Abschnitt 6

Halde, Polymorphie, Vererbung

Beispiel: Umsortieren

Diskriminanten

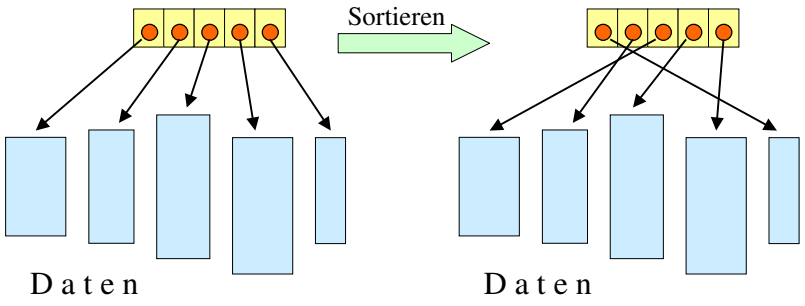
Polymorphie, Generizität

Vererbung, tagged types

Beispiel: Sortieren ohne Umordnen der Daten

Üblicherweise liegen die eigentlichen Daten, die man sortieren möchte, irgendwo im Speicher. Der Zugriff auf sie erfolgt über Zeiger, die in der Regel sehr viel weniger Speicher benötigen.

Konsequenterweise braucht man dann auch nicht die Daten zu sortieren; es genügt, die Zeiger anzuordnen.



Wir benötigen also ein Feld von N Zeigern auf die N Daten.
Wir wählen als Schlüsseltyp hier willkürlich die ganzen Zahlen.

```
type Daten;
type Zugriff is access Daten;

type Daten is record
        Schlüssel: Integer;
        Inhalt: ...;              -- beliebiger Typ
end record;
...
S: array (1..N) of Zugriff;
```

Aufgabe: Das Schlüsselfeld S und die Daten seien bereits gegeben. Ordne das Feld S so um, dass für alle i, j mit i < j gilt: S(i).Schlüssel ≤ S(j).Schlüssel.

In folgendem Programmstück implementieren wir Bubblesort (i ist vom Indextyp des Feldes S; weiter: Boolean; die Hilfsvariable H ist vom Typ Zugriff):

```
i := S'First; weiter := true;
while (i in S'First..S'Pred(Last)) and weiter loop
    weiter := false;
    for j in reverse Succ(i)..S'Last loop
        if S(Pred(j)).Schlüssel > S(j).Schlüssel then
          H:=S(Pred(j)); S(Pred(j)):=S(j); S(j):=H;
          weiter := true; end if;
    end loop;
    i := Succ(i);
end loop;
```

An der Tafel vorgetragen:

- Variationen bei String bzw. arrays mit unspezifizierten Grenzen.
- Diskriminanten bei Verbundtypen (record). Beispiel:

```
type Puffer (Laenge: natural := 80) is record
        position: 1..laenge := 1;
        Inhalt: String (1..laenge);
end record;
P1: Puffer;
P2: Puffer (30);
P3: Puffer (laenge => 1000);
```

Den gesetzte Parameter kann man später nicht mehr ändern. Braucht man mehr Platz, so muss man ein neues größeres Element erzeugen und das alte an dessen Anfang umspeichern.

Beispiel bei varianten Records, wobei eine Variable im Prinzip auf Zahlen wie auch auf Zeichen verweisen kann:

```
type Zwei (Wahl: Boolean) is record
   case Wahl is
   when true => Zahl: Integer;
   when false => Z: Character; end case;
end record;
type Ref_Zwei is access Zwei;
X: Ref_Zwei;   ...
X := new Zwei (Wahl => true);
X.Zahl := 27;    -- dagegen ist  X.Z := 'A';  hier nicht möglich;
X := new Zwei (Wahl => false);
X.Z := Character'Val(65);
...
```

Polymorphie (aus dem Griechischen: *Vielgestaltigkeit*) ist z.B. aus der Biologie, aus den Grundlagen der Mathematik und aus der Linguistik bekannt.

In der Informatik ist Polymorphie ein wichtiges Prinzip. Mit ihr lässt sich die Wiederverwendung von Programmteilen (engl.: reuse) erleichtern. Die Polymorphie äußert sich durch folgende Maßnahmen:

Möglichst lange den konkreten Datentyp von Variablen offen lassen. Man denke an unspezifizierte Feldgrenzen bei arrays.

Möglichst lange die konkrete Realisierung offen lassen. Z.B. Spezifikations- und Implementierungsteil trennen.

Parametrisierung von Paketen und Unterprogrammen, um diese für möglichst viele Anwendungen einsetzen zu können.

In der Programmierung spricht man in folgenden Fällen von Polymorphie:

1. Mehrfachverwendung von ***Namen*** (hierzu zählt das Überladen in Ada).

2. ***Variablen*** können je nach aktueller Umgebung Elemente verschiedener Datentypen bezeichnen oder als Werte besitzen.

Ein Beispiel hierfür sind in Ada Untertypen. Betrachte

```
subtype natural is integer range 0..integer'Last;
subtype positive is natural range 1..natural'Last;
X: integer; Y: natural; Z: positive;
```

Y und Z werden hierbei auch als Variable des Typs integer angesehen, sind also polymorph.

```
type Vektor is array (integer range <>) of float;

procedure etwas (X, Y: in out Vektor) is
Summe: Vektor (X'Range);
begin  ...  for J in Y'Range loop Summe(J) := ...  end loop;
          ...
end etwas;
```

In diesem Beispiel sind X und Y Variablen eines Datentyps, dessen Größe (in Form der Indexgrenzen) unbekannt ist. Der Datentyp von X und Y steht also erst nach der Auswertung der zugehörigen aktuellen Parameter fest. Der Datentyp von Summe ist erst nach dem Aufruf bekannt und er kann bzgl. des Indexbereichs jedes Mal anders sein. Die Variablen X, Y und Summe kann man daher als polymorph auffassen.

3. Parametrisierung von ***Typen*** eines Programms. Betrachte zum Beispiel Polynome:

Ein Polynom ist eine Abbildung p: M → M der Form
$p(x) = a_n \cdot x^n + a_{n-1} \cdot x^{n-1} + ... + a_1 \cdot x + a_0$,
wobei auf M eine Addition und eine Multiplikation definiert sein muss. Mathematisch ist M in der Regel ein "Ring", z.B. $\mathbb{N}_0$, $\mathbb{Z}$, $\mathbb{Q}$ oder $\mathbb{R}$. Weiterhin muss $a_n \neq 0$ sein (Ausnahme: das Null-Polynom $p(x) = 0$ für alle x); n heißt der Grad von p.

Der Typ "Polynom" hat also die Parameter n und Ring (dies soll der zu M gehörende Datentyp sein): array (0..n) of Ring.

Schreibweise (mit subtype exponent is integer range 0..max):

```
type polynom (n: exponent; type ring) is
        record A: array (0..n) of ring; end record;
```

(In Ada muss der Typ anders eingesetzt werden, s.u.)

Diesen parametrisierten Typ kann man dann im Programm benutzen, um konkrete Variablen zu deklarieren:

P : polynom(n =>4, ring => float); oder mit Initialisierung:
Q : polynom(4, integer) := (2, 0, 6, 5, 3);
Dies bezeichnet das Polynom (A(0) sei 2 usw.):
$q(x) = 3 \cdot x^4 + 5 \cdot x^3 + 6 \cdot x^2 + 2.$

In Ada sind parametrisierte Typen erlaubt, allerdings sind hierbei keine Typen als Parameter vorgesehen:

```
type polynom (n: maxexp) is
record A: array (0..n) of integer; end record;
Q: polynom := (4, (2, 0, 6, 5, 3));
```

Typen als Parameter in anderen Typen kann man in Ada durch parametrisierte Pakete realisieren (Generizität).

4. Ein ***Unterprogramm*** oder ein ***Paket*** heißt polymorph, wenn mindestens eine Spezifikation eines Datentyps oder wenn der Datentyp eines formalen Parameters oder des Ergebnisses polymorph ist. Bekannte Beispiele für solche Polymorphie sind Sortierverfahren, Integrale usw.

Eine spezielle Form von vielfacher Verwendbarkeit von Ada-Programmeinheiten (vor allem Unterprogramme und Moduln) bezeichnet man als Generizität. Man legt hierbei ein Schema an, das erst bei der Deklaration zugehöriger Variablen durch konkrete Datentypen und Konstanten ausgefüllt werden muss. Das Schlüsselwort lautet in Ada "generic", hinter dem die Typen und ggf. zusätzliche Funktionen auflistet sind; anschließend folgt dann die jeweilige Programmeinheit.

Generizität in Ada

Unter Generizität versteht man in Ada die Parametrisierung von Programmeinheiten mit Datentypen, Unterprogrammen und Moduln. Beispiel:

Die Datenstruktur "Liste" lässt sich unabhängig von den in ihr verwalteten Daten erklären und sollte in einer allgemeinen Form als Bibliotheks-Modul vorhanden sein. Ein Programm sollte seine konkreten Listen zu Beginn hieraus ableiten, indem der Modul importiert ("with") und durch den aktuell benötigten Datentyp konkretisiert wird.

Wir erläutern dies an Ada-Beispielen und beginnen mit dem Vertauschen zweier Inhalte.

Der Spezifikationsteil und der Implementierungsteil müssen bei generischen Einheiten in Ada prinzipiell getrennt angegeben werden. Der variabel gehaltene Spezifikationsbereich wird mit "generic" eingeleitet:

```
generic
    type element is private;
procedure Tausch (A, B: in out element);

procedure Tausch (A, B: in out element) is
H: element;
begin  H:=A; A:=B; B:=H;  end Tausch;
```

Bei folgender Deklaration

X, Y: integer; C, D: Boolean;

kann man die generische Prozedur Tausch dann wie folgt konkretisieren:

```
procedure BoolTausch is new Tausch (Boolean);
procedure IntTausch is new Tausch (integer);
... IntTausch(X, Y); ... BoolTausch(C, D); ...
```

Wegen des Überladens kann man in Ada auch schreiben:

```
procedure Austausch is new Tausch (Boolean);
procedure Austausch is new Tausch (integer);
... Austausch (X, Y); ... Austausch (C, D); ...
```

Wir betrachten nun ein anderes Beispiel in Ada mit einem Datentyp "item", auf dem eine Ordnung ">" definiert sein möge. Wir sortieren ein Feld aus max Komponenten mittels Bubble Sort:

Sortieren (leider kein ganz korrektes Ada, siehe später):

```
generic
   max: Positive;
   type T is private;
   type VektorT is array (1..max) of T;
procedure SORT (A: in out VektorT);

procedure SORT (A: in out VektorT) is
procedure Austausch is new Tausch (T);
weiter: Boolean := true;
begin
while weiter loop  weiter := false;
   for I in 1..max-1 loop
   if A(I) > A(I+1)
      then weiter := true; Austausch(A(I), A(I+1)); end if;
   end loop; end loop;
end SORT;
```

Nicht erlaubt

Wir machen hieraus eine neue Prozedur.

Dieser Operator ist bekannt zu machen

In Ada darf der Wert von Max nicht innerhalb derselben Deklaration bereits verwendet werden und der Operator " > " muss bekannt gegeben werden. Letzteres geschieht durch Angabe mit "with". Nun kann man aber den Typ VektorT nicht mehr außen bekannt machen. Daher muss man statt der Prozedur eine andere Einheit, das "Paket" (package) wählen. So erhalten wir:

```
generic
   Max: Positive;
   type T is private;
   with function ">"(L,R:T) return Boolean;
package Sortpaket is
   type VektorT is array (1 .. Max) of T;
   procedure Sort (A: in out VektorT);
end Sortpaket;
```



```
package body Sortpaket is
  generic   type Element is private;
  procedure Tausch (A, B: in out Element);
  procedure Tausch (A, B: in out Element) is
    H: Element; begin  H:=A; A:=B; B:=H; end Tausch;
  procedure Sort (A: in out VektorT) is
    Weiter: Boolean := true;
    procedure Austausch is new Tausch(T);
  begin
    while Weiter loop
      Weiter := false;
      for I in 1..Max-1 loop
        if A(I)>A(I+1) then Weiter := true; Austausch(A(I), A(I+1));
        end if;
      end loop;
    end loop;
  end Sort;
end Sortpaket;
```

Verwendung im weiteren Verlauf des Programms:

```
        -- Im Deklarationsteil:
with Sortpaket; with Ada.Integer_Text_IO;
package SortpaketInt is new Sortpaket(500, integer, ">");
R: SortpaketInt.VektorT;
        -- Im Anweisungsteil:
...                         -- Fülle das Feld R mit ganzen Zahlen
SortpaketInt.Sort(R);
...
```

Hinweis: Wenn Sie im package SortpaketInt die Operation ">" durch "<" ersetzen, so wird absteigend sortiert!

Hinweise zur Syntax für generische Unterprogramme und Pakete:

```
generic_declaration ::=  generic_subprogram_declaration |
                         generic_package_declaration

generic_subprogram_declaration ::=
        generic_formal_part  subprogram_specification ;

generic_package_declaration ::=
        generic_formal_part  package_specification ;

generic_formal_part ::=
    generic {generic_formal_parameter_declaration | use_clause}
```

```
generic_formal_parameter_declaration ::=
    formal_object_declaration |
    formal_type_declaration   |
    formal_subprogram_declaration |
    formal_package_declaration

formal_subprogram_declaration ::=
    with subprogram_specification [is subprogram_default] ;

formal_package_declaration ::=
    with package defining_identifier is new
    generic_package_name  formal_package_actual_part ;

formal_type_declaration ::=
    type defining_identifier[discriminant_part] is
                                    formal_type_definition ;

formal_package_actual_part ::= (<>) | [generic_actual_part]
```

usw.

Vererbung

Die Welt ist voller Hierarchien und Beziehungen. In einer Modellierung soll sich dies widerspiegeln.
Im Bereich der Datentypen (und der Objekte, siehe später) verwendet man hierfür die Vererbung. Die Idee ist einfach:

1. Hat man einen Datentyp deklariert, so kann man durch Hinzufügen weiterer Komponenten hieraus weitere Datentypen ableiten ("Spezialisierung", Unterdatentypen).
2. Liegen mehrere Datentypen vor, die gewisse Komponenten gemeinsam haben, so kann man diese Gemeinsamkeiten als einen eigenen Datentyp herausziehen ("Generalisierung", Oberdatentyp) und die gegebenen Datentypen zu gemeinsamen Unterdatentypen des neuen Oberdatentyps machen.

Nun braucht man nur noch Sprachelemente, um die Erweiterung oder Einschränkung von Typen zu beschreiben.

Typerweiterung oder Spezialisierung: In Ada muss ein Datentyp, der später erweitert werden soll, mit dem Zusatz "tagged" deklariert werden (engl. tag = Etikett, Zusatz). Wir betrachten einen Typ zur Beschreibung einiger Attribute eines Fahrzeugs:

```
type Fahrzeug is tagged record
        Hersteller: array (1..30) of character;
        Höchstgeschwindigkeit: positive;
        Neupreis: delta 0.01 range 0.0 .. 50_000_000.0;
end record;
```


Will man nun Attribute für einen Bus zusammenfassen, so reicht es, obige Deklaration einfach zu erweitern mittels

```
type Bus is new Fahrzeug with record
        Sitzplätze, Stehplätze: positive;
        Wendekreis: delta 0.001 range 0.0 .. 40.0;
        Achsenzahl: positive;
end record;
```

Dies entspricht der folgenden Deklaration:

```
type Bus is record
        Sitzplätze, Stehplätze: positive;
        Wendekreis: delta 0.001 range 0.0 .. 40.0;
        Achsenzahl: positive;
        Hersteller: array (1..30) of character;
        Höchstgeschwindigkeit: positive;
        Neupreis: delta 0.01 range 0.0 .. 50_000_000.0;
end record;
```

Sprechweisen:

Man sagt, "Bus" ist ein aus "Fahrzeug" abgeleiteter Typ.

Man sagt, die Komponenten "Hersteller", "Neupreis" und "Höchstgeschwindigkeit" wurden vom Typ "Fahrzeug" auf oder an den Typ "Bus" vererbt.

Man nennt diesen Vorgang, Eigenschaften an andere Einheiten weiterzureichen, "Vererbung" (engl. inheritance).

Man sagt auch, "Bus" sei eine "Spezialisierung" oder "Erweiterung" oder "Untertyp" (aber nicht im Sinne der Untertypen von Ada) von oder zu "Fahrzeug".

Man sagt, "Fahrzeug" ist "Generalisierung" oder Obertyp von oder zu "Bus".

Man nennt die Obertypen auch (direkte) "Eltern", die Untertypen (direkte) "Kinder". Setzt man dies transitiv fort, so spricht man von "Vorfahren" bzw. "Nachkommen".

Es werden also alle Deklarationen des Typs Fahrzeug auf den Typ Bus vererbt.

(Hinweis: Diese Vererbung wird bei Paketen und Objekten, wo nicht nur Erweiterungen, sondern auch Umdefinitionen erfolgen, besonders nützlich.)

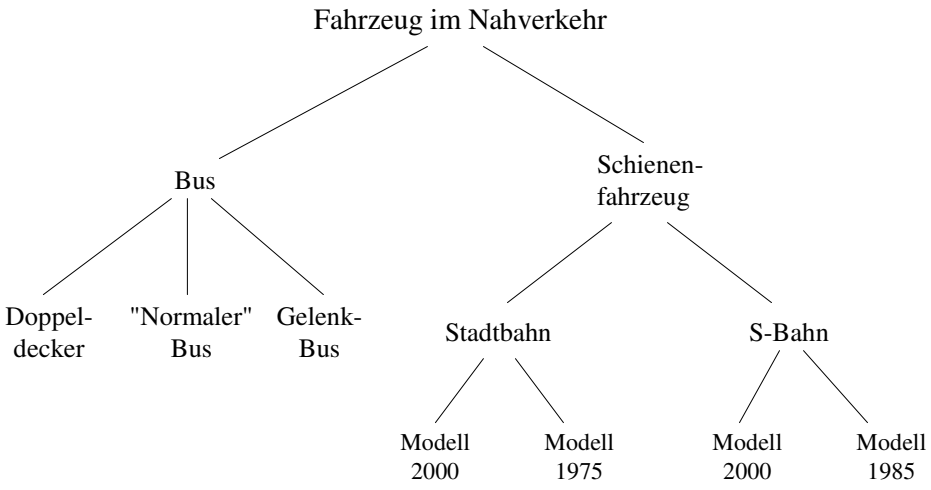
Das Gleiche kann man für eine S-Bahn tun:

```
type SBahn is new Fahrzeug with record
        Sitzplätze, Stehplätze: positive;
        Waggonlänge: delta 0.001 range 0.0 .. 400.0;
        Achsenzahl: positive;
end record;
```

Hieraus kann man weitere Typen ableiten (in Ada muss man explizit angeben, wenn man nichts hinzufügen will):

```
type Normaler_Bus is new Bus with null record;

type Doppeldecker is new Bus with record
        Höhe: float;
end record;

type Gelenkbus is new Bus with record
        Länge: float;
end record;

type Stadtbahn is new Fahrzeug with record
        Sitzplätze, Stehplätze: positive;
        AnzahlKartenEntwerter: positive;
end record;
```

Hinweise: Die Eigenschaft "tagged" vererbt sich automatisch auf alle abgeleiteten Typen. Ableitbare Typen erkennt man an "tagged" oder "with" in ihrer Definition.



Aufbau einer Vererbungs-Hierarchie

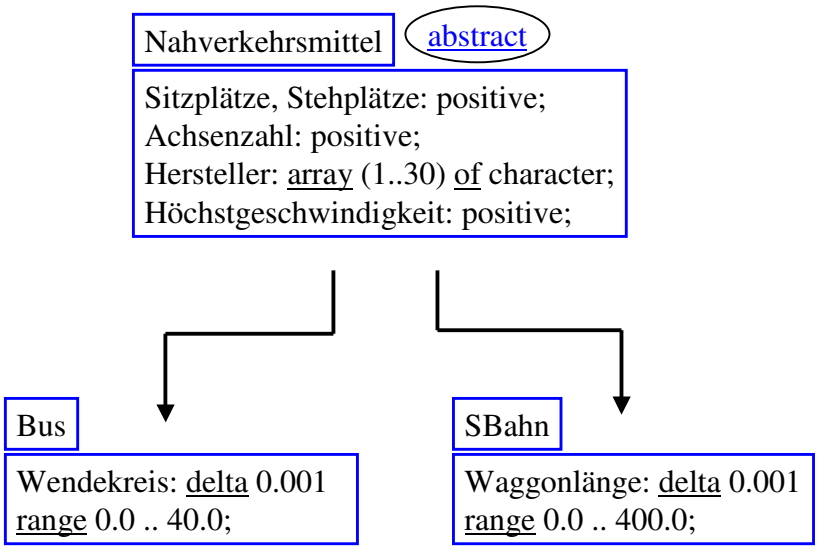
Zusammenfassen oder Generalisierung: Wir formulieren dies ebenfalls sofort in Ada. Gegeben seien:

```
type Bus is record
        Sitzplätze, Stehplätze: positive;
        Wendekreis: delta 0.001 range 0.0 .. 40.0;
        Achsenzahl: positive;
        Hersteller: array (1..30) of character;
        Höchstgeschwindigkeit: positive;
end record;

type SBahn is new Fahrzeug with record
        Hersteller: array (1..30) of character;
        Höchstgeschwindigkeit: positive;
        Sitzplätze, Stehplätze: positive;
        Waggonlänge: delta 0.001 range 0.0 .. 400.0;
        Achsenzahl: positive;
end record;
```

In einen Obertyp herausziehen

Wir deklarieren also einen neuen Typ "Nahverkehrsmittel", aus dem wir die Typen Bus und SBahn ableiten können:

```
type Nahverkehrsmittel is abstract tagged record
        Sitzplätze, Stehplätze: positive;
        Achsenzahl: positive;
        Hersteller: array (1..30) of character;
        Höchstgeschwindigkeit: positive;
end record;

type Bus is new Nahverkehrsmittel with record
        Wendekreis: delta 0.001 range 0.0 .. 40.0;
end record;

type SBahn is new Nahverkehrsmittel with record
        Waggonlänge: delta 0.001 range 0.0 .. 400.0;
end record;
```



Hier tritt das Sprachelement "abstract" auf. Es bedeutet, dass man einen Datentyp deklariert hat, den man nur für die Vererbung verwendet, zu dem man aber keine Variablen oder formalen Parameter deklarieren darf. Bei dem "abstract"-Typ "Nahverkehrsmittel" sind also Deklarationen folgender Art *verboten*

X: Nahverkehrsmittel;
function F (A: in Nahverkehrsmittel) return Boolean; ...

In unserem Beispiel ist der Zusatz "abstract" nicht notwendig gewesen, doch sollte man ihn stets verwenden, wenn man den jeweiligen Typ nur in der "Vererbungs-Hierarchie" einsetzt.

Umdefinitionen (redefinition)

Bei der Vererbung kann man vererbte Komponenten neu definieren. Die vererbten Komponenten sind dann wegen der Sichtbarkeitsregel automatisch ausgeblendet (können aber bei allgemeinen Strukturen wie packages über Qualifizierung mit Punkt-Notation dennoch angesprochen werden). Beispiel:

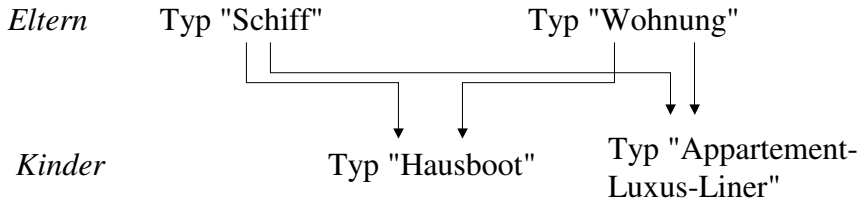
```
type wenig is 1..20;

type Kleinbus is new Bus with record
        Sitzplätze: wenig;
end record;
```

Die ererbte Komponente "Sitzplätze" wird durch diese neue Komponente "Sitzplätze" im Type Kleinbus überschrieben (engl.: overridden).

Anmerkung: Mehrfachvererbung

Nach dem bisherigen Konzept kann ein Datentyp höchstens einen (direkten) Obertyp besitzen. Man spricht von Einfach-Vererbung (single inheritance).

Manchmal sollen aber Eigenschaften mehrerer Datentypen an einen Datentyp weitergereicht werden. Diesen Vorgang nennt man Mehrfach-Vererbung (multiple inheritance).



In Ada ist Mehrfachvererbung nicht zugelassen. Die Gründe hierfür liegen zum einen in Problemen der Implementierung, zum anderen in der Fehleranfälligkeit bei der Verwendung mehrfacher Ableitungen, vor allem wenn sie erst zur Laufzeit nachvollzogen werden können.

Die Programmierer sind hier selbst verantwortlich, wenn durch Vererbung gleicher Namen, die aber verschiedene Typen in den unterschiedlichen Eltern bezeichnen, Namenskonflikte oder andere Mehrdeutigkeiten entstehen. In Java ist Mehrfachvererbung erlaubt.
Generell sollte man die Mehrfachvererbung sparsam und "sehr kontrolliert" verwenden.