

Software Engineering

Teil 2: Einführung in die UML und OCL

April 2018

Dr. Christian Bartelt



UNIVERSITATEA
BABEŞ-BOLYAI

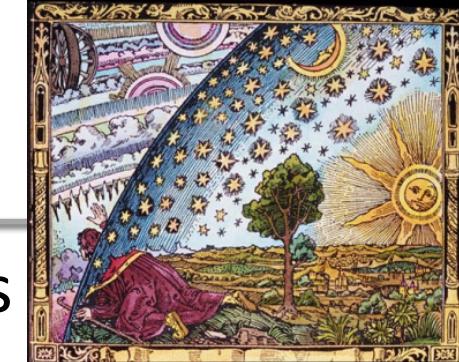
Universität Mannheim
Institut für Enterprise Systems InES
Schloss
68131 Mannheim / Germany

UNIVERSITY OF
MANNHEIM

Inhalt

- **Beschreibungstechniken und Basiswissen**
 - **Grundlagen der Objektorientierung (Wiederholung)**
 - Übersicht zur UML
- **Strukturdiagramme**
 - Klassendiagramm
 - Paketdiagramm
 - Objektdiagramm
 - Kompositionssstrukturdiagramm
 - Komponentendiagramm
 - Verteilungsdiagramm
- **Verhaltensdiagramme**
 - Use-Case-Diagramm
 - Aktivitätsdiagramm
 - Zustandsautomat
- **Interaktionsdiagramme**
 - Sequenzdiagramm
 - Kommunikationsdiagramm
 - Timingdiagramm
 - Interaktionsübersichtsdiagramm
- **OCL (Object Constraint Language)**

Die (Programmier)welt vor der Objektorientierung...



- Die Objektorientierung ist ursprünglich als Programmierparadigma entstanden.
- Ein **Programmierparadigma** beschreibt die Prinzipien, die einer Programmiersprache zugrunde liegen.
- Frühes Paradigma: die prozedurale Programmierung
 - Prinzipielle Idee: ein Computerprogramm besteht aus Daten (Datenstrukturen), die Elemente/Konzepte aus der Realität darstellen, und Prozeduren/Funktionen, die diese Daten verändern.
 - Daten und Funktionen also recht strikt getrennt

Was sind Objekte..?

- ...sind Gegenstände von Interesse
 - einer Beobachtung
 - einer Untersuchung
 - einer Messung
- ...sind Exemplare, etwas individuelles
 - Exemplare von Dingen (Auto, Roboter)
 - Personen (Kind, Kunde, Mitarbeiter)
 - Begriffe der realen Welt (Bestellung, Reservierung) oder der Vorstellungswelt (juristische Person)
- ...was macht Objekte aus
 - Objekte können physisch oder konzeptionell sein, sie haben eine **Identität**
 - Objekte verfügen über Eigenschaften (**Attribute**) wie Größe, Name, Form usw.
 - Objekte weisen Vorgänge (**Methoden**) auf (Dinge, die sie tun können).
Beispiele: Wert festlegen, Bildschirm anzeigen, Geschwindigkeit erhöhen

Beispiel: von Katzen und Mäusen...

- Beispiel: Programm, das Verhalten von Haustieren wie Katzen simuliert



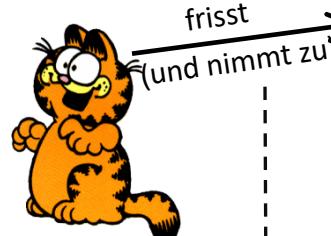
katze1 : {name="Minka",

fell="Schwarz-weiß", gewicht=4.0kg}

katze2 : {name="Garfield",
fell="Orange-Schwarz-Getigert",
gewicht=35.5kg}



katze3 : {name="Tom", fell="Grau-
weiß", gewicht=4.5kg}



maus1 : {gewicht=0.03kg, größe=9cm}

Daten



maus2 : {gewicht=0.28kg,
größe=8cm}

Funktionen

fressen(katze2, maus1) -> Katze2.gewicht = 35.53kg

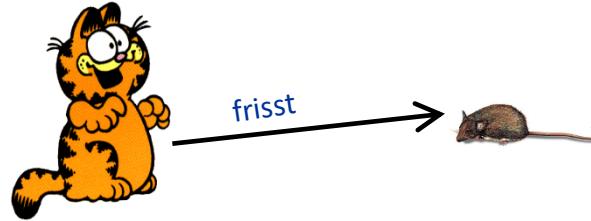
Prozedurale Katzen und Mäuse

- Daten(strukturen) und Funktionen müssen definiert werden:

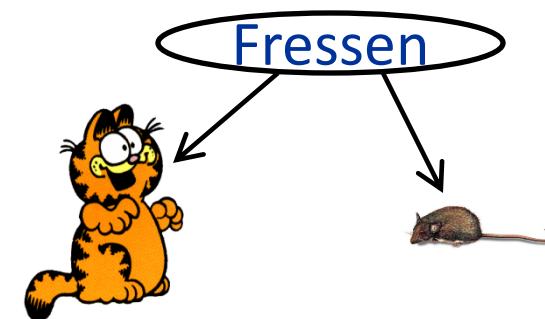
```
Katze {  
    name      : String;  
    fell      : String;  
    gewicht   : real;  
}  
  
Maus {  
    gewicht : real;  
    größe   : int;  
}  
  
function fressen (Katze betrachteteKatze,  
                    Maus gefresseneMaus) {  
    betrachteteKatze.gewicht = betrachteteKatze.gewicht +  
    sehrKomplizierteFormel(gefresseneMaus.gewicht);  
}
```

Probleme mit dieser Vorstellung

- **Problem 1:** Diese Vorstellung ist nicht sehr realistisch.
Funktion `fressen(...)` erweckt den Anschein, das Fressen würde von außen initiiert. Aber ist es nicht so, dass jede Katze *von sich aus* die Eigenschaft hat, fressen zu können?
- Oder umgangssprachlich: was ist realistischer?
 - a) „Garfield frisst die Maus“ oder
 - b) „Es gibt einen Vorgang ‚Fressen‘, der zwischen Garfield und dieser (bestimmten) Maus stattfindet.“
- Oder grafisch:



versus



Probleme mit dieser Vorstellung

- **Problem 2:** Wird ein solches System gut erweiterbar sein? Was ist mit anderen Haustieren wie Hunden? Mit anderen „Nahrungsmitteln“?

```
function fressen (Katze betrachteteKatze,  
                    Maus gefresseneMaus) ...
```

```
function fressen (Hund betrachteterHund,  
                    Hundefutter ...)
```

```
function fressen (Katze betrachteteKatze,  
                    Katzenfutter ...)
```

•

•



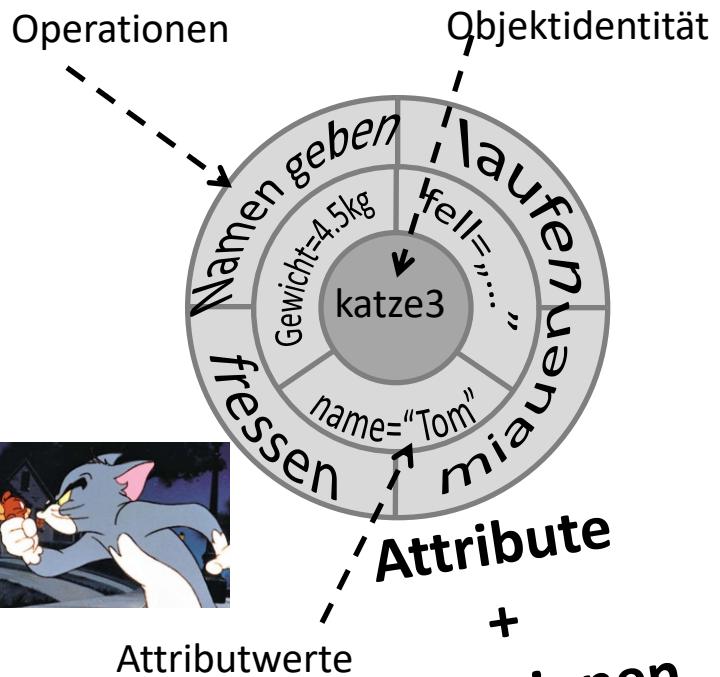
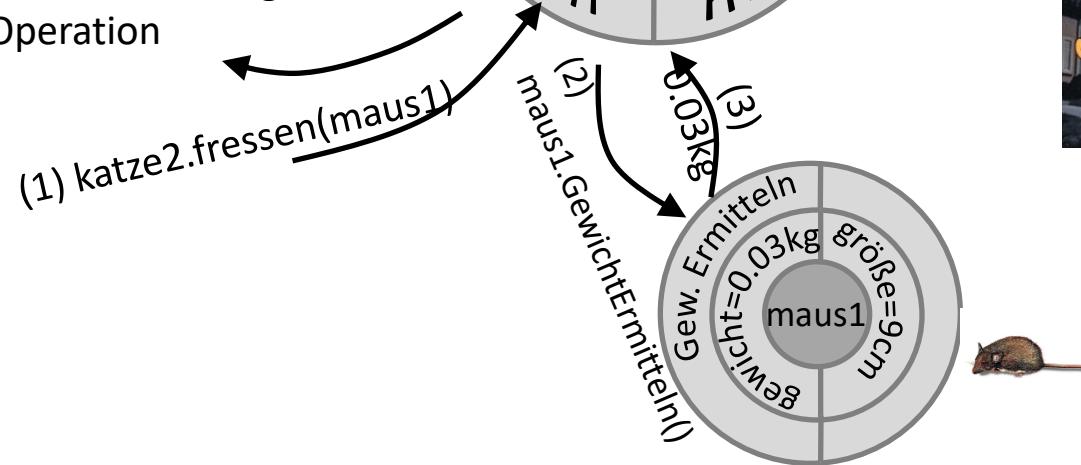
Das Paradigma der Objektorientierung

- Strikte Trennung zwischen Daten und Funktionen wird hierbei aufgehoben – ein Computerprogramm besteht aus sog. **Objekten**.
- Objekte haben einen Zustand, der durch **Attributwerte** beschrieben wird
- Objekte haben ein definiertes Verhalten, das durch **Operationen / Methoden** beschrieben wird. Diese können den Zustand , also Attributwerte verändern .
- Des Weiteren besitzt jedes Objekt eine **Objektidentität**, die es eindeutig von anderen Objekten unterscheidet.
- Objekte können miteinander kommunizieren, indem sie Operationen auf anderen Objekten aufrufen.

Objekte – Katzen objektorientiert



(4) Effekt: gewicht=35.53kg
nach Ausführung der
Operation



Klassen von Objekten, oder: was ist eine Katze?

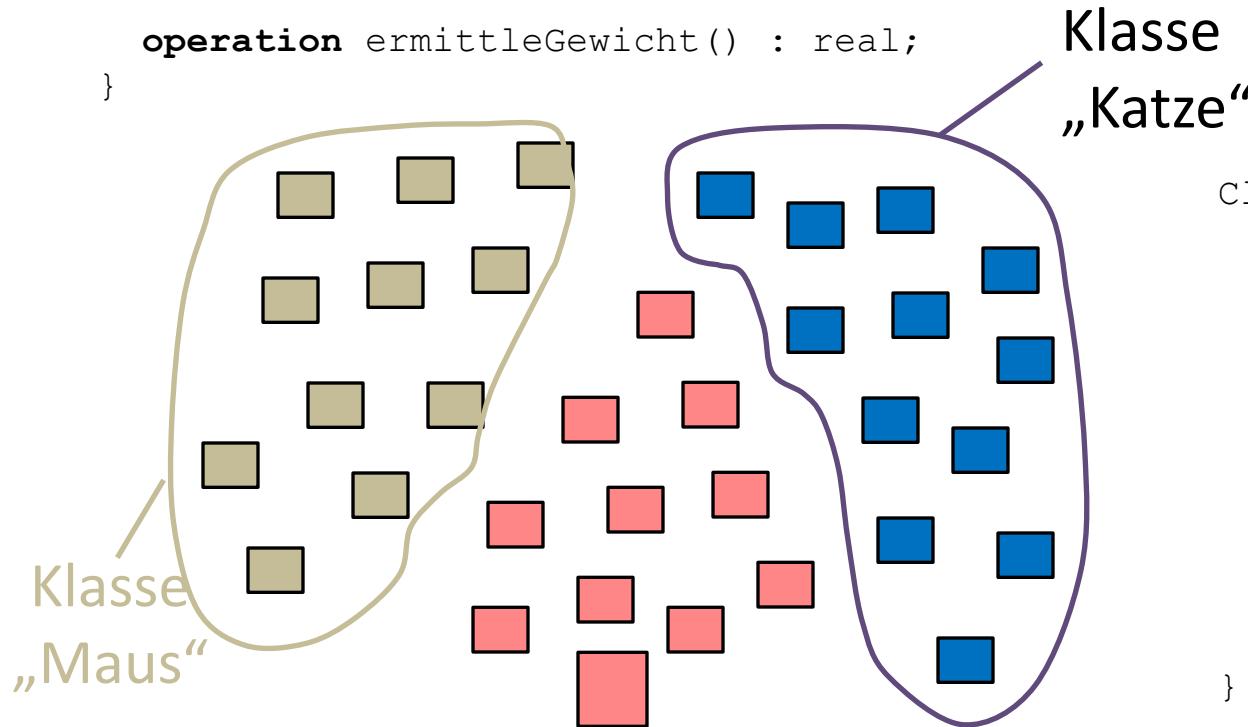
- Im prozeduralen Paradigma haben wir durch eine Datenstruktur definiert, welche Eigenschaften eine Katze im Allgemeinen besitzt. Diese definiert den **Typ** „Katze“. Das Füllen der Datenstruktur mit konkreten Werten beschreibt eine bestimmte Katze.
- Im OO-Paradigma werden Typen durch **Klassen** definiert. Klassen beschreiben Mengen von ähnlichen Objekten. Ähnlich heißt: Die Objekte besitzen die gleichen Attribute, aber unterschiedliche Attributwerte, und die gleichen Operationen.
- Objekte, die zu einer Klasse gehören, werden auch **Instanzen** dieser Klasse genannt.
- Beispiel für eine Klasse: die Menge aller Katzen

Klassen als Mittel der Abstraktion

- Klassen können als **Abstraktion** von Objekten verstanden werden, d.h. es werden Details weg gelassen, Gemeinsamkeiten werden verallgemeinert.
- Bei Klassen bedeutet dies z.B.: die konkreten **Attributwerte** werden ignoriert und zu einem Attribut abstrahiert
- Bsp.:
 - Attributwerte der Objekte: „Minka“, „Garfield“, „Tom“
 - Attribut der Klasse „Katze“: „name“ vom Typ String

Klassen visualisiert

```
Class Maus {  
    gewicht : real;  
    größe   : int;  
  
    operation ermittleGewicht() : real;  
}
```



```
Class Katze {  
    name      : String;  
    fell     : String;  
    gewicht  : real;  
  
    operation fressen(Maus m);  
    operation miauen();  
    operation laufen();  
    .  
    .  
    .  
}
```

Erweiterung des Beispiels

- Was tun, wenn man Hunde betrachten will?
- Eine Möglichkeit ist, eine neue Klasse zu definieren:

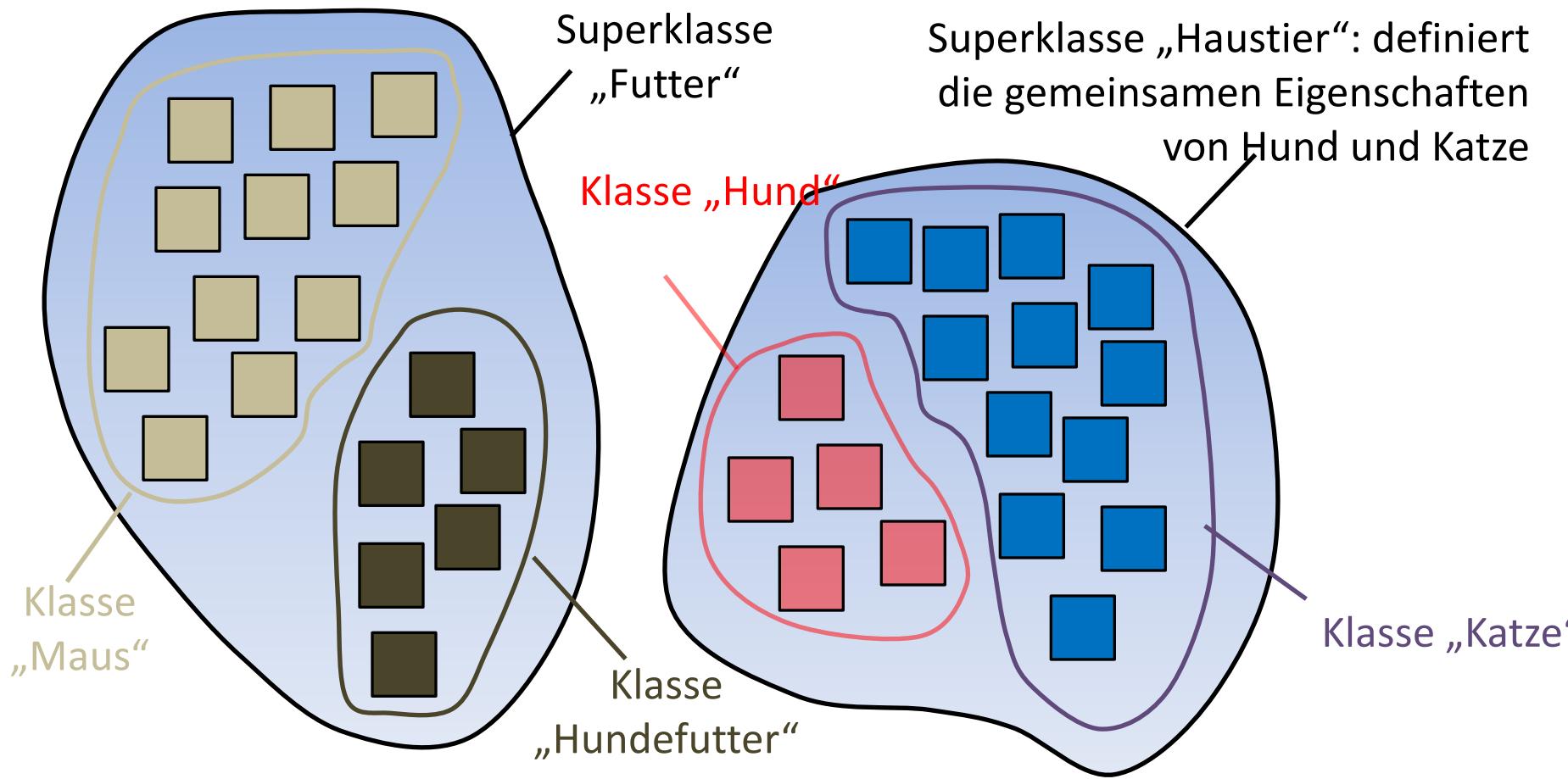
```
Class Hund {  
    name      : String;  
    fell      : String;  
    gewicht   : real;  
  
    operation fressen(Hundefutter f);  
    operation bellen();  
    operation laufen();  
  
    .  
    .  
    .  
}
```

Was fällt auf?

- Viele Gemeinsamkeiten mit Klasse „Katze“
- Dennoch Unterschiede (z.B. bei den verfügbaren Operationen)

- } Separate Klasse aus den gleichen Gründen unpraktisch wie bei zusätzlichen Datenstrukturen im prozeduralen Paradigma!

Abbildung von Gemeinsamkeiten - Superklassen



Superklassen

```
Class Haustier {  
    name      : String;  
    fell      : String;  
    gewicht   : real;  
  
    operation fressen(Futter f);  
    operation laufen();  
}
```

Subklasse

```
Class Katze extends Haustier {  
  
    operation miauen();  
}
```

Subklasse

```
Class Hund extends Haustier {  
    hundemarkenNummer : int;  
  
    operation bellen();  
}
```

Die Subklassen enthalten nur noch die
speziellen Eigenschaften, nicht die
gemeinsamen Eigenschaften!

Literatur

- UML Glasklar
 - Mario Jeckle, Chris Rupp, Jürgen Hahn, Barbara Zengler, Stefan Queins
- UML 2: Das umfassende Handbuch (Galileo Computing)
 - Christoph Kecker
- UML Distilled: A Brief Guide to the Standard Object Modeling Language
 - Martin Fowler, Kendall Scott
- Object Management Group
 - www.omg.org

Was ist die UML?

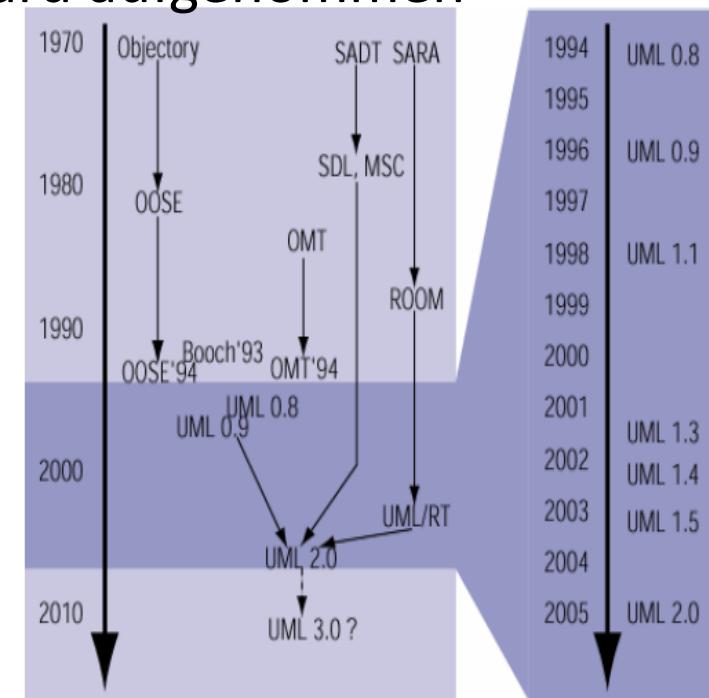
- UML ist eine Sprache – um sie sinnvoll anwenden zu können, muss man auch ihre Konzepte verstehen und was man mit ihr beschreiben kann.
- Gegenstand der UML sind (in den meisten Fällen) objektorientierte Systeme und ihre Entwicklung (Analyse, Design, Implementierung, etc.). Viele ihrer Konzepte entstammen der Objektorientierung.

Was ist die UML – und was nicht

- UML steht für „Unified Modeling Language“ – aktuelle Version ist 2.4.1 Spezifikation abrufbar unter www.omg.org.
- UML wird von Object Management Group (OMG) gepflegt und weiter entwickelt.
- Gestartet als Versuch viele existierende Modellierungssprachen der Softwaretechnik unter einen Hut zu bringen.
- Ist eine standardisierte, graphische Sprache um (objektorientierte) Systeme zu spezifizieren, zu konstruieren und zu dokumentieren. Daher sehr umfangreich (Spezifikation >1000 Seiten).
- Was sie **nicht** ist: eine Methode oder ein Vorgehen, die beschreiben, wie man Software erstellen sollte!

Historisches zur UML

- Zunächst begonnen als „Unified Method“ (1994) von Grady Booch und Jim Rumbaugh. Später dazu gestoßen Ivar Jacobson („The Three Amigos“)
- Später von der OMG als Standard aufgenommen
 - > Version 1.0 (1997)
 - > Version 2.0 (2005)
 - > Version 3.0 ?



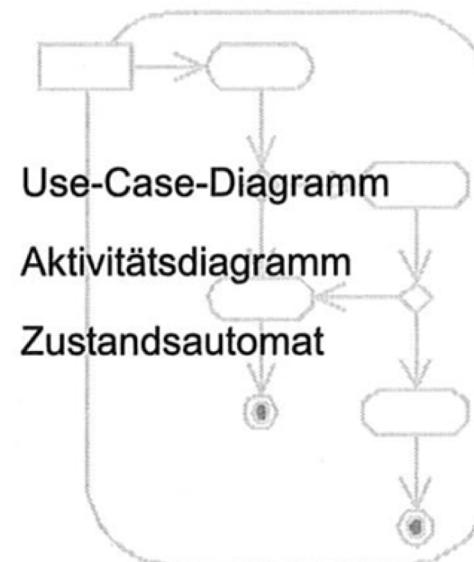
Die Diagrammarten der UML

Diagramme der UML 2

Strukturdiagramme



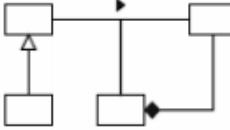
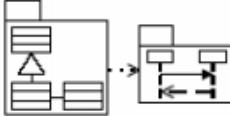
Verhaltensdiagramme



Interaktionsdiagramme



UML – Angehängt

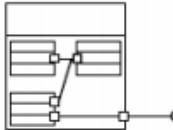
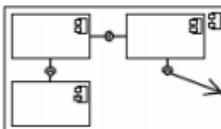
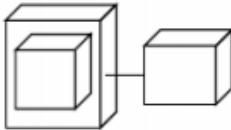
Diagrammtyp	Diese zentrale Frage beantwortet das Diagramm	Stärken
Klassendiagramm 	Aus welchen Klassen besteht mein System und wie stehen diese untereinander in Beziehung?	Beschreibt die statische Struktur des Systems. Enthält alle relevanten Strukturzusammenhänge/Datentypen. Brücke zu dynamischen Diagrammen. Normalerweise unverzichtbar.
Paketdiagramm 	Wie kann ich mein Modell so schneiden, dass ich den Überblick bewahre?	Logische Zusammenfassung von Modellelementen. Modellierung von Abhängigkeiten/ Inklusion möglich.
Objektdiagramm 	Welche innere Struktur besitzt mein System zu einem bestimmten Zeitpunkt zur Laufzeit (Klassendiagrammschnappschuss)?	Zeigt Objekte u. Attributbelegungen zu einem bestimmten Zeitpunkt. Verwendung beispielhaft zur Veranschaulichung Detailniveau wie im Klassendiagramm. Sehr gute Darstellung von Mengenverhältnissen.

Übersicht zur UML

Beschreibungstechniken und Basiswissen

aus: UML 2 glasklar

UML – Angehängt

Diagrammtyp	Diese zentrale Frage beantwortet das Diagramm	Stärken
Kompositionssstruktur-diagramm 	Wie sieht das Innenleben einer Klasse, einer Komponente, eines Systemteils aus?	Ideal für die Top-Down-Modellierung des Systems (Ganz-Teil-Hierarchien). Zeigt Teile eines „Gesamtelements“ und deren Mengenverhältnisse. Präzise Modellierung der Teile-Beziehungen über spezielle Schnittstellen (Ports) möglich.
Komponentendiagramm 	Wie werden meine Klassen zu wieder verwendbaren, verwaltbaren Komponenten zusammengefasst und wie stehen diese in Beziehung?	Zeigt Organisation und Abhängigkeiten einzelner technischer Systemkomponenten. Modellierung angebotener und benötigter Schnittstellen möglich.
Verteilungsdiagramm 	Wie sieht das Einsatzumfeld (Hardware, Server, Datenbanken, ...) des Systems aus? Wie werden die Komponenten zur Laufzeit wohin verteilt?	Zeigt das Laufzeitumfeld des Systems mit den „greifbaren“ Systemteilen. Darstellung von „Softwareservern“ möglich. Hohes Abstraktionsniveau, kaum Notationselemente.

UML – Angehängt

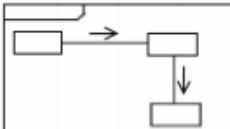
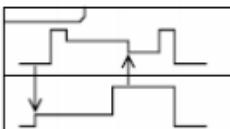
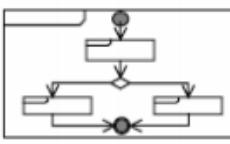
Diagrammtyp	Diese zentrale Frage beantwortet das Diagramm	Stärken
Use-Case-Diagramm 	Was leistet mein System für seine Umwelt (Nachbarsysteme, Stakeholder)?	Außensicht auf das System. Geeignet zur Kontextabgrenzung. Hohes Abstraktionsniveau, einfache Notationsmittel.
Aktivitätsdiagramm 	Wie läuft ein bestimmter flussorientierter Prozess oder ein Algorithmus ab?	Sehr detaillierte Visualisierung von Abläufen mit Bedingungen, Schleifen, Verzweigungen. Parallelisierung und Synchronisation. Darstellung von Datenflüssen.
Zustandsautomat 	Welche Zustände kann ein Objekt, eine Schnittstelle, ein Use Case, ... bei welchen Ereignissen annehmen?	Präzise Abbildung eines Zustandsmodells mit Zuständen, Ereignissen, Nebenläufigkeiten, Bedingungen, Ein- und Austrittsaktionen. Schachtelung möglich.
	Wer tauscht mit wem welche Informationen in welcher Reihenfolge aus?	Darstellung d. Informationsaustauschs zwischen Kommunikationspartnern Sehr präzise Darstellung der zeitlichen Abfolge auch mit Nebenläufigkeiten.

Übersicht zur UML

Beschreibungstechniken und Basiswissen

aus: UML 2 glasklar

UML – Angehängt

Diagrammtyp	Diese zentrale Frage beantwortet das Diagramm	Stärken
Kommunikationsdiagramm 	Wer kommuniziert mit wem? Wer „arbeitet“ im System zusammen?	Stellt den Informationsaustausch zwischen Kommunikationspartnern dar. Überblick steht im Vordergrund (Details und zeitliche Abfolge weniger wichtig).
Timingdiagramm 	Wann befinden sich verschiedene Interaktionspartner WS 2013/2014 in welchem Zustand?	Visualisiert das exakte zeitliche Verhalten von Klassen, Schnittstellen,.. Geeignet für die Detailbetrachtungen, bei denen es wichtig ist, dass ein Ereignis zum richtigen Zeitpunkt eintritt.
Interaktionsübersichtsdiagramm 	Wann läuft welche Interaktion ab?	Verbindet Interaktionsdiagramme (Sequenz-, Kommunikation- und Timingdiagramme) auf Top-Level-Ebene. Hohes Abstraktionsniveau.

Inhalt

- **Beschreibungstechniken und Basiswissen**
 - Grundlagen der Objektorientierung (Wiederholung)
 - Übersicht zur UML
- **Strukturdiagramme**
 - **Klassendiagramm**
 - Paketdiagramm
 - Objektdiagramm
 - Kompositionssstrukturdiagramm
 - Komponentendiagramm
 - Verteilungsdiagramm
- **Verhaltensdiagramme**
 - Use-Case-Diagramm
 - Aktivitätsdiagramm
 - Zustandsautomat
- **Interaktionsdiagramme**
 - Sequenzdiagramm
 - Kommunikationsdiagramm
 - Timingdiagramm
 - Interaktionsübersichtsdiagramm
- **OCL (Object Constraint Language)**

Klassendiagramm – was ist das?

- Zentrales Konzepte der OOA/D sind Objekte bzw. Klassen.
- In UML werden Klassen durch Klassendiagramme modelliert.
- Fokus von Klassendiagrammen: statische Struktur
 - Attribute und Operationen von Klassen
 - Beziehungen zwischen Klassen
- Wohl die am häufigsten verwendete Diagrammform der UML

Klassendiagramm

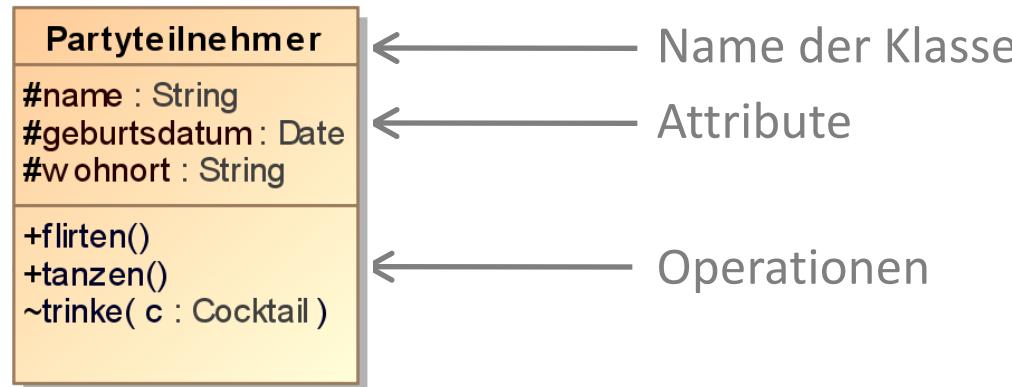
Strukturdiagramme

Motivation

- Die (eine) logische Struktur eines Systems wird während eines Projekts immer wieder benötigt:
 - Konzeptuelle, analytische Modellierung
 - Umsetzung von Fachkonzepten in UML, Modellierung der Anwendungsdomäne
 - Abstrahiert von Details, wie bspw. Operationen und Typen von Attributen
 - Klassen werden bspw. benutzt, um Eingabetypen von Use Cases und Aktivitäten zu modellieren
 - Logische, design-orientierte Modellierung
 - Modellierung der technischen Umsetzung (oftmals zur Codegenerierung)
 - Wesentlich detaillierter in der Anwendung als in obigem Fall

Klassen

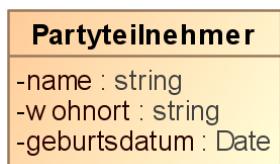
- Klassen haben einen **Namen**, eine Menge von **Attributen** und eine Menge von **Operationen**



Operationen können
weggelassen werden...

...ebenso wie Attribute...

...oder beide
Blöcke!



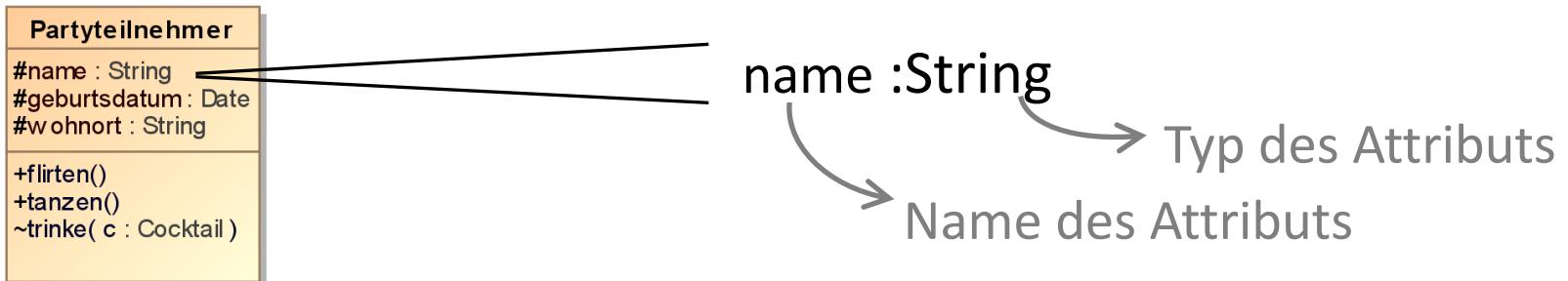
Klassendiagramm

Strukturdiagramme

Attribute

- Attribute beschreiben Eigenschaften der Klasse.
- Attribute haben einen Namen und können getypt sein.

Deklaration von Attributen in UML:



Mögliche Typen für Attribute:

Vordefinierte Datentypen

•int:	Ganze Zahlen (0, 1, -3, ...)
•double:	reelle Zahlen (1.2, 3.14, ...)
•String:	Zeichenketten ("abc", "123", ...)
•boolean:	Wahrheitswerte (true, false)

Außerdem:

Ebenfalls möglich und häufig benutzt werden Enumerationstypen (s. später) und eigene Klassen:

Gastgeber : Partyteilnehmer

Attributdeklaration - Mengenattribute

- Angabe von **Multiplizitäten** möglich um zu modellieren, dass ein Attribut nicht einen einzelnen Wert speichert, sondern eine **Menge von Werten**.



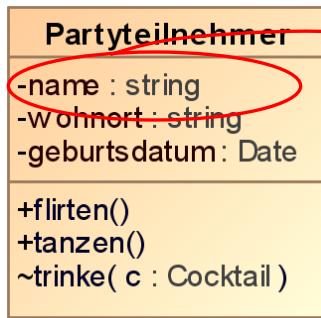
Multiplizität
in der Form:
[Untergrenze..Obergrenze]

allTeilnehmer ist eine beliebig große Menge von Partyteilnehmer oder genauer: eine beliebig große Menge von Instanzen von „Partyteilnehmer“.

Multiplizität	Beschreibung (Größe der Instanzenmenge)
0..*	Beliebig groß, auch die leere Menge ist möglich
*	= 0..*
1..*	Beliebig groß, aber mindestens 1
5..*	Beliebig groß, aber mindestens 5
2..2	Genau 2 (Abk.: [2])
3..6	Mind. 3, max. 6
n..m	Mind. n, max. m (Verallgemeinerung obiger Fälle)

Attributdeklaration - Sichtbarkeiten

- Objekte interagieren miteinander
 - durch Ändern der Attributwerte oder
 - durch Aufrufen von Operationen
- Sichtbarkeiten regeln, welche Objekte auf die Attribute einer Klasse zugreifen können.
=> So können Attribute vor ungewünschtem Zugriff geschützt werden.

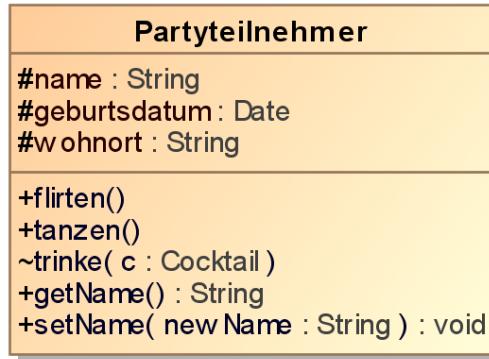


- name :
string

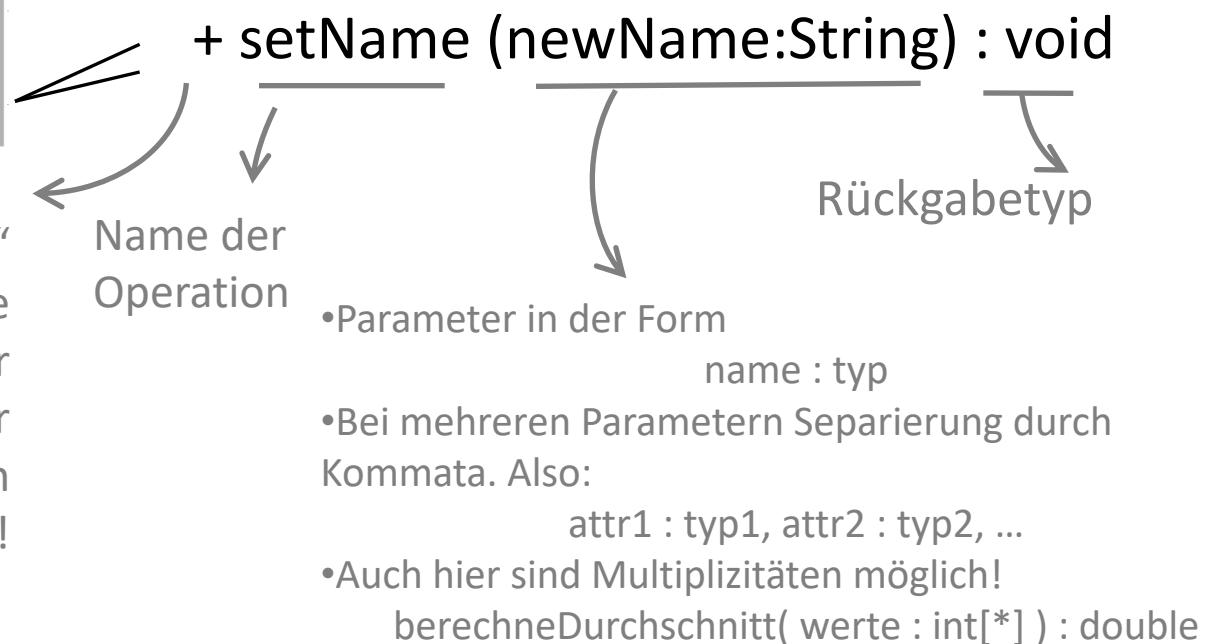
Sichtbarkeit: „-“ bedeutet „private“. Instanzen anderer Klassen können auf name **nicht** zugreifen! Das Attribut ist nach außen **nicht** sichtbar!

Weitere Sichtbarkeiten
lernen wir später kennen!

Deklaration von Operationen

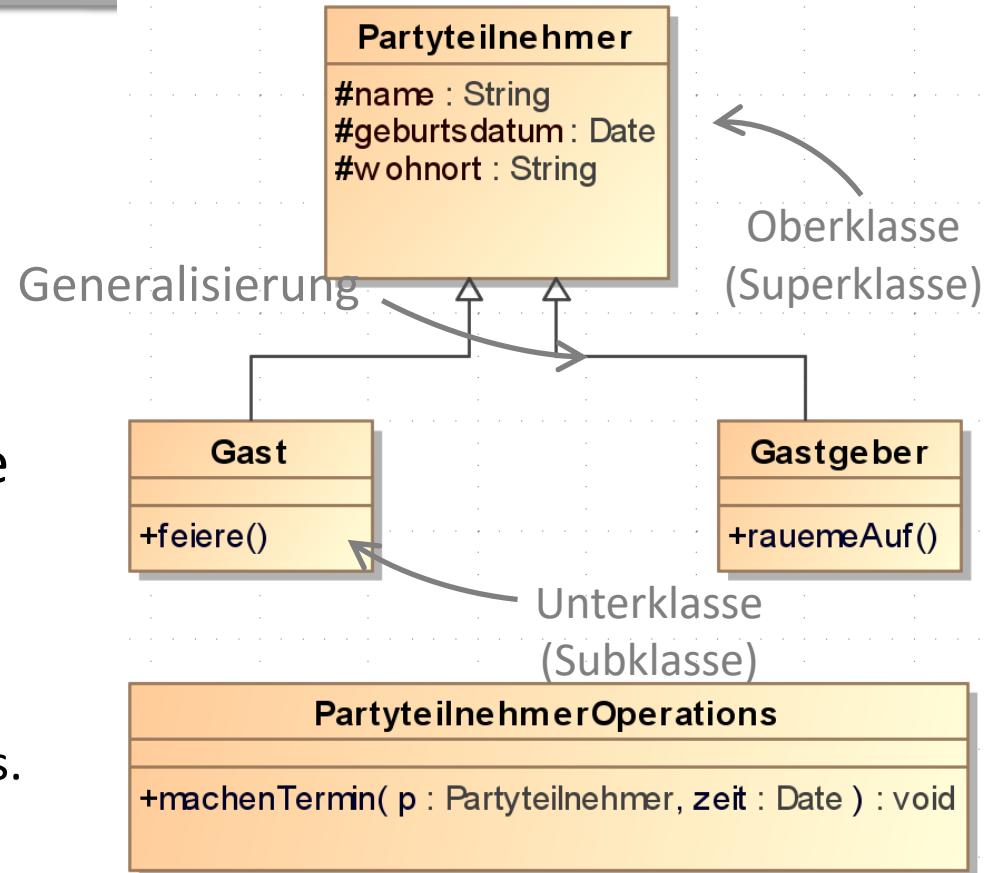


Sichtbarkeit: „+“ bedeutet „public“. Die Operation ist für Instanzen aller anderen Klassen sichtbar und aufrufbar!



Was tun mit Gemeinsamkeiten zwischen Klassen – Generalisierung

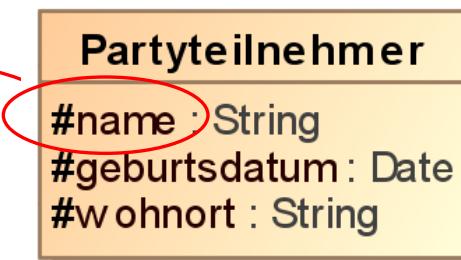
- Idee: Einführung einer gemeinsamen Oberklasse, die die gemeinsamen Eigenschaften beinhaltet.
- Die detaillierte Klassen werden zu Unterklassen, die die gemeinsamen Eigenschaften „erben“
 - Diese werden nicht explizit in den Unterklassen deklariert (s. Diagramm)



Wichtiges Detail der Generalisierung: Sichtbarkeiten beachten!

- Private Attribute sind in anderen Klassen nicht sichtbar
-> auch in Unterklassen nicht!
- Öffentliche Attribute sind in allen Klassen sichtbar
-> nicht gewollt!
- Weitere Sichtbarkeit erforderlich: „protected“

Sichtbarkeit: „#“
bedeutet „protected“.
Das Attribut ist nur für
Instanzen derselben
Klasse oder ihrer
Unterklassen sichtbar!



Sichtbarkeiten im Überblick



Bei folgenden Kombinationen kann C2 auf das Attribut a zugreifen:

... und für C2 gilt

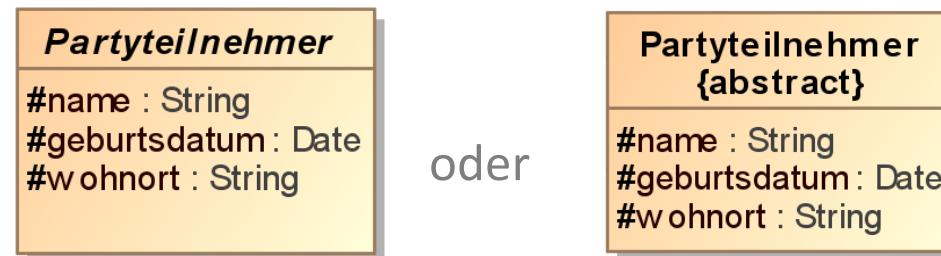
a ist deklariert

Name	Symbol	C2 = C1	C2 ist Unterklasse in anderem Paket	C2 im gleichen Paket	C2 anderswo
private	-	✓			
protected	#	✓	✓	✓	
package	~	✓		✓	
public	+	✓	✓	✓	✓

„Jeder Partyteilnehmer ist ein...“ – Nicht instanzierbare Klassen

- Instanzen von Klassen sind Objekte. Im Beispiel ist aber jeder Partyteilnehmer ein Gast, ein Gasgeber, oder oder oder... niemals aber nur ein „Partyteilnehmer“.
- Das bedeutet: Partyteilnehmer wird nicht instanziert. Die Klasse dient nur zur Strukturierung. Partyteilnehmer ist eine **abstrakte Klasse**.
- Operationen von Partyteilnehmer müssen nicht durch Methoden implementiert werden.

Name in *kursiv*

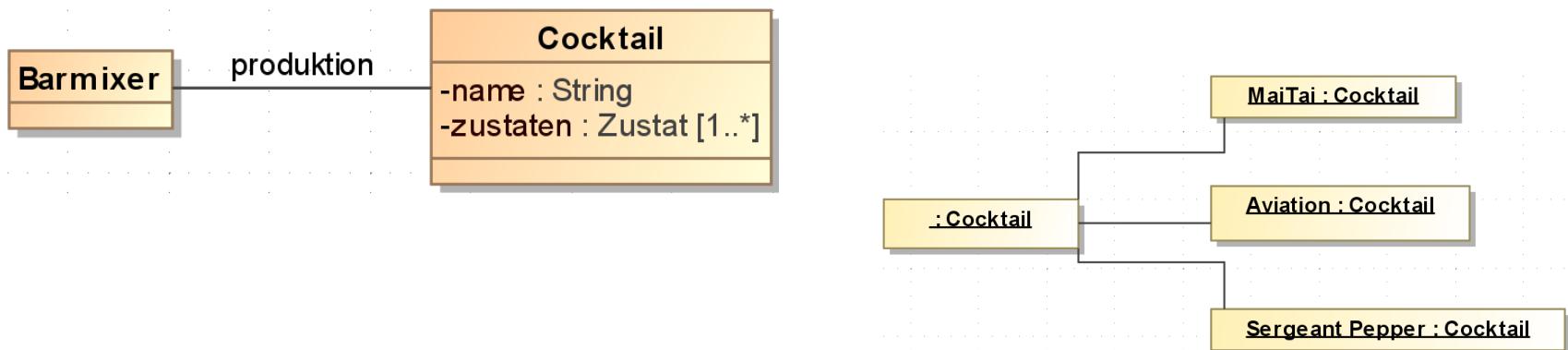


Schnittstellen

- Schnittstellen sind ähnlich zu Klassen, sozusagen die abstrakteste Form von Klassen:
 - Nicht instanzierbar
 - Keine privaten Eigenschaften
 - Zu **keiner** Operation wird eine Methode implementiert
- Klassen können Schnittstellen realisieren
 - Dazu müssen sie zu jeder Operation eine Methode implementieren

Die Verbindungen zwischen Klassen - Assoziationen

- Klassen existieren nicht losgelöst voneinander – oft bestehen Beziehungen zwischen ihnen.
- Bsp.: jedes Cocktail wird von einem Barmixer produziert, bei dem Partyteilnehmer die Getränke beschaffen kann
=> „Barmixer“ und die Beziehung zu „Cocktail“ sollten auch modelliert werden!



Klassendiagramm

Strukturdiagramme

Assoziationen im Detail

- Obligatorisch ist nur der Name einer Assoziationen
- Weitere optionale Details:
 - Rollennamen an den Assoziationsenden (+Sichtbarkeit)

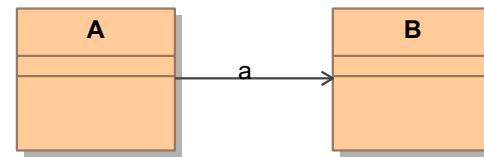


- Erläuterung der Leserichtung



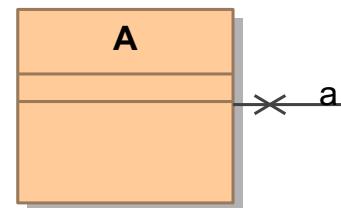
Assoziationen im Detail

- Anzeige der Navigierbarkeit: Durch Pfeilspitzen kann man ausdrücklich modellieren, dass man von einer Instanz an einem Ende der Assoziation zu der Instanz am anderen Ende navigieren kann:



Von einer Instanz von a kann auf die verbundene Instanz von b zugegriffen werden, aber nicht umgekehrt.

Oft auch zu sehen: Nicht navigierbare Enden werden durchgekreuzt:



Assoziationen im Detail

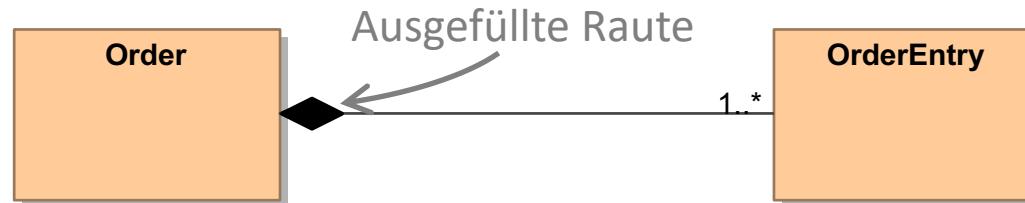
- Auch bei Assoziationsenden benutzt: Multiplizitäten.
Man will auch Sachverhalte modellieren wie: Ein Anbieter hat mehrere Produkte im Angebot, jedes Produkt wird aber genau von einem Anbieter angeboten:



Sind Multiplizitäten weggelassen, wird „1“ angenommen.

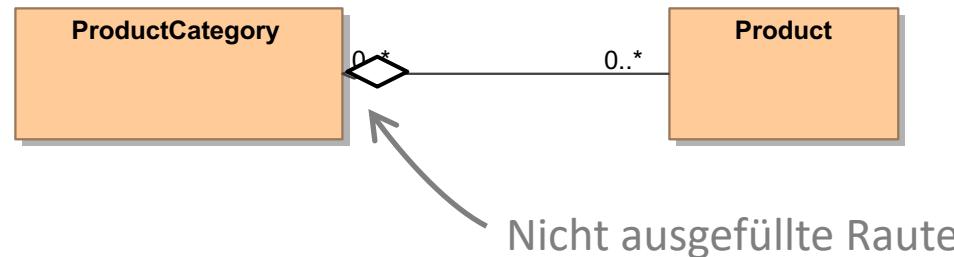
Sonderformen von Assoziationen – Komposition

- Oft wird eine sog. „Teil-Ganzes“-Beziehung modelliert :
 - Person – Gruppe
 - Auto – Räder (-Chassis, Lenkrad, etc.)
 - Mensch – Kopf (-Rumpf, -Arme, etc.)
- Dafür gibt es in UML Sonderformen der Assoziation
 - **Komposition:** Das Ganze „besitzt“ seine Teile, jedes Teil gehört zu *maximal* einem Ganzen.
 - Wird das Ganze gelöscht, werden auch seine Teile gelöscht.



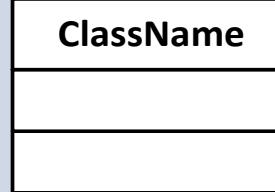
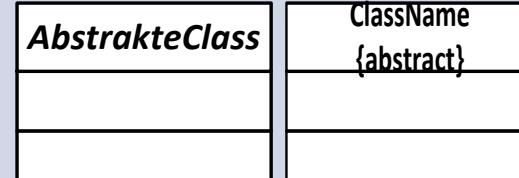
Sonderformen von Assoziation – Aggregation

- Aggregation: Ist ebenfalls eine Teil-Ganzes-Beziehung. Ein Teil kann jedoch zu mehreren Ganzen gehören (eine Person kann mehreren Gruppen angehören).
- Wird das Ganze gelöscht, lebt das Teil weiter!

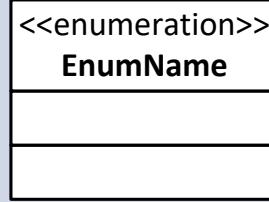
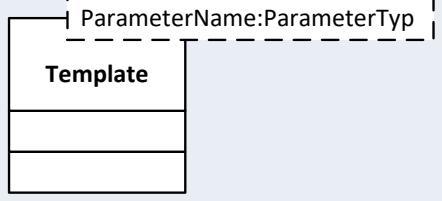


- Ein Produkt kann in mehrere Kategorien eingeordnet werden!
- Wird eine Kategorie von Produkten gelöscht, existieren die Produkte doch trotzdem weiter.

Appendix – Auflistung der wichtigsten Elemente in Klassendiagrammen.

Elementtyp	Symbole	Anmerkung
normale Klasse		
Schnittstelle		
Abstrakte Klasse		<ul style="list-style-type: none">- Klassenname Kursiv- Oder Schlüsselwort „abstract“

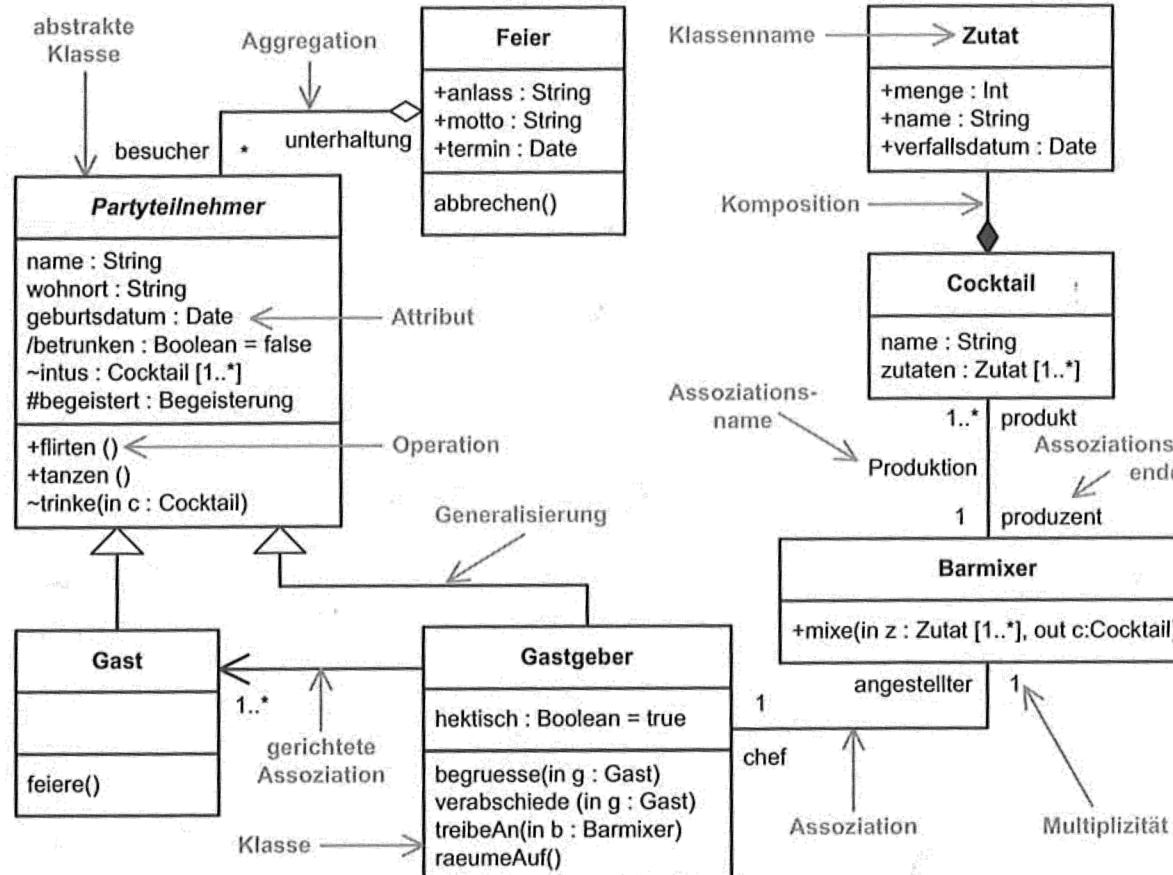
Appendix – Auflistung der wichtigsten Elemente in Klassendiagrammen.

Elementtyp	Symbole	Anmerkung
Stereotyp		
Template		
Paket		

Appendix – Auflistung der wichtigsten Elemente in Klassendiagrammen

Elementtyp	Symbole	Anmerkung
Assoziation		
Aggregation		
Komposition		
Vererbung		
Schnittstellen-realiserierung		

Klassendiagramm



Grafik: Rupp, Queins, Zengler, UML2 Glasklar, Seite 103

Inhalt

- **Beschreibungstechniken und Basiswissen**
 - Grundlagen der Objektorientierung (Wiederholung)
 - Übersicht zur UML
- **Strukturdiagramme**
 - Klassendiagramm
 - **Paketdiagramm**
 - Objektdiagramm
 - Kompositionssstrukturdiagramm
 - Komponentendiagramm
 - Verteilungsdiagramm
- **Verhaltensdiagramme**
 - Use-Case-Diagramm
 - Aktivitätsdiagramm
 - Zustandsautomat
- **Interaktionsdiagramme**
 - Sequenzdiagramm
 - Kommunikationsdiagramm
 - Timingdiagramm
 - Interaktionsübersichtsdiagramm
- **OCL (Object Constraint Language)**

Was ist ein Paket?

- **Pakete und Unterpakete**

Pakete gruppieren Elemente und definieren Namensräume, in denen sich diese Elemente befinden.

Beziehungen zwischen Paketen

- Import

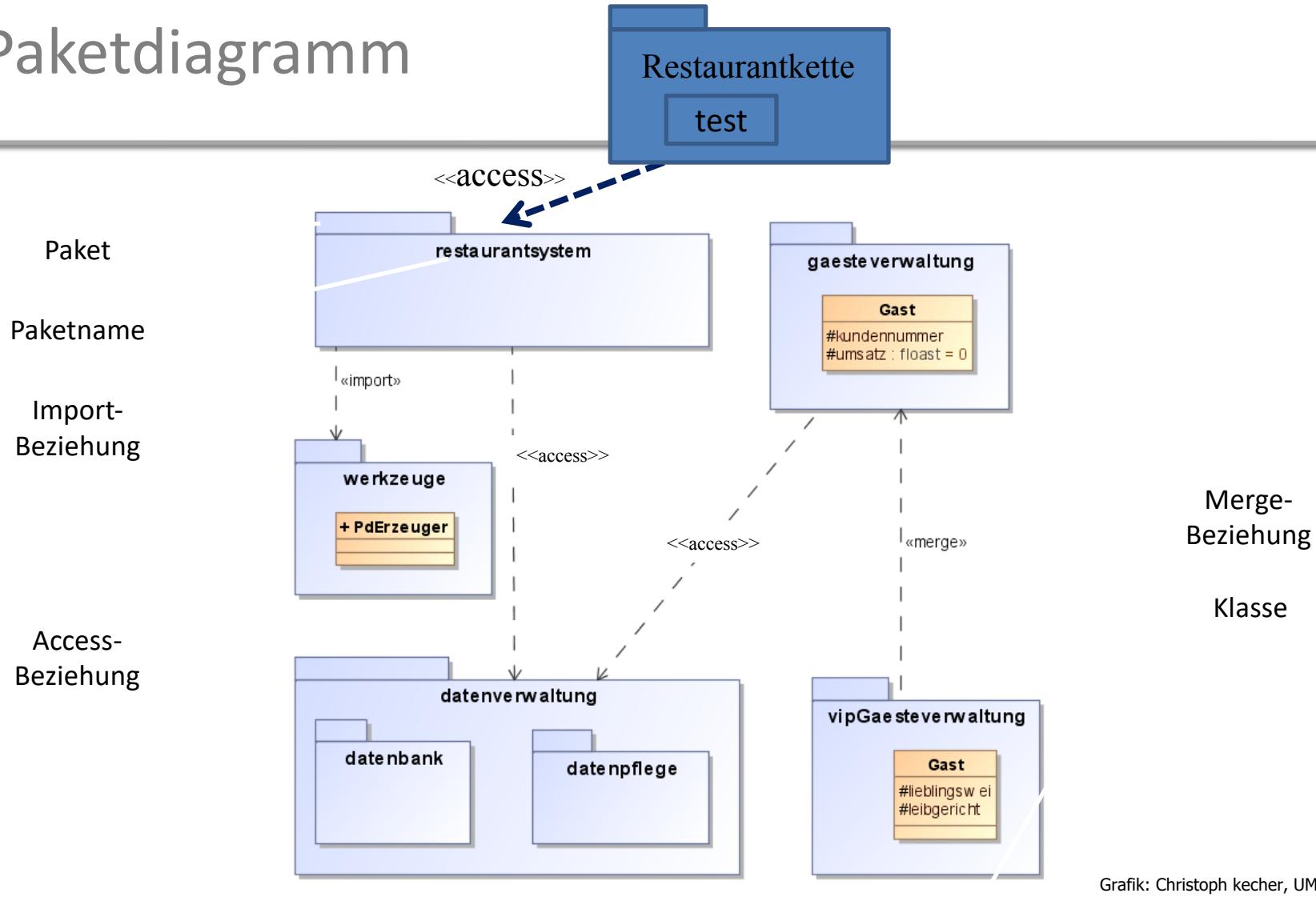
Ein Paket-Access ist eine Beziehung, die alle **Namen öffentlicher Elemente** eines Pakets zum importierenden Paket **als öffentlich** hinzufügt.
- Access

Ein Paket-Access ist eine Beziehung, die alle **Namen öffentlicher Elemente** eines Pakets zum importierenden Paket **als private** hinzufügt.
- Merge:

Ein Paket-Merge definiert eine Beziehung zwischen zwei Paketen, bei der die **nicht privaten Inhalte** des Zielpakets in die Inhalte des Quellpakets **verschmolzen werden**.

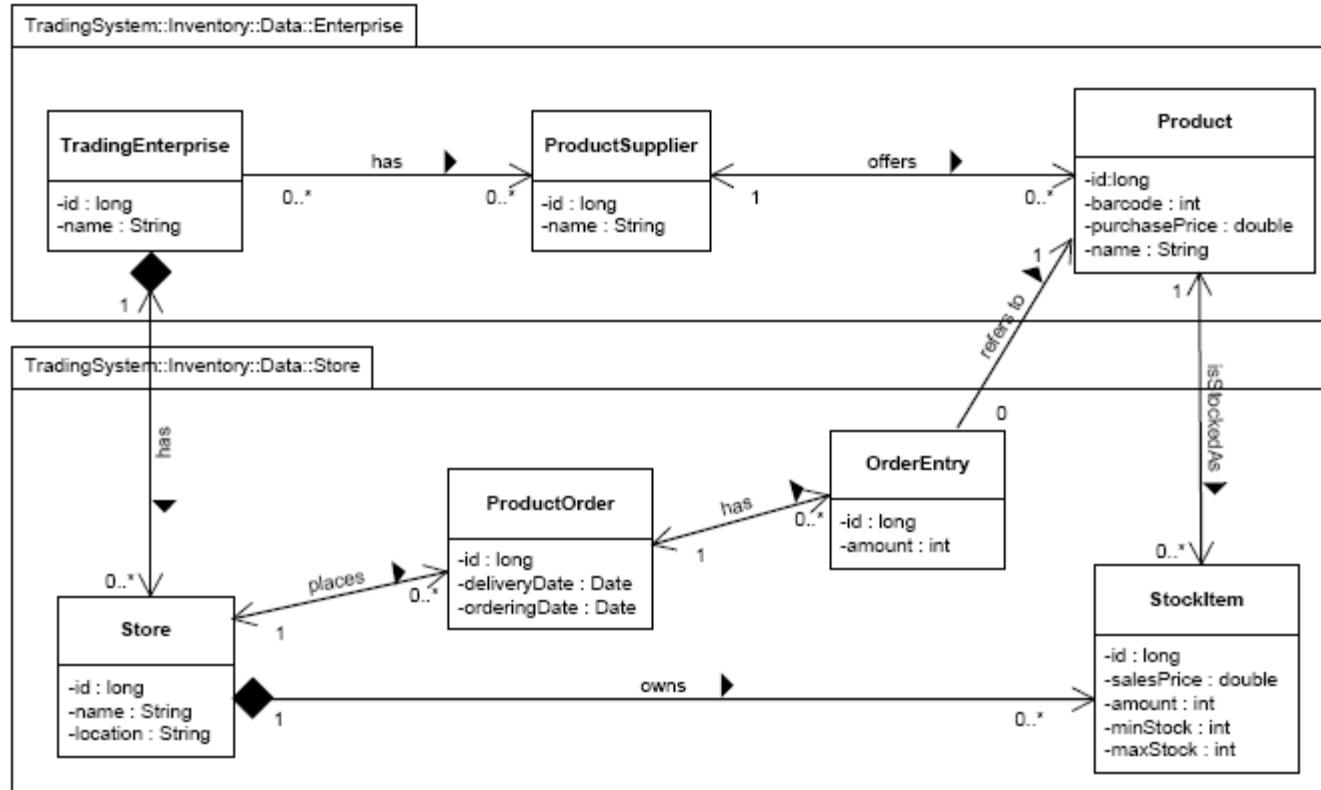
Kecher, UML2

Paketdiagramm



Grafik: Christoph kecher, UML2, Seite 174

Beispiel für ein „realistisches“ Klassendiagramm



Paketdiagramm

Strukturdiagramme

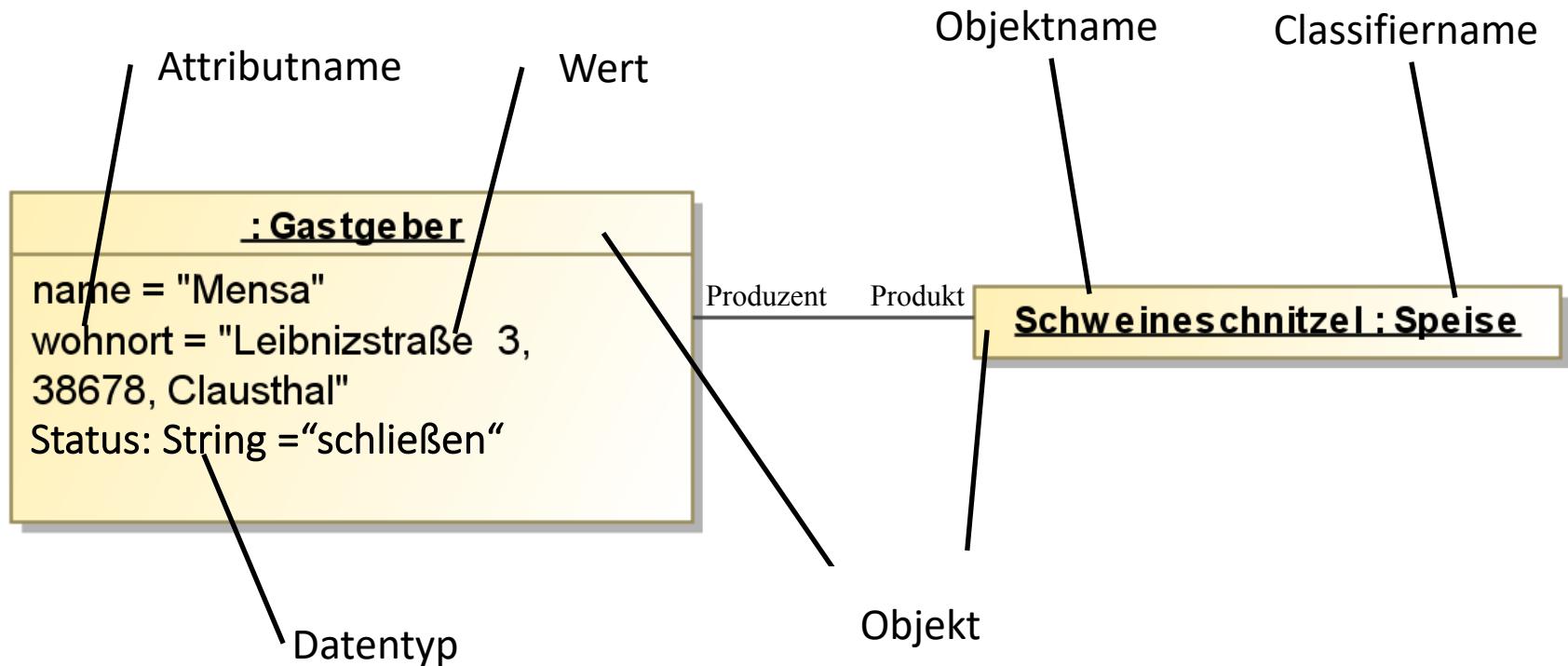
Inhalt

- **Beschreibungstechniken und Basiswissen**
 - Grundlagen der Objektorientierung (Wiederholung)
 - Übersicht zur UML
- **Strukturdiagramme**
 - Klassendiagramm
 - Paketdiagramm
 - **Objektdiagramm**
 - Kompositionssstrukturdiagramm
 - Komponentendiagramm
 - Verteilungsdiagramm
- **Verhaltensdiagramme**
 - Use-Case-Diagramm
 - Aktivitätsdiagramm
 - Zustandsautomat
- **Interaktionsdiagramme**
 - Sequenzdiagramm
 - Kommunikationsdiagramm
 - Timingdiagramm
 - Interaktionsübersichtsdiagramm
- **OCL (Object Constraint Language)**

Was ist Objektdiagramm?

- Als Sonderfall eines Klassendiagramms
- Modellierung der Beziehungen zwischen
 - Objekte
 - Attributen
 - Komponenten
- Nur notwendig, wenn komplexe Klassendiagramme verdeutlicht und überprüft werden sollen

Objektdiagramm



Elemente in einem Objektdiagramm

- Objekte mit
 - Namen und Objekttyp
 - Attribute aus
 - Attributname, Datentyp und aktueller Wert
- Link zwischen Objekten
 - Linkname
 - leserrichtung

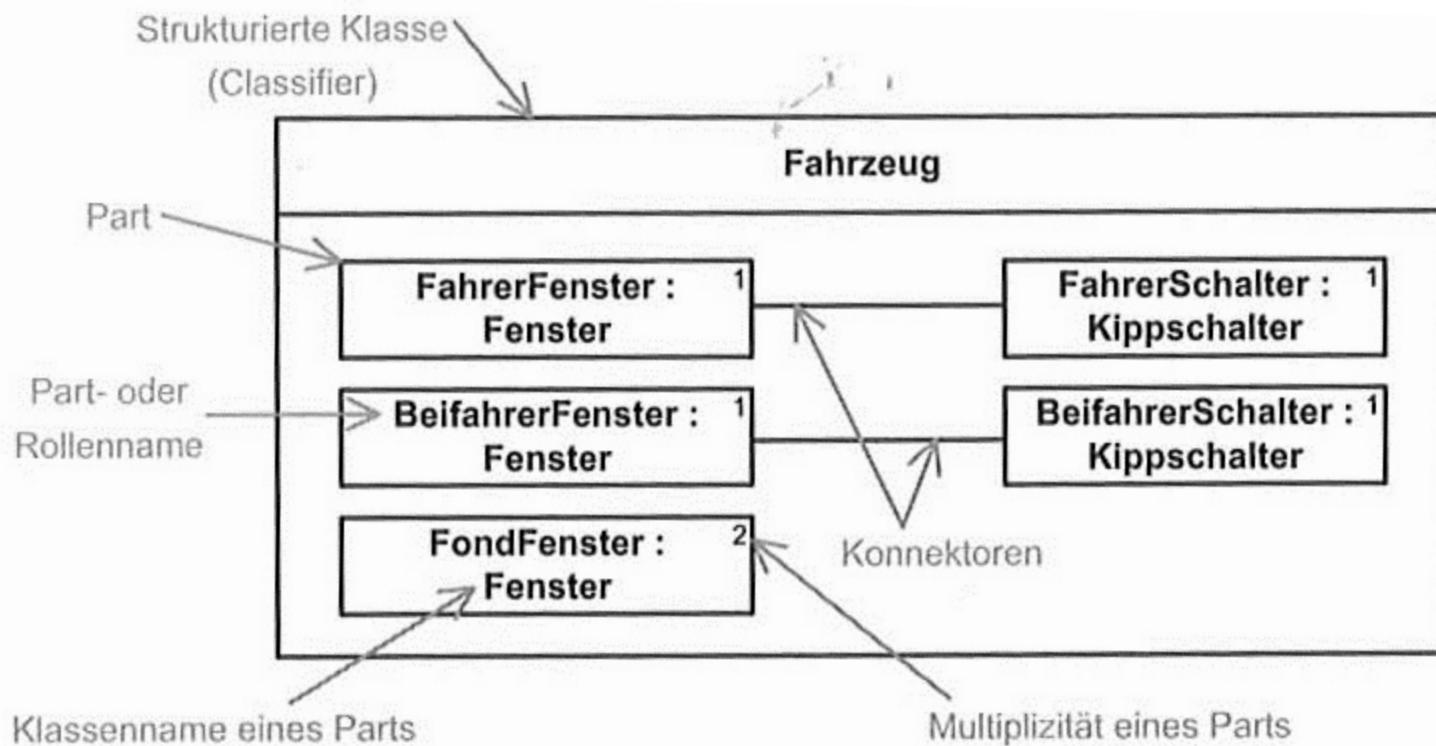
Inhalt

- **Beschreibungstechniken und Basiswissen**
 - Grundlagen der Objektorientierung (Wiederholung)
 - Übersicht zur UML
- **Strukturdiagramme**
 - Klassendiagramm
 - Paketdiagramm
 - Objektdiagramm
 - **Kompositionssstrukturdiagramm**
 - Komponentendiagramm
 - Verteilungsdiagramm
- **Verhaltensdiagramme**
 - Use-Case-Diagramm
 - Aktivitätsdiagramm
 - Zustandsautomat
- **Interaktionsdiagramme**
 - Sequenzdiagramm
 - Kommunikationsdiagramm
 - Timingdiagramm
 - Interaktionsübersichtsdiagramm
- **OCL (Object Constraint Language)**

Was ist Kompositionsstrukturdiagramm?

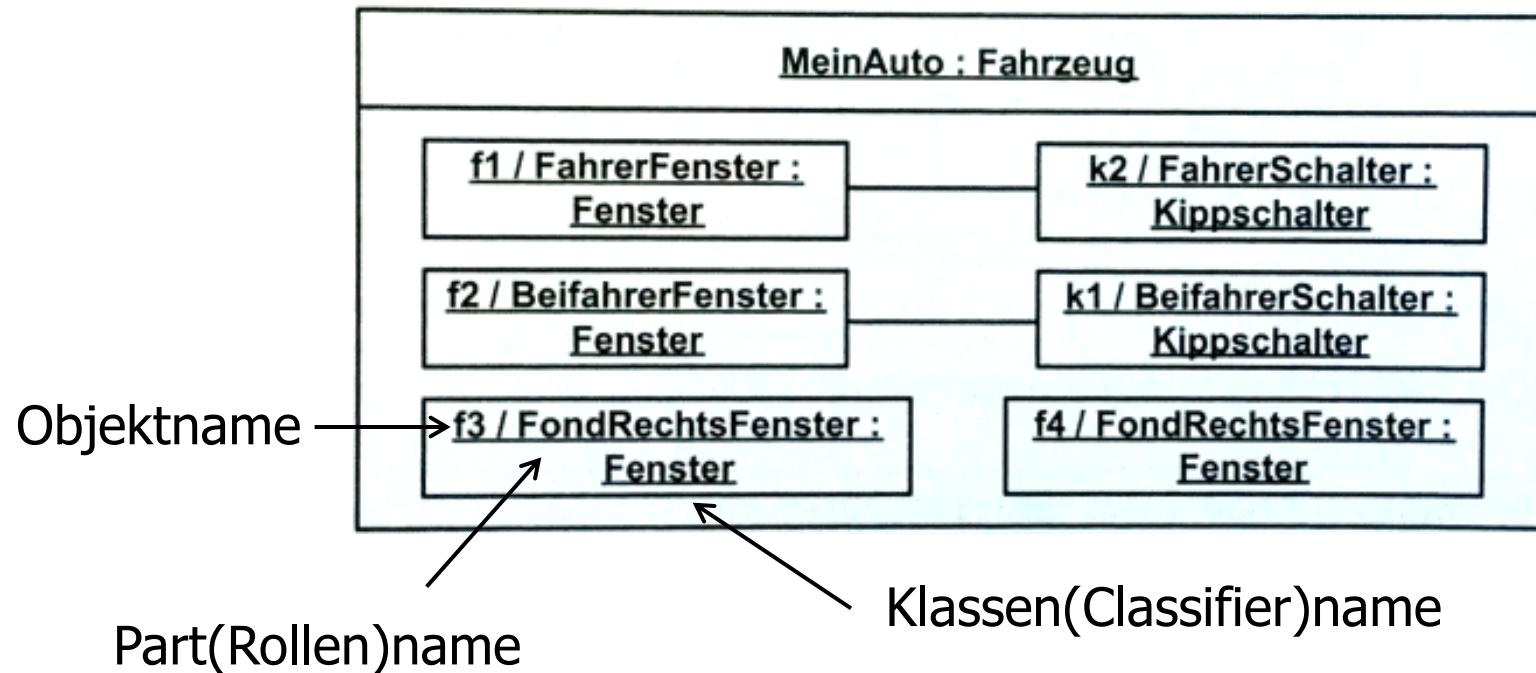
- Als Architekturdiagramme gezeichnet
- Beschreiben
 - der Interne Struktur einer Klasse oder Komponente
 - der Interaktionsbeziehungen zu anderen Systembestandteilen
- Verwendet in den frühen Phase wie Analyse oder Design

Kompositionssstrukturdiagramm



Grafik: Rupp, Queins, Zengler, UML2 Glasklar, Seite 195

Kompositionssstrukturdiagramm auf Instanzebene

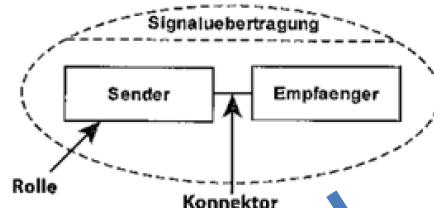


Notation:

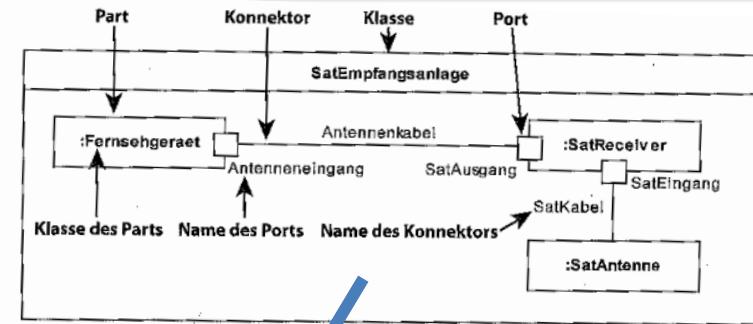
Objektname / Part (Rollen) name : Klassen (Classifier) name

Grafik: Rupp, Queins, Zengler, UML2 Glasklar, Seite 196

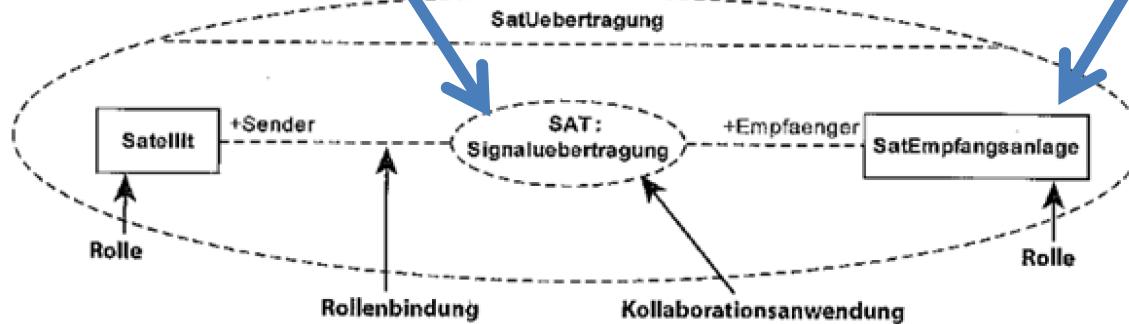
Kollaborationen und die Anwendung



Kollaboration



Instanziert von



- Top-Down-Lösung
- Teil-Lösungsmuster zu einem großen Lösungsmuster

Grafik: Christopf Kecher, Seite 126

Elemente in einem Kompositionssstrukturdiagramm (1)

- Top-Level Komponente bestehend aus
 - Parts mit
 - Part (Rollen)nam,
 - Parttyp
 - Multiplizitäten
 - Konnektoren
- Notation der Partbezeichnung

Part (Rollen) name : Parttyp [Multiplizität]

Elemente in einem Kompositionssstrukturdiagramm (2)

- Kollaboration
 - Name
 - Klasse (Rolle)
 - Konnektor

Inhalt

- **Beschreibungstechniken und Basiswissen**
 - Grundlagen der Objektorientierung (Wiederholung)
 - Übersicht zur UML
- **Strukturdiagramme**
 - Klassendiagramm
 - Paketdiagramm
 - Objektdiagramm
 - Kompositionssstrukturdiagramm
 - **Komponentendiagramm**
 - Verteilungsdiagramm
- **Verhaltensdiagramme**
 - Use-Case-Diagramm
 - Aktivitätsdiagramm
 - Zustandsautomat
- **Interaktionsdiagramme**
 - Sequenzdiagramm
 - Kommunikationsdiagramm
 - Timingdiagramm
 - Interaktionsübersichtsdiagramm
- **OCL (Object Constraint Language)**

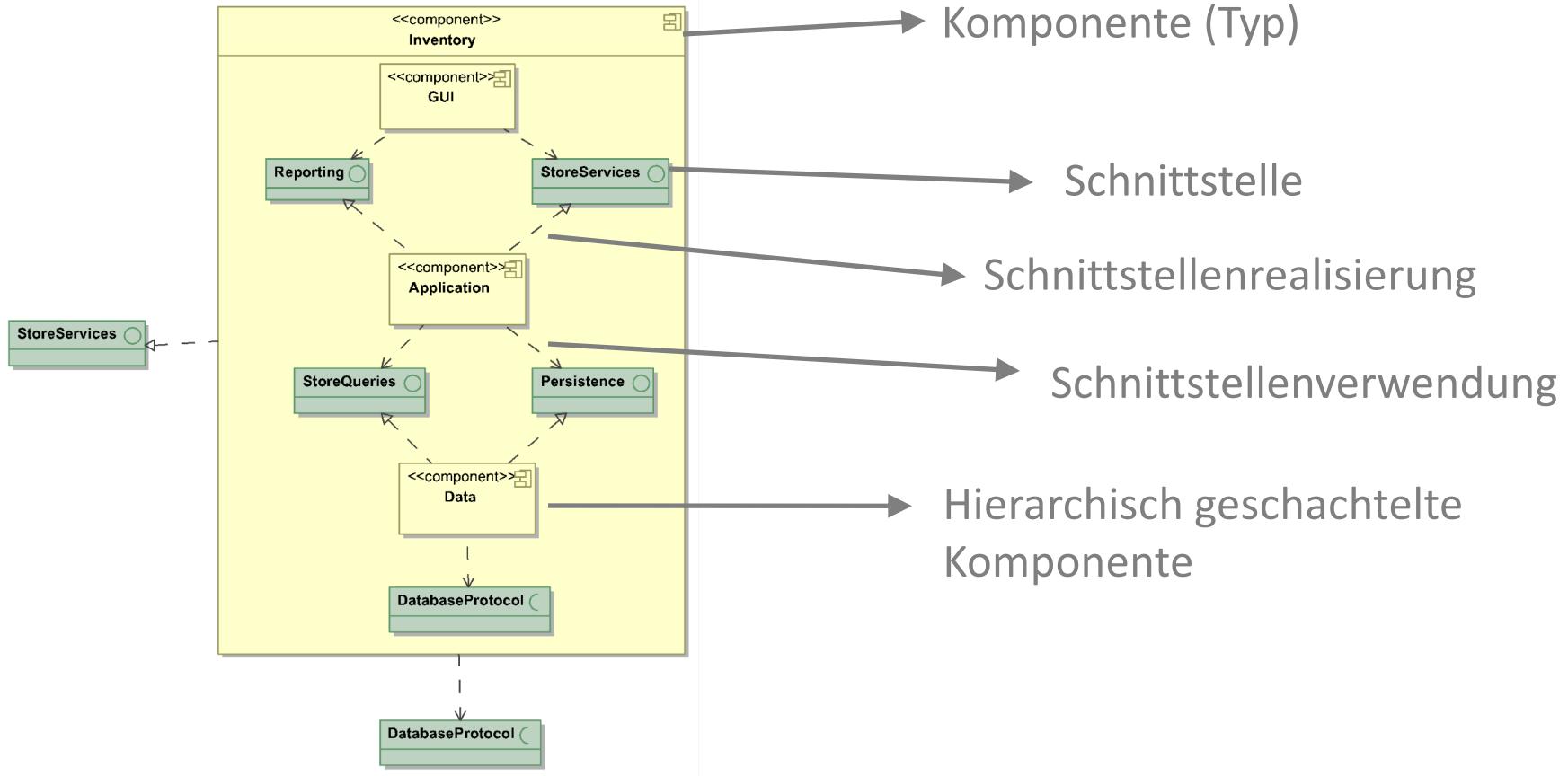
Komponentendiagramm – was ist das?

- Komponentendiagramme (KD) beschreiben eine Struktur eines Systems in Form von Komponenten, Schnittstellen und den Verbindungen zwischen Komponenten.
- Dabei wird meist die logische Struktur oder die physikalische Struktur eines Systems beschrieben.
- Komponenten stellen Bestandteile mit einem abgrenzbaren und definierten Verhalten dar. Dieses wird über Schnittstellen beschrieben. Komponenten mit den selben Schnittstellen können gegeneinander ausgetauscht werden, ohne dass das restliche System geändert werden muss.

Motivation

- Komponentendiagramme (Architektur) stellen die Struktur des Systems größer dar als später erstellte, detailliertere Klassendiagramme (Design).
- Sie können daher als wichtiges Hilfsmittel beim Übergang von der Anforderungsanalyse zum Entwurf des Systems sein.

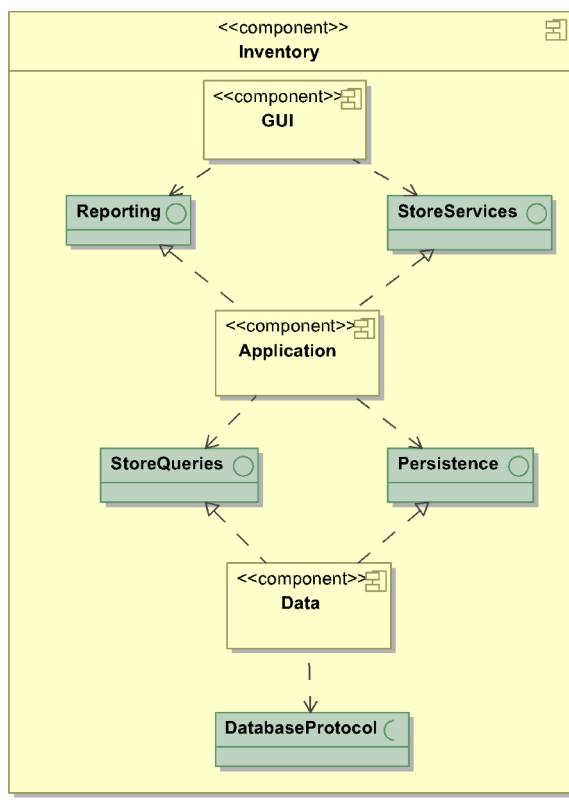
Wichtige Elemente im Überblick



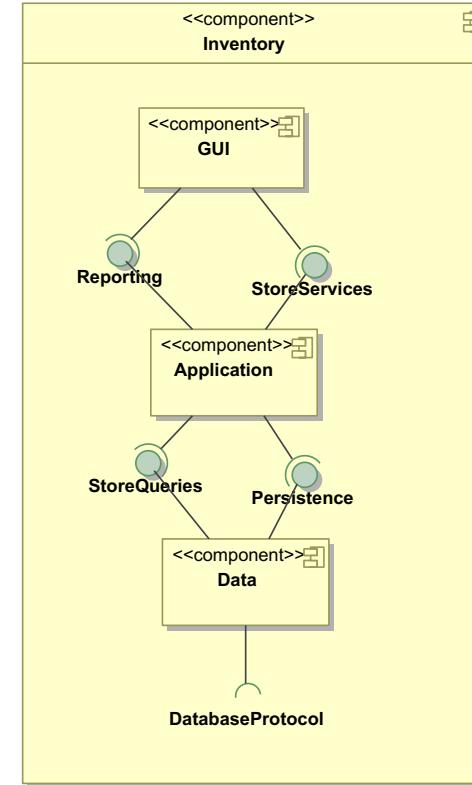
Komponentendiagramm

Strukturdiagramme

Unterschiedliche Notationen



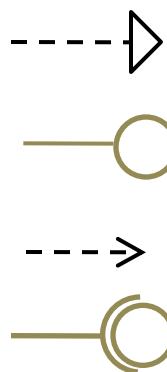
entspricht



Komponentendiagramm

Strukturdiagramme

Wichtige Elemente



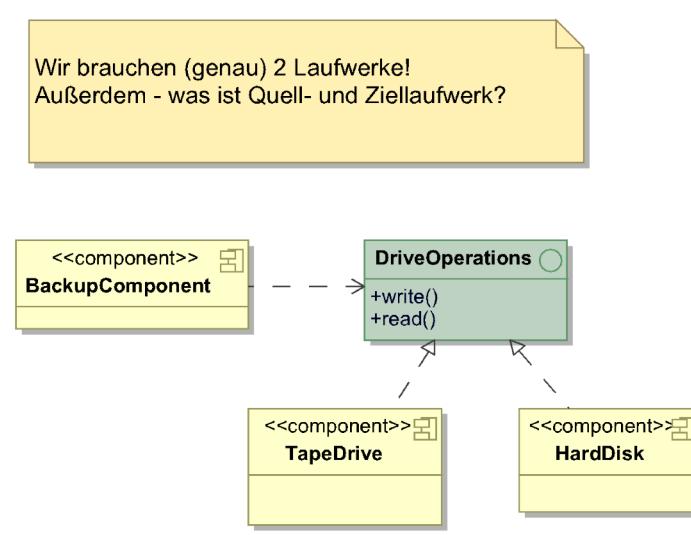
„Lollipop-Notation“

- **Komponenten:** kapseln modulare Teile eines Systems -> bspw. Gruppen von Komponenten, Klassen, etc. Besitzen ein Verhalten, das über die angebotenen und erforderlichen Schnittstellen definiert wird. Definieren einen Typ.
- **Schnittstellen:** Definieren eine Menge von Operationen, implementieren diese aber nicht. Sind ähnlich (aber nicht identisch) zu vollständig abstrakten Klassen.
- **Schnittstellenrealisierung:** zeigt an, dass die Komponente die verbundene Schnittstelle realisiert, das heißt „in ihrem Inneren“ eine Implementierung bereitstellt.
- **Schnittstellenverwendung:** zeigt an, dass die Komponente die verbundene Schnittstelle benötigt und verwendet, um ihr eigenes Verhalten umzusetzen (Aufruf von Operationen)

Reichen diese Beschreibungselemente?

- Mit diesen Mitteln kann man beschreiben, welche Schnittstellentypen eine Komponente anbietet oder benötigt.
- Was nicht geht: Beschreibung, dass eine Komponente eine bestimmte Anzahl von Instanzen einer Schnittstelle benötigt!
- Bsp.:

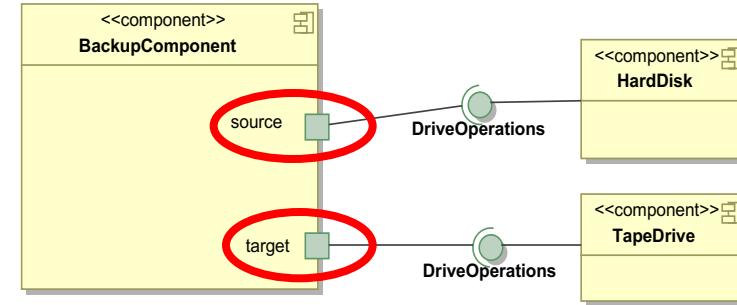
Wir brauchen (genau) 2 Laufwerke!
Außerdem - was ist Quell- und Ziellaufwerk?



Problem: Mit bisherigen Mitteln können wir nur über die Typebene sprechen, nicht über Instanzen!

Ports

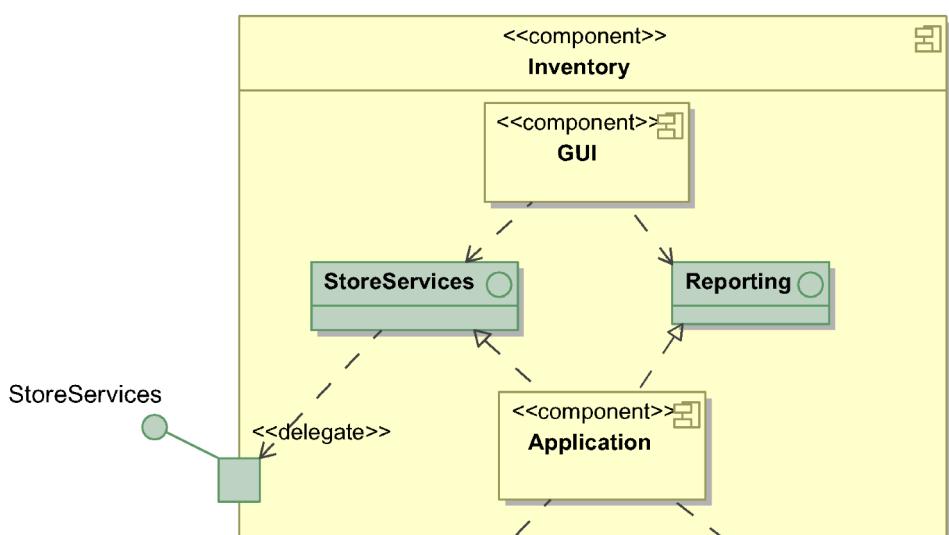
- Sachverhalt im Beispiel: Quelle und Ziel der Backupkomponente sind zwar vom gleichen Typ, müssen aber unterscheidbar sein. Wir brauchen sozusagen zwei unterschiedliche Variablen.
- Lösung: so genannte **Ports**



Ports

- stellen eindeutig identifizierbare Interaktionspunkte zwischen der Außenwelt und der internen Realisierung einer Komponente dar (haben einen eindeutigen Namen)
- können getypt und mit Multiplizitäten versehen werden
- können mit benötigten und angebotenen Schnittstellen versehen werden.
- Wird ein interner Classifier (Klasse, Komponente) als Typ angegeben, bietet der Port die durch den Classifier realisierten Interfaces an

Delegation



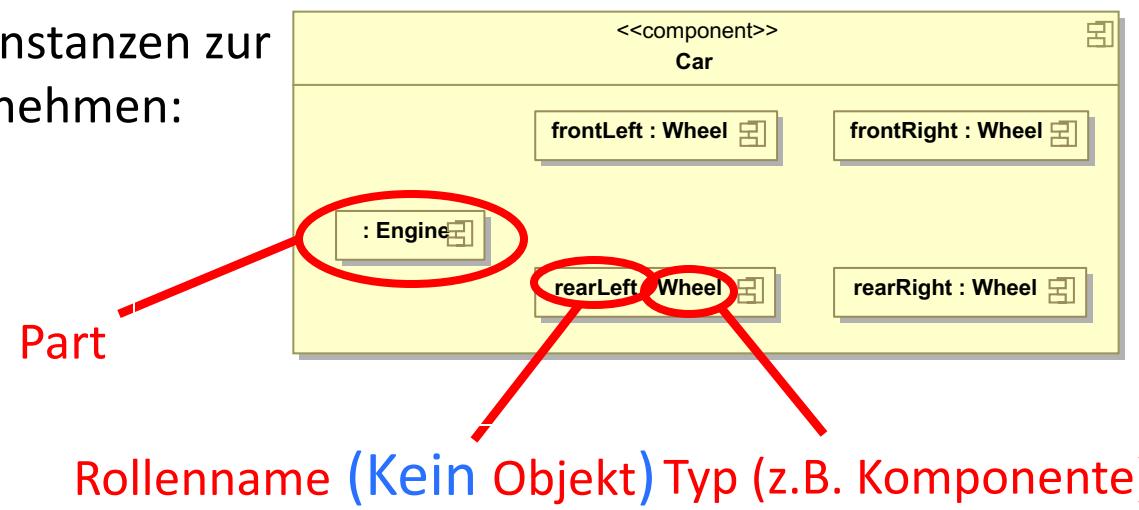
Über Ports und „Delegationskonnektoren“ werden Schnittstellen von inneren Komponenten nach außen geleitet .

Der innere Zusammenhalt – Teile einer Komponente

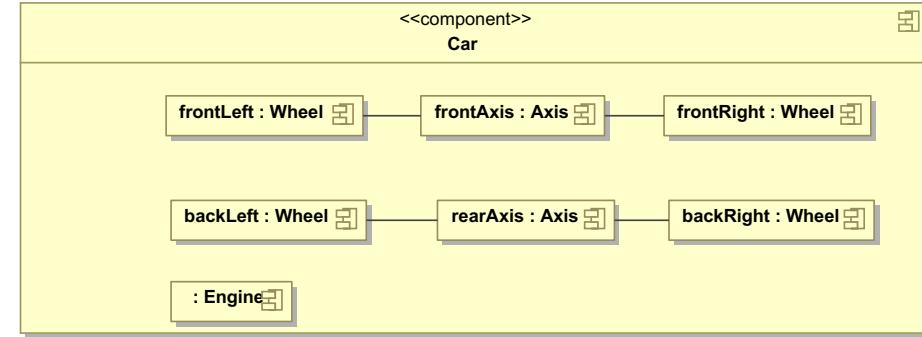
- Mit den bisherigen Mitteln ist darstellbar, welche Typen eine Komponente bereitstellt, ggf. auch innere Typen
- Über ihre Realisierung wird aber noch nichts gesagt!
- Umgangssprachlich:
 - Darstellbar: „Mit einem Auto kann ich fahren (angebotene Schnittstelle), dafür hat es Räder, Achsen, (innere Typen)
 - Nicht darstellbar: „Ein Auto hat vier Räder, zwei Achsen, eine verbindet die Räder vorne links und vorne rechts...“
- Verwendung von **Kompositionsstrukturdiagrammen** unter Benutzung von Parts und Konnektoren.

Parts und Konnektoren

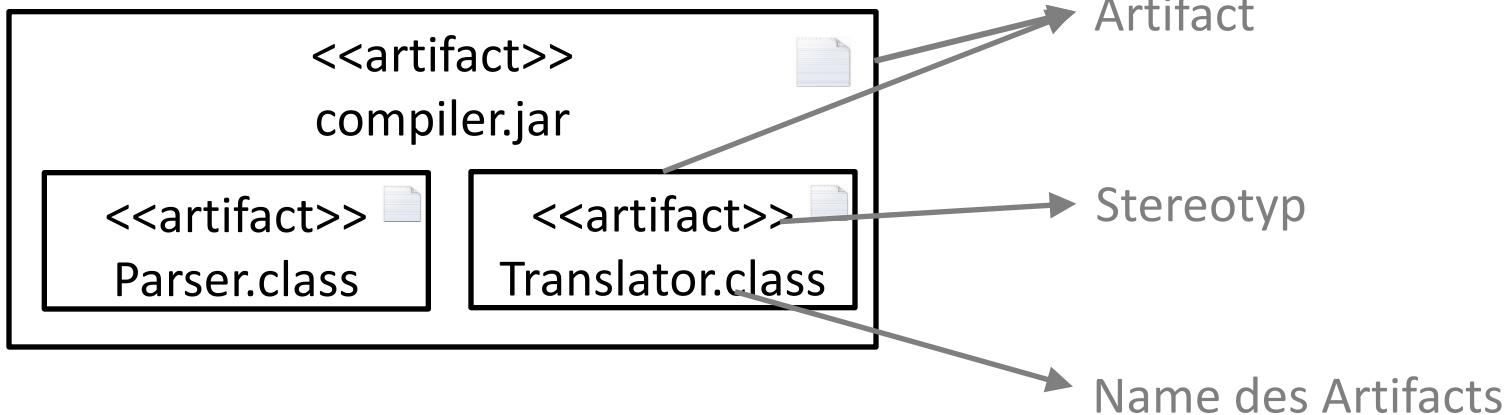
Parts bezeichnen i. A. Rollen, die Instanzen zur Realisierung der Komponente einnehmen:



- Verbindung von Parts über Konnektoren. Konnektoren bezeichnen ebenfalls Rollen für Links zwischen Instanzen



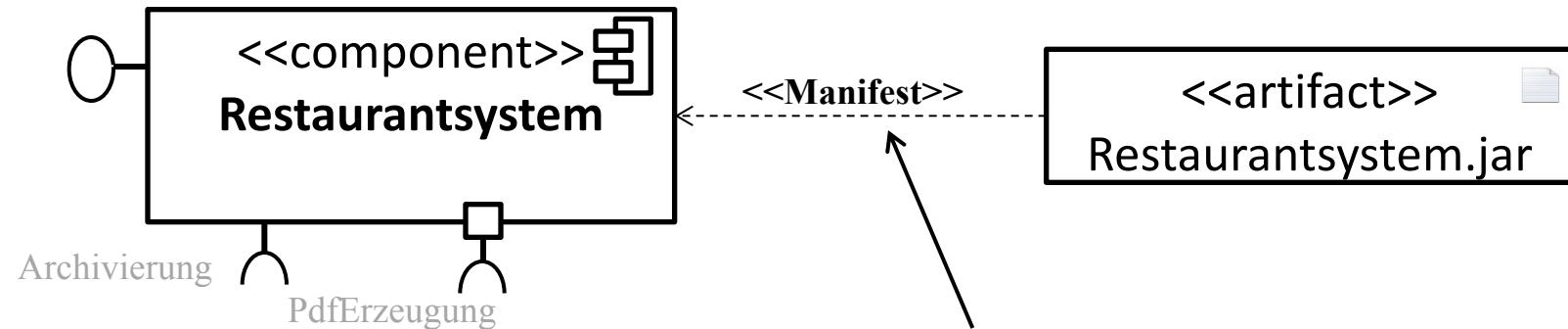
Artifact



- Eine physische Informationseinheit
- Beim Softwareentwicklungsprozess verwenden
- Schachteln ist möglich
- Können konkretisiert in Stereotypen wie
<<File>>, <<document>>, <<executable>>,
<<source>>, <<library>>

Artifact: Beziehung

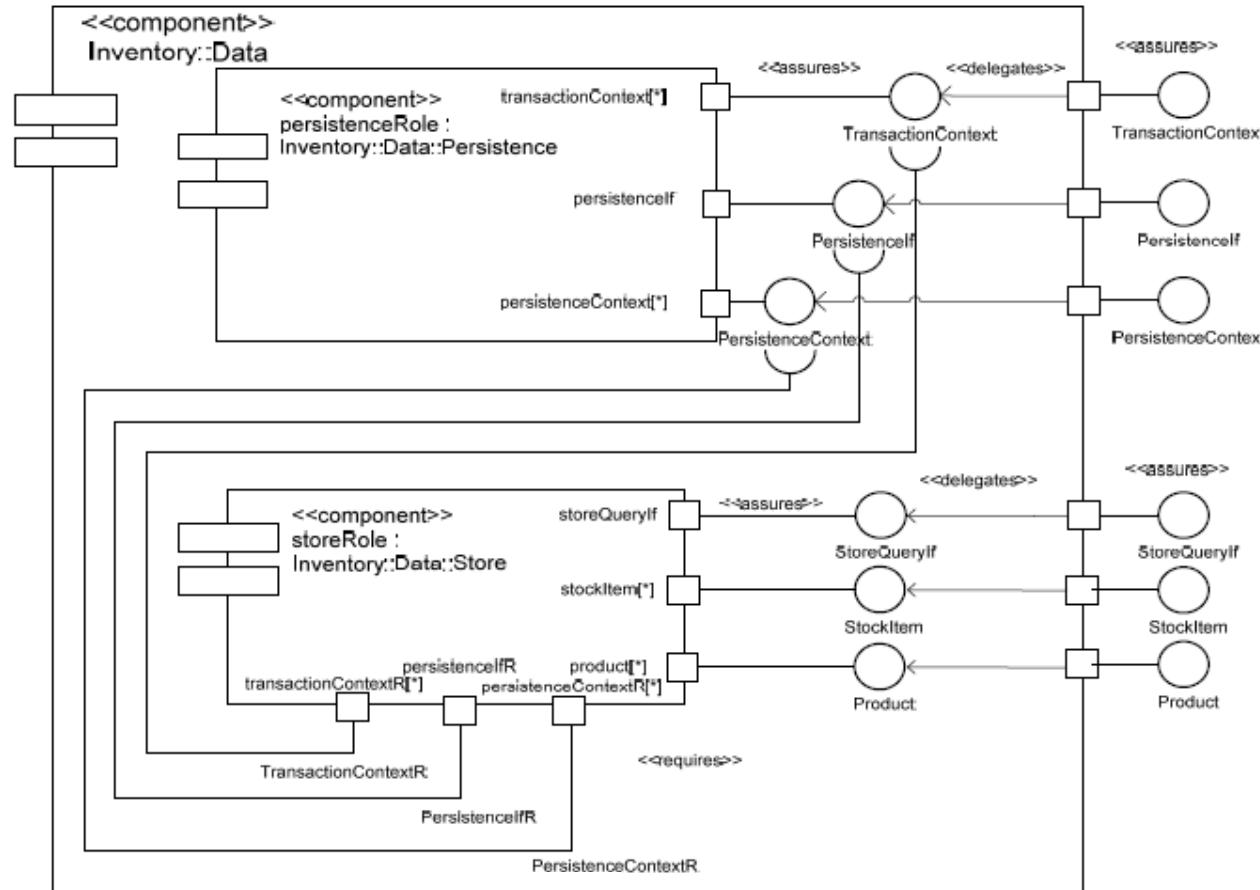
UmlExport



Manifest-Beziehung

- Modellierung der Realisierung einer Komponente
- UML Stereotypen für Manifestationsbeziehung
 - <<tool-generated>>
 - <<custom code>>

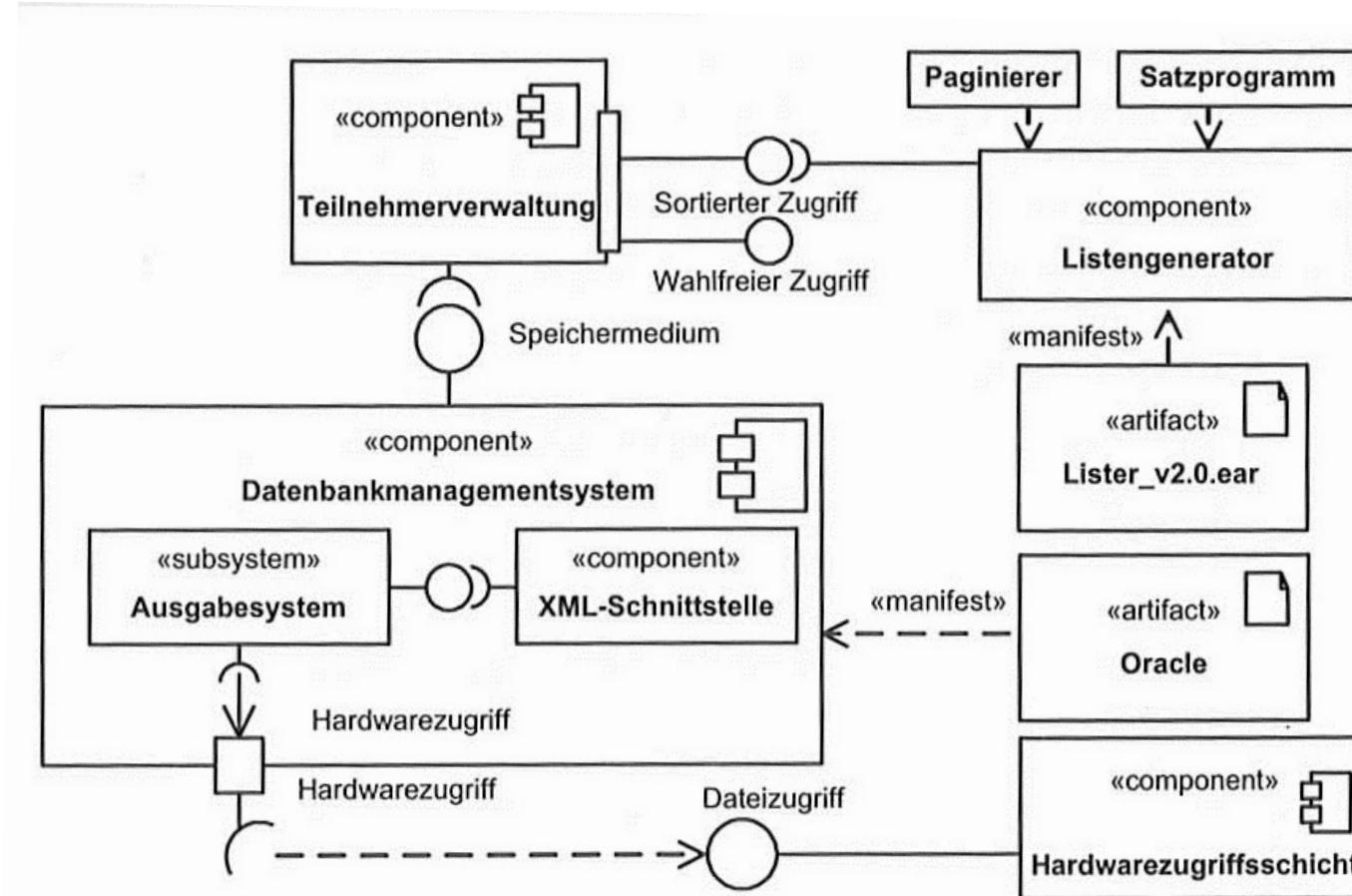
Beispiel



Komponentendiagramm

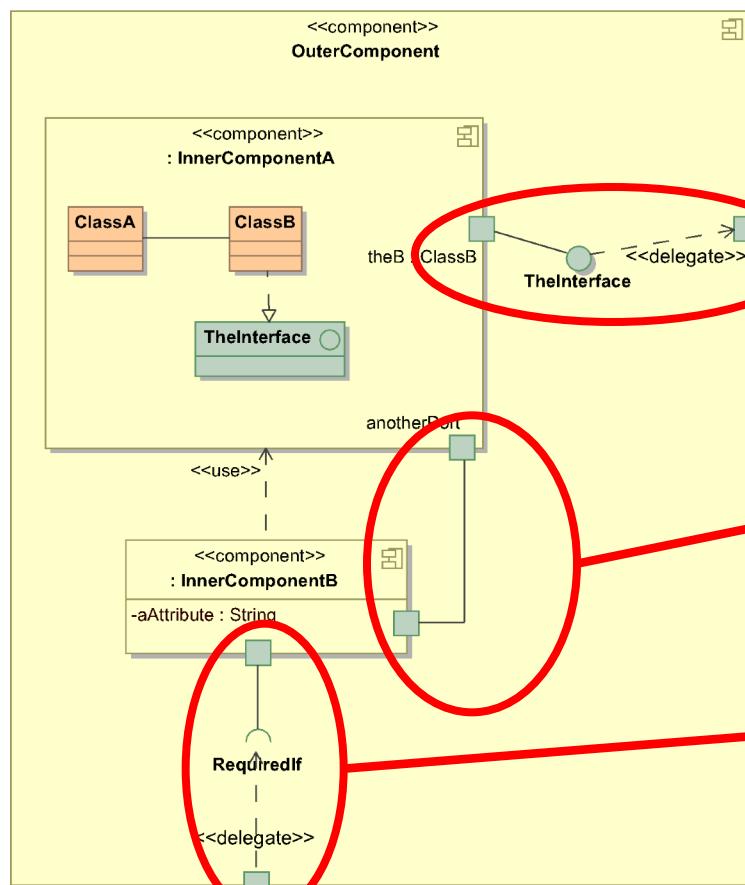
Strukturdiagramme

Anwendungsbeispiel



Grafik: Rupp, Queins, Zengler, UML2 Glasklar, Seite 213

Noch einmal im Überblick...



Komponentendiagramm

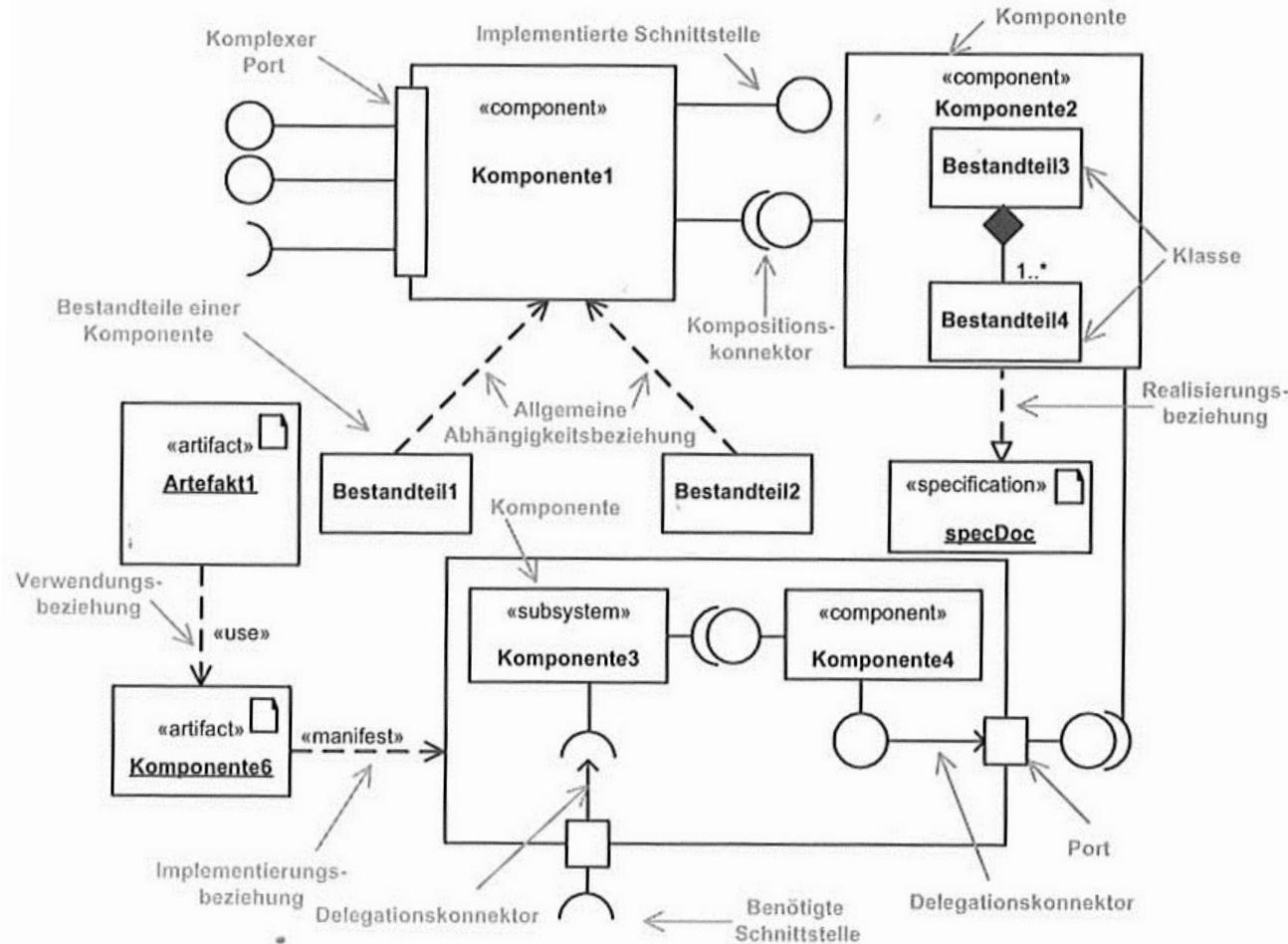
Strukturdiagramme

Delegation von
Schnittstellen
(Provided: Pfeilspitze
nach außen)

Konnektor
zwischen Ports

Delegation von
Schnittstellen (Required:
Pfeilspitze nach innen)

Komponentendiagramm

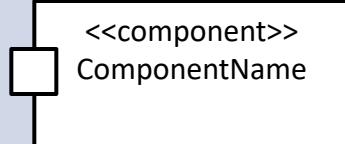
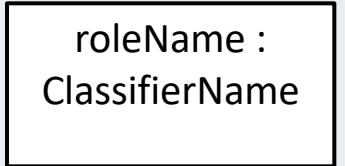
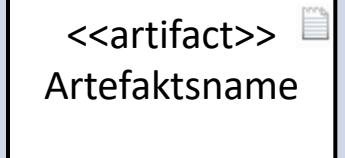


Grafik: Rupp, Queins, Zengler, UML2 Glasklar, Seite 212

Appendix: Auflistung der wichtigsten Elemente

Elementtyp	Symbole	Anmerkung
Komponente	<div style="border: 1px solid black; padding: 10px;"><<component>> ComponentName</div>	
Schnittstelle	<div style="border: 1px solid black; padding: 10px;"><<interface>> InterfaceName</div>	
Komponente Implementiert Schnittstelle	 <div style="border: 1px solid black; padding: 10px;"><<component>> ComponentName</div>	
Komponente verwendet Schnittstelle	 <div style="border: 1px solid black; padding: 10px;"><<component>> ComponentName</div>	

Appendix: Auflistung der wichtigsten Elemente

Elementtyp	Symbole	Anmerkung
Port		
Part		
Artefakt		
Konnektor		

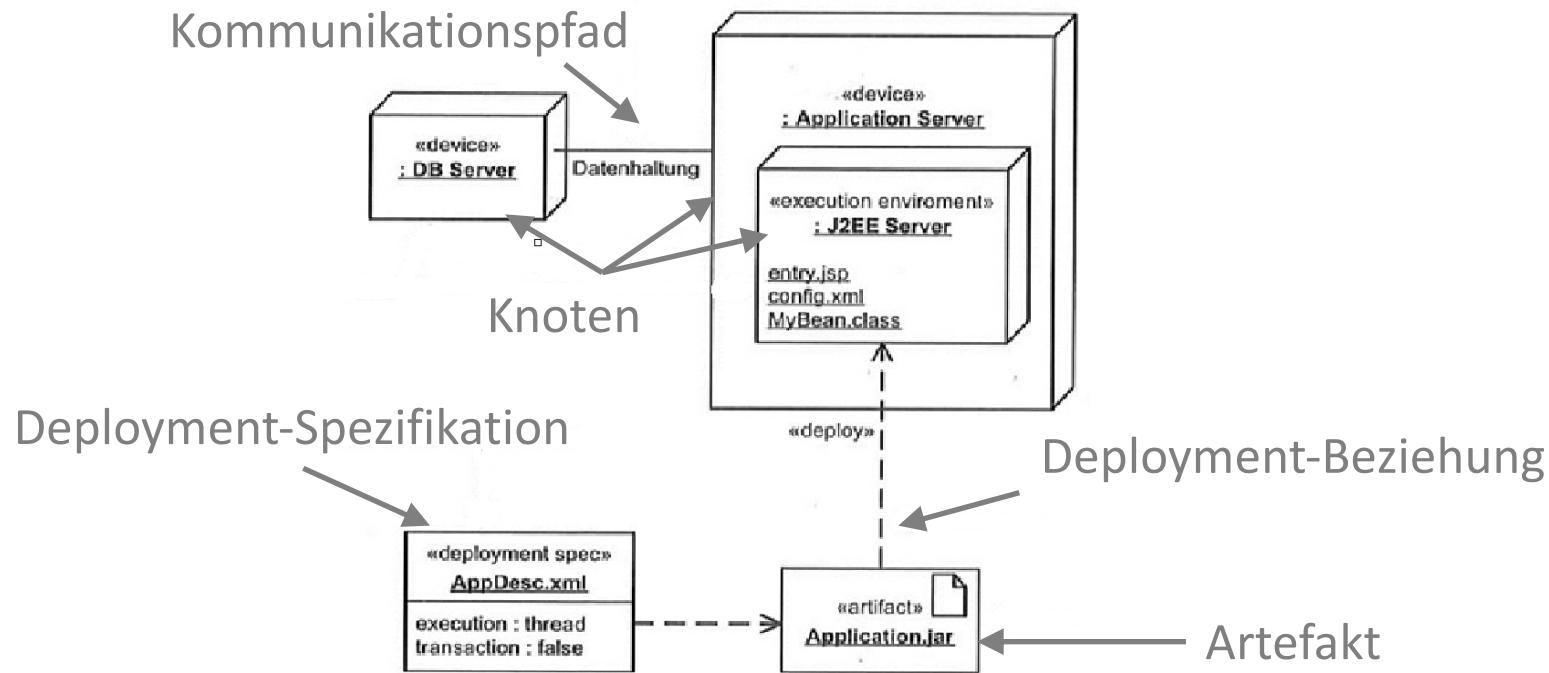
Inhalt

- **Beschreibungstechniken und Basiswissen**
 - Grundlagen der Objektorientierung (Wiederholung)
 - Übersicht zur UML
- **Strukturdiagramme**
 - Klassendiagramm
 - Paketdiagramm
 - Objektdiagramm
 - Kompositionssstrukturdiagramm
 - Komponentendiagramm
 - **Verteilungsdiagramm**
- **Verhaltensdiagramme**
 - Use-Case-Diagramm
 - Aktivitätsdiagramm
 - Zustandsautomat
- **Interaktionsdiagramme**
 - Sequenzdiagramm
 - Kommunikationsdiagramm
 - Timingdiagramm
 - Interaktionsübersichtsdiagramm
- **OCL (Object Constraint Language)**

Was ist Verteilungsdiagramm?

- Modellieren der Architektur verteilter System zur Laufzeit
- Zuordnung von Artefakten auf Hardware-Einheiten
- Darstellen der Kommunikationsverbindungen und Abhangigkeit zwischen den Knoten
- Parallel zu Komponentendiagramm eingesetzt
- Überwiegend in der Design-Phase erstellt

Verteilungsdiagramm



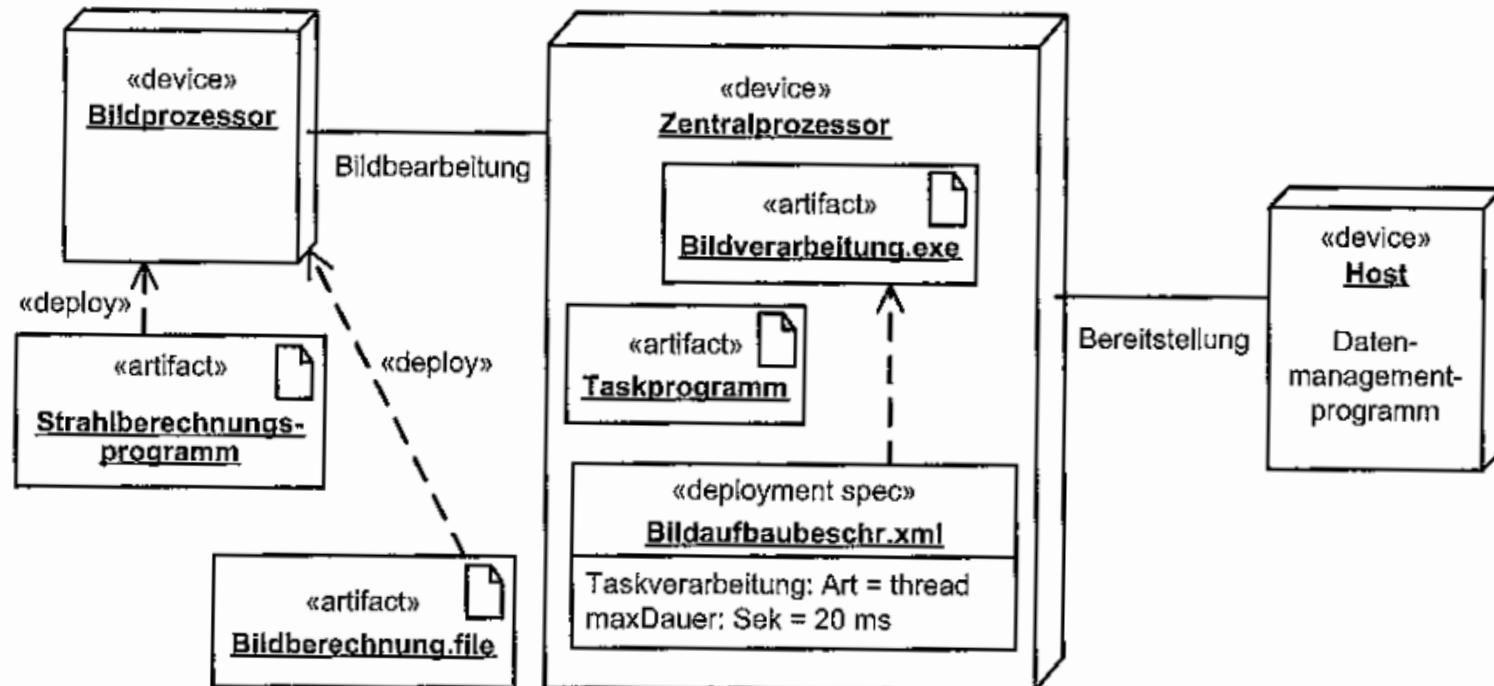
Grafik: Rupp, Queins, Zengler, UML2 Glasklar, Seite 224

Verteilungsdiagramm: Beziehung

- Kommunikationspfad
 - Ungerichteter
 - Gerichteter Kommunikationspfad
 - Benutzerdefinierte Stereotyp ist möglich
 - Verteilungsbeziehung
 - Verteilungsbeziehung (Deploy)
-
- The diagram illustrates two types of relationships in UML distribution diagrams. On the left, under 'Kommunikationspfad', there are three entries: 'Ungerichteter' represented by a simple horizontal line, 'Gerichteter Kommunikationspfad' represented by a horizontal line with an arrowhead pointing to the right, and 'Benutzerdefinierte Stereotyp ist möglich' represented by a horizontal line with the stereotype '<<WLAN>>' written above it. On the right, under 'Verteilungsbeziehung', there is one entry: 'Verteilungsbeziehung (Deploy)' represented by a dashed horizontal line ending in a right-pointing arrowhead, with the stereotype '<<Deploy>>' written above it.

Grafik: Rupp, Queins, Zengler, UML2 Glasklar, Seite 224

Verteilungsdiagramm - Anwendungsbeispiel



Grafik: Rupp, Queins, Zengler, UML2 Glasklar, Seite 225

Verteilungsdiagramm

Strukturdiagramme

Elemente in einem Verteilungsdiagramm

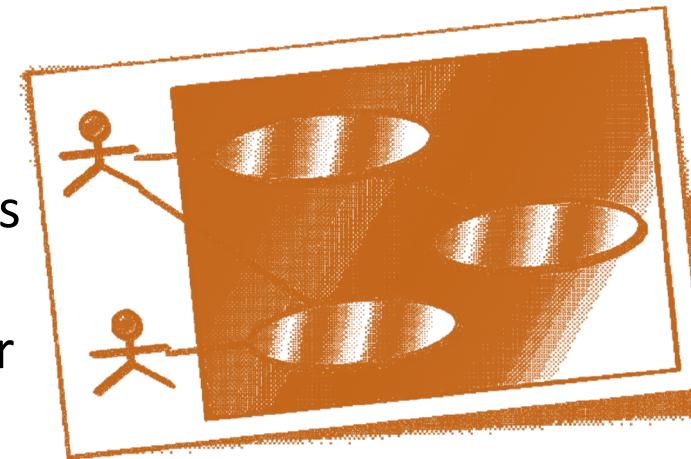
- Knoten
- Kommunikationspfad zwischen den Knoten
- Komponenten
- Artefakte
- Deployment-Spezifikationen (z.B. Konfigurationsskripte)
- Deployment-Beziehungen (Wo wird welches Artefakt „installiert“)

Inhalt

- **Beschreibungstechniken und Basiswissen**
 - Grundlagen der Objektorientierung (Wiederholung)
 - Übersicht zur UML
- **Strukturdiagramme**
 - Klassendiagramm
 - Paketdiagramm
 - Objektdiagramm
 - Kompositionssstrukturdiagramm
 - Komponentendiagramm
 - Verteilungsdiagramm
- **Verhaltensdiagramme**
 - **Use-Case-Diagramm**
 - Aktivitätsdiagramm
 - Zustandsautomat
- **Interaktionsdiagramme**
 - Sequenzdiagramm
 - Kommunikationsdiagramm
 - Timingdiagramm
 - Interaktionsübersichtsdiagramm
- **OCL (Object Constraint Language)**

Ein Use-Case – was ist das?

- „auf deutsch“: ein Anwendungsfall
- Es gibt
 - Use-Case-Diagramme
 - Textuelle Use-Case-Beschreibungen
 - z.B. Tabellen
- Ein Use-Case-Diagramm oder Anwendungsfall-Diagramm zeigt das *externe Verhalten* eines Systems *aus der Sicht der Nutzer*, indem es Use-Cases und deren Beziehungen zueinander darstellt.

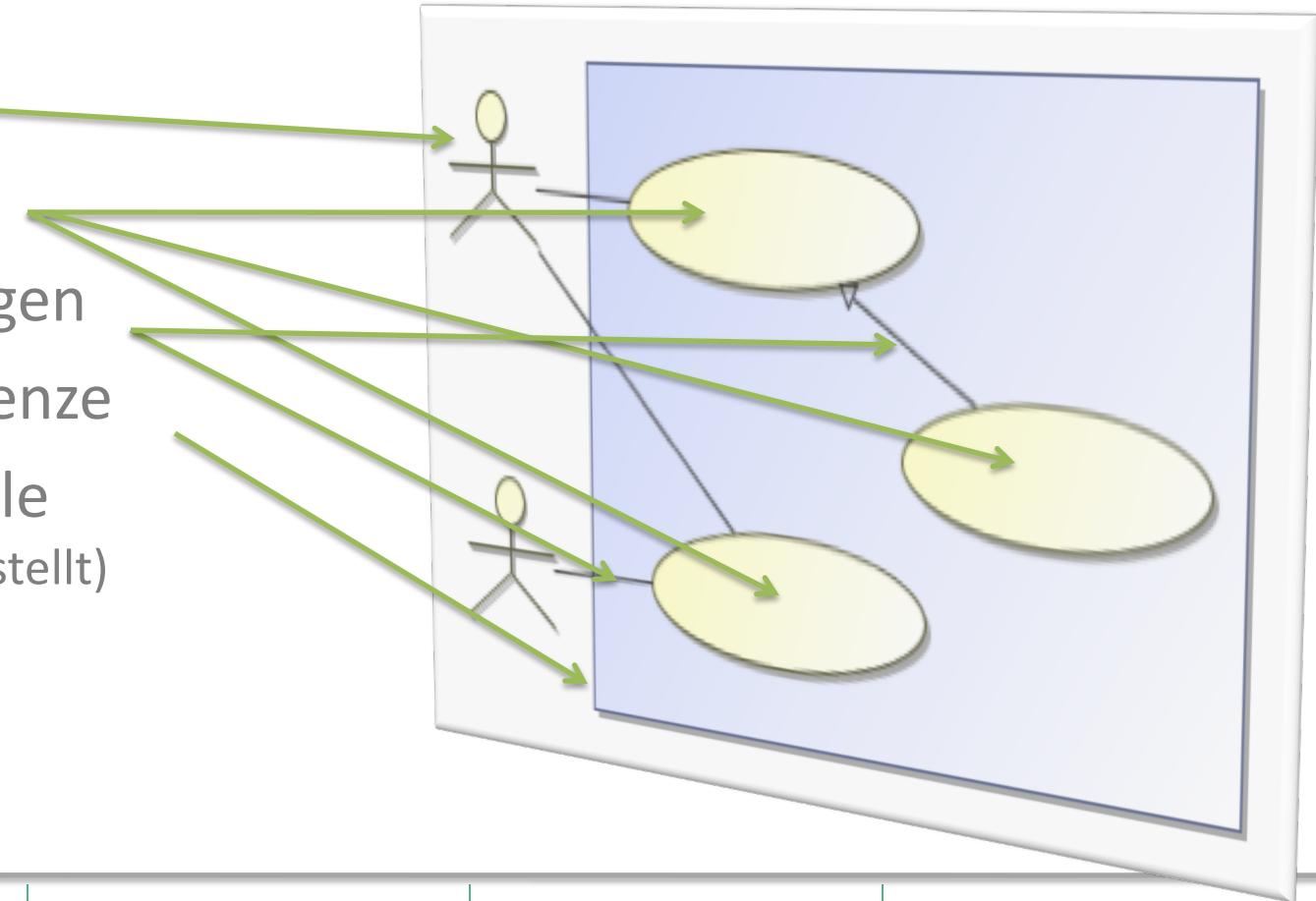


Motivation

- einfacher Einstieg in die Analyse
- jeder Anwendungsfall zeigt, wie der Nutzer mit dem System interagiert, um ein bestimmtes Ziel zu erreichen
- „Was soll mein geplantes System eigentlich leisten?“
 - hilft, nicht zu sehr ins Detail abzurutschen
 - was statt wie
 - „Beschreibung des funktionalen Verhaltens“
→Funktionsumfang

Wichtige Elemente

- Akteur
- Use-Case
- Beziehungen
- Systemgrenze
- Systemteile
(nicht dargestellt)



Wichtige Elemente - Beziehungen



„einfache“ Assoziation



Import-beziehung:
Zwingender Unterprogrammaufruf

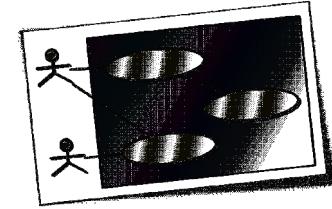


Erweiterungs-beziehung:
Optionaler Unterprogrammaufruf



Generalisierungs-beziehung

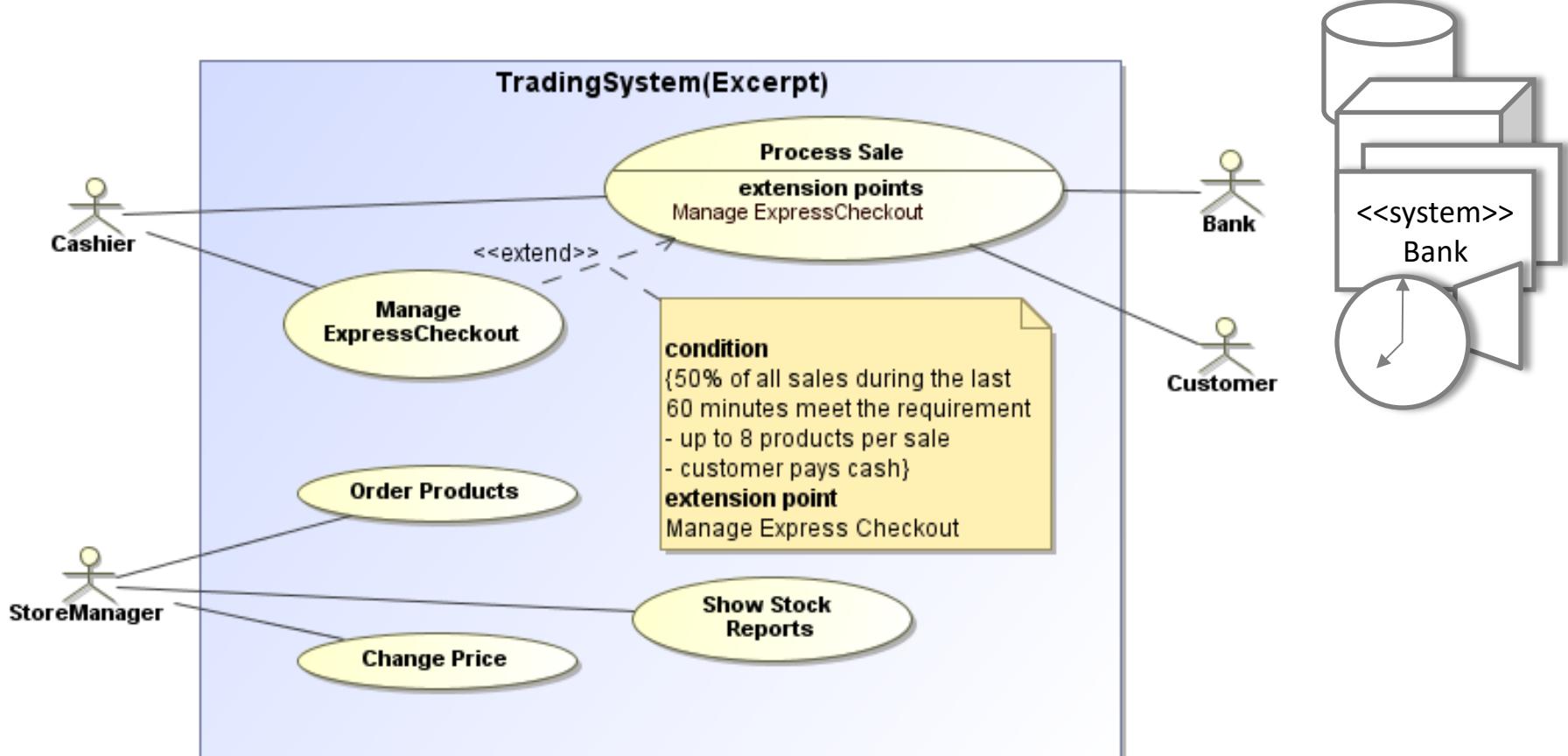
Wichtige Elemente (pro Use Case)



Nummer, Name

Kurzbeschreibung	ein kurzer erklärender (strukturierter) Satz zur Übersicht
Akteure	Personen (Rollen) oder externe Systeme, die aktiv mit dem System interagieren oder einen Nutzen von dem Anwendungsfall haben. Ein Anwendungsfall kann mit mehreren Akteuren verbunden sein.
Kategorie	muss, soll, oder kann der Anwendungsfall realisiert werden?
Auslöser	ein Akteur oder eine Funktion [...] die den Ablauf startet.
Vorbedingung	eine Bedingung, die erfüllt sein muss, damit der Ablauf gestartet wird – beim Auto muss zum Beispiel der Schlüssel im Schloss stecken, damit man starten kann.
Eingabe	für den Ablauf benötigte Informationen
Ausgabe	Ergebnis des Ablaufs
Nachbedingung	eine Bedingung, die erfüllt sein muss, um den Anwendungsfall zu beenden.
Ablauf	beschreibt den Standardablauf – was soll passieren, wenn der Anwendungsfall gestartet ist? Achtung, hier werden keine Ausnahmen behandelt!

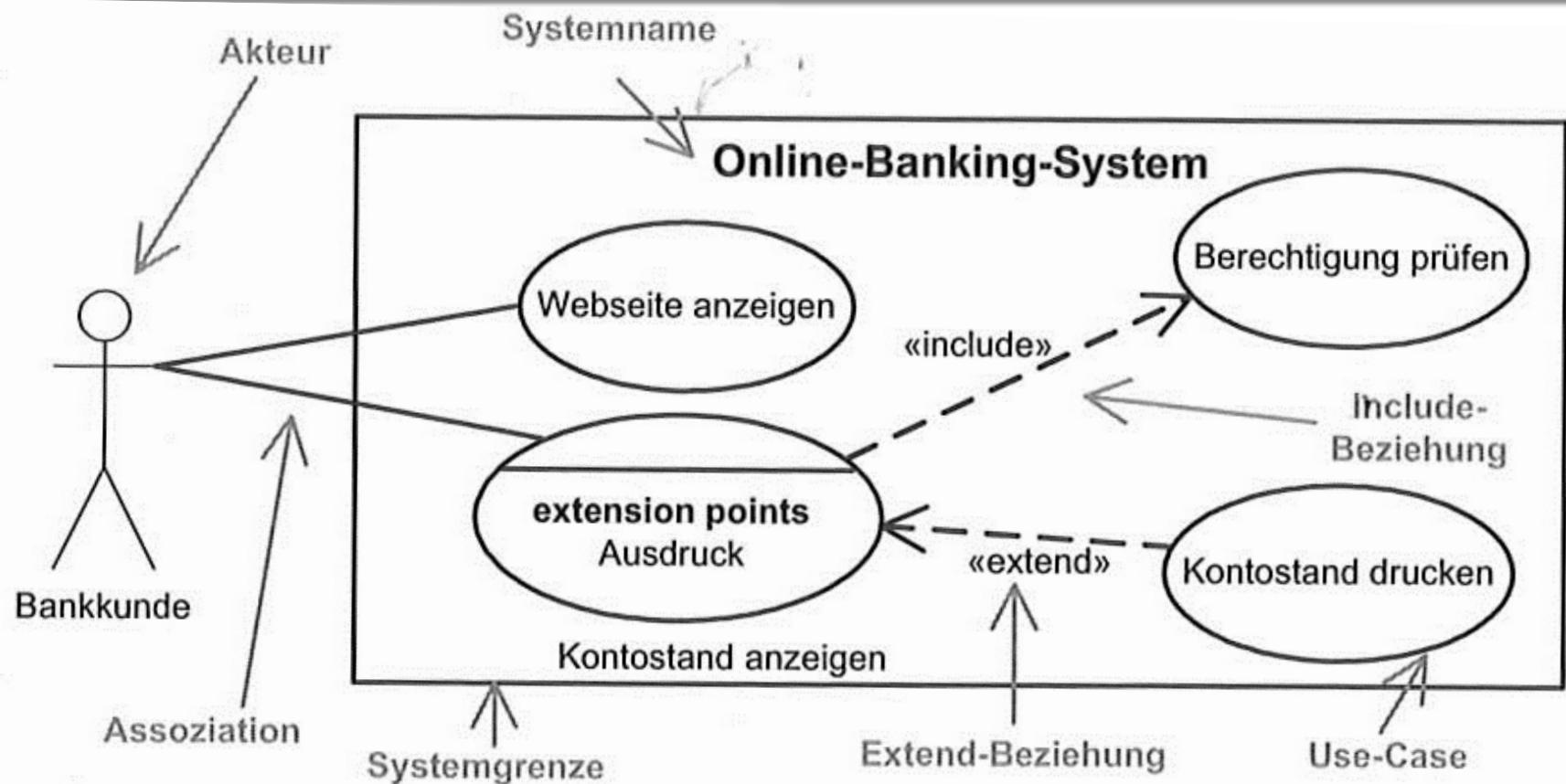
Beispiel



Use-Case-Diagramm

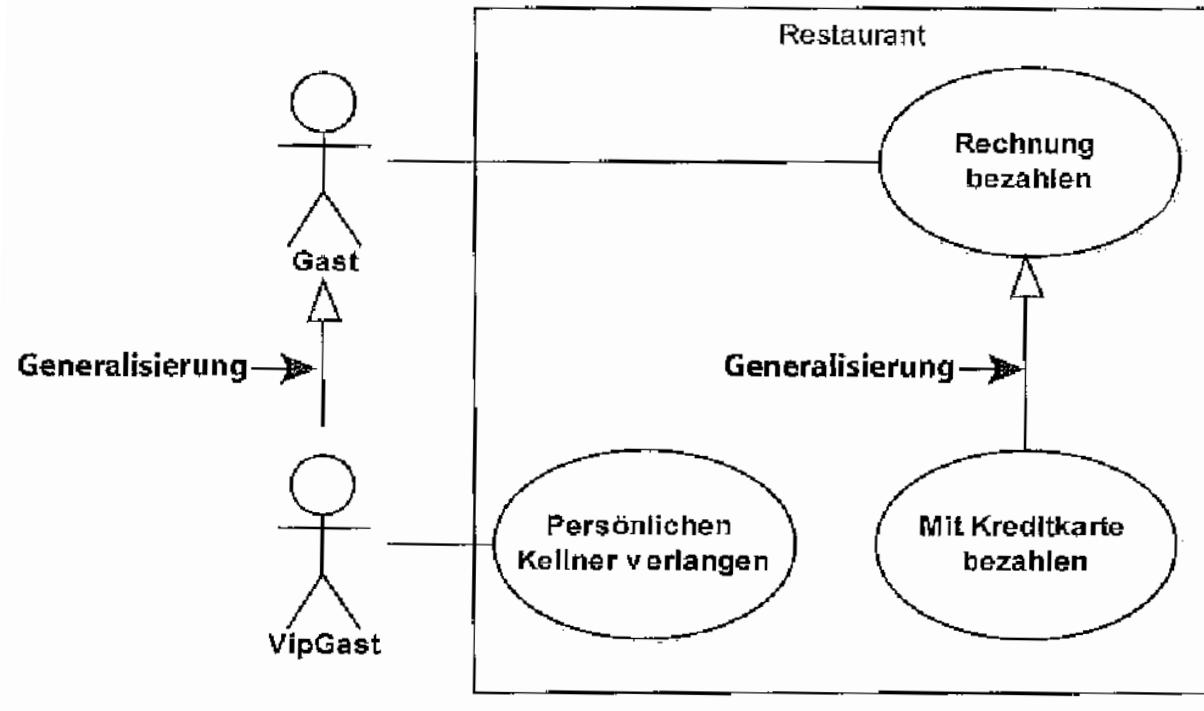
Verhaltensdiagramme

Use-Case Diagramme



Grafik: Rupp, Queins, Zengler, UML2 Glasklar, Seite 239

Use-Case Diagramme (2)



Grafik: Rupp, Queins, Zengler, UML2 Glasklar, Seite 252

Vorsicht, Falle!

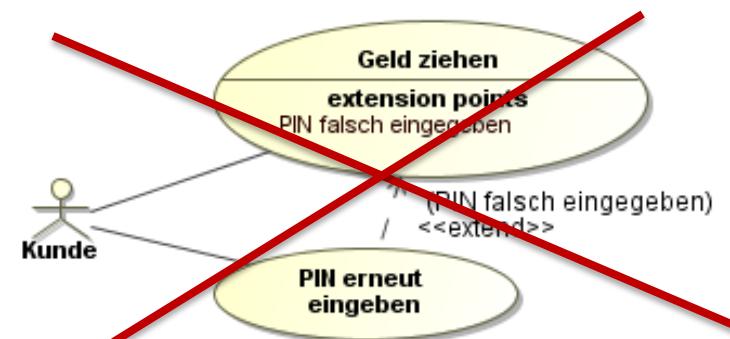
- Falle 1: Alternative Abläufe als Erweiterungen darstellen
- Falle 2: Geschäftsprozesse darstellen
 - Business Use-Cases
- Falle 3: Kommunizierende Use-Cases
- Falle 4: Funktionale Dekomposition
- Falle 5: schlechtes Layout

Vorsicht, Falle!

Falle 1: Alternative Abläufe als Erweiterungen darstellen

- Kontext
 - Ein alternativer Ablauf eines Use-Cases wird als einzelner Use Case in einer Erweiterung des Use-Cases modelliert
- Problem
 - Der Ursprüngliche Use-Case ist nun unvollständig
 - Das Use-Case-Modell ist nun komplexer
- Lösung
 - Die Erweiterung entfernen: Der Ablauf der Erweiterung wird als alternativer Ablauf in den ursprünglichen Ablauf integriert.
- Beispiel

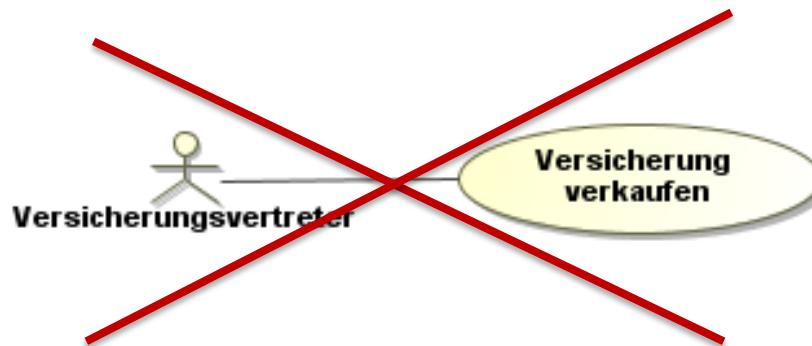
Der Basis-Use-Case ist ohne die Erweiterung unvollständig



Vorsicht, Falle!

Falle 2: Geschäftsprozesse darstellen

- Kontext
 - Ein Geschäftsprozess wird als System-Use-Case dargestellt.
- Problem
 - Es gibt Unterbrechungen im Use-Case-Ablauf, die durch die Beteiligung von mehreren Rollen oder durch Tage- und Wochenlange Pausen (Übergaben etc.) entstehen können.
- Lösung
 - Individuelle System-Abläufe identifizieren und auch als solche modellieren.
- Beispiel



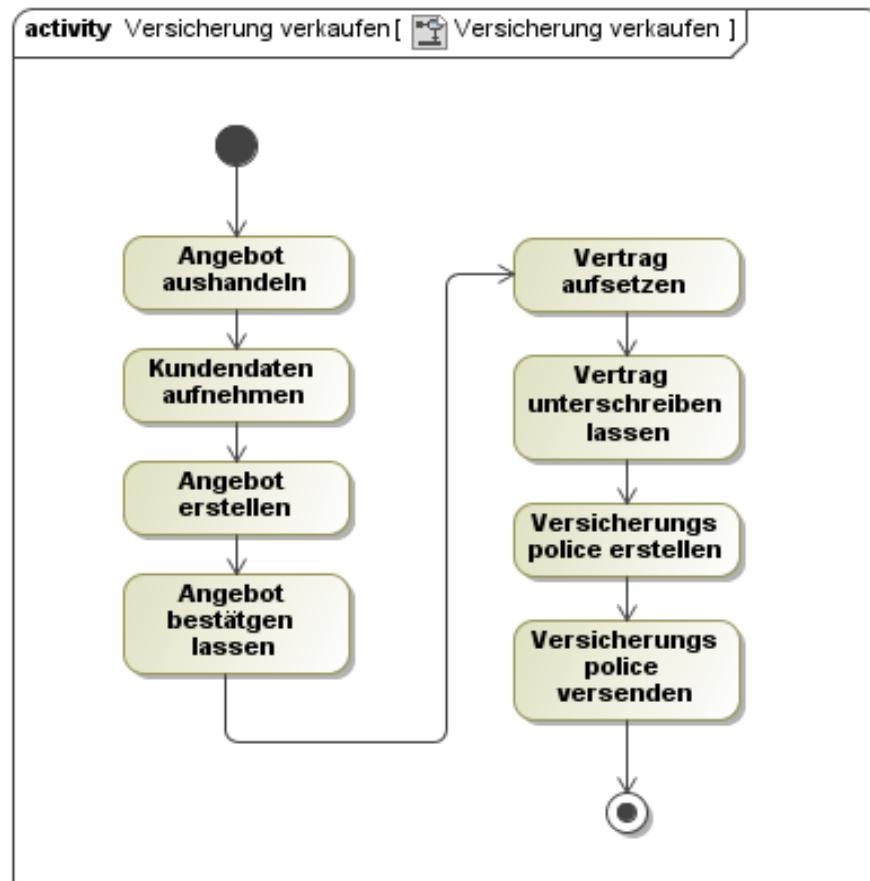
Use-Case-Diagramm

Verhaltensdiagramme

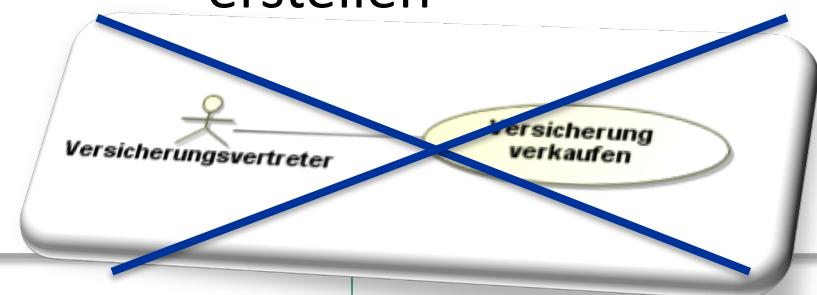
Dieser Use-Case beschreibt mehrere Benutzungen (nicht zusammen-hängend,) mit dem System.
1 Use-Case = 1 Benutzung

Vorsicht, Falle!

Falle 2: Geschäftsprozesse darstellen (Beispiel)



- Was sind Interaktionen mit einem System:
 - Kundendaten aufnehmen
 - Angebot erstellen
 - Vertrag aufsetzen
 - Versicherungspolice erstellen



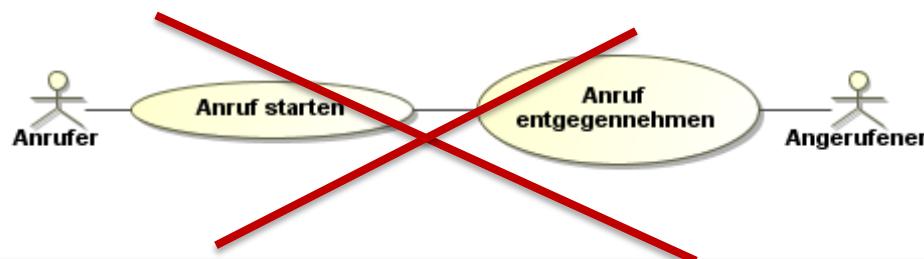
Use-Case-Diagramm

Verhaltensdiagramme

Vorsicht, Falle!

Falle 3: Kommunizierende Use-Cases

- Kontext
 - Wenn man zwei Use-Cases mit einer Assoziation dazwischen modelliert, impliziert dies, dass die Use-Cases miteinander kommunizieren.
- Problem
 - Ein Use-Case steht für einen Ablauf: Abläufe können nicht miteinander kommunizieren.
 - Unvollständige Abläufe werden modelliert (nur die beiden verbundenen Abläufe zusammen scheinen einen vollständigen Ablauf zu ergeben)
- Lösung
 - Fassen Sie die beiden Abläufe, die Sie kommunizieren lassen würden, zu einem Use-Case zusammen.
 - Alternative: Sind die Use-Cases in sich vollständig, überlegen Sie, ob eine include- oder extend-Beziehung besteht.
- Beispiel



Die Abläufe beider Use-Cases ergeben einen vollständigen Ablauf. Alleine sind sie nicht vollständig.

Vorsicht, Falle!

Falle 4: Funktionale Dekomposition

- Kontext
 - Ein großer Use-Case wird durch include-Beziehungen in mehrere Use-Cases zerlegt, die jeweils als eine Unterfunktion des großen Use-Cases modelliert sind.
- Problem
 - Das Use-Case-Modell wird unnötig komplex.
 - Ein komplexes Modell erhöht „Wartungskosten“ und ist Fehleranfälliger.
- Lösung
 - Verwenden Sie include-Beziehungen sparsam. Fügen Sie wenn möglich die Abläufe der inkludierten Use-Cases dem übergeordneten Use-Case hinzu.
- Beispiel



Use-Case-Diagramm

Verhaltensdiagramme

Die inkludierten Use-Cases beschreiben weder die Benutzung des Systems noch wiederverwendbare Sequenzen.

Vorsicht, Falle!

Falle 5: schlechtes Layout

- Kontext
 - Ein Use-Case-Modell ist nicht intuitiv aufgebaut
- Problem
 - Diagramme werden unverständlich.
 - Es werden falsche Annahmen in solche Diagramme rein interpretiert.
- Lösung
 - Zeichnen Sie Modelle neu:
 - Beachten Sie die Leserichtung
 - Reduzieren Sie die Menge der Use-Cases in einem Diagramm
 - Vermeiden Sie sich überschneidende Kanten
 - Verwenden Sie aussagekräftige Namen
- Beispiel



Use-Case-Diagramm

Verhaltensdiagramme

Diagramme werden in der westlichen Welt von links nach rechts gelesen.

Hier sieht das nun so aus, als ob der Angerufene den Anruf startet.

Inhalt

- **Beschreibungstechniken und Basiswissen**
 - Grundlagen der Objektorientierung (Wiederholung)
 - Übersicht zur UML
- **Strukturdiagramme**
 - Klassendiagramm
 - Paketdiagramm
 - Objektdiagramm
 - Kompositionssstrukturdiagramm
 - Komponentendiagramm
 - Verteilungsdiagramm
- **Verhaltensdiagramme**
 - Use-Case-Diagramm
 - **Aktivitätsdiagramm**
 - Zustandsautomat
- **Interaktionsdiagramme**
 - Sequenzdiagramm
 - Kommunikationsdiagramm
 - Timingdiagramm
 - Interaktionsübersichtsdiagramm
- **OCL (Object Constraint Language)**

Aktivitätsdiagramm – was ist das?

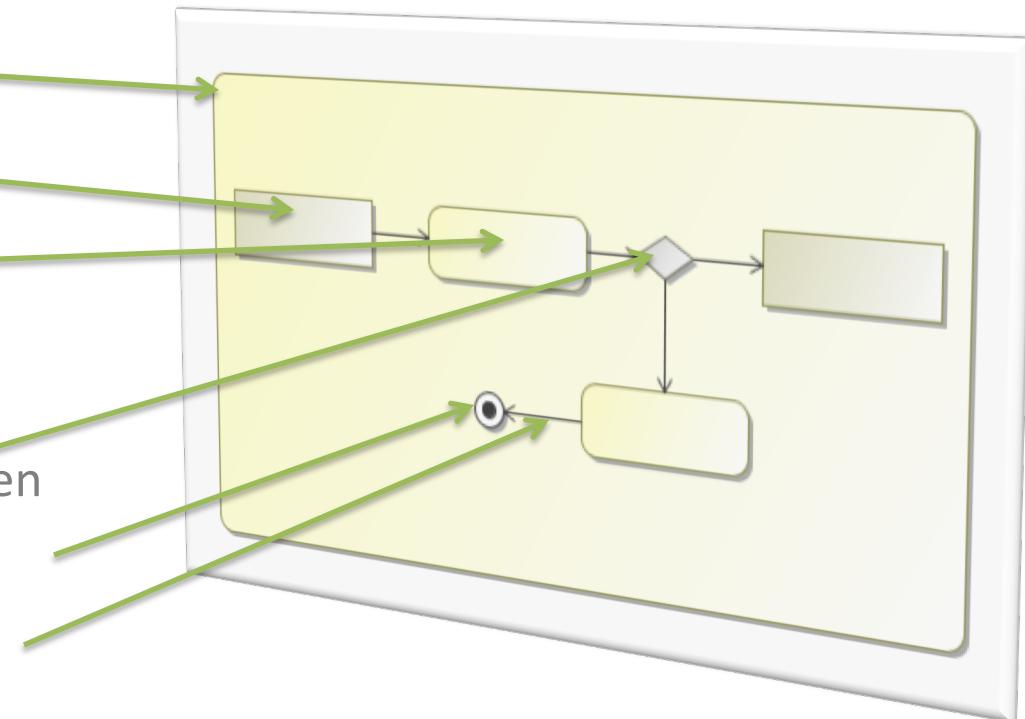
- Darstellung von Abläufen
 - Use-Case-Ablauf
 - Ablauf einer Operation
 - Ablauf eines Geschäftsvorfalls
- Ein Aktivitätsdiagramm
 - stellt analog zu einer Straßenkarte den Rahmen und die Regeln von Verhaltensabläufen auf detailliertem Niveau dar
 - umfasst Start- und Endpunkte (Auf- und Abfahrten), Verzweigungen (Kreuzungen, Kreisverkehre) bestimmte Bedingungen (Einbahnstraße, Gewichtsbeschränkungen) und vieles mehr
 - stellt nicht die eigentlichen Abläufe (Straßenverkehr, der jeden Tag variiert) dar, sondern die Regeln, die alle möglichen Abläufe beschreiben

Motivation

- Darstellung von Abläufen
 - Geschäftsprozesse
 - (Standard)Abläufe von Use-Cases
- „Wie realisiert mein System ein bestimmtes Verhalten?“
 - eine vom System zu bewältigende Aufgabe (oder eine Aufgabe, bei der das System beteiligt ist), wird in Einzelschritte zerlegt

Wichtige Elemente

- Aktivitäten
- Objektknoten
- Aktionen
- Kontrollelemente zur Ablaufsteuerung
 - z.B. Start- und Endknoten
- verbindende Kanten



Wichtige Elemente

Kontrollelemente (1)

- Kontrollelemente zur Ablaufsteuerung
 - befinden sich zwischen Aktionen oder Objektknoten
 - Flusssteuerung des Ablaufs
 - Geben Entscheidungsregeln oder Bedingungen vor, wann und in welcher Reihenfolge einzelne Aktionen durchgeführt oder Objektknoten verändert werden
- Beispiele:
 - Start-/Endeknoten
 - Verzweigungs-/Verbindungsknoten
 - Parallelisierungs-/Synchronisationsknoten

Wichtige Elemente

Kontrollelemente (2)

- Kontrollelemente zur Ablaufsteuerung



Start-
knoten

- Ein **Startknoten** markiert den Startpunkt eines Ablaufs bei Aktivierung einer Aktivität.

- » Eine Aktivität darf beliebig viele Startknoten besitzen
- » Eine Aktivität muss keinen Startknoten besitzen

- Ein **Endknoten für Aktivitäten** beendet die gesamte Aktivität.

- » Beendet sofort die gesamte Aktivität
- » Parallel ausgeführte Aktionen werden ebenfalls beendet



Endknoten
für
Aktivitäten

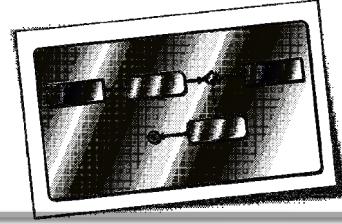


Endknoten
für Kontroll-
flüsse

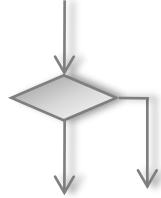
- Ein **Endknoten für Kontrollflüsse** beendet nur einen einzelnen Ablauf.

- » Markiert nur das Ende eines Ablaufs
- » Ist nur dann mit der Beendigung der gesamten Aktivität gleichzusetzen, wenn die Abläufe vorher nicht parallelisiert wurden (→ es gibt nur diesen einen Ablauf)

Wichtige Elemente Kontrollelemente (3)

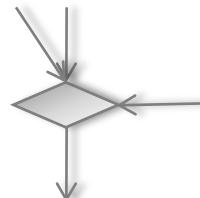


- Kontrollelemente zur Ablaufsteuerung



Verzweigungs-
knoten

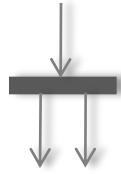
- Ein **Verzweigungsknoten** spaltet eine Kante in mehrere Alternativen auf.
 - » Besonders interessant, wenn ein Ablauf in einer Aktivität von bestimmten Bedingungen abhängig ist.



Verbindungs-
knoten

- Ein **Verbindungsknoten** ist das Gegenstück zu einem Verzweigungsknoten, er führt Kanten zusammen.
 - » Beschreibt ein „ODER“

• Kontrollelemente zur Ablaufsteuerung

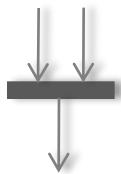


Parallelisierungs-knoten

- Ein **Parallelisierungsknoten** teilt den Ablauf, der über genau eine eingehende Kante geführt wird, in parallele Abläufe (mehrere ausgehende Kanten) auf.

» ermöglicht nebenläufiges (paralleles) Abarbeiten von Aktionen

» der Kontroll- oder Objektfluss muss anschließend wieder zusammengeführt (Synchronisationsknoten) oder einzelne „Stränge“ beendet werden



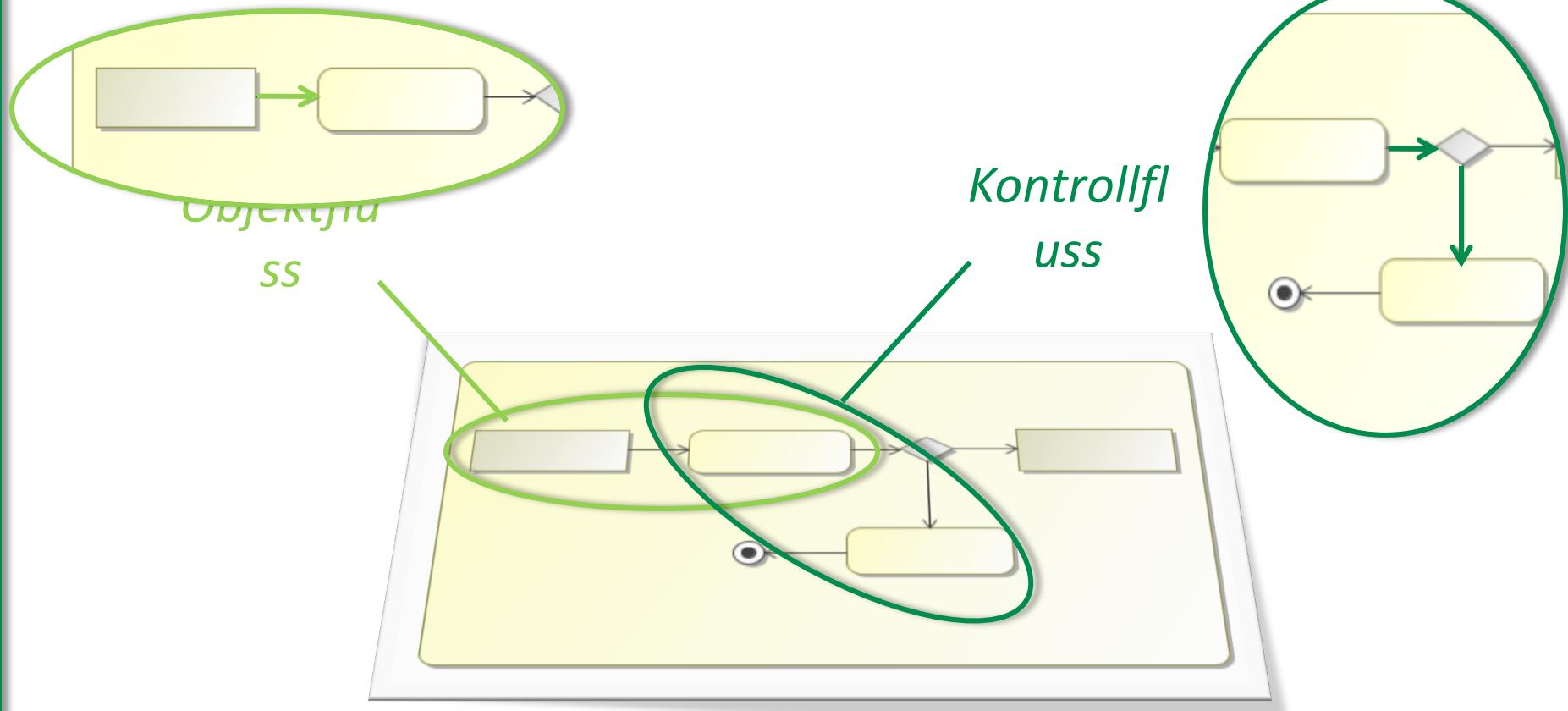
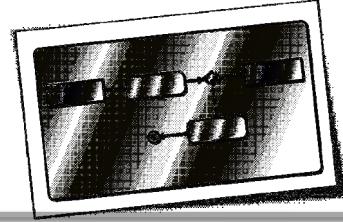
Synchronisations-knoten

- Ein **Synchronisationsknoten** führt parallele Abläufe

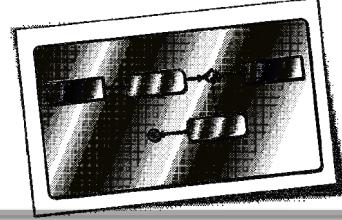
» Der Ablauf wird erst dann fortgesetzt, wenn beide vorherigen Aktionen (...) durchgeführt wurden

» Beschreibt ein „UND“

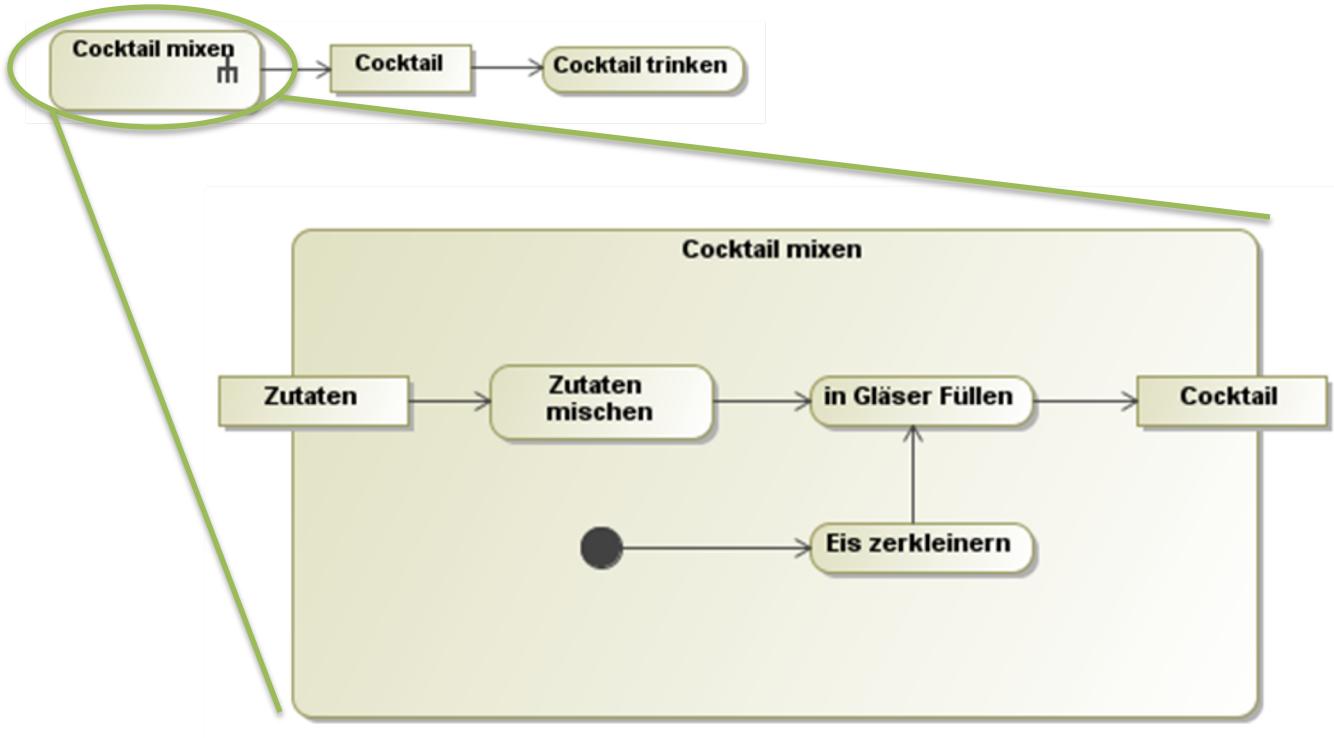
Wichtige Elemente: Kanten



Strukturierung durch Aktivitätsaufruf



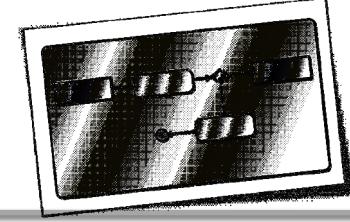
- Bilden von Hierarchien



Aktivitätsdiagramm

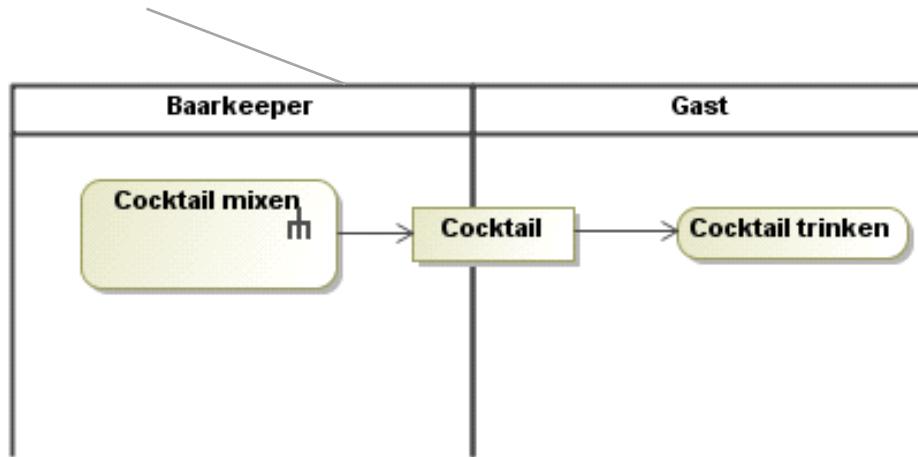
Verhaltensdiagramme

Strukturierung durch Aktivitätsbereiche



- Zuordnen von Aktionen

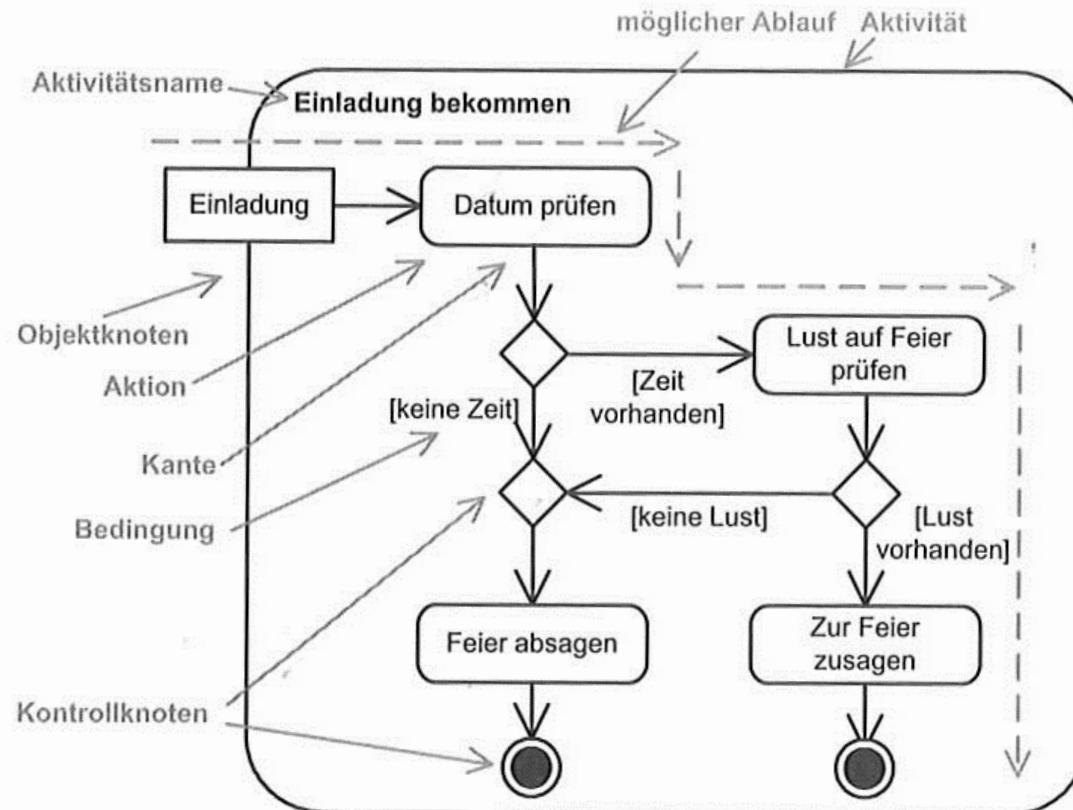
Aktivitätsbereich



Aktivitätsdiagramm

Verhaltensdiagramme

Aktivitätsdiagramm

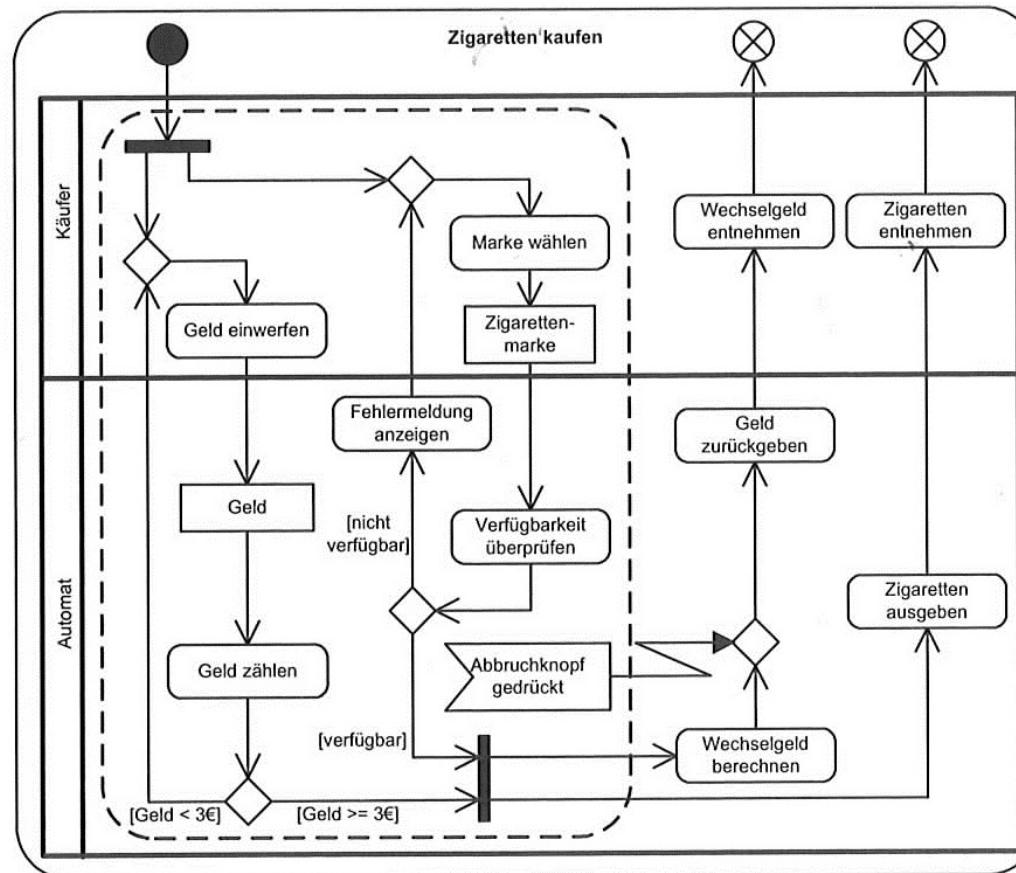


Grafik: Rupp, Queins, Zengler, UML2 Glasklar, Seite 261

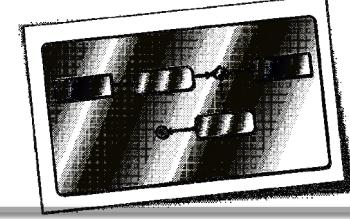
Aktivitätsdiagramm

Verhaltensdiagramme

Aktivitätsdiagramm - Anwendungsbeispiel



Angehängt: Objektfluss



- Pin-Notation



- Objektknoten und Parameterübergabe

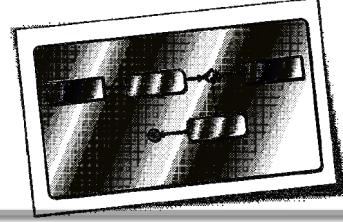
- Objektfluss mit Parametern mit gleichem Namen und gleichem Typ



- Objektfluss mit voneinander verschiedenen Eingangs- und Ausgangsparametern



Angehängt: Besondere Aktionen



- Signale und Ereignisse
 - Bei einer **SendSignalAction** (Signal-Aktion Senden) wird von dem Signalsender ein Signal aus Eingabedaten erstellt und versendet.
 - Sobald das Signal versendet wurde, gilt diese Aktion als beendet und der Kontrollfluss der Aktivität läuft weiter
 - Eine **AcceptEventAction** (Ereignis-Aktion Akzeptieren) ist das Gegenstück zur SendSignalAction.
 - Erreicht der Ablauf einen Ereignisempfänger, bleibt er so lange in dieser Aktion, bis das erwartete Ereignis eintritt. Anschließend wird die Abarbeitung fortgesetzt

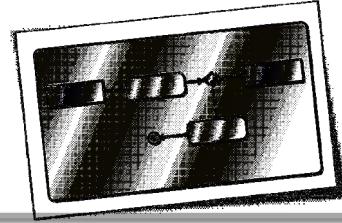


sende Signal

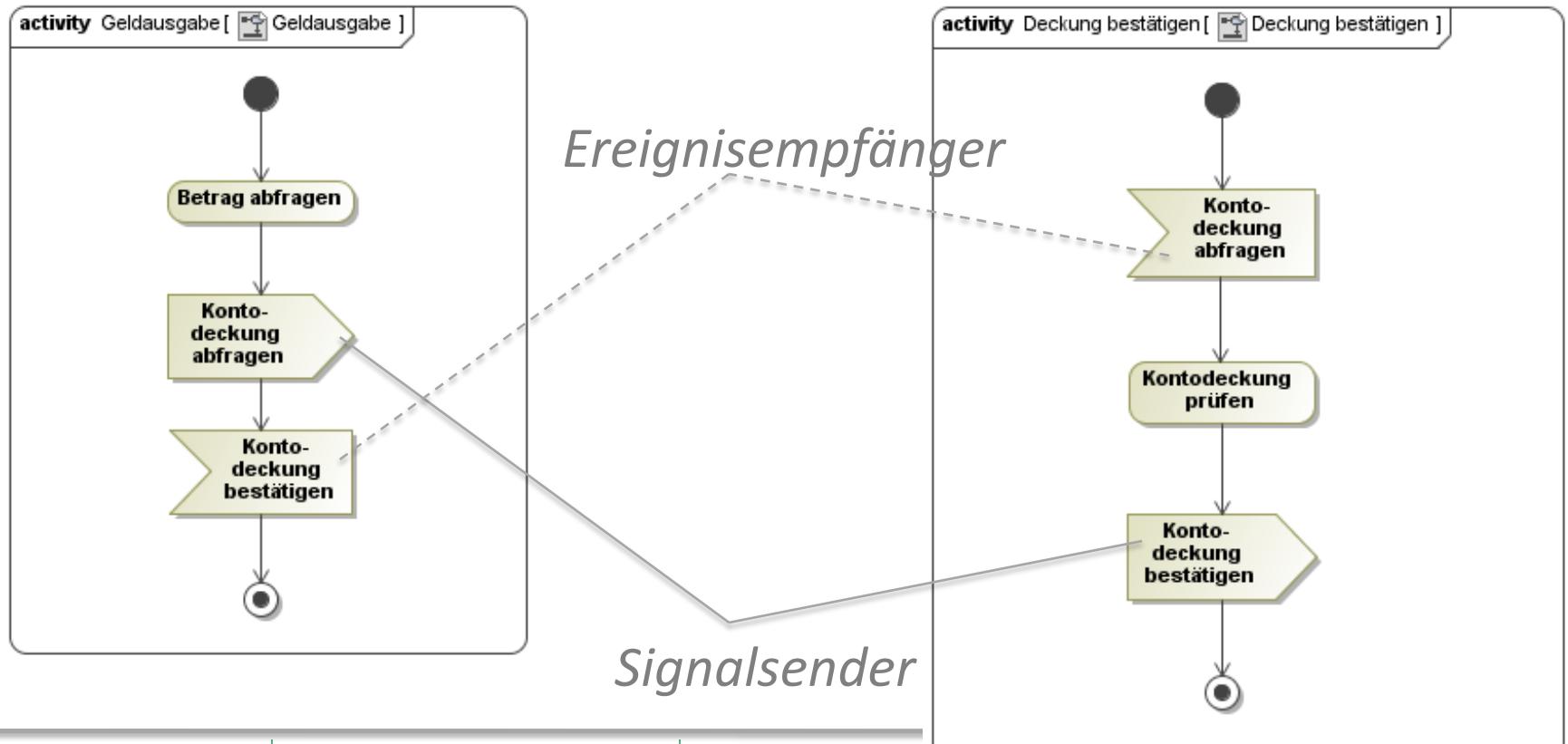


empfange
Ereignis

Angehängt: Besondere Aktionen



- Signale und Ereignisse (Beispiel)



Inhalt

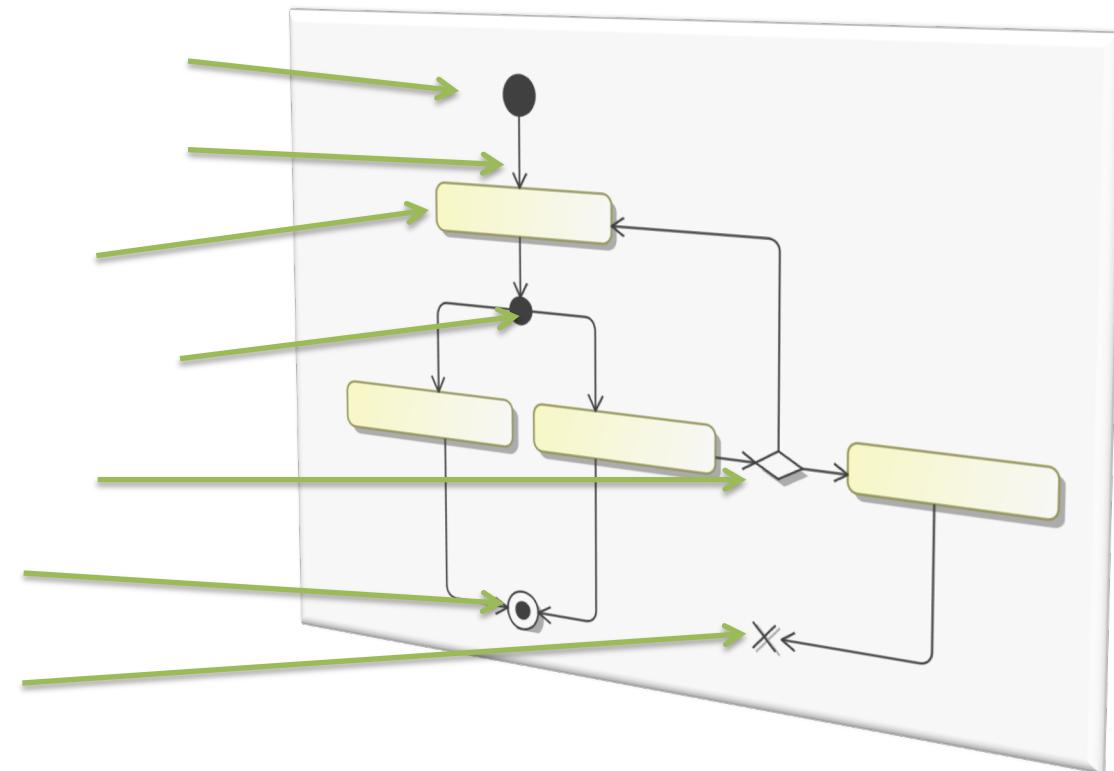
- **Beschreibungstechniken und Basiswissen**
 - Grundlagen der Objektorientierung (Wiederholung)
 - Übersicht zur UML
- **Strukturdiagramme**
 - Klassendiagramm
 - Paketdiagramm
 - Objektdiagramm
 - Kompositionssstrukturdiagramm
 - Komponentendiagramm
 - Verteilungsdiagramm
- **Verhaltensdiagramme**
 - Use-Case-Diagramm
 - Aktivitätsdiagramm
 - **Zustandsautomat**
- **Interaktionsdiagramme**
 - Sequenzdiagramm
 - Kommunikationsdiagramm
 - Timingdiagramm
 - Interaktionsübersichtsdiagramm
- **OCL (Object Constraint Language)**

Zustandsdiagramm

- Die Modellierung eines Systems von
 - Zustände (ein bestimmte Zeitpunkt)
 - Zustandsübergänge
 - Ereignisse
 - Aktionen
- Die Modellierung des Verhaltens von
 - Benutzungsoberflächen
 - Kommunikationsprotokollen
 - etc.

Wichtige Elemente

- Startpunkt
- Trigger
- Zustand
- Kreuzungspunkte
- Endscheidung
- Endzustand
- Terminator



Wichtige Elemente

Zustand
Entry / Verhalten
Exit / Verhalten
Do / Verhalten
Trigger [Guard] / Verhalten
Trigger [Guard] / defer

-Syntax des Zustand:
Event [Guard] / Verzögerung des Events



Startzustand



Endzustand



Terminator



Eintrittspunkt



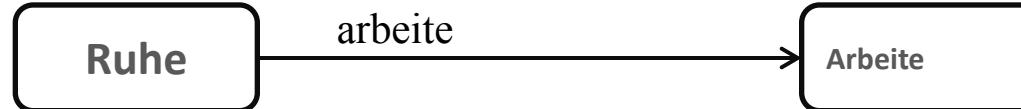
Austrittspunkt

- Startzustände als Erstes aktiv Zustand.
- Terminator für „erfolgreichen“ Durchlaufen oder Abbrauch der Zustandsautomatenausführung
- Ein- und Austrittspunkt geben mehrere Möglichkeit zum Betreten oder Verlassen der Zustände.

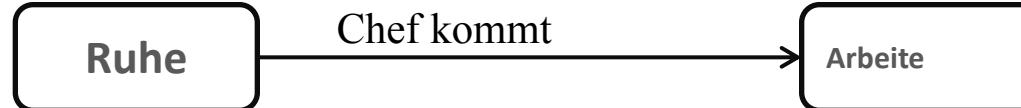
Wichtige Elemente Transition

- Event

CallEvent



SignalEvent



ChangeEvent



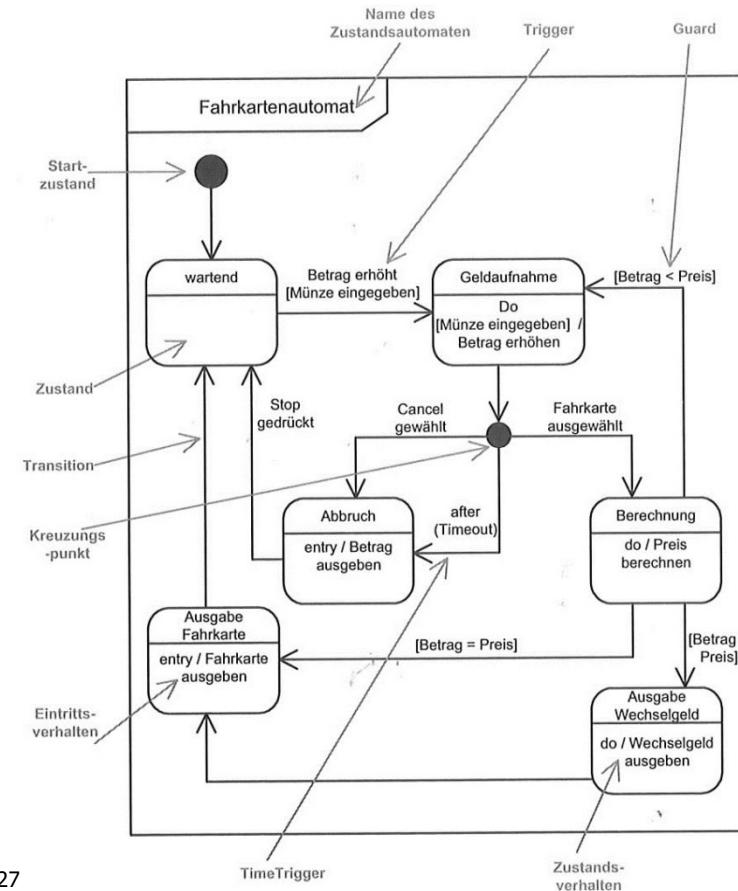
TimeEvent



- Guard
- Effekt



Zustandsautomat

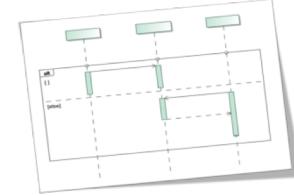


Grafik: Rupp, Queins, Zengler, UML2 Glasklar, Seite 327

Inhalt

- **Beschreibungstechniken und Basiswissen**
 - Grundlagen der Objektorientierung (Wiederholung)
 - Übersicht zur UML
- **Strukturdiagramme**
 - Klassendiagramm
 - Paketdiagramm
 - Objektdiagramm
 - Kompositionssstrukturdiagramm
 - Komponentendiagramm
 - Verteilungsdiagramm
- **Verhaltensdiagramme**
 - Use-Case-Diagramm
 - Aktivitätsdiagramm
 - Zustandsautomat
- **Interaktionsdiagramme**
 - **Sequenzdiagramm**
 - Kommunikationsdiagramm
 - Timingdiagramm
 - Interaktionsübersichtsdiagramm
- **OCL (Object Constraint Language)**

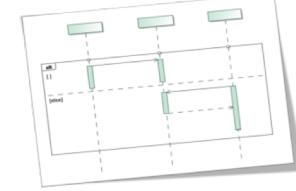
Sequenzdiagramm – was ist das?



- Darstellung von
 - Interaktionen
 - Kommunikationsabläufen in einem Szenario
 - Informationsaustausch zwischen beliebigen Kommunikationspartnern
 - innerhalb eines Systems oder
 - zwischen Systemen
- Ein Sequenzdiagramm
 - ist ein **Interaktionsdiagramm** und
 - beschreibt den **Ablauf der Kommunikation**
 - in einer Gruppe von Objekten

Sequenzdiagramm

Interaktionsdiagramme

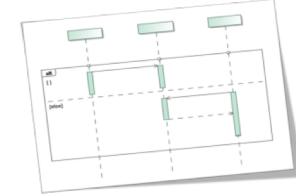


Motivation

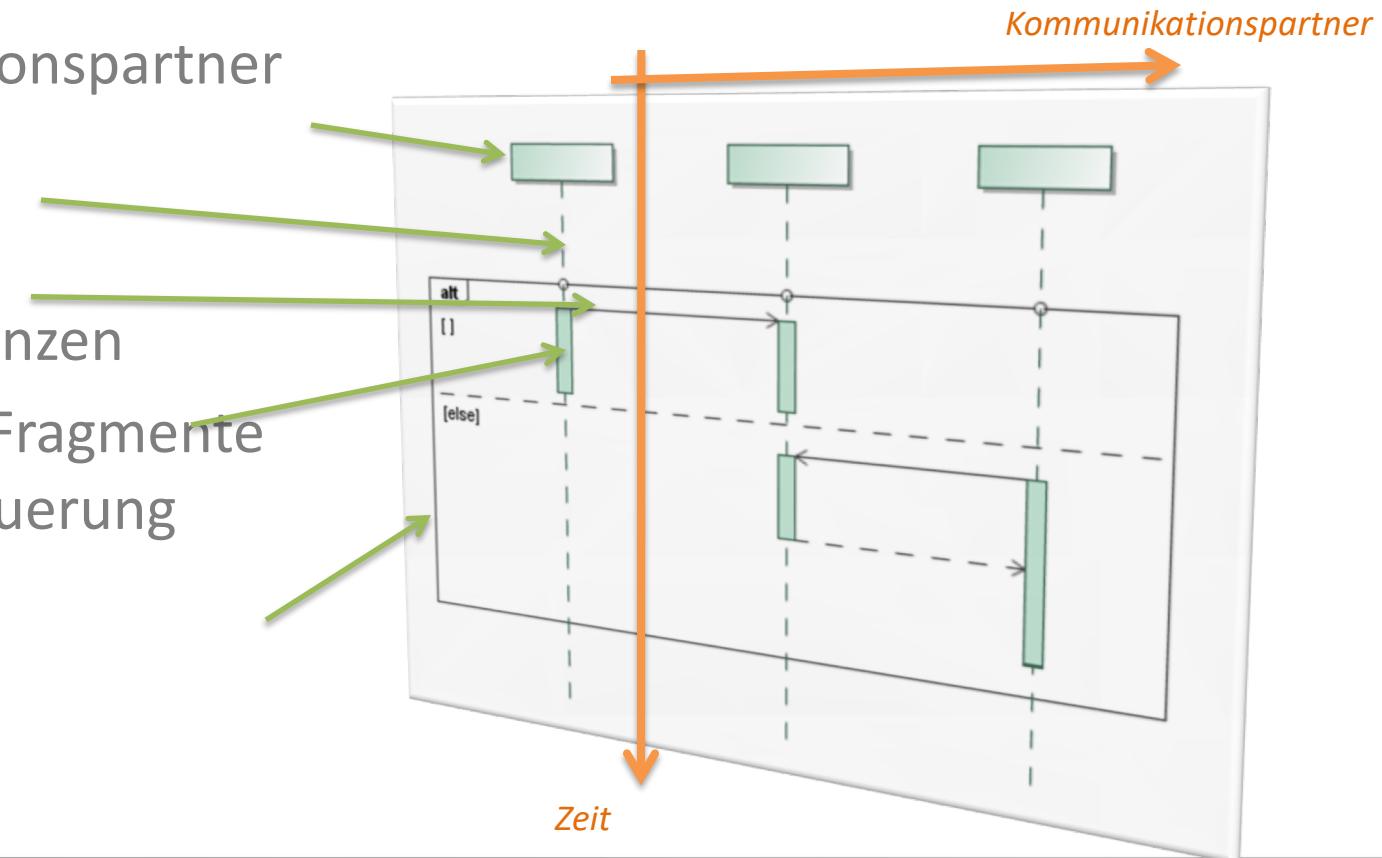
- Ein Sequenzdiagramm zeigt, wie Systemteile, Komponenten, etc. untereinander interagieren, welche Nachrichten oder Daten sie austauschen
- „Wie läuft die Kommunikation in meinem System ab?“
 - feste Reihenfolgen, zeitliche und logische Ablaufbedingungen, Schleifen und Nebenläufigkeiten
- Verwendung
 - Protokoll zur Beschreibung des Nachrichtenaustausches
 - Veranschaulichung des Ablaufes ereignisgesteuerter, interaktiver Programme



Wichtige Elemente



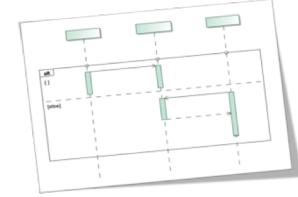
- Kommunikationspartner
- Lebenslinien
- Nachrichten
- Aktionssequenzen
- Kombinierte Fragmente
zur Ablaufsteuerung



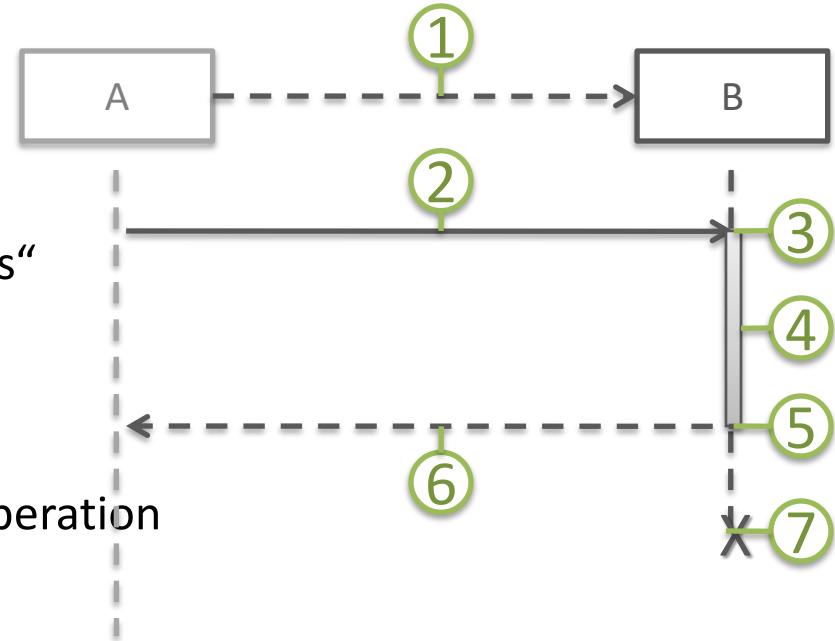
Sequenzdiagramm

Interaktionsdiagramme

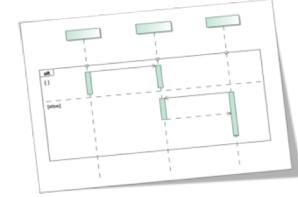
Wichtige Elemente: Kommunikationspartner



1. Erzeugungsauftrag
2. Nachricht
 - Z.B. Aufruf einer Operation
3. Start der Ausführung
4. Nachricht wird abgearbeitet
 - Kommunikationspartner B „tut etwas“
→ Ausführungssequenz
5. Ende der Ausführung
6. Antwortnachricht
 - Z.B. Mitteilen des Ergebnisses der Operation
 - Rücksprung
7. Destruktor
 - Instanz wird gelöscht
 - Lebenslinie endet

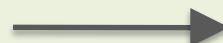


Wichtige Elemente: Nachrichten (1)



Darstellung

*synchroner
Operationsaufruf*



*asynchroner
Signal-/Operationsaufruf*



*synchrone
Antwort*



*asynchrone
Antwort*



Gefundene Nachricht



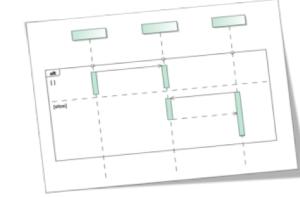
Nachricht, deren Sender
nicht bekannt ist

Verlorene Nachricht



Nachricht, deren Empfänger
nicht bekannt ist

Eingehakt: Kommunikationsarten



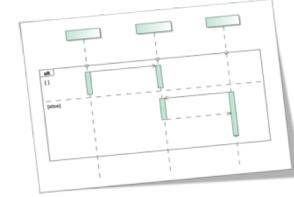
- **Synchrone Kommunikation**
 - Sender fährt nach Sendeereignis erst mit Abarbeitung fort, wenn vom Empfänger gesendete Antwortnachricht eingetroffen



- **Asynchrone Kommunikation**
 - Sender fährt direkt nach Sendeereignis mit Abarbeitung fort ohne auf Empfängersignal zu warten



Wichtige Elemente: Nachrichten (2)

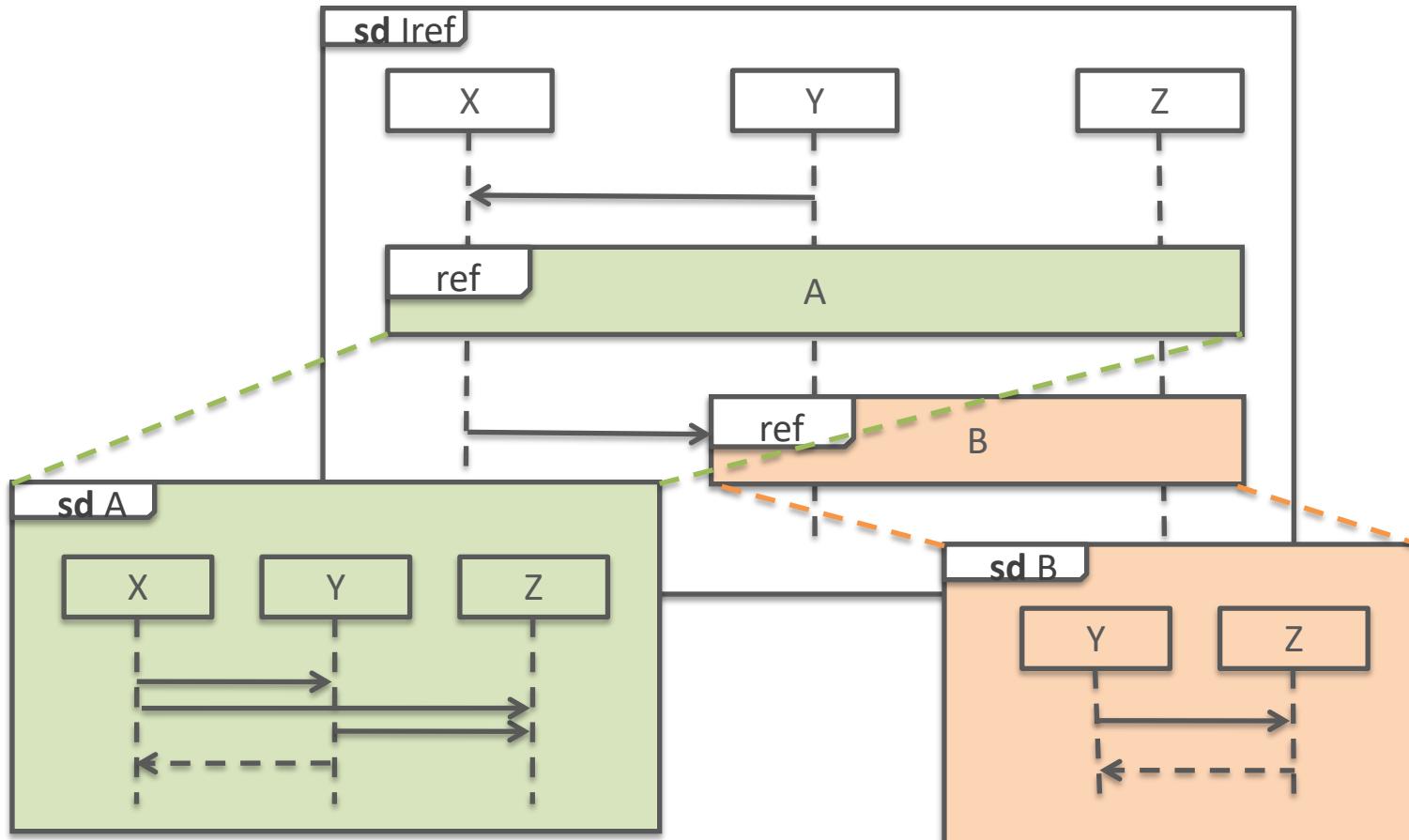
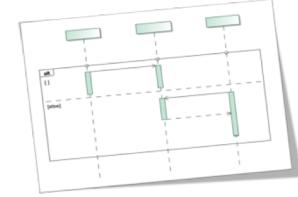


- Argumente einer Nachricht
 - Argumente können sein:
 - Attribute der sendenden Lebenslinie
 - Konstanten
 - Ein- und Ausgabeparameter der Interaktion
 - Attribute des Classifiers, zu dem die Interaktion gehört
 - Generische Werte (Wildcards)
 - Anzahl der Argumente = Anzahl der Parameter der aufzurufenden Operation
- Syntax einer Antwort
 - Name samt Aufrufparametern wird am Antwortpfeil wiederholt
 - Rückgabewerte werden hinter einem Doppelpunkt notiert

Sequenzdiagramm

Interaktionsdiagramme

Wichtige Elemente: Interaktionsreferenz

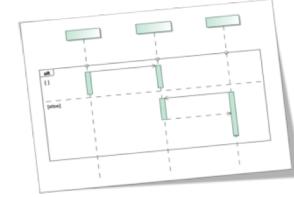


Sequenzdiagramm

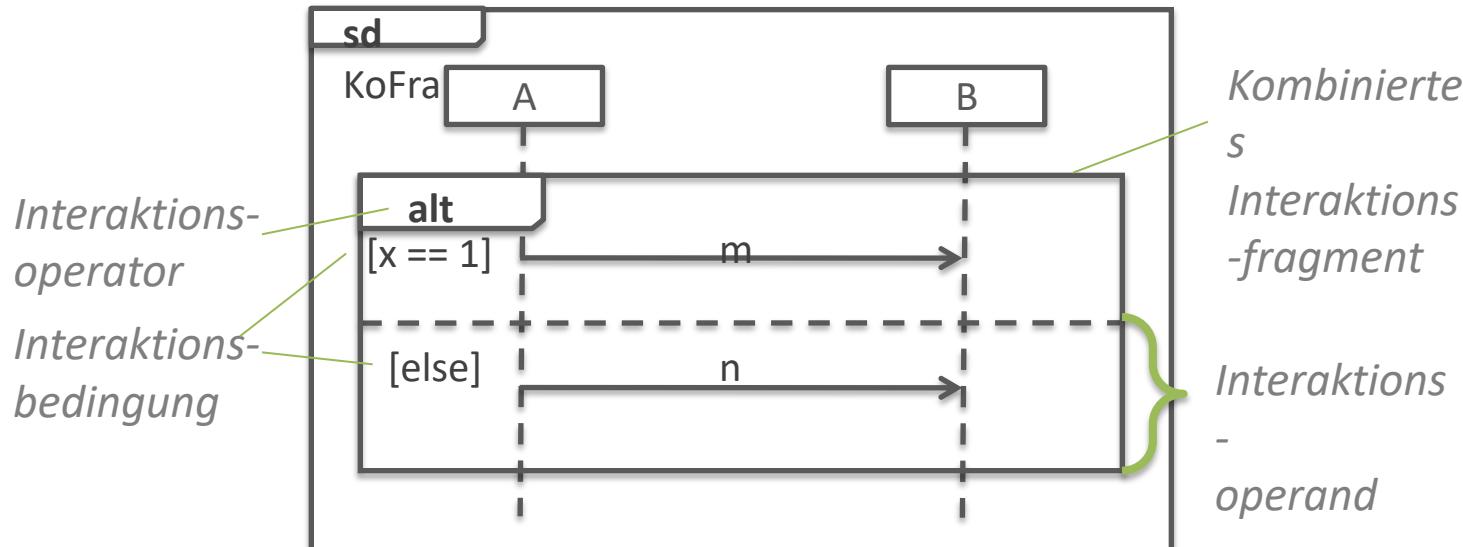
Interaktionsdiagramme

siehe auch UML glasklar
Ausgabe 3

Wichtige Elemente: Kombinierte Fragmente (1)



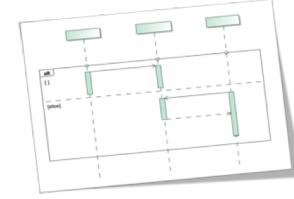
- Kombinierte Fragmente kennzeichnen einen Teil einer Interaktion, für den bestimmte Regeln gelten.
- Diese Regeln beeinflussen **Auswahl**, **Reihenfolge** und **Häufigkeit** von Nachrichten in diesem Fragment.



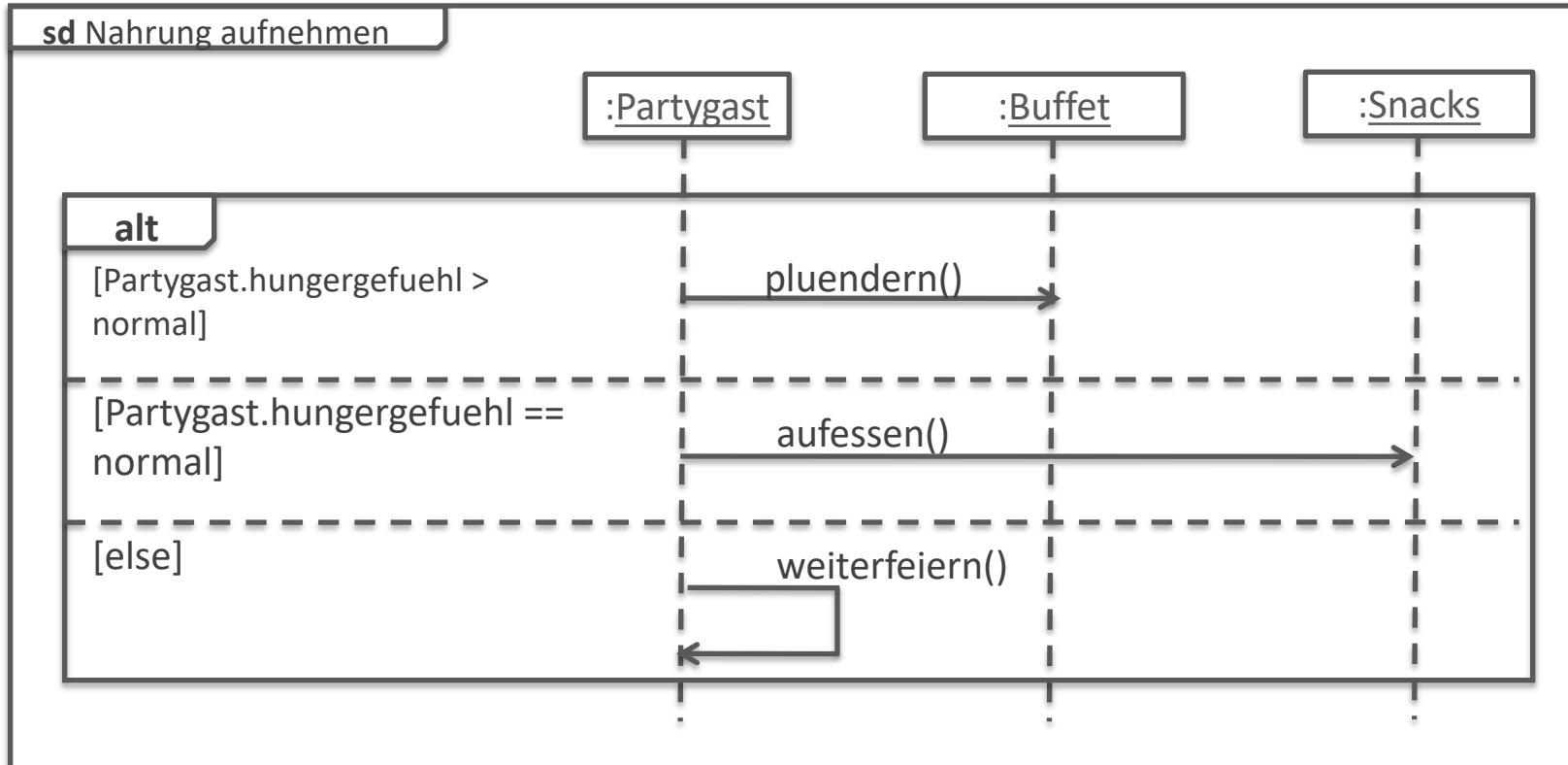
Sequenzdiagramm

Interaktionsdiagramme

Wichtige Elemente: Kombinierte Fragmente (2)



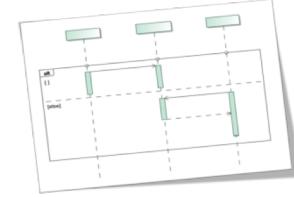
- Beispiel: Nahrungsaufnahme während einer Party



Sequenzdiagramm

Interaktionsdiagramme

Wichtige Elemente: Kombinierte Fragmente (3)



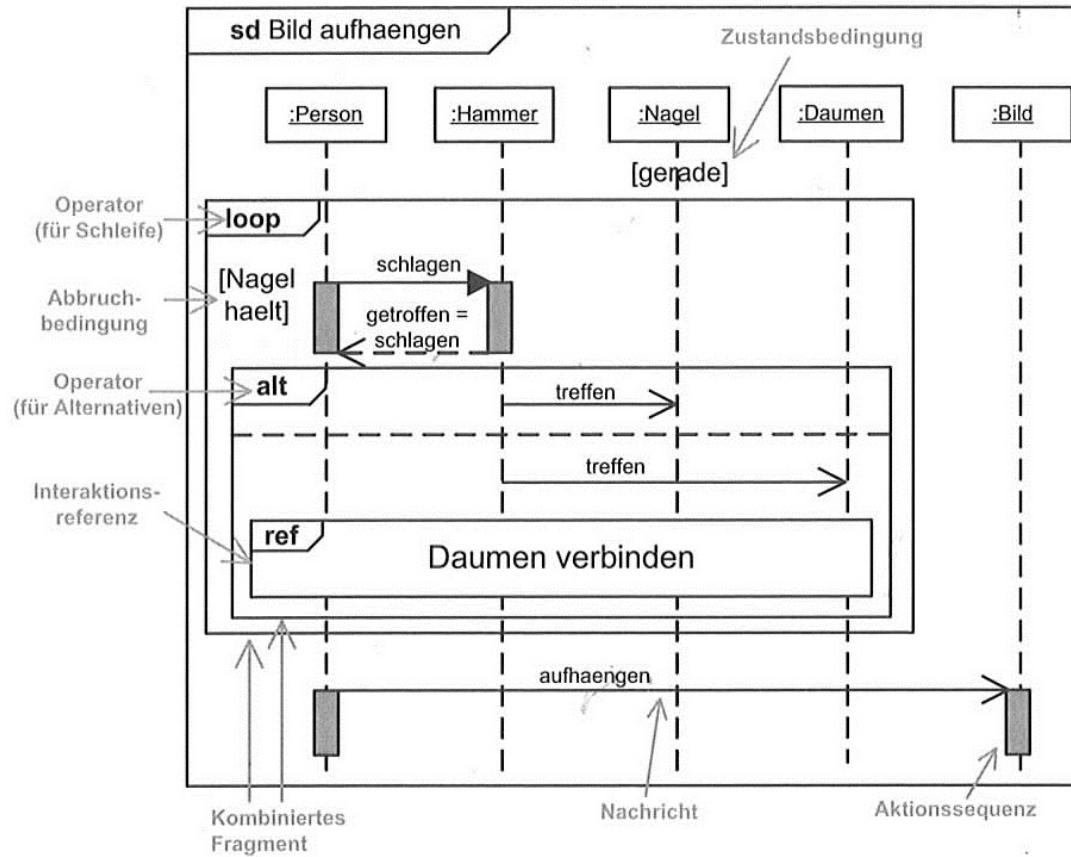
- **Auszug:**

Deutsche Bezeichnung	Englische Bezeichnung	Kürzel im Diagramm	Damit modellieren Sie...
Alternative Fragmente	Alternative	alt	...alternative Ablaufmöglichkeiten
Optionales Fragment	Option	opt	...optionale Interaktionsteile
Abbruchfragment	Break	break	...Interaktionen in Ausnahmefällen
Schleife	Loop	loop	...iterative Interaktionen
Parallele Fragmente	Parallel	par	...nebenläufige Interaktionen

Sequenzdiagramm

Interaktionsdiagramme

Sequenzdiagramm



Grafik: Rupp, Queins, Zengler, UML2 Glasklar, Seite 405

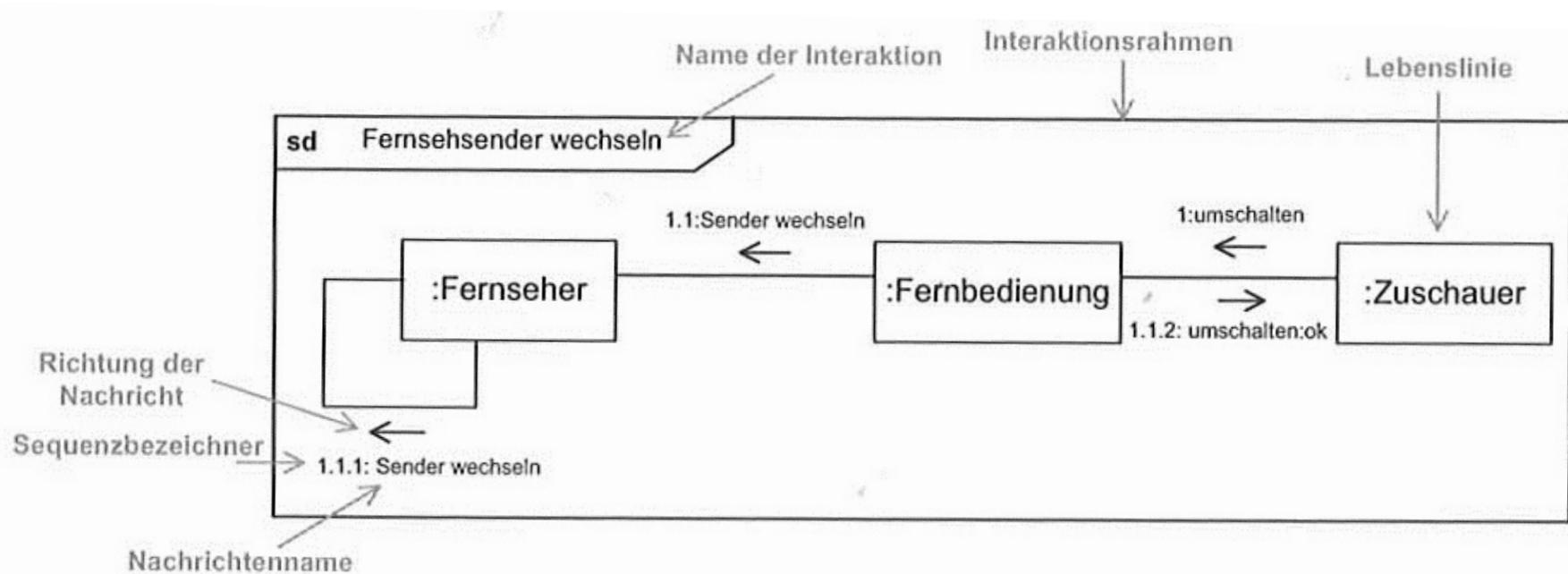
Inhalt

- **Beschreibungstechniken und Basiswissen**
 - Grundlagen der Objektorientierung (Wiederholung)
 - Übersicht zur UML
- **Strukturdiagramme**
 - Klassendiagramm
 - Paketdiagramm
 - Objektdiagramm
 - Kompositionssstrukturdiagramm
 - Komponentendiagramm
 - Verteilungsdiagramm
- **Verhaltensdiagramme**
 - Use-Case-Diagramm
 - Aktivitätsdiagramm
 - Zustandsautomat
- **Interaktionsdiagramme**
 - Sequenzdiagramm
 - **Kommunikationsdiagramm**
 - Timingdiagramm
 - Interaktionsübersichtsdiagramm
- **OCL (Object Constraint Language)**

Was ist Kommunikationsdiagramm?

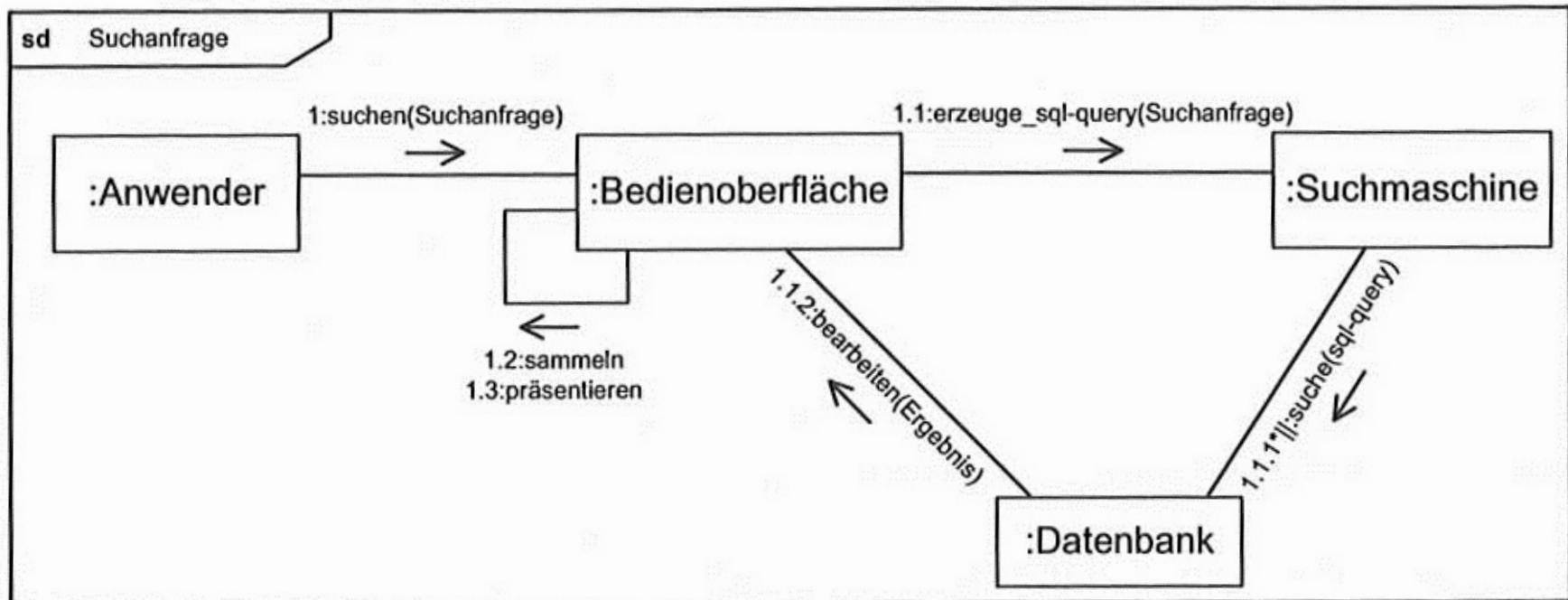
- Beschreiben der Interaktion zwischen Objekten mit dem Fokus auf die Struktur ihrer Kommunikationsbeziehungen.
- Modellieren der Nachrichtenaustausch während einer Interaktion
- auf mittleren Abstraktionsniveau
- Meisten zum Ergänzen Sequenzdiagramm verwenden

Kommunikationsdiagramm



Grafik: Rupp, Queins, Zengler, UML2 Glasklar, Seite 470

Anwendungsbeispiel für ein Kommunikationsdiagramm



Grafik: Rupp, Queins, Zengler, UML2 Glasklar, Seite 471

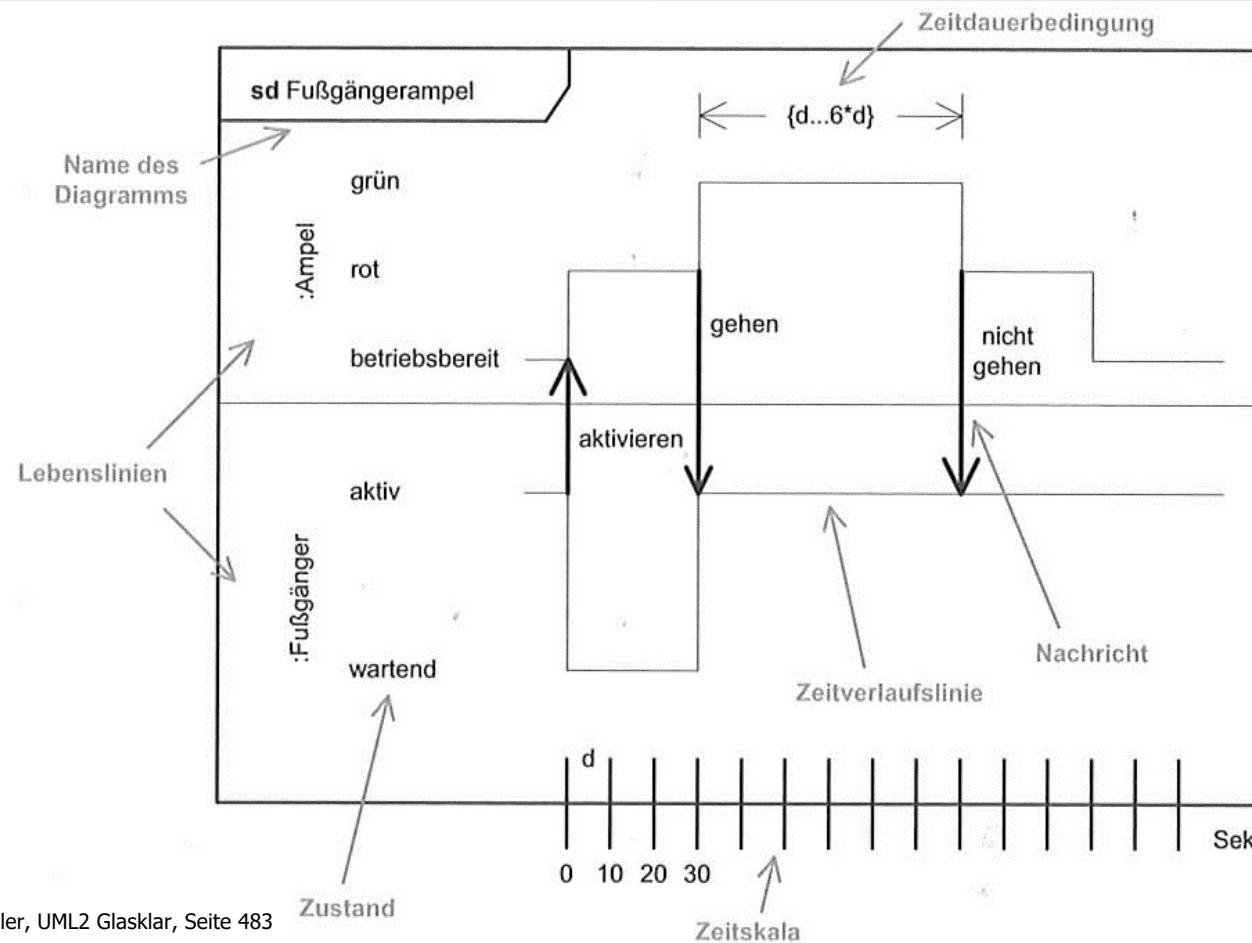
Inhalt

- **Beschreibungstechniken und Basiswissen**
 - Grundlagen der Objektorientierung (Wiederholung)
 - Übersicht zur UML
- **Strukturdiagramme**
 - Klassendiagramm
 - Paketdiagramm
 - Objektdiagramm
 - Kompositionssstrukturdiagramm
 - Komponentendiagramm
 - Verteilungsdiagramm
- **Verhaltensdiagramme**
 - Use-Case-Diagramm
 - Aktivitätsdiagramm
 - Zustandsautomat
- **Interaktionsdiagramme**
 - Sequenzdiagramm
 - Kommunikationsdiagramm
 - **Timingdiagramm**
 - Interaktionsübersichtsdiagramm
- **OCL (Object Constraint Language)**

Was ist ein Timingdiagramm?

- Zeigen des zeitlichen Verhalten von klassen in einem System
- Mehr auf zeitliche Aspekte der Zustandswechsel
- Mehr in der Elektrotechnik angewendet

Timingdiagramm

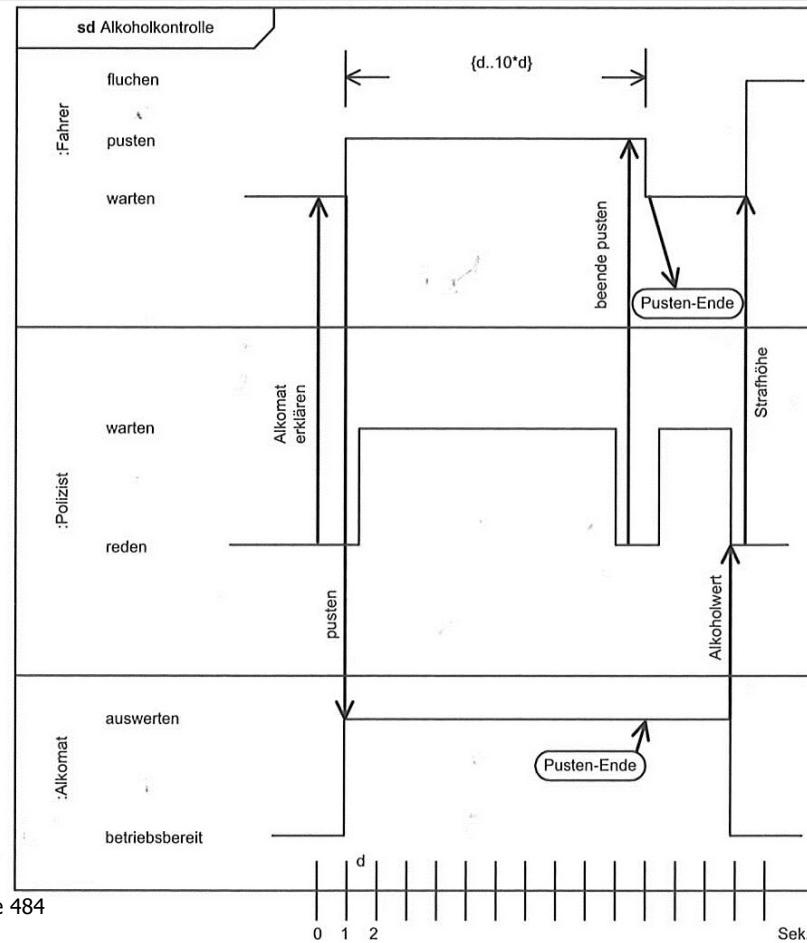


Grafik: Rupp, Queins, Zengler, UML2 Glasklar, Seite 483

Timingdiagramm

Interaktionsdiagramme

Anwendungsbeispiel für ein Timingdiagramm



Grafik: Rupp, Queins, Zengler, UML2 Glasklar, Seite 484

Timingdiagramm

Interaktionsdiagramme

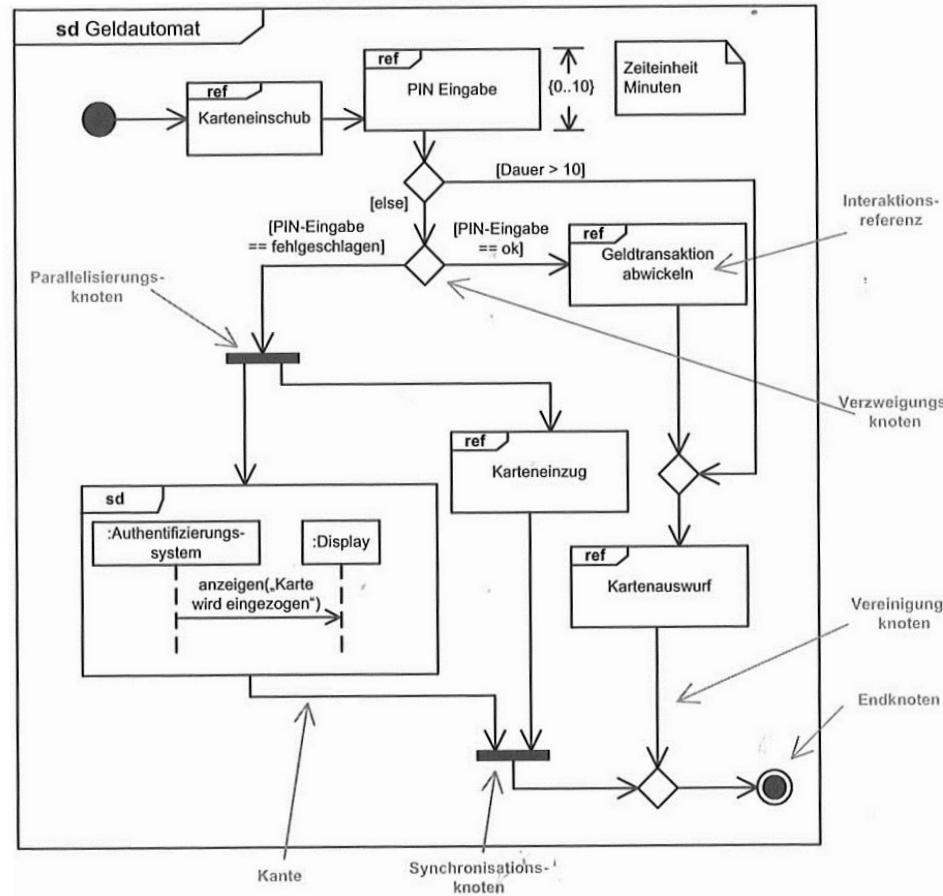
Inhalt

- **Beschreibungstechniken und Basiswissen**
 - Grundlagen der Objektorientierung (Wiederholung)
 - Übersicht zur UML
- **Strukturdiagramme**
 - Klassendiagramm
 - Paketdiagramm
 - Objektdiagramm
 - Kompositionssstrukturdiagramm
 - Komponentendiagramm
 - Verteilungsdiagramm
- **Verhaltensdiagramme**
 - Use-Case-Diagramm
 - Aktivitätsdiagramm
 - Zustandsautomat
- **Interaktionsdiagramme**
 - Sequenzdiagramm
 - Kommunikationsdiagramm
 - Timingdiagramm
 - **Interaktionsübersichtsdiagramm**
- **OCL (Object Constraint Language)**

Was ist Interaktionsübersichtsdiagramm?

- Eine Variante eines Aktivitätsdiagramm
- Zusammenspiel verschiedener Interaktion
- Überblick über einzelne Interaktionen und deren Reihenfolge

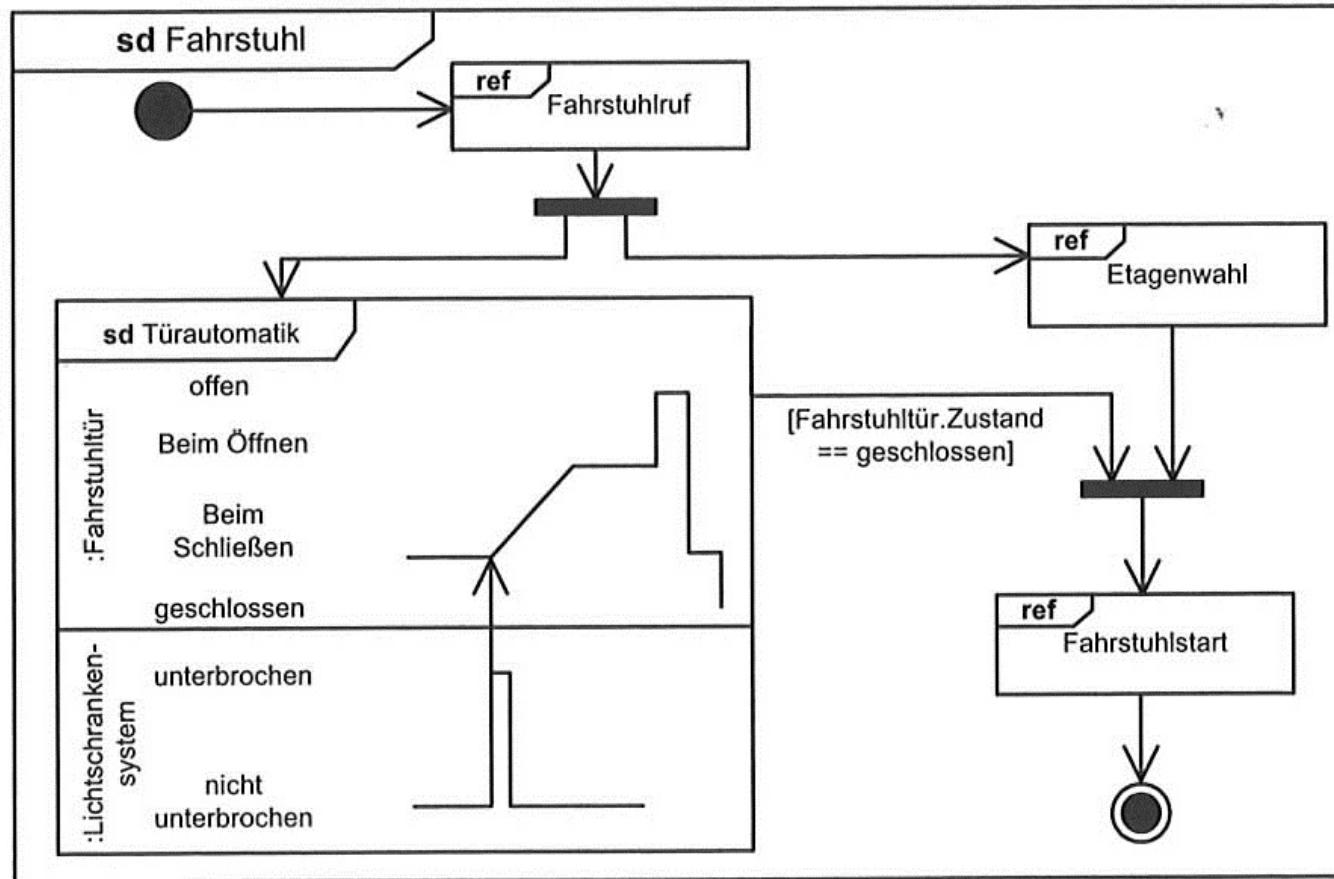
Interaktionsübersichtsdiagramm



Interaktions-
übersichtsdiagramm

Interaktionsdiagramme

Anwendungsbeispiel



Interaktions-
übersichtsdiagramm

Interaktionsdiagramme

Inhalt

- **Beschreibungstechniken und Basiswissen**
 - Grundlagen der Objektorientierung (Wiederholung)
 - Übersicht zur UML
- **Strukturdiagramme**
 - Klassendiagramm
 - Paketdiagramm
 - Objektdiagramm
 - Kompositionssstrukturdiagramm
 - Komponentendiagramm
 - Verteilungsdiagramm
- **Verhaltensdiagramme**
 - Use-Case-Diagramm
 - Aktivitätsdiagramm
 - Zustandsautomat
- **Interaktionsdiagramme**
 - Sequenzdiagramm
 - Kommunikationsdiagramm
 - Timingdiagramm
 - Interaktionsübersichtsdiagramm
- **OCL (Object Constraint Language)**

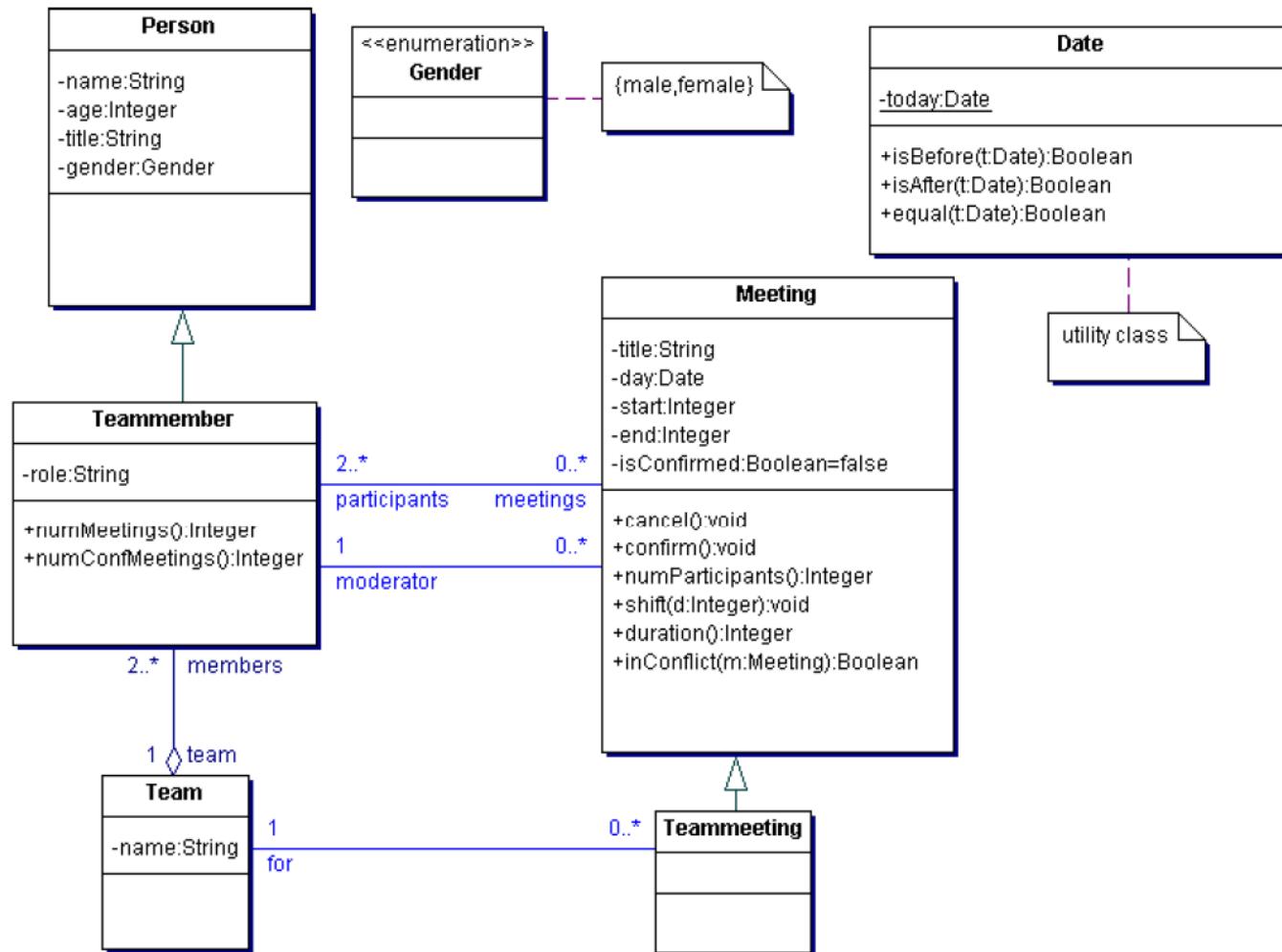
OCL (Object Constraint Language)

- ergänzt die Unified Modeling Language (UML)
- formale Sprache für die Definition von Constraints (Zusicherungen) und Anfragen auf UML-Modellen
- standardisiert (OMG)
- fügt graphischen (UML-)Modellen präzisierte Semantik hinzu
- inzwischen allgemein akzeptiert, viele Erweiterungen
- ...

Definition Constraint nach Brügge, Dutoit

- Eine **Einschränkung** ist ein Prädikat, dessen Wert wahr oder falsch ist.
- Boolesche Ausdrücke sind ... Einschränkungen. ...
- OCL erlaubt die formale Spezifikation von Einschränkungen für einzelne Modellelemente (z.B. Attribute, Operationen, Klassen) sowie für Gruppen von Modellelementen (z.B. Assoziationen)“
- Wir benutzen im folgenden weiter den Begriff des Constraints.

Beispiel – UML Diagramm



Invariante

- **Definition**
 - Eine **Invariante** ist ein Constraint, das für ein Objekt während seiner ganzen Lebenszeit wahr sein sollte.
- **Syntax**

```
context <class name>  
inv [<constraint name>] : <OCL expression>
```

Invariante - Beispiel

```
context Meeting
```

```
inv: self.end > self.start
```

- **self** bezieht sich immer auf das Objekt, für das das Constraint berechnet wird

- Äquivalente Formulierungen:

```
context Meeting
```

```
inv: end > start
```

- Vergabe eines Namens für das Constraint:

```
context Meeting
```

```
inv startEndConstraint: self.end > self.start
```

- Sichtbarkeiten von Attributen u.ä. werden durch OCL standardmäßig ignoriert.

Precondition (Vorbedingung)

- Pre- und Postconditions sind Constraints, die die Anwendbarkeit und die Auswirkung von Operationen spezifizieren, ohne dass dafür ein Algorithmus oder eine Implementation angegeben wird.
- **Definition**
 - Eine **Precondition** ist ein Boolescher Ausdruck, der zum Zeitpunkt des Beginns der Ausführung der zugehörigen Operation wahr sein muss.
- **Syntax**

```
context <class name>::<operation> (<parameters>)
pre [<constraint name>] : <OCL expression>
```

Precondition - Beispiele

```
context Meeting::shift(d:Integer)  
pre: self.isConfirmed = false
```

```
context Meeting::shift(d:Integer)  
pre: d>0
```

```
context Meeting::shift(d:Integer)  
pre: self.isConfirmed = false and d>0
```

Postcondition (Nachbedingung)

- **Definition**
 - Eine **Postcondition** ist ein Boolescher Ausdruck, der unmittelbar nach der Ausführung der zugehörigen Operation wahr sein muss.
- **Syntax**

```
context <class name>::<operation> (<parameters>)
post [<constraint name>] : <OCL expression>
```

Postcondition - Beispiele

```
context Meeting::duration() : Integer  
post: result = self.end - self.start
```

- result bezieht sich auf den Rückkehrwert der Operation

```
context Meeting::confirm()  
post: self.isConfirmed = true
```

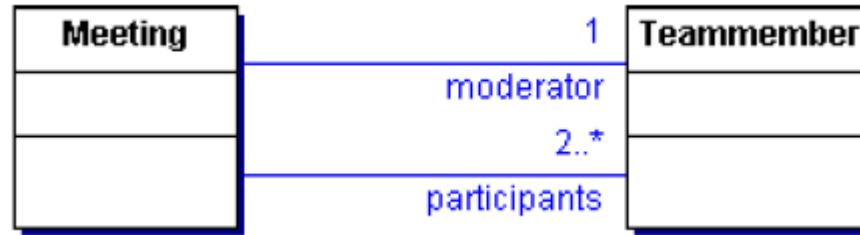
```
context Meeting::shift(d: Integer)  
post: start = start@pre + d and end = end@pre + d
```

- start@pre bezieht sich auf den Wert vor Ausführung der Operation
- start bezieht sich auf den Wert nach Ausführung der Operation
- @pre ist nur in Postconditions erlaubt

Navigationsausdrücke

- Assoziationsenden (Rollennamen) können verwendet werden, um von einem Object im Modell/System zu einem anderen zu navigieren (**Navigation**)
- Navigationen werden in OCL als Attribute behandelt (dot-Notation).
- Der Typ einer Navigation ist entweder
 - **Nutzerdefinierter Typ** (Assoziationsende mit Multiplizität maximal 1)
 - **Kollektion** von nutzerdefinierten Typen (Assoziationsende mit Multiplizität > 1)

Navigationsausdrücke (Beispiele)

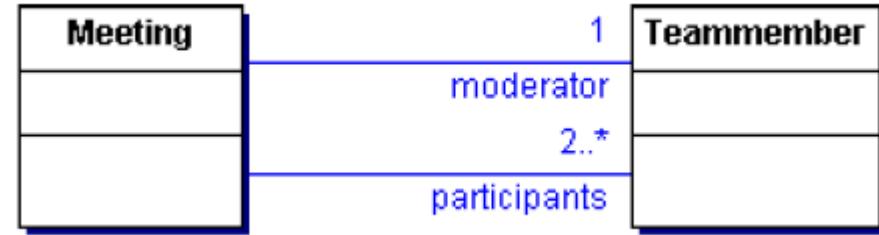


- **Nutzerdefinierter Typ**
 - z.B. moderator
 - Navigation von Meeting ist vom Typ Teammember

context Meeting

inv: **self.moderator.gender = Gender::female**

Navigationsausdrücke (Beispiele)



- **Kollektion**

- z.B. participant
- Navigation von Meeting ist vom Typ Set Teammember
- Operationen auf Kollektionen werden in der „Pfeilnotation“ (->) geschrieben
- Kurznotation für die collect-Operation ist die dot-Notation(für self->collect (participants) besser self.participants)

```
context Meeting
```

```
inv: self->collect(participants)->size() >= 2
```

```
context Meeting
```

```
inv: self.participants->size() >= 2
```

Operationen auf Kollektionen (1)

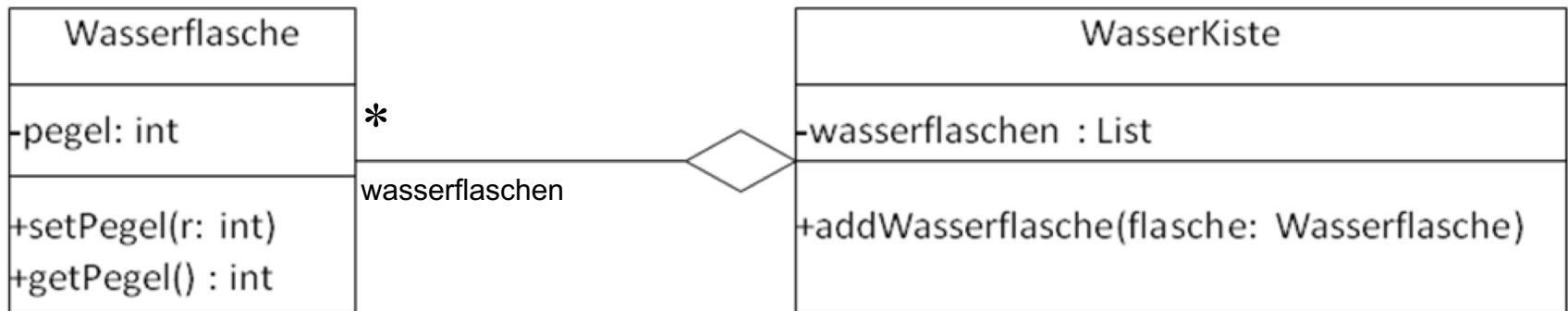
- 22 Operationen mit unterschiedlicher Semantik in Abhängigkeit vom Kollektionstyp, z.B.
 - Vergleichsoperationen (`=`, `<>`)
 - Konvertierungsoperationen (`asBag()`, `asSet()`,
`asOrderedSet()`, `asSequence()`)
 - Verschiedene including- und excluding-Operationen
 - Mengenoperationen
(`union`, `intersection`, `minus`, `symmetricDifference`)
 - Operationen auf sortierten Kollektionen (z.B. `first()`,
`last()` `indexOf()`)

Operationen auf Kollektionen (2)

- Iterierende Operationen auf allen Kollektionstypen, z.B.
 - any(expr)
 - collect(expr)
 - exists(expr)
 - forAll(expr)
 - isUnique(expr)
 - one(expr)
 - select(expr)
 - reject(expr)
 - sortedBy(expr)

Abschließende Beispiele (1)

- Drücken Sie mit Hilfe von OCL aus, dass die Wasserkiste niemals leer sein darf. Des Weiteren können sich nicht mehr als 12 Flaschen in der WasserKiste befinden.



CONTEXT: WasserKiste

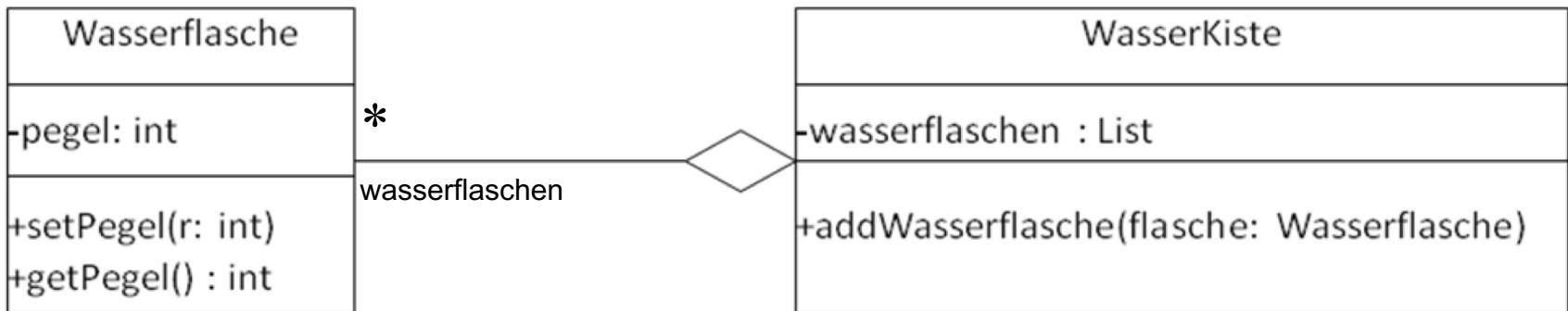
INV: SELF.wasserflaschen->size() > 0 AND SELF.wasserflaschen->size() <= 12

CONTEXT: WasserKiste

INV: NOT SELF.wasserflaschen->isEmpty() AND SELF.wasserflaschen->size() <= 12

Abschließende Beispiele (2)

- Geben Sie ein OCL-Constraint an, das folgenden Sachverhalt festlegt. Es muss sich mindestens eine Wasserflasche in der Wasserkiste befinden, deren Pegel höher als 3 Einheiten ist.



CONTEXT: WasserKiste

INV: SELF.wasserflaschen->exists(c | c.pegel > 3)

Literatur

- Spezifikation (siehe OMG-Webseite)
- Warmer, J., Kleppe, A.: The Object Constraint Language. Precise Modeling with UML. Addison-Wesley, 1999
- Warmer, J., Kleppe, A.: The Object Constraint Language Second Edition. Getting Your Models Ready For MDA. Addison-Wesley, 2003
- Brügge, B., Dutoit, A.H.: Objektorientierte Softwaretechnik mit UML, Entwurfsmustern und Java. PEARSON Studium, 2004
- Dr. Birgit Demuth, Einführung in OCL, Fakultät Informatik, Institut SMT, Lehrstuhl Softwaretechnologie