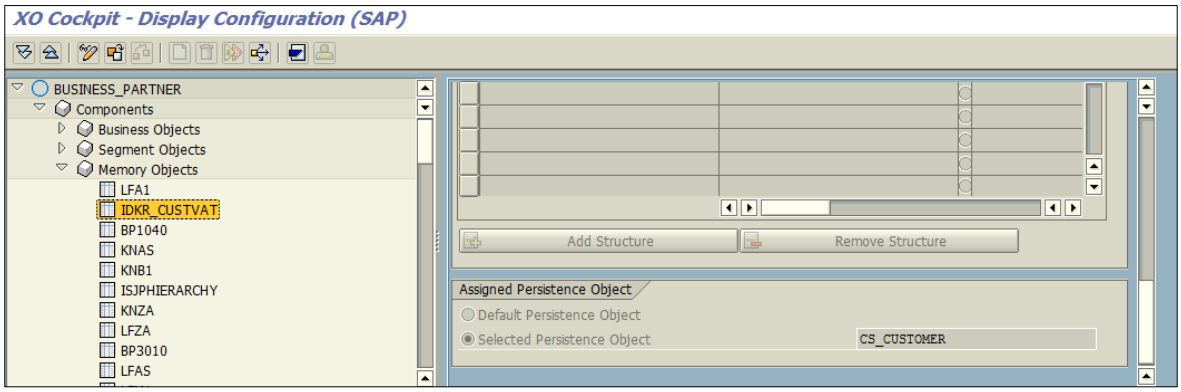
b) Memory Object

Open tree node “Memory Objects”. Switch to change mode and create a new memory object entry.



Name of Memory Object = Table name in SE11
Name = can be chosen arbitrarily
Alternative Key = KUNNR or LIFNR or other key field name of your table
Individual Implementation = Name of memory object class created in step 5.1.1.

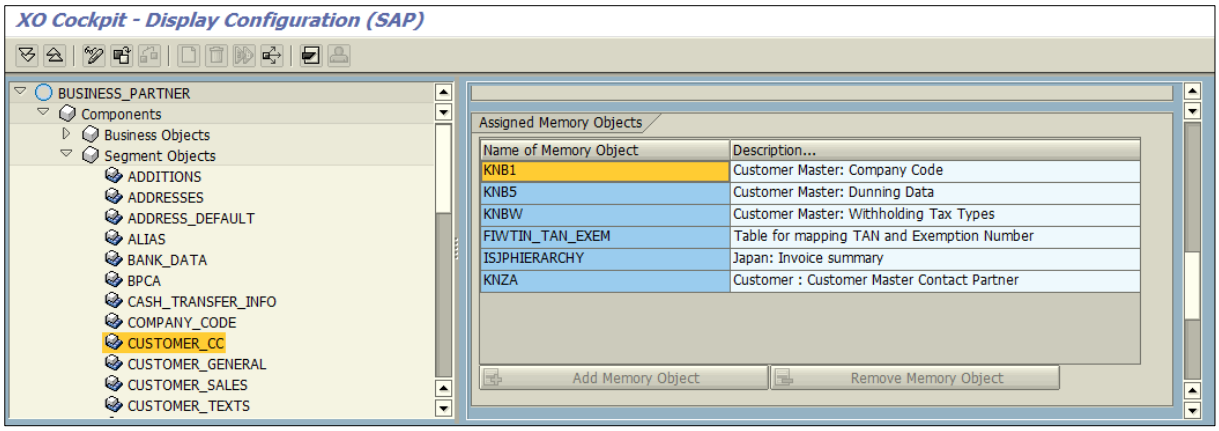
Scroll down to…



Selected Persistence Object = CUSTOMER or VENDOR when your table uses KUNNR or LIFNR
Selected Persistence Object = Name of your own persistence object from last step

c) Segment Object assignment

After creating the memory object this has to be assigned to the correct segment object.
Open tree node “Segment Objects” and assign your memory object to the segment object by drag and drop or button functionality.



The correct segment object depends on the customer/vendor segment to which your table belongs:

CUSTOMER_GENERAL	= General customer data	– e.g. KNA1, KNAT
CUSTOMER_CC	= Company Code data	– e.g. KNB1, KNB5
CUSTOMER_SALES	= Sales Area data	– e.g. KNVV, KNVI
VENDOR_GENERAL	= General vendor data	– e.g. LFA1, LFAT
VENDOR_CC	= Company Code data	– e.g. LFB1, LFB5
VENDOR_PURCHASE	= Purchasing Org. data	– e.g. LFM1, LFM2

5.2 Scenario B – Integration of core table appends

All appends of customer/vendor core tables like KNA1, KNB1, LFA1 or LFB1 are supported by the customer vendor integration (CVI) enhancement project. It is necessary to integrate all further append fields of the core tables using the standard classes like CVI_MO_KNA1_ENH or CVI_MO_LFB1_ENH. **Do not create new memory object classes inheriting from these core classes!**

Background information

The standard way to create new classes inheriting from the standard classes and replacing them in transaction XO80 will lead to technical problems. In case extensions have already used an older core class to inherit from, the new enhancement will not be visible in the extension and an exchange of the inherited class is technically not possible.

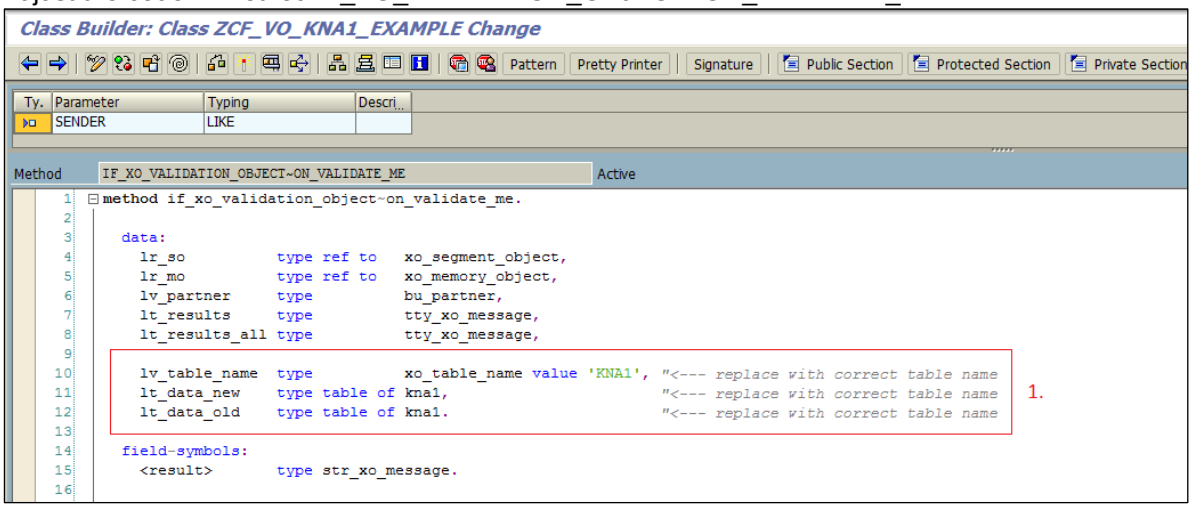
The purpose of this chapter is to describe an alternative approach how applications can enhance the business partner by customer/vendor core fields without the need of creating new memory object classes. The needed validations will be implemented in a validation class which is registered in XO customizing as validation object.

5.2.1 Create Validation Object (VO) classes

The maintenance of new fields from the customer/vendor core tables is already possible after integrating the fields into BDT and making sure that the changes are transferred to XO-memory in the PAI function modules. The only missing part is the validation of the new fields. In order to implement the validation the following steps need to be processed. As an example we refer to table KNA1 in the below screen shots. If you integrate appends of another core table unequal KNA1 replace objects, parameters and variable names accordingly.

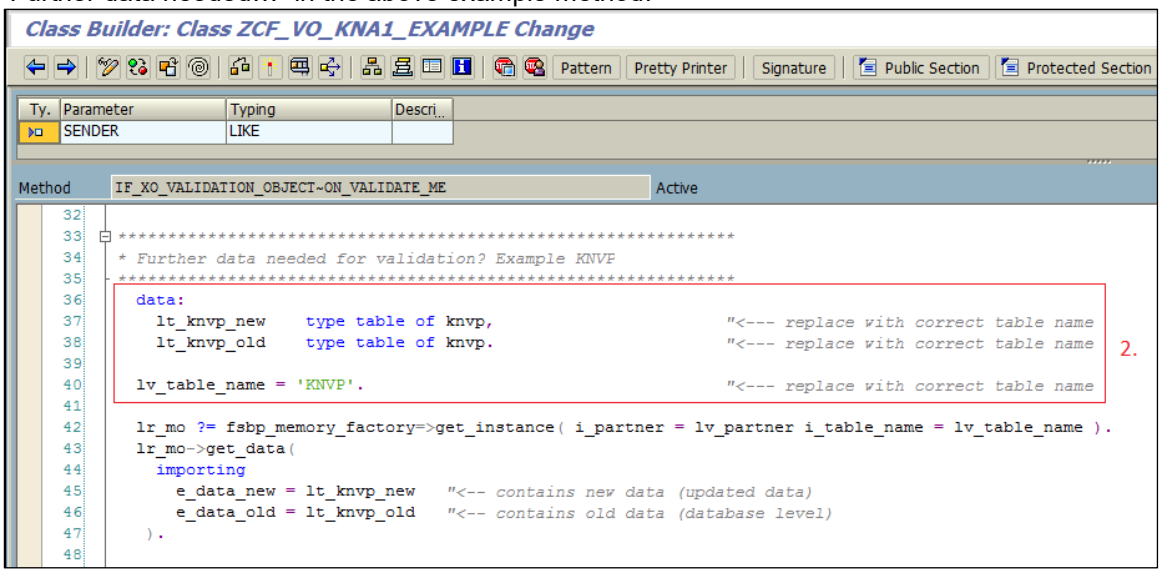
1. Create an own validation object class as copy of template class ZCF_VO_KNA1_EXAMPLE in system ER9/001.

2. Adjust the code in method IF_XO_VALIDATION_OBJECT~ON_VALIDATE_ME



1. Use the correct table type of the customer/vendor table for these variables

3. If you need further data from other memory objects (database tables) check the section "Further data needed…" in the above example method:



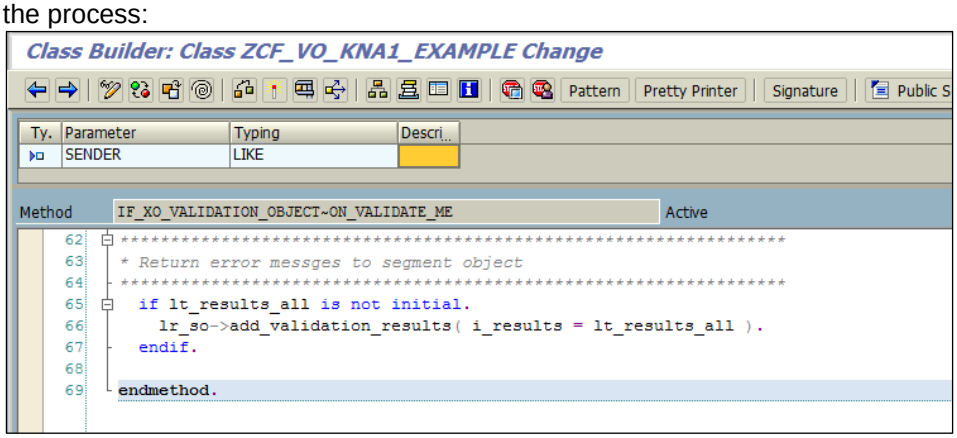
2. Use the correct table type of the customer/vendor table for these variables

4. Rename and adjust the existing (template) field validation methods VALIDATE_KNA1_FIELD* or create new ones (static and public)

5. Delete superfluous template methods

6. Implement a call to all your validation methods in the interface method IF_XO_VALIDATION_OBJECT~ON_VALIDATE_ME.

7. Implement a call to method LR_SO->ADD_VALIDATION_RESULTS in the interface method IF_XO_VALIDATION_OBJECT~ON_VALIDATE_ME to return validation error messages to the process:



8. The created static validation methods (like example method VALIDATE_KNA1_FIELD) should be called in the corresponding PAI modules as well. With this, the validation of the user input is directly processed after entering (when pressing enter).

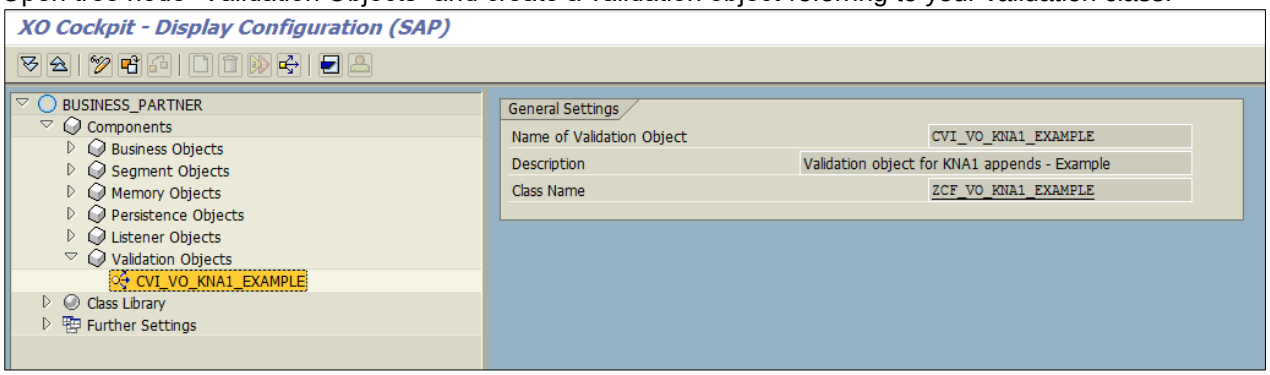
Attention
Make sure that your implementation does not contain any BDT specific logic, like calling methods from class CVI_BDT_ADAPTER or calling function module BUS_MESSAGE_STORE to process an error message! Validation objects are also processed in API processes, so that BDT-specific coding could lead to a short-dump during runtime.

Please also have look at SAP note 2293713 if want to integrate foreign key checks provided by classes CMD_EI_API_CHECK and VMD_EI_API_CHECK.

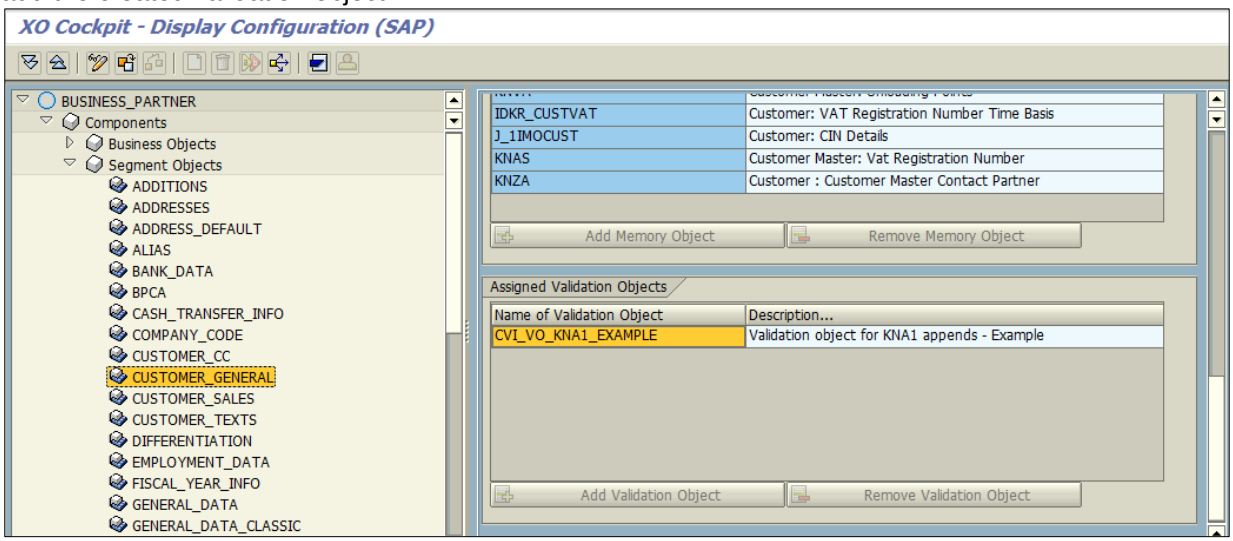
5.2.2 XO Customizing

In the XO customizing (transaction XO80) open business object type BUSINESS_PARTNER.

Open tree node “Validation Objects” and create a validation object referring to your validation class.



Open the corresponding segment object (for KNA1 this is for example CUSTOMER_GENERAL) and add the created validation object.



The correct segment object depends on the customer/vendor segment to which your memory object is assigned:

CUSTOMER_GENERAL	= General customer data	– e.g. KNA1, KNAT
CUSTOMER_CC	= Company Code data	– e.g. KNB1, KNB5
CUSTOMER_SALES	= Sales Area data	– e.g. KNVV, KNVI
VENDOR_GENERAL	= General vendor data	– e.g. LFA1, LFAT
VENDOR_CC	= Company Code data	– e.g. LFB1, LFB5
VENDOR_PURCHASE	= Purchasing Org. data	– e.g. LFM1, LFM2

6 CVI – Saving data to database

While maintaining business partner data in transaction BP the data are only retrieved and transferred back to the XO memory and validations are executed. Saving of changed customer/vendor data to database is collectively triggered on commit after the button “Save” has been pressed. The reason for this is that the customer/vendor data need to be saved together (either save all data or no data). As the central data need to be mapped from the corresponding business partner structure before they are available in the customer/vendor structure, the data can only be saved to database after the mapping has been performed. Mapping and saving to database is triggered by the business partner outbound processing which starts the synchronization

In case of questions concerning this chapter, contact Alain Bacchi.

The CVI offers two enhancement spots for saving data that is not part of the CVI core enhancement: CUSTOMER_EXTENSION and VENDOR_EXTENSION. These enhancement spots contain several BAdI definitions, which provide initialize, validate and save functionality.

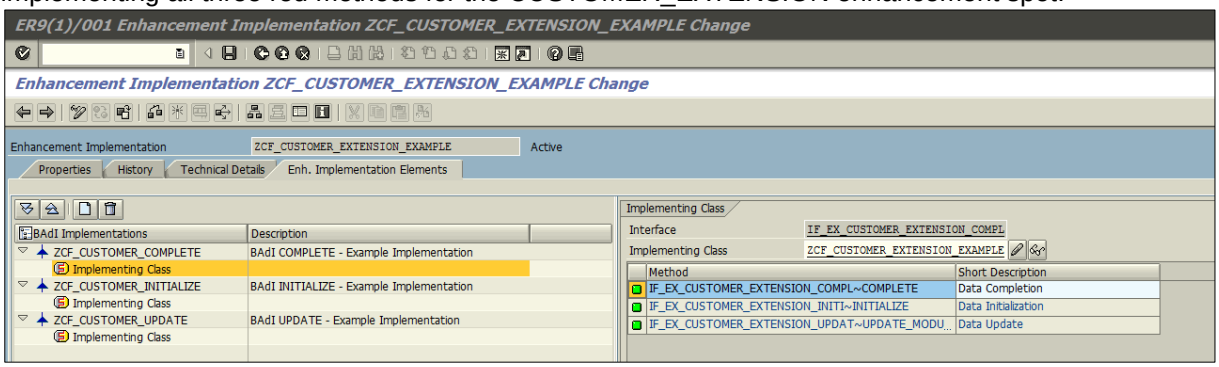
	CUSTOMER_EXTENSION	VENDOR_EXTENSION
BAdI	CUSTOMER_EXTENSION_AUTH_CHECK	VENDOR_EXTENSION_AUTH_CHECK
	CUSTOMER_EXTENSION_CHECK	VENDOR_EXTENSION_CHECK
	CUSTOMER_EXTENSION_COMPLETE	VENDOR_EXTENSION_COMPLETE
	CUSTOMER_EXTENSION_INITIALIZE	VENDOR_EXTENSION_INITIALIZE
	CUSTOMER_EXTENSION_OUTBOUND	VENDOR_EXTENSION_OUTBOUND
	CUSTOMER_EXTENSION_UPDATE	VENDOR_EXTENSION_UPDATE

How-to recommendation

It is necessary to create one enhancement implementation only for each enhancement spot. Each of these two enhancement implementations should use only one single implementing class containing implementations for all BADIs that are marked in red in the above table. So in case you integrate both customer and vendor data in your application you will have only two enhancement implementations and only two implementing classes after finishing the implementation.
Both classes must contain at least one public static attribute to store KUNNR / LIFNR.

Example

In System ER9/001, we have created enhancement implementation ZCF_CUSTOMER_EXTENSION_EXAMPLE using class ZCF_CUSTOMER_EXTENSION_EXAMPLE implementing all three red methods for the CUSTOMER_EXTENSION enhancement spot:



6.1 Scenario A – Integration of application owned tables

In this scenario, the data of the newly integrated tables need to be retrieved from the memory object and then saved to database. To save the data into the corresponding database table BAdIs CUSTOMER_EXTENSION_UPDATE or VENDOR_EXTENSION_UPDATE are needed.

In addition it is necessary to implement BAdIs CUSTOMER_EXTENSION_COMPLETE respectively VENDOR_EXTENSION_COMPLETE. The reason for this is that the UPDATE BAdIs are called without KUNNR / LIFNR of the currently processed customer/vendor. For supporting mass data processing, it is important to make sure that the UPDATE BAdIs only save the data of the currently processed customer/vendor.

To have the current customer/vendor number available in the update BAdIs implement the following logic in your implementation of CUSTOMER_EXTENSION_COMPLETE respectively VENDOR_EXTENSION_COMPLETE.

1. Read customer/vendor number from the transferred structure CS_CUSTOMER / CS_VENDOR
2. Store customer/vendor number in the corresponding attribute of your implementing class
3. Read customer/vendor number in your implementation of BAdI CUSTOMER_EXTENSION_UPDATE / VENDOR_EXTENSION_UPDATE from the corresponding attribute of your implementing class.

In the implementation of method IF_EX_CUSTOMER_EXTENSION_UPDATE~UPDATE_MODULES execute the following steps:

1. Make sure that your application is currently active. If your application is not active do not process the coding within your BAdI implementation
2. Retrieve the data from the corresponding memory object
3. Ensure that you process only data of the current customer/vendor
4. Call your database update function module in update task.
5. In addition, the creation of change documents has to be triggered here.

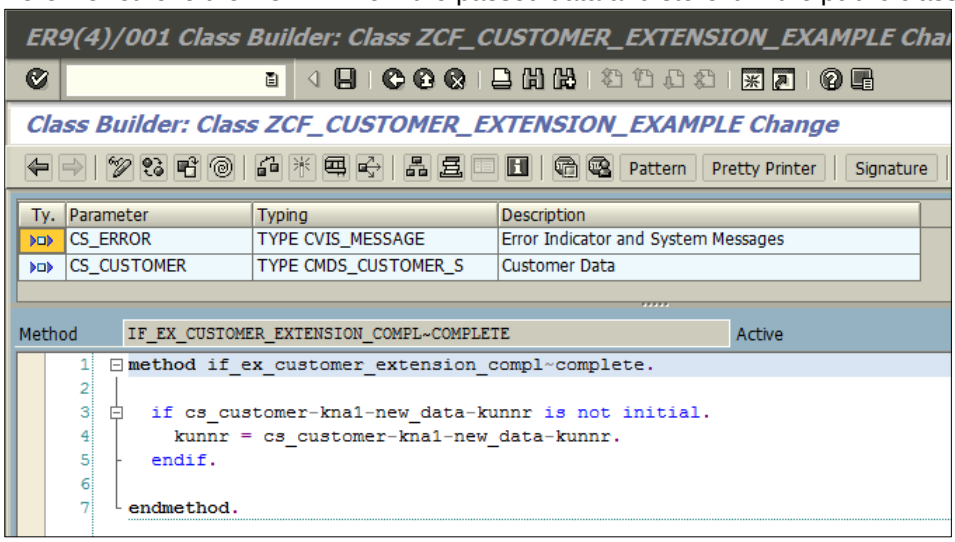
Attention: **Exit your BAdI implementation, when your application is not active!**

Database updates must be performed in update task!

Example Implementation

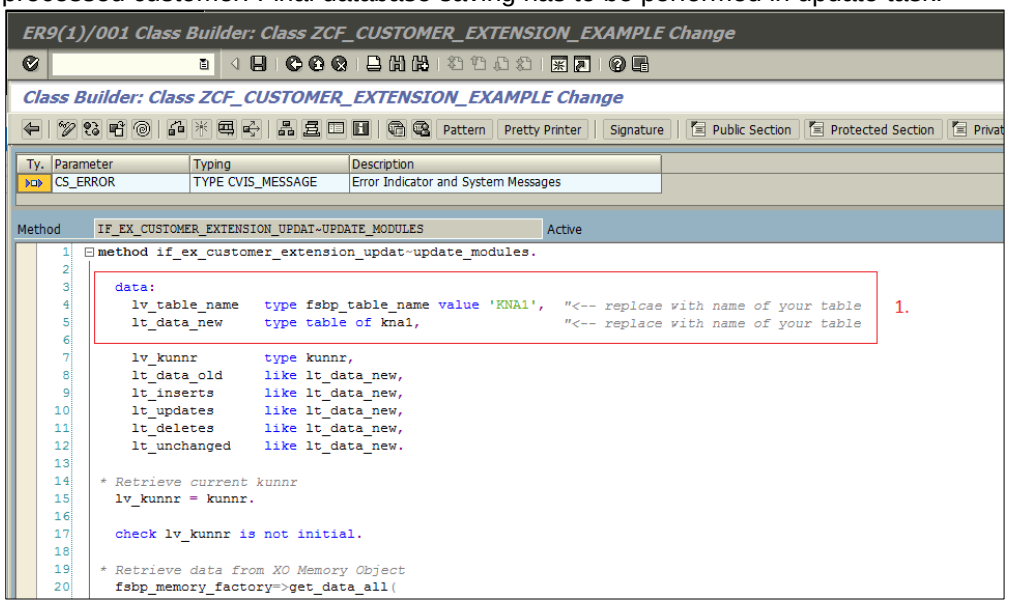
In ER9/001, an implementation for BAdIs CUSTOMER_EXTENSION_COMPLETE and CUSTOMER_EXTENSION_UPDATE has been created using the same implementing class.
The implementing class is ZCF_CUSTOMER_EXTENSION_EXAMPLE and contains example implementations for methods IF_EX_CUSTOMER_EXTENSION_COMPL~COMPLETE and IF_EX_CUSTOMER_EXTENSION_UPDAT~UPDATE_MODULES.
The class also contains a public static attribute to store KUNNR.

BAdI implementation CUSTOMER_EXTENSION_COMPLETE
Here we retrieve the KUNNR from the passed data and store it in the public class attribute.



Remark: This BAdI will be called multiple times for every changed database table in case of simultaneous changes on different data segments (KNA1, KNB1, KNVV …). Only in the first call, you can rely on a filled KNA1 / LFA1 segment containing the needed KUNNR / LIFNR. Therefore, the check that KUNNR / LIFNR is not initial is needed here.

BAdI implementation CUSTOMER_EXTENSION_UPDATE
Here we retrieve the data from XO and make sure that we only save the data of the currently processed customer. Final database saving has to be performed in update task.



1. Use the correct table type for these variables

6.2 Scenario B – Integration of core table appends

For scenario B, you have to check if the fields of your core table append are included in the structure CVIS_EI_EXTERN.

6.2.1 Append fields do exist in CVIS_EI_EXTERN

Nothing to do here. Since all your append fields exist in CVIS_EI_EXTERN these will be processed by the CVI standard functionality.

6.2.2 Append fields do not exist in CVIS_EI_EXTERN

Before continuing here contact Alain Bacchi (I023145) from LO-MD-BP-SYN to evaluate if your fields can be added to CVIS_EI_EXTERN. If this is possible, the steps of this section are not needed!

Only continue with this chapter, if your fields cannot be added to CVIS_EI_EXTERN!

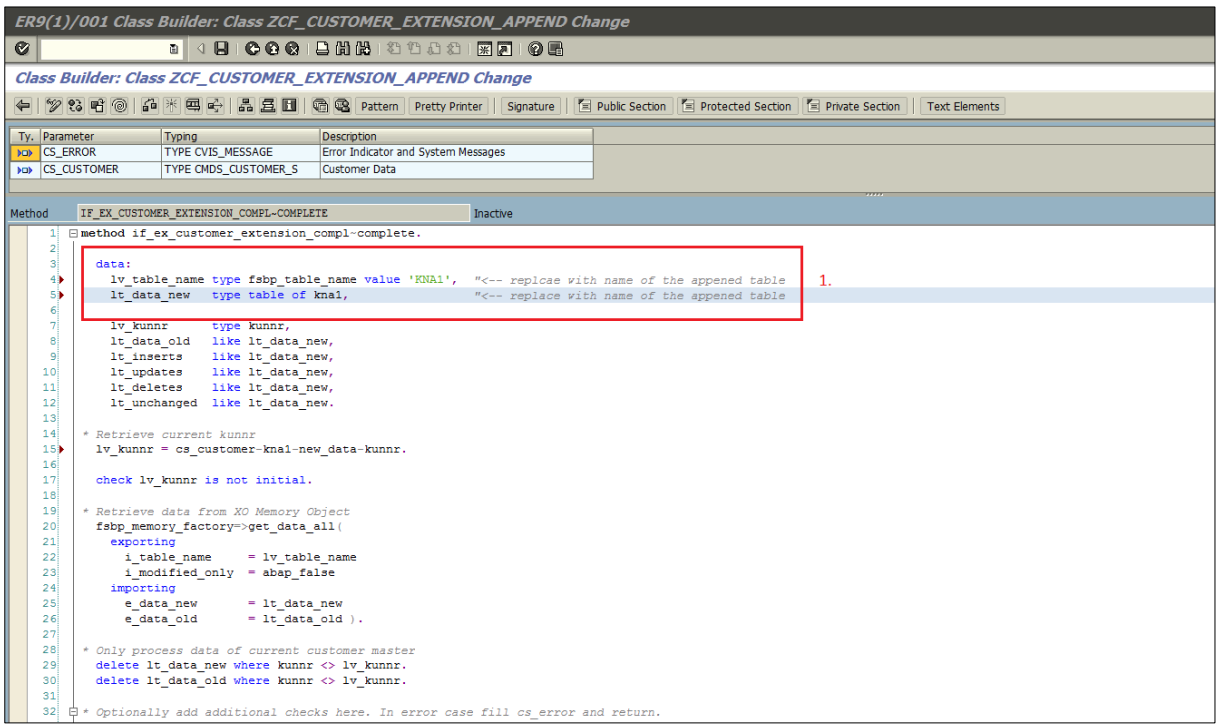
The data from your memory object must be extracted and added to the corresponding core table.
Therefore, the BAdI CUSTOMER_EXTENSION_COMPLETE or VENDOR_EXTENSION_COMPLETE has to be used. In the implementation the corresponding field values are retrieved from the memory object and have to be added/updated in the corresponding structure of the changing parameter CS_CUSTOMER respectively CS_VENDOR.

In the implementation of the COMPLETE BAdI, you also have to make sure that you only process the data of the current customer / vendor.
Additionally writing change documents has to be implemented here.

An implementation of the UPDATE BAdI is not needed in this scenario.

Example Implementation

In system ER9/001, we have created an example implementation for the COMPLETE BAdI. In class ZCF_CUSTOMER_EXTENSION_APPEND->IF_EX_CUSTOMER_EXTENSION_COMPL~COMPLETE data is retrieved from the XO memory and is filtered for the currently processed customer.



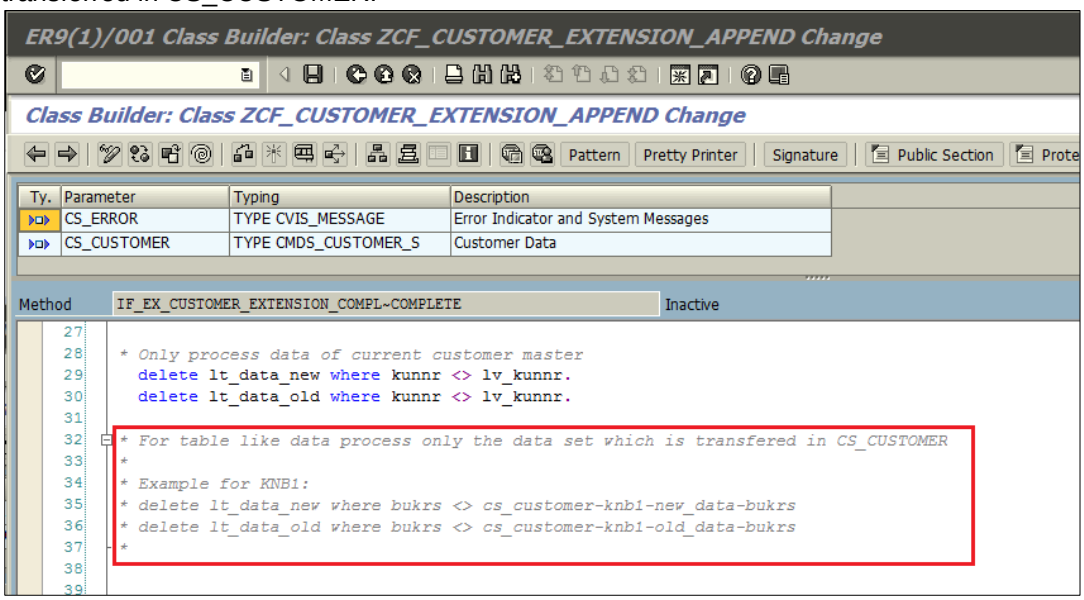
1. Use the correct table type for these variables

Attention: **Exit your BAdI implementation, when your application is not active!**

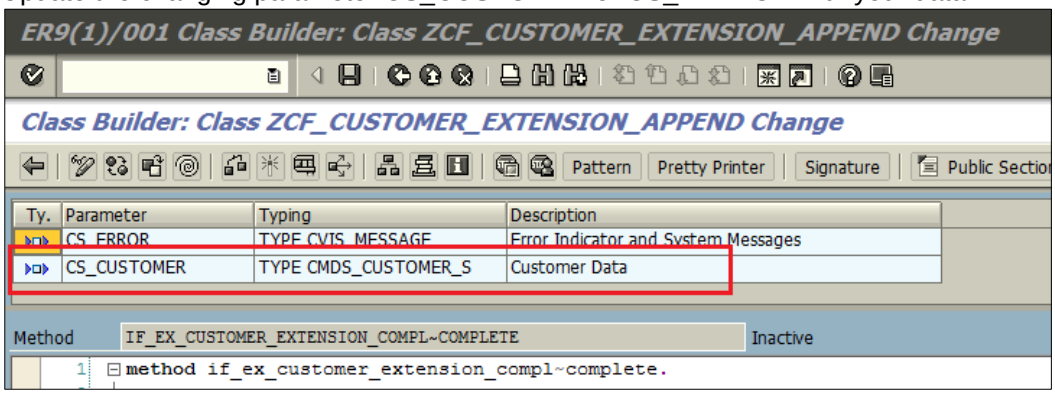
Do not save the data of core tables to the database in your implementation!

Since XO always returns all memory entries of a specified table (i.e. if more than one business partner respectively customer/vendor has been touched the data of all changed customers/vendors are returned) you have to delete all entries from the return table that do not belong to the currently processed customer/vendor. This logic is also true for other key fields like company code or sales area. In the example of table KNB1 this means all other company code data have to be deleted from the retrieved memory tables lt_data_new and lt_data_old.

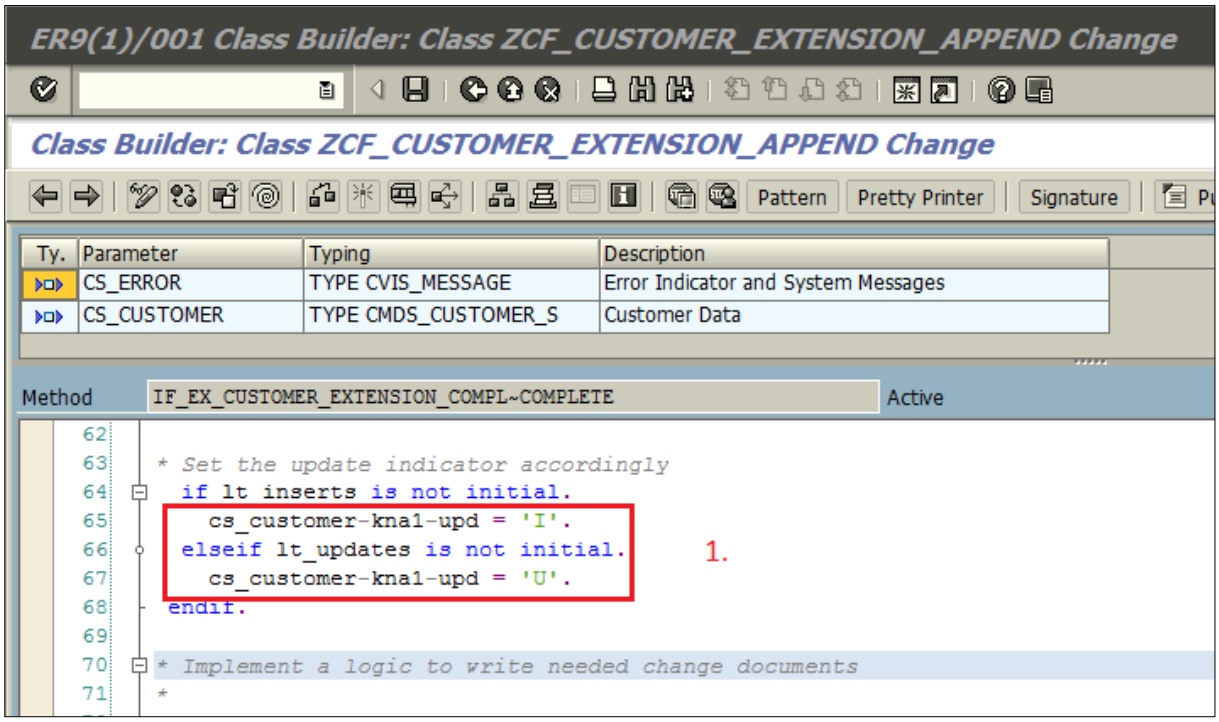
For table like data (e.g. KNB1) you have to make sure that you only process the data set which is transferred in CS_CUSTOMER:



Update the changing parameter CS_CUSTOMER or CS_VENDOR with your data.



Set the update indicator of the used structure as well:



1. Select the correct structure of CS_CUSTOMER (KNA1, KNB1,…)

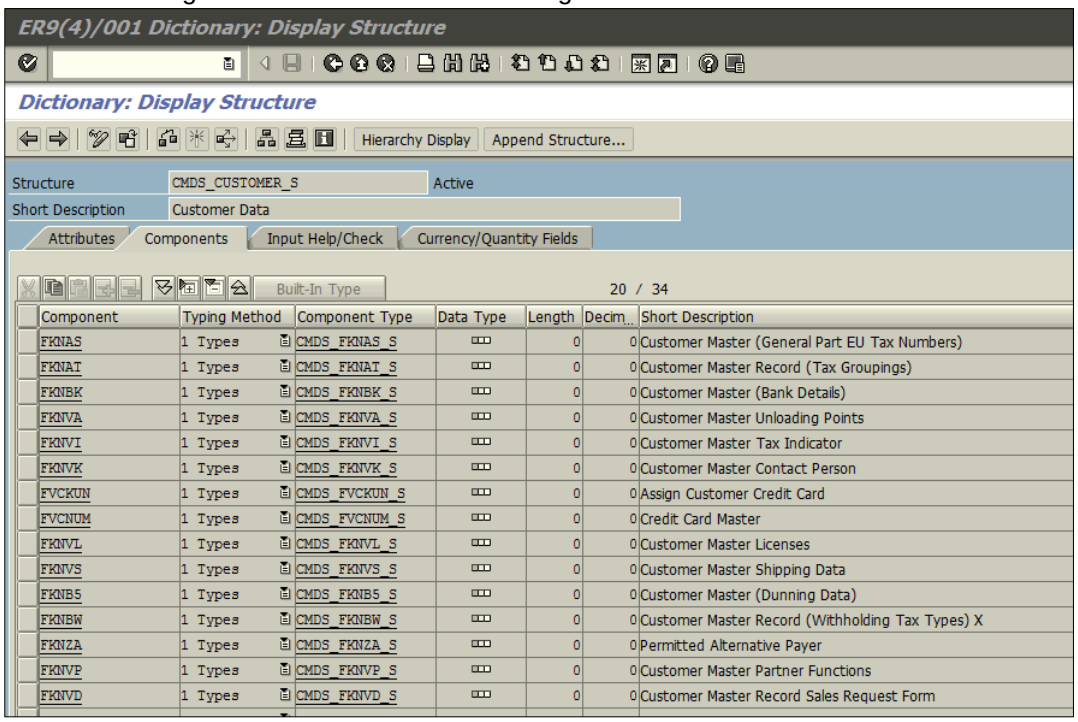
Attention:

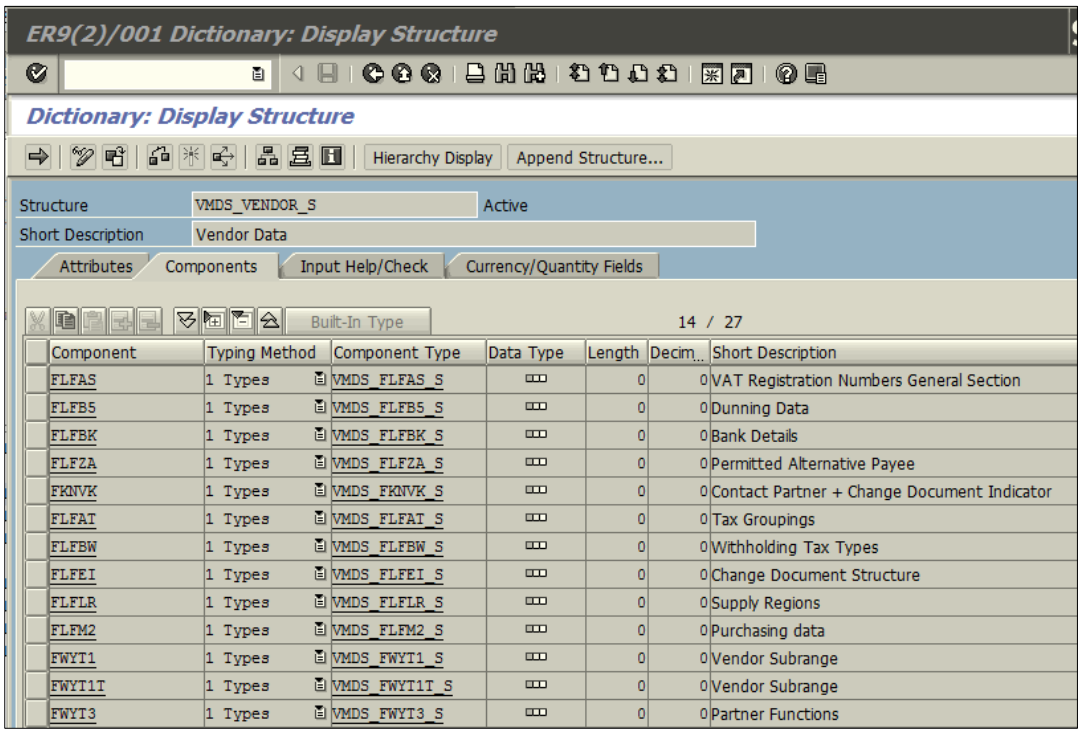
Exit your BAdI implementation, when your application is not active!

Do not set field UPD when your application is not used and/or your data was not changed!

Special cases

For the following table like datasets the following structures have to be considered:





These F-components contain tables for NEW_DATA and OLD_DATA. For handling appends on these tables you need to adjust your logic to transfer the correct entries retrieved from XO to CS_CUSTOMER / CS_VENDOR. Also make sure to set the update indicator for every entry in the tables.

6.2.3 Foreign key checks of appended fields

The fact that a field is appended to a core table does not imply that this field is part of CVI core. If it is not part of CVI core, no foreign key check is performed automatically.

If in such a case a foreign key check is needed, you have to implement the corresponding BAdI:

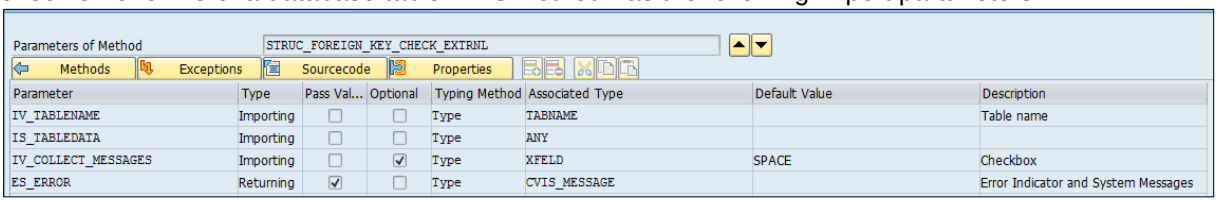
	CVI_CUSTOM_MAPPER_ENH	
BAdI Method	MAP_CUSTOMER_FOREIGN_KEY_TABLE	Map customer relevant foreign key check data
	MAP_VENDOR_FOREIGN_KEY_TABLE	Map vendor relevant foreign key check data

Details on Foreign key check enablement

Requirement
Industry specific CVI enhancements need to perform a foreign key check at the PAI level for the fields of their application. There is no public method available which executes the foreign key check.

Solution
For processing generic foreign key checks in each of the two classes VMD_EI_API_CHECK and CMD_EI_API_CHECK for vendor master and customer master the following two methods have been introduced:
STRUC_FOREIGN_KEY_CHECK_EXTRNL and
FOREIGN_KEY_CHECK_EXTERNAL.

1. Method STRUC_FOREIGN_KEY_CHECK_EXTRNL: This method is used to perform foreign key check on one line of a database table. This method has the following import parameters:

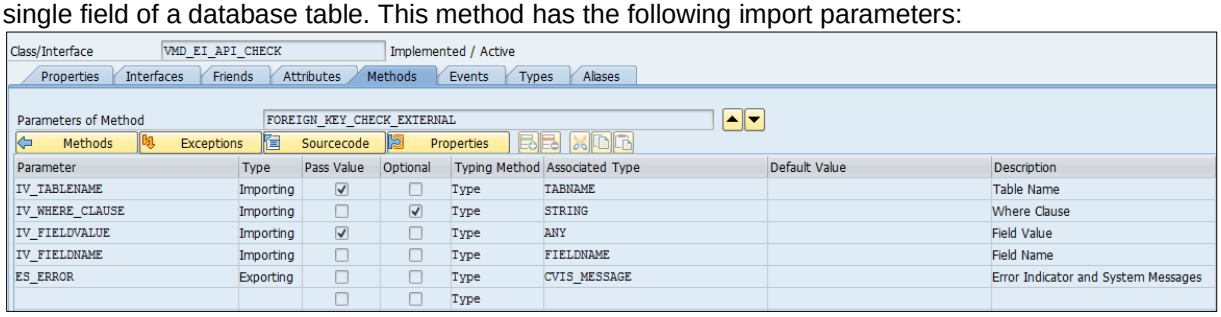


Parameter	Type	Pass Val...	Optional	Typing Method	Associated Type	Default Value	Description
IV_TABLENAME	Importing	☐	☐	Type	TABNAME		Table name
IS_TABLEDATA	Importing	☐	☐	Type	ANY		
IV_COLLECT_MESSAGES	Importing	☐	☑	Type	XFELD	SPACE	Checkbox
ES_ERROR	Returning	☑	☐	Type	CVIS_MESSAGE		Error Indicator and System Messages

IV_TABLENAME: Table name as string.
IS_TABLEDATA: One line of table IV_TABLENAME.

This method calls the FOREIGN_KEY_CHECK_EXTERNAL method internally transferring the where clause (in parameter IV_WHERE_CLAUSE).

2. FOREIGN_KEY_CHECK_EXTERNAL: This method is used to perform foreign key check on a single field of a database table. This method has the following import parameters:



Parameter	Type	Pass Value	Optional	Typing Method	Associated Type	Default Value	Description
IV_TABLENAME	Importing	☑	☐	Type	TABNAME		Table Name
IV_WHERE_CLAUSE	Importing	☐	☑	Type	STRING		Where Clause
IV_FIELDVALUE	Importing	☑	☐	Type	ANY		Field Value
IV_FIELDNAME	Importing	☐	☐	Type	FIELDNAME		Field Name
ES_ERROR	Exporting	☐	☐	Type	CVIS_MESSAGE		Error Indicator and System Messages
		☐	☐	Type			

IV_TABLENAME: Table name as string.
IV_WHERE_CLAUSE: Optional parameter that must not be provided when called from outside the class. This parameter is only provided when the method is called from method STRUC_FOREIGN_KEY_CHECK_EXTRNL.
IV_FIELDNAME Name of the field in table <IV_TABLENAME> for which the foreign key check has to be performed.
IV_FIELDVALUE: Value of the field in table <IV_TABLENAME> for which the foreign key check has to be performed.

7 Mass Data Processing – Central API

Consuming applications can maintain their industry specific fields in mass processing by calling method CL_MD_ BP_MAINTAIN=>MAINTAIN.

Before passing data to CL_MD_ BP_MAINTAIN=>MAINTAIN it is highly recommended to perform the pre-requisite field checks with method CL_MD_ BP_MAINTAIN=>VALIDATE_SINGLE. To get all the relevant validation checks the BP grouping in the BP complex structure and the account grouping in the customer/vendor complex structure have to be transferred along with all the other key data.

Again, the same two scenarios as in dialog processing need to be distinguished:

7.1 Scenario A – Integration of application owned tables

The following steps need to be executed:

- Call method CL_MD_BP_MAINTAIN=>MAINTAIN_NON_CORE_VALUE with your table name and your data. This method converts the transferred data into a data type (RAWSTRING) which can be transferred to method CL_MD_BP_MAINTAIN=> MAINTAIN. The converted data are returned in parameter ET_TABLE_EXT.

- The mass processing is started by calling method CL_MD_BP_MAINTAIN=> MAINTAIN. Before calling this method the converted data (ET_TABLE_EXT) need to be integrated into the complex structure that is transferred to CL_MD_BP_MAINTAIN=>MAINTAIN (sub-structure CVIS_EI_EXTERN-EXT_APPL_DATA)

- The transferred data of application owned tables will be automatically saved in the XO memory and can be retrieved in the update BADI later on as described in section 6.1.

Please find the below example code:

```
cl_md_bp_maintain=>maintain_non_core_value(
  exporting
    i_tabname      = 'J_1IMOCUST'    " Table Name
    i_table        = lt_mocust       " Table Data
  importing
    e_messages     = lt_messages     "Error Messages
    et_table_ext   = lt_ext_data      "Converted DATA
).

read table lt_bussiness_partner assigning <lt_bussiness_partner> index 1.
<lt_bussiness_partner>-ext_appl_data = Lt_ext_data.


cl_md_bp_maintain=>maintain(
  exporting i_data = lt_bussiness_partner
  importing e_return = lt_api_result
).
```

7.2 Scenario B – Integration of core table appends

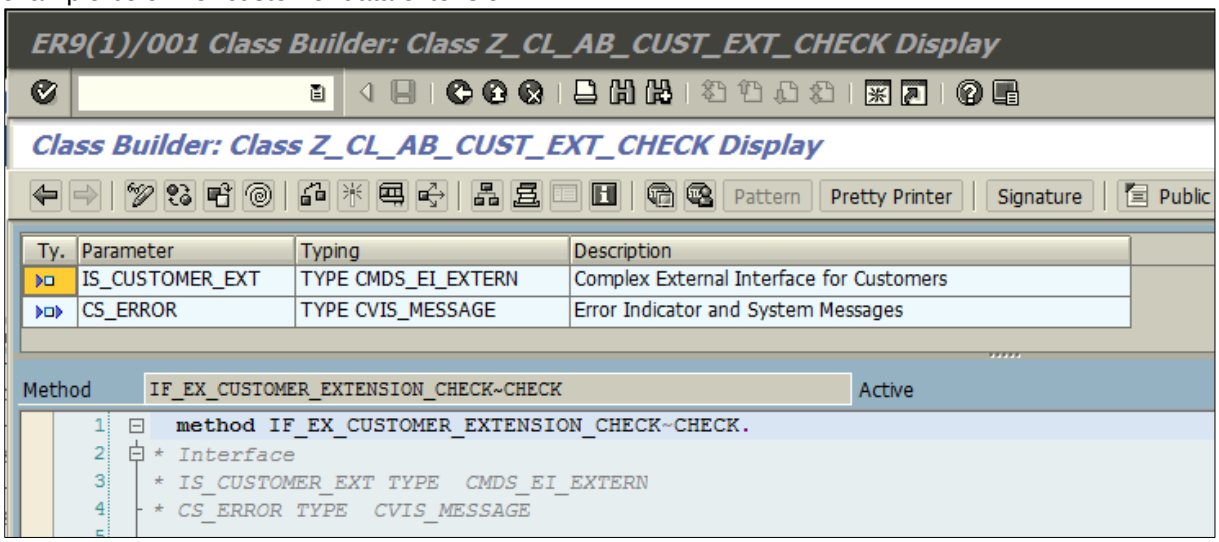
Contact Alain Bacchi (I023145) from LO-MD-BP-SYN to evaluate if your fields can be added to the complex structure CVIS_EI_EXTERN. If your fields are available in the complex structure you can transfer them directly to method CL_MD_ BP_MAINTAIN=>MAINTAIN to update your fields.

7.3 Data validation in mass processing

For both scenarios A and B, you have to create an implementation of the corresponding Check-BAdI in an enhancement implementation that belongs to your area. In the following table you find the available BADIs that can be implemented for the two enhancement spots CUSTOMER_EXTENSION and VENDOR_EXTENSION.

	CUSTOMER_EXTENSION	VENDOR_EXTENSION
BAdI	CUSTOMER_EXTENSION_AUTH_CHECK	VENDOR_EXTENSION_AUTH_CHECK
	CUSTOMER_EXTENSION_CHECK	**VENDOR_EXTENSION_CHECK**
	CUSTOMER_EXTENSION_COMPLETE	VENDOR_EXTENSION_COMPLETE
	CUSTOMER_EXTENSION_INITIALIZE	VENDOR_EXTENSION_INITIALIZE
	CUSTOMER_EXTENSION_OUTBOUND	VENDOR_EXTENSION_OUTBOUND
	CUSTOMER_EXTENSION_UPDATE	VENDOR_EXTENSION_UPDATE

If you want to validate your data, you have to implement your validation logic in the check BAdI. See example below for customer data extension:



Call your validation methods within this BAdI implementation to validate the extended data that is passed in parameter IS_CUSTOMER_EXT. In case of an error, fill the returning parameter CS_ERROR accordingly with the error indicator and the error message(s) to be returned.

Please also have look at SAP note 2293713 if want to integrate foreign key checks provided by classes CMD_EI_API_CHECK and VMD_EI_API_CHECK.

7.4 Committing changes and error handling

In order to execute the data changes and to trigger the database update, function module BAPI_TRANSACTION_COMMIT must be called after the call of method CL_MD_BP_MAINTAIN=> MAINTAIN. A simple COMMIT WORK is not sufficient, because it does not refresh the buffers. Since the CVI is running on commit, all errors raised during synchronization will create an entry in the PPO (Post-Processing-Office), in case PPO is set to active in customizing, or lead to a short dump in case PPO is set to inactive. In case of an error with active PPO, the BP will be saved, whereas the corresponding customer/vendor will not be saved (which means a data inconsistency between the objects until the error has been solved and the changes have been reloaded). In case of a short dump due to inactive PPO, all changes will be rolled back and no changes will be saved in any of the objects.

In case PPO is active, you can call method CL_MD_BP_MAINTAIN-> GET_PPO_MESSAGES after function module BAPI_TRANSACTION_COMMIT has been executed. This method will return all logged PPO entries of the current LUW in parameter E_RETURN.

You can use transaction MDS_PPO2 to display the created PPO entries.

7.5 Example report and support

In internal system ER9 you can find the report ZCENTRAL_API_DEMO, which provides an example of a mass call transferring also application owned tables.

If you need support with the Central API or mass processing in general please contact DEV team from LO-MD-BP-SYN.

8 Debugging

Important places to have a look in debugger, when problems occur:

Class CVI_BDT_ADAPTER_INTERN (or corresponding sub-classes): set breakpoints in methods "event_*", for example:

- EVENT_DSAVC: xo event finalize is processed to replace the preliminary number ##1 by the final number
- EVENT_DSAVE: xo-events ON_COMMIT_START and ON_COMMIT_END are triggered (responsible for saving the data to database)
  -> Exception: for CVI datasets the saving to database is done in CVI BAdIs
- EVENT_DLVE2: xo event cleanup is triggered to clear the complete memory and all object instances (in dialog in transaction BP the XO_EVENT is registered on commit level 99, so that it is processed after the synchronization has been finished – otherwise the memory-object methods ON_DELIVER_CUSTOMER_DATA data would not return any data when called during mapping)

XO_BUSINESS_FACTORY=>GET_INSTANCE: here the XO-instance for the current business partner is created (or returned)

Memory object classes (like CVI_MO_KNA1_ENH):

- in method GET_DATA(_NEW) the data are read from memory, on first call the data are selected from database by communication with persistence layer
- in method SET_DATA(_NEW) changed data are transferred to the memory for later saving
- in methods VALIDATE_* the validations for the different fields of the corresponding database table are processed
- in method VALIDATE_INTERN all VALIDATE_* methods are processed for an overall check of all fields of the current table

Mapping (is processed on commit!) (Belongs to component AP-MD-BP-SYN – central BP team – manager: Gurumoorthy H)

- class CVI_MAPPER triggers the mapping (Customer: Method MAP_BPS_TO_CUSTOMERS, Vendor: MAP_BPS_TO_VENDORS)
- actual mapping of different datasets is done in class CVI_FM_BP_CUSTOMER
- Interesting method in class CVI_FM_BP_CUSTOMER is GET_ENHANCEMENT_DATA, here event DELIVER_CUSTOMER_DATA is raised, on which all memory object classes of CVI-standard have registered themselves; as a consequence in all memory classes the method ON_DELIVER_CUSTOMER_DATA is triggered, by this the data from business partner xo memory are transferred to the complex customer structure
- The logic for enhancement data is – however – different and can be derived from the cookbook (POC) of our Globalization Services colleagues (not final yet, will be sent to you in the course of next week – alternatively contact Alain Bacchi)

Logic for vendor is analogous in class CVI_FM_BP_VENDOR

Save data to database

- method CMD_EI_API= >MAINTAIN (belongs to component LO-MD-BP-SYN, contact: Alain Bacchi)
- Standard tables are saved here automatically
- Implemented BADIs within the CVI

