
Continental
The Future in Motion
C1TT
No & Low Code
C1TT
Continental Institut
für Technologie
und Transformation

C1TT Continental Institut für Technologie und Transformation

Objektorientierte Softwareentwicklung

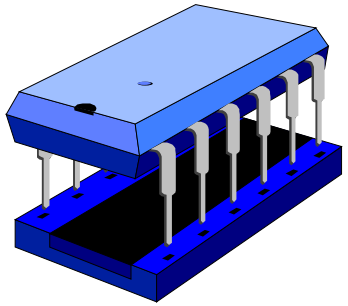
Einstieg in die Praxis von Analyse, Design und Implementierung

1

GRUNDLAGEN DER OBJEKTORIENTIERUNG

Objektorientierung

- kommt ursprünglich aus der Programmierung (OOP),
- ist ein Denkansatz (Paradigma),
- bedeutet für die Analyse: “Objekte” sollen die Realität möglichst natürlich abbilden, ohne künstlich zu zerlegen.
- bedeutet für die Realisierung: Objektorientierung beschreibt, wie wieder verwendbare Software-Bausteine (Software-IC) aussehen sollen



Was ist objektorientierte Systementwicklung?

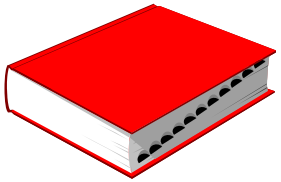
- Softwaresystem besteht aus Objekten, die
  - Dienste bereitstellen und
  - Dienste anderer Objekte benutzen
- Analyse:
  - Begriffe des Anwendungsbereichs verstehen
  - Objektmodell für Begriffe konstruieren
- Design:
  - Entscheidungen über technische Umsetzung
  - Erweiterung, Verfeinerung des Objektmodells
- Programmierung:
  - Objektstruktur und Dienste implementieren

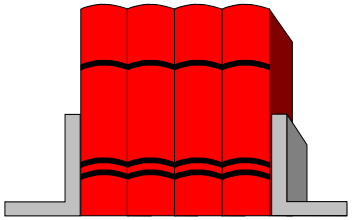
Wiederverwendbarkeit und Modularisierung

Was ist ein Objekt?

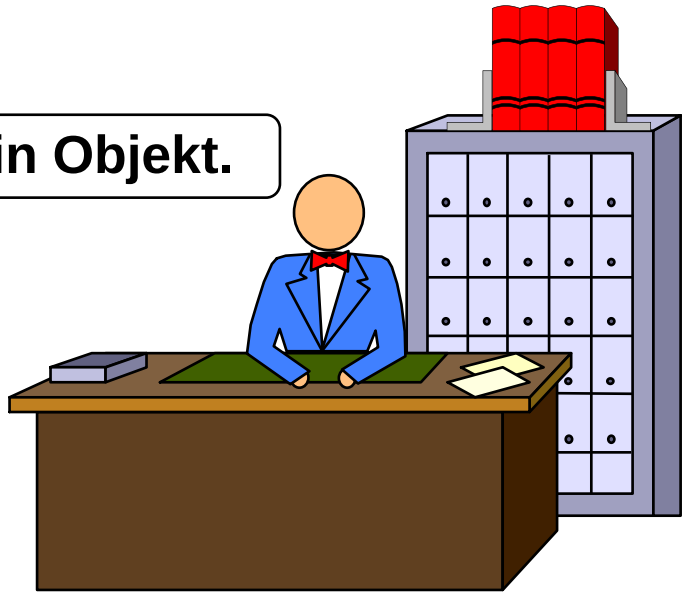
Meine Katze ist ein Objekt.

- Das Buch “Mit Objekten auf Du und Du”
- Der Bibliotheksbenutzer “Müller”
- Das Buchregal “2712”
- ...



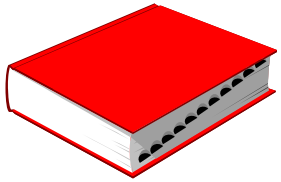


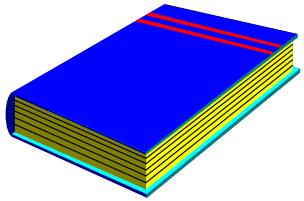
Alles ist ein Objekt.

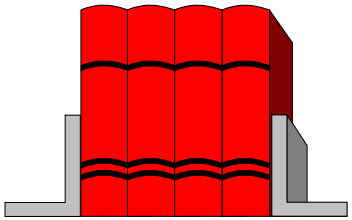


Objekte

- Objekte sind:
  - Gegenstände,
  - Geräte,
  - Ereignisse,
  - Strukturen,
  - Rollen,
  - Örtlichkeiten,
  - alles, wovon man sich einen Begriff machen kann

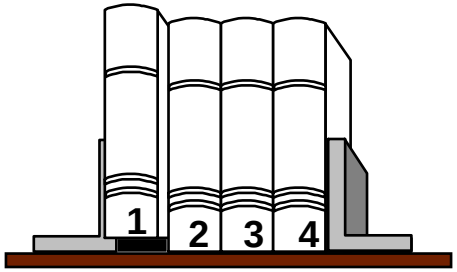




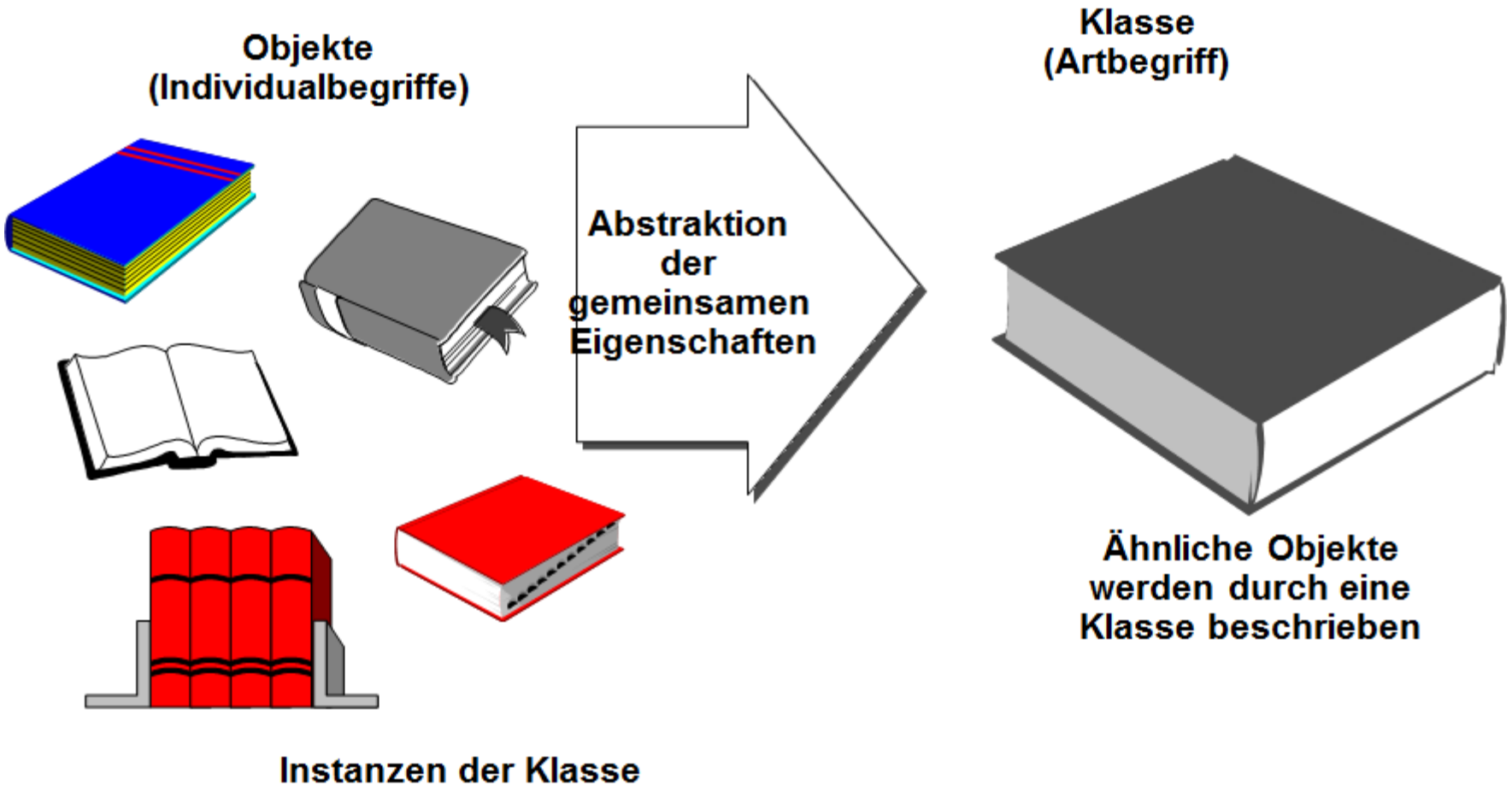


Eigenschaften von Objekten

- Ein Objekt
  - repräsentiert einen “Individualbegriff”,
  - hat eine eigene Identität,
  - zeigt ein für seine Art typisches Verhalten,
  - hat zu jedem Zeitpunkt einen Zustand, der für das Verhalten ausschlaggebend sein kann.

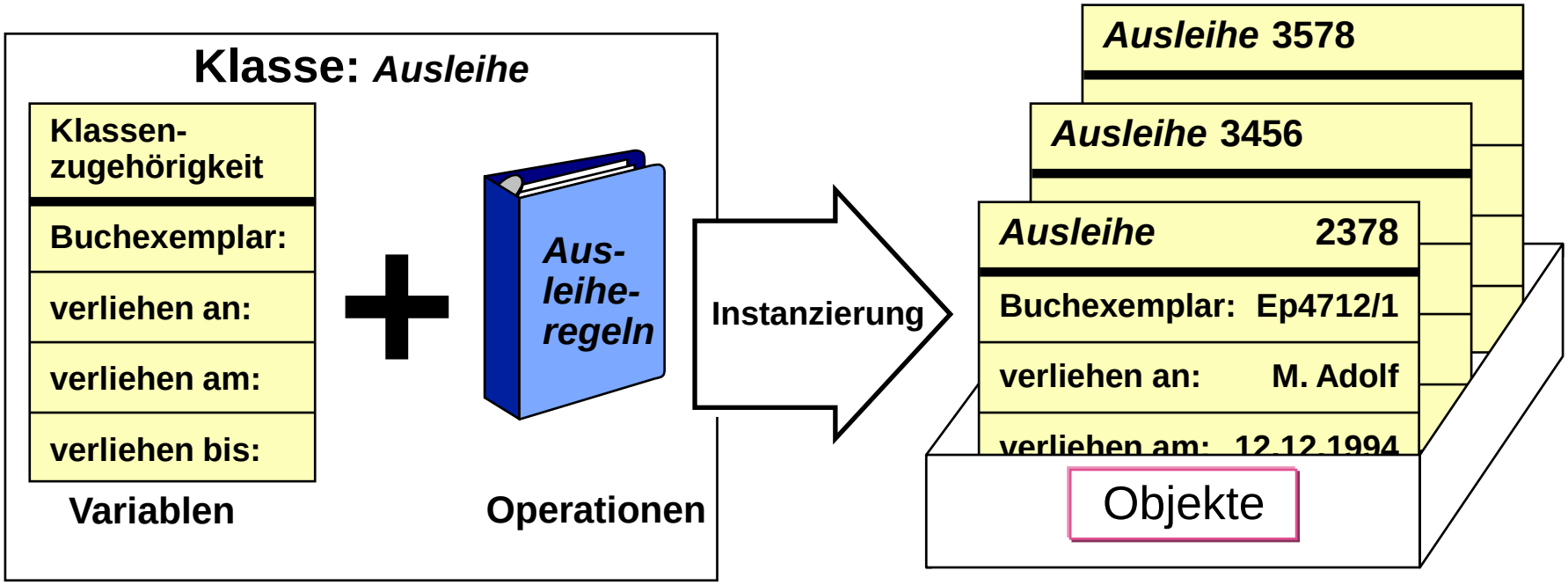


Objekt und Klasse



Klasse

- beschreibt den internen Aufbau der Objekte und
- ihr mögliches (artgerechtes) Verhalten;
- ist die Vorlage für beliebig viele gleichartige Objekte (Instanziierung).



Attribute und Beziehungen

- Die strukturellen (statischen) Eigenschaften eines Objekts werden durch die Spezifikation in der Klasse bestimmt.
- Sie werden durch Attribute und Beziehungen ausgedrückt
  und
- bieten Platz für die Informationen, die den Zustand der jeweiligen Objekte beschreiben.

Operationen

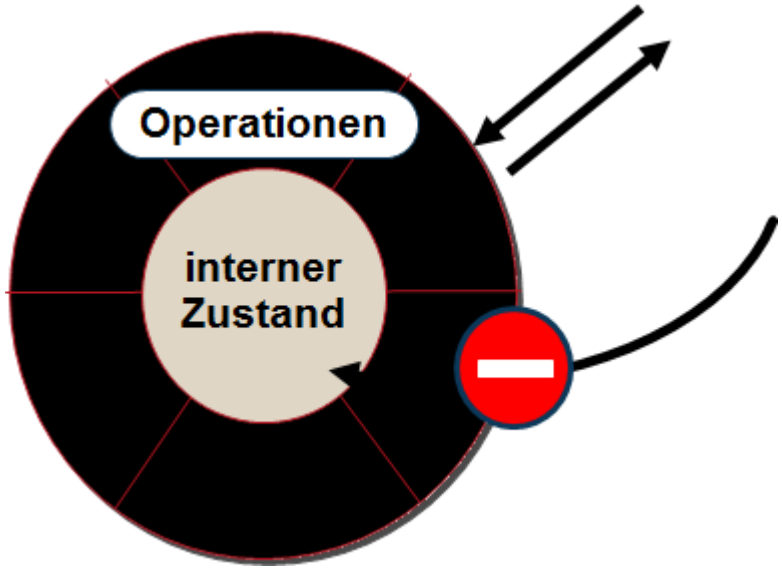
Dienste, Services oder Methoden

- **sind der verhaltensorientierte Teil,**
- **sind die dynamischen Eigenschaften,**
- **sind die Dienste, die ein Objekt oder eine Klasse bereithält,**
- **bestimmen, was man mit den Objekten machen kann (Nützlichkeit).**

Datenkapselung

- Operationen sind die einzige Möglichkeit, um mit den privaten Daten eines Objekts in Verbindung zu treten:
  - um den Zustand des Objekts zu verändern
  - um Auskunft über den Zustand des Objekts zu erhalten

  ⇨ Prinzip der Datenkapselung
- Umgekehrt sind alle Operationen öffentlich und können von anderen Objekten in Anspruch genommen werden.



Die Nutzbarkeit einer Klasse

Continental Institut für Technologie und Transformation

- **Stack**
  - erzeugen
  - pop
  - push
  - Inhalt des obersten Elements erfragen
  - Füllstand prüfen
  - löschen
- **Konto**
  - eröffnen
  - einzahlen
  - auszahlen
  - Kontostand erfragen
  - Kontobewegungen erfragen
  - Quartalsabschluss durchführen
  - Jahresabschluss durchführen
  - schließen
  - Kontoinhaber ermitteln

... wird nur über die erlaubten Operationen definiert

Datenabstraktion

- Nur die Schnittstelle wird veröffentlicht
- Zur Schnittstelle gehören
  - Name der Operationen
  - Parameter
  - Rückgabewert
  - Vorbedingung, d.h. welcher Zustand muss vorliegen, damit die Operation sinnvoll angewendet werden kann und
  - Nachbedingung, d.h. welcher Zustand liegt nach der Ausführung der Methode vor.
  - evtl. mögliche Fehlerzustände
- Implementation Hiding - die konkrete Implementierung ist nicht bekannt

Objekte und Client/Server-Prinzip

⇨ Programming by Contract

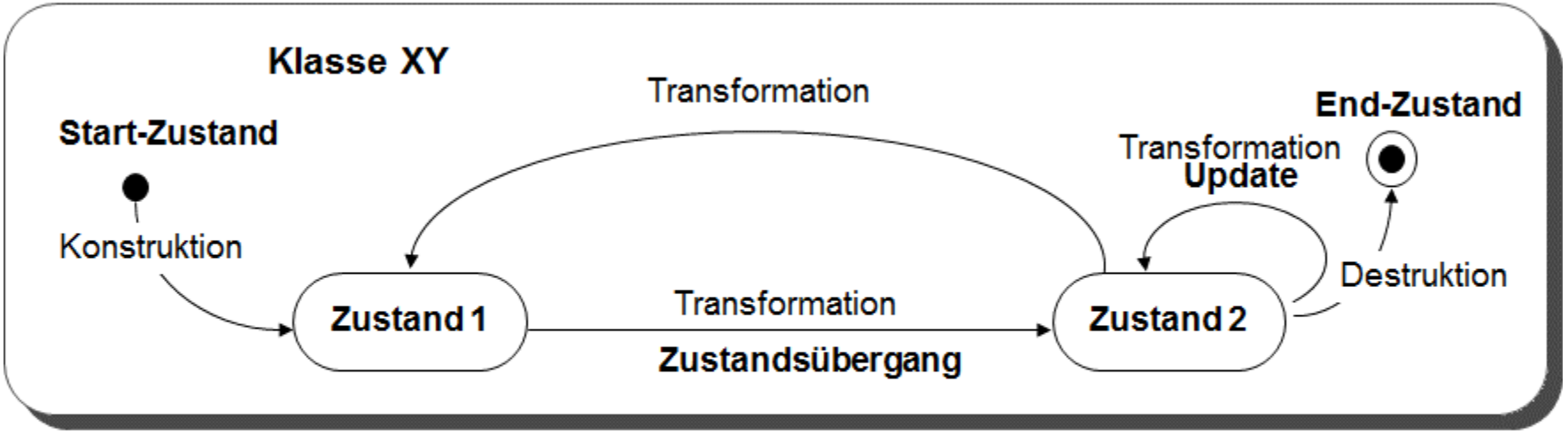
- Der Server (Objekt bzw. seine Klasse) verpflichtet sich,
  - die Spezifikation der Schnittstelle als implementierte Leistung bereitzustellen und
  - die Clients verlassen sich darauf.
- ein Client darf keine Annahmen über die interne Implementierung im Server treffen.
- Server dürfen keine Annahmen über die Art der Clients oder den jeweiligen Verwendungskontext treffen.

Protokoll

- Die Menge aller Operationen, die eine Klasse als Schnittstelle anbietet, heißt Protokoll.
- Das Protokoll ist öffentlich.
- Die darin enthaltenen Operationen können von anderen Objekten als Dienste in Anspruch genommen werden.

Objekt als Zustandsautomat

Das Protokoll muss den gesamten Lebenszyklus eines Objektes abdecken



- **Zustandsänderung**
  - Konstruktion: Aufgabe der Klasse selbst!
  - Transformation / Zustandsübergang
  - Destruktion

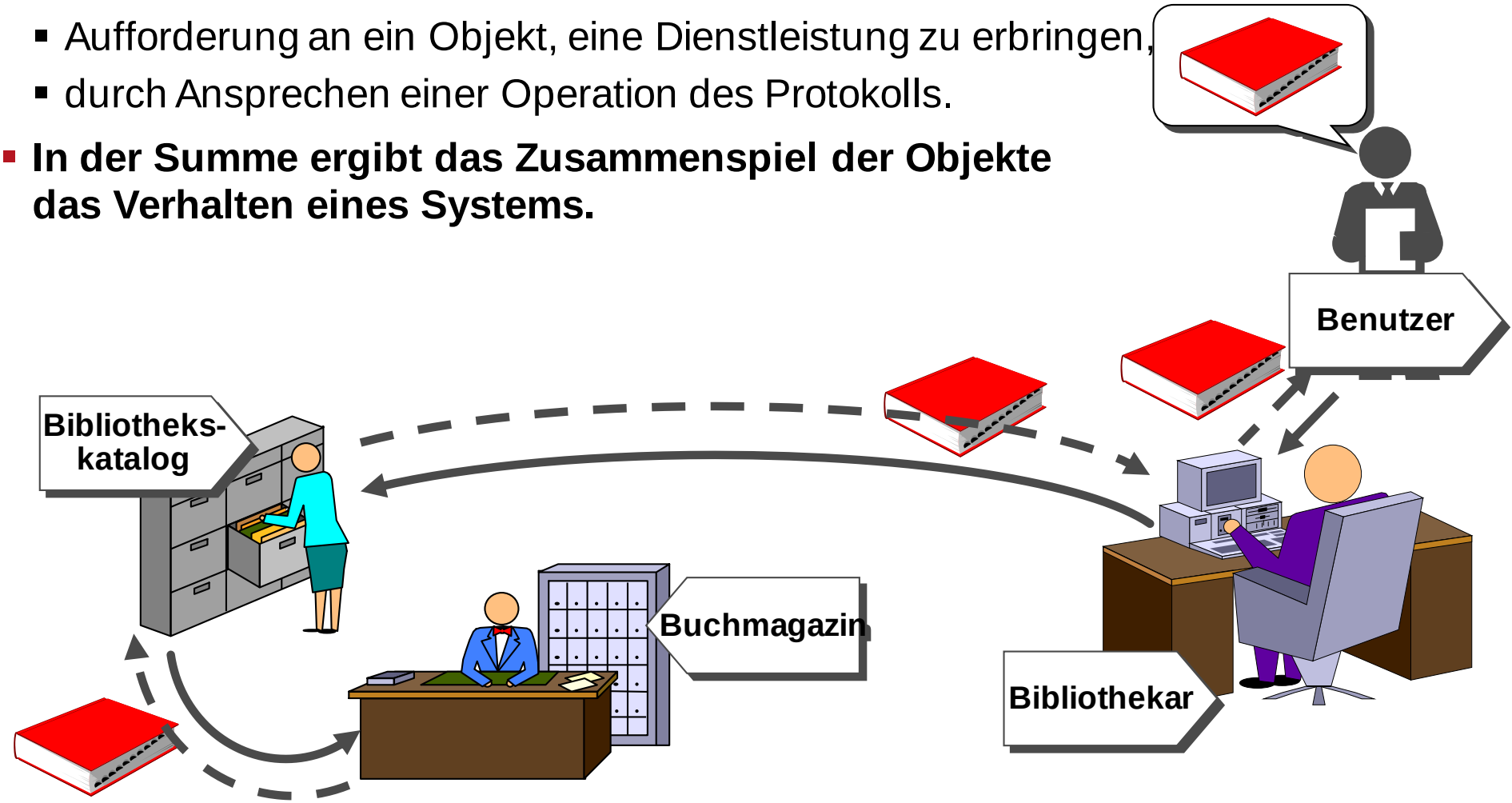
Methoden vs. Operationen

- Operationen spezifizieren die dynamischen Eigenschaften
- Das Protokoll ist die Summe aller Operationen
- Operationen werden durch Methoden implementiert
- Prozeduren und Funktionen sind somit entweder
  - Methoden, die als Operationen im Protokoll verzeichnet und damit öffentlich zugänglich sind

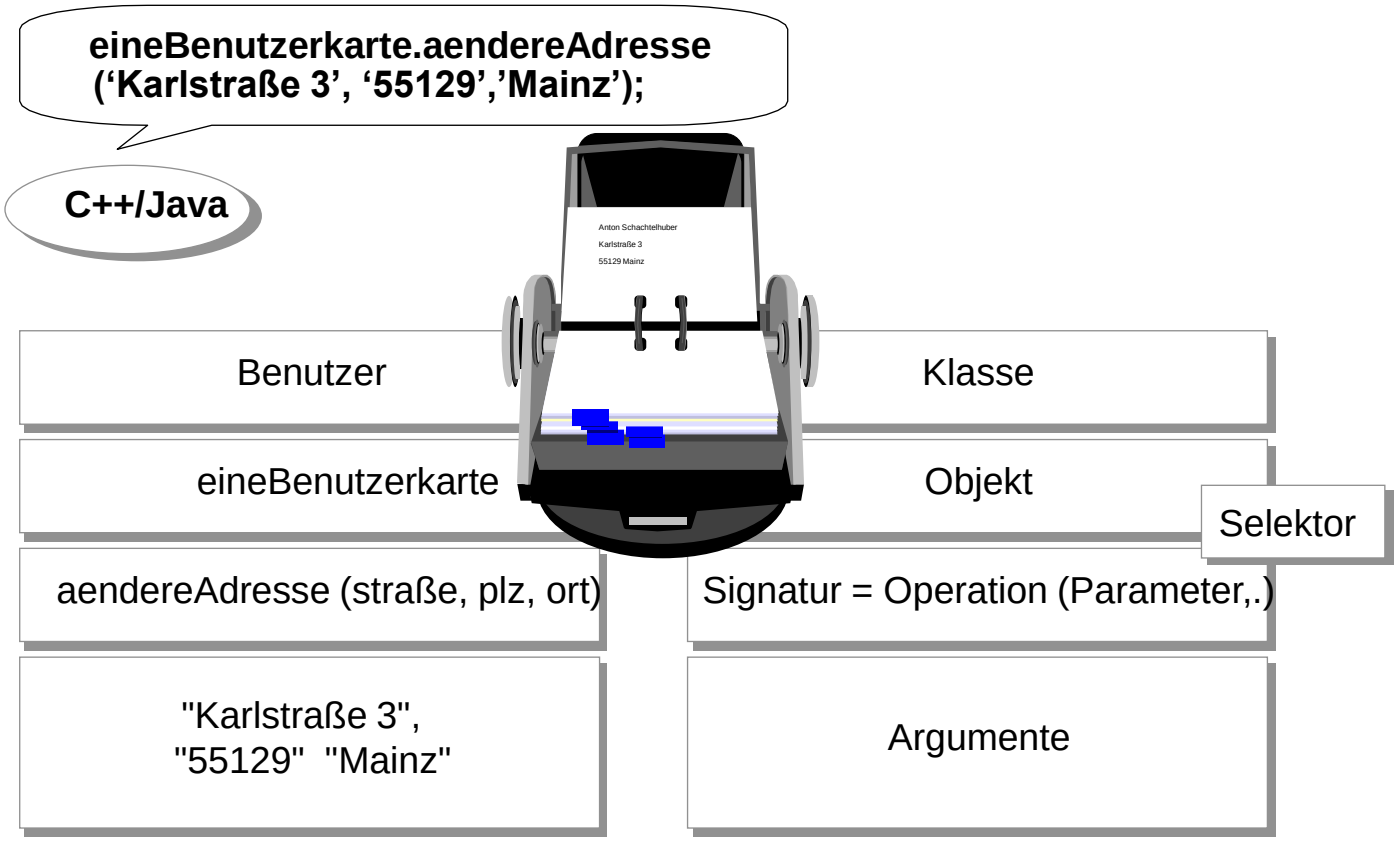
  oder
  - nach den Prinzipien der strukturierten Programmierung entworfene private Unterprogramme
- Methoden tun nichts, was andere bereits tun

Nachricht

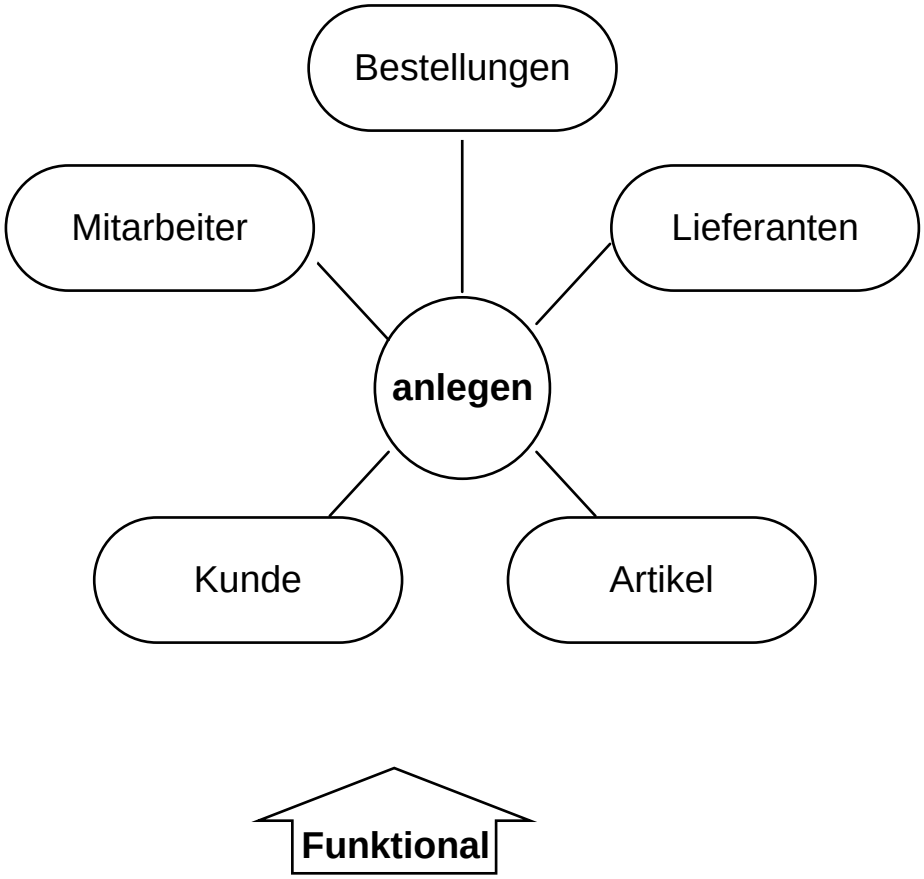
- **Objekte kommunizieren untereinander durch das Versenden von Nachrichten:**
  - Aufforderung an ein Objekt, eine Dienstleistung zu erbringen,
  - durch Ansprechen einer Operation des Protokolls.
- **In der Summe ergibt das Zusammenspiel der Objekte das Verhalten eines Systems.**

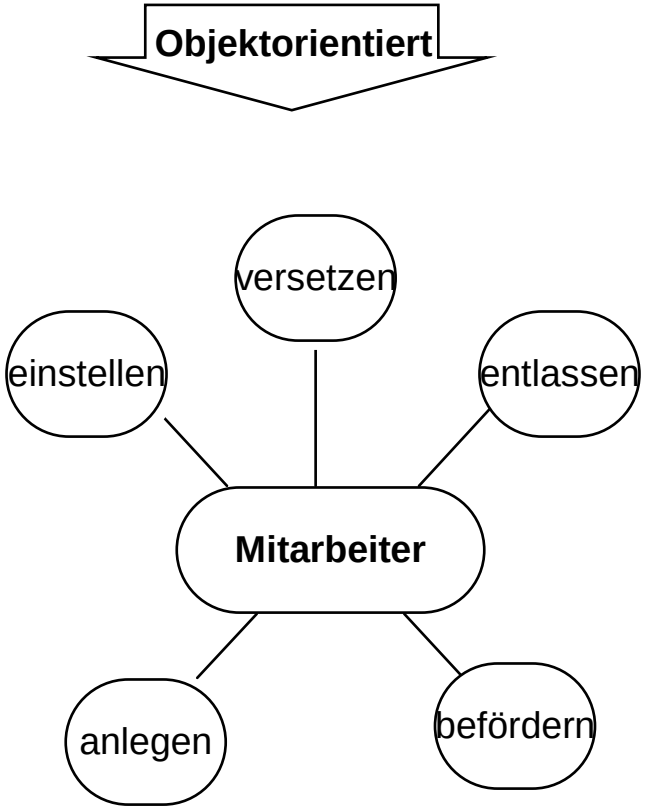


Nachricht

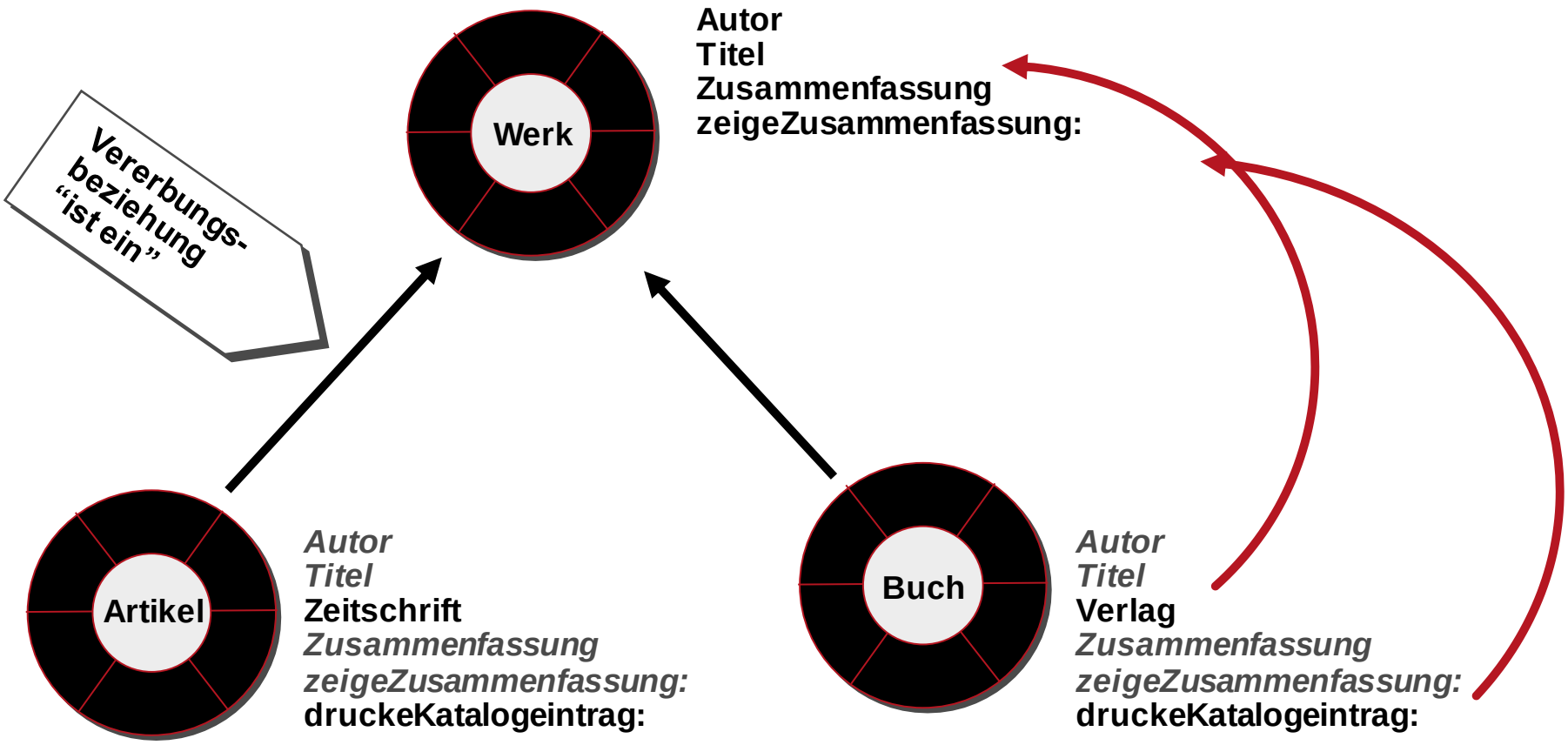

eineBenutzerkarte.aendereAdresse
(‘Karlstraße 3’, ‘55129’,’Mainz’);
C++/Java
Anton Schachtelhuber
Karlstraße 3
55129 Mainz
Benutzer
Klasse
eineBenutzerkarte
Objekt
Selektor
aendereAdresse (straße, plz, ort)
Signatur = Operation (Parameter,.)
"Karlstraße 3",
"55129" "Mainz"
Argumente

Objektparadigma





Generalisierung und Spezialisierung



Vererbung

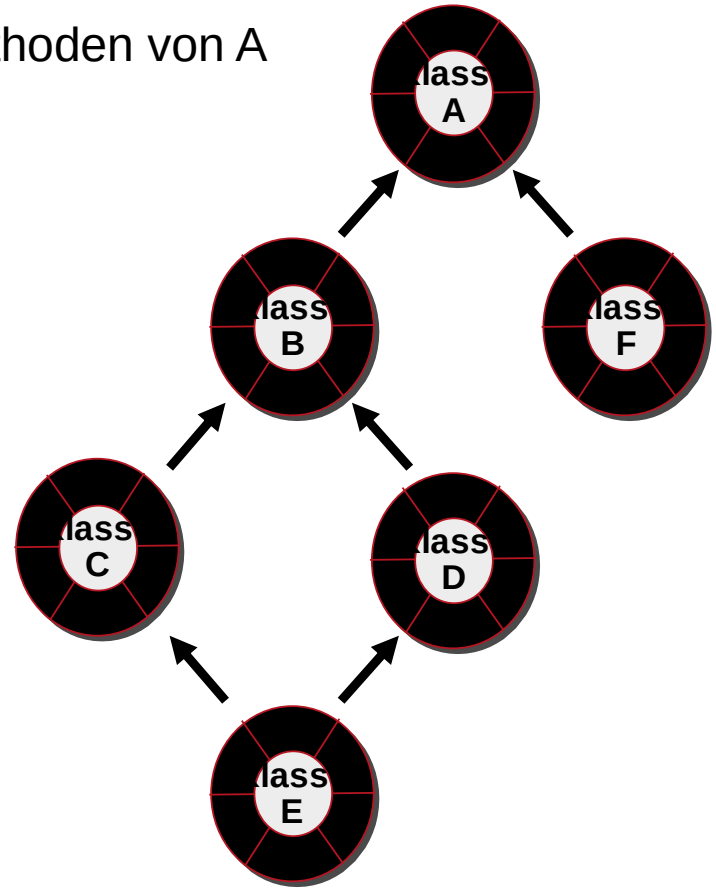
- **Wiederverwendung von Eigenschaften der Oberklasse durch Unterklasse(n).**
- **Vererbung unterstützt**
  - Generalisierung / Abstraktion
  - Spezialisierung / Konkretisierung
  - Erweiterung von Klassen, die nicht verändert werden sollen (z.B. gekaufte oder alte Klassen)
- **Unterklasse spezifiziert nur die Unterschiede in den Eigenschaften**
  - hinzufügen
  - verändern
  - konkretisieren
  - umbenennen
  - (unterdrücken)!!

is_a
is_Kind_of

superclass
subclass
inheritance

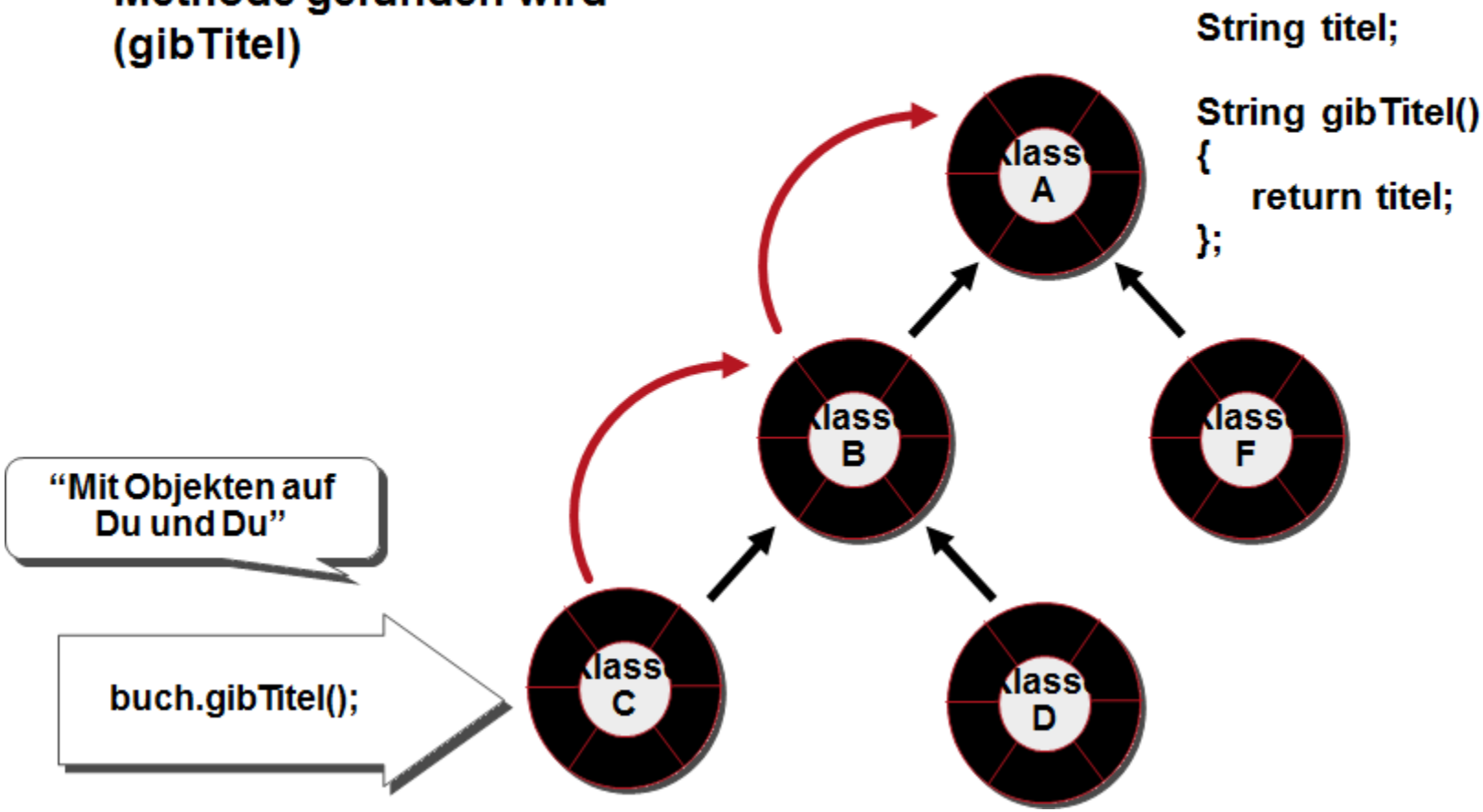
Vererbung und Klassenhierarchie

- Beziehung zwischen Klassen
  - Klasse B ist eine Unterklasse von A
  - Jedes Instanz von B ist erbt Attribute / Methoden von A
  - Die Klasse B enthält nur die speziali-sierenden Unterschiede zur Klasse A
- Vererbung ist transitiv
  - C erbt von B, B erbt von A, damit erbt C auch alle Eigenschaften von A, die nicht durch B verändert wurden
- Mehrfachvererbung möglich
  - E erbt von C und D
  - E erbt (transitiv) von B über C und über D
  - ☝ *dabei gibt es jedoch Konfliktpotential*



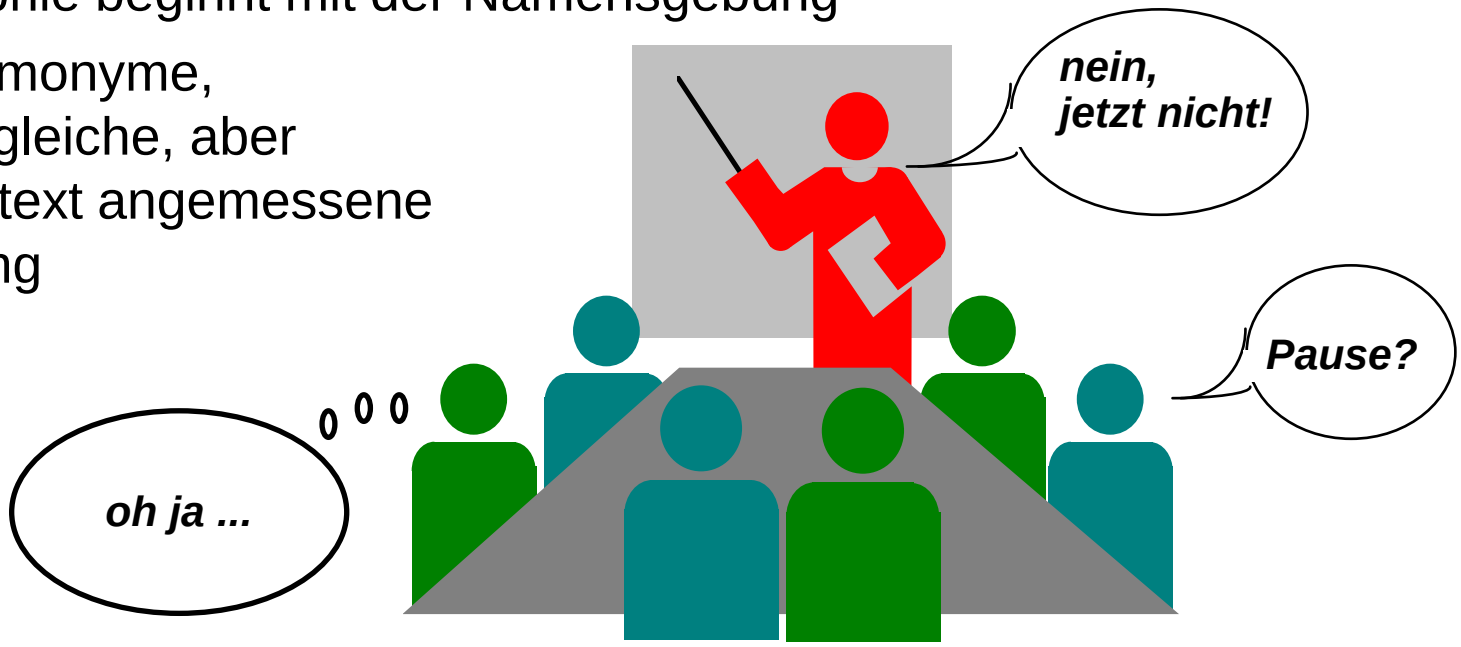
Von der Nachricht zur Methode

- **Methodensuche in der Vererbungshierarchie, bis die Methode gefunden wird (gibTitel)**

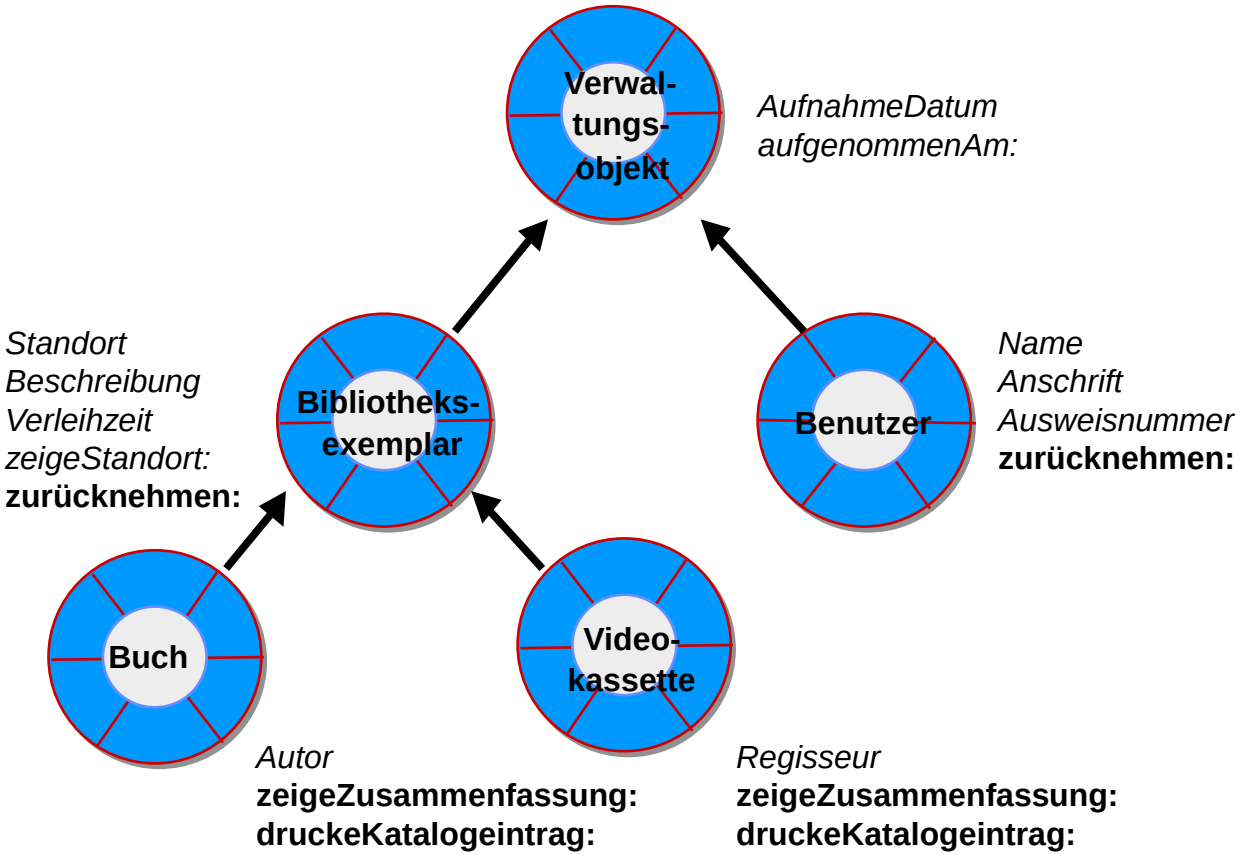


Polymorphie

- griechisch "Vielgestaltigkeit"
- gleiche Operation, unterschiedliche Methoden
- dieselbe Nachricht löst bei Objekten verschiedener Klassen unterschiedliche Reaktionen aus
- Polymorphie beginnt mit der Namensgebung
- keine Homonyme, sondern gleiche, aber dem Kontext angemessene Bedeutung



Polymorphie in der Vererbungshierarchie



Dynamisches und statisches Binden

- Statisches Binden
  - Die Adresse der Methode wird durch den Linker an der aufrufenden Stelle in den Objektcode eingetragen.
- DLLs
- Dynamisches Binden
  - ermittelt Operation und Adresse der zugehörigen Methode erst zur Laufzeit,
  - muss bei Polymorphie verwendet werden,
  - wird auch als spätes Binden bezeichnet.
- Interpretieren ist nicht gleich dynamisches Binden!
  - Interpretieren = Quelltext übersetzen zur Laufzeit

Überladen

Überladen von Operationsnamen oder Operatoren

- Anwendungsbeispiele
  - Typspezifische Definition der Vergleichsoperatoren

```
IF Eintrag1 > Eintrag2 THEN ...
```

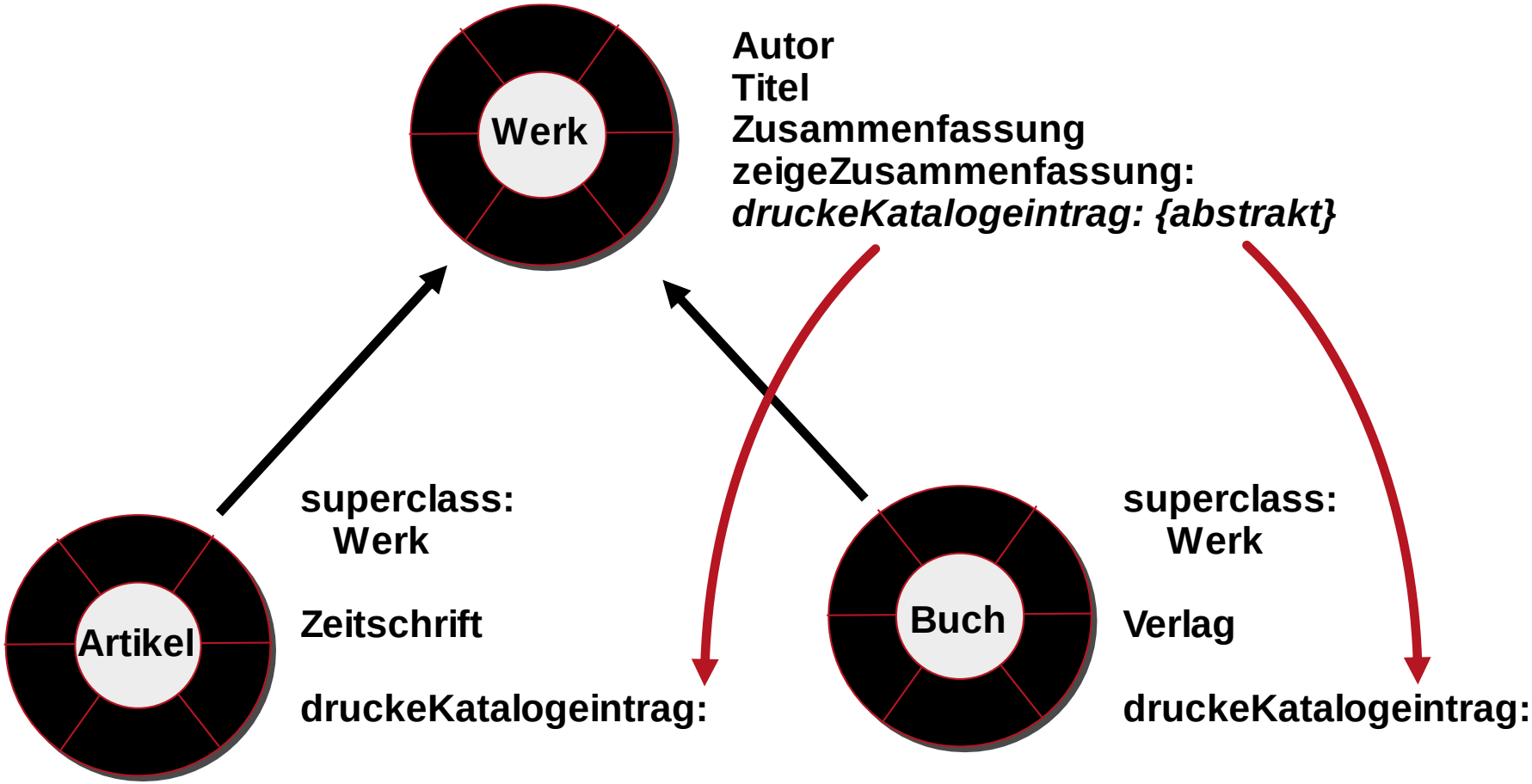
  - Konstruktor in unterschiedlichen Varianten:

```
neuesBuch = new Buch (Titel, Autor);
neuesBuch = new Buch (Titel, Autor, Herausgeber);
```

  - Defaultwerte für Parameter:

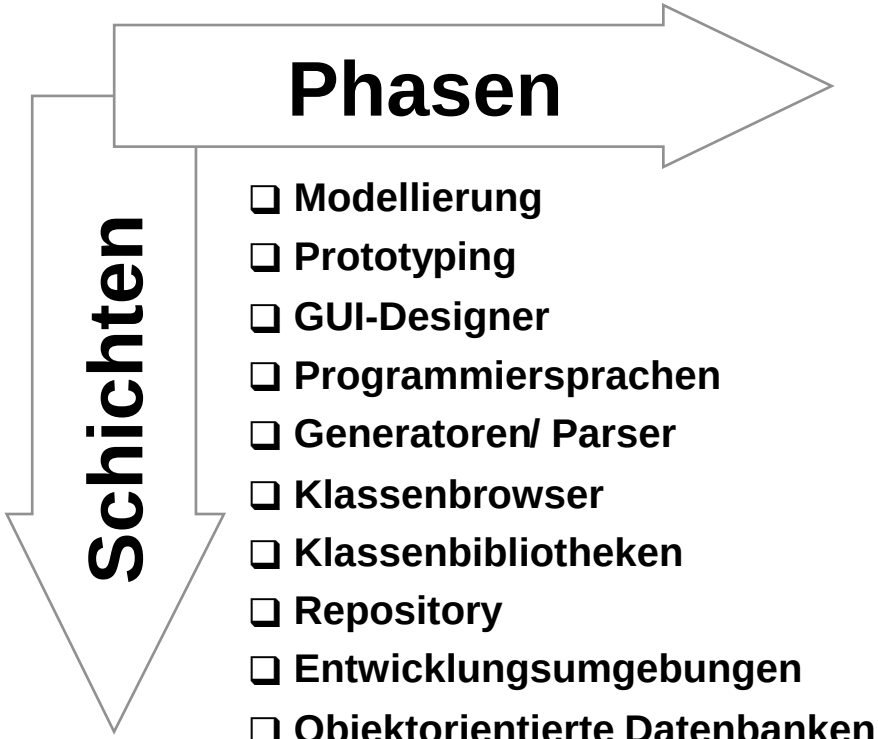
```
buch.verleihen (Ausleihdatum, Ausleihfrist);
buch.verleihen (Ausleihdatum);
                // verwendet Standardfrist
buch.verleihen ();     // ab heute mit Standardfrist
```

Abstrakte Klasse



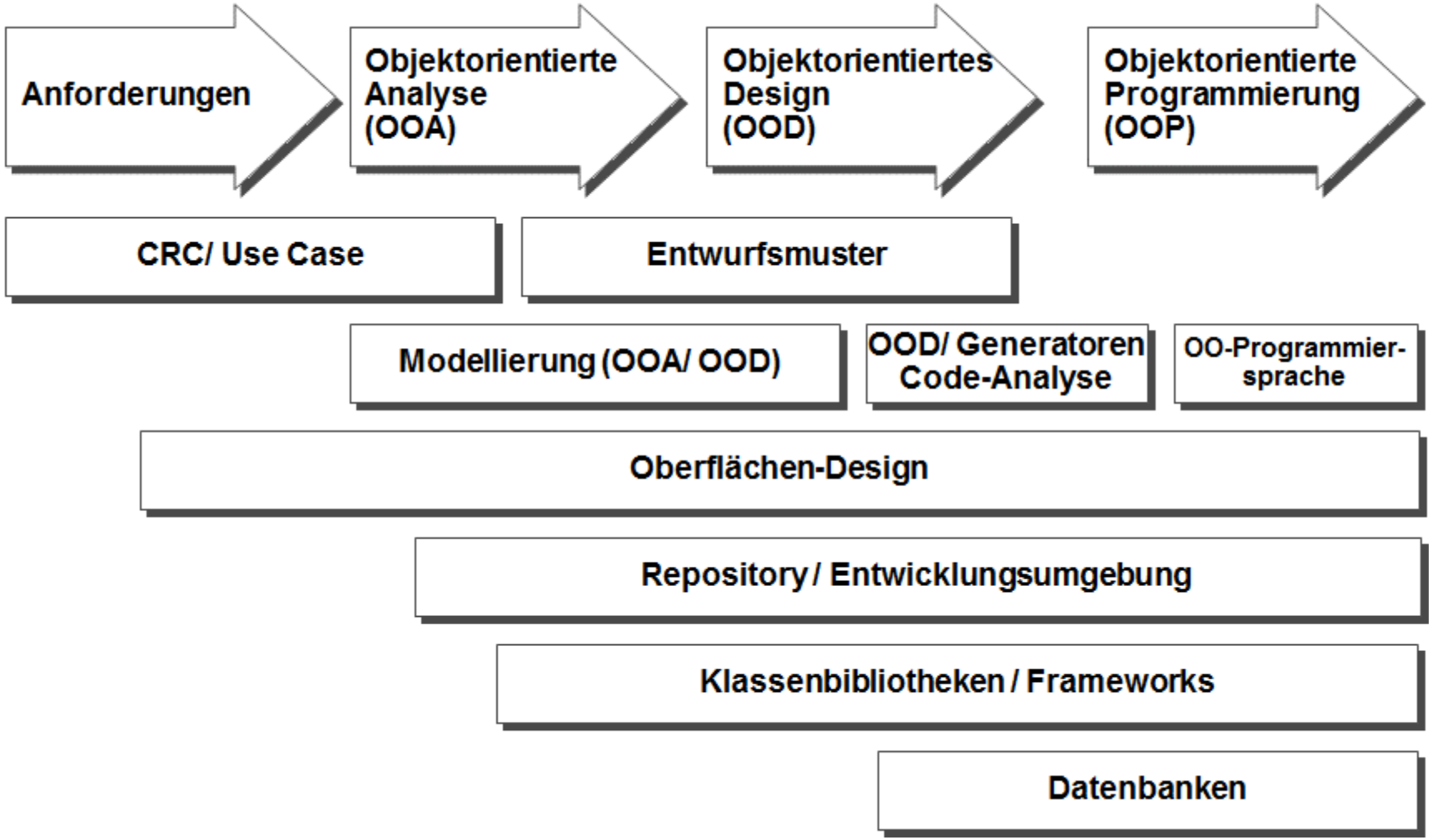
2

TECHNOLOGIEN IN DER OBJEKTORIENTIERTEN SYSTEMENTWICKLUNG

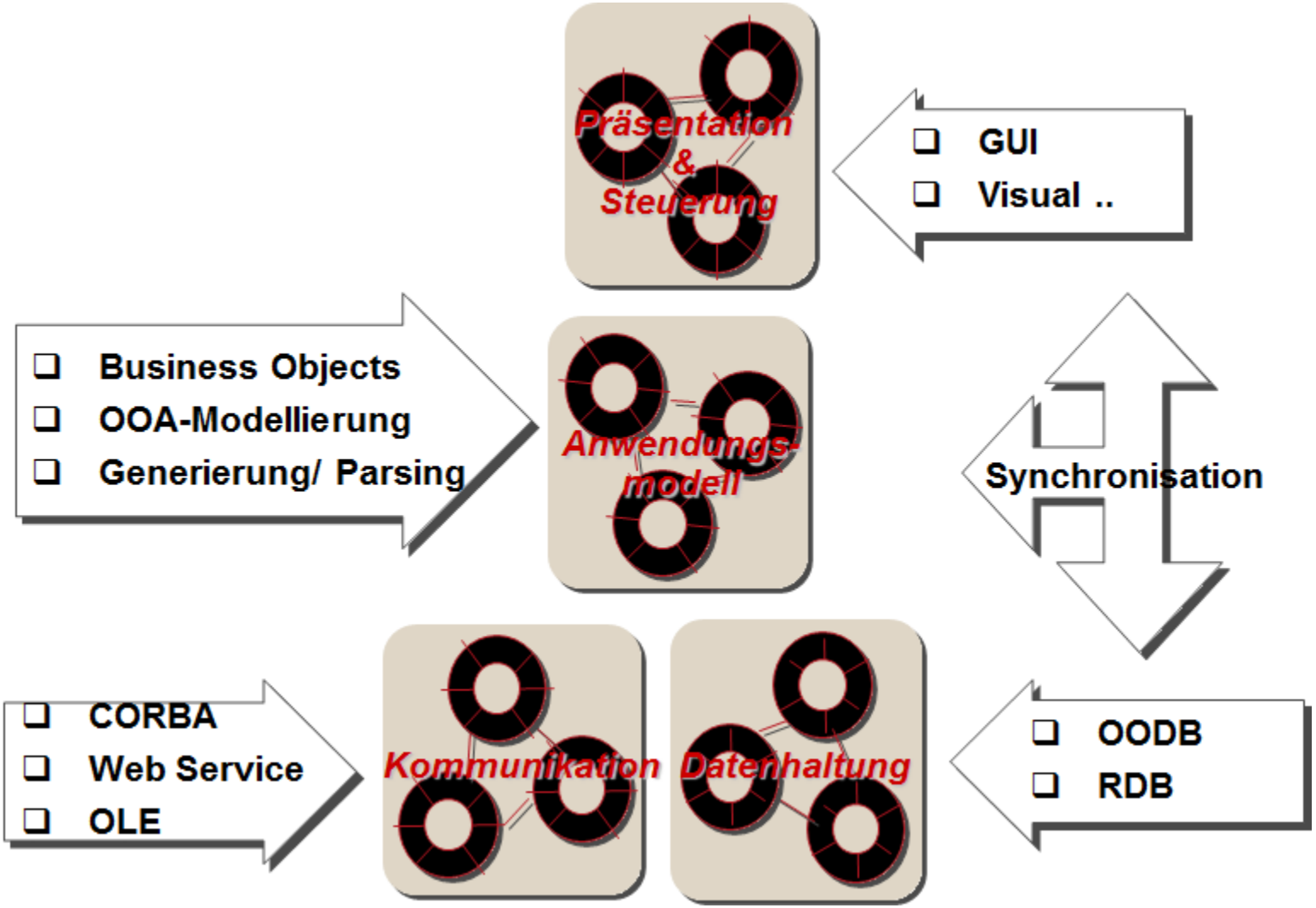


- Modellierung
- Prototyping
- GUI-Designer
- Programmiersprachen
- Generatoren/ Parser
- Klassenbrowser
- Klassenbibliotheken
- Repository
- Entwicklungsumgebungen
- Objektorientierte Datenbanken

Objekttechnik im Phasenverlauf



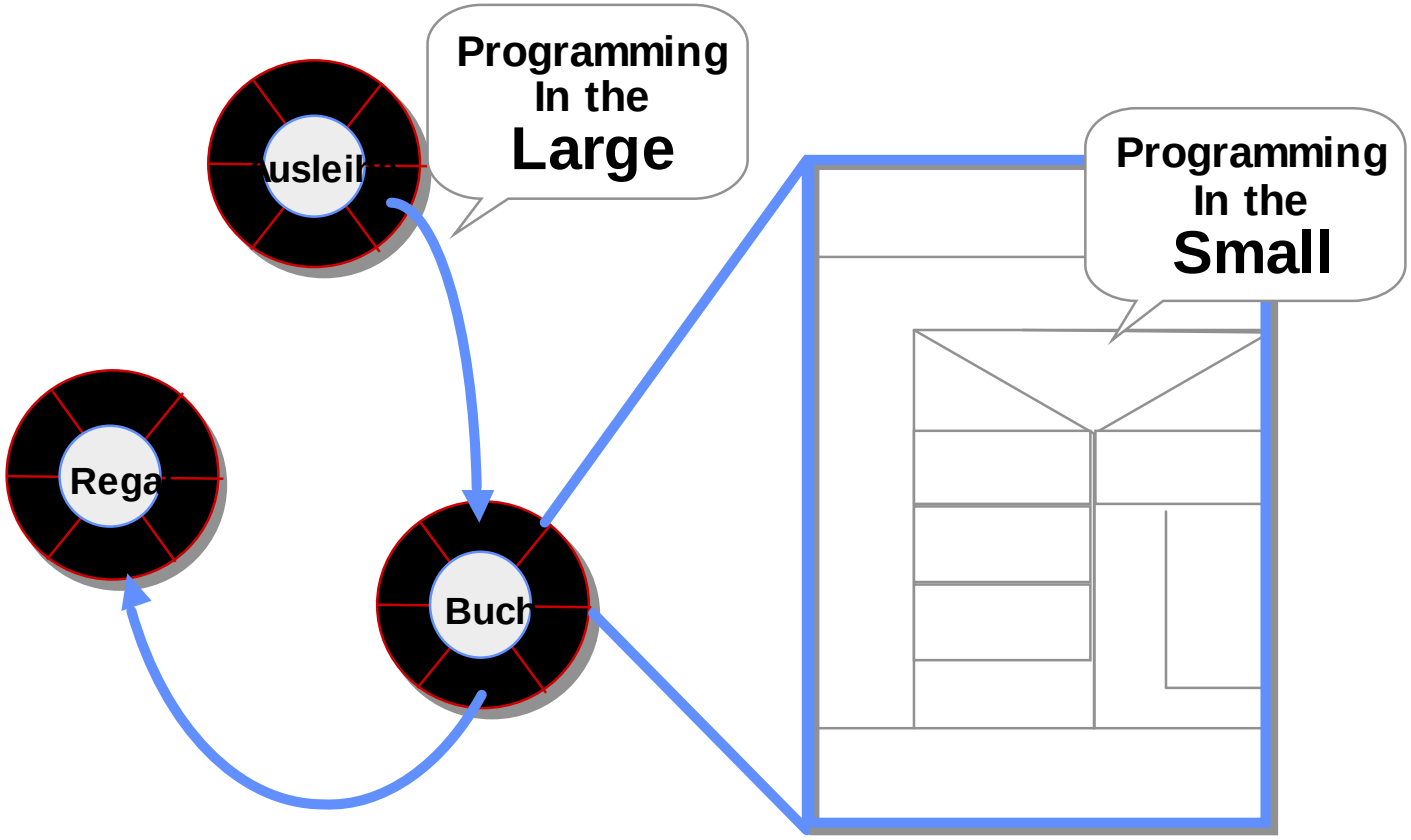
Objekttechnik im Schichtenmodell



Programmiersprachen

- **Simula (1967)**
  - Simulation
- **Smalltalk (1980)**
  - Interaktion
- **C++ (1983)**
  - C mit Klassen
- **Eiffel (1988)**
  - Sicherheit
- **OO-Cobol (1997)**
  - Investitionssicherung
- **Object PASCAL**
  - Object PASCAL/ Apple und Turbo-PASCAL
- **ADA 95**
  - 1995 freigegebene ANSI-Überarbeitung von ADA
- **JAVA**
  - Die "Internet"-Sprache
- **....**

Strukturierte vs. objektorientierte Programmierung



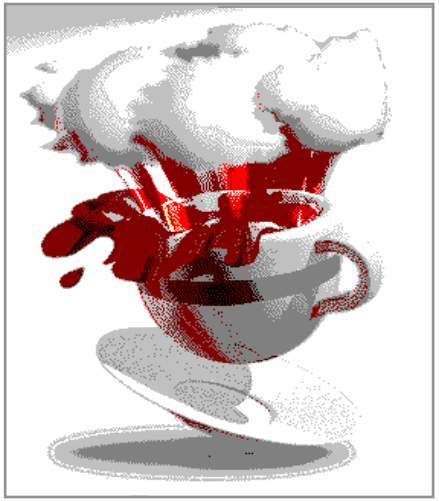
Übersicht

Merkmal	Smalltalk	C++	JAVA
Vererbung	einfach	mehrfach	einfach
Typbindung	„dynamisch“	sicher	sicher
Bindung	dynamisch	statisch / dynamisch	dynamisch / statisch
Garbage Collection	automatisch	nein	automatisch
Übersetzung	Interpreter und Compiler	Compiler	Interpreter und Compiler
Polymorphie	ja	ja	ja
Kapselung	ja	ja (mit „Hintertüren)	ja
Exception handling	ja	ja	ja

- Bjarne Stroustrup (AT&T Bell Labs.)
  - Erweiterung von C um die Konzepte von Simula
  - C++ = Inkrement von C
  - Laufzeit-Effizienz
- Hybride Sprache
  - Normalerweise statische Bindung, dynamische Bindung möglich
  - Operator Overloading
  - Zusätzliches Schlupfloch in die Kapsel durch Friend-Beziehung („befreundete“ Klassen dürfen private Daten direkt bearbeiten)
- ANSI-Standard 1998

Java

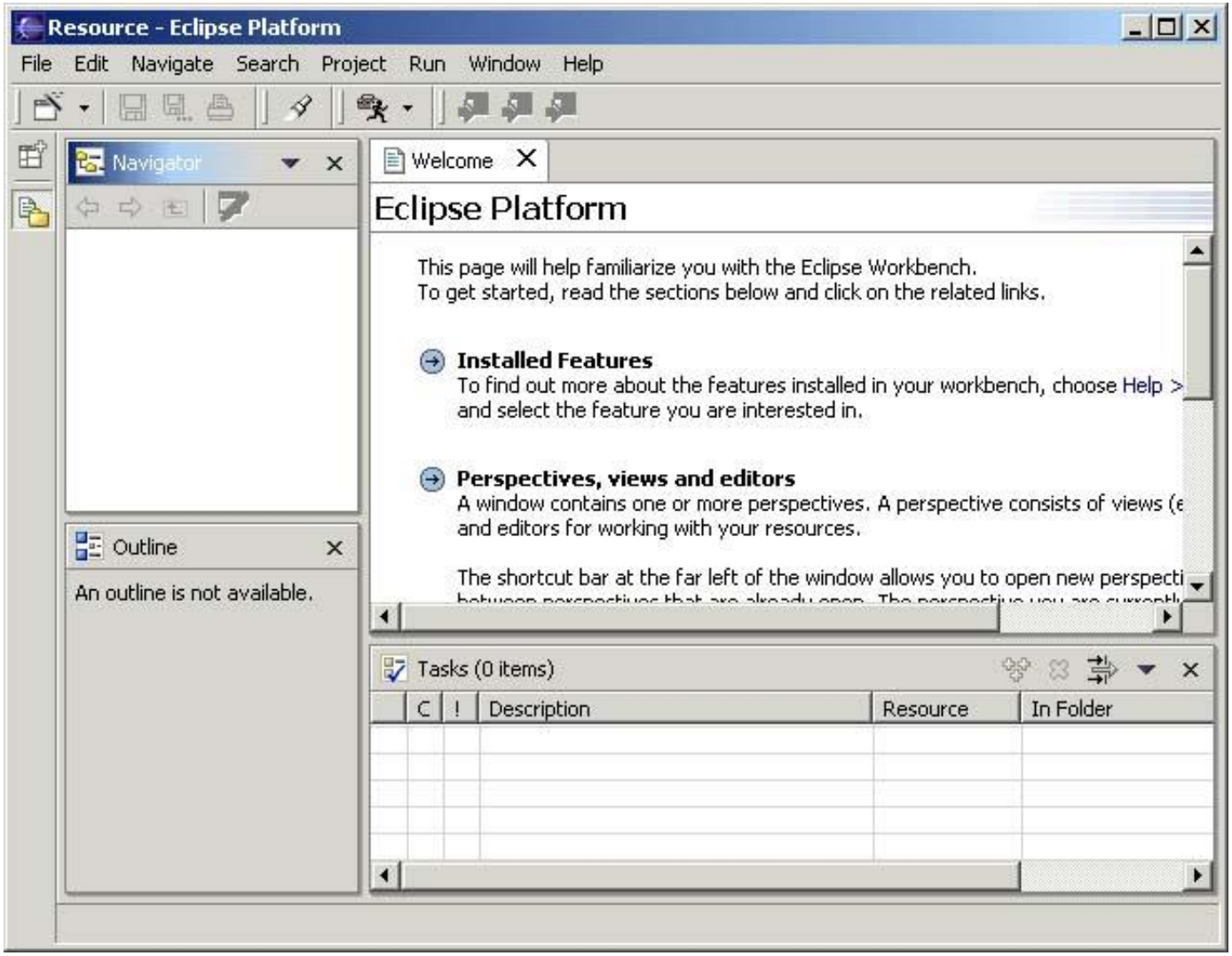
- James Goslin (Sun)
  - Vereinfachung und Erweiterung von C++
  - Übernahme von Konzepten aus Smalltalk
- Ursprünglich Vermarktung als "Internet"-Sprache
  - Plattformunabhängigkeit: Write once, run anywhere
  - Inzwischen Mehrzweck-Sprache
  - Anwendungsserver
- Standardisierung im „Java Community Process“



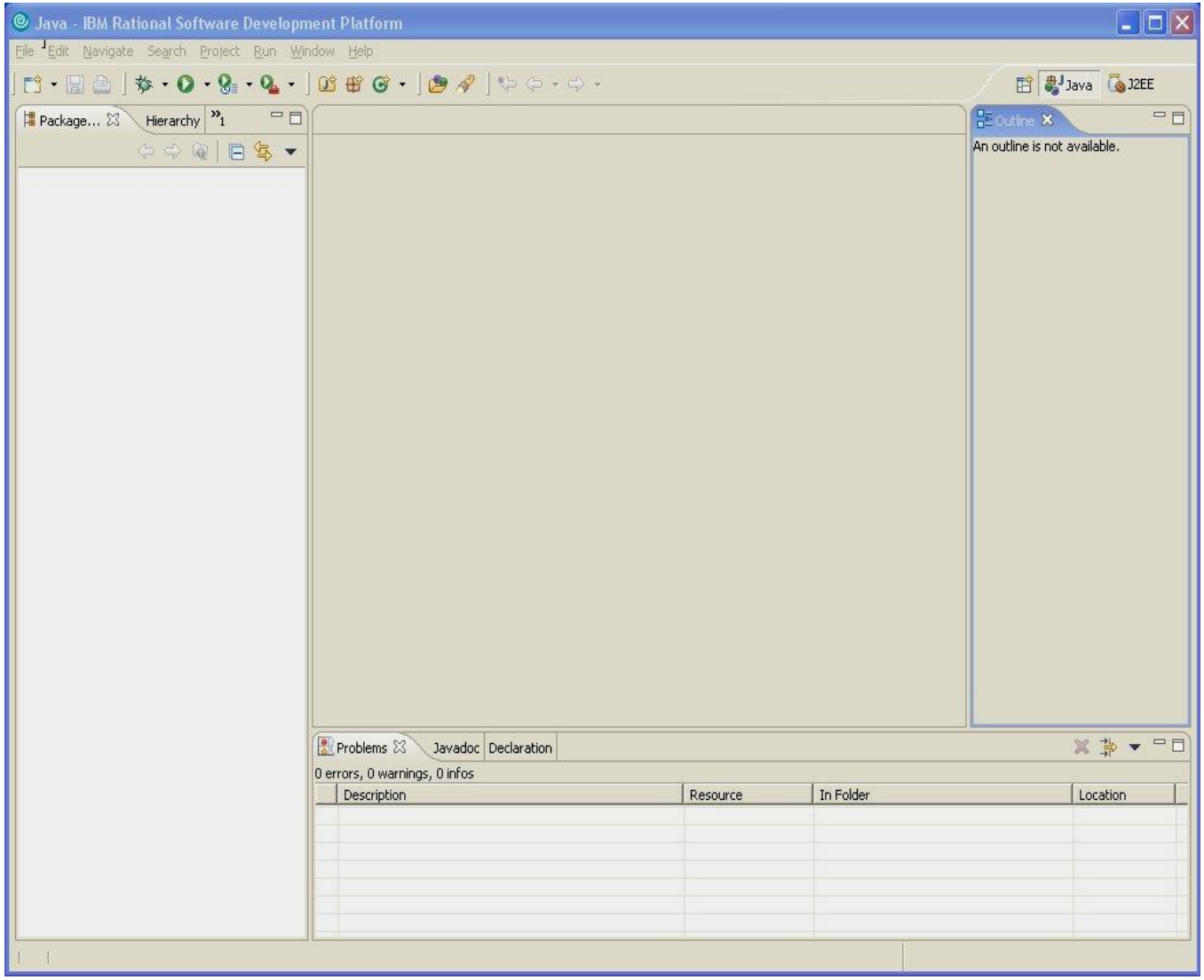
Eigenschaften von Entwicklungsumgebungen

- Workbench zur zentralen Organisation aller Artefakte
- Browser mit verschiedenen Filtern zur besseren und übersichtlichen Darstellung der Klassenbibliothek
- Teilweise Repository-basiert. Trend geht allerdings zu externen Versionsverwaltungen
- Editor zur komfortablen Programmierung inkl. diverser Hilfsmittel, Werkzeuge (Code-Assistent, Inspector, Debugger etc.)
- Ressourcen-Editor zur Erstellung von Oberflächen
- Smart-Guides oder Wizards zur Erstellung von verschiedenen Anwendungen oder Komponenten
- Modell Driven Architecture (MDA) Ansätze
- .....

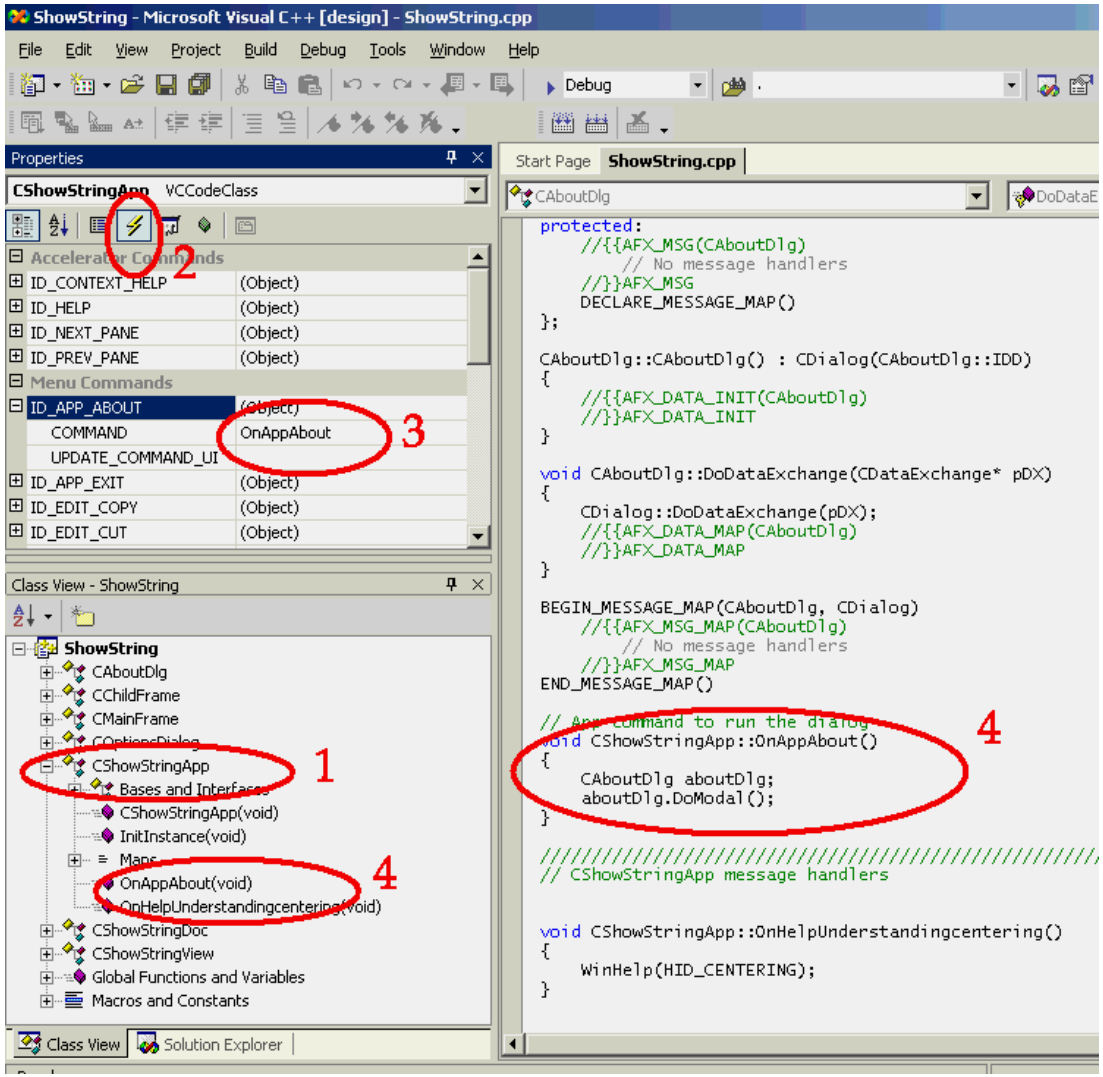
Eclipse


Resource - Eclipse Platform
File Edit Navigate Search Project Run Window Help
Navigator
Welcome
Eclipse Platform
This page will help familiarize you with the Eclipse Workbench.
To get started, read the sections below and click on the related links.
Installed Features
To find out more about the features installed in your workbench, choose Help >
and select the feature you are interested in.
Perspectives, views and editors
A window contains one or more perspectives. A perspective consists of views (e
and editors for working with your resources.
The shortcut bar at the far left of the window allows you to open new perspecti
Outline
An outline is not available.
Tasks (0 items)
C
!
Description
Resource
In Folder

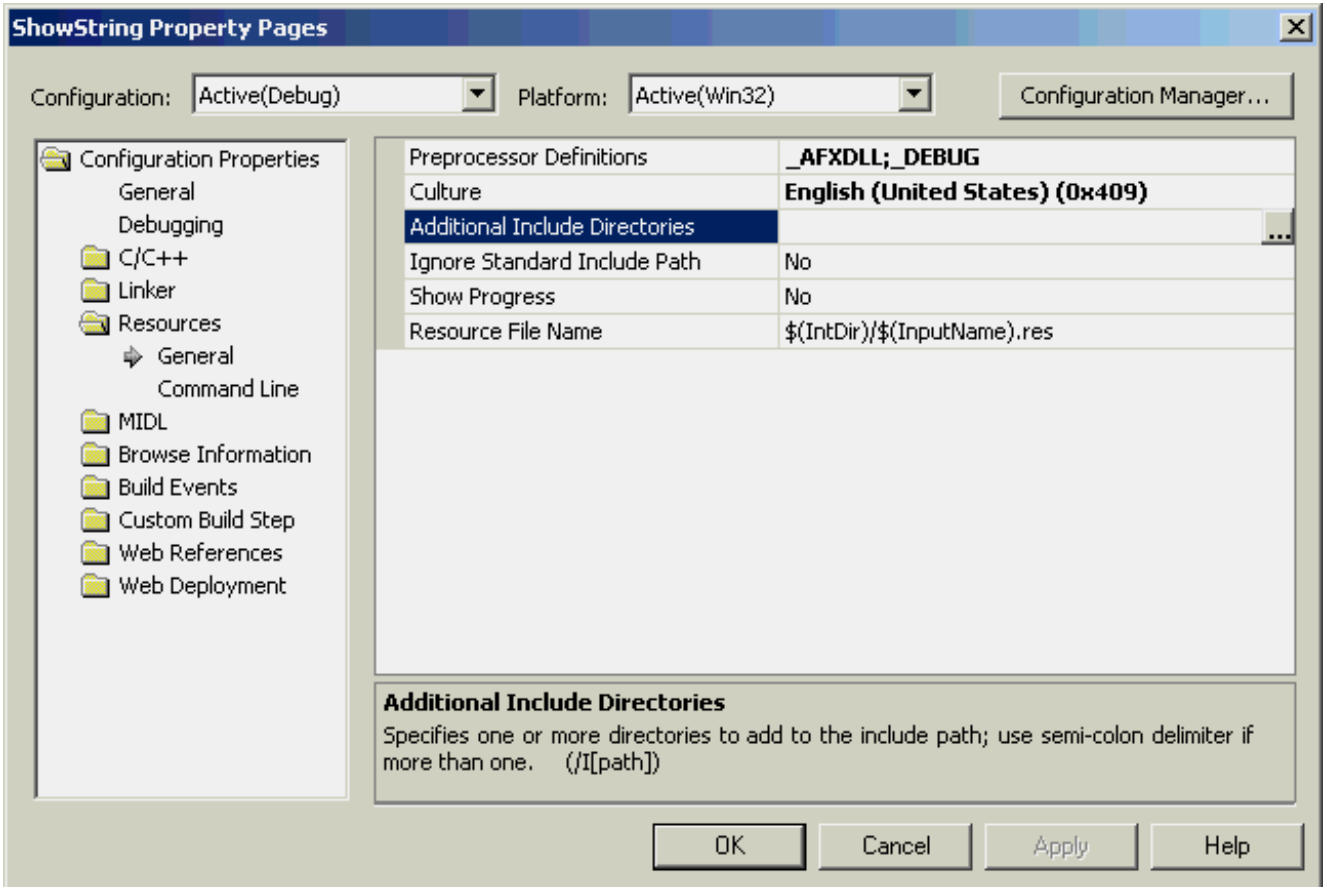
Die Workbench



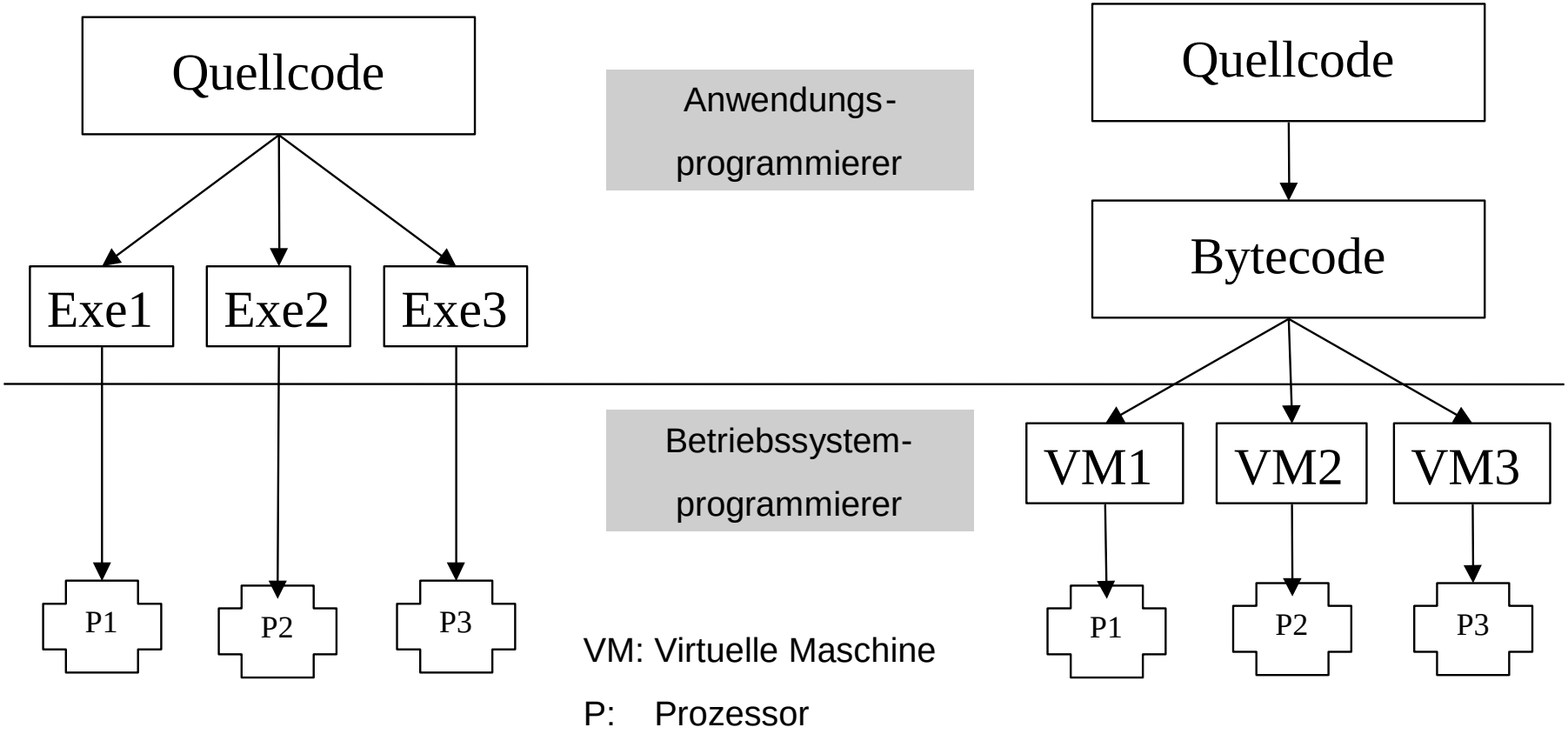
Die Entwicklungsumgebung Visual C++



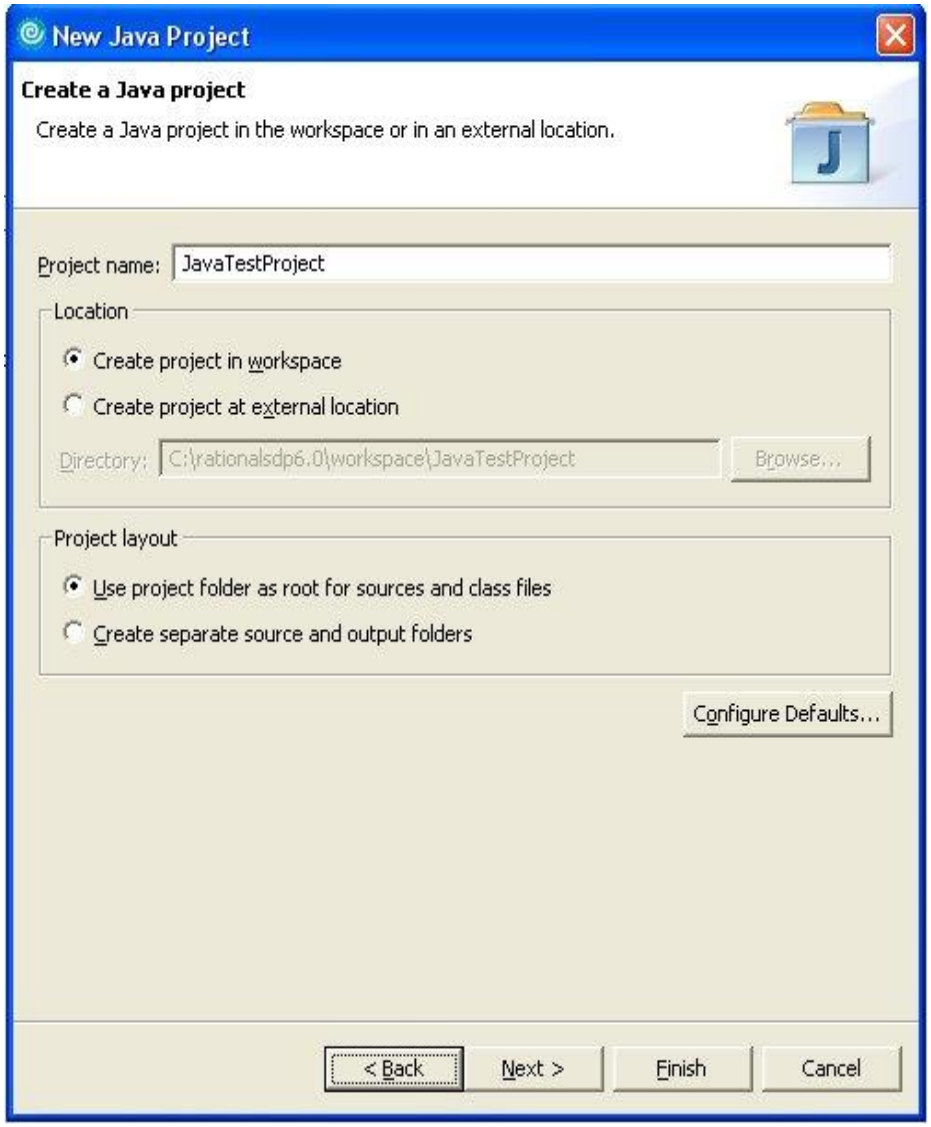
Anpassen der Entwicklungsumgebung



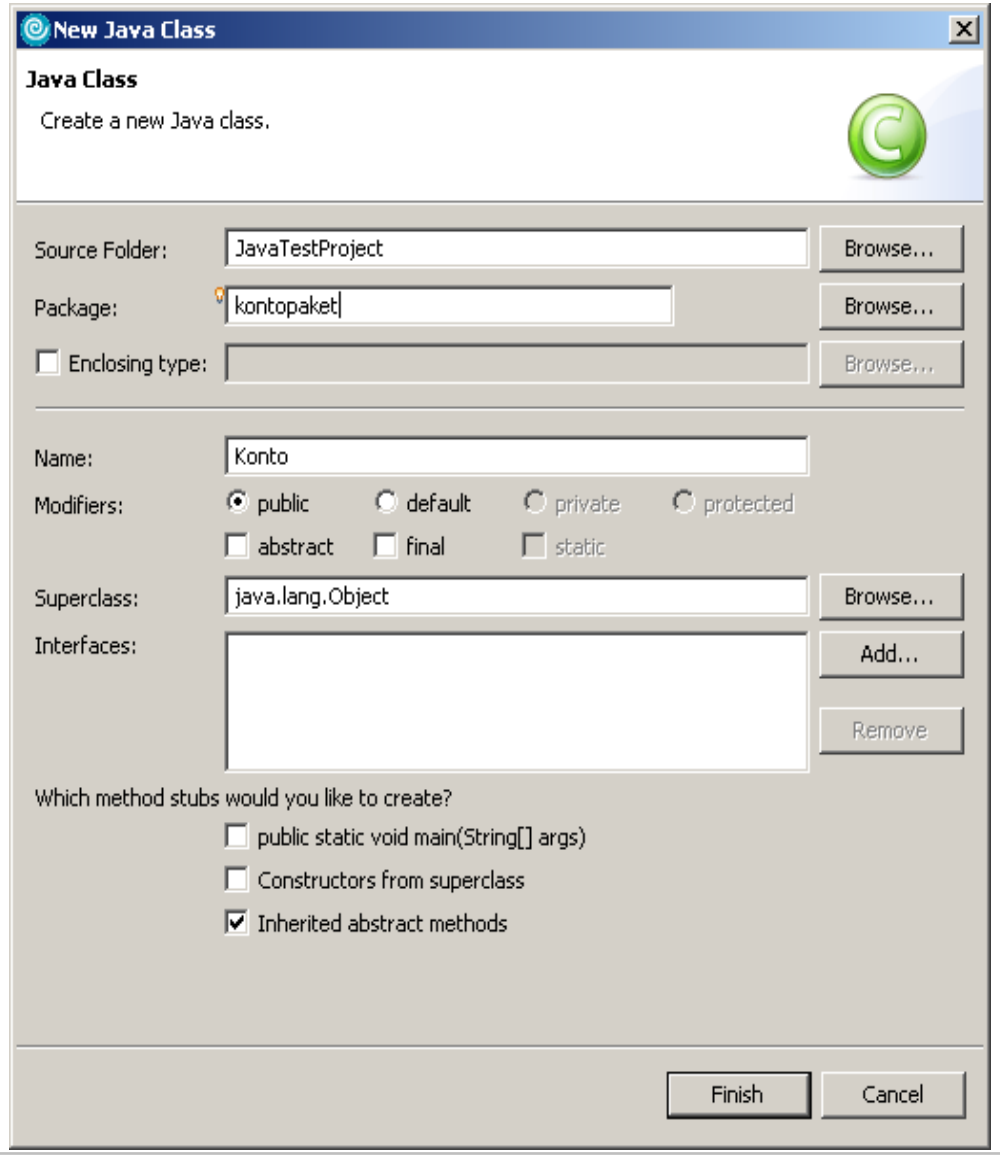
Plattform unabhängige Programmierung



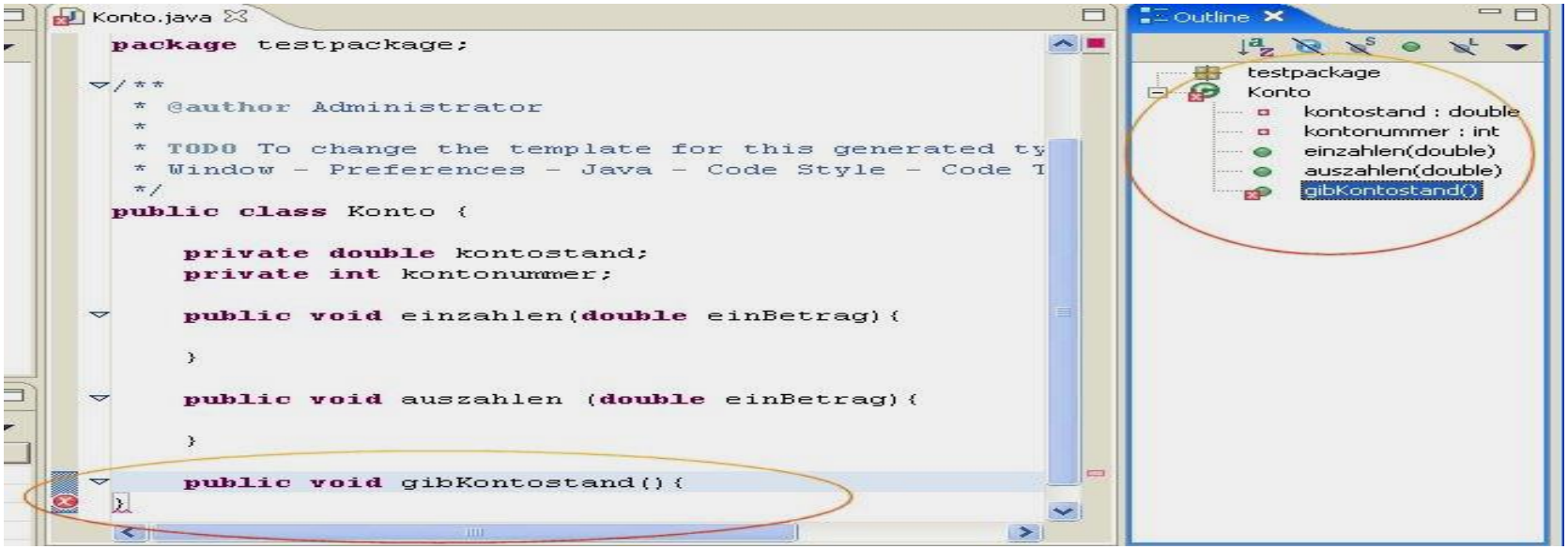
Erstellen eines Projekts



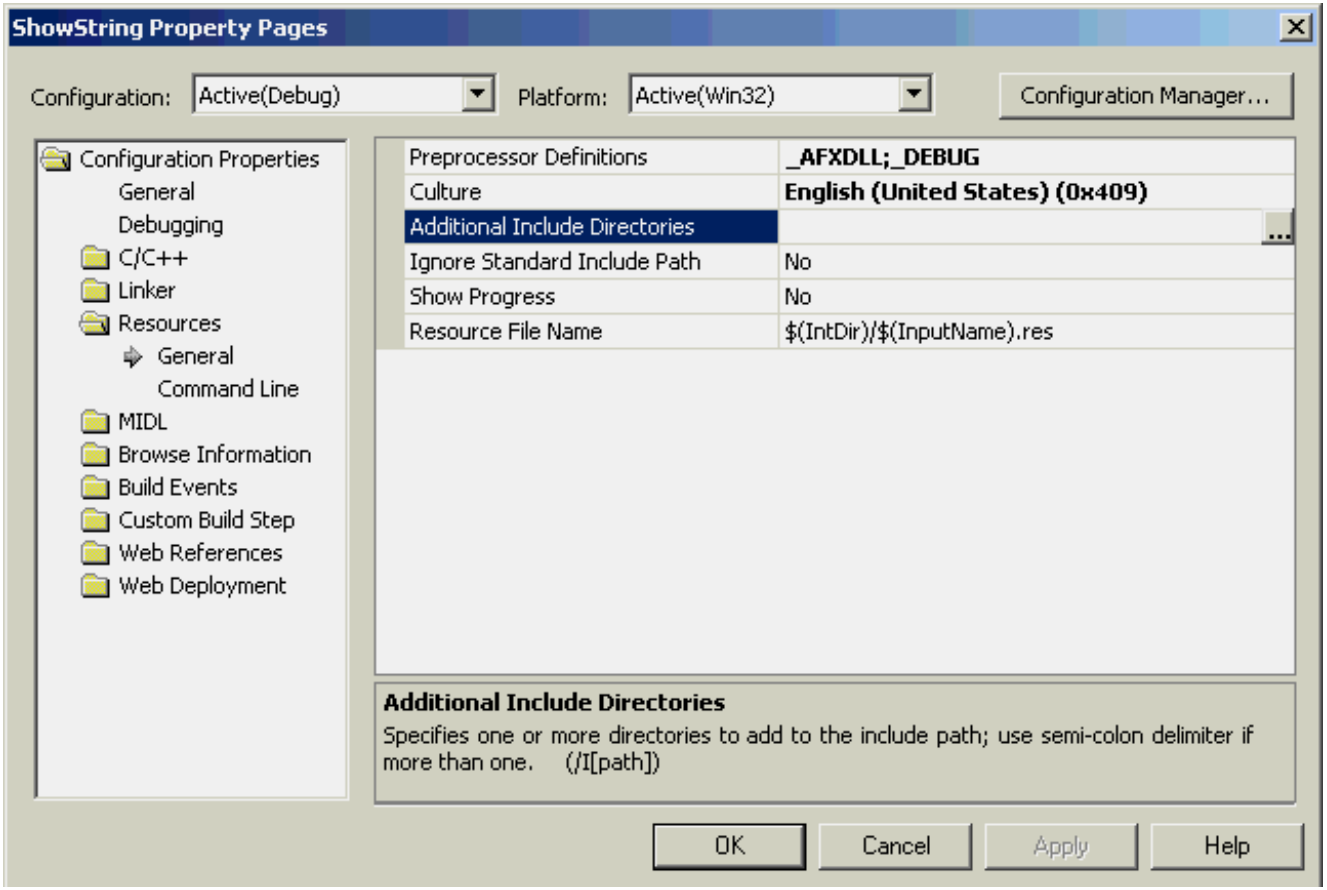
Erzeugen einer Klasse



Die Nutzung des Code-Assistenten



Anpassen der Entwicklungsumgebung



Die Klassenbibliothek

- Object
  - Magnitude
    - Character
    - Date
    - Number
    - Integer
    - Decimal
    - Float
    - Fraction
    - Time
  - Collection
    - Set
    - Array
    - SeqCollection
      - OrderedCollection
      - SortedCollection
      - Dictionary
  - Item
    - ControllerItem
    - ButtonCtrl
    - CheckBoxCtrl
    - ListCtrl
    - ComboCtrl

Eigenschaften von OOA/D-Tools

- Notation
- Stabilität am Markt
- Code-Generierung und Reverse-Engineering
- Handhabung des Werkzeugs
- Abbildung UML und Notationsfeinheiten
- Konsolidierung der Diagramme
- Spezifikationsfenster
- Repository/Code-Basiert/Datenbank-Speicherung
- Eigene Script-Sprache - OCL
- Mehrbenutzer
- Versionsmanagement
- Report-Generierung
- Zusatzwerkzeuge

3

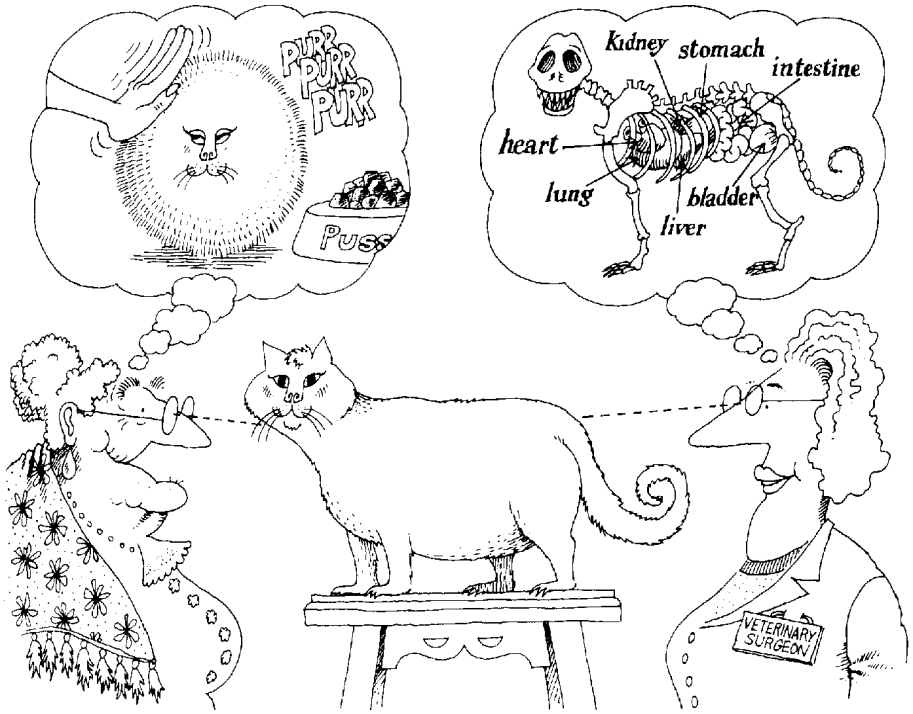
OBJEKTORIENTIERTE ANALYSE

Generelle Vorgehensweise der Analyse

- Was soll das System tun?
  Anforderungsanalyse
  - Projektziel definieren
  - Verstehen des Problems
  - Informationen sammeln (Ereignisse, Geschäftsprozesse)

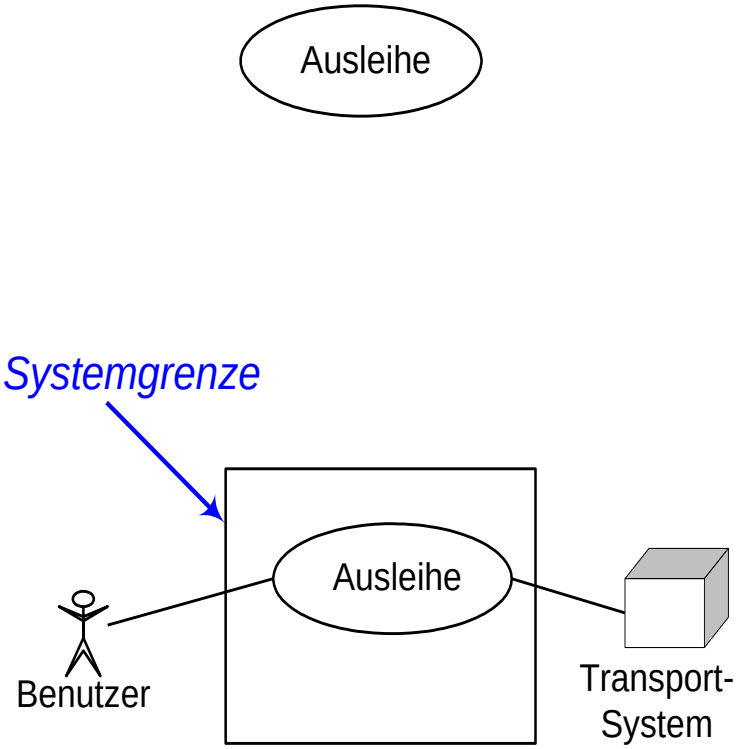
- Womit geht das System um?
  Fachliche Analyse
  - Identifizieren, Abstrahieren und Beschreiben essentieller Objekte
  - Statische Strukturen bestimmen
  - Verantwortlichkeiten und Ereignisse zuordnen
  - Prozesse beschreiben, Kommunikationsstrukturen

Abstraktion der Realität



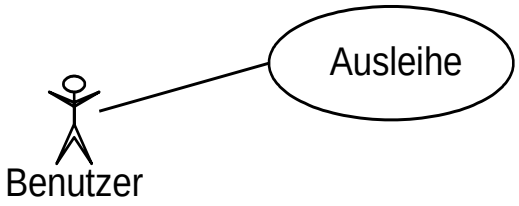
Anwendungsfälle

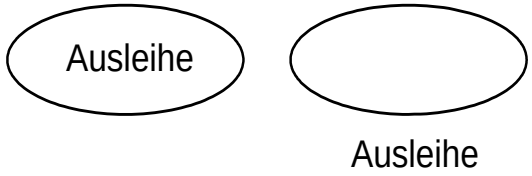
- Anwendungsfall
  - beschreibt eine Menge von Aktionen
  - erzeugt Ergebnis für Akteure
  - wird durch Ereignis initiiert
- Akteur
  - liegt außerhalb des Systems
  - kann ein Mensch oder ein anderes System sein
  - ist an Anwendungsfällen beteiligt
  - primärer Akteur: interessiert am Ergebnis des Anwendungsfalls
  - unterstützender Akteur: wird vom System benutzt

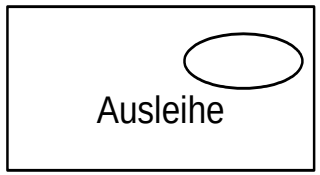


Anwendungsfälle - Regeln und Darstellung

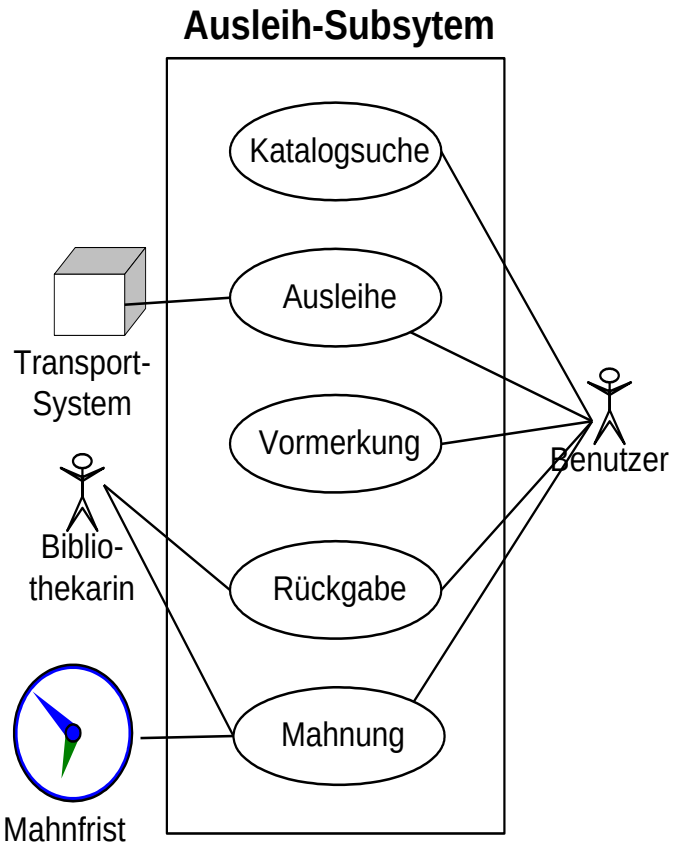
- Regeln für Anwendungsfälle
  - mindestens ein Akteur beteiligt
  - ein auslösender Akteur vorhanden
  - Anwendungsfälle produzieren ein fachliches Ergebnis für den primären Akteur
- Darstellung eines Anwendungsfalls
  - Text in Ellipse
  - Text unter Ellipse
  - Rechteck mit kleinem Ellipsen-Icon



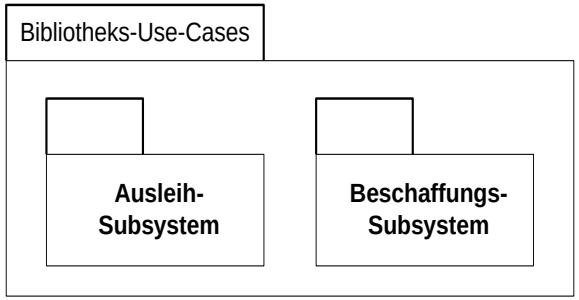




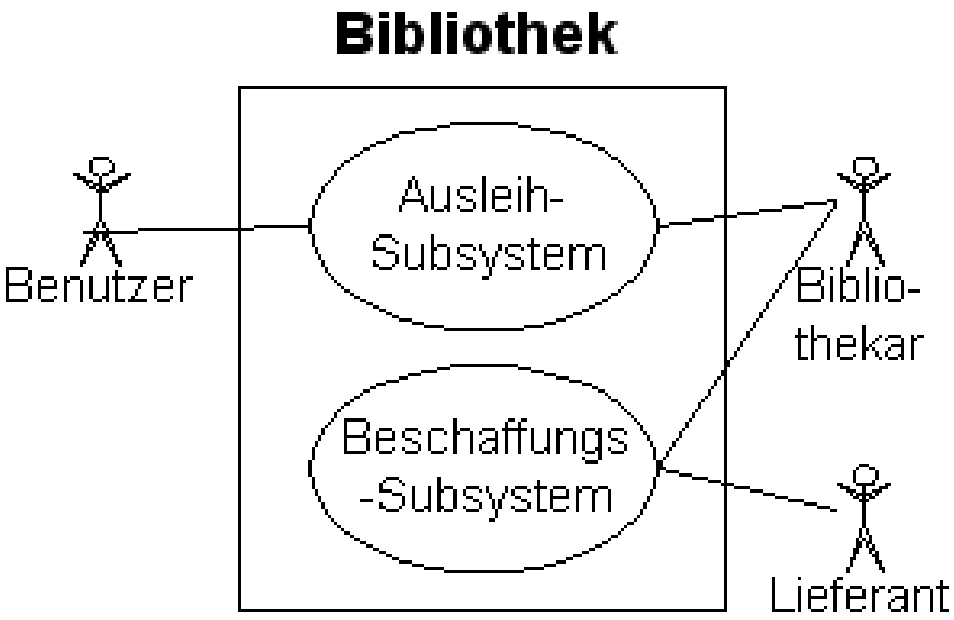
Anwendungsfalldiagramm



- Anwendungsfalldiagramm zeigt
  - Anwendungsfälle
  - Akteure
  - deren Zusammenhänge
- Detaillierung
  - mittels Paketbildung



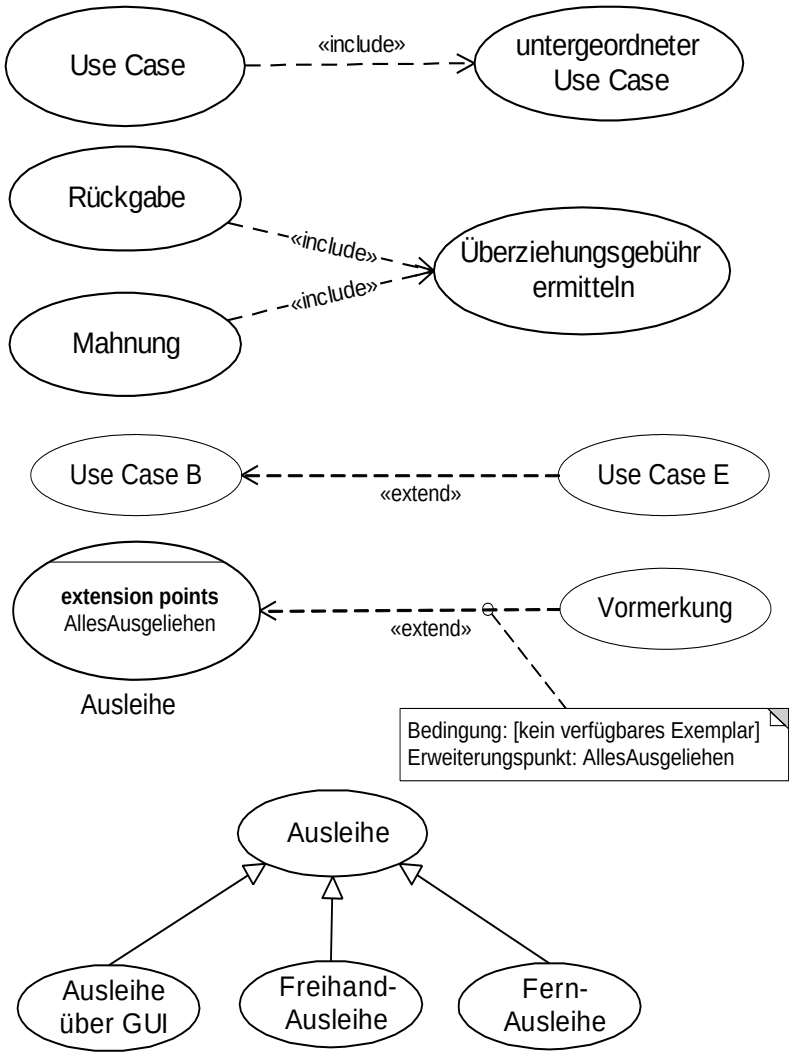
Gliederungsmöglichkeiten



Schablone zur Beschreibung

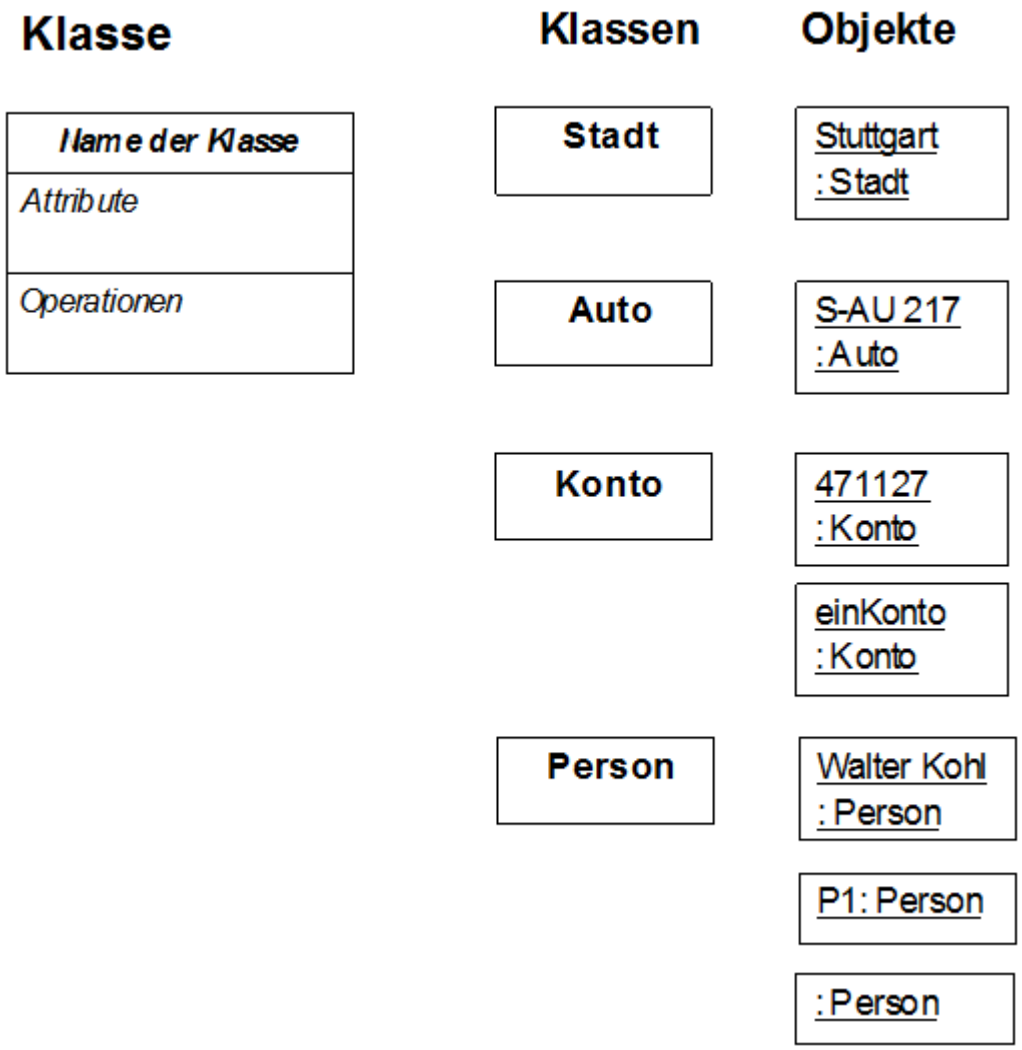
- **Zweck einer Schablone**
  - strukturierte Beschreibung der Anwendungsfälle
- **Aufbau einer einfachen Schablone**
  - Nummer und Name des Anwendungsfalls
  - Kurzbeschreibung:... Ein Satz, worum es geht
  - Akteure:... Auflistung beteiligter Akteure
  - auslösendes Ereignis:... Wie wird der Anwendungsfall ausgelöst?
  - Vorbedingungen:... Systemzustand vorher
  - Ergebnisse:.... für den Akteur
  - Nachbedingungen:... Systemzustand nachher
  - Ablaufbeschreibung:... Reihenfolge der einzelnen Aktivitäten
  - Variationen und Fehlersituationen:... Alternative Abläufe incl. Fehler
  - Anmerkungen / offene Fragen:... Fehlen Informationen? Getroffene Entscheidungen ...
  - Dokumentenverweise:... Beschreibungen, Protokolle, Diagramme,...

Beziehungen zwischen Anwendungsfällen



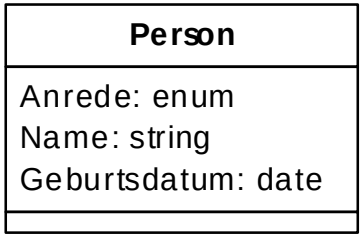
- **<<includes>>**
  - ein Anwendungsfall kommt innerhalb eines anderen vor
- **<<extends>>**
  - ein Anwendungsfall erweitert unter bestimmten Bedingungen einen anderen
- **Spezialisierung**
  - Allgemeiner Anwendungsfall erfährt verschiedene Spezialisierungen

Objekte / Klasse

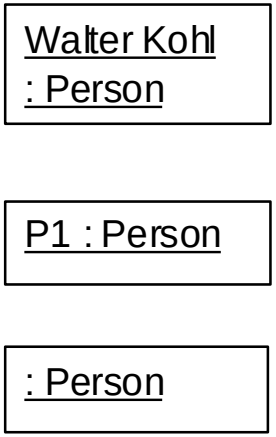


Klasse / Objekte mit Attributen

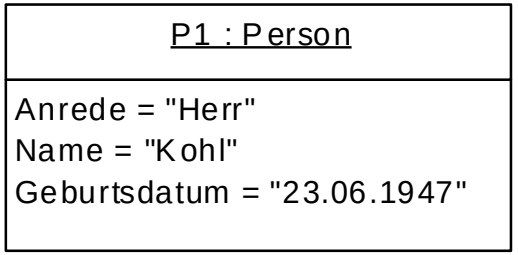
Klasse mit Attributen



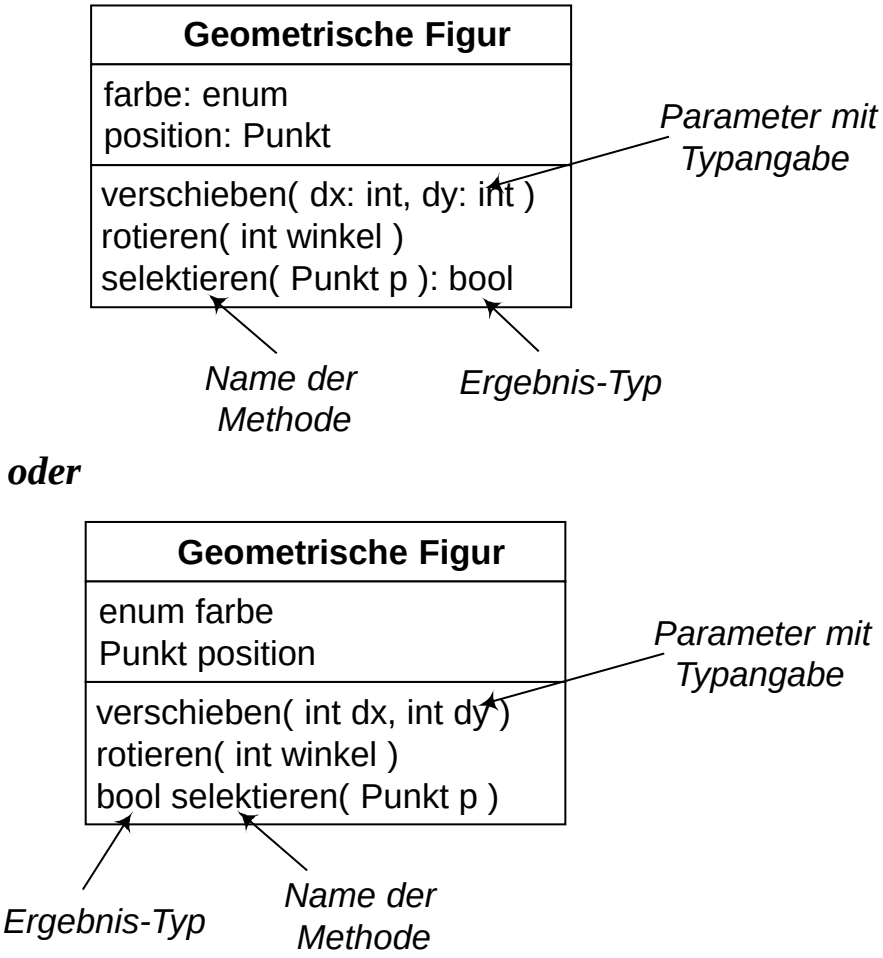
Objekt



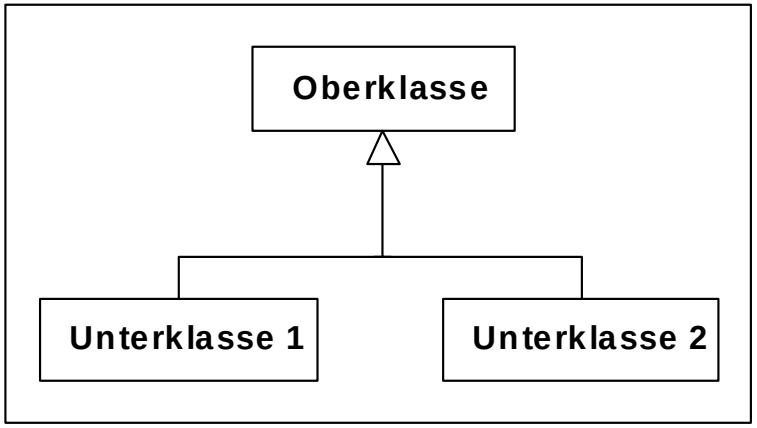
Objekt mit Attributen



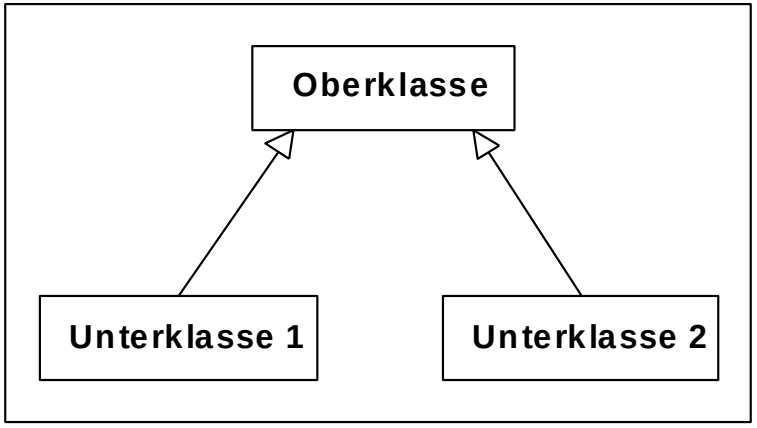
Operationen und Methoden



Generalisierung / Spezialisierung / Vererbung

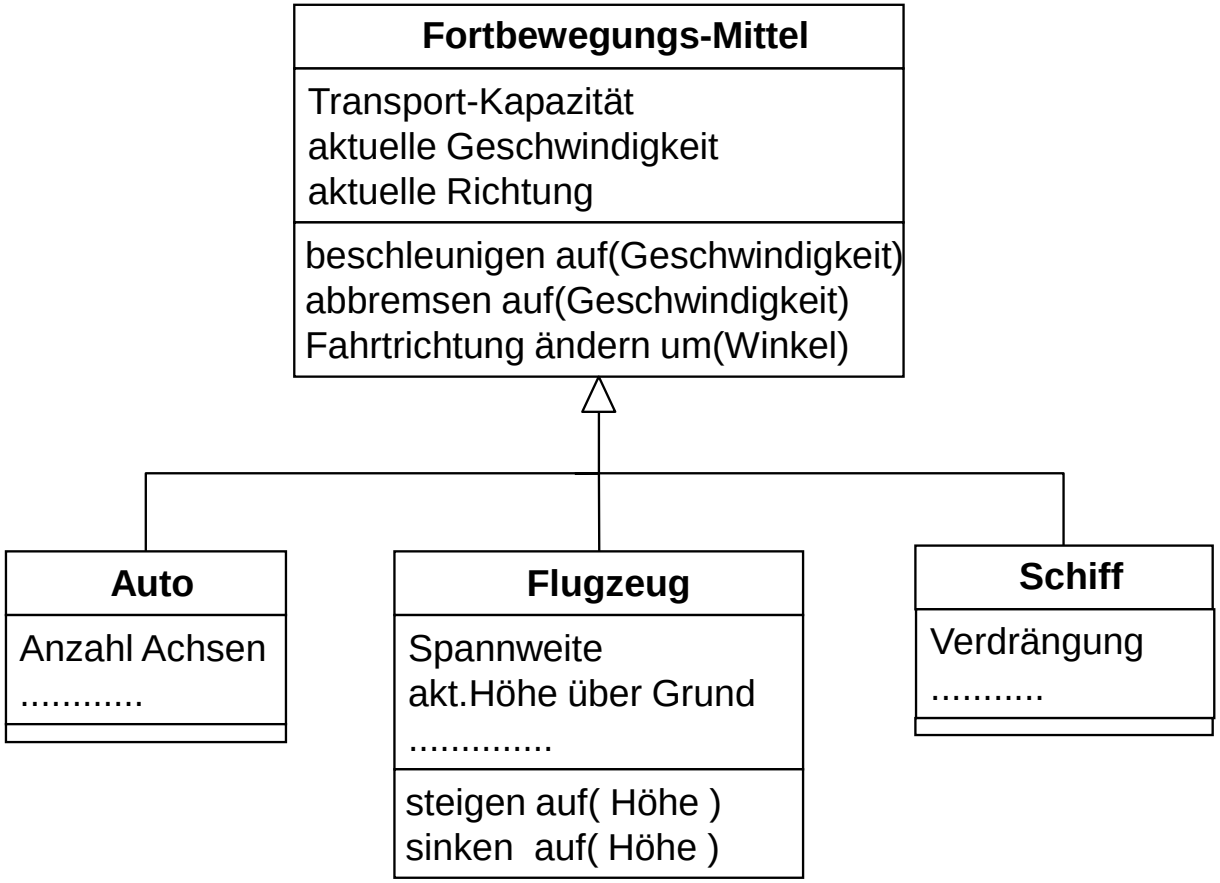


oder

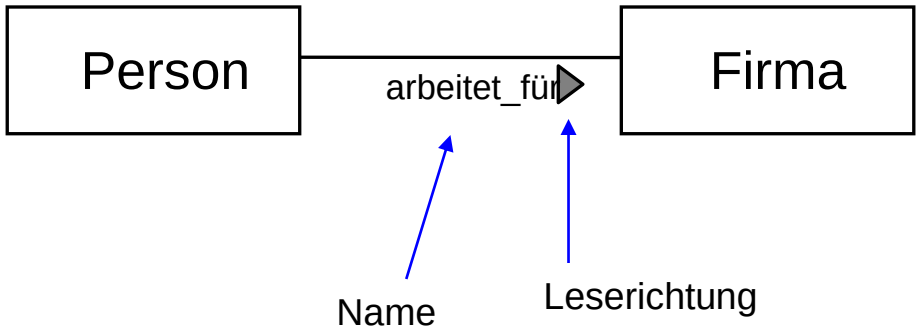


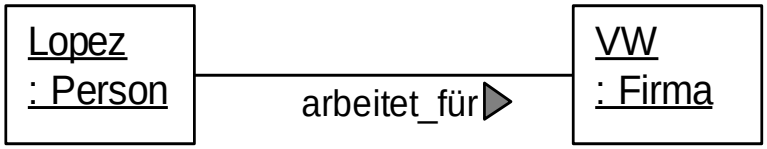
- **Generalisierung**
  - alle Attribute, Assoziationen und Operationen werden vererbt.
  - Im Design genauere Bestimmung, was vererbt wird.

Beispiel Generalisierung



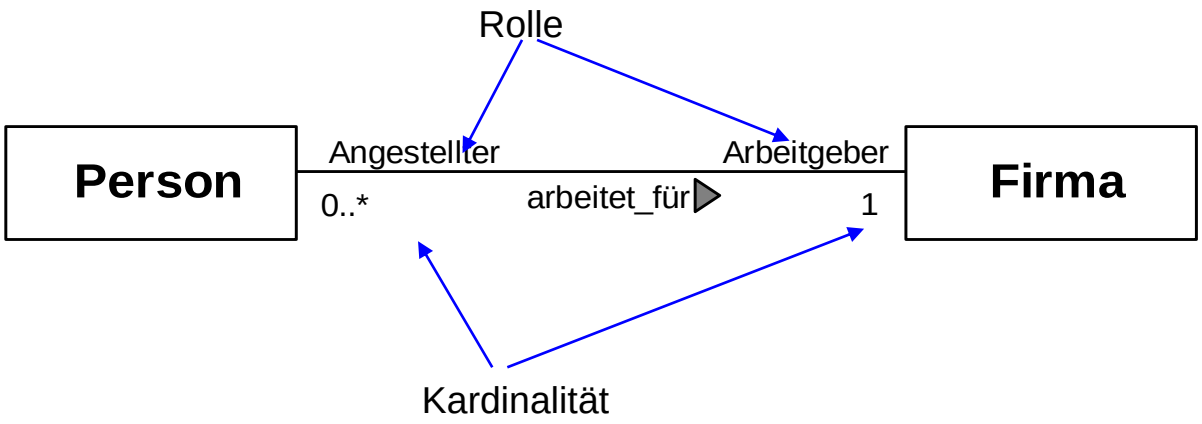
Assoziation





- Assoziation
  - beschreibt eine Beziehung zwischen zwei Klassen
  - dient oft der Objektkommunikation
  - Objektverbindungen sind „Instanzen“ einer Assoziation

Rollen und Multiplizitäten

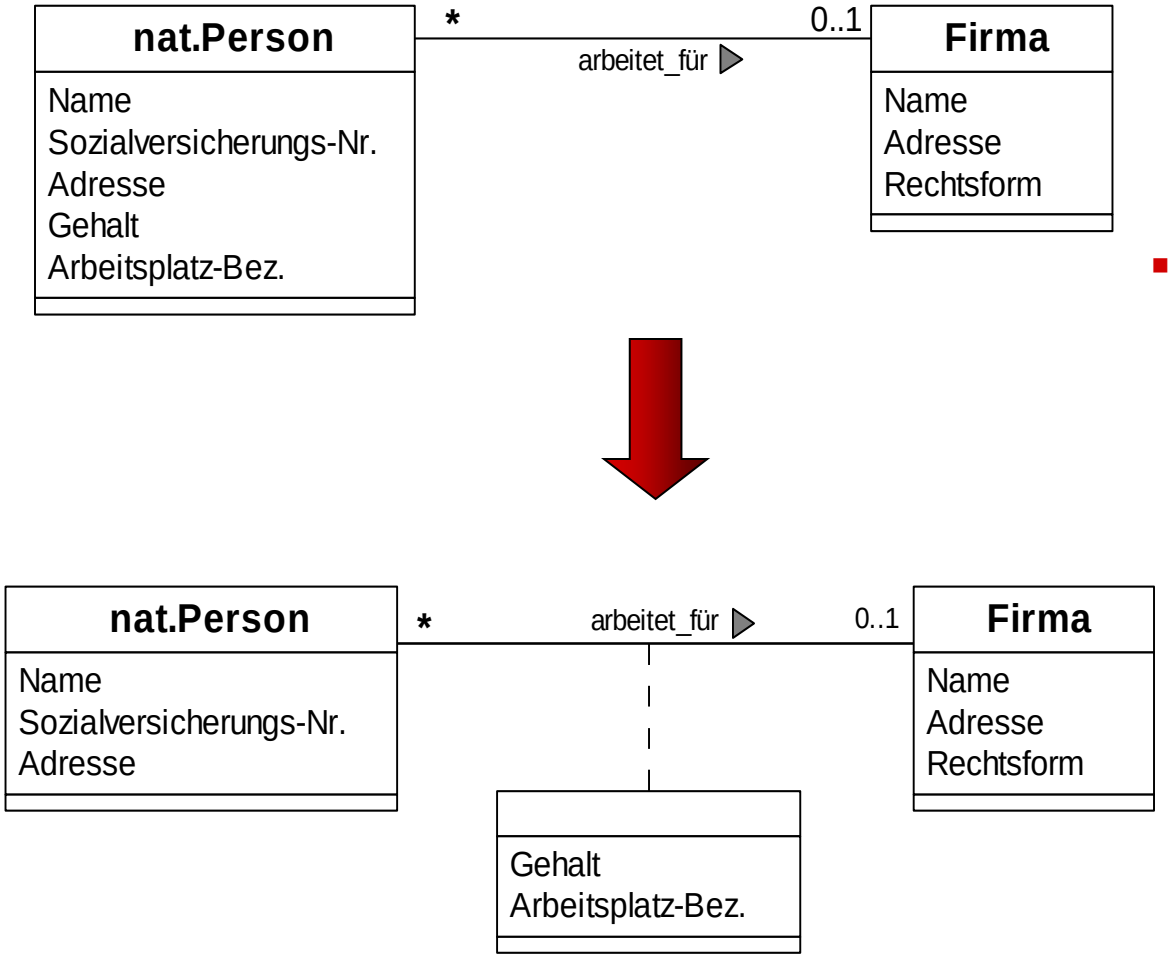


- **Rolle**
  - beschreibt die Rolle eines Objektes für diese Assoziation
  - kann zur Codegenerierung genutzt werden (Person.Arbeitgeber Firma.Angestellter)
- **Kardinalität**
  - beschreibt das Mengenverhältnis zwischen Objekten der Klassen

Beispiele für Kardinalitäten:

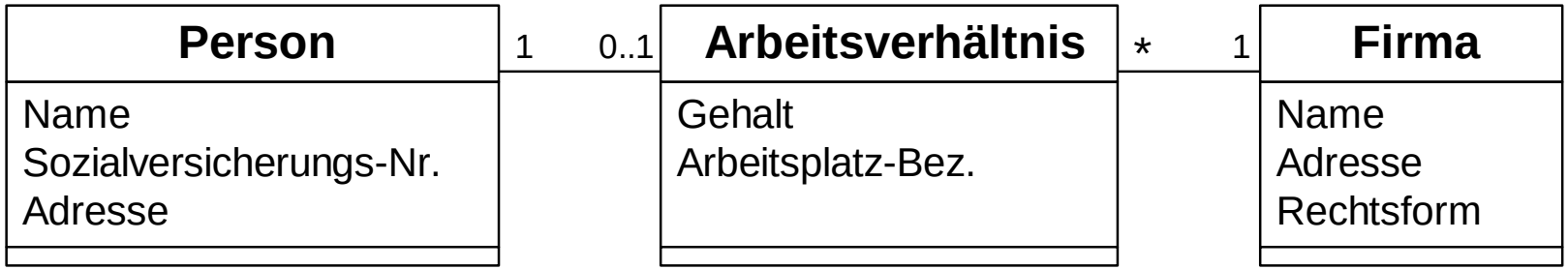
1	genau eins
0..1	null oder eins
*	beliebig viel (incl. Null)
2..8	zwischen zwei und acht (incl.)
1..*	mindestens eins

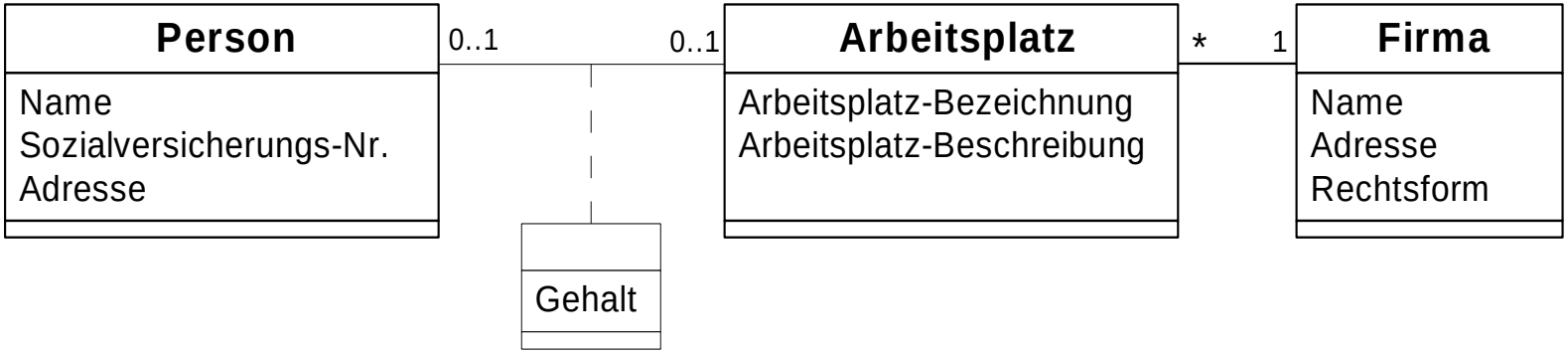
Modellierung mit Assoziationsklasse



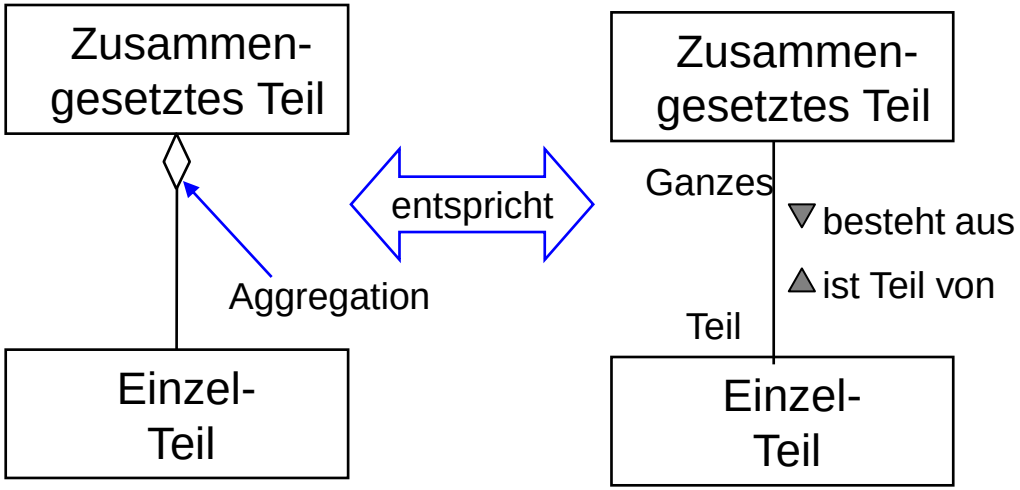
- gibt es Attribute in einer Klasse, die ohne Assoziation ihren Zweck verlieren?

Weiterführende Modellierung des Arbeitsverhältnisses

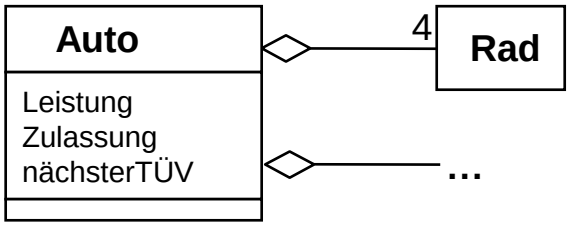




Aggregation

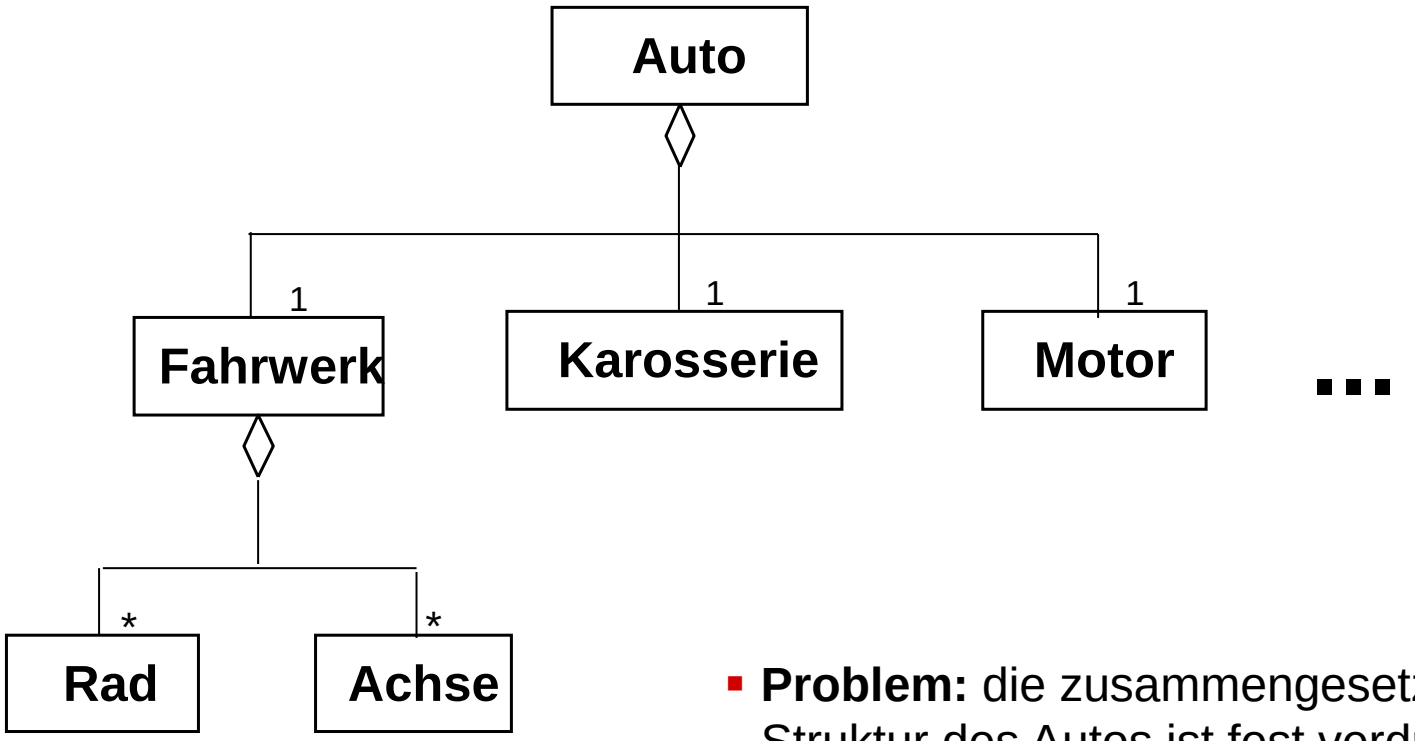


- **Aggregation**
  - ist eine Sonderform der Assoziation
  - modelliert die besteht-aus-Beziehung



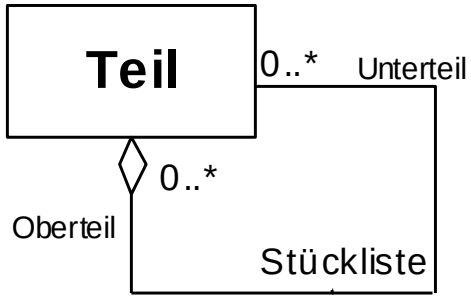
konzeptionelles Diagramm für die Struktur eines Dokuments

Beispiel Aggregation: Auto

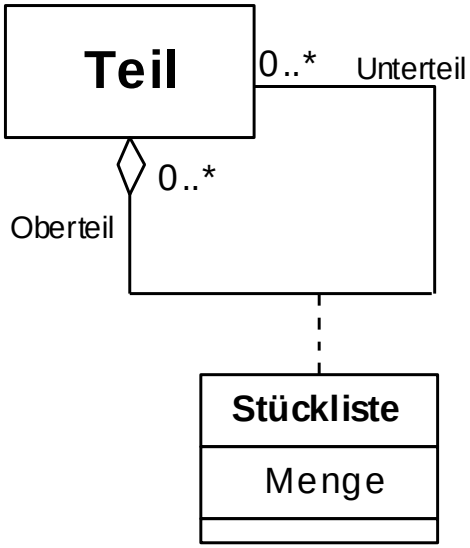


- **Problem:** die zusammengesetzte Struktur des Autos ist fest verdrahtet

Modellierung als Stückliste

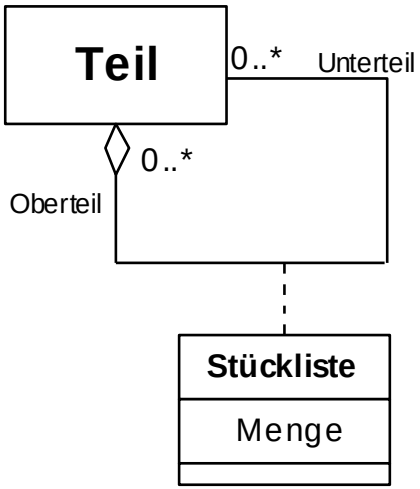


bzw.

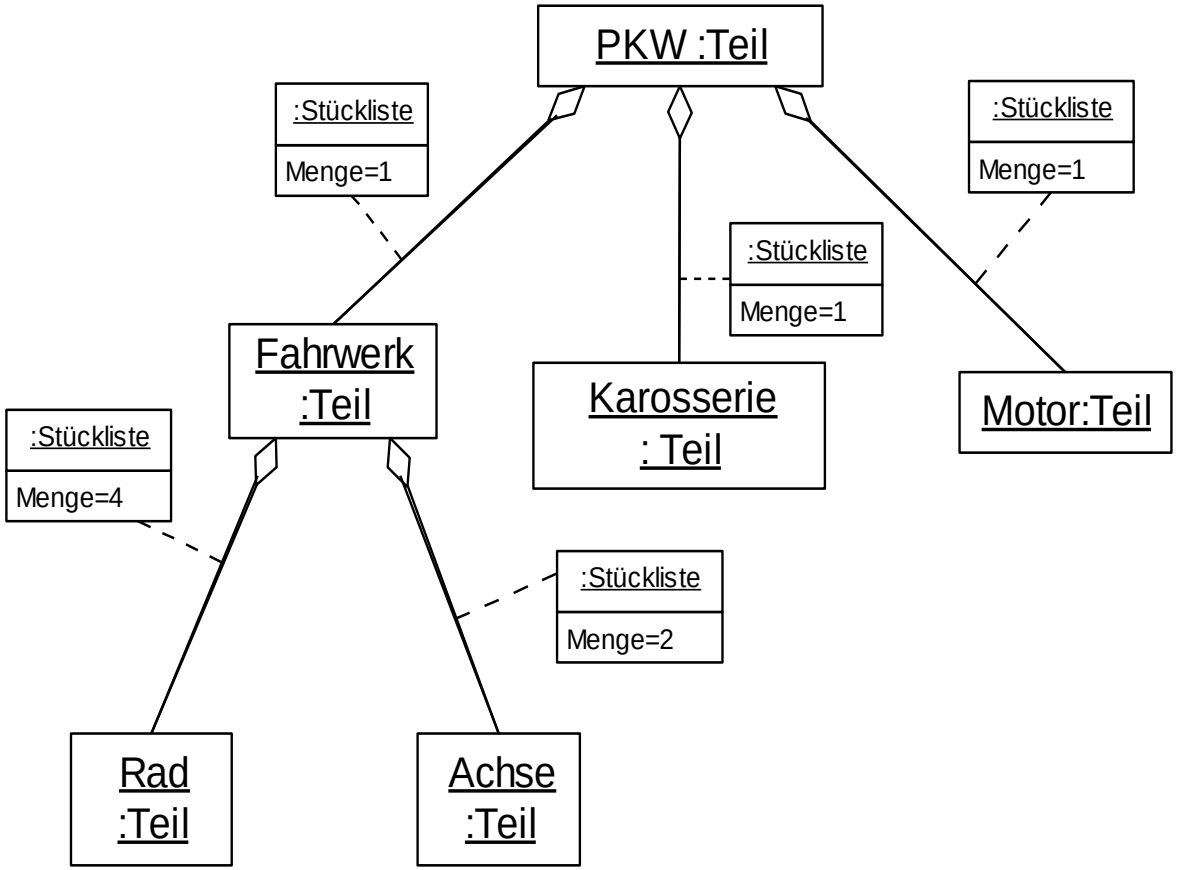


Modellierung als Stückliste

Klassenmodell

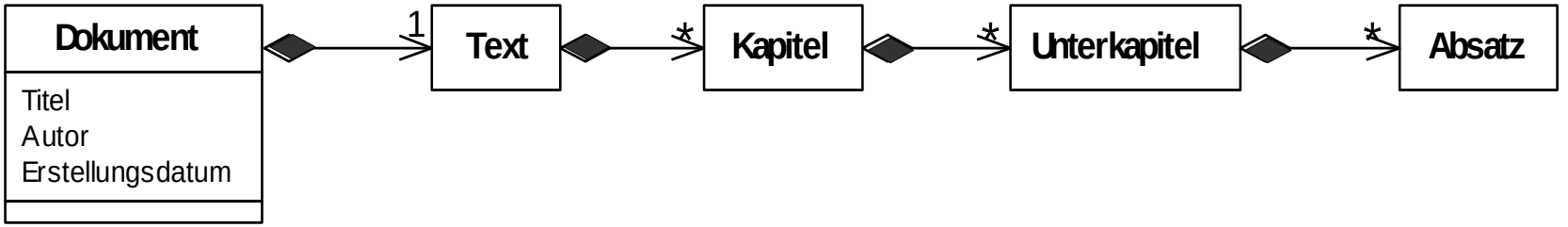


Beispiel auf Objekt-Ebene



Komposition

- **Komposition**
  - ist eine Sonderform der Assoziation
  - Einzelteil ist existenz-abhängig vom Ganzen
  - oft in Kombination mit Navigationsrichtung

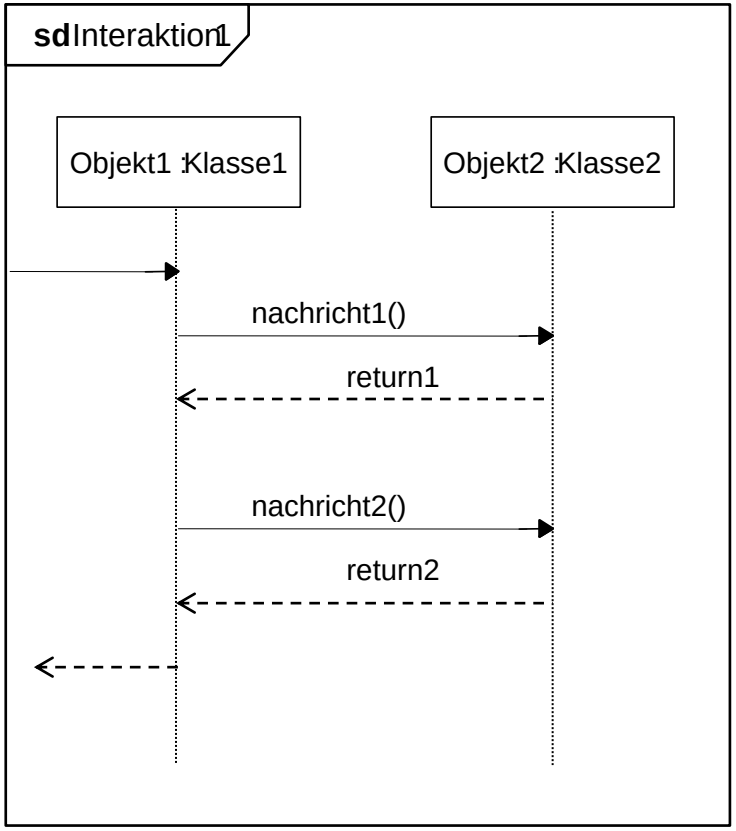


Erstellen des Klassendiagramms (Zusammenfassung)

- **Das Klassenmodell kann mittels folgender Aktivitäten erstellt werden, wobei die Reihenfolge variieren kann und iterativ vorgegangen werden muss:**

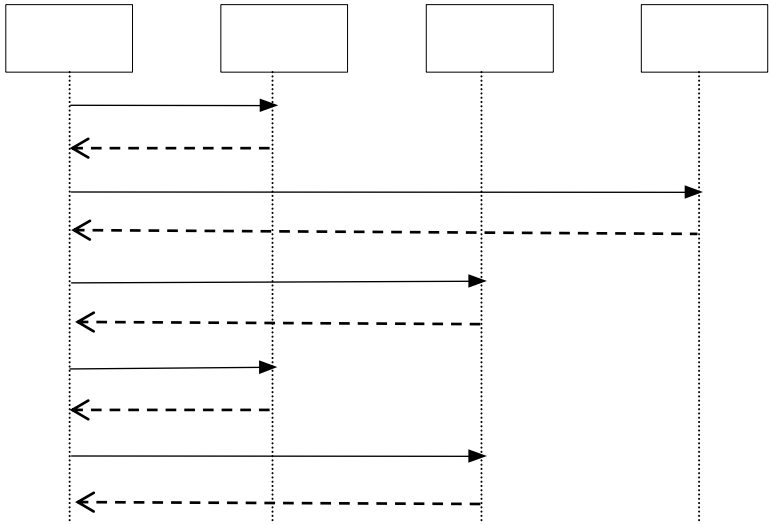
  1. **Identifizieren von Klassen**
  2. **Kandidaten für Klassen finden**
  3. **Klassen-Kandidaten überprüfen**
  4. **Glossar erstellen**
  5. **Assoziationen zwischen den Klassen finden**
  6. **Attribute sammeln**
  7. **Generalisierung**
  8. **Paketierung der Klassen bzw. bilden von Subsystemen**

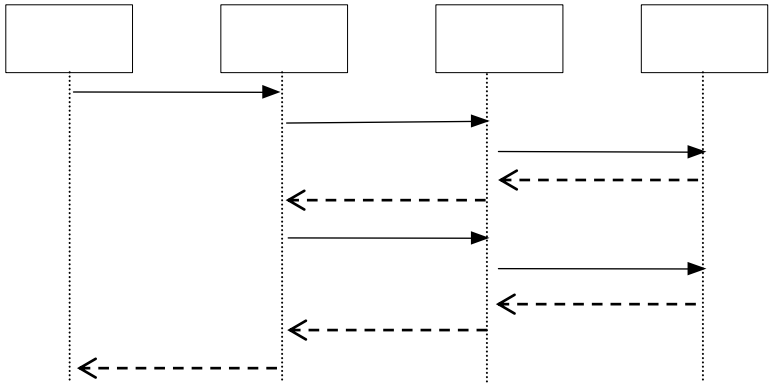
Sequenzdiagramm



- **Sequenzdiagramm**
  - zeigt eine Reihe von Nachrichten, die
  - ... eine ausgewählte Menge von Kommunikationspartnern (Objekte, Akteure, Systeme, ...) in
  - ... einer zeitlich begrenzen Situation austauschen
- **grundlegende Struktur**
  - Kommunikationspartner mit einer vertikalen Lebenslinie
  - Nachrichten als Pfeile zwischen Lebenslinien
  - Zeitablauf von oben nach unten
  - Rahmen mit Namen des Diagramms

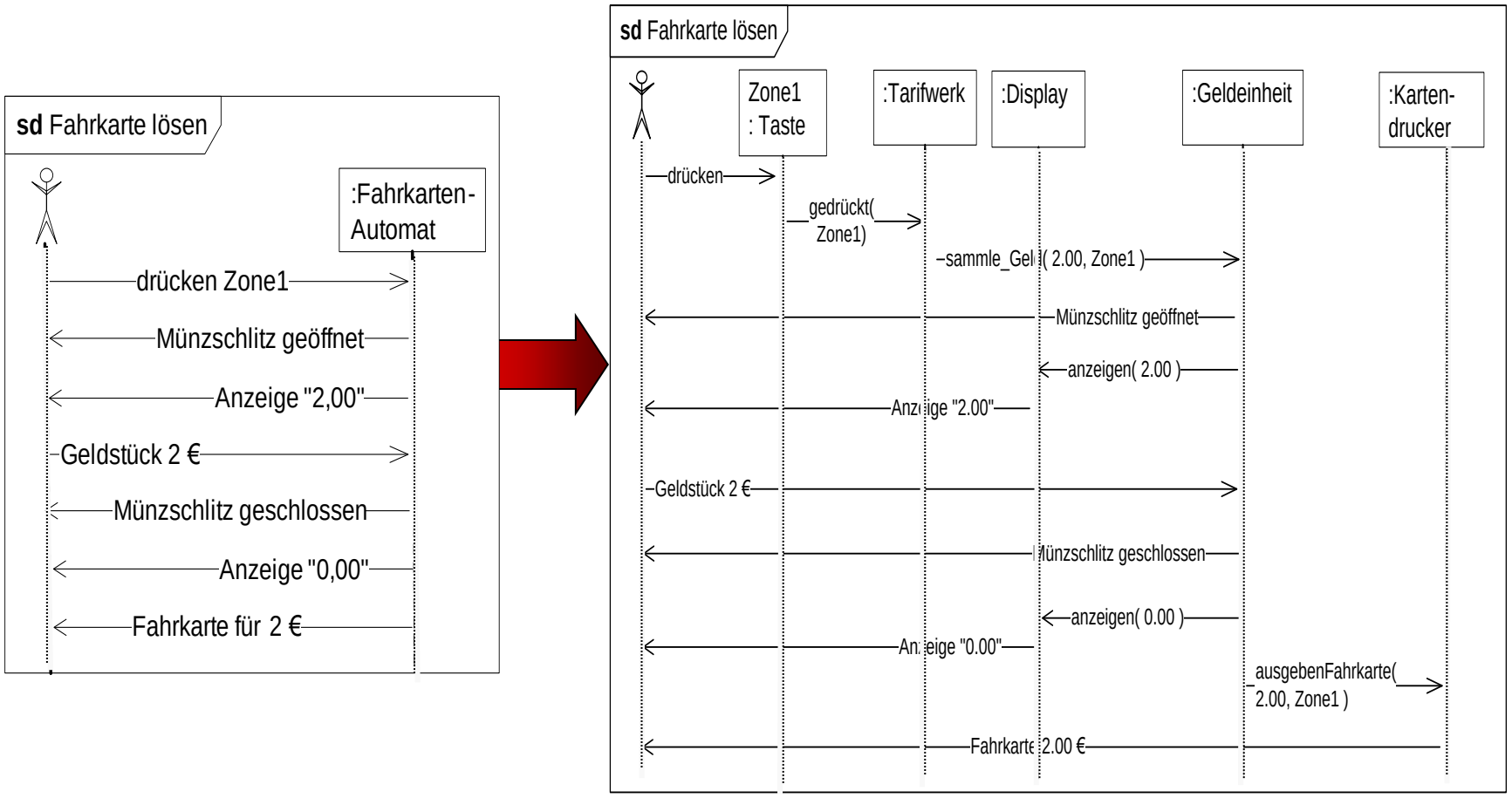
Sequenzdiagramm - zentrale und dezentrale Strukturen



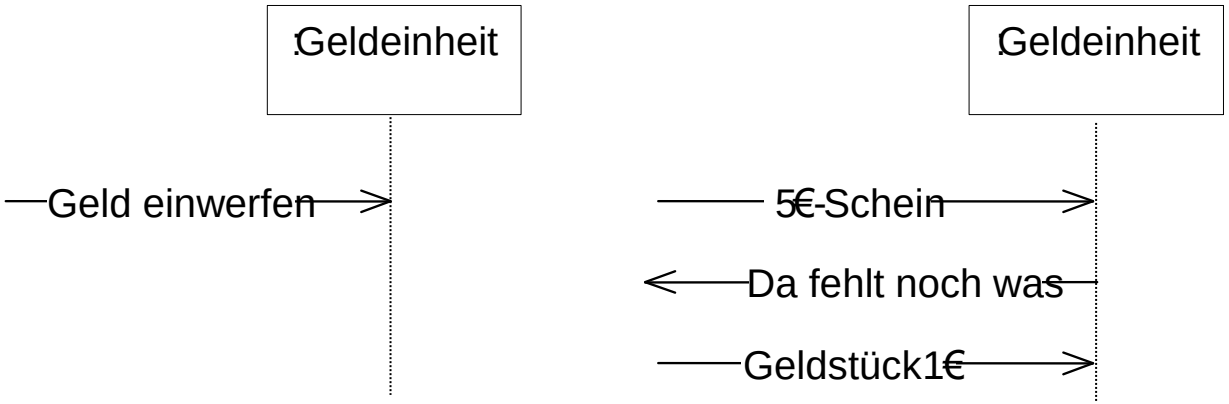


- **zentrale Struktur**
  - Ablaufsteuerung in einem Objekt (Controller)
  - Reihenfolge ist leicht variierbar
  - Vorteilhaft bei variablen Abläufen
- **dezentrale Struktur**
  - typisch für Aggregationen (Rechnung - Rechnungsposten - Artikel)
  - Reihenfolge der Nachrichten unflexibel
  - Vorteilhaft bei starren Abläufen (kein steuerndes Objekt notwendig)

Sequenzdiagramm - Verfeinerung eines „Objekts“

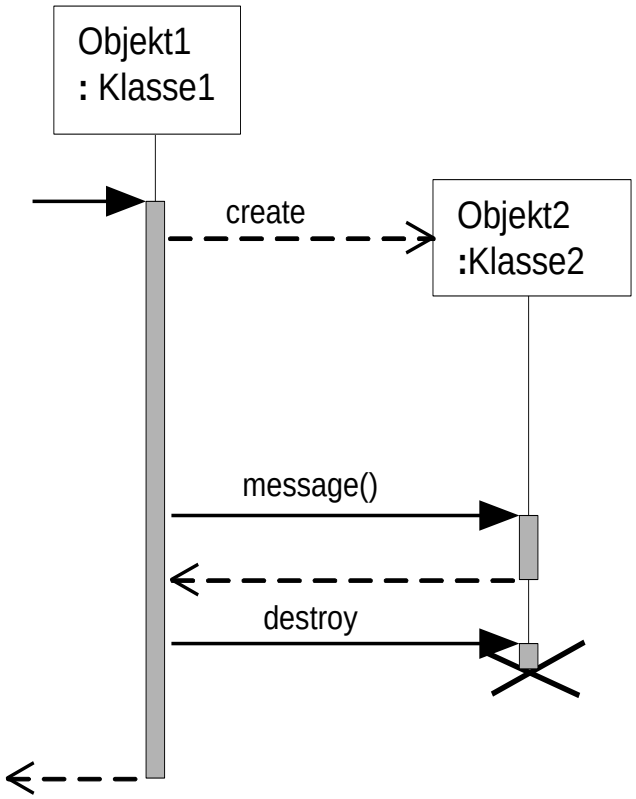


Sequenzdiagramm - Granularität von Nachrichten



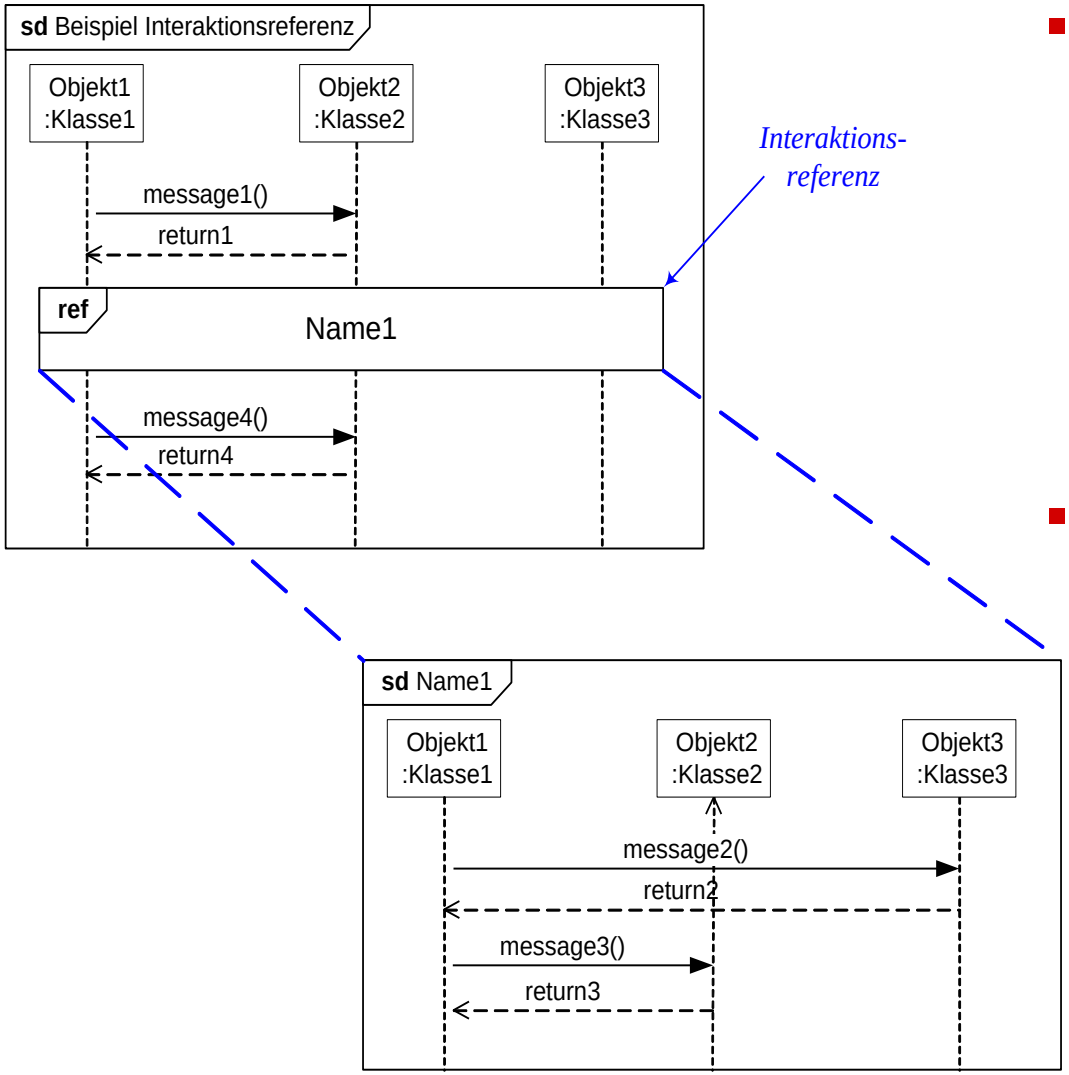
„Geld einwerfen“ entspricht auf der nächsten Stufe mehreren Nachrichten bzw. einem Dialog

Objekte erzeugen und zerstören



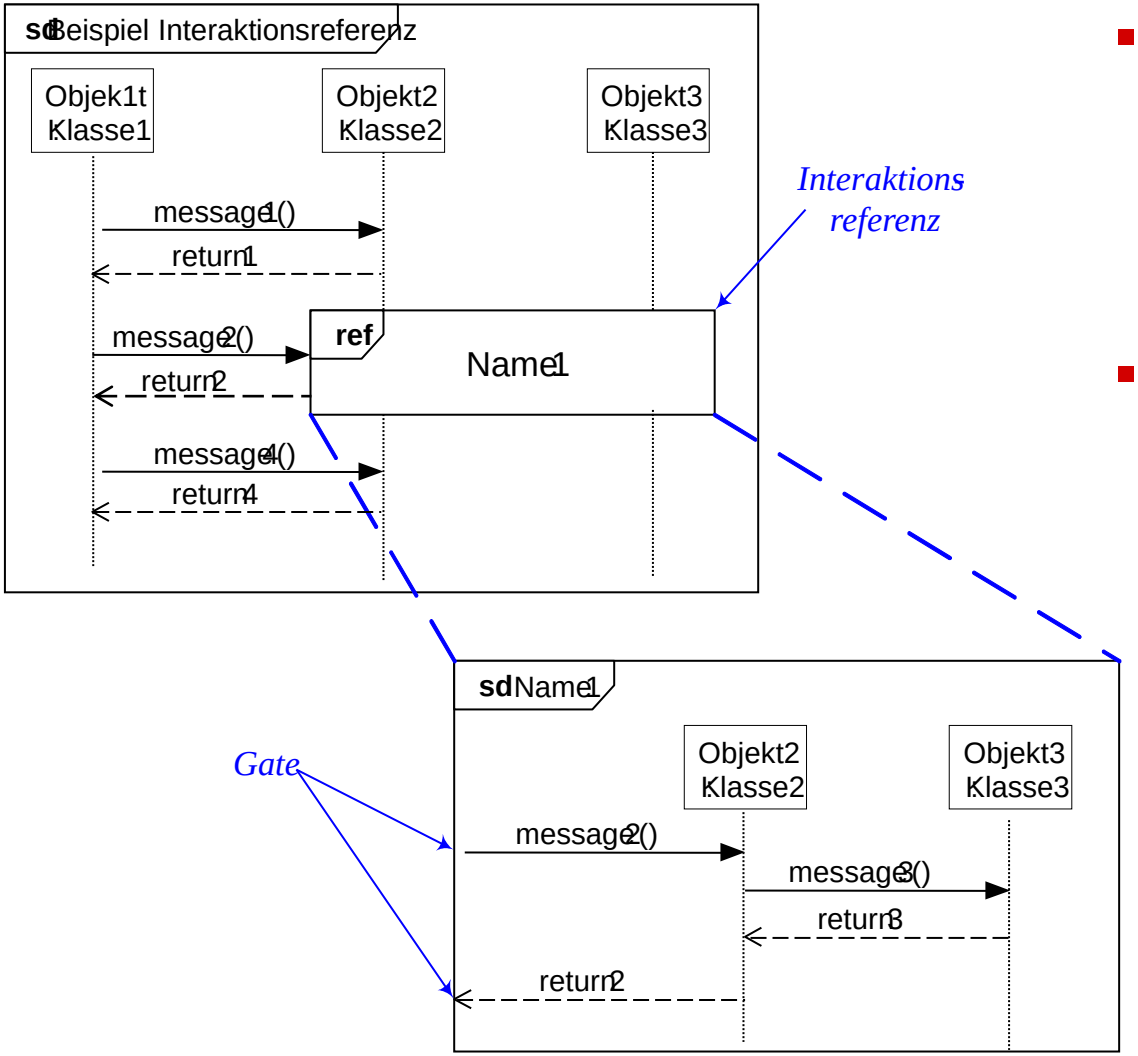
- **Erzeugen von Objekten**
  - eine Nachricht erzeugt das neue Objekt (Objekt2)
  - die Ablaufkontrolle bleibt bei dem erzeugenden Objekt (Objekt1)
- **Zerstören von Objekten**
  - die Lebenslinie eines Objektes wird durch ein Kreuz beendet
  - Zerstören wird durch Nachricht von Außen hervorgerufen oder das Objekt zerstört sich selbst

Sequenzdiagramm: Referenzen auf Unterdiagramme



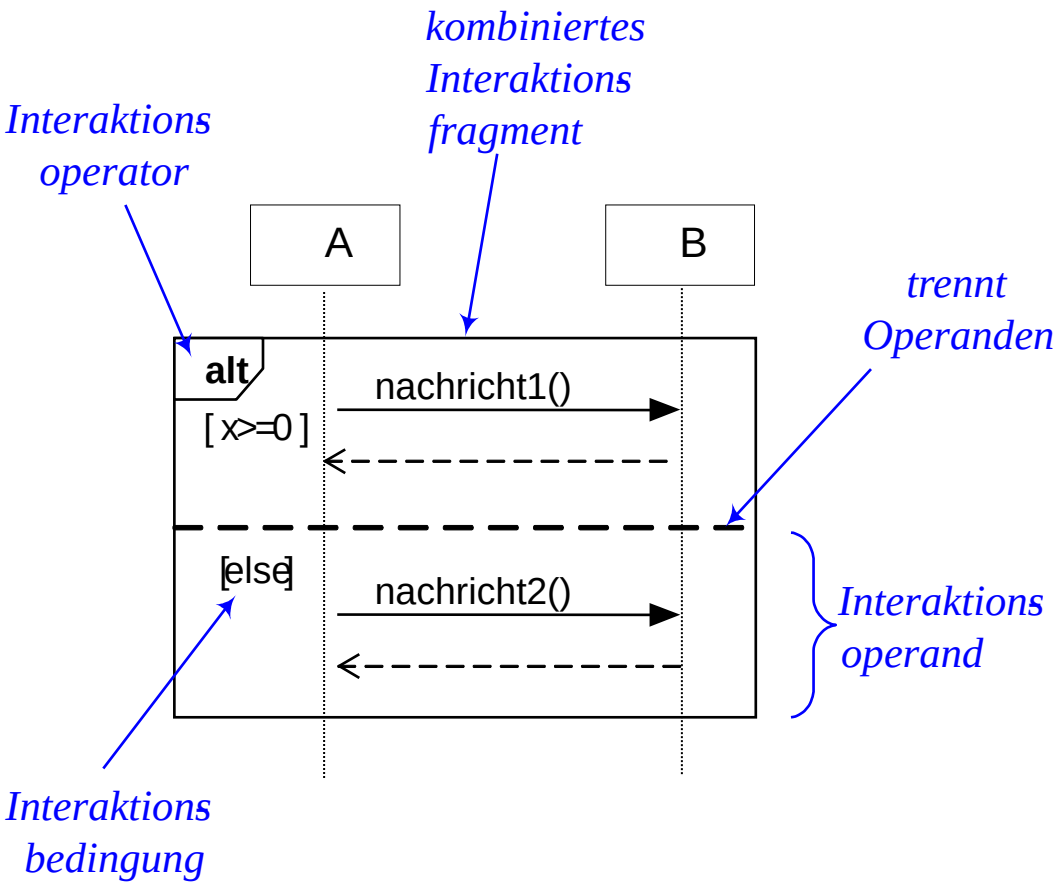
- Referenziertes Diagramm kann eingesetzt werden
  - Gleiche Abläufe in verschiedenen Diagrammen zusammenfassen
  - Komplizierte Detailabläufe in Detaildiagramme auslagern
- Beide müssen die gleichen Lebenslinien überdecken

Verknüpfungspunkte



- Ein Verknüpfungspunkt (Gate) ist ein Punkt auf einem Rahmen, zu dem eine Nachricht hin- oder wegführt.
- Ein Gate kann einen Namen erhalten.

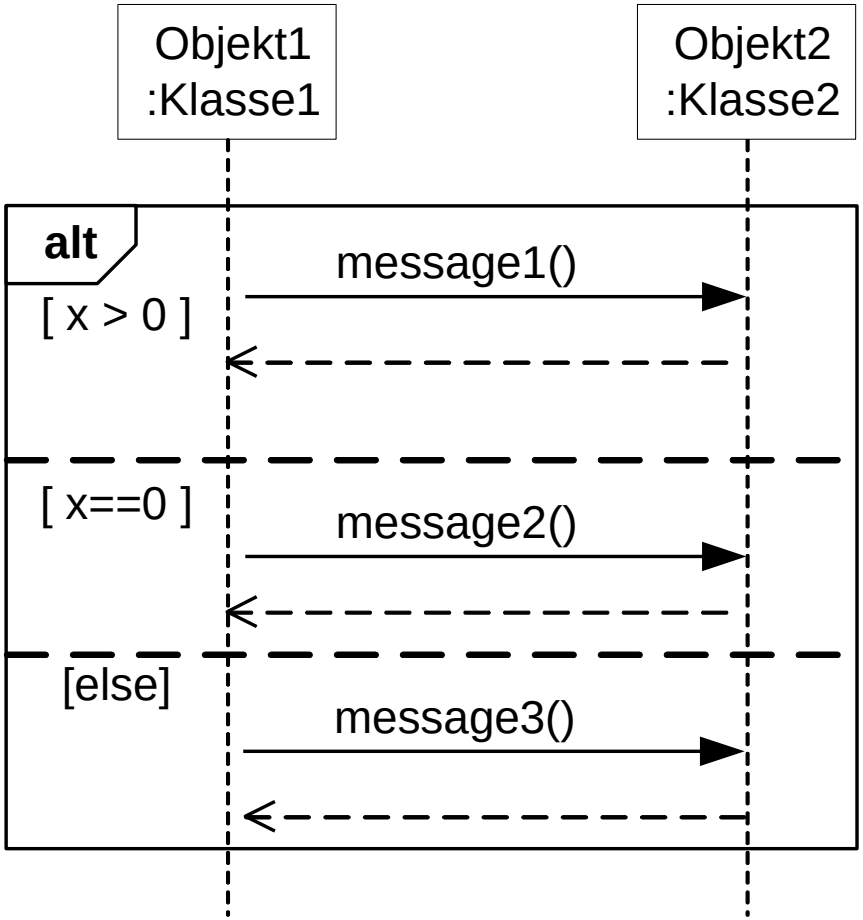
Kombinierte Fragmente



- Mit Fragmenten können
  - alternative Abläufe
  - optionale Abläufe
  - Schleifen
  - parallele Abläufe
  - etc.

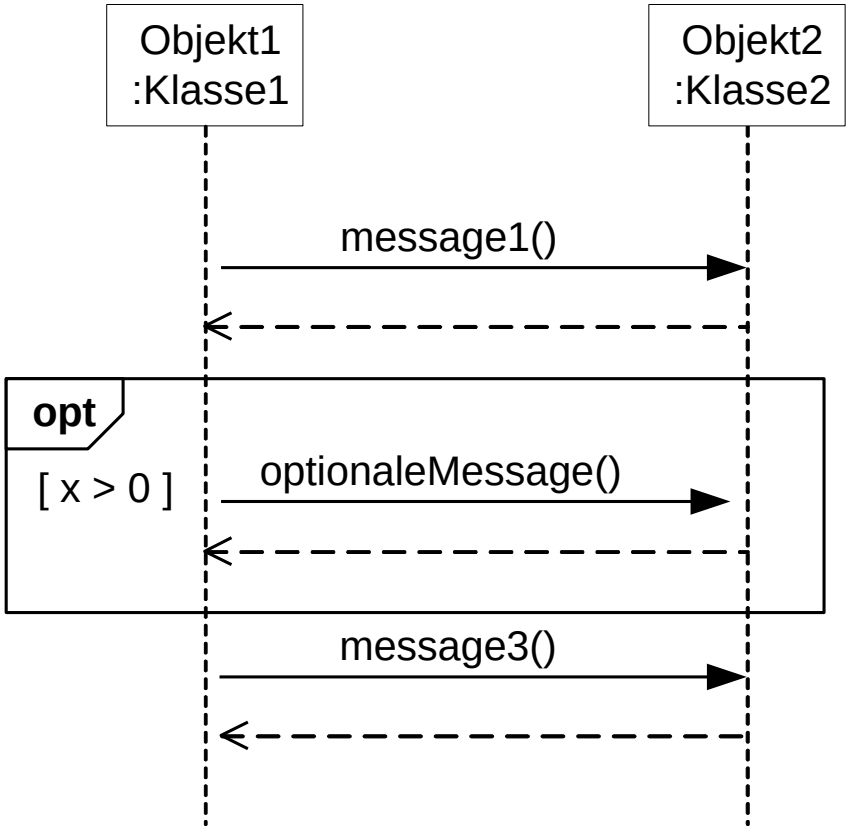
  modelliert werden.
- Ein Interaktionsoperand kann wieder kombinierte Fragmente enthalten (Schachtelung)

Fragment für alternative Abläufe



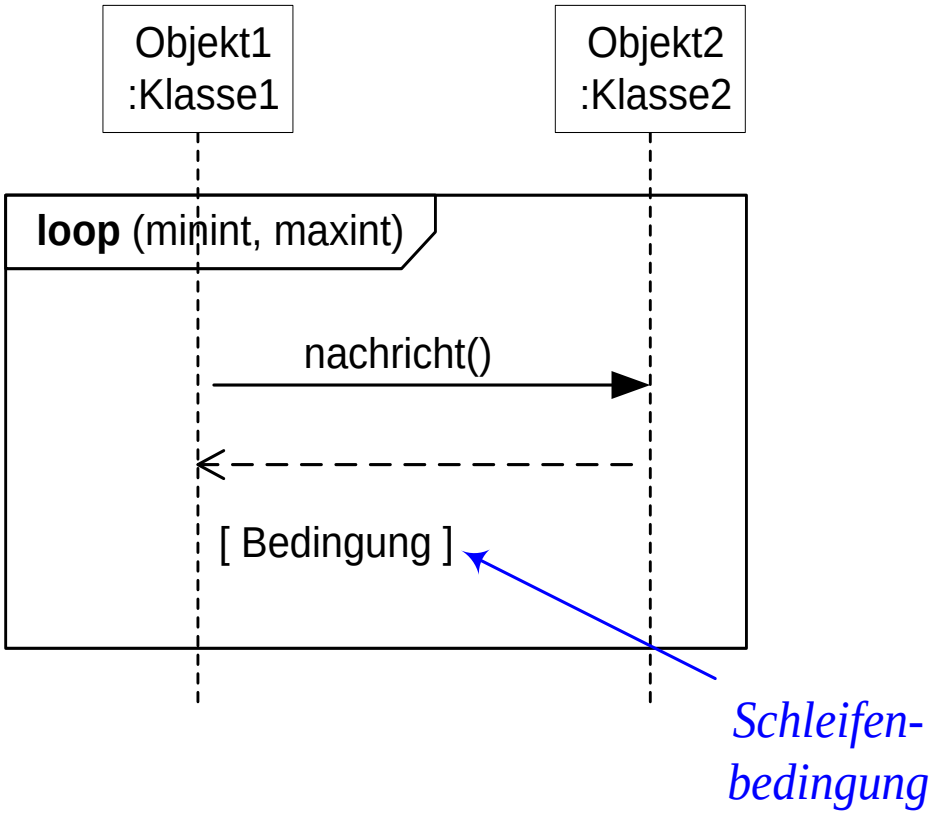
- **Interaktionsoperator alt**
  - pro Alternative ein Operand
  - pro Operand eine Bedingung, [else] als sinnvoller Schlussoperand
  - wie switch in OOPL

Fragment für einen optionalen Teilablauf



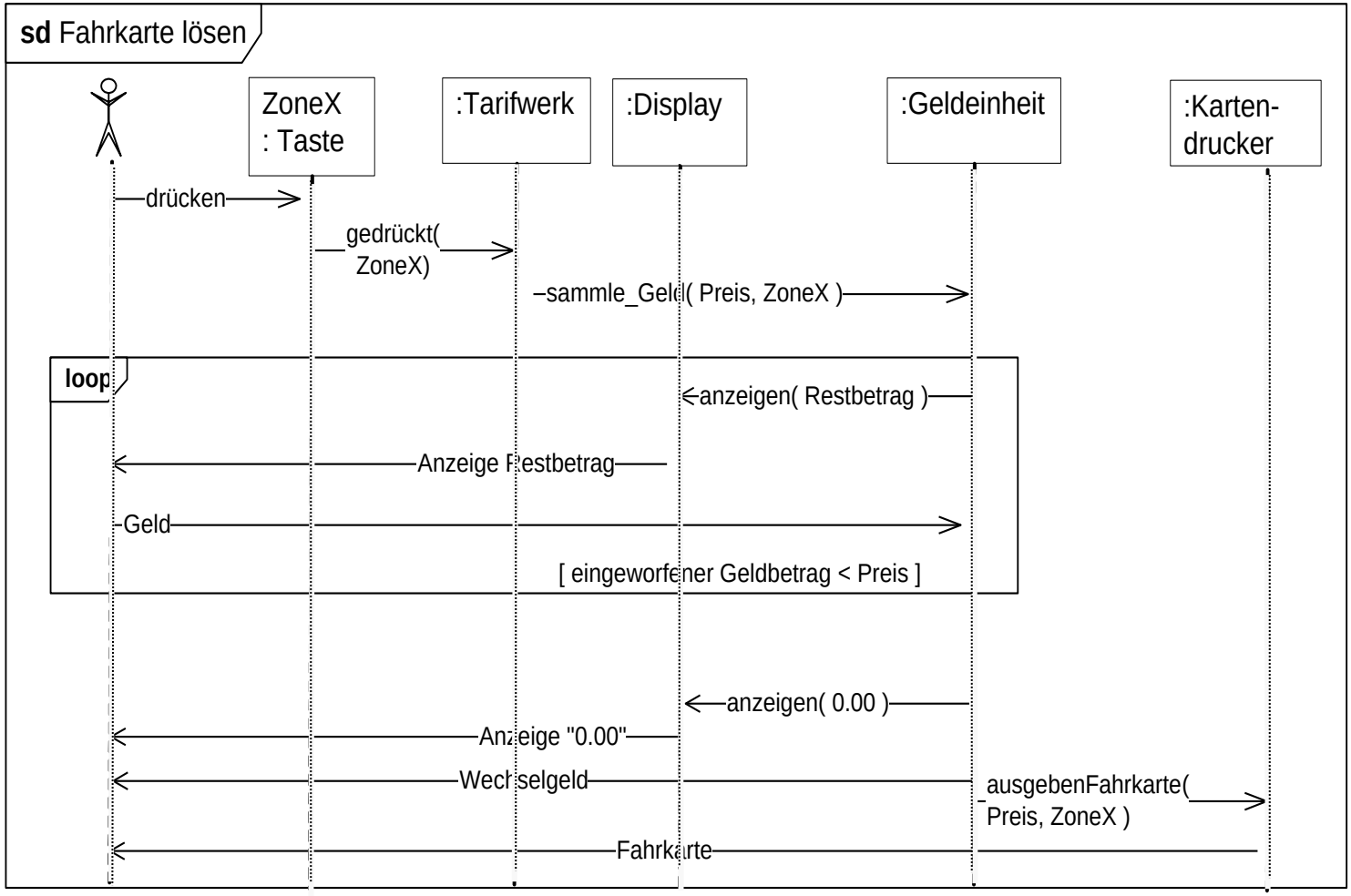
- **Interaktionsoperator opt**
  - ein Operand mit einer Bedingung
  - wie if-then in OOPL

Das Schleifen-Fragment

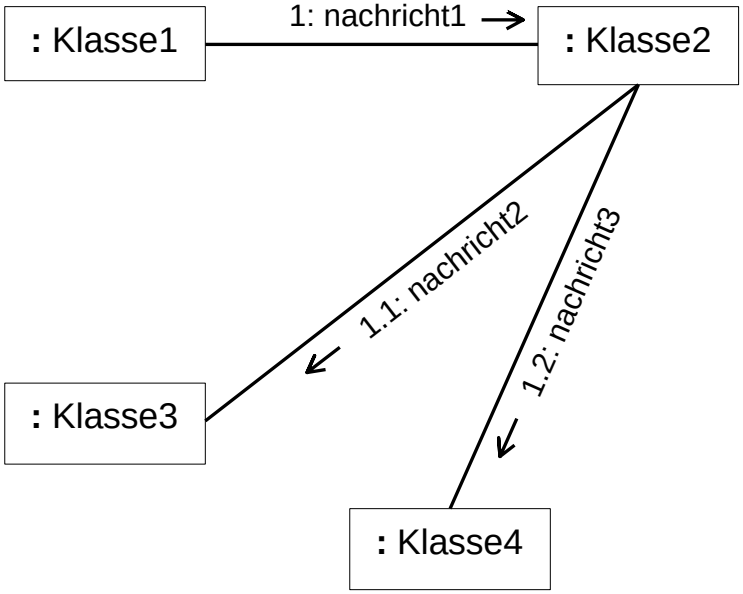


- Interaktionsoperator loop
  - besteht aus 1 Operanden
  - minint gibt die Mindestzahl der Wiederholungen an
  - maxint die Höchstzahl
  - Die Schleifenbedingung wird bei jedem Durchlauf der Schleife an der Stelle ihres Auftretens ausgewertet.

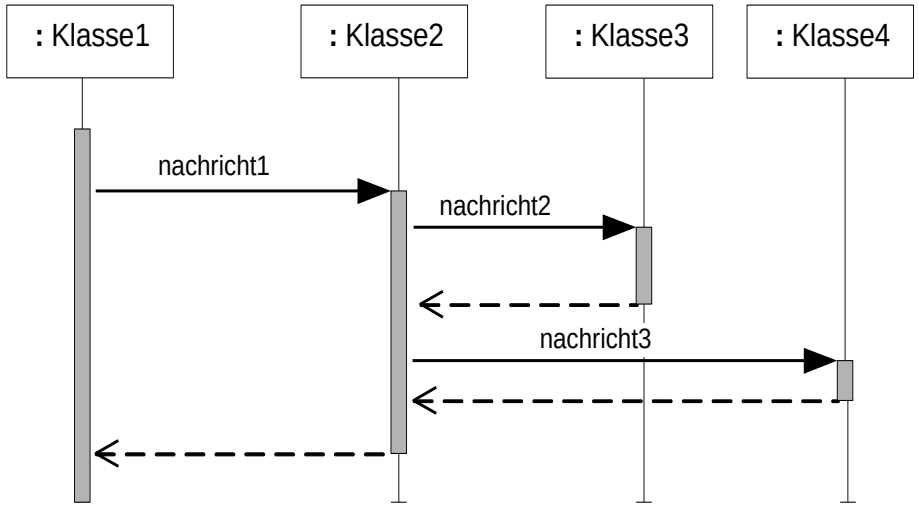
Beispiel für eine Schleife



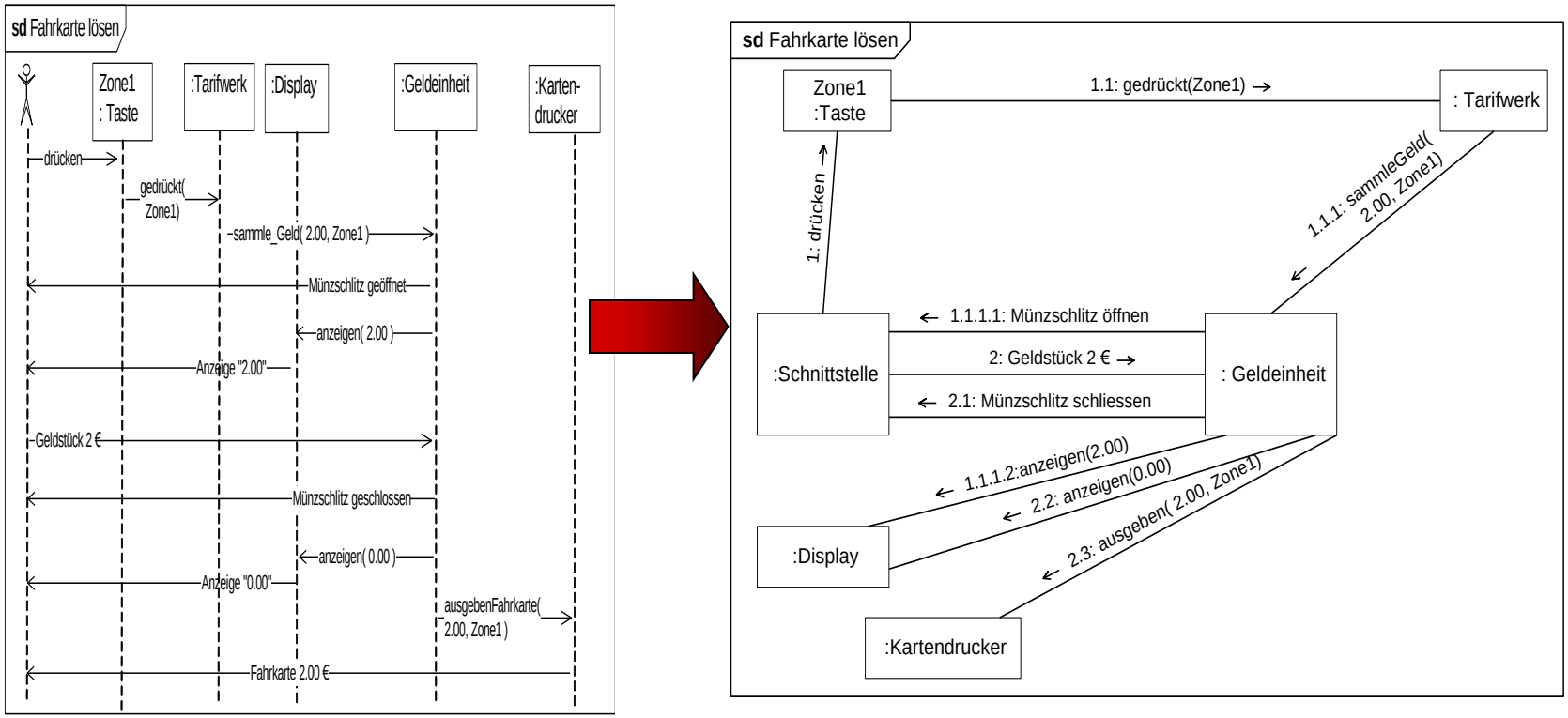
Das Kommunikationsdiagramm: Notation



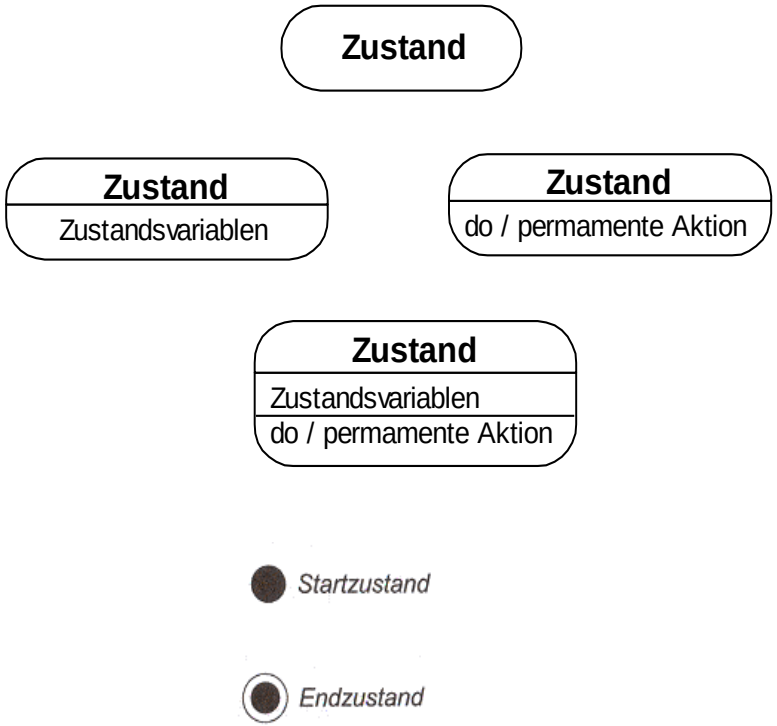
Zum Vergleich:



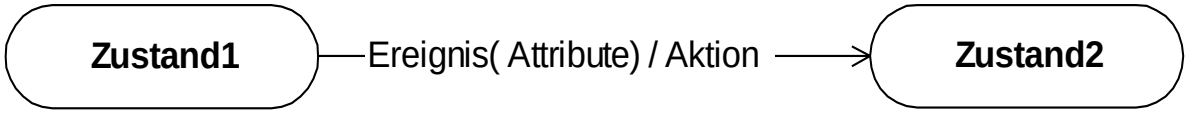
Kommunikationsdiagramm – Beispiel Fahrkartenautomat



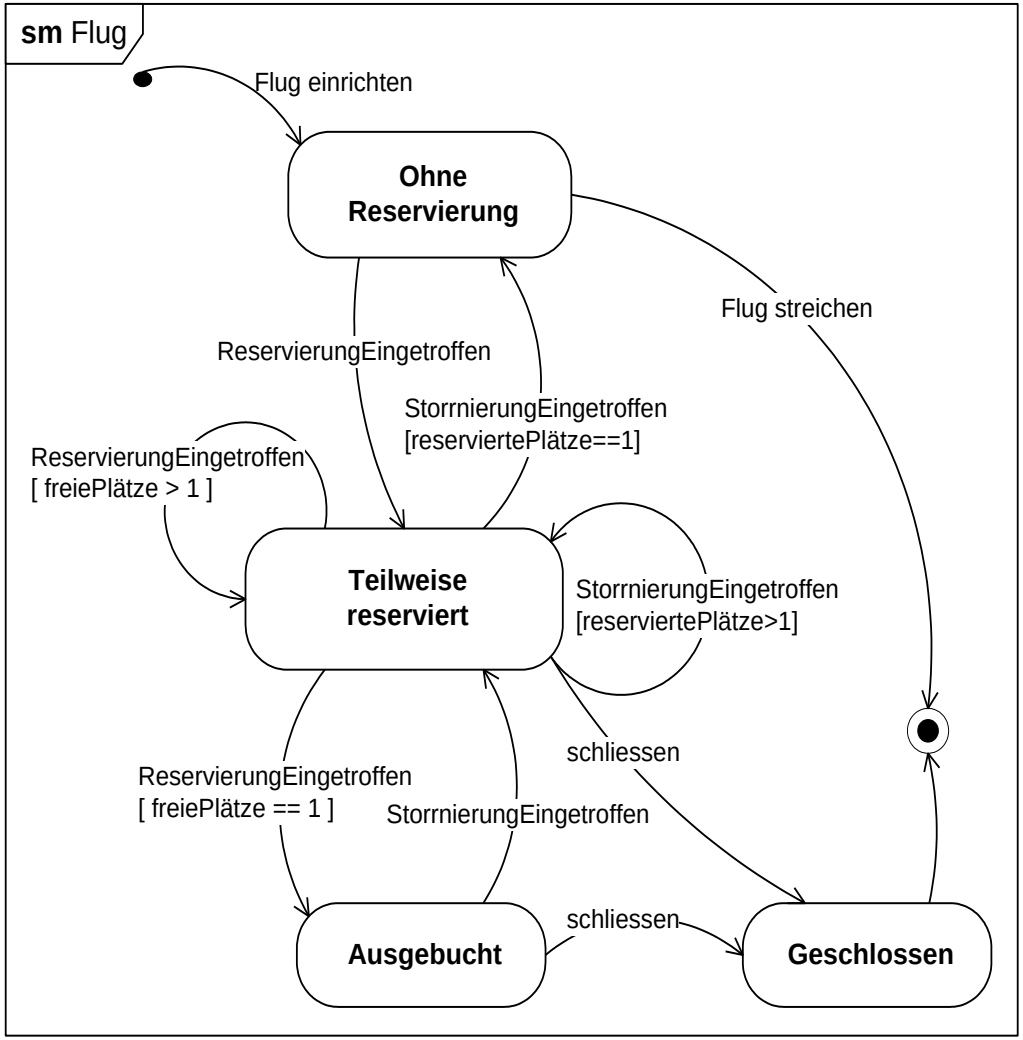
Zustandsdiagramm - Notation



- Notation (Grundlagen)
  - abgerundetes Rechteck
  - optionale Trennung in bis zu 3 Bereiche
    (Name, Zustandsvariablen und Aktionen im Zustand)
  - besondere Symbole für Start- und Endzustand
  - Zustandsübergang:
    Name des Ereignisses und optional eine Aktion

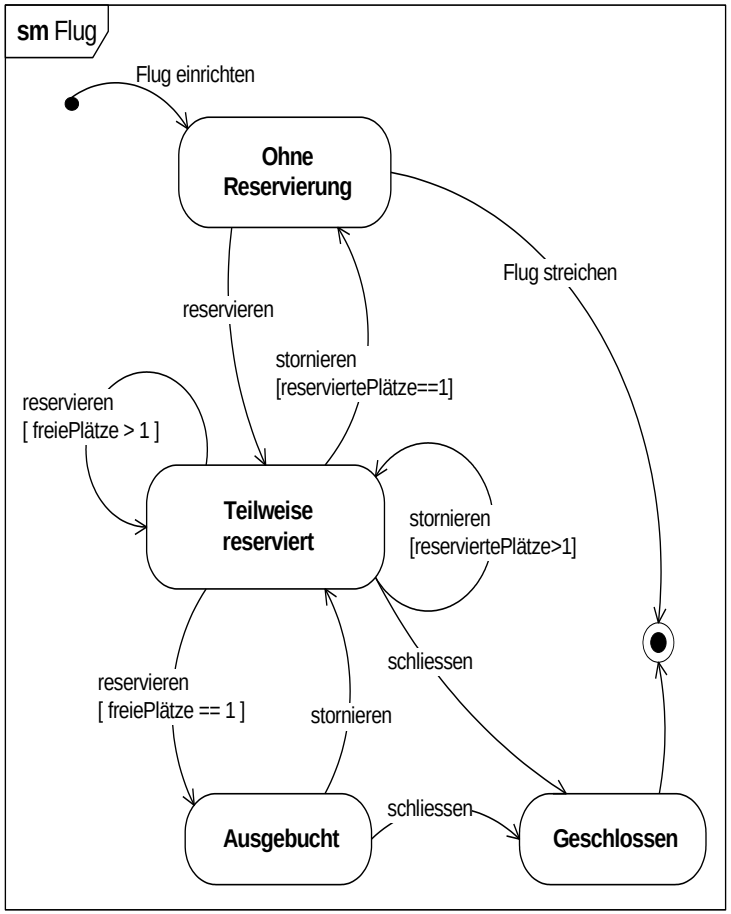


Zustandsdiagramm - Bedingungen



- Bedingungen
  - werden in allen Diagrammen in eckigen Klammern angegeben

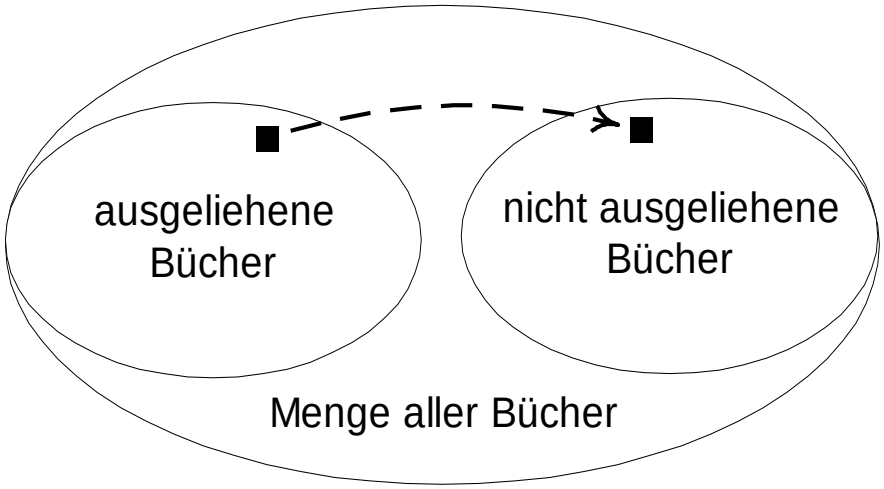
Zustandsdiagramm



- Zustandsdiagramm
  - zeigt Folge von Zuständen, die ein Objekt im Laufe seines Lebens einnehmen kann
  - Ereignisse lösen Zustandswechsel aus
  - Beschreibt endlichen Automaten
    - Menge von Zuständen
    - Menge von Ereignissen, die Zustandsübergänge bewirken
    - Anfangs- und Endzustände
  - Optional Rahmen
- Zustand
  - Zeigt die Zustände einer Klasse / Subsystem / System
  - ist eine Abstraktion einer Menge von Attributwerten

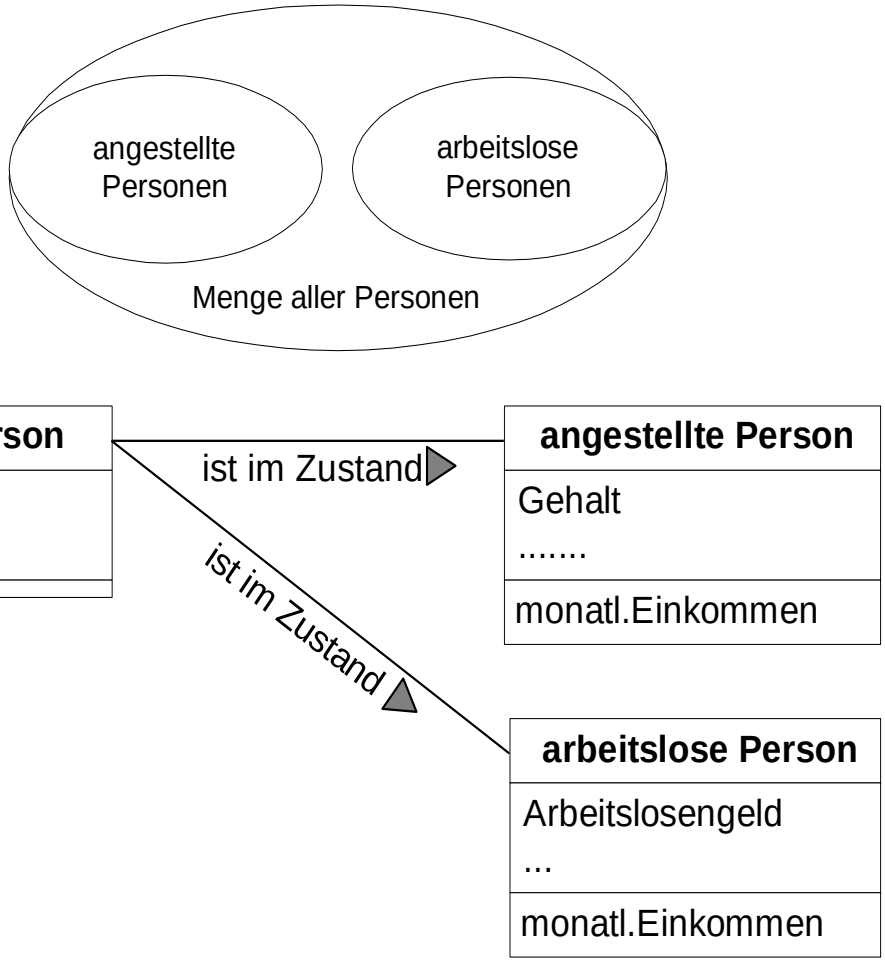
Zustände oder Unterklassen

- Temporäre Unterschiede zwischen Objekten werden als Zustand (oder Rollen) modelliert, inhärente Unterschiede zwischen Objekten als verschiedene Klassen.

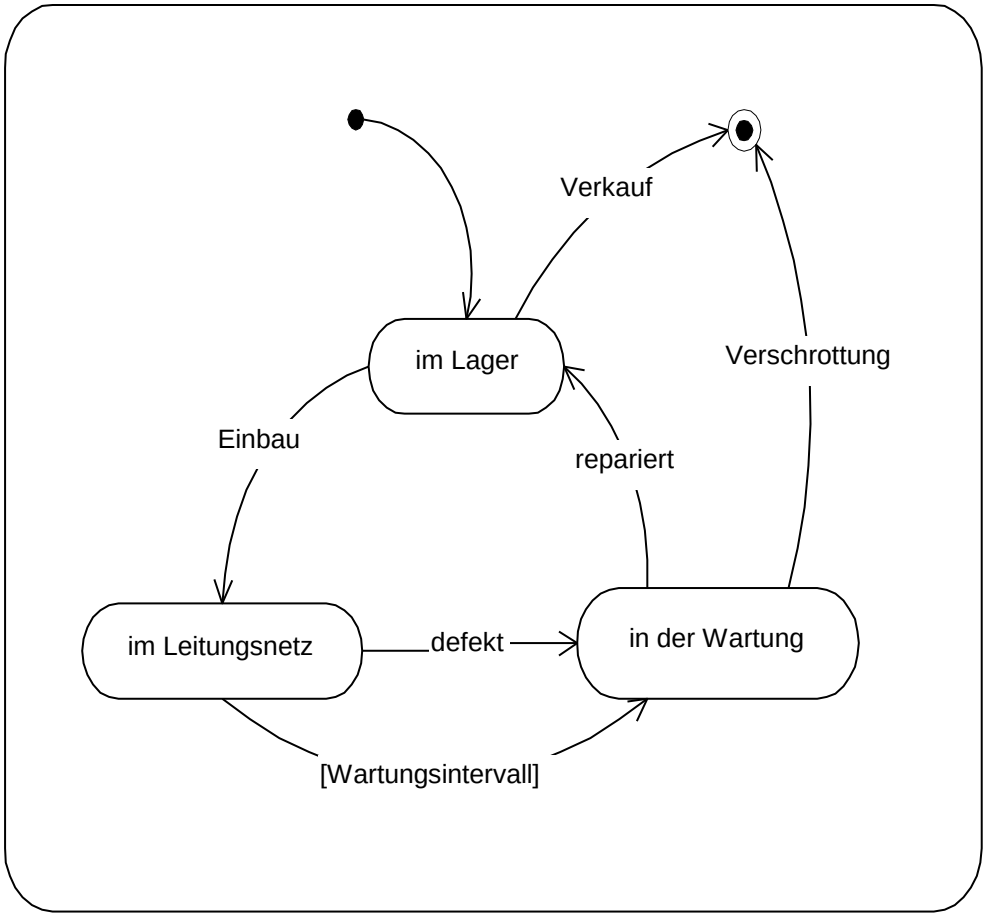


Ein Buch kann von der Menge aller ausgeliehenen Bücher in die Menge aller nicht-ausgeliehenen Bücher wechseln: es sind keine Unterklassen, sondern zwei Zustände

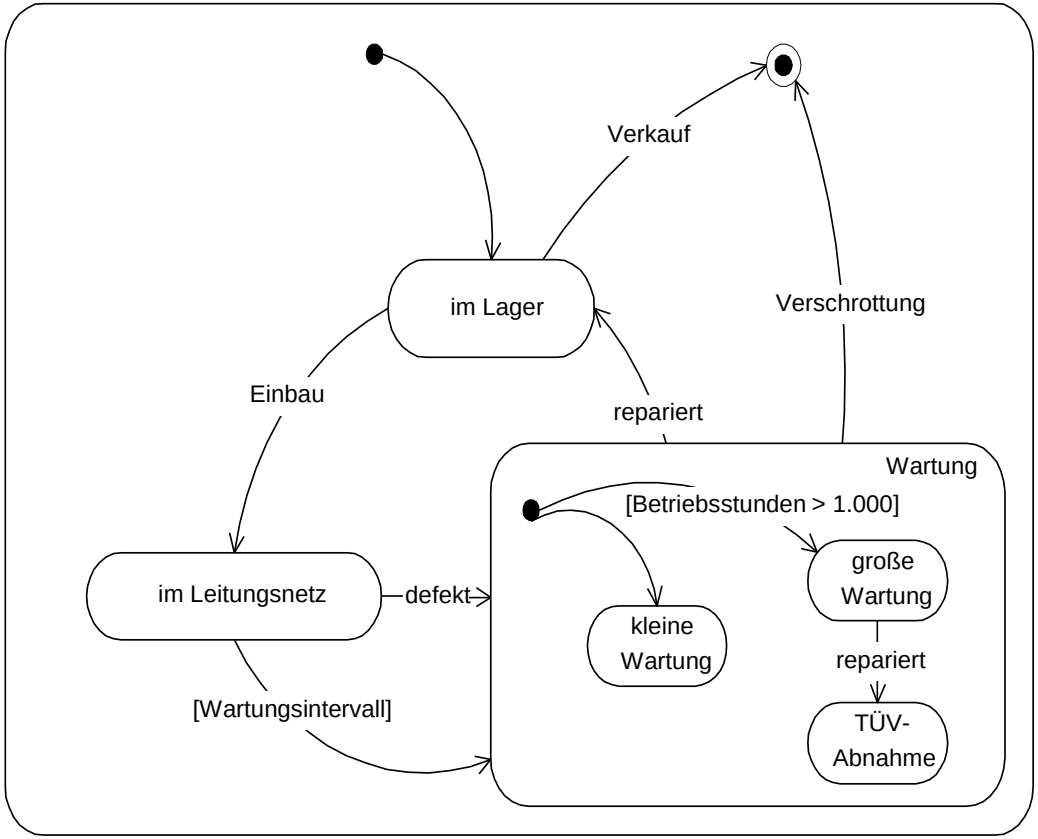
Zustände als Klassen



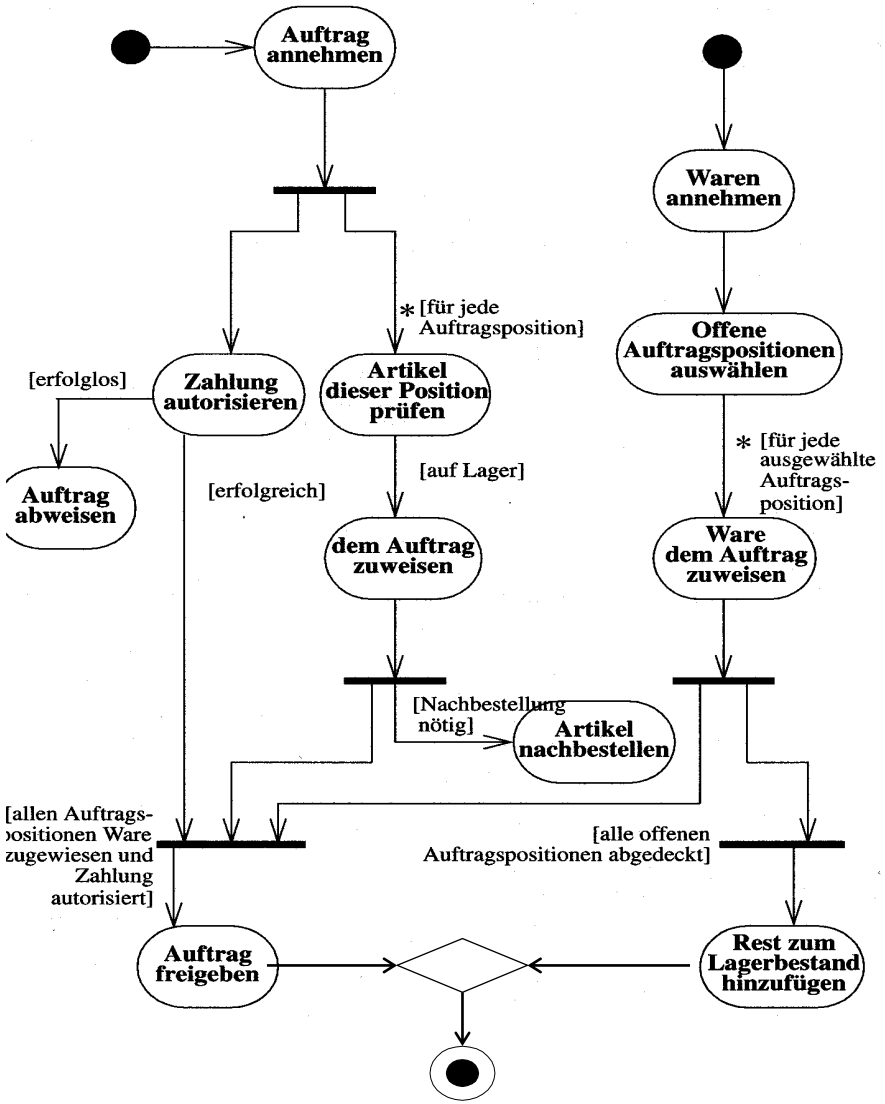
Geschachtelte Zustandsdiagramme



Unterteilung des Zustands Wartung

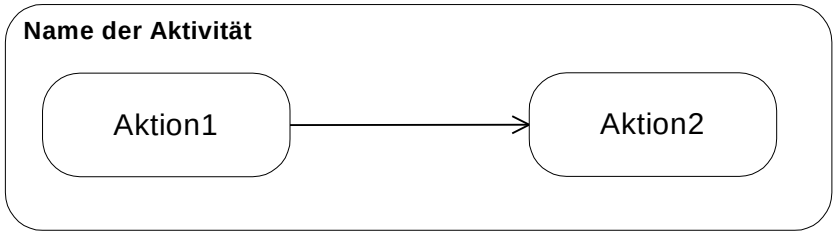


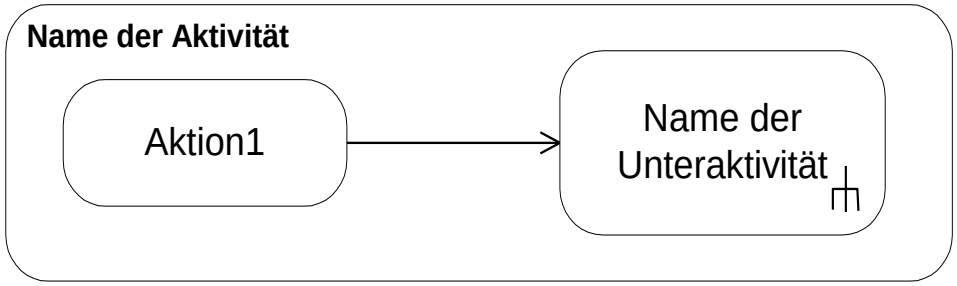
Aktivitätsdiagramm



- Aktivitätsdiagramm
  - Stellt beschreibt die Ablaufmöglichkeiten eines Systems als Abfolge von Aktionen dar
- Ablauforientiertes Diagramm zur
  - Geschäftsprozessmodellierung
  - Darstellung eines Use-Case-Ablaufs
  - Darstellung beliebiger Abläufe

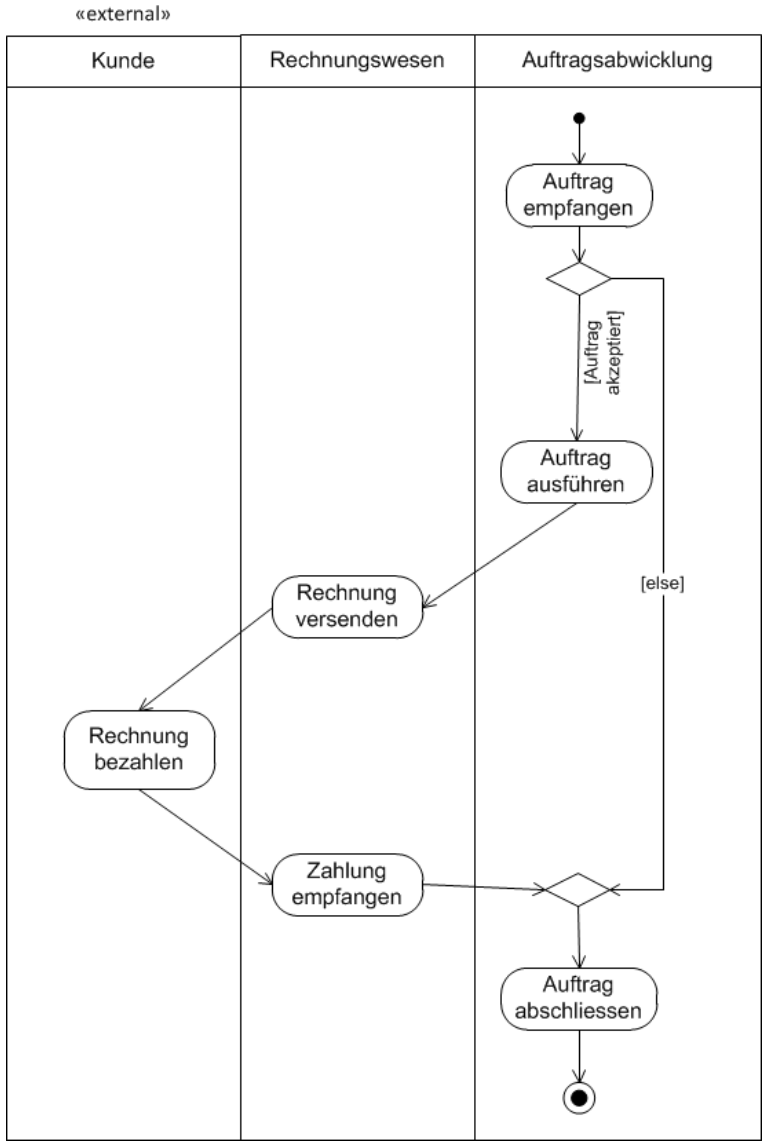
Aktivitätsdiagramm





- **Aktion**
  - ein einzelner Schritt in einem Aktivitätsdiagramm
  - beginnt, wenn die die Vorgänger-Aktion zu Ende ist
- **Aktion kann sein**
  - Ein elementares Verhalten, z.B. eine Berechnung
  - Aufruf der Operation an einem Objekt
  - wiederum eine Aktivität (Schachtelung wird durch stilisierte Harke symbolisiert)

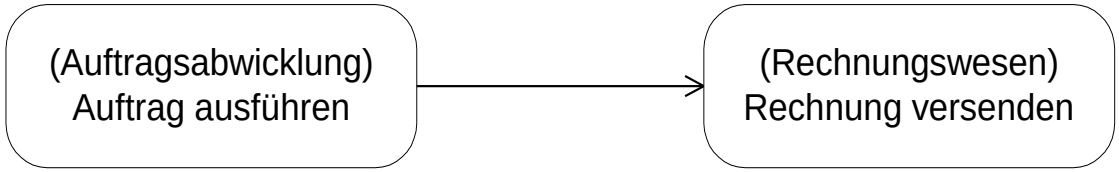
Aktivitätsdiagramm - Aktivitätsbereiche



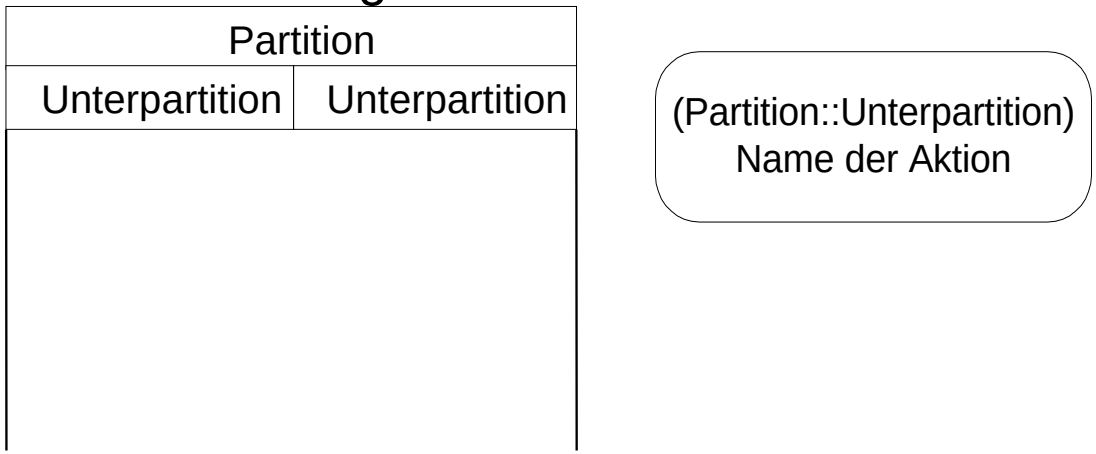
- Gruppierung in Bereiche
  - nach Verantwortlichkeit, z.B.
    - Organisationseinheiten
    - Akteure (Rollen)
    - Subsysteme
  - nach Standort, z.B.
    - Filiale oder Land
    - Rechnerknoten
  - Bereich außerhalb des Modellierungsfokus mit «external» kennzeichnen

Aktivitätsdiagramm - Aktivitätsbereiche

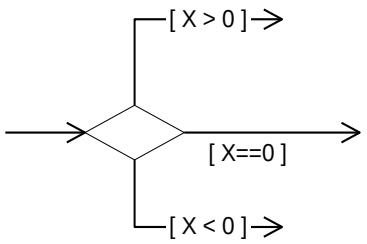
- Alternativdarstellung ohne Schwimmbahnen

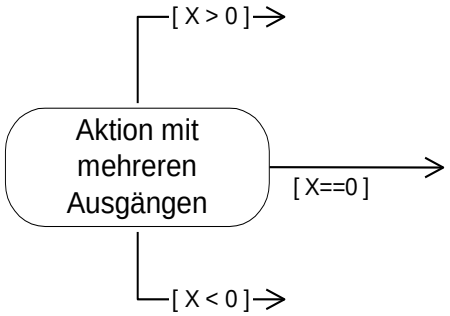


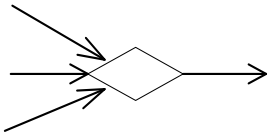
- Hierarchische Gliederung des Bereichs



Aktivitätsdiagramm - Fallunterscheidungen

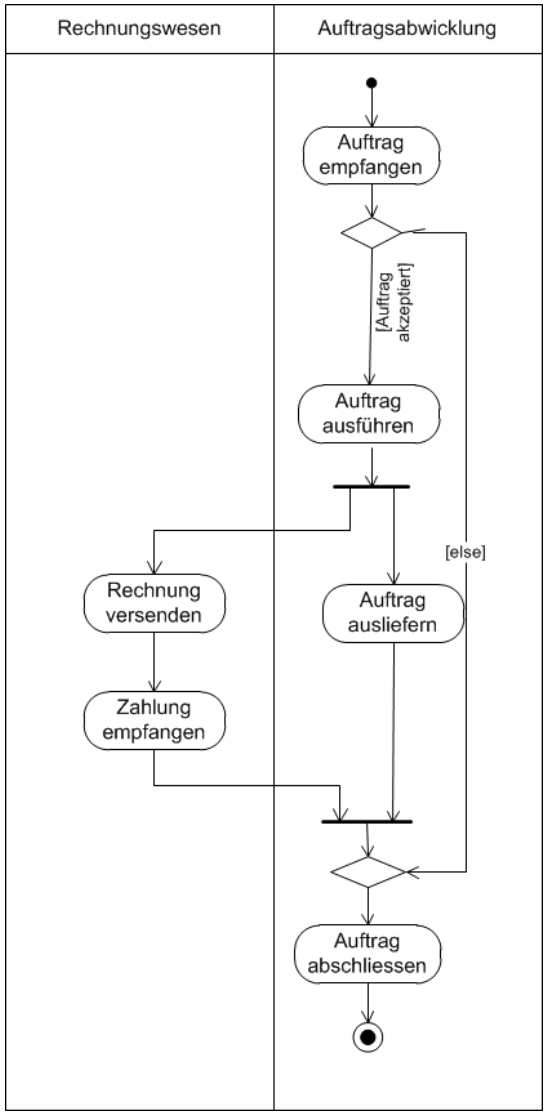


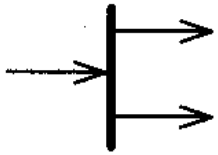


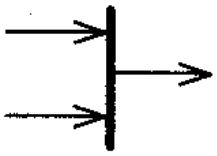


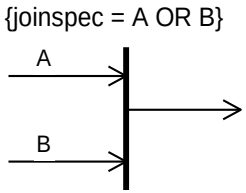
- **Verzweigungsknoten:** nur eine der ausgehenden Kanten wird durchlaufen. Die Bedingungen werden in eckige Klammern geschrieben.
- Statt der Raute können auch mehrere Ausgänge aus einer Aktion verwendet werden.
- **Verbindungsknoten:** Zusammenführung des Kontrollflusses. Stattdessen können auch mehrere Eingänge in eine Aktion verwendet werden.

Aktivitätsdiagramm - Parallelität





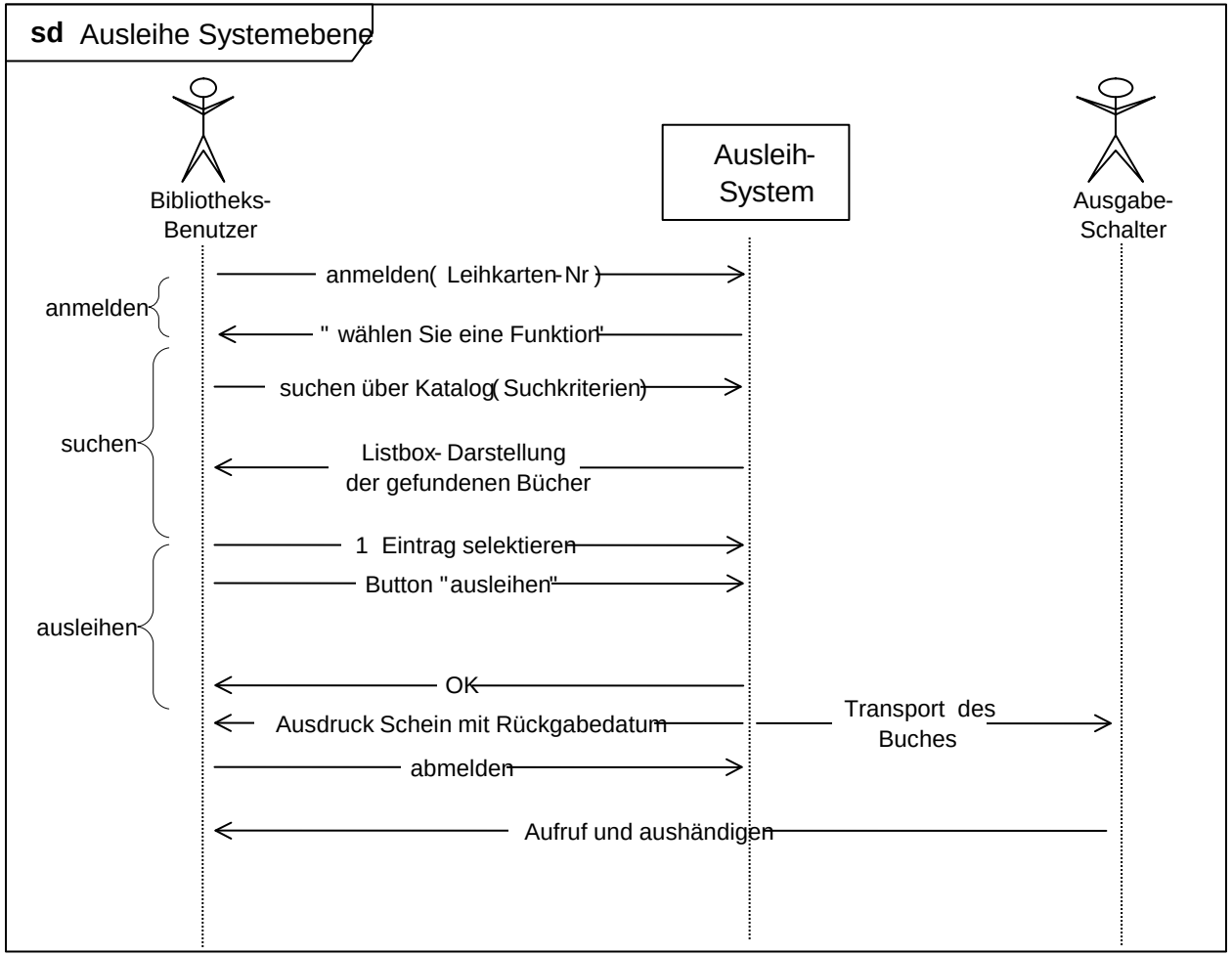




- **Parallelisierungsknoten:** Aufspalten des Kontrollflusses in nebenläufige, unabhängige Prozesse
- **Synchronisationsknoten:** Zusammenführen neben-läufiger Prozesse, beide Vorgängeraktionen müssen zu Ende sein (AND)
- **andere Synchronisation:** kann durch eine Synchronisa-tionsspezifikation am Balken angegeben werden, z.B. {joinspec = OR}

Das System-Sequenzdiagramm

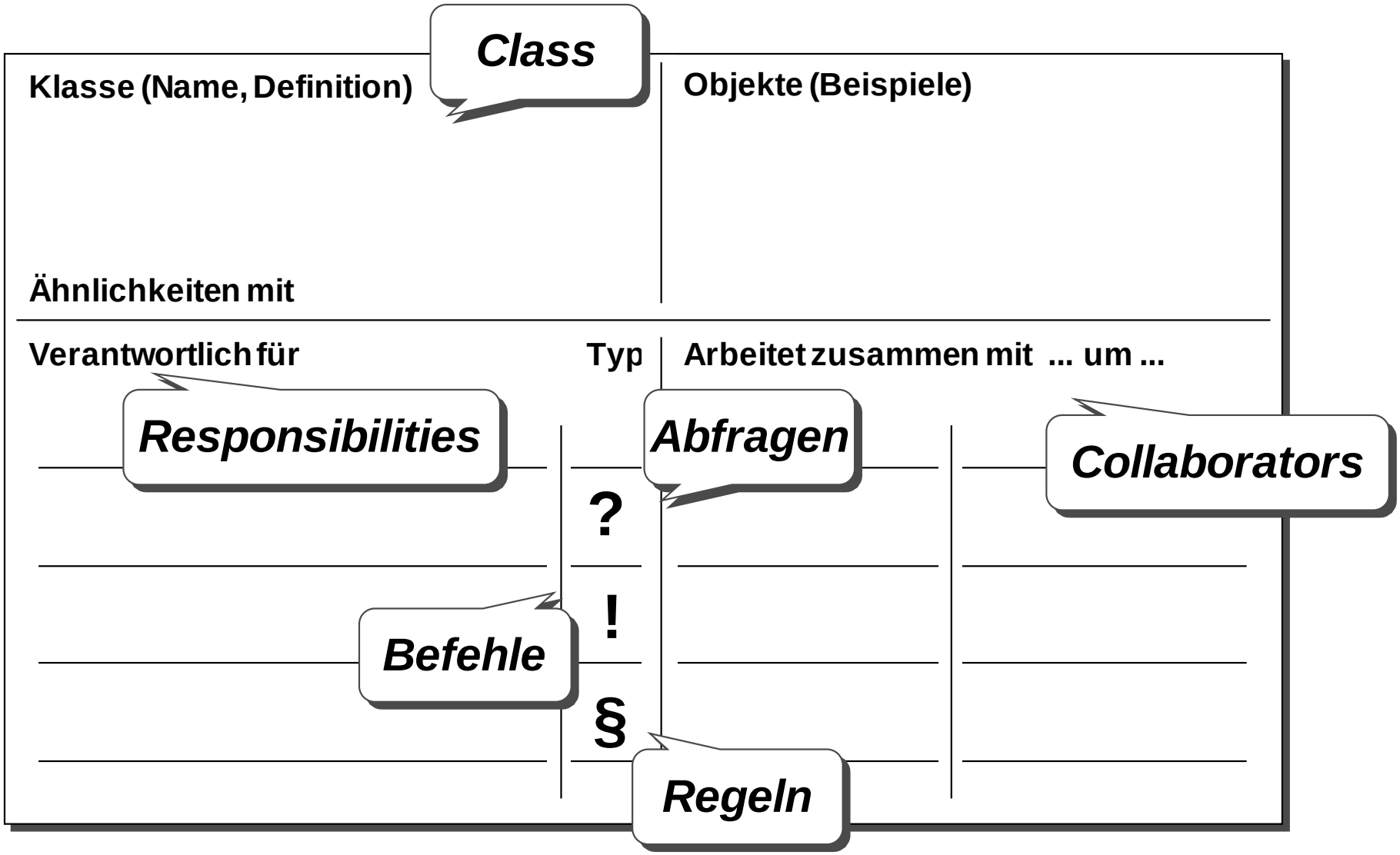
- Zeigt das Zusammenspiel zwischen Akteuren und System



CRC Index Cards

Class Name	***Collaborators***
Responsibilities	

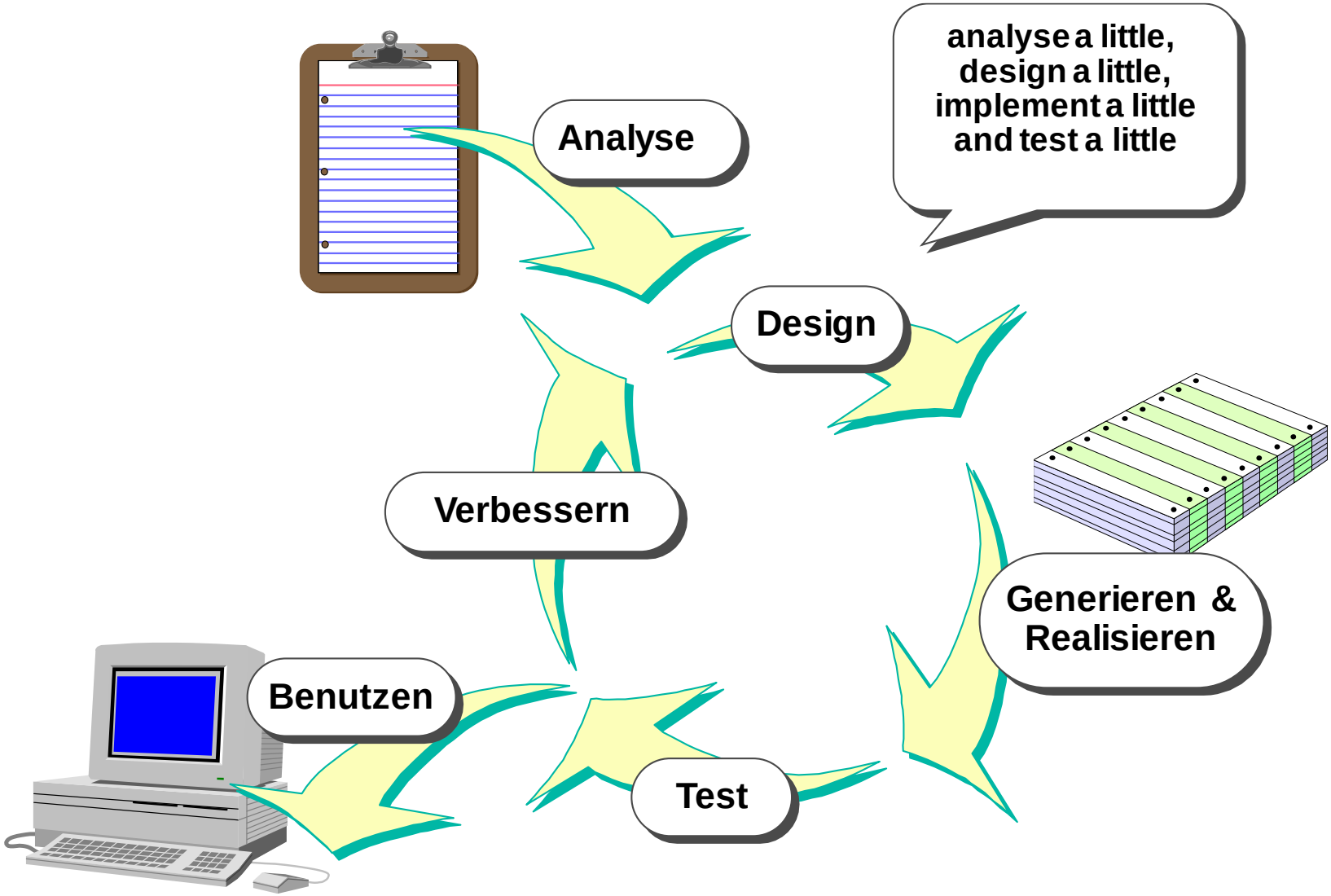
Anforderungen mit CRC-Cards beschreiben



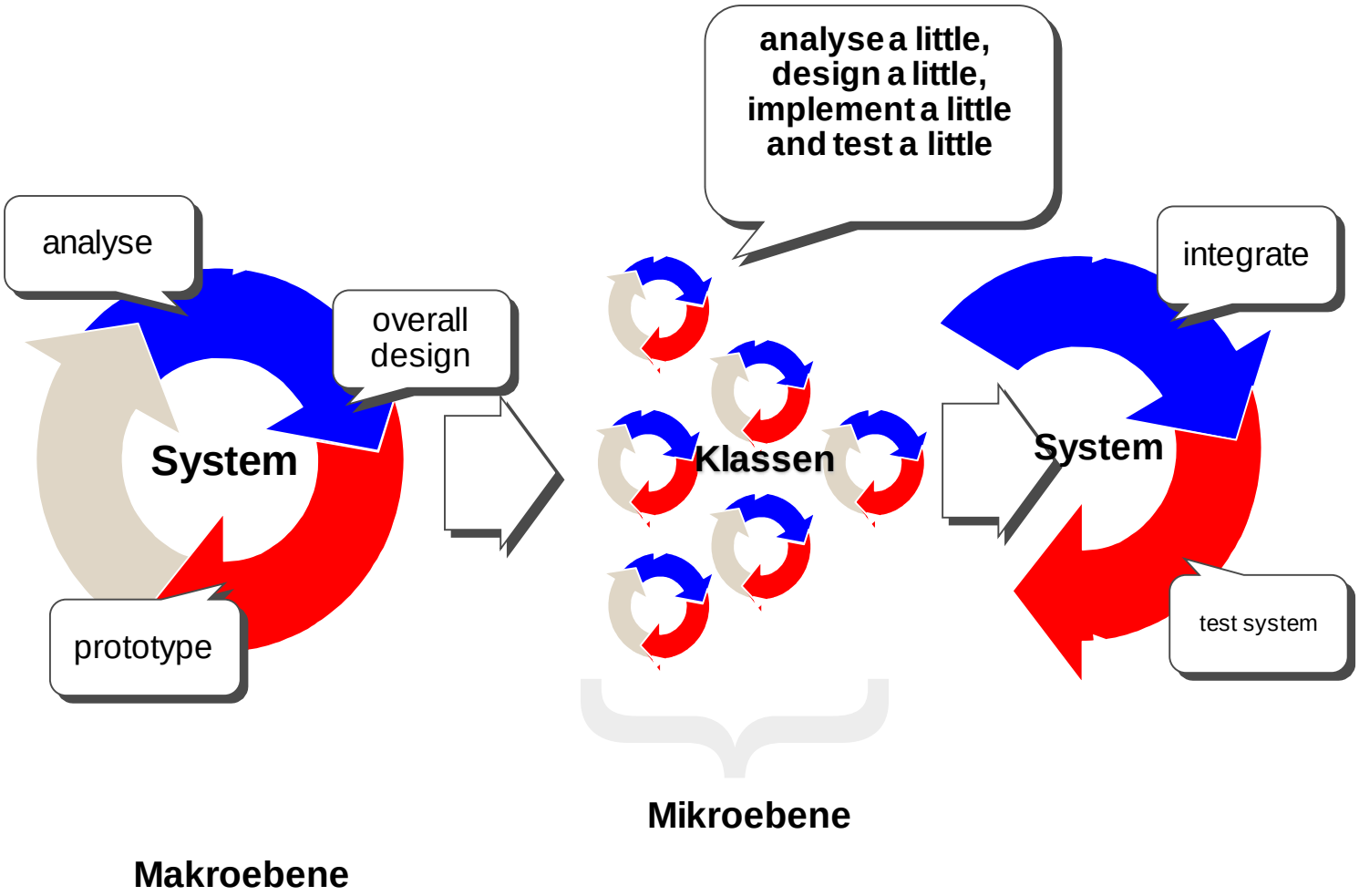
Varianten objektorientierter Vorgehensweisen

- Vorgehensmodell
  - Iterativ oder Wasserfall
  - Formal oder mit Prototyping
- Ansatz
  - datenorientiert
  - ereignisorientiert
  - szenario-orientiert
- Beschreibung
  - der fachlichen Anforderungen oder
  - der geplanten DV-Systeme

Evolutionäre Systementwicklung



Parallele (simultane) Entwicklung



4

OBJEKTORIENTIERTES DESIGN

Abgrenzung zur Analyse

Continental Institut für Technologie und Transformation

Analyse	Design
WAS soll die Software leisten (Spezifikation)	**WIE soll das erreicht werden (Lösung)**
Fachliche Varianten	**Mehrere technische Lösungen möglich => Vor- und Nachteile**
Frei von Technologie	**Technologische Restriktionen**
Zusammenspiel der Objekte	**Entkopplung**
Kernkonzepte identifizieren	**Kapselung**
Grobe Typisierung	**OOPL- Datentypen**
	Polymorphie
Nur fachliche Klassen	**Auch technische Schichten / Klassen**
1 fachliche Klasse Person	**PersonIF, PersonBean, PersonHome**

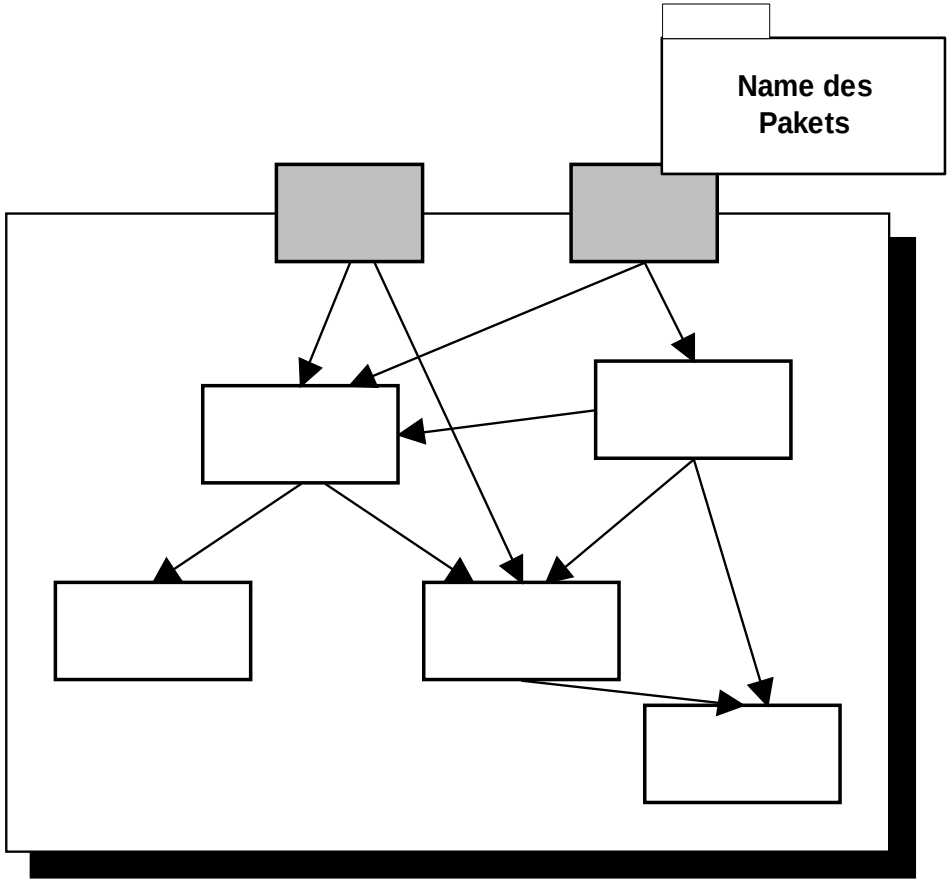
Grundsätze für ein gutes Design

- Minimale Kopplung
- Enge Kommunikationskanäle
- Gleichmäßige Verteilung von Intelligenz
- Starke Klassenbindung
- Pro Aufgabe der Klasse eine Methode
- Gleiches Abstraktionsniveaus der Methoden
- Implementation Hiding
- Ausdruckskraft der Namen

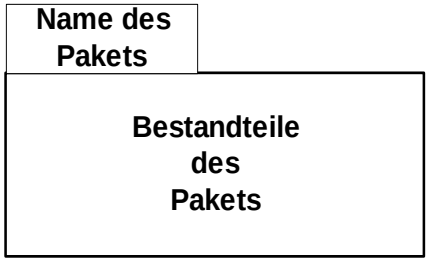
Aufgaben während des Designs

- Entwurf einer Software-Architektur
- Wahl einer Sprache und einer Programmierumgebung
- Entscheidung über die Form der Datenspeicherung
- Entscheidung über die Form der Benutzerschnittstelle
- Auswahl von Klassenbibliotheken für Collection-Klassen u.ä.
- Überarbeitung und Verfeinerung des OOA-Modells

Pakete



oder



Referenz- und Wert-Semantik

- Referenzsemantik: die Variable bzw. das Attribut referenzieren ein anderes Objekt
- Wertsemantik: die Variable bzw. das Attribut beinhaltet einen Wert

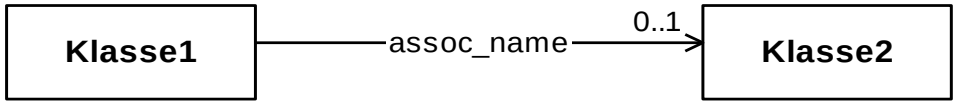
```
Punkt a, b, c;
a = new Punkt( 6, 8 );
b = new Punkt( 7, 9 );
c = a;
a.verschiebe( 1, 1);   // auch c wird verschoben
```

Die Variablen a und c referenzieren dasselbe Punkt-Objekt.

Wertsemantik: `a.x = b.x;`

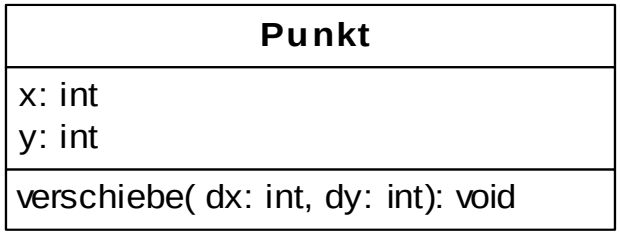
Eine nachfolgende Änderung von b.x hat keine Auswirkung auf a.x

- Referenzsemantik: geeignet für die Implementation von Assoziationen (unabhängige Objekte)



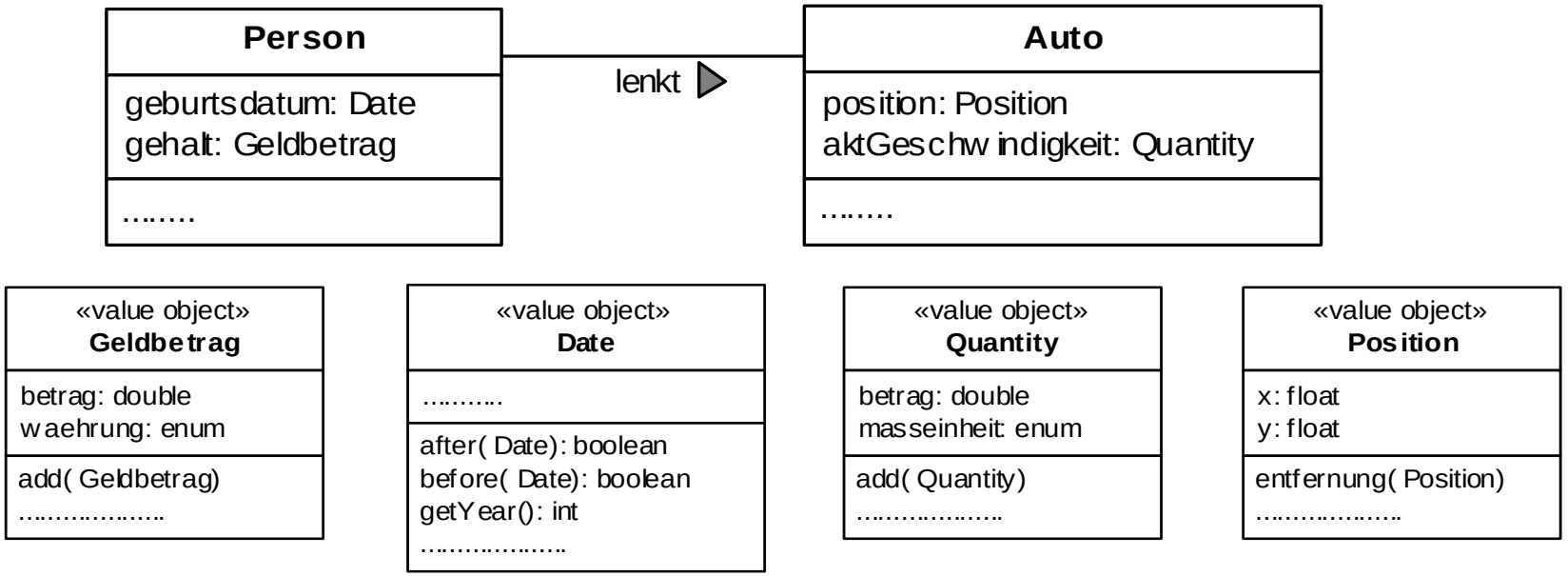

```
class Klasse1  {
       Klasse2 assoc_name;
}
```

- **Wertsemantik:** geeignet für die Implementation von Attributen

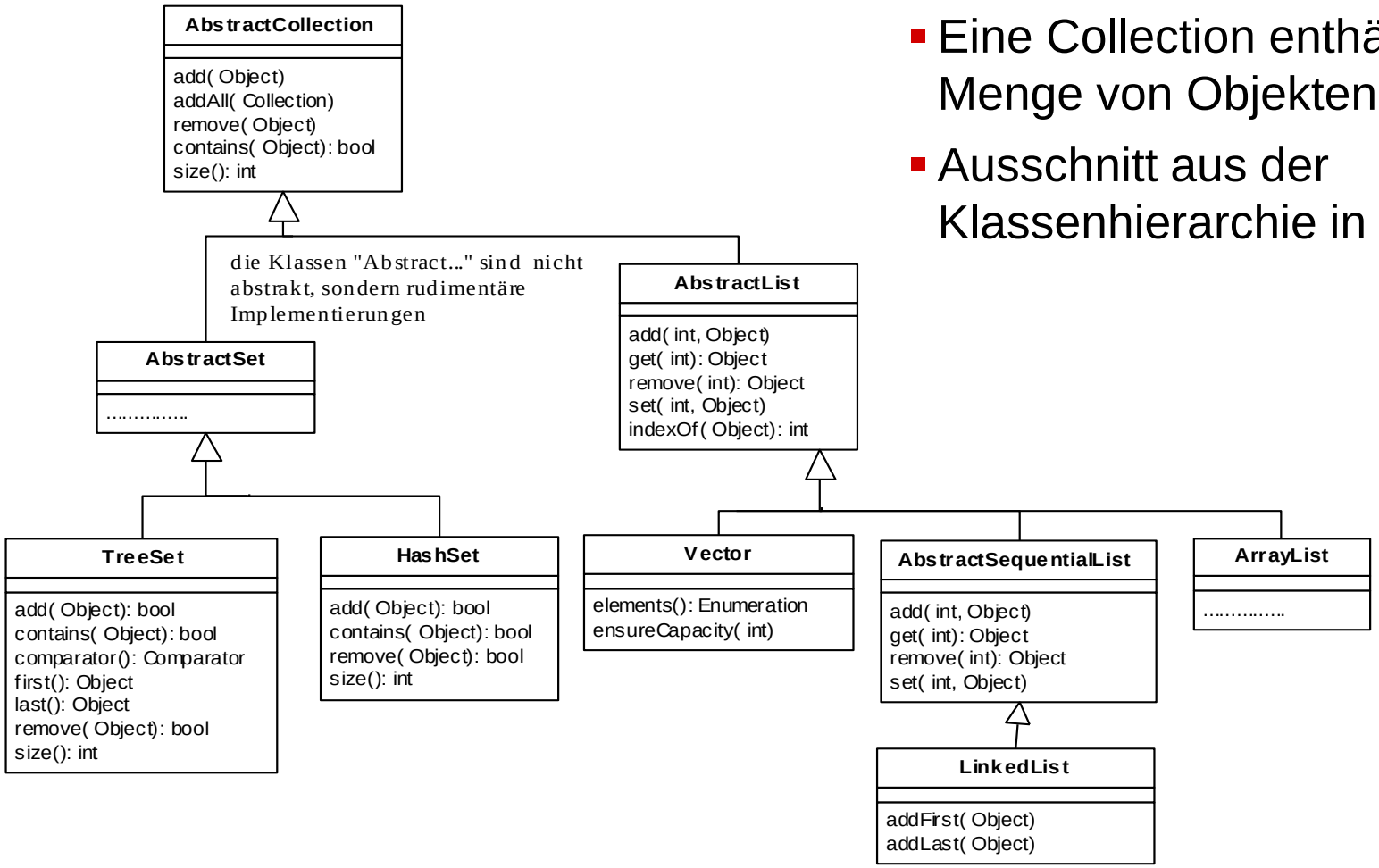


Pattern: Value Object

- *Ein Value Object ist ein Objekt, das semantisch keine eigene Identität hat, d.h. wie ein Attribut mit Wertsemantik behandelt wird.*



- **für zusammengesetzte Attribute** (z.B. Position, Geldbetrag, Quantity,...)
- **zur Bildung von basisfachlichen Datentypen** (z.B. Date, Position, Geldbetrag,...)
- **für Zeichenketten mit Regeln** (z.B. URL, ISBN-Nummer)

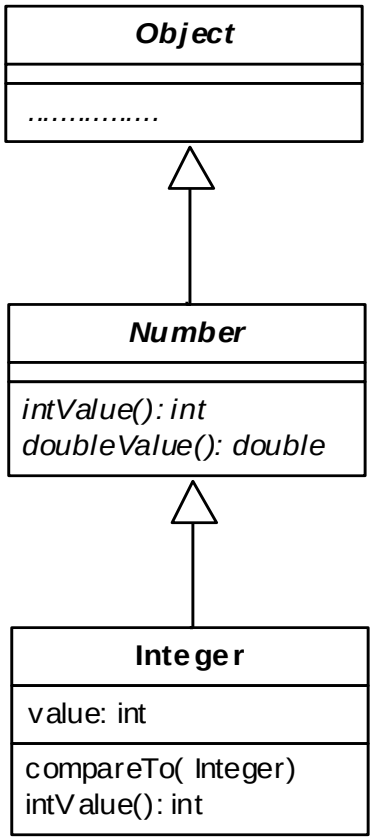


- Eine Collection enthält eine Menge von Objekten.
- Ausschnitt aus der Klassenhierarchie in Java

Wrapper-Klassen

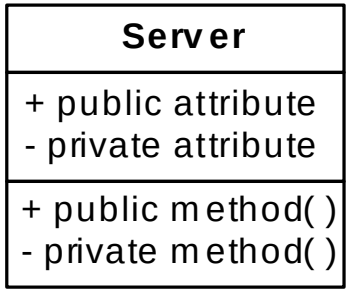
- Eine Wrapper-Klasse
  - hält den Wert eines primitiven Datentyps (int, ...)
  - ist (in Java) immutable, um Wertsemantik sicherzustellen
  - ist ein vollwertiges Objekt, bei dem die OO-Konzepte anwendbar sind
    - kann in Klassenhierarchie strukturiert werden
    - kann in Collections verwaltet werden (auch heterogene Collections)
    - kann in generischen Schnittstellen (Frameworks!) verwendet werden

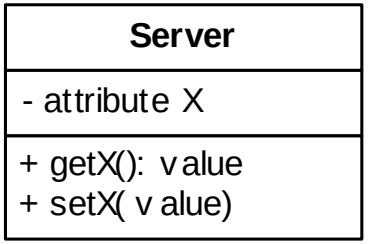
```
transferiere( Object obj)
```



Kapselung

- Designziel: eine Client-Klasse soll nicht abhängig sein von den Implementierungsdetails der Server-Klasse.
- Sichtbarkeiten für Attribute / Methoden:
  - **public:**die Client-Klasse kann direkt auf das Attribut / die Methode zugreifen
  - **private:** die Client-Klasse kann nicht auf das Attribut / die Methode zugreifen, nur innerhalb der kapselnden Klasse kann das Attribut / die Methode verwendet werden



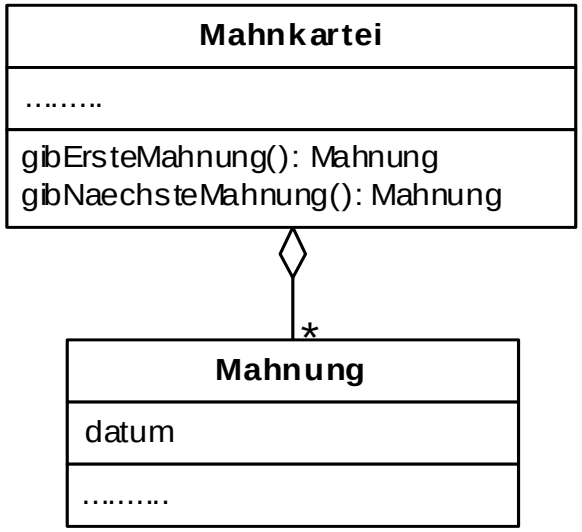


Typen von Methoden

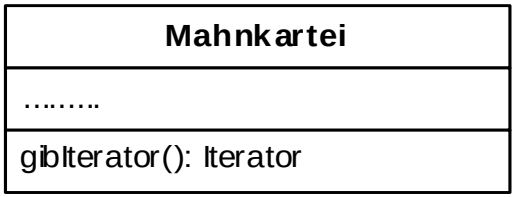
- Methoden
  Es werden folgende Typen von Methoden unterschieden:
  1. **Konstruktoren:** dienen zur Erzeugung von Objekten
     - evtl. Unterteilung in create / initialize
  2. **Destruktoren:** dienen zur Vernichtung von Objekten
     - evtl. Unterteilung in cleanup / terminate
  3. **Anfrage-Methoden:** liefern nur ein Ergebnis, es werden keine Objekte verändert.
     - get()-Methoden und Berechnungs-Methoden
  4. **Änderungs-Methoden:** bewirken eine Änderung im Objekt und/oder benachbarten Objekten.
     - set()-Methoden und höherwertige Dienste wie `abbuchen()`
  5. **Iterator-Methoden:** über eine Menge von Objekten iterieren

Iteration

- Iterations-Methoden beim Mengen-Objekt

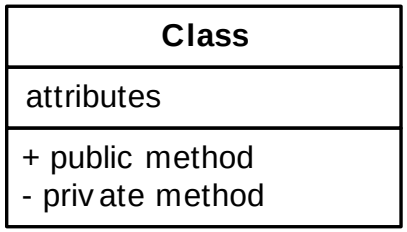


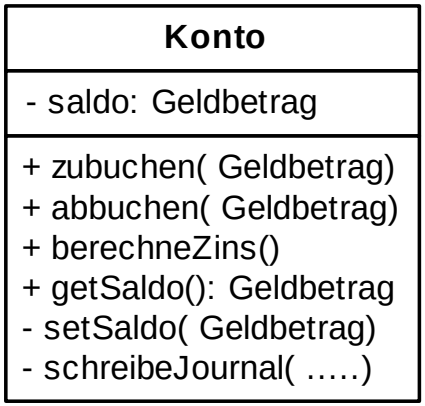
- **Iterations-Methoden beim Iterator-Objekt**




```
class Iterator {
        boolean
hasNext();
        Object next();
}
```

Public und private Methoden
Klassenmethoden

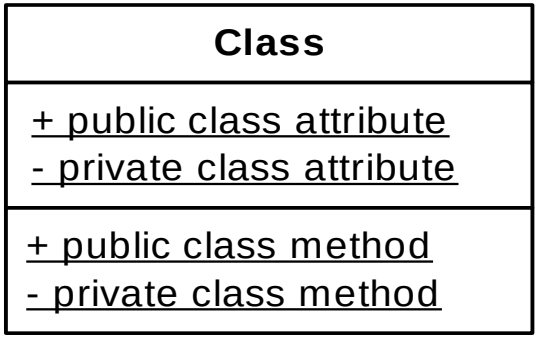




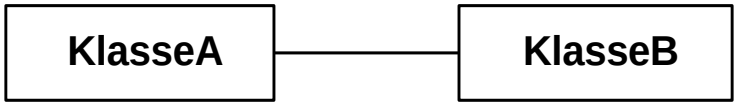
- Klassenattribute
  - gemeinsamer Wert für alle Objekte einer Klasse
- Klassenmethoden
  - Zugriff auf Klassenattribute

Beispiel (C++ und JAVA):

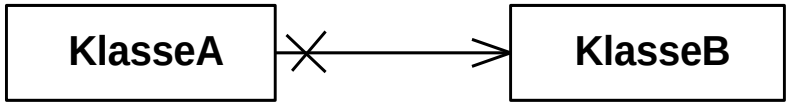
```
class X {
static int myInstanceCounter;
static void incNoOfInstances() { myInstanceCounter++ ; }
static void decNoOfInstances() { myInstanceCounter-- ; }
}
```



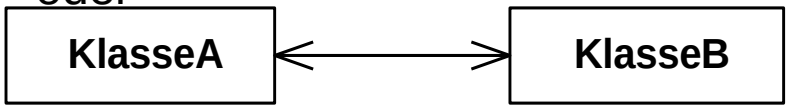
- In der Analyse oft ungerichtete Assoziationen
  - Richtung an beiden Enden unspezifiziert



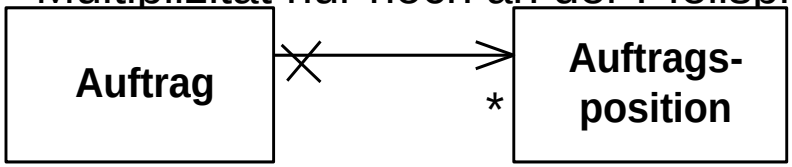
  - **im Design nur gerichtete Assoziationen**



  - oder

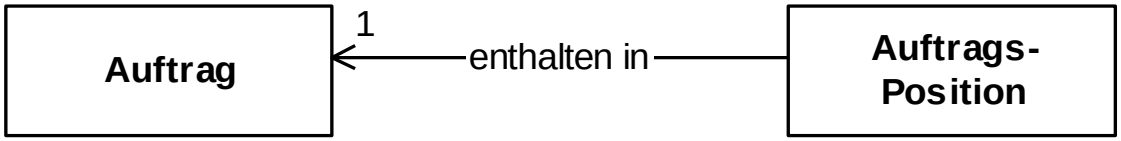


  - Multiplizität nur noch an der Pfeilspitze interessant



Implementierung zu-1 Assoziation

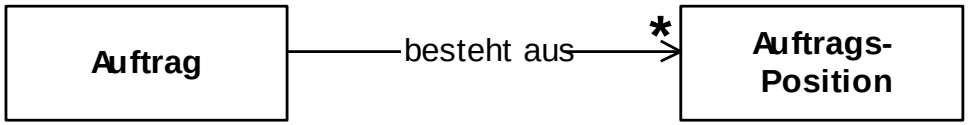
- Kardinalität 0-1: Implementierung mit einem Referenzattribut
- Datentyp des Referenzattributes ist die referenzierte Klasse




```
class Auftragsposition  {
        private Auftrag enthalten_in;
        Auftrag getAuftrag();
        setAuftrag( Auftrag);
        removeAuftrag();
}
```

Implementierung zu-n Assoziation (variables n)

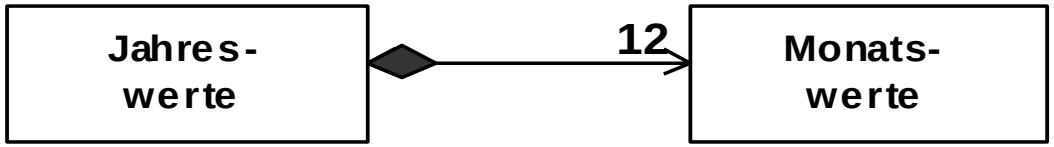
- Kardinalität n: Implementierung mit Collection von Objektreferenzen




```
class Auftrag  {

        ........

        private Vector besteht_aus;
        Iterator getAuftragspositionen();

        addAuftragsposition( Auftragsposition
p);

        removeAuftragsposition( Auftragsposition
p);

}

class Auftragsposition  { ....}
```

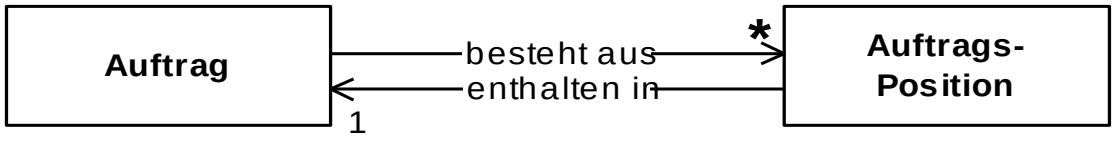
Implementierung zu-n Assoziation (festes n)

- Kardinalität festes n: Implementierung mit Array von Objektreferenzen




```
class Jahreswerte  {
      private Monatswerte[] werte = new
Monatswerte[12];
      Monatswerte getMonatswerte( int monat);
      setMonatswerte( Monatswerte m);
}
class Monatswerte  { // alle Werte für 1 Monat
....}
```

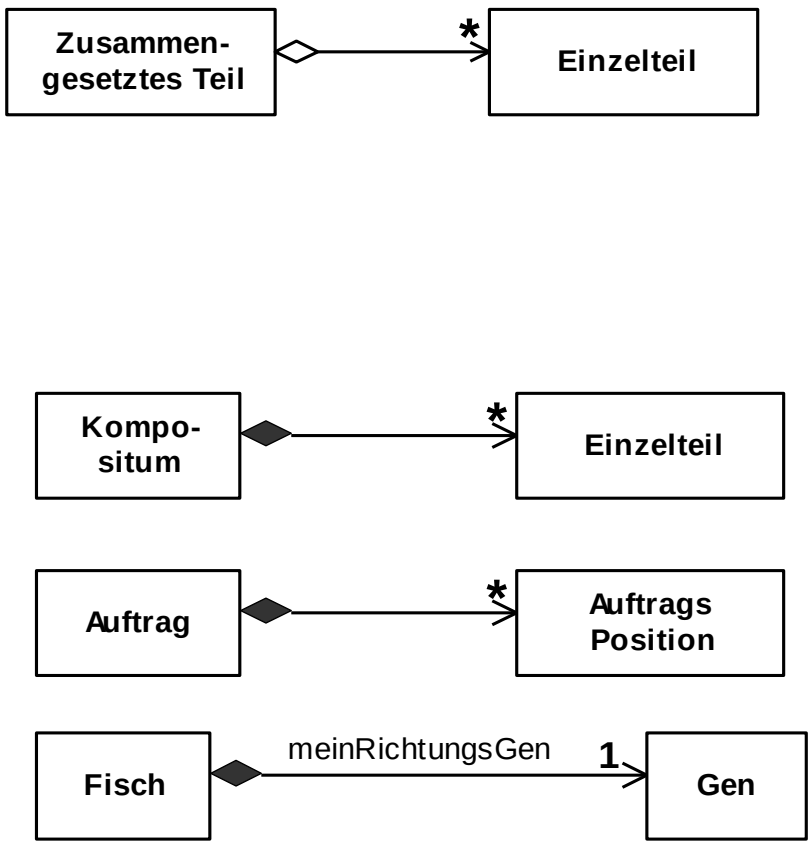
Implementierung bidirektionale Assoziation (1)




```
class Auftrag  {
      Vector besteht_aus;
}
class Auftragsposition  {
      Auftrag enthalten_in;
}
```

- Kombination beider unidirektionaler Assoziationen
- auch n-zu-m möglich
- enge Kopplung

Aggregation / Komposition



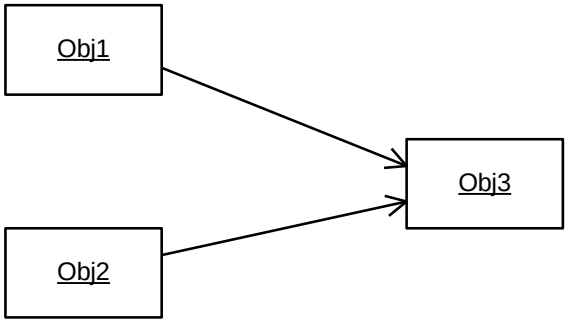
- Eine Aggregation ist wie eine Assoziation zu implementieren
- Jedes Teil wird durch das Ganze mittels einer der Kardinalität entsprechenden Assoziation verwaltet
- Besonderheiten der Komposition
  - Teile sind existenzabhängig vom Ganzen
  - Destruktor des Ganzen löscht Teile (bzw. gibt sie frei)
  - In C++ wird byValue benutzt (Wertsemantik)

```
class Fisch {
  Gen meinRichtungsGen;
}
```

Management von Assoziationen

Typische Fehlermöglichkeit (C++):

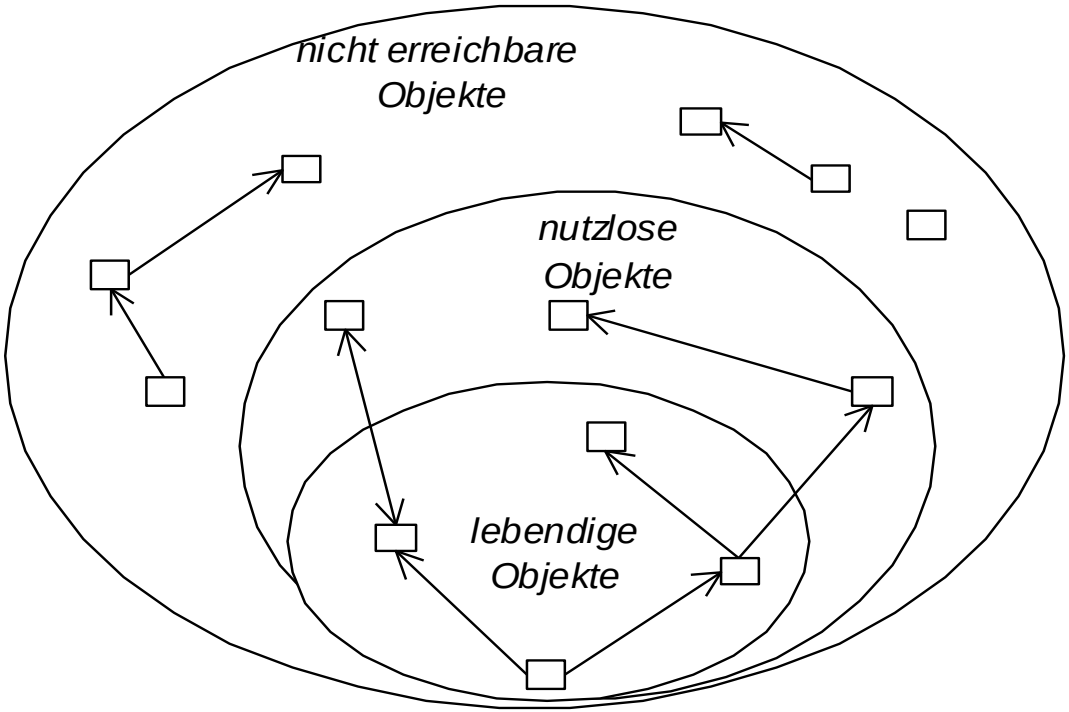
1. Obj1 und Obj2 bauen eine Referenz zu Obj3 auf.
2. Der Destruktor von Obj1 wird aufgerufen. Damit erlischt die Referenz auf Obj3 automatisch.
3. Der Destruktor von Obj2 wird aufgerufen. Damit erlischt die Referenz auf Obj3 automatisch.
4. Auf Obj3 zeigen keine Referenzen mehr, niemand kann mehr den Destruktor aufrufen (memory leak).



Typische Fehlermöglichkeit (C++):

1. Obj1 und Obj2 bauen eine Referenz zu Obj3 auf.
2. Der Destruktor von Obj1 wird aufgerufen. Bevor die Referenz auf Obj3 erlischt, ruft Obj1 den Destruktor von Obj3 auf.
3. Über Obj2 wird auf Obj3 zugegriffen: protection violation.

Das Hauptspeicherproblem:



Das Erzeugen von Objekten

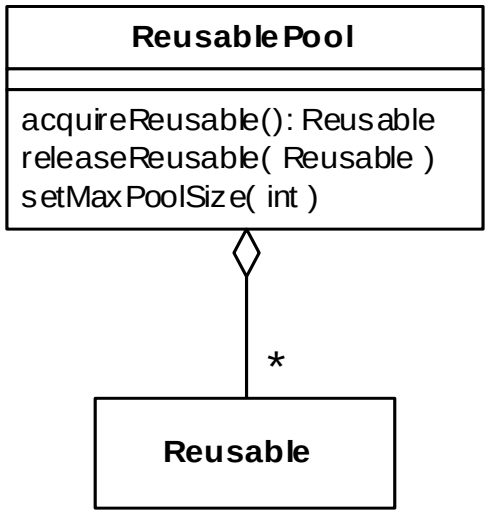
- Möglichkeiten zur Erzeugung von Objekten:
  - Konstruktor: `new Punkt( 6, 8 )`
  - Factory-Methode: `Geldbetrag.create( 60, „EUR“ );`
  - Factory-Objekt: `GeldbetragFactory.create( 60, „EUR“ );`
  - Factory-Methode und Factory können mit einem ***Pool*** kombiniert sein.

Pattern: Object Pool

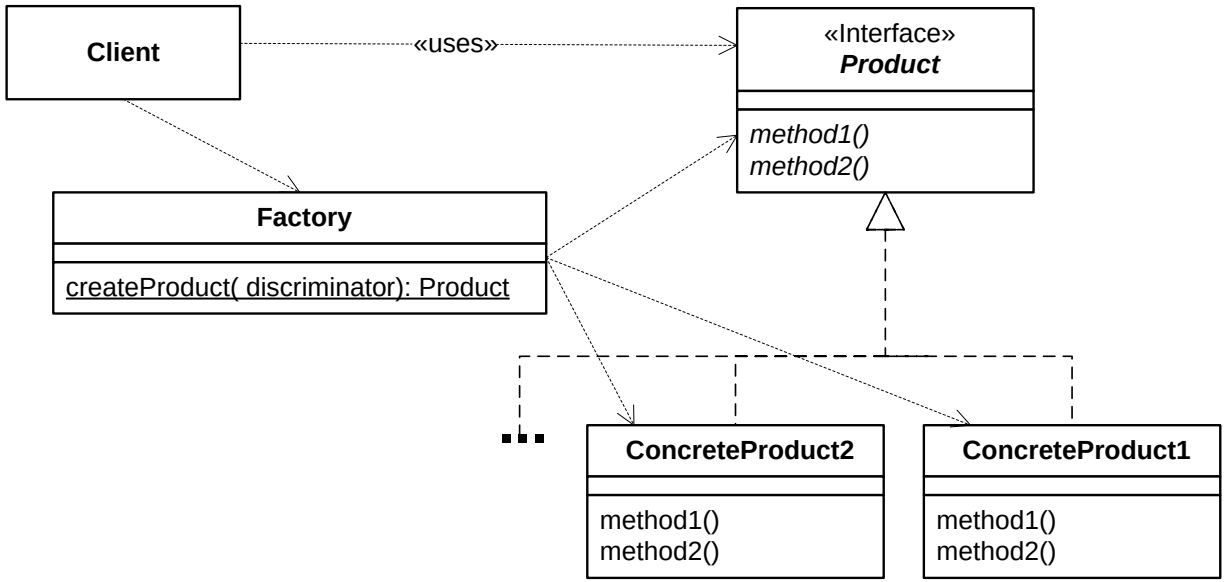
Ein Object Pool organisiert die Wiederverwendung von Objekten.

Er wird eingesetzt, wenn

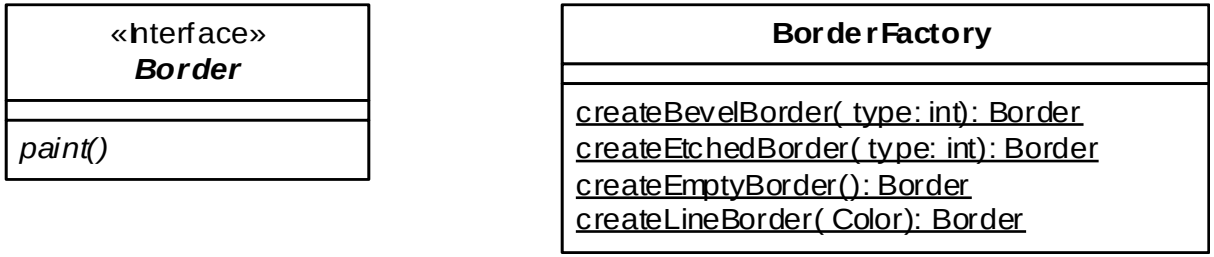
- nur ein begrenzte Anzahl von Objekten erzeugt werden kann
- die Erzeugung eines einzelnes Objekts relativ teuer ist
- die Benutzungszeit eines einzelnen Objekts nur sehr kurz ist und damit die Relation zum Zeitaufwand zur Erzeugung des Objekts unvorteilhaft ist.



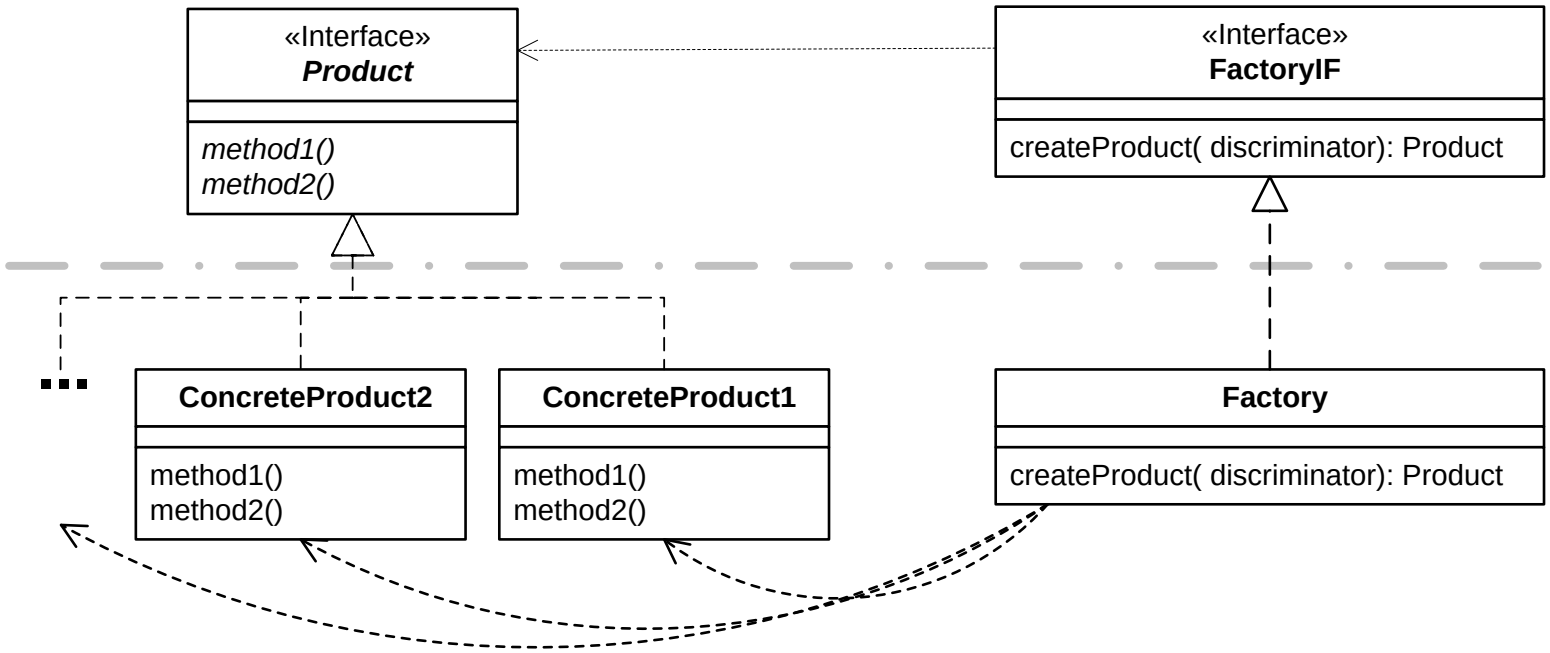
Pattern: Factory (1)



Beispiel:

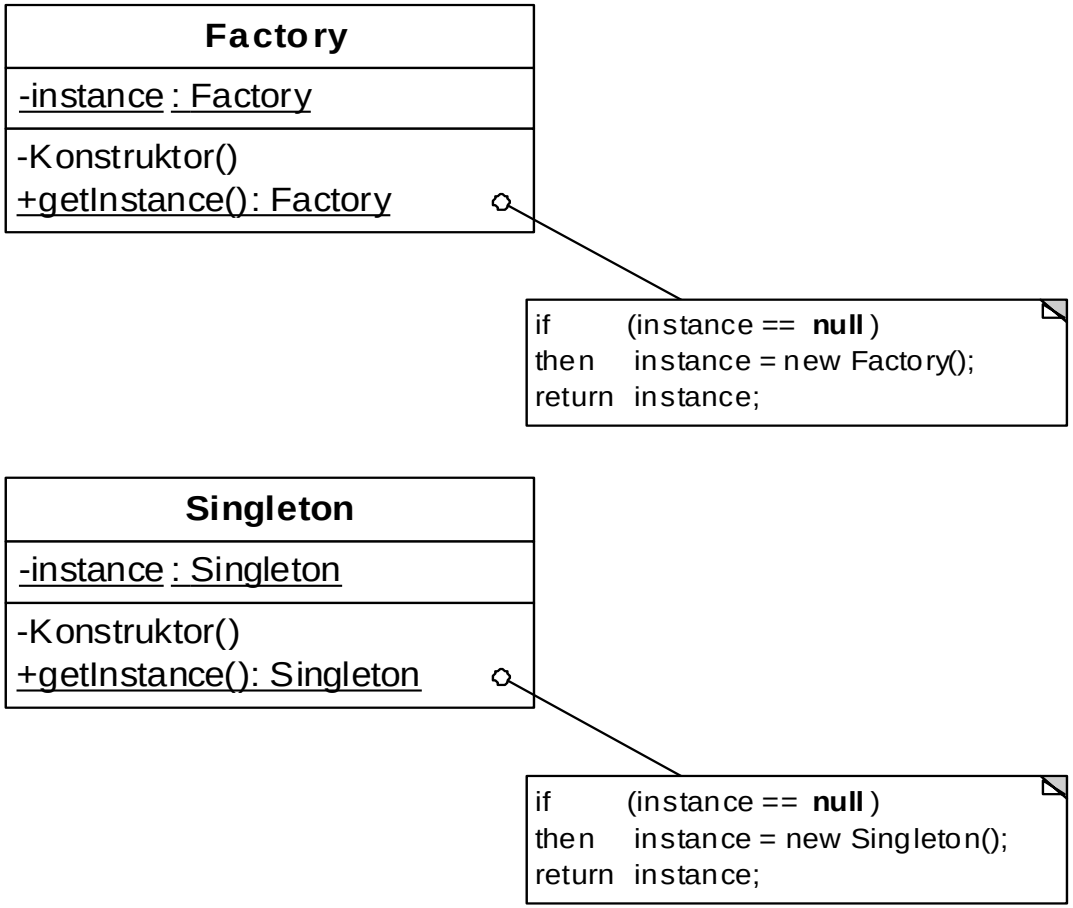


Pattern: Factory (2)



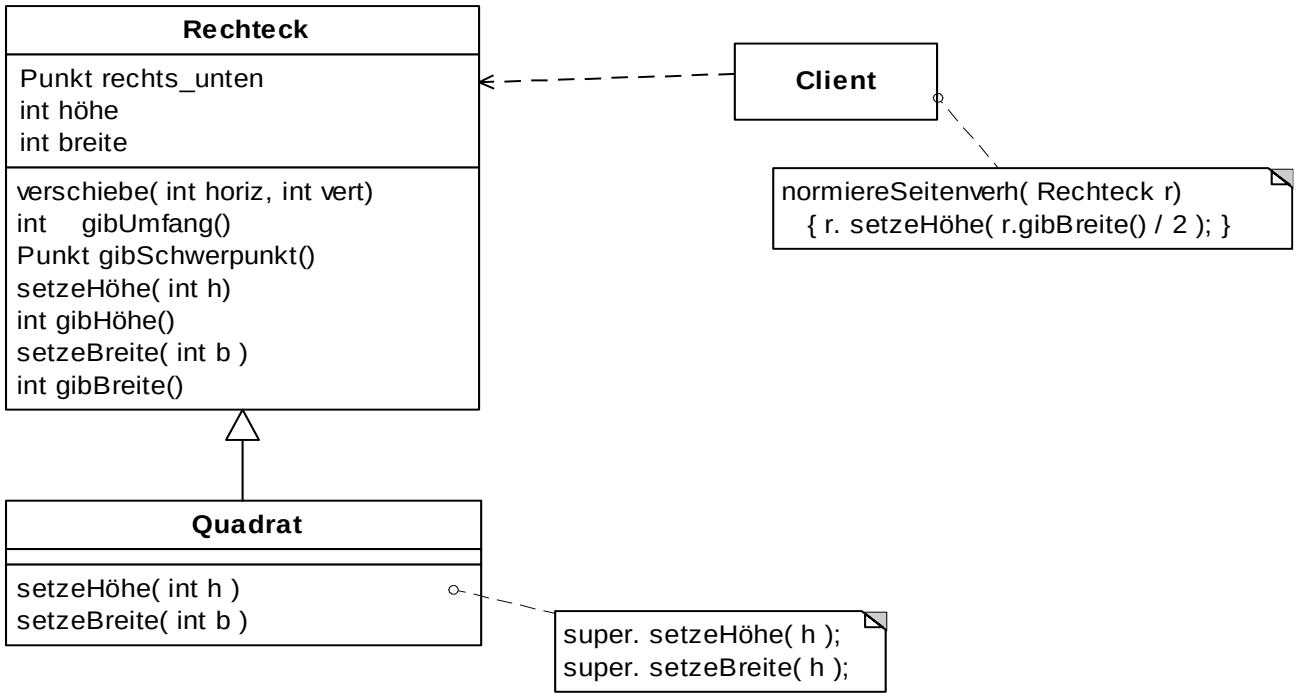
Oben: Framework
Unten: Applikationsspezifisch

Pattern: Singleton

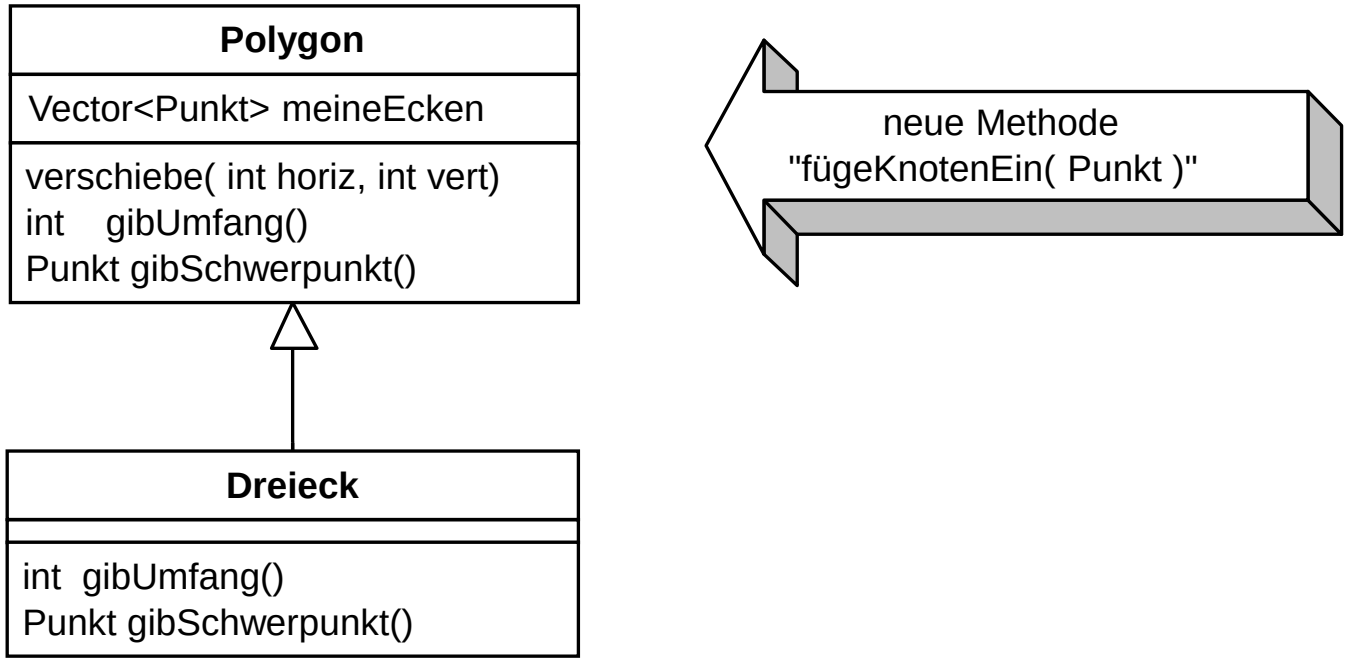


Das Liskov-Substitution-Prinzip (LSP)

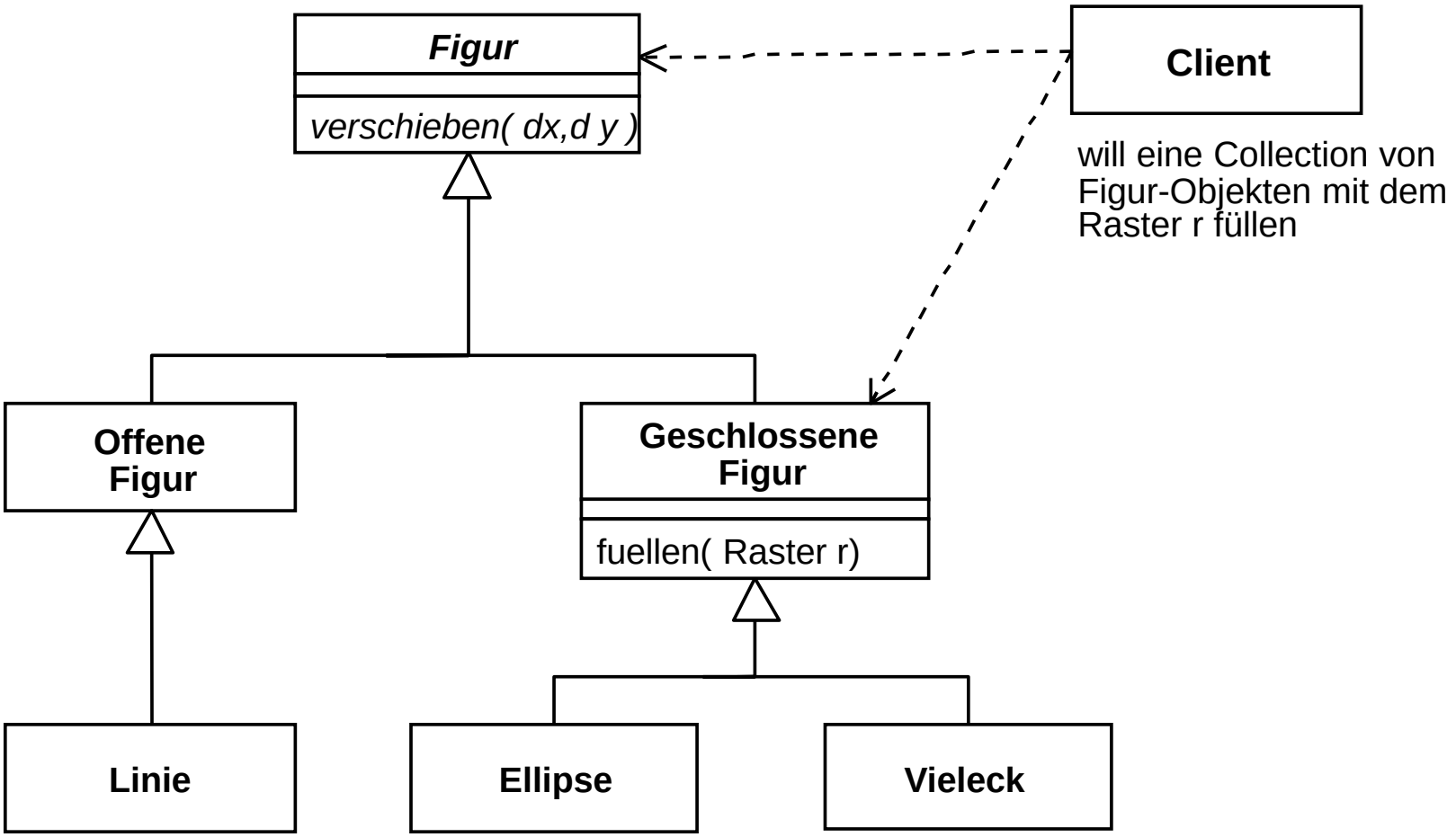
- Eine Unterklasse kann über die Schnittstelle der Oberklasse benutzt werden, ohne dass der Client den Unterschied bemerkt.



Das Liskov-Substitution-Prinzip (LSP)

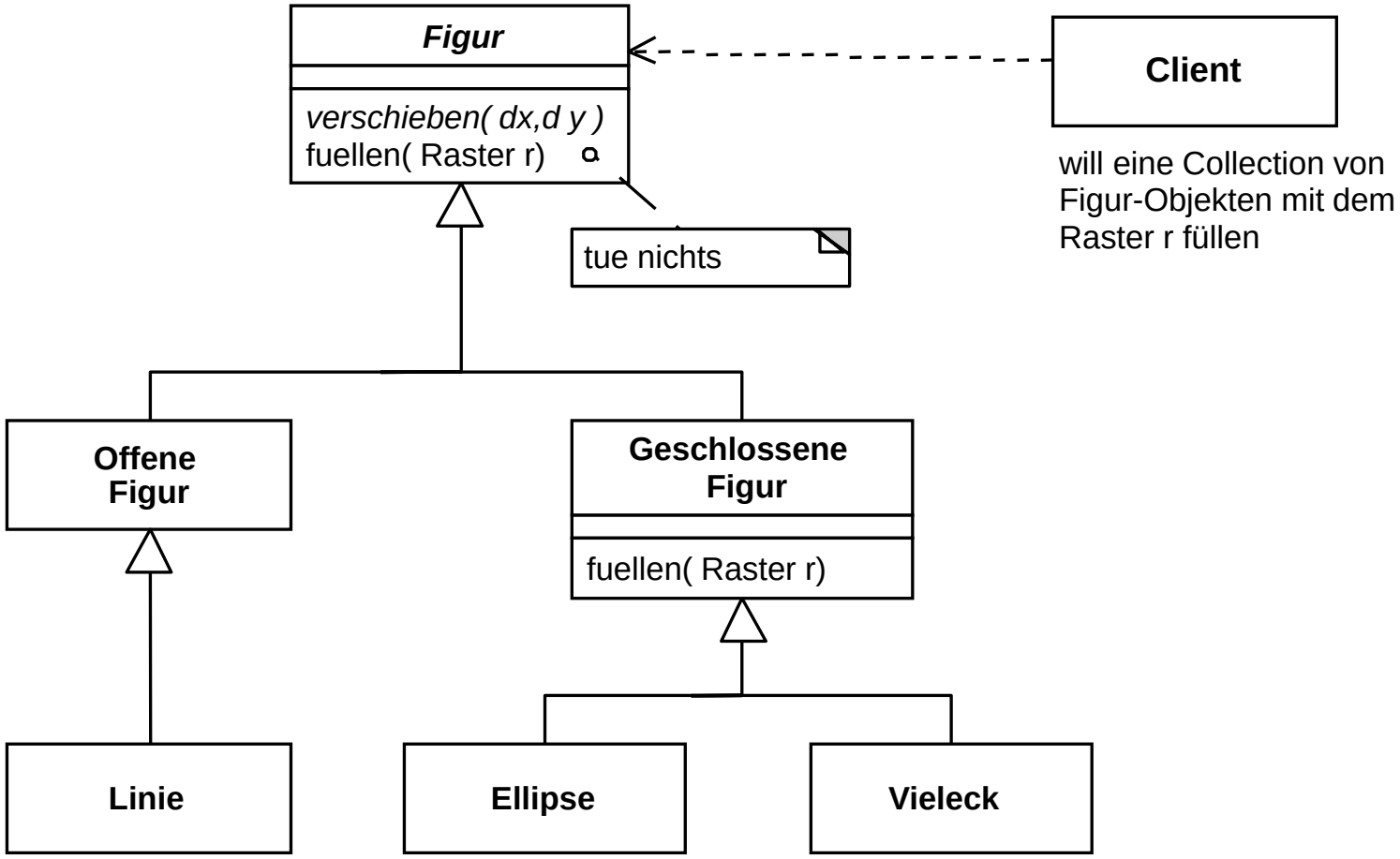


Anpassen der Klassenhierarchie

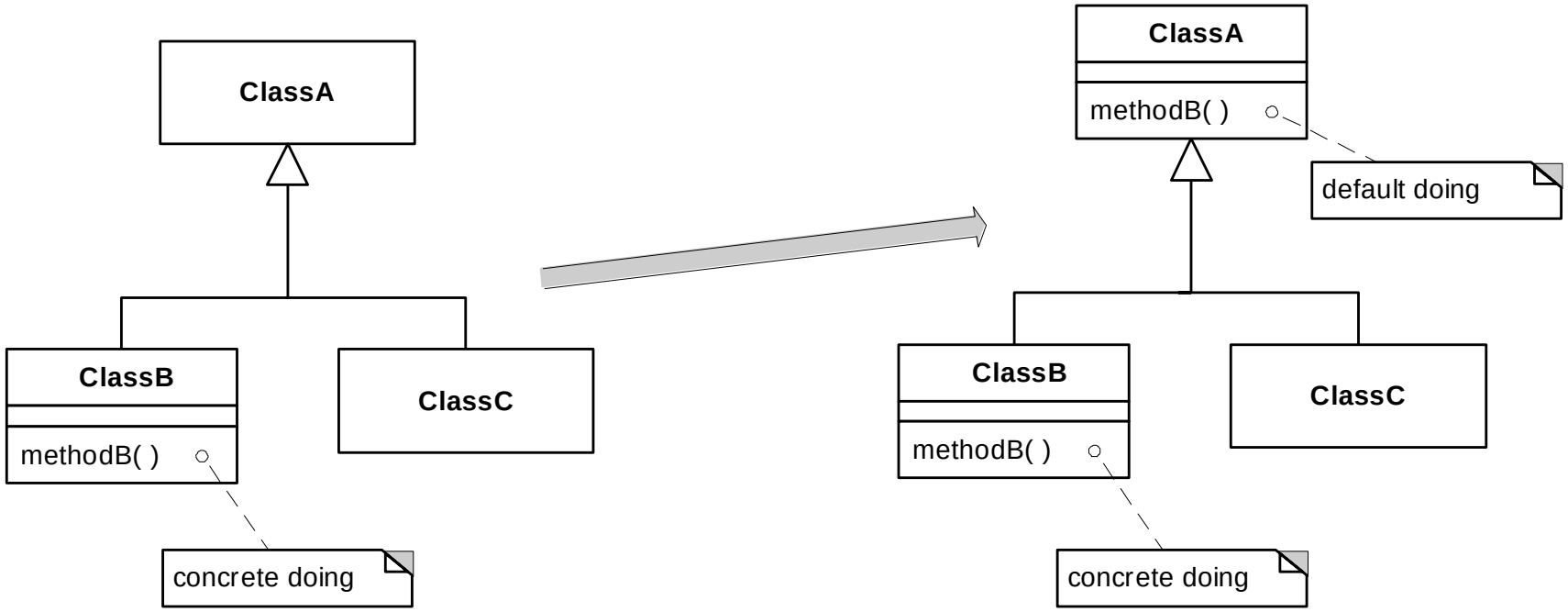


Anpassen der Klassenhierarchie

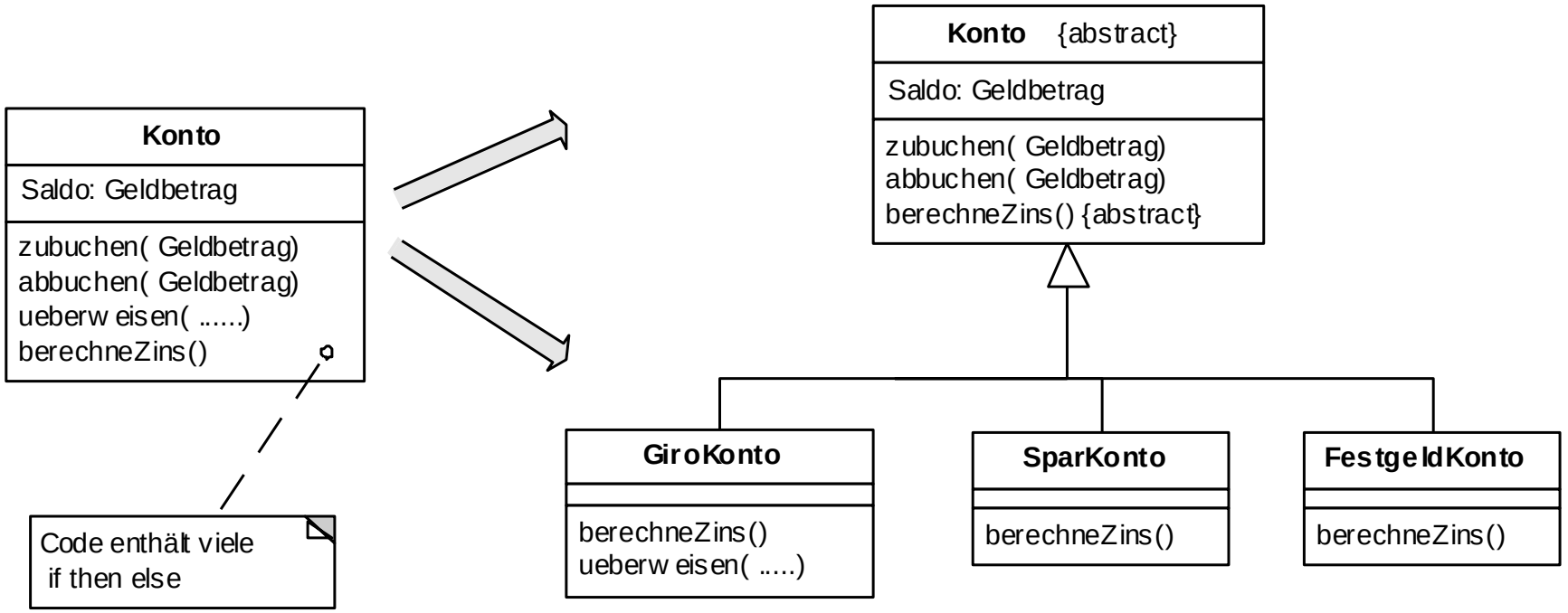
Client kann voll polymorph arbeiten.



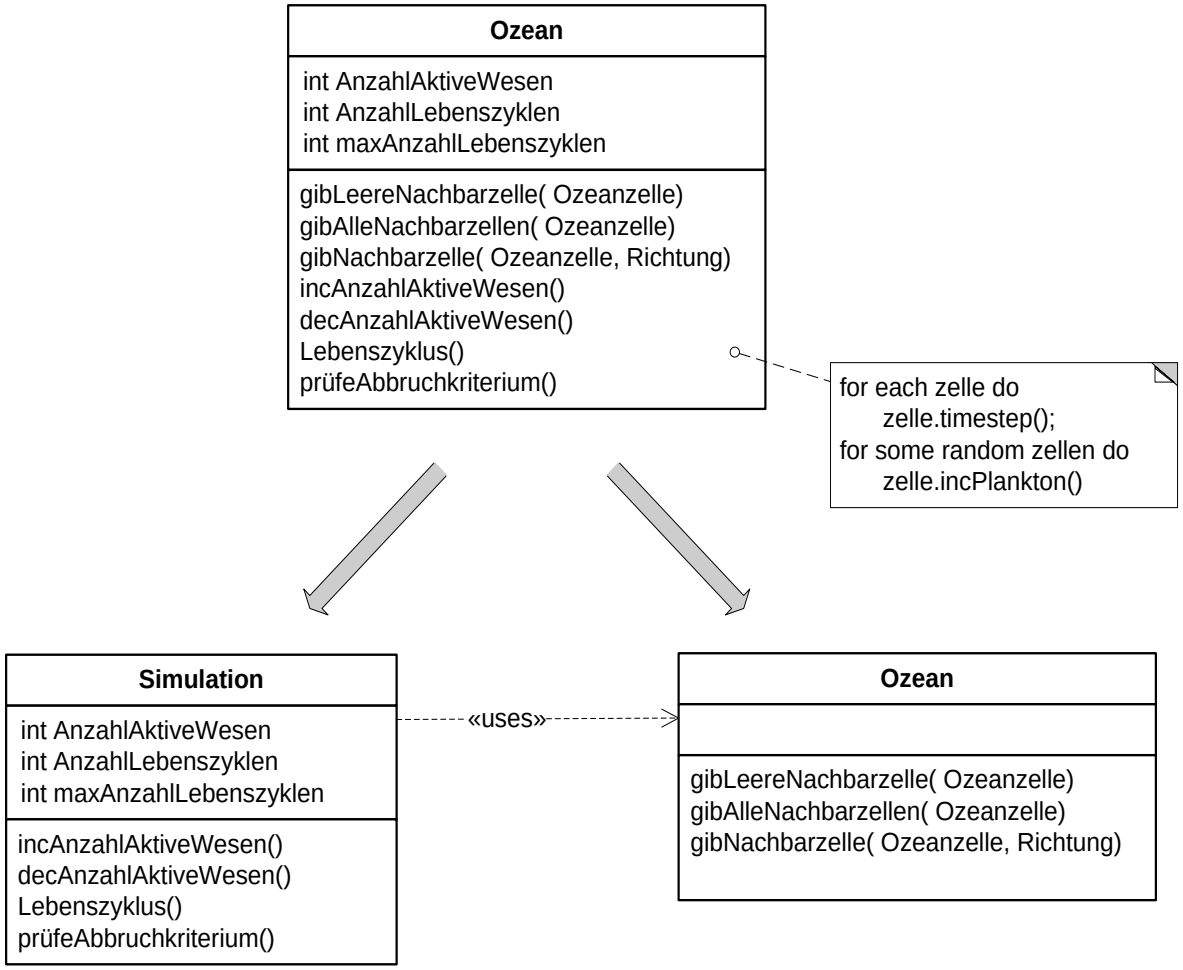
Anpassen der Klassenhierarchie



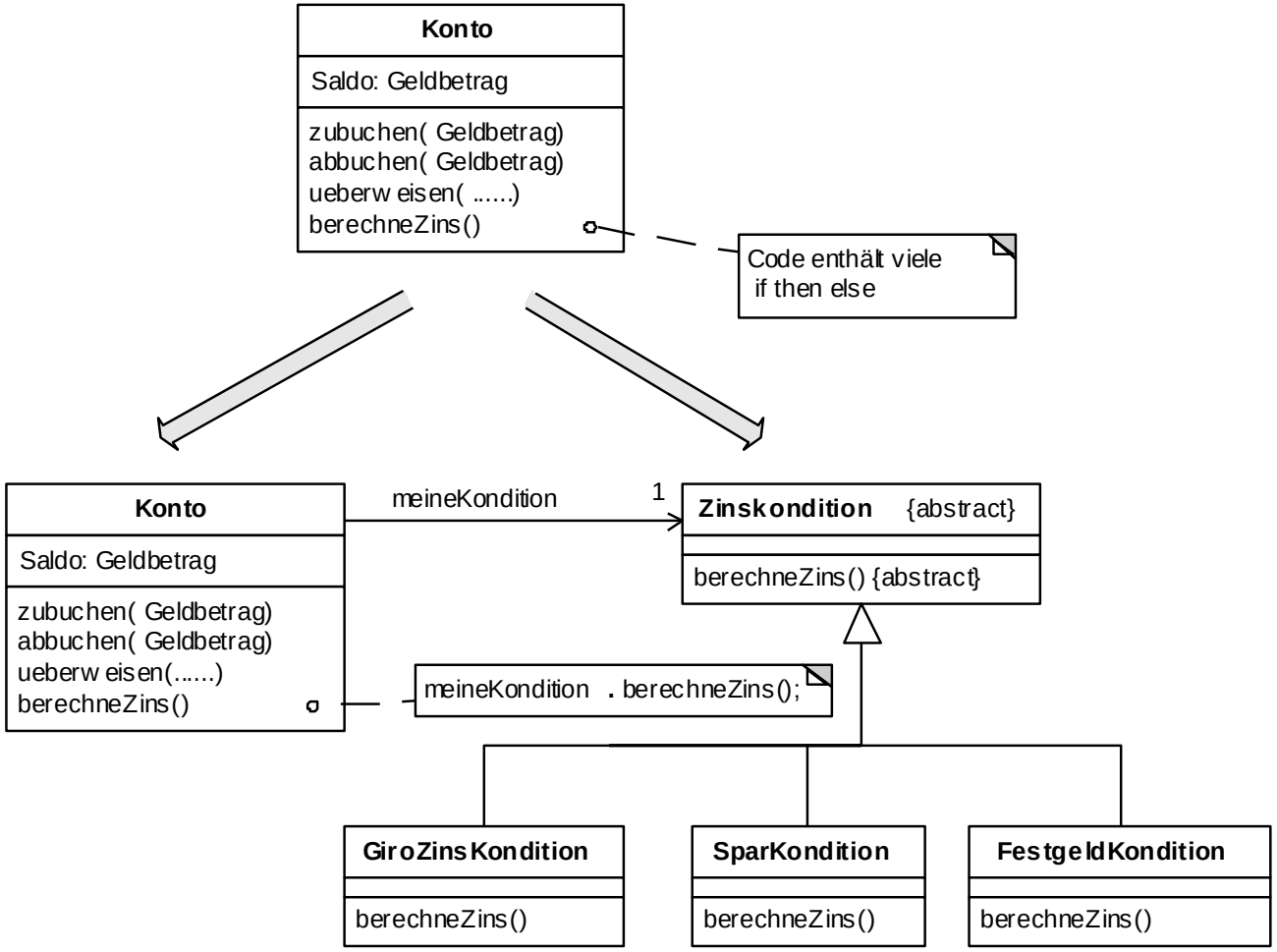
Vertikale Klassenspaltung



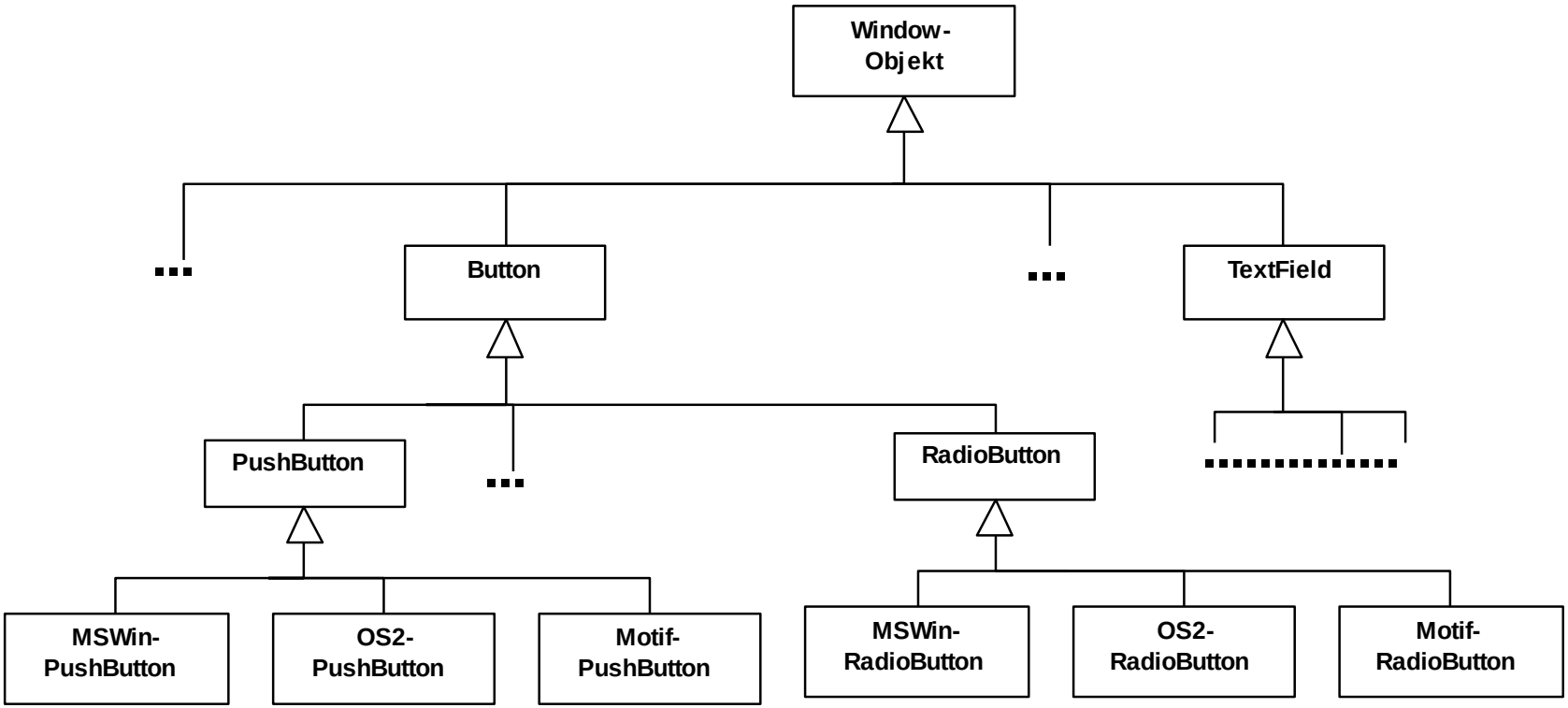
Horizontale Klassenspaltung



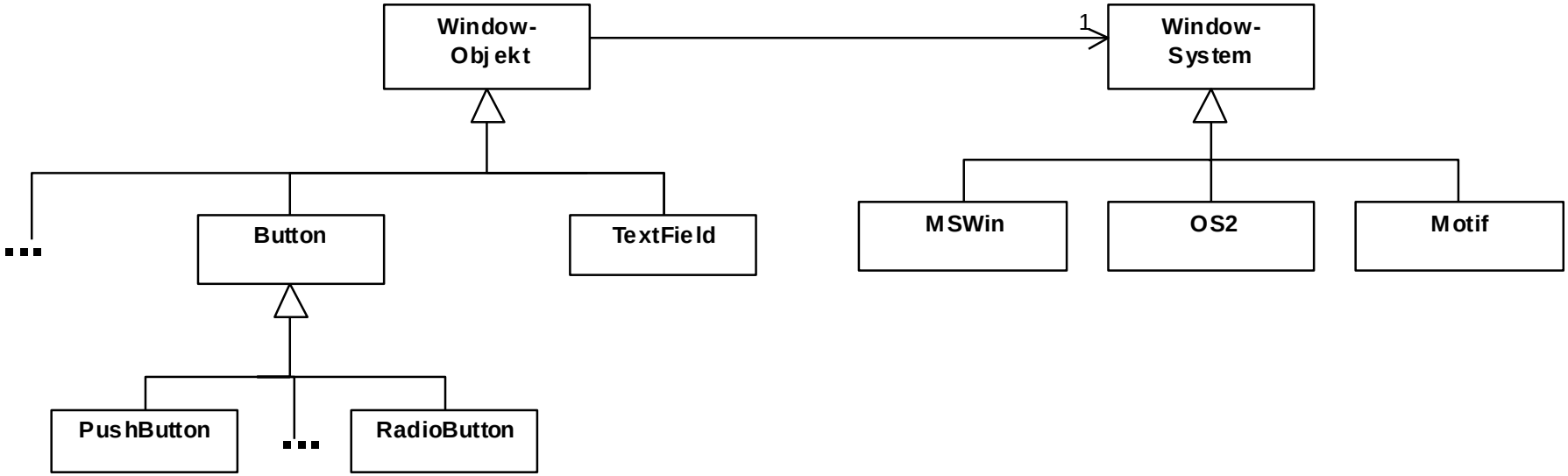
Horizontale Klassenspaltung



Vererbung durch Delegation ersetzen

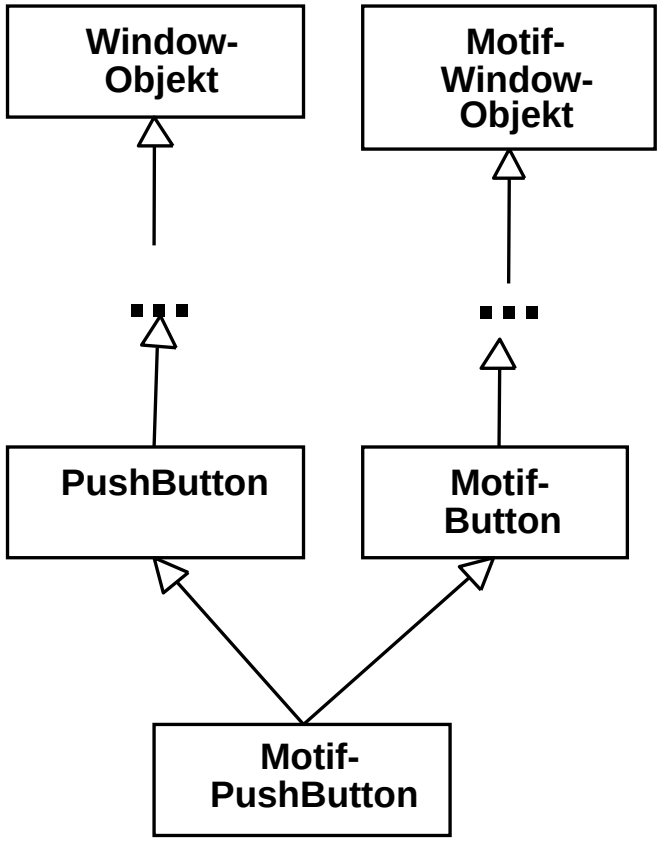


Vererbung durch Delegation ersetzen

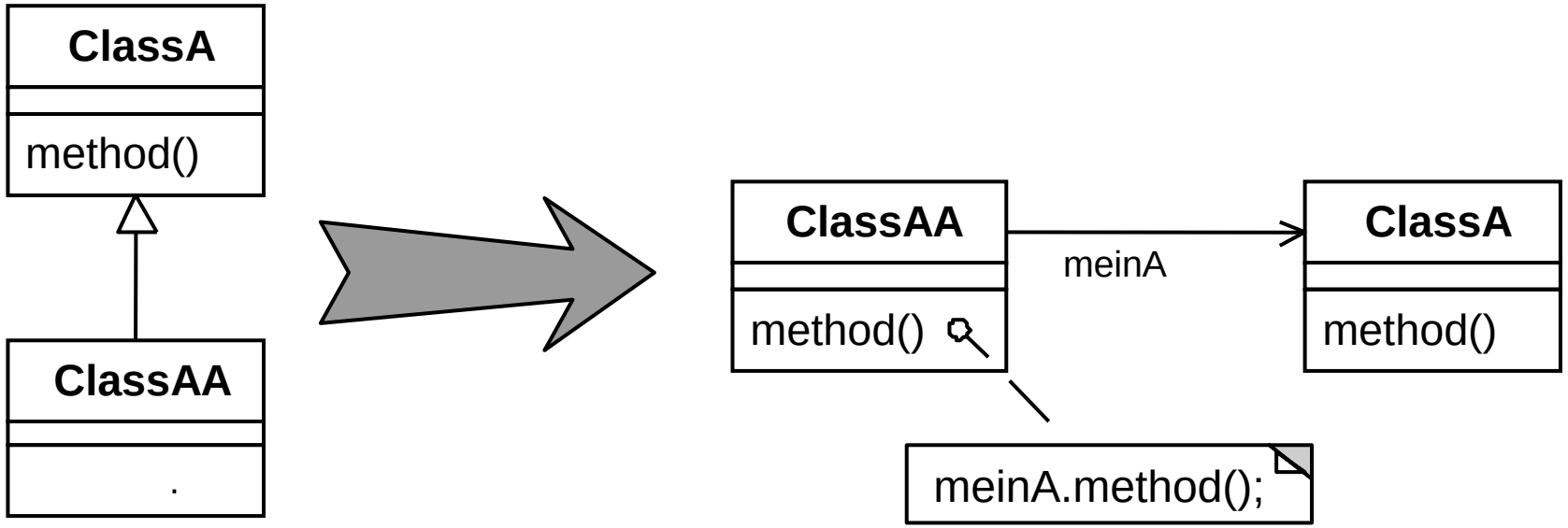


Die Assoziation dient zum Delegieren von Nachrichten: falls ein PushButton die Nachricht *draw()* erhält, delegiert er diese an das WindowSystem-Objekt, z.B. das Motif-Objekt.

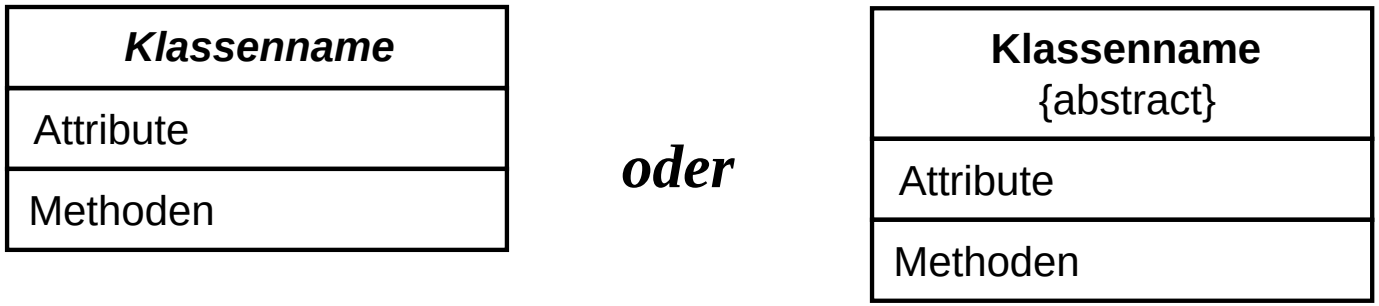
Multiple Vererbung durch Delegation ersetzen



Schema zur Ersetzung von Vererbung durch Delegation



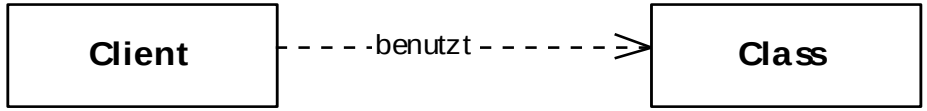
Notation abstrakte Methode / Klasse

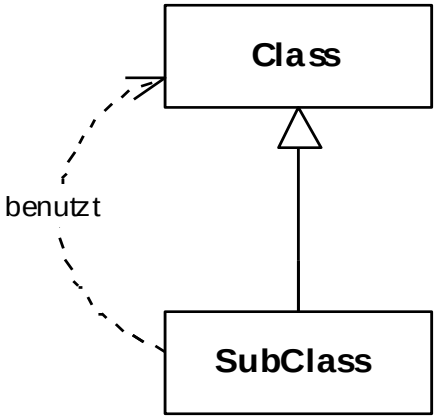


Horizontale / vertikale Schnittstelle

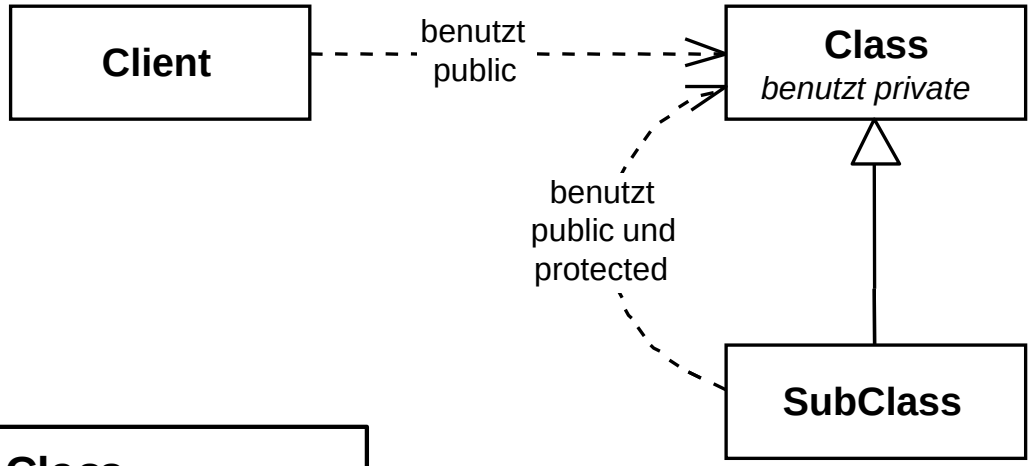
- Horizontale Schnittstelle: die Schnittstelle zum Client

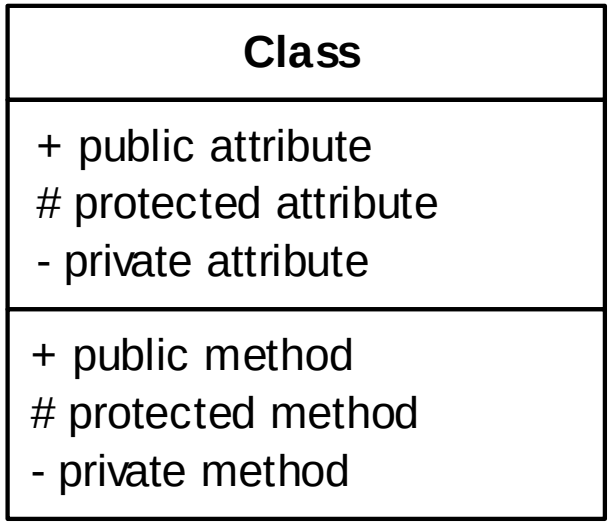
- Vertikale Schnittstelle: die Schnittstelle zu den Unterklassen





Zugriffsrechte für die horizontale / vertikale Schnittstelle





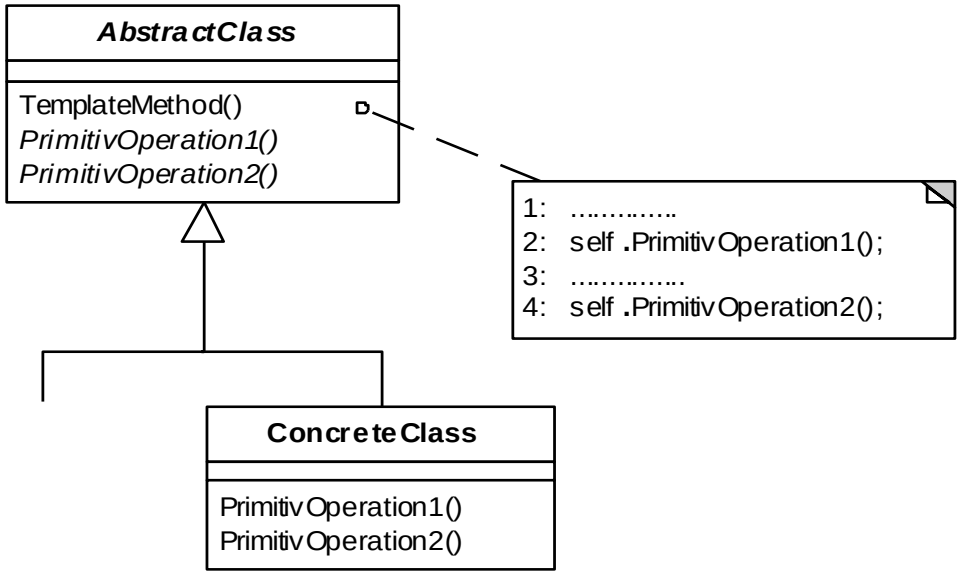
Verhältnis Methoden in Oberklassen zu Unterklassen

Methode in Oberklasse	*Methode in Unterklasse*	*Bezeichnung*
abstrakt	**Vollständige Impl.**	**Abstrakte Methode**
Minimale Impl.	**Meist spezielle, ersetzende Impl.**	**Default-Methode**
Vollständige Impl.	**Wahlfrei durch spezielle Impl. Ersetzt**	**Basis-Methode**
Vollständige Impl.	**Darf nicht ersetzt werden**	**Basis-Methode**
Schematische Impl.	**Einzelschritte sind unterklassenspezifisch**	**Schablonen-Methode**

Schablonen-Methode

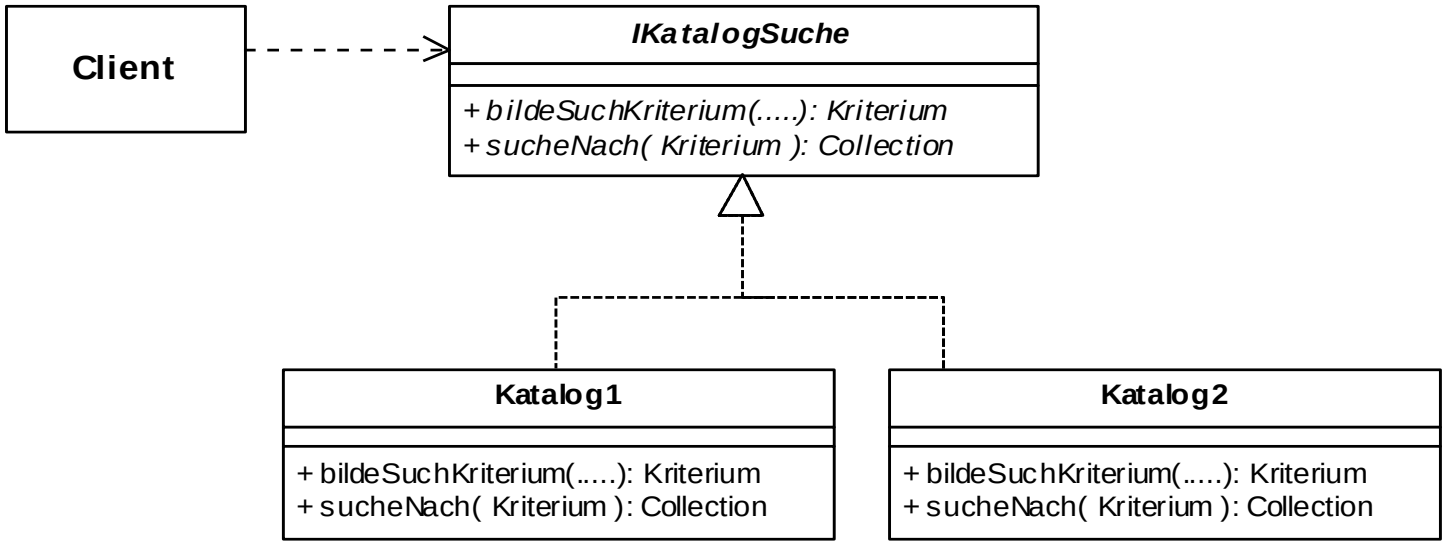
- Schablonen-Methode
  - Implementiert nur eine Schablone (Ablaufsteuerung)
  - Die einzelnen Schritte der Schablone können unterklassenspezifisch sein.

- aufgerufene Methoden sind ...
  - Default-Methoden
  - Basis-Methoden
  - abstrakte Methoden
  - normale Methoden

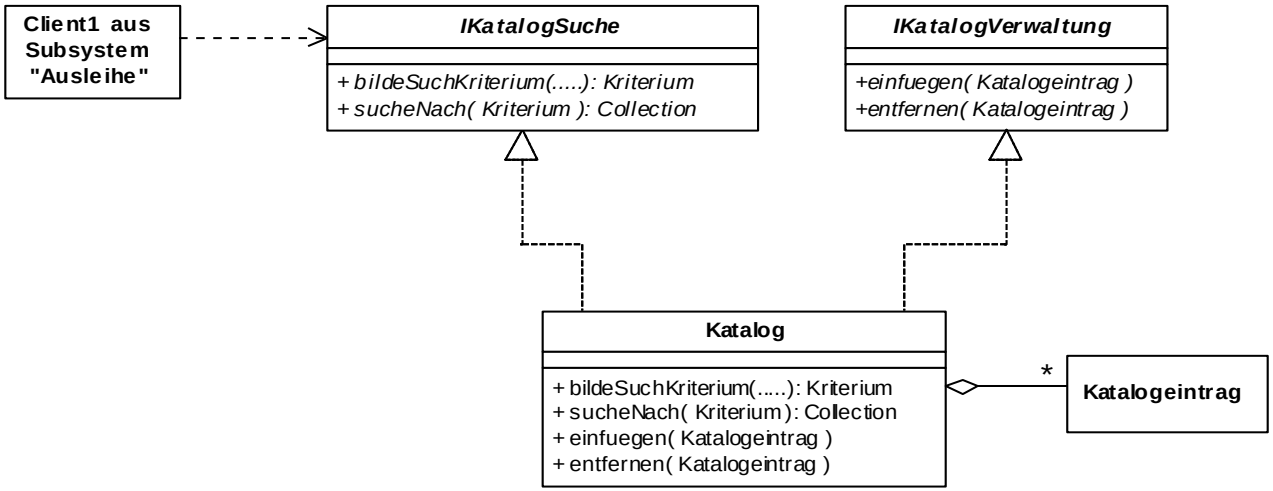


Klasse realisiert Interface

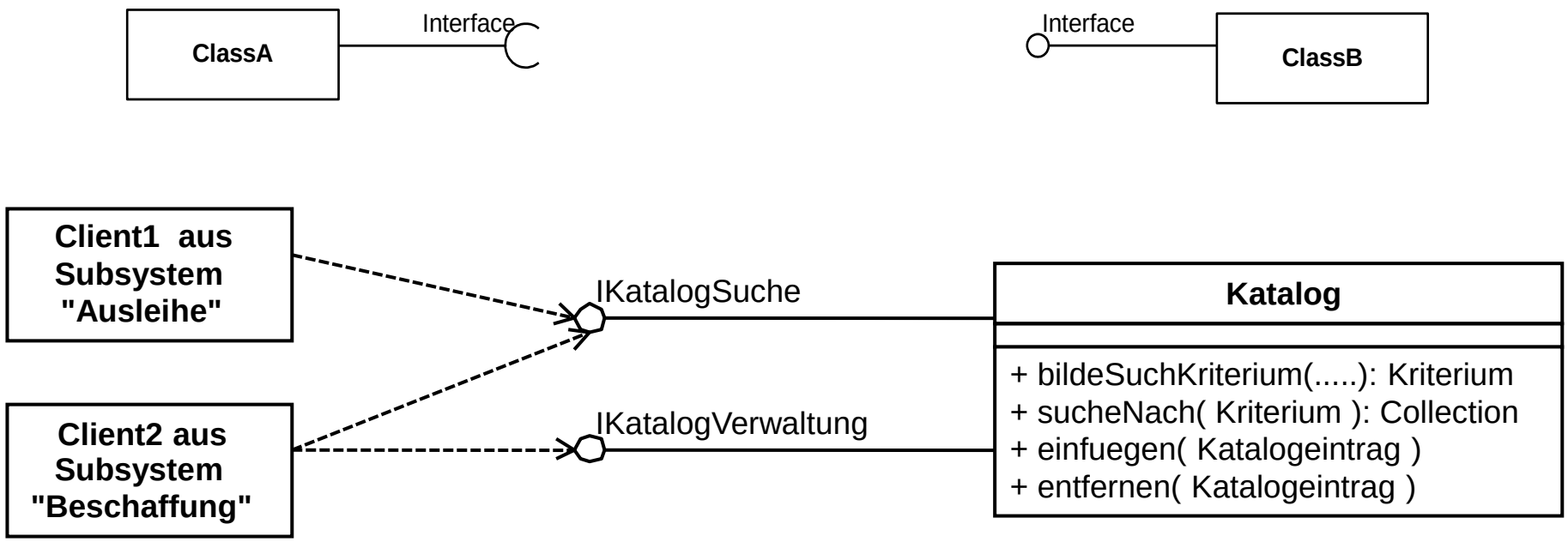
- Eine Klasse *realisiert* ein Interface (gestrichelter Vererbungspfeil)



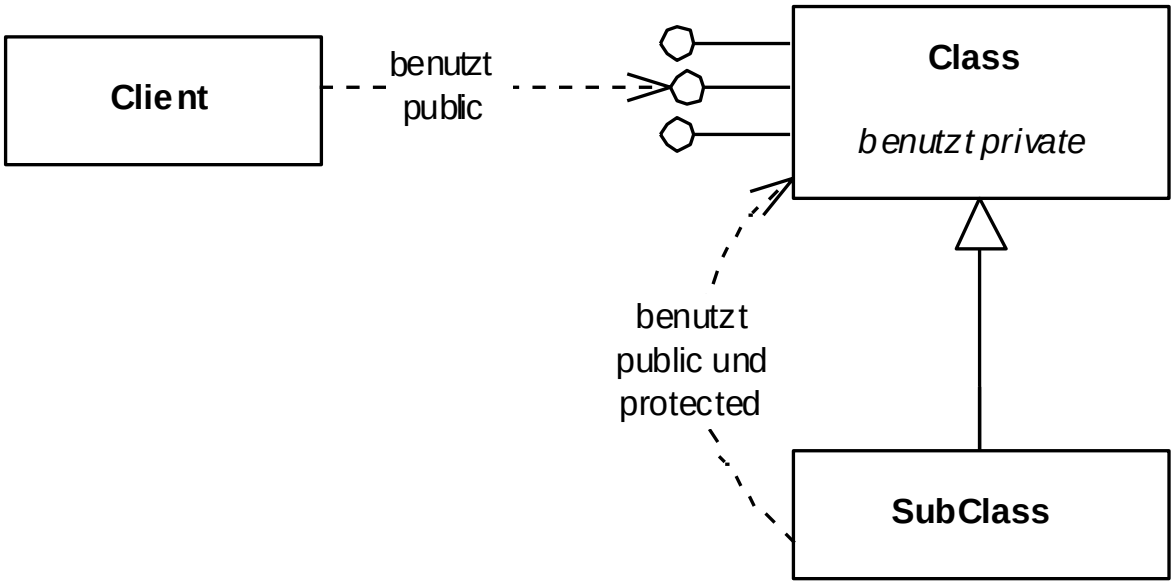
Die gesamte Schnittstelle einer Klasse kann *mehrere* Interfaces beinhalten, von der jede einen bestimmten *Aspekt* der Klasse ausdrückt.



Interfaces: UML-Notation

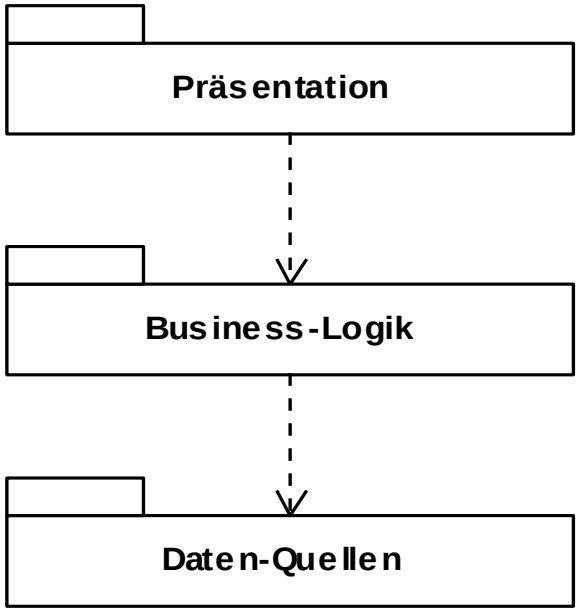


Zusammenfassung



Schichten

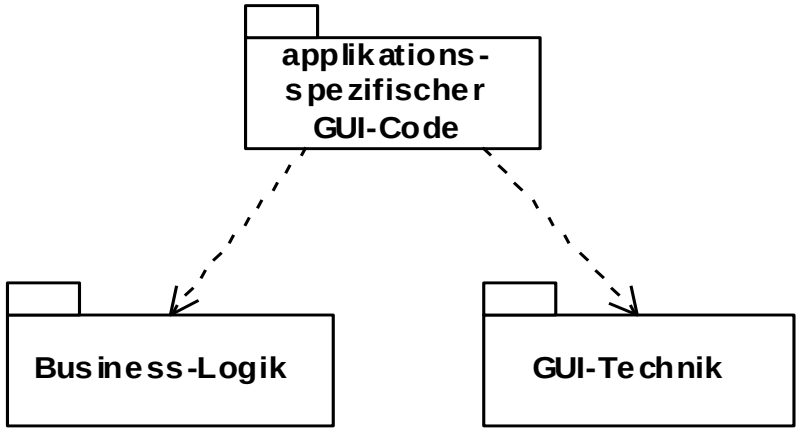
- Eine Schicht ist für sich verständlich
- Schichten sind austauschbar
- Minimale Abhängigkeiten
- Eine untere Schicht, verschiedene darauf aufbauende Schichten



Trennung von Präsentation und Business-Logik

Vorteile:

- wenn nicht getrennt, ist manche Business-Logik mehrfach codiert.
- es sind mehrere Benutzeroberflächen für die gleiche Business-Logik denkbar
- verschiedene Zielumgebungen für die Präsentation
- beide können sich unabhängig voneinander ändern
- Heterogene Systeme (Business-Logik und Präsentation auf verschiedenen Rechnern)
- beide befassen sich grundlegend mit unterschiedlichen Fragestellungen.
- Code zur Präsentation ist sehr stark technisch geprägt
- bessere Spezialisierung der Entwickler

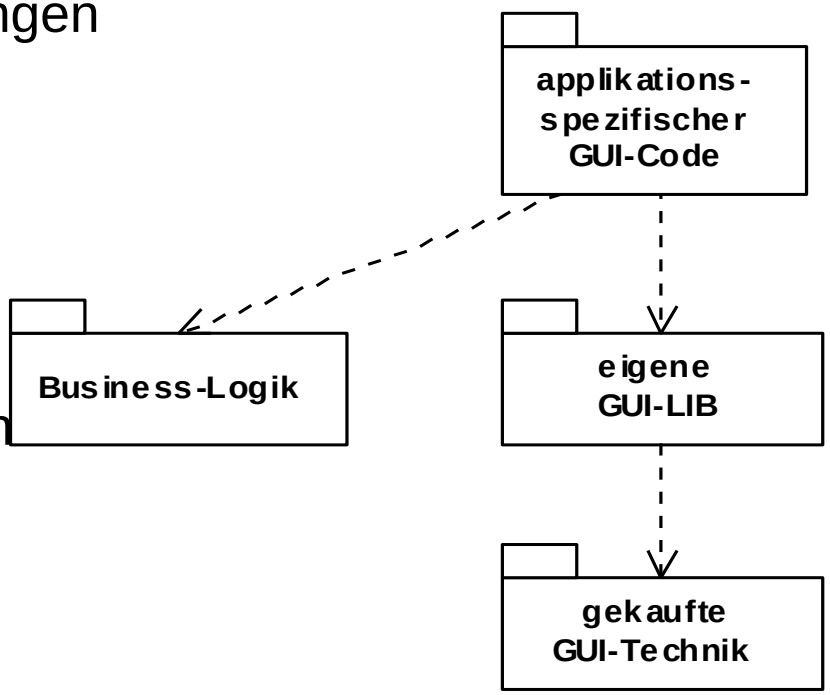


Nachteile:

- sehr aufwendig (teuer) zu erstellen
- zusätzlicher Lernaufwand
- kann Innovationen im GUI-Bereich blockieren (wenn Sun/Microsoft neue Features in der neuen Version ausliefern, dauert es, bis die eigene GUI-Lib diese Features unterstützt)

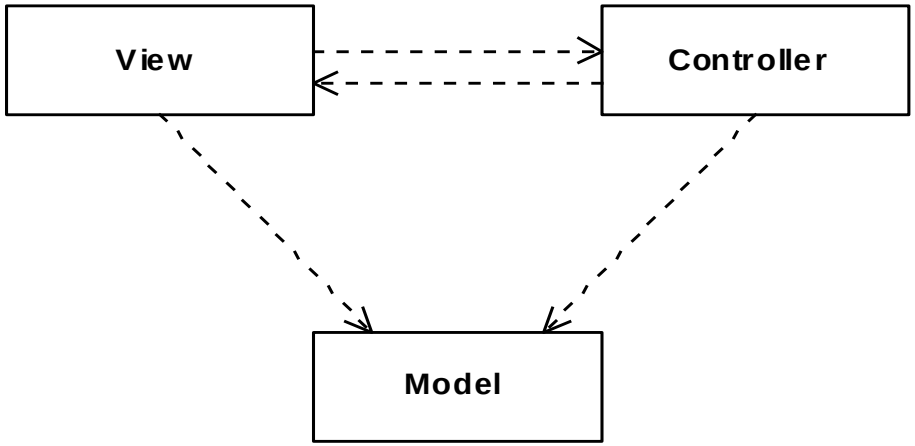
Eigene GUI-Abstraktionsschicht

- Entkoppelt von den Versionszyklen der gekauften GUI-Technik
- Höhere Abstraktionsgrade möglich, verschiedene GUI-Implementierungen
  - MFC oder....
  - Swing oder SWT oder....
  - Swing oder HTML
- Teuer zu erstellen
- Zusätzlicher Lernaufwand
- Kann GUI-Innovationen blockieren



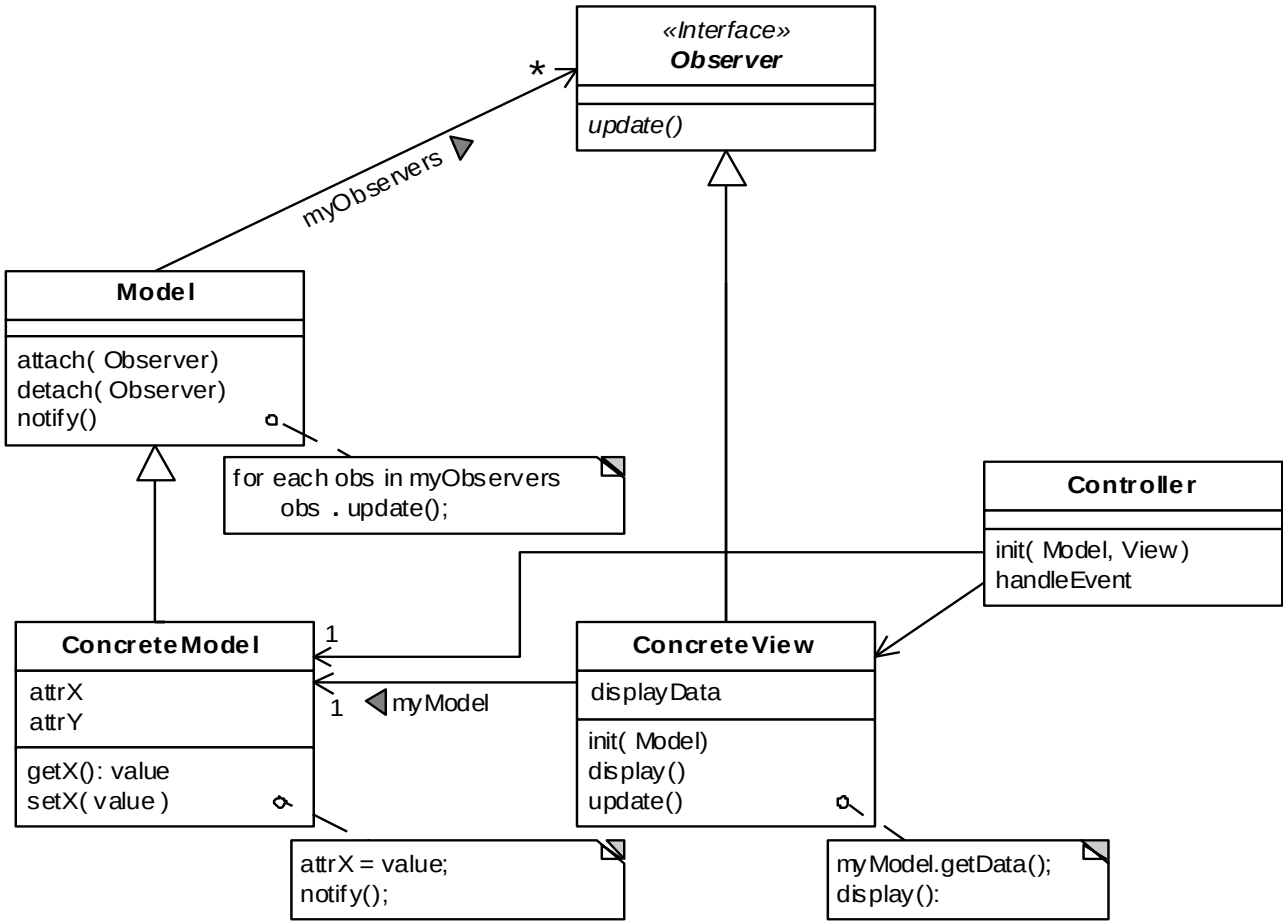
Die Kopplung von Präsentation und Business-Logik: MVC

- Model: enthält die Applikationsdaten und -Logik
- View: ist für die Darstellung verantwortlich
- Controller: ist für die GUI-Steuerung verantwortlich.
  Der Controller nimmt Benutzereingaben entgegen und führt die entsprechenden Änderungen auf dem Model durch.



MVC: Synchronisation mehrerer Views

- Die Views werden bei Änderungen vom Model benachrichtigt.
- Die Benachrichtig ist abstrakt (Interface).



Die Objekt-Identität

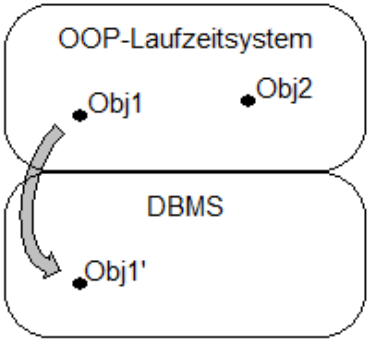


Abbildung der Identität zwischen OOP und DB

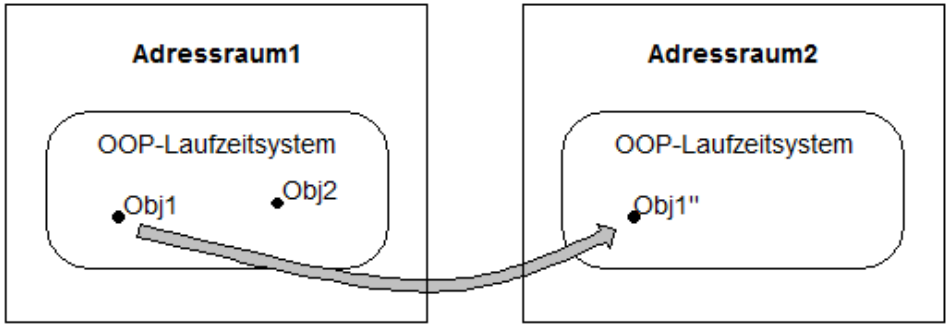


Abbildung der Identität zwischen Adressraum 1 und Adressraum 2

RDBMS: Abbildung einer Klasse

- 1 Klasse => 1 Tabelle

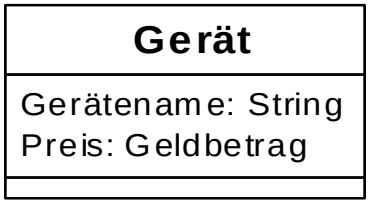


Tabelle ***Gerät***

Attributname	**Definitionsbereich**	**NULL?**
~~Geräte-ID~~	OID	Nein
Gerätename	string	Nein
Preis	Geldbetrag	Ja

RDBMS: N:M-Assoziationen

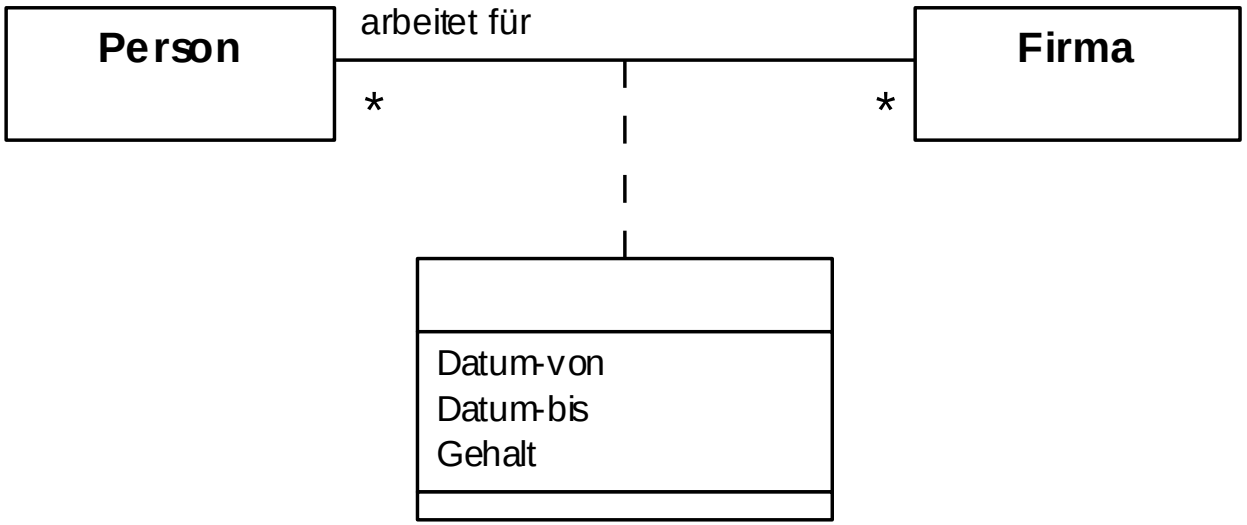


Tabelle ***arbeitet-für***

Attributname	Definitionsbereich	NULL?
Person-ID	OID	Nein
Firma-ID	OID	Nein
Datum-von	Date	Nein
Datum-bis	Date	Ja
Gehalt	Geldbetrag	Ja

RDBMS: 1:N-Assoziationen

Tabelle ***Firma***

Attributname	Definitionsbereich	NULL?
Firma-ID	OID	Nein
Name	string	Nein
Adresse	Adresse	Ja

Tabelle ***Gerät***

Attributname	Definitionsbereich	NULL?
Geräte-ID	OID	Nein
Gerätename	string	Nein
Preis	Geldbetrag	Ja

Tabelle ***stellt_her***

Attributname	Definitionsbereich	NULL?
Firma-ID	OID	Nein
Geräte-ID	OID	Nein

RDBMS: 1:N-Assoziationen

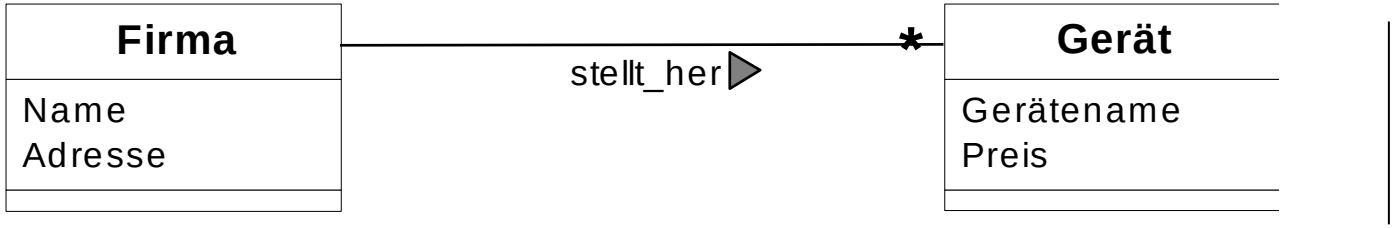


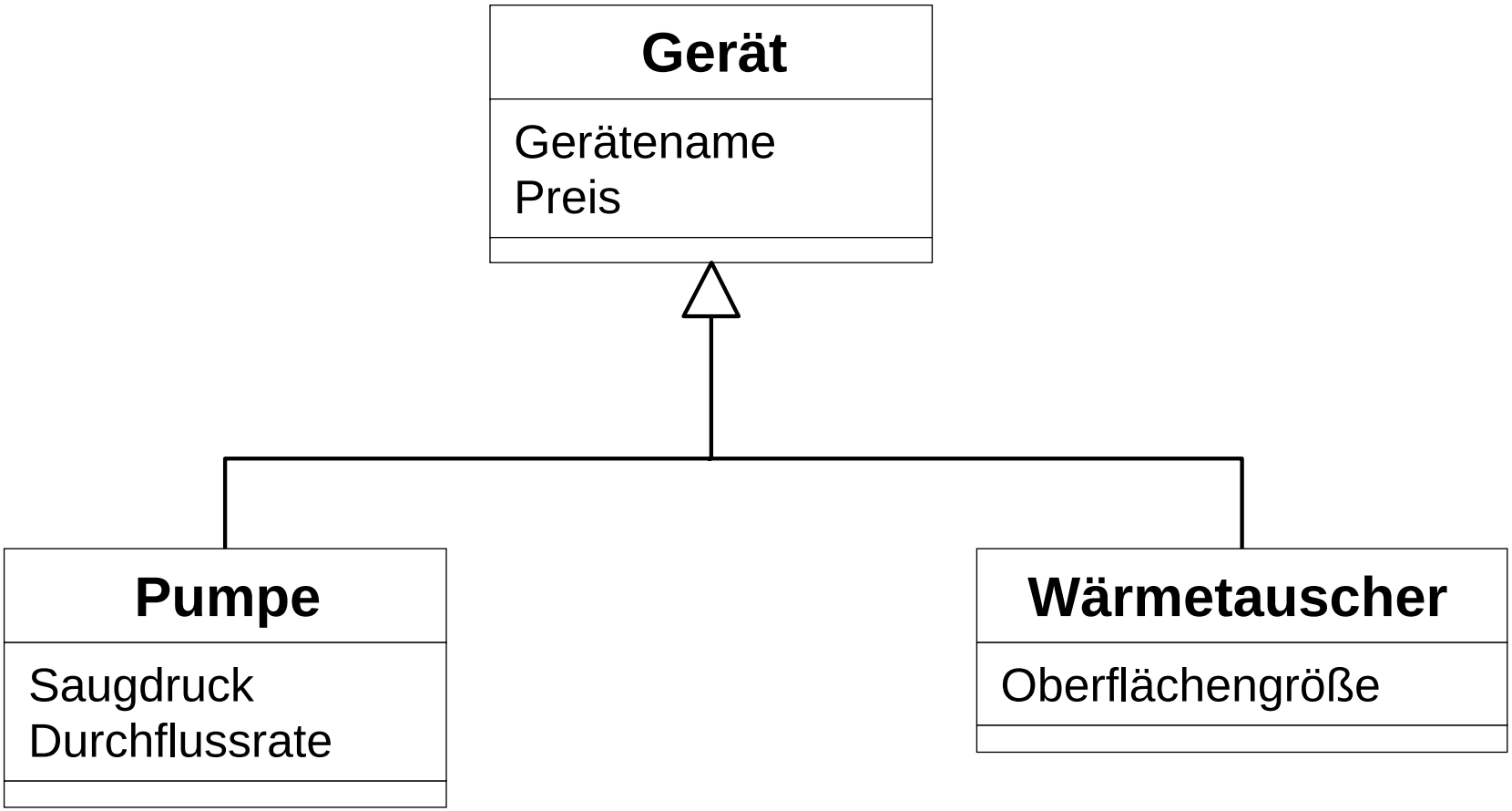
Tabelle ***Firma***

Attributname	Definitionsbereich	NULL?
Firma-ID	OID	Nein
Name	string	Nein
Adresse	Adresse	Ja

Tabelle ***Gerät***

Attributname	Definitionsbereich	NULL?
Geräte-ID	OID	Nein
Gerätename	string	Nein
Preis	Geldbetrag	Ja
Hersteller-Firma	OID	Ja

RDBMS: Vererbung (0)



1. Jede Klasse wird einzeln auf eine Tabelle abgebildet

Tabelle ***Gerät***

Attributname	**Definitionsbereich**
Geräte-ID	OID
Gerätename	string
Preis	Geldbetrag
Gerätetyp	

Tabelle ***Pumpe***

Attributname	**Definitionsbereich**
Geräte-ID	OID
Saugdruck	integer
Durchflussrate	integer

Tabelle ***Wärmetauscher***

Attributname	**Definitionsbereich**
Geräte-ID	OID
Oberfläche	cm^2

2. Wiederholung der Oberklasse-Attribute in jeder Unterklasse

Tabelle ***Pumpe***

Attributname	Definitionsbereich
Geräte-ID	OID
Gerätename	string
Preis	Geldbetrag
Saugdruck	integer
Durchflussrate	integer

Tabelle ***Wärmetauscher***

Attributname	Definitionsbereich
Geräte-ID	OID
Gerätename	string
Preis	Geldbetrag
Oberfläche	cm^2

3. Alle Unterklassen-Attribute in die Oberklasse

Tabelle ***Gerät***

Attributname	**Definitionsbereich**
Geräte-ID	OID
Gerätename	string
Preis	Geldbetrag
Geräte-Typ	Aufzählung
Saugdruck	integer
Durchflussrate	integer
Oberfläche	cm^2