Einführung in die Programmierung

Prof. Dr. Bertrand Meyer

Lektion 11:Einführung in die Konzepte der Vererbung und Generizität

Programm für heute (und nächstes Mal)

Zwei fundamentale Mechanismen für mehr Ausdruckskraft und Verlässlichkeit:

- Generizität (*genericity*)
- Vererbung (*inheritance*)

Mit den dazugehörigen (genauso wichtigen) Begriffen:

- Statische Typisierung (*static typing*)
- Polymorphie (*polymorphism*)
- Dynamisches Binden (*dynamic binding*)

Aus der zweiten Vorlesung

```
class
    PREVIEW
inherit
    ZURICH_OBJECTS
feature
    explore
            -- Die Stadt erkunden.
        do
            Central.highlight
            Polyterrasse.highlight
            Polybahn.add_transport
            Zurich_map.animate
        end
end
```

Den Begriff einer Klasse erweitern

Vererbung

Abstraktion

WAGEN_
MENGE

Generizität

Typ-Parametrisierung

Typ-Parametrisierung

STADT_
LISTE

WAGEN_
LISTE

PERSONEN_
LISTE

VERKETTETE_WAGEN_
LISTE

Spezialisierung

Vererbung und Generizität verbinden

Vererbung

*WAGEN_*
MENGE

*PERSONEN_*
MENGE

Generizität

STADT_
LISTE

*WAGEN_*
LISTE

PERSONEN_
LISTE

VERKETTETE_
STADT_LISTE

VERKETTETE_
*WAGEN_**LISTE***

Generizität

Uneingeschränkt:

LIST [G]

e.g. *LIST* [*INTEGER*], *LIST* [*PERSON*]

Eingeschränkt:

HASH_TABLE [*G* —> *HASHABLE*]

VECTOR [*G* —> *NUMERIC*]

Generizität: Typ-Sicherheit gewährleisten

Wie können wir konsistente „Container“-Datenstrukturen definieren, z.B. eine Liste von Konten oder eine Liste von Punkten?

Ohne Generizität vielleicht so:

c : STADT ; p : PERSON

staedte : LIST …

leute : LIST …

----

leute.extend (p)

staedte.extend (c)

c := staedte.last

c.stadt_operation

Aber: Was passiert bei einer falschen Zuweisung?

Mögliche Ansätze

1. Den Code duplizieren, von Hand oder mit Hilfe eines Makroprozessors

2. Bis zur Laufzeit warten; falls die Typen nicht passen, eine Laufzeitausnahme (Smalltalk) werfen

3. Konvertieren („cast") aller Werte zu einem universalen Typ, wie z.B. „Void-Zeiger" in C

4. Parametrisieren der Klasse, indem ein expliziter Name *G* für den Typ der Containerelemente angegeben wird. Dies ist der Ansatz in Eiffel. Auch die neusten Versionen von Java, .NET und andere Sprachen verwenden diesen Ansatz

Eine generische Klasse

Formaler generischer Parameter

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

Um die Klasse zu verwenden: Benutzen sie eine generische Ableitung (*generic derivation*), z.B.

Tatsächlicher generischer Parameter

staedte : *LIST* [*STADT*]

Gebrauch generischer Ableitungen

staedte : *LIST* [*STADT*]

leute : *LIST* [*PERSON*]

c : *STADT*

p : *PERSON*

...

staedte.extend (*c*)

leute.extend (*p*)

c := *staedte.last*

c.stadt_operation

STATISCHE TYPISIERUNG

Folgendes wird der Compiler zurückweisen:

- *leute.extend* (*c*)
- *staedte.extend* (*p*)

Statische Typisierung

Typsicherer Aufruf (*type-safe call*):

Während der Ausführung: ein Featureaufruf *x.f* , so dass das an *x* gebundene Objekt ein Feature hat, das *f* entspricht.

[Verallgemeinerung: mit Argumenten (z.B. *x.f* (*a*, *b*))]

Überprüfer für statische Typen (type checker):

Ein auf ein Programm anwendbares Werkzeug (z.B. ein Compiler) das — für alle Programme, die es akzeptiert — garantiert, dass jeder Aufruf in jeder Ausführung *typsicher* ist.

Statisch typisierte Sprache:

Eine Programmiersprache, für die es möglich ist, einen *Überprüfer für statische Typen* zu schreiben.

Der Gebrauch von Generizität

LIST [*STADT*]

LIST [*LIST* [*STADT*]]

…

Ein Typ ist nicht länger das Gleiche wie eine Klasse!

(Aber ein Typ **basiert** weiterhin auf einer Klasse)

Was ist ein Typ?

(Für Einfachheit nehmen wir an, dass jede Klasse entweder keinen oder genau einen generischen Parameter hat)

Ein Typ ist von einer der folgenden zwei Arten:

- *C*, wobei *C* der Name einer nicht-generischen Klasse ist
- *D* [*T*], wobei *D* der Name einer generischen Klasse und *T* ein Typ ist

Eine generische Klasse

Formaler generischer Parameter

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

Um die Klasse zu verwenden: Benutzen sie eine generische Ableitung, z.B.

Tatsächlicher generischer Parameter

```
staedte : LIST [STADT]
```

Aus der zweiten Vorlesung

```
class
    PREVIEW
inherit
    ZURICH_OBJECTS
feature
    explore
            -- Die Stadt erkunden.
        do
            Central.highlight
            Polyterrasse.highlight
            Polybahn.add_transport
            Zurich_map.animate
        end
end
```

Den Begriff einer Klasse erweitern

Vererbung

Abstraktion

*WAGEN_*
MENGE

Generizität

Typ-Parametrisierung

STADT_
LISTE

*WAGEN_*
LISTE

Typ-Parametrisierung

PERSONEN_
LISTE

VERKETTETE_*WAGEN_*
LISTE

Spezialisierung

Die duale Natur von Klassen

Eine Klasse ist ein Modul.

Eine Klasse ist ein Typ*.

> * Oder ein Typ-Muster (*template*) (siehe Generizität)

Eine Klasse als Modul:

- Gruppiert Menge von verwandten Diensten
- Erzwingt das Geheimnisprinzip (nicht jeder Dienst kann von ausserhalb genutzt werden)
- Hat Klienten (Module, die sie benutzen) und Versorger (Module, die sie benutzt)

Eine Klasse als Typ:

- Bezeichnet mögliche Laufzeitwerte (Objekte und Referenzen), die Instanzen des Typs
- Kann für Deklaration von Entitäten benutzt werden (die solche Werte repräsentieren)

Wie die beiden Ansichten zusammenpassen

Die Klasse, als *Modul* gesehen:

gruppiert eine Menge von Diensten (die Features der Klasse),

die genau den auf die Instanzen der Klasse (als *Typ* gesehen) anwendbaren Operationen entsprechen.

(Beispiel: die Klasse *PUBLIC_TRANSPORT*, Features *line*, *position*, *destination*, *speed*, *move*)

Grundlagen der Vererbung

Prinzip:

Beschreibung einer neuen Klasse als Erweiterung oder Spezialisierung einer existierenden Klasse.

(Oder mehreren, mit Hilfe von ***Mehrfachvererbung***)

Falls *B* von *A* erbt:

- Als Module: Alle Dienste von *A* sind in *B* verfügbar. (Möglicherweise mit unterschiedlicher Implementation)

- Als Typen: Immer, wenn eine Instanz von *A* erwartet wird, ist auch eine Instanz von *B* erlaubt. ("ist eine Art von"-Beziehung (is-a relationship))

Terminologie

Wenn *B* von *A* erbt (indem Klasse *B* Klasse *A* in ihrer **inherit**-Klausel auflistet):

- *B* ist ein **Erbe** von *A*
- *A* ist ein **Vorgänger** von *B*

Für eine Klasse *A*:

- Die **Nachkommen** von *A* sind *A* selbst und (rekursiv) die Nachkommen von *A*s Erben.
- **Echte Nachkommen** sind obige ohne *A*.

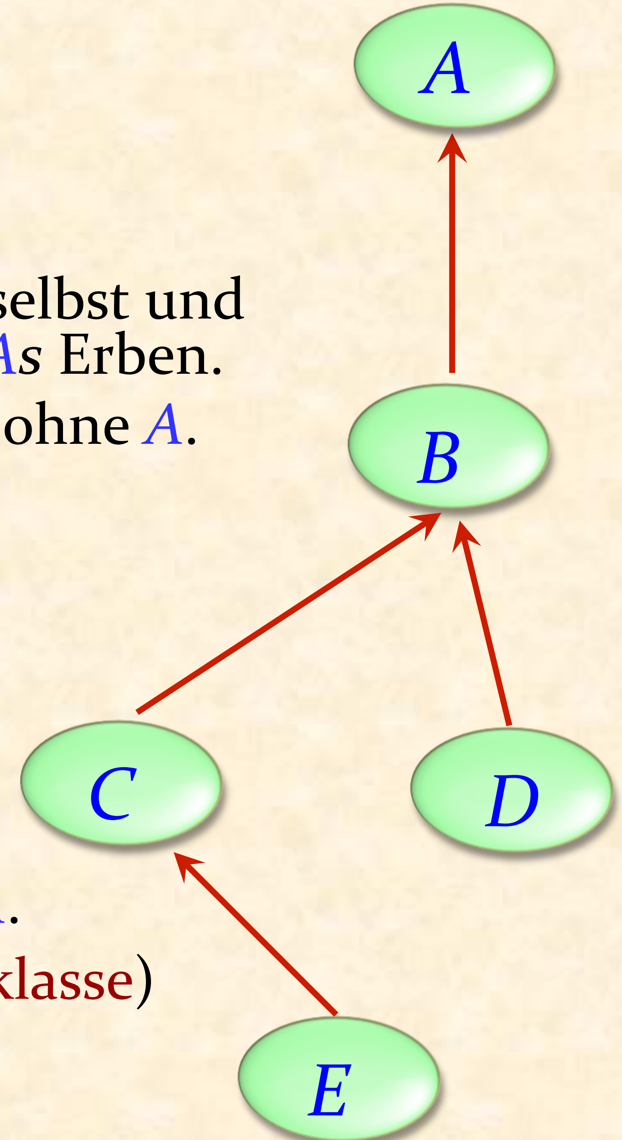
Umgekehrte Notation:

- **Vorfahre**
- **Echter Vorfahre**

Genauerer Begriff der Instanz:

- **Direkte Instanzen** von *A*
- **Instanzen** von *A* : Die direkten Instanzen von Nachkommen von *A*.

(Andere Terminologien: Unterklasse, Oberklasse)



Beispielhierarchie (in Traffic)

*	aufgeschoben (deferred)
+	wirksam (effective)
++	redefiniert (redefined)

*
MOBILE

*position**
*move**

*
TRANSPORT

position+
capacity
passenger_count
load

line
departed
destination
move+

PUBLIC_TRANSPORT

TAXI

take
move+

TOUR_TRAM

name
move++

Features im Beispiel

Feature

name: STRING

-- Name der Rundfahrt.

line: LINE

-- Linie, welcher der Wagen
-- folgt.

load (n: INTEGER)

-- Lade *n* Passagiere auf.

move (dt: INTEGER)

-- Position nach *dt* Milli-
-- sekunden aktualisieren.

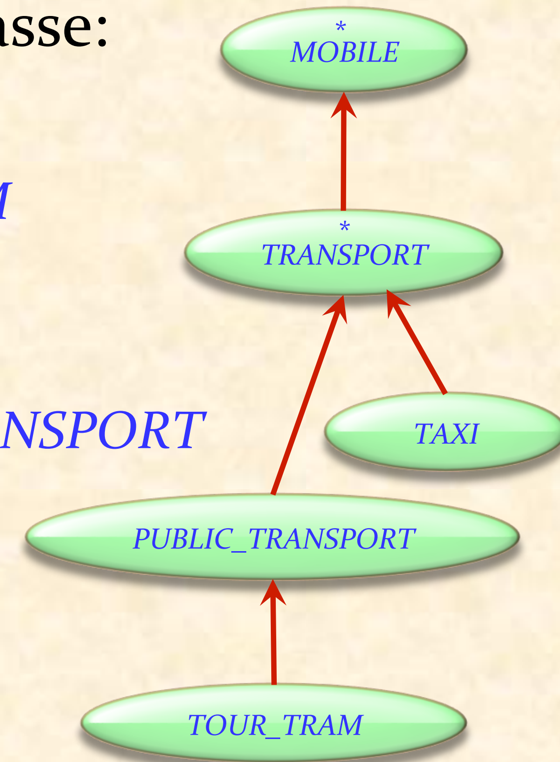
Aus der Klasse:

TOUR_TRAM

PUBLIC_TRANSPORT

TRANSPORT

MOBILE



Features vererben

```
deferred class
    TRANSPORT
inherit
    MOBILE
feature
    [... Rest der Klasse...]
end
```

Alle Features von *MOBILE* sind auch in *TRANSPORT* verfügbar

```
class
    PUBLIC_TRANSPORT
inherit
    TRANSPORT
feature
    [... Rest der Klasse ...]
end
```

Alle Features von *TRANSPORT* sind auch in *PUBLIC_TRANSPORT* verfügbar

```
class
    TOUR_TRAM
inherit
    PUBLIC_TRANSPORT
feature
    [... Rest der Klasse ...]
end
```

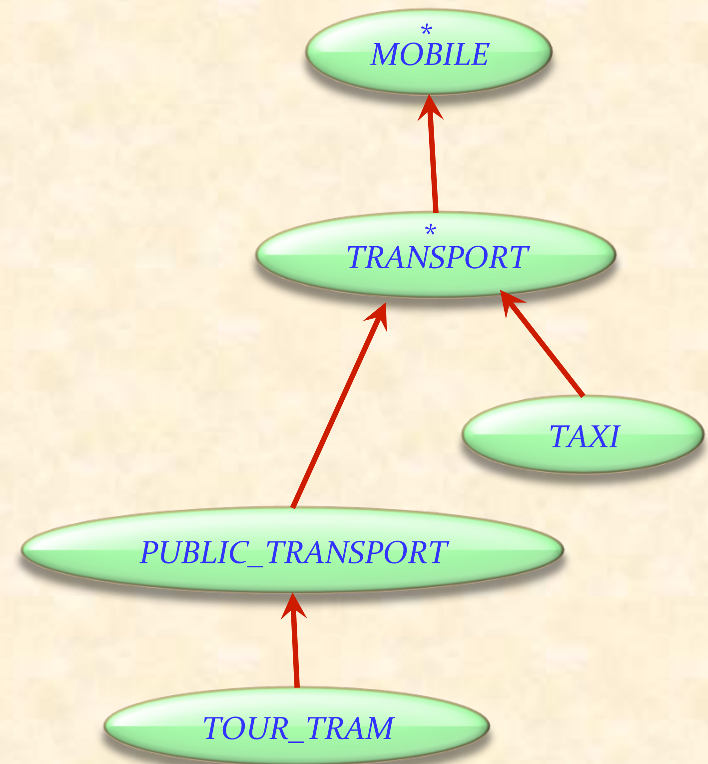
Alle Features von *PUBLIC_TRANSPORT* sind auch in *TOUR_TRAM* verfügbar

Vererbte Features

m : *MOBILE*; *t* : *TRANSPORT*; *p*: *PUBLIC_TRANSPORT*;
r : *TOUR_TRAM*

m•*move*(...)
t •*load* (...)
p •*line* -- Ein Ausdruck
r•*name* -- Ein Ausdruck

r •*move* (...)
r •*load* (...)
r •*line* -- Ein Ausdruck
r •*name* -- Ein Ausdruck



Definitionen: Arten von Features

Ein “**Feature einer Klasse**” ist:

- Ein **vererbtes** Feature, falls es ein Feature eines Vorfahrens ist, oder

- Ein **direktes** Feature, falls es in der Klasse selbst definiert und nicht vererbt ist. In diesem Fall sagt man, dass die Klasse das Feature **einführt**

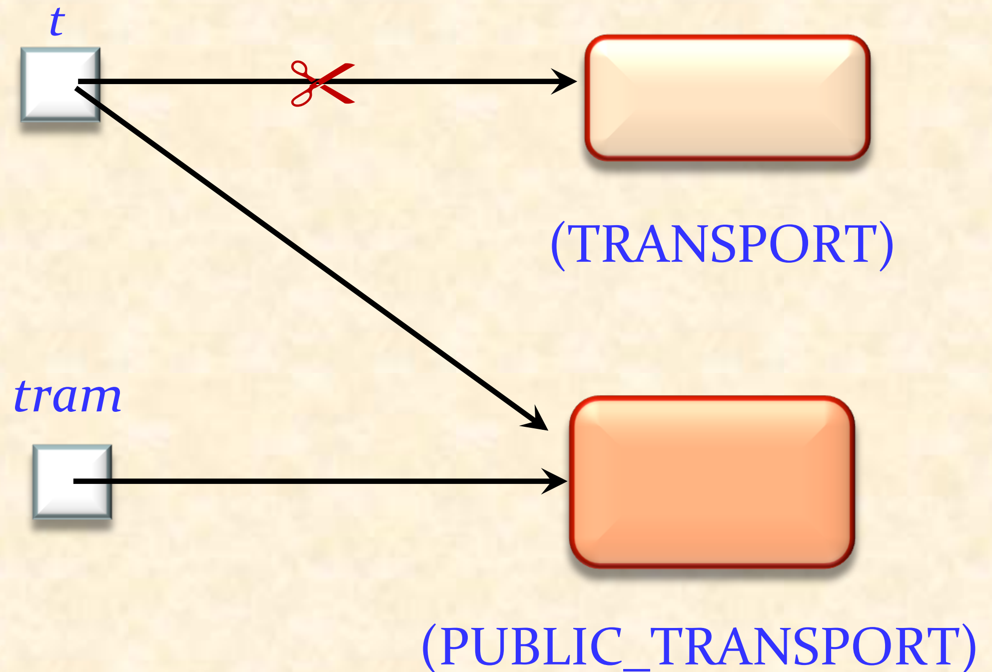
Polymorphe Zuweisung

t : *TRANSPORT*
tram : *PUBLIC_TRANSPORT*

Ein **echter Nachkomme** des ursprünglichen Typs

cab: TAXI

t := *tram*



Interessanter:

if *bedingung* **then**
t := *tram*
else
t := *cab*
...
end

Zuweisungen

Zuweisung:

ziel **:=** *ausdruck*

Bis jetzt (ohne Polymorphie):

ausdruck war immer vom **gleichen Typ** wie *ziel*

Mit Polymorphie:

Der Typ von *ausdruck* ist ein **Nachkomme** des Typs von *ziel*

Polymorphie gilt auch für Argumente

```
reise_zeit (t : TRANSPORT): REAL_64
        do ... end
```

Ein spezifischer Aufruf:

```
reise_zeit (tram)
```

Der Typ des eigentlichen Arguments ist ein **echter Nachkomme** des Typs des formalen Arguments

Definitionen: Polymorphie

engl. *Attachment*

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 zur Laufzeit — in Folge einer polymorphen Bindung — zu einem Objekt eines anderen Typs gebunden werden.

Polymorphie ist die Existenz dieser Möglichkeiten.

Definitionen: statischer und dynamischer Typ

Der **statische Typ** einer Entität ist der Typ ihrer Deklaration im zugehörigen Klassentext.

Falls der Wert einer Entität während einer Ausführung an ein Objekt gebunden ist, ist der Typ dieses Objekts der **dynamische Typ** der Entität zu dieser Zeit.

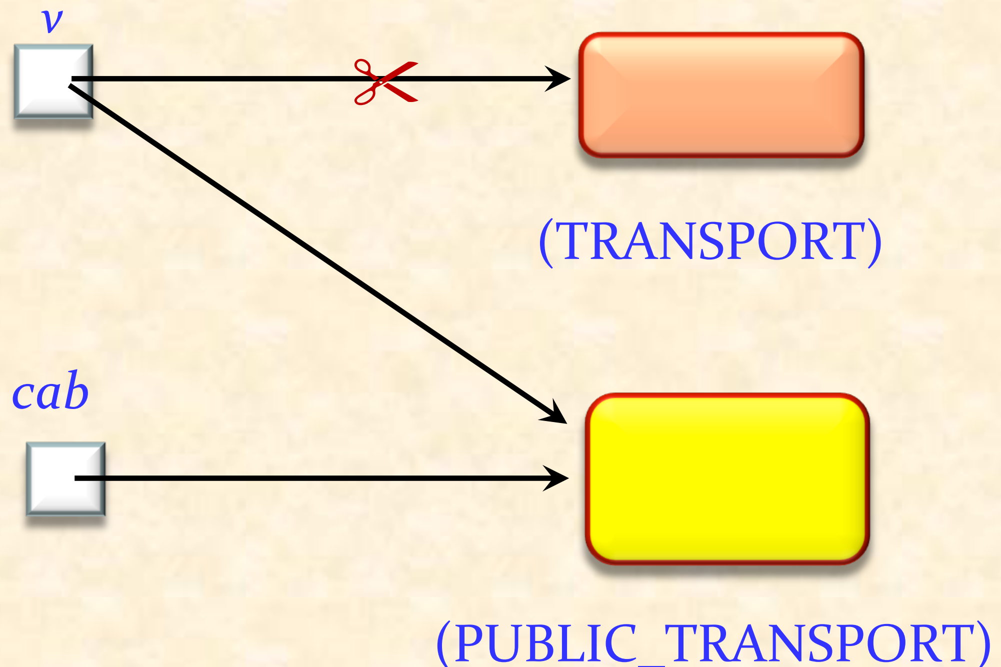
Statischer und dynamischer Typ

t : *TRANSPORT*

tram : *PUBLIC_TRANSPORT*

Statischer Typ von *t* :
TRANSPORT
Dynamischer Typ nach dieser Zuweisung:
PUBLIC_TRANSPORT

t := *tram*



Grundlegende Typ-Eigenschaft

Statischer und dynamischer Typ

Der dynamische Typ einer Entität ist immer konform zu ihrem statischem Typ

(vom Typ-System garantiert)

Statische Typisierung

Typsicherer Aufruf (während der Ausführung):

Ein Featureaufruf *x.f* , so dass das an *x* gebundene Objekt ein Feature hat, das *f* entspricht.

[Verallgemeinerung: mit Argumenten (z.B. *x.f (a, b)*)]

Überprüfer für statische Typen:

Ein auf ein Programm anwendbares Werkzeug (z.B. ein Compiler) das - für alle Programme, die es akzeptiert - garantiert, dass jeder Aufruf in jeder Ausführung *typsicher* ist.

Statisch typisierte Sprachen:

Eine Programmiersprache, für die es möglich ist, einen *Überprüfer für statische Typen* zu schreiben.

Vererbung und statische Typisierung

Elementare Typisierungsregel bei Vererbung

Eine polymorphe Bindung ist nur dann gültig, wenn der Typ der Quelle mit dem Typ des Ziels konform ist

konform: Grunddefinition

Referenztypen (nicht generisch): *U* ist **konform** zu *T* falls *U* ein Nachkomme von *T* ist

Ein *expandierter* Typ ist nur zu sich selbst konform

Konformität: präzisere Definition

Ein Referenztyp *U* ist zu einem Referenztyp *T* **konform**, falls:

- Sie den gleichen generischen Parameter haben und *U* ein Nachkomme von *T* ist, oder
- Sie beide generische Ableitungen mit der gleichen Anzahl tatsächlicher Parameter sind, der Vorfahre von *U* ein Nachkomme der Klasse *T* ist und jeder tatsächliche Parameter von *U* (rekursiv) zum jeweiligen tatsächlichen Parameter von *T* konform ist

Ein expandierter Typ ist nur zu sich selbst konform.

Statische Typisierung

Typsicherer Aufruf (während der Ausführung):

Ein Featureaufruf *x.f* , so dass das an *x* gebundene Objekt ein Feature hat, das *f* entspricht.

[Verallgemeinerung: mit Argumenten (z.B. *x.f (a, b)*)]

Überprüfer für statische Typen:

Ein auf ein Programm anwendbares Werkzeug (z.B. ein Compiler) das - für alle Programme, die es akzeptiert - garantiert, dass jeder Aufruf in jeder Ausführung *typsicher* ist.

Statisch typisierte Sprachen:

Eine Programmiersprache, für die es möglich ist, einen *Überprüfer für statische Typen* zu schreiben.

Das Vererbungs- und Konformitätssystem

Klassen (vom Progammierer definiert)

Noch eine Beispielhierarchie

center * FIGURE * display * rotate*

* OPEN_ FIGURE

perimeter *

* CLOSED_ FIGURE

SEGMENT

POLYLINE

perimeter +

+ POLYGON

perimeter +

+ ELLIPSE

...

...

perimeter ++

TRIANGLE

RECTANGLE

side1 side2 diagonal

perimeter ++

SQUARE

CIRCLE

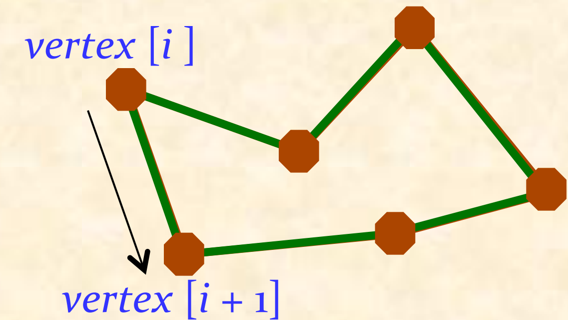
perimeter ++

*	aufgeschoben
+	wirksam
++	redefiniert

Redefinition 1: Polygone

```
class POLYGON inherit
        CLOSED_FIGURE
create
        make
feature
        vertex : ARRAY [POINT]
        vertex_count : INTEGER
        perimeter : REAL
                        -- Länge des Umfangs.
           do
              from ... until ... loop

                    Result := Result + vertex [i ] . distance (vertex [i + 1])
                     ...
                 end
           end
invariant
        vertex_count >= 3
        vertex_count = vertex.count
end
```



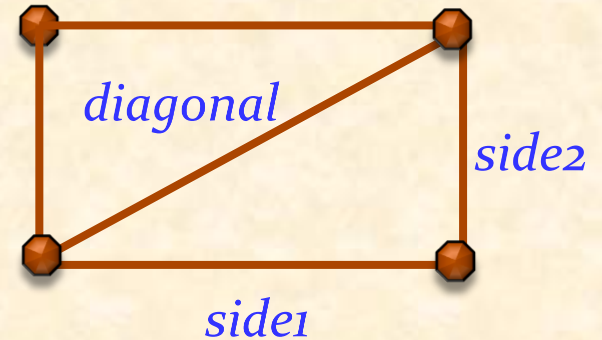
Redefinition 2: Rechtecke

```
class RECTANGLE inherit
        POLYGON
            redefine
                    perimeter
            end
create
        make

feature
        diagonal, side1, side2 : REAL

        perimeter : REAL
                    -- Länge des Umfangs.
            do Result := 2 * (side1 + side2) end
invariant

        vertex_count = 4
end
```



Vererbung, Typisierung und Polymorphie

Annahme:

p : *POLYGON* ; *r* : *RECTANGLE* ; *t* : *TRIANGLE*

x : *REAL*

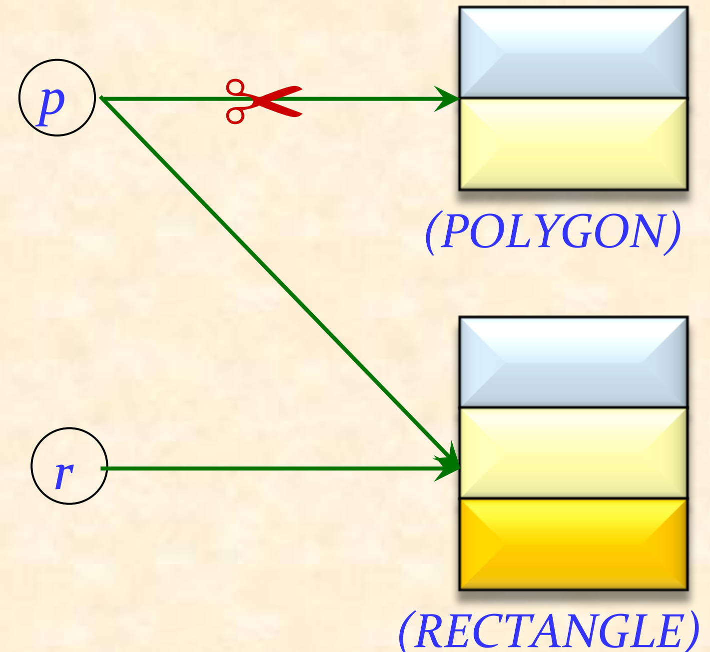
Erlaubt:

x := *p*.*perimeter*

x := *r*.*perimeter*

x := *r*.*diagonal*

p := *r*



NICHT erlaubt:

x := *p*.*diagonal* -- Auch direkt nach *p* := *r* !

r := *p*

Dynamisches Binden

Was ist hier das Resultat (falls *ein_test* wahr ist)?

```
if ein_test then
        p := r
else
        p := t
end

x :=  p.perimeter
```

Redefinition: Eine Klasse kann ein geerbtes Feature ändern.
Beispiel: *RECTANGLE* redefiniert *perimeter* .

Polymorphie: *p* kann zur Laufzeit mehrere Formen haben.

Dynamisches Binden: Das Resultat von *p.perimeter* hängt vom der Laufzeitform von *p* ab.

Definition: Dynamisches Binden (Dynamic binding)

Dynamisches Binden (eine semantische Regel):

- Jede Ausführung eines Featureaufrufs ruft das am besten zum Typ des Zielobjekts adaptierte Feature auf.

Binden und Typisierung

(Für einen Aufruf *x*•*f*)

Statische Typisierung: Die Garantie, dass es mindestens eine Version von *f* gibt.

Dynamisches Binden: Die Garantie, dass jeder Aufruf die geeignetste Version von f aufruft.

Ohne dynamisches Binden?

```
display (f: FIGURE)
    do
        if "f ist ein CIRCLE" then
            ...
        elseif "f ist ein POLYGON" then
            ...
        end
    end
```

Und ähnlich für alle Routinen!

Lästig; muss immer wieder geändert werden, wenn es einen neuen Figurentyp gibt.

Mit Vererbung und zugehörigen Techniken

Mit:

```
f : FIGURE
c : CIRCLE
p : POLYGON
```

und:

```
create c.make (...)
create p.make (...)
```

Initialisieren:

```
if ... then
    f := c
else
    f := p
end
```

Danach einfach:

```
f.move (...)
f.rotate (...)
f.display (...)
    -- und so weiter für
    -- jede Operation von f !
```

Vererbung: Zusammenfassung 1

Typenmechanismus: erlaubt es, Datenabstraktionen zu klassifizieren.

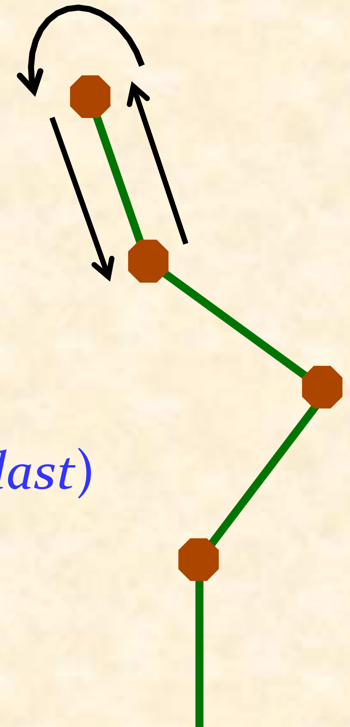
Modul-Mechanismus: erlaubt es, neue Klassen als Erweiterung von existierenden Klassen zu erstellen.

Polymorphie: Flexibilität *mit* Typ-Sicherheit.

Dynamisches Binden: automatische Adaption der Operation auf das Ziel für modularere Softwarearchitekturen.

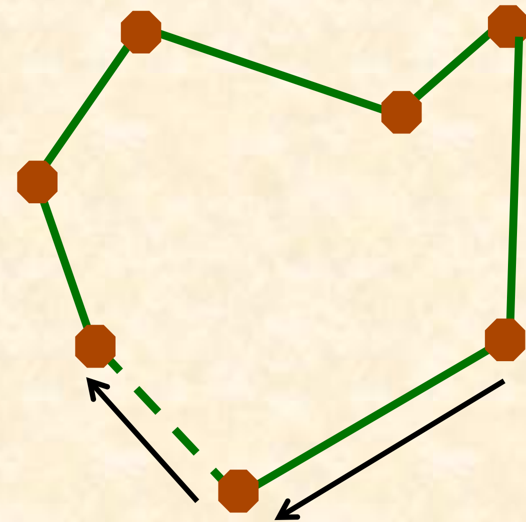
Redefinition

```
class PUBLIC_TRANSPORT feature
	departed : STATION                      -- Abgangsstation.
	arriving : STATION                      -- Ankunftsstation.
	towards_last : BOOLEAN                  -- Fahrtrichtung.
	destination : STATION              -- Zielstation.
	move (dt : INTEGER)
			-- Position nach dt Millisekunden aktualisieren.
		do
			[...]
			departed := arriving
			if arriving = destination then
				-- Richtung wechseln.
				towards_last := not towards_last
			end
			arriving := line.next_station (departed, towards_last)
			[...]
		end
	[...]
end
```



Redefinition 2: Tram-Rundfahrt

```
class TOUR_TRAM inherit
      PUBLIC_TRANSPORT
            redefine move end
feature
      move (dt : INTEGER)
                  -- Position nach dt Millisekunden aktualisieren.
            do
                  [...]
                  departed := arriving
                  if arriving = line.last then
                        -- Richtung nicht wechseln.
                        arriving := line.first
                  else
                        arriving := line.next_station
                              (departed, towards_last)
                  end
                  [...]
            end
      [...]
end
```



Dynamisches Binden

Was ist hier das Resultat (falls *i_feel_like_fondue* wahr ist)?

m: MOBILE, tram9 : PUBLIC_TRANSPORT, fondue_tram : TOUR_TRAM

```
if i_feel_like_fondue then
        m := fondue_tram
else
        m := tram9
end
m.move (5)
```

Redefinition: Eine Klasse kann ein geerbtes Feature ändern. Beispiel: *TOUR_TRAM* redefiniert *move*.

Polymorphie: *m* kann zur Laufzeit mehrere Formen haben.

Dynamisches Binden: Das Resultat von *m.move* hängt von der Laufzeitform von m ab.

Dynamisches Binden

Es gibt mehrere Versionen von *move*.

MOBILE * — *move**

TRANSPORT * — erbt von *MOBILE*

PUBLIC_TRANSPORT — *move+* — erbt von *TRANSPORT*

TAXI — *move+* — erbt von *TRANSPORT*

TOUR_TRAM — *move++* — erbt von *PUBLIC_TRANSPORT*

↑ erbt von

* aufgeschoben

Den Begriff einer Klasse erweitern

Vererbung

Abstraktion

*WAGEN_*
MENGE

Generizität

Typ-Parametrisierung

Typ-Parametrisierung

STADT_
LISTE

*WAGEN_*
LISTE

PERSONEN_
LISTE

VERKETTETE_*WAGEN_*
LISTE

Spezialisierung

Den Begriff einer Klasse erweitern

Vererbung

WAGEN_ MENGE

PERSONEN_ MENGE

Generizität

STADT_ LISTE

WAGEN_ LISTE

PERSONEN_ LISTE

VERKETTETE_ STADT_LISTE

VERKETTETE_ WAGEN_LISTE

Konformität

Vorher definiert für nicht-generisch abgeleitete Typen:

Was wir gesehen haben

Die fundamentalen O-O Mechanismen:

- Vererbung
- Polymorphie
- Dynamisches Binden
- Statische Typisierung
- Generizität

Unser Programm für den zweiten Teil

Repetition der (eingeschränkter) Generizität

Vererbung: aufgeschobene Klassen

Vererbung: Was passiert mit den Verträgen?

Vererbung: Wie können wir den tatsächlichen Typ eines Objektes bestimmen?

Nicht in dieser Stunde, aber später: Mehrfachvererbung, Umbenennungen etc.

Generizität (Repetition)

Ohne Einschränkung

LIST [G]

e.g. *LIST* [*INTEGER*], *LIST* [*PERSON*]

Eingeschränkt

HASH_TABLE [*G* —> *HASHABLE*]

VECTOR [*G* —> *NUMERIC*]

Eine generische Klasse (Repetition)

Formaler generischer Parameter

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

Um die Klasse zu verwenden: Wird eine generische Ableitung benutzt, z.B.

Tatsächlicher generischer Parameter

staedte : *LIST* [*STADT*]

Gebrauch generischer Ableitungen (Repetition)

staedte : *LIST* [*STADT*]
leute : *LIST* [*PERSON*]
c : *STADT*
p : *PERSON*
...

staedte.extend (*c*)
leute.extend (*p*)

c := *staedte.last*

c.stadt_operation

STATISCHE TYPISIERUNG

Folgendes wird der Compiler zurückweisen:

- *leute.extend* (*c*)
- *staedte.extend* (*p*)

Generizität: Zusammenfassung

- Mechanismus zur Typerweiterung
- Vereint Typ-Sicherheit und Flexibilität
- Ermöglicht parametrisierte Klassen
- Besonders nützlich für Container-Datenstrukturen, wie z.B. Listen, Arrays, Bäume, …
- “Typ” ist nun ein wenig allgemeiner als „Klasse“

Definition: Typ

Wir benutzen Typen, um Entitäten zu deklarieren:

x : SOME_TYPE

Mit dem bisherigen Mechanismus ist ein **Typ**:

- Entweder eine nicht-generische Klasse, z.B.
  STATION

- Oder eine generische Ableitung, z.B. der Name einer Klasse, gefolgt von einer Liste von **Typen**, die tatsächlichen generischen Parameter, in Klammern, z.B.
  LIST [STATION]
  LIST [ARRAY [STATION]]

Vererbung und Generizität verbinden

Vererbung

WAGEN_ MENGE

PERSONEN_ MENGE

Generizität

STADT_ LISTE

WAGEN_ LISTE

PERSONEN_ LISTE

VERKETTETE_ STADT_LISTE

VERKETTETE_ WAGEN_LISTE

Generizität + Vererbung: eingeschränkte Generizität

```
class VECTOR [G          ] feature
        plus alias "+" (other : VECTOR [G]): VECTOR [G]
                        -- Summe des aktuellen Vektors und other.
                require
                        lower = other.lower
                        upper = other.upper
                local
                        a, b, c : G
                do
                        ... Siehe nachher...
                end
        ... andere Features ...
end
```

Addieren zweier Vektoren

$u \quad + \quad v \quad = \quad w$

2

1 $\quad a \quad + \quad b \quad = \quad c$

Eingeschränkte Generizität

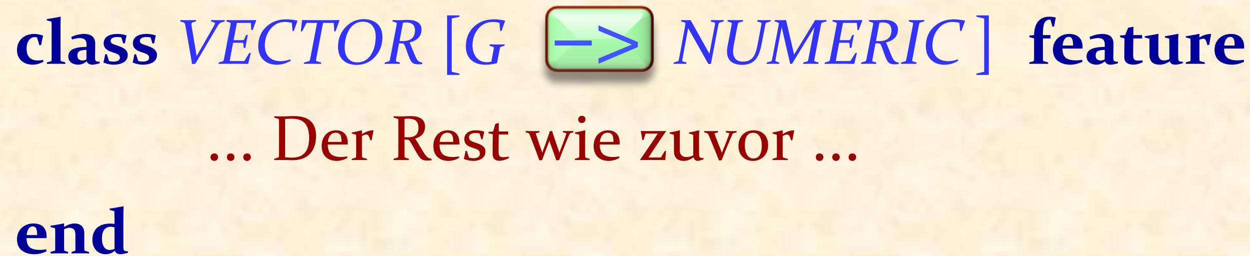
Rumpf von *plus* **alias** "+":

```
create Result.make (1, count)

from
        i := 1
until
        i > count
loop
        a := Current [i]
        b := other [i ]
        c := a + b          -- Benötigt “+” Operation auf G!
        Result [i ] := c
        i := i + 1
end
```

Die Lösung

Die Klasse *VECTOR* deklarieren als:



Die Klasse *NUMERIC* (von der Kernel-Bibliothek) enthält die Features *plus* **alias** "+", *minus* **alias** "-" etc.

Die Lösung verbessern

Machen sie aus *VECTOR* selbst ein Nachkomme von *NUMERIC* :

class *VECTOR* [*G* –> *NUMERIC*] **inherit**

NUMERIC

feature

... Rest wie vorher, einschliesslich **infix** "+"...

end

Dies ermöglicht die folgenden Definitionen:

v : *VECTOR* [*INTEGER*]

vv : *VECTOR* [*VECTOR* [*INTEGER*]]

vvv : *VECTOR* [*VECTOR* [*VECTOR* [*INTEGER*]]]

Vererbung und Generizität verbinden

Vererbung

Generizität

WAGEN_
MENGE

PERSONEN_
MENGE

STADT_
LISTE

WAGEN_
LISTE

PERSONEN_
LISTE

VERKETTETE_
STADT_LISTE

VERKETTETE_
WAGEN_LISTE

fleet: LIST [TRANSPORT]

t: TRANSPORT

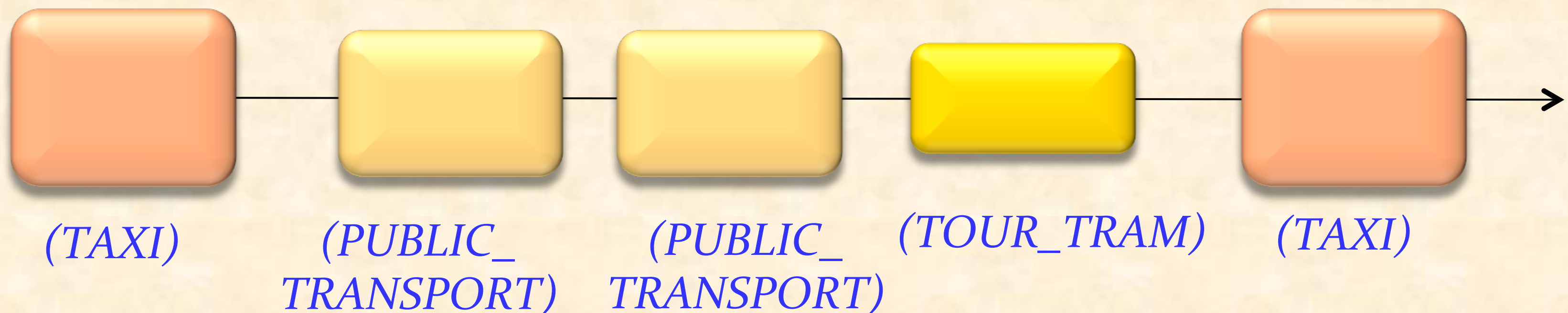
extend (v : G)

-- Ein neues Vorkommen von *v* hinzufügen.

...

fleet.extend (t)

fleet.extend (tram)

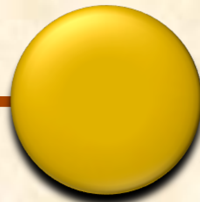
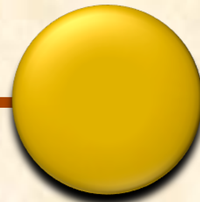
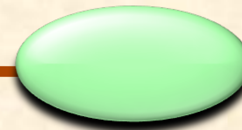


Generizität + Vererbung 2: Polymorphe Datenstrukturen

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

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

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

(POLYGON) (CIRCLE) (CIRCLE) (ELLIPSE) (POLYGON)

Beispielhierarchie

center *
* FIGURE
display *
rotate*

* OPEN_ FIGURE

perimeter *

* CLOSED_ FIGURE

SEGMENT

POLYLINE

perimeter +

\+ POLYGON

perimeter +

\+ ELLIPSE

...

...

TRIANGLE

RECTANGLE

side1
side2
diagonal

perimeter ++

perimeter ++

SQUARE

CIRCLE

perimeter ++

* aufgeschoben
\+ wirksam
++ redefiniert

Mit polymorphen Datenstrukturen arbeiten

bilder: LIST [FIGURE]

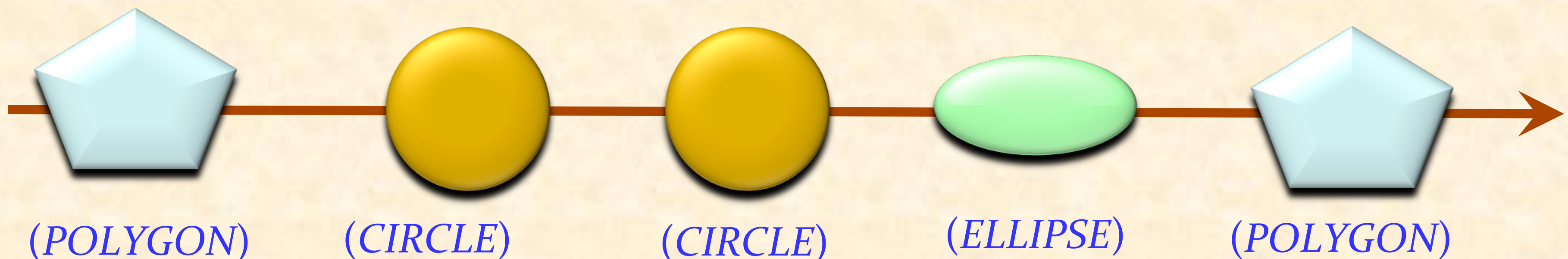
...

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

c • *item* • *display*

end

Dynamisches Binden



Definition (Polymorphie, angepasst)

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.

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

center *

* FIGURE

display *
rotate*

* OPEN_ FIGURE

perimeter *

* CLOSED_ FIGURE

SEGMENT

POLYLINE

perimeter +

\+ POLYGON

perimeter +

\+ ELLIPSE

...

...

TRIANGLE

RECTANGLE

side1
side2
diagonal

perimeter ++

perimeter ++

SQUARE

CIRCLE

perimeter ++

* aufgeschoben
\+ wirksam
++ redefiniert

Aufgeschobene Klassen in EiffelBase

*CONTAINER
*BOX
*COLLECTION
*TRAVERSABLE
*FINITE
*INFINITE
*BAG
*SET
*HIERARCHICAL
*LINEAR
*BOUNDED
*UNBOUNDED
*COUNTABLE
*TABLE
*ACTIVE
*INTEGER_ INTERVAL
*BILINEAR
*RESIZABLE
*INDEXABLE
*CURSOR_ STRUCTURE
*DISPENSER
*SEQUENCE
...
...
ARRAY
STRING
HASH_TABLE
*STACK
*QUEUE

* aufgeschoben

Ein aufgeschobenes Feature

In *ITERATION_CURSOR*:

```
forth
    require
        not after
    deferred
    ensure
        index = old index + 1
    end
```

Aufgeschobene und wirksame Features mischen

In der gleichen Klasse

wirksam!

```
search (x : G)
        -- Gehe zur ersten Position nach der
        -- aktuellen, wo x auftritt, oder after
        -- falls es nicht auftritt.
    do
        from until after or else item = x loop
            forth
        end
    end
```

aufgeschoben

“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

Anwendung von aufgeschobenen Klassen

Analyse und Entwurf, von oben nach unten (top-down)

Systematik

Gemeinsames Verhalten zusammenfassen

Aufgeschobene Klassen in EiffelBase

CONTAINER *
BOX *
COLLECTION *
TRAVERSABLE *
FINITE *
INFINITE *
BAG *
SET *
HIERARCHICAL *
LINEAR *
BOUNDED *
UNBOUNDED *
COUNTABLE *
TABLE *
ACTIVE *
INTEGER_INTERVAL *
BILINEAR *
RESIZABLE *
INDEXABLE *
CURSOR_STRUCTURE *
DISPENSER *
SEQUENCE *
...
...
ARRAY
STRING
HASH_TABLE
STACK *
QUEUE *

* aufgeschoben

Eine Anwendung: Undo/Redo

Dieses Beispiel nutzt wiederum mächtige polymorphe Datenstrukturen

Wir werden nun nur eine Skizze sehen, die Details werden in der Lektion über Agenten besprochen.

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.

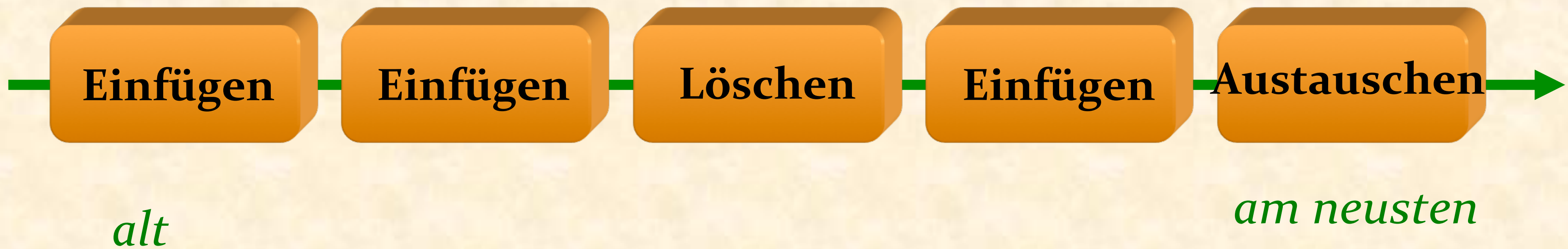
Wenn der Benutzer ein „Undo“ verlangt, stelle den zuletzt gesicherten Zustand wieder her.

Aber: Verschwendung von Ressource, insbesondere Speicherplatz.

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

Die „Geschichte“ einer Sitzung speichern

Die Geschichte-Liste:



geschichte : *LIST* [*BEFEHL*]

Was ist ein “Befehl” (*Command*) -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 “**Löschen**”-Objekt brauchen wir:

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

Allgemeiner Begriff eines Befehls

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

Die Befehl-Klassenhierarchie

*	aufgeschoben
+	wirksam

*execute**

*undo**

*
BEFEHL

\+
LÖSCHEN

\+
EINFÜGEN

…

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

…

execute⁺
undo⁺
index

…

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)

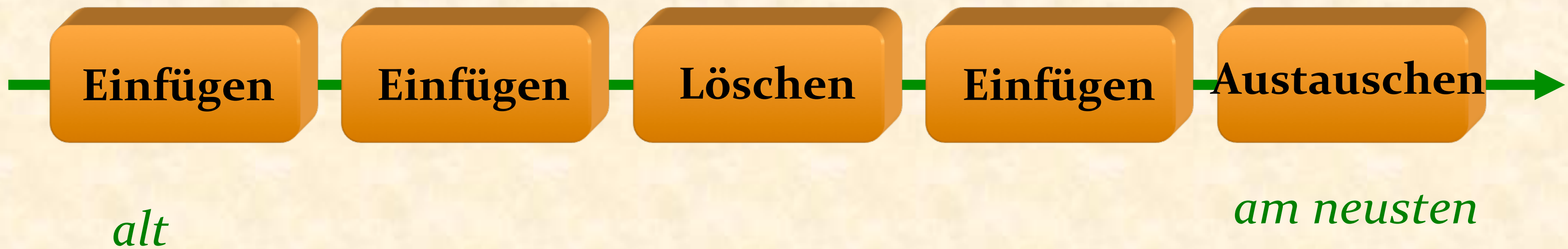
```
class LÖSCHEN inherit BEFEHL feature
        controller : EDIT_CONTROLLER
                        -- Zugriff auf das Geschäftsmodell.
        line : STRING
                        -- Zu löschende Zeile.
        index : INTEGER
                        -- Position der zu löschenden Zeile.

        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
```

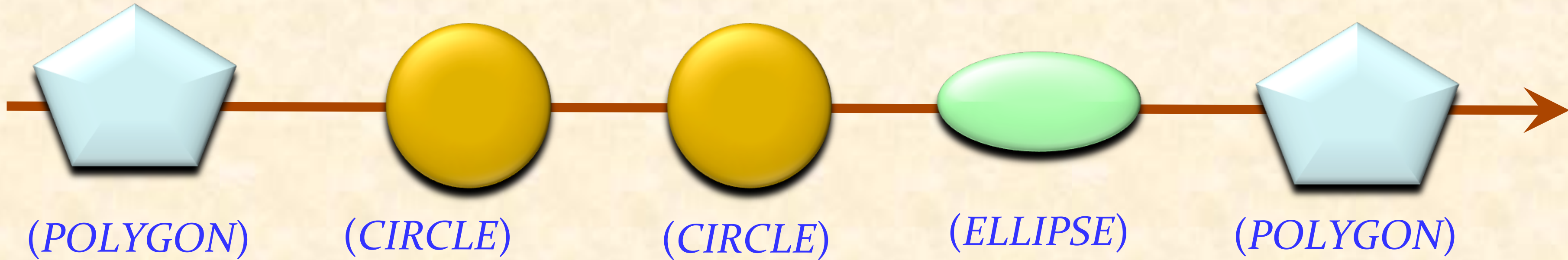
Die Geschichte-Liste

Eine polymorphe Datenstruktur



geschichte : LIST [BEFEHL]

Erinnerung: Liste von Figuren



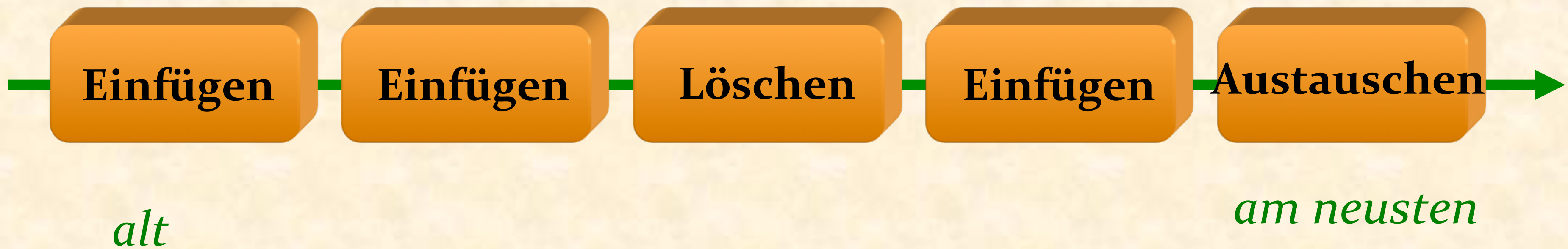
bilder.*extend* (*p1*) ; *bilder*.*extend* (*c1*) ; *bilder*.*extend* (*c2*)
bilder.*extend* (*e*) ; *bilder*.*extend* (*p2*)

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



geschichte : *LIST* [*BEFEHL*]

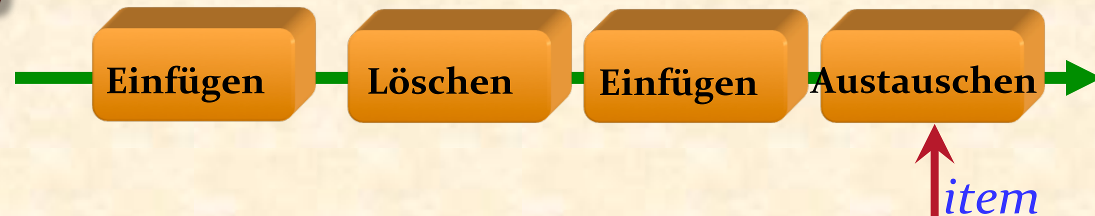
cursor: *ITERATION_CURSOR* [*BEFEHL*]

Einen Benutzerbefehl ausführen

```
decode_user_request

if "Anfrage ist normaler Befehl" then
        "Erzeuge ein Befehlsobjekt c , der Anforderung entsprechend"
        geschichte.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

*	aufgeschoben
+	wirksam

*execute**

*undo**

*
BEFEHL

+
LÖSCHEN

+
EINFÜGEN

...

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

...

execute+
undo+
index

...

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

“ Object-Test Local”

Zu prüfender Ausdruck

```
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.

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

<, <=,
>, >=,
…
(Totale Ordnungs-relation)

COMPARABLE

NUMERIC

+, –,
*, /
…
(kommutativer Ring)

INTEGER

REAL

STRING

COMPLEX

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
```

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,
    omitted …

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
```

Verträge und Vererbung

Problem: Was passiert bei Vererbung mit

- Klasseninvarianten?
- Vor- und Nachbedingungen von Routinen?

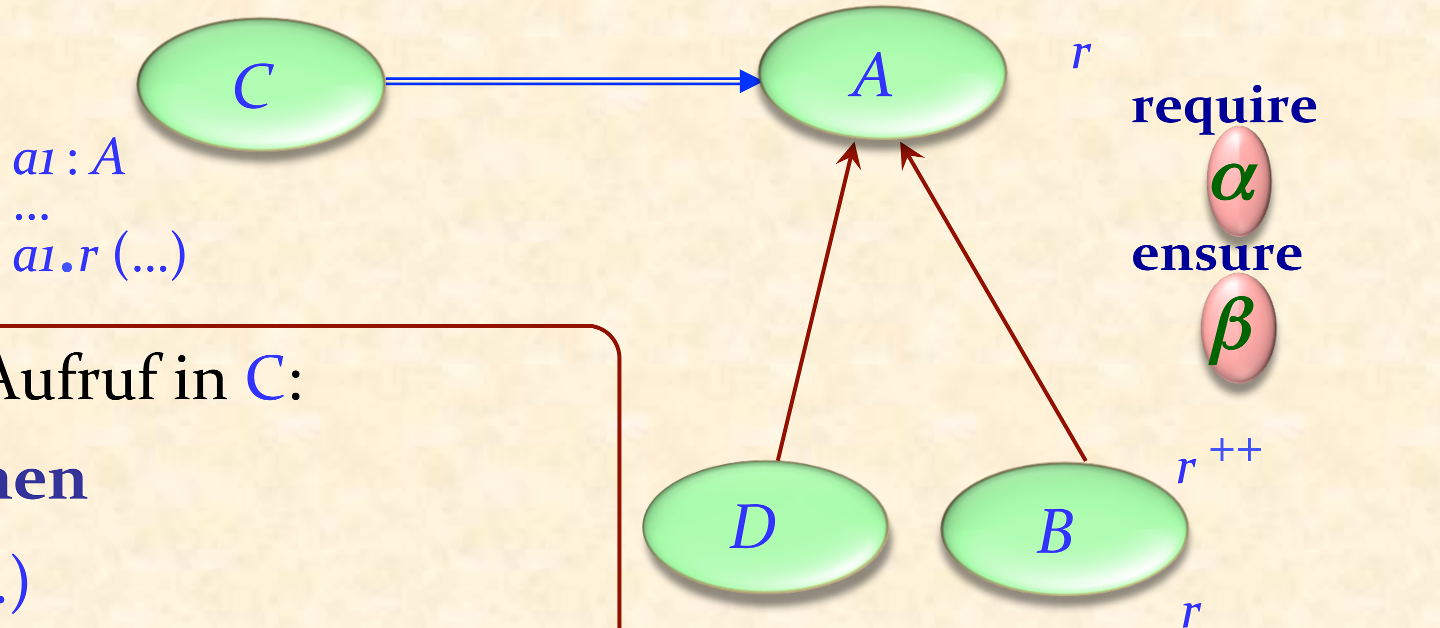
Invarianten

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

Klient von

erbt von

++ Redefinition

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*

Was wir gesehen haben

Aufgeschobene Klassen und ihre Rolle in Softwareanalyse und –entwurf.

Verträge und Vererbung

Den „tatsächlichen“ Typen eines Objektes herausfinden