ABAP-Programmierung

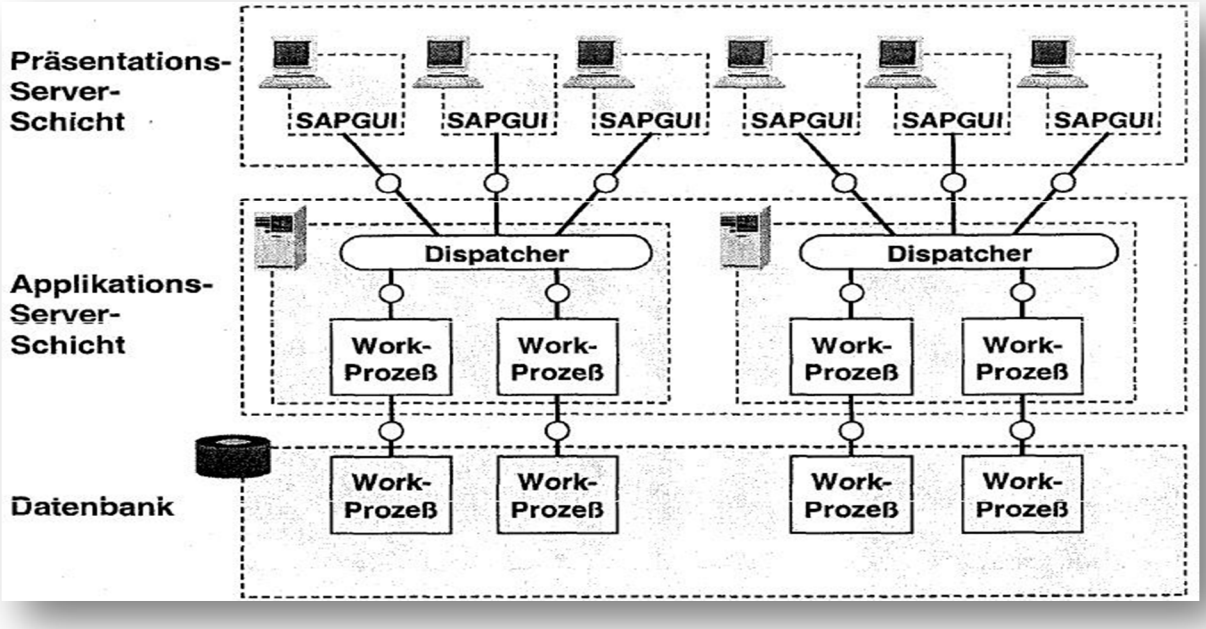
Prof. Dr. Peter Hohmann

INHALTSVERZEICHNIS

1. DAS SAP-System

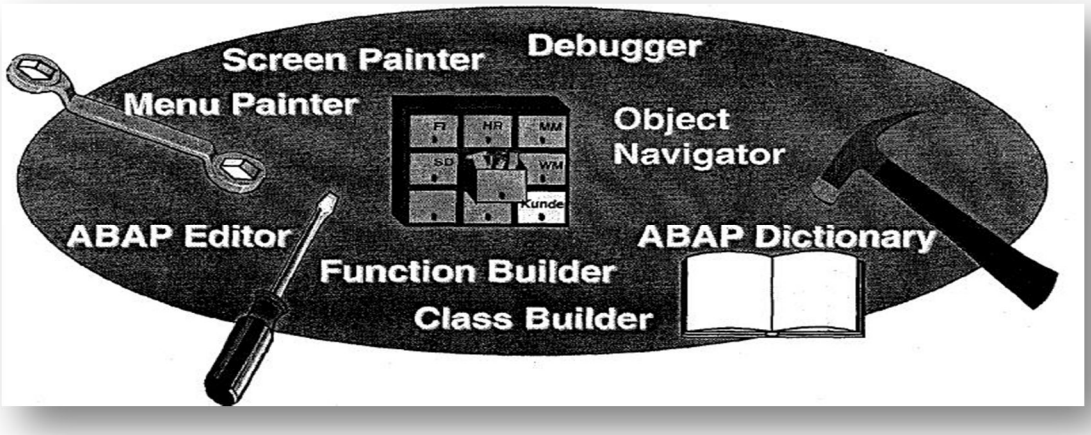
1.1. Client-Server-Architektur von SAP

SAP verfügt über eine Client-Server-Architektur. Die Präsentationsschicht wird von dem sogenannten SAP-GUI, welches auf dem Arbeitsplatz-PC installiert ist, realisiert. Über das GUI sind Zugriffe über Intranet und/oder Extranet möglich. Die Applikationsschicht / Verarbeitungsschicht wird vom Dispatcher verwaltet. Der Dispatcher ist ein SAP-Systemprogramm, das die Abläufe, die Reihenfolge der Bearbeitung und die Koordination der SAP-Workprozesse (Dialogprozesse, Verarbeitungsprozesse, Batchprozesse und Spoolprozesse) erledigt. Die Datenbankschicht wird im SAP-System von Datenbankprogrammen wie SAP-DB, ORACLE, INFORMIX, MAXDB, SAP-DB, ADABAS oder MS-SQL-SERVER realisiert.



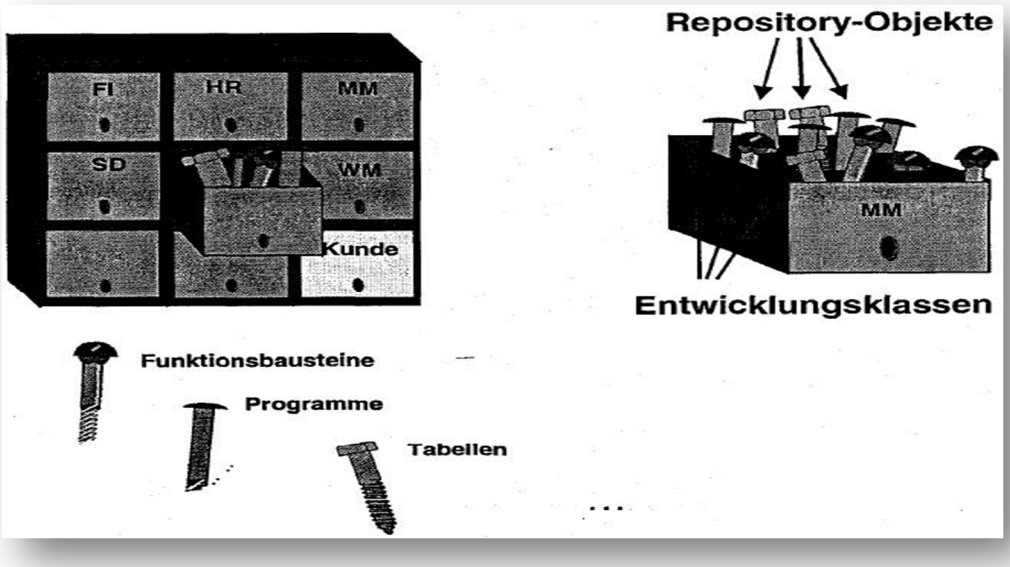
1.2 Werkzeuge der ABAP Workbench

Die ABAP-Workbench ist die Entwicklungsumgebung des SAP-Systems und besteht aus folgenden wichtigen Komponenten:



1.3. Repository /Data Dictionary (DD)

ABAP-Programme basieren auf einem Repository (R) und Data Dictionary (DD). Das Repository (Datenbank mit Entwicklerinformationen) ist ein leistungsfähiges Hilfsmittel von ABAP und speichert alle wichtigen Komponenten eines ABAP-Programms wie Sourcecodeprogramme, Bildschirmmasken, Menüeinstellungen, Sprachübersetzungen, Listendefinitionen, Funktionsbausteine, Includes usw.. Zur Strukturierung im Repository werden sogenannte Pakete verwendet. Pakete (Entwicklungsklassen) sind logische Begriffe unter denen Programme usw. wieder auffindbar sind.

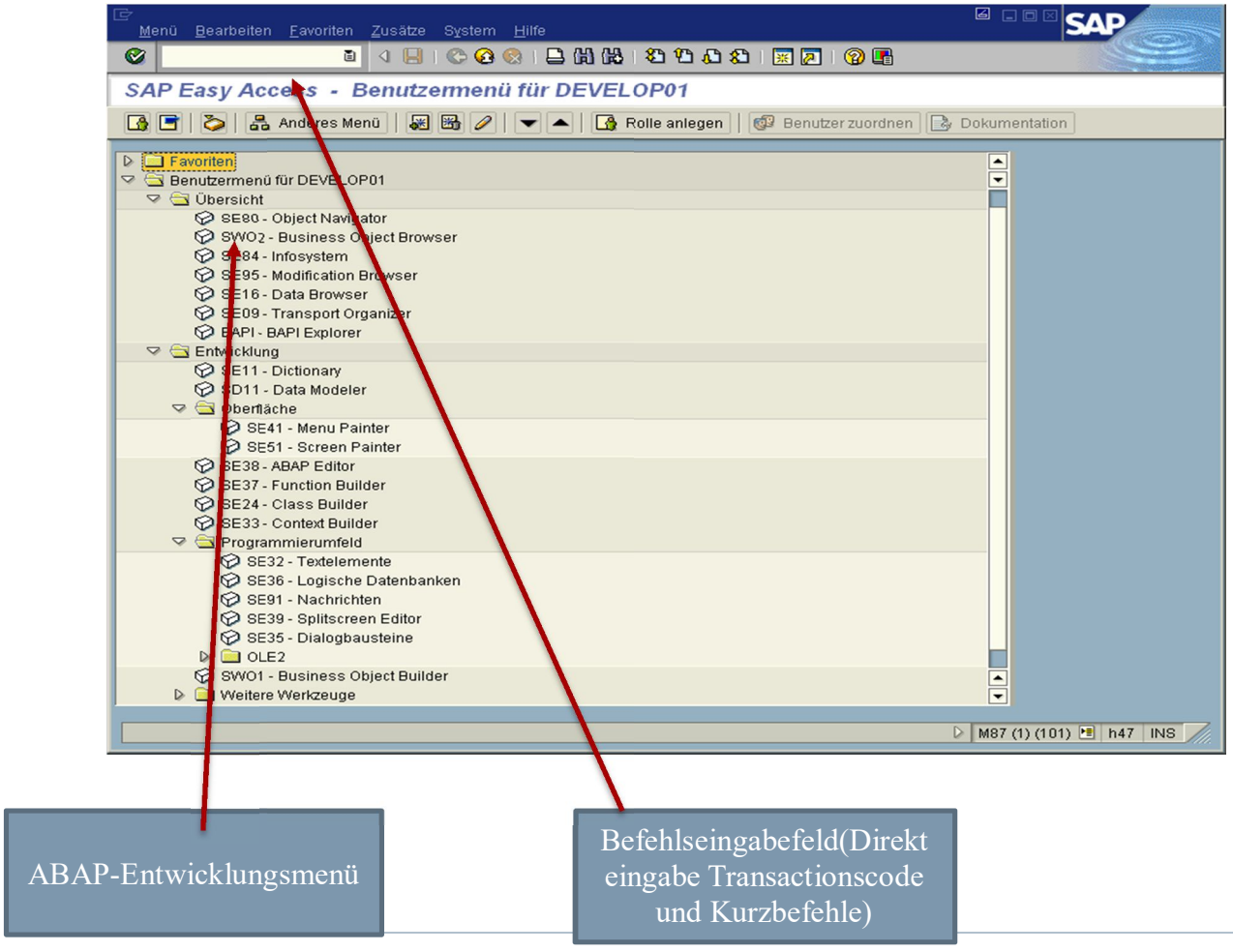


SAP-Abbildungen
Das Repository wird vom Data Dictionary ergänzt, welches den Datenkatalog des ABAP-Systems enthält. Im Data Dictionary werden Datenstrukturen, Tabellenbeschreibungen, Feldeigenschaften und Schlüsselverknüpfungen abgelegt.

1.4 Transaktionen

Transaktionen in der SAP-Begriffswelt sind ablauffähige Programme, die über einen Kurzcode aufgerufen werden können. Wichtige Transaktionscodes für die ABAP-Programmierung sind z.B.

- **SE80** = Objekt Navigator (*dies ist das wichtigste Programm für Sie*)
- **SE11** = ABAP-Dictionary
- **SE37** = Function Builder
- **SE24** = Class Builder
- **SE38** = Editor
- **SE04** = User Übersicht d.h. Prozess-Liste
- **SE16** = Data Browser
- **ST05** = SQL-Trace
- **SW01** = BAPIs
- **CMOD** = User Exits



Kurzbefehle im Transaktionsfeld

Die Eingabe folgender Kürzel in das Befehlseingabefeld bewirkt die Auslösung der folgenden Funktionen:

/o = Neuer Modus /o oder neuer Modus mit einem bestimmten Programm starten z.B. /ose80

/n = Aktuelles Programm abbrechen /n

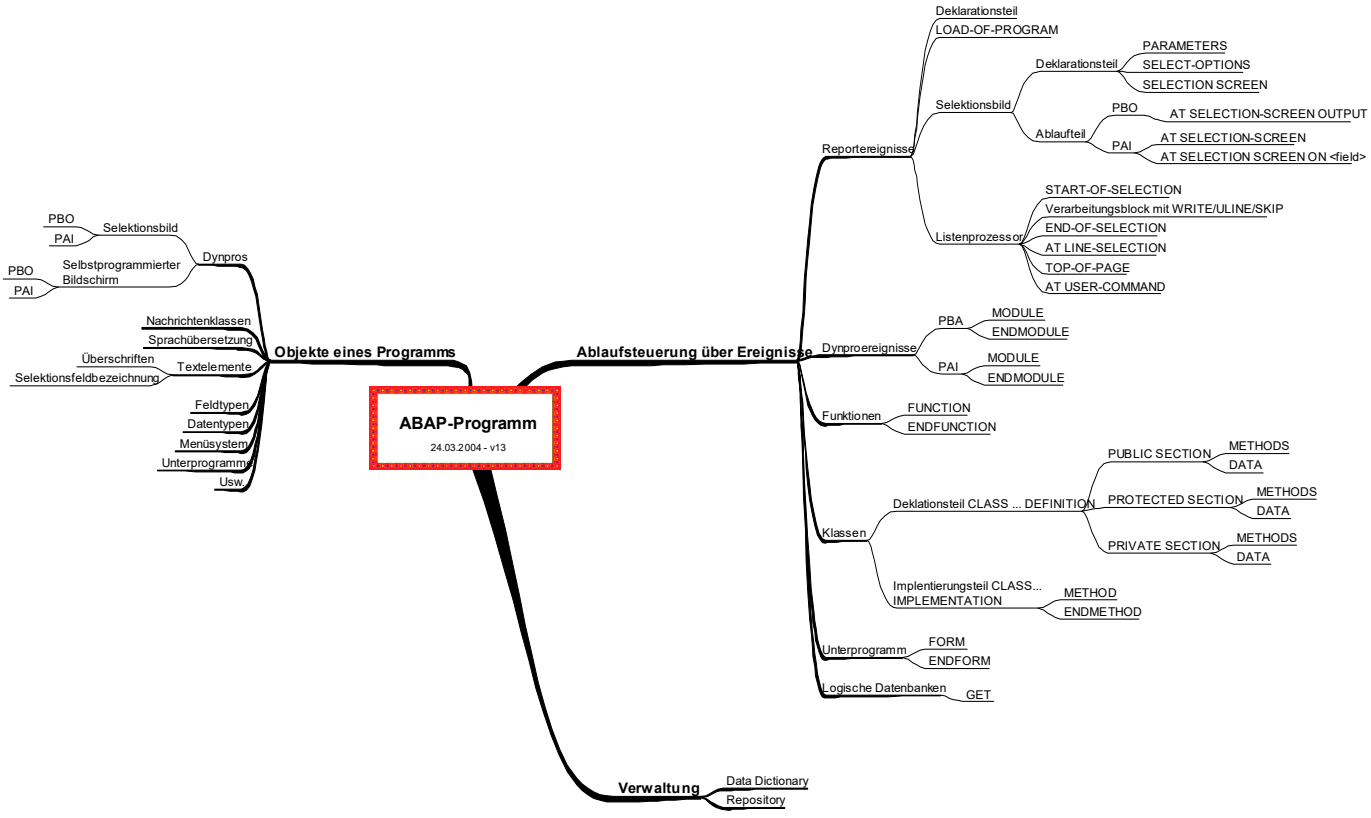
/h = Debugging starten. Alternativ aus dem Menü System-Hilfsmittel –Debugging (Dynpro oder ABAP)

1.5 Transaktionen die von Bedeutung sind

- **SU01 -** Benutzerverwaltung, hier kann man z.B. wunderbar (sobald man dazu aufgefordert wird sein Kennwort bei der Anmeldung zu ändern) das Kennwort zurücksetzen bzw. zurücksetzen lassen, z.B. auch fünfmal hintereinander. Hier werden aber auch alle Berechtigungen (das beliebte SAP_ALL und SAP_NEW) vergeben.
- **SM04 -** falls man sich selber ins Abseits gestellt hat, kann man hier die noch offenen Modi 'abschießen'
- **ST22 -** Shortdumpanalyse, dummererweise ist einer der ersten Texte in der Shortdumpanalyse 'Was können Sie tun' - 'Bitte drucken Sie diesen Text aus'. Wieviele Wälder sind wohl für diesen bekloppten Satz schon abgeholzt worden. Hier kann man sich in aller Ruhe im Nachhinein (auch vom Vortag) den kompletten Shortdump auf den Bildschirm holen.
- **SE09 -** auch ST10, Einstieg ins 'Transportwesen'. Die eigenen - aber auch die vom Kollegen - offenen und abgeschlossenen Transportaufträge lassen sich anzeigen.
- **SM36 -** Jobs anlegen
- **SM37 -** Jobübersicht, z.B. abgebrochene Jobs checken
- **SM59 -** RFC-Destinationen (Pflege auch aller TCP/IP-Verbindungen)
- USMM - Systemvermessung
- **SM50 -** Prozessübersicht (z.B. wenn irgendwas allzu lange dauert kann man das hier 'sehen' und auch abbrechen über den Menübefehl 'Prozeß' - 'Abbrechen mit Core', u.U. zweimal wiederholen. Wenn das nicht klappt '...ohne Core')
- **SM35 -** Jobverarbeitung, Mappen freigeben etc.
- **STMS -** Transport Management System (Transporte importieren etc.)
- **SE01 -** Transportaufträge nach Nummer
- **SE10 -** Customizing-Aufträge (auch über andere Transaktionen aufrufbar)
- **ST01 -** System Trace, hier können u.a. die Berechtigungsprüfungen getraced werden

2. ABAP die Programmiersprache

2.1 Allgemeiner Überblick



2.1.1 Eigenschaften der ABAP-Programmiersprache

- ABAP/4 ist eine interpretative Sprache der 4. Generation.
- Alle Metadaten werden im aktiven ABAP/4-Dictionary/Repository abgelegt.
- ABAP/4 ist mehrsprachenfähig. Textelemente wie Titel, Überschriften etc. werden getrennt vom Programmcode gespeichert.
- Die Sprache ist ereignisorientiert, d.h. die einzelnen Abschnitte eines Programms definieren Reaktionen auf Ereignisse.
- Betriebswirtschaftliche Datentypen und Operationen (z.B. rechnen mit Datumsfeldern) werden besonders unterstützt.
- ABAP/4 enthält ein SQL-Subset für die DBMS-unabhängige Durchführung von Datenbankoperationen.

2.1.2 Mehrsprachigkeit von ABAP

- Alle Texte, die auf dem Bildschirm und in Ausdrücken verwendet werden, können außerhalb des Programmcodes als sprachabhängige Texte abgelegt werden.
- Diese Texte können dann vom Entwickler in verschiedene Sprachen übersetzt werden.
- Zur Laufzeit wird der Text dann in der durch den Benutzer bei der Anmeldung angegebenen Sprache angezeigt. Ist die entsprechende Sprache nicht vorhanden, wird der Text in „Originalsprache“ gezeigt.
- Generell sollten in Programmen keine hardcodierten Texte, sondern immer Textsymbole verwendet werden.

2.1.3 Ausführung eines ABAP-Programms

- Ein ABAP/4-Programm wird zur Laufzeit in einen ausführbaren Code kompiliert. Dies bezeichnet man als **Generierung**. Die generierte Form wird dann im ABAP/4-Repository gespeichert.
- Ein generiertes Programm wird neu generiert, sobald sich der Programmcode oder eines der referenzierten Dictionary-Objekte geändert hat.
- Die Notwendigkeit zur Neugenerierung wird vom Laufzeitsystem automatisch erkannt. Der Systementwickler muss sich nicht darum kümmern.

2.1.4 Syntax von ABAP-Programmen

Allgemeine Syntax

- ABAP/4-Anweisungen enden immer mit einem Punkt.
- Eine Anweisung kann sich über mehrere Zeilen erstrecken.
- Anweisungen können durch Zusätze qualifiziert werden.
- Ein Wort wird auf beiden Seiten durch ein Leerzeichen begrenzt (auch z.B. unmittelbar auf eine Klammer folgend).
- Literalzeichen werden in einfache Anführungszeichen gesetzt. Enthält das Literal ein Anführungszeichen, so ist dieses doppelt zu schreiben, z.B.
  - `WRITE 'Meier''s Lexikon'.`
- Kommentare werden durch * (in der ersten Spalte) oder “ (an beliebiger Stelle) begonnen. Der Rest der Zeile wird dann als Kommentar betrachtet.

Zeichen zur Strukturierung von Befehlen

- . = Abschluss eines Befehls. **ACHTUNG:** ABAP verlangt nach jedem Befehl einen Punkt.
- : = Der Doppelpunkt nach einem ABAP Schlüsselwort (z.B. TYPE, DATA, WRITE) erlaubt es, mehrere Befehle mit Komma getrennt hintereinander auszuführen.

```
WRITE: / feld1,  feld2,  feld3.
```

- , = Trennzeichen für Befehlsaufreihung (siehe Beispiel WRITE)
- **'müller'** = Beginn und Ende eines konstanten Wertes.
- = Verwenden Sie bei der Namensvergabe keine Bindestriche sondern Unterstriche, da ABAP die Bindestriche nutzt, um Satznamen und Feldnamen miteinander zu verbinden. z.B.

```
TYPES: BEGIN OF typ_satz,
        feld_1(10) TYPE C,
                   feld_2(10) TYPE C,
         END OF typ_satz.
DATA:   satz TYPE typ_satz.
satz-feld_1 = 'FELD1'.
satz-feld_2 = 'FELD2'.
WRITE: / satz.
```

Kommentare

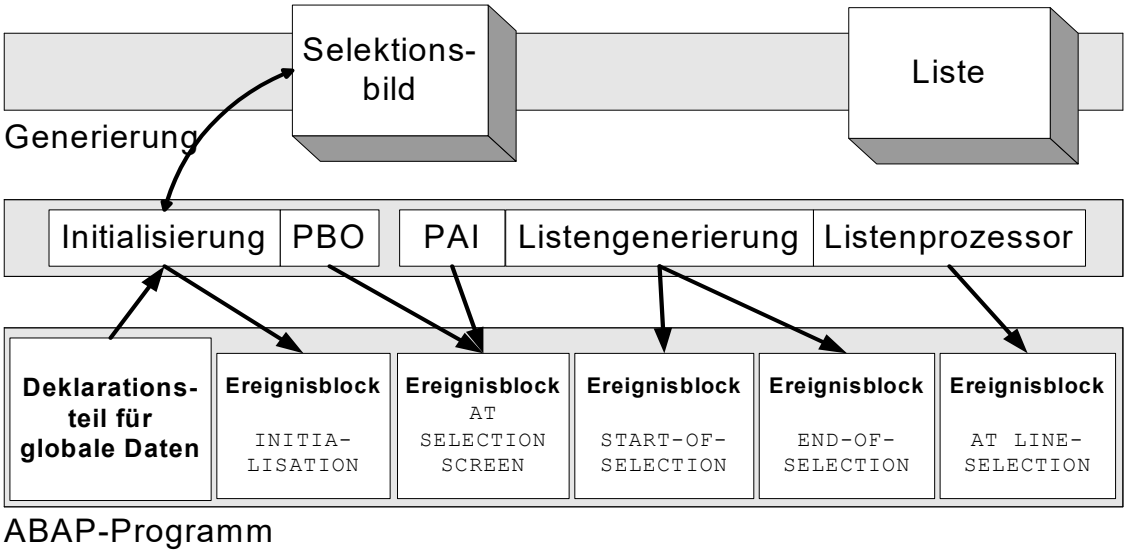
- *= eine Kommentarzeile beginnt mit einem Stern in der 1. Spalte
- " = Einleiten eines Kommentars an einer beliebigen Stelle im Sourceprogramm

2.2 Programmtypen

ABAP unterscheidet unterschiedliche Programmtypen. Diese Programmtypen sind:

Ausführbare Programme (Reports/Listenprogramm):

Ausführbare Programme werden auch als Reports bezeichnet. Das Starten von ausführbaren Programmen nennt sich auch „Reporting“. Reports können allerdings heute auch Dynpros (Bildschirmdialoge) enthalten. Die Hauptverarbeitungsblöcke eines Reports sind im Folgenden grafisch aufgezeigt. Ausführbare Programme werden mit der Anweisung REPORT eingeleitet.



ABAP-Programm

Bedeutung der einzelnen Verarbeitungsblöcke:
Initialisierungsereignis
LOAD-OF-PROGRAM = Am Beginn des Programms z.B. zur Vorbelegung von Selektionsbildern oder Variablen. Dieses Ereignis tritt genau einmal auf und zwar dann, wenn das ABAP-Programm in den Speicher geladen wird.

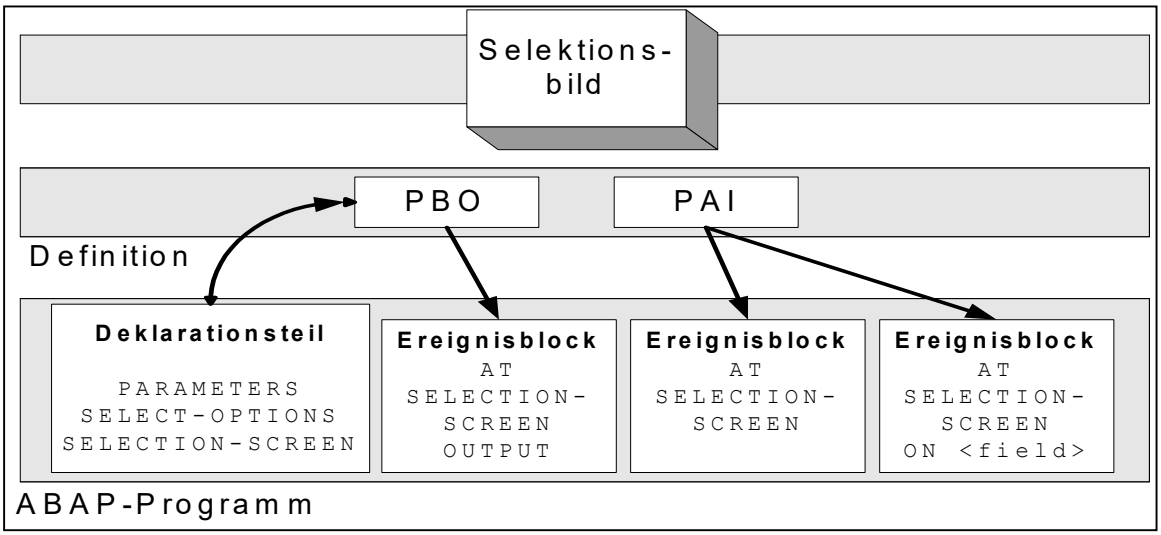
Beispiel: Vorbelegung eines Parameterfeldes

```
PARAMETERS: zw_datum TYPE sy-datum.

 LOAD-OF-PROGRAM.
zw_datum = sy-datum - 7.
START-OF-SELECTION.
WRITE: zw_datum.
```

ACHTUNG: Aus Kompatibilitätsgründen existiert alternativ noch der Ereignisblock **INITIALIZATION**. Allerdings ist er nur für ausführbare Programme verfügbar.

Ereignisse bei Verwendung von Selektionsbildern (PBO/PAI)

AT SELECTION-SCREEN = Dieses Ereignis wird im Zusammenhang mit Selektionsbildern in ausführbaren Programmen, Modulpools oder Funktionen ausgelöst. In diesem Ereignisblock können Selektionsbilder vorbereitet oder Benutzeraktionen ausgewertet werden. Dabei ist es möglich, verschiedene



Ereignisse zu bearbeiten wie das nachfolgende Bild zeigt: Die Ereignisblöcke eines **Selektionsbildes** haben folgende Bedeutung:

- **AT SELECTION-SCREEN OUTPUT** = Vor der Ausführung des Selektionsbildes (PBO) können Aktionen ausgeführt werden z.B. Vorbelegungen **MOVE** ´X` **TO** radio1**.**
- **AT SELECTION-SCREEN** = Dieser Ereignisblock wird als letztes Ereignis der Selektionsbildverarbeitung ausgelöst (PAI) z.B.

```
IF    radio1 = 'X'.
MESSAGE e001(zhoh).
ENDIF.
```

- **AT SELECTION-SCREEN ON field** = Nach der Bearbeitung des Selektionsbildes können Aktionen (PAI) ausgeführt werden. Sie können z.B. die Benutzereingabe prüfen und eventuell Fehlernachrichten ausgeben.

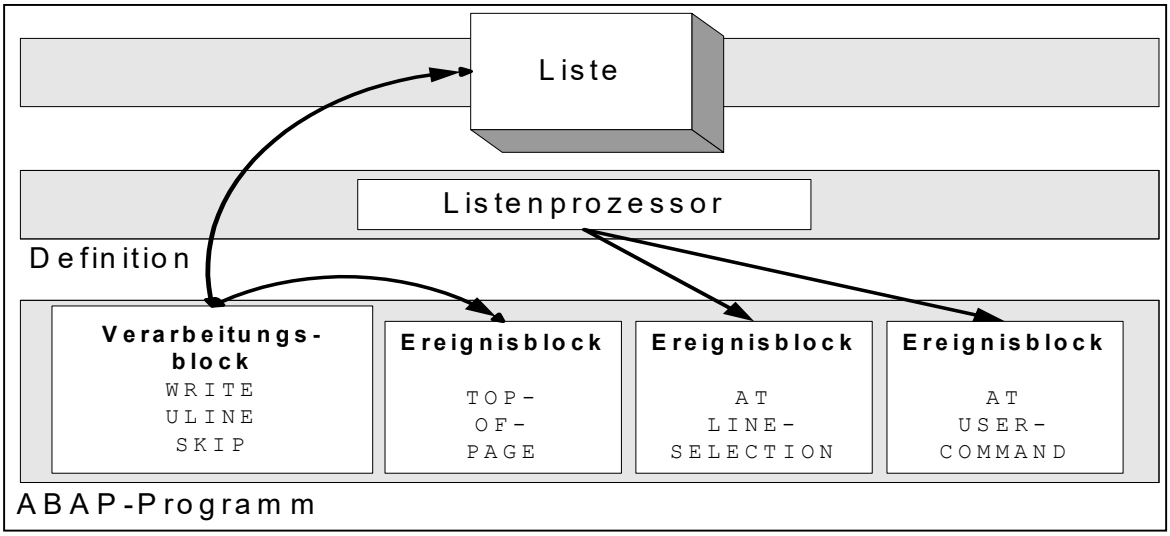
Hauptprogrammereignis der Listengenerierung

START-OF-SELECTION = Dieses Ereignis tritt bei der Ausführung eines ausführbaren Programms auf. Man kann diesen Verarbeitungsblock auch als Hauptprogrammbeginn bezeichnen. Das bedeutet, dass alle Anweisungen, die nicht explizit einem anderen Ereignisblock zugeordnet wurden, automatisch in den Ereignisblock START-OF-SELECTION zugeordnet werden.

END-OF-SELECTION = Schlussbemerkungen z.B. Ausführen von Anweisungen nach dem letzten GET-Ergebnis (logische Datenbank)

Ereignisse des Listenprozessors (Anzeige der Liste)

Eine **Liste** verfügt wiederum über eigene Ereignisblöcke. Die Ereignisblöcke einer **Liste** sind der nachfolgenden Abbildung zu entnehmen:

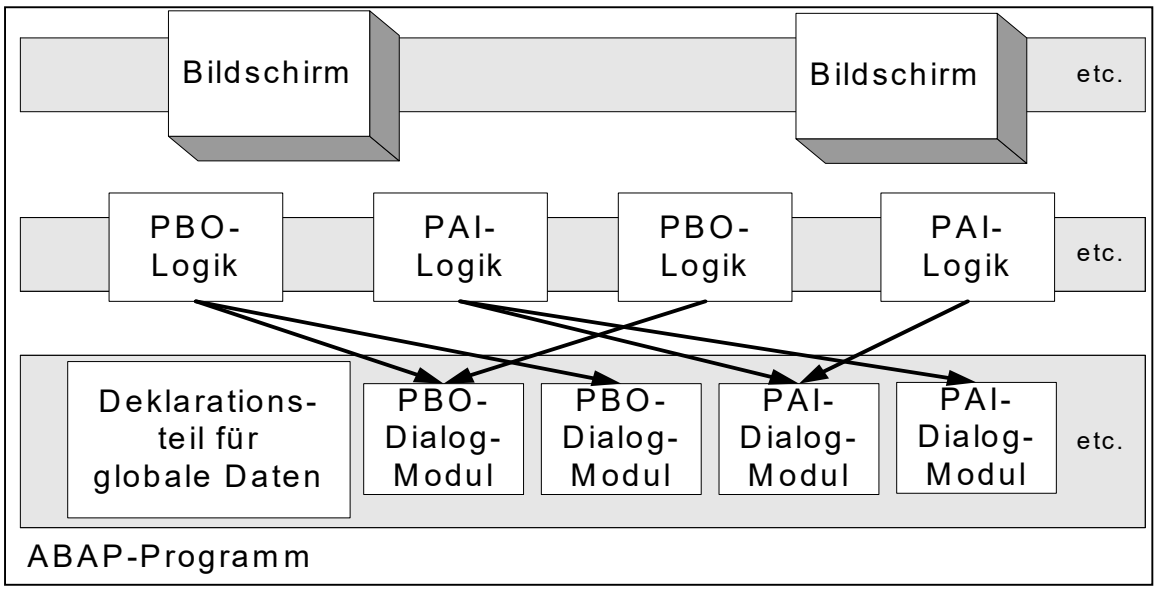


Die Bedeutung der Ereignisse sind:

- **TOP-OF-PAGE**: Listenkopf gestalten.
- **END-OF-PAGE**: Listenfuss gestalten.
- **AT LINE-SELECTION** = Sprung in diesen Bereich, wenn auf der Liste/Report ein DB-CLICK auf einer Zeile durchgeführt wurde. Die Übergabe von Variablen aus einer Liste in **AT LINE-SELECTION** erfolgt mit dem Befehl **HIDE**.
- **AT USER-COMMAND** = Reaktionsmöglichkeit auf Benutzeraktionen mit Hilfe des Systemfeldes sy-ucomm

▪ Modulpools/Dialogprogramm/Dynproprogramme (Typ M – Programm):

Diese Programme werden ausschließlich über die Ablauflogik von Dynpros gesteuert. Enthalten sämtliche Dialogmodule der zugehörigen Dynpros, werden als „Modulpool“ bezeichnet. Modulpools werden mit der Anweisung PROGRAM eingeleitet.



Dynproereignisse/Dialogmaskenereignisse

Dynproereignisblöcke werden nicht im ABAP-Programm, sondern in der Dynproablauflogik implementiert. Die Dynproablauflogik steuert die Verarbeitung von Bildschirmbildern. Dynpros werden mit dem Befehl **CALL SCREEN** #### (4stellige Nummer des Dynpros) im ABAP-Programm aufgerufen**. PBO** (Process Before Output) und **PAI** (Process After Input) sind spezielle Verarbeitungsblöcke, die vor und nach einem Dynpro aufgerufen werden, aber wie bereits erwähnt nicht im ABAP-Programm stehen sondern in Bereich Ablauflogik des Screenpainters. ABAP-Prozeduren können in der Weise aufgerufen werden, dass innerhalb von PBO und PAI ein MODUL aufgerufen wird. Dieses Modul enthält dann die notwendigen ABAP-Abfragen, Kontrollen usw. und ist im ABAP Programm eingebunden.
Die Ereignisblöcke sind demzufolge:

- **PROCESS BEFORE OUTPUT (PBO):** PBO ist der Verarbeitungsblock vor der Anzeige eines Dynpros. Die Prozeduren (ABAP-Befehle) z.B. zur Befüllung der Maske werden in den Unterroutinen mit dem Namen MODUL eingebettet. Die Syntax ist:

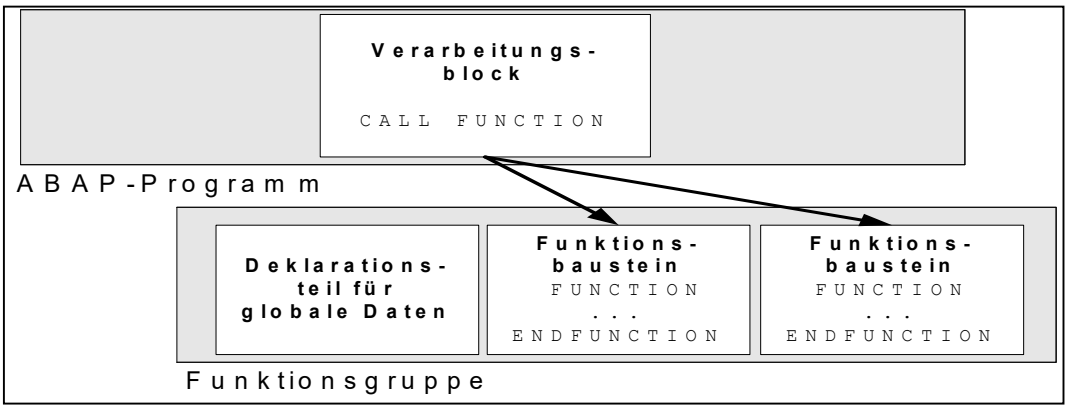
```
MODULE name OUTPUT.
    Anweisungen .
ENDMODUL .
```

- **PROCESS AFTER INPUT (PAI):** PAI ist der Verarbeitungsblock nach der Anzeige eines Dynpros. Die Prozeduren (ABAP-Befehle) z.B. zur Befüllung der Maske werden in den Unterroutinen mit dem Namen MODUL eingebettet. Die Syntax ist:

```
MODULE name INPUT.
Anweisungen .
ENDMODUL.
```

Auch der Feldbezug und damit die Auslösung eines PAI-Ereignisses ist möglich mit dem Befehl.
FIELD dynprofeld **MODULE** name **ON INPUT.** "Aufruf bei Initialwertabweichung
FIELD dynprofeld **MODULE** name **ON REQUEST.** "Bei Inhaltsveränderungen
FIELD dynprofeld **MODULE** name. "Beim Verlassen des Feldes
CHAIN ---- ENDCHAIN. Chain/Endchain definiert eine Verarbeitungskette
CHAIN-INPUT, CHAIN-REQUEST.
CHAIN.
FIELD f7, f8.
FIELD f0 MODULE m0 ON INPUT (ON REQUEST).
ENDCHAIN.

- **Funktionsbausteine (Typ F – Programm):** Dienen ausschließlich als Container für Funktionsgruppen und Funktionsbausteine. Können auch . Datendeklarationen und Unterprogramme enthalten. Der Aufruf einer Funktion erfolgt mit CALL. Die Verwaltung von Funktionsbausteinen erfolgt im FUNCTION-BUILDER. Funktionsgruppen werden mit der Anweisung FUNCTION-POOL eingeleitet.



- **KLASSENPOOLS (TYP K):** Die Grundlage einer objektorientierten Programmiersprache sind Klassen. Eine Klasse ist die Vorlage eines Objektes, wie der Datentyp die Vorlage für ein Datenobjekt ist. Seit SAP-Release 4.5 können mit ABAP objects Klassen definiert und Objekte erzeugt werden. Globale Klassen werden mit dem Class Builder der ABAP-Workbench in der Klassenbibliothek verwaltet. Lokale Klassen werden innerhalb eines beliebigen ABAP-Programms definiert und sind dort auch sichtbar. Ein Classpool könnte genau eine globale Klasse und beliebig viele lokale Klassen enthalten. Das Grundgerüst einer Klasse sieht folgendermaßen aus:

CLASS classname DEFINITION.
PUBLIC SECTION.
.....

PROTECTED SECTION.
.....
PRIVATE SECTION.
.....
ENDCLASS.
CLASS classname **IMPLEMENTATION.**
...
ENDCLASS.

Beispiel für ein Programm mit lokalen Klassen zur Kursgewinnermittlung:

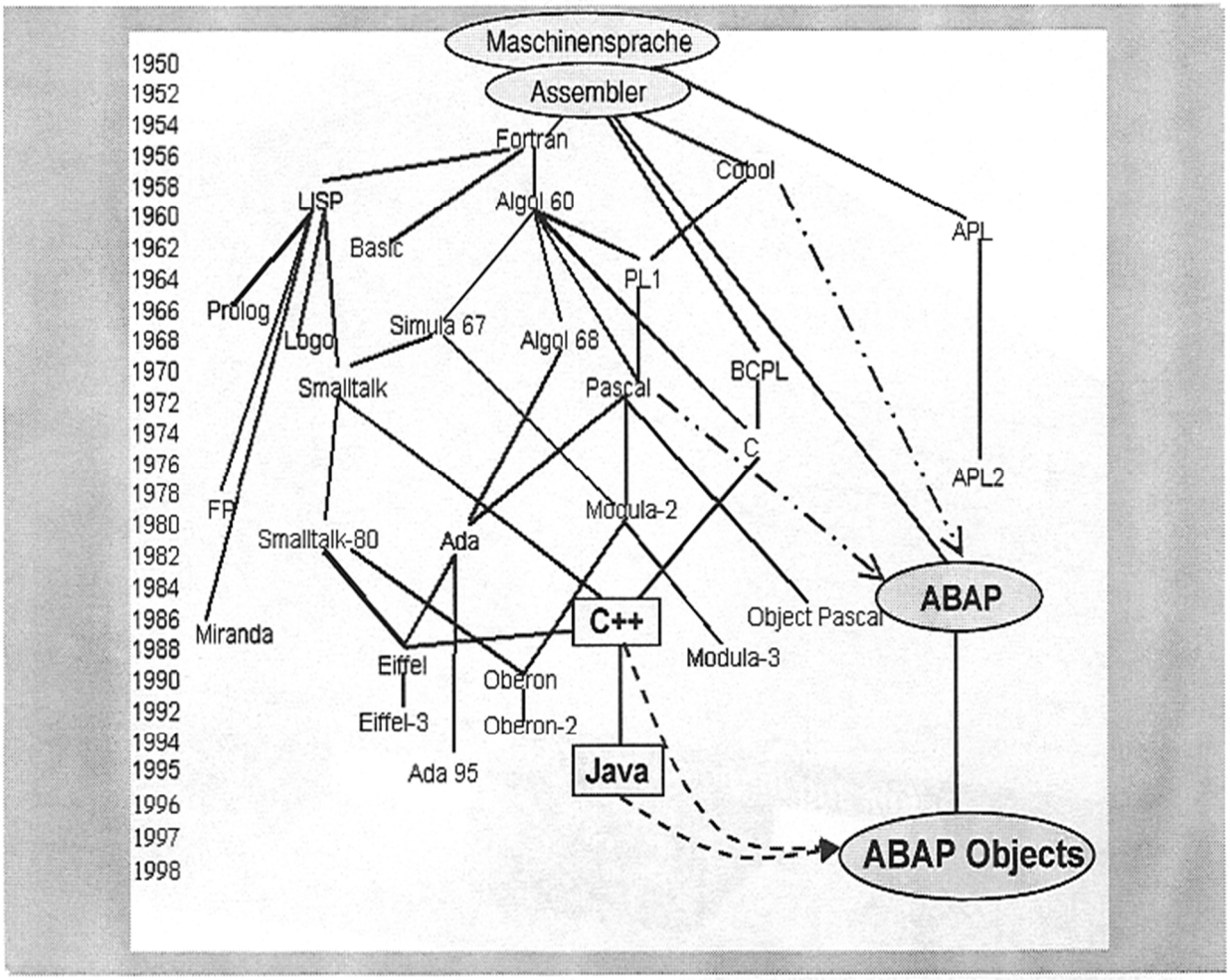
```
REPORT  z_objekte
        .
*------------Globale Variable-------------------------------
----*
TYPES:         t_zahl(12) TYPE p DECIMALS 2.
PARAMETERS:    wa_kurs TYPE t_zahl,
               wa_divi TYPE t_zahl.

DATA:          wa_kgv TYPE t_zahl.

*----- Klassendefinition-----*

CLASS berechnung DEFINITION.
PUBLIC SECTION.

 METHODS: kursgewinn   IMPORTING   l_kurs TYPE t_zahl
                                   l_divi  TYPE t_zahl
                       EXPORTING   l_kgv   TYPE t_zahl.

  PROTECTED SECTION.
  PRIVATE SECTION.
ENDCLASS.                    "berechnung DEFINITION
*---------------------------------------------------------------
---------*
*       CLASS berechnung IMPLEMENTATION
*---------------------------------------------------------------
---------*
CLASS berechnung IMPLEMENTATION.
  METHOD kursgewinn.
    COMPUTE  l_kgv = l_kurs / l_divi.
  ENDMETHOD.                    "kursgewinn

ENDCLASS.                    "berechnung IMPLEMENTATION
*---------------------------------------------------------------
---------*
*----- Referenzvariable --------------------*
*---------------------------------------------------------------
---------*
DATA: ref_berechnung TYPE REF TO berechnung.

START-OF-SELECTION.
*---------------------------------------------------------------
---------*
*-----Bilden einer Instanz  /Aufruf der Methode --------*
*---------------------------------------------------------------
---------*
  CREATE OBJECT ref_berechnung.
```

```
CALL METHOD ref_berechnung->kursgewinn
  EXPORTING
    l_kurs = wa_kurs
    l_divi = wa_divi
  IMPORTING
    l_kgv  = wa_kgv.

WRITE: / 'Ergebnis',    wa_kgv.
```

- **Interfacepools (Typ J):** Interfacepools werden mit der Anweisung INTERFACE-POOL eingeleitet. Sie enthalten die Definition eines globalen Interfaces, das in beliebigen globalen und lokalen Klassen implementiert werden kann.

- **Include-Programm (Typ I – Programm):** „Include“-Programme; dienen
- als Mittel um Programmtexte in kleine editierbare Einheiten zu gliedern.

2.3 Historie der Programmiersprache ABAP



2.4. Sprachbestandteile

2.4.1 Datentypen

Mit den Typen sind Informationen verknüpft, wie die Daten abgelegt werden, welche Wertebereiche und welche Operationen auf dem Datenobjekt erlaubt sind. Die Verwendung des Datentyps erfolgt mit dem Befehl TYPES und DATA. Typisiert können die Datentypen als solche mit Längenangaben und als solche ohne. Weiterhin haben die Datentypen entweder eine statische oder eine variable Länge.

TYP	Bezeichnung	Syntax (TYPES/DATA)
C	Charakter, alphanum. Text	feld(länge) TYPE C.
N	Numerische Zeichen	feld(länge) TYPE N.
X	Hexadezimalzahl	feld TYPE X.
P	Gepackte Zahl	feld (länge) TYPE P DECIMALS länge
I	Integer (ganze Zahl) (4Byte)	feld TYPE I.
F	Gleitkommazahl (8Byte)	feld TYPE F.
STRING/XSTRING	Beliebige Zeichen/Bytekette	feld TYPE STRING.
D	Datumsfeld (8Byte)	feld TYPE D.
T	Zeit/TIME	feld TYPE T.

Datentypen mit Längenangaben

- **C** = Charakter Anlage einer Instanz unter Verwendung von TYPES.

```
TYPES: typ_feld(10) TYPE C.
DATA: feld1 TYPE typ_feld.
DATA:  feld2 TYPE typ_feld.
```

Anlage einer Instanz unter Bezug auf ein anderes DATA-Feld.

```
DATA: feld2 LIKE feld1.
```

Anlage einer Instanz ohne Verwendung von TYPES, sondern direkt in DATA.

```
DATA: feld1(10) TYPE C.
```

- **N** = Numerisches Zeichen

```
DATA: num1(10) TYPE N.
```

- **X** = Hexadezimalzahl (Byte)
- **P** = gepackte Zahl. Hier ist zusätzlich die Anzahl der Nachkommastellen anzugeben.

```
DATA: num1(10) TYPE P DECIMALS 2.
```

Datentypen ohne Längenangaben

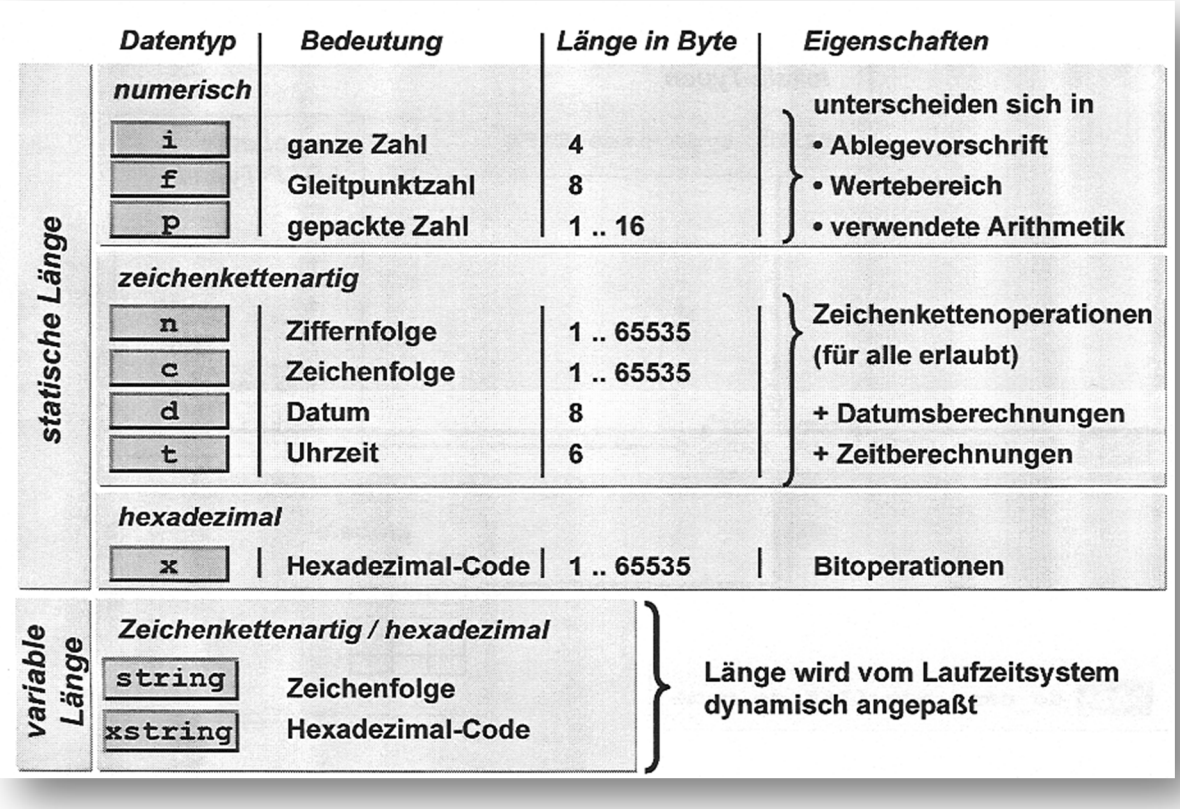
- **I** = Integer d.h. ganze Zahl
- **F** = Gleitkommazahl d.h. Floating Point
- **STRING** = Zeichenfolge Länge variabel
- **D** = Datum (Die Besonderheit am Datum in ABAP ist, dass die Möglichkeit besteht zwei Datumswerte voneinander abzuziehen, um die Differenztage zu ermitteln.) Format YYYYMMDD

- **T** = **Zeit** Format HHMMSS

ACHTUNG: In der **TYPES**-Anweisung wird ein Datentyp gebildet, in der **DATA**-Anweisung eine Instanz (realer Speicherplatz). Mit dem Zusatz **VALUE** kann ein Anfangswert gesetzt werden, fehlt ein Anfangswert, so wird die Variable typkorrekt initialisiert. Mit **LIKE** können Sie sich auf den Typ eines bereits definierten Datenobjekts (Instanz) beziehen. Mit den Anweisung **CONSTANTS** können Zahlenliterale und Textliterale erstellt werden.

```
CONSTANTS kon1(10) TYPE  C   VALUE 'ERICH'.
```

Konvertierungen zwischen den Datentypen sind teilweise möglich und im Compiler integriert.



2.4.2 Datenobjekte/Feldsymbole/Datenreferenzen

Datenobjekte müssen im Programm definiert werden und sind nur während der Ausführung des ABAP-Programms im Hauptspeicher verfügbar. Datenobjekte werden mit der Anweisung DATA erzeugt. Die Typisierung (Zuweisung eines Datentyps) eines Datenobjektes kann entweder mit Bezug auf eine vorgeschaltete TYPES-Anweisung,

einem Bezug zum Data-Dictionary oder direkt in der DATA-Anweisung erfolgen. Als typische Datenobjekte können Einzelfelder, Satzstrukturen und interne Tabellen erzeugt werden.

Erzeugen von Datenobjekten mit DATA unter Bezug auf einen Datentyp erstellt mit TYPES

Erstellen von Instanzen/Datenobjekt mit DATA unter bezug auf einen Datentyp erstellt mit TYPES. **ACHTUNG**: Es ist empfehlenswert die Instanzenbildung immer mit Bezug auf Typdeklarationen aus dem DD oder aus dem eigenen Programm (ABAP-Schlüsselwort = TYPES) anzulegen. Kommt der Datenfeldtyp aus dem DD entfällt der Befehl TYPES, da die Typdefinition durch das DD erfolgt ist. Tabellen werden unter dem Punkt „Interne Tabellen“ noch genauer betrachtet.

1.Einfaches Datenobjekt definieren:

Erstellen eines Datentyps mit TYPES und Erzeugung eines Datenobjektes mit DATA.

Allgemeine Struktur: **TYPES:** dat_typ(länge) **{TYPE** datentyp **/LIKE** objektname**}**
DATA: datenobjekt **TYPE** dat_typ **VALUE** vorbelegung**.**

Beispiel:

```
TYPES:    feldtyp1(15) TYPE C ,
          feldtyp2(10) TYPE P DECIMALS 2.

DATA:     datenfeld1 TYPE feldtyp1 VALUE 'HUGO',
          datenfeld2 TYPE feldtyp2.
```

2.Satzstrukturobjekte definieren:

Erstellen eines Satzdatentyps. Eine Struktur kann entweder manuell im jeweiligen ABAP-Programm oder im DD erzeugt werden

Allgemeine Struktur: **TYPES: BEGIN OF** strukt_typ**,**
komp **{TYPE** typ / **LIKE** objektname**},**
komp **{TYPE** typ / **LIKE** objektname**}**
END OF strukt_typ**.**
DATA: satzstruktur **TYPE** strukt_typ**.**

Beispiel:

```
TYPES:    BEGIN OF satz_struktur,
          feld1(20) TYPE C,
          feld2(10) TYPE C,
END OF satz_struktur.
DATA: struktur TYPE satz_struktur.
```

(Die Felder der Instanzstruktur können mit den Namen struktur-feld1 bzw. struktur-feld2 angesprochen werden)

3.Tabellenobjekt definieren:

Die satz_struktur kann manuell im Programm definiert sein (siehe Beispiel) oder aus dem DD kommen (siehe auch Vertiefung „Interne Tabellen“).

Allgemeine Struktur: **TYPES** tab_strukt {**TYPE/LIKE**} {**STANDARD TABLE / SORTED TABLE / HASHED TABLE**} **TABLE OF** {datenfeld / satz_struktur} **WITH** {**UNIQUE / NON-UNIQUE**} {feldname / **DEFAULT KEY**}.

DATA: tabelle **TYPE** tab_strukt.

Beispiel:

```
TYPES:   BEGIN OF satz_struktur,
  feld1(20) TYPE C,
  feld2(10) TYPE C,
END OF satz_struktur.
TYPES:   tabellen_struktur   TYPE  STANDARD TABLE OF satz_struktur.
```

dann

```
DATA: int_tabelle TYPE tabelle_struktur.
```

ACHTUNG: In R/3 sind über 20.000 verschiedene DB-Anwendungstabellen (cirka 17000) und Strukturtabellen im Data-Dictionary (DD) gespeichert. In Anwendungsprogrammen wird daher häufig Bezug auf solche im DD angelegten Datentypen, Satzstrukturtypen oder Tabellenstrukturtypen genommen. Standarddatentypen in R/3 sind z.B. CHAR1, CHAR2 usw. die man z.B. bei der Neuanlage einer Datenbanktabelle oder einer Struktur für ein Dynpro verwenden kann und muss. Solche Datentypen bilden sich elementar aus den oben genannten Datentypen und sind als Datentypen im DD abgespeichert.

Direkte Erzeugung von Datenobjekten mit DATA

DATA = DATA erlaubt mit Hilfe der Datentypen eine direkte typgerechte Instanzierung eines Datenfeldes, einer Datenstruktur oder einer Tabelle.

1.Einfaches Datenobjekt erzeugen:

Allgemeine Struktur: **DATA:** datenobjekt(länge) {**TYPE** datentyp /**LIKE** objektname} **VALUE** vorbelegung.

DATA: feld1(20) **TYPE C,**
feld2(10) **TYPE C.**

2. Satzstrukturobjekte erzeugen:

Allgemeine Struktur:

DATA: **BEGIN OF** datenobjekt**,**
komp {**TYPE** typ / **LIKE** objektname}**,**
komp {**TYPE** typ / **LIKE** objektname}
END OF datenobjekt**.**

DATA: **BEGIN OF name,**
feld1(10) **TYPE C,**
feld2((20) **TYPE C,**
END OF name.

3. Tabellenobjekte erzeugen:

Eine Tabellendeklaration wird ausschließlich über TYPES bzw. über das DD durchgeführt. Aus Gründen der Kompatibilität zu älteren ABAP-Versionen gibt es noch den direkten Tabellenbefehl. Dieser sollte nicht mehr verwendet werden.

Sonderform LIKE

Bezieht man sich bei einem Datenobjekt auf keinen Datentyp, sondern auf ein Datenobjekt verwendet man die Anweisung LIKE.

DATA: datenobjekt **LIKE** datenobjekt **VALUE** vorbelegung**.**

Vorbelegung von Datenfeldern und Konstanten

Datenobjekte werden bei der Erzeugung mit Initialwerten belegt. Die Anweisung VALUE erzeugt eine abweichende Vorbelegung.

- **VALUE** = Vorbelegung einer Variablen mit einem Wert
  Beispiel:

```
DATA:  feld1        TYPE P  VALUE 451.
DATA:  feld2(30)    TYPE C  VALUE 'Oelpumpe'.
```

- **CONSTANTS** = Mit CONSTANTS können Datenobjekte als Konstanten festgelegt werden.

  Allgemeine Struktur:

  CONSTANTS konstante (länge) **TYPE** (datentyp i,p,t usw.) **VALUE** vorbelegung**.**

Feldsymbole und Datenreferenzen

Der übliche Weg auf Datenobjekte zuzugreifen ist über deren Namen. Bei der Deklaration des Datenobjektes wird der Name und die technischen Eigenschaften festgelegt. Mit dem Namen erhält man den Dateninhalt und kann mit dem Datenfeld arbeiten. Alle Angaben zum Datenobjekt sind fest, und können zur Laufzeit nicht mehr geändert werden. Daher wird auch vom einem **statischen Datenobjekt** gesprochen.

Feldsymbole

Feldsymbole sind Platzhalternamen für Datenobjekte aus dem Hauptspeicher, auf die mit Hilfe der Datenreferenz (Hauptspeicheradresse des Datenobjektes) zugegriffen werden kann. Feldsymbole zeigen demzufolge nur auf ein Datenobjekt und reservieren selbst keinen physischen Speicherplatz. Verweist ein Feldsymbol auf ein Datenfeld, so wird dessen Inhalt dem Feldsymbol zugewiesen. Generische Feldsymbole(z.B. mit ANY typisiert) übernehmen den Datentyp des Datenobjekts. Bei vollständiger Typdefinition des Feldsymbols (z.B: TYPE i) erfolgt eine Prüfung. Feldsymbole müssen deklariert und die Datenobjektinhalte müssen zugewiesen werden. Die Spitzen-Klammern gehören zum Namen des Feldsymbols. Die Feldsymboldeklaration erfolgt mit:

FIELD-SYMBOLS <feldsymbol> (**TYPE** type/**LIKE** datenobjakt)(**ANY TABLE**).

Die Zuweisung zum Feldsymbol wird als **statisch** bezeichnet, wenn der Name des Datenobjektes bereits beim Aufruf des Programms bekannt ist d.h. mit DATA definiert wurde. Die Syntax ist:

ASSIGN datenfeld **TO** <feldsymbol>.

Folgendes Beispiel verdeutlicht die Feldsymbolzuweisung für statisch benannte Datenobjekte.

```
REPORT  z01_datenreferenz_v3
  .
*---- Erzeugen des Feldsymbols-----------------------------*
field-symbols: <feldsymbol> type any.

DATA: name(30) TYPE c.

START-OF-SELECTION.
*---- Zuweisung eines Datenobjektes zum Feldsymbol --------*
  ASSIGN name TO <feldsymbol>.
*---- Zuweisung auf das Feldsymbol und damit auf das Datenobjekt--
--*
  MOVE 'Müller'  TO <feldsymbol>.
  WRITE name.
```

Ist der Name des Datenobjektes, dass einem Feldsymbol zugewiesen werden soll, erst zur Laufzeit bekannt, kann die folgende Variante des ASSIGN-Befehls angewendet werden:

ASSIGN (datenfeld) **TO** <feldsymbol>.

Dabei ist datenfeld der Name des zuzuweisenden Feldes. In diesem Fall wird dies als eine dynamische Zuweisung bezeichnet. Das folgende Beispiel zeigt eine Zuweisung einer beliebigen Tabellenstruktur.

```
*&---------------------------------------------------------------------*
*& Report  Dynamische Feldzuweisung                                    *
*&                                                                     *
*&---------------------------------------------------------------------*

REPORT  z01_datenreferenz_v6                                          .
*----- Tabelle E070 dynamisch ansprechen
DATA: tabname TYPE tabname  VALUE 'E070',
*----- Zwischenrecord
      record(2048).
*---- Def. des Feldsymbols
FIELD-SYMBOLS: <rec>, <field>.

START-OF-SELECTION.
*---- Lesen der dynamischen Tabelle
  SELECT * FROM (tabname) INTO record UP TO 100 ROWS.
*--- Zuweisung der Tabellenstruktur mit Casting in record
    ASSIGN record TO <rec>  CASTING TYPE (tabname).
*---
- Feldweise Ausgabe über ASSING STRUCUTRE... (siehe spätere Erklärung).
    DO.
      ASSIGN COMPONENT sy-index OF STRUCTURE <rec> TO <field>.
      IF sy-subrc <> 0.
        EXIT.
      ENDIF.
      WRITE: /(3) sy-index, <field>.
    ENDDO.
  ENDSELECT.
```

Casting von Datenobjekten

Bei der Zuweisung von Datenobjekten an Feldsymbole können die Feldsymbole mit beliebigen Datentypen typisiert (GECASTET) werden. Hierbei wird die implizite und explizite Typangabe unterschieden. Bei Casting werden die Feldeigenschaften des definierten Feldsymbols auf das Datenobjekt übertragen. Bei Angaben **ohne** Casting ist es genau umgekehrt, d.h. die Dateneigenschaften des Datenobjektes vererben sich auf das Feldsymbol.

Bei der sogenannten **impliziten Typangabe** (d.h. das Feldsymbol ist Typisiert)

ASSIGN feld **TO** <feldsymbol> **CASTING**

wird der Inhalt des zugewiesenen Datenobjektes so interpretiert, als ob er vom Typ des Feldsymbols wäre. Länge und Ausrichtung des Datenobjekts müssen mit dem Feldsymbol verträglich sein. Folgendes Beispiel soll dies verdeutlichen. Die Zuweisung von sy-datum zu dem Feldsymbol würde ohne den Zusatz CASTING nicht funktionieren.

```
REPORT  z01_datenreferenz_v4                                              .

TYPES: BEGIN OF t_datum,
 jahr(4) TYPE n,
 monat(2) TYPE n,
 tag(2)   TYPE n,
 END OF t_datum.

FIELD-SYMBOLS: <feldsymbol> TYPE t_datum.

START-OF-SELECTION.
*--- Casting ist notwendig, das sy-datum nicht typkompatibel
*--- zu t_datum ist.
  ASSIGN sy-datum TO <feldsymbol> CASTING .

* ASSING sy-datum to <feldsymbol>. "Dies würde nicht funktionieren

  WRITE: sy-datum, /, <feldsymbol>-jahr,
  /, <feldsymbol>-monat, /, <feldsymbol>-tag.
```

Bei der **expliziten Typangabe** kann ein generisches Feldsymbol typisiert werden. Der Befehl hierfür ist

ASSIGN feld **TO** <feldsymbol> **CASTING** (**TYPE** type/**LIKE** datenobjekt) **DECIMALS** dec

Beim Zugriff auf das Feldsymbol wird der Inhalt des zugewiesenen Datenobjektes so interpretiert, als ob er vom angegebenen Typ wäre. Folgendes Beispiel soll dies verdeutlichen.

```
REPORT  z01 datenreferenz v5                                              .
*--- Definition einer Struktur -------------*
TYPES: BEGIN OF t_datum,
       jahr(4) TYPE n,
       monat(2) TYPE n,
       tag(2) TYPE n,
       END OF t_datum.
*------ Feldsymbole generisch und n ----------*
FIELD-SYMBOLS: <feldsymbol> TYPE ANY,
               <feld>       TYPE n.

START-OF-SELECTION.
*---- Zuweisung von sy-datum zum Feldsymbol ---------*
  ASSIGN sy-datum TO <feldsymbol> CASTING TYPE t_datum.

  WRITE: /, sy-datum.

  DO.
    ASSIGN COMPONENT sy-index OF STRUCTURE <feldsymbol> TO <feld>.
    IF sy-subrc <> 0.
      EXIT.
    ENDIF.
    WRITE: /, 'Index', sy-index, 'Feldwert', <feld>.
  ENDDO.
```

Dynamische Adressierung von Strukturkomponenten

Eine besondere Form der ASSIGN-Anweisung ist die dynamische Adressierung von Strukturkomponenten. Folgende Syntax der Anweisung ASSIGN erlaubt die dynamische Adressierung der Komponenten:

ASSIGN COMPONENT position **OF STRUKTURE** struktur **TO** <feldsymbol>.

Das nachfolgende Programm verdeutlicht die Anwendung kurz:

```
REPORT  z01_datenreferenz_v2                                          .

DATA: wa_spfli TYPE spfli.

START-OF-SELECTION.
*---- Lesen eines Datensatzes ------------------------------*
  SELECT SINGLE * FROM spfli INTO wa_spfli.
*---- Aufruf des Unterprogramms ----------------------------*
  PERFORM zeilenausgabe USING wa_spfli.

*---Übergabeparameter der gelesenen Satzes -------------------*

FORM zeilenausgabe USING u_line TYPE any.
*--------- Erzeugen des Feldsymbols -------------------------*
  FIELD-SYMBOLS: <fs_felder> TYPE ANY.
*--------- Ausgabeschleife für die einzelnen Felder ----------*
  DO.
*--- sy-index ist ein Systemindex der automatisch mit dem
*--- Schleifendurchlauf hochgezählt wird.
    ASSIGN COMPONENT sy-index OF STRUCTURE u_line TO <fs_felder>.
*-- Abfrage auf ungültige Zuweisung, wenn sy-subrc = 4 (ungültig)
*--- wird die Schleife verlassen
    IF sy-subrc <> 0.
      EXIT.
    ENDIF.
    WRITE: /, <fs_felder>.
  ENDDO.
ENDFORM.
```

Datenreferenzen

Da alle Datenobjekte im Hauptspeicher abgelegt sind, verfügt jedes Datenobjekt über eine Adressierung im Hauptspeicher. Diese Adresse wird als Datenreferenz von Datenobjekten bezeichnet. Die Adresse eines Datenobjektes wird vom ABAP-Laufzeitsystem festgelegt und ist nicht vom Entwickler zu beeinflussen. Referenzen werden in Referenzvariablen gespeichert. In ABAP werden Datenreferenzvariablen für Datenobjekte und Objektreferenzvariablen zur Methodeninstanzierung unterschieden.

Eine **Datenreferenzvariable** wird angelegt mit:

DATA datenref **TYPE REF OF DATA**.

Nach der Definition ist die Datenreferenzvariable initial und verweist auf keine Datenobjekte.

Datenreferenzen können für jedes statische Datenelement mit dem Befehl

GET REFERENZ OF datenobjekt **INTO** datenreferenz.

beschafft werden. Um allerdings auf den Inhalt eines Datenobjektes zuzugreifen, auf den eine Referenz zeigt, muss diese zuerst mit dem Befehl

ASSIGN datenreferenz**->* TO** <feldsymbol>

dereferenziert werden, d.h. wieder zu einem Datenobjekt gemacht werden. Hierzu verwnden wir ein Feldsymbol. Das ->* Symbol gehört zum Datenreferenznamen. Bei erfolgreicher Zuweisung ist sy-subrc = null. Generische Feldsymbole (ANY) übernehmen den Datentyp des Datenobjektes. Bei vollständiger Typisierung des Feldsymbols (i, p, d usw.) wird die Verträglichkeit der Datentypen geprüft. Das nachfolgende Beispiel verdeutlicht nochmals die Anwendung:

```
REPORT  z01_datenreferenz                                              .
*------ Datenreferenzvariablen -----------*
DATA: datenref     TYPE REF TO data,
      datenref_int TYPE REF TO data.
*------ Variable int ----------------------*
DATA: int TYPE i VALUE 5.
*------ Definition von Feldsymbolen ----------------------*
FIELD-SYMBOLS <feldsymbol> TYPE ANY.

START-OF-SELECTION.
*------ Beschaffung der Referenz zu dem Datenfeld ----------------------*
  GET REFERENCE OF int INTO datenref.

  MOVE datenref  TO datenref int.
*------ Übertragen des Feldinhaltes in ein Feldsymbol -------------------
---*
  ASSIGN datenref_int->* TO <feldsymbol>.

  WRITE: 'Ausgabe des Feldsymbolinhalts der Variable int', <feldsymbol>.
```

Alle fest definierten Datenobjekte (statische Datenobjekte), die im Deklarationsteil eines Programms angelegt sind, sind bereits beim Programmstart vorhanden. Es besteht allerdings auch die Möglichkeit ein Datenobjekt zur Laufzeit mit dem Befehl:

CREATE DATA datenreferenz (**TYPE** (type/name) / **LIKE** datenobjekt)

zu erzeugen. Ein dynamisches Datenobjekt lebt, so lange eine Referenzvariable auf sie zeigt. Der GARBAGE COLLECTOR löscht bei fehlender Referenz automatisch. Der Datentyp wird beim Erzeugen des dynamischen Datenfeldes mitgegeben. Folgendes Beispiel soll dies verdeutlichen.

```
REPORT  z01_datenreferenz_v1                                           .
*-------- Definition der Referenzvariablen --*
DATA: datenreferenz TYPE REF TO data.
*-------- Anlage des Feldsymbols ------------*
FIELD-SYMBOLS <feldsymbol> TYPE i.

START-OF-SELECTION.
```

```
*-------- Erzeugen einer Variablen mit dem Type i --*
  CREATE DATA datenreferenz TYPE i.
*-------- Dereferenzierung der Datenreferenz -------*
  ASSIGN datenreferenz->* TO <feldsymbol>.
*----- Wertzuweisung und Ausgabe -------------------*
  MOVE 1000    TO <feldsymbol>.
  WRITE <feldsymbol>.
```

2.4.3 Parameters/Selektionsbild

Die Befehle PARAMETERS und SELECT-OPTIONS erlauben den einfachen Aufbau von Bildschirmdialogen für das Reporting. Programme mit diesen Befehlen beginnen immer mit dem Aufruf eines Dialogbildschirms. Die Ablauflogik der Ereignisse für den Dialogbildschirm ist der Zeitpunkt PBO (AT SELECTION-SCREEN OUPUT) und PAI (AT SELECTION-SCREEN und AT SELECTION SCREEN ON field). Der aufgerufene Bildschirmdialog wird erst mit einer Verarbeitungsbestätigung verlassen.

Bildschirmauswahl für eine Variable

PARAMETERS = Mit der Deklaration PARAMETERS stellt ABAP einen kompletten Bildschirmdialog für die mit PARAMETERS definierten Variablen zur Verfügung. Folgende Beispiele geben Beispiele für die Verwendung von PARAMETERS:

Allgemeine Struktur: PARAMETERS parname **(**laenge**)**
{TYPE typname / **LIKE** objektname**} [DECIMALS** anzahl**]**
[screen optionen **]**
OBLIGATORY „Mussfeld"
VISIBLE LENGTH „Sichtbare Länge"
NO-DISPLAY „Eingabe unterdrücken"
AS CHECKBOX
RADIOBUTTON GROUP grname
[value optionen**]**
DEFAULT wert
LOWER CASE
VALUE CHECK.

PARAMETERS: eingabe (20) **TYPE C.**

oder

TYPE: typ_eingabe(20) **TYPE C.**
PARAMETERS: eingabe **TYPE** typ_eingabe.

oder falls bereits ein Datenobjekt vorhanden ist

PARAMETERS: pdatum **LIKE** sy-datum.

ACHTUNG:

Textbeschreibungen zu einem Selektionsbildschirm können mit Hilfe des Sprungs "Springen-Textelemente-Selektionstest" sinngerecht durchgeführt werden. Übersetzungen von Texten in Fremdsprachen werden durch den Sprung „Springen-Übersetzung" erzielt.

Bildschirmauswahl über eine Selektionstabelle

SELEKTIONSBILDER = Mit Selektionsbilder können von bis Ausgrenzungen vorgenommen werden. Für das Selektionsbild muss eine Instanz verfügbar sein, deren Felder von bis ausgegrenzt werden können. Das einleitende Schlüsselwort für die Ausgrenzung ist SELECT-OPTIONS.

Allgemeine Struktur: SELECT-OPTIONS sel_tabelle **FOR** datenobjekt
[screen optionen]
OBLIGATORY „Mussfeld"
VISIBLE LENGTH „Sichtbare Länge"
NO-DISPLAY „Eingabe unterdrücken"
AS CHECKBOX
RADIOBUTTON GROUP grname
[value optionen]
DEFAULT wert
LOWER CASE
VALUE CHECK.

Die Ausgrenzung umfasst, basierend auf einer Ausgrenzungstabelle, die uns ABAP automatisch zur Verfügung stellt: VON – BIS oder Einzelwert Ein- und Ausgrenzung. Für die Select-Anweisung gibt es darüber hinaus eine spezielle **WHERE** Klausel mit **IN** für den Datenbank **SELECT**. Nachfolgendes Beispiel verdeutlicht die Anwendung von Selektionsbildern.

```
DATA:      satz_dbtabelle TYPE dbtablle.
*----- Deklarationsteil ----------------*
SELECT-OPTIONS:   auswahl1  FOR  satz_dbtabelle-feld1,
           auswahl2  FOR satz_dbtabelle-feld2.

     START-OF-SELECTION.

*----- Standardausgabe ----------------*
*---- IN = automatische Ausgrenzung der Selectionstabelle
     SELECT * FROM dbtabelle INTO satz_dbtabelle
                 WHERE feld1   IN auswahl1 AND
                       feld2   IN auswahl2.
          ENDSELECT.
*----- Ereignisteile PAI----------------*
    AT SELECTION-SCREEN OUTPUT.
                                  CLEAR auswahl1.
      CLEAR auswahl2.
*----- Ereignisteile PAO----------------*
    AT SELECTION-SCREEN.
    AT SELECTION-SCREEN ON auswahl1.
```

Im Hintergrund des SELECT-OPTIONS Befehls wird eine Tabelle mit Kopfsatz geführt. Die Struktur der Tabelle beinhaltet die Ein- und Ausgrenzungsbedingungen für das Datenfeld und besteht aus den Feldern SIGN, OPTION, HIGH und LOW. Wenn eine Vorbelegung der Selections-Option gewünscht wird, kann die Tabelle mit Aus- und Eingrenzungen z.B. im Event AT SELECTION-SCREEN OUTPUT gefüllt werden.

```
REPORT  z01_option_v1                                                  .

DATA:       satz_dbtabelle TYPE spfli.
*----- Deklarationsteil ----------------*
SELECT-OPTIONS:   auswahl1  FOR satz_dbtabelle-connid.
*---- Alternative zu Verwendung der Auswahlkopfzeile
*---- Anlage eines WA-Bereichs mit der Struktur der Tabelle--*
DATA: wa  LIKE LINE OF auswahl1.

START-OF-SELECTION.

*----- Standardausgabe ----------------*
*---- IN = automatische Ausgrenzung der Selectionstabelle
SELECT * FROM spfli INTO satz_dbtabelle
               WHERE connid   IN auswahl1.

ENDSELECT.
*----- Ereignisteile PAI----------------*
AT SELECTION-SCREEN OUTPUT.
  CLEAR auswahl1.
*---- Vorbelegung
*---- I = Inclusiv, E=Exclusiv
  auswahl1-sign = 'I'.
*--- BT=Werte Zwischen, EQ=Wert Gleich
  auswahl1-option = 'BT'.
*---- Kleinster und Größter Wert
  auswahl1-low = '0001'.
  auswahl1-high = '0088'.
*---- Füllten den Tabelle über Kopfsatz
  APPEND auswahl1.
*---- Alternativ: Füllen über den WA Bereich
  wa-sign = 'E'.
  wa-option = 'EQ'.
  wa-low    = '0017'.
  wa-high   = '0000'.
  APPEND wa TO auswahl1.
*----- Ereignisteile PAO----------------*
AT SELECTION-SCREEN.
```

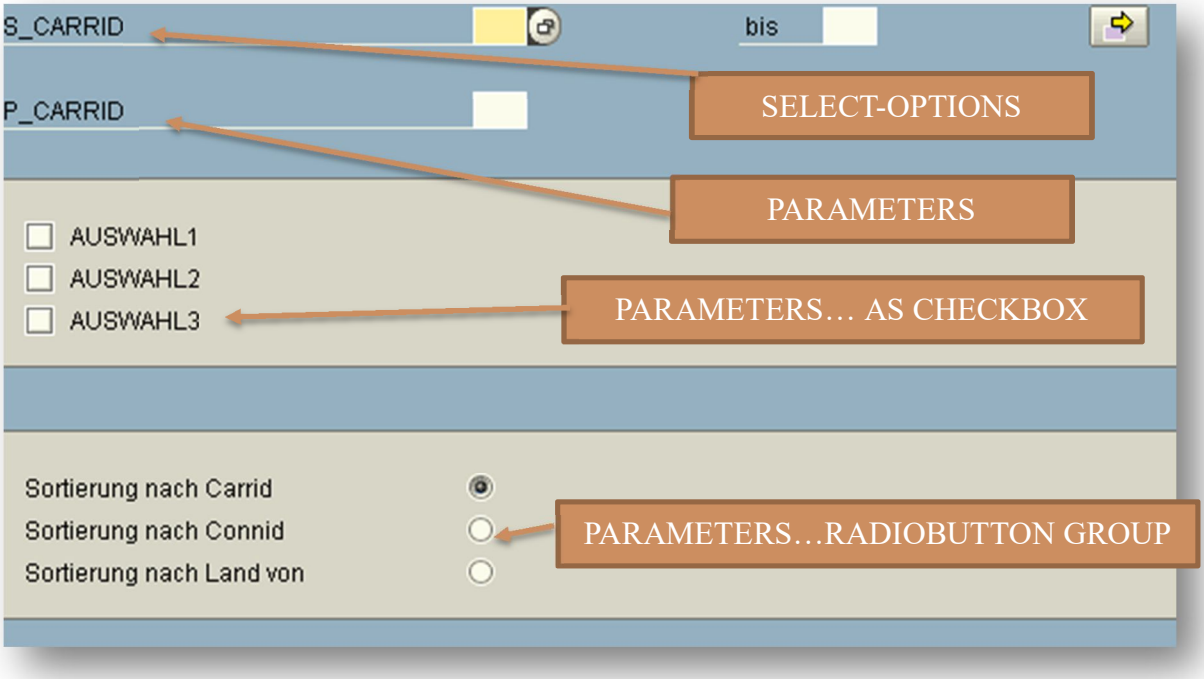
Komplexe Selektionsbilder mit Frames, Radiobuttons und Checkboxen

Wie das nachfolgende Beispiel zeigt können Selektionsbilder und Parameter auch zu komplexen Bildschirmen zusammengeführt werden. Das System verwaltet Selektionsbildschirme/Parameter unter der Dynpronummer 1000. Das nachfolgende Beispiel zeigt einen komplexen Bildschirm.

Deklaration:

---- Syntax zum Aufbau eines Selektionsbildschirms----------

```
SELECT-OPTIONS s_carrid FOR wa-carrid.
*---- Leerzeile ----------*
SELECTION-SCREEN SKIP.
*---- Parameters und Auswahlbilder
PARAMETERS: p_carrid TYPE spfli-carrid.
*---- Leerzeile ----------*
SELECTION-SCREEN SKIP.
*---- Syntax zum Aufbau eines Selektionsbildschirms----------*
SELECTION-SCREEN BEGIN OF BLOCK sortierung1 WITH FRAME.
*---- Syntax einer CHECKBOX-Eingabe ----------*
PARAMETERS: auswahl1 AS CHECKBOX,
        auswahl2 AS CHECKBOX,  " DEFAULT 'X',
        auswahl3 AS CHECKBOX.
SELECTION-SCREEN END OF BLOCK sortierung1.
*---- Leerzeile ----------*
SELECTION-SCREEN SKIP.
*---- Syntax für RADIOBUTTON----------*
SELECTION-SCREEN BEGIN OF BLOCK sortierung2 WITH FRAME.
PARAMETERS:    radio1 RADIOBUTTON GROUP gr1,
               radio2 RADIOBUTTON GROUP gr1, " DEFAULT 'X',
               radio3 RADIOBUTTON GROUP gr1.
SELECTION-SCREEN END OF BLOCK sortierung2.
```



Zur Formatierung von Selektions-/Parameterbildschirmen stehen folgen Befehle zur Verfügung:

- **SELECTION-SCREEN SHIP** (anzahl). "Leerzeile"
- **SELECTION-SCREEN ULINE** (pos/laenge) "Unterstrich"
- **SELECTION-SCREEN COMMENT** (pos/variable) "Kommentar"

Bereichsdefinitionen:

SELECTION-SCREEN BEGIN OF LINE.
(Selektionsbildelemente)

```
SELECTION-SCREEN END OF LINE.

SELECTION-SCREEN BEGIN OF BLOCK nameWITH FRAME
                                        {TITEL [titel]}.
        (Selektionsbildelemente)
SELECTION-SCREEN END OF BLOCK name.
```

Eigenständige Selektionsbilder

Deklarationsteil:

```
SELECTION-SCREEN BEGIN OF SCREEN dynnr {TITLE [title]}
                                        {AS WINDOW}
    (Selektionsbild-Elemente)
SELECTION-SCREEN END OF SCREEN dynnr.
```

Ablaufteil:

```
CALL SELECTION-SCREEN dynnr  (STARTING AT x1 y1)
                             (ENDING AT x2 y2).
```

2.4.4 Systemvariablen

Das ABAP-System stellt dem Programmierer verschiedene Systemvariablen zur Verfügung. Systemvariablen werden automatisch mit Werten befüllt und sind im DD typisiert. Eine der wichtigsten Systemvariablen ist SY-SUBRC. Über diese Variable können z.B. erfolgreiche Datenbankzugriffe oder die erfolgreiche Bearbeitung einer internen Tabelle kontrolliert werden. Die Kontrolle des erfolgreichen Zugriffs auf eine Datenbanktabelle wird z.B. wie folgt realisiert.

- **SY-SUBRC** = RETURNCODE nach Datenbankzugriffen, Tabellenzugriffen usw.. Diese Variable gibt an, ob ein Befehl erfolgreich ausgeführt wurde. Z.B. KZ für den Zustand nach einem Datenbankzugriff (0 = Erfolgreich).

Beispiel Zugriff auf eine Datenbanktabelle:

```
REPORT  abc.                                                    .

SELECT * FROM dbtabelle.
    WRITE: / dbtabelle.
  ENDSELECT.
  IF SY-SUBRC <> 0.
       WRITE: / 'Keine Daten gefunden'.
  ELSE.
    ULINE.
WRITE: / 'Alle Datensätze ausgeben'.
ENDIF.
```

Weitere wichtige Systemvariablen sind z.B.:

- **SY-DATUM** = Systemdatum (YYYYMMDD)
- **SY-UZEIT** = Systemzeit (HHMMSS)
- **SY-UNAME** = Benutzername
- **SY-UCOMM** = Returncode aus der Menüleiste/Drucktastenleiste/ Funktionstasten bei einer Liste
- **SY-PFKEY** = Name der derzeit aktiven Menüleiste/Drucktastenleiste/Funktionstasten
- **SY-INDEX** = Anzahl der Schleifendurchläufe
- **SY-MANDT** = angemeldete Mandantennummer

Listenvariablen:

- **SY-LSIND** = Drilldownvariable in einer Liste, die angibt, wie viele Detailsprünge bereits durchgeführt wurden.
- **SY-TITLE** = Reporttitel
- **SY-LINCT** = Zeilenanzahl der REPORT-Anweisung
- **SY-SINSZ** = Zeilenbreite der REPORT-Anweisung
- **SY-SROWS** = Zeilenanzahl im Fenster (gesamter Scrollbereich)
- **SY-SCOLS** = Spaltenanzahl im Fenster
- **SY-PAGNO** = Seitennummer der aktuellen Seite
- **SY-LINNO** = Zeilennummer der aktuellen Zeile
- **SY-COLNO** = Spaltennummer der aktuellen Spalte
- **SY-UCOMM** = Funktionscodefeld in einer Liste d.h. welche Funktionstaste wurde vom Anwender gedrückt.

2.4.5 Listenausgaben

Grundbefehle

Für die Listenausgabe stehen verschiedene Befehle zur Verfügung.

- **WRITE** = Ausgabe einer Reportzeile.

Allgemeine Struktur:

WRITE: [AT] [/] [position] [(länge)] feld.
[Formatierungsinformation**]**
LEFT-JUSIFIED / CENTERED / RIGHT-JUSTIFIED
USING EDIT MASK
COLOR [farbnummer**]**
Usw.
[Listenelemente**] „Darstellung spezieller Listenelemente**
.

Achtung: Aufgrund der vielfältigen Optionen und Zusätze für **Farben, Bitmaps usw.** Im Write-Befehl benutzen Sie die Funktion „MUSTER“

```
WRITE: /01 'Testzeile1' ,  /20 `Testzeile2`.
```

- „/“ = Leerzeile
- „ , “ = Trennung der Aufgabe von Variablen oder Textzeilen.
  Um ein Komma zu verwenden, muss nach dem ABAP-Schlüsselwort WRITE ein Doppelpunkt stehen (siehe Beispiel).
- `textzeile`
- Zahl = Spaltenpositionierung

```
WRITE: /10 'Guten Tag' , 'Druck ab Spalte 10'.
```

⊗ **ULINE** = Trennlinie z.B.

Allgemeine Struktur: **ULINE [AT [/] [position] [(laenge)]** feld**.**
WRITE: `Testzeile 1`.
ULINE.

- **NEW-LINE.** “für einen Zeilenvorschub“

SKIP anzahl = Leerzeile z.B.
Allgemeine Struktur: **SKIP [**anzahl**].**

SKIP 5 „fünf Leerzeilen drucken“

- **Listenlänge im Kopf der Liste definieren**
  REPORT name **LINE-COUNT** länge**.**

- **Listenbreite in Kopf der Liste schalten**
  REPORT name **LINE-SIZE** breite**.**

- **Absolute Positionierung des Listencursors**
  POSITION col**.** **„für die horizontale Position“**
  SKIP TO LINE line**. “für die vertikale Position”**

- **Listenkopf und Listenfuß:** Ein Listenkopf kann innerhalb von TOP-OF-PAGE erzeugt werden. Ein Listenfuß mit END-OF-PAGE.
  TOP-OF-PAGE.
  WRITE ´Liste der Ausgaben` **.**
  ULINE.
  END-OF-PAGE.
  ULINE.
  WRITE: ‘Ende’**.**
- **Lesen einer Zeile in einer angezeigten Liste:** Ohne Index wird die aktuelle Zeile gelesen, auf die ein Ereignis stattfindet.
  READ LINE zeile (**INDEX** indexnr).
- **Verändern einer Zeile in einer angezeigten Liste:** Ohne Index wird die aktuelle Zeile modifiziert, auf der gerade ein Ereignis stattfindet.
  MODIFY LINE line **(INDEX** indexnr**).**

ACHTUNG: Wollen Sie aus einem Dialogmodul auf eine Liste verzweigen, ist es notwendig den Reportgenerator aufzurufen. Hierzu müssen Sie in dem Dialogmodul folgende Anweisung einfügen:

LEAVE TO LIST-PROCESSING (AND RETURN TO SCREEN dynpronr**.**

Eine Rückkehr zu einem Dynpro ist möglich mit dem Befehl

LEAVE LIST-PROCESSING.

Bei der automatischen Anzeige einer Liste in ausführbaren Programmen setzt die Laufzeitumgebung einen Standardlistenstatus, der die listenspezifischen Funktionen wie Blättern usw. enthält.
Sie können mit dem Befehl **SET PF-STATUS space** die Listenoberfläche selbst programmieren und die Listendialogstandards ausschalten. Wie bei der Dialogprogrammierung können Sie nun einen selbstdefinierten Listenstatus vor der Listenanzeige mit der Anweisung **SET PF-STATUS statusname** erzeugen.

Programmierbare Ereignisblöcke in einer Liste sind **AT LINE-SELECTION** und **AT USER-COMMAND**

AT USER-COMMAND

Mit dem Ereignisblock AT USER-COMMAND kann auf entsprechende Benutzeraktionen innerhalb einer Liste reagiert werden. Der Funktionscode wird im Systemfeld sy-ucomm zurückgegeben (Bei Dialogprogrammen im definierten OK-CODE)
Beispiel:

```
AT USER-COMMAND.
  CASE sy-ucomm.
    WHEN 'SELE'.
* Anweisung
    WHEN 'INFO'.
* Anweisung

  ENDCASE.
```

AT LINE-SELECTION.

Das Ereignis **AT LINE-SELECTION** wird bei einem Doppelklick auf eine Listenzeile ausgelöst. Um einen **DRILL-DOWN** einfach zu realisieren, kann man mit der Anweisung „**HIDE variablen**“ den Inhalt der aktuellen Listenzeile (sy-linno) und der aktuellen Listenstufe (sy-lsind) in einer „unsichtbaren“ Tabelle speichern.

HIDE = Übergabe der angegebenen Felder von dem Report in die AT LINE-SELECTION.

Der Hide-Befehl speichert die gewünschten Übergabevariablen einer angezeigten Liste in einer internen von ABAP verwalteten Tabelle. Diese Tabelle kann nun zum Detailsprung verwendet werden. Im Ereignisblock **AT LINE-SELECTION** kann so das Ereignis „Doppelclick“ auf eine Reportzeile ausgewertet werden.

Beispiel:

```
DATA:
*---- Fluggesellschaften
      wa_spfli TYPE spfli,
*---- Flüge der Fluggesellschaften
      wa_sflight TYPE sflight.
PARAMETERS: eing(10).
START-OF-SELECTION.
  SELECT * FROM spfli INTO wa_spfli.
    WRITE: /,
          wa_spfli-carrid,
          wa_spfli-connid,
          wa_spfli-countryfr,
          wa_spfli-cityfrom,
          wa_spfli-airpfrom,
          wa_spfli-countryto,
          wa_spfli-cityto.
    HIDE:  wa_spfli-carrid,  wa_spfli-connid.
  ENDSELECT.
*---- Überschrift.
TOP-OF-PAGE.
  WRITE: / 'Fluggesellschaften'.
  ULINE.

TOP-OF-PAGE DURING LINE-SELECTION.
  CASE sy-lsind.
    WHEN 1.
      WRITE: / 'Ausgabe von Details'.
    WHEN OTHERS.
      WRITE: / 'Keine korrekte Liste'.
  ENDCASE.
*----- Ausdruck der Selektionswerte ----------------*
AT LINE-SELECTION.
  CASE sy-lsind.
    WHEN 1.
      SELECT *      FROM sflight INTO wa_sflight
                  WHERE   ( carrid = wa_spfli-carrid )
                  AND     ( connid = wa_spfli-connid ).

        WRITE: /, wa_sflight-carrid,
              wa_sflight-connid,
              wa_sflight-fldate,
              wa_sflight-price,
              wa_sflight-currency,
              wa_sflight-planetype,
              wa_sflight-seatsmax,
              wa_sflight-seatsocc,
              wa_sflight-paymentsum,
              wa_sflight-seatsmax_b,
              wa_sflight-seatsocc_b,
              wa_sflight-seatsmax_f,
              wa_sflight-seatsocc_f.
      ENDSELECT.
      IF sy-subrc <> 0.
         WRITE: 'keine Ergebnisse'.
      ENDIF.
     WHEN OTHERS.
       WRITE: 'Keine Selektion', sy-lsind.
    ENDCASE.
```

Ausgabe eines Detailsprungs als Dialogfenster:

Man kann auch die Verzweigungsliste als Dialogfenster darstellen. Hierzu gibt es die Anweisung:

```
WINDOW     STARTING   AT zeile spalte
            ENDING   AT zeile spalte.
```

2.4.6 Zeichenkettenbearbeitung

Mit den folgenden Befehlen können Zeichenketten bearbeitet werden:

⊗ **CONCATENATE** = die Anweisung dient der Verknüpfung von Zeichenketten
CONCATENATE feld1 feld2 feld3 **INTO** gesamtfeld.
CONCATENATE feld1 feld2 feld3 **INTO** gesamtfeld
SEPARATED BY ´;´.

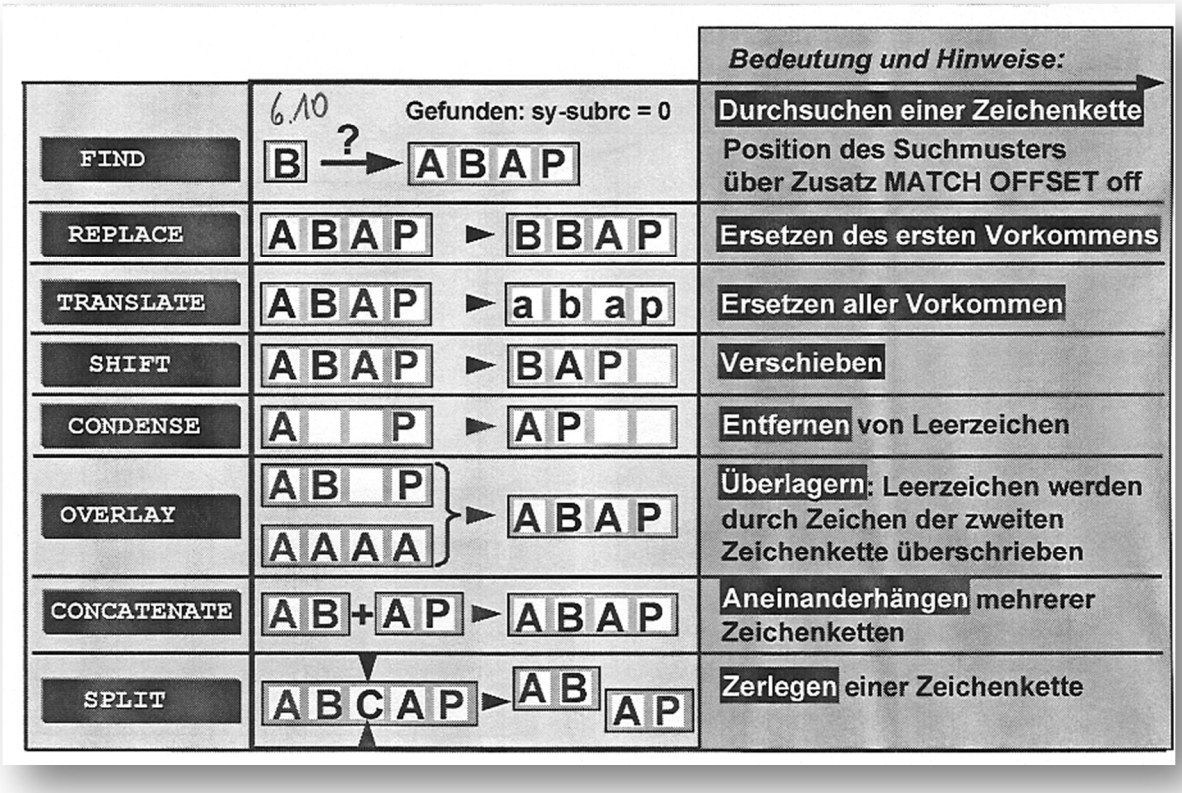
⊗ **SPLIT** = die Anweisung dient der Zerlegung von Zeichenketten
gesamtfeld = ´Huber,Maier,Hohmann´.
SPLIT gesamtfeld **AT** ´**,**´ **INTO** feld1 feld2 feld3.

⊗ **SHIFT** = diese Anweisung dient der Verschiebung von Zeichenketten**.**

⊗ **REPLACE** = Ersetzt das erste Auftreten des gewünschten Wertes durch
einen anderen Wert.
REPLACE ´H´ **WITH** ´X´ **INTO** gesamtfeld.

⊗ **TRANSLATE** = Ersetzt ein (wirklich ein Zeichen) Zeichen durch ein anderes Zeichen. **TRANSLATE** gesamtfeld **USING** ´HX´.

⊗ **SEARCH** = Sucht nach einer bestimmen Zeichenkette. Bei Erfolg hat der SY-SUBRC den Wert Null.
SEARCH gesamtfeld **FOR** ´Hohmann´.
IF sy-subrc = 0.
WRITE: / ´Suche war erfolgreich´.
ENDIF.



2.4.7 Grundrechenarten

Der Computebefehl kann alle notwendigen Rechenoperationen durchführen.

COMPUTE = Sie können in ABAP beliebig „tief“ geschachtelte arithmetische Ausdrucke programmieren. Beachten Sie dabei, dass die Klammern und Operatoren Schlüsselwörter sind, also mindestens in ein Leerzeichen eingefasst werden müssen.
Allgemeine Struktur: **COMPUTE** ergebnis =
Operand1 **{+,-,*,/,DIV,MOD,**}** operand2.

COMPUTE ergebnis = zahl1 * 100 / max.

Weitere gültige Operatoren sind:

⊗ +	**Addition**	**ADD x TO y.**
⊗ -	**Subtraktion**	**SUBTRACT x FROM y.**
⊗ *	**Multiplikation**	**MULTIPLY x BY y.**
⊗ /	**Division**	**DIVIDE x BY y.**
⊗ **	**Potenz**	**z = x ** y.**
⊗ **DIV ganzzahlige Division ohne Rest**		**z = x DIV y.**
⊗ **MOD Rest bei ganzzahliger Division**		**z = x MOD y.**

2.4.8 Variablenübertragung

Zur Variablenübertragung kann der **MOVE**-Befehl verwendet werden. Mit MOVE erfolgt die Werteübertragung von LINKS nach RECHTS. Wird kein MOVE verwendet, ist die Wertübertragung von RECHTS nach LINKS.

Allgemeine Struktur:

MOVE source **TO** ziel.
ziel = source.

MOVE-CORRESPONDING source **TO** ziel.
CLEAR feld.

- **MOVE** = Übertragung von Links nach Rechts.
  MOVE source TO ziel.

  Sonderform von MOVE ist die Teilübertragung von Zeichen.
  ***-- Dieser Befehl überträgt ab der 7 Position**
  ***-- 8 Zeichen in feld1.**
  MOVE gesamtbereich+6(8) TO feld1.

- **=** = Übertragung von Rechts nach Links z.B. ***B = A.***

Zur Übertragung ganzer Strukturen mit gleichem Variablennamen steht MOVE-CORRESPONDING zur Verfügung.

- **MOVE-CORRESPONDING** = Schieben von namensgleichen Variablen von einer Struktur in eine andere. In Zusammenhang mit der Übertragung von namensgleichen Variablen verwendet ABAP den Bindestrich d.h. als namensgleiche Variablen werden alle die Variablen geschoben, die nach dem Bindestrich namensgleich sind. Deshalb können Bindestriche nicht für Variablennamen bei der Programmierung verwendet werden.

```
TYPES: BEGIN OF stru_a,
    A(10) TYPE  C,
    B(10) TYPE  D,
     END OF stru_a.
DATA: satz_a TYPE stru_a,
    satz_b TYPE stru_a.

MOVE-CORRESPONDING satz_a TO  satz_b.
*-- ist genauso wie
MOVE satz_a-a  TO satz_b-a.
MOVE satz_a-b TO satz_b-b.
*-- ist genauso wie
satz_b-a = satz_a-a.
satz_b-b = satz_a-b.
```

- **CLEAR** feld1 = Setzt eine Variable, eine Struktur oder eine Tabelle auf den Initialwert zurück.

2.4.9 Logische Steuerungen

- **IF** = Abfrage einer Bedingung; Schachtelung mit ELSEIF
  Logische Ausdrücke für den Vergleich

  Sind z.B. =, >, <, >=,<=,<> **oder EQ ,GT, LT, GE, LE, NE.** Bedingungen sind **AND, OR, NOT**. Sondervergleich sind **IF feld1 IS INITIAL** oder **IF feld1 BETWEEN feld2 AND feld3.**

  Allgemeine Struktur:
  IF bedingung_1.
  <anweisungsblock_1>
  ELSEIF bedingung_2.
  <anweisungsblock_2>

  ELSE.
  <anweisungsblock_n>

  ENDIF.

```
IF feld1 = 10.
WRITE: / 'Die Zahl 10 ist aufgetreten'.
ELSEIF feld1 = 20.
WRITE: / 'Die Zahl 20  ist aufgetreten'.
ENDIF.
```

- **CASE** = Abfrage von Bedingungen
  Allgemeine Struktur:
  CASE variable.
  WHEN wert_1 **[OR…..].**
  <anweisungsblock_1>
  WHEN wert_2 **[OR…..].**
  <anweisungsblock_2>

  WHEN OTHERS.
  <anweisungsblock_n>
  ENDCASE.

```
CASE feld1.
WHEN 10.
    WRITE: /  'Die Zahl 10 ist aufgetreten'.
WHEN 20.
```

```
    WRITE: /     'Die Zahl 20 ist aufgetreten'.
WHEN OTHERS.
     WRITE ´andere Zahlen´.
ENDCASE.
```

- **DO** = Anweisungen zwischen DO und ENDDO X-Mal durchführen

Allgemeine Struktur:
DO [n **TIMES]**
<anweisungsblock>
ENDDO.

```
DO 4 TIMES.
WRITE: / ´Testausgabe´.
ENDDO.
```

- **WHILE** = Die Anzahl der Durchgänge zwischen WHILE und ENDWHILE ist von einer Bedingung abhängig.

Allgemeine Struktur:
WHILE bedingung**.**
<anweisungsblock>
ENDWHILE.

```
      zaehler = zero.
WHILE zaehler = 5.
WRITE: / ´Testausgabe´.
COMPUTE zaehler = zaehler + 1.
ENDWHILE.
```

ACHTUNG: Mit dieser Anweisung können sehr schnell Endlosschleifen programmiert werden.

- **LOOP AT** = Schleife zur komfortablen Abarbeitung einer internen Tabelle vom ersten bis zum letzten Tabellenplatz.

Allgemeine Struktur:
LOOP AT itab **INTO** wa
FROM idx1 **TO** idx2
WHERE log_ausdruck

<anweisungsblock>

ENDLOOP.

Itab = interne Tabelle
wa = workingarea

LOOP AT interne_tabelle INTO wa
WRITE: / wa-tabellen_feld1, wa-tabellen_feld2.
ENDLOOP.

- **EXIT** = Schleifen können mit EXIT verlassen werden.

```
DO 3 TIMES.
IF jn_kz = ´N´.
        EXIT.
ELSEIF jn_kz = ´J´.
        WRITE: / `Schleifendurchlauf´.
ENDIF.
ENDDO.
```

2.4.10 Interne Tabellen

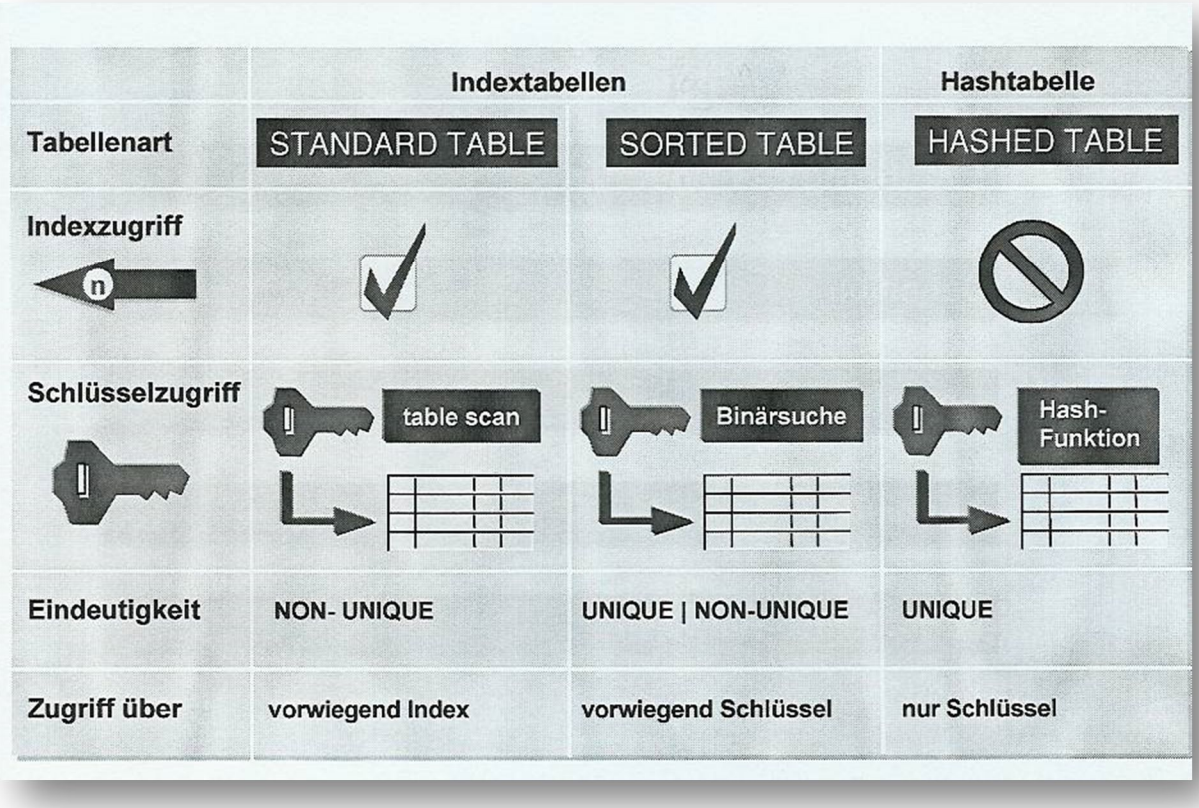
Es werden verschiedene Arten von internen Tabellen unterschieden. So kann eine Tabelle als Standardtabelle, sortierte Tabelle und Hash-Tabelle definiert werden. In Abhängigkeit der Definition sind unterschiedliche Befehle (APPEND, INSERT, DELETE, READ, MODIFY) auf die Tabelle anwendbar. Der Zugriff auf eine interne Tabelle kann entweder über den Index (durchnummerierte Zeilen der Tabelle von 1 bis n) oder einem Schlüssel (Attribut einer Tabelle) erfolgen. Die Typisierung ist wie folgt:

	Standard	**Sortierte Tabelle**	**Hash Tabelle**
INDEX	JA	JA	Nein
Schlüssel	(JA möglich)	JA	JA
Eindeutigkeit des Schlüssels	NON-UNIQUE	UNIQUE und NON-UNIQUE	UNIQUE
Zugriff auf Zeilen	Index (Schlüssel nur mir READ)	Index oder Schlüssel	Nur Schlüssel
Befehl APPEND (anhängen)	JA	NEIN	NEIN
Befehl INSERT	Nein (Wirkung wie APPEND)	JA (Schlüssel)	JA (Schlüssel)
Befehl READ	Index oder Schlüssel	Index oder Schlüssel	Schlüssel
Befehl DELETE	Index	Index oder Schlüssel	Schlüssel
Befehl MODIFY	Index	Index oder Schlüssel	Schlüssel
Befehl SORT	JA	Nein	Nein
Befehl LOOP AT	JA	JA	JA
Befehl CLEAR	JA	JA	JA

Deklaration = Deklaration von internen Tabellen immer über TYPES bzw. DD.

Allgemeine Struktur: **TYPES** tab_strukt **{TYPE/LIKE}**

{**STANDARD TABLE** / **SORTED TABLE** / **HASHED TABLE**}
TABLE OF {datenfeld / satz_struktur}
WITH {UNIQUE / NON-UNIQUE}
{feldname / **DEFAULT KEY**



- **Standard-Tabelle:**

```
TYPES:   BEGIN OF struktur_satz,
       feld1(10) TYPE C,
       feld2(10) TYPE C,
       END OF struktur_satz.
TYPES :  stru_tabelle TYPE STANDARD TABLE OF struktur_satz.
```

Oder:

```
TYPES:   stru_tabelle TYPE STANDARD TABLE OF struktur_satz
WITH NON-UNIQUE KEY feld1.

DATA: tabelle TYPE stru_tabelle.
```

- **Sortierte Tabelle:**

```
TYPES:  BEGIN OF struktur_satz,
      feld1(10) TYPE C,
      feld2(10) TYPE C,
       END OF struktur_satz.
```

```
TYPES:   stru_tabelle TYPE SORTED TABLE OF struktur_satz
WITH NON-UNIQUE KEY feld1.
```

Oder:

```
TYPES: stru_tabelle TYPE SORTED TABLE OF struktur_satz
       WITH UNIQUE KEY feld1.
DATA: tabelle TYPE stru_tabelle.
```

- **Hash-Tabelle:**

```
TYPES:   stru_tabelle TYPE HASHED TABLE OF struktur_satz
WITH UNIQUE KEY feld1.
```

Operationen auf interne Tabellen (Struktur = Datensatzstruktur)

- **APPEND** = Append hängt den Inhalt einer Struktur an eine interne Tabelle an. Diese Operation wird nur bei **Standardtabellen** unterstützt.

Allgemeine Struktur:
APPEND {zeile / **LINES OF** jtab} **TO** itab.

APPEND satz to tabelle.

- **INSERT** = fügt den Inhalt der Struktur in eine interne Tabelle ein. Bei Standardtabellen bewirkt dies ein Anhängen, bei **sortierten Tabellen** ein sortiertes Einfügen und bei **Hash-Tabellen** ein Einfügen gemäß einem Hash-Algorithmus. INSERT wirkt bei Standardtabellen wie APPEND.

Allgemeine Struktur:
INSERT {zeile / **LINES OF** jtab}
INTO TABLE itab.

Oder:

INSERT zeile INTO itab **[INDEX idx]**

INSERT satz INTO TABLE tabelle <condition>.
Einfügen nach Tabellenlogik (Sortiert oder Hash, Anfügen bei Standardtabellen).
INSERT satz INTO TABLE tabelle.
Einfügen über Index
INSERT satz INTO tabelle INDEX indnr.

- **READ TABLE** = Read kopiert den Inhalt einer Zeile einer internen Tabelle in eine Struktur.
Allgemeine Struktur:
READ TABLE itab **INTO** *wa* **INDEX** idx **.**

Oder:

READ TABLE itab **INTO** wa **FROM** key**.**

Lesen nach INDEX:

```
READ TABLE tabelle INTO satz INDEX indnr.
```

Lesen nach Schlüssel:

```
READ TABLE tabelle INTO satz  FROM key.
```

Oder

```
READ TABLE tabelle INTO satz WITH KEY key.
```

ACHTUNG: nach dem Lesen steht die Indexnummer in der Systemvariablen SY-TABIX. Wenn Sie diese Variable in ein Datenfeld sichern , können anschließend alle notwendigen INDEX-Operationen durchgeführt werden.

```
DATA  ind TYPE SY-TABIX.     .
.
.
READ TABLE tabelle INTO satz WITH KEY feld1 = ‚Giessen’.
  MOVE SY-TABIX TO ind.
.
.
MODIFY tabelle FROM satz INDEX ind.
```

- **DELETE** = Delete löscht eine Zeile aus der internen Tabelle.
  Allgemeine Struktur:
  DELETE itab **INDEX** idx**.**
  oder
  DELETE TABLE itab **FROM** key**.**
  oder
  DELETE itab **[FROM** idx1**] [TO** idx2**].**
  oder
  DELETE itab **WHERE** log_ausdruck**.**

  DELETE tabelle <condition>
  Löschen mit Index
  DELETE tabelle **INDEX** indnr**.**
  Löschen von bis
  DELETE tabelle **FROM** vonindnr **TO** bisindnr**.**
  Löschen mit Schlüssel
  DELETE TABLE tabelle **FROM** key**.**
  oder
  DELETE TABLE tabelle **WITH TABLE KEY** key
  Löschen mit logischen Ausdruck
  DELETE tabelle **WHERE** logischer_ausdruck
  z.B. feld1 = 100.

- **MODIFY** = Modify überschreibt eine Zeile einer internen Tabelle mit dem Inhalt einer Struktur.

  Allgemeine Struktur:

  MODIFY TABLE itab **FROM** zeile.
  oder
  MODIFY itab **FROM** zeile **INDEX** idx.

  Veränderung mit Index

  MODIFY tabelle **FROM** satz **INDEX** indnr.

  Veränderung mit Schlüssel: In diesem Fall wir in der internen Tabelle nach der ersten Zeile gesucht, deren Schlüsselwert mit den Werten des Satzes übereinstimmen.

  MODIFY TABLE tabelle **FROM** satz.

  Veränderung von einzelnen Datenfeldern

  MODIFY tabelle **FROM** satz **TRANSPORTING** feld1, feld2.

  Veränderung mit logischem Ausdruck

  MODIFY tabelle **FROM** satz **WHERE** logischer_ausdruck.

  oder

  MODIFY tabelle **FROM** satz **TRANSPORTING** feld1, feld2 **WHERE** logischer_ausdruck.

Operationen auf die gesamte Tabelle

- **SORT** = Sortieren der Standard- und Hash-Tabellen nach beliebigen Feldern der Tabelle

  Allgemeine Struktur:

  SORT itab **ASCENDING** **[AS TEXT]**
  DESCENDING
  BY comp **ASCENDING** **[AS TEXT]**
  DESCENDING

```
SORT tabelle BY feld1 DESCENDING   BY feld1.
SORT tabelle BY feld1 ASCENDING    BY feld1.
```

- **REFRESH** tabelle oder **CLEAR** tabelle = Löschen der Tabelleninhalte

Mengenoperationen auf interne Tabellen = Löschen der gesamten internen Tabelle.

- **LOOP AT** = Schleife über die Tabelle

  Allgemeine Struktur:

  LOOP AT itab **INTO** wa

```
        FROM idx1 TO idx2
        WHERE log_ausdruck    .

        <anweisungsblock>
ENDLOOP.
LOOP AT tabelle INTO satz <condition>.
            ….Anweisungen
ENDLOOP.
```

Beispiele Lesen per Index:

```
LOOP AT tabelle INTO satz FROM 1 TO 5.
ENDLOOP.
LOOP AT tabelle INTO satz WHERE feld1 = "ERICH".
```

- **INSERT LINES OF tabelle1 <condition> INTO tabelle2 <condition>**

  ACHTUNG: Nach dem Zugriff auf eine Tabelle (Lesen, Schreiben, Einfügen usw.) sollte der **SY-SUBRC** auf 0=Lesezugriff war OK geprüft werden, um dann die anschließende Operation richtig auszuführen.

2.4.11 Datenbankzugriffe

Funktion	Aufruf	Kommentar
Lesen von Datenbankinhalten in einen Datenbereich Anmerkung : SELECT * Wählt alle Felder der Tabelle aus	**SELECT** <felder> **FROM** <tabelle> **INTO** <wa> **WHERE** <Bedingung> **GROUP BY** feld **HAVING** feld **ORDER BY** feld <ABAP-Anweisungen> **ENDSELECT**.	Schlüsselwort Datenfelder/Datensatz Tabellenname Datenbereich (DATA) Ausgrenzung
Lesen von einem Datensatz ohne ENDSELECT.	**SELECT SINGLE** <felder> **FROM** <tabelle> **INTO** <wa> **WHERE** <Bedingung>.	Schlüsselwort Datenfelder/Datensatz Tabellenname Datenbereich (DATA) Ausgrenzung
Lesen in eine interne Tabelle	**SELECT** <felder>	Schlüsselwort Datenfelder/Datensatz

	FROM <tabelle> **INTO TABLE** <wa> **WHERE** <Bedingung>	Tabellenname Interne ABAP-Tabelle Ausgrenzung
Lesen in Verbindung mit dem Befehl SELECT-OPTIONS	**SELECT** <felder> **FROM** <tabelle> **INTO** <wa> **WHERE** dbfeld **IN** auswahlfeld	Schlüsselwort Datenfelder/Datensatz Tabellenname Datenbereich mit Tabellenstruktur Datenbankfeld ausgegrenzt mit Selektionsfeld aus dem Befehl SELECT-OPTIONS
Erweiterung der WHERE-Klausel	**WHERE** feld =,<=,>=, **NOT**, **AND**,**OR** bedingung	
	WHERE feld **BETWEEN** wert1 and wert2	
	WHERE feld **LIKE** '%xxx'	
	WHERE feld **IN** ('wert1', wert2 usw.)	
ABAP-JOIN	**SELECT** <felder> **FROM** <tabelle1> **AS** A **INTO** <wa> **INNER JOIN** <tabelle2> **AS** B **ON** A~feld = B~feld. **ENDSELECT**	Schlüsselwort Felder Tabelle Datenbereich Tabelle2 Syntax: Alias~Tabellenfeld
Löschen	**DELETE**	Siehe Detail
Spaltenaktualisierung	**UPDATE**	Siehe Detail
Veränderung	**MODIFY**	Siehe Detail
Einfügen	**INSERT**	Siehe Detail

Allgemeine Befehle

- **SELECT** = Lesen aus einer Datenbanktabelle und Abstellen der Ergebnisse in eine interne Tabelle (Workbereich mit DATA definiert) oder in einem Satz (Workbereich mit DATA definiert).
  Erfolgreiche Select-Zugriffe werden mit dem Wert 0 in der Systemvariablen **SY-SUBRC** quittiert.
  Für die Änderungen von Datenbanktabelleninhalten gibt es die Open SQL-Anweisungen. Beachten Sie allerdings, dass diese Anweisungen weder Berechtigungen noch die Datenkonsistenz überprüfen:

- **INSERT:** Mit der Anweisung INSERT können Sie Zeilen in eine Datenbanktabelle einfügen. Der Arbeitsbereich (satz_tabelle) sollte die gleiche Struktur wie die Datenbanktabelle haben. Die Operation kann nur dann durchgeführt werden, wenn die Datenbanktabelle noch keinen Eintrag mit dem

gleichen Primärschlüssel hat. Ansonsten wird in sy-subrc der Fehlercode 4 eingestellt.

Einfügen von einem Satz:

```
INSERT dbtabelle FROM satz.
```

oder

```
INSERT INTO dbtabelle VALUES satz.
```

Einfügen über eine interne Tabelle:

```
INSERT dbtabelle FROM TABLE int_tabelle.
```

Der Zusatz **ACCEPTING DUPLICATE KEYS** verhindert den Abbruch bei doppeltem Primärschlüssel

- **UPDATE**: Mit Update können einzelne Zeilen in einer Datenbanktabelle geändert werden. Dabei können einzelne Spalten als auch die gesamte Zeile geändert werden.

  Änderung einer Tabellenzeile:

  ```
  UPDATE dbtabelle FROM satz.
  ```

  Änderung von einzelnen Spalten:

  ```
  UPDATE dbtabelle SET    db_feld1 = satz_feld1
                          db_feld2 = satz_feld2
         WHERE ....
  ```

  Änderung mehrerer Zeilen:

  ```
  UPDATE dbtabelle FROM TABLE int_tabelle.
  ```

- **MODIFY**: MODIFY fasst UPDATE und INSERT zusammen. Ist noch keine Zeile mit dem Primärschlüssel vorhanden wird eine Zeile eingefügt, ansonsten wird die Zeile aktualisiert. Nachteil von MODIFY ist die gegenüber INSERT bzw. MODIFY schlechtere Performance.

  Modifizieren von einem Satz:

  ```
  MODIFY dbtabelle FROM satz.
  ```

  Einfügen über eine interne Tabelle:

  ```
  MODIFY dbtabelle FROM TABLE int_tabelle.
  ```

- **DELETE**: Eine Zeile in einer Datenbanktabelle kann mit DELETE gelöscht werden. Die Auswahl der Zeile kann entweder über die WHERE-Klausel erfolgen oder es kann der Arbeitsbereich verwendet werden.

  Löschen mit der WHERE-Klausel

  ```
  DELETE FROM dbtabelle WHERE feld1=100.
  ```

  Löschen über den Arbeitsbereich

  ```
  DELETE dbtabelle FROM satz.
  ```

  Löschen über eine interne Tabelle

  ```
  DELETE dbtabelle FROM TABLE int_tabelle.
  ```

  Kontrolle der Operation mit folgendem Befehl.

  ```
  IF sy-subrc NE 0.
  ROLLBACK WORK.
  WRITE: / 'Fehlermeldung'.
  ENDIF.
  ```

Mengenorientiertes Auslesen von Datensätzen:

Allgemeine Struktur: **SELECT** <result> **FROM** <tabelle> "Welche Tabelle ?
INTO <target> "Wohin ?
WHERE <condition> "Welche Zeilen ?

Alle Sätze und Felder einer Tabelle lesen:

```
SELECT * FROM tabelle INTO workbereich.
…  Anweisungen
ENDSELECT.
```

Alle Sätze und einzelne Felder einer Tabelle lesen:

```
SELECT feld1 feld2 feld3 FROM tabelle INTO workbereich.
  Anweisungen
ENDSELECT.
```

Sätze mit Ausgrenzung lesen:

```
SELECT  * FROM tabelle INTO workbereich
WHERE feld1 = "XZY".
  Anweisungen
ENDSELECT.
```

Einzelsatzlesen

Einzelsatzlesen d.h. es lassen sich einzelne Sätze aus der Datenbanktabelle lesen. Mit der CORRESPONDING FIELDS OF der INTO-Klausel füllen Sie den Zielbereich Feldweise, wobei nur die Felder berücksichtigt werden, die namensgleich sind.

- **SELECT SINGLE * FROM** tabelle **INTO** workbereich**.**
- **SELECT SINGLE * FROM** tabelle **INTO CORRESPONDING FIELD OF** workbereich **WHERE** feld1 = 100 and feld2 = 200**.**

Datenbankinhalte in eine interne Tabelle

Datenbankinhalte können auch leicht in interne Tabellen eingelesen werden.

SELECT * FROM tabelle **INTO TABLE** interne_tabelle**.**

VIEWS und JOINS

- **Statische Verknüpfung (VIEWS)** = die statische Verknüpfung kann im ABAP-Dictionary definiert werden und wird als VIEW bezeichnet. Es gibt verschiedene Arten von VIEWS. Detailinformationen entnehmen Sie bitte der SAP-Bibliothek unter Basis – ABAP Workbench – BC ABAP Dictionary – Views.

```
DATA: BEGIN OF ausgabetabelle,
      feld1  LIKE tabelle1-feld1,
```

```
            feld2  LIKE tabelle1-feld2,
            feld3  LIKE tabelle2-xyz1,
            END OF ausgabetabelle.
            SELECT * FROM testview1
          INTO CORRESPONDING
      FIELDS OF ausgabetabelle
          ORDER by feld1.
          WRITE: / ausgabebereich.
        ENDSELECT.
```

- **Dynamische Verknüpfung:** Die dynamische Verknüpfung können Sie über ABAP-Anweisungen implementieren und wird als ABAP-Join bezeichnet. Zur Laufzeit wird in der Datenbankschnittstelle eine entsprechende Datenbankabfrage generiert. Detailinformationen entnehmen Sie bitte der Schlüsselwortdokumentation zur SELECT – FROM-Klausel.

```
DATA: BEGIN OF ausgabetabelle,
      feld1  LIKE tabelle1-feld1,
      feld2  LIKE tabelle1-feld2,
      feld3  LIKE tabelle2-xyz1,
     END OF ausgabetabelle.
SELECT tab1~feld1 tab1~feld2 tab2~xyz1
INTO CORRESPONDING FIELDS OF ausgabetabelle
FROM tabelle1 AS tab1
INNER JOIN tabelle2 AS tab2
ON tab1~feld1 = tab2~feld1.
WRITE: / ausgabetabelle.
ENDSELECT.
```

Beim Inner-Join müssen dringend die 'Schlangenlinien' als Trenner zwischen Tabellenname und Feldname verwendet werden. Wenn man das vergisst erzeugt das meistens keinen Syntax-Fehler, aber das Ergebnis des Selects ist fehlerhaft.

Inner Joins gehen natürlich auch als 'select single':

```
SELECT SINGLE tab1~feld1 tab1~feld2 tab2~feld1 FROM tab1
INNER JOIN tab2 ON tab2~feld1 = tab1~feld1
INTO CORRESPONDING FIELDS OF wa
WHERE  tab1~mandt  = sy-mandt.
```

Der doppelte INNER JOIN [über mehrere Tabellen]:

```
      SELECT  tab1~feld_01
              tab2~feld_01
              tab2~feld_02
              tab3~feld_01
      FROM tab1
      INNER JOIN tab2    ON   tab2~feld_01 = tab1~feld_01
      INNER JOIN tab3    ON   tab3~feld_01 = tab1~feld_01
                         AND   tab3~feld_02 = tab2~feld_02
      INTO CORRESPONDING FIELDS OF TABLE wa
        WHERE tab1~feld 01 = '100'
*---- Select-Option Tabelle
          AND tab1~feld_02 IN s_feld_02
```

Schachtelung von SELECT-Befehlen.

ACHTUNG: Es ist auch möglich eine Tabelle mit SELECT zu lesen und eine zweite Tabelle mit SELECT SINGLE dazu zulesen. Dies ist allerdings eine für Massendaten ungeeignete Zugriffsform, weil sie schlechte Performance bietet.

```
DATA: BEGIN OF ausgabetabelle,
    feld1  LIKE tabelle1-feld1,
    feld2  LIKE tabelle1-feld2,
    feld3  LIKE tabelle2-xyz1,
    END OF ausgabetabelle.
SELECT feld1 feld2
INTO CORRESPONDING FIELDS OF ausgabetabelle.
*        *-- XYZ-key der Tabelle2 entspricht dem tabelle1-feld1
```

Und

```
*-- wird zur Ausgrenzung verwendet.
SELECT SINGLE xyz1 FROM tabelle2 WHERE xyz-key = feld1.
    WRITE: / ausgabebereich.
ENDSELECT.
```

LOGISCHE Datenbanken

- **Logische Datenbanken**
  ACHTUNG: Falls wiederverwendbare Komponenten vorhanden sind, die kompliziertere Datenbeschaffung kapseln, sollten diese Komponenten verwendet werden. Es stehen vier Techniken zur Verfügung
  - ⊗ Methoden von globalen Klassen aufrufen
  - ⊗ Methoden von Business-Objekten aufrufen,
  - ⊗ Funktionsbausteine aufrufen,
  - ⊗ Logische Datenbanken einbinden.

Logische Datenbanken sind Datenbeschaffungsprogramme, die logisch zusammengehörende Daten in hierarchischer Reihenfolge liefern. Detailinformationen zur Einbindung von logischen Datenbanken entnehmen Sie bitte der SAP-Bibliothek BASIS – ABAP Programmierung und Laufzeitumgebung – ABAP Datenbankzugriffe – Logische Datenbanken.

- **GET** = Datenbeschaffung über logische Datenbank z.B. ***GET SFLIGHT***. Vorteile der logischen Datenbank sind: keine Kenntnis des Entwicklers über Schlüsselfelder usw. notwendig, Bessere Performance, ein Selektionsbildschirm wird automatisch erzeugt und muss nicht im Report definiert werden, Berechtigungsprüfungen müssen nicht berücksichtigt werden. Eine Änderung in der logischen Datenbank wirkt sich auf alle Programme, welche die logische Datenbank nutzen, aus. Die Ablauflogik in einem Programm mit einer logischen Datenbank ist:

START-OF-SELECTION
Anweisung
GET datentabelle

(Alternativ: **GET** datentabelle datenfeld1 datenfeld2 datenfeld3**).**
Anweisungen
END-OF-SELECTION.
Anweisung (Wird aufgerufen nach dem letzten GET-Ergebnis)

Umgang mit dem Data Dictonary

- **Deklaration von DB-Tabellen** = Datenbanktabellen werden grundsätzlich im DD mit allen Schlüsseln und Attributen definiert. (Detailanweisung folgt)

2.4.12 MESSAGE

Die Nachrichten sind notwendig, um Fehlermeldungen usw. am Bildschirm für Dialogprogramme anzuzeigen. Zur Pflege der Nachrichten steht die Transaktion SE91 zur Verfügung. Die Nachrichten können nach folgenden Klassen unterschieden werden.

Typ = t	**Darstellung**	**Verarbeitung**
A	Dialogfenster	Programmabbruch
E	Statuszeile	Abbruch Verarbeitungsblock
I	Dialogfenster	Programmfortsetzung nach Ausgabe
S	Statuszeile	Programmfortsetzung nach Ausgabe
W	Statuszeile	Wie E
X	Keine	Laufzeitfehler mit Kurzdump

MESSAGE = Aufruf von Standardfehlermeldungen (Aufruf über MUSTER)

Die Anlage einer Message erfolgt durch Vorwärtsnavigation. Für die Anlage einer eigenen Nachrichtenklasse verwenden Sie einen Namen im Kundennamensraum Y oder Z. Um eine Nachricht anzulegen, verwenden Sie dreistellige Nummern und die entsprechende Nachrichtenklasse. Innerhalb des Nachrichtenkurztextes können Sie bis zu vier (4) Übergabeparameter einsteuern. Dies erfolgt mit dem Platzhalter „&1“ , „&2“ usw.. Beispiel: „Satz &1 konnte nicht angelegt werden“. Die Nachrichten stehen in der Systemtabelle T100.

Im ABAP-Programm müssen Sie folgende Programmzeile erfassen:

MESSAGE tnnn **(**klasse**) (WITH** f1...f4**).** „f1 bis f4 sind Übergabevariable
(Beispiel: MESSAGE i014.)
oder

MESSAGE ID klasse **TYPE t NUMBER** nnn **(WITH** f1...f4**).**

Eine Nachrichtenklasse kann auch für ein Programm hinter dem Befehl REPORT festgelegt werden.

REPORT reportname **MESSAGE-ID** klasse**.**

2.4.13 Berechtigungsschutz

AUTHORITY-CHECK = SAP verfügt über ein komplexes Berechtigungs- und Rollensystem zur Verhinderung von nicht zugelassenen Daten- und Programmzugriffen. Mit dem Schlüsselwort AUTHORITY-CHECK und entsprechender Übergabeparameter werden die Berechtigungen geprüft und in der Systemvariablen SY-SUBRC zurückgegeben.

Beispiel für Zugriff auf Datenbanksatz und deren Berechtigung:

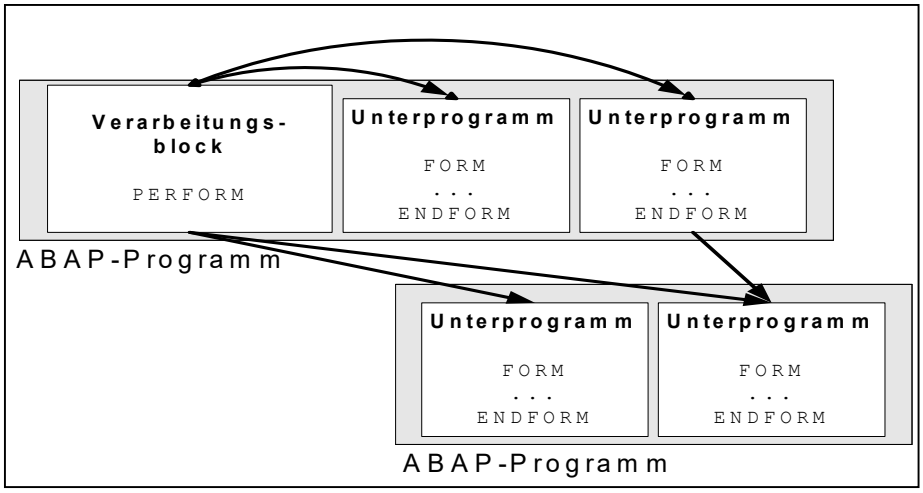
```
AUTHORITY-CHECK OBJECT ´objektname`
*---- Objektname ist z.B. ein Datenfeld im DD
        ID ´tabellenname´     FIELD ´feldname´
        ID ´ACTVT´            FIELD         ´02´.
*--- Rückgabe von sy-subrc: 0=Berechtigt
        IF sy-subrc ne 0.
              Anweisung
        ENDIF.
```

ACTYT = Kennung, die Art der Aktivität auf die Datenbank. Die Arten der Aktivitäten sind 01=Anlegen, 02=Ändern, 03=Anzeigen, 04=Löschen

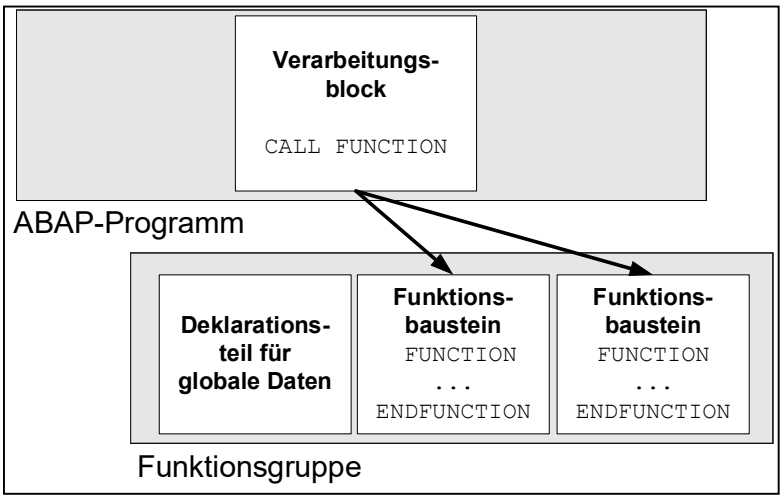
2.4.14 Modularisierung

Zur Modularisierung von ABAP-Programmen stehen verschiedene Konzepte zur Verfügung

- **PERFORM/FORM**: Unterprogramme als lokale Prozeduren, in jedem ABAP-Programm verwendbar. Die Daten können global oder lokal vereinbart werden. Es gibt verschiedene Arten von Parameterschnittstellen. Nachfolgendes Bild zeigt die Grundlogik:



- **Funktionsbausteine**: Funktionsbausteine sind eigenständig arbeitende Programmteile, die eine globale Funktionalität zur Verfügung stellen. Zum erzeugen einer Funktion steht der Function Builder zur Verfügung.



- **INCLUDE**: Quelltext-Module bedeutet, dass Teile des Quellcodes in sogenannte INCLUDE-Programme ausgelagert werden können. Diese Quellcodeteile sind nur verwendbar als Programmcodeauslagerung.
- **KLASSEN**: Klassen sind ebenfalls möglich und ersetzen immer mehr die Funktionsbausteine.

PERFORM/FORM Unterroutinen innerhalb eines Programms

Info: Form als Unterprogramm soll nicht mehr verwendet werden → OO ist das Thema

- **PERFORM/FORM** = Aufruf von Unterroutine mit Variablenübergabe mit **USING** oder **CHANGING**. Es funktioniert auch ohne Angabe von USING oder CHANGING. Dies sollte jedoch im Hinblick auf spätere Objektorientierung des Programms nicht verwendet werden.

Allgemeine Struktur: **PERFORM** unterprog **[USING** par1 par2….**]**
[CHANGING par1 par2….**].**

```
                                    (par1, par2 … =
                                    Aktualparameter)
          FORM unterprog        [USING {VALUE} par1 par2]
                                [CHANGING {VALUE} par1 par2]
                                (par1,par2… = Formalparameter)
                                [datentypisierung]

          ENDFORM.
```

```
PERFORM  XYZ USING feld1
….
FORM XYZ USING feld_a.
```

Formen von Empfangsparameterdefinitionen:

```
FORM XYZ USING  feld_a              .
        = Vererbung des Typs des Aktualparameters an den
          Formalparameter.
FORM XYZ USING feld_a TYPE ANY
        = Typisierung bleibt offen
FORM XYZ USING feld_a TYPE stru_feld_a.
        = Typisierung z.B durch eine Struktur, Tabelle oder Datenfeld
             WICHTIG: Typkonformität ist wichtig.
```

Die Art und Weise, wie Daten des Hauptprogramms (Aktualparameter) den Datenobjekten des Unterprogramms (Formalparameter) übergeben werden, wird in der Schnittstelle festgelegt. Es gibt drei Möglichkeiten

- **CALL-BY-VALUE** = Dem Unterprogramm soll eine lokale Kopie des Aktualparameters übergeben werden. Das bedeutet, dass sich die Wertzuweisungen an den Formalparameter überhaupt nicht auf den Aktualparameter auswirken. Eignung nur für den Import von Daten aus dem Hauptprogramm in das Untermodul. Call-by-Value wird erzielt, indem man eine solche Variable mit USING parameter übergibt und im Unterprogramm mit USING VALUE (parameter) empfängt.

```
PERFORM unterprogramm        USING a1.
FROM unterprogramm           USING VALUE (b1).
```

- **CALL-BY-REFERENZ** = Dem Unterprogramm wird die aktuelle Adresse des Aktualparameters übergeben. Das bedeutet, dass sich Wertezuweisungen an den Formalparameter direkt auf den Aktualparameter auswirken. Haupt- und Unterprogramm verwenden den gleichen Speicherbereich. Eignung für den Import und Export von Daten aus dem Hauptprogramm in das Unterprogramm und zurück. Call-by-Referenz wird erzielt, indem man einen Übergabeparameter unter CHANGING oder USING aufführt. CHANGING und USING haben die gleiche Bedeutung.

```
PERFORM unterprogramm        CHANGING  a1.
FROM unterprogramm using     CHANCING  b1.
```

oder mit gleicher Bedeutung

```
PERFORM unterprogramm        USING  a1.
```

FROM unterprogramm using **USING** b1**.**

- **CALL-BY-VALUE-AND-RESULT** = Dem Unterprogramm soll eine lokale Kopie des Aktualparameters mit Wertrückgabe übergeben werden. Das bedeutet, dass sich Wertzuweisungen an den Formalparameter erst beim Verlassen des Unterprogramms auf den Aktualparameter auswirken. Diese Übergabe sollten Sie wählen, wenn Sie sicherstellen wollen, dass bei einem vorzeitigen Abbruch des Unterprogramms die Aktualparameter noch nicht verändert worden sind.

  PERFORM unterprogramm **CHANGING** a1**.**
  FROM unterprogramm **CHANGING VALUE** (b1).

Funktionsbaustein zur Modularisierung von ABAP-Programmen.

- **CALL FUNKTION** = Aufruf von Funktionsbausteinen
  CALL FUNCTION ´funktionsbaustein´
  EXPORTING übergabevariablen "Übergabe vom Hauptprogramm in den F-Baustein
  IMPORTING rückgabevariablen "Rückgabe vom F-Baustein in das Hauptprogramm
  EXCEPTION statusvariable. "Mit Raise gefangener Fehler aus dem F-Baustein

  ***---- Auswertung des Fehlerstatus in sy-subrc**
  IF SY-SUBRC = 1
  WRITE ´Fehlerhafter Funktionsaufruf´.
  ENDIF.

ACHTUNG:
Für die Integration eines Funktionsaufrufes in Ihr Programm verwenden Sie immer die MUSTER Funktion. Der Erfolg eines Funktionsaufrufs wird über die Systemvariable SY-SUBRC = 0 (0=Erfolgreich) abgefragt. Die Erstellung einer Funktion wird mit Hilfe des Funktionsbilders SE37 Function Builder durchgeführt.

Die wichtigsten Parameter eines Funktionsbausteines sind:

- **Import-Parameter**: Dies sind Formalparameter, über die ein Aufrufer Eingabeparameter an den Funktionsbaustein übergeben kann. Die Import-Parameter können als Optional gekennzeichnet werden. Optionale Parameter können einen Startwert zugeordnet bekommen, falls der Aufrufer keinen Übergabewert mitgibt.

- **Export-Parameter**: Dies sind Formalparameter, über die ein Aufrufer die Rückgabewerte eines Funktionsbausteines im Aktualparameter seines Programms übernehmen kann. Die Entgegennahme von Export-Parametern ist optional.

- **Changing-Parameter**: Dies sind Formalparameter, die als Import- und Export-Parameter dienen. Die Übergabe erfolgt vom aufrufenden Programm und wird an das aufrufende Programm zurückgegeben.

Die Typisierung der Parameter erfolgt über das Data-Dictionary.

- **Ausnahmefehler**: Ausnahmefehler werden über EXCEPTION ausgetauscht und über RAISE „gefangen“.

  Die Anlage und Verwaltung eines Funktionsbausteins erfolgt mit dem Funktionsbuilder. Um Funktionsbausteine zu erfassen wird zunächst eine Funktionsgruppe im Objekt-Navigator angelegt, anschließend wird der Funktionsbaustein angelegt. Der Funktionsbuilder ist in der Weise ausgestaltet, dass mit Hilfe der TAB-Laschen im Programm, die notwendigen Angaben gemacht werden können.

SAP R/3 verfügt über ein vielzahl unterschiedlicher Methoden um Dienste zu erfüllen Im Internet unter der Adresse http://www.geocities.com/victorav15/sapr3/abapfun.html sind solche Funktionsbausteine aufgelistet (Stand 1.Juni 2006).

INCLUDE name. Ein Include dient der Auslagerung von Sourcecodeteilen

CLASS-Konzept:

siehe 2.4.17

2.4.15 DYNPRO-Programmierung - Aufruf von Masken

Bildschirmstandardmasken

Ein Dynpro ist nichts anderes als eine Dialogmaske.

- **CALL SCREEN ####. (4-stellige Nummer)** = Aufruf von Dynpros. Voraussetzung für das Befüllen von Dynpros aus der Anwendung ist die TABLES Anweisung. TABLES erzeugt eine Instanz, um Daten an das Dynpro zu senden. Mit der Vorwärtsnavigation können nicht angelegte Dynpros erzeugt werden. Dynpros werden über eine vierstellige Nummer identifiziert.

- **TABLES dbtabelle** = Für Dynpros wird die Instanz vorzugsweise mit TABLES dbtabelle erstellt. TABLES wirkt dann wie DATA und dient als Kommunikationsbereich zu den DYNPROS. Mit der Tablesanweisung werden Kommunikationsbereiche (workbereiche) für das Dynpro erzeugt. Dies ist in zukünftigen ABAP-Standards der einzige Anwendungsfall für die Anweisung TABLES. Hinter dem Schlüsselwort TABLES kann eine DB-Tabelle oder DD-Struktur stehen. Mit der Anweisung TABLES wird eine Instanz für das Führen von Dynpro-Workbereichen geschaffen. Daten in diesem Bereich werden durch MOVE oder MOVE-CORRESPONDING geschoben. Z.B. **TABLES sflight.**

- **MODULE** name **OUTPUT** Anweisungen **ENDMODULE**. = Unterprogramm für das PBI

- **MODULE** name **INPUT**. Anweisungen **ENDMODULE**. = Unterprogramm für das PAI.

- **LEAVE TO SCREEN** nummer = Innerhalb der MODULE kann auf eine andere Maske gesprungen werden.

- **DATA:** ok_code **LIKE** sy-ucomm = Notwendige Vereinbarung einer Variablen, um mit einem Dynpro Events auszutauschen. Im ok_code können z.B. in der Maske mit Funktionstasten oder über die Buttonzeile alphanumerische CODES ausgetauscht werden. Diese Codes müssen **immer großgeschrieben werden**. Beispiel für ein solchen Rückgabecode kann ein Button Speichern auf der Dynpromaske sein, der den Wert „SAVE“ zurückgibt. Innerhalb der MODULE (PAI) muss dieser Wert ausgewertet werden und eine entsprechende Programmierung erfolgen z.B. Speicherung des Datensatzes auf Platte.

SUBMASKEN

Innerhalb eines Dynpros können Submasken angelegt werden. Hierzu ist es notwendig auf die Hauptmaske einen Bereich für die Submaske zu reservieren. Diese auf der Hauptmaske angelegte Submaske erhält einen eindeutigen Namen (so wie eine Variable) , der in der Ablauflogik des Dynpros im PBO mit folgendem Befehl für die Submaske vorbereitet wird:

```
CALL SUBSCREEN name_des_subcreens INCLUDING
'programmname_des_hauptprogramms' 'maskennummer_des_subscreens'.
```

und im PAI mit

```
CALL SUBSCREEN SUBTEST.
```

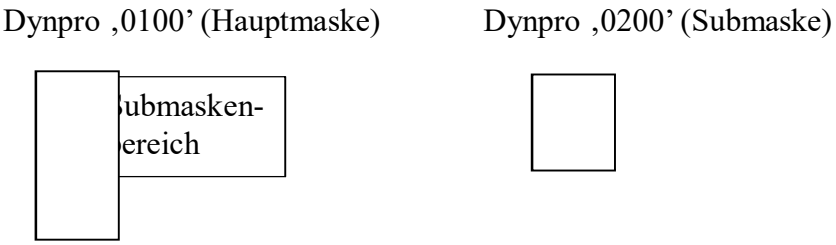
nachbearbeitet wird. Die Stellung des Befehls in der Ablauflogik des Dynpros zeigt das folgende Beispiel:

Beispiel für ein Aufruf einer Submaske innerhalb der Hauptmaske:

```
PROCESS BEFORE OUTPUT.
      MODULE status_0100.
      CALL SUBSCREEN SUBTEST INCLUDING '
ZBLATT4_UEB2_01_DYNPRO_MASK'   '0200'.

PROCESS AFTER INPUT.

 MODULE user_command_0100.
 CALL SUBSCREEN SUBTEST.
```

Die in der Hauptmaske anzuzeigende Submaske wird wie die Hauptmaske als Dynpro entworfen, allerdings mit der Eigenschaft „Submaske“. Submasken können keinen

OK_CODE besitzen, sondern teilen' sich den OK_CODE mit der Hauptmaske. Eine sinnvolle Anwendung für Submasken ist, innerhalb einer Hauptmaske je nach Anwendungsfall unterschiedliche Submasken anzuzeigen.



Überschrift einer Dynpromaske

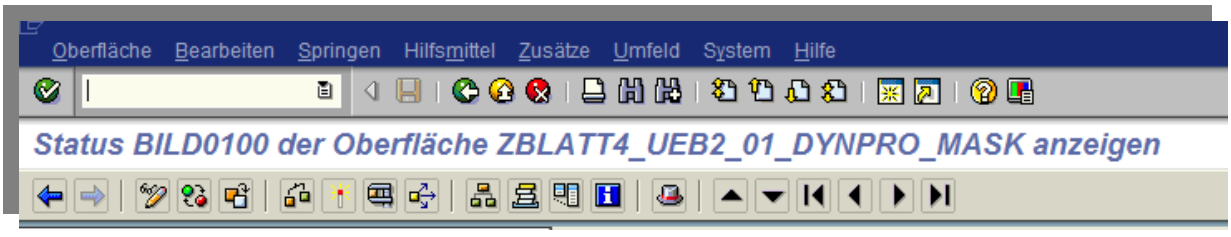
Um eine Überschrift über ein Dynpro zu erhalten, muss der Befehl **SET TITLEBAR** ‚name' in das „MODUL ‚name' OUTPUT" des Hauptprogramms eingefügt werden. Die Anlage der anzuzeigenden Überschrift erfolgt dann durch Vorwärtsnavigation. Sofern das „MODUL ‚name' OUPUT" mit Hilfe der Vorwärtsnavigation aus dem Screen-Painter angelegt wurde, generiert das Programm die SET TITLEBAR Zeile bereits als Kommentar in das Modul ein.

GUI-STATUS-ZEILEN

Um eine Menü-, Drucktasten- und Funktionstastenzeile anzulegen verfügt das Programm über einen eigenen Designer. Die Aktivierung der GUI-Statuszeile in einem Dynpro erfolgt mit dem Befehl

SET PF-STATUS ‚name'

in dem „MODUL ‚name' OUTPUT" des Hauptprogramms. Die Anlage der anzuzeigenden GUI-STATI erfolgt dann durch Vorwärtsnavigation. Sofern das „MODUL ‚name' OUPUT" mit Hilfe der Vorwärtsnavigation aus dem Screen-Painter angelegt wurde, generiert das Programm die SET PF_STATUS Zeile bereits als Kommentar in das Modul ein. Die Auswertung der gedrückten Tasten oder des Menüs erfolgt im PAI über den **OKCODE**.



Drucktastenleiste

ACHTUNG: Nicht nur bei der Dialogprogrammierung ist die Programmierung von GUI-STATUS-Zeilen möglich sondern auch bei der Reportprogrammierung. Bei der Reportprogrammierung können die gedrückten Tasten oder die ausgewählten Menüs im Ereignisbereich AT USER-COMMAND ausgewertet werden. Der Rückgabewert für die gedrückte Taste steht in der Systemvariablen sy-ucomm.

Beispiel für die Auswertung der gedrückten Tasten in einem REPORT:

```
AT USER-COMMAND.
CASE sy-ucomm.
     WHEN 'SELE'.
    ……..
     WHEN 'BACK'.
ENDCASE..
```

Selbstprogrammierte Eingabeüberprüfungen

Es ist häufig notwendig, auf ein Datenfeld in einem Dynpro eine Eingabeüberprüfung vorzunehmen bzw. mit Hilfe des gerade eingegebenen Feldes z.B. einen Datensatz zu lesen. Um eine Eingabeprüfung vorzunehmen stehen folgende Befehle zur Verfügung:

- **Ausführen des PAI nachdem man auf dem Dynprofeld gestanden hat und dieses verlässt.**
  FIELD dynprofeld **MODULE** modulname **ON INPUT.**
- **Aufruf des Moduls, wenn sich der Inhaltswert eines Dynprofeldes verändert.**
  FIELD dynprofeld **MODULE** modulname **ON REQUEST.**
- **Aufruf des Moduls, wenn das Feld verlassen wird**
  FIELD dynprofeld **MODULE** modulname**.**
- **Aufruf des Moduls, wenn eine Feldgruppe verlassen wird**
  CHAIN-INPUT (Alternativ: CHAIN oder CHAIN-REQUEST)
  FIELD dynprofeld1 **MODULE** modulname**.**
  FIELD dynprofeld“ **MODULE modulname ON INPUT.**
  ENDCHAIN.
  Die Eingabe der Befehlzeilen erfolgt im Sreen-Painter.

2.4.16 Aufruf von Programmen

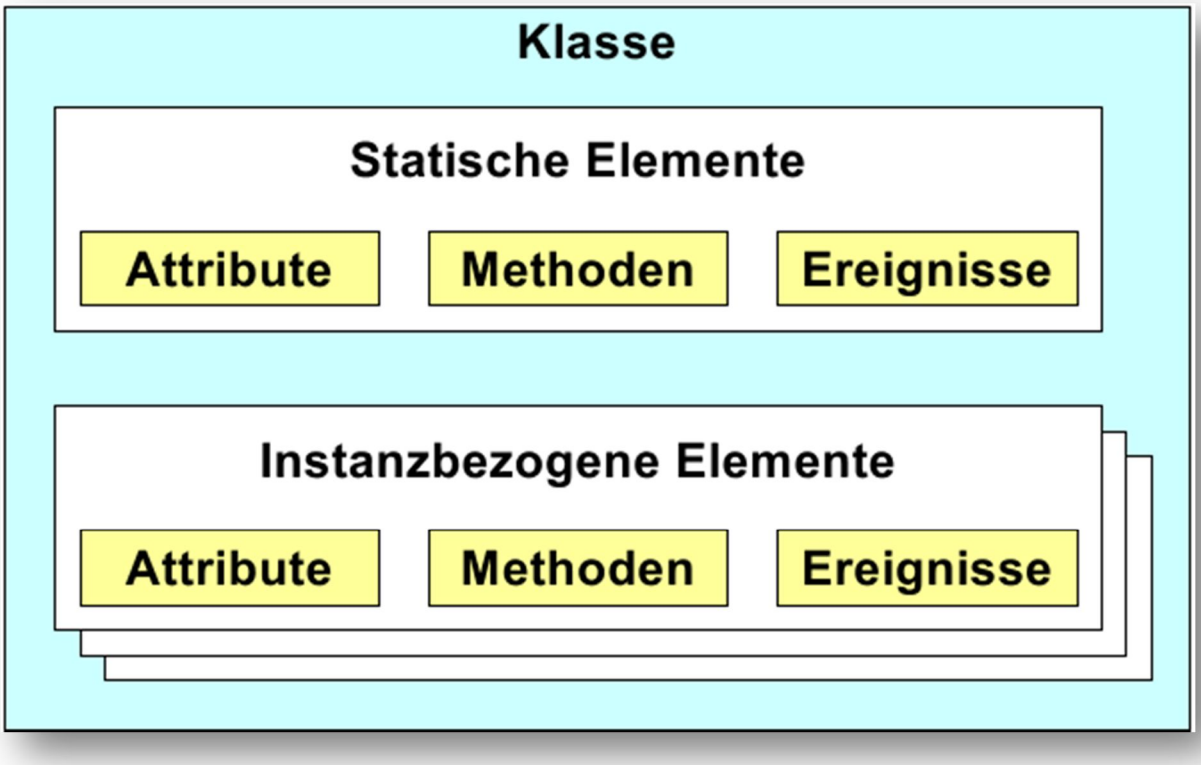
- Aufruf mit vollständigem Verlassen des aufrufenden Programms.
  - **SUBMIT** programmname oder
  - **LEAVE TO TRANSACTION** transaktionscode**.**
- Aufruf mit Rückkehr an die aufrufende Stelle
  - **SUBMIT** programmname **AND RETURN** oder
  - **CALL TRANSACTION** transaktionscode.
- Programm vollständig beenden
  - **LEAVE PROGRAM.**

2.4.17 Objektorientierte Programmierung

Zentrale Begriffe der objektorientierten Programmierung

- **Objekt**: Ein Objekt ist ein Objekt der realen Welt.

- **Klasse**: Eine Klasse ist die einmalige DV-technische Beschreibung der Eigenschaften und Funktionen eines Objekts.
- **Instanz**: Eine Instanz ist die softwaretechnische Repräsentation eines einzelnen Objekts zur Laufzeit in einem DV-System.
- **Attribute/Statische Attribute**: Ein instanzabhängiges Attribut beschreibt ein einzelnes Exemplar(Instanz) einer Klasse, ein statisches Attribut hingegen beschreibt eine Eigenschaft der Klasse an sich.
- **Methoden/Statische Methode**: Während die Attribute einer Instanz deren Eigenschaften beschreiben, beschreiben Methoden deren Verhalten. Eine Methode ist eine Funktion, die jede Instanz der Klasse ausführen kann. Eine statische Methode ist eine Funktion, die die Klasse selbst ausführen kann.



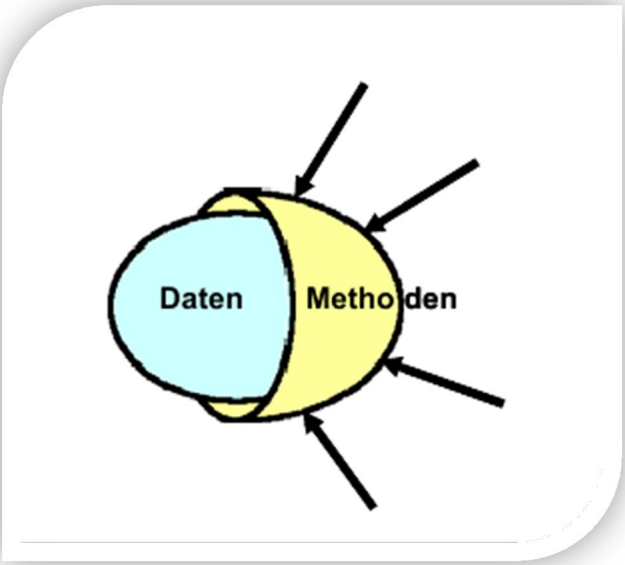
- **Konstruktor/Statischer Kontruktor:** Eine besondere Methode ist der Konstruktor. Er wird immer dann automatisch aufgerufen, wenn eine neue Instanz einer Klasse erzeugt werden soll, und sorgt so z.B. für die Initialisierung der Attribute – insbesondere, um obligatorische Werte zu versorgen. Ein Konstruktor besitzt daher nur Importparameter. Ein Konstruktor ist eine Methode, die immer dann aufgerufen wird, wenn eine Instanz der betreffenden Klasse neu erzeugt wird. Ein **statischer Konstruktor** wird genau einmal aufgerufen, wenn der erste Zugriff auf ein Element einer Klasse erfolgt.
- **Ereignis/Event:** Instanzen und Klassen können Ereignisse (Events) auslösen, andere Instanzen (sogar von anderen Klassen) und Klassen selbst können auf diese Ereignisse reagieren. Eine solche Möglichkeit gibt es in klassischen Programmiersprachen nicht. Bei der Deklaration einer Klasse wird

festgelegt, auf welche Ereignisse welcher Klassen die Klasse reagieren soll. Die Reaktion selbst ist immer eine Methode der reagierenden Klasse. Solche Methoden nennt man Event-Handler. Ereignisse können Attribute exportieren, die dann von den reagierenden Methoden abgefragt werden können.

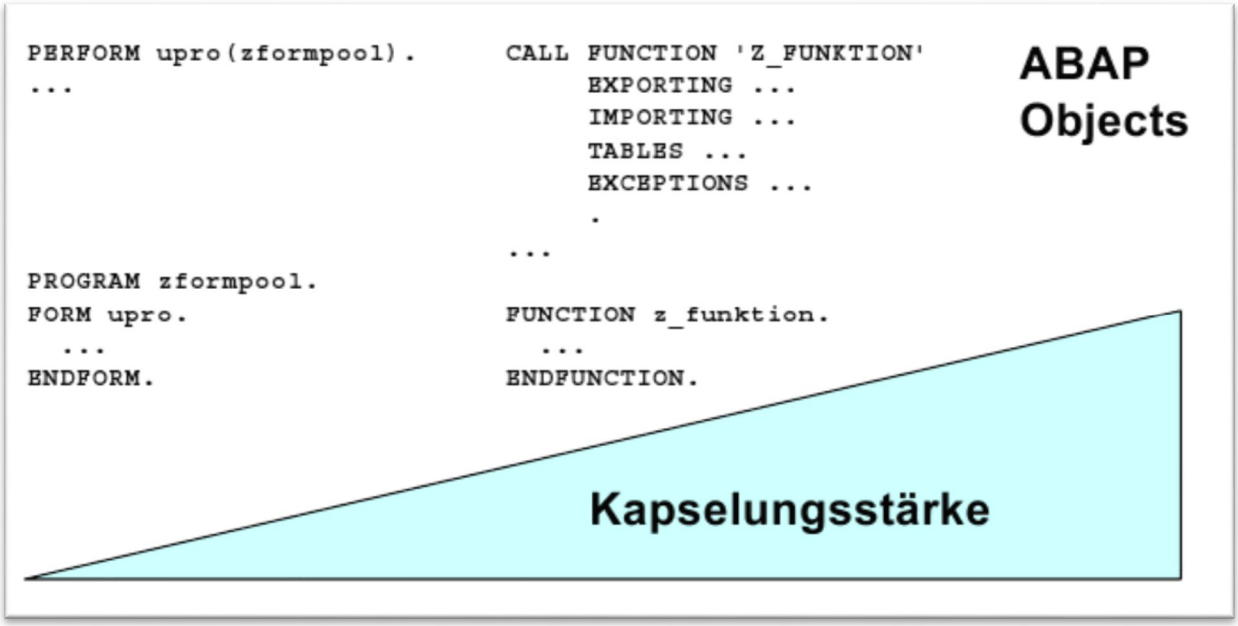
- **Kapselung**: Ein wichtiger Aspekt der OO-Technologie ist die Kapselung. Bereits bei der Beschreibung einer Klasse trennt man diese in zwei Teile:
  - Deklarationsteil
  - Implementationsteil

  Der Deklarationsteil stellt gewissermaßen deren Beschreibung – und damit ihre Schnittstelle – nach außen dar. Dort wird deklariert,
  - welche Attribute die Klasse/deren Instanzen besitzt/besitzen,
  - welche Methoden die Klasse/deren Instanzen ausführen kann/können,
  - welche Ereignisse die Klasse/deren Instanzen auslösen kann/können.

  Der Deklarationsteil ist derjenige Teil, den ein Programmierer betrachten soll, wenn er die Klasse benutzen möchte.Im Implementationsteil hingegen findet sich das P rogrammcoding, in dem die Methoden realisiert sind. Dieser Abschnitt sollte für den Verwender einer Klasse verborgen bleiben. Die eigentliche Laufzeit-Kapselung wird über die Mechanismen der Sichtbarkeit erreicht. Hierüber wird bestimmt, welche Elemente einer Klasse der (eventuell schädlichen) Umwelt offengelegt werden. Wenn ein Attribut z.B. PRIVATE und ein Zugriff nur über PUBLICMethoden erlaubt ist, so spricht man davon, dass die Klasse ihre Attribute über Methoden verschalt.



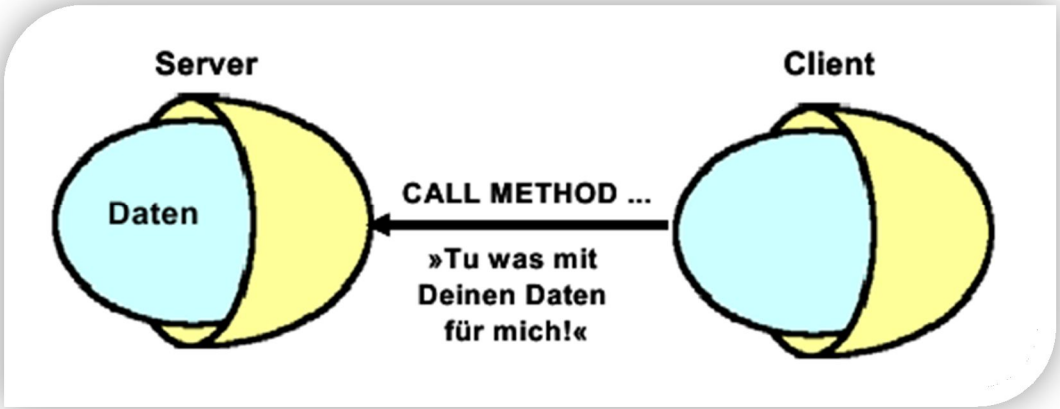
Die Stärke der Datenkapselung steigt von FORM über Funktionsbaustein zur Klasse.



- **Sichtbarkeit:** Die Möglichkeit von Attributen, Methoden und Ereignissen, sich teilweise zu verbergen, d.h. sich von außen (= von anderen Klassen) unverwendbar zu machen, wird bestimmt durch ihre Sichtbarkeit. Hierbei unterscheidet man drei verschiedene Stufen der Sichtbarkeit:

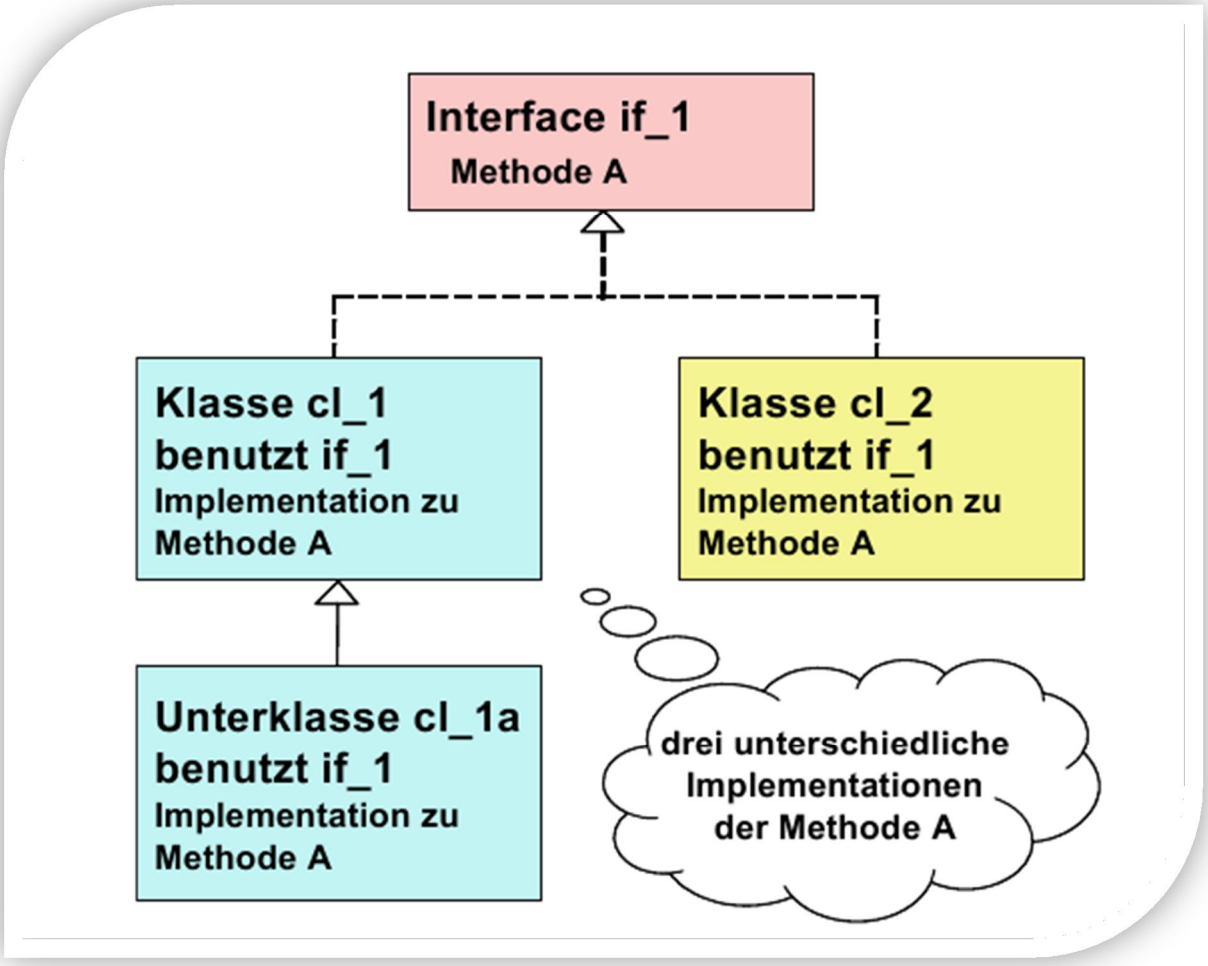
- **PUBLIC**: Die Klassenkomponente ist für alle Klassen sichtbar. Sie definiert damit die Schnittstelle der Klasse nach außen.
- **PROTECTED**: Die Klassenkomponente ist nur für die Klasse selbst und deren Erben (Verwender) sichtbar.
- **PRIVATE**: Die Klassenkomponente ist nur für die Klasse selbst sichtbar.

- **Delegationsprinzip:** Auf die Daten einer Instanz oder einer Klasse darf nur dasjenige Programmcoding zugreifen, das auch für die Korrektheit zuständig ist. Somit hat jede Klasse die alleinige Kontrolle über »ihre« Daten. Prinzip: Tu was mit Deinen Daten für mich !



- **Vererbung:** ABAP-Objekt erlaubt nur eine Einfachvererbung.

- **Interfaces**: ABAP Objects erlaubt lediglich die Einfachvererbung. Wenn nun mehrere Klassen mit gleichnamigen Methoden und auch noch gleichen Übergabeparametern zu realisieren sind, so müsste der Programmierer penibel darauf achten, dass die öffentlichen Deklarationen der Klassen identisch sind (die Implementationen sind natürlich unterschiedlich). Um dieses Problem zu umgehen, hat SAP ein neues Konstrukt, das Interface, eingeführt. Klassen können Interfaces deklarieren und dann über diese angesprochen werden. Aber auch andere Klassen können dasselbe Interface benutzen. Im Gegensatz zur Vererbung gibt es nicht nur mehrere Verwender der Methode A, sondern mehrere Implementierer.



- **Polymorphie:** Polymorphie (Vielgestaltigkeit) entsteht, wenn Klassen die gleiche Funktionalität mit unterschiedlichen Methoden implementieren. Dies kann über eine Vererbungsbeziehung geschehen, indem die Methode der Oberklasse in der Unterklasse mit demselben Namen und derselben Schnittstelle deklariert, aber unterschiedlich implementiert wird. Eine solche Doppeldeklaration nennt man Überdefinition.

Grundaufbau von Klassen

Definition einer Klasse

```
CLASS klassendefinition DEFINITION.
  PUBLIC SECTION.
*---- Definition von Klassenmethoden
    CLASS-METHODS: CONSTRUCTOR.
    CLASS-METHOD:  name …..
*---- Definition von Instanzmethoden
    METHODS: CONSTRUCTOR.
    METHODS: name   IMPORTING var TYPE i
          EXPORTING var(20) TYPE c
          CHANGING ....
  PROTECTED SECTION.
  PRIVATE SECTION.
    DATA: …
*---- Klassenvariable
    CLASS-DATA: ...
ENDCLASS.
```

Implementierung einer Klasse

```
CLASS klassendefinition IMPLEMENTATION.
*----- Implementierung der Constructormethode
  METHODE CONSTRUCTOR.
  ….
  ENDMETHODE.
*----- Implementierung der anderen Methoden
  METHODE name.
    DATA: …
    COMUTE ….
    SELECT …. usw.
  ENDMETHODE.
… usw.
ENDCLASS.
```

Instanzreferenzvariable

DATA: refvar TYPE REF TO klasse.

Aufruf von Methoden

Klassenmethode: Klassenname => Methodenname (variable).

Instanzmethode: CREATE OBJECT: Instanzreferenzvariable
EXPORTING ….

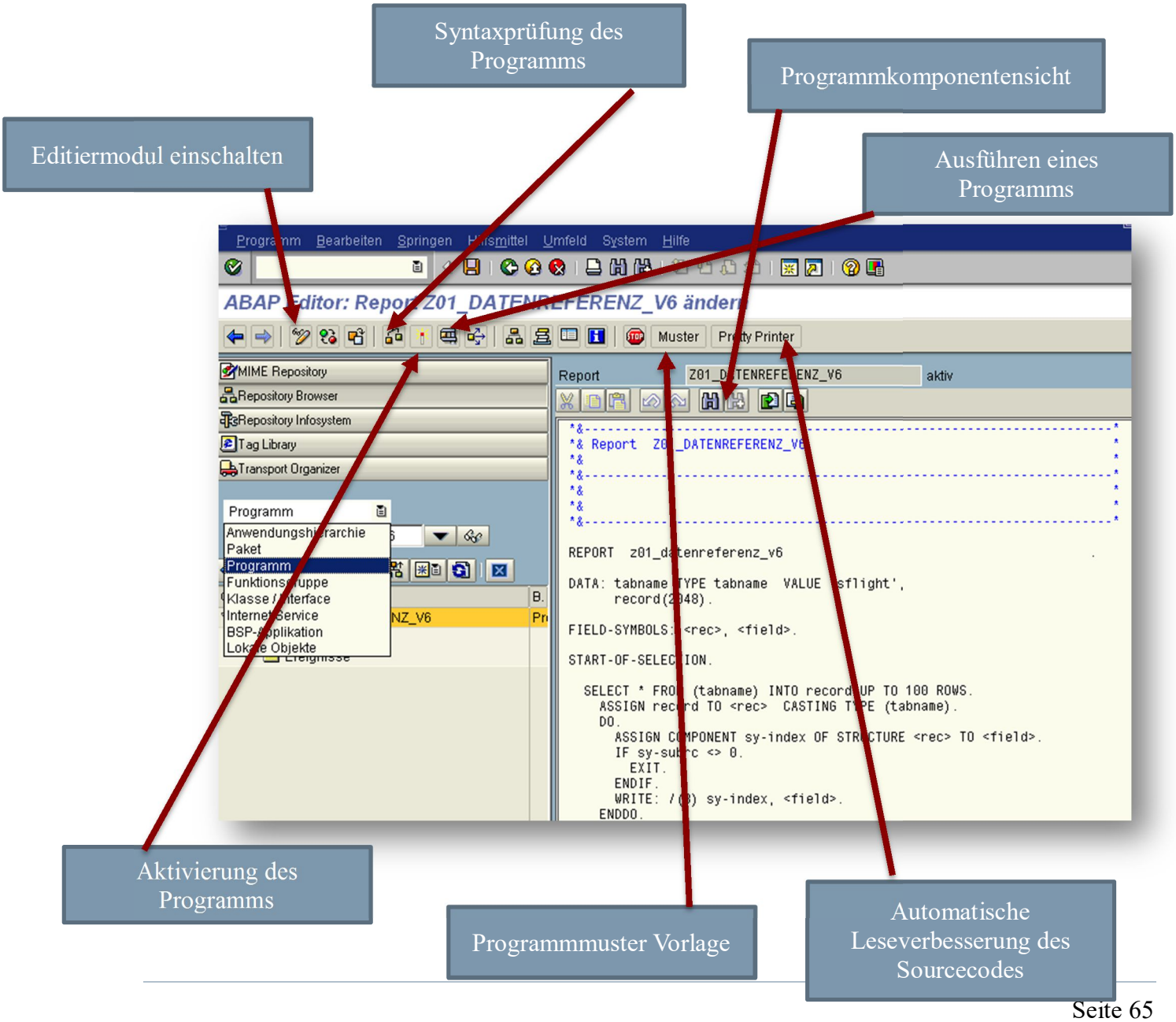
Kurzform: Instanzreferenzvariable ->Methodenname (variable).

Langform: CALL METHOD Instanzreferenzvariable->Methodenname
EXPORTING ….
IMPORTING …
CHANGING ….

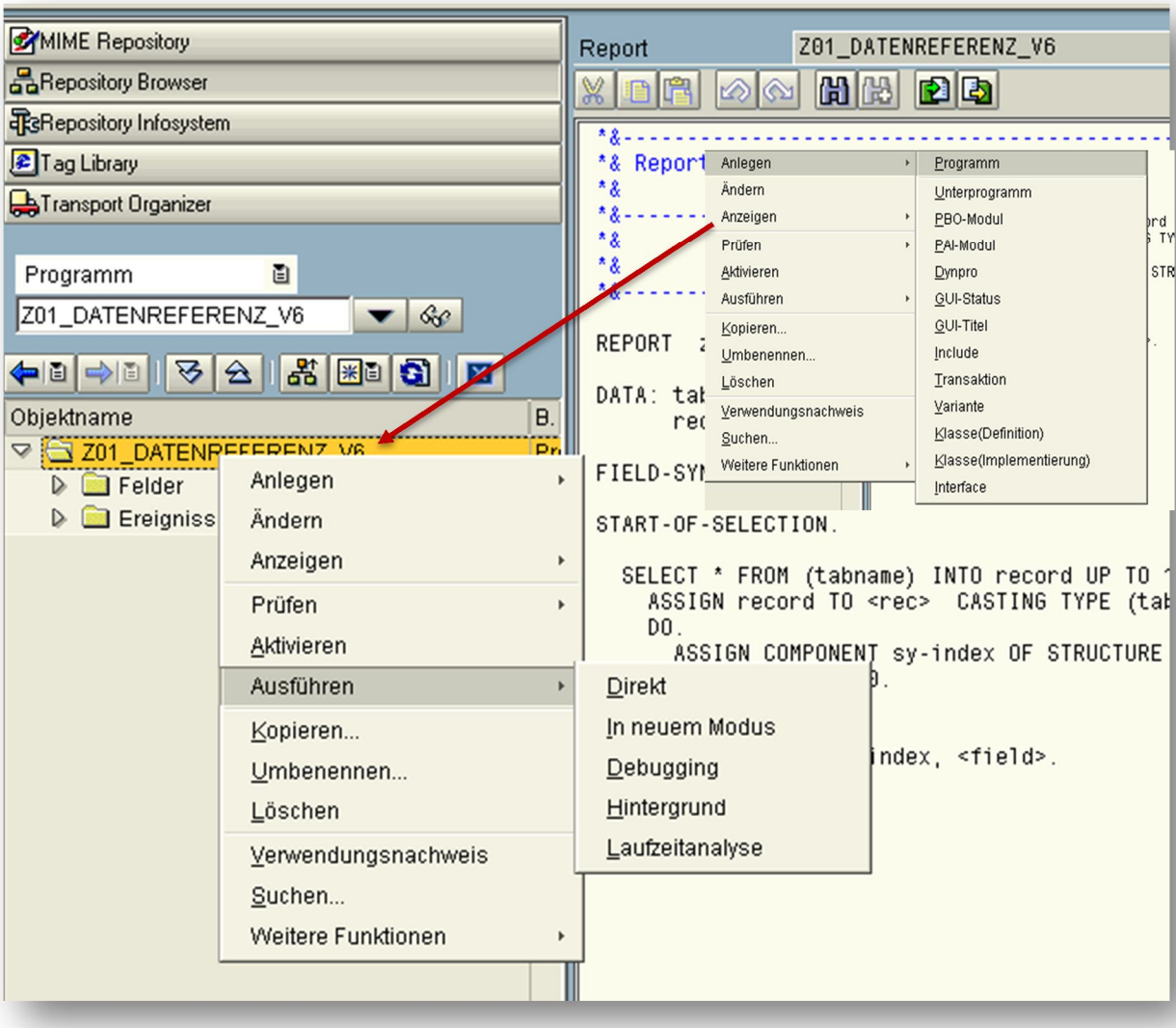
2.5. Bedienung

2.5.1 Object Navigator

Zur Entwicklung von ABAP-Programmen gibt es den Object Navigator. Der Object Navigator ist das Werkzeug mit dem die Objekte eines ABAP-Programms bearbeitet werden können. Solche Objekte sind z.B. Entwicklungspakete, ABAP-Programme, Einträge im ABAP-Dictionary, Transaktionen, GUI-Status, Funktionsgruppen, Funktionsbausteine, Masken, Klassen, Meldungen, Übersetzungen, Dokumentationen usw. Eine besondere Stärke des Object Navigators ist die sogenannte Vorwärtsnavigation. Mit Hilfe der Vorwärtsnavigation ist es möglich auf Details wie z.B. Tabellenbeschreibungen, Meldungen, Dynpros usw. zuzugreifen ohne den Sourcecode, der mit dem ABAP-Editor bearbeitet wird, zu verlassen. Im Folgenden sieht man einige wichtige Funktionen.



Weitere wichtige Funktionen sind mit einem rechten Mausclick zu erreichen. Hier können die Programme ausgeführt oder im Debugging-Mode ausgeführt werden. Programme können kopiert, gelöscht und umbenannt werden. Neue Objekte können angelegt werden.



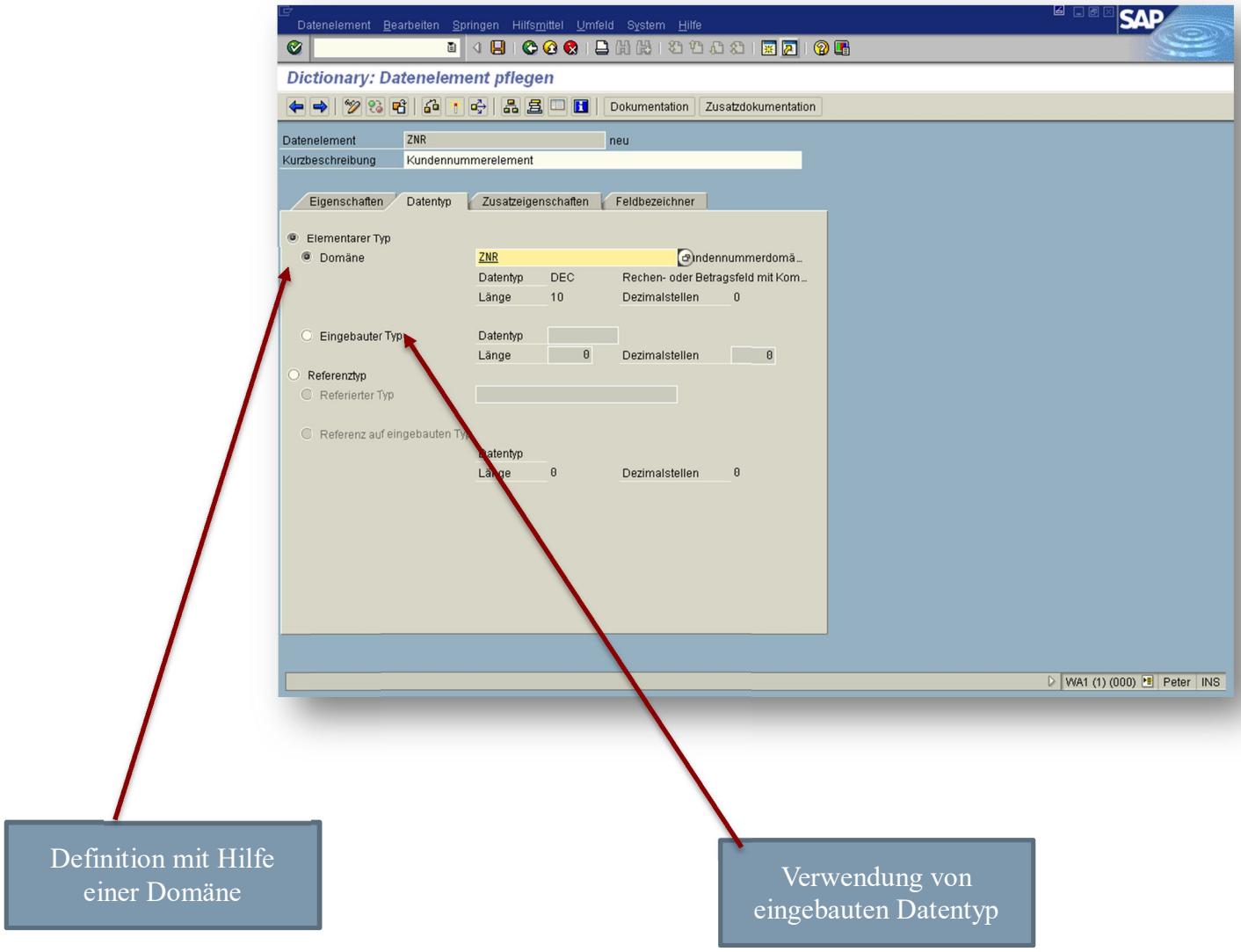
2.5.2 Data Dictionary (DD) (SE11)

Im ABAP-Dictionary werden alle notwendigen Informationen über die Datenbanktabellen datenbank- und plattformunabhängig gespeichert. Damit muss der Programmierer über keine Kenntnisse von konkreten Aufbewahrungsorten, Laufwerken usw. der Daten verfügen. Das ABAP-DD kann als ein Informationssammelbecken verstanden werden, deren Funktionalität über die eines üblichen relationalen Datenbanksystems hinausgeht. Wichtige Bestandteile des Data-Dictionary sind die Tabellen, Felder, Datenelemente und Domänen.

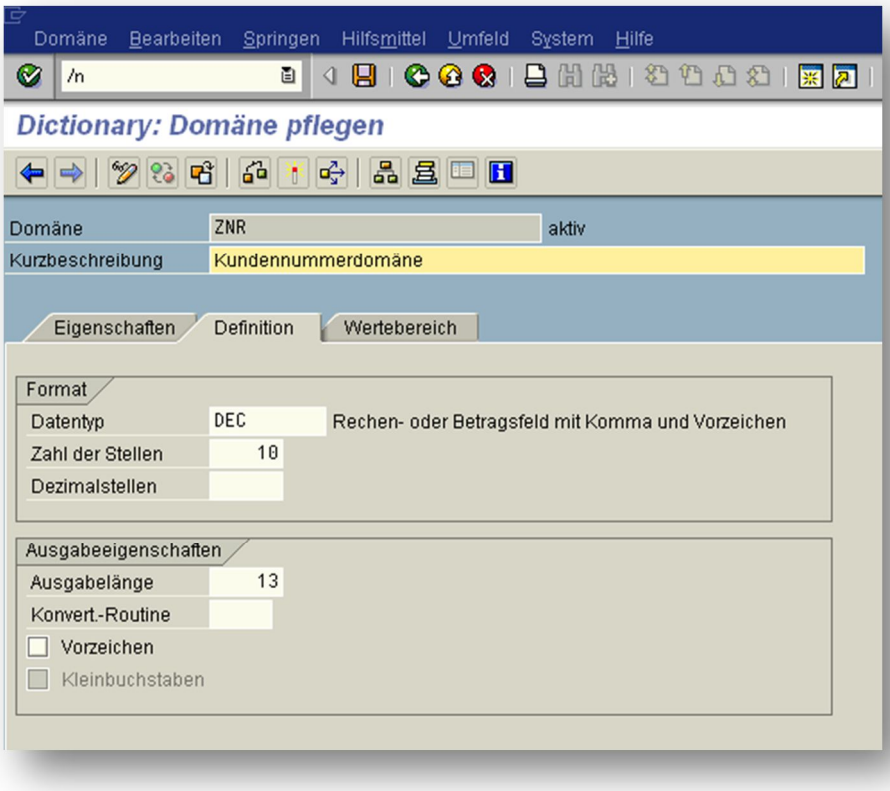
- **Tabellen**: Eine Tabelle umfasst mehrere Felder, die die Eigenschaften eines realen Objektes beschreiben z.B. Tabelle Kunden bestehend aus den Feldern Kundennummer, Kundenname und den Eigenschaften numerisch für Kundennummer und alphanumerisch für Kundenname.

- **Felder**: Felder sind die einzelnen Spalten einer Tabelle und haben technische Eigenschaften, die sie von den Datenelementen vererbt bekommen. ACHTUNG: Am Anfang einer SAP-Tabelle steht immer das Feld MANDT mit dem Datenelement MANDT

- **Datenelemente**: Die Datenelemente vererben den Feldern einer Tabelle ihre Eigenschaften. Für die Beschreibung eines Datenelements können direkt eingebaute Datentypen oder eine Domäne verwendet werden. Datenelemente bauen üblicherweise auf Domänen auf.



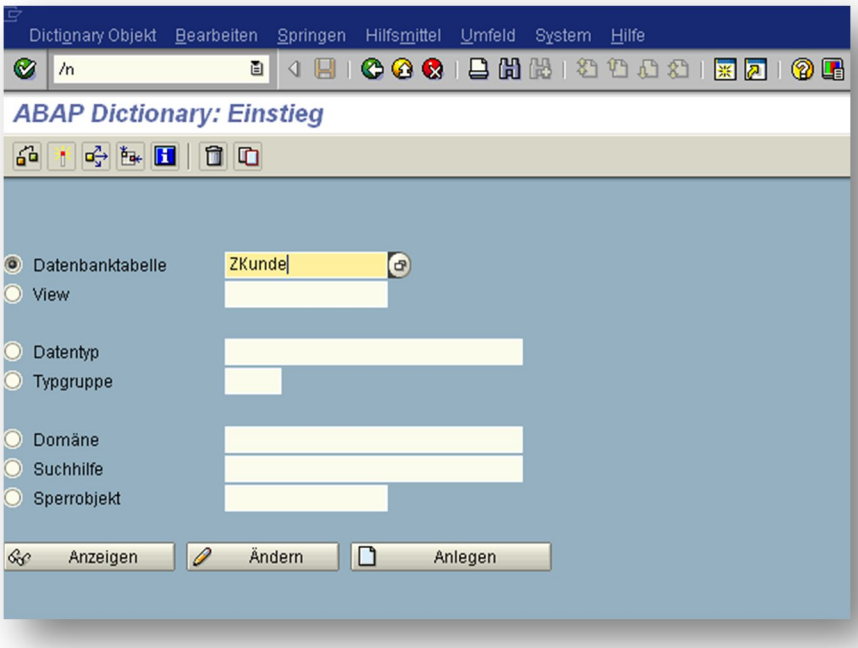
- **Domäne**: Domänen dienen der Definition der technischen Eigenschaften von Datenelementen. Innerhalb der Domäne können neben der Definition auch die Wertebereiche festgelegt werden.



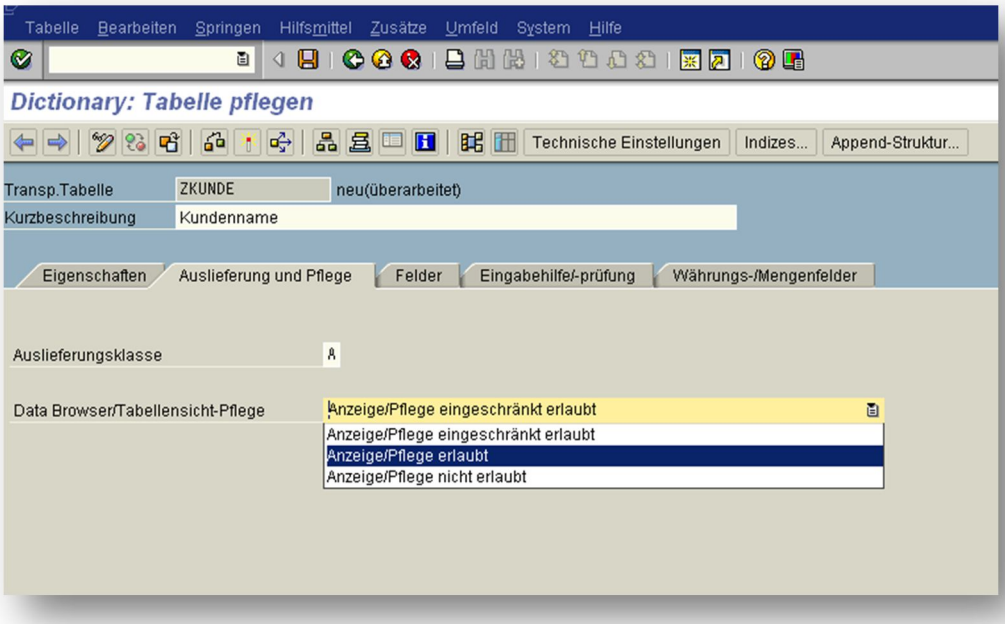
Demzufolge verfügt das DD von ABAP über eine Vererbungshierarchie, welche bei der Domäne beginnt, sich über das Datenelement bis zum Feld einer Tabelle fortsetzt. Bei der Neuanlage von Tabellen sind daher die notwendigen Datenelemente und Domänen zu definieren bzw. aus den bestehenden Datenelementen und Domänen zu entnehmen. Bei manchen Datenfeldern lohnt sich nicht die aufwendige Anlage von Datenelement und Domäne. Daher gibt es auch die Möglichkeit, Tabellenfelder mit direktem Bezug auf einen Datentyp, also ohne Datenelement und Domäne, anzulegen.

Die Anlage einer Tabelle wird wie folgt durchgeführt:

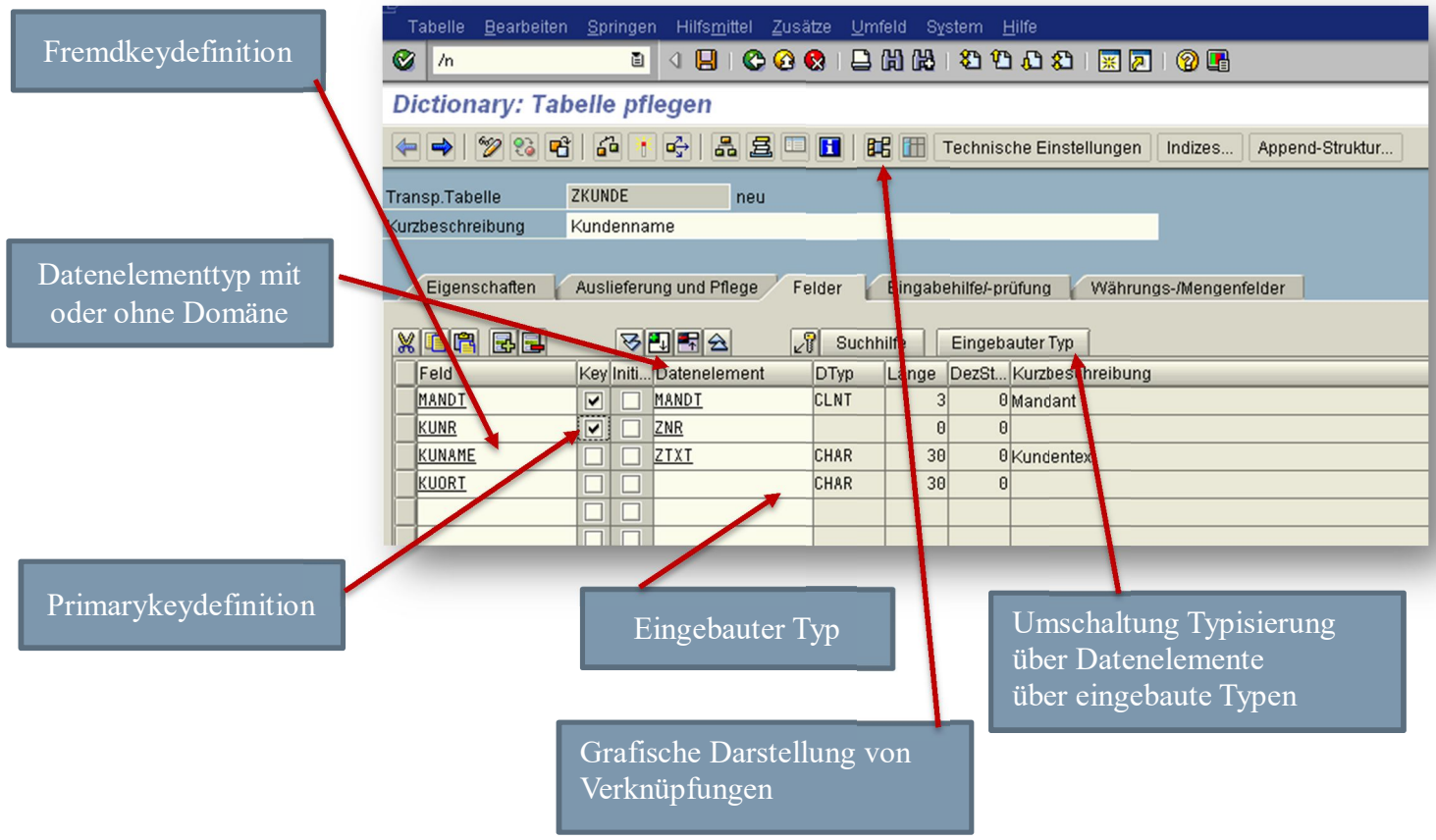
- **Angabe eines neuen Tabellennamens (ZKUNDE)**



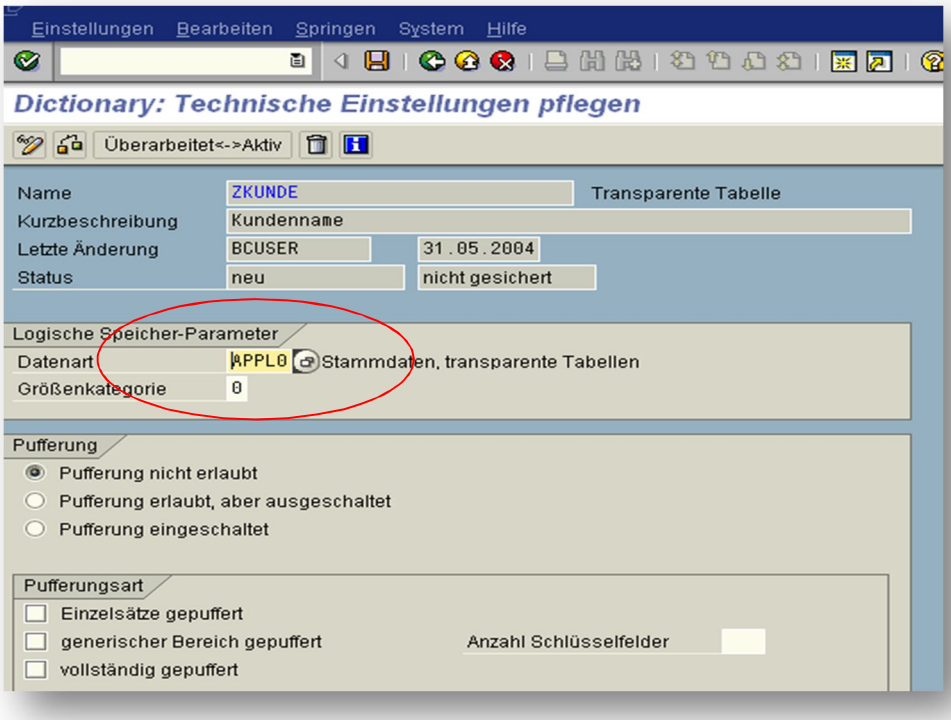
- **Angabe von Auslieferung und Pflegedaten**



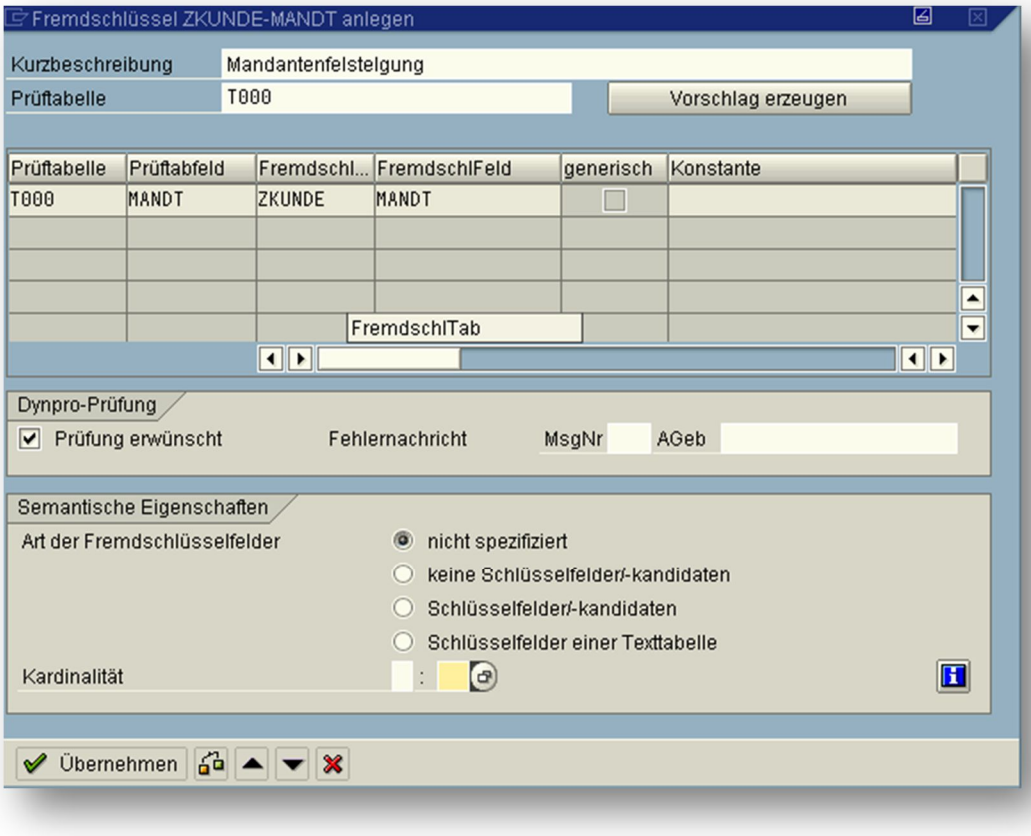
- **Definition der Felder über eingebaute Felder oder über ein eigens definiertes Datenelement**
- **Festlegung von einem Primärkey**



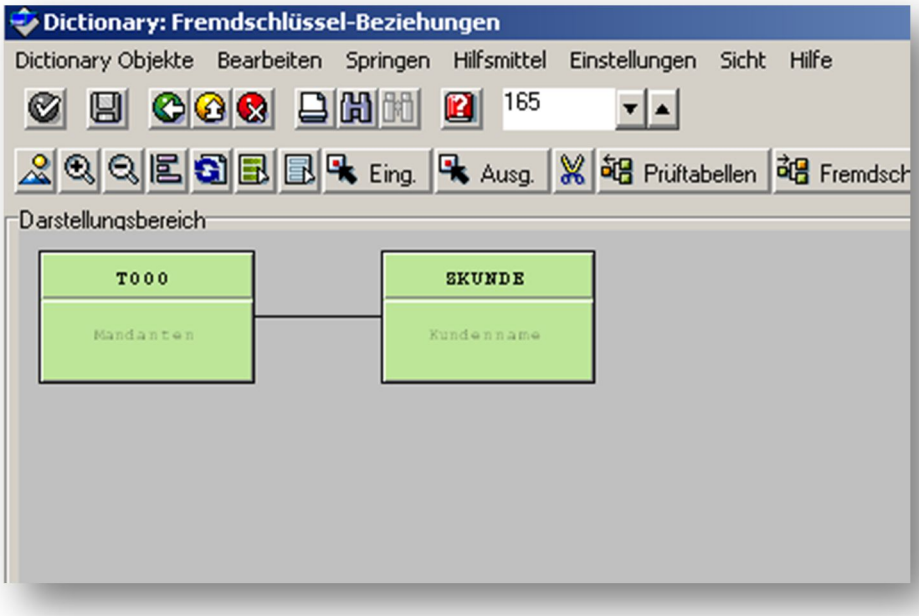
- **Datenelemente und Domänen müssen angelegt und aktiviert werden. ACHTUNG: Alle Domänen und Datenelemente, die nicht aktiviert sind, erscheinen auch nicht.**
- **Festlegung der technischen Einstellungen**



- **Festlegung der Fremdkeys: Über Fremdkeyfestlegung werden die Tabellen miteinander verknüpft . Im Folgenden Verknüpfungsfenster wird eine Fremdkeyfestlegung getroffen.**



- **Wenn Tabellen miteinander verknüpft sind, kann diese Verknüpfung grafisch dargestellt werden.**



Folgende Eingebaute Datentypen hat ABAP im DD.

Dictionary-Typ	Bedeutung	Zulässige Stellen m	ABAP-Typ
DEC	Rechen-/Betragsfeld	1-31, in Tab.: 1-17	p((m+1)/2)
INT1	1-Byte-Integer	3	b
INT2	2-Byte-Integer	5	s
INT4	4-Byte-Integer	10	i
CURR	Währungsfeld	1-17	p((m+1)/2)
CUKY	Währungsschlüssel	5	c(5)
QUAN	Menge	1-17	p((m+1)/2)
UNIT	Einheit	2-3	c(m)
PREC	Genauigkeit	2	x(2)
FLTP	Gleitpunktzahl	16	f(8)
NUMC	Numerischer Text	1-255	n(m)
CHAR	Character	1-255	c(m)
LCHR	Long Character	256-max	c(m)
STRING	Variable Zeichenfolge	1-max	string
RAWSTRING	Variable Bytefolge	1-max	xstring
DATS	Datum	8	d
ACCP	Buchungsperiode	6	n(6)
TIMS	Zeit	6	t
RAW	Bytefolge	1-255	x(m)
LRAW	Lange Bytefolge	256-max	x(m)
CLNT	Mandant	3	c(3)
LANG	Sprache	intern 1, extern 2	c(1)

Tabelle 4.2 Die eingebauten Typen des ABAP Dictionarys. Die Anzahl der Stellen bedeutet hier nicht die Feldlänge in Bytes, sondern die Anzahl der gültigen Positionen ohne Aufbereitungszeichen. Die ABAP-Typen b und s bei INT1 und INT2 haben nur interne Bedeutung. Bei LCHR und LRAW ist max der Wert eines vorangehenden INT2-Felds, bei LANG bedeutet »intern« im Dictionary und »extern« Darstellung auf Bildschirmbildern.

2.5.3 Individuelle Statuszeile

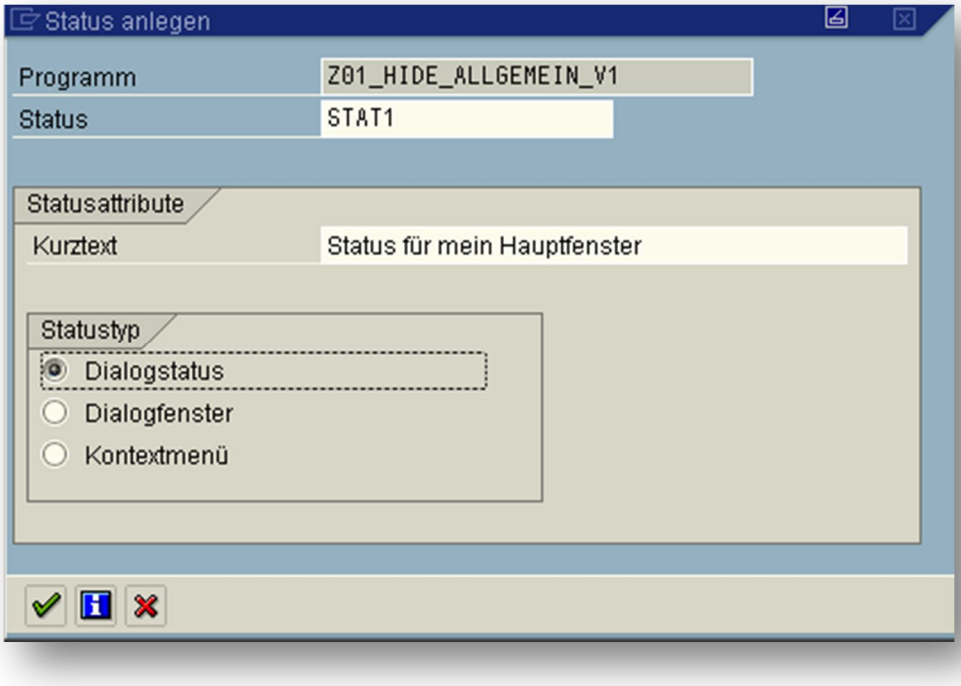
Mit ABAP können für die Programme individuelle Funktionscodes vergeben werden. Hierzu muss im aufrufen Programm der Befehl SET PF-STATUS ‚statusname' angegeben werden. Durch Vorwärtsnavigation können nun die individuellen Funktionscodes angelegt werden.

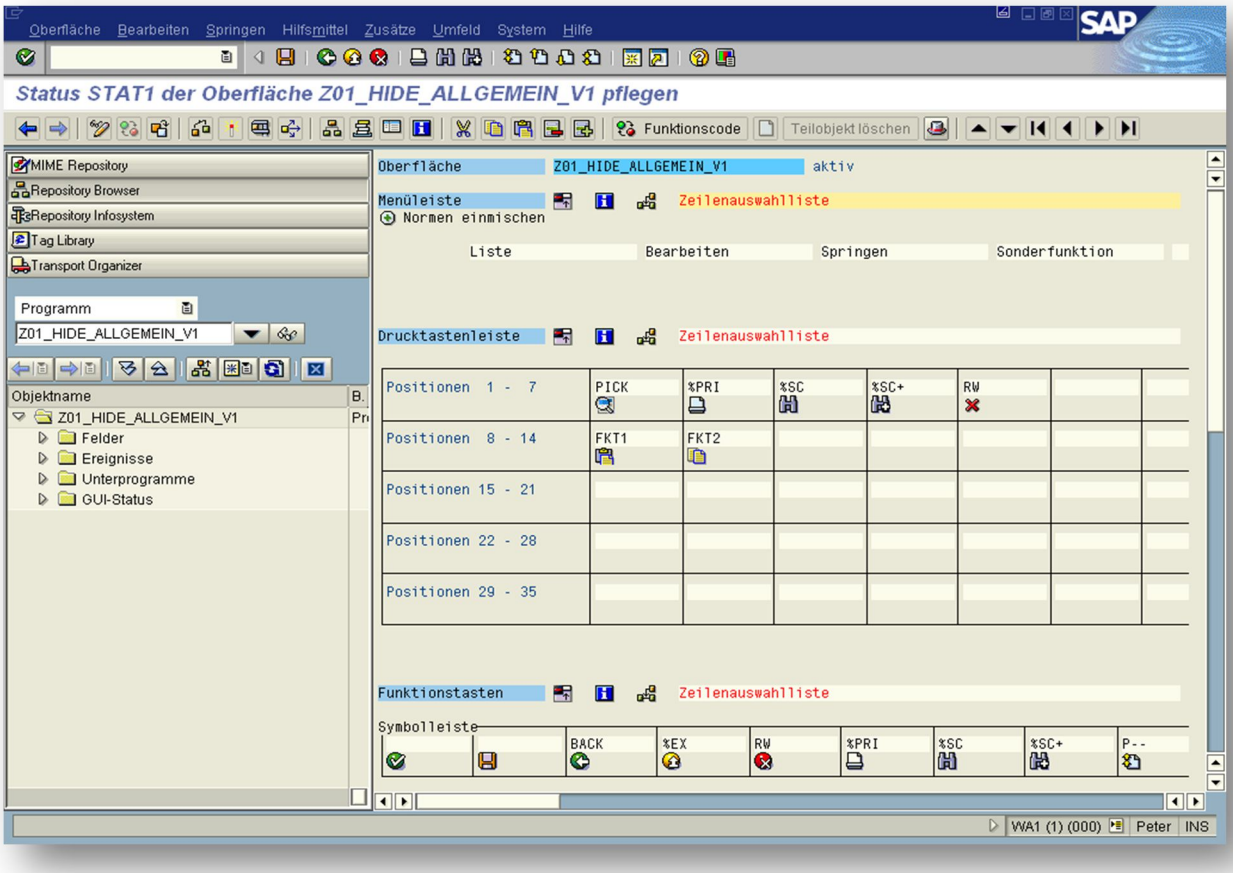
```
REPORT  z01_hide_allgemein_v1                                    .

DATA: zeile TYPE i VALUE 1.

START-OF-SELECTION.
```

```
  FORMAT COLOR 4.
  move 1   to zeile.
  write: / 'Hauptanzeige', sy-lsind.
  perform ausgabe.
*--- Anlage einer Statuszeile
  SET PF-STATUS 'STAT1'.
*--- Abfangen des indivuellen Funktionscodes
AT USER-COMMAND.
  CASE sy-ucomm.
    WHEN 'FKT1'.
      WRITE: / 'FUNKTION1', sy-lsind.
      move 100 to zeile.
      FORMAT COLOR 5.
      PERFORM ausgabe.
    WHEN 'FKT2'.
      WRITE: / 'FUNKTION2', sy-lsind.
      FORMAT COLOR 6.
      move 200 to zeile.
      PERFORM ausgabe.
  ENDCASE.
*---------------------------------------------------------------------*
```

Anlage einer Statuszeile

Mit Doppelclick auf die “STAT1” wird das Statusanlagefenster geöffnet, in dem als Pflichteingabe ein Kurztest eingegeben werden muss und anschließend wird auf das Hauptfenster der Statuszeilenpflege gesprungen.

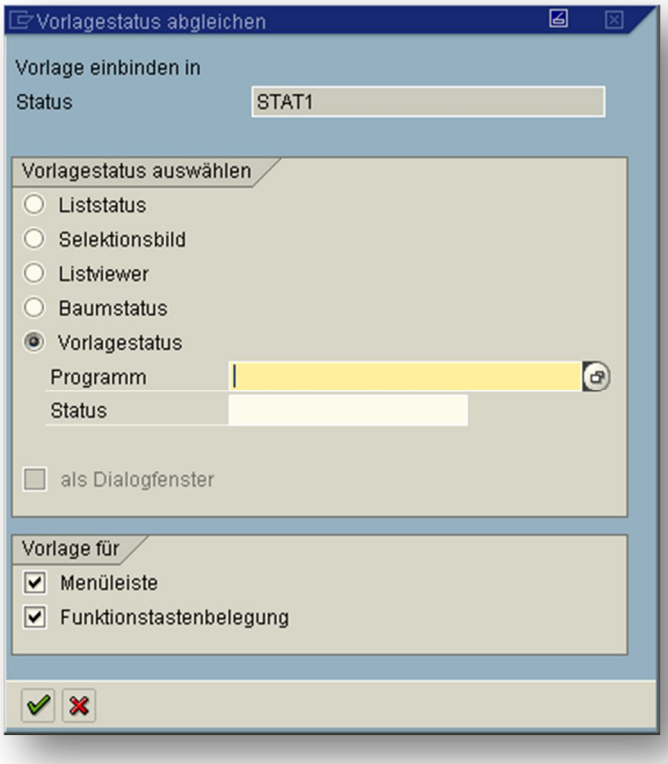




Standardvorbelegung

Um die Standardfunktionen z.B. eines Reports zu erhalten ist es notwendig, vor Eingabe der individuellen Funktionstasten/Menüleiste/Drucktastenleiste eine Vorlage zu generieren. Dies erfolgt über das Menü mit (ZUSÄTZE/VORLAGE abgleichen)

Im anschließenden Detailfenster können Standardvorlagen wie Liststatus, Selektionsbild usw. oder aus bereits vorhandenen Programmen Statuszeilen kopiert werden.

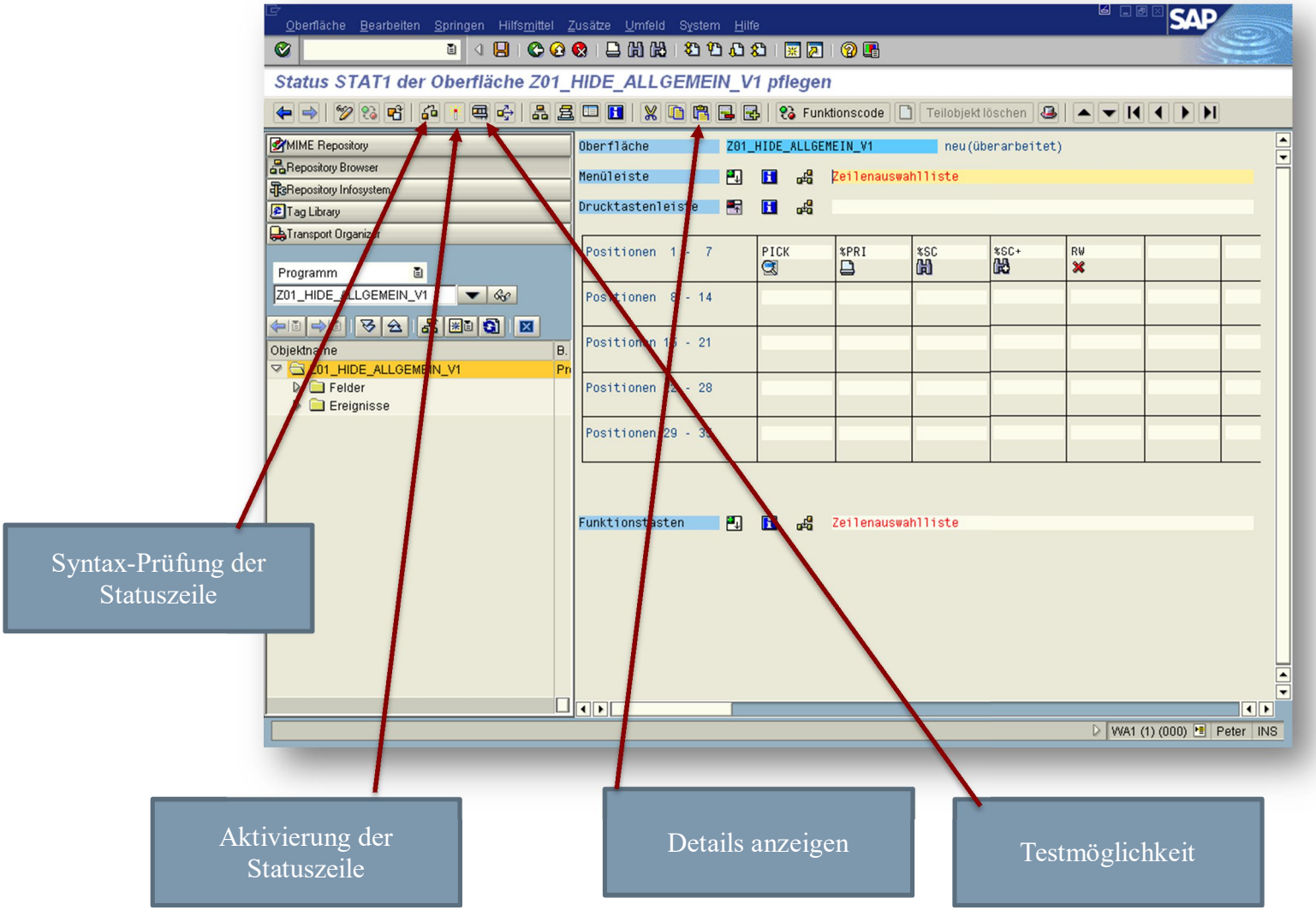


Grundlogik der Statusbearbeitung

Damit ist die notwendige Standardstatuszeile erzeugt und nun es kann eine individuelle Erweiterung vorgenommen werden. Die Grundlogik ist hierbei wie folgt:

- Jedem Menüleistenpunkt, jeder Drucktaste und jeder Funktionstaste ist ein Kurzcode zugeordnet.
- Mit der Wahl von einem Menüleistenpunkt, einer Drucktaste oder einer Funktionstaste wird der hinterlegte Code in die Systemvariable SY-UCOMM übertragen.
- Im Programm kann die Auswertung der Drucktaste innerhalb des Events AT USER-COMMAND ausgewertet und damit individuell reagiert werden.

Die Bedienoberfläche ist umfangreich und ermöglicht neben der Anlage auch den Test und was nicht zu vergessen ist, die Aktivierung.



Erzeugen einer individuellen Funktionstaste

Für die Definition von individuellen Bedienleisten stehen folgende Elemente zur Verfügung:

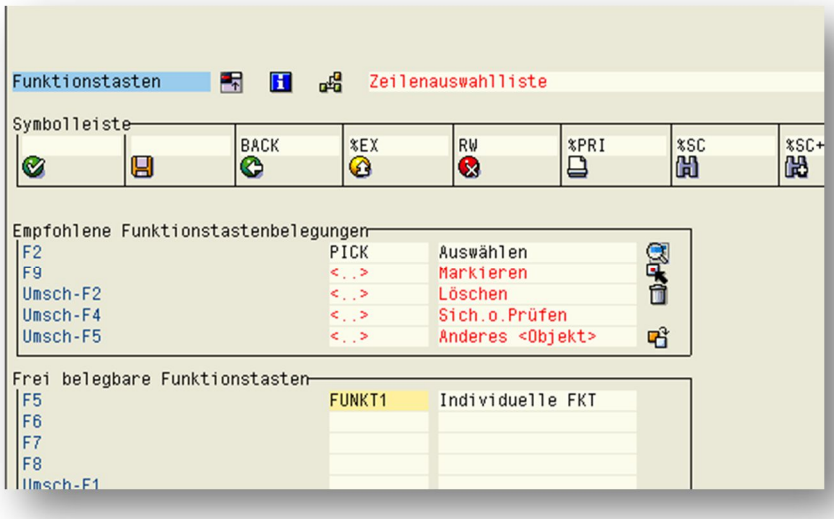
- Symbolleiste: Die Symbolleiste ist fest definiert
- Funktionstasten: Funktionstasten sind teilweise fest belegt bzw. frei für die Definition
- Drucktastenzeile: Anzeige von Drucktasten in Form von Bitmaps oder Texten
- Menüzeilen

Symbolleiste

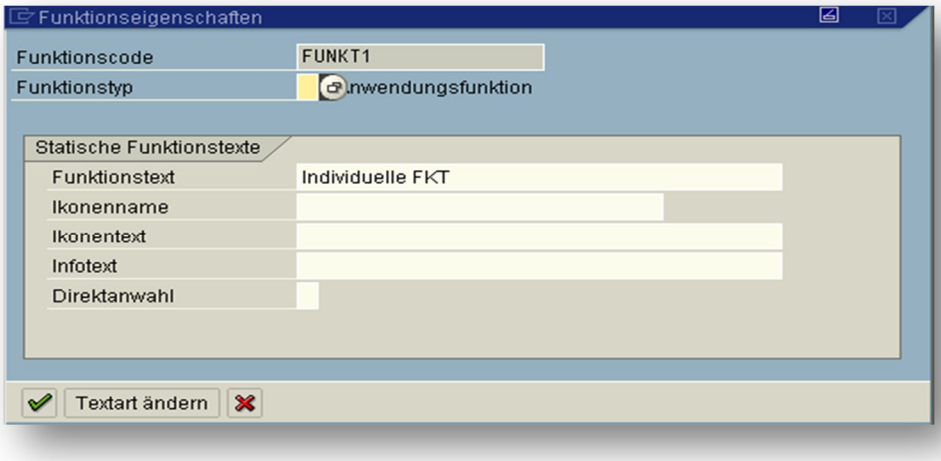
Die Symbolleiste ist „fest“ definiert und entspricht dem SAP-Standard. Sie kann nicht geändert werden. Allerdings können alle Symbole außer der Datenfreigabetaste aktiviert oder deaktiviert werden.

Funktionstasten

Es gibt festbelegte und freie Funktionstasten. Freie Funktionstasten werden im unteren Teil der Symbolleiste angezeigt. Hier können individuelle Funktionstasten definiert werden. Funktionstasten können im Übrigen alle Funktionscodes sein, die einen Funktionscode auslösen können z.B. CTRL A.
Die Funktionstaste z.B. F1(Fragezeichen), F3(Grüner Pfeil), F4(Lupe) und F10(Menüleiste aktivieren), F11(Gelber Ordner-Symbol), F12(Rotes Kreuz), F15 (Gelder Pfeil) werden durch das System erkannt und ausgewertet. Den Funktionstasten F1, F4 und F10 sind keine Systemcodes zugeordnet. Weitere Funktionstasten sind von SAP für bestimmte Anwendungen reserviert, solche sind z.B. Löschen oder Selektieren und können einer Liste von definierten Funktionscodes entnommen werden. Empfohlene Funktionstastenbelegungen werden im oberen Teil angezeigt.

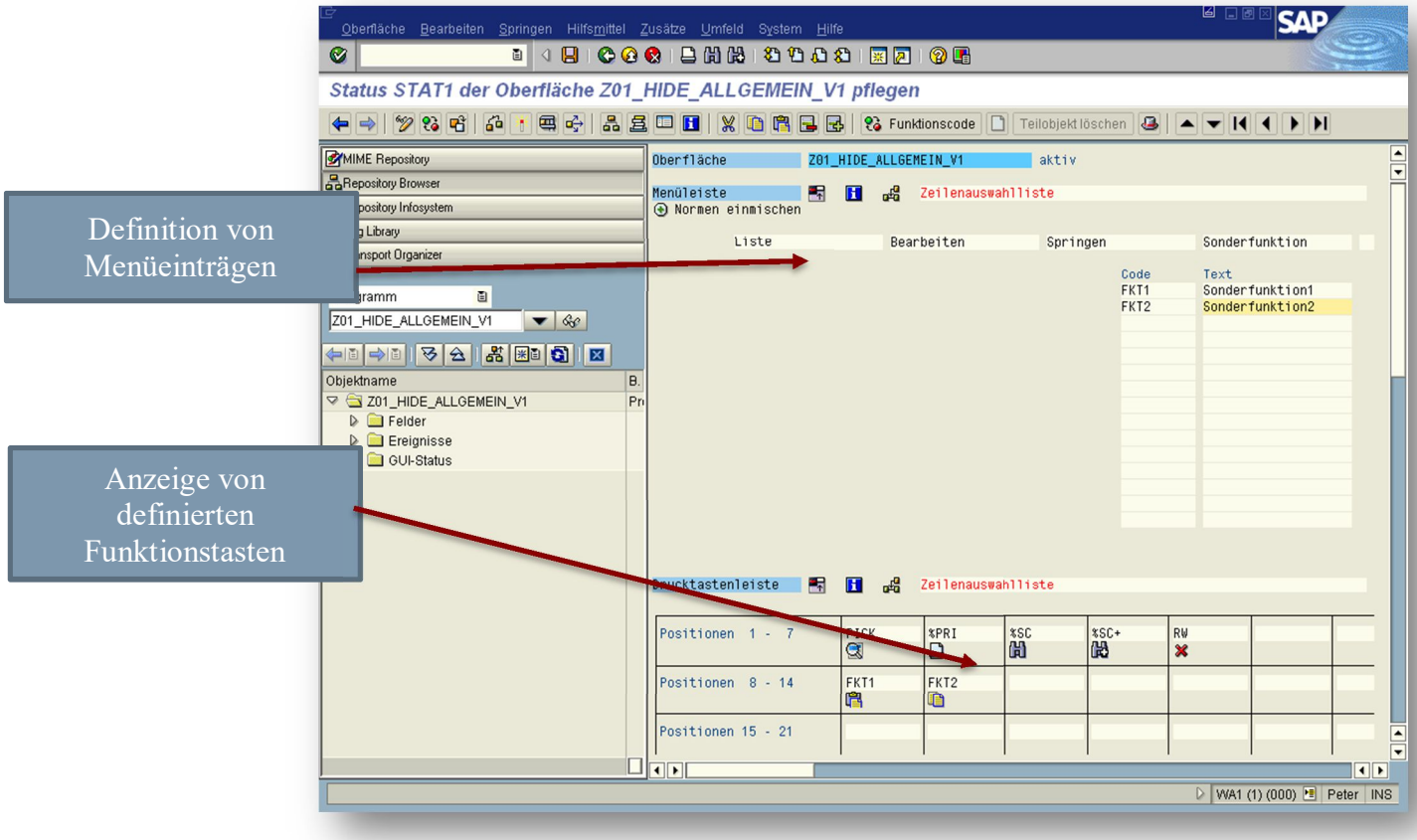


Die Anlage und Änderung der Eigenschaften einer individuellen, definierten Funktionstaste z.B. „FUNKT1“ erfolgt durch Vorwärtsnavigation und im Fenster „ Funktionseigenschaften“ werden sie angepasst. Angelegte Funktionstasten können in der Drucktastenleiste zur Anzeige gebracht werden.



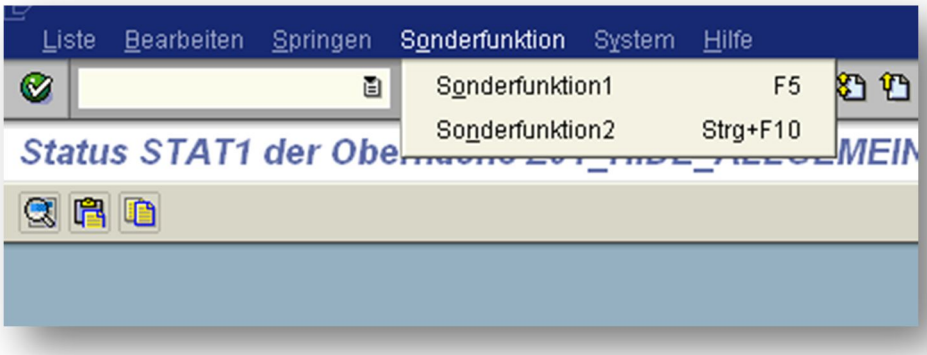
Drucktastenleiste und Menüzeile

Mit der Drucktastenleiste werden die Funktionstasten zur Anzeige auf dem Bildschirm gebracht. Anwendungsmenüs werden in der Menüleiste angezeigt.



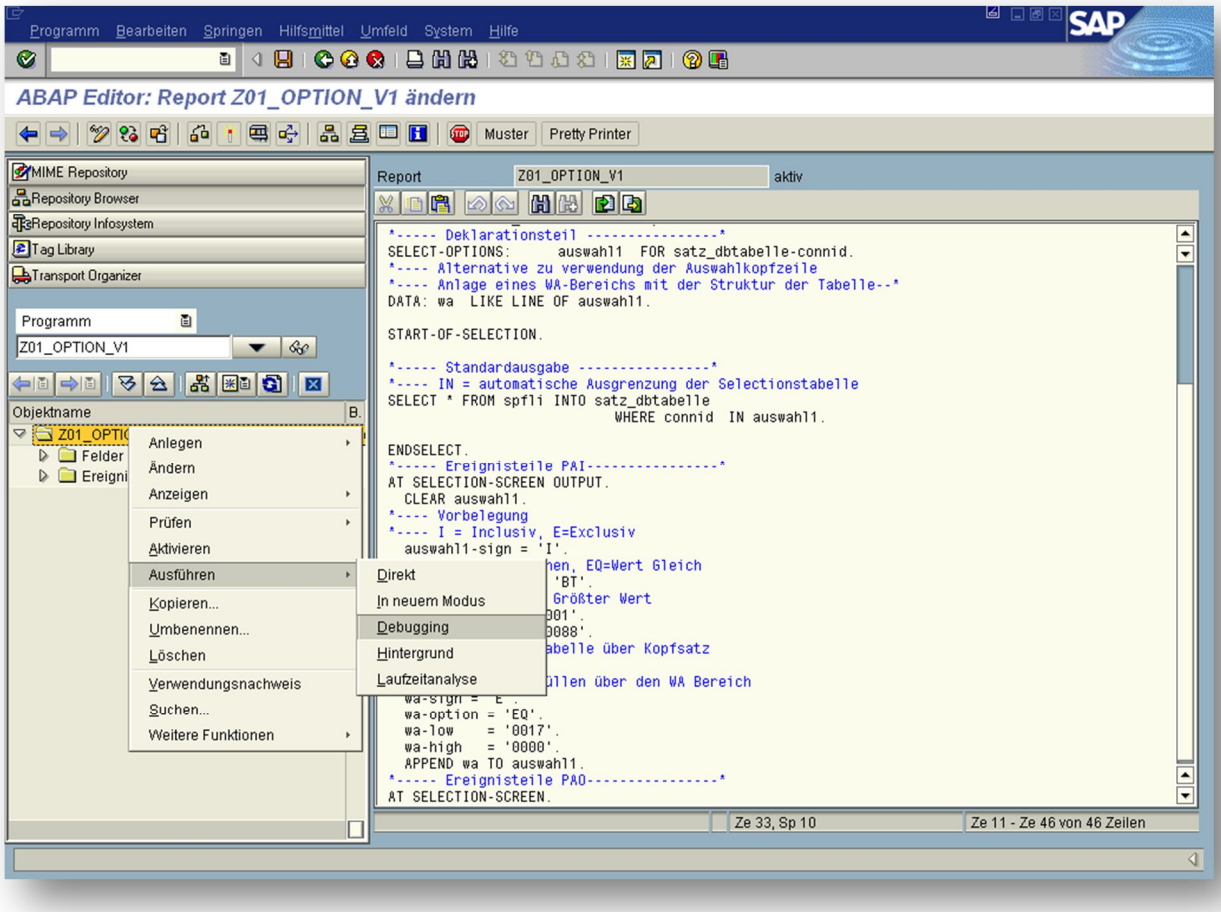
Syntax/Aktivierung/Test

Im Hauptmenü müssen die definierten Statuszeilen auf Syntax geprüft und aktiviert werden. Nur aktivierte Statuszeilen werden erkannt. Mit dem Testsymbol kann das Erscheinungsbild der Statuszeile unabhängig vom Programm gesichtet werden.

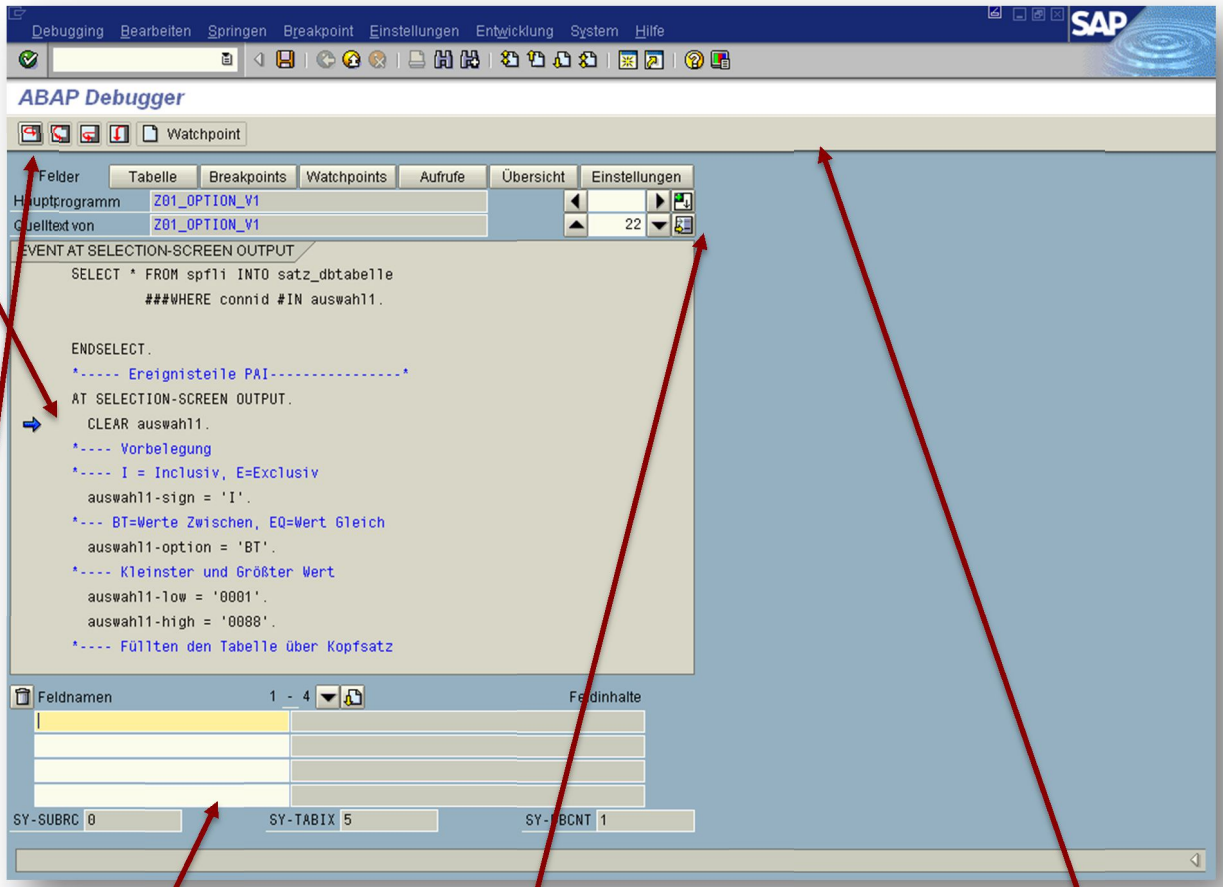


2.5.4 Debugger

Der Start des Debuggers erfolgt durch Aufruf des syntaxabhängigen Menüs und Wahl der Punkte ‚Ausführen, Debugging'.



Die Form des Debugging-Vorgangs und dessen Umfang, d.h. ob z.B. auch die vom Runtime automatisch dazugenerierten System-Programmteile durchlaufen werden, wird in dem Menüpunkt ‚Einstellungen' des Debuggers festgelegt. Mit den Push-Buttens „Felder, Tabelle, Breakpoints usw." können die Sichten auf unterschiedliche Debuggingobjekte eingerichtet werden. Im nachfolgenden Bild sind die wichtigen Funktionen angezeigt.



Position im Programm (Doppelclick setzt Breakpoint)

Schritte durch das Programm

Anzeige von Feldinhalten, Tabellen, Adressen usw.

Zeilensprünge durch das Programm

Sichten auf Debuggingobjekte

3. Ausgewählte Programmbeispiele

3.1 DATA und TYPES

Datentypen mit Längenangaben:

Im folgenden sind Beispiele für Datentypen mit Längenangaben zu finden.

Datentyp p (packed numbers)

```
DATA: number TYPE p DECIMALS 2.
START-OF-SELECTION.
Number =  3 / 4.
Write: number.
```

Datentyp c (character fields)

```
DATA text(72) TYPE c.
START-OF-SELECTION.
Text = 'Dies ist der Inhalt des Textes'.
IF text <> spaces.
   WRITE: / text.
ENDIF.
```

Datentyp n (numerisches Textfeld)

```
DATA plz(5) TYPE n.
START-OF-SELECION.
MOVE '35578' to plz.
WRITE plz.
```

HEX x

```
DATA hex(3) TYPE x.
Hex = 'F72AB3'.
*Inhalt = 247, 42,179
```

Datentyp ohne Längenangabe:

Daneben gibt es solche Datentypen, die keine Längenangabe erfordern.

Numerischer Typ I

```
TYPES: tindex TYPE i.
DATA: index TYPE tindex.
START-OF-SELECTION.
index = 5.
         DO index TIMES.
```

```
    WRITE: ,Zahl', index.
ENDDO.
```

Numerischer TYP F (floating point numbers Wertebereich 1x10-307 und 1x10+308)

```
DATA ergebnis TYPE f.
START-OF-SELECTION.
Ergebnis = sqrt( 2 ).
WRITE ergebnis.
```

Datentype d (Datum)

```
DATA datum TYPE d.
START-OF-SELECTION.
Datum = sy-datum.
Datum = datum + 2.
WRITE datum.
```

Zeittyp t (HHMMSS)

```
DATA zeit TYPE t.
START-OF-SELECTION.
zeit = sy-uzeit.
Zeit = zeit + 3600.
WRITE zeit.
zeit+2(4) = ‚0000’.
WRITE zeit.
```

Datentypen variabler Länge:

Einige Datentypen verfügen über eine flexible Länge

Datentype string (character string)

```
DATA text TYPE string.
START-OF-SELECTION.
Text= ‘ABAP macht Spass’.
WRITE text.
```

Datentyp x_string (byte strings)

```
DATA x_string.
X_string = ‘FF’.
```

KONVERTIERUNGSREGELN:

Zwischen den Datentypen gibt es Konvertierungsregeln (Ausnahme: Datum und Zeit). ABAP wendet diese Konvertierungsregeln wenn möglich automatisch an.

BESTIMMUNG von Datentypen:

Welcher Datentyp ein Feld hat kann mit dem Befehl DESCRIBE ermittelt werden. Der Datentyp ANY ist z.B. vollständig generisch und kann jeden Datentyp automatisch annehmen. Anhand des nachfolgenden Beispiel wird gezeigt, wie mit generischen

Datentypen gearbeitet wird. Mit dem Befehl DESCRIBE können die Feldeigenschaften ermittelt werden.

```
REPORT  z01_data_type_v1                                                .
DATA: f1(20) TYPE c,
      f2 TYPE p DECIMALS 2.


START-OF-SELECTION.
  PERFORM: test USING f1,
           test USING f2.

*---------------------------------------------------------------------*
*  FORM test
*---------------------------------------------------------------------*
*
*---------------------------------------------------------------------*
*  -->  U_INPUT
*---------------------------------------------------------------------*
FORM test USING u_input TYPE any.
  DATA: t(1) TYPE c,
  l TYPE i,
  d TYPE i.

  DESCRIBE FIELD u_input TYPE t LENGTH l IN BYTE MODE DECIMALS d.
  write: / 'Typ:', t, 'Länge', l, 'Dezimalstellen', d.
ENDFORM  .
                "test
```

Konstanten

```
CONSTANTS pi TYPE f VALUE '3.14159265359'.
DATA: radium TYPE p DECIMALS 2,
          Area    TYPE p DECIMALS 2.
START-OF-SELECTION.
Area = pi * radiums ** 2.
```

TYPISIERUNG aus dem DD:

Datenfelder im Programm können über das DD typisiert werden.

```
REPORT  z01_data_type_v2                                                .
*----- Struktur mit zwei Feldern aus dem DD ------------*
DATA: BEGIN OF wa,
      carrid TYPE spfli-carrid VALUE 'LH',
      connid TYPE spfli-connid VALUE '400',
      END OF wa.
*--- Workingstruktur von spfli (siehe Vorwärtnavigation DD)
data: wa_spfli type spfli.

START-OF-SELECTION.
*---- Übertragen der Feldinhalt carrid, connid
  MOVE-CORRESPONDING wa TO wa_spfli.
```

Grundform von TYPES und DATA: Im Nachfolgenden werden nochmals die Grundformen von TYPES und DATA gezeigt.

- **Datentypen**: TYPES type(len) TYPE type (DECIMALS dec).
- **Datenobjekte**: DATA dobj(len) TYPE type (DECIMALS dec) (VALUE val).
- **Datenobjekt als Typ:** TYPES(DATA) .... LIKE dobj.

Elementare Datentypen und Datenobjekte

```
REPORT  z01_data_type_v3                                                .

TYPES: terg(10) TYPE p DECIMALS 2.
DATA: nr1 TYPE i VALUE 3,
      nr2 LIKE nr1 VALUE 4,
      erg TYPE terg.

START-OF-SELECTION.
  erg = nr1 / nr2.
  WRITE erg.
```

Strukturierende Datentypen und Datenobjekte

```
REPORT  z01_data_type_v4                                                .

TYPES: BEGIN OF tstrasse,
       name(40) TYPE c,
       nr(4) TYPE c,
       END OF tstrasse.

DATA: BEGIN OF adresse,
      name(30) TYPE c,
      strasse TYPE tstrasse,
      BEGIN OF city,
            plz(5) TYPE n,
            name(40) TYPE c,
      END OF city,
      land(3) TYPE c VALUE 'DE',
END OF adresse.

start-of-selection.
*------ Beachten Sie die Namenszusammensetzung ----*
adresse-name = 'Peter Maier'.
adresse-strasse-name = 'Obergasse'.
adresse-strasse-nr   = '12'.
adresse-city-name = 'Wetzlar'.
```

Einfügen von Strukturkomponenten

```
REPORT  z01_data_type_v5                                                .
```

```
DATA: BEGIN OF strasse,
         name(40) TYPE c,
         nr(4) TYPE c,
      END OF strasse.

DATA: BEGIN OF ort,
        plz(5) TYPE n,
        name(40) TYPE c,
      END OF ort.

DATA: BEGIN OF adresse,
   name(30) TYPE c.
        INCLUDE STRUCTURE strasse AS str
              RENAMING WITH SUFFIX _str.
        INCLUDE STRUCTURE ort AS o
               RENAMING WITH SUFFIX _o.
DATA END OF adresse.

START-OF-SELECTION.
  adresse-name = 'Müller'.
  adresse-name_str = 'Obergasse'.
  adresse-plz_o = '35578'.
```

3.2 Interne Tabellen

Definition von Tabellen:

	Standard: STANDARD TABLE	**Sortierte Tabelle**	**Hash Tabelle**
Definition	itab TYPE STANDARD TABLE OF struktur WITH NON-UNIQUE KEY feld.	itab TYPE SORTED TABLE OF struktur OF struktur WITH UNIQUE KEY feld.	Itab TYPE HASHED TABLE of struktur WITH UNIQUE KEY feld.
INDEX	JA	JA	Nein
Schlüssel	(JA möglich)	JA	JA
Eindeutigkeit des Schlüssels	NON-UNIQUE	UNIQUE und NON-UNIQUE	UNIQUE
Zugriff auf Zeilen (bevorzugt)	Index (Schlüssel nur mir READ)	Index oder Schlüssel	Nur Schlüssel
Befehl APPEND (anhängen)	JA	NEIN	NEIN
Befehl INSERT	Nein (Wirkung wie APPEND)	JA (Schlüssel)	JA (Schlüssel)
Befehl READ	Index oder Schlüssel	Index oder Schlüssel	Schlüssel
Befehl DELETE	Index	Index oder Schlüssel	Schlüssel
Befehl MODIFY	Index	Index oder Schlüssel	Schlüssel
Befehl SORT	JA	Nein	Nein
Befehl LOOP AT	JA	JA	JA

Befehl CLEAR	JA	JA	JA

ACHTUNG: Die meisten Zugriffe sind sowohl über Index als auch über Schlüssel möglich. Die Systemvariable **SY-TABIX** enthält den letzten Index.

```
*&---------------------------------------------------------------------*
*& Report  Z_INTERNE_TABELLEN_B02_01                                   *
*&                                                                     *
*&---------------------------------------------------------------------*

REPORT  z_interne_tabellen_b02_01                                     .


TYPES: BEGIN OF tadresse,
            vname(40) TYPE c,
            nname(40) TYPE c,
       END OF tadresse.

DATA: wa TYPE tadresse,

*---- Standardtabelle mit bzw. ohne Schlüssel
      itab_std   TYPE STANDARD TABLE OF tadresse WITH KEY nname,
      itab_index TYPE SORTED   TABLE OF tadresse WITH NON-UNIQUE KEY
                                                              nname,
      itab_hash  TYPE HASHED   TABLE OF tadresse WITH UNIQUE KEY nname.

DATA: wadb TYPE STANDARD TABLE OF spfli,
      wa1  TYPE spfli.

INITIALIZATION.
  CLEAR itab_std.


START-OF-SELECTION.
*--- Verarbeitung von STd-Tabellen
  PERFORM fuelle_std.
  PERFORM verarbeite_std.
  ULINE.
  ULINE.
  PERFORM fuelle_index.
  PERFORM verarbeite_index.
  ULINE.
  ULINE.
  PERFORM db_tabelle.


*&---------------------------------------------------------------------*
*&      Form  verarbeite_std
*&---------------------------------------------------------------------*
*       text
*----------------------------------------------------------------------*
FORM verarbeite_std.
*--- Sortierung einer Tabelle
  SORT itab_std BY vname.
*--- Ausgabe einer Tabelle
  WRITE: / 'Ausgabe über eine Loop-Schleife',  /.
  LOOP AT itab_std INTO wa.
```

```
    WRITE: wa-vname, wa-nname, /.
  ENDLOOP.
  ULINE.
*---- Alle Zugriff entweder über Index oder Schlüssel -------------*
*---- Direktzugriff über einen Schlüssel --------------------------*
  MOVE 'rühl' TO wa-nname.
  READ TABLE itab_std WITH TABLE KEY nname = wa-nname INTO wa.
  PERFORM fehlermeldung.
  WRITE: / 'Direktzugriff über Key', 60 wa-vname, wa-nname.
*---- Direktzugriff über eine Index
  READ TABLE itab_std INDEX 1 INTO wa.
  WRITE: / 'Direktzugriff über Index', 60  wa-vname, wa-nname.
*--- Index der Tabelle ---------------------------------------*
  WRITE: / 'Akuteller Index letzer Zugriff', sy-tabix.

*----- Löschen des Tabellenzeile mit Key
  ULINE.
  MOVE 'hohmann' TO wa-nname.
*---- Löschen mit Key
  DELETE TABLE itab_std WITH TABLE KEY nname = wa-nname.
*----- Delete mit index.
  DELETE itab_std INDEX 2.
  LOOP AT itab_std INTO wa.
    WRITE:/  wa-vname, wa-nname, /.
  ENDLOOP.

  ULINE.
  ULINE.


*---- Lesen eines Eintrags und Verändern
  DATA: int TYPE i.
  wa-nname = 'hohmann'.
  READ TABLE itab_std WITH TABLE KEY nname = wa-nname INTO wa.
  int = sy-tabix.
  wa-nname = 'hohmann'.
  wa-vname = 'peter franz-josef'.
*---- Indexmodifizierung
  MODIFY itab_std from wa index int.
  clear wa.
  wa-nname = 'hohmann'.
  READ TABLE itab_std WITH TABLE KEY nname = wa-nname INTO wa.
      WRITE:/  wa-vname, wa-nname, /.


ENDFORM.                    "verarbeite_std


*&---------------------------------------------------------------------*
*&      Form  fuelle_std
*&---------------------------------------------------------------------*
*       text
*----------------------------------------------------------------------*
FORM fuelle_std.
*---- Befüllen der Tabelle
  wa-vname = 'peter'.
  wa-nname = 'hohmann'.
  APPEND wa  TO itab_std.
  PERFORM fehlermeldung.
```

```
  wa-vname = 'peter'.
  wa-nname = 'hohmann'.
  APPEND wa  TO itab_std.
  PERFORM fehlermeldung.


  wa-vname = 'erich'.
  wa-nname = 'rühl'.
  APPEND wa  TO itab_std.
  PERFORM fehlermeldung.

  wa-vname = 'klaus'.
  wa-nname = 'armbrüster'.
  APPEND wa  TO itab_std.
  PERFORM fehlermeldung.

ENDFORM.                    "fuelle_std

*&---------------------------------------------------------------------*
*&      Form  fehlermeldung
*&---------------------------------------------------------------------*
*       text
*----------------------------------------------------------------------*
FORM fehlermeldung.
  IF sy-subrc <> 0.
    WRITE: 'Fehler in der Tabelle'.
  ENDIF.
ENDFORM.                    "fehlermeldung


*&---------------------------------------------------------------------*
*&      Form  db_tabelle
*&---------------------------------------------------------------------*
*       text
*----------------------------------------------------------------------*
FORM db_tabelle.

  SELECT * FROM spfli INTO TABLE wadb.
  PERFORM fehlermeldung.

  SORT wadb BY connid.

  LOOP AT wadb INTO wa1.
    WRITE: wa1-mandt,
           wa1-carrid,
           wa1-connid,
           wa1-countryfr,
           wa1-cityfrom,
           wa1-airpfrom,
           wa1-countryto,
           wa1-cityto,
           wa1-airpto,
           wa1-fltime,
           wa1-deptime,
           wa1-arrtime,
           wa1-distance,
           wa1-distid,
           wa1-fltype,
           wa1-period,
```

```
          / .
  ENDLOOP.


ENDFORM.                    "db_tabelle
*&---------------------------------------------------------------------*
*&      Form  verarbeite_index
*&---------------------------------------------------------------------*
*       text
*----------------------------------------------------------------------*
FORM verarbeite_index.
  ULINE.
  WRITE: / 'Ausgabe der Tabelle mit Sorted table', /.
*--- Ausgabe einer Tabelle
  LOOP AT itab_index INTO wa.
    WRITE: wa-vname, wa-nname, /.
  ENDLOOP.
  ULINE.
  MOVE 'rühl' TO wa-nname.
*  READ TABLE itab_index key nname INTO wa.
  PERFORM fehlermeldung.
  WRITE: wa-vname, wa-nname.
ENDFORM.                    "verarbeite_std



*&---------------------------------------------------------------------*
*&      Form  fuelle_index
*&---------------------------------------------------------------------*
*       text
*----------------------------------------------------------------------*
FORM fuelle_index.
*---- Befüllen der Tabelle
*---- Schlüsselzugriff INSERT wa INTO table.
*---- Indexzugriff     INSERt wa INTO itab INDEX idx.
  wa-vname = 'peter'.
  wa-nname = 'hohmann'.
  INSERT wa  INTO TABLE itab_index.
  PERFORM fehlermeldung.

  wa-vname = 'peter'.
  wa-nname = 'hohmann'.
  INSERT wa  INTO TABLE itab_index.
  PERFORM fehlermeldung.


  wa-vname = 'erich'.
  wa-nname = 'rühl'.
  INSERT wa  INTO TABLE itab_index.
  PERFORM fehlermeldung.

  wa-vname = 'klaus'.
  wa-nname = 'armbrüster'.
  INSERT wa  INTO TABLE itab_index.
  PERFORM fehlermeldung.

ENDFORM.                    "fuelle_index
```

3.3 Event-Struktur Report

```
*&---------------------------------------------------------------------*
*& Report  Z01_ITAB                                                    *
*&                                                                     *
*&---------------------------------------------------------------------*

REPORT  z01_itab   LINE-SIZE 100    .
*-- Deklarativer Bereich ----*

*---- Flache Struktur -------*
DATA: wa TYPE sflight.
*---- Tabelle ---------------*
DATA: watab TYPE STANDARD TABLE OF sflight.

*---- Ausgrenzung eines Feldes ----*
PARAMETERS: eingabe   TYPE sflight-carrid.

*---- Ausgrenzung über eine geschenkte Tabelle ----*
SELECT-OPTIONS: s_connid FOR  wa-connid.


SELECTION-SCREEN SKIP.

SELECTION-SCREEN BEGIN OF BLOCK sortierung WITH FRAME TITLE sort.
PARAMETERS: sort1 AS CHECKBOX,
            sort2 AS CHECKBOX DEFAULT 'X',
            sort3 AS CHECKBOX.
SELECTION-SCREEN END OF BLOCK sortierung.

*---- Ausführung beim Start des Programms / Construktor ----*
LOAD-OF-PROGRAM.
  MOVE 'LH'    TO eingabe.

*---- PBO -------------------*
AT SELECTION-SCREEN OUTPUT.
  MOVE 'X'     TO sort1.
  MOVE ' '     TO sort2.
  move ' '     to sort3.

*---- PAI -------------------*
AT SELECTION-SCREEN.
*----- IF-Abfrage ------------*
  IF  sort1 <> 'X'
  AND sort2 <> 'X'
  AND sort3 NE 'X'.
    MESSAGE e001(zhoh).
  ENDIF.
*--- PAI auf ein Feld -------*
AT SELECTION-SCREEN ON eingabe.
  IF eingabe = '   '.
    MESSAGE e003(zhoh).
*  else.
*     message s002(zhoh).
  ENDIF.

*--- Hauptprogramm ----------*
```

```
START-OF-SELECTION.

  SELECT * FROM sflight INTO TABLE watab WHERE connid IN s_connid
                                         and   carrid = eingabe .


  CASE sy-subrc .
    WHEN  0.
      PERFORM sortierung.

      LOOP AT watab INTO wa.
        WRITE: wa-carrid,
               wa-connid,
               wa-fldate,
               /.
      ENDLOOP.
    WHEN OTHERS.
      WRITE 'Keine gültigen Einträge gefunden' .
      ULINE.
  ENDCASE.


END-OF-SELECTION.

*---------------------------------------------------------------------*
*  FORM sortierung
*---------------------------------------------------------------------*
*
*---------------------------------------------------------------------*
FORM sortierung.
  IF sort1 = 'X'.
    SORT watab BY carrid.
  ENDIF.
  IF sort2 = 'X'.
    SORT watab BY connid.
  ENDIF.
  IF sort3 = 'X'.
    SORT watab BY fldate.
  ENDIF.
ENDFORM.                    "sortierung

*---Listenanfang ---*
TOP-OF-PAGE.
  WRITE: 'Dies ist der Listenkopf'.
*---- Listenende ---*

END-OF-PAGE.
  uline.
  WRITE: ' Dies ist das Listenende'.
*----- Doppelclick auf eine Listenzeile---*
AT LINE-SELECTION.

*---- Reaktion auf Benutzereingabe sy-ucomm---*
AT USER-COMMAND.
```

Im nachfolgenden Beispiel werden die wichtigsten Events eines Report-Programms gezeigt.

3.4 Einfache Reports

(siehe Übung)

3.5 Interaktive Reports

In der Reportliste sind für das interaktive Reporting verschiedene Ereignisse von Bedeutung, um Anwenderaktivitäten zu erkennen und zu verarbeiten. So löst der Doppelklick mit der Maus oder die F2-Funktionstaste das Ereignis AT LINE-SELECTION. Mittels der Anweisung HIDE können Informationen über die selektierte Zeile ermittelt werden. HIDE muss in die jeweiligeListe programmiert werden. Innerhalb von AT LINE-SELECTION kann z.B. mit SET PF-STATUS eine neue Statuszeile für die Unterliste (nicht für die Grundliste) erzeugt werden. Seitenüberschriften der Grundliste ergeben sich mit TOP OF PAGE. Für die Verzweigungsliste ist der Befehl TOP OF PAGE DURING LINE-SELECTION. Alle Funktionscodes, die nicht automatisch vom System verarbeitet werden, können in AT USER-COMMAND ausgewertet werden. Alle gewählten Funktionen stehen im Feld SY-UCOMM.

Wichtige Systemfelder beim interaktiven Report sind:

Systemfeld	Bedeutung
SY-LSIND	Nummer der Liste (Grundliste = 0)
SY-CUROW	Zeilenposition des Cursors in der aktuellen Liste
SY-CUCOL	Spaltenposition des Cursors in der aktuellen Liste
SY-LISEL	Inhalt der selektierten Zeile
SY-LILLI	Absolute Zeilennummer der selektierten Zeile
SY-LISTI	Nummer der Liste der selektierten Zeilen
SY-PFKEY	Name des aktuellen Status der Oberfläche
SY-CPAGE	Erste angezeigte Seite
SY-STARO	Erste angezeigte Zeile
SY-STACO	Erste angezeigte Spalte

Verzweigungsliste mit HIDE

```
REPORT  z_hite_beispiel01 LINE-SIZE 120         .

DATA:
*---- Fluggesellschaften
      wa_spfli TYPE spfli,
*---- Flüge der Fluggesellschaften
      wa_sflight TYPE sflight.
PARAMETERS: eing(10).



START-OF-SELECTION.
```

```
  SELECT * FROM spfli INTO wa_spfli.
    WRITE: /,
           wa_spfli-carrid,
           wa_spfli-connid,
           wa_spfli-countryfr,
           wa_spfli-cityfrom,
           wa_spfli-airpfrom,
           wa_spfli-countryto,
           wa_spfli-cityto.
    HIDE:  wa_spfli-carrid,  wa_spfli-connid.
  ENDSELECT.
*---- Überschrift.
TOP-OF-PAGE.
  WRITE: / 'Fluggesellschaften'.
  ULINE.

TOP-OF-PAGE DURING LINE-SELECTION.
  CASE sy-lsind.
    WHEN 1.
      WRITE: / 'Ausgabe von Details'.
    WHEN OTHERS.
      WRITE: / 'Keine korrekte Liste'.
  ENDCASE.
*----- Ausdruck der Selektionswerte -----------------*
AT LINE-SELECTION.
  CASE sy-lsind.
    WHEN 1.
      SELECT *      FROM sflight INTO wa_sflight
                    WHERE    ( carrid = wa_spfli-carrid )
                    AND      ( connid = wa_spfli-connid ).

        WRITE: /, wa_sflight-carrid,
               wa_sflight-connid,
               wa_sflight-fldate,
               wa_sflight-price,
               wa_sflight-currency,
               wa_sflight-planetype,
               wa_sflight-seatsmax,
               wa_sflight-seatsocc,
               wa_sflight-paymentsum,
               wa_sflight-seatsmax_b,
               wa_sflight-seatsocc_b,
               wa_sflight-seatsmax_f,
               wa_sflight-seatsocc_f.
       ENDSELECT.
       if sy-subrc <> 0.
           write: 'keine Ergebnisse'.
       endif.
      WHEN OTHERS.
        WRITE: 'Keine Selektion', sy-lsind.
    ENDCASE.
```

Zum Zeitpunkt AT LINE-
SELECTION muss nicht immer eine weitere Liste aufgerufen werden. Auch der
Aufruf eines Dialogs ist möglich. Im folgenden Beispiel ist der Aufruf e
ines Funktionsbaustein gezeigt. Die Besonderheit an diesem Programm ist,
dass die Sonderfunktion READ TEXTPOOL programmname INTO itab LANGUAGE spr
ache verwendet wird. Darüber hinaus wird eine interne Tabelle mit Kopfzei

le verwendet. Die Kopfzeile hat den Sinn den Zugriff auf die Tabellenzeil
e zu erlauben, in dem der gewünschte Satz aus dem Tabellenbereich in die
Kopfzeile geschrieben wird. Alternativ könnte mit LOOP zugegriffen werden
bzw. mit READ in einen Arbeitsbereich gelesen werden.

```
*&---------------------------------------------------------------------*
*& Report  Z01_HIDE_UND_FUNKTIONSBAUSTEIN                              *
*&                                                                     *
*&---------------------------------------------------------------------*
*&                                                                     *
*&                                                                     *
*&---------------------------------------------------------------------*

REPORT  z01_hide_und_funktionsbaustein  LINE-SIZE 250        .
*-------- trdir enthält die Namen der Programme -------------------*
*-------- trdir enthält nur allgemeine Verwaltungsinformationen --*
DATA: wa     TYPE trdir,
*-------- textpool enthält die Texte des Programms
      watext TYPE textpool.


*--- Interne Tabelle mit der Textstruktur und Kopfzeile -----*
DATA: BEGIN OF itext OCCURS 20.
        INCLUDE STRUCTURE textpool.
DATA: END OF itext.

START-OF-SELECTION.
*---- Lesen des Programmnames -----------*
  SELECT * FROM trdir INTO wa WHERE
                  cnam     = sy-uname
                  AND appl <> 'S' AND
                  subc IN ('1', 'M', 'F').
    WRITE: /, wa-name.
*---- Sicherung des Satzes für at line-selection
    HIDE wa.

*---- Rücksetzen auf den typgerechten Initialwert der Kopfzeile
    CLEAR itext.
*---- Rücksetzen der internen Tabelle ohne Kopfzeile
    REFRESH itext.
*--- Sonderbefehl zum Lesen von Textelementen eines Programms
*--- READ TEXTPOOL programmname INTO itab LANGUAGE sprache.
    READ TEXTPOOL wa-name INTO itext LANGUAGE sy-langu.
*---- Finden der Beschreibung mit Text_ID = "R" über die Kopfzeile
*---- Kopfzeile Initialisiert
    CLEAR itext.
*---- Suchbegriff in der Kopfzeile
    itext-id = 'R'.
*--- Lesen aus der Tabellenzeile in die Kopfzeile
    READ TABLE itext.
*---- Schreiben der Beschreibung
    IF sy-subrc = 0.
      WRITE: /(60) itext-entry.
      HIDE wa.
    ENDIF.
    WRITE /.

  ENDSELECT.

  CLEAR wa.
```

```
AT LINE-SELECTION.
  IF NOT wa IS INITIAL.

    CALL FUNCTION 'EDITOR_PROGRAM'
      EXPORTING
        program    = wa-name
        MESSAGE    = ' '
        display    = 'X' "spaces für editierbar, x für anzeigen
        trdir_inf = wa.
    CLEAR wa.
```

3.6 Parameters, Select-Options, Selection-Screen

Im Nachfolgenden werden die folgenden Reportdialogbefehle erläutert:

- **Selektionsbildschirm:** Select-Options erlaubt die Ausgrenzung von bis
- **Parameters:** Erlaubt die Eingabe von Eingabewerten
- **Selection-screen:** Programmierung von Selektionsbildschirmen**.**

Beispiel: Radiobutton und CHECKBOX

```
REPORT  z_interne_tabellen_b03_01                                .


*------ Definition einer Tabelle -------------------*
DATA: wadb TYPE STANDARD TABLE OF spfli,
      wa   TYPE spfli.

*---------------------------------------------------------------------*
*---------------------------------------------------------------------*

*---- Aufbau eines Selektionsbildschirms----------*
SELECT-OPTIONS s_carrid FOR wa-carrid.
*---- Parameters und Auswahlbilder
SELECTION-SCREEN SKIP.
PARAMETERS: p_carrid TYPE spfli-carrid.

*--- Leerzeichen auf dem Bildschirm
SELECTION-SCREEN SKIP.

*---- Checkbox mit Auswahl
SELECTION-SCREEN BEGIN OF BLOCK sortierung1 WITH FRAME.
PARAMETERS: auswahl1 AS CHECKBOX,
            auswahl2 AS CHECKBOX,   " DEFAULT 'X',
            auswahl3 AS CHECKBOX.
SELECTION-SCREEN END OF BLOCK sortierung1.

SELECTION-SCREEN SKIP.
SELECTION-SCREEN BEGIN OF BLOCK sortierung2 WITH FRAME.
PARAMETERS: radio1 RADIOBUTTON GROUP gr1,
            radio2 RADIOBUTTON GROUP gr1, " DEFAULT 'X',
            radio3 RADIOBUTTON GROUP gr1.
SELECTION-SCREEN END OF BLOCK sortierung2.

*---------- Bereich der Initialisierung
```

```
INITIALIZATION.
  CLEAR wadb.
  CLEAR wa.

*------ Verarbeitungsblock vor Screen-Aufruf
AT SELECTION-SCREEN OUTPUT.
radio1 = 'X'.

*------ Verarbeitungsblock nach Screen-Aufruf
AT SELECTION-SCREEN.

  IF auswahl1 <> 'X' AND auswahl2 <> 'X' AND auswahl3 NE 'X'.
    MESSAGE e001(zhoh).
*   Keine Daten eingegeben !!!!!


  ENDIF.



START-OF-SELECTION.
  PERFORM db_tabelle.

*&---------------------------------------------------------------------*
*&      Form  db_tabelle
*&---------------------------------------------------------------------*
*       text
*----------------------------------------------------------------------*
FORM db_tabelle.
*----- Ausgrenzung einer Selektion in dem Select-Anweisung
  SELECT * FROM spfli INTO TABLE wadb WHERE carrid IN s_carrid.
  PERFORM fehlermeldung.
*---- Sortierung der internen Tabellen
  IF radio1 = 'X'.
    SORT wadb BY carrid.
  ENDIF.
  IF radio2 = 'X'.
    SORT wadb BY connid.
  ENDIF.
  IF radio3 = 'X'.
    SORT wadb BY countryfr connid.
  ENDIF.
*---- Ausgabe der internen Tabelle
  LOOP AT wadb INTO wa.
    WRITE:
            wa-carrid,
            wa-connid,
            wa-countryfr,
            wa-cityfrom,
            wa-airpfrom,
            wa-countryto,
            wa-cityto,
            wa-airpto,
            wa-fltime,
            wa-deptime,
            wa-arrtime,
            wa-distance,
            wa-distid,
            wa-fltype,
            wa-period,
```

```
          / .
  ENDLOOP.


ENDFORM.                    "db_tabelle
*&---------------------------------------------------------------------*
*&      Form  verarbeite_index
*&---------------------------------------------------------------------*
*       text
*----------------------------------------------------------------------*
FORM fehlermeldung.
  IF sy-subrc <> 0.
    WRITE: 'Fehler in der Tabelle'.
  ENDIF.
ENDFORM.                    "fehlermeldung
```

Beispiel: Funktionstasten und deren Bearbeitung

```
REPORT  z01_selektionsbildschirm                                .


DATA: g_ucomm LIKE sscrfields-ucomm,
      wa LIKE sscrfields.
*---- Bildschirminterface
TABLES sscrfields.
*--- Drei Leerzeilen
SELECTION-SCREEN SKIP 3.
*---- Funktionstasten
SELECTION-SCREEN FUNCTION KEY: 1, 2.
*--- Pushbutton und deren Positionen
SELECTION-SCREEN PUSHBUTTON 5(15)  pb1 USER-COMMAND 0001.
SELECTION-SCREEN PUSHBUTTON 25(15) pb2 USER-COMMAND 0002.
SELECTION-SCREEN PUSHBUTTON 45(15) pb3 USER-COMMAND 0003.
*---- Texte füllen
INITIALIZATION.
  sscrfields-functxt_01 = 'T1'.
  sscrfields-functxt_02 = 'T2'.
  pb1 = 'Taste 3'.
  pb2 = 'Taste 4'.
  pb3 = 'Taste 5'.
*--- Sicherung von ucomm, Auslöse Ucomm setzen
At selection-screen.
g_ucomm = sscrfields-ucomm.
SSCRFIELDS-ucomm = 'ONLI'.

start-of-selection.
  WRITE: 'Ausgelöst wurde die Taste'.
  CASE g_ucomm.
    WHEN 'FC01'.
      WRITE 1.
    WHEN 'FC01'.
      WRITE 1.
    WHEN 'FC02'.
      WRITE 2.
    WHEN '0001'.
      WRITE 3.
    WHEN '0002'.
      WRITE 4.
```

```
    WHEN '0003'.
      WRITE 5.

  ENDCASE.
```

Beispiel: Aufruf verschiedener Bildschirme

```
REPORT  z01_selektionsbs_v2                                              .

*---- Bildschirminterface
TABLES sscrfields.
*--- Steuerungsparameter
DATA: g_ucomm    TYPE sscrfields-ucomm,
      sich_ucomm type sscrfields-ucomm.
*--- Bildschirmmaske für separater Aufruf
SELECTION-SCREEN BEGIN OF SCREEN 10 TITLE win01 AS WINDOW.
PARAMETERS: a(20) TYPE c.
SELECTION-SCREEN END OF SCREEN 10.

SELECTION-SCREEN BEGIN OF SCREEN 20 TITLE win02 AS WINDOW.
PARAMETERS: b(20) TYPE c.
SELECTION-SCREEN END OF SCREEN 20.


*--- Pushbutton und deren Positionen
SELECTION-SCREEN PUSHBUTTON 5(15)  pb1 USER-COMMAND 0001.
SELECTION-SCREEN PUSHBUTTON 25(15) pb2 USER-COMMAND 0002.


*---- Texte füllen
INITIALIZATION.
  pb1 = 'Lesen der SFLIGHT'.
  pb2 = 'Lesen der SPFLI'.

*--- Sicherung von ucomm, Auslöse Ucomm setzen
AT SELECTION-SCREEN.
  CLEAR: a, b.
  MOVE sscrfields-ucomm TO g_ucomm.
  CASE g_ucomm.
    WHEN '0001'.
      move g_ucomm to sich_ucomm.
      CALL SELECTION-SCREEN 10.
      MOVE 'ONLI' TO sscrfields-ucomm.
    WHEN '0002'.
      move g_ucomm to sich_ucomm.
      CALL SELECTION-SCREEN 20.
      MOVE 'ONLI' TO sscrfields-ucomm.
  ENDCASE.

START-OF-SELECTION.

  WRITE: / 'Parameter A:', a.
  WRITE: / 'Parameter B:', b.

*  CASE sich_ucomm.
*    WHEN '0001'.
*      PERFORM lesen_sflight.
*    WHEN '0002'.
```

```
*      PERFORM lesen_spfli.
*  ENDCASE.

*---------------------------------------------------------------------*
*  FORM lesen_sflight
*---------------------------------------------------------------------*
*
*---------------------------------------------------------------------*
FORM lesen_sflight.
  DATA: wa TYPE sflight.
  ULINE.
  SELECT * FROM sflight INTO wa.
    WRITE: / wa-connid, wa-carrid.
  ENDSELECT.

ENDFORM.                    "lesen_sflight
*&---------------------------------------------------------------------*
*&      Form  lesen_spfli
*&---------------------------------------------------------------------*
*       text
*---------------------------------------------------------------------*
*  -->  p1        text
*  <--  p2        text
*---------------------------------------------------------------------*
FORM lesen_spfli .
  DATA: wa TYPE spfli.
  ULINE.
  SELECT * FROM spfli INTO wa.
    WRITE: / wa-connid, wa-carrid.
  ENDSELECT.
```

3.7 Message-System

Um das Message-System zu verwenden müssen folgende Schritte durchlaufen werden:

- Anlage einer Gruppe
- Anlage von Meldungen
- Zeigen von Übergabevariablen

Der allgemeiner Aufbau eines Message-Befehls ist:

MESSAGE ID '<message class>' TYPE '<message_type>'
NUMBER <nnn> WITH <v1> <v2> < v3> < v4>.

- message_class = Messageklasse, Auswählbar oder mit Vorwärtsnavigation anlegbar
- t = Message Typ

Typ i	Bedeutung	Verarbeitung
S	Statusnachricht	Das Programm wird nach der MESSAGE fortgesetzt.
I	Information	Das Programm wird nach der MESSAGE fortgesetzt

W	Warnung	Der Verarbeitungsblock wird abgebrochen, die Anzeige der vorhergehenden Listenstufe bleibt erhalten.
E	Fehler	Der Verarbeitungsblock wird abgebrochen, die Anzeige der vorhergehenden Listenstufe bleibt erhalten.
A	Abbruch	Das Programm wird abgebrochen
X	Kurzdump	Der Verarbeitungsblock wird abgebrochen, die Anzeige der vorhergehenden Listenstufe bleibt erhalten.

- nnn = Messagenummer, dreistellige Nummer, Auswählbar oder durch Vorwärtsnavigation anlegbar.
- v1 bis v4 = Übergabe bis zu vier Informationen an die Messagebox . Diese werden, abhängig vom Nachrichttext, dynamisch in diese eingesetzt.

Beispielprogramm:

```
REPORT  zbeispiel_message_system      .

TYPES: type_satz TYPE spfli.
DATA:  satz TYPE type_satz.
PARAMETER: ausw TYPE spfli-carrid.


START-OF-SELECTION.

  SELECT SINGLE * FROM spfli INTO satz WHERE carrid = ausw.
*--- Neue Klasse anzeigen z.B. ZDEVEL01
  IF sy-subrc NE 0.
    message id 'ZSS2002' type 'I' number 000 with ausw.
  else.
    message id 'ZSS2002' type 'I' number 003.
    WRITE: satz-carrid,
           satz-connid,
           satz-countryfr,
           satz-cityfrom,
           satz-airpfrom,
           satz-countryto,
           satz-cityto.

  ENDIF.
```

3.8 Funktionsbausteine

Struktur von Funktionsbausteinen

- Import-Parameter (Übergabe an den Funktionsbaustein ACHTUNG: Variablenübertragung von RECHTS nach LINKS)

- Export-Parameter (Rückgabe aus dem Funktionsbaustein ACHTUNG: Variablenübertragung von LINKS nach RECHTS)
- Changing-Parameter (Ein- und Ausgabe in und aus dem Funktionsbaustein ACHTUNG: Variablenübertragung von RECHTS nach LINKS)
- Tabellen-Parameter (Übergabe interner Tabellen an den Funktionsbaustein ACHTUNG: Variablenübertragung von RECHTS nach LINKS)
- Ausnahmen (Exceptions); (Ausnahmeregeln: exceptions feldname ACHTUNG: Fangen der Fehler über RAISE und Auswertung in Hauptprogramme in der Systemvariablen sy-subrc).
- Anlage einer Funktionsgruppe (Funktionsgruppen müssen vor der Verwendung eines Bausteins angelegt werden)
- Anlage eines Funktionsbausteins

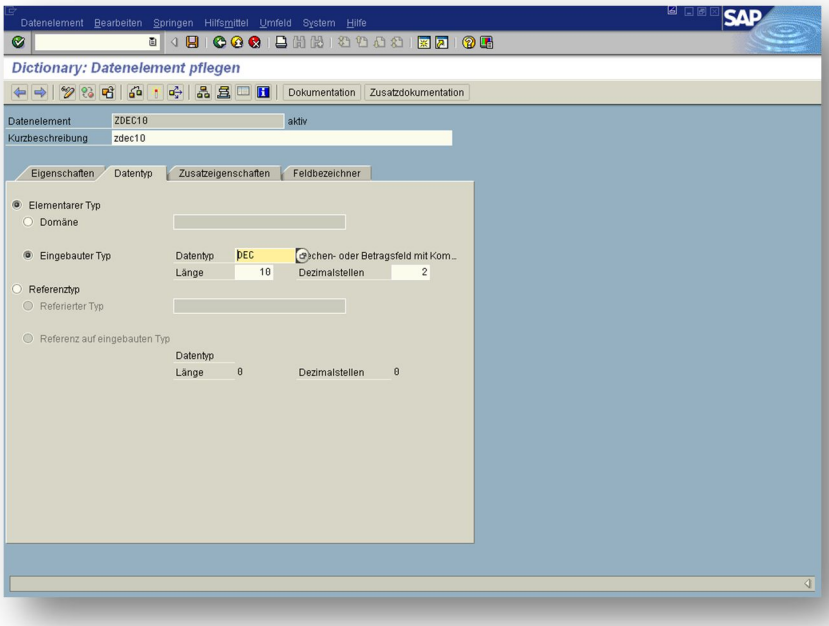
Einfaches Beispiel: Rechnen in Funktionsbausteinen

Hauptprogramm

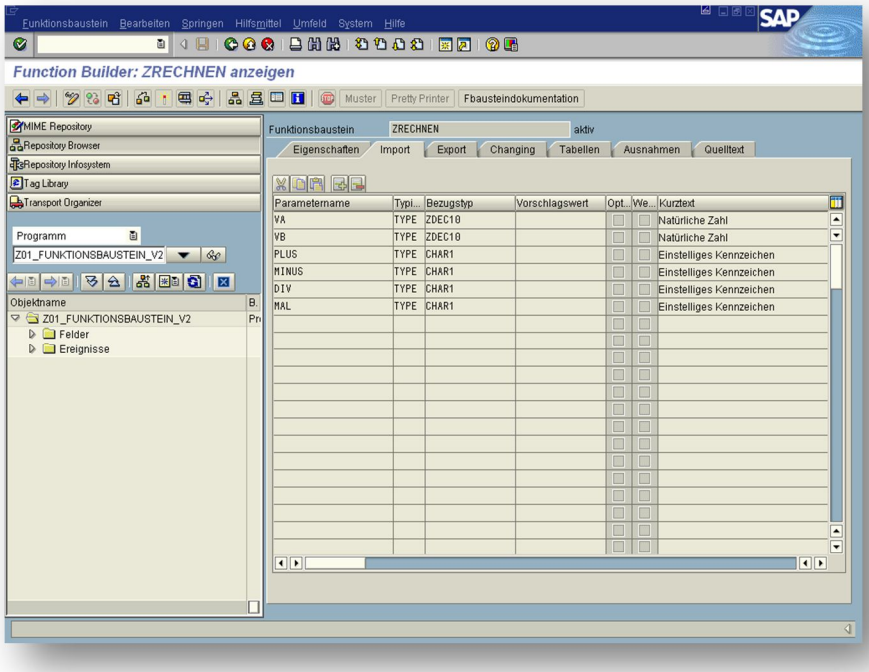
```
*&---------------------------------------------------------------------*
*& Report  Z HITE BEISPIEL01                                           *
*&                                                                     *
*&---------------------------------------------------------------------*
* sy-lsind = aktueller Listenindex

*&---------------------------------------------------------------------*
*& Report  Z_INTERNE_TABELLEN_B03_01                                   *
*&                                                                     *
*&---------------------------------------------------------------------*
*&                                                                     *
*&                                                                     *
*&---------------------------------------------------------------------*
*&---------------------------------------------------------------------*
*& Report  Z01_SELEKTIONSBILDSCHIRM                                    *
*&                                                                     *
*&---------------------------------------------------------------------*
*&                                                                     *
*&                                                                     *
*&---------------------------------------------------------------------*

*&---------------------------------------------------------------------*
*& Report  Z01_SELEKTIONSBS_V2                                         *
*&                                                                     *
*&---------------------------------------------------------------------*
*&                                                                     *
*&                                                                     *
*&---------------------------------------------------------------------*


*&---------------------------------------------------------------------*
*& Report  ZBEISPIEL_MESSAGE_SYSTEM                                    *
*&---------------------------------------------------------------------*
*& Report  Z01_FUNKTIONSBAUSTEIN_V2                                    *
*&                                                                     *
*&---------------------------------------------------------------------*
*&                                                                     *
*&                                                                     *
*&---------------------------------------------------------------------*
```

```
REPORT  z01_funktionsbaustein_v2                                        .
data: erg type zdec12.
SELECTION-SCREEN SKIP 2.
PARAMETERS: plus RADIOBUTTON  GROUP gr1 DEFAULT 'X',
            minus RADIOBUTTON GROUP gr1,
            mal   RADIOBUTTON GROUP gr1,
            div   RADIOBUTTON GROUP gr1.
SELECTION-SCREEN ULINE.
PARAMETERS: a TYPE zdec10,
            b TYPE zdec10.

SELECTION-SCREEN ULINE.
*PARAMETERS: erg TYPE zdec12.

AT SELECTION-SCREEN OUTPUT.


AT SELECTION-SCREEN.
*--- Lösung mit einem Unterprogramm
*  PERFORM zrechnen USING a b erg.
*--- Lösung mit einem Funktionsbaustein
  CALL FUNCTION 'ZRECHNEN'
    EXPORTING
      va     = a
      vb     = b
      plus   = plus
      minus  = minus
      div    = div
      mal    = mal
    IMPORTING
      verg   = erg
    EXCEPTIONS
      fehler = 1
      OTHERS = 2.

 start-of-selection.
 write: 'Ergebnis.:  ', erg.
  .
*---- Alternative Lösung-------------------*
*FORM zrechnen USING va vb verg.
*  IF plus = 'X'.
*    COMPUTE verg = va + vb.
*  ENDIF.
*  IF minus = 'X'.
*    COMPUTE verg = va - vb.
*  ENDIF.
*  IF mal = 'X'.
*    COMPUTE verg = va * vb.
*  ENDIF.
*  IF div = 'X'.
*    IF va NE 0 AND vb NE 0.
*      COMPUTE verg = va / vb.
*    ENDIF.
*  ENDIF.
*ENDFORM.                    " Zrechnen
```

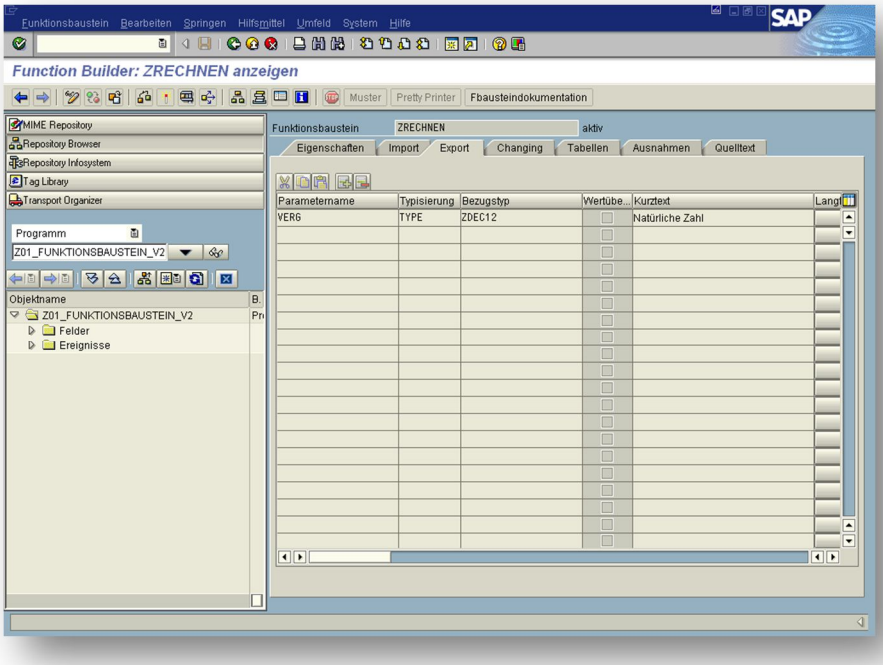
Anlage und Verwendung von Datentypen aus dem DD (ZDEC10 und ZDEC12)



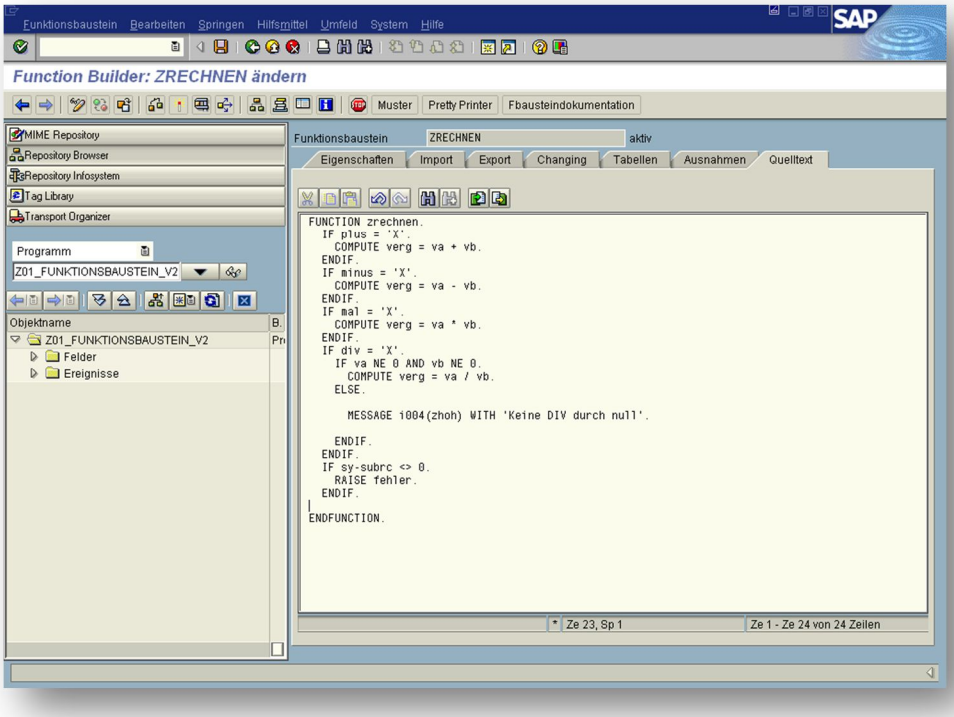
- Funktionsbaustein (Inhalt und Übergabevariablen)
  - Import (Karteikarte)



Export (Karteikarte)



- ## Quelltext (Karteikarte)



- **Beispiel: Lesen und Ausgabe der Inhalte von SPLIGHT über Funktionsbausteine**

- **Hauptprogramm:**

```
REPORT  z01_funktionsbaustein_v1                                .

PARAMETERS: e_carrid TYPE sflight-carrid,
            e_connid TYPE sflight-connid.


DATA: tabelle TYPE zsflight_tab,
      wa      TYPE sflight.

START-OF-SELECTION.

  CALL FUNCTION 'ZLESEN_SFLIGHT'
*--- Übergabevariablen von RECHTS nach LINKS
   EXPORTING
     ueberg_carrid               = e_carrid
     ueberg_connid               = e_connid
*----- Lesen der eigen definierten Tabelle -----------*
*----- Rückgabevariablen von LINKS nach RECHTS
  IMPORTING
    RUECK SFLIGHT TABELLE       = tabelle
*-------- Übergabe mit der Funktionsbausteintabelle ---*
*    TABLES
*      s_tab                      = tabelle
   EXCEPTIONS
     fehler                     = 1
     OTHERS                     = 2.
*------ Auswertung des Fehlers --------------*
  CASE sy-subrc.
    WHEN 1.
      MESSAGE i004(zhoh) WITH 'Keinen Datensatz gefunden' sy-subrc.
    WHEN 2.
      MESSAGE i004(zhoh) WITH 'Anderer Fehler' sy-subrc.
  ENDCASE.

*--- Ausgabe der Tabelle in einer Liste über den Funktionsbaustein
CALL FUNCTION 'ZAUSGABE_SFLIGHT'
 EXPORTING
   AUSG_TAB        = tabelle
* TABLES
*   AUSG_TABE       =
  EXCEPTIONS
    FEHLER          = 1
    OTHERS          = 2.
*---- Abfangen des Fehlers, dass die Tabelle leer ist (&1 &2---*
IF sy-subrc <> 0.
      MESSAGE i004(zhoh) WITH 'Tabelle ohne Inhalt' sy-subrc.
ENDIF.

FUNCTION zlesen_sflight.
*"----------------------------------------------------------------------
*"*"Lokale Schnittstelle:
```

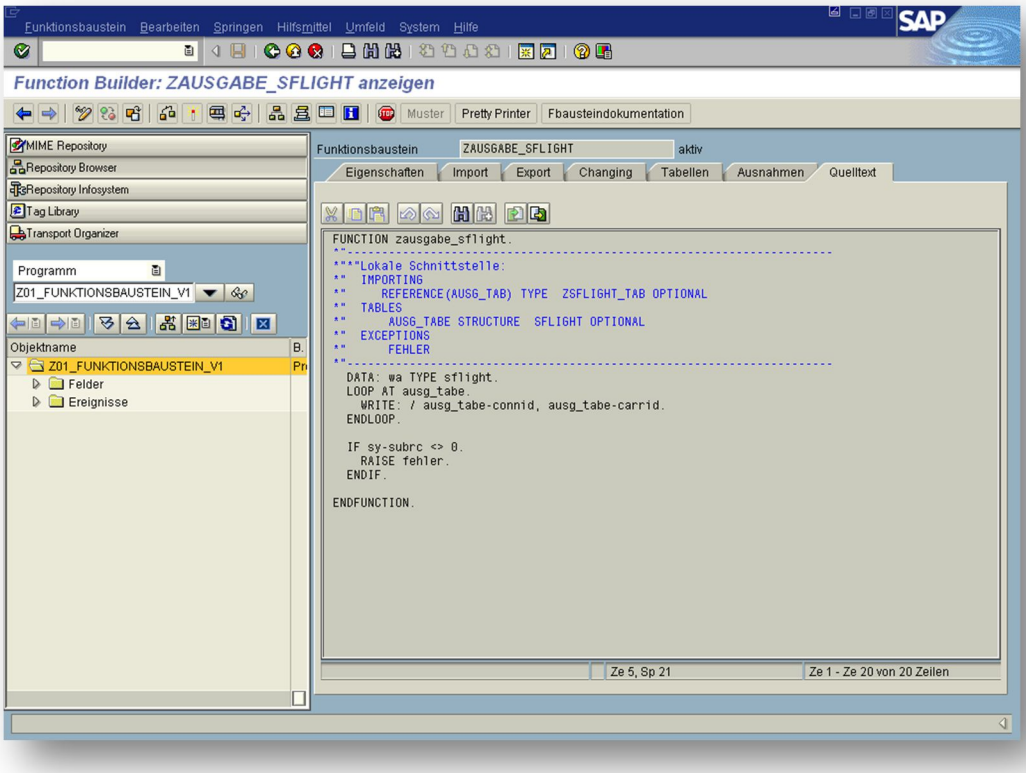
```
*"  IMPORTING
*"     REFERENCE(UEBERG_CARRID) TYPE  SFLIGHT-CARRID
*"     REFERENCE(UEBERG_CONNID) TYPE  SFLIGHT-CONNID
*"  EXPORTING
*"     REFERENCE(RUECK_SFLIGHT_TABELLE) TYPE  ZSFLIGHT_TAB
*"  TABLES
*"      S_TAB STRUCTURE  SFLIGHT
*"  EXCEPTIONS
*"      FEHLER
*"----------------------------------------------------------------------

  SELECT * FROM sflight INTO TABLE rueck_sflight_tabelle WHERE carrid =
  ueberg_carrid AND connid = ueberg_connid.

  SELECT * FROM sflight INTO TABLE s_tab WHERE carrid =
  ueberg_carrid AND connid = ueberg_connid.
  IF sy-subrc = 4.
    RAISE nothing_found.
  ENDIF.

ENDFUNCTION.
```

- **Funktionsbausteine1 (ZLESEN_SFLIGHT)**
- IMPORT-Parameter
  (ueberg_carrid type sflihgt_carrid)
  (ueberg_connid type sflihgt_connid)
- EXPORT-Parameter
  ACHTUNG: Anlage einer Internen-Tabelle im DD als Rückgabe aus Funktionsbaustein ZLESEN_SFLIGHT mit dem Namen ZSFLIGHT_TAB oder Alternativ Verwendung der Tabelle im Funktionsbaustein
- EXCEPTION-Variable
  z.B. FEHLER
- **Programmcode**

```
FUNCTION zlesen_sflight.
*"----------------------------------------------------------------------
*"*"Lokale Schnittstelle:
*"  IMPORTING
*"     REFERENCE(UEBERG_CARRID) TYPE  SFLIGHT-CARRID OPTIONAL
*"     REFERENCE(UEBERG_CONNID) TYPE  SFLIGHT-CONNID OPTIONAL
*"  EXPORTING
*"     REFERENCE(RUECK_SFLIGHT_TABELLE) TYPE  ZSFLIGHT_TAB
*"  TABLES
*"      S_TAB STRUCTURE  SFLIGHT OPTIONAL
*"  EXCEPTIONS
*"      FEHLER
*"----------------------------------------------------------------------
*--- Lesen in eine eigen definierte Tabelle--------------------------*
  SELECT * FROM sflight INTO TABLE rueck_sflight_tabelle
            WHERE carrid =  ueberg_carrid AND connid = ueberg_connid
            .
*---- Lesen in die Tabelle des Funktionsbausteins -------------------*
SELECT * FROM sflight INTO TABLE s_tab
         WHERE carrid = ueberg_carrid AND connid = ueberg_connid
         .
```

```
  IF sy-subrc = 4.
    RAISE fehler.
  ENDIF.


ENDFUNCTION.
```

- **Funktionsbaustein 2 (ZSCHREIBEN_SFLIGHT)**
  - IMPORT-Parameter : Keine
    AUSG_TABELLE Tabelle mit der Typisierung ZSFLIGHT_TAB aus dem DD oder Definition in Tabelle
  - Keinen EXPORT-Parameter
  - EXCEPTIONS Fehler (falls Tabelle leer)



3.9 Select

Das nachfolgende Beispiel zeigt die Eingabe von Daten in einen Parameterbildschirm und Speicherung der Daten in der Datenbank. Es wird mit einer eigenen Datenbanktabelle gearbeitet.

```
REPORT  z_eigene_tabellen    LINE-SIZE 200.


PARAMETERS: akt     TYPE char1,
```

```
            kunr    TYPE zkunde01-kunr,
            kuname  TYPE zkunde01-kuname,
            kustr   TYPE zkunde01-kuort,
            kuplz   TYPE zkunde01-kuplz,
            kuort   TYPE zkunde01-kuort,
            kunotiz TYPE zkunde01-kunotiz.


DATA: wa TYPE zkunde01.

AT SELECTION-SCREEN OUTPUT.
  akt = 'D'.

AT SELECTION-SCREEN.

  IF akt <> 'N' AND
     akt <> 'L' AND
     akt <> 'Ä' AND
     akt <> 'D'.
    MESSAGE e009(zhoh).
*   Keine Daten eingegeben !!!!!
  ENDIF.

  IF ( kunr   = 0   OR
       kuname = ' ' OR
       kuort  = ' ' ) AND
       ( akt NE 'D' ).
    MESSAGE e010(zhoh).
  ENDIF.

  CASE akt.
    WHEN 'Ä'.
      SELECT SINGLE * FROM zkunde01 INTO wa WHERE kunr = kunr.
      IF sy-subrc <> 0.
        WRITE: 'Update nur bei gültigen Satz möglich'.
      ELSE.
        PERFORM einstellen.
        UPDATE zkunde01 FROM wa.
      ENDIF.
    WHEN 'L'.
      DELETE FROM zkunde01 WHERE kunr = kunr.
      IF sy-subrc <> 0.
        ROLLBACK WORK.
      ELSE.
        WRITE: 'Erfolgreich gelöscht'.
      ENDIF.
    WHEN 'N'.
      SELECT SINGLE * FROM zkunde01  INTO wa WHERE kunr = kunr.
      IF sy-subrc <> 0.
        PERFORM einstellen.
        INSERT zkunde01 FROM wa.
      ELSE.
        WRITE: 'Neuanlage nicht möglich, da Satz bereits vorhanden '.
      ENDIF.
  ENDCASE.

START-OF-SELECTION.
  SELECT * FROM zkunde01 INTO wa.
    WRITE: /, wa-kunr, wa-kuname, wa-kustr,
              wa-kuplz, wa-kuort, wa-kunotiz.
```

```
  ENDSELECT.

*&---------------------------------------------------------------------*
*&      Form  einstellen
*&---------------------------------------------------------------------*
*       text
*----------------------------------------------------------------------*
FORM einstellen.
  wa-kunr   = kunr.
  wa-kuname = kuname.
  wa-kuplz  = kuplz.
  wa-kuort  = kuort.
  wa-kunotiz  = kunotiz.

ENDFORM.                    "einstellen
```

3.10 Dialogprogrammierung

3.10.1 Einfache Dynproprogrammierung

Das nachfolgende Beispiel zeigt ein Dynproprogramm mit Verwendung von

- Eingabefeldern,
- Pushbutton und
- Radiobutton.

Des Weiteren werden das Message-System und der Reportgenerator aus dem Dialog aufgerufen.
Die Bestandteile des Programms sind:

- ABAP-Programm
- Dynpro
- Statuszeilen
- Messagesystem

ABAP-Programm

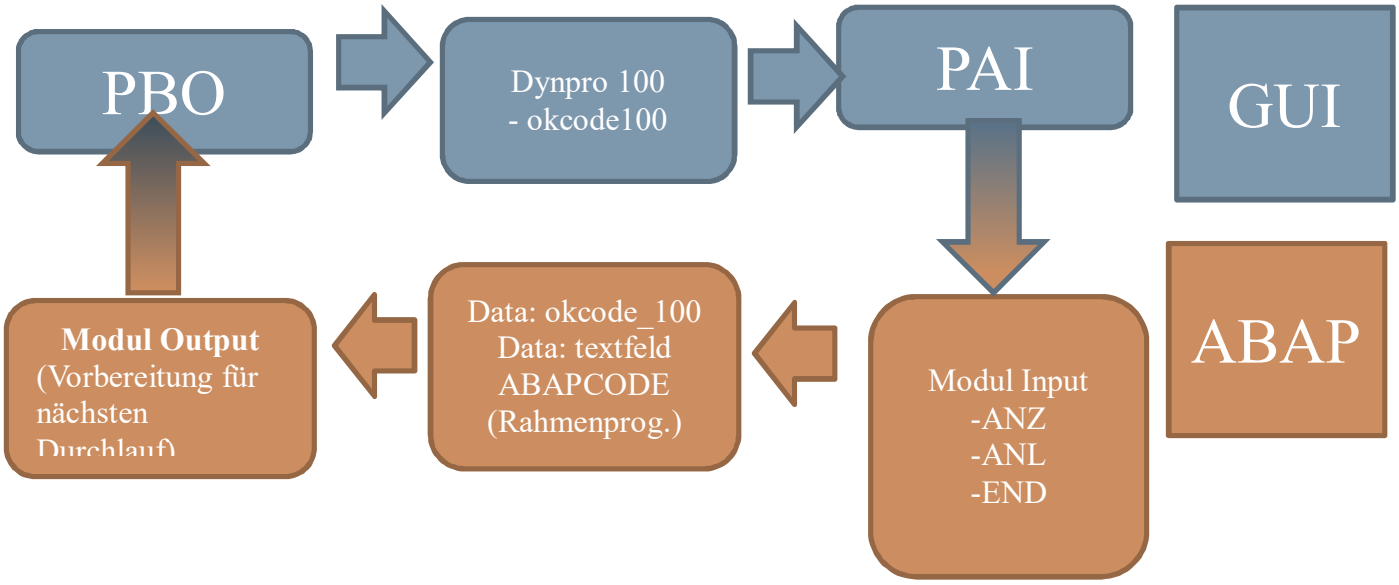
```
REPORT  z01_dynpro_v2                                                  .
*---- Typisierung einer Struktur -----------*
TYPES: BEGIN OF kuadr,
           kuname(30) TYPE c,
           kuort(30)  TYPE c,
       END OF kuadr.
*------- Instanzierung der Struktur --------*
DATA: wa TYPE kuadr.
*-------- Meldungsfeld ---------------------*
DATA: meld(80) TYPE c.
*------ Kommunikationsvariable für die Radiobuttons,OKCODE(identischer
*------ Inhalt mit sy-ucomm, Checkboxen--------------*
DATA: cb1(1) TYPE c,
      cb2(1) TYPE c,
      cb3(1) TYPE c,
      rb1(1)  TYPE c,
      rb2(1)  TYPE c,
      rb3(1) TYPE c,
      okcode LIKE sy-ucomm.

START-OF-SELECTION.
  CLEAR: sy-ucomm, okcode.
*---- Dynproaufruf (laden der Maske)-----------*
  CALL SCREEN '100'.

*&---------------------------------------------------------------------*
*&      Module  STATUS_0100  OUTPUT
*&---------------------------------------------------------------------*
MODULE status_0100 OUTPUT.
*--- Statuszeile und Titel --------------*
  SET PF-STATUS 'STAT1'.
  SET TITLEBAR  'TITL1'.
```

```
*--- Vorbelegung check-box---------------*
  MOVE 'X'   TO cb1.
*---  Vorbelegung Radio-Button ----------*
  MOVE 'X'   TO rb2.
  CLEAR: rb1,rb2.

ENDMODULE.                 " STATUS_0100  OUTPUT
*&---------------------------------------------------------------------*
*&      Module  USER_COMMAND_0100  INPUT
*&---------------------------------------------------------------------*
MODULE user_command_0100 INPUT.
  CASE okcode.
    WHEN 'END'.
*------ Verlassen des Dialogs ------------*
      LEAVE TO SCREEN 0.
    WHEN 'BUT1' .
*------ Buttondruckauswertung wurde gedrückt ----------*
      MOVE 'Button 1 gedrückt' TO meld.
      PERFORM ausgabe.
    WHEN 'BUT2'.
      MOVE 'Button 2 gedrückt' TO meld.
      PERFORM ausgabe.
  ENDCASE.
ENDMODULE.                 " USER_COMMAND_0100  INPUT

*----- Auswertung der User-Aktion in der Liste ----------------*
AT USER-COMMAND.
  CASE sy-ucomm.
    WHEN 'END'.
      LEAVE LIST-PROCESSING.
  ENDCASE.
*---------------------------------------------------------------------*
*  FORM ausgabe
*---------------------------------------------------------------------*
FORM ausgabe.
*----- Prüfung auf Inhalt von sy-ucomm --------------*
  CASE okcode..
    WHEN 'BUT1'.
*----- Ausgabe einer Meldung mit Übergabeparameter --*
      MESSAGE s004(zhoh) WITH meld.
    WHEN 'BUT2'.
*----- Verzweigung auf den Listprozessor
      LEAVE TO LIST-PROCESSING AND RETURN TO SCREEN 100.
*----- Statusfeld setzen
      SET PF-STATUS 'STAT2'.
*----- Ausgabe über eine Liste
      WRITE: / 'Button 2'.
  ENDCASE.

ENDFORM.                    "ausgabe
*&---------------------------------------------------------------------*
*&      Module  kuname_modul  INPUT
*&---------------------------------------------------------------------*
*       text
*---------------------------------------------------------------------*
module kuname_modul input.
      move 'Feld Kuname wurde geändert' to meld.
      MESSAGE s004(zhoh) WITH meld.

endmodule.                 " kuname_modul  INPUT
```

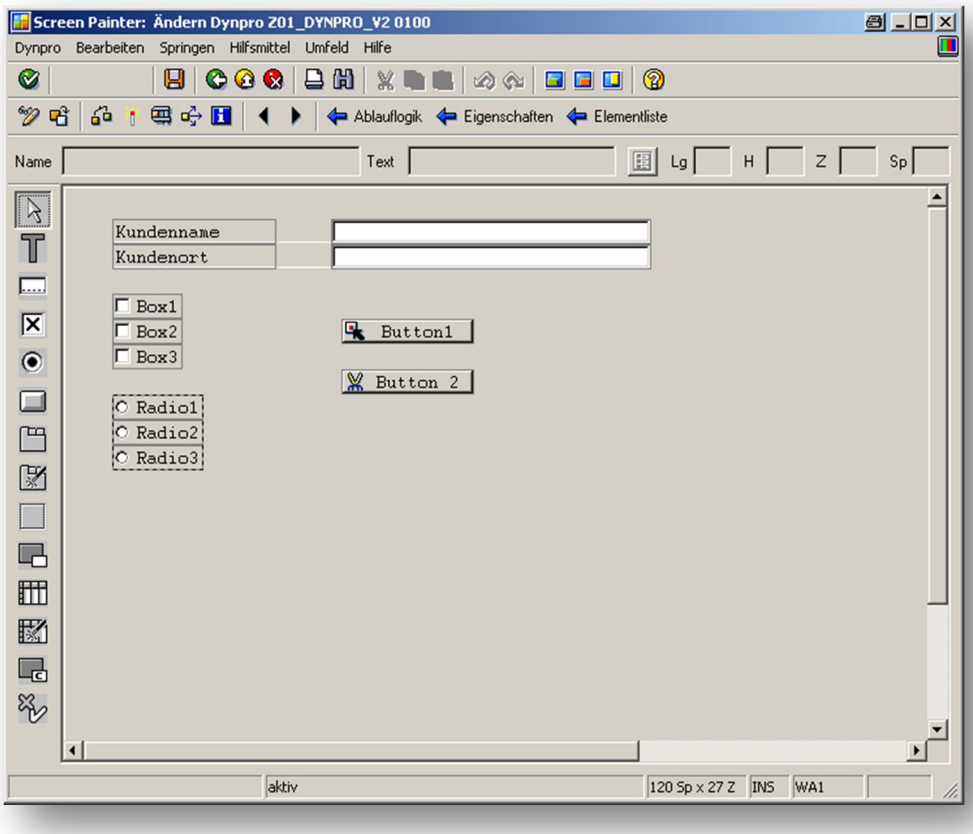
Dynpro

- **Ablauflogik**

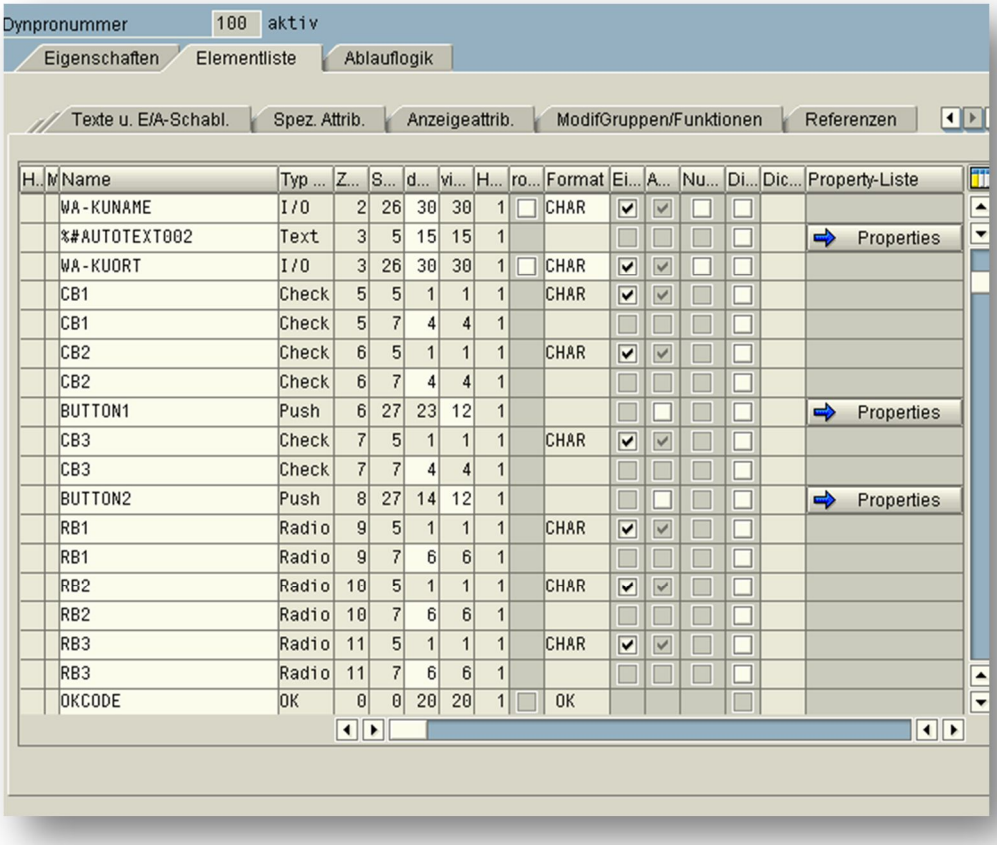
```
PROCESS BEFORE OUTPUT.
MODULE STATUS_0100.

PROCESS AFTER INPUT.
MODULE USER_COMMAND_0100.
FIELD wa-kuname MODULE kuname_modul on request.
```



- **Dynprobild**


Screen Painter: Ändern Dynpro Z01_DYNPRO_V2 0100
Dynpro Bearbeiten Springen Hilfsmittel Umfeld Hilfe
Ablauflogik Eigenschaften Elementliste
Name Text Lg H Z Sp
Kundenname
Kundenort
Box1
Box2
Box3
Button1
Button 2
Radio1
Radio2
Radio3
aktiv
120 Sp x 27 Z
INS
WA1

- **Elementenliste:** Die Variablen im ABAP-Programm haben den gleichen Namen



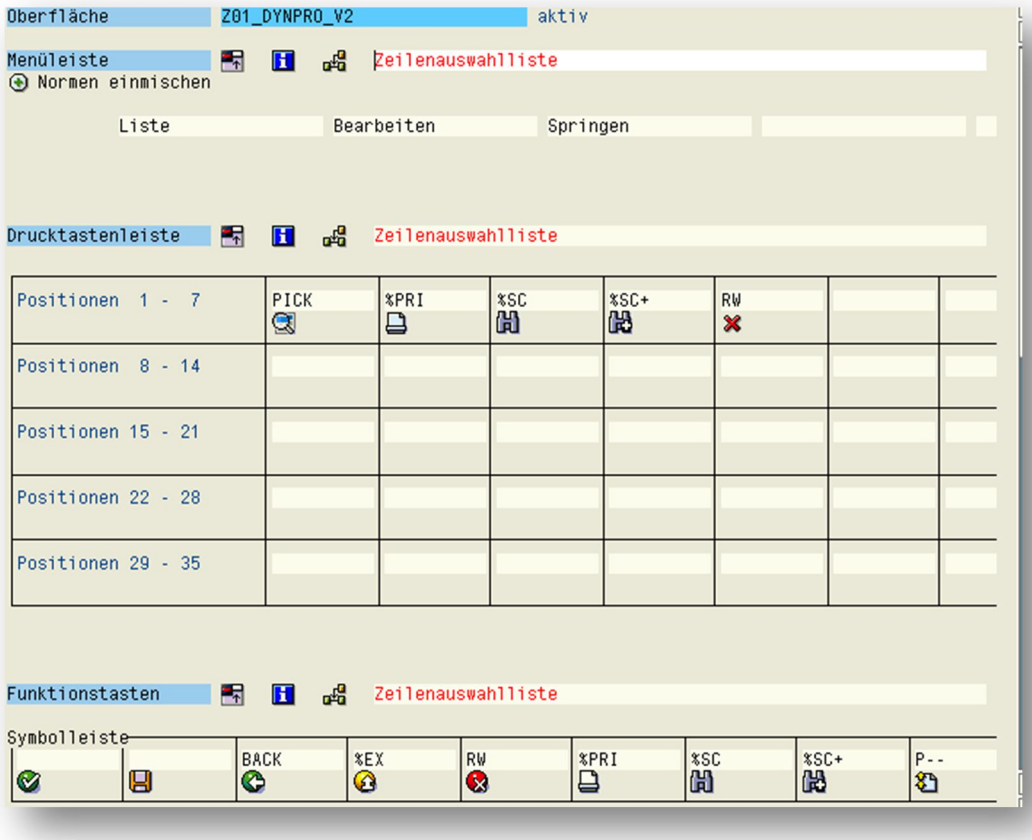
Statuszeilen

Für die Statuszeile der Dynpromaske(STAT1) sind in der Symbolleiste die Felder END und SPEICH aktiviert worden.

```
SET PF-STATUS 'STAT1'.
```

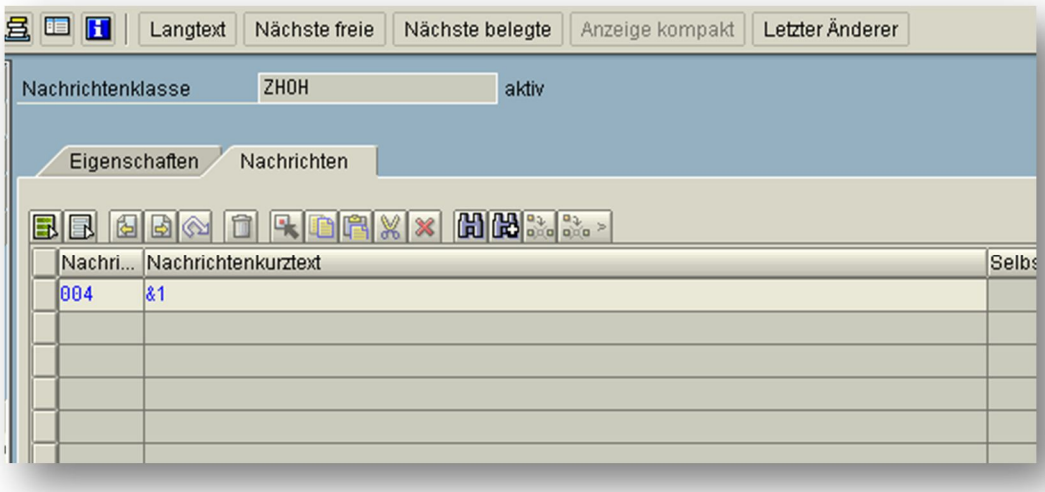
Für die Listenausgabe ist eine eigene Statuszeilendefinition (STAT2) erforderlich. Hier wurde mit der Vorbelegung über den Standard gearbeitet.

```
SET PF-STATUS 'STAT2'.
```



Messagesystem

```
MESSAGE s004(zhoh) WITH meld. "meld wird in &1 übergeben
```



Das nachfolgende Beispiel zeigt ein Dynproprogrammierung zur Pflege einer selbst angelegten Datenbanktabellen.

ABAP-Hauptprogramm

```
REPORT  z01_dynpro_v3                                             .

DATA:
*---- Workbereich für die Datenbanksätze ----*
      wa TYPE zkunde,
      neuanlage(1) TYPE c,
*---- Auslösemodus -------------------------*
      okcode LIKE sy-ucomm.
*---- Kommunikationsbereich DB-Tabelle -- Dynpro ---*
TABLES zkunde.

START-OF-SELECTION.
*---- Aufruf der Maske---------*
  CALL SCREEN '100'.

*&---------------------------------------------------------------------*
*&      Module  STATUS_0100  OUTPUT
*&---------------------------------------------------------------------*
*       text
*----------------------------------------------------------------------*
MODULE status_0100 OUTPUT.
  SET PF-STATUS '100'.
  CLEAR: okcode, wa.
*  SET TITLEBAR 'xxx'.
ENDMODULE.                 " STATUS_0100  OUTPUT
*&---------------------------------------------------------------------*
*&      Module  USER_COMMAND_0100  INPUT
*&---------------------------------------------------------------------*
*       text
*----------------------------------------------------------------------*
MODULE user_command_0100 INPUT.
  CASE okcode.
    WHEN 'ZURUECK'.
      ROLLBACK WORK.
      CLEAR: zkunde, wa, neuanlage.
    WHEN 'ANZ'.
      PERFORM anzeige.
    WHEN 'SPEICH'.
      PERFORM speichern.
    WHEN 'LOE'.
      PERFORM loeschen.
    WHEN 'END'.
      LEAVE TO SCREEN 0.
  ENDCASE.

ENDMODULE.                    "user_command_0100 INPUT
" USER_COMMAND_0100  INPUT

*----- User-Aktion auf die Liste --------------*
AT USER-COMMAND.
  CASE sy-ucomm.
    WHEN 'END'.
```

```
      LEAVE LIST-PROCESSING.
  ENDCASE.
*---------------------------------------------------------------------*
*  FORM anzeige
*---------------------------------------------------------------------*
FORM anzeige.
  LEAVE  TO LIST-PROCESSING.
  SET PF-STATUS 'LIST'.
  SELECT * FROM zkunde INTO wa.
    WRITE: / wa-kunr, wa-kuname, wa-kuort.
  ENDSELECT.
ENDFORM.                    "anzeige

*---------------------------------------------------------------------*
*  FORM loeschen
*---------------------------------------------------------------------*
FORM loeschen.
*--- Dynpro-Kommunikationsbereich nach WA  ---*
  MOVE zkunde TO wa.
*--- Löschen des ausgewählten Satzes ----------*
  DELETE FROM zkunde WHERE kunr = wa-kunr.
*---- Fehlerkontrolle -------------------------*
  IF sy-subrc <> 0.
    ROLLBACK WORK.
  ELSE.
    CLEAR: zkunde, wa.
    MESSAGE s005(zhoh).
  ENDIF.
ENDFORM .                    "loeschen
*---------------------------------------------------------------------*
*  FORM speichern
*---------------------------------------------------------------------*
FORM speichern.
*---- Kommuniktionsbereich in WA -----*
  MOVE zkunde TO wa.
  IF neuanlage = 'J'.
*--- Neuanlage eines Satzes ----------*
    INSERT zkunde FROM wa.
  ELSE.
*-- Update von Feldern ----------------*
    UPDATE zkunde FROM wa.
  ENDIF.
ENDFORM.                    "speichern
*&---------------------------------------------------------------------*
*&      Module  lesen  INPUT
*&---------------------------------------------------------------------*
*      Lesen in Abhängigkeit vom eingegebenen Feld kunr
*---------------------------------------------------------------------*
MODULE lesen INPUT.
*--- Eingabefeld für Eingrenzung schieben ---------------*
  MOVE zkunde-kunr  TO wa-kunr.
*---- Lesen in den WA-Satz
  SELECT SINGLE * FROM zkunde INTO wa WHERE kunr = wa-kunr.
*----- Erfolgloses Lesen bedeutet Neuanlage
  IF sy-subrc <> 0.
    MOVE 'J'   TO neuanlage.
    MOVE wa    TO zkunde.
  ELSE.
*---- Erfolgreiches Lesen bedeutet füllen der Maske -----*
    MOVE 'N'  TO neuanlage.
```

```
    MOVE wa    TO zkunde.
  ENDIF.
ENDMODULE.                    " lesen  INPUT
```

3.10.2 Erweiterte Dynproprogrammierung

Inhale des nachfolgenden Beispiels ist:

- Erfassung von Submasken
- Feldaktionen (FIELD dynprofeld MODUL modulname (ON INPUT/ON REQUEST) und eventuell Verknüpfung von Feldaktionen mit (CHAIN……. ENDCHAIN)
- Assistenten

Logik der Submasken:

- Erfassung einer Hauptmaske mit reserviertem Submaskenbereich
- Eindeutige Namensvergabe der Submaske
- Erweiterung der Dynpro-Verarbeitungslogik um
  PBI mit
  CALL SUBSCREEN name_des_subcreens INCLUDING 'programmname_des_hauptprogramms' 'maskennummer_des_subscreens'.
  PAI mit:
  CALL SUBSCREEN SUBTEST.
  - Erfassung der Submaske mit Kennung SUBMASKE

Feldaktionen

- Erfassung eines Datenerfassungsprogramms mit Feldprüfung
  - Ausführen des PAI nach dem man auf dem Dynprofeld gestanden hat und dieses verlässt.
    FIELD dyprofeld MODULE modulname ON INPUT.
  - Aufruf des Moduls, wenn sich der Inhaltswert eines Dynprofeldes verändert.
    FIELD dynprofeld MODULE modulname ON REQUEST.
  - Aufruf des Moduls, wenn sich das Feld verlassen wird
    FIELD dynprofeld MODULE modulname.
- Beispielprogramm Stammdatenpflege für ZWKN01

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

REPORT  ZBEISPIEL_ERW_DYNPRO          .

types: type_satz type zkunde01.
data:  satz      type type_satz.
data:  okcode    like sy-ucomm.

start-of-selection.
```

```
call screen '0100'.
*&---------------------------------------------------------------------*
*&      Module  status_0100  OUTPUT
*&---------------------------------------------------------------------*
*       text
*----------------------------------------------------------------------*
module status_0100 output.
SET PF-STATUS '0100'.
endmodule.                 " status_0100  OUTPUT
*&---------------------------------------------------------------------*
*&      Module  user_command_0100  INPUT
*&---------------------------------------------------------------------*
*       text
*----------------------------------------------------------------------*
module user_command_0100 input.
if okcode = 'BACK'.
   leave to screen 0.
endif.
endmodule.                 " user_command_0100  INPUT
*&---------------------------------------------------------------------*
*&      Module  kulesen  INPUT
*&---------------------------------------------------------------------*
*       text
*----------------------------------------------------------------------*
module kulesen input.
select single * from zkunde01 into satz where kunr = satz-kunr.
if sy-subrc ne 0.
   MESSAGE e000(zss2002) WITH satz-kunr.
*   Der Datensatz '&1' ist nicht vorhanden
endif.
endmodule.                 " kulesen  INPUT
```

Einträge im SCREEN-PAINTER

```
PROCESS BEFORE OUTPUT.

  MODULE status_0100.

*
PROCESS AFTER INPUT.

  MODULE user_command_0100.
  FIELD  satz-kunr MODULE kulesen ON REQUEST.
```

Assistenten

Vorgehensweise Erstellung einer Maske mit Grid.
Erweiterung der Maske um ausprogrammierte Module

```
PROCESS BEFORE OUTPUT.
.............
  MODULE status_0100.

*
PROCESS AFTER INPUT.
.............
  MODULE user_command_0100.
  FIELD  satz-kunr MODULE kulesen ON REQUEST.
```

3.10.3 Dialogprogramm in Verbindung mit Objekten

Das nachfolgende Beispiel zeigt wie die Dialogprogrammierung unter Einsatz von fertigen Dialog-Methoden programmiert wird.

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

REPORT  z_erw_dynpro_mit_objekt                                       .
*---- Globale Variablen ----*
DATA: okcode TYPE sy-ucomm.
*---- Kommunikationsbreich Dynpro ----*
TABLES: spfli.
*--- Klassendefinition
*---------------------------------------------------------------------*
*       CLASS anw DEFINITION
*---------------------------------------------------------------------*
*
*---------------------------------------------------------------------*
CLASS anw DEFINITION.
  PUBLIC SECTION.
    METHODS: constructor,
             lesen_saetze IMPORTING l_carrid TYPE spfli-carrid,
             fuelle_list.
  PROTECTED SECTION.
  PRIVATE SECTION.
*------ Tabelle mit den Ergenbissen aus der Datenbank
    DATA: spfli_tab TYPE TABLE OF spfli,
*--- Referenzvariablen für zwei globale Klassen
          container TYPE REF TO cl_gui_custom_container,
          alv_list  TYPE REF TO cl_gui_alv_grid.

ENDCLASS.                    "anw DEFINITION
*---- Klassenimpementierung
*---------------------------------------------------------------------*
*       CLASS anw DEFINITION
*---------------------------------------------------------------------*
*
*---------------------------------------------------------------------*
CLASS anw IMPLEMENTATION.
  METHOD constructor.
*---- Erzeugen einer Instanz von cl_gui_custom_container ----*
*---- füllen des Mussfeldes container_name
    CREATE OBJECT container
      EXPORTING
        container_name              = 'LIST_AREA'
        .
    IF sy-subrc <> 0.
*   MESSAGE ID SY-MSGID TYPE SY-MSGTY NUMBER SY-MSGNO
*              WITH SY-MSGV1 SY-MSGV2 SY-MSGV3 SY-MSGV4.
    ENDIF.
*---- Aufruf der
    CREATE OBJECT alv_list
```

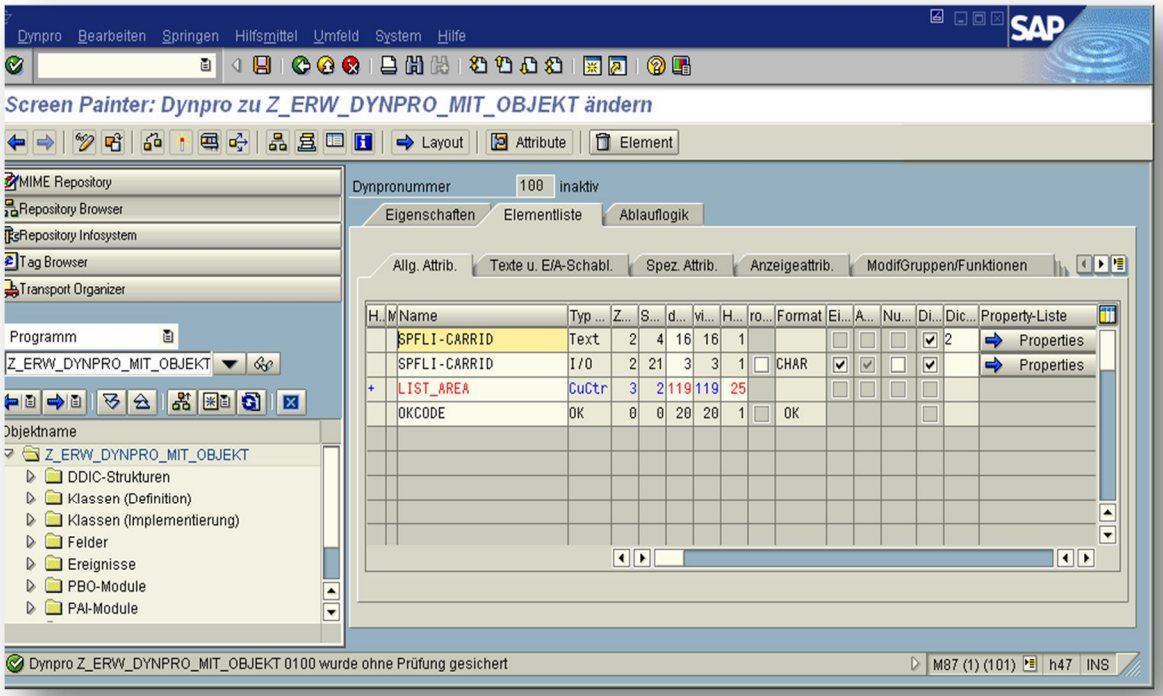
```
      EXPORTING
         i_parent          = container
        .
    IF sy-subrc <> 0.
* MESSAGE ID SY-MSGID TYPE SY-MSGTY NUMBER SY-MSGNO
*            WITH SY-MSGV1 SY-MSGV2 SY-MSGV3 SY-MSGV4.
    ENDIF.

*---- Erzeugen der Grid-Struktur ---------------------------*
    CALL METHOD alv_list->set_table_for_first_display
      EXPORTING
        i_structure_name = 'SPFLI'
      CHANGING
        it_outtab        = spfli_tab.

  ENDMETHOD.                    "constructor
*---- Füllen der Liste
  METHOD fuelle_list.
  call method alv_list->refresh_table_display.
  ENDMETHOD.                    "fill_list

  METHOD lesen_saetze.
    SELECT * FROM spfli INTO TABLE spfli_tab WHERE carrid = l_carrid.
  ENDMETHOD.                    "lesen_saetze
ENDCLASS.                   "anw IMPLEMENTATION
*---- Globale Referenzvariable ----*
DATA objekt_ref TYPE REF TO anw.



*--- Start des Hauptprogramms -------------*
START-OF-SELECTION.
*------- Erzeugen einer Instanz der Klasse o_anwendung ------------*
  CREATE OBJECT objekt ref.
*------- Maskenaufruf ------------*
  CALL SCREEN '100'.
*&---------------------------------------------------------------------*
*&      Module  STATUS 0100  OUTPUT
*&---------------------------------------------------------------------*
*       text
*----------------------------------------------------------------------*
MODULE status 0100 OUTPUT.
*---- Erzeugen einer Statuszeile ------------*
  SET PF-STATUS '100'.
*  SET TITLEBAR 'xxx'.
  CALL METHOD objekt_ref->fuelle_list.

ENDMODULE.                 " STATUS_0100  OUTPUT
*&---------------------------------------------------------------------*
*&      Module  USER_COMMAND_0100  INPUT
*&---------------------------------------------------------------------*
*       text
*----------------------------------------------------------------------*
MODULE user_command_0100 INPUT.

  IF okcode = 'BACK' OR
     okcode = 'EXIT' OR
     okcode = 'CANCEL'.
    LEAVE PROGRAM.
  ELSE.
```

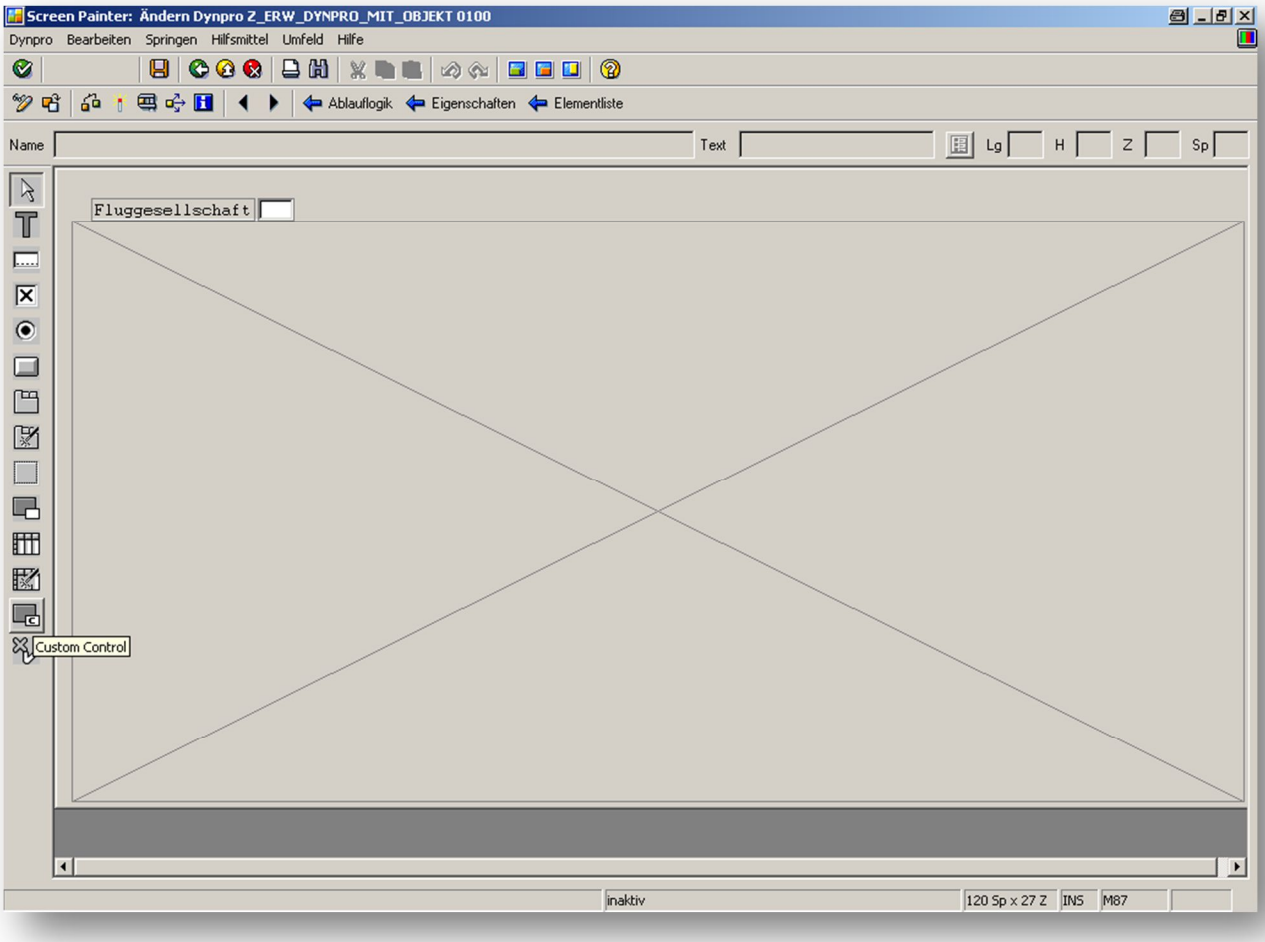
```
*--- Aufruf der Methode lesen_saetze und Ausgrenzung durch Maskeneingabe
    CALL METHOD objekt_ref->lesen_saetze
      EXPORTING
        l_carrid = spfli-carrid.
  ENDIF.

ENDMODULE.                    " USER_COMMAND_0100  INPUT
```

DYNPRO-Steuerung

```
ROCESS BEFORE OUTPUT.
 MODULE STATUS_0100.
*
PROCESS AFTER INPUT.
 MODULE USER_COMMAND_0100.
```

DYNPRO-ELementliste





3.11 Logische Datenbanken

Mit logischen Datenbank werden auf Datenbanktabellen zugegriffen. Das folgende Beispiel zeigt zunächst eine Lösung der Tabellenanzeige spfli und spflight mit Hilfe von SELECT und SELECT-OPTIONS.

```
REPORT  zblatt4_beispiel_ohne_logdb    .


TABLES: spfli, sflight.

SELECT-OPTIONS:
        flugges FOR spfli-carrid,
        flugnr  FOR spfli-connid,
        flugdat FOR sflight-fldate,
        zielort FOR spfli-cityto.

START-OF-SELECTION.
  SELECT * FROM spfli WHERE carrid IN flugges
```

```
                    AND   connid IN flugnr
                    AND   cityto IN zielort.

    WRITE: / spfli-carrid, spfli-connid, spfli-cityfrom, spfli-cityto.
    SELECT * FROM sflight WHERE carrid = spfli-carrid
                          AND   connid = spfli-connid
                          AND   fldate IN flugdat.
      WRITE: /5 sflight-fldate, sflight-seatsocc.
    ENDSELECT.
  ENDSELECT.
  IF sy-subrc NE 0.
    WRITE: / 'Keine Daten gefunden'.
    MESSAGE i001(15).
*   V1-Verbuchung abgeschlossen

  ENDIF.
```

Alternative kann die Tabellenanzeige für spfli und spflight mit Hilfe von Logischen Datenbanken erfolgen, wie da nachfolgende Beispiel zeigt.

```
*&---------------------------------------------------------------------*
*& Report  zblatt4_beispiel_logische_db                                *
*&                                                                     *
*&---------------------------------------------------------------------*

REPORT  zblatt4_beispiel_logische_db  .

TABLES: spfli, sflight, sbook.
SELECT-OPTIONS:
zielort FOR spfli-cityto.

START-OF-SELECTION.

GET spfli.
  WRITE: / spfli-carrid, spfli-connid, spfli-cityfrom, spfli-cityto.

GET sflight.
  write: /5 sflight-fldate, sflight-seatsocc.
```

ACHTUNG: Bei der Anlage des Programms muss die logische Datenbank F1S angegeben werden. Hier ist eine Hierarchie SPFLI, SFLIGHT und SBOOK

```
*&---------------------------------------------------------------------*
*& Report  zblatt4_beispiel_logische_db
*
*&
*
*&---------------------------------------------------------------------*

REPORT  zblatt4_beispiel_logische_db  .

TABLES: spfli, sflight, sbook.
SELECT-OPTIONS:
zielort FOR spfli-cityto.

START-OF-SELECTION.

GET spfli.
```

```
      WRITE: / spfli-carrid, spfli-connid, spfli-cityfrom, spfli-cityto.

   GET sflight.
     write: /5 sflight-fldate, sflight-seatsocc.
```

Im nachfolgenden Beispiel wird die Ausgabe noch erweitert.

```
REPORT  z_logische_db01.
DATA: anzahl TYPE i,
      gewicht type i.
*-- NODES hat die gleiche Funktion wie TABLES
NODES: spfli, sflight, sbook.
* oder
* TABLES: spfli, sflight, sbook.
*---- Manuelle Ausgrenzung ------------------*
*SELECT-OPTIONS:   zielort1 FOR spfli-cityto.
*PARAMETERS:       zielart3 TYPE spfli-cityto.

START-OF-SELECTION.

*-- Hierarchie
*--- SPLFI ---> SPFLIGHT ---> SBOOK
*--- LATE bedeutet
*--- SPLFLI<---- SPLFIGHT <---- SBOOK

*----- Auslesereignis aus einer logischen Datenbank für SPLFI----------*
GET spfli.
*    field spfli-carrid, spfli-connid, spfli-cityfrom, spfli-cityto

  WRITE: /, 'Fluggesellschaft', spfli-carrid, 'Flugnr.', spfli-connid,
'von', spfli-cityfrom, 'nach', spfli-cityto.

*---- Nach Ausgabe der SFLIGHT und SBOOCK --> Gruppensumme ----*
 GET spfli LATE .
   write: 'Gesamtanzahl Sitze: ', anzahl.
   WRITE: 'Ende des Flugs'.
   move 0  to anzahl.
   uline.
*----- Auflistung der sfilght-Flüge-----*
 GET sflight.
   WRITE: /5 'Flugnr.', sflight-connid, 'Tag', sflight-fldate,
   'Anzahl Sitze', sflight-seatsocc.
   compute anzahl = anzahl + sflight-seatsocc.

*--- Nach Ausgabe der SBOOK--> Gruppensumme
GET sflight LATE FIELDS carrid.
    WRITE:   'Gesamtgewicht: ', gewicht.
    move 0 to gewicht.
*---- Summation der Gewicht ---------------*
GET sbook FIELDS luggweight.
  gewicht = gewicht + sbook-luggweight.

END-OF-SELECTION.
  WRITE: 'Ende der Ausgabe'.
```

3.12 Objekte

Das nachfolgende Beispiel zeigt die Programmstruktur für die objektorientierte Programmierung d.h. lokale Klassendefinition, lokale Klassenimplementierung und Instanzierung im Hauptprogramm.

```
REPORT  z_objekte                                .

TYPES: t_zahl(12) TYPE p DECIMALS 2.

PARAMETERS: wa_kurs TYPE t_zahl,
      wa_divi TYPE t_zahl.

DATA: wa_kgv TYPE t_zahl.

*----- Klassendefinition-----*
CLASS berechnung DEFINITION.
 PUBLIC SECTION.
  METHODS: kursgewinn IMPORTING l_kurs TYPE t_zahl
                    l_divi TYPE t_zahl
            EXPORTING l_kgv  TYPE t_zahl.
 PROTECTED SECTION.
 PRIVATE SECTION.
ENDCLASS.                    "berechnung DEFINITION

*---------------------------------------------------------------------*
*       CLASS berechnung IMPLEMENTATION
*---------------------------------------------------------------------*
*
*---------------------------------------------------------------------*
CLASS berechnung IMPLEMENTATION.
 METHOD kursgewinn.
  COMPUTE  l_kgv = l_kurs / l_divi.
 ENDMETHOD.                    "kursgewinn

ENDCLASS.                    "berechnung IMPLEMENTATION
*----- Referenzvariable --------------------*
DATA: ref_berechnung TYPE REF TO berechnung.

START-OF-SELECTION.
 CREATE OBJECT ref_berechnung.

 CALL METHOD ref_berechnung->kursgewinn
  EXPORTING
   l_kurs = wa_kurs
   l_divi = wa_divi
  IMPORTING
   l_kgv  = wa_kgv.
```

```
WRITE: / 'Ergebnis',    wa_kgv.
```

3.13 Perform/Form

Mit Perform können Unterprogramme aufgerufen werden. Diese Unterprogramme erlauben die Übergabe von Variablen als Referenz oder als Wert. Die Kombinationen und entsprechende Beispiele im folgenden.

Defintion	USING	USING VALUE	CHANGING	CHANGING VALUE
Übergabeart	Referenzübergabe	Wertübergabe	Referenzübergabe	Wertübergabe
Wirkung auf den Aktualparameter	JA	NEIN	JA	JA (ENDFORM, EXIT, CHECK) bzw. NEIN

```
REPORT  z01_form                                                .
TYPES: typ(10)  TYPE p DECIMALS 2,
       typ1(3)  TYPE p DECIMALS 1.
*---- Ergebnis ----------------------*
DATA:       e TYPE typ.
*----- Eingabeparameter -------------*
PARAMETERS: a type typ,
            b type typ.

START-OF-SELECTION.
*------- USING (Referenzübergabe, Änderungs des Aktualparameters JA---*
  MOVE: 0  TO e.
  PERFORM u_rech USING a b e.
  WRITE: / 'Fall1: using', a, b, e.
*------ USING VALUE (Wertübergabe, Änderung des Aktualparameters Nein -*
  MOVE: 0  TO e.
  PERFORM v_rech USING a b e.
  WRITE: / 'Fall2: using value', a, b, e.
*--- Lokale Variable kann nicht ausgegeben werden
*  write: / c.

*---- CHANGING (Referenzübergabe, Änderung des Aktualparameters JA -*
*---- (Unterscheidung CHANGING/USING nur Dokumentationscharakter ---*
*----- CHANGING Ohne Änderung, USING mit Änderung ------------------*
  MOVE: 0  TO e.
  PERFORM c_rech USING a b e.
  WRITE: / 'Fall3: using value', a, b, e.
*---- CHANGING VALUE (Wertübergabe, Änderung des Aktualparameters JA
*---- bei EDNFORM, CHECK oder EXIT --------------------------------*
*---- (Unterscheidung CHANGING/USING nur Dokumentationscharakter ---*
*----- CHANGING Ohne Änderung, USING mit Änderung ------------------*
   MOVE: 0  TO e.
  PERFORM c_v_rech CHANGING a b e.
  WRITE: / 'Fall4: changing value', a, b, e.


*---------------------------------------------------------------------*
*------- USING (Referenzübergabe, Änderungs des Aktualparameters JA---*
*---------------------------------------------------------------------*
```

```
FORM u_rech USING a
*------ Dies wäre inkompatibel (a TYPE typ1)
                  b
                  e.
  COMPUTE e = a + b.

ENDFORM.                    "u_v_rech
*---------------------------------------------------------------------*
*------ USING VALUE (Wertübergabe, Änderung des Aktualparameters Nein -*
*---------------------------------------------------------------------*
FORM v_rech USING value(a) value(b) value(e).
*---- lokale Variable -----------*
  DATA: c TYPE typ VALUE 100.
  COMPUTE e = a + b + c.
  WRITE: / 'Ausgabe: using value im Form', a, b, e.

ENDFORM.                    "u_rech

*---------------------------------------------------------------------*
*----- CHANGING (Referenzübergaben, Änderung des Aktualparameters Ja--*
*---------------------------------------------------------------------*
FORM c_rech CHANGING a b e.
  COMPUTE e = a + b.
ENDFORM.                    "u_v_rech

*---------------------------------------------------------------------*
*---- CHANGING VALUE (Wertübergabe, Änderung des Aktualparameters JA
*---- bei ENDFORM, CHECK oder EXIT ----------------------------------*
*---- (Unterscheidung CHANGING/USING nur Dokumentationscharakter -----*
*----- CHANGING Ohne Änderung, USING mit Änderung ------------------*
*------ACHTUNG: Funktion hat nicht die gewünscht Arbeitsweise---------*
FORM c_v_rech CHANGING value(a) value(b) value(c).
  COMPUTE e = a + b.
  WRITE: / 'Ausgabe: changing value im Form', a, b, e.

ENDFORM.                    "c_v_rech
```

Das gleiche Problem mit einer Klasse realisiert:

```
REPORT  z01_objekt                              .

TYPES: zahl TYPE p DECIMALS 2.
*---- Eingabeparameter Hauptprogramm -----------*
PARAMETERS: f_a TYPE zahl, f_b TYPE zahl.
*----- Ergebnisfeld im Hauptprogramm
DATA:       f_e TYPE zahl.

*---------------------------------------------------------------------*
*       CLASS rechnung  DEFINTION
*---------------------------------------------------------------------*
CLASS rechnung DEFINITION.
 PUBLIC SECTION.
  METHODS: addition IMPORTING c_a TYPE zahl
                              c_b TYPE zahl
                    EXPORTING c_e TYPE zahl.
 PROTECTED SECTION.
 PRIVATE SECTION.
```

```
ENDCLASS.                    "rechnung  DEFINTION

*----- Referenzvariable für die Klasse ------------------------------*
*----- ACHTUNG: Referenzvariable nach der Methodendefinition --------*
DATA: ref_rechnen TYPE REF TO rechnung.

*---------------------------------------------------------------------*
*       CLASS berechnung IMPLEMENTATION
*---------------------------------------------------------------------*
CLASS rechnung IMPLEMENTATION.
 METHOD: addition.
  COMPUTE c_e = c_a + c_b.
 ENDMETHOD.                     "rechnung
ENDCLASS.                    "rechnung IMPLEMENTATION

START-OF-SELECTION.
*--- Instanzierung -----------------*
 CREATE OBJECT ref_rechnen.
*--- Aufruf der Methode ------------*
 CALL METHOD ref_rechnen->addition
  EXPORTING
   c_a = f_a
   c_b = f_b
  IMPORTING
   c_e = f_e.

 WRITE: / 'Ergebnis', f_e.
```

Das gleiche Problem mit einem Funktionsbaustein realisiert:

```
REPORT  z01_funktionsbaustein                                   .

*----- Typisierung von za im DataDictonary als Zahl --------------*
*----- als dec(10) decimals 2.
DATA: f_e TYPE za.
*----- Eingabeparameter-------------------------------------------*
PARAMETERS: f_a TYPE za,
            f_b TYPE za.

START-OF-SELECTION.
*----- Aufruf eines UnterProgramms -------------------*
  CALL FUNCTION 'ADDITION'
    EXPORTING
      a      = f_a
      b      = f_b
    IMPORTING
      e      = f_e
    EXCEPTIONS
      fehler = 1
      OTHERS = 2.

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

```
  WRITE: 'Ergebnis', f_e.
```

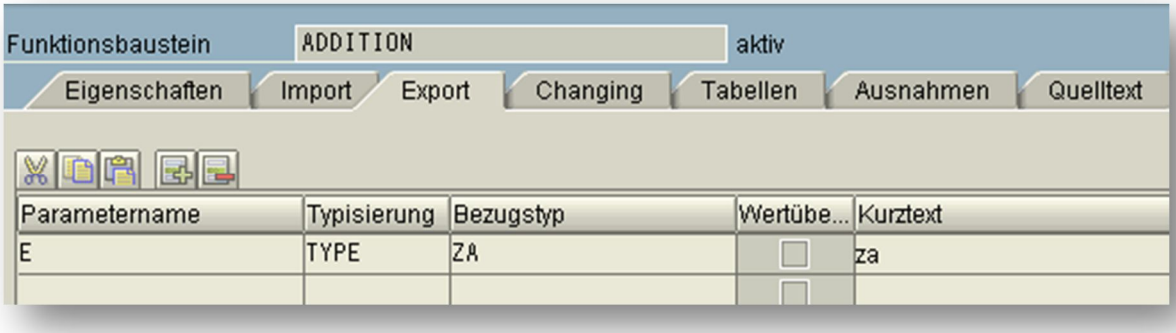
Funktionsbaustein

a) Quelltext

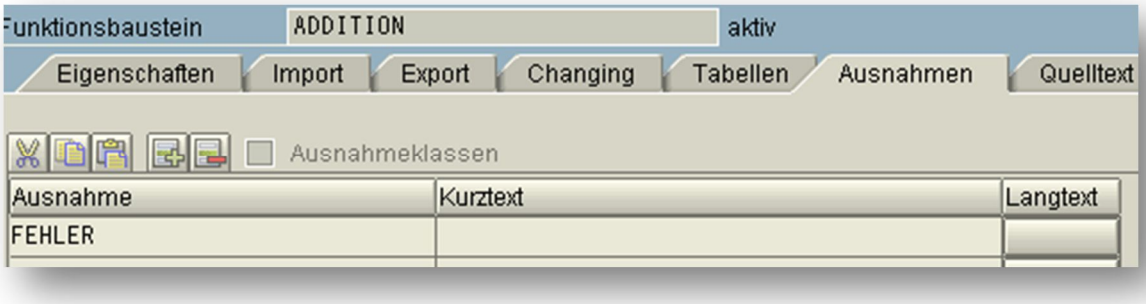
```
FUNCTION ADDITION.
*"----------------------------------------------------------------------
*"*"Lokale Schnittstelle:
*"  IMPORTING
*"     REFERENCE(A) TYPE  ZA OPTIONAL
*"     REFERENCE(B) TYPE  ZA OPTIONAL
*"  EXPORTING
*"     REFERENCE(E) TYPE  ZA
*"  EXCEPTIONS
*"      FEHLER
*"----------------------------------------------------------------------

compute e = a + b.
*------------ Fehler-Fangen ------------*
if sy-subrc <> 0.
   raise fehler.
endif.


ENDFUNCTION.
```

a) **Importing**

c) **Exporting**



d) **Ausnahmen**



3.14 Mini-SAP-System 4.6/6.2

Das Mini-SAP-System ist als Client-Server-System zu installieren. Einerseits gibt es ein Xserver für die Datenbank (ADABAS), einen SAP-Server und das GUI.

Das Mini-SAP-Sytem ist ein Evaluationssystem das vier Wochen zum Test zur Verfügung steht. TIPP: Stellen Sie das Datum vor (z.B. 01.01.2020) und Installieren Sie dann. Das System erhalten Sie kostenlos bei SAP oder ist verschiedenen Büchern beigelegt. Nach vier Wochen muss die Testlizenz verlängert werden. Die Installation ist einfach und doch kann es zu schwer nachvollziebaren Problemen kommen. Die Installation kann bis zu 4 Std. dauern. Lesen Sie in jedem Fall die Installationshinweise (Web_as_demo.htm) aufmerksam durch. Der hierzu notwendige Zeitbedarf zahlt sich für Sie mehrfach aus. Für die Installation kann es ratsam sein, die CDs auf Platte zu legen und dann zu installieren. Vor der Installation sollten Sie in jedem Fall den MS-Loopback Adapter installieren und für eine statische IP zu konfigurieren z.B. 192.168.17.3/255.255.255.0 . Sie finden den Loopback Adapter unter „Systemsteuerung / Hardware / Neue Hardware hinzufügen / manuell / Netzwerkadapter / Microsoft / Microsoft Loopbackadapter „. In der Hosts muss der Eintrag auf den Loopback-Adapter erfolgen C:\Windows\system32\drivers\etc\hosts z.B. „192.168.17.3 mein PC“. Mit ping „ping meinPC“ könne Sie testen. Die Datei C:\Windows\system\32\drivers\etc\services (Windows XP) darf keinen Eintrag für Port 3600 enthalten (ggfs. Auskommentieren), da nach der Installation folgende Einträge vorhanden sind. sapdp00 3200/tcp, sapmsBSP 3600/tcp, sapgw00 3300/tcp. Durch die Möglichkeit der nicht korrekten Installation sollten Sie Ihr System vor der Installation mit einem Wiederherstellungspunkt versehen; Sicher ist Sicher. Der Wiederstellungspunkt erstellen sind unter: „Start / Programme / Zubehör / Systemprogramme / Systemwiederherstellung / (*) Einen Wiederherstellungspunkt“ erstellen. Jetzt kann installiert werden. Treten Probleme auf muß ggf. vor der deinstallation der Dienst Xserver angehalten werden (Systemsteuerung / Verwaltung / Dienste / Beenden), der Rücksetzpunkt aktiviert oder alle Einträge SAP aus der Registry gelöscht werden. Ist der Server erfolgreich installiert kann das GUI installieren werden „F:\SAP Installations CD 1\MINIGUI\setup.exe“. Für die Kommunkation mit der Server (dieser muss gestartet sein) geben Sie ein Konfigurationsprofil mit folgenden Informationen ein. Bezeichnung: MINI-SAP; Anwendungsserver: meinPC; SAP-System: (*) R3; Systemnummer: 00. Der Login erfolgt mit Client: 000, User: bcuser und Pass: minisap.

Hilfe und Flugdatenmodell müssen separat installiert werden. Die Hilfe finden Sie auf der ersten CD unter (L:\DOCU\HTMLHELP\HELPDATA\EN\EN.zip) und direkt in SAP mit

der Transaktion "SR13" (siehe "CD1:/docu/extdoc.htm). Dann funktioniert auch "Hilfe / Hilfe zur Anwendung". ABAP-Beispiele gibt es ebenfalls. Die Beispiele sind klein, meist verständlich und ausführbar. Zu finden im Menü unter "Umfeld / Beispiele / APAB-Beispiele".Das Flugmodell, welches in der Vorlesung verwendet wird ist nach der Installation zunächst leer. Es gibt allerdings ein Programm FLIGHT_MODEL_DATA_GENERATOR mit dem die Beispieldaten eingespielt werden können. Kopieren Sie mit Copy+Paste den Quellcode aus dem ProgrammFLIGHT_MODEL_DATA_GENERATOR, in ein eigenes z.B. ZDATEN. Speichern Sie das Programm, Aktualiseren und Ausführen. Dann sollten die Beispieldaten vorhanden sein.

Im Standardmandanten funktioniert z.B. der DataBrowser nicht. Um diese Problem zu lösen können Sie die Transaktion SCC4 ausführen und somit den DataBrowser für die Mandanten 000 aktivieren. Eine andere Möglichkeit ist die Anlage von eigenen Benutzern und einem Eignen Mandanten. Hierzu folgende Vorgehensweise:

1. Anmelden an Mandant 000
    - user: sap*
    - pass: pass
    - lang: de
2. Mandanten anlegen
    - Transaktion SCC4 aufrufen
    - Auf Bleistift-Symbol klicken
    - Schalftläche "Neue Einträge" klicken
    - Mandant: 101 ABAP Kurs
    - Ort: Friedberg
    - Logisches System: keine Eingabe
    - Rolle des Mandanten: Training/Education
    - Sonstige Defaults beibehalten und Sichern (Strg+S)
    - Abmelden
3. Mandantenkopie erstellen
    - Anmelden an Mandant 101
    - user: sap*
    - pass: pass
    - lang: de
    - Transaktion SCCL aufrufen
    - Wählen Sie in der Maske aus:
        - Zielmandant: 101
        - Selektiertes Profil: SAP_CUST (geht, obwohl ausgegraut!)
        - Quellmandant 000
    - "Sofort starten" anklicken
    - "Fortfahren" klicken
    - Es dauert etwas, aber es sollten keine Fehler auftreten.
4. Benutzer anlegen
    - Transaktion SU01 aufrufen
    - Für den Benutzer: nachname, vorname, funktion -->pflegen

- "Anlegen (F8)" anklicken
- Exkurs: Adresspflege "Firmenname erfassen". Folgen Sie den Dialogen !
  - Beliebigen Firmenname eingeben: Meine Firma
  - Land: DE
  - Zeitzone: CET
  - Häckchen anklicken; Jetzt kann der Benutzer weiter gepflegt werden
- Reiter "Adresse"
  - Nachname und ggfs. weitere Felder ausfüllen
- Reiter "Logondaten"
  - Initialkennwort und Kennwortwiederholung ausfüllen, z.B. "init"
- Reiter "Festwerte"
  - Anmeldesprache "de"
- Reiter "Profile" folgende Profile eintragen:
  - SAP_ALL
  - S_A.SYSTEM

Speichern

- Abmelden

5. Als Benutzer anmelden
   - user: nachname
   - pass: init
   - lang: de
   - Neues Passwort vergeben. Fertig.

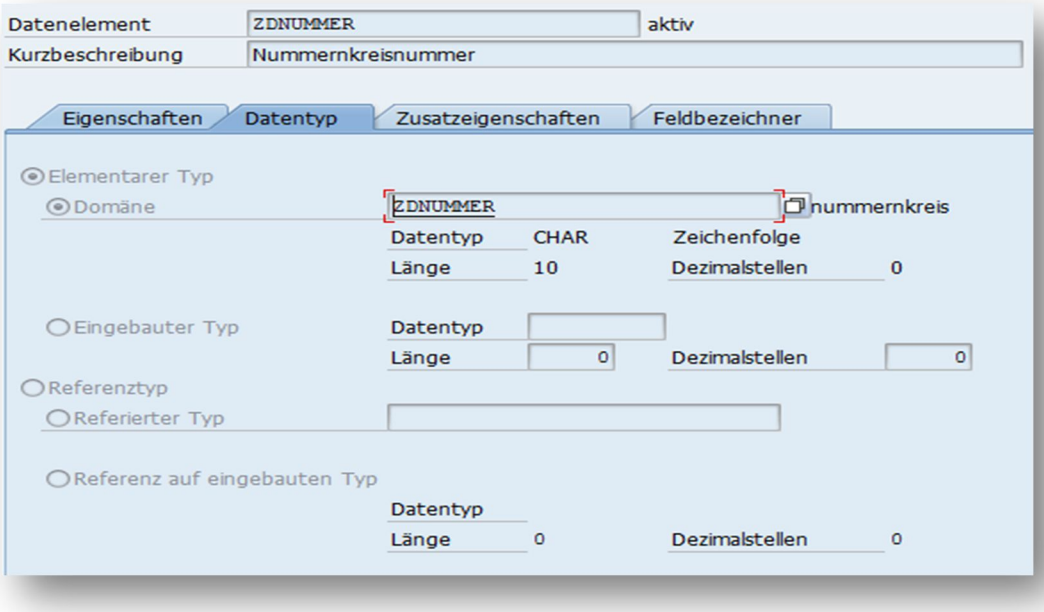
Zur Entwicklung fehlt allerdings der Developschlüssel aus dem OSS von SAP.

3.15 Nummernkreissystem in SAP

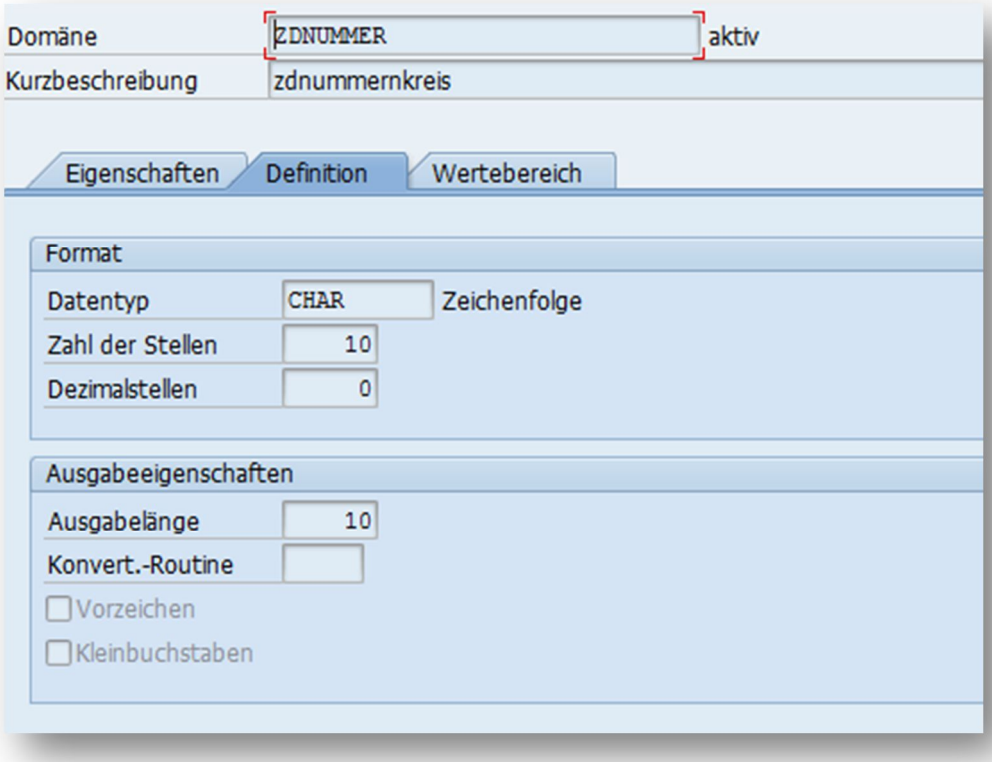
Das SAP-System verfügt über ein Nummernkreissystem, welches es erlaubt, fortlaufende Nummern für einen Primärschlüssel zu definieren und in ABAP-Programmen abzurufen. Die Hinterlegung der Nummernkreise erfolgt zentral in der Tabelle NRIV. Zur Pflege dieser Tabelle gibt es die Transaktion SRNO. Nummernkreise haben einen Namen und können innerhalb des Namens nochmals nach Nummernunterkreisen unterschieden werden.

Die Verwendung von Nummernkreisen und deren Verwendungen können der nachfolgenden Darstellungen entnommen werden:

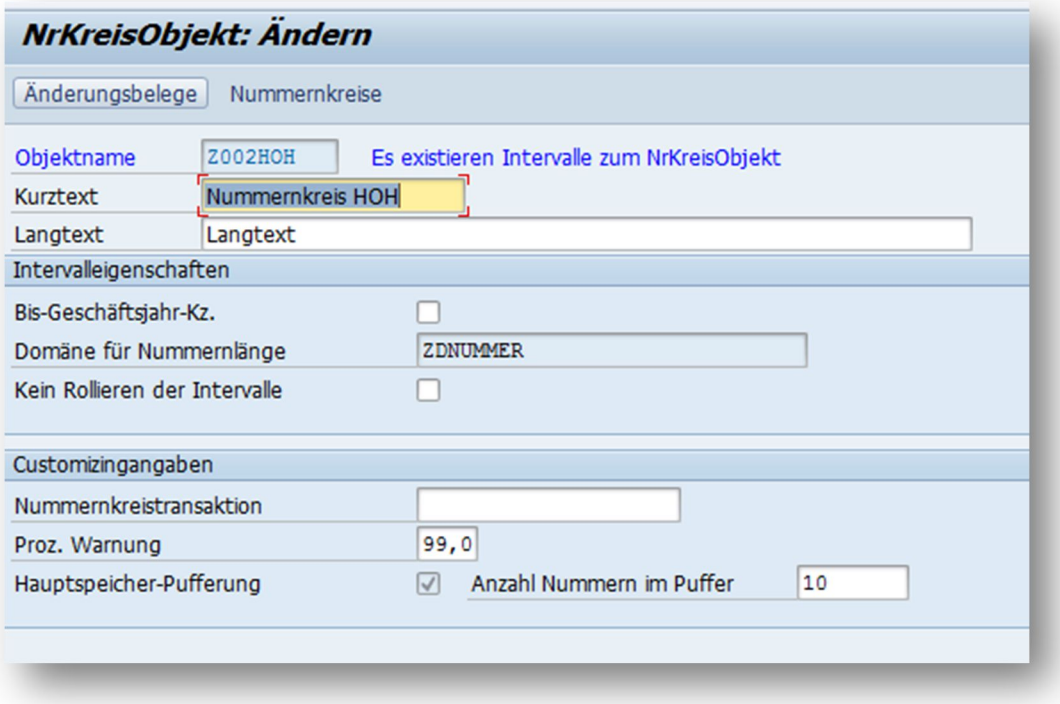
Schritt 1: Anlage eines Datenelement mit Domäne über SE11. Das Datenelement und Domäne dient später der Aufnahme der Nummer und Kommunikation mit dem Funktionsbaustein number_get_next. Das Datenelement hat die technische Eigenschaften CHAR (10).



Schritt 2: Definition der Domäne mit CHAR (10) für die Spezifikation des Datenelements in SE11. In dem vorliegenden Beispiel sind Datenelement und Domäne namensgleich (ZDNUMMER). Dies ist allerdings nicht erforderlich.

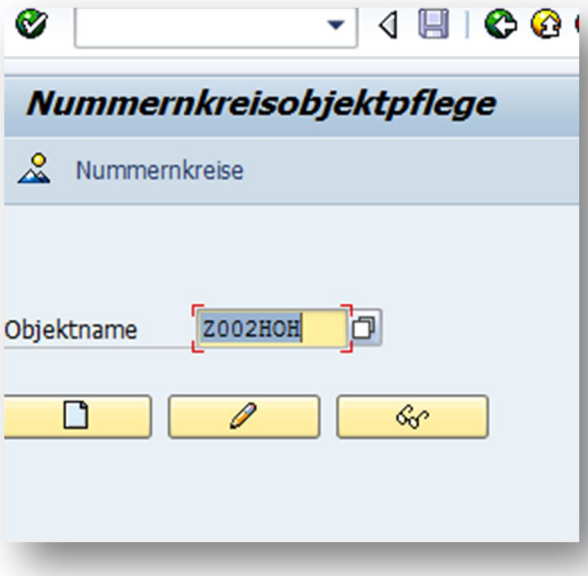


Schritt 3: Mit der Transaktion SNRO kann der Nummernkreis angelegt werden. Es gibt jeweils einen Nummernkreisname – hier im Beispiel Z002HOH und es können über



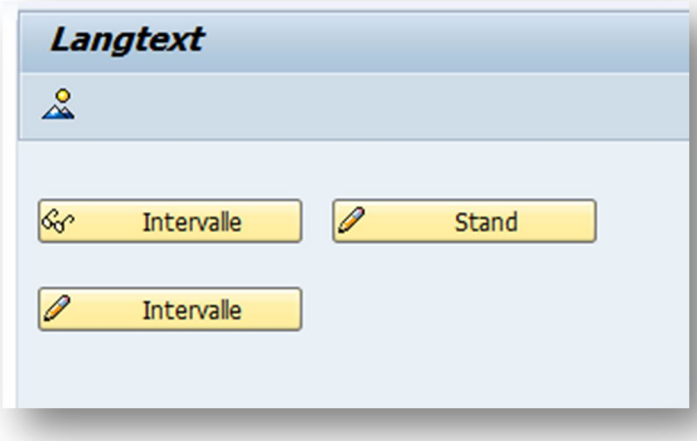
Satzarten Unternummernkreise angelegt werden. Im hier vorliegenden Fall wird die Satzart 01 (Nr.) verwendet.

- Anlage der Nummernkreisnummer mit der Transaktion SNRO

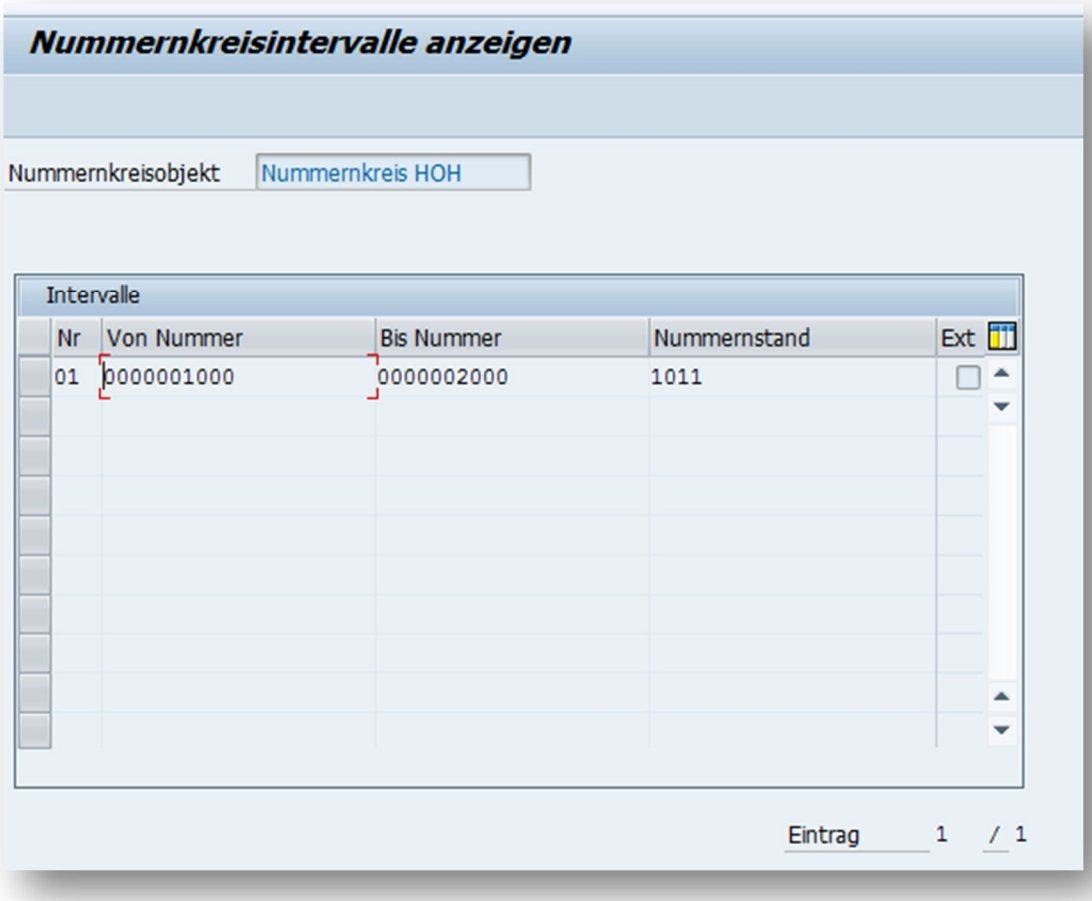


Nummernkreis Grundeinstellungen

- **Einstellung der Nummernkreise bezüglich der Intervalle**



- **Nummernkreiseinstellungen z.B. Satzart 01 und Intervall**



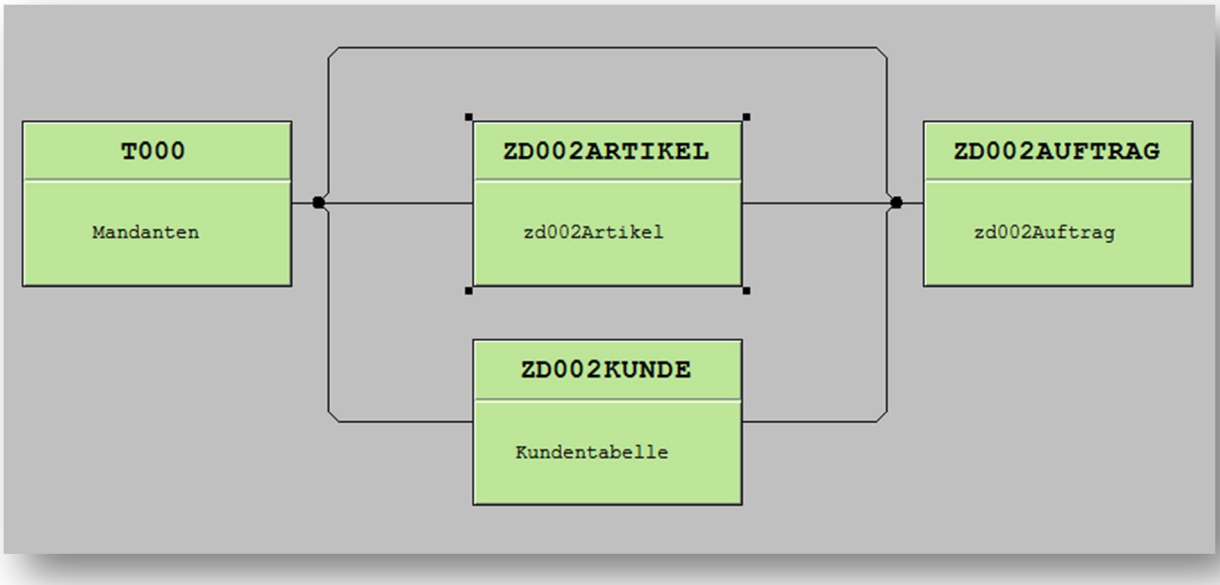
Schritt 4: Verwendung von Nummernkreisen im ABAP-Programm erfolgt über den Funktionsbaustein number_get_next. Mit Hilfe der Übergabeparameter mit dem Nummernkreisname und der Satznummer kann der angelegt Nummernkreis angesprochen werden. Rückgabewert des Funktionsbausteins ist die nächst höhere Nummer. Im nachfolgenden ABAP-Beispiel wird die Nutzung des zuvor angelegten Nummernkreises Z002HOH gezeigt.

Beispielprogramm AUFRUF Nummernkreis

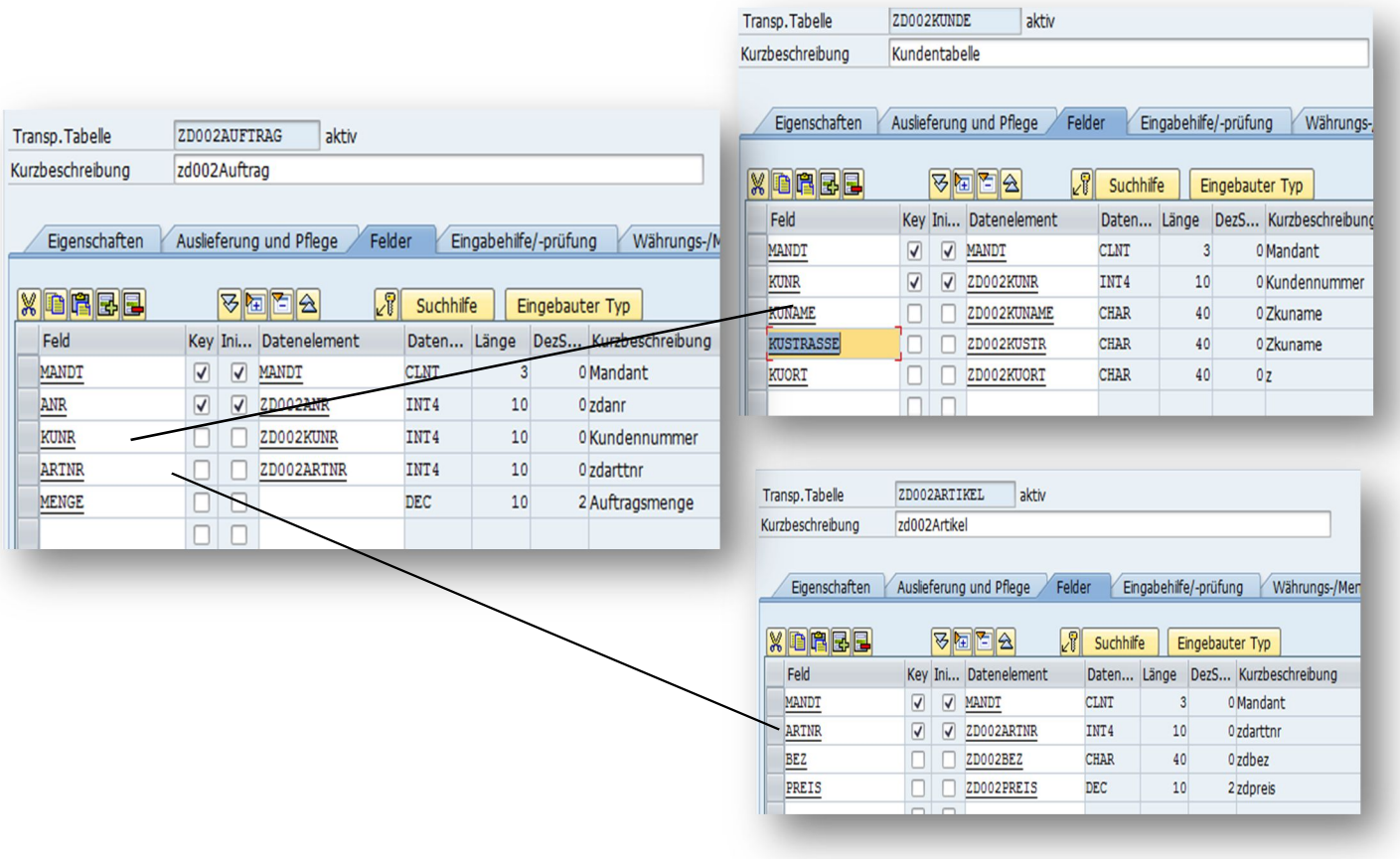
```
*&---------------------------------------------------------------------*
*& Report ZDEV002_NUMMERNKREIS

REPORT zdev002_nummernkreis.
--- datenelement zdnummer und domäne zdnummer
--- 10 stellen char
data: nummer type  zdnummer.

--- übergabe an den funktionsbaustein, typisiert
--- aus nummernkreistabelle → zentrale sap entwicklung
data:  ueb_nr type nrnr value '01',
 ueb_obj type nrobj value 'Z002HOH'.
--- tabelle der zentralen nummernkreise
data: wa type nriv.

START-OF-SELECTION.
*--
- Aufruf der nächsten Nummer aus dem Nummernkreis Z002hoh, Satzart 01 -
*-- Einbindung mit Muster
  CALL FUNCTION 'NUMBER_GET_NEXT'
    EXPORTING
      nr_range_nr             = ueb_nr
      object                  = ueb_obj
* QUANTITY='1'
*SUBOBJECT=' '
*TOYEAR='0000'
*IGNORE_BUFFER = ' '
    IMPORTING
      number                  = nummer
*QUANTITY=
*RETURNCODE=
    EXCEPTIONS
      interval_not_found      = 1
      number_range_not_intern = 2
      object_not_found        = 3
      quantity_is_0           = 4
      quantity_is_not_1       = 5
      interval_overflow       = 6
      buffer_overflow         = 7
      OTHERS                  = 8.
  IF sy-subrc <> 0.
    MESSAGE ID sy-msgid TYPE sy-msgty NUMBER sy-msgno
    WITH sy-msgv1 sy-msgv2 sy-msgv3 sy-msgv4.
  ENDIF.
*--- Ausgabe der nächsten Nummer
  WRITE: nummer.
```

3.16 Suchmethoden, Views und Suchhilfe

Die erfolgreiche Anlage von Datenbanktabellen sind Voraussetzung für die Definition von Suchmethoden auf die Datenfelder. Suchmethoden können nur für Attribute einer Tabelle definiert werden, die über Datenelemente technisch klassifiziert wurden. Mit den Datenelemente werden die Suchmethode verbunden. Datenfelder die in einer Suchfunktion angezeigt werden sollen, müssen ebenfalls über Datenelemente definiert werden. Für eine Suchhilfe ist es gleichgültig, ob eine Tabelle oder ein View verwendet wird. Als Beispiel für die Definition von Suchhilfen wird folgendes Datenmodell herangezogen:



1. Suchhilfe über Datentelemente: Die Attribute der Tabellen sind für alle in den Suchhilfen angezeigten Attribute über Datenelemente definiert.

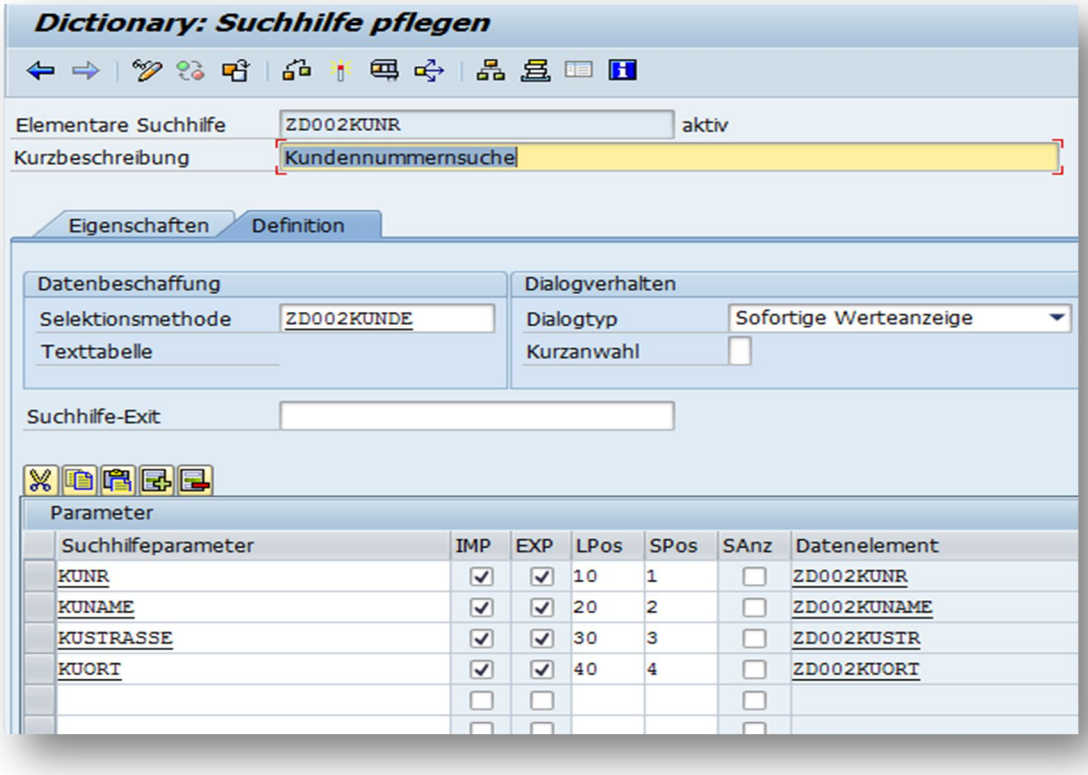


Vorgehensweise am Beispiel: Für die Primärschlüssel der Datenbanktabelle Kunden wird mit Hilfe des Datenelements ZD002KUNR, welches den Primärschlüssel technisch definiert, eine Suchhilfen über das Datenelement ZD002KUNR angelegt. Dies erfolgt in der Tablasche „Zusatzeigenschaften“ des Datenelements.

Durch Doppelklick auf den Namen der Suchhilfe kann die Suchhilfe erstellt oder falls bereits vorhanden geändert werden.

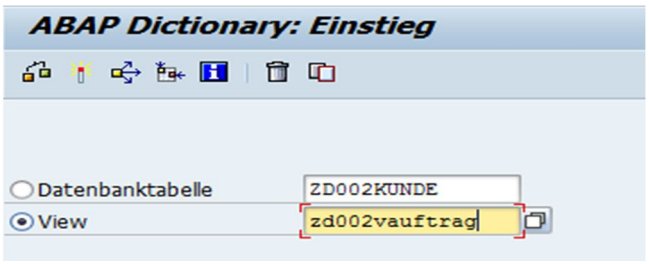
Nun öffnet sich die Suchhilfe pflege. Im Feld Selektionsmethode wird entweder ein View oder eine Tabelle angegeben. Über den Dialogtyp kann festgelegt werden, in welcher Art

und Weise sich die Suchhilfe verhält z.B. Direktanzeige oder mit vorheriger Ausgrenzungsanzeige. Die Definition der Suchhilfeparameter beziehen sich auf die Attributsbezeichnungen in Views oder Tabellen. Für alle anzuzeigenden Attributen müssen Datenelemente definiert sein. Mit der „Lpos“ gibt man die Position in Suchfenster an. Der Wert darf sich nicht wiederholen. „Spos“ gibt die Reihenfolge im Fenster an. „SPos“ 0 bedeutet keine Anzeige. „IMP“ und „EXP“ definieren, ob die Suchfelder aus dem aufrufenden Programm gefüllt werden und ob die gefundenen Werte an die aufrufenden Maske zurückgegeben werden.

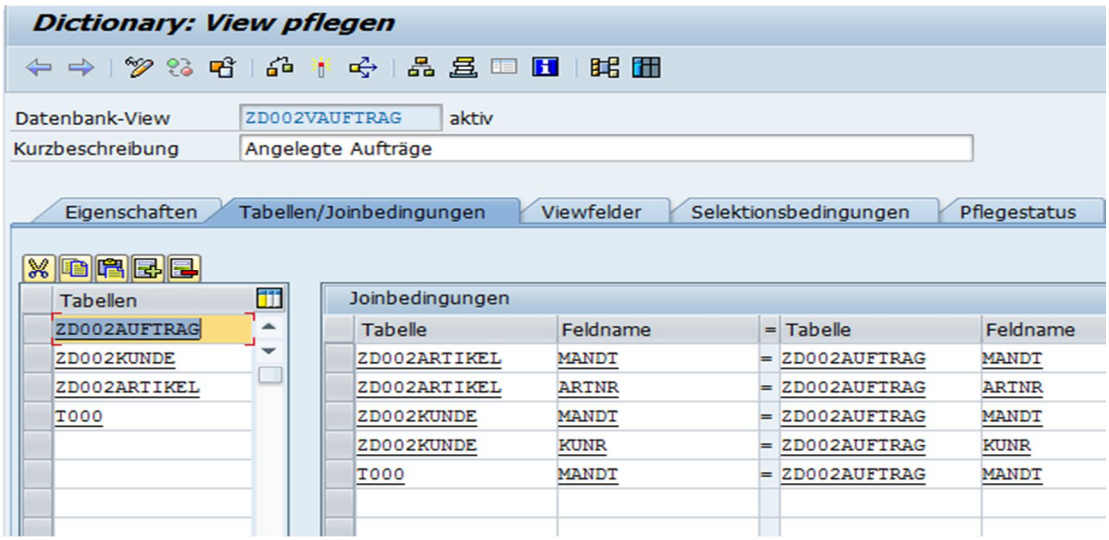


Nach erfolgreicher Aktivierung stehen damit die Suche für den Primärschlüssel zur Verfügung.

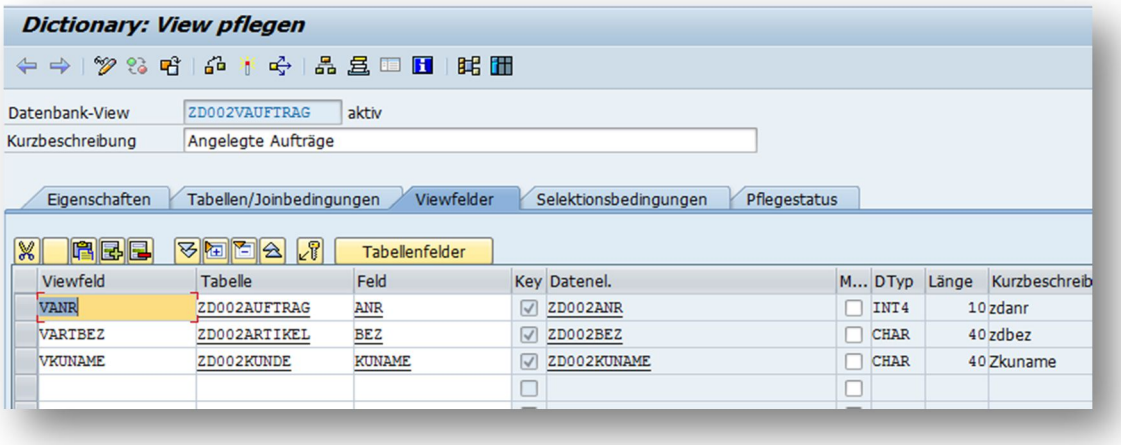
2. Suchhilfe über View: Im DD können auch View definiert werden, über die eine Datensatzsuche gestaltet werden kann. Im vorausgehenden Fall ist es für die Suche über die Aufträge angezeigt, auch Kundenname und Artikelbezeichnung anzugeben. Daher wird im DD eine View wie folgt festgelegt:



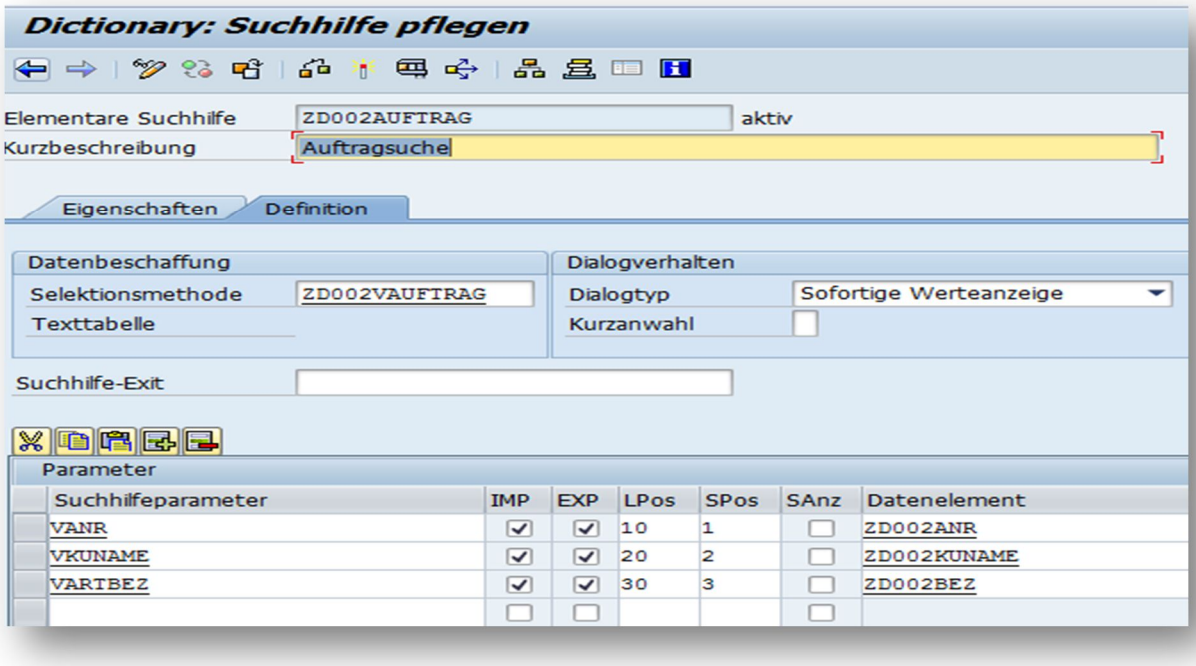
- **Ausgrenzung der Joinbedingungen.** Im Fall des vorliegenden Datenmodells (Kunde, Artikel, Auftrag) ist es für die Vermeidung eines kartesischen Produktes notwendig, die Kundenummer und die Artikelnummer im Auftrag mit dem Primärschlüsseln aus Artikel und Kunde auszugrenzen. Dies erfolgt beim Join im DD durch nachfolgenden Dialog.



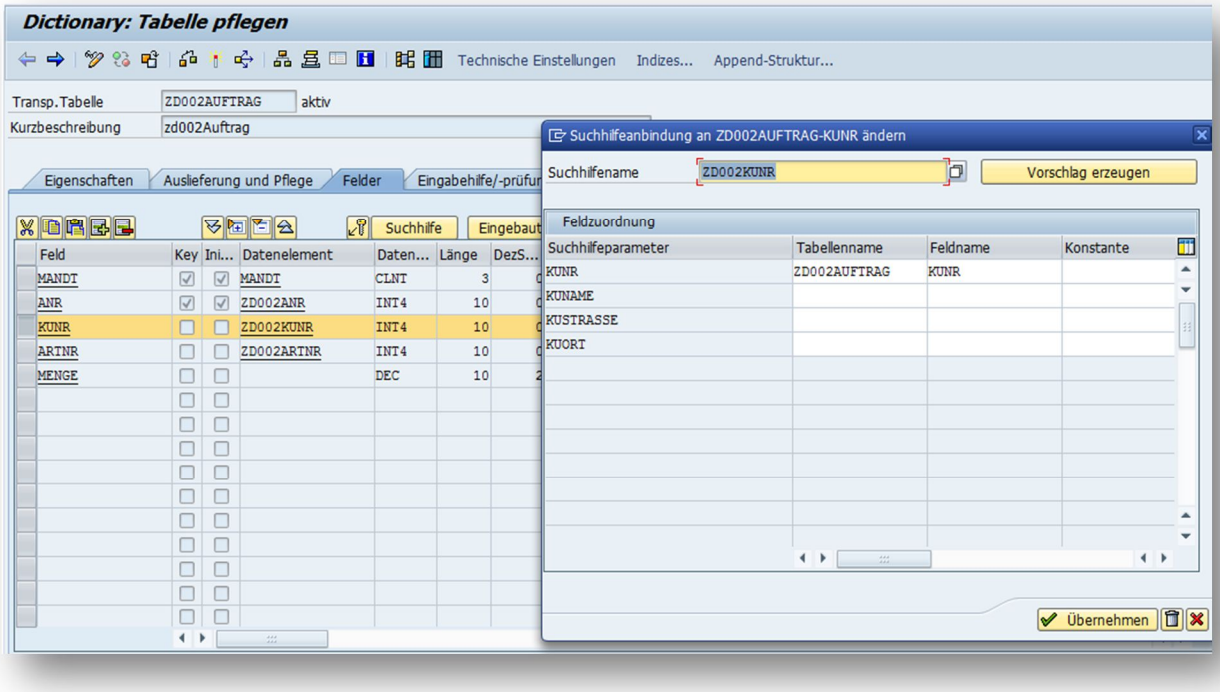
- **Definition der Viewfelder**: Nunmehr können die View-Felder in der nächsten Tablasche angegeben werden.



Weiter Einstellungen bieten die Tablaschen Selektionsbedingungen und Pflegestatus. So definierte Joints können wieder für die Suchhilfe verwendet werden.



3. Verwendung von Suchhilfen für Fremdschlüsselattribute. Definierte Suchhilfen auf Primärschlüssel können auch für Fremdschlüssel verwendet werden. Hierzu bietet die Attributsübersicht einer Tabelle das Butten „Suchhilfe“. Es öffnet sich die Suchhilfeanbindungen, die es erlaubt eine bestehende Suchmethode mit einem Feld (Attribut) zu verbinden. INFO: Wird keine Suchhilfe erstellt, erzeugt ABAP nur eine Primärkey-Liste als Suchhilfe – also nur der Schlüssel.



4. Literatur

- Horst Keller, Sascha Krüger, ABAP Objects, Einführung in die SAP-Programmierung, 2., korrigierte Auflage, Bonn 2001
  - → ACHTUNG: Das Buch enthält ein ABAP-Entwicklungssystem auf CD. Die Installation ist auf WIN2000, XP und Linux möglich.
- Horst Keller, Joachim Jacobitz, ABAP Objects Referenz; 1.Auflage, Bonn 2002
- Karl-Heinz Kühnhauser, Einstieg in ABAP, Bonn 2005
- Horst Keller, ABAP-Referenz, 2. Auflage, Bonn 2004
- Horst Keller, ABAP-Schnellreferenz; Bonn 2005
- Günther Färber, Julia Kirchner, ABAP-Grundkurs, 3.Auflage, Bonn 2005

5.SAP-Online-Hilfe im Internet

http://help.sap.com/