ALDI
ABAP Development Policy

Table of Content

1 Document Control

1.1 Processing Status

Version	Status	Name Author	Type of processing	Date
1.40	Initial version	ALDI	Initial ABAP development guidelines	04/07/2019
1.50	Submitted for review	Anugrah Prakash	SAP ABAP Development guidelines submitted for review	05/01/2021
2.0	Approved	DQA@aldi-sued.com	Approved revised ABAP development guidelines	07/01/2021
2.1	Submitted for review	Darpan Mehta	SAP ABAP Development guidelines submitted for review	09/11/2021
3.0	Approved	DQA@aldi-sued.com	Approved revised ABAP development guidelines	09/11/2021
3.1	Submitted for review	Darpan Mehta	SAP ABAP Development guidelines submitted for review	21/03/2022
4.0	Approved	DQA@aldi-sued.com	Approved revised ABAP development guidelines	21/03/2022

1.2 Approval Status

Version	Status	Approved by (Name)	Remarks/Findings	Date
4.0	Approved	DQA@aldi-sued.com	Approved after alignments	21/03/2022

2 Abstract

2.1 Introduction

The high level of flexibility and extensibility of SAP software involves both benefits and disadvantages. On the one hand, the software can be adjusted to meet customer-specific requirements to an optimum degree, thus significantly increasing the additional value generated using the software. At the same time, extensibility entails the risk of customer-specific developments that are complex, difficult to maintain and prone to errors. The purpose of this document is to define standardised guidelines for designing ALDI-specific requirements in order to develop these in an easy-to-maintain, efficient and secure manner. Standard rules ensure production of development objects which are well structured, robust, consistent, maintainable and readable by third person with small effort.

Inadequate quality in adjustments to the SAP standard entails risks for ALDI. The guidelines presented in this document have been created with the objective of ensuring appropriate quality of adjustments to SAP and thus preventing potential risks.

	Risks entailed by inadequate quality	Objective of the policy
Security	Sabotage, industrial espionage, unwanted press coverage caused by data leaks, downtime of live systems	Preventing unauthorised users from accessing and/or editing critical data
Compliance	Failed industrial audits, breaches of compliance requirements or legal provisions (e.g., data protection)	Ensuring and being able to prove that developed software fulfils relevant compliance standards and legal provisions
Performance	Decline in user acceptance and/or additional costs for faster hardware required to compensate for deficient software	Ensuring that the existing hardware can be used in an optimised way, investment protection and increase in employee satisfaction as applications can be used more efficiently
Robustness	Decline in user acceptance and rising operating costs due to a lack of user productivity as well as error analyses and maintenance work by technicians	Ensuring continuous operation of business applications Preventing a lack of productivity due to system downtimes
Maintainability	High maintenance costs and increase in the overall proneness to errors of applications	Ensuring a well-documented and user-friendly program structure as well as sustainable and cost-effective maintenance of applications

2.2 Scope

The SAP Development Policy is binding for all SAP systems included in the AHEAD programme. It is mandatory that any aspects specified in the following be implemented in new SAP developments. The policy also applies to SAP Notes that are implemented within the customer namespace. The IIT Expansion and IIT Security departments will review the implementation of the development policies at their own discretion. For the duration of the AHEAD project, assessments will be carried out by the Quality Assurance (QA) staff within the project team.

3 ABAP General programming guidelines

This section describes effective and recommended programming guidelines for applications that are based on the ABAP programming language. It explains how comprehensible ABAP code can be developed using standard SAP means. This development guideline has been created for SAP Net Weaver 7.5release or higher.

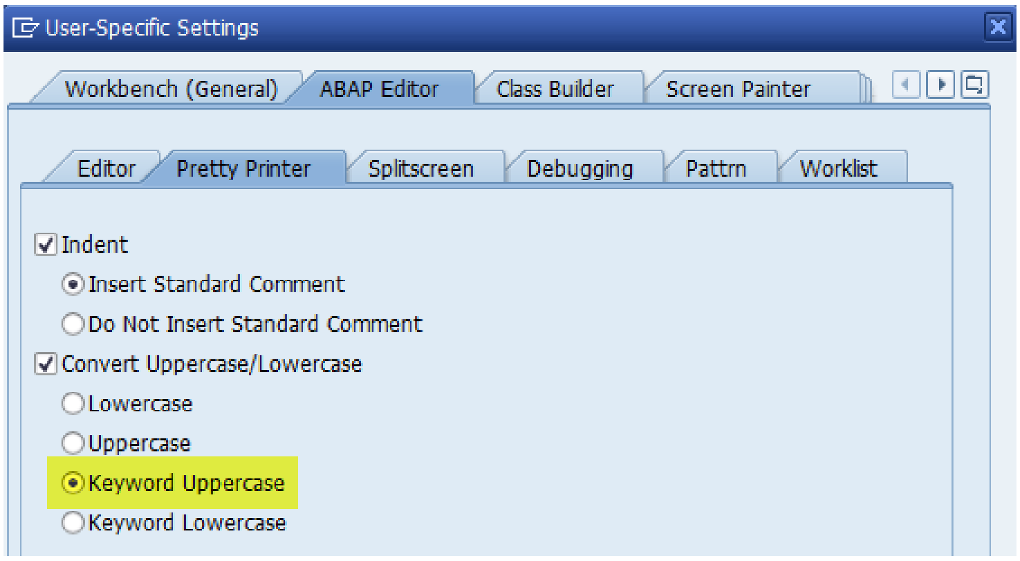
3.1 Structure and Style - Standardisation, readability, maintainability

The aim of this section is to give the developer several rules that allow to structure the development objects, thoughts on maintainability and efficient creation of customer-specific developments.

These rules are mandatory for all customer-specific development. Exceptions can only be handled on a case-to-case basis with prior approval.

3.1.1 Pretty Printer & code readability

- Use 'Pretty Printer' to ensure standardised formatting of source code
- Use the following settings in Pretty Printer.



[✓] Indent
[✓ Insert Standard Comment]
Do Not Insert Standard Comment
[✓] Convert Uppercase/Lowercase
Lowercase
Uppercase
[✓ Keyword Uppercase]
Keyword Lowercase

Figure 1: 'Pretty Printer' settings

In addition to the mandatory usage of the Pretty Printer, a few simple additional rules should be followed in custom code:

- Maximum of one statement per program line. Use empty lines to separate different logical sections.

```
CLASS class DEFINITION.
  PUBLIC SECTION.
    METHODS meth.
ENDCLASS.

CLASS class IMPLEMENTATION.
  METHOD meth.
    DATA: itab TYPE TABLE OF dbtab,
          wa   TYPE dbtab.
    SELECT *
          FROM dbtab
          WHERE column = ' '
          INTO TABLE @itab.
    IF sy-subrc <> 0.
      RETURN.
    ENDIF.
    LOOP AT itab INTO wa.
      ...
    ENDLOOP.
```

```
 ENDMETHOD.
ENDCLASS.
```

- Comment using “instead of * to achieve indentation by the Pretty Printer.
- A dash “—” shall never be used in program data or subroutine names except in table-field notations.

3.1.2 Program attributes

The attribute “Unicode checks active” & “Fixed point arithmetic” must always be set. These Flags must not be deleted during development.

3.1.3 Database operations

All Create/Read/Update/Delete operations on the database should use proper BAPIs or APIs provided by SAP to ensure data consistency. Only if such BAPIs/APIs are not available use Call-Transaction or Batch-Input processing via SAP standard transactions. In general, direct access to database tables using Open SQL has to be avoided if any of the previous options is available.
When working with custom <client> tables the similar rules apply. Whenever creating such a table, provide encapsulated access. Create custom class that takes care of all database operations for the <client> table.

3.1.4 Code Modularisation / reusable code

Following rules regarding structure and readability of coding have been defined that should be followed for every development. These are generic rules to improve maintainability of the code.

“If ABAP Objects becomes your default programming model, Methods will naturally become the most preferable instrument for modularising your programs”

- Functional units of code and repeated sections of code should be modularized (e.g. into separate methods).
- Encapsulating recurring program logic in methods and function modules in order to prevent unnecessary duplicates of code.
- Duplicate code may lead to incorrect functions as not all relevant modifications may be considered and adjusted to the new situation in case of changes.
- No logic is to be created within the main program itself. The main program exclusively consists in calls of the modules (function modules and methods).
  Large program blocks must be implemented as functional units. It makes the logical structure of a program more transparent. This holds especially for very complex programs. Functional units must be preferably methods in case of large programs.
- The selection screen, data definitions and modules are to be encapsulated appropriately. For e.g.
  - Selection Screen can be encapsulated in a Top Include Viz.
    Include program <program name>_TOP
    This part contains the definition of global data including selection screen.

- Data Definitions can be encapsulated in a Method.

  However, use of global data definition must be restricted to those that are necessary for technical reasons. Data must be declared in appropriate visibility sections of classes or as temporary working data in methods.

- Example of a modularised simple report in ABAP OO,
  - SEL – Class for SELECTION criteria
  - DATA (Model) – DATA Selection & processing class
  - ALV (View) – Class to Display data

  Since the idea is to modularise and decouple as much as possible. Move all the data selection and processing related logic into a DATA class. This DATA Class would have different methods to retrieve and process the data based on the selection criteria.

  Instead of creating the selection criteria as parameters to the method or Attributes of the data class and having those embedded in the DATA class, Create a separate class – SEL Class. This would provide an advantage of passing the SEL object to DATA class as well as to ALV class if required.
  Next step is to decouple the View related functionality out of the report and therefore to create ALV class.

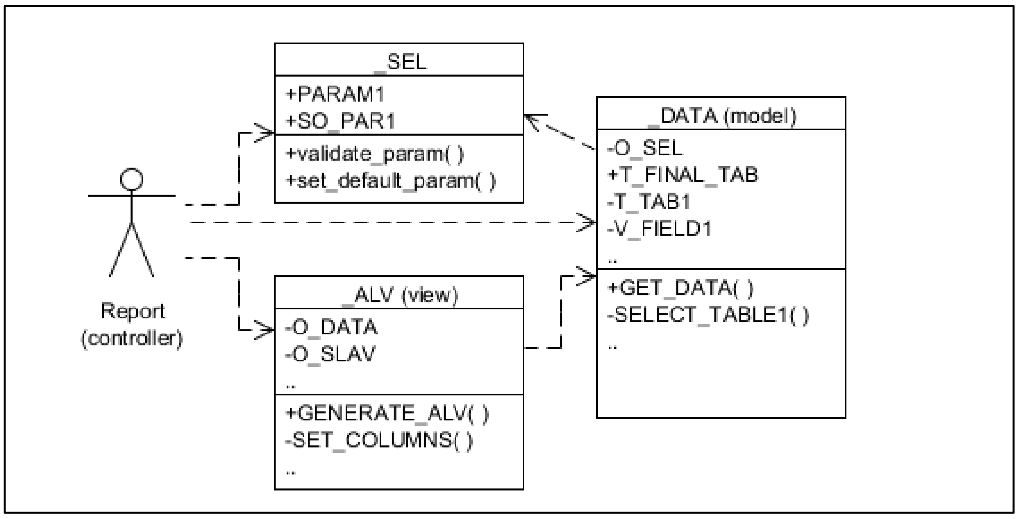


Figure 2: Program structure

3.1.4.1 Header Comment

We distinguish between the program header and line comments within the code. The header comment should contain the details (Development Id, user id of the author, change log, Change Request no., documentation, and reason for changes). Example of a program header:

```
*---------------------------------------------------------------------
* Identification
* Author         : USERID,
* Creation date   : 02.01.20XX
* Owner          : USERID, (Business/Functional)
* Development No.  : XXXXX
```

```
 * Short Description:
 * THIS IS AN EXAMPLE OF A PROGRAM HEADER
 *
 *----------------------------------------------------------------------
 * Changes
 * Chg. date  Developer Owner CRT-ReferenceID  Description
 * 02.02.20XX User     User  XXXXXXXX
 *----------------------------------------------------------------------
REPORT zapaprint MESSAGE-ID zapa.
```

Figure 4: Header Comment

SAP allows maintenance of “Pattern”. One single pattern Z_PROGRAM_HEADER will be created to store program header block containing information like “Owner”, “Creation Date” etc. as shown above. This should be used for all programs to maintain consistency across all programs developed in a project.
To maintain a pattern, open any custom program in change mode in SE38 and follow the path...

Utilities -> More utilities -> Edit pattern -> Create pattern

3.1.4.2 Data declaration

The most important declarative statements for data types and data objects (with virtually identical syntax) are:

- TYPES for data types
- DATA for variables
- CONSTANTS for constants

Declarations must be declared at the beginning of a modularisation unit – directly after the introducing statement and before the first functional statement. Sequence of declaration:

- Types
- Constants
- Data objects
- Field symbols
- local Class Declaration
- local Class Implementation

Data types and data objects are declared for one of the following contexts using declarative statements or inline declarations:

- ABAP programs:
  Declarative statements in the global declaration section define data types and data objects that are valid during the runtime of the program and visible in the entire program.
- Classes:
  Declarative statements in the declaration section of a class define data types and data objects (attributes) of a class and define their visibility.

- Instance attributes are valid during the lifetime of an instance of the class (object).
- Static attributes are valid from the first time the class is accessed until the end of the program.

Declarative Statements and Inline Declarations

DATA and @DATA Inline ABAP declarations available from release 7.40 to help make your code cleaner and more readable. This should be the preferred choice for data declaration over front up declaration mentioned above.

Inline ABAP DATA allows you to declare your internal table variables or work areas within the code that uses them. This is done simply by adding the DATA(wa_data) or @DATA(it_data) statements.

Data Declaration - Earlier

```
DATA:     it_ekko TYPE STANDARD TABLE OF EKKO,
   wa_ekko TYPE EKKO.
READ IT_EKKO INTO WA_EKKO.
```

Data Declaration – Inline Declaration:

```
READ TABLE it_ekko INDEX 1 INTO DATA(wa_ekko_new).
```

3.1.4.3 Subroutines & Methods

Subroutines: Modularization (subroutines) of code shall be used for logical break-up of code into smaller, readable, manageable and reusable blocks. However, care shall be taken to avoid over-modularization (as less as possible as much as necessary).

1. Reusability of subroutines shall be increased by passing variables through parameters to and from the subroutine, wherever possible.
2. External Subroutines: Subroutines that are part of other programs shall be called only if absolutely necessary, e.g. calling subroutines in standard SAP programs.
3. For internal subroutines (FORMs):
   i. Give a meaningful description of the processing of the form in the form header where the generated text “text” appears.
   ii. A short description of what a Subroutine does should precede the subroutine definition.
   iii. Hard coding, Parameter type mismatches and other such issues related to subroutines would have to be corrected as part of SLIN/SCI checks.

Methods: Methods are the object oriented terminology in ABAP for functions that belong to a class. The behaviour is the same as functions and subroutines. The main difference is within the intention, object oriented methods should modify attributes of the instance which it belongs too.

ABAP class methods support the following interface options.

- IMPORTING are values being sent to the function module, like USING from subroutines.
- EXPORTING are values that will change after execution, like CHANGING from subroutines.
- CHANGING much like exporting, and also CHANGING from subroutines.
- RETURNING If the function is assigned to a variable, this value is assigned to the variable

3.1.5 Object-oriented programming model

All New developments are to be implemented in an object-oriented manner (classes and methods instead of FORMs).

Exception

The following properties are still missing in ABAP Objects. They are needed to replace classic processing blocks with methods:

- Remote Method Invocation (RMI) as a replacement for Remote Function Call (RFC)
- A replacement for the call of update function modules (**CALL FUNCTION IN UPDATE TASK**)
- A replacement for the call of subroutines during **COMMIT WORK** and **ROLLBACK WORK** (**PERFORM ON COMMIT/ROLLBACK**)
- Object-oriented handling of classic dynpros, including selection screens as a replacement for dialog transactions, **CALL SCREEN** and **CALL SELECTION-SCREEN**
- Dynamic creation of classes as a replacement for classic dynamic program creation (**GENERATE SUBROUTINE POOL**)
- Direct support of background processing as a replacement for the call of
- executable programs (**SUBMIT VIA JOB**)

In these cases, the following classic processing blocks can still be created in new programs:

- Function modules are still required for RFC and the update and are recommended for calling classic dynpros and selection screens.
- Subroutines are still required for **PERFORM ON COMMIT$/ROLLBACK** and in dynamically generated subroutine pools (**GENERATE SUBROUTINE POOL**).
- Dialog modules and event blocks for selection screen events are still required in function groups that wrap the classic dynpros and selection screens.
- The **START-OF-SELECTION** event block is still required in executable programs that are intended for background processing.

Within this type of processing block, however, the execution should be delegated immediately to a suitable method. This does not have to be a method of a global class, but it can be located in the associated main program within the scope of a local class. To ensure that the system implements the same stricter check in these processing blocks as in the methods, the obsolete statements check (OO context) can be activated in the extended program check.

3.1.6 Development language

The originally language of development objects has to be English. All developments are to be carried out in English (GUI login: English).
Therefore, all development objects are to be created and maintained with English as the source language. The language must not be switched from English to German in order to allow the use of multilingual capabilities and automatic replacement of stored language files.
Program code must be readable and understandable by everyone who knows ABAP. If programs or parts of programs are not understandable by reading the source code or the documentation, you must use comments to explain the logic (why is it code in a particular way).

3.1.7 Localisation Language

- The application must be available in the following languages:
    - German
    - English
    - Italian
    - Slovenian
    - Hungarian
    - French
- Translations will have to be maintained by the development team and requested from the functional/application consultants. Translations need to be tested during Unit testing as a verification step.
- Language-dependent texts within programs must not be hard-coded and need to be cross verified through SCI. Instead they are to be maintained as text elements (program texts, class texts and 'Online Text Repository' [OTR] items), standard texts or message classes.
- Language-dependent (customisable) texts must be stored in separate text tables.

3.1.8 Obsolete language elements and development objects

It is to be avoided wherever possible to create the development objects listed below as customer-specific objects. Instead, they should be replaced with methods:

- FORM routines

⇨ For borderline cases, a search providing a reliable list of all obsolete language elements can be conducted via the 'ABAPDOCU' transaction using the 'obsolete elements' search term.

3.1.9 Portability, transferability

The applications to be created for ALDI are to be programmed in a portable manner, i.e. in such a way that they continue to be operational without any adjustments after a transfer to a successor system:

- URLs used as links: Only relative specifications are to be used, variables for servers/ports are to be automatically generated via existing SAP functions.

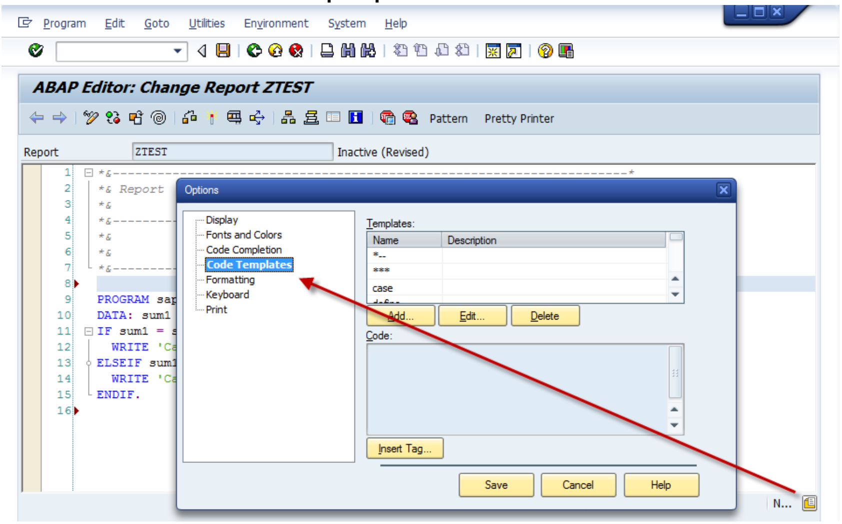
- Path information is not to be defined in an absolute manner. If absolute path information cannot be avoided, it is to be stored in the form of customising tables or mapped using logical directories/files ('FILE' transaction).

3.1.10 Using SAP Unreleased Function Modules

SAP standard product uses unreleased function modules. During the custom development, it is recommended to use the customer released function modules but use of unreleased function module is allowed. To have governance on it, the ATC (ABAP test Cockpit) has been enhanced to report the usage of unreleased function module as Error during check.
Developer(s) can submit the usage of unreleased function module as an exception with reasoning for approval during the release of transport.

3.2 Comments

- Meaningful comments shall be used in the code to enhance understanding of the logic for future changes. Comments should explain the intention of coding.
- Comment templates which provide a standardised comment layout for all developers can be used via the 'Options' dialog. The 'Code Templates' function is available for this purpose:



Report ZTEST Inactive (Revised)

```
*&---------------------------------------------------------------*
*& Report
*&
*&---------
*&
*&
*&---------

PROGRAM sap
DATA: sum1
IF sum1 = s
  WRITE 'Ca
ELSEIF sum1
  WRITE 'Ca
ENDIF.
```

Templates:

Name	Description
*--	
***	
case	

Figure 7: 'Options' dialog, code templates

Code templates can be entered using the <TAB> keyword. For example:
The 'CASE<TAB>' command inserts the following:

```
CASE .
 WHEN .
 WHEN .
 WHEN OTHERS.
ENDCASE.
```

The created code templates will be stored in a file on each developer PC. This also enables separate distribution in order to ensure the use of a uniform set of templates. Storage location: The location depends on the configuration.

However, the user directory is the standard location: *C:\Users\<username>\AppData\Roaming\SAP\SAP GUI\ABAP Editor*

Commenting changes

- Change to an existing code must be commented. The comments should be short and should describe the need for change and how it was changed.
- SAP Versioning will take care of what has been changed. The comments should only describe why it was changed and by whom.

When sending a message it is recommended to include the message or the text as comment. This makes at the same time clear what the purpose of the message is. Examples:

```
IF sy-subrc NE 0.
* No record found in table Z991k
  MESSAGE e001(zsfunctionmodules) RAISING no_z991k_record_found.
ENDIF.
```

Each form contains a description of its function in the generated heading. Along with the parameters. Commenting is illustrated in the examples below

Poor commenting

```
* If CSTTO is 6 then call form ABC
```

Acceptable commenting

```
* If the customer is a government (CSTTP = 6) call the form ABC
* that updates the Medicaid rebate file
```

Better commenting

```
* For government customers we must keep track of how much is sold each
* month and at what price. This information is later used to calculate
* rebates. Call the form that is used to capture this information for the
* current customer order.
```

3.3 Hardcoding Text Literals / Text Output formatting

Hard-coded values are not permissible where relevant text literals should be implemented as text elements. This serves to ensure that code using text literals can be easily read and also allows portability to other languages.
Hard-coding shall be avoided in the code in all sections after the global data declaration. The only allowed exceptions shall be function calls and return code checking.

Literals (critical to the functionality of the program) that are liable to change should be defined as data in a configuration table with requirement for change request for any changes in the data. This shall depend on the project requirement, but efforts shall be made to convince the project owners for this requirement due to its impact on the programs.

3.4 Accessibility of customer-specific customising tables

Any customer-specific customising tables must be made accessible to authorised key users. For this purpose, a maintenance view, which can be accessed via a transaction and via an entry in the relevant menu tree, is to be created for every table. The requirements of the authorisation concept are to be met.

3.5 Exception and error handling

1. All programs, modules and classes must be provided with adequate error handling.
2. Undefined cancellations must not be permitted.
3. The 'SY-SUBRC' system status or intercepted error classes must be analysed in all cases. Further processing without an analysis is not permissible. In case of a user error an error message must be given. In case of an error not caused by the user, for every individual situation must be decided of an error message is required.
4. The statements CALL FUNCTION and CALL METHOD do not automatically set the variable SY-SUBRC. The variable SY-SUBRC is only set when the following occurs.
   - First, the called module needs to use non-class-based exceptions.
   - Second, a value not equal to zero needs to be assigned to the defined exceptions in the EXCEPTIONS section of the statement.
5. Always assign a value to the special exception OTHERS. This assures that the variable SY-SUBRC is set even if the list of exceptions of the called module changes. Especially when calling RFC function modules a value should be assigned to the exception OTHERS. The reason is that RFC function modules might raise special RFC-related exceptions.
6. The value of the global variable always needs to be checked immediately after the statement that sets SY-SUBRC. The reason is that any subsequent statement might change the value of SY-SUBRC.

➔ Exception means a deviation from the normal program flow.
➔ Exception handling means the way in which deviations are detected, caught and handled.

- Implementing proper exception handling makes a program behave more predictable and enhance the program's technical quality. Simply put, good exception handling will result in fewer errors.

  At least the following types of exceptions have to be handled:
  - Return code (note that not every return code indicates an exception)
  - Unexpected data in conditions
  - Interface parameter of a subroutine
  - Exception generated by a function module
  - Class-based exceptions
  - Runtime error

Acting on exceptions
The main criteria that determine the steps to be taken after an exception has been detected are:

- Should the program send a message?
- Should the program execute clean up actions?
- Should the program ask for a correction of data?
- Should the program continue after the exception has been handled?

- As a rule, use CLASS BASED EXCEPTIONS. The other exceptions (Classic Exceptions, Catchable runtime errors) are only there for backward compatibility and are mostly superseded by Class-Based exceptions.
- It is important to catch all exceptions that can be handled at the point they occur.
- Exceptions that cannot be handled need to be declared in methods and function modules where they might throw such an exception. Exceptions should be handled in a consistent manner within an application. I.e. for the same error situation the same error handling or exception should be used.

- All class-based exceptions of all modules of a program need to be handled using a TRY…CATCH…-ENDTRY statement

- Make sure that EMPTY CATCH is never used.

- Base Class CX_ROOT should only be used in CATCH clause if exception handling cannot handle all possible unknown errors.
  - E.g. an example is the exception CX_SY_ZERODIVIDE to handle a division by zero error.

- Carefully choose the type of your own exceptions classes by inhering from the predefined base classes (CX_STATIC_CHECK, CX_DYNAMIC_CHECK, CX_NO_CHECK).
  - CX_DYNAMIC_CHECK generally should be used as a base class when introducing class-based exceptions and preferred over CX_STATIC_CHECK due to enforced syntax related checks.

- The exception class CX_NO_CHECK is useful for exceptions that cannot be handled appropriately by a caller. Example of such exceptions would be the failure of a secondary database connection or other events that cannot be corrected in the program code.

- In cases where exceptions cannot be handled, check preconditions of the statement which raises an exception that cannot be handled prior to its execution.
  - E.g., OPEN DATASET causes a short dump if the user does not have sufficient authorisations to open a file. To prevent this, the user authorisation needs to be checked (via the function module AUTHORITY_CHECK_DATASET) prior to calling OPEN DATASET.
- No error checking can be done for errors occurring in an update task. SAP automatically handles these and will abort the changes made depending on the priority of the included function modules.
- When a function module has been called, the return code field, if one is defined, shall be checked for success or failure.
- Avoid using the MESSAGE statement in modules without direct user interaction and in modules used in defined user dialog layers. The MESSAGE statement in conjunction with certain message types and execution modes might cause

explicit COMMITs in the context of dynpros or the termination of a connection in the context of RFC function calls.

3.5.1 LOGGING of errors, exceptions and messages

Errors, exceptions and log messages in general should be stored in the business application log.
This enables a central checking of all messages via transaction "Application Log: Display Logs"
(SLG1). Furthermore, the transaction "Application Log: Object Maintenance" (SLG0) enables the
Definition of custom log objects.

Application log should be the preferred source of repository for errors and warnings that occur during background jobs processing or interfaces.

The main advantages of the usage of the business application log are:

1. Central Repository: the business application log provides a central repository. This central repository simplifies the administration of different applications.
2. Reusability: the business application log and the related function modules and classes provide comprehensive functionalities for logging without the need for custom development.

Examples of the functionalities are:

- Integration of custom fields into a log object
- Hierarchical display of messages and their aggregation into problem clusters
- Functions to interactively read additional message information upon display of a message
- Persistent storage of messages in the log
- Ability to integrate the protocol display into custom development objects (via sub-screens, controls or pop ups)

Detailed documentation on all function modules, and so on, can be found in the system by executing the program SBAL_DOCUMENTATION

For quick reference of available functions, check this at help.sap.com
https://help.sap.com/viewer/addb96cd90c945dfb3182865363bbc47/7.51.0/en-US/4e23b1720771417fe10000000a15822b.html

The example programs "SBAL_DEMO*..." can be used to understand the functionality of Application Log.

3.6 Handling Time Zones and Localization

Be aware that users and external systems may operate in different time zones. The functional specification for a new development shall describe if and how time zones

shall be realized. The usage of time zones shall be consistent for the whole new application.
If not specified otherwise use the localization setting for the current user to convert and display data.
Interfaces and data communication must ensure that the time zone information is transferred properly and shall use the ISO-8601 format for the message data type, whenever that is possible. The ABAP class CL_GDT_CONVERSION is very useful in that context.
The time zone logic and behaviour must be well documented. It is recommended to store times in UTC. Testing of proper handling time zones requires special attention. Useful tests require users or systems (on one side) and the server (on the other side) to actually be in different time zones. Consider to verify the Daylight Saving Time Switch as well, if relevant.

3.7 Unit of Measurements & Currency Codes

- Always ensure to consider the Unit of Measure for any processing related to weights / mass / quantity operations. If you need to include a new quantity field in a custom table always have an associated Unit of Measure field for the quantity field.
- Same for Currency fields. Every operation on currency fields should take Currency code field in consideration. Every currency field added to a table should be accompanied by a currency code field.

3.8 Checkpoint Groups

Use Checkpoint Groups and the commands “ASSERT ID…”, “BREAK-POINT ID…”, “LOG-POINT” to make implicit assumptions in the coding visible and traceable. **It is good a programming practice**

3.9 Data Definition

Under all circumstances, avoid the use of global data. Exceptions to the rule are Screen Parameters and Select Options. Use of global variables must be avoided wherever possible. Especially global work areas must not be used as they often cause problems and reduce maintainability significantly.
Data objects must be named in accordance with Naming Standards. Within existing programs, the current data objects are not renamed. Class-Methods / INCLUDES are created for data declarations and any data structures that will be used in more than one ABAP program.
<u>Guideline:</u>
- Variable names should be descriptive, meaningful and understandable.
- During declaration, all fields and tables must be initialised in such a way that no random values can occur [for direct or indirect use], e.g., by means of the ‘Clear’ or ‘Refresh’ functions.
- Use field symbols judiciously. If "field symbols" are used, their use must be fully explained in program comments at the place where the symbol is defined and used.
- Avoid use of field symbols with character or byte type data objects if the only reason is to achieve Dynamic offset/length programming.
- Before accessing the data referenced by field symbol or a reference variable, always ensure that the field symbol IS ASSIGNED or the reference variable IS BOUND thereby preventing runtime errors.

3.10 Constants

- Using centrally defined constants instead of hard-coded strings, numbers, usernames, organisational units or data (e.g. attributes of global classes and interfaces)
- Using customer-specific customising tables for replacing variables if the variables are not defined as global constants.

- Use Constants instead of numeric literals.
- Constants can be defined in public sections of global classes or in global interfaces
- Constants, which belong together, shall be grouped.
- E.g. :

```
CONSTANTS: BEGIN OF gc_status,
        inactive TYPE i VALUE 0,
        active   TYPE i VALUE 1,
        on_hold  TYPE i VALUE 2,
      END OF gc_status.

IF iv_status = gc_status-active.
 ...
ENDIF.
```

3.11 Parameters in subroutines

The ‘LIKE’ or ‘TYPE’ statements must be used for all parameters. Combined with good programming style, this enables the ‘ABAP Compiler’ to generate more efficient code (performance can be enhanced to reach up to twice the speed).
Example:

```
FORM <subroutine>       TABLES        ct_tab1 LIKE <tab1>[]
                  USING  iv_param1 LIKE <data>
                         iv_param2 LIKE <dd-field>
                         is_param3 LIKE <dd-structure>
                  CHANGING      cv_param4 TYPE <standard data type>
                                cv_param5 TYPE <user defined data type>
....
ENDFORM.
```

3.12 Nesting (LOOP..ENDLOOP)

- “COMMIT”, “UPDATE” or “SELECT” statements or function calls shall not be used in LOOPs unless otherwise required by design, e.g. When doing mass updates, a “COMMIT” should be performed approximately every 1000 records, and would normally be done in a LOOP. Large loops shall be optimized by modularisation to improve readability and re-usability.
- Avoid complex nesting loops and conditions. Especially when using nested loops, implement parallel cursor techniques.
- Join two internal tables using parallel cursor (ASSIGNING).

Note: In this example, TAB1 and TAB2 are sorted by K in ascending order.

```
I2 = 1.
LOOP AT TAB1 ASSIGNING <FS_TAB1>.
   LOOP AT TAB2 ASSIGNING <FS_TAB2> FROM I2.
     IF <FS_TAB2>-K <> <FS_TAB1>-K.
         I2 = SY-TABIX.
         EXIT.
     ENDIF.
   ENDLOOP.
ENDLOOP.
```

3.13 Conditions (IF', 'CASE')

- Conditions ('IF', 'CASE', etc.) are to be defined in such a way that all cases are covered at all times, i.e. at the end of the definition, it must be specified how the complement of the included cases is to be handled ('ELSEIF', 'OTHERS', etc.).
- CHECK shall be used instead of instead of IF...ENDIF whenever possible. This is not structured programming; however, this may affect the performance considerably if used many times in a subroutine to avoid nesting. However, use discretion while using CHECK and EXIT statements in enhancements. (Exits, Sections, Points etc).Use compare operators (=, <, >, <>) instead of EQ, NE, LT, GT
- Never perform condition checks with hardcoded values.

3.14 Data Processing

Data may not always be used in the form that it is available to the program. Data may be processed to achieve the required output/ format of output.

- Necessary format masks shall be used when outputting data, e.g. time and date masks.
- Project-wide translation subroutines should be used for translation of data into a specific formats or specific values.
- When outputting currencies or quantities, units shall be considered.
- When data is exchanged with systems other than SAP, sign conversions, field types, field width, etc. shall be carefully considered and the result checked.
- When moving data from variable of one type to variable of another type, sign conversion, truncation, overflow, type mismatch, output format etc. shall be carefully considered and checked.
- When using addition, multiplication features, field overflow shall always be checked and coded for.
- For any dataset operation statements like OPEN, READ, TRANSFER and CLOSE the system return code SY-SUBRC shall be checked.

3.15 Record Locking

- Locks are defined generically as "lock objects" in the Data Dictionary.

- A lock entry is a lock object and locks a particular data object, such as a correction request or a table entry.
- Normally, lock entries are set and deleted automatically when programs access and subsequently release data objects.
- Locks should be retained for as short a time as possible but as long as necessary.
- To lock SAP standard business objects at the database level the respective SAP standard lock objects should be used.
- After an update has been performed the lock of a business object should be released again by
  - Calling the appropriate de-queue function module. In this context special care should be given to the scope parameter if update function modules are used.
- While using Locks, the sequence of Update should be
  - Lock the related object.
  - Select the related data. i.e. Read Data
  - Update the records
  - Unlock the objects.

You can use different types of locks to synchronize database access.
The **lock mode** describes what type of lock it is. The lock modes are listed in the table below.

Type of Lock	Lock Mode	Meaning
Shared Lock	**S** (**S**hared)	Several users (transactions) can access locked data at the same time in display mode. Requests from further shared locks are accepted, even if they are from different users. An exclusive lock (E) set by another user on an object that already has a shared lock will be rejected. Every extended exclusive lock (X) will also be rejected.
Exclusive Locks	**E** (**E**xclusive)	An exclusive lock protects the locked object against all types of locks from other transactions. Only the same lock owner can reset the lock (accumulate).
Exclusive but not cumulative lock	**X** (e**X**clusive non-cumulative)	Whereas exclusive locks can be requested several times by the same transaction and released one by one, an exclusive, non-cumulative lock can only be requested once by the same transaction. Each further lock request will be rejected.
Optimistic Lock	**O** (**O**ptimistic)	Optimistic locks initially behave like shared locks and can be converted into exclusive locks.

- Note: There are also lock modes U, V, and W. these only perform the collision check, but do not set a lock, and cannot therefore appear in the lock table.

- SAP Lock concept is a very important aspect of developing applications in SAP and hence should be given the due importance.
- For more details , Kindly go through the below SAP Help site -
  https://uacp2.hana.ondemand.com/viewer/a7b390faab1140c087b8926571e942b7/7.51.1/en-US/47daeac909dd3020e10000000a42189d.html

3.16 Other General recommendations

- Classic list programming is no longer to be used. The use of ALV Grid Control (ALV = SAP List Viewer) is the minimum requirement for displaying lists.
- Keep programs to a reasonable length. Programs should be long enough so that there is not an excessive amount of data being passed between two related programs but should be short enough so that the same program does not perform many discrete functions.
- Observe prudence while using Macros. According to SAP, macros can also be used as a means of program organisation. However, compiling customer-specific macros is not permissible. Existing SAP macros may be used.
- Separating presentation logic and application logic (Model-View-Controller (MVC) concept) in order to enable the use of different user interface technologies
- When generating output files:
    - The files have to be stored in the file system using logical file paths and names.
    - Using directories on the system drive is not permissible.
- Removal of program parts that are not used or not accessible DO NOT COMMENT AND KEEP THEM IN THE CODE. Dead Code should be removed from the program, i.e., the fields that are never referenced and code which can never logically executed.
- Do not use complicated arithmetic expression, avoid COMPUTE, ADD, and other keyword whenever possible, e.g. using DATA = DATA + 1 instead of ADD 1 to DATA.
- Using dynamic programming in a well-weighted and controlled manner, i.e. the following aspects are to be avoided:
    - It must be ensured that the created dynamic code can be understood by other developers in order to allow further maintenance by these developers.
    - Potential errors are to be prevented in order to avoid runtime errors.
- Do not use pseudo comment in code.
    - The character string "#EC after a statement or a part of a statement, which is followed by an ID with the prefix "CI_", defines a pseudo comment for Code Inspector.
    - These pseudo comments must not be used to hide warnings from Code Inspector for the statement in question.
- Do not use obsolete statements. Obsolete statements are still available only for reasons of compatibility with older releases. These statements may still be encountered in older programs but should not be used in new programs.
- Data selection from database table with dynamic table name should not be performed.
- SELECT statement using the option of BYPASSING BUFFER on a buffered table is not recommended due to its impacts on the overall performance.

- Reports and Includes with a comment/code ratio of less than or equal to a 30 percentage (default) are not recommended.
- Database HINTS (e.g. **%_HINTS HDB)** must not be used in code.
  Syntax:
  ```
  ...%_HINTS db @dbhint1
  [db @dbhint2
  ... ] ...
  ```
- GET/SET PARAMETER ID should be avoided.
- It is not allowed to retrieve variable values from the memory of the previous programs in ABAP stack. In cases when such retrieval is the only way to achieve the functionality, the same has to be aligned with and pre-approved by the DQA team (DQA@aldi-sued.com).

3.17 Data dictionary

- Existing data elements are to be used. New data elements may only be used to map customer-specific fields that have not yet been described by SAP.
- SAP standard structures and tables must not be modified.
- If an extension of the SAP standard is required that is not covered by the extension options provided by SAP, the extension may only be implemented using the 'Append' function.
  - Customer-specific fields are to be inserted at the end of the structure/table.
  - In customer-specific (append) structures/table types for SAP standard tables/structures are to start with 'Z' and field names are to start with 'ZZ'
  - Fields must not be removed (only permissible in exceptional cases, implementation in tables required).
  - Only one customer-specific append structure may be implemented per structure/table/table type.
- Customer-specific tables are to be created as transparent tables. Customer-specific tables are to meet the following requirements:
  - Insert the client field (CLIENT/MANDT) in all tables
  - Client-independent tables may only be used in justified exceptional cases.
- Determine if creating a table maintenance view will suffice when accessing the table (instead of creating a new screen).
- Determine if creating a view will optimise database accesses.
- Custom tables must be archive able (technical setting). For this purpose, archiving and retention periods are to be proposed and recorded in writing as early as during the concept design phase wherever possible and in particular when relating to transaction data.
- Check tables are to be used for all fields within customer-specific tables wherever possible.

3.18 Functional Groups and Function Modules

Functional Groups: A function group must contain only function modules that relate to each other as an integrated solution. This is because the whole function group is loaded into the internal session when one of the function modules is called.
Unless otherwise not possible, always use Classes with Static Methods in place of Function Modules.

Function Modules: Function modules are contained in programs named after the function group i.e. SAPL<function group>. The documentation for a function module, however, must be maintained in each function module. The documentation must state clearly the description of all parameters of the function module

- Exception handling must be performed by the calling program to detect success or failure of the processing within the function module.
- After a call function, it is the developer's responsibility to check the return code and provide meaningful messaging
  - ***Incorrect:***

```
IF sy-subrc <> 0.
MESSAGE ID sy-msgid TYPE sy-msgty NUMBER sy-msgno
WITH sy-msgv1 sy-msgv2 sy-msgv3 sy-msgv4.
Endif.
```

Correct:

Raise exception when SUBRC is NOT 0.
Refer to Section 3.5 for Exception Handling.

3.19 Module pool

A module pool is a program object that is used to group together all the objects that form an on-line program. The name should describe the purpose of the dialog module. The documentation must state clearly the screens, statuses, chaining, operation modes, menus, and general flow structures used.

- Tab-strips shall be used to navigate between different sub-screens where such navigation is required by design. ALV Grids shall be used instead of step-loops.
- All the screens shall be designed in line with the SAP standard screens with standard functions, functionalities and navigation (e.g. BACK, EXIT, CANCEL, SAVE, DELETE, CONTINUE, etc.) where applicable, unless required otherwise by design.

Modules

Small single purpose modules shall be created and used.

If the module is specific to one screen, the name of the module should have the format xx….x_0100 where 0100 would be the screen number. If the module is not specific to one screen, the name should not have a screen number on the end.

Example:

```
MODULE status_0100 OUTPUT.
  . . .
ENDMODULE.
```

3.20 Messages / Message class

Message class

- Before creating a customer-specific class/message, it must be checked whether the SAP standard provides suitable messages in this environment. Important: SAP standard messages must not be adjusted (no modification option).

- Customer-specific message classes must be created with the same content and meaning for all languages used. For reasons of comprehensibility, the definition and translations are to be agreed with the functional/business teams.
- No highlighting or punctuation (e.g. '!!', '??', all letters capitalised) is to be considered for these texts, technical terms must not be used as they may not be understood by the user (e.g. 'Data record not available in VBAK').
- The maximum length of a message must not exceed 73 characters. Variables must not be used to expand messages. Variables may only be used for actual variable content available in the programs. If notifications are used that are not displayed via the 'Message' command, a special code line, which is not executable, is to be inserted to ensure that the notification can be found via the where-used list. however
- Class exceptions can handle messages and where-used list will work in that case.

Messages:
Where possible, use variables in messages. Messages, above all, should be meaningful to the user.
Long text must be maintained for messages wherever possible as this will enhance the user's understanding of the message.
Messages and long text shall be maintained in all the required native languages of the end-users that will make use of these messages and long texts.
New message classes shall be based on logical grouping of usage where sensible, e.g. for group of codes in a custom application, for group of codes for specific modules, group of similar type of codes like reports, etc.

- Where possible, variables shall be used in messages to provide clarity to messages, e.g. it is not sufficient to generalize that Customer is not defined for Company code, this shall be further clarified to the user with the customer number and the company code.
- When long text is not maintained for a message, the self-explanatory option shall be checked while creating the message.
- Generic messages '& & & &' are to be avoided and instead specific messages with data values to be passed as variables instead of texts.

3.21 Remote Function module (RFC)

A remote function call (RFC) is the call of a function module that runs in a different system to the calling program. Although it is also possible to call a function module in the same system as an RFC, normally RFCs are used when the caller and the called function module run in different systems.

You can use the CALL FUNCTION statement to call remote functions, just as you would call local function modules. However, you must include an additional DESTINATION clause to define where the function should run:

In contrast to the normal function module call, the following restrictions apply to an RFC:

- For each call that is made using synchronous RFC, a database commit is performed. For this reason, a synchronous RFC must not be used between Open SQL statements that open or close a database cursor.
- In a remotely called function module, you must not use statements that close the current context and thus the connection. An example of this is the statement LEAVE PROGRAM or SUBMIT without the addition RETURN.
- In the case of a synchronous RFC, dynpros and selection screens that are called in a remotely called function module are displayed in the calling system if the calling program is executed in dialog processing, and if the user defined in the destination has dialog authorisation. The screen data is transmitted by the RFC interface to the calling system. In this case, you can display lists that are written in a remotely called function module by using LEAVE TO LIST-PROCESSING.
- As only pass by value is used for the RFC, when exceptions do occur, you can never access interim results when a synchronous RFC is made.
- Information messages are warnings are handled in the same way as status messages.

For more details on RFC and RFC programming, kindly, go through the below SAP Help Site
https://uacp2.hana.ondemand.com/viewer/ff18034f08af4d7bb33894c2047c3b71/7.5.5/en-US/4888068ad9134076e10000000a42189d.html

3.22 Packages

Packages are designed to modularize, encapsulate and decouple development objects in the SAP system. This allows the developer to organize development objects that belong together. Therefore all objects that are part of one development must be part of one package. E.g. all components of a program (includes, transactions, structures, tables ...) must be assigned to one common package
Using the packages hierarchy is useful to join several packages within one main package. E.g. when several programs are technically independent but belong to one common business process.
Package allocations:
Package allocations are to be structured hierarchically. The top node is to be mapped as a structural package. The mapped sub-packages are to be allocated to the main package.
Use of the '[_country code]' wildcard is optional and applicable to country-specific adjustments to the relevant package.

Hierarchy	**Pattern**
Structural package	Z<SAP_APPLICATION>
Main package	Z<SAP_APPLICATION>_<APPL_COMPONENT>[_Country code]
Sub-package	Z<SAP_APPLICATION>_<APPL_COMPONENT>_<SUB_CONTENT>[_country code]

Table 11: Naming convention for development packages

Country	**Code**
Germany	DE

USA	US
Australia	AU
UK/Ireland	GB
Austria/Hungary	AT

3.23 Internal tables

Internal tables are a central construct within the application development with ABAP Specific attention should be given when defining internal tables, in particular large ones. For information about the use of the SIZE parameter read the help documentation for DATA/INTERNAL TABLES/PERFORMANCE NOTES ON INTERNAL TABLES

- Internal tables shall be declared without header lines, using separate work areas.
- The variable name of a table must be a unique identifier for this table
- Different types of internal tables, viz. standard, sorted, hashed, etc. shall be considered for use when such functionality is required. Apart from performance advantages that can be obtained, the different table types can have a different syntax for table operations.
- Please note: Working with internal tables can have a significant impact on the performance and runtime of the programs.
- The collect statement should not be used unless absolutely necessary and then only on small amounts of data. CLEAR statement shall be used to clear the contents of an internal table work area/ header line immediately after contents of such a work area/ header line are no longer required. However, CLEAR statement shall be used carefully, as this statement shall clear the table contents or the internal tables with header lines

For more details on Internal Table, Kindly go through the below SAP Help Site:
https://uacp2.hana.ondemand.com/viewer/a7b390faab1140c087b8926571e942b7/7.51.1/en-US/29deb7eddc4f47cf8c919661cdf52de2.html

3.24 Code changes for Process involving Legacy system

It is recommended to have minimum custom developments or changes to the solution in SAP system. So, wherever possible, give priority to make developments or changes in **legacy** systems to achieve the required functionality. Changes to SAP system should be done only if it is **more feasible** than doing changes to **legacy** systems involved.

4 SAP standard tables updates - Business Application Programming Interface (BAPI)

In general, SAP tables are not to be directly updated by programs (‘INSERT’, ‘UPDATE’, ‘DELETE’). SAP tables can be updated using the following methods:

- A ‘BAPI’ is to be used if available.
- If a ‘BAPI’ is not available, it must be checked whether a relevant function module is available.

- As a last means, a 'CALL TRANSACTION' may be used to call up tables (authorisation must be considered, is to be checked separately).

Other updating options have not been approved and must be agreed with the SAP Design Authority if required.

Deviations from the above-described restriction based on the existing approval process are only permissible if all of the requirements listed below are fulfilled. Such developments are categorised as Scope 4 in accordance with the governance model.

- The coding containing unwanted language elements is derived from SAP Notes <u>and</u>
- configuration table entries are exclusively adjusted <u>and</u>
- the concerned report is not regularly included in a batch <u>and</u>
- Changes are logged in the database log.

4.1 Custom BAPI Development:

BAPI must be part of the Business Object Repository (BOR). Use Business Object Builder (BOB – Transaction SW01) to associate a BAPI with a Business Object. Please refer to SAP BAPI Programming Guide for more detail on programming with BAPI's. In 6.0, use transaction BAPI to get the relevant documentation. Other sources are http://service.sap.com/bapi

- Use standard Method names for standard functionality i.e. GetList, GetDetail, GetStatus, CreateFromData.
- Separate individual components of a BAPI name by using Upper and lower case.
- BAPI Interface consists of Import and Export only. No Tables and no Exceptions should be used in the BAPI Interface.
- All messages need to be returned to the calling program via Return Structures:
  BAPIRET1 or BAPIRETURN
  BAPIRET2
- Adding a BAPI to an existing SAP object should be done using Subtypes and should not have a Z* as part of the naming convention. Z* is only used for BAPI's belonging to custom objects.
- All messages that can be issued by a BAPI must be listed in the BAPI Documentation
- Some Basic Rules:
  - Always perform database changes using the update task
  - No screen output allowed in BAPI
  - Cannot cause program termination i.e. message type 'A'
  - Cannot dispatch commit work - Must use BapiService.TransactionCommit (Function Module BAPI_TRANSACTION_COMMIT)

5 SAP ENHANCEMENTS (Adjusting SAP functions)

5.1 General

No changes must be made to SAP standard development objects. Modifying SAP software is a likely source of errors and creates excess work during software upgrades.

If developments can only be implemented by changing SAP standard objects, ensure that these changes are kept to a minimum. Bigger changes to SAP objects must be restricted, if possible, to one branch, and the entire processing logic must be implemented in the form of a customer development (within a method). Especially implementations of explicit and implicit enhancement options as well as modifications must be completely encapsulated from the SAP standard coding.

Add-ons that are attached to exits have the following advantages:

- They do not affect standard SAP source code. When new functionality is added to an R/3 system using SAP exits, the code and screens created are encapsulated as separate objects. These customer exits are linked to standard applications, but they are distinct from the standard SAP software package.
- They do not affect software upgrades. When new functions are added using SAP exits, the objects are customer objects, which adhere to strict naming conventions. When the time comes to upgrade a software release, the special names of customer objects ensure that they will not be affected by either changes or additions within the standard software package. As a result, you do not need to save and then re-enter add-ons attached to exits

5.1.1 Extensions

User exits, customer exits, BTEs (business transaction events) and BAdIs (business add-ins) are the preferred options for including extensions, if they are available and that the interface can provide the required data.

5.1.2 Enhancement Framework

If the technology specified in 5.1.1 (Extensions) is not available, enhancements can be used as a means for adding extensions. If enhancements are used, it must be ensured that the compatibility of support packages and upgrades that are to be implemented is assessed by means of the 'SPAU_ENH' transaction.

Explicit and Implicit Enhancement Options:

- The developer of the corresponding development object must insert the options of one kind into the coding so that enhancements can be done there at a later time. These preconceived enhancement possibilities are called explicit enhancement options.
- You can perform enhancements on implicit enhancement options without the developer of the appropriate compilation unit having to do anything. Enhancement options are always available in programs, global classes, function modules, and includes.

  In other words, the implicit enhancement options are for free, provided by the framework, while the explicit ones need to be inserted explicitly, as the name indicates.

Explicit Enhancement options:

There are two types of explicit enhancement options:

- **Enhancement points** allow you to insert source code plug-ins. These are additional code lines that, if they exist, are executed there additionally. Explicit enhancement options of the type Enhancement Section behave in the same way - the only difference being that the source code plug-in replaces the section in the original code.
- **Business Add-Ins (BAdIs)** are "hooks" for object plug-ins. A BAdI definition comprises an interface with methods. BAdIs are enhanced by classes that implement the BAdI interface. If you instantiate a BAdI and then call its methods, you can, among other things, specify which method implementations are to be carried out based on filter values. In other words, a BAdI method call is a dynamic method call with a specified interface, for which it is not determined until runtime which method implementations are to take place.

Implicit Enhancement Options

Implicit enhancement options are fixed points in compilation units - that is, points that remain intact even if the code is changed:

- You can always insert source code plug-ins before the first and after the last line of includes, methods, reports, and function modules.
- You can always add further optional parameters to function modules.
- For global classes, there are different permanent, implicit options for enhancements: You can insert additional attributes or methods, and you can add optional parameters to existing methods.

Refer
https://wiki.scn.sap.com/wiki/display/ABAP/The+new+Enhancement+Framework+and +the+new+kernel-based+BAdI for more details about enhancement framework.

Refer https://wiki.scn.sap.com/wiki/pages/viewpage.action?pageId=77988719 to view sample example for enhancement framework.

5.1.3 Menu exits

Items may be added to the pull-down menus of standard R/3 applications (for example, to call up new screens or to trigger entire add-on applications). SAP creates menu exits by defining special menu items in the Menu Painter. These special entries have function codes that begin with + (plus sign). The text of the menu item is specified when activating the item within an add-on project.

5.1.4 Screen exits

Fields may be added to the screens within R/3 applications. SAP creates screen exits by placing special sub-screen areas within a standard R/3 screen and calling a customer sub-screen from within flow logic of the standard screen.

5.1.5 Adjustments within the ALDI namespace

Adjustments to copies of SAP standard code within the customer namespace require a high level of maintenance effort. Therefore, minimise these adjustments to ONLY cases where no other solution is feasible. This also requires technical lead approval.

6 SAP Workflow

SAP workflow is designed through BOR(Business object repository) or OOPS ABAP. But, enhancing standard objects or using Subtypes is not a best practice anymore and we must make use of ABAP classes instead.

Advantages of using OOPS ABAP for Workflow,

- It uses the newer ABAP editor
- Object Type can be used within or in combination of the ABAP class to make use of existing functionality
- Object Types encapsulate functionality within SAP in an 'object wrapper' and can be accessed uniquely under an identifying key, normally comprising the keys of an ABAP Dictionary table.

6.1 SAP Workflow components

Workflow has following components

- **Event:** Event is triggered from on specific activity in system. Event can either be standard or it may be raised using custom code.

- **Object:** Object here is defined using Class or Business Object. Event is typically and should be associated with a BO/Class. It offers better tracking and reporting if we use object events.

- **Workflow:** Workflow defined with the Event in the system is triggered

- **Task:** Tasks defined in the workflow body are executed sequentially.

- **Steps:** There are numerous steps available within the workflow like process control, send mail & container operation which are executed within the instance of the workflow.

- **Rule:** Rule is used to derive agents when agent determination is dynamic and requires complex logic to be implemented.

- **Agent:** Agent is the end user who receives the workitem for its action. Agents can be divided into actual agents who receive the workitem, possible agents are the group of agents entitled to receive the workitem and excluded agents are the group of people who must be excluded from the process of approval.

- **Workitem:** Workitem is a single instance of a foreground task in workflow which can be uniquely identified using a number called as workitem. The workitem is only identified using a unique number and there are other properties associated with it for eg, workitem type, Deadlines, workitem text.

- **Business Workplace:** Business workplace is the application where agent receives the workitem for further processing.

6.2 Implementation of OOPS ABAP based Workflow

Workflow uses the IF_WORKFLOW interface to generically and efficiently handle all references to ABAP OO for Workflow classes. So, we need to implement IF_**WORKFLOW** interface for accessing workflow objects in class. Also, make sure that all the inherited methods are implemented, at least with blank code, and activated.
Following methods are inherited while implementing IF_WORKFLOW interface.

- **BI_PERSISTENT~FIND_BY_LPOR:** This is a Static method that is used to convert from the Local Persistent Object Reference used by workflow to a unique instance of the ABAP Class. It's used when workflow needs to implicitly instantiate the ABAP Class.

- **BI_PERSISTENT~LPOR:** This is an Instance method that converts from the current instance of the ABAP Class to the Local Persistent Object Reference used by workflow. It's used when the workflow wants to pass a reference to the ABAP Class.

- **BI_PERSISTENT~REFRESH:** This method is used to refresh the object, i.e. reload database and other stored values. It can be used to clear reload existing attributes that were filled in Constructor and Class Constructor methods.

- **BI_OBJECT~DEFAULT_ATTRIBUTE_VALUE:** This method is used in workflow inboxes, such as the Universal Worklist, and workflow administration tools to indicate which unique instance of the class is being referenced. By default, whatever value is in the INSTID field of the Local Persistent Object Reference will be shown. If you want to show a different attribute, just add the relevant code.
For example, if a class is representing an instance of a Plant, then the key might be Plant Id, but to show the attribute "Description" instead, add the line: GET REFERENCE OF me->description INTO result.

- **BI_OBJECT~EXECUTE_DEFAULT_METHOD:** This method is used in workflow inboxes, such as the Universal Worklist, and workflow administration tools,.

- **BI_OBJECT~RELEASE:** This method is used by the garbage collector when deleting an instance.

- For further details refer to link
https://wiki.scn.sap.com/wiki/pages/viewpage.action?pageId=55566

6.3 Usage of OOPS ABAP attributes in Workflow and task

Ensure that there is at least ONE KEY Attribute which can uniquely represent the instance.

We need to use constructor method to derive instance attributes and class constructor method to derive static attributes. So, while accessing another class objects (methods), attributes are already derived.
When accessing static attributes, always check in buffer first
Refer (https://wiki.scn.sap.com/wiki/display/ABAP/Using+ABAP+OO+attributes+in+workflows+and+tasks) for more explanation about how to use OOPS ABAP attributes in workflow and task.

6.4 Events

Method **SAP_WAPI_CREATE_EVENT** is used to raise event in BOR (Business object repository) approach and CL_SWF_EVT_EVENT in OOPS ABAP approach.
SAP_WAPI_* Function modules should be only used as interface of other applications to access SAP Workflow
For determination / filling of the event container also the class **cl_swf_evt_event** method get_event_container and method set should be used.
CL_SWF_EVT_EVENT class contains RAISE method (for raising an event immediately) and RAISE_IN_UPDATE_TASK(for raising an event in the update task of a Logical Unit of Work).
Follow these steps to raise event in OOPS ABAP:

- Instantiate an empty event container
- Add event parameter name/value to the event container. Avoid overloading container with too many parameters. Only elements which are needed for the workflow should be added. Do not put a whole Business objects within it
- Raise the event passing the prepared event container

Refer
https://wiki.scn.sap.com/wiki/display/ABAP/Raising+ABAP+OO+events+for+workflow
for more explanation about how to raise OOPS ABAP based events with example

Terminating Events of Workflow
Workflows must always terminate when the workflow is no longer relevant, e.g. when the object is deleted or a form is withdrawn.
If only termination of the workflow is required, then the terminating event should be specified in the Basic Data of the workflow header in the Events section of the Version Dependent section, using the “Cancel Workflow” response.

- **Terminating Events:** These are used with asynchronous tasks. For example, a PO have to be send to agent for change and you do not wish to stop the workitem until the PO is actually changed by the agent. Then we will create a asynchronous task with termination event BUS2012, Event: CHANGED. On trigger of this changed event, the workitem completed.
- **Terminating Workflow:** Workflow is terminated using process control step in workflow. We can cancel the workflow using that.

  So according to the explanation above, Process control step can be used to close the workflow.

6.5 Standards

Basic data
Always have the business object key in the workflow item text and within in title of workitem.

Description
At least have the triggering mechanism, e.g. transactions, user exits, reports, change documents, and a basic process overview in the workflow description. When standard workflows are copied, note the SAP template name and abbreviation in the task description.

Triggering Event-Linkage
Make sure that proper binding is maintained to the element used in workflow process

Container Elements
Make sure to create importing parameters for all importing parameters to help with binding with event

Header data
Where appropriate, maintain a workflow administrator in the "Responsible" tab.

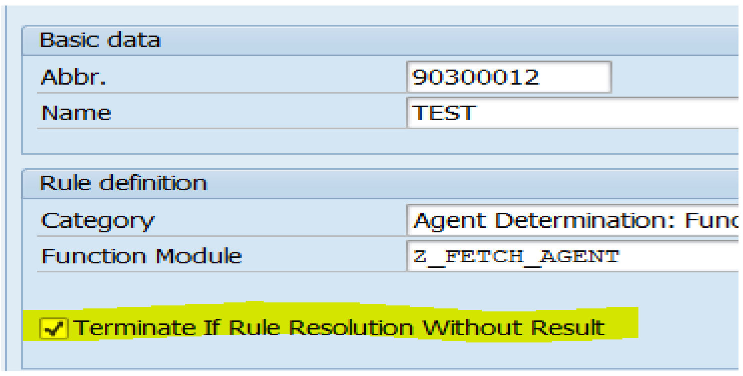
Step data
Always maintain step outcome names – useful for workflow logs.
For user decision steps, remove function module in the "Work Item" display tab.

6.6 Roles

- Roles are the determination of the agents during execution of a workflow task. Role Resolution takes place at runtime, depending on the information from the current process. The result of Role Resolution is a list of agents who are responsible for the actual processing of the task.
- Default terminate if Role Resolution has no result.

6.7 Rule Development

- While developing any Rule to determine the agents of a foreground task. Keep in mind to terminate the workflow if rule resolution fails.


Basic data

Abbr.	90300012
Name	TEST

Rule definition

Category	Agent Determination: Func
Function Module	Z_FETCH_AGENT

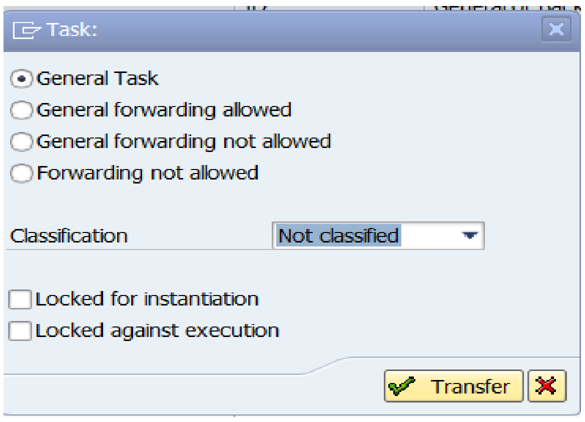
[✓] Terminate If Rule Resolution Without Result

- Use binding between workflow rule to read the workflow container elements instead of reading element values using workitem id.

6.8 Transport

It is recommended that in each target system the workflow test environment (SWUD) and consistency checks are used to compile and check the workflows before use.
In case if any component is missed while transporting a workflow, Tcode RE_RHMOVE30 can be used for report which transports everything that is missing.

The agent resolution definition of a foreground task has to be transported using report 'RHMOVE30'


Task:

[✓ General Task]
General forwarding allowed
General forwarding not allowed
Forwarding not allowed

Classification: Not classified

☐ Locked for instantiation
☐ Locked against execution

6.9 General Recommendations

6.9.1 Enhance standard task

It is not preferable to enhance the standard task. Instead create a copy of standard task and add modifications to it.

6.9.2 Version of workflow

It's preferable to create new version of workflow while enhancing custom/standard workflow. So, we will have separate version and previous modifications are not lost. Workflow has only one active version at a time.
Refer
https://wiki.scn.sap.com/wiki/display/ABAP/Version+Management+In+Workflows for version managements in workflow.

6.9.3 Number range

Number Range used should be between 900 and 999

6.9.4 Miscellaneous

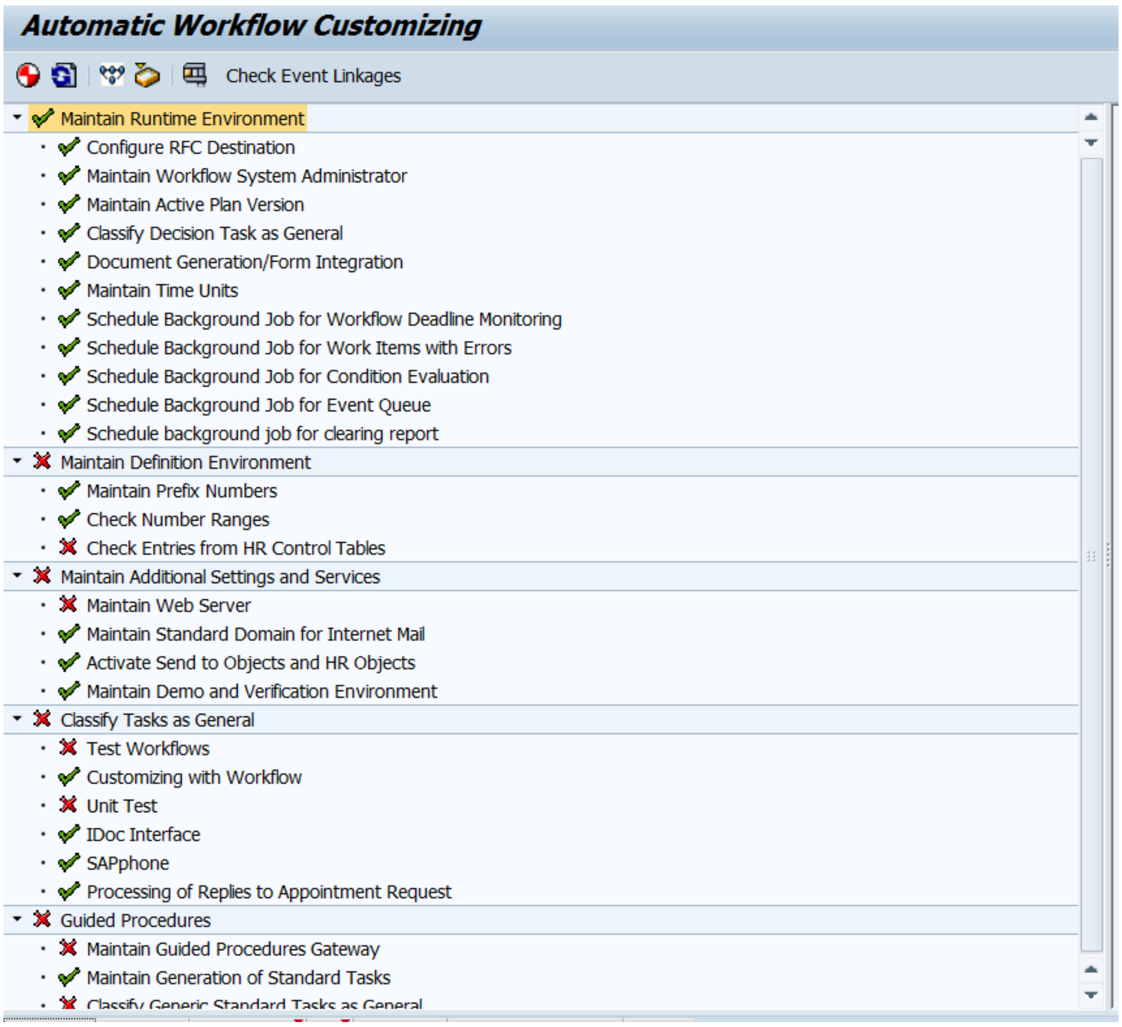
- Minimise use of Z-tables in the overall design of workflow. They should be created only for business use, they should not be a part of workflow technical design.
- Refrain from using function modules for starting workflow; it should be event based design

- Make use of RSWUWFML2 email notification job in workflow
- Incorporate the check of approver should not be the same as initiator in all the dialog steps
- **Exception Handling:**
  - Exception handling should be done wherever required. Exceptions help to easily analyse the issue through workflow log.
  - Exceptions can be raised in Business Object as well Class based workflow.
- **Asynchronous task:**
  - Termination event should be mandatorily defined with appropriate binding when using an asynchronous task.
- **AD-HOC task:**
  - While using the adhoc task functionality, it should be made sure that all the adhoc tasks which are passed in the workflow should have same interface, i.e., container elements, as the base task.
- **Mail Tasks:**
  - Preferably make use of external notifications for workflow mails.
  - In case mail has to be sent from workflow then make use of background tasks. Also, Make use of CL_BCS class to improve formatting of the mail text.
- **Binding:**
  - All the mandatory elements should be bonded from event container to workflow container.
- **Archiving of workitem :**

Reference link for archiving
https://help.sap.com/saphelp_nw70ehp3/helpdata/en/8d/3e704c462a11d189000000e8323d3a/frameset.htm

6.9.5 Check SWU3 Configuration before developing workflows in any client

- RFC Destination 'WORKFLOW_LOCAL_XXX' should be configured.
- User 'WF-BATCH' should be configured with SAP_ALL or equivalent roles.
- All the background reports as specified in the screenshot should be scheduled in backend.



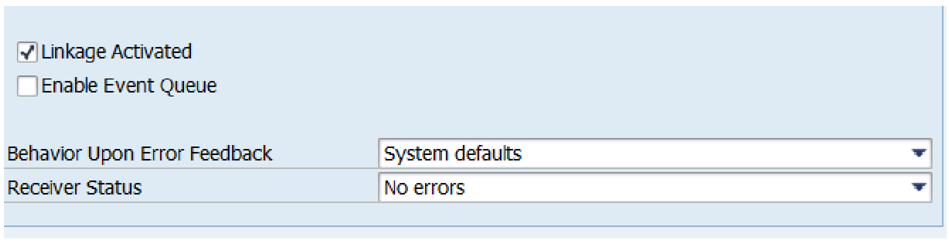
Automatic Workflow Customizing

Check Event Linkages

- ✓ Maintain Runtime Environment
  - ✓ Configure RFC Destination
  - ✓ Maintain Workflow System Administrator
  - ✓ Maintain Active Plan Version
  - ✓ Classify Decision Task as General
  - ✓ Document Generation/Form Integration
  - ✓ Maintain Time Units
  - ✓ Schedule Background Job for Workflow Deadline Monitoring
  - ✓ Schedule Background Job for Work Items with Errors
  - ✓ Schedule Background Job for Condition Evaluation
  - ✓ Schedule Background Job for Event Queue
  - ✓ Schedule background job for clearing report
- ✗ Maintain Definition Environment
  - ✓ Maintain Prefix Numbers
  - ✓ Check Number Ranges
  - ✗ Check Entries from HR Control Tables
- ✗ Maintain Additional Settings and Services
  - ✗ Maintain Web Server
  - ✓ Maintain Standard Domain for Internet Mail
  - ✓ Activate Send to Objects and HR Objects
  - ✓ Maintain Demo and Verification Environment
- ✗ Classify Tasks as General
  - ✗ Test Workflows
  - ✓ Customizing with Workflow
  - ✗ Unit Test
  - ✓ IDoc Interface
  - ✓ SAPphone
  - ✓ Processing of Replies to Appointment Request
- ✗ Guided Procedures
  - ✗ Maintain Guided Procedures Gateway
  - ✓ Maintain Generation of Standard Tasks
  - ✗ Classify Generic Standard Tasks as General

6.10 Object type workflow linkage

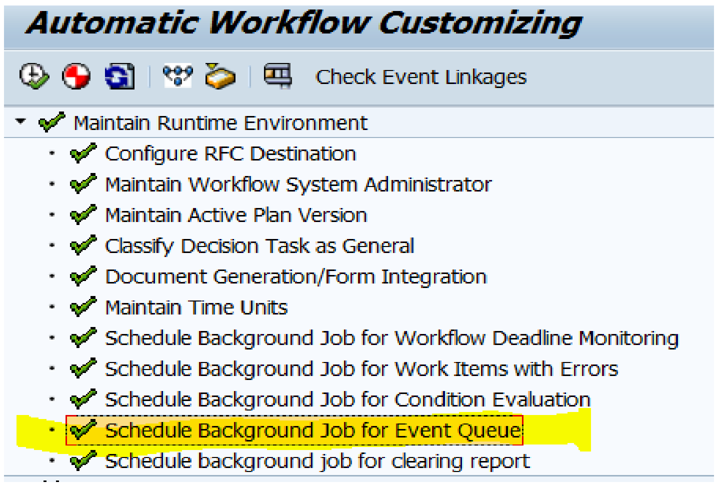
- Linkage should be activated between object type (BO/CL) and workflow template in transaction code ‘SWE2’.
- Behaviour upon Error Feedback should be set as ‘Do not change Linkage’. This is required as upon any error the linkage should not get de-activated.



☑ Linkage Activated
☐ Enable Event Queue

Behavior Upon Error Feedback	System defaults
Receiver Status	No errors

- Event Queue is enabled when large number of receiver events is expected to be triggered in a short amount of time. This is done to reduce the system load as the events are then pushed to the queue instead of being executed immediately. This parameter is however checked if the following option in SWU3 is enabled, which runs a background job at specific intervals.

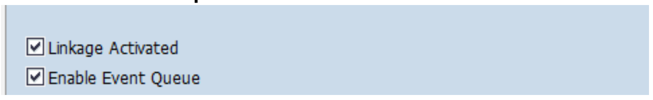


Automatic Workflow Customizing

Check Event Linkages

- Maintain Runtime Environment
  - Configure RFC Destination
  - Maintain Workflow System Administrator
  - Maintain Active Plan Version
  - Classify Decision Task as General
  - Document Generation/Form Integration
  - Maintain Time Units
  - Schedule Background Job for Workflow Deadline Monitoring
  - Schedule Background Job for Work Items with Errors
  - Schedule Background Job for Condition Evaluation
  - Schedule Background Job for Event Queue
  - Schedule background job for clearing report

Enable event queue in this case.



☑ Linkage Activated
☑ Enable Event Queue

6.11 Business Workplace

There are various options available for workflow inbox :

- POWL (Personal object worklist)
  It is normally used with web dynpro/fpm applications. Every Workflow and its relevant task and there parameters along with web dynpro application has to be configured before they start working in POWL. Each POWL has different configurations to be completed.
  For eg. : IBO_INBOX.

- UI5 Inbox
  UI5 Inbox is fully configured using LPD_CUST and SPRO Odata Channel. Every Workflow and its relevant task and there parameters along with web dynpro application has to be configured in the customizing view.

- SAP Business workplace in R/3
  Classical way of dispatching the workitem to the agents workflow INBOX.

- Netweaver Portal (UWL Configuration)
  UWL Configuration (Universal Worklist) is used within Netweaver portal for configuring workflow into portal. These are HTML Files containing workflow task, Web dynpro, Parameter's information

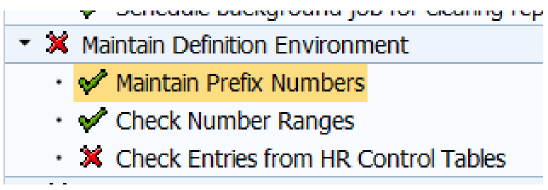
- Custom Application
  Custom Application could be created and workflow can be integrated incase of process specific application and its relevant workitems have to be displayed. Workflow can be integrated into application by using standard function modules.

- For each fo the above Workflow inbox. Below parameters needs to be defined in the respective view :

- Foreground task – The user decision/foreground task defined in the workflow needs to be defined in the view/UWL config.
- Application Name – WD/FPM Application name needs to be defined against the foreground task which will be used to provide information to the agent and record the decision.
- Parameters – The parameters which are need to instantiate the web dynpro / fpm application have to be defined at the workflow task level and then passed to the WD/FPM.

6.12 Naming Conventions

- Workflow task number are auto generated but the prefix of workflow and tasks can be configured in the Transaction SWU3.


Maintain Definition Environment
Maintain Prefix Numbers
Check Number Ranges
Check Entries from HR Control Tables

- Starting events are configured within the class/BO. All the naming conventions specified for class/BO should be followed.

  Abbreviation should be meaningful and contain <Project_Namespace>_<Process>_<Description>
  There is no standard restriction as it is a plain text

7 ADOBE Forms

For details guidelines for developing ADOBE forms please follow SAP recommendation available on 'SAP Interactive Forms by Adobe'.

7.1 Adobe form development using FLM tool

Logical Form design:

Form Type Code

This is a four-character code. The first two letters define the application area while the second two denote the form's function. Each form code should accurately describe the form's application area and function. This will be helpful, for example when managing form routings.

	Element	Value	Comments
1	Prefix	2 character application area abbreviation	e.g. OM (organisational management)
2	Identifier	2 character abbreviation to describe forms functions (note, this may be a number if the form has previously been known by a number, e.g. FN10	e.g. 10, LV (leavers)
	Examples	OMLV FN11	OM Leavers form FIN11 (raise sales order or credit note)

Version

This should start at 00 and increment by 1 with every live release of the form.

Sub-form Catalogue

This is a logical or physical grouping of fields that defines their characteristics on the form. The following sub-form names should be used in the sub-form catalogue:

Sub-form	Mandatory / Optional	Parent	Max Occurs
ROOT	Mandatory		0
HEADER	Mandatory	ROOT	1
FOOTER	Mandatory	ROOT	1
MAIN	Mandatory	ROOT	1
ITEMS	Optional	MAIN	(depends)

Ideally, there should be no more than 3 levels of depth to the sub-forms, i.e. no further than MAIN, HEADER, ITEMS.

Field Catalogue

The field catalogue contains the list of fields that are to be used in the form. Field names must consist of alphanumeric characters only, underscores are allowed but spaces are not. Capital letters should be used for field names.

Element	Value
Prefix	Single capital letter character to denote the sub-form that the field will appear in.
Separator	_ (underscore)
Identifier	Capital letter words to describe the field. If more than one word, separate by underscore.

XML File Location

All XML files created by the FLM Form Wizard must be stored in the fixed location on the drive.

Form Categorises

This is a 2-character category code plus a description. Every form must reside within one of these codes, and they assist the user in selecting the correct form.

Form Categories:

Category Code	Category Code Description	Comments
OM	Organisational Management	Used for all SAP 'OM' forms
FN	Finance	Used for all SAP 'FIN' forms
PR	Procurement	Used for all Procurement / Supplier Relationship Management forms
EP	Estates Planning & Management	Used for all EPM forms
PA	Personnel Administration	Used for all PA forms

7.2 Posting Adaptors

This is the function module that applies the business logic to the business application, i.e. the wrapper for the BAPIs and function modules that will update the transactional system.

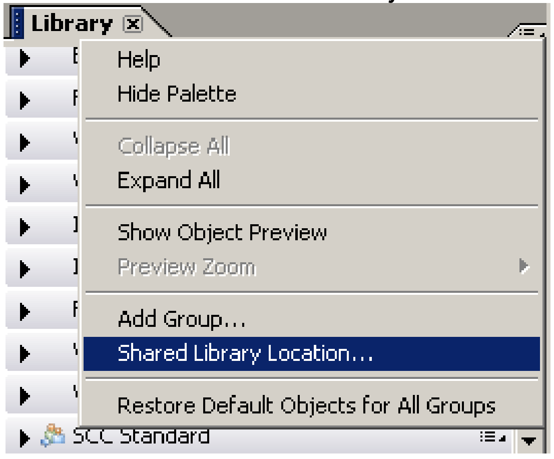
Posting Adaptors should be named as follows:

Element	Value	Comments
Prefix 1	Z	To denote customer development.
Separator 1	_ (underscore)	
Prefix 2	Form type	e.g. FN09, OMLV
Separator 2	_ (underscore)	
Description	Brief description of form purpose	e.g. LEAVERS
Suffix	Lowest compatible BASIS release	e.g. 470, 700
Examples	Z_OMLV_LEAVERS_700 Z_FN09_RAISEORDER_470	

7.3 General Adobe forms design / Physical Design

Font
In general the standard font is Arial 12 point. Arial 8 point may be used in the footer. Shared palette can be created, which has the controls set as Arial 12 point, rather than the default of 10 point and can be used. Fonts and size to be used across different forms should be aligned with business before start of development.
To use this palette:

Choose 'Shared Library Location' from the 'Library' palette:



Object Names:
Objects on the form should be named according to their object type and the field that they represent.

	Element	Value	Comments
1	Prefix	Abbreviation of the type of object.	e.g. txt (for text box), ddl (for drop down list)
2	Identifier	Representation of the field name	e.g. LastName
	Examples	txtLastName ddlCountry	

The following table shows the abbreviation for each object:

Abbrev.	Object Type	Abbrev.	Object Type
But	Button	Num	Numeric field
Chk	Check box	Pw	Password field
Dt	Date/Time	Rad	Radio button
Ddl	Drop-down list	Sub	Sub-form
Img	Image	Txt	Text field
Dec	Decimal field	Lst	List box
Sig	Signature field	Tab	Table
Lbl	Text (Label)		

Denoting Mandatory fields

Mandatory fields should be marked with a red star (RGB 255, 0, 0). A key to the meaning of the star should be displayed at the top of the form:
* = required information
If, on form submission, a user fails to complete a mandatory field, the field caption should change from black to orange (RGB 238, 118, 0) and the field background should change from white to light orange (RGB 253, 254, 230) with a border the same colour as the caption text.

7.4 Performance Aspects

The following information is targeted mainly at developers, who can use it to achieve an optimal level of runtime performance when using Interactive Forms based on Adobe software. Unless specified otherwise, these recommendations apply both to interactive forms in Web Dynpro and to PDF-based print forms.

Recommendation for Form Builder

For each business scenario design separate form that uses its own data retrieval methods. Do not implement condition-driven output of multiple forms from the same from object for the better performance.
Structure the form context in accordance with the logical flow of the data. Place the context nodes used at the top of the form at the initial position in the context tree.
Deactivate any context nodes that you do not need.

Recommendations for calling forms and form output

- When you create forms for mass printing, design the data retrieval process in the application program and interface in accordance with your individual requirements. To support you in your work use the function module FP_FIELD_LIST, which determines all the fields used in the interface.
- Bundle print forms (Cache forms that you want to process with Adobe document services)

Recommendations for Adobe Lifecycle Designer

For interactive forms use form caching function in the designer.
Minimise the amount of communication that takes place with the server in interactive form. Only use those interactive functions (such as dynamic value help that sends queries to the server) that are strictly necessary.
As far as possible keep scripting to the minimum
If possible use only following character sets, even if the designer supports others: Courier, Arial, Times Roman. These character sets do not need to imbed in the form, which improves performance.

Creating forms for integration with Web Dynpro and ZCI

Forms which are created in form builder for Web Dynpro application should have XML based interface and form layout as “ZCI Layout”. You can use transaction SFP_ZCI_UPDATE to make any interactive forms you created with an older version of SAP Net Weaver ZCI-Compliant. For this prerequisites are

1. You use SAP Net Weaver 7.0 SPS 8 or higher
2. You use SAPGUI release 6.40 support package levels 20 or higher.
3. You use Adobe Life Cycle Designer 8.0

a. If you want to update forms from ISR framework to ZCI, you must first run report FP_CHECK_REPORT with the ISR object and update only those forms which are active and are not being edited.

7.5 Program-Driven Generation of Form Templates

You can use all interfaces from the package SAXFT.
Each interface corresponds to precisely one tag in the XFA specification.
You generate the root node by using CREATE_OBJECT to instantiate an object from the class CL_SXFT_TEMPLATE. For this object, you can call the method GET_FACTORY( ) to create a factory object. You can generate all subsequent XFA elements by using a “create” method of the factory method to instantiate the corresponding object. You create the structure of the XFA document as follows:

- You use “set” methods to set simple attributes and to send the attribute value to the VALUE parameter in quotation marks.
- You use “set” methods to set attributes that are themselves elements, and to send an object to the VALUE parameter that represents the attribute (and the new element).
- You use the method IF_SXFT_NODE~APPEND_CHILD(NEW_CHILD) to set child elements. The interface IF_SXFT_NODE is included in all interfaces that represent XFA elements.

Grouping interfaces are used for attributes that are used very frequently

- The interface IF_SXFT_ALIGNMENT groups the "set" methods for the attributes VALIGN and HALIGN. This interface is included in all interfaces that represent elements with these attributes.
- The interface IF_SXFT_MEASUREMENT groups the "set" methods for the attributes X, Y, W, H, and *Layout*. This interface is included in all interfaces that represent elements with these attributes.

General Note:

- It is recommended that Program-Driven generation be used only for simple form templates. Complex form design should be handled by form builder (transaction SFP) only.
- Do not create multiple forms for different languages for same business requirement. Example: There should be one form for same requirement for languages German and English.

7.6 Handling errors and problems

Interactive Forms based on Adobe software provide you with comprehensive support for detecting and resolving errors and problems when you create, display and print forms.

Utilities in the Form Builder:

The Form Builder that you use to create PDF-based print forms and interactive forms in Web Dynpro for ABAP offers you a wide range of functions for:

- Checking and testing the PDF based print form
- Saving Runtime information and generated PDF locally.

You activate this function in the Form Builder. When you call the form, it gives you detailed information about the form data and runtime information specified by Adobe document services.

Utilities in the Application Program that Calls the Form:

The following function modules provide you with support when you work with PDF-based print forms:

- Getting the Used Interface Fields
  - *Function Module: FP_FIELD_LIST*
- Troubleshooting for Runtime Errors FPRUNX001, 002 and 004
  - *Function Module: FP_GET_LAST_ADS_ERRSTR and FP_GET_LAST_ADS_TRACE*

You use these function modules to find out the causes of any errors you encounter when displaying forms.

Communication Errors, Runtime Errors, No Form Output:

If you encounter any connection errors or configuration errors when you use Adobe document services, see the description under Problem Analysis Scenarios for Adobe Document Services for detailed troubleshooting information.

8 Security and compliance

The general guidelines described in the following are to be adhered to during ABAP development in order to ensure compliance with legal requirements and ALDI-internal security requirements.

Ensuring auditability of the source code: It must be possible to analyse the customised ABAP source code to identify deficiencies by means of manual examination or static code analysis tools at any time. Therefore, any means of making ABAP coding invisible are not permissible (i.e. code must not be hidden from debuggers). Also Editor Lock should not be applied.

Programs that can be started online may be allocated a transaction code if specified. This must be then used in an authorisation object to determine who is allowed to call the program.

If specified, programs can also be allocated an authorisation group in the attributes. This authorisation group can then be assigned to an authorisation profile and will ensure that only authorised users will be able to start or edit the program.

There are also SAP authorisation objects for running batch input programs and background jobs. In those cases the authorisation can be given to a so-called background user. In the case of batch input this user will also require all the necessary authorisations for running the transaction.

If direct database updates are performed using commands like insert, delete, update, modify, etc. (e.g. mass changes to master data), then the developer should perform authorisation check using special roles created by the authorisation team.

8.1 Authorisation check

In the case of developments within the ALDI namespace, authorisation checks have to be included in all programs to protect data from the unauthorised users and to prevent users from executing programs without the relevant permissions.

For this purpose, authorisation checks using standard SAP authorisation objects or customer-specific authorisation objects can be developed in cooperation with the authorisation team following ALDI SAP Authorisation concepts.

Either the ‘AUTHORITY-CHECK’ ABAP command or the ‘AUTHORITY-CHECK’ function module can be used for an authorisation check.

For access to tables, a check for editing or viewing permissions is to be conducted using the ‘S_TABU_NAM’ authorisation object. If transactions can be called up using the ‘CALL TRANSACTION’ command within a program, the ‘WITH AUTHORITY-CHECK’ additional specification must be applied since the SAP system will otherwise not perform an authorisation check for this transaction.

The developer has to take notice of the following rules:

1. An authorisation check must be implemented in all executable development objects (e.g. reports, posting functions, critical table maintenance dialogs, SAP-queries, etc.). Before writing the statement 'authority-check' the developer has to identify which critical data have to be protected. Depending on these data the fields of the authorisation object are to be defined. In most cases, the field 'ACTVT' and the organisation level fields like Company Code (BUKRS), Sales Organisation (VKORG), Plant (WERKS) or functional fields like Order Type (AUFART) and so on are such fields which are to be checked. In general, it has to be ensured that the authority check is always executed for every user. Addition ‘FOR USER’ must or hard coding the username for which the authority check will be executed is strictly prohibited.

2. It is recommended to use the ‘CALL TRANSACTION’ command instead of ‘SUBMIT’ command for programs which have Transaction codes (T-Code) attached. The calling

program has to check the S_TCODE authorisation for the called program. Prefer function module 'AUTHORITY_CHECK_TCODE' to execute this necessary check.

3. The assignment of the authorisation objects to a transaction has to be maintained using transaction SU24. This assignment facilitates role development as the required authorisation objects will be automatically entered in the authorisation profile when adding a transaction code to a role. Thus this SU24-Maintenance is a must-have. The developer is responsible for this maintenance as only this developer knows what is checked in the coding he has developed.

   Remark: SU24-Maintenance is a must-have. Not only the authorisation objects which are directly checked in the program, but also such authorisation objects which are indirectly checked in called programs (e.g. called function modules which have an own authority check) have to be maintained.
4. Developer should be aware of any regulation that concerns authorisation and the accessibility of data to any user. Like Group Regulation and IT Security Plan. All applying access and authorisation concepts must be adhered to.
5. Make sure your coding (e.g. reports) does not corrupt the SAP authorisation concept. If SAP standard authorisation object don't match the given access to data customer, new authorisation objects must be created and used instead.
6. Front end reports should be assigned authorisation only via transaction codes.
7. The SAP Authorization concept can be referred to at the link.
8. Usage of generic placeholder values for authorization restrictions are not recommended for example &NC& as authorization group in TMG attributes etc.

8.2 Programming

- Authorisation checks must also be implemented in custom developments:
  - Using appropriate authorisation objects
  - Implementing authorisation checks in accordance with the SAP standard
  - Processing the return values of the authorisation checks
  - Having the applicability of the authorisation objects tested by the relevant technical department
  - Using SAP standard function modules/methods since these already include authorisation checks in most cases. Avoiding direct read and update functions

- Client (SAP system client) separation must also be implemented in custom developments:
  - Data may only be processed on clients on which the executing user is logged in.
  - Having the applicability of the clients tested by the relevant technical department
  - In general, Customer-specific tables should be created for specific clients, not for all clients.
- In general, operating system commands must not be used via programs (ABAP) in SAP. Exceptions must be discussed with both the affected technical department (e.g. SSV, AD, DB) and the Security team.

- In order to avoid security risks, dynamic programming may only be used in exceptional cases that have been agreed with the IIT Security team and approved by project management.
- The use of technology for rendering source code invisible is strictly prohibited.
- Only the Open SQL standard commands provided by SAP may be used for database queries. Native SQL commands are not permissible. If a query using native SQL commands is necessary in individual cases, this must be agreed with project management and the IT Security team. The described situation is to be considered an exception and must thus be documented outside of SAP.

8.3 Monitoring and control

- If critical SAP notifications are identified during development work, project management is to be informed accordingly even if the notification is not directly linked to the resolution of a development-specific problem.
- Traceability must be ensured at all times/all changes to tables must be logged for the executing user.

9 Performance Guidelines

SAP performance guidelines can be referred inside the ABAP documentation that can be called from within the ABAP editor by hitting F1 on a statement. In the search field, enter Performance and have a look at the results.

The general guidelines described in the following are to be considered in the context of ABAP development in order to prevent performance bottlenecks from the beginning.

- Avoiding unnecessary coding/functions/data
    - Lean coding of live applications
    - Explicit retrieval of required data
    - No code duplicates; encapsulation in function modules and methods instead. Methods are recommended.

- Performance analysis
  1. Identifying the greatest time drivers in ST12/SAT
  2. Detailed performance analysis:
     a. Performance bottlenecks caused by ABAP: ST12/SAT
     b. Performance bottlenecks caused by database queries: SQL Trace ST05

9.1 Database access

- Normalisation of the underlying data model
    - While designing custom data applications, care should be taken that the new custom table or structures have proper keys, table relationships and indexes defined to avoid data redundancy. This is important for better performance during data retrieval and updates.

- Efficient use of indices:

- 'Client' field as the first field of an index in the case of client-specific tables
- A maximum of five indices in case of frequent data access for modification purposes
- A maximum of five selective fields per index, sorted in descending order by selectivity
- Database access only via the index in the case of large tables (sequential database access not permissible)

- Reducing the amount of data that is transferred to the application layer to a minimum:
  - Avoid using Select *. It should be used only if necessary. Try to give the list of fields which are needed. It should be entered in the order that they reside on the database.
  - Limit selections to the required rows with use of proper WHERE clause
    - Use Key Fields in Where Clause for row selection. If complete Key is entered, use 'SELECT SINGLE'
    - only for non-HANA systems, if key fields are not available try to use Secondary Index fields

  Note that effectiveness of this (Key Fields of Index Fields) will increase with value entered for maximum number of uninterrupted sequence of key/index fields, starting from 1st key/index field, excluding Client.

  - Comprehensive use of existing indices for retrieval purposes
  - Performing existence checks using 'SELECT SINGLE' for fields in the selection index
  - Do not use 'SELECT....ENDSELECT' command. Instead, the records should be selected in an internal table and processing logic should be applied in the LOOP-ENDLOOP.
  - Try to avoid multiple select queries by collecting all the records in internal table using single select Query and getting data from internal table in further logic
  - Never assume that SELECT statements return data in key sorted order. A "SORT" statement shall be used when data is required to be sorted.
  - All hard coding used for SELECT statements shall be included as parameters on selection screens that shall be used with variants. When this is not possible, constants that are easily identifiable in the data declaration section shall be used.
  - It is advisable not to hard code anything in the code or declare constants if it is likely to be changed in future. Instead the parameter table TVARC should be used or customised table to be built where found applicable
  - In the case of large tables, inner joins may only be used across index fields.
  - Use a select list with aggregate functions instead of checking and computing, when try to find the maximum, minimum, sum and average value or the count of a database column, thus network load is considerably less.
  - ORDER BY statement is used in SELECT only if it can use an index else sorting is effective by reading into an internal table and use the SORT statement in the program.

- Updating individual fields is preferable to updating entire data records.
- Using array access statements for a consolidated retrieval of the required data instead of individual queries
- Avoid Open SQL statements within loops and nested 'SELECTs', instead collect data in an internal table and use this for performing the operation.
- Use 'FOR ALL ENTRIES IN tab' instead of Select statement inside a Loop. Make sure that the table used for "For All Entries" is not empty when using this additional specification.
- VIEW or JOIN should be used to replace nested SELECT statement, thus network load will be considerably less. If the nested SELECT is used the inner select statement is executed several times which might be an overhead.
- Always use maximum possible filters during the data selection to avoid unnecessary data getting included in the dataset. This will ensure faster data retrieval and further processing.

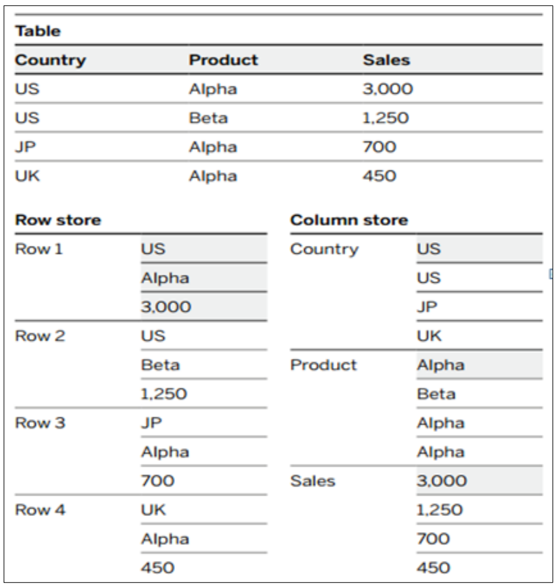
9.2 Internal tables and references

- Based on the size of the data to be processed using internal table it is recommended to use the appropriate internal table category for optimal performance. Refer the SAP help on **'Selecting the Table Category'** for more details.
- Using keys that match the relevant table type, using secondary keys of a different type for varied access to the same data
- Defining specific sort fields for sorting and deleting adjoining duplicates
- Internal tables must always be sorted and searched using the 'BINARY SEARCH' additional specification. Note – This is implicitly handled for Sorted tables.
- Sort statements should not be used inside loop…endloop for the same internal table.
- In the case of large tables, memory requirements are to be assessed and the preselection is to be enhanced if necessary.
- In general, Internal tables must not be copied within the program if it contains large of data (requires a large amount of time and main memory), access via references
- Field symbols should be used over work areas for any type of access to internal tables. This also includes access such as loop..endloop and read statements.
- Refreshing internal tables if they are no longer used via the corresponding statement. Thus, it is ensured that they contain only up-to-date information if they are used once more.
- In general, always pass internal tables 'as reference' instead of 'as values' in parameters.

10 ABAP ON HANA

SAP HANA is different by design as compared to classic or common relational database systems. It stores all data in-memory, in columnar format as well as row format and compressed. Due this structure sums, materialised views and aggregates are not required. This makes HANA very fast and helps to reduce the database footprint. Everything is calculated on-demand, on the fly, in main memory. This makes it possible for companies to run OLTP and analytics applications on the same instance at the same time, and to allow for any type of real-time, ad hoc queries and analyses.
SAP HANA by design supports columnar storage of data and hence we don't require a secondary index, as this reduces additional load on the database. Secondary index are

required for some cases where there is huge CPU consumption for highly selective queries on very large non-primary key fields. Please refer SAP Note 1794297- Secondary Indexes for the business suite on HANA. Conceptually, a database table is two-dimensional data structure with cells organised in rows and columns. Computer memory, however, is organised as a linear sequence. For storing a table in linear memory, two options can be chosen, as shown in below figure. Row storage stores a sequence of records that contain the fields of one row in the table. In the column store, the entries of the column are store in the contiguous memory locations.



Table

Country	Product	Sales
US	Alpha	3,000
US	Beta	1,250
JP	Alpha	700
UK	Alpha	450

Row store

Row 1	US
	Alpha
	3,000
Row 2	US
	Beta
	1,250
Row 3	JP
	Alpha
	700
Row 4	UK
	Alpha
	450

Column store

Country	US
	US
	JP
	UK
Product	Alpha
	Beta
	Alpha
	Alpha
Sales	3,000
	1,250
	700
	450

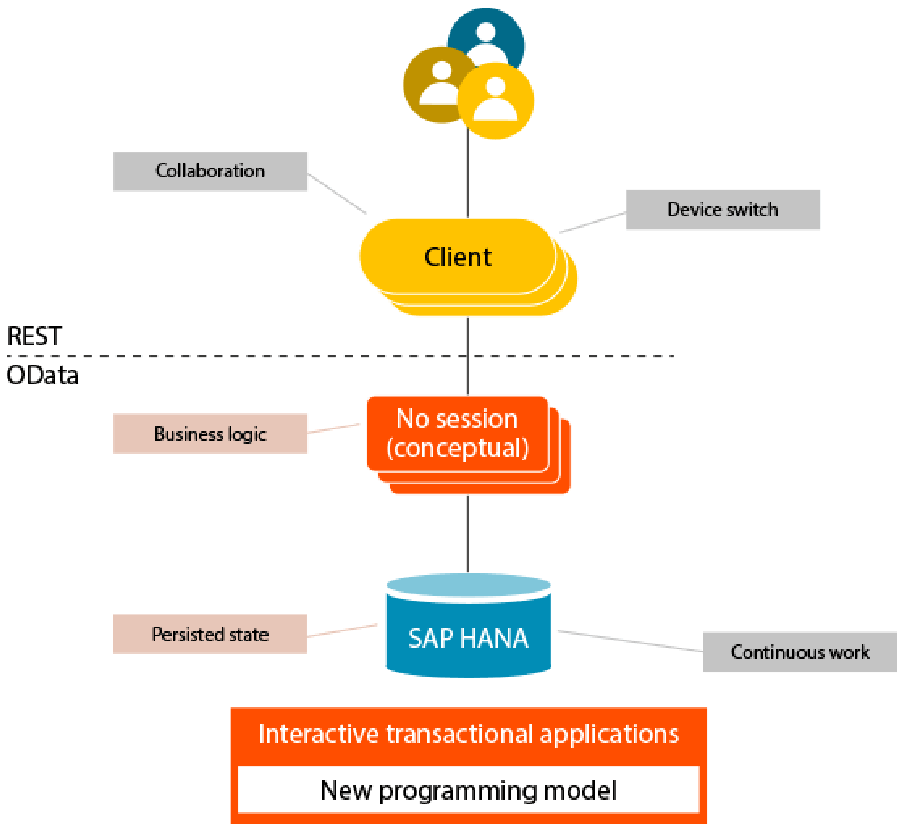
10.1 RESTful Architecture:

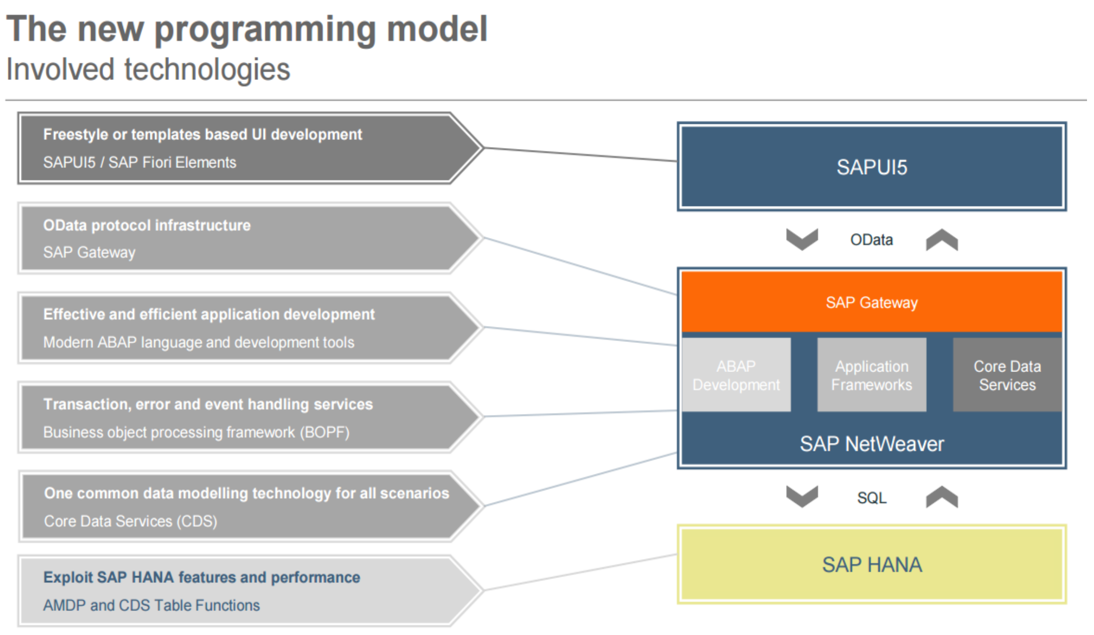
To enable collaboration and seamless shifting between devices while preserving work continuity features that modern users depend on a REST based architecture is required. With a REST-based architecture, you achieve more loosely coupled layers that interact and interface in a stateless fashion using REST-based protocols based on HTTP, such as the OData protocol.

In the traditional Dynpro and Web Dynpro world, the application state is kept in the back-end session residing on the application server (in the well-known roll area), which disables device switch and continuity of work. Sessions simply time out if not continued on that server. In simple SAP Fiori applications, traditional logic is carried out in a "batch-input" fashion, meaning input is gathered in several steps from the end user and then written against the database. The state of the application is held on the client level, which does not allow for continuous work or collaborative scenarios.

The new ABAP programming model follows a REST-based approach. The application state is persisted in SAP HANA at the database level, and no session is present on the application server level other than the business logic, which is executed in the current step (below Figure). This approach allows users to switch devices and to collaborate.

For performance reasons, not all session states are persisted — only the essential state is saved, which consists of changed original data, interaction data (such as error status), undo/redo history, and acquired locks, for instance. All the session data that can be easily reconstructed on the fly (such as value help) is reconstructed when the application is resumed. The program model also shields all technical session context (such as logon data, authentication, and established database connection) implicitly.




The new programming model
Involved technologies
Freestyle or templates based UI development
SAPUI5 / SAP Fiori Elements
OData protocol infrastructure
SAP Gateway
Effective and efficient application development
Modern ABAP language and development tools
Transaction, error and event handling services
Business object processing framework (BOPF)
One common data modelling technology for all scenarios
Core Data Services (CDS)
Exploit SAP HANA features and performance
AMDP and CDS Table Functions
SAPUI5
OData
SAP Gateway
ABAP Development
Application Frameworks
Core Data Services
SAP NetWeaver
SQL
SAP HANA

Refer- https://assets.cdn.sap.com/sapcom/docs/2017/03/867e02d4-ac7c-0010-82c7-eda71af511fa.pdf for more details about RESTful architecture

10.2 Choosing between column and row store

With SAP HANA you can specify whether a table is to be stored by column or by row.

Row Store is recommended if:

- The table has small number of rows, such as configuration tables.

- The application needs to process only a single record at a time (many selects or updates of single records)
- The application typically needs to access the complete record.
- The columns contain mainly distinct values so the compression rate would be low.
- Aggregation and fast searching are not required.

Row store is used, for example, for SAP HANA database metadata, for application server internal data such as ABAP* server system tables and for configuration data. In addition, application developers may decide to put application tables in row store if the criteria given above are matched.

Column Stored is recommended if

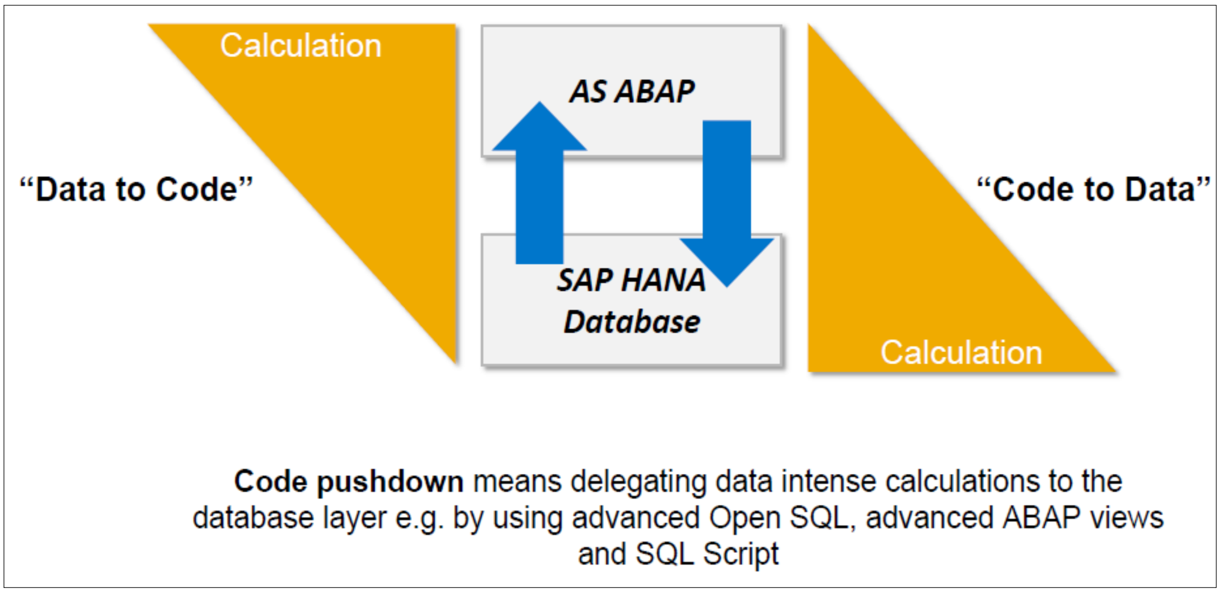
- Calculations are executed in single column or few columns only.
- The table is searched based on the values of a few columns.
- The table has large number of columns.
- The table has large number of rows and columnar operations are required (aggregate, scan and so on).
- The majority of columns contains only a few distinct values resulting higher comparison rates

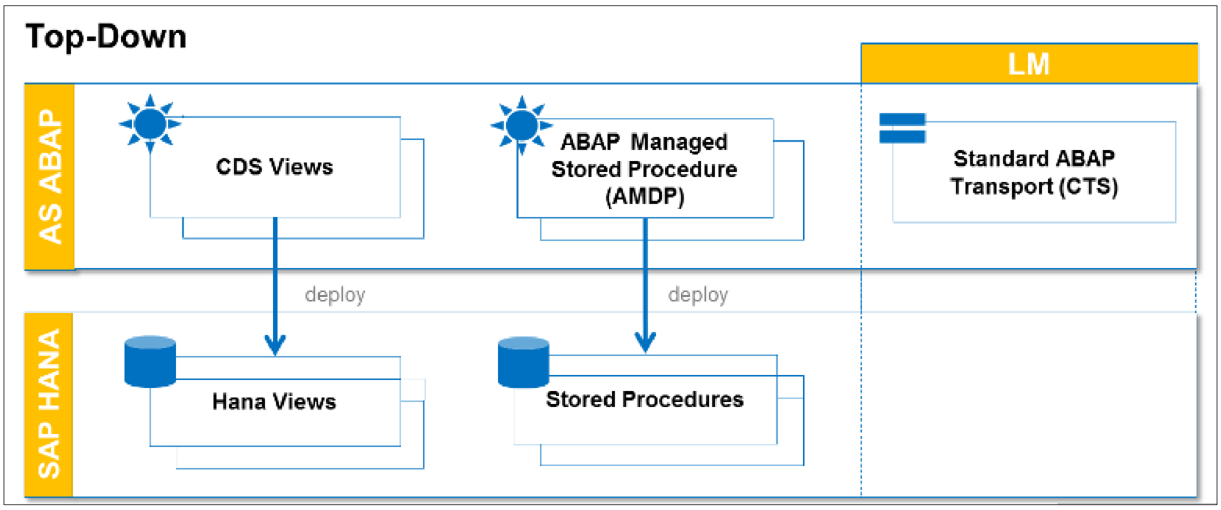
Advantages of Columnar Tables: when the above criteria is fulfilled columnar tables have several advantages

- Higher performance for column operations
- Higher data compression rates
- Elimination of additional indexes
- Parallelisation
- Higher performance of column operations

10.3 SAP HANA Code pushdown techniques

Code pushdown means delegating data intense calculations to the database layer. It does not mean push ALL calculations to the database, but only those that make sense (For example:-Calculations which will take long time to execute on large dataset should be performed at database level and the rest of operations on application server to optimise the performance. An easy example is if you want to calculate the amount of all positions of invoices. You should not select all positions of those invoices and calculate the sum in a loop. This can be easily done by using an aggregation function (here SUM ()) on the database.


Calculation
"Data to Code"
AS ABAP
SAP HANA Database
"Code to Data"
Calculation
Code pushdown means delegating data intense calculations to the database layer e.g. by using advanced Open SQL, advanced ABAP views and SQL Script



The SP5 release takes the next step to enabling code pushdown with ABAP for SAP HANA: The "Top-Down approach". It enables developers to continue working in their ABAP environment and still have similar possibilities to leverage the power of SAP HANA.

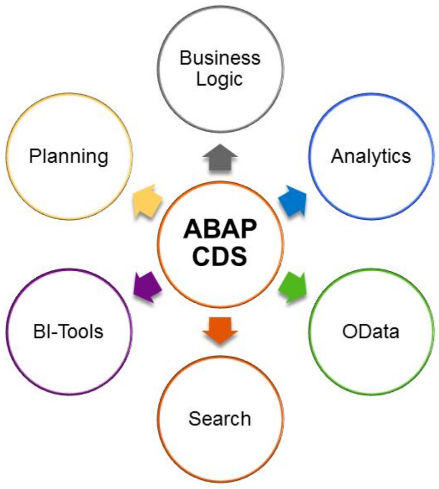
Refer https://blogs.sap.com/2014/02/03/abap-for-hana-code-push-down/ link for more details about SAP HANA Code Push Down techniques.

10.4 Code Pushdown can be achieved by using:

- **ABAP Core database Service Views (CDS) :** Core Data Services (CDS) is an enhancement of SQL which allows a simple and harmonised way for the definition and consumption of semantically rich data models–– independent of the consumption technology. CDS artefacts are design-time definitions that are used to generate the corresponding run-time objects, when the CDS document that contains the artefact definitions is activated in the SAP HANA repository. To define an ABAP CDS View via the necessary DDL Source File, it is necessary to do this via Eclipse with ADT.

Overview of CDS view capabilities:

- **New built-in SQL functions** provided for conversion, string, date and time
- **Currency and unit conversion functions**
- **New domain-specific annotations** provided for areas such as OData, BW and Search
- **Associations** defined in the basis view can now be used **in CDS extensions.**
- **Extension of CDS view**
- Definition and consumption of **declarative authorisations** based on PFCG data using CDS Data Control Language (DCL) roles.
- CDS Table Functionsfor allowing the use of natively implemented database functions in SAP HANA directly from CDS (HANA breakout scenarios).
- CDS view with **input parameters** are now supported **on all databases**; meaning fallback no longer required.



Performance recomandenation for CDS View:

- Attributes, which are able to filter the result set best are processed as deep (=as soon) as possible in the CDS View layers. This helps to avoid fetching data from tables during the processing, which is filtered out on a higher layer anyway.
- Do not use the OR operator or <> within the ON condition of a join as this most of the time results in bad performance.
- The virtual data model should be built using CDS Views. The views have to be separated into different types – they have to be structured according to thei visibility and content via CDS Annotations. The concept and view types are described in here: https://blogs.sap.com/2016/08/30/the-semantically-rich-data-model-an-abap-based-cds-views-example/
- Each and every access to the database should be executed over the virtual data model (= CDS Views).
- CDS Annotations should be used in order to support the automatic UI generation of fiori elements.
- CDS Views, which should be reused by several views should be generally low in complexity.

Refer - https://assets.cdn.sap.com/sapcom/docs/2016/02/bedb9729-5f7c-0010-82c7-eda71af511fa.pdf for more detials about CDS view.

Refer -
https://help.sap.com/viewer/cc0c305d2fab47bd808adcad3ca7ee9d/7.51.3/en-US/630ce9b386b84e80bfade96779fbaeec.html
for CDS annonations

Refer - https://help.sap.com/viewer/f2e545608079437ab165c105649b89db/7.51.3/en-US/4ed2245c6e391014adc9fffe4e204223.html for creation of CDS view.

Refer https://help.sap.com/http.svc/rc/abapdocu_751_index_htm/7.51/en-US/index.htm?file=abennews-751-abap_cds.htm to view new features of CDS view which are available in Release 7.51.
https://help.sap.com/http.svc/rc/abapdocu_751_index_htm/7.51/en-US/index.htm?file=abennews-751-abap_cds.htm

- **ABAP Managed stored procedure (AMDP):** ABAP Managed Database Procedure (AMDP) is a class-based framework for managing and calling database procedure in ABAP. This is implemented as a method of a global class marked with IF_AMDP_MARKER_HDB interface, corresponding database procedure is created at the first call of the AMDP method. The advantage is same we can transport only AMDP class to the next level (Supported by CTS and transport management) Usage of AMDPs binds the application to the SAP HANA database. Best practise to avoid AMDP, if not really necessary as HANA has problems with parallelization at the execution of these methods, especially if they contain control structures and loops. So, preferably, we need to use ABAP CDS, Open SQL instead of AMDP or use an AMDP along with CDS Table Function instead of plain AMDP which can be join with CDS view.

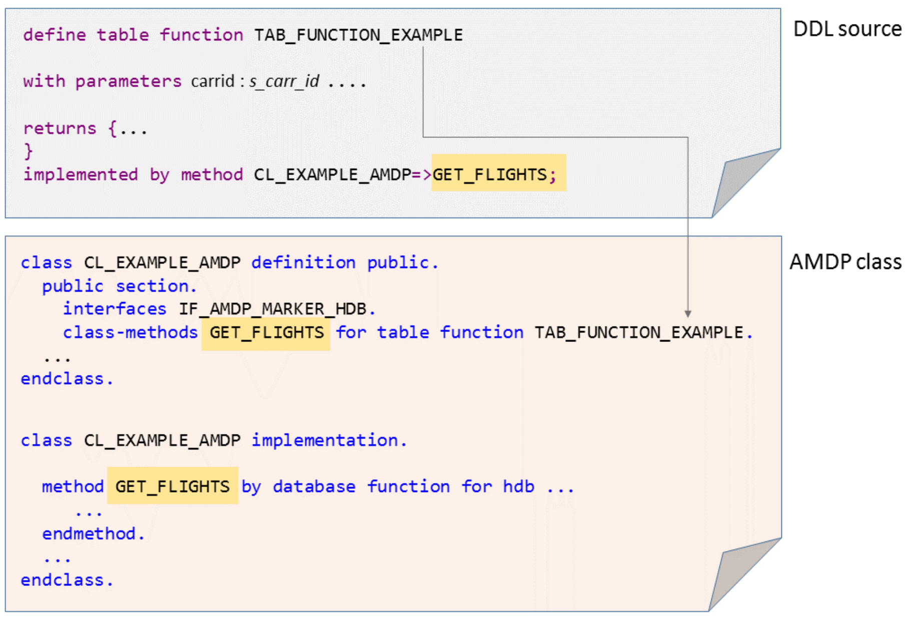
  Refer https://help.sap.com/viewer/6811c09434084fd1bc4f40e66913ce11/7.51.3/en-US/3e7ce62892d243eca44499d3f5a54bff.html to more detail about AMDP procedures.

- **CDS Table functions:** CDS table functions define table functions that are implemented natively on the database and can be called in CDS. In contrast to the CDS views, the CDS table functions can be implemented using SQL script. This implementation is done within an AMDP method of an AMDP class and is managed as an AMDP function by the AMDP framework in the database system.

  As a best practice to use CDS view for implementing business logic. In case some business logic must be implemented and pushed to the database, which is not possible with CDS Views than we need to implement CDS table functionality along with an AMDP method.

  CDS table function includes the following components:

  - The actual CDS entity of the table function that is generated in the ABAP Dictionary
  - The CDS table function implementation (ABAP class library)



```
define table function TAB_FUNCTION_EXAMPLE

with parameters carrid : s_carr_id ....

returns {...
}
implemented by method CL_EXAMPLE_AMDP=>GET_FLIGHTS;
```

```
class CL_EXAMPLE_AMDP definition public.
  public section.
    interfaces IF_AMDP_MARKER_HDB.
    class-methods GET_FLIGHTS for table function TAB_FUNCTION_EXAMPLE.
  ...
endclass.


class CL_EXAMPLE_AMDP implementation.

  method GET_FLIGHTS by database function for hdb ...
     ...
  endmethod.
  ...
endclass.
```

Refer 'https://help.sap.com/saphelp_nw75/helpdata/en/e5/529f75afbc43e7803b30346a56f963/content.htm' for more details about CDS table function.

Refer 'https://help.sap.com/viewer/de2486ee947e43e684d39702027f8a94/2.0.01/en-US/297af2926307446cbbfb1a8f96fec941.html 'link to more details about SQL Script.

- **Native SQL:** Native SQL allows you to use all database-specific SQL statements in an ABAP program. In general ABAP CDS views are sufficient for data access and usage of Native SQL is restricted. Native SQL must be used only after approval form technical lead. Native SQL statements embedded between EXEC SQL and ENDEXEC do not fall within the scope of ABAP and do not follow ABAP syntax.In Native SQL, Developers needs to be taken care of passing parameters/variables because ABAP(Example: n, d, and t and decimal floating point numbers) and database system both have different data types. Native SQL does not support automatic client handling.

  Native SQL statements can behave in different ways on different platforms (particularly when compared with Open SQL). This mainly affects the following:

  - Handling of blanks in strings
  - Calculation rules and roundings in arithmetic calculations
  - Overflow behavior

  Refer http://help-legacy.sap.com/abapdocu_750/en/abennative_sql.htm for more details about Native SQL.

- **SAP List Viewer with Integrated Data Access (ALV with IDA) :** SAP List Viewer with Integrated Data Access offers application developers option to use in-memory database which is used to display very large quantities of data and performing sorting, grouping, or filtering operations with a very fast response time. We are using cl_salv_gui_table_ida class for it.

IDA framwork will not select all data but analyse required columns, sorting, filters, where conditions and execute select query based on visible section of ALV(Visible rows and columns).
Refer https://help.sap.com/saphelp_nw75/helpdata/en/f0/1df4a12b0c41c59fc72d007915ea43/content.htm to understand IDA framework.
Refer https://help.sap.com/saphelp_nw75/helpdata/en/ef/eb734c8e6f41939c39fa15ce51eb4e/content.htm for more details about ALV with IDA and technical restriction.
Refer http://zevolving.com/2016/07/salv-ida-integrated-data-access-introduction/ for consumption of ALV with IDA.

10.5 ABAP Development Tools (ADT)

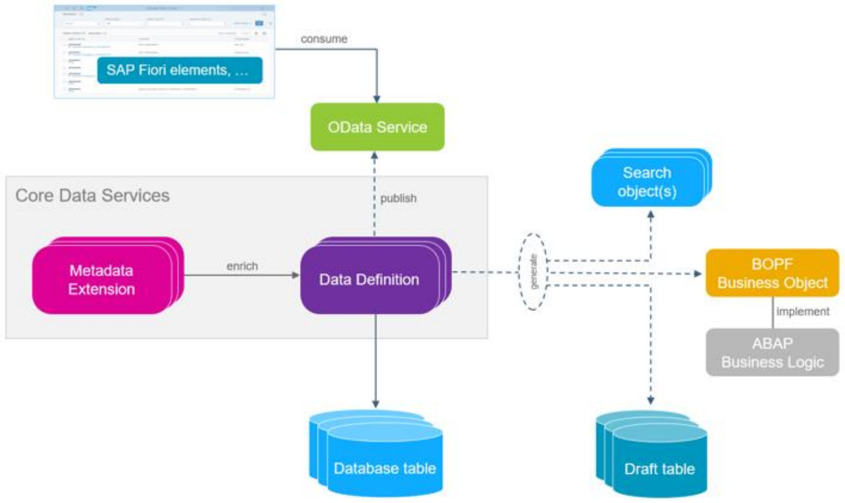
ABAP Development Tools for SAP NetWeaver (in short: ABAP Development Tools) is the ABAP IDE (Integrated Development Environment) built on top of the Eclipse platform(Eclipse 4.5 (Mars) or 4.6 (Neon)). Its main objective is to support developers in today's increasingly complex development environments by offering state-of the art ABAP development tools. These tools include strong and proven ABAP lifecycle management on the open Eclipse platform with powerful UI (user interface) capabilities.
Eclipse with ADT is the tool to go with in the future and therefore all developers should focus on the development with Eclipse besides a few restrictions like BOPF for older NW releases.

The integrated development workbench (Transaction SE80) provides all necessary features for a classical ABAP development project. ABAP CDS artificats can only be created using ABAP Development Tools(ADT). Eclipse is a Integrated a development workbench used for several development languages like Java, C#.
Refer - https://help.sap.com/viewer/7bfe8cdcfbb040dcb6702dada8c3e2f0/7.51.3/en-US/a3314a7fd9384ce8a40eff2d3b144628.html to more details.
Refer - Usage and installation guidelines of ADT Tool

inst_guide_abap_development_tools.pdf

10.6 Developing New Transactional Apps with Draft Capabilities

With SAP Net-Weaver AS for ABAP 7.51 innovation package SP02 or higher draft concept enables the developer to develop the user interface in a way, that the user can store draft versions of his work, without persisting it in the final state. This gives users flexibility to keep unsaved changes if an editing activity is interrupted, to prevent data loss if an app terminates unexpectedly, as a locking mechanism to prevent multiple users from editing the same object concurrently. Developers should consider this new capability in cases where this will be useful. The figure below provides an overview of the main development objects and technologies involved when creating a transactional, draft-enabled Fiori apps based on the new ABAP programming model.



Refer https://help.sap.com/viewer/cc0c305d2fab47bd808adcad3ca7ee9d/7.51.2/en-US/d36820f082c84085b6634be4576e351a.html

10.7 Some classical performance recommendations for database access

Below are golden rules regarding performance tuning on SAP HANA.

Golden Rule	**Detail/Example**	**HANA Relevance**
Keep the result sets small	Do not retrieve rows from the database and discard them on the application server using CHECK or EXIT, e.g. in SELECT loops. Make the WHERE clause as specific as possible.	
Minimise the amount of transferred data	Use SELECT with a field list instead of SELECT * in order to transfer just the columns you really need. Use aggregate functions (COUNT, MIN, MAX, SUM, AVG) instead of transferring all the rows to the application server.	Queries which are implemented with SELECT * can be slower with HANA, if the columnar storage is used.
Minimise the number of data transfers.	Use JOINs and or sub-queries instead of nested SELECT loops. Use SELECT.. FOR ALL ENTRIES instead of lots of SELECTs or SELECT SINGLEs. Use array variants of INSERT,	Arrays will be more efficient with column based architecture. Nested SELECTs will be causing more inefficiency (relatively speaking) then with row based databases.

	UPDATE, MODIFY, and DELETE.	
Minimise the search overhead.	Define and use appropriate secondary indexes.	Seconday indexes are generally not required in SAP HANA except for some special scenario as explained in the introduction of this chapter. Hence this rule has lost some of its importance.
Keep load away from the database.	Avoid reading data redundantly. Use table buffering where possible and do not bypass it. Sort data in your programs (unless ordering is with the primary table key).	In terms of HANA, you still want to keep unnecessary load away from the database. You DO however want to give the database your most data-intensive calculations to the database. This could be achieved by heavy SQL called from the application side or code pushdown to SAP HANA.

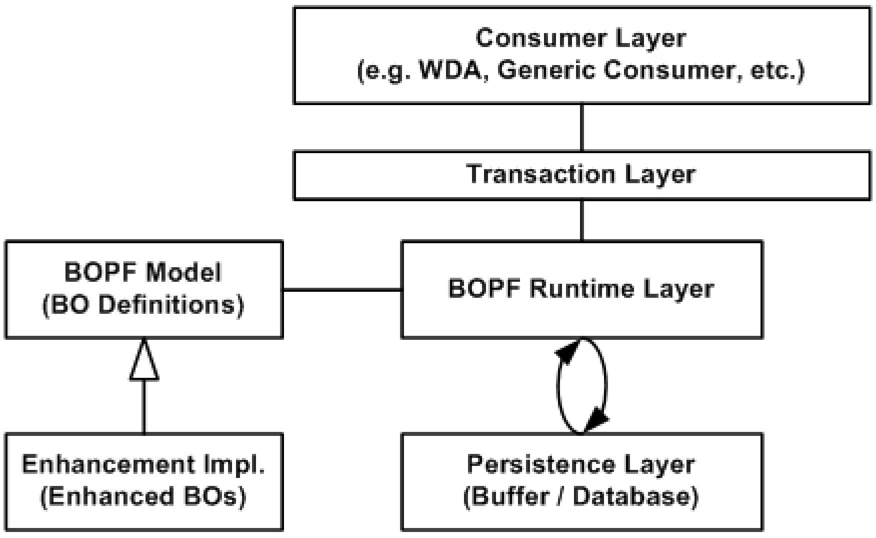
Refer https://help.sap.com/viewer/bed8c14f9f024763b0777aa72b5436f6/1.0.12/en-US/ to view performance guidelines for ABAP Development on the SAP HANA Database.

11 Business Object Processing Framework (BOPF)

Business Object Processing Framework is an ABAP OO-based framework that provides a set of generic services and functionalities to speed up, standardise, and modularise your development. BOPF manages the entire life cycle of your business objects and covers all aspects of your business application development.
Generally Eclipse is the tool to go for creation of BOPF related artifacts however it is recommended to use BOBX if there are some features missing in Eclipse.

11.1 Business Object Processing Framework (BOPF) architecture:

Business Object Processing Framework is a layer based architecture. Please find below screenshot for it.



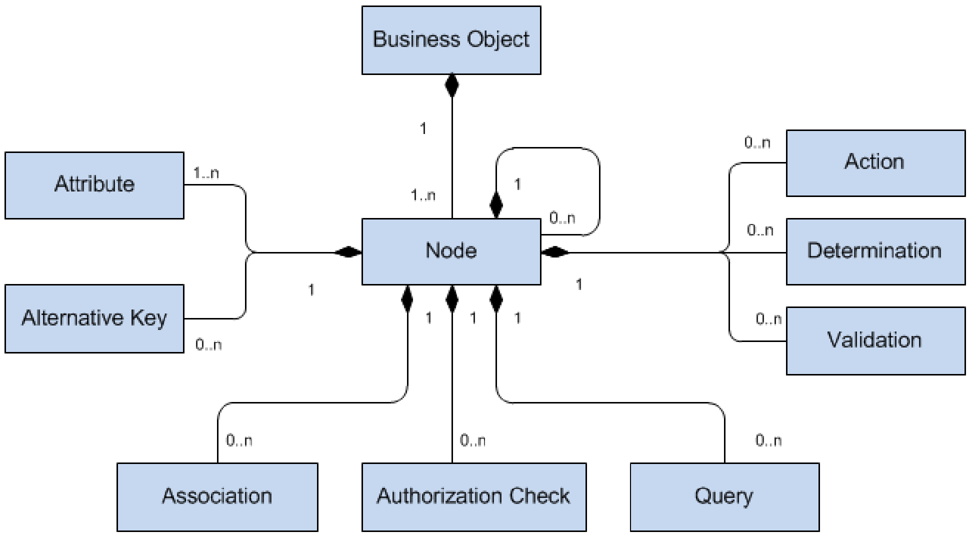
- **Consumer Layer:** At the consumer layer, we can utilise the BOPF API methods to create new Business Objects, search for existing Business Objects, update selected Business Objects.
- **Transaction Layer:** Interactions with Business Objects within the BOPF are brokered through a centralised transaction layer which handles low-level transaction handling details such as object locking, etc.
- **BOPF Runtime Layer:** This layer contains all of the functionality required to instantiate Business Objects, trigger their functionality.
- **Persistence Layer:** This layer supporting storage of Business objetcs data within the database and supporting data buffering via shared memory as well as the definition of transient nodes and attributes that are loaded on demand.

Refer- https://archive.sap.com/documents/docs/DOC-45425 for more details about it.

https://help.sap.com/saphelp_nw751abap/helpdata/en/31/d2958acf714f4e9aeb42d85c517523/frameset.htm for creation and usage of BOPF.

11.2 Business object metadata model:

In BOPF, a business object is represented as a hierarchical tree of nodes. A single node includes a set of semantically related attributes and the corresponding business logic. For each node, several types of entities can be defined to describe the specific business logic part of the business object.



- **Action:** An action is an entity assigned to an individual node of a business object that is used to implement a service (operation or behavior) of the business object.

- **Determination:** In BOPF, a determination is an entity of a business object node that is used to provide functions that are automatically executed as soon as a certain trigger condition is fulfilled. A determination is triggered internally on the basis of changes made to the node instance of a business object. The trigger conditions are checked by BOPF at different points during the transaction cycle, depending on the determination times and the changing operations on the relevant node instances. For each determination, it is necessary to specify both the points of time and the changes that form the trigger condition. Changes can include creating, updating, deleting, or loading node instances. You can use a determination primarily to compute data that is derived from the values of other attributes. The determined attributes and the determining attributes of the trigger condition either belong to the same node or to different nodes.

- **Validation:** In BOPF, a validation is an entity of a business object node that is triggered in certain situations to check various aspects of a given set of node instances. In particular, it is used to either validate whether a specific action can be executed on a given node instance (action validations) or whether a set of node instances is consistent (consistency validations). Validations never modify any node instance data but return the messages and keys of failed (inconsistent) node instances.

- **Query:** In BOPF, a query is a business object entity that is always assigned to a certain node. Queries never modify any node instance data. The result of the query execution is a set of keys of all node instances that match the query criteria. In order to provide various types of search criteria, a suitable filter structure can be associated with a query.

Refer - https://help.sap.com/viewer/aa7fc5c3c1524844b811735b9373252a/7.51.3/en-US/e5ea9085cfe2494faacae415ff8131da.html for more details about Business object metadata model

11.3 CDS view with BOPF

In ABAP 7.51, the semantic link between the CDS definition and its corresponding BOPF representations is established by a set of well-defined annotations that are added using the DDL editor. When the CDS contains BOPF annotations, a corresponding BOPF model and definitions are automatically activated in the background

Refer - http://sapinsider.wispubs.com/Assets/Articles/2016/October/SPI-Introducing-ABAP-751 for implementation of CDS view with BOPF.

Refer- https://help.sap.com/viewer/cc0c305d2fab47bd808adcad3ca7ee9d/7.51.3/en-US/896496ecfe4f4f8b857c6d93d4489841.html

For CDS annotations for BOPF.

12 ABAP Naming Conventions

Naming conventions describe the standardised and binding requirements for naming software objects (e.g. classes, function modules) and/or objects within the source code (e.g. variables).
In general names for all data fields and parameters should be chosen to describe their meaning and contents. The standard SAP names have to be used if SAP data elements or tables are referenced.
Upper-case and lower-case characters inside a name are problematic when the Pretty Printer is used, thus parts of names should be delineated by using underscores.
SAP UI5/BSP developments should also follow these guidelines

12.1 Namespace

Customer-specific objects and SAP objects will be separated in accordance with SAP Note 16466. However, the 'Y' prefix must not be used. Please use the 'Z' prefix instead.
'Z' prefix will be used for all objects created as part of the AHEAD GLOBAL Template.
'Y[country code]" prefix will be used for localised country / region specific developments during AHEAD Rollouts.
A request for the '/ALDI/' namespace will not be submitted and thus, a corresponding namespace will not be used.

Country	Code
Germany	DE
USA	US
Australia	AU
UK/Ireland	GB
Austria/Hungary	AT

12.2 Generally binding naming conventions

Adherence to the naming conventions specified in the following sections is mandatory. An overview of the components/abbreviations that are used when compiling naming conventions is provided in the table below.

Abbreviation	Definition
E2E	End to end stream AHEAD uses combinations of SAP Module as E2E stream (F2M, Q2C).
K	Constant
C	Alphanumerical character
N	Numerical character
…	User-defined length

Table 1: Abbreviations used in naming conventions

Values for Modules [mm]:

SAP Modules	
Finance to Manage	F2M
Invest to Divest	I2D
Market to Customer	M2C
Master Data Management	MD M
Order to Cash	O2C
Purchase to Pay	P2P
Product to Market	P2M
Return to Refund	R2R
Strategy to Assortment	S2A

12.3 Naming conventions

The following abbreviations for data types [t] apply in all below sections:

Type	Convention
variables	V
Structures	S
Tables	T
Data references	R
Class references	O
Interface references	O
BAdI references	O
Exception class references	X

12.3.1 Repository Objects

Type	Convention	Example
Development packages	ZE2E…	ZP2PRUECK
Reports	ZE2ER_<Text to represent use>	Z~~E~~P2PR_…

Test Programs	Z<DeveloperID>...	
Interface Programs	ZE2EI_<Text to represent use>	ZF2MI_..
Conversion Program	ZE2EC_<Text to represent use>	ZP2PC_..
Module pools	SAPMZE2E **Screen GUI status:** 4 digit no. same as the screen no. Screen GUI title: Last 3 digits of the screen number	SAPMZO2C
Include programs	ZE2EN..._ <include type*>	ZR2R_TOP
Transactions	ZE2E...	ZI2D01
Message classes	ZE2E...	ZF2MRUECK
BAPI	ZE2EBAPI_...	ZD2FBAPI_
Web Dynpro ABAP	ZE2E...	ZF2M_ADMIN

Table 2: Naming convention for structural elements

* An include program name begins with the name of the module pool without the 'SAP' prefix. The type of the include program is to be specified at the end.

- TOP Top Include
- PBO Process Before Output Include
- PAI Process After Input Include
- FORMSInclude for subroutines
- CLASS Include for local classes

If required, CLASS may be further subdivided into a definition include and an implementation include program. However, this subdivision is not mandatory.

12.3.2 Data dictionary objects

The name of all objects that are part of one development should start with the same prefix.

Type	**Convention**	**Example**
Tables	ZE2E...	ZP2PORDER
Views	ZE2E..._V	ZR2RORDER_V
Table types	ZE2E..._TT	ZP2MDOCUMENT_TT
Structure	ZE2E..._S	ZF2MDOCUMENT_S
Data elements/domains	ZE2E...	ZD2F2MFFTDATE
Search help	ZE2E..._SH	ZR2RPROJE_SH
Lock objects	EZ_E2E<Table>	EZ_ZO2CMINFO
Type groups	ZE2E	ZP2P02
Fields in Structure/Table type/Transparent table	Maximum possible, use standard SAP field names if referenced data element is standard SAP data element. While appending fields to standard SAP table or structures prefix ZZ to the new field names. E.g. ZZ<Text to	

	represent use>	
Database table secondary index	SAP Standard Table: Z<xx> Custom table: <xxx>	
Append Structures	Ideally should use SAP proposed name while creating append structure. If for any reason the same could not be used, use following convention… **Z**<Table name>**_<…>**	

Table 3: Naming convention for data dictionary objects

Based on the requirement that structures, data elements, etc. are to contain the name of the module, redundant names are permitted if definitions of the same type have to be used in different modules. In this context, FI and CO are to be considered as a joint module for which the 'FA' identification code can be used.

12.3.3 Classes and interfaces

Class and interface names should have a meaningful name reflecting the functionality of the class being developed.

Type	**Convention**	**Example**
Classes	ZCL_E2E_…	ZCL_O2C_BILL
Implementation class	ZCL_E2E_IM_…	
Interfaces	ZIF_E2E…	ZIF_O2C_BOOKING
Exceptions	ZCX_E2E_,,,	ZCX_O2C_ERROR

Table 4: Naming convention for classes and interfaces

Class Attributes

Attributes of a class are to be categorised as 'private' / 'protected'. Descriptive names are to be used for attributes. Attributes are to be fetched via the 'get' and 'set' methods.

Type	**Convention**
Variable attribute	MV_…
Structure attribute	MS_…
Table attribute	MT_…
Static attribute	MS[t]_…
Constants	MC_..

Table 5: Naming convention for attributes

Class Methods

Short, descriptive names in English are to be used for methods which reflects the functionality of the method. The following prefixes indicate the purpose of the relevant method:

Type	**Convention**
Defining values	set_…
Returning values	get_…
Storing data in the database	save_…

Sending information	send_...

Table 6: Naming convention for methods

The exact function is to be specified in the description of the method. If sufficient space is not available, a comprehensive documentation of the method is to be provided as part of the documentation of the class.

Method signatures

The parameters of a method are referred to as follows.

Parameter type	**Prefix**
Importing	i[t]_...
Exporting	e[t]_...
Changing	c[t]_...
Returning	r[t]_...

Table 7: Naming convention for methods

[t] – Data type (e.g. v = variable, t = table, s = structure)

12.3.4 Function groups and function modules

Objects that are part of a function group are to be named as follows.

Type	**Convention**	**Example**
Function groups	ZE2E_...	ZD2F_CONFIG
Function modules	Z_E2E_...	Z_O2C_INVOICE_FOR_CUSTOMER
Function modules parameters	**Import parameters: i**[t]_.... **Export parameters: e**[t]_.... **Table parameters:** [t]t_.... > **Changing parameters: C[t]_....** **Exceptions:** <Name to represent use>	

Table 8: Naming convention for function groups and function modules

12.3.5 Enhancements

Enhancements are to be named as follows.

Type	**Convention**	**Example**
Enhancement spot	ZES_...	ZES_MV45A
Enhancement definition	ZED_...	ZED_ MV45A
Enhancement implementation	ZEI_...	ZEI_ MV45A

Table 9: Naming convention for enhancements

12.3.6 Forms

Forms are to be named as follows.

Type	**Convention**	**Example**
Adobe Forms form	ZE2EA_...	ZD2FA_VBK1

Adobe Forms interface	ZE2EA_..._IF	ZF2MA_VKB_IF
Smart Forms/SAP script form	ZE2EF_...	ZO2CF_VBK1
Form style & smart style	ZE2EF_...	ZD2FF_VBK1
Form Print Program	ZE2EF_...	ZI2DF_VBK1

Table 10: Naming convention for forms

12.3.7 Transports

Short descriptions of transports are to be named as follows.

Convention	**Purpose**	**Type**	**Example**
PTE2E_<Reference no.>:Description	**I**ncident **C**hange **P**roject	**C**ustomizing **W**orkbench **M**aster data	PCXXX_INnnnn: Basic configuration (collect. transport) PWF2M: GAPnnn_Transformation AUTH FIELD CLASS CCFIAA: CRnnn_Agent assignment

Table 11: Naming convention for transports

- P Purpose of the transport: **I**ncident, **C**hange, **P**roject
- T Type: **C**ustomising, **W**orkbench, **M**aster data
- E2E Module: F2F for example
- Please note: XX/XX for overarching modules (MM)/overarching subcomponents (UU)
- Reference no. : - Incident no. / Change Request no. / GAP no. / Development id (optional)

12.3.8 Data definitions and data declarations

Any data field and parameter must have a two character prefix that enables the developer to identify several aspects:

- Visibility of the data field (local, global, object, class) or in case of a parameter the usage (importing, exporting, etc.)
- The general type (elementary, structure, table, object etc.)
- It is recommended to indicate the sorting type of a table in its name with a third character. Add '_H' for hashed tables and '_S' for sorted tables

Data definitions and data declarations are to be named as specified below. Descriptive names are to be used.

Declaration type	**Naming convention**
Local variable	l[t]_...
Global variable	g[t]_...
Static variable	s[t]_...
Field symbol	<[t]_...>
Global constant	gc_...

Ranges	r_
‘s_*’ select option	s_...
‘p_’ parameter (selection screen)	p_...
Internal tables – Standard	[g/l]t_<Name>
Internal tables – Sorted	[g/l]t_s_<Name>
Internal tables – Hash	[g/l]t_h_<Name>
Work areas	ls_<Name>
FORM parameter	
Using	i[t]_...
Changing	c[t]

Table 12: Naming convention for data definitions and data declarations

12.3.9 Validations/substitutions

Current validations:

- ALDI-H ALDI validation header
- ALDI-P ALDI validation item/position
- ALDI-D ALDI validation complete document

12.3.10 Standard texts

Standard texts are to be named as follows.

Type	Convention	Example
Standard text	ZE2ETX_... **Exception:** Custom naming conventions for building Logo library to allow dynamic logo selection.	ZE2ETX_HEADER

Table 13: Naming convention for standard texts

12.3.11 Floor Plan Building Blocks

A Floor Plan Application is composed of one or more GUIBBs/ Freestyle UIBBs (Web Dynpro Components / Applications) and/or an Application Configurations.

Component Configuration The Component Configuration name should be a self-explanatory text with respective Floor Plan type / GUIBB. **Position 1:** 'ZWDC' should be used as the prefix for all the custom Component Configuration **Position 2:** use '_' **Position 3:** use 3 characters to specify the Application area. E.g. F2M **Position 4:** use '_' **Position 5:** use 3 characters to specify the type of Floor Plan / Generic component. E.g. GAF or QAF or OIF or OVP or IDR For FORM Component use FRM For LIST Component use LST For TREE Component use TRE For TABBED Component use TAB For SEARCH Component use SRH For Analytics List Component use ALT For Launch Pad Component use LPD **Position 6:** use '_' **Position 7 and onwards**: Use these positions to give a compound name text that could be self-explanatory and reflects the purpose of program. This could be like TAX_REPORT or DISPLAY_MATERIAL. Example: ZWD_F2M_GAF_TAX_REPORT 'ZWD' denotes the custom Webdynpro configuration 'F2M' denotes the Application area (Finance to Manage) 'GAF' denotes the type of the FLOORPLAN and GUIBB 'TAX_REPORT' denotes the purpose of the Program / configuration
Variant ID and Name The Variant ID and Name inside a Floor Plan should be self-explanatory text and both should be same. **Position 1:** 'ZVRT' should be used as prefix for all the Variant IDs and Names. **Position 2:** use '_' **Position 3 and onwards:** Use these positions to give a compound name text that could be self-explanatory and reflects the purpose of variant. E.g., MASS_DEAL Example: ZVRT_MASS_DEAL 'ZVRT' denotes Custom Variant 'MASS_DEAL" denotes the purpose of the Variant

Dialog Box ID and Name The Dialog Box ID and Name inside a Floor Plan should be self- explanatory text and both should be same. **Position 1:** 'ZDBX' should be used as prefix for all the Variant IDs and Names. **Position 2:** use '_' **Position 3 and onwards:** Use these positions to give a compound name text that could be self-explanatory and reflects the purpose of Dialog Box. E.g., CONFIRM_DEAL_ Example: ZDBX_CONFIRM_DEAL 'ZDBX' denotes Custom Dialog Box 'CONFIRM_DEAL' denotes purpose of the dialog box
Feeder Class The Feeder class Name should be self-explanatory text with respective GUIBB/ UIBB. **Position 1:** 'ZCL' should be used as prefix for all custom classes **Position 2:** use '_' **Position 3**: use 3 characters to specify the project. E.g. COS or GPP **Position 4:** use '_' **Position 5:** use 3 characters to specify the type of GUIBB. Example: For FORM Component use FRM For LIST Component use LST For TREE Component use TRE For TABBED Component use TAB For SEARCH Component use SRH For Analytics List Component use ALT For Launch Pad Component use LPD **Position 5:** use '_' **Position 6 and onwards:** Use these positions to give a compound name text that could be self-explanatory and reflects the purpose of Feeder Class. Eg:- EMP_DETAILS Example: ZCL_COS_FRM_EMP_DETAILS 'ZCL' denotes the Custom Class 'COS' denotes the name of the Programming 'FRM' denotes the type of GUIBB (FORM) 'EMP_DETAILS' denotes the purpose of the Feeder Class

12.3.12 BOPF naming conventions:

BOPF objects are to be named as follow.

BOPF objects	Naming Convention

BOPF BO	<customer namespace>/<project namespace>_BOPF_<BO_Name>
Persistent Structure	<customer namespace>/<project namespace>_S_<Node_Name>_P
Transient Structure	<customer namespace>/<project namespace>_S_<Node_Name>_T
Database Table	<customer namespace>/<project namespace>_<Node_Name>
Combined Structure	<customer namespace>/<project namespace>_S_<Node_Name>
Combined Table Type	<customer namespace>/<project namespace>_TT_<Node_Name>
Constants Interface	<customer namespace>/<project namespace>_IF_<Node_Name>_C
Determination Class	<customer namespace>/<project namespace>_CL_D_<Node_Name>_<Determination_Name>
Association Class	<customer namespace>/<project namespace>_CL_AS_<Node_Name>_<Association_Name>
Action Class	<customer namespace>/<project namespace>_CL_A_<Node_Name>_<Action_Name>
Validation Class	<customer namespace>/<project namespace>_CL_V_<Node_Name>_<Validation_Name>
Query Class	<customer namespace>/<project namespace>_CL_Q_<Node_Name>_<Validation_Name>

12.3.13 CDS view, AMDP and CDS Table Function naming conventions:

Objects	Naming Convention
DDL Source file/CDS view	<customer namespace>/<project namespace>_CDS_<P/I/C>_*
SQL view	<customer namespace>/<project namespace>_<view type>_<P/I/C>_*
CDS Table Function	<customer namespace>/<project namespace>_CDSTF_<P/I/C>_*
AMDP	<customer namespace>/<project namespace>_CL_AMDP_<class name>

*P = Private helper view independent from basic, interface or consumption
I = Interface view
C = Consumption view

1.1.1.1 Header Comment in CDS View

The header comment in CDS view should contain the details (Development Id, user id of the author, change log, Change Request no., documentation and reason for changes). Example of a CDS view header:

```
//----------------------------------------------------------------------
// Identification
// Author              : USERID,
// Creation date       : XX.XX.XXXX
// Owner               : USERID, (Business/Functional)
// Stream/Development No. : XXXXX
// Short Description :
// THIS IS AN EXAMPLE OF A CDS VIEW HEADER
//
//----------------------------------------------------------------------
// Changes
// Chg. date    Developer Owner CRT-ReferenceID  Description
// XX.XX.XXXX    User       User      XXXXXXXX
//----------------------------------------------------------------------
```

Figure 4: Header Comment

This should be used for all CDS view in order to maintain consistency across all CDS view developed in a system.

12.3.14 Analytic, Attribute and Calculation View and Stored Procedure

Objects	Naming Convention
Analytic View	<customer namespace>/<project namespace>_ANA_VIEW_*
Attribute View	<customer namespace>/<project namespace>_ATR_VIEW_*
Calculation View	<customer namespace>/<project namespace>_CAL_VIEW_*
Stored Procedure	<customer namespace>/<project namespace>_*

12.3.15 Cloned object naming convention:

Type	**Convention**	**Example**
Development packages		
Reports	ZE2ER_<Original program name>_<Text to represent use>	ZP2PR_...
Test Programs	Z<DeveloperID>...	
Interface Programs	ZE2EI_<Original program name>_<Text to represent use>	ZF2MI_..
Conversion Program	ZE2EC_<Original program name>_<Text to represent use>	ZP2PC_..
Module pools	SAPMZE2E_<Original program name>_ **Screen GUI status:** 4 digit no. same as the screen no. Screen GUI title: Last 3 digits of	SAPMZO2C

	the screen number	
Include programs	ZE2EN_<Original program name>_ <include type*>	ZR2R__<Original program name>_TOP
t Transactions e	ZE2E_<Original transaction code>	ZI2D_<Original transaction code>
Message classes	ZE2E_<Original message class>	ZF2M_<Original message class>
BAPI I f	ZE2EBAPI_<Original BAPI name>	ZD2FBAPI_<Original BAPI name>
Web Dynpro ABAP	ZE2E_<Original program name>	ZF2M_<Original program name>

m
aximum length doesn't support the above naming convention then adjust the Original program name and text in such a way that original program can be easily identified.

13 New ABAP Syntax

This development guidelines is applicable for S4 as well as non-S4 systems so new syntax can only be used wherever possible.

13.1 Inline Declaration

13.1.1 Declaration of a lhs-variable for a simple assignment

Old syntax:

```
DATA text TYPE string.
text = `X`.
```

New syntax:

```
DATA(text) = `X`.
```

Where X is

- any data object
- a functional method call
- a call of a built-in function
- an arithmetic expression
- a string expression
- a bit expression
- a constructor expression
- a table expression

13.1.2 Declaration of table work areas

Old syntax:

```
DATA wa like LINE OF itab.
LOOP AT itab INTO wa.
...
ENDLOOP.
```

New syntax:

```
LOOP AT itab INTO DATA(wa).
 ...
ENDLOOP.
```

13.1.3 Declaration of a helper variable

Old syntax:

```
DATA cnt TYPE i.
FIND ... IN ... MATCH COUNT cnt.
```

New syntax:

```
FIND ... IN ... MATCH COUNT DATA(cnt).
```

13.1.4 Declaration of a result

Old syntax:

```
DATA xml TYPE xstring.
CALL TRANSFORMATION ... RESULT XML xml.
```

New syntax:

```
CALL TRANSFORMATION ... RESULT XML DATA(xml).
```

13.1.5 Declaration of actual parameters

Old syntax:

```
DATA a1 TYPE ...
DATA a2 TYPE ...

oref->meth( IMPORTING p1 = a1
            IMPORTING p2 = a2
             ... )
```

New syntax:

```
oref->meth( IMPORTING p1 = DATA(a1)
            IMPORTING p2 = DATA(a2)
            ... ).
```

13.1.6 Declaration of reference variables for factory methods

Old syntax:

```
DATA ixml TYPE REF TO if_ixml.
DATA stream_factory TYPE REF TO if_ixml_stream_factory.
DATA document    TYPE REF TO if_ixml_document.
ixml = cl_ixml=>create( ).
stream_factory = ixml->create_stream_factory( ).
document    = ixml->create_document( ).
```

New syntax:

```
DATA(ixml) = cl_ixml=>create( ).
DATA(stream_factory) = ixml->create_stream_factory( ).
DATA(document)    = ixml->create_document( ).
```

13.1.7 Field Symbols

For field symbols there is the new declaration operator FIELD-SYMBOL(...) that you can use at exactly three declaration positions.

```
ASSIGN ... TO FIELD-SYMBOL(<fs>).
LOOP AT itab ASSIGNING FIELD-SYMBOL(<line>).
...
ENDLOOP.
READ TABLE itab ASSIGNING FIELD-SYMBOL(<line>) ...
```

13.2 MOVE-CORRESPONDING for Internal Tables

Move-corresponding should only be used when source and target structures are locally defined in the object and are not part of the ABAP dictionary.
Syntax:
MOVE-CORRESPONDING itab1 TO itab2 EXPANDING NESTED TABLES KEEPING TARGET LINES.

New additions EXPANDING NESTED TABLES and KEEPING TARGET LINES allow to resolve tabular components of structures and to append lines instead of overwriting existing lines.

13.3 LET Expressions in Constructor Expressions

New LET expressions as sub expressions of constructor expressions allow you to define local variables or field symbols as auxiliary fields of constructor expressions.
Example:
LET expression in a table construction using the VALUE operator.

```
cl_demo_output=>new( )->write(
            VALUE text( LET it = `be` IN
                        ( |To { it } is to do|       )
                        ( |To { it }, or not to { it }| )
                        ( |To do is to { it }|       )
                        ( |Do { it } do { it } do|    ) ) )->display( ).
```

13.4 CORRESPONDING Operator

The new constructor operator CORRESPONDING allows to execute a “MOVE-CORRESPONDING” for structures or internal tables at operand positions. Besides assigning components of the same name you can define your own mapping rules.
Example:
struct2 = CORRESPONDING #(struct1 MAPPING b4 = a3).

13.5 Table Comprehensions

A new FOR sub expression for constructor expressions with operators NEW and VALUE allows to read existing internal tables and to construct new tabular contents from the lines read.
Example:
Construction of an internal table itab2 from lines and columns of an internal table itab1. You can also use the CORRESPONDING operator to construct the lines.

```
DATA(itab2) = VALUE t_itab2( FOR wa IN itab1 WHERE ( col1 < 30 )
                             ( col1 = wa-col2 col2 = wa-col3 ) ).
```

13.6 Meshes

Meshes are special structures whose components are internal tables, which can be linked to each other using associations. Associations are evaluated by specifying mesh paths in suitable expressions and statements.
Example:
A mesh flights is declared from a mesh type t_flights. In t_flights you have tabular components as so called mesh nodes that are linked by associations. A structured data object root is constructed to serve as the start line for the following LOOP over a mesh path. The results are

lines from sflight that are found by following the mesh path evaluating the associations between its mesh nodes.

```
TYPES:
        t_scarr TYPE SORTED TABLE OF scar WITH UNIQUE KEY carrid,
        t_spfli TYPE SORTED TABLE OF spfli WITH UNIQUE KEY carrid connid,
        t_sflight TYPE SORTED TABLE OF sflight WITH UNIQUE KEY carrid connid fldate.
TYPES:
        BEGIN OF MESH t_flights,
         scarr TYPE t_scarr
           ASSOCIATION to_spfli TO spfli
                                      ON carrid = carrid USING KEY primary_key,
         spfli TYPE t_spfli
                ASSOCIATION to_sflight TO sflight
                        ON carrid = carrid AND
                      connid = connid USING KEY primary_key,
         sflight TYPE t_sflight,
        END OF MESH t_flights.
DATA:
        flights TYPE t_flights.
...
DATA(root) = flights-scarr[ carrname = 'United Airlines' ].
LOOP AT
         flights-scarr\to_spfli[ root ]\to_sflight[ ]
                                        INTO DATA(wa).
                                                                      ...
ENDLOOP.
```

13.7 Open SQL

From 7.40, SP05

Lists in Open SQL statements can and should be separated by a comma.

Host variables in Open SQL statements can and should be escaped by a @.

Only then you will be able to use other new functions that are based on a new SQL parser in the ABAP kernel.

Example:

```
            SELECT carrid, connid, fldate FROM sflight
                    INTO CORRESPONDING FIELDS OF TABLE @sflight_tab
                    WHERE carrid = @carrier AND
                    connid = @connection
                    ORDER BY carrid, connid.
```

13.8 SQL Expressions

We can use SQL expressions in a column list behind SELECT. The result of such an expression is calculated on the database (code push down!) and written into the respective columns of the result set. Operands can be data base columns or host variables.

Possible expressions are

- elementary values
- arithmetic expressions

- arithmetic functions abs, ceil, floor, div, mod
- castings with cast
- string concatenations with &&
- coalesce
- case

Expressions can be mixed and prioritized with parentheses.
Examples:

```
SELECT id, num1, num2,
        cast( num1 AS fltp ) / cast( num2 AS fltp ) AS ratio,
        div( num1, num2 ) AS div,
        mod( num1, num2 ) AS mod,
        @offset + abs( num1 – num2 ) AS sum
        FROM demo_expressions
        INTO CORRESPONDING FIELDS OF TABLE @results
        ORDER BY SUM DESCENDING.
SELECT id, CASE char1
        WHEN ‘aaaaa’ THEN ( char1 && char2 )
        WHEN ‘xxxxx’ THEN ( char2 && char1 )
        ELSE @else
        END AS text
        FROM demo_expressions
        INTO CORRESPONDING FIELDS OF TABLE @results.
```

13.9 MOVE CORRESPONDING

The new system class CL_ABAP_CORRESPONDING allows you to assign components of structures or internal tables with dynamically specified mapping rules.
The mapping rules are created in a mapping table that is passed to a mapping object, e.g. as follows:

```
DATA(mapper) =
        cl_abap_corresponding=>create(
         source     = struct1
        destination = struct2
        mapping     = VALUE cl_abap_corresponding=>mapping_table(
        ( level   = 0
        kind    = cl_abap_corresponding=>mapping_component
        srcname = ‘...’
        dstname = ‘...’ )
        ( level   = 0
        kind    = cl_abap_corresponding=>mapping_component
        srcname = ‘...’
        dstname = ‘...’ )
        ( level   = 0
        kind    = cl_abap_corresponding=>mapping_component
        srcname = ‘...’
        dstname = ‘...’ ) ) ).
```

This is a simple example, where all structure components are on top level (0) and where all components are to be mapped (kind = cl_abap_coresponding=>mapping_component). More

complicated forms involve nested structures and exclusions. With srcname and dstname the component names can be specified dynamically. The table setup is similar to the mapping-clause of the CORRESPONDING operator.
After creating the mapping object, all you have to do is to execute the assignment as follows:

```
mapper->execute( EXPORTING source      = struct1
                           CHANGING  destination = struct2 ).
```

You can do that again and again for all structures or internal tables that have the same types as those used for creating the mapping object.

13.10 Value Operator VALUE

The value operator VALUE is a constructor operator that constructs a value for the type specified with type.

- ... VALUE dtype|#( ) ...
  Constructs an initial value for any data type.
- ... VALUE dtype|#(comp1 = a1 comp2 = a2 ...) ...
  Constructs a structure where for each component a value can be assigned.
- ... VALUE dtype|#((...) (...) ...) ...
  Constructs an internal table, where for each line a value can be assigned. Inside inner parentheses you can use the syntax for structures but not the syntax for table lines directly. But you can nest VALUE operators.

Note that you cannot construct elementary values (which is possible with instantiation operator NEW) – simply because there is no need for it.
For internal tables with a structured line type there is a short form that allows you to fill columns with the same value in subsequent lines

```
VALUE dtype|#( col1 = dobj11 ... ( col2 = dobj12 col3 = dobj13 ... )
               ( col2 = dobj22 col3 = dobj23 ... )
               ...
               col1 = dobj31 col2 = dobj32 ... ( col3 = dobj33 ... )
                                               ( col3 = dobj43 ... )
               ... ).
```

13.11 Instantiation Operator NEW

The instantiation operator NEW is a constructor operator that creates an object (anonymous data object or instance of a class).

- ... NEW dtype(value) ...
  Creates an anonymous data object of data type dtype and passes a value to the created object. The value construction capabilities cover structures and internal tables (same as those of the VALUE operator).
- ... NEW class(p1 = a1 p2 = a2 ...) ...
  Creates an instance of class class and passes parameters to the instance constructor.
- ... NEW #(...) ...
  Creates either an anonymous data object or an instance of a class depending on the operand type.

You can write a component selector -> directly behind NEW type(...).

13.12 Reference Operator REF

The reference operator REF constructs a data reference at operand positions.

- ... REF dtype|#(dobj) ...

Results in a data reference pointing to dobj with the static type specified by type. With other words, REF is the short form for GET REFERENCE OF dobj INTO.

You need it always when you have to pass data references to somewhere and don't want to create helper variables for that purpose.

13.13 Conversion Operator CONV

The conversion operator CONV is a constructor operator that converts a value into the type specified in type.

- ... CONV dtype|#(...) ...
  You use CONV where you needed helper variables before in order to achieve a requested data type.

  Old syntax:
  Example for parameter passing

  Method cl_abap_codepage=>convert_to expects a string but you want to convert a text field.
  DATA text TYPE c LENGTH 255.
  DATA helper TYPE string.
  DATA xstr   TYPE xstring.
  helper = text.
  xstr = cl_abap_codepage=>convert_to(source = helper).

  New syntax:
  Example for parameter passing

  Method cl_abap_codepage=>convert_to expects a string but you want to convert a text field.
  DATA text TYPE c LENGTH 255.
  DATA(xstr) = cl_abap_codepage=>convert_to(source = CONV string(text)).

  In such cases it is even simpler to write
  DATA text TYPE c LENGTH 255.
  DATA(xstr) = cl_abap_codepage=>convert_to(source = CONV #(text)).

13.14 Casting Operator CAST

The casting operator CAST is a constructor operator that executes an up or down cast for reference varaibles with the type specified in type.

- ... CAST dtype|class|interface|#(...) ...
  You use CAST for a down cast where you needed helper variables before in order to cast with ?= to a requested reference type.
  You use CAST for an up cast, e,g, with an inline declaration, in order to construct a more general type.
  You can write a compnent selector -> directly behind CAST type(...).

  Old syntax:

DATA structdescr TYPE REF TO cl_abap_structdescr.
structdescr ?= cl_abap_typedescr=>describe_by_name('T100').
DATA components TYPE abap_compdescr_tab.
components = structdescr->components.

New syntax:
DATA(components) = CAST cl_abap_structdescr(
cl_abap_typedescr=>describe_by_name('T100'))->components.

13.15 Conditional operators COND and SWITCH

- ... COND dtype|#(WHEN log_exp1 THEN result1
  [WHEN log_exp2 THEN result2]
  ...
  [ELSE resultn]) ...
  constructs a result of the specified type that depends on logical expressions.
- ... SWITCH dtype|#(operand
  WHEN const1 THEN result1
  [WHEN const2 THEN result2]
  ...
  [ELSE resultn]) ...
  constructs a result of the specified type that depends on a case differentiation.

With other words: IF and CASE as expressions in operand positions!

13.16 New Internal Table Functions

Line existence:
A new predicate function that returns true if a line exists:

IF line_exists(itab[...]).
...
ENDIF.
Where itab[...] is a table expression that works here like READ TABLE ... TRANSPORTING NO FIELDS.
(Note that we have already some other predicate functions in ABAP: matches and the contains-variants in the context of string processing).

Line index:
A new table function that returns the number of the line in the index used (primary or secondary index):
DATA(idx)= line_index(itab[...]).
Where itab[...] is a table expression that works here like READ TABLE ... TRANSPORTING NO FIELDS. If no line is found, the value 0 is returned. For Hash-Tables or hashed secondary keys, always the value -1 is returned.
(Note that this is the second table function in ABAP. The other one is lines).

13.17 FOR Expressions

With 7.40, SP05, the first version of the iteration operator FOR was introduced. You can use it in constructor expressions with VALUE and NEW for so called table comprehensions, as e.g.

```
DATA(itab2) = VALUE t_itab2( FOR wa IN itab1 WHERE ( col1 < 30 )
                            ( col1 = wa-col2 col2 = wa-col3 ) ).
```

This is an expression enabled version of LOOP AT itab. It didn't take long to ask also for expression enabled versions of DO and WHILE (couldn't stand them in otherwise expression enabled examples any more ...).
Therefore, with 7.40, SP08 we also offer conditional iterations with FOR:

```
... FOR i = ... [THEN expr] UNTIL|WHILE log_exp ...
```

13.18 GROUP BY for Internal Tables

We know the GROUP BY clause from SQL. There was not such a clause for internal tables up to now. All we had was that clumsy group level processing with statements AT NEW ... that relied on the order of table columns and contents that is sorted respectively.
With release 7.40, SP08 there is a real GROUP BY clause for LOOP AT itab that is much more powerful than the SQL one.

```
DATA flights TYPE TABLE OF spfli WITH EMPTY KEY.
SELECT * FROM spfli
       WHERE carrid = '...'
       INTO TABLE @flights.
DATA members LIKE flights.
LOOP AT flights INTO DATA(flight)
  GROUP BY ( carrier = flight-carrid cityfr = flight-cityfrom )
  ASCENDING
  ASSIGNING FIELD-SYMBOL(<group>).
  CLEAR members.
 LOOP AT GROUP <group> ASSIGNING FIELD-SYMBOL(<flight>).
    members = VALUE #( BASE members ( <flight> ) ).
  ENDLOOP.
  cl_demo_output=>write( members ).
ENDLOOP.
cl_demo_output=>display( ).
```

Looks like dreaded nested LOOPs, but it isn't quite that – no quadratic behaviour! What happens here is that the first LOOP statement is executed over all internal table lines in one go and the new GROUP BY addition groups the lines. Technically, the lines are bound internally to a group that belongs to a group key that is specified behind GROUP BY. The group key is calculated for each loop pass. And the best is, it need not be as simple as using only column values, but you can use any expressions here that normally depend on the contents of the current line, e.g. comparisons, method calls, The LOOP body is not evaluated in this phase!
Only after the grouping phase, the LOOP body is evaluated. Now a second (but not nested) loop is carried out over the groups constructed in the first phase. Inside this group loop you can access the group using e.g. the field symbol <group> that is assigned to the group in the above example. If you want to access the members of the group, you can us the new LOOP AT GROUP statement, which enables a member loop within the group loop. In the example, the members are inserted into a member table and displayed.

13.19 String Templates

The purpose of a string template is to create a new character string out of literal texts and embedded expressions.
A string template is defined by using the | (pipe) symbol at the beginning and end of a template.

```
DATA: character_string TYPE string.
character_string = |This is a literal text.|.
```

Embedded expressions are defined within a string template with curly brackets { expression }. Note that a space between bracket and expression is obligatory.
An expression can be a data object (variable), a functional method, a predefined function or a calculation expression. Some examples are:

```
character_string = |{ a_numeric_variable }|.
character_string = |This resulted in return code { sy–subrc }|.
character_string = |The length of text element 001 ({ text–001 }) is { strlen( text–001 ) }|.
```

13.20 Calculation assignments

+=, -=, *=, /= - Calculation Assignments

Syntax:

```
lhs                                                        +=
                                                          | -=
                                                          | *=
  | /= rhs.
```

These assignments have the same function as the following assignments of arithmetic expressions:

```
lhs          = lhs          +          (          rhs          ).

lhs          = lhs          -          (          rhs          ).

lhs          = lhs          *          (          rhs          ).

lhs = lhs / ( rhs ).
```

The content of lhs

- has the result of the parenthesized expression rhs added to it,
- or has the result of the parenthesized expression rhs subtracted from it,
- or is multiplied by the result of the parenthesized expression rhs,
- or is divided by the result of the parenthesized expression rhs,
- And the result is assigned to lhs. The calculation type is determined accordingly.

The following applies to the operands lhs and rhs:
This is a result position and the following can be specified (if numeric):

- Variables
- Writable expressions

This is a numeric expression position and the following can be specified (if numeric):

- Data objects
- Constructor expressions
- Arithmetic expressions
- Table expressions
- Functions

Type inference with the character # is not currently possible in constructor expressions. Character-like expressions and bit expressions cannot be specified.
Division by the value 0 is undefined and raises a handle able exception. The only situation where division by 0 does not raise an exception is if the dividend is also 0. Here, the result is set to 0.

Notes:
Alongside data objects, calculation assignments also allow expressions in the operand positions. This makes them more powerful than the statements ADD, SUBTRACT, MULTIPLY, DIVIDE, and also makes these statements obsolete.
No inline declarations are possible for lhs.
Calculation assignments can currently only be specified as standalone statements. They cannot be used in expressions, for example after NEXT in reduction operator REDUCE.
See also the concatenation assignment operator (&&=).
Example:
The variable n has the value 1.50 after the calculation assignments.

```
DATA n TYPE p DECIMALS 2.
n += 1.
n += 1.
n += 1.
n += 1.
n -= 2.
n *= 3.
n /= 4.
```

13.21 Client Handling

The new additions

- USING [ALL] CLIENTS [IN] in queries
- USING [ALL] CLIENTS [IN] in write statements

Make it possible to switch implicit client handling from the current default client to multiple clients.
This makes the addition CLIENT SPECIFIED obsolete in queries and obsolete in the write statements UPDATE SET and DELETE FROM.
Syntax

```
... { USING { CLIENT clnt }
    | { CLIENTS IN @client_range_tab }
    | { CLIENTS IN T000 }
    | { ALL CLIENTS } } ...
```

14 Upgrade

14.1 SPDD Adjustment

All the objects captured in SPDD during upgrade must be adjusted in such way that SAP new features as well as ALDI specific custom changes are adopted.
Pilot SAP Note must be re-implemented with valid version. If valid version of pilot SAP Note is not available, then connect with SAP to request valid version.

14.2 SPAU and SPAU_ENH Adjustment

All the objects captured in SPDD during upgrade must be adjusted in such way that SAP new features as well as ALDI specific custom changes are adopted.
Pilot SAP Note must be re-implemented with valid version. If valid version of pilot SAP Note is not available, then connect with SAP to request valid version.

14.3 Cloned Object Adjustment

Cloned objects are those objects which are created through copy of SAP standard objects. These objects must be recreated by copy of upgraded version of objects. This is necessary to adopt SAP new features.

14.4 SAP Note Adjustment

All the obsolete implemented SAP Note must be de-implemented and re-implemented with correct version.

15 SAP Note implementation strategy

SAP Note implementation strategy document link:
https://extranet.aldi-sued.com/projects/ito-tpm3-solman/Shared%20Documents/Additional%20Documentations/Note_implementation_v2.0.pptx

Note: As per the process, TCS TAM team is responsible to implement the SAP notes and move them across systems. SAP Note should not be implemented directly by Development team.

16 External Links

The documents and templates listed in the table below are Global Standard Operating Procedures and Policies that are directly associated with this document and used to create this document.
Please reference documents whenever it is useful to create a better understanding.

Document ID	Document title
	▪ ALDI: Base version '20-development-policy_1.2_EN' released for ASA on 3-Nov-2016 ▪ CAPGEMINI: Base version '3.5.3_iSAP PRICEFW Development Standards and Procedures'. ▪ SAP help website – https://help.sap.com