5. Bemerkungen zur modularen Programmierung

Literatur:
Gehring/Röscher: Einführung in Modula-2. de Gruyter 1989
Pomberger: SWT und Modula-2. Hanser 1987, 2., bearb. Auflage
Pomberger/Blaschek: Software Engineering. Hanser 1993

5.1. Allgemeine Grundlagen

■ **bisherige Praxis**: Aufgabe → Algorithmus → Programm
(mit entspr. Strukturierung)

➔ Methode für das Programmieren im „Kleinen“
➔ nutzen Bib. (Datei) von Prozeduren

■ **Forderungen zur SW–Entwicklung**:

1. Wiederverwendbarkeit (Anwend. auf Problemklassen)
2. Robustheit (bzgl. fehlerhafter Eingaben)
3. Anpaß- bzw. Erweiterbarkeit (auf leicht abgeänderte Aufg.-stell.)
4. arbeitsteiliger Entwicklungsprozeß

➔ Welche Methoden geeignet?

ein Verfahren – (Prinzip der) **modularen Programmierung**
(Programmierung im „Großen“)

■ Grundidee:

Aufgabe → Menge von Teilaufgaben ⇒ **Teil**programme
(≠ Unterprogramme)

Teilprogramm soll

+ überschaubare Einheit def. (relat. Begriff)

+ gewisse Funktionseinheit darstellen

+ Dienste bereitstellen
- Datenobj. in Form von Konst./Var.
- Datentypen
- Operationen (Proz., Fkt.)

mit dem Ziel, daß andere Teilprogramme diese Dienste nutzen können (kann auch Dienste anderer nutzen).

➔Bezeichnung: **MODUL**

Definition 5.1.

Ein **Modul** ist eine **Zus.-fassung von Objekten und Operationen** mit folgenden Eigenschaften:

1. Eine Kommunikation mit der Umgebung ist nur über definierte ***Schnittstellen*** möglich.

2. Die Zus.-fassung von Moduln zu einem Programm ist ***ohne Kenntnis des Modulinnern*** möglich.

3. Die ***Korrektheit***(-sprüfung) soll ohne Einbindung in das Ges.-programm möglich sein.

■ Modul ≠ **Prozedur**; Verallg. bzgl. Prozedur betrifft 2 Punkte:

1. **Schnittstellenbeschreibung :**

   hier klar definiert die

   a) vom Modul zu benutzenden Dienste
      (➔ **Import–Schnittst.**: welche Objekte und Operationen genutzt werden)
   b) vom Modul bereitgestellten Dienste
      (➔ **Export–Schnittst.**: welche Objekte und Operationen nach außen bereitgestellt werden)

2. **pr.-sprachliche Unterstützung :**

   Compiler muß Gültigkeitsprüfung der Export–Import–Bezieh. zum Zeitpunkt der Übersetzung ermöglichen

➔ Konsequenzen für PS
   z.B. in Pseudo–Kode:

```
MODULE  modulname;

    IMPORT  >liste der benöt. objekte und operationen<;

    EXPORT  >liste der außerh. verfügb. obj. u. oper.<;

    >Def. der zu exportier. Objekte und Operationen<;

    >Def. von internen Obj. und Operationen<;

    >Initialisierungsteil<;

END  {MODULE}.
```

Bsp.: zur Verdeutlichung der Unterschiede zw. Prozedur und Modul

Aufgabe – Lesen einer Zeichenfolge und deren Umkehrung.

1. Einfachste Lösung: Schreibe in Feld und gebe von „hinten“ an aus.
2. Nutze LIFO–Prinzip: organisiere Daten als Stapel (stack)

→ in Pascal:

```
PROGRAM bsp(input, output);

    TYPE  stack = ^stackelem;
          Stackelem = RECORD
                        info:  char;
                        next: stack
                      END;
    VAR  stapel: stack;     c: char;     z: boolean;

    PROCEDURE push(elem:char; VAR res:boolean);
      {Element in „stapel“ schreiben; res = true zeigt an, daß
       Operation ausgeführt}

    PROCEDURE pop(VAR elem:char; VAR res:boolean);
      {Lesen und Löschen des aktuellen Top–Elementes}

    FUNCTION empty:boolean;
      {Prüft, ob „stapel“ leer}

{HP}  BEGIN
        stapel := NIL;
        WHILE NOT (>endeanzeige<) DO
              BEGIN    READ(input, c);
                       push(c, z)
              END;
        WHILE NOT (>endeanzeige<) DO
              BEGIN    pop(c, z);
                       WRITE(output, c)
              END;
      END.
```

■ Klassen von Modulen

- einfache Funktions- oder Prozedurmodule
- Datenkapsel
- Abstr. Datentyp (ADT)

2 einander ergänzende Abstraktionskonzepte →

prozedurale A. trennt das „Was ?“ von Details des „Wie ?“

➔ spezifiziere, was eine Prozedur ausführen soll;
danach diese Prozedur für Problemlösung nutzbar;
später Implementierung;

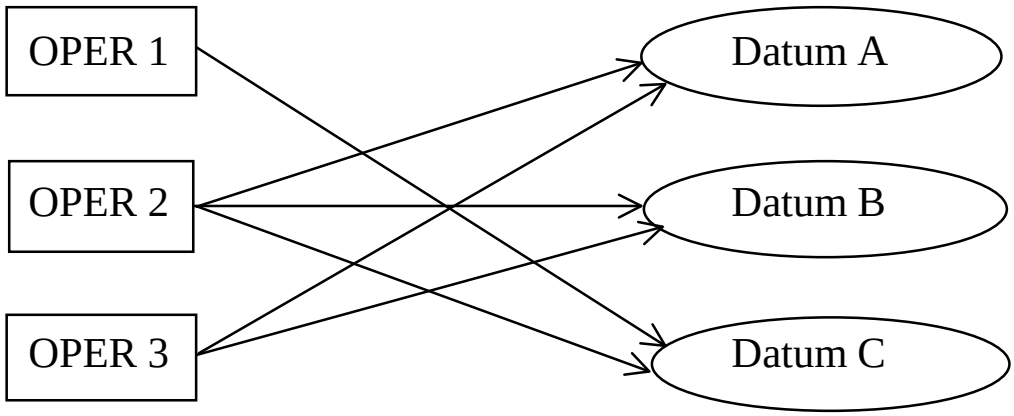
Datenabstraktion

➔ Idee: spezifiziere Datenobjekte für ein Problem **und** die darauf auszuführenden Operationen → logische Sicht auf DO

<u>1. einfacher Modul:</u>

- Idee so alt wie Idee der Fkt. und Proz. in höheren PS
- Konzept der funktionalen/prozeduralen Abstraktion
- sein Interface def. durch
  - ■ Def.-bereich D (Menge zuläss. Übergabewerte)
  - ■ Wertevorrat V (Menge möglicher Resultatwerte)
- seine Leistung def. als **<u>eine</u>** Operation Op : D → V
- **<u>keine</u>** Speicherwirkung, d.h., bei v=Op(d) werden Werte d∈ D und v ∈ V nicht im Modul gespeichert
- klass. PS unterstützen diese Modultechnik durch Fkt.- und Prozedurausdrucksmittel

Verwendung von Prozeduren (zur Strukturierung) ➔ viele Prozeduren greifen auf globale Datenbestände zu



im Ergebnis:
a) die DS sind schwer änderbar
b) derartig aufgebaute Pr.-systeme schwer zu testen { bei Fehlern i.allg. schwer, die verantwortl. Operation (Prozedur) zu finden }

! Unbeschränkter Zugriff auf gemeinsame Daten ist riesige Fehlerquelle !!!

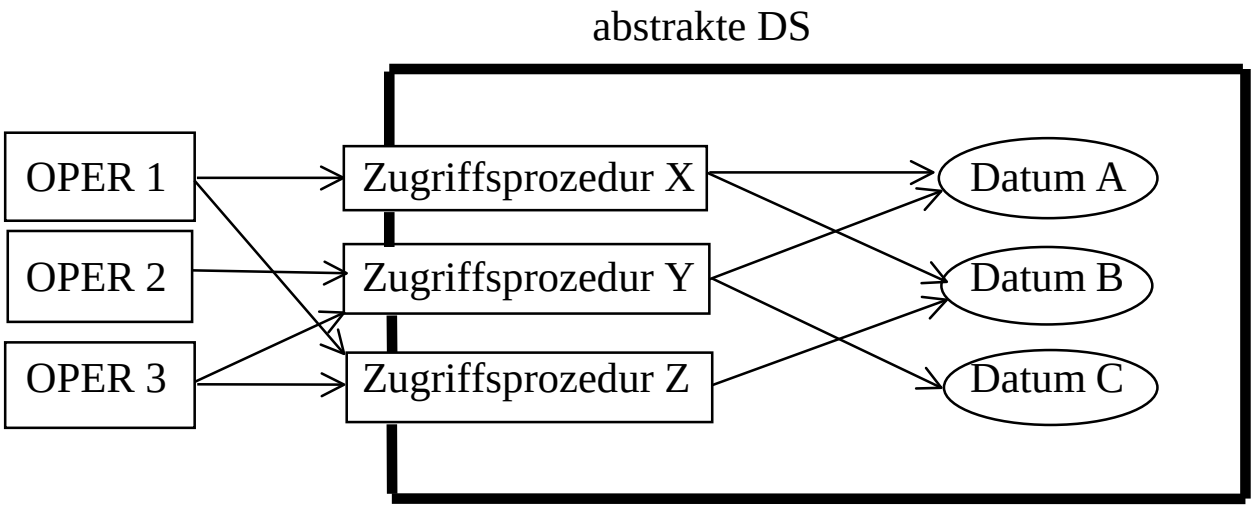
➔ als ein AUSWEG: **Geheimhaltung** solcher Datenstrukturen vor ihrer Umgebung durch Zus.-fassung der Datenstruktur mit versch. Zugriffsprozeduren zu einem Modul ➔

2. Abstrakte Datenstruktur:

■ Grundlage: Konzept der Datenabstraktion
ein abstr. DO, dessen Eig. durch eine endl. Menge von Operationen über diesem ADO def. sind

■ **abstr. DS** := Menge von Objekten und Menge von Operationen, die auf die Objekte angewendet werden können

Zugriff von 3 Operationen auf eine abstrakte Datenstruktur



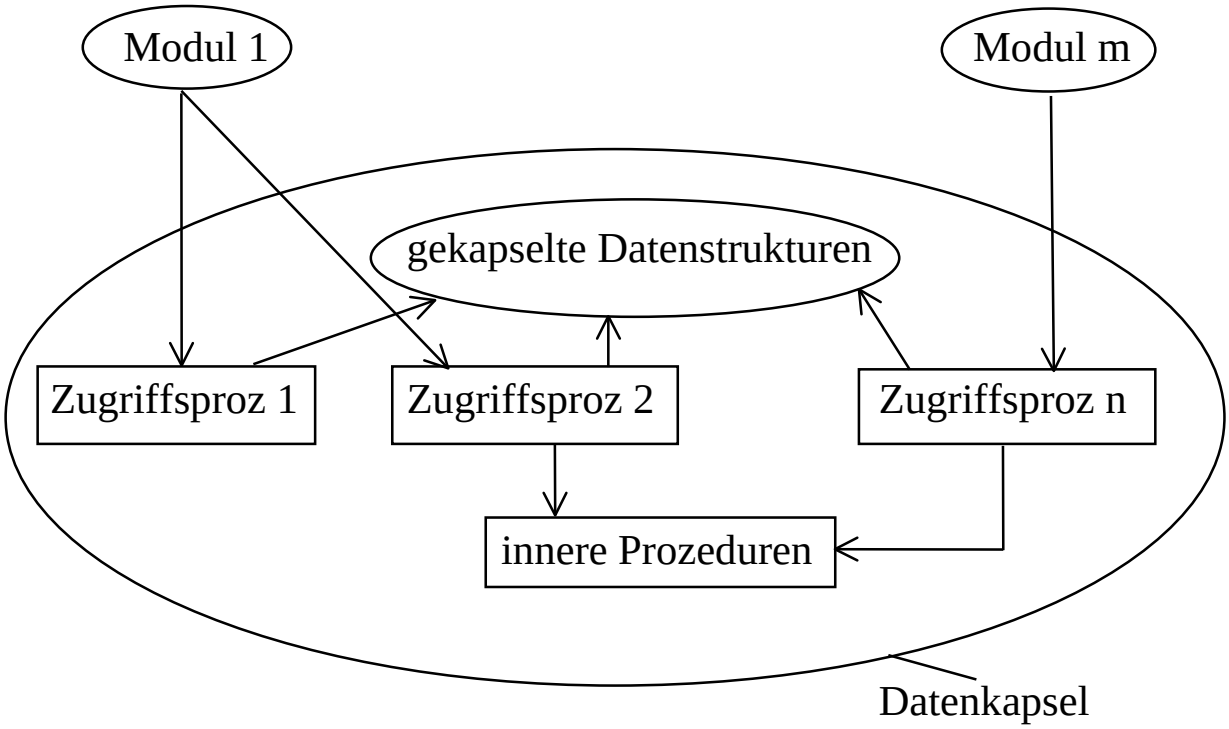
auf Daten nur noch über ausgezeichnete Operationen zugegriffen (Zugriffsoperationen, -prozeduren)

- Unterschied zu 1.: Speicherwirkung
- Bsp.: Stapel STACK (nach Org.-prinzip LIFO)
  WS QUEUE (nach Org.-prinzip FIFO)
  Liste LIST

- *Implementierung* am besten durch **Datenkapsel** := Modul aus der Deklaration von **Daten und** einer Sammlung von **Prozeduren**, die diese Daten verwalten.
  Einige dieser Prozeduren sind von außen zugänglich – die *Zugriffsprozeduren*. Die Daten selbst und alle Prozeduren, die keine Zugriffsprozeduren sind, sind von außen nicht zugänglich.

  ➔ **Kapselung** (encapsulation)

➔ etwas allgemeineres Schema:



■ Nachteile der Datenkapselung:

dyn. und statische Ineffizienz

■ Vorteile:

- Datenbestand tritt nach außen nicht in Erscheinung (Informationsverbergung, -hiding)
  - ➔ schränkt Fehlermöglichkeiten ein
  - ➔ erleichtert Testen
  - ➔ erleichtert Änderungen von Datenrepräsentationen
- schließt Verwendung externer Größen aus
  - ➔ Beitrag zum Entkoppeln von Moduln
- fördert Verwendung abstr. Datentypen und abstr. DS
- fördert Top–Down–Entwurf
- schützt davor, zu früh an best. Repräsentation der Daten zu denken
- erhöht Lesbarkeit und hilft bei Dokumentation

3. abstrakter Datentyp :

- abstr. DS zur Realisierung nur eines Datenbestandes
  sind mehrere Exemplare gleichzeitig zu bearbeiten ➔

abstrakte Datentyp

:= Menge von Objekten, die alle dieselbe Datenstruktur haben und deren Eigenschaften durch eine endliche Menge von über diesen Objekten ausführbaren Operationen definiert sind

- ADT erlaubt, beliebig viele ADO einer Klasse zu erzeugen, zu manipulieren und einzeln zu vernichten
- Implementierung wieder als Datenkapseln; Begriff AD–„TYP“ assoziiert den Unterschied zur Datenkapsel
- auch hier Zugriffsroutinen; dabei aber Daten nicht mehr im Modul gekapselt, sondern jeder Operation als Parameter übergeben
- Modul, der ADT realisiert, muß über Operation verfügen, durch die Variablen (Exemplare) des betreffenden ADT´s in beliebiger Zahl geschaffen werden können; ein solches geschaffenes Objekt heißt ADS oder ADO

Bemerkung:

einfacher Modul	-	ein ADT mit genau einer Operation
Datenkapsel	-	Modul, der zwar alle Operationen eines ADT realisiert, aber nur genau ein Objekt bereitstellt

Modul, der ADT realisiert – muß Operation haben, durch die Variablen des betreffenden ADT´s in beliebiger Zahl geschaffen werden können (ein solches Objekt heißt ADS oder ADO)

■ Beispiel: **Telefonverzeichnis**

- **Problem:** Verz. von Namen und Nr. erzeugen; später neue Namen und Nr. hinein; Nr. ändern; versch. Namen ausgeben
- **Lösung:**

A) I/O–Anforderungen:

Inputs:

Anfangsverz.: Namen und Nr. über eigene Zeile eines Text–Files nacheinander gelesen

zusätzl. Einträge: durch Nutzer über Tastatur

Outputs:

indiv. Nr.: Name und Nr. jeder durch Nutzer ausgew. Person auf eigenen BS–Zeile

akt. Verz.: Name und Nr. aller Einträge zeilenweise in ein Text–File

B) Entwurf der Datenstruktur:

2 Objekte – Verzeichnis als Einheit; indiv. Eintrag; jedes als ADT;

Operationen für Objekt „*Eintrag*“:

lesen + speichern eines neuen Eintrages
Anzeige eines neuen Eintrages
Prüfung des Namens eines Eintrages
Abfrage des Namens oder der Nummer eines Eintrages

Operationen für Objekt „*Verzeichnis*“

erzeuge neues Verzeichnis
lesen aller Einträge in Verz. aus Text–File
einfügen eines neuen Eintrages in Verzeichnis
editieren, entfernen, finden eines Eintrages im Verz.
schreiben Verz.-inhalt in Text–File

Bem.: Verz. als lineare Liste → allg. Informationsgehalt.

C) Spezifikation für ADT *Eintrag* :

Eintrag → Name und Nummer als Zeichenketten

folg. Parameter benutzt:

AnEntry vom Typ Eintrag
AName vom Typ Namensstring
ANumber vom Typ Tel.-nr.-string

Operatoren:

Init(VAR AnEntry, AName, ANumber) →

speichert AName und ANumber in AnEntry

ReadEntry(VAR AnEntry) →

liest Name und Nr. von Tastatur in AnEntry

DisplayEntry(AnEntry) →

zeigt Name und Nr. auf BS an

EqualName(AnEntry, AName) →

(Fkt.) testet, ob AName der Name von AnEntry

GetName(AnEntry) →

(Fkt.) liefert Name von AnEntry

GetNumber(AnEntry) →

(Fkt.) liefert Nummer von AnEntry

D) Spezifikation für ADT *LinList*:

Verz. als lin. Liste von Elementen des Typs Eintrag
noch Zeiger auf (aktive) El. der Liste
nur darauf unmittelbarer Zugriff
folg. Parameter: Directory vom Typ LinList
AnEntry vom Typ Eintrag

Operatoren:

Create(VAR Directory) →
erzeugt leeres Verz.; Zähler der Anzahl Einträge wird Null gesetzt; Zeiger auf 1. zuläss. Position

ReadLinList(VAR Directory) →
liest aus Text-File; Zähler zählt; Zeiger auf nächste zuläss. Position

Insert(VAR Directory, AnEntry) →
neuen Eintrag am Ende; Zähler +1; Zeiger rückt weiter

WriteLinList(Directory) →
schreibt Einträge in gespeicherten Reihenfolge in ein Text–File; jeder Eintrag auf 2 separaten Zeilen des Files

GetEntry(Directory, VAR AnEntry) →
liefert Eintrag der Zeigerposition

PutEntry(VAR Directory, AnEntry) →
speichert AnEntry in Directory an Zeigerposition; vorhandener Inhalt wird überschrieben

GetCount(Directory) →
(Fkt.) gibt Anzahl Einträge von Directory zurück

versch. Operatoren für Zeiger ➔

GoFirstEntry(VAR Directory) →
positioniert Zeiger auf 1. Eintrag

GoLastEntry(VAR Directory) →
positioniert Zeiger auf letztem Eintrag

GoNextEntry(VAR Directory) →
positioniert Zeiger auf folgendem Eintrag

GoFindEntry(Directory, VAR AnEntry) →
positioniert Zeiger auf Eintrag mit gleichem Namen wie AnEntry; Zeiger auf 1. Position nach Ende, wenn AnEntry nicht gefunden

EndOfLinList(Directory) →
(Fkt.) testet Zeigerposition; gibt TRUE zurück, wenn Zeiger auf 1.Stelle nach Liste zeigt; FALSE, wenn Zeiger auf Listenelement

E) Nutzung der Operatoren der ADT´s :

(1) Einrichten eines neuen Verz. → Freunde : LinList;
→ Neu, Ziel: Eintrag;

Create(Freunde);
ReadLinList(Freunde);

(2) neuen Eintrag über Tastatur

ReadEntry(Neu);
Insert(Freunde, Neu);

(3) Änderung der Nr. eines spez. Eintrages

```
ReadEntry(Ziel);
GoFindEntry(Freunde, Ziel);
IF NOT EndOfLinList(Freunde)
    THEN PutEntry(Freunde, Ziel);
```

(4) Ausgabe eines spez. Eintrages des Verzeichnisses

```
Init(Ziel, 'namensstring', '');
GoFindEntry(Freunde, Ziel);
IF NOT EndOfLinList(Freunde)
    THEN
        BEGIN
            GetEntry(Freunde, Ziel);
            GetNumber(Ziel)
        END;
```

usw., usw., ...

Bem.: Bisher über die Implementierung kein Wort!

■ **Bem. zur Methodik** der Arbeit mit (solchen) Moduln:

- SW–Entwicklung soll arbeitsteil. Prozeß sein;
  seien M1 und M2 zwei Moduln, wobei M2 etwas aus M1 importiert
  ➔ M1 muß vor M2 verfügbar sein ?
  ➔ ? Was heißt „Arbeitsteilung“ ???

  Möglich: Jeder arbeitet an Teilaufgabe, deren Zw.-entw. testbar sind.
  ➔ 2 Forderungen:
  1. **separate** Übersetzbarkeit
  2. Modul zerfällt in
     a) Teil, der nur die Schnittstellenbeziehungen beschreibt und
     b) Teil, der nur die Operationen realisiert, d.h., in
     Definitionsteil und **Implementierungsteil.**

- bildlich: Modul ⇔ (Pascal–)Programm

  Spezialgeschäft ⇔ Kaufhaus

  ↓

  2 Teile des Moduls:

  Def.-teil: Angebotsliste des Geschäfts
  keine Rezepte; animiert nur zum Kauf

  Impl.-teil: „Hinterzimmer“ des Geschäftes
  dort Rezepte bekannt **und** umgesetzt

  ➔ Def.-teil für Interface, zeigt sichtbaren Funktionen
  ➔ Impl.-teil für Ausführ. der Arbeit, zeigt inneren Mechanismus

Bem.: Diese 2 Teile auch zur Unterstützung der Pr.-ung mit ADT’s – Interface beschreibt den ADT, Implementierung beschreibt seine Konkretisierung.

➔ Zerlegung einer Aufgabe := Aufschreiben der
Schnittstellenbeziehungen zw. den Modulen.

- **Modul soll :**

  + überschaubar sein;

  + minimale Schnittstellen haben
    (funktional zus.-gehör. Dinge zus.;
    geringer Import – Export)

  + unabhängig sein
    (Änder. im Modul soll für Umgebung ohne Wirkung sein)

    ➔Modul**nutzung** = Nutzung der Dienste

  ! Modul kann nicht gerufen werden,
  höchstens sein(e) Dienst(e) !!!

  + abgeschlossen sein
    (in funktionaler Sicht; z.B. als Datenkapsel)

  + testbar sein
    (die Funktionalität muß in Ordnung gehen;
    sie muß unabhängig von Umgebung testbar sein)

5.2. Modul–Implementation in Pascal

■ Ausgangspunkt:
- Pascal ist keine modulare PS (➔ alles über Prozeduren)
- TP als Erweiterung schon mit Elementen der modul. Progr.
- echt modulare PS: Ada, Modula-2, Euklid, …

■ Realisierung in TP: (i.w. ab 4.0)
- ∃ Übersetzungseinheiten ➔ können **separat compiliert** werden
- heißen hier **UNIT** := Programm–Modul als Sammlung von Typendeklarationen, Konstanten, Variablen, Prozeduren und Funktionen, das separat übersetzbar ist und von anderen Programmen (ohne erneute Compilation) verwendet werden kann.

  ➔TP–Programm sehr ähnlich

■ Aufbau einer Unit:

Spracherweiterung durch Grundsymbole UNIT, INTERFACE, IMPLEMENTATION und USES

```
UNIT   unitname;

INTERFACE
     USES     unitliste;
              öffentliche Deklarationen;

IMPLEMENTATION
     USES     unitliste;
              private Deklarationen;

[    Initialisierungsteil    ]

END.
```

➔ Unit besteht aus

- Kopf: **UNIT** *Name*;

- Interfaceteil: **INTERFACE** {zur Beschreib. des Exp./Imp. bzw. der Schnittstelle}
  USES {Grundsymbol für zu import.Teil}
  >>Liste der Units, deren Dienste benötigt werden<<;
  {öffentlich benutzte Units}

  { öffentliche Deklarationen: Def. der exportierbaren Dienste; das bedeutet u.a., daß nur die Köpfe der Proz./Fkt. angegeben werden }
  >>Def. von Konstanten und Variablen<<;
  >>Def. von Typen<<;
  >>Def. von Prozeduren und Funktionen<<;

- Implem.-teil: **IMPLEMENTATION** {priv. Teil der Unit }

  USES {lokal bzw. privat benutzte Units}
  >>Liste der Units, deren Dienste benötigt werden<<;

  { private Deklarationen: Proz./Fkt., deren Kopf nicht im Interface–Teil deklariert; die dort deklarierten nun hier implementiert, wobei Kopf ohne Parameterliste möglich}
  >>Def. der lokalen Objekte<<;
  >>Def. der Proz. und Fkt. des Interface–Teiles<<;

- Initialis.-teil: **BEGIN**
  >>Anweisungen<<

{ optional; hier stehender Kode jedesmal beim Start des Anwendungs-programms ausgeführt; z.B. für Anfangswerte von best. Objekten }

- Ende der Unit:**END.**

Bem.: USES–Anw. sowohl im Interface, als auch in Implementierung. Eine andere Unit aber nur an einer Stelle möglich.

■ Beispiel: (Pseudo–)Zufallszahlen

```
UNIT    zufall;

INTERFACE
    FUNCTION rand: Integer;

IMPLEMENTATION
    CONST   max=32767.0; a0=30000.0; b0=17000.0;
    VAR     a,b,c: Real;
    FUNCTION rand;
        BEGIN
            c := a+b;
            IF c > max THEN c := c – (max+1);
            c := c * 2.0;
            IF c > max THEN c := c – max;
            a := b; b := c;
            rand := trunc( c )
        END;
BEGIN
    a := a0; b := b0
END.
```

```
PROGRAM zzahlen;            {HP}

USES zufall, crt;                   {import. Teile}
VAR i,n:Integer;

BEGIN
  CLRSCR;
  WRITE('Geben Sie die Anzahl benötigter ZZ ein !  n =  ');
  READLN(n);
  FOR i:=1 TO n DO WRITELN(rand:10);
  READLN
END.
```

- **Bem.:**
  1. Def.-teil erzeugt keine ZZ; beschreibt nur das Interface des Programms – Lieferant von ZZ.
  2. Er garantiert nicht für fehlerfreie Arbeit des angebotenen Dienstes (Funktion rand). Garantie nur dafür, daß das Interface so ist wie angegeben.
  3. Nutzung der Unit zufall: Kann Implementierung der Unit zufall ändern ohne Auswirkungen auf Programm zzahlen. **Geheimnisprinzip:** Wie etwas gemacht wird, bleibt verborgen; nur Anwendungsbedingungen werden öffentlich gemacht.

■ Arbeitshinweise (programmtechn. Art):

- Unit zuerst editiert; i.d.R. unter gleichem Namen abgespeichert wie sie editiert wurde

  ➔ **zufall.PAS** { Dateityp }

- Compilation wie bei Programm; anstelle einer .EXE–Datei nun aber .TPU–Datei

  ➔ **zufall.TPU** {Turbo Pascal Unit }

  ! Unit–Name muß mit Datei–Namen identisch sein !!!

  Compiler–Option *Memory* → Unit steht im Speicher

  Disk → Unit in Datei–System der Festplatte

- Für Aufnahme einer Unit in Anw.-programm → ihr Name nach USES

  ➔Compiler sucht nach (Datei der) Unit

  {Anw.-pr. und Unit nicht im gleichen Verzeichnis → Pfad angeben}

- ∃ Stand.-bib. von Units ➔ **TURBO.TPL** { TP–Library }

  + Unit DOS: kann aus einer Unit auf DOS zurückgreifen
  + Unit CRT: versch. Prozeduren zur BS–Gestaltung, z.B. clrscr
  + Unit PRINTER: erleichtert Arbeit mit Drucker
  + Unit SYSTEM: ist die TP „Runtime Library“; autom. in Pr.
  + Unit GRAPH; …
  + Unit OVERLAY: Overlay–Manager für Overlay–Units (die in einem Pr. denselben Speicherplatz benutzen)

Bem.: Wird Unit genutzt, muß sie gesucht werden.
Suche startet in TPL, danach zu TPU.
➔ TPU–Unit zu TPL–Unit umwandeln.

■ Das **Include–Prinzip**:

- sei in File–System ein Pr.-text unter best. Dateinamen abgelegt
- nun (als Übersetzungseinheit durch Compiler akzeptiert) an best. Stelle zum Ztpkt. der Compilation holen
  ➔ { $I dateiname }
  eigentlich Kommentar mit Compilerdirektive I = Include
- primitive Makro–Form: Text wird hereingeholt
  Nachteil: Sehe nicht, was das Pr.-teil macht !

■ Das **Overlay–Prinzip**:

- „kleines“ Pr. kann sehr groß werden über Unit–Prinzip
  ➔ Speicher reicht zum Übersetzungsztpkt. nicht aus
- Aber: Brauche nie gleichzeitig alle Pr.-teile.
  ➔ Idee: Dienste zur Laufzeit bereitgestellt und nach Benutzung „überschrieben“.
  ➔ dyn. Laden (zur Laufzeit) zur Abarbeitung von Pr.
- zu Lasten der Laufzeiteffizienz
- TP hat entspr. Stand.–Unit dafür