ABAP – Eine Einführung

Prof. Dr. Manfred Scheer
Technische Hochschule Mittelhessen
Fachbereich MNI
SS 2015

Prof. Dr. Manfred Scheer
Technische Hochschule Mittelhessen
Fachbereich MNI

ABAP – Eine Einführung

Allgemeines:

Diese Zusammenfassung ist in Verbindung mit Vorlesung seit dem SS 2003 entstanden. Die hier zusammengestellte Beschreibung der Programmiersprache ABAP mit ABAP-OO soll kein Handbuch oder ABAP-Buch ersetzen, sondern sie soll die Vorlesung und das Praktikum begleiten. Demzufolge ist weder der Sprachumfang vollständig aufgeführt noch ist die Beschreibung der Sprache in der Komplexität dargestellt, wie dies die ABAP Programmiersprache vorsieht. Ein Blick in die Online-Hilfe, die entsprechenden Handbücher oder weiterführende Literatur lohnt sich deshalb immer.

Um die Lesbarkeit der Sprachbeschreibung zu erhöhen gilt folgende Vereinbarung:

- ABAP-Programme werden in der Schrifttype **`Courier`** dargestellt. Schlüsselworte und Zusätze, die Teil der ABAP-Sprache sind, werden mit großen Buchstaben geschrieben.
- Vom Programmierer festzulegende Namen (Bezeichner) sind mit kleinen Buchstaben dargestellt. Dort wo es möglich ist, wird mit dem Namen auch ein inhaltlicher Bezug hergestellt, etwa ***`position`*** oder ***`laenge,`*** falls die Variable für eine Position oder Länge steht. Referenzvariable erhalten das Präfix ***ref_***.
- Für die Beschreibung gelten folgende Metazeichen:

`[ ]`	optionale Teile einer ABAP-Beschreibung.
`{ | }`	alternative Teile einer ABAP-Anweisung. Die Alternativen sind durch **`|`** getrennt.
`< >`	ABAP-Anweisungen oder Teile davon oder ein Anweisungsblock, die hier nicht näher beschrieben werden bzw. die vom Programmierer syntaktisch korrekt umgesetzt werden müssen.
`. . . .`	Wiederholung der Struktur (alternative oder optionale Teile).

Inhaltsverzeichnis

1 SAP-System

1.1 Diverse Einstellungen

Anpassung Layout: Das lokale Layout des SAP-Systems lasst sich mit Hilfe der Funktion  "***Anpassen des lokalen Layouts***" in der Symbolleiste verändern. U.a. lässt sich bei der Auswahl "***Optionen...***" das Ausgeben von Nachrichten einstellen (z.Bsp. Infofenster bei erfolgreicher Aktion deaktivieren usw.).

Technische Namen: Das Anzeigen der Kurzbezeichnungen der SAP-Transaktionen wird im Menü "***Zusätze – Einstellungen***" aktiviert.

Object Navigator: Einstellungen des Object Navigators werden im Menü "***Hilfsmittel – Einstellungen***" innerhalb des Object Navigators vorgenommen. U.a. kann hier der ***ABAP-Editor*** mit ***Pretty-Printer*** benutzerspezifisch eingestellt oder der grafische ***Screen Painter*** aktiviert werden.

1.2 SAP-Transaktionen

Direkter Aufruf: Im Eingabefeld der Symbolleiste können SAP-Transaktionen direkt aufgerufen werden. Mit ***/n<techn. Name>*** wird die bestehende Aktionen abgeschlossen und die neue Transaktion gestartet. Mit ***/o<techn. Name>*** wird die SAP-Transaktion in einem neuen Modus gestartet (neues Fenster).

Beispiel: **/nSE80** ruft den Object Navigator auf, die bestehende SAP-Transaktion wird beendet. Im SAP-System sind über 16000 Transaktionen hinterlegt.

Wichtige Transaktionen :

techn. Name	Beschreibung
SE80	Object Navigator (Entwicklungsumgebung)
SE11	ABAP Dictionary
SE16	Data Browser (Anzeigen von Tabelleninhalten)
SE24	Class Builder
SE36	logische Datenbank
SE37	Function Builder (Funktionsbausteine)
SE35	Dialogbausteine
SE38	ABAP Editor
SE63	Übersetzung
SE84	Infosystem
SE91	Nachrichten (Meldungen)
SE93	Transaktionen
SE09	Transportorganisator
SE30	ABAP Objects Laufzeitanalyse
SM04	angemeldete Benutzer
SM50	Prozessübersicht
SMX	anzeigen der eigenen Jobs
SE61	Dokumentation R/3
ABAPDOKU	ABAP-Dokumentation mit Beispielen
ABAPHELP	ABAP-Hilfe

2 Grundlagen der ABAP-Sprache

2.1 Kommentare, Bezeichner, Schlüsselworte und Klammern

Kommentare:

```
* Dies ist ein Kommentar. Der * muss in der ersten
* Spalte stehen.
```

oder

```
" Zeichen, die nach einem " stehen werden als Kommentar
" interpretiert.
```

Namenskonvention: Erlaubt sind alle Buchstaben (ohne Umlaute), Ziffern und der Unterstrich _.Der Name muss mit einem Buchstaben beginnen, den Bindestrich sollte man in einem Namen nicht verwenden, da er für den Zugriff auf Strukturelemente benötigt wird. Nicht erlaubt sind vordefinierte Wörter (wie **space**, **sy-uname** etc.), ABAP-Schlüsselwörter und deren Zusätze.

Schlüsselworte: Schlüsselwörter der ABAP-Programmiersprache sind reservierte Wörter mit einer festgelegten Semantik. Sie dürfen nicht als Bezeichner verwendet werden.

Klammern: Ausdrücke können in runde Klammern **()** eingeschlossen werden. Runde Klammernpaare erzwingen eine vorrangige Auswertung innerhalb von Ausdrücken. Klammern werden als ABAP-Worte aufgefasst. Dies hat als Konsequenz, dass vor und hinter einer Klammer eine Leerstelle eingefügt werden muss.

2.2 Datentypen, Typ- und Variablendeklaration

Elementare Datentypen :

Typ	Beschreibung	Byte	Initalwert
i	Integer, ganze Zahl	4	0
f	Float, Gleitpunktzahl	8	0
n	numerischer Text	1 (1-65535)	'0....0'
p	Packed, gepackte Zahl	8 (1-16)	0
c	Character, alphanum. Text	1	' '
d	Date, Datumsangabe	8	'0....0'
t	Time, Zeitangabe	6	'0....0'
x	Hexadecimal	1 (1-65535)	'0....0'
string	beliebige Zeichenfolgen		
xstring	beliebige Bytefolgen		

Typdeklaration:

```
TYPES typ_name[(laenge)]
      {TYPE typ | LIKE name} [DECIMALS dezimal]
```

Datendeklaration:

```
* Typen i, f, d, t, string, xstring
DATA name {TYPE typ | LIKE name} [VALUE wert].

* generische Typen c, n, p, x
DATA name[(laenge)] {TYPE typ | LIKE name}
      [DECIMALS dezimal] [VALUE wert] .
```

Konstante:

```
CONSTANTS name[(len)] {TYPE typ | LIKE name}
     [DECIMALS dezimal] VALUE wert .
```

Literale:

```
'Textliteral', 123456, -4711, '12.34', '-56.89'
'12.3456-04', '+47.12E+4', '1E160'
```

Struktur-Typen:

```
TYPES typ_name {TYPE typ | LIKE name} .
```

typ_name kann sich auf eine beliebige bestehende Typ- oder Datendefinition beziehen (elementarer Typ, Struktur-, Tabellen, Referenztyp, DD-Typ).

```
{TYPES:|DATA:}
    BEGIN OF struktur,
        komp {TYPE typ | LIKE name}  ,
       [komp {TYPE typ | LIKE name}  ,]...
    END OF struktur .
```

Attribute von Datenobjekten bestimmen:

```
DESCRIBE FIELD datenobjekt [LENGTH laenge] [TYPE typ]
   [OUTPUT-LENGTH ausgabe] [<weitere Attribute>].
```

Ermittlung von Datenobjektattributen zur Laufzeit (siehe dazu auch Online-Hilfe).

interne Tabellen:

```
{TYPES|DATA} itab {TYPE | LIKE} <tabkind>
       OF {linetype / lineobject}
      [WITH {UNIQUE|NON-UNIQUE} <keys>]
      [INITIAL SIZE n] .
```

mit:

```
<tabkind>: {[STANDARD] TABLE|SORTED TABLE|HASHED TABLE}
```

```
<keys>: {KEY col1 .. coln |KEY table_line|DEFAULT KEY }
```

Attribute interner Tabellen bestimmen:

```
DESCRIBE TABLE itab [LINES anzahl] [OCCURS initial]
      [KIND tabellen_art] .
```

Mit **LINES** wird die Anzahl der gefüllten Zeilen, mit **OCCURS** die initiale Größe der internen Tabelle ermittelt. **KIND** ermittelt die Tabellenart ('T' für Standard-Tabellen, 'S' für sortierte und 'H' Hash-Tabellen). Um die Anzahl der gefüllten Zeilen zu bestimmen gibt es die Funktion **LINES(** *itab* **)** .

Feldsymbole:

```
FIELD-SYMBOLS <name> [<gen. oder vollständiger Typ>] .
```

Deklariert ein Feldsymbol mit dem Namen ***<name>***. Die spitzen Klammern sind Teil der Syntax, sie identifizieren Feldsymbole im Programmtext. Ohne Typangabe übernimmt das Feldsymbol alle technischen Eigenschaften des zugewiesenen Felds, mit Typangabe erfolgt bei der Zuweisung eine Typprüfung. Mit Feldsymbolen wird immer das Datenobjekt selbst angesprochen, auf das das Feldsymbol verweist (vergleichbar mit dereferenzierten Zeigern).

Zuweisung an ein Feldsymbol (statisches ASSIGN):

```
ASSIGN datenobjekt TO <feldsymbol> .
```

Das Datenobjekt kann wiederum ein Feldsymbol sein.

Datenreferenz: **{TYPES|DATA}** *dref* **TYPE REF TO DATA .**

Anlegen eines Datenreferenztyps / einer Datenreferenzvariablen. Die Datenreferenzvariable ist initial, d.h. sie verweist noch auf kein Datenobjekt.

CREATE DATA *dref* **{TYPE** *typ* **| LIKE** *name***} .**
CREATE DATA *dref* **TYPE (***name***) .**

Erzeugt dynamisch zur Laufzeit ein Datenobjekt, die Referenzvariable ***dref*** zeigt auf dieses Datenobjekt.

GET REFERENCE OF *object* **INTO** *dref* **.**

Eine Datenreferenz auf ein bestehendes Datenobjekt wird in eine Referenzvariable gestellt. ***object*** ist ein statisch deklariertes Datenobjekt oder ein Feldsymbol, das auf ein Datenobjekt zeigt.

ASSIGN *dref***->* TO <*feldsymbol*> [CASTING** *typ***] .**

Derenferenziert eine Datenreferenz, um den Zugriff auf den Inhalt des Datenobjekts zu ermöglichen. Bei erfolgreicher Zuweisung wird SY-SYBRC auf 0 gesetzt, ist die Dereferenzierung nicht erfolgreich wird 4 zugewiesen.

2.3 Systemfelder

Systemfelder sind Namen für Datenfelder, die vom SAP-System verwaltet und belegt werden. Über sie erhält man Information über diverse Zustände des Systems.

sy-uline	horizontale Linie der Länge 255
sy-vline	vertikale Linie
sy-datum	Systemdatum (YYYYMMDD)
sy-uzeit	Systemzeit (HHMMSS)
sy-abcde	Alphabet
sy-index	Anzahl der Schleifendurchläufe
sy-subrc	Returncode
sy-dbcnt	Anzahl gelesene Datensätze
sy-uname	Name des angemeldeten Benutzers
sy-repid	Name des aktuellen Programms
sy-mandt	angemeldete Mandantennummer
sy-langu	aktuell eingestellte Sprache
sy-dbsys	Name des zentralen DB-Systems
sy-sysid	Name des R/3-Systems
sy-saprl	Releasestand des SAP-Systems

Dynpro und Listenverarbeitung:

sy-dynnr	Dynpronummer
sy-pfkey	Name des aktuellen PF-Status
sy-ucomm	enthält den ausgelösten Funktionscode (s.h. OK-CODE)
sy-lsind	aktueller Listenindex
sy-listi	Liststufe auslösendes Ereignis
sy-lilli	Nummer der selektierten Zeile
sy-lisel	Inhalt der selektierten Zeile
sy-linct	Anzahl der Zeilen einer Liste
sy-linsz	aktuelle Zeilenbreite der Liste (s.h. LINE-SIZE)

2.4 Operatoren, Funktionen, Wertzuweisung

Arithmetische Operatoren: Sie haben numerische Operanden und liefern einen numerischen Rückgabewert. Bei unterschiedlichen Typen wird automatisch eine Typumwandlung bei konvertiblen Datenobjekten durchgeführt. Bei inkonvertiblen Datenobjekten wird ein Compiler- bzw. ein Laufzeitfehler ausgelöst. Bei arithmetischen Operationen gelten die üblichen Regeln: "Punkt- geht vor Strichrechnung", bei gleichen Operationen Auswertung von "links nach rechts", Klammerausdrücke werden vorrangig ausgewertet.

Bitte beachten: vor und hinter dem Operator muss eine Leerstelle eingefügt werden.

Operation	aritm. Ausdruck	spez. Schlüsselwort
Addition	`r = op1 + op2.`	`ADD op2 TO op1.`
Subtraktion	`r = op1 - op2.`	`SUBTRACT op2 FROM op1.`
Multiplikation	`r = op1 * op2.`	`MULTIPLY op2 BY op1.`
Division	`r = op1 / op2.`	`DIVIDE op1 BY op2.`
Ganzzahldivison	`r = op1 DIV op2.`	
Modulo	`r = op1 MOD op2.`	
Potenzieren	`r = op1 ** op2.`	

Vergleichsoperatoren: Vergleichs- oder relationale Operatoren vergleichen Ausdrücke miteinander. Das Ergebnis ist ein boolscher Wert (Wahrheitswerte), also **`wahr`** oder **`falsch.`**

Operator	Bezeichnung
= oder **EQ**	gleich
<> oder **><** oder **NE**	ungleich
< oder **LT**	kleiner
<= oder **LE**	kleiner gleich
> oder **GT**	größer
>= oder **GE**	größer gleich

Logische Operatoren: Logische oder boolsche Operatoren verknüpfen Wahrheitswerte miteinander. Das Ergebnis ist wiederum ein Wahrheitswerte. Auswertungsreihenfolge: **`NOT`** vor **`AND`**, **`AND`** vor **`OR`**.

Operator	Bezeichnung
NOT	nicht, Verneinung
AND	und-Verknüpfung
OR	oder-Verknüpfung

Wertzuweisung: Die Übertragung erfolgt 1 zu 1 bei kompatiblen Datentypen, es erfolgt eine "sinnvolle" Typkonvertierung. Bei nicht konvertiblen Datenobjekten wird ein Compiler-bzw. Laufzeitfehler erzeugt.

```
destination = source .

MOVE source TO destination .

MOVE-CORRESPONDING s_struct TO d_struct .

CLEAR source .     " Initialwert zuweisen
```

math. Funktionen: An alle aufgeführten mathematischen Funktionen wird ein Parameter übergeben, das Ergebnis wird unter dem Namen der Funktion zurückgegeben. Die Klammern der Parameterliste werden als ABAP-Wort interpretiert und müssen demzufolge mindestens eine Leerstelle vor und hinter dem Parameter haben.

Beispielaufruf:
```
DATA result   TYPE  f.
r = sqrt ( '5.56' ) .
```

Operator	Bezeichnung
abs , sign	Absolutwert , Vorzeichen (Rückgabe -1, 0, 1)
ceil , floor	kleinste / größte ganze Zahl >= / <= Argument
trunc , frac	ganzzahliger Teil / Dezimalteil des Arguments
sin , cos , tan	trigonometrische Funktionen
exp , log	Exponentialfunktion zur Basis e, nat. Logarithmus
sqrt	Quadratwurzel

Zeichenketten verarbeiten:

verketten:
```
CONCATENATE teil1 ... teiln INTO zeichenkette
            [SEPARATED BY c].
```

zerlegen:
```
SPLIT zeichenkette AT zeichen INTO teil1 ... teiln .
```

verschieben:
```
SHIFT zeichenkette [BY n PLACES]
      [{LEFT|RIGHT|CIRCULAR}].
```

suchen:
```
SEARCH zeichenkette FOR muster
       [STARTING AT n1] [ENDING AT n2].
```

ersetzen:
```
REPLACE zkette1 WITH zkette2 INTO zkette [LENGTH n].
```

verdichten:
```
CONDENSE zeichenkette [NO-GAPS] .
```

umsetzen:
```
TRANSLATE zeichenkette {TO UPPER CASE |
          TO LOWER CASE|USING rule}
```

2.5 Anweisungen, Kontrollstrukturen

Anweisungen : Anweisungen stellen die kleinsten ausführbaren Einheiten eines Programms dar sie kann eine Deklaration enthalten, einen Ausdruck auswerten oder den Ablauf des Programms steuern. Der Punkt markiert das Ende einer Anweisung.

Verzweigungen:

```
IF bedingung_1 .
   < anweisungsblock_1 >
[ELSEIF bedingung_2 .
  < anweisungsblock_2 >]...

[ELSE .
  < anweisungsblock_n >]
ENDIF .
```

```
CASE variable .
  WHEN wert_1 [ OR .... ] .
   < anweisungsblock_1 >
[ WHEN wert_2 [ OR .... ] .
  < anweisungsblock_2 > ]...

[ WHEN OTHERS .
  < anweisungsblock_n > ]
ENDCASE .
```

Schleifen:

```
WHILE bedingung .
   < anweisungsblock >
ENDWHILE .

DO [ n TIMES ] .
   < anweisungsblock >
ENDDO .
```

Schleifen beenden **[CONTINUE .]**

Rücksprung zum Schleifenkopf, Schleifenbedingung wird erneut geprüft.

[CHECK *bedingung* **.]**

Rücksprung zum Schleifenkopf, falls die Bedingung bei der Auswertung **false** ergibt, ansonsten wird die nächste Anweisung nach **CHECK** ausgeführt.

[EXIT .]

Sofortiger Abbruch der Schleife, die nächste Anweisung nach dem Schleifenrumpf wird ausgeführt.

2.6 Interne Tabellen

Interne Tabellen sind dynamische, tabellenartige Strukturen im Hauptspeicher. Sie werden bestimmt durch die Tabellenart (Organisation der Tabelle), den Zeilentyp (Strukturierung der Spalten) und den Tabellenschlüssel (eindeutig, nicht eindeutig). Interne Tabellen werden auch als Ersatz für nicht vorhandene Arrays genutzt.

Zugriffsarten:

	Standardtabelle	sortierte Tabelle	Hash-Tabelle
Indexzugriff	*ja*	*ja*	*nein*
Schlüsselzugriff	*ja*	*ja*	*ja*
Schlüsselwerte	*nicht eindeutig*	*eind. o. nicht eind.*	*eindeutig*
bevorzug. Zugriff	*vorwieg. Index*	*vorwieg. Schlüssel*	*nur Schüssel*

interne Tabellen:

```
{TYPES|DATA} itab {TYPE | LIKE} <tabkind>
       OF {linetype / lineobject}
      [WITH {UNIQUE|NON-UNIQUE} <keys>]
      [INITIAL SIZE n] .
```

mit: **<*tabkind*>: {STANDARD TABLE|SORTED TABLE|HASHED TABLE}**

<*keys*>: {KEY col_1 **..** col_n **|KEY tableline | DEFAULT KEY }**

Attribute interner Tabellen bestimmen:

```
DESCRIBE TABLE itab [LINES anzahl] [OCCURS initial]
      [KIND tabellen_art] .
```

Mit **LINES** wird die Anzahl der gefüllten Zeilen, mit **OCCURS** die initiale Größe der internen Tabelle ermittelt. **KIND** ermittelt die Tabellenart (**'T'** für Standard-Tabellen, **'S'** für sortierte und **'H'** Hash-Tabellen).

Zeilenanzahl:

```
LINES( itab ) .
```

Gibt die Anzahl der gefüllten Zeilen einer internen Tabelle zurück.

Zeilen einfügen:

```
APPEND {wa_zeile | LINES OF jtab} TO itab .

INSERT {wa_zeile | LINES OF jtab}
        INTO TABLE itab .

INSERT wa_zeile INTO itab [INDEX idx] .
```

Zeilen ändern:

```
MODIFY TABLE itab FROM wa_zeile .

MODIFY itab FROM wa_zeile INDEX idx .
```

Zeilen lesen:

```
READ TABLE itab INDEX idx INTO wa_zeile .

READ TABLE itab FROM keyline INTO wa_zeile .

READ TABLE itab WITH TABLE KEY
     comp1 = wert1 ... compn2 = wertn INTO wa_zeile .
```

keyline ist eine strukturierte Variable mit derselben Struktur wie die interne Tabelle *itab*, in der der Keywert eingetragen wird.

Zeilen löschen:

```
DELETE itab INDEX idx .

DELETE TABLE itab FROM key .

DELETE itab [FROM idx1] [TO idx2] .

DELETE itab WHERE log_ausdruck .
```

Verarbeiten in einer Schleife:

```
LOOP AT itab INTO wa_zeile
      [{ FROM idx1 TO idx2 | WHERE log_ausdruck }] .

   < Anweisungsblock > .

ENDLOOP .
```

Sortieren:

```
SORT itab [{ASCENDING | DESCENDING}]
      [ BY {comp1 [{ASCENDING | DESCENDING}]} ... ] .
```

Löschen / Freigeben:

```
REFRESH itab .

FREE itab .
```

3 ABAP-Programme

3.1 Ereignisse in Typ 1-Programmen

Report:

```
REPORT z_name [MESSAGE-ID klasse] [LINE-SIZE size]
       [LINE-COUNT count] [NO STANDARD PAGE HEADING].
[ <globale Typ- und Datendeklarationen> ]
[LOAD-OF-PROGRAM .]
[INITIALIZATION.]
[ <Selektionsbildereignisse> ]
START-OF-SELECTION.
[END-OF-SELECTION.]

[ <Listenereignisse> ]
```

Selektionsbild:

```
[AT SELECTION-SCREEN.]              " PAI-Ereignis
[AT SELECTION-SCREEN ON feld.]      " PAI für ein Feld
[AT SELECTION-SCREEN OUTPUT.]       " PBO-Ereignis
```

Listen:

```
[TOP-OF-PAGE.]
[TOP-OF-PAGE DURING LINE-SELECTION.]
[END-OF-PAGE.]
[AT LINE-SELECTION.]     " Steuerung bei Doppelklick
[AT USER-COMMAND.]       " Steuerung über Funktionscodes
```

3.2 Selektionsbilder

Selektionsbilder sind Bildschirmbilder (Dynpros), die über eine Deklaration im Programm definiert werden. Sie werden benutzt, um in Reports standardisierte Benutzereingaben zu ermöglichen. Die Anzeige im Programm erfolgt automatisch durch das Laufzeitsystem. Bei Variablen sind hier nur maximal 8 Zeichen erlaubt.

Eingabefelder:

```
PARAMETERS p_name[(laenge)]
   {TYPE typ | LIKE name} [DECIMALS dez]
   [DEFAULT wert] [LOWER CASE]
   [OBLIGATORY]   [VALUE CHECK]   .
```

Auswahlfelder:

```
PARAMETERS p_name AS CHECKBOX
           [DEFAULT wert] .

PARAMETERS p_name RADIOBUTTON GROUP gruppe
           [DEFAULT wert] .
```

Radiobuttons mit demselben Gruppennamen werden gruppiert, d.h. es kann nur genau ein Button selektiert werden, zuvor selektierter Button wird deselektiert.

Wertemengen:

```
SELECT-OPTIONS sel_tab FOR d_objekt
   ....  [NO INTERVALS] [NO-EXTENTION] .
```

Formatierung:

```
SELECTION-SCREEN SKIP [n].

SELECTION-SCREEN ULINE [[/]pos(laenge)].

SELECTION-SCREEN COMMENT
            [/]pos(laenge) variable .
```

Leerzeilen, Striche oder zusätzliche Texte (Kommentare) einfügen.

```
SELECTION-SCREEN BEGIN OF LINE .

     < Selektionsbild-Elemente >

SELECTION-SCREEN END OF LINE .
```

Selektionsbilder werden innerhalb des Blocks nicht untereinander, sondern nebeneinander angeordnet (ohne vorangestelltem Text!!).

```
SELECTION-SCREEN BEGIN OF BLOCK name
             [WITH FRAME [TITLE titel]] .

    < Selektionsbild-Elemente >

SELECTION-SCREEN END OF BLOCK name .
```

Selektionsbilder werden in einem Block mit Rahmen und Titel dargestellt.

eigenständige Bilder:

```
* Definition
SELECTION-SCREEN BEGIN OF SCREEN dynnr
          [TITLE titel]] [AS WINDOW] .

    < Selektionsbild-Elemente >

SELECTION-SCREEN END OF SCREEN dynnr .

* Aufruf
CALL SELECTION-SCREEN dynnr
    [STARTING AT x1  y1]
    [ENDING AT x2  y2] .
```

Das Selektionsbild wird in einem eigenständigen, freien Windowfenster definiert. Der Aufruf erfolgt nicht mehr automatisch durch das Laufzeitsystem, sondern muss im Programm explizit erfolgen.

3.3 Ausgaben in Listen

Ausgabe:

```
WRITE [AT][/][position][(laenge)] feld
      [UNDER anderes_feld] [DECIMALS anz] [EXPONENT ex]
      [COLOR nr] [<Farboptionen siehe FORMAT>]
      [NO-GAP] [NO-SIGN] [NO-ZERO]
      [{LEFT-JUSTIFIED | CENTERED | RIGHT-JUSTIFIED}]
      [USING EDIT MASK maske] [DD/MM/YYYY]
      [INPUT AS CHECKBOX]
      [QUICKINFO info] .
```

Leerzeilen:

```
SKIP { TO LINE n | [ n ] } .
```

horizontale Linie:

```
ULINE [AT [/][position][(laenge)]].
```

vertikale Linie:

```
WRITE [AT] [/][position] sy-vline.
```

Formatierung:

```
FORMAT [COLOR {{nr | OFF} |  = farbe}]
       [INTENSIFIED [{ON | OFF} | = flag]
       [INVERSE     [{ON | OFF} | = flag]
       [RESET] .
```

Bei **INTENSIFIED ON** oder **0** wird eine intensive (Standard), bei **OFF** oder ungleich **0** eine schwache Hintergrundfarbe verwendet.

INVERSE OFF setzt die Hintergrundfarbe (Standard), **INVERSE ON** die Vordergrundfarbe.

Wird die Farbe oder Option dynamisch zur Laufzeit mit Hilfe der Variablen ***flag*** gesetzt, so muss die Farbe bzw. Option mit dem Zuweisungsoperators (**=**) zugewiesen werden.

Farben:

0	**COL_BACKGROUND**	GUI-abhängig
1	**COL_HEADING**	Graublau
2	**COL_NORMAL**	Hellgrau
3	**COL_TOTAL**	Gelb
4	**COL_KEY**	Blaugrün
5	**COL_POSITIVE**	Grün
6	**COL_NEGATIVE**	Rot
7	**COL_GROUP**	Violett

Windowfenster:

```
WINDOW STARTED AT x1  y1 [ ENDING AT x2  y2 ] .
```

Nur für die Listenverarbeitung: Die aktuelle Ausgabe erfolgt in ein modales Windowfenster. Die linke obere Ecke des Fensters entspricht der Spalte ***x1*** und der Zeile ***y1***.

3.4 Open SQL

Programme, die in "ABAP Open SQL" geschrieben wurden, sind unabhängig von dem eingesetzten Datenbanksystem und dessen "native SQL". Die Programme sind in jedem SAP-System lauffähig.

Zugriffsfunktionen:

Schlüsselwort	Funktion
SELECT	*Daten von DB-Tabellen lesen*
INSERT	*Zeilen in DB-Tabellen einfügen*
UPDATE	*Zeileninhalte in DB-Tabellen ändern*
MODIFY	*Zeilen in DB-Tabellen einfügen oder ändern*
DELETE	*Zeilen aus DB-Tabellen löschen*

DB-Tabelle lesen:

```
SELECT          < was >
      FROM        < woher >
      INTO        < wohin >
      [ WHERE     < Bedingung > ]
      [ GROUP BY  < Felder > ]
      [ HAVING    < Bedingung > ]
      [ ORDER BY  < Felder > ]  .
   [< ABAP-Anweisungen > .]...
ENDSELECT .
```

Die einfache Leseoperation ist quasi eine "**While**-Schleife", die solange ausgeführt wird, bis alle Sätze mit den angegebenen Bedingungen gelesen wurden. Das Lesen erfolgt satzweise, die gelesenen Sätze werden im Rumpf der SELECT-Anweisung direkt verarbeitet.

WHERE-Klauseln:

```
* Einfache logische Ausdrücke mit =, <, >, <>,
* <=, >=, NOT, AND, OR, IS NULL
... WHERE carrid = 'UA'

* Intervallprüfung mit BETWEEN
... WHERE seatmax BETWEEN 200 AND 300

* Musterprüfung (zeichenartige Spalten) mit LIKE
* (% für beliebige, _ genau ein Zeichen)
  ... WHERE cityto LIKE '%town'

* Wertelistenüberprüfung mit IN
  ... WHERE cityto IN ('Berlin', 'LONDON')
```

einzelne Sätze lesen:

```
SELECT SINGLE   < was >
   FROM         < woher >
   INTO         < wohin >
   [ < weitere Klauseln > ] .
 [ < Datensatz verarbeiten > ] .
```

in int. Tabelle lesen:

```
SELECT          < was >
   FROM         < woher >
   INTO TABLE   < interne Tabelle >
   [ < weitere Klauseln > ] .
 [ < interne Tabelle verarbeiten > ] .
```

Join:

```
SELECT .. < was >.
   FROM tab_A {[INNER] | LEFT [OUTER]} JOIN tab_B
   ON feld_A1 = feld_B1  [feld_A2 = feld_B2]
   INTO         < wohin >
   [ < weitere Klauseln > ] .
   [< ABAP-Anweisungen > .]...
ENDSELECT .
```

DB-Zeilen einfügen:

```
INSERT INTO db_tabelle VALUES wa_zeile .
INSERT db_tabelle FROM wa_zeile .
INSERT db_tabelle FROM TABLE itab
       [ACCEPTING DUPLICATE KEYS].
```

DB-Zeilen ändern:

```
UPDATE db_tabelle FROM wa_zeile .
UPDATE db_tabelle FROM TABLE itab .
UPDATE db_tabelle
       SET s1 = wert1 [s2 = wert2]...
       [WHERE <bedingung>] .
```

DB-Zeilen modifiz.:

```
MODIFY db_tabelle FROM wa_zeile .
MODIFY db_tabelle FROM TABLE itab .
```

3.5 Nachrichten

Nachrichtentypen:

Typ	Darstellung	Verarbeitung
A	*Dialogfenster*	*Programmabbruch, Rückkehr zum nächst höheren Niveau*
E	*Statuszeile*	*Abbruch Verarbeitungsblock, voriger Bildschirm erneut anzeigen*
I	*Dialogfenster*	*Programmfortsetzung nach Ausgabe der Nachricht*
S	*Statuszeile (Folgebild)*	*Programmfortsetzung nach Ausgabe der Nachricht*
W	*Statuszeile*	*Wie Typ E*
X	*keine*	*Laufzeitfehler mit Kurzdump*

Nachrichten senden:

```
* Statische Spezifizierung der Nachricht
MESSAGE tnnn[(klasse)] [WITH f1 ... f4] .
```

Durch den Zusatz **MESSAGE-ID** *klasse* hinter der **REPORT**-Anweisung gilt die angegebene Klasse für alle Nachrichten im Programm. Die Klasse in der **MESSAGE**-Anweisung kann dann entfallen. Die Platzhalter in den Nachrichtentexten werden durch die angegeben Texte *f1* bis *f4* ersetzt.

```
* Dynamische Spezifizierung der Nachricht
MESSAGE ID klasse TYPE t NUMBER nnn
        [WITH f1 ... f4] .

* Nachricht als freier Text
MESSAGE '<freie Nachricht>' TYPE t1 [DISPLAY LIKE t2] .
```

Die Nachrichten werden mit der Transaktion **SE91** gepflegt, sie stehen in der Tabelle **T100** (*Sprachenschlüssel, Klasse, Nummer, Nachrichtentext*). Ist keine Mehrsprachigkeit gefordert, genügt die **MESSAGE**-Anweisung mit einem freien Text als Nachricht.

4 Verzweigungslisten (interaktive Listen)

Die erste Liste, die mit Hilfe der **WRITE**-Anweisung im Ereignis START-OF-SELECTION erzeugt wird, ist die Grundliste. Mit Hilfe der Ereignisverarbeitung (siehe Ereignisse **AT LINE-SELECTION, AT USER-COMMAND**) können Verzweigungslisten (bis max. 20 weitere Listen) erzeugt werden, die aufgrund einer Benutzeraktion (Doppelklick, Auswahl einer Listenzeile, Funktionscodes) angestoßen werden. Die Daten der vorherigen Liste (i.d.R. die ausgewählte Listenzeile) können an die nachfolgende Liste zur Erzeugung einer Detailliste weiter gereicht werden.

Systemfelder:

SY-LSIND	aktueller Listenindex (Grundliste = 0)
SY-LISTI	Listenindex, die das Listenereignis ausgelöst hat
SY-LILLI	Nummer der selektierten Zeile
SY-LISEL	Inhalt der selektierten Zeile
SY-CUROW	Zeilenposition des Cursors (selektierte Zeile)
SY-CUCOL	Spaltenposition des Cursors (selektierte Zeile)

Wertübergabe: **HIDE** *feld* **.**

Beim Aufbau der Liste werden mit der **HIDE**-Anweisung die Teile der Liste des Datenbereichs ***feld*** in den Hide-Bereich weg geschrieben, die an die nachfolgende Liste weitergegeben werden sollen. Markiert der Benutzer eine Listezeile mit einem Doppelklick, so wird die selektierte Zeile im Ereignis **AT LINE-SELECTION** aus dem HIDE-Bereich im Datenobjekt ***feld*** wieder zur Verfügung gestellt.

Listen modifizieren:

```
MODIFY LINE n [INDEX i | OF CURRENT PAGE | PAGE p]
            [ <Modifikationen> ] .

MODIFY CURRENT LINE [ <Modifikationen> ] .
```

Modifiziert die Zeile ***n*** / die aktuelle Zeile der Liste. Je nach Option wird die ganze Zeile, ausgewählte Felder oder deren Inhalt verändert. Die **MODIFY**-Anweisung bezieht sich immer auf die zuletzt mit **F2** ausgewählte bzw. die zuletzt mit **READ** gelesene Zeile. Ohne den Zusatz *<Modifikationen>* wird der aktuelle Inhalt der Zeile in das Systemfeld **SY-LISEL** gestellt.

<Modifikationen>:

```
{LINE FORMAT option1 [optioin2]... |
 FIELD FORMAT feld1 format1 [feld2 format2]... |
 FIELD VALUE feld1 FROM wert1
             [feld2 FROM wert2]...}
```

option1, *option2* sind Formatierungsoptionen der **FORMAT**-Anweisung, *format1*, *format2* steht für eine ganze Formatierungsanweisung.

Zeilen lesen:

```
READ LINE n [INDEX i]  [OF CURRENT PAGE | PAGE p]
           [LINE VALUE INTO wa_zeile]
           [FIELD VALUE feld1 [INTO progfeld1]
                      [feld2 [INTO progfeld2]... ] ] .
```

Liest die Zeile ***n*** nach einem interaktiven Ereignis aus der Liste aus. Je nach Option wird die ganze Zeile oder ausgewählte Felder ausgelesen. Ohne Zusätze wird der Inhalt der Zeile in das Systemfeld **SY-LISEL** gestellt und alle HIDE-Informationen gelesen.

```
READ CURRENT LINE [FIELD VALUE feld1 [INTO progfeld1]
                                [feld2 [INTO progfeld2]... ] ] .
```

Liest die aktuelle Zeile nach einem interaktiven Ereignis oder eine zuvor mit der `READ LINE` Anweisung gelesene Zeile aus der Liste aus.

Attribute lesen:

```
DESCRIBE LIST NUMBER OF {LINES | PAGES} anz [INDEX i] .
```

Schreibt die Anzahl der Zeilen bzw. Seiten in die Variable anz.

```
DESCRIBE LIST PAGE p [INDEX i] [<Optionen>] .
<Optionen>: {LINE SIZE col | LINE COUNT len | LINES l}
```

Liest die in `Optionen` angegebene Eigenschaften (Seitenbreite, Seitenlänge, Anzahl Zeilen).

5 Unterprogramme

Unterprogramme werden i.d.R. lokal am Ende eines Programms definiert. Sie können innerhalb des Programms beliebig oft aufgerufen werden. Der Zugriff aus einem anderen Programm auf ein Unterprogramm ist nicht möglich. Unterprogramme werden auch innerhalb von Funktionsbausteinen und Dialogbausteinen eingesetzt. Die Funktionalität der logischen Datenbanken wird über Unterprogramme realisiert.

Definition:

```
FORM up_name [{USING | CHANGING}
  {VALUE(p1) | p1}    [<Typangabe>]]... .

   <ABAP-Anweisungen>

ENDFORM .
```

Aufruf:

```
PERFORM up_name [USING p11 [p12]...]
                [CHANGING p21 [p22]...] .
```

Die Schlüsselworte **USING** und **CHANING** legen nicht den Übergabemechanismus fest, sondern sie dokumentieren nur, ob ein Wert übergeben oder an das rufende Programm zurück gereicht wird!

Parametertypen:

```
* vollständige Typisierung
TYPE { d | f | i | t | typ |
     LINE OF itab }

LIKE name

* Generische Typisierung
TYPE {ANY | c | n | p | x |
  [{ANY | INDEX | SORTED | HASHED}] TABLE}
```

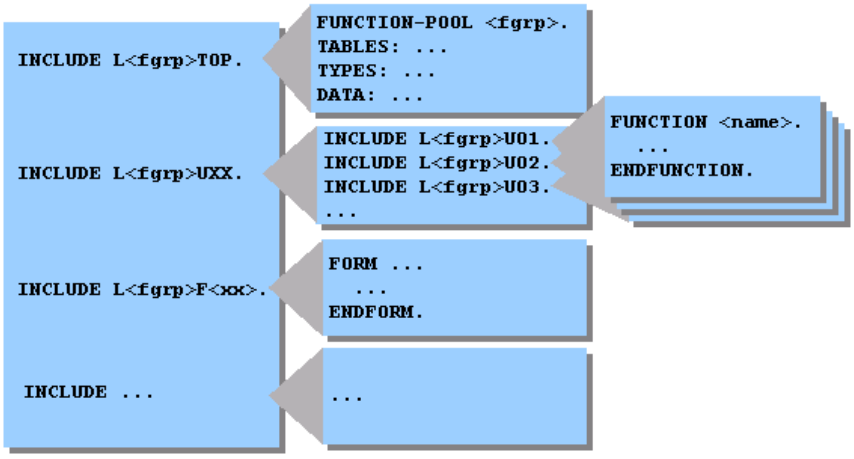
6 Funktionsbausteine

Funktionsbausteine sind global verfügbare Prozeduren / Funktionen, die in jedem ABAP-Programm aufgerufen werden können. Mehrere Funktionsbausteine und ein globale Datenbereich werden in einer Funktionsgruppe zusammengefasst (Rahmenprogramm), auf den globalen Datenbereich können alle Funktionsbausteine der Gruppe zugreifen. Beim Aufruf eines Funktionsbausteins wird die gesamte Funktionsgruppe in den Speicher geladen.

Funktionsbausteine werden mit dem Function-Builder angelegt. Sie können "***remote-fähig***" gemacht werden, so dass auch Fremdprogramme auf Daten des SAP-Systems zugreifen können. Die Funktionsgruppe mit den Funktionsbausteinen werden mit Hilfe von geschachtelten INCLUDE-Anweisungen aufgebaut, die der Function Builder verwaltet.

Funktionsgruppen / Funktionsbausteine können alle Komponenten eines Programms enthalten (Unterprogramme, Dynpros, Selektionsbilder, Module usw.).

Aufbau:



Deklaration: Funktionsbausteine können nur mit Hilfe des **Function Builders** angelegt werden. Die Definition der Schnittstellen erfolgt über Registereinträge.

Sichtweise beachten: Bei der Definition erhält die Funktion über die **IMPORTING**-Schnittstelle die Parameter, über die **EXPORTING**-Schnittstelle werden die Werte an das rufende Programm zurück gereicht.

Aufruf:

```
CALL FUNCTION 'funktionsbaustein'
    [EXPORTING     f1 = a1 ... fn = an]
    [IMPORTING     f1 = a1 ... fn = an]
    [CHANGING      f1 = a1 ... fn = an]
    [EXCEPTIONS    e1 = r1 ... en = rn]
    [OTHERS = r0] .
```

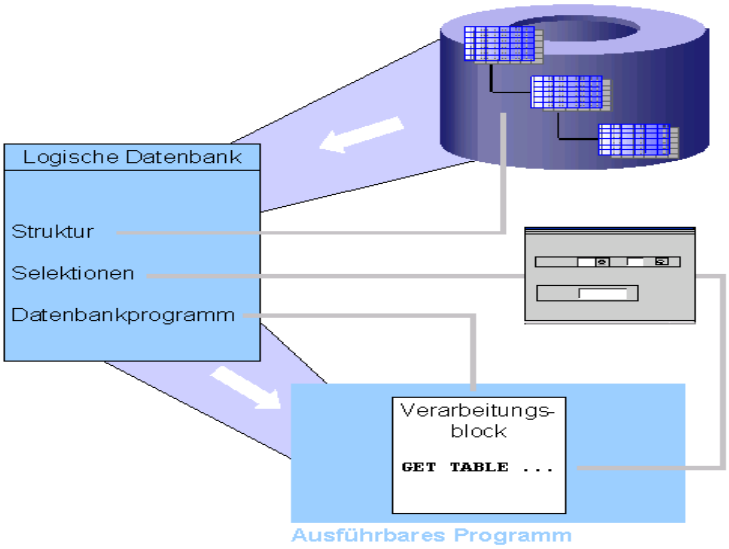
Den formalen Parametern (***f1*** bis ***fn***) werden die aktuellen Werte (***a1*** bis ***an***) zugewiesen, die Exceptions (***e1*** bis ***en***) erhalten konkrete Werte (***r1*** bis ***rn***). Diese Werte werden nach dem Aufruf dem Systemfeld **SY-SUBRC** (Return-Code) übergeben. Steht in dem Systemfeld ein Wert **> 0**, hat die zuvor aufgerufene Funktion eine Ausnahme (Fehler) ausgelöst.

Sichtweise beachten: Die **EXPORTING**-Schnittstelle beim Aufruf wurde bei der Deklaration als **IMPORTING**-Schnittstelle beschrieben, die **IMPORTING**-Schnittstelle beim Aufruf entspricht der **EXPORTING** -Schnittstelle bei der Deklaration

7 Logische Datenbank

In einer logischen Datenbank werden die Leseoperationen, Struktur der Tabellen, mit denen gearbeitet wird und die Selektionsbilder vom eigentlichen ABAP-Programm abgetrennt. Im ABAP-Programm (Report) stehen danach nur noch die Verarbeitungsfunktionen. Damit können sehr häufig gleiche oder gleichartige Leseoperation der Programme ausgelagert und an zentraler Stelle verwaltet werden. Eine logische Datenbank kann mit beliebig vielen ABAP-Programme verknüpft werden. Für das Anlegen und Verwalten der logischen Datenbank gibt es ein eigenständiges Werkzeug.

Aufbau:



Datenbankprogramm der logischen DB

Schnittstelle:

```
TABLES: tabelle1 [, tabelle2]...  .
```

Die **TABLES**-Anweisung ist eine veraltete Form für die strukturierte Zeilenvariable in der **SELECT**-Anweisung. Sie ist identisch mit folgender **DATA**-Anweisung:

```
DATA: tabelle1 TYPE tabelle1, [tabelle2 TYPE tabelle2].
```

Unterprogramme:

```
FORM INIT .
FORM PBO .
FORM PAI .

FORM PUT_knoten .

   < Leseoperationen und sonstige ABAP-Anweisungen >

   PUT knoten .

ENDFORM .
```

In diesem Unterprogramm stehen die Leseoperationen für die angegebene Tabelle (Knoten). Die Anweisung **PUT** *knoten* löst das entsprechende **GET**-Ereignis im Report aus.

Reports mit der logischen DB

Schnittstellen:
```
NODES: knoten_name1 [, knoten_name2]...  .
TABLES: tab_name1 [, tab_name2]...  .
```

Verarbeitung:
```
START-OF-SELECTION .

* Ereignis für die Verarbeitung des Knotens tab_name
[GET tab_name [LATE] [FIELDS f1 f2 ...] .]....

* letzter Verarbeitungsblock vor dem Programmende
END-OF-SELECTION .
```

Regeln für die Gruppenverarbeitung mit der logischen DB

Verarbeitung: Im Ereignisblock **GET** *tab_name* erfolgt die Verarbeitung (z.Bsp. Summenbildung, Min., Max. etc) des Knotens *tab_name*.

Vorverarbeitung: Im **GET-**Ereignis des Vorgängerknotens (in Bezug zum Knoten *tab_name*) erfolgt die Initialisierung der Variablen.

Nachverarbeitung: Im **GET-LATE-**Ereignis des Vorgängerknotens (in Bezug zum Knoten *tab_name*) werden die Ergebnisse der Verarbeitung ausgegeben bzw. die Überträge für die übergeordneten Knoten gebildet.

8 Dialogprogrammierung

8.1 Dynpro

Dynpro-Ablauflogik:
```
PROCESS BEFORE OUTPUT.
   MODULE modul_1 .

PROCESS AFTER INPUT.
   MODULE modul_2 .

[PROCESS ON HELP-REQUEST.]

[PROCESS ON VALUE-REQUEST.]
```

ABAP-Programm:
```
* PBO-Module
MODULE modul_1 OUTPUT.
   <ABAP-Anweisungen>
ENDMODULE.

* PAI-Module
MODULE modul_2 INPUT.
   <ABAP-Anweisungen>
ENDMODULE.
```

Aufruf:

```
CALL SCREEN dynpro_nr
   [STARTING AT x1 y1] [ENDING AT x2 y2].
CALL SELECTION-SCREEN dynpro_nr
   [STARTING AT x1 y1] [ENDING AT x2 y2].
```

Folgedynpro:

```
SET SCREEN dynpro_nr .
```

Dynpro beenden:

```
LEAVE SCREEN .
LEAVE TO SCREEN dynpro_nr .
LEAVE PROGRAM
```

Listprozessor:

```
LEAVE to LIST-PROCESSING [AND RETURN TO SCREEN dynnr].
LEAVE LIST-PROCESSING
```

Wechselt vom Dynpro-Prozessor zum Listprozessor, alle bisherigen Ausgaben der Grundliste werden angezeigt. Nach Beendigung des Listprozessors geht die Kontrolle wieder an den Dynpro-Prozessor.

8.2 GUI-Status, GUI-Titel

Der GUI-Status und der GUI-Titel sind eigenständige Objekte, die über nachfolgende Anweisungen dem aktuellen Fenster zugewiesen werden. Mit dem GUI-Status wird das Menü, die Druck- und Funktionstastenleiste des Fensters beschrieben. Hierfür gibt es einen speziellen Editor.

Aufruf:

```
SET PF-STATUS '<name>'  .
SET TITLEBAR '<titel>'  .
```

wichtige Funktionscodes:

Symbol	Taste	Code	Beschreibung
	F3	**BACK**	zurück
	Umsch F3	**EXIT**	beenden
	F12	**CANCEL**	abbrechen
		**SAVE**	sichern
	F2	**SELE**	auswählen
		PICK	AT LINE-SELECTION (nur Listen)
	F9	**MARK**	markieren
	shift F2	**DELETE**	löschen
		**ENTER**	Eingabe
	Strg P	**PRINT**	drucken

9 Objektorientierte Programmierung

9.1 Klassendeklaration und Implementierung

Definition:

```
CLASS klassen_name DEFINITION
      [ABSTRACT | FINAL]
      [INHERITING FROM oberklassen_name ].
   PUBLIC SECTION.
   ...<öffentliche Deklarationen>
   PROTECTED SECTION.
   ...<geschützte Deklarationen>
   PRIVATE SECTION.
   ...<private Deklarationen>
ENDCLASS
```

Datendeklarationen und Deklarationen der Methodensignaturen (Schnittstellen) in den Sichtbarkeitsbereichen (Attribute und Methoden für Instanzen und Klassen).

```
CLASS klassen_name DEFINITION [DEFERRED] .
```

Zeigt dem Compiler, dass eine Klassendefinition weiter unten im Code steht. Diese Anweisung ist notwendig, falls eine Klasse vor ihrer Definition benutzen wird.

Zugriffsrechte:

```
CLASS klassen_name DEFINITION
   [CREATE {PUBLIC | PROTECTED | PRIVATE} ] .
```

Steuerung der Instanziierung: Öffentliche, geschützte, private Instanziierbarkeit.

```
CLASS klassen_name DEFINITION
   [FRIENDS klasse_1 ... klasse_n ] .
```

Freundschaft unter Klassen ermöglicht den Zugriff auf private und geschützte Komponenten. Freundschaft ist ein einseitiges Verhältnis, sie ist nicht vererbbar.

Implementierung:

```
CLASS klassen_name IMPLEMENTATION .
   ...<Implementierung der Methoden>
ENDCLASS
```

Implementierung der Instanz- und Klassenmethoden (ohne Schnittstellen).

Instanz- und Klassenattribute:

```
{DATA | CLASS-DATA} name[(laenge)]
   {TYPE typ | LIKE name}
      [DECIMALS dezimal] [VALUE wert][READ-ONLY].
```

Siehe auch Datentyp, Typ- und Variablendeklaration.

Deklaration Instanz- u. Klassenmethoden:

```
{METHODS | CLASS-METHODS} methoden_name
         [ABSTRACT][FINAL]
   [IMPORTING <Liste form. Parametern>]
   [EXPORTING <Liste form. Parametern>]
   [CHANGING <Liste form. Parametern>]
   [RETURNING VALUE(var) TYPE typ]
   [EXCEPTIONS <Liste mit Ausnahmen>] .
```

Die Schnittstellendefinitionen erfolgt analog zu den Funktionsbausteinen!

Sichtweise beachten: Bei der Definition erhält die Methode über die **IMPORTING**-Schnittstelle die Parameter, über die **EXPORTING**-Schnittstelle werden die Werte an das rufende Programm zurück gereicht.

Liste formale Parameter:

```
{f_par | NAME(f_par)} {TYPE typ / LIKE dobj}
      [OPTIONAL | DEFAULT d_wert]
   [ <nächster/weitere Parameter> ]...  .
```

Namenskonventionen: Methodenparameter beginnen mit ***i_*** bei **IMPORTING**-, ***e_*** bei **EXPORTING**-, ***c_*** bei **CHANGING**-, ***r_*** bei **RESULT**-Parameter, um Verwechslungen mit den Attributen zu vermeiden:

Redefinition:

```
METHODS methoden_name REDEFINITION .
```

Überdefinition geerbter Methoden, Schnittstelle darf nicht geändert, Methode muss im Implementierungsteil neu implementiert werden.

Konstruktoren: Konstruktoren werden wie Methoden deklariert und implementiert, ihr Name ist festgelegt (geschützt): **CONSTRUCTOR** für den Instanzkonstruktor und **CLASS_CONSTRUKTOR** für den Klassenkonstruktor. Konstruktoren müssen in der **PUBLIC SECTION** definiert werden, tatsächliche Zugriffsrechte werden bei der Klasse festgelegt (siehe erweiterte Zugriffsrechte). Instanzkonstruktoren können nur **IMPORTING**-Parameter besitzen, beim Werfen einer Exception wird die Instanz nicht angelegt. Klassenkonstruktoren haben keine Parameter und werfen keine Exception.

Interfaces:

```
INTERFACE interface_name .
   <DATA Anweisungen>.
   <CLASS-DATA Anweisungen>.
   <METHODS Deklarationen>.
   <CLASS-METHODS Deklarationen>.
ENDINTERFACE .
```

Ein Interface enthält die gleichen Komponenten wie eine Klasse (ohne **SECTIONs**, ohne Implementierungsteil).

```
INTERFACES interface_name .
```

```
ALIASES alias_name FOR interface_name~methoden_name .
```

Implementierung der Interfaces und Aliasnamen im öffentlichen Deklarationsteil der Klassen.

Methoden-implementierung:

```
METHOD {methoden_name |
        interface_name~methode |.alias_name } .
   <ABAP-Anweisungen>
ENDMETHOD .
```

9.2 Objekte, Methodenaufrufe

```
                * Referenzvariable für das Objekt
```
Deklaration: `DATA ref_variable TYPE REF TO klassen_name .`

Instanziierung: `CREATE OBJECT ref_variable [<Parameter>] .`

Die Parameter des Konstruktors werden wie die Methodenparameter behandelt.

ME / SUPER: **ME** ist eine Referenzvariable, die auf das aktuelle (eigene) Objekt zeigt. Sie wird u.a. benutzt, um den Zugriff auf Attribute und Methoden im eigenen Objekt eindeutig zu machen oder die eigene Instanz weiter zu verarbeiten.

SUPER ist eine Referenzvariable, die auf die direkte Oberklasse verweist. Damit wird der Zugriff auf Attribute und Methoden der Oberklasse realisiert.

Methodenaufrufe:

Instanzmethoden: `ref_variable->methodenname( [<Parameter>] ).`

`variable = ref_objekt->methodenname( [<Parameter>] ).`
Funktionale Methoden geben genau einen Wert zurück.

Hinweis: Nachfolgender Methodenaufruf mit **"CALL METHOD ..."** ist für statische Methoden obsolet. Er sollte nur noch für den dynamischen Methodenaufruf benutzt werden.

`CALL METHOD ref_variable->methodenname [<Parameter>] .`

<Parameter>:
```
[EXPORTING    f1 = a1 ... fn = an]
[IMPORTING    f1 = a1 ... fn = an]
[CHANGING     f1 = a1 ... fn = an]
[RECEIVING    f1 = a1 ]
[EXCEPTIONS   e1 = r1 ... en = rn]
[OTHERS = r0]
```

Den formalen Parametern (f_1 bis f_n) werden die aktuellen Werte (a_1 bis a_n) zugewiesen, die Exceptions (e_1 bis e_n) erhalten konkrete Werte (r_1 bis r_n). Diese Werte werden nach dem Aufruf dem Systemfeld **SY-SUBRC** (Return-Code) übergeben. Steht in dem Systemfeld ein Wert **> 0**, hat die zuvor aufgerufene Methode eine Ausnahme (Fehler) ausgelöst.

Sichtweise beachten: Die **EXPORTING**-Schnittstelle beim Aufruf wurde bei der Deklaration als **IMPORTING**-Schnittstelle beschrieben, die **IMPORTING**-Schnittstelle beim Aufruf entspricht der **EXPORTING** -Schnittstelle bei der Deklaration (analog zu den Funktionsbausteinen).

Klassenmethoden: `klassen_name=>methodenname( [<Parameter>] ).`

Beim Aufruf der Klassenmethode steht anstelle einer Referenzvariable die Klasse, der Operator ist **=>** (Parameter siehe Instanzmethoden).

Kurzformen: `ref_variable->methodenname( [<Parameter_Kurzform>] ).`
`klasse_name=>methodenname( [<Parameter_Kurzform >] )`

<Parameter_Kurzform>: **{** *datenObjekt* **/** *f1=a1 ... fn=an* **}**

Die Kurzformen können nur eingesetzt werden, falls nur Daten an die Methode übergeben werden (siehe EXPORTING-Parameter beim Aufruf!). Das Schlüsselwort **EXPORTING** kann hier entfallen.

9.3 Ereignisse

Definition:

```
CLASS klassen_name DEFINITION .
  {PUBLIC | PROTECTED | PRIVAT} SECTIONS .
    {EVENTS: | CLASS-EVENTS:}
       ereignis_name [EXPORTING VALUE(fi)
                              TYPE typ ...] .
ENDCLASS.
```

Ereignisse werden ähnlich wie Methoden definiert, die optionale Parameterschnittstelle legt die Schnittstelle des Behandlers fest. Impliziter, vordefinierter Parameter ist ***sender***.

Ereignis auslösen:

```
RAISE EVENT ereignis_name
   [EXPORTING ei = ai  ...] .
```

In einer Methode der Klasse wird das Ereignis ausgelöst. Die Parameterschnittstelle muss mit der Schnittstelle des Ereignisses korrespondieren.

Ereignisbehandler:

```
{METHODS | CLASS-METHODS] handler_name
      FOR EVENT ereignis_name OF klassen_name
      [IMPORTING  {ei ... | sender }] .
```

Jede Klasse kann einen Ereignisbehandler für die Ereignisse enthalten, dies sind speziell gekennzeichnete Methoden. Schnittstelle muss mit der des Ereignisses korrespondieren, die Eingabeparameter werden nicht typisiert. Die Typisierung wird von den Ausgabeparametern des Ereignisses übernommen. Typ des impliziten Parameters ***sender*** ist ***klassen_name***.

Registrierung:

```
SET HANDLER handler1 [handler2] ...
      FOR { ref_variable | ALL INSTANCES }
         [ ACTIVATION {'X' | ' '} .
```

Damit ein Ereignisbehandler auf ein ausgelöstes Ereignis reagiert, muss er zu Laufzeit registriert werden. Erst die Registrierung koppelt den Ereignisbehandler an den Auslöser. Die Kopplung kann jederzeit wieder aufgehoben werden. Bei Angabe von ***'X'*** wird registriert (Standard), bei ' ' deregistriert.

9.4 Exceptions

Zur Behandlung von Ausnahmen gibt es in ABAP zwei Konzepte: klassische und objektorientierte Exceptens. Diese sind nicht miteinander kompatibel und sind nicht ineinander überführbar. Sie sollten auch nicht gemischt benutzt werden. Die objektorientierte Ausnahmebehandlung orientiert sich an der Ausnahmebehandlung in Java.

Klassische Exceptions:

Exception auslösen:

```
RAISE ausnahme .

MESSAGE <Nachricht> [ RAISING ausnahme ] [<Parameter>].
```

Die Ausnahme wird direkt oder zusammen mit einer Nachricht ausgelöst.

Exception behandeln: An die Ausnahme wird ein Wert gebunden (ganze Zahl), die über die Schnittstelle der Funktion / Methode an das rufende Programm zurück gereicht wird (siehe hierzu Definition und Aufruf von Funktionsbausteinen und Methoden). Dieser Wert wird nach dem Aufruf dem Systemfeld **SY-SUBRC** (Return-Code) übergeben. Steht in dem Systemfeld ein Wert **> 0**, hat die zuvor aufgerufene Funktion / Methode eine Ausnahme (Fehler) ausgelöst

Objektorientierte Exceptions:

Exception werfen:

```
RAISE EXCEPTION ausnahme_objekt .

RAISE EXCEPTION TYPE ausnahme_klasse .
```

Ausnahmen sind Objekte von Ausnahmeklassen. Wie in Java gibt es eine Hierarchie von Ausnahmeklassen, die spezielle Fehlersituationen beschreiben. Alle Ausnahmeklassen beginnen mit dem Präfix **CX_**, Oberklasse aller Ausnahmen ist die Klasse **CX_ROOT**.

Exception behandeln:

```
TRY .
   [<try-Block>]
[CATCH cx_class1 [cx_class2] [INTO oref] .
   [<catch-Block>]   ]...
[CLEANUP  [INTO oref] .
    [<cleanup-Block>]   ]
ENDTRY .
```

Wie in Java werden alle Anweisungen im **TRY**-Block überwacht. Wird eine Ausnahme geworfen, verzweigt das Programm in den entsprechenden **CATCH**-Block. Der **CLEANUP**-Block wird immer durchlaufen.

oref ist eine Referenzvariable vom Typ des Ausnahmeobjekts (oder deren Oberklasse), an die das geworfene Ausnahmeobjekt gebunden wird. Der Methodenaufruf *oref->get_text( )* liefert die Fehlermeldung des Ausnahmeobjekts.

```
METHODS <Methodenname mit Parameter> ...
        [ RAISING cx_class1 [cx_class2]... ] .
```

Die Ausnahme wird nicht aufgefangen und behandelt, sondern an das rufende Programme weiter geleitet.

9.5 Control Frame Work

Für die Objekte des **Control Frame Work** gibt es eine Vielzahl von Methoden, die selbst eine Menge von Parameter aufweisen können, die größtenteils optional sind. In den nachfolgenden Methodenbeschreibungen werden für das Control Frame Work nur eine Auswahl der verfügbaren Methoden beschrieben.

Dort wo ein boolescher Parameter erwartet wird, zeigt **<false>** bzw. **<true>** nur die Standardeinstellung an. Die booleschen Werte **true** oder **false** werden in ABAP durch die Zeichen **'X'** (bzw. irgend ein Zeichen) und **' '** nachgebildet, dafür können auch die implementierten Datennamen **abap_true** oder **abap_false** benutzt werden.

Die hier gewählte Beschreibung lehnt sich an die generierten Muster an, wobei aus Platzgründen nicht immer alle optionalen Parameter aufgeführt werden, dies gilt besonders für die Exceptions. Wird eine Ausnahme ausgelöst, so muss direkt nach dem Aufruf das Systemfeld **sy-subrc** abgefragt werden. Nachfolgendes Codefragment wird i.d.R. als Muster für das Abfragen des Rückkehrcodes generiert:

```
if sy-subrc <> 0 .
* MESSAGE ID SY-MSGID TYPE SY-MSGTY NUMBER SY-MSGNO
*            WITH SY-MSGV1 SY-MSGV2 SY-MSGV3 SY-MSGV4.
endif .
```

Container

Klasse: **cl_gui_custom_container**

Darstellung von Controls auf Dynpros und Subscreens. ***ref_container*** ist eine Instanzvariable vom Typ dieser Klasse.

Instanziierung:

```
CREATE OBJECT <ref_container>
    EXPORTING
     [ parent          =   ]
       container_name = '<Name des Dynpro-Containers>'
     [ style           =   ]
  [EXCEPTIONS <Liste mit Ausnahmen> ]  .
```

Weitere Containerklassen: **CL_GUI_DOCKING_CONTAINER** (Controls als gedocktes Teilfenster), **CL_GUI_SPLITTER_CONTAINER** (Darstellung und Gruppierung mehrer Container in Zellen), **CL_GUI_DIALOGBOX_CONTAINER** (Darstellung von Controls in einem amodalen Dialogfenster).

Picture Control

Klasse: **cl_gui_picture**

Anzeigen von Bildern (**bmp**-, **jpg**-, **gif**-Datein) auf dem Bildschirm. ***ref_picture*** ist eine Instanzvariable vom Typ dieser Klasse.

Instanziierung:

```
CREATE OBJECT <ref_picture>
    EXPORTING
      parent = <ref_container>
    [ name   =   ]
  [ EXCEPTIONS <Liste mit Ausnahmen> ]  .
```

Methoden:

```
<ref_picture>->load_picture_from_url(
    EXPORTING url = <url_link> ) .
<ref_picture>->load_picture_from_url( <url_link> ) .
<url_link> = '<file://c:\verzeichnis\bild.jpg>'
```

Bild in das Picture Control laden.

```
<ref_picture>->clear_picture(  ) .
```

Bild aus der Anzeige löschen.

```
<ref_picture>->set_display_mode(
    EXPORTING display_mode = <dmode> ) .
```

```
<ref_pictur>e->set_display_mode( <dmode> ) .
```

Anzeigeeigenschaften des Bildes ändern. Folgende Klassenkonstanten enthalten die möglichen Modi (*<dmode>*): `display_mode_strech, display_mode_normal, display_mode_normal_center, display_mode_fit, display_mode_fit_center.`

Events:

```
PICTURE_CLICK  ,  PICTURE_DBLCLICK
cl_gui_picture=>EVENTID_PICTURE_CLICK
cl_gui_picture=>EVENTID_PICTURE_DBLCLICK
```

Event und Event-ID des Klick- bzw. des Doppelklickereignisses.

Textedit Control

Klasse: `cl_gui_textedit`

Einfacher Texteditor mit den üblichen Funktionen. ***ref_textedit*** ist eine Instanzvariable vom Typ dieser Klasse.

Instanziierung:

```
CREATE OBJECT <ref_textedit>
    EXPORTING
    [ wordrap_mode     = wordrap_at_windowborder  ]
    [ wordrap_position = -1  ]
      parent           = <ref_container>
  [ EXCEPTIONS <Liste mit Ausnahmen> ]  .
```

Methoden:

```
<ref_textedit>->set_text_as_stream(
    EXPORTING text = <text_itab>
  [ EXCEPTIONS <Liste mit Ausnahmen> ]  .

<ref_textedi>t->set_text_as_r3table(
    EXPORTING table = <text_itab>
  [ EXCEPTIONS <Liste mit Ausnahmen> ]  .
```

Text in den Editor laden. ***text_itap*** ist eine interne Tabelle vom Typ **c** (256 Byte). Als ***stream*** kann der Text Steuerzeichen enthalten, bei einer ***r3table*** entspricht jede Zeile der internen Tabelle einer Editorzeile.

```
<ref_textedi>t->get_text_as_stream(
  [ EXPORTING
       only_when_modified     = <false> ]
    IMPORTING
       text                   = <text_itab>
     [ is_modified            =   ]
  [ EXCEPTIONS <Liste mit Ausnahmen> ]  .

<ref_textedit>->get_text_as_r3table(
  [ EXPORTING
      only_when_modified     = <false> ]
    IMPORTING
      table                  = <text_itab>
    [ is_modified            =   ]
  [ EXCEPTIONS <Liste mit Ausnahmen> ]  .
```

Text aus dem Editor lesen.

```
<ref_textedit>->set_readonly_mode( [{'X' | ' '}] ).
```

Zwischen dem Lese- und Eingabemodus umschalten.

```
<ref_textedit>->set_toolbar_mode(
     [ EXPORTING statusbar_mode   = <false> ]
     [ EXCEPTIONS <Liste mit Ausnahmen> ]  .

<ref_textedit>->set_statusbar_mode(
     [ EXPORTING statusbar_mode   = <false> ]
     [ EXCEPTIONS <Liste mit Ausnahmen> ]  .
```

Sichtbarkeit der Toolbar, Statuszeile ein- und ausschalten.

```
<ref_textedit>->find_and_replace(
    EXPORTING
        [ case_sensitive_mode     = <false>  ]
          replace_string          = <r_string>
          search_string           = <s_string>
        [ whole_word_mode         = <false>  ]
  [ CHANGING string_found         = <f_string> ]
  [ EXCEPTIONS <Liste mit Ausnahmen> ]  .

<ref_textedit>-> replace_all(
    EXPORTING
        [ case_sensitive_mode     = <false>  ]
          replace_string          = <r_string>
          search_string           = <s_string>
        [ whole_word_mode         = <false>  ]
  [ CHANGING counter              = <i_count>  ]
  [ EXCEPTIONS <Liste mit Ausnahmen> ]  .

<ref_textedit>->find_and_select_text(
    EXPORTING
        [ case_sensitive_mode     = <false>  ]
          search_string           = <s_string>
        [ whole_word_mode         = <false>  ]
  [ CHANGING string_found         = <f_string> ]
  [ EXCEPTIONS <Liste mit Ausnahmen> ]  .
```

Sucht ***r_string*** im Text und ersetzt in durch ***s_string*** *bzw.* markiert ihn.

```
<ref_textedit>->save_as_local_file(
  [ EXPORTING file_name                = <dateiname> ]
  [ EXCEPTION error_cntl_call_methode = 1 ] ) .

<ref_textedit>->open_local_file(
  [ EXPORTING file_name                = <dateiname> ]
  [ EXCEPTION error_cntl_call_methode = 1 ] ) .
```

Speichert den Text in eine lokale Datei bzw. lädt eine lokale Datei. Falls der Parameter ***dateiname*** nicht gesetzt ist, wird ein Dialogfenster zur Eingabe des Dateinamens erzeugt.

Hinweis: Die Klasse **cl_gui_frontend_services** enthält u.a. Klassenmethoden (**gui_upload, gui_download**) zum Laden und Speichern von Dateien und zur Kommunikation mit dem lokalen Betriebssystem.

```
<ref_textedit>->delete_text(
     [ EXCEPTION error_cntl_call_methode = 1 ] ) .
```

Löscht den Text im Editor.

HTML-Viewer Control

Der SAP HTML Viewer ist ein von SAP entwickeltes Control für den Einsatz innerhalb des SAP GUI. Der SAP HTML Viewer benutzt die Funktionen, die vom verwendeten Web-Browser auf dem aktuellen Frontend offengelegt werden. Mit Hilfe der Klasse **CL_DD_DOCUMENT** können dynamische Dokumente mit HTML beschrieben und mit dem Viewer angezeigt werden.

Klasse: **cl_gui_html_viewer**

ref_viewer ist eine Instanzvariable vom Typ dieser Klasse.

Instanziierung:

```
CREATE OBJECT <ref_viewer>
    EXPORTING
      parent = <ref_container>
    [ name   =   ]
  [ EXCEPTIONS <Liste mit Ausnahmen> ]  .
```

Methoden:

```
<ref_viewer>->show_url(
    EXPORTING
      url             = <akt_url>
    [ frame           =         ]
    [ in_place        = <true>  ]
  [ EXCEPTIONS <Liste mit Ausnahmen> ]  .
```

Lädt die URL und zeigt den Inhalt im HTML-Viewer.

```
<ref_viewer>->go_back(
  [ EXCEPTION cntl_error = 1 ] ) .

<ref_viewer>->go_forward(
  [ EXCEPTION cntl_error = 1 ] ) .

<ref_viewer>->go_home(
  [ EXCEPTION cntl_error = 1 ] ) .

<ref_viewer>->stop(
  [ EXCEPTION cntl_error = 1 ] ) .

<ref_viewer>->do_refresh(
  [ EXCEPTION cntl_error = 1 ] ) .
```

Methoden zum Navigieren (zurück, vorwärts, Startseite, stoppen, wiederholen).

```
<ref_viewer>->close_document(
  [ EXCEPTION cntl_error = 1 ] ) .
```

Schließt das Dokument.

```
<ref_viewer>->set_document_charset(
    exporting
       html_charset      = <charset>
  [ EXCEPTION cntl_error = 1 ] ) .
```

Setzt den Zeichensatz für das Dokument.

Kalender Control

Das SAP Kalender Control bietet eine einfache und intuitive Oberfläche, mit der ein Datum oder ein Datumsintervall eingegeben und angezeigt werden kann.

Klasse: `cl_gui_calender`

ref_calender ist eine Instanzvariable vom Typ dieser Klasse.

Instanziierung:

```
CREATE OBJECT <ref_calender>
    EXPORTING
      parent      = <ref_container>
      view_style  = <i_style>
    [ focus_date  = <datum>  ]
    [year_begin   =  <jahr>  ]
    [year_end     =  <jahr>  ]
    [week_end     =          ]
    [ name        =  <name>  ]
  [ EXCEPTIONS <Liste mit Ausnahmen> ]  .
```

Methoden:

```
<ref_calender>->get_selection(
    IMPORTING
      date_begin          =  <datum>
    [ date_end            =  <datum>    ]
    [ selection_table     =  <tabelle>  ]
  [ EXCEPTIONS cntl_error = 1 ] ) .
```

Gibt das selektierte Datum bzw. Datumsintervall zurück..

```
<ref_calender>->set_selection(
    EXPORTING
      date_begin           =  <datum>
    [ date_end             =  <datum>     ]
    [ selection_table      =  <tabelle>   ]
    [ no_scroll            =  {'X' | ' '} ]
  [ EXCEPTIONS cntl_error  =   1 ] ) .
```

Selektiert das Datum bzw. Datumsintervall.

```
<ref_calender>->set_today(
    EXPORTING
      today               =  <datum>
  [ EXCEPTIONS cntl_error = 1  ] ) .
```

Setzt / selektiert das aktuelle Datum.

Events: `DATE_SELECTED , F2 , F12`

Ein Datum wurde selektiert, F2 bzw. F12 wurde gedrückt..

ALV Grid Control

Das ALV Grid Control (ALV = **S**A**P** **L**ist **V**iewer) ist ein flexibles Werkzeug zur Listendarstellung. Es bietet typische Listenoperationen als generische Funktionen an und ist um eigene Funktionen erweiterbar. Das ALV Grid Control kennt über 300 Methoden und ist sehr komplex. Zur Darstellung einer einfachen Tabelle, dessen Struktur im Data Dictionary hinterlegt ist, genügt nachfolgende Methode mit wenigen Parameter. Das ALV Objektmodel bietet hier eine wesentlich einfachere Möglichkeit, tabellenartige Darstellungen von Daten zu realisieren.

Hat die darzustellende Tabelle keinen direkten Bezug zum Data Dictionary, müssen die Spalten der Tabelle über einen Feldkatalog beschrieben werden.

Klasse: `cl_gui_alv_grid`

ref_grid ist eine Instanzvariable vom Typ dieser Klasse.

Instanziierung:

```
CREATE OBJECT <ref_grid>
    EXPORTING
    [ i_shelltype   =  0  ]
      i_parent      = <ref_container>
    [ i_appl_events = <false>   ]
  [ EXCEPTIONS <Liste mit Ausnahmen> ]  .
```

Methoden:

```
<ref_grid>->set_table_for_first_display(
    EXPORTING
       i_structure_name  =  '<Typ im Data Dict.>'
    CHANGING
       it_outtab     =  <interne Datentabelle> ) .
```

Daten in das Grid laden. Falls die Datenstruktur im Diktionär hinterlegt ist, müssen für eine Standard-Funktionalität nur der Parameter **`it_outtab`** und **`i_structure_name`** angegeben werden.

```
<ref_grid>->get_selected_columns(
    IMPORTING et_index_columns  =  <index>   ) .
```

```
<ref_grid>->get_selected_rows(
    IMPORTING et_index_rows   =  <index>
            [ et_row_no       =  <nr> ] ) .
```

Ausgewählte Spalten, Zeilen holen.

9.6 ALV Objektmodell

Das ALV Objektmodell ist eine Verschalung (objektorientierte Verpackung) des ALV Grid und des Tree Controls. Damit lassen sich sehr einfach tabellarische oder hierarchische Listen auf einem Dynpro erzeugen. Die aufwendige Beschreibung der Listen mit Hilfe eines Feldkataloges entfällt und auch das explizite definieren eines Dynpros ist nicht notwendig. Die Klasse **`cl_salv_table`** beschreibt einfache, zweidim. Tabelle, mit **`cl_salv_hierseq_table`** werden hierarchisch-seq. Listen und mit **`cl_salv_tree`** Baumstrukturen beschrieben.

Beschrieben werden nur einige Möglichkeiten der einfachen Tabelle. Weitere benutzte Klassen sind: **`cl_salv_functions`** (Funktionsleiste), **`cl_salv_display_settings`** (Anzeigeeigenschaften), **`cl_salv_selections`** (Selektionsoptionen), **`cl_salv_events_table`** mit Oberklasse **`cl_salv_events`** (Event Handling).

Klasse cl_salv_table

Instanziierung:

```
cl_salv_table=>factory(
  [ EXPORTING [ r_container  = <ref_container> ]
              [ list_display = {'X' | ' '} ]]
    IMPORTING   r_salv_table = <ref_table>
    CHANGING    t_table      = <itab> ) .
```

Die Instanziierung erfolgt mit einer Factory-Methode, in der Variablen ***ref_table*** wird eine Instanz dieser Klasse zurückgegeben. Gleichzeitig wird auch die darzustellende Tabelle mit übergeben. ***itab*** ist eine interne Tabelle mit den Daten für das Grid.

Methoden:

```
<ref_table>->set_data(
    changing t_table = <itab>  ) .
```

Interne Tabelle mit den Daten für die zweidim. Tabelle setzen.

```
<ref_table>->set_screen_status(
    EXPORTING  pfstatus     = '<Statusname>' ]
               report       = SY-REPID
             [ set_funcions = <mode> ] .
```

Setzt den GUI-Staus für das Dynpro (nicht Einsetzbar für Grids in einem Container). Werte für *mode:* **0** für keine Drucktasten, **1** für eine Standardauswahl und **2** für alle Funktionstasten. Den Standard GUI-Status **SALV_TABLE_STANDARD** findet man in der Funktionsgruppe **SALV_METADATA_STATUS**.

```
<ref_table>->display( ) .
```

Daten im Dynpro / Dynpro-Container anzeigen.

```
<ref_func> = <ref_table>->get_functions( )
<ref_func>->set_all( [ {'X' | ' '} ] )
```

Instanz der Funktionsleiste holen und alle Funktionen aktivieren. Anstelle eines **'X'** bzw. **' '** für **true** bzw. **false** können auch die ABAP-Konstanten **abap_true** bzw. **abap_false** benutzt werden. Der Parameter für die Methode **set_all** ist optional, wird kein Wert übergeben wird `true` angenommen.

```
<ref_display> = <ref_table>->get_display_settings( )
<ref_display>->set_striped_pattern( {'X' | ' '} )
<ref_display>->set_list_header( '<Neue Überschrift>' )
```

Instanz des Objekts für die Anzeigeeigenschaft holen.
Die Methode **set_list_header** setzt eine neue Überschrift für das Dynpro, **set_striped_pattern** schaltet das Streifenmuster ein bzw. aus. Beim Steifenmuster werden die Zeilen des Grid im Wechsel heller bzw. dunkler dargestellt.

```
<ref_select> = <ref_table>->get_selections( )
<ref_select>->set_set_selection_mode( [<mode>] )
<itab_row> = <ref_select>->get_selected_rows (  )
<itab_col> = <ref_select>->get_selected_columns (  )
```

Instanz für die Selektionsoptionen holen. Werte für *mode:* **0** für keine (Standard-wert), **1** für einfache und **2** für Mehrfachselektion.

itab_row (TYPE *salv_t_row*), *itab_col* (TYPE *salv_t_column*) sind interne Tabellen mit den selektierten Zeilennummern bzw. Inhalten der selektierten Spalten.

Events:

```
ADDED_FUNCTION , LINK_CLICK , DOUPLE_CLICK
<ref_events> = <ref_table>->get_event( )
```

Liefert alle Events als Objekt der Klasse **cl_salv_events_table**. Die Events sind in der Oberklasse **cl_salv_events** implementiert.

9.7 Object Services

Die Object Services enthalten zentrale Dienste, die nicht durch Sprachelemente von ABAP Object abgedeckt werden. Dazu zählen der **Persistenz-** und **Query-Dienst** für persistente Objekt und der **Transaktionsdienst** für den Verbuchungsvorgang. Voraussetzung für die Nutzung der Dienste ist, dass alle Datenbanktabellen bis zu den Datenelementen vollständig im Data Dictonary spezifiziert sind. Weiterhin müssen alle Klassen für die persistenten Objekte mit dem Class Builder angelegt werden. Die Klassennamen für die Datenbankobjekte beginnen alle mit **CL_** bzw **ZCL_**, die zugehörigen Agentenklassen mit **CA_** bzw **ZCA_** und dessen abstrakte Oberklasse mit **CB_** bzw. **ZCB_**.

Für die Datenbanktabellen existieren für alle Attribute **set**- und **get**-Methoden mit dem Namen der Attribute und dem Präfix **SET_** bzw. **GET_**. Analog existieren für die Schnittstellen formale Parameter mit dem Namen ***i_***<***Attributname***> . Für **<*DB_tabelle*>** steht in der nachfolgenden Beschreibung eine im Data Dictionary angelegte Datenbanktabelle.

Klassen / Typen:

```
zcl_<DB_tabelle>      " Klasse der DB_Tabelle
zca_<DB_tabelle>      " Klasse des zugehörigen Agenten
zcb_<DB_tabelle>      " Oberklasse des Agenten

if_os_query_manager " Objekttyp des Query-Managers
if_os_query           " Objekttyp der Query

<db_tabelle>          " DB-Tabelle im Data Dictionary

osreftab              " Typ im DD (internen Tabelle) für
                      " die Ergebnismenge
```

Agent instanziieren:

```
<ref_agent> = zca_<DB_tabelle>=>agent .
```

Erzeugt die Instanz des Klassenakteurs (Agenten). Die Instanz wird in der statischen Variablen **agent** der KLasse ***zca_***<***DB_tabelle***> gehalten.

Persistenz-Dienst

Objekt lesen:

```
<ref_obj> = <ref_agent>->get_persistent(
                              i_<attribut> = <key_value> ) .
```

Daten eines persistenten Objekts aus der DB-Tabelle mit dem Key ***key_value*** einlesen und als Objekt der Klasse ***zcl_<DB_tabelle>*** zurückgeben.

Objekt löschen:

```
<ref_agent>->delete_persistent(
                    .i_<attrib> = <key_value>).
```

Daten eines persistenten Objekts löschen. Nach **COMMIT WORK** wird der Löschprozess angestoßen und der Datensatz aus der DB-Tabelle mit dem Key ***key_value*** gelöscht. Ist ein Löschen nicht möglich wird eine Exception geworfen.

Objekt erzeugen:

```
<ref_obj> = <ref_agent>->create_persistent(
                  i_<attribut> = <key_value>
                [i_<attribut> = attrib_value ]... ) .
```

Persistentes Objekt der Klasse ***zcl_<DB_tabelle>*** erzeugen. Der formale Parameter **i_**<***attribut***> steht für ein beliebiges Nichtschlüssel-Attribut des Objekt, für jedes Attribut muss ein entsprechender weiterer formaler Parameter mit Werten übergeben werden. Nach **COMMIT WORK** wird der Buchungsprozess angestoßen und der Datensatz in die Datenbank mit dem Key ***key_value*** geschrieben. Ist ein Schreiben nicht möglich wird eine Exception geworfen.

Query-Dienst

Query-Manager:

```
<ref_q_manager> = cl_os_system=>get_query_manager( ).
```

Instanz des Query Managers mit Hilfe der Dienstklasse **cl_os_system** und dessen Factory-Methode erzeugen. Typ des Query Managers ist das Interface **if_os_query_manager.**

Query:

```
<ref_query> = <ref_q_manager>->create_query(
            i_filter = '<attribut> = PAR1' ) .
```

Query-Instanz mit Hilfe des Query_Managers als Typ des Interfaces **if_os_query** erstellen. Übergeben werden die Parameter für den Filter: <***attribut***> ist der Spaltenname der DB-Tabelle (siehe DD), für die der Filter ausgeführt wird. An den Parameter **PAR1** wird der Wert gebunden, der bei Ausführung der Abfrage übergeben wird.

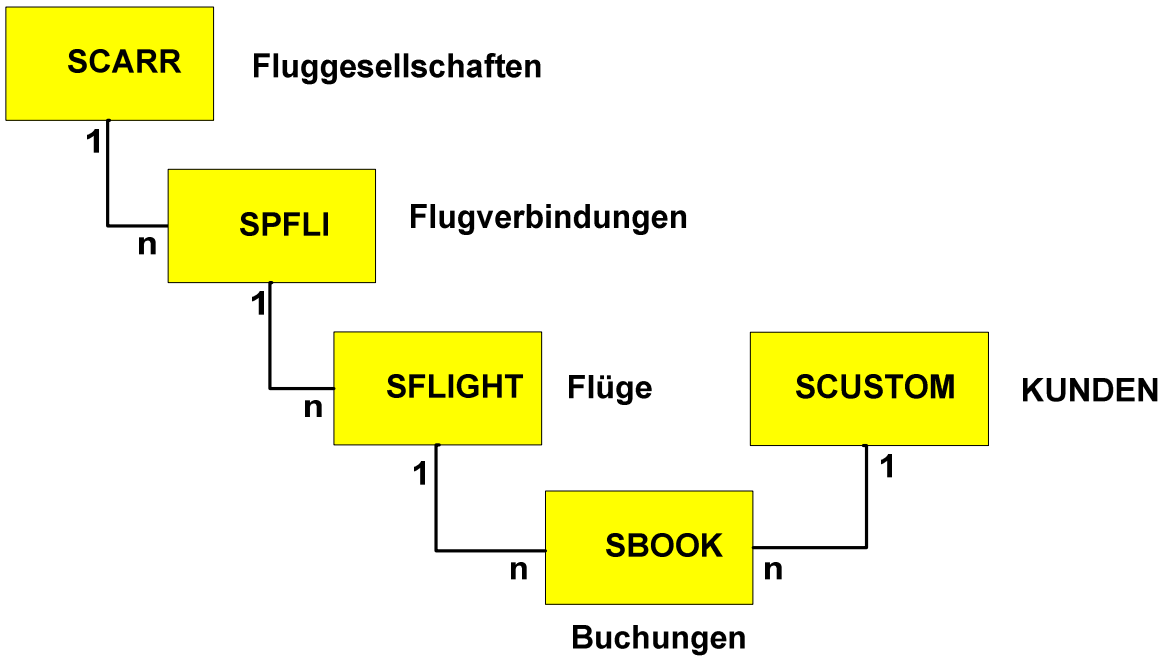
Query ausführen:

```
<obj_itab> = <ref_agent>->
      if_os_ca_persistency~get_persitent_by_query(
                i_query = <ref_query>
                i_par1  = <attribut_value> ) .
```

Der Methode wird das Query-Objekt und der Wert für den Filter übergeben (siehe auch **PAR1**, Methode **create_query**). Das Ergebnis der Abfrage ist eine interne Tabelle vom Typ ***osreftab*** mit den Referenzen der gelesenen Objekte. Mit der **LOOP-**Anweisung werden alle Referenzen aus der internen Tabelle gelesen, mit Hilfe der **get-**Methoden der Objekte kann dann auf deren Werte zugegriffen werden.

10 Das SAP-Flugdatenmodell

Das SAP-Flugdatenmodell wird in den SAP-Schulungen und in der ABAP-Dokumentation zu Demonstrations- und Übungszwecken benutzt. Nachfolgend werden die am häufigsten benutzen Tabellen beschrieben:



SCARR = (**CARRID**, *CARRNAME*, *CURRCODE*, *URL*)

SPFLI = (**CARRID**, **CONNID**, *COUNTRYFR*, *CITYFROM*, *AIRFROM*, *COUNTRYTO*, *CITYTO*, *AIRTO*, *FLTIME*, *DEPTIME*, *ARRTIME*, *DISTANCE*, *DISTID*, *FLTYPE*, *PERIOD*)

SFLIGHT = (**CARRID**, **CONNID**, **FLDATE**, *PRICE*, *CURRENCY*, *PLANETYPE*, *SEATSMAX*, *SEATSOCC*, *PAYMENTSUM*, *SEATSMAX_B*, *SEATSOCC_B*, *SEATSMAX_F*, *SEATSOCC_F*)

SBOOK = (**CARRID**, **CONNID**, **FLDATE**, **BOOKID**, *CUSTOMID*, *CUSTTYPE*, *SMOKER*, *LUGGWEIGHT*, *WUNIT*, *INVOICE*, *CLASS*, *FORCURAM*, *FORCURKEY*, *LOCCURAM*, *LOCCURKEY*, *ORDER_DATE*, *COUNTER*, *AGENCYNUM*, *CANCELLED*, *RESERVED*, *PASSNAME*, *PASSFORM*, *PASSBIRTH*)

SCUSTOM = (**ID**, *NAME*, *FORM*, *STREET*, *POSTBOX*, *POSTCODE*, *CITY*, *COUNTRY*, *REGION*, *TELEPHONE*, *CUSTTYPE*, *DISCOUNT*, *LANGU*, *EMAIL*, *WEBUSER*)