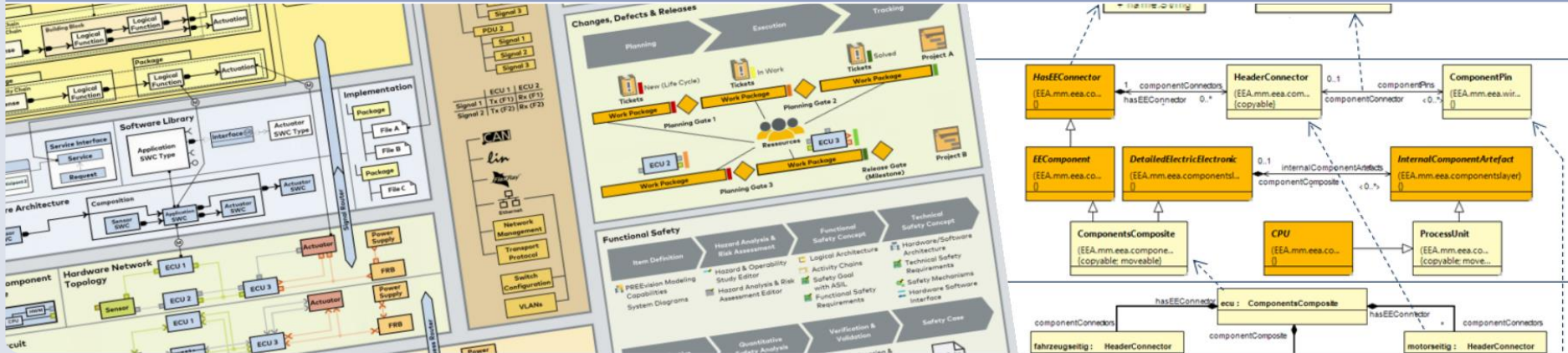


Vorlesung Software Engineering (SE)

Wintersemester 2017/2018

Kapitel 5 – Softwareentwurf



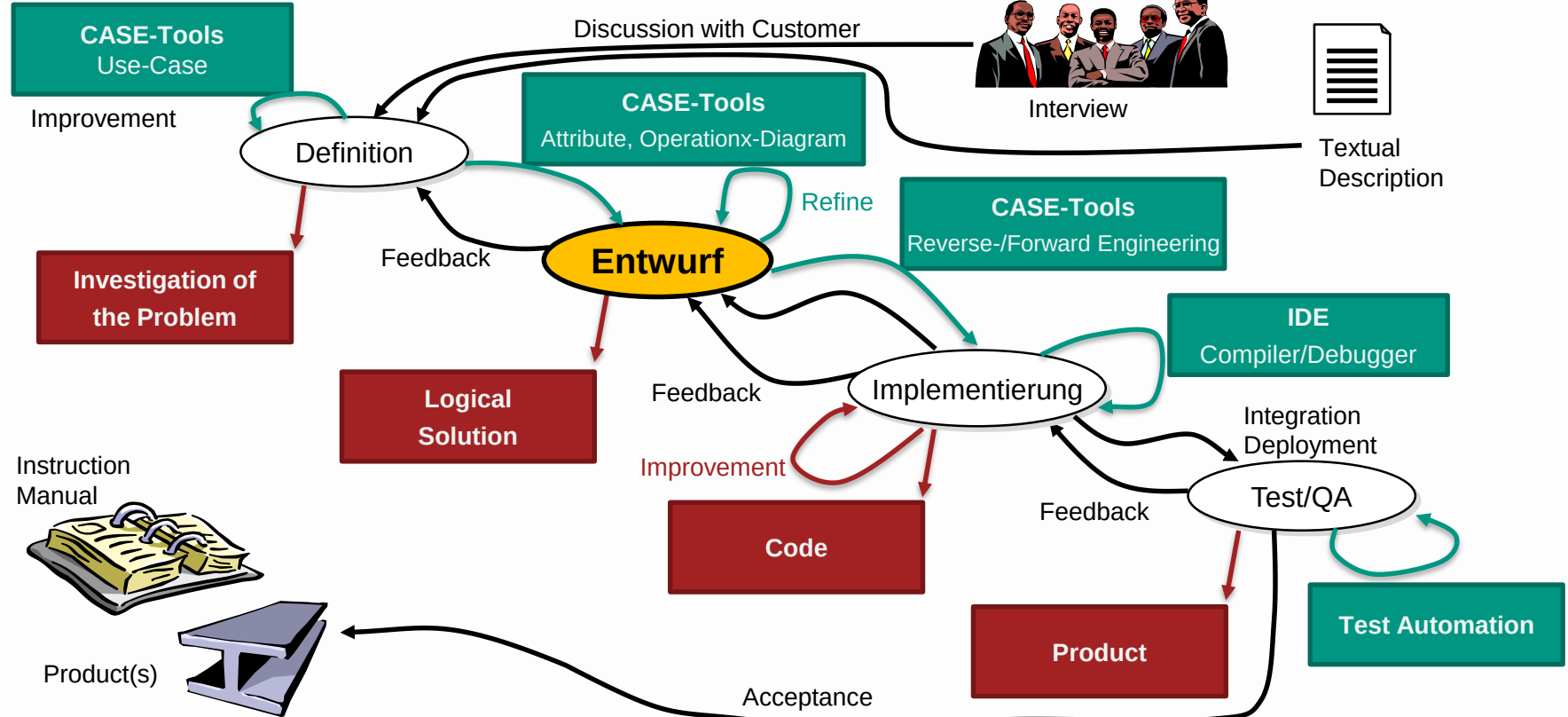
5. Softwareentwurf

Software Engineering



Inhalt – Softwareentwurf

- 5.1 Einführung und Begriffe
- 5.2 Strukturierungsformen
- 5.3 Evolution der Entwurfskonzepte/-methoden
- 5.4 Modularer Entwurf (MD)
- 5.5 Objektorientierter Entwurf (OOD) / Objektorientierte Programmierung (OOP)



Aufgabe des Entwerfens

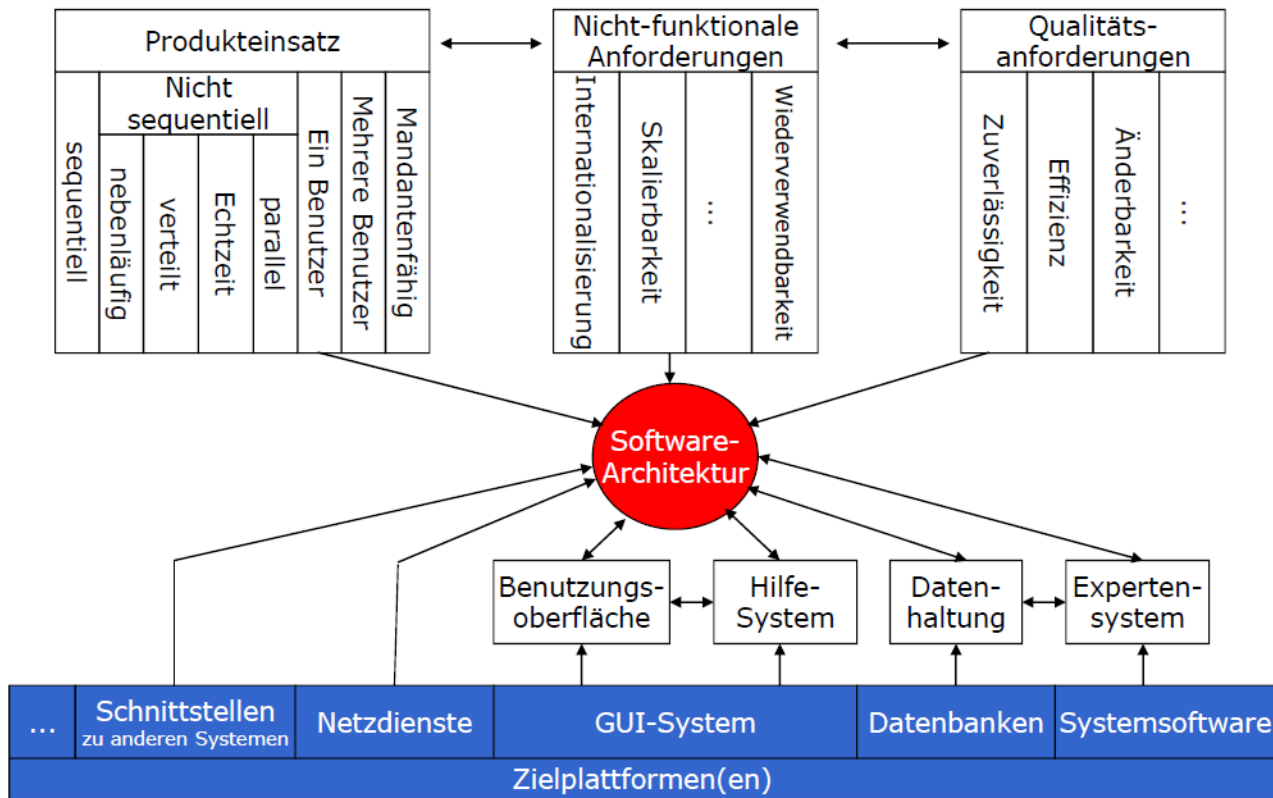
- Aus den gegebenen Anforderungen an ein Software-Produkt eine **software-technische Lösung** im Sinne einer Softwarearchitektur entwickeln
- **Softwarearchitektur**
 - Gliederung eines Softwaresystems in **Subsysteme und Module** (Aufstellung der Bestandshierarchie)
 - Beschreibung der **Funktion** und **Schnittstellen** der Komponenten
 - Beschreibung der **Beziehungen** zwischen diesen Komponenten (Benutzt-Relation)
- Entwerfen wird auch »**Programmieren im Großen**« genannt
- **Ausgangspunkt**: Ergebnisse der *Definitionsphase*

Vor den Entwurfsaktivitäten **Einflussfaktoren** klären:

- Einsatzbedingungen
- Umgebung/Randbedingungen
- Nichtfunktionale Produkt- und Qualitätsanforderungen

Nach dem „**Was bauen wir?**“ der Definitionsphase folgt nun das „**Wie strukturieren wir es?**“ unter Berücksichtigung der Randbedingungen.

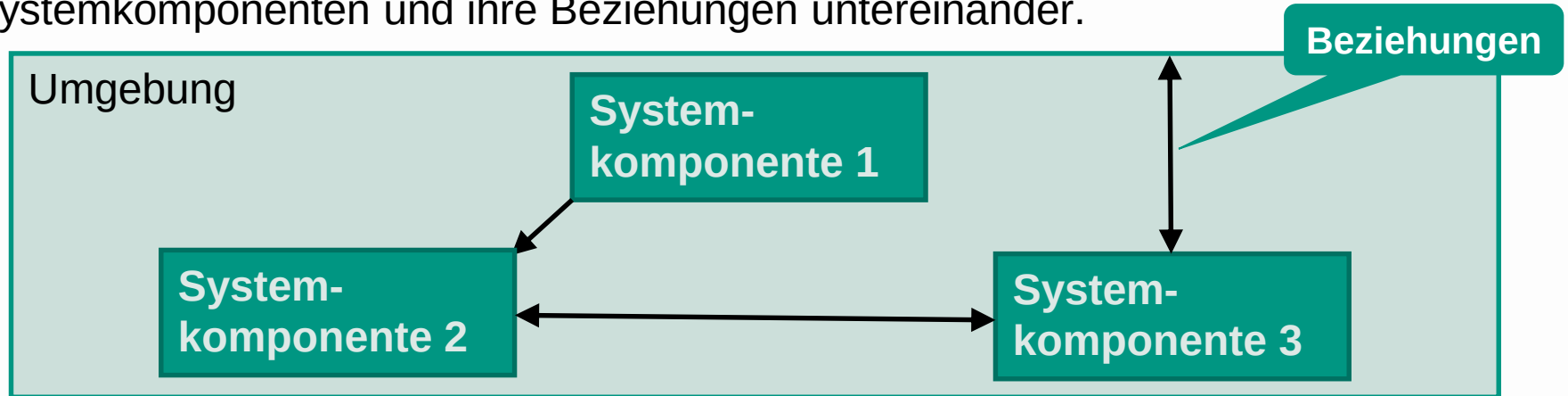




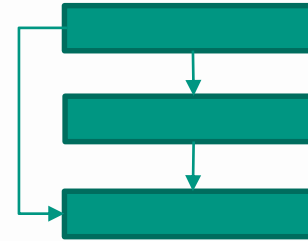
Quelle: Prof. K.-P. Fährnich (nach Balzert)

Ziel des Softwareentwurfs ist es, für das zu entwerfende Produkt eine **Software-Architektur** zu erstellen, die **funktionale** und **nicht funktionale** *Produktanforderungen* sowie **allgemeine** und **produktspezifische** *Qualitätsanforderungen* erfüllt und die *Schnittstellen* zur Umgebung versorgt.

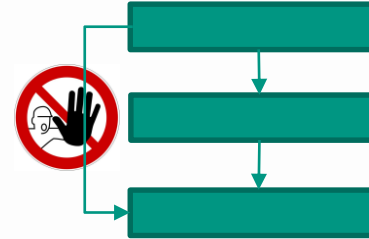
Eine *Software-Architektur* beschreibt die Struktur des Softwaresystems durch Systemkomponenten und ihre Beziehungen untereinander.



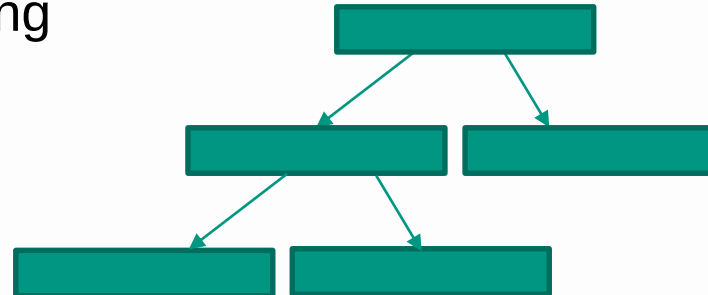
Schichten mit **linearer** Ordnung



Schichten mit **striker** Ordnung



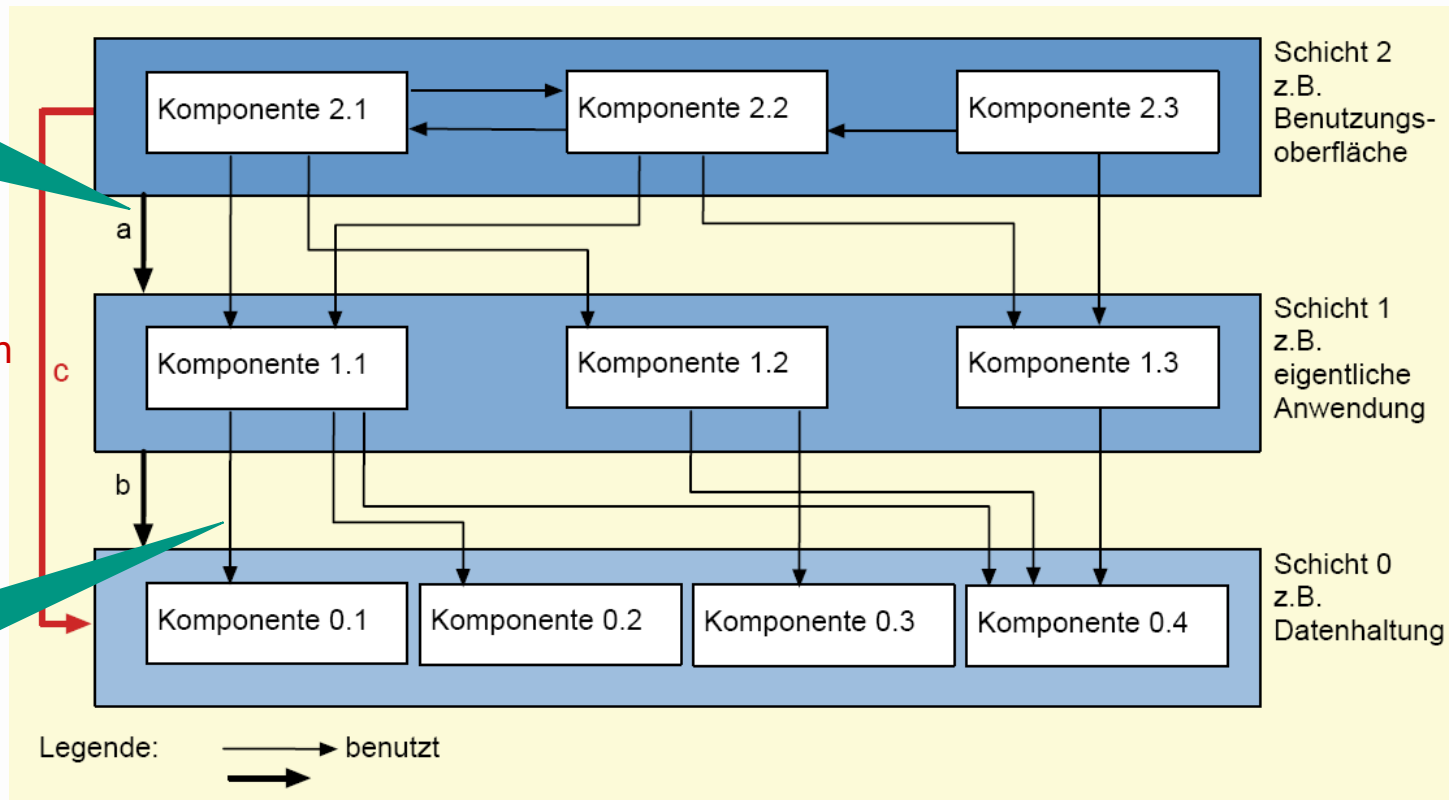
Schichten mit **baumartiger** Ordnung



a: Schicht 2
benutzt
Schicht 1

Der Benutzt-Pfeil
c ist beim linearen
Schichtenmodell
nicht zugelassen

c: Komponente
1.1 benutzt
Komponente 0.1



Anwendungsbereich

- Wenn die **Dienstleistungen** einer Schicht sich auf **demselben Abstraktionsniveau** befinden *und*
- wenn die Schichten entsprechend ihrem Abstraktionsniveau **geordnet** sind, so dass eine Schicht **nur die Dienstleistungen der tieferen Schichten** benötigt

Voraussetzung

Es muss ein **natürliches Abstraktionskriterium** geben

- Eine Benutzt-Beziehung allein reicht nicht aus!
- Es muss eine geeignete **Granularität** für die Schichten gefunden werden.

Pros

- ✓ **Übersichtliche Strukturierung** in Abstraktionsebenen oder virtuelle Maschinen
- ✓ **Keine zu starke Einschränkung** des Entwerfers, da er neben einer strengen Hierarchie noch eine liberale Strukturierungsmöglichkeit innerhalb der Schichten besitzt
- ✓ Es werden die **Wiederverwendbarkeit**, die **Änderbarkeit**, die **Wartbarkeit**, die **Portabilität** und die **Testbarkeit** unterstützt.
(Schichten können ausgetauscht, hinzugefügt, verbessert und wiederverwendet werden; Testen von unten oder von oben her).

Cons

- × **Effizienzverlust**, da alle Daten über verschiedene Schichten transportiert werden müssen (bei linearer Ordnung). Dies gilt auch für Fehlermeldungen.
- × Eindeutig voneinander **abgrenzbare Abstraktionsschichten** lassen sich **nicht immer definieren**.
- × Innerhalb einer Schicht kann **Chaos** herrschen.

Evolution der Entwurfskonzepte-/methoden

→ Diagramme siehe Vorlesung: SSE

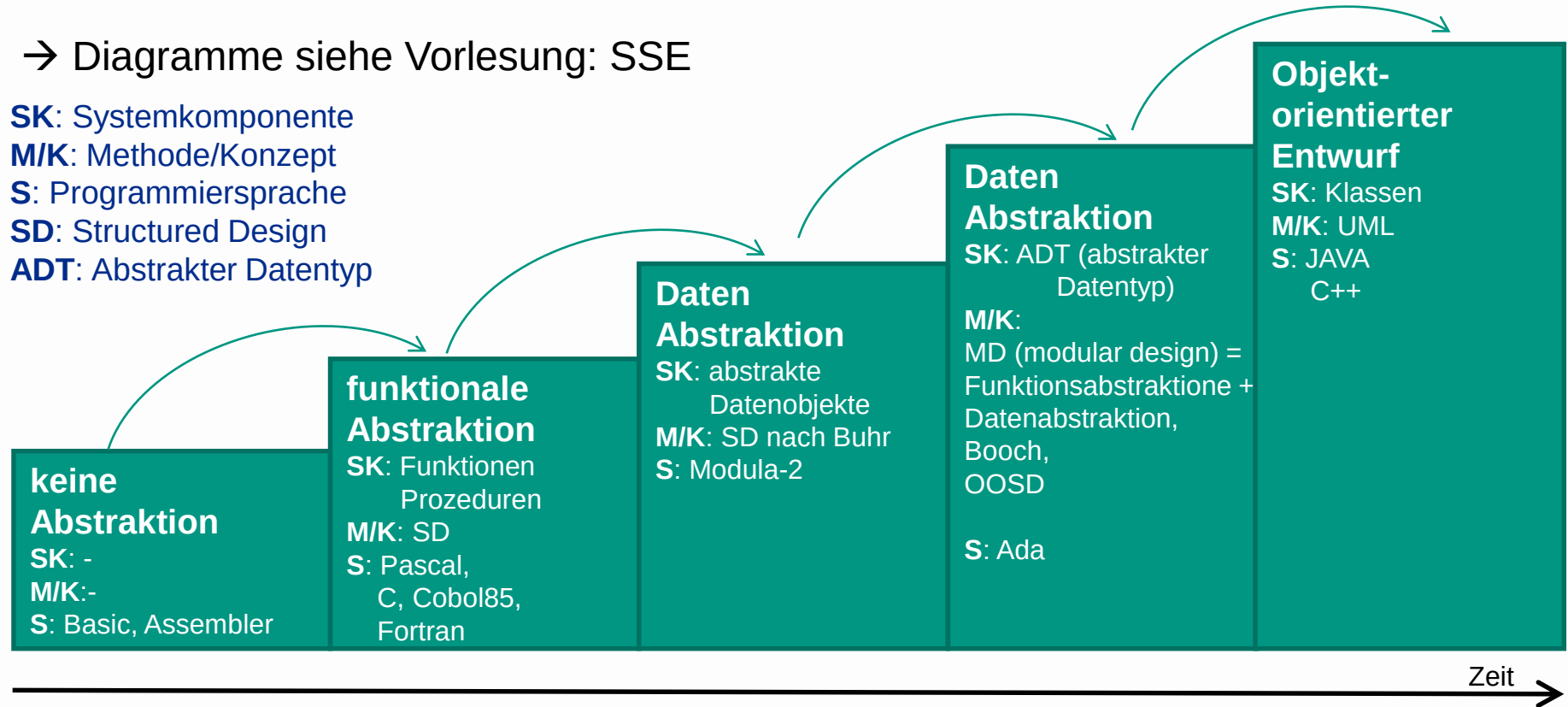
SK: Systemkomponente

M/K: Methode/Konzept

S: Programmiersprache

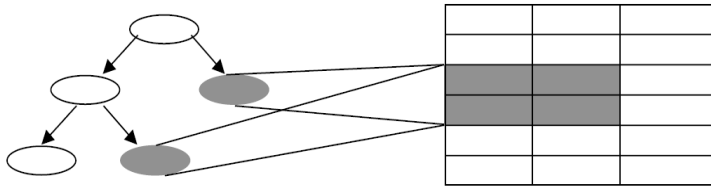
SD: Structured Design

ADT: Abstrakter Datentyp



Strukturierter Ansatz (SD)

Separation von Funktionen und Daten

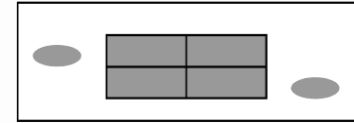


Funktionshierarchie

Externe Datenbasis oder interner Variablenkeller

Objektorientierter Ansatz (OO)

Integration von Funktionen und Daten



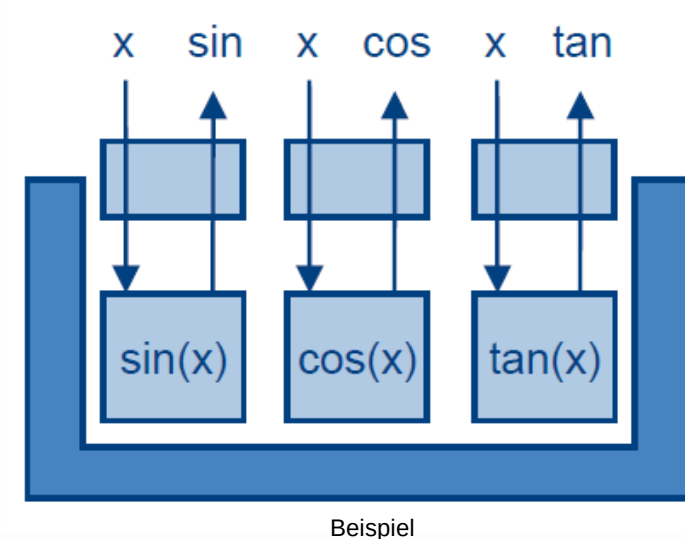
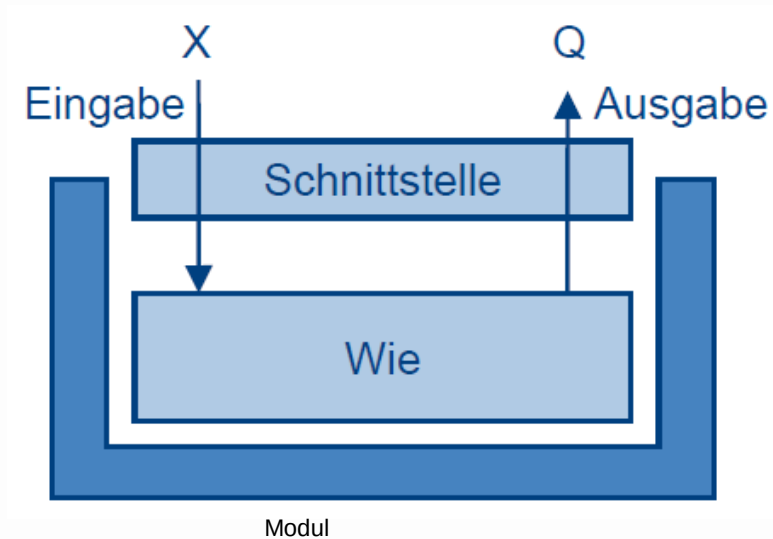
Objekt = (kleine) Dateneinheit +
lokale Funktionen

Vergleich bezüglich „Funktionen und Daten“

Modularer Ansatz (MD)

dazwischen

Modul = (große) Funktionsgruppe + lokale Daten



→ **Systemarchitektur** beim modularen Entwurf:

1. Modulführer

(Grobentwurf, *module guide*, *software architecture*):

- Beschreibung der Funktion jedes Moduls und der **Gliederung in Subsysteme**;
- benutzt **Entwurfsmuster**, z.B. Schichtenarchitektur oder Fließbandarchitektur.

2. Modulschnittstellen (*module interfaces*):

- **Genaue Beschreibung** der von jedem Modul zur Verfügung gestellten Komponenten (Typen, Variablen, Unterprogramme etc.), informell oder formal.
- für Module mit **Ein-/Ausgabe** auch genaue Beschreibung der entsprechenden **Formate**.

3. Benutzt-Relation (*Uses relation*):

- Beschreibt, *wie* sich Module und Subsysteme *untereinander* benutzen (azyklischer, gerichteter Graph).

4. Feinentwurf (*detailed design*):

- Beschreibung der *modul-internen* Datenstrukturen und *Algorithmen*
- bei Implementierung in Assembler auch *vollständige Programmierung* der Module in Pseudocode.
- Der *Pseudocode* ist in einer höheren, meist hypothetischen Programmiersprache abgefasst und wird in der Implementierungsphase von Hand *in Assembler* umgesetzt.

Modul / Geheimnisprinzip



Modul (abstraktes Datenobjekt, *Module*):

Ein *Modul* ist eine **Menge von Programmkomponenten**, die nach dem *Geheimnisprinzip* gemeinsam entworfen und geändert werden.

Programmkomponenten sind Typen, Klassen, Konstanten, Variablen, Datenstrukturen, Prozeduren, Funktionen, Prozesse (Fäden), Prozess-Eingänge, Makros etc.



Geheimnisprinzip (*Information Hiding*):

Jedes Modul verbirgt eine wichtige **Entwurfsentscheidung** hinter einer wohldefinierten **Schnittstelle**, die sich bei einer Änderung der Entscheidung nicht mitändert.

— **David Parnas**

Grund

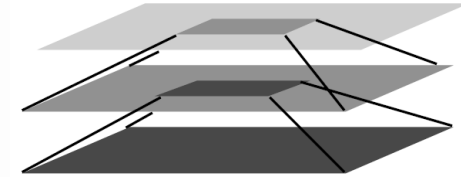
Nur was verborgen und unbenutzt ist, kann **ohne Risiko geändert** werden.

Ziel: Komplexität beherrschbar machen durch...

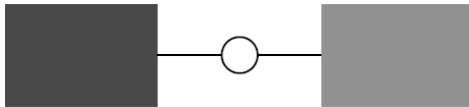
Zerlegung in Teilsysteme
(Verantwortungsbereiche)



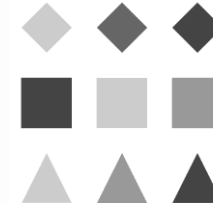
Verschiedene Abstraktionsebenen
(Spezialisierungshierarchie)



Klare Schnittstellen zwischen
Teilsystemen (Protokolle)



Vorgefertigte Teile –
Wiederverwendung (Baukastenprinzip)



Prinzipielle Vorteile von Objektorientierung

Lokale Kombination von Daten und Operationen,
gekapselter Zustand

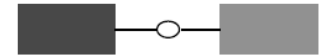
Definiertes **Protokoll**, Nachrichten zwischen Objekten

Vererbung und **Polymorphie** (Spezialisierung,
Generalisierung)

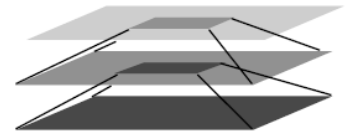
Benutzung vorgefertigter **Klassenbibliotheken**,
Anpassung durch Vererbung mit **Ergänzung** und
Redefinition



Zuständigkeitsbereiche



Klare Schnittstellen



Hierarchie



Baukastenprinzip

Objektorientierter Entwurf (OOD) – Programmierung

Im *objektorientierten Entwurf* behalten die Prinzipien des modularen Entwurfs (Flexibilisierung der Software durch das **Geheimnisprinzip**) ihre Gültigkeit.

Schnittstellen verbergen Entwurfsentscheidungen, die veränderbar bleiben sollen.

Die Analoga zum **Modul** sind die **Klasse** und das **Paket**:

- Im Paket werden mehrere Klassen, die **gemeinsame Entwurfsentscheidungen** kapseln, zusammengefasst
- Eine allein stehende Klasse kann auch ein ganzes Modul verwirklichen; i.d.R braucht man aber **mehrere Klassen pro Modul**

Allerdings ergeben sich im OO-Entwurf zusätzliche Möglichkeiten, die im modularen Entwurf schwieriger darzustellen sind:

- ✓ **Mehrfach-Instanziierung** von Klassen,
- ✓ **Vererbung** und **Polymorphie**,
- ✓ **Variantenbildung** in einem Programm durch Mehrfachimplementierung einer Schnittstelle.

Klasse

Objekt

Vererbung

Spezialisierung

Methode

Attribut

Zustand eines Objekts

Overriding

Overloading

Polymorphismus

Binding

→ Begriffe werden später erklärt ...

Paradigmen (I)



Modulares / Prozedurales Programmieren

Allgemein: Software schrittweise *verfeinern*



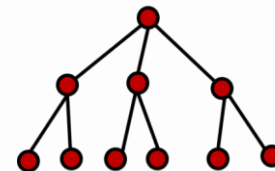
Top Down

Gesamtprogramm im Mittelpunkt

- Funktionen werden implementiert
- Funktionen werden verfeinert

Problem

- Untere Stufen *sequentiell* zu erarbeiten, sonst Applikation nicht lauffähig
- Benötigte Funktionen *müssen* bekannt sein



Paradigmen (II)



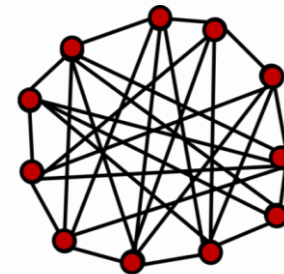
Bottom Up

Modulare Sichtweise

- Komponenten auf unteren Stufen *erarbeiten*
- Nach oben *abstrahieren* (Module)

Problem

- Eigenständige *Lauffähigkeit* der unteren Stufen



Hybridmodelle

Mischung aus Top Down / Bottom Up

Paradigmen (III)

Frage nach effizienter Entwicklung von *leistungsfähiger* und *wartbarer* Software

Motivation zum Übergang zu OO

- **Daten** stehen im Vordergrund
- **Funktionen sind kurzlebiger** als Daten
- Software ist **Änderungen** unterworfen

Folgen

- In **Objekten** denken!
- **Funktionen** kapseln!
- Schwerpunkt auf **Gesamtdesign**, statt Einzelfunktionen





Modularität

- Kooperierende Bausteine
- Schnittstellen
- Ersetzen / Verbessern / Verfeinern



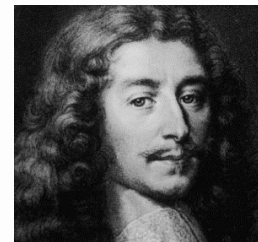
Wiederverwertbarkeit

- Vorhandene Klassen nutzen
- Eigene Klassen integrieren



» **Wer sich zu viel mit dem Kleinen abgibt, wird unfähig für Großes**

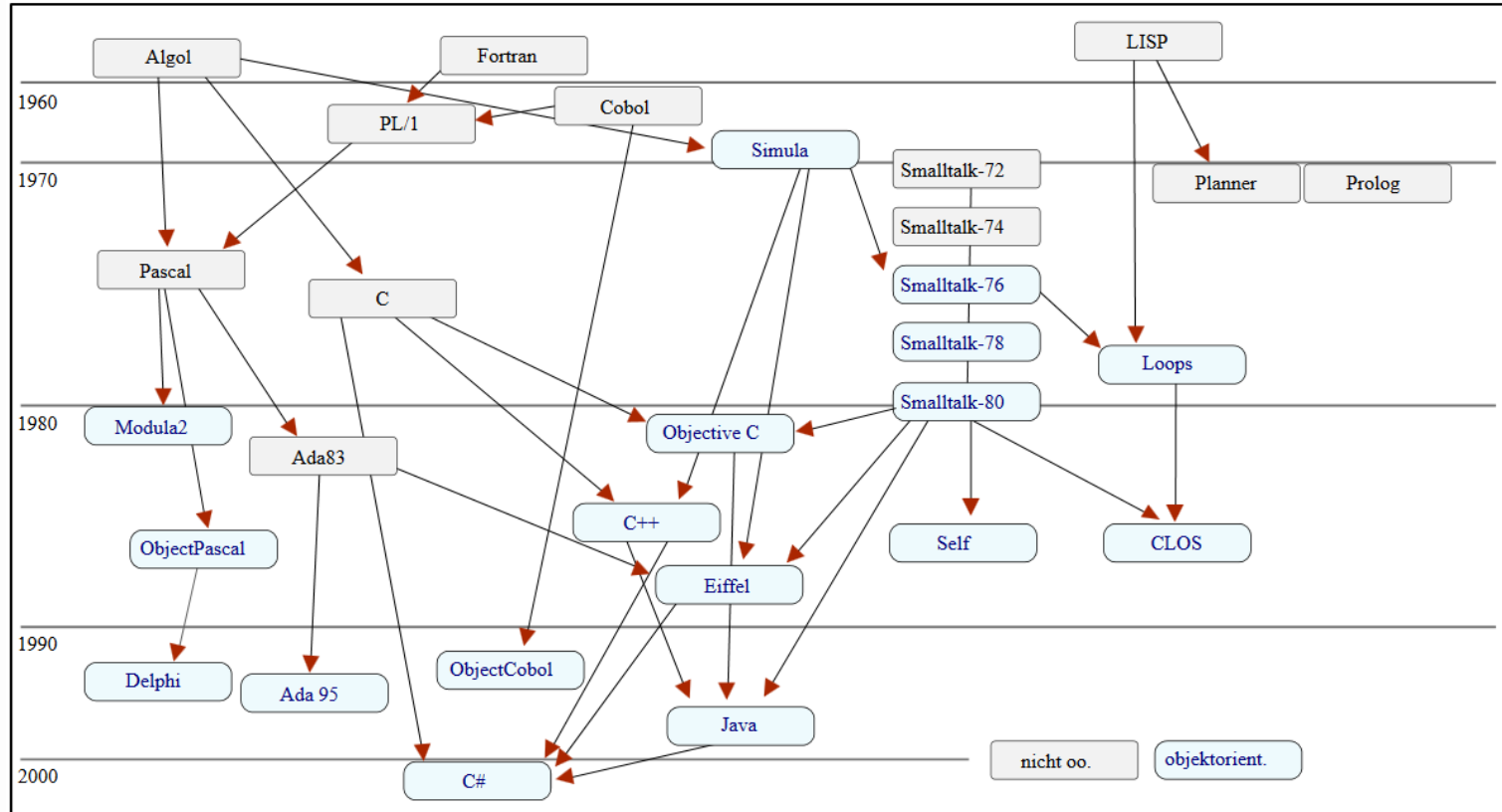
— Francois Rochefoucauld (1613–1680)



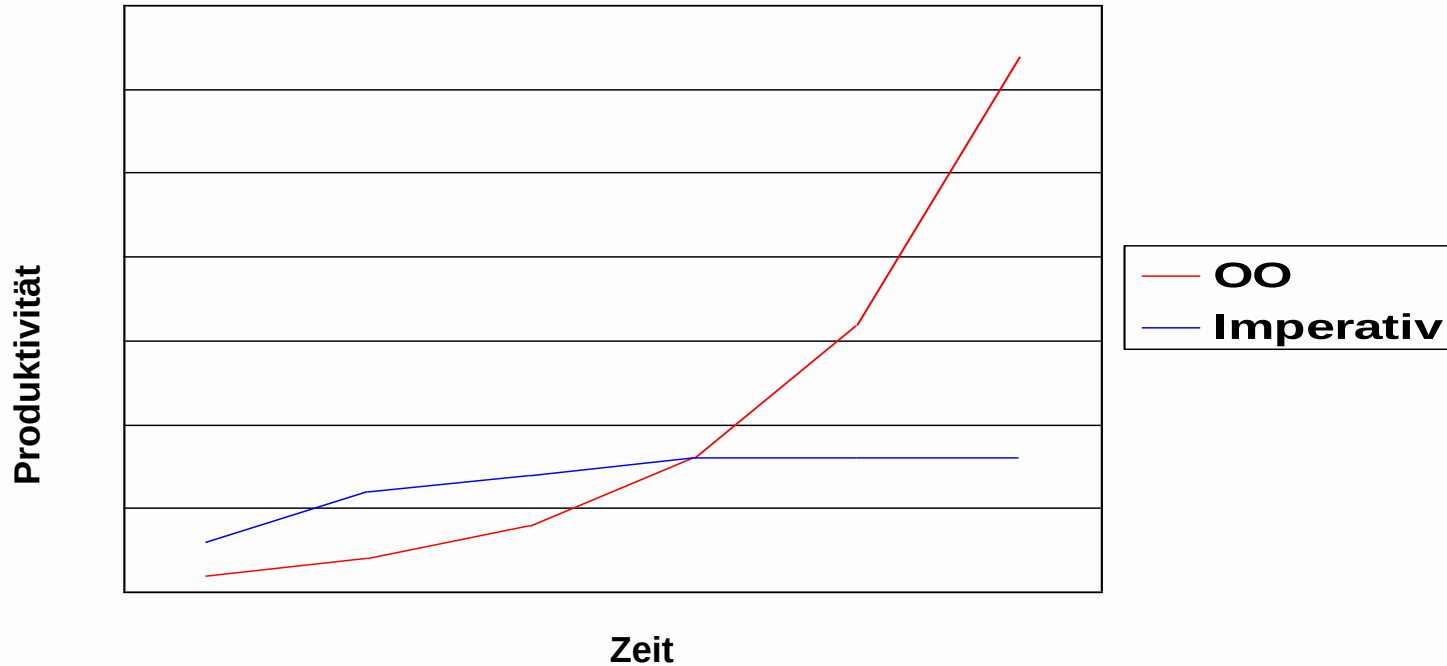
Bjarne Stroustrup (Schöpfer von C++), sagte treffend über den Vergleich von C und C++:

» **C makes it easy to shoot yourself in the foot; C++ makes it harder, but when you do, it blows away your whole leg.**

Geschichte der Programmiersprache



Zeit-Produktivität



Programmiersprachen

imperative Sprachen

Ada, ALGOL, BASIC, C, COBOL, FORTRAN, Modula-2, PL/1, Pascal, Simula, Snobol

funktionale und applikative Sprachen

Lisp, Logo

prädikative Sprachen

Prolog

objektorientierte Sprachen

Smalltalk, Eiffel, Oberon, Java, Turbo-Pascal, C++, Fortran2000

imperative Sprachen (I)

Ein Programm besteht aus einer Menge von Befehlen.

$$n! = 1 * 2 * 3 * \dots * n$$

```
10 let n=...  
20 gosub 50  
30 nf=nfak  
40 end  
50 let nfak=1  
60 for i=1 to n  
70 nfak=nfak*i  
80 next i  
90 return
```

BASIC

Prozedurale Sprachen gruppieren Befehle in **Unterprogrammen**.

```
int fac(int n) {  
    if (n==0) return 1;  
    return n*fac(n-1);  
}  
  
...  
  
nf = fac(n);
```

C

Ein Programm besteht aus einer **Menge von Funktionen** und deren Anwendung.

```
(fac (n)
  (cond
    ((equal n 0) 1)
    (times n (fac
      (difference n 1)
    )
  )
)
...
(fac n)
```

Lisp

In prädikativen Sprachen wird eine Menge von Fakten und Regeln vorgegeben.

```
fac(0,1) .  
fac(n,f) -> fac(n-1,f/n) .
```

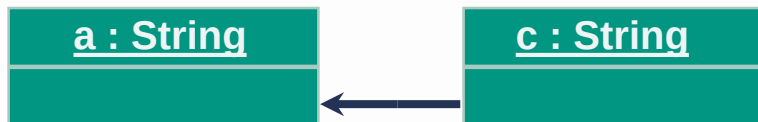
Prolog

objektorientierte Sprachen

alle zum Lösen eines Problems nötigen Informationen werden als Objekte aufgefasst. **Objekte empfangen und versenden Nachrichten**

```
String a = „eins“;  
String b = „zwei“;  
String c;  
c = a + b; // „einszwei“  
c = a.substring(0,2); // „ei“
```

JAVA



Abstrakter Datentyp (I)



Das **Basiskonzept** der objektorientierten Programmierung bildet der ***abstrakte Datentyp* (ADT)**

Ein ADT wird definiert durch:

- seinen **Wertebereich** (Sorte)
- die über dem Wertebereich erklärten **Operationen**
- eine Menge von **Vorbedingungen**
- eine Menge von **Axiomen**

Abstrakter Datentyp (II)

Eine Sorte kann formal **als Struktur** dargestellt werden

Beispiel:



Bzw:



Abstrakter Datentyp (III)

einem ADT werden Operationen zugeordnet
(die nicht notwendig nur Operanden vom Sortentyp enthalten)

Beispiel:

```
vec = ADT(koord)

  SetX: (vec, Integer);

  +: (vec, vec) -> vec;

  norm: (vec) -> Integer;

end;
```


Abstrakter Datentyp (IV)

Zur vollständigen Beschreibung eines ADT's gehören zusätzlich...

... die Vorbedingungen

```
pre: x = 0; y = 0;
```

... eine Menge von Axiomen (Spezifikationen), die das Verhalten der Operationen festlegen

```
(vec a, vec b) = (x = a.x + b.x, y = a.y + b.y)
```

Abstrakter Datentyp (V)

Abstrakte Datentypen werden als Klassen implementiert.

Beispiel (JAVA):

```
class vec {  
    private Integer x,y;  
    public vec(Integer theA, Integer theB) {  
        x = theA; y = theB;  
    }  
    public vec add(vec a, vec b) {  
        return new vec(a.x + b.x, a.y + b.y);  
    }  
}
```

Ich-Ansatz für Analyse / Design

» Ich bin

... Klasse XY «

» Ich habe

... folgende Eigenschaften (Attribute) «

» Ich kann

... folgende Operationen «

Beispiel: Point

- Ich bin ein Punkt.
- Ich habe x, y -Koordinaten.
- Ich kann...
 - ...mich verschieben.
 - ...meine Position festlegen.

Übliche Darstellung

Unified Modeling Language (UML)

Sichtbarkeit:

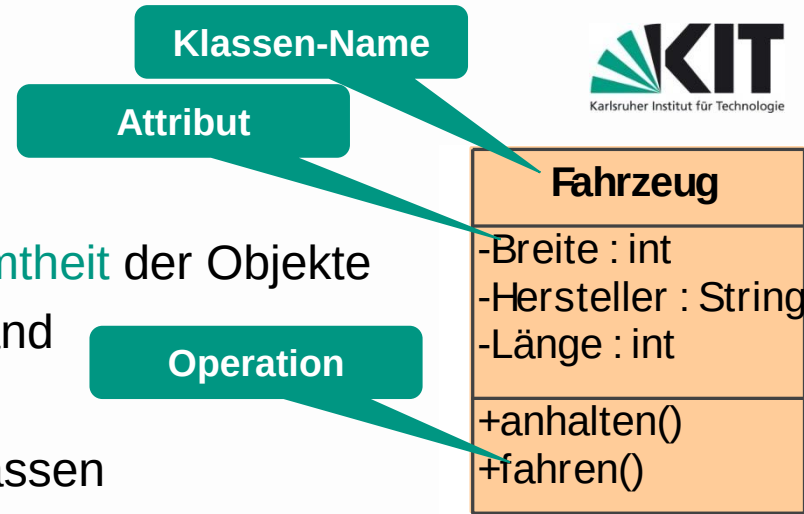
+ public - private # protected

java::awt::Point
+ x: int + y: int
+ Point(in p: Point) + Point() + Point(in x: int, in y: int) + equals(in obj: Object): boolean + getLocation(): Point + getX(): double + getY(): double + move(in x: int, in y: int) + setLocation(in x: double, in y: double) + setLocation(in x: int, in y: int) + setLocation(in p: Point) + toString(): String + translate(in dx: int, in dy: int)

→ UML: siehe Vorlesung SSE

Klasse

- Beschreibt das Verhalten der Gesamtheit der Objekte
- Hat keine Identität und keinen Zustand
- Muster für ein Objekt
 - Attribute, Operationen, innere Klassen



Eine **Instanz** ist die konkrete Realisierung eines Objektes einer Klasse

- eine Instanz ist genau ein Objekt
- Instanzen haben einen Gültigkeitsbereich (Lebensbereich)
- Instanzen werden durch einen **Konstruktor** erzeugt und einen **Destruktor** zerstört
 - Konstruktoraufruf: `new`
 - Destruktoraufruf: `delete`

Begriffe: Klasse und Objekt

```
Point p = new Point();
p.x = 12;
p.y = 45;
p.setLocation(-3, 2);
```

```
Point p = new Point();
p.setLocation();
```

- setLocation(double x, double y) void - Point
- setLocation(int x, int y) void - Point
- setLocation(Point p) void - Point
- setLocation(Point2D p) void - Point2D

Changes the point to have the specified location. This method is included for completeness, to parallel the setLocation method of Component. Its behavior is identical with move(int, int).

Parameters:

- x** the x coordinate of the new location.
- y** the y coordinate of the new location.

See Also:

```
java.awt.Component.setLocation(int, int)
java.awt.Point.getLocation
java.awt.Point.move(int, int)
```

```
Point p = new Point();
```

```
p.
```

- x int - Point
- y int - Point
- clone() Object - Point2D
- distance(double arg0, double arg1) double - Point2D
- distance(Point2D arg0) double - Point2D
- distanceSq(double arg0, double arg1) double - Point2D
- distanceSq(Point2D arg0) double - Point2D
- equals(Object arg0) boolean - Point
- getClass() Class - Object
- getLocation() Point - Point

Begriffe: Objekt

```
Point p1 = new Point();  
p1.x = 12;  
p1.y = 45;  
p1.setLocation(-3, 2);  
Point p2 = new Point();  
p2.x = 12;  
p2.y = 45;  
p2.setLocation(-3, 2);  
if (p1 == p2) { return 10; }  
if (p1.equals(p2)) { return 12; }
```

Objekt-Name

Klassen-Name

p1 : Pointx = 12
y = 45
location = (-3,2)p2 : Pointx = 12
y = 45
location = (-3,2)

Attribut-Name

Attribut-Wert

Objektidentität

Inhaltliche Identität

- obwohl p1 und p2 die **gleichen** Attributwerte haben, sind es voneinander **verschiedene Objekte**

Konstruktor (Objektinitialisierung)

- **Argumente** der Konstruktormethode werden bei Objekterzeugung folgendermaßen an den Konstruktor übergeben:

```
new Klassenname (Argument1, Argument2, ...)
```
- **Anweisungen** im Konstruktor werden auf neu allokiertem Speicher ausgeführt.
- Ohne Angabe eines Konstruktors ist nur der **leere Konstruktor** ohne Argumente definiert.
- Ist ein (nichtleerer) Konstruktor definiert, *muss* der leere Konstruktor explizit definiert werden, um noch benutzt werden zu dürfen.

Freigabe/Zerstörung von Objekten

In vielen Sprachen durch explizite Statements im Programm:

➤ **free-** oder **delete-Methoden**

Dies funktioniert – ist aber häufige *Fehlerquelle*:

Beispiel

1. **x**, **y** verweisen auf ein Objekt
 2. **free (x)** gibt Speicher frei
 3. **y** wird zu späterem Zeitpunkt verwendet
- **Problem!** Speicheradresse existiert, Speicherinhalt kann beliebig sein

Alternative in Java: **Garbage Collection**

Ist die letzte Referenz auf ein Objekt verschwunden, da...

... die Referenzen den Gültigkeitsbereich verlassen haben, oder da...

... **null**-Referenzen zugewiesen wurden,

so wird das Objekt der Garbage Collection zugeführt.

➤ Es sind keine **free**- oder **delete**-Methoden nötig.

- ✓ Funktioniert auch bei zyklischen Referenzen der Objekte untereinander
- ✓ Mit **System.gc()** ; wird eine Empfehlung an die Garbage Collection ausgegeben, aktiv zu werden. In der Regel ist dies aber nicht nötig.

Begriffe: Klasse und Objekt



Pakete (*Packages*)

Gruppe von thematisch zusammengehörigen Klassen und Interfaces.



Varianten

a) Absolute Qualifizierung (CLASSPATH)

```
java.awt.Point p = new java.awt.Point();
```

b) Relative Qualifizierung (CLASSPATH + Importpfad)

```
import java.awt.Point;  
  
Point p = new Point();
```

```
class Fahrzeug {  
    public String Hersteller;  
    private int Breite;  
    private int Laenge;  
    public void fahren();  
    public void anhalten();  
}  
...
```

Fahrzeug

-Breite : int
+Hersteller :String
-Laenge : int

+anhalten()
+fahren()

Begriffe: Klasse und Objekt

Variable

Klassen der Variable

Objekt-Erzeugung

Klassen-Name

```
..  
Fahrzeug meinFahrzeug = new Fahrzeug();  
meinFahrzeug.Hersteller = "Mercedes";  
Fahrzeug meinZweitfahrzeug = new Fahrzeug();  
meinZweitfahrzeug.Hersteller = "VW";  
}
```

Objekt-Referenz

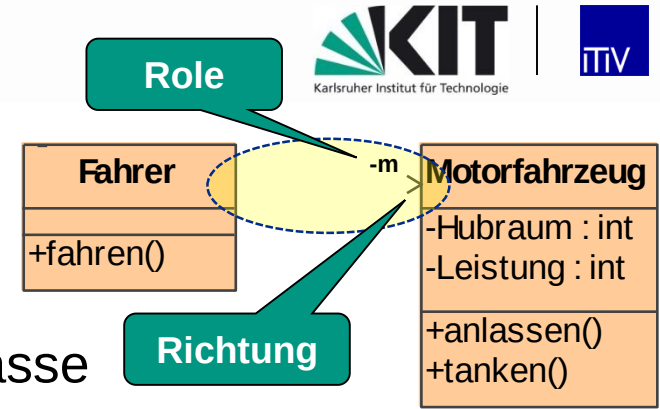
Attribut-Name

Attribut-Wert

Assoziation



- Klassen, die zusammenarbeiten.
- Gerichtet und ungerichtet.
- Benutzen das Interface der anderen Klasse

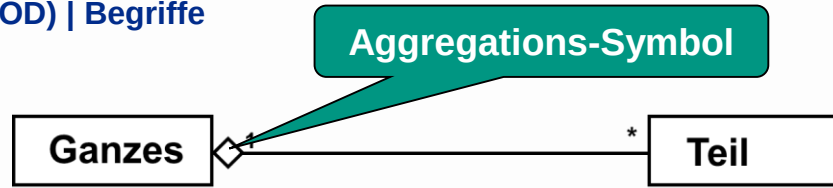


```
class Fahrer {
    private Motorfahrzeug;
    public void fahren() {
        m.anlassen();
        m.tanken();
    } ...
}
```

Aggregation



Aggregation



Eine Beziehung, bei der Komponenten(-Objekte) als Teile des Aggregat(-Objektes) aufgefasst werden.

Das Aggregat vertritt die Komponenten und kontrolliert sie; die Komponenten sind **dem Aggregat untergeordnet**.

Sprechweise

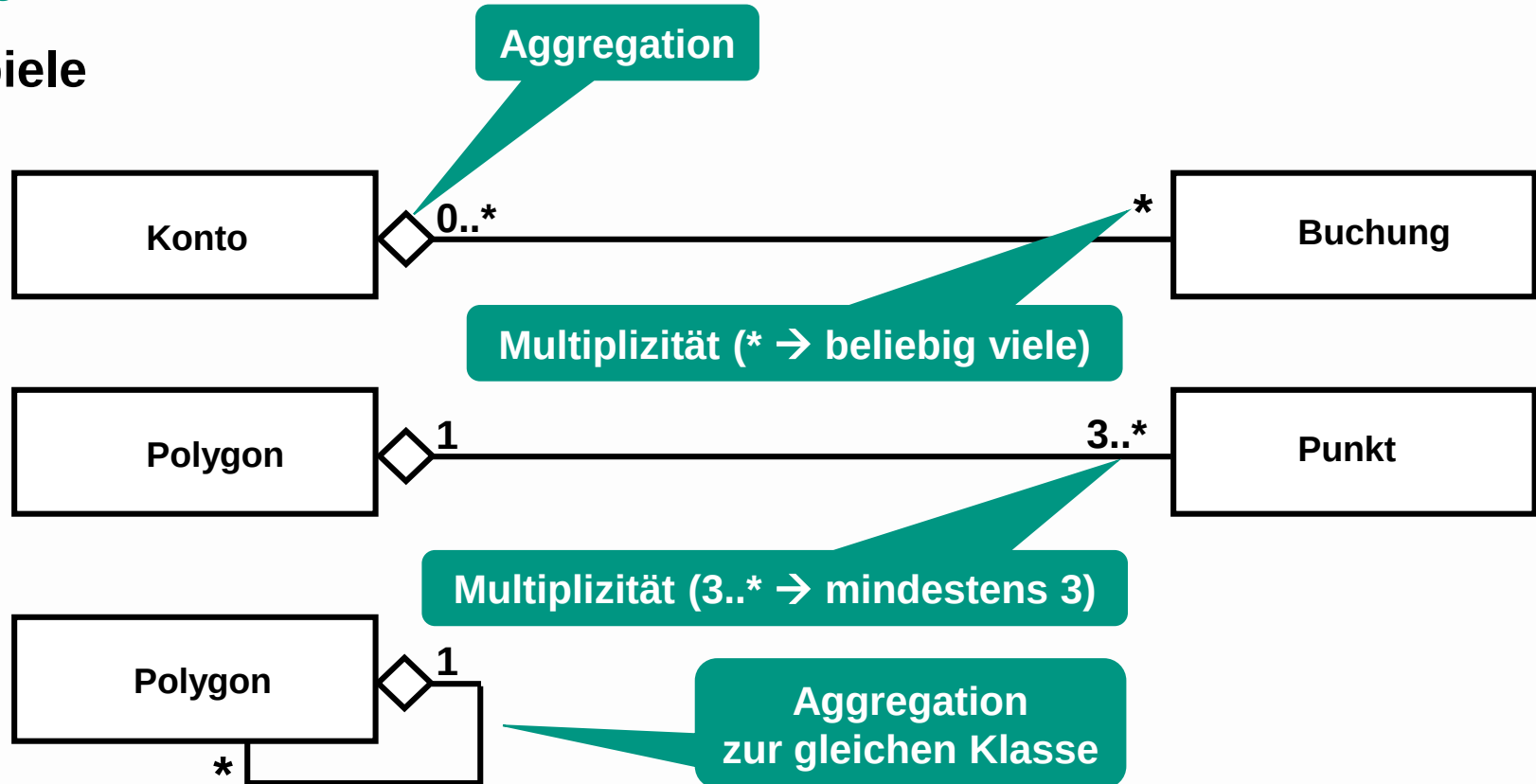
- »Das Aggregat **enthält seine Komponenten**.«
- »Die Komponente ist Teil des Aggregats.«

Aggregation ist **Spezialfall der Assoziation**, aber:

- **asymmetrisch** (nur eine der Klassen ist Aggregat)
- die Beziehung ist als „**enthält**“ bzw. „**ist-Teil-von**“ vordefiniert

Aggregation

Beispiele



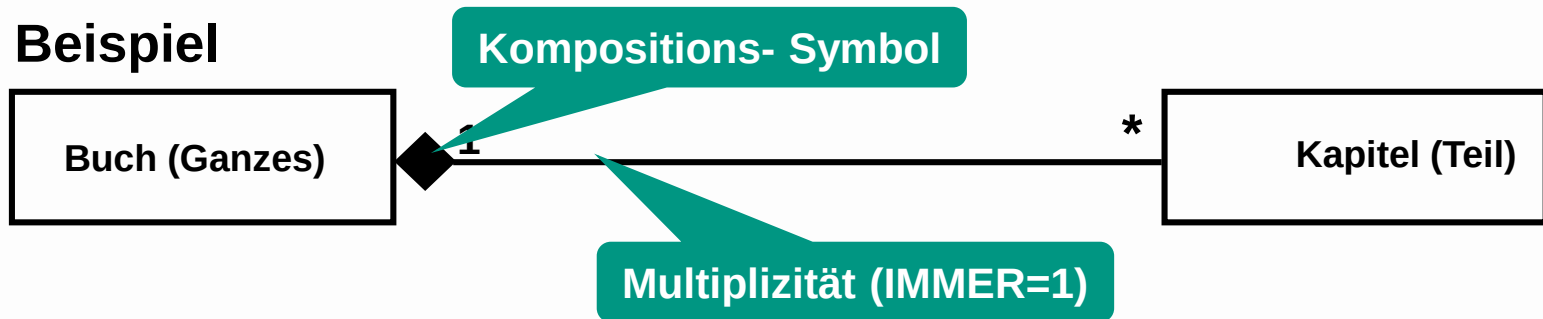
Komposition

 **Komposition**
Eine Aggregation, bei der die **Existenz** der Komponenten an die Existenz des Aggregats **gekoppelt** ist.

Das Aggregat **delegiert** Aufgaben an die Komponenten.

Eine Komponente darf **nur zu genau einem** Aggregat gehören.

Beispiel



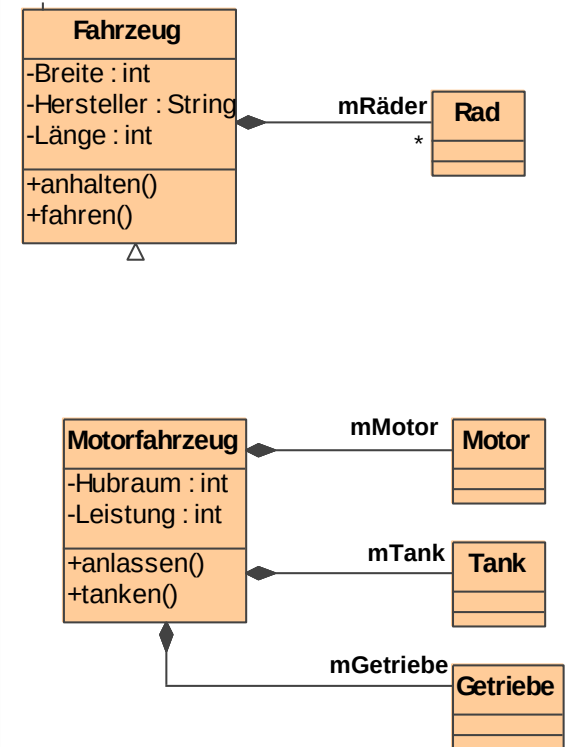
Kompositions-Assoziation



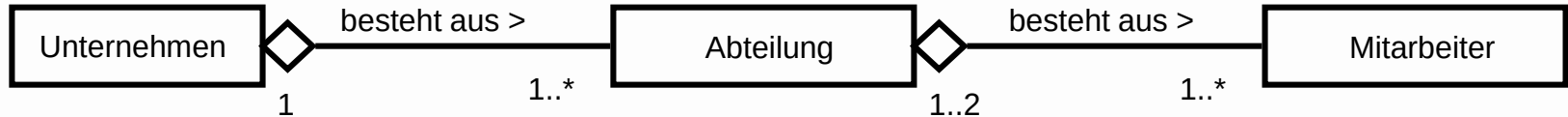
„Ist-Teil-Von“- oder „Besteht-Aus“-Beziehung

```
class Fahrzeug {
    RadList mRaeder;
}

class Motorfahrzeug extends Fahrzeug {
    Motor mMotor;
    Tank mTank;
    Getriebe mGetriebe;
}
```

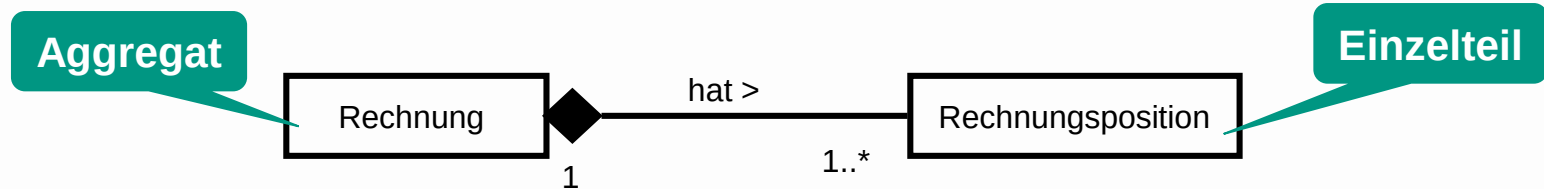


Normale Aggregation



Komposition

(Einzelteile sind vom Aggregat existenzabhängig,
d.h. sie können ohne das Aggregat nicht existieren.)



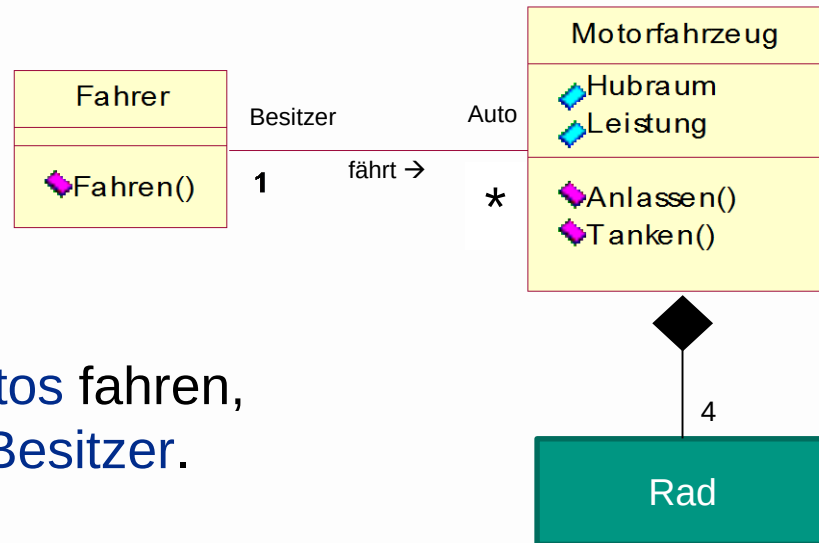
Hinweis: Bei Komposition auf der Aggregatseite immer 1 (kann nur zu genau einem gehören).

Kardinalität (Multiplizitäten)

 Anzahl der beteiligten Objekte an Assoziation und Aggregation

Beispiele:

- Ein Objekt der Klasse **Auto** hat 4 **Räder**.
Jedes **Rad** gehört zu einem **Auto**.
- Ein **Fahrer** kann beliebig viele **Autos** fahren, jedes **Motorfahrzeug** hat einen **Besitzer**.



Notationen:

1

1..4

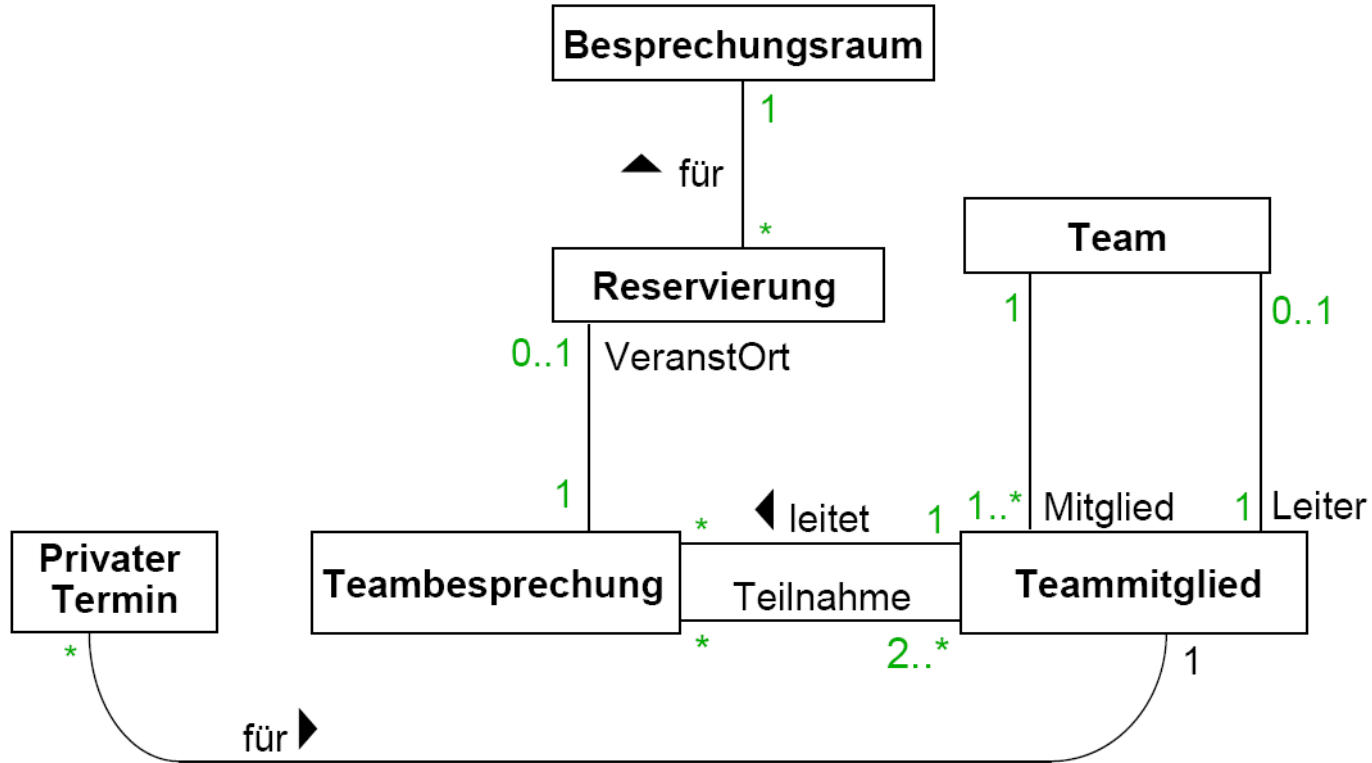
1...n

+

0..n

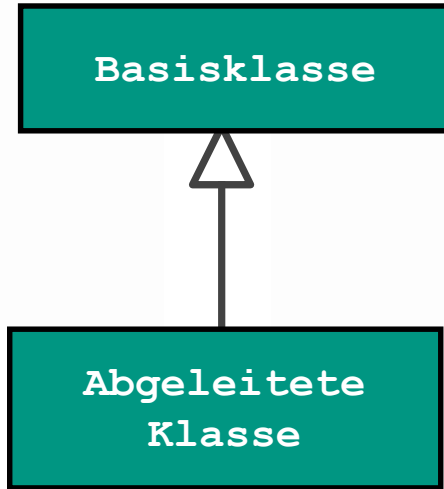
*

Beispiel: Multiplizitäten

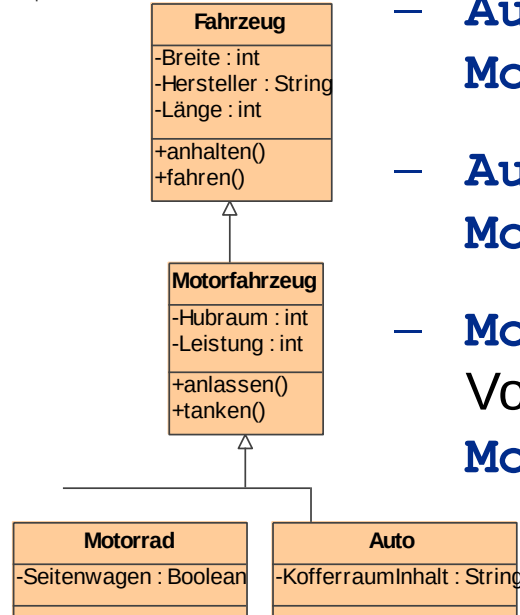


Vererbung

Notation

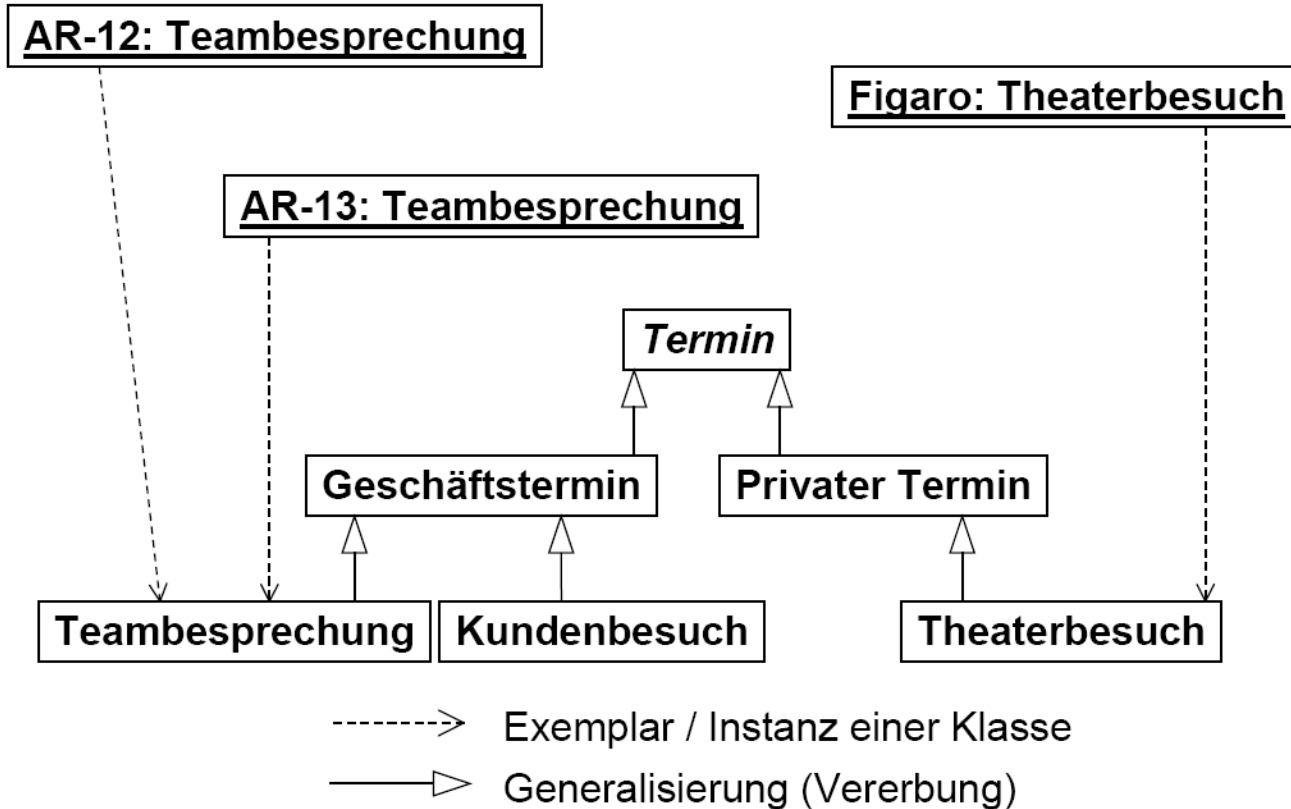


Beispiel



- **Auto** erbt von **Motorfahrzeug**
- **Auto** ist ein **Motorfahrzeug**
- **Motorfahrzeug** ist ein Vorfahr von **Auto** und **Motorrad**

Beispiel Klassendiagramm mit Vererbung



Vererbung

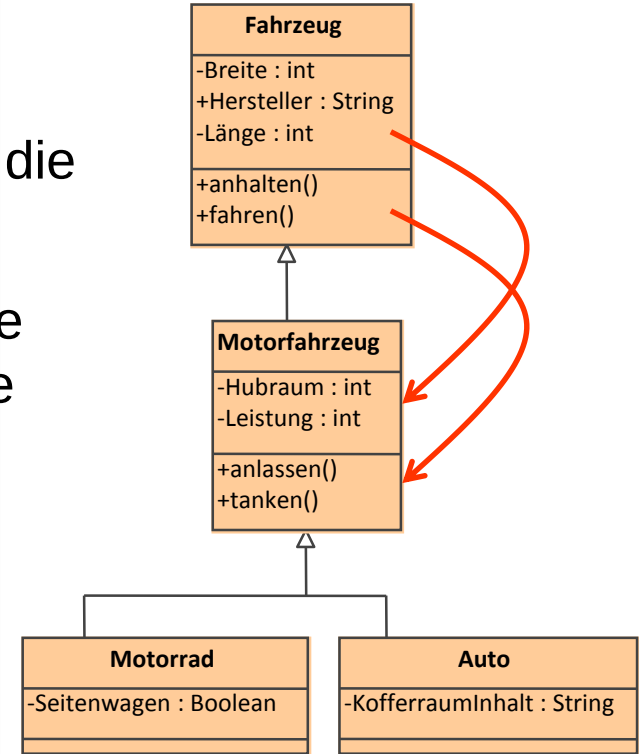
Weitergeben von *Attributen* und *Methoden* an die Erben (*Basisklasse* an eine *abgeleitete Klasse*)

Spezialisierung: Objektmenge der Unterklasse ist Teilmenge der Objektmenge der Oberklasse

Typhierarchie: jedes Objekt des Untertyps verhält sich wie ein Objekt des Obertyps

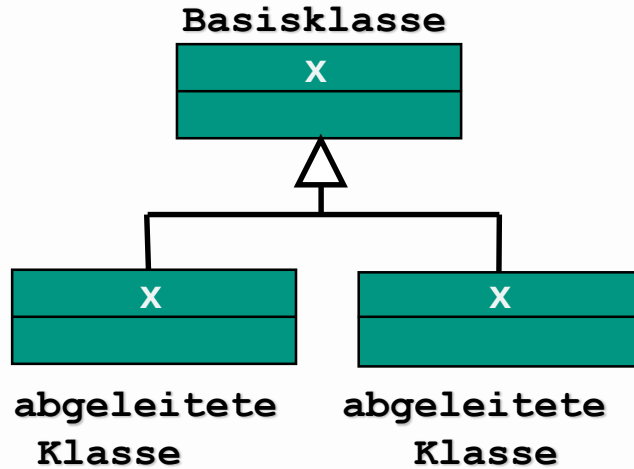
Klassenhierarchie: Vererbung der Implementierung

➤ Welche Methoden und Attribute hat **Auto** ?

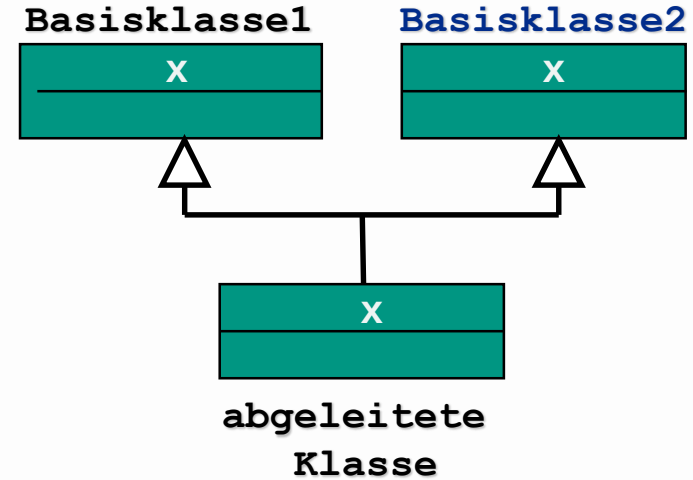


Einfache und mehrfache Vererbung

es wird ebenfalls zwischen **einfacher** und **mehrfacher** Vererbung unterschieden



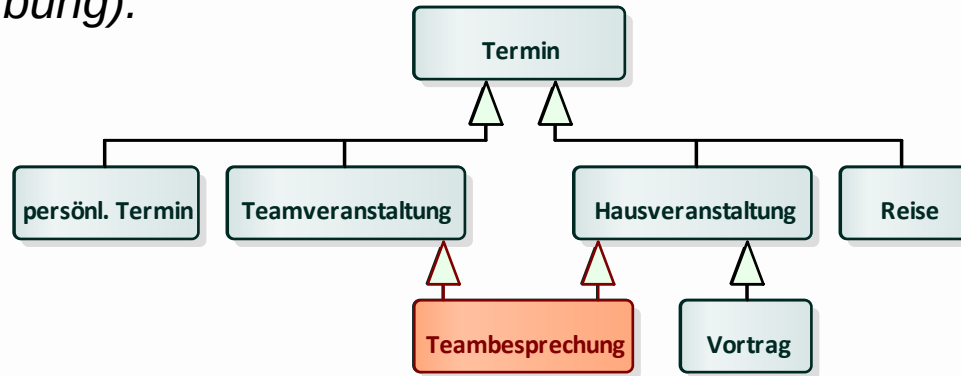
einfache Vererbung



mehrfache Vererbung

Mehrfachvererbung

In **Analysemodellen** treten oft unabhängige Dimensionen der Spezialisierung auf (*Mehrfachvererbung*).



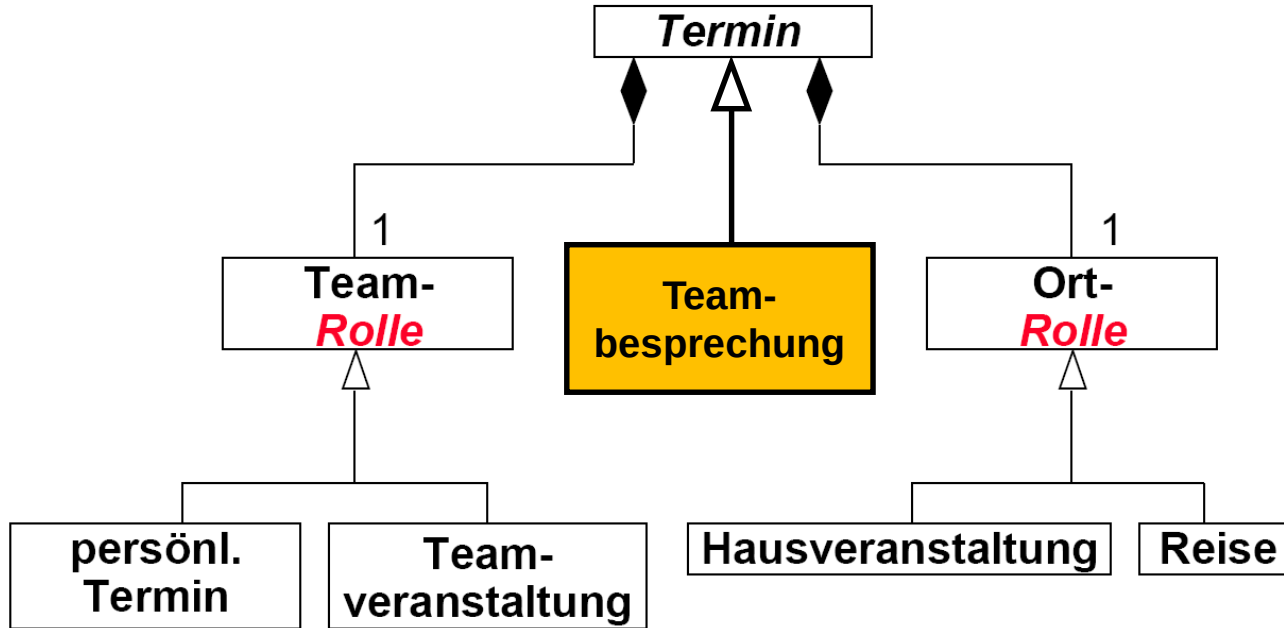
In **Entwurfsmodellen** sollten Mehrfachvererbung beseitigt werden.

Techniken dazu sind:

- Ersatz von Vererbung durch **Komposition**
- Definition von **Schnittstellen**

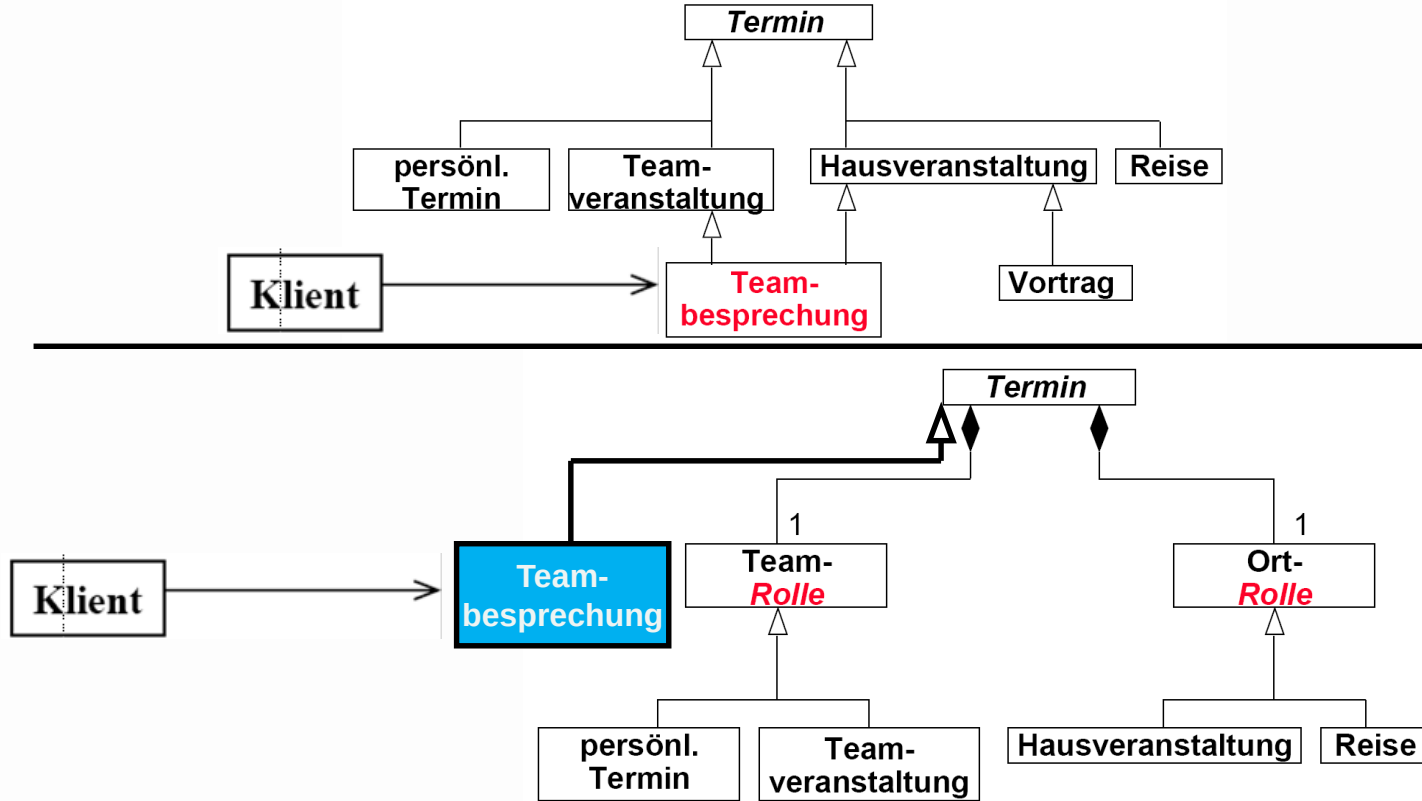
Composite Reuse Principle (CRP)

Ersatz von Vererbung durch Komposition

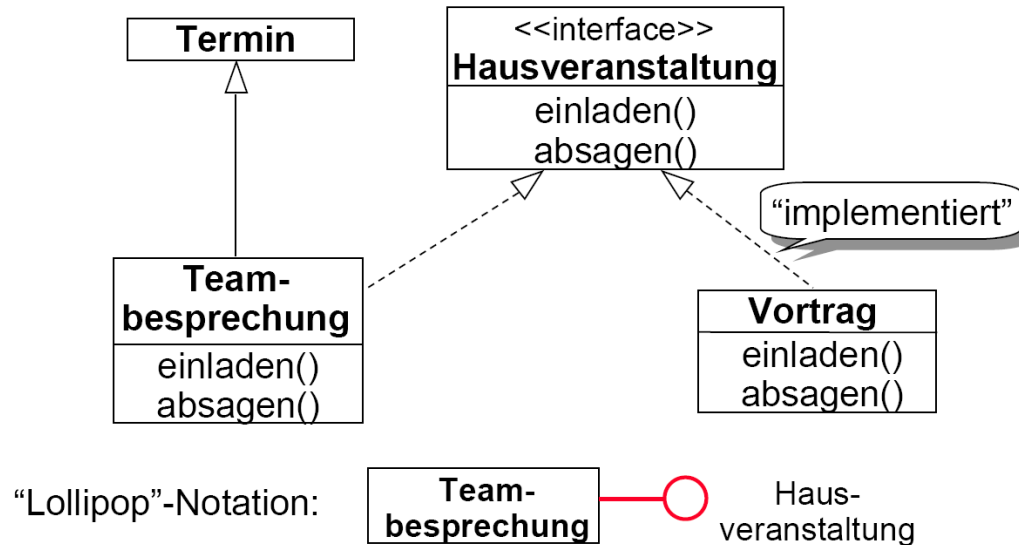


→ Teambesprechung *nun wie realisiert?*

Vergleich: Mehrfachvererbung und Komposition



Ersatz von Vererbung durch Schnittstellen

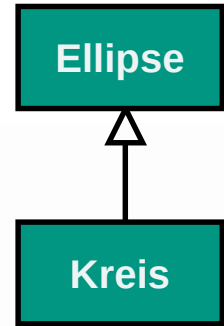


Guter Software-Entwurf sichert *Homogenität*.

- Gleichartige Funktionalität soll in gleicher Weise aufrufbar sein.
- *Schnittstelle* (interface) ist ein Sprachkonstrukt von UML und Java.

Vererbung: Methoden-Weitergabe

```
Ellipse::Skalieren(Real theA, Real theB) {  
    a=a*theA; b=b*theB; }
```



speziell auf die **Weitergabe von Methoden** bezogen ist...

Ersetzen: das vollständige
Überschreiben einer Methode

```
Kreis::Skalieren(Real f) {  
    a=a*f; b=b*f;  
}
```

Verfeinerung: die ererbte Methode wird
innerhalb der neuen Methode gerufen

```
Kreis::Skalieren(Real f) {  
    super.Skalieren(f, 1.0); b=b*f;  
}
```

Delegation: die Ausführung einer
Methode wird an ein Objekt der
Oberklasse (Prototyp) weitergereicht

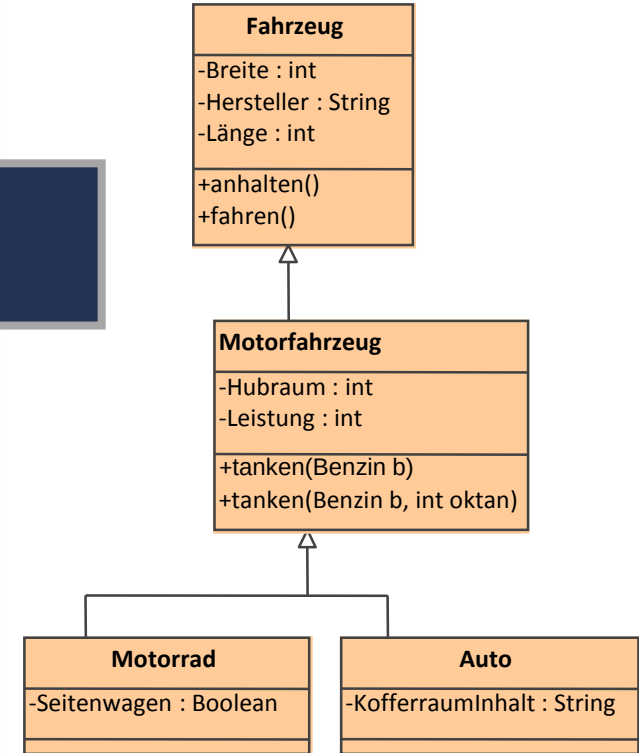
```
Kreis::Skalieren(Real f) {  
    super.Skalieren(f, f);  
}
```

Vererbung: Overloading

universeller Polymorphismus: Overloading

- Methode mit unterschiedlichen Signaturen

```
void tanken(Benzin b);  
void tanken(Benzin b, int oktan);
```



neben dem *universellen Polymorphismus* gibt es den *ad-hoc Polymorphismus*

```
void tanken(Benzin b) ;  
void tanken(Diesel b) ;
```

ad-hoc P. basiert auf dem Konzept der **impliziten Typkonversion**

Vererbung (III)

Polymorphie: **Overriding**

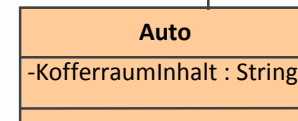
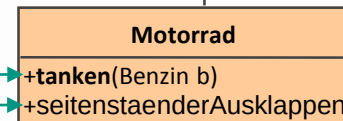
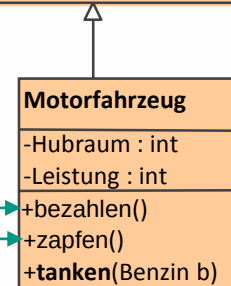
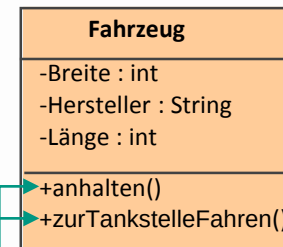
- Überschreiben von Methoden in abgeleiteten Klassen

```
void tanken(Benzin b) {  
    zurTankstelleFahren();  
    anhalten();  
    zapfen();  
    bezahlen();  
}
```

Motorfahrzeug

```
void tanken(Benzin b) {  
    zurTankstelleFahren();  
    anhalten();  
    seitenstaenderAusklappen();  
    zapfen();  
    bezahlen();  
}
```

Motorrad

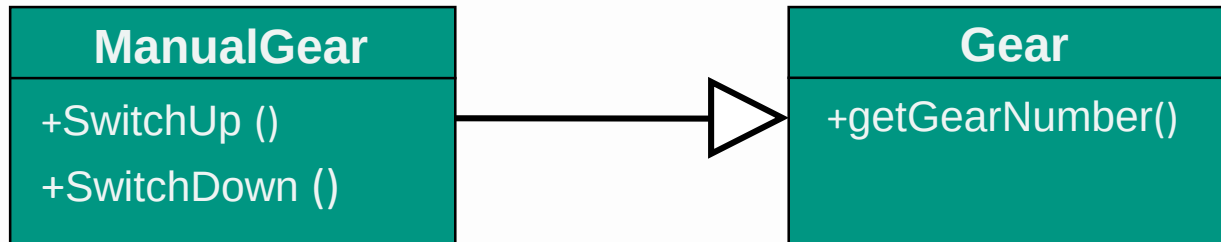


das Konzept der Polymorphie erlaubt somit **unterschiedliche Sichtweisen** auf Objekte

Beispiel

eine Instanz der Klasse **ManualGear** ist ebenfalls mit den Eigenschaften der Klasse **Gear** ausgestattet

➤ kann also wie eine solche agieren

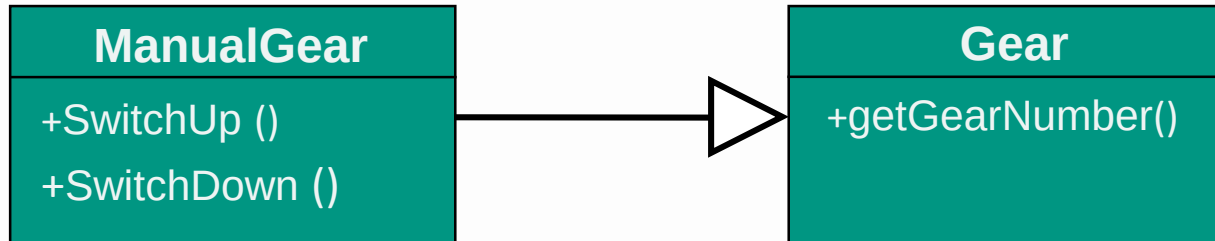


Polymorphie 4

eine Variable vom Typ einer Superklasse kann somit auch eine **Instanz vom Typ einer Subklasse** aufnehmen

Beispiel

```
Gear g;  
g = new ManualGear();  
int n = g.getGearNumber();  
g.SwitchUp() ;
```



Bindung 1



Der *Bindungsmechanismus* beschreibt, wie eine Methode einer Klasse aufgerufen wird.

Verschiedene **Bindungskonzepte** erlauben eine **effiziente Wiederverwendung** bzw. gemeinsame Nutzung von Code auf sehr elegante Art und Weise

→ **statische** und **späte** Bindung

→ **virtuelle** Methode

Statische Bindung

Während der Übersetzung wird die Klassenzugehörigkeit der Instanz bestimmt und daraus die zu rufende Methode ermittelt

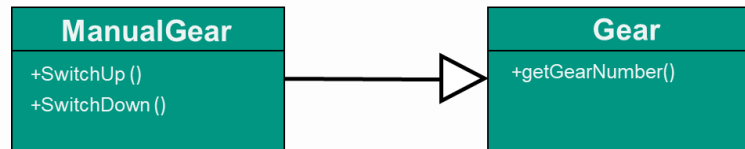
Beispiel

```
ManualGear g = new ManualGear();  
g.SwitchUp();
```

ruft die Methode der Klasse **ManualGear**

```
Gear g = new ManualGear();  
g.SwitchUp();
```

ruft die Methode der Klasse **Gear**



Späte Bindung

Erst zur Laufzeit wird die Klassenzugehörigkeit der Instanz bestimmt.

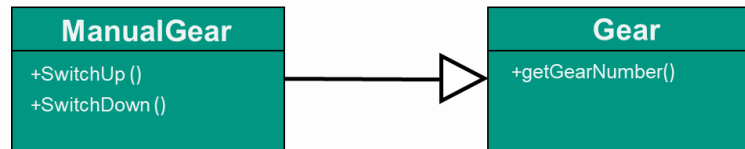
Beispiel

```
ManualGear g = new ManualGear();  
g.SwitchUp();
```

ruft die Methode der Klasse **ManualGear**

```
Gear g = new ManualGear();  
g.SwitchUp();
```

ruft ebenfalls Methode von **ManualGear**



Späte Bindung wird **nicht von allen Sprachen unterstützt**

standardmäßig wird **eine Methode statisch** gebunden

dynamisch zu bindende Methoden werden durch ein spezielles Schlüsselwort (meist **virtual**) gekennzeichnet

➤ ‚virtuelle Methoden‘

In JAVA sind alle Methoden ‚virtual‘

Beispiel in C++

```
class Gear {  
    Init();  
    virtual SwitchUp() { ... };  
}  
class ManualGear : Gear {  
    Init();  
    virtual SwitchUp() { ... };  
}  
Gear g = new ManualGear();  
  
g.Init();      // Gear::Init  
g.SwitchUp();  // ManualGear::SwitchUp
```

C++:

ein Verfahren zum Lösen von Gleichungssystemen

$$\mathbf{A} \mathbf{x} = \mathbf{b}$$

wird implementiert.

Ist A dünn besetzt, so werden die *nicht-0 Elemente*
normalerweise als Liste abgelegt, um Speicherplatz zu sparen

➤ der Löser müsste *neu implementiert* werden

Beispiel 2

die Neuimplementation des Löser ist *nicht nötig*, wenn der Löser erklärt wird über:

```
class Matrix {  
    int n, m;  
    Datatype data[maxz][maxs];  
    virtual real Get(int z,int s);  
    virtual void Set(int z,int s,real v);  
}
```

```
Matrix Solver(Matrix A, Matrix b);
```

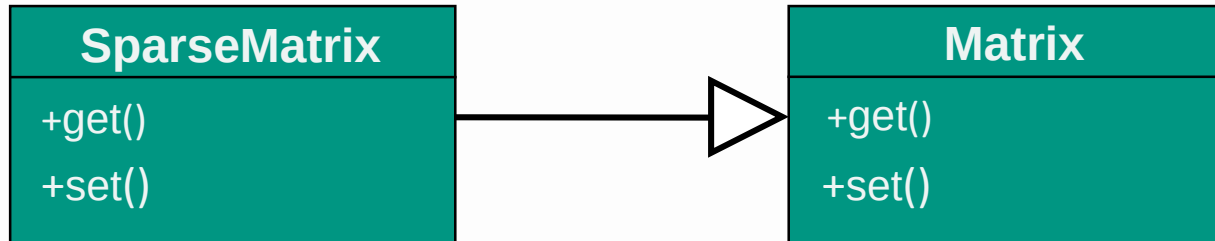
Beispiel 3

```
Matrix Solver(Matrix A, Matrix b) {  
    int i, j, k;  
    real f;  
    Matrix x = new Matrix();  
    for (k = 0; k < A.n; k++) {  
        if (abs(A.Get(k, k)) < eps) swap();  
        for (i = k+1; i < A.n; i++) {  
            f=A.Get(i,k)/A.Get(k,k);  
            for (j = k+1; j < A.m; j++) {  
                A.Set(i, j, A.Get(i, j) - f*A.Get(k, j));  
                b.Set(i, b.Get(i) - f * b.Get(k));  
            }  
        }  
        ... return x;  
    }  
}
```

Beispiel 4

eine dünn besetzte Matrix kann dann z.B. erklärt werden als:

```
class SparseMatrix : Matrix {  
    int n, m;  
    Datalist data;  
    virtual real Get(int z, int s);  
    virtual void Set(int z, int s, real v);  
}
```



Beispiel 5

der Löser kann auch dann auf **SparseMatrix**-Objekte angewandt werden, obwohl sein Quelltext nicht verfügbar ist (Bibliothek)

```
Matrix A = new SparseMatrix();  
Matrix b = new Matrix();  
Matrix x;  
  
...  
  
x=Solver(A, b);
```

Sichtbarkeit

Scopes

Public (+)

- Für alle sichtbar

Protected (#)

- Innerhalb Vererbungslinie sichtbar
- Nicht alle OO-Sprachen

Private (-)

- Nach außen nicht sichtbar, nur innerhalb deklarierender Klasse
- Variablen-Zugriff nur über separate Methoden
- *Anm:* in C++ mittels **friend** trotzdem direkter Zugriff erlaubt

SparseMatrix

```
+ get ()  
+ set ()  
- trace ()  
# getInfo()  
~ setPrefix()
```

Package (~)

- Innerhalb des gleichen Paketes wie public

```
class Fahrzeug {  
    public String Hersteller;  
    protected int Hubraum;  
    public int Leistung;  
    private String SchluesselNr;  
    public void fahren();  
    public void anhalten();  
    void nurInDiesemPaketPublic();  
}
```

Package protected

Kapselung



Interface einer Klasse

- Interface = Schnittstelle
- **Kontrakt:** „Was kann ich von einer Klasse erwarten?“
- Enthält nur Methoden und Attribute, die **von außen zugreifbar** sind

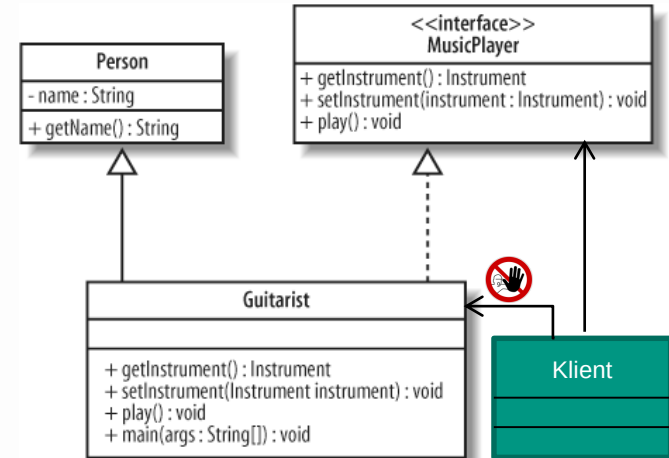
Vorteile

- Verteiltes Entwickeln
- Übersichtlichkeit

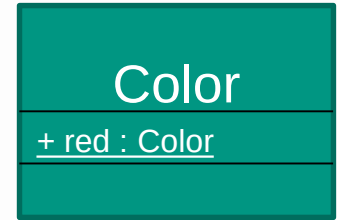


Kapselung

Ermöglicht **Austausch von Klassen**
durch ‚bessere‘ Varianten und Versionen



Klassenattribute existieren für alle Objekte der Klasse **nur einmal** gemeinsam.



Beispiel

Anzahl der Instanzen einer Klasse kann als Attribut der Klasse aufgefasst werden
Global zu definierende Werte, z.B. Farben:

```
Color.red wobei Color eine Klasse ist

public final static Color red = new Color(255,0,0);

Color x = Color.red;
```

Klassenmethoden als eine Art Funktionsbibliothek

Beispiel

Umwandlung von `float` in `String`:

```
String st = Float.toString(12.24);
```

Float

+ toString(float) : String

Abstrakter Datentyp

- Arten
- Zugeordnete Operationen

Klassenkonzept

- Klasse
- Instanz
- Objekt

Vererbung

- einfach
- mehrfach

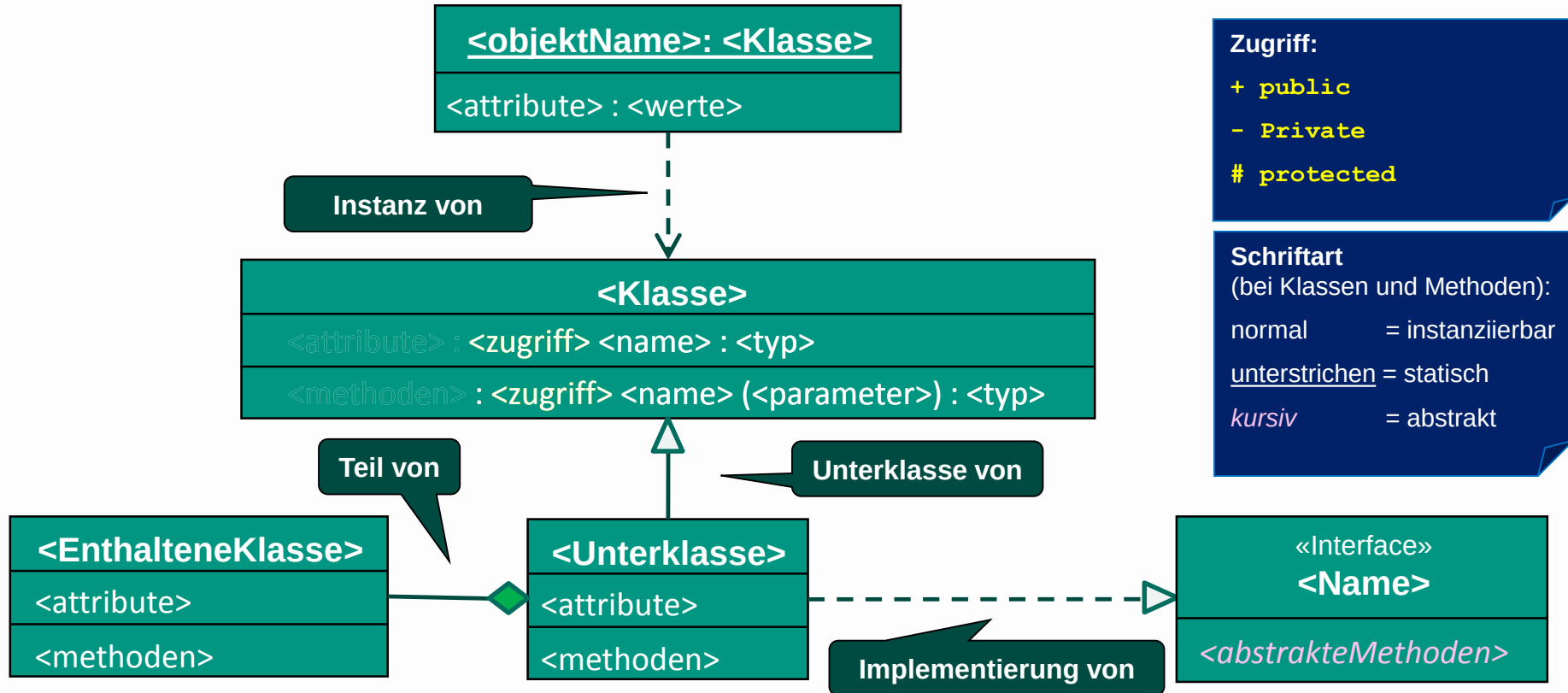


Polymorphie

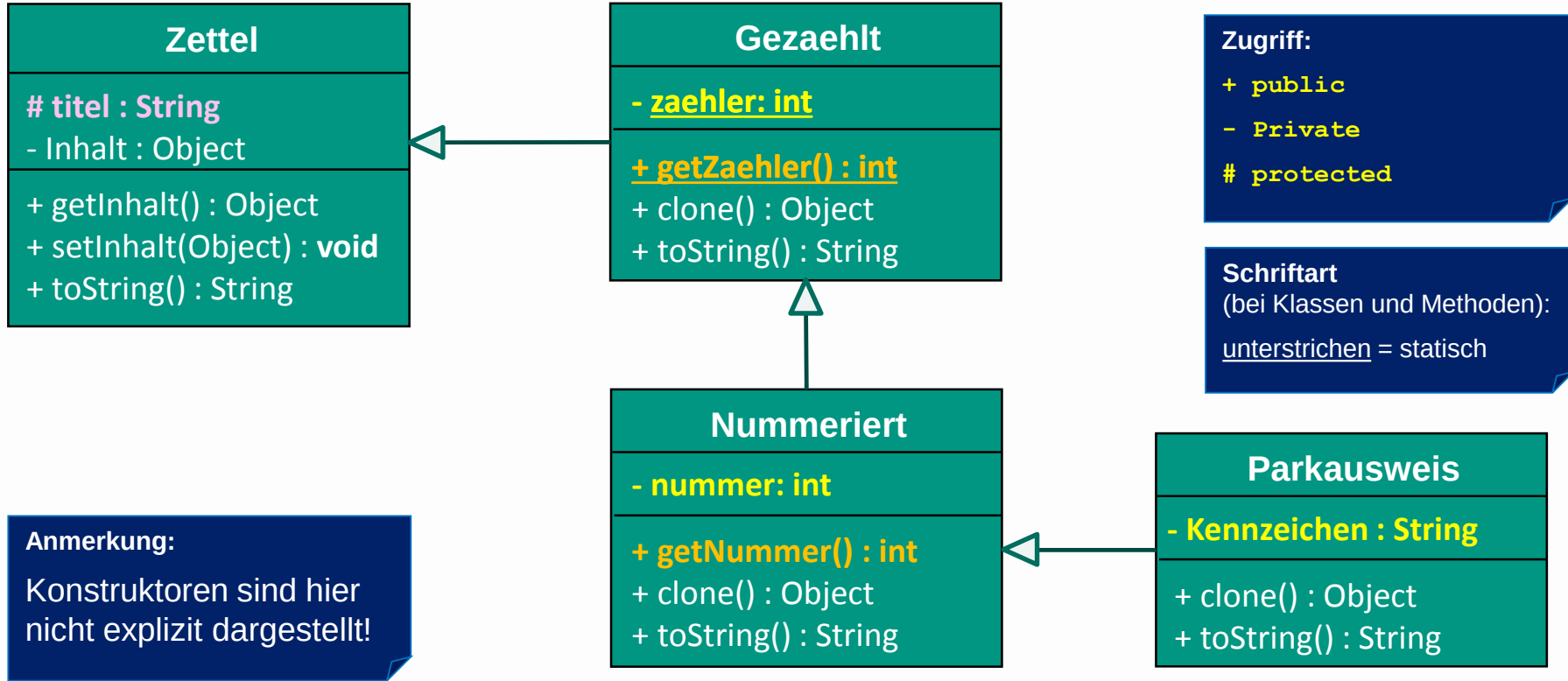
Bindung

- statische, späte
- virtuelle Methode

Zusammenfassung: UML-Elemente



Beispiel: Parkausweis

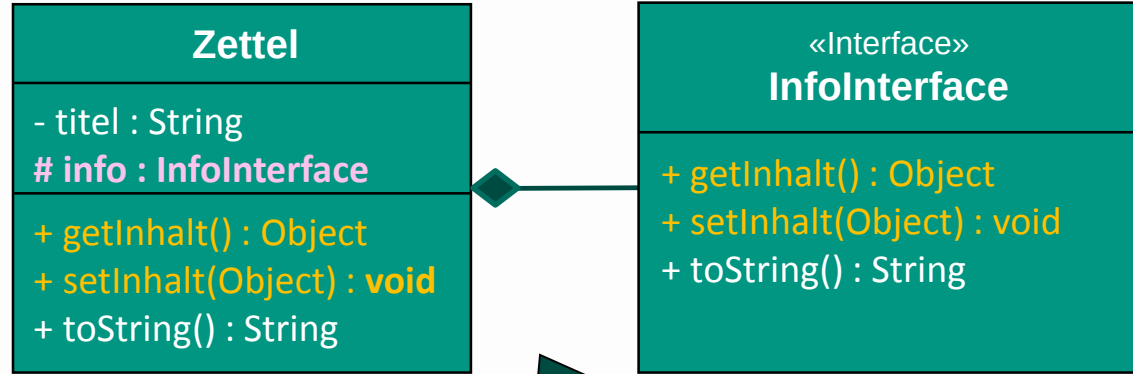


Delegation in UML

Delegation

zuvor `Zettel.getInhalt()`
jetzt `Zettel.info.getInhalt()`

Partner `info` leistet die Arbeit



„enthält“-Beziehung

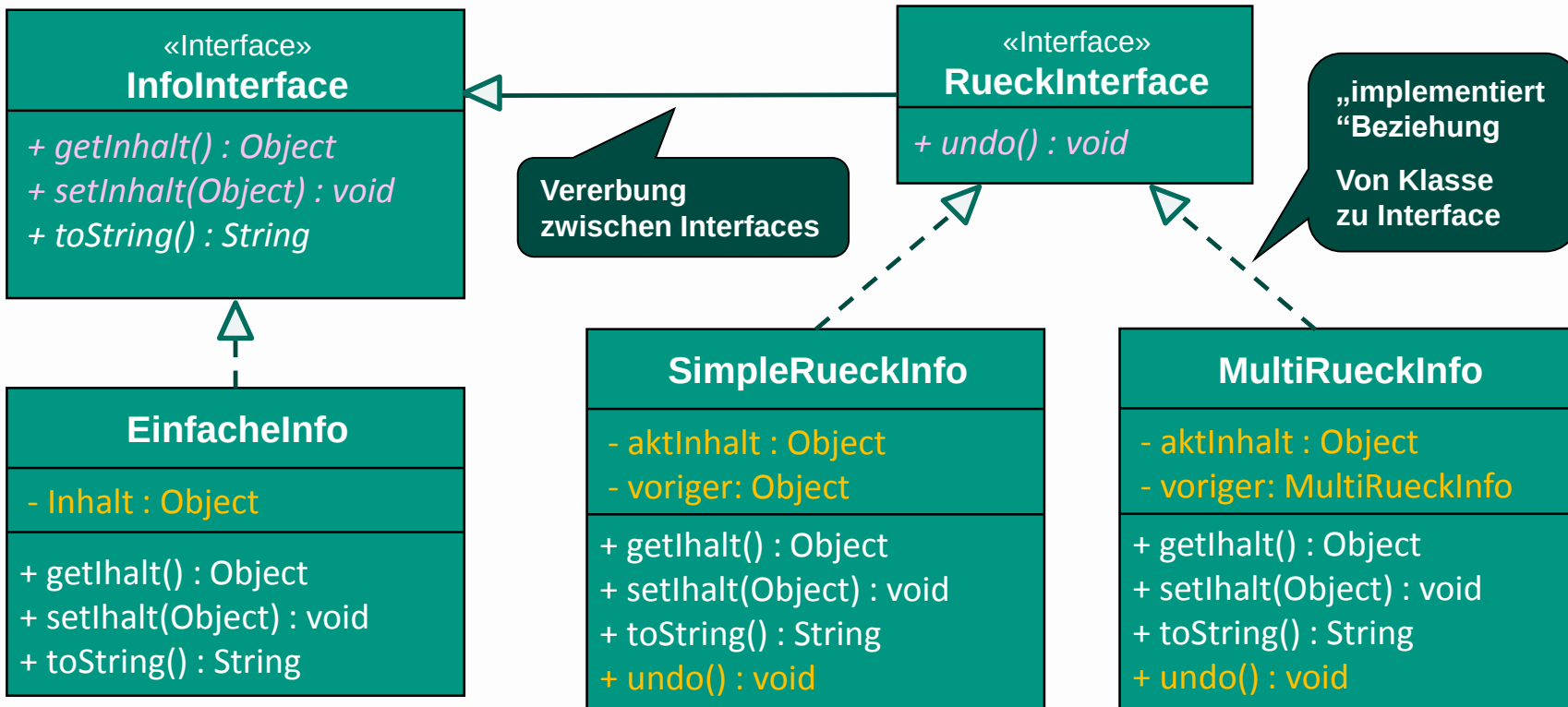
Hier: **Komposition**



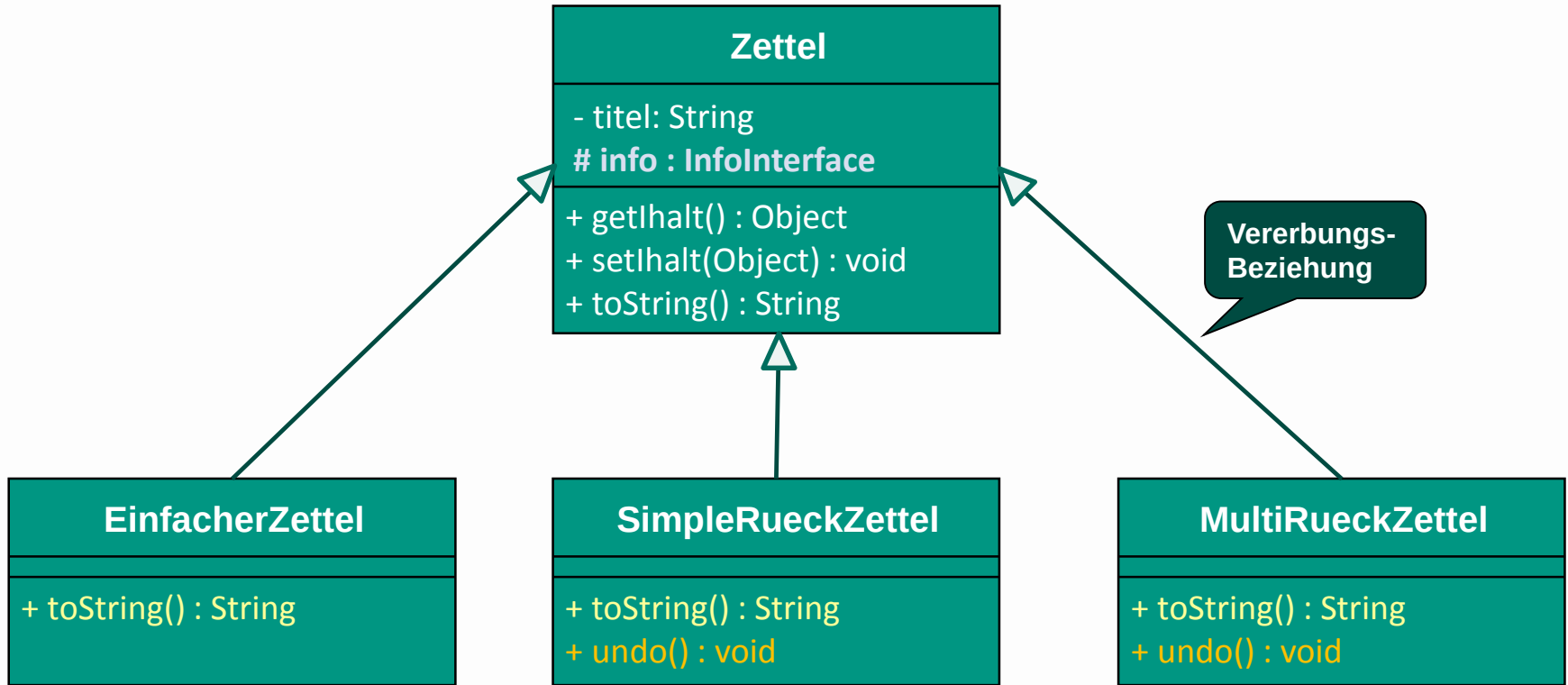
Sonst meist: **Aggregation**



Info-Anpassungen (Implementierung)



Zettel-Anpassungen (Reimplementierung)



Halbrichtig oder falsch?

- ? Java lässt keine eigenen Typdefinitionen zu.
- ? Daten sind Variablen.
- ? Objekte und Instanzen sind Teile einer Klasse.
- ? Objekte/Instanzen sind die einzelnen Variablen der jeweiligen Klasse.
- ? Die Klasse beinhaltet Variablen und Methoden. Diese Variablen gehören zur gesamten Klasse und nicht zu der Instanz der Klasse.
- ? Das **Institut für Technik der Informationsverarbeitung** ist ein Objekt der Klasse **Uni**.
- ? Mit einem Konstruktor legt man ein Objekt in der **main**-Methode an.

UML Symbole

