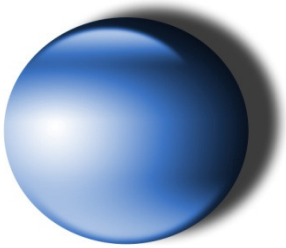
Software Engineering

RWTH AACHEN



Einführung in die Softwaretechnik

Vorlesung, Teil 2

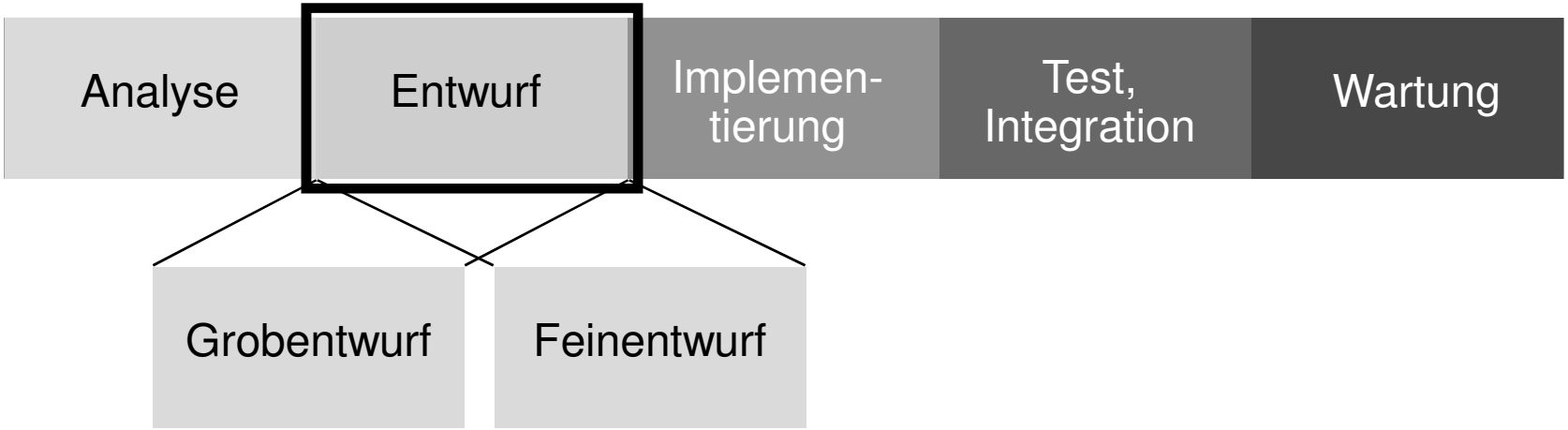
Kapitel 6. Software- und Systementwurf

Prof. Dr. Bernhard Rumpe
Lehrstuhl Informatik 3 (Software Engineering)
Rheinisch-Westfälische Technische Hochschule Aachen

http://www.se-rwth.de/

6. Software- & Systementwurf

6.1. Entwurfsprinzipien



Prof. Dr. Bernhard Rumpe
Software Engineering
RWTH Aachen

http://www.se-rwth.de/

Literatur:

- Sommerville 10
- Balzert Band I LE 23
- Balzert Band II LE 17

Software-Entwurf

- Ausgangspunkt:
  - Anforderungsspezifikation (Pflichtenheft) und
  - Funktionale Spezifikation (Produktdefinition)
- Ziel:
  - Vom “WAS" zum “WIE": Vorgabe für Implementierung
- Zentrale Begriffe:
  - Subsystem
    - in sich geschlossen
    - eigenständig funktionsfähig mit definierten Schnittstellen
    - besteht aus Komponenten
  - Komponente
    - Baustein für ein Softwaresystem (z.B. Modul, Klasse, Paket)
    - benutzt andere Komponenten
    - wird von anderen Komponenten benutzt
    - kann auch aus Unterkomponenten bestehen

Gliederung des Entwurfsprozesses

- Architekturentwurf
- Subsystem-Spezifikation
- Schnittstellen-Spezifikation

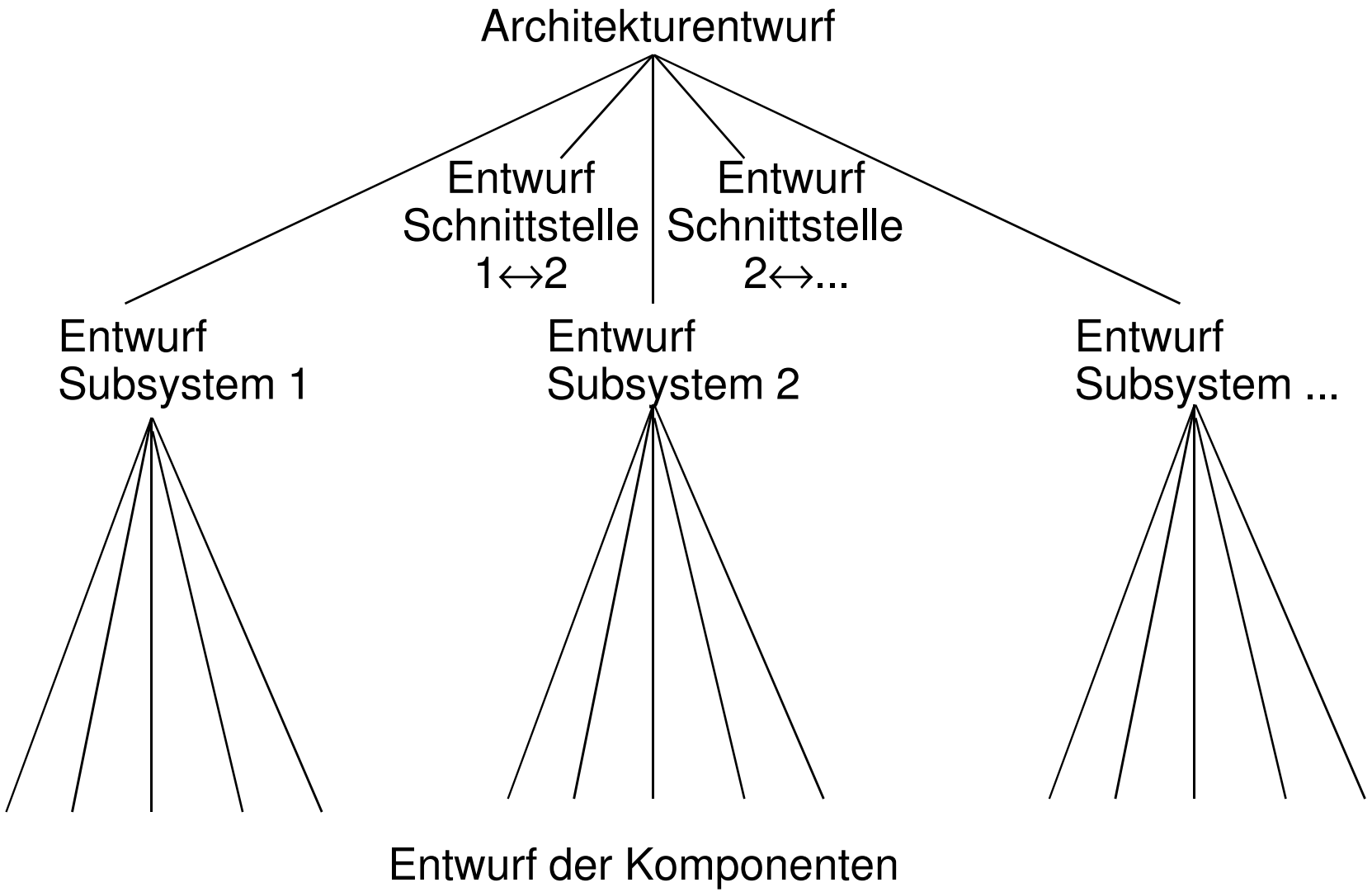
Gesamtstruktur des Systems (Grobentwurf)

- Komponentenentwurf
- Datenstrukturentwurf
- Algorithmenentwurf

Detailstruktur des Systems (Feinentwurf)

- Grobentwurf:
  - weitgehend unabhängig von Implementierungssprache
- Feinentwurf
  - angepasst an die Implementierungssprache und Plattform

Arbeitsteilung beim Entwurf



Kriterien für "guten" Entwurf

- Korrektheit
  - Erfüllung der Anforderungen
  - Wiedergabe aller Funktionen des Systemmodells
  - Sicherstellung der nichtfunktionalen Anforderungen
- Verständlichkeit & Präzision
  - Gute Dokumentation
- Anpassbarkeit
- Hohe Kohäsion innerhalb der Komponenten
- Schwache Kopplung zwischen den Komponenten
- Wiederverwendung

- Diese Kriterien gelten auf allen Ebenen des Entwurfs (Architektur, Subsysteme, Komponenten)

Kohäsion

- Kohäsion ist ein Maß für die Zusammengehörigkeit der Bestandteile einer Komponente.
- Hohe Kohäsion einer Komponente erleichtert Verständnis, Wartung und Anpassung.
- Frühere Ansätze zur Kohäsion wie:
  - ähnliche Funktionalitäten zusammenfassen
    - führten *nicht* unbedingt zu stabiler Systemstruktur
- Bessere Kohäsion wird erreicht durch:
  - Prinzipien der Objektorientierung (Datenkapselung)
  - Einhaltung von Regeln zur Paketbildung
  - Verwendung geeigneter Muster zu Kopplung und Entkopplung
  - "Kohärente" Klasse:
    - Es gibt keine Partitionierung in Untergruppen von zusammengehörigen Operationen und Attributen

Kopplung

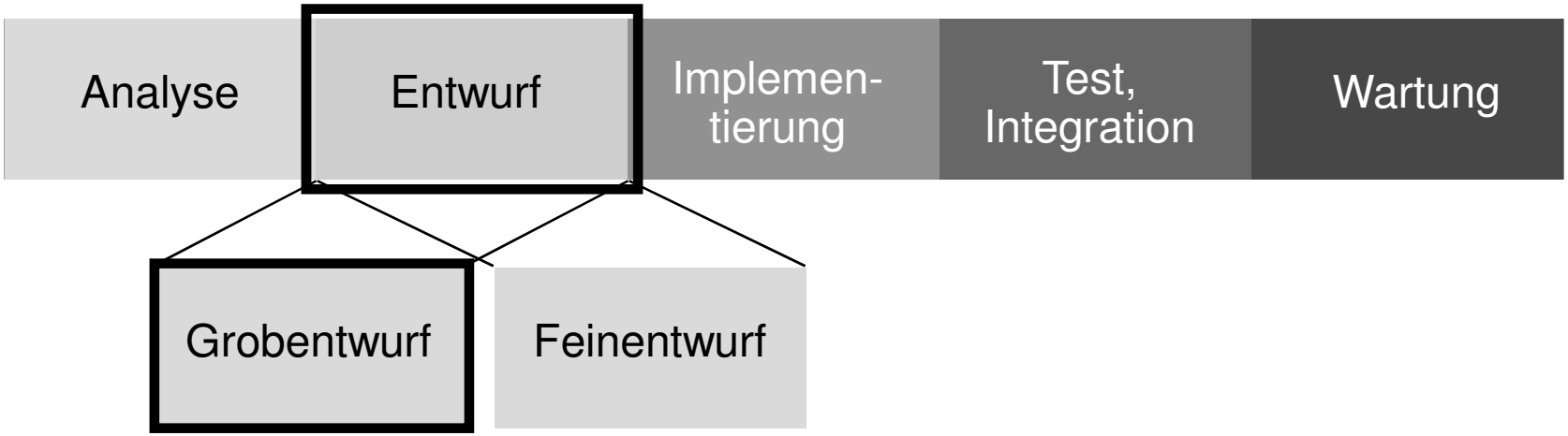
- Kopplung ist ein Maß für die Abhängigkeiten zwischen Komponenten.
- Geringe Kopplung erleichtert die Wartbarkeit und macht Systeme stabiler.
- Arten der Kopplung:
  - Datenkopplung (gemeinsame Daten)
  - Schnittstellenkopplung (gegenseitiger Aufruf)
  - Strukturkopplung (gemeinsame Strukturelemente)
- Reduktion der Kopplung:
  - Kopplung kann nie auf Null reduziert werden!
  - Schnittstellenkopplung ist akzeptabel, da höhere Flexibilität
  - Datenkopplung vermeiden!
    - aber durch Objektorientierung meist gegeben
  - Strukturkopplung vermeiden !
    - z.B. keine Vererbung über Paketgrenzen hinweg
- Entkopplungsbeispiel: get/set-Methoden statt Attributzugriff

Interne Wiederverwendung

- Interne Wiederverwendung (*reuse*) ist ein Maß für die Ausnutzung von Gemeinsamkeiten zwischen Komponenten
- Reduktion der Redundanz
- Erhöhung der Stabilität und Ergonomie
- Hilfsmittel für Wiederverwendung:
  - im objektorientierten Entwurf: Vererbung, Parametrisierung
  - im modularen und objektorientierten Entwurf: Module/Objekte mit allgemeinen Schnittstellen (Interfaces)
- Aber: Wiederverwendung kann die Kopplung erhöhen:
  - Schnittstellenkopplung und Strukturkopplung

6. Software- & Systementwurf

6.2. Softwarearchitektur



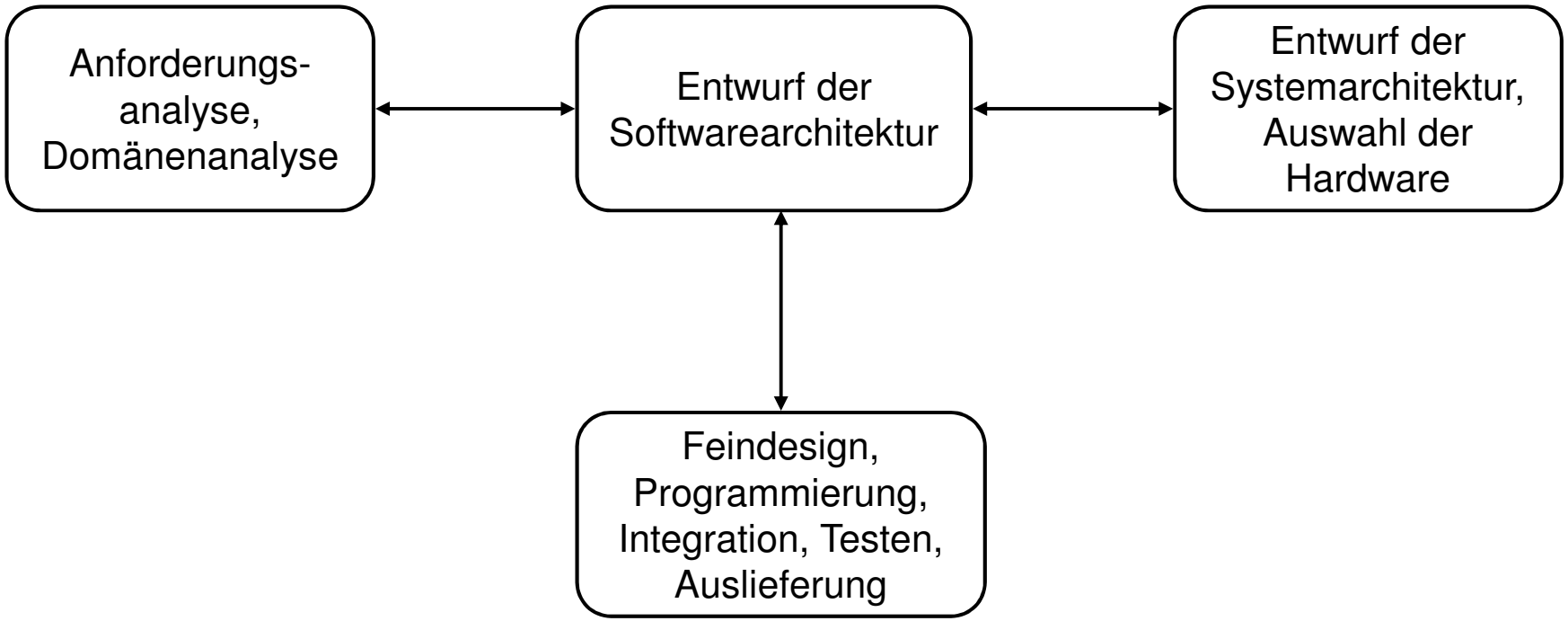
Prof. Dr. Bernhard Rumpe
Software Engineering
RWTH Aachen

http://www.se-rwth.de/

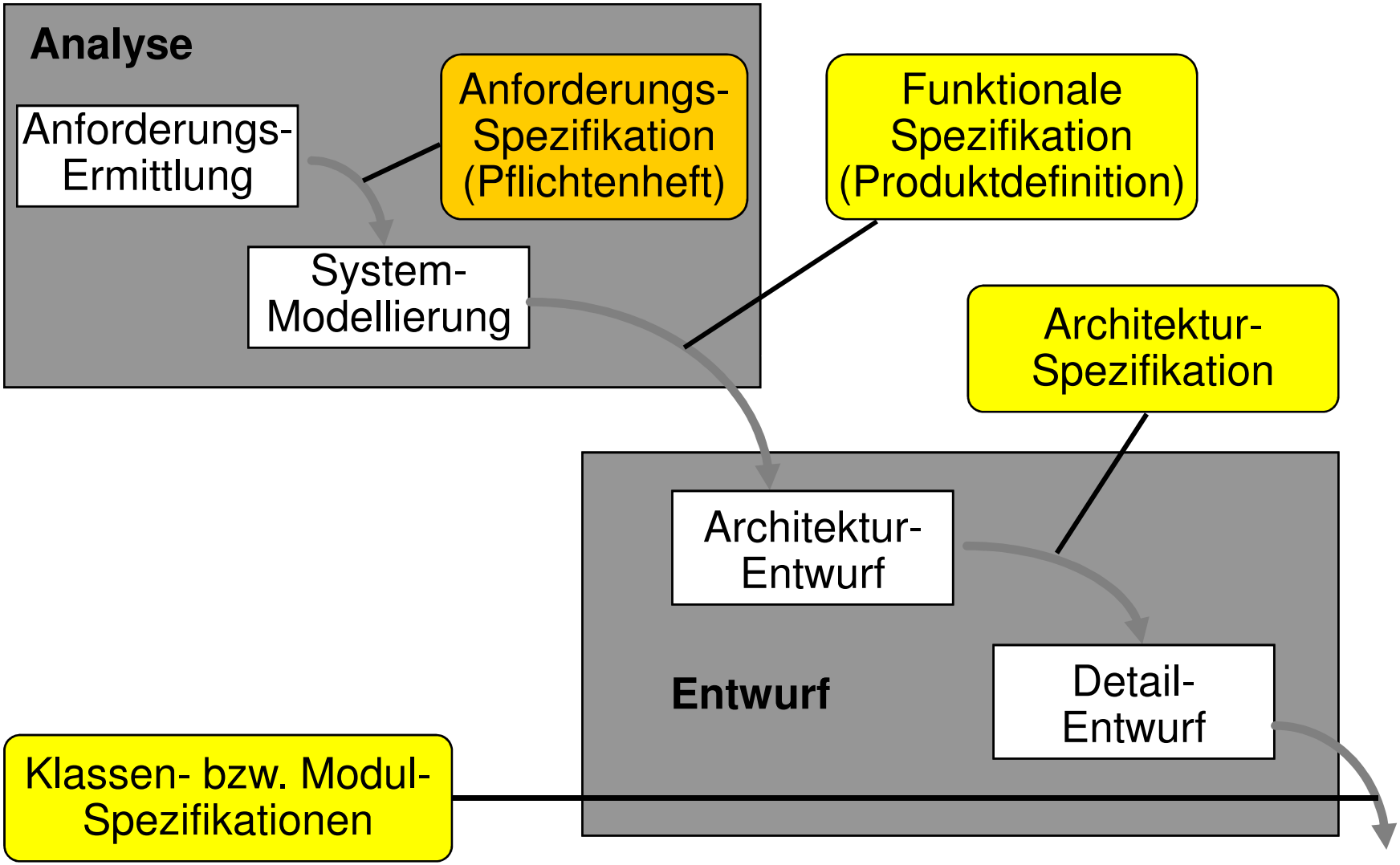
Literatur:

- Sommerville 10
- Balzert LE 23
- Shaw/Garlan: Software Architecture, 1996
- Bass/Clements/Kazman: Software Architecture in Practice, Addison-Wesley 1998
- P. Kruchten, The 4+1 view model of architecture, IEEE Software, Nov. 1995, 12(6)

Architekturentwurf im Kontext der SW-Entwicklung



Von der Analyse zum Entwurf



Softwarearchitektur in der Praxis

- Architekturspezifikation wird zu oft nicht als separates Dokument gefordert.
- Häufig wird funktionale Spezifikation und Architekturspezifikation in einem Dokument realisiert.
  - denn „WAS“ zu spezifizieren, ohne auf grobe Strukturen des „WIE“ einzugehen ist oft nicht möglich.
  - Dennoch: die grobe Systemarchitektur wird der Entwurfs-Aktivität zugeordnet
- Ist Hardware involviert (Steuergeräte im Auto, Telekommunikations-Anlagen etc.), so wird oft bereits dadurch eine physische Architektur vorgegeben. So kann es durchaus sinnvoll sein, erste Architekturskizzen bereits in die Anforderungsbeschreibung aufzunehmen.
- Logische Systemarchitektur und physische Architektur sind nicht notwendig identisch.

Beispielarchitektur

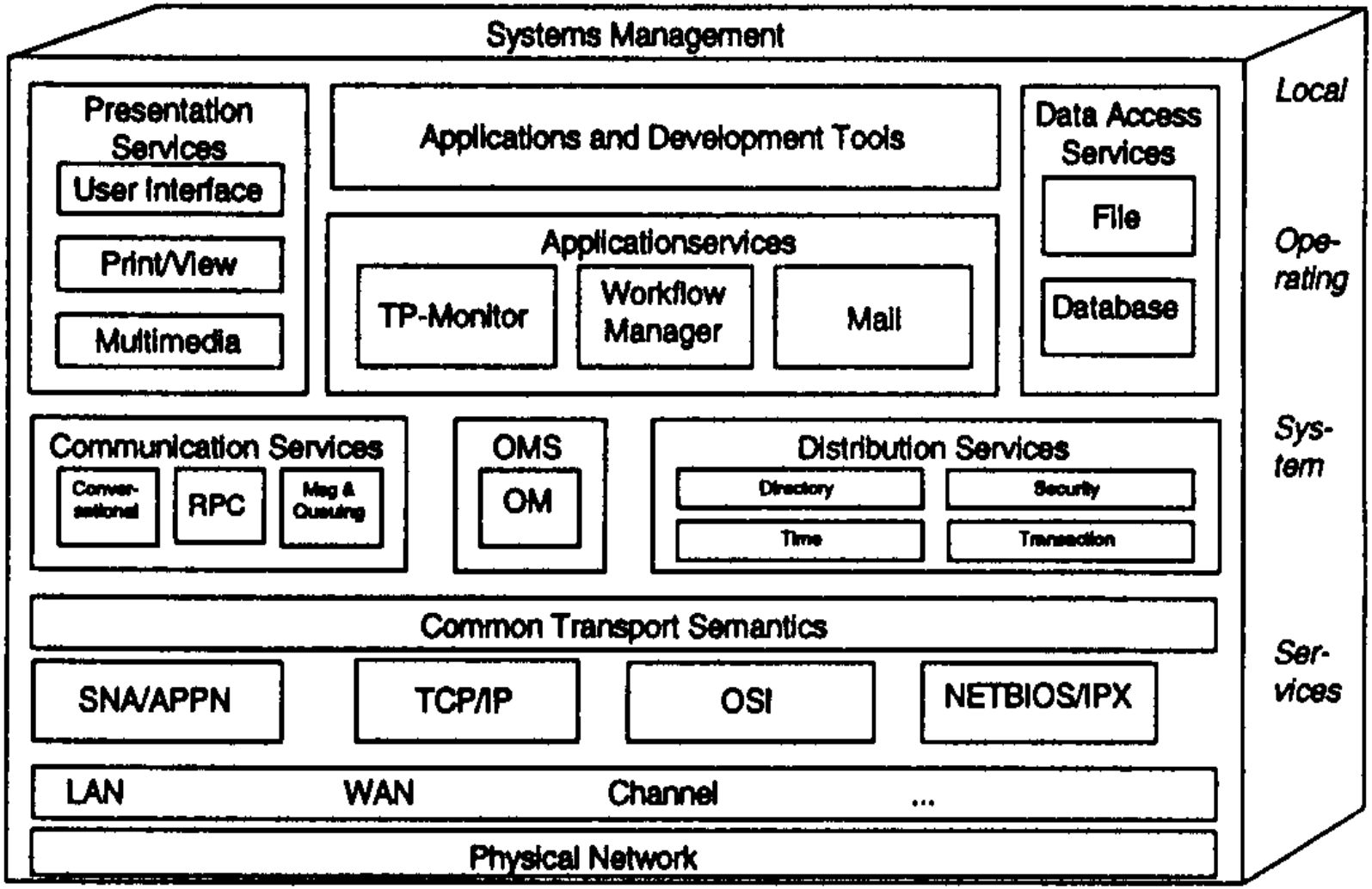
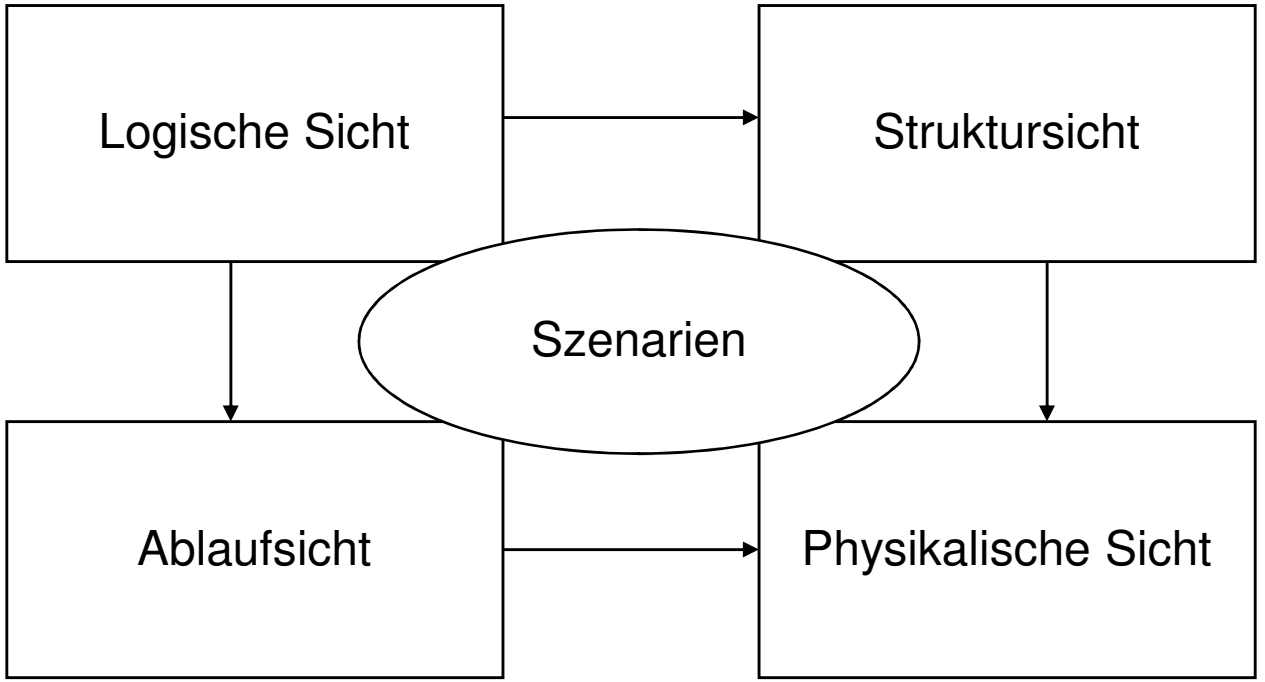


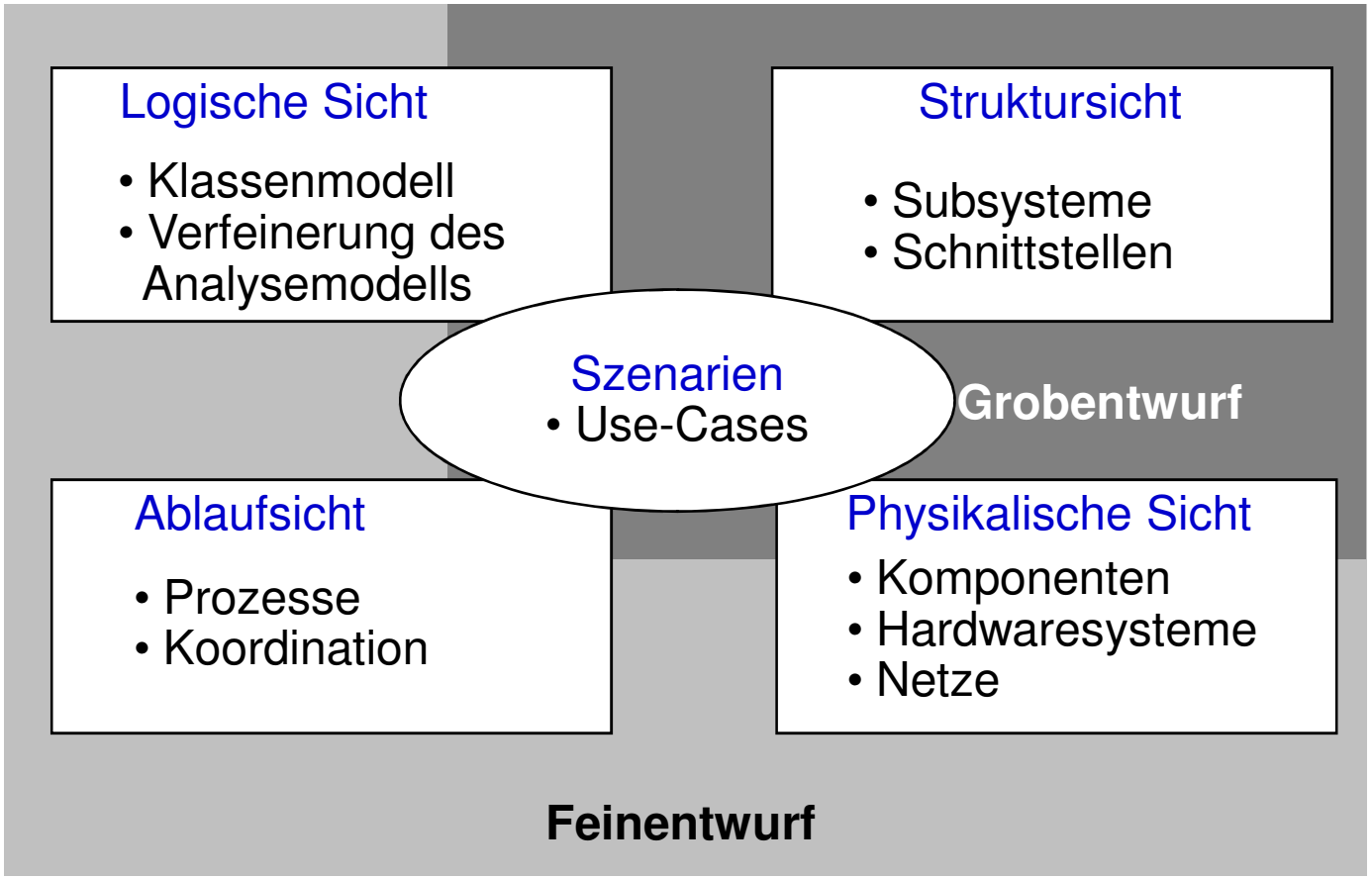
Abb. 4.3. IBMs Open Blueprint

„4+1 Sichten“-Modell der Softwarearchitektur (aus dem Rational Unified Process - RUP)

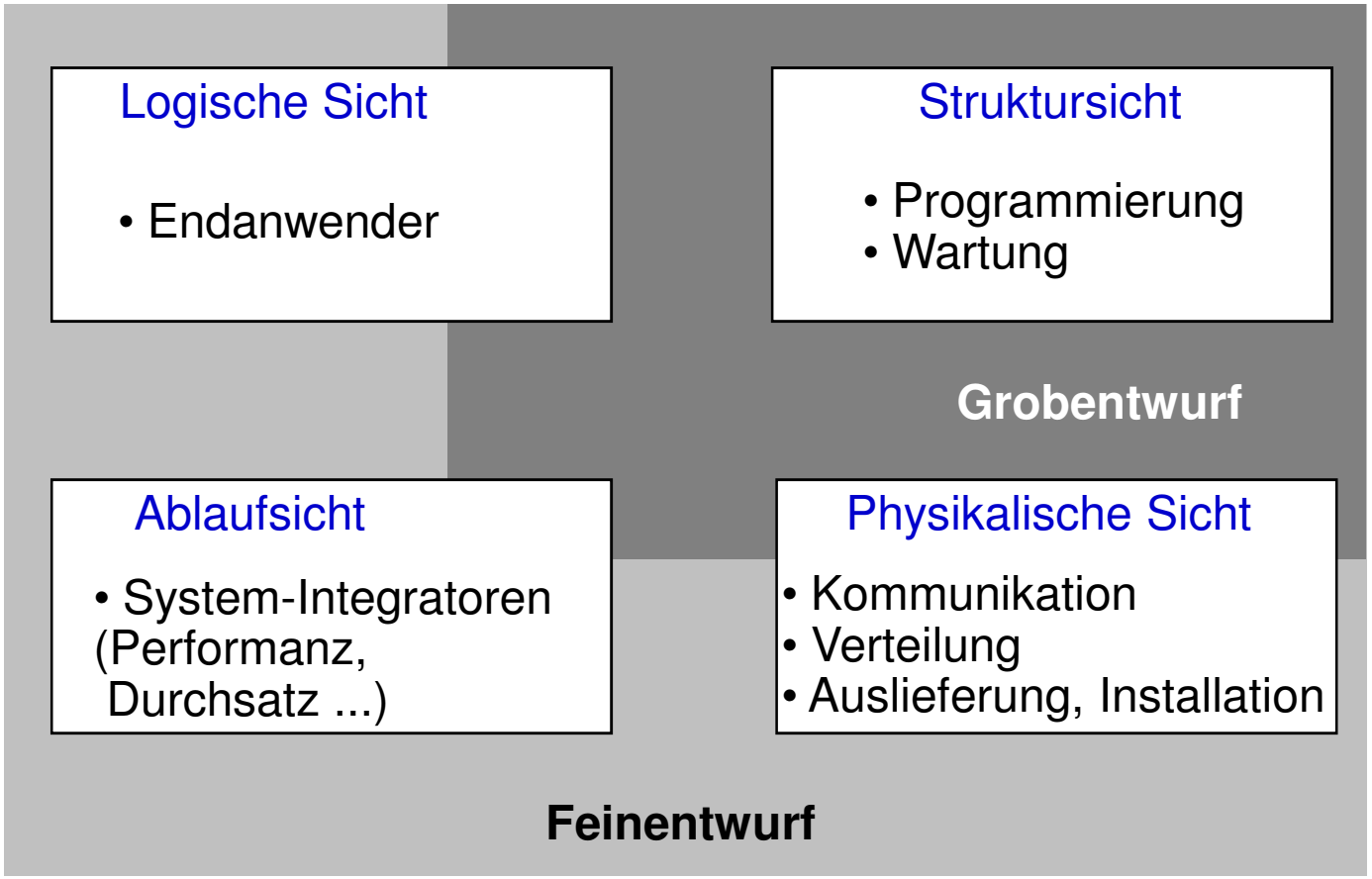


Philippe Kruchten, The 4+1 view model of architecture, IEEE Software, November 1995, 12(6), pp. 42-50

Bestandteile der 4+1 Sichten

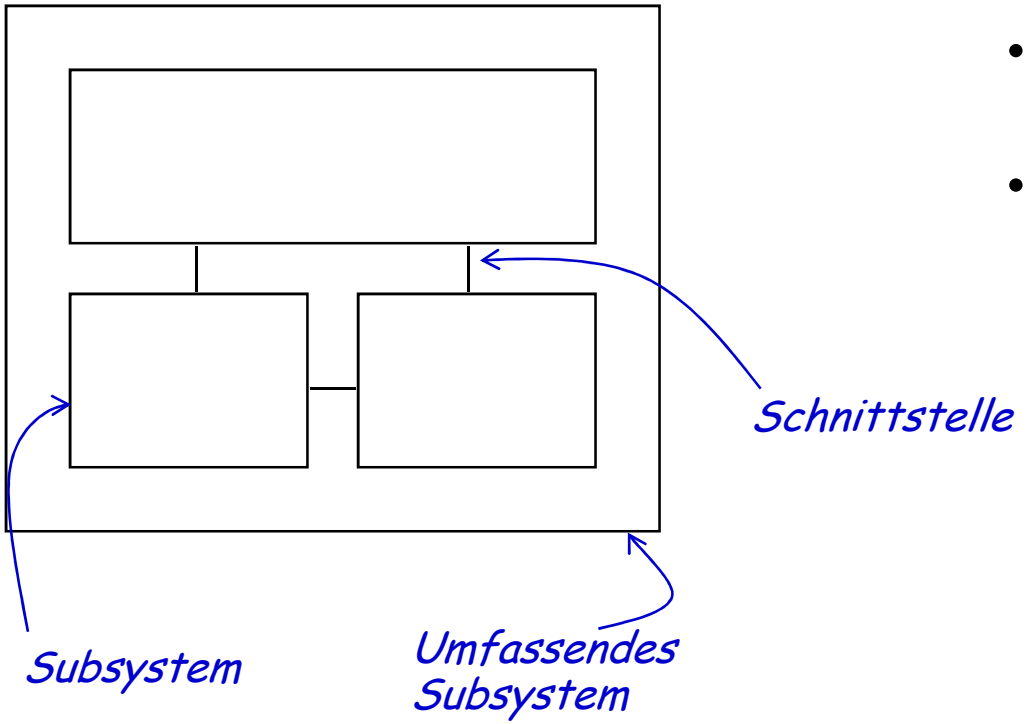


Primäre Zielgruppe/Aufgabe jeder der vier Sichten



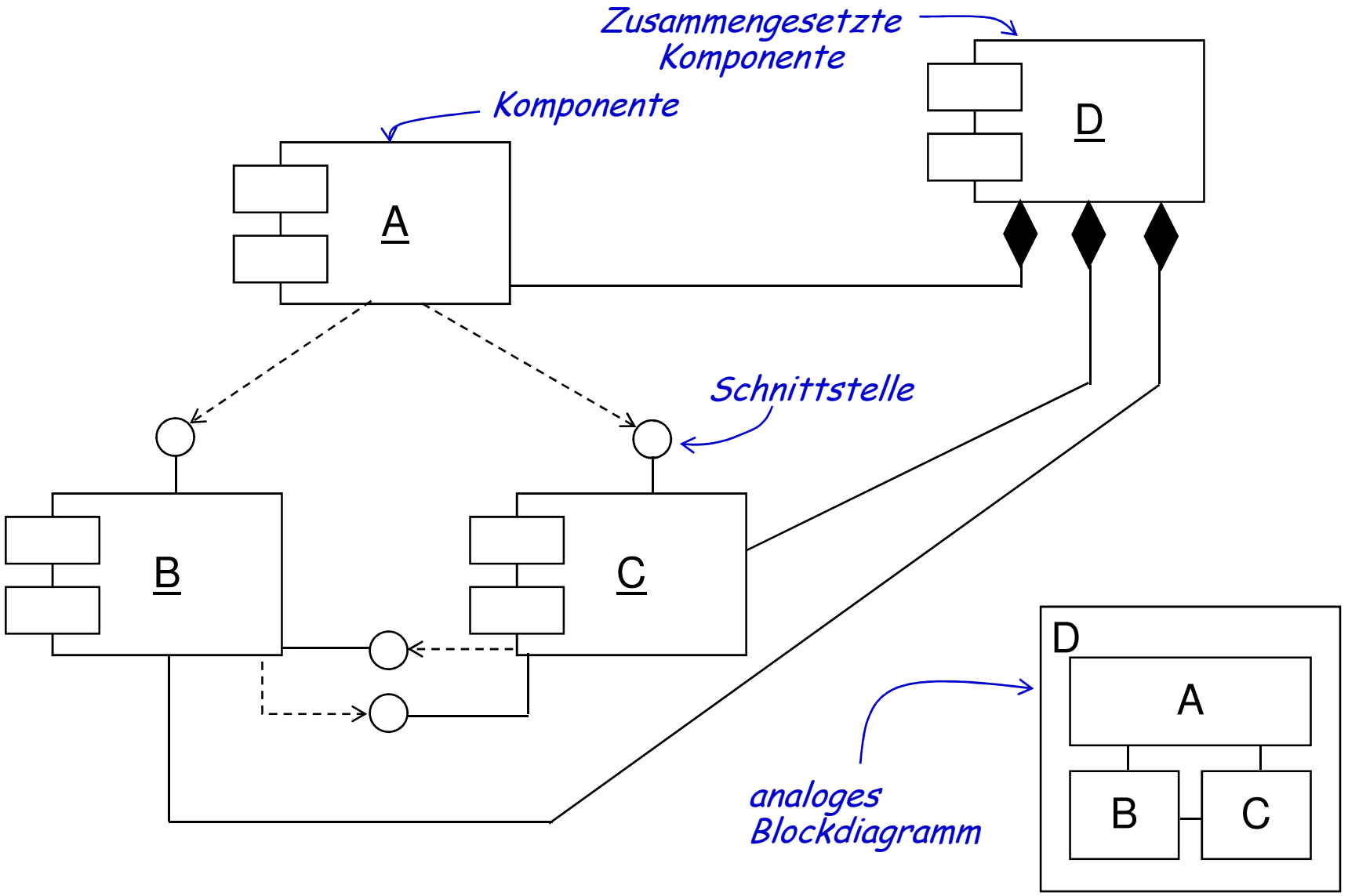
Blockdiagramme

- Blockdiagramme sind kein Bestandteil von UML!
  (Gleichwertige Notation in UML: Implementierungsdiagramm)
- Blockdiagramme sind ein verbreitetes Hilfsmittel zur Skizzierung der logischen **Struktur** einer Systemarchitektur.



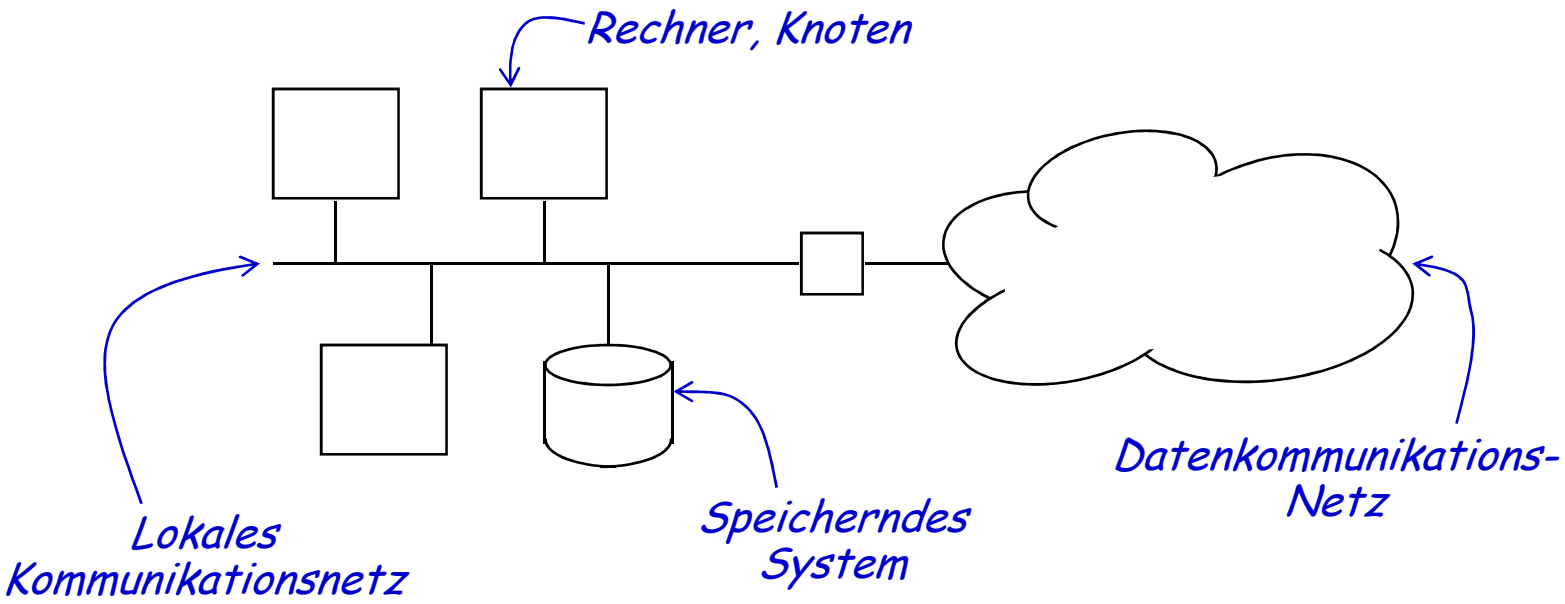
- Subsystem umfasst Objekte bestimmter Klassen
- Schnittstelle ist klar definiert (Aufrufschnittstelle, Kommunikationsprotokoll)

UML: Implementierungsdiagramm



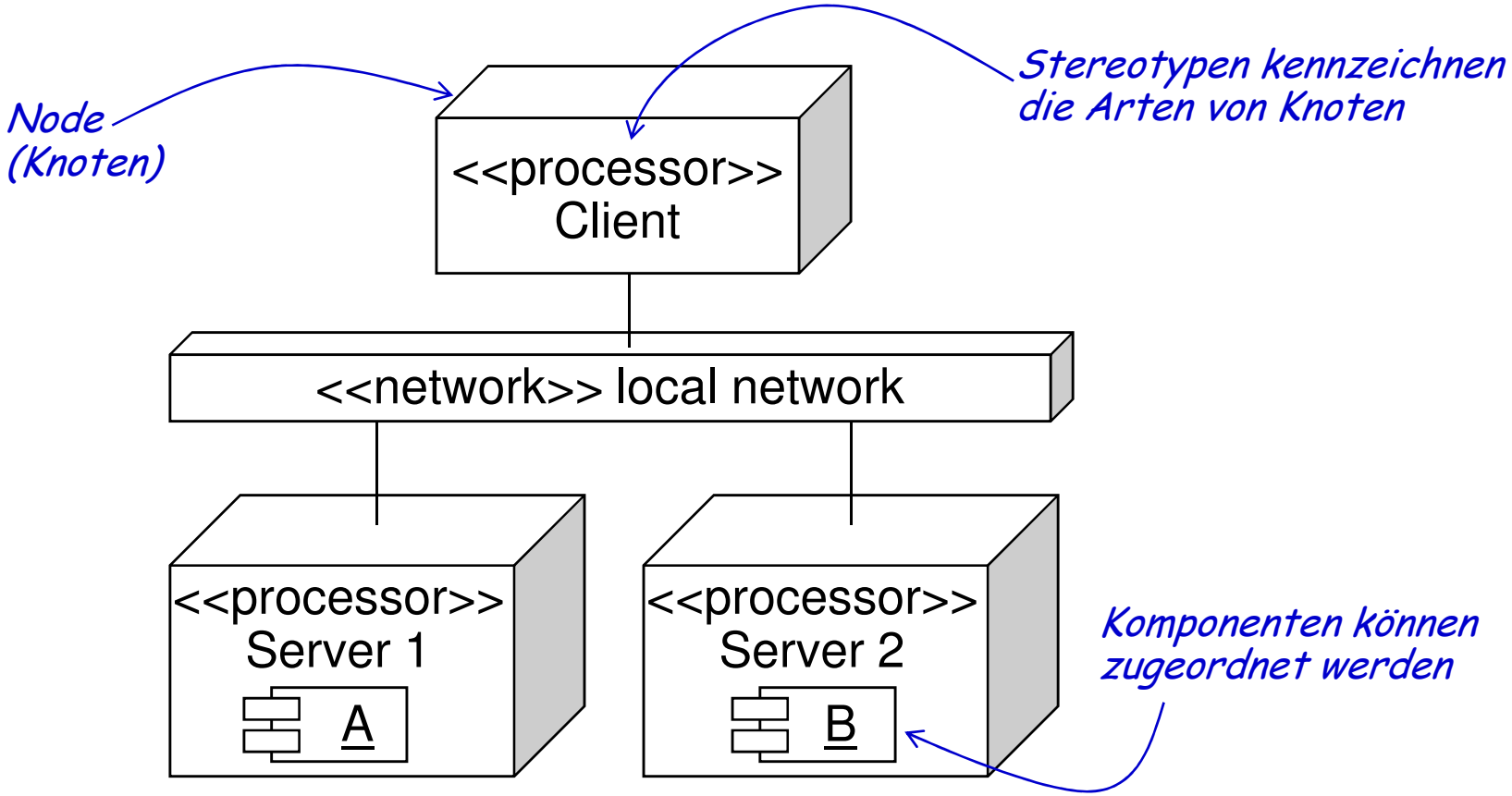
Konfigurationsdiagramme

- Konfigurationsdiagramme sind (noch) nicht Bestandteil von UML!
- Konfigurationsdiagramme sind das meistverbreitete Hilfsmittel zur Beschreibung der physikalischen Verteilung von System-Komponenten.

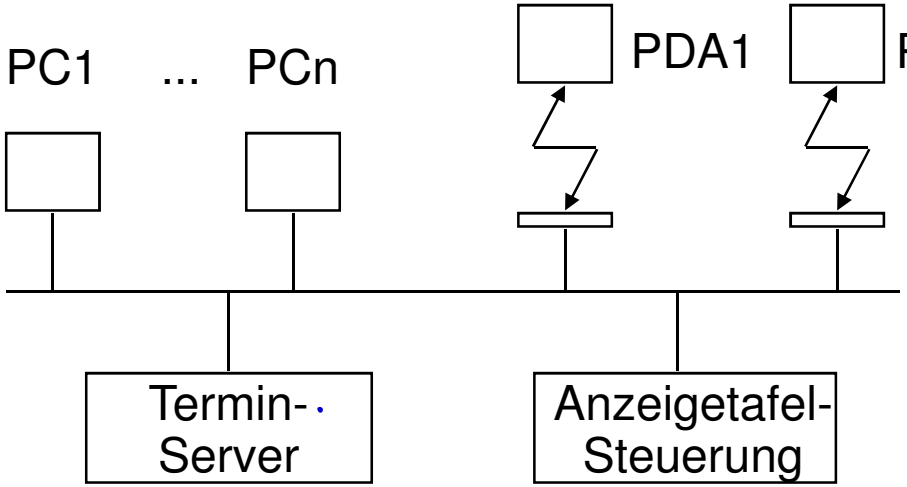


UML: Verteilungsdiagramm

- engl.: *deployment diagram*
- zeigt die physische Verteilung von Systemen

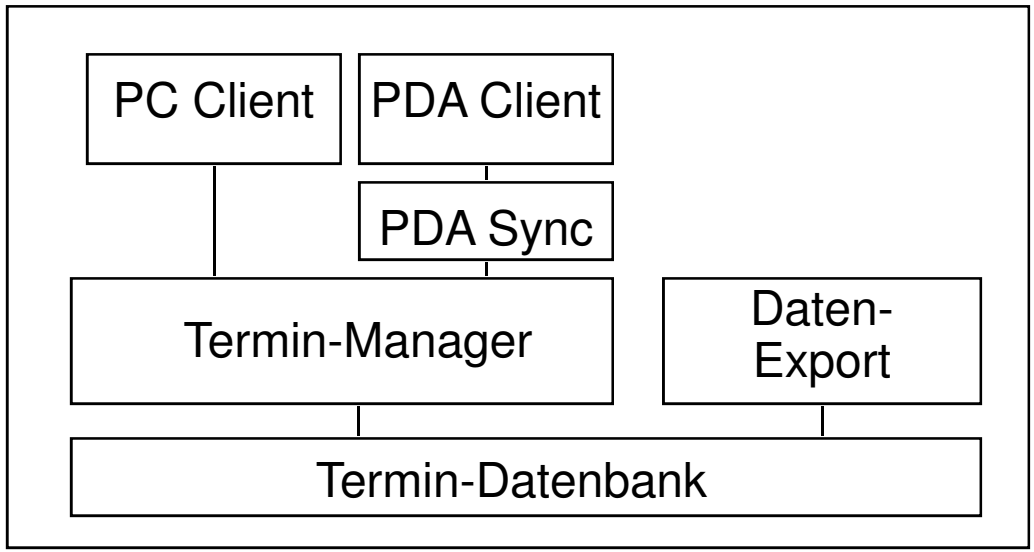


Beispiel Terminverwaltung



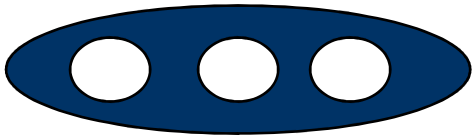
Physikalische Konfiguration

Blockdiagramm

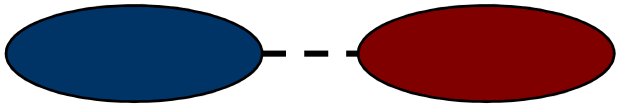


Kriterien für guten Entwurf

- Wie bereits diskutiert ist auf Kohäsion und Kopplung zu achten:



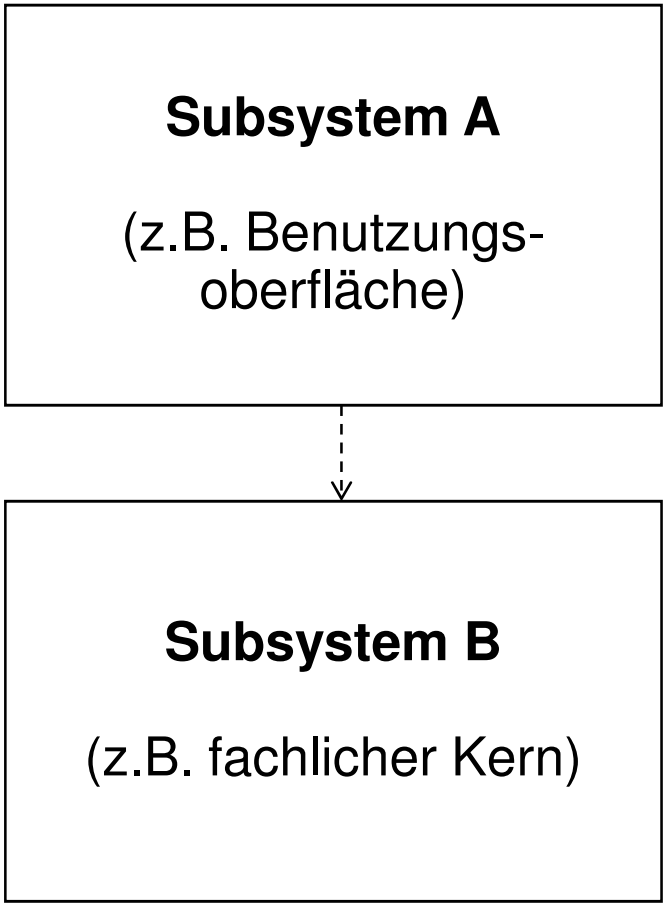
- Hohe Kohäsion:
  - Kohäsion = "Zusammenhalt"
  - Die Dinge sollen in Struktureinheiten zusammengefasst werden, die inhaltlich zusammengehören.



- Niedrige Kopplung:
  - Kopplung = Abhängigkeiten
  - Einzelne Struktureinheiten sollen möglichst unabhängig voneinander sein.

- Daneben allgemeine Eigenschaften, z.B.:
  Korrektheit, Anpassbarkeit, Verständlichkeit, Ressourcenschonung

Hohe Kohäsion + Niedrige Kopplung bei Subsystemen



- **Hohe Kohäsion:**
  Subsystem B darf keine Information und Funktionalität enthalten, die zum Zuständigkeitsbereich von A gehört und umgekehrt.
- **Niedrige Kopplung:**
  Es muss möglich sein, Subsystem A weitgehend auszutauschen oder zu verändern, ohne Subsystem B zu verändern.

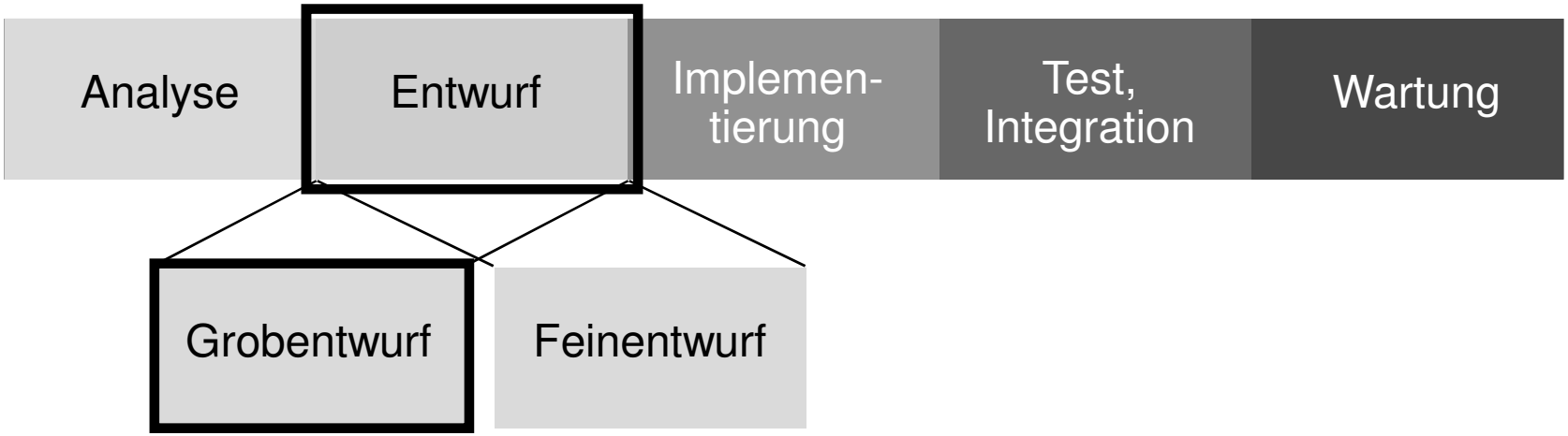
  Änderungen von Subsystem B sollten nur möglichst einfache Änderungen in Subsystem A nach sich ziehen.
- *Beispiele zur konkreten technischen Realisierung siehe später (MVC-Architektur, Entwurfsmuster)*

Qualitätssicherung mittels Szenarien

- Szenarien (für Anwendungsfälle) sind von zentraler Bedeutung:
  - Integration der verschiedenen Sichten
  - Kriterium für Architekturbewertung (Auswahl alternativer Muster)
  - Qualitätssicherung (Review)
- Bewertung für Softwarearchitekturen:
  - Architektur(en) festlegen
    - Im Architekturentwurf: Alternativen
    - Bei der abschließenden Qualitätssicherung: gewählte Architektur
  - Szenarien durchspielen
    - „Direkte Szenarien“: Auf der Architektur gut realisierbar
    - „Indirekte Szenarien“: Nur nach Architekturerweiterung realisierbar
  - Architekturen bewerten nach:
    - Anzahl der direkten Szenarien
    - Aufwand zur Modifikation für indirekte Szenarien
    - Abschätzung der Effizienz

6. Software- & Systementwurf

6.3. Architekturmuster



Prof. Dr. Bernhard Rumpe
Software Engineering
RWTH Aachen

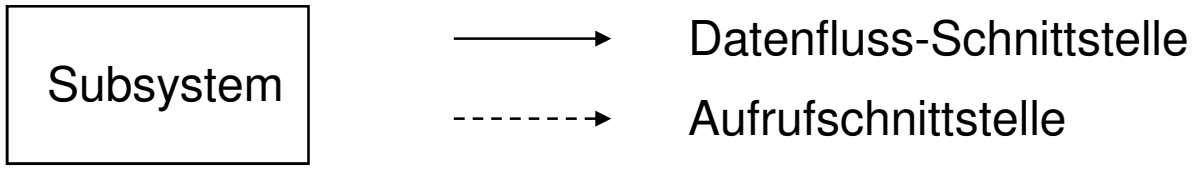
http://www.se-rwth.de/

Literatur:

- Literatur:
- Gamma/Helm/Johnson/Vlissides: Design Patterns, Addison-Wesley 1994 (= „Gang of Four“, „GoF“)
- Buschmann/Meunier/Rohnert/Sommerlad/ Stal: A System of Patterns, Wiley 1996

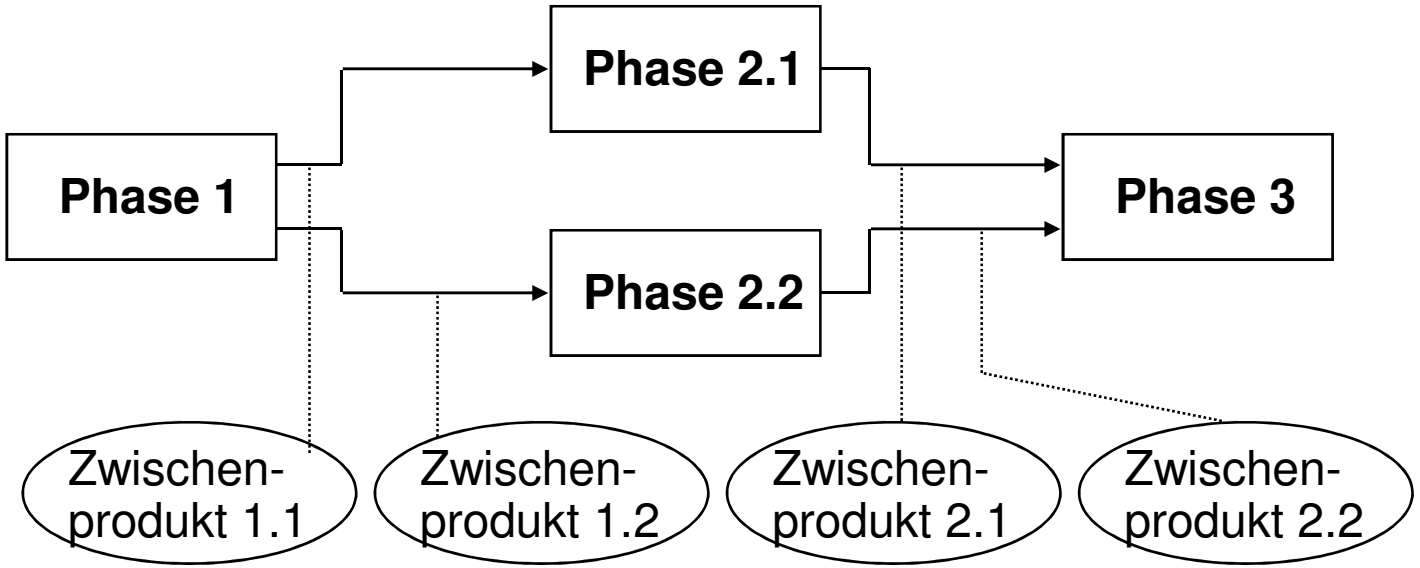
Architekturmuster für die Struktursicht

- Struktursicht der Architektur:
  - Zerlegung in Subsysteme eigenständiger Funktionalität
  - Keine Aussage über physikalische Verteilung
  - Darstellung meist durch Blockdiagramme:



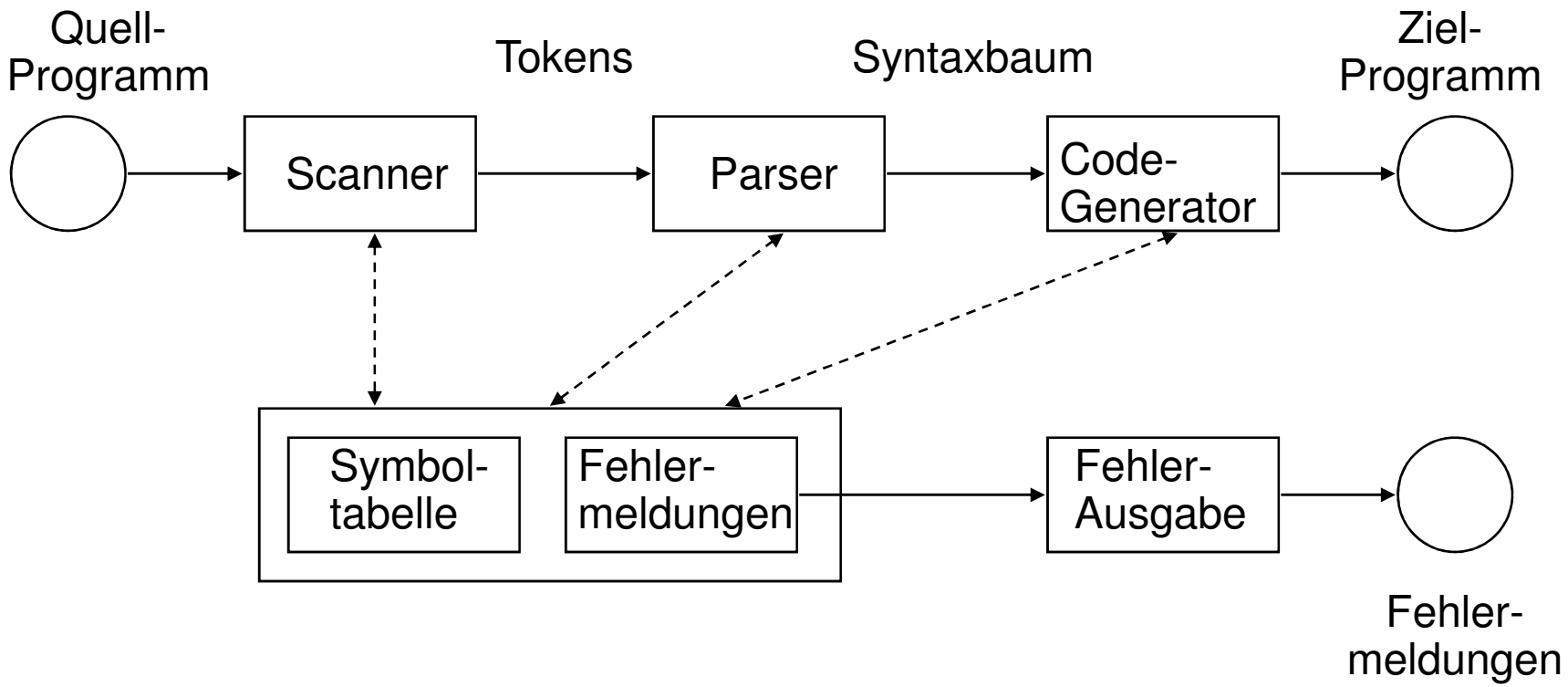
- Muster (Architekturmuster, Architekturstile):
  - Kette (Chain)
  - Schichten
  - Interpreter

Architekturmuster „Pipes & Filters“ (oder „Kette“)



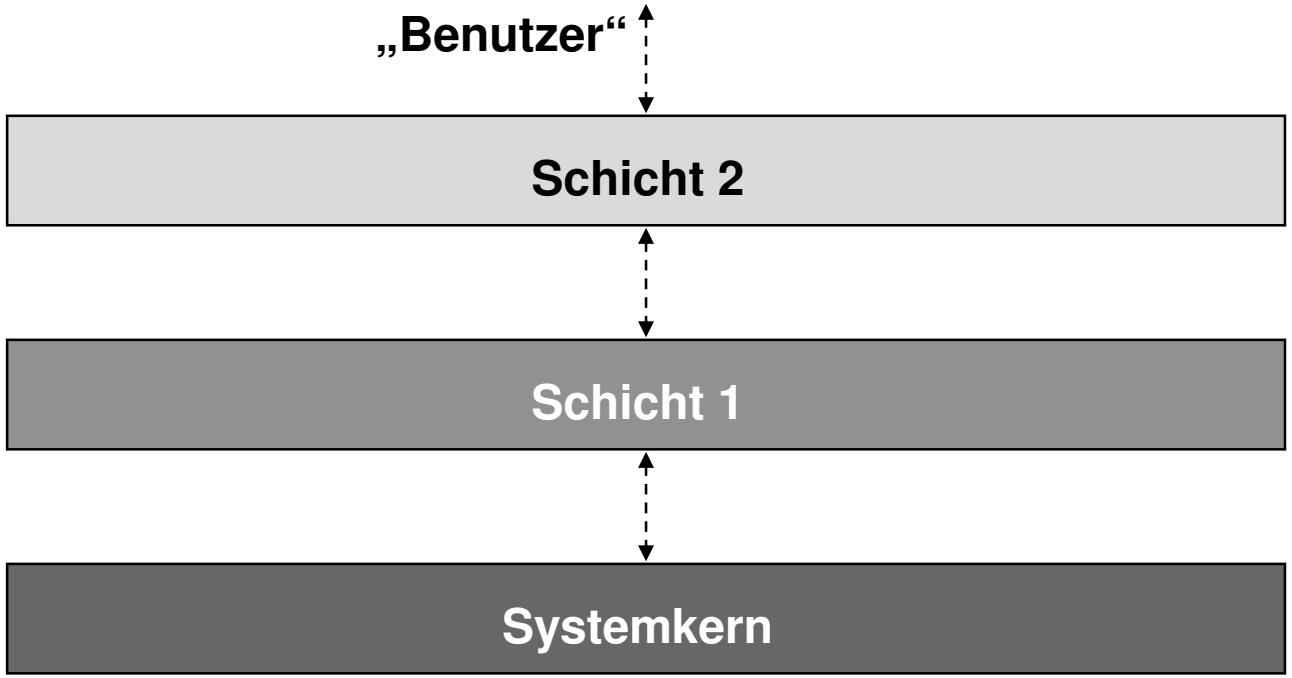
- Inkrementelle oder phasenweise Verarbeitung
- Beispiele:
  - UNIX pipes
  - Batch-sequentielle Systeme
  - Compiler-Grundstruktur

Beispiel: Compiler-Architektur



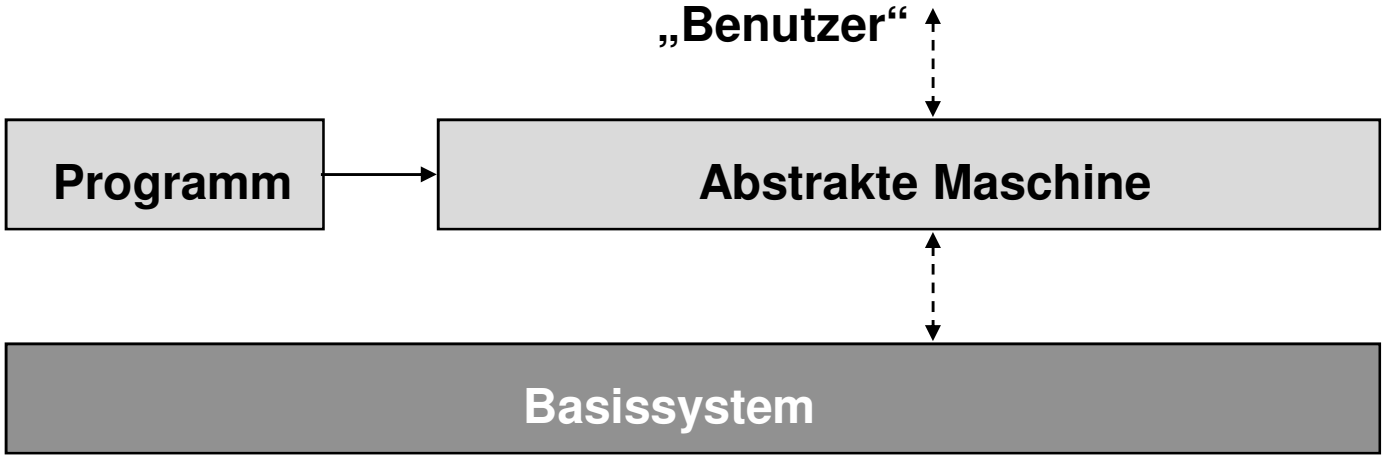
- Kombination von Ketten

Architekturmuster "Schichten"



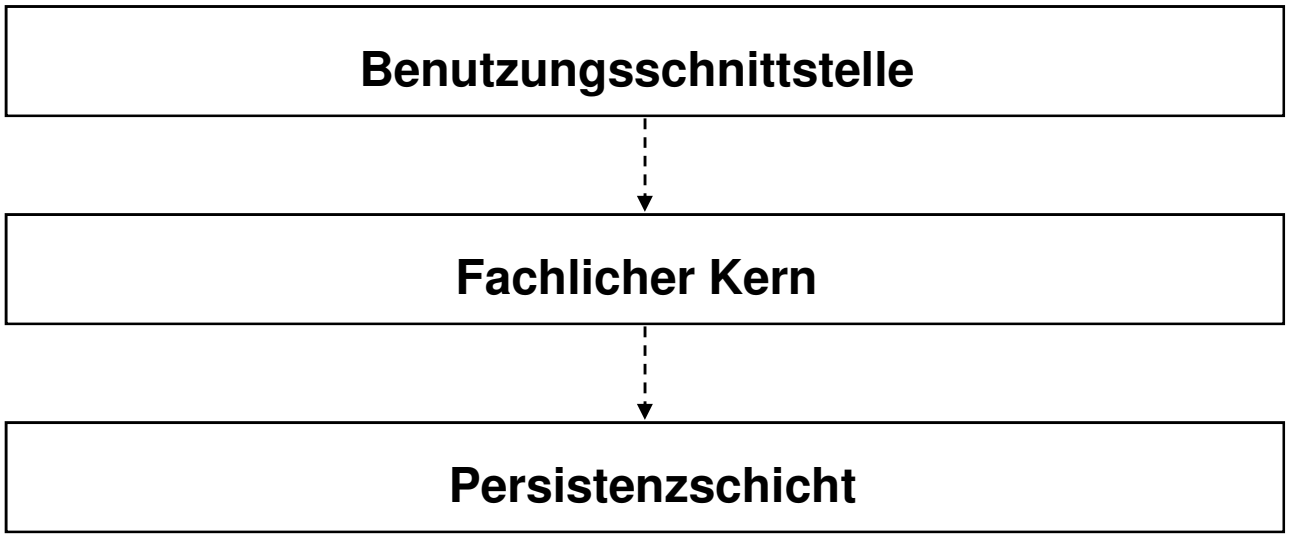
- Jede Schicht bietet Dienste (nach oben) und nutzt Dienste (von unten)
- Beispiele:
  - Kommunikationsprotokolle
  - Datenbanksysteme, Betriebssysteme

Architekturmuster "Interpreter"



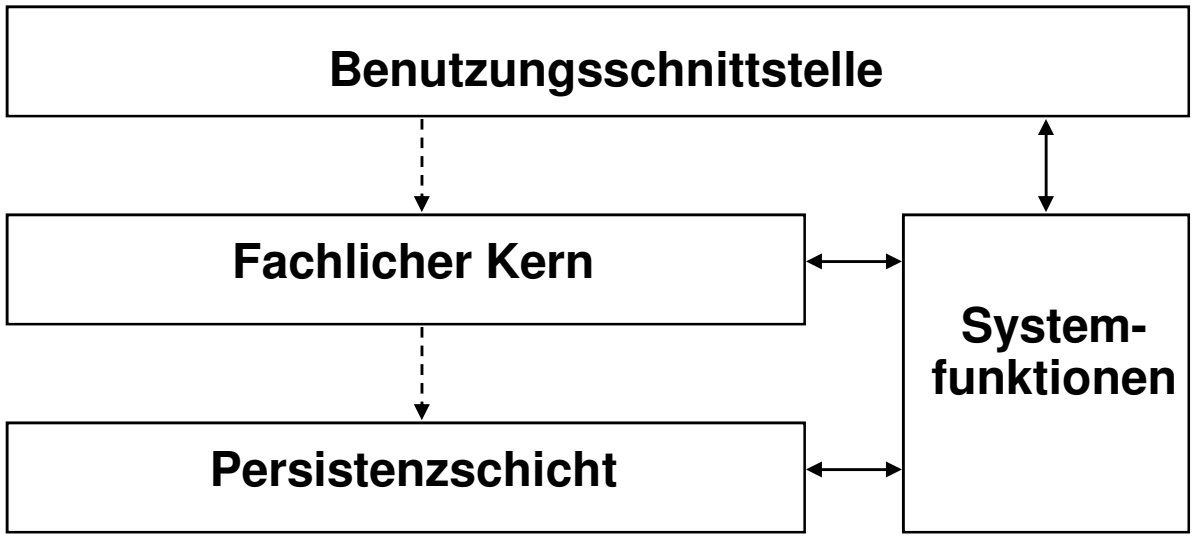
- Schichtenarchitektur mit Parametrisierung
- Beispiele:
  - Portable Sprachimplementierung (z.B. Java Virtual Machine)
  - Emulation von Systemarchitekturen (z.B. Soft Windows)

Beispiel: 3-Schichten-Referenzarchitektur



- Entwurfsregeln:
  - Benutzungsschnittstelle greift ***nie*** direkt auf Datenhaltung zu.
  - Persistenzschicht verkapselt Zugriff auf Datenhaltung, ist aber nicht identisch mit dem Mechanismus der Datenhaltung (z.B. Datenbank).
  - Fachlicher Kern basiert auf dem Analyse-Modell
- Erlaubt das Aufsetzen von interaktiven, batch, etc. Benutzerschnittstellen und den Austausch von Datenbanken

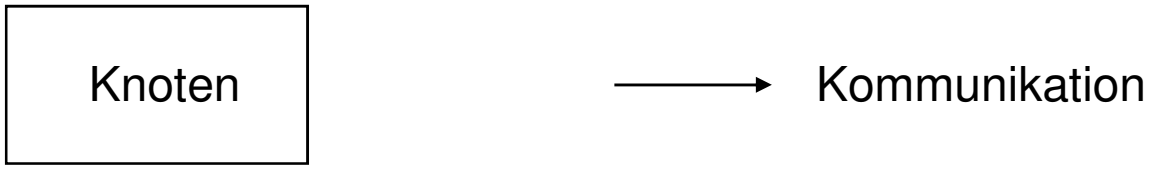
Variante: 3-Schichten-Referenzarchitektur



- Beispiele für Systemfunktionen:
  - Verkapselung von plattformspezifischen Funktionen
  - Schnittstellen zu Fremdsystemen

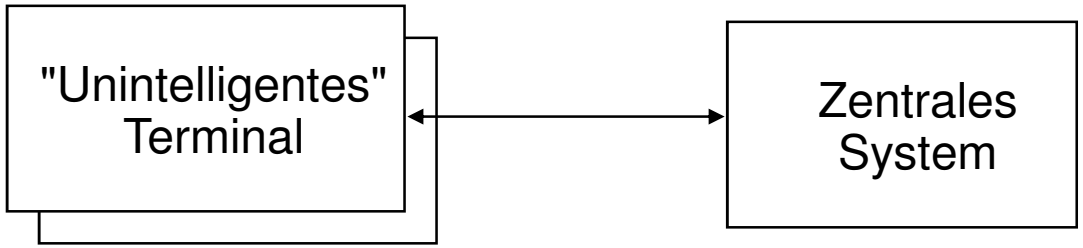
Architekturmuster für die physikalische Sicht

- Physikalische Sicht der Architektur:
  - Aufteilung der Funktionalität auf Knoten (Rechner) eines Netzes
  - Darstellung meist durch Konfigurationsdiagramme:



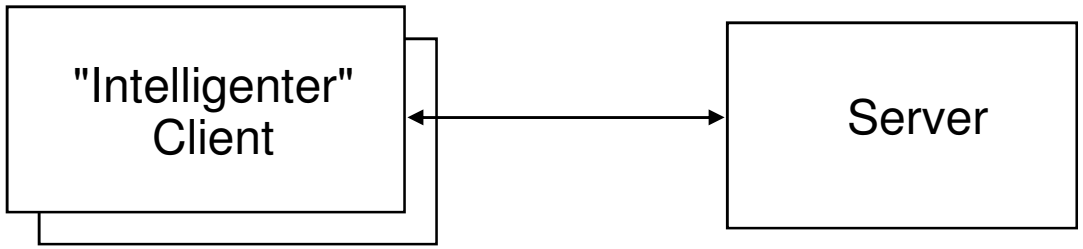
- Muster (Verteilungsmuster):
  - Zentrales System
  - Client/Server:
    - Two-Tier (Thin-Client, Fat-Client)
    - Three-Tier (GUI; Applikationskern, Datenhaltung)
  - Föderation

Verteilungsmuster "Zentrales System"



- Beispiele:
  - Klassische Großrechner-("Mainframe"-)Anwendungen
  - Noch einfachere Variante:
    Lokale PC-Anwendungen (identifizieren Zentrale und Terminal)

Verteilungsmuster "Client/Server"

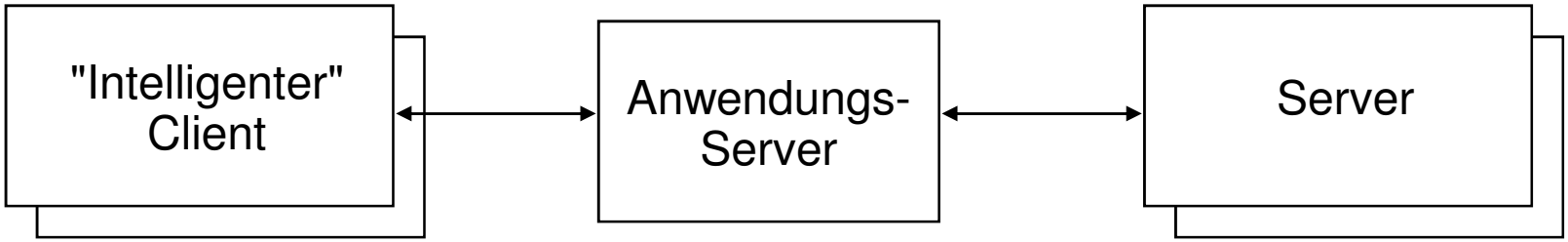


- Sogenannte "Two-Tier" Client/server-Architektur
- Andere Namen:
  - "Front-end" für "Client", "Back-end" für "Server"
- Client:
  - Benutzungsschnittstelle
  - Einbindung in Geschäftsprozesse
  - Entkoppelt von Netztechnologie und Datenhaltung
- Server:
  - Datenhaltung, evtl. Fachlogik

"Thin-Client" und "Fat-Client"

- Thin-Client:
  - Nur die Benutzungsschnittstelle auf dem Client-System
  - Ähnlich zu Zentralem System, aber oft Download-Mechanismen
  - Anwendungen:
    - "Screen-Scraping"
      (Umsetzung traditioneller Benutzungsschnittstellen in moderne Technologie)
- Fat-Client:
  - Teile der Fachlogik (oder gesamte Fachlogik) auf dem Client-System
  - Hauptfunktion des Servers: Datenhaltung
  - Entlastung des Servers
  - Zusätzliche Anforderungen an Clients (z.B. Installation von Software)

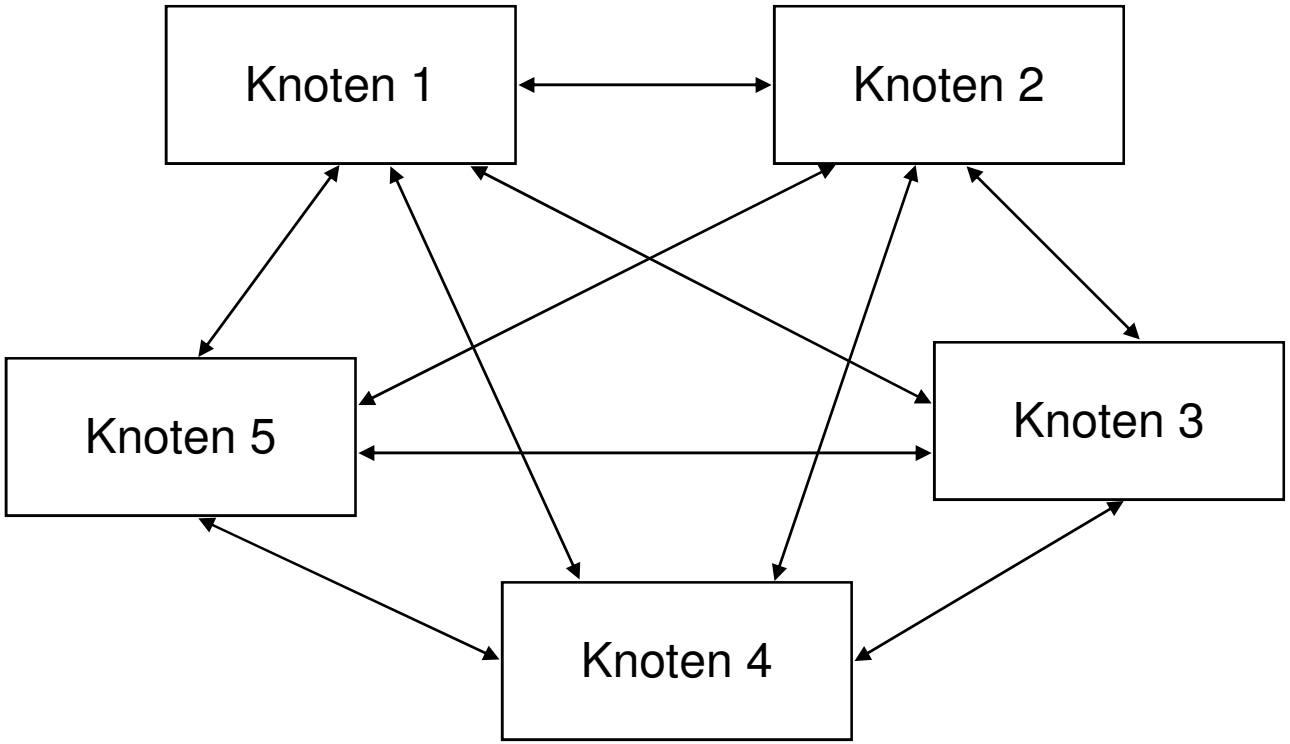
Verteilungsmuster "Three-Tier Client/Server"



- Client:
  - Benutzungsschnittstelle
  - evtl. Fachlogik
- Anwendungsserver:
  - evtl. Fachlogik
  - Verteilung von Anfragen auf verschiedene Server
- Server:
  - Datenhaltung, Rechenleistung etc.
- Kommunikation unter Servern meist breitbandig.

- Heute üblich!

Verteilungsmuster "Föderation"



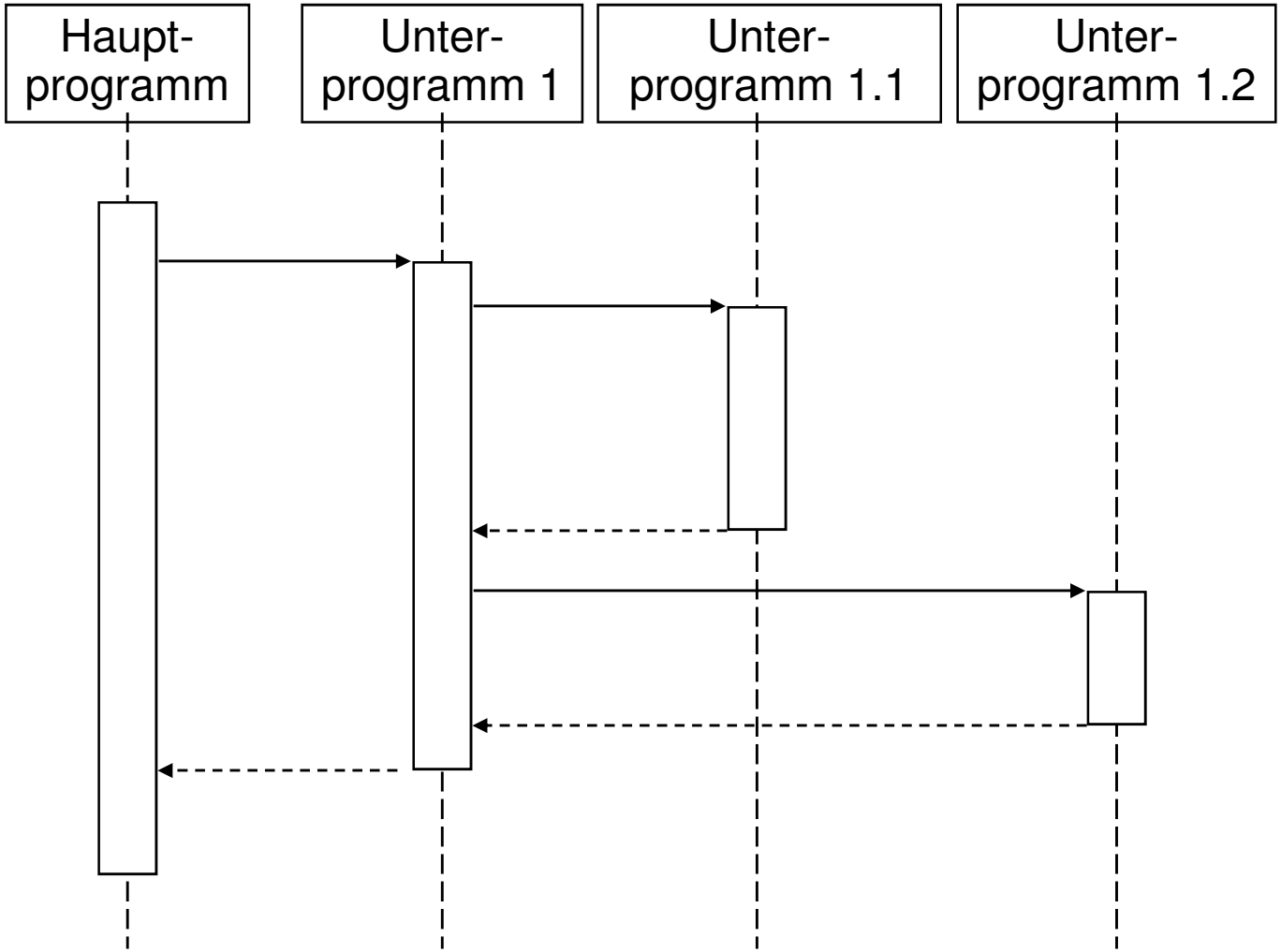
- Gleichberechtigte Partner *(peer-to-peer)*
- Unabhängigkeit von der Lokation und Plattform von Funktionen
- Verteilte kommunizierende Objekte

Architekturmuster der Ablaufsicht

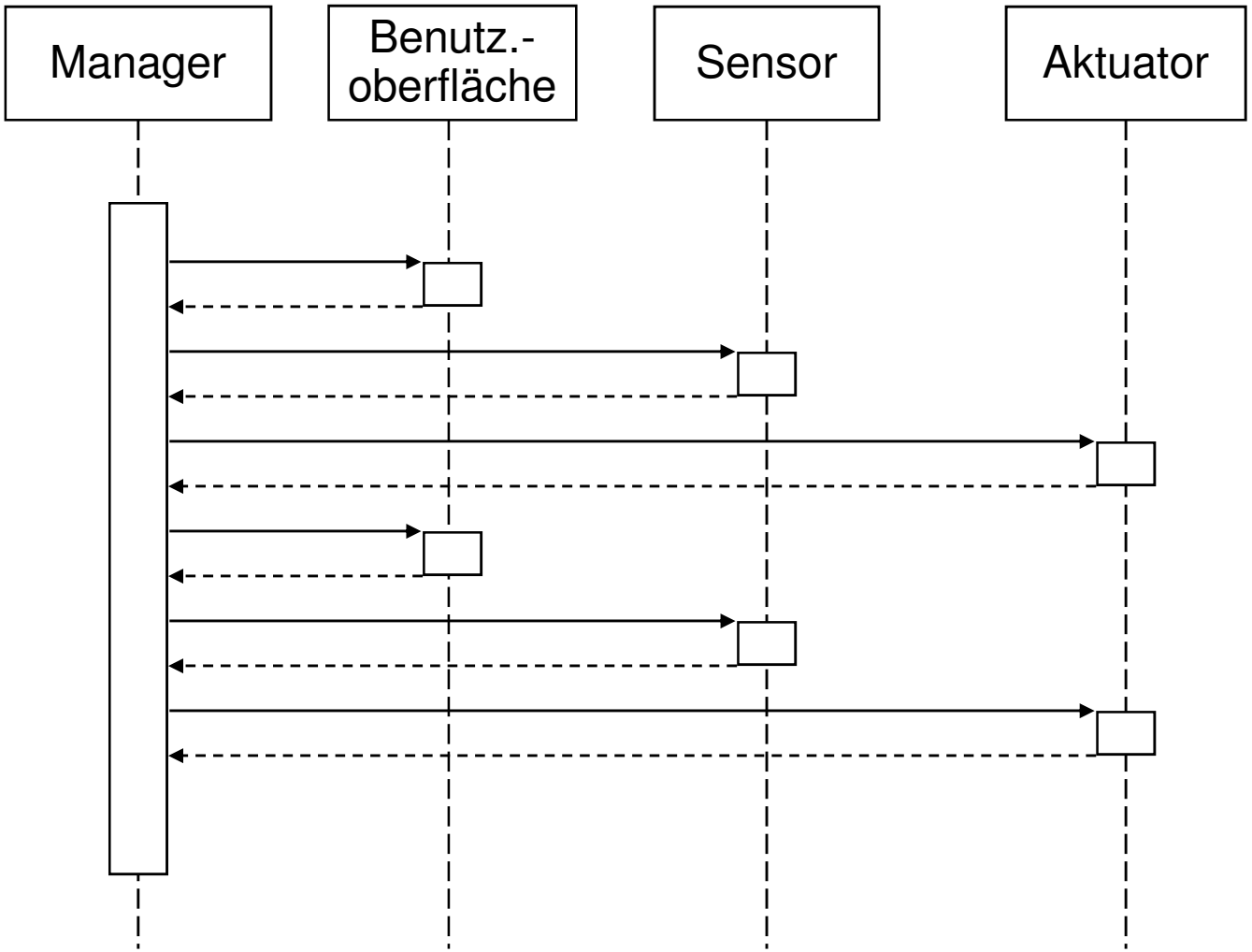
- Ablaufsicht der Architektur:
  - Definition nebenläufiger Systemeinheiten (z.B. Prozesse)
  - Steuerung der Abfolge von Einzelfunktionen
  - Synchronisation und Koordination
  - Reaktion auf externe Ereignisse
- Darstellung z.B. durch Sequenzdiagramme

- Muster (Steuerungsmuster):
  - Zentrale Steuerung
    - Call-Return
    - Master-Slave
  - Ereignis-Steuerung
    - Selective Broadcast
    - Interrupt

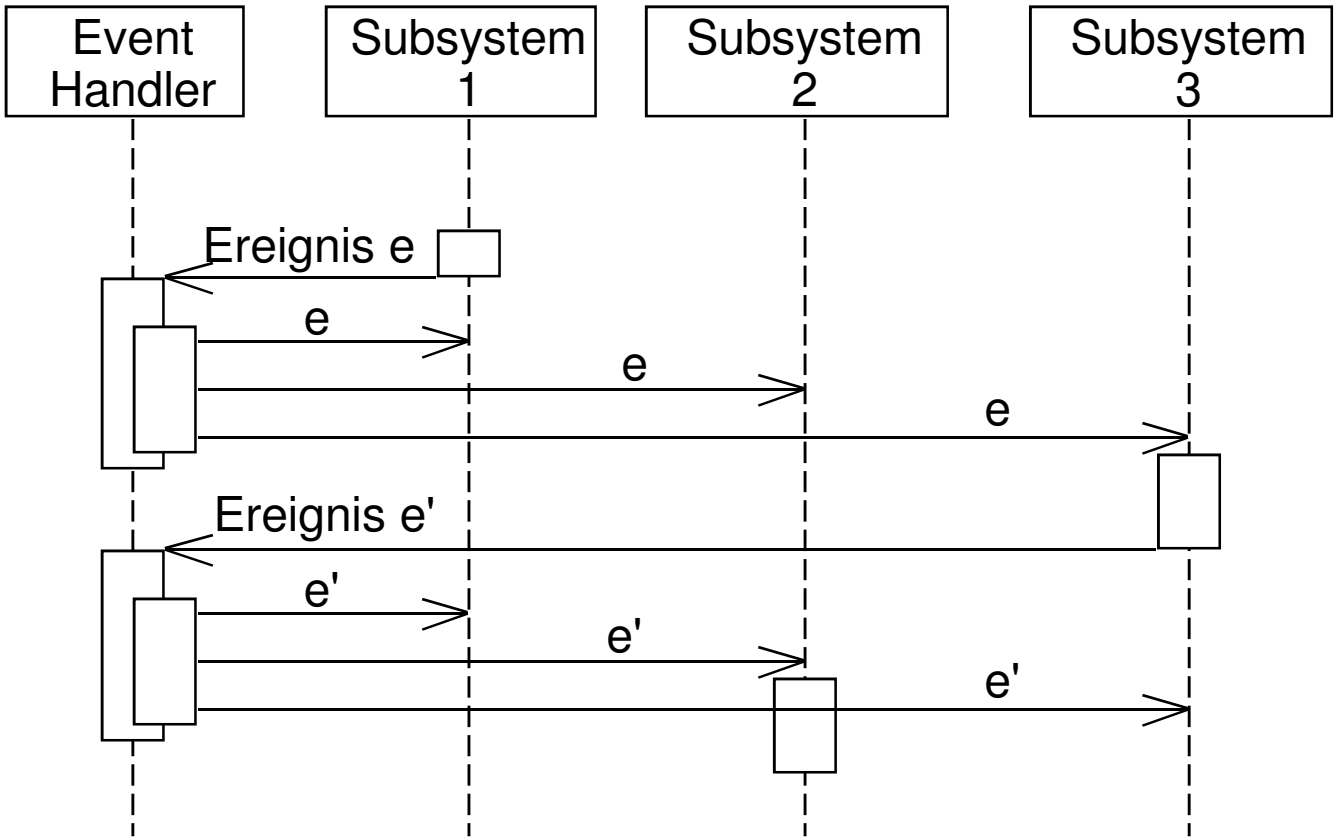
Steuerungsmuster "Call-Return"



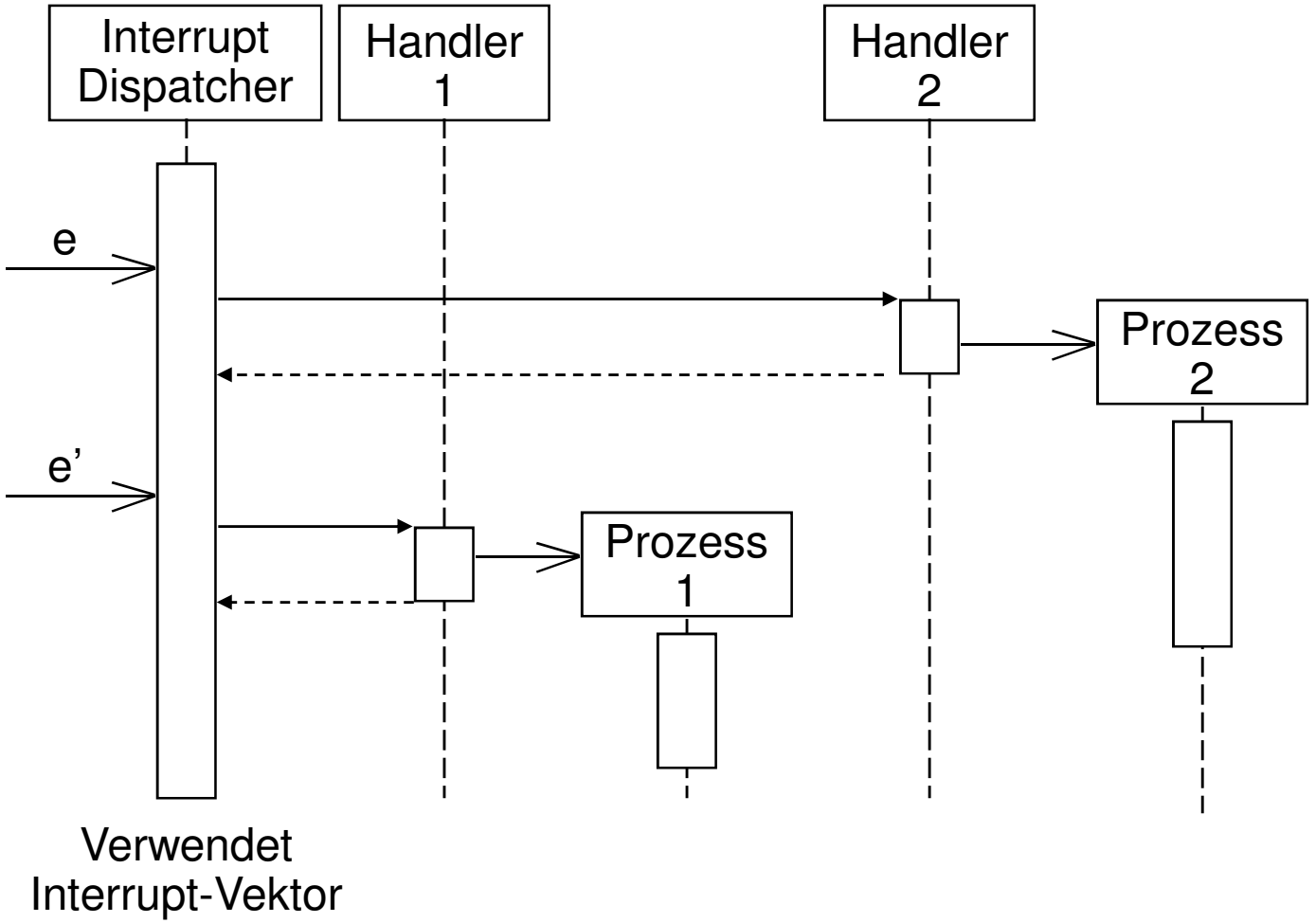
Steuerungsmuster "Master-Slave"



Steuerungsmuster "Selective Broadcast"



Steuerungsmuster "Interrupt"



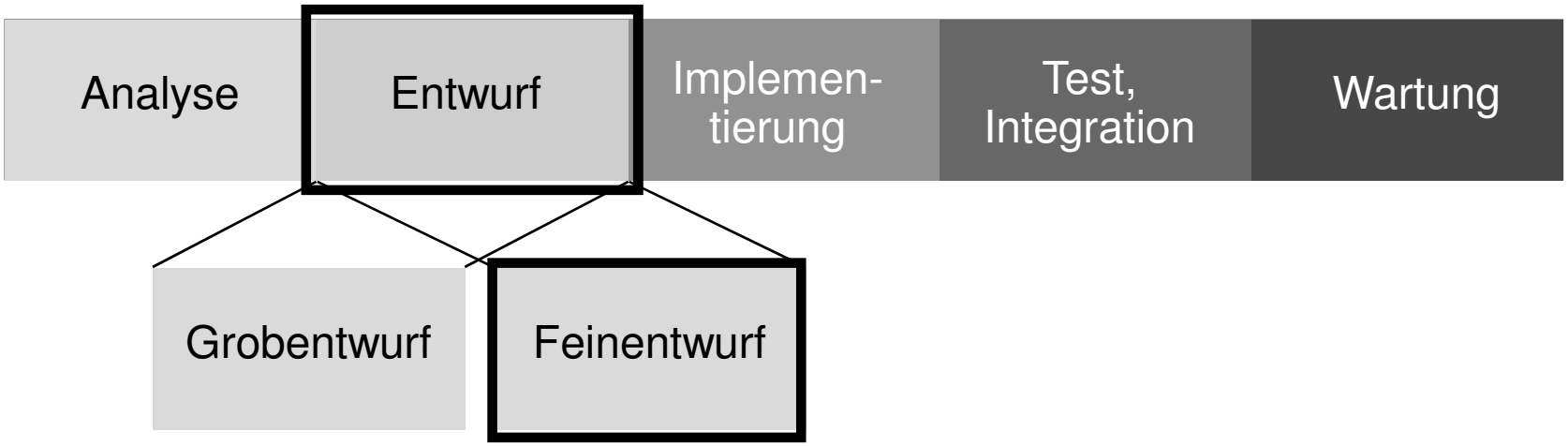
Zusammenfassung
4.3 Architekturmuster

- Architekturmuster beschreiben erprobte Strukturierungsformen für die Architektur eines Systems

- Architekturmuster beschreiben:
  - Struktur
  - physikalische Verteilung
  - Zuordnung von Prozessen auf Prozessoren
  - Kommunikationsformen und –protokolle

- Schichtenbildung ist ein mächtiges Strukturierungsmittel

6. Software- & Systementwurf
6.4. Objektorientierter Feinentwurf mit Klassendiagrammen



Prof. Dr. Bernhard Rumpe
Software Engineering
RWTH Aachen

http://www.se-rwth.de/

Objektorientierter Feinentwurf

- Ausgangspunkt:
  - Grobdefinition der Architektur:
    - Zerlegung in Subsysteme
      (evtl. unter Verwendung von Standardarchitekturen)
    - Verteilungskonzept
    - Ablaufmodell
- Ergebnis:
  - OO-Modell für jedes Subsystem der Architektur
  - OO-Modell für unterstützende Subsysteme
    - unter Berücksichtigung gewählter Technologien
  - Spezifikationen der Klassen
  - Spezifikationen von externen Schnittstellen

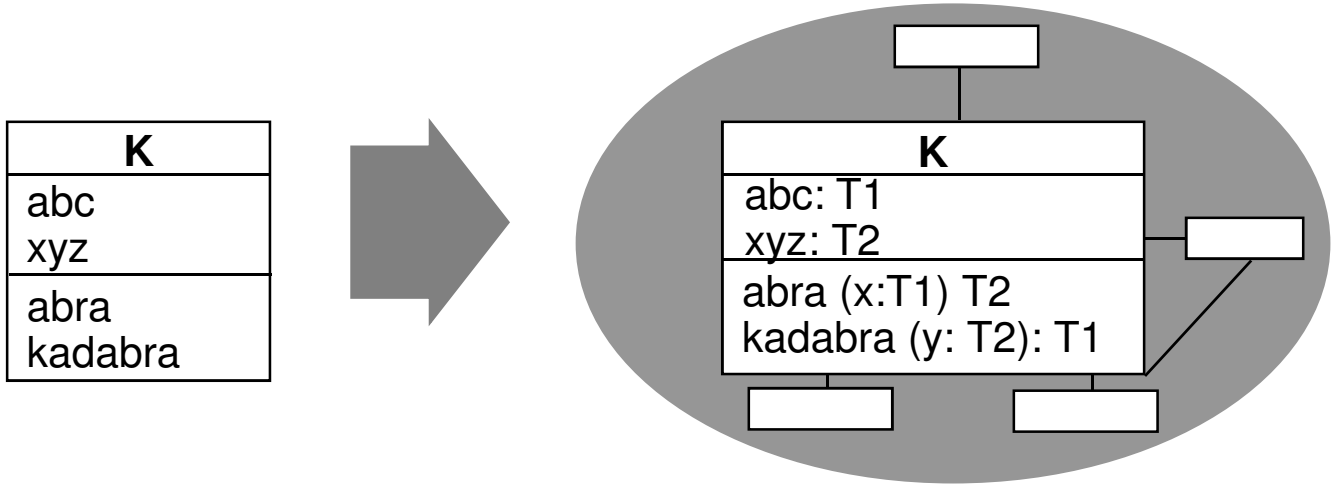
Verfeinerung des Analysemodells

- Fachlicher Kern: Mehr Details als im Analysemodell
  - Listen der Attribute und Operationen: vollständig
  - Attribute und Operationen: Datentypen, Sichtbarkeit
  - Operationen: Spezifikation (z.B. Vor- und Nachbedingungen)
  - Assoziationen/Aggregationen:
    Navigationsrichtung, Ordnung, Qualifikation

- Zusätzliche Klassen/Pakete:
  - Einbindung in Infrastruktur, Altsysteme etc.
  - Anpassungs- und Entkopplungsschichten für gewählte Technologien
    (z.B. Datenzugriffsschicht, CORBA-Schnittstellen, XML-Anschluss...)

Verfeinerung des Analysemodells

- Grobskizze:



UML im Entwurf

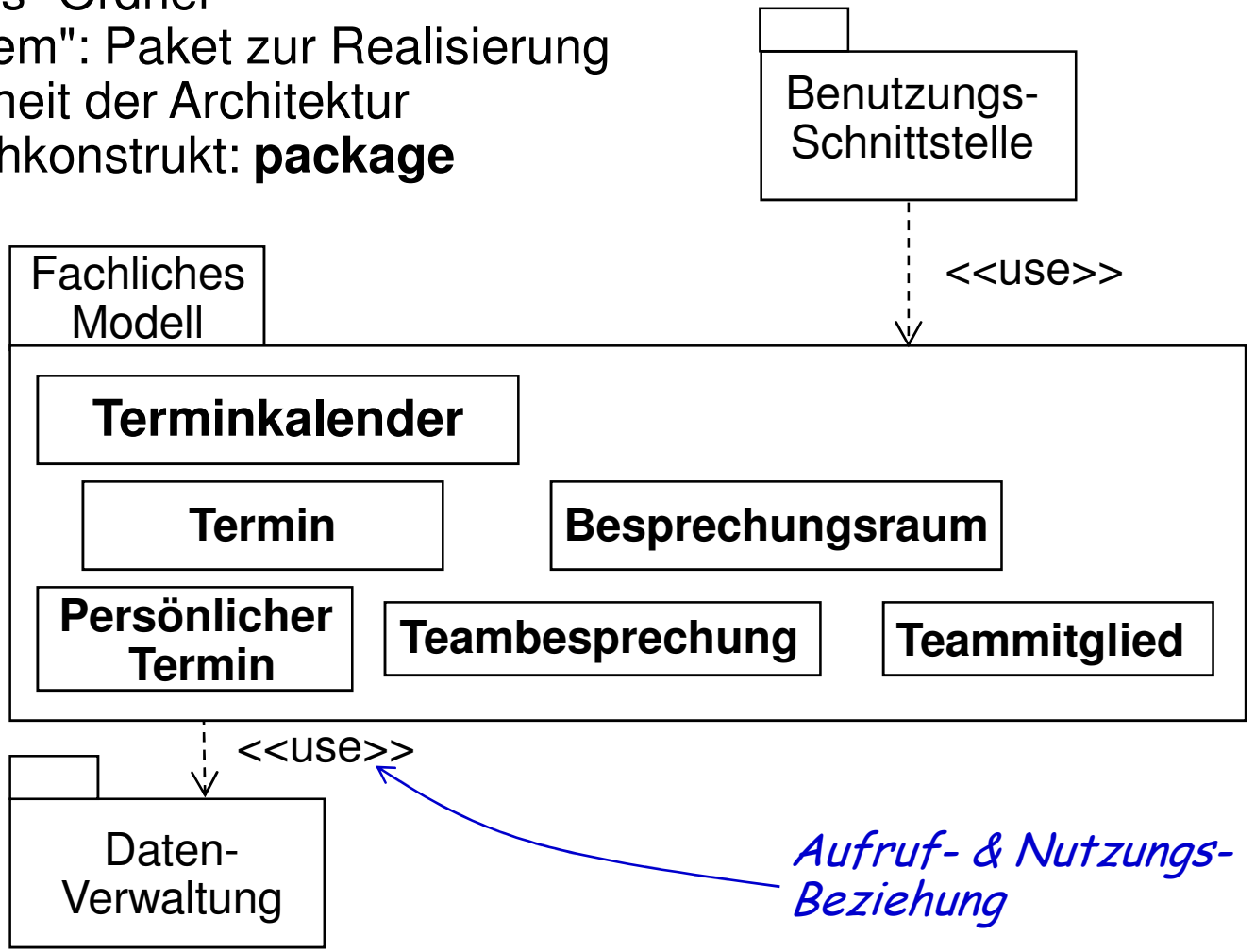
- generell: Analysemodelle werden im Entwurf umgebaut
- Insbesondere Klassendiagramme erhalten dazu eine neue Bedeutung:
  - In der Analyse repräsentieren Klassen meist Einheiten der realen Welt
  - Im Entwurf stellen Klassen einen Teil des Softwaresystems dar
  - Es findet eine Detaillierung und Präzisierung statt
- Statecharts werden (soweit nicht direkt in Einzelspezifikationen von Methoden zerlegt) ebenfalls detailliert
- Andere UML-Diagramme werden im Feinentwurf vor allem als Vorlagen (Aktivitäts-, Sequenzdiagramme, Use Cases) oder zur Strukturierung im Grobentwurf (Komponentendiagramme) eingesetzt, selbst aber nicht detailliert.

UML zum logischen Detailentwurf

Analyse-Modell	**Entwurfs-Modell**
Notation: UML	Notation: UML
Objekte: Fachgegenstände	Objekte: Softwareeinheiten
Klassen: Fachbegriffe	Klassen: Schemata
Vererbung: Begriffsstruktur	Vererbung: Programmableitung
Annahme perfekter Technologie	Erfüllung konkreter Rahmenbedingungen
Funktionale Essenz	Gesamtstruktur des Systems
Völlig projektspezifisch	Ähnlichkeiten zwischen verwandten Projekten
Grobe Strukturskizze	Genaue Strukturdefinition
	Mehr Struktur & mehr Details

Pakete und Subsysteme

- UML:
  - Pakete als "Ordner"
  - "Subsystem": Paket zur Realisierung einer Einheit der Architektur
- Java-Sprachkonstrukt: **package**



Sichtbarkeit

	Sichtbarkeits-Symbol			
UML	+	#	–	
Java	**public**	**protected**	**private**	*(default)*
Sichtbar für:				
Gleiche Klasse	**ja**	**ja**	**ja**	**ja**
Andere Klasse, gleiches Paket	**ja**	**ja** / **nein***	**nein**	**ja**
Andere Klasse, anderes Paket	**ja**	**nein**	**nein**	**nein**
Unterklasse, gleiches Paket	**ja**	**ja**	**nein**	**ja**
Unterklasse, anderes Paket	**ja**	**ja**	**nein**	**nein**

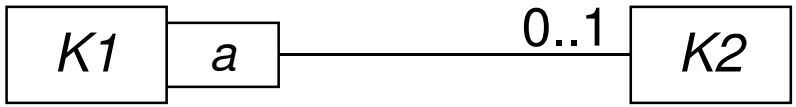
* In UML und C++ “nein”, in Java “ja”.

Qualifizierte Assoziation

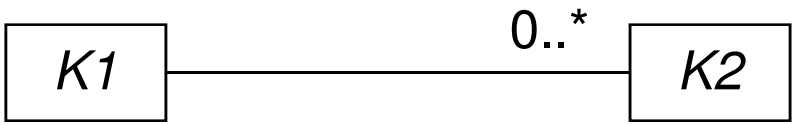
- **Definition:** Eine *Qualifikation (Qualifier)* ist ein Attribut für eine Assoziation zwischen Klassen K1 und K2, durch das die Menge der zu einem K1-Objekt assoziierten K2-Objekte *partitioniert* wird.

  Zweck der Qualifikation ist direkter Zugriff unter Vermeidung von Suche.

- **Notation:**

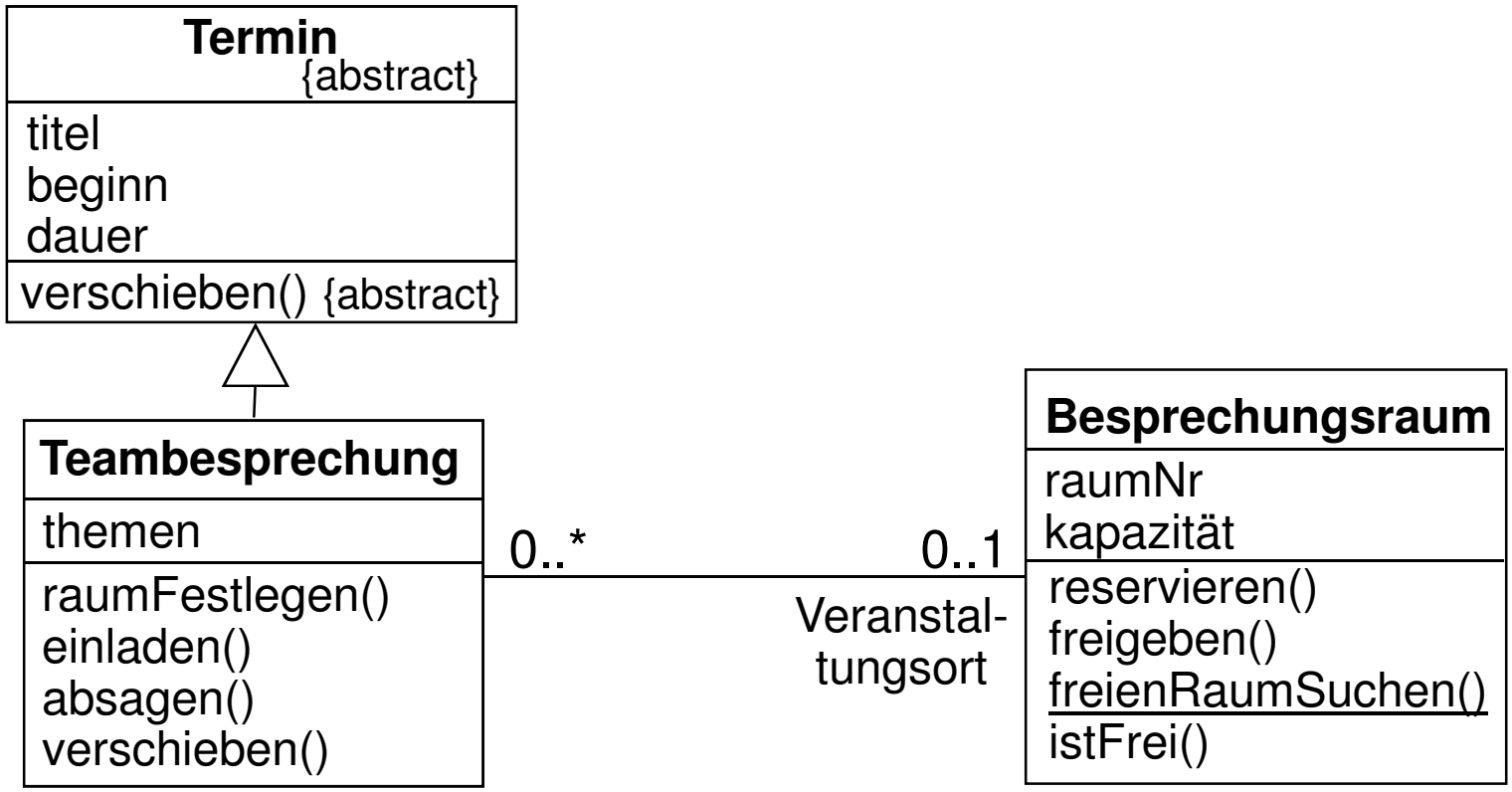


als Detaillierung von:



Bedeutung vor allem im Zusammenhang mit Datenbanken (Indizes), aber auch mit geeigneten Datenstrukturen nach Java abbildbar.

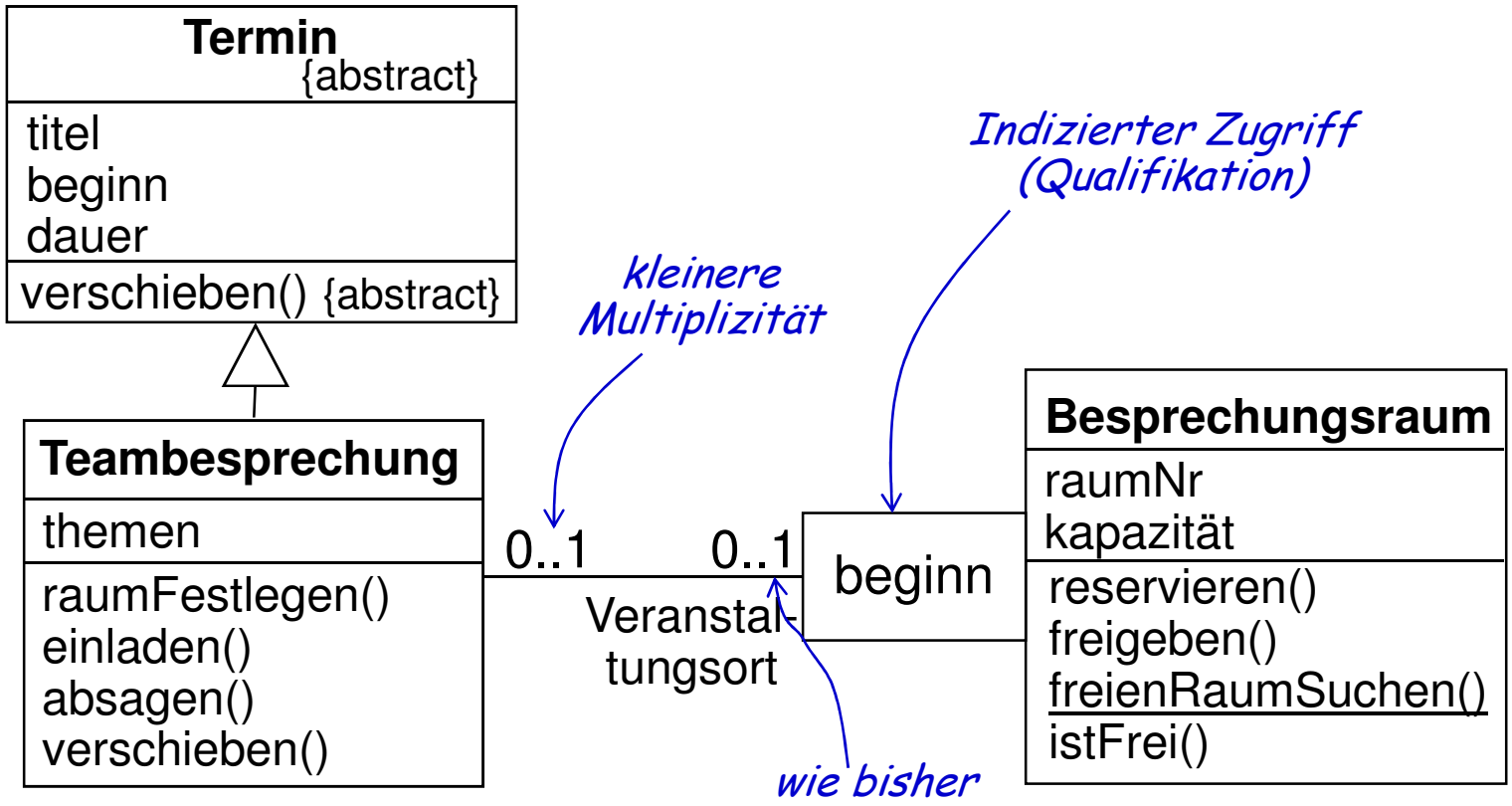
Qualifizierte Assoziation: Beispiel (1)



Raum12.istFrei(start=04.05.02 10:00, dauer=60min);

führt zu einer Suche über alle assoziierten Teambesprechungen !

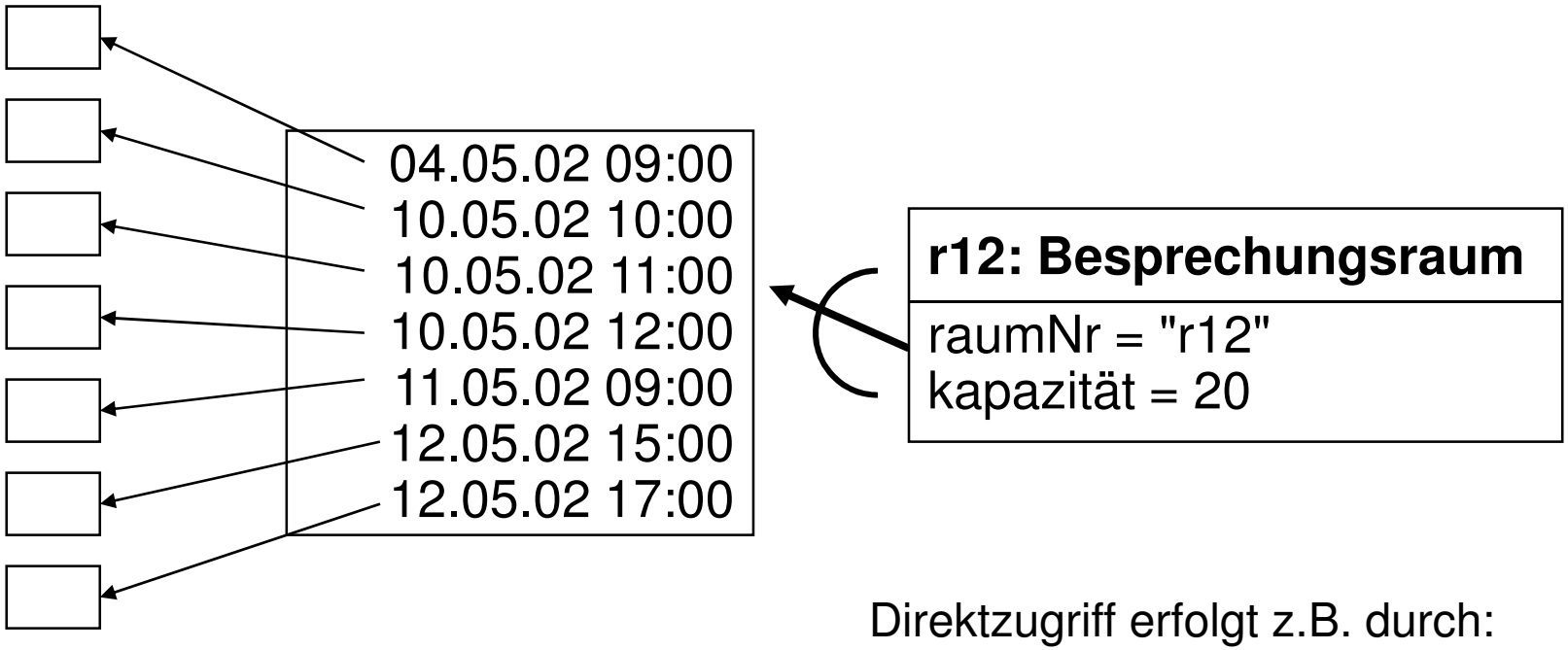
Qualifizierte Assoziation: Beispiel (2)



Raum12.istFrei(start=04.05.02 10:00, dauer=60min);

kann direkt nach Datum abfragen, ob eine Assoziation besteht

Realisierung einer qualifizierten Assoziation



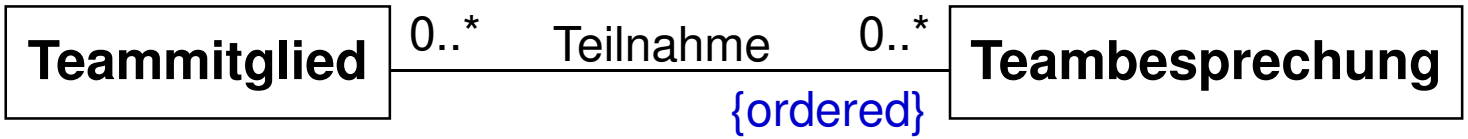
Teambesprechungs-Objekte

Direktzugriff erfolgt z.B. durch:

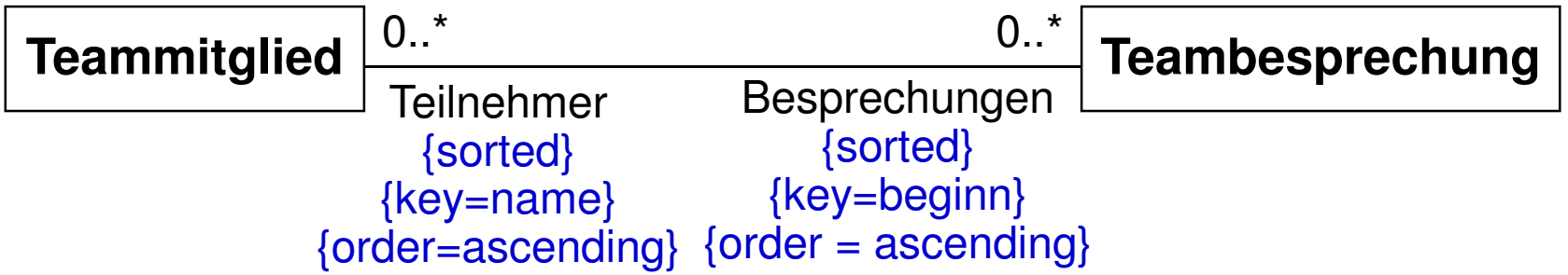
Hashfunktion
(Berechnung des Indexwerts aus gegebenem Datum)

Sortierte Baumstruktur

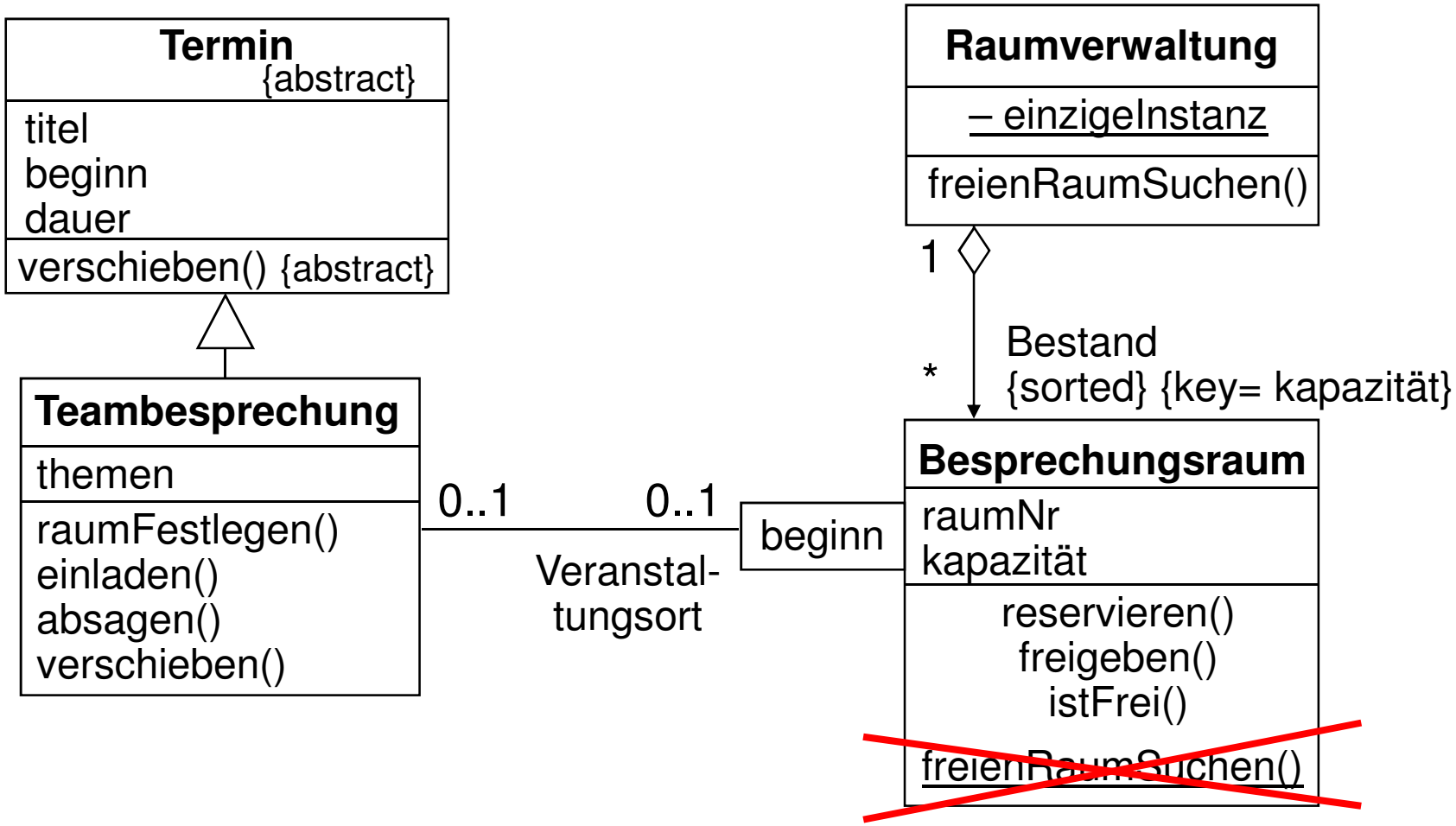
Geordnete und sortierte Assoziation



- {ordered} an einem Assoziationsende:
  - Es besteht eine feste Reihenfolge, in der die assoziierten Objekte durchlaufen werden können → Oft ist Zugriff über Listen, Iteratoren möglich
  - Mehrfachvorkommen eines Objekts sind verboten
- Default: die assoziierten Objekte sind als *Menge* strukturiert.
- Weitere Einschränkungen möglich,
  z.B. die Forderung nach Sortierung gemäß bestimmter Attribute:



Verwaltungsklassen

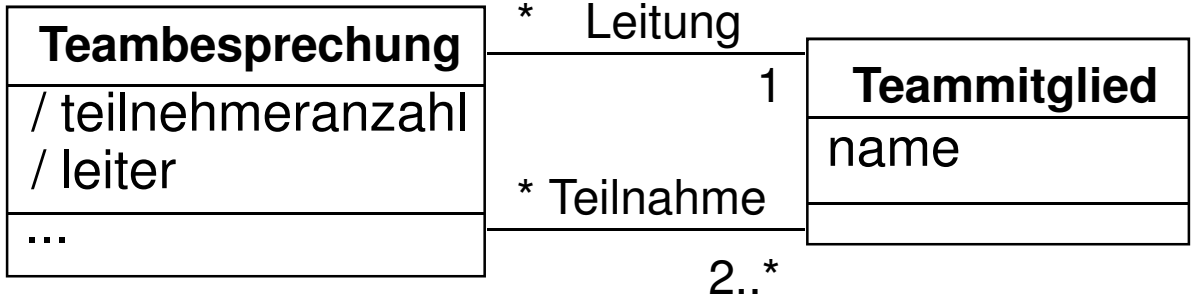


Abgeleitete (redundante) Elemente

- **Definition** Ein ***abgeleitetes*** Modellelement (z.B. Attribut, Assoziation) ist ein Modell-Element, das jederzeit aus anderen (nicht abgeleiteten) Elementen rekonstruiert werden kann.
- **Notation**

  / Modellelement oder

  Modellelement {derived}
- **Beispiele:**



{leiter = Leitung.name}
{teilnehmeranzahl = Teilnahme->size}

Ableitung kann formuliert werden mit der Object Constraint Language OCL, ein weiterer Teil der UML.

Parameter und Datentypen für Operationen

- Analysephase:
  - oft Operationsname ausreichend
  - ggf. Parameternamen ohne weitere Information

- Entwurfsphase:
  - Parameter und Datentypen der Operationen genau festlegen !

- **Beispiele** (Klasse Besprechungsraum)**:**

  + freienRaumSuchen
    (plaetze: int, start: Date, dauer: int=60, wunschraum:
    Besprechungsraum): Besprechungsraum

  – istFrei(beginn: Date, dauer: int):boolean

  + reservieren (für: Termin):boolean;

Spezifikation von Operationen

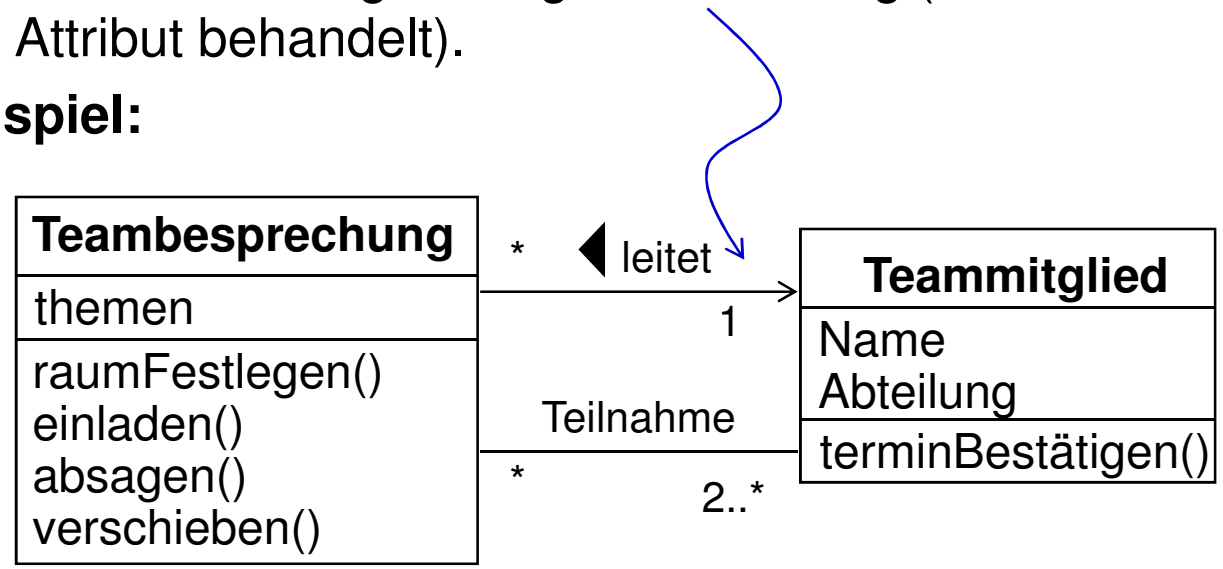
- **Definition** Die *Spezifikation* einer Operation legt das Verhalten der Operation fest, ohne einen Algorithmus festzuschreiben.
- Grundprinzip:

> Es wird das *"Was"* beschrieben und noch nicht das *"Wie"*.

- Häufigste Formen von Spezifikationen:
  - Text in natürlicher Sprache (oft mit speziellen Konventionen)
    - Oft in Programmcode eingebettet (Kommentare)
    - Werkzeugunterstützung zur Dokumentationsgenerierung, z.B. "javadoc"
  - Vor- und Nachbedingungen
  - Tabellen, spezielle Notationen
  - "Pseudocode" (Programmiersprachenartiger Text)
    - nur mit Vorsicht zu verwenden - oft zu viele Details festgelegt !

Navigationsrichtung von Assoziationen

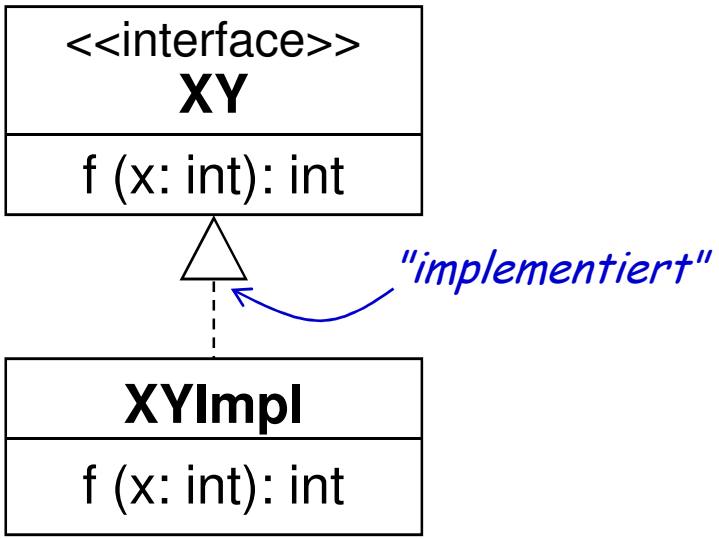
- Assoziationen werden verwendet, um im Objektgeflecht zu *navigieren*.
- Assoziationen sind im Normalfall in beiden Richtungen navigierbar (d.h. werden auf beiden Seiten wie ein Attribut behandelt).
- Spezialfall: einseitige Navigationsrichtung (d.h. nur auf einer Seite wie Attribut behandelt).
- **Beispiel:**



Schnittstellen

- Guter Software-Entwurf sichert *Homogenität* und *Ergonomie*.
  - Gleichartige Funktionalität soll in gleicher Weise aufrufbar sein.
  - *Schnittstelle (interface)* ist ein Sprachkonstrukt von UML und Java.
- **Definition:** Eine Schnittstelle ist eine abstrakte Klasse, die keine Attribute und keine Operationsrümpfe (Implementierungen) enthält.
  - Sammlung von Operations-Signaturen

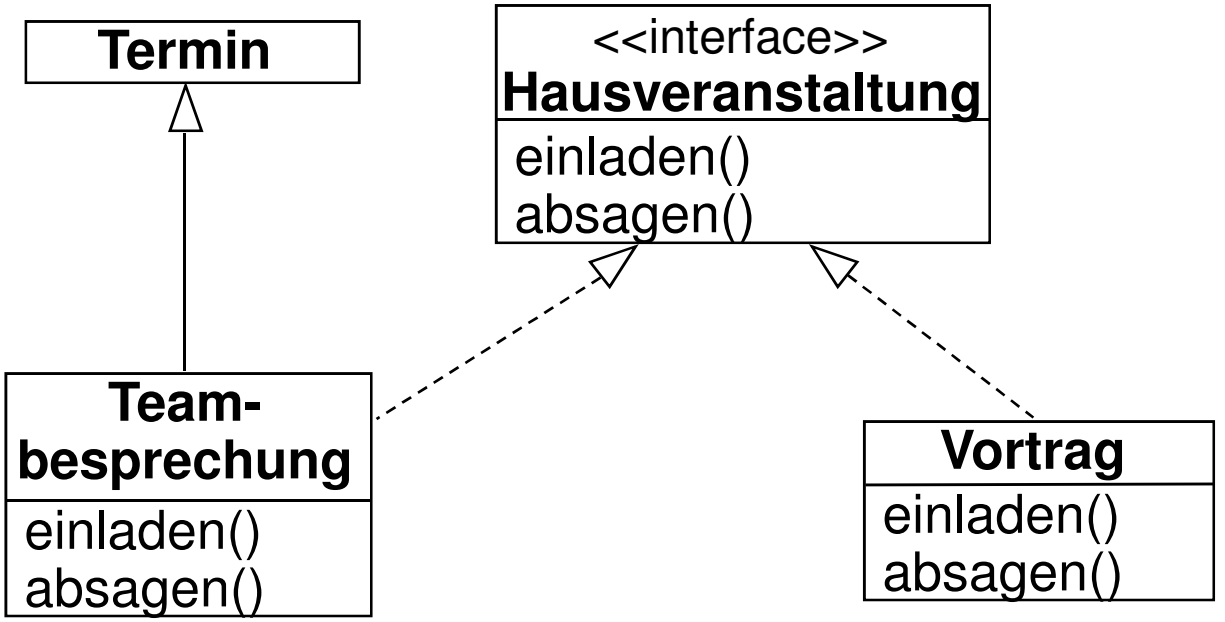
UML:



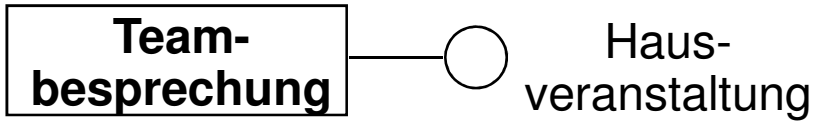
Java:

```
interface XY {
  int f (int x);
}

class XYImpl implements XY {
  public int f (int x) {
     ... hier Rumpf von f ...
  }
...
}
```

Einfache Vererbung durch Schnittstellen



Hinweis: "*lollipop*"-Notation für Schnittstellen ist weit verbreitet und in UML ebenfalls zulässig (und gleichwertig):



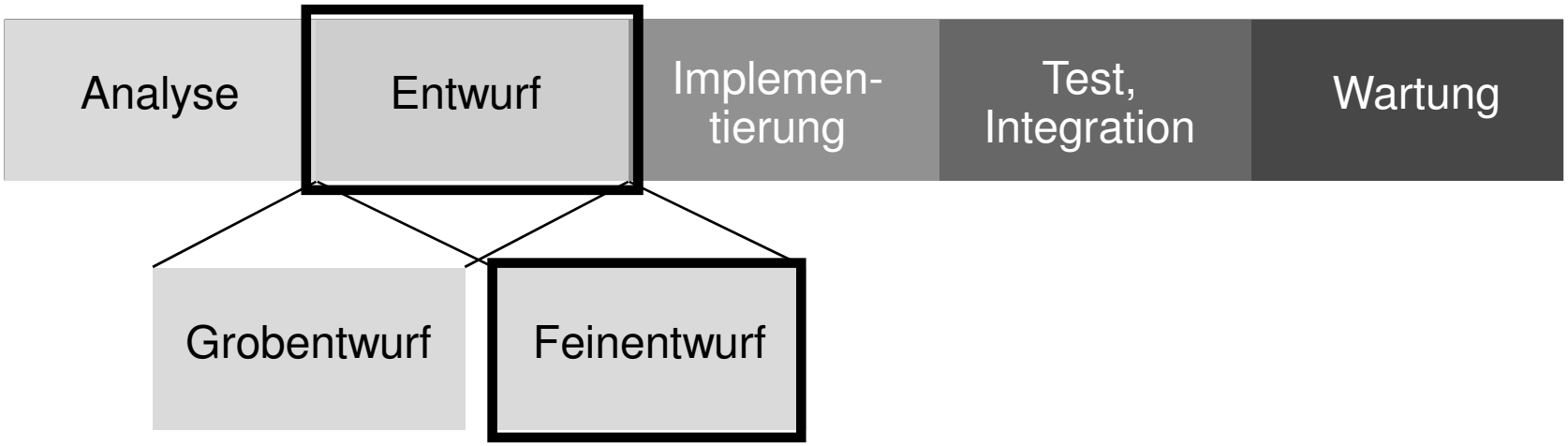
Schnittstellen und abstrakte Klassen

Abstrakte Klasse	**Schnittstelle**
Enthält Attribute und Operationen	Enthält nur Operationen (und ggf. Konstante)
Kann Default-Verhalten festlegen	Kann kein Default-Verhalten festlegen
Default-Verhalten kann in Unterklassen redefiniert werden	Unterklassen müssen Verhalten definieren
Java: Unterklasse kann nur von einer Klasse erben	Java: Eine Klasse kann mehrere Schnittstellen implementieren
	Schnittstelle ist eine spezielle Sicht auf eine Klasse

Zusammenfassung: UML-Klassenmodelle in Analyse und Entwurf

Analyse-Modell	**Entwurfs-Modell**
Skizze: Teilweise unvollständig in Attributen und Operationen	Vollständige Angabe aller Attribute und Operationen
Datentypen und Parameter können noch fehlen	Vollständige Angabe von Datentypen und Parametern
Noch kaum Bezug zur Realisierungssprache	Auf Umsetzung in gewählter Programmiersprache bezogen
Keine Überlegungen zur Realisierung von Assoziationen	Navigationsangaben, Qualifikation, Ordnung, Verwaltungsklassen
	Entscheidung über Datenstrukturen
	Vorbereitung zur Anbindung von Benutzungsoberfläche und Datenhaltung an fachlichen Kern

6. Software- & Systementwurf
6.5. Entwurfsmuster



Prof. Dr. Bernhard Rumpe
Software Engineering
RWTH Aachen

http://www.se-rwth.de/

Literatur:

• Gamma/Helm/Johnson/Vlissides: Design Patterns, Addison-Wesley 1994
(= „Gang of Four“, „GoF“)
• Buschmann/Meunier/Rohnert/Sommerlad/ Stal: A System of Patterns, Wiley 1996

Beschreibung eines Entwurfsmusters (nach GoF)

- Name
- Problem
  - Motivation
  - Anwendungsbereich
- Lösung
  - Struktur (Klassendiagramm)
  - Bestandteile (meist Klassen-, Assoziations- und Operationsnamen):
    - "Rollennamen", d.h. Platzhalter für Bestandteile der Anwendung
    - feste Bestandteile der Implementierung
  - Objektinteraktion (Abläufe, evtl. Sequenzdiagramm)
- Diskussion
  - Vor- und Nachteile
  - Abhängigkeiten, Einschränkungen
  - Spezialfälle
  - Bekannte Verwendung

Beispiel: Text zu "Adapter" (1)

Ausschnitt aus "Design Patterns":

ADAPTER

Also known as: Wrapper

Motivation:

Sometimes a toolkit class that's designed for reuse isn't reusable only because its interface doesn't match the domain-specific interface an application requires.

Consider for example a drawing editor that lets users draw and arrange graphical elements (lines, polygons, text etc) into pictures and diagrams. ... (eine Seite Text)

Applicability:

Use the Adapter pattern when

•you want to use an existing class, and its interface does not match the one you need.

•you want to create a reusable class that cooperates with unrelated or unforeseen classes ...

Beispiel: Text zu "Adapter" (2)

Ausschnitt aus "Design Patterns":

ADAPTER (Fortsetzung)

Structure: (Zwei Klassendiagramme + weitere Information)

Consequences: (1 Seite Text)

Implementation: (2 Seiten Text mit Klassendiagrammen)

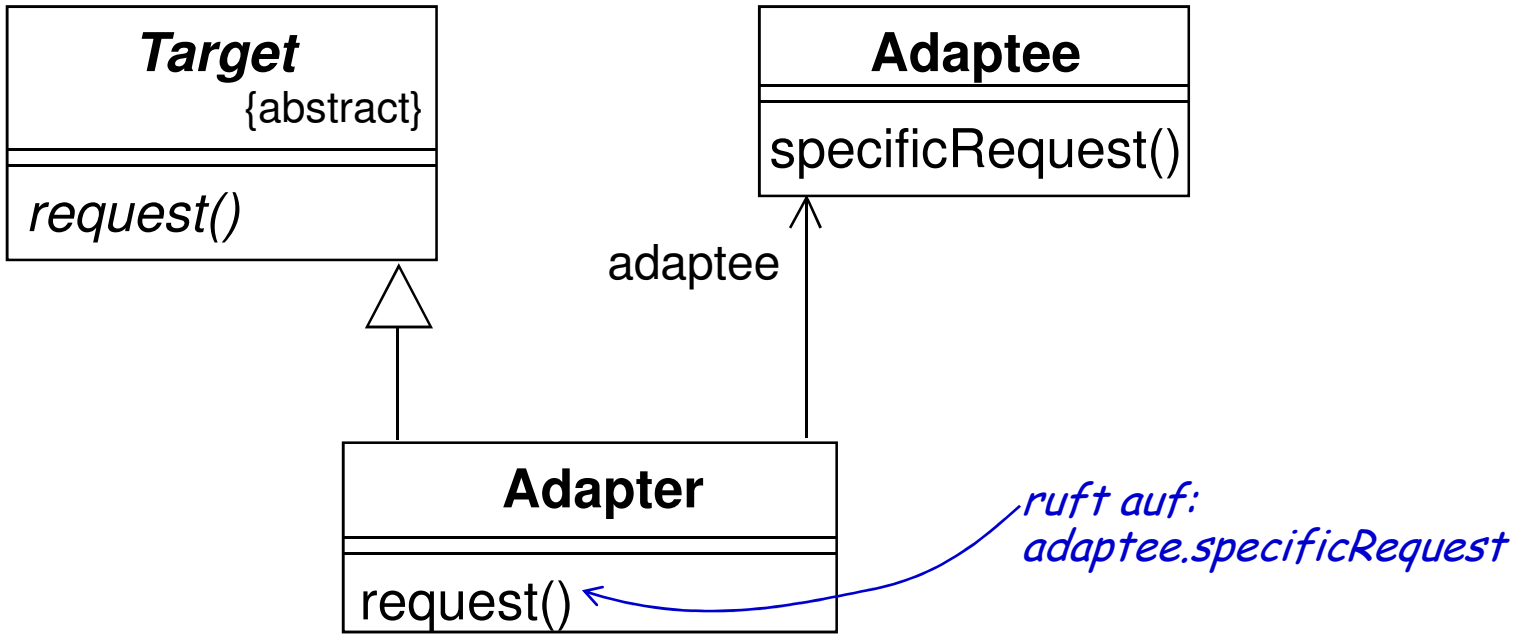
Sample Code: (3 Seiten Text mit C++-Beispielen)

Known Uses: (1 Seite Text)

Related Patterns: (1/4 Seite Text)

Strukturmuster Adapter
Variante 1: Objektadapter

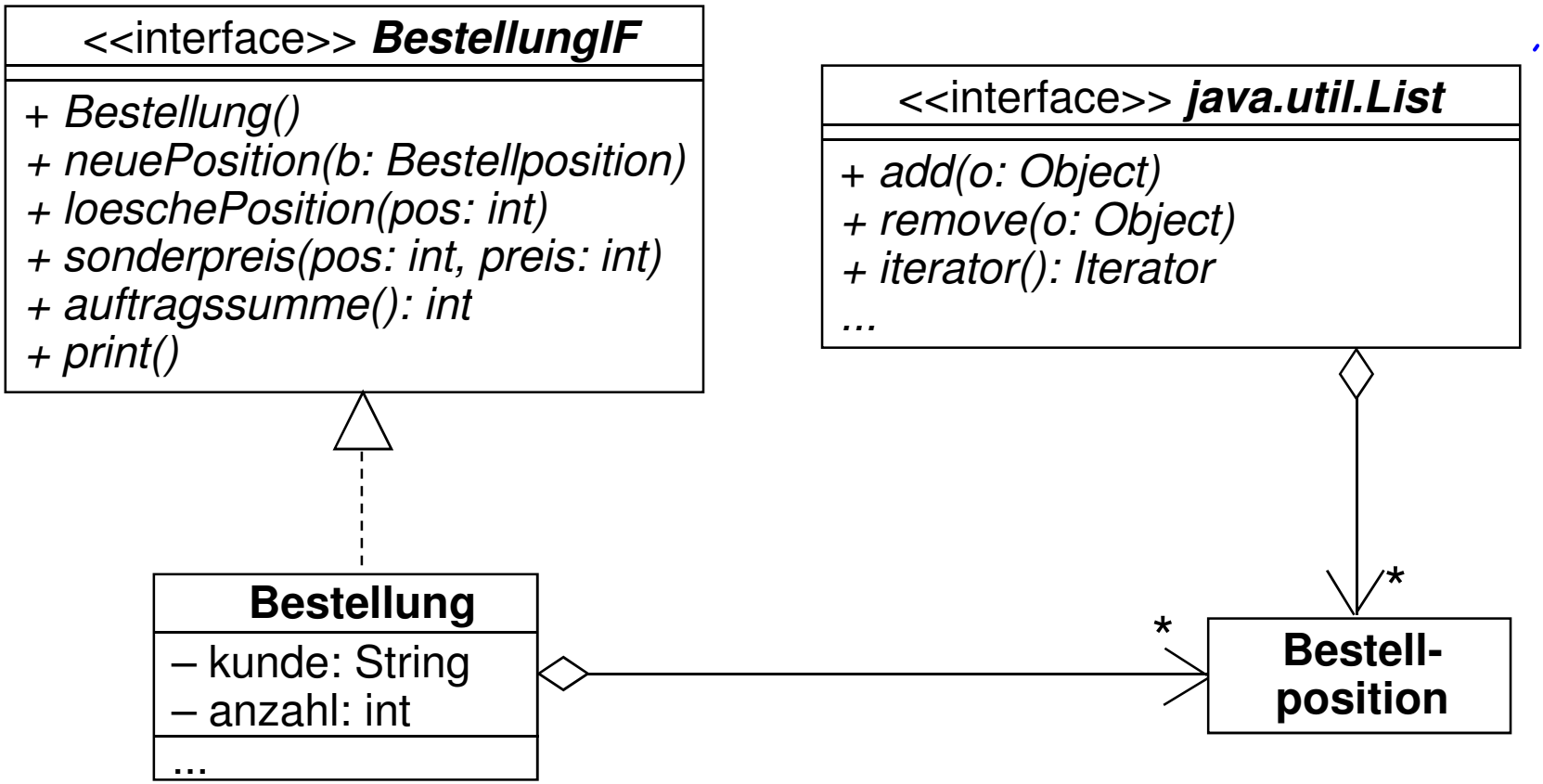
- Name: **Adapter** (auch: Wrapper)
- Problem:
  - Anpassung der Schnittstelle eines vorgegebenen Objekts (*adaptee*) auf eine gewünschte Schnittstelle (*target*)
- Lösung:



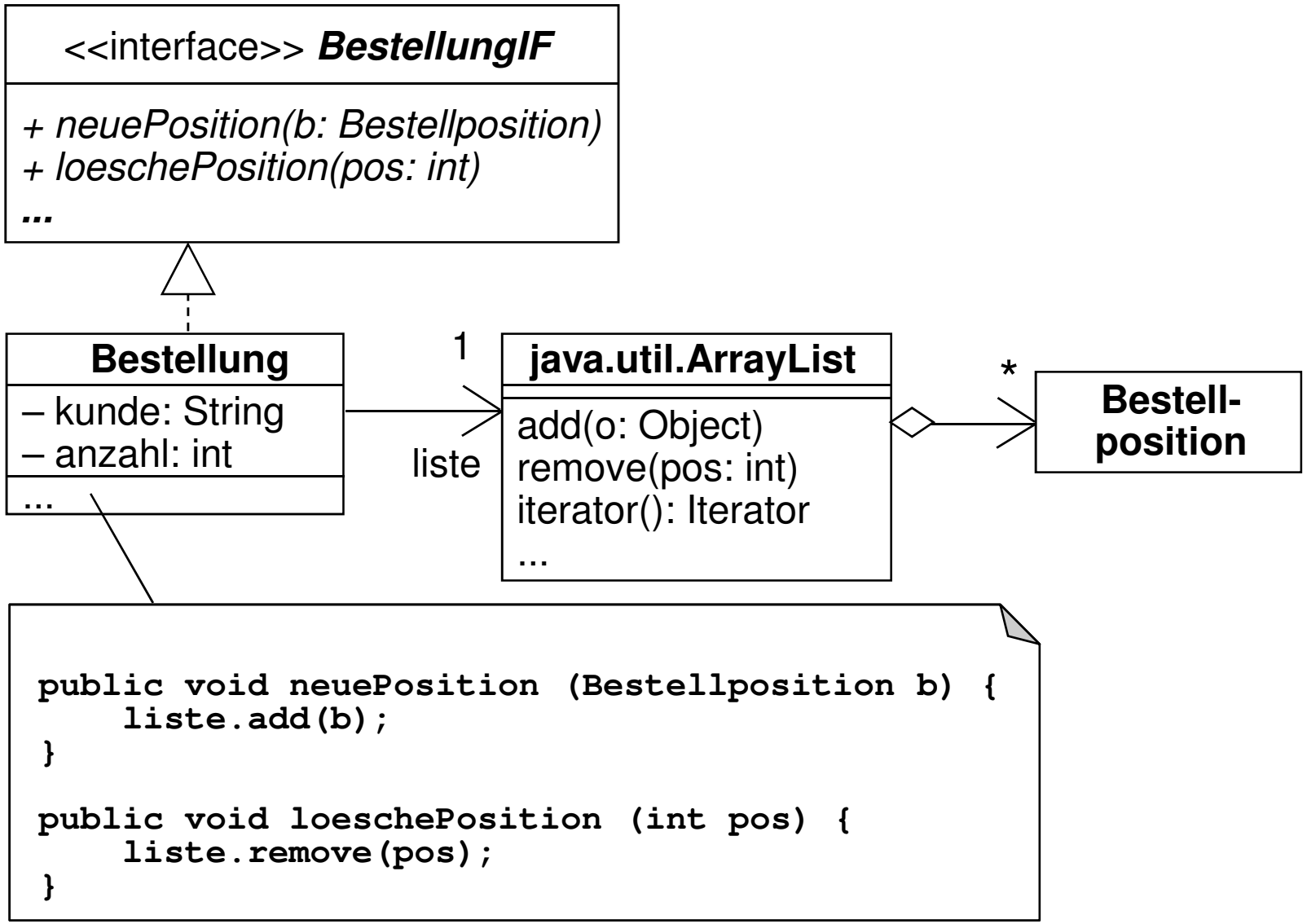
Objektadapter-Beispiel (1)

discuss

Schönheitsfehler?

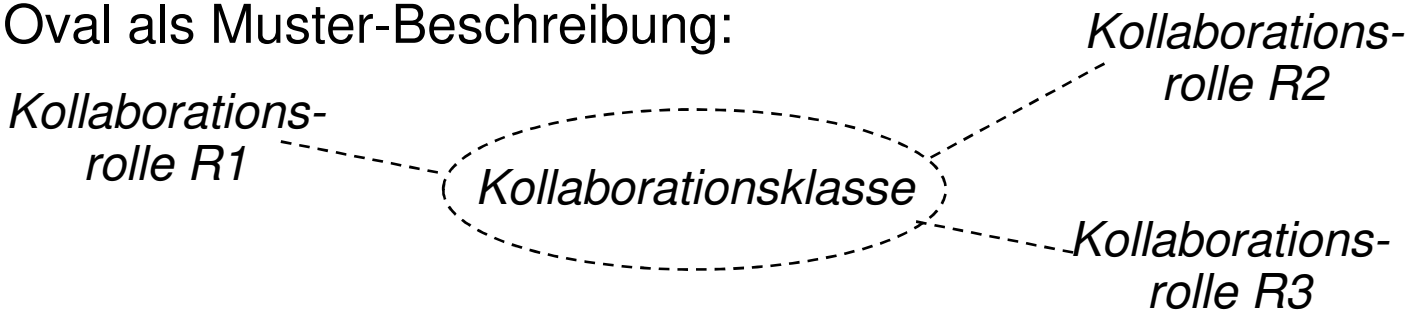


Objektadapter-Beispiel (2)



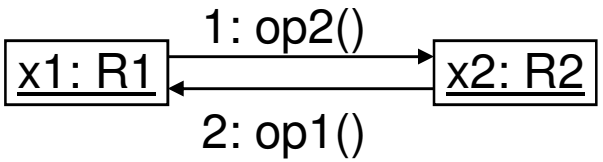
UML-Notation für Entwurfsmuster

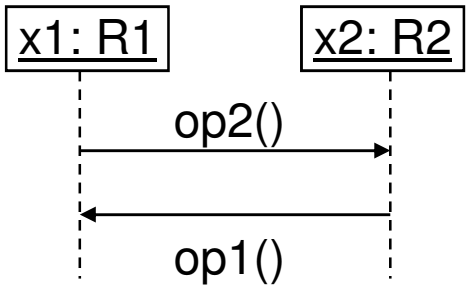
- Oval als Muster-Beschreibung:



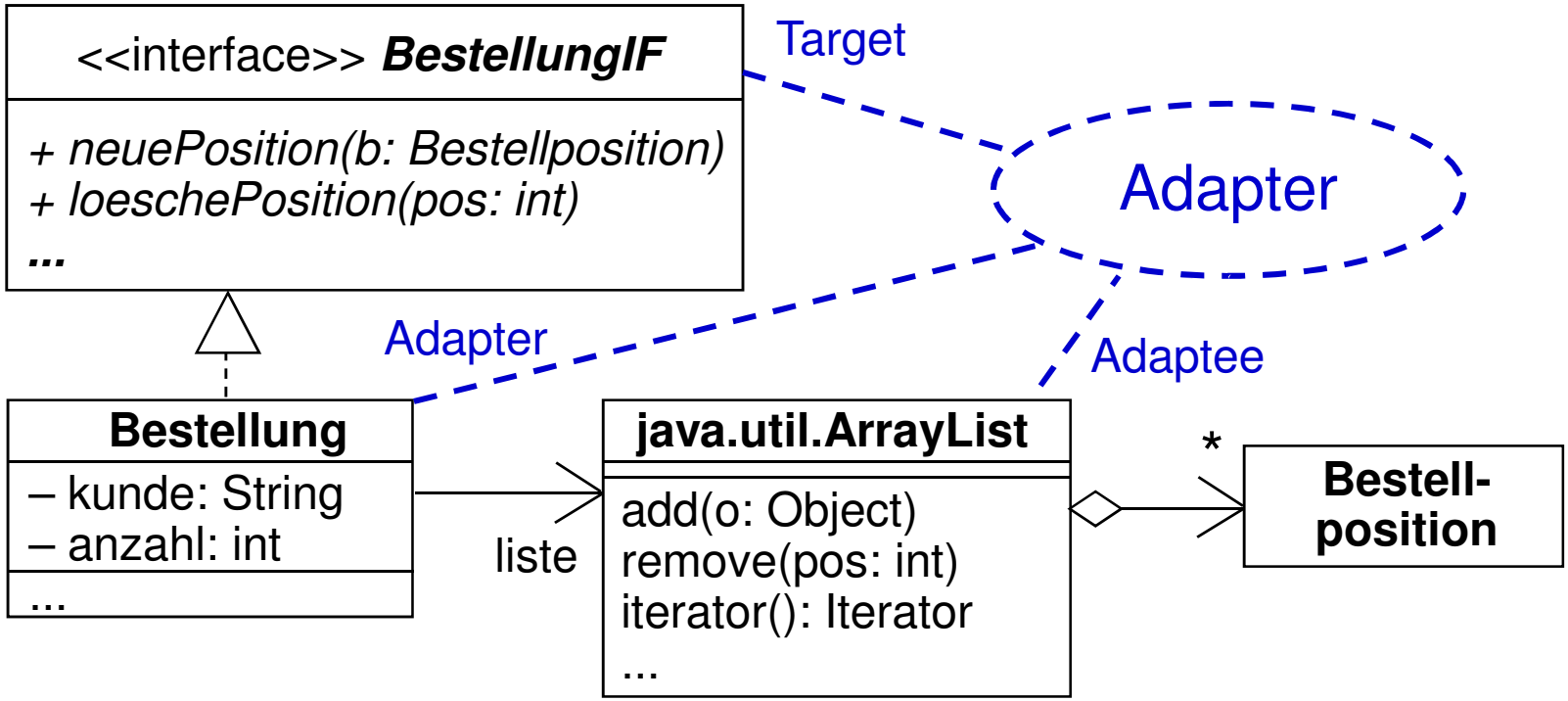
  - Muster:
    Kollaborationsklasse = Mustername,
    Rollen = Platzhalternamen

- Ablauf einer Kollaboration beschrieben durch Kollaborationsdiagramm (äquivalent zu Sequenzdiagramm):





Objektadapter-Beispiel in UML (3)

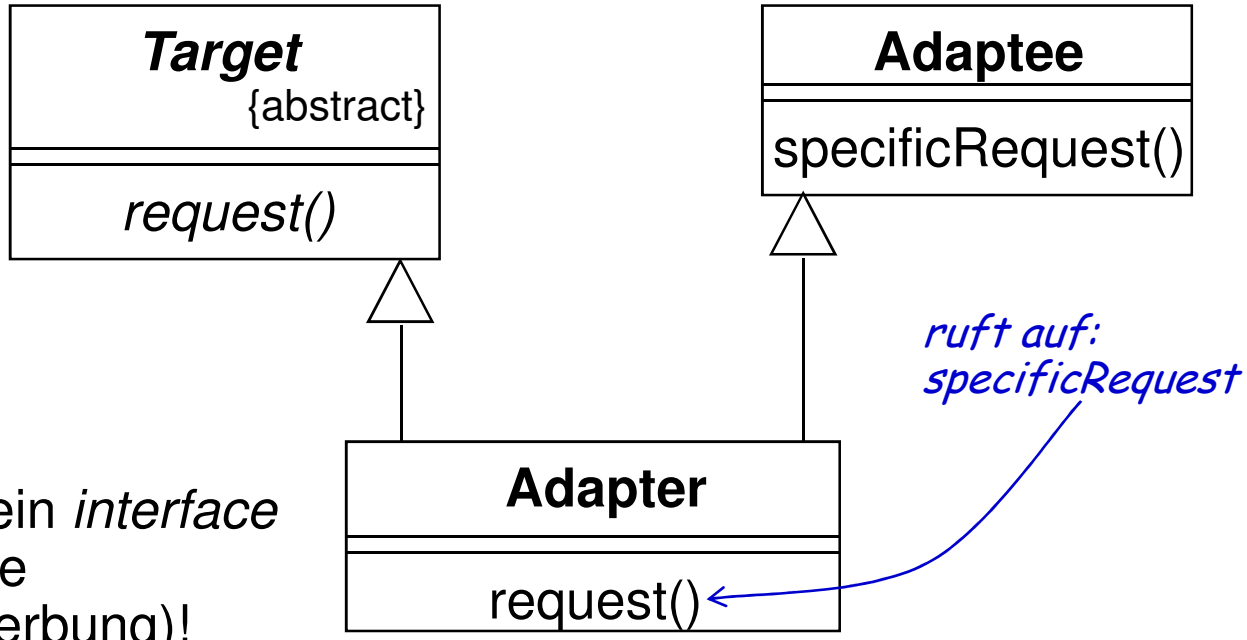


Anwendung eines Musters

- Kein mechanisches "Pattern Matching"!
  - Eher Übertragung der Idee des Musters
- Grundstruktur des Musters sollte sich wiederfinden lassen
  - ggf. vorliegenden Entwurf etwas anpassen bzw. anders darstellen
- Auch Verhaltensschema muss im Code ähnlich zur Musteridee auftreten:
  - Muster:
    Adapter.request() ruft Adaptee.specificRequest() auf
    (und tut nicht wesentlich mehr)
  - Konkreter Fall:
    Bestellung.neuePosition() ruft ArrayList.add() auf
    Bestellung.loeschePosition() ruft ArrayList.remove() auf
    ...

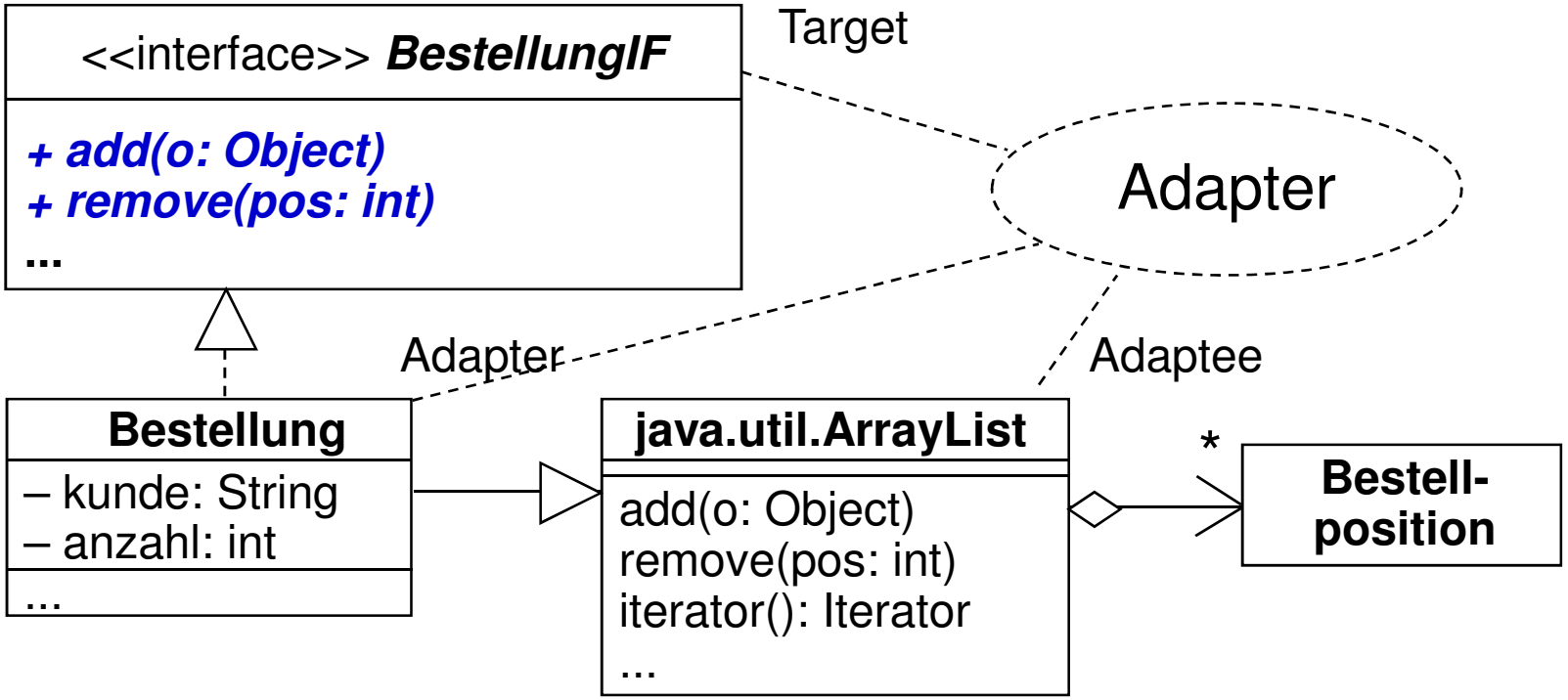
Strukturmuster Adapter
Variante 2: Klassenadapter

- Name: **Adapter** (auch: Wrapper)
- Problem:
  - Anpassung der Schnittstelle eines vorgegebenen Objekts (*adaptee*) auf eine gewünschte Schnittstelle (*target*)
  - Viele Operationen sind identisch in Adaptee und Target haben aber unterschiedliche Namen oder Argumentreihenfolgen
- Lösung:



Java: Target muss sogar ein *interface* sein (da keine Mehrfachvererbung)!

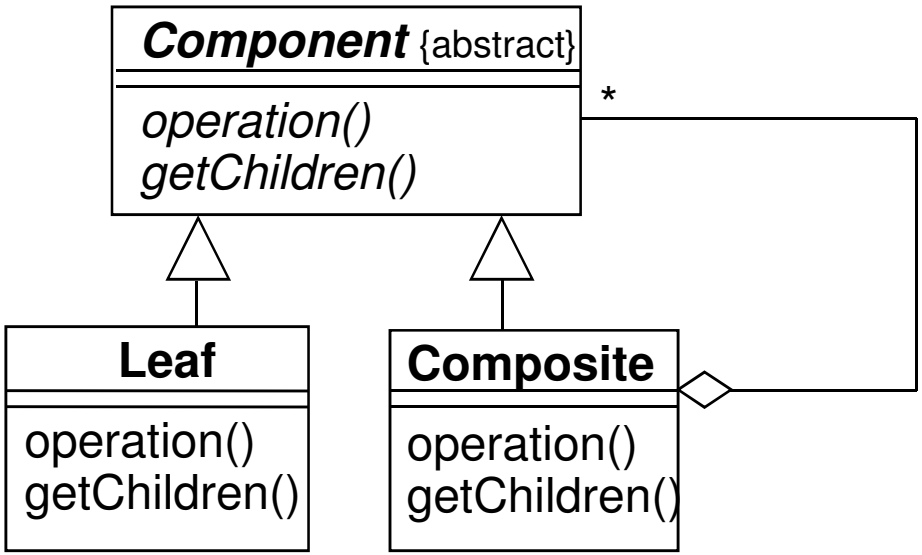
Klassenadapter-Beispiel



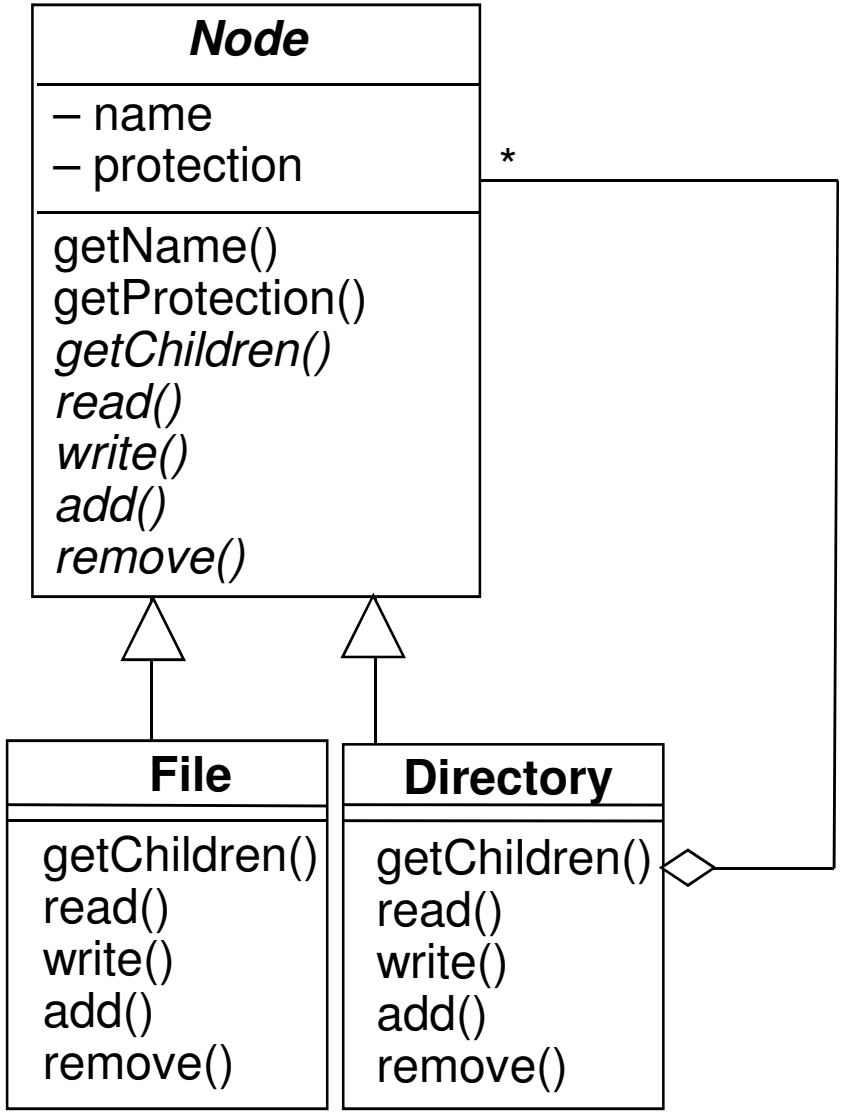
Achtung:
Es werden *alle* Operationen des Adaptees an den Adapter vererbt, auch eventuell unerwünschte !

Entwurfsmuster Composite

- Strukturmuster
- **Problem:** Hierarchische Struktur von Objekten
- **Lösung:** Einheitliche abstrakte Schnittstelle für „Blätter“ und Verzweigungsknoten eines Baumes



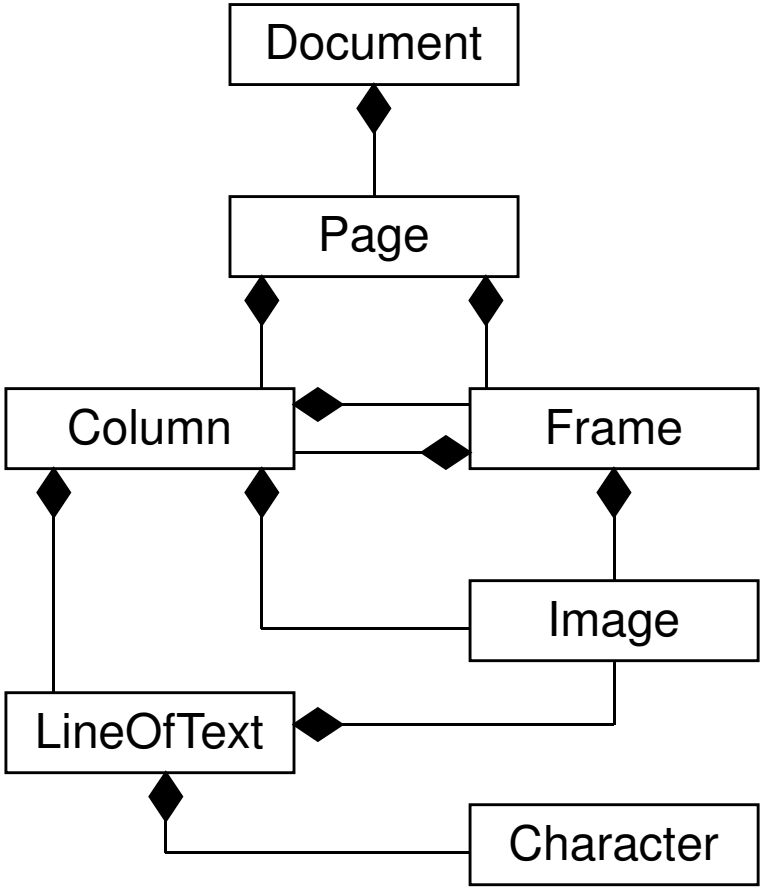
Anwendung des Composite-Musters



- flexible Zugriffsschicht auf Dateisysteme
  - Gemeinsame Operationen auf Dateien und Verzeichnissen:
  - Name, Größe, Zugriffsrechte, ...
- Teile-Strukturen für Geräte
- Ahnentafeln (Bäume...)

Composite - Genaueres Beispiel

- Aufgabe: Dokument-Struktur und -Formatierung (Grand, S. 170)
- Erstes Klassendiagramm (aus Analyse):

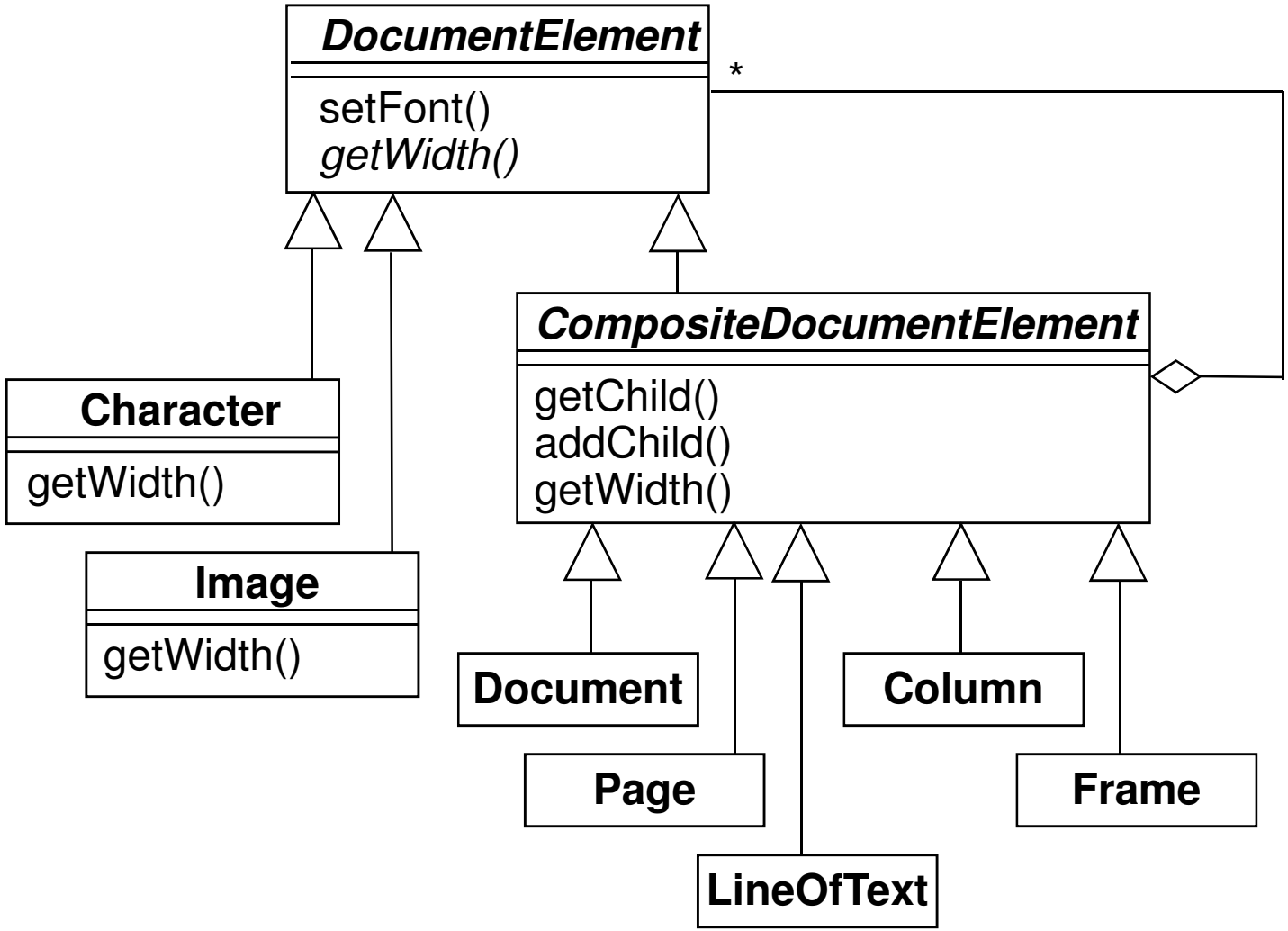


Grundidee: Verantwortlichkeiten trennen

- "Separation of concerns":
  - Jede Einheit soll ***einen*** Aufgabenkomplex gut lösen.
  - Der Aufgabenkomplex einer Einheit soll in sich geschlossen sein (hohe *Kohäsion*).
  - Die Einheit soll so wenig wie möglich von anderen Einheiten abhängen
    (niedrige *Kopplung*).

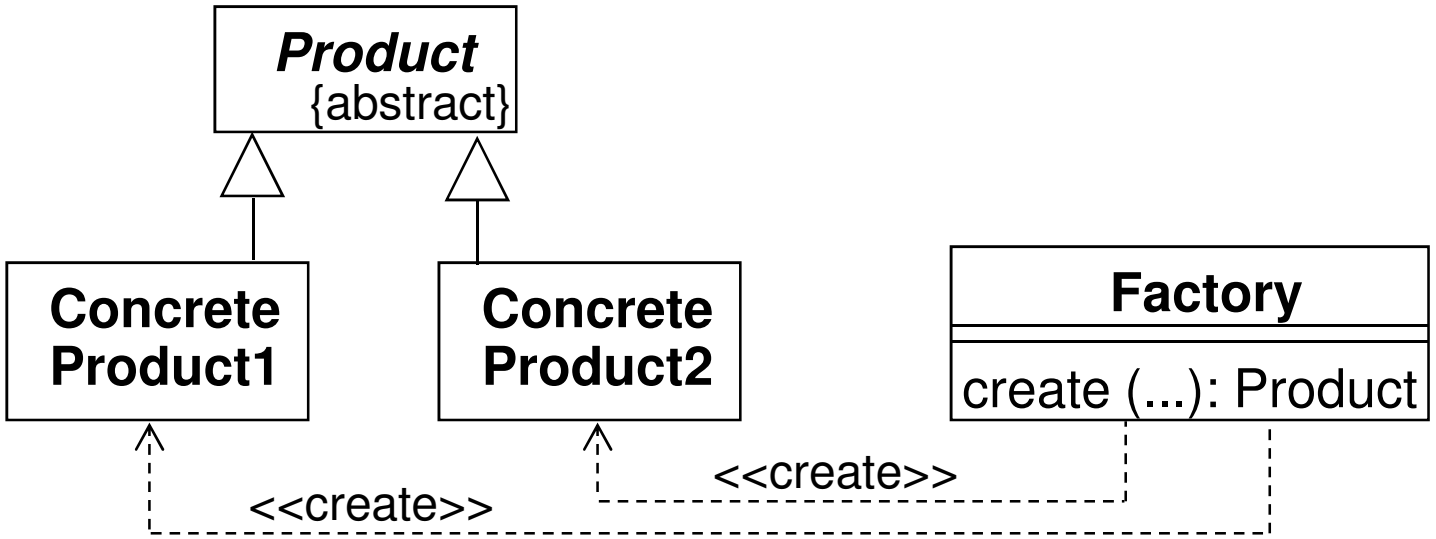
- Praktische Anwendung im Composite-Muster:
  - Column sollte nicht davon abhängen, dass gerade LineOfText, Image und Frame zulässige Elemente sind. (Analog für andere Klassen)
  - Die Mechanismen zur Realisierung der Komposition sollten in einer Klasse zusammengefasst werden.
  - Es gibt einige Operationen, die für (fast) alle Dokument-Elemente einheitlich verwendbar sind. Die Liste dieser Operationen sollte in einer Klasse zusammengefasst werden.

Anwendung des Composite-Musters

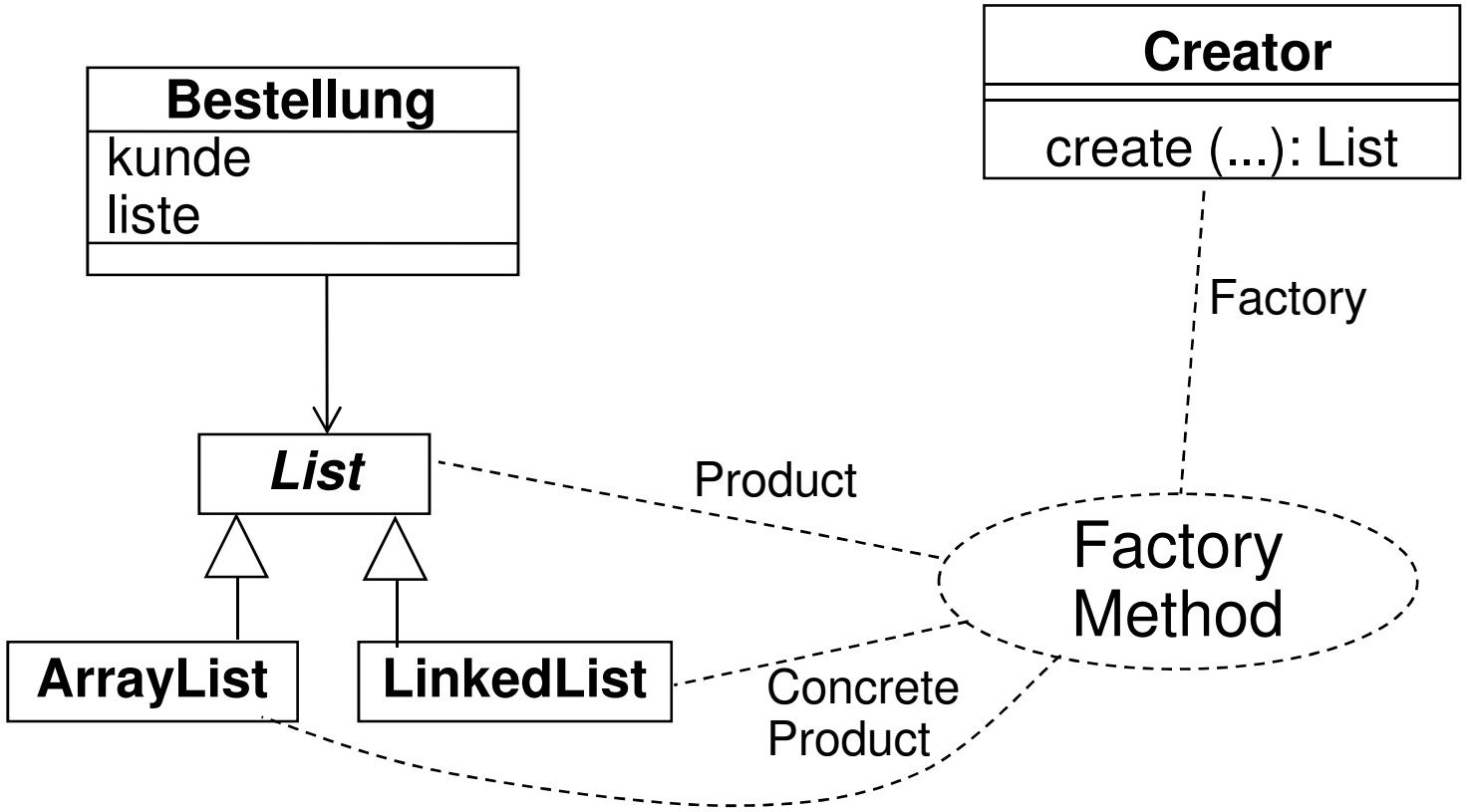


Erzeugungsmuster Factory Method

- Name: **Factory Method**
  (dt.: Fabrikmethode, auch: Virtueller Konstruktor)
- Problem:
  - Bei der Erzeugung von Objekten soll zwischen Varianten gewählt werden; dies soll aber zum Zeitpunkt der Erzeugung geschehen, ohne dass der Auftraggeber der Erzeugung damit beschäftigt ist.
- Lösung:

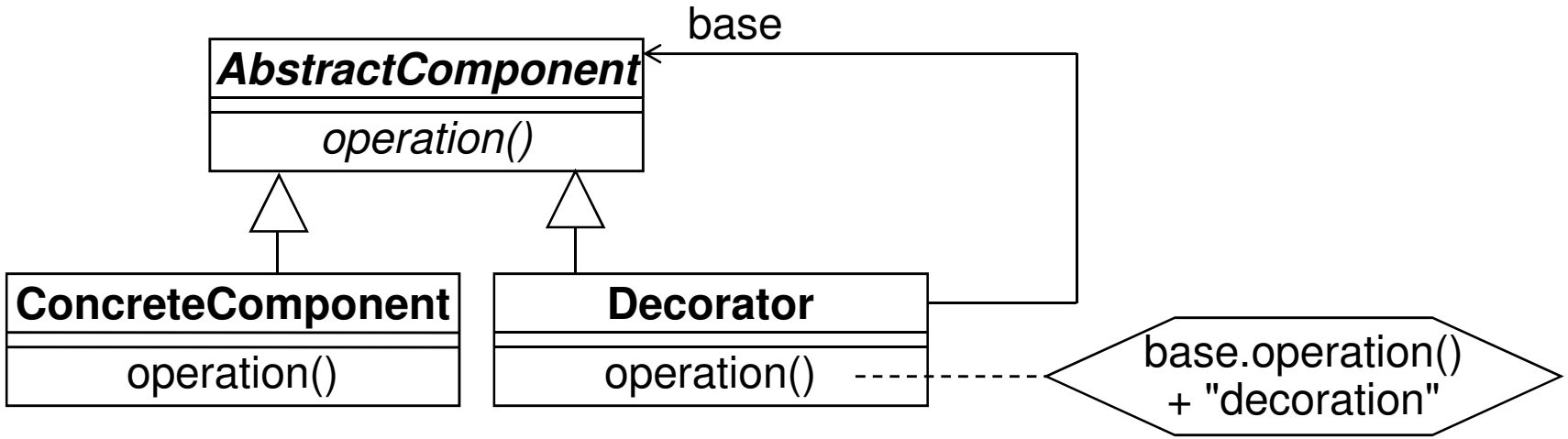


Factory Method - Beispiel



Entwurfsmuster Decorator

- Strukturmuster
- **Problem:** Zu einer Klasse, die eine abstrakte Schnittstelle implementiert, sollen flexibel weitere Eigenschaften hinzugefügt werden, so dass eine neue Klasse entsteht, die die gleiche Schnittstelle implementiert.
- **Lösung:** Definition einer Klasse für Zwischenobjekte, die die Operationen an die Originalklasse delegieren, nachdem sie ggf. zusätzliche Funktionalität erbracht haben.



Achtung: Etwas vereinfachte Version (ohne abstrakte Decorator-Oberklasse)

Entwurfsmuster Singleton

- Erzeugungsmuster
- **Problem:** Manche Klassen sind nur sinnvoll, wenn sichergestellt ist, dass immer höchstens eine Instanz der Klasse besteht (und diese bei Bedarf erzeugt wird).
- **Lösung:**
  - Modellebene: Klassen als Singleton auszeichnen
  - Programmebene: Sprachabhängig
- Beispiel (Java):

```
class Singleton {
  private static Singleton theInstance;

  private Singleton () {
  }

  public static Singleton getInstance() {
    if (theInstance==null)
      theInstance = new Singleton();
    return theInstance;
  }
}
```

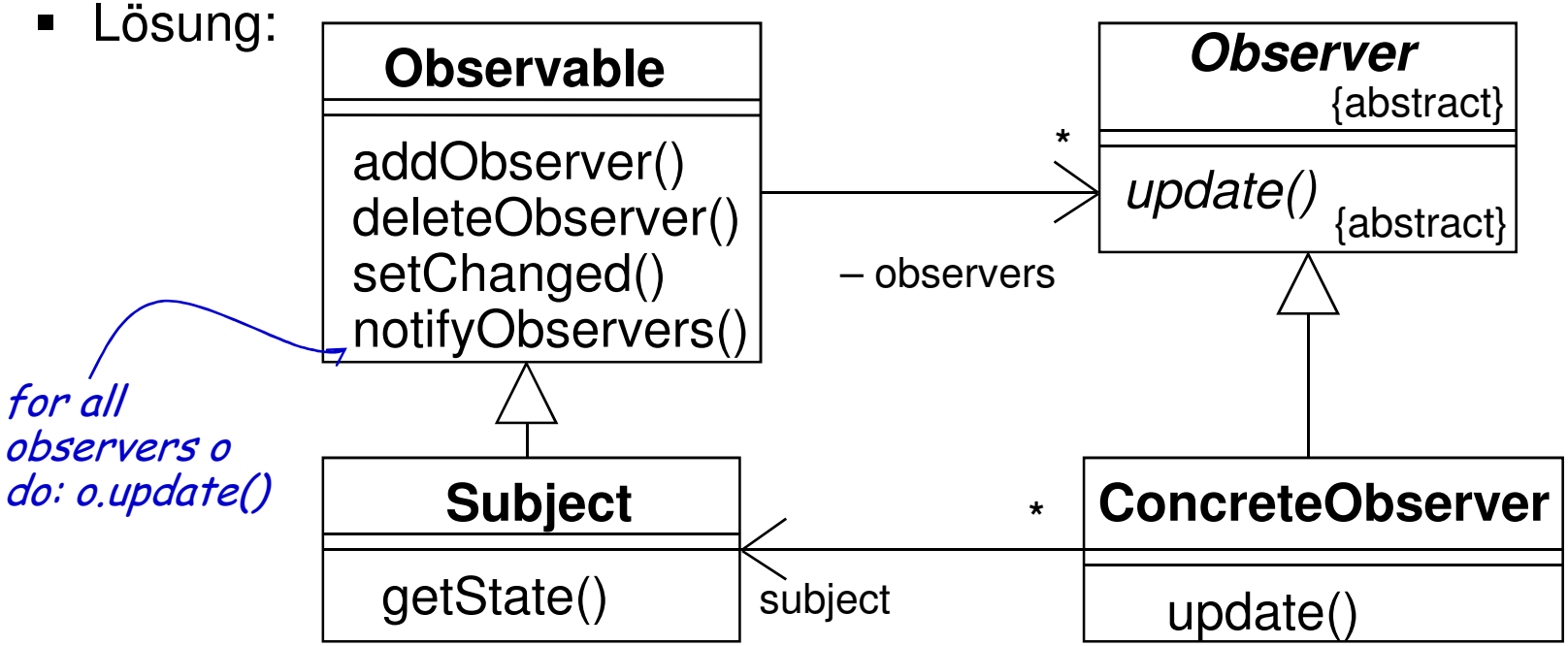
Entwurfsmuster Singleton

- Beispiele:
  - Protokoll-, DB- Schnittstellen
  - „Manager“-Klassen sind meist Singletons.

- Lösungen sind technisch verschieden je nach Programmiersprache, umfassen aber immer die selben wesentlichen Elemente (Variable theInstance, Modifikationen beim Konstruktor).

Verhaltensmuster Observer

- Name: **Observer** (dt.: Beobachter)
- Problem:
  - Mehrere Objekte sind interessiert an bestimmten Zustandsänderungen eines Objektes
- Lösung:



Konkrete Realisierungen weichen meist in Details ab (z.B. *Interface* Observer) !

java.util.Observable, java.util.Observer

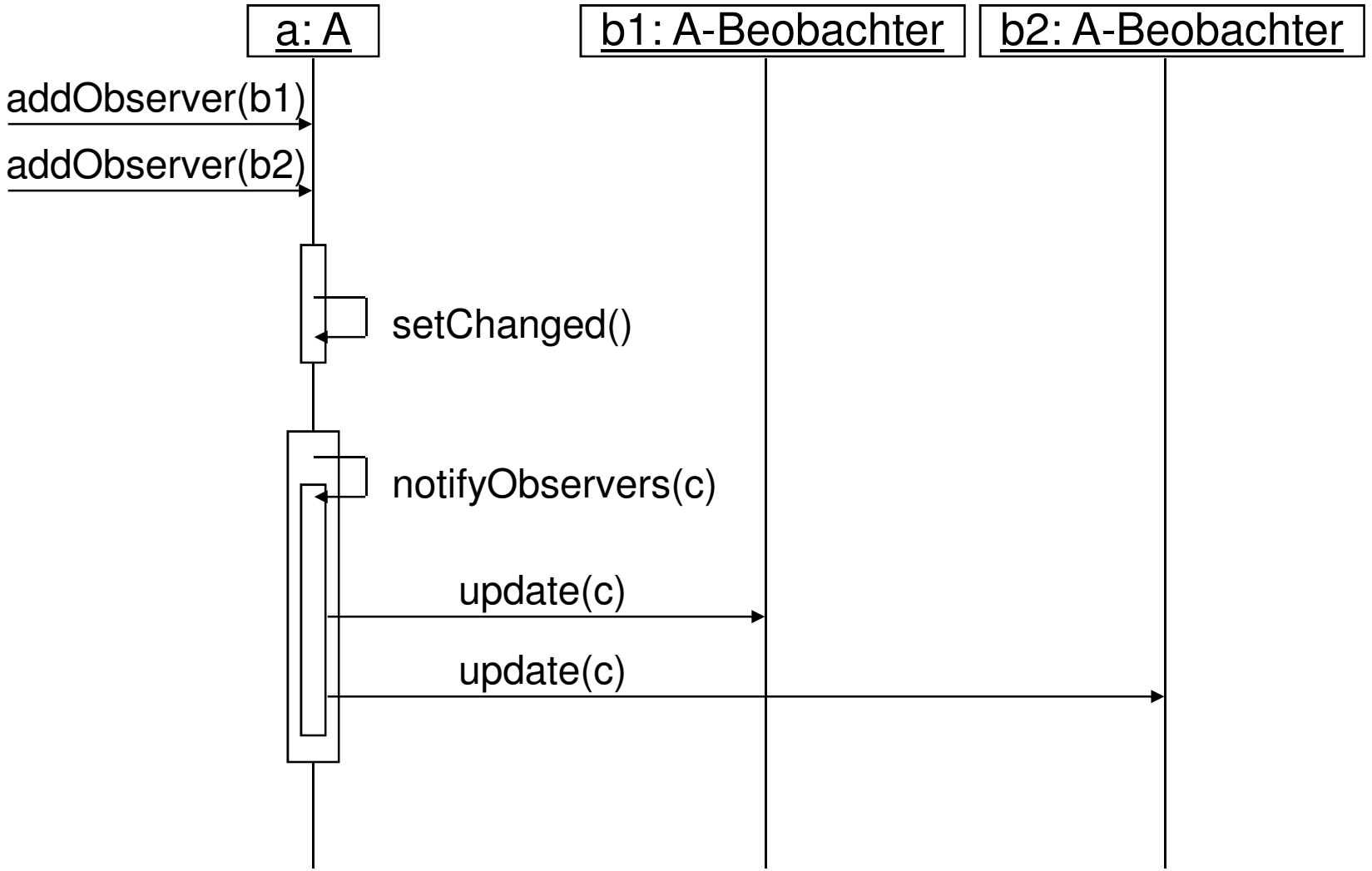
```
public class Observable {

  public void addObserver (Observer o);
  public void deleteObserver (Observer o);

  protected void setChanged();
  public void notifyObservers ();
  public void notifyObservers (Object arg);
}

public interface Observer {

  public void update (Observable o, Object arg);
}
```

Argumente für notifyObservers():

– meist nur Art der Änderung, nicht gesamte Zustandsinformation

– Beobachter können normale Methodenaufrufe nutzen, um sich näher zu informieren.

Beispielablauf beim Observer



Vorteile des Observer-Musters

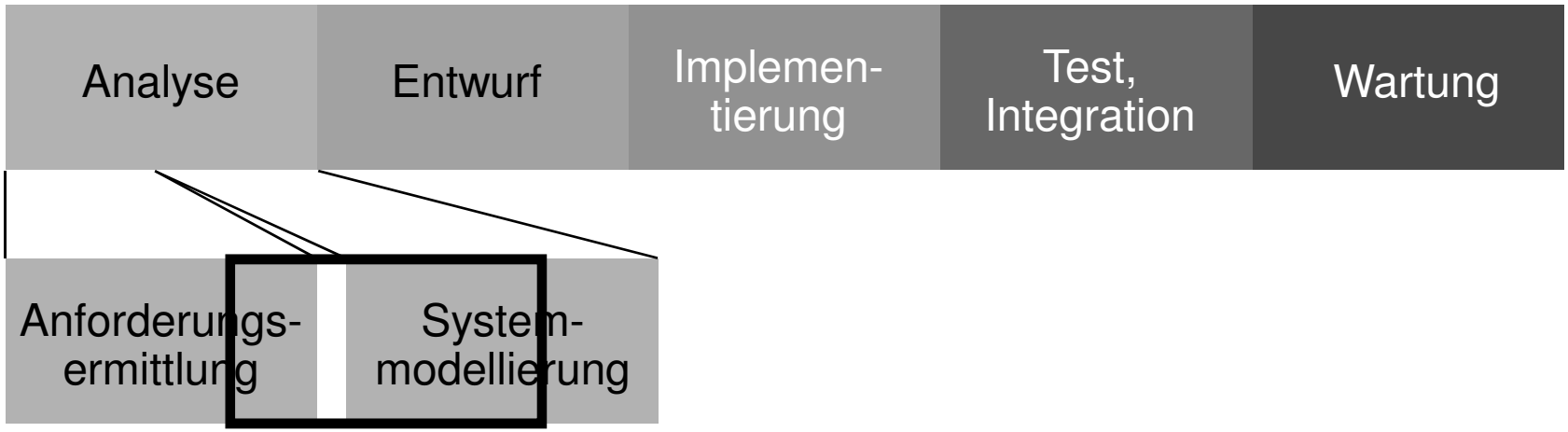
- Jede Klasse des Modells definiert lokal,
  - welche Veränderungen beobachtbar sind (d.h. ***aktiv*** der Umwelt mitgeteilt werden) und
  - wann die Mitteilung erfolgen soll.
- Eine beobachtete Klasse
  - hat keine Information darüber, was die Beobachter konkret tun;
  - muss nicht geändert werden, wenn sich Beobachter verändern;
  - muss nicht geändert werden, wenn neue Beobachter dazukommen oder Beobachter wegfallen.

- Methodik:
  - Hinweise auf wichtige Zustandsveränderungen gibt das UML-Zustandsdiagramm.

Zusammenfassung 6.5: Entwurfsmuster

- Die 23 von den GoF definierten Entwurfsmuster lassen sich in drei Kategorien teilen:
  - Struktur-,
  - Erzeugungs- und
  - Verhaltensmuster
- Eine Reihe weiterer Entwurfsmuster wurde in den letzten Jahren entwickelt, die teilweise spezifische Problemstellungen behandeln
- Entwurfsmuster sind ein sehr hilfreiches Instrument zur
  - Verbesserung der Struktur des Codes
  - Kommunikation über Entwurfsentscheidungen
- „Muster“ werden nicht direkt umgesetzt, sondern den Anforderungen angepasst: Muster diesen als Schablonen für Entwurfsideen.
- In dieser Vorlesung wurden einige der wesentlichen Muster beschrieben.

Nachtrag:
6.6. Muster in der Objektorientierten Analyse



Prof. Dr. Bernhard Rumpe
Software Engineering
RWTH Aachen

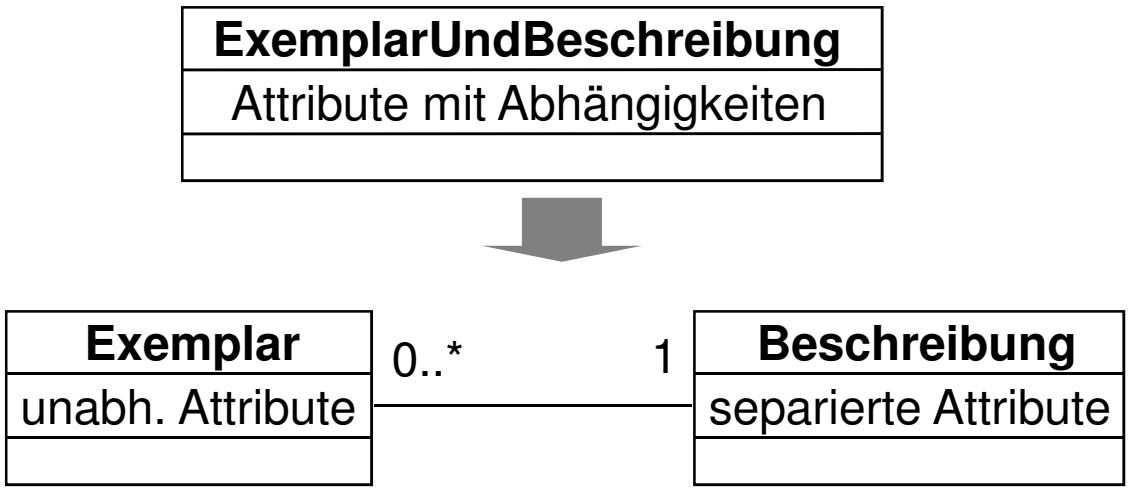
http://www.se-rwth.de/

Literatur:

• Balzert Band I, LE 13
• Martin Fowler: Analysis Patterns 1997
• Scott Ambler: Building Object-Oriented Applications That Work, SIGS Books 1998 (Kap. 4)

Muster: Exemplar und Beschreibung

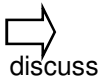
- Universelles Muster (nach Coad et al. 95)
- **Engl. Name:** Item-Item Description
- **Problem:** Manche Attribute nehmen bei vielen Instanzen immer wieder die gleichen Kombinationen von Werten an (Attribut-Abhängigkeit).
- **Lösung:** Einführung einer neuen Klasse und einer Aggregation.

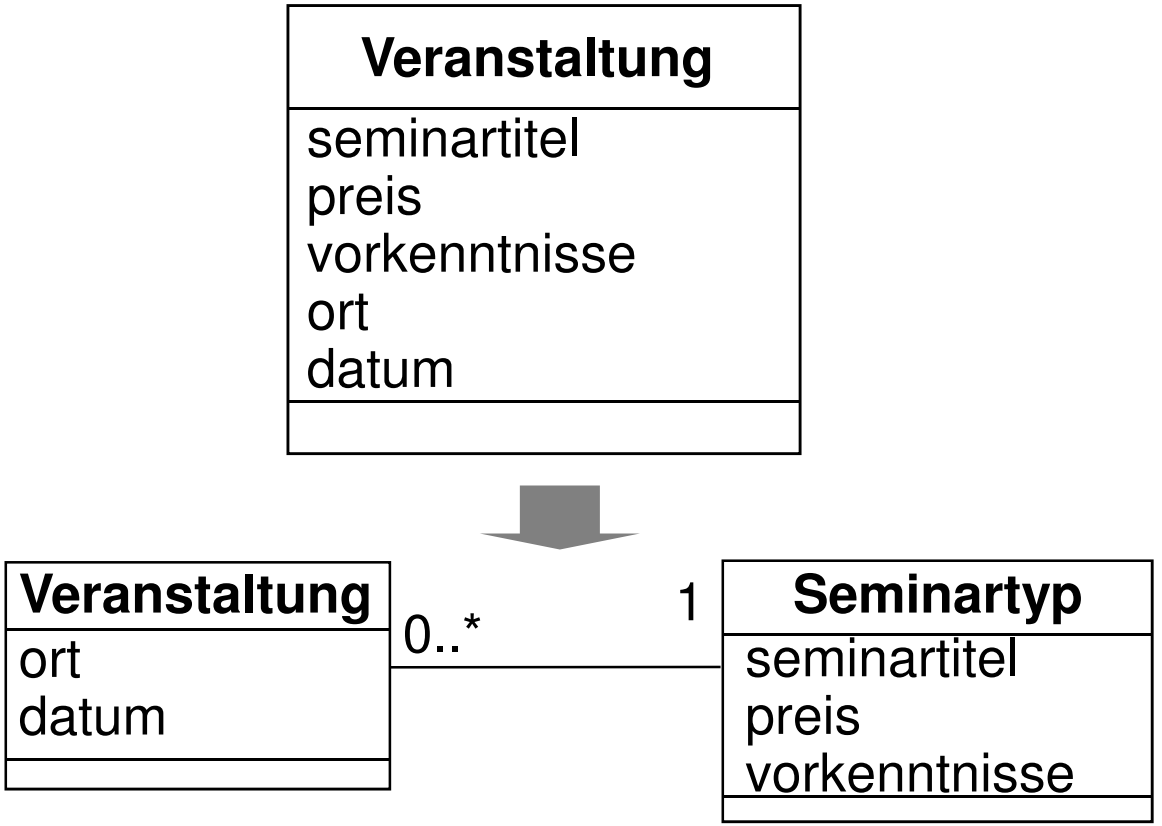


- vgl. dieses Muster mit Normalisierung von relationalen DB-Modellen

Transformation mit Mustern: Beispiel

- Typische Anwendungsform für Muster ("Refactoring"):
  - Finden verbesserungswürdiger Strukturen in Modellen
  - Ersetzen durch besser strukturierte Fassung





Wiederverwendung mit Musterkatalogen

- Ein ***Muster*** ist eine schematische Lösung für eine Klasse verwandter Probleme. Muster beschreiben *Erfahrungswissen*.
- Darstellung eines Musters
  - Name, evtl. Synonyme
  - Problem
    - Motivation, Anwendungsbereich
  - Lösung(en)
    - Struktur (Klassendiagramm)
    - Bestandteile (schematische Klassen- und Objektnamen)
    - Beschreibung, evtl. Abläufe, z.B. Sequenzdiagramm
  - Diskussion
    - Vor- und Nachteile, Abhängigkeiten, Einschränkungen
  - Beispielanwendungen
  - Verwandte Muster (Ähnlichkeiten)

Muster: Koordinator

- Universelles Muster (nach Heide Balzert 99, Balzert 96)
- **Problem:** Eine (zwei- oder mehrstellige) Assoziation besitzt Attribute, die zu keiner der beteiligten Klassen gehören.
- **Lösung:**
  Einführung einer eigenen Koordinator-Klasse

Koordinator: Beispiel

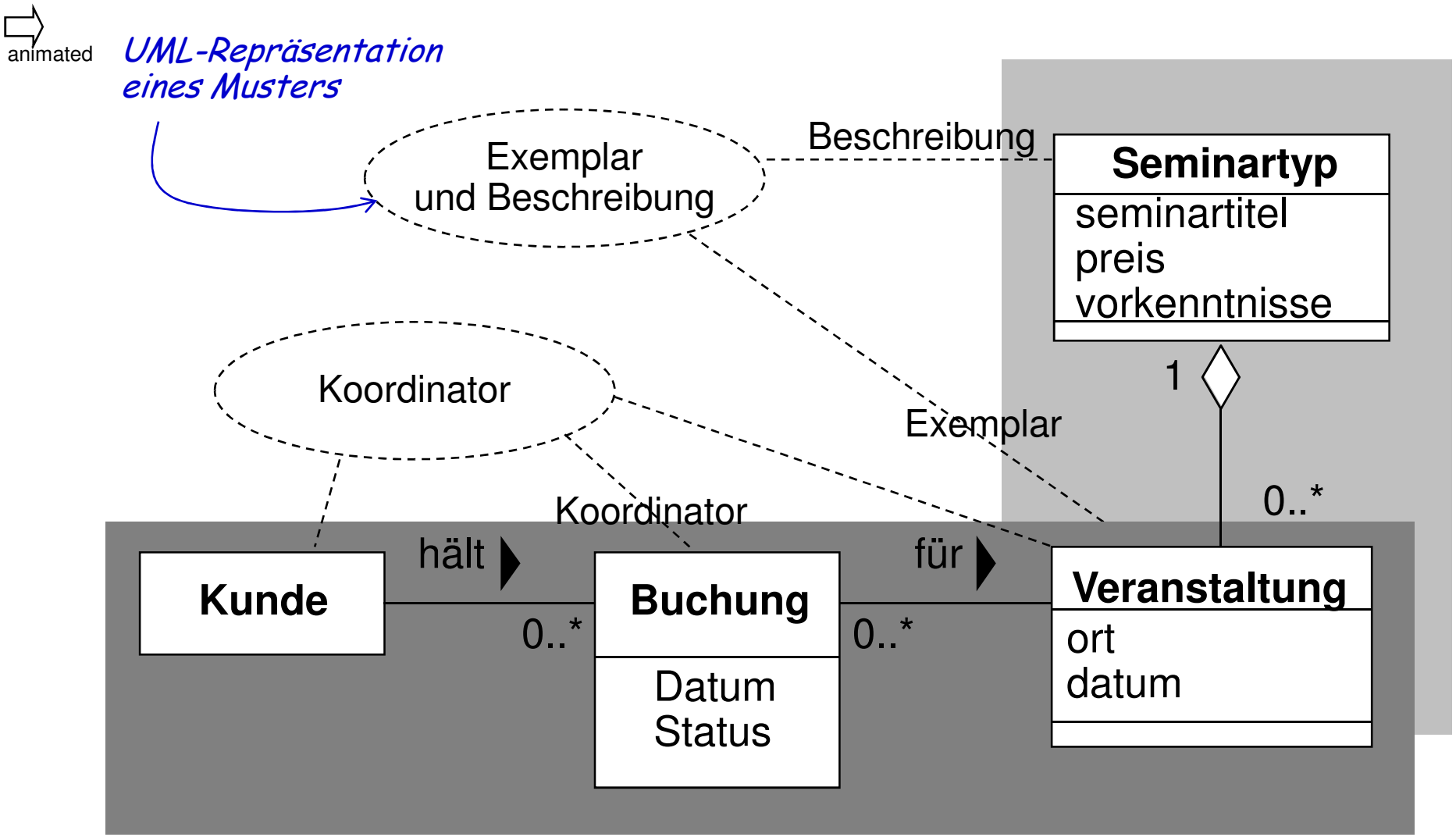

discuss

Buchungsdatum ?
Buchungsstatus ?

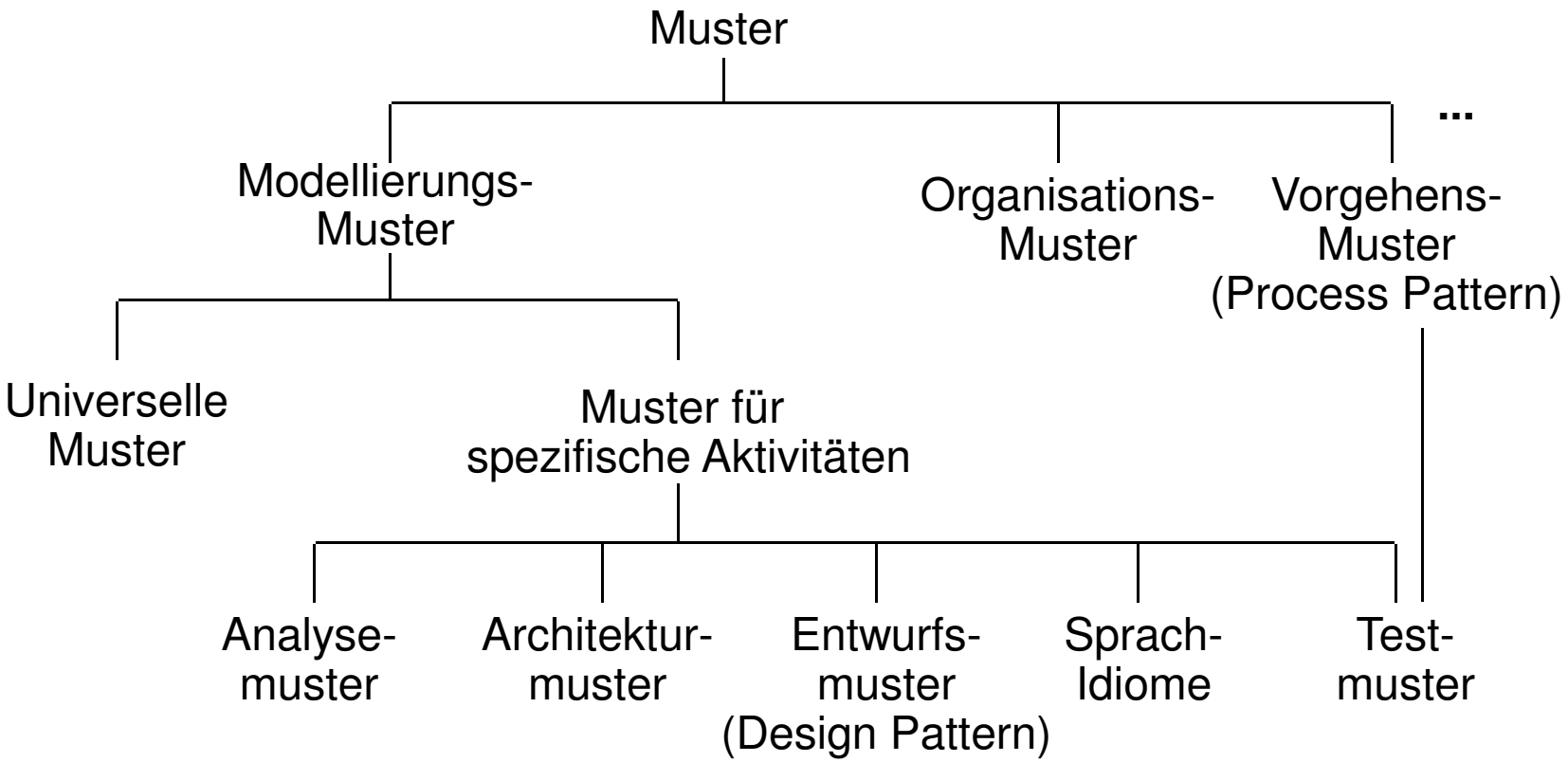
- ein wesentliches Merkmal dieses Musters ist die beidseitige Kardinalität „0-*“, die es verhindert Attribute der ein oder anderen Klasse zuzuordnen.
- Typisch für Koordinatorklassen ist, dass sie selbst nur wenige Attribute und Operationen besitzen, und mit mehreren Klassen in Verbindung stehen.

Kombination von Musteranwendungen

animated



Klassifikation von Mustern

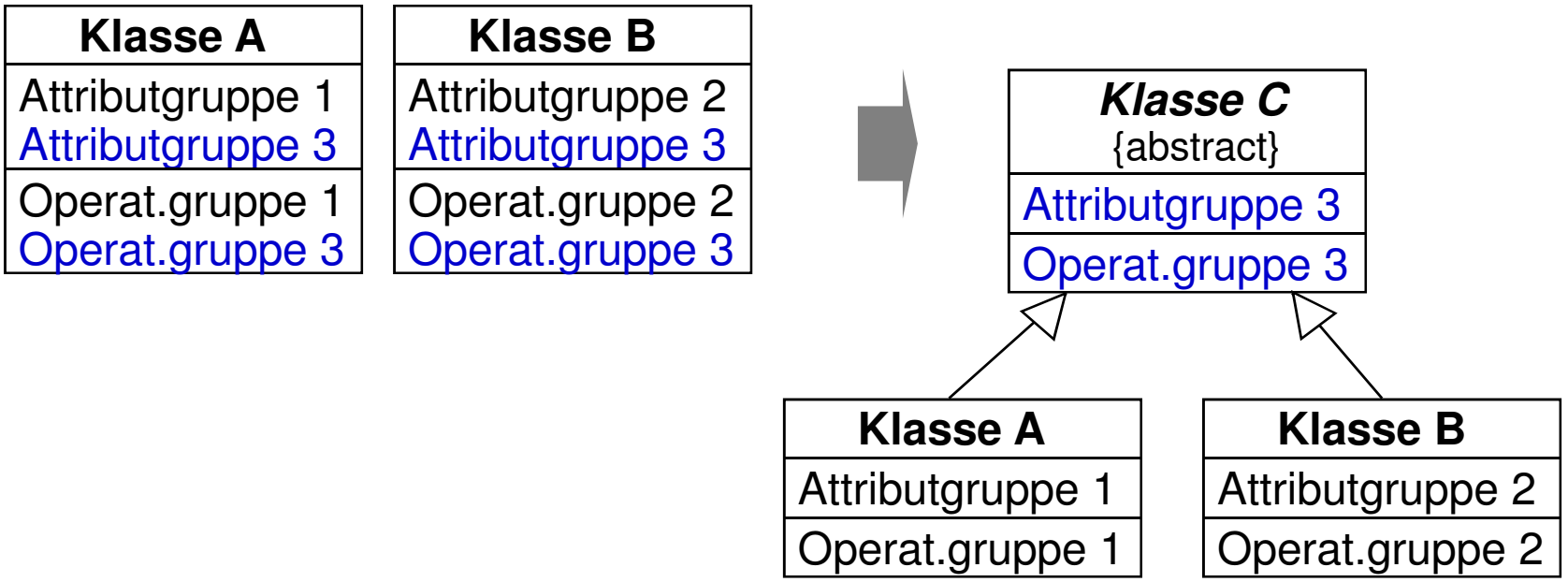


Weitere Unterteilung:

- Allgemein anwendbare Muster (z.B. Komposition)
- Domänenspezifische Muster (z.B. Kontostruktur im Finanzbereich)

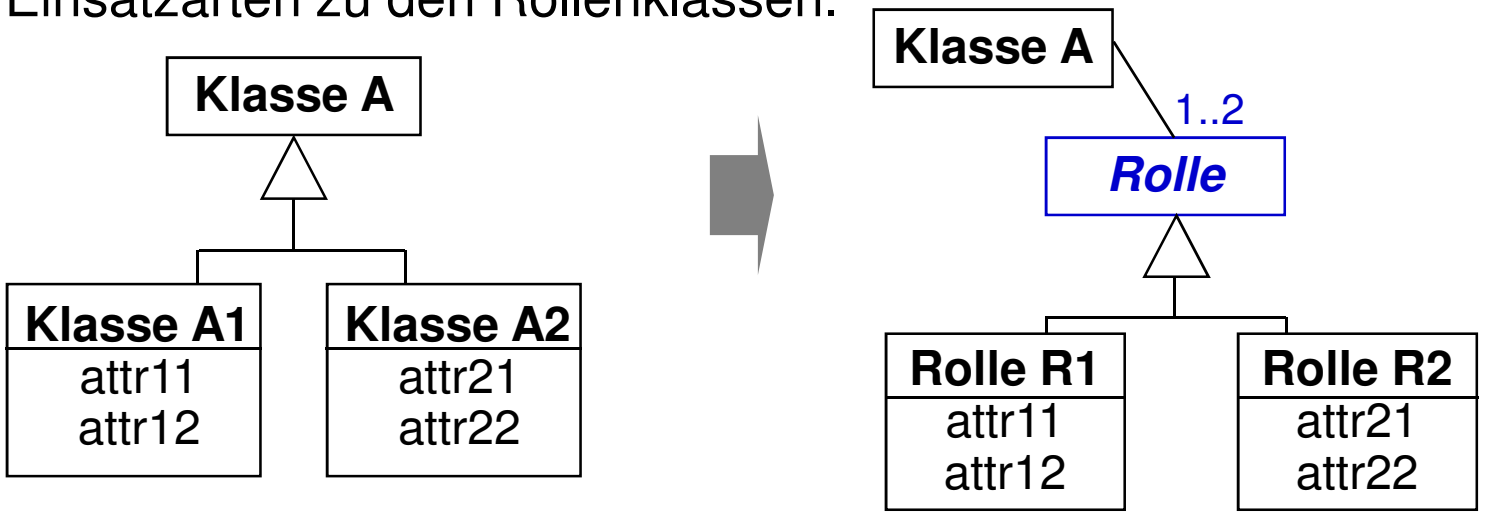
Muster: Abstrakte Oberklasse

- *Universelles Muster* (nach Balzert 96)
- **Problem:** Klassen enthalten Gruppen identischer Attribute und Operationen.
- **Lösung:** Separieren der identischen Bestandteile in einer abstrakten Oberklasse.



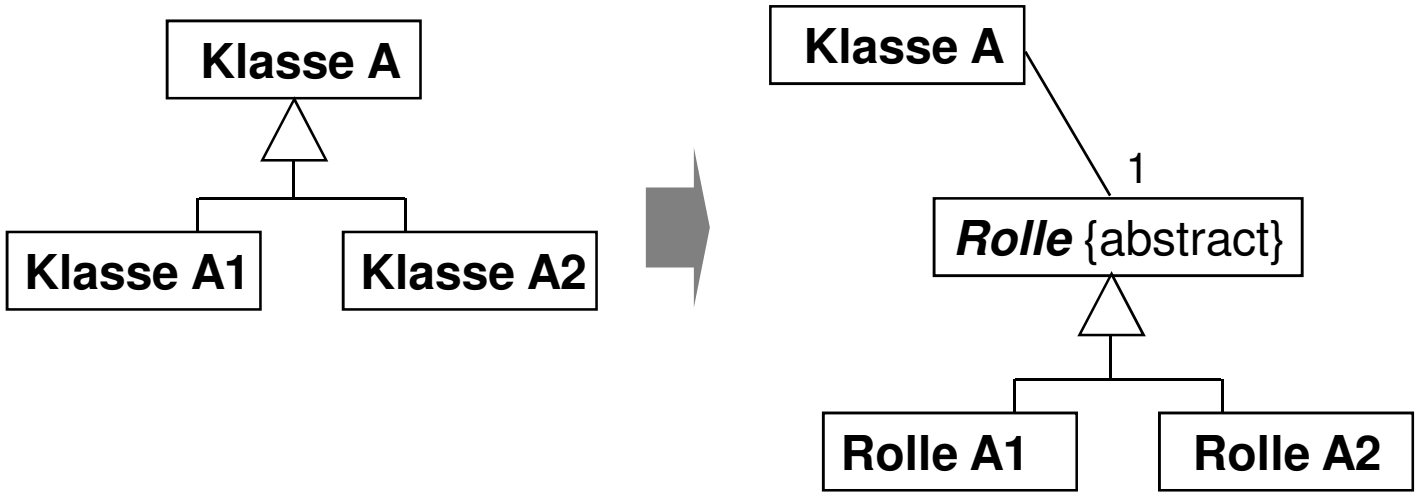
Muster: Rollen

- *Universelles Muster* (angelehnt an Heide Balzert 99)
- **Problem:** Es bestehen zwei oder mehr verschiedene Einsatzarten einer gegebenen Klasse. Unterklassenbildung würde "überlappende" Vererbung und damit Mehrfachzugehörigkeit von Objekten in Klassen erfordern.
- **Lösung:**
  Einführung einer Assoziation zu einer "Rollen"-Klasse mit Multiplizität entsprechend der Einsatzarten. Zuordnung von Attributen der Einsatzarten zu den Rollenklassen.



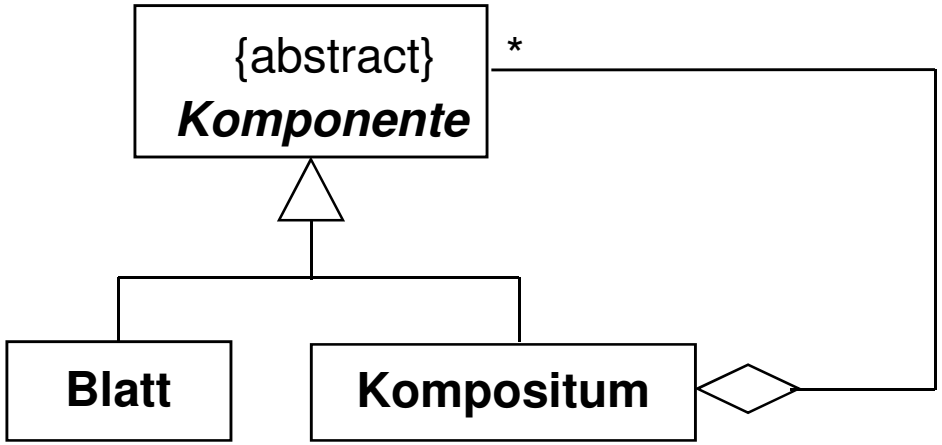
Muster: Wechselnde Rollen

- *Universelles Muster* (nach Heide Balzert 99)
- **Problem:** Ein Objekt einer Unterklasse A1 einer Oberklasse A kann in eine andere Unterklasse A2 von A wechseln.
- **Lösung:** Einführung einer abstrakten Rollenklasse.
- Variante des Musters „Rolle“ mit dynamischem Auswechseln des Rollen-Objekts



Muster: Komposition

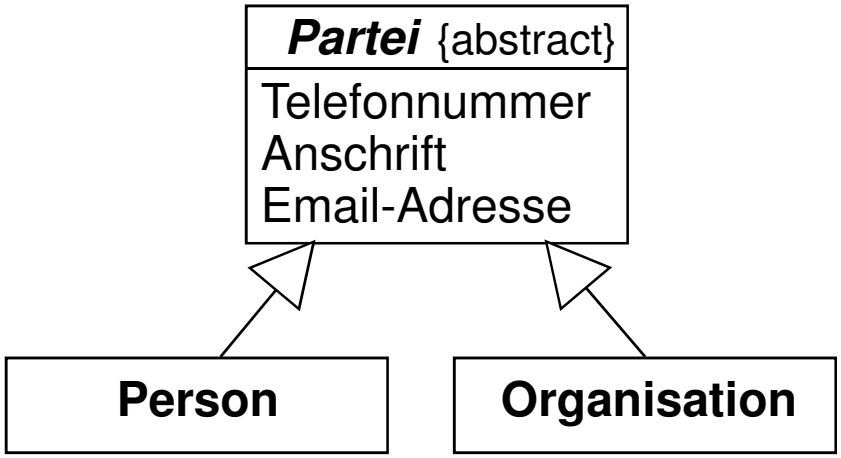
- *Universelles Muster* (nach Gamma et al. 95)
- **Andere Namen:** *Composite* (engl.), Stückliste (Heide Balzert)
- **Problem:** Tiefe und evtl. veränderliche Hierarchie von Aggregationen
- **Lösung:** Abstrakte Oberklasse mit verschiedenen Blattklassen sowie einer Kompositklasse als Unterklassen.



Komposition ist ein sehr häufig angewandtes Muster:
Bäume, Teile-Ganzes-Hierarchien etc. sind damit realisierbar.

Partei (Party)

- *Spezifisches Analysemuster* (nach Fowler 97)
- **Problem:** Organisationen und Personen spielen ähnliche Rollen gegenüber dem System
- **Lösung:** Abstrakte Oberklasse "Partei".
- **Regel:** Überprüfe bei Personen und Organisationen, ob es sich nicht um allgemeinere "Parteien" handelt.



"Partei" vs. "Abstrakte Oberklasse"

- Das Muster "Partei" ist ein Spezialfall des Musters "Abstrakte Oberklasse".
- Gründe für separate Darstellung:
  - Anpassung an ein Fachgebiet
  - Fachterminologie (!)
  - Geringerer Abstraktionsgrad: mehr Detailliertheit

- Spezifische Analysemuster sind meist Anwendungen und Zusammensetzungen universeller Muster.
- Ganz wesentlich ist hier auch die Vokabular-Bildung.
  - Wenn Muster allgemein eingeführt sind, so reicht die Erwähnung des Begriffs „Party“ statt eine detaillierte Spezifikation.
    Dies erleichtert die Analyse immens!

Zusammenfassung
6.6. Muster in der Objektorientierten Analyse

- In der Analyse werden eingesetzt:
  - Universelle Muster
  - Spezifische Analysemuster
    - mehr oder minder fachgebietsübergreifend
    - fachgebietsspezifisch
- Universelle Muster:
  - "Leitfaden" zur Modellierung
  - Bausteine für andere Muster
- Spezifische Analysemuster:
  - Nur wenige Kataloge publiziert
    - Fowler: Katalog für betriebswirtschaftliche Anwendungen
  - viele Firmen- und projektspezifische Kataloge