TABC42 ABAP Programming Techniques 2/2

TABC42 2/2

R/3 System

Release 46B

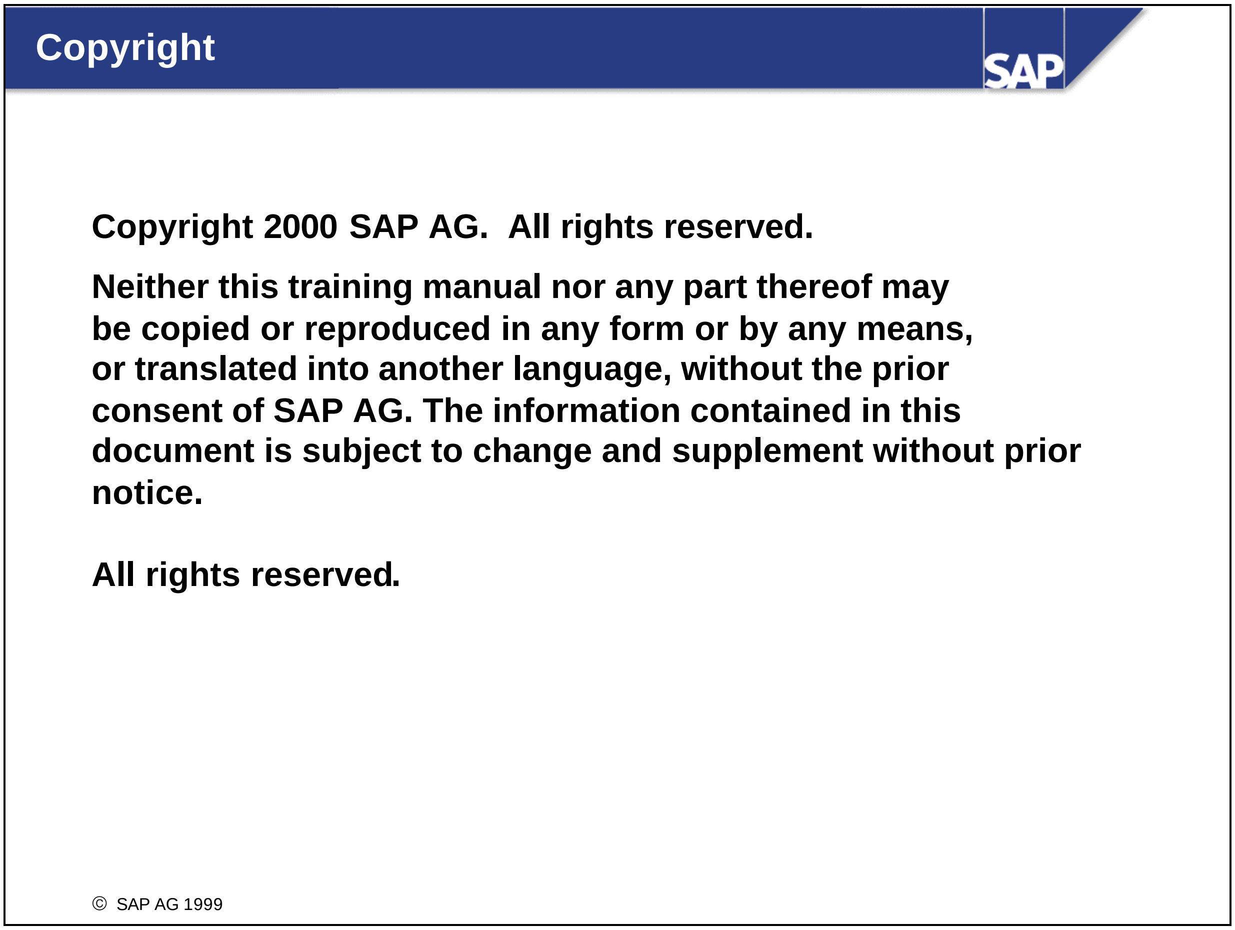
30.05.2000

TABC42 2/2

ABAP Programming Techniques

Part 2 of 2

- R/3 System
- Release 4.6B
- May 2000
- Material number 50039584



Copyright

Section Overview

Section **Basis Technology Overview**

Section **ABAP Workbench Concepts and Tools**

Section **Managing ABAP Development Projects**

Section **ABAP Dictionary**

Section **ABAP Programming Techniques**

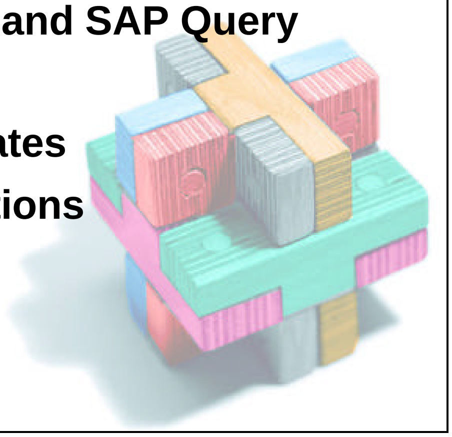
Section **Techniques for List Creation and SAP Query**

Section **Transaction Programming**

Section **Programming Database Updates**

Section **Enhancements and Modifications**

Section **Data Transfer**



Section: Programming Database Updates

Content: Programming Database Updates

Unit **Introduction**

Unit **Database Updates with Open SQL**

Unit **LUWs and Client Server Architecture**

Unit **SAP Lock Concept**

Unit **Organizing Database Updates**

Unit **Complex LUW Programming**

Appendix

Course Overview

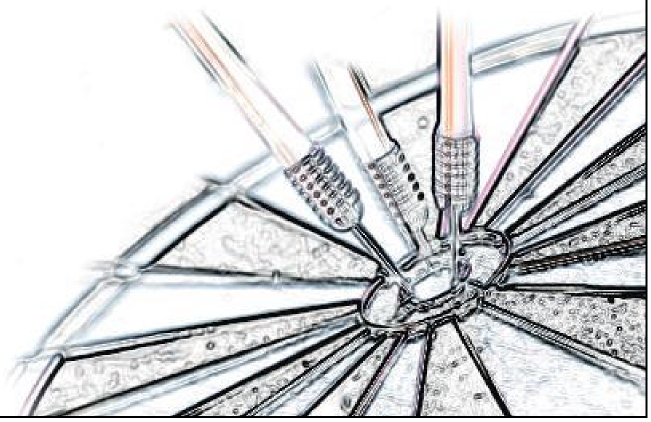
Contents:

- Course goals
- Course objectives
- Course content
- Course overview diagram
- Main business scenario

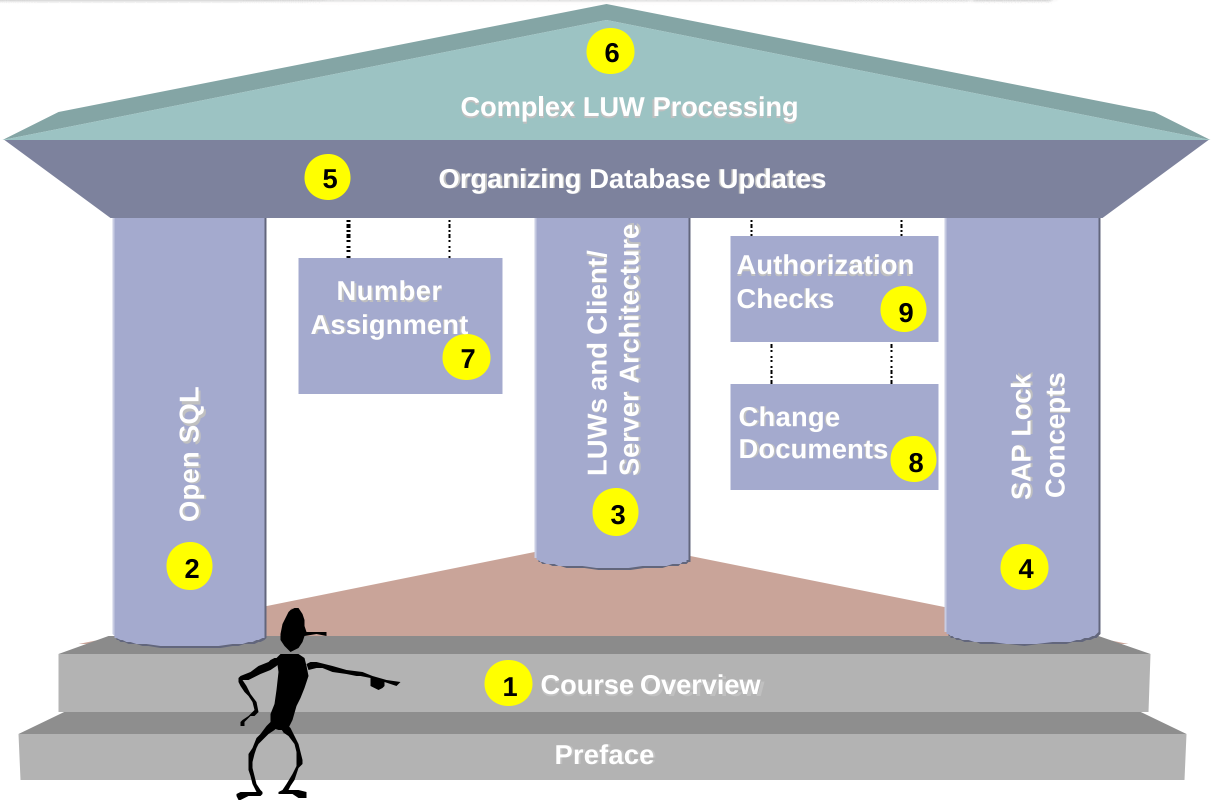
Course Goals

This course will prepare you to:

- Program database updates in the same way that you process transactions in the SAP R/3 System



Course Overview Diagram

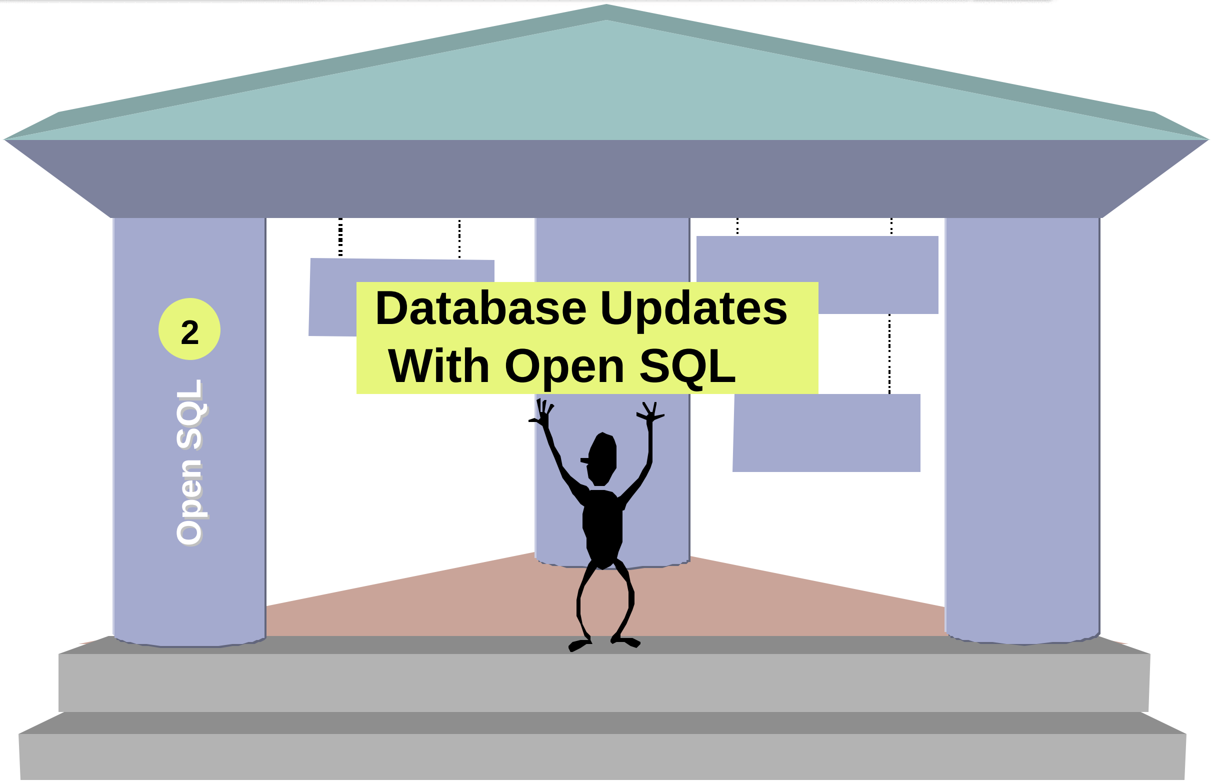


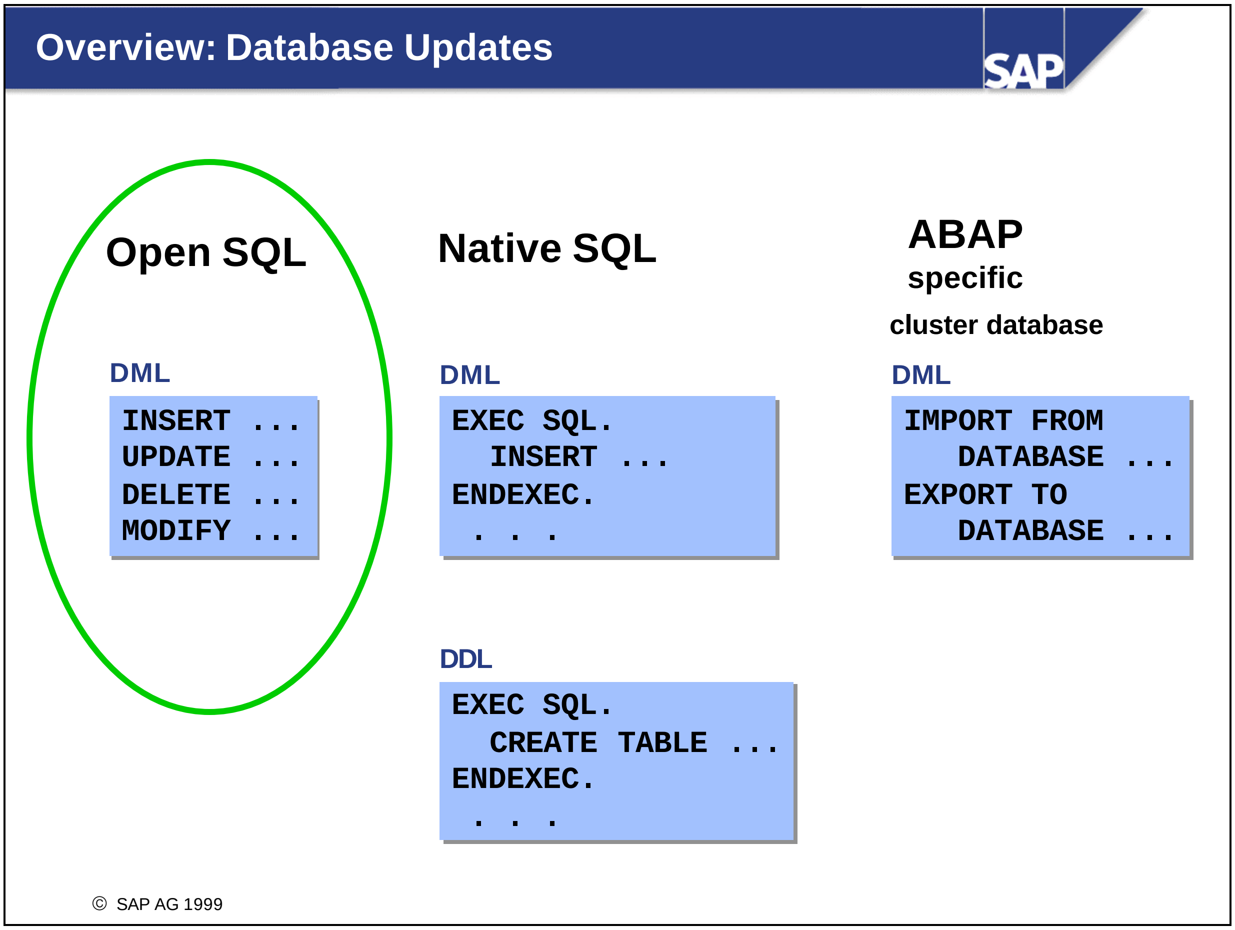
Database Updates With Open SQL

Contents:

- Open SQL
- Single record operations
- Set operations

Course Overview Diagram - Open SQL





Overview: Database Updates

Open SQL

DML

```
INSERT ...
UPDATE ...
DELETE ...
MODIFY ...
```

Native SQL

DML

```
EXEC SQL.
  INSERT ...
ENDEXEC.
 . . .
```

DDL

```
EXEC SQL.
  CREATE TABLE ...
ENDEXEC.
 . . .
```

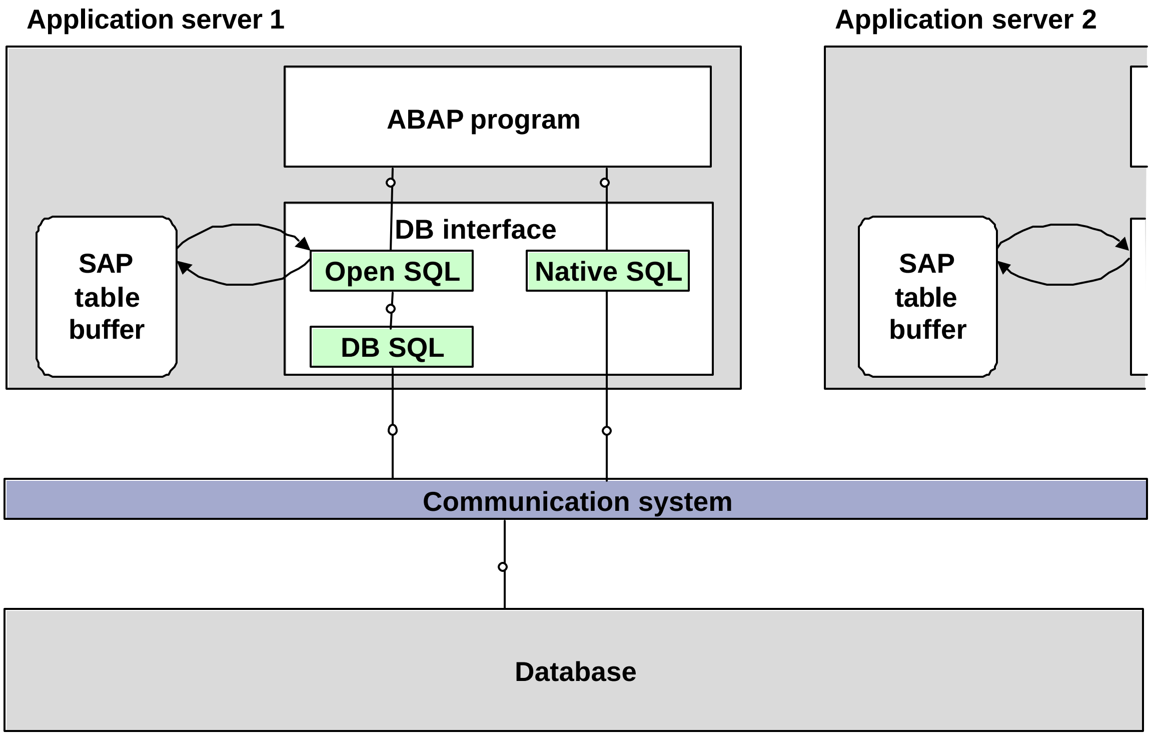
ABAP specific cluster database

DML

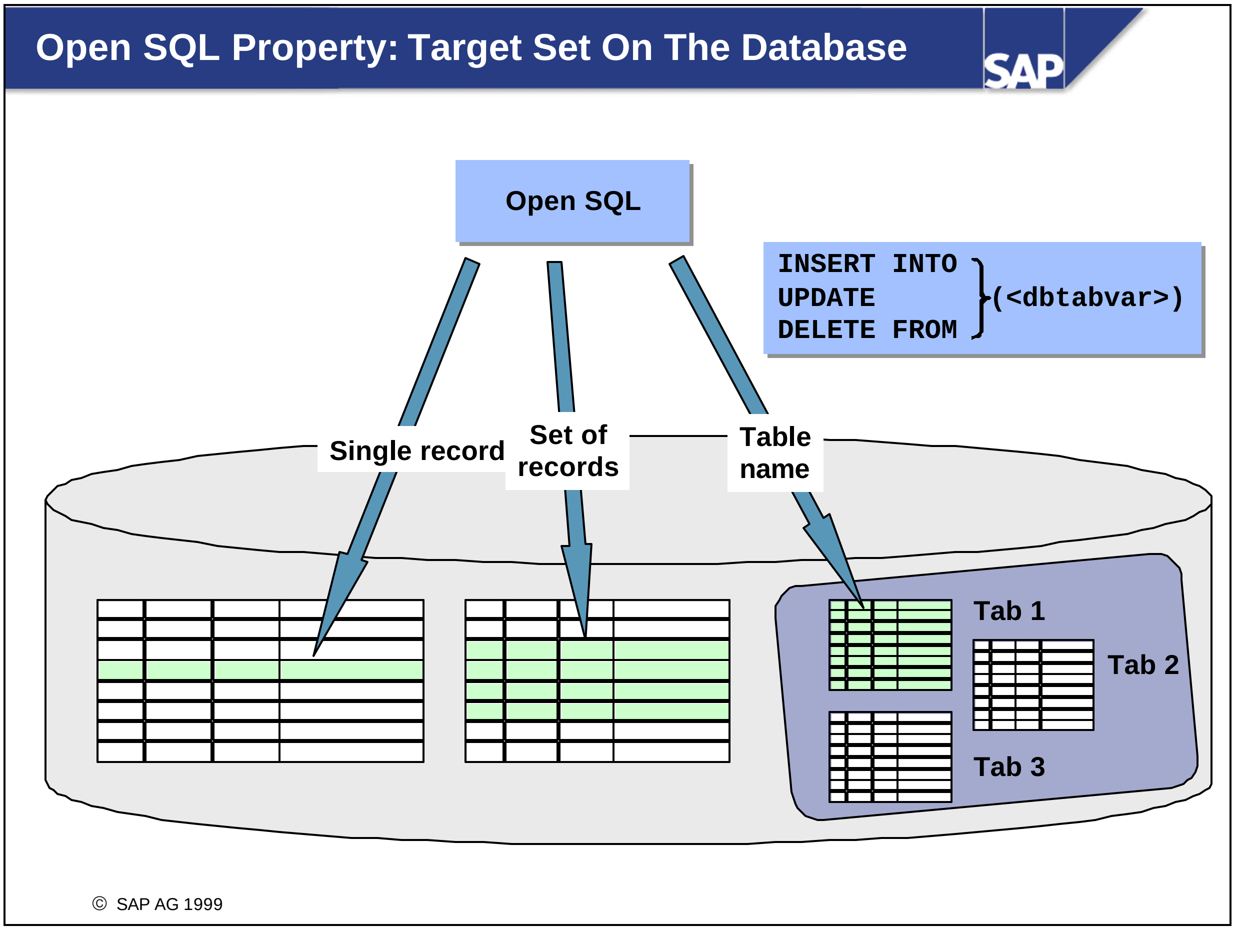
```
IMPORT FROM
   DATABASE ...
EXPORT TO
   DATABASE ...
```

- You can update databases either using ABAP's Open SQL commands, or with the database-specific commands of your database's Native SQL command set.
- You can access ABAP cluster databases using special ABAP commands.
- You can access the data in database tables using the Open SQL commands. The command set includes operations of the Data Manipulation Language (DML). The Data Definition Language (DDL) operations are not available in Open SQL, as these functions are performed by the ABAP Dictionary.
- Native SQL commands allow you to carry out both DML and DDL operations.
- The commands for ABAP cluster databases enable operations to be carried out on the data in the cluster databases. The tables themselves are created in the ABAP Dictionary as transparent tables. For general information on cluster tables, refer to the course appendix.
- For further information on Native and Open SQL, see the ABAP Editor keyword documentation for the term **SQL**.

Open SQL: Portability and Buffering



- Each time you access the database using Open SQL, the database interface of each work process (application server) converts this to a database-specific command. For this reason, the ABAP programs themselves are independent of the database used and can be transferred to other system platforms (with a different database system) without additional programming requirements.
- SAP database tables can be buffered at the application server level. The aims of buffering are to
  - Reduce the time needed to access data with read accesses. Data on the application server can be accessed more quickly than data on the database.
  - Reduce the load on the database. Reading the data from application server buffers reduces the number of database accesses.
- The buffered tables are accessed exclusively via database interface mechanisms.
- Database accesses with Native SQL enable database-specific commands to be used. This requires a detailed knowledge of the syntax in question. Programs that use Native SQL commands need additional programming after they are transported to different system environments (different database systems), since the syntax of the SQL commands generally varies from one database to the next.



Open SQL Property: Target Set On The Database

Open SQL

```
INSERT INTO ⎫
UPDATE      ⎬(<dbtabvar>)
DELETE FROM ⎭
```

Single record

Set of records

Table name

Tab 1

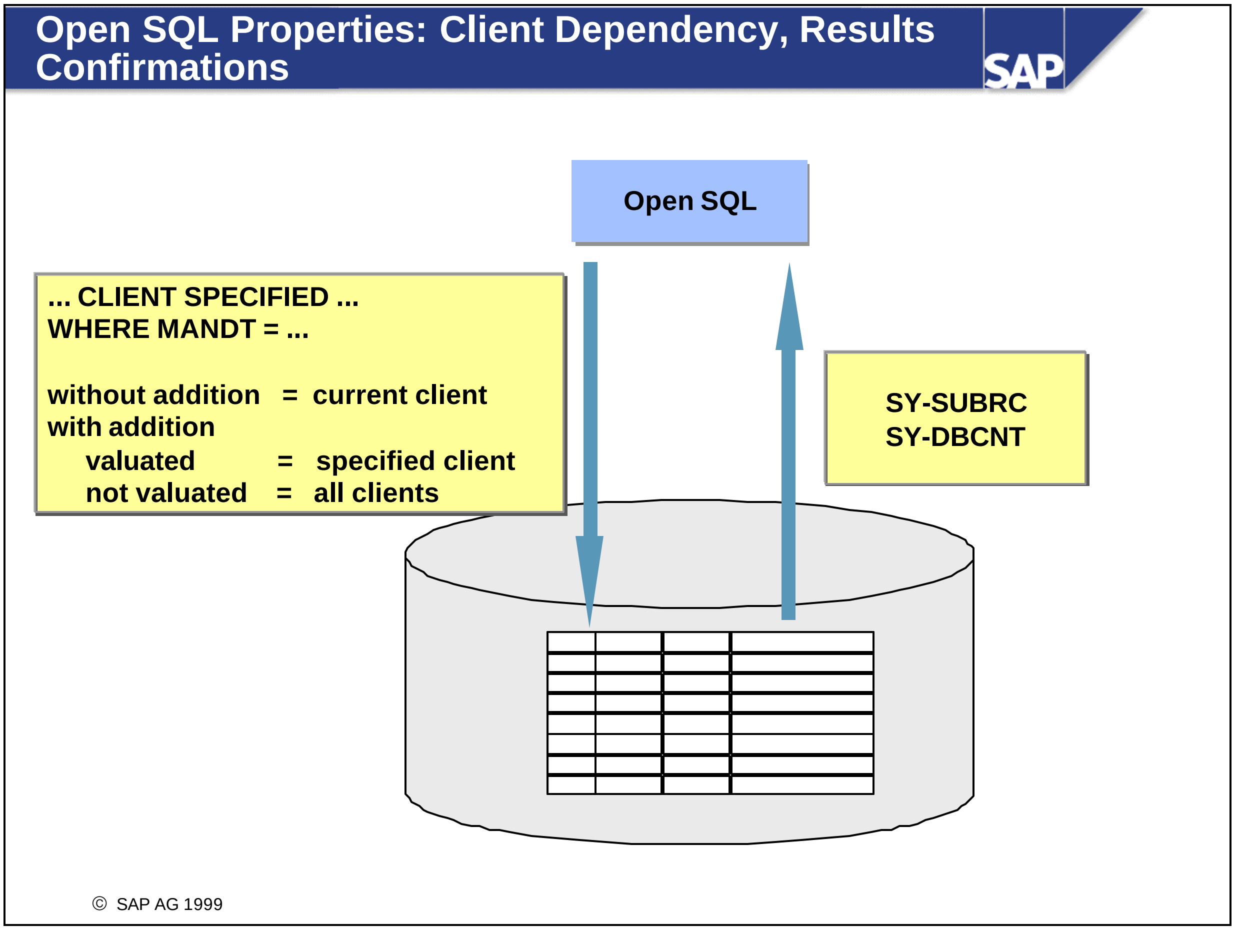
Tab 2

Tab 3

© SAP AG 1999

- The target quantity can be limited on the database using all the Open SQL commands discussed here.
- One or more rows can be processed with a SQL command. Each command also provides the option of specifying the table name dynamically.
- In addition to this, each type of operation has a syntax variant, which can be used to change individual fields in a row.
- With masked field selections (`WHERE <field1> LIKE '<search_mask>'`), note that '_' masks an individual character and '%' masks a character string of any length (in line with the SQL standard).
  Example: If the airlines Alitalia (`carrid` = 'AZ') and American Airlines (`carrid` = 'AA') offer flights in the `SFLIGHT` table, you can change the price for both airlines (and for all other airlines whose ID codes begin with 'A') to 1000 USD as follows:

```
    UPDATE sflight
                  SET  price      = '1000'
                       currency   = 'USD'
                  WHERE  carrid LIKE 'A%'.
```



Open SQL Properties: Client Dependency, Results Confirmations

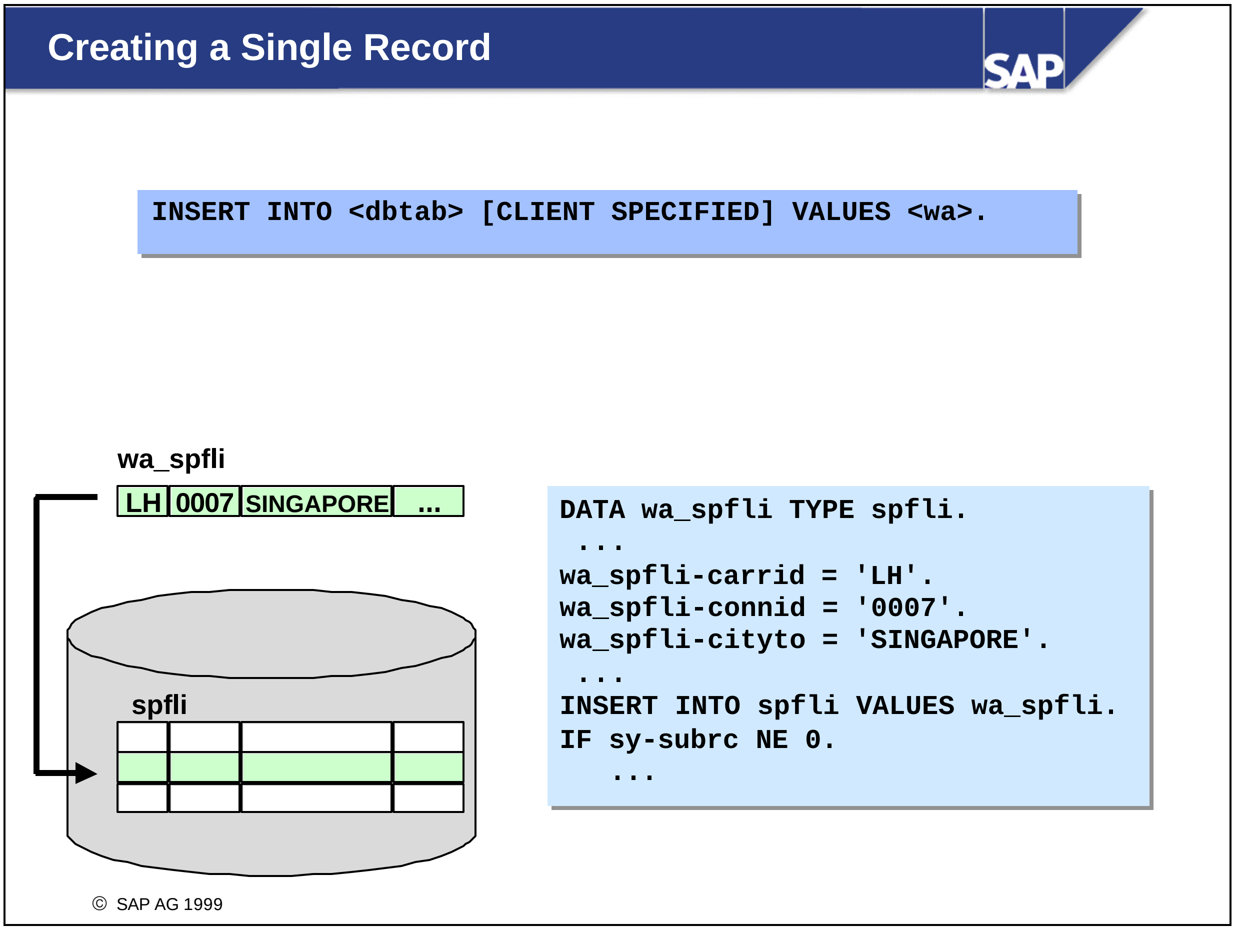
Open SQL

... CLIENT SPECIFIED ...
WHERE MANDT = ...

without addition = current client
with addition
valuated = specified client
not valuated = all clients

SY-SUBRC
SY-DBCNT

- For all Open SQL commands, you can edit data in the current client (standard). To do so, you do not specify any command additions and leave the client field **non valuated.**
- If you want to edit data from other clients explicitly, use the SQL command with the addition `CLIENT SPECIFIED` and enter the number of the client in which the SQL operation is to be carried out in the `WHERE` clause of the command.
- All Open SQL commands return confirmation of the success or failure of the database operation in the form of a return code. This is always returned by the database interface in the `sy-subrc` system field. The return code '0' (zero) always means that the operation has been completed successfully. All other values mean that errors have occurred. For further details, please refer to the keyword documentation for the command in question.
- In addition, the `sy-dbcnt` system field displays the number of records for which the desired database operation was actually carried out.
- Note that Open SQL commands do not perform any automatic authorization checks. You need to carry these out separately (see unit *Authorization Checks*).

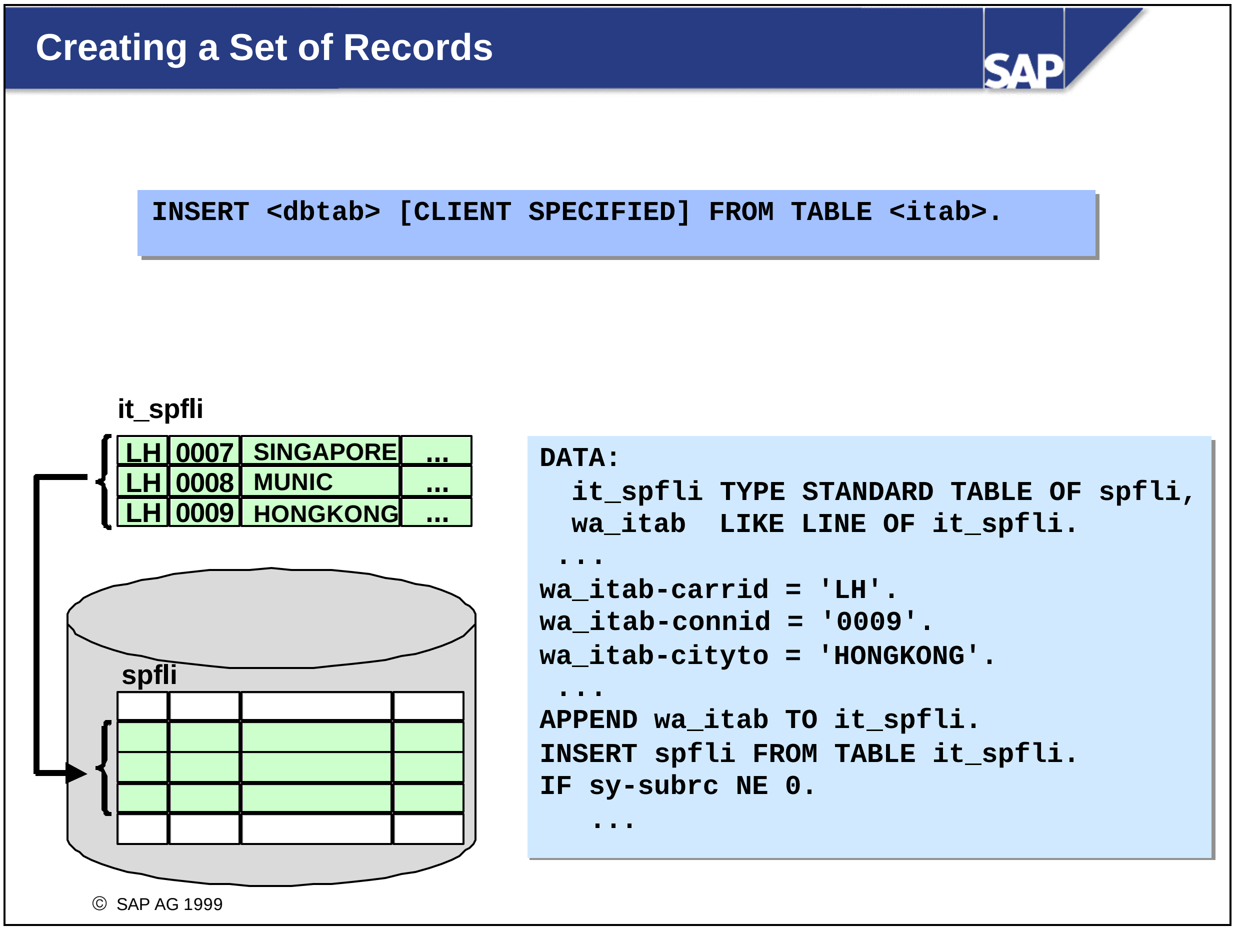


Creating a Single Record

```
INSERT INTO <dbtab> [CLIENT SPECIFIED] VALUES <wa>.
```

```
DATA wa_spfli TYPE spfli.
 ...
wa_spfli-carrid = 'LH'.
wa_spfli-connid = '0007'.
wa_spfli-cityto = 'SINGAPORE'.
 ...
INSERT INTO spfli VALUES wa_spfli.
IF sy-subrc NE 0.
   ...
```

- To insert a new row in a database table, enter the command `INSERT INTO <dbtab> VALUES <wa>.` To do so, you must specify the data to be written to the database in the `<wa>` structure (key and non-key fields) before the command.
- The `<wa>` structure must be typed according to the row structure of the database table to be updated `(DATA <wa> TYPE <dbtab>).`
- Rows can also be inserted for views. However, there are two restrictions here: The view may only contain fields from one table and must be created in the ABAP Dictionary with maintenance status 'read and change'.
- The `INSERT` command has the two return codes '0' (row could be inserted) and '4' (row could not be inserted, as a row with the same key already exists).
- The following ABAP short forms exist:
  - Short form 1: `INSERT <dbtab> [CLIENT SPECIFIED] FROM <wa>.`
  - Short form 2: `INSERT <dbtab> [CLIENT SPECIFIED].`
- The second short form requires that the data, which is to be added to the database, be available in a table work area called `<dbtab>.` This table work area must be declared in the program with `TABLES: <dbtab>`.
- The second short form is forbidden using ABAP Objects.



Creating a Set of Records

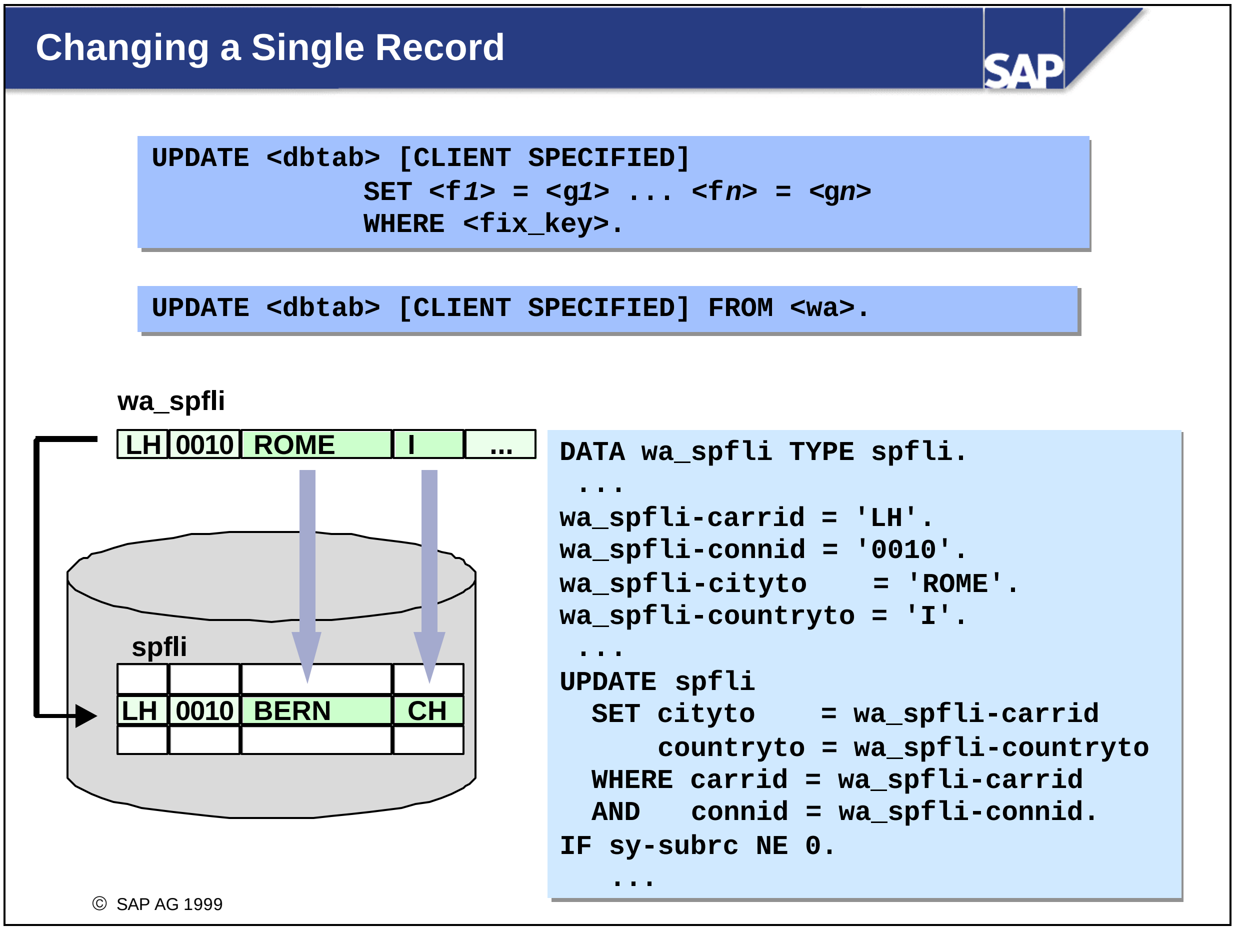
```
INSERT <dbtab> [CLIENT SPECIFIED] FROM TABLE <itab>.
```

it_spfli

LH	0007	SINGAPORE	...
LH	0008	MUNIC	...
LH	0009	HONGKONG	...

```
DATA:
  it_spfli TYPE STANDARD TABLE OF spfli,
  wa_itab  LIKE LINE OF it_spfli.
 ...
wa_itab-carrid = 'LH'.
wa_itab-connid = '0009'.
wa_itab-cityto = 'HONGKONG'.
 ...
APPEND wa_itab TO it_spfli.
INSERT spfli FROM TABLE it_spfli.
IF sy-subrc NE 0.
   ...
```

- You can use the command `INSERT <dbtab> FROM TABLE <itab>` to create several rows in a database table. The internal table `<itab>` contains the data in the rows that are to be inserted. The internal table `<itab>` must be typed to row type `<dbtab>`.
- If the operation can be carried on all rows, the return code `sy-subrc` returns the value zero. If even one data record cannot be created, a runtime error is triggered. This means that no data record is inserted by the command.
- You can prevent the runtime error occurring with the addition `ACCEPTING DUPLICATE KEYS`. In the event of an error, the addition sets return code 4 instead of the runtime error. The data records that were successfully inserted are not rejected (no DB ROLLBACK)
- The `sy-dbcnt` system field contains the number of rows that were successfully inserted in the database.



Changing a Single Record

```
UPDATE <dbtab> [CLIENT SPECIFIED]
             SET <f1> = <g1> ... <fn> = <gn>
             WHERE <fix_key>.
```

```
UPDATE <dbtab> [CLIENT SPECIFIED] FROM <wa>.
```

```
DATA wa_spfli TYPE spfli.
 ...
wa_spfli-carrid = 'LH'.
wa_spfli-connid = '0010'.
wa_spfli-cityto    = 'ROME'.
wa_spfli-countryto = 'I'.
 ...
UPDATE spfli
  SET cityto    = wa_spfli-carrid
      countryto = wa_spfli-countryto
  WHERE carrid = wa_spfli-carrid
  AND   connid = wa_spfli-connid.
IF sy-subrc NE 0.
   ...
```

- The command UPDATE <dbtab> SET <f1> = <g1> ... <fn> = <gn> WHERE <fix_key> allows you to change data in one row in a database table. After the SET command, you specify the fields in the rows whose values you want to change and the key of the database row in the WHERE clause. The key must be specified completely; each individual field must be specified with the relational operator '='.
- For numeric fields, the data following the SET command may be specified in the form of a "calculation rule" carried out on the database: f = g, f = f + g, f = f - g.
- The command has the two return codes 0 (row could be changed) and 4 (row could not be changed).
- Rows can also be changed in views. However, there are two restrictions here: The view may only contain fields from one table and must be created in the ABAP Dictionary with maintenance status 'read and change'.
- The following short forms exist:
  - Short form 1: UPDATE <dbtab> FROM <wa>.
  - Short form 2: UPDATE dbtab.
- With short form 1, the entire data record must have been written to the <wa> structure (key and non-key fields) before it is called up. The <wa> structure must be typed to the row type of the database table (DATA: <wa> TYPE <dbtab>. The short form is not field-specific, but sends the entire structure to the database interface.
- The second short form requires that the data, which is to be updated in the database, be available in a table work area called <dbtab>. This table work area must be declared in the program with TABLES: <dbtab>.
- The second short form is forbidden using ABAP Objects

- If identical changes are to be made to several rows in a table, use the syntax specified on the slide. Using the `WHERE` clause, specify the rows for which the change is to be carried out.
- The following "calculations" are also possible here for the numerical fields to be changed: `f = g`, `f = f + g`, `f = f - g`.
- The command has the two return codes 0 (at least one row has been changed) and 4 (no rows could be updated).
- The `sy-dbcnt` field contains the number of updated rows in the database table.
- There is a short form `UPDATE <dbtab> SET <f1> = <g1> ...<fn> = <gn>.` This requires that a table work area has been created with `TABLES <dbtab>` and changes the fields specified after `SET` for all rows in the current client.
- The short form is forbidden using ABAP Objects.

- If changes are to be made to several rows in a database table, whereby the changes for each row are determined via an internal table, use the syntax `UPDATE <dbtab> FROM TABLE <itab>.` Here, the internal table `<itab>` contains the data of the rows to be changed (key and non-key fields). The internal table `<itab>` must have the row type `<dbtab>.`
- The command has the two return codes 0 (all rows have been updated) and 4 (at least one row of the internal table was not used to update the database; the remaining rows have been updated).
- The system field `sy-dbcnt` contains the number of rows that have been updated in the database.

- The `MODIFY` command is SAP-specific. It includes the operations of the two commands `INSERT ...` and `UPDATE ...`:
  - In other words, `MODIFY <dbtab> FROM <wa>` inserts a new data record if the structure `<wa>` specifies a data record that does not yet exist in the database.
  - If the `<wa>` structure specifies an existing data record, the command updates the row in question.
- Using the different syntax variants, you can make changes to individual rows, make similar changes to several rows, and carry out operations on sets of records.
- All variants of the `MODIFY ..` syntax have the two return codes 0 (all rows were inserted or updated) and 4 (at least one line was not inserted or updated).
- The operation can also be carried out on views. However, there are two restrictions here: The view may only contain fields from one table and must be created in the ABAP Dictionary with maintenance status 'read and change'.
- The field `sy-dbcnt` contains the number of rows that have been changed or inserted in the database.

- The command `DELETE FROM <dbtab> WHERE <fixkey>` enables one row to be deleted from a database table. In the `WHERE` clause, specify all the key fields with the relational operator '='.
- The command has the two return codes 0 (row has been deleted) and 4 (row has not been deleted).
- A row can also be deleted from views. However, there are two restrictions here: The view may only contain fields from one table and must be created in the ABAP Dictionary with maintenance status 'read and change'.
- The following short forms exist:
  - Short form 1: `DELETE <dbtab> [CLIENT SPECIFIED] FROM <wa>,`
  - Short form 2: `DELETE <dbtab> [CLIENT SPECIFIED].`
- Short form 1 requires that the `<wa>` structure has been filled with the key fields of the row to be deleted before it is called up. The structure must have the row type `<dbtab>.`
- Short form 2 requires that the key fields of the row to be deleted be available in a table work area called `<dbtab>.` This table work area must be declared in the program with `TABLES: <dbtab>`.
- The second short form is forbidden using ABAP Objects.

- The command `DELETE FROM <dbtab> WHERE <condition>` enables several rows to be deleted from a database table. Here, you can specify the rows that are to be deleted with the `WHERE` clause.
- The command has the two return codes 0 (at least one row was deleted) and 4 (no rows were deleted).
- The system field `sy-dbcnt` contains the number of rows that have been updated on the database.

- To delete several specific rows from a database using a database operation, use the statement `DELETE <dbtab> FROM TABLE <itab>.` The internal table `<itab>` here contains the key fields for the rows that are to be deleted. The internal table `<itab>` must have the row type `<dbtab>.`
- The command has the two return codes 0 (all rows have been deleted) and 4 (at least one row could not be deleted, the rest have been deleted).
- There are two ways of deleting all the rows from a table in the current client:
  - Either `DELETE FROM <dbtab> WHERE <field> IN <itab>` with a blank internal table `<itab>`
  - or `DELETE FROM <dbtab> WHERE <field> LIKE '%'`.
- The number of rows deleted from the database is shown in the system field `sy-dbcnt`.

- If you receive a return code other than zero from the database interface in response to an Open SQL statement for changing data in the database, you should make sure that the database is reset to the status it had before the change attempt was made. You can do this by means of a database rollback. The database rollback undoes any changes made to the current database LUW (see the next unit).
- For return codes from DB change statements (Open SQL), the most suitable means of triggering a database rollback is to send a termination dialog message (A message or X message). This triggers a database rollback and terminates the associated program.
- All other message types (E,W, I) also involve a dialog but **do not** trigger a database rollback.
- You can also trigger a database rollback using the ABAP statement ROLLBACK WORK (without terminating the program at the same time). You should **not use** the ROLLBACK WORK statement directly, unless you do not want to reset the program context (unlike a termination dialog message) (see unit *Organizing Database Updates)*.

Unit: Keywords for DB Updates
Topic: Single Record Changes

At the conclusion of these exercises, you will be able to:

- Insert and modify single records in database tables.

The program **SAPBC414T_CREATE_CUSTOMER_01** enables new customer data to be entered in screen 100.

Extend this program to include the database dialog:
After the function code SAVE is triggered (e.g. by clicking the *Save* icon), the customer data is to be written to the **database table SCUSTOM**.

Program:	SAPMZ##_CUSTOMER1
Transaction code:	Z##_CUSTOMER1
Template:	SAPBC414**T**_CREATE_CUSTOMER_01
Model solution:	SAPBC414**S**_CREATE_CUSTOMER_01

1-1 Copy the program template **SAPBC414T_CREATE_CUSTOMER_01** with all sub-objects to **SAPMZ##_CUSTOMER1** (## is the group number). Assign transaction code **Z##_CUSTOMER1** to the program.

1-2 The ABAP statements for the database dialog are encapsulated in the **subroutine SAVE_SCUSTOM**. The subroutine has already been created (and is empty).

1-2-1 Insert the new customer data record in the database table SCUSTOM. The set message S015 is to be output if the new data record is inserted successfully. If the data record was not inserted successfully, the termination message A048 is to be output.

The customer data is stored in the structure SCUSTOM.

The message class BC414 is set as an addition for the PROGRAM statement and therefore is globally valid (throughout the program).

Optional Exercise

Unit: Keywords for DB Updates

Topic: Changing Data Sets

At the conclusion of these exercises, you will be able to:

- Insert and modify data sets in database tables.

In the program **SAPBC414T_UPDATE_STRAVELAG,** a list is generated that presents the data of the travel agencies maintained in the **STRAVELAG** table. The user can select the travel agency data that is to be changed on the next screen 100 by selecting one or more rows.

Extend the program to include the database dialog:
The changed data is to be saved to the **STRAVELAG** database table by clicking the *Save* icon (function code SAVE) on screen 100.

Program:	SAPMZ##_UPDATE_STRAVELAG
Template:	SAPBC414**T**_UPDATE_STRAVELAG
Model solution:	SAPBC414**S**_UPDATE_STRAVELAG

2-1 Copy the program template **SAPBC414T_ UPDATE_STRAVELAG** with **all** sub-objects to **SAPMZ##_UPDATE_STRAVELAG** (## is the group number). As this is a type 1 program, a transaction code is not required.

2-2 The database dialog is initiated by triggering the function code SAVE. Here, the **subroutine SAVE_CHANGES**, which contains the database dialog, is called up in the PAI module USER_COMMAND_0100 (screen 100). This subroutine has already been created (empty).

2-2-1 Save the **changed** address data to the database table STRAVELAG. When doing so, note the performance aspects. If the change is successful, the set message S030 is to be output. If it is unsuccessful, information message I048 is to be output.

The travel agency data is buffered in the **internal table ITAB_TRAVEL** (work area WA_TRAVEL). The rows in the internal table have the same structure as those in STRAVELAG, with the exception of the additional field **MARK_CHANGED** (C(1)). If the address data on the screen 100 has been changed, MARK_CHANGED has the value 'X'. Otherwise it is blank or 0.

The model solutions provided here repeat the statements of the flow logic and ABAP program parts that will be required.

The exercises for course BC414 are designed to expand on two larger programs accompanying the contents of the unit in question. For the sake of clarity, not all of the model solutions are provided with complete coding. The following procedure is used instead:

- The model solution for the activity in which a program is edited for the first time is displayed completely.
- Any model solutions that expand on this only explain flow logic, subroutines, and modules, which have changed or appear for the first time. The statements in the repeated modularization units that need to be completed in order to solve the activity are highlighted in **bold**.
- A complete version of both programs is provided in the appendix in the training folder.

The second activity in the unit on *Database Updates With Open SQL*, which is marked as *optional*, is an exception to this procedure. Since the program associated with this activity is not dealt with in the following units, the model solution for this activity is explained fully.

Unit: Keywords for DB Updates
Topic: Single Record Changes

Model Solution SAPBC414S_CREATE_CUSTOMER_01

Module Pool

```
*&---------------------------------------------------------------------*
*& Modulpool         SAPBC414S_CREATE_CUSTOMER_01                      *
*&---------------------------------------------------------------------*
INCLUDE BC414S_CREATE_CUSTOMERTOP.
INCLUDE BC414S_CREATE_CUSTOMERO01.
INCLUDE BC414S_CREATE_CUSTOMERI01.
INCLUDE BC414S_CREATE_CUSTOMER_01F01.
```

SCREEN 100

```
PROCESS BEFORE OUTPUT.
  MODULE status_0100.

PROCESS AFTER INPUT.
  MODULE exit AT EXIT-COMMAND.
  MODULE save_ok_code.
  FIELD: scustom-name MODULE mark_changed ON REQUEST.
  MODULE user_command_0100.
```

TOP Include

```
*&---------------------------------------------------------------------*
*& Include BC414S_CREATE_CUSTOMERTOP                                   *
*&---------------------------------------------------------------------*
PROGRAM  sapbc414s_create_customer MESSAGE-ID bc414.

DATA: answer, flag.
DATA: ok_code LIKE sy-ucomm, save_ok LIKE ok_code
```

```
TABLES: scustom.
```

PBO Modules

```
*------------------------------------------------------------------*
***INCLUDE BC414S_CREATE_CUSTOMERO01 .
*------------------------------------------------------------------*

*&-----------------------------------------------------------------*
*&      Module  STATUS_0100  OUTPUT
*&-----------------------------------------------------------------*
MODULE STATUS_0100 OUTPUT.
  SET PF-STATUS 'DYN_0100'.
  SET TITLEBAR 'DYN_0100'.
ENDMODULE.                 " STATUS_0100  OUTPUT
```

PAI Modules

```
*------------------------------------------------------------------*
***INCLUDE BC414S_CREATE_CUSTOMERI01 .
*------------------------------------------------------------------*

*&-----------------------------------------------------------------*
*&      Module  EXIT  INPUT
*&-----------------------------------------------------------------*
MODULE exit INPUT.
  CASE ok_code.
    WHEN 'EXIT'.
      IF sy-datar IS INITIAL AND flag IS INITIAL.
* no changes on screen 100
        LEAVE PROGRAM.
      ELSE.
        PERFORM ask_save USING answer.
        CASE answer.
          WHEN 'J'.
            ok_code = 'SAVE&EXIT'.
          WHEN 'N'.
            LEAVE PROGRAM.
          WHEN 'A'.
            CLEAR ok_code.
            SET SCREEN 100.
      ENDIF.
    WHEN 'CANCEL'.
      IF sy-datar IS INITIAL AND flag IS INITIAL.
* no changes on screen 100
        LEAVE TO SCREEN 0.
      ELSE.
        PERFORM ask_loss USING answer.
        CASE answer.
          WHEN 'J'.
            LEAVE TO SCREEN 0.
          WHEN 'N'.
            CLEAR ok_code.
            SET SCREEN 100.
        ENDCASE.
      ENDIF.
  ENDCASE.
ENDMODULE.                             " EXIT  INPUT
*&-----------------------------------------------------------------*
*&      Module  SAVE_OK_CODE  INPUT
*&-----------------------------------------------------------------*
MODULE save_ok_code INPUT.
  save_ok = ok_code.
  CLEAR ok_code.
ENDMODULE.                             " SAVE_OK_CODE  INPUT

*&-----------------------------------------------------------------*
*&      Module  USER_COMMAND_0100  INPUT
*&-----------------------------------------------------------------*
MODULE user_command_0100 INPUT.
  CASE save_ok.
    WHEN 'SAVE&EXIT'.
      PERFORM save.
      LEAVE PROGRAM.
    WHEN 'SAVE'.
      IF flag IS INITIAL.
        SET SCREEN 100.
      ELSE.
        PERFORM save.
        SET SCREEN 0.
    WHEN 'BACK'.
      IF flag IS INITIAL.
        SET SCREEN 0.
      ELSE.
        PERFORM ask_save USING answer.
        CASE answer.
          WHEN 'J'.
            PERFORM save.
            SET SCREEN 0.
          WHEN 'N'.
            SET SCREEN 0.
          WHEN 'A'.
            SET SCREEN 100.
        ENDCASE.
      ENDIF.
  ENDCASE.
ENDMODULE.                             " USER_COMMAND_0100  INPUT


*&------------------------------------------------------------------*
*&      Module  MARK_CHANGED  INPUT
*&------------------------------------------------------------------*
MODULE mark_changed INPUT.
* set flag to mark changes were made on screen 100
  flag = 'X'.
ENDMODULE.                             " MARK_CHANGED  INPUT
```

FORM Routines

```
*-------------------------------------------------------------------*
***INCLUDE BC414S_CREATE_CUSTOMER_01F01 .
*-------------------------------------------------------------------*


*&------------------------------------------------------------------*
*&      Form  NUMBER_GET_NEXT
*&------------------------------------------------------------------*
*       -->P_WA_SCUSTOM  text
*-------------------------------------------------------------------*
FORM number_get_next USING p_scustom LIKE scustom.
  DATA: return TYPE inri-returncode.
* get next free number in the number range '01'
* of number range object'SBUSPID'
  CALL FUNCTION 'NUMBER_GET_NEXT'
       EXPORTING
            nr_range_nr = '01'
            object      = 'SBUSPID'
       IMPORTING
            number      = p_scustom-id
            returncode  = return
       EXCEPTIONS
            OTHERS      = 1.
  CASE sy-subrc.
    WHEN 0.
      CASE return.
        WHEN 1.
* number of remaining numbers critical
          MESSAGE s070.
        WHEN 2.
* last number
          MESSAGE s071.
        WHEN 3.
* no free number left over
          MESSAGE a072.
      ENDCASE.
    WHEN 1.
* internal error
      MESSAGE a073 WITH sy-subrc.
  ENDCASE.
ENDFORM.                                          " NUMBER_GET_NEXT
```

```
*&------------------------------------------------------------------*
*&      Form  ASK_SAVE
*&------------------------------------------------------------------*
*       -->P_ANSWER  text
*-------------------------------------------------------------------*
FORM ask_save USING p_answer.
  CALL FUNCTION 'POPUP_TO_CONFIRM_STEP'
       EXPORTING
            textline1 = 'Data has been changed.'(001)
            textline2 = 'Save before leaving transaction?'(002)
            titel     = 'Create Customer'(003)
       IMPORTING
            answer    = p_answer.
ENDFORM.                               " ASK_SAVE

*&------------------------------------------------------------------*
*&      Form  ASK_LOSS
*&------------------------------------------------------------------*
*       -->P_ANSWER  text
*-------------------------------------------------------------------*
FORM ask_loss USING p_answer.
  CALL FUNCTION 'POPUP_TO_CONFIRM_LOSS_OF_DATA'
       EXPORTING
            textline1 = 'Continue?'(004)
            titel     = 'Create Customer'(003)
       IMPORTING
            answer    = p_answer.
ENDFORM.                               " ASK_LOSS


*&------------------------------------------------------------------*
*&      Form  ENQ_SCUSTOM
*&------------------------------------------------------------------*
FORM enq_scustom.
  CALL FUNCTION 'ENQUEUE_ESCUSTOM'
       EXPORTING
            id             = scustom-id
       EXCEPTIONS
            foreign_lock   = 1
            system_failure = 2
            OTHERS         = 3.
  CASE sy-subrc.
    WHEN 0.
    WHEN 1.
      MESSAGE e060.
    WHEN OTHERS.
      MESSAGE e063 WITH sy-subrc.
  ENDCASE.
ENDFORM.                               " ENQ_SCUSTOM


*&-----------------------------------------------------------------*
*&      Form  DEQ_ALL
*&-----------------------------------------------------------------*
FORM deq_all.
  CALL FUNCTION 'DEQUEUE_ALL'.
ENDFORM.                               " DEQ_ALL
*&-----------------------------------------------------------------*
*&      Form  SAVE
*&-----------------------------------------------------------------*
FORM save.
* get SCUSTOM-ID from number range object SBUSPID
  PERFORM number_get_next USING scustom.
* save new customer
  PERFORM save_scustom.
ENDFORM.                               " SAVE


*&-----------------------------------------------------------------*
*&      Form  SAVE_SCUSTOM
*&-----------------------------------------------------------------*
FORM save_scustom.
  INSERT INTO scustom VALUES scustom.
  IF sy-subrc <> 0.
* insertion of dataset in DB-table not possible
    MESSAGE a048.
  ELSE.
* insertion successfull
```

```
  ENDIF.
ENDFORM.                                          " SAVE_SCUSTOM
```

Solutions

Unit: Keywords for DB Updates

Topic: Changing Data Sets

Model Solution SAPBC414S_UPDATE_STRAVELAG

Module Pool

```
*&---------------------------------------------------------------------*
*& Modulpool         SAPBC414S_UPDATE_STRAVELAG                        *
*&---------------------------------------------------------------------*
INCLUDE bc414s_update_stravelagtop.
INCLUDE bc414s_update_stravelagf01.
INCLUDE bc414s_update_stravelago01.
INCLUDE bc414s_update_stravelagi01.
INCLUDE bc414s_update_stravelage01.
```

SCREEN 100

```
PROCESS BEFORE OUTPUT.
  MODULE STATUS_0100.
* fill table control (only agencies, marked on list)
  LOOP AT ITAB_TRAVEL INTO WA_TRAVEL WITH CONTROL TC_STRAVELAG.
    MODULE TRANS_TO_DYNPRO.
  ENDLOOP.
*

PROCESS AFTER INPUT.
  MODULE EXIT AT EXIT-COMMAND.
  LOOP AT ITAB_TRAVEL.
    CHAIN.
     FIELD: STRAVELAG-STREET, STRAVELAG-POSTBOX, STRAVELAG-POSTCODE,
            STRAVELAG-CITY, STRAVELAG-COUNTRY, STRAVELAG-REGION,
            STRAVELAG-TELEPHONE, STRAVELAG-URL, STRAVELAG-LANGU
```

```
* mark datasets, that were changed in table control (subset of all
* agencies, thet were shown on table control)
      MODULE SET_MARKER ON CHAIN-REQUEST.
    ENDCHAIN.
  ENDLOOP.
  MODULE SAVE_OK_CODE.
  MODULE USER_COMMAND_0100.
```

TOP Include

```
*&-----------------------------------------------------------------*
*& Include BC414S_UPDATE_STRAVELAGTOP                              *
*&-----------------------------------------------------------------*
PROGRAM  sapbc414s_update_stravelag NO STANDARD PAGE HEADING
                                    LINE-SIZE 120
                                    LINE-COUNT 10
                                    MESSAGE-ID bc414.

* Line type definition for internal table itab_travel
TYPES: BEGIN OF stravel_type.
        INCLUDE STRUCTURE stravelag.
TYPES:            mark_changed,
       END OF stravel_type.

* Standard internal table for travel agency data buffering and
* corresponding workarea
DATA: itab_stravelag LIKE STANDARD TABLE OF stravelag
      WITH NON-UNIQUE KEY agencynum,
      wa_stravelag TYPE stravelag.

* Workarea for transport of field values from/to screen 100
TABLES: stravelag.

* Transport function code from screen 100
DATA: ok_code TYPE sy-ucomm, save_ok LIKE ok_code.

* Table control structure on screen 100
CONTROLS: tc_stravelag TYPE TABLEVIEW USING SCREEN '0100'.

* Internal table to collect marked list entries, corresponding
* workarea
DATA: itab_travel TYPE STANDARD TABLE OF stravel_type
      WITH NON-UNIQUE KEY agencynum,
      wa_travel TYPE stravel_type.

* Mark field displayed as checkbox on list
DATA: mark.
```

```
* Flags:
DATA: flag,         "changes performed on table control
      modify_list.  "modification of list buffer is neccessary

* Positions of fields on list
CONSTANTS: pos1 TYPE i VALUE   1,
           pos2 TYPE i VALUE   3,
           pos3 TYPE i VALUE  14,
           pos4 TYPE i VALUE  40,
           pos5 TYPE i VALUE  71,
           pos6 TYPE i VALUE  82,
           pos7 TYPE i VALUE 108.
```

PBO Modules

```
*-------------------------------------------------------------------*
***INCLUDE BC414S_UPDATE_STRAVELAGO01 .
*-------------------------------------------------------------------*

*&-----------------------------------------------------------------*
*&      Module  STATUS_0100  OUTPUT
*&-----------------------------------------------------------------*
MODULE status_0100 OUTPUT.
  SET PF-STATUS 'DYNPRO'.
  SET TITLEBAR 'DYNPRO'.
ENDMODULE.                             " STATUS_0100  OUTPUT


*&-----------------------------------------------------------------*
*&      Module  TRANS_TO_DYNPRO  OUTPUT
*&-----------------------------------------------------------------*
MODULE trans_to_dynpro OUTPUT.
* Field transport to screen
  MOVE-CORRESPONDING wa_travel TO stravelag.
ENDMODULE.                             " TRANS_TO_DYNPRO  OUTPUT
```

PAI Modules

```
*-------------------------------------------------------------------*
***INCLUDE BC414S_UPDATE_STRAVELAGI01 .
*-------------------------------------------------------------------*

*&-----------------------------------------------------------------*
*&      Module  USER_COMMAND_0100  INPUT
*&-----------------------------------------------------------------*
MODULE user_command_0100 INPUT.
  CASE save_ok.
    WHEN 'SAVE'.
      IF flag IS INITIAL.
* enries on table control not changed.
        SET SCREEN 0.
```

```
* at least one field on table control changed
        PERFORM save_changes.
        SET SCREEN 0.
      ENDIF.
  ENDCASE.
ENDMODULE.                             " USER_COMMAND_0100  INPUT


*&-----------------------------------------------------------------*
*&      Module  SAVE_OK_CODE  INPUT
*&-----------------------------------------------------------------*
MODULE save_ok_code INPUT.
  save_ok = ok_code.
  CLEAR: ok_code.
ENDMODULE.                             " SAVE_OK_CODE  INPUT


*&-----------------------------------------------------------------*
*&      Module  EXIT  INPUT
*&-----------------------------------------------------------------*
MODULE exit INPUT.
  CASE ok_code.
    WHEN 'CANCEL'.
      IF sy-datar IS INITIAL AND flag IS INITIAL.
* no changes performed on screen
        LEAVE TO SCREEN 0.
      ELSE.
* at least one field on table control changed.
        PERFORM popup_to_confirm_loss_of_data.
      ENDIF.
  ENDCASE.
ENDMODULE.                             " EXIT  INPUT


*&-----------------------------------------------------------------*
*&      Module  SET_MARKER  INPUT
*&-----------------------------------------------------------------*
MODULE set_marker INPUT.
  MOVE-CORRESPONDING stravelag TO wa_travel.
```

```
* mark datasets in internal table as modified
  MODIFY TABLE itab_travel FROM wa_travel.
* at least one dataset is modified in table control
  flag = 'X'.
ENDMODULE.                             " SET_MARKER  INPUT
```

Events

```
*-------------------------------------------------------------------*
*   INCLUDE BC414S_UPDATE_STRAVELAGE01                              *
*-------------------------------------------------------------------*

*&------------------------------------------------------------------*
*&   Event START-OF-SELECTION
*&------------------------------------------------------------------*
START-OF-SELECTION.
* Read data from STRAVELAG into internal table ITAB_STRAVELAG
  PERFORM read_data USING itab_stravelag.
* Write data from ITAB_STRAVELAG on list
  PERFORM write_data.


*&------------------------------------------------------------------*
*&   Event TOP-OF-PAGE
*&------------------------------------------------------------------*
TOP-OF-PAGE.
* Write page title and page heading
  PERFORM write_header.


*&------------------------------------------------------------------*
*&   Event END-OF-SELECTION
*&------------------------------------------------------------------*
END-OF-SELECTION.
* Set PF-Status and Title of list
  SET PF-STATUS 'LIST'.
  SET TITLEBAR 'LIST'.


*&------------------------------------------------------------------*
*&   Event AT USER-COMMAND
*&------------------------------------------------------------------*
AT USER-COMMAND.
  CLEAR: modify_list, flag, itab_travel.
* Collect data corresponding to marked lines into internal table
  PERFORM loop_at_list USING itab_travel.
```

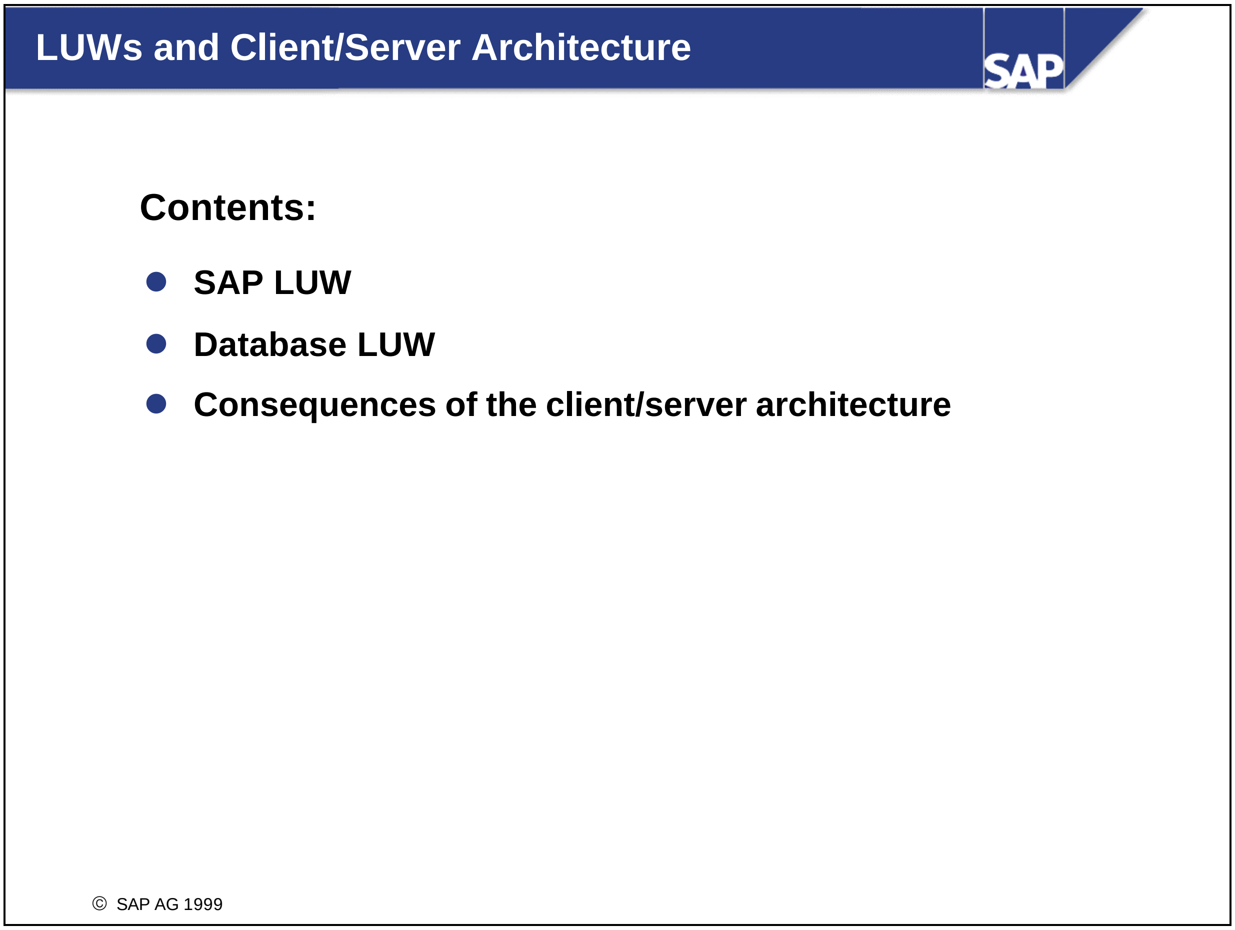
```
* Call screen if any line on list was marked
  CHECK NOT itab_travel IS INITIAL.
  PERFORM call_screen.
* Modify list buffer if database table was modified -> submit report
  CHECK NOT modify_list IS INITIAL.
  SUBMIT (sy-cprog).
```

FORM Routines

```
*------------------------------------------------------------------*
***INCLUDE BC414S_UPDATE_STRAVELAGF01 .
*------------------------------------------------------------------*

*&-----------------------------------------------------------------*
*&      Form  READ_DATA
*&-----------------------------------------------------------------*
*      -->P_ITAB_STRAVELAG  text
*------------------------------------------------------------------*
FORM read_data USING p_itab_stravelag LIKE itab_stravelag.
  SELECT * FROM stravelag
           INTO CORRESPONDING FIELDS OF TABLE p_itab_stravelag.
ENDFORM.                               " READ_DATA


*&-----------------------------------------------------------------*
*&      Form  WRITE_DATA
*&-----------------------------------------------------------------*
FORM write_data.
  LOOP AT itab_stravelag INTO wa_stravelag.
    WRITE AT: /pos1 mark AS CHECKBOX,
               pos2 wa_stravelag-agencynum COLOR COL_KEY,
               pos3 wa_stravelag-name,
               pos4 wa_stravelag-street,
               pos5 wa_stravelag-postcode,
               pos6 wa_stravelag-city,
               pos7 wa_stravelag-country.
    HIDE: wa_stravelag.
  ENDLOOP.
ENDFORM.                               " WRITE_DATA


*&-----------------------------------------------------------------*
*&      Form  WRITE_HEADER
*&-----------------------------------------------------------------*
FORM write_header.
  WRITE: / 'Travel agency data'(007), AT sy-linsz sy-pagno.
  ULINE.
  FORMAT COLOR COL_HEADING.
  WRITE AT: /pos2 'Agency'(001),
             pos3 'Name'(002),
             pos4 'Street'(003),
             pos5 'Postal Code'(004),
             pos6 'City'(005),
             pos7 'Country'(006).
  ULINE.
ENDFORM.                               " WRITE_HEADER
```

```
*&------------------------------------------------------------------*
*&      Form  LOOP_AT_LIST
*&------------------------------------------------------------------*
*      -->P_ITAB_AGNECYNUM  text
*-------------------------------------------------------------------*
FORM loop_at_list USING p_itab_travel LIKE itab_travel.
  DO.
    CLEAR: mark.
    READ LINE sy-index FIELD VALUE mark.
    IF sy-subrc <> 0.
      EXIT.
    ENDIF.
    CHECK NOT mark IS INITIAL.
    APPEND wa_stravelag TO p_itab_travel.
  ENDDO.
ENDFORM.                               " LOOP_AT_LIST


*&------------------------------------------------------------------*
*&      Form  CALL_SCREEN
*&------------------------------------------------------------------*
FORM call_screen.
* Initialize table control on screen
  REFRESH CONTROL 'TC_STRAVELAG' FROM SCREEN '0100'.
* Show screen in modal dialog box.
  CALL SCREEN 100 STARTING AT  5  5
                  ENDING   AT 80 15.
ENDFORM.                               " CALL_SCREEN


*&------------------------------------------------------------------*
*&      Form  POPUP_TO_CONFIRM_LOSS_OF_DATA
*&------------------------------------------------------------------*
FORM popup_to_confirm_loss_of_data.
  DATA answer.
  CALL FUNCTION 'POPUP_TO_CONFIRM_LOSS_OF_DATA'
       EXPORTING
            textline1 = 'Cancel processing of travel agencies?'(008)
            titel     = 'Cancel processing'(009)
       IMPORTING
            answer    = answer.
  CASE answer.
    WHEN 'J'.
      LEAVE TO SCREEN 0.
    WHEN 'N'.
      LEAVE TO SCREEN '0100'.
  ENDCASE.
ENDFORM.                               "
POPUP_TO_CONFIRM_LOSS_OF_DATA
```

```
*&------------------------------------------------------------------*
*&      Form  SAVE_CHANGES
*&------------------------------------------------------------------*
FORM save_changes.
* declare internal table and workarea of same linetype as DB table
  DATA: itab TYPE STANDARD TABLE OF stravelag,
        wa LIKE LINE OF itab.
* search for datasets changed on the screen
  LOOP AT itab_travel INTO wa_travel
       WHERE mark_changed = 'X'.
* fill workarea fitting to DB table
    MOVE-CORRESPONDING wa_travel TO wa.
* fill corresponding internal table
    APPEND wa TO itab.
  ENDLOOP.
* mass update on stravelag -> best performance
  UPDATE stravelag FROM TABLE itab.
* check success
  IF sy-subrc = 0.
* all datasets are successfully updated
    MESSAGE s030.
  ELSE.
* at least one dataset from the internal table could not be updated
* on the database table
    MESSAGE i048.
  ENDIF.
* Flag: List does not show correct data any more
  modify_list = 'X'.
ENDFORM.                               " SAVE_CHANGES
```

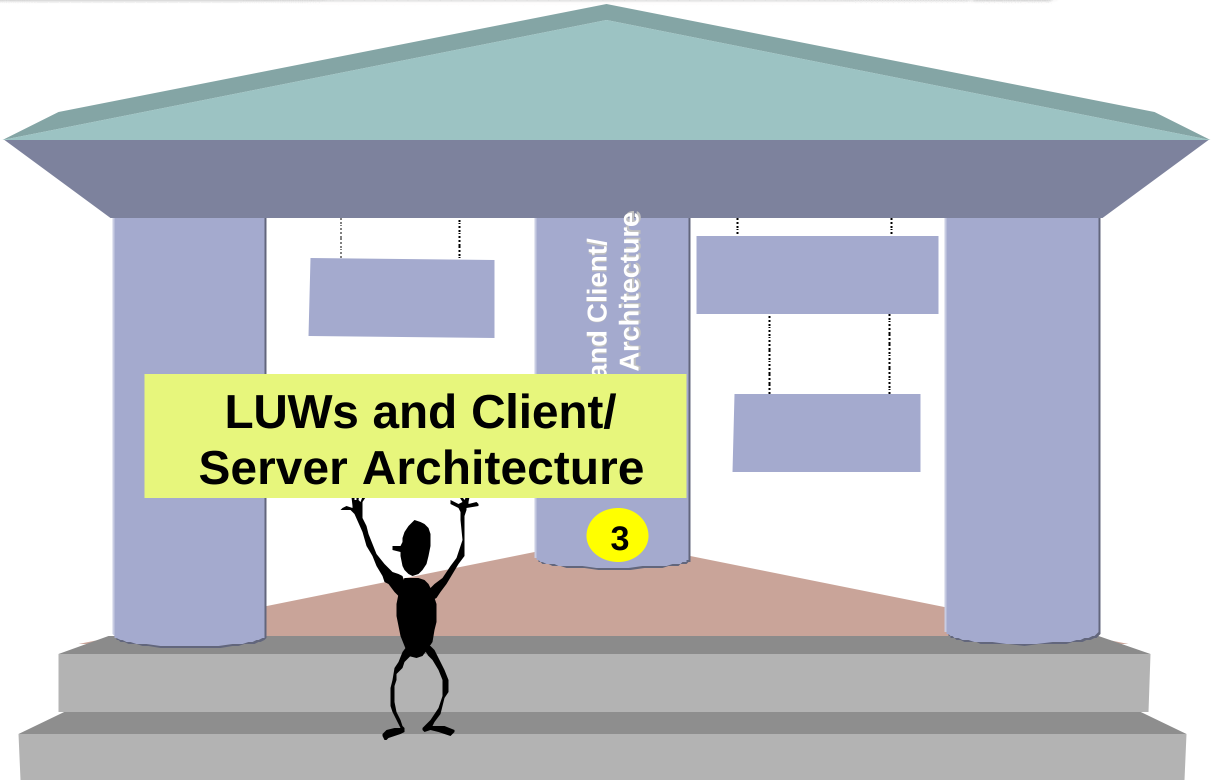


LUWs and Client/Server Architecture

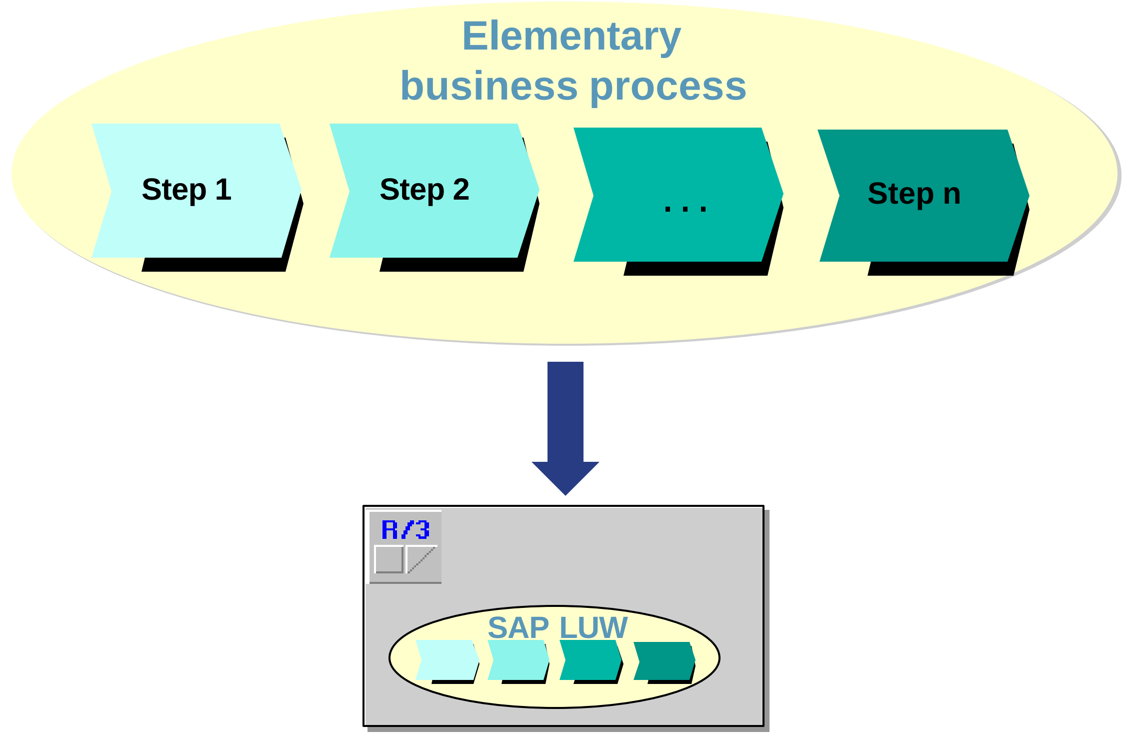
Contents:

- SAP LUW
- Database LUW
- Consequences of the client/server architecture

Course Overview Diagram - LUWs

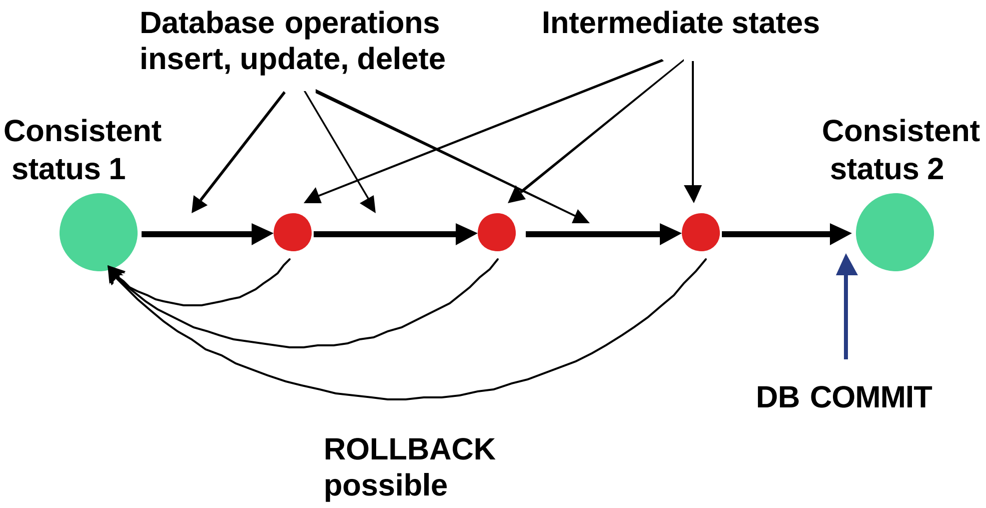


SAP Logical Unit of Work (SAP LUW): Preliminary Definition



- An **SAP logical unit of work** (LUW) is a functionally complete set of steps within a business process in the R/3 System.
- The process steps must be logically related.
- SAP LUWs work on an all-or-nothing principle: Either all or none of its steps are carried out.
- The business process to be mapped must be basic. For example, you would not have a single SAP LUW consisting of all of the steps between a customer processing an order and an invoice being produced. Instead, you would split this up into separate parts, each of which would then be represented in the R/3 System by its own LUW. What constitutes an "elementary" process depends on the overall process and how you have modeled it.
- For further information, see the ABAP Editor keyword documentation for the term **transaction processing** .

Database LUW



Database operations
insert, update, delete

Intermediate states

Consistent
status 1

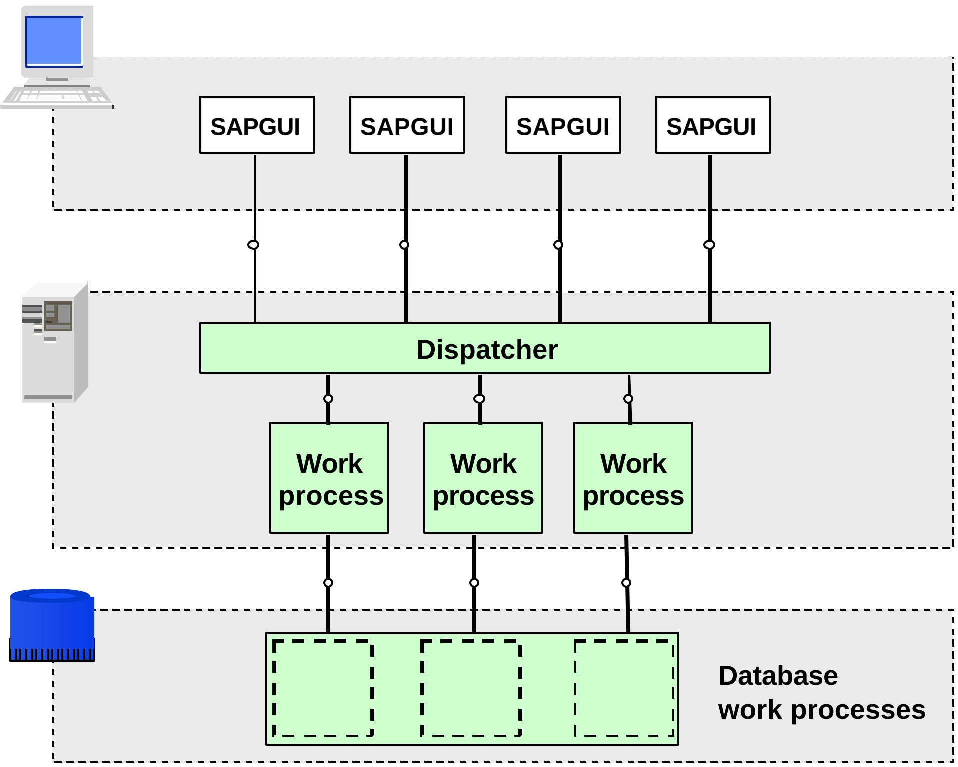
Consistent
status 2

DB COMMIT

ROLLBACK
possible

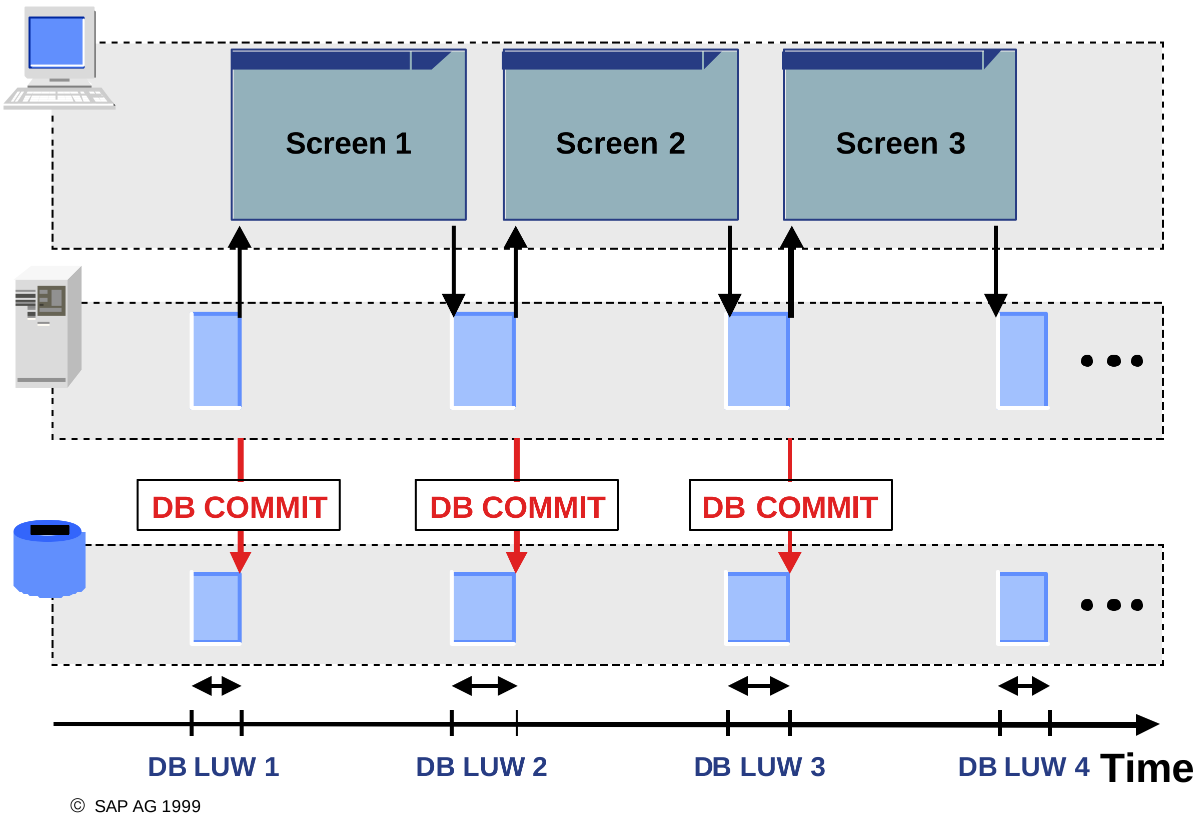
- A **database logical unit of work** (LUW) is a non-separable sequence of database operations. At the beginning and end of the LUW, the database is in a consistent state.
- The database LUW is either fully carried out by the database system, or is not carried out at all.
- A database LUW is opened with every dialog step and by a database commit of the previous DB LUW.
- The database LUW is closed with a database commit. It is only in the commit that the data is written to the database (after which it can no longer be reversed). Before the database commit, you can undo the changes using a database rollback. Here, the database is reset to the status that it had before the first change was made to the current DB LUW.
- Data that has been written to the database permanently with a database commit cannot be rolled back.
- Database LUWs allow you to encapsulate logically related actions from a business process. For example, when transferring sums of money in financial accounting, you must deduct an amount from one account and then add it to another account. Before and after the process, the data is consistent, but in between the two steps, it can be inconsistent.
- For further information, see the ABAP Editor keyword documentation for the term **transaction processing**.

R/3 Software View: Client/Server Architecture



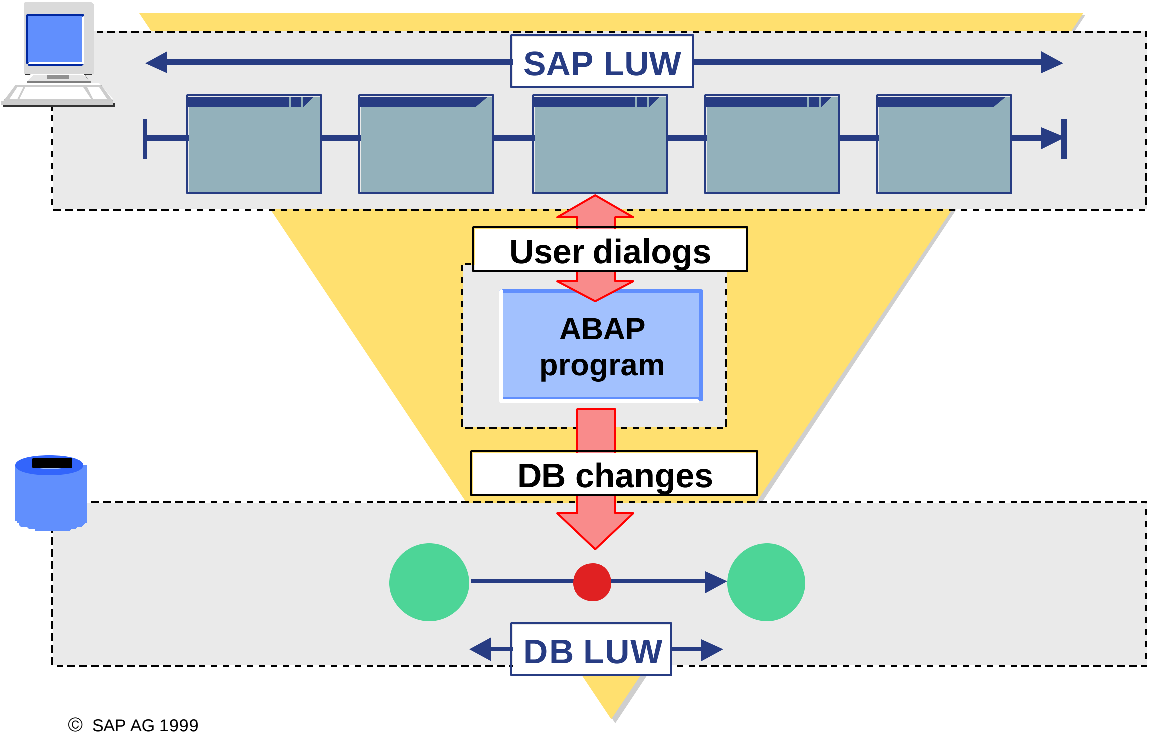
- The SAP R/3 System is based on the three-tier architecture of a client/server system. The three tiers are the database, application, and presentation server layers.
- This architecture, along with the distribution of users' requests (**user dispatching**), leads to a **highly-efficient**, **cost-effective multi-user** system.
- The three-tier architecture means that a la rge number of users with **low-cost** desktop computers (with low performance) can be mapped to a small number of **high-performance** (and considerably **more expensive**) work processes on application servers. Each work process on an application server is assigned a work process on a high-performance database server.
- Distributing user requests to work processes assigns individual clients at presentation server level to a work process for a particular period. In turn, the work process uses another work process in the database. After the work process has processed the user input in a dialog step, the user, along with the program context, is removed from the work process, which can then be used by another user.
- The three-tier architecture is far more scalable than a "fat" client architecture, in which the presentation and application levels run on one server. With a three-tier architecture, the number of database users is considerably lower than the number of users active in the system. This has a positive effect on the behavior of the database.

System Architecture: Implicit DB Commit



- The three-tier architecture of the R/3 System has certain consequences for process handling. When a work process is released for use by another user (client), an implicit database commit is triggered for the database process assigned to it (via a basis program).
- Work processes on the application server and database are released before each user dialog. This ensures that long user dialogs in which the system is "only displaying a screen" are not included in database LUWs. The duration of the user interaction will be longer than the DB LUW duration. Shorter database LUWs lead to less load on the database.
- Implicit commits on the database are triggered whenever the work process has to wait. This includes:
  - When the system sends a new screen
  - When the system sends a dialog message
  - When you make a synchronous remote function call (RFC)
  - When you use the `CALL TRANSACTION <t_code>` or `SUBMIT <program>` statement.

Target: Bundling The DB Changes Of An SAP LUW



- Using an SAP LUW to represent a business process chain usually involves user dialogs as well as the changes to the database. The aim of an R/3 transaction is to represent the information exchanged in the SAP LUW as an indivisible unit in the database. This means that an SAP LUW can only use a **single** database LUW.
- Since SAP LUWs usually involve several database LUWs, you need to bundle the database changes in a single database LUW within your transaction.

LUWs and Client/Server Architecture: Unit Summary

You are now able to:

- Explain the meaning of the terms database LUW and SAP LUW
- Explain why you need to bundle changes to database tables in the client/server architecture of the R/3 System

Exercises

Unit: LUW Concepts

At the conclusion of these exercises, you will be able to:

- Assess function modules and subroutines for LUW processing suitability.

The program **SAPBC414T_BOOKINGS_01** allows you to cancel bookings for a flight. For this purpose, bookings can be prepared for cancellation by selecting the appropriate checkbox on screen 200.

Implement the database dialog:
By clicking the *Save* icon (function code SAVE) on screen 200, the bookings selected in the **SBOOK** table are to be changed. In addition and within the same database LUW, the flight in question must be modified in the **SFLIGHT** table (the total number of bookings and number of reserved seats will change as a result of the cancellation). The changes made to the data records of both database tables must be made within one database LUW. Existing function modules are to be used for this purpose.

Program:	SAPMZ##_BOOKINGS1
Transaction code:	Z##_BOOKINGS1
Template:	SAPBC414**T**_BOOKINGS_01
Model solution:	SAPBC414**S**_BOOKINGS_01

1-1 Copy the program template **SAPBC414T_ BOOKINGS_01** with **all** sub-objects to **SAPMZ##_BOOKINGS1** (**##** is the group number) and assign transaction code **Z##_BOOKINGS1** to the program. Familiarize yourself with the program functionality.

1-2 The ABAP statements for the database updates are to be encapsulated in the **subroutine SAVE_MODIFIED_BOOKING**, which is called up from the PAI module USER_COMMAND_0200 (screen 200).

The database update is to be performed using the available function modules. A choice of two function modules is available for each table: **UPDATE_SBOOK**, **UPDATE_SBOOK_A**, **UPDATE_SFLIGHT** and **UPDATE_SFLIGHT_A**. Calling up the function modules with the right combination and sequence will ensure that the data remains consistent throughout all of the database tables in the case of an error.

1-2-1 Which function modules must be called up and in what order? For this purpose, check the source code in the function modules for ABAP statements, which terminate the database LUW prematurely and can, therefore, result in inconsistent data being written to the tables permanently.

1-2-2 Call up the function modules in the appropriate order from the subroutine SAVE_MODIFIED_BOOKING.

1-2-3 Deal with the exceptions of the function modules. Possible user messages:

Flight / bookings updated	⇒	Message 034
Error with flight / booking update	⇒	Message 044
Updates unsuccessful	⇒	Message 048
Flight sold out	⇒	Message 045
Flight does not exist	⇒	Message 046

The data records to be changed in the database table SBOOK are buffered in the **internal table ITAB_SBOOK_MODIFY**.

The key fields of the corresponding flight can be captured via the **WA_SFLIGHT structure.**

For information on the functionality of the template, see the attached graphic.

Unit: LUW Concepts

Model Solution SAPBC414S_BOOKINGS_01

Module Pool

```
*&-----------------------------------------------------------------*
*& Modulpool         SAPBC414S_BOOKINGS_01                         *
*&-----------------------------------------------------------------*
INCLUDE BC414S_BOOKINGS_01TOP.
INCLUDE BC414S_BOOKINGS_01O01.
INCLUDE BC414S_BOOKINGS_01I01.
INCLUDE BC414S_BOOKINGS_01F01.
INCLUDE BC414S_BOOKINGS_01F02.
INCLUDE BC414S_BOOKINGS_01F03.
INCLUDE BC414S_BOOKINGS_01F04.
INCLUDE BC414S_BOOKINGS_01F05.
INCLUDE BC414S_BOOKINGS_01F06.
```

SCREEN 100

```
PROCESS BEFORE OUTPUT.
  MODULE STATUS_0100.
*
PROCESS AFTER INPUT.
  MODULE EXIT AT EXIT-COMMAND.
  MODULE SAVE_OK_CODE.
  CHAIN.
* cancel booking: check if flight exists or flight can be created
    FIELD: SDYN_CONN-CARRID, SDYN_CONN-CONNID, SDYN_CONN-FLDATE.
    MODULE USER_COMMAND_0100.
  ENDCHAIN.
```

SCREEN 200

```
PROCESS BEFORE OUTPUT.
  MODULE STATUS_0200.
  MODULE TRANS_DETAILS.
  CALL SUBSCREEN SUB1 INCLUDING SY-CPROG '0201'.
  LOOP AT ITAB_BOOK INTO WA_BOOK WITH CONTROL TC_SBOOK.
    MODULE TRANS_TO_TC.
* allow only modification of bookings, that are not allready
cancelled
    MODULE MODIFY_SCREEN.
  ENDLOOP.
*
PROCESS AFTER INPUT.
  LOOP AT ITAB_BOOK.
* mark changed bookings in internal table itab_book
    FIELD SDYN_BOOK-CANCELLED MODULE MODIFY_ITAB ON REQUEST.
  ENDLOOP.
  MODULE EXIT AT EXIT-COMMAND.
  MODULE SAVE_OK_CODE.
  MODULE USER_COMMAND_0200.
```

SCREEN 201

```
PROCESS BEFORE OUTPUT.
PROCESS AFTER INPUT.
```

SCREEN 300

```
PROCESS BEFORE OUTPUT.
  MODULE STATUS_0300.
  MODULE TABSTRIP_INIT.
  MODULE TRANS_DETAILS.
  CALL SUBSCREEN TAB_SUB INCLUDING SY-CPROG SCREEN_NO.
*
PROCESS AFTER INPUT.
  CALL SUBSCREEN TAB_SUB.
  MODULE EXIT AT EXIT-COMMAND.
  MODULE SAVE_OK_CODE.
  MODULE TRANS_FROM_0300.
  MODULE USER_COMMAND_0300.
```

SCREEN 301

```
PROCESS BEFORE OUTPUT.
* MODULE HIDE_BOOKID.
PROCESS AFTER INPUT.
```

SCREEN 302

```
PROCESS BEFORE OUTPUT.
PROCESS AFTER INPUT.
```

SCREEN 303

```
PROCESS BEFORE OUTPUT.
PROCESS AFTER INPUT.
```

TOP Include

```
*&-----------------------------------------------------------------*
*& Include BC414S_BOOKINGS_01TOP                                   *
*&-----------------------------------------------------------------*
PROGRAM  sapbc414s_bookings_01 MESSAGE-ID bc414.

* line type of internal table itab_book, used to display bookings in
* table control
TYPES: BEGIN OF wa_book_type.
INCLUDE: STRUCTURE sbook.
TYPES:   name TYPE scustom-name,
         mark,
       END OF wa_book_type.

* work area and internal table used to display bookings in table
* control
DATA: wa_book TYPE wa_book_type,
      itab_book TYPE TABLE OF wa_book_type.

* bookings to be modified on database table
DATA: itab_sbook_modify TYPE TABLE OF sbook.

* change documents: bookings before changes are performed
DATA: itab_cd TYPE TABLE OF sbook WITH NON-UNIQUE KEY
      carrid connid fldate bookid customid.

* work areas for database tables spfli, sflight, sbook.
DATA: wa_sbook TYPE sbook, wa_sflight TYPE sflight, wa_spfli TYPE
      spfli.

* complex transactions: number of the customer created in the called
* transaction
data: scust_id(20).

* transport function codes from screens
DATA: ok_code TYPE sy-ucomm, save_ok LIKE ok_code.
* define subscreen screen number on tabstrip, screen 300
DATA: screen_no TYPE sy-dynnr.
* used to handle sy-subrc, which is determined in form
* transporting fields to/from screen
TABLES: sdyn_conn, sdyn_book.
* table control declaration (display bookings),
* tabstrip declaration (create booking)
CONTROLS: tc_sbook TYPE TABLEVIEW USING SCREEN '0200',
          tab TYPE TABSTRIP.
```

PBO Modules

```
*-------------------------------------------------------------------*
***INCLUDE BC414S_BOOKINGS_01O01 .
*-------------------------------------------------------------------*

*&------------------------------------------------------------------*
*&      Module  STATUS_0100  OUTPUT
*&------------------------------------------------------------------*
MODULE status_0100 OUTPUT.
  SET PF-STATUS 'DYN_100'.
  SET TITLEBAR 'DYN_100'.
ENDMODULE.                             " STATUS_0100  OUTPUT


*&------------------------------------------------------------------*
*&      Module  STATUS_0200  OUTPUT
*&------------------------------------------------------------------*
MODULE status_0200 OUTPUT.
  SET PF-STATUS 'DYN_200'.
  SET TITLEBAR 'DYN_200' WITH sdyn_conn-carrid sdyn_conn-connid
                              sdyn_conn-fldate.
ENDMODULE.                             " STATUS_0200  OUTPUT


*&------------------------------------------------------------------*
*&      Module  STATUS_0300  OUTPUT
*&------------------------------------------------------------------*
MODULE status_0300 OUTPUT.
  SET PF-STATUS 'DYN_300'.
  SET TITLEBAR 'DYN_300' WITH sdyn_conn-carrid sdyn_conn-connid
                              sdyn_conn-fldate.
ENDMODULE.                             " STATUS_0300  OUTPUT


*&------------------------------------------------------------------*
*&      Module  TRANS_DETAILS  OUTPUT
*&------------------------------------------------------------------*
MODULE trans_details OUTPUT.
  MOVE-CORRESPONDING: wa_spfli   TO sdyn_conn.
```

```
                      wa_sflight TO sdyn_conn,
                      wa_sbook   TO sdyn_book.
ENDMODULE.                             " TRANS_DETAILS  OUTPUT


*&------------------------------------------------------------------*
*&      Module  TRANS_TO_TC  OUTPUT
*&------------------------------------------------------------------*
MODULE trans_to_tc OUTPUT.
  MOVE-CORRESPONDING wa_book TO sdyn_book.
ENDMODULE.                             " TRANS_TO_TC  OUTPUT
```

```
*&-----------------------------------------------------------------*
*&      Module  MODIFY_SCREEN  OUTPUT
*&-----------------------------------------------------------------*
MODULE modify_screen OUTPUT.
  LOOP AT SCREEN.
    CHECK screen-name = 'SDYN_BOOK-CANCELLED'.
    CHECK ( NOT sdyn_book-cancelled IS INITIAL ) AND
          ( sdyn_book-mark IS INITIAL ).
    screen-input = 0.
    MODIFY SCREEN.
  ENDLOOP.
ENDMODULE.                             " MODIFY_SCREEN  OUTPUT


*&-----------------------------------------------------------------*
*&      Module  TABSTRIP_INIT  OUTPUT
*&-----------------------------------------------------------------*
MODULE tabstrip_init OUTPUT.
  CHECK tab-activetab IS INITIAL.
  tab-activetab = 'BOOK'.
  screen_no = '0301'.
ENDMODULE.                             " TABSTRIP_INIT  OUTPUT


*&-----------------------------------------------------------------*
*&      Module  HIDE_BOOKID  OUTPUT
*&-----------------------------------------------------------------*
MODULE hide_bookid OUTPUT.
* hide field displaying customer number when working with number
range
* object BS_SCUSTOM
  LOOP AT SCREEN.
    CHECK screen-name = 'SDYN_BOOK-BOOKID'.
    screen-active = 0.
    MODIFY SCREEN.
  ENDLOOP.
ENDMODULE.                             " HIDE_BOOKID  OUTPUT
```

PAI Modules

```
*-------------------------------------------------------------------*
***INCLUDE BC414S_BOOKINGS_01I01 .
*-------------------------------------------------------------------*

*&------------------------------------------------------------------*
*&      Module  EXIT  INPUT
*&------------------------------------------------------------------*
MODULE exit INPUT.
  CASE ok_code.
    WHEN 'CANCEL'.
      CASE sy-dynnr.
        WHEN '0100'.
          LEAVE PROGRAM.
        WHEN '0200'.
          LEAVE TO SCREEN '0100'.
        WHEN '0300'.
          LEAVE TO SCREEN '0100'.
        WHEN OTHERS.
      ENDCASE.
    WHEN 'EXIT'.
      LEAVE PROGRAM.
    WHEN OTHERS.
  ENDCASE.
ENDMODULE.                            " EXIT  INPUT


*&------------------------------------------------------------------*
*&      Module  SAVE_OK_CODE  INPUT
*&------------------------------------------------------------------*
MODULE save_ok_code INPUT.
  save_ok = ok_code.
  CLEAR ok_code.
ENDMODULE.                            " SAVE_OK_CODE  INPUT


*&------------------------------------------------------------------*
*&      Module  USER_COMMAND_0100  INPUT
*&------------------------------------------------------------------*
```

```
MODULE user_command_0100 INPUT.
  CASE save_ok.
****************************CANCEL BOOKING**************************
    WHEN 'BOOKC'.
      PERFORM read_sflight USING wa_sflight sysubrc.
* process returncode - if flight does not exist: e-message
      PERFORM process_sysubrc_bookc.
      PERFORM read_spfli USING wa_spfli.
      PERFORM read_sbook USING itab_book itab_cd.
      REFRESH CONTROL 'TC_SBOOK' FROM SCREEN '0200'.
****************************CREATE BOOKING**************************
    WHEN 'BOOKN'.
      PERFORM read_sflight USING wa_sflight sysubrc.
* process returncode - if flight does not exist: e-message
      PERFORM process_sysubrc_bookn.
      PERFORM read_spfli USING wa_spfli.
      PERFORM initialize_sbook USING wa_sbook.
    WHEN 'BACK'.
      SET SCREEN 0.
    WHEN OTHERS.
      SET SCREEN '0100'.
  ENDCASE.
ENDMODULE.                             " USER_COMMAND_0100  INPUT


*&-----------------------------------------------------------------*
*&      Module  USER_COMMAND_0200  INPUT
*&-----------------------------------------------------------------*
MODULE user_command_0200 INPUT.
  CASE save_ok.
    WHEN 'SAVE'.
* collect marked (changed) data sets in seperate internal table
      PERFORM collect_modified_data USING itab_sbook_modify.
* perform database changes
      PERFORM save_modified_booking.
      SET SCREEN '0100'.
    WHEN 'BACK'.
      SET SCREEN '0100'.
    WHEN OTHERS.
  ENDCASE.
ENDMODULE.                             " USER_COMMAND_0200  INPUT


*&-----------------------------------------------------------------*
*&      Module  MODIFY_ITAB  INPUT
*&-----------------------------------------------------------------*
MODULE modify_itab INPUT.
  wa_book-cancelled = sdyn_book-cancelled.
  wa_book-mark = 'X'.
  MODIFY itab_book FROM wa_book INDEX tc_sbook-current_line.
ENDMODULE.                             " MODIFY_ITAB  INPUT


*&-----------------------------------------------------------------*
*&      Module  USER_COMMAND_0300  INPUT
*&-----------------------------------------------------------------*
MODULE user_command_0300 INPUT.
  PERFORM tabstrip_set.
  CASE save_ok.
    WHEN 'NEW_CUSTOM'.
      PERFORM create_new_customer.
      SET SCREEN '0300'.
    WHEN 'SAVE'.
      PERFORM save_new_booking.
      SET SCREEN '0100'.
    WHEN 'BACK'.
      SET SCREEN '0100'.
    WHEN OTHERS.
      SET SCREEN '0300'.
  ENDCASE.
ENDMODULE.                             " USER_COMMAND_0300  INPUT


*&-----------------------------------------------------------------*
*&      Module  TRANS_FROM_0300  INPUT
*&-----------------------------------------------------------------*
MODULE trans_from_0300 INPUT.
  MOVE-CORRESPONDING sdyn_book TO wa_sbook.
```

FORM Routines

F01

```
*-------------------------------------------------------------------*
***INCLUDE BC414S_BOOKINGS_01F01 .
*-------------------------------------------------------------------*

*&------------------------------------------------------------------*
*&      Form  COLLECT_MODIFIED_DATA
*&------------------------------------------------------------------*
*      -->P_ITAB_SBOOK_MODIFY  text
*-------------------------------------------------------------------*
FORM collect_modified_data USING p_itab_sbook_modify
                                 LIKE itab_sbook_modify.
  DATA: wa_book LIKE LINE OF itab_book,
        wa_sbook_modify LIKE LINE OF p_itab_sbook_modify.
  CLEAR: p_itab_sbook_modify.
* Only bookings are collected, that
* 1) have been changed (mark = 'X')
* 2) shall be cancelled (cancelled = 'X')
  LOOP AT itab_book INTO wa_book
       WHERE    mark = 'X'
       AND cancelled = 'X'.
    MOVE-CORRESPONDING wa_book TO wa_sbook_modify.
    APPEND wa_sbook_modify TO p_itab_sbook_modify.
  ENDLOOP.
ENDFORM.                               " COLLECT_MODIFIED_DATA


*&------------------------------------------------------------------*
*&      Form  INITIALIZE_SBOOK
*&------------------------------------------------------------------*
*      -->P_WA_SBOOK  text
*-------------------------------------------------------------------*
FORM initialize_sbook USING p_wa_sbook TYPE sbook.
  CLEAR p_wa_sbook.
  MOVE-CORRESPONDING wa_flight TO p_wa_sbook.
```

```
ENDFORM.                               " INITIALIZE_SBOOK


*&---------------------------------------------------------------------*
*&      Form  PROCESS_SYSUBRC_BOOKC
*&---------------------------------------------------------------------*
FORM process_sysubrc_bookc.
  CASE sysubrc.
    WHEN 0.
      SET SCREEN '0200'.
    WHEN OTHERS.
      MESSAGE e023 WITH sdyn_conn-carrid sdyn_conn-connid
                        sdyn_conn-fldate.
  ENDCASE.
ENDFORM.                               " PROCESS_SYSUBRC_BOOKC


*&---------------------------------------------------------------------*
*&      Form  PROCESS_SYSUBRC_BOOKN
*&---------------------------------------------------------------------*
FORM process_sysubrc_bookn.
  CASE sysubrc.
    WHEN 0.
      SET SCREEN '0300'.
    WHEN OTHERS.
      MESSAGE e023 WITH sdyn_conn-carrid sdyn_conn-connid
                        sdyn_conn-fldate.
  ENDCASE.
ENDFORM.                               " PROCESS_SYSUBRC_BOOKN


*&---------------------------------------------------------------------*
*&      Form  TABSTRIP_SET
*&---------------------------------------------------------------------*
FORM tabstrip_set.
  IF save_ok = 'BOOK' OR save_ok = 'DETCON' OR save_ok = 'DETFLT'.
    tab-activetab = save_ok.
  ENDIF.
  CASE save_ok.
```

```
      screen_no = '0301'.
    WHEN 'DETCON'.
      screen_no = '0302'.
    WHEN 'DETFLT'.
      screen_no = '0303'.
  ENDCASE.
ENDFORM.                               " TABSTRIP_SET



*&-----------------------------------------------------------------*
*&      Form  CREATE_NEW_CUSTOMER
*&-----------------------------------------------------------------*
FORM create_new_customer.
********************** TO BE IMPLEMENTED LATER *********************
ENDFORM.                               " CREATE_NEW_CUSTOMER

*&-----------------------------------------------------------------*
*&      Form  NUMBER_GET_NEXT
*&-----------------------------------------------------------------*
FORM number_get_next USING p_wa_sbook LIKE sbook.
********************** TO BE IMPLEMENTED LATER *********************
ENDFORM.                               " NUMBER_GET_NEXT
```

F02

```
*----------------------------------------------------------------*
*   INCLUDE BC414S_BOOKINGS_01F02
*----------------------------------------------------------------*

*----------------------------------------------------------------*
*   FORM ENQ_SFLIGHT
*----------------------------------------------------------------*
FORM enq_sflight.
******************** TO BE IMPLEMENTED LATER *********************
ENDFORM.                             "ENQ_SFLIGHT


*----------------------------------------------------------------*
*   FORM ENQ_SBOOK
*----------------------------------------------------------------*
FORM enq_sbook.
******************** TO BE IMPLEMENTED LATER *********************
ENDFORM.                             "ENQ_SBOOK


*----------------------------------------------------------------*
*   FORM ENQ_SFLIGHT_SBOOK
*----------------------------------------------------------------*
FORM enq_sflight_sbook.
******************** TO BE IMPLEMENTED LATER *********************
ENDFORM.                             "ENQ_SFLIGHT_SBOOK


*----------------------------------------------------------------*
*   FORM DEQ_ALL
*----------------------------------------------------------------*
FORM deq_all.
******************** TO BE IMPLEMENTED LATER *********************
ENDFORM.                             "DEQ_ALL
```

F03

```
*   INCLUDE BC414S_BOOKINGS_01F03
*-------------------------------------------------------------------*


*&------------------------------------------------------------------*
*&      Form  READ_SFLIGHT
*&------------------------------------------------------------------*
*      -->P_WA_SFLIGHT  text
*      -->P_SYSUBRC     text
*-------------------------------------------------------------------*
FORM read_sflight USING p_wa_sflight TYPE sflight
                        p_sysubrc LIKE sy-subrc.
  SELECT SINGLE * FROM sflight INTO p_wa_sflight
         WHERE carrid = sdyn_conn-carrid
         AND   connid = sdyn_conn-connid
         AND   fldate = sdyn_conn-fldate.
  p_sysubrc = sy-subrc.
ENDFORM.                               " READ_SFLIGHT
*&------------------------------------------------------------------*
*&      Form  READ_SBOOK
*&------------------------------------------------------------------*
*      -->P_ITAB_BOOK  text
*      -->P_ITAB_CD    text
*-------------------------------------------------------------------*
FORM read_sbook USING p_itab_book LIKE itab_book
                      p_itab_cd   LIKE itab_cd.
  TYPES: BEGIN OF wa_custom_type,
           id TYPE scustom-id,
           name TYPE scustom-name,
         END OF wa_custom_type.
  DATA: wa_custom TYPE wa_custom_type,
        itab_custom TYPE STANDARD TABLE OF wa_custom_type
        WITH NON-UNIQUE KEY id,
        wa_book LIKE LINE OF p_itab_book,
        wa_cd   LIKE LINE OF p_itab_cd.
  CLEAR: p_itab_book, p_itab_cd.
* Select customer names in buffer table (array fetch)
  SELECT id name FROM scustom INTO CORRESPONDING FIELDS
         OF TABLE itab_custom.
* Select all bookings on selected flight (array fetch)
```

```
         WHERE carrid = sdyn_conn-carrid
         AND   connid = sdyn_conn-connid
         AND   fldate = sdyn_conn-fldate.
* read customer names corresponding to customer number from buffer
* table
  LOOP AT p_itab_book INTO wa_book.
    READ TABLE itab_custom INTO wa_custom WITH TABLE KEY
                                            id = wa_book-customid.
    wa_book-name = wa_custom-name.
    MODIFY p_itab_book FROM wa_book.
    MOVE-CORRESPONDING wa_book TO wa_cd.
    APPEND wa_cd TO p_itab_cd.
  ENDLOOP.
  SORT p_itab_book BY bookid customid.
ENDFORM.                               " READ_SBOOK


*&------------------------------------------------------------------*
*&      Form  READ_SPFLI
*&------------------------------------------------------------------*
*      -->P_WA_SPFLI  text
*-------------------------------------------------------------------*
FORM read_spfli USING p_wa_spfli TYPE spfli.
  SELECT SINGLE * FROM spfli INTO p_wa_spfli
         WHERE carrid = sdyn_conn-carrid
         AND   connid = sdyn_conn-connid.
  IF sy-subrc <> 0.
    MESSAGE e022 WITH sdyn_conn-carrid sdyn_conn-connid.
  ENDIF.
ENDFORM.                               " READ_SPFLI
```

F04

```
*-------------------------------------------------------------------*
*   INCLUDE BC414S_BOOKINGS_01F04
*-------------------------------------------------------------------*


*&------------------------------------------------------------------*
*&      Form  SAVE_MODIFIED_BOOKING
*&------------------------------------------------------------------*
FORM save_modified_booking.
* Modify data on database tables sbook and sflight
  CALL FUNCTION 'UPDATE_SBOOK'
       EXPORTING
            itab_sbook     = itab_sbook_modify
       EXCEPTIONS
            update_failure = 1
            OTHERS         = 2.
  CASE sy-subrc.
    WHEN 0.
      PERFORM update_sflight.
    WHEN OTHERS.
      MESSAGE a044 WITH wa_sflight-carrid wa_sflight-connid
                        wa_sflight-fldate.
  ENDCASE.
ENDFORM.                               " SAVE_MODIFIED_BOOKING

*&------------------------------------------------------------------*
*&      Form  UPDATE_SFLIGHT
*&------------------------------------------------------------------*
FORM update_sflight.
  CALL FUNCTION 'UPDATE_SFLIGHT'
       EXPORTING
            carrier          = wa_sflight-carrid
            connection       = wa_sflight-connid
            date             = wa_sflight-fldate
       EXCEPTIONS
            update_failure   = 1
            flight_full      = 2
            flight_not_found = 3
            OTHERS           = 4.
  CASE sy-subrc.
    WHEN 0.
      MESSAGE s034 WITH wa_sflight-carrid wa_sflight-connid
                        wa_sflight-fldate.
    WHEN 1.
      MESSAGE a044 WITH wa_sflight-carrid wa_sflight-connid
                        wa_sflight-fldate.
    WHEN 2.
      MESSAGE a045.
    WHEN 3.
      MESSAGE a046.
    WHEN OTHERS.
      MESSAGE a048.
  ENDCASE.
ENDFORM.                                       " UPDATE_SFLIGHT
```

```
*&---------------------------------------------------------------*
*&      Form  SAVE_NEW_BOOKING
*&---------------------------------------------------------------*
FORM save_new_booking.
********************* TO BE IMPLEMENTED LATER *********************
ENDFORM.                               " SAVE_NEW_BOOKING
```

F05

```
*----------------------------------------------------------------*
*   INCLUDE BC414S_BOOKINGS_01F05
*----------------------------------------------------------------*


*&---------------------------------------------------------------*
*&      Form  CONVERT_TO_LOC_CURRENCY
*&---------------------------------------------------------------*
*      -->P_WA_SBOOK  text
*----------------------------------------------------------------*
FORM convert_to_loc_currency USING p_wa_sbook TYPE sbook.
  SELECT SINGLE currcode FROM scarr INTO p_wa_sbook-loccurkey
         WHERE carrid = p_wa_sbook-carrid.
  CALL FUNCTION 'CONVERT_TO_LOCAL_CURRENCY_N'
       EXPORTING
            client           = sy-mandt
            date             = sy-datum
            foreign_amount   = p_wa_sbook-forcuram
            foreign_currency = p_wa_sbook-forcurkey
            local_currency   = p_wa_sbook-loccurkey
       IMPORTING
            local_amount     = p_wa_sbook-loccuram
       EXCEPTIONS
            no_rate_found    = 1
            overflow         = 2
            no_factors_found = 3
            no_spread_found  = 4
            derived_2_times  = 5
            OTHERS           = 6.
  IF sy-subrc <> 0.
    MESSAGE e080 WITH sy-subrc.
  ENDIF.
ENDFORM.                               " CONVERT_TO_LOC_CURRENCY
```

F06

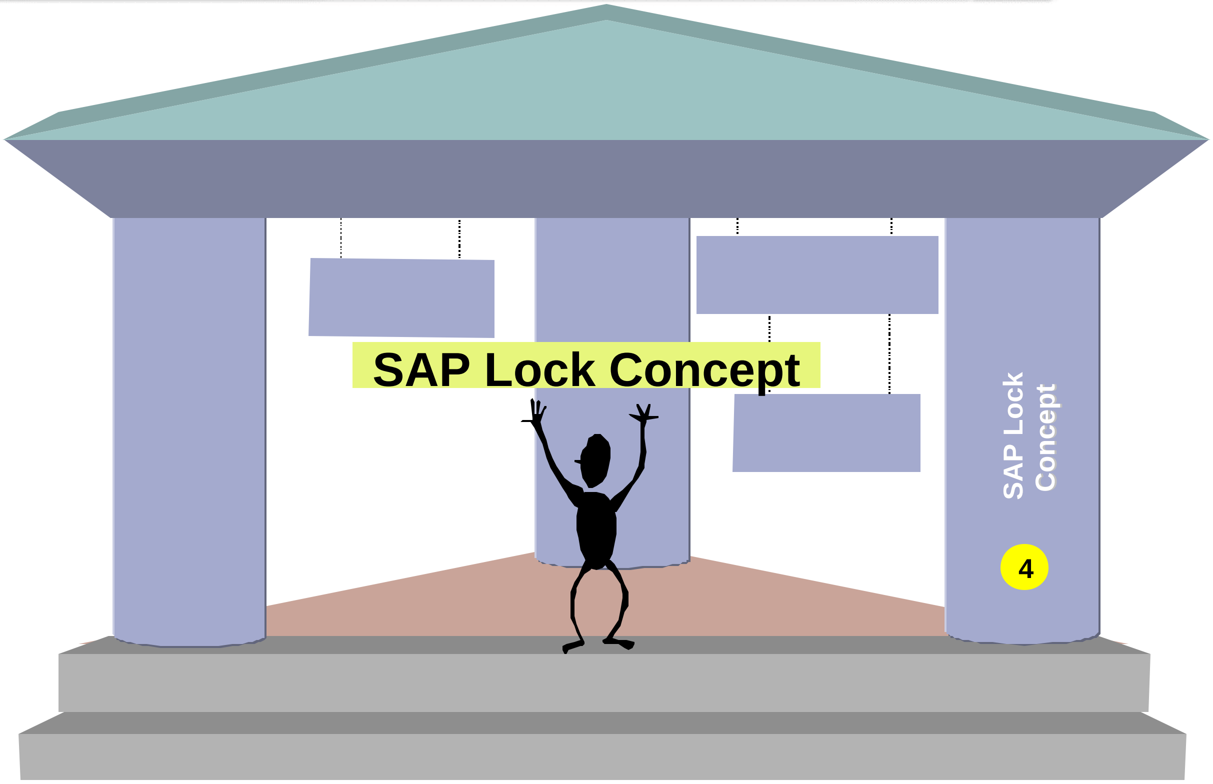
```
*-------------------------------------------------------------------*
*   INCLUDE BC414S_BOOKINGS_01F06
*-------------------------------------------------------------------*

*&------------------------------------------------------------------*
*&      Form  CREATE_CHANGE_DOCUMENTS
*&------------------------------------------------------------------*
FORM create_change_documents.
********************* TO BE IMPLEMENTED LATER **********************
ENDFORM.                               " CREATE_CHANGE_DOCUMENTS
```

SAP Lock Concept

Contents:

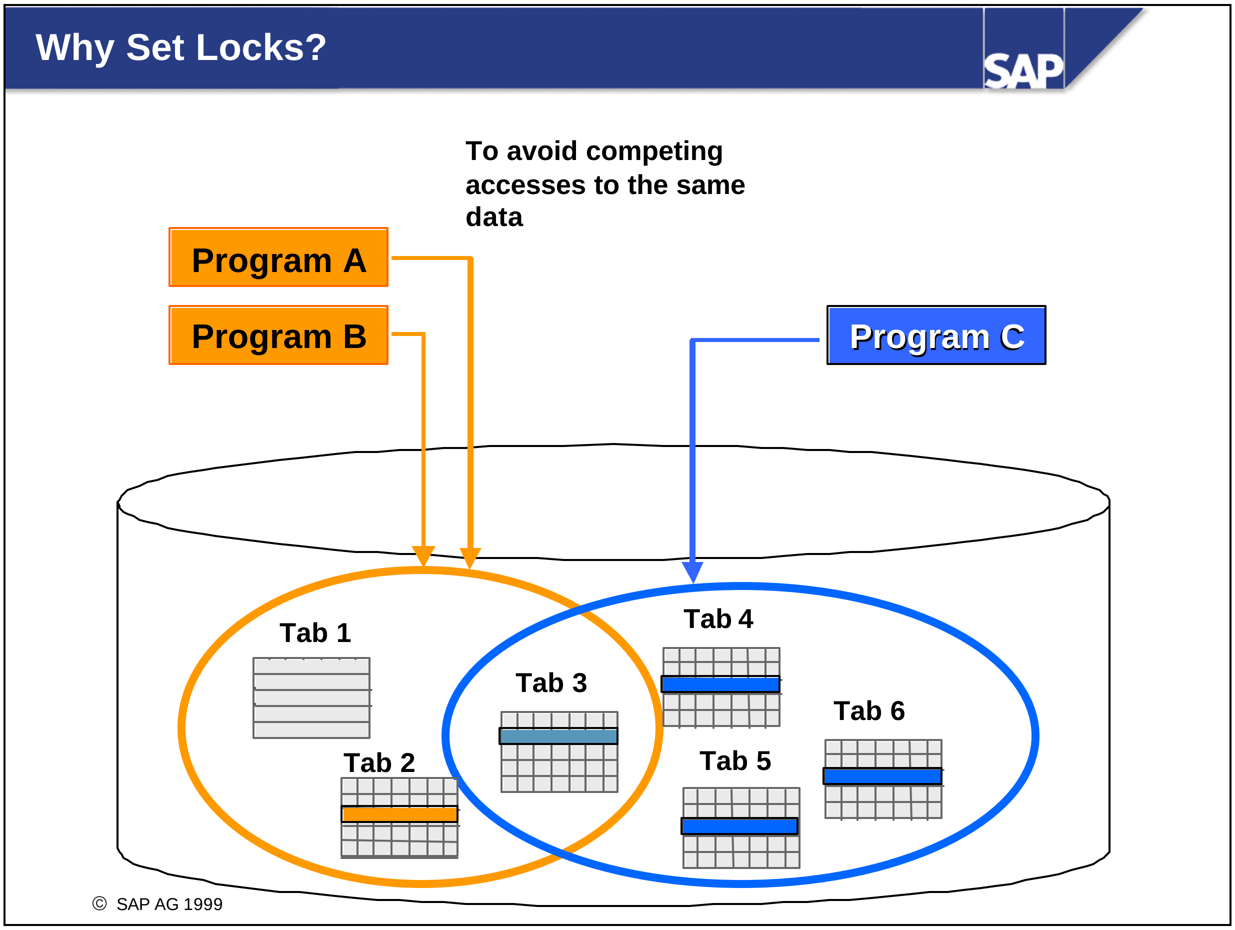
- Lock modules
- Lock objects
- Monitoring
- Using locks

Course Overview Diagram - SAP Lock Concept



Overview: Overview

- Overview
- Setting and releasing locks
- Lock objects
- Using locks: time sequence



Why Set Locks?

To avoid competing accesses to the same data

Program A

Program B

Program C

Tab 1

Tab 2

Tab 3

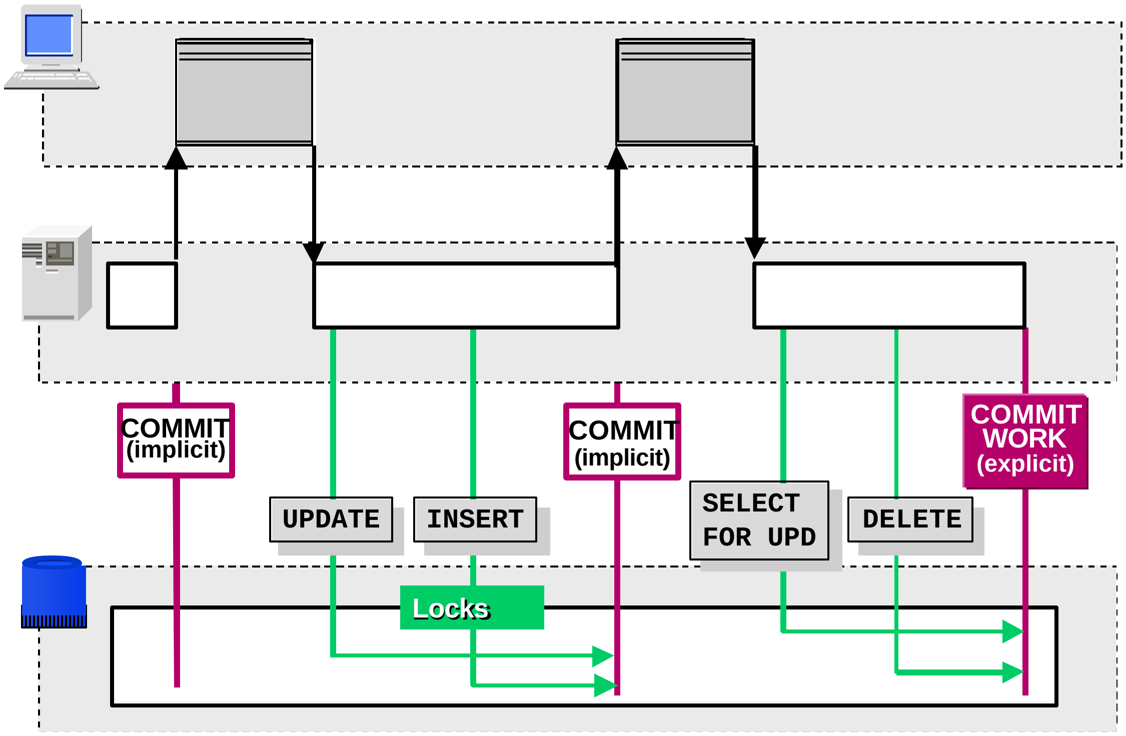
Tab 4

Tab 5

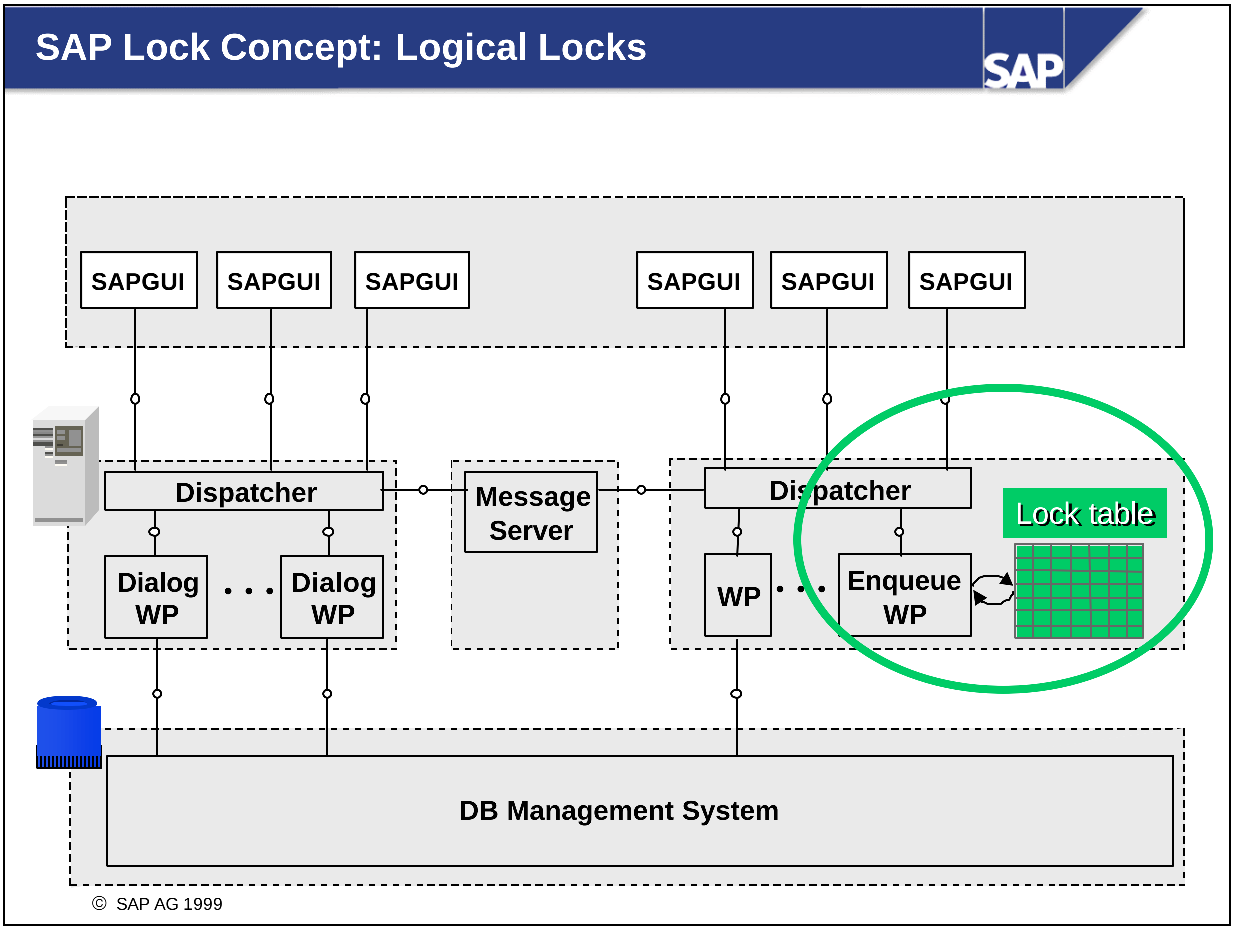
Tab 6

- If several users are competing to access the same resource or resources, you need to find a way of synchronizing the access in order to protect the consistency of your data.
- Example: In a flight booking system, you would need to check whether seats were still free before making a reservation. You also need a guarantee that critical data (the number of free seats in this case) cannot be changed while you are working with the program.
- Locks are a way of coordinating competing accesses to a resource. Each user requests a lock before accessing critical data.
- It is important to release the lock as soon as possible, so as not to hinder other users unnecessarily.

Database Locks Are Not Enough



- Whenever you make direct changes to data on the database in a transaction, the database system sets corresponding locks.
- The database management system (DBMS) physically locks the table entries that you want to change (`INSERT`; `UPDATE`, `MODIFY`), and those that you read from the database and intend to change (`SELECT SINGLE <f> FROM <dbtab> FOR UPDATE`). Other users who want to access the locked record or records must wait until the physical lock has been released. In such a case, the ABAP program waits until the lock has been released again.
- At the end of the database transaction, the database releases all of the locks that it has set during the transaction.
- In the R/3 System, this means that each database lock is released when a new screen is displayed, since a change of screen triggers an implicit database commit.



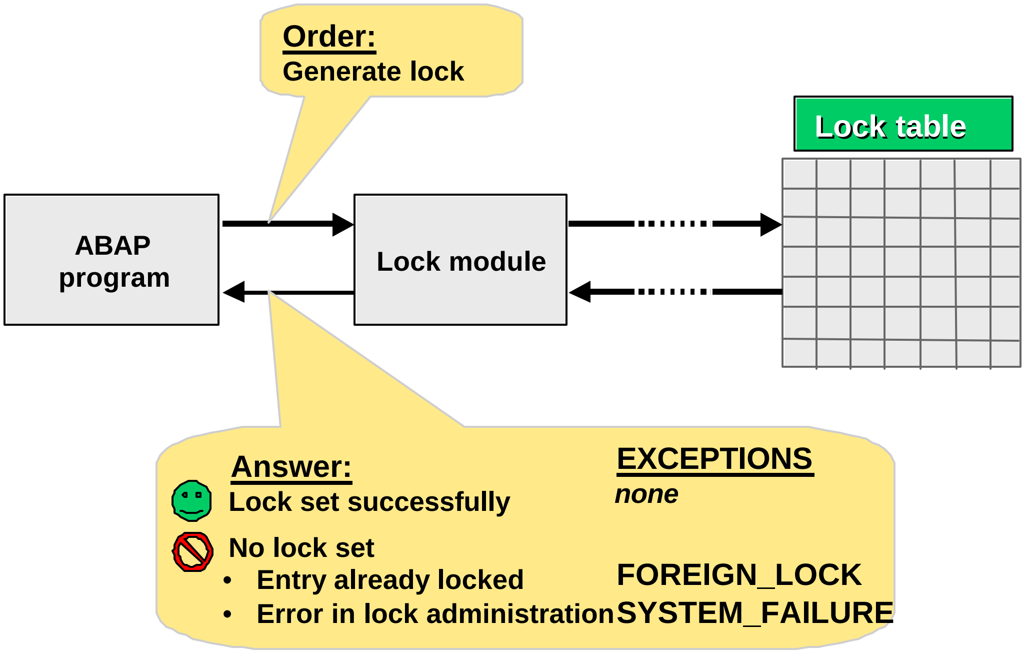
SAP Lock Concept: Logical Locks

- To keep a lock set through a series of screens (from the dialog program to the update program), the R/3 System has a **global lock table** at the application server level, which you can use to set **logical locks** for table entries.
- One application server contains this lock table and a special **enqueue work process**, which administers all requests for logical locks in the R/3 System. All logical lock requests of the R/3 System run using this work process.
- You can also use logical locks to "lock" table entries that do not yet exist on the database (inserting new lines). You cannot do this with physical database locks.
- For further information, see the ABAP Editor keyword documentation for the term **Locking**.

Overview: Setting and Releasing Locks

- Overview
- Setting and releasing locks
- Lock objects
- Using locks: time sequence

Setting and Deleting Logical Locks



Order:
Generate lock

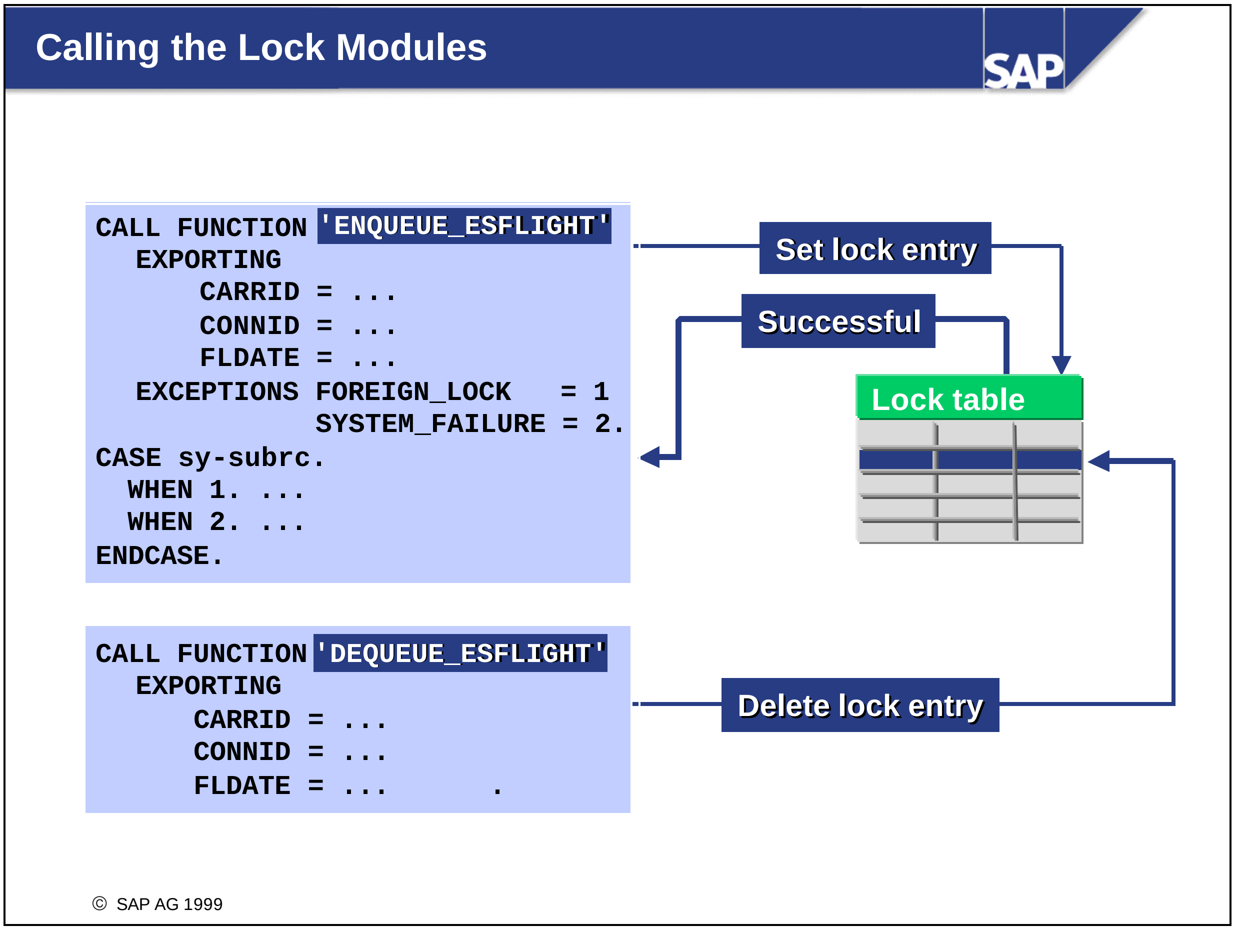
Lock table

ABAP program

Lock module

Answer:	EXCEPTIONS
🙂 Lock set successfully	*none*
🚫 No lock set	
• Entry already locked	FOREIGN_LOCK
• Error in lock administration	SYSTEM_FAILURE

- Logical locks are generated when an entry is written in the lock table. You use function modules to do this.
- You can only set a lock if the relevant table entry is not already locked.
- The SAP transaction receives information on the success of a lock request from a return code sent via the `EXCEPTION` interface of the function module. In other words, the control is returned to the program using the function module. The ABAP program does not need to wait.
- The SAP transaction can react appropriately by analyzing the return code.
- Another user cannot gain access to work with the same table entries that are already locked.
- Depending on the bundling technique in use for database updates), the program must delete the lock entries it generated using a lock module, or have them deleted indirectly (see unit *Organizing Database Updates*).
- If the user terminates the program that generated the lock entries (usually a dialog program), the locks are released automatically (implicitly). You can do this by entering `/n` in the command field, or with the statements `LEAVE PROGRAM`, `LEAVE TO TRANSACTION`, and 'A' or `'X'` messages.



Calling the Lock Modules

```
CALL FUNCTION 'ENQUEUE_ESFLIGHT'
   EXPORTING
        CARRID = ...
        CONNID = ...
        FLDATE = ...
   EXCEPTIONS FOREIGN_LOCK   = 1
              SYSTEM_FAILURE = 2.
CASE sy-subrc.
  WHEN 1. ...
  WHEN 2. ...
ENDCASE.
```

```
CALL FUNCTION 'DEQUEUE_ESFLIGHT'
   EXPORTING
        CARRID = ...
        CONNID = ...
        FLDATE = ...        .
```

Set lock entry

Successful

Lock table

Delete lock entry

- When you call an ENQUEUE function module, the dialog program tries to generate a lock entry.
- The export parameters identify the table entry (or entries) that you want to lock.
- The program that generates the locks (usually dialog program) analyzes the return code for lock requests and reacts accordingly.
- If the lock could not be set, you should normally output an error message.
- At the end of the dialog program, you can use the corresponding DEQUEUE function module to delete the entries from the lock table.
- DEQUEUE function modules have no exceptions. If you try to release an entry that is not locked, this has no effect.
- If you want to release all of the locks that you have set, at the end of your dialog program, you can use the function module DEQUEUE_ALL.

- The lock table contains the lock arguments for each table (for lock arguments, see the following slide).
- To display the lock table, use transaction SM12.

- The entries in the lock table are standard. Locks are always set using the values of the key fields in a table. These form the **lock argument**.
- You pass the values for the lock argument to the lock modules via their interface (function module `IMPORT` parameters).
- If you fail to set any of these parameters, the system interprets it generically, that is, the lock is set for all table lines that meet the criteria specified in the other parameters. The client parameter is an exception to this rule, where the default client `SY-MANDT` applies.

- Lock entries must be assigned to a **lock mode**.
- There are three different lock modes:
  - Mode 'E' for write locks: This is set if you want to write data to the database (change, create, or delete).

    Example: You want to book a seat for a flight. Once you have chosen the flight you want to book, you should ensure that no other customer books the same flight, to prevent the last free seat from being occupied more than once. (Technically speaking, you must lock the flight in the SFLIGHT table -> SEATSOCC field = number of occupied seats).
  - Mode 'S' for read locks: This is set if you want to ensure that the data, which you are reading from the database in your program, is not changed by other users while the program is running. You do not want to change the data itself in your program.

    Example: You are a travel agent and quote a customer the price for a flight that he or she is considering booking. While the customer is considering whether to buy the flight, you want to ensure that the price is not changed by another employee.
  - Mode 'X' for write locks: Like mode 'E', mode 'X' is used for writing data to the database. The technical difference between mode 'X' and mode 'E' is that locks of mode 'X' are not accumulated while a program is being executed. (For further details, see the following pages).

- If someone tries to lock the same data record again with a second program (different user), the various lock modes take effect as follows:
  - Write locks ('E' or 'X') mean that any lock attempts from other users are refused, irrespective of the mode in which the lock is attempted.
  - If a data record is locked in mode 'S' (shared), further locks in mode 'S' may be set by other users. Lock attempts in other lock modes ('E' or 'X') are refused.

- If you want to try to lock a data record more than once while a program is running (for example using a function module that you call up, which sets locks itself), the lock system reacts in the following way:
  - Mode 'E' write locks are not refused. Instead, a cumulative counter is incremented. The same applies to read locks (mode 'S').
  - If a data record is locked in mode 'E', a lock request generates a second lock, which is marked as a read lock.
  - If a data record is locked in mode 'S' and no further read locks are set by other users, a lock attempt in mode 'E' is possible. This generates a second entry in the lock table (for mode 'E').
  - If a data record is locked in mode 'X', all further lock requests are refused.

- If you want to ensure that you are reading up-to-date data in your program (with the intention of changing and returning this to the database), you should use the following procedure for lock requests and database accesses in your program:
    - First, lock the data that you want to edit.
    - Then read the current data from the database.
    - In the next step, process (change) the data in your program and write this to the database.
    - In the final step, release the locks that you set at the beginning.
- This procedure ensures that your changes run fully with lock protection and that you only read data that has been changed consistently by other programs (provided that these also use the SAP lock concept and follow the procedure described here).

- Lock modules are created for **lock objects** and not tables.
- Lock objects are maintained in the dictionary. Customer lock objects must begin with "EY" or "EZ".
- A lock object is a logical object composed of a list of tables that are linked by foreign key relationships. Lock modules are generated for these objects and enable common lock entries to be set for all tables contained in the lock object. This allows combinations of table entries to be locked.

  Example: A lock object that contains the tables SFLIGHT and SBOOK enables a flight with its bookings to be locked.
- The list of tables for a lock object consists of a **primary table**. Further table entries are referred to as **secondary tables**. Only tables with foreign key relationships to the primary table can be used as secondary tables.
- With lock objects, you can assign different names for the parameters that describe the fields of the lock arguments for the lock modules. The names of the table fields (key fields of the tables) are proposed by the system.
- You can specify the lock mode (a write lock 'E' or 'X' or a read lock 'S') for each table. These function as default values for the lock modules.

- After you have assigned tables and default lock modes, lock objects must be generated.
- When you activate a lock object, the system generates an `ENQUEUE` and a `DEQUEUE` function module.
- These have the names `ENQUEUE_<object_name>` and `DEQUEUE_<object_name>` respectively.

- If you want to ensure that you are reading current data in your program (with the intention of changing and returning this to the database), you should use the following procedure in your program for lock requests and database accesses:

    1. Lock the data that you want to edit.
    2. Read the current data from the database.
    3. Process (change) the data in your program and write this to the database.
    4. Release the locks that you set at the beginning.

- This procedure ensures that your changes run fully with lock protection and that you only read data, which has been changed consistently by other programs (with the restriction that these are also using the SAP lock concept and following the procedure described).

- If you change the order of the four steps to *Read* -> *Lock* -> *Change* -> *Unlock*, you run the risk that the data read by your program will not be up to date. Your program can read data before another user's program writes changes to the database. This means that a user of your program will make decisions for entries that are not based on up-to-date data from the database. For this reason, you should always follow the recommended procedure.

- Requesting a lock from a program is a communication step with lock administration. The communication step requires a certain time interval. If your program sets locks for several objects, this interval occurs more than once.
- By using so-called local **lock containers** , you can reduce these communication intervals with lock administration. To do so, collect the required lock requests of your program and send them together to lock administration.
- The locks (delayed execution) can be collected when the lock modules are called. For this purpose, qualify the IMPORT parameter_collect with 'X'. The data transferred via the lock module interface is then registered in a list (lock container) as a lock request that needs to be executed.
- The lock container can be terminated using the FLUSH_ENQUEUE function module and sent to lock administration.
- When the lock orders of a lock container can be executed, the lock container is deleted.
- If one of the locks in a container cannot be set, the function module FLUSH_ENQUEUE triggers the exception FOREIGN_LOCK. In this case, none of the registered lock requests is executed. The registered locks remain in the lock container.
- You can delete the contents of an existing lock container with the function module RESET_ENQUEUE.
- The specified function modules have release status *internally-released.*

Unit: SAP Lock Concept

Using the SAP Lock Concept

At the conclusion of these exercises, you will be able to:

- Call and use lock modules.
- Locate the places in programs where locks must be set and released in order to ensure that the data to be changed is protected adequately against competing accesses.

The program **SAPMZ##_BOOKINGS1** from the previous unit is to be changed to include locks that will prevent the booking data from being canceled and the flight data from being changed.

Program:	SAPMZ##_BOOKINGS2
Transaction code:	Z##_BOOKINGS2
Template:	SAPBC414**T**_BOOKINGS_02
Model solution:	SAPBC414**S**_BOOKINGS_02

1-1 Copy your solution **SAPMZ##_BOOKINGS1** or the program template **SAPBC414T_BOOKINGS_02** with **all** sub-objects to **SAPMZ##_BOOKINGS2** (## is your group number). Assign transaction code **Z##_BOOKINGS2** to the program.

1-2 Call the lock modules **ENQUEUE_ESFLIGHT**, **ENQUEUE_ESBOOK**, **ENQUEUE_ESFLIGHT_SBOOK** and **DEQUEUE_ALL** in subroutines. The subroutines in question are already created (blank) and combined in the **Include MZ##_BOOKINGS2F02.** To supply the interface parameters for the lock modules, use the fields in the structures SDYN_CONN and SDYN_BOOK.

1-3 Provide solutions for the exceptions of the lock modules. Possible user messages:

Data record is already being edited	⇒	Message 060
Processing terminated (booking already locked)	⇒	Message 061
Flight and/or bookings are already being edited	⇒	Message 062
Lock request not successful	⇒	Message 063

1-4 Protect the database changes related to the **booking cancellations** by calling up the corresponding lock modules (by calling up the corresponding subroutines). If a user action calls up screen 100, the locks must be canceled.

The lock module ENQUEUE_ESFLIGHT enables locks to be set for entries in table SFLIGHT. The lock module ENQUEUE_ESBOOK enables locks to be set for entries in table SBOOK. The lock module ENQUEUE_ESFLIGHT_SBOOK enables locks to be set in both tables at the same time (SFLIGHT, SBOOK) (reason: to lock a flight with booking(s)).

OPTIONAL

1-5 Extend your program for creating a new customer to include the necessary lock module calls. The calls ENQUEUE_ESCUSTOM (lock customer) and DEQUEUE_ALL (remove all locks) are already coded and encapsulated in the subroutines ENQ_SCUSTOM and DEQ_ALL (Include BC414T_CREATE_CUSTOMER_02F01).

1-5-1 Copy your solution **SAPMZ##_CUSTOMER1** or the program template **SAPBC414T_ CREATE_CUSTOMER_02** with **all** sub-objects to **SAPMZ##_CUSTOMER2** (## is your group number). Assign transaction code **Z##_CUSTOMER2** to the program.

1-5-2 Insert the call for the subroutines ENQ_SCUSTOM and DEQ_SCUSTOM at the appropriate places in your program. When should the customer data record be locked? Locate all the places at which the data record lock must be canceled. Familiarize yourself with the program flow, using the debugger if necessary.

Optional Exercise

Unit: SAP Lock Concept

Optional Exercise: Lock Objects

At the conclusion of these exercises, you will be able to:

- Search for and find lock objects.

2-1 Find out which function modules are maintained for logically locking flights, bookings, and flights with all dependent bookings in the system.

Unit: SAP Lock Concept

Model Solution SAPBC414S_BOOKINGS_02

PAI Modules

```
*-------------------------------------------------------------------*
***INCLUDE BC414S_BOOKINGS_02I01 .
*-------------------------------------------------------------------*


*&-----------------------------------------------------------------*
*&      Module  EXIT  INPUT
*&-----------------------------------------------------------------*
MODULE exit INPUT.
  CASE ok_code.
    WHEN 'CANCEL'.
      CASE sy-dynnr.
        WHEN '0100'.
          LEAVE PROGRAM.
        WHEN '0200'.
* remove all database locks
          PERFORM deq_all.
          LEAVE TO SCREEN '0100'.
        WHEN '0300'.
          LEAVE TO SCREEN '0100'.
        WHEN OTHERS.
      ENDCASE.
    WHEN 'EXIT'.
      LEAVE PROGRAM.
    WHEN OTHERS.
  ENDCASE.
ENDMODULE.                             " EXIT  INPUT
```

```
*&      Module  USER_COMMAND_0100  INPUT
*&-----------------------------------------------------------------*
MODULE user_command_0100 INPUT.
  CASE save_ok.
****************************CANCEL BOOKING**************************
    WHEN 'BOOKC'.
* set database lock for selected flight and depending bookings
      PERFORM enq_sflight_sbook.
      PERFORM read_sflight USING wa_sflight sysubrc.
      PERFORM process_sysubrc_bookc.
      PERFORM read_spfli USING wa_spfli.
      PERFORM read_sbook USING itab_book itab_cd.
      REFRESH CONTROL 'TC_SBOOK' FROM SCREEN '0200'.
****************************CREATE BOOKING**************************
    WHEN 'BOOKN'.
      PERFORM read_sflight USING wa_sflight sysubrc.
      PERFORM process_sysubrc_bookn.
      PERFORM read_spfli USING wa_spfli.
      PERFORM initialize_sbook USING wa_sbook.
    WHEN 'BACK'.
      SET SCREEN 0.
    WHEN OTHERS.
      SET SCREEN '0100'.
  ENDCASE.
ENDMODULE.                             " USER_COMMAND_0100  INPUT


*&-----------------------------------------------------------------*
*&      Module  USER_COMMAND_0200  INPUT
*&-----------------------------------------------------------------*
MODULE user_command_0200 INPUT.
  CASE save_ok.
    WHEN 'SAVE'.
      PERFORM collect_modified_data USING itab_sbook_modify.
      PERFORM save_modified_booking.
* remove all database locks
      PERFORM deq_all.
      SET SCREEN '0100'.
    WHEN 'BACK'.
```

```
      PERFORM deq_all.
      SET SCREEN '0100'.
    WHEN OTHERS.
      SET SCREEN '0200'.
  ENDCASE.
ENDMODULE.                                       " USER_COMMAND_0200  INPUT
```

FORM Routines

F01

```
*------------------------------------------------------------------*
***INCLUDE BC414S_BOOKINGS_02F01 .
*------------------------------------------------------------------*

*&-----------------------------------------------------------------*
*&      Form  PROCESS_SYSUBRC_BOOKC
*&-----------------------------------------------------------------*
FORM process_sysubrc_bookc.
  CASE sysubrc.
    WHEN 0.
      SET SCREEN '0200'.
    WHEN OTHERS.
* remove all database locks
      PERFORM deq_all.
      MESSAGE e023 WITH sdyn_conn-carrid sdyn_conn-connid
                        sdyn_conn-fldate.
  ENDCASE.
ENDFORM.                               " PROCESS_SYSUBRC_BOOKC
```

F02

```
*------------------------------------------------------------------*
*   INCLUDE BC414S_BOOKINGS_02F02
*------------------------------------------------------------------*

*------------------------------------------------------------------*
*   FORM ENQ_SFLIGHT
*------------------------------------------------------------------*
FORM enq_sflight.
  CALL FUNCTION 'ENQUEUE_ESFLIGHT'
       EXPORTING
            carrid         = sdyn_conn-carrid
            connid         = sdyn_conn-connid
            fldate         = sdyn_conn-fldate
       EXCEPTIONS
            foreign_lock   = 1
```

```
           system_failure = 2
           OTHERS         = 3.
  CASE sy-subrc.
    WHEN 0.
    WHEN 1.
      MESSAGE e060.
    WHEN OTHERS.
      MESSAGE e063 WITH sy-subrc.
  ENDCASE.
ENDFORM.                                       "ENQ_SFLIGHT
```

```
*------------------------------------------------------------------*
*   FORM ENQ_SBOOK
*------------------------------------------------------------------*
FORM enq_sbook.
  CALL FUNCTION 'ENQUEUE_ESBOOK'
       EXPORTING
            carrid         = sdyn_book-carrid
            connid         = sdyn_book-connid
            fldate         = sdyn_book-fldate
            bookid         = sdyn_book-bookid
            customid       = sdyn_book-customid
       EXCEPTIONS
            foreign_lock   = 1
            system_failure = 2
            OTHERS         = 3.
  CASE sy-subrc.
    WHEN 0.
    WHEN 1.
      MESSAGE e061.
    WHEN OTHERS.
      MESSAGE e063 WITH sy-subrc.
  ENDCASE.
ENDFORM.                               "ENQ_SBOOK


*------------------------------------------------------------------*
*   FORM ENQ_SFLIGHT_SBOOK
*------------------------------------------------------------------*
FORM enq_sflight_sbook.
  CALL FUNCTION 'ENQUEUE_ESFLIGHT_SBOOK'
       EXPORTING
            carrid         = sdyn_conn-carrid
            connid         = sdyn_conn-connid
            fldate         = sdyn_conn-fldate
       EXCEPTIONS
            foreign_lock   = 1
            system_failure = 2
            OTHERS         = 3.
  CASE sy-subrc.
```

```
    WHEN 0.
    WHEN 1.
      MESSAGE e062.
    WHEN OTHERS.
      MESSAGE e063 WITH sy-subrc.
  ENDCASE.
ENDFORM.                               "ENQ_SFLIGHT_SBOOK


*------------------------------------------------------------------
*   FORM DEQ_ALL
*------------------------------------------------------------------*
FORM deq_all.
  CALL FUNCTION 'DEQUEUE_ALL'.
ENDFORM.                               "DEQ_ALL
```

F03

```
*------------------------------------------------------------------*
*   INCLUDE BC414S_BOOKINGS_02F03
*------------------------------------------------------------------*

*&-----------------------------------------------------------------*
*&      Form  READ_SPFLI
*&-----------------------------------------------------------------*
*      -->P_WA_SPFLI  text
*--------------------------------------------------------------------
--*
FORM read_spfli USING p_wa_spfli TYPE spfli.
  SELECT SINGLE * FROM spfli INTO p_wa_spfli
         WHERE carrid = sdyn_conn-carrid
         AND   connid = sdyn_conn-connid.
  IF sy-subrc <> 0.
* remove all database locks
    PERFORM deq_all.
    MESSAGE e022 WITH sdyn_conn-carrid sdyn_conn-connid.
  ENDIF.
ENDFORM.                               " READ_SPFLI
```

OPTIONAL:

Model Solution SAPBC414S_CREATE_CUSTOMER_02

FORM Routines

F01

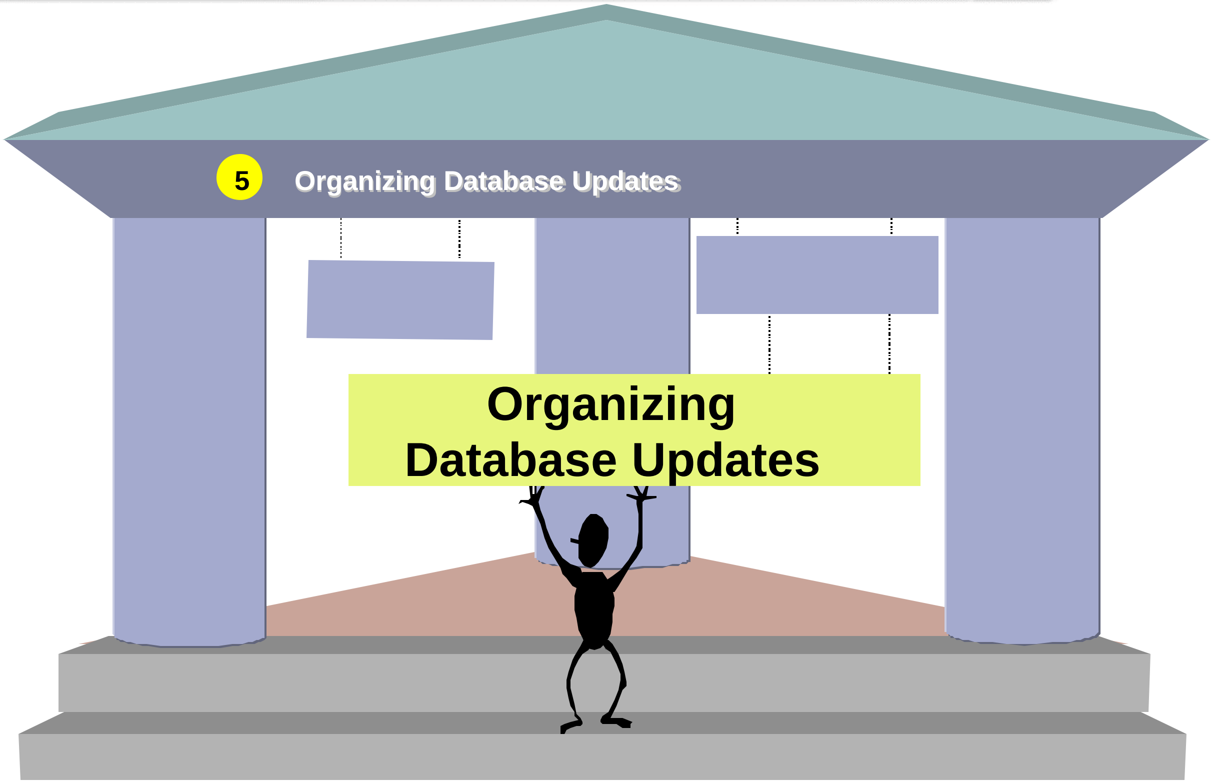
```
*---------------------------------------------------------------*
***INCLUDE BC414S_CREATE_CUSTOMER_02F01 .
*---------------------------------------------------------------*

*&--------------------------------------------------------------*
*&      Form  SAVE
*&--------------------------------------------------------------*
FORM save.
* lock dataset
  PERFORM enq_scustom.
  PERFORM number_get_next USING scustom.
  PERFORM save_scustom.
* unlock dataset
  PERFORM deq_all.
ENDFORM.                              " SAVE
```

Organizing Database Updates

Contents:

- Changes from the dialog
  - Direct
  - Using delayed subroutines
- Update techniques
  - Asynchronous, local, and synchronous updates
  - V1 and V2 updates
  - The concept of the SAP LUW

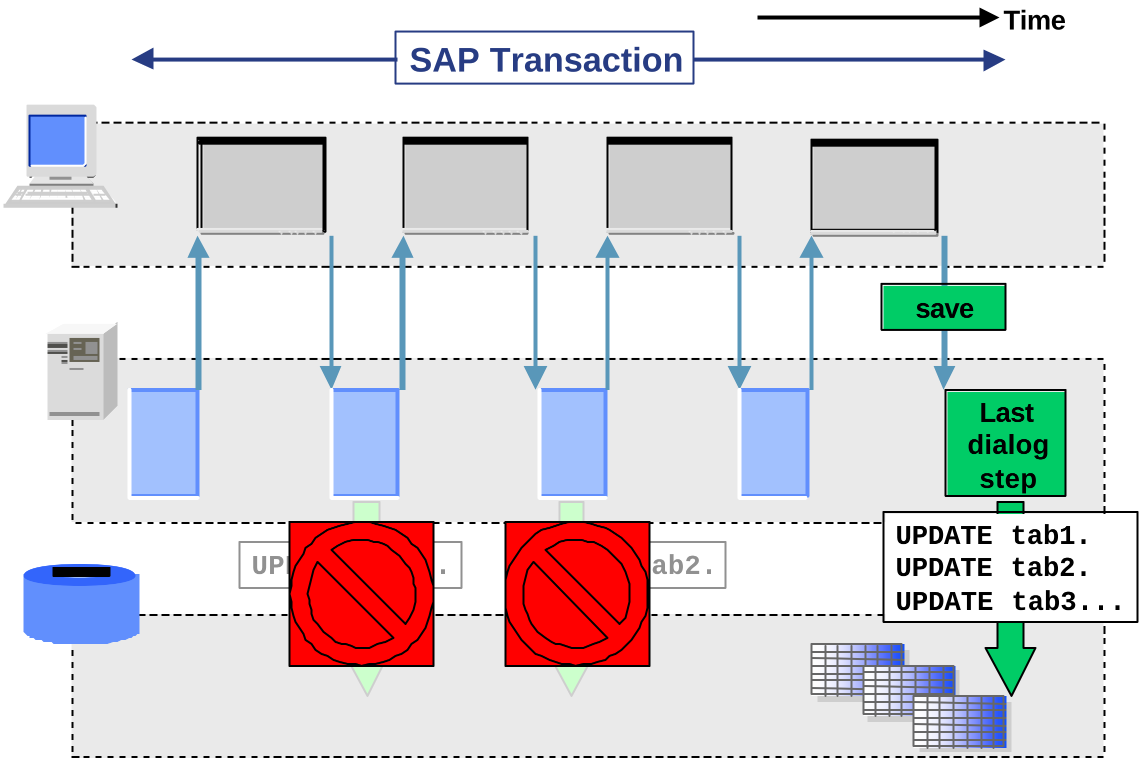
Course Overview Diagram - Database Updates



Overview Organizing Database Updates: Direct Changes From the Dialog

- Changes from the dialog
  - Direct
  - Using delayed subroutines
- Update techniques

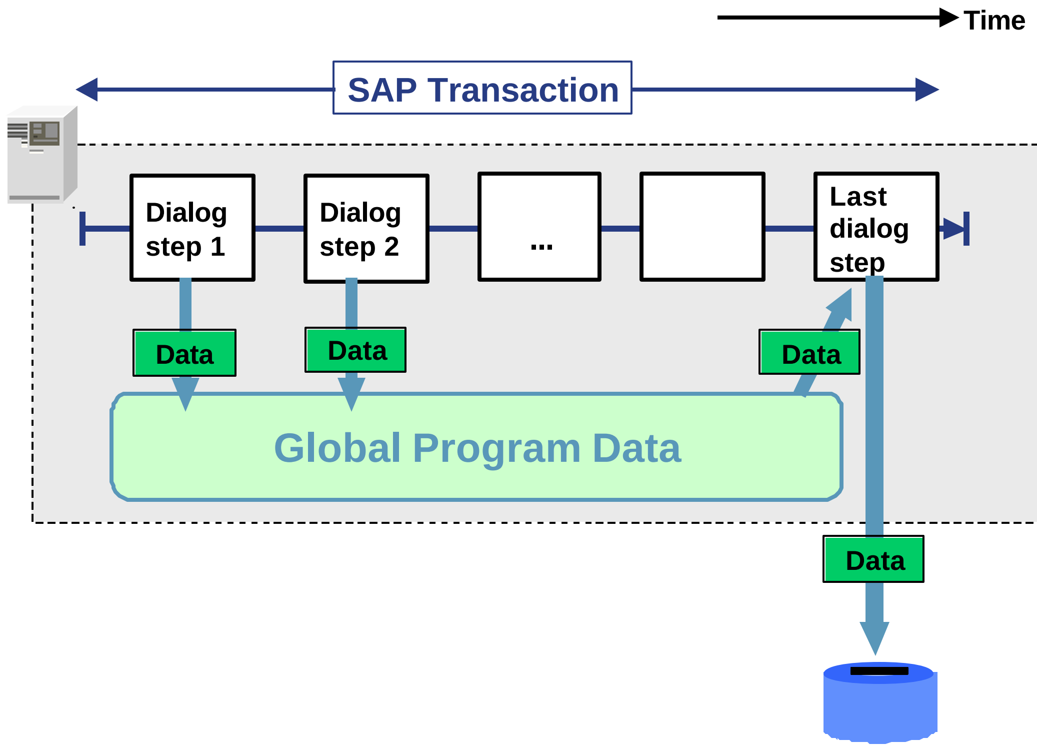
Direct Changes from the Dialog: Timescale



```
UPDATE tab1.
UPDATE tab2.
UPDATE tab3...
```

- If your transaction executes database updates from the dialog program, you must bundle all of your database updates into a single dialog step (usually the last). This is the only way to ensure that your database changes are processed on the all-or-nothing principle.

Direct Changes From the Dialog: Data Flow


Time

SAP Transaction

Dialog step 1

Dialog step 2

...

Last dialog step

Data

Data

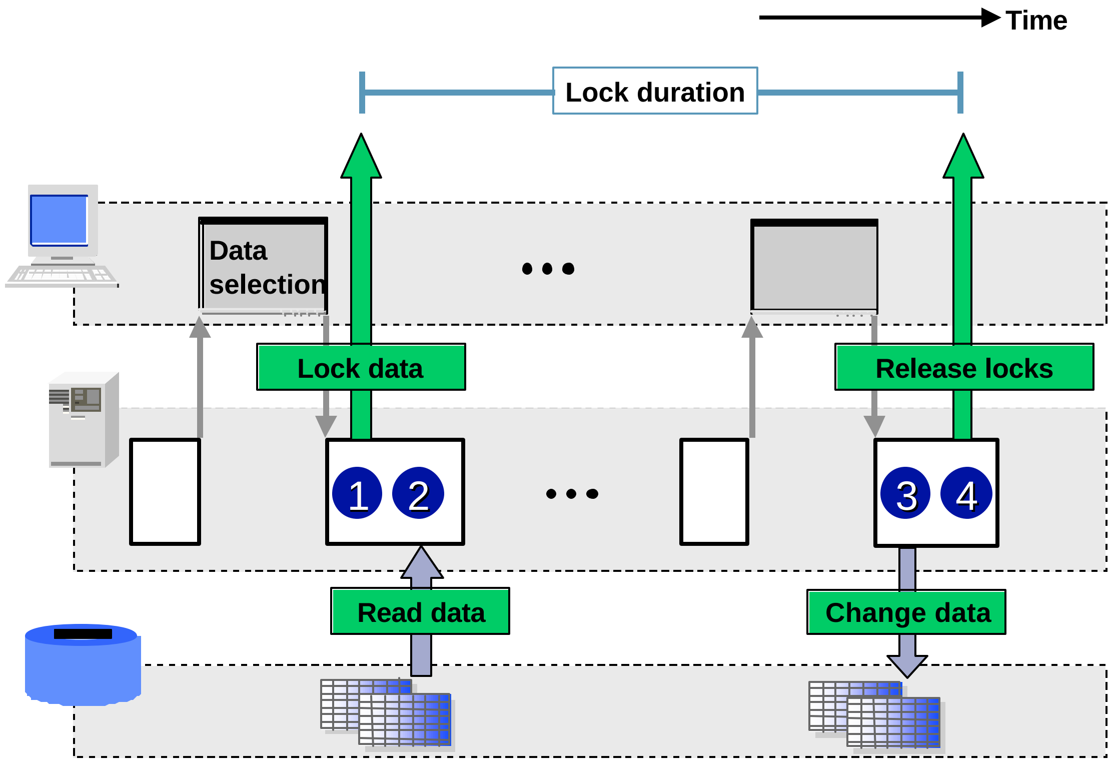
Data

Global Program Data

Data

- With database changes from the dialog program, you must save the data you want to change in the global program data until the database changes are made. This data is written to the database with the status it had for the last dialog step.

Direct Changes From the Dialog: Locks

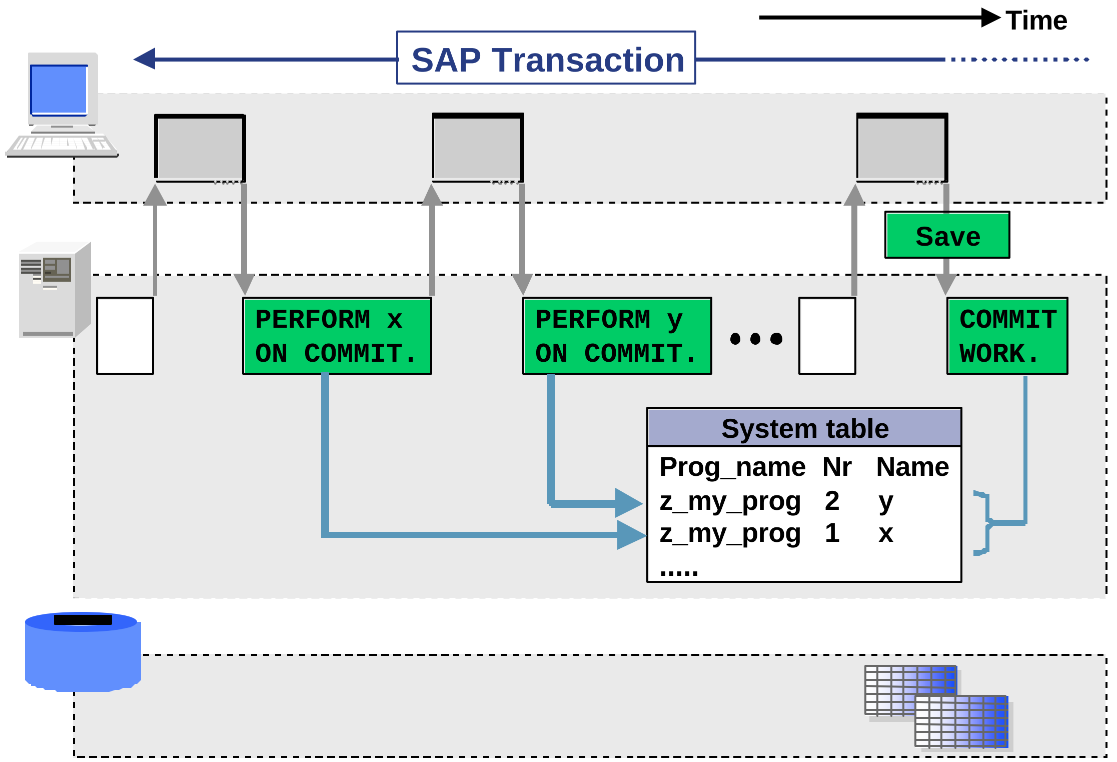


- With database changes from the dialog, your program must set and release SAP locks itself. The following order is recommended:
  - Lock data
  - Read data
  - Update data on the database
  - Release locks
- Note that the lock entries must be deleted by a program. To do so, you can either call up the lock modules of the object for releasing DEQUEUE_<object_name> or the function module DEQUEUE_ALL. For more detailed information, consult the function module documentation.

Overview of Changes From the Dialog: Using Delayed Subroutines

- Changes from the dialog
  - Direct
  - Using delayed subroutines
- Update techniques

PERFORM ON COMMIT: Timescale (1)

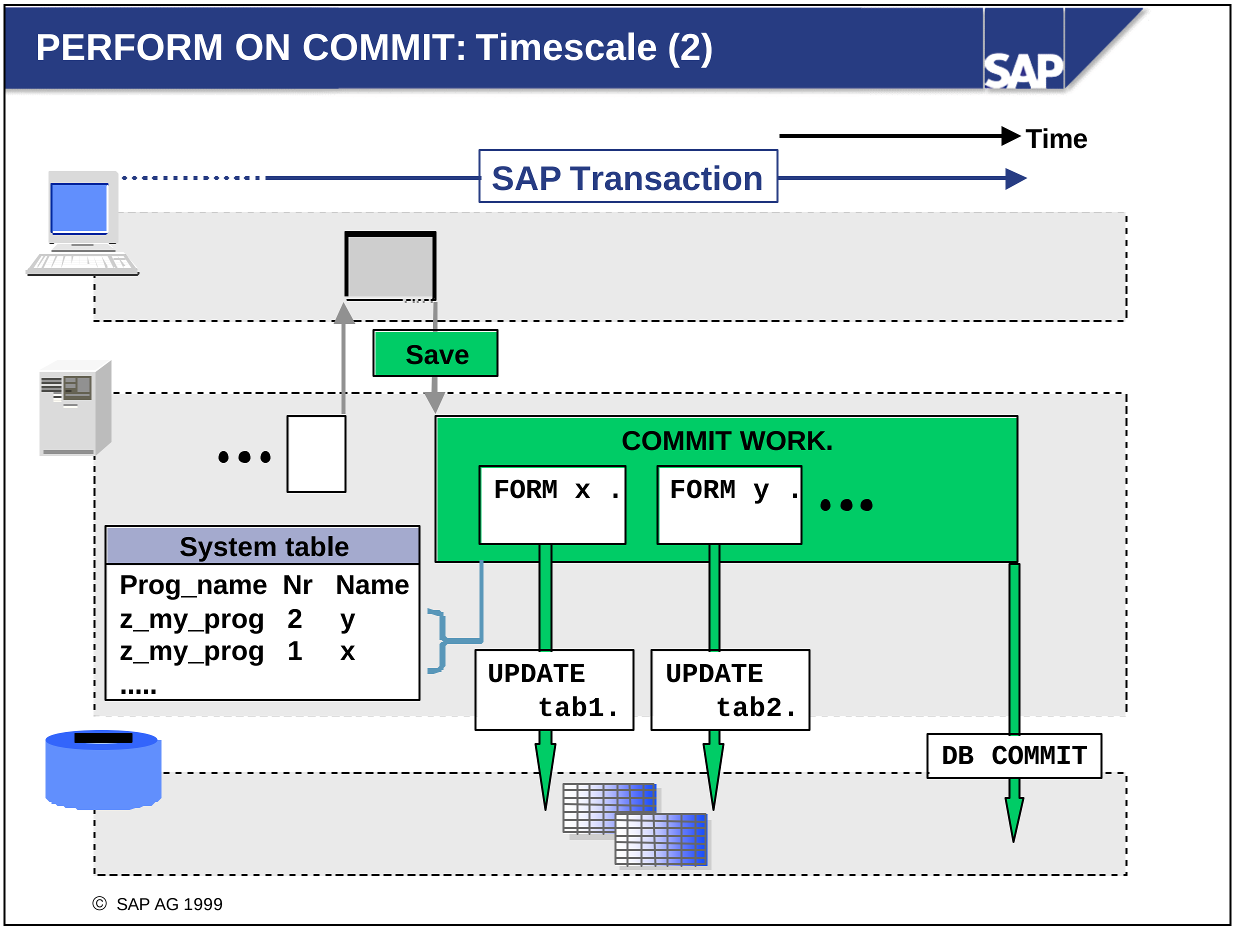


System table

Prog_name	Nr	Name
z_my_prog	2	y
z_my_prog	1	x
.....		

- Database updates from the dialog can be executed in bundled form by using the special subroutine technique `PERFORM <subroutine> ON COMMIT`.
- The statement `PERFORM <subroutine> ON COMMIT` registers the subroutine that has been called up. This will not be executed until the next `COMMIT WORK` statement is reached.
- If the database updates are encapsulated in the subroutines, they can be separated from the program logic and relocated to the end of the LUW processing.
- Each subroutine registered with `PERFORM ON COMMIT` is only executed once per LUW. Calls can be made more than once (no errors); the subroutine, however, is only executed once.
- From release 4.6 onward nested `PERFORM ON COMMIT` calls lead to a runtime error.

PERFORM ON COMMIT: Timescale (2)



System table

Prog_name	Nr	Name
z_my_prog	2	y
z_my_prog	1	x
.....		

- The `COMMIT WORK` statement carries out all subroutines registered to be executed and triggers a database commit (ends the DB LUW).

- Unlike normal subroutines, those that you call using the `ON COMMIT` addition do not have an interface. They work instead with **global data**, that is**,** the values of the data objects **at the point where the subroutine is actually run**. This can also include Imports from memory.
- The `PERFORM ... ON COMMIT` technique can also be used in the update. This will be discussed later.
- For further information, see the ABAP Editor keyword documentation for the term **`PERFORM`**.

- Update techniques allow you to separate user dialogs from the database changes. Both are executed by different programs, which generally run in different work processes.
- You work with a program that manages the user dialogs. It is referred to as a dialog program.
- You use a so-called update program that updates the data received by the dialog program on the database. No dialogs run in the update programs.

- Step 1: The dialog program receives the data changed by the user and writes it to a special log table. The entries in this table function as requests. The data contained in the log table will be written to the database later by the update program.
- A dialog program can write several entries to the log table.
- The entries in the log table represent an LUW, in other words they will either be executed on the database together or not at all (all-or-nothing principle).

- Step 2: The dialog program completes the logical data packet that was written to the log table. The SAP LUW finishes in the dialog part and informs the Basis system that a packet needs to be updated.

- Step 3: A basis program reads the data associated with the LUW from the log table and supplies it to the update program.

- Step 4: The update program accepts the data transferred to it and updates the database entries.

- Step 5: If the update program runs successfully, a Basis program deletes all entries for the LUW from the log table.
- In the event of an error, the entries remain in the log table and are flagged as errored.
- The option of informing users by mail that an update action has failed can be set using the profile parameters `rdisp/vb_mail_user_list` and `rdisp/vbmail`.
    - The parameter `rdisp/vbmail` can be set to `'0'` (no mail is sent in the event of an error) or '`1`' (a mail is sent in the event of an error).
    - The `rdisp/vb_mail_user_list` parameter setting specifies who will be informed in the event of an error (`rdisp/vbmail = 1`) (`$ACTUSER` informs the user who generated the data record to be updated).
- The monitor transaction for update orders is SM13.

- The dialog program and the update program can be linked in various ways:
  - Asynchronously
  - Synchronously
  - Via a local update

- Technical implementation of the update concept requires a so-called update program as well as the program that manages the user dialog. The update program tasks are carried out by special function modules called **update modules**.
- Create an update function module by choosing the processing radiobutton property '`update module`'.
- Update modules, like other function modules, have an interface for transferring data. The interface for update function modules only includes `IMPORTING` and `TABLES` parameters. These must be typed using reference fields or structures.
- Export parameters and exceptions are ignored in update modules.
- The function module contains the actual database update statements.

- The entries in the log file are generated from the dialog program. They are generated by calling up the associated update function module. The function module must be called using the addition `IN UPDATE TASK`. This ensures that the module is not executed immediately. Instead, the current data from the function module interface is written to the log table.
- For every `CALL FUNCTION ... IN UPDATE TASK` statement in the dialog program, the system generates an entry in the log table containing the name of the update function module and the associated parameters.

- All of the update requests in an SAP LUW are stored under the same update key (VB key). The update key is a unique key.
- When the system reaches the next `COMMIT WORK` statement, a log header is generated for the corresponding log entries, concluding the set of update entries for that SAP LUW. The log header contains information on the dialog program that wrote the log entries, as well as information on the update modules to be executed.
- As well as the header entry, the ABAP command `COMMIT WORK` ensures that the dispatcher process is informed about the availability of a further update packet.

- In a dialog consisting of several steps, you can store multiple entries in the update log table that are then processed following the ABAP `COMMIT WORK` command.
- However, you may also need to delete the update requests of the current SAP LUW using a `ROLLBACK WORK` statement.
- In a `ROLLBACK WORK` statement, the system:
  - Deletes all form routines registered using `PERFORM <subroutine> ON COMMIT`
  - Deletes all database update requests from the log
  - Triggers a rollback on the database, followed by a database commit
  - Starts a new SAP LUW
- With relation to database changes already completed in the dialog, the `ROLLBACK WORK` statement means that all changes in the current database LUW are undone.
- The `ROLLBACK WORK` statement deletes all lock entries generated up to now from the dialog program.

- The `ROLLBACK WORK` statement does not affect the program context, in other words all data objects (program-specific objects and objects from function groups that may be used) remain unchanged, they are **NOT** reset.
- You can generally only reset the data objects of your program by ending the dialog program. Therefore, you should **not use** the `ROLLBACK WORK` statement directly. Instead, trigger an implicit rollback by sending a termination message (type `A`). This ensures that all of the data from the program is also reset when the program terminates.

- The task of an update module is to pass the requests for database updates to the database and to evaluate their return codes.
- If the database cannot successfully complete an update, the update function module must be able to react.
- If you want to trigger a database rollback in the update task, you can use a termination message. This triggers an implicit database rollback.
- The rollback ends the update task. The log entry belonging to the SAP LUW is flagged as containing an error. The termination message is also entered in the log.
- The system sends an express mail to the relevant user, telling him or her that the update was unsuccessful. You can examine the log entry by using Transaction SM13.
- You **may not** use the explicit ABAP statements **`COMMIT WORK`** or **`ROLLBACK WORK`** in an **update module** .

- If your program is to run using locks, you must record the locks in the lock table. These are inherited by the update modules with the ABAP command `COMMIT WORK` and can then no longer be accessed by the dialog program.
- To ensure that the update modules run with the protection of locks, the lock entries must not be released before the `COMMIT WORK`.
- You do not need to release the locks explicitly in the update modules, since they are automatically released at the end of the update process by a basis program.
- The locks are also released if one of the update modules triggers a database rollback by sending a termination message.
- If the update modules allow failed update requests to be reprocessed (see V1 update), you should note that the data in the database tables at the point of reprocessing may be different from that at the point of the failed update attempt. Reprocessing failed update requests is only useful if the data to be updated is not dependent on the database status (e.g. writing of a document failed because of a tablespace overflow).
- Failed update requests are reprocessed without locks.

- In **asynchronous update**, the dialog program and update program run separately.
  - The dialog program writes the update requests into the log table VBLOG in the database.
  - You conclude the dialog part of the SAP LUW with the `COMMIT WORK` statement. A new SAP LUW immediately starts in the dialog program, which can carry on processing user dialogs without interruption. The dialog program does not wait for the update program to finish.
  - The update program is run on a special update work process. This need not be on the same server as the corresponding dialog work process.
  - The SAP LUW that began in the dialog program is continued and then closed by the update program.
- The log table VBLOG can be implement as a cluster file in your system, or be replaced with the transparent tables VBHDR, VBMOD, VBDATA, and VBERROR.
- Asynchronous updates are useful in transactions where the database updates take a long time and the "perceived performance" by the shorter user dialog response time is important.
- Asynchronous update is the standard technique for dialog processing.
- The entries that have a HEADER can be analyzed in SM13.

- In **local update**, the update functions are run on the same dialog process used by the dialog program containing the `COMMIT WORK` statement.
- To do this, you must include the `SET UPDATE TASK LOCAL` statement in the dialog program. The effect of this is that update requests are kept in main memory rather than being written into table VBLOG in the database.
- When the system reaches the `COMMIT WORK` statement, the corresponding update modules are processed in the dialog work process currently being used by the dialog program. If all of the update modules run successfully, a database commit is triggered. If not, a database rollback occurs.
- Once the update function modules have been processed, the dialog program resumes with a new SAP LUW.
- The `SET UPDATE TASK LOCAL` flag can only be set if no other update requests were generated for the same LUW before the program was called up.
- The `SET UPDATE TASK LOCAL` flag is effective until the next `COMMIT WORK` or `ROLLBACK WORK` command.

- With **synchronous updates,** the dialog program waits for the end of the update modules. The dialog program does not begin to process the new SAP LUW until the update modules have terminated.
- To switch from asynchronous to synchronous update, use the `AND WAIT` addition in the `COMMIT WORK` statement.
- The entries that have a HEADER can be analyzed in SM13.

- Asynchronous update is useful in transactions where subsequent user dialogs do not depend on the database updates being made immediately. Once the update task has been called, control returns directly to the user.
- Local update is particularly useful for processing dialog transactions in the background. There is no contact with the database table VBLOG, and if the program is running alone on the server, local update is faster than either synchronous or asynchronous update. If, as is the usual case, several users are using the server, the speed of the program depends on the total server load.
- Synchronous update is useful in transactions where you want to use the advantages of update techniques (logging, opportunity to reprocess failed update requests), but where subsequent user dialogs nevertheless do depend on the results of the update. One particular application for this technique is in "transactions within transactions" - where one transaction uses other transactions as modularization units (CALL TRANSACTION <t_code>). When you use this method, you can determine in the call the update technique that you want the transaction to use. For further information, see the keyword documentation in the ABAP Editor for the term **CALL TRANSACTION**.

- Update function modules can be separated into **two groups**. The group determines when the function module is processed: Function modules that are classified as **V1** can be further divided into two subclasses: *Start immediately* or *Start immediately, no restart*. **V2** function modules are processed asynchronously after all **V1** update modules have finished running.
- If you have used the *Start immediately* (**V1**) option, you can update any records that contained errors manually, using Transaction SM13. If you use the *Start immediately, no restart* (**V1**) option, this is not possible. **V2** update function modules (*Start delayed*) can always be manually updated.
- **V1** update function modules do not normally run using the SAP lock concept. In other words, the **V1** update program is executed with the protection of the locks from the dialog program.
- Any lock entries are released at the end of the V1 update. **V2** update function modules always run without logical locks.
- You can also classify an update module using attribute 'Coll. run' (collective run). This option is used SAP internal only (special form of V2 update, asynchronously, start via program `RSM13005`).

- The flow diagrams discussed up to now all deal with V1 updates.
- Update requests for V2 update modules are also generated by the dialog program.
- V1 update modules generate update requests in table VBLOG in synchronous and asynchronous update, and in main memory in local update.
- V2 update modules generate entries in VBLOG and always run asynchronously.

- V1 update modules are handled by the system with priority and are executed before the V2 update requests.
- V1 updates can be performed synchronously, asynchronously, or locally.

- V2 update function modules are not **processed until all V1 update function modules have been successfully processed**.
- The V2 update function modules run in a separate DB LUW. They are executed in a V2 update work process. If there are no V2 update work processes set up in your system, the V2 update function modules run in a V1 update work process.
- Once all of the V2 update function modules have been executed successfully, the V2 update requests are deleted from VBLOG.
- If an error occurs in a V2 update function module to which the function module reacts with a termination error message, the system triggers a database rollback. All of the V2 changes in the SAP LUW are undone and an error flag is set in table VBLOG for all of the V2 update requests.
- V2 update function modules run without SAP locks.
- The division between V1 and V2 update function modules allows you to set 'high priority' and 'low priority' updates.
- V2 update function modules are used for low-priority tasks, such as writing statistics to the database.

- The locks generated in the dialog program are usually inherited by the V1 update modules when the update takes place. This is controlled by the `SCOPE` interface parameter of the lock modules. When `SCOPE = 2`, the V1 update programs inherit the locks that are set in the dialog program.
- 2 is the default setting for `SCOPE` when you call a lock module.
- You do not need to release the locks explicitly in your program, since they are automatically released at the end of the V1 update process.
- The locks are also released if one of the V1 update modules triggers a database rollback by sending a termination message.

- An SAP LUW maps updates, which are logically related and usually involve several dialog steps, to a database LUW. The database updates are encapsulated via update modules.
- SAP LUWs are supported specifically by R/3:
  - Locks (Scope = 2)
  - The CALL FUNCTION IN UPDATE TASK call mechanism
  - The command COMMIT WORK
  - Type 'A' or 'X' dialog messages.

- An SAP LUW can be divided into three phases (three-phase model).
- Dialogs, user entries, and their input checks take place in phase 1. Calls of update modules are not allowed here, since they might be registered more than once during an error dialog (E message). Phase 1 ends when the first update module is called. The data to be updated must be held in global program data during phase 1.
- Preparations for database updates take place in phase 2. Phase 2 begins when the first update module is called and ends with the `COMMIT WORK` statement. The system must now respond to any errors with a type '`A`' or '`X`' dialog message. The `COMMIT WORK` that concludes phase 2 should only be set at the top level if call hierarchies are used, since the lower-level modularization units in the hierarchy are not aware of the status of the program context.
- The database updates are performed in phase 3. The system must always respond to any errors in phase 3 with a type '`A`' or '`X`' dialog message. This leads to a ROLLBACK of the complete database LUW as well as a termination of the update.

- Local update processing isactivated using the ABAP command `SET UPDATE TASK LOCAL`. The update type can only be changed if it is processed before the first update module is called.
- With local updates, the update modules are executed in the dialog work process that is currently performing the SAP LUW.
- As is the case with synchronous updates, the user must wait while the update modules are being executed.
- Local updates should be used for:
  - Transactions that are carried out in the background (batch) (`CALL TRANSACTION USING`) Exception: If unbuffered number assignments and higher parallel processing is requested at the same time.
  - Dialog transactions with very few database changes (3 - 5 statements) for which the dialog behavior is not critical.
- Note that the fewer the number of users making changes simultaneously, the better the response time of the database.

- Synchronous updates are triggered by the ABAP statement COMMIT WORK AND WAIT. With a synchronous update, the update modules are executed in an update work process.
- Unlike asynchronous updates, the dialog part of the transaction is stopped while the update modules are being executed.
- The success or failure of the update is displayed in system field `sy-subrc` once the update has been completed.

- For every action on the database that prompts table updates, the record to be changed is locked physically by the database. The same applies if you are reading with `SELECT ... FOR UPDATE`.
- Other users cannot change the same data for the duration of the lock.
- To reduce the lock duration on the database, you should use the following rule to program the database updates carried out by the update modules:
  - First, new table entries should be created. These present the smallest 'problem' for the other users.
  - You should then perform table updates that are not critical to performance. As a rule, these are the tables that are accessed 'simultaneously' by as few users as possible.
  - Tables that are central resources in the system (which many users access at once) should always be changed as late as possible.

- To lock the central tables (performance critical) for as short a time as possible, you can use `PERFORM uprog ON COMMIT` in the update.
- For this purpose, encapsulate the changes to the central tables in `FORM` routines and call these up in the update using `PERFORM ON COMMIT`. The `FORM` routines are then not executed until the last update module has been processed.
- After the last update module has been processed, a program executes the ABAP command `COMMIT WORK`, which then performs the `FORM` routines registered in the update.

Unit: Organizing Database Updates

At the conclusion of these exercises, you will be able to:

- Perform database updates using the asynchronous update technique

The program **SAPMZ##_BOOKINGS2** from the previous unit is to be changed or enhanced so that database updates can be performed using the asynchronous update technique.

Canceling bookings:
To implement the asynchronous update technique, the existing source code needs to be adjusted here.

Creating a new booking:
The database dialog part is to be implemented here. The data for a new booking is entered on screen 300. Clicking the *Save* icon (function code SAVE) on screen 300 is to insert the new bookings in the **SBOOK** table and modify the flight in question in the **SFLIGHT** table. The updates are to be performed within a DB LUW and using the asynchronous update technique.

Program:	SAPMZ##_BOOKINGS3
Transaction code:	Z##_BOOKINGS3
Template:	SAPBC414**T**_BOOKINGS_03
Model solution:	SAPBC414**S**_BOOKINGS_03

1-1 Copy your solution **SAPMZ##_BOOKINGS2** or the program template **SAPBC414T_BOOKINGS_03** with **all** sub-objects to **SAPMZ##_BOOKINGS3** (## is your group number). Assign transaction code **Z##_BOOKINGS3** to the program.

1-2 Canceling existing bookings:

1-2-1 **Function modules UPDATE_SFLIGHT** and **UPDATE_SBOOK** are used to update the table entries in the DB tables SLFIGHT and SBOOK. Can these function modules also be used to perform the updates using the update technique?

1-2-2 Modify your program so that the updates to the DB tables **SFLIGHT** and **SBOOK** are performed using the update technique:

- Call up the corresponding function modules capable of performing updates in the **SAVE_MODIFIED_BOOKING subroutine**
- Insert the statement **COMMIT WORK** in the PAI module **USER_COMMAND_0200**
- Note that the locks (SCOPE = 2) are inherited by the update program and, therefore, are not released explicitly in the dialog program.

1-3 Generating a new booking:
To generate a new entry in the DB table SBOOK, use the function module **INSERT_SBOOK,** which is capable of performing updates. This function module is to be called up in the **subroutine SAVE_NEW_BOOKING**. The subroutine is called up from the PAI module USER_COMMAND_0300 (screen 300) and is already created (blank).

1-3-1 Call up the function modules **INSERT_SBOOK** and **UPDATE_SFLIGHT**, which are capable of performing updates, to update the DB tables SBOOK and SFLIGHT using the update technique.

1-3-2 Insert the statement **COMMIT WORK** in the PAI module **USER_COMMAND_0200.**

1-3-3 Lock the flight and the booking by calling up the corresponding lock modules. Call up **subroutine ENQ_SFLIGHT** and **ENQ_SBOOK** in the appropriate places. If a user action calls up screen 100, release the locks.

The booking data is held in structure **WA_SBOOK**.

Unit: Organizing Database Updates

Model Solution SAPBC414S_BOOKINGS_03

PAI Modules

```
*------------------------------------------------------------------*
***INCLUDE BC414S_BOOKINGS_03I01 .
*------------------------------------------------------------------*

*&-----------------------------------------------------------------*
*&      Module  EXIT  INPUT
*&-----------------------------------------------------------------*
MODULE exit INPUT.
  CASE ok_code.
    WHEN 'CANCEL'.
      CASE sy-dynnr.
        WHEN '0100'.
          LEAVE PROGRAM.
        WHEN '0200'.
          PERFORM deq_all.
          LEAVE TO SCREEN '0100'.
        WHEN '0300'.
* remove all database locks
          PERFORM deq_all.
          LEAVE TO SCREEN '0100'.
        WHEN OTHERS.
      ENDCASE.
    WHEN 'EXIT'.
      LEAVE PROGRAM.
    WHEN OTHERS.
  ENDCASE.
ENDMODULE.                             " EXIT  INPUT
```

```
*&-----------------------------------------------------------------*
*&      Module  USER_COMMAND_0100  INPUT
*&-----------------------------------------------------------------*
MODULE user_command_0100 INPUT.
  CASE save_ok.
****************************CANCEL BOOKING**************************
    WHEN 'BOOKC'.
      PERFORM enq_sflight_sbook.
      PERFORM read_sflight USING wa_sflight sysubrc.
      PERFORM process_sysubrc_bookc.
      PERFORM read_spfli USING wa_spfli.
      PERFORM read_sbook USING itab_book itab_cd.
      REFRESH CONTROL 'TC_SBOOK' FROM SCREEN '0200'.
****************************CREATE BOOKING**************************
    WHEN 'BOOKN'.
* lock flight in Table SFLIGHT, which will be modified when new
* booking is saved
      PERFORM enq_sflight.
      PERFORM read_sflight USING wa_sflight sysubrc.
      PERFORM process_sysubrc_bookn.
      PERFORM read_spfli USING wa_spfli.
      PERFORM initialize_sbook USING wa_sbook.
    WHEN 'BACK'.
      SET SCREEN 0.
    WHEN OTHERS.
      SET SCREEN '0100'.
  ENDCASE.
ENDMODULE.                             " USER_COMMAND_0100  INPUT


*&-----------------------------------------------------------------*
*&      Module  USER_COMMAND_0200  INPUT
*&-----------------------------------------------------------------*
MODULE user_command_0200 INPUT.
  CASE save_ok.
    WHEN 'SAVE'.
      PERFORM collect_modified_data USING itab_sbook_modify.
      PERFORM save_modified_booking.
* start asynchronous update and new SAP-LUW
      COMMIT WORK
```

```
* database locks are removed by update program
      SET SCREEN '0100'.
    WHEN 'BACK'.
      PERFORM deq_all.
      SET SCREEN '0100'.
    WHEN OTHERS.
      SET SCREEN '0200'.
  ENDCASE.
ENDMODULE.                             " USER_COMMAND_0200  INPUT
```

```
*&-----------------------------------------------------------------*
*&      Module  USER_COMMAND_0300  INPUT
*&-----------------------------------------------------------------*
MODULE user_command_0300 INPUT.
  PERFORM tabstrip_set.
  CASE save_ok.
    WHEN 'NEW_CUSTOM'.
      PERFORM create_new_customer.
      SET SCREEN '0300'.
    WHEN 'SAVE'.
      PERFORM save_new_booking.
* start asynchronous update and new SAP-LUW
      COMMIT WORK.
* database locks are removed by update program
      SET SCREEN '0100'.
    WHEN 'BACK'.
* remove all database locks
      PERFORM deq_all.
      SET SCREEN '0100'.
    WHEN OTHERS.
      SET SCREEN '0300'.
  ENDCASE.
ENDMODULE.                             " USER_COMMAND_0300  INPUT
```

FORM Routines

F01

```
*------------------------------------------------------------------*
***INCLUDE BC414S_BOOKINGS_03F01 .
*------------------------------------------------------------------*

*&-----------------------------------------------------------------*
*&      Form  PROCESS_SYSUBRC_BOOKN
*&-----------------------------------------------------------------*
FORM process_sysubrc_bookn.
  CASE sysubrc.
```

```
      SET SCREEN '0300'.
    WHEN OTHERS.
* remove all database locks
      PERFORM deq_all.
      MESSAGE e023 WITH sdyn_conn-carrid sdyn_conn-connid
                        sdyn_conn-fldate.
  ENDCASE.
ENDFORM.                               " PROCESS_SYSUBRC_BOOKN
```

F04

```
*------------------------------------------------------------------*
*   INCLUDE BC414S_BOOKINGS_03F04
*------------------------------------------------------------------*

*&-----------------------------------------------------------------*
*&      Form  SAVE_MODIFIED_BOOKING
*&-----------------------------------------------------------------*
FORM save_modified_booking.
  CALL FUNCTION 'UPDATE_SBOOK' IN UPDATE TASK
       EXPORTING
            itab_sbook = itab_sbook_modify.
* no exception handling when using asynchronous update technique
  PERFORM update_sflight.
ENDFORM.                               " SAVE_MODIFIED_BOOKING


*&-----------------------------------------------------------------*
*&      Form  UPDATE_SFLIGHT
*&-----------------------------------------------------------------*
FORM update_sflight.
  CALL FUNCTION 'UPDATE_SFLIGHT' IN UPDATE TASK
       EXPORTING
            carrier    = wa_sflight-carrid
            connection = wa_sflight-connid
            date       = wa_sflight-fldate.
* no exception handling when using asynchronous update technique
ENDFORM.                               " UPDATE_SFLIGHT


*&-----------------------------------------------------------------*
*&      Form  SAVE_NEW_BOOKING
*&-----------------------------------------------------------------*
FORM save_new_booking.
  PERFORM convert_to_loc_currency USING wa_sbook.
* lock booking on DB table sbook to be created
  PERFORM enq_sbook.
  CALL FUNCTION 'INSERT_SBOOK' IN UPDATE TASK
       EXPORTING
```

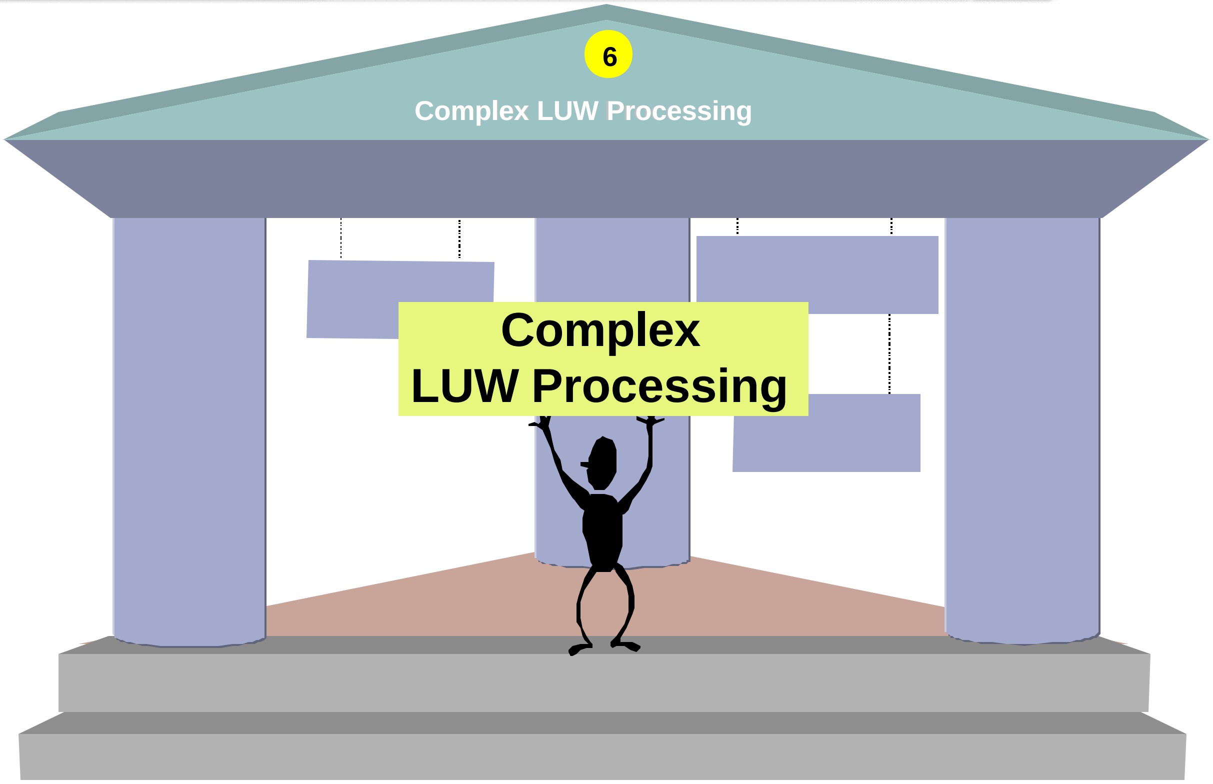
```
            wa_sbook = wa_sbook.
* no exception handling when using asynchronous update technique
  PERFORM update_sflight.
ENDFORM.                               " SAVE_NEW_BOOKING
```

Complex LUW Processing

Contents:

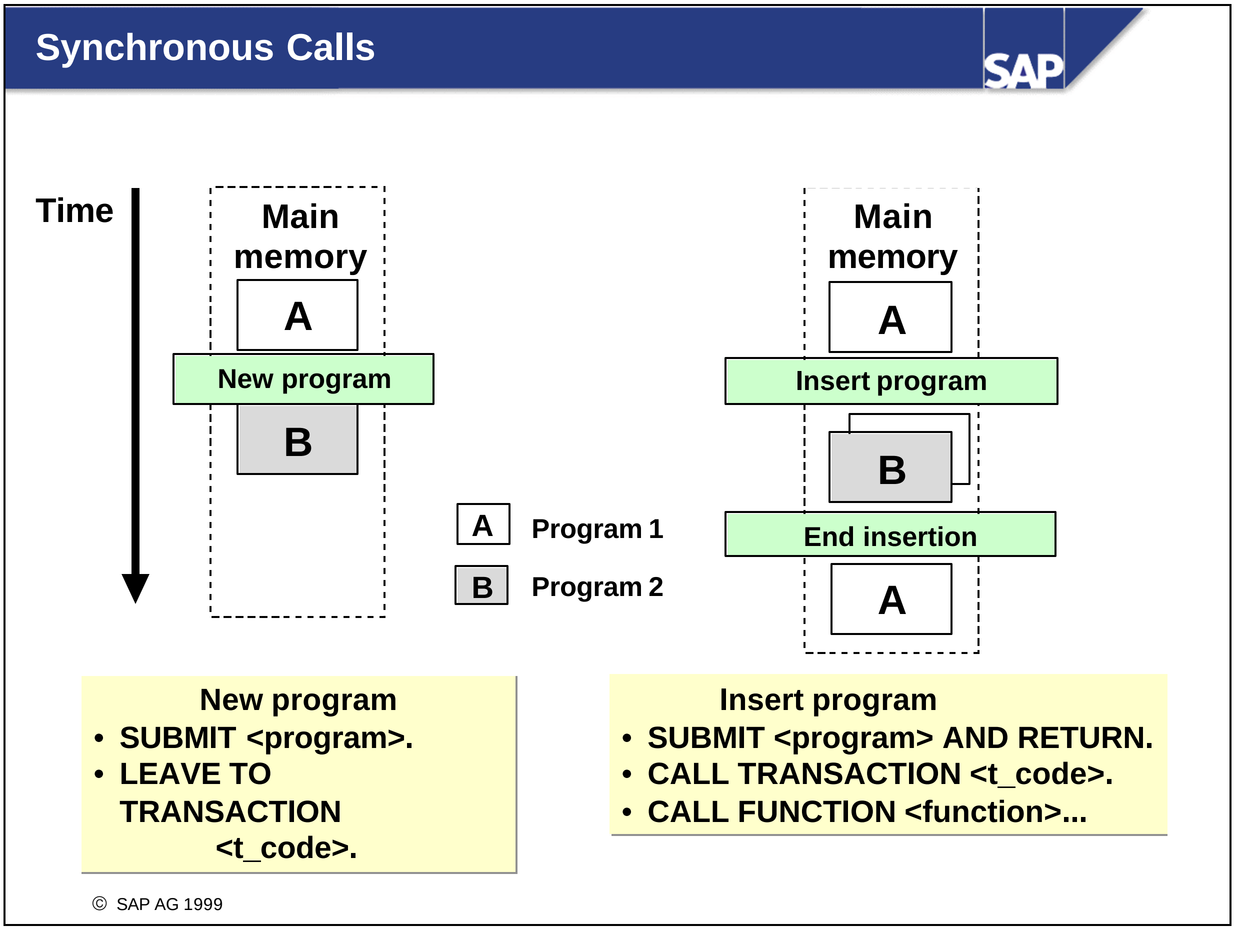
- Call techniques for programs
- The logical memory level model
- Data transfer between programs
- Complex LUWs
- Lock behavior for complex LUWs

Course Overview Diagram - Complex LUW Processing



Overview Complex LUW Processing: Call Techniques for Programs

- Call techniques for programs
- The logical memory level model
- Data transfer between programs
- LUW processing for program calls
- Locks for program calls



Synchronous Calls

Time

Main memory

A

New program

B

A Program 1

B Program 2

Main memory

A

Insert program

B

End insertion

A

New program

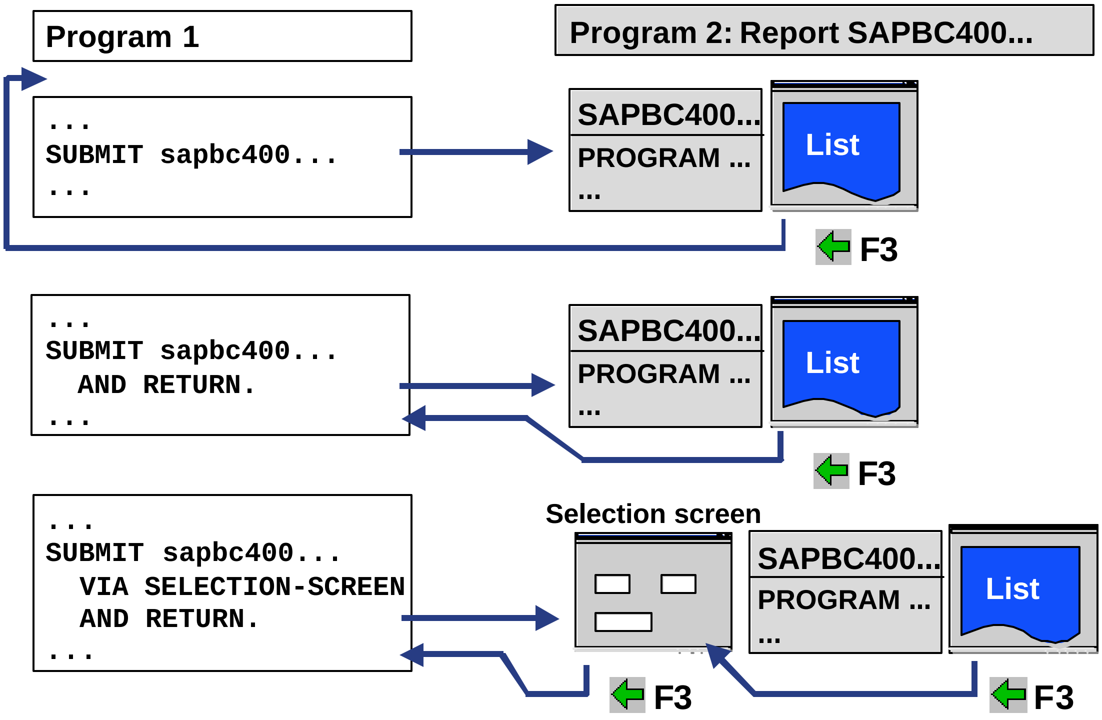
- SUBMIT <program>.
- LEAVE TO TRANSACTION <t_code>.

Insert program

- SUBMIT <program> AND RETURN.
- CALL TRANSACTION <t_code>.
- CALL FUNCTION <function>...

- In an ABAP program, there are two ways of executing another program synchronously:
  - Terminate the current program and start the other program (`SUBMIT <program>`, `LEAVE TO TRANSACTION <t_code>`)
  - Call the other program without terminating the other program. The calling program is interrupted, and the system returns to it when the program that it has called is finished (`CALL TRANSACTION`, `SUBMIT <program> AND RETURN`, `CALL FUNCTION`).
- You can only use `SUBMIT <program>` and `SUBMIT <program> AND RETURN` to start executable programs (program type 'Executable program', formerly program type '1').
- You use `LEAVE TO TRANSACTION <t_code>` and `CALL TRANSACTION <t_code>` to start programs that have a transaction code.
- You use `CALL FUNCTION <function>` to execute a function module.
- The executed commands differ with regard to the visibility of program data in the calling program and the called program, and in their behavior with regard to LUW processing.

Calling an Executable Program



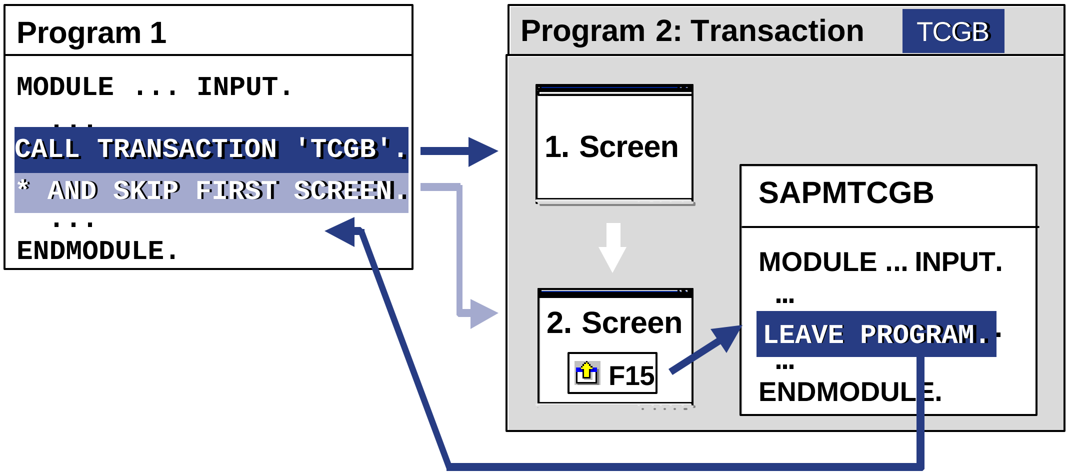
```
...
SUBMIT sapbc400...
...
```

```
...
SUBMIT sapbc400...
  AND RETURN.
...
```

```
...
SUBMIT sapbc400...
  VIA SELECTION-SCREEN
  AND RETURN.
...
```

- With the `SUBMIT` statement, you start programs that are directly executable programs.
- The addition `VIA SELECTION-SCREEN` is used to send the selection screen of the program (if the program has a standard selection screen).
- To return to the calling program after the program has finished, use the addition `AND RETURN`.
- Calling an executable program allows you to use a logical database to read data.
- For further information, see the keyword documentation in the ABAP Editor for **`SUBMIT`**.

Calling a Transaction



Program 1

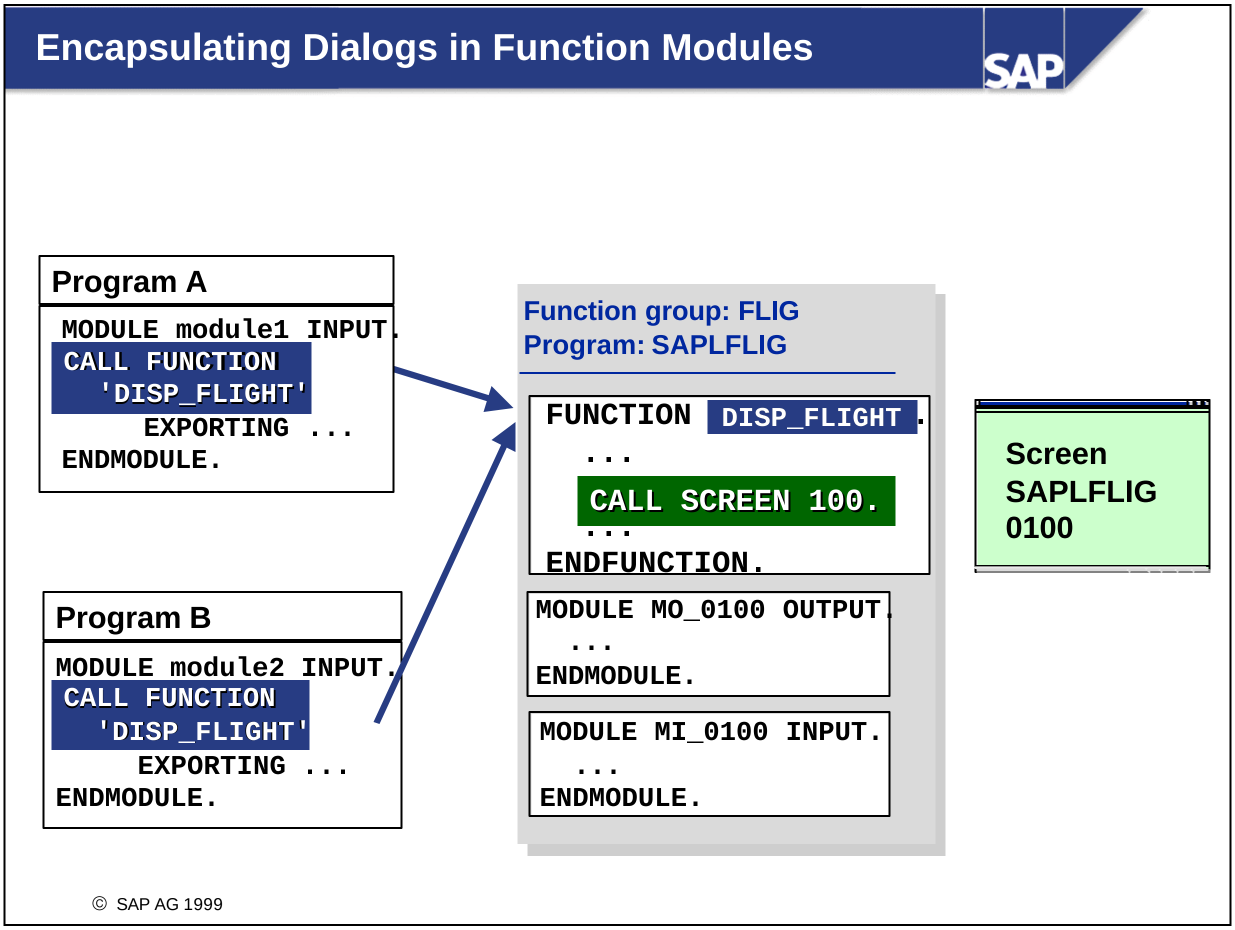
```
MODULE ... INPUT.
  ...
CALL TRANSACTION 'TCGB'.
* AND SKIP FIRST SCREEN.
  ...
ENDMODULE.
```

Program 2: Transaction TCGB

SAPMTCGB

```
MODULE ... INPUT.
  ...
LEAVE PROGRAM.
  ...
ENDMODULE.
```

- You can execute ABAP programs with a transaction code `<t_code>` using the statement `CALL TRANSACTION <t_code>.` When the program that you called has terminated, the system continues processing the calling program.
- If the transaction that you call in the `CALL TRANSACTION <t_code>` statement uses update techniques, you can determine which technique it should use (synchronous or asynchronous) using the `FUNCTION` parameter in the call.
- To exit an ABAP program, use the `LEAVE PROGRAM` statement. If you use this statement in a program that you have called using `CALL TRANSACTION <t_code>` or `SUBMIT <program> AND RETURN`, the system returns to the calling program. Otherwise, the system returns to the application menu from which you started the program.
- To initiate a transaction with the transaction code `<t_code>`, use the `LEAVE TO TRANSACTION <t_code>` statement. This does not allow you to return to where the transaction was called from. This statement has the same effect as entering `/n<t_code>` in the command field.
- For further information, see the keyword documentation in the ABAP Editor for **`CALL`** and **`LEAVE`**.



Encapsulating Dialogs in Function Modules

Program A

```
MODULE module1 INPUT.
CALL FUNCTION
   'DISP_FLIGHT'
      EXPORTING ...
ENDMODULE.
```

Program B

```
MODULE module2 INPUT.
CALL FUNCTION
   'DISP_FLIGHT'
      EXPORTING ...
ENDMODULE.
```

Function group: FLIG
Program: SAPLFLIG

```
FUNCTION DISP_FLIGHT.
  ...
  CALL SCREEN 100.
  ...
ENDFUNCTION.
```

```
MODULE MO_0100 OUTPUT.
  ...
ENDMODULE.
```

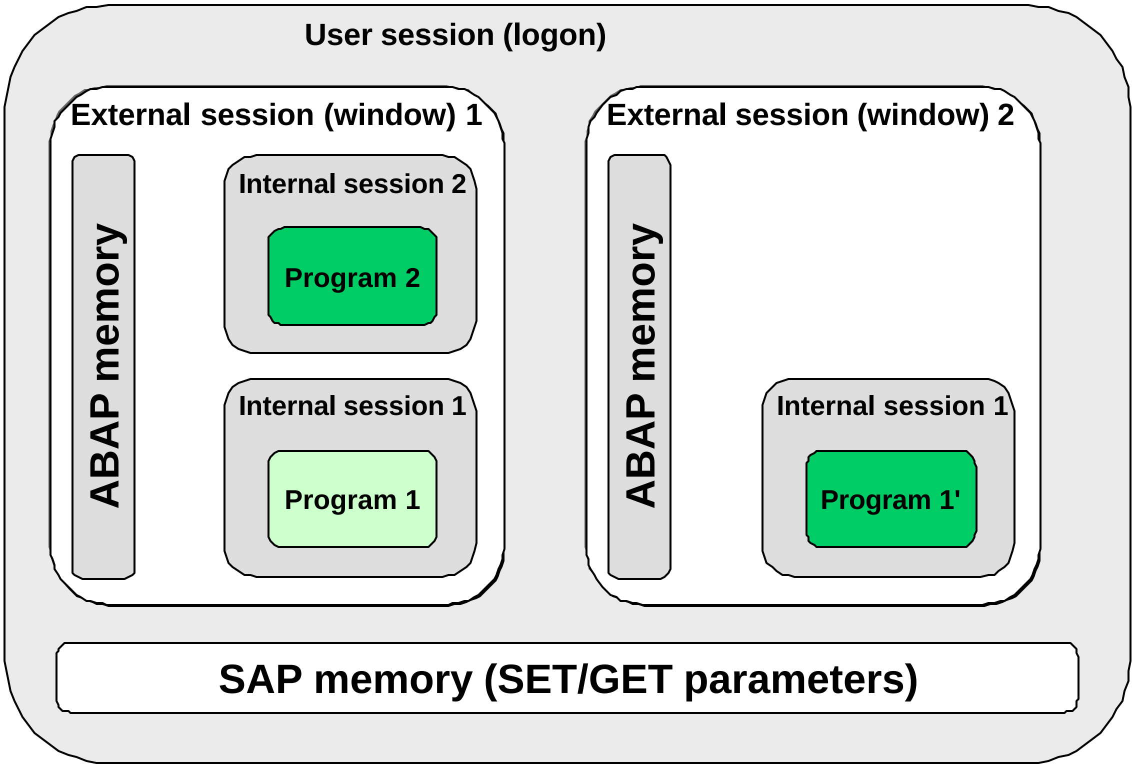
```
MODULE MI_0100 INPUT.
  ...
ENDMODULE.
```

- You can encapsulate dialogs in reusable function modules.
- If you call up a screen within a function module, this screen belongs to the program of function group `SAPL<f_group>` of the function module.

Overview of Complex LUW Processing: Logical Memory Level Model

- Call techniques for programs
- ➤ The logical memory level model
- Data transfer between programs
- LUW processing for program calls
- Locks for program calls

The Logical Memory Level Model



User session (logon)

External session (window) 1

ABAP memory

Internal session 2

Program 2

Internal session 1

Program 1

External session (window) 2

ABAP memory

Internal session 1

Program 1'

SAP memory (SET/GET parameters)

- A logical memory model illustrates how the main memory is distributed from the view of executable programs. A distinction is made here between **external sessions** and **internal sessions**.
- An external session is usually linked to an R/3 window. You can create an external session by choosing *System/Create session*, or by entering `/o<t_code>` in the command field. An external session is broken down further into internal sessions. Program data is only visible within an internal session. Each external session can include up to 20 internal sessions (stacks).
- Every program you start runs in an internal session.
- All "squares" with rounded "corners" displayed in the status diagram represent a set of data objects in the main memory.
- The data in the main memory is only visible to the program concerned.

- `CALL TRANSACTION <tcode>` and `SUBMIT <program> AND RETURN` open a new internal session that forms a new program context. The internal sessions in an external session form a memory stack. The new session is added to the top of the stack.

- When a program has finished running, the top internal session in the stack is removed, and the calling program resumes processing.
- The same occurs when the system processes a `LEAVE PROGRAM` statement.

- `LEAVE TO TRANSACTION` removes all internal sessions from the stack and opens a new one containing the program context of the calling program.
- The ABAP memory is **initialized** after the program is called. In other words, you cannot transfer any data to a program called with `LEAVE TO TRANSACTION <tcode>` via the ABAP memory.

- `SUBMIT <program>` replaces the internal session of the program performing the call with the internal session of the program that has been called. The new internal session contains the program context of the called program with which it is performed.

- When a function module is called, the following steps are executed:
  - A check is made to establish whether your program has called a function module of the same function group previously.
  - If this is not the case, the system loads the associated function group to the internal session of the calling program as an additional program group. This initializes its global data.
  - If your program used a function module of the same function group before the current call, the function module that you have called up at present can access the global data of the function group. The function group is not reloaded.
- Within the internal session, all of the function modules that you call from the same group access the global data of that group.
- If, in a new internal session, you call a function module from the same function group as in internal session 1, a new set of global data is initialized for the second internal session. This means that the data accessed by function modules called in session 2 may be different from that accessed by the function modules in session 1.

- You can call function modules asynchronously as well as synchronously. To do so, you must extend the function module call using the addition `STARTING NEW TASK '<name>'`. Here, `'<name>'` is a symbolic name in the calling program that identifies the external session, in which the called program is executed.
- Function modules that you call using the addition `STARTING NEW TASK '<name>'` are executed independently of the calling program. The calling program is not interrupted.
- To make function modules available for local asynchronous calls, you must identify them as executable remotely (processing type: *Remote-enabled module*).
- For further information, see the keyword documentation in the ABAP Editor for **`CALL FUNCTION`**.

- There are various ways of transferring data between programs that are running in different program contexts (internal sessions). You can use:

  (1) The interface of the called program (standard selection screen, or interface of a subroutine, function module, or dialog module)

  (2) ABAP memory

  (3) SAP memory

  (4) Database tables

  (5) Local files on your presentation server.

- For further information about transferring data using database tables and the *shared buffer*, refer to the keyword documentation in the ABAP Editor for the terms **`EXPORT`** and **`IMPORT`**.

- For further information about transferring data between an ABAP program and your presentation server, refer to the documentation for the function modules **`WS_UPLOAD`** and **`WS_DOWNLOAD`**.

- Function modules have an interface, which you can use to pass data between the calling program and the function module itself (there is also a comparable mechanism for ABAP subroutines). If a function module supports RFC, certain restrictions apply to its interface.
- If you are calling an ABAP program that has a standard selection screen, you can pass values to the input fields. There are two options here:
  - By using a variant of the standard selection screen in the program call
  - By passing actual values for the input fields in the program call

- If you want to call a report program without displaying its selection screen (default setting), but still want to pass values to its input fields, there is a variety of techniques that you can use.
- The `WITH` addition allows you to assign values to the parameters and select-options fields on the standard selection screen.
- If the selection screen is to be displayed when the program is called, use the addition: `VIA SELECTION-SCREEN.`
- Use the *pattern button* in the ABAP Editor to insert a program call via `SUBMIT`. The structure shows you the names of data objects that you can complete with the standard selection screen.
- For further information on working with variants and further syntax variants for the `WITH` addition, see the key word documentation in the ABAP Editor for **`SUBMIT.`**

- You can use SAP memory and ABAP memory to pass data between different programs.
- The SAP memory is a user-specific memory area for storing field values. It is available in all of the open sessions in a user's terminal session, and is reset when the terminal session ends. You can use its contents as default values for screen fields. All **external sessions** can access SAP memory. This means that it is only of limited use for passing data between internal sessions.
- The ABAP memory is also user-specific, and is local to each external session. You can use it to pass any ABAP variables (fields, structures, internal tables, complex objects) between the **internal sessions** of a single external session.
- Each external session has its own ABAP memory. When you end an external session (`/i` in the command field), the corresponding ABAP memory is released automatically.

- To copy a set of ABAP variables and their current values (data cluster) to the ABAP memory, use the `EXPORT TO MEMORY ID <id>` statement. The `<id>` (up to 32 characters) is used to identify the different data clusters.
- If you repeat an `EXPORT TO MEMORY ID <id>` statement to an existing data cluster, the new data overwrites the old.
- To copy data from ABAP memory to the corresponding fields of an ABAP program, use the `IMPORT FROM MEMORY ID <id>` statement.
- The fields, structures, internal tables, and complex objects in a data cluster in ABAP memory must be declared identically in both the program from which you exported the data and the program into which you import it.
- To release a data cluster, use the `FREE MEMORY ID <id>` statement.
- You can import just parts of a data cluster with `IMPORT`, since the objects are named in the cluster.

- In the SAP memory, you can define memory areas (SET/GET parameters, or parameter IDs), which you can then address by a name of up to 20 characters.
- You can fill these memory areas either using the contents of input/output fields on screens, or using the ABAP statement:

  ```
      SET PARAMETER ID '<parameter_id>' FIELD <value>.
  ```

  The memory area with the name <parameter_id> now has the value <value>.
- You can use the contents of a memory area to display a default value in an input field on a screen.
- You can also read the memory areas from the SAP memory using the ABAP statement GET PARAMETER ID <parameter_id> FIELD <field>. The field <field> then contains the value from parameter <parameter_id>.
- The link between an input/output field and a memory area in SAP memory is inherited from the data element on which the field is based. You can enable the set parameter or get parameter attributes in the input/output field attributes.

- Once you have set the S*et parameter* attribute for an input/output field, you can fill it with default values from SAP memory. This is particularly useful for transactions that you call from another program without displaying the initial screen. For this purpose, you must activate the Set parameter functionality for the input fields of the first screen of the transaction.
- You can:

  (1) Copy the data that is to be used for the first screen of the transaction to be called to the parameter ID in the SAP memory. To do so, use the statement `SET PARAMETER` immediately before calling the transaction.

  (2) Start the transaction using `CALL TRANSACTION <t_code>` or `LEAVE TO TRANSACTION <t_code>.` If you do not want to display the initial screen, use the `AND SKIP FIRST SCREEN` addition.

  (3) The system program that starts the transaction fills the input fields that do not already have default values and for which the *Get parameter* attribute has been set with values from SAP memory.
- The *Technical information* for the input fields in the transaction you want to call contains the names of the parameter `IDs` that you need to use.
- Parameter IDs should be entered in table `TPARA`. This happens automatically if you create them via the *Object navigator*.

- Programs that you call using the statements `SUBMIT <program>`, `LEAVE TO TRANSACTION <t_code>`, `SUBMIT <program> AND RETURN`, or `CALL TRANSACTION <t_code>` run in their own SAP LUW, and update requests receive their own update key.
- When you use `SUBMIT <program>` and `LEAVE TO TRANSACTION <t_code>`, the SAP LUW of the calling program ends. If no `COMMIT WORK` statement occurred before the program call, the update requests in the log table remain incomplete and cannot be processed. They can no longer be executed. The same applies to inline changes that you make using `PERFORM … ON COMMIT`. Data that you have written to the database using inline changes is committed the next time a new screen is displayed.
- If you use `SUBMIT <program> AND RETURN` or `CALL TRANSACTION <t_code>` to insert a program and then return to the calling program, the SAP LUW of the calling program is resumed when the called program ends. The LUW processing of calling and called programs is independent. In other words, inline changes are committed the next time a new screen is displayed. Update requests and calls using `PERFORM ... ON COMMIT` require an independent `COMMIT WORK` statement in the SAP LUW in which they are running.
- Function modules run in the same SAP LUW as the program that calls them.

- If you call transactions with nested calls, each transaction needs its own `COMMIT WORK`, since each transaction maps its own SAP LUW.
- The same applies to calling executable programs, which are called using `SUBMIT <program> AND RETURN.`

- The statement `CALL TRANSACTION` allows you to
  - Shorten the user dialog when calling using `CALL TRANSACTION <tcode> USING <itab>.`
  - Determine the type of update (asynchronous, local, or synchronous) for the transaction called. For this purpose, use the addition `CALL TRANSACTION <tcode> USING <itab> UPDATE 'update_mode'`, where `update_mode` can have the values `A` (asynchronous), `L` (local), or `S` (synchronous).
- Combining the two options enables you to call several transactions in sequence (logical chain), to reduce their screen sequence, and to postpone processing of the SAP LUW 2 until processing of the SAP LUW 1 has been completed.

- When you call a function module asynchronously using the `CALL FUNCTION <function> STARTING NEW TASK ' '` statement, it runs in its own SAP LUW.

- Programs that are executed with a `SUBMIT <program> AND RETURN` or `CALL TRANSACTION <t_code>` statement start their own LUW processing. You can use these to perform nested (complex) LUW processing.
- You can use function modules as modularization units within an SAP LUW.
- Function modules that are called asynchronously are suitable for programs that allow parallel processing of some of their components.
- All techniques are suitable for including programs with purely display functions.
- Note that a function module called with `CALL FUNCTION <f> STARTING NEW TASK` is executed as a new logon. It, therefore, sees a separate SAP memory area. You can use the interface of the function module for data transfers.
  Example: In your program, you want to call a display transaction that is displayed in a separate window (amodal). To do so, you encapsulate the transaction call in a function module, which you set as to *Remote-enabled module*. You use the function module interface to accept values that you write to the SAP memory. You then call up the transaction in the function module using `CALL TRANSACTION <tcode> AND SKIP FIRST SCREEN`. You call the function module itself asynchronously.

- Type ‘`E`' locks for nested program calls may be requested more than once from the same object. This behavior can be described as follows:
    - Lock entries from function modules called synchronously increment the cumulative counter, and are therefore successful.
    - Lock entries from programs called with `CALL TRANSACTION` or `SUBMIT <p> AND RETURN` are refused. The object to be locked by the called program is displayed as already locked by another user.
- Programs that you call using `SUBMIT <program>` or `LEAVE TO TRANSACTION <t_code>` cannot come into conflict with lock entries from the calling program, since the old program ends when the call is made. When a program ends, the system deletes all of the lock entries that it had set.
- Lock requests belonging to the same user from different R/3 windows or logons are treated as lock requests from other users.

Unit: Complex LUW Processing

At the conclusion of these exercises, you will be able to:

- Use the CALL TRANSACTION <tcode> technique for modularization at program level
- Use the SAP memory to transfer data

New bookings can be entered in program **SAPMZ##_BOOKINGS3** (see last exercise). One requirement, however, is that the posting customer is already maintained in the system.

Clicking the *Create new customer* icon (function code NEW_CUSTOM) on screen 300 will enable a customer to be created from the posting program. For this reason, program **SAPMZ##_CUSTOMER2** (transaction code: Z##_CUSTOMER2) will be called up using the CALL TRANSACTION <tcode> technique. You have created this program in exercise 1 for the *Database Updates with Open SQL* unit and enhanced it in the optional part of exercise 1 for the *SAP Lock Concept* unit.

Program:	SAPMZ##_BOOKINGS4
Transaction code:	Z##_BOOKINGS4
Template:	SAPBC414**T**_BOOKINGS_04 /
	SAPBC414**T**_CREATE_CUSTOMER_03
Model solution:	SAPBC414**S**_BOOKINGS_04 /
	SAPBC414**S**_CREATE_CUSTOMER

1-1 Copy your solution **SAPMZ##_BOOKINGS3** or the program template **SAPBC414T_BOOKINGS_04** with **all** sub-objects to **SAPMZ##_BOOKINGS4** (## is the group number). Assign transaction code **Z##_BOOKINGS4** to the program.

1-2 Copy the program template **SAPBC414T_CREATE_CUSTOMER_03** with **all** sub-objects to **SAPMZ##_CUSTOMER3** (## is the group number) and assign transaction code **Z##_CUSTOMER3** to the program.

1-3 The transaction call for creating a new customer is to be encapsulated in the **CREATE_NEW_CUSTOMER subroutine.** The subroutine is called up from the PAI module USER_COMMAND_0300 (screen 300) and is already created (blank).

1-3-1 Implement the transaction call. Call your transaction **Z##_CUSTOMER3**.

1-4 The customer number is determined in the **SAPMZ##_CUSTOMER3** program using an internal number assignment, in other words it is assigned by the application itself. The SAP memory is to be used to transfer the customer number to the calling program.

1-4-1 Change the **SAPMZ##_CUSTOMER3** program so that the customer number is written to the SAP memory after a customer has been created successfully. To which SET/GET parameter must the customer number be assigned?

1-4-2 Change the calling program **SAPMZ##_BOOKINGS4** so that the customer number appears in the appropriate field of the subscreen 301 after a customer has been created successfully.

You can display the name of the SET/GET parameter that is assigned to this field via the **F1 Help** for a screen field.

SAPMZ##_CUSTOMER3: The **customer number** is in the data object **SCUSTOM-ID**

Unit: Complex LUW Processing

Model Solution SAPBC414S_BOOKINGS_04

FORM Routines

F01

```
*-----------------------------------------------------------------*
***INCLUDE BC414S_BOOKINGS_04F01 .
*-----------------------------------------------------------------*


*&----------------------------------------------------------------*
*&      Form  CREATE_NEW_CUSTOMER
*&----------------------------------------------------------------*
FORM create_new_customer.
  CALL TRANSACTION 'BC414S_CREATE_CUST'.
* Called Transaction set the SET/GET Parameter CSM??
  GET PARAMETER ID 'CSM' field scust_id.
* scust_id <> initial -> customer created -> clear customid to get
* customer number via SET/GET Parameters
  check not scust_id is initial.
  clear: wa_sbook-customid.
ENDFORM.                               " CREATE_NEW_CUSTOMER
```

Musterlösung SAPBC414S_CREATE_CUSTOMER

FORM Routines

F01

```
*------------------------------------------------------------------*
***INCLUDE BC414S_CREATE_CUSTOMERF01 .
*------------------------------------------------------------------*

*&-----------------------------------------------------------------*
*&      Form  SAVE_SCUSTOM
*&-----------------------------------------------------------------*
FORM save_scustom.
  INSERT INTO scustom VALUES scustom.
  IF sy-subrc <> 0.
* initialize SCUSTOM-ID in SAP-MEMORY
    SET PARAMETER ID 'CSM' FIELD space.
* insertion of dataset in DB-table not possible
    MESSAGE a048.
  ELSE.
* write SCUSTOM-ID back to SAP-MEMORY
    SET PARAMETER ID 'CSM' FIELD scustom-id.
* insertion successfull
    MESSAGE s015 WITH scustom-id.
  ENDIF.
ENDFORM.                               " SAVE_SCUSTOM
```

Appendix

Content:

- Complete model solution for program: Creating customer data
- Complete model solution for program: Canceling/creating bookings
- Slide index

Solutions

Program: Generating Customer Data Records
Complete Transaction

Model Solution SAPBC414S_CREATE_CUSTOMER

Module Pool

```
*&-----------------------------------------------------------------*
*& Modulpool         SAPBC414S_CREATE_CUSTOMER                     *
*&-----------------------------------------------------------------*
INCLUDE BC414S_CREATE_CUSTOMERTOP.
INCLUDE BC414S_CREATE_CUSTOMERO01.
INCLUDE BC414S_CREATE_CUSTOMERI01.
INCLUDE BC414S_CREATE_CUSTOMERF01.
```

SCREEN 100

```
PROCESS BEFORE OUTPUT.
  MODULE status_0100.

PROCESS AFTER INPUT.
  MODULE exit AT EXIT-COMMAND.
  MODULE save_ok_code.
  FIELD: scustom-name MODULE mark_changed ON REQUEST.
  MODULE user_command_0100.
```

TOP Include

```
*&-----------------------------------------------------------------*
*& Include BC414S_CREATE_CUSTOMERTOP                               *
*&-----------------------------------------------------------------*
PROGRAM  sapbc414s_create_customer MESSAGE-ID bc414.
```

```
DATA: answer, flag.
DATA: ok_code LIKE sy-ucomm, save_ok LIKE ok_code.
TABLES: scustom.
```

PBO Modules

```
*------------------------------------------------------------------*
***INCLUDE BC414S_CREATE_CUSTOMERO01 .
*------------------------------------------------------------------*

*&-----------------------------------------------------------------*
*&      Module  STATUS_0100  OUTPUT
*&-----------------------------------------------------------------*
MODULE STATUS_0100 OUTPUT.
  SET PF-STATUS 'DYN_0100'.
  SET TITLEBAR 'DYN_0100'.
ENDMODULE.                 " STATUS_0100  OUTPUT
```

PAI Modules

```
*------------------------------------------------------------------*
***INCLUDE BC414S_CREATE_CUSTOMERI01 .
*------------------------------------------------------------------*

*&-----------------------------------------------------------------*
*&      Module  EXIT  INPUT
*&-----------------------------------------------------------------*
MODULE exit INPUT.
  CASE ok_code.
    WHEN 'EXIT'.
      IF sy-datar IS INITIAL AND flag IS INITIAL.
* no changes on screen 100
        LEAVE PROGRAM.
      ELSE.
        PERFORM ask_save USING answer.
        CASE answer.
          WHEN 'J'.
            ok_code = 'SAVE&EXIT'.
          WHEN 'N'.
            LEAVE PROGRAM.
          WHEN 'A'.
            CLEAR ok_code.
            SET SCREEN 100.
      ENDIF.
    WHEN 'CANCEL'.
      IF sy-datar IS INITIAL AND flag IS INITIAL.
* no changes on screen 100
        LEAVE TO SCREEN 0.
      ELSE.
        PERFORM ask_loss USING answer.
        CASE answer.
          WHEN 'J'.
            LEAVE TO SCREEN 0.
          WHEN 'N'.
            CLEAR ok_code.
            SET SCREEN 100.
        ENDCASE.
      ENDIF.
  ENDCASE.
ENDMODULE.                             " EXIT  INPUT
```

```
*&------------------------------------------------------------------*
*&      Module  SAVE_OK_CODE  INPUT
*&------------------------------------------------------------------*
MODULE save_ok_code INPUT.
  save_ok = ok_code.
  CLEAR ok_code.
ENDMODULE.                             " SAVE_OK_CODE  INPUT


*&------------------------------------------------------------------*
*&      Module  USER_COMMAND_0100  INPUT
*&------------------------------------------------------------------*
MODULE user_command_0100 INPUT.
  CASE save_ok.
    WHEN 'SAVE&EXIT'.
      PERFORM save.
      LEAVE PROGRAM.
    WHEN 'SAVE'.
      IF flag IS INITIAL.
        SET SCREEN 100.
      ELSE.
        PERFORM save.
        SET SCREEN 0.
      ENDIF.
    WHEN 'BACK'.
      IF flag IS INITIAL.
        SET SCREEN 0.
      ELSE.
        PERFORM ask_save USING answer.
        CASE answer.
          WHEN 'J'.
            PERFORM save.
            SET SCREEN 0.
          WHEN 'N'.
            SET SCREEN 0.
          WHEN 'A'.
            SET SCREEN 100.
        ENDCASE.
      ENDIF.
  ENDCASE.
ENDMODULE.                             " USER_COMMAND_0100  INPUT


*&------------------------------------------------------------------*
*&      Module  MARK_CHANGED  INPUT
*&------------------------------------------------------------------*
MODULE mark_changed INPUT.
* set flag to mark changes were made on screen 100
  flag = 'X'.
ENDMODULE.                             " MARK_CHANGED  INPUT
```

FORM Routines

```
*------------------------------------------------------------------*
***INCLUDE BC414S_CREATE_CUSTOMERF01 .
*------------------------------------------------------------------*

*&-----------------------------------------------------------------*
*&      Form  NUMBER_GET_NEXT
*&-----------------------------------------------------------------*
*      -->P_WA_SCUSTOM  text
*------------------------------------------------------------------*
FORM number_get_next USING p_scustom LIKE scustom.
  DATA: return TYPE inri-returncode.
* get next free number in the number range '01'
* of number range object'SBUSPID'
  CALL FUNCTION 'NUMBER_GET_NEXT'
       EXPORTING
            nr_range_nr = '01'
            object      = 'SBUSPID'
       IMPORTING
            number      = p_scustom-id
            returncode  = return
       EXCEPTIONS
            OTHERS      = 1.
  CASE sy-subrc.
    WHEN 0.
      CASE return.
        WHEN 1.
* number of remaining numbers critical
          MESSAGE s070.
        WHEN 2.
* last number
          MESSAGE s071.
        WHEN 3.
* no free number left over
          MESSAGE a072.
      ENDCASE.
    WHEN 1.
* internal error
      MESSAGE a073 WITH sy-subrc.
  ENDCASE.
ENDFORM.                               " NUMBER_GET_NEXT


*&------------------------------------------------------------------*
*&      Form  ASK_SAVE
*&------------------------------------------------------------------*
*      -->P_ANSWER  text
*-------------------------------------------------------------------*
FORM ask_save USING p_answer.
  CALL FUNCTION 'POPUP_TO_CONFIRM_STEP'
       EXPORTING
            textline1 = 'Data has been changed.'(001)
            textline2 = 'Save before leaving transaction?'(002)
            titel     = 'Create Customer'(003)
       IMPORTING
            answer    = p_answer.
ENDFORM.                               " ASK_SAVE
*&------------------------------------------------------------------*
*&      Form  ASK_LOSS
*&------------------------------------------------------------------*
*      -->P_ANSWER  text
*-------------------------------------------------------------------*
FORM ask_loss USING p_answer.
  CALL FUNCTION 'POPUP_TO_CONFIRM_LOSS_OF_DATA'
       EXPORTING
            textline1 = 'Continue?'(004)
            titel     = 'Create Customer'(003)
       IMPORTING
            answer    = p_answer.
ENDFORM.                               " ASK_LOSS


*&------------------------------------------------------------------*
*&      Form  ENQ_SCUSTOM
*&------------------------------------------------------------------*
FORM enq_scustom.
  CALL FUNCTION 'ENQUEUE_ESCUSTOM'
       EXPORTING
       EXCEPTIONS
            foreign_lock   = 1
            system_failure = 2
            OTHERS         = 3.
  CASE sy-subrc.
    WHEN 1.
* dataset allready locked
      MESSAGE e060.
    WHEN 2 OR 3.
* locking of dataset not possible for other reasons
      MESSAGE e063 WITH sy-subrc.
  ENDCASE.
ENDFORM.                               " ENQ_SCUSTOM


*&-----------------------------------------------------------------*
*&      Form  DEQ_ALL
*&-----------------------------------------------------------------*
FORM deq_all.
  CALL FUNCTION 'DEQUEUE_ALL'.
ENDFORM.                               " DEQ_ALL


*&-----------------------------------------------------------------*
*&      Form  SAVE_SCUSTOM
*&-----------------------------------------------------------------*
FORM save_scustom.
  INSERT INTO scustom VALUES scustom.
  IF sy-subrc <> 0.
* initialize SCUSTOM-ID in SAP-MEMORY
    SET PARAMETER ID 'CSM' FIELD space.
* insertion of dataset in DB-table not possible
    MESSAGE a048.
  ELSE.
* write SCUSTOM-ID back to SAP-MEMORY
    SET PARAMETER ID 'CSM' FIELD scustom-id.
* insertion successfull
    MESSAGE s015 WITH scustom-id.
  ENDIF.
```

```
*&-----------------------------------------------------------------*
*&      Form  SAVE
*&-----------------------------------------------------------------*
FORM save.
* lock dataset
  PERFORM enq_scustom.
* get SCUSTOM-ID from number range object BC_SCUSTOM
  PERFORM number_get_next USING scustom.
* save new customer
  PERFORM save_scustom.
* unlock dataset
  PERFORM deq_all.
ENDFORM.                               " SAVE
```

Solutions

Program: **Canceling/Creating Bookings**
Complete Transaction

Model Solution SAPBC414S_BOOKINGS

Module Pool

```
*&-----------------------------------------------------------------*
*& Modulpool         SAPBC414S_BOOKINGS                            *
*&-----------------------------------------------------------------*
INCLUDE bc414s_bookingstop.
INCLUDE bc414s_bookingso01.
INCLUDE bc414s_bookingsi01.
INCLUDE bc414s_bookingsf01.
INCLUDE bc414s_bookingsf02.
INCLUDE bc414s_bookingsf03.
INCLUDE bc414s_bookingsf04.
INCLUDE bc414s_bookingsf05.
INCLUDE bc414s_bookingsf06.
INCLUDE fbc414_cdocscdc.
```

SCREEN 100

```
PROCESS BEFORE OUTPUT.
  MODULE STATUS_0100.
*
PROCESS AFTER INPUT.
  MODULE EXIT AT EXIT-COMMAND.
  MODULE SAVE_OK_CODE.
  CHAIN.
* cancel booking: check if flight exists or flight can be created
    FIELD: SDYN_CONN-CARRID, SDYN_CONN-CONNID, SDYN_CONN-FLDATE.
    MODULE USER_COMMAND_0100.
  ENDCHAIN.
```

SCREEN 200

```
PROCESS BEFORE OUTPUT.
  MODULE STATUS_0200.
  MODULE TRANS_DETAILS.
  CALL SUBSCREEN SUB1 INCLUDING SY-CPROG '0201'.
  LOOP AT ITAB_BOOK INTO WA_BOOK WITH CONTROL TC_SBOOK.
    MODULE TRANS_TO_TC.
* allow only modification of bookings, that are not allready
* cancelled
    MODULE MODIFY_SCREEN.
  ENDLOOP.
*
PROCESS AFTER INPUT.
  LOOP AT ITAB_BOOK.
* mark changed bookings in internal table itab_book
    FIELD SDYN_BOOK-CANCELLED MODULE MODIFY_ITAB ON REQUEST.
  ENDLOOP.
  MODULE EXIT AT EXIT-COMMAND.
  MODULE SAVE_OK_CODE.
  MODULE USER_COMMAND_0200.
```

SCREEN 201

```
PROCESS BEFORE OUTPUT.
PROCESS AFTER INPUT.
```

SCREEN 300

```
PROCESS BEFORE OUTPUT.
  MODULE STATUS_0300.
  MODULE TABSTRIP_INIT.
  MODULE TRANS_DETAILS.
  CALL SUBSCREEN TAB_SUB INCLUDING SY-CPROG SCREEN_NO.
*
PROCESS AFTER INPUT.
  CALL SUBSCREEN TAB_SUB.
  MODULE EXIT AT EXIT-COMMAND.
  MODULE SAVE_OK_CODE.
  MODULE TRANS_FROM_0300.
  MODULE USER_COMMAND_0300.
```

SCREEN 301

```
PROCESS BEFORE OUTPUT.
  MODULE HIDE_BOOKID.
PROCESS AFTER INPUT.
```

SCREEN 302

```
PROCESS BEFORE OUTPUT.
PROCESS AFTER INPUT.
```

SCREEN 303

```
PROCESS BEFORE OUTPUT.
PROCESS AFTER INPUT.
```

TOP Include

```
*&---------------------------------------------------------------*
*& Include BC414S_BOOKINGSTOP                                    *
*&---------------------------------------------------------------*
PROGRAM  sapbc414s_bookings MESSAGE-ID bc414.

* change documents: data definitions for use of function modules
INCLUDE fbc414_cdocscdt.

* line type of internal table itab_book, used to display bookings in
* table control
TYPES: BEGIN OF wa_book_type.
INCLUDE: STRUCTURE sbook.
TYPES:   name TYPE scustom-name,
         mark,
       END OF wa_book_type.

* work area and internal table used to display bookings in table
* control
DATA: wa_book TYPE wa_book_type,
      itab_book TYPE TABLE OF wa_book_type.

* bookings to be modified on database table
DATA: itab_sbook_modify TYPE TABLE OF sbook.

* change documents: bookings before changes are performed
DATA: itab_cd TYPE TABLE OF sbook WITH NON-UNIQUE KEY
      carrid connid fldate bookid customid.

* work areas for database tables spfli, sflight, sbook.
DATA: wa_sbook TYPE sbook, wa_sflight TYPE sflight, wa_spfli TYPE
spfli.

* complex transactions: number of the customer created in the called
* transaction
data: scust_id(20).

* transport function codes from screens
DATA: ok_code TYPE sy-ucomm, save_ok LIKE ok_code.
```

```
DATA: screen_no TYPE sy-dynnr.
* used to handle sy-subrc, which is determined in form
DATA  sysubrc LIKE sy-subrc.

* transporting fields to/from screen
TABLES: sdyn_conn, sdyn_book.
* table control declaration (display bookings),
* tabstrip declaration (create booking)
CONTROLS: tc_sbook TYPE TABLEVIEW USING SCREEN '0200',
          tab TYPE TABSTRIP.
```

PBO Modules

```
*-------------------------------------------------------------------*
***INCLUDE BC414S_BOOKINGSO01 .
*-------------------------------------------------------------------*

*&------------------------------------------------------------------*
*&      Module  STATUS_0100  OUTPUT
*&------------------------------------------------------------------*
MODULE status_0100 OUTPUT.
  SET PF-STATUS 'DYN_100'.
  SET TITLEBAR 'DYN_100'.
ENDMODULE.                             " STATUS_0100  OUTPUT


*&------------------------------------------------------------------*
*&      Module  STATUS_0200  OUTPUT
*&------------------------------------------------------------------*
MODULE status_0200 OUTPUT.
  SET PF-STATUS 'DYN_200'.
  SET TITLEBAR 'DYN_200' WITH sdyn_conn-carrid sdyn_conn-connid
                              sdyn_conn-fldate.
ENDMODULE.                             " STATUS_0200  OUTPUT


*&------------------------------------------------------------------*
*&      Module  STATUS_0300  OUTPUT
*&------------------------------------------------------------------*
MODULE status_0300 OUTPUT.
  SET PF-STATUS 'DYN_300'.
  SET TITLEBAR 'DYN_300' WITH sdyn_conn-carrid sdyn_conn-connid
                              sdyn_conn-fldate.
ENDMODULE.                             " STATUS_0300  OUTPUT


*&------------------------------------------------------------------*
*&      Module  TRANS_DETAILS  OUTPUT
*&------------------------------------------------------------------*
MODULE trans_details OUTPUT.
  MOVE-CORRESPONDING: wa_spfli   TO sdyn_conn.
```

```
                      wa_sflight TO sdyn_conn,
                      wa_sbook   TO sdyn_book.
ENDMODULE.                             " TRANS_DETAILS  OUTPUT


*&-----------------------------------------------------------------*
*&      Module  TRANS_TO_TC  OUTPUT
*&-----------------------------------------------------------------*
MODULE trans_to_tc OUTPUT.
  MOVE-CORRESPONDING wa_book TO sdyn_book.
ENDMODULE.                             " TRANS_TO_TC  OUTPUT
```

```
*&---------------------------------------------------------------------*
*&      Module  MODIFY_SCREEN  OUTPUT
*&---------------------------------------------------------------------*
MODULE modify_screen OUTPUT.
  LOOP AT SCREEN.
    CHECK screen-name = 'SDYN_BOOK-CANCELLED'.
    CHECK ( NOT sdyn_book-cancelled IS INITIAL ) AND
          ( sdyn_book-mark IS INITIAL ).
    screen-input = 0.
    MODIFY SCREEN.
  ENDLOOP.
ENDMODULE.                             " MODIFY_SCREEN  OUTPUT


*&---------------------------------------------------------------------*
*&      Module  TABSTRIP_INIT  OUTPUT
*&---------------------------------------------------------------------*
MODULE tabstrip_init OUTPUT.
  CHECK tab-activetab IS INITIAL.
  tab-activetab = 'BOOK'.
  screen_no = '0301'.
ENDMODULE.                             " TABSTRIP_INIT  OUTPUT


*&---------------------------------------------------------------------*
*&      Module  HIDE_BOOKID  OUTPUT
*&---------------------------------------------------------------------*
MODULE hide_bookid OUTPUT.
* hide field displaying customer number when working with number
range
* object BS_SCUSTOM
  LOOP AT SCREEN.
    CHECK screen-name = 'SDYN_BOOK-BOOKID'.
    screen-active = 0.
    MODIFY SCREEN.
  ENDLOOP.
ENDMODULE.                             " HIDE_BOOKID  OUTPUT
```

PAI Modules

```
*-------------------------------------------------------------------*
***INCLUDE BC414S_BOOKINGSI01 .
*-------------------------------------------------------------------*

*&------------------------------------------------------------------*
*&      Module  EXIT  INPUT
*&------------------------------------------------------------------*
MODULE exit INPUT.
  CASE ok_code.
    WHEN 'CANCEL'.
      CASE sy-dynnr.
        WHEN '0100'.
          LEAVE PROGRAM.
        WHEN '0200'.
          PERFORM deq_all.
          LEAVE TO SCREEN '0100'.
        WHEN '0300'.
          PERFORM deq_all.
          LEAVE TO SCREEN '0100'.
        WHEN OTHERS.
      ENDCASE.
    WHEN 'EXIT'.
      LEAVE PROGRAM.
    WHEN OTHERS.
  ENDCASE.
ENDMODULE.                             " EXIT  INPUT

*&------------------------------------------------------------------*
*&      Module  SAVE_OK_CODE  INPUT
*&------------------------------------------------------------------*
MODULE save_ok_code INPUT.
  save_ok = ok_code.
  CLEAR ok_code.
ENDMODULE.                             " SAVE_OK_CODE  INPUT


*&------------------------------------------------------------------*
*&      Module  USER_COMMAND_0100  INPUT
```

```
*&------------------------------------------------------------------*
MODULE user_command_0100 INPUT.
  CASE save_ok.
****************************CANCEL BOOKING**************************
    WHEN 'BOOKC'.
      PERFORM enq_sflight_sbook.
      PERFORM read_sflight USING wa_sflight sysubrc.
* process returncode - if flight does not exist: e-message
      PERFORM process_sysubrc_bookc.
      PERFORM read_spfli USING wa_spfli.
      PERFORM read_sbook USING itab_book itab_cd.
      REFRESH CONTROL 'TC_SBOOK' FROM SCREEN '0200'.
****************************CREATE BOOKING**************************
    WHEN 'BOOKN'.
      PERFORM enq_sflight.
      PERFORM read_sflight USING wa_sflight sysubrc.
* process returncode - if flight does not exist: e-message
      PERFORM process_sysubrc_bookn.
      PERFORM read_spfli USING wa_spfli.
      PERFORM initialize_sbook USING wa_sbook.
    WHEN 'BACK'.
      SET SCREEN 0.
    WHEN OTHERS.
      SET SCREEN '0100'.
  ENDCASE.
ENDMODULE.                             " USER_COMMAND_0100  INPUT


*&------------------------------------------------------------------*
*&      Module  USER_COMMAND_0200  INPUT
*&------------------------------------------------------------------*
MODULE user_command_0200 INPUT.
  CASE save_ok.
    WHEN 'SAVE'.
* collect marked (changed) data sets in seperate internal table
      PERFORM collect_modified_data USING itab_sbook_modify.
* perform database changes
      PERFORM save_modified_booking.
* create change documents
```

```
      COMMIT WORK.
* Unlocking data sets is executed by the update program !!
      SET SCREEN '0100'.
    WHEN 'BACK'.
      PERFORM deq_all.
      SET SCREEN '0100'.
    WHEN OTHERS.
      SET SCREEN '0200'.
  ENDCASE.
ENDMODULE.                             " USER_COMMAND_0200  INPUT


*&-----------------------------------------------------------------*
*&      Module  MODIFY_ITAB  INPUT
*&-----------------------------------------------------------------*
MODULE modify_itab INPUT.
  wa_book-cancelled = sdyn_book-cancelled.
  wa_book-mark = 'X'.
  MODIFY itab_book FROM wa_book INDEX tc_sbook-current_line.
ENDMODULE.                             " MODIFY_ITAB  INPUT
```

```
*&-----------------------------------------------------------------*
*&      Module  USER_COMMAND_0300  INPUT
*&-----------------------------------------------------------------*
MODULE user_command_0300 INPUT.
  PERFORM tabstrip_set.
  CASE save_ok.
    WHEN 'NEW_CUSTOM'.
      PERFORM create_new_customer.
      SET SCREEN '0300'.
    WHEN 'SAVE'.
      PERFORM save_new_booking.
      COMMIT WORK.
* Unlocking data sets is executed by the update program !!
      SET SCREEN '0100'.
    WHEN 'BACK'.
      PERFORM deq_all.
      SET SCREEN '0100'.
    WHEN OTHERS.
      SET SCREEN '0300'.
  ENDCASE.
ENDMODULE.                             " USER_COMMAND_0300  INPUT


*&-----------------------------------------------------------------*
*&      Module  TRANS_FROM_0300  INPUT
*&-----------------------------------------------------------------*
MODULE trans_from_0300 INPUT.
  MOVE-CORRESPONDING sdyn_book TO wa_sbook.
ENDMODULE.                             " TRANS_FROM_0300  INPUT
```

FORM Routines

F01

```
*------------------------------------------------------------------*
***INCLUDE BC414S_BOOKINGSF01 .
*------------------------------------------------------------------*


*&-----------------------------------------------------------------*
*&      Form  COLLECT_MODIFIED_DATA
*&-----------------------------------------------------------------*
*      -->P_ITAB_SBOOK_MODIFY  text
*------------------------------------------------------------------*
FORM collect_modified_data USING p_itab_sbook_modify
                                 LIKE itab_sbook_modify.
  DATA: wa_book LIKE LINE OF itab_book,
        wa_sbook_modify LIKE LINE OF p_itab_sbook_modify.
  CLEAR: p_itab_sbook_modify.
* Only bookings are collected, that
* 1) have been changed (mark = 'X')
* 2) shall be cancelled (cancelled = 'X')
  LOOP AT itab_book INTO wa_book
       WHERE    mark = 'X'
       AND cancelled = 'X'.
    MOVE-CORRESPONDING wa_book TO wa_sbook_modify.
    APPEND wa_sbook_modify TO p_itab_sbook_modify.
  ENDLOOP.
ENDFORM.                               " COLLECT_MODIFIED_DATA



*&-----------------------------------------------------------------*
*&      Form  INITIALIZE_SBOOK
*&-----------------------------------------------------------------*
*      -->P_WA_SBOOK  text
*------------------------------------------------------------------*
FORM initialize_sbook USING p_wa_sbook TYPE sbook.
  CLEAR p_wa_sbook.
  MOVE-CORRESPONDING wa_sflight TO p_wa_sbook.
  MOVE: wa_sflight-price    TO p_wa_sbook-forcurram,
        wa_sflight-currency TO p_wa_sbook-forcurkey,
ENDFORM.                                " INITIALIZE_SBOOK


*&------------------------------------------------------------------*
*&      Form  PROCESS_SYSUBRC_BOOKC
*&------------------------------------------------------------------*
FORM process_sysubrc_bookc.
  CASE sysubrc.
    WHEN 0.
      SET SCREEN '0200'.
    WHEN OTHERS.
      PERFORM deq_all.
      MESSAGE e023 WITH sdyn_conn-carrid sdyn_conn-connid
                        sdyn_conn-fldate.
  ENDCASE.
ENDFORM.                                " PROCESS_SYSUBRC_BOOKC
```

```
*&------------------------------------------------------------------*
*&      Form  PROCESS_SYSUBRC_BOOKN
*&------------------------------------------------------------------*
FORM process_sysubrc_bookn.
  CASE sysubrc.
    WHEN 0.
      SET SCREEN '0300'.
    WHEN OTHERS.
      PERFORM deq_all.
      MESSAGE e023 WITH sdyn_conn-carrid sdyn_conn-connid
                        sdyn_conn-fldate.
  ENDCASE.
ENDFORM.                               " PROCESS_SYSUBRC_BOOKN


*&------------------------------------------------------------------*
*&      Form  TABSTRIP_SET
*&------------------------------------------------------------------*
FORM tabstrip_set.
  IF save_ok = 'BOOK' OR save_ok = 'DETCON' OR save_ok = 'DETFLT'.
    tab-activetab = save_ok.
  ENDIF.
  CASE save_ok.
    WHEN 'BOOK'.
      screen_no = '0301'.
    WHEN 'DETCON'.
      screen_no = '0302'.
    WHEN 'DETFLT'.
      screen_no = '0303'.
  ENDCASE.
ENDFORM.                               " TABSTRIP_SET


*&------------------------------------------------------------------*
*&      Form  NUMBER_GET_NEXT
*&------------------------------------------------------------------*
*      -->P_WA_SBOOK  text
*-------------------------------------------------------------------*
FORM number_get_next USING p_wa_sbook LIKE sbook.
```

```
  DATA: return TYPE inri-returncode.
* get next free number in the number range '01' of number range
* object 'SBOOKID'
  CALL FUNCTION 'NUMBER_GET_NEXT'
       EXPORTING
            nr_range_nr = '01'
            object      = 'SBOOKID'
            subobject   = p_wa_sbook-carrid
       IMPORTING
            number      = p_wa_sbook-bookid
            returncode  = return
       EXCEPTIONS
            OTHERS      = 1.
  CASE sy-subrc.
    WHEN 0.
      CASE return.
        WHEN 1.
* number of remaining numbers critical
          MESSAGE s070.
        WHEN 2.
* last number
          MESSAGE s071.
        WHEN 3.
* no free number left over
          MESSAGE a072.
      ENDCASE.
    WHEN 1.
* internal error
      MESSAGE a073 WITH sy-subrc.
  ENDCASE.
ENDFORM.                               " NUMBER_GET_NEXT


*&-----------------------------------------------------------------*
*&      Form  CREATE_NEW_CUSTOMER
*&-----------------------------------------------------------------*
FORM create_new_customer.
  CALL TRANSACTION 'BC414S_CREATE_CUST'.
* Called Transaction set the SET/GET Parameter CSM??
* scust_id <> initial -> customer created -> clear scustomid to get
* customer number via SET/GET Parameters
  CHECK NOT scust_id IS INITIAL.
  CLEAR: wa_sbook-customid.
ENDFORM.                               " CREATE_NEW_CUSTOMER
```

F02

```
*------------------------------------------------------------------*
*   INCLUDE BC414S_BOOKINGSF02
*------------------------------------------------------------------*

*&-----------------------------------------------------------------*
*&      Form  ENQ_SFLIGHT
*&-----------------------------------------------------------------*
FORM enq_sflight.
  CALL FUNCTION 'ENQUEUE_ESFLIGHT'
       EXPORTING
            carrid         = sdyn_conn-carrid
            connid         = sdyn_conn-connid
            fldate         = sdyn_conn-fldate
       EXCEPTIONS
            foreign_lock   = 1
            system_failure = 2
            OTHERS         = 3.
  CASE sy-subrc.
    WHEN 0.
    WHEN 1.
      MESSAGE e060.
    WHEN OTHERS.
      MESSAGE e063 WITH sy-subrc.
  ENDCASE.
ENDFORM.                               " ENQ_SFLIGHT
*&-----------------------------------------------------------------*
*&      Form  ENQ_SFLIGHT_SBOOK
*&-----------------------------------------------------------------*
FORM enq_sflight_sbook.
  CALL FUNCTION 'ENQUEUE_ESFLIGHT_SBOOK'
```

```
       EXPORTING
            carrid         = sdyn_conn-carrid
            connid         = sdyn_conn-connid
            fldate         = sdyn_conn-fldate
       EXCEPTIONS
            foreign_lock   = 1
            system_failure = 2
            OTHERS         = 3.
  CASE sy-subrc.
    WHEN 0.
    WHEN 1.
      MESSAGE e062.
    WHEN OTHERS.
      MESSAGE e063 WITH sy-subrc.
  ENDCASE.
ENDFORM.                               " ENQ_SFLIGHT_SBOOK


*&-----------------------------------------------------------------*
*&      Form  ENQ_SBOOK
*&-----------------------------------------------------------------*
FORM enq_sbook.
  CALL FUNCTION 'ENQUEUE_ESBOOK'
       EXPORTING
            carrid         = sdyn_book-carrid
            connid         = sdyn_book-connid
            fldate         = sdyn_book-fldate
            bookid         = sdyn_book-bookid
            customid       = sdyn_book-customid
       EXCEPTIONS
            foreign_lock   = 1
            system_failure = 2
            OTHERS         = 3.
  CASE sy-subrc.
    WHEN 0.
    WHEN 1.
      MESSAGE e061.
    WHEN OTHERS.
      MESSAGE e063 WITH sy-subrc.
```

```
ENDFORM.                               " ENQ_SBOOK


*&-----------------------------------------------------------------*
*&      Form  DEQ_ALL
*&-----------------------------------------------------------------*
FORM deq_all.
  CALL FUNCTION 'DEQUEUE_ALL'.
ENDFORM.                               " DEQ_ALL
```

F03

```
*------------------------------------------------------------------*
*   INCLUDE BC414S_BOOKINGSF03
*------------------------------------------------------------------*

*&-----------------------------------------------------------------*
*&      Form  READ_SFLIGHT
*&-----------------------------------------------------------------*
*      -->P_WA_SFLIGHT  text
*      -->P_SYSUBRC     text
*------------------------------------------------------------------*
FORM read_sflight USING p_wa_sflight TYPE sflight
                        p_sysubrc LIKE sy-subrc.
  SELECT SINGLE * FROM sflight INTO p_wa_sflight
         WHERE carrid = sdyn_conn-carrid
         AND   connid = sdyn_conn-connid
         AND   fldate = sdyn_conn-fldate.
  p_sysubrc = sy-subrc.
ENDFORM.                               " READ_SFLIGHT


*&-----------------------------------------------------------------*
*&      Form  READ_SBOOK
*&-----------------------------------------------------------------*
*      -->P_ITAB_BOOK  text
*      -->P_ITAB_CD    text
*------------------------------------------------------------------*
```

```
FORM read_sbook USING p_itab_book LIKE itab_book
                      p_itab_cd   LIKE itab_cd.
  TYPES: BEGIN OF wa_custom_type,
           id TYPE scustom-id,
           name TYPE scustom-name,
         END OF wa_custom_type.
  DATA: wa_custom TYPE wa_custom_type,
        itab_custom TYPE STANDARD TABLE OF wa_custom_type
        WITH NON-UNIQUE KEY id,
        wa_book LIKE LINE OF p_itab_book,
        wa_cd   LIKE LINE OF p_itab_cd.
  CLEAR: p_itab_book, p_itab_cd.
* Select customer names in buffer table (array fetch)
  SELECT id name FROM scustom INTO CORRESPONDING FIELDS
         OF TABLE itab_custom.
* Select all bookings on selected flight (array fetch)
  SELECT * FROM sbook INTO CORRESPONDING FIELDS OF TABLE p_itab_book
         WHERE carrid = sdyn_conn-carrid
         AND   connid = sdyn_conn-connid
         AND   fldate = sdyn_conn-fldate.
* read customer names corresponding to customer number from buffer
* table
  LOOP AT p_itab_book INTO wa_book.
    READ TABLE itab_custom INTO wa_custom WITH TABLE KEY
                                            id = wa_book-customid.
    wa_book-name = wa_custom-name.
    MODIFY p_itab_book FROM wa_book.
    MOVE-CORRESPONDING wa_book TO wa_cd.
    APPEND wa_cd TO p_itab_cd.
  ENDLOOP.
  SORT p_itab_book BY bookid customid.
ENDFORM.                               " READ_SBOOK


*&-----------------------------------------------------------------*
*&      Form  READ_SPFLI
*&-----------------------------------------------------------------*
*      -->P_WA_SPFLI  text
*------------------------------------------------------------------*
FORM read_spfli USING p_wa_spfli TYPE spfli
```

```
  SELECT SINGLE * FROM spfli INTO p_wa_spfli
         WHERE carrid = sdyn_conn-carrid
         AND   connid = sdyn_conn-connid.
  IF sy-subrc <> 0.
    PERFORM deq_all.
    MESSAGE e022 WITH sdyn_conn-carrid sdyn_conn-connid.
  ENDIF.
ENDFORM.                               " READ_SPFLI
```

F04

```
*-------------------------------------------------------------------*
*   INCLUDE BC414S_BOOKINGSF04
*-------------------------------------------------------------------*

*&------------------------------------------------------------------*
*&      Form  SAVE_MODIFIED_BOOKING
*&------------------------------------------------------------------*
FORM save_modified_booking.
* Modify data on database tables sbook and sflight
  CALL FUNCTION 'UPDATE_SBOOK' IN UPDATE TASK
       EXPORTING
            itab_sbook = itab_sbook_modify.
  PERFORM update_sflight.
ENDFORM.                               " SAVE_MODIFIED_BOOKING


*&------------------------------------------------------------------*
*&      Form  UPDATE_SFLIGHT
*&------------------------------------------------------------------*
FORM update_sflight.
  CALL FUNCTION 'UPDATE_SFLIGHT' IN UPDATE TASK
       EXPORTING
            carrier    = wa_sflight-carrid
            connection = wa_sflight-connid
            date       = wa_sflight-fldate.
ENDFORM.                               " UPDATE_SFLIGHT
```

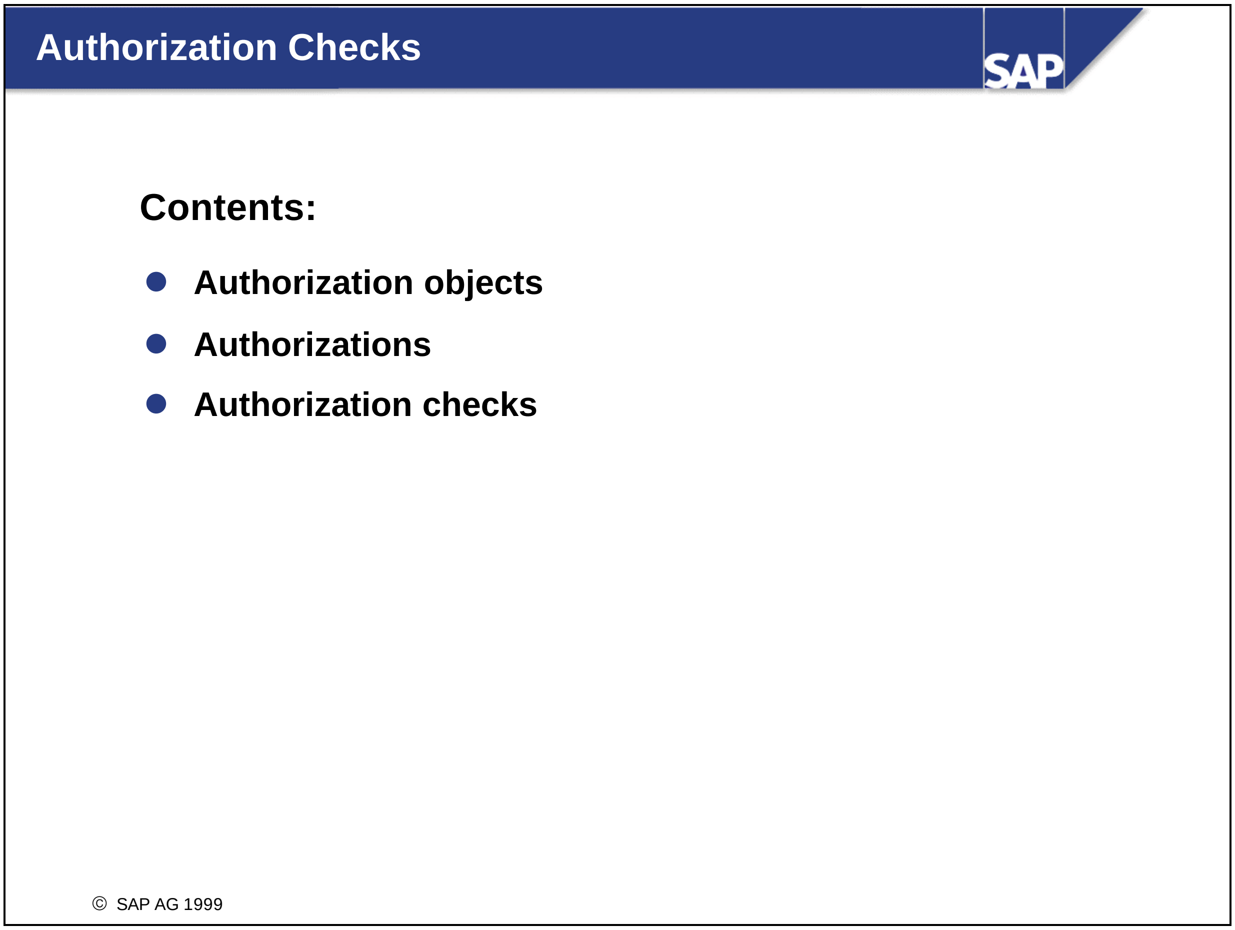
```
*&-------------------------------------------------------------------*
*&      Form  SAVE_NEW_BOOKING
*&-------------------------------------------------------------------*
FORM save_new_booking.
* transform amount from foreign currency to local currency (of
carrier)
  PERFORM convert_to_loc_currency USING wa_sbook.
* number ranges: Get next number (internal)
  PERFORM number_get_next USING wa_sbook.
* lock booking to be created
  PERFORM enq_sbook.
  CALL FUNCTION 'INSERT_SBOOK' IN UPDATE TASK
       EXPORTING
            wa_sbook = wa_sbook.
  PERFORM update_sflight.
ENDFORM.                               " SAVE_NEW_BOOKING
```

F05

```
*--------------------------------------------------------------------*
*   INCLUDE BC414S_BOOKINGSF05
*--------------------------------------------------------------------*

*&-------------------------------------------------------------------*
*&      Form  CONVERT_TO_LOC_CURRENCY
*&-------------------------------------------------------------------*
*      -->P_WA_SBOOK  text
*--------------------------------------------------------------------*
FORM convert_to_loc_currency USING p_wa_sbook TYPE sbook.
  SELECT SINGLE currcode FROM scarr INTO p_wa_sbook-loccurkey
         WHERE carrid = p_wa_sbook-carrid.
  CALL FUNCTION 'CONVERT_TO_LOCAL_CURRENCY_N'
       EXPORTING
            client           = sy-mandt
            date             = sy-datum
            foreign_amount   = p_wa_sbook-forcuram
            foreign_currency = p_wa_sbook-forcurkey
            local_currency   = p_wa_sbook-loccurkey
       IMPORTING
            local_amount     = p_wa_sbook-loccuram
       EXCEPTIONS
            no_rate_found    = 1
            overflow         = 2
            no_factors_found = 3
            no_spread_found  = 4
            derived_2_times  = 5
            OTHERS           = 6.
  IF sy-subrc <> 0.
    MESSAGE e080 WITH sy-subrc.
  ENDIF.
ENDFORM.                               " CONVERT_TO_LOC_CURRENCY
```

F06

```
*-------------------------------------------------------------------*
*   INCLUDE BC414S_BOOKINGSF06
*-------------------------------------------------------------------*

*&------------------------------------------------------------------*
*&      Form  CREATE_CHANGE_DOCUMENTS
*&------------------------------------------------------------------*
FORM create_change_documents.
  LOOP AT itab_sbook_modify INTO sbook.
* read unchanged data from buffer table into *-work area
    READ TABLE itab_cd FROM sbook INTO *sbook.
* define objectid from key fields of sbook
    CONCATENATE sbook-mandt sbook-carrid sbook-connid
                sbook-fldate sbook-bookid sbook-customid
                INTO objectid SEPARATED BY space.
* fill interface parameters of function, which itself is encapsulated
* in form CD_CALL_BC_BOOK
    MOVE: sy-tcode        TO tcode,
          sy-uzeit        TO utime,
          sy-datum        TO udate,
          sy-uname        TO username,
          'U'             TO upd_sbook.
* perform calls the neccessary function to create change document
* 'in update task'
    PERFORM cd_call_bc_book.
  ENDLOOP.
ENDFORM.                              " CREATE_CHANGE_DOCUMENTS
```



Authorization Checks

Contents:

- **Authorization objects**
- **Authorizations**
- **Authorization checks**

- You can check authorizations using the SAP authorization concept.
- The authorization concept uses **authorization objects** and **authorizations**.
- Authorization objects are repository objects and are maintained in the ABAP Dictionary. They consist of a name and up to ten logically-related fields that are used in the authorization check. Authorization objects define a logical grouping of fields whose values will be used in the authorization check. The above example uses the authorization object S_CARRID, which combines airline (CARRID) and activity (ACTVT, with the four possible values create, change, display, and delete).
- An authorization for an authorization object is a concrete set of values for the fields of an authorization object.
- Authorizations are grouped by profiles (business activities), which are assigned to users in their user master records.
- For further information, see the ABAP Editor keyword documentation for the term *Authorization concept*.

- In an **authorization check**, you specify the object and values that the user needs in an authorization in his or her user master record.
- In our example, we want to check whether the user has authorization for the object S_CARRID in which the field CARRID (airline) has the value 'LH' and the field ACTVT (activity) has the value '03' for 'display'. The activity codes are listed in tables **TACT** and **TACTZ** and are also documented in the relevant authorization objects.
- In the AUTHORITY CHECK, you must specify **all** fields of the object, otherwise, the return code will be unequal to zero. If you do not want to perform a check for one field, enter DUMMY in the field.
- The most important return codes for the AUTHORITY-CHECK statement are:
  - 0: The user has an authorization with the correct values.
  - 4: The user does not have the required authorization.
  - 8: You did not list all of the fields in the authorization object, so the check was unsuccessful.
- For a full list of all return codes, see the keyword documentation in the ABAP Editor for `AUTHORITY-CHECK`.
- You can only enter single fields after the FIELD addition, not selection tables. However, there are function modules that can perform an AUTHORITY-CHECK for all values in a selection table.
- Use the model for the AUTHORITY-CHECK in the ABAP Editor. This model inserts all names of the authorization object fields.

- The R/3 System contains tools that help you to administer authorizations and assign them to user master records.
- Authorizations are always assigned to a user using **authorization profiles**.
- Authorization profiles consist of a set of authorizations and are used to administer authorizations that are required for a particular activity (work center description).

- When you call a transaction using its transaction code, a system program starts to perform automatic authorization checks.
- Firstly, a system program checks whether the transaction is listed in the table TSTC and whether it is locked. Using the entries in the TSTC table, the system program determines the name of the ABAP program and the number of the first screen.
- Next, the system program uses the authorization object S_TCODE to see whether the user is authorized to use the transaction.
- After that, it checks whether a particular field of an authorization object is assigned to the transaction. The user calling the transaction must have an authorization for the authorization object listed in table TSTCA in his or her user master record, that also contains the values specified in table TSTCA.
- If the user has this authorization, the system starts the transaction. If not, the transaction is not started, and the system displays an error message.
- After this, the authorization checks in the ABAP program (`AUTHORITY-CHECK`) are processed.

Section: Enhancements and Modifications

Content: Enhancements and Modifications

Unit **Course Overview**

Unit **Changing the SAP Standard**

Unit **Personalization**

Unit **Enhancements to Dictionary Elements**

Unit **Enhancements using Customer Exits**

Unit **Business Transaction Events**

Unit **Business Add-Ins**

Unit **Modifications**

Unit **Summary**

Introduction: Contents

- Course goal
- Course objectives
- Course contents
- Course overview diagram
- Main business scenario
- Introduction

Changing the SAP Standard

Contents:

- **Overview of the Change Levels**
- **Decision diagram**
- **Change techniques**

- You can adjust the R/3 System to meet your needs in the following ways:
  - **Customizing**: This means setting up specific business processes and functions for your system according to an implementation guide. The need for these changes has already been foreseen by SAP and an implementation procedure has been developed.
  - **Personalization**: This means making changes to certain fields' global display attributes (setting default values or fading fields out altogether), as well as creating user-specific menu sequences.
  - **Modifications**: These are changes to SAP Repository objects made at the customer site. If SAP delivers a changed version of the object, the customer's system must be adjusted to reflect these changes. Prior to Release 4.0B these adjustments had to be made manually using upgrade utilities. From Release 4.5A, this procedure has been automated with the Modification Assistant.
  - **Enhancements**: This means creating Repository objects for individual customers that refer to objects that already exist in the SAP Repository.
  - Customer Developments: This means creating Repository objects unique to individual customers in a specific namespace reserved for new customer objects.
- Customizing and most personalization is done using tools found in AcceleratedSAP; customer developments, enhancements, and modifications are all made using the tools available in the ABAP Workbench.

- If your requirements cannot be met by Customizing or personalization, you may either start a development project or try using a CSP solution (= Complementary Software Product).
- A development project falls into the customer development category if the SAP standard does not already contain functions similar to the one you are trying to develop. If, however, a similar SAP function exists, try to assimilate it into your development project by either enhancing or modifying it, by using a user exit, or simply by making a copy the appropriate SAP program.
- Modifications can create problems, as new versions of SAP objects must be adjusted after an upgrade to coincide with modified versions of SAP objects you have created. Prior to Release 4.0B these adjustments had to be made manually using upgrade utilities. From Release 4.5A, this procedure has been automated with the Modification Assistant.
- Thus, you should only make modifications if:
  - Customizing or personalizing cannot satisfy your requirements
  - Enhancements or user exits are not planned
  - It would not make sense to copy the SAP object to the customer namespace.

- The Business Engineer is made up of all SAP **implementation tools**. These include:
    - **The R/3 Reference Model**
      contains all of the models used to describe R/3 (the process model, the data model, the organization model)
    - **The Implementation Guide** (IMG)
    - A complete list of all Customizing changes

- Personalization accelerates and simplifies how business cases are processed by the R/3 System. During personalization, individual application transactions are adjusted to meet the business needs of your company as a whole or even to the needs of specific user groups within your company. All unnecessary information and functions found in the transaction are switched off.
- Global display attributes allow you to define default values for specific screen fields. You can also suppress individual fields or table control columns in a particular transaction, or even a whole screen.
- Role-based menus, favorites, and shortcuts allow you to adjust **menu sequences** to reflect the needs of different user groups within your company.

- Modifications are changes to SAP objects in customer systems. They are:
    - executed with the help of user exits (these are subroutines reserved for customers that have been inserted in objects in the SAP namespace)
    - 'hard-coded' at various points within SAP Repository objects.
- Customer developments are programs developed by customers that can call SAP Repository objects. Example: A customer creates a program that calls an SAP function module.
- The enhancement concepts embody the reverse of this principle: SAP programs call Repository objects that you, as a customer, created or changed. Example: You use a function module exit called by an SAP program. You can enhance your system at the following levels:
    - in ABAP programs (**function module exit**)
    - on GUI interfaces (**menu exit**)
    - on screens by inserting a subscreen in an area specified by SAP (**screen exit**)
    - on screens by processing customer code that refers to a specific field on the screen (**field exit**)
    - in ABAP Dictionary tables or structures (**table enhancement**)

- SAP provides two ways to enhance tables and structures with fields.
  - Structures
  - Customizing includes ("CI includes")
- Both techniques allow you to attach fields to a table without actually having to modify the table itself.
- Append structures may only be assigned to a single table. A table may, however, have several append structures attached to it. During activation, the system searches for all active append structures for that table and attaches them to the table.
- Append structures differ from include structures in how they refer to their tables. In order to include fields from an include structure in a table, you must add an '.INCLUDE...' line to the table. In this case, the table refers to the substructure. Append structures, on the other hand, refer to their tables. In this case, the tables themselves are not altered in any way by the reference.

- Append structures allow you to attach fields to a table without actually having to modify the table itself. Table enhancements using append structures therefore do not have to be planned by SAP developers. An append structure can only belong to exactly one table.
- In contrast, CI_includes allow you to use the same structure in multiple tables. The include statement must already exist in the SAP table or structure. Table enhancements using CI_includes do, however, have to be planned by SAP developers.

- Field exits need not be prepared by the SAP application developer. You can create a field exit for any screen input field that has a Dictionary reference. The reference object is the data element.
- The unit "Enhancements to Dictionary Elements" discusses how the field exits work.

- The purpose of a program enhancement is always to call an object in the customer namespace. You can use the following techniques here:
    - A special exit function module is called by the SAP application program. The function module is part of a function group that is handled in a special manner by the system.
    - Business transaction events
      The SAP application program dynamically calls a function module in the customer namespace.
    - Business add-ins
      The application program calls a method of a class or instance of a class. This class lies in the customer namespace.

- Program enhancements permit you to execute additional program logic in SAP application programs. SAP currently provides the techniques outlined above.
- The advantages and restrictions of the particular enhancement techniques will be discussed in detail in later units.

- Menu enhancements permit you to add additional menu entries to an SAP standard menu. The system provides two options here:
  - Customer exits
  - Business add-ins
- The additional menu entries are merged into the GUI interface.
- When the function code is implemented, you can change the text of the menu entry, and - provided the SAP developer specified this option - change the icons.

- Screen exits belong to the customer exits. They allow you to display additional objects in an SAP application program screen. The SAP developer must:
  - Define the subscreen areas
  - Specify the corresponding calls in the flow logic
  - Provide the framework for the data transport
  - Include the screen exit in an enhancement
  - Maintain the documentation
- You can implement screen exits by creating subscreens, possibly with flow logic. You also have to implement the data transport.
- How you implement screen exits will be discussed in the "Enhancements using Customer Exits" unit.

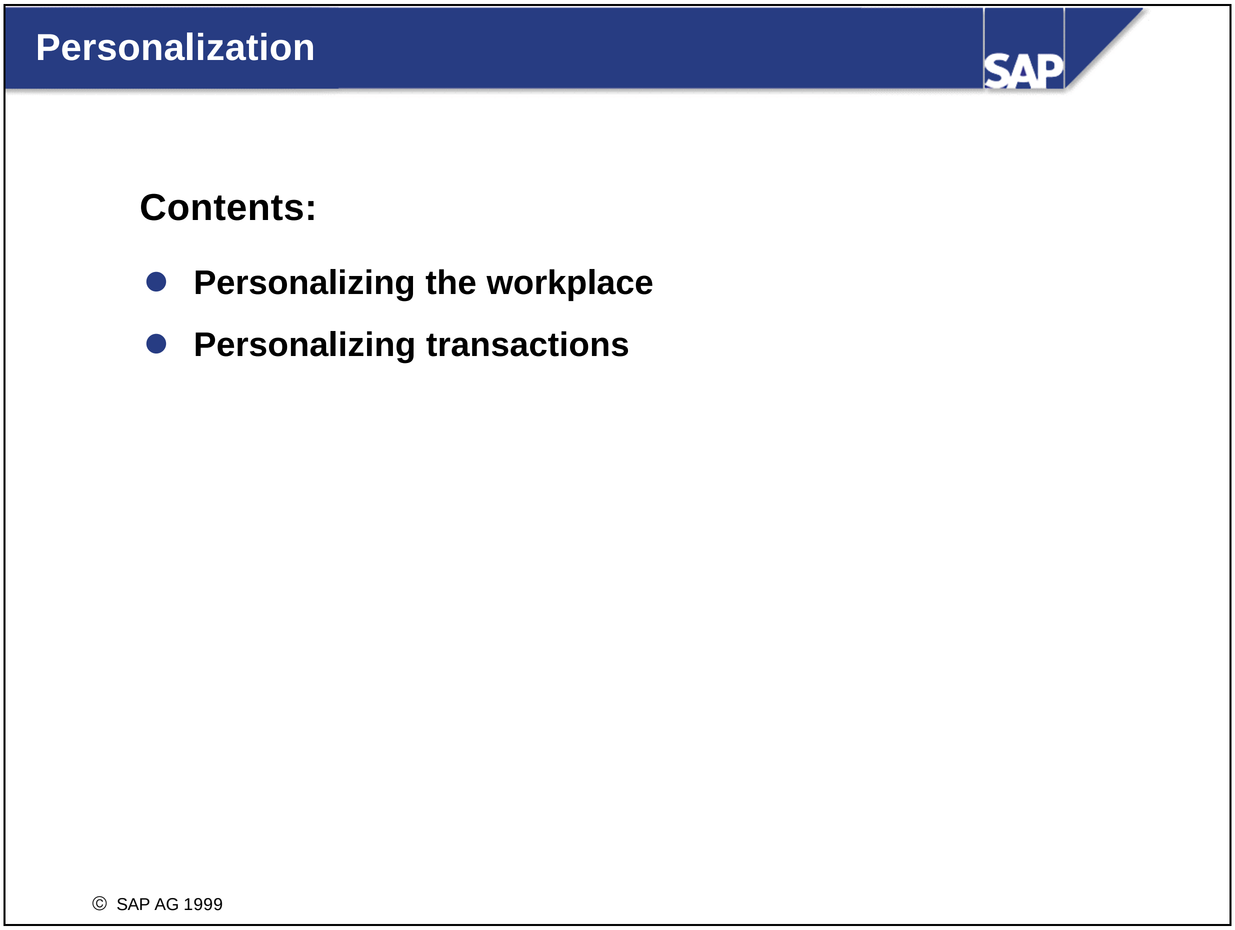
- Any change that you make your system to an object that has been delivered by SAP is known as a modification.
- Modifications can lead to complications at upgrade. When SAP delivers a new version of the object, you must decide whether the new object should be used, or whether you want to continue using your old object.
- Prior to Release 4.0B, modifications were only recorded at Repository object level (for example, an include program).
- Since Release 4.5A, the granularity for recording modifications has been finer. This has been made possible by the Modification Assistant, as we will see in the "Modifications" unit.
- The modification adjustment process has also been overhauled. How modifications are adjusted is also part of the "Modifications" unit.

Key to icons in the exercises and solutions

?	**Exercises**
✎	**Solutions**
☰	**Objectives**
❯❯	**Business scenarios**
💡	**Hints and tips**
⚡	**Warning or caution**

Data used in exercises

Type of data	**Data in training system**
Tables	SFLIGHT00 .. SFLIGHT18
Data elements	S_CARRID00 .. S_CARRID18
Programs	SAPBC425_EXIT_00 .. SAPBC425_EXIT_18
Transaction codes	BC425_00 .. BC425_18
Programs	SAPBC425_BOOKING_00 .. SAPBC425_BOOKING_00



Personalization

Contents:

- Personalizing the workplace
- Personalizing transactions

- The SAP System adjusts itself to the user's style of working: When the system is started, the users are only offered functions that are typical in their daily work. There is no unnecessary navigating through functions that are not used. In the past, user menus could be called in the Session Manager or in the dynamic menu in R/3. With Release 4.6A, the role-based menu is output in the form of a tree for each user.
- When you select a function, it is started in the same session. This function replaces the role-based menu. The role-based menu appears again automatically when you leave a transaction or when you start a new session.
- In the maintenance screen for activity groups (Transaction PFCG), the administrator can combine the menu structure for an activity group consisting of transactions, reports, and Internet/Intranet links to a user menu. You can choose any structure and description for the functions contained.
- The enterprise menu is no longer available with Release 4.6A.

- Typical questions at a work center are:
  - What function should be performed at this work center?
  - Which menus are needed?
  - What authorizations do the users need?
  - Which users are involved here?
- The goal of personalization is to answer these questions in the R/3 System.
- The tools provided by R/3 for this purpose are area menus and activity groups.
- We will now see how these tools can be used to adapt the work center to the user's needs as effectively as possible.

- Area menus were also included prior to this release. They can contain:
  - Transactions
  - References to other area menus
  - Executable programs (new)
  - Lists created by programs (new)
- From this release onwards, you can include programs in area menus that create lists directly.
- You can assign users an area menu as their start menu. These users no longer see the complete SAP menu when they log onto R/3, but only the menu items that they require. By integrating the report trees, users obtains a complete view of their work environment.
- Area menus can also be linked to activity groups.

- In contrast to previous releases, area menus are displayed in tree form starting with Release 4.6. This gives the user a clearer overview of the available options.
- The objects that can be included in the area menu are listed in the right part of the graphic.

- Use Transaction SE43 to create an area menu. You can call this transaction with the given path.
- Assign a name in the corresponding customer namespace and create the area menu.
- You can include the area menus in your list of favorites in the GUI for faster editing at a later time.

- You build area menus by creating entries in the tree structure. Position the cursor and choose the corresponding icon for insertion at the same level or one level down. In the popup window that now appears, choose a description and the corresponding transaction code.
- You can also insert reports (objects that create lists, such as ABAP programs, queries, and so on)
- You can no longer store lists in report trees as of Release 4.6A. Report trees have been integrated in the new area menus.
- With *List*--> *Save* --> *Report tree* you can store lists for the program. Since the lists are stored program-specifically, you can display them in the corresponding area menus.

- During an upgrade, existing area menus are automatically migrated to the new structure. You can make further entries in these new area menus.

- With Release 4.6, SAP has implemented user-oriented R/3 operations. When the R/3 application is started, a tree structure appears in the initial screen containing the entries the user needs for his daily work.
- These role-based menus go beyond the scope of the area menus. Only the menu structure can be defined for area menus. You can define them as you like for role-based menus. They also use the functions of the Profile Generator.
- By using specific role-based menus you can set the following individually:
  - Menu structure
  - Profiles
  - User assignments
- The term "activity group" is synonymous in R/3 with "role-based menu." You can edit activity groups using the Profile Generator.

- Before you create your own activity groups, you should evaluate the predefined workplace examples that SAP delivers in Release 4.6A. You can use these workplace examples just as they are delivered in the SAP System.
- Delivered activity groups should not be changed. You can combine several activity groups to form a composite activity group. which may also include activity groups delivered by SAP.

- To create an activity group, choose the appropriate button on the initial R/3 screen.
- Assign a name for the activity group in the customer namespace and press *Create*. The system displays the maintenance screen for activity groups.
- The activity group naming conventions are defined as follows:

• SAP*	delivered by SAP
• Rest	customer namespace

- There are several ways to build the menu for your activity group. You can copy sub-trees and menu entries from
  - the SAP menu
  - another activity group
  - an area menu
- You can also maintain single entries. These can be
  - a transaction code
  - a report in which a transaction code is automatically generated
  - a hyperlink (e.g. web address or a path on the local machine)
- You cannot maintain single entries if it is a composite activity group.

- The system determines the authorization objects used in the given transactions. The assignment of single authorization objects for a transaction using Transaction SU22 provides the basis for this determination.
- Transaction SU22 also specifies for the particular authorizations whether or not:
  - there must be a check
  - there are default values
- Using these default values makes maintaining authorizations much simpler. You only have to maintain authorizations marked with the yellow icon. If you do not do so, full authorization is automatically given.
- In the last step, a profile is generated from your entries. The system proposes a name T-<number>, which you can change here, but not later on. Enter a meaningful name.

- Next assign the relevant users to the activity group.
- Once you have assigned the users, you must adjust the user master profiles accordingly. The profile that was created is automatically assigned to the given users.
- A user can be assigned to more than one activity group. Each time you change an activity group, you must also adjust the user masters again.

- SAP delivers more than one hundred preconfigured activity groups. Choose the one most suitable for the particular work center and assign the users. Adjust the user master records.
- You can change activity groups delivered by SAP. However, these changes are lost during an upgrade. We therefore recommend that you copy the delivered activity groups and adjust the copy.

- In the last section we introduced the user-specific appearance of the interface, which is implemented using activity groups. In addition, there are ways to set single transactions to the needs of your enterprise or of individual user groups. In this section we will see how a transaction can be simplified without being modified.

- In this example you see two screens of an SAP transaction that should be redesigned using a transaction variant.
- Screen 100 is changed as follows: Fields are hidden; field attributes are changed; buttons are hidden.
- Screen 200 shows the following changes: buttons moved and screen inserted (with GuiXT). We will be discussing the use of GuiXT in more detail later.

- A transaction variant is a reference to a set of screen variants.
- You can create any number of screen variants for a screen. The transaction variant consists of these screen variants.

- You can create different kinds of transaction variants for an SAP transaction:
  - a standard variant
  - any number of "normal" transaction variants
- The standard variant is executed at runtime instead of the SAP delivered transaction. No new transaction code is required.
- A normal transaction variant will be called with its own transaction code of type "variant transaction".

- To create transaction variants, choose the component *Personalization* from the entry *AcceleratedSAP* in the SAP menu and then *Transaction variant*. You go to the transaction for maintaining transaction variants.
- Enter the name of the transaction from which you want to create a variant. The name of the variant must be unique in the system and be in the customer namespace.
- With the menu option *Goto*, choose whether you want to create a client-specific or a cross-client transaction variant.
- To create the variant, choose the appropriate button in the application toolbar.

- Pressing "Screen entries" starts the transaction in CALL mode.
- Triggering a dialog also triggers PAI of the current screen. The system sends another screen in which you can evaluate the fields of the screen.
- Also read the online documentation about transaction variants.
- The screen that was evaluated is stored as a screen variant when you continue. This will be discussed next.

- A screen variant is an independent Repository object, which has a unique name in the system. The name is constructed as follows:
    - Variant name
    - Client (only for client-specific transaction variants)
    - Screen number
- Here you specify whether or not field contents should be copied to the screen variant. You can set various attributes for the individual fields: You can undo or hide the input status of a field. You can find a detailed list of options in the online documentation about transaction variants.

- The GuiXT tool permits you to design the individual screens in a more flexible manner. GuiXT uses a script language to
  - Position objects on the screen,
  - Set attributes,
  - Include new objects.
- If you press "GuiXT", an editor window appears where you can enter the script. You can also choose GuiXT files stored on your local machine.
- You can also import scripts created on the local machine and export them there.

- You can change the layout of a screen with the script language used by GuiXT. You can
  - Move objects
  - Insert screens
  - Insert pushbuttons
  - Insert value helps
  - Change the input attributes of fields
  - Delete screen elements
- You are provided with a complete documentation of GuiXT with the installation. You can find more information on the homepage of the GuiXT vendor (http://www.synactive.com).

- You have the following options for starting a transaction variant:
  - Test environment
  - Transaction code of type "variant transaction"
  - User menu
- You can test the transaction flow in the test environment of the transaction variant maintenance routine. This is intended primarily for developers creating transaction variants.
- To insert a variant transaction in a user menu or activity group, you must create a transaction code of type "variant transaction".

- To start a transaction variant from a menu, you must create a transaction code of type "variant transaction". You can navigate there directly from the maintenance screen for the transaction variants. Alternatively you can start the corresponding transaction from the ABAP Workbench.

- You can insert the transaction in a menu by choosing one of the following two options: maintenance of
  - Activity group or
  - Area menu.
- The user can immediately see the changes made in this way.

Unit: Personalization

Topic: Creating a development class

For correct development you need a development class.

1-1 Create a development class.

1-1-1 The development class should be named ZBC425_##. (## = group number).

1-1-2 Assign the development class to a change request.

Exercises

Unit: Personalization

Topic: Create and enhance area menus, create user roles

At the conclusion of this exercise, you will be able to:

Create an area menu that you will use as initial screen for the rest of the training course. This area menu will contain all the entries you need for working efficiently during the course. This includes both the transactions specific to this course and all the transactions of the ABAP Workbench.

Only some R/3 functions will be actively used at a work center. The user should therefore only see a small selection of the transactions of the complete R/3 menu.

1-1 How do I create an area menu?

1-1-1 Which transaction / menu path can be used to create area menus? Include the transaction in your list of favorites.

1-1-2 Create an area menu called **ZBC425_##**. Adhere to the naming convention (## = group number).

1-2 Create a folder "Application programs" in the structure you created.

1-2-1 Create entries for the following topics that you will use during the training course:
Transaction **BC425_##**.

1-2-2 Create another folder named "Development" and include the following transactions: **SHD0**, **SPRO**, **PFCG**.

1-2-3 Insert a reference to area menu **S001** (ABAP Workbench) in your structure.

1-2-4 Enter the area menu you created as the start menu in your user fixed values. What does your start menu look like when you start a new session?

1-2-5 Check your results.

2-1 Create a user role.

2-1-1 Which transaction can be used to create a user role? How can you get there quickly?

2-1-2 Create a user role named **ZBC425_##** (## = group number).

2-1-3 Include the area menu you created in the user role.

2-1-4 Create a new folder. Insert the program **SAPBC425_BOOKING_##** here.

It is not the aim of this training course to fully explain the SAP authorization concept. In this exercise we will simply create a menu that can be used as a user-specific menu, without maintaining the profile.

2-1-5 Maintain the authorization data: Assign full authorization for the displayed sub-trees.

Assign full authorization by selecting the corresponding traffic light icon for the relevant sub-tree.

2-1-6 Insert authorization object **S_CARRID** in the authorizations manually. Assign the following authorization here:

Actions	All
Airline	Everything except for U*

2-1-7 Assign your user BC425-## this user role. Adjust the user master records.

2-1-8 Check your results. What options do you now have to start transactions?

The changes take effect immediately. Create a new session to see the changes in the initial menu. Check your user in the user maintenance screen (SU01).

Exercises

Unit: Personalization

Topic: Transaction variants

At the conclusion of this exercise, you will be able to:

- Significantly simplify use of a transaction with screen variants and transaction variants.

Your users complain that Transaction **BC425_TAVAR** is much too difficult to use (despite the Enjoy initiative). Actually you only have to fill in a few fields. A number of other fields, however, are superfluous. Help your users by simplifying use of the transaction.

1-1 Create a transaction variant for Transaction **BC425_TAVAR**.

1-1-1 How do you get to the maintenance screen for transaction variants? Include the corresponding transaction in your list of favorites.

1-1-2 Give the variant a name: **ZBC425##** (## = group number).

1-2 Go through the transaction screen by screen and create a screen variant for each of the screens. You should make the following changes:

1-2-1 Initial screen: Initialize the first two fields with "DE", "Frankfurt" and cancel the ready for input status.

1-2-2 Second screen: Set column "APT" of the table control to not visible. Deactivate menu function "BACK".

1-2-3 Third screen: Deactivate menu function "BACK".

1-3 Create a transaction code for the variant. Transaction name: **ZBC425##**.

1-4 Include the variant in the area menu you created.

1-5 Test your results.

Unit: Personalization

Topic: Creating area menus

1-1 You can create an area menu by choosing the following menu path in the SAP menu:

Tools ® ***ABAP Workbench*** ® ***Development*** ® ***Other tools*** ® ***Area menus***

1-1-1 Alternatively you can choose Transaction SE43

1-1-2 Choose the menu path ***System*** ® ***User profile*** ® ***Expand favorites*** to include the transaction in your list of favorites.

1-2 Create the folder using the corresponding pushbutton or menu entry.

1-2-1 Enter the transaction code in the right column: After you confirm your entry, the short text for the transaction is displayed. Complete the entries.

1-2-2 Create another folder as described above. Insert transactions **SHD0**, **SPRO**, **PFCG** in the list.

1-2-3 Position the cursor on the root node and choose *Insert*. Enter transaction code **S001** and set attribute "Reference". Complete the entry.

1-2-4 Choose the menu path ***System*** ® ***User profile*** ® ***Own data*** to define the area menu as start menu. You can no longer go to the SAP menu.

1-2-5 Create a new session. If you choose "SAP Menu", the menu you defined as start menu is displayed.

2-1 Creating a role (activity group).

2-1-1 Choose the corresponding pushbutton "Create menu" in the initial screen or the entry in the area menu you created or choose transaction code **PFCG.**

2-1-2 Create an activity group named **ZBC425_##** an (## = group number). Give it a short description and maintain the description of the activity group.

2-1-3 Include the area menu you created in the activity group.

2-1-4 Create a new folder. Insert the program **SAPBC425_BOOKING_##** by choosing the pushbutton "+Report".

2-1-5 Maintain the authorization data: Choose the appropriate tab title. Choose "Change authorization data". A list with a tree-like structure appears. The individual sub-trees have a yellow traffic light. Give full authorization for the displayed sub-trees by selecting the traffic light and confirming the next modal dialog box.

2-1-6 Insert authorization object **S_CARRID** by choosing pushbutton "+Manual". The object appears in an appropriate sub-tree which now has the attribute "manual". Expand the sub-tree and maintain the field values:

Field name or data class	**Values**
Action	*
Airline	*A to T** *V to Z**

Save the authorizations. Copy the profile name. Generate the profile.

2-1-7 Choose tab title "User". Enter your user **BC425-##**. Save your entry. Adjust the user master records by selecting the right pushbutton.

2-1-8 Create a new session. You can now toggle between the user menu and the SAP menu.

Solutions

Unit: Personalization

Topic: Transaction variants

1-1 Create a transaction variant for Transaction **`BC425_TAVAR`**:

1-1-1 You can go to the maintenance screen for transaction variants in different ways, for example with
SAP Menu → Tools → AcceleratedSAP → Personalizing → Transaction variants.
Transaction **`SHD0`** is started. Choose
System → User profile → Expand favorites to include the transaction in your list of favorites.

1-1-2 Enter the name of the transaction from which you want to create a variant in field "Transaction". Enter the name of the variant in field "Variant": **`ZBC425##`** (## = group number).

1-2 Execute the transaction screen by screen. Enter the corresponding values in the input fields. Leave the screen with the appropriate pushbutton and create a screen variant for each of the screens.

1-2-1 Initial screen: Assign the first two fields the values "DE" and "Frankfurt". Leave the screen by pressing the appropriate pushbutton. In the next popup window mark the checkbox "Copy field values" and the corresponding checkboxes for the screen objects. Give the screen variant a short description: Save the screen variant.

1-2-2 Second screen: Leave the screen by pressing the appropriate pushbutton. Mark "Copy values" again in the next dialog box. Mark column "FLH" of the table control as not visible. Choose the pushbutton for menu functions and deselect function code "BACK". Save the screen variant.

1-2-3 Third screen: Leave the screen with the "Save" function. Deactivate menu function "BACK" analogously to 1-2-2. Save the screen variant.

1-2-4 A list with a summary of all the screen variants that were created appears. You can now check your entries again. Save them to finally create the transaction variant.

1-3 To create a transaction code for the variant you can call the transaction code maintenance routine. Alternatively you can select the menu path ***Goto → Create transaction code*** from transaction SHD0. Give it the name **`ZBC425##`**.

1-4 Go to the area menu maintenance routine (Transaction SE43). Include the transaction variant in your area menu **ZBC425_##**. Proceed as described in the exercise on maintaining area menus.

Enhancements to Dictionary Elements

Contents:

- Table enhancements
- Field exits

- Tables and structures can be expanded in one of two different ways:
- Append structures allow you to enhance tables by adding fields to them that are not part of the standard. With append structures, customers can add their own fields to any table or structure they want.
- Append structures are created for use with a specific table. However, a table can have multiple append structures assigned to it.
- If it is known in advance that one of the tables or structures delivered by SAP needs to have customer-specific fields added to it, the SAP application developer includes these fields in the table using a Customizing include statement.
- The same Customizing include can be used in multiple tables or structures. This ensures consistency in these tables and structures whenever the include is extended.
- Nonexistent Customizing includes do not lead to errors.

- Append structures allow you to attach fields to a table without actually having to modify the table itself.
- Append structures may only be assigned to a single table. A table may, however, have several append structures attached to it. Whenever a table is activated, the system searches for all active append structures for that table and attaches them to the table. If an append structure is created or changed and then activated, the table it is assigned to is also activated, and all of the changes made to the append structure take effect in the table as well.
- You can use the fields in append structures in ABAP programs just as you would any other field in the table.
- Note: If you copy a table that has an append structure attached to it, the fields in the append structure become normal fields in the target table.

- You create append structures in the customer namespace. This protects them from being overwritten at upgrade or during release upgrade. New versions of standard tables are loaded during upgrades. The fields contained in active append structures are then appended to the new standard tables when these new standard tables are activated for the first time.
- From Release 3.0, the field sequence in the ABAP Dictionary can differ from the field sequence in the database. Therefore, no conversion of the database table is necessary when adding an append structure or inserting fields into an existing one. All necessary structure adjustment is taken care of automatically when you adjust the database catalog (**`ALTER TABLE`**). The table's definition is changed when it is activated in the ABAP Dictionary and the new field is appended to the database table.
- Pay attention to the following points when using append structures:
  - You cannot create append structures for pool and cluster tables.
  - If a table contains a long field (either of data type **LCH**R or **LRAW**), then it is not possible to expand the table with an append structure. This is because long fields of this kind must always be the last field in their respective tables. No fields from an append structure may be added after them.
  - If you use an append structure to expand an SAP table, the field names in your append structure must be in the customer namespace, that is, they must begin with either YY or ZZ. This prevents naming conflicts from occuring with any new fields that SAP may insert in the future.

- Some of the tables and structures delivered with the R/3 standard contain special include statements: Customizing includes. These are often inserted in those standard tables that need to have customer-specific fields added to them.
- In contrast to append structures, Customizing includes can be inserted into more than one table. This provides for data consistency throughout the tables and structures affected whenever the include is altered.
- Customizing include programs are part of the customer namespace: all of their names begin with 'CI_'. This naming convention guarantees that nonexistent Customizing includes do not lead to errors. No code for Customizing includes is delivered with the R/3 standard.
- You create Customizing includes using special Customizing transactions. Some are already part of SAP enhancements and can be created by using project management (see the unit on 'Enhancements using Customer Exits').
- The Customizing include field names must lie in the customer namespace just like field names in append structures. These names must all begin with either 'YY' or 'ZZ'.
- When adding the fields of a Customizing include to your database, adhere to same rules you would with append structures.

- Every time they define a data element, the SAP application programmers define keywords in different lengths and a short description for each data element.
- You create field exits in *Project management.* Field exits are processed when the user leaves a screen that contains a field which refers to a data element containing a field exit.

- SAP lets you create a field exit for every input-ready screen field that has been created with reference to the ABAP Dictionary. The additional program logic is stored in a function module and is executed at a specific point in the PAI logic.
- The slide shows the order in which processing takes place. Before the PAI logic of the screen is executed, the system performs the following checks: First the system checks if all the required fields have been filled in. If a required field is empty, the screen is shown again.
- The system then checks that data has been entered in the correct format.
- Any defined field exits are executed next. For example, by sending an error message you can have the screen sent again.
- Once all the field exits have been checked, the screen is processed as normal.
  - Field transport
  - Foreign key check
  - Processing the PAI module

- Field exits take you from a screen field with a data element reference to a function module. Field exits can be either global or local:
- **Global field exits** are not limited to a particular screen: If a global exit's data element is used on several screens, the system goes to the function module for all these screens after activating the field exit. Here you can, for example, edit the contents, force a new entry to be made by outputting an error message, or prohibit certain users from proceeding further.
- **Local field exits** are valid for one screen only. If you assign a screen from a specific program to a field exit, then the system will go to the appropriate function module from this screen once the exit has been activated.
- You can either create a global field exit or up to 36 local field exits for a data element, but not both.
- Each exit number refers to a different function module. Field exit function modules adhere to the following naming convention:

• Prefix:	FIELD_EXIT_
• Name:	<Data element>
• Suffix (for local field exit):	_0 to _9, _A to _Z

- To create field exits, choose *Utilities* in the ABAP Workbench. Choose *Enhancements* and then *Project management* to edit field exits and to implement customer exits. Do not create field exits directly from the Function Builder.
- Choose *Goto* -> *Global enhancements* -> *Field exits* to start the transaction for maintaining field exits. To create a new enhancement, use the menu *path Text Enhancements* -> *Create.*
- Enter the name of the data element to which your screen field refers in the modal dialog box. The Function Builder is started with a special naming convention and interface options. The system specifies the name of the field exit. Do not change this name. Create the function module in a customer function group.
- The function module must be assigned to an existing customer function group.
- The function module interface is fixed and cannot be changed. The function module has an import parameter `INPUT` and export parameter `OUTPUT`. The contents of the screen field are stored in parameter `INPUT`. The contents of `OUTPUT` are returned in the screen field when you leave the function module.

- Field exits are not transported automatically. Therefore, you must assign the value of INPUT to OUTPUT in your source code. Otherwise the screen field would be blank after executing the field exit.
- The following ABAP statements are not allowed in field exit function modules:
  - CALL SCREEN, CALL DIALOG, CALL TRANSACTION, SUBMIT
  - COMMIT WORK, ROLLBACK WORK
  - COMMUNICATION RECEIVE
  - EXIT FROM STEP-LOOP
  - MESSAGE I, MESSAGE W
  - STOP, REJECT
- When you debug a screen that is referenced by a field exit, the field exit code is ignored by the debugger. As with any normal function module, you can, however, debug the field exit code in the Function Builder's test environment.

- You can create local field exits that relate to a specific screen. A global field exit must already exist. Edit the local field exit based on the global field exit.
- You can create up to 36 local field exits, each of which carries a unique suffix. The system proposes a name for the function module; you should use this name.
- Defining local field exits means that the function module of the global field exits initially created is no longer used. However, you must not delete it, for technical reasons. The field exits in the system would be deleted if you deleted the global function module of the field exit from the list.

- You must activate the field exit as well as the function module. Also note that field exits are only taken into account during screen execution if the R/3 profile parameter **abap/fieldexit = YES** has been set for all application servers. (This profile parameter is set to 'NO' by default).
- If you declare field exits for multiple screen fields, you have **no control** over the order in which they are procesesd. Iin particular, you cannot access the contents of other screen fields in a field exit.
- Also Read Note 29377 about field exits.

Unit: Enhancements to Dictionary Objects

Topic: Table enhancements

At the conclusion of this exercise, you will be able to:

- Enhance tables with append structures.

You work as a computer specialist for a large travel agency. Your company uses R/3. One of the transactions in your R/3 System has been specially tailored to process air travel data. Your fellow employees use transaction **`BC425_##`** to display flight information when helping customers. They would like more information about a flight, for example the pilot's name or the main meal.

Your flight data is stored in table **`SFLIGHT##`**. You need to add two columns to this table without modifying it.

1-1 How can you add these two fields to table **`SFLIGHT##`** without modifying it?

1-1-1 How do you go about enhancing table **`SFLIGHT##`**?

1-1-2 Enhance table **`SFLIGHT##`** with a technique that does not require modifications.

1-2 Create an append structure for table **`SFLIGHT##`**.

1-2-1 Include two fields in the structure:
One should contain the **pilot's name** (character string of length 25)
and one should contain the **meal** (character string of length 20).

1-2-2 Define the types of the fields. Choose the type that strikes you as most suitable.

You will use these fields later on in a screen. Doing a little more work now will save you work later on.

Exercises

Unit: Enhancements to Dictionary Elements

Topic: Field exits

At the conclusion of this exercise, you will be able to:

- Implement a field exit that can be used to make supplementary checks of a screen field.

The transaction that your co-workers use to display flight information (**`BC425_##`**) allows you to access data for all airline carriers. The customer service personnel, however, should only be able to access the airlines for which it has explicit authorization.

1-1 What is the name of the program for the above transaction?

1-1-1 What is the name of the data element referenced by the input field for the airline?

1-1-2 Are the requirements met for linking a field exit to this screen field?

1-2 How can you create a field exit?

1-2-1 Create a field exit for the screen field found under 1-1. Reference the corresponding data element.

1-2-2 The Function Builder is started. Can you change the interface of the Function Builder? If you need a function group, create one named **`ZBC425_##`**.

1-2-3 What do you have to code in the source text? Program an authorization check. You can you perform an authorization check?

1-2-4 If the check is negative, send a message to this effect. You can create it yourself (message class **`ZBC425_##`**) or use message **`010`** in message class **`BC425`**.

1-2-5 Activate the function module and the field exit.

1-3 Check your results.

1-3-1 For which airline(s) do *you* not have authorization?

1-4 Create a local field exit for screen 0100 of Transaction **BC425_##** based on the global field exit.

Use a second session.

Unit: Enhancements to Dictionary Objects

Topic: Table enhancements

1-1 An append structure is the only way to enhance a transparent table (**`FLIGHT##`** is such a table) without modifying it.

1-1-1 How do you go about enhancing table **`SFLIGHT##`**? You can work with append structures just like with "normal" structure definitions. They are created from a table (or structure). Call the ABAP Dictionary (Transaction SE11 or the Object Navigator ***Single objects* → *Edit Dictionary objects***). Enter table name **`SFLIGHT##`** and choose *Display*.

1-1-2 Enhance table **`SFLIGHT##`** with the append technique. The detailed procedure is described below:

1-2 Create your append structure using either the menu option ***Goto* → *Append structures…*** or its corresponding pushbutton; accept the name that the system suggests. Give the append structure a short description and save it under the development class you created.

1-2-1 Include two fields in the structure: The field names must begin with YY or ZZ. For example **`YYPILOT`** and **`YYMEAL`**.
One should contain the **pilot's name** (character string of length 25)
and one should contain the **meal** (character string of length 20).

1-2-2 Create one data element each to define the field type. Ideally you should use forward navigation. Enter the name **`Z_PILNAME##`** and double-click on the field. Give the data element a short description and an adequate field label. Create a data element called **`Z_MEAL##`** for the meal. Don't forget to activate the data element.

1-3 Activate the append structure. If an error occurs, you can find details in the activation log.

Solutions

Unit: Modifications to Dictionary Elements

Topic: Field exits

1-1 The name of the program for transaction **BC425_##** is **SAPBC425_FLIGHT##**. You can get this information with the menu path ***System* → *Status***.

1-1-1 The name of the data element to which the input field for the airline refers is **S_CARRID##**. You get it by placing the cursor on the "Airline" field in the corresponding screen. Choose ***F1*** there and ***Technical info*** in the next dialog box.

1-1-2 To check that the requirements are satisfied, go to the Screen-Painter for screen 0100. You can see there that attribute "Dictionary" is set in the general attributes of the element list.

1-2 You can create a field exit for transaction **CMOD** (menu path in the ABAP Workbench: ***Utilities* → *Enhancements* → *Project management***). In the menu, choose ***Goto* → *Text enhancements* → *Field exits***. You are now in program RSMODPRF, which creates field exits.

1-2-1 In the menu, choose ***Field exit* → *Create***. In the next dialog box enter **S_CARRID** as data element name and choose "Continue".

1-2-2 The Function Builder is started. The name of the function module is already defined in the input field. If you choose "Create", the function module is created.

You have to assign the function module to a function group. Create function group **ZBC425_##** in a second session.

Can you change the interface of the Function Builder?

1-2-3 Use the statement pattern to add the source text for the AUTHORITY-CHECK. The source text should be as follows:

```
output = input.
AUTHORITY-CHECK OBJECT 'S_CARRID'
         ID 'CARRID' FIELD input
         ID 'ACTVT' FIELD '03'.
IF sy-subrc <> 0.
  MESSAGE e010(bc425).
```

1-3 Activate the function module. Go back. The list of field exits appears again. Activate the field exit with ***Field exit*** → ***Activate***.

Enhancements Using Customer Exits

Contents:

- Introduction
- Enhancement management
- Function module exits
- Menu exits
- Screen exits

- Application enhancements allow customers to enhance their application functions. Customer exits are preplanned by SAP and generally consist of several components.
- Application enhancements are inactive when delivered and can be completed and activated by customers as they are needed.
- Application enhancement characteristics:
- Each enhancement provides you with a set of preplanned, precisely defined functions.
- Each interface between SAP and customer functions is clearly defined.
- As a customer, you do not need in-depth knowledge of how to implement SAP applications.
- You do not need to adjust enhancements at upgrade because of new functions that SAP has developed.

- The SAP application programmer creates SAP enhancements from function module exits, menu exits and screen exits. A management function is provided for this purpose (transaction code SMOD).
- Customers are given a catalog containing an overview of existing SAP enhancements. They can then combine the SAP enhancements they want into an enhancement project using transaction CMOD.

- SAP enhancements are made up of component parts. These components include function module exits, menu exits, and screen exits. A specific component may be used only once in a single SAP enhancement (this guarantees the uniqueness of SAP enhancements).
- Customer enhancement projects consist of SAP enhancements. Each individual SAP enhancement may be used only once in a single customer enhancement program (this guarantees the uniqueness of a customer project).

- The SAP application programmer plans possible application enhancements in an application and defines the necessary components. These components are combined in SAP enhancements.
- The programmers document their enhancements as best they can, so that customers can implement the enhancements without having to analyze program source code or screen source code.

- First, create an enhancement project and then choose the SAP enhancements that you want to use.
- Next, edit your individual components using the project management function and document the entire enhancement project.
- Finally, activate the enhancement project. This activates all of the project's component parts.

- Transaction CMOD starts the project management function. You must give your enhancement project a name. SAP recommends that you think up a naming convention for all of your projects. You can, for example, include the project's transaction or module pool in its name. All enhancement project names must be unique.
- Next, go to the project's attributes and enter a short text describing the enhancenent project. The system inserts all of the project's other attributes (such as created by, created on, or status).

- Use the project management function to assign SAP enhancements to customer enhancement projects. Enter the names of the SAP enhancements you want to use on the appropriate screen.
- The search function gives you a catalog-like overview of existing SAP enhancements. From there you can select those enhancements that are of interest to you.

- Use the product management function to edit the components of your enhancement project.
- Depending on whether the component you are editing is a function module, a menu entry, or a subscreen, you branch to either the Function Builder, a dialog box for entering menu entries, or to the Screen Painter.

- Activation of an enhancement project affects all of its components. After it has been activated successfully, the project has the status *active*.
- During activation, all programs, screens, and menus containing components that belong to the project are regenerated (programs at the time they are executed). After activation, you can see the effect of the enhancements in your application functions.
- The *Deactivate* function allows you to reset an active enhancement project's status to *inactive*.

- When the enhancement project was created, you should have assigned it to a change request. Each of the component pieces (include programs, subscreens, menu exits, and so on) should be assigned to the same change request. Using the same change request allows you to transport the entire enhancement at the same time.

- Function module exits allow customers to implement additional logic in application functions. SAP application programmers define where function module exits are inserted and what kind of data they transfer. SAP programmers also create an exit's corresponding function modules complete with short text, interface, and documentation, as well as describing each function module exit's intended purpose in the SAP documentation.
- You write the source code for the function modules yourself. If need be, you can also create your own screens, text elements, and includes for the function group.
- The system processes your ABAP code when the enhancement project (of which your function module is a component) is activated. Function module exits have no effect prior to enhancement project activation.

- This graphic shows the flow of a program providing an enhancement in the form of a function module exit.
- The exit function module is called in the PAI logic of a screen at a position determined by the SAP application developer. Within the function module, the user can add functions in the customer namespace using an include.

- SAP application programmers use the ABAP statement **CALL CUSTOMER-FUNCTION 'nnn'** to call function modules, where **nnn** is a three-digit number. (where '**nnn**' is a three-digit number). The application programmer must also create the function module he wants to call and its related function group.
- These function modules belong to function groups whose names begin with X (X function groups).
- The following naming convention applies to these function modules:
  - Prefix: **EXIT**
  - Name: name of the program that calls the function module
  - Suffix: three-digit number
  - The three parts of the name are separated by two underscores.
- The **CALL CUSTOMER-FUNCTION** statement is only executed if the enhancement project has been activated. Multiple calls of the same function module are all activated at the same time.

- The most frequently asked question concerning enhancements is: how can you see if an application program offers a function module exit? There are a number of ways to find the answer to this question.

- To see quickly if an application program offers a function module exit, you can follow the path on the left-hand side of the graphic: (The menu path *System* → *Status* always displays the name of the current application program). In our example a suitable character string would be "CALL CUSTOMER". Use the *Find* icon and search globally in the program. If your search does not provide any results, you can define a larger search area. Determine the environment for the corresponding program and look for the specific character string in the program environment.

- The right side of the graphic shows you how to find the name of the required enhancement using search tools. You can restrict the search in the R/3 Repository Information System using different criteria: These are:
  - Development class (also try generic entries)
  - Technical name of the enhancement

- Use the project management (transaction: CMOD) function to edit function modules for function module exits.
- Use the button for editing components to go directly to the function module editor (display mode).
- DO NOT change the function module itself. It is especially important that you do not alter the interface in any way. The function module, however, contains an **`INCLUDE`** statement for an include program that you have to create in the customer namespace.
- Double-click on the include name beginning with ZX. This automatically takes you to the editor of the include program, where you can enter your code.

- To understand how an X function group works, you need to understand how a normal function group works:
  - A function group consists of includes. The system assigns unique names to the includes for different objects. Some of the include names are simply proposals and some cannot be changed.
  - Global data is stored in the TOP include. This include is generated automatically when a function group is created.
  - Function modules are stored in includes with sequential numbering, and they in turn are all stored in an include ending with UXX.
  - You can freely choose the names of the includes for all other objects (subroutines, modules, events, etc.). However, we advie you to accept the proposed names.

- Exit function groups created by SAP application programmers for enhancment exits contain include programs that begin with either 'LX' or 'ZX'. You can only edit includes beginning with a 'Z', since they are stored in the customer namespace.
- No further function modules may be added to the function group.
- The include program **`ZxaaaUnn`** contains the source code for the function modules of a function module exit.

- SAP application programmers can declare global data in include program LXaaaTAP.
- You can decla re your global data in include `ZXaaaTOP`.
- Include program LXaaaTOP also contains the `FUNCTION-POOL` statement, which may not be changed. Therefore, you must always include the message class in parentheses when outputting messages - for example, **`MESSAGE E500 (EU)`**.

- The **INCLUDE** statement for program ZXaaaUnn is in a **FUNCTION - ENDFUNCTION** block. Because of this, neither events, nor subroutines(**FORM**), nor modules (**MODULE**) are allowed here. They can, however, be created in separate includes, which is explained later. Data declarations made here with **DATA** are valid locally in this function module.
- The SAP application programmer can also make a proposal for the source text. In this case, an **INCLUDE LXaaFnn** is created (where **nn** is the internal number for the function module in the include **LXaaaUXX**). Documentation is also provided within the SAP enhancement. You can copy the source code from this include into your own customer include program ZXaaaUnn using the project management transaction.
- You can create your own text elements for the function group.

- SAP application programmers can supply you with default subroutines in include LXaaaF01.
- There could be further includes containing specific sub-objects.
  - LX...F01 contains subroutines delivered by SAP.
  - LX...E01 contains the events belonging to the X function group.
  - LX...O01 contains PBO modules for screens to be delivered.
  - LX...I01 contains the corresponding PAI modules.

- Subroutines, modules, and interactive events `(AT…)` are created as include programs and included enhancements using include program ZXaaaZZZ.
- Additional includes must adhere to the following naming convention:
  - ZXaaaFnn for subroutines,
  - ZXaaaOnn for PBO modules,
  - ZXaaaInn for PAI modules,
  - ZXaaaEnn for events.

- You can use **CALL SCREEN** to call your own screens. Create the related include programs for the PBO and PAI modules in include program **ZXaaaZZZ**.
- Use forward navigation (select an object and then double-click on it) to create your own screens and modules.
- Screens created in this manner are automatically given the name of the function module's main program (**SAPLXaaa**). The PBO modules for these screens can be found in include **ZXaaaO01**, the PAI modules in include **ZXaaaI01**.

- You can enhance SAP applications by adding your own processing logic at predefined points.
- Such enhancements can include your own screens with their corresponding processing logic and graphical user interface, as well as text elements created by customers.

- Menu exits allow you to attach your own functions to menu options in SAP menus. SAP application programmers reserve certain menu entries in your GUI interface for this. This allows you to define a text for the reserved menu entry and add your own logic, often in the form of a related function module exit. Once you activate menu exits, they become visible in the SAP menu. Whenever this menu option is chosen, the system processes either a function provided by SAP application programmers or your own function that you have implemented in a function module exit.

- In order for you to be able to implement menu exits, SAP application programmers must equip the GUI interface with function codes that begin with a plus sign ('+').
- These function codes are inactive at first and do not appear in the GUI until you have activated them. They do not appear on the screen.

- Menu exits are edited with the project management transaction (CMOD).
- The pushbutton for editing components calls a dialog box where you can enter short descriptions and choose a language for each additional menu entry.
- You may not make any changes to the GUI interface.

- SAP application programmers determine where a program reads additional function codes and how it reacts--- either with a function module exit or with a predefined function.

- You can implement menu exits based on reserved function codes. The SA P application programmer defines the relevant function codes, assigns them to menus, and often provides a function module exit.
- Menu exits and function module exits are both part of the same SAP enhancement.
- No pushbuttons may be assigned to additional function codes.
- You can, however, make changes to the various menu entries and activate their function codes.

- Screen exits allow you to make use of reserved sections of a main screen (subscreen areas). You can either display additional information in these areas or input data. You define the necessary input and output fields on a customer screen (subscreen).

- Subscreens are rectangular areas on your screen that are reserved for displaying additional screens at runtime. Each subscreen area can be filled with a different screen (of type subscreen) at runtime.

- The R/3 System determines which screen will be displayed in a subscreen area at PBO. The general syntax is as follows:
  `CALL SUBSCREEN <subscreen_area> INCLUDING <prg> <screen_no>.`
- For each subscreen, PAI and PBO events are processed just as if the subscreen were a normal screen.
- The sequence of "**`CALL SUBSCREEN`**" statements in your main screen's flow logic directly determines in what order the flow logic of individual subscreens is processed.
- Caution:
  - Function codes are only processed in the main screen's flow logic
  - You are not allowed enter a name for a subscreen's command field
  - You are not allowed to define GUI statuses for subscreens
  - No value for next screen may be entered in a subscreen's flow control

- The SAP application programmer can reserve multiple subscreen areas for a screen.
- The subscreen is called during flow control of the main screen with the `CALL CUSTOMER-SUBSCREEN` statement. The name of the subscreen area must be defined without apostrophes. The function group to which the subscreen belongs is defined statically in apostrophes, but the screen number can be kept variable by using fields; it must always have four places.
- Screen exit calls are inactive at first, and are skipped when a screen is processed.
- Only after a corresponding subscreen has been created in an enhancement project, and this project has been activated, will the system process the screen exit.
- You create subscreens in X function groups. Normally, these function groups also contain function module exits.

- Whenever the statement `CALL CUSTOMER-SUBSCREEN <area> INCLUDING <X-function-pool> <screen_number>` occurs at PBO in the flow control of a screen, a subscreen is included in the subscreen area defined by SAP application programmers. At this point, all modules called during the PBO event of the subscreen are also processed.
- The PAI event of a subscreen is processed when the calling screen calls the subscreen during its PAI event using the statement `CALL CUSTOMER-SUBSCREEN <area>`.

- The global data of the calling program is not known to the X function group that contains your subscreen; SAP application programmers use function module exits to explicitly provide this data to subscreens.
- In order to facilitate data transport, modules are called in the flow control of the calling program that contain function module exits for transferring data via interface parameters.
- Function modules belonging to these kinds of function module exits can be found in the same function groups as their corresponding subscreens.

- Data must be transported in the other direction as well, since global data from the X function group that contains your subscreen is not known to the calling program either. For this reason, SAP application programmers use function module exits to return any data to the calling program that was changed in the subscreen.
- This is done by calling a module during the main screen's PAI event that contains a function module exit for returning customer data via interface parameters.

- Subscreens are edited with the project management transaction (CMOD).
- The technical names of screen exits consist of the name of the calling program, a four-digit screen number, and the name of the subscreen area, followed by the name of the X function group's program and the number of the subscreen.

- You must create the subscreen as well as the corresponding PBO and PAI modules. The SAP development environment supports creation with forward navigation.
- Make sure that your subscreens are of screen type *subscreen* the first time you create them.
- You are not allowed to change any of the interfaces in the X function group that the subscreen and the function module exits belong to, nor are you allowed to add any of your own function modules.
- See also the restrictions listed on the slide entitled 'Calling Subscreens'.

- Screen exits allow you to determine the layout of certain portions of a screen yourself. You can use these areas to display additional information, or to collect and process data.
- Screen exits must be predefined (planned) by an SAP application programmer. Use the statement `CALL CUSTOMER-SUBSCREEN` to integrate these preplanned subscreen areas into the flow control of the calling screen at PBO and PAI events.
- As soon as you activate an enhancement project that contains a subscreen as a component, the calling screen is regenerated and the subscreen is displayed the next time the application function is called.

Unit: Customer exits

Topic: Function module exit

At the conclusion of this exercise, you will be able to:

- Implement an enhancement with a function module exit.

Your co-workers have asked you to alter Transaction **BC425_##** so that every time they try to display the details of a flight in the past, a warning message is displayed.

Adjust the program so that there is a warning when a flight in the past is selected. Try to avoid modifying the program.

1-1 Check if it is possible to enhance the transaction.

1-1-1 Did the SAP developer implement a customer exit for the given transaction that you can use to add the required functionality?

1-1-2 What is the name of the corresponding enhancement? Choose the enhancement that you can use to implement a supplementary check when you leave the first screen of the transaction.

1-2 Implement the enhancement.

1-2-1 Name the enhancement project **TG##CUS1.**

1-2-2 Program the following check:
Check that the date that was entered is prior to today's date (that is,. lies in the past). If this is the case, display a warning containing an appropriate text.

1-2-3 Create a suitable message in message class **ZBC425_##** or use message **011** in message class **BC425**.

1-2-5 Check your results.

Exercises

Unit: Customer exits

Topic: Menu exit

At the conclusion of this exercise, you will be able to:

- Implement an enhancement with a menu exit in combination with a function module exit.

Your co-workers are thrilled with the new functions that you have built into the system. The new warning messages in transaction **BC425_##** help them to avoid selecting flights from the past. However, they want more...

They want you to allow them to display a list of bookings for their current flight from within the flight display transaction. They already use a program that generates this kind of list, but up until now they have always had to call the program separately. It is called **SAPBC425_BOOKING_##**.

1-1 Examine transaction **BC425_##**. Are there any points in the transaction where you could call another program (or perhaps even a menu option that could allow you to call another program)?

1-1-1 Did the SAP developer implement a customer exit for the given transaction that you can use to add the required functionality?

1-1-2 What is the name of the corresponding enhancement? Choose the enhancement with which you can implement a menu enhancement.

1-2 Implement the enhancement.

1-2-1 Name the enhancement project **TG##CUS2.**

1-2-2 Edit the components of the enhancement. Start the specified program by choosing the supplementary menu entry. Return to Transaction **BC425_##** again when you leave the list.

1-2-3 Pass the relevant parameters to Program **SAPBC425_BOOKING_##**. Note the data provided in the function module exit.

1-3 Check your results.

Exercises

Unit: Customer exits

Topic: Screen exit

At the conclusion of this exercise, you will be able to:

- Display further fields in the screen of an SAP transaction and fill them.

"It would be really great if the details lists of Transaction **BC425_##** for displaying flights would also display further data…".

You accept this new challenge from your co-workers and try to solve the problem without having to modify the transaction. Specifically, you start looking for a way to add a couple of new fields to the second screen of this transaction (screen number 200).

1-1 What kind of possibilities are there to place additional fields on a screen? Take a closer look at screen 200 in Transaction **BC425_##** and see if this is possible.

1-1-1 Is there a screen exit for enhancing the screen?

1-1-2 If this is the case, what is the name of the corresponding enhancement?

1-2 Implement the enhancement for doing the following (project name: **TG##CUS3**):

1-2-1 Enhance the screen with three fields. The following should appear:

- Pilot's name
- Meal
- Number of seats still free on this flight.

1-2-2 Ensure that the data is correctly transported to the subscreen.

1-3 Check your results.

Consult the ABAP Help for the SUBMIT statement.

Unit: Customer exits

Topic: Function module exit

1-1 You can check if the transaction offers customer exits as follows:

1-1-1 ***System*** ® ***Status*** gives you the name of the corresponding program (**SAPBC425_FLIGHT##**)

1-1-2 You now have several ways to look for customer exits: You can either search for the character string **CALL CUSTOMER-FUNCTION** globally in the program or you can use the R/3 Repository Information System to search for enhancements containing the program name in the technical name of the component (restrict the search with ***SAPBC425_FLIGHT##*** in the component name).
The enhancement you were looking for has the name **SBC##E01**. The documentation for the enhancement shows that it is intended for supplementary checks of the first screen of the transaction.

1-2 Choose transaction **CMOD** to implement the enhancement.

1-2-1 You can go to transaction CMOD with the menu path ***Tools*** ® ***ABAP Workbench*** ® ***Utilities*** ® ***Enhancements*** ® ***Project management***. Create a project named **TG##CUS1** here and save it.

1-2-2 Include enhancement **SBC##E01** that you found in your project.

1-2-3 Edit the components. The source text of the exit function module appears in the Function Builder. Create the include by double-clicking. Your source text could be as follows:

```
IF flight-fldate < sy-datum.
  MESSAGE w011(bc425) WITH sy-datum.
ENDIF.
```

1-2-4 Activate your include. Go back to project management and activate your enhancement project.

Solutions

Unit: Customer exits

Topic: Menu exit

1-1 Examine the transaction as in the last exercise. It is advisable to search with the R/3 Repository Information System.

1-1-1 Ideally you can use the R/3 Repository Information System to search for a suitable enhancement containing the program name in the technical name of the component (restrict the search with ***SAPBC425_FLIGHT##*** in the component name).

1-1-2 The enhancement you were looking for has the name **SBC##E02**.

1-2 Choose transaction **CMOD** to implement the enhancement.

1-2-1 You can go to transaction CMOD with the menu path ***Tools*** ® ***ABAP Workbench*** ® ***Utilities*** ® ***Enhancements*** ® ***Project management***. Create a project named **TG##CUS2** here and save it.

1-2-2 Include enhancement **SBC##E02** that you found in your project. Edit the enhancement's components. Assign a menu text. Edit the function module exit by double-clicking. Create the customer include using forward navigation.

1-2-3 The source text of the include should be as follows for group:

```
SUBMIT sapbc425_booking_00
    WITH so_car = flight-carrid
    WITH so_con = flight-connid
    WITH so_fld = flight-fldate
   AND RETURN.
```

Activate the include program. Activate the enhancement project.

Solutions

Unit: Customer exits

Topic: Screen exits

1-1 Look at the transaction screens in the Screen Painter. You will see that screen 200 of transaction **BC425_##** offers a screen exit.

1-1-1 Examine the flow logic of the screens for character string **CALL CUSTOMER-SUBSCREEN**. You will see that screen 200 of transaction **BC425_##** offers a screen exit.

1-1-2 You can get the name of the enhancement for example by searching in the R/3 Repository Information System (see previous exercises). The name of the enhancement is **SBC##E03**.

1-2 Implement the enhancement in the same way as described in the previous exercises. Create a project called **TG##CUS3** in transaction CMOD. Include enhancement **SBC##E03** in your project. Edit the enhancement's components.

1-2-1 Use the screen exit to enhance the screen. You can create screen 0500 by double-clicking on the enhancement component. Make sure that you choose screen type "Subscreen". Copy the fields from the corresponding structure **SFLIGHT##** of the Dictionary. You have two options for placing a field on the screen for the free places:
You can declare a variable in the TOP include of the X function group, generate the program. You can then place this program field on the screen. Generate the screen. (or: You can enhance your append structure. You should not do this in the exercise since the trainer fills the fields of the append structure with a program. Enhancing the append structure could result in errors in this program.).

1-2-2 Use the function module exit for a correct data transport. Create the customer include and enter the following source text (example for group 00):

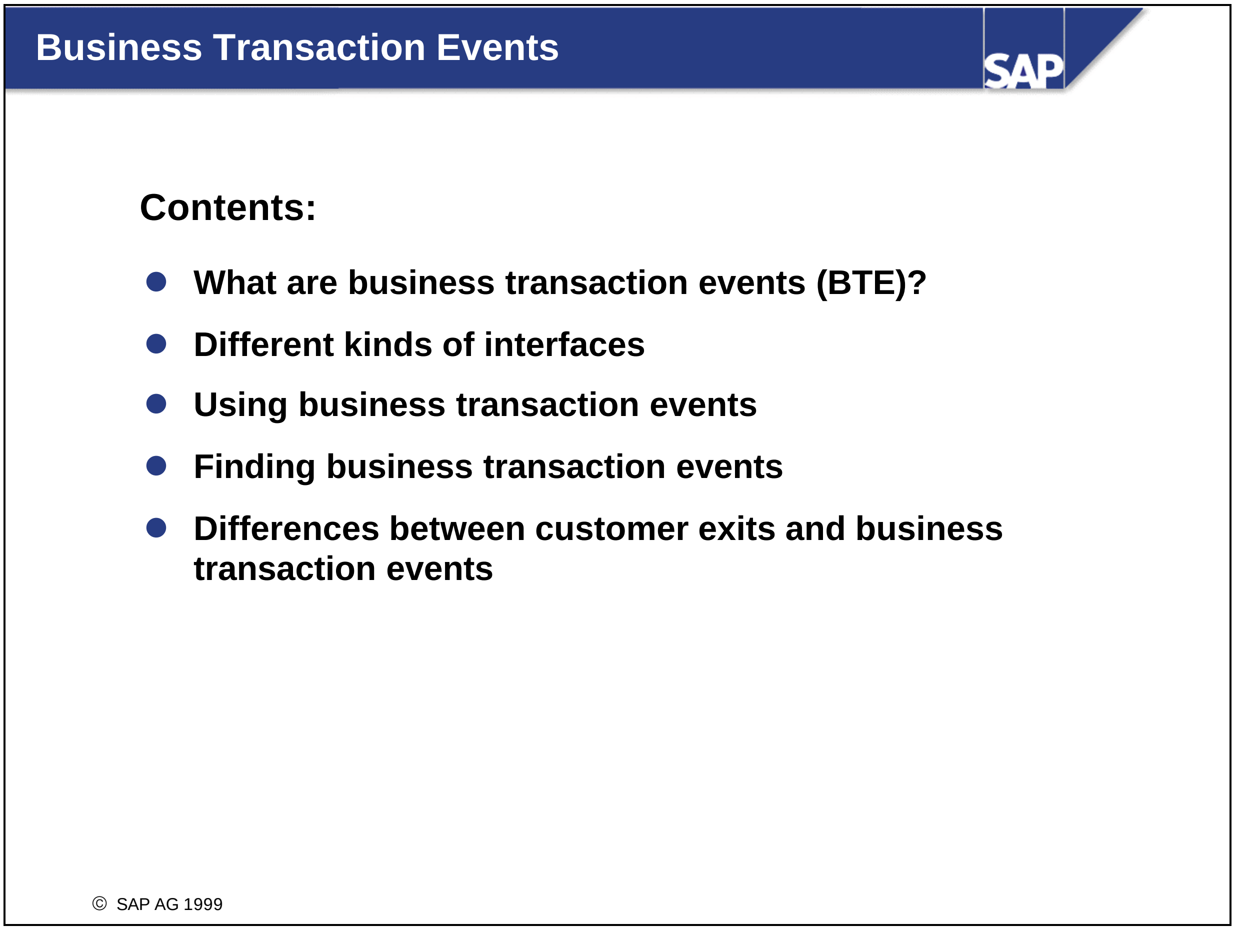
```
MOVE-CORRESPONDING flight TO sflight00.
seatsfree =
    flight-seatsmax – flight-seatsocc.
```

TOP include:

```
TABLES: sflight00.
DATA:   seatsfree type s_seatsocc.
```

Activate the programs. Activate the enhancement project.

1-3 Execute transaction BC425_## and check your results.



Business Transaction Events

Contents:

- What are business transaction events (BTE)?
- Different kinds of interfaces
- Using business transaction events
- Finding business transaction events
- Differences between customer exits and business transaction events

- Compared with earlier releases, the software delivery process now looks quite different. Previously, only two parties were involved in the delivery: SAP produced the software, and delivered it to its end-customers. Customers could enhance this standard using customer exits.
- However, now that the software is more component-oriented, more parties have become involved in the process: SAP provides the R/3 standard to each SAP Industry Solution, which uses it as a basis to add on its own encapsulated functions. The next link in the chain might be a partner firm, which builds its own Complementary Software Program (CSP) solution based on R/3. The last link in the chain is the customer, as before.
- All of the parties involved in this process are potential users and providers of enhancements. This requirement cannot be satisfied by customer exits, which can only be used once. Consequently, SAP developed a new enhancement technique in Release 4.0, which allows enhancements to be reused.

- Business Transaction Events (BTE) allow you to attach additional components, in the form of a function module, for example, to the R/3 system.
- Business Transaction Events use one of the following types of interfaces:
- **Publish & Subscribe interfaces**
  These interfaces inform external software that certain events have taken place in an SAP standard application and provide them with the data produced. The external software cannot return any data to the R/3 System.
- **Process interfaces**
  These interfaces are used to control a business process differently than the way in which it is handled in the standard R/3 System. They intervene in the standard process, and return data to the SAP application.
- You can attach various external developments to the R/3 System. You can create additional developments using the ABAP Workbench.

- The example above pertains to Publish & Subscribe interfaces. In this case, data only flows in one direction - from the SAP application to the additional component.
- SAP application developers make interfaces available to you at certain callup points in a transaction. You can deposit additional logic at these points.
- In a very basic scenario, SAP partners and customers can use the interfaces themselves. In this case business transaction events function in much the same manner as customer exits (see the unit on "Enhancements using Customer Exits").

- The above scenario also pertains solely to Publish & Subscribe interfaces.
- In contrast to customer exits, business transaction events allow you to use an interface for multiple types of additional logic.
- If this is the case, you must decide which bit of logic you want to execute at what time.
- Both of your enhancements exist side by side with out impeding each other; however, at runtime only one of the enhancements can be processed.

- Publish & Subscribe interfaces:
  - Allow you to start one or more (multiple) **additional operations** when a particular event is triggered. They do not influence the standard R/3 program in any way.
  - Multiple operations do not interfere with each other.
  - Add-on components can only import data.
  - Possible uses: Additional checks (authorizations, existing duplicates, and so on)
- Process interfaces:
  - In contrast to Publish & Subscribe interfaces, data exchange takes place in both directions with process interfaces. This influences the number of additions that can be attached to the interface.
  - When an event is triggered, a process in the standard program can only be replaced by **a single external process** using the process interface.
  - If you are using an add on from an SAP partner that uses a process interface, this enhancement is processed at runtime. If you choose to use this same process interface for one of your own developments, the partner enhancement is dismissed and your own enhancement is processed at runtime instead.

- The graphic shows the flow of an SAP program. The program contains an enhancement in the form of a Business Transaction Event. The program calls a service function module, which determines and processes the active implementation of the enhancement. The naming convention for these function modules is OPEN_FI_PERFORM_<n>_E (or OPEN_FI_PERFORM_<n>_P).
- This function module determines the active implementations for each enhancement and stores them in an internal table. The implementing function modules are processed in the sequence defined by the internal table. At this point the system also considers the conditions under which the function module will be processed in the customer namespace - for example, the country or application. These conditions are also shown as filter values.

- This graphic shows the syntax used to call a program enhancement using a business transaction event.
- In the SAP application program, a function module is called with the name "OPEN_FI_PERFORM_<no>_E" (or, for process interfaces, "OPEN_FI_PERFORM_<no>_P"). The application program passes data to the service function module using the interface. SAP developers have already designed the interface.
- The service function module searches for active implementations and places them in an internal table. They are then processed in a loop.

- Business transaction events allow you to implement additional logic in a task function, similar to function module exits. SAP application programmers determine where to place business transaction events in a task function and what data should be transferred at each point. They also create sample function modules complete with short texts, an interface, and documentation, and describe the functions possible with the enhancement in the accompanying SAP documentation.
- First, SAP application programmers assign a business transaction event an eight digit number by which it can be identified. These numbers should observe a particular convention. For example, the fifth and sixth digits should be identical with events in the same program.
- The SAP developer registers the event and creates a template function module, **`sample_interface_<n>`** , which establishes the interface for the user.

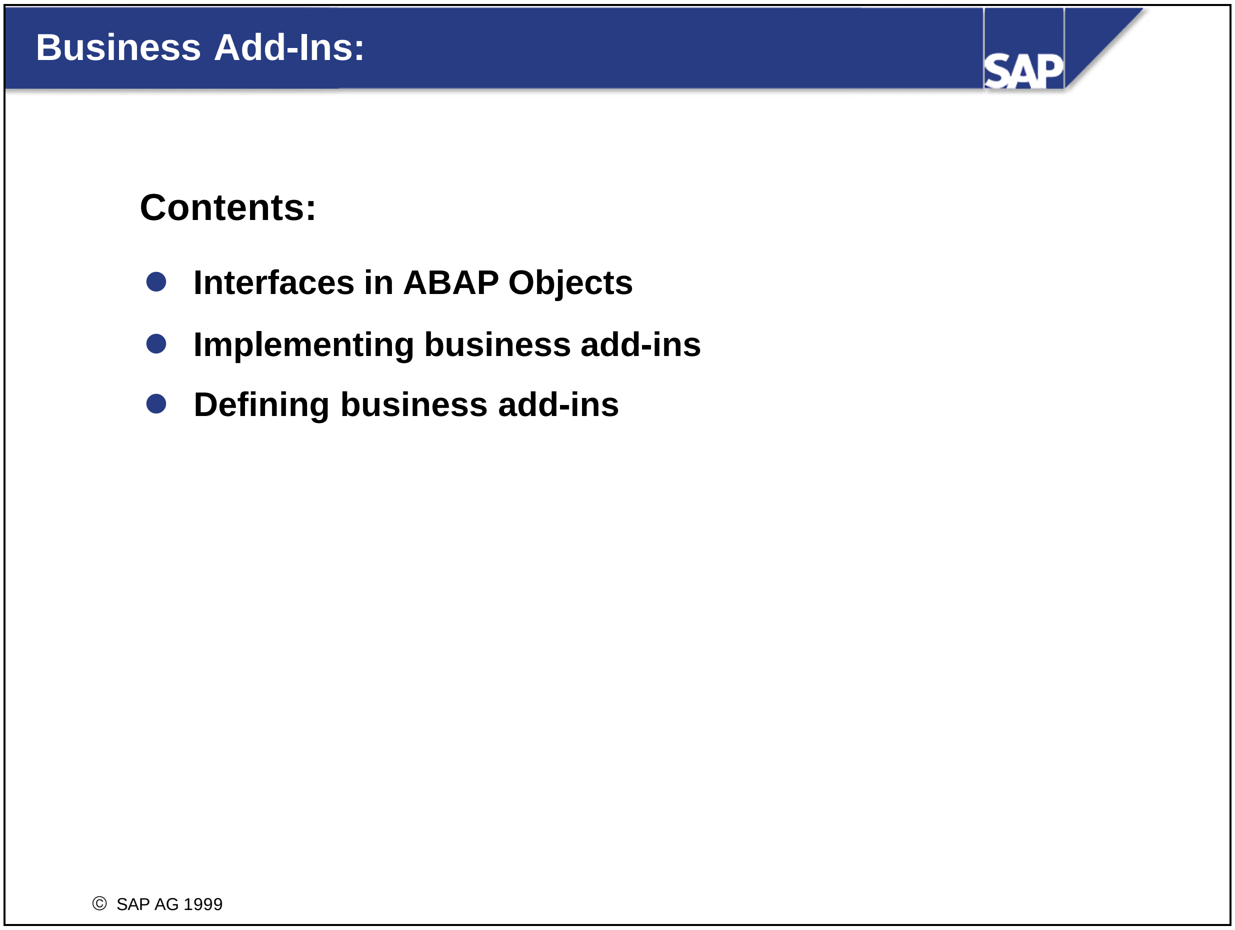
- To find out directly whether an application transaction offers business transaction events, you can use the procedure described on the left-hand side of the graphic. In the program source text, search for the character string "OPEN_FI_PERFORM". The number that completes the name of the function module is also the name of the event.
- In the SAP Customizing Implementation Guide (IMG), you will find the entry "Use business transaction events " under the "Financial Accounting Global Settings" node of the Financial Accounting area. Choosing this entry calls a transaction (FIBF) where you can execute all of the actions necessary for using Business Transaction Events.
- Under *Environment*, you will find search functions that you can use to identify appropriate business transaction events. You can view the documentation for the event from the list.

- The "Environment "menu of the service transaction FIBF contains two programs that you can use to search for BTEs. You can restrict the search by using various parameters.
- The BTEs that the system finds are displayed in a list. You can then:
  - Display the model function module (start the Function Builder and copy it, for example)
  - Display the interface
  - Display the documentation
- The documentation provides a clear explanation of how to use the enhancment and any restrictions that apply to it.

- Use service transaction FIBF to create a product. A product groups together a collection of enhancements.
- You can create products for various layers in the delivery chain. They define a sequence for processing the implementations of a business transaction event.
- You can only switch each product on or off as a whole entity. This allows the user to control which enhancements should be processed and which should not. It also ensures the integrity of the whole enhancement.

- You can use the transaction FIBF (called when you selected "Use business transaction events" from the financial accounting hierarchy) to carry out all necessary activities prior to using a business transaction event.
- First, you must choose an interface to attach your function module to. The *Interface* button displays the parameter structure for the interface you have selected. You can also use the documentation to determine what functions each interface allows you to perform.
- Use the ABAP Workbench to copy the sample function module **`sample_interface_<n>`** to the customer namespace (z_*) of a customer function group. You must not change the interface. You can fill the module with any source text except COMMIT WORK. Don't forget to activate the function module.
- Create a product in transaction FIBF.
- Assign a number to your function module and product.

- In contrast to customer exits, business transaction events are **client-specific**. This means that the same event can be used in different clients for different purposes.
- Business transaction events may also be used more than once.
- With Publish & Subscribe interfaces, you can choose which enhancement you want to use.
- With process interfaces, the system executes a single component in the hierarchical sequence SAP application, add on, customer.



Business Add-Ins:

Contents:

- Interfaces in ABAP Objects
- Implementing business add-ins
- Defining business add-ins

- A class is an abstract description of an object. Each object only exists while the program is running. In this unit, when we talk about objects, we may actually mean the abstract description (the class), depending on the context.
- An object is described by its class and consists of two layers - an inner and an outer layer.
    - Public components: The public components are those components of the class (for example, attributes and methods) that are visible externally. All users of the class can use the public components directly. The public components of an object form its **interface.**
    - Private components: These components are only visible within an object. Like the public components, the private components can be attributes and methods.
- The aim of object orientation is to ensure that a class can guarantee its own consistency. Consequently, the data of an object is normally "internal", that is, represented using private attributes. The internal (private) attributes of a class can only be changed by methods of the class. As a rule, the public components of a class are methods. The methods work with the data in the class and ensure that it is always consistent.
- Objects also have an identity to differentiate it from other objects with the same attributes and methods.

- Until Release 4.0, the nearest thing to objects were function groups and function modules.
- When you call a function module, an instance of its function group - with all of its data definitions - is loaded into the memory area of the internal session. An instance is a real **software object**. An ABAP program can therefore load instances of different function groups by calling function modules, but only one instance of each function group can exist at a time.
- The principle difference between real object orientation and function modules is that a program can work with instances of different function groups, but not with several instances of a single function group. For example, suppose a program wanted to manage several independent counters, or several orders at the same time. If we did this using a function group, we would have to program an instance management to differentiate between the instances (using numbers, for example).

- In practice, it is very cumbersome to implement instance management within a function group. Consequently, the data is usually in the calling program, and the function modules work with this data. This causes various problems. For example, all of the users have to work with the same data structures as the function group. If you want to change the internal data structure of a function group, you will affect a lot of users, and the implications of the changes are often hard to predict.
- Another problem is that all users have **copies** of the data, and it is difficult to keep them consistent when changes are made.
- Working with global data in function groups is dangerous, because it is almost impossible in a complex transaction to control when each function group is loaded.
- These problems have been solved with the introduction of **classes**. Data and functions are defined in **classes** instead of function groups. An ABAP program can then work with any number of runtime instances that are based on the **same** template. Instead of loading a single runtime instance of a function group implicitly when you call a function module, ABAP programs can create runtime instances of classes explicitly. The individual runtime instances are uniquely identifiable objects, and are addressed using object references.

- Interfaces are defined **independently** of classes.
- They can contain declarations for elements such as attributes and methods.
- Interfaces are implemented by classes
- The classes then have a uniform external point of contact. They must provide all of the **functions** of the interface by implementing its methods.
- In a program, you can create **reference variables** with reference to interfaces. However, you cannot instantiate an interface.
- Interface references can, however, point to **objects of different classes**.

- Business add-ins, unlike customer exits, take into account the changes to the software delivery process. The top part of the graphic illustrates the typical delivery process: It no longer merely consists of software provider and end user. Instead, it can now contain a whole chain of intermediate software providers like SAP Industry Solutions (IS) and partners.
- Below this is a diagram explaining how business add-ins work. Enhancements are made possible by SAP application programs. This requires at least one interface and an adapter class that implements it. The interface is implemented by the user.
- The main advantage of this concept is the capacity for reuse. Once implemented, a business add-in can be reimplemented by other links in the software chain (as shown on the right in the graphic).
- Furthermore, an implementation can also offer business add-ins of its own.

- A business add-in contains the components of an enhancement. Currently, each business add-in can contain the following components:
  - Program enhancements
  - Menu enhancements
- In future releases, the other components included in customer exits will also be available as add-in components.
- Several components are created when you define a business add-in:
  - Interface
  - Generated class (add-in adapter)
- The generated class performs the following tasks:
  - Filtering: If you implement a filter-dependent business add-in, the adapter class ensures that only the relevant imple mentations are called
  - Control: The adapter class calls the active implementations.

- This graphic shows the process flow of a program that contains a business add-in call. It enables us to see the possibilities and limitations inherent in business add-ins.
- Not displayed: You must declare a reference variable in the declaration part.
- In the first step, an existing service class, CL_EXITHANDLER, creates an object reference. We will discuss the precise syntax later on. This completes the preparations for using the program enhancement.
- When you define a business add-in, the system generates an adapter class, which implements the interface. In call (2), the interface method of the adapter class is called. The adapter class searches for all of the implementations of the Business Add-In and calls the implemented methods.

- This graphic contains the syntax with which you call a business add-in. The numbered circles correspond to the calls from the previous page.
- First, you must define a reference variable with reference to the business add-in interface. The name of the reference variable does not necessarily have to contain the name of the business add-in.
- In the first step (1), an object reference is created. This creates an instance of the generated adapter class, restricted to the methods of the interfaces ("narrowing cast").
- You can use this object reference to call the required methods (2).

- There are various ways of searching for business add-ins:
- You can search in a relevant application program for the string "CL_EXITHANDLER". If a business add-in is called from the program, the "GET_INSTANCE" method of this class must be called.
- You can then reach the definition of the business add-in using forward navigation. The definition also contains documentation and a guide for implementing the Business Add-In.
- You can also use search tools: Since SAP provided fewer than 50 Business Add-Ins in Release 4.6A, even a list of them all is still manageable.
- However, you can also use the application hierarchy to restrict the components in which you want to search. Start the Repository Information System, then choose Environment -> EXIT techniques -> Business Add-Ins" to start the relevant search program.
- Alternatively, you can use the relevant entries in the IMG.

- To implement business add-ins, use transaction SE19 (*Tools* -> *ABAP Workbench* -> *Utilities* -> *Business Add-Ins* ->*Implementation*).
- Enter a name for the implementation and choose *Create*. A dialog box appears. Enter the name of the business add-in. The maintenance screen for the business add-in then appears.
- Alternatively, you can use the Business Add-In definition transaction to reach its implementations. The menu contains an entry "Implementation", which you can use to get an overview of the existing implementations. You can also create new implementations from here.

- You can assign any name to the implementing class. However, it is a good idea to observe the proposed naming convention. The suggested name is constructed as follows:
  - Namespace prefix, Y, or Z
  - CL_ (for class)
  - IM_ (for implementation)
  - Name of the implementation
- To implement the method, double-click its name. The system starts the Class Builder editor.
- When you have finished, you must activate your objects.

- In the implementing class, you can create your own methods that you then call from the interface method.
- You cannot create them using forward navigation. Instead, you must define a regular method in the Class Builder (along with its interface). Specify a visibility for the method, and implement it.

- Use the "Activate" icon to activate the implementation of a Business Add-In. From now on, the methods of the implementation will be executed when the relevant calling program is executed.

- If you deactivate the implementation, the methods will no longer be called. However, the corresponding calls in the application program are still processed. The difference is that the instance of the adapter class will no longer find any active implementations. Unlike the "CALL CUSTOMER-FUNCTION" call, the "CALL METHOD CL_EXITHANDLER=>GET_INSTANCE" call is still executed even if there are no implementations. The same applies to the statement calling the method of the adapter class.

- You can only activate or deactivate an implementation in its original system. Changing it anywhere else constitutes a modification. The activation or deactivation must be transported into subsequent systems.

- If a business add-in can only have one implementation, there can still be more than one implementation in the same system. However, only one can be active at any time.

- As with customer exits, you can use menu enhancements with Business Add-Ins. However, the following conditions must be met:
  - The developer of the program you want to enhance must have planned for the enhancement.
  - The menu enhancement must be implemented in a BAdI implementation.

- Function codes of menu enhancements begin with a plus sign '+'.
- The menu entry will only appear if there is an active business add-in implementation containing the corresponding enhancement.

- You can only create function codes for business add-ins that can only be used once. Moreover, the business add-in cannot be filter-dependent.
- These restrictions are necessary to ensure that there are no conflicts between two or more implementations.

- If the user chooses the menu entry in the program to which the function code "+<exit>" is assigned, the system processes the relevant method call.
- The method call and the menu enhancement belong inseparably to one another. Having the former without the latter would make no sense. For this reason, it is important that the two enhancement components are combined in a single enhancement - the business add-in.

- To create a BAdI, use the BAdI Builder (*Tools* -> *ABAP Workbench* -> *Utilities* -> *Business Add-Ins* -> *Definition*).

- A business add-in has two important attributes that you must define:
  - Reusable
  - Filter-dependent
- If you want the business add-in to support multiple parallel implementations, select *Reusable*. The sequence in which the implementations will be processed is not defined. Even if the business add-in does not support multiple use, you can still have more than one implementation for it. However, only one implementation can be **active** at a time.
- If you make a business add-in filter-dependent, you can make calls to it depending on certain conditions. You must specify the filter type in the form of a data element. The value table of the domain used by the data element contains the valid values for the implementation.
- When the enhancement method is called, a filter value must be passed to the interface.

- You can include function codes in a Business Add-In definition (similarly to menu exits in customer exits). To do this, enter the program name and function code, and a short description in the relevant fields.
- Restrictions:
  - It is not currently possible to create BAdIs that consits **only** of menu enhancements (function codes).
  - If you use menu enhancements, you cannot reuse a BAdI or make it filter-dependent.

- The system proposes a name for the interface and the generated class. You can, in principle, change the name of the interface to anything you like. However, your BAdI will be easier to understand if you retain the proposed name.
- The name of the generated class is composed as follows:
  - Namespace prefix
  - CL_ (to signify a class in general)
  - EX_ (stands for "exit")
  - Name of Business Add-In
- If you double-click on the interface name, the system switches to the Class Builder, where you can define the interface methods.
- A BAdI interface can have several interface methods.

- You can use all of the normal functions of the Class Builder. For example, you can:
  - Define interface methods
  - Define interface parameters for the methods
  - Declare the attributes of the interface
- If the business add-in is filter-dependent, you must define an import parameter **flt_val** for each method. Otherwise, you define the interface parameters you need for the enhancement.

- Once you have finished working on your interface, you must activate it. This generates the adapter class for the Business Add-In.
- If you change the interface, the adapter class is automatically regenerated.
- You can also generate the adapter class explicitly at any time by choosing *Utilities* -> *Regenerate* from the initial screen of the Business Add-In maintenance transaction.

- To call a business add-in method in an application program, you must include three statements in the program:
- Declare a reference variable (1) with reference to the business add-in interface (in our example, "exit_ref").
- Call the static method GET_INSTANCE of the service class CL_EXITHANDLER (2). This returns an instance of the required object. This involves an implicit narrow cast, so that only the interface methods of the object with the reference variable "exit_ref" can be addressed.
- You can now call all of the methods of the business add-in. Make sure you specify the method interfaces correctly.

- If your Business Add-In is filter-specific, you must pass an appropriate value to the parameter **flt_val**.

- Business add-ins are a natural extension of the conventional enhancement technique. They have taken over the administration layer from customer exits, along with the availability of the various enhancement components.
- They adopted the idea of reusability from Business Transaction Events, and have been implemented using a consistent object-oriented approach.
- The object-oriented implementation provides previously unavailable opportunities. For example, it would be possible to enhance the object "Document". It would be possible to provide a new instance of the enhancement for each individual document.
- The components in parentheses in the graphic have not yet been implemented:
  - Screen enhancements
  - Table enhancements
- These enhancement components are planned for later releases. There will then also be a migration tool for converting previous enhancements into the new form.

Unit: Business Add-Ins

Topic: Using Business Add-Ins

At the conclusion of this exercise, you will be able to:

- Implement an enhancement with business add-ins

The customer service personnel of the agency wants the list of bookings that you implemented in the exercise on menu exits to contain more information. The list should contain the name of the customer in addition to his customer number.

1-1 Check if program **SAPBC425_BOOKING_##** (## = group number) can be enhanced.

1-1-1 Check the program for ways in which it can be enhanced.

1-1-2 Check if an enhancement option is suitable for outputting further information in the list.

1-2 Implement the enhancement you found. Name of the implementation: **ZBC425IM##**.

1-2-1 What data is passed to the interfaces of the methods? Are there already fields here that should be displayed in the list?

1-2-2 Table **SCUSTOM** contains the information about the customers. Get the customer's name from his customer number. Output the name.

1-3 Format the list.

1-3-2 How can you move the vertical line so that the additional fields are displayed within the frame?

1-3-2 Is the **CHANGE_VLINE** method suitable for changing the position of the vertical line? If so, use it.

1-4 Check your results.

Exercises

Unit: Business Add-Ins

Topic: Create Business Add-Ins

At the conclusion of this exercise, you will be able to:

- Create a business add-in and offer an enhancement in a program with business add-in technology

Develop your own supplementary components for the R/3 System. You want to offer an enhancement that can implement subsequent software layers in a program.

You deliver a program that outputs list of flight connections. You want to provide your customers with the following enhancement options using a Business Add In: Double-clicking on a line should implement further actions. Your customers should be able to built a details list.

Test your enhancement in the second part of the exercise: This details list should show all the flights for a connection.

1-1 Create a program that outputs list of flight connections.

1-1-1 To do so, copy program **SAPBC425_TEMPLATE** to the name **ZBC425_BADI_##**.

1-1-2 Assign your program to a development class and a change request.

1-2 Create a business add-in.

1-2-1 The name of the business add-in is **ZBC425##**.

1-2-2 Create a method. Define the interface.

1-2-3 Which parameter do you have to pass to the interface?

1-3 Edit the program so that a user can double-click on a line to output the details list.

1-3-1 Implement event **AT LINE-SELECTION**.

1-3-2 Insert the statements that are necessary for calling a business add-in in the program: Declare a reference variable; instantiate an object of the business add-in class; implement the call of the business add-in method at the right place in the program.

1-4 Implement the enhancement (name of the implementation: **ZB425##IM**).

1-4-1 A details list should be output when you double-click on a line of the list of the application program. The flight dates of the selected connection should be output in the details list. Table **SFLIGHT##** contains the flight dates.

1-4-2 Read the relevant data from table **SFLIGHT##** to an internal table with Array-Fetch. Then output selected fields of the internal table.

1-4-3 Which variables (attributes of the implementing class) do you have to declare? How do you declare an internal table? Where can you declare a table type?

1-5 Check your results.

Unit: Business Add-Ins

Topic: Using Business Add-Ins

1-1 Check if program **SAPBC425_BOOKING_##** (## = group number) can be enhanced as follows:

1-1-1 From the list display: Place the cursor in the list and choose *F1* → *Technical info.* Double-click on the program name (You can also start directly in the ABAP Editor.). Look for the character string **CL_EXITHANDLER** in the program. Double-click on the transfer parameter **exit_book**. Double-click on the interface used to define the type of exit_book. The Class Builder is started. Make a where-used list for the interface in classes. A class **CL_EX_BADI_BOOK##** is displayed. The name of the business add-in is thus **BADI_BOOK##**.

1-1-2 Start transaction **SE18** (business add-in definition). Read the documentation about business add-ins.

1-2 Implementing the enhancement From transaction **SE18** you go to the transaction for creating implementations of business add-ins with *Implementation* → *Create.* Name of the implementation: **ZBC425IM##**.

1-2-1 You can display the interface parameters by double-clicking on the method in transaction **SE18**. The transfer structure does not contain the fields that you want to display in the list. You have to read the corresponding data separately.

1-2-2 Double-click on the method name to go to the Editor. A proposal for implementing the methods is given below (group 00):

```
METHOD if_ex_badi_book00~output.
  DATA:
      name TYPE s_custname.
  SELECT SINGLE name
      FROM scustom
      INTO name
      WHERE id = i_booking-customid.
  WRITE: name.
ENDMETHOD.
```

1-3 The **change_vline** method is provided for formatting the list. You can move the right edge of the list here.

1-3-1 Parameter **c_pos** defines the position of the right vertical line.

1-3-2 The method can be implemented as follows:

```
METHOD if_ex_badi_book00~change_vline.
  c_pos = c_pos + 25.
ENDMETHOD.
```

Solutions

Unit: Business Add-Ins

Topic: Creating Business Add-Ins

1-1 Copy the template program as specified in the exercise.

1-2 To create business add-ins, start transaction SE18 (in the ABAP Workbench: *Utilities* → *Enhancements* → *Business add-ins* → *Definition*).

1-2-1 Choose **ZBC425##** as the name of the business add-in. Enter a short description and save your entries.

1-2-2 Choose the tab page "Interface". Double-click on the name of the interface. The Class Builder is started. Enter the name of a method. Give a short description. Choose "Parameters" to define the interface.

1-2-3 Define two importing parameters whose types are defined with **S_CARR_ID** (airline) and **S_CONN_ID** (connection number). Activate the interface. The adapter class is also generated.

1-3 Source text of the program with business add-in:

```
*&---------------------------------------------------------*
*& Report  SAPBC425_TEMPLATE*
*&---------------------------------------------------------*
REPORT  sapbc425_badi.

DATA:
    wa_spfli TYPE spfli,
    it_spfli TYPE TABLE OF spfli WITH KEY carrid connid.

* Reference Variable for BAdI
DATA:
    exit_ref TYPE REF TO zif_ex_bc42500.

* Selection Screen
SELECTION-SCREEN BEGIN OF BLOCK carrier
                              WITH FRAME TITLE text-car.
SELECT-OPTIONS: so_carr FOR wa_spfli-carrid.
SELECTION-SCREEN END   OF BLOCK carrier.
```

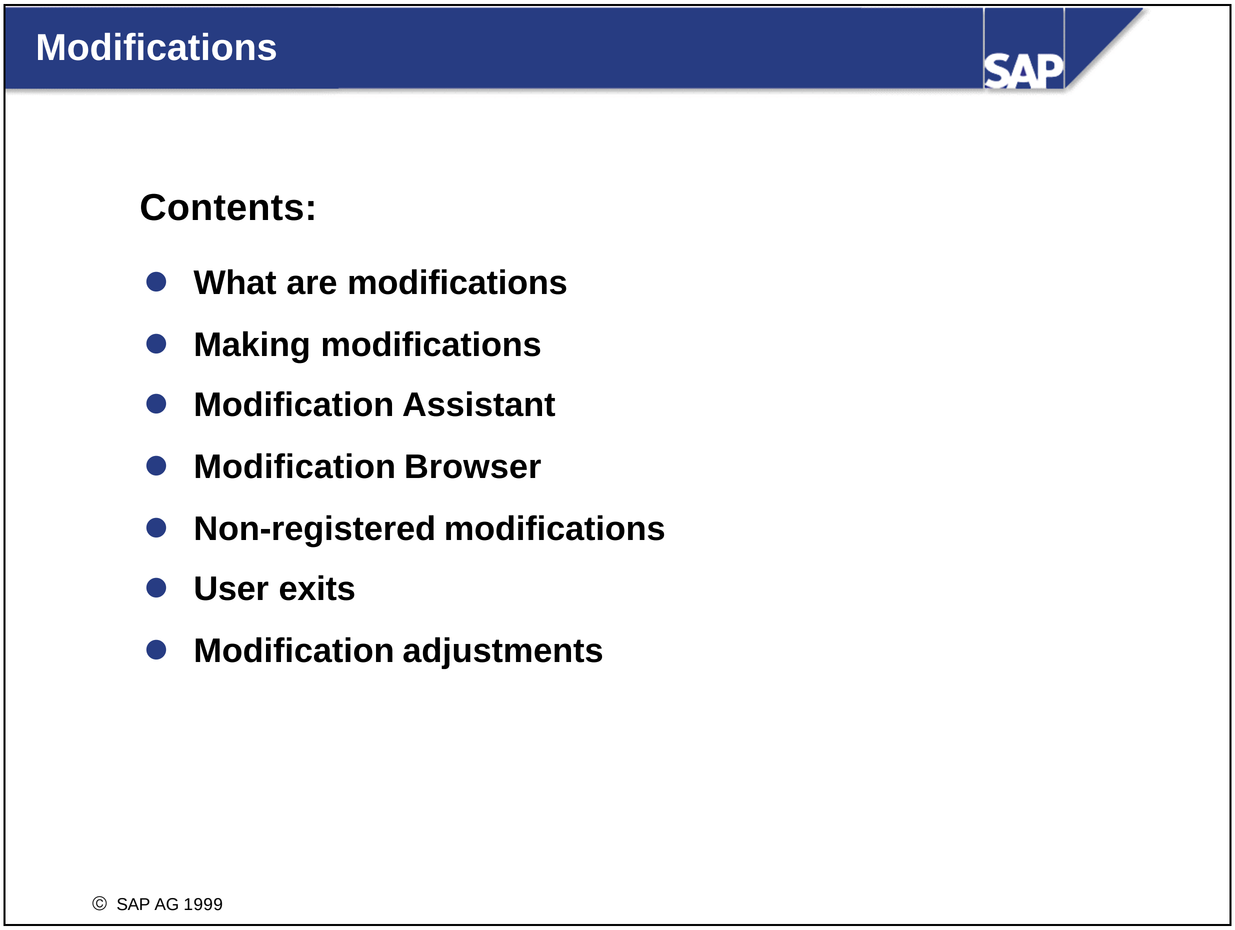
```
*&----------------------------------------------------------*
*&   Event START-OF-SELECTION
*&----------------------------------------------------------*
START-OF-SELECTION.

  CALL METHOD cl_exithandler=>get_instance
           CHANGING
               instance = exit_ref.

  SELECT *
      FROM spfli
      INTO CORRESPONDING FIELDS OF TABLE it_spfli
      WHERE carrid IN so_carr.
*&----------------------------------------------------------*
*&   Event END-OF-SELECTION
*&----------------------------------------------------------*
END-OF-SELECTION.

  LOOP AT it_spfli INTO wa_spfli.

    WRITE: / wa_spfli-carrid,
             wa_spfli-connid,
             wa_spfli-countryfr,
             wa_spfli-cityfrom,
             wa_spfli-countryto,
             wa_spfli-cityto,
             wa_spfli-deptime,
             wa_spfli-arrtime.

    HIDE:    wa_spfli-carrid,
             wa_spfli-connid.

  ENDLOOP.

  CLEAR wa_spfli.

*&----------------------------------------------------------*
*&   Event AT LINE-SELECTION.
```

```
AT LINE-SELECTION..

  CHECK NOT wa_spfli-carrid IS INITIAL.

  CALL METHOD exit_ref->lineselection
           EXPORTING
               i_carrid = wa_spfli-carrid
               i_connid = wa_spfli-connid.
clear wa-spfli.
```

1-4 Implement the business add-in. From transaction SE18 *choose Implementations → Create*. Give the implementation the name ZBC425##_IM. Choose the tab "Interface" and double-click on the name of the method. The Editor is started. Enter the source text here:

```
METHOD zif_ex_bc42500~lineselection.

  DATA:
      it_flights TYPE TABLE OF sflight00,
      wa_flights TYPE sflight00.

  FORMAT COLOR COL_HEADING.
  WRITE: / text-hea, i_carrid, i_connid.
  FORMAT COLOR COL_NORMAL.

  SELECT *
      FROM sflight00
      INTO CORRESPONDING FIELDS OF TABLE it_flights
      WHERE carrid = i_carrid AND
            connid = i_connid.

  LOOP AT it_flights INTO wa_flights.
    WRITE: / wa_flights-fldate,
             wa_flights-planetype,
             wa_flights-price CURRENCY wa_flights-currency,
             wa_flights-currency,
             wa_flights-seatsmax,
             wa_flights-seatsocc.
  ENDLOOP.
ENDMETHOD.
```

Activate the implementation.



Modifications

Contents:

- What are modifications
- Making modifications
- Modification Assistant
- Modification Browser
- Non-registered modifications
- User exits
- Modification adjustments

- An object is original in **only one system** In the case of objects delivered by SAP, the original system is at SAP itself. These objects are only copies in customer systems. This applies to your development system and all other systems that come after it.

- If you write your own applications, the objects that you create are original in your development system. You assign your developments to a change request, which has the type *Development/Correction*.

- This request ensures that the objects are transported from the development system into the subsequent systems.

- Changes to an original are called **corrections.** They are recorded in a change request whose tasks have the type "Development/correction".
- If, on the other hand, you change a copy (an object outside its own original system), the change is recorded in a task with the type "**Repair**". Repairs to SAP objects are called **modifications.**
- When you repair your own objects (for example, if something goes wrong in your production system), you can correct the original in your development system straight away. **When you change copies, you must correct the original immediately!**
- However, you cannot do this with SAP objects, because they are not original in any of your systems.
- You should only modify the SAP standard if the modifications you want to make are absolutely necessary for optimizing workflow in your company. Be aware that good background knowledge of application structure and flow are important prerequisites for deciding what kind of modifications to make and how these modifications should be designed.

- Whenever you upgrade your system, apply a support package, or import a transport request, conflicts can occur with modified objects.
- Conflicts occur when you have changed an SAP object and SAP has also delivered a new version of it. The new object delivered by SAP becomes an active object in the Repository of your system.
- If you want to save your changes, you must perform a **modification adjustment** for the objects. If you have a lot of modified SAP objects, your upgrade can be slowed down considerably.
- To ensure consistency between your development system and subsequent systems, you should only perform modification adjustments in your development system. The objects from the adjustment can then be transported into other systems.

- A registered developer must register registers changes to SAP objects. Exceptions to this registration are matchcodes, database indexes, buffer settings, customer objects, patches, and objects whose changes are based on automatic generation (for example , in Customizing). If the object is changed again at a later time, no new query is made for the registration key. Once an object is registered, the related key is stored locally and automatically copied for later changes, regardless of which registered developer is making the change. For the time being, these keys remain valid even after a release upgrade.
- How do you benefit from SSCR (SAP Software Change Registration)?
  - Quick error resolution and high availability of modified systems
    All objects that have been changed are logged by SAP. Based on this information, SAP's First Level Customer Service can quickly locate and fix problems. This increases the availability of your R/3 system.
  - Dependable operation
    Having to register your modifications helps prevent unintended modification. This in turn ensures that your R/3 software runs more reliably.
  - Simplification of upgrades
    Upgrades and release upgrades become considerably easier due to the smaller number of modifications.

- If you want to change an SAP Repository object, you must provide the Workbench Organizer with the following information:
    - SSCR key
    - Change Request
- We saw above how you get an SSCR key. If you now continue to change the object, you must confirm the following warning dialogs: At this point, you can still cancel the action without repairing the object.
- The Workbench Organizer asks you to enter a change request, as it would for your own objects. The object is automatically added to a repair task. The change request has the following functions:
    - Change lock
      After the task has been assigned, only its owner can change the object.
    - Import lock
      The object cannot be overwritten by an import (upgrade or support package).
    - Versions
      The system generates a new version of the object (see below).

- After development is finished, the programmer releases the task. At this point, the programmer must document the changes made. The objects and object locks valid in the task are transferred to the change request. If the developer confirms the repair, the import lock passes to the change request. If the developer does not confirm the repair when releasing the task, the import lock remains in place. Only the developer can release this lock.
- Once the project is completed, you release the change request. This removes all of the change request's object locks. This applies both to the change locks and the import locks.
- When the change request is released, the objects are copied from the R/3 database and stored in a directory at operating system level. They can then be imported into subsequent systems by the system adminstrator.
- After the modifications have been imported into the quality system, the developer must test them and check the import log of the request.

- When you release a change request, a complete version of all objects contained in the change request is written to the versions database.
- If you transport the Repository object again later, the current object becomes a complete copy and the differences between the old and the new object are stored in the versions database as a backwards delta.
- Whenever you assign a Repository object to a task, the system checks whether the current version agrees with the complete copy in the versions database. If not, a complete copy is created. This process is also initiated the first time you change an object, since SAP does not deliver versions of Repository objects.
- The versions of a Repository object provide the basis for modification adjustment. To support adjustment, information on whether the version was created by SAP or by the customer is also stored.

- Encapsulate customer source code in modularization units instead of inserting it directly into SAP source code (with, for example, customer function module calls in program source code, or customer subscreen calls for additional screen fields).

- When encapsulating the customer portions of a program, be sure to use narrow interfaces.
- You should define a standard for all of your company's modification documentation (see the following slides).
- You should also maintain a list of all modifications to your system (a modification log - see the following slides).
- All requests that contain repairs must be released before an upgrade so that all relevant customer versions can be written to the versions database (the system compares versions during adjustment) .
- Repairs must also be confirmed prior to upgrade, otherwise the object being repaired is locked and cannot be imported.

- Any modifications that you make to ABAP Dictionary objects that belong to Basis components are lost at upgrade--- these objects revert to their earlier form and **no adjustment help** is offered. This can lead to the contents of certain tables being lost.

- The aim of the Modification Assistant is to make modification adjustments easier. In the past, the granularity of modifications was only at include program level. Today, a finer granularity is available. Now, modifications can be recorded at subroutine or module level.
- This is because (among other reasons) the modifications are registered in a different layer. As well as providing finer granularity, this means that you can reset modificaitons, since the original version is not changed.

- If, in the past, you modified an include for which SAP provided a new version in an upgrade, a modification adjustment was necessary. The modification adjustment had to be performed line by line. The system provided little support.
- The Modification Assistant has changed this situation considerably. Modifications are now recorded with finer granularity. For example, if you modify a subroutine, the rest of the include remains unchanged. If SAP delivers a new version of the include, the system looks to see if there is also a new version of that subroutine. If this is not the case, your changes can be incorporated into the new version automatically.

- The original version of each software layer comprises the originals from the previous layer plus current modifications.

- Above is a list of the tools supported by the Modification Assistant.
- In the ABAP Editor, you can use modification mode to change source code. Only a restricted range of functions is available in this mode. You can add, replace, or comment out source code, all under the control of the Modification Assistant.
- Changes to layout and flow logic in the Screen Painter are also recorded.
- The Modification Assistant also records changes in the Menu Painter and to text elements, as well as the addition of new function modules to an existing function group.
- To avoid conflicts in the upgrade, table appends are also logged by the Modification Assistant.

- If you want to change an SAP object, you must provide the following information:
  - SSCR key
  - Change request
- The system informs you that the object is under the control of the Modification Assistant. Only restricted functions are available in the editor.
- You can switch the Modification Assistant on or off for the entire system changing the R/3 profile parameter **eu/controlled_modification**. SAP recommends that you always work with the Modification Assistant.
- You can switch off the Modification Assistant for single Repository Objects. Once you have done so, the system no longer uses the fine granularity of the Modification Assistant.

- In modification mode, you have access to a subset of the normal editor tools. You can access these using the appropriate pushbuttons. For example, in the ABAP Editor, you can:
  - Insert
    The system generates a framework of comment lines between which you can enter your source code.
  - Replace
    Position the cursor on a line and choose *Replace*. The corresponding line is commented out, and another line appears in which you can enter coding. If you want to replace several lines, mark them as a block first.
  - Delete
    Select a line or a block and choose *Delete*. The lines are commented out.
  - Undo modifications
    This undoes all of the modifications you have made to this object.
  - Display modification overview
    Choose this function to display an overview of all modifications belonging to this object.

- The graphic shows the result of changes made with Modification Assistant.
- The Modification Assistant automatically generates a framework of comment lines describing the action. The comment also contains the number of the change request to which the change is assigned, and a number used for internal administration.

- The "modification overview" icon provides you with an overview of the modifications you have made in the current program.
- The display is divided up according to the various modularization units. This corresponds to the structure used by the Modification Assistant to record the modifications.

- You can reset all of the modifications that you have made to the current object using the Modification Assistant by choosing this function. The record of the modifications is also deleted.
- Remember that you cannot selectively undo modifications to an object. You can only undo modifications based on the "all or nothing" principle.

- The Modification Browser provides an overview of all of the modified objects in the system. The Modification Browser differentiates between modifications made with the Modification Browser and those made without.
- On the initial screen of the Modification Browser, you can restrict the selection according to various criteria. This allows you to find modifications in a particular area.
- The Modification Assistant displays the hit list in tree form. Objects are arranged by:
  - Modification type (with/without the Assistant)
  - Object type (PROG, DOMA, DTEL, TABL, ...)

- SAP recommends that you use Modification Assistant to make changes to R/3 objects. Changes without the use of the Modification Assistant should be avoided. However, should this be necessary, you should document your modifications in the source code as follows:
  - **Preliminary corrections**
    SAP note, repair number, changed by, changed on, valid until
  - **Customer functions that have been inserted**
    subject area, repair number, changed by, changed on, INSERTION
  - **Customer functions that have replaced SAP functions**
    subject area, repair number, changed by, changed on, REPLACEMENT The SAP functions that you do not need should not be deleted, but commented out instead
- **Subject areas** are specified in the relevant process design blueprint (for example, subject area SD_001 = pricing).

- SAP recommends that you **keep a record of all modifications** that have been made to your system (that is, of any changes you have made to Repository objects in the SAP namespace).
- The following information should be logged for each modification:
  - Object type (program, screen, GUI status, ...)
  - Object name
  - Routine (if applicable)
  - Subject area (according to process design blueprint or technical design)
  - Repair number
  - Changed on
  - Changed by
  - Preliminary correction? (yes/no)
  - OSS note number, valid until Release x.y
  - Amount of time necessary to recreate modification during adjustment (measured in hours).

- A module pool is organized as a collection of include programs. This is particularly useful for making the program easier to understand. The organization is similar to that of function groups. In particular, the naming convention, by which the last three letters of the name of the include program identify its contents, is identical.
- The main program, as a rule, contains the include statements for all of the include programs that belong to the module pool.
- The includes described as "special" includes in the program are themselves only include programs - technically, they are not different. These programs are only delivered once.

- User exits are a type of system enhancement that were originally developed for the R/3 Sales and Distribution Module (SD). The original purpose of user exits was to allow the user to avoid modification adjustment.
- A user exit is considered a modification, since technically objects in the SAP namespace are being modified.
- The SAP developer creates a special include in a module pool. These includes contain one or more subroutines routines that satisfy the naming convention `userexit_<name>`. The calls for these subroutines have already been implemented in the R/3 program. Usually global variables are used.
- After delivering them, SAP never alters includes created in this manner; if new user exits must be delivered in a new release, they are placed in a new include program.

- User exits are actually empty subroutines that SAP developers provide for you. You can fill them with your own source code.
- The purpose behind this type of system is to keep all changes well away from program source code and store them in include programs instead. To this end, SAP developers create various includes that fulfill the naming conventions for programs and function groups. The last two letters in the name of the include refer to the include that the customer should use: "Z" is usually found here.
- Example: Program `SAPM45A`
  Include `M45AFZB`
- This naming convention guarantees that SAP developers will not touch this include in the future. For this reason, includes of this nature are not adjusted during modification upgrade.
- The subroutine call is already implemented in the programt. The interface is already defined. Normally, subroutines of this type only work with global data.
- If any new user exits are delivered by SAP with a new release, then they are bundled into new includes that adhere to the same naming convention.

- You can find a list of all user exits in the SAP Reference Implementation Guide.
- There, you will also find documentation explaining why SAP developers have created a particular user exit.
- Follow the steps described in the Implementation Guide.

- The set of objects for adjustment is derived from the set of new objects delivered by SAP in a new release. This is compared with the set of objects you have modified on your R/3 system..
- The intersection of these two sets is the set of objects that must be adjusted when you import an upgrade or support package.

- During modification adjustment, old and new versions of ABAP Repository objects are compared using transactions SPDD and SPAU.
- You **do not** have to call transaction SPDD to adjust Dictionary objects if:
  - No changes have been made to SAP standard objects in the Dictionary
  - You have only added customer objects to your system. Only SAP objects that have been changed must be adjusted using this transaction.
- All other ABAP Repository objects are adjusted using transaction SPAU. Upgrade program `R3up` tells you to start the transaction after upgrade has finished. You have 30 days to use transaction SPAU after an upgrade. After 30 days, you must apply for a SSCR key for each object that you want to adjust.
- Transaction SPAU first determines which objects have been modified. Then it determines which of these objects have a new version in the current upgrade. Modification adjustment allows you to transfer the modifications you have made in your system to your new R/3 Release.

- Use transaction SPDD to adjust the following ABAP Dictionary objects during the modification adjustment process:
- Domains
- Data elements
- Tables (structures, transparent tables, pool, and cluster table, together with their technical settings)
- These three object types are adjusted directly after the Dictionary object import (before the main import). At this point in time, no ABAP Dictionary objects have yet been generated. To ensure that no data is lost, it is important that any customer modifications to domains, data elements, or tables are undertaken prior their generation.
- Changes to other ABAP Dictionary objects, such as lock objects, matchcodes, or views, cannot result in loss of data. Therefore, these ABAP Dictionary objects are adjusted using transaction SPAU after both main import and object generation have been completed. You can use transaction SPAU to adjust the following object types:
    - ABAP programs, interfaces (menus), screns, matchcode objects, views, and lock objects.

- During modification adjustment, you should use two different change requests to implement the changes you have made: one for SPDD adjustments and another for SPAU adjustments. These change requests are then transported into other R/3 systems you want to adjust. This guarantees that all actual adjustment work takes place solely in your development system.
- When upgrading additional R/3 systems, all adjustments exported from the first system upgrade are displayed during the ADJUSTCHK phase. You decide which adjustments you want to accept into your additional systems and these are then integrated into the current upgrade. Afterwards, the system checks to see if all modifications in the current R/3 system are covered by the change requests created during the first system upgrade. If this is the case, no adjustments are made during the current upgrade.
- Note: For this process to be effective, it is important that all systems involved have identical system landscapes. This can be guaranteed by first making modifications in your development system and then transporting them to later systems before you upgrade the development system. You can also guarantee that all of your systems have an identical system landscape by creating your development system before upgrade as a copy of your production system and then refraining from modifying the production system again until after upgrade.

- Version compare is also used during or after an upgrade for modification adjustment.
- During modification adjustment, version compare determines the number of SAP objects that you a) changed in the system and that b) were then overwritten by SAP at upgrade.
- Version compare allows you to find where changes were made and transfer them to your new SAP version if you want.

- The icons in front of the individual objects that need adjustment show how they can be adjusted. The possible methods are:
    - Automatically
      The system could not find any conflicts. The changes can be adopted automatically
    - Semi-automatically
      The individual tools support you in adjusting the objects.
    - Manually
      You must process your modifications with no special support from the system. In this case, the modification adjustment does allow you to jump directly into the relevant tool.
- Adjusted objects are identified by a green tick.
- If you want to use the new SAP standard version, use *Restore original.* If you do this, you will have no further adjustment work in future.

Unit: Modifications

Topic: Making modifications

At the conclusion of this exercise, you will be able to:

- Implement modifications using the Modification Assistant.
- Implement non-registered modifications

In addition to the robust functions of the R/3 System, you also want to implement further functions.

Incorrect functions are very occasionally delivered. This requires inserting corrections before the corresponding support package can be applied.

The Modification Assistant does not allow some modifications. If they are implemented nevertheless, you can deactivate the Modification Assistant.

1-1 Modify R/3 objects. Use the Modification Assistant where possible. The objects to be changed are specified below:

1-2 Modify program **SAPBC425_BOOKING_##**.

1-2-1 Enhance the header so that the column with the customer's name also has a header.

1-2-2 Create a new variable for counting the data records. Output the counter in the last column of the list.

1-2-3 Also read fields **LUGGWEIGHT** and **WUNIT** of table **SBOOK** and output them in the list.

1-3 Modify program **SAPBC425_FLIGHT##**.

1-3-1 Change the layout of screen 0100: Insert a frame around the three input fields. Create a pushbutton and assign it function code **MORE**.

1-4 Modify data element **S_CARRID##**.

1-4-1 Change the field labels to:
short: "Airl"
medium: "Carrier".

1-4-2 Modify the documentation for this data element. Create a meaningful text.

1-5 Check your modifications in the Modification Browser.

Exercises

Unit: Modifications

Topic: Modification Adjustments

At the conclusion of this exercise, you will be able to:

- Adjust modifications

You must adjust the modifications made in the system after applying a support package or an upgrade.

1-1 Adjust the modifications you made to the objects you imported into the system.

Unit: Modifications

Topic: Making modifications

1-1 Modification of R/3 objects. The Assistant can usually be used if the modification icons exist.

1-1-1 <Detailed solution>

1-1-2

1-2 Modification of program **SAPBC425_BOOKING_##**.

1-2-1 You can change the header either directly from the list (***System* → *List* → *List header***) or in the Editor.

1-2-2 You can create a new variable directly in the R/3 program. Use the insert function of the Modification Assistant. Ideally you keep the changes locally in subroutine **data_output**. Also output the counter.
Alternatively you can implement this functionality in the enhancement, which would not cause a modification.

1-2-3 Read additional fields **LUGGWEIGHT** and **WUNIT** of table **SBOOK** and output them in the list.
Enhance the **SELECT** statement by these two fields. Output the fields in subroutine **data_output**.

1-3 Modification of program **SAPBC425_FLIGHT##**.

1-3-1 Use the Screen Painter to change the layout of screen 0100.

1-4 Modification of data element **S_CARRID##**.

1-4-1 Start the maintenance transaction for data elements. Place the cursor on the corresponding object and choose the modification icon. You can enter new text in the next dialog box.

1-4-2 Choose the "Documentation" pushbutton and enter new text.

1-5 To check the modification choose the Modification Browser (transaction **SE95**). Limit the selection with the user name or change request/task.

Solutions

Unit: Modifications

Topic: Modification Adjustments

1-1 Start the Patch Manager (transaction **SPAM**). Call transaction **SPAU** with the menu path ***Extras* → *Adjust modifications***. You can adjust the modifications here.

Epilog

Contents:

- Summary
- Evaluation of the different enhancement techniques

- Modifications can be categorized as **'critical'** if:

  They affect numerous other Repository objects (such as Dictionary objects or function modules)

  Modification adjustment is either difficult (as with menus, pushbuttons, and GUI interfaces up to 4.5A) or not supported by a tool (transaction codes, message classes, logical databases)

- Without the Modification Assistant (prior to Release 4.5A), both modifying GUI statuses and GUI titles, as well as assigning customer function modules to SAP function groups, should be considered **'critical'** activities.

- SAP only changes the following Repository objects in an **upwardly compatible** manner. They should therefore be considered **'uncritical'** by customers who want to call them:
  - Function modules that have been released
  - BAPIs
  - Includes for user exits
  - Screen, program, menu, and field exits
- After an upgrade, you must **test** customer reports that call SAP objects, as well as all objects displayed in the upgrade utility SPAU. This is also true for Repository objects that have been automatically adjusted using the Modifications Assistant (from Release 4.5A).
- You must be familiar with the processing logic of your application in order to be able to adjust programs properly.

- Modification adjustment is not necessary if you avoid making changes to SAP objects.
- Use program enhancements and appends with SAP tables to enhance SAP objects in such a way that your changes cannot be overwritten by SAP at upgrade.
- From Release 3.0, you can use Online Correction Services to import and cancel support packages and patches automatically (instead of having to insert preliminary corrections manually).

- Modification has the advantage that your live Repository objects do not lose their connection to the SAP standard. Copying, on the other hand, has the advantage that no modification adjustment will be necessary for your live Repository objects during subsequent upgrades.
- Choose copying instead of modifying if:
  - You have to make numerous changes to an SAP program
  - Your requirements will not be met by the standard in future R/3 releases
- During copying, pay attention to a Repository object's environment as well. You should only decide whether to modify or copy after having informed yourself of the consequences for the main program, as well as for all of the includes attached to the main program. The same holds true for function groups and function modules.

- ABAP development projects can be evaluated according to the following criteria:
  - What will implementation cost, measured in manpower (creating the concept, implementation, testing)?
  - How will the ABAP development project influence:
    - Production operation performance?
    - The amount of adjustment at upgrade?
- By calling SAP objects in your own Repository object, you can drastically reduce the amount of effort needed to implement your object. However, any changes that SAP makes to the Repository object you choose to call may make extra adjustment necessary after an upgrade. For example, SAP could conceivably change the user interface of a screen for which you have written a batch input program.

- Naming conventions allow you to avoid naming conflicts and give your Repository objects meaningful names (that can be understood by others).
- The following naming conflicts can occur:
  - **An SAP Repository object and a customer Repository object conflict**
    SAP Repository objects and customer Repository objects should be separated from each other by strict adherence to SAP naming conventions. OSS note 16466 gives you an overview of the current naming conventions for customer Repository objects (usually names that begin with either Y or Z).
  - **Two customer Repository objects conflict**
    Naming conflicts can also occur between customer Repository objects in decentralized development scenarios where more than one development system is being used. You can avoid naming conflicts in this area by reserving a special namespace for development areas within the customer namespace. The Workbench Organizer checks to make sure that you adhere to these conventions by making entries in view **V_TRESN**.
  - **Complementary software and customer Repository objects conflict**
    You can avoid naming conflicts when importing complementary software from SAP partners by reserving special namespaces in SAP OSS. In addition, **from Release 4.0** SAP partners can apply for prefixes in SAP OSS that they can tack on to the beginning of their Repository objects' names (For additional information, refer to OSS notes 84282 and 91032, or the white paper 'Development Namespaces in the R/3 System', order number E:50021723 [English] and D:50021751 [German]).