



Einführung in die Programmierung

Prof. Dr. Bertrand Meyer

Lektion 14: Mehrfachvererbung

Abstraktionen kombinieren



Gegeben sind die Klassen

➤ TRAIN_CAR, RESTAURANT

Wie würden Sie eine Klasse DINER implementieren?



Separate Abstraktionen kombinieren:

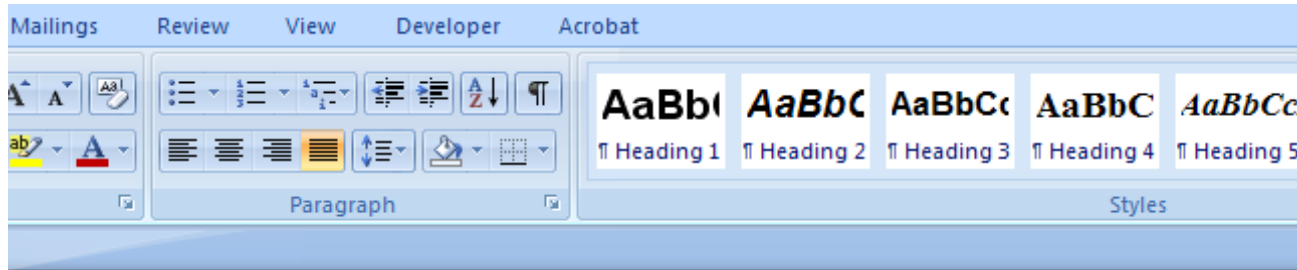
- Restaurant, Eisenbahnwagen
- Taschenrechner, Uhr
- Flugzeug, Vermögenswert
- Zuhause, Fahrzeug
- Tram, Bus

Vergessen Sie alles, was Sie gehört haben!

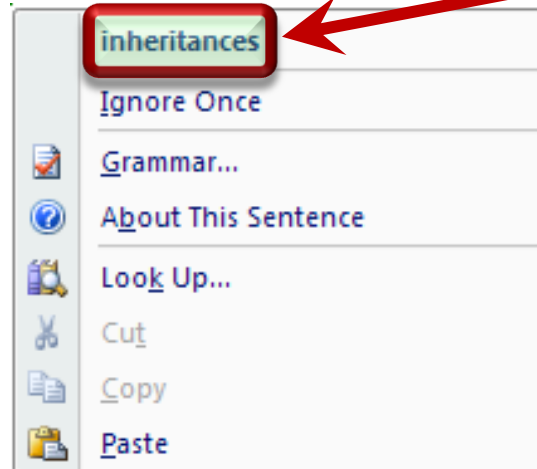
Mehrfachvererbung ist **nicht** das Werk des Teufels

Mehrfachvererbung schadet ihren Zähnen **nicht**

(Auch wenn Microsoft Word sie scheinbar nicht mag:

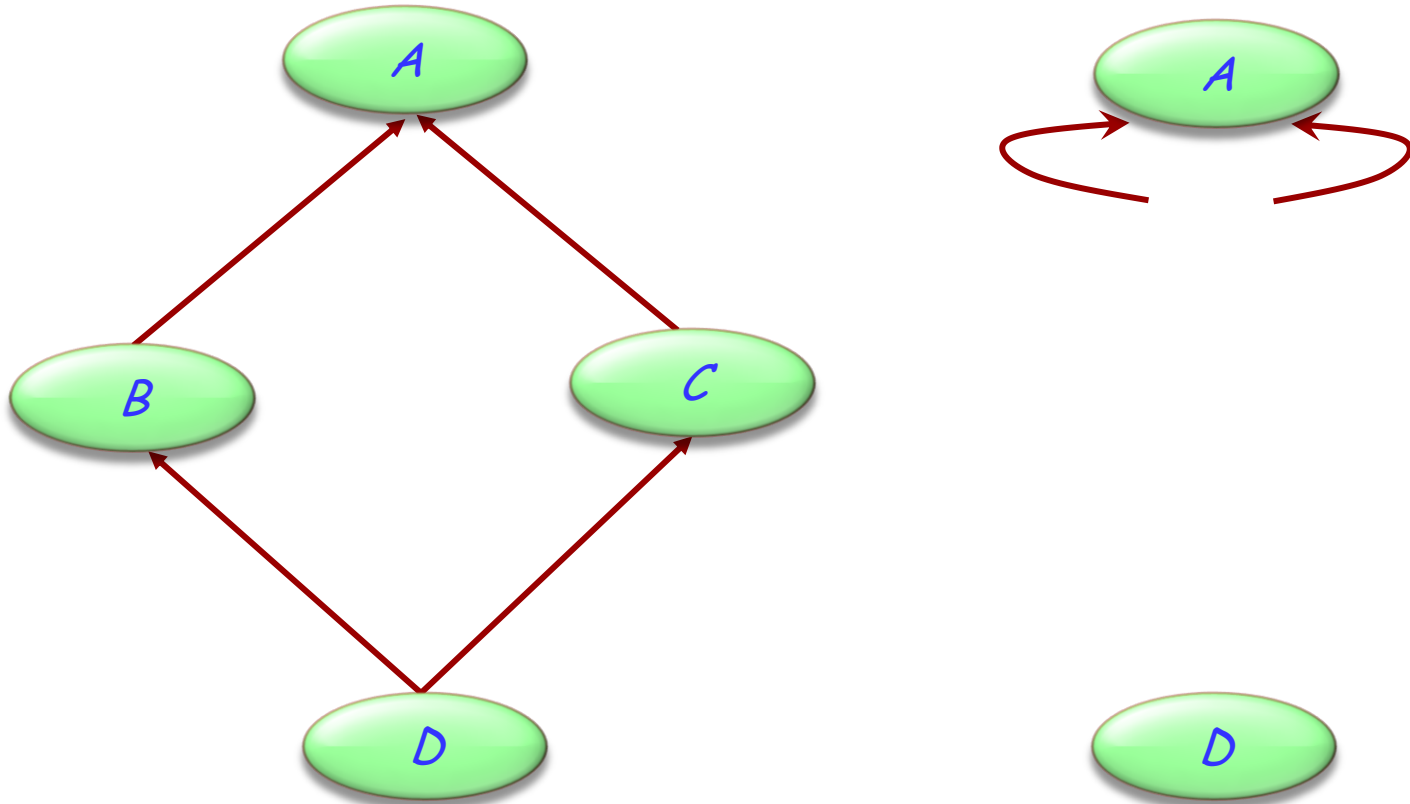


Object-oriented programming would become a mockery of itself if it had to renounce multiple inheritance.



)

Dies ist **wiederholte** Vererbung, nicht Mehrfachvererbung



Nicht der allgemeine Fall
(Obwohl es häufig vorkommt; warum?)

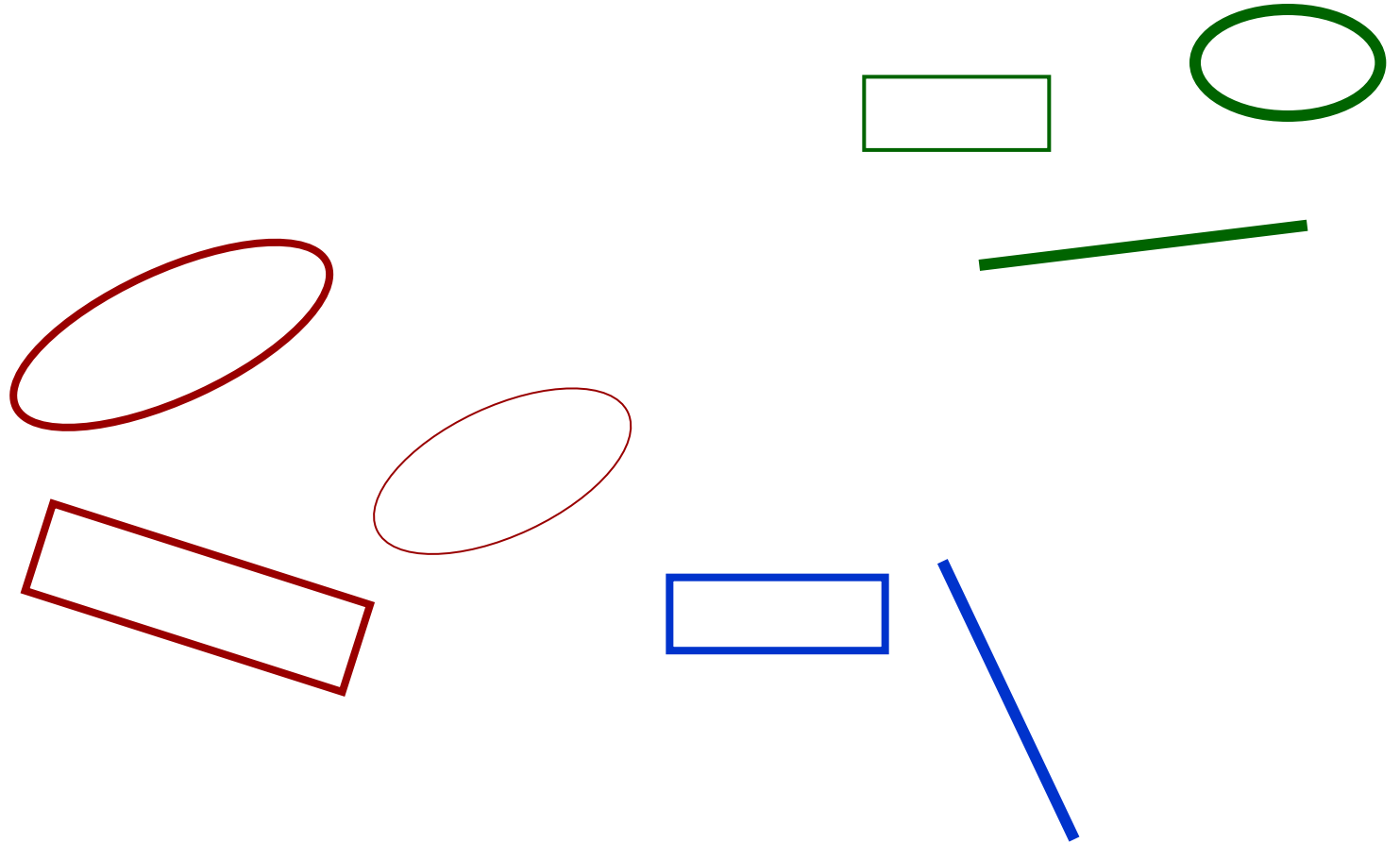


Dieser Teil der Vorlesung orientiert sich an Eiffel

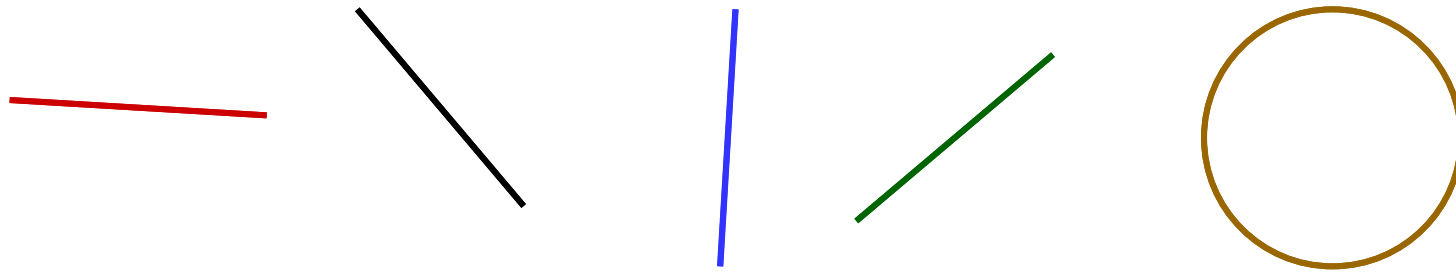
Java und C# Mechanismen (Einfachvererbung von Klassen, Mehrfachvererbung von Schnittstellen) werden aber auch behandelt

C++ hat unterstützt ebenfalls Mehrfachvererbung, aber ich werde nicht versuchen, diese zu beschreiben.

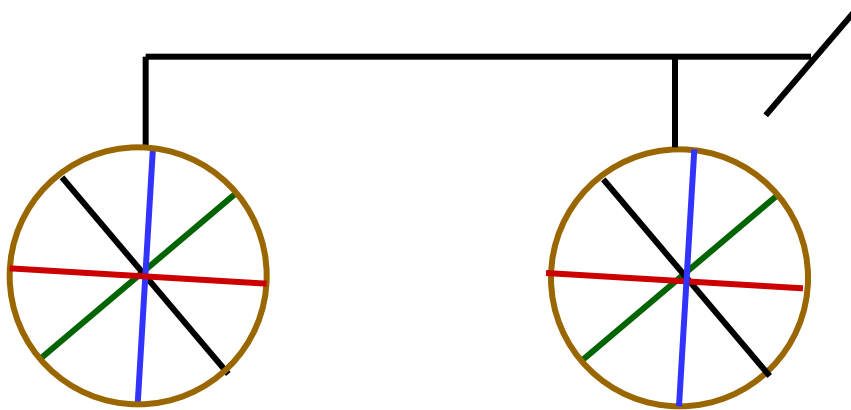
Zusammengesetzte Figuren



Mehrfachvererbung: Zusammengesetzte Figuren



Einfache Figuren



Eine zusammengesetzte Figur

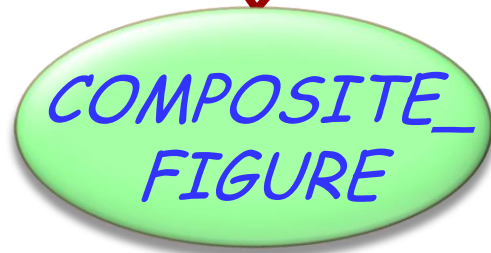
Den Begriff der zusammengesetzten Figur definieren



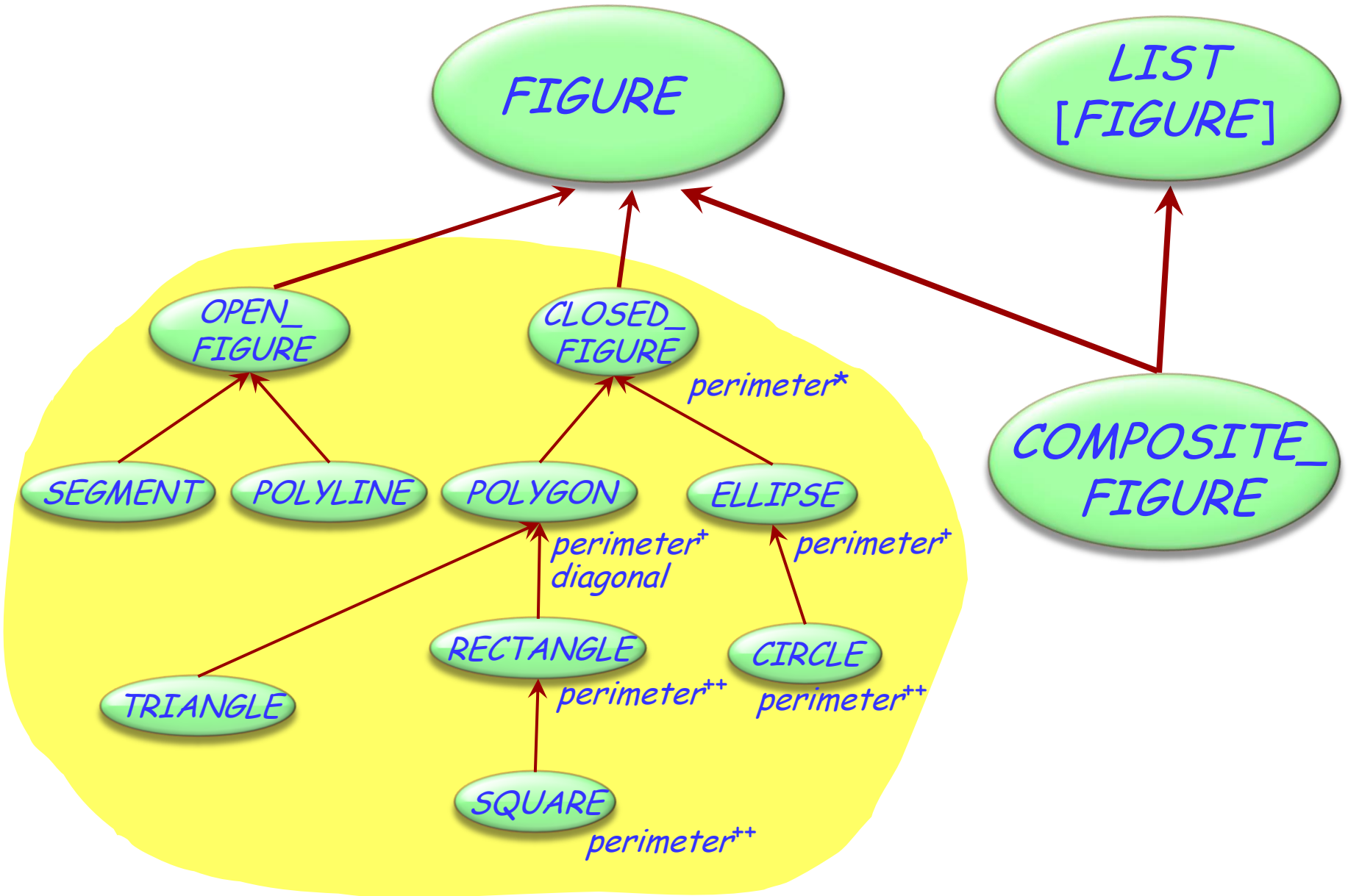
*center
display
hide
rotate
move
...*



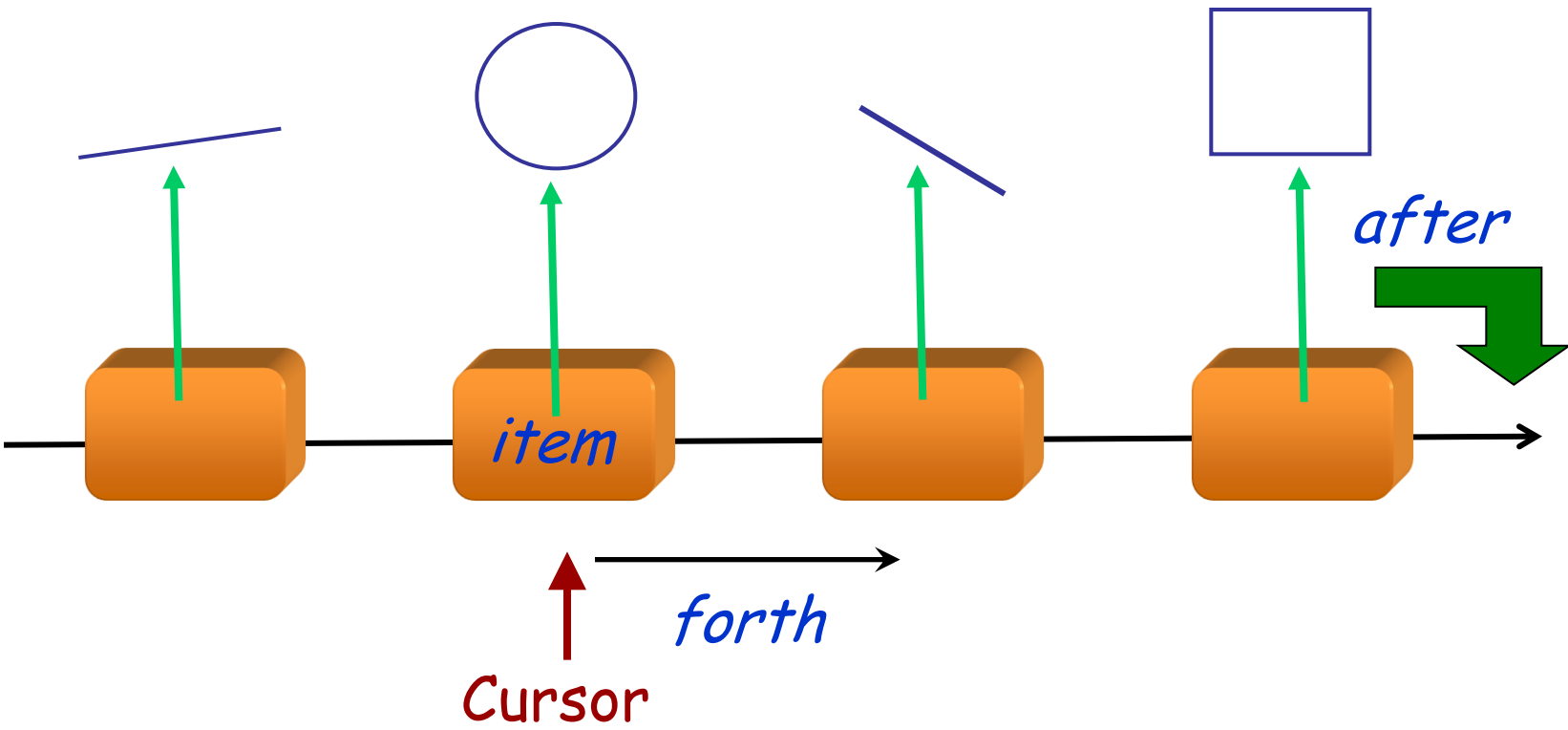
*count
put
remove
...*



In der allgemeinen Struktur



Eine zusammengesetzte Figur als Liste



Zusammengesetzte Figuren



```
class COMPOSITE_FIGURE inherit  
    FIGURE  
    LIST[FIGURE]
```

```
feature
```

```
    display
```

-- Jede einzelne Figur der Reihenfolge
-- nach anzeigen.

```
do
```

```
    from start until after loop
```

item.display

```
    forth
```

```
end
```

```
end
```

... Ähnlich für *move*, *rotate* etc. ...

```
end
```

Benötigt
dynamisches Binden

Eine Abstraktionsebene höher gehen

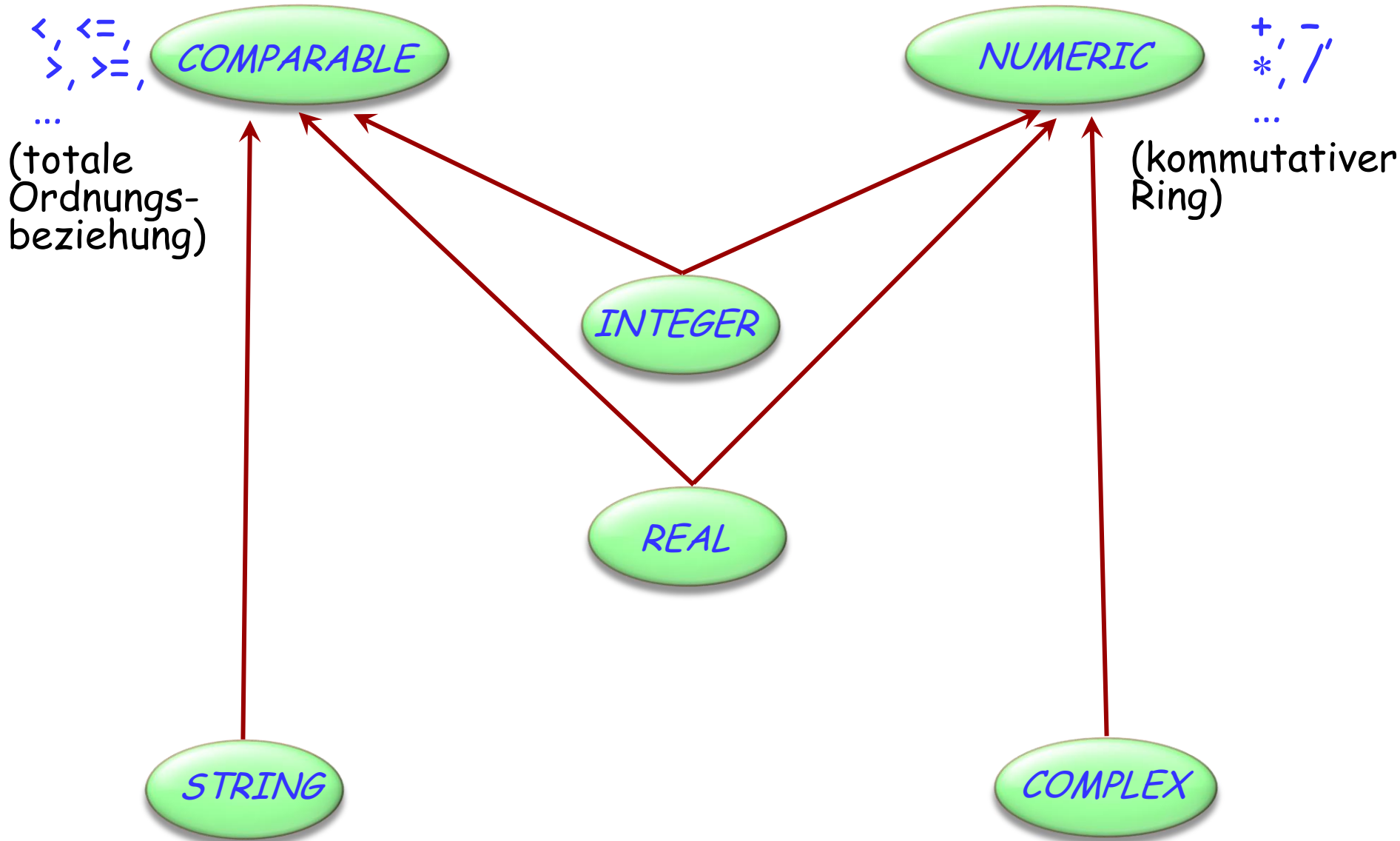


Eine einfachere Form der Prozeduren *display*, *move* etc. kann durch den Gebrauch von Iteratoren erreicht werden.

Benutzen Sie dafür **Agenten**.

Wir werden diese in ein paar Wochen behandeln (aber Sie dürfen das Kapitel gerne schon im Voraus lesen).

Mehrfachvererbung: Abstraktionen kombinieren





Keine Mehrfachvererbung für Klassen

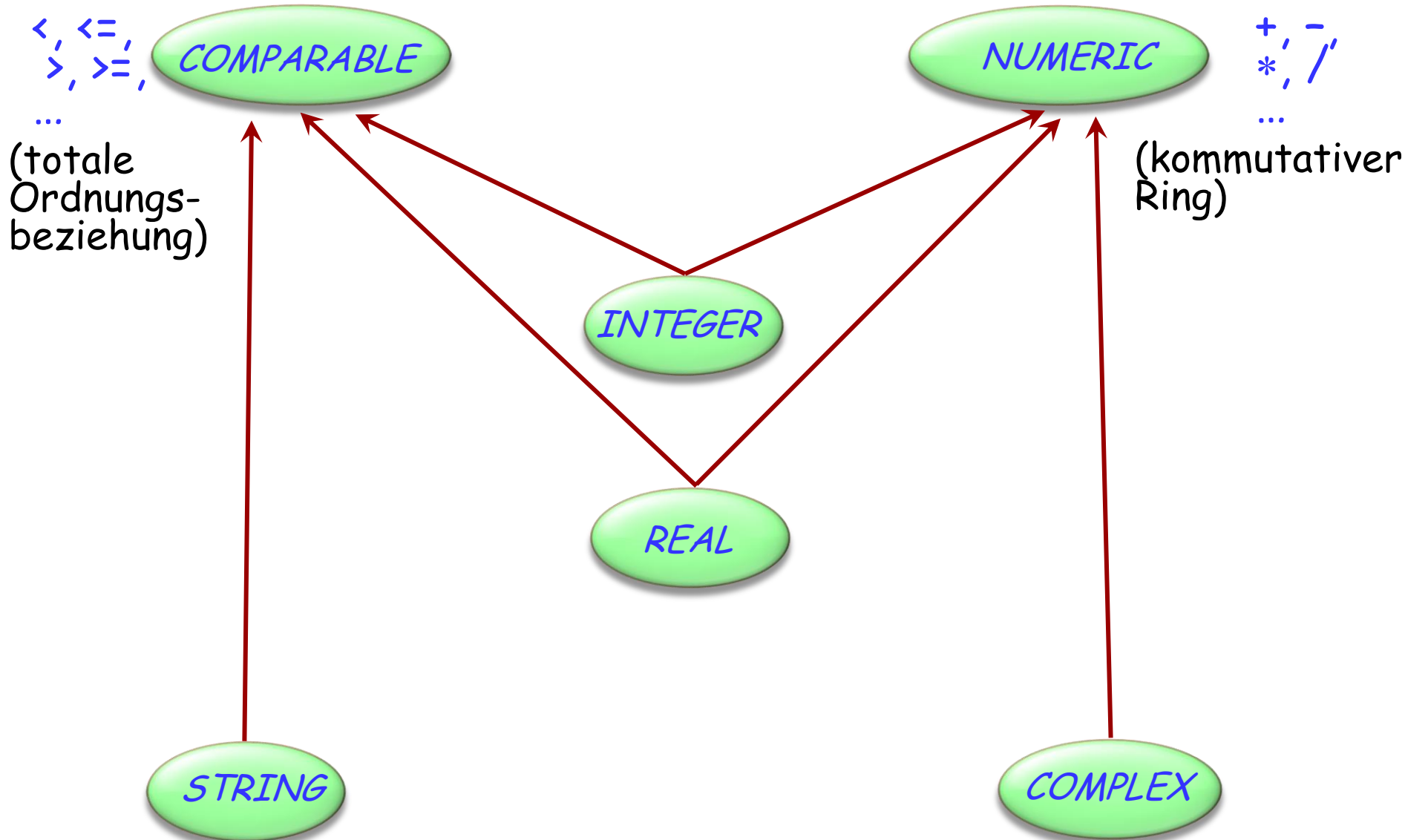
“Schnittstellen“: Nur Spezifikationen (keine Verträge)

- Ähnlich wie komplett aufgeschobene Klassen (ohne wirksame Features)

Eine Klasse kann:

- Von höchstens einer Klasse erben
- Von beliebig vielen Schnittstellen erben

Mehrfachvererbung: Abstraktionen kombinieren



Wie schreiben wir die Klasse *COMPARABLE*?



```
deferred class COMPARABLE[G] feature
```

```
  less alias "<" (x: COMPARABLE[G]): BOOLEAN
```

```
  deferred
```

```
  end
```

```
  less_equal alias "<=" (x: COMPARABLE[G]): BOOLEAN
```

```
  do
```

```
    Result := (Current < x or (Current = x))
```

```
  end
```

```
  greater alias ">" (x: COMPARABLE[G]): BOOLEAN
```

```
  do Result := (x < Current) end
```

```
  greater_equal alias ">=" (x: COMPARABLE[G]): BOOLEAN
```

```
  do Result := (x <= Current) end
```

```
end
```



Typisches Beispiel für ein *lückenhaftes Programm*.

Wir brauchen das volle Spektrum von vollständig abstrakten (aufgeschobenen) Klasse bis zu komplett implementierten Klassen.

Mehrfachvererbung hilft uns, Abstraktionen zu kombinieren.

Ein typisches Beispiel aus der Eiffel-Bibliothek



```
class ARRAYED_LIST [G] inherit  
    LIST[G]  
    ARRAY[G]
```

feature

... Implementiere *LIST*-Features mit *ARRAY*-Features ...

end

For example:

```
i_th(i: INTEGER): G  
    -- Element mit Index 'i'
```

do

```
Result := item (i)
```

end

Feature von *ARRAY*

Man könnte auch **Delegation** benutzen...



```
class ARRAYED_LIST [G] inherit  
  LIST[G]
```

feature

```
  rep: ARRAY[G]
```

... Implementiere *LIST*-Features mit *ARRAY*-
Features, auf *rep* angewendet...

end

For example:

```
  i_th(i: INTEGER): G
```

```
    -- Element mit Index 'i'.
```

```
  do
```

```
    Result := rep.item (i)
```

```
  end
```

Nicht-konforme Vererbung



class

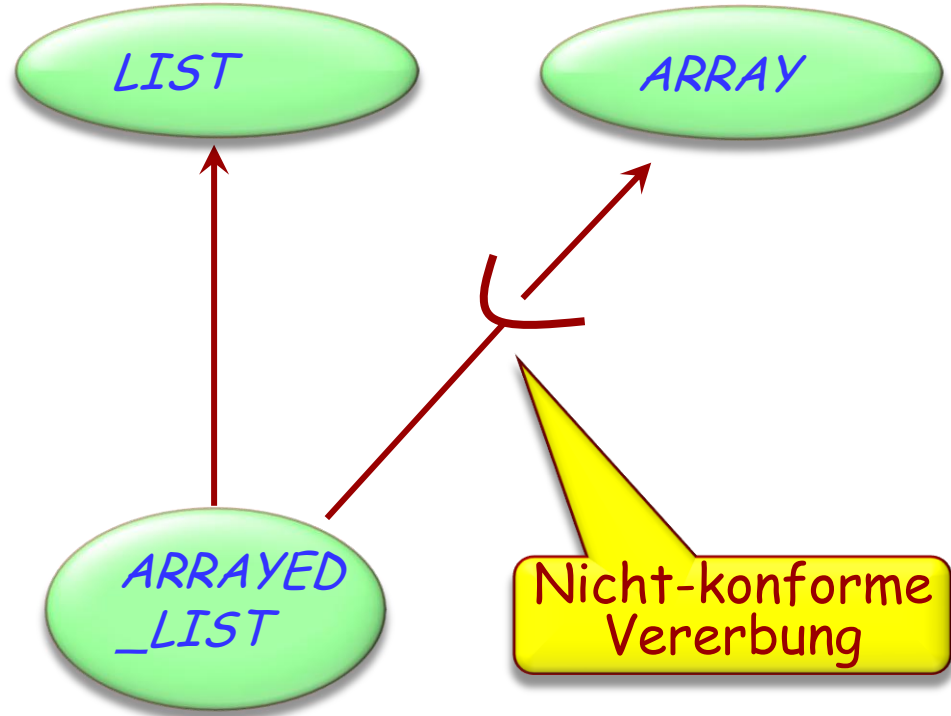
ARRAYED_LIST [G]

inherit

LIST [G]

inherit {*NONE*}

ARRAY [G]

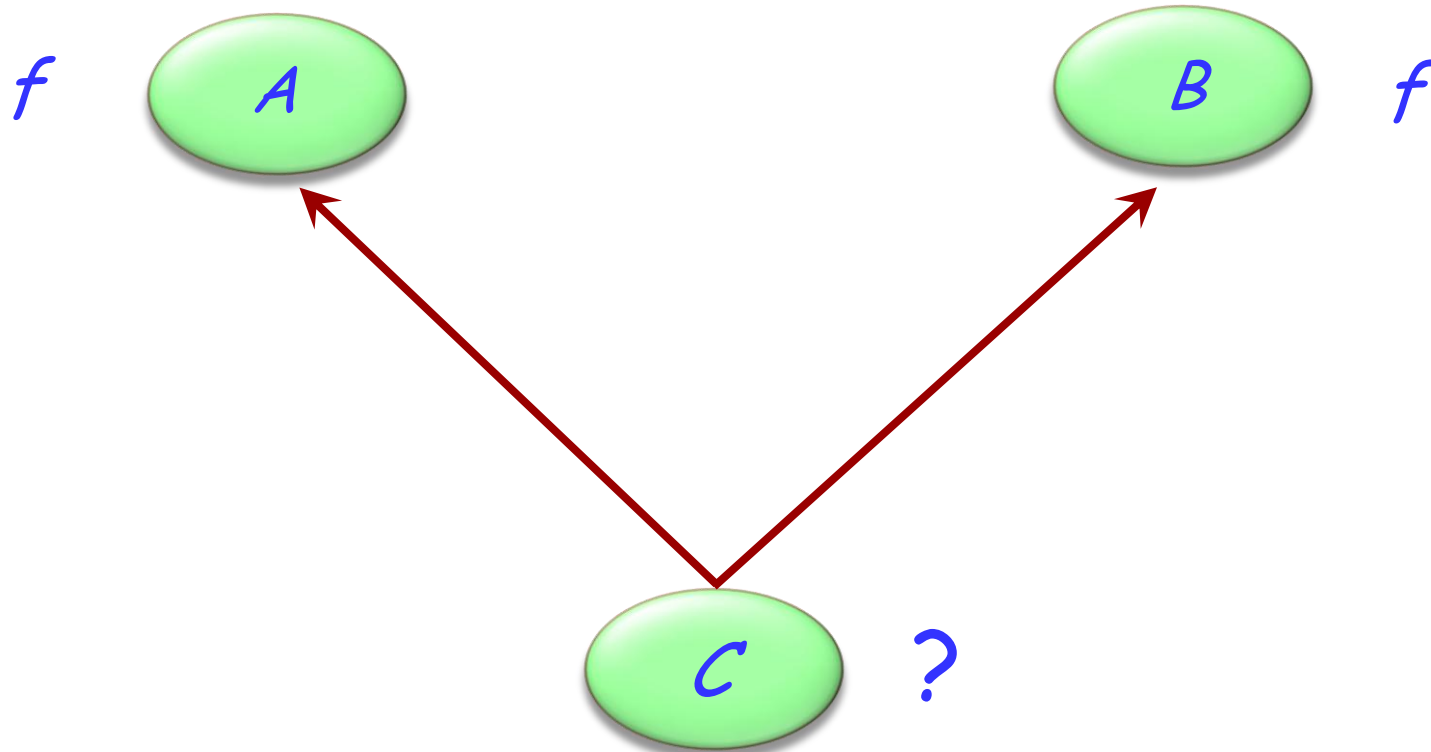


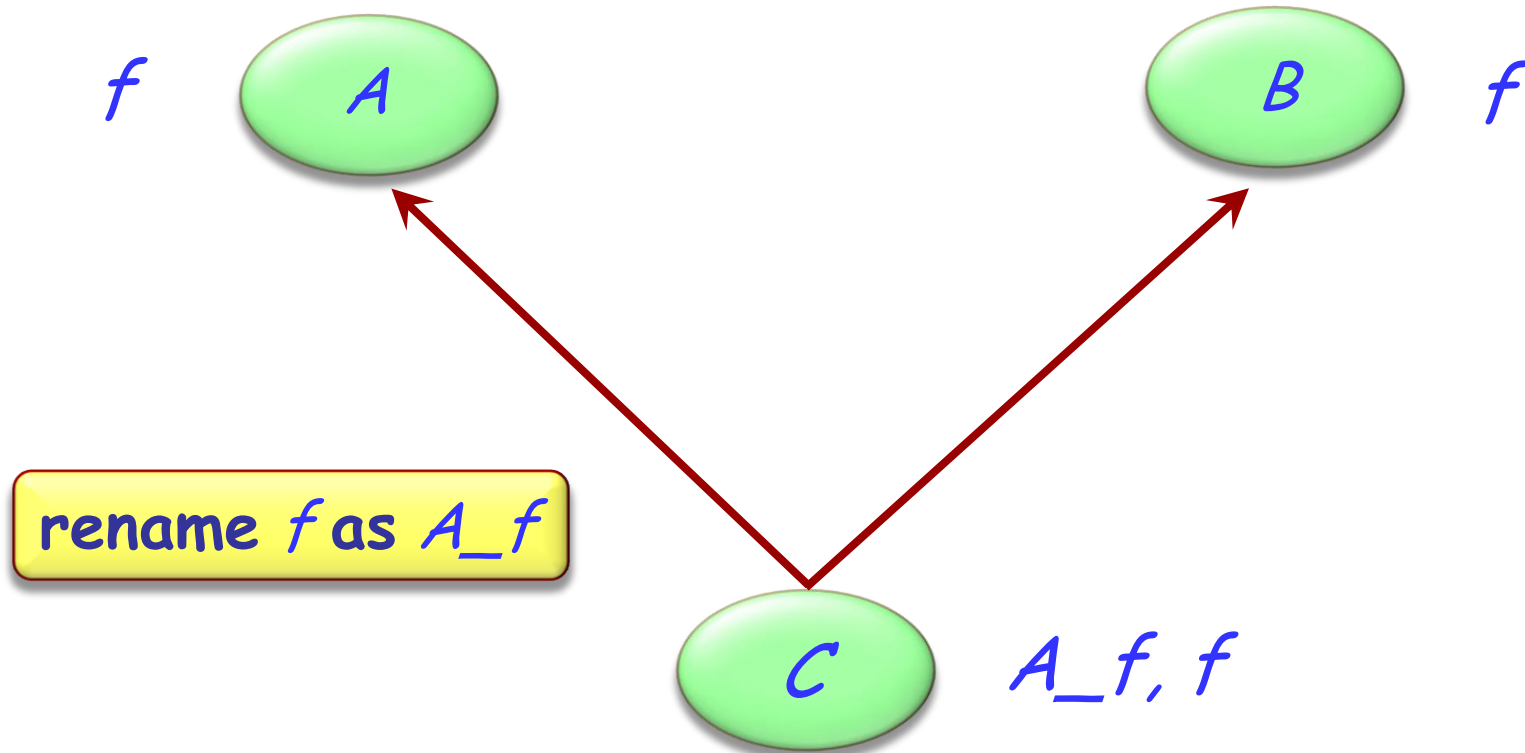
feature

... Implementiere *LIST*-Features mit *ARRAY*-Features ...

end

Mehrfachvererbung: Namenskonflikte





a1: A

b1: B

c1: C

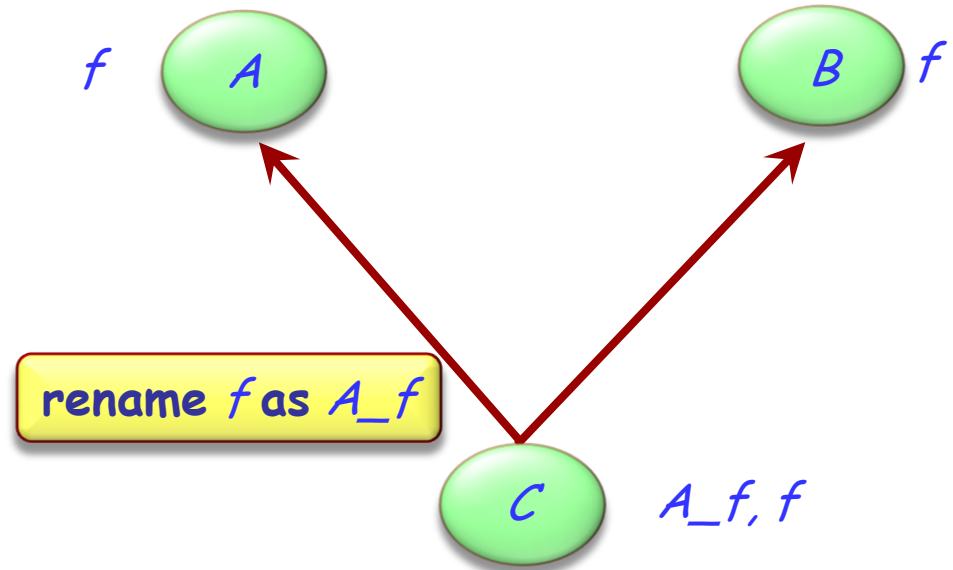
...

c1.f

c1.A_f

a1.f

b1.f



Ungültig:

➤ *a1.A_f*

➤ *b1.A_f*



Ein Namenskonflikt muss beseitigt werden, es sei denn, er geschieht:

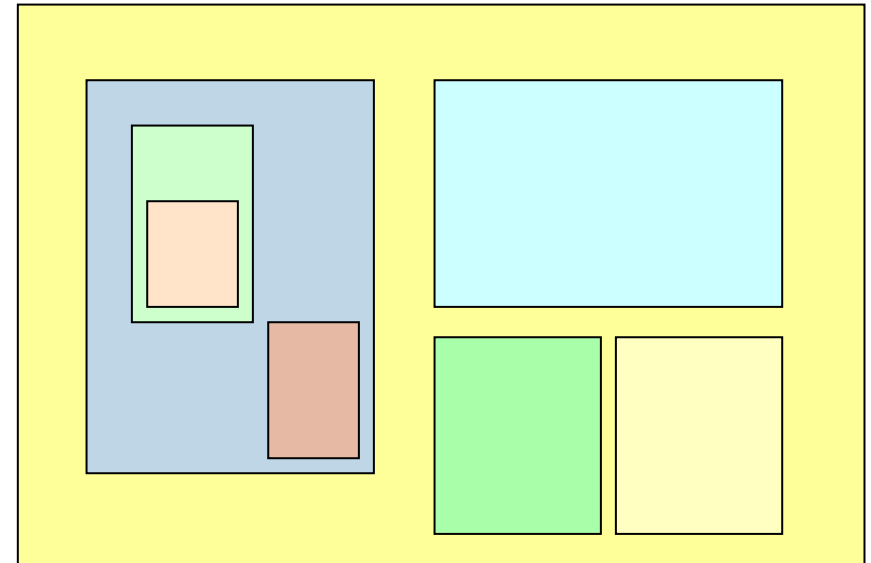
- Durch wiederholte Vererbung (d.h. kein wirklicher Konflikt)
- Zwischen Features, von denen höchstens eines wirksam ist.
(d.h. die übrigen sind aufgeschoben)

Noch eine Anwendung von Umbenennungen



Eine (lokal) bessere Terminologie ermöglichen.

Beispiel: *child* (*TREE*); *subwindow* (*WINDOW*)



Umbenennungen, um die Terminologie zu verbessern



"Graphische" Features: *height, width, change_height, change_width, xpos, ypos, move...*

"Hierarchische" Features: *superwindow, subwindows, change_subwindow, add_subwindow...*

```
class WINDOW inherit  
  RECTANGLE  
  TREE[WINDOW]
```

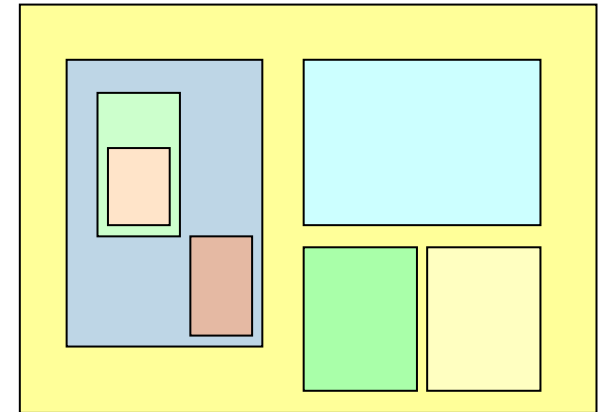
```
  rename
```

```
    parent as superwindow,  
    children as subwindows,  
    add_child as add_subwindow
```

```
  end
```

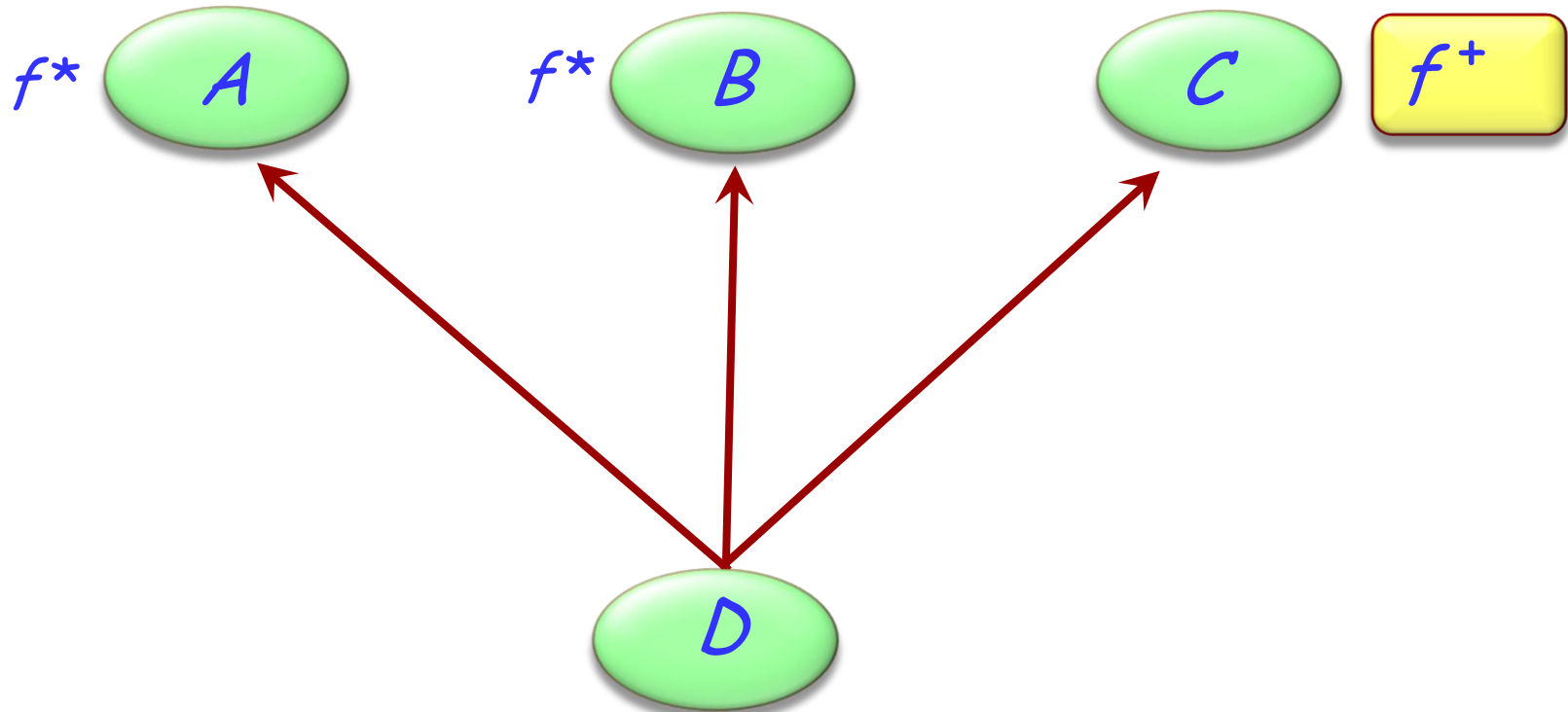
```
feature
```

```
end
```



ABER: Siehe
Stilregeln
betreffend
einheitlichen
Featurenamen

Features verschmelzen

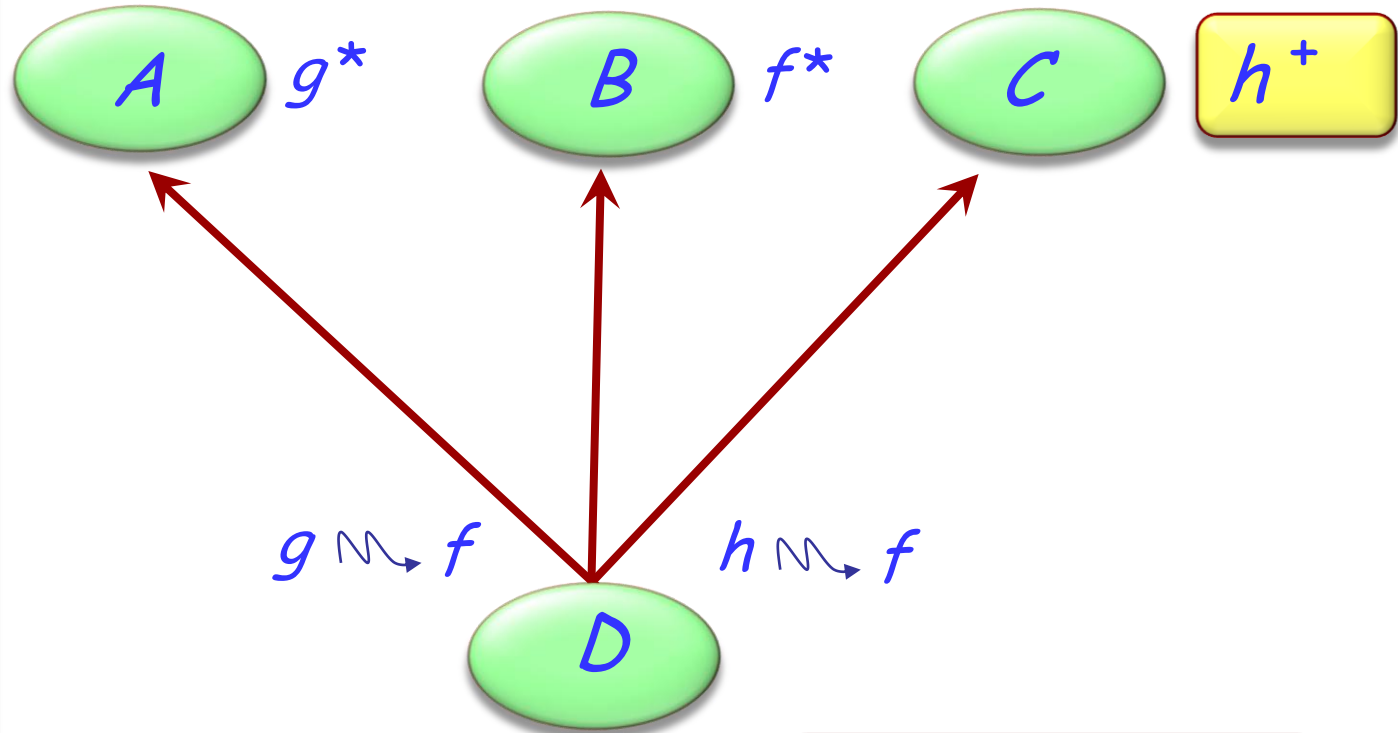


* aufgeschoben
+ wirksam

Features verschmelzen: Mit verschiedenen Namen

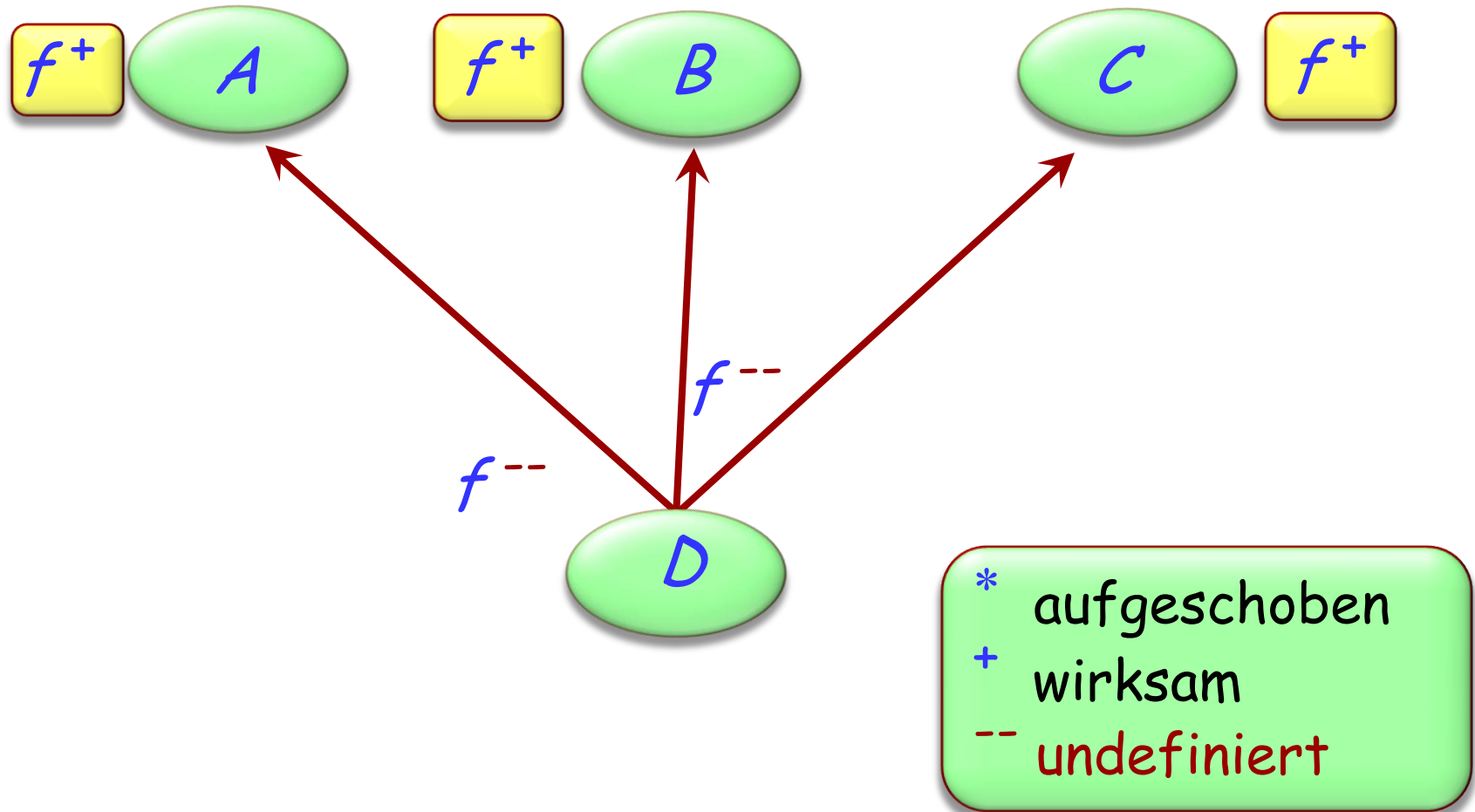


```
class
  D
  inherit
    A
    rename
      g as f
    end
    B
    C
    rename
      h as f
    end
  feature
    ...
  end
```



* aufgeschoben
+ wirksam
 \rightsquigarrow Umbenennung

Features verschmelzen: wirksame Features





deferred class

T

inherit

S

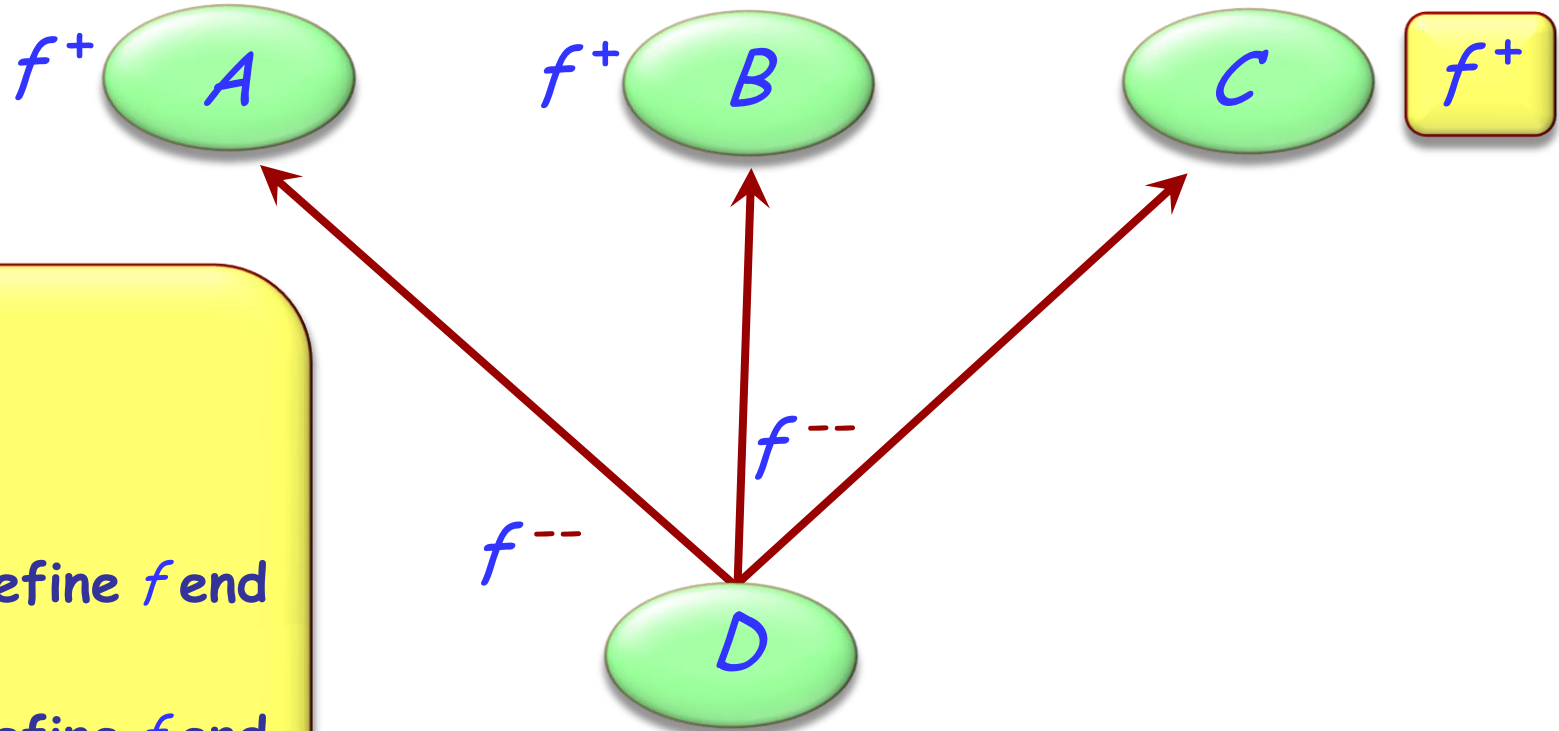
undefine *v* end

feature

...

end

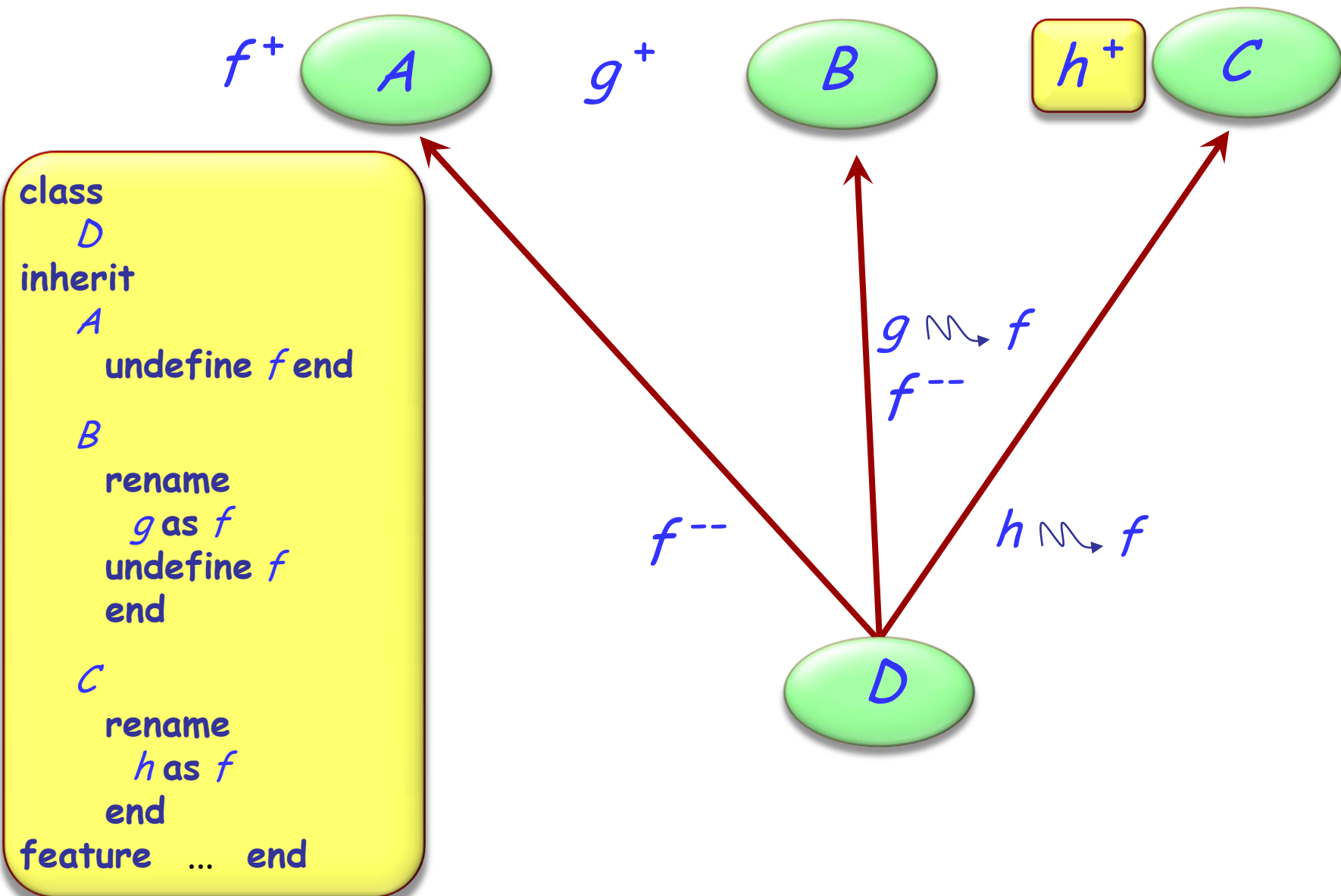
Verschmelzen durch undefinition



```
class
  D
inherit
  A
  undefine f end
  B
  undefine f end
  C
feature
  ...
end
```

* aufgeschoben
+ wirksam
-- undefiniert

Verschmelzen von Features mit unterschiedlichen Namen





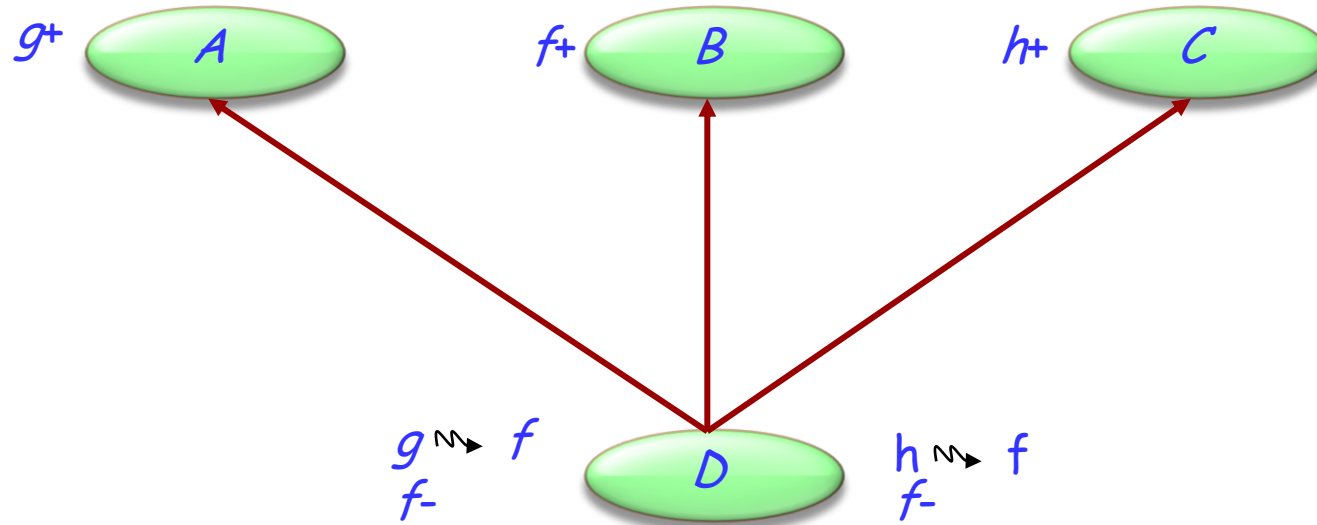
Wenn geerbte Features alle den gleichen Namen haben, besteht kein schädlicher Namenskonflikt, falls:

- Sie alle eine kompatible Signatur haben
- Maximal eines von ihnen wirksam ist

Die Semantik eines solchen Falls:

- Alle Features zu einem verschmelzen
- Falls es ein wirksames Feature gibt, wird dessen Implementierung übernommen

Verschmelzung von Features: wirksame Features



$a1: A$
 $a1.g$

$b1: B$
 $b1.f$

$c1: C$
 $c1.h$

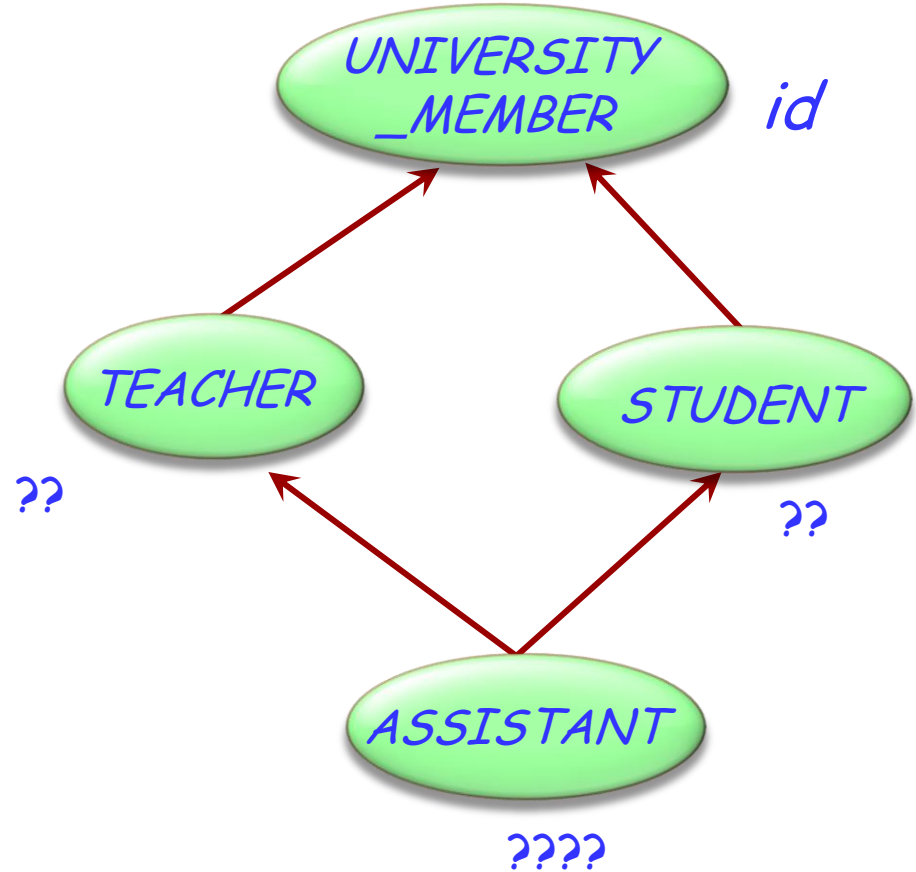
$d1: D$
 $d1.f$

Ein Spezialfall der Mehrfachvererbung



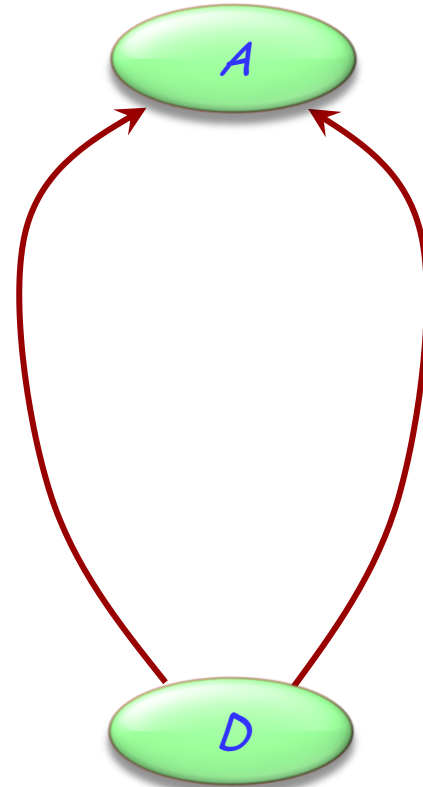
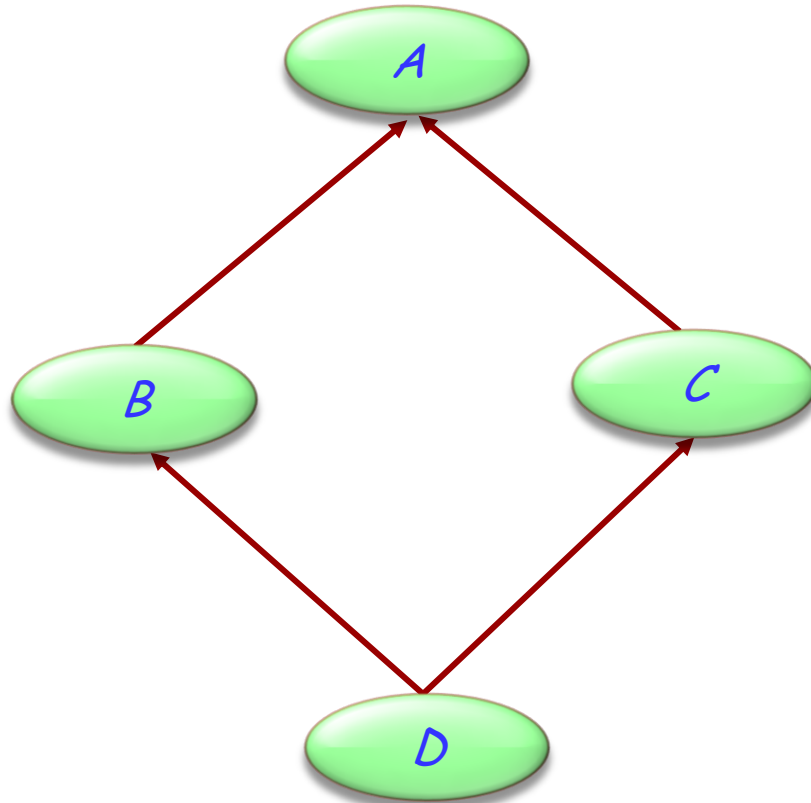
Ermöglicht einer Klasse, zwei oder mehrere Vorfahren zu haben.

Beispiel: *ASSISTANT* erbt von *TEACHER* und *STUDENT*.



Dies ist ein Fall der **wiederholten** Vererbung.

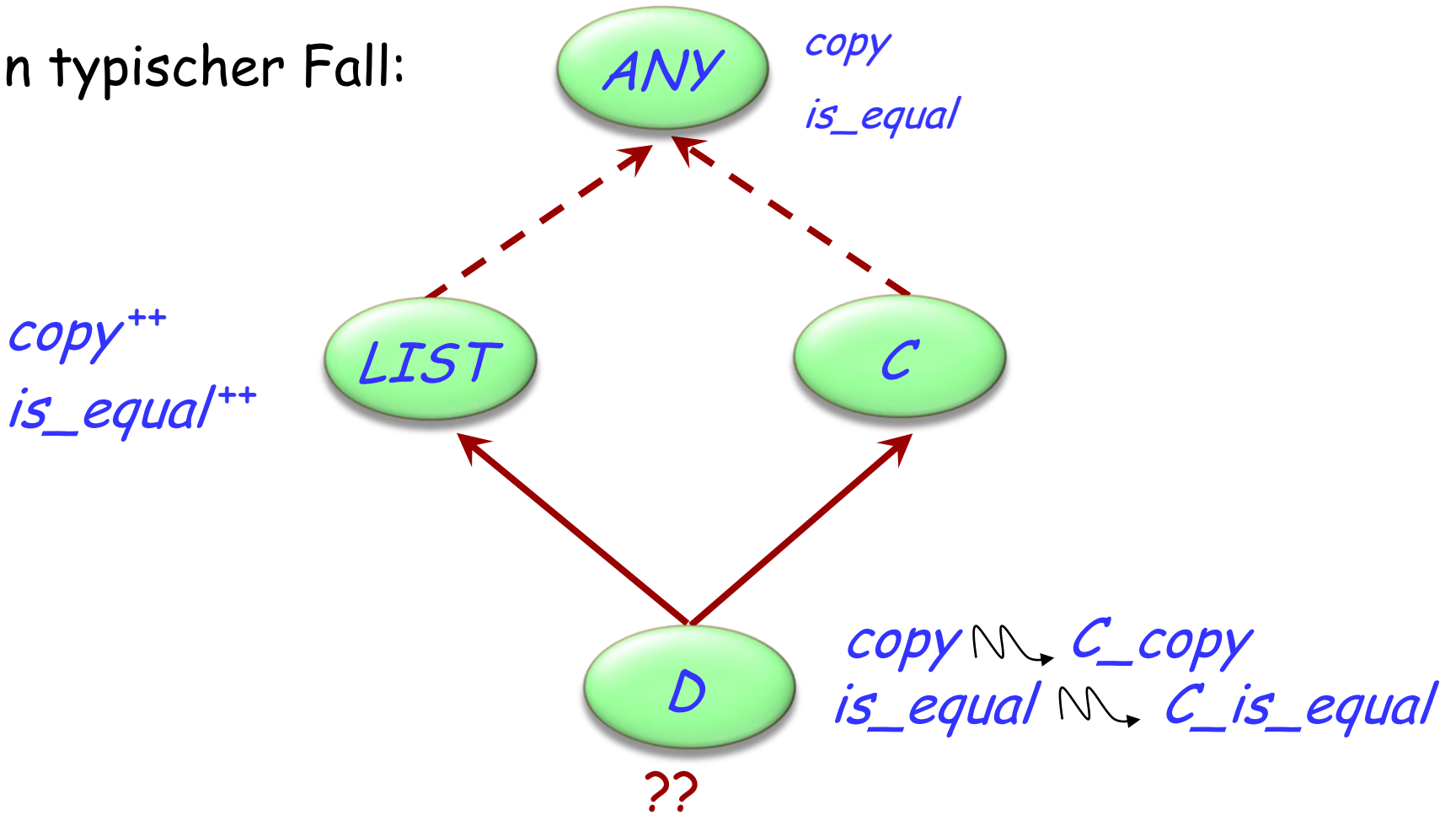
Indirekt und direkt wiederholte Vererbung



Mehrfachvererbung ist auch wiederholte Vererbung



Ein typischer Fall:



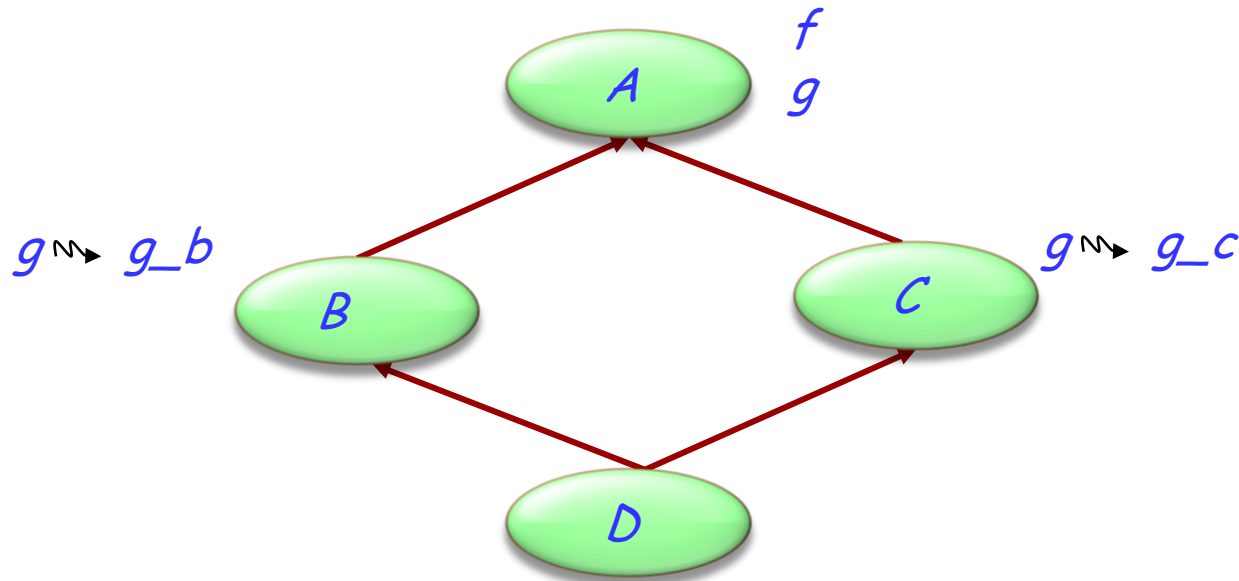


Wenn geerbte Features alle den gleichen Namen haben, besteht kein schädlicher Namenskonflikt, falls:

- Sie alle eine kompatible Signatur haben
- Maximal eines von ihnen wirksam ist

Die Semantik eines solchen Falls:

- Alle Features zu einem verschmelzen
- Falls es ein wirksames Feature gibt, wird dessen Implementierung übernommen



Features, wie z.B. f , die auf ihren Vererbungspfaden nicht umbenannt wurden, werden geteilt (shared).

Features, wie z.B. g , die unter unterschiedlichem Namen geerbt werden, werden vervielfältigt (*replicated*).

Wann ist ein Namenskonflikt akzeptabel?



(Konflikt zwischen n direkten oder geerbten Features derselben Klasse. Alle Features haben denselben Namen)

- Sie müssen alle kompatible Signaturen haben.
- Falls mehr als eines wirksam ist, müssen diese alle vom gleichen Vorfahren (durch wiederholte Vererbung) abstammen.

Der Bedarf nach „select“



Eine mögliche Doppeldeutigkeit entsteht durch Polymorphie und dynamisches Binden:

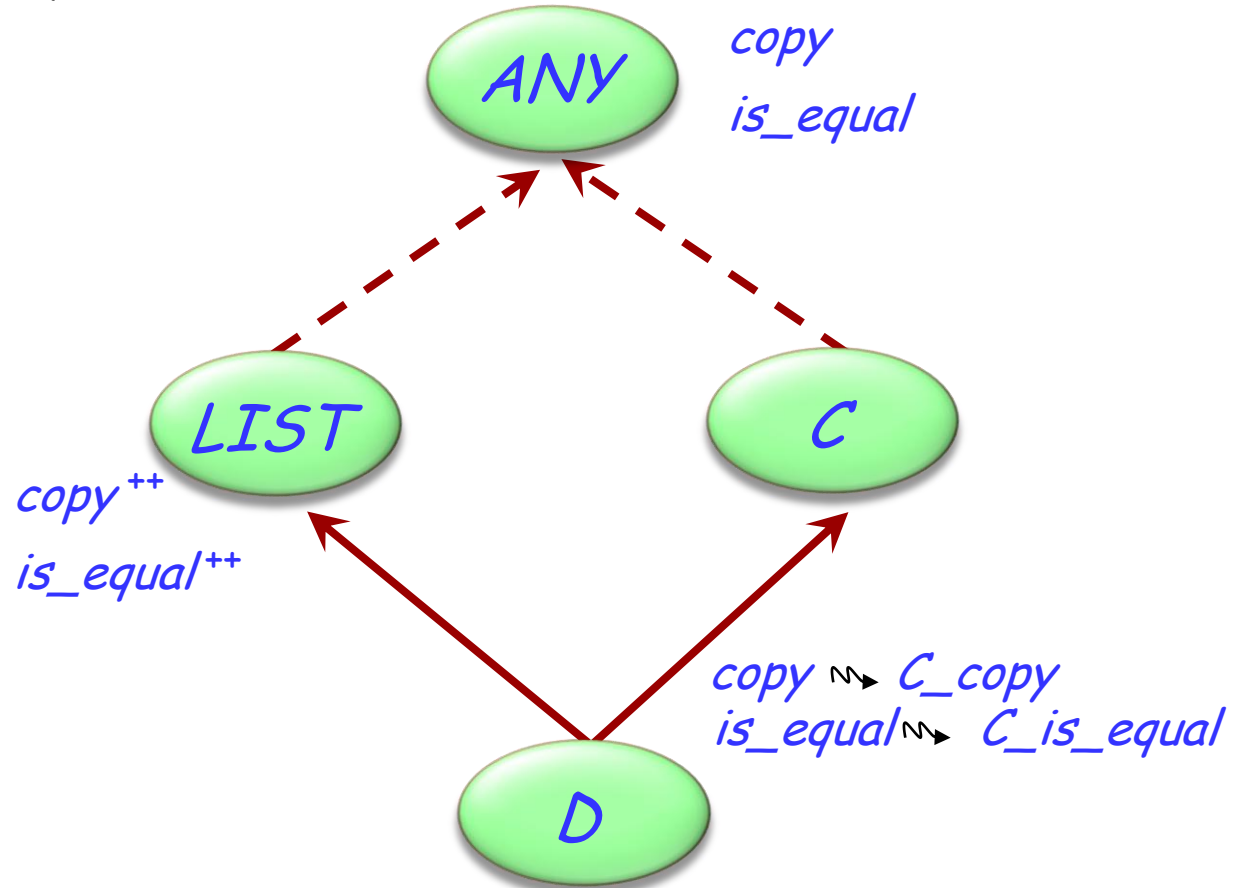
a1: ANY

d1: D

...

a1 := d1

a1.copy(...)



Die Doppeldeutigkeit auflösen



class

D

inherit

LIST [T]

```
select
    copy,
    is_equal
end
```

C

rename

copy as C_copy,
is_equal as C_is_equal,

...

end



Einige Spielchen, die man mit Vererbung spielen kann:

- Mehrfachvererbung
- Verschmelzen von Features
- Wiederholte Vererbung