

Objektorientiert entwerfen und implementieren (326)

Objektorientiert entwerfen und implementieren (326)

Grafisches Konzept: dezember und juli, Kommunikation und Design (www.dezemberundjuli.ch)

Druck: Mikro+Repro AG, Baden

Auflage: 1. Ausgabe 2006

Ausgabe: N0046

Sprache: DE

Alle Rechte, insbesondere die Übersetzung in fremde Sprachen, vorbehalten. Das Werk und seine Teile sind urheberrechtlich geschützt. Jede Verwertung in anderen als den gesetzlich zugelassenen Fällen bedarf der vorgängigen schriftlichen Zustimmung von der Stiftung Wirtschaftsinformatikschule Schweiz bzw. des Inhabers der Lizenzrechte.

Mit freundlicher Genehmigung:

Copyright © 2006, Universität Stuttgart Institut für Automatisierungs- und Softwaretechnik (IAS)

Inhaltsverzeichnis

Kapitel 3	Statische Konzepte in der objektorientierten Analyse (144)	
	3.1 Statische vs. dynamische Konzepte	146
	3.2 Assoziation	150
	3.3 Aggregation und Komposition	174
	3.4 Vererbung	186
	3.5 Paket	201
	3.6 Erweiterungsmechanismen der UML	207
	3.7 Zusammenfassung	212
Kapitel 4	Dynamische Konzepte in der objektorientierten Analyse (216)	
	4.1 Anwendungsfall	218
	4.2 Botschaft	232
	4.3 Szenario	237
	4.4 Zustandsautomat	254
	4.5 Zusammenfassung objektorientierte Konzepte	272
	4.6 Zusammenfassung	279
Kapitel 5	Analyseprozess und Analysemuster (282)	
	5.1 Analyseprozess	284
	5.2 CRC-Karten	295
	5.3 Analysemuster	308
	5.4 Checklisten	328
	5.5 Beispiel „Waschtrockner“	341
	5.6 Zusammenfassung	352
Kapitel 6	Konzepte und Notationen des objektorientierten Entwurfs (352)	
	6.1 Von der Analyse zum Entwurf	354
	6.2 Konzepte des objektorientierten Entwurfs	365
	6.3 Modellierung von Programmabläufen	384
	6.4 Architekturentwurf	391
	6.5 Entwurfsregeln und -heuristiken	399
	6.6 Zusammenfassung	402
Kapitel 7	Entwurfsmuster und Frameworks (405)	
	7.1 Entwurfsmuster, Frameworks, Klassenbibliotheken	407
	7.2 Fabrikmethode-Muster	416
	7.3 Singleton-Muster	421
	7.4 Beobachter-Muster	425
	7.5 Anwendungsbeispiel	429
	7.6 Zusammenfassung	436

Die angegebenen Zahlen beziehen sich auf die Foliennummern.

Über dieses Lehrmittel

Inhalt und Aufbau dieses Lehrmittels

Das Modul 326 soll die Kompetenz:

**Eine objektorientierte Analyse (OOA) in ein objektorientiertes Design (OOD)
überführen, implementieren, testen und dokumentieren**

vermitteln.

Die vorliegende Foliensammlung basiert auf dem Kurs Einführung in die Informatik III der Universität Stuttgart.

Siehe dazu: <http://www.ias.uni-stuttgart.de/info3/lehrmaterialien/umdruck.html>

Das vollständige Vorlesungsskript umfasst 7 Kapitel, wobei für das vorliegende Modul 326 die Kapitel 3 bis 7 ausgewählt wurden, wobei der eigentliche Schwerpunkt objektorientierte Analyse (OOA) im Kapitel 4 und das objektorientierte Design (OOD) im Kapitel 5 behandelt wird.

Einleitung

Kapitel 1 Einführung in die Objektorientierung

Kapitel 2 Basiskonzepte und Notationen der Objektorientierung

Inhalt des vorliegenden Lehrmittels:

Kapitel 3 Statische Konzepte in der objektorientierten Analyse

Kapitel 4 Dynamische Konzepte in der objektorientierten Analyse

Kapitel 5 Analyseprozess und Analysemuster

Kapitel 6 Konzepte und Notationen des objektorientierten Entwurfs

Kapitel 7 Entwurfsmuster und Frameworks

Kapitel 8 Implementierung objektorientierte Konzepte

Kapitel 9 Komponentenbasierte Softwareentwicklung

Kapitel 10 UML 2.0

Handlungsziele

Gemäss I-CH wurden diesem Modul die folgende Kompetenz und die aufgeführten Handlungsziele zugeordnet:

Modulnummer	326
Titel	Objektorientiert entwerfen und implementieren
Kompetenz	Eine objektorientierte Analyse (OOA) in ein objektorientiertes Design (OOD) überführen, implementieren, testen und dokumentieren.
Handlungsziele	<p>1. Anforderung analysieren, daraus die fachlichen Klassen festlegen und in ein entsprechendes Design überführen.</p> <p>2. Dynamische und statische Strukturen zwischen Objekten resp. Klassen mittels Unified Modeling Language, UML (Klassen-/Sequenzdiagramme) darstellen.</p> <p>3. Das objektorientiert erstellte Design implementieren.</p> <p>4. Erfüllung der Anforderung prüfen.(Systemtest).</p> <p>5. Klassen- und Systemdokumentation vervollständigen.</p>
Kompetenzfeld	Application Engineering
Objekt	Applikation mit 3-5 fachlichen Klassen (z.B. Bibliothek, Wertschriftendepot, Börsenticker usw.).
Niveau	3
Voraussetzungen	Objektorientiert implementieren (Modul 226)
Anzahl Lektionen	40
Anerkennung	Eidg. Fähigkeitszeugnis Informatiker/Informatikerin
Modulversion	1.1
MBK Release	R3
Harmonisiert am	16.09.2005

Den gegebenen 5 Handlungszielen sind Handlungsnotwendige Kenntnisse zugeordnet (siehe Tabelle nächste Seite)

Handlungsnotwendige Kenntnisse

Handlungsziel	Handlungsnotwendige Kenntnisse
1.	<ol style="list-style-type: none">1. Kennt die Elemente eines Use Case Model und kann an Beispielen aus dem Alltag erläutern, welche Sachverhalte damit abgebildet werden können.2. Kennt Vorgehensprinzipien zur Klassenfindung (Nomenverfahren, CRC-Karte usw.) und kann die Bedeutung eines iterativen Design-Prozesses aufzeigen.3. Kennt Kriterien, nach denen Objekte der realen oder Vorstellungswelt zueinander in Beziehung gebracht werden können.4. Kann aufzeigen, wie und warum die Lösung auf mehrere Pakete verteilt wird.
2.	<ol style="list-style-type: none">1. Kann Sachverhalte durch UML Diagramme (Klassen- und Sequenzdiagramm) abbilden.
3.	<ol style="list-style-type: none">1. Kann erläutern, wie ein Klassenmodell mit einer objektorientierten Programmiersprache umgesetzt werden kann.2. Kann aufzeigen, welche Vorteile ein iteratives Vorgehen für die Implementation des Designs aufweist.3. Kann darlegen, warum der Einsatz wiederverwendbarer Klassen und Konzepte (Idee der Pattern) die Effizienz für Design und Implementation steigert.4. Kennt die grundlegenden Funktion eines CASE Tools und kann erläutern, mit welchen Funktionen das objektorientierte Design unterstützt wird.
4.	<ol style="list-style-type: none">1. Kann auf Grund der Use Cases Testfälle bestimmen und ausarbeiten.2. Kennt die grundlegenden Schritte die bei einem Systemtest durchlaufen werden müssen und kann aufzeigen, welchen Beitrag diese zu einem qualitativ guten Ergebnis leisten.
5.	<ol style="list-style-type: none">1. Kennt die Struktur einer Systemdokumentation und kann ihre Bedeutung für Wartung und Nachvollziehbarkeit darlegen.

Kapitel 3 Statische Konzepte in der objektorientierten Analyse

<u>3.1 Statische vs. dynamische Konzepte</u>	<u>146</u>
<u>3.2 Assoziation</u>	<u>150</u>
<u>3.3 Aggregation und Komposition</u>	<u>174</u>
<u>3.4 Vererbung</u>	<u>186</u>
<u>3.5 Paket</u>	<u>201</u>
<u>3.6 Erweiterungsmechanismen der UML</u>	<u>207</u>
<u>3.7 Zusammenfassung</u>	<u>212</u>

Kapitel 3 Statische Konzepte in der objektorientierten Analyse

3.1 Statische vs. dynamische Konzepte

3.2 Assoziation

3.3 Aggregation und Komposition

3.4 Vererbung

3.5 Paket

3.6 Erweiterungsmechanismen der UML (zum Selbststudium)

3.7 Zusammenfassung

Lernziele

- Erklären können, was eine Assoziation ist
- Erklären können, was eine assoziative Klasse und eine qualifizierte Assoziation sind
- Erklären können, was Aggregation und Komposition bedeuten
- Erklären können, was Vererbung ist
- Erklären können, was ein Paket ist
- UML-Notation für Assoziation, Vererbung und Paket anwenden können
- Assoziationen und Vererbung in einem Text identifizieren und darstellen können
- Klassen zu Paketen gruppieren können



Kapitel 3 Statische Konzepte in der objektorientierten Analyse

3.1 Statische vs. dynamische Konzepte

3.2 Assoziation

3.3 Aggregation und Komposition

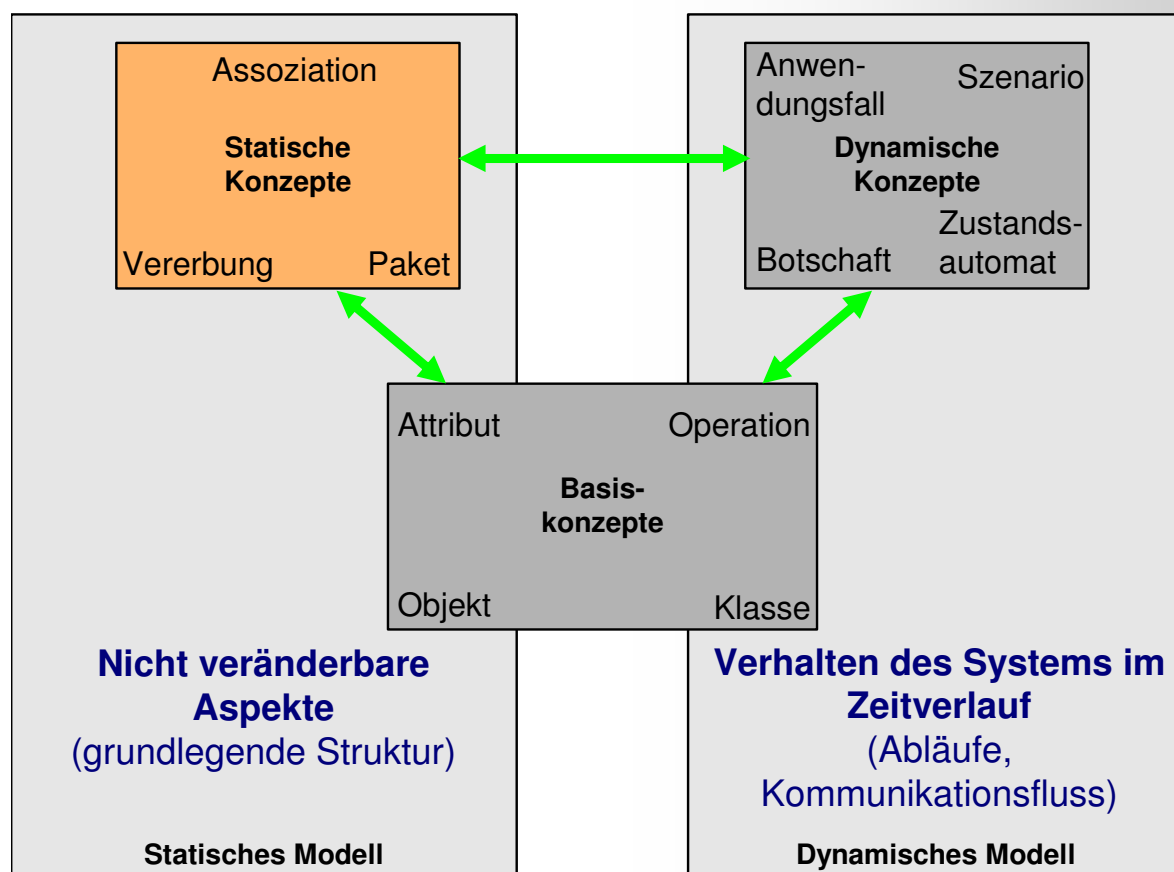
3.4 Vererbung

3.5 Paket

3.6 Erweiterungsmechanismen der UML (zum Selbststudium)

3.7 Zusammenfassung

OOA-Modell (Analysemodell):



OOA-Modell (Analysemodell):

- Das **statische** Modell zeigt die Struktur
 - **Klassen** beschreiben die Elemente des Fachkonzepts
 - **Assoziationen** beschreiben die Beziehungen zwischen den Klassen
 - **Vererbungsstrukturen** beschreiben die Verallgemeinerung von Klassen
 - **Attribute** beschreiben Eigenschaften der Klassen (Daten des Systems)
 - **Paketen** teilen das System in Teilsysteme auf
- Das **dynamische** Modell zeigt Funktionsabläufe
 - **Anwendungsfälle** beschreiben die durchzuführenden Aufgaben auf einem sehr hohen Abstraktionsniveau
 - **Szenarios** zeigen, wie Objekte miteinander kommunizieren, um eine bestimmte Aufgabe zu erledigen
 - **Zustandsautomaten** beschreiben die Reaktionen eines Objekts auf verschiedene Ereignisse



Frage zu 3.1 : Statisches oder dynamisches Modell

- Ordnen Sie die folgenden Aussagen dem statischen oder dem dynamischen Modell zu

Aussage	Statisches Modell	Dynamisches Modell
Beschreibt das Verhalten des zu entwickelnden Softwaresystems		<input checked="" type="checkbox"/>
Bildet den stabilen Kern des objektorientierten Modells	<input checked="" type="checkbox"/>	
Beschreibt die Beziehungen zwischen Klassen (bzw. ihren Objekten)	<input checked="" type="checkbox"/>	
Zeigt, wie Objekte miteinander kommunizieren, um eine bestimmte Aufgabe zu erledigen		<input checked="" type="checkbox"/>
Modelliert die Struktur des Softwaresystems	<input checked="" type="checkbox"/>	
Beschreibt die Reaktionen eines Objekts auf verschiedene Ereignisse		<input checked="" type="checkbox"/>
Enthält die Aufteilung des Systems in Teilsysteme	<input checked="" type="checkbox"/>	



Kapitel 3 Statische Konzepte in der objektorientierten Analyse

3.1 Statische vs. dynamische Konzepte

3.2 Assoziation

3.3 Aggregation und Komposition

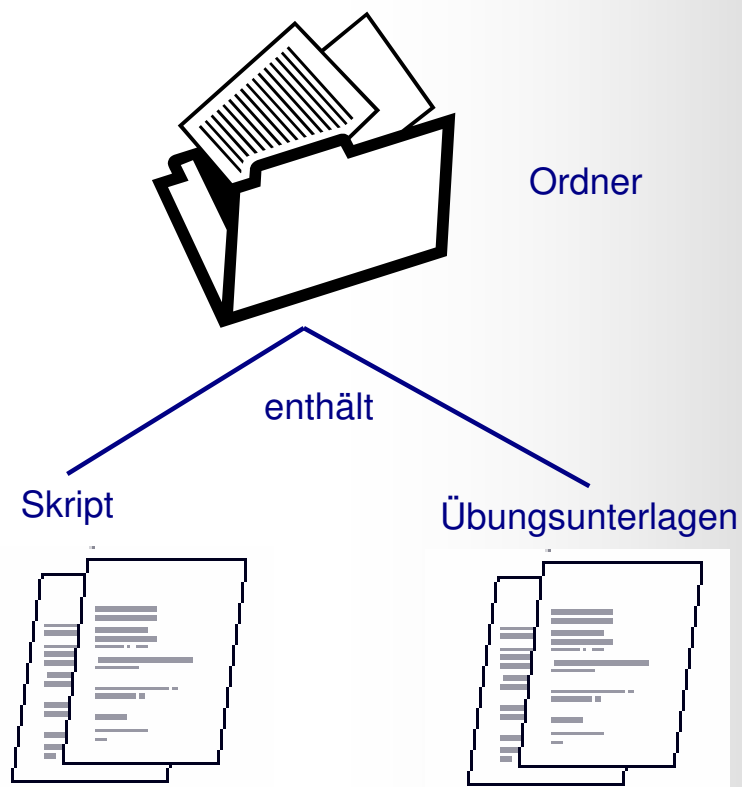
3.4 Vererbung

3.5 Paket

3.6 Erweiterungsmechanismen der UML (zum Selbststudium)

3.7 Zusammenfassung

Beziehungen zwischen Objekten (Tafelanschrieb)



Was ist eine Assoziation?

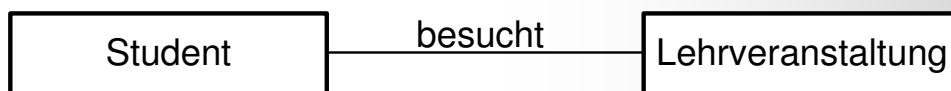
Definition:

Eine **Assoziation** modelliert Verbindungen zwischen Objekten einer oder mehrerer Klassen.

Eine Assoziation modelliert stets Verbindungen zwischen Objekten, nicht zwischen Klassen!

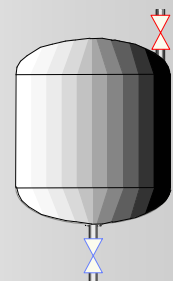
Beispiel:

Objekte der Klasse Student (z.B. Paul, Susi oder Peter) haben eine Verbindung zu Objekten der Klasse Lehrveranstaltung (z.B. Einführung in die Informatik III am Dienstag)



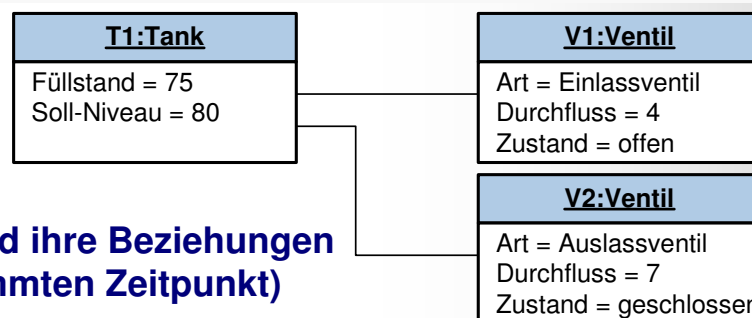
Beispiel: Assoziation zwischen Tank und Ventil

– Klassendiagramm



Zeigt Klassen und ihre Beziehungen (statische Modellelemente)

– Objektdiagramm



Zeigt Objekte und ihre Beziehungen (zu einem bestimmten Zeitpunkt)

⇒ Die Menge aller Verbindungen wird als Assoziation zwischen den Objekten der Klassen Tank und Ventil bezeichnet.

Eigenschaften von Assoziationen

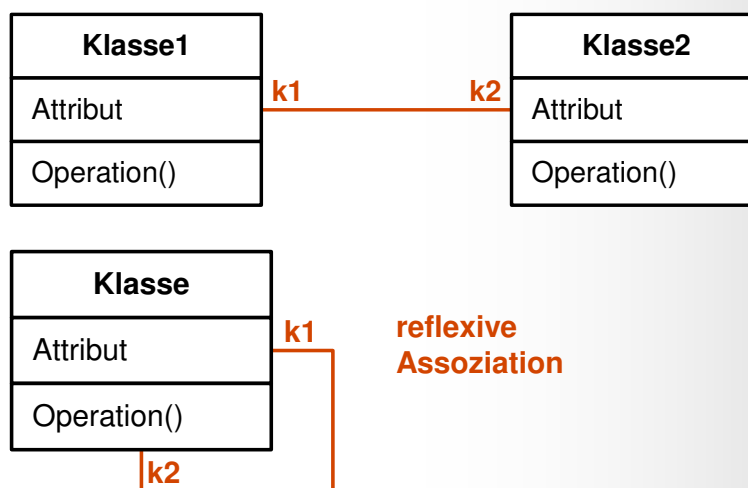
- Es gibt binäre (zwischen zwei Objekten) und höherwertige Assoziationen
 - Eine reflexive Assoziation besteht zwischen Objekten derselben Klasse
 - Eine Assoziation hat eine Richtung (Navigierbarkeit)
 - "Welches Objekt ist über die Beziehung informiert?"
 - unidirektional
 - bidirektional
 - Assoziationen sind in der Systemanalyse inhärent bidirektional
- Objekte "kennen" sich gegenseitig**
- 3 Arten von Assoziation
 - einfache Assoziation
 - Aggregation
 - Komposition

UML Notation einer Assoziation

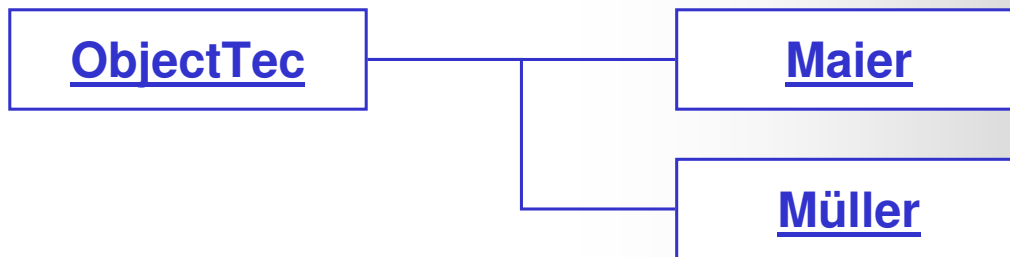
- Binäre Assoziation
 - Linie zwischen einer oder zwei Klassen
 - Assoziationsname
 - An jedem Ende der Linie steht die Wertigkeit bzw. Kardinalität (*multiplicity*)

"Wie viele Objekte kann ein bestimmtes Objekt kennen"

 - An jedem Ende kann ein Rollenname stehen



Assoziation im Objektdiagramm (Tafelanschrift)



- Frage: Wie sieht das Klassendiagramm zu diesem Objektdiagramm aus?











Name einer Assoziation

- Beschreibt Semantik (Bedeutung) der Assoziation
- Beschreibt meistens nur eine Richtung der Assoziation
- Ein schwarzes Dreieck gibt die Leserichtung an
- Name darf fehlen, wenn die Bedeutung der Assoziation offensichtlich ist

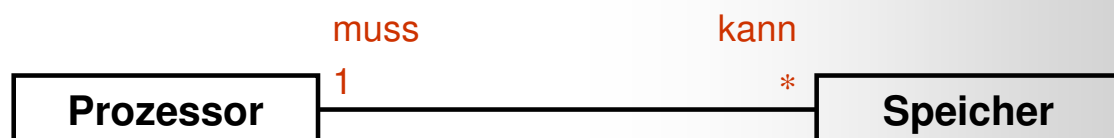


UML Notation der Kardinalität (Anzahl der Elemente)

1		genau 1
0..1		0 bis 1
*		0 bis viele
3..*		3 bis viele
0..2		0 bis 2
2		genau 2
2, 4, 6		2, 4 oder 6
1..5, 8, 10..*		nicht 6, 7 oder 9

Bedeutung der Kardinalität

- Kann-Assoziation
 - Untergrenze: Kardinalität 0
- Muss-Assoziation
 - Untergrenze: Kardinalität 1 oder größer



- ⇒ Prozessor kann mehrere Speicher besitzen
- ⇒ Es gibt Prozessoren, die keinen Speicher besitzen
- ⇒ Jeder Speicher gehört genau zu einem Prozessor

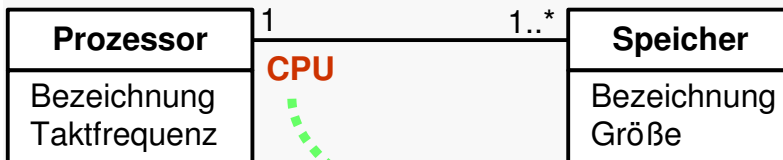


- ⇒ Jeder Prozessor muss mindestens einen Speicher besitzen

Rollenname (1)

- Beschreibt Bedeutung eines Objekts in einer Assoziation
- Binäre Assoziationen besitzen maximal zwei Rollen
- Wird an das Ende der Assoziation bei der Klasse geschrieben, deren Bedeutung in der Assoziation die Rolle beschreibt

- Beispiel für Klassendiagramm



- Beispiel für Implementierung

```

class Speicher
{
    Prozessor CPU;
    . . .
}
  
```

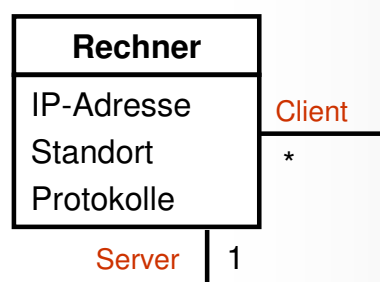
→ Rollennamen tragen zur Verständlichkeit des Modells mehr bei als der Assoziationsname

Rollenname (2)

- Rollenname ist **nicht** optional ...
 - wenn zwischen zwei Klassen mehr als eine Assoziation besteht

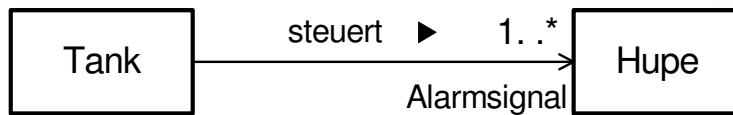


- bei reflexiven Assoziationen



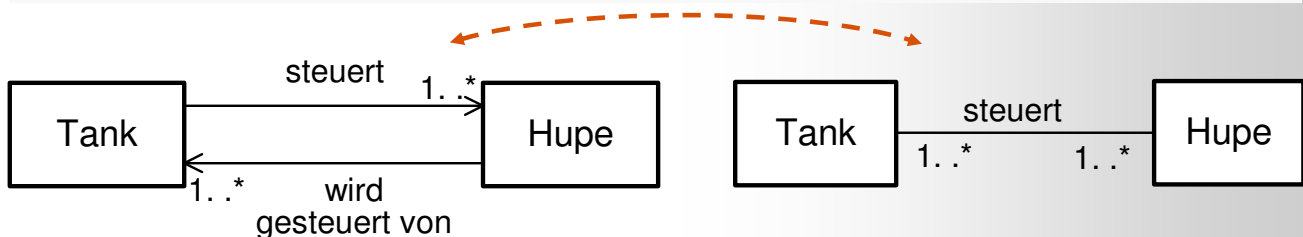
Gerichtete Assoziation

- Nur ein Objekt ist über Beziehung informiert (unidirektionale Navigierbarkeit)
- Nur in die Navigationsrichtung können Botschaften geschickt werden.
- Darstellung durch geöffnete Pfeilspitze



⇒ Einer Hupe ist nicht bekannt, von welchen Tanks sie als Alarmsignal verwendet wird.

- Jede bidirektionale Assoziation kann durch zwei unidirektionale Assoziationen ausgedrückt werden

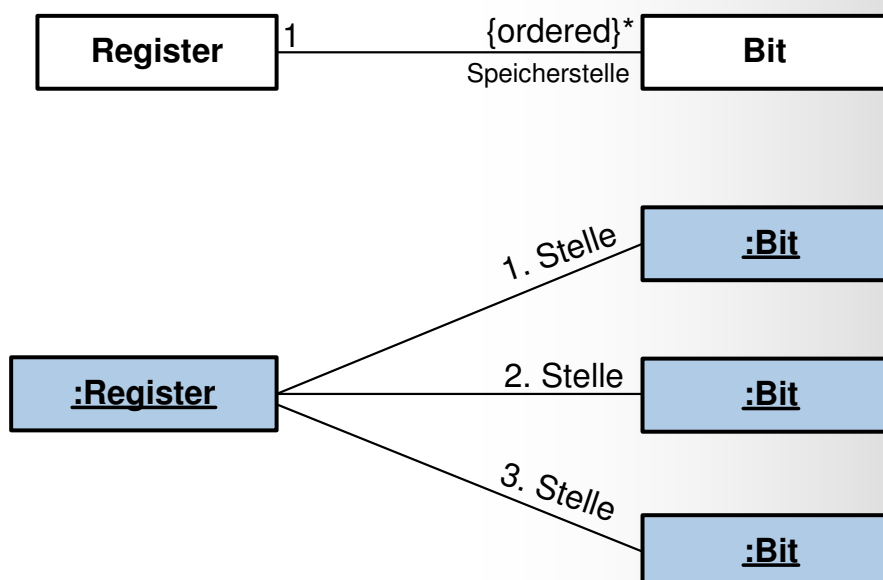


⇒ Navigierbarkeit in der Analyse nur in Ausnahmefällen festlegen

Geordnete Assoziation

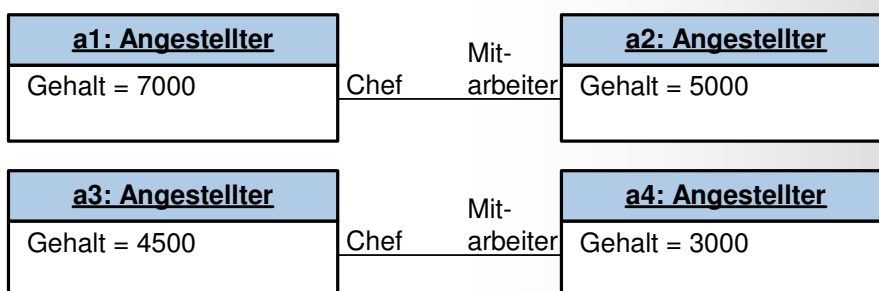
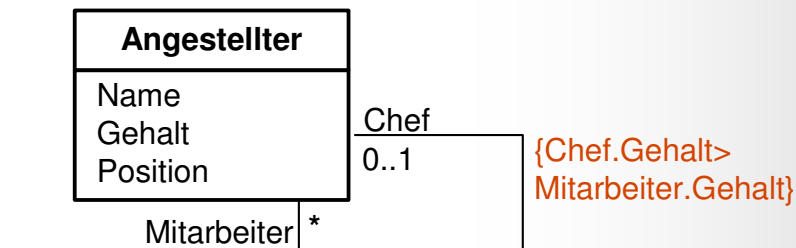
- Zeigt an, dass die Menge der Objektverbindungen geordnet ist
- Möglich bei Kardinalität größer eins
- Kennzeichnung der Ordnung durch das Schlüsselwort **{ordered}**

Keine Aussage über Definition der Ordnung (z.B. zeitlich, alphabetisch)



Restriktion (constraint) einer Assoziation

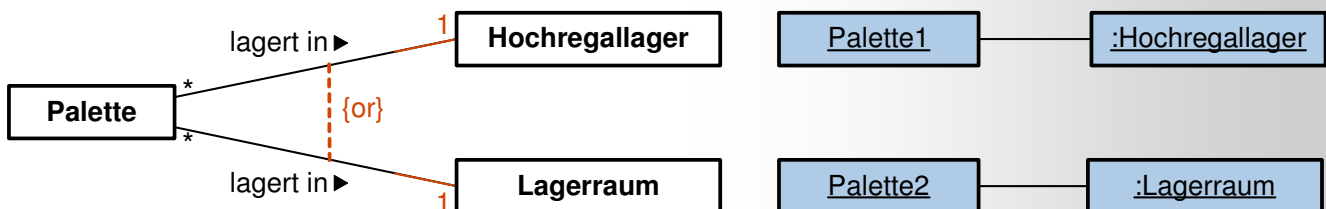
- Bedingung, die stets erfüllt sein muss
- Einschränkung der möglichen Inhalte eines Modellelements



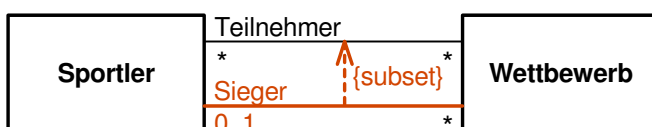
or-Restriktion einer Assoziation

- Zu jedem beliebigen Zeitpunkt kann nur eine von mehreren möglichen Assoziationen gelten
- **or**- Restriktion kann sich auf mehr als zwei Assoziationen beziehen
- Wichtig: Einbezogene Klassen müssen unterschiedliche Rollennamen haben

Fehlt der Rollename, gilt: Name der Klasse = Rollename

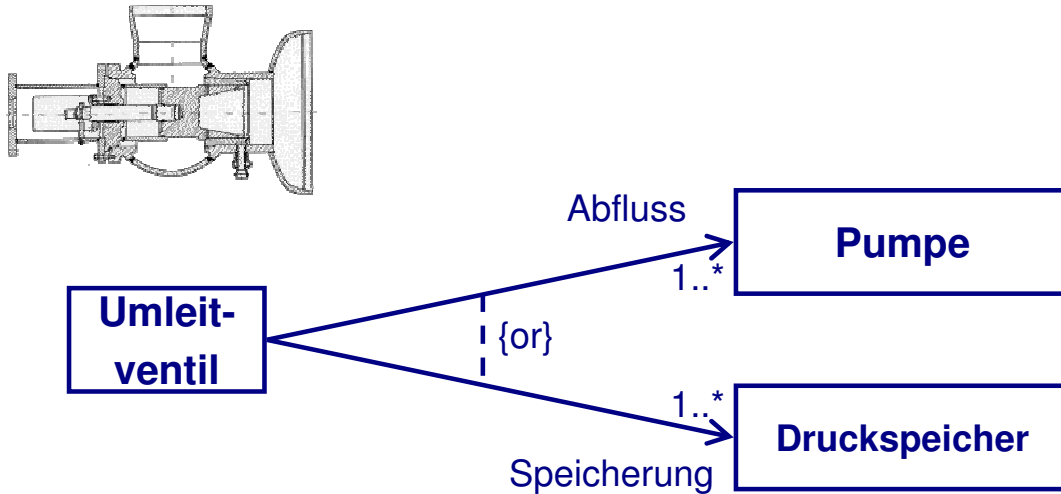


subset-Restriktion einer Assoziation



⇒ siehe Kap. 3.6
Erweiterungsmechanismen

Beispiel für or-Restriktion: Umleitventil (Tafelanschrieb)

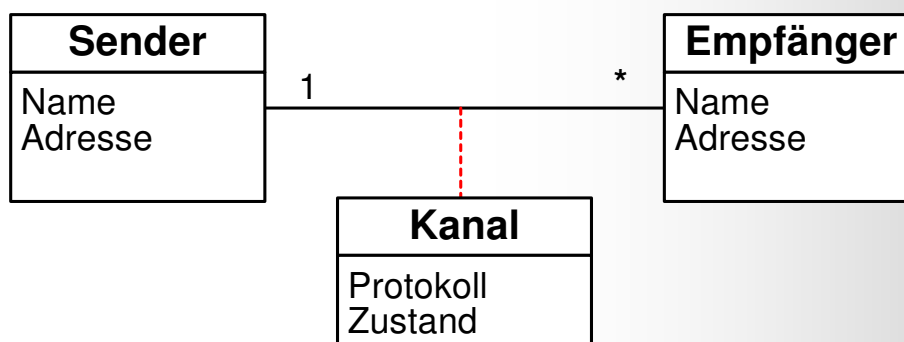


– Java Code

```
class Umleitventil {
    Pumpe Abfluss;
    Druckspeicher Speicherung;
    ...
}
```

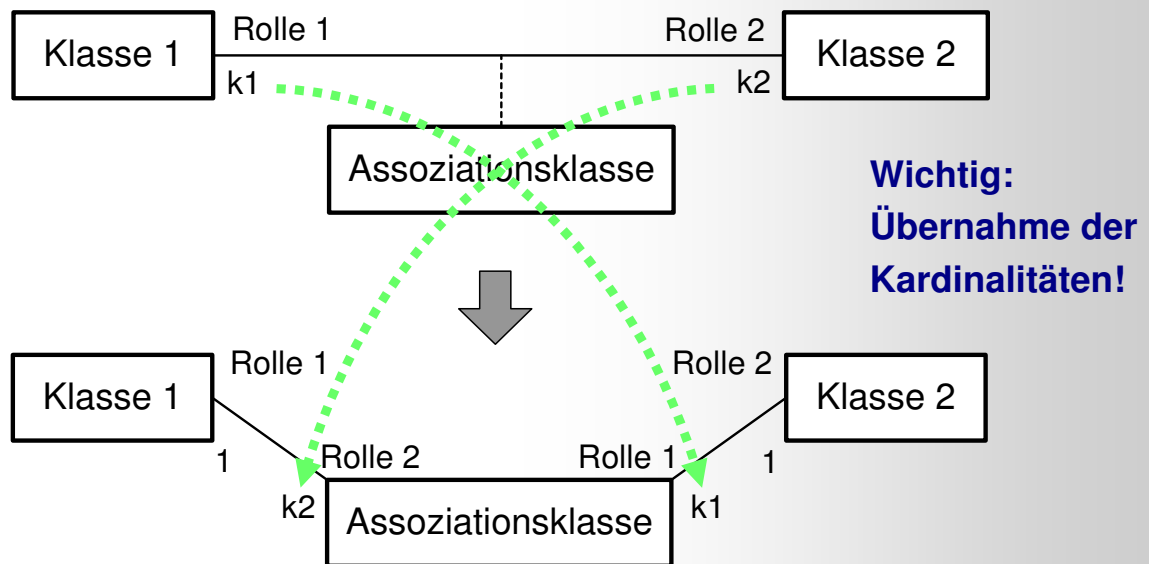
Assoziative Klasse (association class)

- Assoziationen können zusätzlich die Eigenschaften einer Klasse besitzen
- Darstellung mit Klassensymbol und gestrichelter Linie
- Name der Assoziation und der Assoziationsklasse sind stets identisch



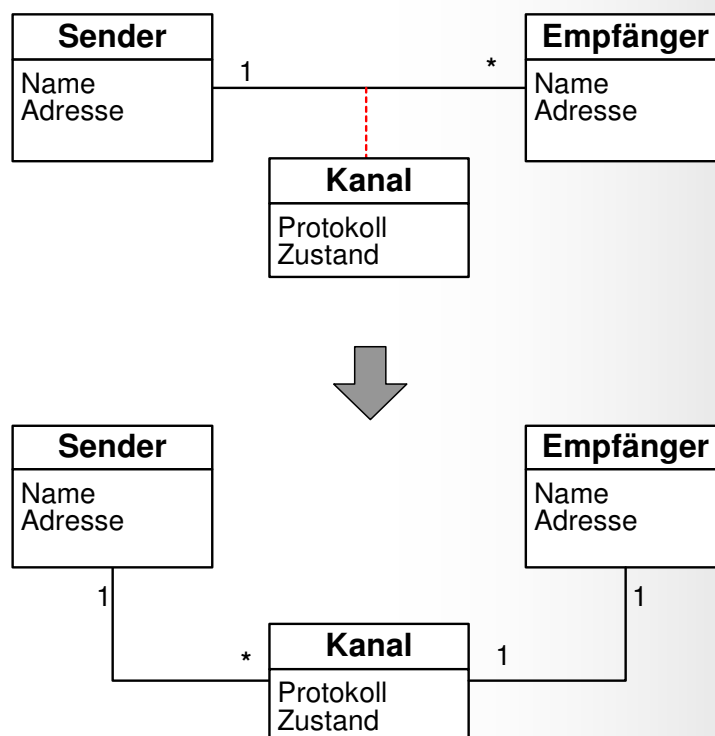
Verwendung von assoziativen Klassen (1)

- Erforderlich für Attribute und Operationen, die weder der einen noch der anderen Klasse zugeordnet werden können, sondern Eigenschaft der Beziehung selbst sind.
- Werden nur in der Analyse verwendet
- Im Entwurf Umwandlung in eine richtige Klasse



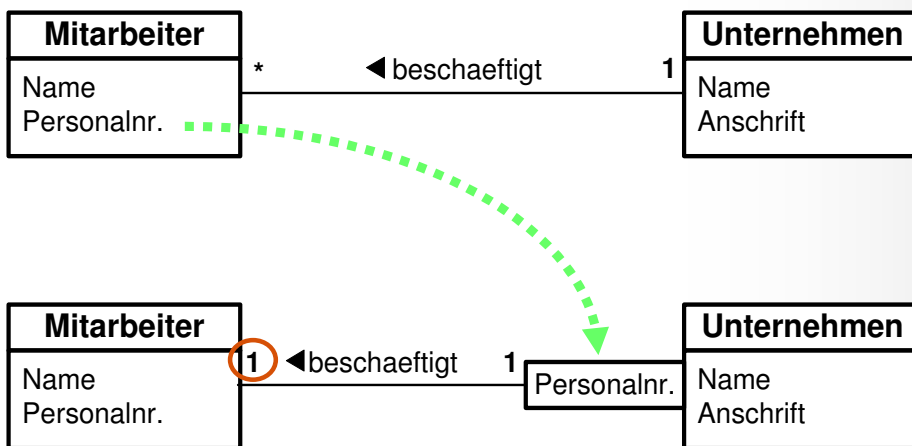
Verwendung von assoziativen Klassen (2)

- Beispiel: Umwandlung einer Assoziationsklasse



Qualifizierte Assoziation

- Einteilung der Menge der assoziierten Objekte durch spezielles Attribut, dessen Wert ein oder mehrere Objekte auf der anderen Seite selektiert
- Erhöhen den Informationsgehalt des Klassenmodells
- Das qualifizierende Attribut wird in einem Rechteck an der Seite der Klasse notiert



⇒ Ein Unternehmen beschäftigt eine Menge von Mitarbeitern

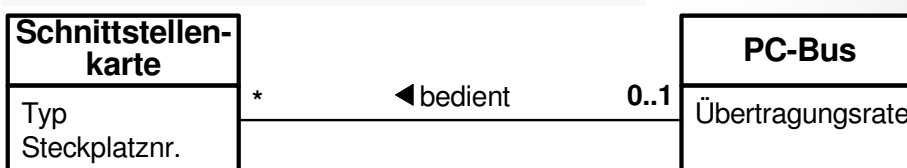
⇒ Jeder Mitarbeiter gehört genau zu einem Unternehmen

⇒ Mitarbeiter werden über ihre Personalnummer identifiziert

Qualifikationsangaben verändern die Kardinalität !

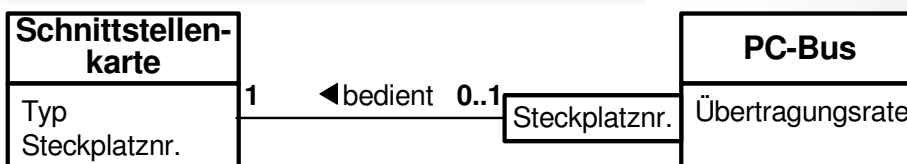
Beispiel zu qualifizierte Assoziation

- Ohne Qualifikation



Aussagekraft: Ein PC-Bus bedient viele Schnittstellenkarten

- Mit Qualifikation



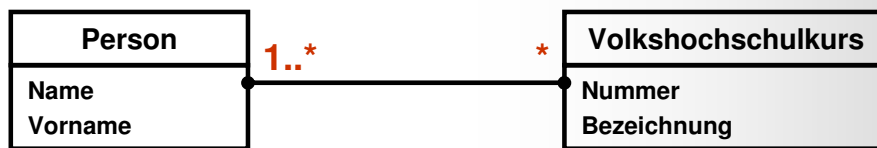
Aussagekraft: Ein PC-Bus-Objekt zusammen mit einer Steckplatznummer selektiert genau eine Schnittstellenkarte

- Veranschaulichung:

Bezeichnung(Karte)	Steckplatznr.	Typ (PC-Bus)
Graphikkarte	1	IDE-Bus
Soundkarte	2	IDE-Bus
Festplattencontroller	5	SCSI-Bus

Frage zu 3.2

Beschreiben Sie mit eigenen Worten, welche Art von Beziehung zwischen den zwei Klassen in dem nachfolgend abgebildeten Modell besteht.

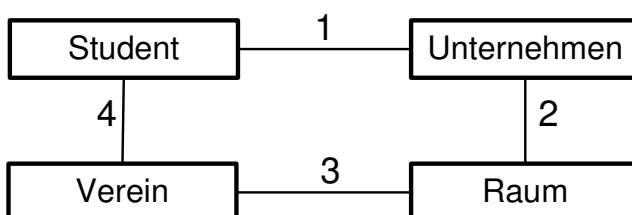


Antwort

- Assoziation zwischen den Klassen Person und Volkshochschulkurs
- Eine Person kann null oder mehrere VHS-Kurse besuchen
- Jeder VHS-Kurs hat mindestens einen Teilnehmer

Frage zu 3.2

Folgendes Diagramm ist gegeben:

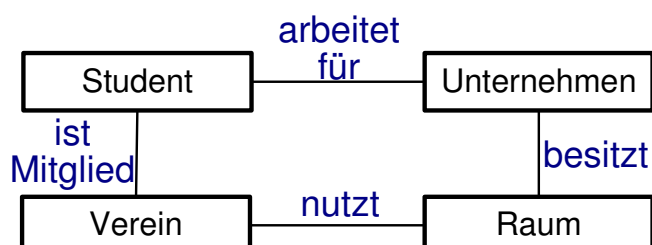


- a) "arbeitet für", b) "besitzt",
 c) "nutzt", d) "ist Mitglied",
 e) "befreundet mit",
 f) "schraubt"

Wie könnte die Bezeichnung der Assoziationen lauten?

Antwort

Möglichkeit 1:



Möglichkeit 2:



Kapitel 3 Statische Konzepte in der objektorientierten Analyse

3.1 Statische vs. dynamische Konzepte

3.2 Assoziation

3.3 Aggregation und Komposition

3.4 Vererbung

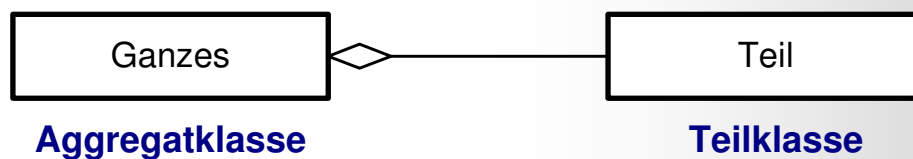
3.5 Paket

3.6 Erweiterungsmechanismen der UML (zum Selbststudium)

3.7 Zusammenfassung

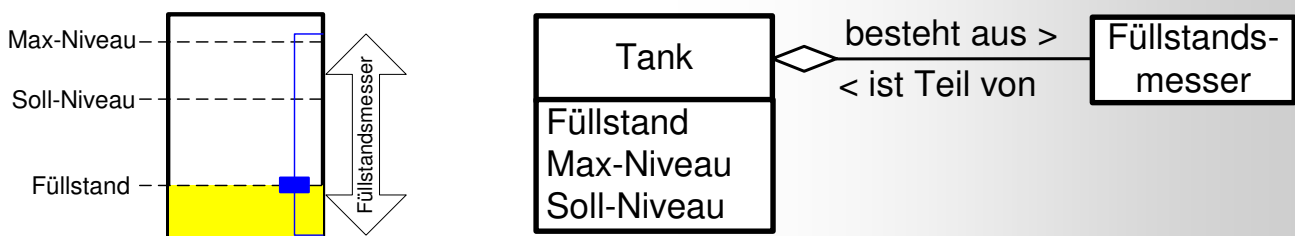
Aggregation

Definition: Eine **Aggregation** bezeichnet eine "Teil-Ganzes"- (whole-part) oder "ist-Teil-von"-Beziehung zwischen Klassen



- Aggregation ist eine spezielle Form der Assoziation
- Lässt sich durch **ist Teil von** bzw. **besteht aus** beschreiben (Ganzes & Teile)
- Raute kennzeichnet das Ganze

Beispiel:



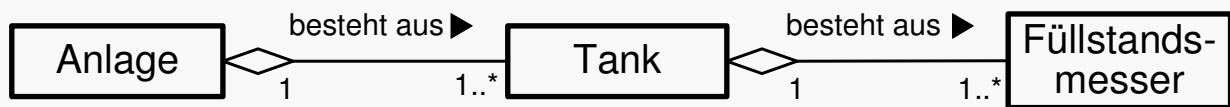
Eigenschaften der Aggregation

- ist asymmetrisch

wenn B Teil von A ist, dann darf A nicht Teil von B sein

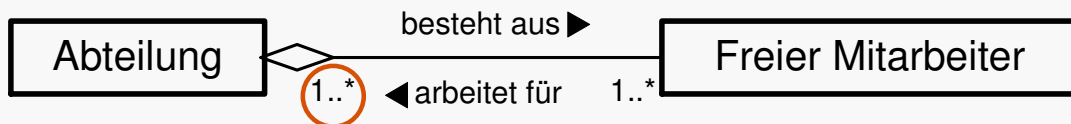
- ist transitiv

wenn A Teil von B und B Teil von C, dann ist auch A Teil von C



- Shared Aggregation

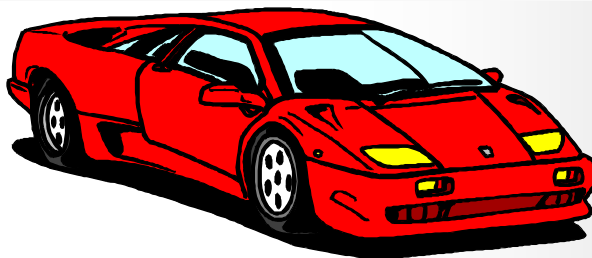
B darf gleichzeitig Teil von A und Teil von C sein



- Das Ganze übernimmt Aufgaben stellvertretend für seine Teile

Transitivität einer der Aggregation (Tafelanschrieb)

- Beispiel Pkw:

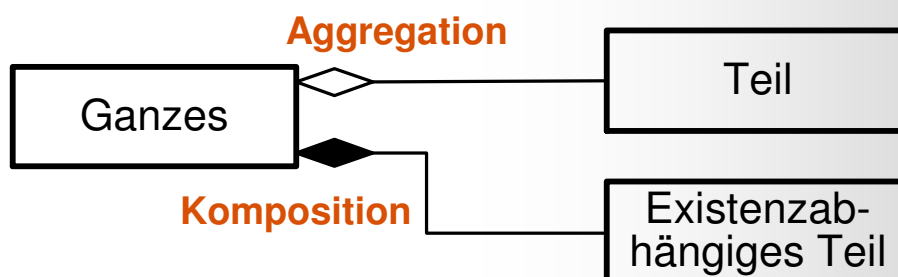


Wann liegt eine Aggregation vor?

- Eine Aggregation liegt vor, wenn die folgenden Fragen mit ja beantwortet werden können:
 1. Ist die Beschreibung "Teil von" zutreffend?
 2. Werden manche Operationen auf das "Ganze" automatisch auch auf die "Teile" angewandt?
 3. Pflanzen sich manche Attribute vom "Ganzen" auf alle oder einige "Teile" fort?
 4. Ist die Verbindung durch eine Asymmetrie gekennzeichnet, bei der die "Teile" dem "Ganzen" untergeordnet sind?
- Beispiele für Aggregationsbeziehungen
 - Das Ganze und seine Teile
Bsp.: Pkw (Ganzes) und Motor (Teil)
 - Der Behälter und sein Inhalt
Bsp.: Kaffeemaschine (Behälter) und Kaffeepulver (Inhalt)
 - Die Kollektion und ihre Mitglieder
Bsp.: Firma (Kollektion) und Angestellte (Mitglieder)

Komposition

Definition: Eine Aggregation mit starker Bindung (Existenzabhängigkeit) wird als **Komposition** bezeichnet



- Kennzeichnung durch gefüllte Raute

Beispiel:



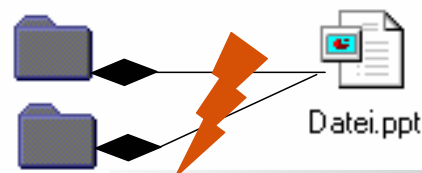
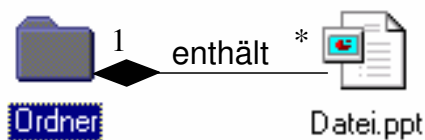
Eigenschaften der Komposition

Zusätzlich zur Aggregation gilt:

- Jedes Objekt der Teilklasse kann zu einem bestimmten Zeitpunkt nur Komponente eines einzigen Objekts der Aggregatklasse sein
 - Kardinalität der Aggregatklasse ≤ 1
 - Ein Teil darf evtl. auch anderem Ganzen zugeordnet werden (aber nicht gleichzeitig)
- Dynamische Semantik des Ganzen gilt auch für seine Teile (propagation semantics)

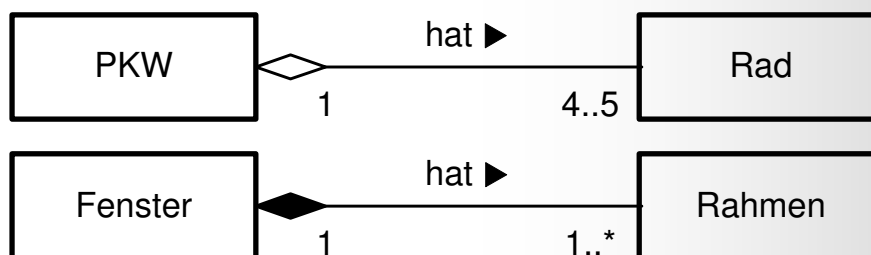
Wird das ganze kopiert, werden auch seine Teile kopiert

- Beispiel Ordner und Datei:



Nicht gleichzeitig möglich!

Vergleich Aggregation vs. Komposition (2)

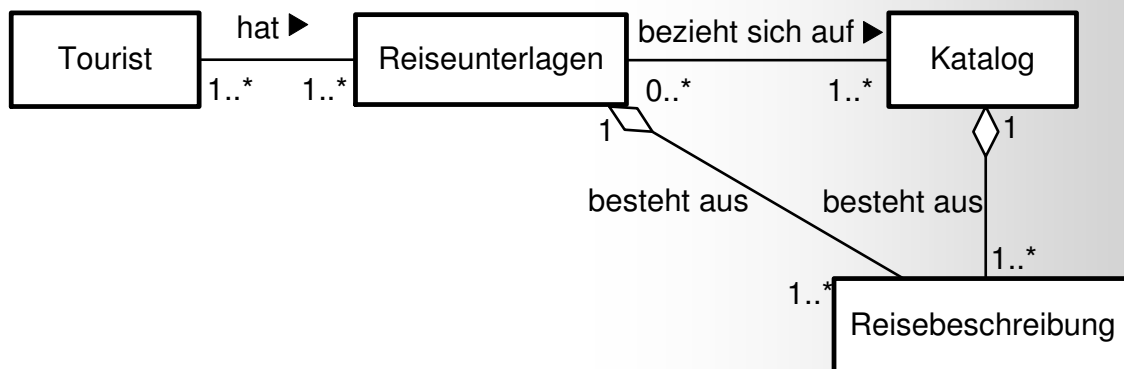


- Aggregation: "Pkw hat Räder"
 - Räder gehören notwendigerweise zu einem Auto
Aggregation
 - Räder können aber eigenständig und zwischen Pkw austauschbar betrachtet werden
keine Komposition
- Komposition: „Fenster hat Rahmen“
 - Wird das Fenster gelöscht, werden auch alle existenzabhängigen Einzelteile mitgelöscht
Komposition

Beispiel Reiseunternehmen

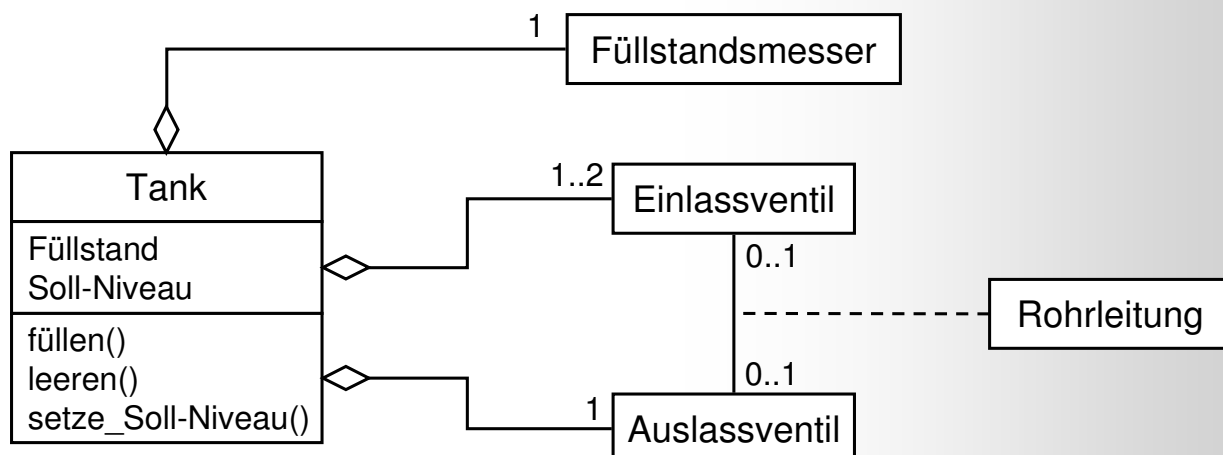
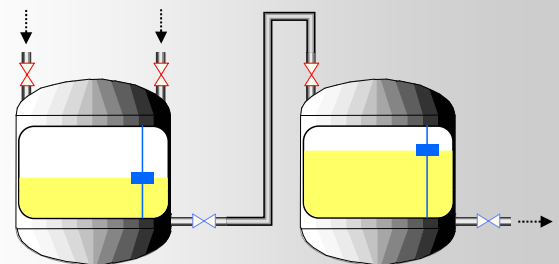
Ein Reiseunternehmen erarbeitet ein Klassenmodell für die Verwaltung seiner Reisekataloge, seiner Reisebeschreibungen und der Reiseunterlagen. Selbstverständlich beschreibt eine Klasse auch den Touristen, der die Reisen bucht.

- Ein Reisekatalog enthält eine oder mehrere Reisebeschreibungen.
- Die Reiseunterlagen für den Touristen werden aus einzelnen Reisebeschreibungen, eventuell aus verschiedenen Katalogen, zusammengestellt.
- Mehrere Touristen können auch gemeinsame Reiseunterlagen erhalten.



Beispiel Tanksystem

- Ein Tanksystem besteht aus mehreren Tanks, Ein- bzw. Auslassventilen, Füllstandsmessern und Rohrleitungen
- Jeder Tank hat 1-2 Einlassventile, ein Auslassventil und einen Füllstandsmesser
- Je zwei Ventile können über Rohrleitungen miteinander verbunden sein.



Frage zu 3.3

1. Gegeben seien die Klassen Haus und Zimmer. Welche Art Beziehung würden Sie zwischen beiden modellieren?

- ☐ Die einfache Assoziation "hat Beziehung zu"
- ☐ Die Aggregation "besteht aus"
- ☒ Die Komposition "besteht aus"

Begründung: Ein Haus und seine Zimmer sind sehr eng miteinander verbunden. Ein Zimmer existiert ohne das Haus nicht mehr, und auch ein Haus existiert nicht, wenn alle Zimmer zerstört sind.

2. Gegeben seien die Klassen Haus und Mieter. Welche Art der Assoziation würden Sie modellieren?

- ☒ Die einfache Assoziation "hat"
- ☐ Die Aggregation "setzt sich zusammen aus"
- ☐ Die Komposition "besteht aus"

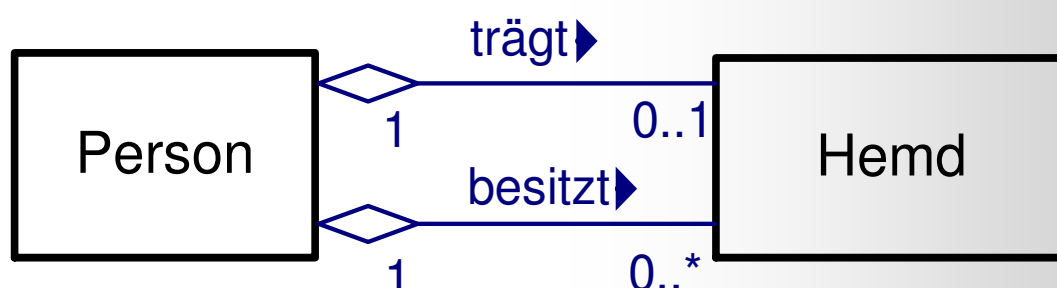
Begründung: Die Beziehung zwischen Haus und Mietern ist nicht so innig. Aggregation ist auch noch vertretbar, aber etwas übertrieben.

Frage zu 3.3

Drücken Sie in UML die Relation zwischen einer Person und ihren Hemden aus

Antwort

- Eine Person hat eine Aggregationsbeziehung mit ihren Hemden auf zwei Ebenen:



Kapitel 3 Statische Konzepte in der objektorientierten Analyse

3.1 Statische vs. dynamische Konzepte

3.2 Assoziation

3.3 Aggregation und Komposition

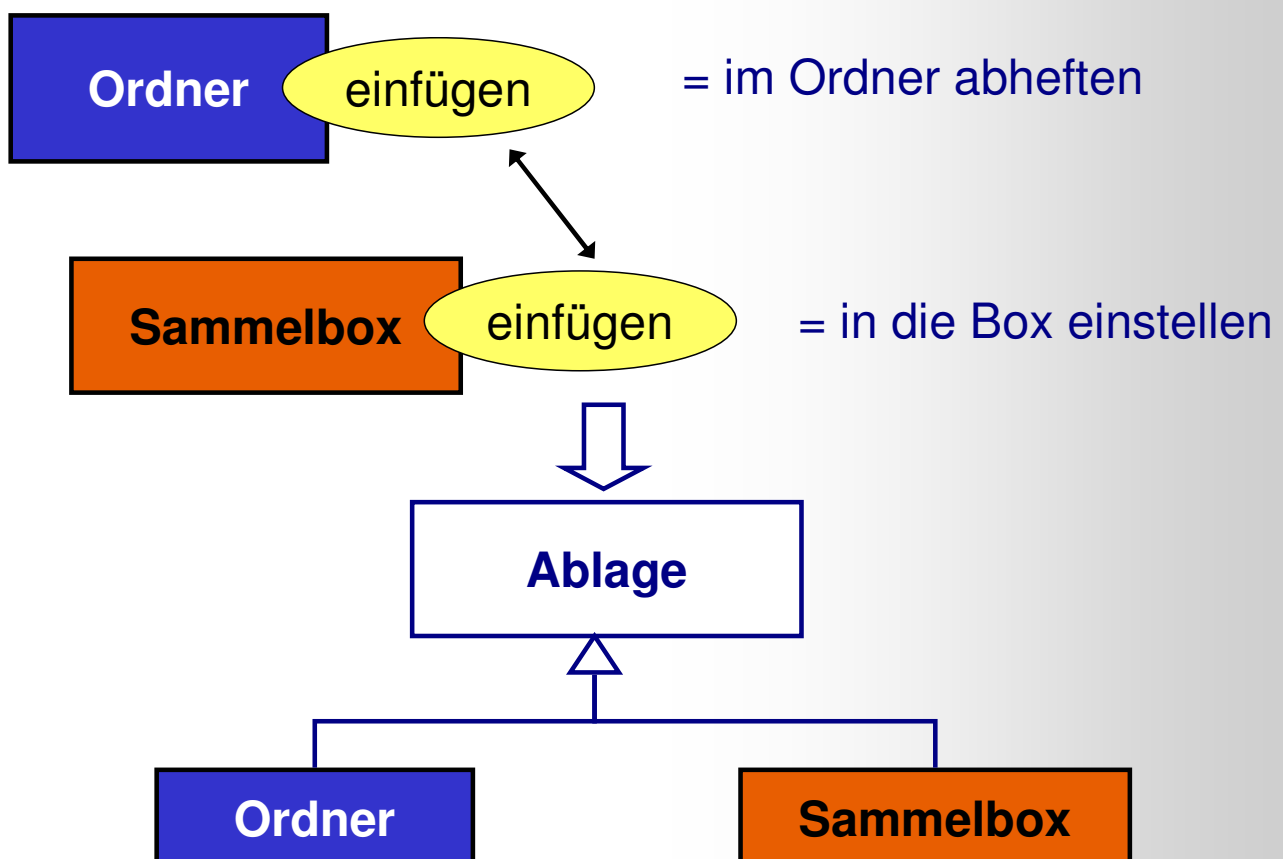
3.4 Vererbung

3.5 Paket

3.6 Erweiterungsmechanismen der UML (zum Selbststudium)

3.7 Zusammenfassung

Ähnliches Verhalten von Objekten (Tafelanschrieb)



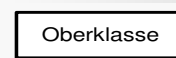
Was ist Vererbung?

Definition: Eine **Vererbung** (*generalization*) ist eine Beziehung zwischen einer allgemeinen Klasse (Basisklasse) und einer spezialisierten Klasse.

Vererbung findet nur zwischen Klassen statt!

- Vererbung ist ein Abstraktionsprinzip zur hierarchischen Strukturierung (Einordnung der Klassen in eine Hierarchie)
- Ziel: Gemeinsame Eigenschaften und Verhaltensweisen zusammenfassen
- Spezialisierte Klasse ist vollständig konsistent mit der Basisklasse, hat aber zusätzliche Informationen (Attribute, Operationen, Assoziationen)

- Allgemeine Klasse = Oberklasse (*super class*)



Generalisierung

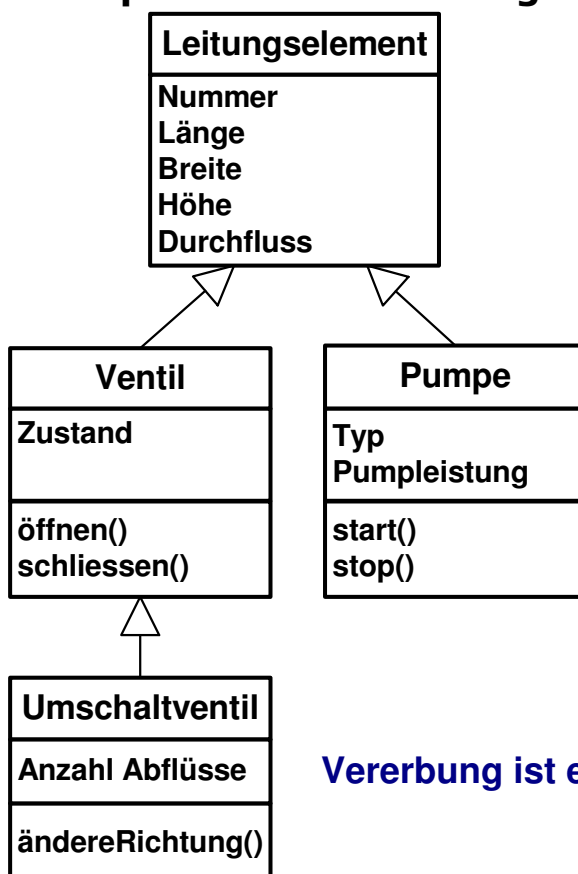
- Spezialisierte Klasse = Unterklasse (*sub class*)



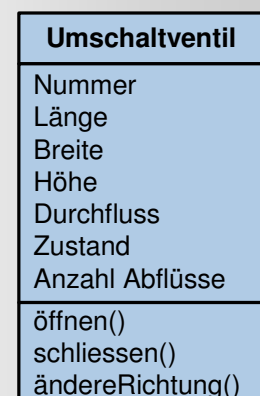
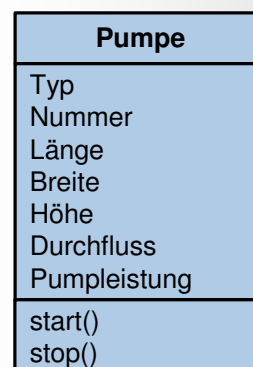
Spezialisierung

Substitutionsprinzip: Objekt der spezialisierten Klasse kann **überall** verwendet werden, wo ein Objekt der Basisklasse erlaubt ist, aber nicht anders herum!

Beispiel einer Vererbungsstruktur

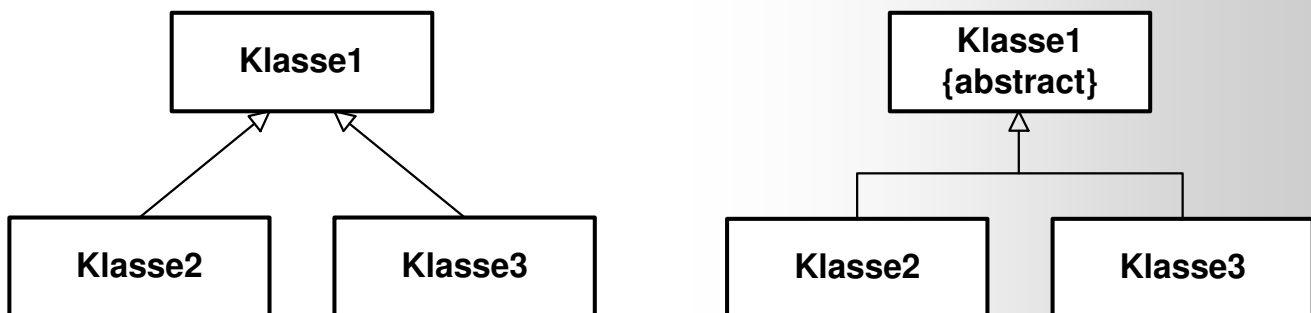


Vererbung ist eine "ist ein" Beziehung



UML Notation der Vererbung

- Weißes bzw. transparentes Dreieck bei der Basisklasse

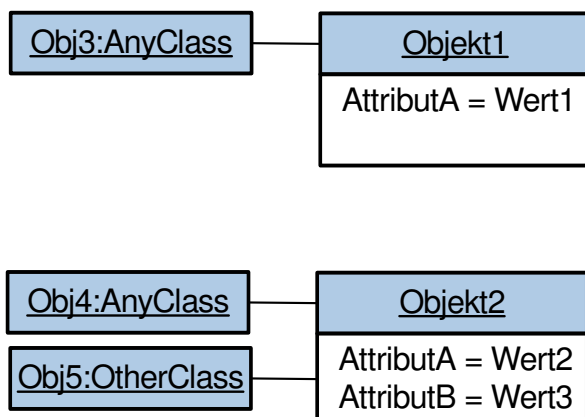


Beide Darstellungen sind gleichwertig und können alternativ verwendet werden

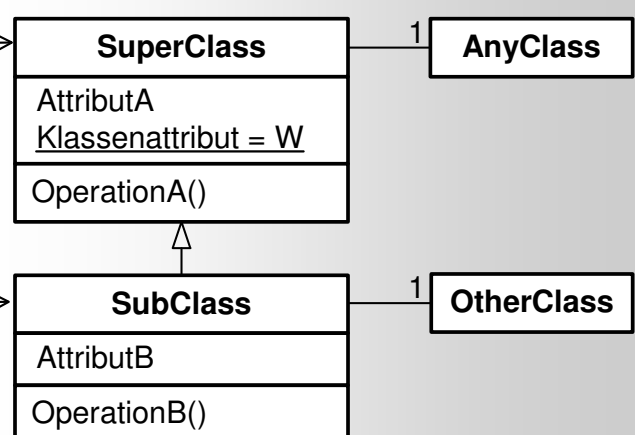
Was wird vererbt ?

- Operationen, Attribute und Assoziationen

Objektdiagramm



Klassendiagramm

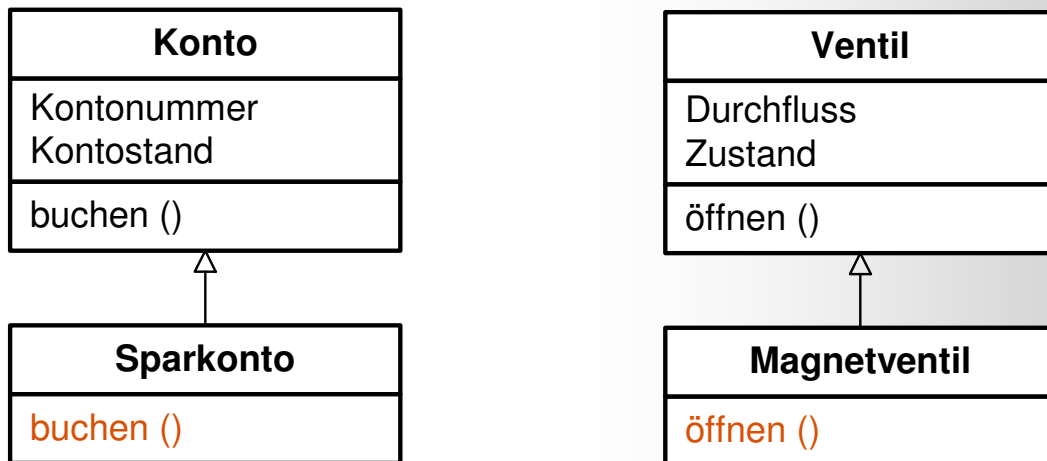


1. Attribut A von Superclass wird nach Subclass vererbt
2. OperationenA() von Superclass auch auf Subclass anwendbar
3. Klassenattribut mit dem Wert W von Superclass an Subclass
4. Assoziation zwischen Superclass und AnyClass an Subclass

Überschreiben von Operationen

- Unterklassen können das Verhalten ihrer Oberklassen verfeinern, redefinieren bzw. überschreiben (*redefine, override*)
- Operationen können in der Unterklasse überschrieben, aber nicht eliminiert werden

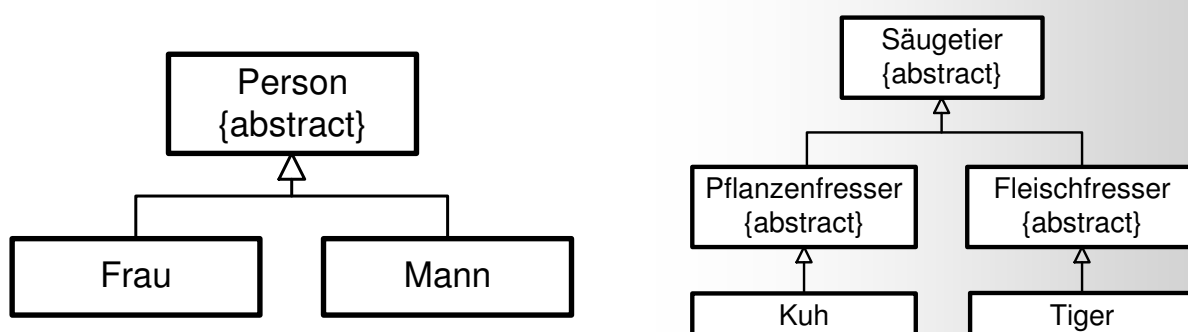
Zum Überschreiben muss die Operation der Unterklasse die gleiche Signatur haben wie in der Oberklasse.



Abstrakte Klasse

- Unterscheidung zwischen abstrakten und konkreten Klassen
- Von einer abstrakten Klasse können keine Objekte erzeugt werden
- Verwendung, um ihre Informationen an spezialisierte Klassen zu vererben
- Kennzeichnung durch *kursiven* Namen oder durch Merkmal **{abstract}**
- Eine abstrakte Klasse kann abstrakte Operationen enthalten. Abstrakte Operationen müssen in der Unterklasse implementiert werden.

Abstrakte Klasse macht nur Sinn, wenn eine konkrete Klasse von ihr erbt



Polymorphismus

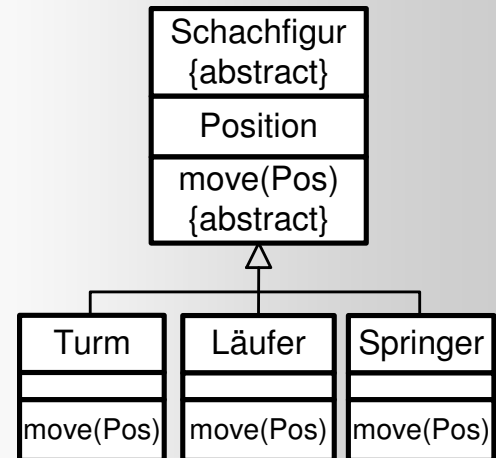
Definition:

Eine Botschaft ist **polymorph**, wenn sie bei unterschiedlichen Objekten Methoden aktiviert, die verschiedene Semantiken besitzen.

- Polymorphie bedeutet unterschiedliches Verhalten verschiedener Objekte auf die gleiche Botschaft

Beispiel

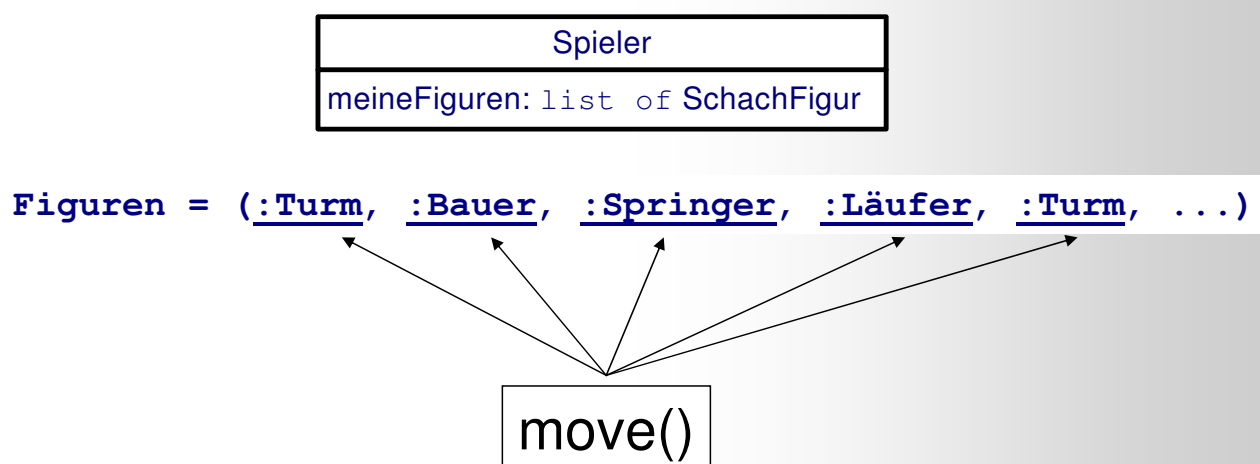
- Methode `move(Position)` bei den verschiedenen Spielfiguren unterschiedlich implementiert
 - Turm: Nur geradeaus
 - Läufer: Nur diagonal
 - Springer: 1 Feld gerade, 1 Feld diagonal



⇒ **Gleiche Botschaft `move(Position)` löst unterschiedliche Reaktion aus**

Funktionsweise des Polymorphismus (Tafelanschrieb)

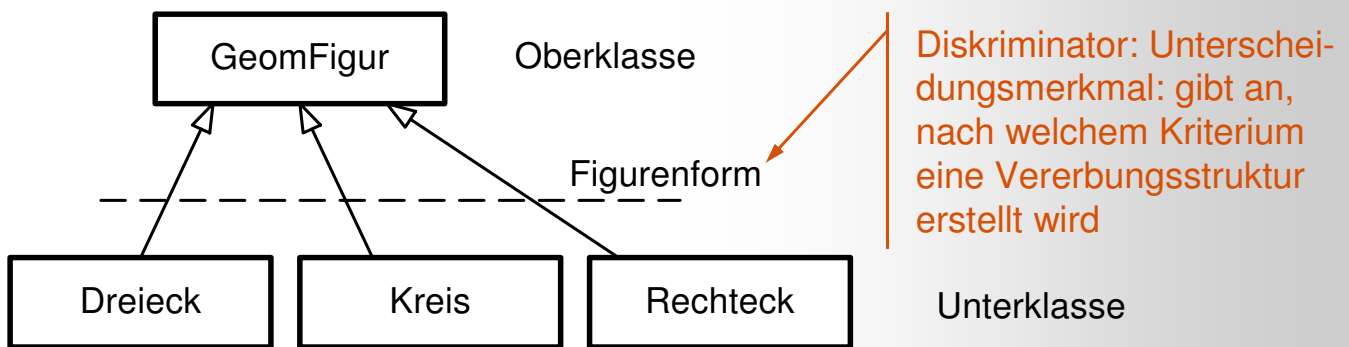
- Beispiel Schachspiel



- Methode `move(Pos)` → unterschiedliche Reaktionen
- Z.B. `Figuren[2].move(a3)`

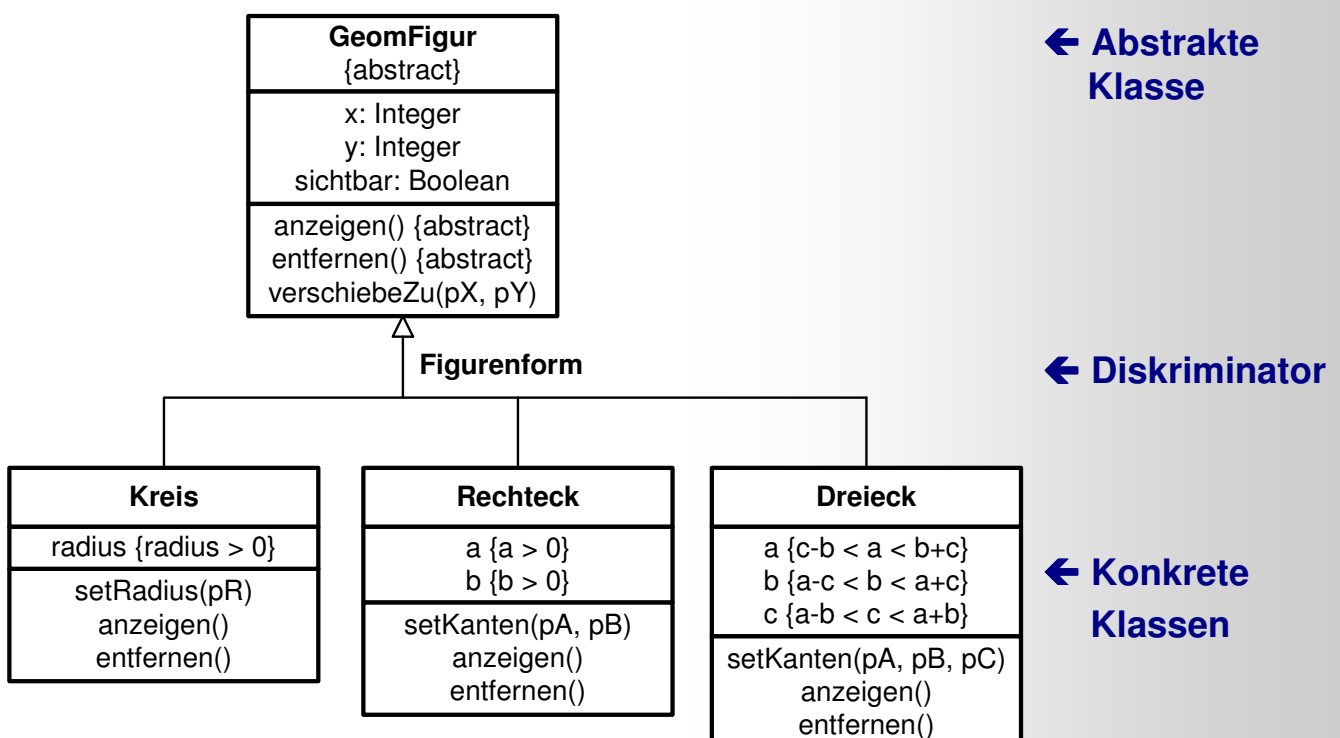
Beispiel für Vererbung (1)

- Auf einer Fensterfläche sollen Kreise, Rechtecke und Dreiecke angezeigt, verschoben und entfernt werden können.
- Die Begriffe Kreis, Rechteck und Dreieck können generalisiert und ganz allgemein als geometrische Figuren bezeichnet werden.
- Die Klassen Kreis, Rechteck und Dreieck sind demnach Spezialisierungen der gemeinsamen Oberklasse GeomFigur



Beispiel für Vererbung (2)

- Klassendiagramm



Beispiel für Vererbung (3)

– Java-Programmcode (Ausschnitt)

```
abstract class GeomFigur
{
    int x, y;
    boolean sichtbar;

    public abstract void anzeigen();
    public abstract void entfernen();
    public void verschiebeZu(int pX,pY)
    {
        if (sichtbar)
        {
            entfernen();
            this.x = pX;
            this.y = pY;
            anzeigen();
        } else
        {
            this.x = pX;
            this.y = pY;
        }
    }
}
```

```
class Rechteck extends GeomFigur
{
    int a, b;

    public void anzeigen()
    {...}

    public void entfernen()
    {...}

    public void setKanten(int pA, pB)
    {
        if ((a > 0) && (b > 0))
        {
            this.a = pA;
            this.b = pB;
        }
    }
}
```

Vor- und Nachteile der Vererbung

- + Unterstützung der Änderbarkeit
 - Änderung von Attributen in der Oberklasse wirkt sich automatisch auf alle Unterklassen aus
- + Einsparung von Code
- + Unterstützung der Wiederverwendbarkeit
- Verletzung des Geheimnisprinzips
 - Unterklasse ist von Änderungen der Oberklasse abhängig
 - Um die Unterklasse zu verstehen, muß auch die Oberklasse betrachtet werden

Vererbung ist ein mächtiges, aber auch gefährliches Konzept

Frage zu 3.4

Kann ein Objekt Attribute und Methoden erben?

Antwort

- ☐ Ja
- ☒ Nein
- ☐ Ja, außer Klassenattribute und -operationen

Begründung

- Die Klasse kann Attribute und Methoden von anderen Klassen erben
- Objekt bekommt die Attribute und Methoden seiner Klasse bei der Erzeugung aufgeprägt
- Damit profitiert das Objekt indirekt von der Vererbung
- Selbst steht es aber in keiner Vererbungsbeziehung

**Kapitel 3 Statische Konzepte in der
objektorientierten Analyse**

3.1 Statische vs. dynamische Konzepte

3.2 Assoziation

3.3 Aggregation und Komposition

3.4 Vererbung

3.5 Paket

3.6 Erweiterungsmechanismen der UML (zum Selbststudium)

3.7 Zusammenfassung

Was ist ein Paket?

Definition:

Ein **Paket** (*package*) fasst Modellelemente (z.B. Klassen) zusammen

- Pakete schaffen eine bessere Übersicht über ein großes Modell
- Ein Paket kann selbst Pakete enthalten
- Beschreibung der Systemstruktur auf einer hohen Abstraktionsebene

Das vollständige System kann als ein großes Paket aufgefasst werden

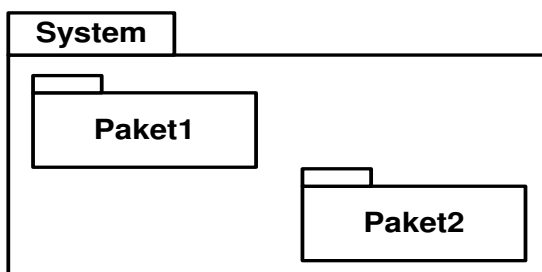
Beispiel: Brauerei



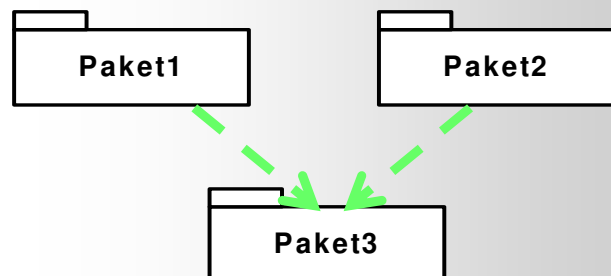
UML-Notation von Paketen

- Paketname muss im gesamten System eindeutig sein
- Paketdiagramm: enthält Pakete und deren Abhängigkeiten

Notation von Paketen



Abhängigkeiten von Paketen



Paket1 und Paket2 sind von Paket3 abhängig

- Jede Klasse (jedes Modellelement) gehört zu höchstens einem Paket
- Es kann in mehreren anderen Paketen darauf verwiesen werden
- Ein Paket definiert einen Namensraum für alle enthaltenen Klassen
- Verweis von außerhalb des Pakets
 - `Paket::Klasse` oder `Paket1::Paket11::Paket111::Klasse`

Vererbung bei Paketen

- Abgeleitete Pakete erben öffentliche (public) und geschützte (protected) Elemente des Oberpaketes
- Elemente können in den abgeleiteten Paketen überschrieben werden

Vorteile der Aufteilung in Pakete

- + Besseres Verständnis durch Betrachtung von Teilsystemen bzw. Systemausschnitten
- + Vermeidung von Namenskonflikten
- + Zugriffskontrolle für Elemente in Paketen, Kapselung
- + Vereinfachung der Testphase durch separates Testen von Paketen



Frage zu 3.5

Welche Richtlinien zur Bildung von Paketen sind sinnvoll?

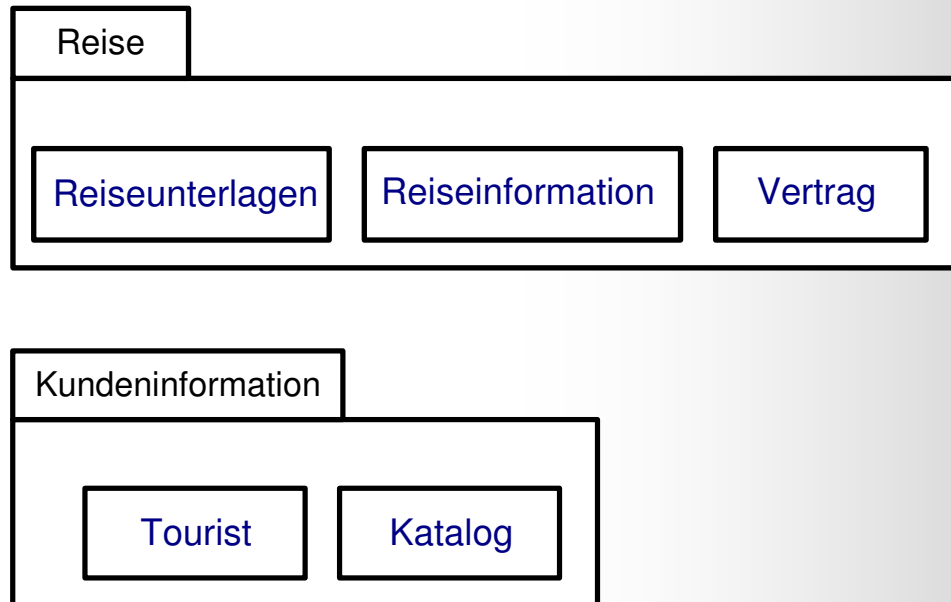
Antwort

- ☒ Zusammenfassung von Klassen mit intensiver Kommunikation
- ☐ Es müssen mind. drei Pakete pro System existieren
- ☐ Paketname aus Namen der beinhalteten Klassen ableiten
- ☒ Identifikation aus zusammengehörigen Anwendungsfällen
- ☒ Aggregationen, Kompositionen und Vererbungsstrukturen im selben Paket zusammenfassen



Frage zu 3.5

Unterteilen Sie die Klassen Reiseunterlagen, Reiseinformation, Vertrag, Tourist und Katalog in die Pakete Reise und Kundeninformation.

Antwort**Kapitel 3 Statische Konzepte in der
objektorientierten Analyse**

3.1 Statische vs. dynamische Konzepte

3.2 Assoziation

3.3 Aggregation und Komposition

3.4 Vererbung

3.5 Paket

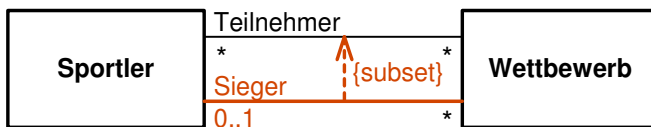
3.6 Erweiterungsmechanismen der UML (zum Selbststudium)

3.7 Zusammenfassung

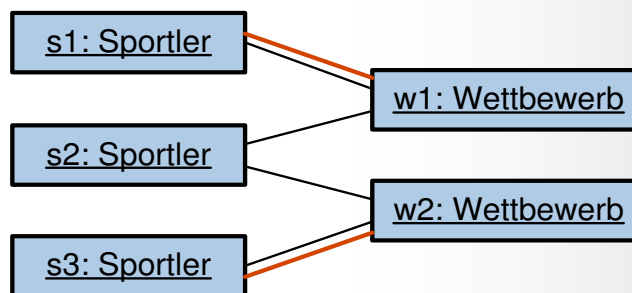
Weitere Arten von Restriktionen bei Assoziationen

– subset-Restriktion

- **subset**- Restriktion bildet Teilmenge
- Existiert nur, wenn die Hauptassoziation auch existiert

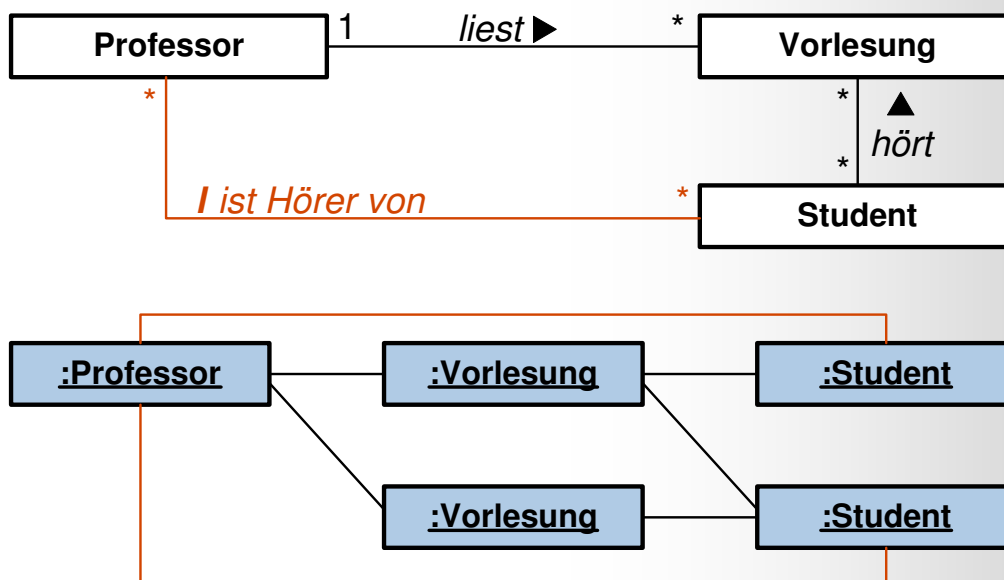


Ein Sportler kann nur Sieger eines Wettbewerbs sein, an dem er auch teilgenommen hat



Abgeleitete Assoziation (derived association)

- Assoziation, deren konkrete Objektbeziehungen jederzeit aus den Werten anderer Objektbeziehungen und Objekte abgeleitet werden (⇒ redundant)
- wird durch das Präfix „/“ gekennzeichnet
- Ableitungsvorschrift wird ggf. als Restriktion notiert



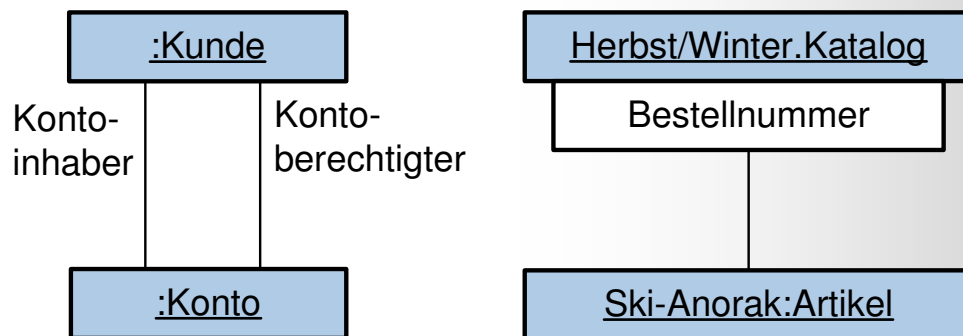
Verwendung von Assoziationen in Objektdiagrammen

- Steigerung des Aussagegehalts eines Objektdiagramms durch Verwendung von

- Rollennamen

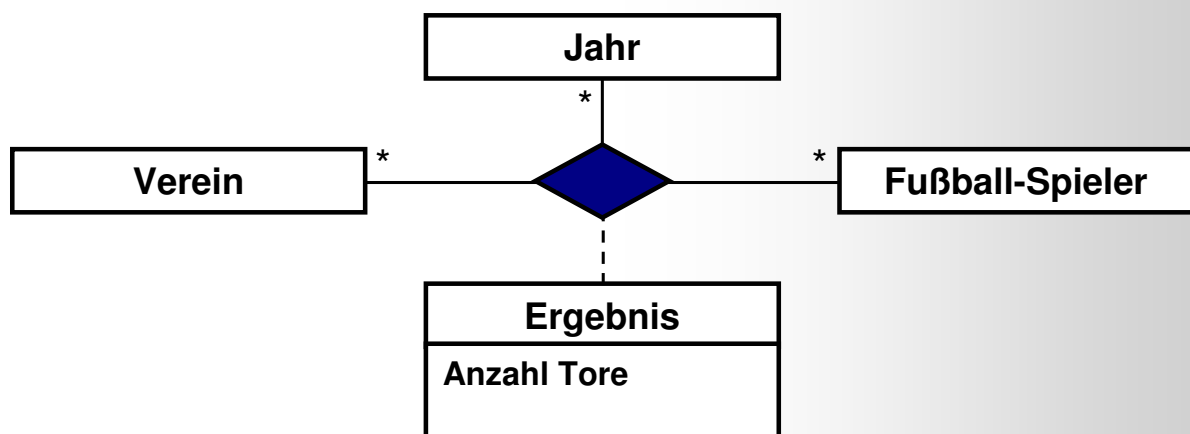
Wird der Assoziationsname an die Objektverbindung angetragen, dann muss er unterstrichen werden !

- Qualifikationsangaben
- Symbole für Aggregation bzw. Komposition (siehe Kap. 3.3)



Höherwertige Assoziation

- Prinzipiell sind auch Assoziationen zwischen drei und mehr Klassen möglich
- Bezeichnung: n-äre Assoziation (n-ary association)
- Ternäre und höhere Assoziationen können keine Aggregation oder Komposition bilden



Kapitel 3 Statische Konzepte in der objektorientierten Analyse

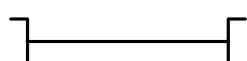
- 3.1 Statische vs. dynamische Konzepte
- 3.2 Assoziation
- 3.3 Aggregation und Komposition
- 3.4 Vererbung
- 3.5 Paket
- 3.6 Erweiterungsmechanismen der UML (zum Selbststudium)
- 3.7 Zusammenfassung**

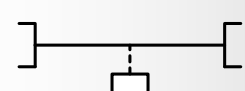
Zusammenfassung Kapitel 3

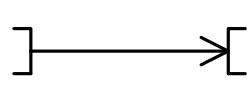
- Eine **Assoziation** modelliert Verbindungen zwischen Objekten einer oder mehrerer Klassen
- Sonderfälle der Assoziation sind **Aggregation** und **Komposition**
- **Vererbung** beschreibt die Beziehung zwischen einer allgemeinen Klasse (Basisklasse) und einer spezialisierten Klasse
- Ein **Paket** gruppiert Modellelemente und ermöglicht die Darstellung des Softwaresystems auf einem höheren Abstraktionsniveau

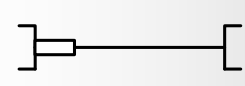
 Vererbung

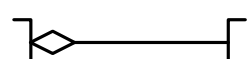
 Komposition

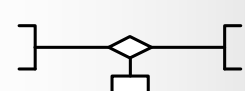
 Assoziation

 Assoziative Klasse

 Gerichtete Assoziation

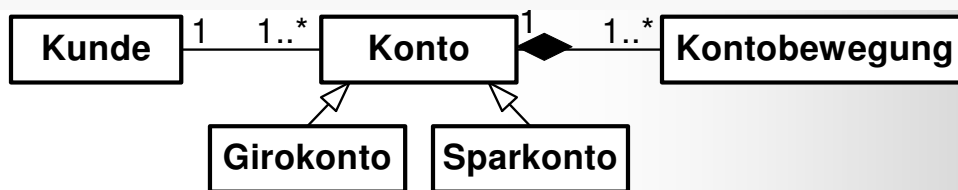
 Qualifizierte Assoziation

 Aggregation

 Höherwertige Assoziation

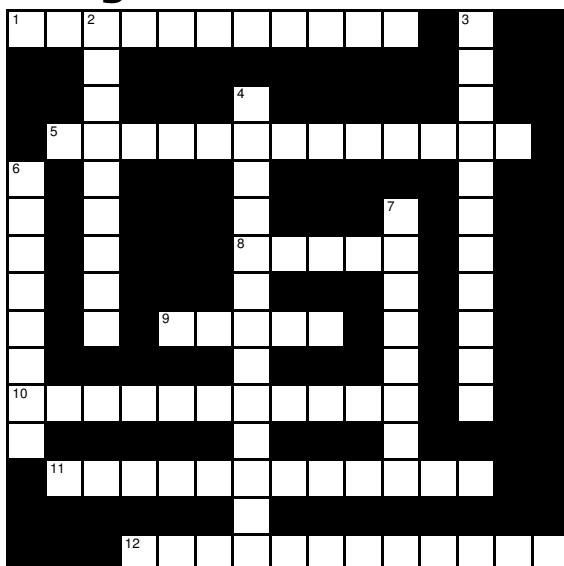
Frage zu Kapitel 3

Beschreiben Sie in dem Klassendiagramm für eine **Kontoverwaltung** die Beziehungen zwischen den Klassen !



- Zwischen **Kunde** und **Konto** besteht eine einfache Assoziation, weil weder der Kunde Teil von Konto ist, noch umgekehrt
- Zwischen **Konto** und **Kontobewegung** existiert eine **Komposition**, weil eine whole-part-Beziehung vorliegt: dynamische Semantik eines Kontos gilt stets für alle Kontobewegungen
- Weil bei der Eröffnung eines Kontos gleich eine Einzahlung vorzunehmen ist, ist bei Kontobewegung die Kardinalität 1..* eingetragen
- **Komposition** von Konto zu Kontobewegung wird an beide Unterklassen **vererbt**, d.h. jedes Sparkonto- und Girokonto-Objekt hat Kontobewegungen
- Die **Vererbung** der **Assoziation** zu **Kunde** bedeutet, dass Verbindungen zwischen Kunde und Girokonto bzw. Sparkonto existieren

Frage: Kreuzworträtsel zu Kapitel 3



Waagrecht

- Assoziation, bei der die beteiligten Klassen keine gleichwertige Beziehung führen, sondern eine Ganze-Teile-Hierarchie darstellen
- Unterscheidungsmerkmal in Generalisierungs- und Spezialisierungsbeziehungen
- Beschreibt die Bedeutung eines Objektes in einer Assoziation
- Ansammlungen von Modellelementen beliebigen Typs, mit denen das Gesamtmodell in kleinere überschaubare Einheiten gegliedert wird
- Strenge Form der Aggregation, bei der die Teile vom Ganzen existenzabhängig sind
- Anzahl möglicher Objektverbindungen in einer Assoziation
- Assoziation, bei der die referenzierte Menge der Objekte durch Attribute in Partitionen unterteilt wird, heißt ...

Senkrecht

- Eine Assoziation, bei der von der einen beteiligten Assoziationsrolle zur anderen direkt navigiert werden kann, nicht aber umgekehrt, ist ...
- Modellelement zur Beschreibung von Beziehungen zwischen Objekten einer oder mehrerer Klassen
- Eine beidseitig direkt navigierbare Assoziation bezeichnet man als ...
- Wenn von einer Basisklasse keine Objekte erzeugt werden können, sondern nur aus abgeleiteten Klassen, bezeichnet man eine solche Klasse als ...
- Eine Assoziation, bei der die Objektverbindungen sich in einer bestimmten Reihenfolge befinden, ist ...

Kapitel 4 Dynamische Konzepte in der objektorientierten Analyse

<u>4.1 Anwendungsfall</u>	<u>218</u>
<u>4.2 Botschaft</u>	<u>232</u>
<u>4.3 Szenario</u>	<u>237</u>
<u>4.4 Zustandsautomat</u>	<u>254</u>
<u>4.5 Zusammenfassung objektorientierte Konzepte</u>	<u>272</u>
<u>4.6 Zusammenfassung</u>	<u>279</u>

Kapitel 4 **Dynamische Konzepte in der objektorientierten Analyse**

4.1 Anwendungsfall

4.2 Botschaft

4.3 Szenario

4.4 Zustandsautomat

4.5 Zusammenfassung der objektorientierten Konzepte
(zum Selbststudium)

4.6 Zusammenfassung

Lernziele

- Erklären können, was ein Anwendungsfall ist
- Erklären können, was eine Botschaft ist
- Erklären können, was ein Szenario ist
- Erklären können, was ein Zustandsautomat ist und welche Rolle er im dynamischen Modell spielt
- Erklären können, was ein Aktivitätsdiagramm ist
- Erklären können, wie das Klassendiagramm und Diagramme des dynamischen Modells zusammenwirken
- Anwendungsfälle modellieren können
- Sequenz- und Kollaborationsdiagramme erstellen können
- Zustandsdiagramme erstellen können



Kapitel 4 Dynamische Konzepte in der objektorientierten Analyse

4.1 Anwendungsfall

4.2 Botschaft

4.3 Szenario

4.4 Zustandsautomat

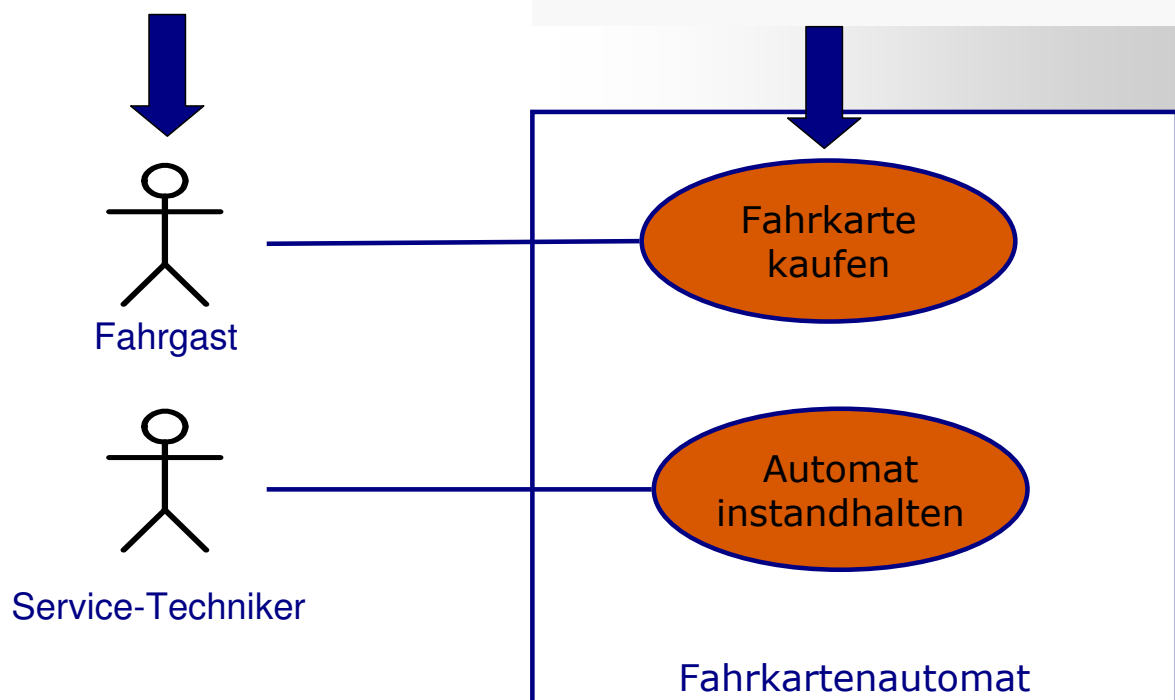
4.5 Zusammenfassung der objektorientierten Konzepte (zum Selbststudium)

4.6 Zusammenfassung

Benutzersicht auf einen Fahrkartenautomaten (Tafelanschrieb)

1. Frage: Wer arbeitet mit einem Fahrkartenautomat?

2. Frage: Was will der jeweilige Benutzer mit Fahrkartenautomaten machen?



Begriff Anwendungsfall (use case)

Definition:

Ein **Anwendungsfall (use case)** besteht aus mehreren zusammenhängenden Aufgaben, die von einem Akteur durchgeführt werden, um ein Ziel zu erreichen bzw. ein gewünschtes Ergebnis zu erstellen.

- Begriff **use case** von **Jacobson** in die Objektorientierung eingeführt
- Einige Autoren übersetzen **use case** mit Geschäftsprozess

Ziel von Anwendungsfällen

- Spezifikation der ergebnisorientierten Arbeitsabläufe bei Benutzung der zu realisierenden Software
- Ermitteln, welche Aufgaben mit dem neuen Softwaresystem zu bewältigen sind, um die gewünschten Ergebnisse zu erzielen

Anwendungsfall in einem Informationssystem

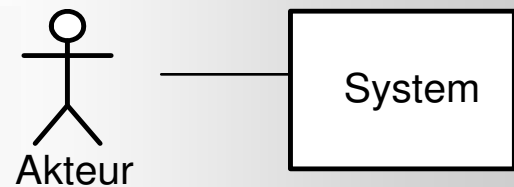
- Sequenz von zusammengehörenden Transaktionen, die von einem Akteur im Dialog mit einem System durchgeführt werden, um ein Ergebnis von messbarem Wert zu erhalten
- Transaktion = Menge von Verarbeitungsschritten, von denen entweder alle oder keiner ausgeführt wird
- Alle Anwendungsfälle zusammen dokumentieren alle Möglichkeiten der Benutzung des Systems (**use case model**)

Problematik in der Analyse

- Nicht abzusehen, ob alle Aufgaben durch die Software realisiert werden oder ob auch organisatorische Schritte enthalten sind, in denen der Benutzer Entscheidungen treffen oder bestimmte Aktivitäten durchführen muss

Was ist ein Akteur (actor) ?

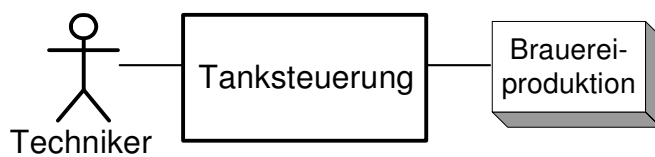
- Rolle, die ein Benutzer des Systems spielt
- Häufig eine Person
- Kann auch Organisationseinheit oder anderes System sein



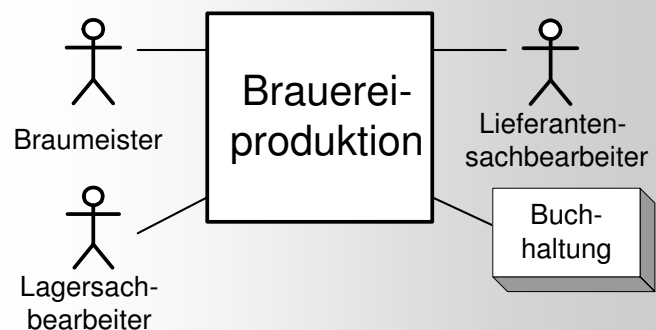
- **Befindet sich immer außerhalb des Systems** **Akteur beeinflusst System**

Beispiele für Akteure

- Tanksteuerung



- Brauereiproduktion



Beschreibung von Anwendungsfällen

- Ein Anwendungsfall wird semiformal oder informal (umgangssprachlich) beschrieben
- Beschreibung als Folge von einzelnen Aktionen
- Aktionen für bessere Übersichtlichkeit durchnummeriert
- Unterscheidung zwischen
 - Standardfall (häufigster Fall)
 - Erweiterungen
 - Alternative Abläufe
- Verwendung von Spezifikationsschablonen
- Beschreibung stets unabhängig von der Benutzungsoberfläche!

Oberfläche ändert sich häufig

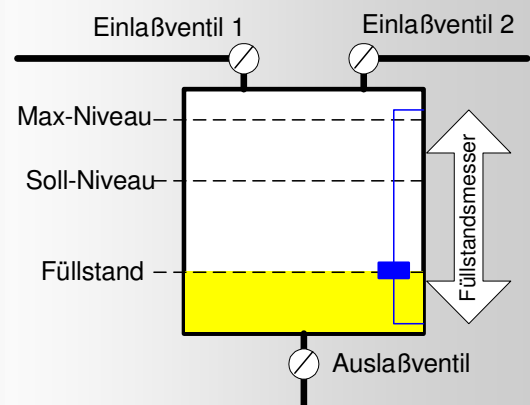
Anwendungsfall-Spezifikationsschablone (Use Case template)

Anwendungsfall:	<Name des Anwendungsfalls>
Ziel:	<globale Zielsetzung bei erfolgreicher Ausführung des Anwendungsfalls>
Kategorie:	<primär (notwendig, häufig benötigt) sekundär (notwendig, selten benötigt) optional (nützlich, nicht unbedingt notwendig)>
Vorbedingung:	<erwarteter Zustand, bevor der Anwendungsfall beginnt>
Nachbedingung Erfolg:	<>
Nachbed. Fehlschlag:	<>
Akteure	<Alle Akteure, die den Anwendungsfall ausführen>
Auslösendes Ereignis:	<>
Beschreibung:	<Hier wird der Standardfall beschrieben> 1 <Erste Aktion> 2 <Zweite Aktion>
Erweiterungen:	1a <Erweiterung des Funktionsumfangs der ersten Aktion>
Alternativen: :	1a <Alternative Ausführung der ersten Aktion>

Beispiel für Anwendungsfall: Tank füllen

Umgangssprachliche Beschreibung im Lastenheft:

- Der Techniker oder die Brauereiproduktion kann das gewünschte Flüssigkeitsniveau eingeben, wenn der Tank leer ist.
- Der Techniker oder die Brauereiproduktion startet das Füllen des Tanks.
- Beim Füllen des Tanks werden beide Einlassventile geöffnet und die Flüssigkeit strömt in den Tank ein, bis die vorgegebene Füllhöhe (Soll-Niveau) erreicht ist.
- Der Techniker kann auch die maximale Füllhöhe (Max-Niveau) vorgeben.



Identifizierte Akteure:

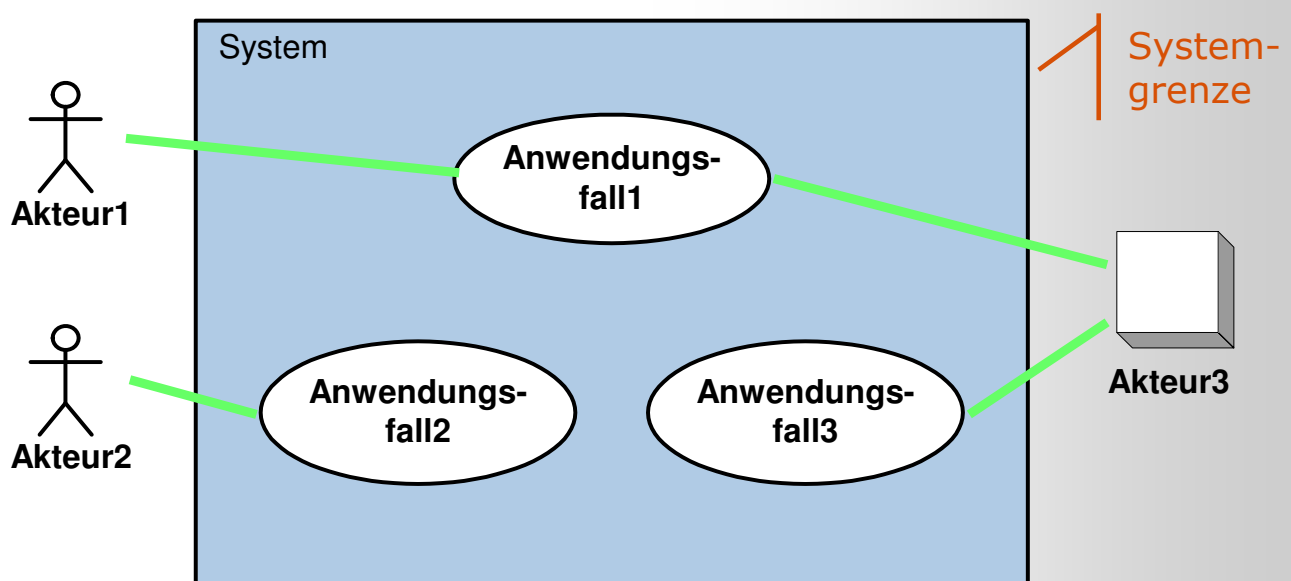
- Techniker
- Brauereiproduktion

Beispiel für Anwendungsfall: Tank füllen

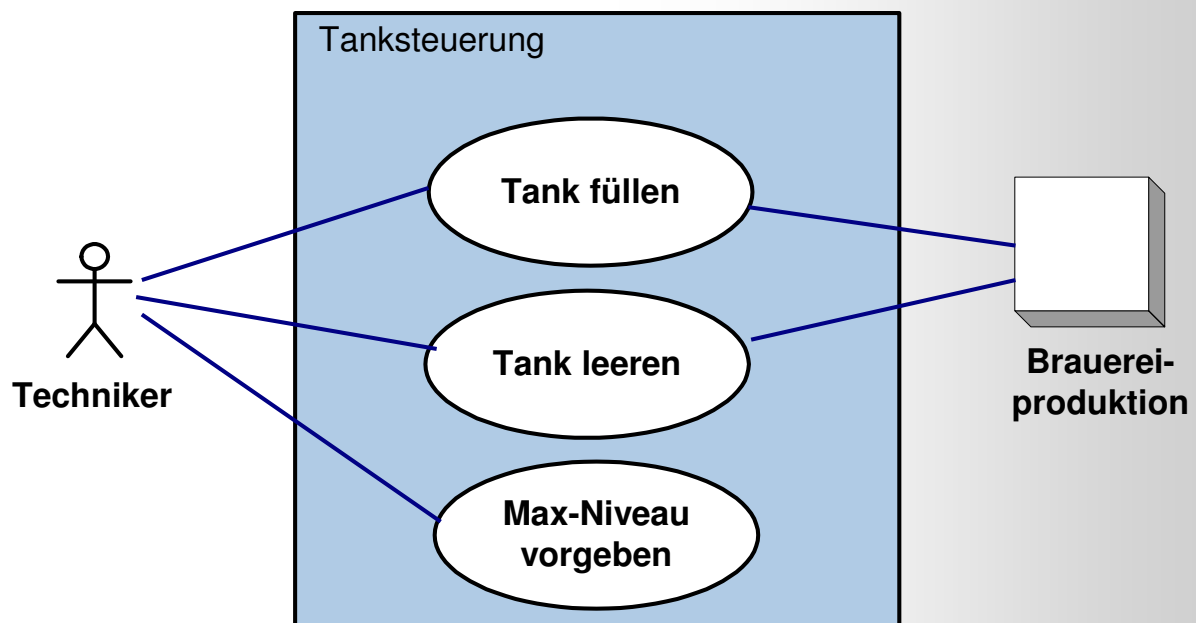
Anwendungsfall:	Tank füllen
Ziel:	Tank auf vorgegebenes Flüssigkeitsniveau füllen
Kategorie:	primär
Vorbedingung:	Tank leer
Nachbedingung Erfolg:	Füllstand auf vorgegebenem Flüssigkeitsniveau
Nachbed. Fehlschlag:	Tank nicht gefüllt, Fehlermeldung
Akteure	Techniker, Brauereiproduktion
Auslösendes Ereignis:	Eingabe Soll-Niveau
Beschreibung:	<ol style="list-style-type: none"> 1 Techniker gibt Soll-Niveau ein 2 Techniker startet Füllen des Tanks 3 Einlassventile öffnen 4 Wenn Soll-Niveau erreicht, Einlassventile schließen
Erweiterungen:	<ol style="list-style-type: none"> 1a Tank nicht leer → Fehlermeldung 1b Soll-Niveau größer Max-Niveau → Fehlermeldung 4a Soll-Niveau nicht erreicht → Fehlermeldung
Alternativen: :	<ol style="list-style-type: none"> 1a Brauereiproduktion gibt Soll-Niveau vor 2a Brauereiproduktion startet Füllen des Tanks

UML-Notation **Anwendungsfalldiagramm** (use case diagram)

Übersicht über die Beziehungen zwischen Akteuren und Anwendungsfällen



Beispiel: Anwendungsfalldiagramm Tanksteuerung

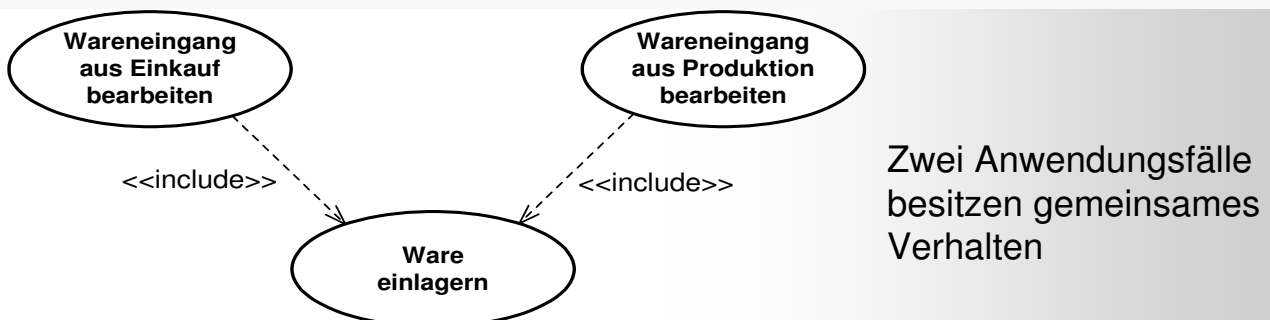


Beziehungen im Anwendungsfalldiagramm

- Extends-Beziehung



- Include-Beziehung



⇒ Vermeidung redundante Anwendungsfall-Spezifikationen

Vorführung zu §4.1 Use Case

Frage zu 4.1 (1)

Identifizieren Sie aus dem folgenden Lastenheftauszug die Akteure und Anwendungsfälle und erstellen Sie daraus ein Anwendungsfalldiagramm !

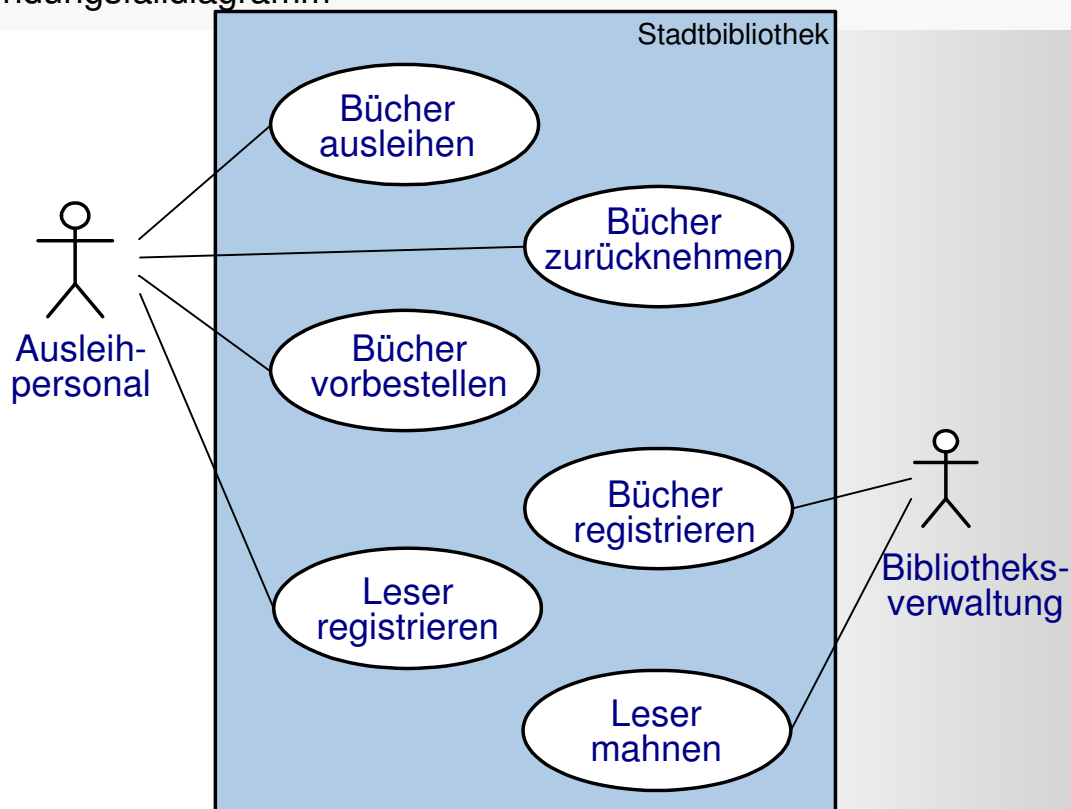
„Für eine Stadtbibliothek soll ein Softwaresystem entwickelt werden. Das Ausleihpersonal soll damit Bücher ausleihen, zurücknehmen und vorbestellen können. Weiterhin sollen Leser registriert werden. Die Bibliotheksverwaltung soll neue Bücher registrieren können und säumige Leser mahnen können.“

Antwort

- Identifizierte Akteure:
- Ausleihpersonal
 - Bibliotheksverwaltung
- Identifizierte Anwendungsfälle:
- Bücher ausleihen
 - Bücher zurücknehmen
 - Bücher vorbestellen
 - Leser registrieren
 - Bücher registrieren
 - Leser mahnen

**Frage zu 4.1 (2)**

Anwendungsfalldiagramm



Kapitel 4 Dynamische Konzepte in der objektorientierten Analyse

4.1 Anwendungsfall

4.2 Botschaft

4.3 Szenario

4.4 Zustandsautomat

4.5 Zusammenfassung der objektorientierten Konzepte (zum Selbststudium)

4.6 Zusammenfassung

Der Begriff der Botschaft

Definition:

Eine **Botschaft** (**message**, auch: **Nachricht**) ist die Aufforderung eines Senders (client) an einen Empfänger (server, supplier) eine Dienstleistung zu erbringen.

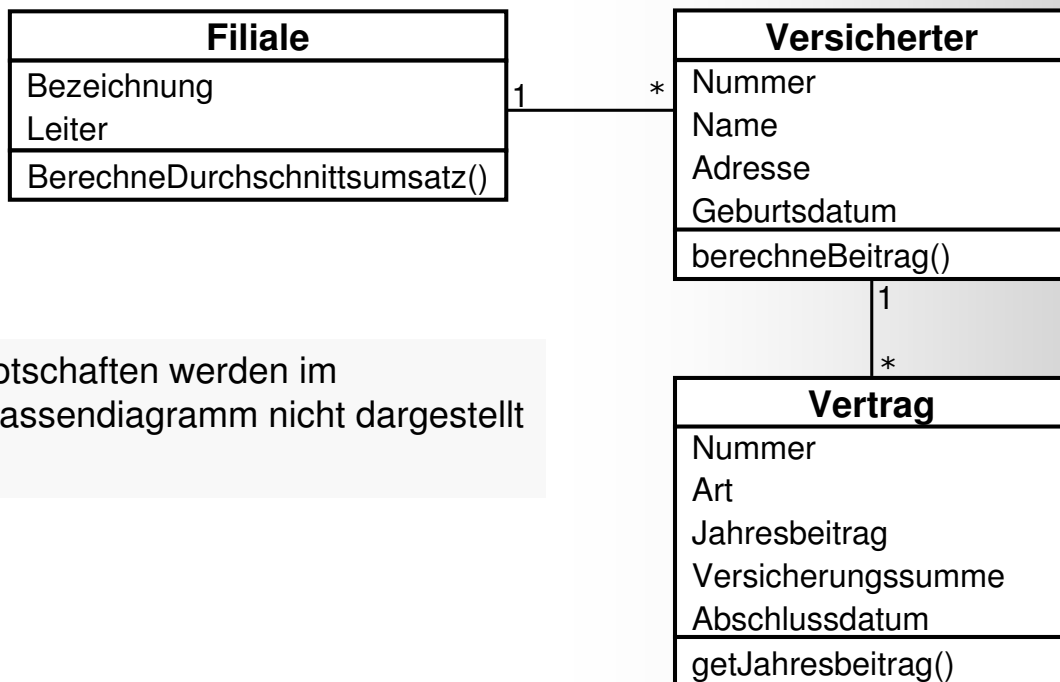
- Empfänger interpretiert diese Botschaft und führt eine Operation (gleichen Namens) aus
- Sender der Botschaft weiß nicht, wie die entsprechende Operation ausgeführt wird
- Schnittstelle der Klasse = Menge der Botschaften, auf die Objekte dieser Klasse reagieren können
- Botschaften werden in verschiedenen UML Diagrammarten verwendet

Objekte kommunizieren untereinander über Botschaften

Beispiel für Botschaften (1)

Berechnung des Durchschnittsumsatzes für jede Filiale einer Versicherung:

- Um für jeden Versicherten die Beitragssumme zu ermitteln, muss der Jahresbeitrag eines jeden Vertrags dieses Versicherten bekannt sein.

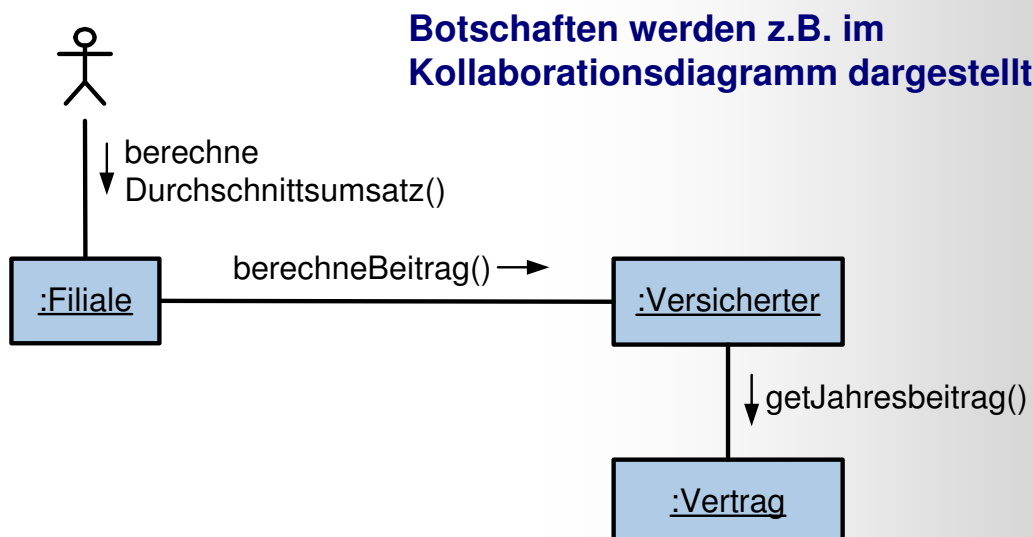


⇒ Botschaften werden im Klassendiagramm nicht dargestellt

Beispiel für Botschaften (2)

Berechnung des Durchschnittsumsatzes für jede Filiale einer Versicherung:

- Wenn die Filiale die Botschaft `berechneDurchschnittsumsatz()` erhält, dann sendet sie jedem ihrer Versicherten die Botschaft `berechneBeitrag()`, die wiederum die Botschaft `getJahresbeitrag()` an alle ihre Vertragsobjekte schickt.



Frage zu 4.2

Welche Aussagen zur Verwendung von Botschaften sind richtig?

- ☒ Beliebige Objekte können über Botschaften kommunizieren
- ☒ Zum Austausch von Botschaften muss eine Objektverbindung, aber keine Assoziation zwischen Sender und Empfänger bestehen
- f** ☐ Botschaften sind die Basis für Objektdiagramme
- ☒ Eine Botschaft löst eine Operation gleichen Namens aus
- ☒ Objekte einer Klasse können nur auf die Botschaften reagieren, die in der Schnittstelle der Klasse beschrieben sind

**Kapitel 4 Dynamische Konzepte in der
objektorientierten Analyse**

4.1 Anwendungsfall

4.2 Botschaft

4.3 Szenario

4.4 Zustandsautomat

4.5 Zusammenfassung der objektorientierten Konzepte (zum Selbststudium)

4.6 Zusammenfassung

Was ist ein Szenario?

Definition:

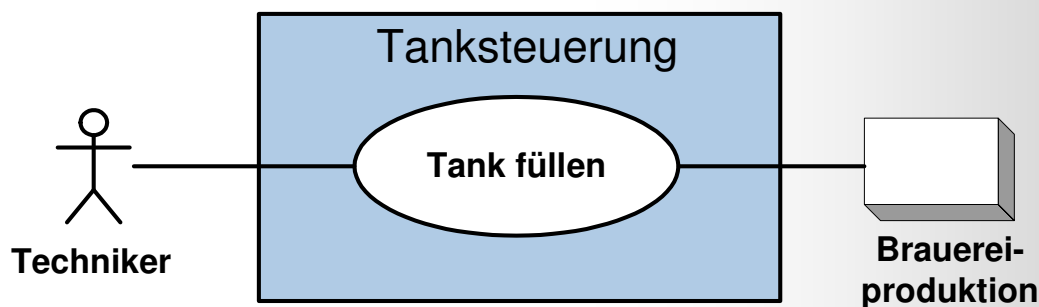
Ein **Szenario (scenario)** ist eine Sequenz von Verarbeitungsschritten, die unter bestimmten Bedingungen auszuführen ist.

- Die Verarbeitungsschritte sollen das Hauptziel des Akteurs realisieren und ein entsprechendes Ergebnis liefern
- Szenarios beginnen mit dem auslösenden Ereignis und werden fortgesetzt, bis das Ziel erreicht ist oder aufgegeben wird

Anwendungsfall und Szenarios

- Ein Anwendungsfall wird durch eine Menge von Szenarios dokumentiert
- Jedes Szenario wird durch eine oder mehrere Bedingungen definiert, die zu einem speziellen Ablauf des Anwendungsfalles führen
- Zwei Kategorien von Szenarios:
 - Szenarios, die eine erfolgreiche Bearbeitung des Anwendungsfalles beschreiben
 - Szenarios, die zu einem Fehlschlag führen

Beispiele für Szenarios zum Anwendungsfall Tank füllen



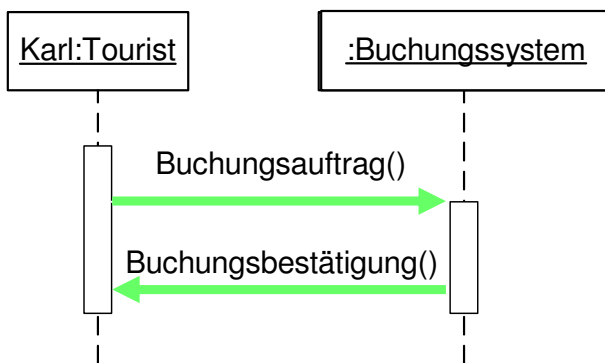
Mögliche Abläufe des Anwendungsfalles

- Tank füllen durch Techniker
- Tank füllen durch Brauereiproduktion
- Tank füllen und Tank nicht leer
- Tank füllen und Soll-Niveau wird nicht erreicht
- ...

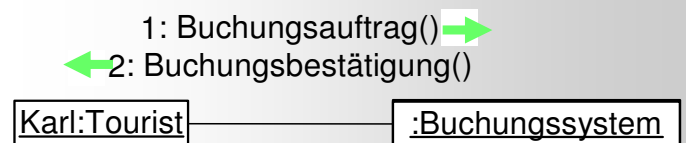
UML-Notation von Szenarios

- **Szenarios** werden durch **Interaktionsdiagramme** (interaction diagrams) dargestellt
- UML bietet zwei Arten von Interaktionsdiagrammen an
 - **Sequenzdiagramm** (sequence diagram)
 - **Kollaborationsdiagramm** (collaboration diagram)

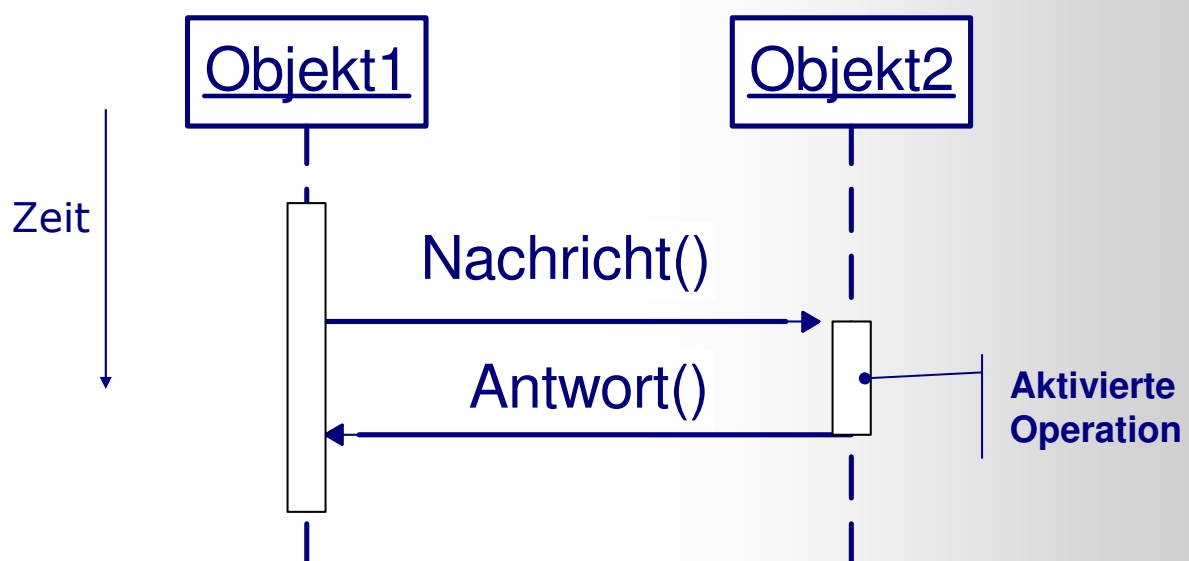
Sequenzdiagramm



Kollaborationsdiagramm



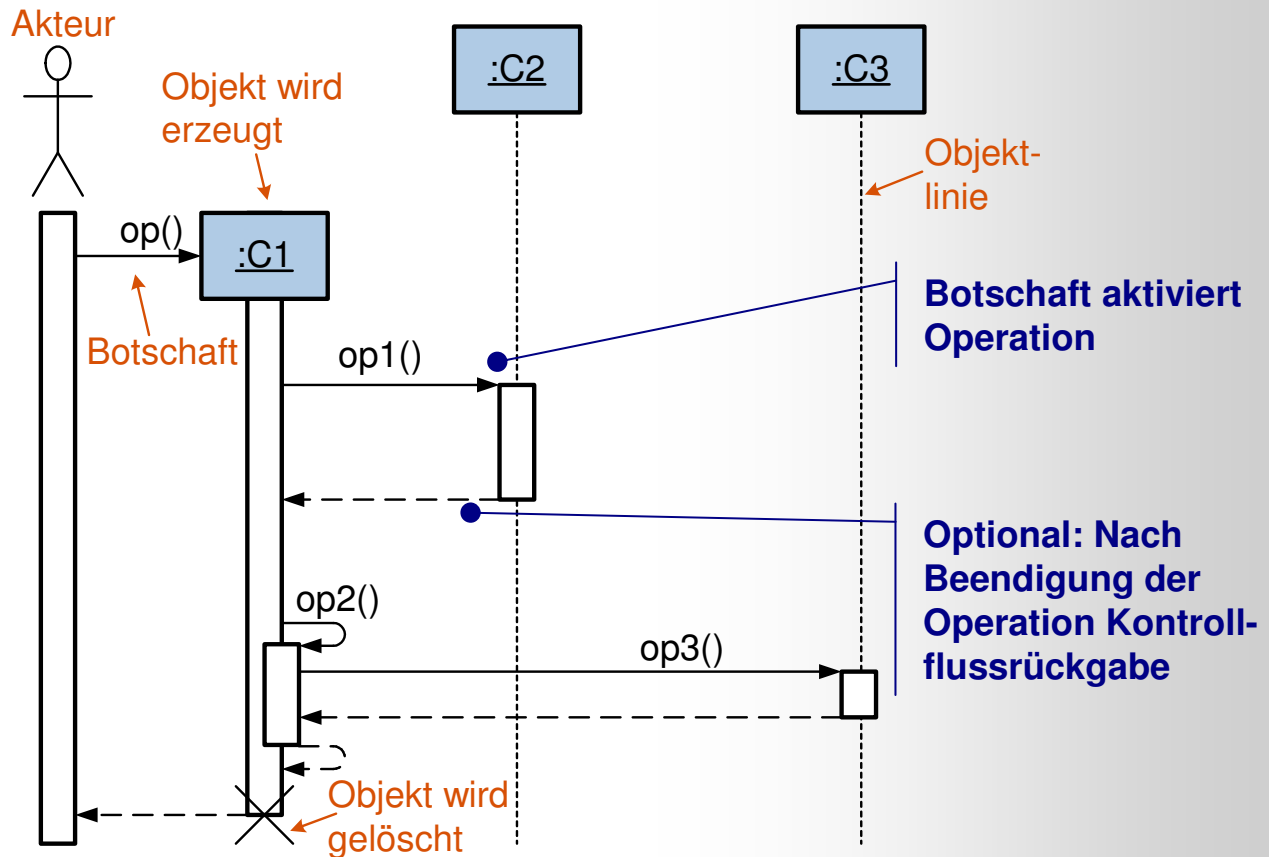
Sequenzdiagramm (Tafelanschrieb)



Erweiterungen:

- **Bedingung (condition):** [**<Bedingung>**] **Operation()**
- **Wiederholung (iteration):** ***Operation()** *oder* ***[<Bedingung>] Operation()**

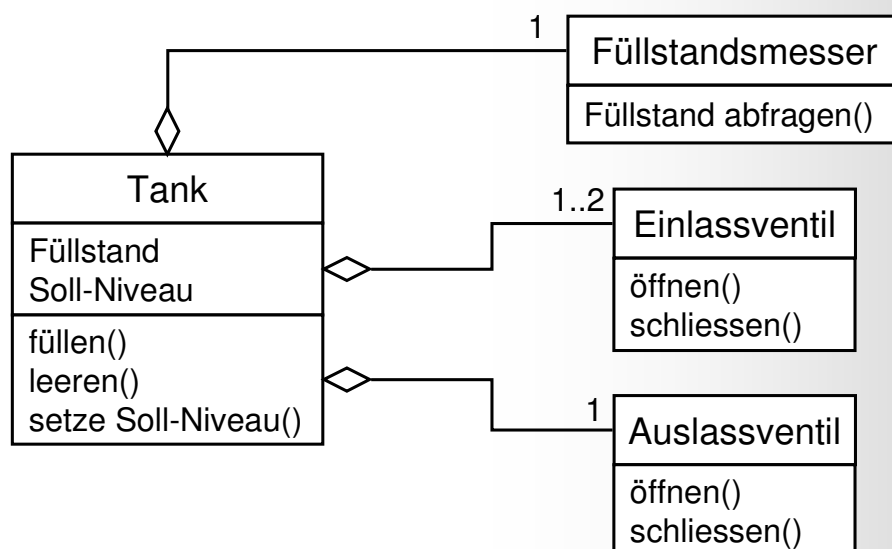
UML-Notation Sequenzdiagramm



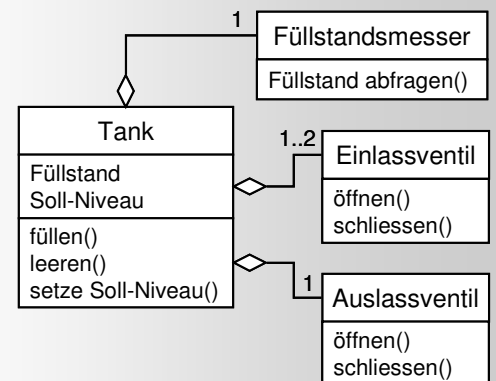
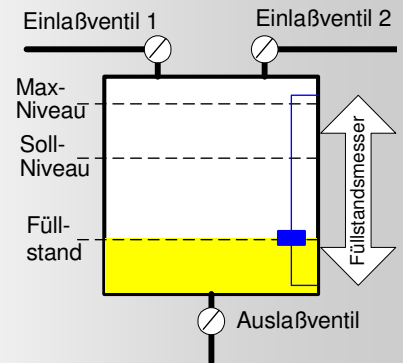
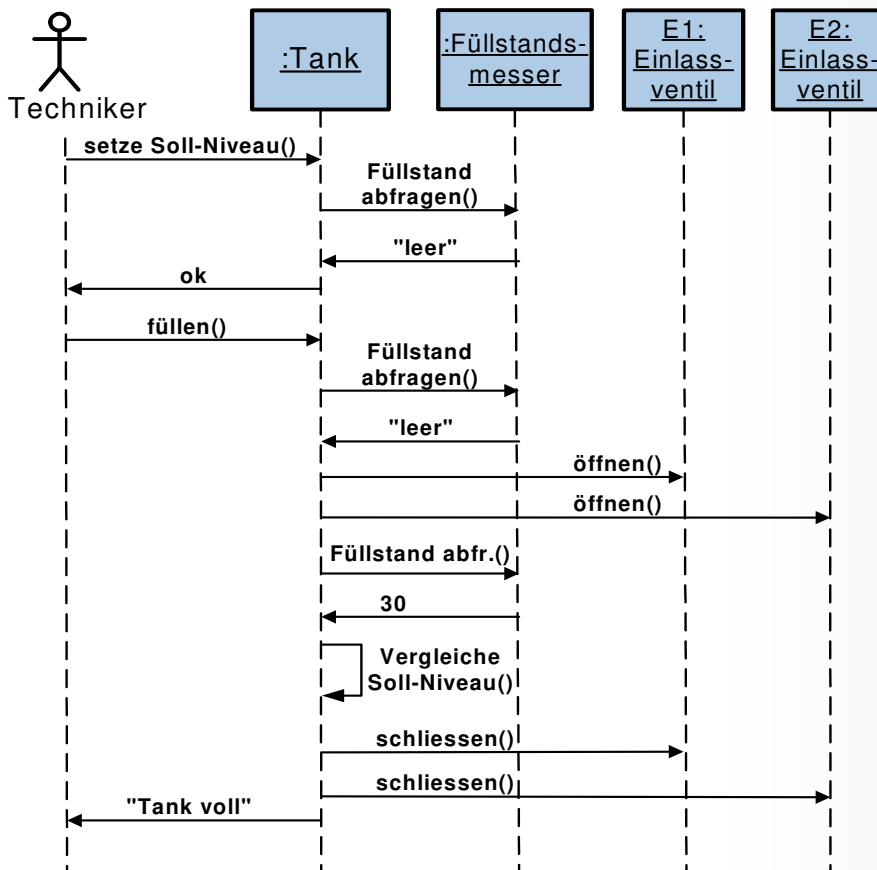
Beispiel: Szenario für den Anwendungsfall Tank füllen

– Ausgangssituation Klassendiagramm

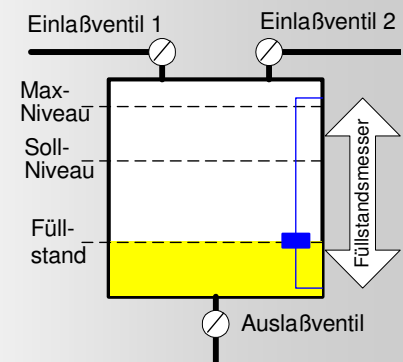
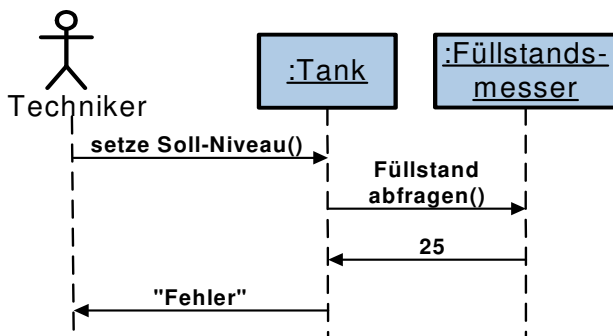
Klassendiagramm stellt nur die Struktur und keine Dynamik dar !



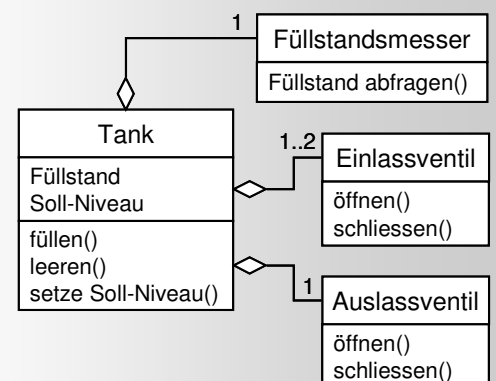
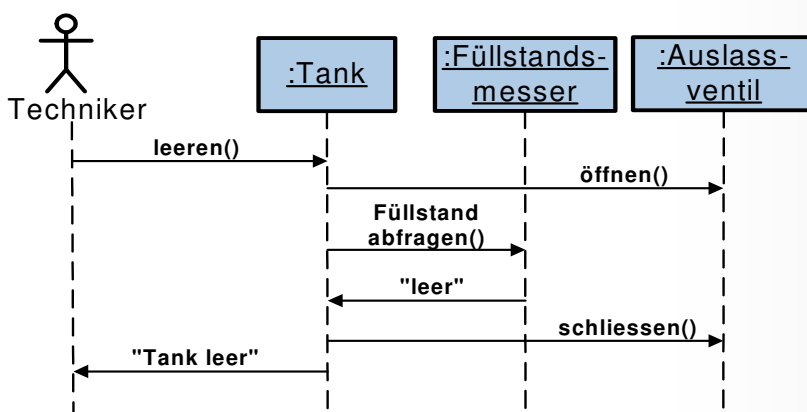
Sequenzdiagramm Tank füllen durch Techniker



Sequenzdiagramm Tank füllen und Tank nicht leer



Sequenzdiagramm Tank leeren



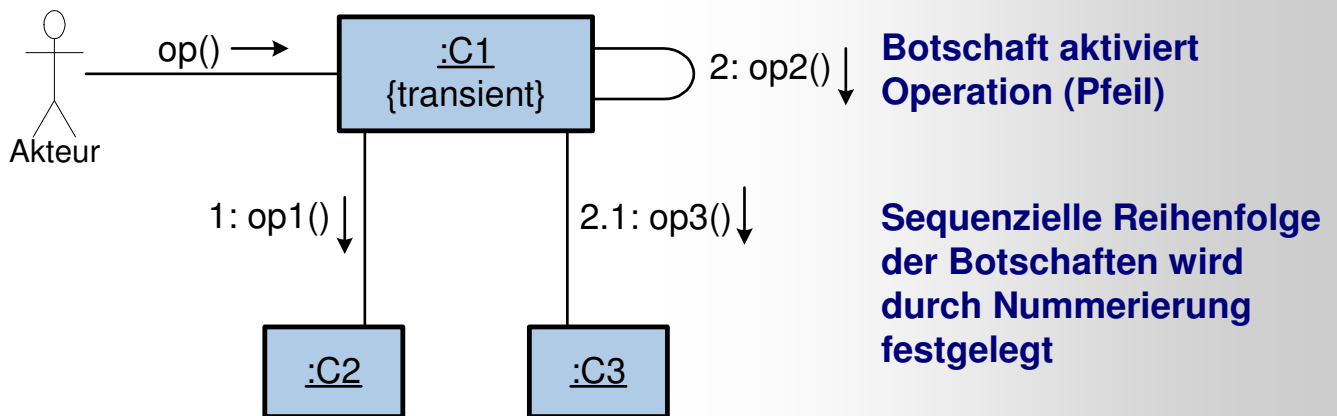
Konsistenzregeln zwischen Sequenzdiagramm und Klassendiagramm

- Botschaften, die an ein Objekt einer Klasse gesendet werden, müssen im Klassendiagramm in der Operationsliste dieser Klasse enthalten sein.
- Verwaltungsoperationen werden im Sequenzdiagramm zusätzlich eingetragen, um die Kommunikation zwischen den Objekten vollständig zu beschreiben, obwohl sie im Klassendiagramm nicht explizit modelliert werden müssen.
 - Bsp: setze Soll-Niveau()

Kollaborationsdiagramm (collaboration diagram)

- Alternative zum Sequenzdiagramm
- Beschreibt Objekte und deren Zusammenarbeit
- Über Objektverbindungen können Botschaften gesendet werden
- Permanente Objektverbindungen
 - Assoziationen
- Temporäre Objektverbindungen
 - bestehen nur für die Dauer der Kommunikation
 - liegen vor, wenn das angesprochene Empfängerobjekt auch ohne Vorliegen einer Assoziation vom Sender eindeutig identifiziert werden kann
 - werden mit Stereotyp <<temp>> gekennzeichnet
- Implizite Objektverbindung (self link)
 - Jedes Objekt kann jederzeit Botschaften an sich selbst senden

UML-Notation Kollaborationsdiagramm



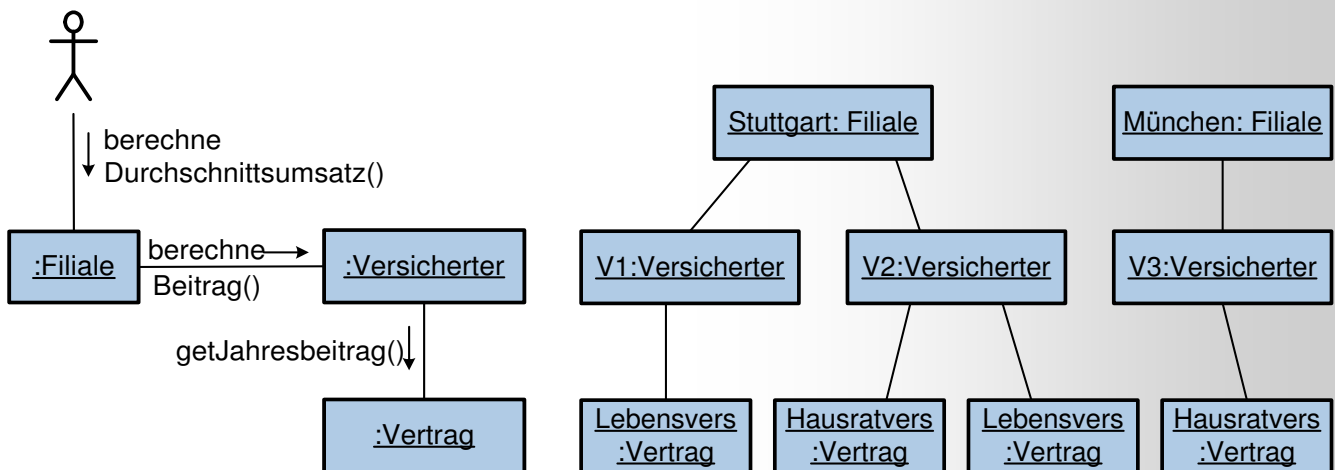
Merkmale:

- {new} Objekt wird erzeugt
- {destroyed} Objekt wird gelöscht
- {transient} Objekt wird sowohl erzeugt als auch gelöscht

Bezeichnung von Objekten in Sequenz- und Kollaborationsdiagrammen

O	Objekt O
O : C	Objekt O der Klasse C
O / R	Objekt O , das die Rolle R spielt
O / R : C	Objekt O der Klasse C , das die Rolle R spielt
: C	Unbenanntes Objekt der Klasse C
/ R	Unbenanntes Objekt, das die Rolle R spielt
/ R : C	Unbenanntes Objekt der Klasse C , das die Rolle R spielt

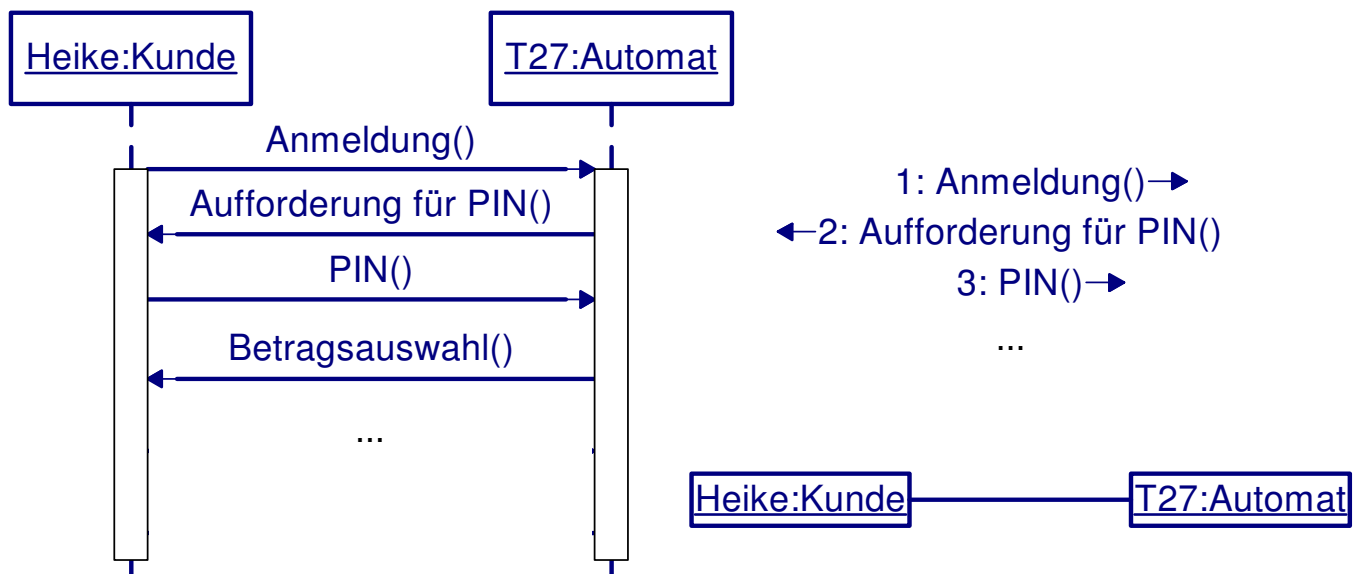
Vergleich Kollaborations- und Objektdiagramm



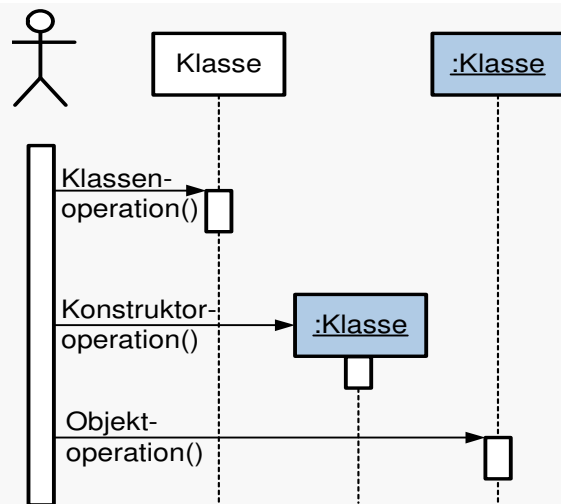
Kollaborationsdiagramm:
Objekt ist Platzhalter für
beliebiges Objekt der Klasse

Objektdiagramm:
Darstellung konkreter Objekte

Vergleich Sequenz- und Kollaborationsdiagramm (Tafelanschrieb)

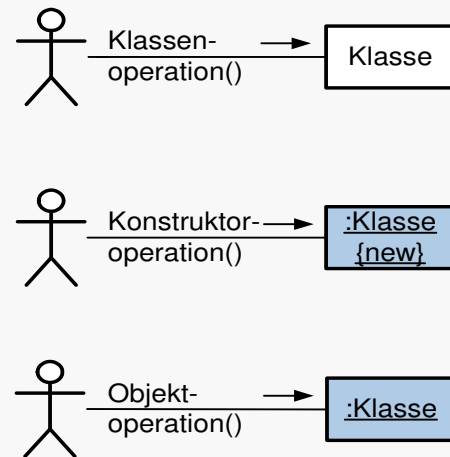


Vergleich Sequenz- und Kollaborationsdiagramm



- Sequenzdiagramm:
Betont die zeitlichen Aspekte
des dynamischen Verhaltens

⇒ **Gut geeignet zur Beschreibung
komplexer Szenarios**



- Kollaborationsdiagramm:
Betont die Verbindungen
zwischen Objekten

⇒ **Gut geeignet zur Beschreibung
der Wirkung komplexer Operationen**

Vorführung zu §4.3 Sequence Diagram and Collaboration Diagram



Frage zu 4.3

Wie wird die Reihenfolge der Botschaften im Sequenzdiagramm festgelegt ?

Wie wird die Reihenfolge der Botschaften im Kollaborationsdiagramm festgelegt ?

Erklären Sie wie Kontrollfluss bei einer Botschaft und der damit ausgelösten Operation im Sequenzdiagramm dargestellt wird ?

Antwort

Die Reihenfolge der Botschaften im Sequenzdiagramm ergibt sich aus der vertikalen Reihenfolge von oben nach unten.

Die Reihenfolge der Botschaften im Kollaborationsdiagramm wird durch hierarchische Nummerierung festgelegt.

Der Kontrollfluss geht durch das Senden einer Botschaft vom sendenden Objekt auf das empfangende Objekt über. Das empfangende Objekt startet eine Operation, deren Lebensdauer durch ein schmales Rechteck auf der Lebenslinie des Objekts dargestellt wird. Am Ende der Operation zeigt ein gestrichelter Pfeil auf das sendende Objekt zurück. Damit geht der Kontrollfluss an die Operation des sendenden Objekts zurück.



Kapitel 4 Dynamische Konzepte in der objektorientierten Analyse

4.1 Anwendungsfall

4.2 Botschaft

4.3 Szenario

4.4 Zustandsautomat

4.5 Zusammenfassung der objektorientierten Konzepte (zum Selbststudium)

4.6 Zusammenfassung

Was ist ein Zustandsautomat?

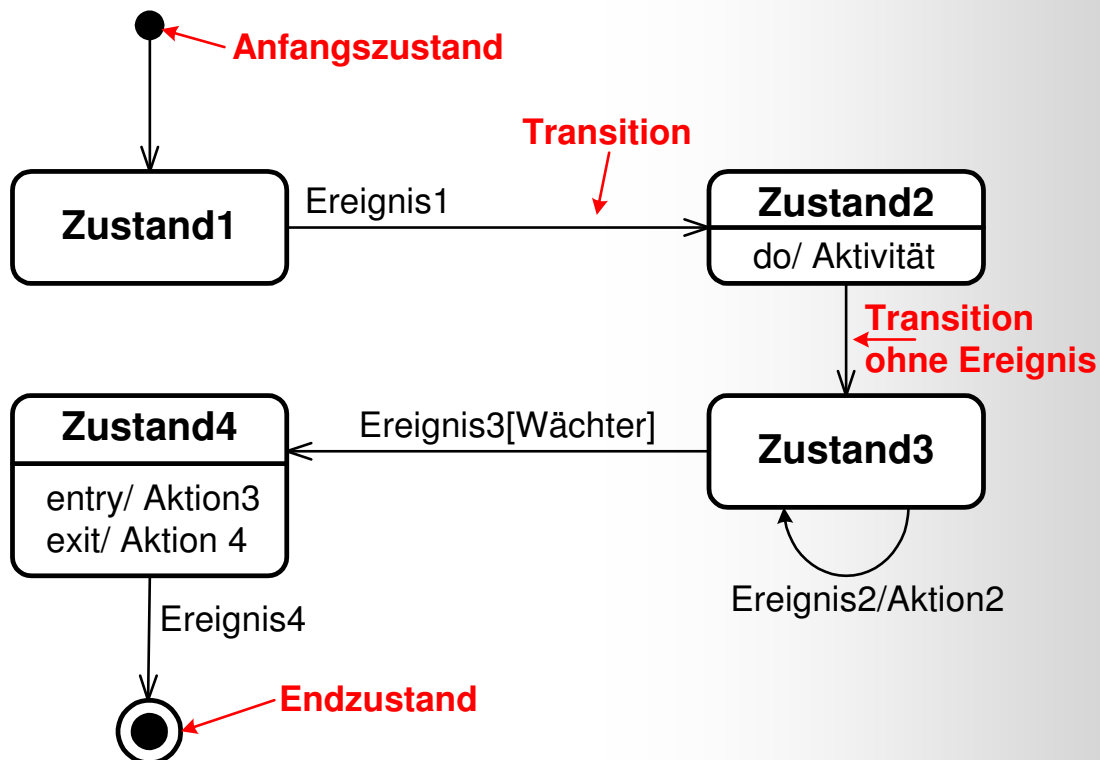
Definition:

Ein **Zustandsautomat** (**finite state machine**) besteht aus Zuständen und Zustandsübergängen (Transitionen).

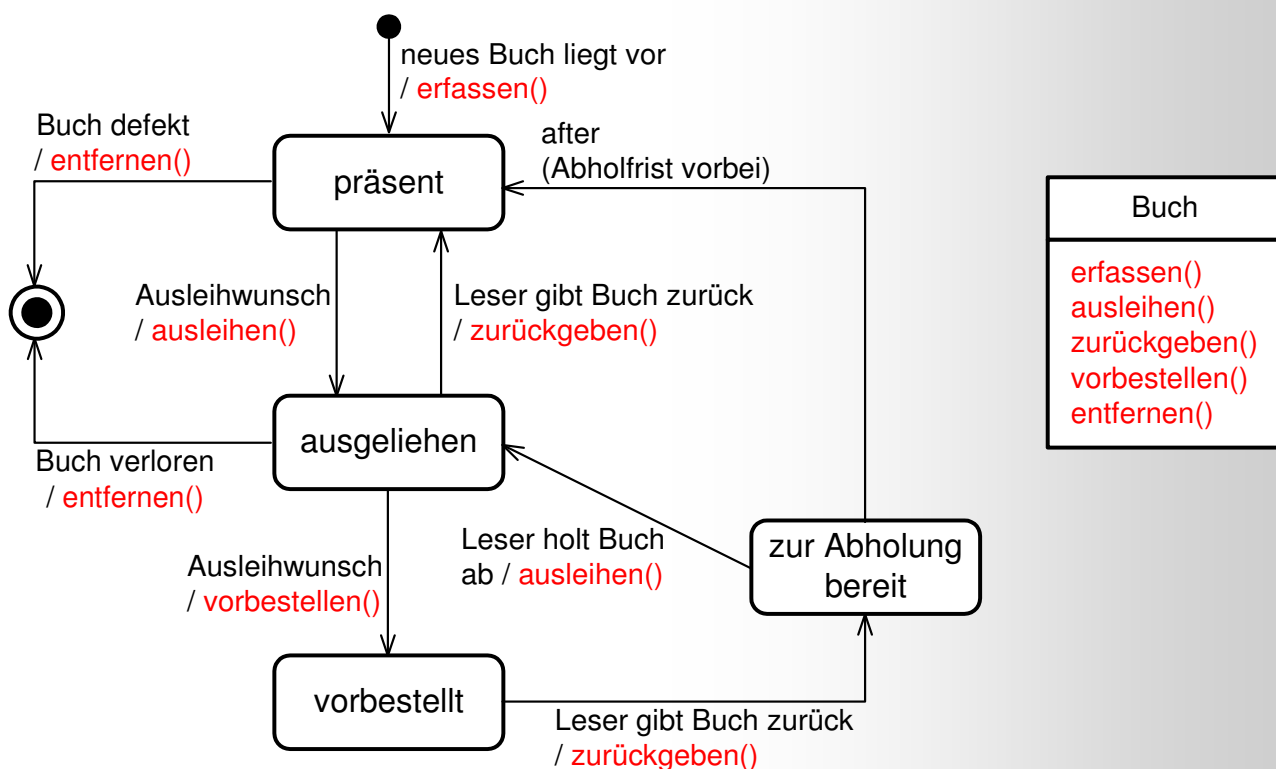
- Ein Zustand ist eine Zeitspanne, in der ein Objekt auf ein Ereignis wartet
- Zustandsübergänge werden durch Ereignisse ausgelöst
- Ein Ereignis tritt immer zu einem Zeitpunkt auf und besitzt keine Dauer
- Zustandsautomaten werden durch Zustandsdiagramme (statechart diagrams) dargestellt.
- Ein Spezialfall der Zustandsdiagramme sind die Aktivitätsdiagramme, die bei Zuständen mit viel Verarbeitung verwendet werden.

Beschreibung des Lebenszyklusses (dynamisches Verhalten) von Objekten

UML-Notation Zustandsdiagramm (statechart diagram)



Beispiel: Lebenszyklus der Klasse Buch



Zustand (1)

- Zustand
 - Zustandsname ist optional
 - Zustände ohne Namen heißen anonyme Zustände und sind alle voneinander verschieden
 - Zustandsname soll kein Verb sein
 - Innerhalb einer Klasse muss jeder Zustandsname eindeutig sein
- Anfangszustand
 - Pseudozustand, der mit einem echten Zustand durch eine Transition verbunden ist
- Endzustand
 - Kein weiteres Ereignis kann mehr folgen
 - Objekt hört auf zu existieren

Zustand

● Anfangszustand

● Endzustand

Zustand (2)

- Verarbeitung in einem Zustand
 - Aktion
 - wird durch ein Ereignis aktiviert
 - ist atomar
 - terminiert selbstständig
 - entry-Aktion **Löst automatisch beim Eintritt in den Zustand aus**
 - exit-Aktion **Löst automatisch beim Verlassen des Zustandes aus**
 - Aktivität
 - beginnt, wenn Objekt einen Zustand einnimmt und endet, wenn es ihn verlässt
 - Zugehöriges Ereignis: do

Zustand

Ereignis / Aktionsbeschr.

Zustand

do / Aktivitätsbeschr.

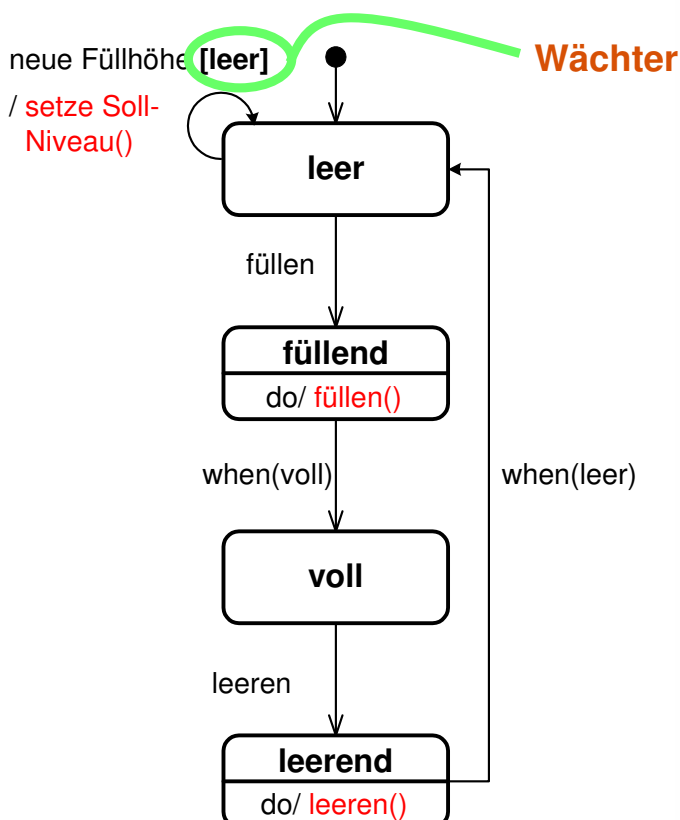
Zustandsübergänge und Ereignisse

- Zustandsübergang (Transition)
 - Verbindet zwei Zustände
 - Wird durch ein Ereignis ausgelöst
 - Kann mit einer Aktion verbunden sein
- Ereignis kann sein
 - Bedingung, die wahr wird, z.B. `when (Temperatur > 100 Grad)`
 - Signal, z.B. `rechte Maustaste gedrückt`
 - Botschaft (Aufruf einer Operation)
 - Eintreten eines bestimmten Zeitpunkts, z.B. `when (01.01.2000)`
 - Verstrichene Zeit, z.B. `after (10 sec)`
- Ereignis kann mit Wächter (guard condition) kombiniert werden
 - Die Transition feuert nur dann, wenn ...
 - das zugehörige Ereignis eintritt und
 - die im Wächter spezifizierte Bedingung erfüllt ist

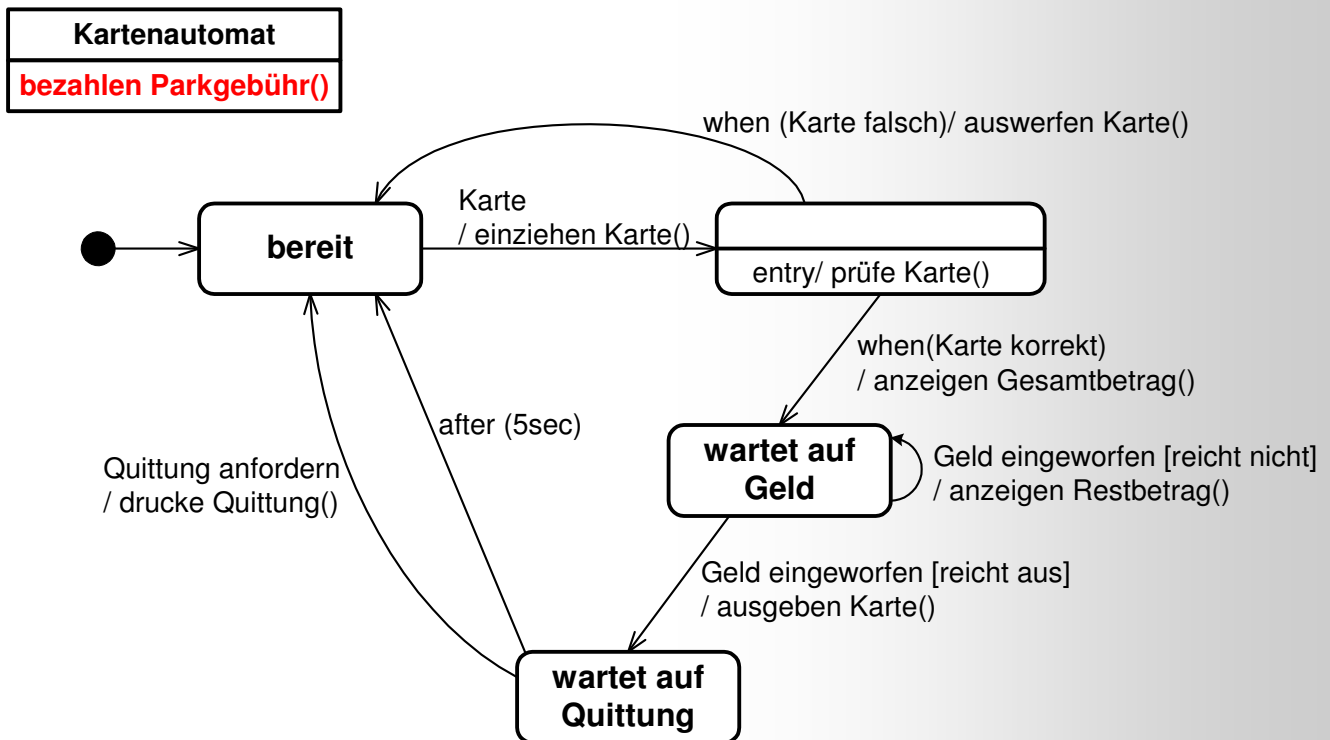
Gesprochen:

Die Transition „feuert“

Beispiel für ein Zustandsdiagramm des Lebenszyklusses einer Klasse: Klasse Tank



Beispiel für ein Zustandsdiagramm einer Klasse mit einer komplexen Operation: Kartenautomat in einem Parkhaus



Anfangszustand ist notwendig, Endzustand ist optional

Verfeinerung von Zuständen

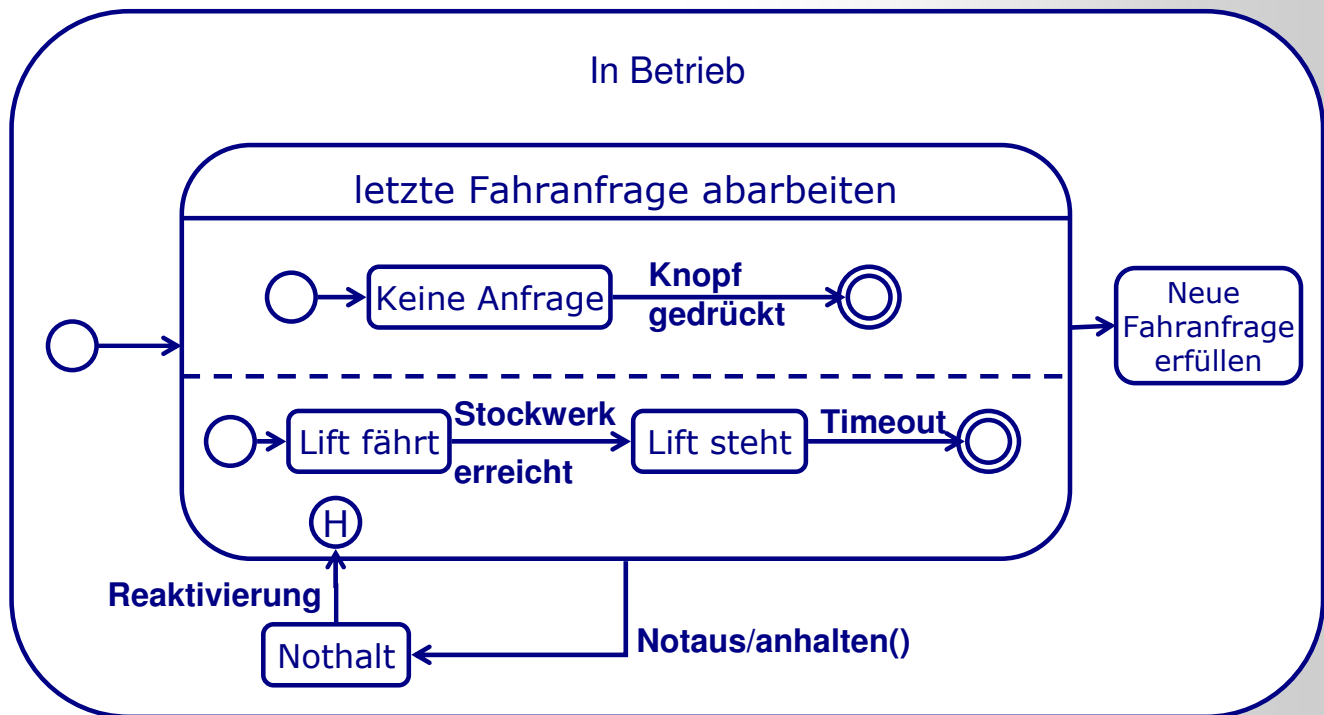
- Ein Zustand kann durch Unterzustände (substates) verfeinert werden
- Alle Unterzustände schließen sich gegenseitig aus
- Mehrere Unterzustände können auch parallel ablaufen
- Ein Zustand, der verfeinert wird, heißt auch zusammengesetzter Zustand
- Eine Transition in einen verfeinerten Zustand entspricht der Transition in den Anfangszustand der Verfeinerung
- Das Verlassen des verfeinerten Zustandes wird im Zustandsdiagramm durch den Endzustand angezeigt

Historienzustand

- Spezieller Anfangszustand in einem zusammengesetzten Zustand (H)
- „Gedächtnis“, welcher Unterzustand zuletzt eingenommen wurde
- Bei Wiedereintritt in zusammengesetzten Zustand automatisch Übergang in letzten Unterzustand

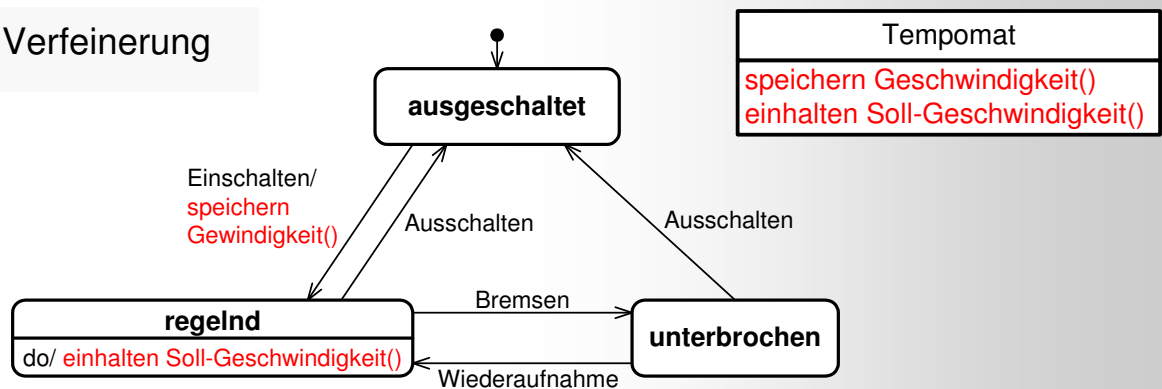
Verfeinerung von Zuständen (Tafelanschrieb)

– Beispiel Lift

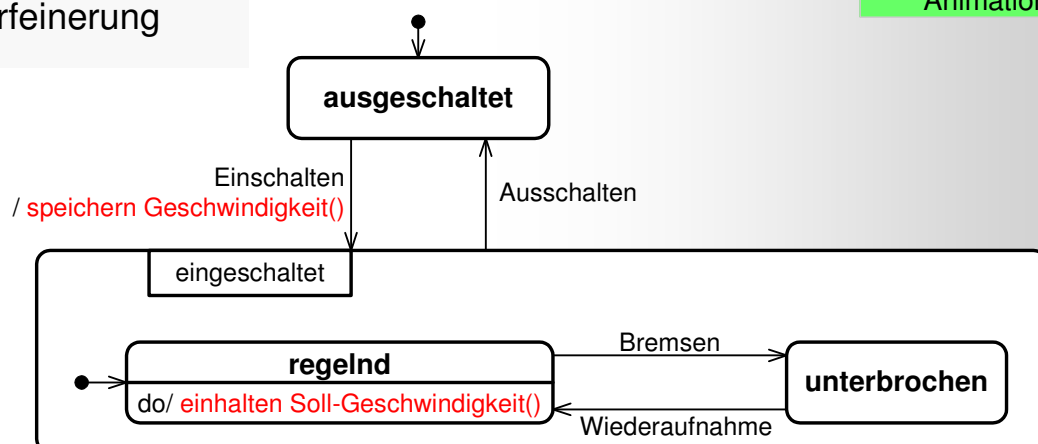


Beispiel für Zustandsdiagramm mit Verfeinerung: Tempomat

– Ohne Verfeinerung



– Mit Verfeinerung



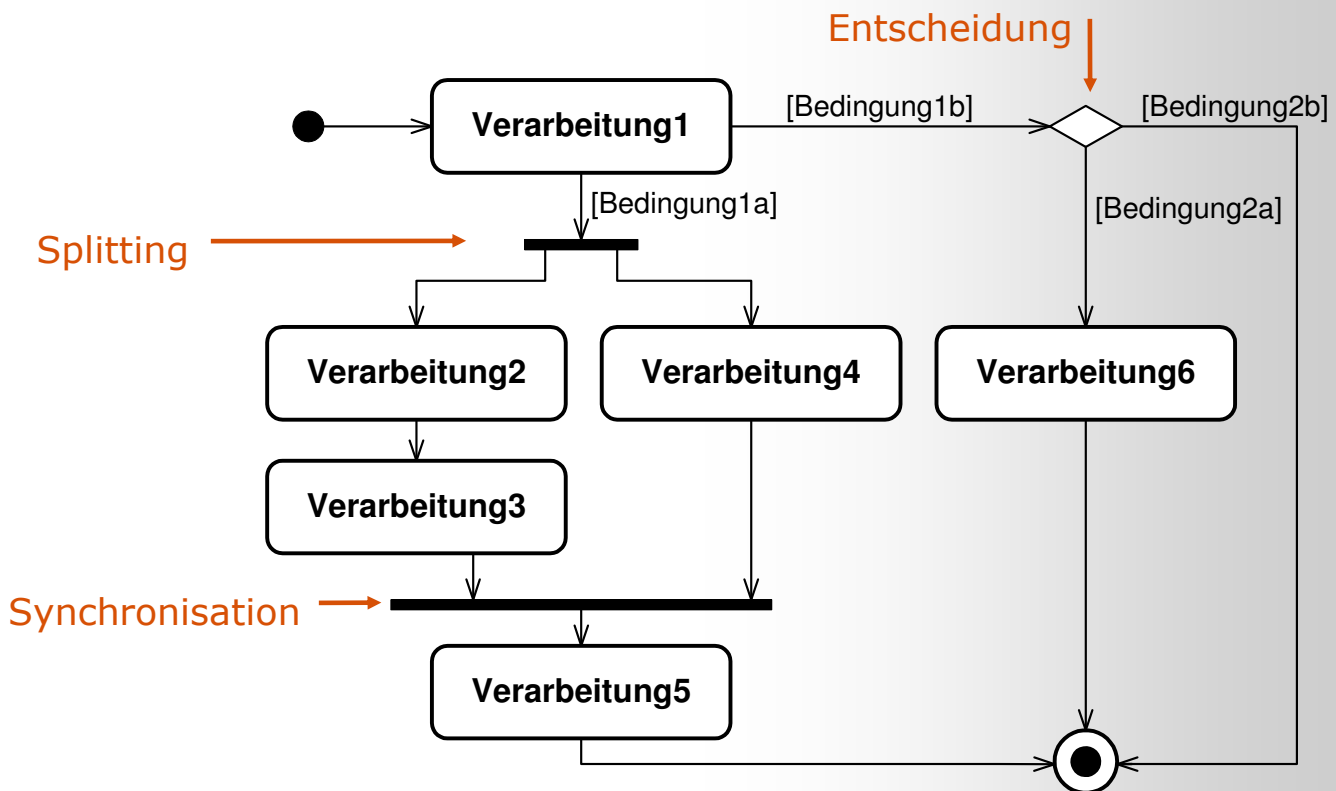
Konsistenzregeln zwischen Zustandsdiagramm und Klassendiagramm

- Als Aktionen und Aktivitäten sind nur Operationen der jeweiligen Klasse zulässig.
- Operationsnamen werden in der Form `Operation()` bei Aktionen und Aktivitäten eingetragen.
- Wenn eine Operation in mehreren Zuständen aktiviert werden kann, so kann sie in Abhängigkeit vom jeweiligen Zustand eine unterschiedliche Wirkung besitzen.
- Erhält ein Objekt in einem Zustand einen Operationsaufruf, wobei diese Operation weder als Aktivität noch als Aktion zur Verfügung steht, dann besitzt die Botschaft keine Wirkung, d.h. das Objekt tut nichts.

Aktivitätsdiagramm (activity diagram)

- Sonderfall des Zustandsdiagrammes
- (Fast) alle Zustände sind mit Verarbeitung verbunden
- Eine Aktivität ist ein einzelner Schritt in einem Verarbeitungsablauf
- Zustand wird verlassen, wenn Verarbeitung beendet ist
- Wächter (guard condition) spezifiziert Verzweigungen im Kontrollfluss
- Spezifikation von parallelen Abläufen
- Aktivitäten oder Aktivitätsdiagramme sind zugeordnet zu
 - einem Anwendungsfall **Im Gegensatz zu Zustandsdiagrammen**
 - einer Klasse
 - einer Operation

UML-Notation Aktivitätsdiagramm

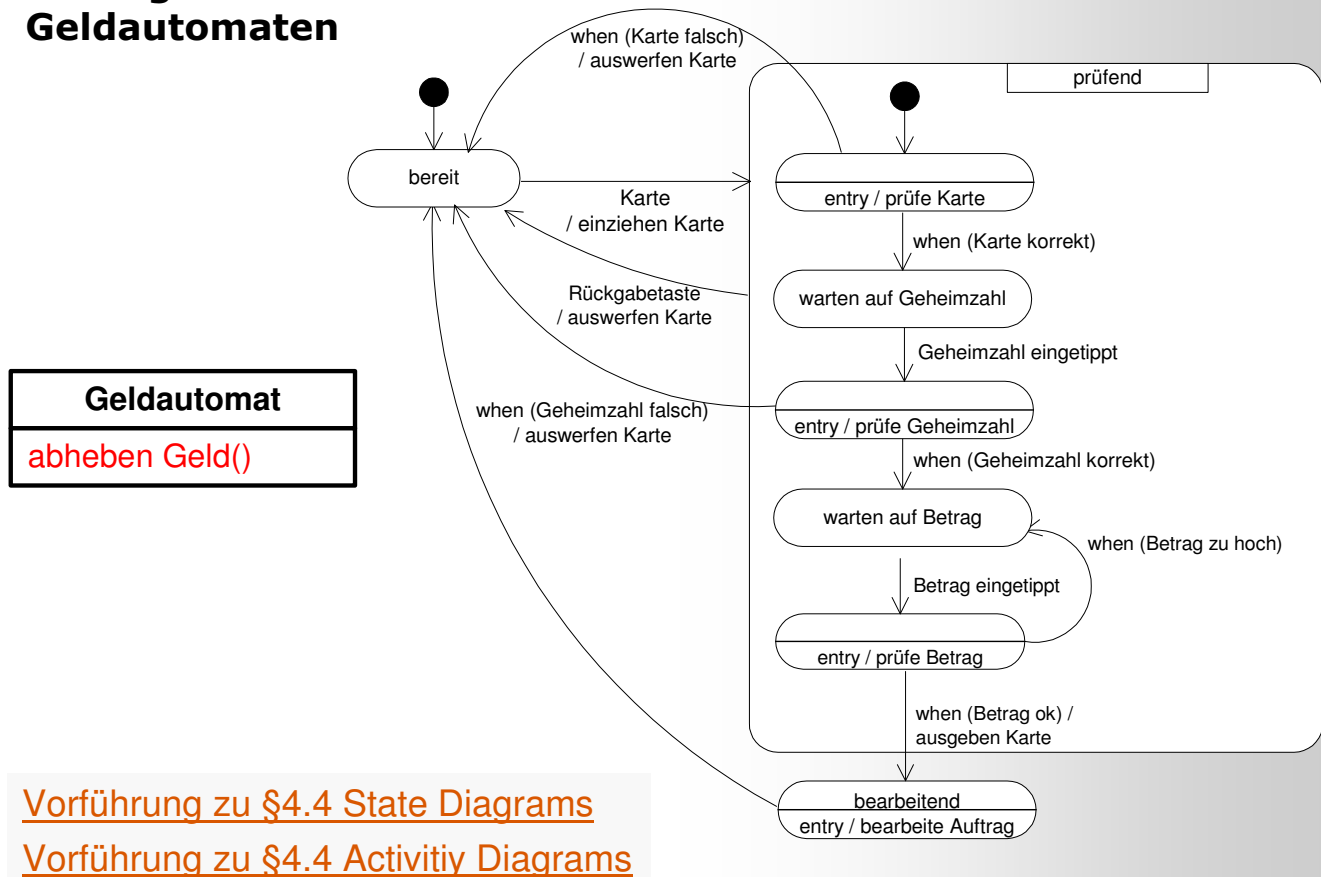


Spezialfall des Zustandsdiagramms

Aufgabe: Zustandsdiagramm für das Abheben von Geld an einem - vereinfachten - Geldautomaten

- Ausgangszustand >>bereit<<.
- Bei Karteneingabe wird Karte eingezogen
- Sofortiger Auswurf falscher Karten und Übergang in bereit
- Karte korrekt → Automat wartet auf Eingabe der Geheimzahl
- Falsche Geheimzahl → Abbruch, Auswurf Karte und Übergang in bereit
- Korrekte Geheimzahl → Automat wartet auf Betragseingabe
- Bei zu hohem Betrag erneute Eingabe
- Bei korrektem Betrag → Kartenausgabe anschließend Bearbeitung des Kundenauftrags
- Anschließend Übergang in bereit
- Jederzeit Betätigung der Rückgabetaste möglich (solange Auftrag noch nicht in Bearbeitung) → Auswurf Karte und Übergang in bereit

Lösung: Abheben von Geld an einem - vereinfachten - Geldautomaten



Frage zu 4.4

Welche Aussagen zu Zustandsdiagrammen sind richtig?

- ☒ Zustandsautomaten beschreiben den Lebenszyklus eines Objektes bzw. einer komplexen Operation
- ☐ Jedes Objekt einer Klasse besitzt einen anderen Zustandsautomaten
- ☒ Bei der Erzeugung eines Objekts wird der Pseudozustand Anfangszustand eingenommen
- ☐ Zustandsübergänge mit Wächter sind nur von der Wächterbedingung abhängig
- ☒ Bei der Löschung eines Objekts geht der Zustandsautomat in den Pseudozustand Endzustand, sofern dieser existiert



Kapitel 4 **Dynamische Konzepte in der objektorientierten Analyse**

4.1 Anwendungsfall

4.2 Botschaft

4.3 Szenario

4.4 Zustandsautomat

**4.5 Zusammenfassung der objektorientierten Konzepte
(zum Selbststudium)**

4.6 Zusammenfassung

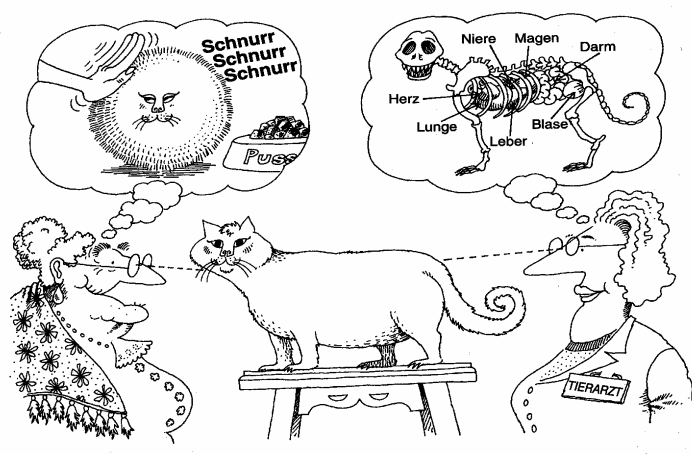
Zusammenfassung der objektorientierten Konzepte

Die wesentlichen Merkmale der Objektorientierung sind

- Abstraktion
 - Vorgehensweise, bei der wesentliche Details ermittelt und unwesentliche ignoriert werden
 - Auch ein Modell oder ein bestimmter Blickwinkel kann als Abstraktion bezeichnet werden
- Kapselung
 - Zusammengehörende Attribute und Operationen sind in einer Einheit – der Klasse – gekapselt
- Assoziation
- Aggregation
 - „hat ein...“ -Beziehung
- Vererbung
 - „ist ein...“ -Beziehung

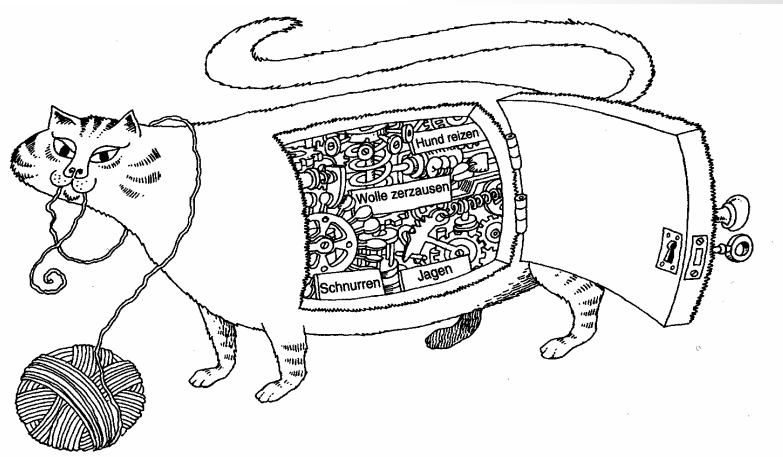
Abstraktion

- Abstraktion zur Bewältigung der Komplexität
- Konzentration auf die wesentlichen Charakteristika bei der Beschreibung eines Gegenstandes oder Sachverhaltes
- relativ zur Perspektive des Betrachters (Problembereich)
- die formale Beschreibung eines Gegenstandes oder Sachverhaltes entspricht einer **Klasse**. Der Gegenstand oder Sachverhalt selbst ist ein **Objekt**.



Kapselung

- ein Objekt ist gekennzeichnet durch Attribute und Verhalten
- von aussen kann ein Objekt nicht manipuliert werden
- verbirgt die Details der Implementierung eines bestimmten Verhaltens
- trennt somit die Schnittstelle einer Abstraktion von ihrer Implementierung



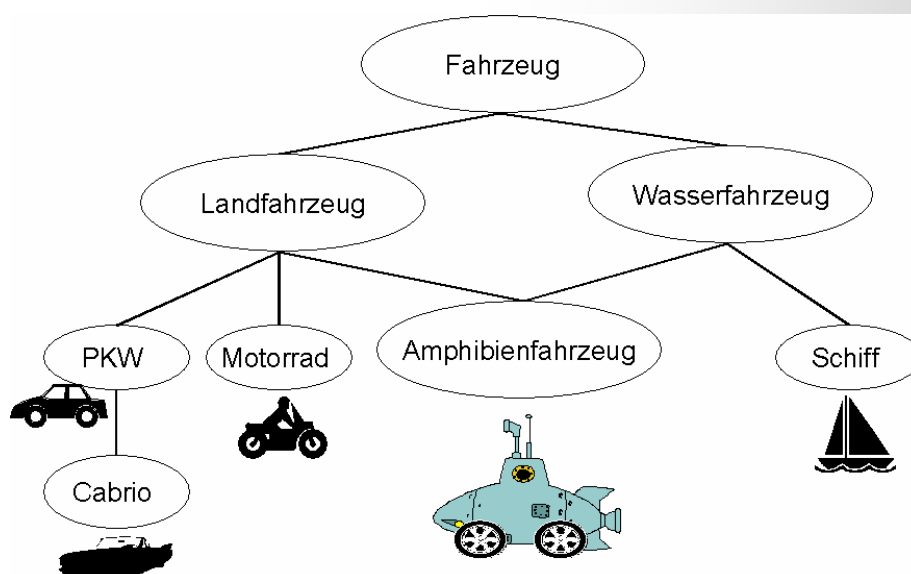
Assoziation und Aggregation

- Assoziation
 - Stellt Beziehung Objekten dar
- Aggregation
 - Stellt Beziehung zwischen einem Ganzen und seinen Teilen dar
 - „... hat ein ...“ oder „...ist Teil von...“ Beziehung



Vererbung

- Stellt eine Verallgemeinerung / Spezialisierung-Hierarchie dar
- Eine untergeordnete Klasse erbt Eigenschaften und Verhalten einer oder mehrerer übergeordneter Klassen
- „... ist ein ...“ Beziehung



Objektorientierte Modelle

Unterteilung der objektorientierten Konzepte in 3 Modelle:

- Basismodell
 - Datenkapselung
 - Abstraktion
 - Konzepte: Klasse, Objekt, Attribut, Methode
- Statisches Modell
 - Beziehung zwischen Modellelementen
 - Beschreibung von strukturellen Zusammenhängen
 - Konzepte
 - Assoziation
 - Vererbung
 - Paket
- Dynamisches Modell
 - Verhalten der einzelnen Modellelemente
 - Aspekt der Systemnutzung
 - Konzepte
 - Anwendungsfall
 - Zustandsdiagramm



Kapitel 4 Dynamische Konzepte in der objektorientierten Analyse

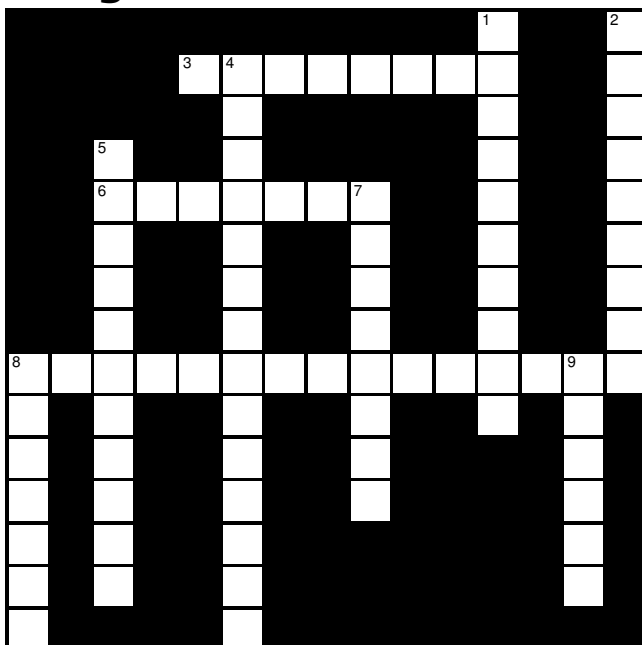
- 4.1 Anwendungsfall
- 4.2 Botschaft
- 4.3 Szenario
- 4.4 Zustandsautomat
- 4.5 Zusammenfassung der objektorientierten Konzepte (zum Selbststudium)
- 4.6 Zusammenfassung**

Zusammenfassung Kapitel 4

- Dynamische Konzepte beschreiben das Verhalten des Systems
- Verschiedene Sichten zur Spezifikation der Änderungen des Systems in Abhängigkeit der Zeit
- Anwendungsfälle: Modellierung der Funktionalität aus Benutzersicht
 - Dokumentation mit Anwendungsfalldiagrammen
- Szenarios: Darstellung des Nachrichtenaustauschs zwischen verschiedenen Objekten
 - Objektkommunikation über Botschaften
 - Sequenzdiagramm: zeitlicher Aspekt
 - Kollaborationsdiagramm: Verbindungen zwischen Objekten
- Zustandsdiagramm: Beschreibung des Verhaltens von Klassen oder Operationen
- Aktivitätsdiagramm: Darstellung paralleler Abläufe



Frage: Kreuzworträtsel zu Kapitel 4



Senkrecht

1. Zustandsübergang, häufig ausgelöst durch ein Ereignis
2. Mechanismus, mit dem Objekte untereinander kommunizieren können
4. Beschreibt einen zusammenhängenden Arbeitsablauf aus der Sicht seiner Akteure
5. Linie in einem Sequenzdiagramm, die den Lebenszeitraum eines Objektes darstellt
7. Sequenz von Bearbeitungsschritten, die unter bestimmten Bedingungen auszuführen sind
8. Situation im Leben eines Objektes, während der eine bestimmte Bedingung erfüllt ist
9. Rolle, die ein Benutzer des Systems spielt

Waagrecht

3. Bedingung, die für einen Zustandsübergang erfüllt sein muss
6. Eine Beziehung zwischen Anwendungsfällen, die besagt, dass ein Anwendungsfall unter bestimmten Umständen durch einen anderen erweitert wird
8. Besteht aus Zuständen und Transitionen



Kapitel 5 Analyseprozess und Analysemuster

<u>5.1 Analyseprozess</u>	<u>284</u>
<u>5.2 CRC-Karten</u>	<u>295</u>
<u>5.3 Analysemuster</u>	<u>308</u>
<u>5.4 Checklisten</u>	<u>328</u>
<u>5.5 Beispiel „Waschtrockner“</u>	<u>341</u>
<u>5.6 Zusammenfassung</u>	<u>352</u>

Kapitel 5 Analyseprozess und Analysemuster

- 5.1 Analyseprozess
- 5.2 CRC-Karten
- 5.3 Analysemuster
- 5.4 Checklisten (zum Selbststudium)
- 5.5 Beispiel „Waschtrockner“
- 5.6 Zusammenfassung

Lernziele

- Erklären können, wie der Analyseprozess ablaufen soll
- Verstehen, wie man zu einem guten Analysemodell kommt
- CRC-Karten in der Analyse einsetzen können
- Erklären können, was ein Muster ist
- Wichtige Muster der Systemanalyse kennen
- Analysemuster in einer Textbeschreibung erkennen und darstellen können
- Analysemuster in einem Klassendiagramm erkennen können
- Anwendungsfälle systematisch identifizieren und dokumentieren können

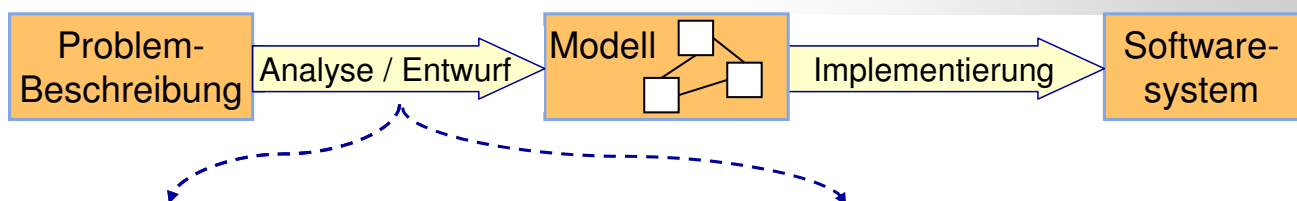


Kapitel 5 Analyseprozess und Analysemuster

5.1 Analyseprozess

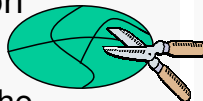
- 5.2 CRC-Karten
- 5.3 Analysemuster
- 5.4 Checklisten (zum Selbststudium)
- 5.5 Beispiel „Waschtrockner“
- 5.6 Zusammenfassung

Methodische Vorgehensweise bei der Analyse



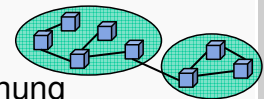
① Zerlegung und Abstraktion

- Zerlegung in kleinere, handhabbare Teilbereiche
- Unterdrückung nicht relevanter Eigenschaften



② Strukturierung

- Geeignete Anordnung der Systemelemente
- Beschreibung der Beziehung zwischen Systemelementen



- **Vorgehensweise** ist entscheidend für erfolgreiche Lösung
- Richtige Vorgehensweise = Gratwanderung zwischen Formalismus und Formlosigkeit
 - Sehr formelle Vorgehensweisen behindern Kreativität
 - Formlose Vorgehensweisen sind chaotisch und nicht sinnvoll

Nur durch richtige Vorgehensweise wird ein gutes Produkt erreicht!

Mögliche Vorgehensweisen bei der Modellerstellung

- Betonung des statischen Modells
 - Entwicklung eines semantischen Datenmodells in objektorientierter Notation
 - Dynamik des Systems wird außer Acht gelassen

Gefahr: Reines Datenmodell, Funktionalität unberücksichtigt

- Betonung des dynamischen Modells
 - use case driven approach
 - scenario driven approach

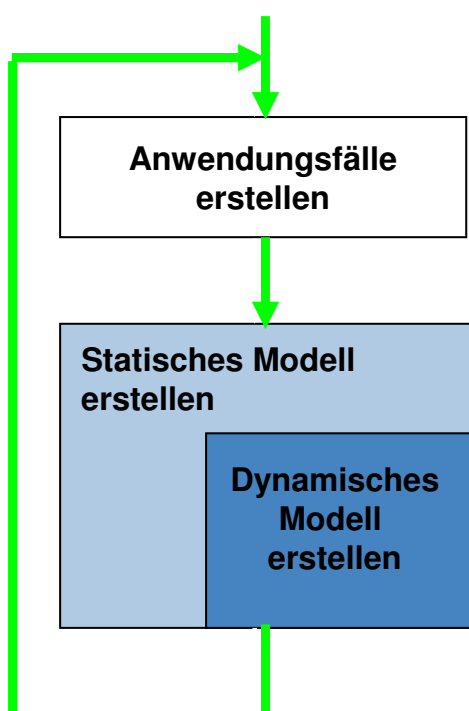
Gefahr: Funktionale Struktur lässt sich nicht direkt auf objektorientierte Architektur abbilden

- Zur Überprüfung des statischen Modells wird das dynamische Modell benötigt und umgekehrt

⇒ **Eine erfolgreiche Modellierung bedingt das Zusammenwirken von statischem und dynamischem Modell**

Makroprozess (1)

Festlegung von methodischen Schritten



- Beschreibung der Reihenfolge der einzelnen Aufgaben zur Erstellung des OOA-Modells auf hohem Abstraktionsniveau
- Berücksichtigt das **Gleichgewicht** (*balancing*) von statischem und dynamischem Modell
- Konzentration auf das statische Modell vor dem dynamischen Modell
 - Größere Stabilität des Modells
 - Schaffung einer wesentlichen Abstraktionsebene durch die Bildung von Klassen
- Wichtig: Paralleles Entwickeln beider Modelle und Berücksichtigung von Wechselwirkungen

Makroprozess (2)

- Ermitteln der relevanten Anwendungsfälle
- Identifikation der Klassen
- Erstellung des statischen Modells
- Parallele Erstellung des dynamischen Modells
- Berücksichtigung der Wechselwirkung beider Modelle

Ablauf:

- | | |
|-------------------------------|---------------------------|
| 1 Anwendungsfälle formulieren | Analyse im Großen |
| 2 Teilsysteme bilden | |
| 3 Klassendiagramme erstellen | Statisches Modell |
| 4 Zustandsdiagramme erstellen | Dynamisches Modell |

Ablauf des Makroprozesses (1)

1. Anwendungsfälle formulieren

- Beschreibung Anwendungsfälle, Anwendungsfalldiagramm erstellen

2. Teilsysteme bilden

bei großen Systemen notwendig

- Gruppierung von Modellelementen zu Teilsystemen ⇔ Paketdiagramm

3. Klassendiagramme erstellen

Statisches Modell

- Klassen identifizieren, Kurzbeschreibung der Klassen
- Assoziationen, Attribute und ggf. Vererbungsstrukturen identifizieren
- Assoziationen vervollständigen
 - Festlegung: einfache Assoziation, Aggregation oder Komposition
 - Kardinalitäten, Rollen, Namen, Restriktionen
- Attribute spezifizieren
 - Vollständige Spezifikation aller identifizierten Attribute

Ergebnis: Klassendiagramm, Objektdiagramm

Ablauf des Makroprozesses (2)

4. Szenarios erstellen

Dynamisches Modell

- Anwendungsfälle durch eine Menge von Szenarios beschreiben
⇒ **Sequenzdiagramm**, ⇒ **Kollaborationsdiagramm**

5. Zustandsautomaten erstellen

- Jede Klasse ist daraufhin zu überprüfen, ob ein nicht-trivialer Lebenszyklus erstellt werden kann
⇒ **Zustandsdiagramm**
- Operationen beschreiben
 - Je nach Komplexitätsgrad der Operation entsprechende Form wählen⇒ **Klassendiagramm**, **Zustandsautomat**, **Aktivitätsdiagramm**

Alternative Makroprozesse

Szenario-basierter Makroprozess

- Bei umfangreichen funktionalen Anforderungen
- Keine alten Datenbestände

Vorgehen:

1. Anwendungsfälle formulieren
2. Szenarios aus den Anwendungsfällen ableiten
3. Interaktionsdiagramme aus den Szenarios ableiten
4. Klassendiagramme erstellen
5. Zustandsdiagramme erstellen

Daten-basierter Makroprozess

- Bei umfangreichem Datenmodell
- Alte Datenbestände existieren
- Umfang der funktionalen Anforderungen ist zunächst unbekannt
- Bei Auskunftssystemen mit flexibel gestalteten Anfragen

Vorgehen:

1. Klassendiagramme erstellen
2. Anwendungsfälle formulieren
3. Szenarios aus Anwendungsfällen ableiten
4. Interaktionsdiagramme aus Szenarios und Klassendiagrammen ableiten
5. Zustandsdiagramme erstellen

Anmerkungen zum Analyseprozess

- Wichtig: Eine schnelle Entwicklung der ersten Version des Modells
 - Zügiger Projektfortschritt
 - Unterstützung der Kommunikation im Team
- Erste Modelle sind wahrscheinlich weder besonders gut, noch in jedem Fall korrekt
- Gute Ideen sind nicht plötzlich da, sie entwickeln sich.
- *DeMarco*
If you wait for a complete and perfect concept to germinate in your mind, you are likely to wait forever.



Häufige Fehler beim Analyseprozess

- Das 100%-Syndrom
- Zu frühe Qualitätsoptimierung
 - Konzentration zunächst auf das fachliche Konzept
 - Danach Optimierung des fachlich korrekten Modells unter Gesichtspunkten eines optimalen OOA-Modells
- Bürokratische Auslegung der Methode
 - *follow the spirit, not the letter of a method*
- Entwurfskriterien in der Analyse berücksichtigen

Hilfsmittel im Analyseprozess

- CRC-Karten **(Erkennen von Strukturen)**
- Analysemuster
(Standardisierung von Problemen, Aufwandreduzierung)
- Checklisten
(Erfahrungswissen, Regeln für die Analyse)



Frage zu 5.1: Analyseprozess

Sie erhalten die Aufgabe, ein 20 Jahre altes Informationssystem neu zu entwickeln, d.h. ein Re-Engineering-Projekt durchzuführen. Für Ihre Arbeit erhalten Sie das ablauffähige System, die Benutzerhandbücher und die Dateibeschreibungen. Außerdem stehen die Benutzer des alten Systems für Interviews zur Verfügung. Wie gehen Sie vor?

Antwort:

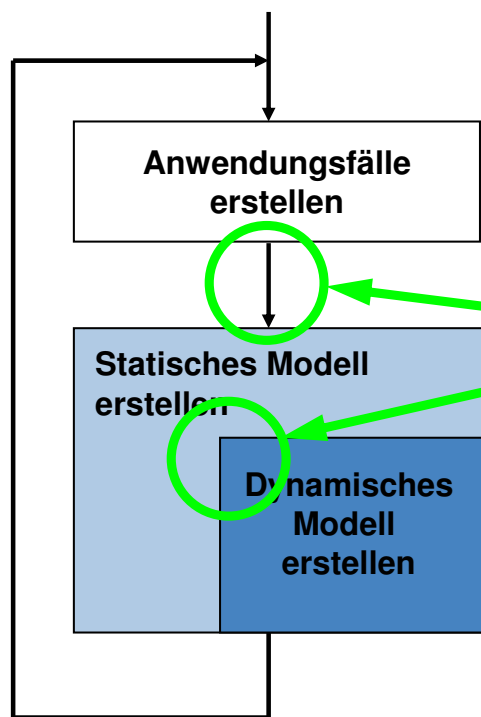
- Vorgehensweise analog
 - Datenbasierter Makroprozess
- Anwendungsfälle werden abgeleitet aus:
 - Interviews mit den Benutzern
 - Dem vorhandenen ablauffähigem System
 - Dem Benutzerhandbuch
- Grundlage für die Erstellung des Klassendiagramms:
 - Existierende Dateibeschreibungen



Kapitel 5 Analyseprozess und Analysemuster

- 5.1 Analyseprozess
- 5.2 CRC-Karten**
- 5.3 Analysemuster
- 5.4 Checklisten (zum Selbststudium)
- 5.5 Beispiel „Waschtrockner“
- 5.6 Zusammenfassung

CRC-Karten im Analyseprozess



Welche Klassen
brauche ich
eigentlich?

CRC-Karten

- Hilfsmittel für den Analyseprozess
- Aufteilung der ermittelten Aufgaben des Systems auf einzelne Klassen
- Verbindung zwischen Anwendungsfällen, statischem und dynamischem Modell
- Ergänzung zum OOA-Modell

Was ist eine CRC-Karte?

Definition:

CRC-Karten sind **Karteikarten** mit dem Namen einer Klasse, ihren Verantwortlichkeiten und ihren Beziehungen zu anderen Klassen

- CRC = Class Responsibility & Collaborations
- Ziele
 - Identifikation von Klassen und Assoziationen zwischen Klassen
 - Identifikation der Verantwortlichkeiten von Klassen
 - Identifikation der Richtung von Assoziationen
- Darstellung von Informationen auf höherer Abstraktionsebene als im Klassendiagramm **Basistechnik bei Analyse und Entwurf**
- Anm.: CRC-Karten gehören nicht zur UML, sind aber ein weit verbreitetes Hilfsmittel

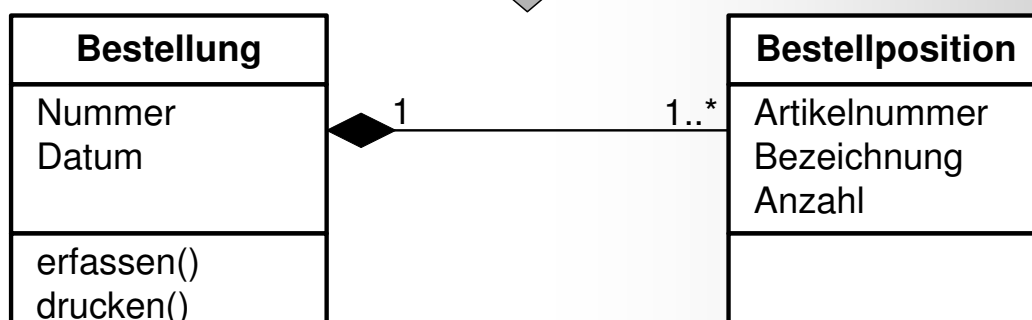
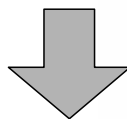
Inhalt einer CRC-Karte

- Eine CRC-Karte besteht aus:
 - Name der Klasse (**class**)
 - Verantwortlichkeiten (**responsibilities**) der Klasse
 - = Menge aller erforderlichen Dienste und dazu notwendige Attribute
 - ⇒ aus den Verantwortlichkeiten ergeben sich häufig Operationen
 - Zusammenarbeit mit anderen Klassen (**collaborations**)
 - = Zur Erfüllung der Aufgaben benötigte andere Klassen

Name der Klasse	
Verantwortlichkeiten <ul style="list-style-type: none"> Aufgaben der Klasse 	Kollaborationen <ul style="list-style-type: none"> Beziehungen zu anderen Klassen

Beispiel CRC-Karte und Klassendiagramm:

Klasse Bestellung	
Verantwortlichkeiten <ul style="list-style-type: none"> verwaltet eine Bestellung delegiert Aufgaben an Bestellpositionen 	Kollaborationen <ul style="list-style-type: none"> Bestellposition



Analyse mit CRC-Karten (1)

- Ausgangspunkt
 - Anwendungsfälle und Sequenzdiagramme
- Vorgehensweise
 - Anwendungsfälle durchgehen und erarbeiten, wie das Klassenmodell die von den Anwendungsfällen geforderte Funktionalität bereitstellt
 1. Klassen identifizieren
 2. Anlegen einer CRC-Karte für jede Klasse
 3. Verantwortlichkeiten und Beziehungen aus dem Text identifizieren und auf CRC-Karte ergänzen
 4. CRC-Karten auf Basis der Beziehungen ordnen (auf den Tisch legen) und in Klassendiagramm übertragen

Erkennen fehlender Verantwortlichkeiten

- Möglich: Rollenspiel in einem Team
 - ⇒ Personen übernehmen die Verantwortlichkeiten
 - ⇒ Szenario walk-through mit CRC-Karten

Analyse mit CRC-Karten (2)

1. Klassen identifizieren

- Substantive herausfiltern
- Wesentliches von Unwesentlichem trennen,
- Synonyme erkennen
- Von konkreten Begriffen des Anwendungsgebiets ausgehen
- Oberklassen (falls überhaupt) nur ausgehend von den Verantwortlichkeiten bilden

2. Anlegen einer CRC-Karte für jede Klasse

Analyse mit CRC-Karten (3)

3a. Verantwortlichkeiten

- charakterisieren die von der Klasse erbrachten Dienste
 - sollen zusammenhängendes Wissen enthalten
 - **Was** wird geleistet; niemals: wie es durchgeführt wird
- ⇒ auf Wesentliches konzentrieren
- ⇒ konstruktiv denken (lässt sich eine solche Klasse bauen?)

– Erkennen von Verantwortlichkeiten

- Welche Informationen kennt eine Klasse?
- Was soll getan werden, welche Dienste können angeboten werden?
- Warum wurde die Klasse entworfen? Ist sie vollständig, kann man sie durch weitere Verantwortlichkeiten abrunden?
- Welche Rollen findet man vor, mit welchen Funktionen sind sie assoziiert?

Analyse mit CRC-Karten (4)

3b. Kollaborationen

- benennen Anbieter von Diensten
- Zusammenarbeit wird benötigt, um Dienste zu erbringen
- also: andere Klassen und eventuell die an Objekte dieser Klassen delegierten Verantwortlichkeiten

– Erkennen von Zusammenarbeit

- Wer liefert Wissen, das eine Klasse benötigt?
- Verantwortlichkeiten betrachten. Bei den Lieferanten von Diensten ggf. die Verantwortlichkeiten ergänzen.

Analyse mit CRC-Karten (5)

- Probleme bei der Aufteilung der Verantwortlichkeiten unter Klassen
 - Zu viele Verantwortlichkeiten pro Klasse
 - ⇒ **geringe Bindung (Kohäsion) der Klassen im Modell**
 - Zu wenig Verantwortlichkeiten pro Klasse
 - ⇒ **hohe Kopplung der Klassen im Modell**
- Klassenmodell zu komplex und schwer änderbar**
- Klassen ohne Verantwortlichkeiten
- Unzusammenhängende Verantwortlichkeiten in einer Klasse
- Die gleiche Verantwortlichkeit in mehreren Klassen
- Klassen mit unbenutzter Verantwortlichkeit

⇒ **Frühes Erkennen von Fehlern im Modell**

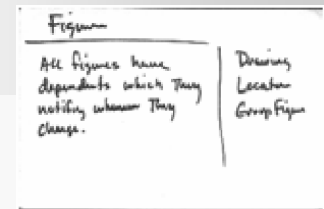
Gewinn durch CRC-Karten

- Griffige Beschreibungstechnik, keine lange Einarbeitungszeit
- Billig, flexibel, nicht gewöhnungsbedürftig, überall einsetzbar
- Einfach zu erweitern und zu ändern
- Erleichtert die Bildung eines eingespielten Teams
- Erlaubt Diskussion mit Anwendern über Anwendungsbegriffe

Klasse CRC-Karte	
Verantwortlichkeiten <ul style="list-style-type: none"> • Klassen & Assoziation identifizieren • Verantwortlichkeiten identifizieren • Richtung von Assoziationen identifizieren 	Kollaborationen <ul style="list-style-type: none"> • Klassendiagramm • Szenarios

Frage zu 5.2 (1)

- Welche Aussagen kann eine Klasse direkt aus ihrer CRC-Karte ableiten?



- ☒ „Das ist gar nicht meine Aufgabe!“
- ☒ „Mir fehlen Informationen!“
- ☒ „Ich kenne den Verantwortlichen nicht!“
- f** ☐ „Diese Aufgabe muss ich erben!“
- ☒ „Ich werde zu kompliziert!“
- f** ☐ „Ich benötige ein Zustandsdiagramm!“
- ☒ „Ich werde nicht gebraucht!“

Frage zu 5.2 (2)

- Erstellen Sie eine CRC-Karte für die Klasse ‘Teambesprechung’ auf Basis der folgenden Kurzbeschreibung:
 „Ein Objekt von ‘Teambesprechung’ beschreibt genau einen Termin, an dem mehrere Teilnehmer teilnehmen sollen und für den ein Besprechungsraum reserviert werden muss.“

Klasse Teambesprechung	
Verantwortlichkeiten <ul style="list-style-type: none"> – Titel wissen – Datum wissen – Teilnehmer kennen – Teilnehmer einladen – Raum festlegen 	Kollaborationen <ul style="list-style-type: none"> – Teilnehmer – Besprechungsraum

Kapitel 5 Analyseprozess und Analysemuster

- 5.1 Analyseprozess
- 5.2 CRC-Karten
- 5.3 Analysemuster**
- 5.4 Checklisten (zum Selbststudium)
- 5.5 Beispiel „Waschtrockner“
- 5.6 Zusammenfassung

Definitionen

Ein **Muster** (*pattern*) ist eine Idee, die sich in einem praktischen Kontext als nützlich erwiesen hat und es wahrscheinlich auch in anderen sein wird [Fowler]

Ein **Analysemuster** ist eine Gruppe von Klassen mit feststehenden Verantwortlichkeiten und Interaktionen [Coad]

- Generalisierte Lösungsideen zu immer wiederkehrenden Analyseproblemen
- Keine fertig codierten Lösungen, sondern Beschreibung eines Lösungsansatzes

Ziele

- Bewährte und erprobte Analysemuster verbessern die Qualität des OOA-Modells
- Verkürzung der Entwicklungszeit des OOA-Modells
- Effektive Kommunikation

Beschreibung von Mustern

- Jedes Muster wird über einen eindeutigen Namen identifiziert
- Beschreibung der Motivation des Musters
- Festlegen der Eigenschaften des Musters
- Erläuterung durch ein oder mehrere Beispiele

Bekannte Muster:

- Liste
- Exemplartyp
- Verbund
- Koordinator
- Rollen
- Wechselnde Rollen
- Historie
- Gruppe
- Gruppenhistorie

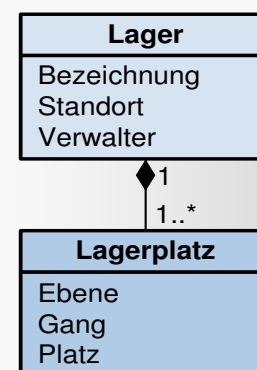
Muster 1: Liste (1)

Liste

Motivation: Lager mit Lagerplätzen

Lager				
Bezeichnung	Süd			
Standort	Dortmund			
Verwalter	HansMüller			
Ebene	Gang	Platz	Belegt mit	
1	1	1	XYZ	
1	1	2	ABC	
1	2	1		

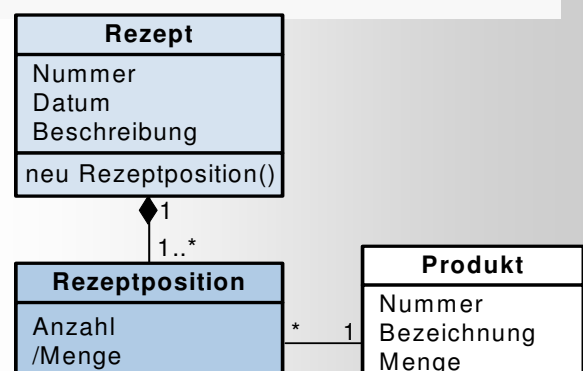
Analyse



Motivation: Rezept mit Rezeptpositionen

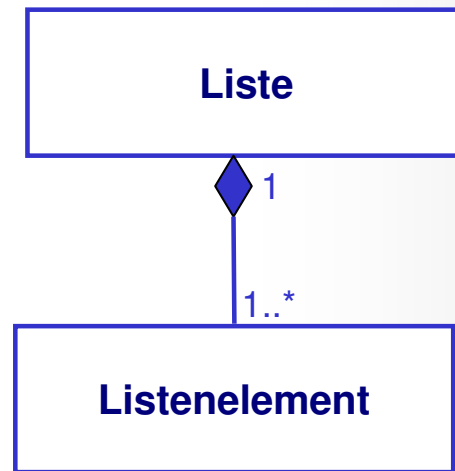
Braurezept				
Nr.	4711	01.01.2004		
Rezept für dunkles Vollbier				
Nr.	Bezeichnung	Einzelmenge	Anzahl	Menge
47	Malz	2,00	12	24,00
11	Hopfen	0,25	2	00,50
Summe				24,50

Analyse



Muster 1: Liste (2) - allgemeine Darstellung

Liste

(Tafelanschrieb)**Muster 1: Liste (3)**

Liste

Eigenschaften

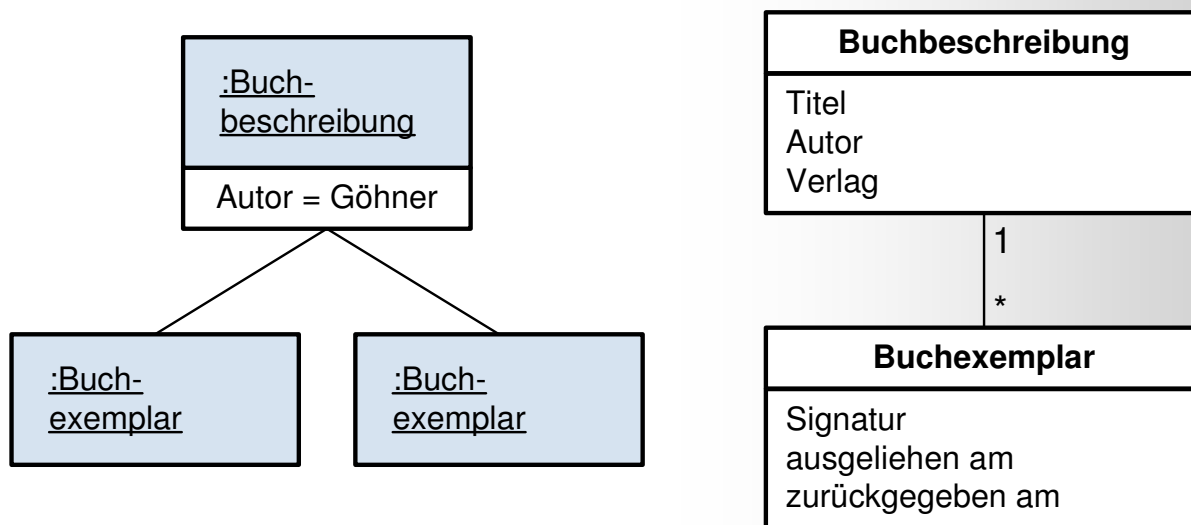
- Komposition
- Ein Ganzes besteht aus **gleichartigen** Teilen **nur eine Teilklasse**
- Teil-Objekte bleiben einem Aggregat-Objekt fest zugeordnet, können jedoch gelöscht werden, bevor das Ganze gelöscht wird
- Attributwerte des Aggregat-Objekts gelten auch für die zugehörigen Teil-Objekte
- Das Aggregat-Objekt enthält mindestens ein Teil-Objekt

Kardinalität ist meist 1..*

Muster 2: Exemplartyp (1)

Exemplartyp

Motivation: Verwaltung von mehreren Exemplaren eines bestimmten Objektes (z.B. Buch mit mehreren Exemplaren)



Muster 2: Exemplartyp (2)

Exemplartyp

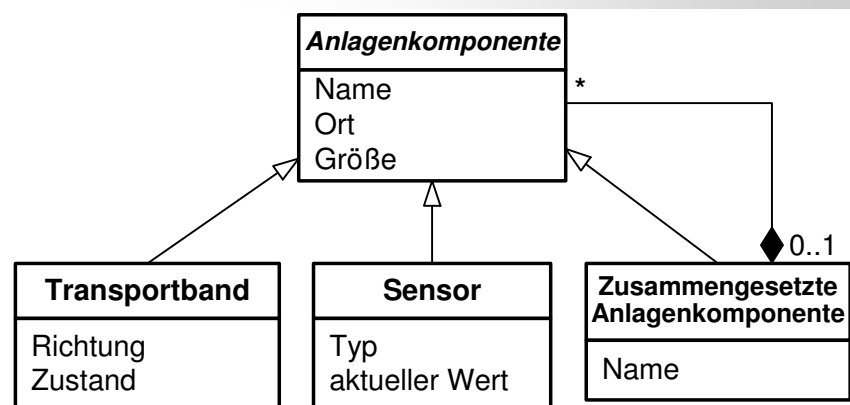
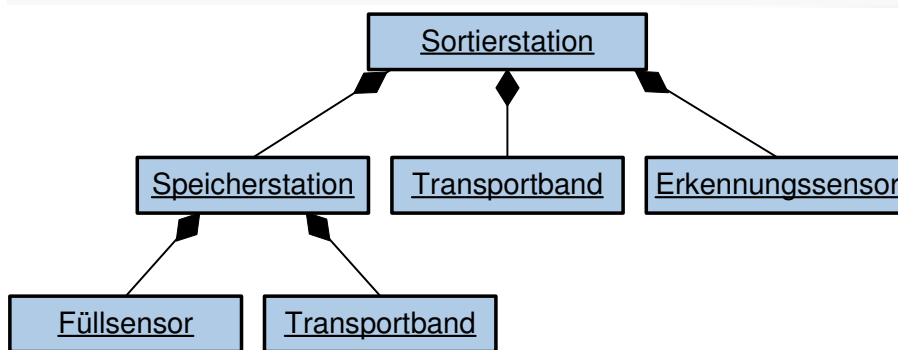
Eigenschaften

- Einfache Assoziation **keine „whole-part“ Beziehung**
- Name der neuen Klasse enthält oft Begriffe wie Typ, Gruppe, Beschreibung, Spezifikation
- Erstellte Objektverbindungen werden nicht verändert. Sie werden nur gelöscht, wenn das betreffende Exemplar gelöscht wird.
- Eine Beschreibung kann – zeitweise – unabhängig von Exemplaren existieren **Kardinalität ist meist ***
- Bei Verzicht auf die neue Klasse, würde als Nachteil lediglich die redundante »Speicherung« von Attributwerten auftreten

Muster 3: Verbund (composite) (1)

Verbund

Motivation: Anlagenstruktur mit **unterschiedlichen** Objekten



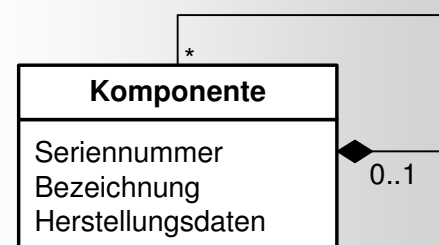
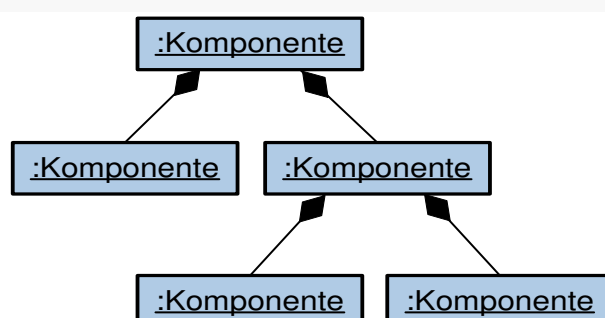
Muster 3: Verbund (2)

Verbund

Eigenschaften

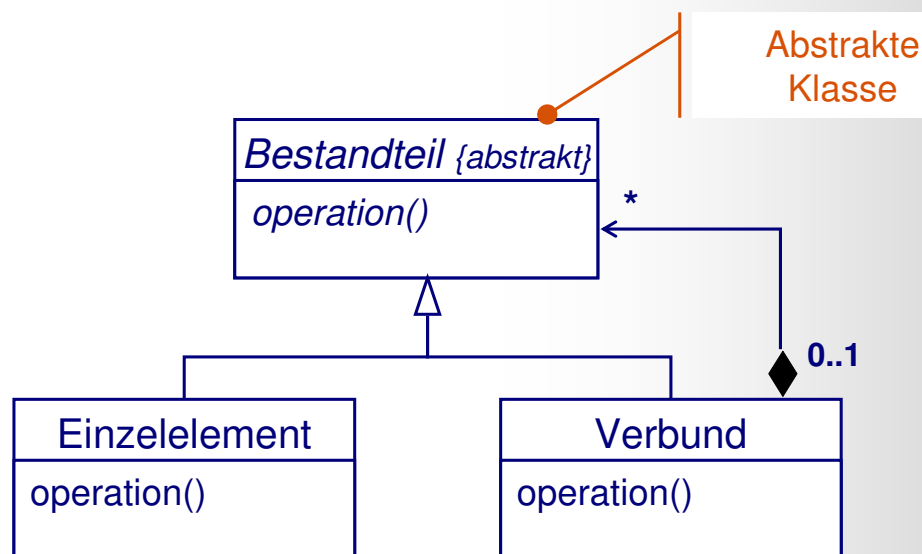
- Komposition
- Aggregat-Objekt und seine Teilobjekte können als Einheit oder auch einzeln behandelt werden **Löschen von Verzeichnissen oder Dateien**
- Teilobjekte können anderem Aggregat-Objekt zugeordnet werden
- Kardinalität der Aggregat-Klasse ist 0..1
- Objekt der Art A kann sich aus mehreren Objekten der Arten A, B und C zusammensetzen

Sonderfall: Komponentenstruktur mit **gleichen** Objekten



Muster 3: Verbund (3) - allgemeine Darstellung (Tafelanschrieb)

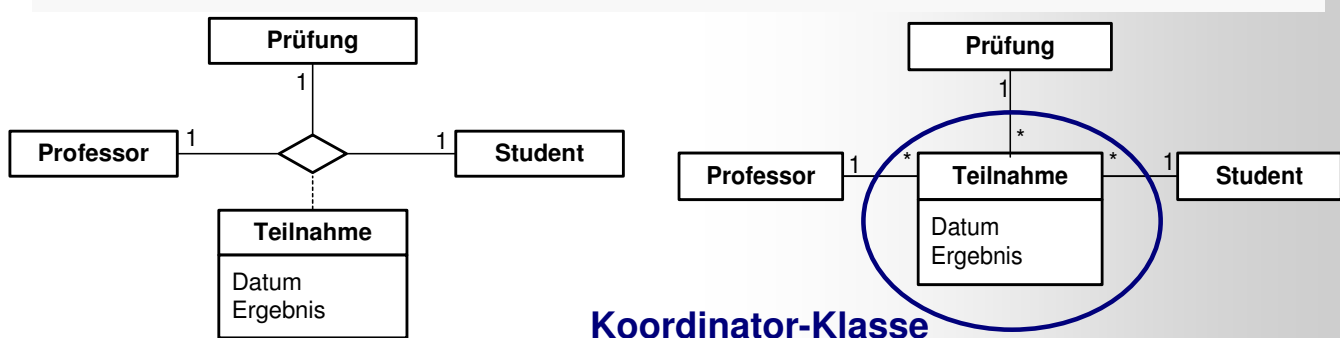
Verbund



Muster 4: Koordinator (1)

Koordinator

Motivation: Beziehung zwischen Professoren, Prüfungen und Studenten



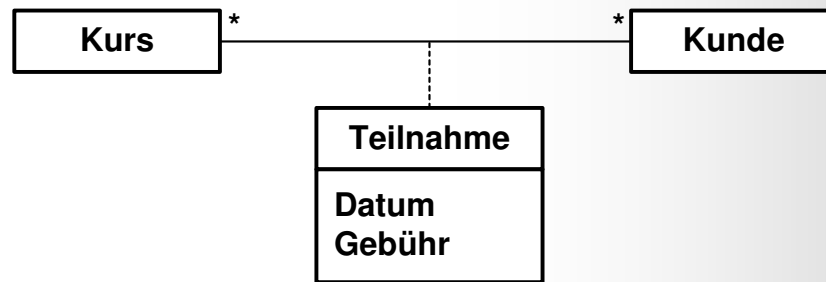
Eigenschaften

- N-äre Assoziationen
- Koordinator-Klasse ersetzt n-äre Assoziation mit assoziativer Klasse ($n \geq 2$)
- Koordinator-Klasse besitzt kaum Attribute/Operationen, sondern mehrere Assoziationen zu anderen Klassen (im Allgemeinen zu genau einem Objekt jeder Klasse)

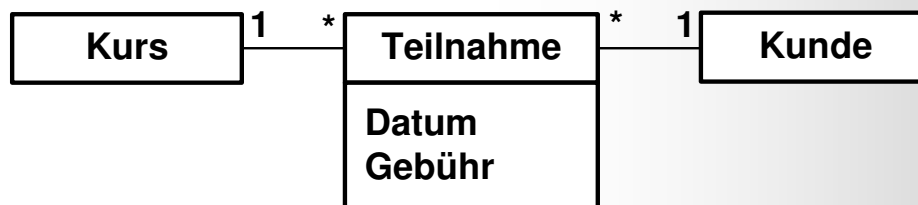
Muster 4: Koordinator (2)

Koordinator

Sonderfall: Assoziative Klasse in binärer Assoziation



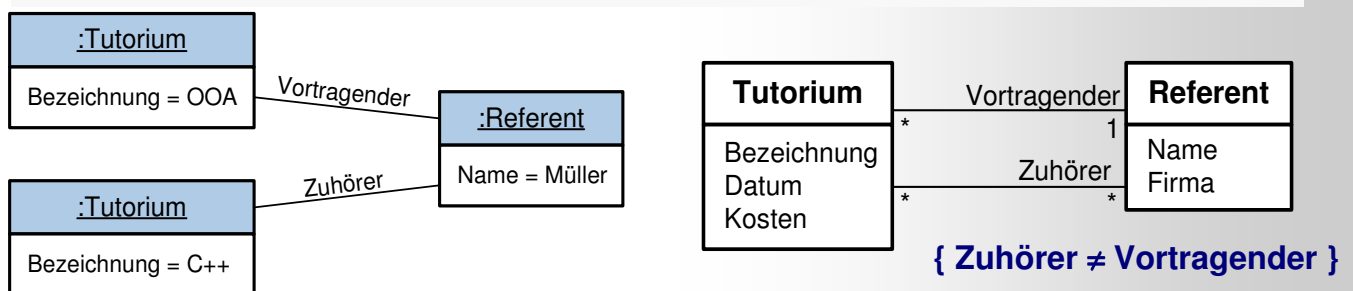
Frage: Wie könnte hier die Koordinator-Klasse lauten?



Muster 5: Rolle

Rollen

Motivation: Tutorium (Referent kann Zuhörer oder Vortragender sein)



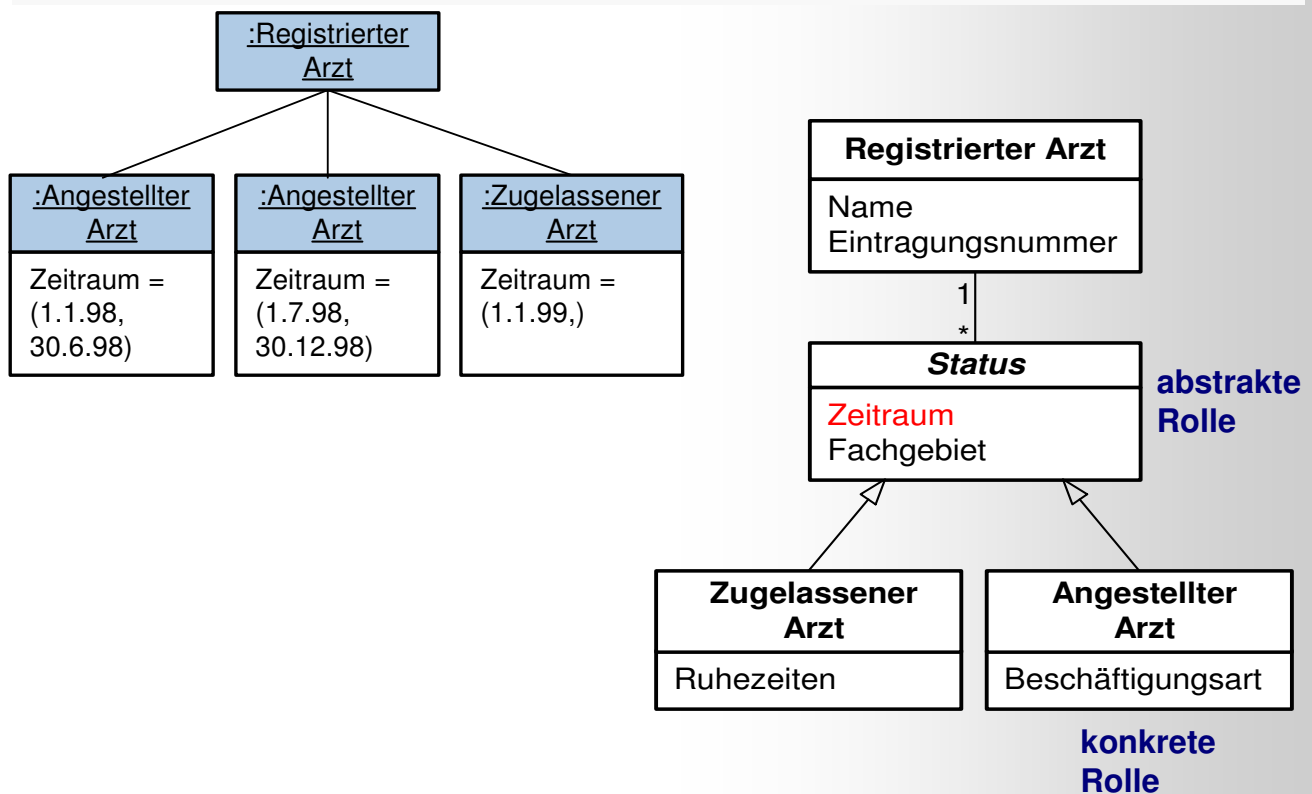
Eigenschaften

- Ein Objekt kann – **zu einem Zeitpunkt** – in Bezug auf Objekte anderer Klassen verschiedene Rollen einnehmen
- Zwischen zwei Klassen existieren zwei oder mehrere Assoziationen
- Objekte, die verschiedene Rollen spielen können, besitzen unabhängig von der jeweiligen Rolle
 - gleiche Eigenschaften (Attribute, Assoziationen)
 - gleiche Funktionalität (Operationen)

Muster 6: Wechselnde Rollen (1)

Wechselnde
Rollen

Motivation: Arten von alternierenden Aufgaben von Ärzten



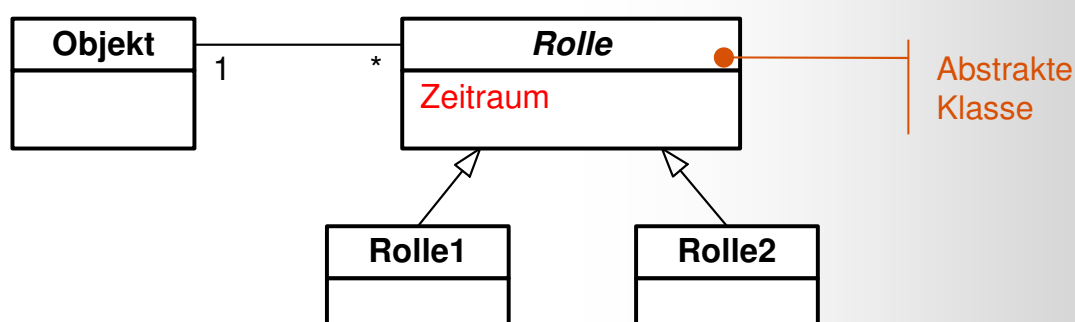
Muster 6: Wechselnde Rollen (2)

Wechselnde
Rollen

Eigenschaften:

- Objekt der realen Welt kann – zu verschiedenen Zeiten – verschiedene Rollen spielen
- In jeder Rolle kann es unterschiedliche Eigenschaften und Operationen besitzen
- Die konkreten Rollen werden mittels Vererbung modelliert
- Objektverbindungen zwischen Objekt und seinen Rollen werden nur erweitert, d.h. weder gelöscht, noch zu anderen Objekten aufgebaut

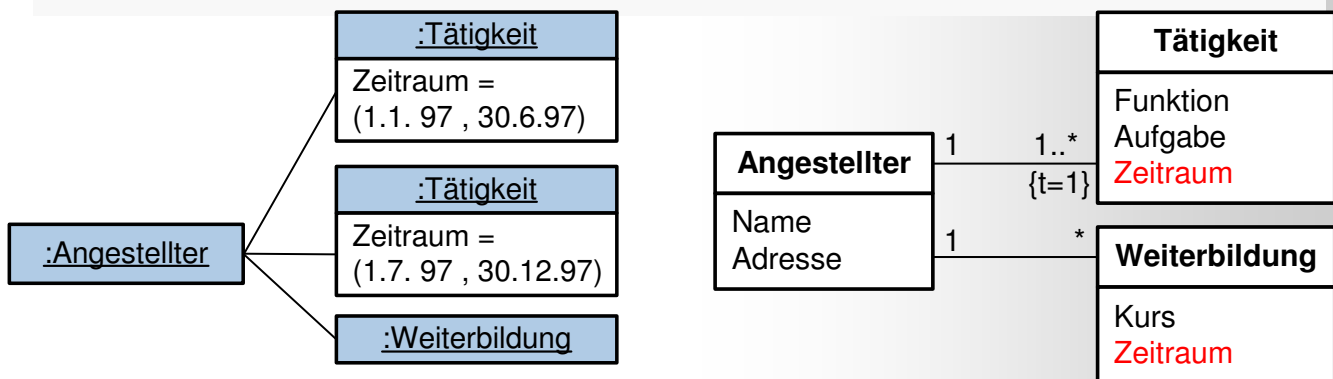
Attribute
Assoziationen



Muster 7: Historie

Historie

Motivation: Tätigkeiten und Weiterbildung eines Angestellten



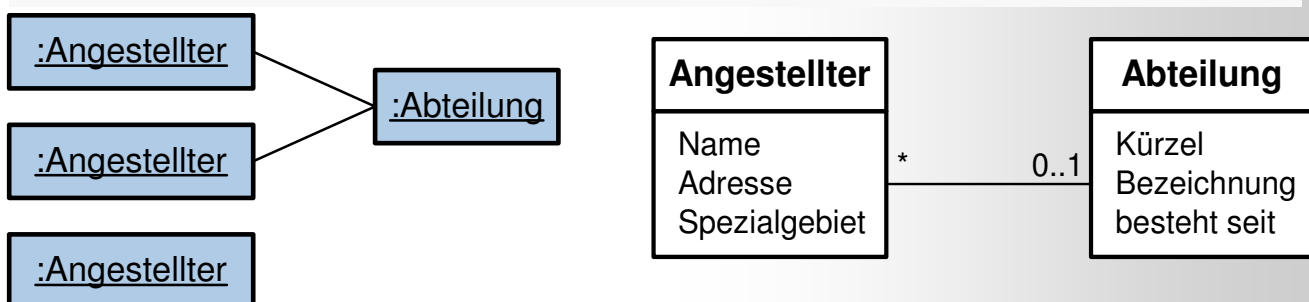
Eigenschaften

- Für ein Objekt sind alle Bewegungen/Vorgänge/Fakten zu dokumentieren, d.h. alle aufgebauten Objektverbindungen bleiben bestehen
- Für jeden Vorgang bzw. jedes Faktum ist der **Zeitraum** festzuhalten
- Objektverbindungen zu Fakten bzw. Vorgängen werden nur erweitert
- Die zeitliche Restriktion $\{t = k\}$ (k = gültige Kardinalität) sagt aus, was zu einem Zeitpunkt gelten muss

Muster 8: Gruppe

Gruppe

Motivation: Angestellte in einer Abteilung zu einem Zeitpunkt



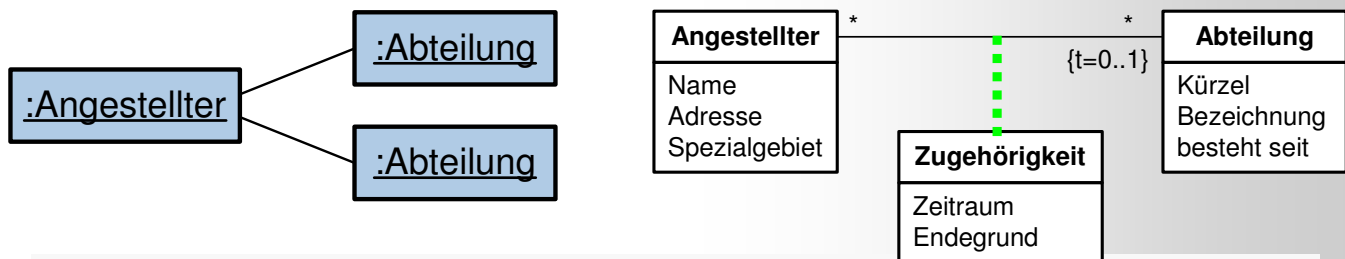
Eigenschaften

- Einfache Assoziation
- Mehrere Einzel-Objekte gehören – zu einem Zeitpunkt – zum **selben** Gruppen-Objekt
- Gruppen
 - können entweder zeitweilig ohne Einzel-Objekte existieren oder
 - müssen immer eine Mindestanzahl von Einzel-Objekten besitzen
- Objektverbindungen können auf- und abgebaut werden

Muster 9: Gruppenhistorie

Gruppen-
historie

Motivation: Angestellte in einer Abteilung über einen Zeitraum



Eigenschaften

- Ein Einzel-Objekt gehört – im Laufe der Zeit – zu **unterschiedlichen** Gruppen-Objekten.
- Historie (Zeitraum) wird mittels einer assoziativen Klasse modelliert. Dadurch ist Zuordnung der Einzel-Objekte zur Gruppe sichtbar.
- Die zeitliche Restriktion $\{t = k\}$ (k = gültige Kardinalität) sagt aus, was zu einem Zeitpunkt gelten muss.
- Objektverbindungen bleiben bestehen, neue Verbindungen werden hinzugefügt.



Fragen zu 5.3 : Analysemuster

- In einem Grafiksystem bilden Kreise und Rechtecke eine Gruppe. Diese Gruppe kann wiederum Teil einer anderen Gruppe sein.
Verbund
- Zu einem Inventarstück in einem Museum sollen der derzeitige Eigentümer, der Vorbesitzer, der Finder und/oder der Überbringer festgehalten werden, die jeweils die gleichen Eigenschaften besitzen. Eine Person kann beispielsweise sowohl Eigentümer als auch Finder sein.
Rollen
- Bei mehreren Videokassetten in einer Videothek handelt es sich um den gleichen Film.
Exemplartyp
- Für Personen sollen die Wohnsitze der letzten 10 Jahre ermittelt werden können. Zu einem Zeitpunkt muss jede Person mindestens einen und kann höchstens zwei Wohnsitze besitzen.
Historie



Kapitel 5 Analyseprozess und Analysemuster

- 5.1 Analyseprozess
- 5.2 CRC-Karten
- 5.3 Analysemuster
- 5.4 Checklisten (zum Selbststudium)**
- 5.5 Beispiel „Waschtrockner“
- 5.6 Zusammenfassung

Was ist ein gutes OOA-Modell?

Nach Fowler sind folgende Grundsätze für die Erstellung eines guten OOA-Modells zu beachten:

- Es gibt keine richtigen oder falschen Modelle. Es gibt nur Modelle, die mehr oder weniger gut ihren Zweck erfüllen.
- Ein gutes Modell ist immer verständlich, d.h. es sieht einfach aus.
- Die Erstellung verständlicher Modelle erfordert viel Aufwand.
- Das Wissen von kompetenten Fachexperten ist absolut notwendig für ein gutes Modell.
- Modellieren Sie kein System, das zu flexibel ist und zu viele Sonderfälle enthält. Diese Modelle sind aufgrund ihrer Komplexität immer schwer verständlich und damit schlechte Modelle.
- Prüfen Sie für jeden Sonderfall, ob er es wert ist, die Komplexität des Modells und des zu realisierenden Systems zu erhöhen.

Was sind Checklisten?

- Beinhalten nützliche Regeln und Richtlinien für die Analyse
- Dienen als Erfahrungspuffer für Entwickler
- Arten von Checklisten:

- Anwendungsfälle

- Teilsystembildung

- Zustandsautomaten

- Operationen

**Analyse im
Großen**

**Dynamisches
Modell**

- Klassen

- Assoziationen

- Attribute

- Vererbung

**Statisches
Modell**

Sinnvoller Aufbau der Checklisten

Konstruktive Schritte	Wie findet man ein Modellelement?
Analytische Schritte	Ist es ein „gutes“ Modellelement? Konsistenzprüfung, Fehlerquellen

Checklisten für die Erstellung von Anwendungsfällen

- Allgemeine Regeln
 - Konzentration auf die primären Anwendungsfälle, um ein Verständnis für den Kern des Systems zu erarbeiten
 - Zu einem Zeitpunkt immer nur an einem Anwendungsfall arbeiten
 - Bei umfangreichen Systemen müssen zuvor Teilsysteme gebildet werden
- Ein Anwendungsfall ...
 - beschreibt immer einen kompletten Ablauf von Anfang bis Ende
 - besteht daher meistens aus mehreren Schritten oder Transaktionen
 - kann ein Schritt eines anderen Anwendungsfalles sein
 - kann im Extremfall auf eine einzige Operation abgebildet werden

Vergleich Anwendungsfall und Funktion

Anwendungsfälle	Klassische funktionale Zerlegung
<ul style="list-style-type: none"> – Beschreibung der mit dem System auszuführenden Arbeitsabläufe – Dokumentation auf hoher Ebene – Reines Analysekonzept 	<ul style="list-style-type: none"> – Beschreibung der Funktionen des Systems – unabhängig vom jeweiligen Arbeitsablauf – Einsatz in Analyse und Entwurf – Verwendung auf allen Abstraktionsebenen

Regeln für die Erstellung von Anwendungsfällen

- Dokumentation der Anwendungsfälle so, dass sie sowohl für die Interviewten als auch für andere Analytiker verständlich sind
- Formulierung der Anwendungsfälle auf einer hohen Abstraktionsebene
- Außerachtlassung von Sonderfällen
- Möglichst aussagekräftige und präzise Benennung der Anwendungsfälle
 - Was wird gemacht? **Beispiel: Bearbeite Anmeldung**
 - Womit wird etwas gemacht? **Storniere Seminar**

Checkliste für Anwendungsfälle (1)

Konstruktive Schritte	<ol style="list-style-type: none"> 1. Akteure ermitteln 2. Anwendungsfälle für die Standardverarbeitung ermitteln 3. Anwendungsfälle für die Sonderfälle formulieren 4. Aufsplitten komplexer Anwendungsfälle 5. Gemeinsamkeiten von Anwendungsfällen ermitteln
Analytische Schritte	<ol style="list-style-type: none"> 6. Kriterien für einen guten Anwendungsfall 7. Konsistenz mit dem Klassendiagramm 8. Fehlerquellen

Checkliste für Anwendungsfälle (2)

1. Akteure ermitteln

- Welche Personen führen diese Aufgaben durch?
- Welche Schnittstellen besitzt das System?

2. Standardverarbeitung beschreiben

- Primäre und ggf. sekundäre Anwendungsfälle betrachten
- Konzentration auf Standardfälle, d.h. keine Sonderfälle betrachten

2a. mittels Akteuren

- Sind die Akteure Personen?
- Welche Arbeitsabläufe lösen sie aus?
- An welchen Arbeitsabläufen wirken sie mit?

2b. mittels Ereignissen

- Erstellen einer Ereignisliste
- Für jedes Ereignis einen Anwendungsfall identifizieren
- Externe und interne Ereignisse unterscheiden

Checkliste für Anwendungsfälle (3)

2. Standardverarbeitung beschreiben ...

2c. mittels Aufgabenbeschreibungen

- Was sind die Gesamtziele des Systems?
- Welches sind die zehn wichtigsten Aufgaben?
- Was ist das Ziel jeder Aufgabe?

3. Sonderfälle beschreiben

- Optionale Teile eines Anwendungsfalls
- Komplexe oder alternative Möglichkeiten
- Aufgaben, die nur selten durchgeführt werden

**Erweiterungen und
Alternativen erstellen**

Vorteil

- Basisfunktionalität ist leicht zu verstehen
- Komplexität wird erst im zweiten Schritt in das System integriert

Checkliste für Anwendungsfälle (4)

4. Aufsplitten komplexer Anwendungsfälle

- Komplexe Schritte als Anwendungsfall spezifizieren
 - Komplexe Anwendungsfälle (viele Sonderfälle) ...
 - in mehrere Anwendungsfälle zerlegen und
 - Gemeinsamkeiten modellieren
 - Umfangreiche Erweiterungen als Anwendungsfälle spezifizieren
- mit include**
- mit extends**

5. Gemeinsamkeiten von Anwendungsfällen ermitteln

- Auf redundanzfreie Beschreibung achten (uses)

6. Kriterien für einen guten Anwendungsfall

- Auftraggeber soll sie lesen und verstehen können
- Beschreibung der Kommunikation der Akteure mit dem System (nicht die interne Strukturen und Algorithmen)
- Standardfall immer komplett beschreiben
- Maximal eine Seite

Checkliste für Anwendungsfälle (5)

7. Konsistenz mit dem Klassendiagramm

- Objektdiagramm erstellen

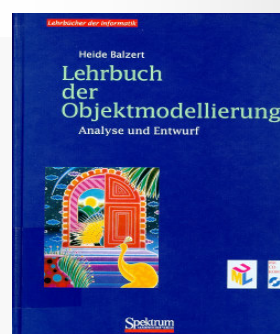
8. Fehlerquellen

- Zu kleine und damit zu viele Anwendungsfälle
- Zu frühe Betrachtung von Sonderfällen
- Zu detaillierte Beschreibung der Anwendungsfälle
- Verwechseln von *include*- und *extends*-Beziehungen
- Beschreibung von Dialogabläufen

Weitere Checklisten siehe:

Heide Balzert:

Lehrbuch der Objektmodellierung :
Analyse und Entwurf



Fragen zu 5.4 : Checklisten

- Welche der folgenden Fragen aus der Checkliste für Szenarios sind analytische oder konstruktive Schritte?

Frage	Konstruktive Schritte	Analytische Schritte
Welches sind die wichtigsten Szenarios für den Anwendungsfall?	<input checked="" type="checkbox"/>	
Wie läuft das Szenario ab (Beteiligte Klassen, Operationen, Reihenfolge)?	<input checked="" type="checkbox"/>	
Ist das Sequenzdiagramm konsistent mit dem Klassendiagramm?		<input checked="" type="checkbox"/>
Sind Details der Benutzungsoberfläche beschrieben?		<input checked="" type="checkbox"/>
Zu welcher Klasse gehören die Operationen?	<input checked="" type="checkbox"/>	
Wie ist das Szenario zu strukturieren?	<input checked="" type="checkbox"/>	
Sind die Empfänger-Objekte erreichbar (existieren Assoziationen)?		<input checked="" type="checkbox"/>

5.5 Beispiel „Waschtrockner“

Kapitel 5 Analyseprozess und Analysemuster

- 5.1 Analyseprozess
- 5.2 CRC-Karten
- 5.3 Analysemuster
- 5.4 Checklisten (zum Selbststudium)
- 5.5 Beispiel „Waschtrockner“**
- 5.6 Zusammenfassung

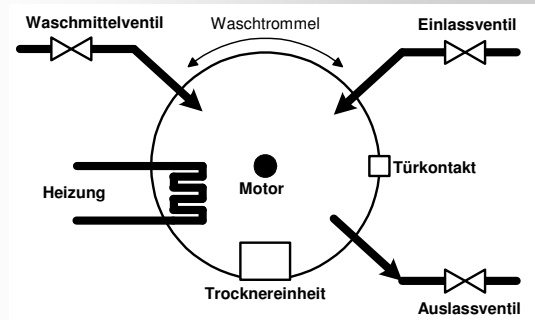
Problembeschreibung (1)

Grundfunktionalitäten der Waschmaschine:

- Waschen
 - Trommel mit Wasser befüllen und Wasser erwärmen
 - Waschmittel einfüllen
 - Waschvorgang: Drehen der Trommel
 - Wasser aus Trommel entleeren
- Spülen
 - Analog zu Waschen, aber ohne Waschmittel und mit kaltem Wasser
- Schleudern
 - Schnelles Drehen der Trommel mit fester Drehzahl
- Trocknen
 - Trocknereinheit einschalten, langsames Drehen der Trommel



Schematischer Aufbau



Problembeschreibung (2)

Waschprogramme:

- Normalwäsche
 - Waschen, Spülen, Schleudern, Trocknen
- Normalwäsche ohne Trocknen
 - Waschen, Spülen, Schleudern
- Wollwäsche
 - Waschen, Spülen, Trocknen



Bedienfeld



Sicherheits-Randbedingungen:

- Tür muss bei allen Vorgängen geschlossen sein
- Es darf nur ein Ventil gleichzeitig geöffnet sein
- Heizung darf nie bei leerer Trommel angeschaltet werden
- Beim Schleudern und Trocknen darf kein Wasser in der Trommel sein

Identifizierung von Anwendungsfällen

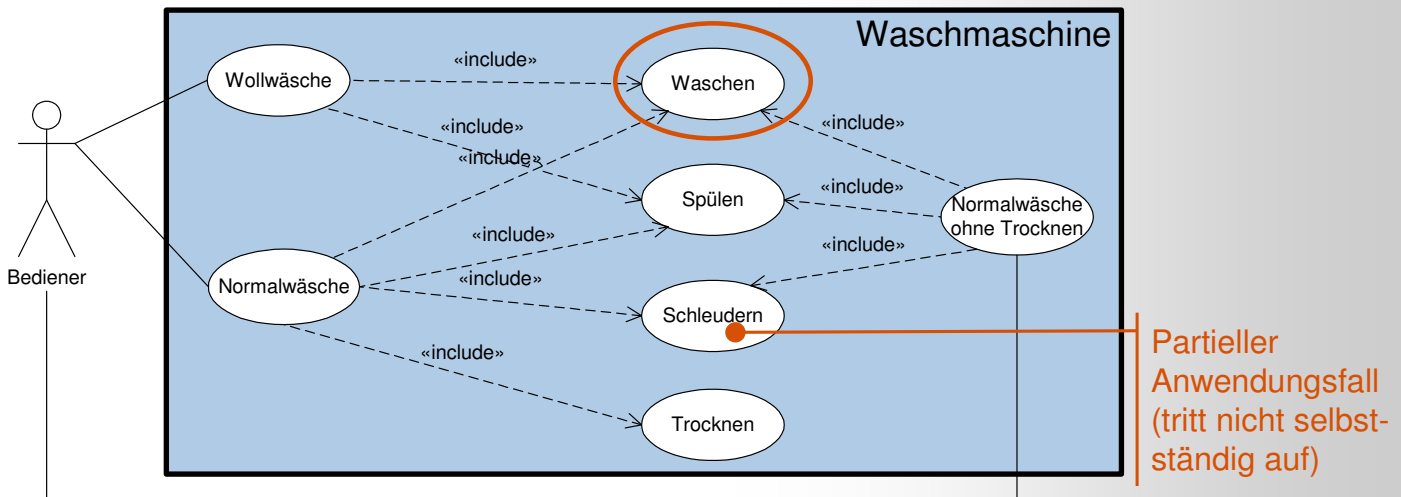
Abgeschlossene Anwendungsfälle:

- Normalwäsche
- Normalwäsche ohne Trocknen
- Wollwäsche

Partielle Anwendungsfälle:

- Waschen
- Schleudern
- Spülen
- Trocknen

Anwendungsfalldiagramm für Waschtrockner



Partieller Anwendungsfall: Waschen

Ziel:	Waschvorgang durchführen
Vorbedingung:	Alle Ventile sind geschlossen
Nachbed. Erfolg:	Waschvorgang abgeschlossen, Trommel entleert
Nachbed. Fehlschlag:	Waschvorgang nicht abgeschlossen / Trommel entleert
Akteure:	-
Auslösendes Ereignis:	Programmwahl
Beschreibung:	<ol style="list-style-type: none"> 1. Überprüfen, ob Tür geschlossen. Falls ja, Zulaufventil öffnen 2. Wenn Wasser eingelaufen: Zulaufventil schließen und Heizung auf Solltemperatur (Temperaturwahlschalter) einstellen 3. Wenn Wasser erwärmt ist, Waschmittelventil öffnen bis Waschmittel vollständig eingelaufen 4. Motor mit entsprechendem Programm starten 5. Wenn Programm beendet, Ablaufventil öffnen 6. Wenn komplettes Wasser abgeflossen, alle Ventile schließen
Erweiterungen:	-
Alternativen: :	1a Tür nicht geschlossen: Waschen nicht starten

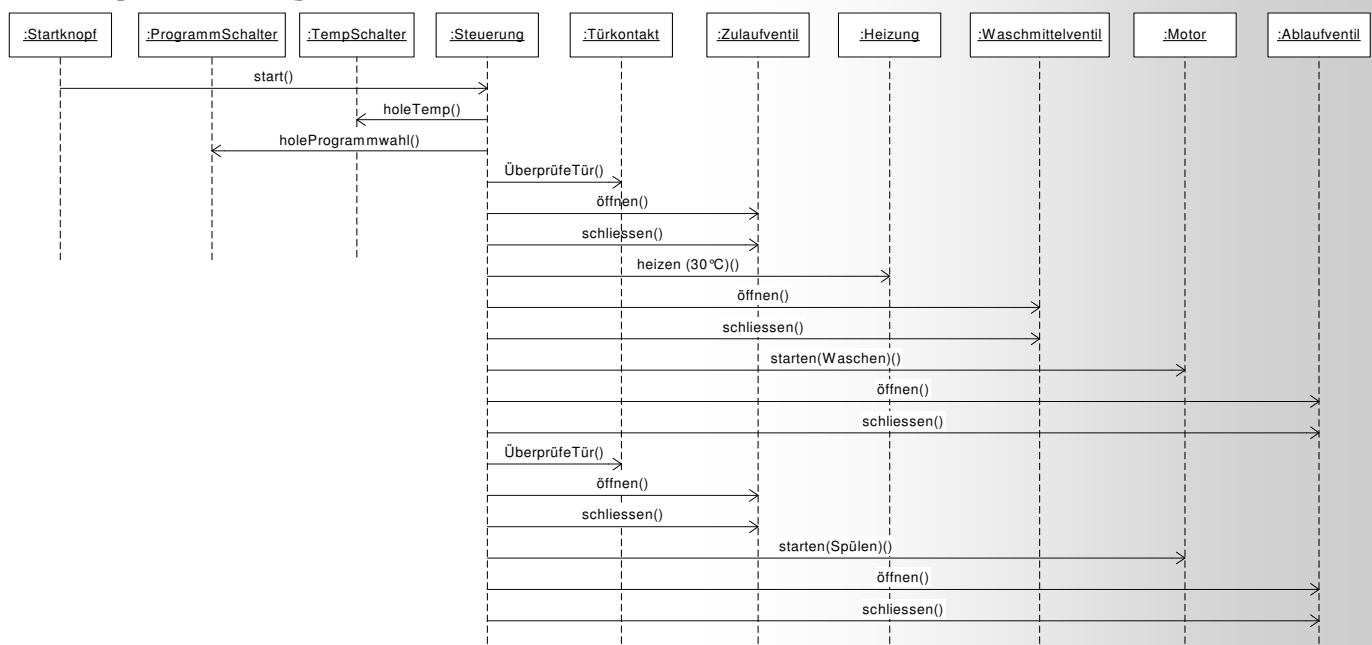
Partieller Anwendungsfall: Trocknen

Ziel:	Trockenvorgang durchführen
Vorbedingung:	Alle Ventile sind geschlossen
Nachbed. Erfolg:	Trockenvorgang abgeschlossen
Nachbed. Fehlschlag:	Trockenvorgang nicht abgeschlossen
Akteure:	-
Auslösendes Ereignis:	Programmwahl
Beschreibung:	<ol style="list-style-type: none"> 1. Überprüfen, ob Tür geschlossen. Falls ja, Ablaufventil öffnen 2. Ist Tür geschlossen, dann Trockeneinheit starten 3. Nach Abschluss des Trockenvorgangs alle Ventile schließen
Erweiterungen:	-
Alternativen: :	1a Tür nicht geschlossen: Trocknen nicht starten

Identifikation der Klassen

- Startknopf, Zulaufventil, Ablaufventil, Waschmittelventil, Heizung, Trocknereinheit, Motor, Türkontakt, Temperaturwahlschalter, Programmwahlschalter, Waschsteuerung

Sequenzdiagramm Waschen

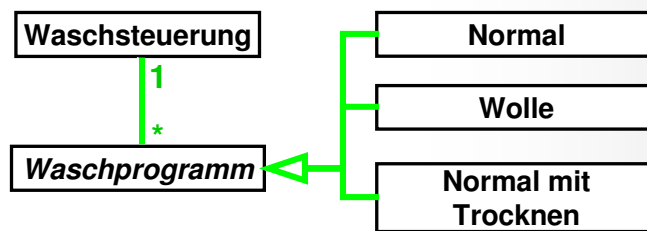


Identifizierung von Mustern:

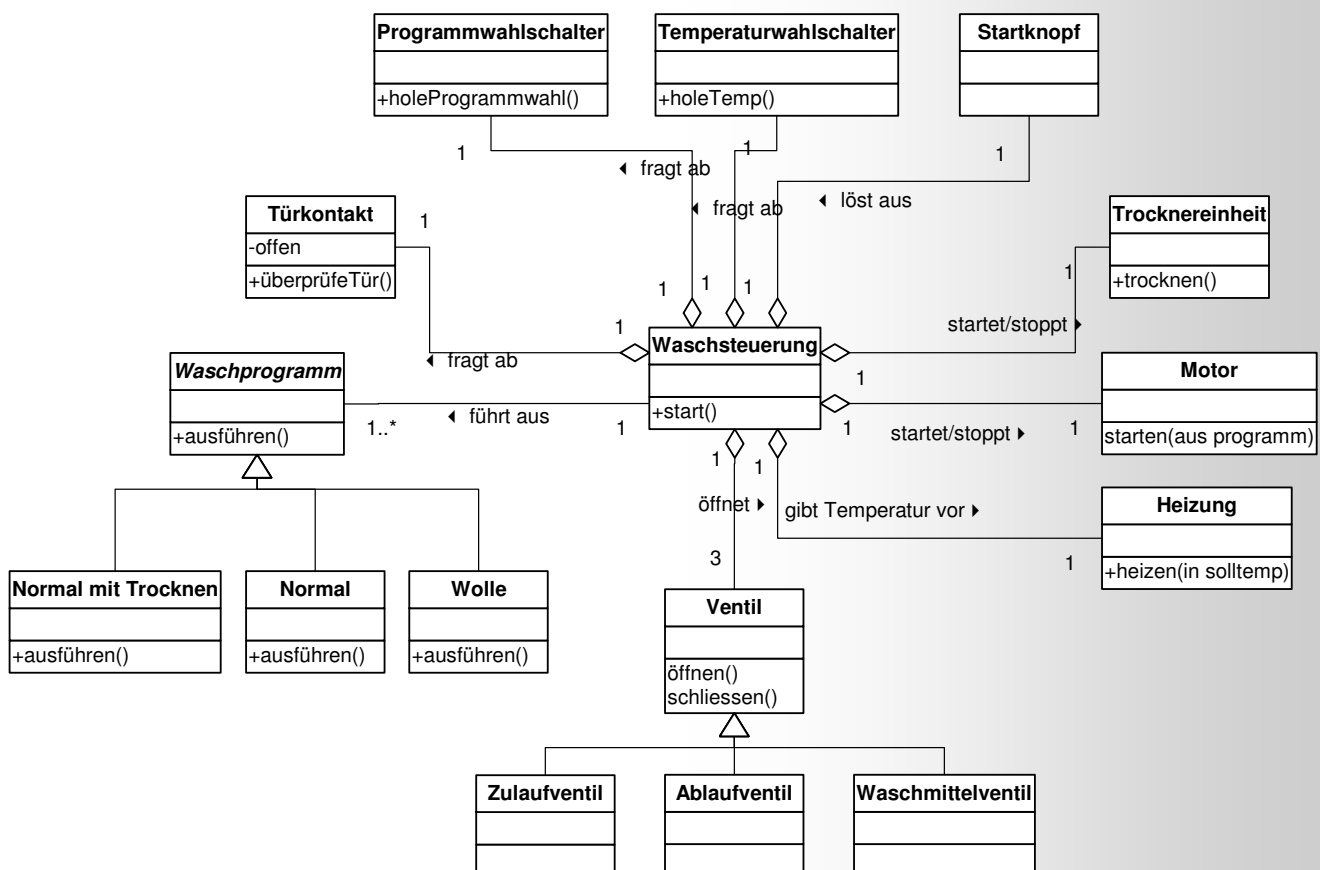
- Idee für Waschsteuerung:
 - Auslagerung der unterschiedlichen Waschprogramme

Muster Waschsteuerung - Waschprogramm (Normal mit Trocknen, Wolle, Normal)

Wechselnde Rollen

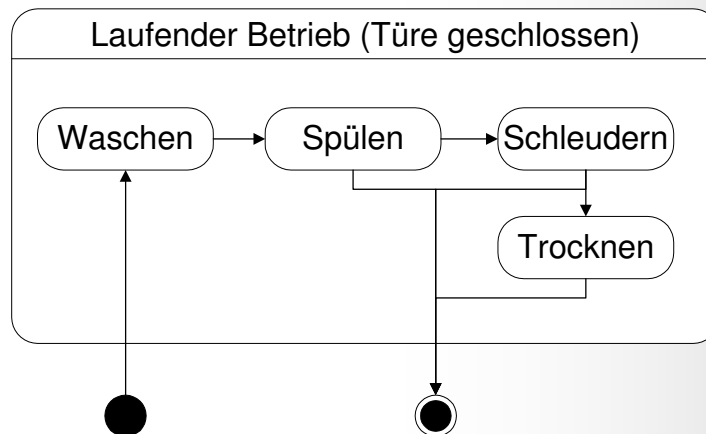


Klassendiagramm



Fragen zu 5.5 : Beispiel „Waschtrockner“

- Erstellen Sie ein einfaches Zustandsdiagramm für den laufenden Betrieb der Waschmaschine

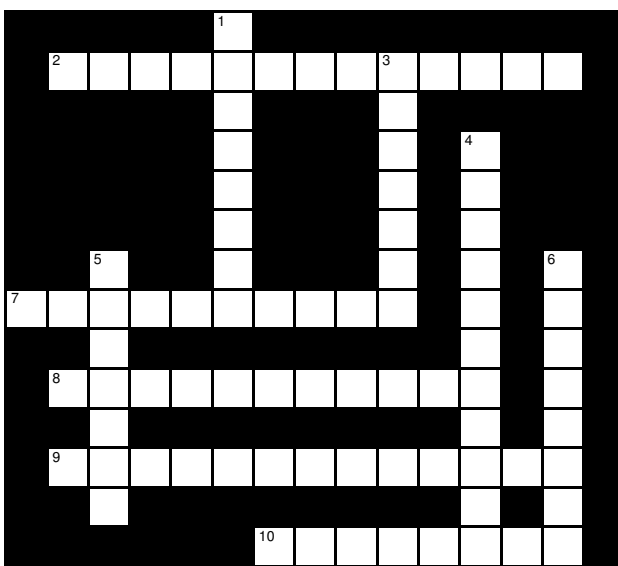
Antwort**Kapitel 5 Analyseprozess und Analysemuster**

- 5.1 Analyseprozess
- 5.2 CRC-Karten
- 5.3 Analysemuster
- 5.4 Checklisten (zum Selbststudium)
- 5.5 Beispiel „Waschtrockner“
- 5.6 Zusammenfassung**

Zusammenfassung Kapitel 5

- Der **Analyseprozess** besteht aus einem **Makroprozess**, der das Gleichgewicht von statischem und dynamischem Modell berücksichtigt
- **CRC-Karten** dienen als Hilfsmittel für den Analyseprozess zur Verbindung zwischen Anwendungsfällen, statischem und dynamischem Modell
- **Muster** ermöglichen die Standardisierung bestimmter Probleme; sie sind katalogisierte Projekterfahrungen
- Es wurden folgende **Analysemuster** beschrieben:
Liste, Exemplar, Verbund, Koordinator, Rollen, wechselnde Rollen, Historie, Gruppe, Gruppenhistorie
- Für jedes objektorientierte Konzept können **Checklisten** zur Unterstützung von Erstellung und Überprüfung eingesetzt werden.
- Die Checkliste **Anwendungsfall** zeigt, wie Anwendungsfälle ermittelt werden, wie *include* / *extends*-Beziehungen eingesetzt werden und was eine gute Beschreibung ausmacht

Frage: Kreuzworträtsel zu Kapitel 5



Senkrecht

1. Ein Anwendungsfall, der nur als Teil eines anderen Anwendungsfalls auftritt, ist ...
3. Alle Aktivitäten im Rahmen des Softwareentwicklungsprozesses bezeichnet, die der Ermittlung, Klärung und Beschreibung der Anforderungen an das System dienen
4. Assoziative Klasse in einer n-ären Assoziation
5. Komposition aus Objekten, bei der das Aggregat-Objekt und seine Teilobjekte als Einheit oder auch einzeln behandelt werden können
6. Analysemuster zur Verfolgung von Fakten über verschiedene Zeiträume

Waagrecht

2. Zusammenarbeit zwischen Klassen
7. Besteht aus Regeln und Richtlinien für die Analyse
8. Vorgehen, bei dem unter einem bestimmten Gesichtspunkt die wesentlichen Merkmale eines Gegenstandes oder Begriffes ermittelt werden.
9. Gruppe von Klassen mit feststehenden Verantwortlichkeiten und Interaktionen
10. Hilfsmittel zur Aufteilung der Verantwortlichkeiten zwischen Klassen

Kapitel 6 Konzepte und Notationen des objektorientierten Entwurfs

<u>6.1 Von der Analyse zum Entwurf</u>	<u>354</u>
<u>6.2 Konzepte des objektorientierten Entwurfs</u>	<u>365</u>
<u>6.3 Modellierung von Programmabläufen</u>	<u>384</u>
<u>6.4 Architekturentwurf</u>	<u>391</u>
<u>6.5 Entwurfsregeln und –heuristiken</u>	<u>399</u>
<u>6.6 Zusammenfassung</u>	<u>402</u>

Kapitel 6 Konzepte und Notationen des objektorientierten Entwurfs

- 6.1 Von der Analyse zum Entwurf
- 6.2 Konzepte des objektorientierten Entwurfs
- 6.3 Modellierung von Programmabläufen
- 6.4 Architekturentwurf
- 6.5 Entwurfsregeln und –heuristiken
- 6.6 Zusammenfassung

Lernziele

- UML-Notation von Klassen, Attributen, Operationen und Assoziationen im Entwurf kennen
- Sichtbarkeit spezifizieren können
- Polymorphismus verstehen und anwenden können
- Erklären können, was Schnittstellen sind
- Erklären können, was Mehrfachvererbung ist
- Objektverwaltung mittels Container-Klassen realisieren können
- Programmabläufe mittels Interaktionsdiagramme modellieren können
- Zustandsautomaten in den Entwurf transformieren können
- Zwei-, Drei- und Mehrschichten-Architekturen unterscheiden können



Kapitel 6 Objektorientierter Entwurf

6.1 Von der Analyse zum Entwurf

- 6.2 Konzepte des objektorientierten Entwurfs
- 6.3 Modellierung von Programmabläufen
- 6.4 Architekturentwurf
- 6.5 Entwurfsregeln und –heuristiken
- 6.6 Zusammenfassung

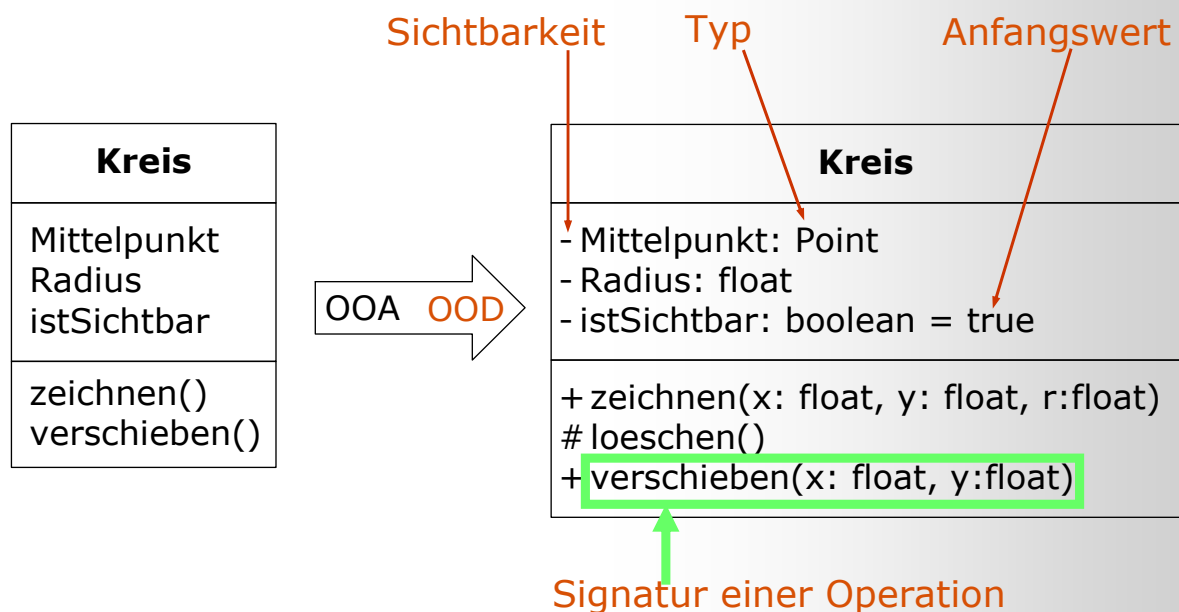
Gegenüberstellung: Entwurf und Analyse

- Ziel der Analyse
 - OOA-Modell als **fachliche Lösung** **Fachkonzept-Modell**
 - Ziel des Entwurfs
 - OOD-Modell als ...
 - Technische Lösung **Programm-Modell**
 - Spiegelbild des **Programms** auf höherem Abstraktionsniveau
 - Veränderung der Schwerpunkte beim Übergang zwischen Analyse und Entwurf
 - Analyse: Problembereich aus **Anwendersicht**
 - Entwurf: Lösungsbereich aus **Entwicklersicht**
 - Vorteil der Objektorientierung
 - Konzepte und Notation der Analyse gelten auch für Entwurf und Implementierung
 - Erweiterung der Konzepte und Notation für den Entwurf
- ⇒ **Fließender Übergang zwischen Analyse und Entwurf. Kein Strukturbruch!**

Tätigkeiten beim Entwurf

- Konsolidierung des Analysemodells
 - Verfeinerung des Modells
 - Hinzufügen zusätzlicher Details **z.B. Objektverwaltung**
 - Berücksichtigung von Randbedingungen
 - Wartbarkeit
 - Wiederverwendbarkeit
 - Technologien **z.B. CORBA, DCOM**
 - etc.
 - Erstellen einer Software-Architektur **gute Architekturen sind einfach**
 - Strukturierung des Programms
- ⇒ **Der Entwurf ist ein iterativ-inkrementeller Vorgang**

UML-Notation von Klassen im Entwurf

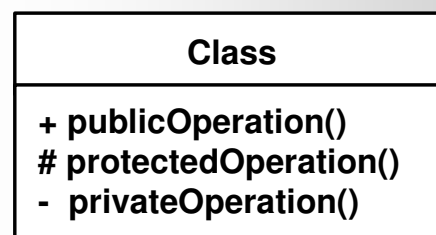
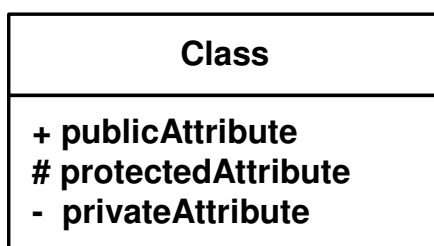


Im Entwurf werden alle Attribute und Operationen eingetragen, die im Programm enthalten sein sollen

Sichtbarkeit

- Analyse: Alle Attribute sind außerhalb der Klasse verborgen und können nur durch Operationen gelesen und geändert werden.
- Entwurf: Differenzierung der Sichtbarkeit
 - **+ public**: sichtbar für alle anderen Klassen
 - **# protected**: sichtbar innerhalb der Klasse und in Unterklassen
 - **- private**: sichtbar nur innerhalb der Klasse

Attribute sollten als *protected* oder *private* definiert werden
- Operationen: Sichtbarkeit analog Attribute



Signatur (signature) einer Operation

Sichtbarkeit Operationsname (Parameterliste): Ergebnistyp

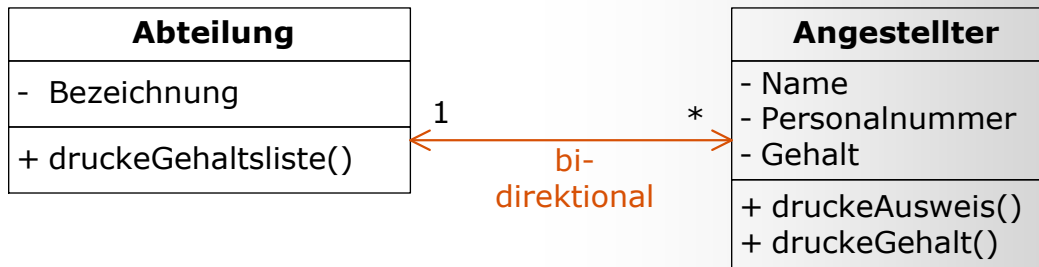
- Sichtbarkeit: [+ | # | -]
- Parameterliste: (Art Parameter-Name: Typ = Anfangswert, ... , ...)
 - Art = [in | out | inout]
- Ergebnistyp: Fehlt, wenn die Operation keinen Wert zurückgibt

Sichtbarkeit, Geheimnisprinzip und Kapselung im Entwurf

- Analyse
 - Alle Attribute sind generell außerhalb der Klasse unsichtbar
 - Kapselung = Geheimnisprinzip
- Entwurf
 - Attribute und Realisierung der Operationen können - trotz Kapselung - von außen sichtbar sein (public)
 - Angabe der Sichtbarkeit entkoppelt Kapselung und Geheimnisprinzip in Entwurf und Implementierung

Navigation von Assoziationen

- Analyse: Alle Assoziationen sind inhärent bidirektional
- Entwurf: Festlegung auf uni- oder bidirektionale **Navigation** von Assoziationen



Bei Kompositionen muss Navigation vom Ganzen zu den Teilen existieren

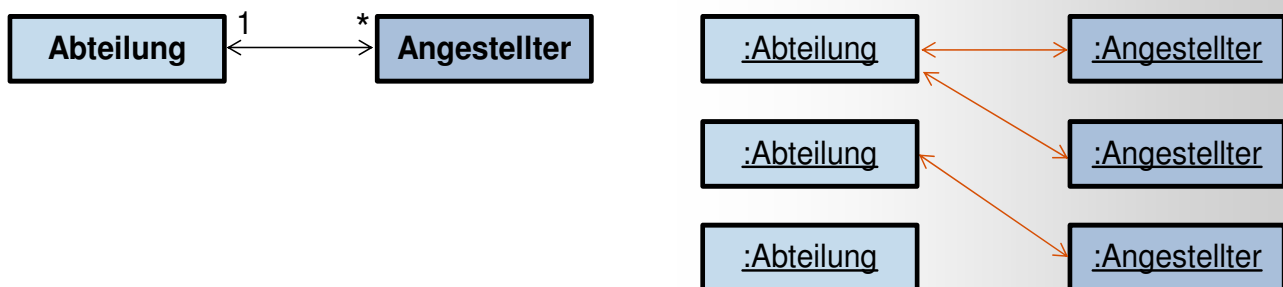


UML-Notation von Assoziationen im Entwurf

- Navigation
 - a) Alle Pfeile eintragen
 - Eine Assoziation ohne Pfeile wird nicht realisiert
 - b) Nicht alle Pfeile eintragen
 - Keine Pfeile: Assoziation wird in beiden Richtungen realisiert
 - Pfeil: Gibt Richtung der Realisierung an
 - Nur sinnvoll, wenn alle Assoziationen realisiert werden
- Kardinalität
 - Im OOD-Modell kann die Angabe der Kardinalität auf einer Seite fehlen, wenn in dieser Richtung keine Navigation stattfindet
- Sichtbarkeit
 - Für Assoziationen können in der UML zusätzlich Sichtbarkeiten angegeben werden

Assoziationen mittels Zeigern realisieren

- Realisierung jeder Richtung einer Assoziation mittels Zeigern (Referenzen) zwischen Objekten
- Jedes Objekt kennt seine assoziierten Objekte
- Konsistentes Auf- und Abbauen der Verbindungen durch die Operationen muss sichergestellt werden
- Kardinalität von 0..1 oder 1 \Rightarrow **Einzelner Zeiger**
- Kardinalität größer 1 \Rightarrow **Menge von Zeigern**



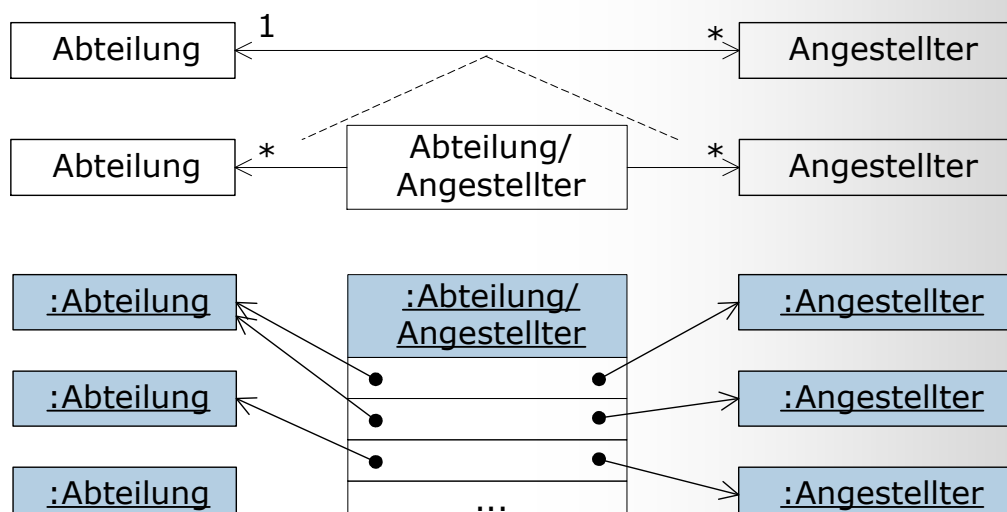
- Wenn keine Ordnung definiert ist
 - Verwendung von Container-Klassen wie `Set`, `Bag` etc. (siehe 6.2)

Assoziationen mittels Klasse realisieren

- Realisierung mittels eigener Klasse
- Assoziierte Objekte kennen sich nicht direkt
- Sinnvoll, wenn die Assoziation nachträglich hinzugefügt werden soll und die Klasse nicht verändert werden soll

Wird nicht im Klassendiagramm dargestellt

z.B. Bibliotheksklassen



Frage zu 6.1

Nennen Sie ein Beispiel für ein Programm(-Bestandteil), welches die Verkapselung erfüllt, aber nicht das Geheimnisprinzip.

Antwort

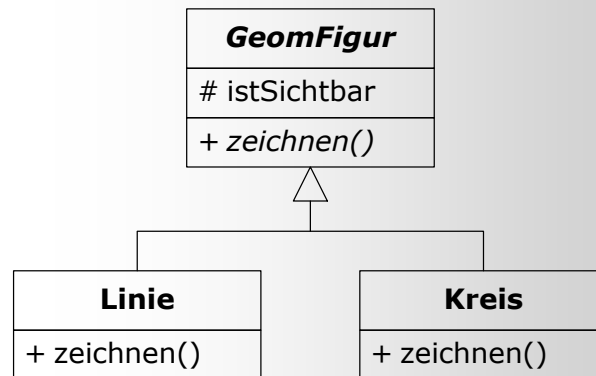
- f ☐ Objektoperationen, die auf Klassenattribute zugreifen.
- ☒ Klasse mit als *public* vereinbarten Attributen.
- f ☐ Attribute, die nicht verschlüsselt abgespeichert werden.

Kapitel 6 Objektorientierter Entwurf

- 6.1 Von der Analyse zum Entwurf
- 6.2 Konzepte des objektorientierten Entwurfs**
- 6.3 Modellierung von Programmabläufen
- 6.4 Architekturentwurf
- 6.5 Entwurfsregeln und –heuristiken
- 6.6 Zusammenfassung

Abstrakte Operation *kursiv*

- Besteht nur aus Signatur
- Besitzt keine Implementierung
- Definiert gemeinsame Schnittstelle für Unterklassen



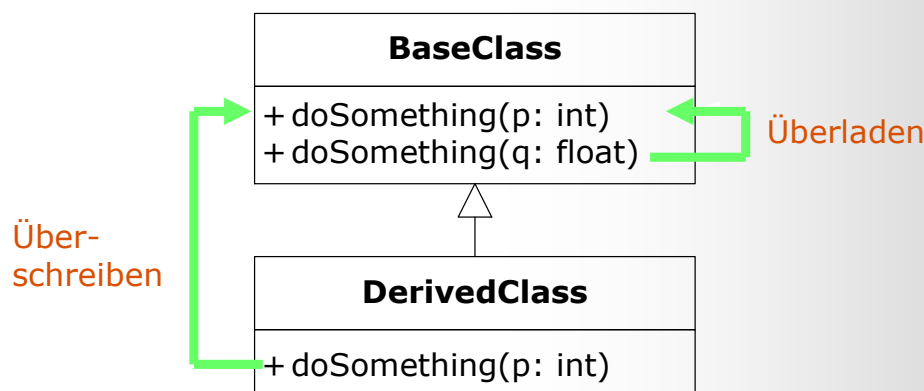
Abstrakte Klasse *kursiv*

- Kann nicht instanziiert werden
- Definiert gemeinsame Eigenschaften und Operationen für Unterklassen
- Man unterscheidet:
 - Abstrakte Klasse mit »normalen«, d.h. implementierten Operationen, die von den Unterklassen geerbt werden
 - Abstrakte Klasse mit einer oder mehreren – abstrakten – Operationen, die in den Unterklassen implementiert werden müssen

Die Oberklasse einer abstrakten Klasse ist auch eine abstrakte Klasse

Überschreiben vs. Überladen von Operationen

- **Überladen** (*overloading*): Verwendung von Operationen mit demselben Namen – jedoch unterschiedlicher Semantik und Implementierung
 - $5 + 2 \rightarrow 7$
 - »alpha« + »bet« \rightarrow »alphabet«
- **Überschreiben**: Redefinition einer Operation durch die Unterklasse
 - Anzahl und Typen der Ein-/Ausgabeparameter bleiben gleich

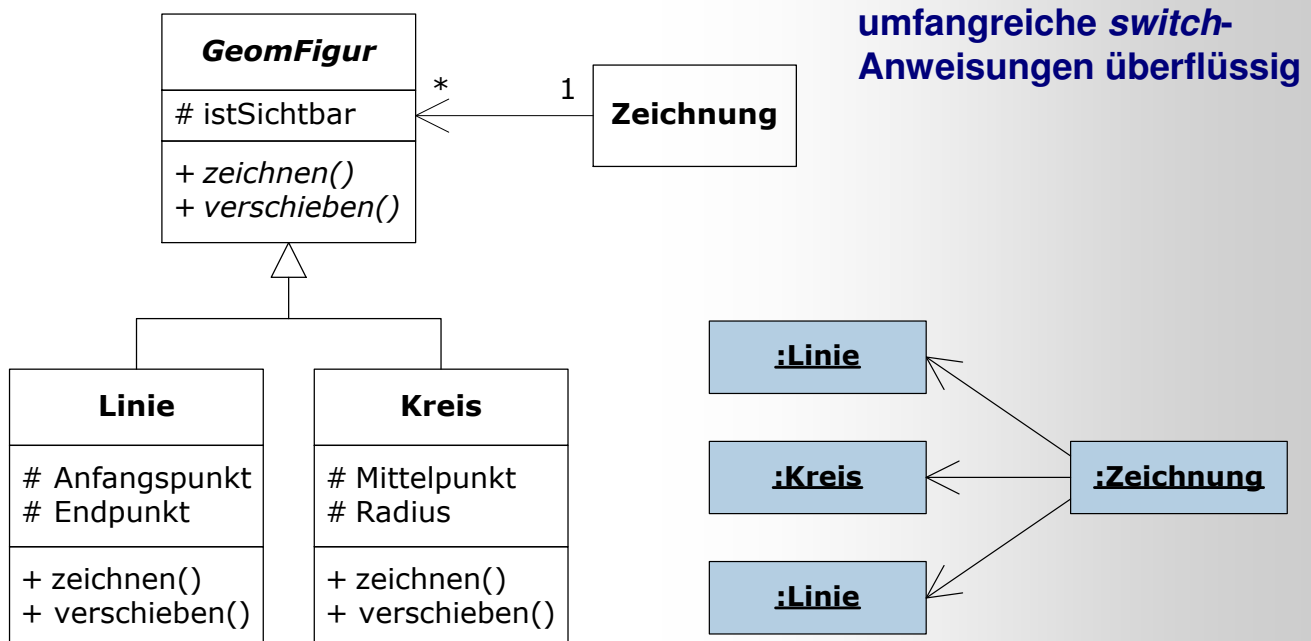


Polymorphismus

Wiederholung:

- Polymorphismus ermöglicht es, den gleichen Namen für gleichartige Operationen zu verwenden, die auf Objekten verschiedener Klassen auszuführen sind
- Das bedeutet:
 - Variablendeklaration kann Objekte verschiedener Klassen bezeichnen
 - Klassen müssen durch eine gemeinsame Oberklasse verbunden sein
 - Jedes Objekt, auf das sich diese Variable beziehen kann, kann auf die gleiche Botschaft auf seine eigene Art und Weise reagieren
- Der Sender der Botschaft muss nur wissen,
 - dass ein Empfängerobjekt das gewünschte Verhalten besitzt
 - Durch gemeinsame Oberklasse sichergestellt
 - aber nicht, zu welcher Klasse das Objekt gehört

Beispiel für Polymorphismus



⇒ Polymorphismus ermöglicht die Entwicklung flexibler und leicht änderbarer Softwaresysteme

Polymorphismus: Frühes Binden vs. spätes Binden

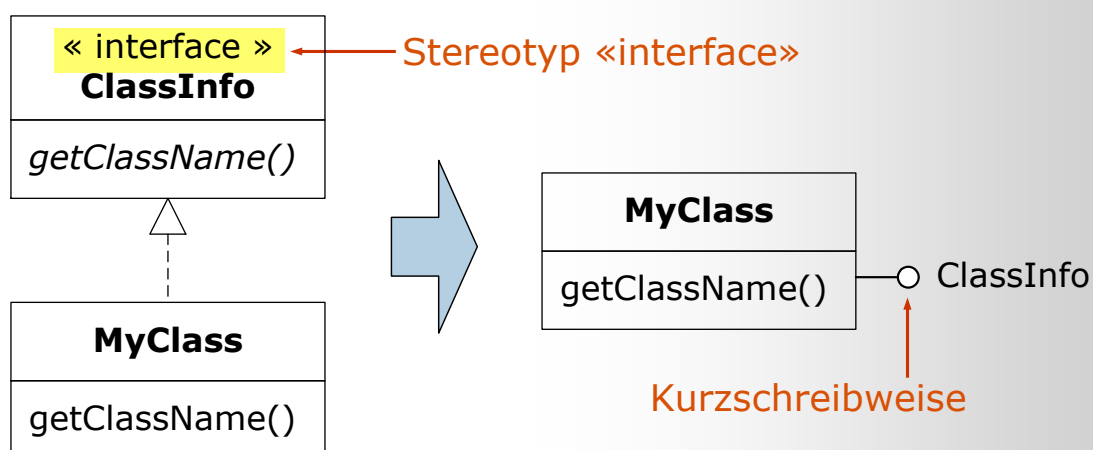
- **Frühes Binden** (statisches Binden)
 - Auszuführende Operation wird zur Übersetzungszeit bestimmt
 - Bei den meisten herkömmlichen Programmiersprachen bekannt, die strenge Typisierung besitzen **Monomorphismus**
 - Polymorphismus nur in Form von Überladen (overloading) von Operationen vorhanden
- **Spätes Binden** (dynamisches Binden)
 - Auszuführende Operation wird erst zur Laufzeit des Programms einer Klasse zugeordnet
 - Möglichkeit, Objekte mit identischen Schnittstellen zur Laufzeit beliebig auszutauschen

Polymorphismus benötigt spätes Binden

Schnittstellen

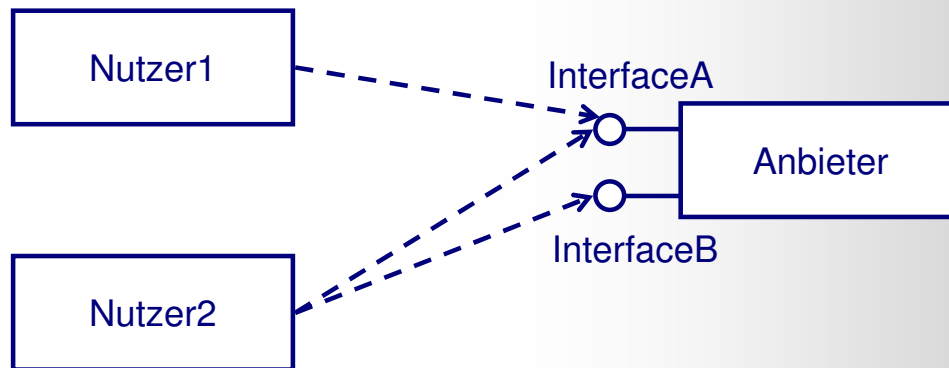
Definition: Eine Schnittstelle (interface) ...

- spezifiziert einen Ausschnitt aus dem Verhalten einer Klasse
- besteht nur aus abstrakten Operationen (**vgl. abstrakte Klasse**)
- wird durch Klassen realisiert / implementiert (gestrichelter Vererbungspfeil)



⇒ **Schnittstellen sind Verträge zwischen Klassen** Sie können nicht einfach geändert werden!

Notation der Nutzung einer Schnittstelle (Tafelanschrieb)



Einfachvererbung vs. Mehrfachvererbung

– Einfachvererbung

Baumstruktur

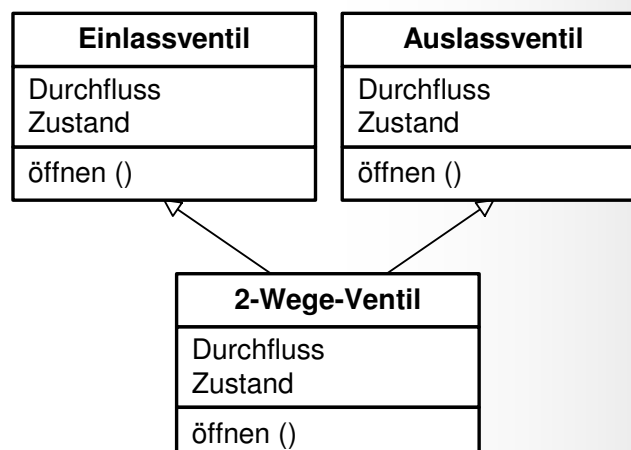
- Jede Klasse besitzt höchstens eine direkte Oberklasse

Mehrfachvererbung

Netzstruktur

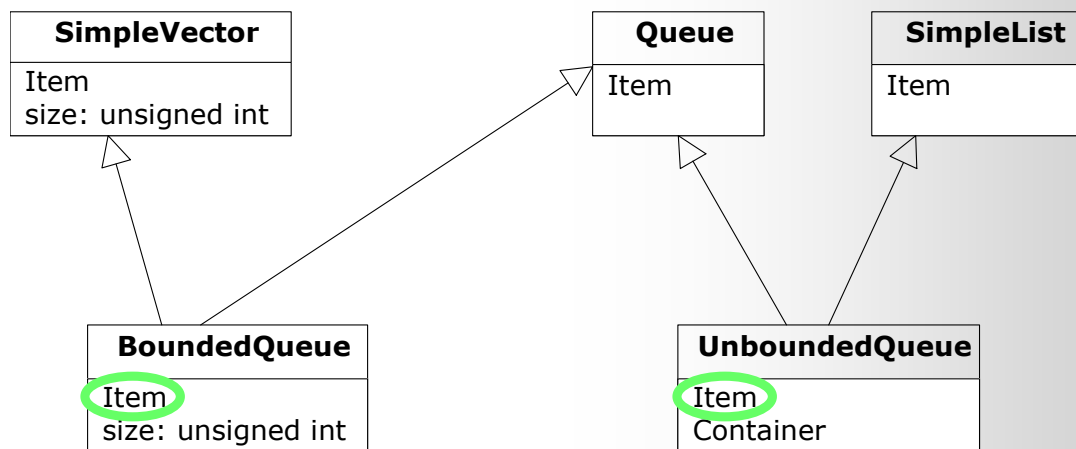
- Jede Klasse kann mehrere direkte Oberklassen besitzen

Beispiel



Mehrfachvererbung

- Jede Klasse kann mehrere direkte Oberklassen besitzen
- Azyklisches Netz mit einer oder mehreren Wurzeln



⇒ **Konflikt: Klasse kann von verschiedenen Oberklassen Attribute oder Operationen gleichen Namens aber unterschiedlichen Inhalts erben!**

Eigenschaften der Mehrfachvererbung

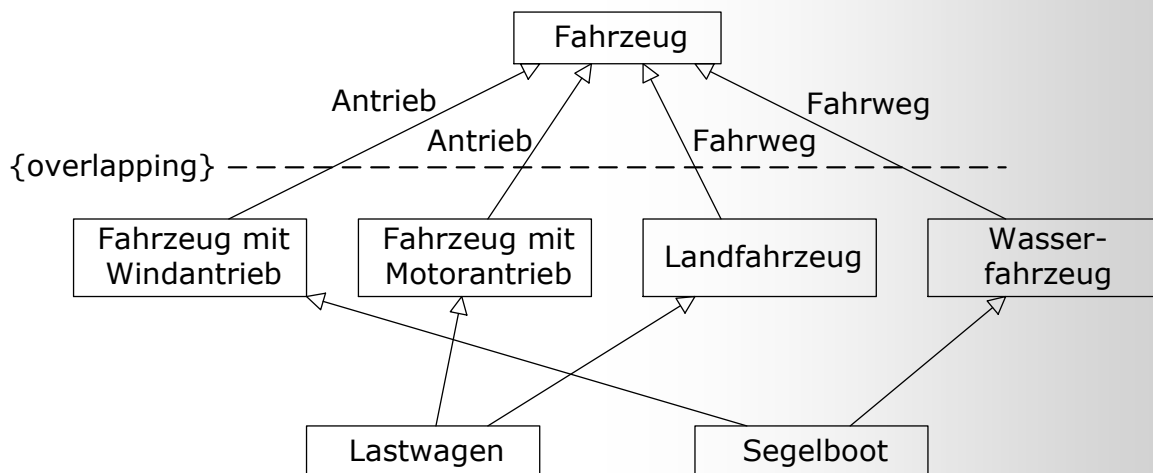
- Probleme der Mehrfachvererbung
 - Gefahr der Spaghetti-Vererbung
 - Vererbungsstruktur, die nur sehr schwierig zu verstehen und zu warten ist
 - Ähnlich einer Spaghetti-Programmierung
- Mehrfachvererbung ist ein mächtiges Instrument
 - Wird nur selten benötigt
 - Bei Problemen, die anders nur sehr schwierig zu lösen sind

Java kennt Mehrfachvererbung nur über Schnittstellen

Restriktionen bei Vererbung

- **overlapping**: Eigenschaften der Unterklassen überschneiden sich
- **complete**: Menge der Unterklassen ist vollständig
- **disjoint**: Eigenschaften der Unterklassen überschneiden sich nicht
- **incomplete**: Es gibt weitere Unterklassen, die noch nicht enthalten sind

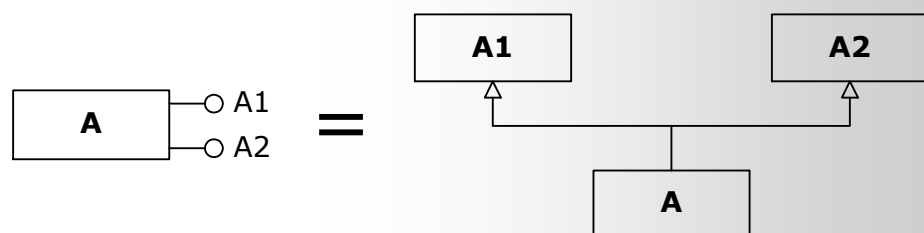
Sowohl bei Einfach- als auch bei Mehrfachvererbung



Delegation als Alternative zur Mehrfachvererbung

- Delegation ist ein Mechanismus, bei dem ein Objekt eine Nachricht nicht vollständig selbst interpretiert, sondern an ein anderes Objekt weiterleitet
- Vermeidung von Mehrfachvererbung mittels Delegation durch
 - Auslagerung von Eigenschaften, die bei einer Vererbungsbeziehung in der Oberklasse anzusiedeln wären, in separate Klassen
 - Einbindung dieser separaten Klassen per Aggregation

- Vererbung:



- Delegation:

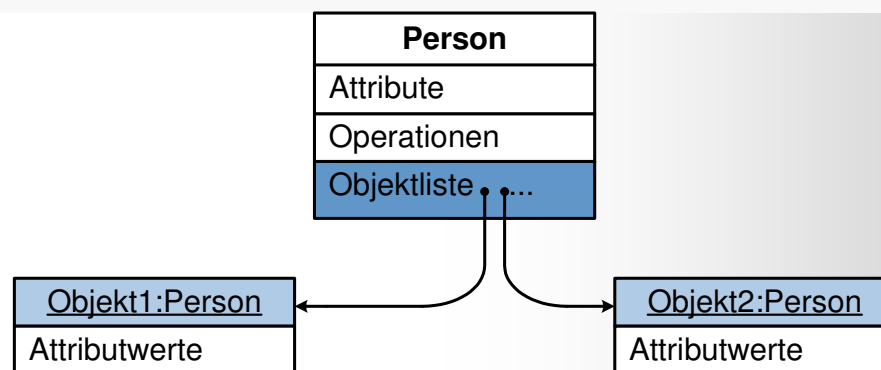


Objektverwaltung

- Wiederholung: Jedes Objekt "**weiß**", zu welcher Klasse es gehört
- Eine Klasse kennt von Natur aus die von ihr erzeugten Objekte **nicht**!
- **Objektverwaltung** bedeutet, die Klasse "führt Buch" über das Erzeugen und Löschen ihrer Objekte.
- Die Klasse erhält dabei die Möglichkeit, Anfragen und Manipulationen auf der Menge der Objekte einer Klasse durchzuführen.

Objektverwaltung nur in bestimmten Fällen sinnvoll

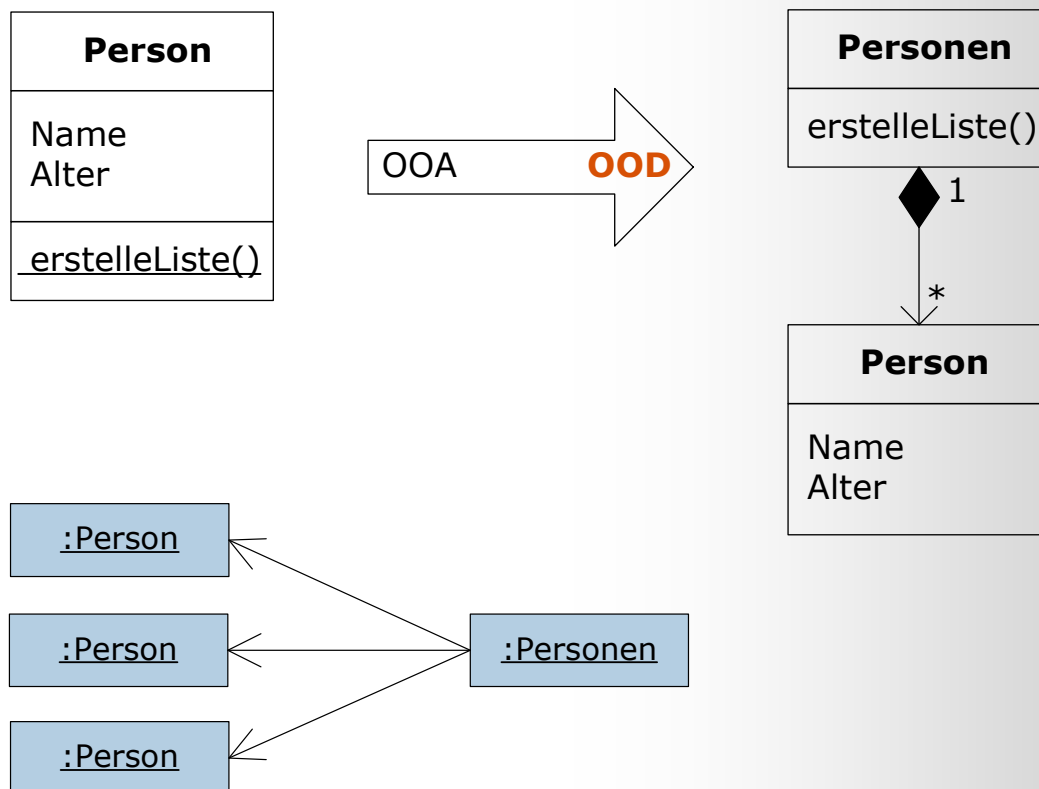
- Beispiel für Objektverwaltung durch Klasse Person



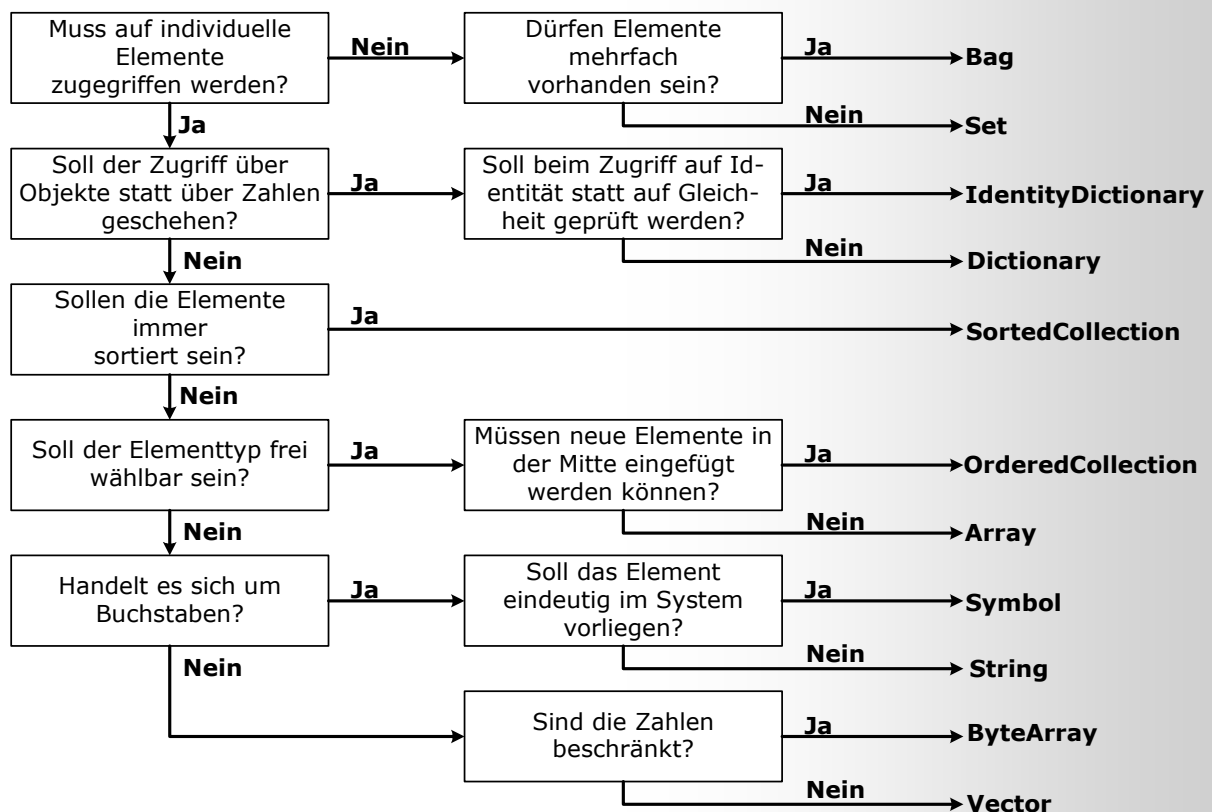
Objektverwaltung mit Hilfe von Container-Klassen

- Objektverwaltung wird mit Hilfe zusätzlicher Container-Klassen realisiert
- Container-Klassen ... **(werden auch Sammlungen genannt)**
 - verwalten eine Menge von Objekten einer anderen Klasse
 - stellen Operationen bereit, um auf die verwalteten Objekte zuzugreifen
 - *Container* = Objekt der Container-Klasse
- Container-Klassen unterscheiden sich in Hinblick auf
 - Sortiermöglichkeiten
 - Zugriffsmechanismen
 - Typbeschränkungen
 - Zulassung von Duplikaten
 - ...
- Beispiele für Container sind
Felder (arrays), Mengen (sets)

Beispiel für Objektverwaltung mit Containerklasse

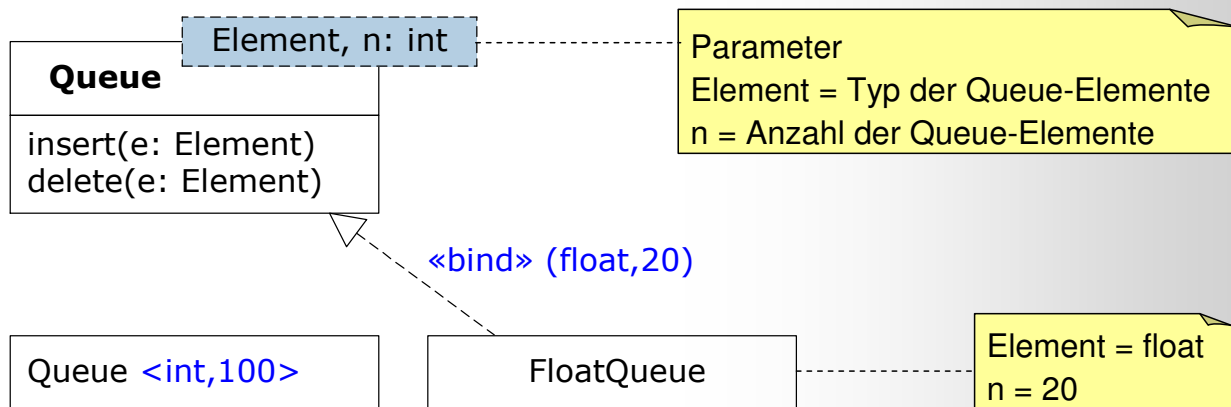


Beispiel Übersicht Container-Klassen



Generische Klassen (*parameterized class, template*)

- Klasse mit einem oder mehreren formalen Parametern
- Mögliche Parameter (**Parameterliste darf nicht leer sein**)
 - Typ
 - Variable und Typ
- Damit eine generische Klasse benutzt werden kann, müssen deren formale Parameter an aktuelle Parameter gebunden werden



- Generische Klasse existieren nicht in Java

Vorführung zu §6.2 Softwareentwurf

Frage zu 6.2

Was ist der Unterschied zw. einer Schnittstelle und einer abstrakten Klasse?

Antwort

Im Gegensatz zu einer Schnittstelle hat eine abstrakte Klasse:

- Attribute, nicht-abstrakte Operationen, Assoziationen

Welche Aussagen zum Polymorphismus-Konzept sind richtig?

- ☒ Der Sender einer Botschaft muss nicht wissen, zu welcher Klasse das Empfängerobjekt gehört
- ☒ Auf das Versenden derselben Botschaft an Objekte unterschiedlicher Klassen einer Vererbungshierarchie kann unterschiedlich reagiert werden
- ☐ Große Mehrfachauswahl-Anweisungen können generell nicht durch Polymorphismus ersetzt werden
- ☐ Das Konzept des "späten Bindens" ist für Polymorphismus nicht relevant
- ☒ Durch "spätes Binden" wird es möglich, Objekte mit identischen Schnittstellen zur Laufzeit beliebig auszutauschen

Kapitel 6 Objektorientierter Entwurf

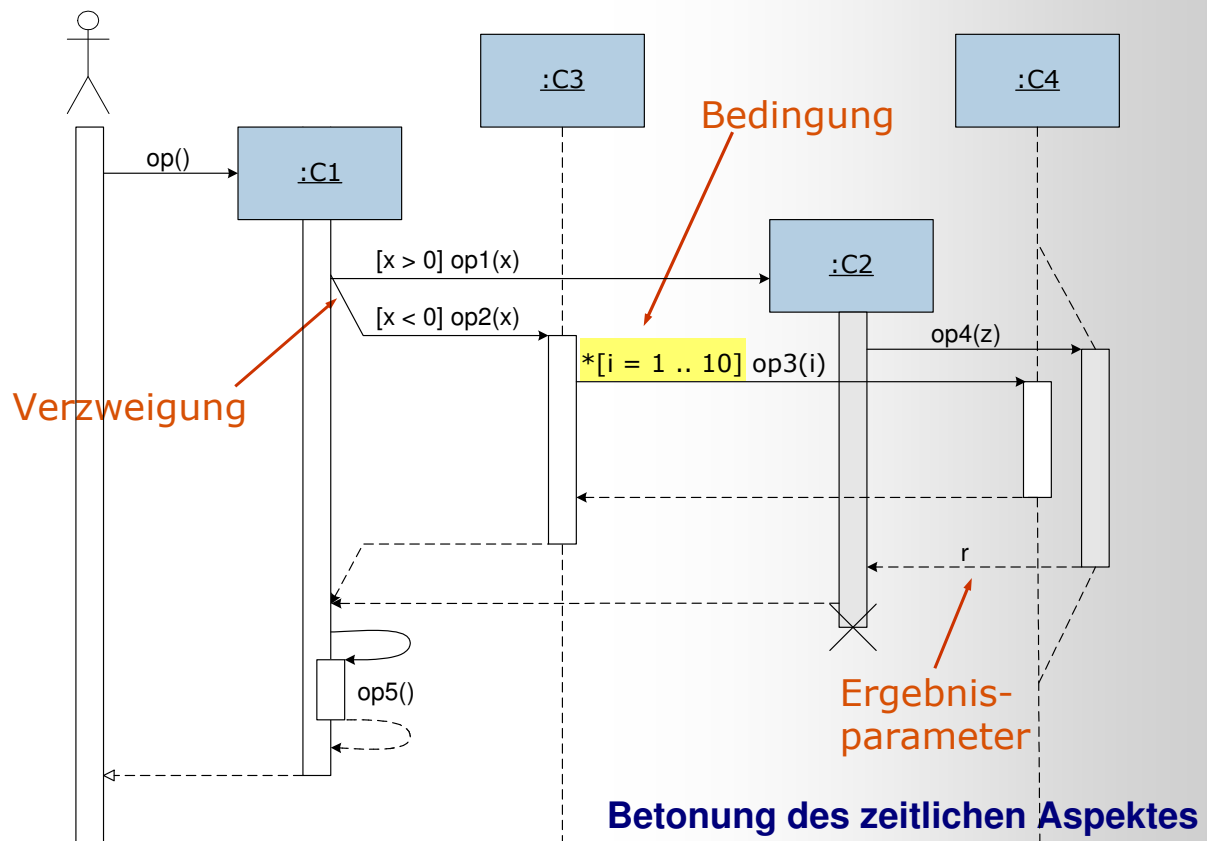
- 6.1 Von der Analyse zum Entwurf
- 6.2 Konzepte des objektorientierten Entwurfs
- 6.3 Modellierung von Programmabläufen**
- 6.4 Architekturentwurf
- 6.5 Entwurfsregeln und –heuristiken
- 6.6 Zusammenfassung

Objektinteraktionen

- Grundmuster bei Objektinteraktionen
 - Zentral gesteuerte Vorgangssteuerung
 - Zentrales Objekt legt Abarbeitungsreihenfolge fest
 - Kooperative Vorgangssteuerung
 - Jedes Objekt hat einen eigenen Aufgabenbereich
 - Abarbeitungsreihenfolge ergibt sich aus gegenseitigen Objektaufrufen
- Darstellung von Objektinteraktionen
 - Sequenzdiagramme
 - Betonen zeitlichen Aspekt
 - Kollaborationsdiagramme
 - Betonen Beziehungen der Objekte und deren Topologie

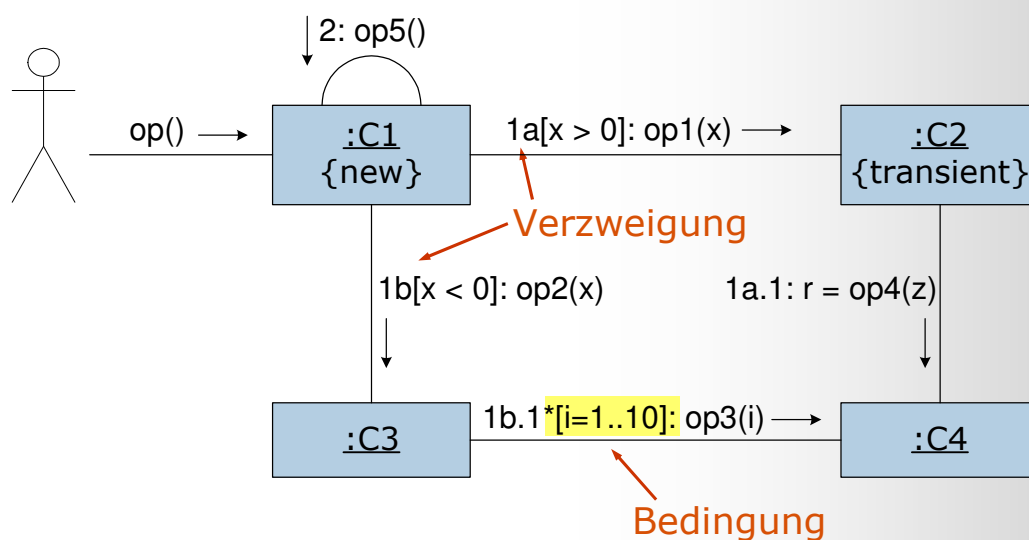
**Kooperative Vorgangssteuerung
ist vorzuziehen**

Sequenzdiagramme im Entwurf



Kollaborationsdiagramme im Entwurf

Betonung der Beziehungen der Objekte



Verwendung von *Multi-Objekt-Symbolen* möglich:

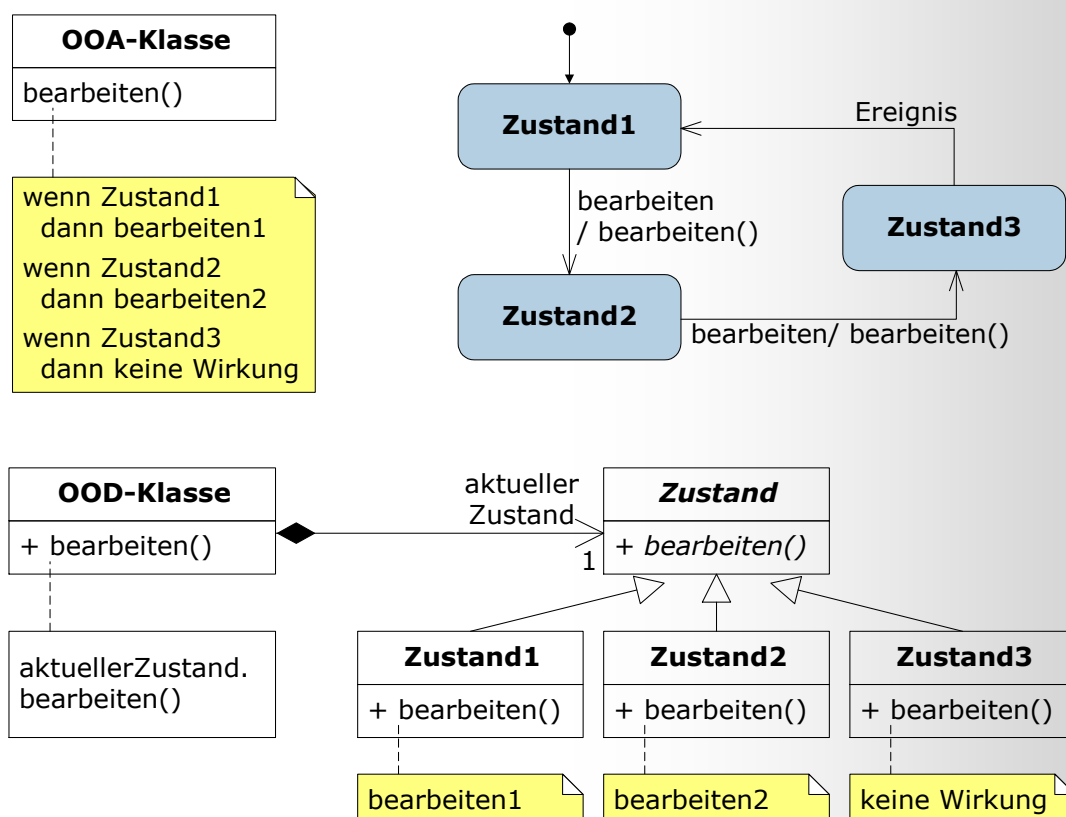
:Objekt

Transformation von Zustandsautomaten im Entwurf

– Ein Zustandsautomat kann nicht direkt in eine Programmiersprache umgesetzt werden. Mögliche Realisierungen sind ...

- Einfache Realisierung mit Zustandsattribut
 - Zustandsattribut speichert aktuellen Zustand des Objekts
 - Jede Operation muss dieses Attribut abfragen
 - Aktualisierung des Zustandsattributs wenn mit der Operation ein Zustandswechsel verbunden ist
- Einfache Realisierung mit Ereignis-interpretierender Operation
 - Operation interpretiert eintreffende Ereignisse
 - Operation löst entsprechende Verarbeitung aus
- Realisierung komplexer Objekt-Lebenszyklen mittels Zustandsmuster
 - Kapselung eines kompletten Zustands in einer Klasse
 - Leichtes Hinzufügen neuer Zustände und Zustandsübergänge
 - Nachteil: Erhöhung der Anzahl der Klassen

Realisierung Zustandsautomat mit Zustandsmuster



Frage zu 6.3

Welche Eigenschaften kennzeichnen die Verwendung Zustandsmuster zur Realisierung von Zustandsautomaten?

- ☐ Verwendung eines globalen Zustandsattributs
- ☒ Kapselung eines einzelnen Zustands in einer Klasse
- ☒ Vorteil: Leichtes Hinzufügen neuer Zustände und Zustandsübergänge
- ☐ Fachobjekt muss prüfen, in welchem Zustand es sich aktuell befindet
- ☒ Fachobjekt delegiert Ausführung einer aufgerufenen Operation an aktuelles Zustandsobjekt

Kapitel 6 Objektorientierter Entwurf

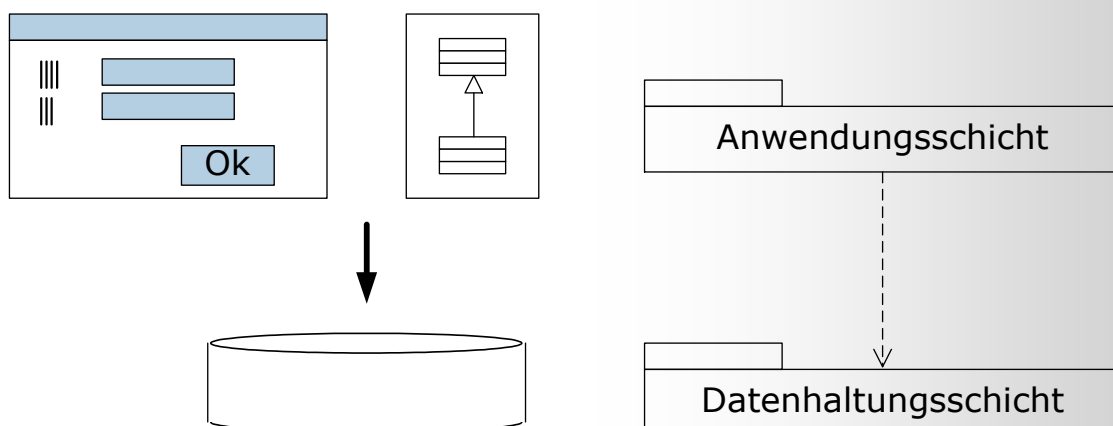
- 6.1 Von der Analyse zum Entwurf
- 6.2 Konzepte des objektorientierten Entwurfs
- 6.3 Modellierung von Programmabläufen
- 6.4 Architekturentwurf**
- 6.5 Entwurfsregeln und –heuristiken
- 6.6 Zusammenfassung

Bedeutung des Architekturentwurfs

- Ziel einer Software-Architektur ist es, ein Entwurfsmodell zu entwickeln, welches das fachliche Modell unter Berücksichtigung bestimmter Randbedingungen umsetzt:
 - Ausschluss von Risiken, Wartbarkeit, Wiederverwendbarkeit ...
- Wesentliches Kennzeichen einer guten Software-Architektur ist die **Struktur**
 - Klassen sind nicht die richtige Granularität. Ein komplexes System kann aus Tausenden von Klassen bestehen.
 - Schichten eignen sich als Strukturierungsmittel **Realisierung z.B. durch Pakete**
- ⇒ **Software-Architekturen sind langlebig. Erfolgreiche Projekte zeichnen sich durch eine durchdachte Software-Architektur aus.**
- Das System sollte so strukturiert werden, dass möglichst **wenige Abhängigkeiten** entstehen.
 - Abhängigkeiten haben einen negativen Einfluss auf die Wartbarkeit, Entwicklungszeit, Wiederverwendung etc.

Zwei-Schichten-Architektur

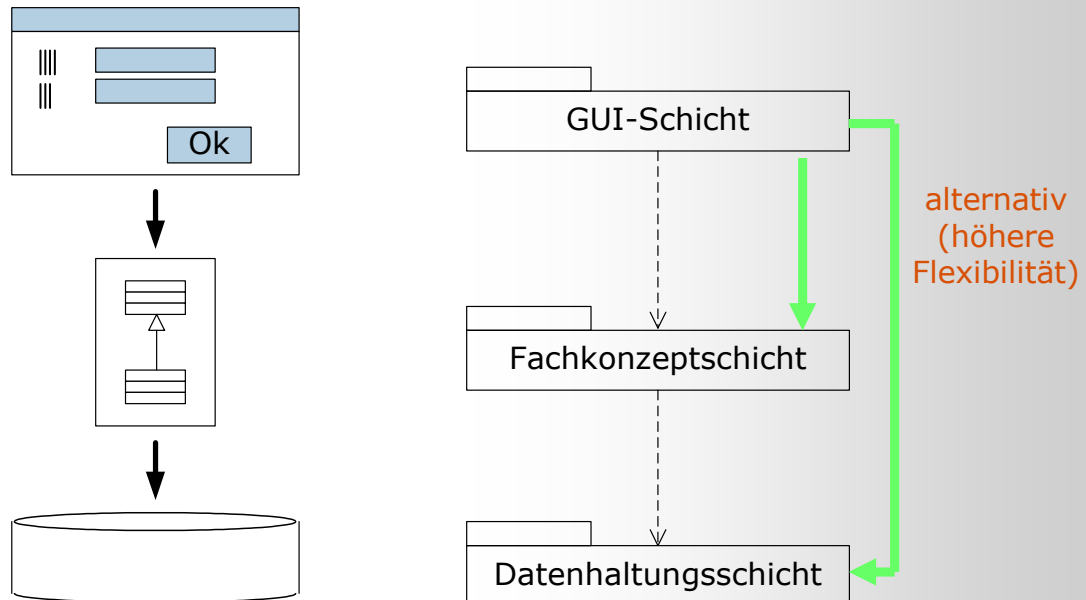
- Anwendungsschicht enthält: **gängige Architekturform z.B. Visual Basic**
 - Fachkonzept
 - Benutzungsoberfläche
- Datenhaltung in der Datenhaltungsschicht



Nachteil: Fachkonzept und Benutzungsoberfläche nicht getrennt

Drei-Schichten-Architektur

- Entkopplung von Fachkonzept und Benutzungsoberfläche
 - Getrennte Entwicklung von Benutzungsoberfläche und Fachkonzept
 - Einfacher Austausch der Benutzungsoberfläche möglich



Modell-View-Control-Architekturkonzept (1)

- Grundlegende Idee der Schichten-Architektur
 - Kein direkter Zugriff auf die Benutzungsoberfläche durch andere Schicht
 - Fachkonzeptschicht besitzt kein Wissen über die Benutzungsoberfläche
- Wie erhält ein Fenster Informationen aus dem Fachkonzept?
 - **Polling:** Benutzungsoberfläche fragt regelmäßig nach Änderungen der Fachkonzeptobjekte
 - **Indirekte Kommunikation:**
 - Fachkonzeptobjekt benachrichtigt die Benutzungsoberfläche über Änderungen
 - Oberfläche holt sich selbstständig die notwendigen Daten
- **MVC-Architektur (Model / View / Control)** zur Entkopplung von Benutzungsoberfläche und Fachkonzept

Modell-View-Control-Architekturkonzept (2)

– View-Objekt

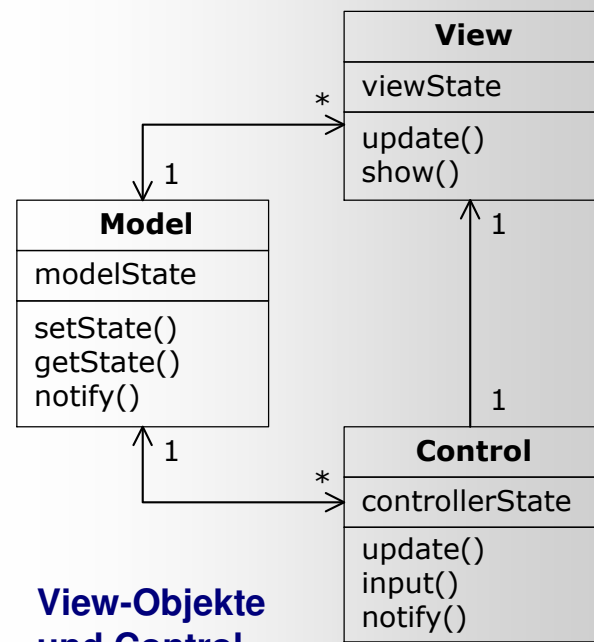
- Präsentation der fachlichen Daten

– Control-Objekt

- Reaktion auf Benutzereingaben
- Aufruf Modellobjekte

– Model-Objekt

- Fachkonzeptobjekt
- Hat keinen direkten Zugriff auf assoziierte View- und Control-Objekte
- Besitzt Liste aller von ihm abhängigen Objekte und muss diese bei einer Aktualisierung benachrichtigen



**View-Objekte
und Control-
Objekte bilden
Paare**

Mehrschichtenarchitektur

– GUI-Schicht

**Bessere Trennung von
GUI und Fachkonzept**

- Präsentation der Informationen

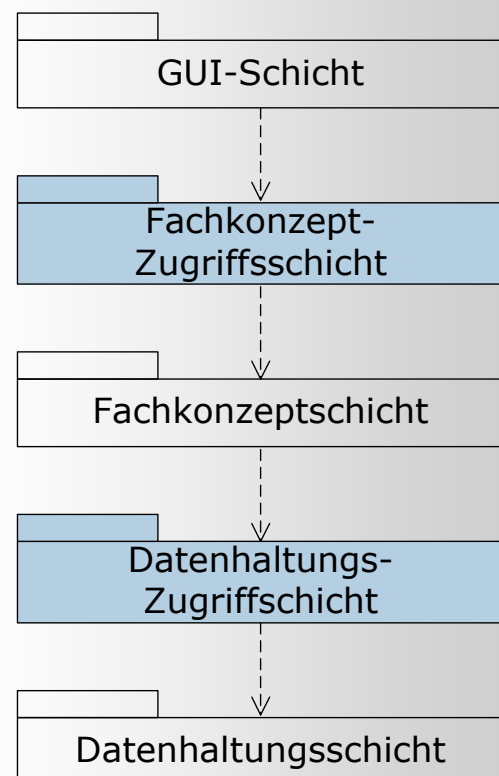
– Fachkonzept-Zugriffsschicht

- Zugriffe auf die Fachkonzeptschicht

– Datenhaltungs-Zugriffsschicht

- Aktualisiert Datenbank
- Liest Daten aus Datenbank

**Fachkonzeptschicht muss sich
nicht um Datenbankzugriff
kümmern**



Frage zu 6.4

Welche Vorteile bietet eine Schichtenarchitektur?

Antwort

- Schichten bieten
 - geeignete Granularität zur Grob-Strukturierung einer Software.
- Entkopplung von Datenhaltung, Fachkonzept und Benutzeroberfläche (bei Verwendung einer Drei-Schichten-Architektur) haben positiven Einfluss auf
 - Wartbarkeit
 - Entwicklungszeit
 - Wiederverwendung
 - Portabilitäteiner Software.

**Kapitel 6 Objektorientierter Entwurf**

- 6.1 Von der Analyse zum Entwurf
- 6.2 Konzepte des objektorientierten Entwurfs
- 6.3 Modellierung von Programmabläufen
- 6.4 Architekturentwurf
- 6.5 Entwurfsregeln und -heuristiken**
- 6.6 Zusammenfassung

Hinweis: Die folgende Aufzählung von Entwurfsregeln- und heuristiken basiert auf

Oestereich, Bernd: *Objektorientierte Softwareentwicklung: Analyse und Design mit der Unified Modeling Language*

4. Auflage, Oldenbourg Verlag, München Wien, 1998

- Entwerfen Sie kohärente Operationen die nur eine einzelne Aufgabe erfüllen. Der Code einer Operation sollte eine Seite nicht überschreiten.
- Verzichten Sie auf Nebeneffekte: Arbeiten Sie in Operationen nicht mit globalen Variablen u.ä. sondern übergeben Sie solche Informationen als Parameter.
- Anstelle von Funktionsmodi sollten besser separate Operationen bereitgestellt werden.
- Eine Unterklasse muss alle Attribute, Operationen und Beziehungen ihrer Oberklasse unterstützen. Eine Unterdrückung dieser Eigenschaften ist zu vermeiden.
- Sofern geerbte Operationen überschrieben werden, sollten sie zum Verhalten der überschriebenen Operation kompatibel sein.

- Streben Sie eine gleichmäßige Verteilung des Wissens über den Anwendungsbereich über alle Klassen an.
- Entwerfen Sie möglichst allgemeingültige Konzepte, d.h. entwerfen Sie in Hinblick auf die Schnittstellen statt auf die Implementierung.
- Client-Server-Beziehungen zwischen den Klassen entwerfen (Kooperationsprinzip).
- Maximieren Sie die innere Bindung von Klassen. Zusammengehörige Verantwortlichkeiten sind in einer Klassen zu konzentrieren.
- Minimieren Sie die äußeren Abhängigkeiten einer Klasse. Halten Sie die Anzahl der verschiedenen Schnittstellen mit anderen Klassen klein.
- Trennen Sie Fachklassen und Ausprägungsklassen.
- Sorgen Sie für einheitliche und treffende Namen, Datentypen und Parameterreihenfolgen.
- Berücksichtigen Sie Extremwerte. Planen Sie ein robustes Verhalten in allen Situationen.
- Berücksichtigen Sie unternehmensspezifische und allgemeine Standards.

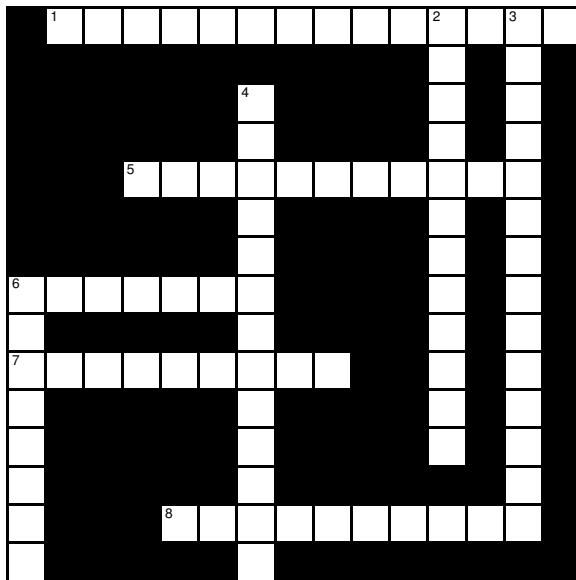
Kapitel 6 Objektorientierter Entwurf

- 6.1 Von der Analyse zum Entwurf
- 6.2 Konzepte des objektorientierten Entwurfs
- 6.3 Modellierung von Programmabläufen
- 6.4 Architekturentwurf
- 6.5 Entwurfsregeln und –heuristiken
- 6.6 Zusammenfassung**

Zusammenfassung Kapitel 6

- Das Konzept der Klasse wird im Entwurf um die Verwendung von **abstrakten**, **generischen** und **Container-Klassen** sowie **Schnittstellen** erweitert
- Für Attribute und Operationen wird die Notation um **Sichtbarkeit** erweitert
- Bei Operationen ist außerdem die vollständige **Signatur** anzugeben
- Die Notation von Assoziationen wird um die **Navigation** und die **Sichtbarkeit** erweitert
- Im Gegensatz zur Analyse tritt beim Entwurf häufig **Vererbung** auf
- **Polymorphismus** ermöglicht es, mittels Vererbung flexible Programme zu entwickeln
- **Sequenzdiagramme** und **Kollaborationsdiagramme** werden im Entwurf zur Beschreibung von Programmabläufen eingesetzt
- Ein System kann als **Zwei-Schichten**-, **Drei-Schichten**- oder **Mehr-Schichten-Architektur** entworfen werden
- Die Grundlage aller Schichtenarchitekturen ist das **Modell-View-Control**-Architekturkonzept

Frage: Kreuzworträtsel zu Kapitel 6



Waagrecht

1. Gleichlautende Nachrichten an kompatible Objekte unterschiedlicher Klassen können ein unterschiedliches Verhalten bewirken können
5. Spezifikation der grundlegenden technischen Struktur eines Systems
6. Mittel zur Strukturierung einer Software-Architektur
7. Eine spezielle Klasse, die als eine mit generischen formalen Parametern versehene Schablone spezifiziert ist, mit der gewöhnliche Klassen erzeugt werden können, nennt man ...
8. Mechanismus, bei dem ein Objekt eine Nachricht nicht (vollständig) selbst interpretiert, sondern an ein anderes Objekt weiterleitet

Senkrecht

2. Definiert die Zugriffsrechte auf Attribute und Operationen einer Klasse
3. Redefinition einer geerbten Operation durch die Unterklasse
4. Beschreibt einen ausgewählten Teil des extern sichtbaren Verhaltens von Modellelementen, d.h. eine Menge von Signaturen
6. Setzt sich zusammen aus dem Namen der Operation, ihrer Parameterliste und der Angabe eines evtl. Rückgabetyps

Kapitel 7 Entwurfsmuster und Frameworks

<u>7.1 Entwurfsmuster, Frameworks, Klassenbibliotheken</u>	<u>407</u>
<u>7.2 Fabrikmethode-Muster</u>	<u>416</u>
<u>7.3 Singleton-Muster</u>	<u>421</u>
<u>7.4 Beobachter-Muster</u>	<u>425</u>
<u>7.5 Anwendungsbeispiel</u>	<u>429</u>
<u>7.6 Zusammenfassung</u>	<u>436</u>

Kapitel 7 Entwurfsmuster und Frameworks

- 7.1 Entwurfsmuster, Frameworks, Klassenbibliotheken
- 7.2 Fabrikmethode-Muster
- 7.3 Singleton-Muster
- 7.4 Beobachter-Muster
- 7.5 Anwendungsbeispiel
- 7.6 Zusammenfassung

Lernziele

- Entwurfsmuster, Frameworks und Klassenbibliotheken unterscheiden können
- Wichtige Entwurfsmuster kennen und erklären können, wo sie eingesetzt werden
- Entwurfsmuster bei der Modellierung einsetzen können
- Entwurfsmuster in einem OOD-Modell erkennen können



Kapitel 7 Entwurfsmuster und Frameworks

7.1 Entwurfsmuster, Frameworks, Klassenbibliotheken

- 7.2 Fabrikmethode-Muster
- 7.3 Singleton-Muster
- 7.4 Beobachter-Muster
- 7.5 Anwendungsbeispiel
- 7.6 Zusammenfassung

Definition Entwurfsmuster (Design Pattern)

Ein **Entwurfsmuster** gibt eine bewährte generische Lösung für ein immer wiederkehrendes Entwurfsproblem, das in einem bestimmten Kontext auftritt

- Entwurfsmuster ermöglichen Wiederverwendung von erfolgreichen Entwürfen und Architekturen
Wiederverwendung von Design!
- Beschreibung eines Entwurfsmusters
 - Name des Musters
 - Muster-“Essenz“ in ein bis zwei Worten
 - Problembeschreibung
 - Wann ist das Muster anwendbar?
 - Lösungsbeschreibung
 - Abstrakte Lösung
 - Konsequenzen
 - Für Evaluierung von Entwurfsalternativen

Klassifikation von Mustern

- Erzeugungsmuster (creational patterns)
 - Helfen, ein System unabhängig davon zu machen, wie seine Objekte erzeugt, zusammengesetzt und repräsentiert werden
- Strukturmuster (structural patterns)
 - Befassen sich mit der Zusammensetzung von Klassen und Objekten zu größeren Strukturen
- Verhaltensmuster (behavioral pattern)
 - Befassen sich mit der Interaktion zwischen Objekten und Klassen
 - Beschreiben komplexe Kontrollflüsse, die zur Laufzeit schwer nachvollziehbar sind

Hinweis: Elementares Buch zum Thema Entwurfsmuster (Design Pattern)

Gamma, Erich et al.: *Design Patterns: Elements of Reusable Object-Oriented Software*

Addison-Wesley Verlag, Reading MA, 1996

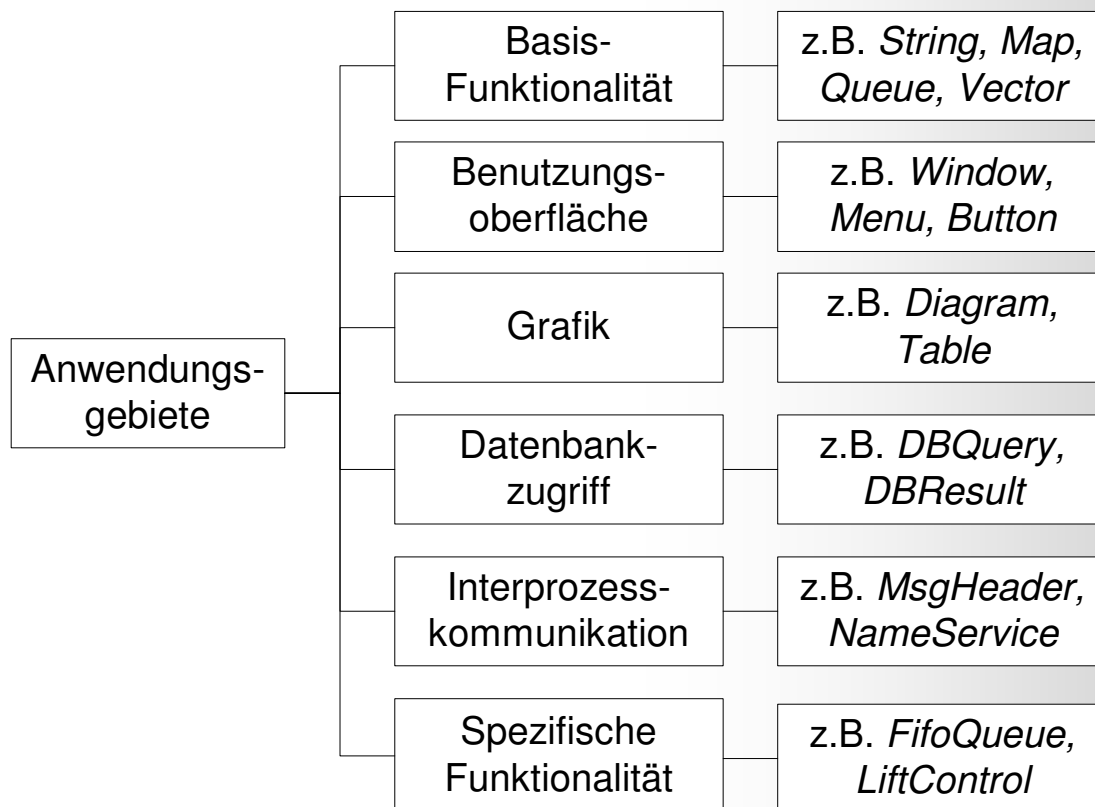
Definition Klassenbibliothek

Organisierte Sammlung von Klassen, aus der ein Entwickler nach Bedarf einzelne Klassen verwendet

- Mögliche Topologien von Klassenbibliotheken:
 - Baum-Topologie
 - Gemeinsame Wurzelklasse
 - Wald-Topologie
 - Bibliothek besteht aus mehreren Baumhierarchien
 - Vorteil: Flachere Vererbungshierarchie im Vergleich zur Baum-Topologie
 - Baustein-Topologie
 - Unabhängige Klassen
 - Verwendung des Konzepts der generischen Klasse zur spezifischen Anpassung

Wiederverwendung von Implementierungscode!

Klassenbibliotheken im Überblick



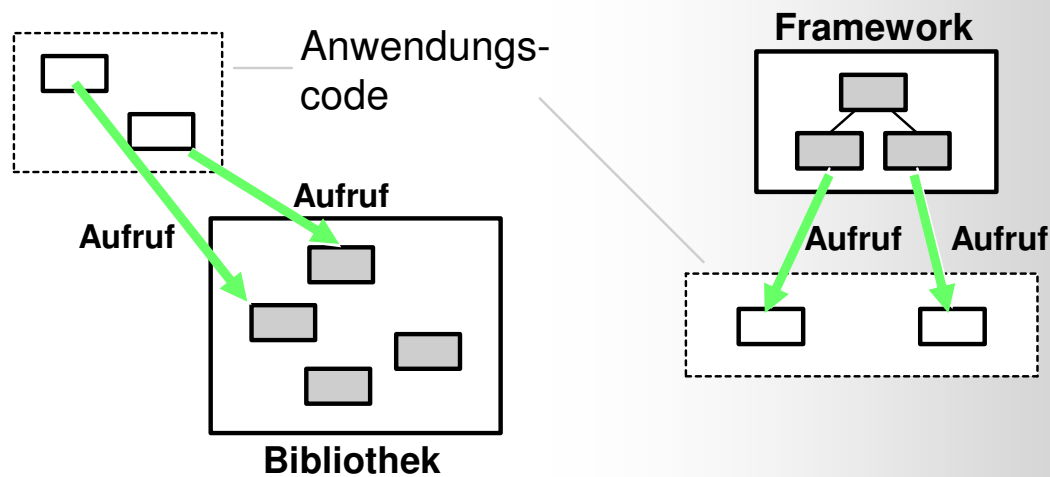
Definition Framework

Menge von kooperierenden Klassen, die einen wiederverwendbaren Entwurf für einen bestimmten Anwendungsbereich **implementieren**

- Wiederverwendung von Architektur + Code
- Besteht aus konkreten und abstrakten Klassen (Schnittstellen), die in einer Programmiersprache implementiert sind
 - Direkte Ausführung ist häufig möglich (Default-Anwendung)
- Frameworks sind immer auf spezifische Anwendungsbereiche ausgelegt.
Beispiele:
 - Erstellung grafischer Editoren
 - Erstellung von Steuerungen
- Anwendungsentwicklung mit Frameworks durch Parametrisierung und Ableitung von Schnittstellen
 - Anwendungsentwickler kann sich um spezifische Aspekte seiner Anwendung kümmern

Merkmale von Frameworks

- Frameworks ermöglichen einen besonders hohen Grad an Wiederverwendung
 - hohe Produktivitäts- und Qualitätssteigerung
 - Wettbewerbsvorteile
- Inversion des Kontrollflusses zur Laufzeit (sog. Hollywood-Prinzip: „Don't call me, I'll call you“):



Wiederverwendung von Design und Code!

Unterschiede zwischen Entwurfsmuster und Framework

- Entwurfsmuster sind abstrakter als Frameworks
 - Werden nur beispielhaft durch Programmcode repräsentiert
 - Anwendung von Entwurfsmustern ist mit einer neuen Implementierung verbunden
- Entwurfsmuster sind kleiner als Frameworks
 - Ein typisches Framework enthält mehrere Entwurfsmuster
- Entwurfsmuster sind weniger spezialisiert als Frameworks
 - Häufig keine Beschränkung auf einen bestimmten Anwendungsbereich

Entwurfsmuster und Frameworks sind eigenständige Wiederverwendungskonzepte mit wertvollen Synergieeffekten!

Frage zu 7.1

Nennen Sie die Gemeinsamkeiten und Unterschiede zwischen Mustern, Frameworks und Bibliotheken.

Antwort

Gemeinsamkeiten

- Unterstützung der Wiederverwendung
- Standardisierung von Software

Unterschiede

– Abstrakteste Form, kein Code	Muster
– Software-Architektur für bestimmte Anwendungsbereiche	Frameworks
– Relativ unabhängige Sammlung von Einheiten zur direkten Verwendung	Bibliotheken

- Frameworks basieren meistens auf Mustern

**Kapitel 7 Entwurfsmuster und Frameworks**

7.1 Entwurfsmuster, Frameworks, Klassenbibliotheken

7.2 Fabrikmethode-Muster

7.3 Singleton-Muster

7.4 Beobachter-Muster

7.5 Anwendungsbeispiel

7.6 Zusammenfassung

Fabrikmethode

– Motivation:

- Verwendung eines Frameworks für eine Anwendung, die mehrere Dokumente gleichzeitig anzeigen kann
- Verwendung der beiden abstrakten Klassen **Application** und **Document** und Modellierung einer Assoziation zwischen ihren Objekten
- Klasse **Application** ist für die Erzeugung neuer Dokumente zuständig
- Softwarekonstrukteur leitet von diesen beiden Klassen seine anwendungsspezifischen Klassen ab

– Problem:

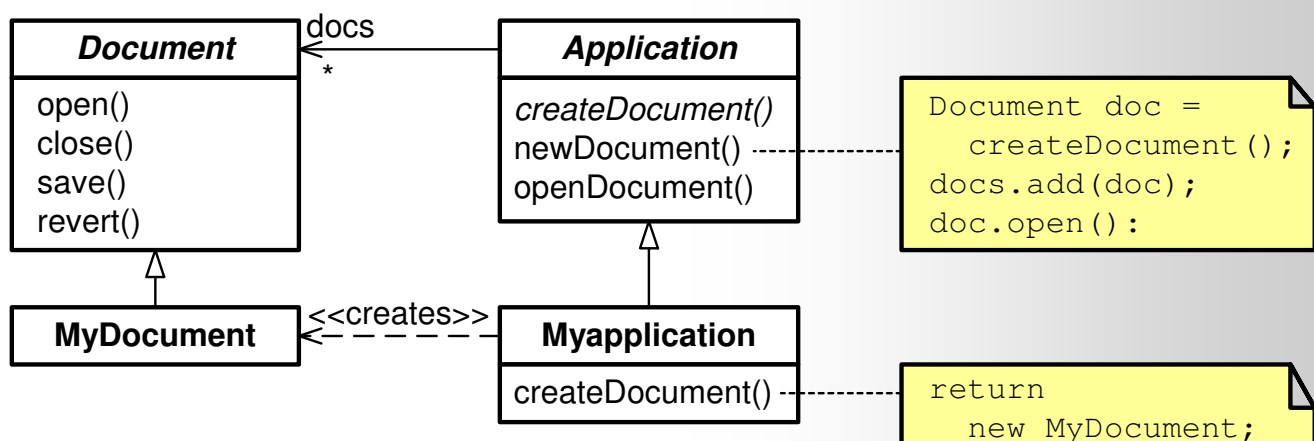
- Erzeugung eines neuen Objekts von **MyDocument** aus der Klasse **MyApplication**
- Framework muss Objekte erzeugen, kennt aber nur die abstrakte Oberklasse, von der es keine Objekte erzeugen darf

Fabrikmethode

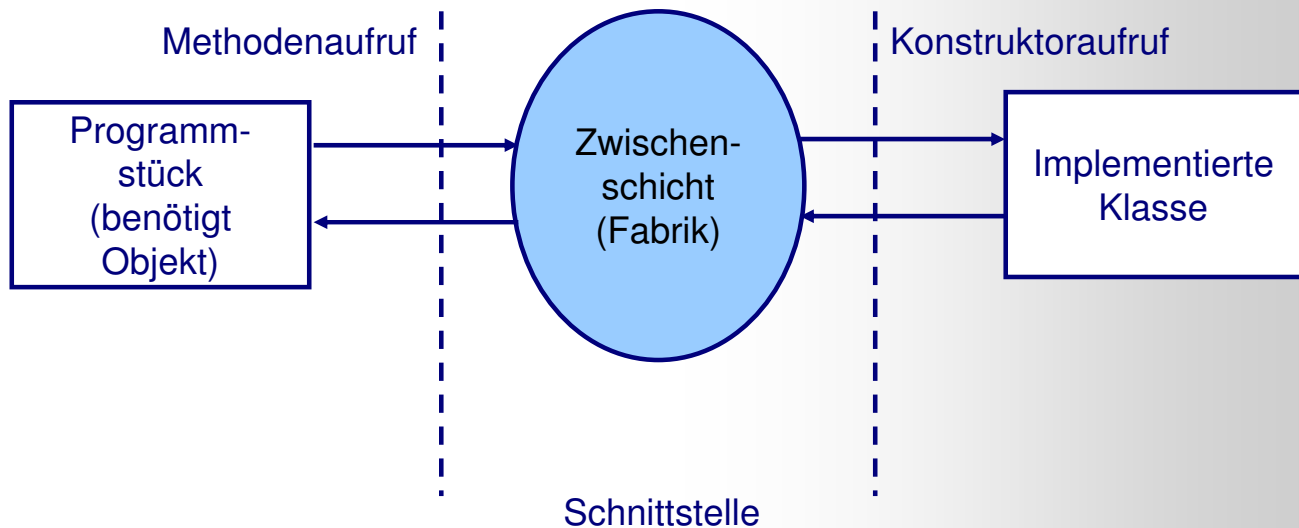
– Lösung:

- Unterklassen von **Application** überschreiben die abstrakte Operation **createDocument()**
 - Exemplar von **MyDocument** wird zurückgegeben
- Nach der Erzeugung eines Objekts von **MyApplication** kann diese spezifische Dokumente erzeugen, ohne deren exakte Klasse zu kennen

createDocument() heißt
Fabrikmethode



Fabrikmethode - das Prinzip (Tafelanschrieb)



Bisher:

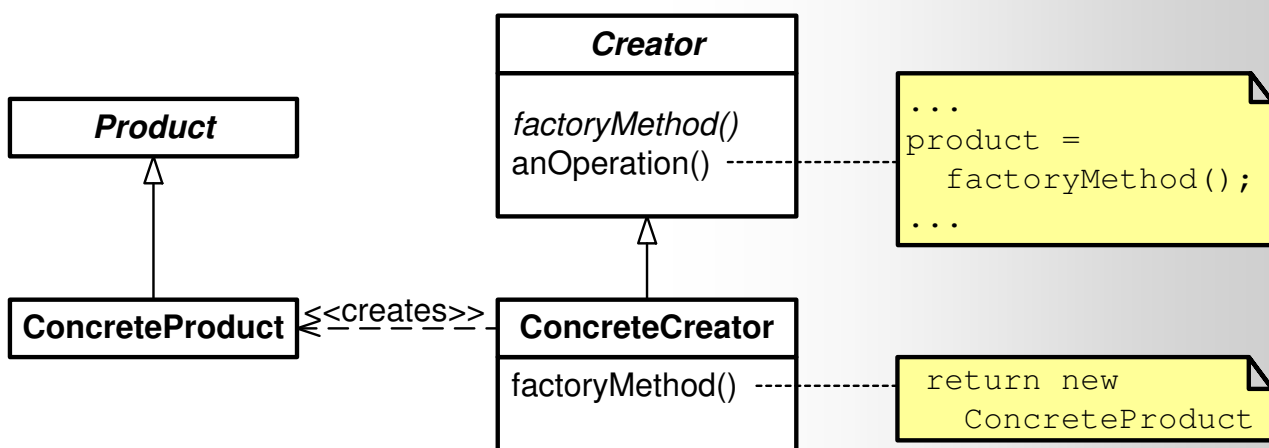
```
meinObjekt = new ImplementierteKlasse()
```

mit Fabrikmethode:

```
meinObjekt = Fabrik.Fabrikmethode()
```

Fabrikmethode

- Anwendbarkeit:
 - Eine Klasse kann oder darf die zu erzeugenden Objekte nicht im Voraus kennen
 - Unterklassen sollen festlegen, wie Objekte erzeugt werden
- Struktur (klassenbasiertes Erzeugungsmuster)

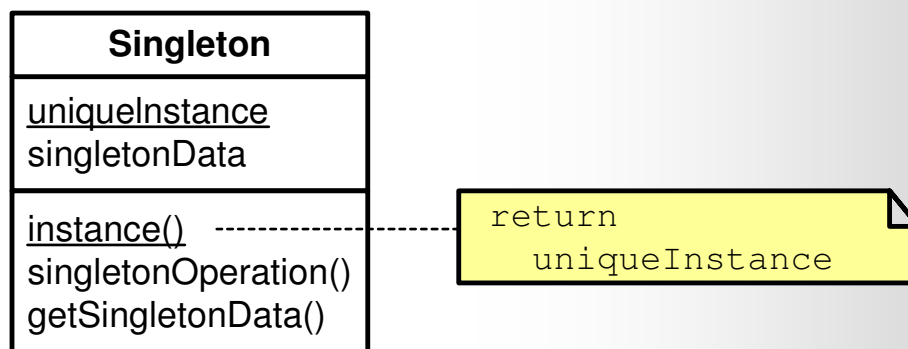


Kapitel 7 Entwurfsmuster und Frameworks

- 7.1 Entwurfsmuster, Frameworks, Klassenbibliotheken
- 7.2 Fabrikmethode-Muster
- 7.3 Singleton-Muster**
- 7.4 Beobachter-Muster
- 7.5 Anwendungsbeispiel
- 7.6 Zusammenfassung

Singleton

- Motivation:
 - Bei manchen Klassen soll genau ein Objekt existieren
Bsp.: Datenbank, Druckerspooler
 - Einfacher Zugriff auf dieses „Einzel“-Objekt durch andere Objekte
- Struktur (Objektbasiertes Erzeugungsmuster)



Singleton

– Implementierung:

- Erzeugung von Instanzen (mit `new()`) soll verhindert werden

```
class Singleton {
    private static Singleton uniqueInstance =
        new Singleton();
    private int singletonData;
    private Singleton() {}
    public static Singleton instance() {
        return uniqueInstance;
    }
    public int getValue() { return singletonData; }
    public void setValue(int x) {singletonData = x; }
}
```

Verhindert
Default-
Konstruktor !

– Verwendung

```
Singleton s = Singleton.instance();
System.out.println(s.getValue());
```

Singleton

– Anwendbarkeit:

- Es soll genau ein Objekt einer Klasse geben und ein einfacher Zugriff darauf bestehen
- Das einzige Exemplar wird durch Spezialisierung mittels Unterklassen erweitert und Klienten können das erweiterte Exemplar verwenden, ohne ihren Code zu ändern

– Konsequenzen:

- Singleton-Muster vs. statische Methoden
 - Flexiblere Lösung als statische Methoden
 - ⇒ Erweiterbarkeit auf n Instanzen
 - ⇒ Design kann Objekt verlangen
- Erweiterung auf mehrere Instanzen
 - kann leicht auf (höchstens) n Instanzen erweitert werden
 - `instance()` Methode kann parametrisiert werden, damit verschiedene Instanzen abgefragt werden können

Kapitel 7 Entwurfsmuster und Frameworks

7.1 Entwurfsmuster, Frameworks, Klassenbibliotheken

7.2 Fabrikmethode-Muster

7.3 Singleton-Muster

7.4 Beobachter-Muster

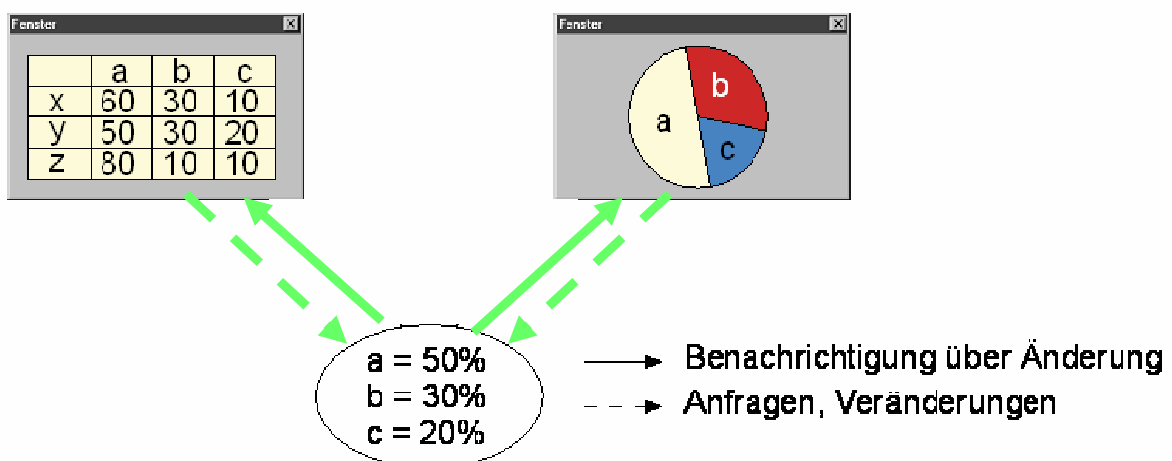
7.5 Anwendungsbeispiel

7.6 Zusammenfassung

Beobachter

– Motivation:

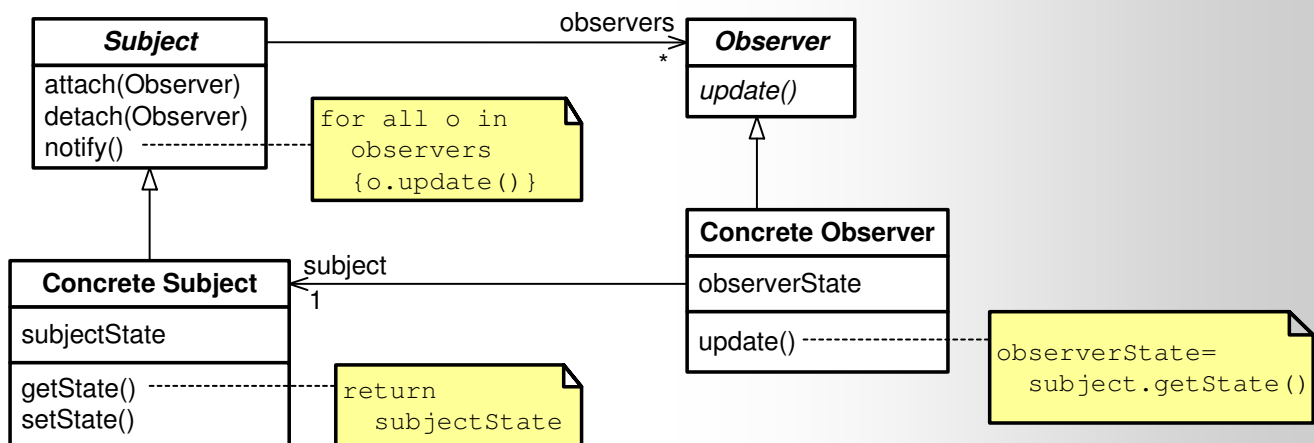
- Objekt enthält Anwendungsdaten
- Darstellung dieser Daten auf verschiedene Arten
- Kreisdigramme sollen sich ändern, wenn die Daten in der Tabelle verändert werden und umgekehrt



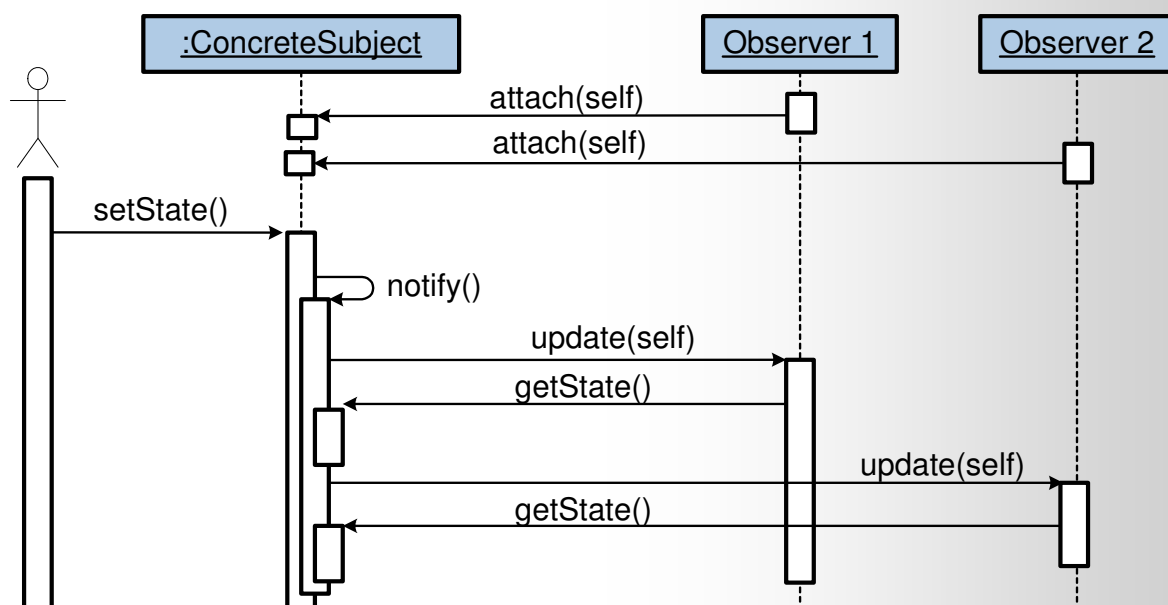
Beobachter

– Struktur

- Abstrakte Klasse **Subject** kennt alle ihre Beobachter (**Observer**) und informiert sie über alle Änderungen
- Abstrakte Klasse **Observer** definiert die Schnittstelle für alle konkreten observer, d.h. für alle Objekte, die über Änderungen eines subjects informiert werden müssen.
- Beobachter kennen sich untereinander nicht
- Synchronisation jedes Beobachters mit dem Zustand des **Subject**



Beobachter - Interaktionen des Musters



– Konsequenzen

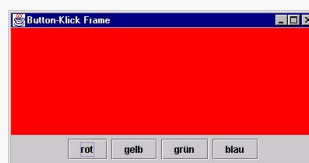
- Das Beobachter-Muster ermöglicht es, Subjekte und Beobachter unabhängig voneinander zu modifizieren
- Beobachter und Subjekte können einzeln wiederverwendet werden
- Neue Beobachter sind ohne Änderung des Subjekts hinzufügbare

Kapitel 7 Entwurfsmuster und Frameworks

- 7.1 Entwurfsmuster, Frameworks, Klassenbibliotheken
- 7.2 Fabrikmethode-Muster
- 7.3 Singleton-Muster
- 7.4 Beobachter-Muster
- 7.5 Anwendungsbeispiel**
- 7.6 Zusammenfassung

Hintergrund: Oberflächen-Programmierung in Java

- Oberflächen bestehen aus verschiedenen Anzeige-Objekten
 - Fenster
 - Buttons
 - Scrollbars
 - Menüs
 - Listen
 - Auswahlboxen
- Java stellt für die Oberflächen-Programmierung die **Java Foundation Classes (JFC)** zur Verfügung, die aus mehreren Paketen bestehen



Java Swing Bibliothek

- API zur Erstellung leistungsfähiger und komplexer grafischer Benutzeroberflächen (GUI)
- Alle Swing-Klassen beginnen mit dem Großbuchstaben J
- Alle Komponenten sind in Java geschrieben (lightweight components)

Ereignisbehandlung mit SWING

- Damit in den Programmen auch auf Aktionen des Benutzers reagiert werden kann, existieren „Abhörerklassen“
- Benutzeraktivitäten lösen Ereignisse aus, die vom GUI-System an das jeweilige Programm weitergegeben werden
 - Ereignisse werden durch Anzeige-Objekte erzeugt und weitergeleitet
- Ereignisquellen
 - Beispiel: Button, der angeklickt wird, Tastatureingabe, Mausbewegung
- GUI-Programme warten auf Ereignisse (events)

→ Anwendung des Beobachtermusters

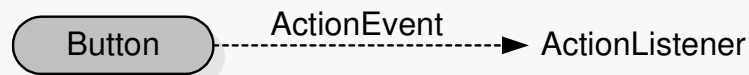
- Vorteile
 - Trennung von Oberfläche und Ereignisbehandlung (Listener-Klassen für bestimmte Arten von Ereignissen)
 - bessere Objektorientierung (Ereigniscode für eine spezielle Klasse von Ereignissen ist vererbbar und an jede Oberfläche "montierbar")
 - schnell, flexibel, übersichtlich

Prinzip der Ereignisbehandlung

- Anzeige-Objekte sind Ereignisquellen
 - Beispiel: Fenster Button, Tastatur, Scrollbar
 - Control-Objekte sind Beobachter / Abhörer (Event-Listener)
 - Benötigen Informationen über Benutzeraktionen
 - Event-Listener Schnittstellen
 - Event-Listener reagieren auf Ereignisse an den Quellen (z.B. ActionListener für Button-Ereignisse)
 - Schnittstelle definiert Operationen, die von Anzeige-Objekten aufgerufen werden, wenn ein bestimmtes Ereignis eintritt
 - Event-Listener müssen für jedes Ereignis, auf das reagiert werden soll, implementiert werden
-
- Control-Objekte müssen sich dann bei Ereignisquelle anmelden, damit diese die geforderten Ereignisse weitermeldet
 - Objekte können sich bei mehreren Ereignisquellen anmelden
 - Bei einer Ereignisquelle mehrere Objekte als „Abhörer“ möglich

Erstellung einer Abhörerklasse

– Prinzip:



– Implementieren eines Event Handlers:

1. Die eigene Event Handler Klasse muss die Listener Schnittstelle implementieren:

```
class MeinEventHandler implements ActionListener {
```

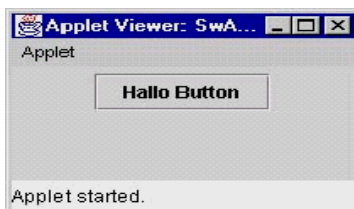
2. Die Event Handler Klasse muss sich als Listener bei einer oder mehreren Komponenten registrieren:

```
Button.addActionListener(ObjektVonMeinEventHandler);
```

3. In der Event Handler Klasse werden die Methoden der Listener Schnittstelle implementiert:

```
public void actionPerformed(ActionEvent e) {
    // Reagiere auf das aufgetretene Ereignis
}
```

Beispiel



Animation

JButton

button

Der Mausklick veranlasst den angeklickten **JButton**, ein Objekt **event** vom Typ **ActionEvent** zu erzeugen. Dieses wird an alle bei dem Button registrierten **ActionListener** Objekte geschickt.

ActionEvent event

Hat alle Informationen bzgl. des Ereignisses und seiner Quelle, sowie eine Beschreibung der Ausgeführten Aktion.

MyListenerClass

myActionListener

implementiert die Event Listener Methode **actionPerformed()** in der festgelegt ist, was beim Eintreten des Events passieren soll. Sie wird von dem Button bei dem das Objekt als **ActionListener** registriert ist aufgerufen.

Event Listener Schnittstellen

– Schnittstellen verschiedener Ereignis-Abhörer

- `ActionListener` zeigt Aktionen auf Button, List, MenuItem, TextField an
- `MouseListener` Abbören von möglichen Maus-Klicks
- `MouseMotionListener` hört auf Mausbewegungen
- `KeyListener` hört die Tastatur ab
- `FocusListener` hört ab, ob eine Komponente den Eingabefokus erhält
- `AdjustmentListener` achtet auf Verschieben der Scrollbar
- `ItemListener` beobachtet die Zustände von Auswahlboxen
- `TextListener` beobachtet Veränderungen in Textfeldern



Kapitel 7 Entwurfsmuster und Frameworks

- 7.1 Entwurfsmuster, Frameworks, Klassenbibliotheken
- 7.2 Fabrikmethode-Muster
- 7.3 Singleton-Muster
- 7.4 Beobachter-Muster
- 7.5 Anwendungsbeispiel

7.6 Zusammenfassung

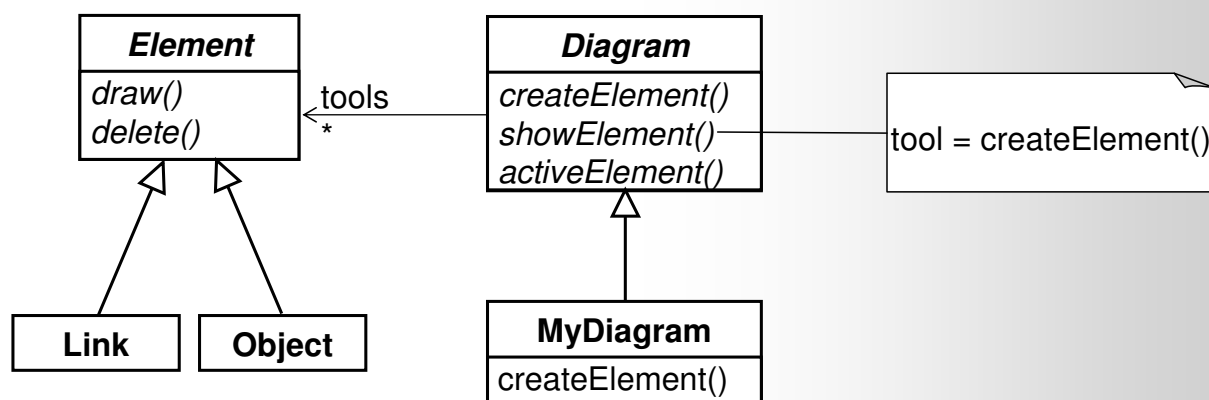
Zusammenfassung Kapitel 7

- **Klassenbibliotheken** dienen der Wiederverwendung von Implementierungscode
- **Entwurfsmuster** beschreiben Lösungen für immer wiederkehrende Entwurfsprobleme
- **Fabrikmethode-Muster** (klassenbasiertes Erzeugungsmuster)
 - Bietet eine Schnittstelle zum Erzeugen eines Objekts an
 - Unterklassen entscheiden, von welcher Klasse das erzeugte Objekt ist
- **Singleton-Muster** (objektbasiertes Erzeugungsmuster)
 - Stellt sicher, dass eine Klasse genau ein Objekt besitzt
 - Ermöglicht einen globalen Zugriff auf dieses Objekt
- **Beobachter-Muster** (objektbasiertes Verhaltensmuster)
 - Bei Änderung eines Objekts werden alle davon abhängigen Objekte benachrichtigt und automatisch aktualisiert
- **Frameworks** stellen Klassen bereit, die als Basisklassen für neu zu erstellende Anwendungen verwendet werden können



Frage zu Kapitel 7

Geben Sie an, welches Muster im folgenden Klassendiagramm beschrieben ist.



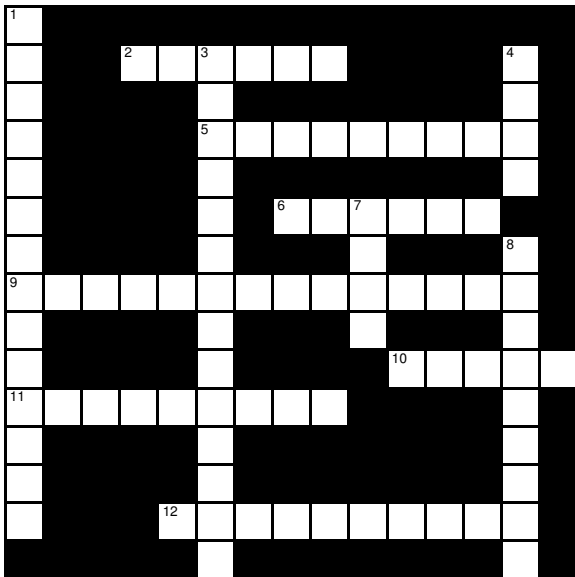
Antwort

Fabrikmethode (*factory method*)

In der Klasse **MyDiagram** wird durch die Fabrikmethode **createElement()** konkret festgelegt, welche Objekte sie erzeugen soll



Frage: Kreuzworträtsel zu Kapitel 7



Waagrecht

2. Entwurfsmuster ermöglichen die Wiederverwendung von ...
5. Name des Prinzips zur Inversion des Kontrollflusses
6. Entwurfsmuster zur Erzeugung von Objekten unbekannter Klassen
9. Muster, die sich mit der Zusammensetzung von Klassen und Objekten zu größeren Strukturen befassen
10. Wichtigster Erfinder von Entwurfsmustern
11. Muster zur Kontrolle der Erzeugung von Objekten einer Klasse
12. Muster zur flexiblen Weiterleitung von Ereignissen an unbekannte Klassen

Senkrecht

1. Generalisierte Lösungsideen zu immer wiederkehrenden Entwurfsproblemen
3. Anwendungsentwicklung mit Frameworks erfolgt durch Parametrisierung und Ableitung von ...
4. Klassenbibliotheken ermöglichen die Wiederverwendung von ...
7. Mögliche Topologie von Klassenbibliotheken
8. Menge kooperierender Klassen, die unter Vorgabe eines Ablaufes eine generische Lösung für eine Reihe ähnlicher Aufgabenstellungen bereitstellen.

