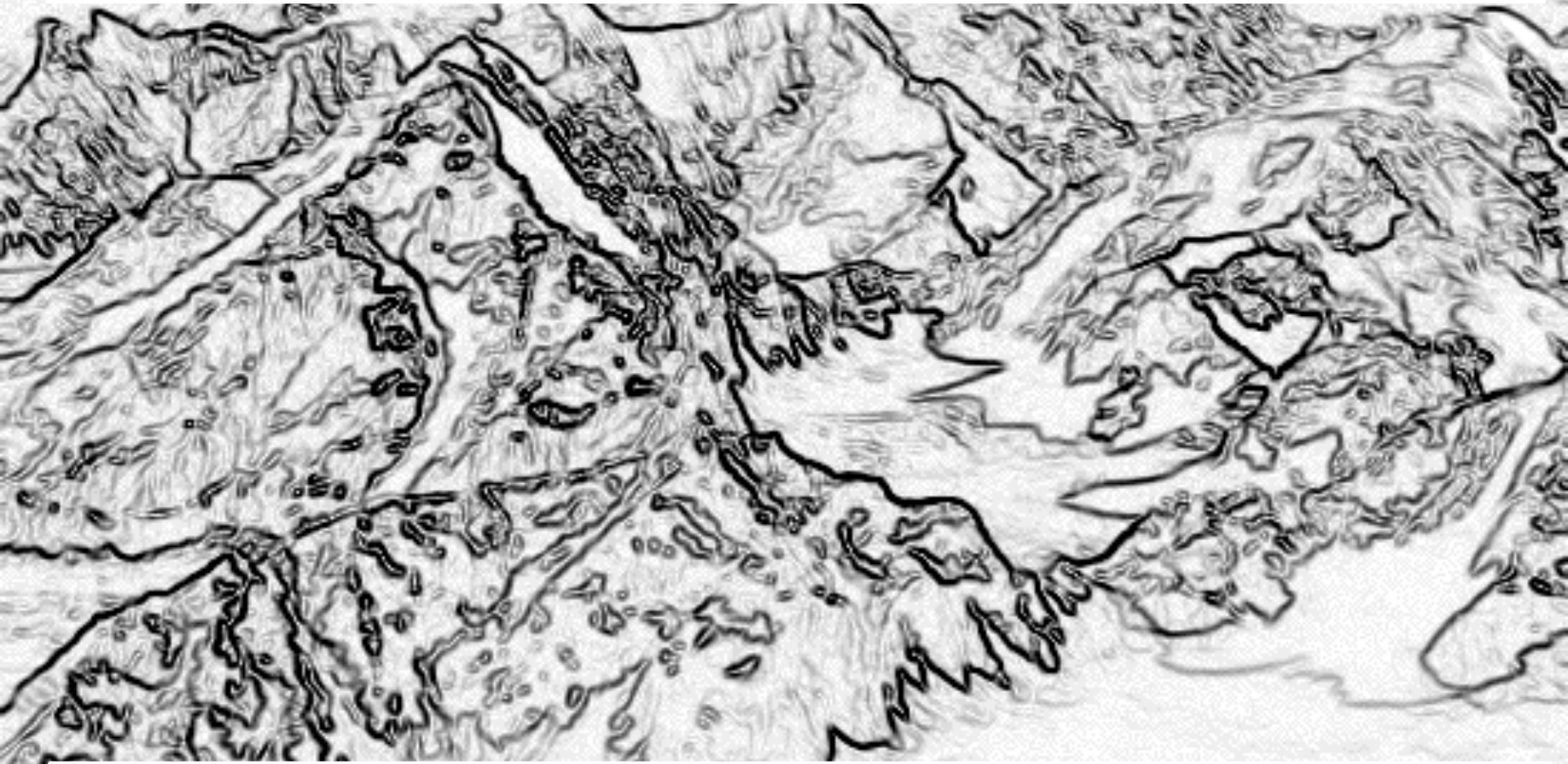


Software-Entwicklung 1

V03: Grundlagen der Objektorientierung



Prof. Maalej & Team - @maalejw



Status der 2. Übungswoche

Zeit	Montag	Dienstag	Mittwoch	Donnerstag	Freitag
Vor mittag	Gruppe 1 Erfüllt: 77%	Gruppe 3 Erfüllt: 51%	Gruppe 5 Erfüllt: 71%	Gruppe 6 Erfüllt: 56%	Gruppe 8 Erfüllt: 73%
Nach mittag	Gruppe 2 Erfüllt: 61%	Gruppe 4 Erfüllt: 65%	Vorlesung	Gruppe 7 Erfüllt: 53%	

Überblick

1

Objektorientierte Sichtweise

2

Grundbegriffe der Objektorientierung

3

Aufbau von Klassendefinitionen

Software-Entwicklung ist mehr als Programmierung



- **Problemlösen**

- Problemverstehen
- Vorschlag einer Lösung und eines Plans
- Experimentieren, programmieren und testen



- **Mit Komplexität umgehen**

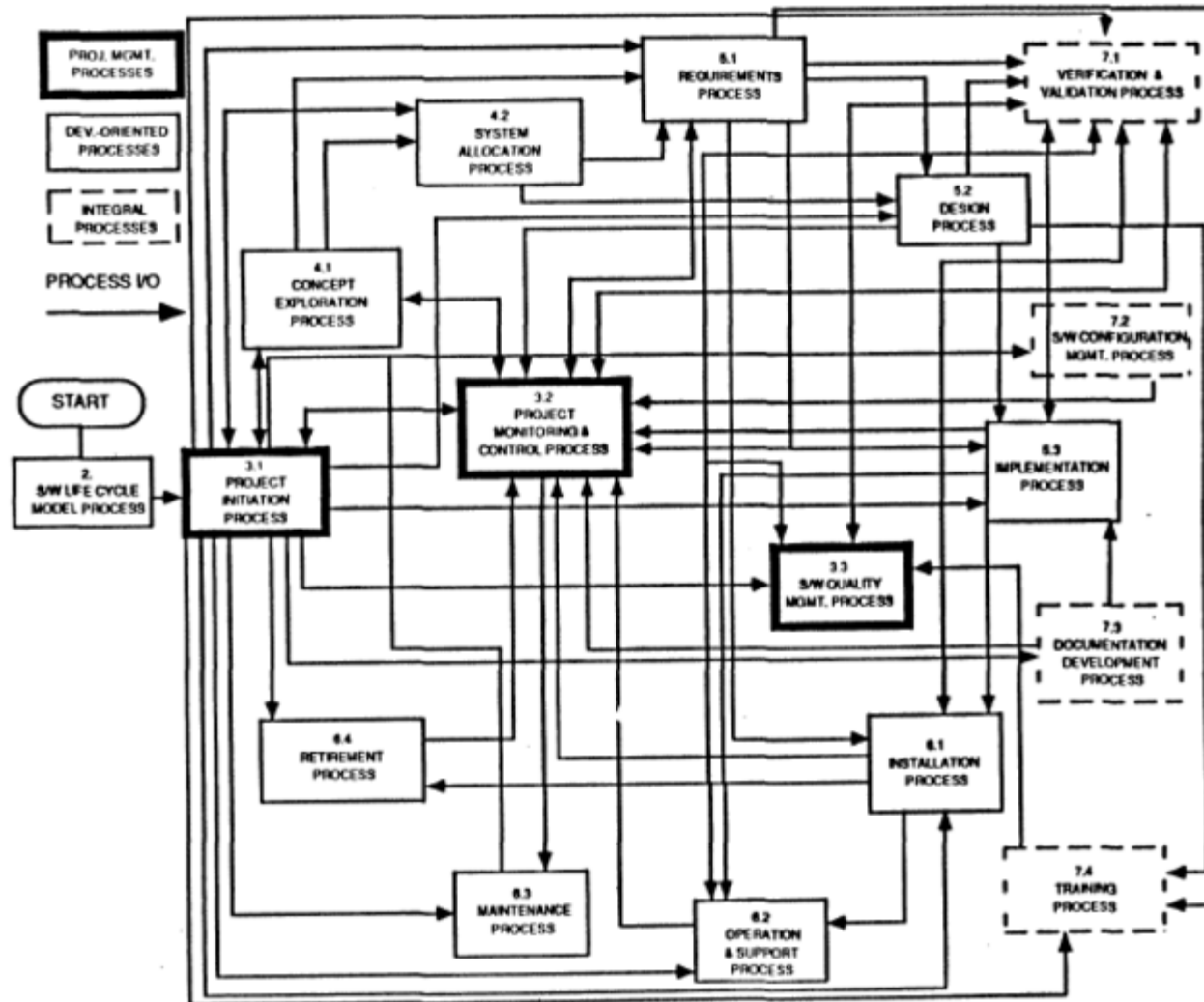
- Erstellen von Abstraktionen und Modellen
- Dekomposition (Zerlegung) des Problems

- **Kommunizieren**

- Mit dem Entwickler-Team
- Mit dem Kunden bzw. mit den Benutzern

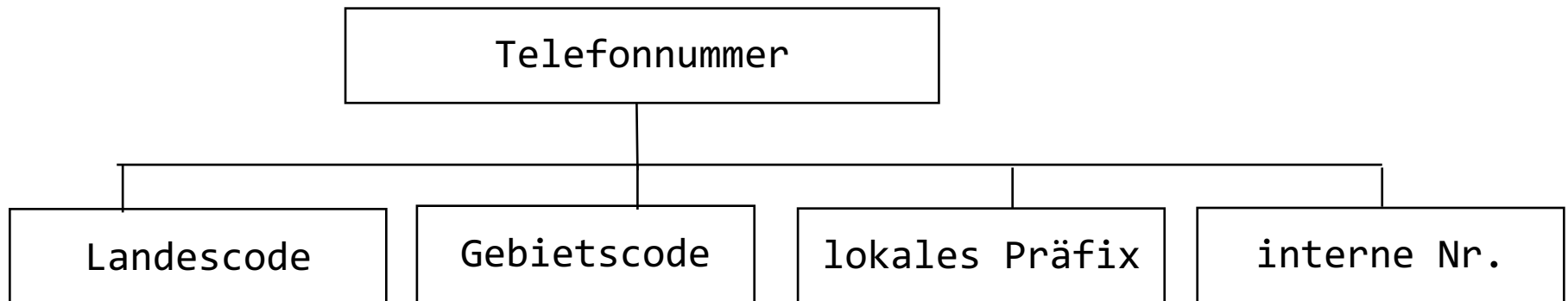


Was ist das Problem mit dieser Zeichnung?



Komplexe Systeme sind schwer zu verstehen

- Das 7 ± 2 Phänomen
 - Unser Kurzzeitgedächtnis kann nicht mehr als 7 ± 2 Dinge merken
 - Meine Tel. Nr: 004940428832073
- Dekomposition:
 - Reduziere Komplexität durch Unterteilung
 - Landescode, Gebietscode, lokales Präfix, interne Nr.



Abstraktionen helfen uns unwichtige Einzelheiten zu ignorieren

- **Modellierung:** Entwicklung von Abstraktionen um spezifische Fragen über das System zu beantworten
- Dabei ignorieren wir irrelevante Einzelheiten



Modelle sind...

...Abstraktionen von Systemen,

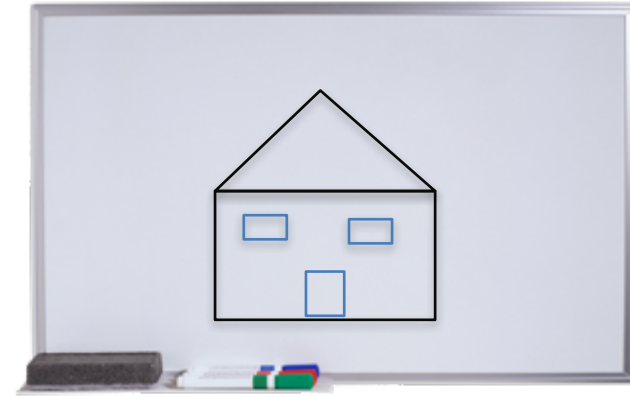
- die nicht mehr existieren
- die aktuell existieren
- die entwickelt werden sollen



Modellieren und Programmieren

Modellieren

- Abbild um zu zeigen, prüfen oder auszuprobieren
- *Beispiel:* Objektorientierte Modellierung



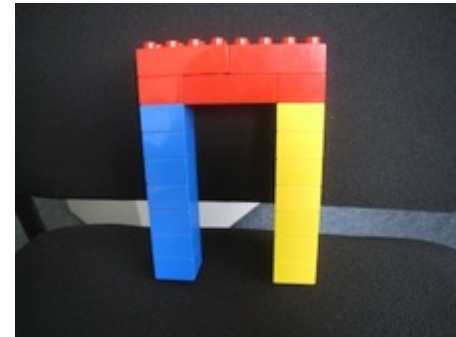
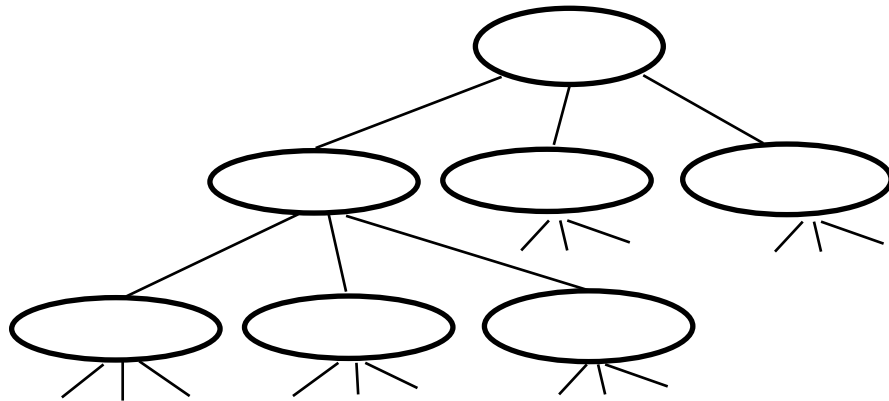
Programmieren

- Programm in einer Programmiersprache schreiben, testen, weiterentwickeln
- Verwendung von Elementen der Programmiersprache, bestimmte Regeln und Vorgehensweisen
- *Beispiel:* Objektorientierte Programmierung in Java



Dekomposition

- Eine Technik um Komplexität zu beherrschen (“divide and conquer”)



- Zwei Hauptarten von Dekomposition
 - Funktionale Dekomposition
 - Objekt-Orientierte Dekomposition

Funktionale vs. objekt-orientierte Dekomposition

- **Funktionale Dekomposition**
 - Das System wird unterteilt in Funktionen
 - Funktionen werden in kleinere Funktionen unterteilt
- **Objekt-orientierte Dekomposition**
 - Das System wird in Objekte unterteilt
 - Objekte in Klassen zusammengeführt
 - Objekte können wiederum in kleinere Objekte zerteilt



Welche Dekomposition ist besser?

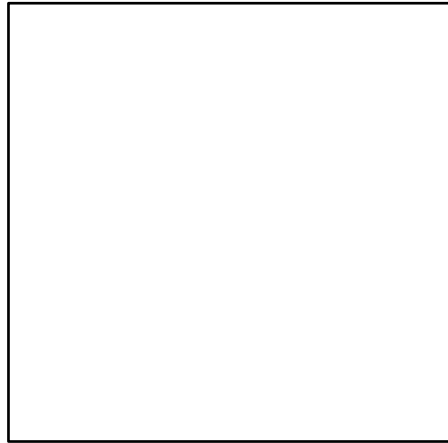
Funktionale Dekomposition

- Beispiel: Grafikprogramm
 - Wie kann ich ein Quadrat zu einem Kreis ändern?

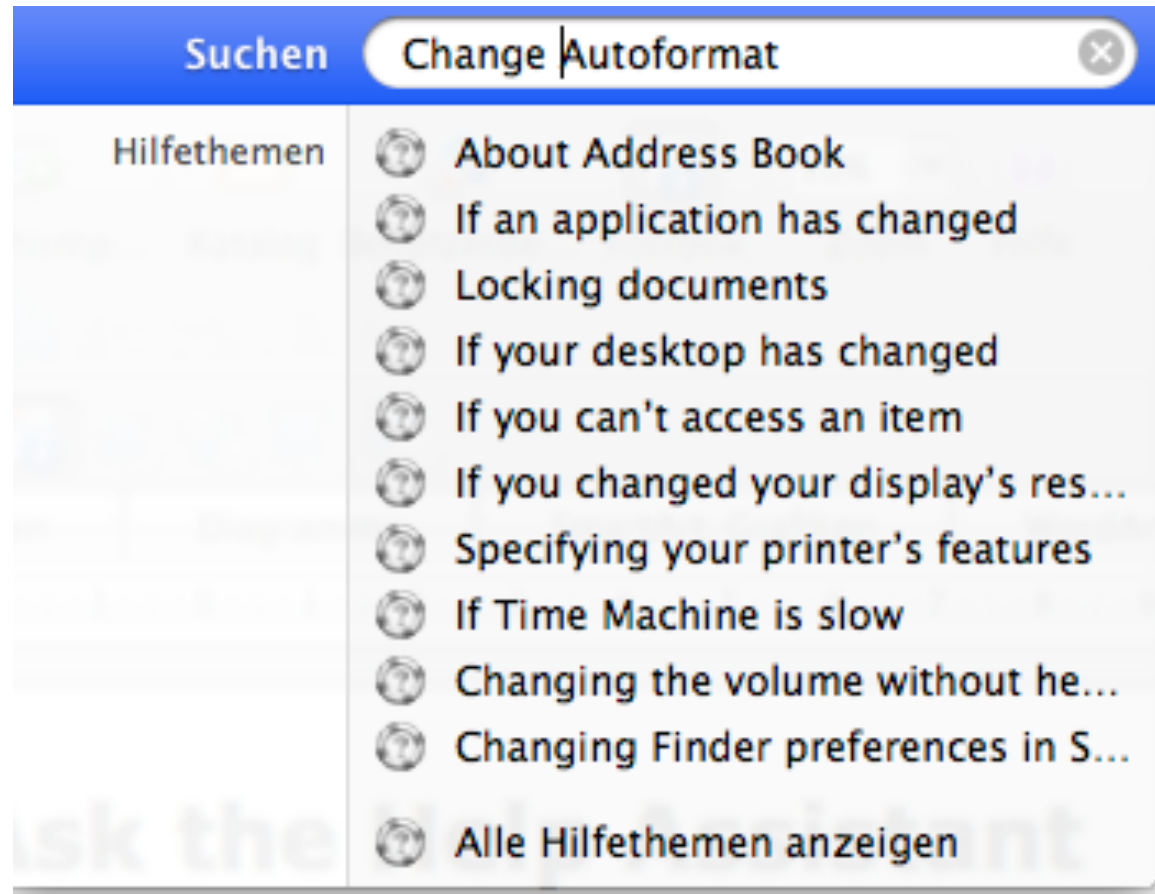


- Lassen Sie uns das in Microsoft Powerpoint ausprobieren.

1. Versuch: rechte Maustaste

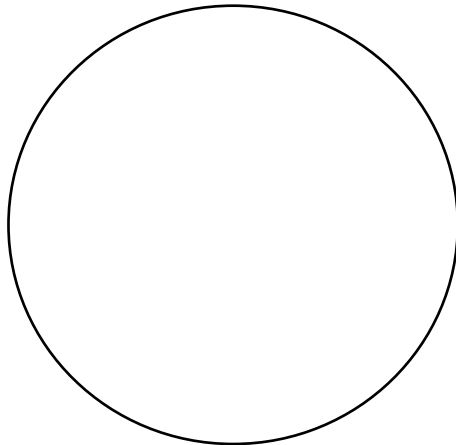


2. Versuch: Hilfe

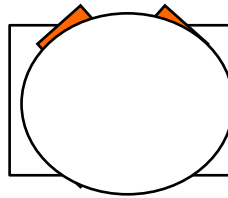


3. Versuch: Mit Hinweis...

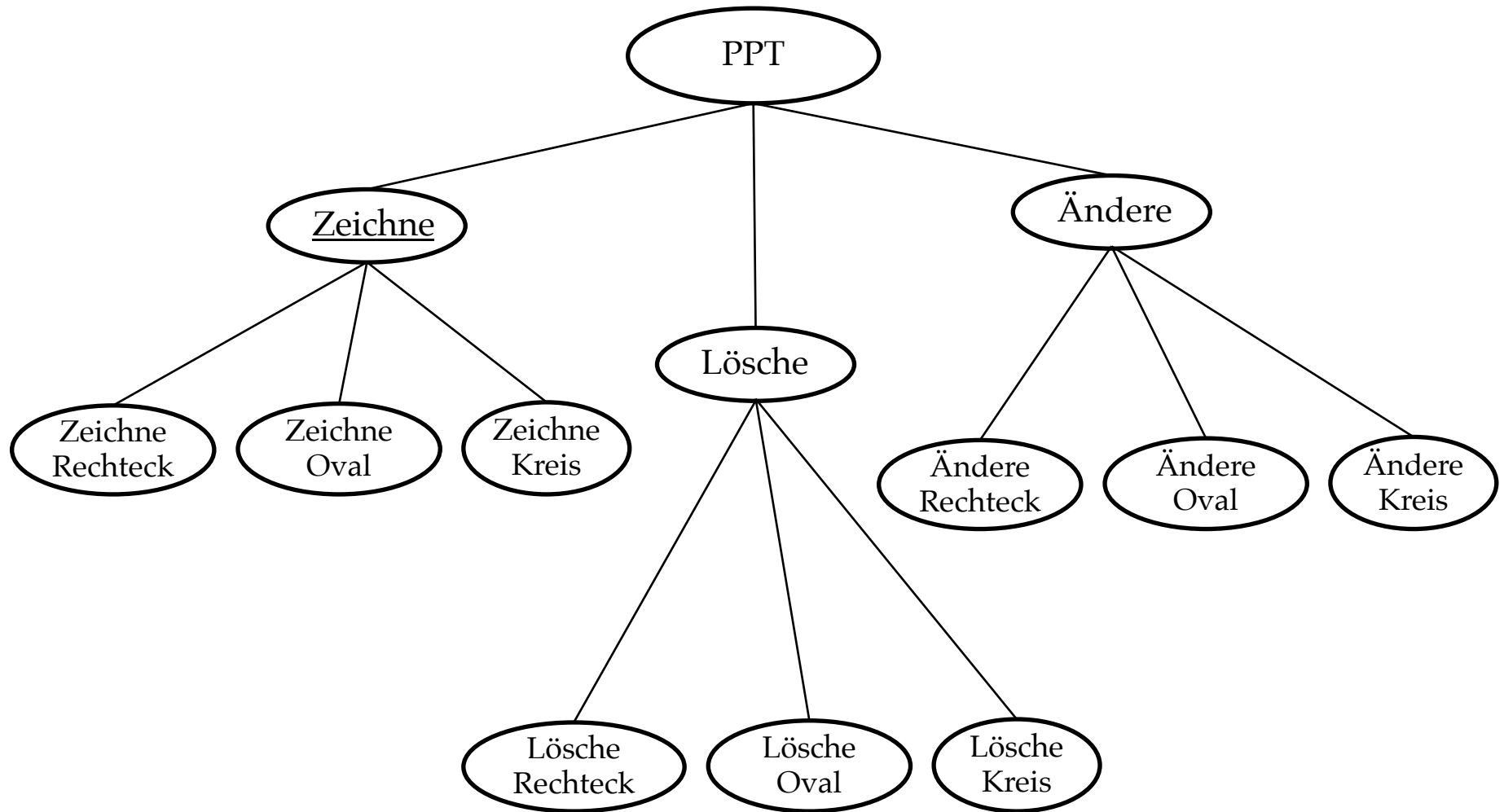
1. "Ansicht>Symbolleisten und Menüs anpassen":
2. Selektiere "Form Ändern"
3. Symbol zu Menubar hinzufügen
4. Klicke Rechteck
5. Klicke "Form Ändern"
6. Selektiere Kreis.



4. Versuch



Funktionale Dekomposition: PPT



Probleme der funktionalen Dekomposition

- Die Funktionalität ist verteilt über das ganze System
 - Source code schwer zu verstehen
 - User interface wird nicht intuitiv
- Konsequenz:
 - Ein Entwickler muss oft das ganze System verstehen um eine kleine Änderung durchzuführen

Objekt-Orientierte Sicht

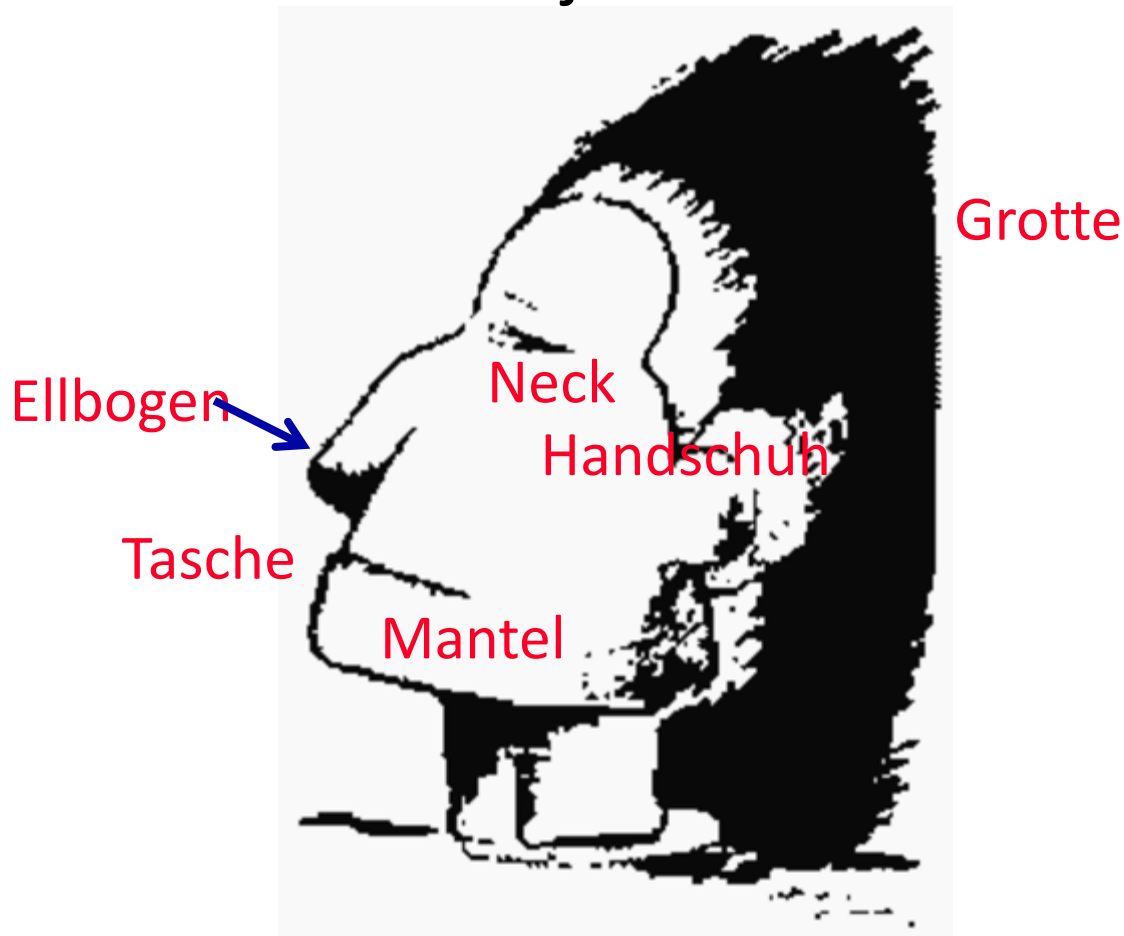
Form
zeichne() lösche() ändere() selektiere()

Ist das wirklich besser?



Was ist das?

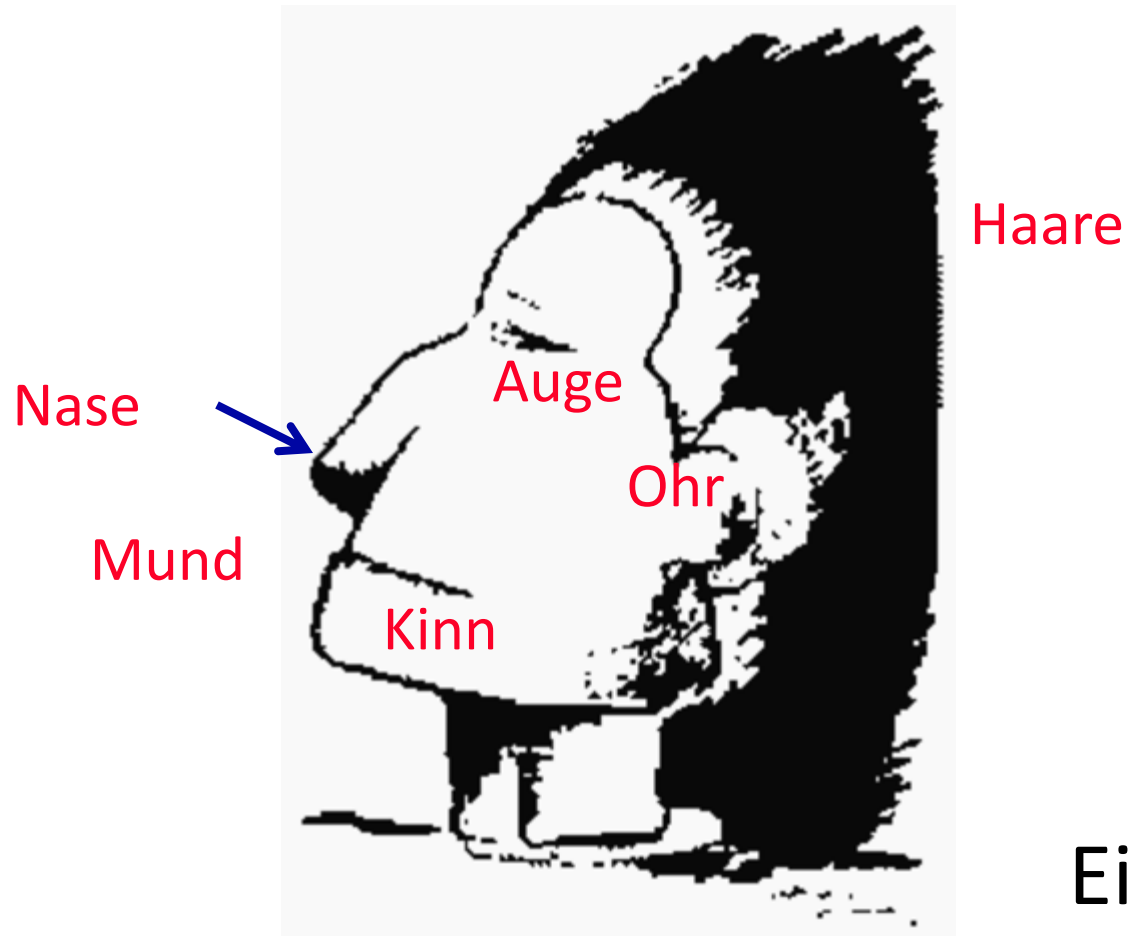
Versuchen wir es mit objekt-orientierter Dekomposition



ein Eskimo geht in die Grotte rein!

Was ist das?

Nochmals:



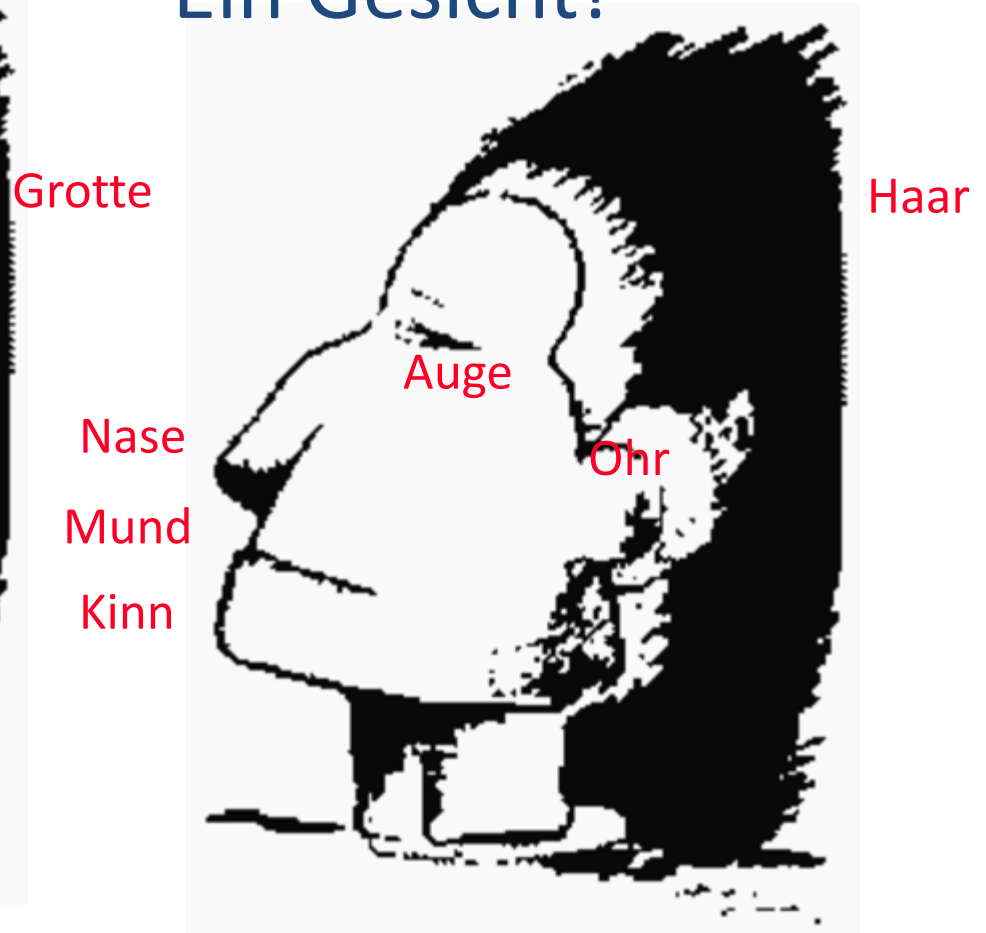
Ein Gesicht!

Zum Vergleich!

Ein Eskimo!



Ein Gesicht!



Identifikation von Objekten

- **Wir können Objekte finden**
 - Philosophie, Wissenschaft, Experimentieren
- **Einschränkung:**
 - Je nachdem was das Ziel des Systems ist, werden wir andere Objekte finden



Kritisch: Ziel des System zuert verstehen!

Was ist das Ziel dieses Systems?



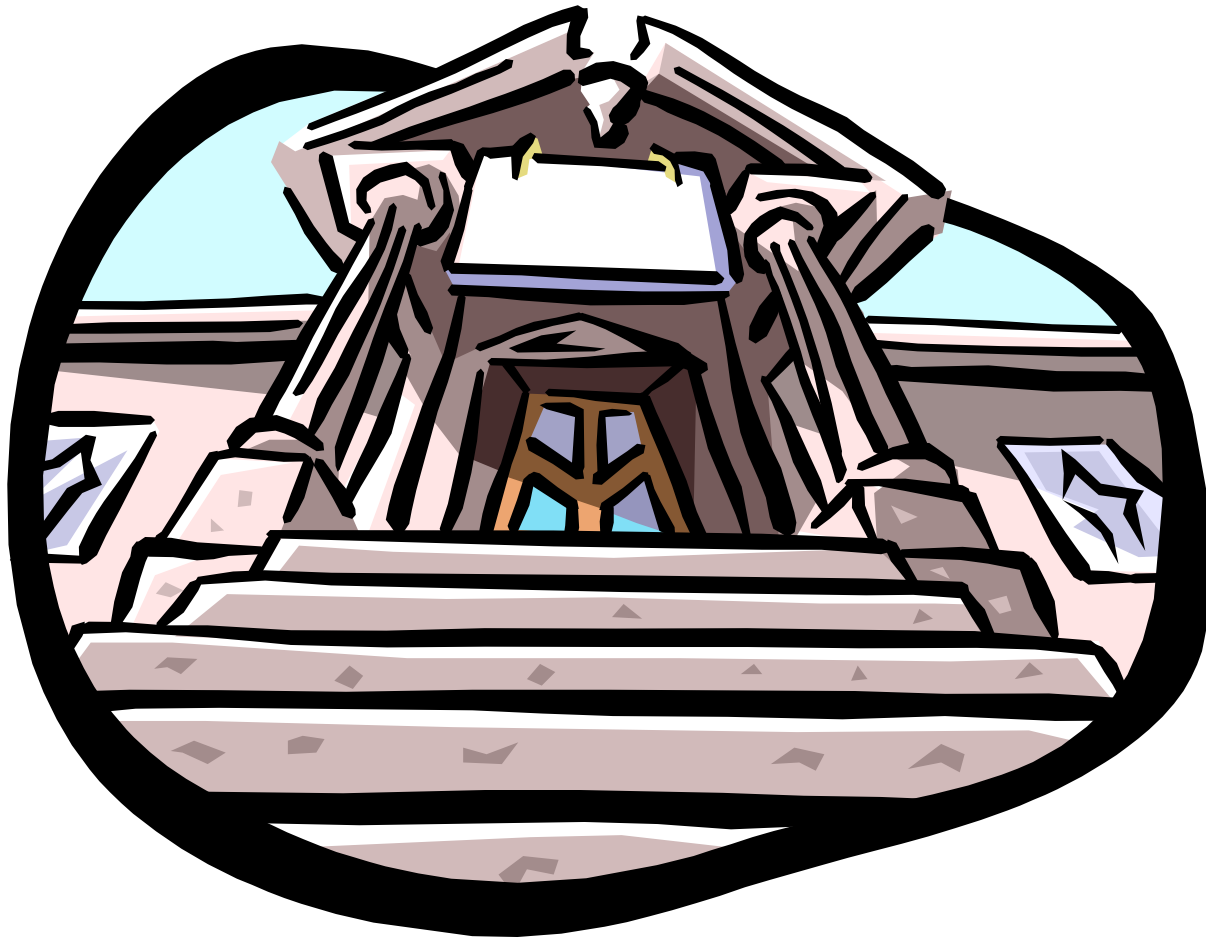
Und das?



Fortgeschrittene Version!



Beispiel: Bank



Kontoauszüge drucken

Der Bankkunde druckt sich am Bankautomat in der Schalterhalle seine Kontoauszüge aus.

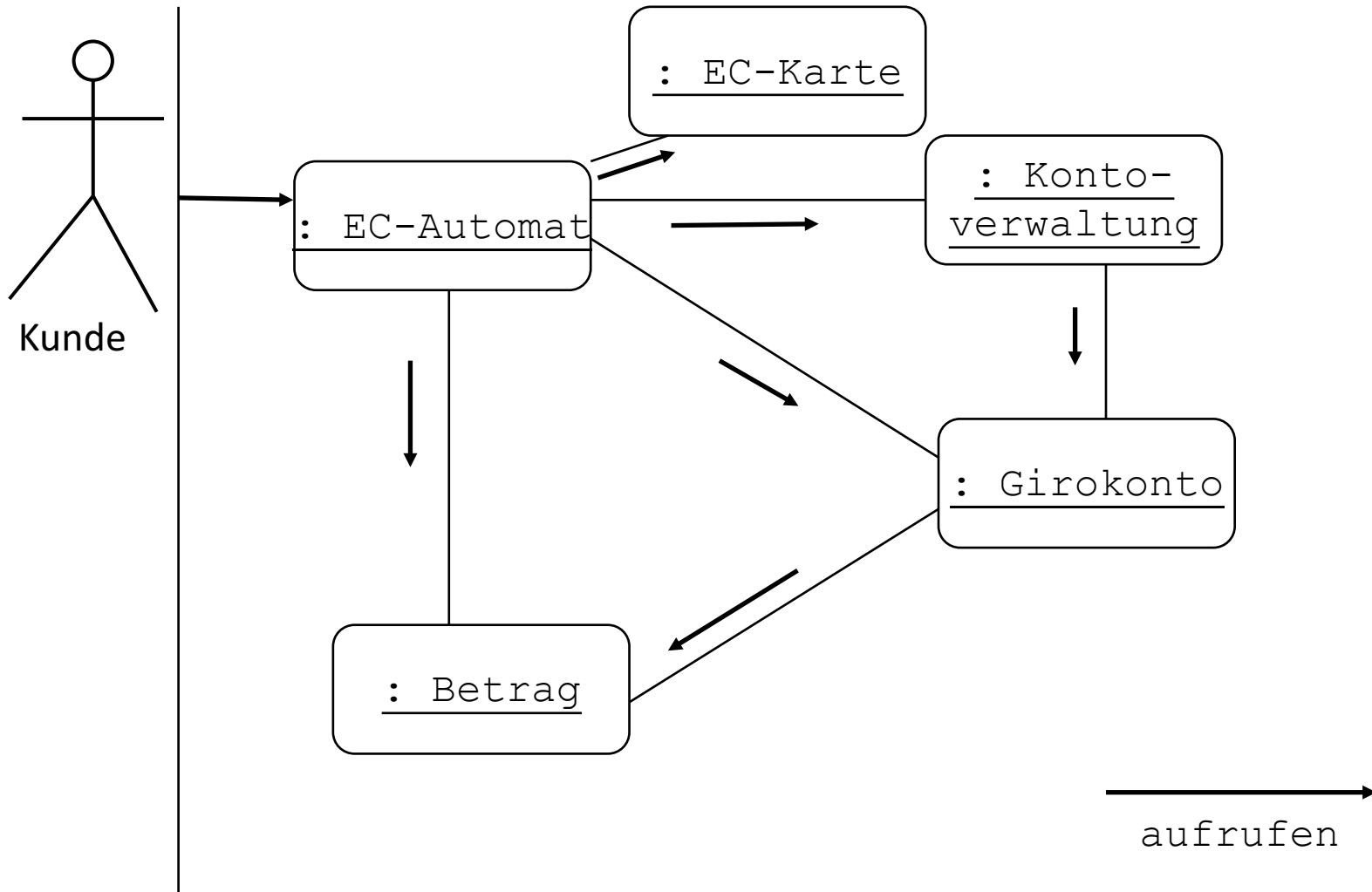


Geldauszahlung

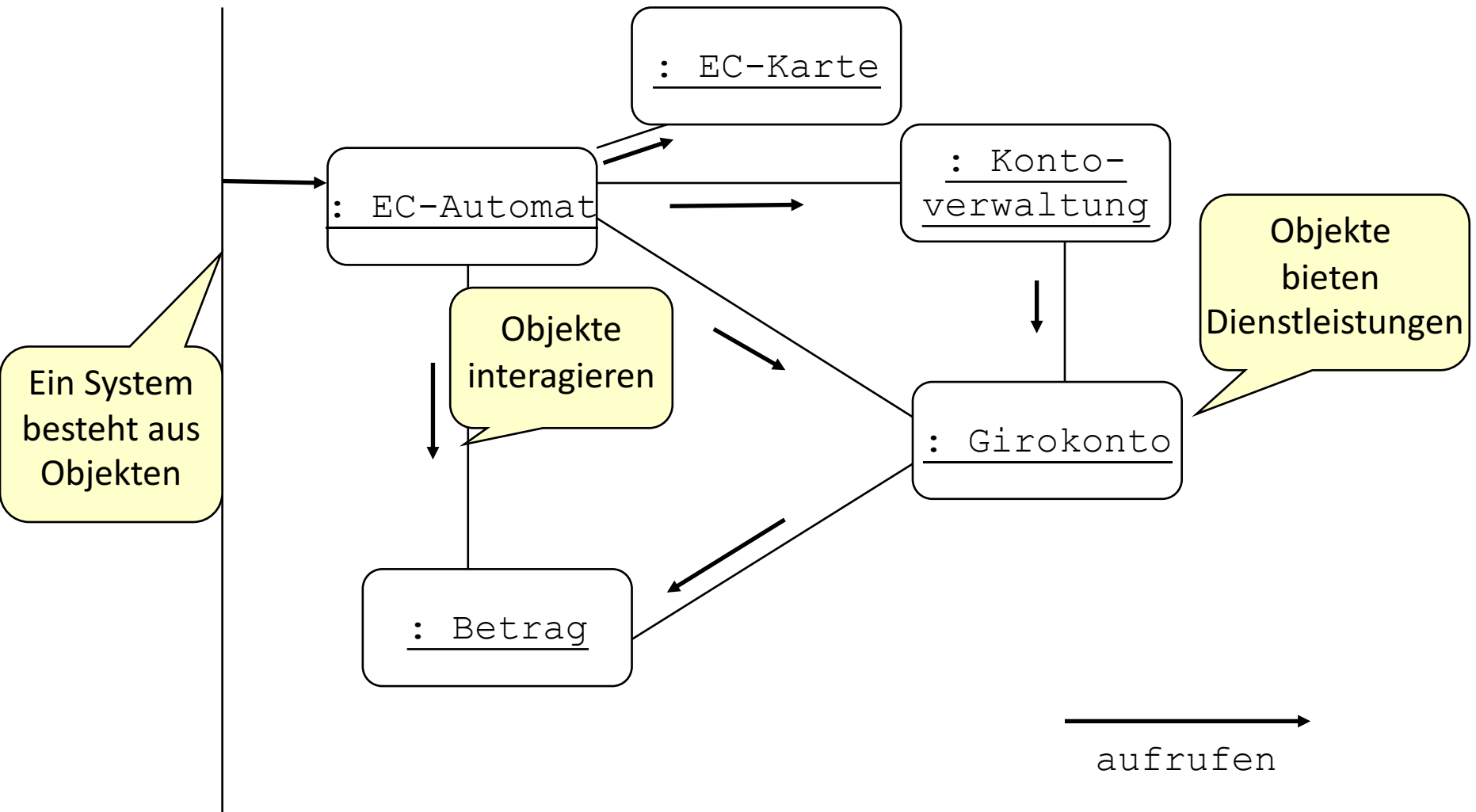
Dann lässt sich der Bankkunde mit seiner ec-Karte 300 EUR am Bankautomat von seinem Girokonto auszahlen.



Akteure interagieren mit einem System von Objekten



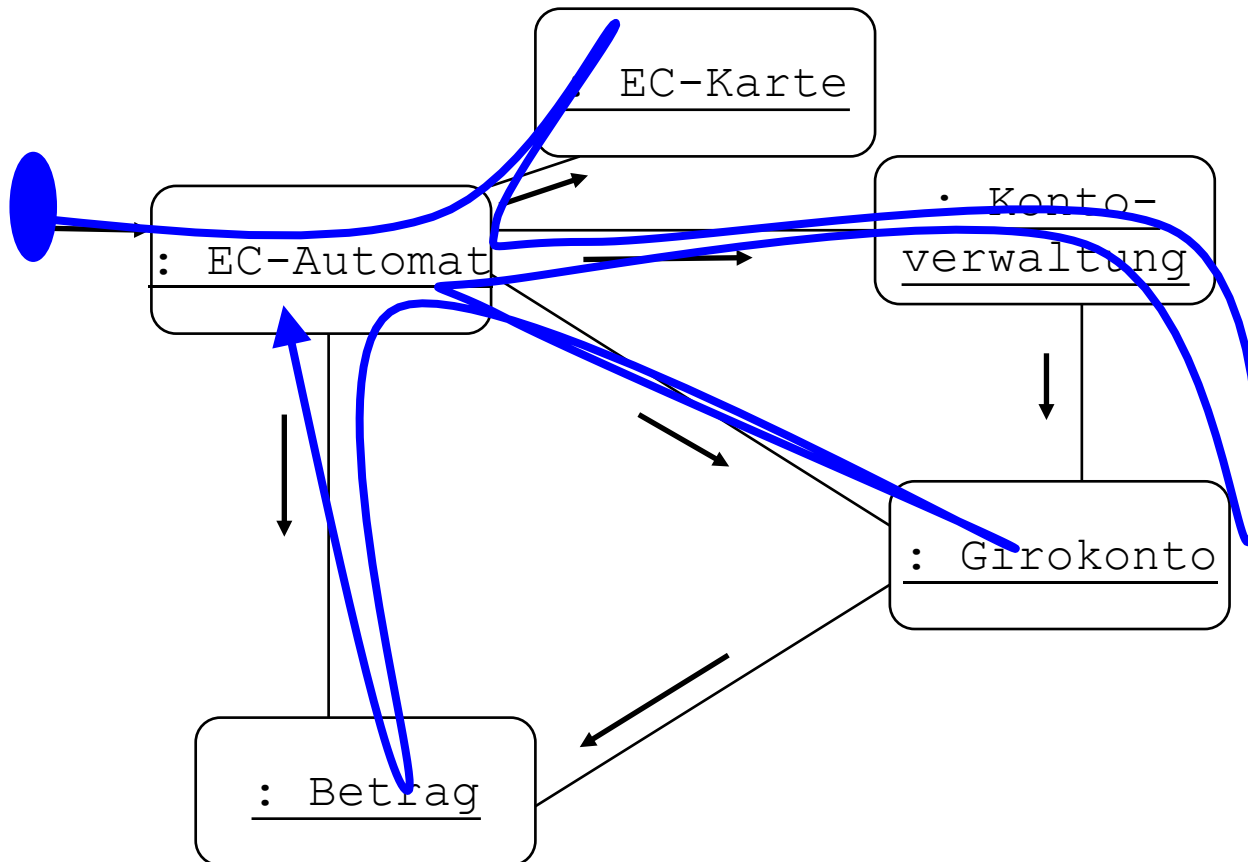
In Objektwelten laufen Prozesse auf Daten ab



Vereinfachung: Ein Prozess „durchläuft“ Objekte

Intuitives Verständnis

- Objekte können unabhängig von anderen Objekten aktiv sein
- Objekte können auf Anfragen von anderen Objekten warten oder parallel arbeiten



Vereinfachtes Modell

- Alle Objekte sind passiv
- Warten darauf, dass eine Dienstleistung angefordert wird
- Erbringen diese auf Anfrage
- Ansonsten untätig

Überblick

1

Objektorientierte Sichtweise

2

Grundbegriffe der Objektorientierung

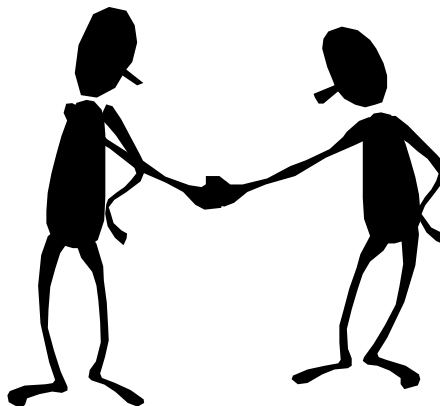
3

Aufbau von Klassendefinitionen

Dienstleister und Klienten

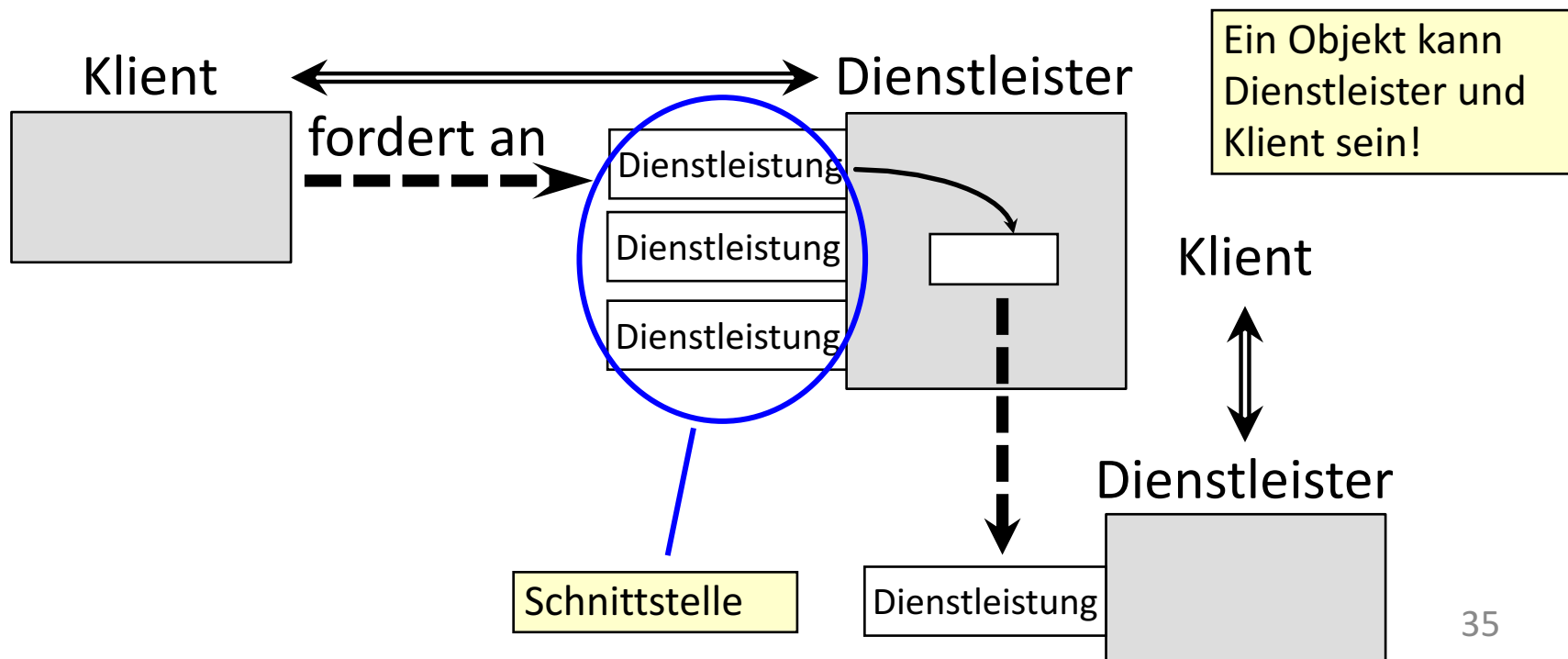


- Nimmt konkrete Dienstleistung eines anderen Objektes in Anspruch
- Leistet bei einer Teilaufgabe einen Dienst



Dienstleistungen an der Schnittstelle

- Objekte bieten Dienstleistungen als **Methoden** an ihrer **Schnittstelle** an
- Dienstleistungen werden von anderen Klienten benutzt
- Klient fordert eine Dienstleistung des Anbieters an
- Der Dienstleister kann selbst Teile seiner Dienstleistung von anderen Dienstleistern einholen

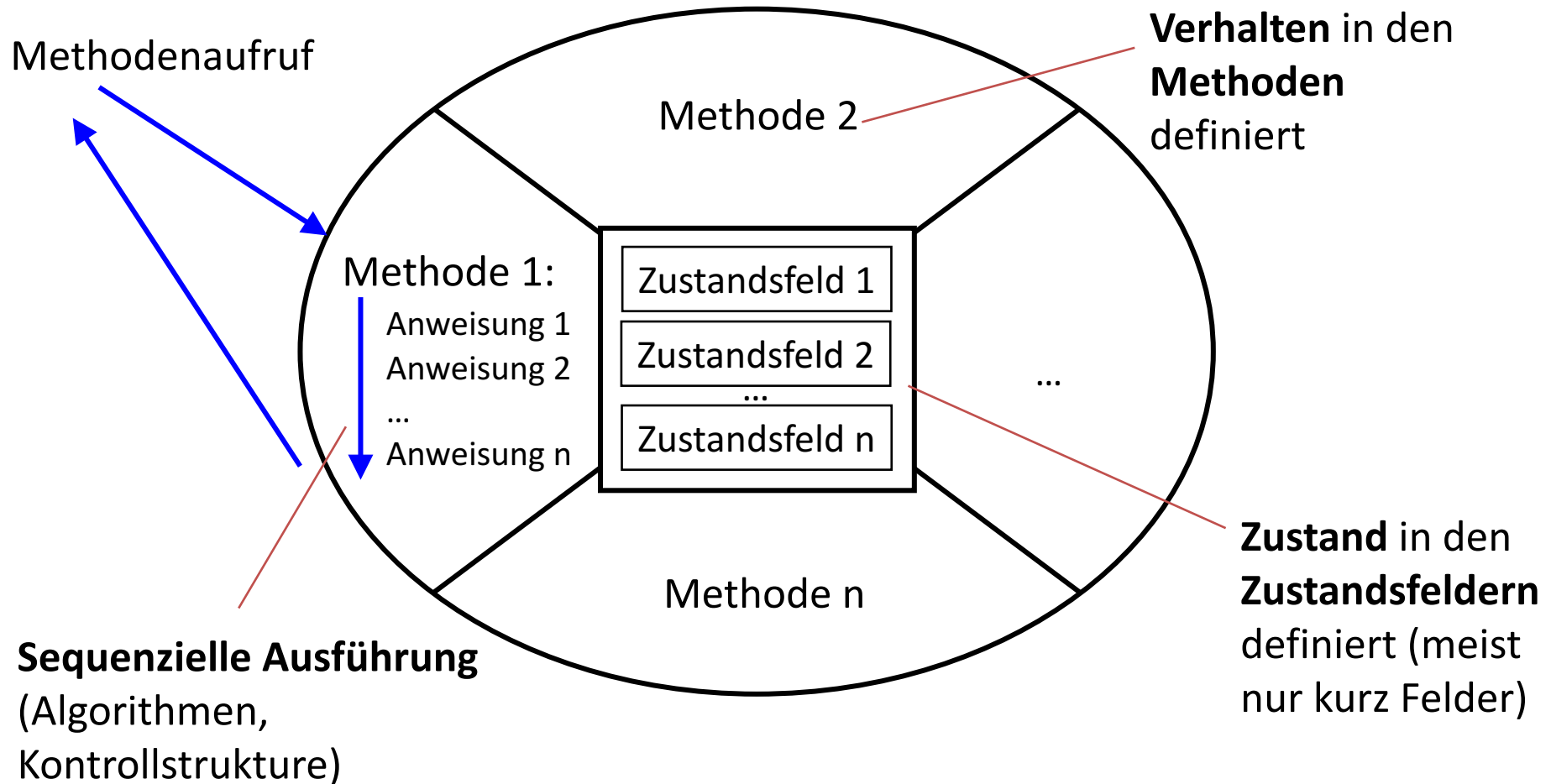


Dienstleistungen: Verhalten und Zustand

	<u>: Girokonto</u>
Zustand	<code>_dispo = 1000 EUR</code> <code>_saldo = 300 EUR</code>
Verhalten	<code>istAuszahlenMöglich(b:Betrag) : Boolean</code> <code>auszahlen(b:Betrag)</code>

- **Verhalten** eines Objekts ist durch seine angebotenen Dienstleistungen (d.h. seine **Methoden**) bestimmt
- Umsetzung dieser Dienstleistungen ist einem Klienten verborgen
- Ein Objekt kann einen **Zustand** haben und seine Dienstleistungen von diesem Zustand abhängig machen

Logische Sicht auf ein Objekt



Beispiel: Exemplar der Klasse Zeichner

Klient ruft
zeichne auf

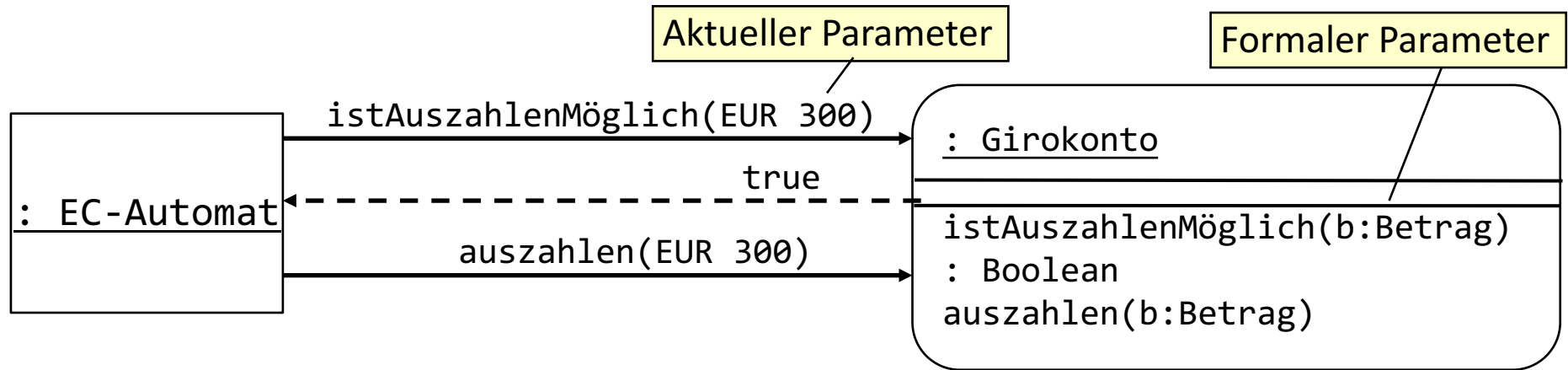
Verhalten: in der
einzigen Methode
definiert

```
void zeichne():  
    wand = new Quadrat();  
    ...
```

keine Felder,
kein Zustand!

Sequenzielle Ausführung
der Anweisungen im
Rumpf

Objekte interagieren über Methodenaufrufe



- Objekte (Klienten) rufen **Methoden** an anderen Objekten (Dienstleistern) auf
- Ein **Methodenaufruf** kann **parametrisiert** werden; der Klient gibt beim Aufruf konkrete Werte als **aktuelle Parameter** an; der Dienstleister arbeitet dann auf Kopien dieser Parameter, die für ihn **formale Parameter** genannt werden
- Der Dienstleister kann nach dem Ende einer Methodenausführung ein **Ergebnis** an den Klienten zurückgeben

Signatur einer Methode

- **Signatur** einer Methode liefert die **für einen Klienten** relevanten Informationen für einen **Methodenaufruf**. In Java umfasst dies:
 - **Name der Methode**
 - **Anzahl, Reihenfolge und Typen der Parameter**
- Beschreibung einer Methode zusätzlich weitere Informationen angeben:
 - Parameternamen
 - Ergebnistyp
 - Methodenkommentar
- Diese Informationen sind in Java **formal nicht Teil der Signatur**.
- Beispiel in Java:

Methode: `boolean istAuszahlenMöglich(Betrag b)`

Signatur: `istAuszahlenMöglich(Betrag)`



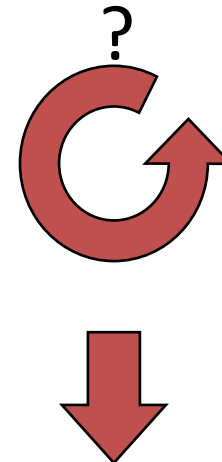
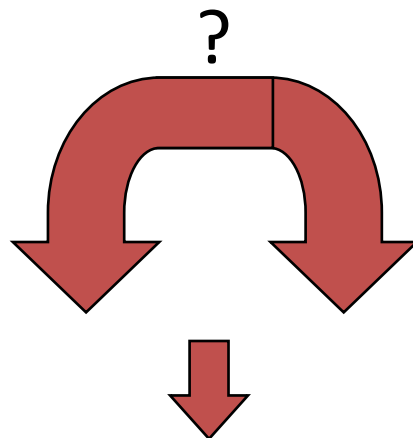
Klassen als Schablonen für Exemplare

Girokonto
<code>_dispo : Betrag</code> <code>_saldo : Betrag</code>
<code>istAuszahlenMöglich(b:Betrag) : Boolean</code> <code>auszahlen(b:Betrag)</code>

- **Exemplare** sind die **Objekte**, die aus **Klassen** heraus erzeugt werden
- Eine Klasse definiert somit das **prinzipielle Verhalten** ihrer Exemplare
- Von einer Klasse können **beliebig viele** Exemplare erzeugt werden
- Jedes Exemplar hat einen **eigenen veränderbaren Zustand**
- Dadurch können Exemplare auf dieselbe Anfrage anders reagieren

Ablaufsteuerung durch Kontrollstrukturen


- Ausführungsreihenfolge einer Methode entspricht zunächst der textlichen Anordnung (**Sequenz**)
- Davon kann aber abgewichen werden. Dazu gibt es spezielle Mechanismen der Ablaufsteuerung:
 - **Fallunterscheidung: if else**
 - **Wiederholung: for while**



Kontrollstruktur 1: Sequenz

- Anweisung werden nacheinander verarbeitet
- Anweisungen sind voneinander getrennt
- Anweisung kann auch die leere Aktion sein („tue nichts“)

Informeller Algorithmus Telefonieren:

 hebe den Hörer ab;
wähle die Telefonnummer;
führe das Gespräch;
lege den Hörer auf.

```
...  
int i = 4;  
int j = 5;  
int k = 6;  
...
```



Kontrollstruktur 2: Fallunterscheidung

- Bedingte Anweisungsfolge
- Das Grundschema der Fallunterscheidung ist:
 - WENN ... DANN ... SONST ... ENDE (*WENN*)

Informeller Algorithmus Telefonieren:

hebe den Hörer ab;

WENN Telefonnummer gespeichert

DANN drücke Kurzwahltaste

SONST wähle die

Telefonnummer

ENDE (*WENN*)

WENN Gesprächspartner antwortet

DANN führe das Gespräch

ENDE (*WENN*)

lege den Hörer auf.

```
if (a < b)
{
    min = a;
}
else
{
    min = b;
}
```



Kontrollstruktur 3: Wiederholung

- Der Mechanismus zur **Wiederholung** von Anweisungen (Schleife):
 - Anweisungsfolgen werden wiederholt ausgeführt
 - Das Ende der Wiederholung ist mit einer **logischen Bedingung** verknüpft.
 - Wir unterscheiden konzeptionell:
 - "Solange-Noch"-Schleifen: SOLANGE ... WIEDERHOLE ... ENDE,
 - "Solange-Bis"-Schleifen: WIEDERHOLE ... BIS

Aus dem Algorithmus Telefonieren:

SOLANGE Geld da

WIEDERHOLE

hebe den Hörer ab;

wirf Geld ein;

führe Gespräch;

lege den Hörer auf;

ENDE .

Aus dem Algorithmus Telefonieren:

hole Liste der

Gesprächspartner

WIEDERHOLE

führe ein Gespräch;

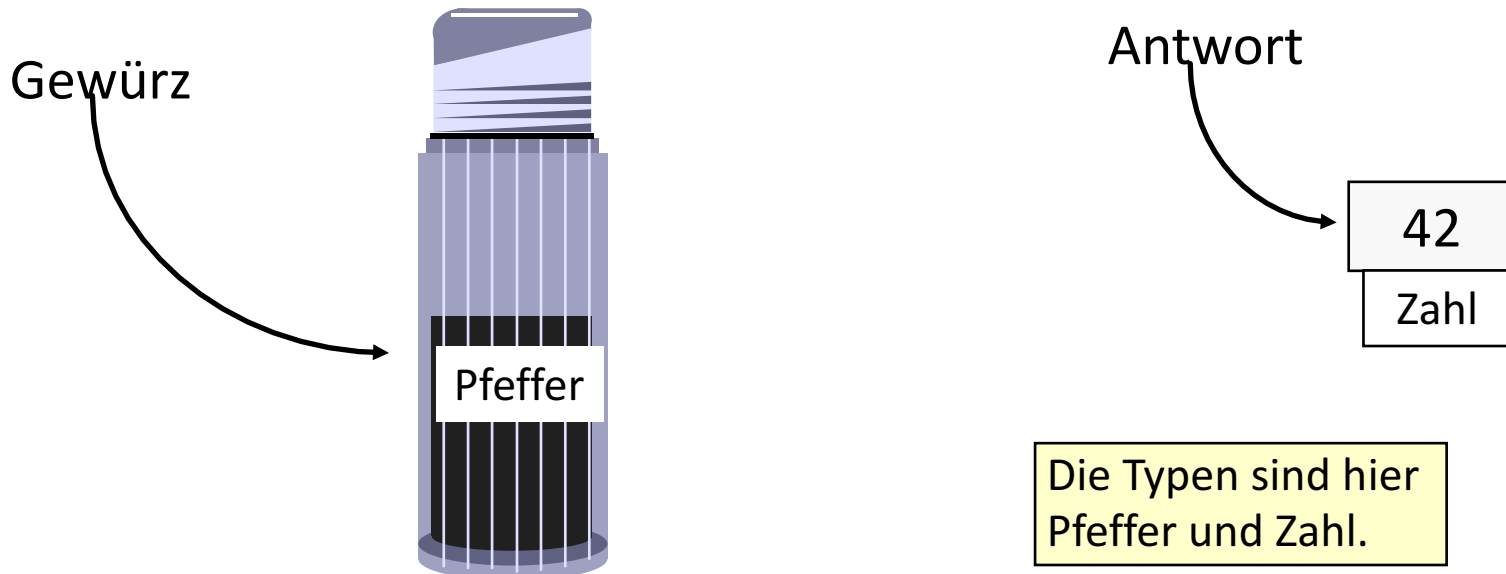
streiche

Gesprächspartner;

BIS Liste abgehakt .

Variablen

- Abstraktion eines physischen **Speicherplatzes**
- Benutzung durch den **Namen** (auch: Bezeichner)
- **Variable** hat den Charakter eines Behälters:
 - Belegung (aktueller Inhalt) kann sich **ändern**
 - **Typ** legt Wertebereich, zulässige Operationen und
 - weitere Eigenschaften fest



Deklaration und Initialisierung

- **Variablen** werden vor der Verwendung bekanntgemacht, d.h. **deklariert**
- Vereinfacht geschieht dies durch:
 - Angabe des **Typs**
 - Vergabe eines **Namens** über einen **Bezeichner** (engl.: identifier)
- Durch die reine Deklaration von Variablen ist deren **Belegung** zunächst meist **undefiniert**
- Erst bei der **Initialisierung** wird eine Variable erstmalig mit einem gültigen Wert befüllt

Deklaration

```
int i;  
boolean b;
```



Deklaration und Initialisierung

```
int i = 42;  
boolean b;  
  
b = true;
```

Zwischenfazit: Objektorientierung ist ein Paradigma

- „Sicht der Welt“, die uns hilft, einen Sachverhalt zu interpretieren und zu verstehen



Zwischenfazit

1 Objekte haben einen **Zustand** und bieten Dienstleistungen an. Der Zustand wird durch Zustandsfelder realisiert.

2 Die für Klienten aufrufbaren **Methoden** eines Objektes bilden seine **Schnittstelle**.

3 Anweisungen in einer Methode werden nach den Prinzipien der **Kontrollstrukturen** ausgeführt.

4 **Variablen**, können dynamisch ihre Belegung ändern.

Überblick

1

Objektorientierte Sichtweise

2

Grundbegriffe der Objektorientierung

3

Aufbau von Klassendefinitionen

Erste Klassendefinition

```
class Girokonto
{
    private int _saldo;

    public void einzahlen( int betrag )
    {
        _saldo = _saldo + betrag;
    }
}
```



- Ein Java-Programm besteht aus **Textdateien**
- Jede Textdatei beschreibt eine **Klasse**
- **Klassendefinition** ist die textuelle Beschreibung einer Klasse
- Die Klassendefinitionen werden mit einem **Editor** bearbeitet

Merkmale unserer ersten Klasse

```
class Girokonto
{
    private int _saldo;

    public void einzahlen( int betrag )
    {
        _saldo = _saldo + betrag;
    }
}
```



- Java-Programme bestehen aus **Klassen** (hier: Girokonto)
- Klasse definiert eine **Methode** (hier: einzahlen)
- Methode erhält einen Parameter (hier: betrag vom Typ **int**)
- Keinen Rückgabewert (hier: Schlüsselwort **void**)
- Im Rumpf der Methode wird ein Wert einem **Zustandsfeld** zugewiesen (hier: _saldo)
- Feld muss **deklariert** sein (hier vom Typ **int**)
- Alternativ nennen wir die Felder in einer Klassendefinition auch **Exemplarvariablen**

Abgleich mit den Prinzipien der Objektorientierung

- Verhalten eines Objekts ist durch seine angebotenen Dienstleistungen (Methoden) bestimmt
- **einzahlen** ist durch **public** für Klienten aufrufbar
- Realisierung dieser (zusammengehörigen) Dienstleistungen ist verborgen
- Kein Zugriff durch Klienten auf die Implementierung von **einzahlen**
- Zustandsfelder sind als interne Strukturen eines Objekts gekapselt
- Das **Feld _saldo** ist durch **private** vor externem Zugriff geschützt
- Auf den Zustand eines Objektes kann nur über seine Dienstleistungen zugegriffen werden
- Hier durch **einzahlen**

```
class Girokonto
{
    private int _saldo;

    public void einzahlen( int betrag )
    {
        _saldo = _saldo + betrag;
    }
}
```



Auswertung: Grobstruktur einer Klassendefinition

```
/**
 * Schnittstellenkommentar der Klasse
 */
class Girokonto
{
    private int _saldo;

    public void einzahlen( int betrag )
    {
        _saldo = _saldo + betrag;
    }
}
```

Kopf der Klasse

Rumpf der Klasse



Klassenkopf: spezifiziert den Namen der Klasse und beschreibt mit dem Schnittstellenkommentar die Aufgabe der Klasse.

Klassenrumpf: beinhaltet Zustandsfelder, Konstruktoren und Methoden, die die Zuständigkeiten der Klasse realisieren.

Auswertung: allgemeine Struktur einer Klassendefinition

```
class Girokonto
{
    private int _saldo;
```

Konstruktor kann fehlen
(Standardkonstruktor)

```
    public void einzahlen( int betrag )
    {
        _saldo = _saldo + betrag;
    }
}
```



```
class <Klassename>
{
    <Felder>
```

```
<Konstruktoren>
```

```
<Methoden>
```

```
}
```

Klassendefinition mit explizitem Konstruktor

```
class Girokonto
{
    private int _saldo;

    public Girokonto()
    {
        _saldo = 0;
    }

    public void einzahlen( int betrag )
    {
        _saldo = _saldo + betrag;
    }
}
```

Konstruktor kann explizit angegeben werden, heißt immer wie die Klasse

```
class <Klassenname>
{
    <Felder>

    <Konstruktoren>

    <Methoden>
}
```



Objekte erzeugen

- Objekte werden zur Laufzeit erzeugt
- Durch einen expliziten Ausdruck mit einem **Schlüsselwort**
- In Java wird das **Schlüsselwort new** verwendet

```
class Zeichner {  
    ...  
    Quadrat wand = new Quadrat();  
    Dreieck dach = new Dreieck();  
    Quadrat fenster = new Quadrat();  
    ...  
    wand.vertikalBewegen(80);  
    fenster.farbeAendern("blau");  
    dach.horizontalBewegen(70);  
    ...  
}
```

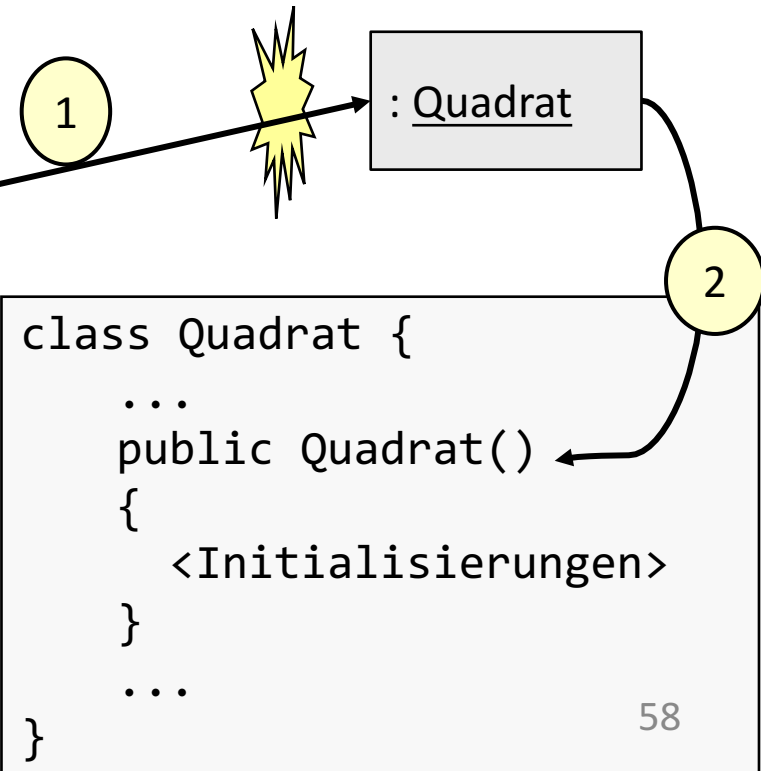


Schlüsselwort: Zeichenfolge, die in einer Programmiersprache eine feste Bedeutung hat (z.B. if). Schlüsselwörter sind (meist) reserviert, d.h. sie dürfen nicht als Namen von Variablen verwendet werden.

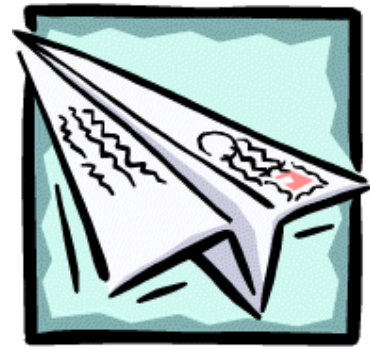
Konstruktoraufruf und Konstruktor

- Ein Konstruktoraufruf (in Java mit **new**) bewirkt zweierlei:
 - 1 Ein neues Objekt der genannten Klasse wird erzeugt
 - 2 Bei diesem Objekt wird der angegebene Konstruktor ausgeführt; ein Konstruktor initialisiert ein neu erzeugtes Objekt

```
class Zeichner {  
    ...  
    Quadrat wand = new Quadrat();  
    Dreieck dach = new Dreieck();  
    Quadrat fenster = new Quadrat();  
    ...  
    wand.vertikalBewegen(80);  
    fenster.farbeAendern("blau");  
    dach.horizontalBewegen(70);  
    ...  
}
```



Methoden aufrufen



- Methodenaufruf richtet sich an ein bestimmtes Objekt, den Adressaten des Aufrufs
- Der Adressat ist entweder explizit angegeben:

```
wand.vertikalBewegen(80)
```

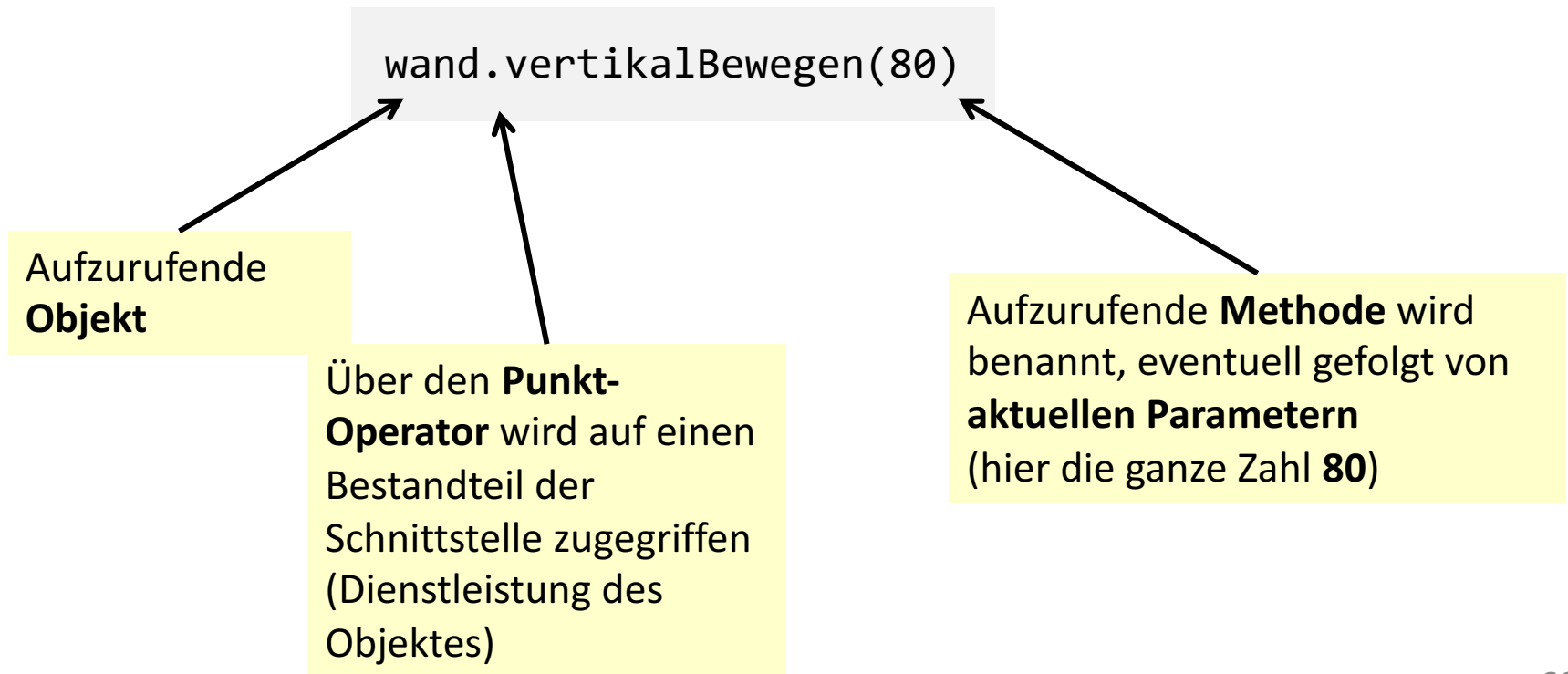
- Gerufene Methode ist üblicherweise Teil der Schnittstelle des gerufenen Objektes
- Oder es wird eine Methode des aktuellen Objektes aufgerufen:

```
zeichneDach(80)
```

- Hilfsmethoden, die nur innerhalb einer Klasse verwendet werden, werden **private** deklariert

Punktnotation

Methoden eines Objekts werden in vielen objektorientierten Sprachen mit der Punktnotation aufgerufen (engl.: dot notation)



Punktnotation in Java

Java folgt der objektorientierten Tradition und verwendet ebenfalls die Punktnotation für Methodenaufrufe an Objekten

wand.vertikalBewegen(80)



Referenz auf ein Objekt

der Punkt

Aktuellen Parameter werden in Java in **runden Klammern** eingeschlossen. Auch wenn keine Parameter definiert sind, werden bei Methodenaufrufen die **runden Klammern immer** angegeben.

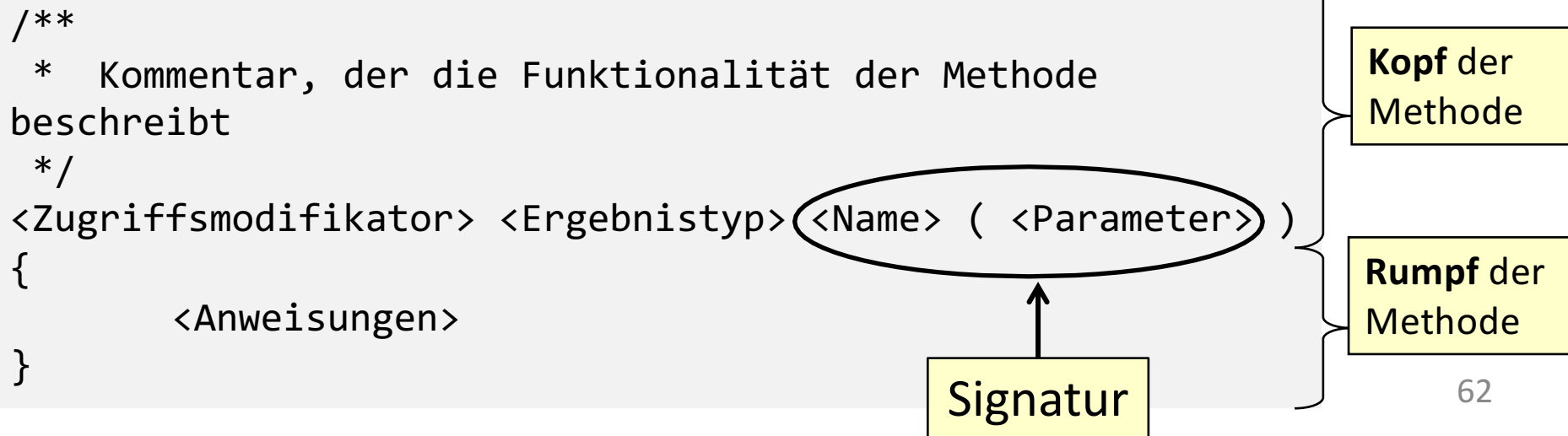
Struktur der Methodendefinition in Java

Methodenköpfe

- Klassen spezifizieren mit den Köpfen ihrer öffentlichen Methoden Dienstleistungen
- Legen fest, wie die Zustände der Objekte sondiert oder verändert werden
- Öffentlichen Methoden bilden die **Schnittstelle** einer Klasse

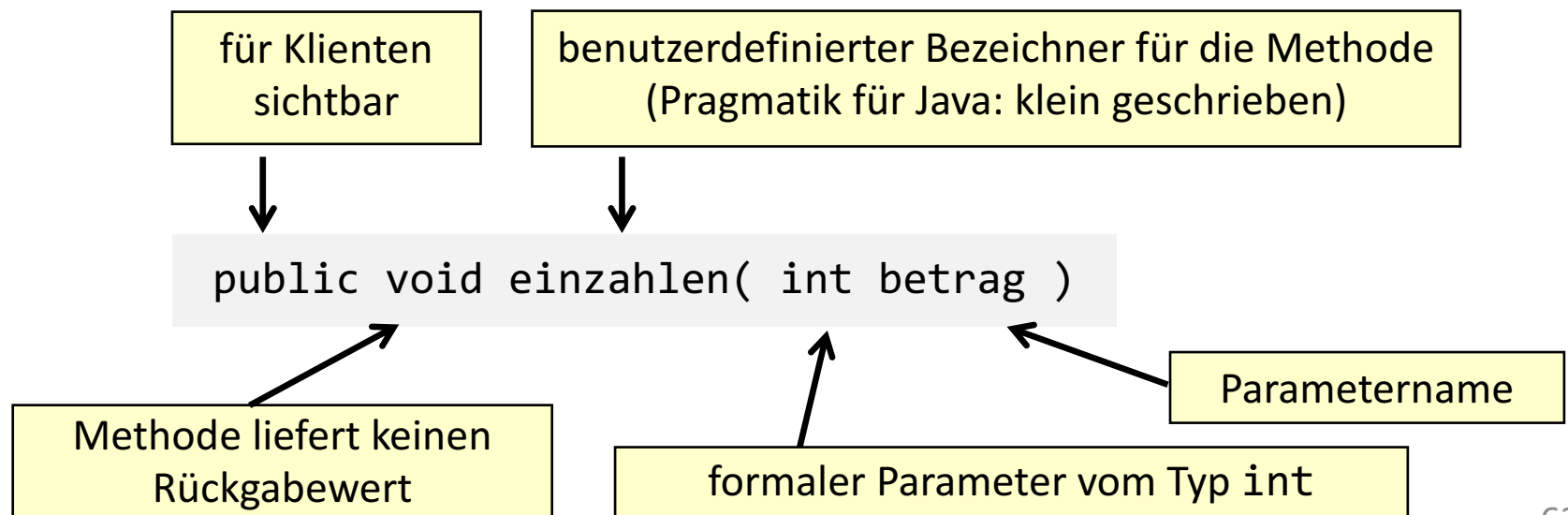
Methodenrumpfe

- Realisieren die versprochenen Dienstleistungen durch eine Implementierung
- Schnittstelle (dem „Kopf“) und der Implementierung (dem „Rumpf“) einer Methode sind strukturell getrennt



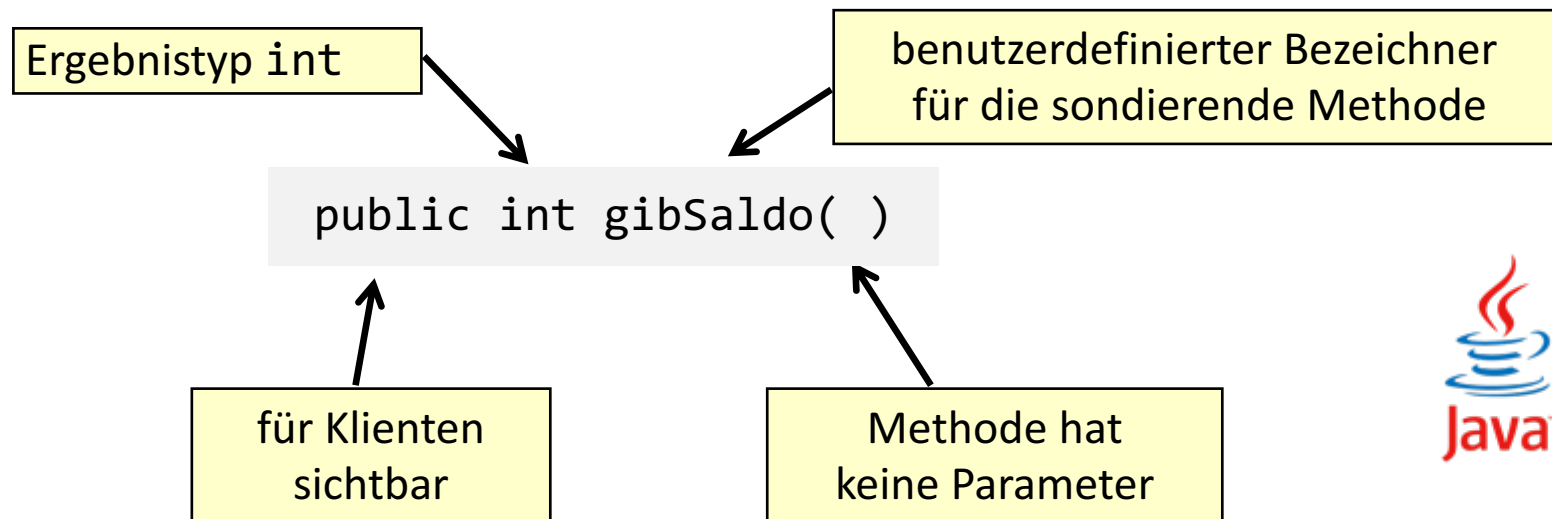
Verändernde Methoden

- Bei Veränderung geben verändernde Methoden keinen Wert zurück (engl.: mutators)
- Für Klienten sind nur die Methoden aufrufbar, die mit **public** als „öffentlich“ deklariert wurden; sie bilden die **Schnittstelle** einer Klasse
- Zur Implementierung werden oft interne Methoden verwendet
- Sie werden in Java als **private** deklariert



Sondierende Methoden

- Verändern den Zustand eines Objektes nicht (engl.: accessor methods)
- Liefern einen (Ergebnis-) Wert von einem vereinbarten (Ergebnis-) Typ
- Ergebnis wird explizit mittels der **return** Anweisung zurückgegeben
- Können deshalb an der Aufrufstelle als Teil von Ausdrücken verwendet werden



Zusammenfassung

1

Klassendefinitionen beschreiben Klassen.

2

Wir erzeugen Objekte durch **Konstruktoraufrufe**. Ein Konstruktor initialisiert den Zustand eines Objektes.

3

Die (Zustands-)Felder eines Objektes halten seinen Zustand. Die Definitionen der Felder sind **Exemplarvariablen**.

4

Eine **Methode** besteht aus einem **Kopf** und einem **Rumpf**. Es gibt **sondierende** (lesende) und **verändernde** Methoden.