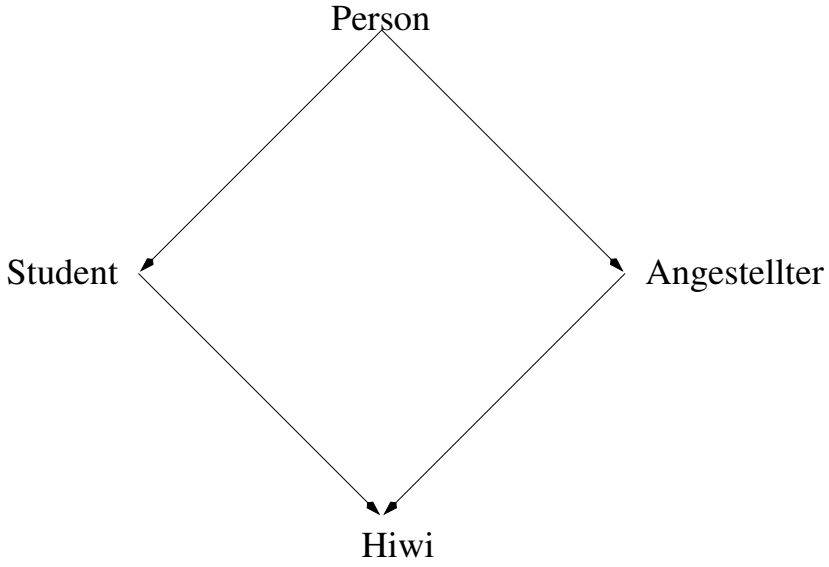
Kapitel 4

Mehrfachvererbung

Beispiel 1:

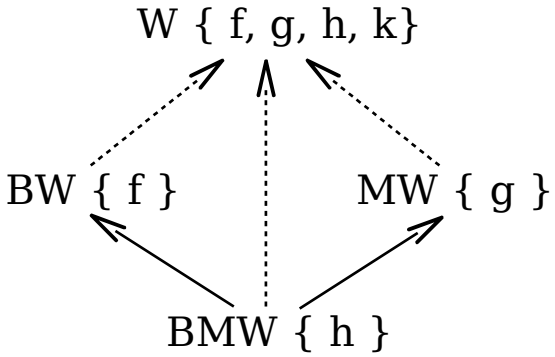


typische „Diamant“ Struktur

Beispiel 2: Fenster mit Rand und Menü; Verteilung von Rand und Menü auf 2 Unterklassen

W = Window, BW = Border Window (Fenster mit Umrandung), MW = Menu Window (Fenster mit Menü), BMW = Bordered Menu Window, f() = Berechnung der Fensterfläche

```
class W {
  virtual f();
  virtual g();
  virtual h();
  virtual k();
};
class MW : virtual W {
  g();
};
class BW : virtual W {
  f();
};
class BMW : BW, MW,
  virtual W {
  h();
}
```



Aufruf `BMW* pbmw; MW* pmw = pbmw; pmw->f();`

ruft `BW::f()` ! (→ Static Lookup)

Dieses Verhalten ist sinnvoll: Wenn man bei einem Fenster mit Rand und Menü den Rand ignoriert (`pmw=pbmw`), muß die Flächenberechnung ihn trotzdem berücksichtigen!

4.1 Interface-Mehrfachvererbung

Interface:

- nur Methodensignaturen und Konstanten; keine Instanzvariablen, keine Objekte
- Interfaces können als Typen verwendet werden, aber konkrete Klassen müssen alle Interface-Methoden implementieren
- Interface-Vererbung (Subtyping) möglich. Beispiel:

```
interface A { static final int x=42;
        void f(Object x); }
interface B { int g(int x);}
interface C extends B { double h(String s);}
class U extends O implements A, C {
  void f(Object x) {...Rumpf...};
  int g(int x) {...Rumpf...};
  double h(String s) {...Rumpf...};
  ... eigene Methoden/Instanzvariablen...}
```

abstrakte Klasse:

- manche Methodenrümpfe können fehlen (abstrakte Methoden)
- Instanzvariablen möglich
- Unterklassen müssen abstrakte Methoden implementieren

Java kennt sowohl abstrakte Klassen als auch Interfaces

Mehrfachvererbung für Klassen gibt es *nicht*

Jedoch: Mehrfachvererbung für Interfaces

Beispiel:

```
interface I1 { ... Konstantendef. ... Signatur ... }
interface I2 { ... }
interface I3 extends  I1, I2
interface I4 { ... }
class A extends B implements I3, I4 { ... }
```

Vorteil: effizient; Klassenhierarchie kann durch Interface-Vererbung ausgedrückt werden

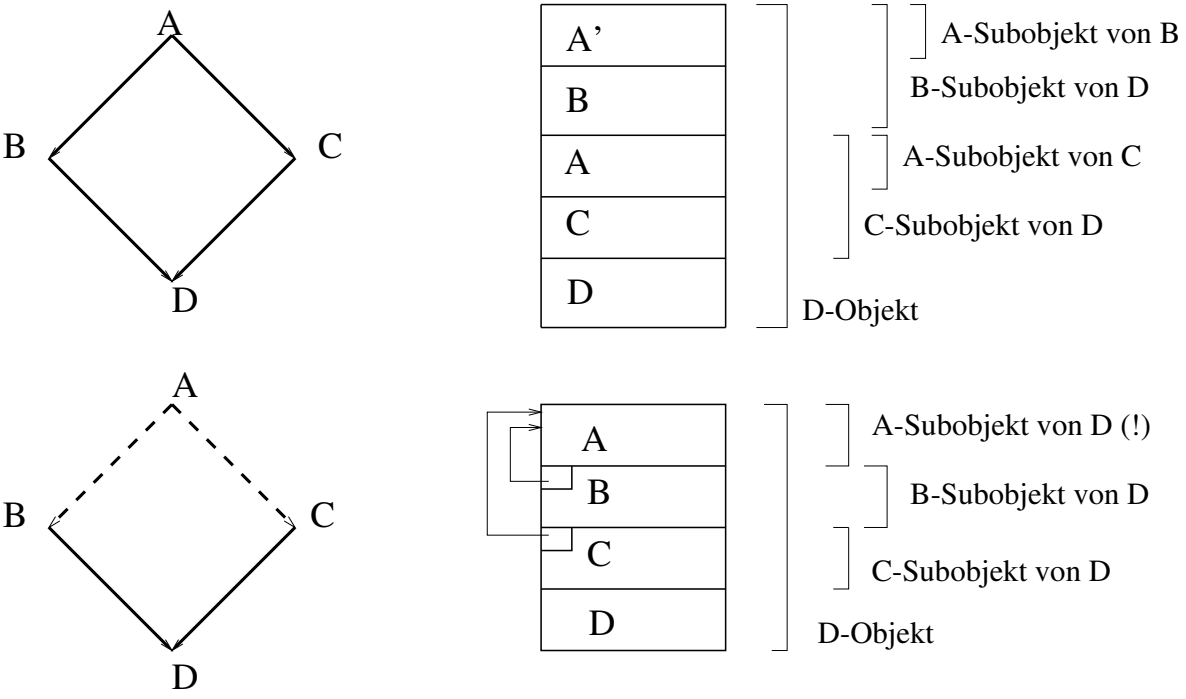
Nachteil: echte Implementierungen können nur von einer Oberklasse geerbt werden, alle Interfacemethoden müssen selbst implementiert werden

4.2 Multiple Subobjekte in C++

C++ kennt virtuelle und nichtvirtuelle Vererbung

- nichtvirtuelle Vererbung (default, durchgezogene Linie im Klassendiagramm): Unterklassenobjekt enthält Oberklassenobjekt physikalisch
- virtuelle Vererbung (gestrichelte Linie): Unterklassenobjekt enthält Pointer auf Oberklassenobjekt

nichtvirtuelle Mehrfachvererbung führt zu *Mehrfachkopien* desselben Subobjektes

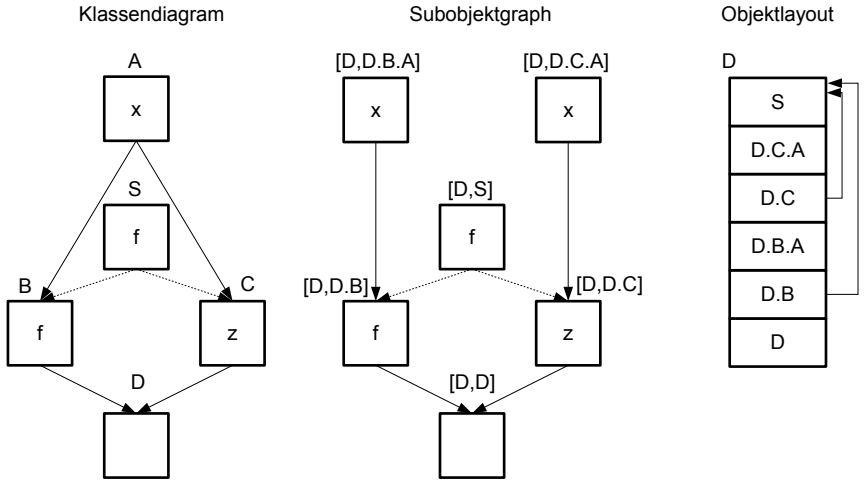


4.3 Subobjektgraphen

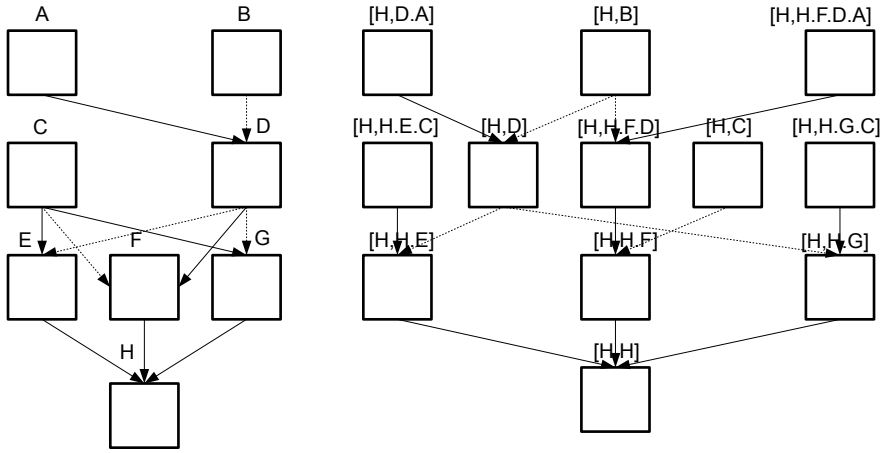
Formalismus zur Beschreibung von Objektlayouts (Rossie/Friedman 1997)

Subobjekte können nur durch vollständige “Vererbungspfade” eindeutig identifiziert werden: $[C, C \cdot B \cdot A]$ bedeutet “das $C \cdot B \cdot A$ -Subobjekt eines C-Objektes”

Beispiel 1:



Beispiel 2:



4.4 Static Lookup

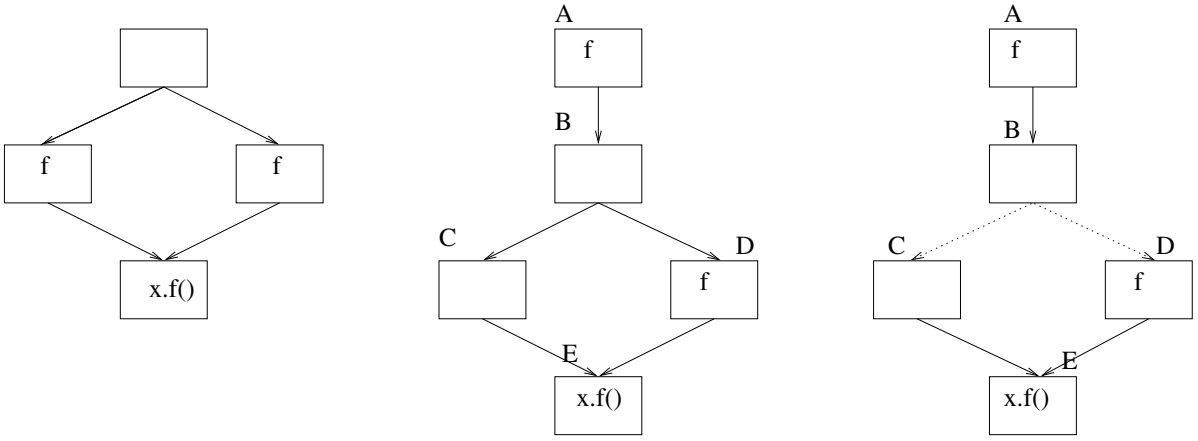
Gegeben: Klasse C in Hierarchie $\mathcal{H}$,
Membername m

Gesucht: Subobjekt von C, in dem m deklariert ist:

$$lookup(C, m) = [C, C \cdot \ldots]$$

Falls keine Mehrfachvererbung: einfach (Aufwärtssuche in der Hierarchie)

in C++: bitter! Beispiele:

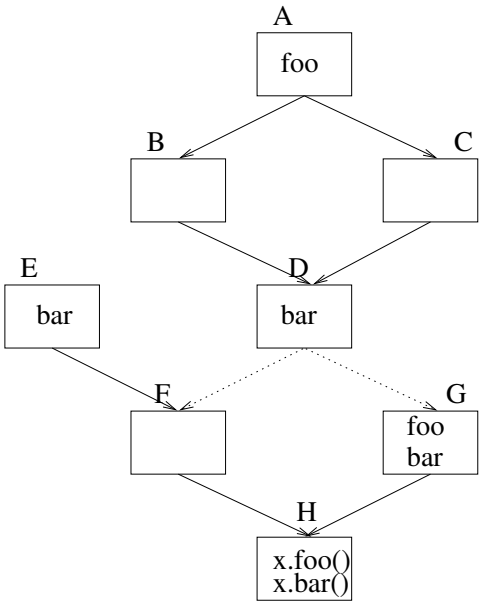


Fall 1: Konflikt, da f mehrdeutig

Fall 2: Konflikt, da mehrfaches A-Subobjekt wg. nichtvirtueller Mehrfachvererbung ⇒ analog zu 1. !

Fall 3: kein Konflikt, da virtuelle Mehrfachvererbung: “A member name f in one subobject B dominates a member name f in subobject A if A is a base class subobject of B”
⇒ $lookup(E, f) = [E, E \cdot D]$

Beispiel 2. Klassenhierarchie:



$Defs(C,m)$ ist die Menge aller Subobjekte von C, die eine Definition von m enthält:

$$\begin{aligned} Defs(H,foo) &= \{[H,D\cdot B\cdot A],[H,D\cdot C\cdot A],[H,H\cdot G]\} \\ Defs(H,bar) &= \{[H,H\cdot F\cdot E],[H,D],[H,H\cdot G]\} \end{aligned}$$

Def. (*Dominanz*) $[C,\alpha] \sqsubseteq [C,\beta]$, wenn es im **Subobjektgraph** (Abwärts)pfad $[C,\beta]$ nach $[C,\alpha]$ gibt

Dominierendstes (größtes) Subobjekt $\sigma = max(A)$ einer Subobjekt-Menge A: dominiert alle $\sigma' \in A$. Falls nicht eindeutig: $\sigma = \bot$

.

$$\begin{aligned} \Rightarrow \quad lookup(C,m) &= max(Defs(C,m)) \\ lookup(H,foo) &= [H,H\cdot G] \\ lookup(H,bar) &= \bot \end{aligned}$$

Formale Definitionen:

A, B, C seien Klassen. Wir schreiben $A <_V B$ für virtuelle Vererbung, $A <_N B$ für nichtvirtuelle Vererbung, sowie $< \; = \; <_V \cup <_N$.

1. $[C, C]$ ist ein Subobjekt
2. Ist $[C, \alpha \cdot A]$ ein Subobjekt und $\exists B : \; A <_N B$, so ist $[C, \alpha \cdot A \cdot B]$ ein Subobjekt
3. Ist $[C, \alpha]$ ein Subobjekt und $\exists A, B : \; C <^* A, \; A <_V B$, so ist $[C, B]$ Subobjekt
4. $[C, \alpha] \sqsubseteq [C, \alpha \cdot A]$
5. $[C, \alpha \cdot A] \sqsubseteq [C, B]$ wenn $A <_V B$
6. $mdc([C, \alpha \cdot A]) = C$ (“most derived class”)
7. $ldc([C, \alpha \cdot A]) = A$ (“least derived class”)
8. $Def(C) = \{m \mid C \text{ enthält Definition von Member } m\}$
9. $Defs(C, m) = \{\sigma \sqsupseteq^* [C, C] \mid m \in Def(ldc(\sigma))\}$

$\sqsubseteq^*$ bezeichnet die transitive Hülle von $\sqsubseteq$

Damit lässt sich statischer Lookup auf Subobjekten wie folgt ausdrücken:

$$\text{lookup}(\sigma, m) \; = \; \max(\{\sigma' \mid \sigma' \sqsupseteq^* \sigma, m \in Def(ldc(\sigma'))\})$$

Hinweis. Aus "historischen" Gründen schreibt man max (das größte Subobjekt), mathematisch ist jedoch das min bezüglich $\sqsubseteq$ gemeint

Falls das "max" nicht eindeutig ist, ist das Ergebnis $= \bot$

Beispiele zu Hierarchie S. 53:

$$[H, H \cdot G] \sqsubseteq [H, D] \sqsubseteq^* [H, D \cdot C \cdot A]$$

$$mdc([H, D \cdot C \cdot A]) = H,\ ldc([H, D \cdot C \cdot A]) = A$$

$$\begin{aligned} \text{lookup}([H,H], foo) &= \max(\{[H, H \cdot G], [H, D \cdot C \cdot A], \\ &\qquad\qquad [H, D \cdot B \cdot A]\}) \\ &= [H, H \cdot G] \end{aligned}$$

$$\text{lookup}([H,H], bar) = \max(\{[H, H \cdot F \cdot E], [H, D], [H, H \cdot G]\}) = \bot$$

4.5 Dynamische Bindung bei Rossie/Friedmann

Dynamische Bindung auf Subobjekten:

$$\begin{aligned}\text{dynBind}(\sigma, m) \;=\; \max(\{\sigma' \mid \sigma' \sqsupseteq^* [mdc(\sigma), mdc(\sigma)],\\ m \in Def(ldc(\sigma'))\})\end{aligned}$$

Beispiele:

1. Hierarchie S. 53 sowie $D\ d = new\ H(); d.foo();$

   Der Cast $H \rightarrow D$ bewirkt, dass d das $[H, D]$ Subobjekt bezeichnet. Es ist $mdc([H, D]) = H$, wir suchen also alle $\sigma' \sqsupseteq^* [H, H]$, die foo enthalten. Also

   $$\begin{aligned}\text{dynBind}([H, D], foo) \;&=\; \max(\{[H, H \cdot G], [H, D \cdot C \cdot A],\\ &\qquad [H, D \cdot B \cdot A]\})\\ &=\; [H, H \cdot G]\end{aligned}$$

2. Hierarchie S. 50 (virtueller Fall), wobei $f()$ in A und C deklariert sei, und $B\ b = new\ D(); b.f();$

   $\Rightarrow \text{dynBind}([D, D \cdot B], f) = \max(\{[D, D \cdot C], [D, A]\}) = [D, D \cdot C]$

Den Zusammenhang zwischen statischer und dynamischer Bindung beschreibt das Lemma:

$$\begin{aligned}\text{dynBind}(\sigma, m) \;&=\; \max(\{\sigma' \mid \sigma' \sqsupseteq^* [mdc(\sigma), mdc(\sigma)],\\ &\qquad m \in Def(ldc(\sigma'))\}\\ &=\; \text{lookup}([mdc(\sigma), mdc(\sigma)], m)\end{aligned}$$

d.h. dynamische Bindung ist wie statischer Lookup angewendet auf den dynamischen Typ

4.6 Rossie/Friedmann und C++

Rossie/Friedmann weicht im Fall von Konflikten durch nichtvirtuelle Mehrfachvererbung von C++ ab, denn C++ berücksichtigt in solchen Situationen zusätzlich den statischen Typ! Damit wird dynamische Bindung auch vom statischen Typ abhängig!

Im letzten Beispiel nichtvirtuell ergibt sich in C++

$$\begin{aligned}\mathrm{dynBind}([D, D \cdot B], f) &= \max(\{[D, D \cdot C], [D, D \cdot C \cdot A], \\ &\qquad [D, D \cdot B \cdot A]\}) \\ &= [D, D \cdot B \cdot A]\end{aligned}$$

obwohl bei R/F $\perp$ rauskommt! Grund: b hat statischen Typ B. Hätte b statischen Typ D, wäre der Aufruf aber auch in C++ mehrdeutig.

Stroustrup behauptet, dass er sich das alles gut überlegt hat ...