



# Einführung in die Programmierung

Prof. Dr. Bertrand Meyer

Lektion 9: Abstraktion



Abstraktion, vor allem funktionale Abstraktion

Der Begriff der Routine

Das letzte Wort zu Features: alle Featurekategorien

Das Prinzip des einheitlichen Zugriffs (Uniform Access Principle)

Abstraktionen und Kundenprivilegien

Das Geheimnisprinzip

# Routine: eine Abstraktion eines Algorithmus



Abstrahieren heisst, die *Essenz* eines Konzeptes zu erfassen und Details und Angaben zu ignorieren

Will heissen:

- Einige Informationen *weglassen*
- Dem Ergebnis der Abstraktion einen *Namen* geben

In der Programmierung:

- Datenabstraktion: **Klasse**
- Abstraktion eines (operativen) Algorithmus: **Routine**

Eine Routine wird auch **Methode** genannt

Oder **Subprogramm** oder **Subroutine**

# Eine Routine ist eine der zwei Featurekategorien...

---



... die andere Kategorie sind die *Attribute*.

Wir sind schon zahlreichen Routinen (als Features) begegnet, allerdings ohne den Namen zu kennen

# Eine Routine

---



*r*(*arg*: *TYPE*; ...)

-- Kopfkomentar.

**require**

*Vorbedingung* (Boole'scher Ausdruck)

**do**

*Rumpf* (Instruktionen)

**ensure**

*Nachbedingung* (Boole'scher Ausdruck)

**end**

# Gebrauch von Routinen



Von unten nach oben (*bottom-up*): Erfasse den existierenden Algorithmus, wenn möglich wiederverwendbar

Von oben nach unten (*top-down*): Platzhalter-Routinen – Eine attraktive Alternative zu Pseudocode

```
build_route  
  -- Eine Route bauen und  
  -- damit arbeiten.  
do  
  create_opera_route  
  Zurich.add_route  
    (Opera_route)  
  Opera_route.reverse  
end
```

```
create_opera_route  
  -- Route erzeugen und  
  -- Teilstrecken hinzufügen.  
do  
  -- TODO  
  -- BM, 26 Oct 2011  
end
```

Methodologie: "TODO"-Einträge sollten informativ sein

# Zwei Arten von Routinen

---



**Prozedur:** gibt kein Resultat zurück

- Ergibt einen **Befehl**
- Aufrufe sind **Instruktionen**

**Funktion:** gibt ein Resultat zurück

*f(arg: TYPE, ...):* **RESULT\_TYPE**

... (Der Rest wie zuvor) ...

- Ergibt eine **Abfrage**
- Aufrufe sind **Ausdrücke**

# Features: Die ganze Wahrheit

---



Eine Klasse wird durch ihre Features charakterisiert.

Jedes Feature ist eine Operation auf den korrespondierenden Elementen: Abfrage oder Befehl.

Features sind der Leserlichkeit halber in verschiedene Kategorien eingeteilt.

Klassenklauseln:

- Noten (Indexierung)
- Vererbung
- Erzeugung
- Feature (mehrere)
- Invariante

Anatomie einer Klasse:

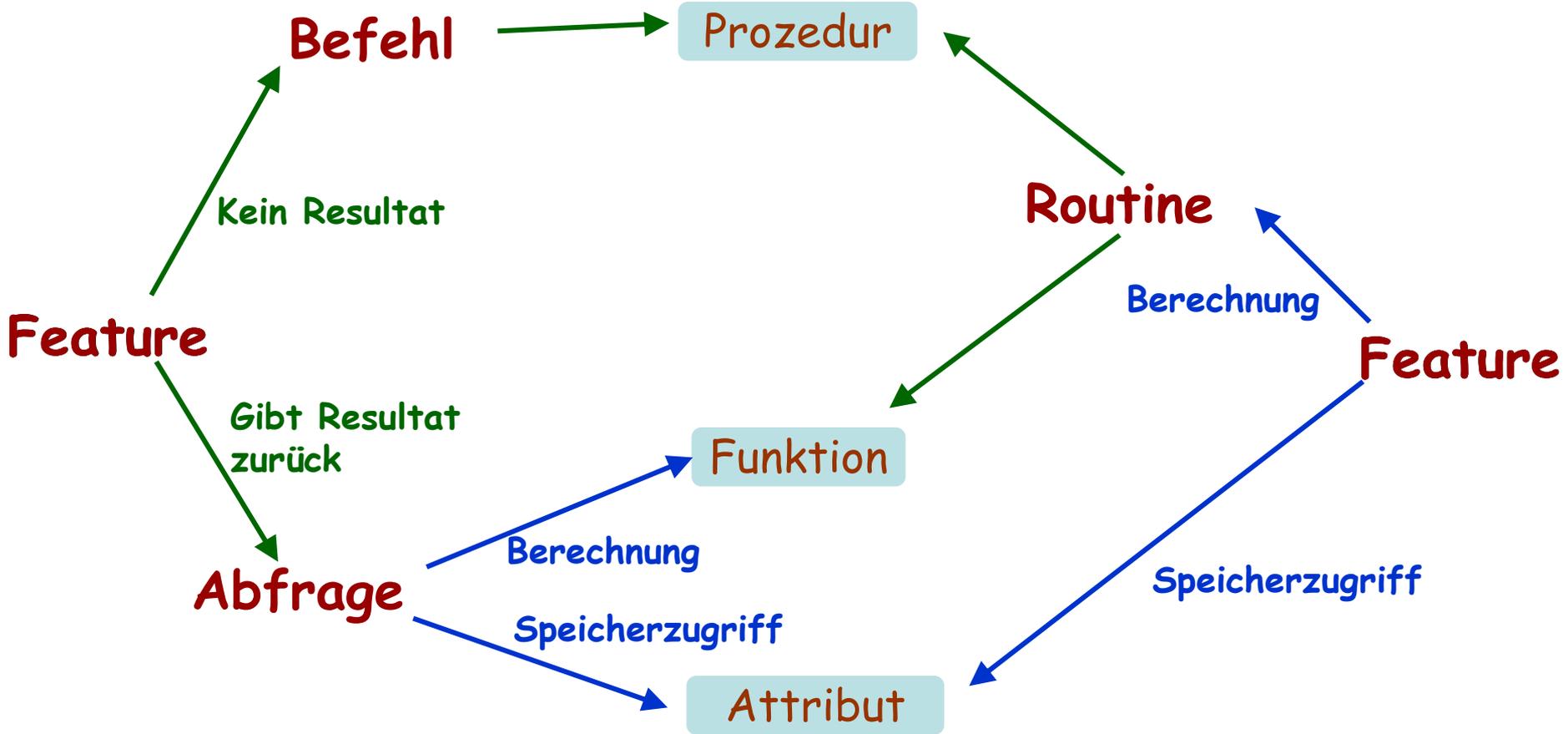


# Features: die ganze Wahrheit



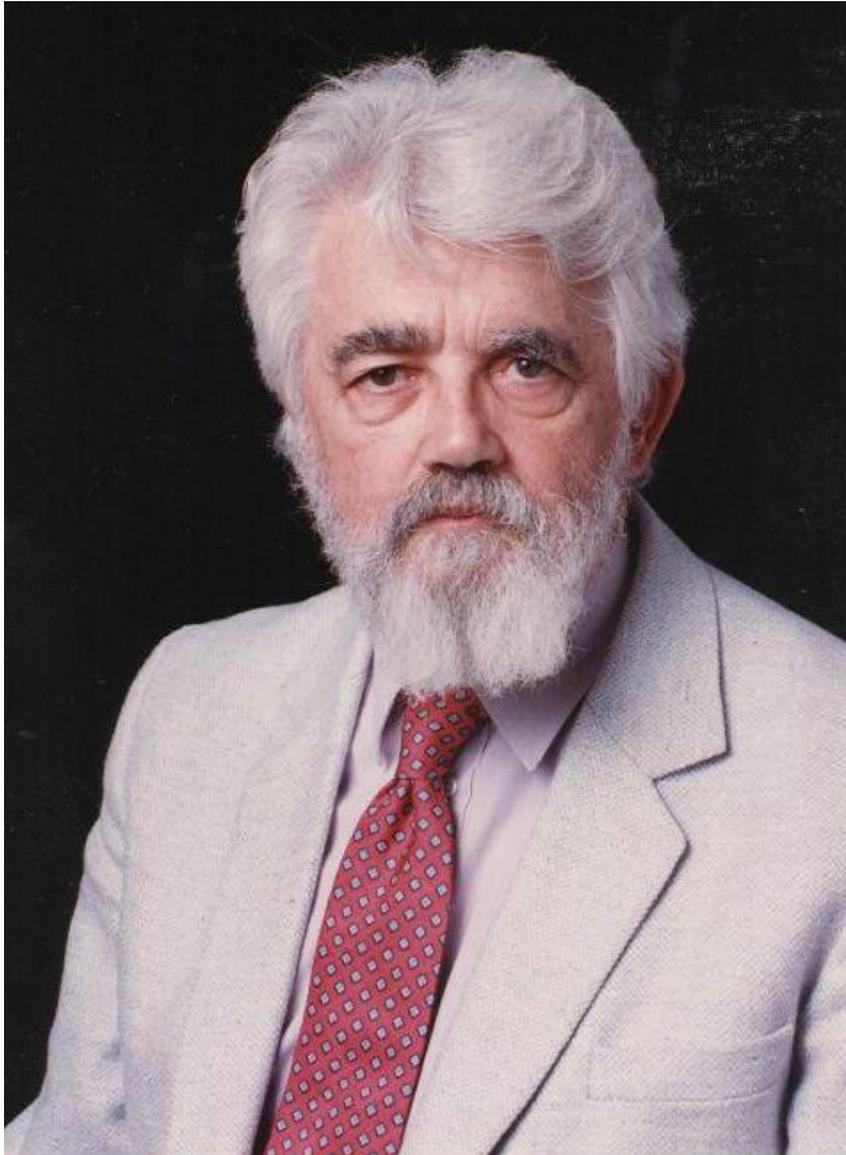
*Kundenansicht  
(Spezifikation)*

*Interne Ansicht  
(Implementation)*



# John McCarthy 1927-2011

---



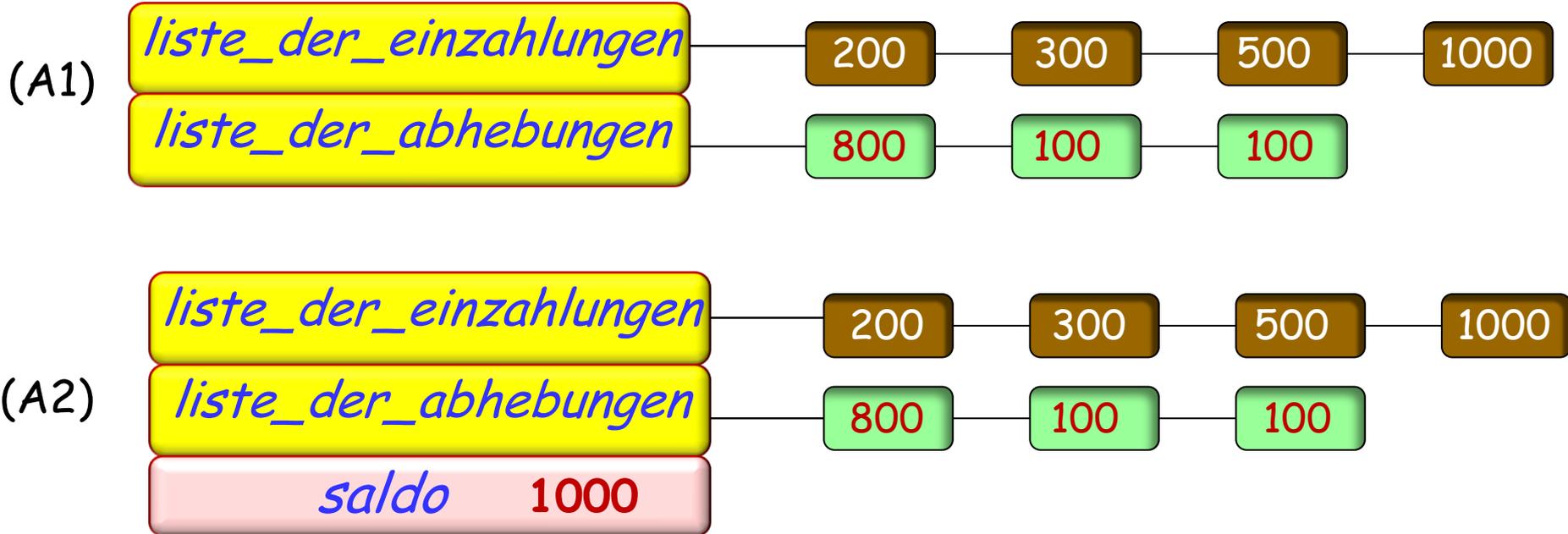
Dem Kunden ist es egal, ob sie  
etwas berechnen oder  
im Speicher nachschauen

*\* Uniform access principle*

# Das Prinzip des einheitlichen Zugriffs: Beispiel



$saldo = liste\_der\_einzahlungen.total - liste\_der\_abhebungen.total$



Ein Aufruf wie z.B. *ihr\_konto.saldo*  
könnte ein Attribut oder eine Funktion benutzen

Dem Kunden ist es egal, ob sie  
etwas berechnen oder  
im Speicher nachschauen

# Das Prinzip des einheitlichen Zugriffs

---



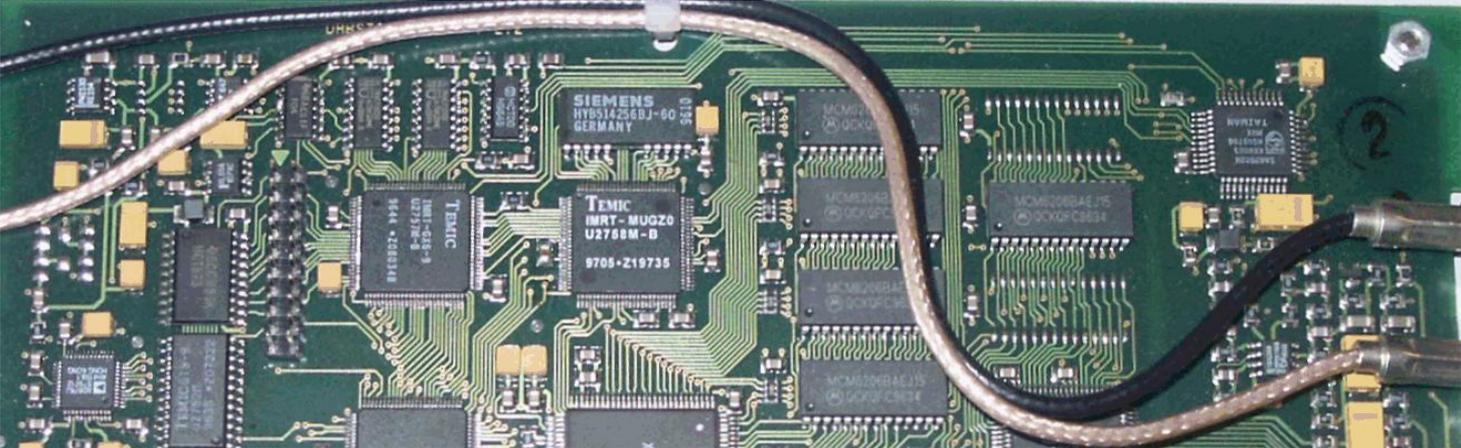
Etwas technischer ausgedrückt:

Eine Abfrage sollte für Kunden auf die gleiche Weise aufrufbar sein, egal ob sie als **Attribut** oder **Funktion** implementiert wurde

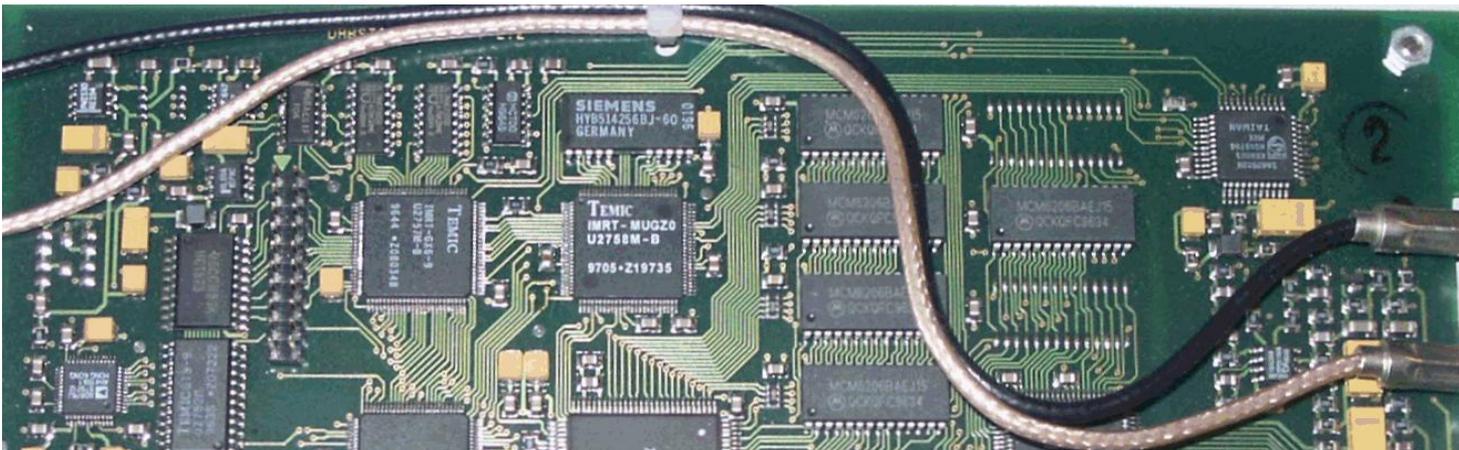
# Ein Objekt hat eine **Schnittstelle**



# Ein Objekt hat eine **Implementation**



# Das Geheimnisprinzip



# Was Kunden tun können

---



```
class STATION feature
```

```
  name: STRING
```

```
    -- Name.
```

```
  position: VECTOR
```

```
    -- Position im Bezug auf das Stadtzentrum.
```

```
  set_position(new_x, new_y: REAL)
```

```
    -- Position setzen.
```

```
  do
```

```
    position.set(new_x, new_y)
```

```
  end
```

```
end
```

# Was Kunden **nicht** tun können



```
class STATION feature
```

```
  name: STRING
```

```
    -- Name.
```

```
  position: VECTOR
```

```
    -- Position im Bezug auf das Stadtzentrum.
```

```
  set_position(new_x, new_y: REAL)
```

```
    -- Position setzen.
```

```
  do
```

```
    position.x := new_x  
    position.y := new_y
```

```
  end
```

```
end
```

NICHT ERLAUBT!



*position.set (3, position.y)*

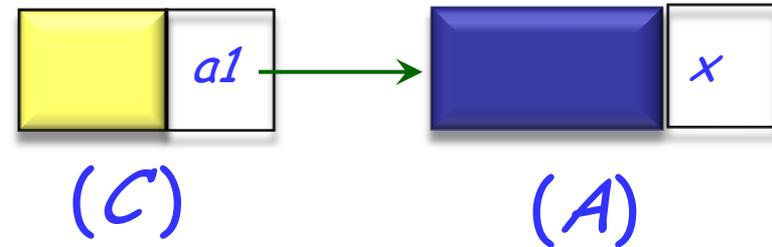
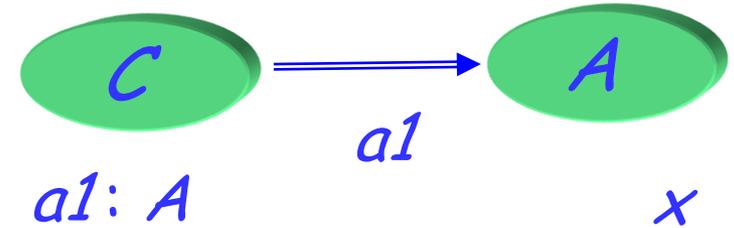
*position.set\_x (3)*

*position.move (0, h)*

# Abstraktion und Kundenprivilegien



Wenn Klasse  $A$  ein Attribut  $x$  hat, was darf eine Kundenklasse  $C$  mit  $a1.x$  tun, wobei  $a1$  vom Typ  $A$  ist?



**Lesezugriff**, falls das Attribut exportiert ist.

$a1.x$  ist ein Ausdruck!

➤ Eine Zuweisung  ~~$a1.x := v$~~  wäre syntaktisch ungültig!!

(Es würde einem Ausdruck etwas zuweisen, wie z.B.:  ~~$a + b := v$~~ )

# Abstraktionsprinzipien anwenden



Um Kunden Schreibprivilegien zu ermöglichen: Definieren sie eine **Setter-Prozedur**, wie z.B.:

```
set_temperature (u: REAL)  
    -- Setzt temperature auf u.  
do  
    temperature := u  
ensure  
    temperature_set: temperature = u  
end
```

Kunden können diese wie folgt aufrufen:

```
x.set_temperature (21.5)
```

# Setter-Befehle voll ausnutzen



*set\_temperature* (*u*: REAL)

-- Setzt Temperaturwert auf *u*.

**require**

nicht\_unter\_minimum:  $u \geq -273$

nicht\_über\_maximum:  $u \leq 2000$

**do**

*temperature* := *u*

*update\_database*

**ensure**

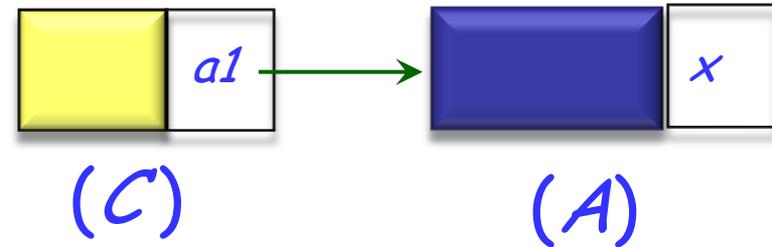
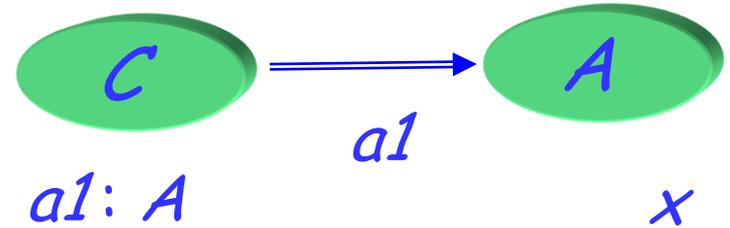
temperatur\_gesetzt: *temperature* = *u*

**end**

# Abstraktion und Kundenprivilegien



Wenn Klasse  $A$  ein Attribut  $x$  hat, was darf eine Kundenklasse  $C$  mit  $a1.x$  tun, wobei  $a1$  vom Typ  $A$  ist?



Lesezugriff, falls das Attribut exportiert ist.

$a1.x$  ist ein Ausdruck!

➤ Eine Zuweisung  ~~$a1.x := v$~~  wäre syntaktisch ungültig!!

(Es würde einem Ausdruck etwas zuweisen, wie z.B.:  ~~$a + b := v$~~ )

# Exportieren (als public deklarieren) eines Attributes



Ein Attribut exportieren heisst in Eiffel, (nur) seine Leserechte zu exportieren

Von ausserhalb erkennt man es nicht als Attribut, nur als **Abfrage**: es könnte auch eine Funktion sein

In C++, Java und C#, werden mit der public-Deklaration eines Attributs\*  $x$  sowohl Schreib- als auch Leserechte exportiert:

➤  $v := a1.x$

➤  $a1.x := v$

Dies führt dazu, dass es fast immer eine schlechte Idee ist, ein Attribut zu exportieren

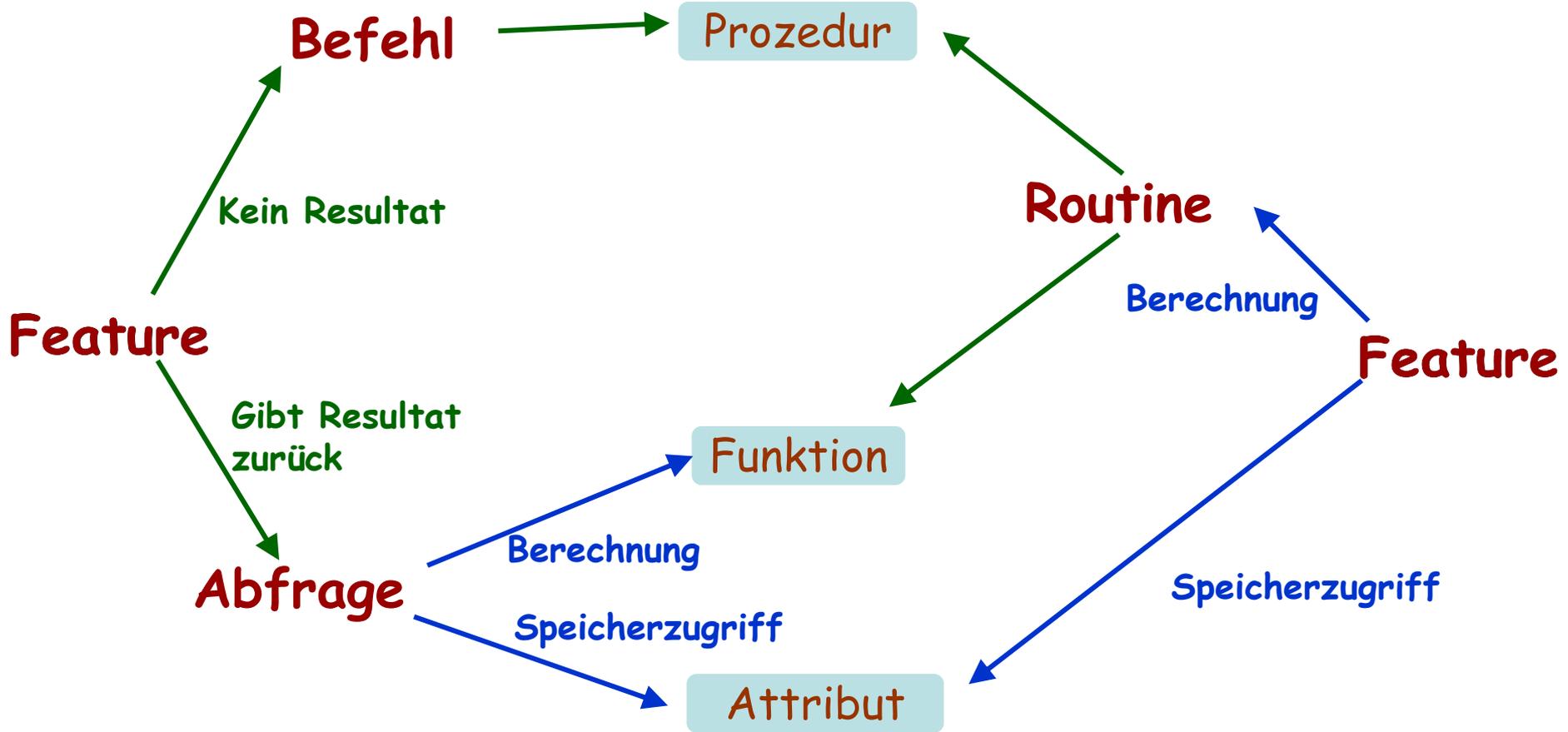
\* (field, member variable)

# Features: die ganze Wahrheit



*Kundenansicht  
(Spezifikation)*

*Interne Ansicht  
(Implementation)*



In C++, Java und C# ist die Standardtechnik, um ein privates Attribut *private\_x* zu exportieren, das Exportieren einer entsprechenden **Getter-Funktion**:

```
x: T
    do
        Result := private_x
    end
```

Eiffel braucht keine Getter-Funktionen: Man kann einfach das Attribut exportieren

Das Attribut wird wie folgt exportiert:

- Nur Leserechte
- **Ohne die Information, dass es ein Attribut ist.** Es könnte auch eine Funktion sein. (Prinzip des einheitlichen Zugriffs)

# Wir wollen beide Arten! (Eiffel-Syntax)



Es ist möglich, eine Abfrage wie folgt zu definieren:

*temperature: REAL* **assign** *set\_temperature*

Dann wird folgende Syntax

*x.temperature := 21.5*

Keine Zuweisung, sondern  
ein Prozeduraufruf!

akzeptiert als Abkürzung für

*x.set\_temperature(21.5)*

Erhält **Verträge** und andere ergänzende Operationen.

In C# gibt es den Begriff des "Property", womit das gleiche Ziel verfolgt wird

# Das Geheimnisprinzip (Information Hiding)



Status der Aufrufe in einem Kunden  
mit *a1: A*:

```
class
  A

feature
  f ...
  g ...

feature {NONE}
  h, i ...

feature {B, C}
  j, k, l ...

feature {A, B, C}
  m, n ...

end
```

- *a1.f, a1.g*: in jedem Kunden gültig.
- *a1.h*: überall **ungültig**  
(auch in *A*s eigenem Klassentext!)
- *a1.j*: nur in *B, C* und deren Nachkommen gültig  
(Nicht gültig in *A*!)
- *a1.m*: nur in *A, B, C* und deren Nachkommen gültig.

# Das Geheimnisprinzip



Das Geheimnisprinzip gilt nur für Benutzung durch Kunden, mittels *qualifizierten* Aufrufen oder Infix-Notation, z.B.: *a1.f*

*Unqualifizierte* Aufrufe (innerhalb einer Klasse) sind vom Geheimnisprinzip nicht betroffen:

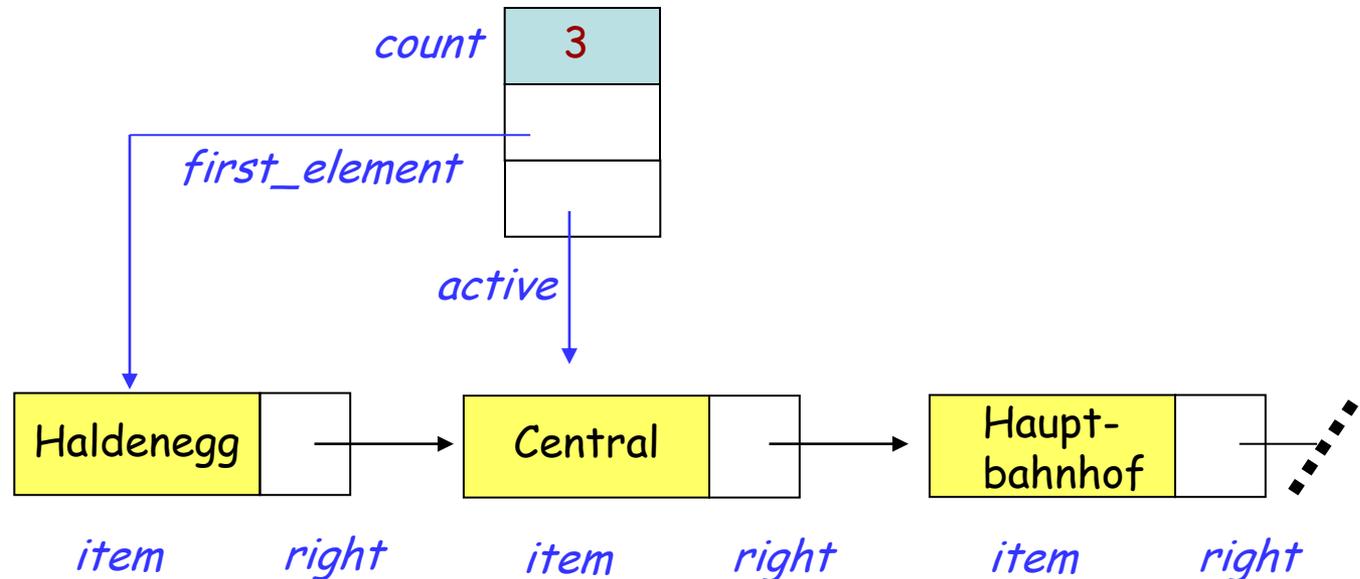
```
class A feature {NONE}
  h do ... end
feature
  f
    do
      ...; h; ...
    end
end
```

# Ein Beispiel für selektiven Export



*LINKABLE* exportiert ihre Features an *LINKED\_LIST*

- Exportiert sie nicht für den Rest der Welt.
- Kunden von *LINKED\_LIST* müssen nichts über die *LINKABLE*-Zellen wissen.



class

*LINKABLE[G]*

Diese Features werden selektiv an *LINKED\_LIST* und ihre Nachkommen exportiert. (Und zu keinen weiteren Klassen.)

feature {*LINKED\_LIST*}

*put\_right*(...) do ... end

*right: G* do ... end

...

end

# LINKABLE



```
class LINKABLE feature {LINKED_LIST}
```

```
  item: STRING
```

```
    -- Wert dieser Zelle.
```



*item*

*right*

```
  right: LINKABLE
```

```
    -- Zelle, welche rechts von dieser Zelle
```

```
    -- angehängt ist (falls vorhanden).
```

```
  put_right(other: like Current)
```

```
    -- Setzt other rechts neben die aktuelle Zelle.
```

```
  do
```

```
    right := other
```

```
  ensure
```

```
    verkettet: right = other
```

```
  end
```

```
end
```



Die volle Kategorisierung von Features

Routinen, Prozeduren, Funktionen

Einheitlicher Zugriff

Geheimnisprinzip

Selektives exportieren

Setter- und Getter-Funktionen

Eiffel: Assigner-Befehle



Kapitel über

➤ **Inheritance (16)**