

- 2.1 Komponenten und Dienste**
- 2.2 Objekte und Objektklassen**
- 2.3 Typisierung von Programmen**
- 2.4 Typisierung von Diensten**
- 2.5 Algorithmenwahl**



## Innensicht

### Imperatives Programmieren

(Konstrukte aus Java)

- Praxis: Basistypen, Anweisungen, Verbunde, Felder, Schleifen, Rekursion
- Theorie: Syntaxbeschreibung, Induktion, Schleifeninvarianten
- Ausnahmebehandlung

### Objektorientiertes Programmieren (Java)

- OO-Klassen
- Objekt und Zustände
- Methoden und Vererbung in Java
- Java: Einführung aller restlicher OO-Sprachkonstrukte

## Außensicht

### Dienste

- Funktionalität/Schnittstellen
- Qualitätsparameter: Aufwand, Zuverlässigkeit, Skalierbarkeit, Persistenz
- Algorithmenwahl

### Objektorientierung

- Objekte, Zustände, Methoden
- Entwurf mittels UML
- Vererbung



## System:

- Abgegrenzter Ausschnitt aus der realen oder gedanklichen Welt zur Erfüllung eines gegebenen Zwecks. Ein System wird bestimmt durch die Beziehungen zu seiner Umwelt, durch die in ihm enthaltenen Bestandteile und deren Beziehungen zueinander sowie durch das dynamische Verhalten.



H.-J. Schneider; Lexikon der Informatik und Datenverarbeitung;  
Oldenbourg-Verlag, 1997

- Kollektion von Gegenständen, die in einem inneren Zusammenhang stehen, samt den Beziehungen zwischen diesen Gegenständen.

## Es lässt sich unterscheiden:

- *Innensicht*
  - Bausteine, deren Beziehungen
- *Außersicht*
  - Systemgrenze, Beziehungen über diese Grenze



G. Goos; Vorlesungen über Informatik: Band 1;  
Springer-Verlag, 1995

## Kapitel 1:

- Betrachtungen zur Innensicht

## Kapitel 2:

- Erste Schritte zur Außersicht



### Definition Dienst

- Allgemein, die Erfüllung von Pflichten
  - Im beruflichen Bereich die Verrichtung der zu erbringenden Leistung



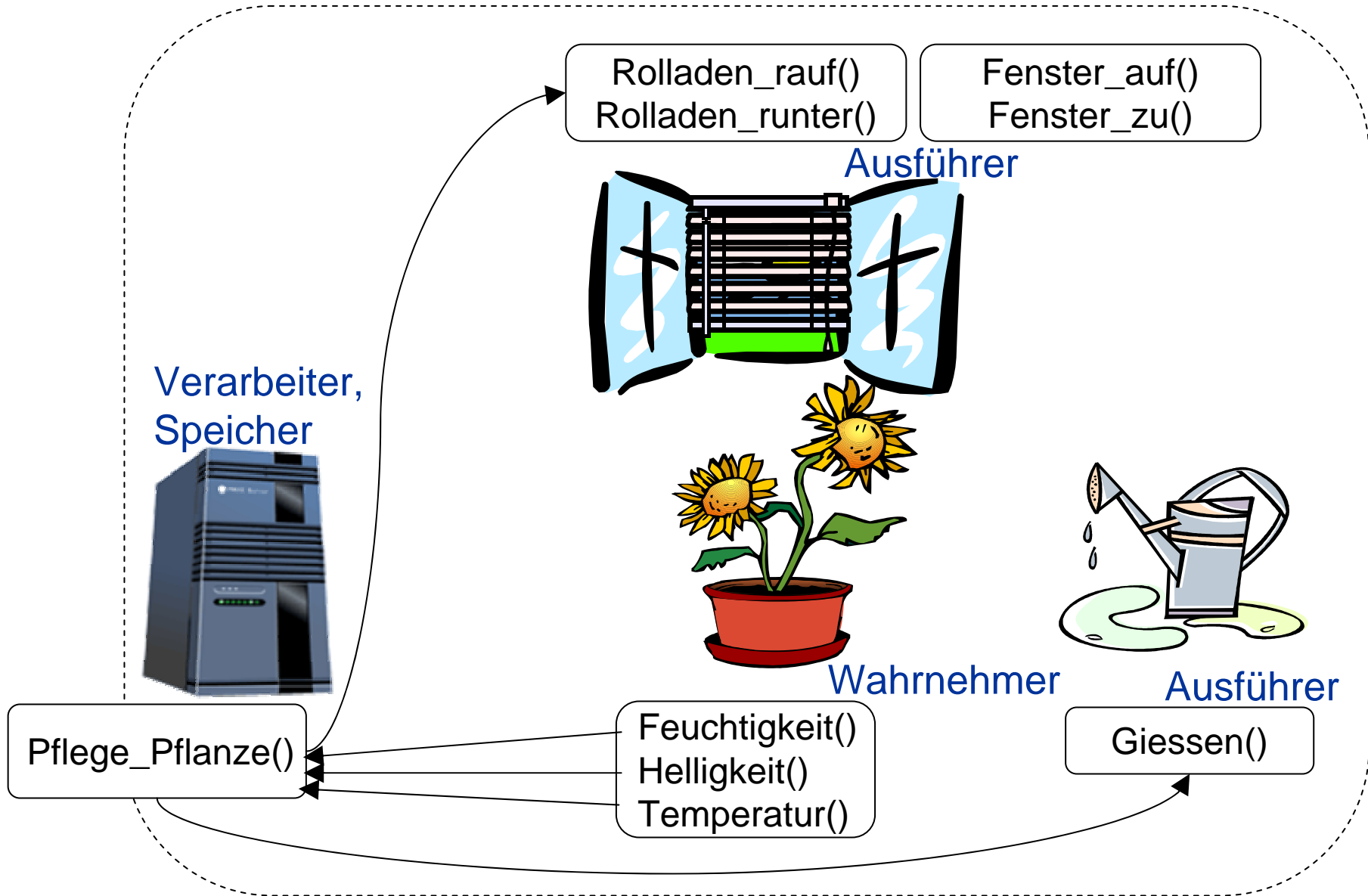
Meyers Großes Taschenlexikon; Band 5,  
B.I. Taschenbuchverlag, 1992

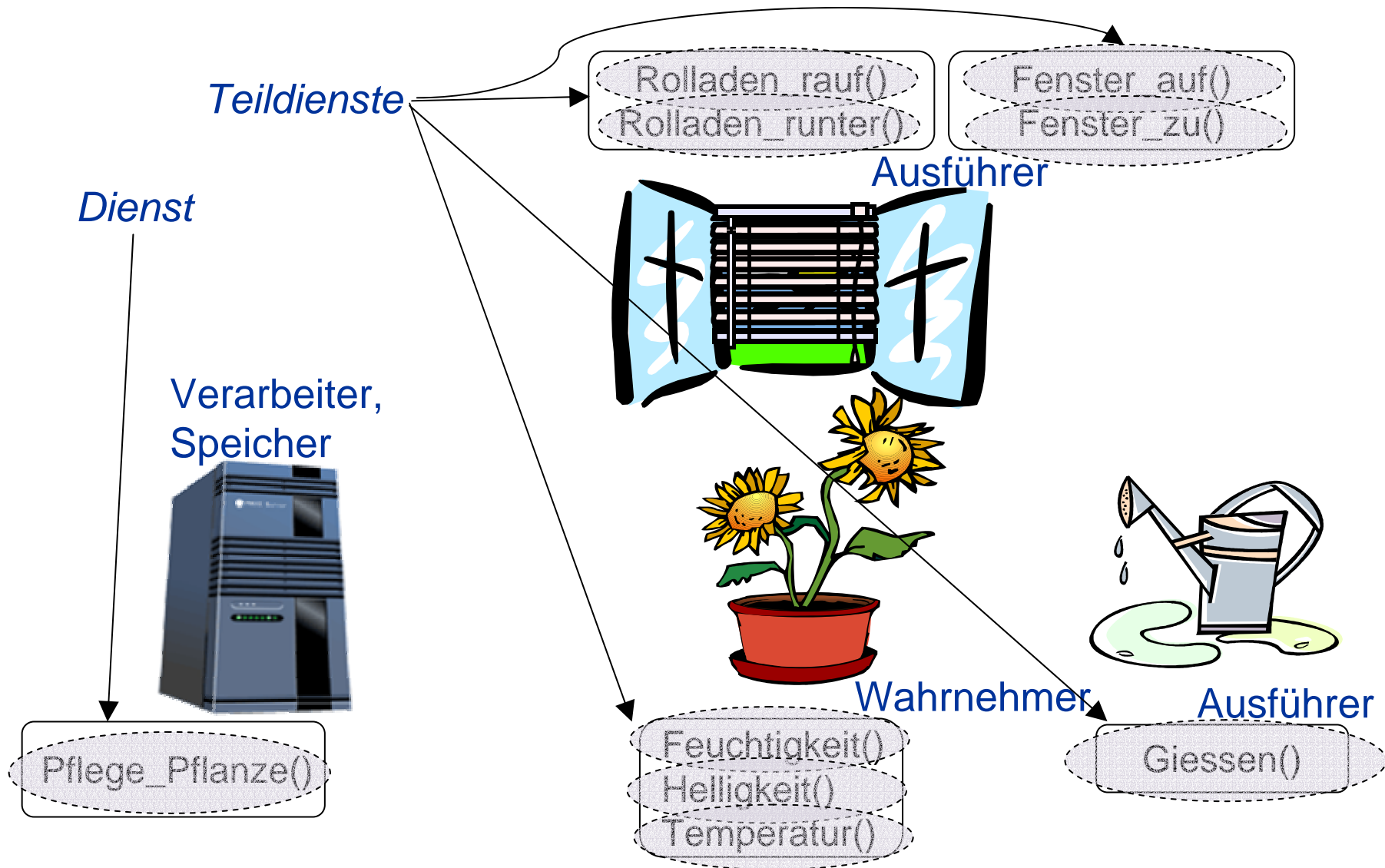
... dies lässt sich direkt auf den Kontext der Vorlesung Informatik 2 übertragen

- Dienst wird sichtbar an der Systemgrenze
- System setzt sich aus Komponenten zusammen, die jeweils einen Teildienst anbieten.
  - Dienst des Systems ergibt sich aus den Teildiensten



# Beispiel Pflanzenpflegedienst: Pflege\_Pflanze ()





Ein Software-**System** ist eine Organisationsstruktur für die Aufgabenteilung in der Informationsverarbeitung.

Eine (Software-)**Komponente** ist Träger von wohldefinierten Aufgaben und somit Baustein eines Software-Systems.

- Ein funktional oder konstruktiv zusammengehöriger abgeschlossener Bestandteil eines Systems. Eine Komponente kann selbst wieder in Komponenten unterteilt sein.

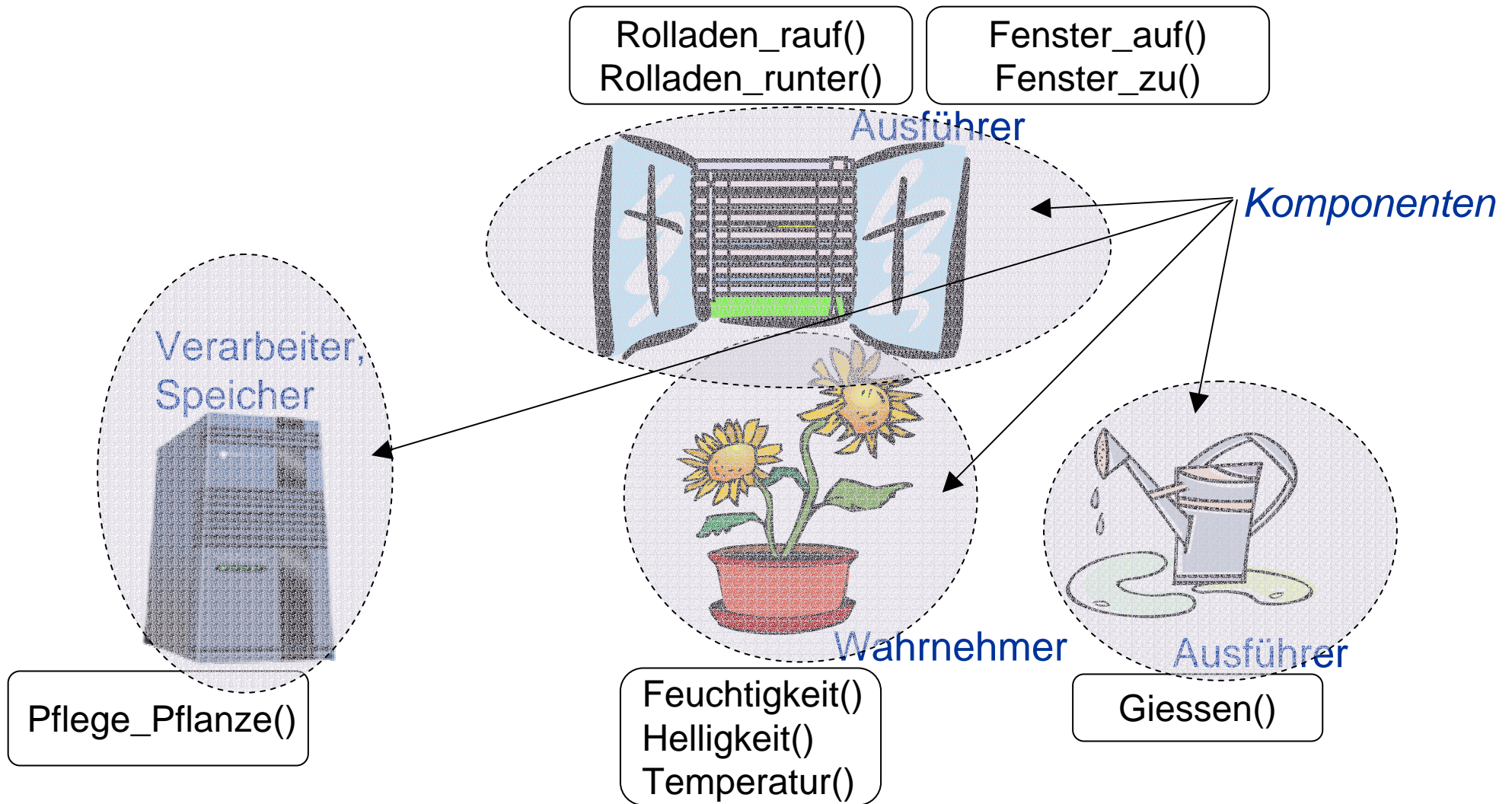


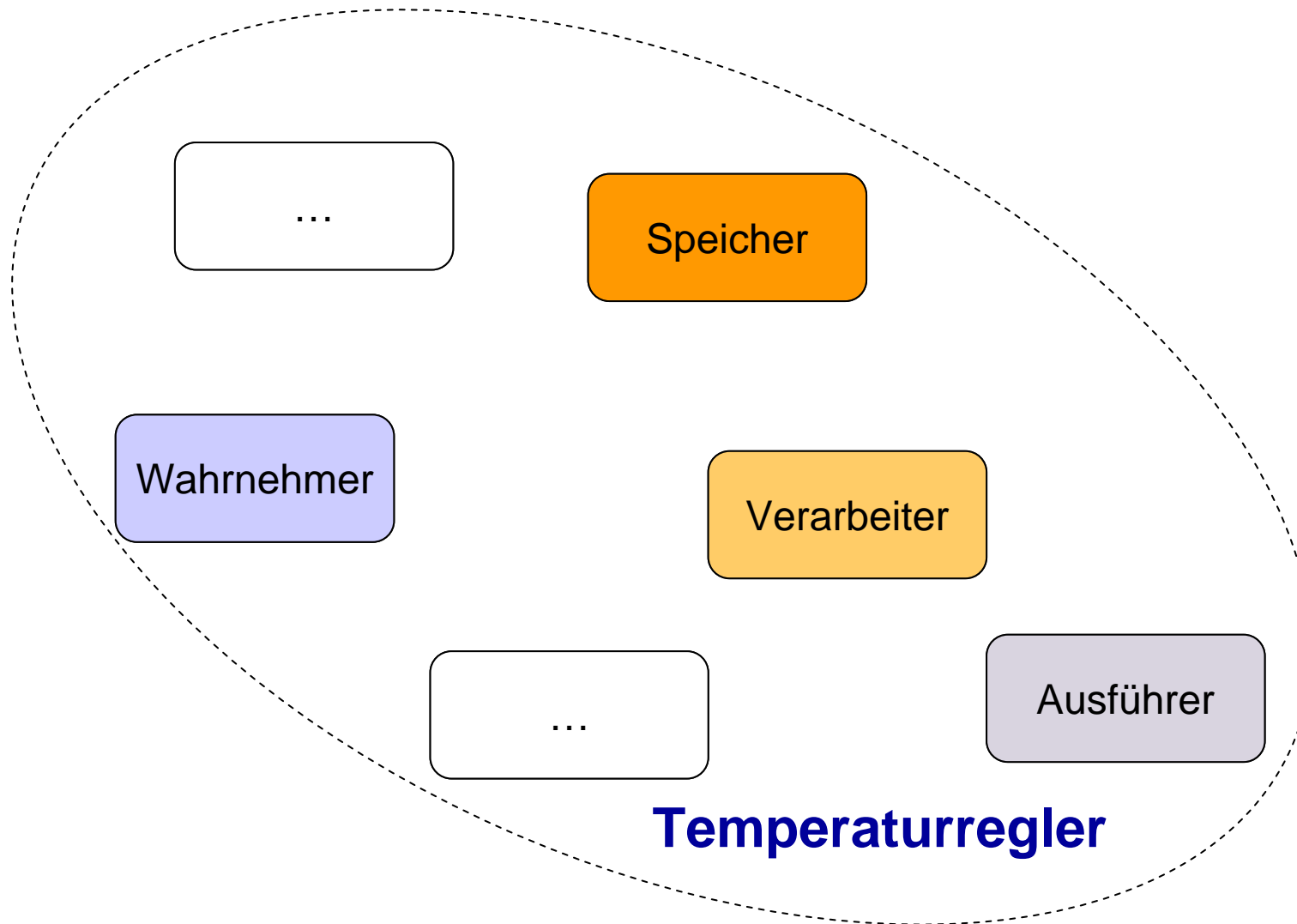
H.-J. Schneider; Lexikon der Informatik und Datenverarbeitung;  
Oldenbourg-Verlag, 1997

Ein Software-System ist somit eine Sammlung von Komponenten, die zur Erfüllung gemeinsamer Aufgaben **zusammenwirken**.

- Ein Software-System kann selbst wieder als Komponente agieren.
- Komponenten interagieren untereinander





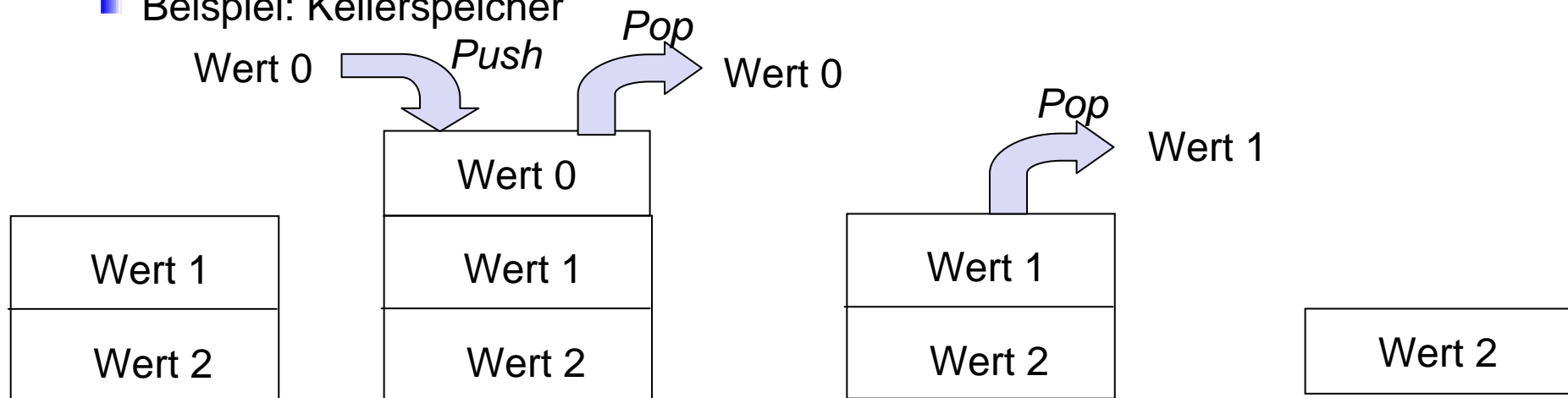


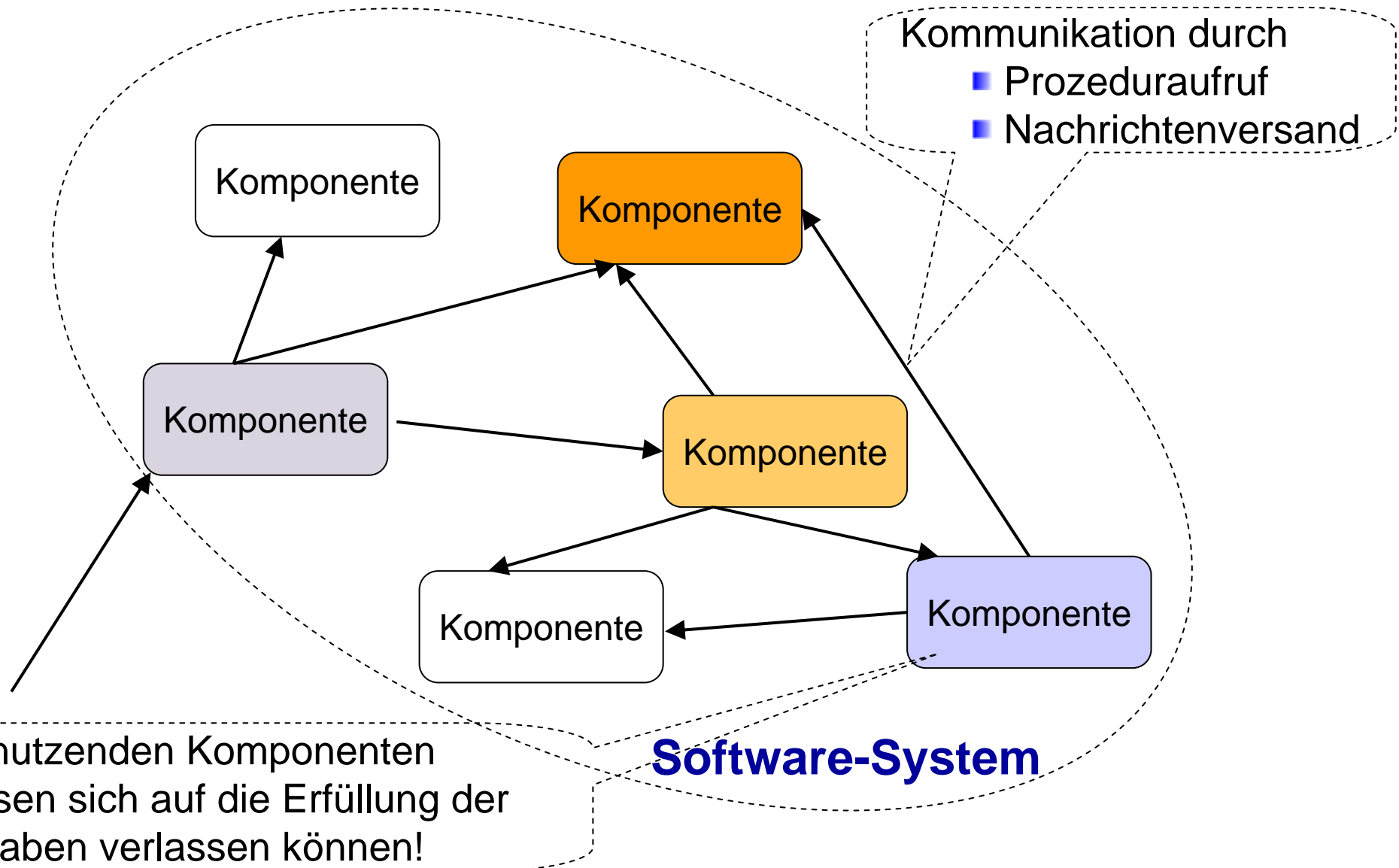
Eine Komponente soll Aufgaben vereinigen, die in einem inneren Zusammenhang stehen (Kohäsion).

- Aufgaben mit geringer Kohäsion sollen durch unterschiedliche Komponenten wahrgenommen werden.

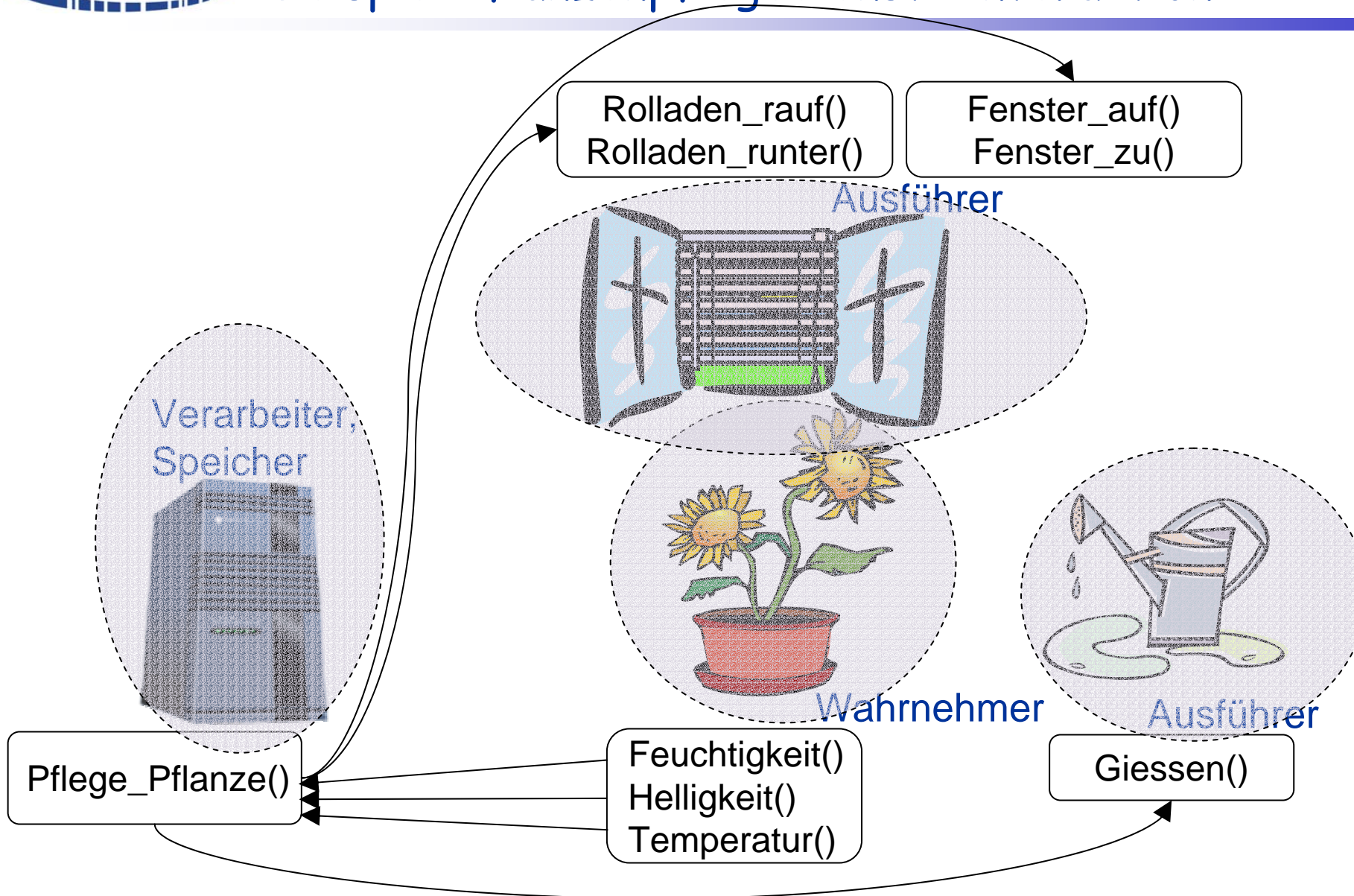
## Extremfälle:

- Komponente bietet eine einzige Funktion.
  - Beispiel: Zufallszahlengenerator
- Komponente bietet mehrere Funktionen
  - Der zeitliche Ablauf der Funktionen definiert einen Zustand.
  - Beispiel: Kellerspeicher





# Beispiel Pflanzenpflegedienst: Interaktion



## Forschung am ITM

- Pflanzenpflegedienst als sehr einfaches Beispiel für Sensornetze

## Komponenten

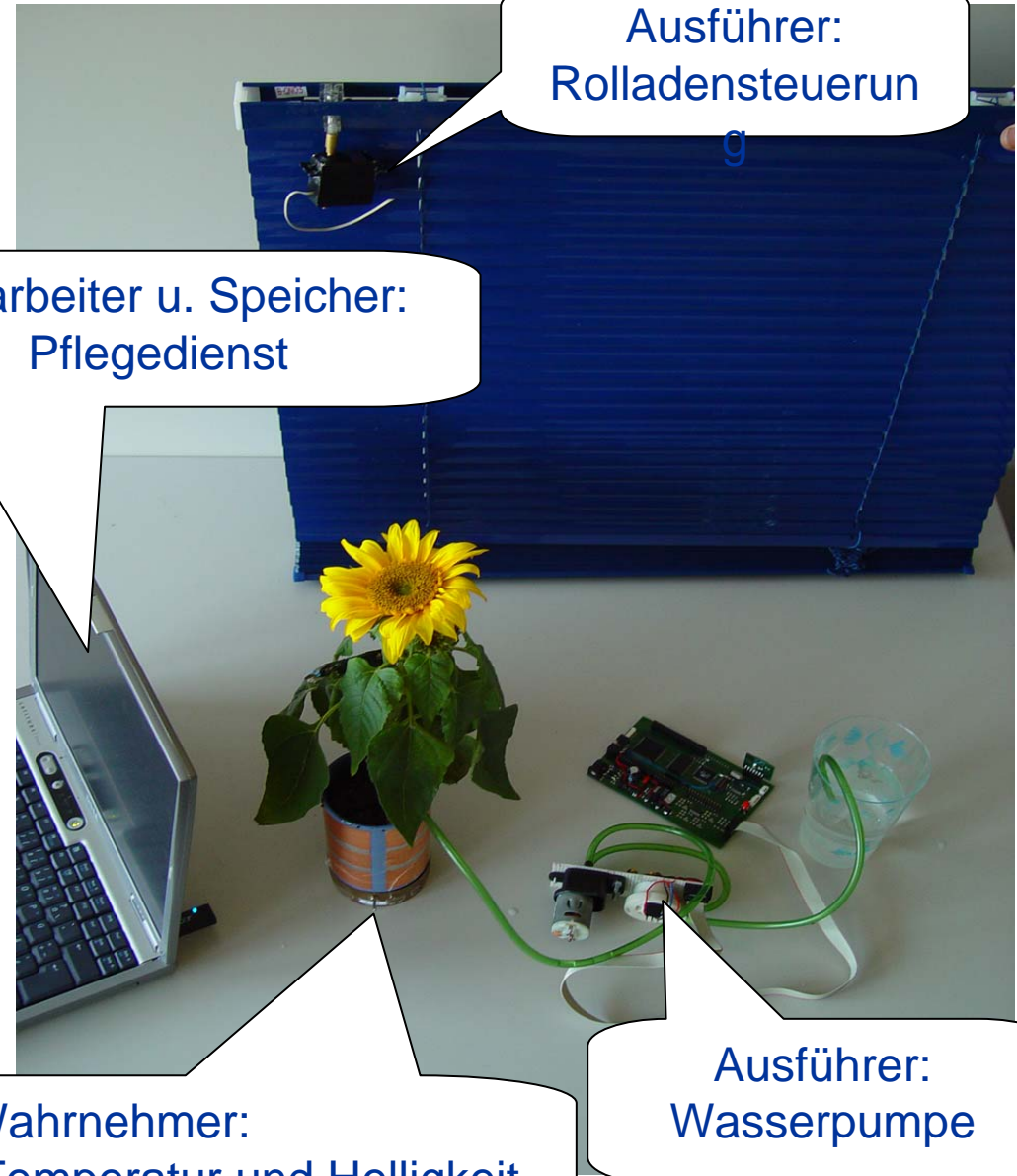
- Wahrnehmer für Temperatur
- Wahrnehmer für Feuchtigkeit
- Verarbeiter, Speicher

## Interaktion

- Drahtlose Kommunikation über Bluetooth

## Weitere Anwendungsbeispiele

- Intelligentes Haus / Bürogebäude
- Assisted Living
- eHealth
- ...

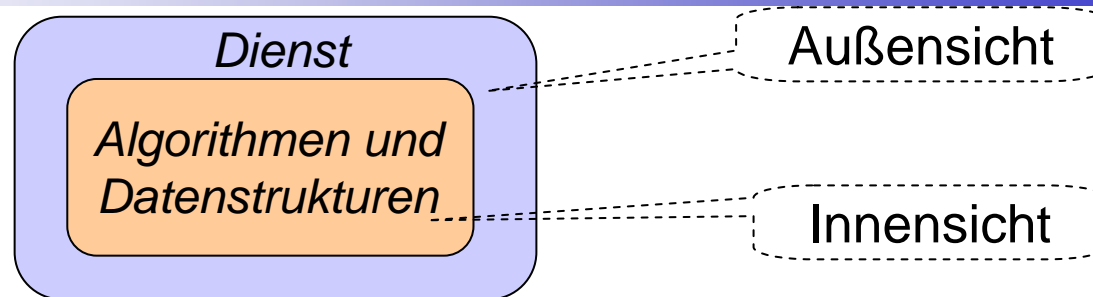


Ausführer:  
Rolladensteuerung

Verarbeiter u. Speicher:  
Pflegedienst

Wahrnehmer:  
Feuchtigkeit, Temperatur und Helligkeit

Ausführer:  
Wasserpumpe



## Dienst einer Komponente:

- Von der Komponente garantierte, von ihr erfüllte Aufgaben.
- Oder auch: Kompetenzen der Komponente.

## Diensteigenschaften:

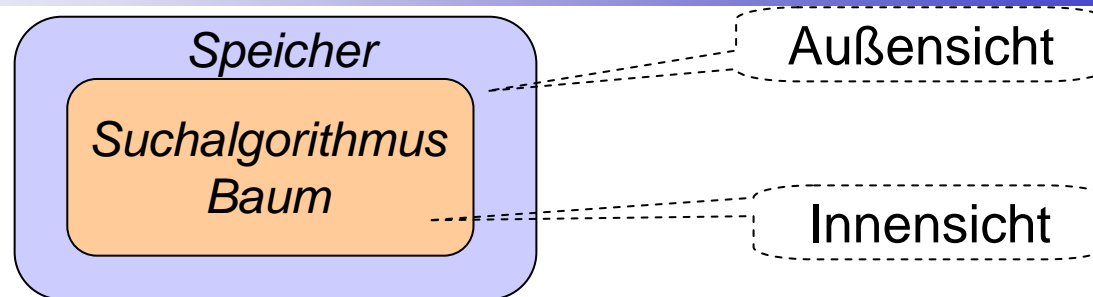
- Funktionale Eigenschaften (**Dienstfunktionalität**):
  - Umfang und Wirkung der anforderbaren Aufgaben.
- Nichtfunktionale Eigenschaften (**Dienstmerkmale** oder **Dienstqualitäten**):
  - Randbedingungen, die bei der Erfüllung der Aufgaben beachtet werden.

## Dienstbeschreibung:

- Dienstfunktionalität: Schnittstelle der Komponente.
- Dienstmerkmale: Begleitende Abmachungen.

## Rollen

- Dienstgeber
- Dienstnehmer



## Dienst einer Komponente:

- Von der Komponente garantierte, von ihr erfüllte Aufgaben.
- Oder auch: Kompetenzen der Komponente.

## Diensteigenschaften:

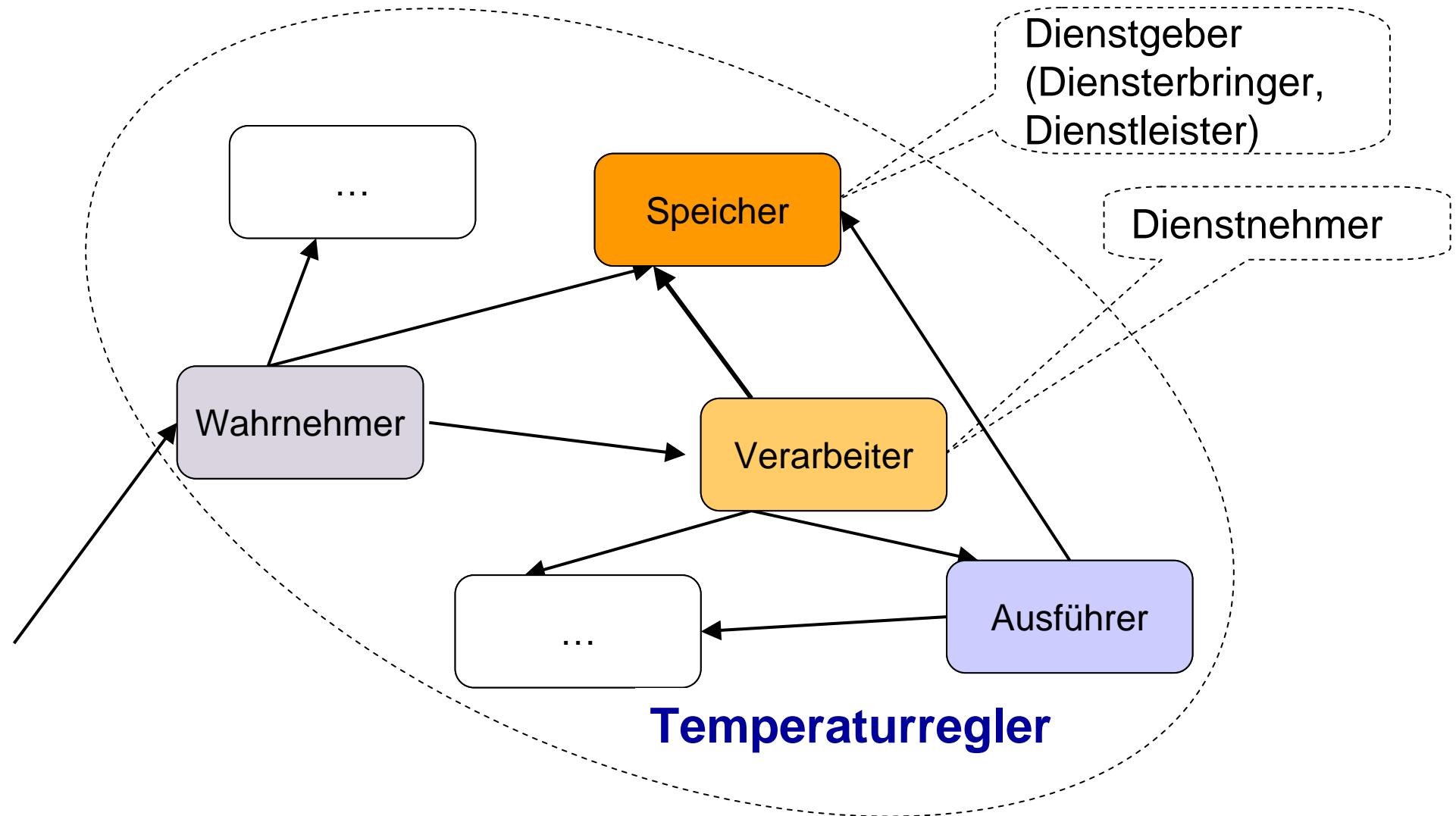
- Funktionale Eigenschaften (**Dienstfunktionalität**):
  - Umfang und Wirkung der anforderbaren Aufgaben.
- Nichtfunktionale Eigenschaften (**Dienstmerkmale** oder **Dienstqualitäten**):
  - Randbedingungen, die bei der Erfüllung der Aufgaben beachtet werden.

## Dienstbeschreibung:

- Dienstfunktionalität: Schnittstelle der Komponente.
- Dienstmerkmale: Begleitende Abmachungen.

## Rollen

- Dienstgeber
- Dienstnehmer



## Beispiel für die Beschreibung der Dienstfunktionalität:

- Abstrakter Datentyp (ADT)

## Beispiel: Keller (engl.: Stack)

- Datentyp mit Konstruktoren

```
data Stack t = CreateStack | Push (Stack t) t
```

- Signatur der anderen Operationen

```
pop      :: Stack t → Stack t
```

```
top      :: Stack t → t
```

```
empty    :: Stack t → Bool
```

- Axiome

```
pop (Push k x) = k
```

```
top (Push k x) = x
```

```
empty (CreateStack) = True
```

```
empty (Push k x) = False
```

**Wirkung** durch Axiome  
(ohne Ausnahmefälle)



## Zuverlässigkeit:

- Korrektheit:
  - Der Dienst, d.h. die Funktionen, werden vereinbarungsgemäß erbracht.
- Robustheit:
  - Bei inneren oder äußeren Störungen oder Fehlverhalten wird ein Zustand erreicht, von dem aus der Betrieb in wohldefinierter Weise fortgesetzt werden kann.

## Aufwand:

- Der mittlere oder maximale Zeit- oder Speicheraufwand überschreitet nicht die vorgegebenen Grenzen.

## Skalierbarkeit:

- Das Wachstum an Dienstgebern, Aufträgen oder Zustandsinformationen bleibt beherrschbar, wirkt sich also z.B. unterproportional beim Aufwand aus.

## Dauerhaftigkeit:

- Zustandsinformationen bleiben auf unbegrenzte Zeit erhalten.

## Portierbarkeit:

- Die Komponente ist auf verschiedenen Plattformen ablauffähig.

### Objekt:

- Autonome, gekapselte Einheit eines bestimmten Typs
- Eigener Zustand, der von außen nicht eingesehen werden kann (lokale Variable)
- Hat ein Verhalten
- Bietet anderen Objekten Dienstleistungen an

→ Dienstgeber

### Anforderung an die Dienstfunktionalität:

- Muss sämtliche Funktionen umfassen, die diesen Zustand einsehen oder manipulieren sollen.

Objekte sind Einheiten gekapselter Daten (genauer Programmvariablen), auf die durch Methoden zugegriffen wird.



M. Broy; Informatik – Band 1, 1998



Im Zusammenhang mit Objekten nennt man eine Funktion einen Operator oder (häufiger) eine Methode.

Eine Methode deckt sich damit mit der früheren Definition als einem Programmstück, bei dem nur die Wirkung nach außen, nicht aber das Programm selbst interessiert.

Eine Methode tritt somit nach außen nur durch seine (Methoden) Signatur in Erscheinung, seine (Methoden)-Implementierung (früher: Methodenrumpf) bleibt gekapselt.

## Dienstleistung:

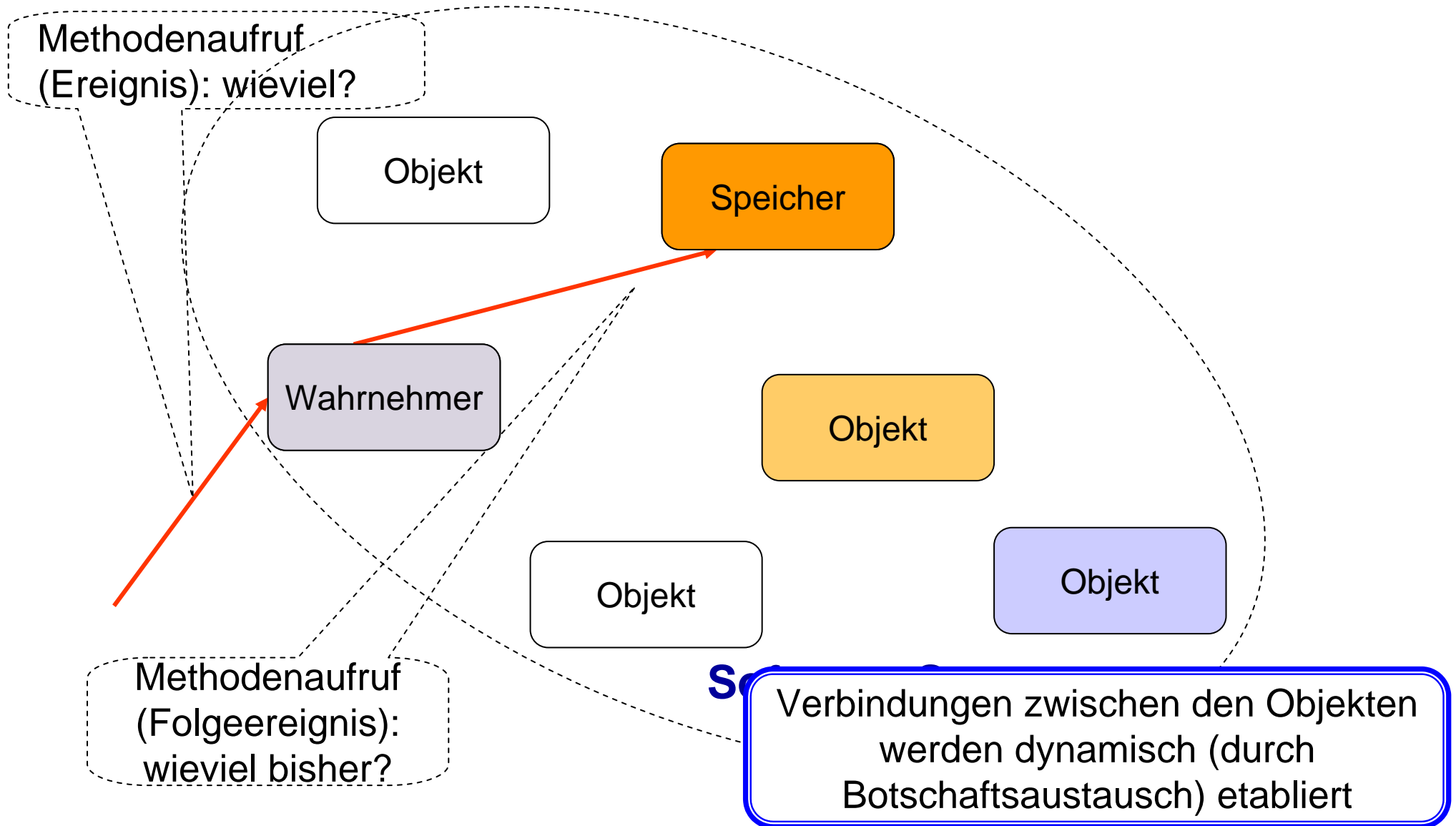
- Um das Erfüllen einer Aufgabe (eine Dienstleistung) anzufordern, muss eine Methode aufgerufen werden.
  - Die übliche Sichtweise ist, dass dazu an das Objekt eine **Botschaft** (Nachricht) gesendet wird, die die Methode benennt und zusätzliche Daten (Methodenparameter) enthält.
  - Aus der Sicht des empfangenden Objektes wird dort ein Ereignis ausgelöst.

## Objekt

- ... ein Objekt definiert als Informationsträger, der einen (zeitlich veränderbaren) Zustand besitzt und für den definiert ist, wie er auf bestimmte „Nachrichten“ (eingehende Mitteilungen an ein Objekt) zu reagieren hat; ...



Duden für Informatik, 1993



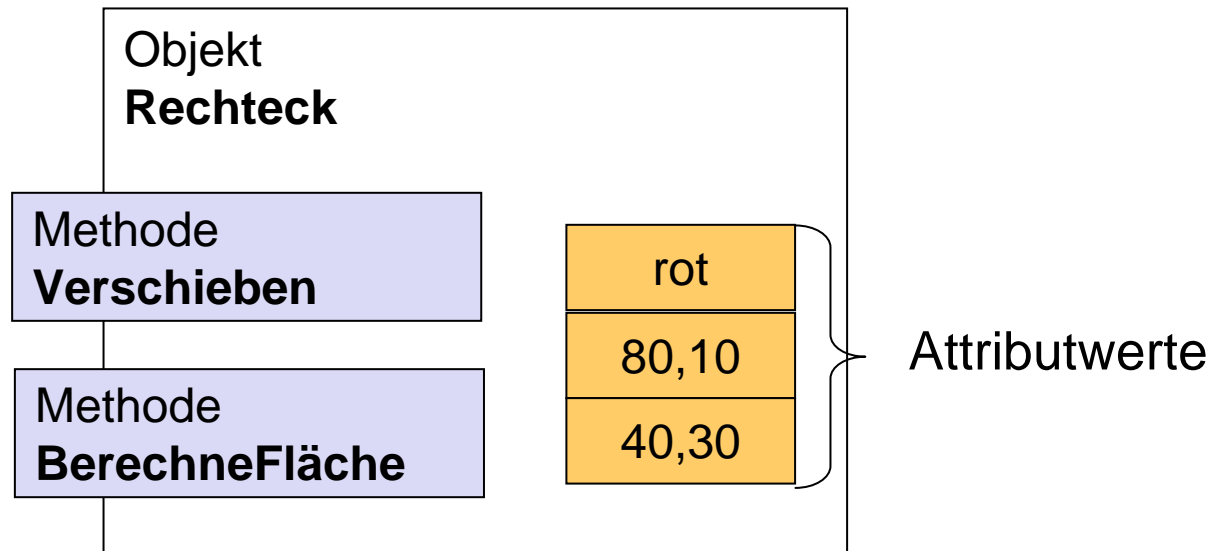
## Identität

- Eigenschaft, durch die sich das Objekt von anderen unterscheidet

## Objekt vs. Wert

- Identität ist unabhängig vom aktuellen Wert der Attribute, die den aktuellen Zustand repräsentieren.
  - Oft wird in Programmiersprachen die Adresse im Speicher als Identität verwendet.
  - Eindeutigkeit ist dadurch gewährleistet.

## Beispiel



- Botschaft: „Verschiebe um 10 mm nach rechts und um 20 mm nach oben“

## Objektorientierung:

- Vereinbarung einer Menge von Objekten mit ihren Diensten.

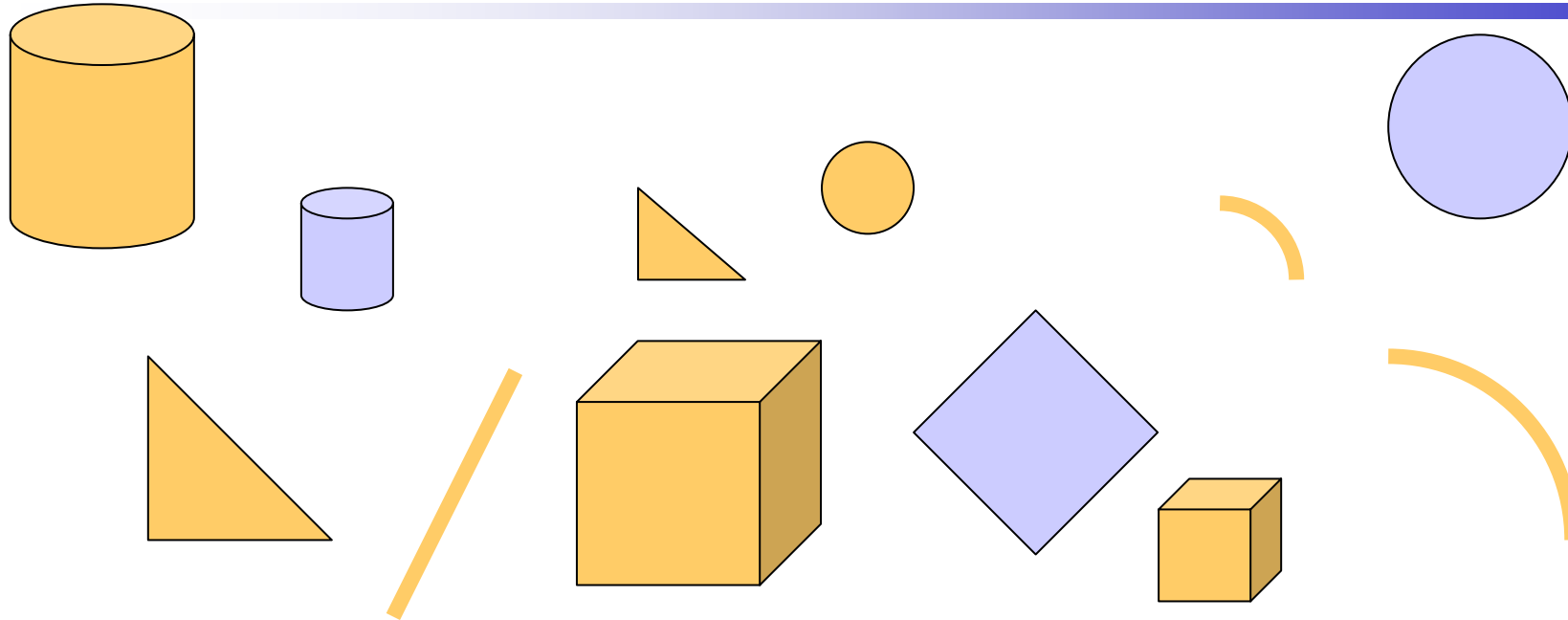
## Erbringen einer Dienstleistung:

- Aufrufen einer Methode eines passenden Objektes.
- Dieses Objekt kann sich der Dienste weiterer Objekte versichern und wird dazu Folgeereignisse auslösen.

Frage: Woher erfährt ein Objekt, welche Dienste es von welchen Objekten beziehen kann?

Antwort: Durch einen Verzeichnisdienst.

- Objekt kann Verzeichnis der interessierenden Dienstgeber als Teil seines Zustandes führen: Verweistechnik.  
⇒ häufig bei zentralen Lösungen.
- Objekt muss sich zuvor passende Dienstgeber benennen lassen: Objekt wendet sich dazu mit Aufgabenbeschreibung an globalen Verzeichnisdienst.  
⇒ üblich bei verteilten Lösungen.



Zu welchen Klassen gehören diese Objekte?

- Form:                      rund, eckig
- Farbe:                    blau, grün
- Größe:                   groß, klein
- Dimension:            Strich, Fläche, Körper
- Erscheinung:          schön, häßlich

## Objektklasse:

- Im Allgemeinen benötigt man zahlreiche Objekte, die gleiches Verhalten zeigen, deren Methoden also (bei gleichem Zustand) gleiche Wirkung zeigen.
- Eine Menge derartiger Objekte lässt sich durch eine Zustandsabstraktion, die Methodensignaturen und die Wirkung der Methoden auf der Zustandsabstraktion beschreiben.

Eine solche Beschreibung definiert damit einen Objekttyp.

## Definition Klasse:

- Eine Klasse ist eine Menge von Objekten mit gleichen Eigenschaften (Attributen) und gleichem Verhalten (Methoden).



G. Saake, K. Sattler; Algorithmen und Datenstrukturen; dpunkt.verlag, 2002

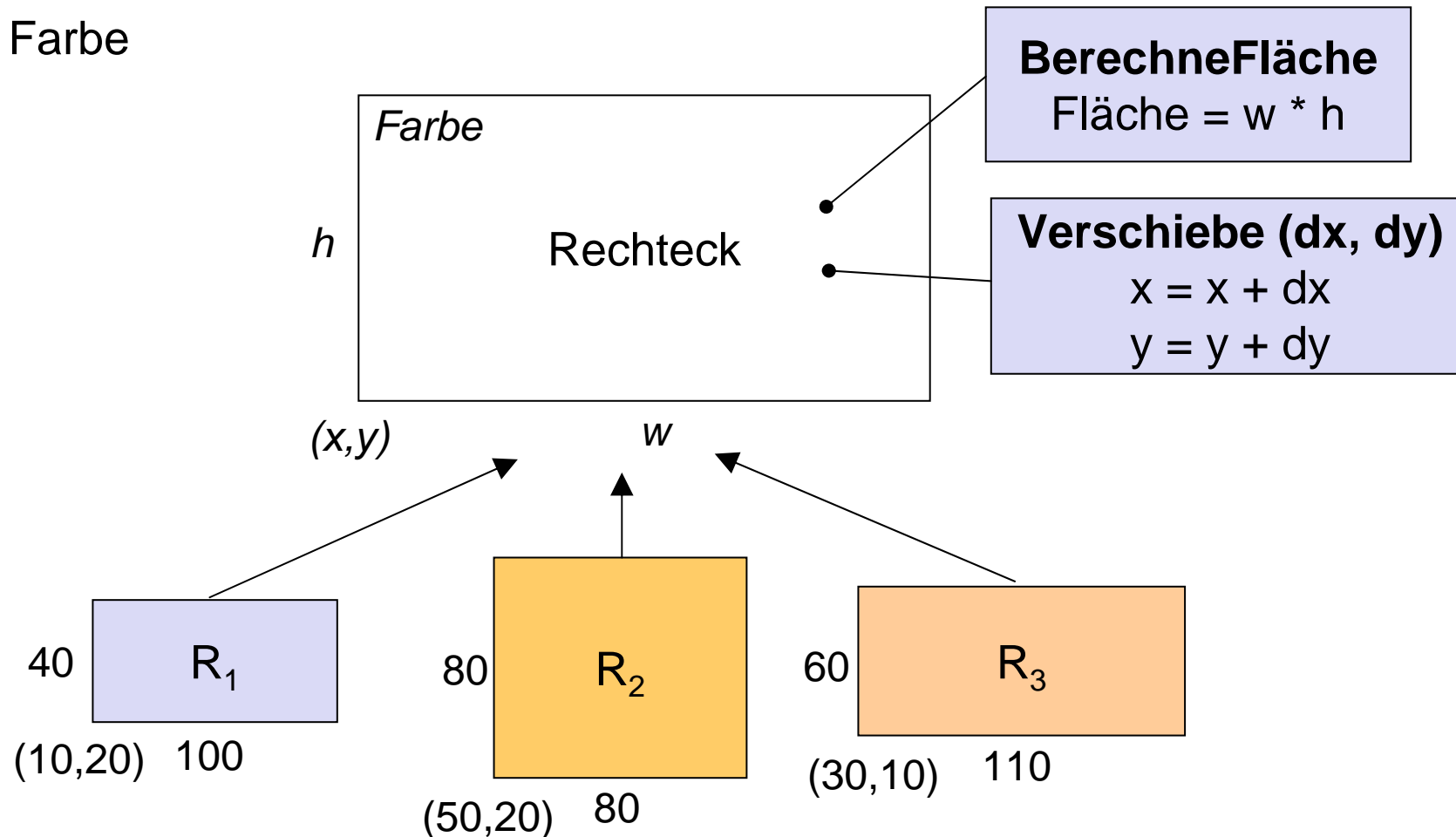
## Anmerkungen:

- In Informatik I wurde als Typklasse eine Gruppe von Typen mit gleichartigen, überladbaren Operatoren bezeichnet. Wir werden später sehen, dass dies auf Objekttypen zutreffen kann. Daher wird üblicherweise von Objektklassen statt von Objekttypen gesprochen.
- Somit sind Objekte Ausprägungen von Objektklassen.



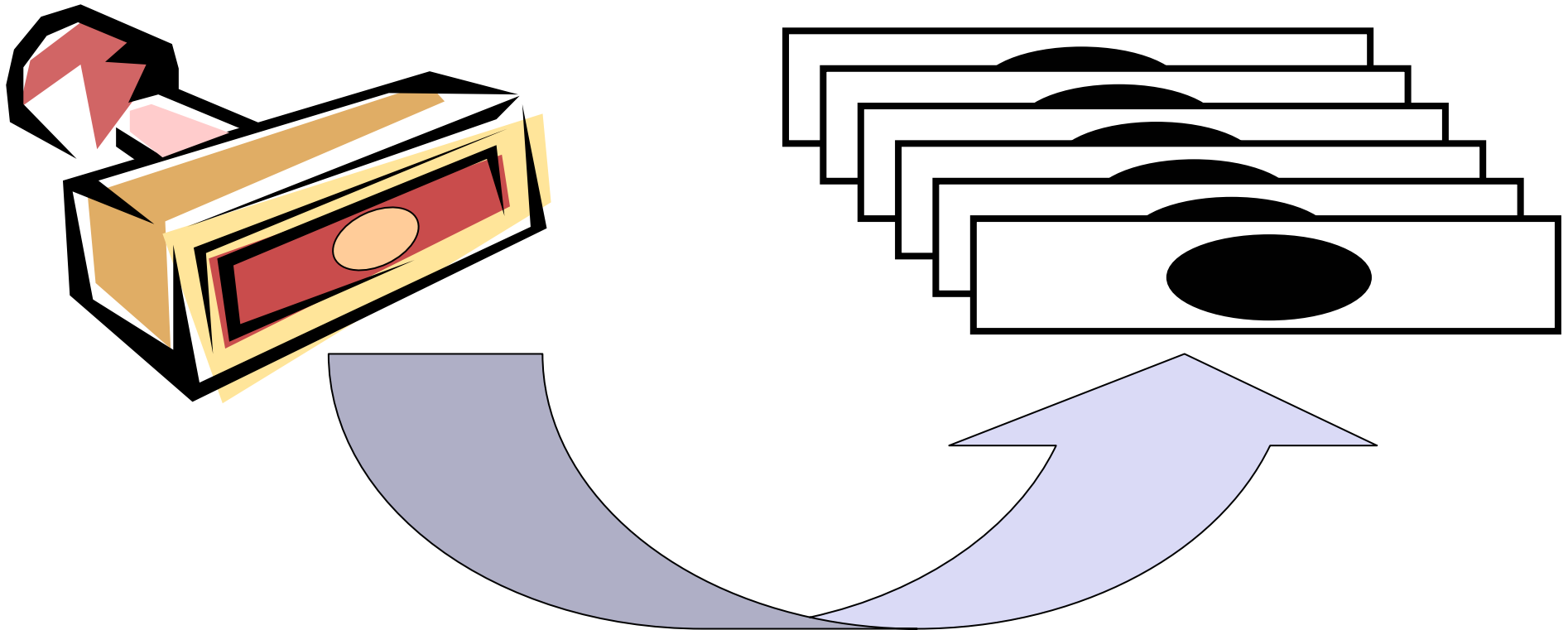
## Klasse *Rechteck*

- Position (x,y)
- Ausdehnung w und h
- Farbe



## Was sind Klassen ?

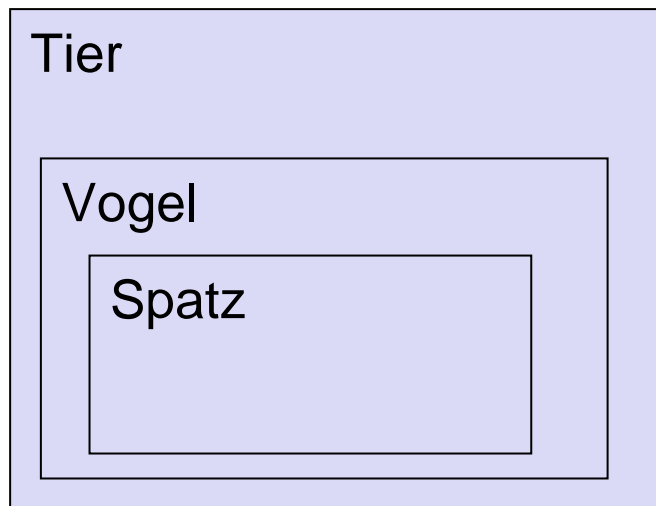
Klassen sind wie Stempel, die Abdrücke sind die Objekte



Die Abdrücke können verändert werden  $\Rightarrow$  Verschiedene Zustände von Objekten!

# Was sind Klassen ?

- Beschreiben, was Objekte gemeinsam haben.
- Definieren Objektstruktur (Attribute)
- Definieren Verhalten der Objekte (Methoden)
- Definieren Botschaften, auf die reagiert werden kann
- Können Verallgemeinerungen/Spezialisierungen anderer Klassen sein
- Haben in der Regel einen Namen (Identität)
- Kapseln klassenspezifische Interna von der Außenwelt ab



## Geheimnisprinzip

- Unmöglichkeit der Manipulation von „außen“
- Kenntnisse über Modul-/Klasseninterna nicht nötig

## Unabhängigkeit

- Programmierung ohne Rücksicht auf andere Module / Klassen
- Änderungen möglich ohne Effekt auf andere Module / Klassen

## Disziplinierung

- Weniger Spaghetticode
- Durchdachte Programme, leichtere Lesbarkeit

### Zur Erinnerung (Informatik I):

- Motivation von Typisierung:
  - Vermeidung von Fehlern, die prinzipiell vermieden werden könnten.

### Beispiel: Im Rechner sind alle Daten Bitfolgen.

- Alle Operationen haben Bitfolgen als Argumente.
- Die Operation entscheidet, wie die Bitfolgen interpretiert werden.
  - z.B. Gleitkommaaddition interpretiert Operanden als Gleitkommazahlen, Ganzzahladdition als ganze Zahlen.
- Fehlerquelle, wenn Programmierer irrtümlich falsche Operanden verwenden.

### Daher Typisierung in Haskell.

### Daher auch Typisierung in Java!

- Bei der imperativen Programmierung muss auch der Zustand typisiert werden.



## Zur Erinnerung (Informatik I):

- Typisierung:
  - weist jedem Ausdruck, insbesondere jeder Variable, einen Typ zu.
- Stark typisierte Sprachen erkennen Typfehler immer zur Übersetzungszeit - also vor Ausführung der Programme.
- Schwach typisierte Sprachen erkennen Typfehler mindestens bei der Ausführung von Programmen.

Java!

## Typfreie Sprachen erkennen Typfehler auch nicht zur Laufzeit.

- Programme können falsche Ergebnisse produzieren, ohne dass dies bemerkt wird!

Zur Erinnerung (Informatik I):

## Vorteile stark typisierter Sprachen:

- Die Bedeutung aller Operationen und Funktionen eines Programms sind vor Ausführung des Programms bekannt.
  - Erleichtert Verifikation von Programmen.
  - Übersetzer können besseren Code erzeugen.
  - Fehlerhafter Gebrauch von Operationen wird vor Ausführung des Programms erkannt.
  - Unterstützt Fehlersuche
    - insbesondere, wenn verschiedene Programmierer ihre Teile zu einem Programm zusammensetzen.

Diese Vorteile sollen auch für die  
Objektorientierung genutzt werden  
⇒ Weitere Motivation für Objekttypen!



Objektklasse



Zustandsabstraktion und Methodenrümpfe nicht sichtbar. Es kann also allein Typisierung der Methodensignaturen betrachtet werden.

- Für die Typisierung interessiert auch nicht die abstrakte Wirkung der Methoden.

Abstrakte Datentypen (ADT) sind ideale Beschreibungen der Dienstfunktionalität von Objektklassen

## Beispiel: Keller

Für  $t$  ist ein konkreter Typ vorzugeben!

```
data Stack t = CreateStack | Push (Stack  
pop      :: Stack t → Stack t  
top      :: Stack t → t  
empty    :: Stack t → Bool
```

**Signatur**

```
pop (Push k x)      = k  
top (Push k x)      = x  
empty (CreateStack) = True  
empty (Push k x)    = False
```

**Wirkung** durch Axiome  
(ohne Ausnahmefälle)

Abstrakte Datentypen (ADT) sind ideale Beschreibungen der Dienstfunktionalität von Objektklassen

## Beispiel: Schlange

```
data Queue t = CreateQueue | Enqueue (Queue t) t
dequeue      :: Queue t → Queue t
front        :: Queue t → t
empty        :: Queue t → Bool
```

**Signatur**

```
dequeue (Enqueue CreateQueue a) = CreateQueue
dequeue (Enqueue s a)           = Enqueue (dequeue s) a
front (Enqueue CreateQueue a)   = a
front (Enqueue s a)             = front s
empty CreateQueue                = True
empty (Enqueue s a)             = False
```

**Wirkung** durch Axiome  
(ohne Ausnahmefälle)

Abstrakte Datentypen (ADT) sind ideale Beschreibungen der Dienstfunktionalität von Objektklassen

## Beispiel: Menge

- (Beachte Schreibweise bei mehreren Parametern.)

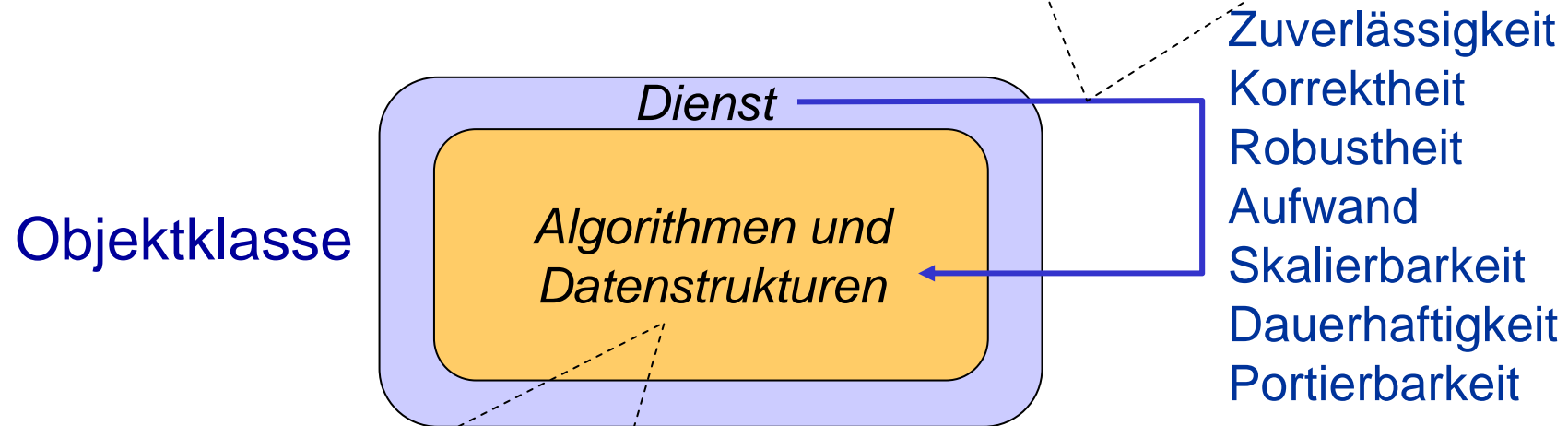
```
data Set t = EmptySet | Elem t (Set t)
single      :: t → Set t
insert      :: Set t → t → Set t
is_elem     :: t → Set t → Bool
delete      :: Set t → t → Set t
union       :: Set t → Set t → Set t
intersect   :: Set t → Set t → Set t
difference  :: Set t → Set t → Set t
empty       :: Set t → Bool
```

**Signatur**



## 2.5 Algorithmenwahl

Das Gewicht, das man den Dienstqualitäten zuordnet, entscheidet darüber, wie die Implementierung der Methoden und damit eine günstige Zustandsabstraktion aussieht.



Zustandsabstraktion durch Vereinbarung geeigneter Typen für die Datenstrukturen. Darauf aufbauend Typisierung der aus den Algorithmen umgesetzten Programme.

## Algorithmus nach Aho, Hopcroft, Ullman:

- Endliche Folge von Instruktionen, endliche Zeit pro Instruktion, eindeutig interpretierbar, enthalten Wiederholungsinstruktionen, terminiert

Algorithmenwahl!

Algorithmus  $\neq$  Problem (1 Problem  $\Rightarrow$  viele Algorithmen)

Algorithmus  $\neq$  Programm (1 Algorithmus  $\Rightarrow$  viele Programme)

- Churchsche These:

„Für jeden Algorithmus, der intuitiv berechenbar ist, gibt es in jeder Programmiersprache (mit ausreichender Mächtigkeit) ein entsprechendes Programm.“

- Für jedes „Programm“ (egal ob in  $\lambda$ -Notation, Java, Haskell) gibt es in jeder anderen Notation eines, das dasselbe tut.

## Alfred V. Aho



- Vizepräsident der Bell Labs
- Computer Science Department at Columbia University.
- Dr. an der Princeton University
- Dipl. an der University of Toronto (Physik)
- IEEE John von Neumann Medal
- Aho ist das "A" in AWK

## John E. Hopcroft



- College of Engineering an der Cornell University
- Dr. an der Stanford University
- 1986 Turing Preis



[www.wikipedia.de](http://www.wikipedia.de)



## Jeffrey D. Ullmann



- Stanford University (Emeritus)
- Dr. an Princeton University
- Dipl. an Columbia University



[www.wikipedia.de](http://www.wikipedia.de)

„I have been writing a few articles about things I think especially stupid [...]: Fundamentalism [...], 4-Way Stop Signs [...], Software Patents [...], University of California (Non)Confidentiality Policy [...]“



**Problem:** Berechne  $s$ , die Summe der ersten  $n$  natürlichen Zahlen

## Algorithmus 1:

```
setze s=0; setze i=1;  
solange i<=n: {setze s=s+i und i=i+1 }
```

## Algorithmus 2:

```
setze s = (n+1)*n/2
```

## Programm 1a:

```
int s=0;  
for (int i=1; i<=n; i++)  
s+=i;
```

## Programm 1b:

```
int i=1; s=0;  
while (i<=n) s+=i++;
```