# Grundvorlesung Informatik

Universität Stuttgart, Studienjahr 2002/03

~~**0. Vorbemerkungen** (14.10.02)~~

~~**1. Grundlagen der Programmierung** (17.10.02 - 5.5.03)~~

**2. Interaktionen** (8.5. - 26.5.03)

**3. Grundlegende Verfahren** (2.6. - 25.7.03)

Klausur (Orientierungsprüfung) am 5. August 2003

Hochschullehrer: Volker Claus, Fakultät 5 "I, E u. I"
Institut für Formale Methoden der Informatik (FMI)



## Teil 2 der Grundvorlesung

### 2. Interaktionen

2.1 Objektorientierung

2.2 Prozesse

2.3 Internet, Vernetzung

2.4 Modellierung

## 2.1 Objektorientierung

Wiederholungen aus früheren Abschnitten

Ergänzungen zu Typen

Realisierung einfacher objektorientierter Ansätze in Ada

Unterbereichsdatentypen und Basistyp, siehe 1.3.2.1
Varianter Typ siehe 1.3.2.4
Pakete (Modul, package) siehe 1.4.3.4
Polymorphie siehe 1.4.4
Vererbung siehe 1.4.5
(und in Ada "abstract types" 1.4.5.2)
Objekte siehe 1.4.7



### 2.1.1 Verbundtypen mit Parametern (Diskriminanten) in Ada

Am Beispiel:

```
type Zeichenpuffer (Laenge: Positive := 80) is
        record  Position: natural range 0..Laenge;
                Inhalt: array (1..Laenge) of Character;
        end record;
```

Spätestens bei der Deklaration von Variablen müssen die Parameter konkret angegeben werden:

```
X: Zeichenpuffer;              -- hier ist Laenge mit 80 vordefiniert
Y: Zeichenpuffer (30);         -- hier ist die Laenge gleich 30
Z: Zeichenpuffer (Laenge => 30) := (0, (1..20 =>' '));
```

## 2.1.2: zu 1.3.2.4 Vereinigung von Typen *in Ada*

Gegeben seien die Mengen $M_1$, $M_2$, ..., $M_n$, die zu den Datentypen $T_1$, $T_2$, ..., $T_n$ gehören. Dann kann man nach folgendem Schema hieraus den Datentyp T konstruieren, dessen Wertemenge die disjunkte Vereinigung dieser Mengen

$$M = M_1 \cup M_2 \cup ... \cup M_n$$

ist. Ada-Darstellung:

```
type T (index: 1..n) is record
    case index is
        when 1 => S1: T1;
        when 2 => S2: T2;   ...
        when n => Sn: Tn;
    end case;
end record;
```

falls mehrere Möglichkeiten, mit senkrechtem Strich trennen:

```
when 1 | 3..6 => Sk: Tk;
```



Statt des "index", der hier aus der Menge {1, 2, ..., n} ist, kann ein beliebiger Name und ein anderer endlicher Datentyp gewählt werden. Dieses auswählende Element heißt "Diskriminator".

Die Menge ist eine disjunkte Vereinigung, weil durch den Diskrimator "index" genau eine Menge ausgewählt wird, die Mengen also als verschieden angesehen werden, auch wenn verschiedene $M_i$ gleiche Elemente enthalten sollten. Wichtig: Die Aufzählung muss vollständig und eindeutig sein!

Natürlich können im record noch weitere Komponenten enthalten sein, so dass man bei der disjunkten Vereinigung in der Programmierung von einem "varianten Anteil" ("variant part") spricht. An folgendem Beispiel wird dies klar.

Bei Fahrzeugen kann man an verschiedenen Dingen interessiert sein: Wir nehmen an, dass für alle Fahrzeuge die Länge, die Breite, die Höhe und die Fahrzeugnummer vorliegen müssen, bei Bussen die Zahl der Sitzplätze, bei Lastkraftwagen die Größe der Ladefläche in qm und bei einem PKW die Zahl der Airbags. Dies führt zu folgendem Datentyp:

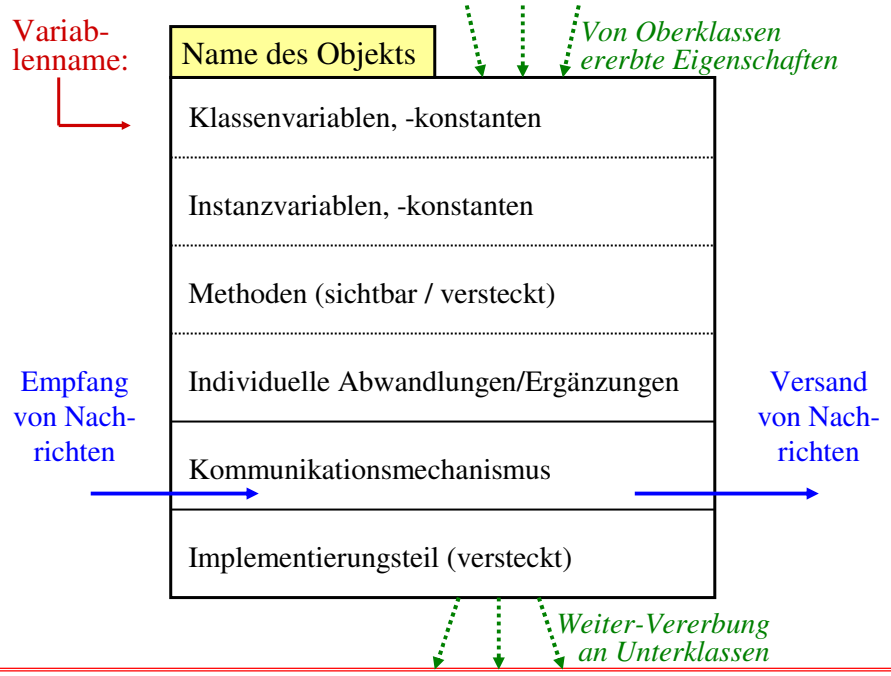
```
type mass is delta 0.001 range 0.0 .. 50.0;
type art is (PKW, Kleinbus, LKW, Bus);
type fahrzeug (modell: art) is record
    länge, breite, höhe: mass;
    nummer: positive;
    case modell is
        when Kleinbus | Bus => sitzplätze: 8 .. 150;
        when LKW => ladefläche: positive;
        when PKW => airbagzahl: 0..10;
    end case;
end record;
```



## 2.1.3: Erinnerung (Wiederholung aus 1.4.7):

Objekte sind in sich geschossene Einheiten, die

- wie Moduln aufgebaut sind: Es gibt ein Schema, genannt "Klasse", das vor allem aus "Attributen" (das sind die einzelnen Datenstrukturen der Variablen) und "Methoden" (das sind die algorithmischen Teile) einschl. der Angaben zur Sichtbarkeit besteht und aus dem ein neues Objekt erzeugt werden kann; das Objekt ist eine Instanz (oder ein "Exemplar" oder eine "Ausprägung") dieser Klasse.
- einen individuellen Zustand besitzen (Speicherzustand der Klassen- und Instanzvariablen),
- miteinander kommunizieren können; dies geschieht durch Nachrichtenaustausch ("message passing"),
- durch Vererbung ihre Eigenschaften an neue Objekte bzw. Klassen weitergeben können,
- Variablen werden oft "dynamisch gebunden".






Prinzipien der Objektorientierung:

1. Es gibt nur Objekte. Jedes Objekt ist eindeutig identifizierbar über seinen Namen.
2. Alles wird über Klassen, Instanzbildung, Zustände, Methoden, Nachrichten und Vererbung realisiert.
3. Objekte handeln in eigener Verantwortung (und sie geben nur bekannt, *was* sie bearbeiten, niemals, *wie* sie dies tun).
4. Klassen werden in Bibliotheken aufbewahrt und stehen allen Programmen und Klassendefinitionen zur Verfügung.
5. Programmieren bedeutet, Klassen festzulegen, hieraus Objekte zu erzeugen und diesen Aufgaben zu übertragen, indem man ihnen geeignete Nachrichten schickt. Die Auswertung der Objekte erfolgt hierbei erst zur Laufzeit (Polymorphie, dynamische Bindung der Objekte an Variable).

Wenn "alles" Objekte sind, so sind konsequenterweise auch Klassen, Nachrichten und die (formalen) Parameter Objekte.

Eine Methode der Klasse *"Klasse"* ist "new". Diese Methode erzeugt aus der Klasse eine Instanz, also ein konkretes Objekt. Jede Klasse ist eine Unterklasse der Klasse *"Klasse"* und hat somit diese Methode ererbt, kann also Instanzen von sich selbst erzeugen.

Eine konkrete Nachricht, die ein Objekt A an ein Objekt B schickt, besitzt meist aktuelle Parameter. Diese sind ebenfalls Objekte. Ebenso erwartet das Objekt A, dass das Objekt B ihm eine Nachricht mit konkreten Objekten als aktuellen Parametern zurückschickt.



Hinweise (1):

a. Klassen bilden *Hierarchien* oder zyklenfreie Abhängigkeitsgraphen (Ober- / Unterklassen, einfache / mehrfache Vererbung).
b. In typisierten Sprachen verweist eine Variable auf ein Objekt ihrer Klasse oder einer ihrer Oberklassen. Anderenfalls muss die Zuordnung laufend auf ihre *Zulässigkeit* überprüft werden.
c. Ist das zu aktivierende Objekt identifiziert, so muss man ermitteln, *welches die angeforderte Methode konkret ist* (eventuell muss man diverse Oberklassen durchsuchen) und ob es sie ausführen kann.
d. Zu jeder objektorientierten Sprache gibt es umfangreiche *Klassenbibliotheken*, aus denen man sich sein Programm aufbauen kann.
e. In der Praxis benötigt man eine *Entwurfsumgebung*, um Programme im Team zu entwickeln, um Erläuterungen, Entwurfsprozesse und verschiedene Dokumentationen zu erstellen und um die Klassenbibliothek zu erweitern.

Hinweise (2): Probleme in der Praxis, kleine Auswahl:

f. Die Sprache muss Erweiterungsmöglichkeiten von Moduln haben, ohne dass hierbei die privaten Informationen bekannt werden. (Das ist oft nicht realisierbar. Ada: child library units.)
g. Es müssen verschiedene Sichtweisen auf ein Objekt möglich sein. (Hierfür kann man Mehrfachvererbung nutzen.)
h. Es müssen Teile getrennt voneinander übersetzbar sein und abgelegt werden, allerdings darf hierdurch die Sichtbarkeit nicht beeinträchtigt werden. (Stichwörter in Ada: separate, stub.)
i. Die Klassenbibliotheken sollten sowohl den Quellcode als auch bereits getrennt compilierte Teile besitzen, und zwar so, dass diese leicht in ein Programm (z.B. über with und use) eingefügt werden können.



Hinweise (3): Probleme in der Praxis, kleine Auswahl:

j. Die Eindeutigkeit von Namen ist zu gewährleisten, z.B. durch Umbenennung importierter Größen (in Ada renames:
```
with Stack_für_Zeichen;
X: StackZ renames Stack_für_Zeichen.S;
function "*" (X,Y: Vektor) return float renames Skalarprodukt;)
```
k. Die Sichtbarkeit in der Vererbungshierarchie ist genau festzulegen. (Beispiel: In der Regel sehen Unterklassen nicht, wie ihre Oberklassen die Methoden realisiert haben; daher können sie diese auch nicht verwenden, um Umdefinitionen vorzunehmen. Ändert man eine Methode ab, so muss man aber meist Zugriff auf jene soeben ausgeblendete Methode haben. In Ada: über Punkt-Notation. In Java durch "super".)
l. Die Sichtbarkeitsregeln müssen auch in der Klassenbibliothek gelten. Beispielsweise wird durch "with" die Sichtbarkeit einer anderen Klasse importiert (wann und wo endet diese?).

Hinweise (4): Probleme in der Praxis, kleine Auswahl:

m. Wie entwirft man "objektorientiert"? Man unterscheidet zwischen OOA und OOD, also "objektorientierte Analyse" und "objektorientierter Entwurf" ("D" = design = Entwurf). In der OOA wird ein Sollkonzept/Pflichtenheft erstellt und für dieses werden geeignete Klassen mit Zusatzinformationen (zeitliche Abläufe, einzuhaltende Bedingungen, Zusammenwirken der Einheiten, ...) erarbeitet. Im OOD werden hieraus die tatsächlichen Klassen, möglichst aus einer Klassenbibliothek und ergänzt um zusätzlich erforderliche Hilfs-Klassen erstellt und die Realisierung in einer Programmiersprache skizziert. (Anschließend folgt die Codierung.)

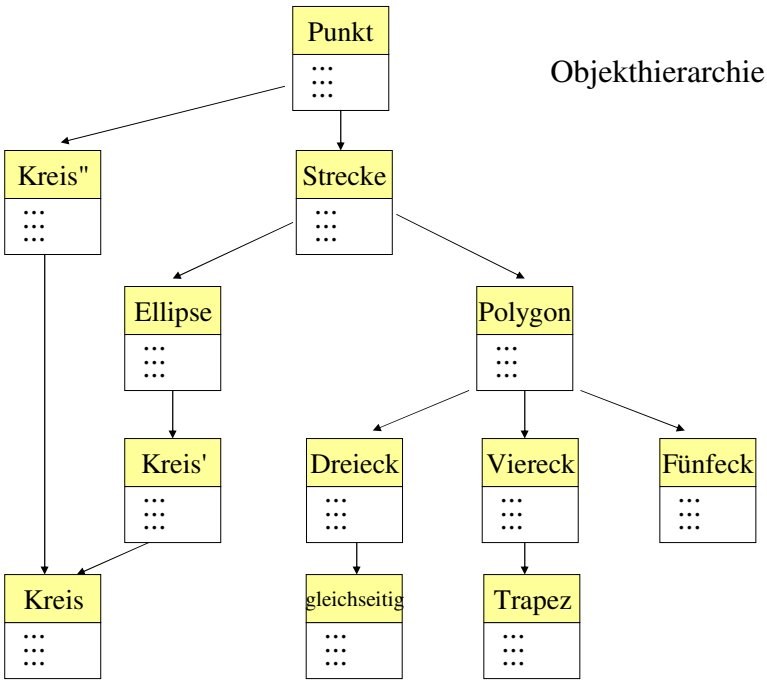
n. Die Zeit, dass man Lösungen zu Problemen von Grund auf neu schrieb, ist vorbei. Heute versucht man, eine Problemlösung so zu beschreiben, dass sie mit Hilfe der vorhandenen Klassen realisiert werden kann. Nur für wenige Probleme werden noch neue Klassen, neue Datentypen, neue Vorgehensweisen entwickelt (die anschließend in die Klassenbibliothek übernommen werden).



Klasse "integer" (Oberklasse sei "Zahlen"):

| integer | Vererbt von "Zahlen" werden: Typ "Binärfolge" und hierauf die Operationen plus, minus, mal, "<", "=", wie sie üblich sind. Der Ausbau zu "integer" erfolgt jetzt: |
|---|---|
| Klassenvariablen | Null: constant Binärfolge := 0; Zehn: constant Binärfolge := 1010 |
| Instanzvariablen | INH: Binärfolge |
| Methoden | function "+" (X: integer) return integer; function quad return integer;<br>function "abs" return Binärfolge; ... |
| Kommunikation | procedure return1 (A:Binärfolge) is begin X := new Integer; X.INH := A; send X end;<br>procedure send (A:Object) is begin "schicke A an den Sender zurück" end; ... |
| Implementierung | procedure "+" (X: integer) is begin return1 (plus (self.INH, X.INH)) end;<br>procedure abs is begin if self.INH<0 then send minus(Null,self.INH) else send self.INH fi end;<br>procedure quad (X: integer) is begin return1 (mal (self.INH, self.INH)) end; ... |

Es bleibt der jeweiligen Sprache überlassen, ob die im sichtbaren Bereich aufgelisteten Funktionen tatsächlich als Funktionen oder auf andere Weise (z.B. wie hier als Prozeduren; aber durch "return1" wird die richtige Sicht nach außen hergestellt) implementiert werden.




Objekthierarchie

17

Ada ist keine "rein objektorientierte" Sprache wie etwa Smalltalk. Objektorientierte Vorgehensweisen lassen sich aber in Ada realisieren, insbesondere über die Spezifikation (auch Schnittstelle genannt) von Paketen.

Wir geben ein Beispiel für objektorientiertes Vorgehen in Ada an. Dieses stammt aus dem Buch von Manfred Nagl, "Softwaretechnik mit Ada95", Kapitel 5.4.

Ziel ist der Entwurf eines Warnsystems, das mit unterschiedlichen Dringlichkeitsstufen arbeitet. Das System wird zunächst in konventioneller Weise präsentiert und anschließend nach objektorientiert neu geschrieben.
(Nähere Erläuterungen erfolgen in der Vorlesung, nachlesbar im o. g. Buch.)

## 2.1.4: Warnsystem, konventionelle Darstellung

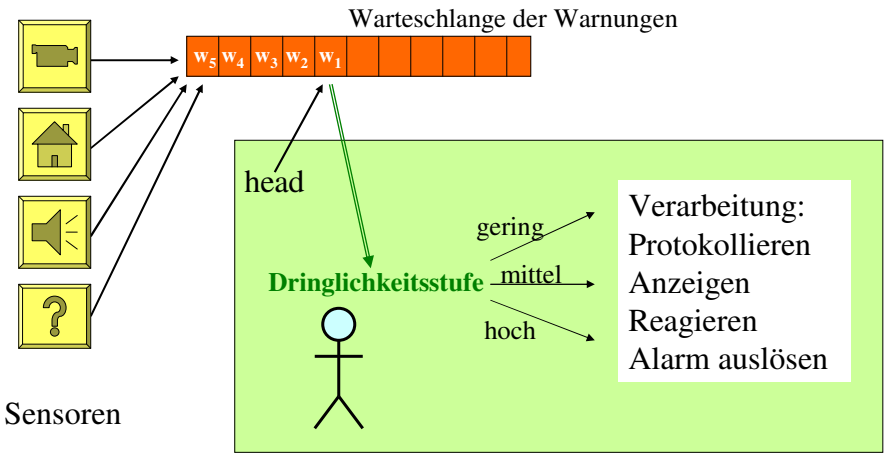
Gegeben seien zwei Pakete, die wir im Folgenden benutzen:

package Kalender is ...
Hier wird ein Typ Zeit bereitgestellt (Jahr, Monat, Tag, Stunde, Minute, Sekunde, Hunderstel-Sekunde), und eine parameterlose Funktion Uhrzeit vom Ergebnistyp Zeit.

package Personalverwaltung is ...
Hier wird ein Typ Person bereitgestellt sowie eine parameterlose Funktion Aufsichtführender vom Ergebnistyp Person.



## Warnsystemskizze



```
package Warnsystem is
type Dringlichkeit is (gering, mittel, hoch);
type Warnung (D: Dringlichkeit) is
   record  Ankunftszeit: Kalender.Zeit;
           Nachricht: String;
           case D is
           when gering => null;
           when mittel | hoch =>
               Verantwortlicher: Personalverwaltung.Person;
               case D is
               when hoch => Alarm_ausgeloest: Kalender.Zeit;
               when others => null;
               end case;
           end case;
   end record;
```



```
-- Fortsetzung von package Warnsystem is

type Anzeigegeraet is (Drucker, Bildschirm, Wandanzeige);
procedure Anzeigen (W: in Warnung; AG: in Anzeigegeraet);
procedure Mitprotokollieren (W: in Warnung);
procedure Alarm_Ausloesen (W: in out Warnung);
procedure Reagieren (W: in out Warnung);
end Warnsystem;
```

Eine Prozedur soll beispielhaft implementiert werden. Wir betrachten hier "Reagieren", die für jede Dringlichkeitsstufe anders arbeitet; sie ist im "package body Warnsystem is ..." zu definieren:

```
procedure Reagieren (W: in out Warnung) is
begin
   W.Ankunftszeit := Kalender.Uhrzeit;
    Mitprotokollieren (W);
   Anzeigen (W, Drucker);
   case W.D is
        when gering => null;
        when mittel | hoch =>
            W.Verantwortlicher:=Personalverwaltung.Aufsichtführender;
            Anzeigen(W, Bildschirm);
            case W.D is
            when hoch =>  Anzeigen (W, Wandanzeige);
                          Alarm_Ausloesen (W);
            when others => null;
            end case;
   end case;
end Reagieren;
```



Nachteile:

Unübersichtlich.

Schwierig zu korrigieren

und schwierig zu warten; jede Korrektur oder Erweiterung erfordert vermutlich eine komplette Neuübersetzung.

Beispiel: Man möchte eine neue Dringlichkeit "Notfall" höchster Dringlichkeit einführen.

Neuer Ansatz in Ada mit tagged Typen.

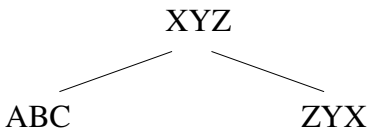
## 2.1.5: Warnsystem, objektorientierte Darstellung

*Vorbemerkung:* tagged Typen in Ada, siehe 1.4.5.1. Schema:

type XYZ is tagged record ... end record;

type ABC is new XYZ with record ... end record;

type ZYX is new XYZ with null record;






All dies kann offen oder geheim stattfinden:

type Vor is tagged record ... end record;
type Nach is new Vor with record ... end record; *oder*
type Nach is new XYZ with private; *und später steht dann:*
private type Nach is new Vor with record ... end record;

type Vor is tagged private; *und später steht dann:*
private type Vor is tagged record ... end record;

*Die abgeleiteten Typen aus* Vor *können wiederum offen oder geheim deklariert werden usw.*

Variablen in Ada müssen einen festen Datentyp haben:

```
type Vor is tagged record H: Integer; end record;
type Nach is new Vor with record K: Float; end record;
type Aehnlich is new Vor with null record;
X: Vor := 17;  Y: Nach := (50, 33.6);  Z: Aehnlich := 8;
begin  X := Y;        -- verboten wegen strenger Typisierung in Ada
       X := Vor(Y);       -- erlaubt, "Konversion" (=Projektion)
       Y := (X with 415.37);  -- erlaubt, Erweitern
       Z := X;        -- verboten wegen strenger Typisierung in Ada
       Z := (X with null record);    -- erlaubt, Erweitern
       X := Vor(Z);       -- erlaubt, "Konversion" (=Projektion)
       ...            -- stets alle hinzukommenden Komponenten auflisten!
end;
```

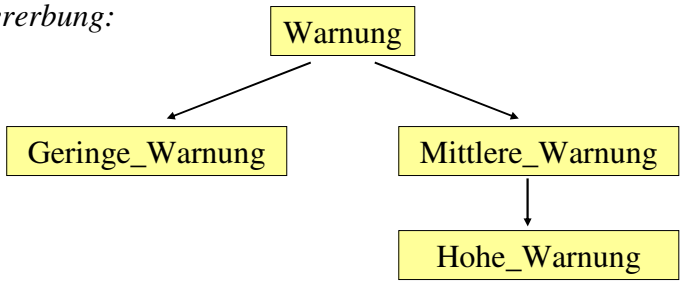


```
package Neues_Warnsystem is
type Anzeigegeraet is (Drucker, Bildschirm, Wandanzeige);
type Warnung is tagged
   record  Ankunftszeit: Kalender.Zeit;
           Nachricht: String;
   end record;
procedure Anzeigen (W: in Warnung; AG: in Anzeigegeraet);
procedure Mitprotokollieren (W: in Warnung);
procedure Reagieren (W: in out Warnung);

type Geringe_Warnung is new Warnung with null record;
procedure Reagieren (G: in out Geringe_Warnung);

type Mittlere_Warnung is new Warnung with
   record  Verantwortlicher: Personalverwaltung.Person;
   end record;
procedure Reagieren (M: in out Mittlere_Warnung);
```

-- Fortsetzung von *package Neues*_Warnsystem *is*

```
type Hohe_Warnung is new Mittlere_Warnung with
   record  Alarm_ausgeloest: Kalender.Zeit;
   end record;
procedure Reagieren (H: in out Hohe_Warnung);
procedure Alarm_Ausloesen (H: in out Hohe_Warnung);
end Neues_Warnsystem;
```

*Vererbung:*






```
package body Neues_Warnsystem is

procedure Reagieren (W: in out Warnung) is
begin  W.Ankunftszeit := Kalender.Uhrzeit;
       Mitprotokollieren (W);
end Reagieren;

procedure Reagieren (G: in out Geringe_Warnung) is
begin  Reagieren (Warnung(G));
       Anzeigen (Warnung(M), Drucker);
end Reagieren;

procedure Reagieren (M: in out Mittlere_Warnung) is
begin  Reagieren (Warnung(M));
          M.Verantwortlicher:=Personalverwaltung.Aufsichtführend
       er;
       Anzeigen (Warnung(M), Bildschirm);
end Reagieren;
```

```
procedure Reagieren (H: in out Hohe_Warnung) is
begin  Reagieren (Mittlere_Warnung(H));
       Anzeigen (Warnung(H), Wandanzeige);
       Alarm_Ausloesen (H);
end Reagieren;

procedure Anzeigen (W: in Warnung; AG: in Anzeigegeraet)
       is separate;

procedure Mitprotokollieren (W: in Warnung) is separate;

procedure Alarm_Ausloesen (H: in out Hohe_Warnung)
       is separate;
...
end Neues_Warnsystem;
```



Will man nun eine neue Dringlichkeitsstufe "Notfall" einführen, so kann man zum Beispiel im package "Neues_Warnsystem" einen Typ mit Prozeduren

```
type Notfall is new Hohe_Warnung with
    record Rettungsdienst_alarmiert: Kalender.Zeit;
    end record;
procedure Rettungsdienst_rufen (N: Notfall);
procedure Reagieren (N: in out Notfall);
```

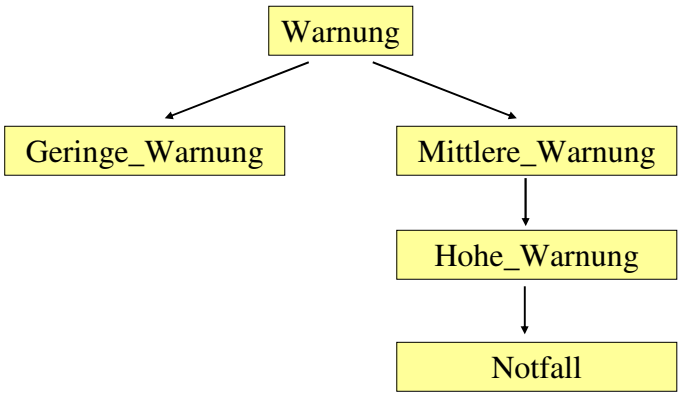
hinzufügen.

Oder man kann ein neues Paket einführen, das auf dem bisherigen package "Neues_Warnsystem" aufbaut:

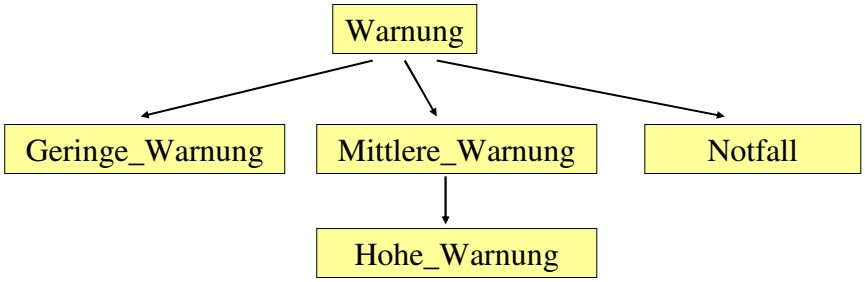
```
with Neues_Warnsystem;
package Aktuelles_Warnsystem is
type Notfall is new Neues_Warnsystem.Hohe_Warnung
   with private;
procedure Rettungsdienst_rufen (N: in Notfall);
procedure Reagieren (N: in out Notfall);
procedure Anzeigen (N: in Notfall;
   AG: Neues_Warnsystem.Anzeigegeraet);
   -- diese Prozedur Anzeigen muss nun neu formuliert werden,
   -- indem z.B. die Anzeige blinkt oder farbig dargestellt wird.
private
   type Notfall is new Neues_Warnsystem.Hohe_Warnung
   with record Rettungsdienst_alarmiert: Kalender.Zeit;
        end record;
end Aktuelles_Warnsystem;
```



*Neue Vererbungshierarchie:*



Man kann das Paket "Aktuelles_Warnsystem" auch direkt auf dem Paket "Warnsystem" aufsetzen, wie es im Buch von Nagl geschieht. Dann kann man folgende Vererbungshierarchie implementieren:






Nun haben wir die Objekte einschließlich Vererbung dargestellt. Wir möchten jedoch eine einheitliche Verarbeitung haben, d.h., es soll für jede eingehende Warnung (egal welcher Dringlichkeitsstufe) die zugehörige Prozedur "Reagieren" ausgeführt werden.

Hierfür muss es einen Datentyp geben, der die (disjunkte) Vereinigung aller Datentypen ist, die aus einem Typ "T" durch Vererbung abgeleitet werden können.

In Ada wird dieser Typ beschrieben durch T'Class.
In unserem Beispiel ist also mit den Typen Warnung, Geringe_Warnung, Mittlere_Warnung, Hohe_Warnung und Notfall zugleich der Typ Warnung'Class als Vereinigung dieser Typen definiert.

Somit können wir nun eine einheitliche Prozedur einführen:

procedure Warnungsbehandlung (X: in out Warnung'Class) is begin ... Reagieren (X); ... end;

Diese Prozedur können wir überall dort deklarieren, wo der Typ "Warnung" und die zu jedem hieraus abgeleiteten Typ gehörigen Prozeduren "Reagieren" sichtbar sind.
Zur Laufzeit wird für jede eintreffende Warnung Y

```
Warnungsbehandlung(Y);
```

aufgerufen. Der "tag" der Warnung legt den Typ von Y fest. Zur Laufzeit wird dann die zugehörige Prozedur "Reagieren" ermittelt und mit Y ausgeführt (dynamisches Binden).



Das Warnsystem wird nun wie folgt realisiert:

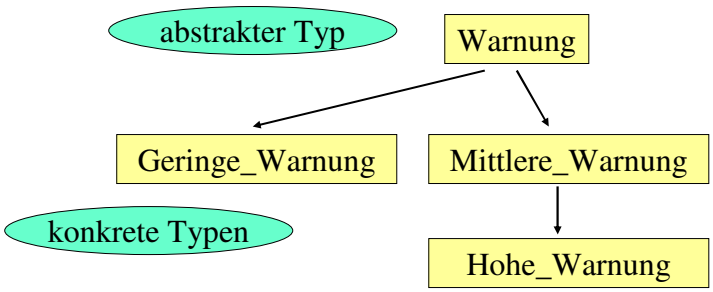
Sofern die Sensoren Signale empfangen, senden sie eine Warnung an das Rechnersystem, wobei in der Warnung mindestens zwei Komponenten gefüllt sind: (der String) Nachricht und der "tag", der den Datentyp angibt.

Diese werden automatisch in eine Warteschlange in der Halde eingetragen; das System fragt diese Schlange (Datentyp queue über dem Typ Warnung'Class) ständig auf neue Einträge ab. Auf das erste Element zeigt eine Zeigervariable "head" vom Typ

```
type Ref_Warnung is access Warnung'Class;
... head: Ref_Warnung; Y: Warnung'Class; ...
while ... loop ...;
  if not isemptyqueue then Y := head;
     Warnungsbehandlung(Y); head := head.next; ... end if; ...
```

Eine andere Möglichkeit, zur Laufzeit die Objekte der Variablen an die richtigen Prozeduren usw. zu binden, verwendet so genannte abtrakte Typen, vgl. 1.4.5.2 (dies hat nichts mit abstrakten Datentypen zu tun!).

Man kann hier eine "Wurzel" der Objekthierarchie festlegen zusammen mit Operationen, die zunächst "abstrakt" bleiben und erst bei der Definition der konkreten Typen ausgefüllt werden. In der Syntax wurde dieser Fall bereits durch basic_declaration ::= abstract_subprogram_declaration in 1.4.1.7 berücksichtigt. Wir erhalten die Vererbung:






```
package Basisalarmsystem is
type Warnen is abstract tagged null record;
procedure Reagieren (Y: in out Warnen) is abstract;
end Basisalalarmsystem;
```

```
with Kalender; with Personalverwaltung;
with Basisalarmsystem;
package W_System is
type Anzeigegeraet is (Drucker, Bildschirm, Wandanzeige);
type Geringe_Warnung is new Basisalarmsystem.Warnen with
        record  Ankunftszeit: Kalender.Zeit;
        Nachricht: String;
        end record;
procedure Anzeigen
        (W: in Geringe_Warnung; AG: in Anzeigegeraet);
procedure Mitprotokollieren (W: in Geringe_Warnung);
procedure Reagieren (W: in out Geringe_Warnung);
```

```
type Mittlere_Warnung is new Basisalarmsystem.Warnen with
record  Ankunftszeit: Kalender.Zeit;
        Nachricht: String;
        Verantwortlicher: Personalverwaltung.Person;
end record;
procedure Reagieren (M: in out Mittlere_Warnung);
procedure Mitprotokollieren (W: in Mittlere_Warnung);

type Hohe_Warnung is new Mittlere_Warnung with
record  Alarm_ausgeloest: Kalender.Zeit;
end record;
procedure Reagieren (H: in out Hohe_Warnung);
procedure Alarm_Ausloesen (H: in out Hohe_Warnung);
end W_System;
```

In Reagieren kann nun ein Aufruf *Mitprotokollieren(Mittlere_Warnung(H));* aufttreten. Überlegen Sie sich, wie man die Prozedur "Anzeigen" mitnutzen kann. Formulieren Sie den "package body" zu W_System aus.