ETH zürich
Chair of Software Engineering

Einführung in die Programmierung

Prof. Dr. Bertrand Meyer

Lektion 18:

Mehr über Vererbung und Agenten

Themen

Aufgeschobene Klassen

Einen Typ erzwingen

Undo-Redo (Komando-Muster)

Mehrfache Vererbung

Verträge in der Vererbung

Aufgeschobene Klassen

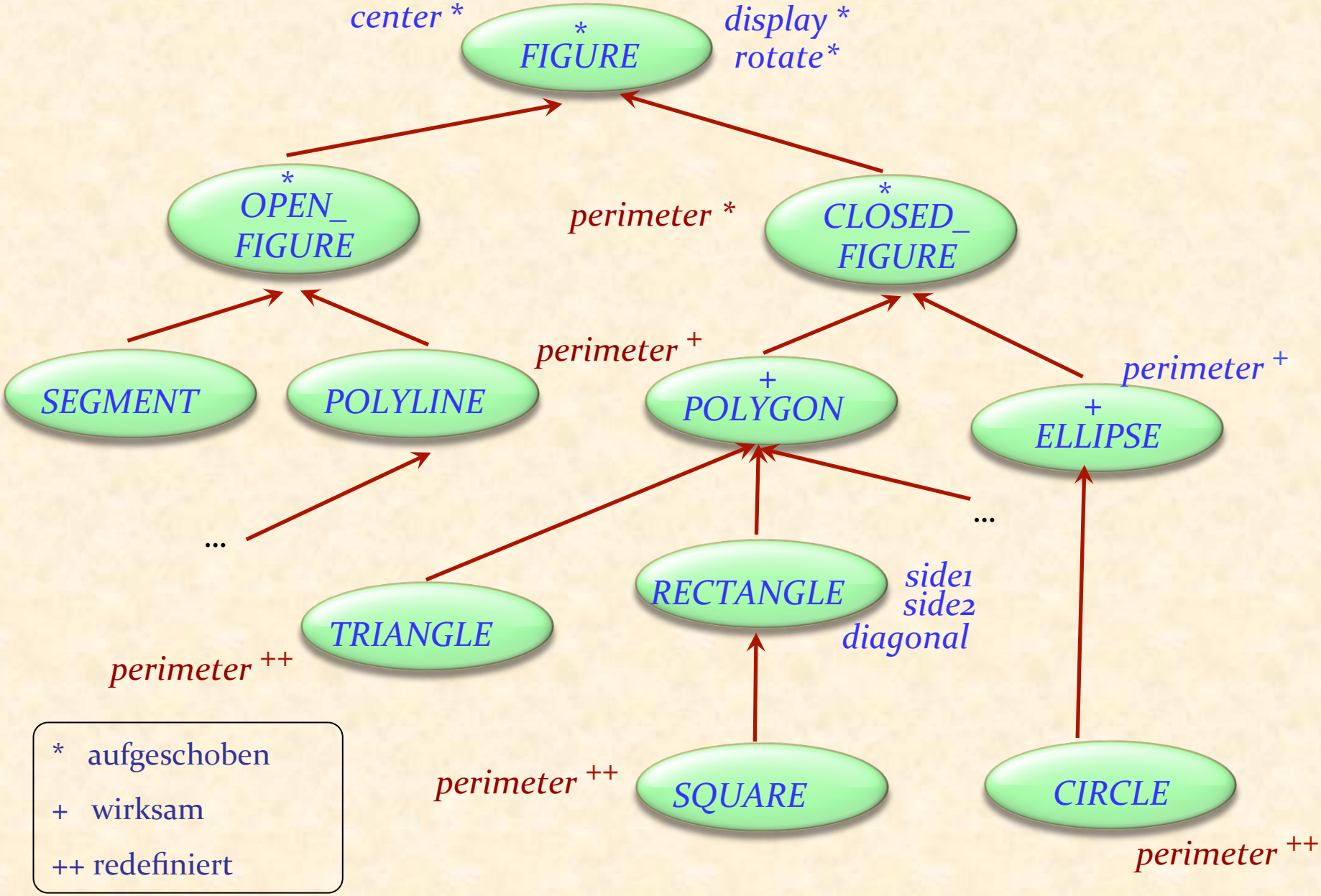
Die Rolle von aufgeschobenen Klasse

Ausdrücken von abstrakten Konzepten, unabhängig von der Implementation.

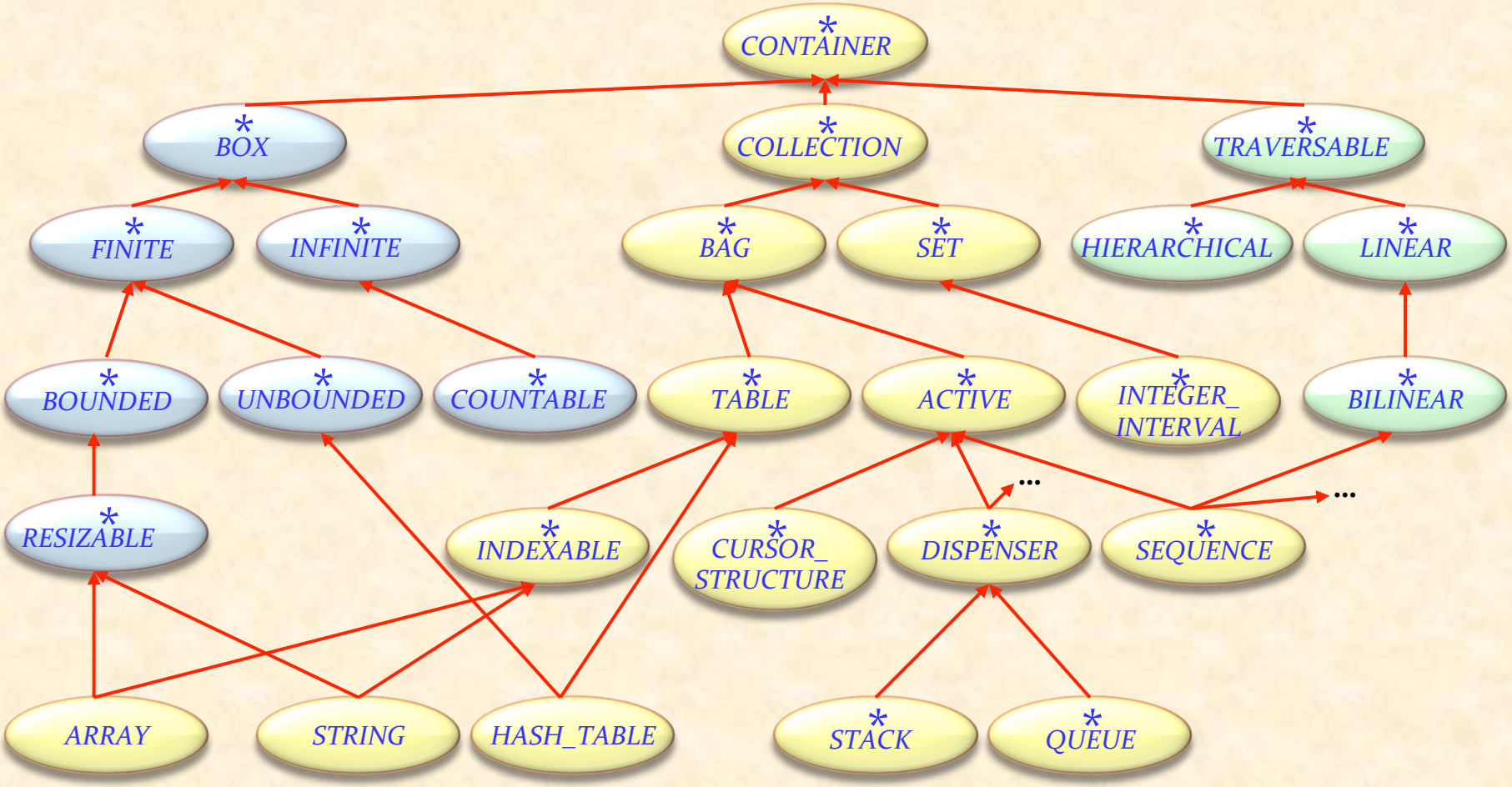
Ausdrücken von gemeinsamen Elementen von mehreren Implementationen.

Terminologie: **wirksam** = nicht aufgeschoben
(d.h. vollständig implementiert)

Beispielhierarchie



Aufgeschobene Klassen in EiffelBase



Ein aufgeschobenes Feature

In *ITERATION_CURSOR* and *LIST*:

forth

require
not *after*

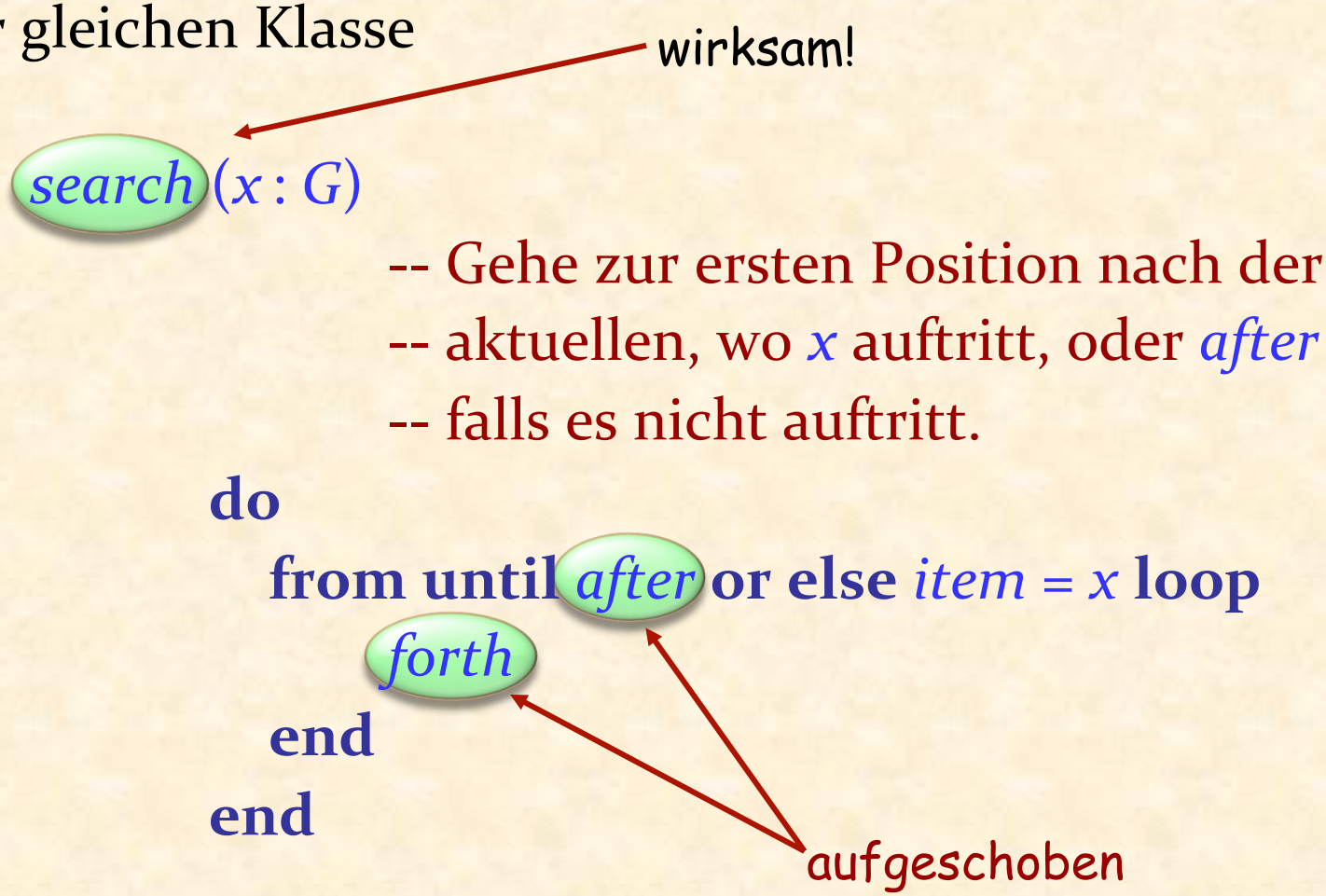
deferred

ensure
index = **old** *index* + 1

end

Aufgeschobene und wirksame Features mischen

In der gleichen Klasse



“Programme mit Lücken“

“Rufen sie uns nicht auf, wir rufen sie auf!”

Eine mächtige Form von Wiederverwendbarkeit:

- Das wiederverwendbare Element definiert ein allgemeines Schema.
- Spezifische Fälle füllen die Lücken in diesem Schema

Kombiniert Wiederverwendung mit Adaption

Aufgeschobene Klassen vs. Java-Schnittstellen

Schnittstellen sind „vollständig aufgeschoben“:

nur aufgeschobene Features

Aufgeschobene Klassen können wirksame Features beinhalten, die auf aufgeschobene zugreifen, wie etwa im *COMPARABLE*-Beispiel

Flexibler Mechanismus, um Abstraktionen schrittweise zu implementieren

Anwendungen von aufgeschobenen Klassen

Abstraktion

Systematik

Analyse und Entwurf auf einer hohen Ebene

...

Analysis-Beispiel: Fernsehstation

class *SCHEDULE* **feature**

segments : *LIST* [*SEGMENT*]

end

Quelle: Object-Oriented Software Construction, 2nd edition, Prentice Hall

Programme

```
note
        Beschreibung :
        " 24-Stunden TV Programm"
deferred class SCHEDULE
feature


        segments : LIST  [SEGMENT ]
                -- Folge von Segmenten.
            deferred
            end


        air_time : DATE
           -- 24-Stunden-Periode
            -- für dieses Programm.
            deferred
            end
```

```
        set_air_time (t : DATE)
                -- Zuweisung des Programms,
                -- das zur Zeit t ausgestrahlt  wird.
            require
                t.in_future
            deferred
            ensure
                air_time = t
            end
        print
                -- Papier-Ausgabe drucken.
            deferred
            end
end
```

Segment

```
note
    Beschreibung: "Individuelle
    Fragmente eines Programms "
deferred class SEGMENT feature
    schedule : SCHEDULE deferred end
        -- Programm, zu welchem das
        -- Segment gehört.
    index : INTEGER deferred end
        -- Position des Segment
        -- in seinem Programm.
    starting_time, ending_time :
        INTEGER deferred end
        -- Beginn und Ende der
        -- geplanten Ausstrahlungszeit.
    next: SEGMENT deferred end
        -- Segment, das als nächstes
        -- ausgestrahlt wird (falls vorh.).
```

```
    sponsor : COMPANY deferred end
        -- Hauptsponsor des Segments.

    rating : INTEGER deferred end
        -- Einstufung (geeignet für Kinder
        etc.).


        … Befehle wie
        change_next, set_sponsor, set_rating,
        nicht eingeschlossen …

    Minimum_duration : INTEGER = 30
        -- Minimale Länge des Segmentes,
        -- in Sekunden.

    Maximum_interval : INTEGER = 2
        -- Maximale Zeit zwischen zwei
        -- aufeinanderfolgenden
        -- Segmenten, in Sekunden.
```

Segment (fortgesetzt)

invariant

in_list: (*1* <= *index*) **and** (*index* <= *schedule.segments.count*)

in_schedule: *schedule.segments.item* (*index*) = **Current**

next_in_list: (*next* /= **Void**) **implies**

(*schedule.segments.item* (*index* + *1*) = *next*)

no_next_iff_last: (*next* = **Void**) = (*index* = *schedule.segments.count*)

non_negative_rating: *rating* >= *0*

positive_times: (*starting_time* > *0*) **and** (*ending_time* > *0*)

sufficient_duration:

ending_time – starting_time >= *Minimum_duration*

decent_interval :

(*next.starting_time*) - *ending_time* <= *Maximum_interval*

end

Werbung

```
note
  Beschreibung:
      „Werbeblock"
deferred class COMMERCIAL inherit
  SEGMENT
      rename sponsor as advertizer end

feature
  primary : PROGRAM  deferred
      -- Programm, zu welchem die
      -- Werbung gehört.

  primary_index : INTEGER
      deferred
      -- Index von 'primary'.
```

```
set_primary (p : PROGRAM)
    -- Werbung zu p hinzufügen.
  require
    program_exists: p /= Void
    same_schedule:  p.schedule  = schedule
    before: p.starting_time  <=
  starting_time
  deferred
  ensure
    index_updated:
          primary_index  = p.index
    primary_updated: primary  = p
  end
```

Werbung (fortgesetzt)

invariant

meaningful_primary_index: *primary_index* = *primary*.*index*

primary_before: *primary*.*starting_time* <= *starting_time*

acceptable_sponsor: *advertizer*.*compatible* (*primary*.*sponsor*)

acceptable_rating: *rating* <= *primary*.*rating*

end

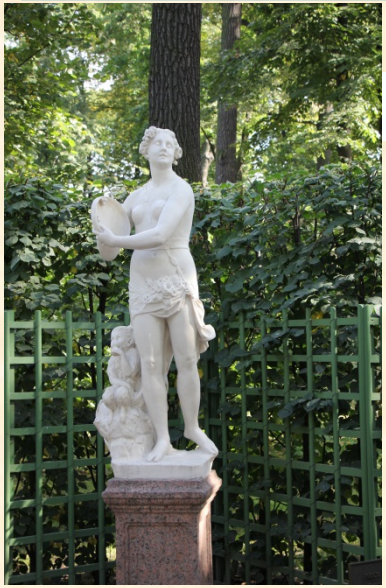
Beispiel: Chemisches Kraftwerk

```
deferred class
        VAT
inherit
        TANK
feature
        in_valve, out_valve : VALVE
                -- Fülle den Tank.
            require
                in_valve.open
                out_valve.closed
            deferred
            ensure
                in_valve.closed
                out_valve.closed
                is_full
            end

        empty, is_full, is_empty, gauge, maximum, ... [Andere Features] ...

invariant
        is_full = (gauge >= 0.97 * maximum)  and  (gauge <= 1.03 * maximum)
end
```

Was wir gesehen haben

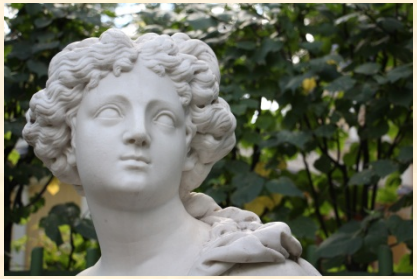
Aufgeschobene Klassen und ihre Rolle in Softwareanalyse und –entwurf

Undo-
Redo







Ein Beispiel: Undo-Redo

Undo/Redo sinnvoll in jeder interaktiven Anwendung

- Entwickeln Sie keine interaktiven Anwendungen, ohne Undo/Redo zu implementieren

Nützliches Design-Pattern ("**Command**" Pattern)

Veranschaulicht die Verwendung von Algorithmen und Datastrukturen

Beispiel für O-O Techniken: Vererbung, Aufgeschobene Klassen, Polymorphe Datenstrukturen, Dynamisches Binden,...

Beispiel einer schönen und eleganten Lösung

Referenzen:

- Kapitel 21 in Object-Oriented Software Construction, Prentice Hall, 1997
- Erich Gamma et al., *Design Patterns,* Addison –Wesley, 1995: "Command pattern"

Die Problemstellung

Dem Benutzer eines interaktiven Systems die Möglichkeit geben, die letzte Aktion rückgängig zu machen

Bekannt als "**Control-Z**"

Soll mehrstufiges rückgängig Machen ("**Control-Z**") und Wiederholen ("**Control-Y**") ohne Limitierung unterstützen, ausser der Benutzer gibt eine maximale Tiefe an

In unserem Beispiel: Ein Texteditor

Begriff der „aktuellen Zeile“ mit folgenden Befehlen:

- Löschen der aktuellen Zeile
- Ersetzen der aktuellen Zeile mit einer Anderen
- Einfügen einer Zeile vor der aktuellen Position
- Vertauschen der aktuellen Zeile mit der Nächsten (falls vorhanden)
- „Globales Suchen und Ersetzen“ (fortan GSE): Jedes Auftreten einer gewissen Zeichenkette durch eine andere ersetzen
- …

Der Einfachheit halber nutzen wir eine zeilenorientierte Ansicht, aber die Diskussion kann auch auf kompliziertere Ansichten angewendet werden

Eine einfache Lösung

Sichern des gesamten Zustandes vor jeder Operation

Im Beispiel: Der Text, der bearbeitet wird
und die aktuelle Position im Text

Wenn der Benutzer ein „Undo“ verlangt, stelle den zuletzt gesicherten Zustand wieder her

Aber: Verschwendung von Ressource, insbesondere Speicherplatz

Intuition: Sichere nur die Änderungen (diff) zwischen zwei Zuständen

Schlüsselschritt im Entwerfen einer Software-Architektur

Die richtigen Abstraktionen finden

(die interessanten Objekt-Typen)

Hier:

Der Begriff eines “Befehls”

Die „Geschichte“ einer Sitzung speichern

Die Geschichte-Liste:

Ältester

Neuster

history : *LIST* [*COMMAND*]

Was ist ein “Command” (Befehl) -Objekt?

Ein Befehl-Objekt beinhaltet genügend Informationen über eine Ausführung eines Befehls durch den Benutzer, um

- Den Befehl auszuführen
- Den Befehl rückgängig zu machen

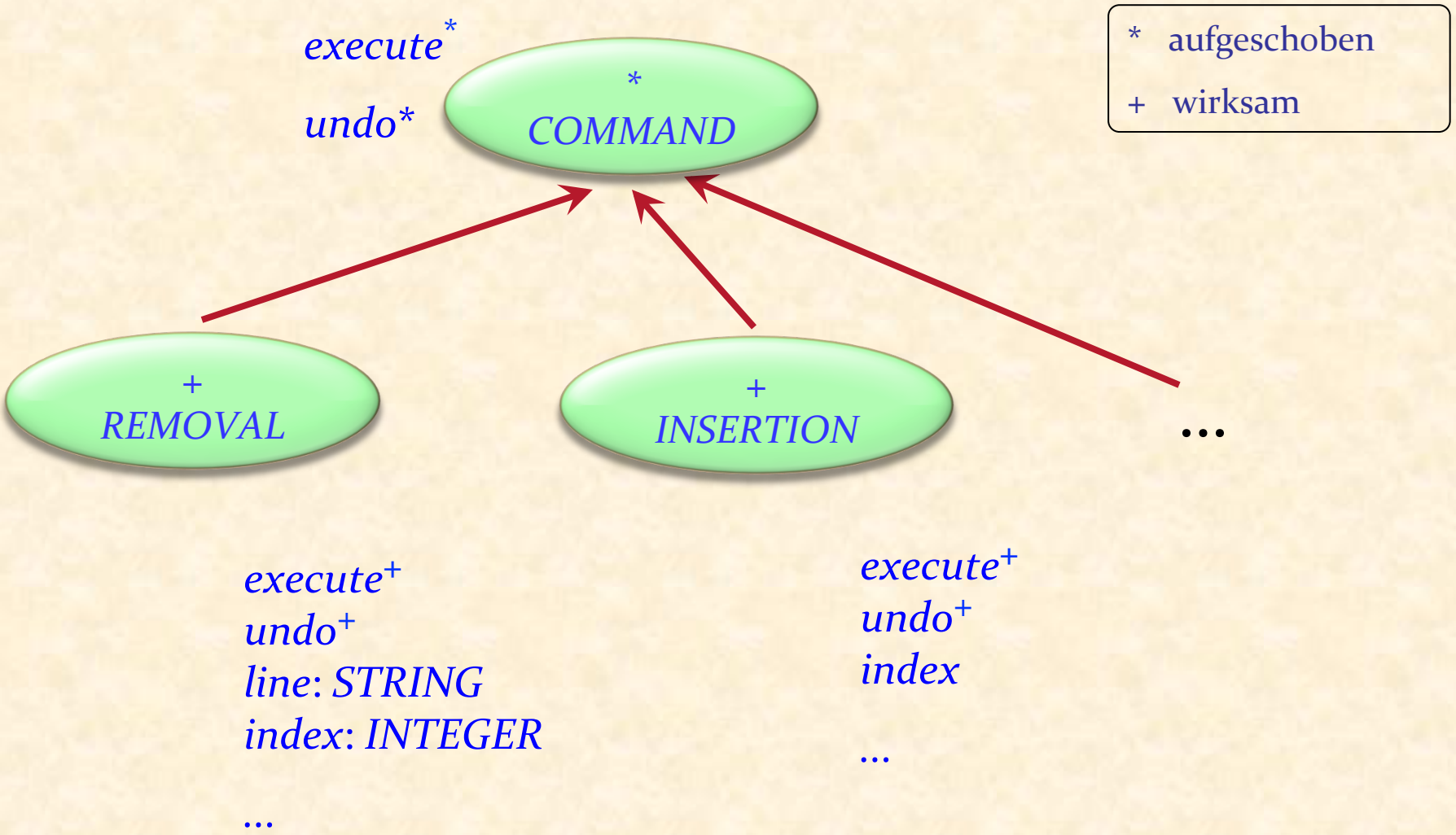
Beispiel: In einem “**Removal**”-Objekt brauchen wir:

- Die Position der zu löschenden Zeile
- Der Inhalt dieser Zeile!

Allgemeiner Begriff eines Befehls

```
deferred class COMMAND feature
    done: BOOLEAN
            -- Wurde dieser Befehl ausgeführt?
    execute
            -- Eine Ausführung des Befehls ausführen.
        deferred
        ensure
            already: done
        end
    undo
    machen.
            -- Eine frühere Ausführung des Befehls rückgängig
        require
            already: done
        deferred
        end
end
```

Die Befehl-Klassenhierarchie



Zugrundeliegende Klasse (Aus dem Geschäftsmodell)

```
class EDIT_CONTROLLER  feature
        text : LIST [STRING]
        position: ITERATION_CURSOR [STRING]
        remove
                -- Lösche Zeile an aktueller Position.
            require
                not off
            do
                position.remove
            end
        put_right (line : STRING)
                -- Füge line nach der aktuellen Position ein.
            require
                not after
            do
                position.put_right (line)
            end
```

... Auch: *item, index, go_ith, put_left* ...

```
end
```

Eine Befehlsklasse (Skizze, ohne Verträge)

```
class REMOVAL  inherit COMMAND feature
        controller : EDIT_CONTROLLER
                        -- Zugriff auf das Geschäftsmodell.
        line : STRING
                        -- Zu löschende Zeile.
        index : INTEGER
                        -- Position der zu löschenden Zeile.

        execute
                        -- Lösche aktuelle Zeile und speichere sie.
                do      line := controller.item ; index := controller.index
                        controller.remove ; done := True
                end

        undo
                        -- Füge vorher gelöschte Zeile wieder ein.
                do      controller.go_i_th (index)
                        controller.put_left (line)
                end
end
```

Die Geschichte-Liste

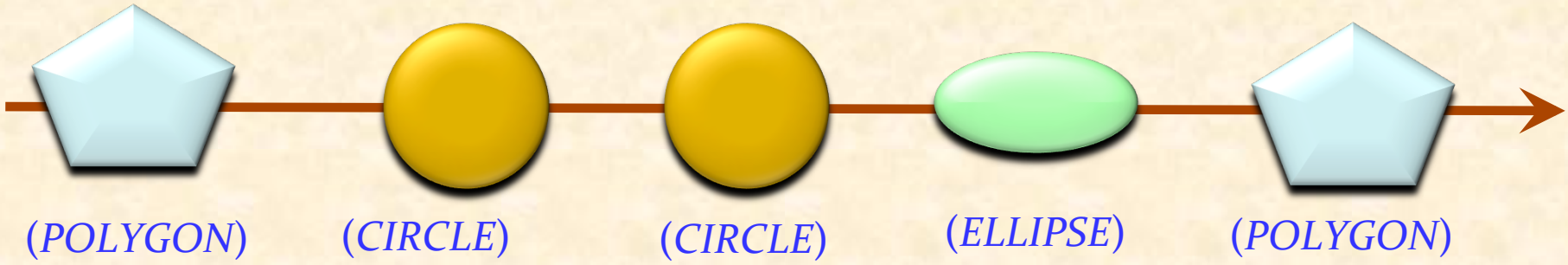
Eine polymorphe Datenstruktur

Ältester

Neuester

history: *LIST* [*COMMAND*]

Erinnerung: Liste von Figuren



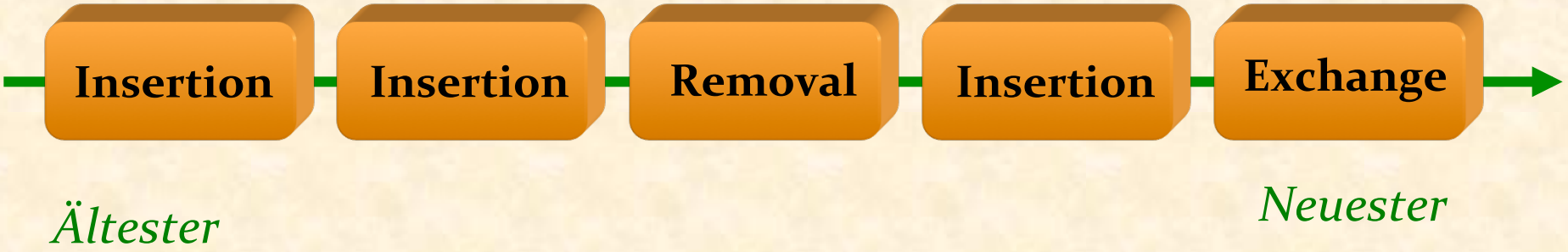

```
bilder.extend (p1) ; bilder.extend (c1) ; bilder.extend (c2)
bilder.extend (e) ; bilder.extend (p2)
```

```
bilder : LIST [FIGURE]
p1, p2 : POLYGON
c1, c2 : CIRCLE
e : ELLIPSE
```

```
class LIST [G] feature
        extend (v : G) do ... end
        last : G
        ...
end
```

Die Geschichte-Liste

Eine polymorphe Datenstruktur



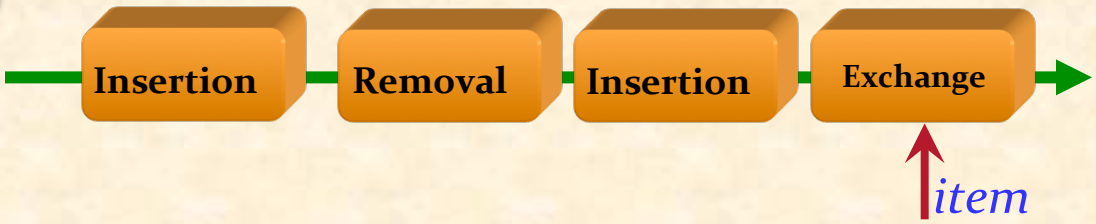
history : *LIST* [*COMMAND*]

cursor: *ITERATION_CURSOR* [*COMMAND*]

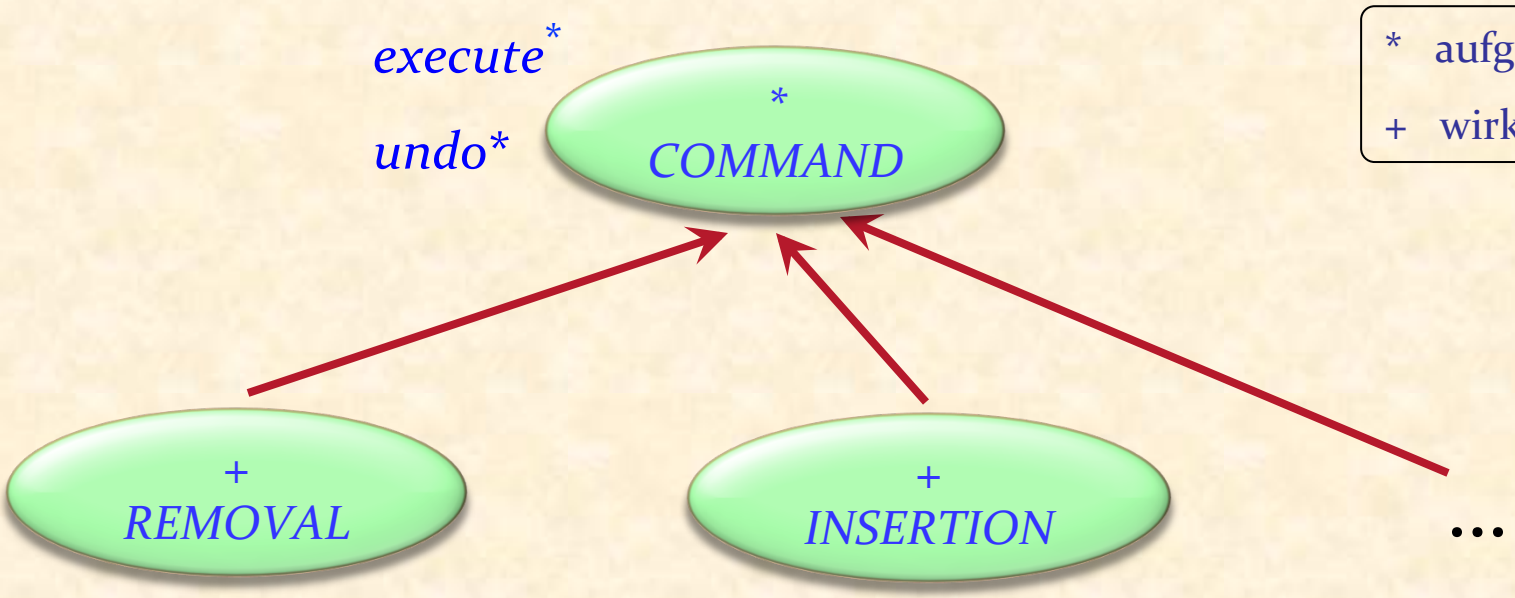
Einen Benutzerbefehl ausführen

```
decode_user_request

if "Anfrage ist normaler Befehl" then
        "Erzeuge ein Befehlsobjekt c , der Anforderung entsprechend"
        history.extend (c)
        c.execute
elseif "Anfrage ist UNDO" then
        if not cursor.before then -- Ignoriere überschüssige Anfragen
            cursor.item.undo
            cursor.back
        end
elseif "Anfrage ist REDO" then
        if not cursor.is_last then -- Ignoriere überschüssige Anfragen
            cursor.forth
            cursor.item.execute
        end
end
```

Pseudocode, siehe nächste Implementation



Die Befehl-Klassenhierarchie



execute+
undo+
line: *STRING*
index: *INTEGER*

...

execute+
undo+
index

...

Befehlsobjekte erzeugen: Erster Ansatz

c : *COMMAND*

...

decode_user_request

if *"Anfrage ist* **Removal***"* **then**

    create {*REMOVAL*} *c*

elseif *"Anfrage ist* **Insertion***"* **then**

    create {*INSERTION*} *c*

else

    etc.

Befehlsobjekte erzeugen: Besserer Ansatz

«Befehl-Factory»

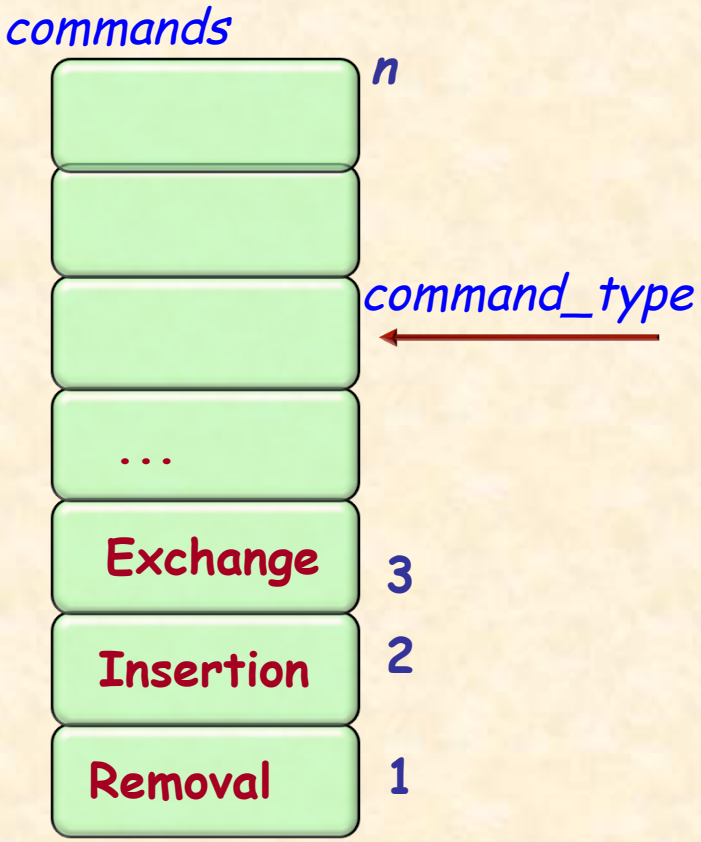
Geben Sie jedem Befehls-Typ eine Nummer

Füllen Sie zu Beginn einen Array *befehle* mit je einer Instanz jedes Befehls-Typen.

Um neue Befehlsobjekte zu erhalten:

"Bestimme *command_type*"

c := (*commands* [*command_type*]).*twin*



Einen "Prototypen" duplizieren

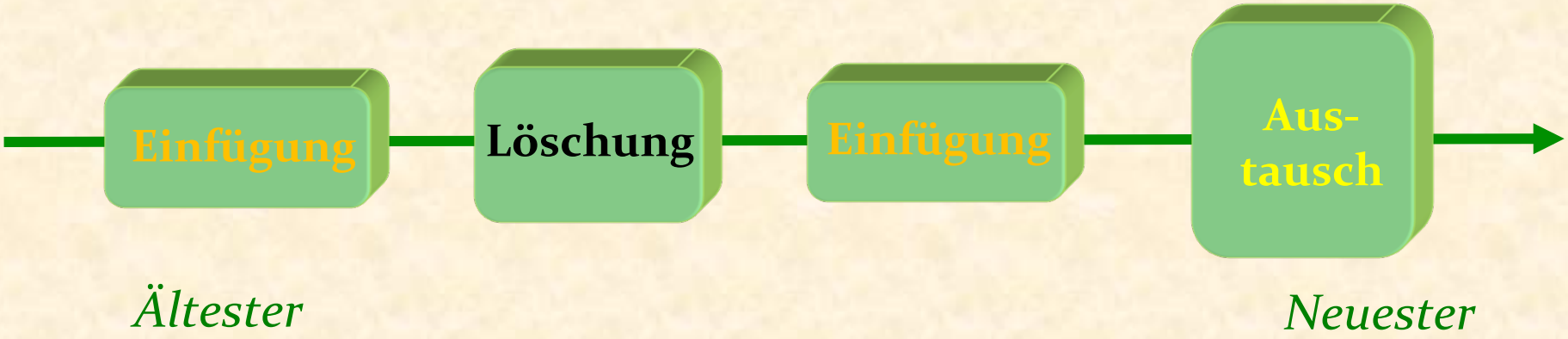
Der Gebrauch von Agenten

Für jeden Benutzerbefehl haben wir zwei Routinen:

- Die Routine, um ihn auszuführen
- Die Routine, um ihn rückgängig zu machen

Die Geschichte-Liste im Undo/Redo-Pattern

history: *LIST* [*COMMAND*]



Die Geschichte-Liste mit Agenten

Die Geschichte wird einfach zu einer Liste von Agentenpaaren:

history: *LIST* [*TUPLE*

Benanntes Tupel

[*execute* : *PROCEDURE* [*ANY, TUPLE*];

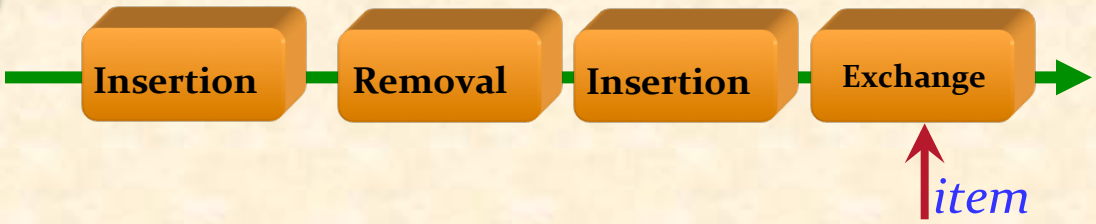
undo : *PROCEDURE* [*ANY, TUPLE*]]

Das Grundschema bleibt dasselbe, aber man braucht nun keine Befehlsobjekte mehr; die Geschichte-Liste ist einfach eine Liste, die Agenten enthält

Einen Benutzerbefehl ausführen (vorher)

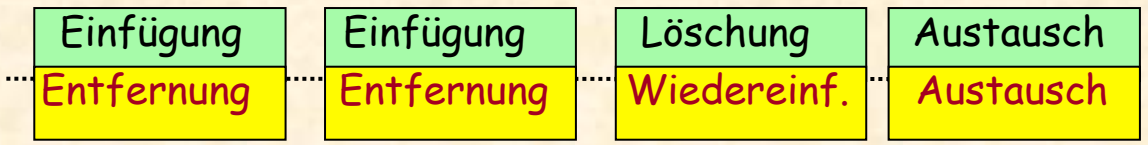
decode_user_request

if "Anfrage ist normaler Befehl" **then**
"Erzeuge ein Befehlsobjekt *c* , der Anforderung entsprechend"
history.extend (c)
c.execute
elseif "Anfrage ist UNDO" **then**
if not *cursor.before* **then** -- Ignoriere überschüssige Anfragen
cursor.item.undo
cursor.back
end
elseif "Anfrage ist REDO" **then**
if not *cursor.is_last* **then** -- Ignoriere überschüssige Anfragen
cursor.forth
cursor.item.execute
end
end

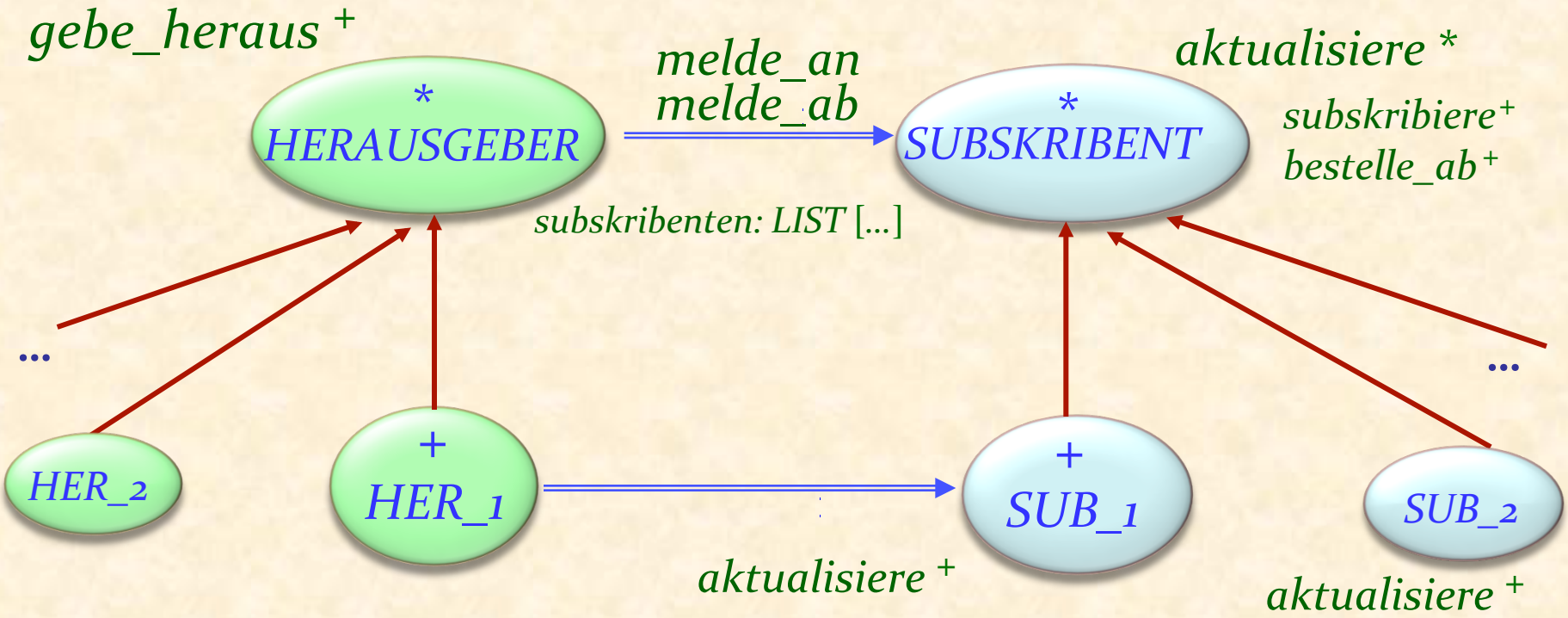


Einen Benutzerbefehl ausführen (jetzt)

```
"Dekodiere Benutzeranfrage mit zwei Agenten do_it and undo_it"
if "Anfrage ist normaler Befehl" then
        from until cursor.is_last loop cursor.remove_right end
        history.extend ([do_it, undo_it ]) ; cursor.forth
        do_it.call ([])
elseif "Anfrage ist UNDO" then
        if not cursor.before then -- Ignoriere überschüssige Anfragen
            cursor.item.execute.call ([])
            cursor.back
        end
elseif "Anfrage ist REDO" then
        if not cursor.is_last then -- Ignoriere überschüssige Anfragen
            cursor.forth
            cursor.item.undo.call ([])
        end
end
```



Erinnerung: das Beobachter-Muster



* aufgeschoben (*deferred*)

\+ wirksam (*effective*)

Erbt von

Kunde (benutzt)

Erinnerung: Ereignis-orientierte Programmierung mit Agenten

Die Subskribenten (Beobachter) registrieren sich bei Ereignissen:

Zurich_map.*left_click*.*subscribe* (**agent** *find_station*)

Der Herausgeber (Subjekt) löst ein Ereignis aus:

left_click.*publish* ([*x_posititon*, *y_position*])

Jemand (normalerweise der Herausgeber) definiert den Ereignis-Typ:

```
left_click : EVENT_TYPE [TUPLE [INTEGER, INTEGER]]
            -- „Linker Mausklick“-Ereignisse
    once
            create Result
    ensure
            exists: Result /= Void
    end
```

Das Undo/Redo- (bzw. Command-) Pattern

Wurde extensiv genutzt (z.B. in EiffelStudio und anderen Eiffel-Tools).

Ziemlich einfach zu implementieren.

Details müssen genau betrachtet werden (z.B. lassen sich manche Befehle nicht rückgängig machen).

Eleganter Gebrauch von O-O-Techniken

Nachteil: Explosion kleiner Klassen

Was wir gesehen haben

Menschen machen Fehler!

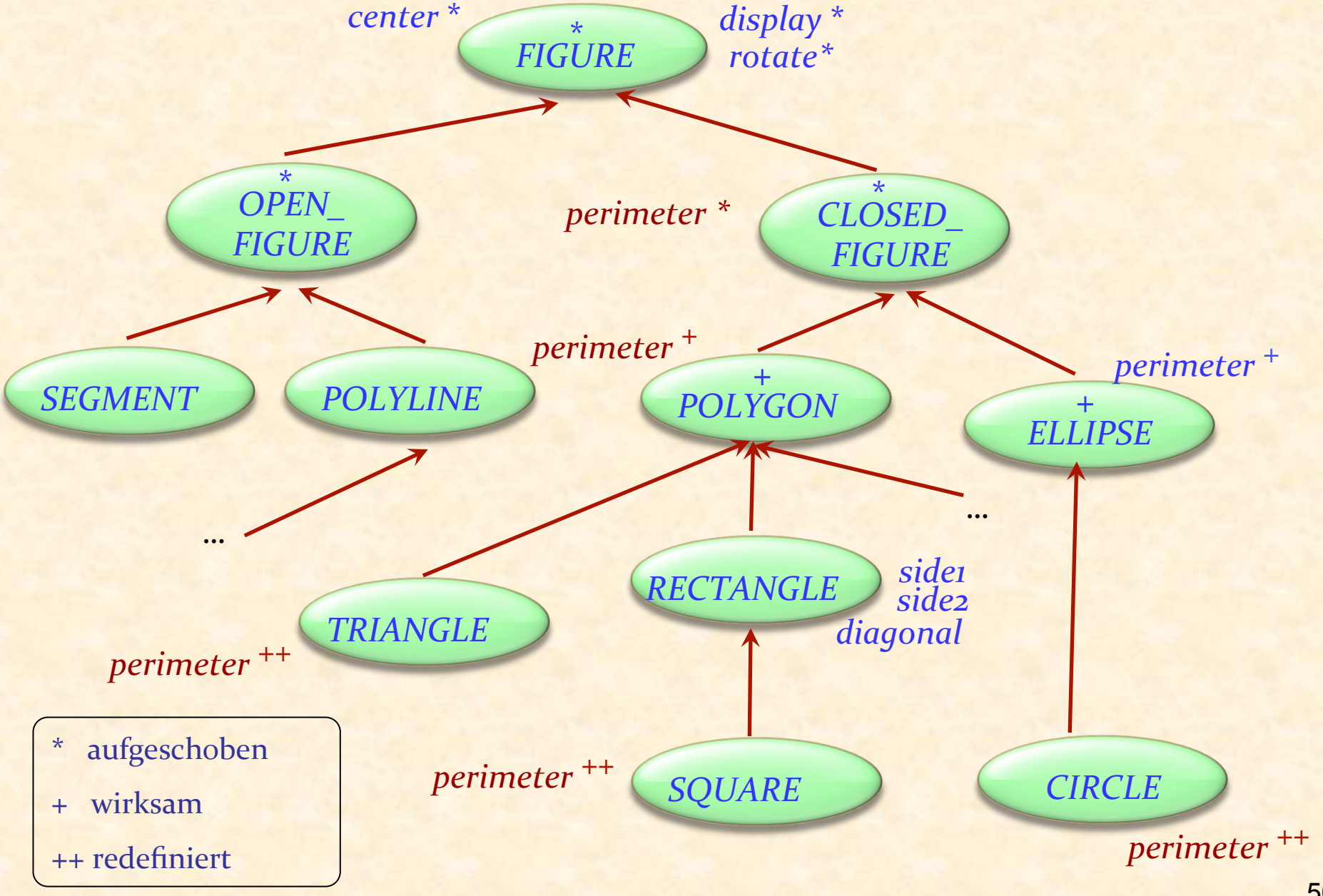
Auch wenn sie keine Fehler machen: sie wollen experimentieren. Undo/Redo unterstützt einen „trial and error“-Stil.

Undo/Redo-Pattern:

- Sehr nützlich in der Praxis
- Weit verbreitet
- Ziemlich einfach zu implementieren
- Exzellentes Beispiel von eleganten O-O-Techniken
- Mit Agenten noch besser!

Einen Typ erzwingen

Beispielhierarchie



Einen Typ erzwingen: Das Problem

```
bilder.store (“FN")
...
                -- Zwei Jahre später:
bilder := retrieved (“FN") – Siehe nachher
x := bilder.last                        -- [1]
print (x.diagonal )     -- [2]
```

Was ist daran falsch?

- Falls *x* als *RECTANGLE* deklariert ist, ist [1] ungültig
- Falls *x* als *FIGURE* deklariert ist, ist [2] ungültig

Einen Typ erzwingen: Der Objekt-Test

Zu prüfender Ausdruck

“ Object-Test Local”

```
if attached {RECTANGLE} bilder.retrieved ("FN") as r then
        print (r.diagonal)
        -- Tu irgendwas mit r, welches garantiert nicht
        -- Void und vom dynamischen Typ RECTANGLE ist.
else
        print ("Too bad.")
end
```

SCOPE der Object-Local

Früherer Mechanismus: Zuweisungsversuch

```
f : FIGURE
r : RECTANGLE
...
bilder.retrieve ("FN")
f := bilder.last
```

r ?= f

```
if r /= Void then
        print (r.diagonal)
else
        print ("Too bad.")
end
```

Zuweisungsversuch (veraltetes Konstrukt)

$x \ ?= y$

mit

$x : A$

Semantik:

- Falls y an ein Objekt gebunden ist, dessen Typ konform zu A ist: Ausführung einer normalen Referenzzuweisung
- Sonst: Mache x Void

Verträge
&
Vererbung

Verträge und Vererbung

Problem: Was passiert bei Vererbung mit

- Klasseninvarianten?
- Vor- und Nachbedingungen von Routinen?

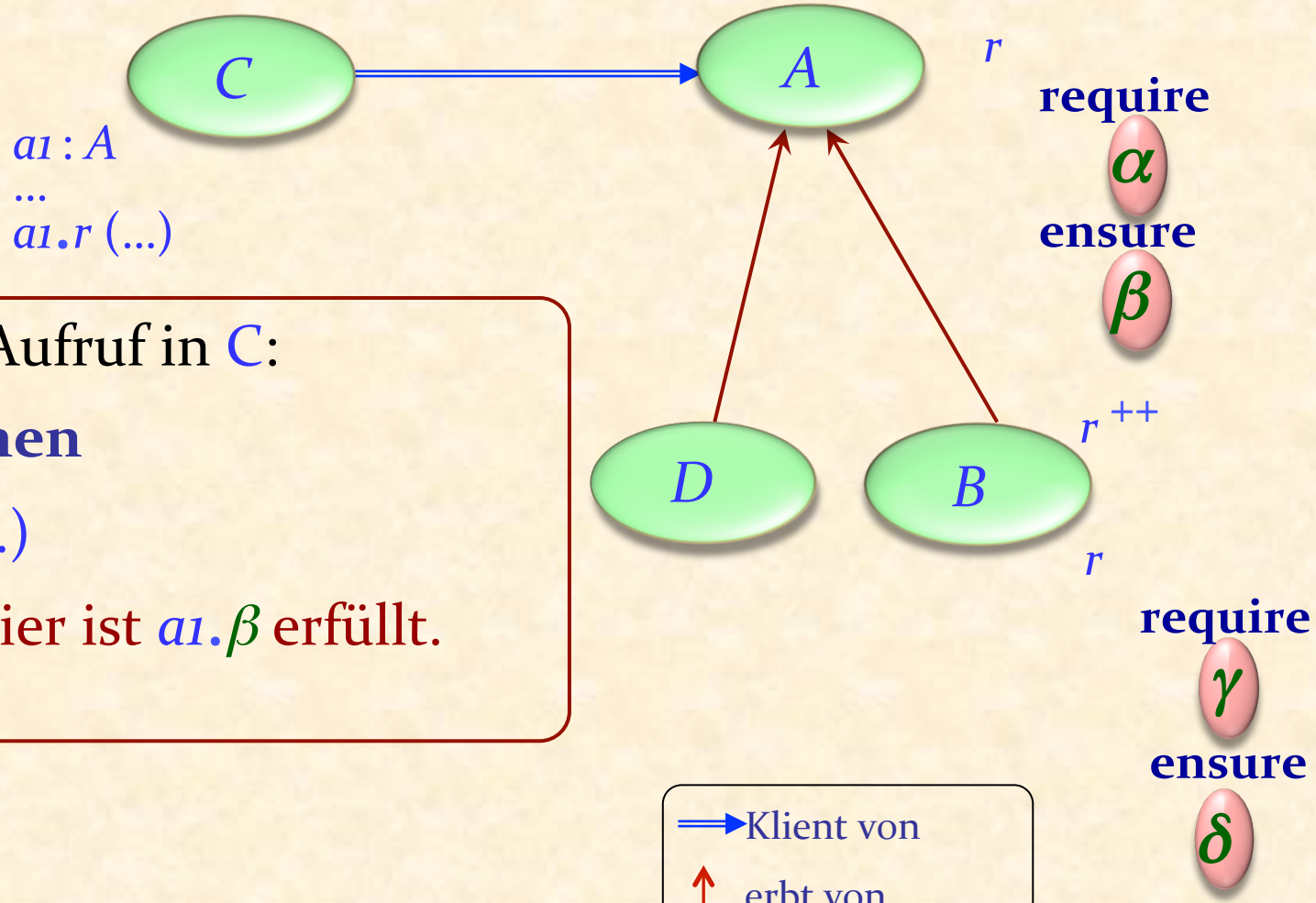
Invarianten

Vererbungsregel für Invarianten:

- Die Invariante einer Klasse beinhaltet automatisch die Invarianten aller Vorfahren, „ver-und-et“.

Die kumulierten Invarianten sind in der flachen Ansicht und der Schnittstellen-Ansicht in Eiffelstudio ersichtlich.

Verträge und Vererbung



Korrekter Aufruf in C:

if *a1*.α **then**

a1.*r* (...)

-- Hier ist *a1*.β erfüllt.

end

Neudeklarierungsregel für Zusicherungen

Wenn eine Routine neu deklariert wird, darf man nur:

- Die Vorbedingung beibehalten oder schwächen
- Die Nachbedingung beibehalten oder stärken

Neudeklarierungsregel für Zusicherungen in Eiffel

Eine simple Sprachregel genügt!

Redefinierte Versionen dürfen keine Vertragsklausel haben (Dann bleiben die Zusicherungen gleich) oder

require else *new_pre*
ensure then *new_post*

Die resultierenden Zusicherungen sind:

- *original_precondition* **or** *new_pre*
- *original_postcondition* **and** *new_post*

Mehrfache Vererbung

Abstraktionen kombinieren

Gegeben sind die Klassen

- EISENBAHNWAGEN, RESTAURANT

Wie würden Sie eine Klasse SPEISEWAGEN implementieren?

Beispiele von Mehrfachvererbungen

Separate Abstraktionen kombinieren:

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

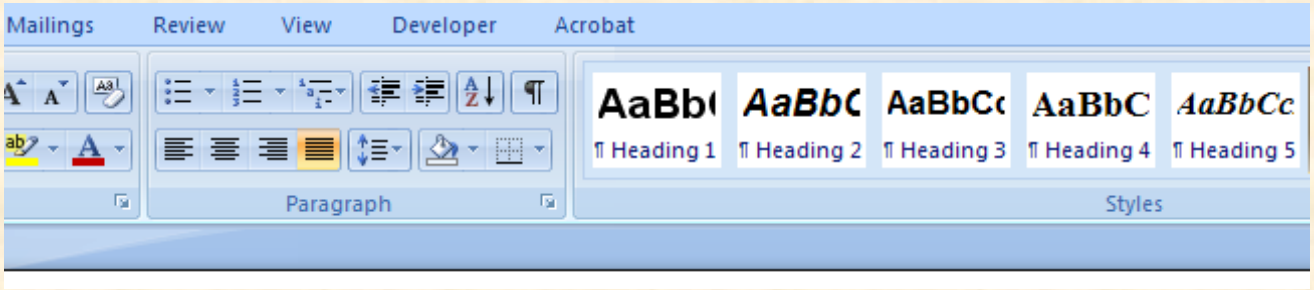
Warnung

Vergessen Sie alles, was Sie gehört haben!

Mehrfachvererbung ist **nicht** das Werk des Teufels

Mehrfachvererbung schadet ihren Zähnen **nicht**

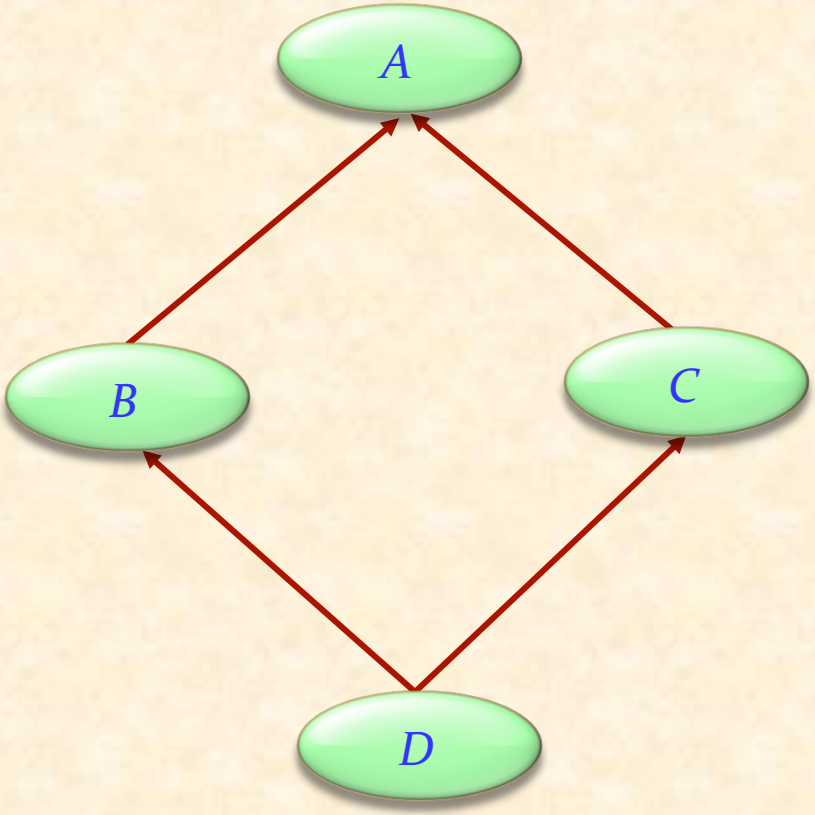
(Auch wenn Microsoft Word sie scheinbar nicht mag:



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

)

Dies ist **wiederholte** Vererbung, nicht Mehrfachvererbung

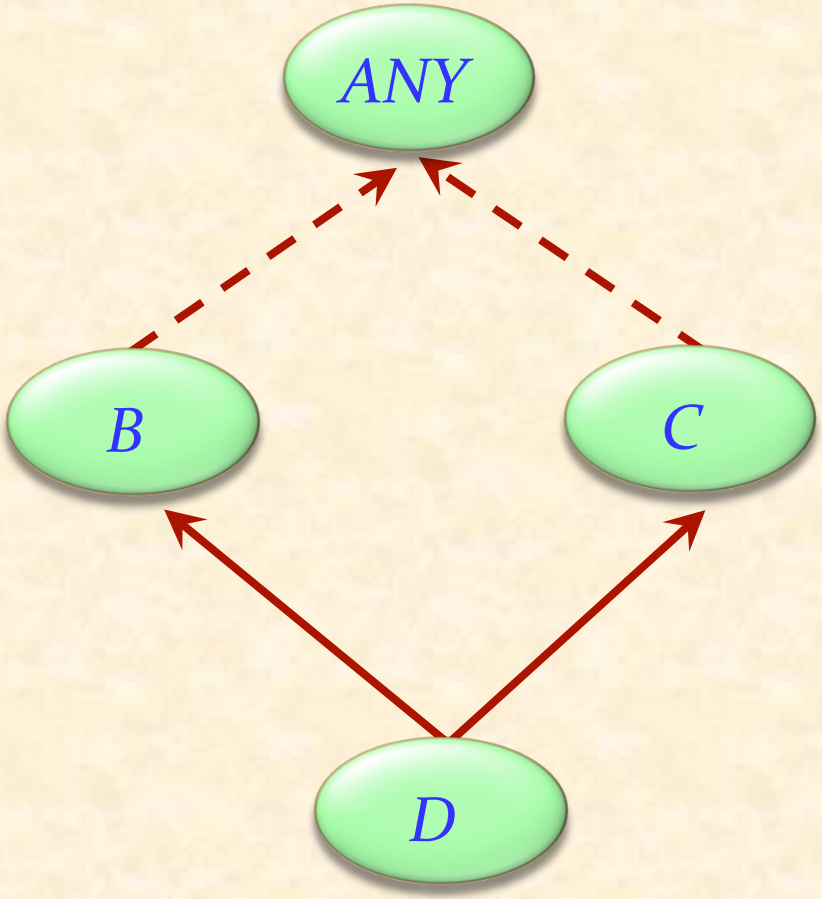


Nicht der allgemeine Fall

(Obwohl es häufig vorkommt; warum?)

Implizite wiederholte Vererbung

In Eiffel, vierfache Vererbung verursacht wiederholte Vererbung:



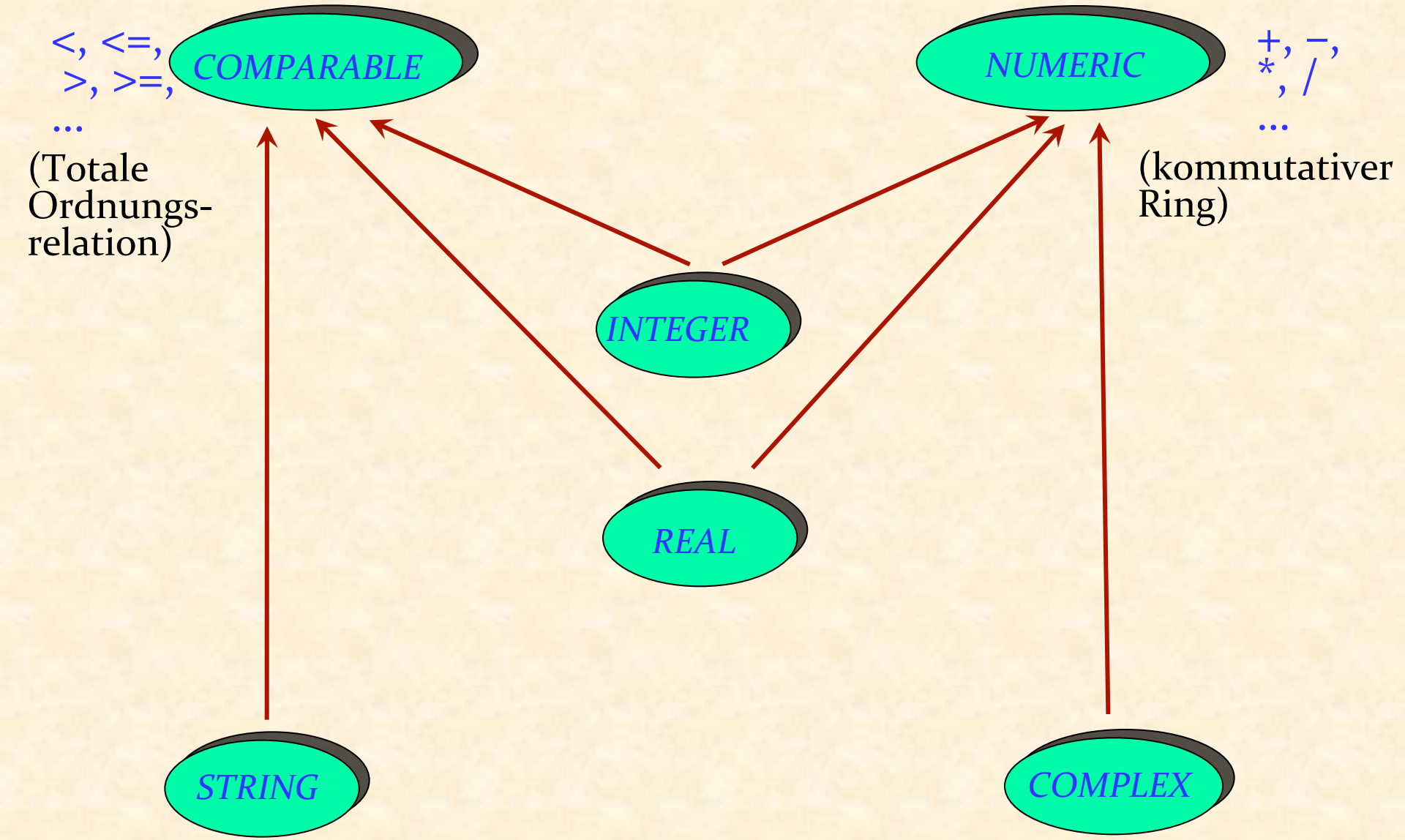
Mehrfachvererbung: die Java und .NET Lösung

Nur Einfachvererbung für Klassen

Mehrfachvererbung von **Schnittstellen**

Eine Schnittstelle entspricht einer vollständig aufgeschobenen Klasse, ohne Implementationen (ohne **do** Klauseln) und Attribute (und auch ohne Verträge)

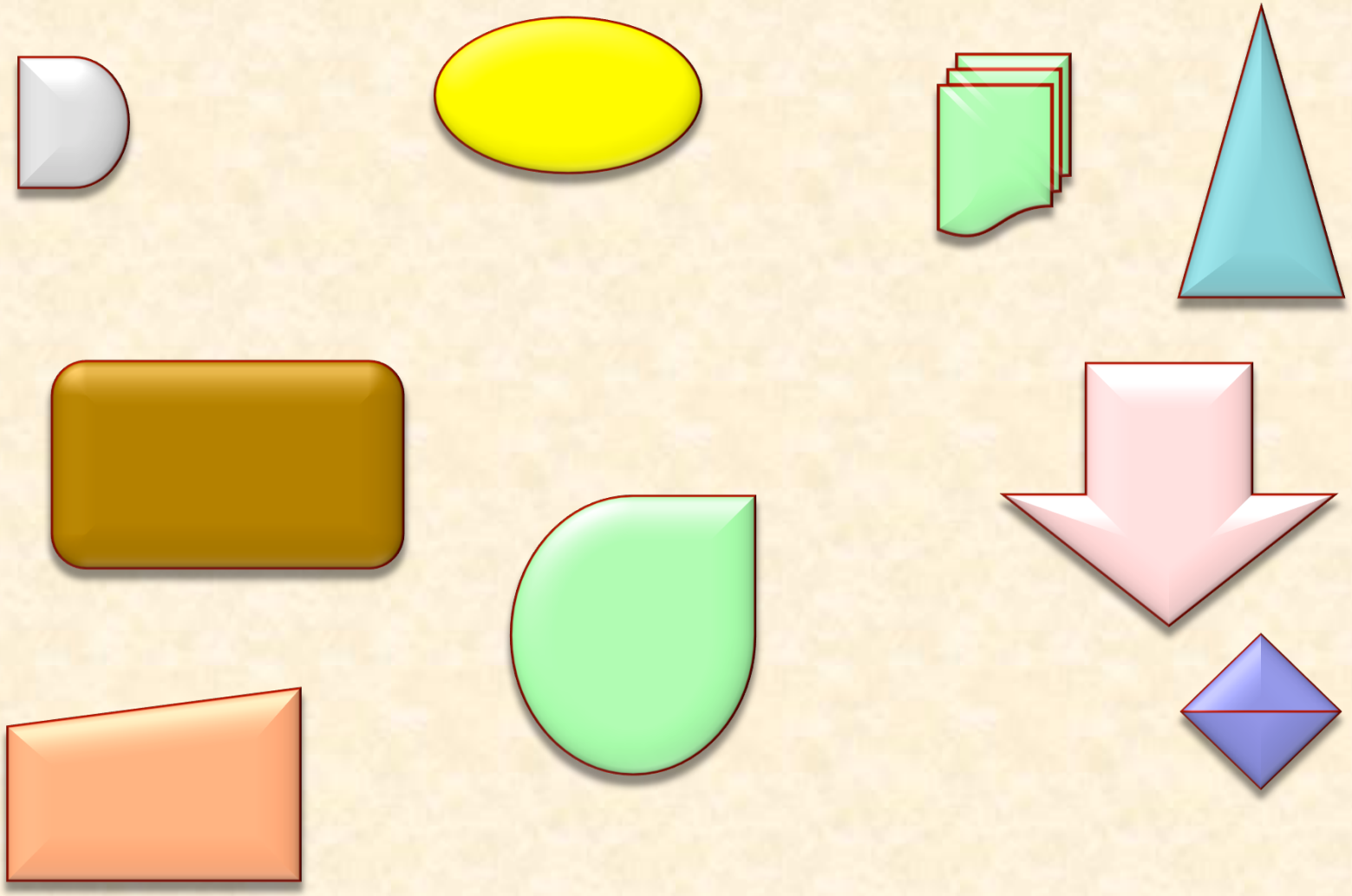
Mehrfachvererbung: Abstraktionen kombinieren



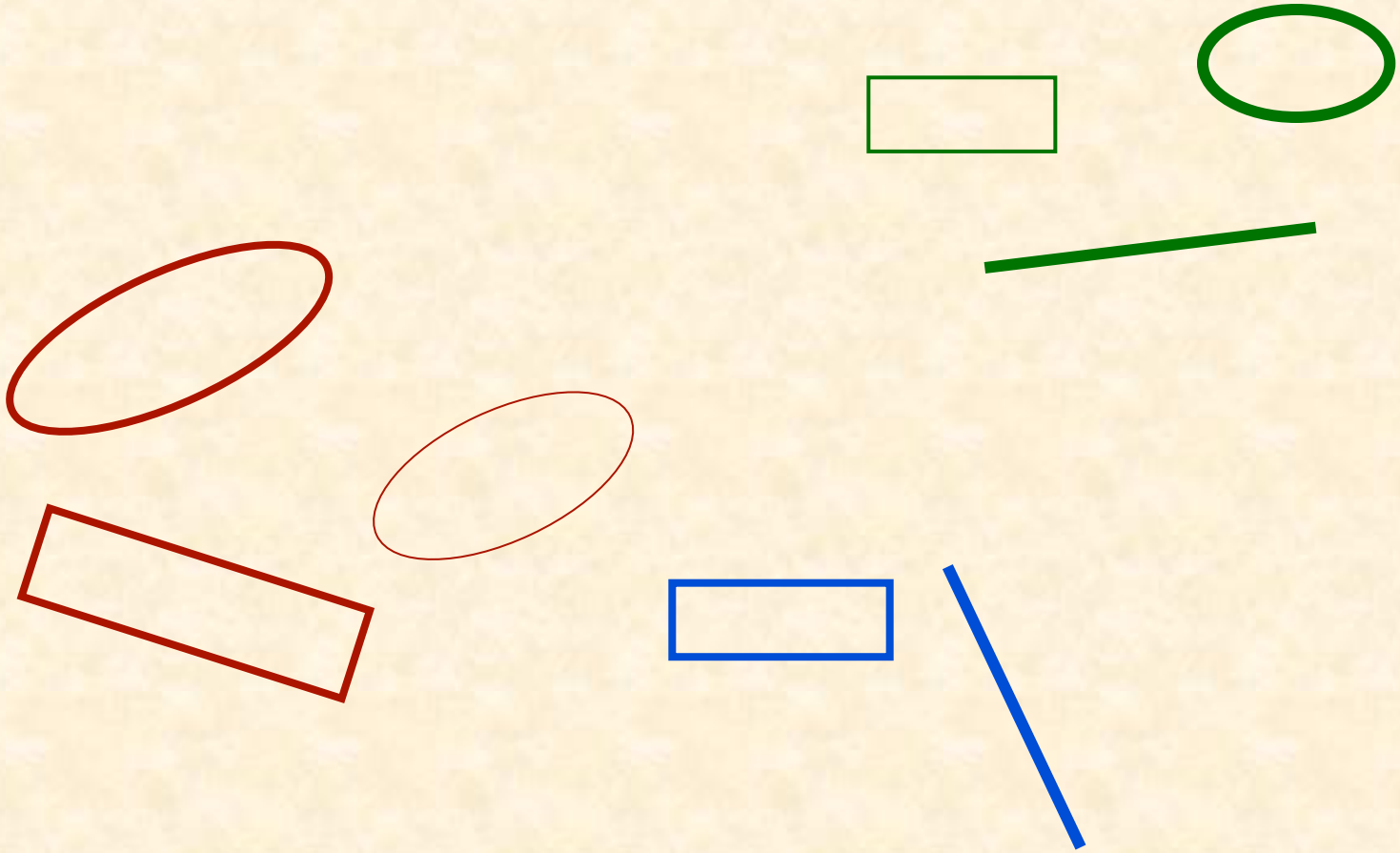
Wie schreiben wir *COMPARABLE* ?

```
deferred class COMPARABLE feature

    less alias "<" (x : COMPARABLE): BOOLEAN
        deferred
        end

    less_equal alias "<=" (x : COMPARABLE): BOOLEAN
        do
            Result := (Current < x or (Current = x))
        end

    greater alias ">" (x : COMPARABLE): BOOLEAN
        do Result := (x  < Current) end

    greater_equal alias ">=" (x : COMPARABLE): BOOLEAN
        do Result := (x  <= Current) end
end
```

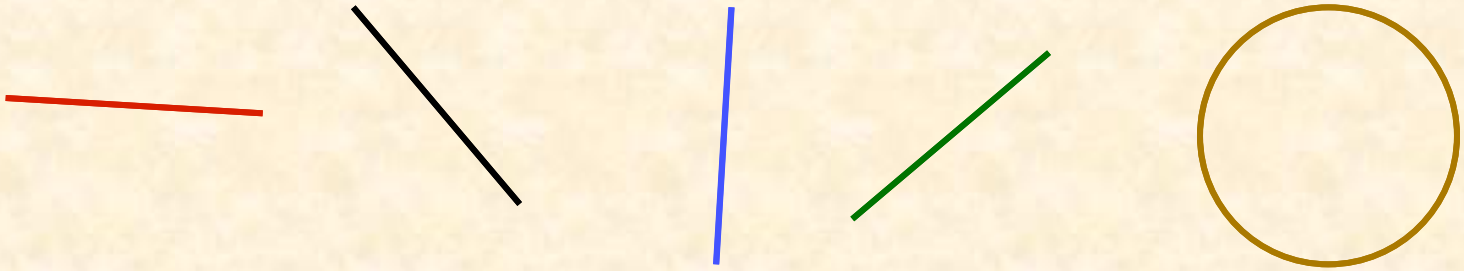
Figuren in einem Graphischen Editor



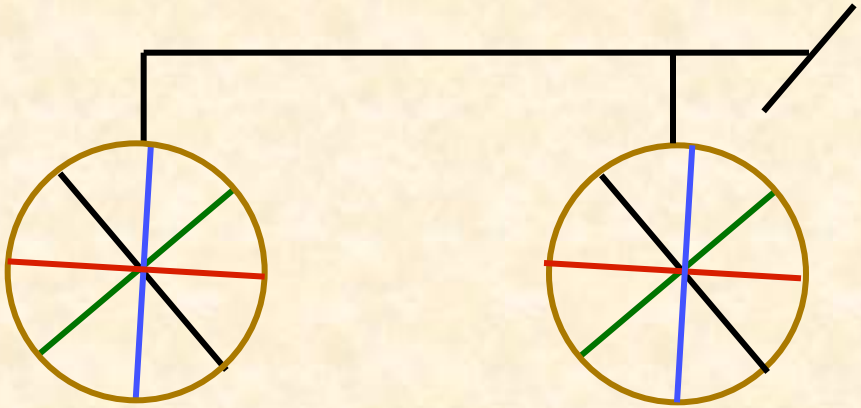
Zusammengesetzte Figuren



Mehrfachvererbung: Zusammengesetzte Figuren

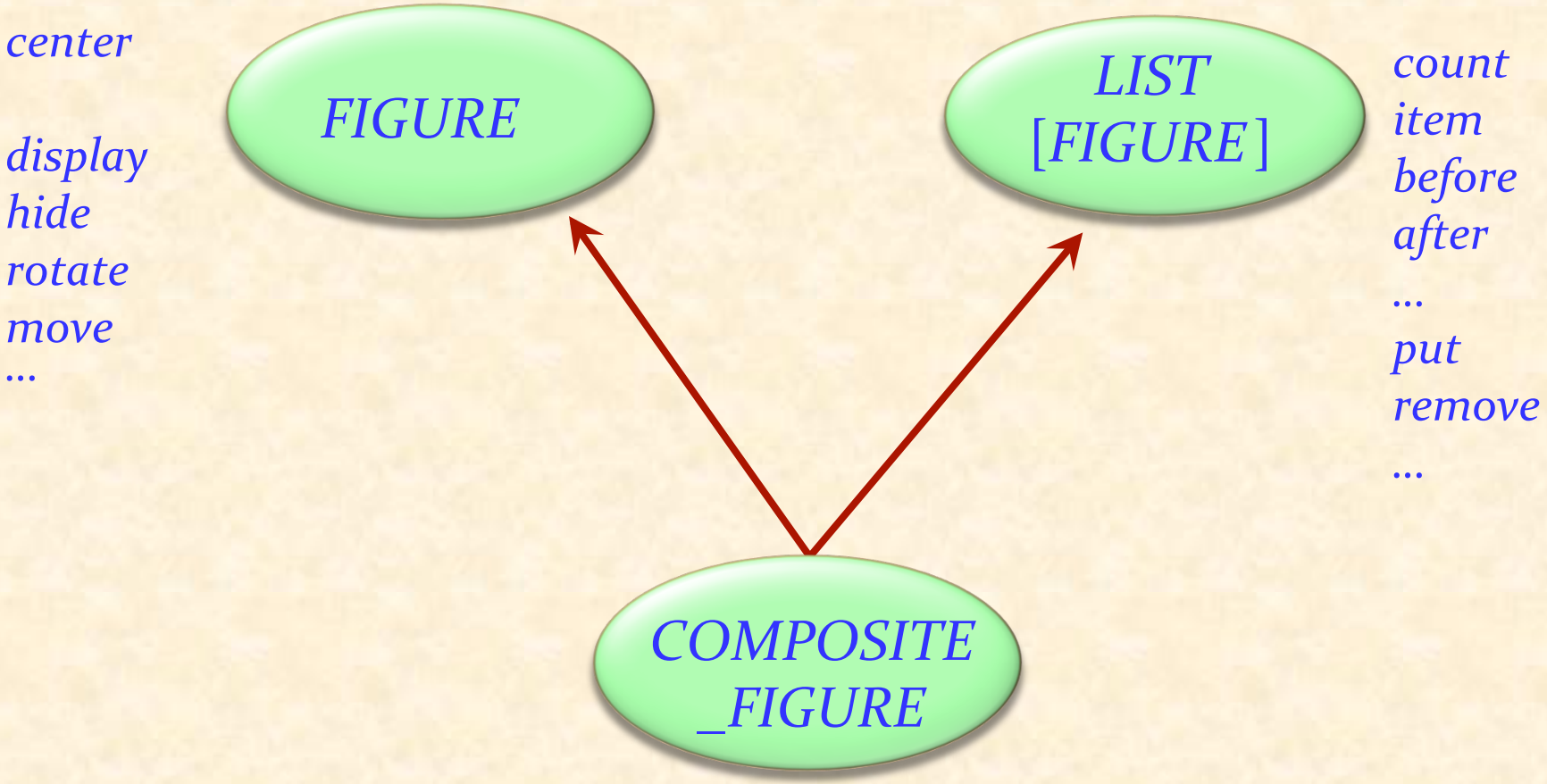


Einfache Figuren

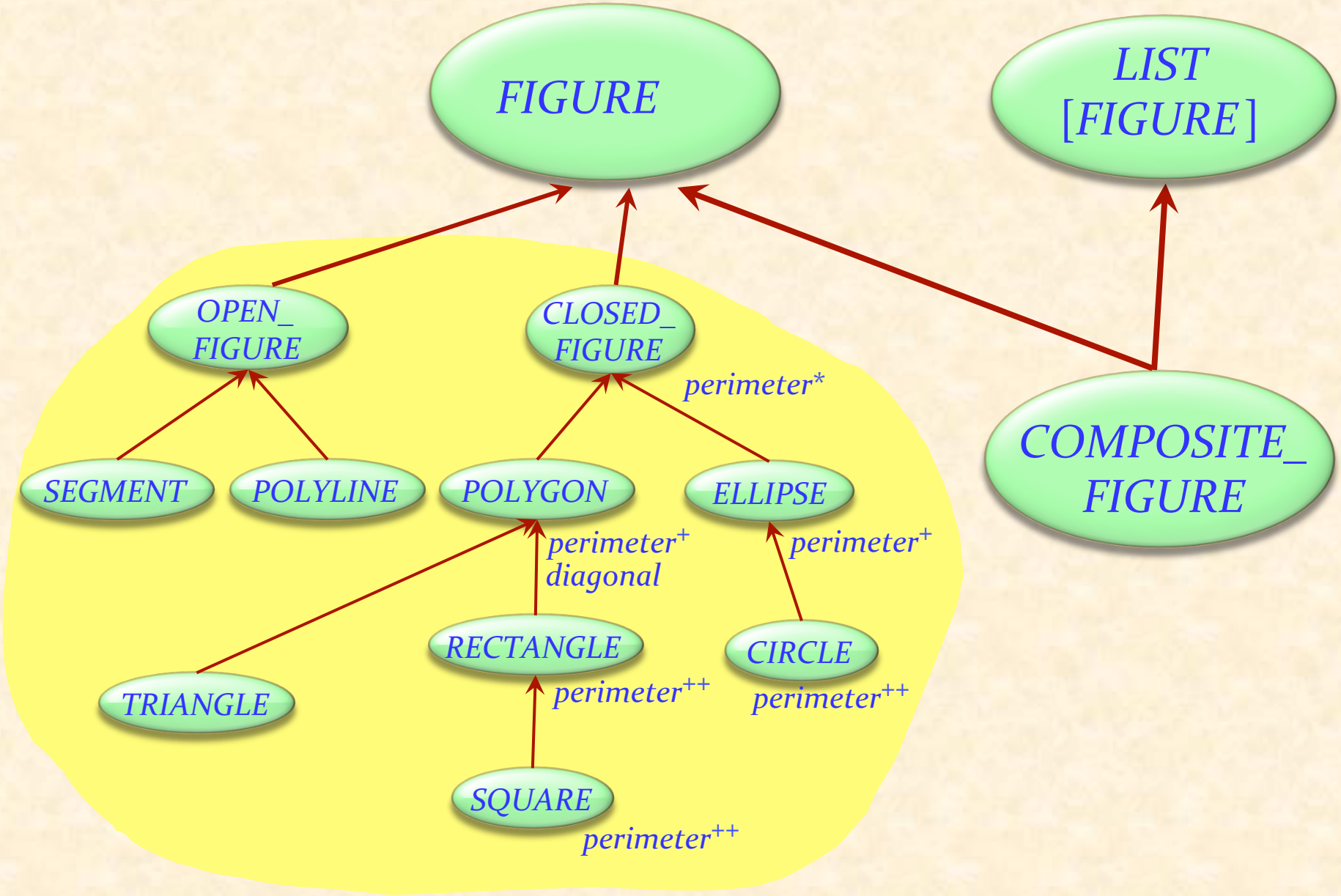


Eine zusammengesetzte Figur

Den Begriff der zusammengesetzten Figur definieren



In der allgemeinen Struktur



(Erinnerung) Mit polymorphen Datenstrukturen arbeiten

(aus Lektion 11)

bilder: LIST [FIGURE]

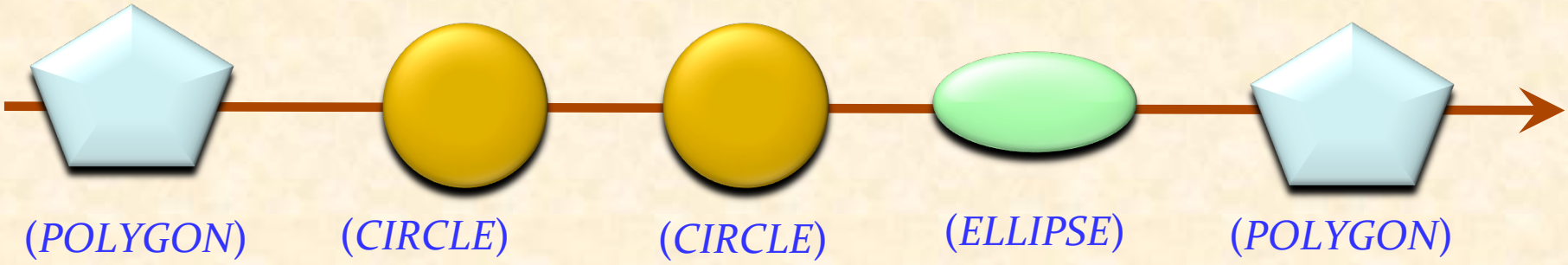
...

across *bilder* **as** *c* **loop**

c.item.display

end

Dynamische Binden



(Erinnerung) Definition: Polymorphie, angepasst

(aus Lektion 11)

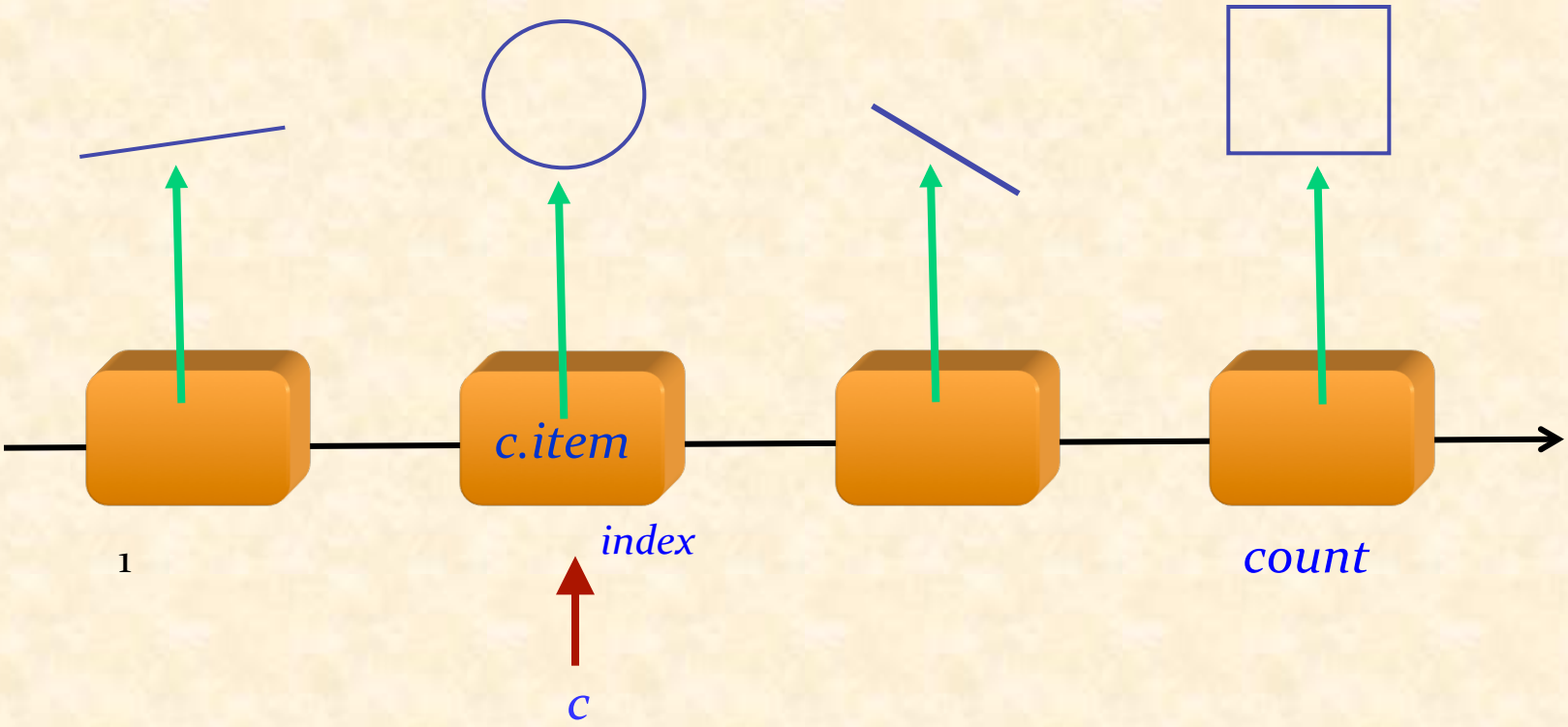
Eine **Bindung** (Zuweisung oder Argumentübergabe) ist **polymorph**, falls ihre Zielvariable und der Quellausdruck verschiedene Typen haben

Eine **Entität** oder ein **Ausdruck** ist **polymorph**, falls sie/er zur Laufzeit — in Folge einer polymorphen Bindung — zu einem Objekt eines anderen Typs gebunden werden

Eine **Container-Datenstruktur** ist **polymorph**, falls sie Referenzen zu Objekten verschiedener Typen enthalten kann

Polymorphie ist die Existenz dieser Möglichkeiten.

Eine zusammengesetzte Figur als Liste



Zusammengesetzte Figuren

```
class COMPOSITE_FIGURE inherit
        FIGURE
        LIST [FIGURE]
feature
        display
                        -- Jede einzelne Figur der Reihenfolge
                        -- nach anzeigen.
                do
                        across Current as c loop
                                c.item.display
                        end
        end
        ... Ähnlich für move, rotate etc. ...
end
```

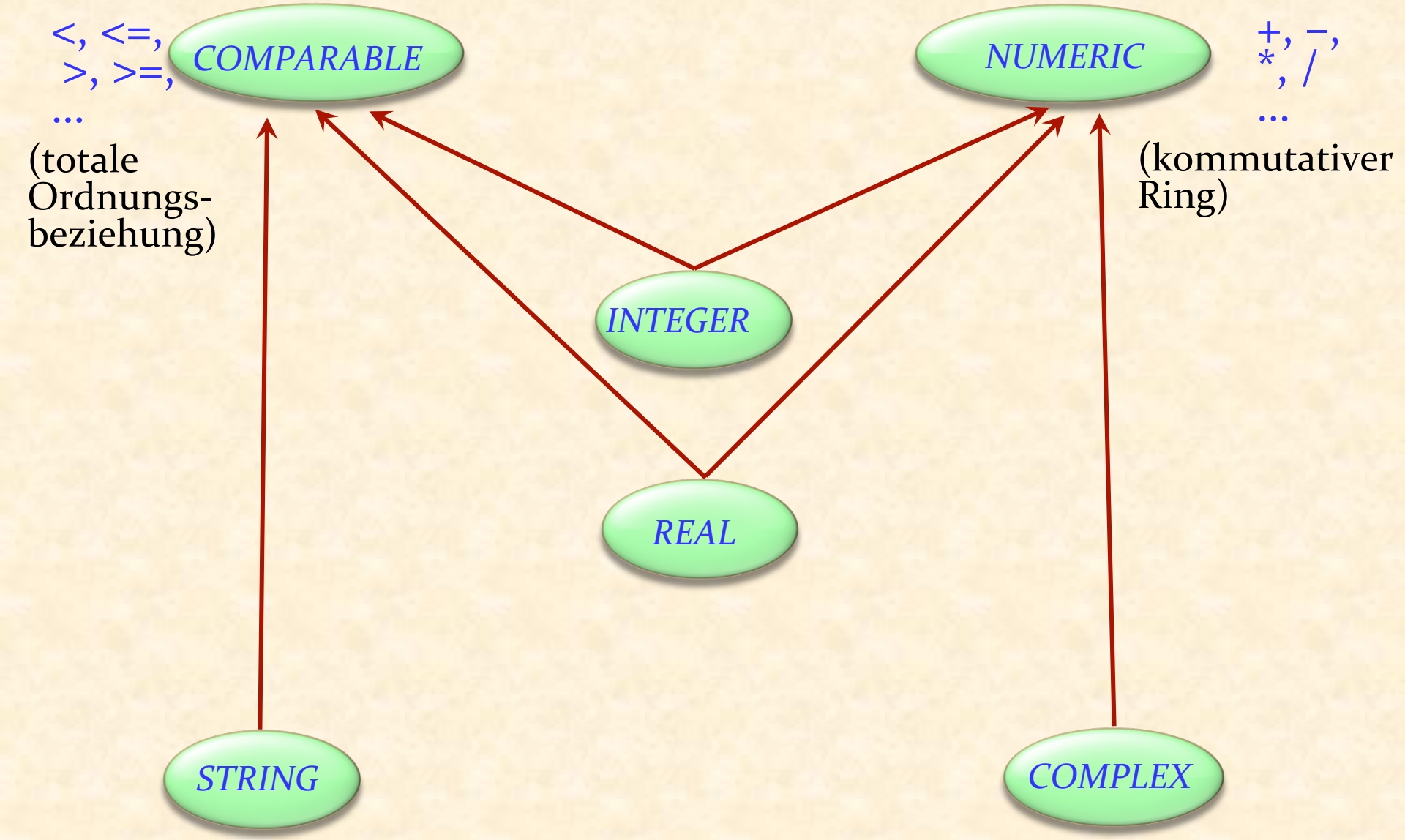
Benötigt dynamisches Binden

Eine Abstraktionsebene höher gehen

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

Benutzen Sie dafür Agenten

Mehrfachvererbung: Abstraktionen kombinieren



Die Lösung von Java und C#

Keine Mehrfachvererbung für Klassen

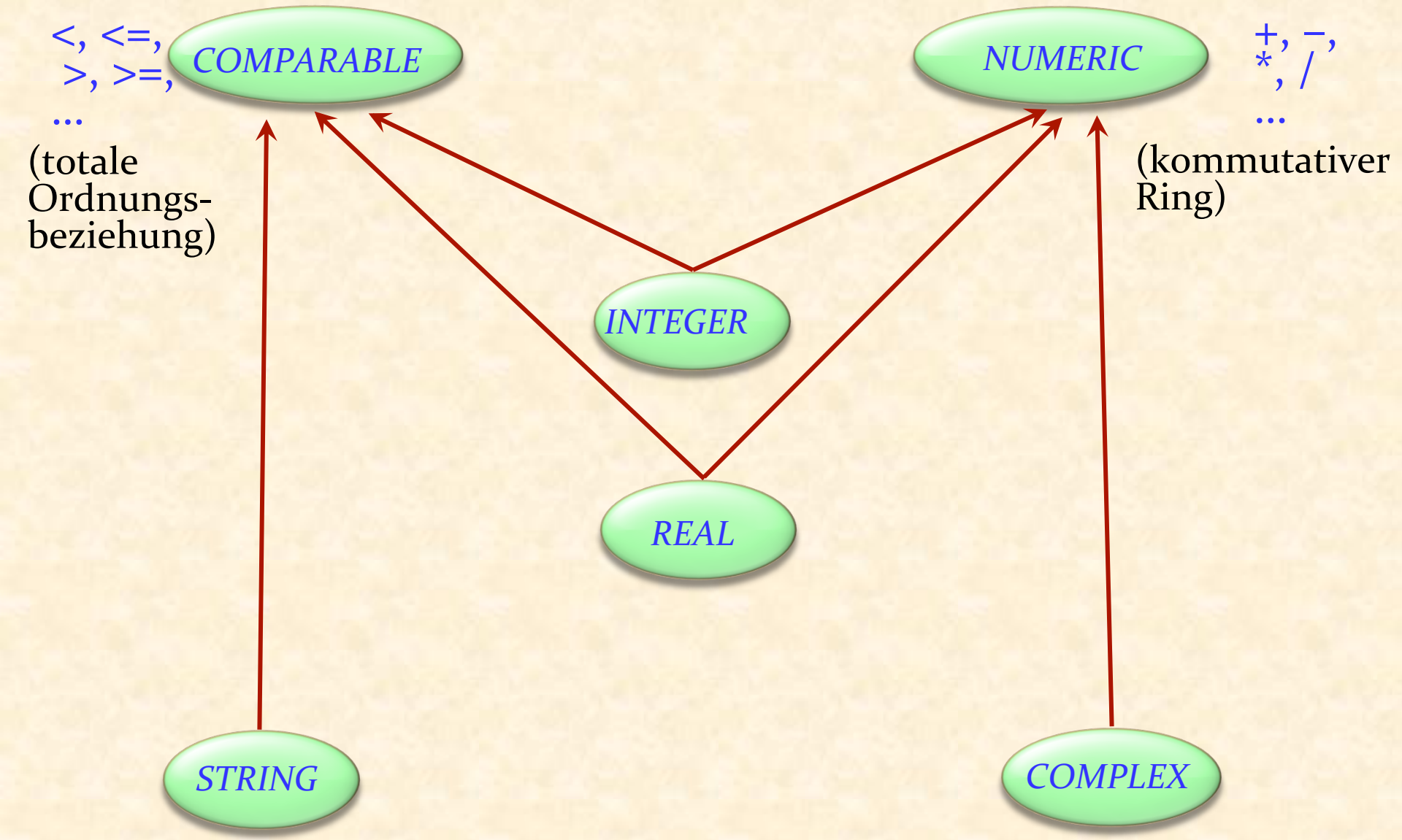
"Interface": Nur Spezifikationen (aber ohne Verträge)

- 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 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
```

Die Moral dieses Beispiels

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

Beispiel:

i_th (*i* : *INTEGER*): *G*

-- Element mit Index `*i*'.

do

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

end

Nicht-konforme Vererbung

class

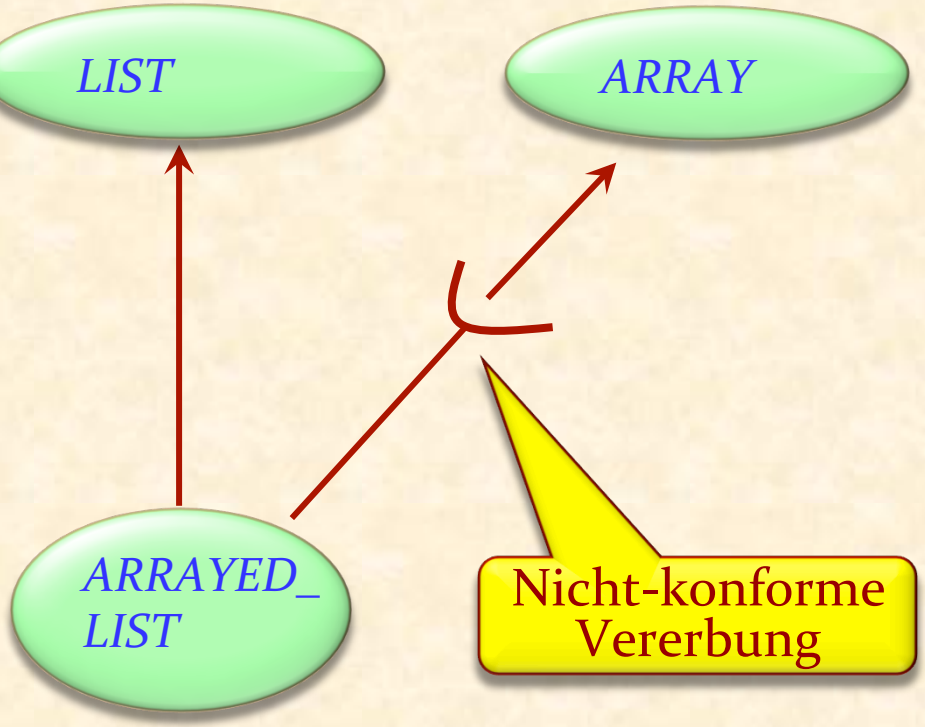
ARRAYED_LIST [*G*]

inherit

LIST [*G*]

inherit {*NONE*}

ARRAY [*G*]

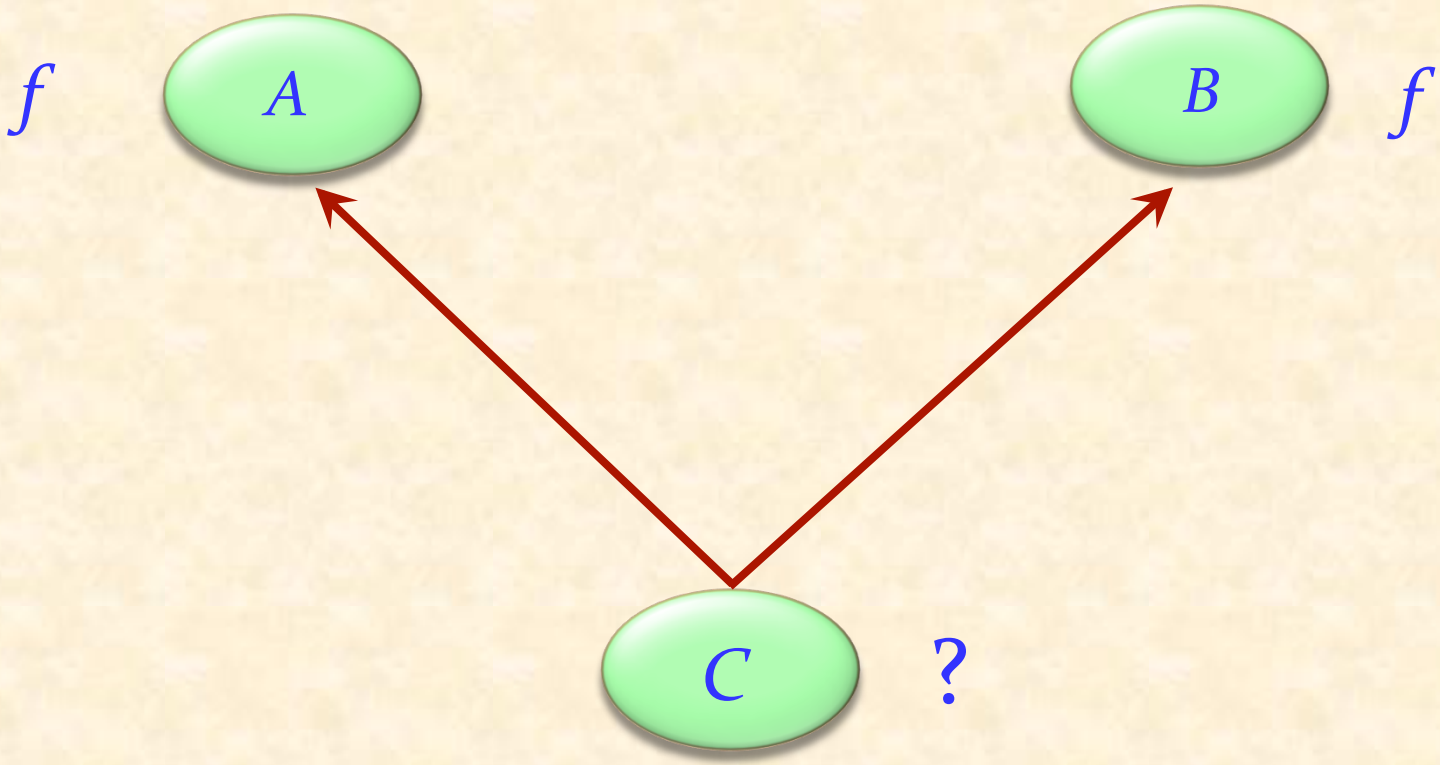


feature

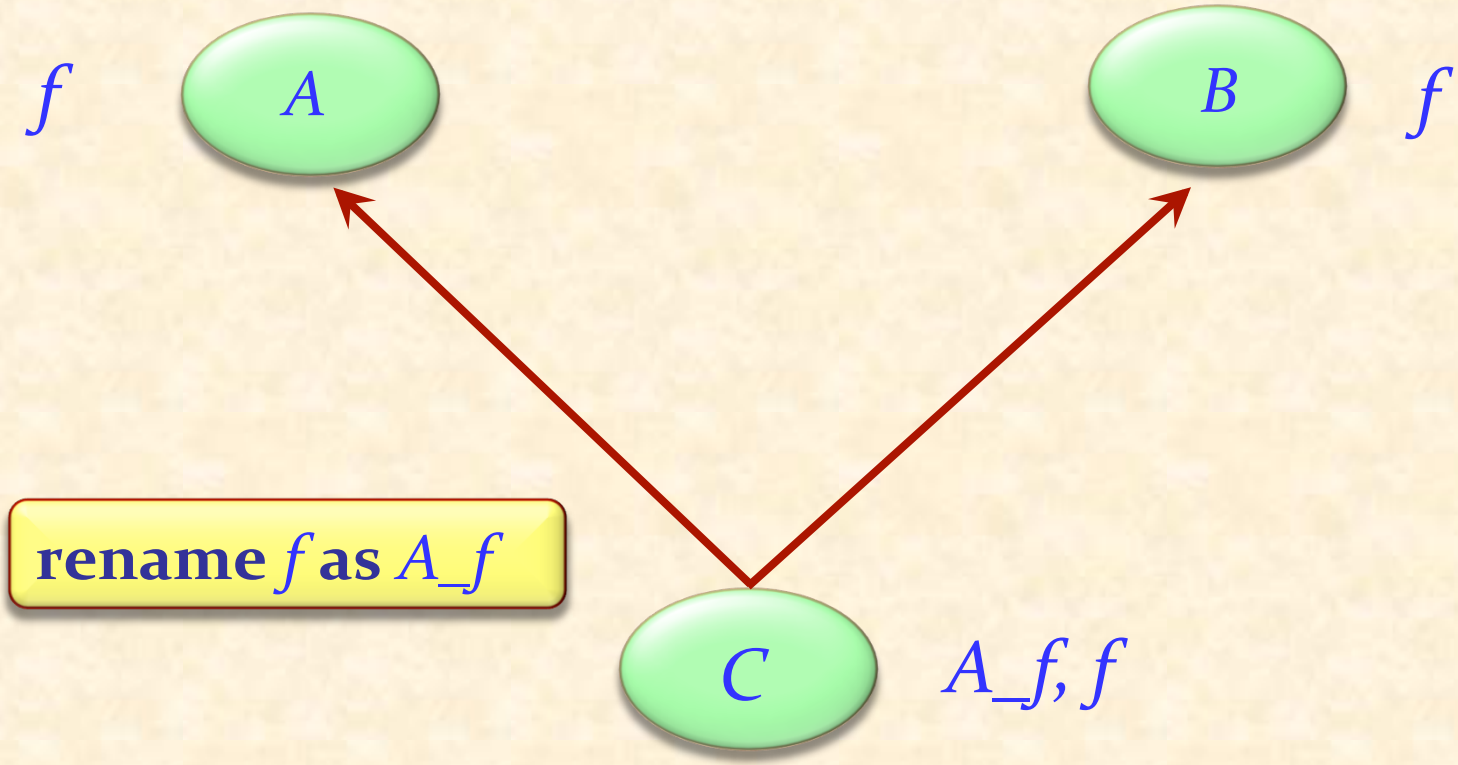
... Implementiere *LIST* -Features mit *ARRAY*- Features ...

end

Mehrfachvererbung: Namenskonflikte



Namenskonflikte auflösen




```
class C inherit
        A rename f as A_f end
        B
...
```

Konsequenzen des Umbenennens

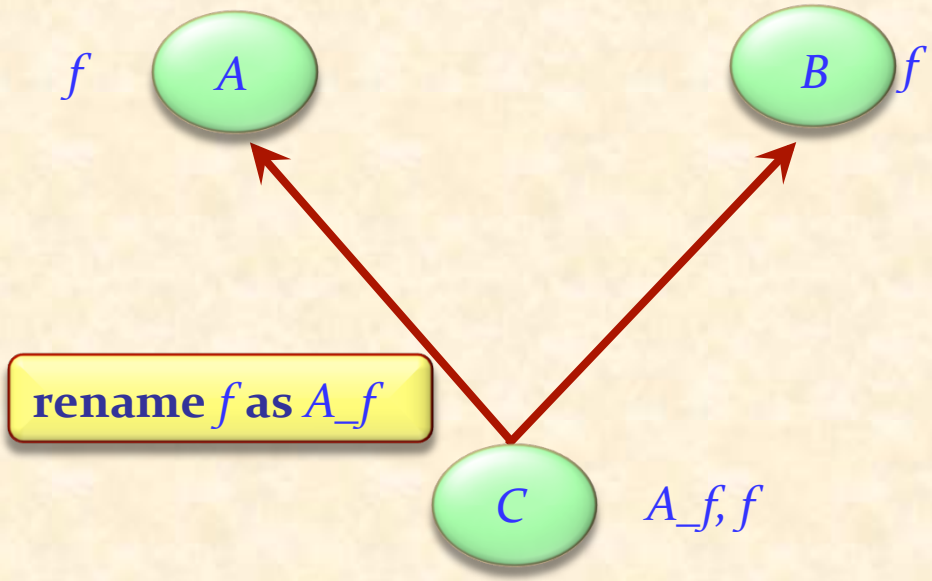
a1 : A
b1 : B
c1 : C
...

c1.f

c1.A_f

a1.f

b1.f



Ungültig:

- *a1.A_f*
- *b1.A_f*

Redefinition und Umbenennen

Redefinition ändert das Feature und behält seinen Namen

Umbenennen behält das Feature und ändert seinen Namen

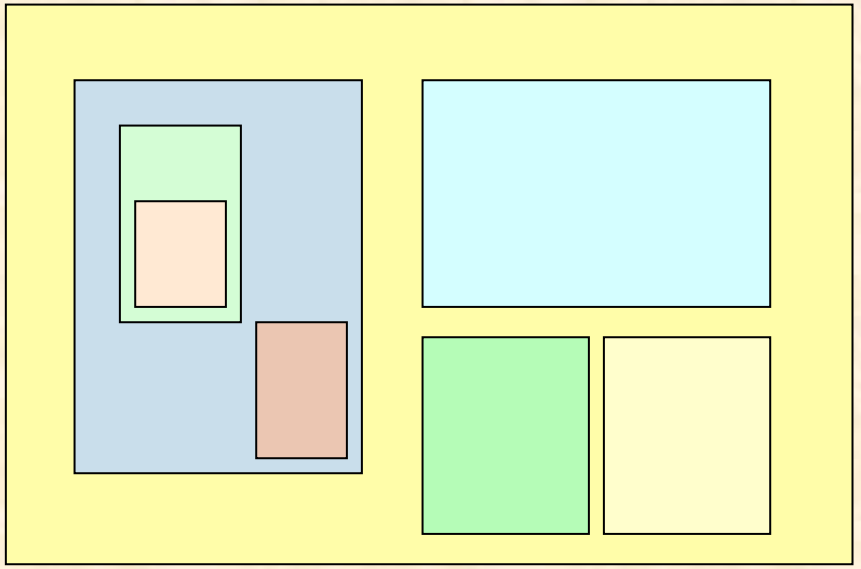
Es ist möglich beide zu kombinieren:

```
class B inherit
        A
                rename f as A_f
                redefine A_f
                end
...
```

Noch eine Anwendung von Umbenennungen

Eine (lokal) bessere Terminologie ermöglichen.

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

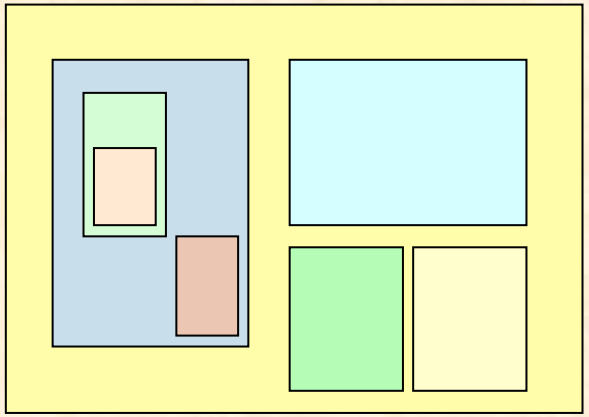


Umbenennungen, um die Terminologie zu verbessern

"Graphische" Features: *height, width, x, y, change_height, change_width, 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
```



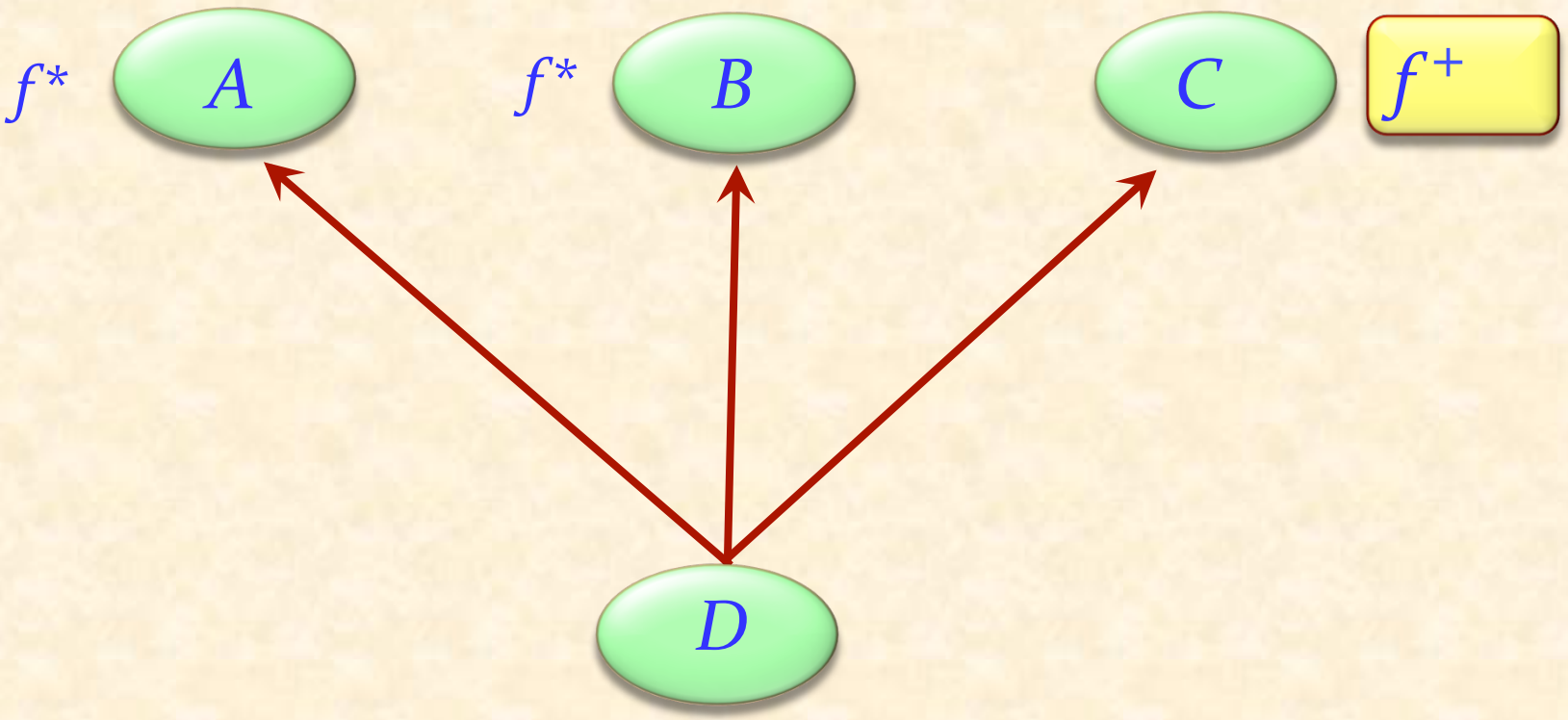
ABER: Siehe Stilregeln betreffend einheitlichen Featurenamen

Sind alle Namenskonflikte schlecht?

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)

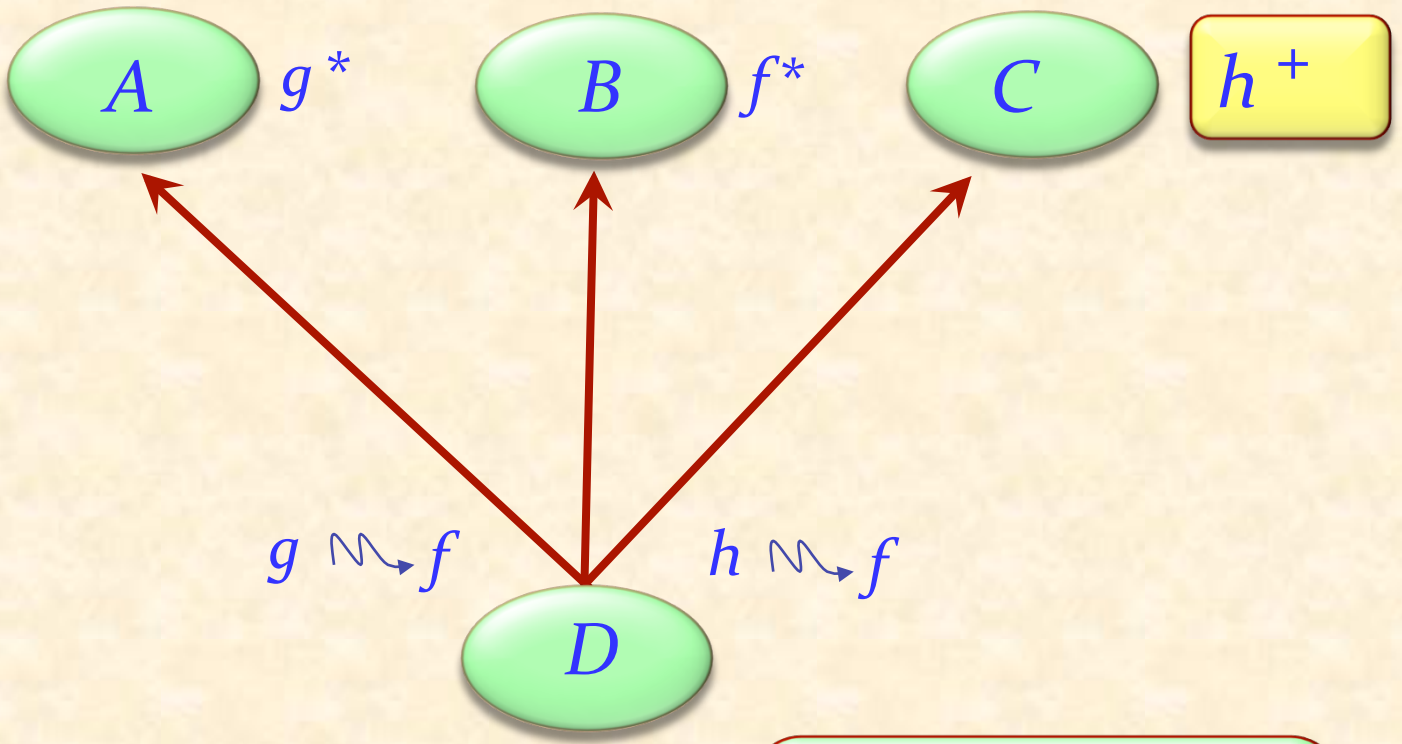
Features verschmelzen



* aufgeschoben
\+ wirksam

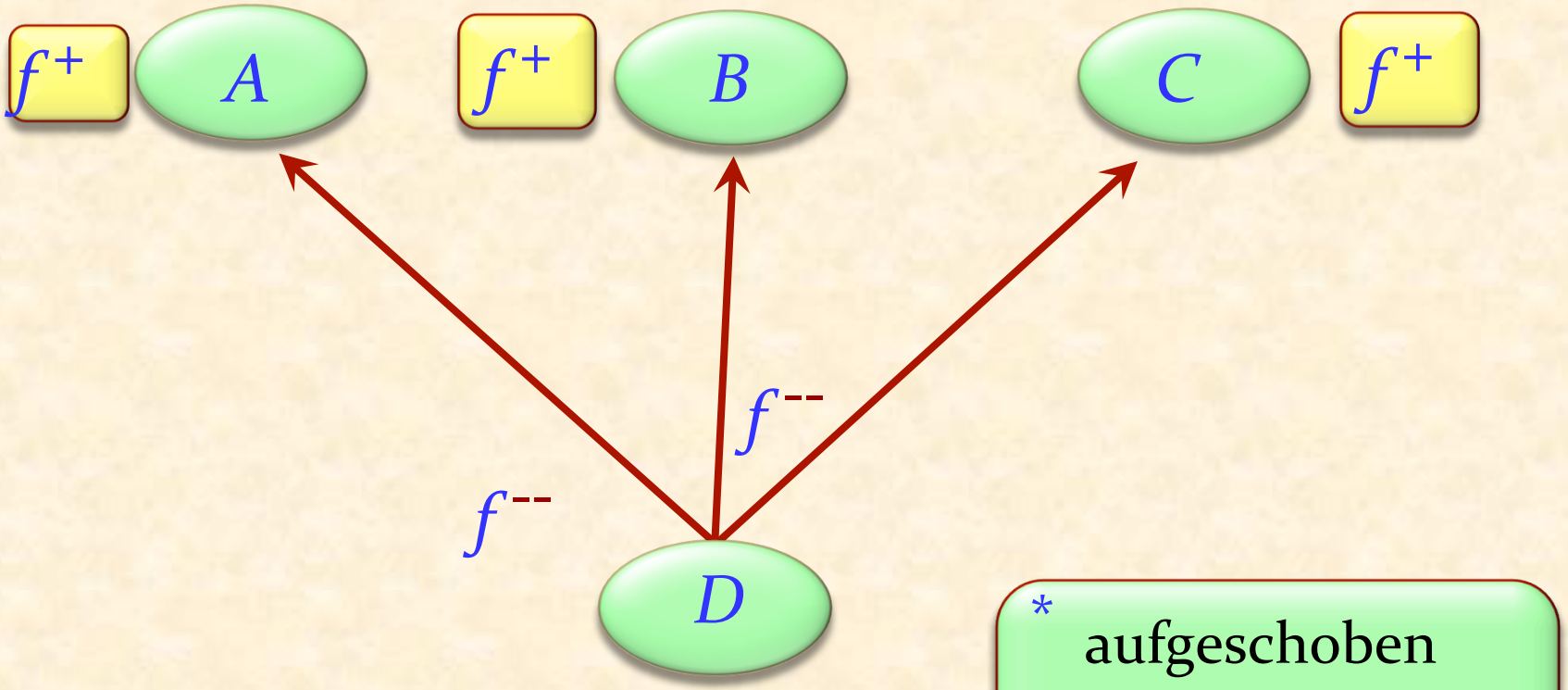
Features verschmelzen: Mit verschiedenen Namen

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



* aufgeschoben
\+ wirksam
↝ Umbenennung

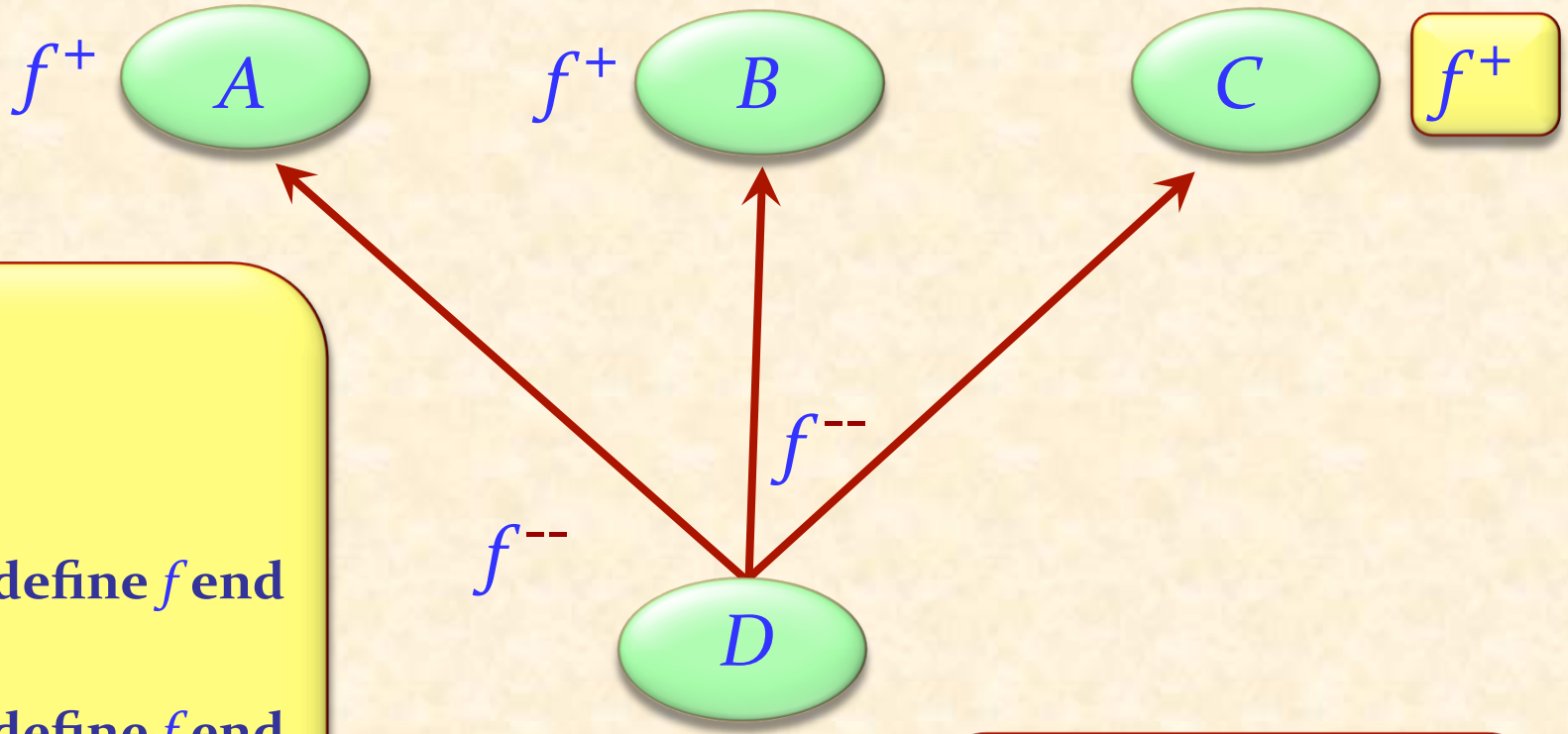
Features verschmelzen: wirksame Features



Undefinition

```
deferred class
    T
inherit
    S
        undefine f end
feature
        ...
end
```

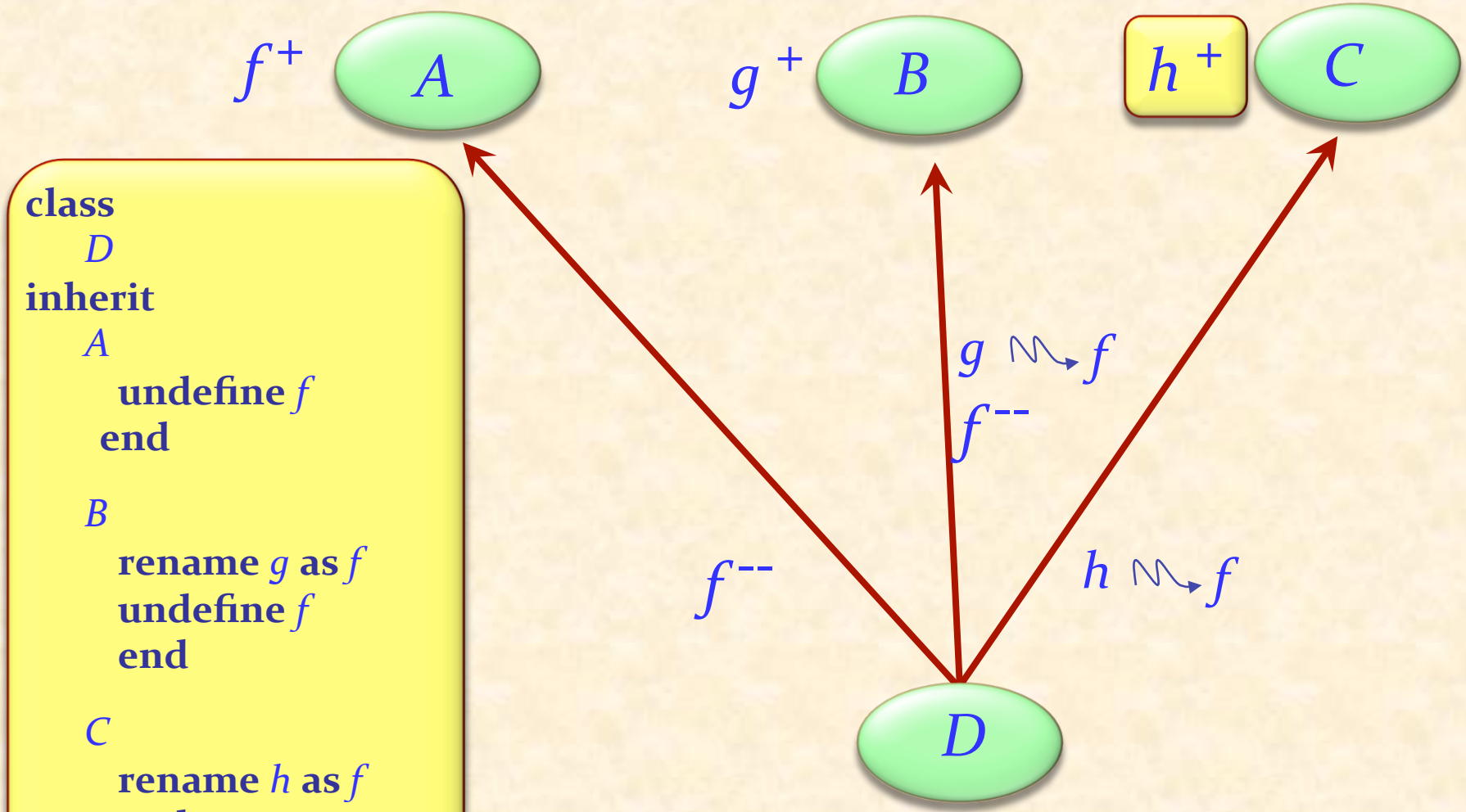
Verschmelzen durch Undefinition




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

* aufgeschoben
\+ wirksam
-- undefiniert

Verschmelzen von Features mit unterschiedlichen Namen



Akzeptable Namenskonflikte

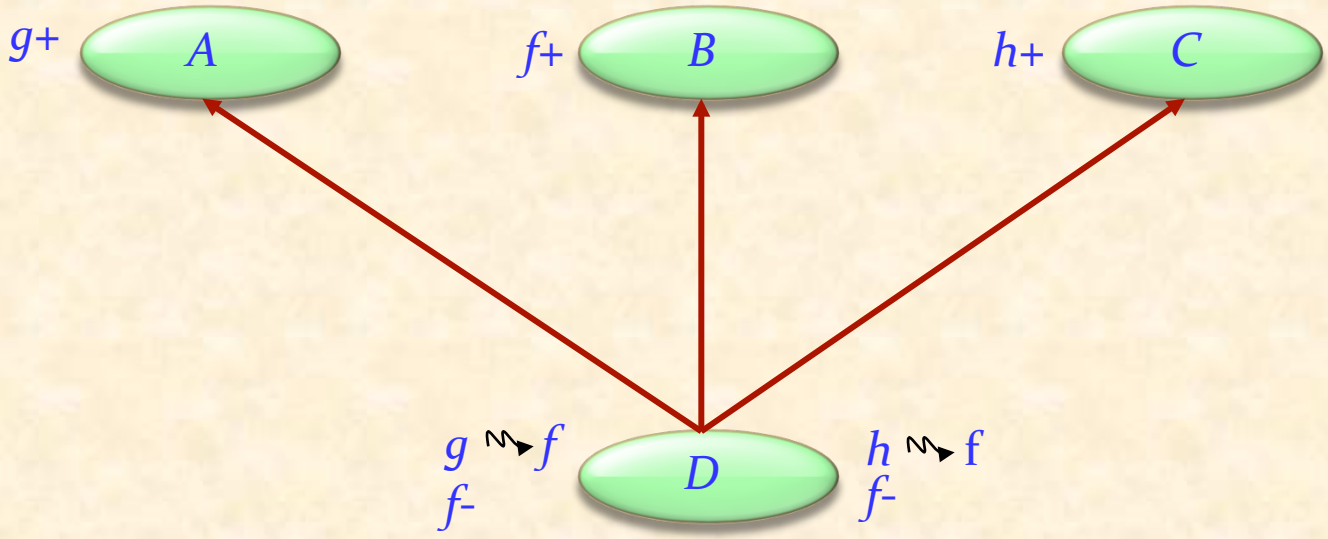
Wenn geerbte Features alle den gleichen Namen haben, besteht kein schädlicher Namenskonflikt, falls:

- 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

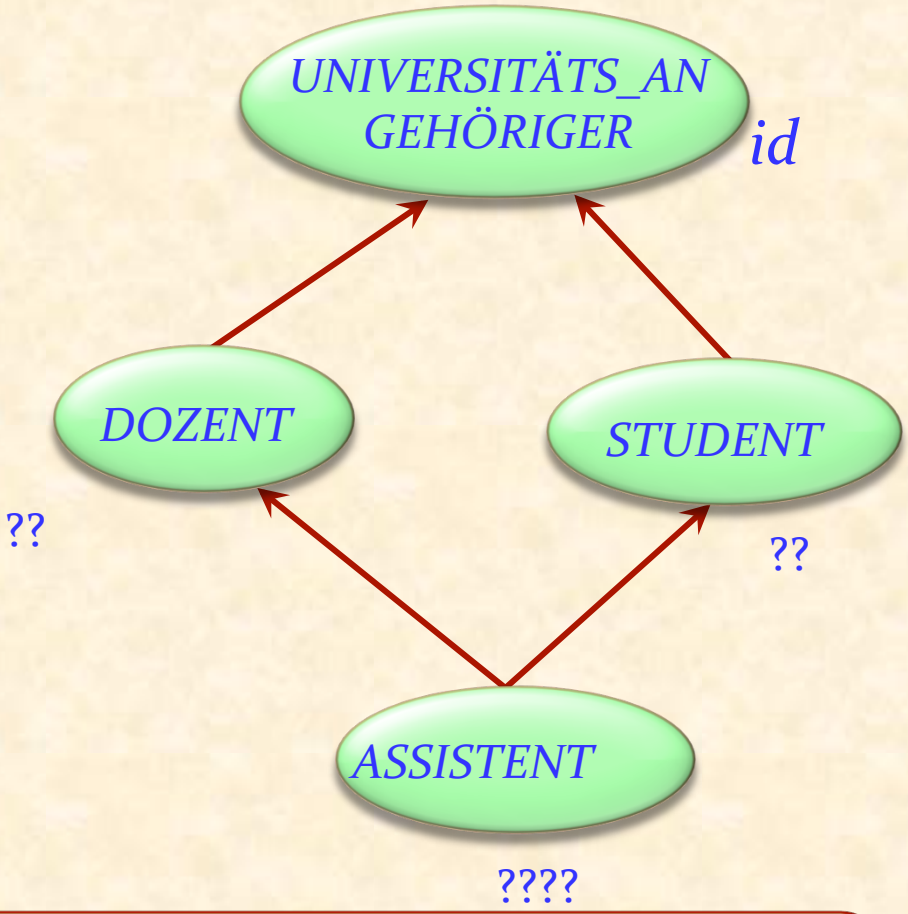


a1: A	*b1: B*	*c1: C*	*d1: D*
a1.g	*b1.f*	*c1.h*	*d1.f*

Ein Spezialfall der Mehrfachvererbung

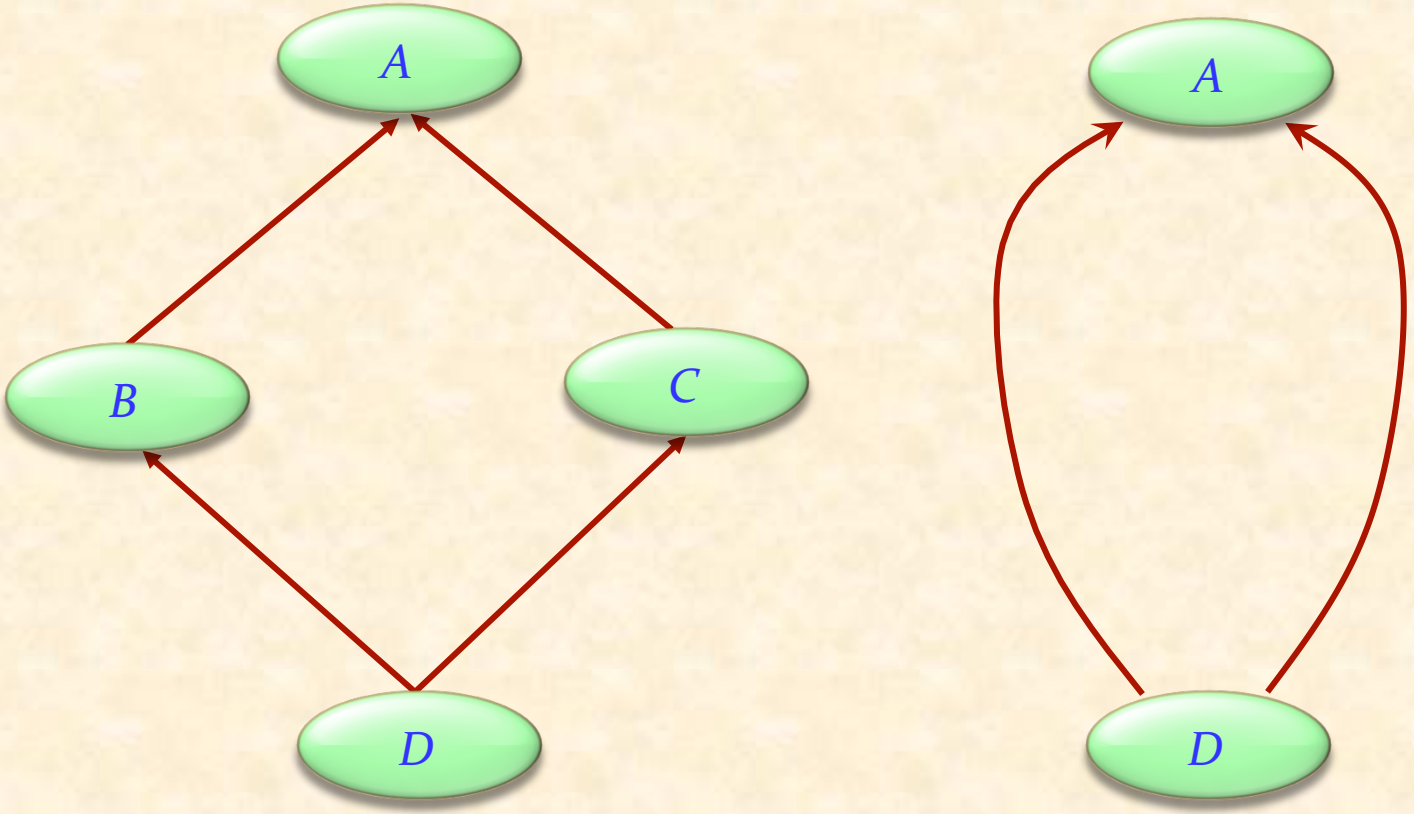
Mehrfachvererbung ermöglicht einer Klasse, zwei oder mehrere Vorfahren zu haben

Beispiel: *ASSISTENT* erbt von *DOZENT* und *STUDENT*



Dieses Beispiel bedingt wiederholte Vererbung: eine Klasse ist ein Nachkomme einer anderen Klasse in mehr als einer Art (durch mehr als einen Pfad)

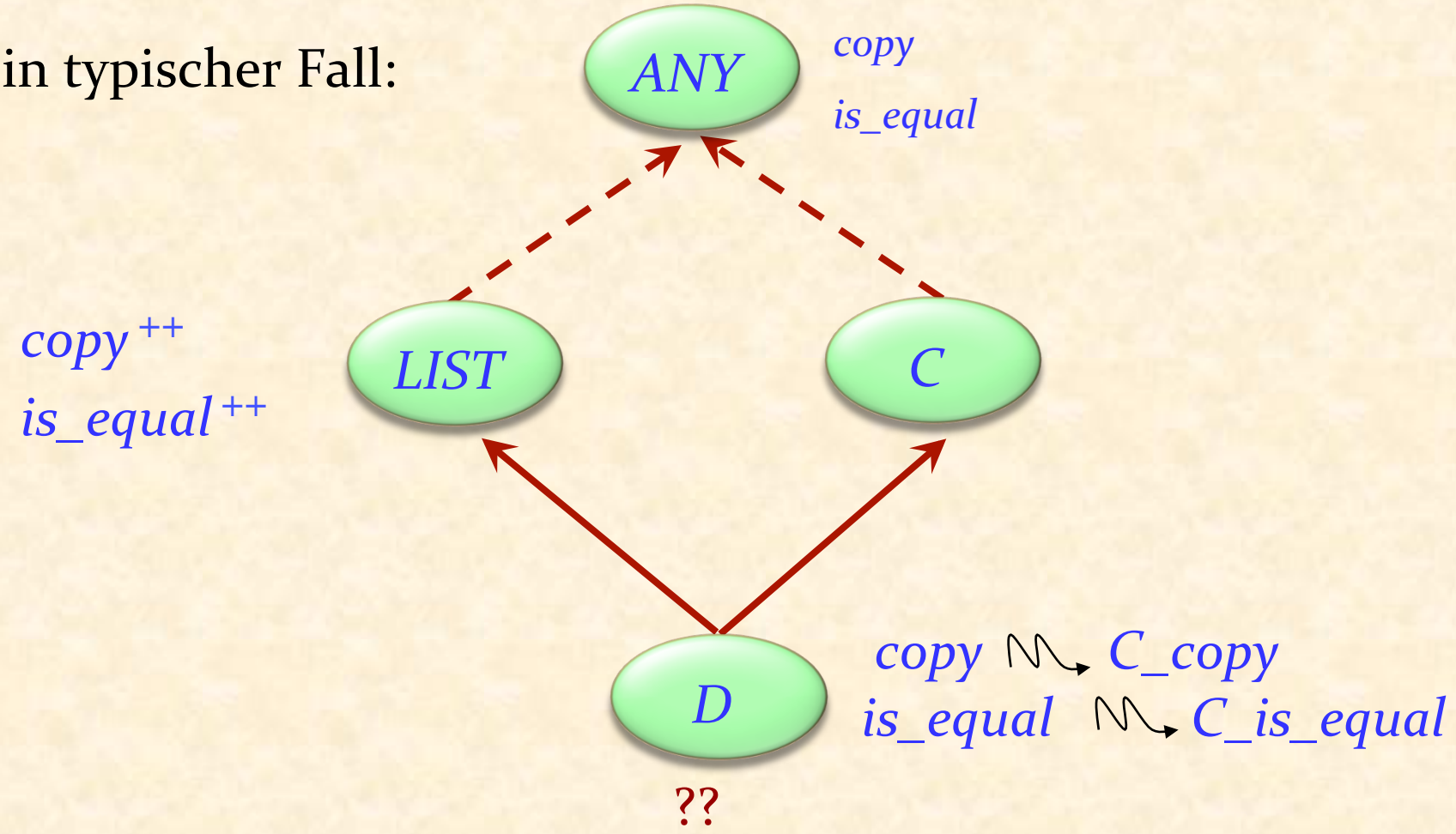
Indirekt und direkt wiederholte Vererbung



Auch als «Diamant des Todes» bekannt

Mehrfachvererbung ist auch wiederholte Vererbung

Ein typischer Fall:



Akzeptable Namenskonflikte

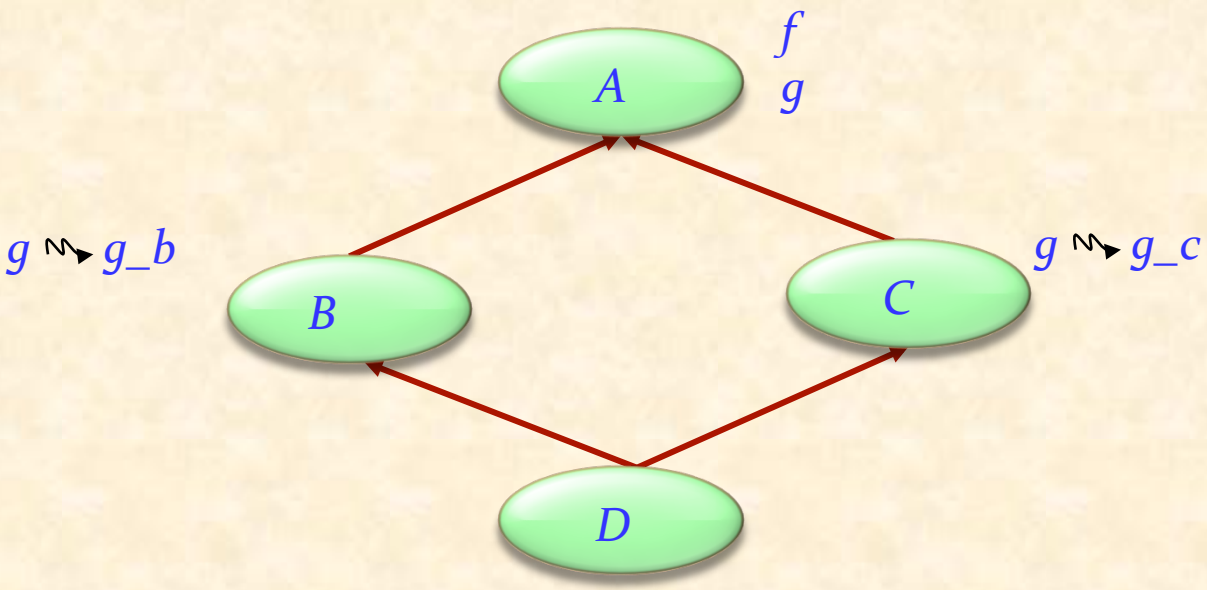
Wenn geerbte Features alle den gleichen Namen haben, besteht kein schädlicher Namenskonflikt, falls:

- 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

Teilen und Abgleichung



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:

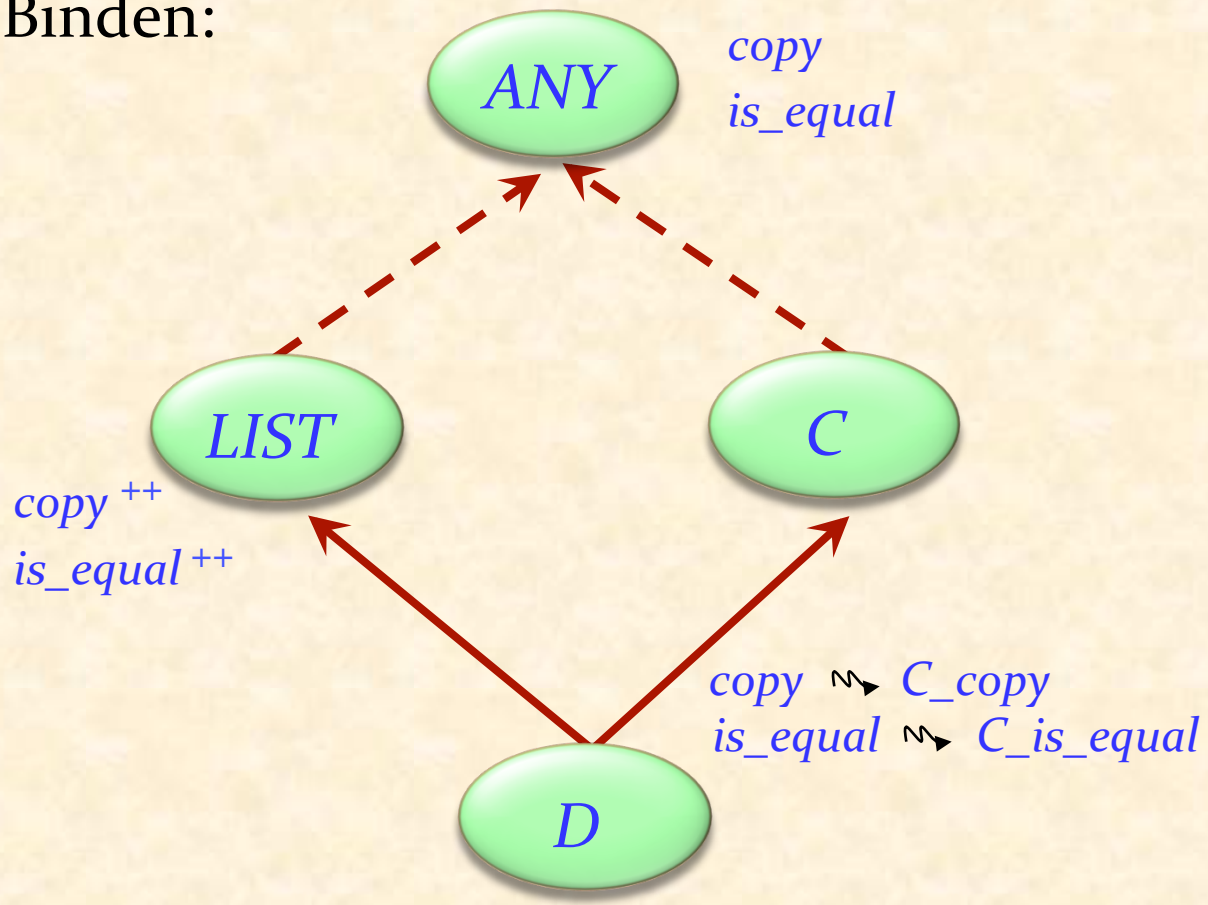
a1 : ANY

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
```

Was wir gesehen haben

Einige Spielchen, die man mit Vererbung spielen kann:

- Mehrfachvererbung
- Verschmelzen von Features
- Wiederholte Vererbung