ETH Zürich
Chair of Software Engineering

Einführung in die Programmierung

Prof. Dr. Bertrand Meyer

Lektion 9: Abstraktion

Heutige Themen

Abstraktion, vor allem funktionale Abstraktion

Der Begriff der Routine

Das Abschlusswort zu Features: Alle Featurekategorien

Das Prinzip des einheitlichen Zugriffs (the Uniform Access principle)

Abstraktionen und Kundenprivilegien

Das Geheimnisprinzip

Routine: eine Abstraktion eines Algorithmus

Abstrahieren heisst, die *Essenz* eines Konzeptes zu erfassen und Details und genaue 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 Arten von Features…

… die andere Art sind die *Attribute*

Wir sind schon vielen Routinen begegnet, allerdings ohne den Namen zu kennen.

Eine Routine

```
r (arg: TYPE; ...)
                -- Kopfkommentar.
        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_a_line
        -- Imaginäre Linie bauen.
    do
        Paris.display
        Metro.highlight
        create_fancy_line
    end
```

```
create_fancy_line
        -- Linie erschaffen und
        -- Stationen ausfüllen.
    do
        -- TODO
        -- BM, 26 Oct 2010
    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 ist durch ihre Features charakterisiert.

Jedes Feature ist eine Operation auf die korrespondierenden Elemente: Abfrage oder Befehl.

Features sind der Leserlichkeit halber in verschiedene Kategorien eingeteilt.

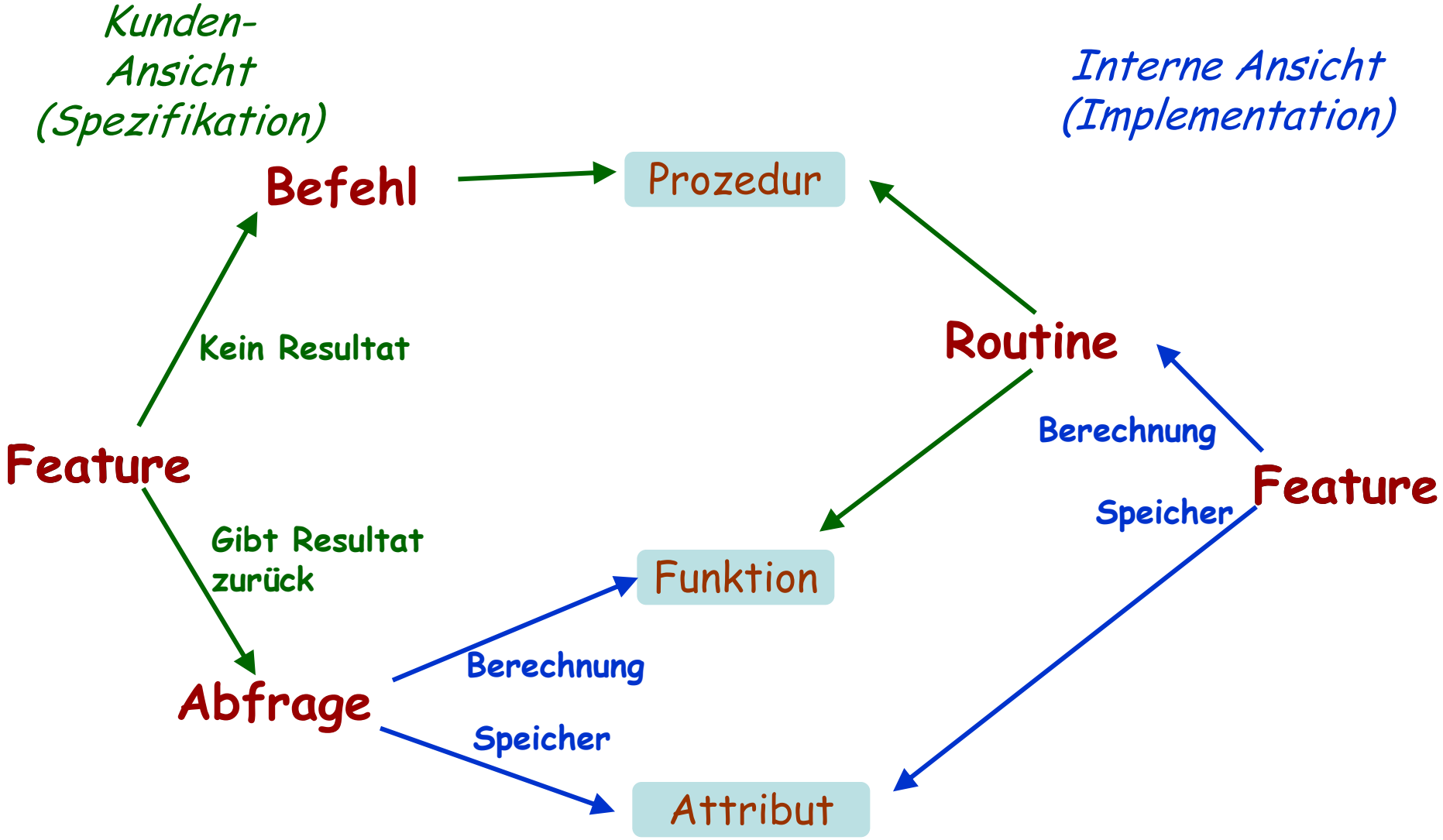
Klassenklauseln:

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

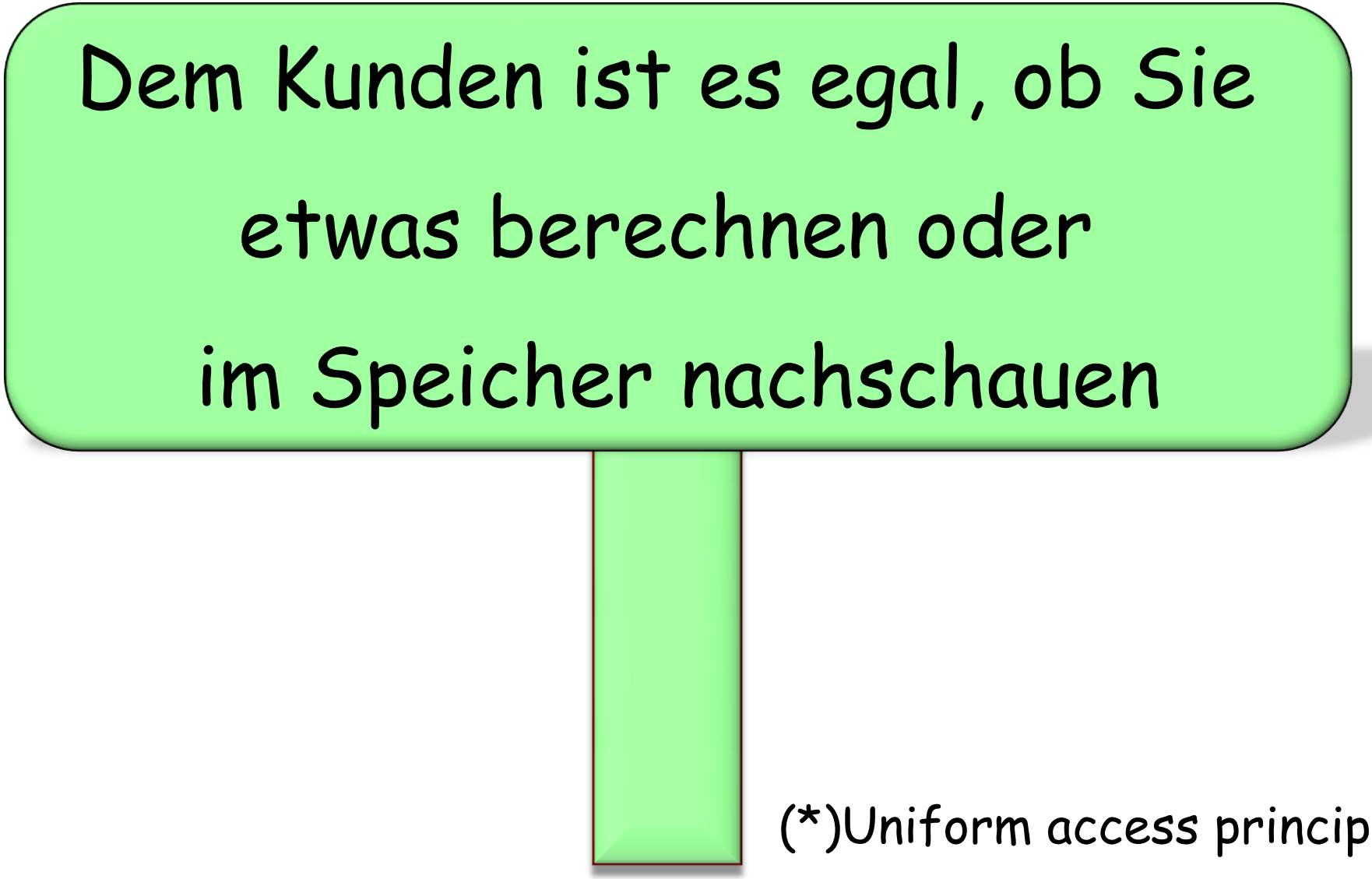
Anatomie einer Klasse:

Demo

Features: die ganze Wahrheit



Das Prinzip des einheitlichen Zugriffs(*)

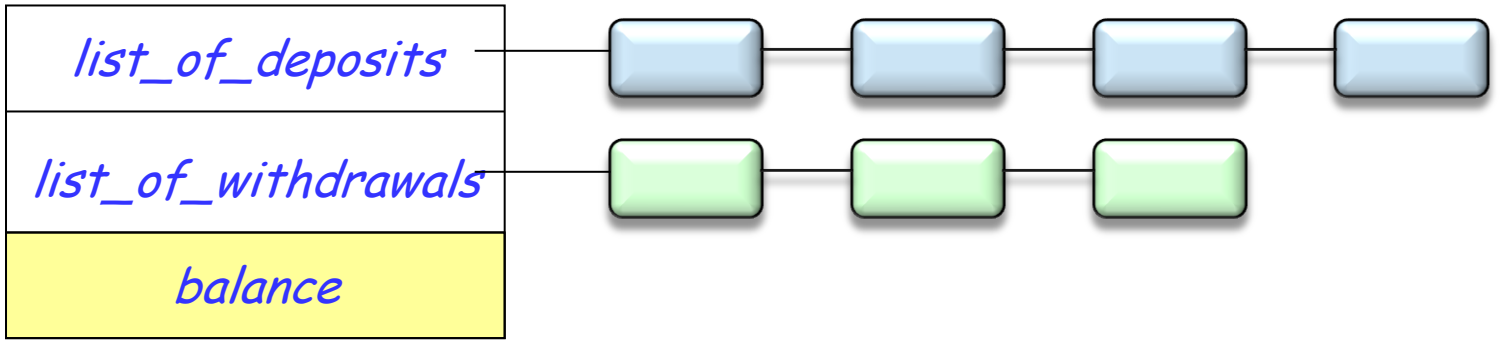


(*)Uniform access principle

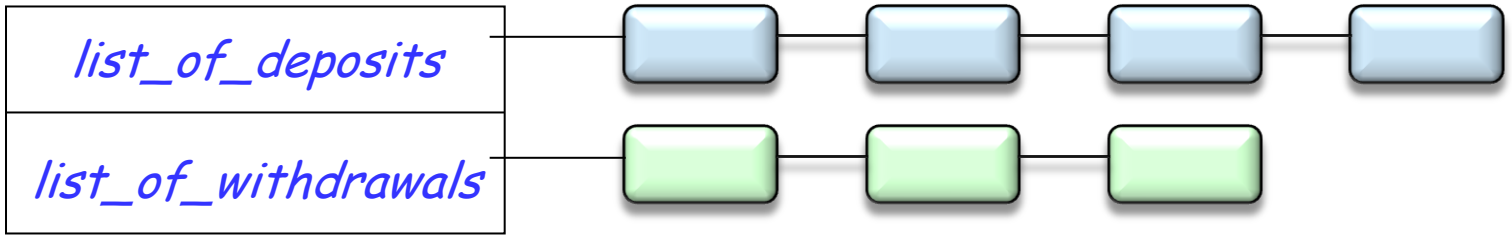
Das Prinzip des einheitlichen Zugriffs: Beispiel

balance = list_of_deposits.total - list_of_withdrawals.total

(A1)

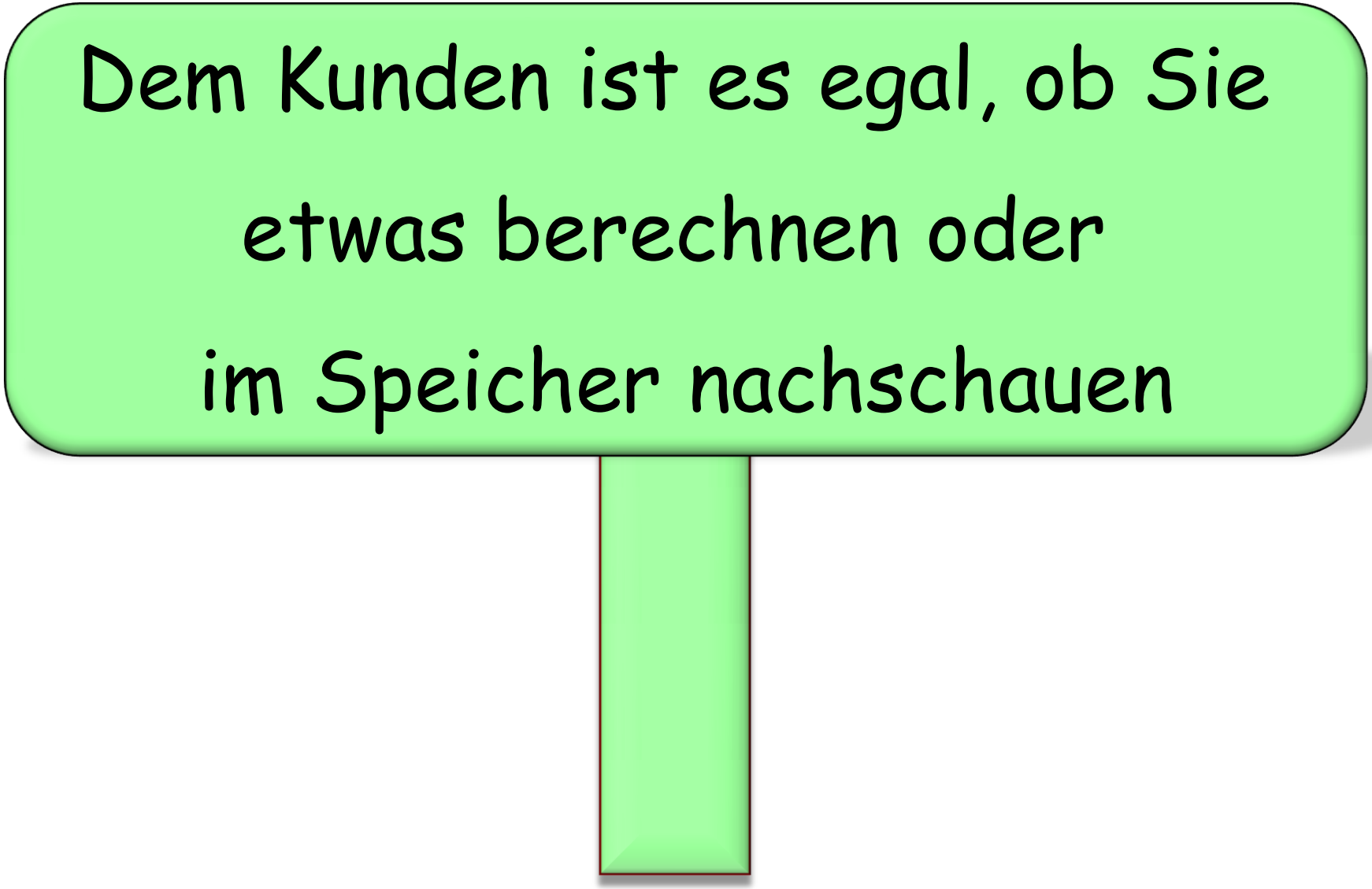


(A2)



Ein Aufruf wie z.B. *your_account.balance*
könnte ein Attribut oder eine Funktion benutzen.

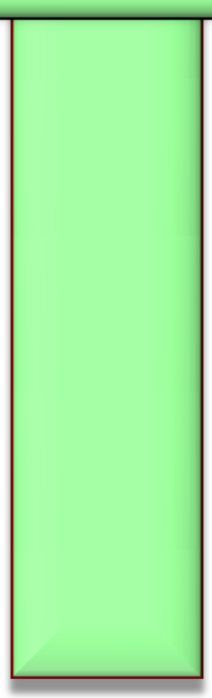
Das Prinzip des einheitlichen Zugriffs



Das Prinzip des einheitlichen Zugriffs

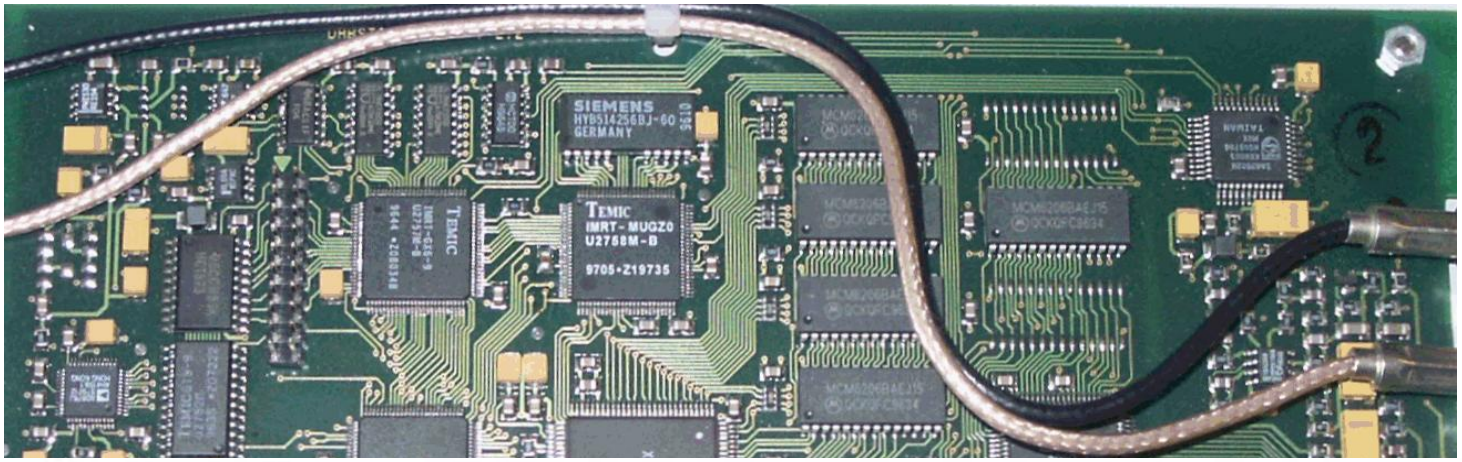
Etwas technischer ausgedrückt:

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

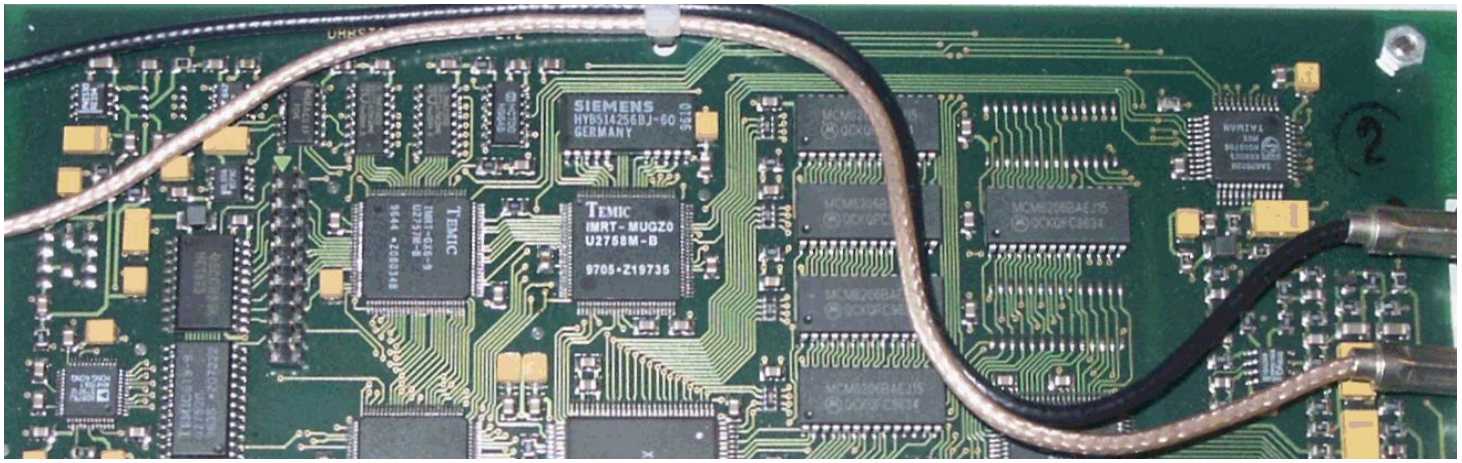


Ein Objekt hat eine **Schnittstelle**

Ein Objekt hat eine Implementation



Das Geheimnisprinzip



Was Kunden tun können

```
class METRO_STATION feature

        x, y: REAL
                -- Koordinaten der Metrostation
        size: REAL
                -- Grösse des umschliessenden Rechtecks

        upper_left: POSITION
                -- Obere linke Position des umschliessenden Rechtecks

        adjust_positions
                                -- Position des umschliessenden Rechtecks
                                -- ändern
                        do
                                upper_left . set (x - size/2, y + size/2)
                                ...
                         end

end
```

Was Kunden nicht tun können

```
class METRO_STATION feature

        x, y: REAL
                -- Koordinaten der Metrostation
        size: REAL
                -- Grösse des umschliessenden Rechtecks

        upper_left: POSITION
                -- Obere linke Position des umschliessenden Rechtecks

        adjust_positions
                                -- Position des umschliessenden Rechtecks
                                -- ändern
                        do
                                upper_left . x := 3
                                ...
                         end

end
```

NICHT ERLAUBT!

Benutzen Sie Prozeduren!

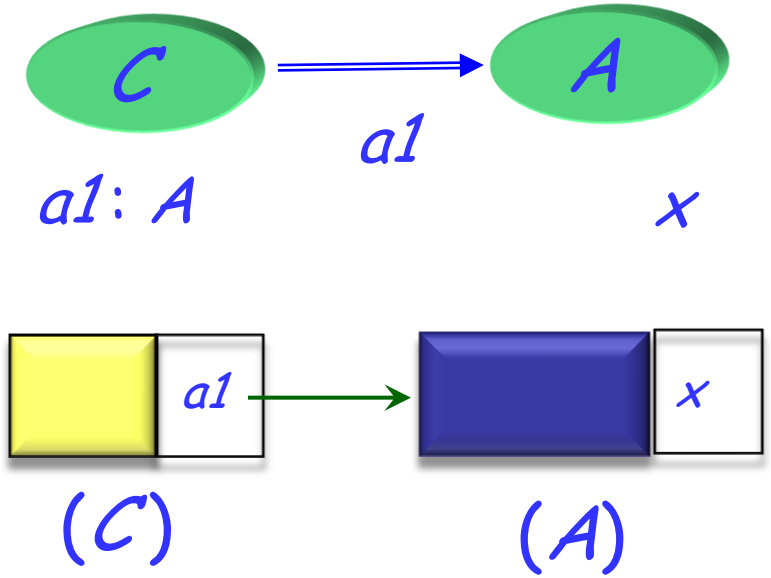
```
upper_left.set(3, upper_left.y)

upper_left.set_x(3 )

upper_left.move(3, 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

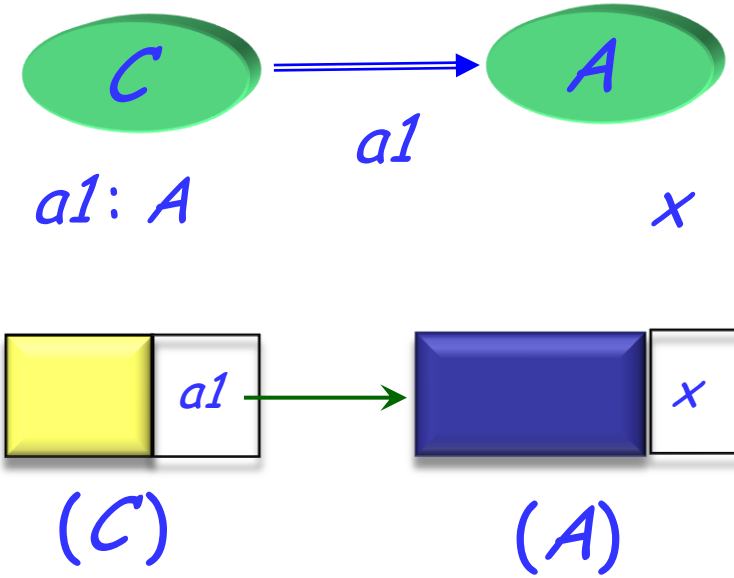
```
set_temperature (u : REAL)
            -- Setzt Temperaturwert auf u.
    require
        not_under_minimum: u >= -273
        not_above_maximum: u <= 2000
    do
        temperature := u
        update_database

    ensure
        temperature_set: 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.

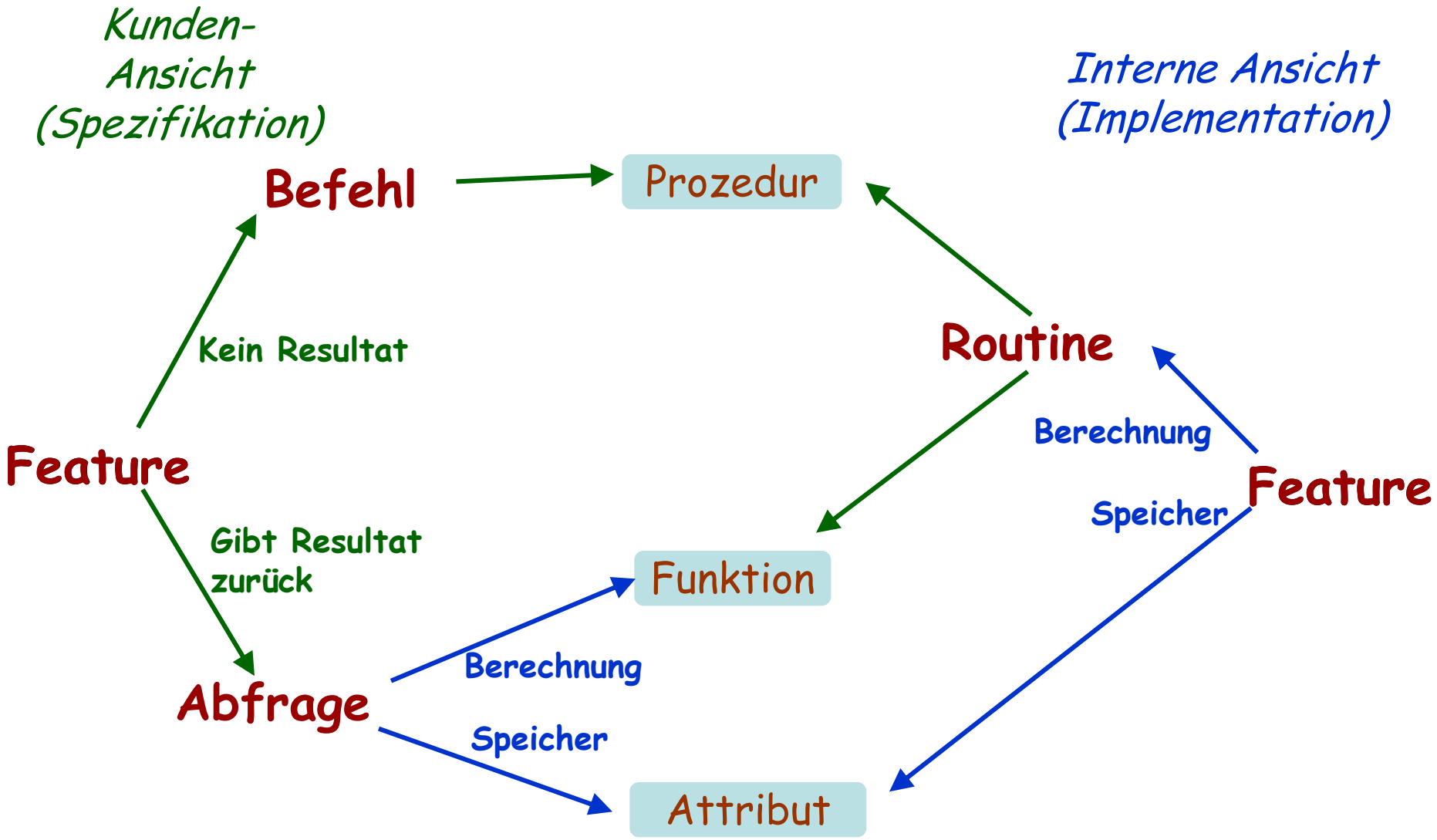
In C++, Java und C#, werden mit der public-Deklaration eines Attributs* *x* sowohl Schreib- als auch Leserechte exportiert:

- *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



Getter-Funktionen

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

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

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

Dies ist sicher: 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 Prozedurenaufruf!

akzeptiert als Abkürzung für

x.set_temperature (21.5)

Erhält **Verträge** und andere ergänzende Operationen.

In C# gibt es den Begriff des "Property", welcher das gleiche Ziel verfolgt.

Das Geheimnisprinzip (Information Hiding)

```
class
    A

feature
        f ...
        g ...

feature {NONE}

        h, i ...

feature {B, C}

        j, k, l ...

feature {A, B, C}

        m, n ...
end
```

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

- *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*

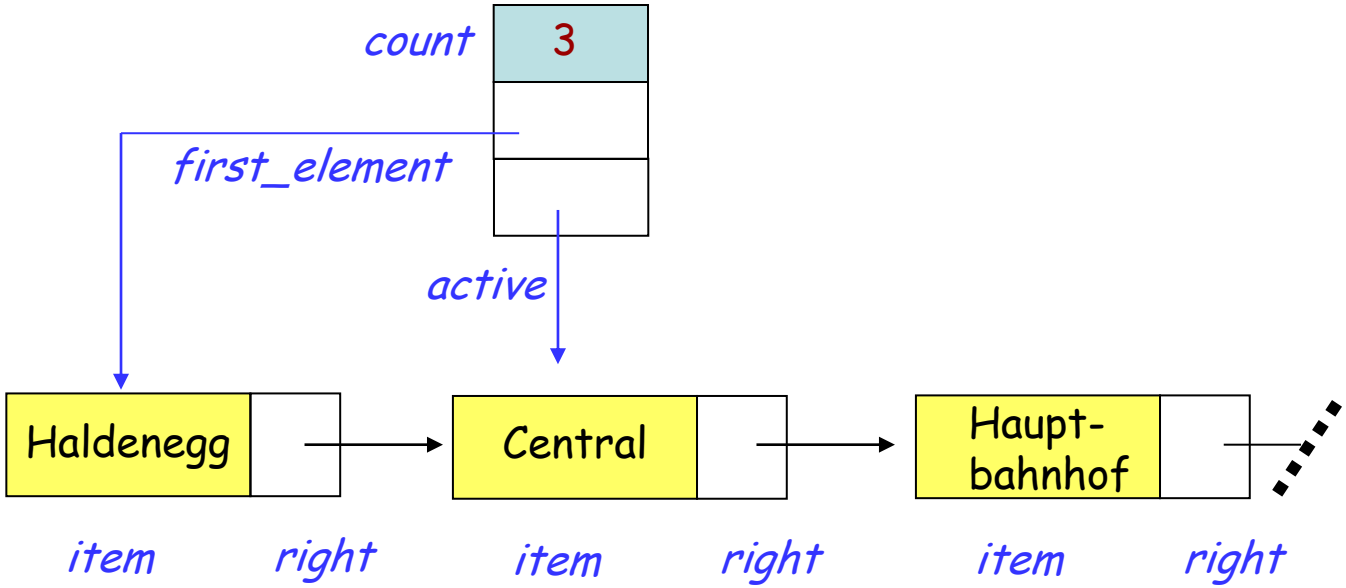
Unqualifizierte Aufrufe (innerhalb einer Klasse) sind vom Geheimnisprinzip nicht betroffen:

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

Ein Beispiel für selektiven Export

LINKABLE exportiert ihre Features nach *LINKED_LIST*

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



Selektiv Exportieren

```
class

    LINKABLE [G]

feature {LINKED_LIST}

        put_right (...) do ... end

        right: G do ... end

        ...
end
```

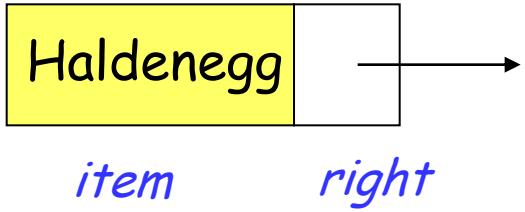
Diese Features werden selektiv nach *LINKED_LIST* und ihre Nachkommen exportiert. (Und zu keinen weiteren Klassen)

LINKABLE

```
class LINKABLE feature {LINKED_LIST}
        item: STRING
                        -- Wert dieser Zelle

        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
                        chained : right = other
                end
end
```



Was wir gesehen haben

Die volle Kategorisierung von Features

Routinen, Prozeduren, Funktionen

Einheitlicher Zugriff

Geheimnisprinzip

Selektives Exportieren

Setter- und Getter-Funktionen

Eiffel: Assigner-Befehle

Leseaufgabe auf nächste Woche

Kapitel über

- Syntax (11)
- Inheritance (16)