SE SOFTWARE ENGINEERING

RWTH AACHEN

Modellbasierte Softwareentwicklung

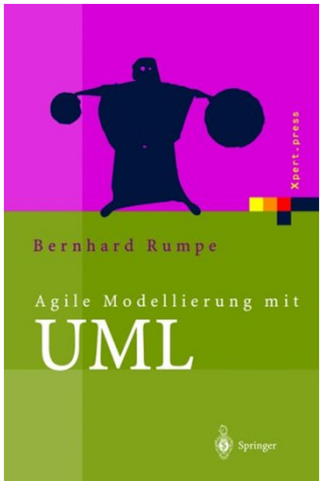
Vorlesungsmaterial
für eine Vorlesung mit 2h

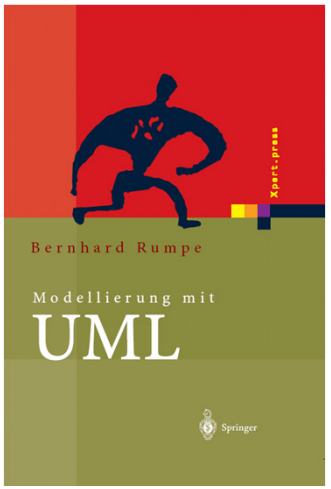
Weiterführendes Material:
http://mbse.se-rwth.de/

Prof. Dr. Bernhard Rumpe
Lehrstuhl für Software Engineering
RWTH Aachen

Vorlesung Modellbasierte Softwareentwicklung

- Modul: Bachelor / Master
  - vertiefende Vorlesung und Übung (2+3)
- Hörerkreis:
  - Informatik
  - und Verwandte
- Voraussetzungen:
  - Gute Programmierkenntnisse, idealerweise Java
  - Einführung in die Softwaretechnik (ggf. begleitend)
- Literatur
  - B. Rumpe:
    Agile Modellierung mit der UML,
    Springer 2011 & 2012 (zwei Bücher)
- **Weiterführendes Material:**
  - **http:/mbse.se-rwth.de/**



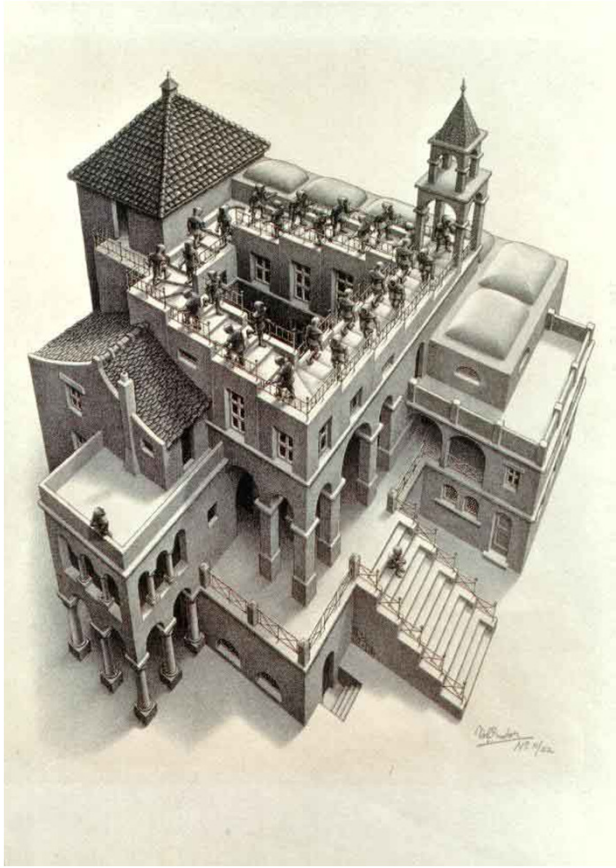


Inhalt

SE SOFTWARE ENGINEERING

RWTH AACHEN

Modellbasierte Softwareentwicklung



- 1. Begriffsklärung und Ziele

Prof. Dr. Bernhard Rumpe
Lehrstuhl für Software Engineering
RWTH Aachen

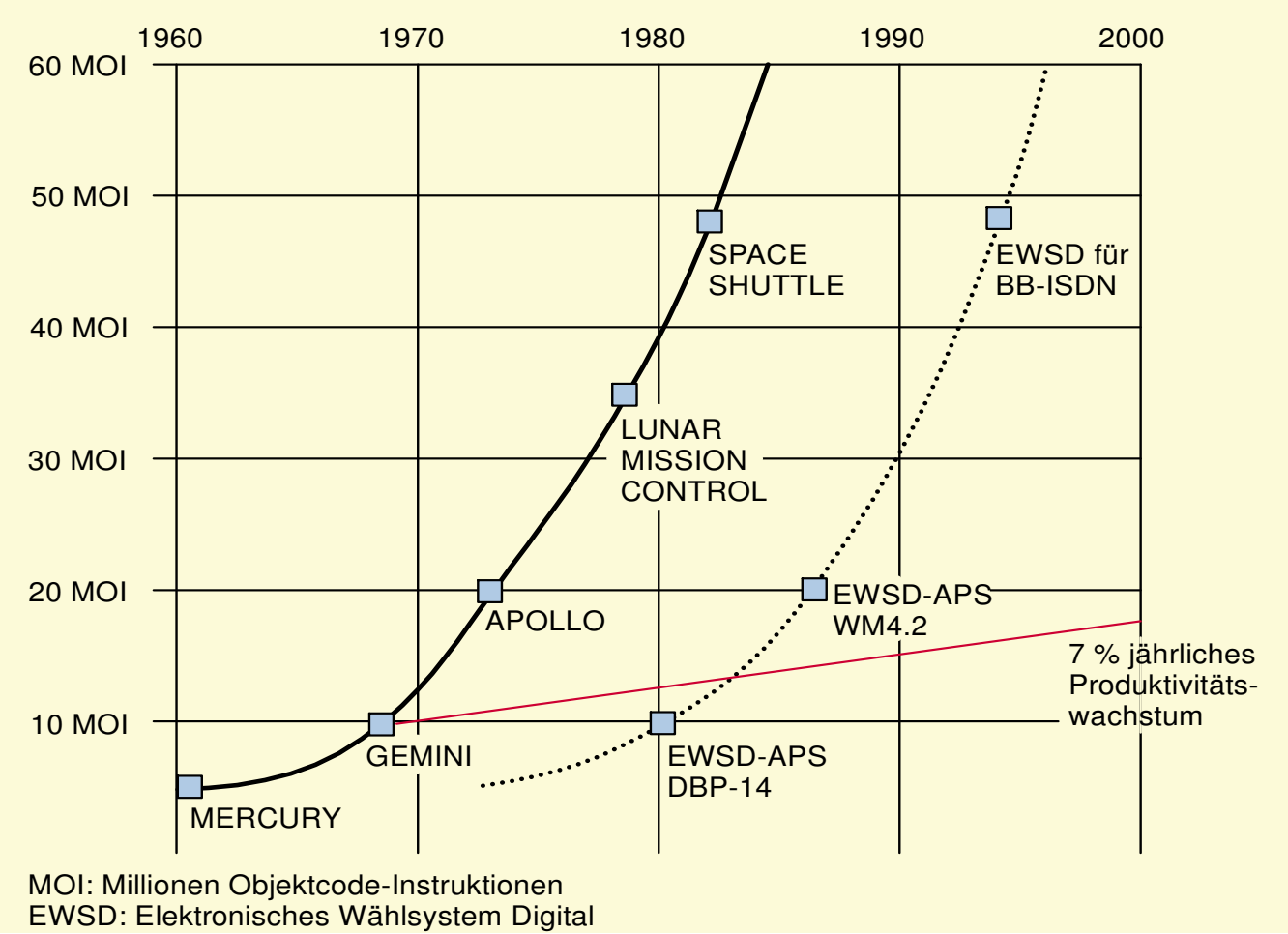
http://mbse.se-rwth.de/

Probleme der Softwareentwicklung:

- Software-Anteil an Produkten nimmt weiterhin dramatisch zu
- Komplexitätssteigerungen um Größenordnungen
- Eingebettete Software:
  - Kühlschrank, Videogerät, Handy, Auto, Flugzeug

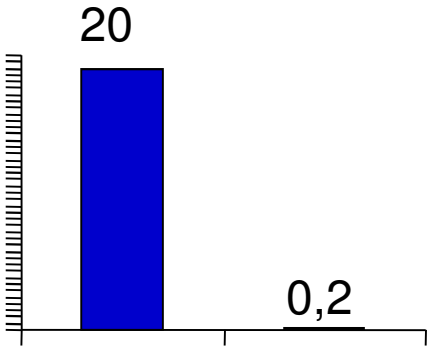
- Typische Probleme scheiternder Projekte:
  - Software zu spät fertig
  - Falsche Funktionalität realisiert
  - Software ist schlecht dokumentiert/kommentiert und kann nicht weiter entwickelt werden
  - Quellcode fehlt
  - Technische Umgebung wechselt
  - Geschäftsmodell / Anforderungen wechseln

Wachsende Komplexität der Software

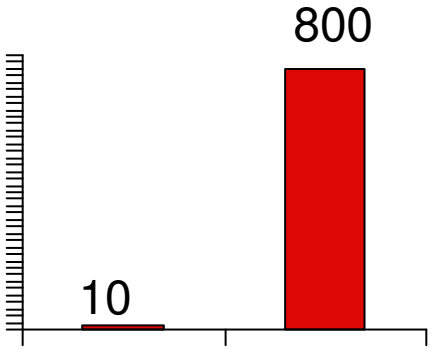


- Siemens EWSD V8.1: 12,5 Millionen LOC,
  ca.190.000 Seiten Dokumentation

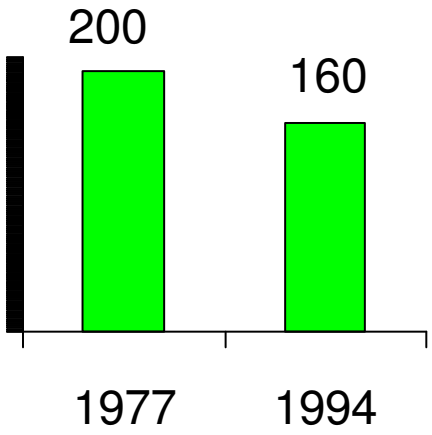
Komplexitätswachstum und Fehlerrate



Anzahl Fehler auf 1000 LOC



Programmgröße (1000 LOC)



Resultierende
absolute Fehleranzahl

Echte Qualitätsverbesserungen
sind nur möglich, wenn die
Steigerung der Programmkomplexität
überkompensiert wird !

(Durchschnittswerte, aus Balzert 96)

Portfolio von Softwareentwicklungs-Techniken

- Ziel:
  Vergrößerung des Portfolio von Techniken, Konzepten und Werkzeugen

- so dass für jedes Problem das richtige Verfahren existiert und von den Entwicklern beherrscht wird.

Bestandteile des Portfolios

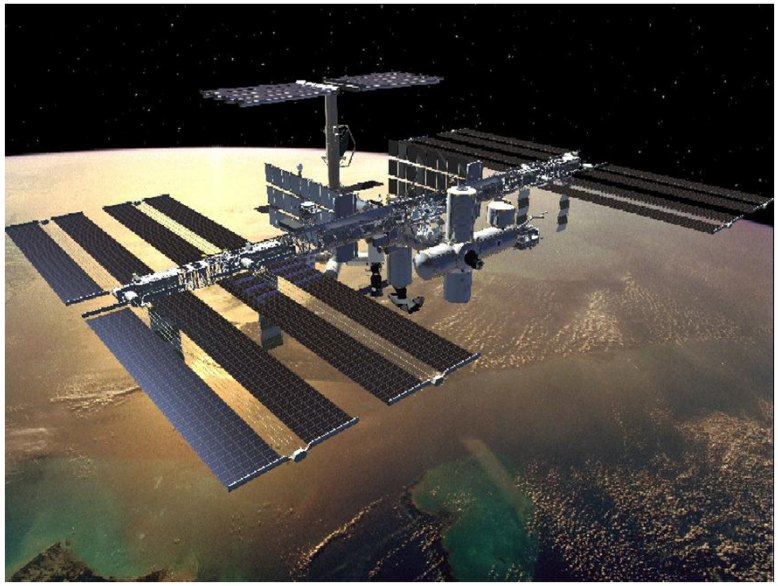
- Beispiele:
  - Konzepte: Hierarchische Zerlegung („Divide Et Impera“), Modularisierung, Zustandsbasierte Entwicklung
  - Werkzeuge: Compiler, SVN, Eclipse
  - Methoden: CRC-Karten zur Anforderungserhebung, Reviews, Testverfahren
  - Sprachen: zur Dokumentation, Implementierung, Modellierung

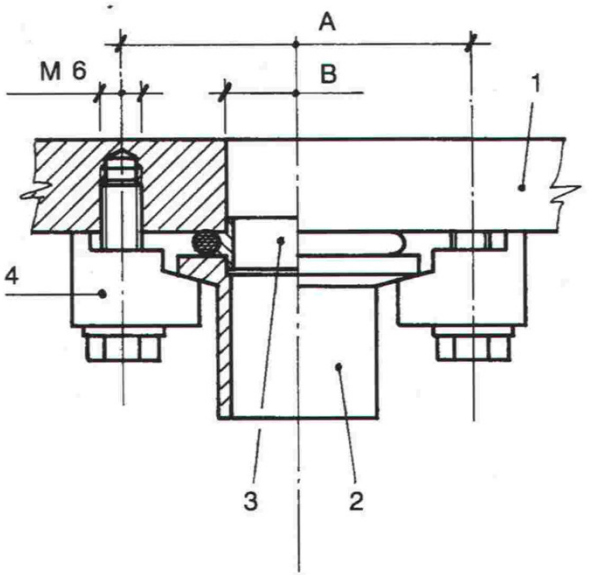
Was ist ein Modell

- Beispiele für Modelle:

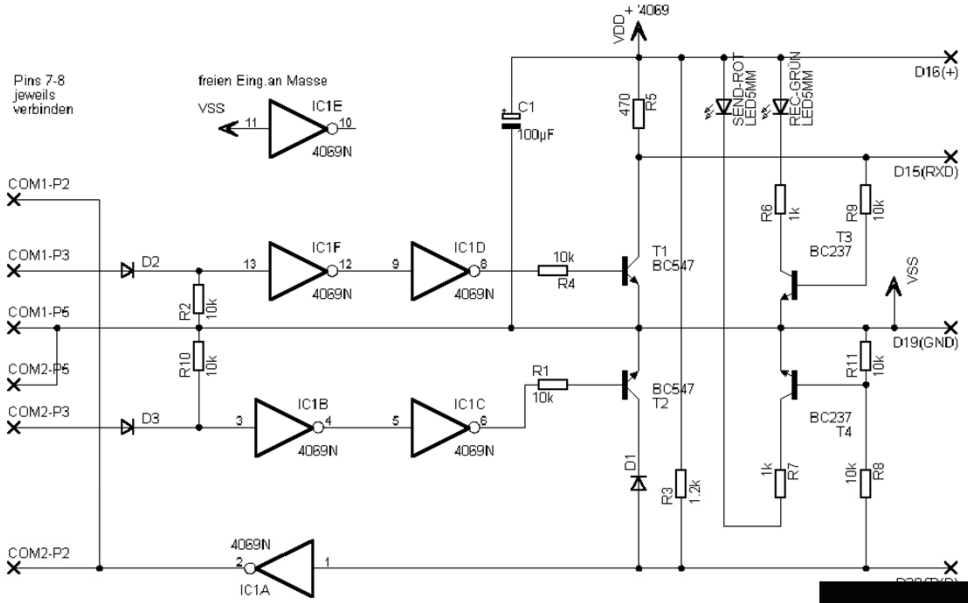
Vorschläge?

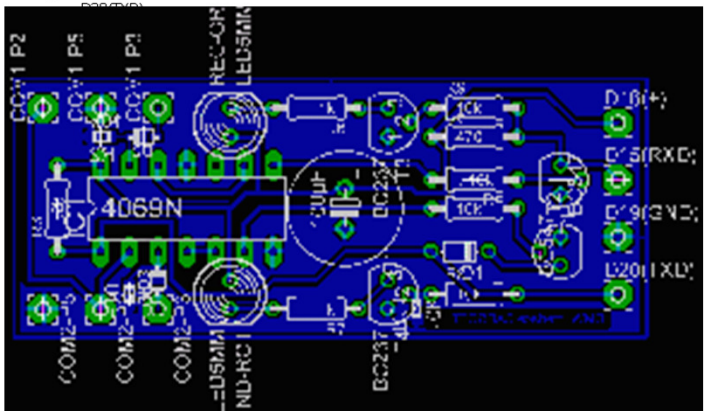
Maschinenbau: Modelle von Geräten in DIN/ISO-Normen



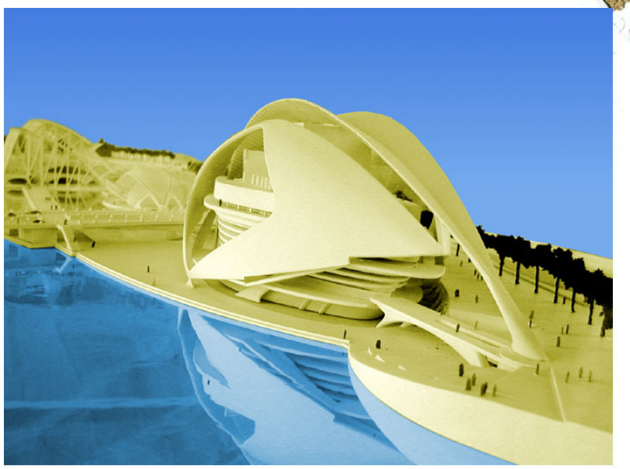


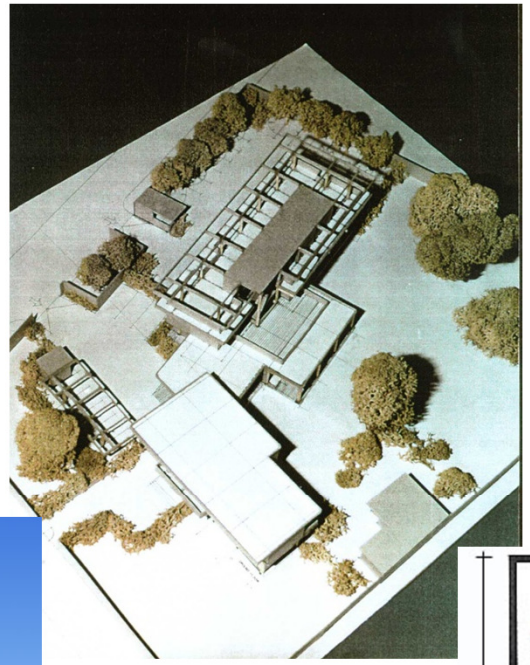
Elektrotechnik: Schaltkreise nach DIN/ISO-Normen

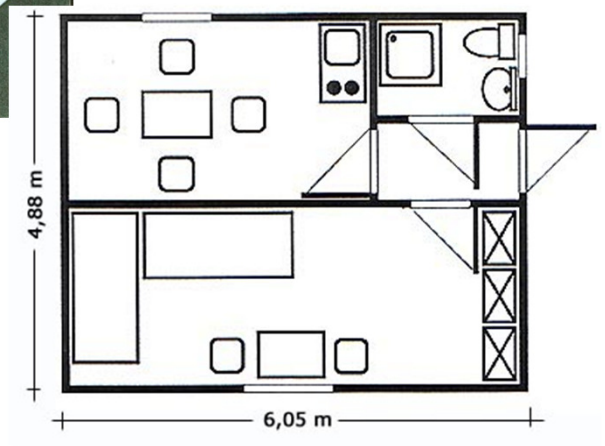




Architektur

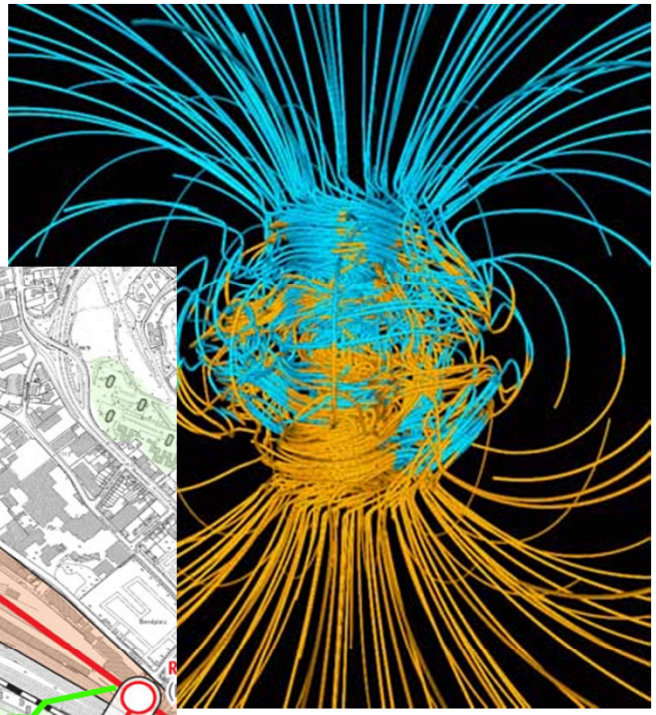


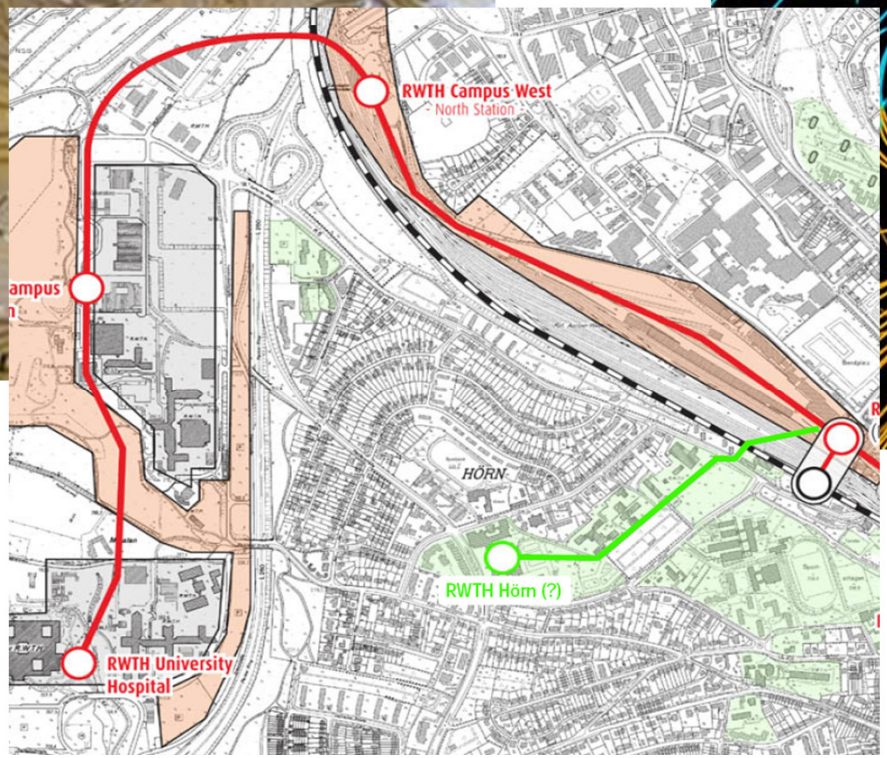




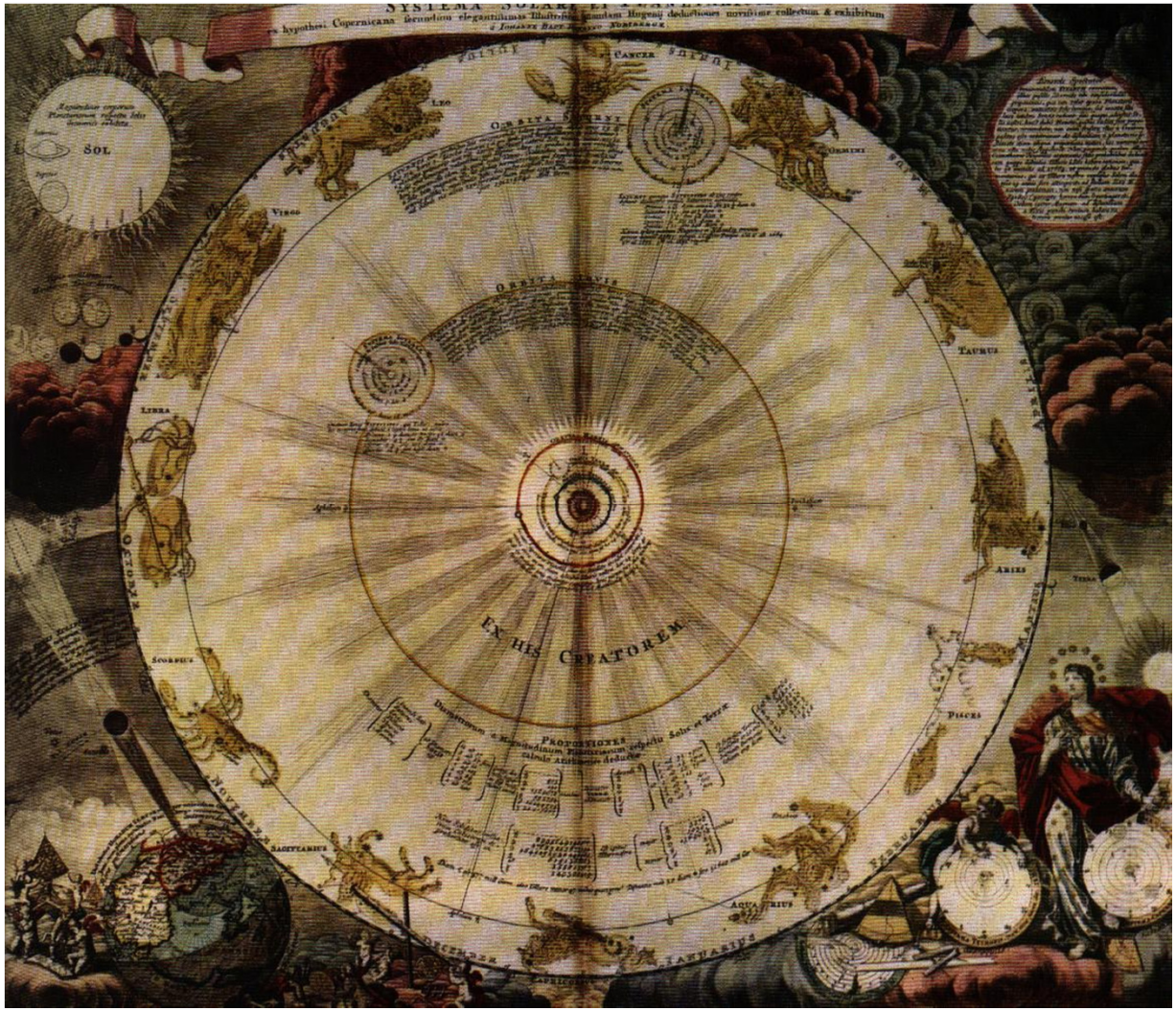
Geographie


Windberg
Obergittersee



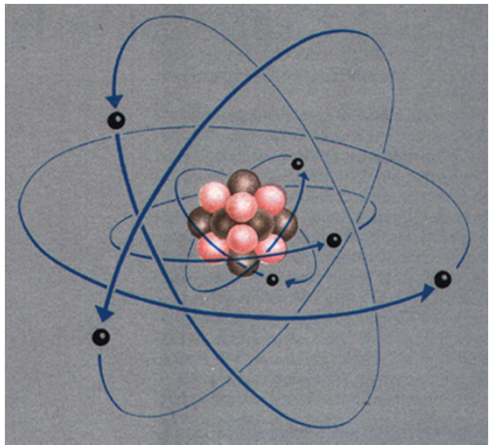

RWTH Campus West
North Station
HÖRN
RWTH Hörn (?)
RWTH University
Hospital

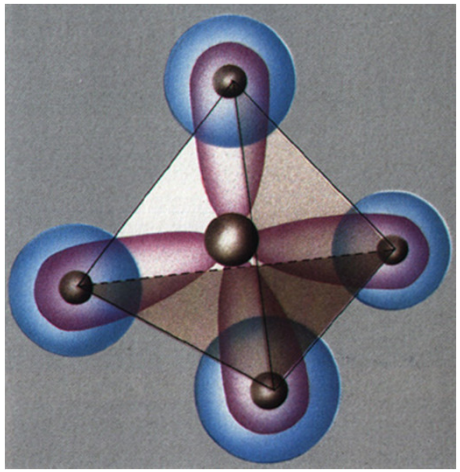
Astronomie: Geozentrisches Modell nach Kopernikus



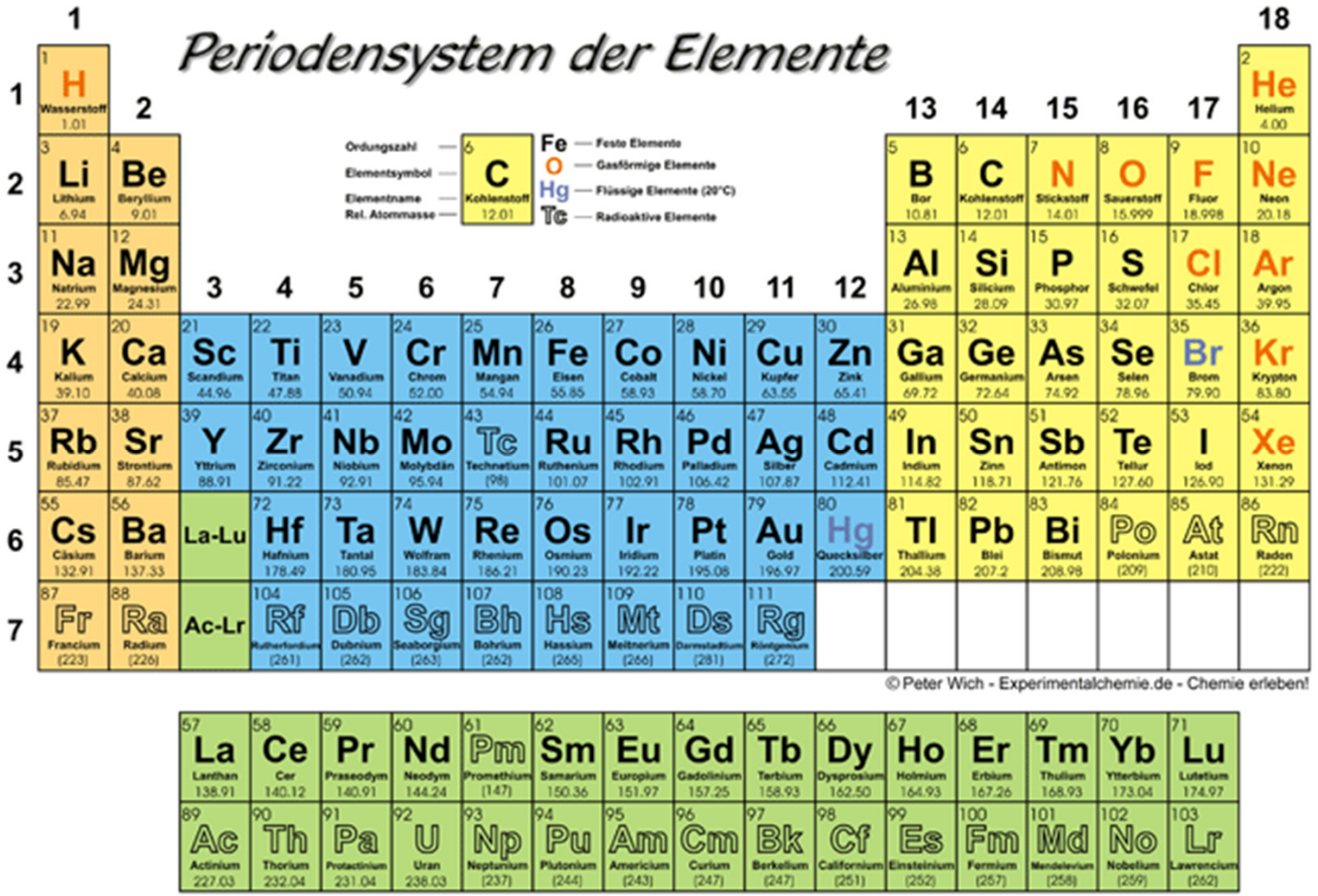
Physik:

- Rutherford‘sches, Bohrsches Atommodell
- Kugelschalenmodell
- Einsteins Relativitätstheorie
- Modell des Urknalls
- ...

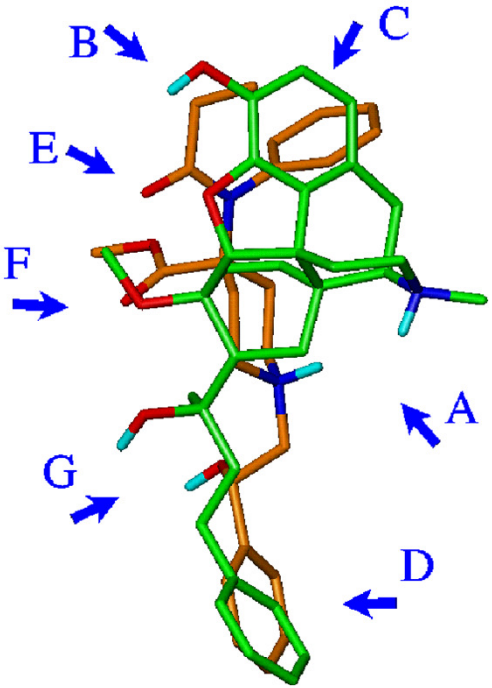




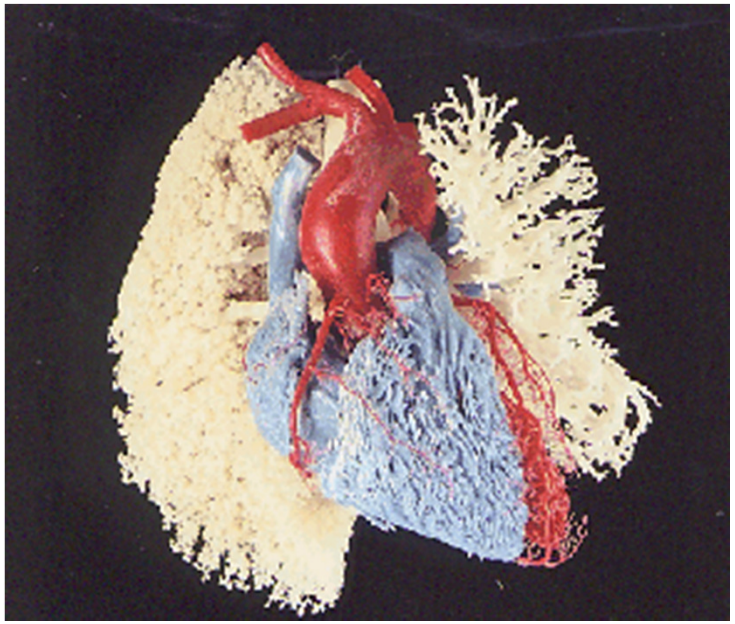
Chemie


Periodensystem der Elemente
Ordungszahl
Elementsymbol
Elementname
Rel. Atommasse
C Kohlenstoff 12.01
Fe Feste Elemente
O Gasförmige Elemente
Hg Flüssige Elemente (20°C)
Tc Radioaktive Elemente
1 H Wasserstoff 1.01
2 He Helium 4.00
3 Li Lithium 6.94
4 Be Beryllium 9.01
5 B Bor 10.81
6 C Kohlenstoff 12.01
7 N Stickstoff 14.01
8 O Sauerstoff 15.999
9 F Fluor 18.998
10 Ne Neon 20.18
11 Na Natrium 22.99
12 Mg Magnesium 24.31
13 Al Aluminium 26.98
14 Si Silicium 28.09
15 P Phosphor 30.97
16 S Schwefel 32.07
17 Cl Chlor 35.45
18 Ar Argon 39.95
19 K Kalium 39.10
20 Ca Calcium 40.08
21 Sc Scandium 44.96
22 Ti Titan 47.88
23 V Vanadium 50.94
24 Cr Chrom 52.00
25 Mn Mangan 54.94
26 Fe Eisen 55.85
27 Co Cobalt 58.93
28 Ni Nickel 58.70
29 Cu Kupfer 63.55
30 Zn Zink 65.41
31 Ga Gallium 69.72
32 Ge Germanium 72.64
33 As Arsen 74.92
34 Se Selen 78.96
35 Br Brom 79.90
36 Kr Krypton 83.80
37 Rb Rubidium 85.47
38 Sr Strontium 87.62
39 Y Yttrium 88.91
40 Zr Zirconium 91.22
41 Nb Niobium 92.91
42 Mo Molybdän 95.94
43 Tc Technetium (98)
44 Ru Ruthenium 101.07
45 Rh Rhodium 102.91
46 Pd Palladium 106.42
47 Ag Silber 107.87
48 Cd Cadmium 112.41
49 In Indium 114.82
50 Sn Zinn 118.71
51 Sb Antimon 121.76
52 Te Tellur 127.60
53 I Iod 126.90
54 Xe Xenon 131.29
55 Cs Cäsium 132.91
56 Ba Barium 137.33
La-Lu
72 Hf Hafnium 178.49
73 Ta Tantal 180.95
74 W Wolfram 183.84
75 Re Rhenium 186.21
76 Os Osmium 190.23
77 Ir Iridium 192.22
78 Pt Platin 195.08
79 Au Gold 196.97
80 Hg Quecksilber 200.59
81 Tl Thallium 204.38
82 Pb Blei 207.2
83 Bi Bismut 208.98
84 Po Polonium (209)
85 At Astat (210)
86 Rn Radon (222)
87 Fr Francium (223)
88 Ra Radium (226)
Ac-Lr
104 Rf Rutherfordium (261)
105 Db Dubnium (262)
106 Sg Seaborgium (263)
107 Bh Bohrium (262)
108 Hs Hassium (265)
109 Mt Meitnerium (266)
110 Ds Darmstadtium (281)
111 Rg Röntgenium (272)
© Peter Wich - Experimentalchemie.de - Chemie erleben!
57 La Lanthan 138.91
58 Ce Cer 140.12
59 Pr Praseodym 140.91
60 Nd Neodym 144.24
61 Pm Promethium (147)
62 Sm Samarium 150.36
63 Eu Europium 151.97
64 Gd Gadolinium 157.25
65 Tb Terbium 158.93
66 Dy Dysprosium 162.50
67 Ho Holmium 164.93
68 Er Erbium 167.26
69 Tm Thulium 168.93
70 Yb Ytterbium 173.04
71 Lu Lutetium 174.97
89 Ac Actinium 227.03
90 Th Thorium 232.04
91 Pa Protactinium 231.04
92 U Uran 238.03
93 Np Neptunium (237)
94 Pu Plutonium (244)
95 Am Americium (243)
96 Cm Curium (247)
97 Bk Berkelium (247)
98 Cf Californium (251)
99 Es Einsteinium (252)
100 Fm Fermium (257)
101 Md Mendelevium (258)
102 No Nobelium (259)
103 Lr Lawrencium (262)

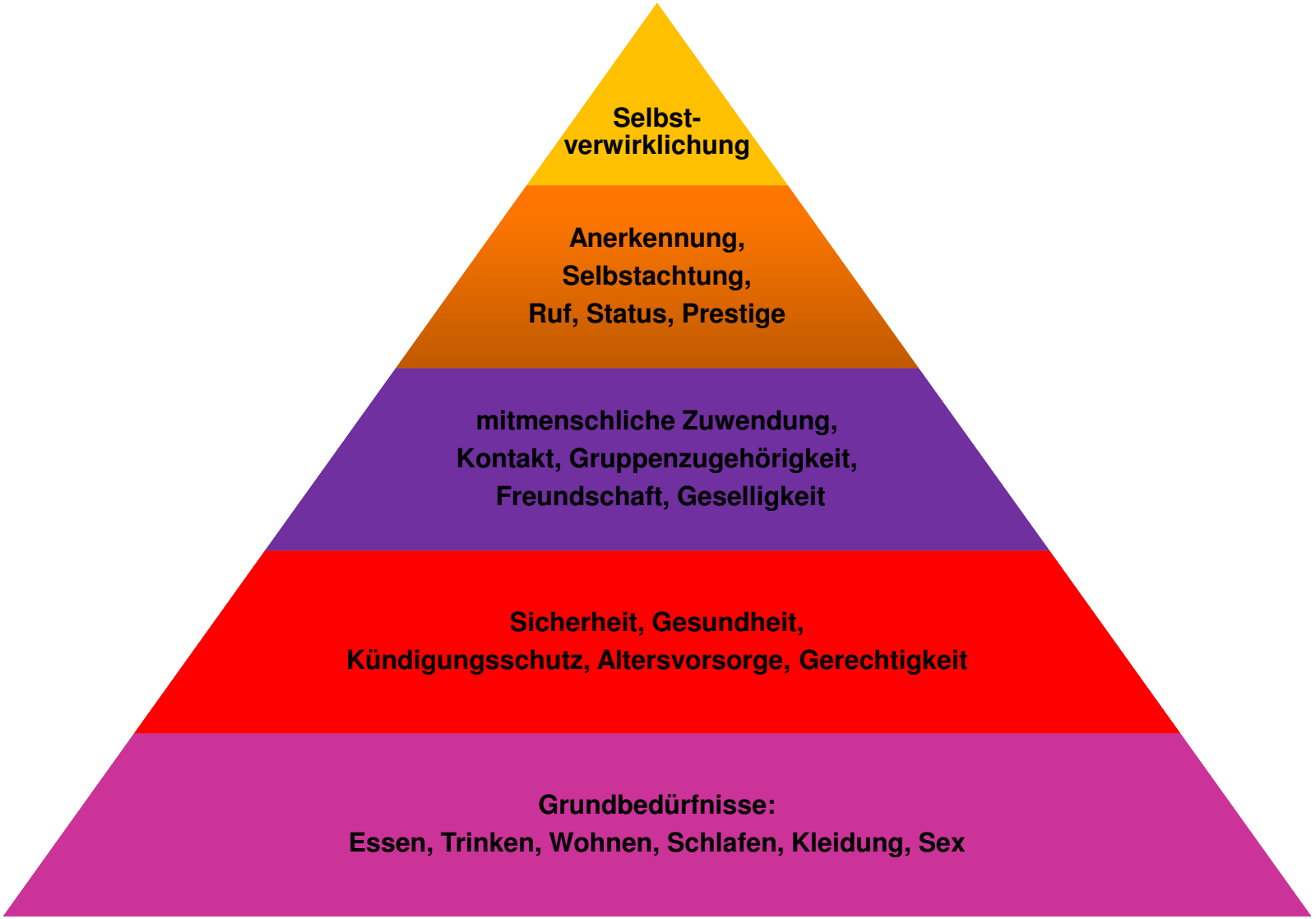
Biologie:
Modelle von Tieren, Enzymen, Molekülen, ...



Medizin, Sicherheitstechnik



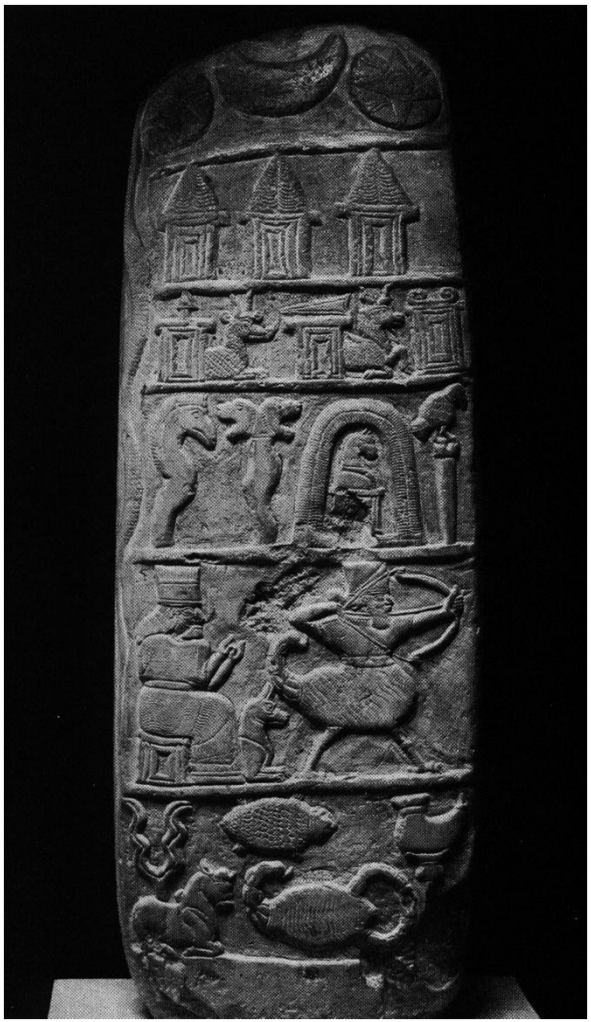
Soziologie: Maslow‘s Bedürfnispyramide



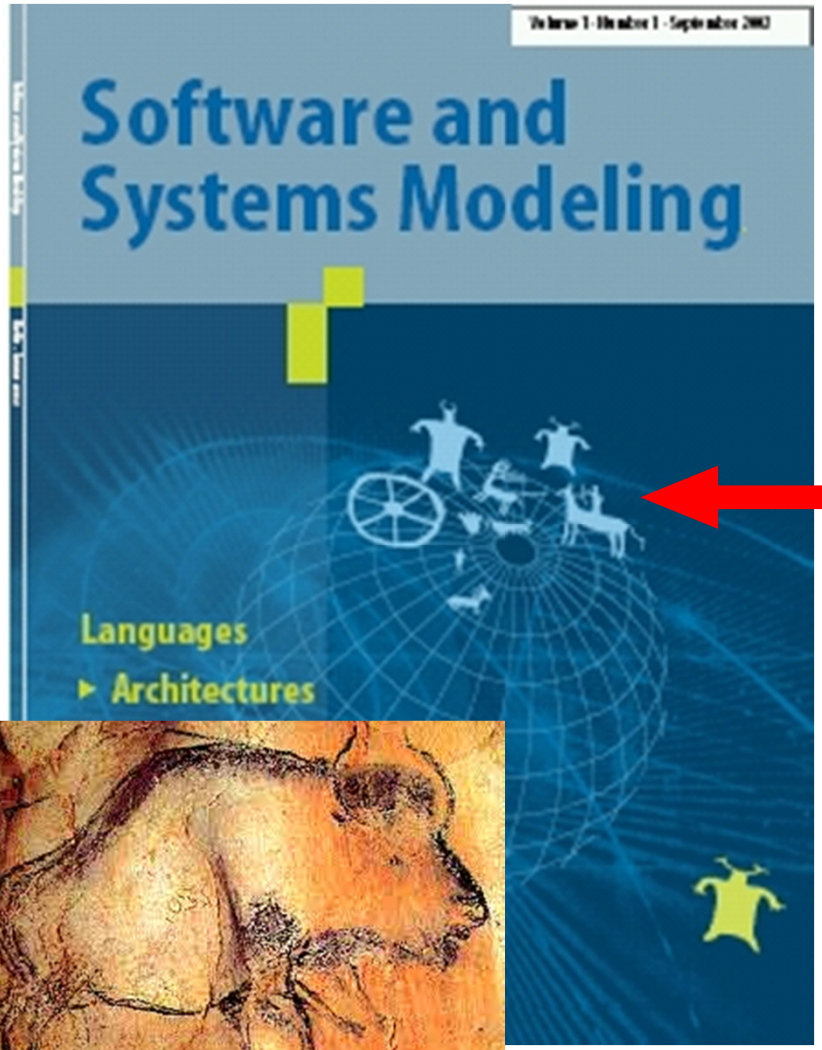
Wirtschaft:

- Modelle
  - des Geldkreislaufs
  - des Verhaltens von Anlegern
  - der Wirtschaftsentwicklung
  - des Verbraucherverhalten
  - ...

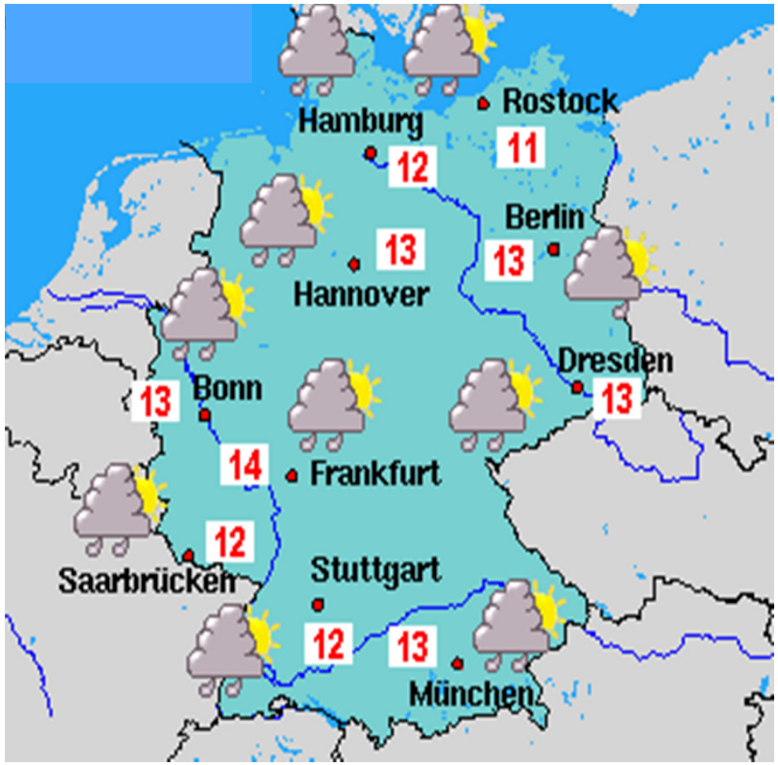
Die ersten Modelle: Hieroglyphen, frühe „Schriften“



Die wirklich ersten (noch erhaltenen) Modelle: Höhlenzeichnungen



Tägliches Leben: Wetterkarte

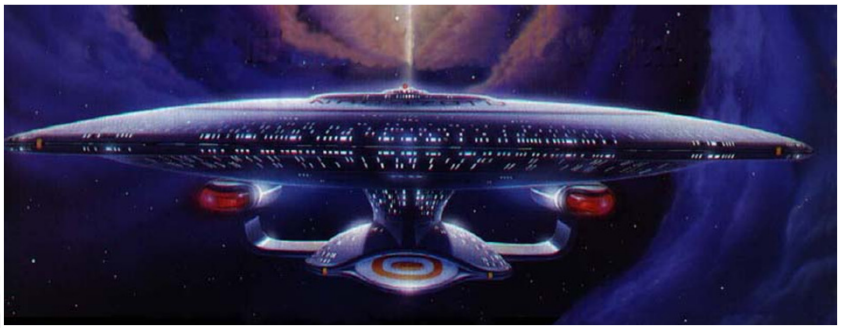


Der Modellbegriff

Ein Modell ist seinem Wesen nach eine in Maßstab, Detailliertheit und/oder Funktionalität verkürzte beziehungsweise abstrahierte Darstellung des originalen Systems. (Stachowiak 1973)

Ein Modell ist eine vereinfachte, auf ein bestimmtes Ziel hin ausgerichtete Darstellung der Funktion eines Gegenstands oder des Ablaufs eines Sachverhalts, die eine Untersuchung oder eine Erforschung erleichtert oder erst möglich macht. (Balzert 2000)

Wer/Was ist kein Modell?



Was ist ein Modell, was das Original?

(Rene Magritte)

Verwendung von Modellen

- Beispiel aus dem Internet:

- „Merksätze zur Verwendung mathematischer Modelle
  - Wenden Sie keine Modellrechnung an, solange Sie nicht die Vereinfachungen, auf denen sie beruht, geprüft und ihre Anwendbarkeit festgestellt haben.

    Merksatz: Unbedingt Gebrauchsanleitung beachten!

  - Verwechseln Sie nie das Modell mit der Realität.

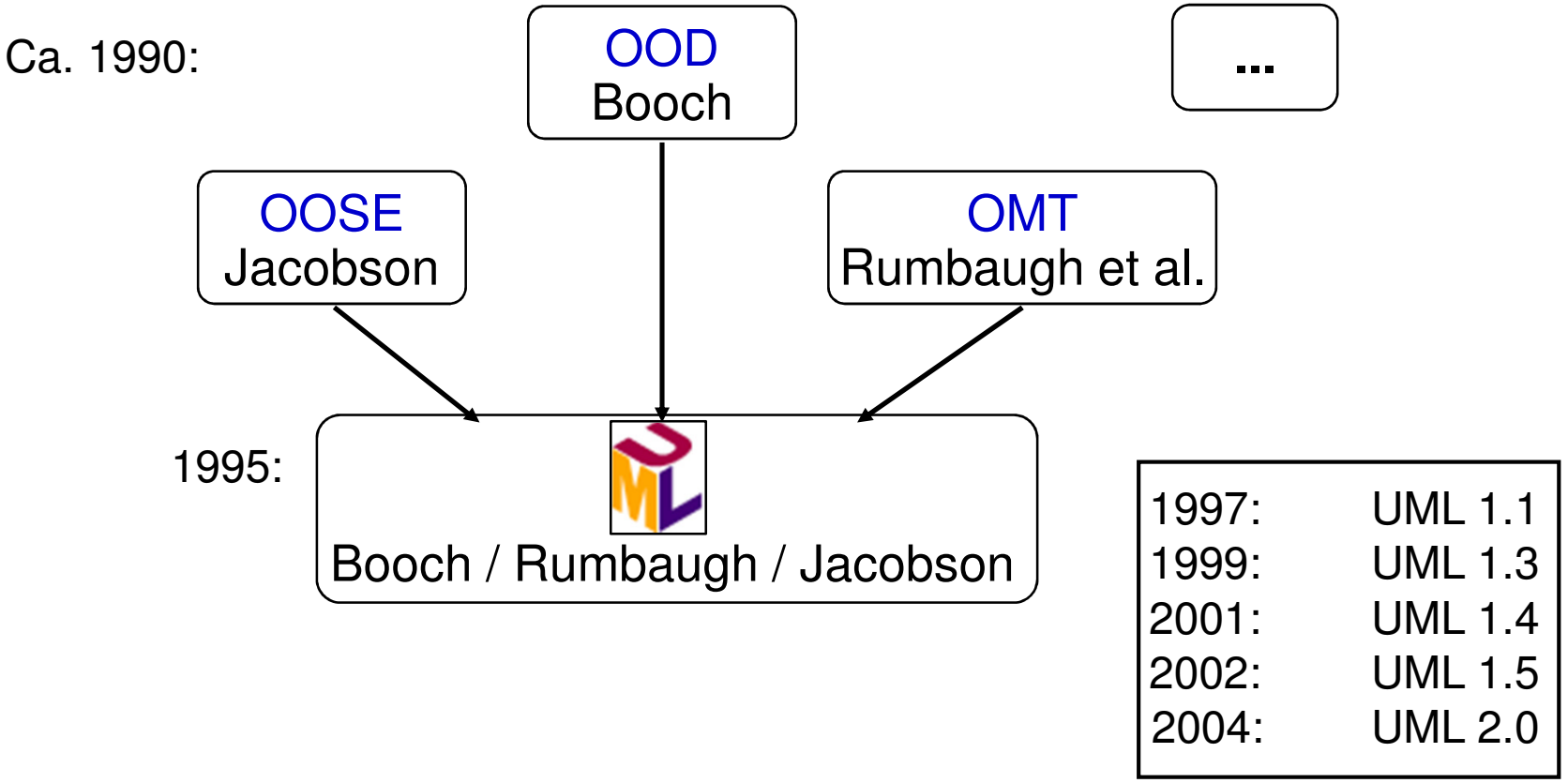
    Merksatz: Versuche nicht, die Speisekarte zu essen!“

Modellierung in der Softwaretechnik

- Industriestandard: Unified Modeling Language
  - 13 Diagrammtechniken (Klassendiagramme, Statecharts etc.)

- Aber auch:
  - Petri Netze
  - Logik
  - Relationen
  - Datenflussdiagramme
  - Nassi-Schneidermann-Diagramme
  - SDL
  - Endliche Automaten
  - etc.
  - Algebraische Spezifikation
  - Entity/Relationship-Model
  - Jackson Structured Diagrams
  - Kontrollflussdiagramme
  - Grammatiken
  - Reguläre Ausdrücke

Unified Modeling Language UML



- UML ist eine Notation der zweiten Generation für objektorientierte Modellierung
- UML ist Industriestandard der OMG (Object Management Group)

Unified Modeling Language

- Graphische Modellierungssprache für Software-Systeme
- Sprachmittel zur Spezifikation, Kommunikation und Dokumentation
  - zwischen Entwicklern
  - Entwicklern mit Anwendern
  - Vereinigung mehrerer Vorgänger-Methoden
- Standardisiert seit September 1997 von der OMG
- Entwickelt von

  Booch, Rumbaugh, Jacobson, Selic, Kobryn, Cook
  und vielen anderen ...
- Besteht aus:
  - Einer Menge von Modellierungskonzepten
  - Einer konkreten Notation

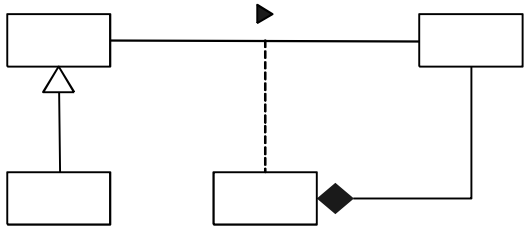
Ziele der UML

- Beschreibung wesentlicher Eigenschaften des Programms wie in einem Bauplan
- Strukturierung des Problems und der Lösung
- Abstraktion von Implementierungsdetails

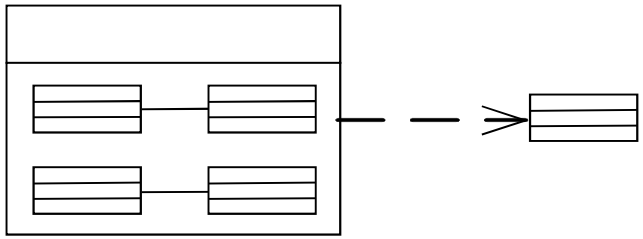
- Definition verschiedener Sichten:
  - Aufgabenverteilung und Workflows
  - Software/System-Architektur
  - Interaktion zwischen Komponenten
  - Verhalten von Komponenten
  - Implementierung
  - Physische Verteilung

Strukturelle Notationen der UML

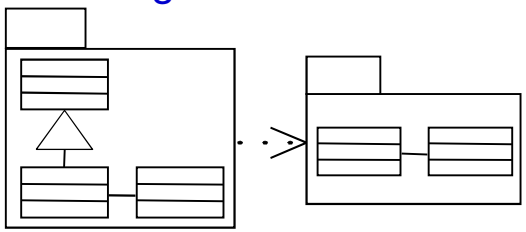
Klassendiagramm



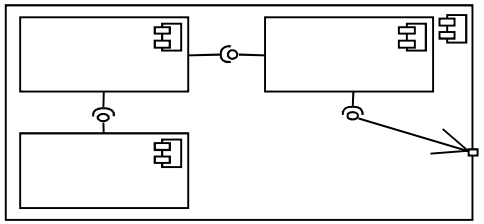
Kompositionsstrukturdiagramm



Paketdiagramm

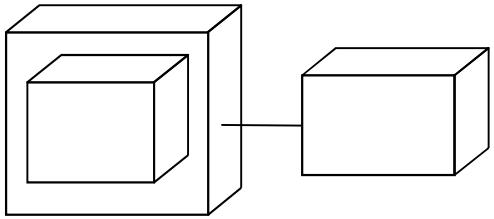


Komponentendiagramm



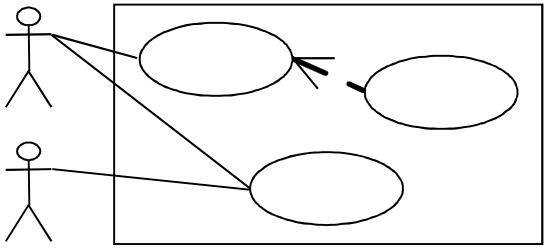
Objektdiagramm

Verteilungsdiagramm

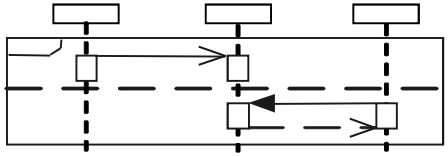


Verhaltensorientierte Notationen der UML

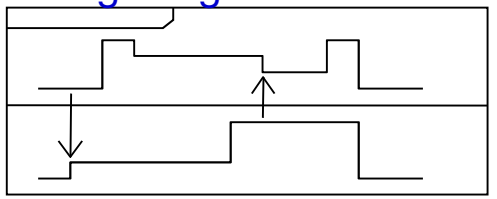
Use-Case-Diagramm



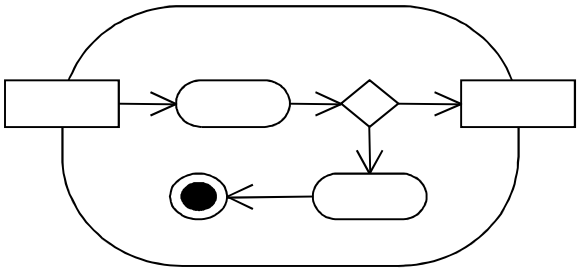
Sequenzdiagramm



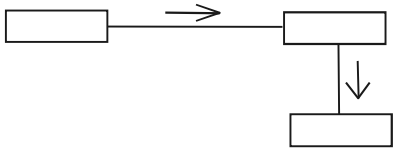
Timing-Diagramm



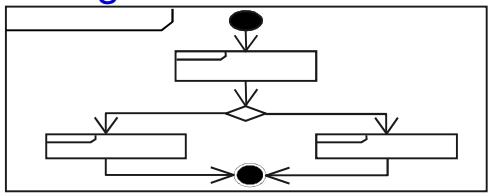
Aktivitätsdiagramm



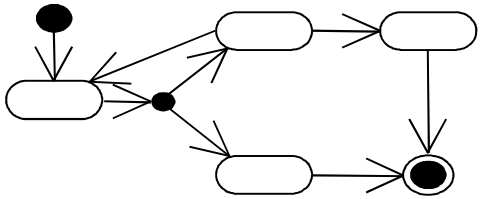
Kommunikationsdiagramm



Interaktionsübersichts-diagramm

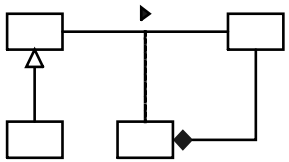
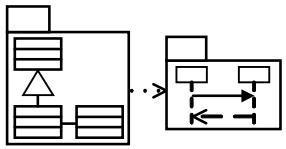
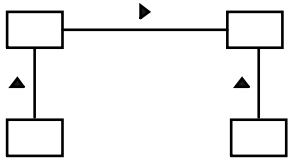


Zustandsautomat

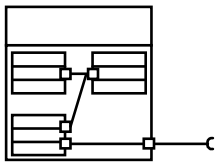
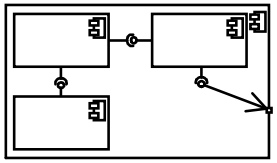
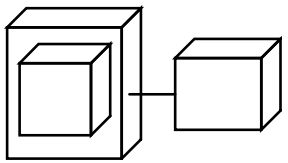


+ Textueller Teil:
  Object Constraint Language (OCL)

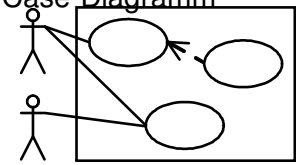
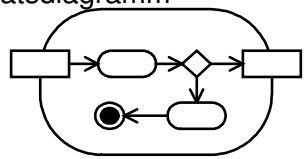
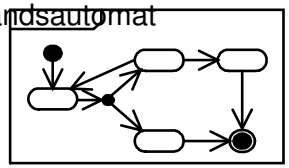
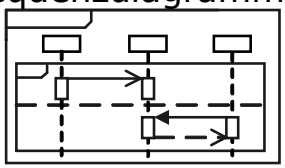
Diagrammtypen der UML 2 im Überblick
(von Mario Jeckle)

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 Struktur-zusammenhänge/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 (Klassendiagramm-schnappschuss)?	Zeigt Objekte u. Attributbelegungen zu einem bestimmten Zeitpunkt. Verwendung beispielhaft zur Veranschaulichung Detailniveau wie im Klassen-diagramm. Sehr gute Darstellung von Mengenverhältnissen.

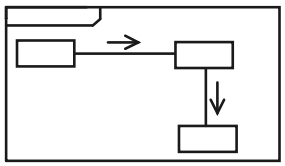
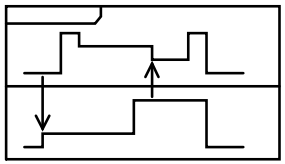
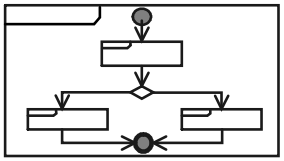
Die Diagrammtypen im Überblick - 2

Diagrammtyp	Diese zentrale Frage beantwortet das Diagramm	Stärken
Kompositionsstrukturdiagramm 	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 Abhängig-keiten 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.

Die Diagrammtypen im Überblick - 3

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 fluss-orientierter 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 Zustands-modells mit Zuständen, Ereignissen, Nebenläufigkeiten, Bedingungen, Ein- und Austrittsaktionen. Schachtelung möglich.
Sequenzdiagramm 	Wer tauscht mit wem welche Informationen in welcher Reihenfolge aus?	Darstellung des Informationsaustauschs zwischen Kommunikationspartnern Sehr präzise Darstellung der zeitlichen Abfolge auch mit Nebenläufigkeiten.

Die Diagrammtypen im Überblick - 4

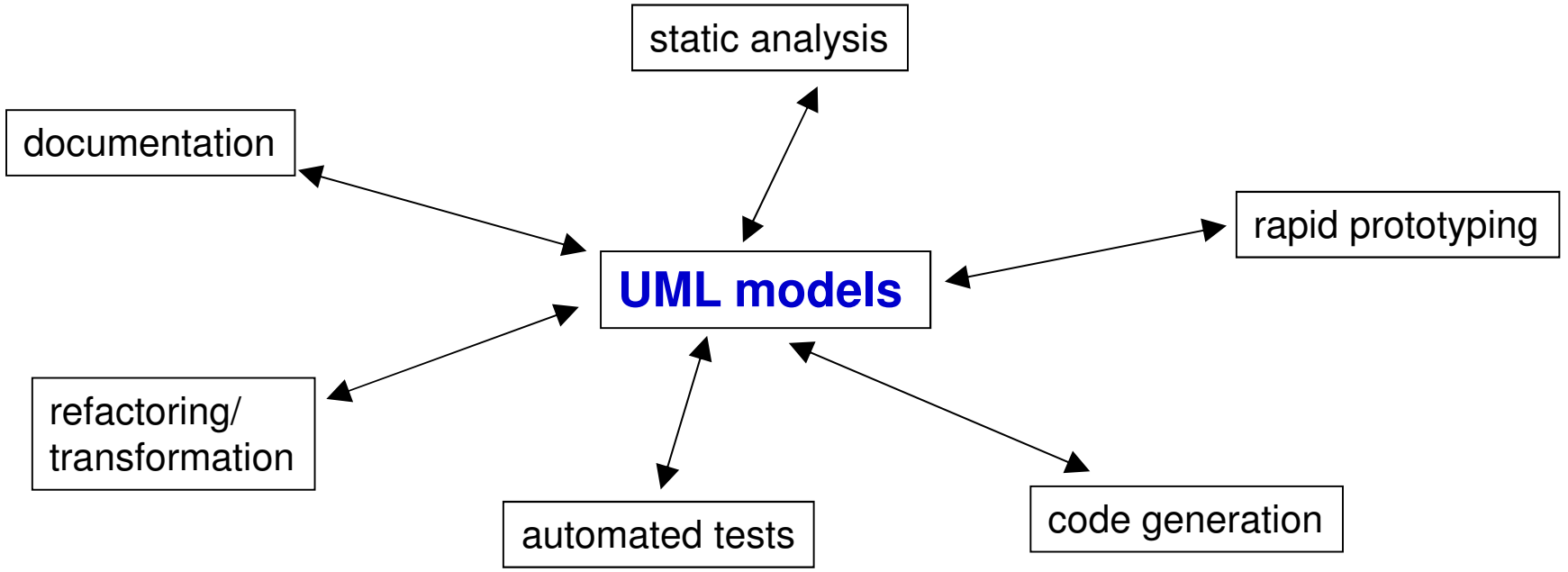
Diagrammtyp	Diese zentrale Frage beantwortet das Diagramm	Stärken
Kommunikations-diagramm 	Wer kommuniziert mit wem? Wer „arbeitet“ im System zusammen?	Stellt den Informationsaustausch zwischen Kommunikationspartnern dar. Überblick steht im Vordergrund (Details und zeitliche Abfolge weniger wichtig).
Timingdiagramm 	Wann befinden sich verschiedene Interaktionspartner 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.
Interaktionsübersichts-diagramm 	Wann läuft welche Interaktion ab?	Verbindet Interaktionsdiagramme (Sequenz-, Kommunikation- und Timingdiagramme) auf Top-Level-Ebene. Hohes Abstraktionsniveau.

Literatur zur UML

- UML 2.3 Beschreibung der OMG (www.omg.org):
  Notation Guide, Semantics, Metamodel, OCL, Summary
- Grady Booch, James Rumbaugh, Ivar Jacobson:
  UML User Guide (veraltet)
- Martin Fowler, Kendall Scott:
  UML Distilled
- Desmond D'Souza, Allan Wills:
  Objects, Components, and Frameworks with UML,
  The Catalysis Approach
- Mario Jeckle, Chris Rupp, Jürgen Hahn, Barbara Zengler, Stefan Queins
  UML 2.0 glasklar
- Martin Hitz, Gerti Kappel
  UML @ Work
- **Bernhard Rumpe**
  Modellierung mit UML, Springer Verlag (zwei Bücher).

Modellbasierte Entwicklung mit der UML

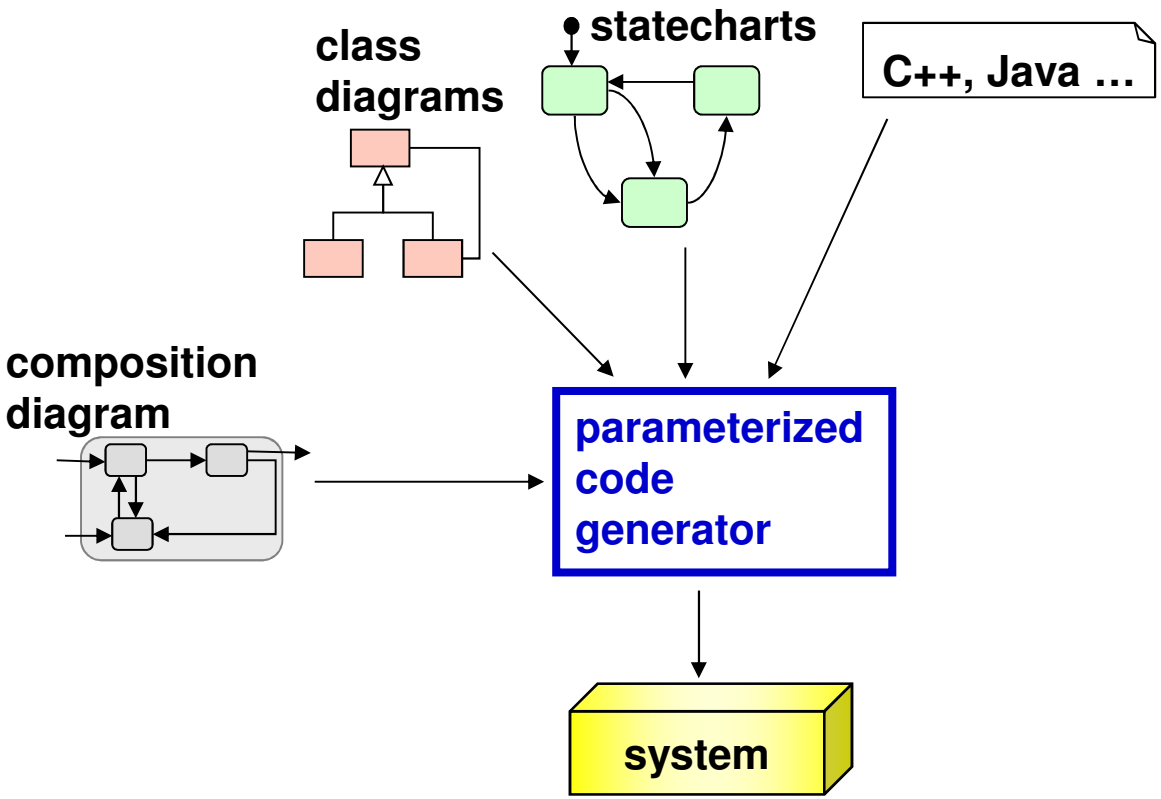
- Models as central notation



- UML serves as central notation for development of software
- UML is programming, test and modelling language at the same time

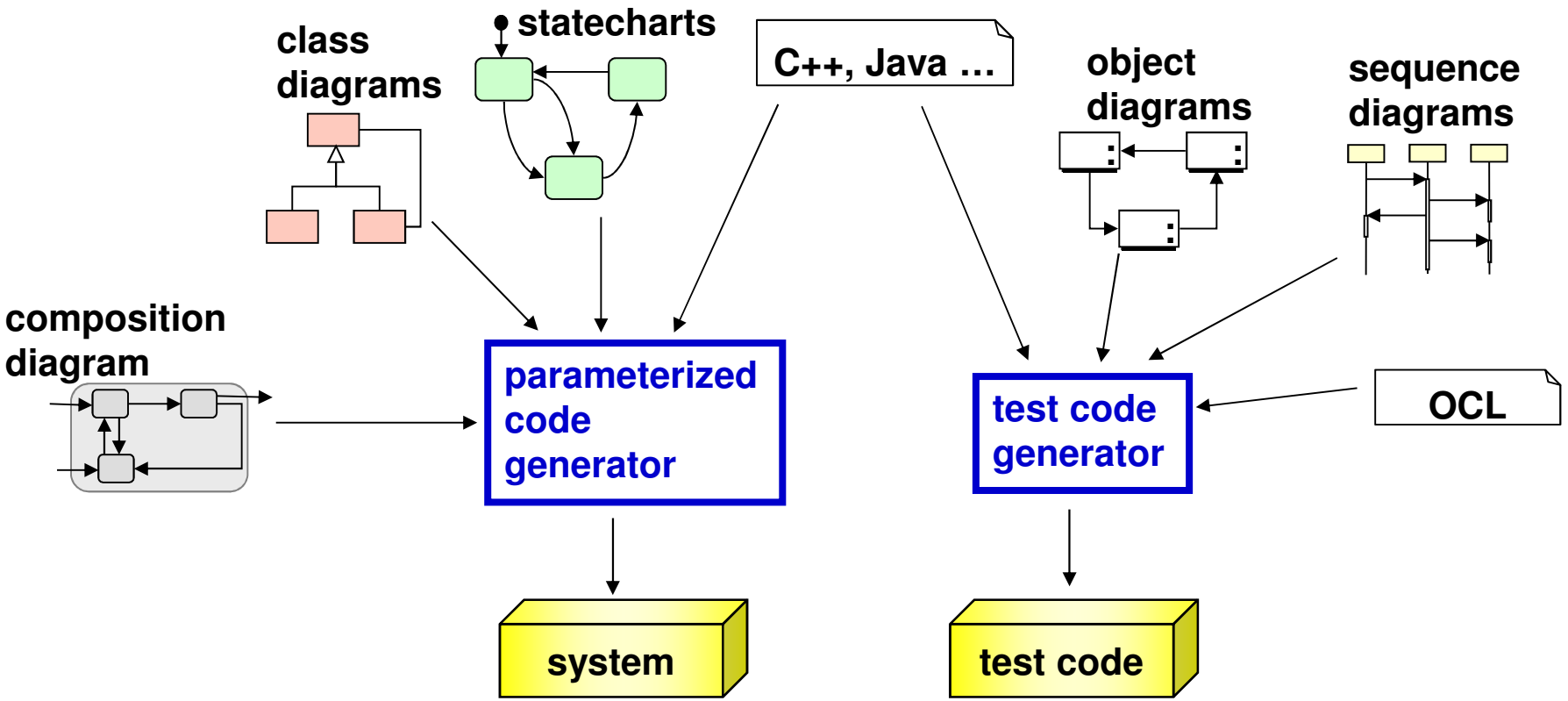
UML-basierte Modellierung

- UML + Code-Rümpfe erlauben Code-Generierung



UML-basierte Modellierung

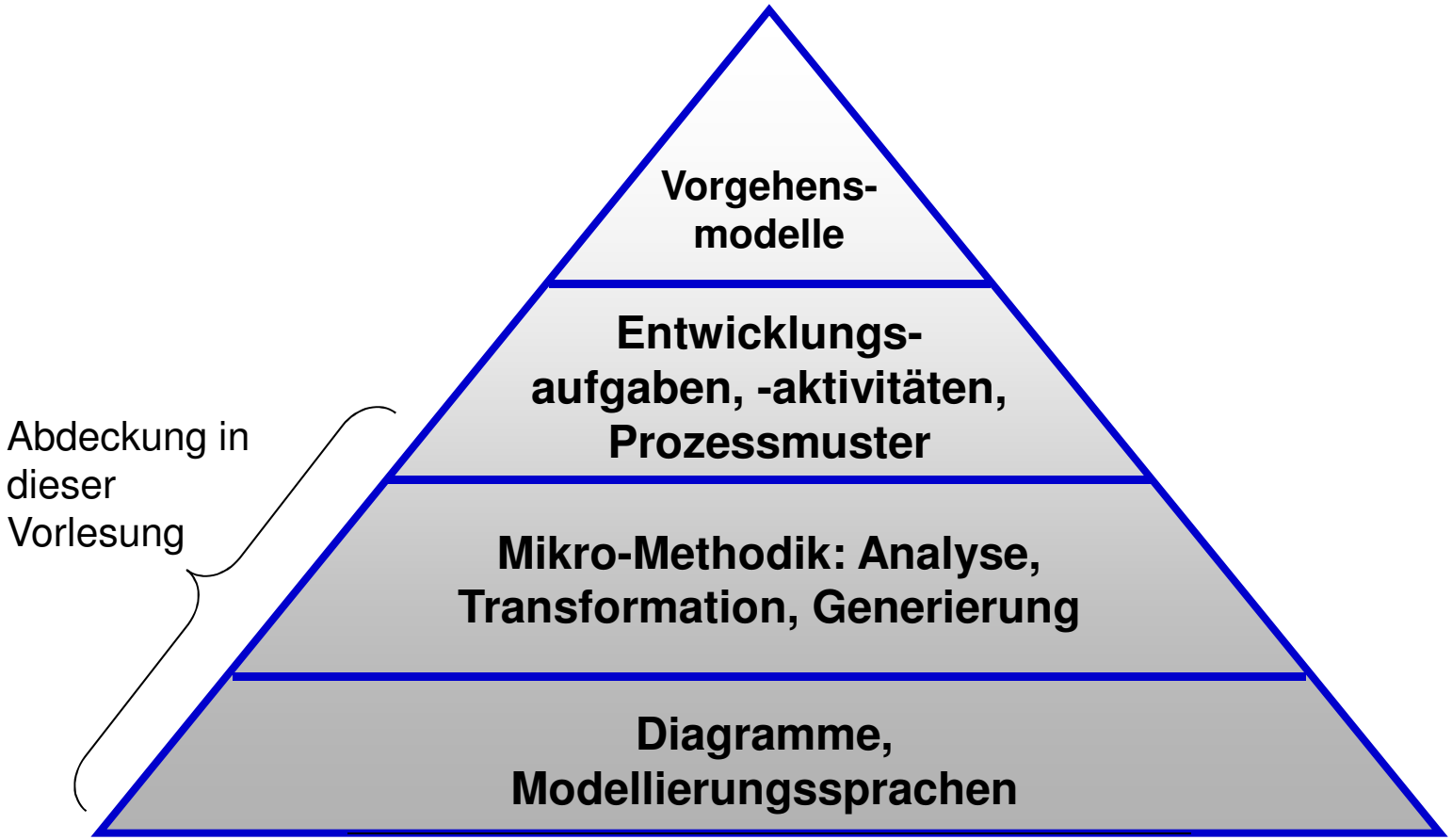
- UML + Code-Rümpfe erlauben Code & Test-Modellierung



⇨ Code- und Testmodelle prüfen gegenseitige Korrektheit

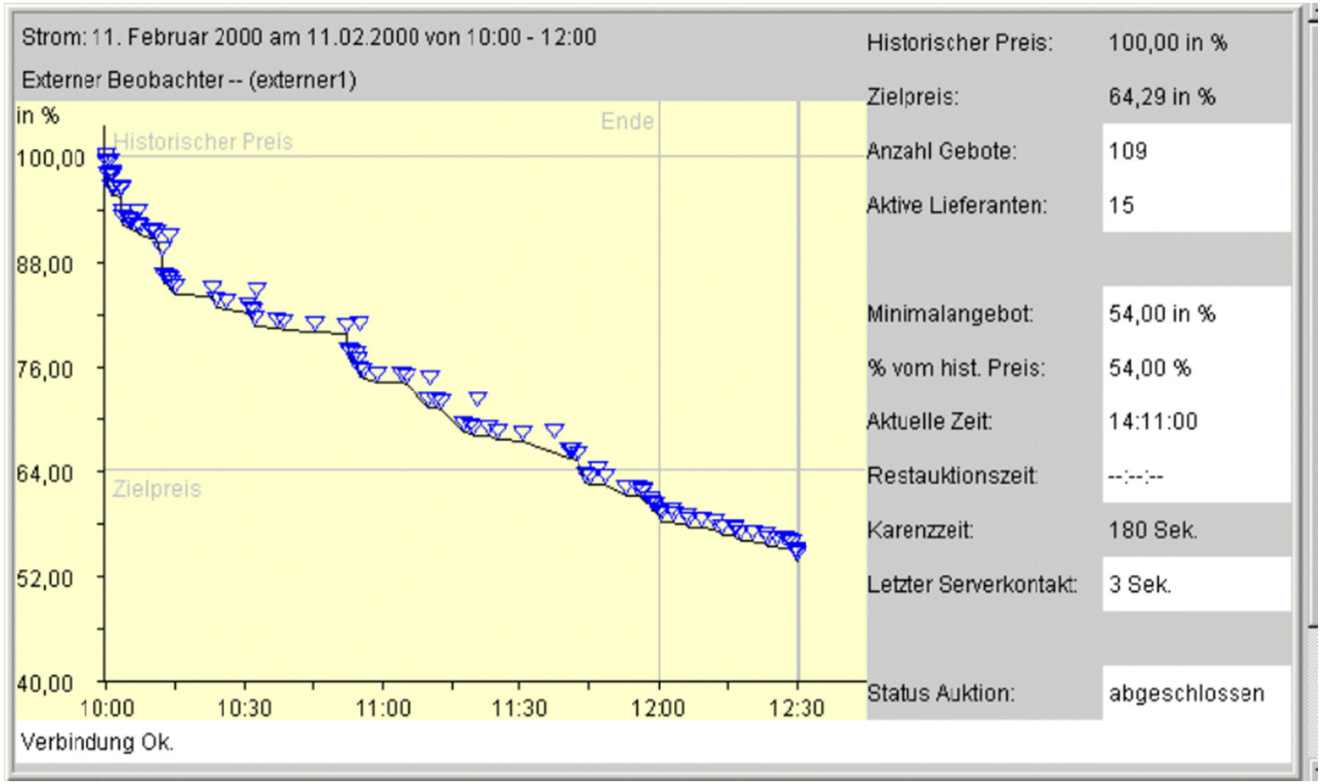
Grundlagen der Modellbildung

- Die Methodik-Pyramide:



Laufendes Beispiel: Online Auktionssystem

- Charakteristika:
  - Mehrere Anbieter bewerben sich um einen Liefervertrag
  - Echtzeitauktion mit ca. 2h Dauer



- Im Beispiel:
- Auktion des Jahres-Strombedarfs einer Großbank mit
- 46% Kostenreduktion

Ausschnitt des Applets in einem Browser

Zusammenfassung 1

- Modellbasierte Softwareentwicklung
  - nutzt Modelle als zentrales Artefakt in der Softwareentwicklung:

- Ein Modell gehört zu einem Original, ist eine Abstraktion des Originals und hat einen auf das Original bezogenen Einsatzzweck

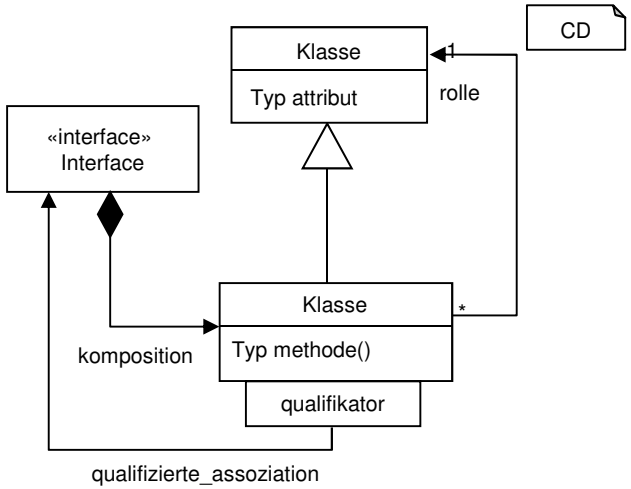
- Die UML ist Industriestandard bei der Modellierung von Softwaresystemen

- Verschiedene Sichten der UML dienen zur
  - Analyse von Eigenschaften des Originals oder zur
  - konstruktiven Generierung von Code oder Tests

SE SOFTWARE ENGINEERING

RWTH AACHEN

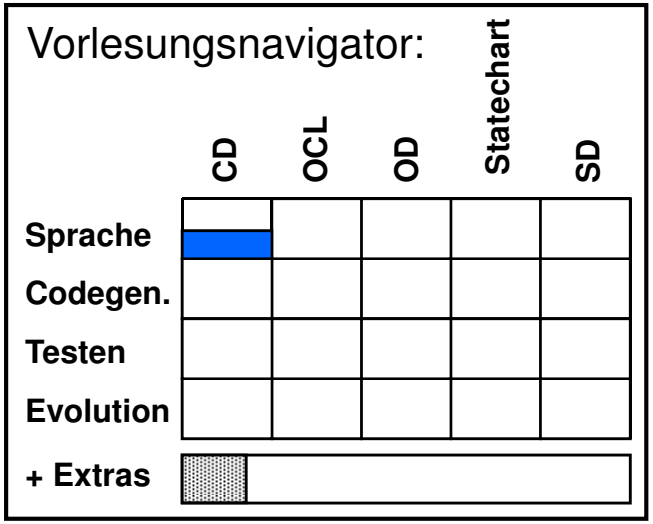
Modellbasierte Softwareentwicklung



- 2. Strukturmodellierung mit Klassendiagrammen
- 2.1. Klassen

Prof. Dr. Bernhard Rumpe
Lehrstuhl für Software Engineering
RWTH Aachen

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



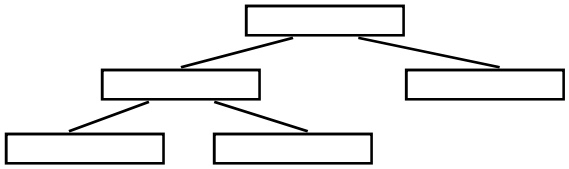
Grundkonzepte der Objektorientierung

- Ein System besteht aus variabel vielen Objekten.
- Ein Objekt hat ein definiertes Verhalten.
  - Menge genau definierter Operationen
  - Operation wird beim Empfang einer Nachricht ausgeführt.
- Ein Objekt hat einen inneren Zustand.
  - Zustand des Objekts ist Privatsache (Kapselungsprinzip).
  - Resultat einer Operation hängt vom aktuellen Zustand ab.
- Ein Objekt hat eine eindeutige Identität.
  - Identität ist fest und unabhängig von anderen Eigenschaften.
  - Es können mehrere verschiedene Objekte mit identischem Verhalten und identischem inneren Zustand im gleichen System existieren.

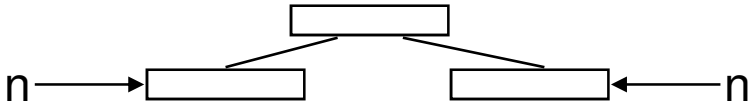
Konzepte der Objektorientierung

- Ein Objekt gehört zu einer Klasse.
  - Die Klasse schreibt das Verhaltensschema und die innere Struktur ihrer Objekte vor.

- Klassen besitzen einen ‘Stammbaum’, in der Verhaltensschema und innere Struktur durch Vererbung weitergegeben werden.
  - Vererbung bedeutet Generalisierung einer Klasse zu einer Oberklasse.



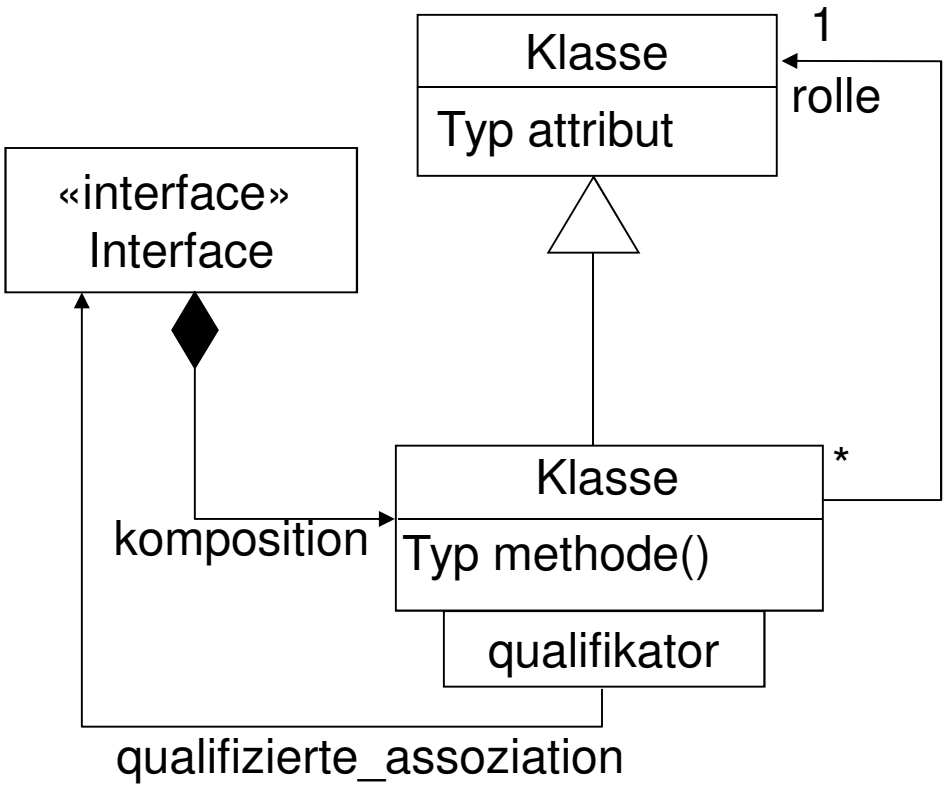
- Polymorphie**:** Eine Nachricht kann verschiedene Reaktionen auslösen, je nachdem zu welcher Unterklasse einer Oberklasse das empfangende Objekt gehört.



Beispiel eines Klassendiagramms

Es handelt sich um ein Klassendiagramm „class diagram" (CD)

CD

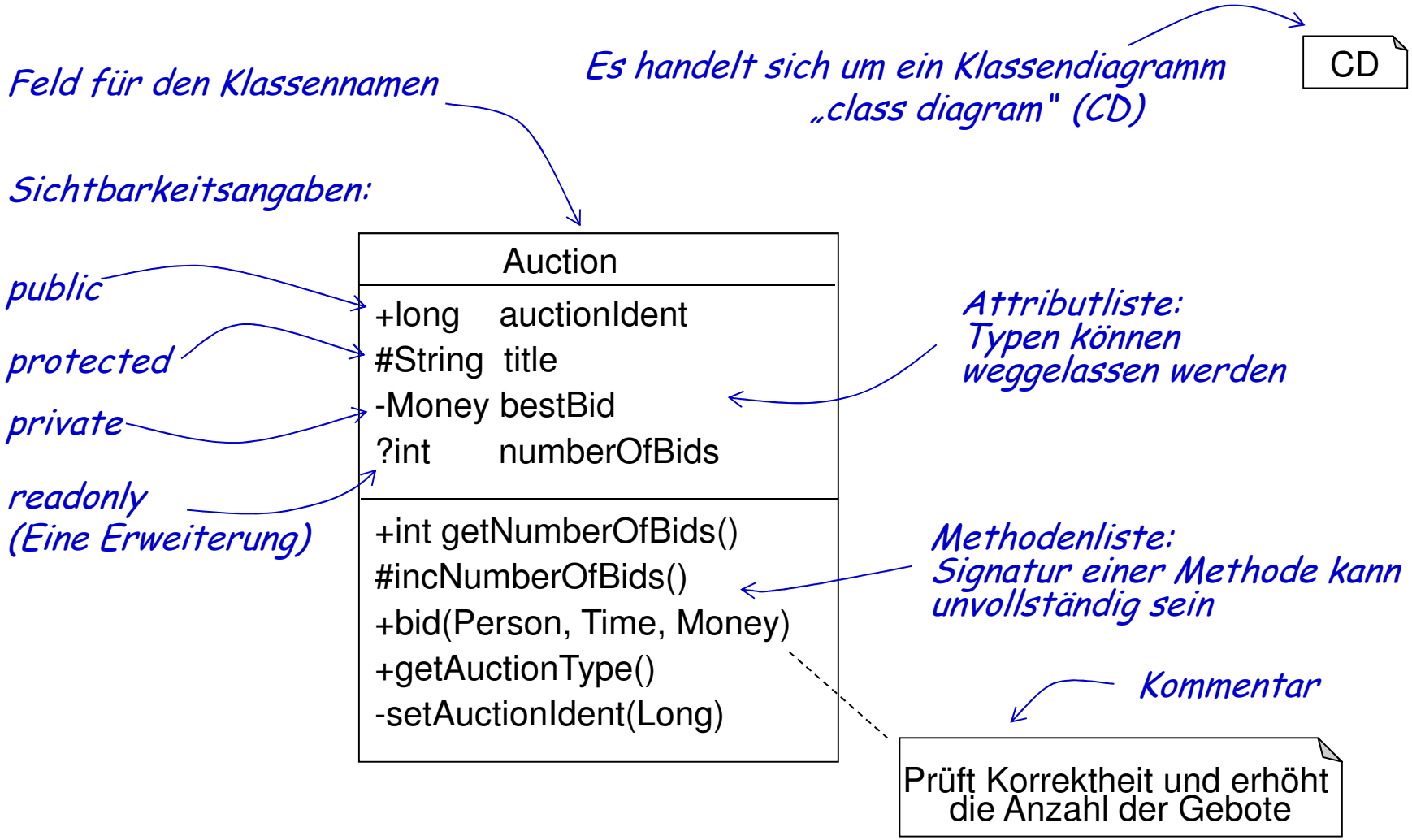


Aufgaben einer Klasse

1. Kapselung von Attributen und Methoden zu einer konzeptuellen Einheit
2. Ausprägung von Instanzen als Objekte
3. Typisierung von Objekten
4. Extension (Menge aller zu einem Zeitpunkt existierenden Objekte)
5. Charakterisierung der möglichen Strukturen eines Systems

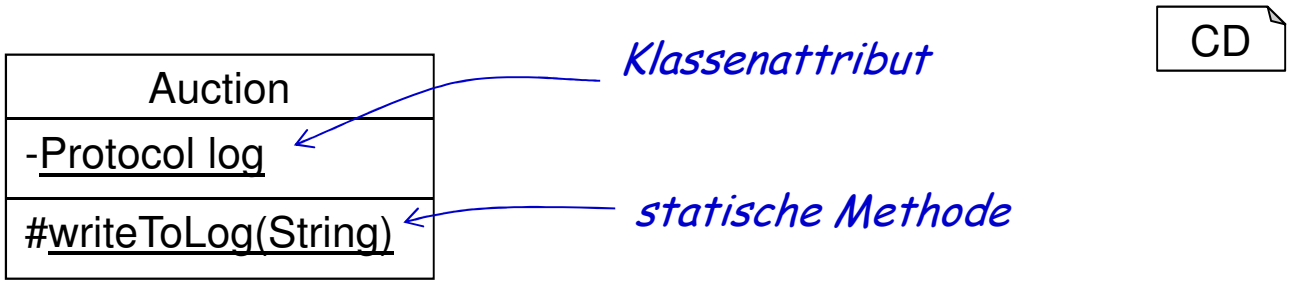
6. Konzeptuelle Modellierung des Anwendungsgebiets
7. Implementierungsbeschreibung
8. Klassencode (die übersetzte, ausführbare Form der Implementierungsbeschreibung)

Klasse mit Attributen und Methoden



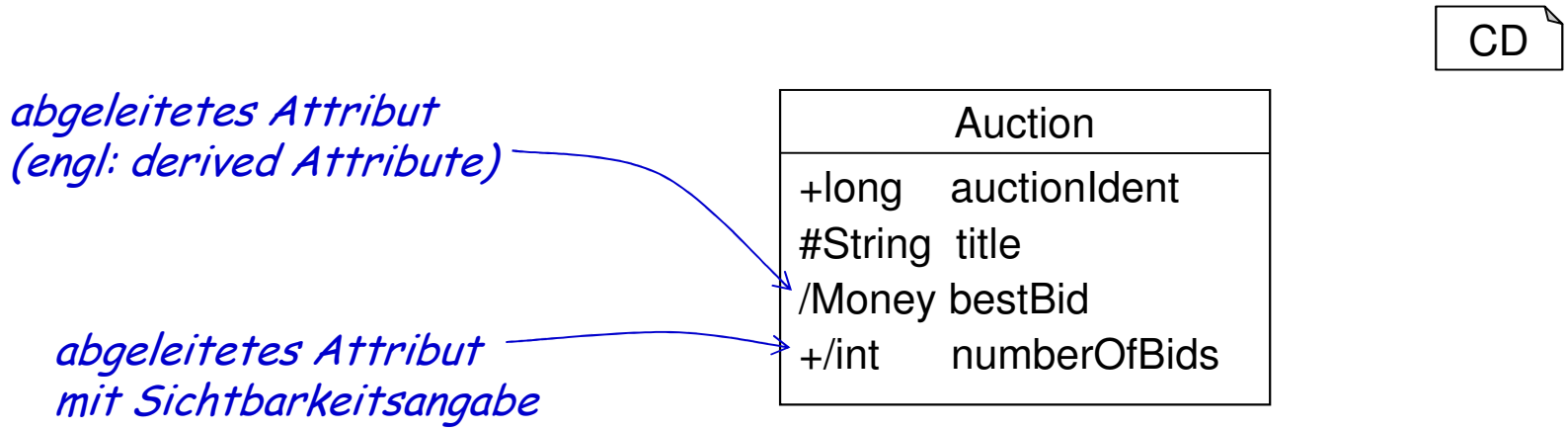
Klassenattribute und Methoden

- Bestimmte Attribute und Methoden gehören nicht zu einzelnen Objekten, sondern der Klasse:
- Java bietet hierfür das „static“ Schlüsselwort, in UML werden solche Elemente unterstrichen.
- Auch Konstruktoren sind statische Elemente, die zur Klasse gehören!
- Methodische Richtline: so wenig wie möglich einsetzen!



Abgeleitete Attribute

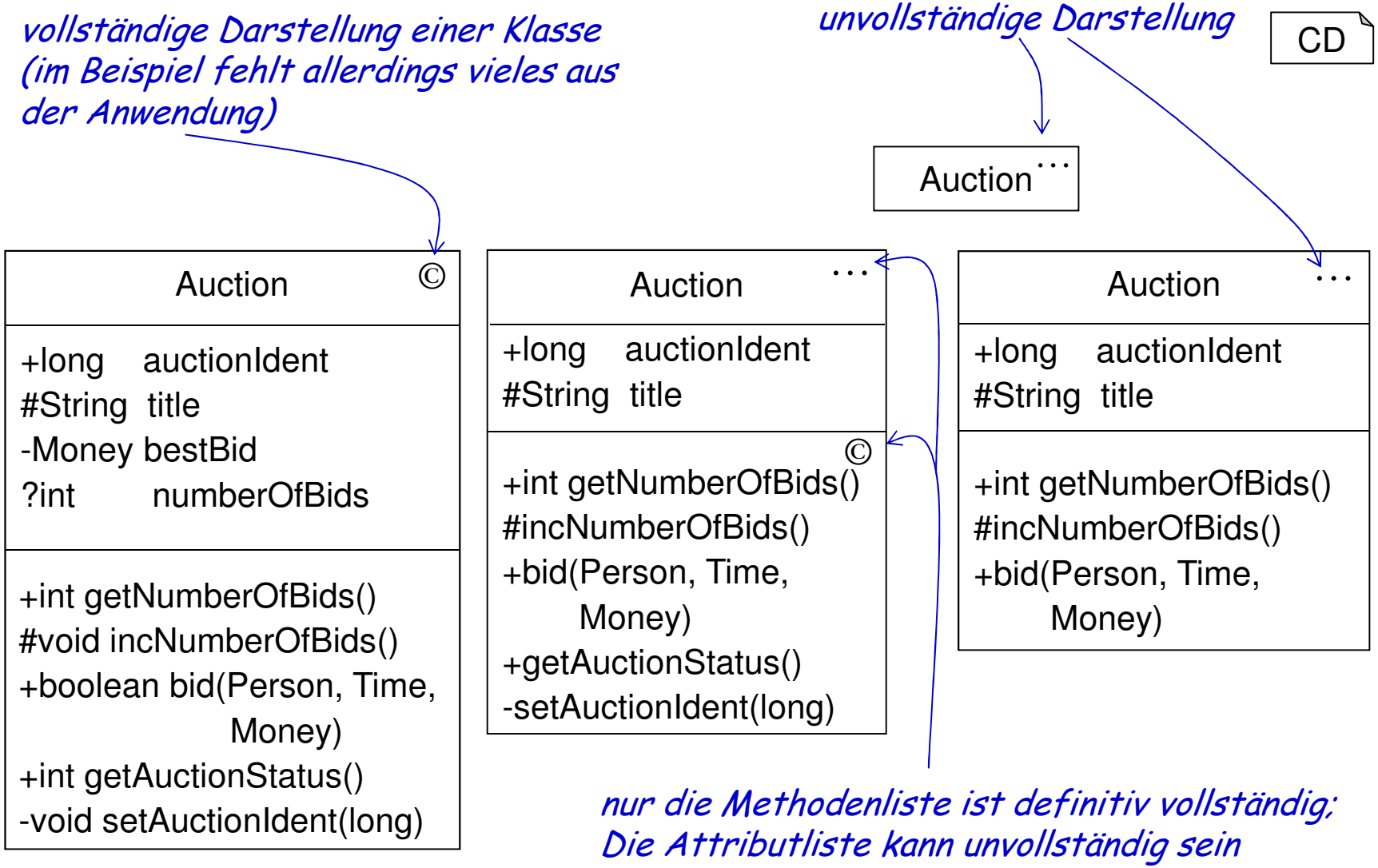
- Wenn ein Attribut aus anderen berechnet (abgeleitet) werden kann, dann Kennzeichnung mit der Markierung „/“.
- Sinnvoll ist dann meist, die Beziehung (Berechnung) des Attributs ebenfalls zu notieren:
  - numberOfBids == bidList.length()



Vollständigkeit der Darstellung?

vollständige Darstellung einer Klasse (im Beispiel fehlt allerdings vieles aus der Anwendung)

unvollständige Darstellung

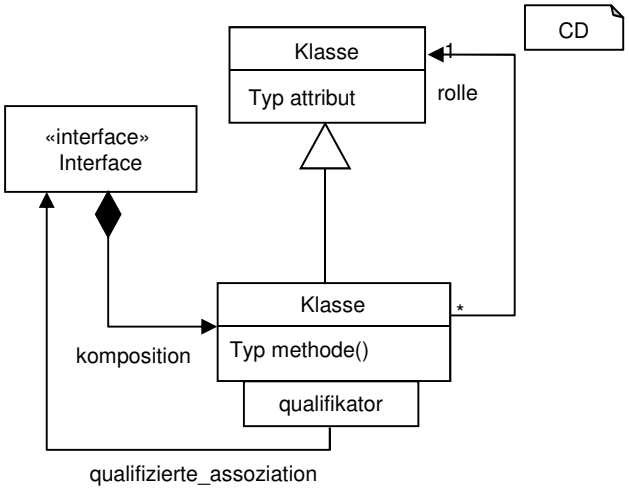


nur die Methodenliste ist definitiv vollständig; Die Attributliste kann unvollständig sein

SE SOFTWARE ENGINEERING

RWTH AACHEN

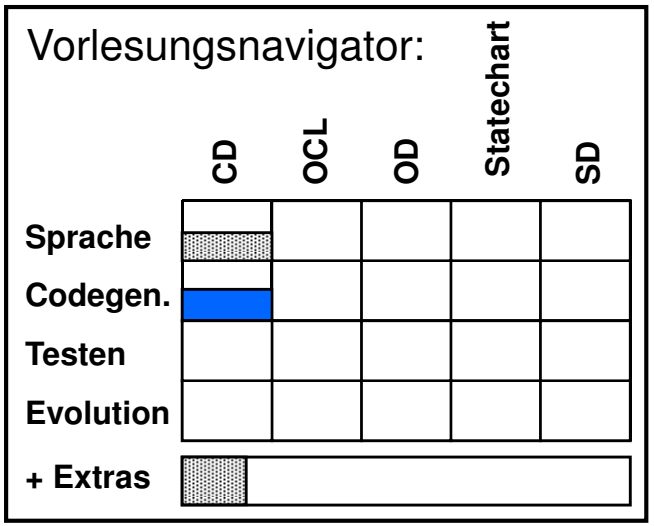
Modellbasierte Softwareentwicklung



- 2. Strukturmodellierung mit Klassendiagrammen
- 2.2. Codegenerierung

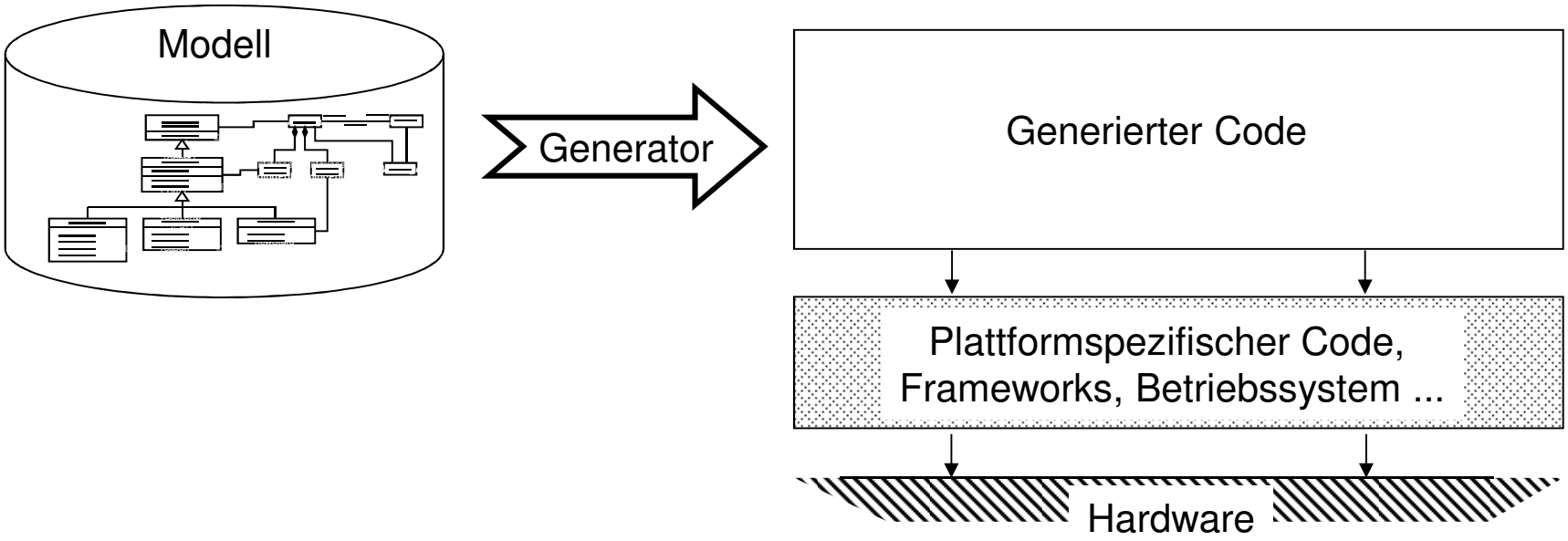
Prof. Dr. Bernhard Rumpe
Lehrstuhl für Software Engineering
RWTH Aachen

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



Code-Generierung

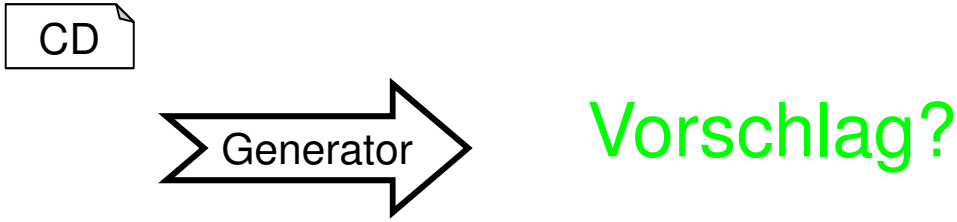
- Prinzip: Abbildung des Modells in eine Programmiersprache



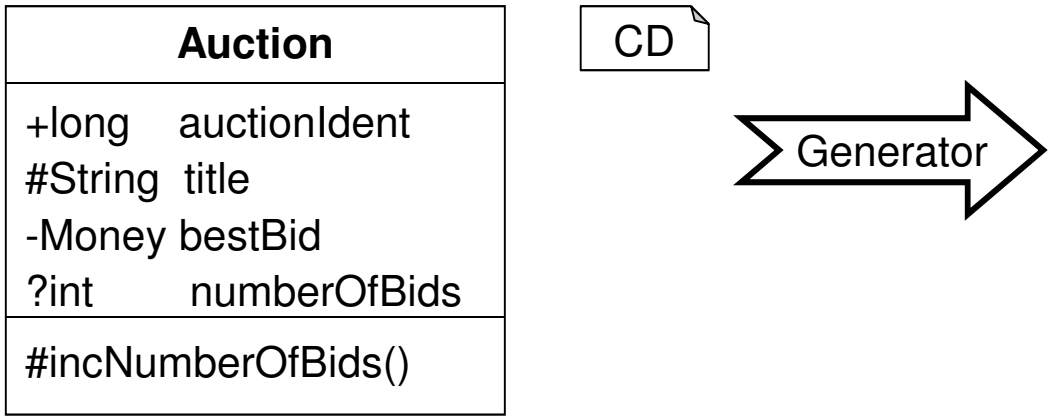
⇨ Selbst wenn kein „automatischer“ Generator zur Verfügung steht, können die Transformationen von UML in Code eingesetzt werden.

Code-Generierung aus einer Klasse

Auction
+long auctionIdent #String title -Money bestBid ?int numberOfBids
#incNumberOfBids()



Code-Generierung aus einer Klasse



JAVA

```
class Auction {
  public long      auctionIdent;
  protected String title;
  private Money    bestBid;
  public int       numberOfBids;

  protected void incNumberOfBids() { ... }
}
```

Probleme der Codegenerierung:

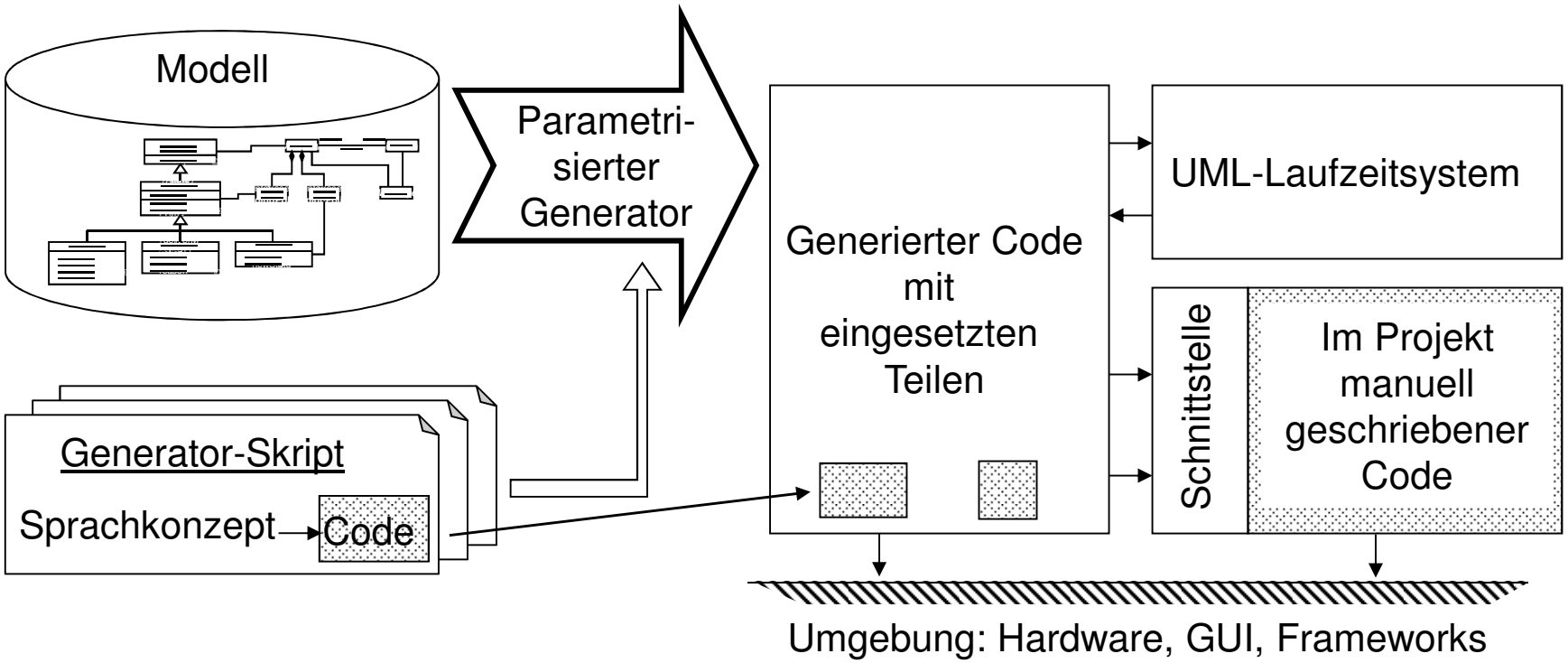
- Klassendiagramm enthält keine Methodenrümpfe?
  - Wie werden diese ergänzt?
- Diagramm ist unvollständig
  - nicht alle Attribute, ...
- Alternative Generierungsformen?
- Diagramm ist inkonsistent
  - Ungültiger Datentyp, Attributname doppelt, ...

Alternative Code-Generierung?

- Mögliche Anforderungen:
  - get/set-Methoden für Attribute
  - Serialisierbarkeit des Objekts
  - Speichern von Objekten in einer Datenbank-Tabelle
    - evtl. sogar die Erzeugung der Tabelle als SQL-Statement
  - Attributzugriff wird durch Security-Manager gesichert
  - Plattformabhängigkeit des Codes

- Unterschiedliche Anforderungen führen zu unterschiedlichen Generatoren
  - Technik 1: Parametrisierung des Generators
  - Technik 2: Generierung gegen eine abstrakte Schnittstelle:
    Bereitstellung eines Laufzeitsystems
    (ähnlich der Java Virtual Machine)

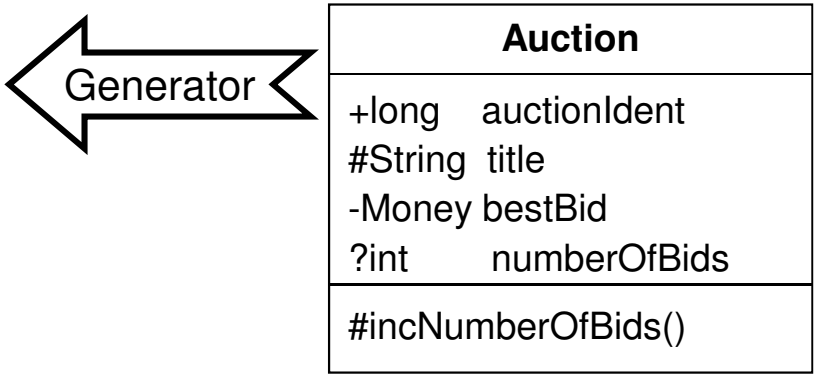
Codegenerator: Parametrisierung + Laufzeitsystem



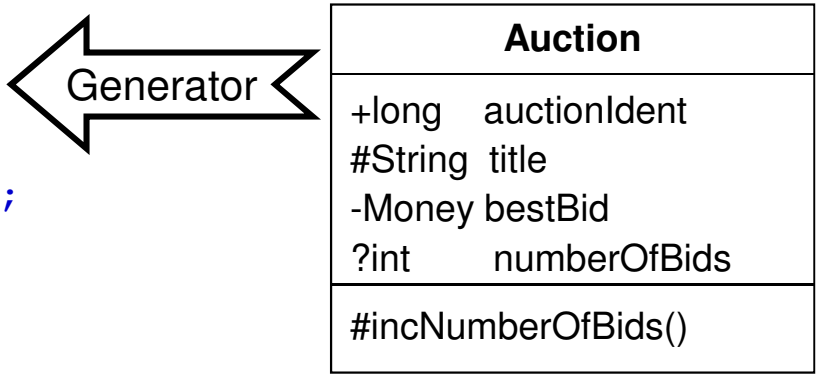
prinzipielle Struktur des generierten Codes
mit Parametrisierung, manuellen Anteilen und einem „Laufzeitsystem"

Code-Generierung mit get/set-Methoden

```
class Auction {




}
```

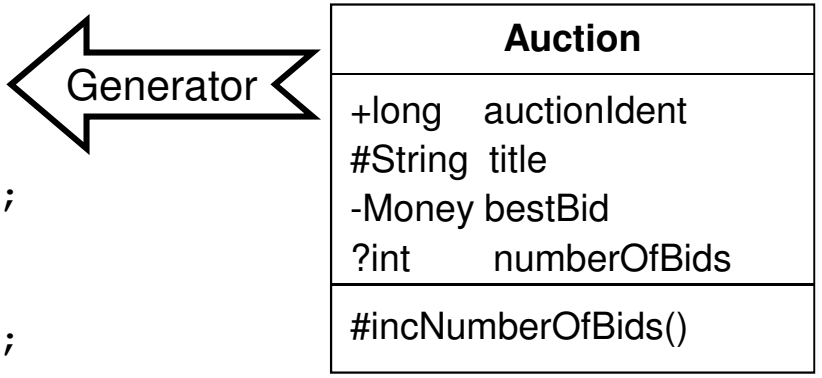


Code-Generierung mit get/set-Methoden




```
class Auction {
  private long   _auctionIdent;

  synchronized public long getAuctionIdent() { return _auctionIdent; }

  synchronized public void setAuctionIdent(long x)  { _auctionIdent =x; }

}
```

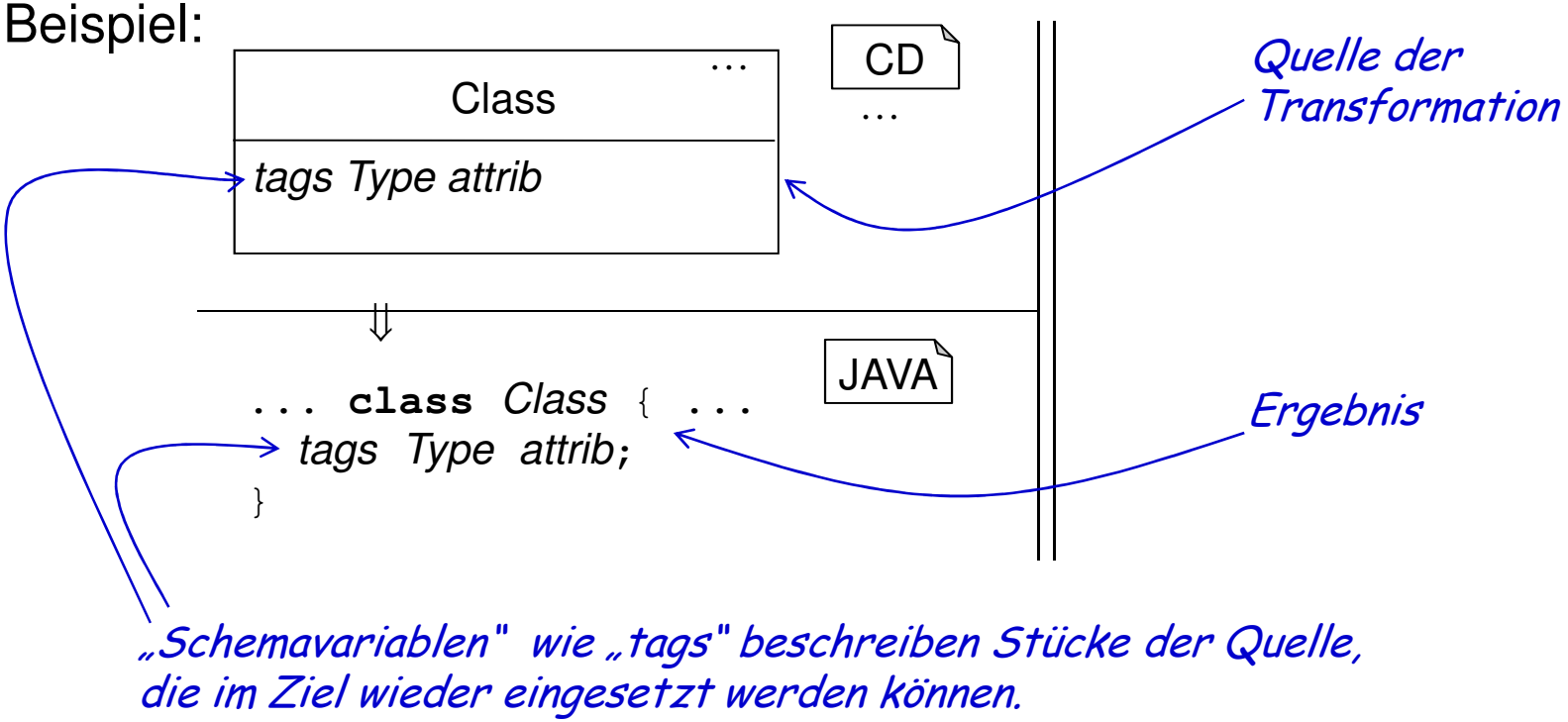
Code-Generierung mit get/set-Methoden




```
class Auction {
  private long   _auctionIdent;
  private String _title;
  private Money  _bestBid;
  private int    _numberOfBids;

  synchronized public long getAuctionIdent() { return _auctionIdent; }
  synchronized protected String getTitle()   { return _title; }
  synchronized private Money getBestBid()    { return _bestBid; }
  synchronized public int getNumberOfBids()  { return _numberOfBids; }

  synchronized public    void setAuctionIdent(long x) { _auctionIdent=x;}
  synchronized protected void setTitle(String x)      { _title =x; }
  synchronized private   void setBestBid(Money x)     { _bestBid =x; }
  synchronized protected void setNumberOfBids(int x)  { _numberOfBids =x;
  }

  synchronized protected void incNumberOfBids() {
    setNumberOfBids(getNumberOfBids()+1);
  }
}
```

Skriptdarstellung für Codegenerierung

- Beispiel:

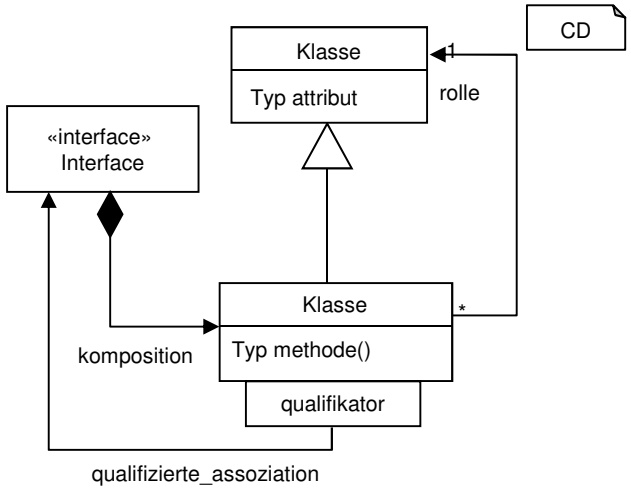


- Gegebenenfalls weitere Transformationen zur Beschreibung der Anpassung einzelner Teile.
- Abhängig von tags („/“, „+“ etc.) können verschiedene Übersetzungsregeln notwendig sein.
- Form der Skripte in Werkzeugen höchst unterschiedlich!

SE SOFTWARE ENGINEERING

RWTH AACHEN

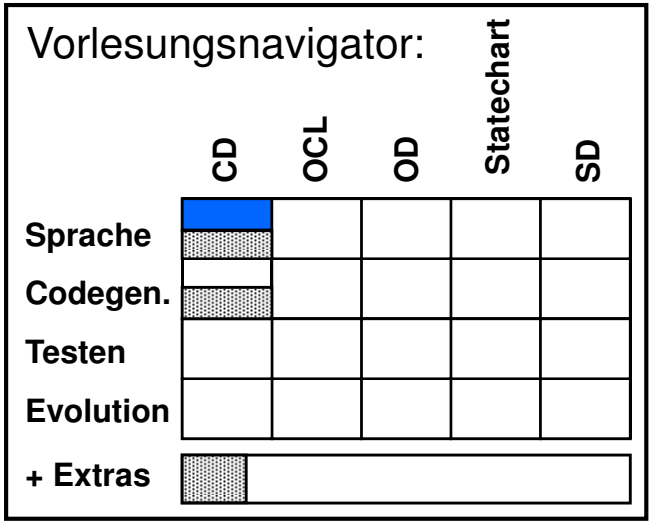
Modellbasierte Softwareentwicklung



- 2. Strukturmodellierung mit Klassendiagrammen
- 2.3. Interfaces, Vererbung, Assoziationen

Prof. Dr. Bernhard Rumpe
Lehrstuhl für Software Engineering
RWTH Aachen

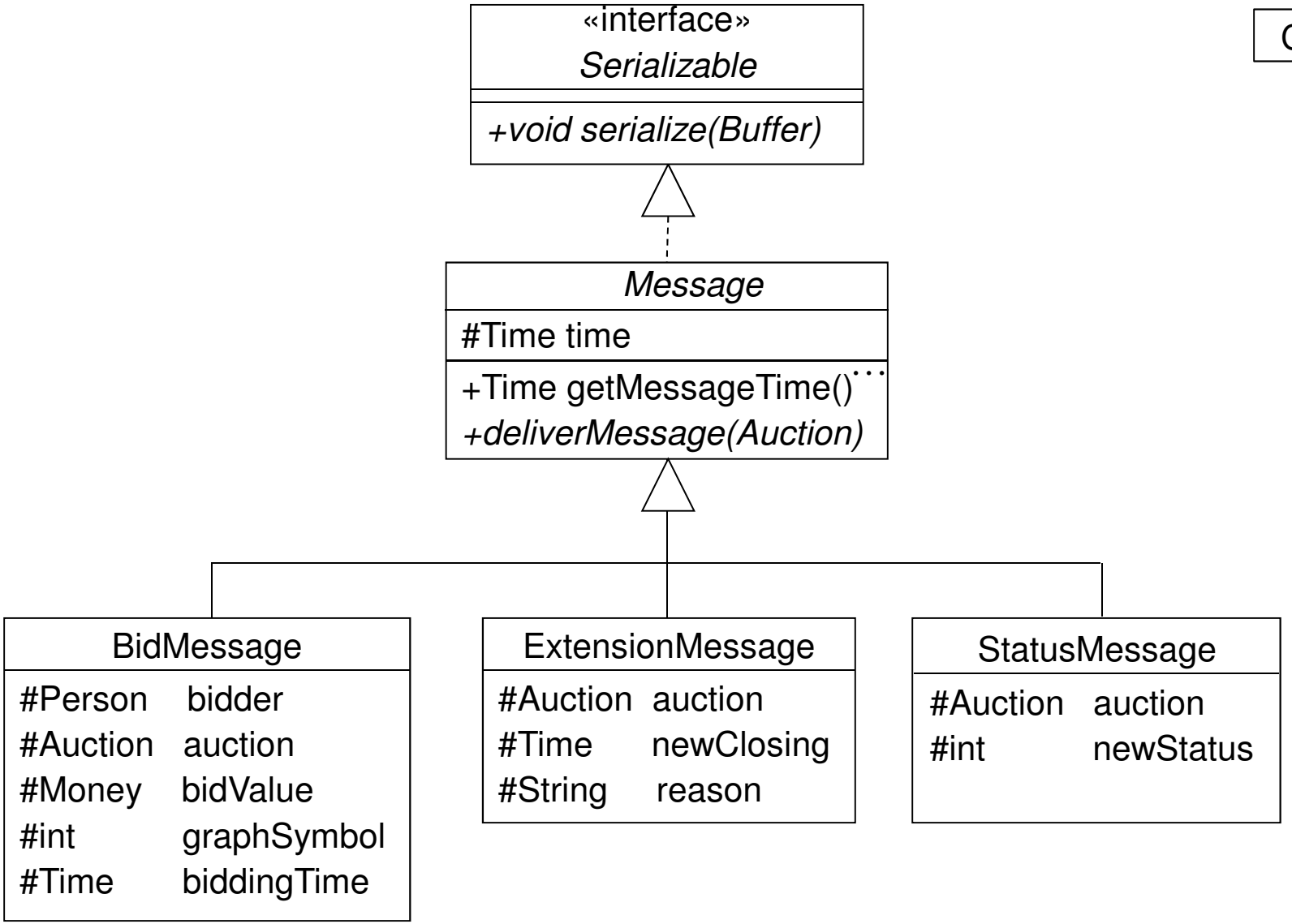
http://mbse.se-rwth.de/



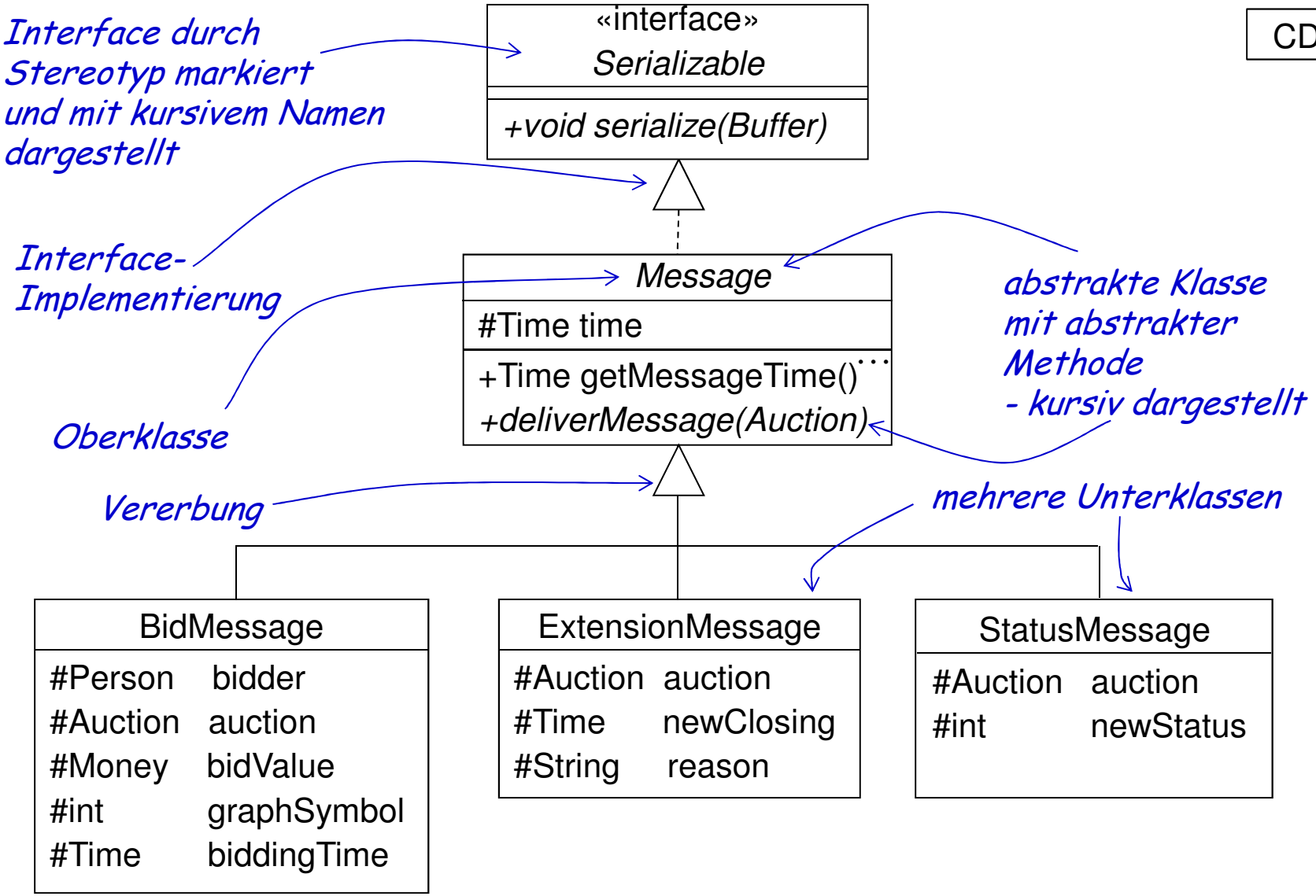
Vererbung, Schnittstellen-Implementierung

Analyze

CD

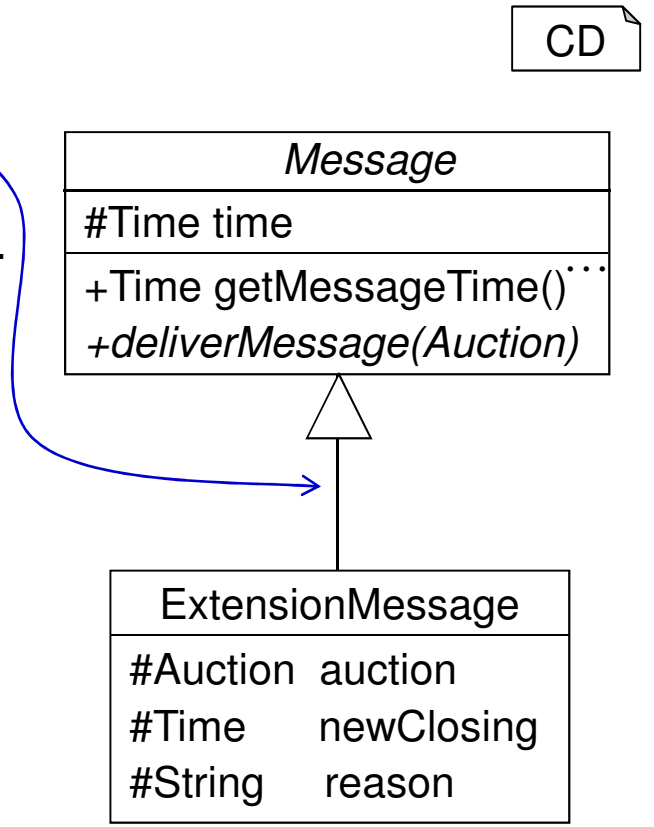


Vererbung, Schnittstellen-Implementierung



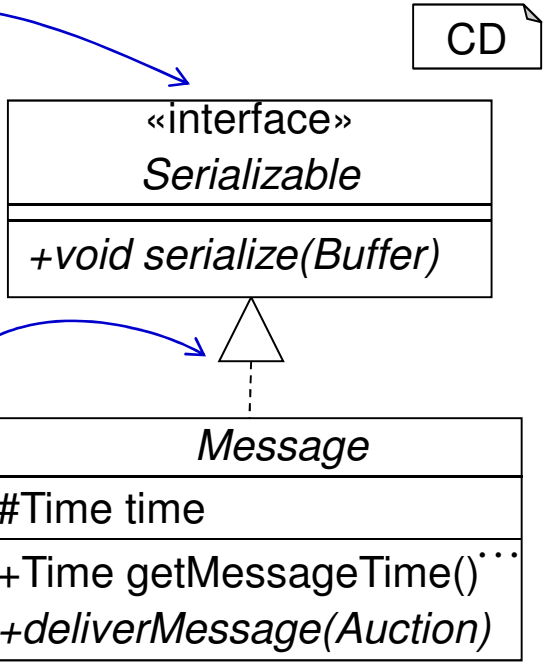
Vererbung

- Technische Sicht:
  - Vererbung zwischen je zwei Klassen
  - Attribute und Methoden werden von Superklasse an Subklasse weiter gegeben.
  - Attribute und Methoden können hinzugefügt werden
  - Methoden dürfen überschrieben werden
- Bedeutung:
  - Mittel zur hierarchischen Strukturierung
  - Subklasse ist ein Subtyp
  - Subklasse beschreibt eine Teilmenge der Objekte der Oberklasse
  - Substitutionsprinzip:
    - Instanzen der Unterklasse sind dort einsetzbar, wo Instanzen der Oberklasse erlaubt sind.



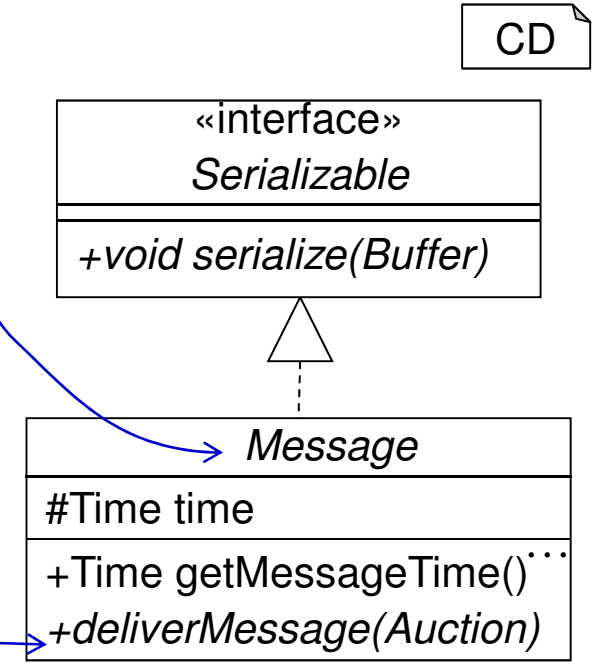
Interfaces, Schnittstellen

- Interface:
  - Ein Interface (Schnittstelle) beschreibt die Signaturen einer zusammengehörenden Sammlung von Methoden.
  - Anders als bei Klassen:
    - keine Attribute (nur Konstanten)
    - keine Methodenrümpfe angegeben.
  - Interfaces besitzen ebenfalls eine Vererbung (Erweiterung)
- Klassen implementieren Interfaces
  - ähnlich zur Vererbung
- Methodischer Einsatz:
  - Strukturierung von Schnittstellen



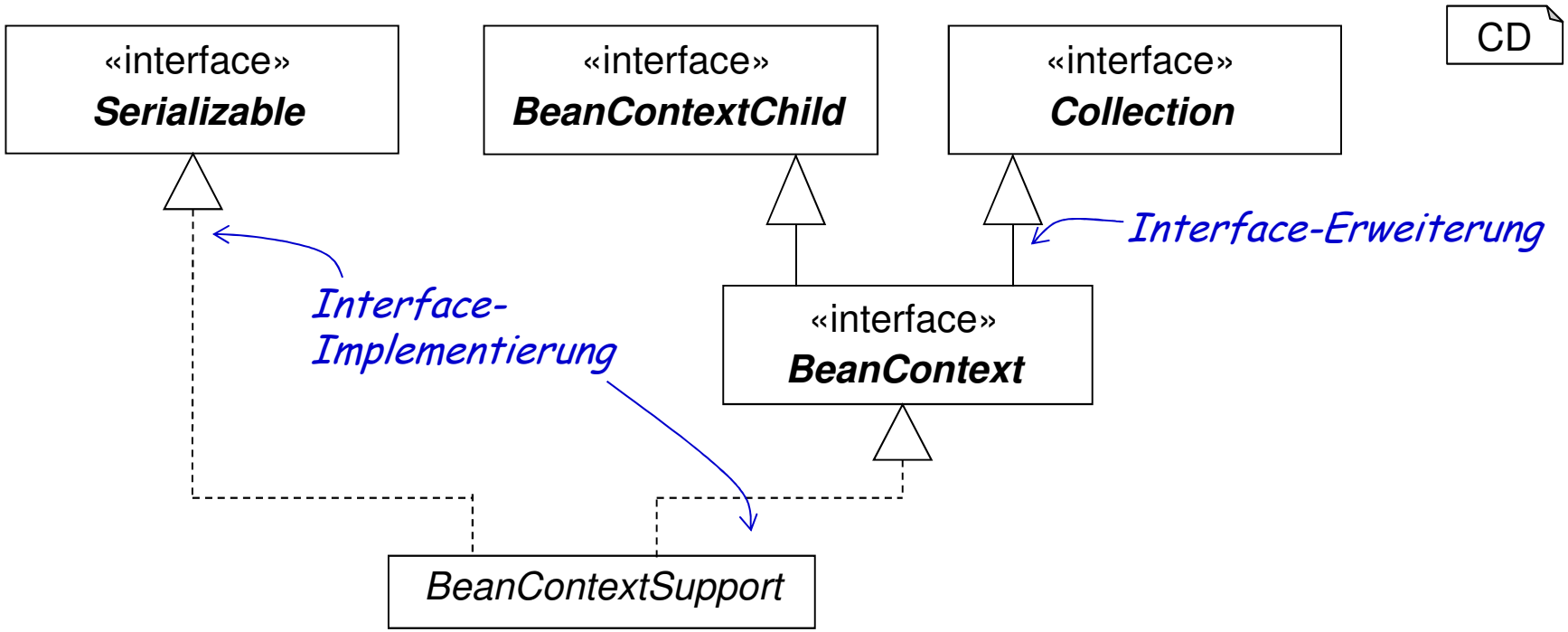
Abstrakte Klasse

- Abstrakte Klasse
  - Darstellung: kursiv
  - bildet eine Mischform zwischen Interface und „normaler“ Klasse
  - Implementierung durch Methodenrümpfe und Attribute sind teilweise vorhanden
  - Abstrakte Methoden ohne Implementierung
  - Aber: Bildung von Instanzen (Objekten) nicht möglich
- UML erlaubt, anders als Java:
  - Klassen erben von mehreren Klassen



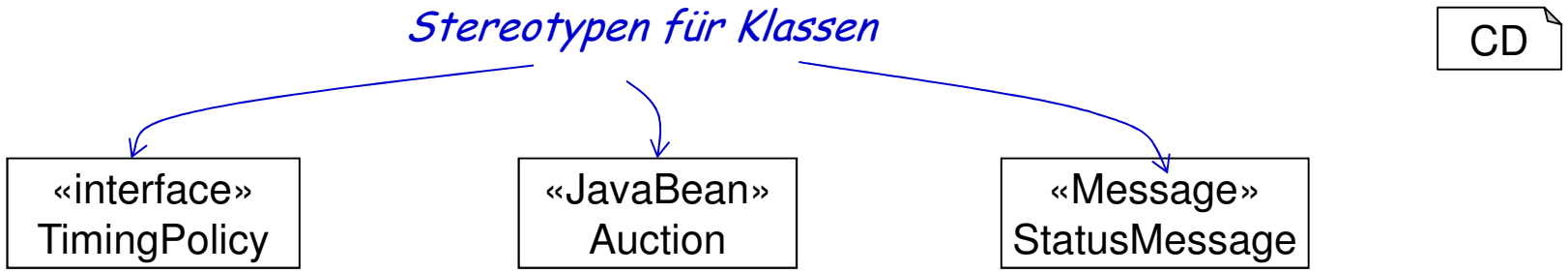
Interface Implementierung

- Ein Interface kann viele andere Interfaces erweitern
- Eine Klasse kann viele Interfaces implementieren
- UML: Eine Klasse kann von vielen Klassen erben
  - Java: von nur einer Klasse erben



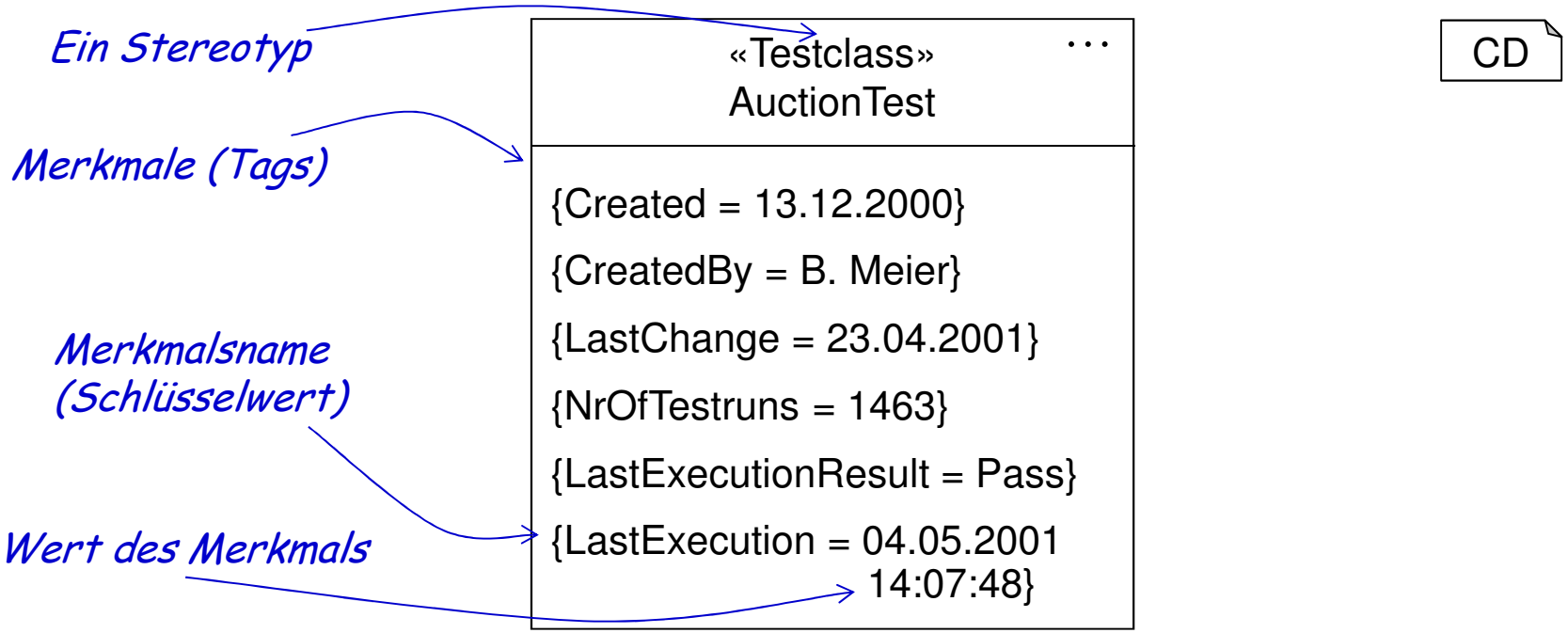
Stereotyp

- Stereotyp klassifiziert Modellelemente
  (beispielsweise Klassen oder Attribute)
- Stereotyp spezialisiert die Bedeutung des Modellelements
  - erlaubt spezielle Darstellung
  - effizientere oder zielspezifische Codegenerierung
  - etc.
- Stereotyp besteht aus <<Name>>
- Ein Stereotyp kann eine Menge von Merkmalen (tags) besitzen.



Merkmal (tag)

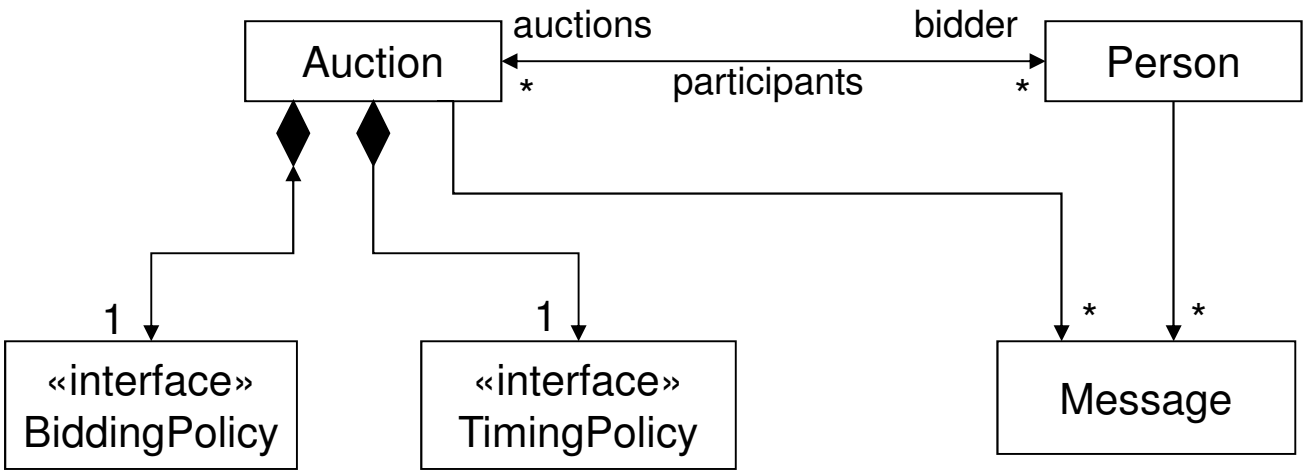
- Merkmal beschreibt eine Eigenschaft eines Modellelements
- Ein Merkmal wird notiert als Paar {tagname=value}
  - Schlüsselwort tagname und
  - Wert value
  - {Eigenschaft=true} ist abgekürzt durch {Eigenschaft}
- Beispiele: {ordered}, {persistent} oder hier aus einem Test-Modell:



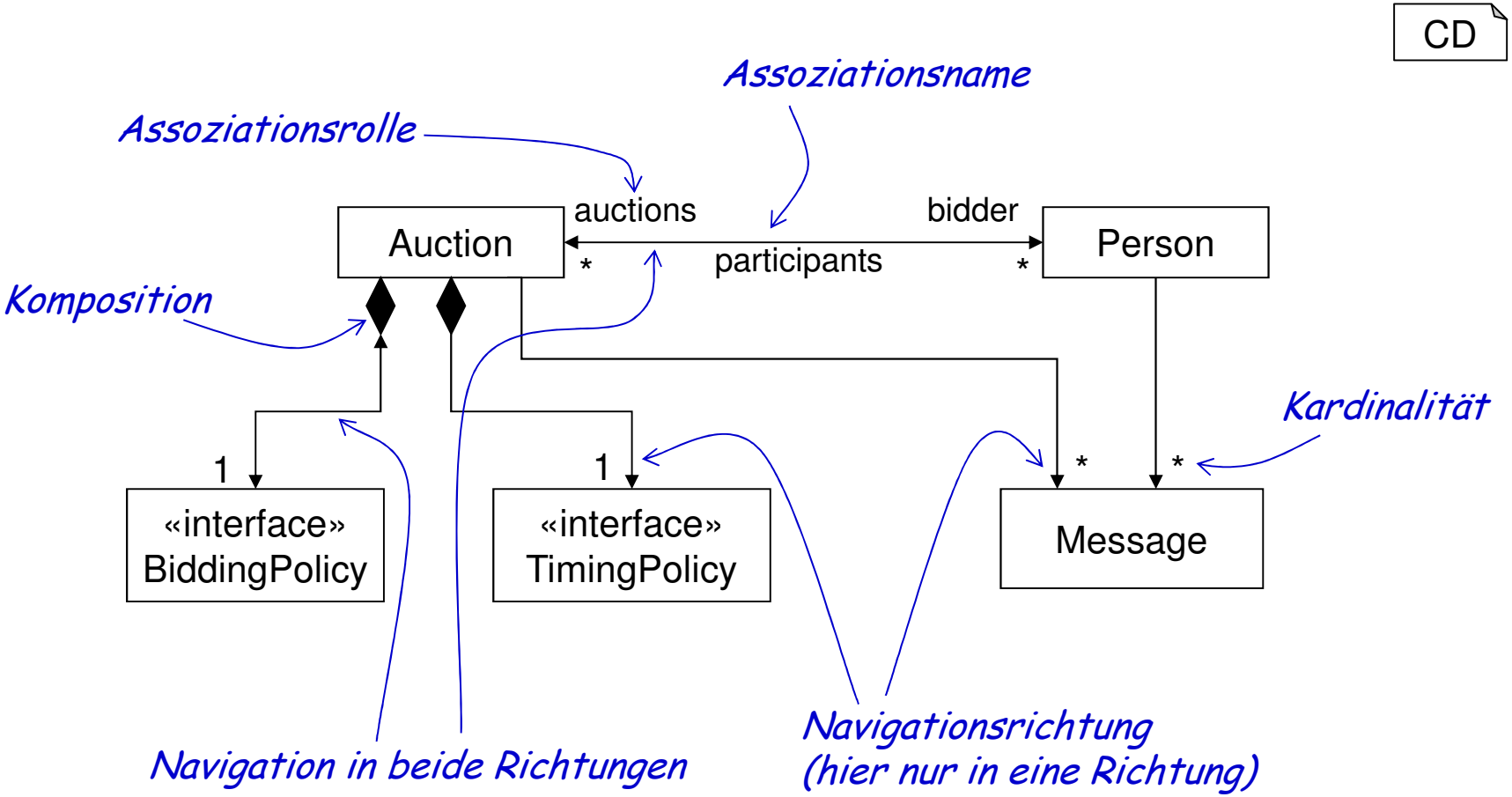
Assoziationen

Analyze

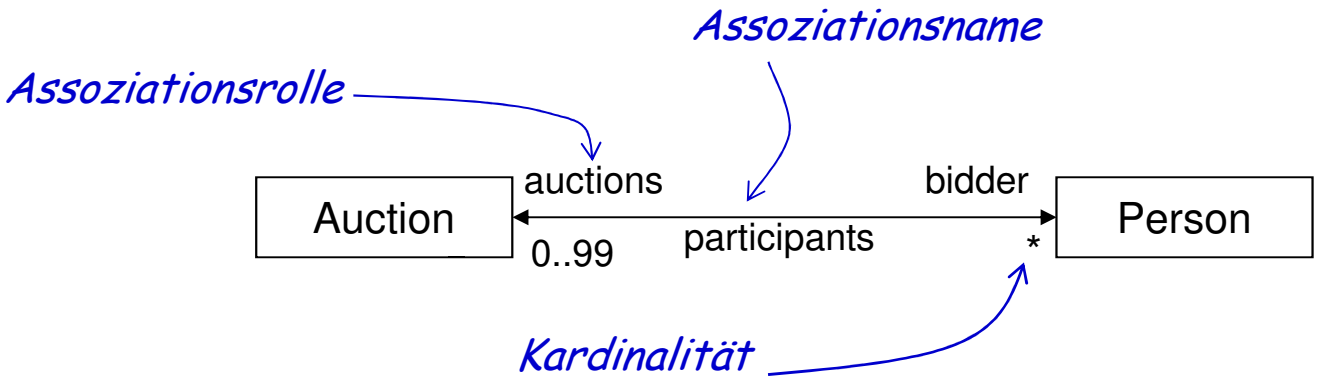
CD



Assoziationen

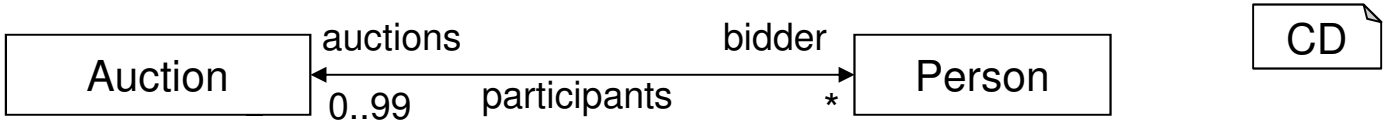


Assoziationen

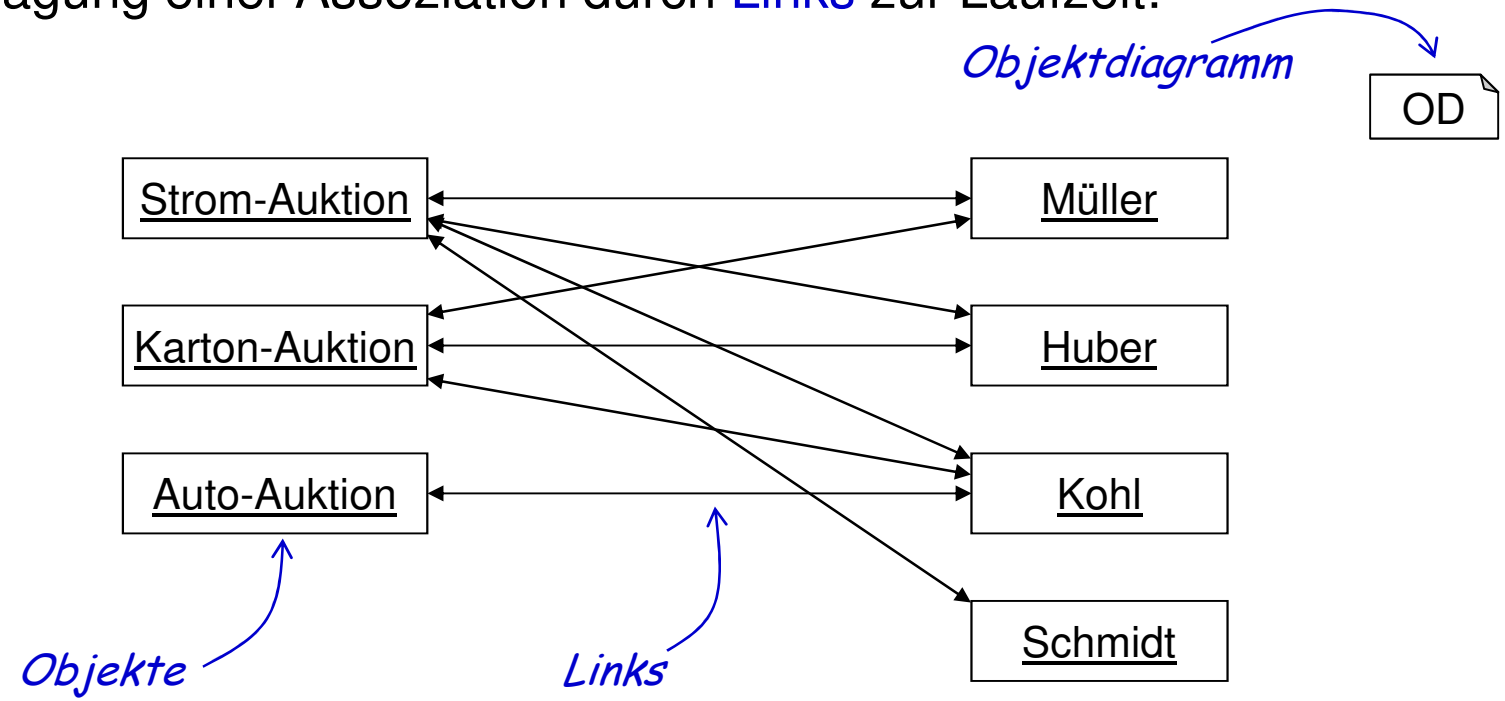


- Assoziation als binäre Beziehung zwischen Klassen
  - „Person participates in Auction“
  - Aus Sicht der Person: Navigation zu den Auktionen, deshalb
    - Assoziationsrolle „auctions“ links
    - Eine Person kann an bis zu 99 Auktionen teilnehmen (0..99)
- Kardinalitäten:
  - genau-eines: 1
  - optional: 0..1
  - beliebig: *
  - nicht-null: 1..* (oder +)
  - feste Intervalle: 3..9,17,21,42..99 (aber nur selten verwendet)

Assoziationen und Links



- Ausprägung einer Assoziation durch Links zur Laufzeit:



Rollennamen

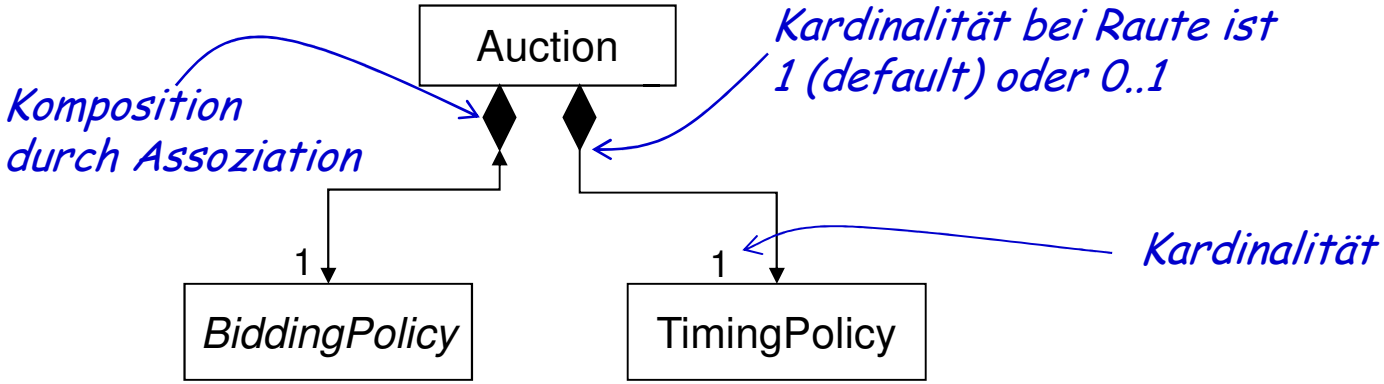
- Rollennamen dienen zur Navigation
  (logisch oder in der Implementierung)
- Was ist wenn der Rollenname fehlt?
  - Ersatzweise, wenn eindeutig:
  - a) Klassenname der gegenüberliegenden Klasse mit kleinen Initialen
  - b) Assoziationsname
- Beispiel:
  - geg: Auction a;
  - gleichwertig sind:
    - a.bidder,

⇨

CD

Komposition

- Komposition = spezielle Form der Assoziation



- Bedeutung:
  - Kompositum aus Teilen zusammengesetzt
  - Objekte bilden eine zusammengehörende Einheit
  - Teile vom Kompositum abhängig
    - Lebenszyklus kombiniert
    - Austauschbarkeit nicht gegeben
  - Allerdings: Interpretationsunterschiede bei Werkzeugen
    - Daher: Projektspezifisch ggf. präzisieren!

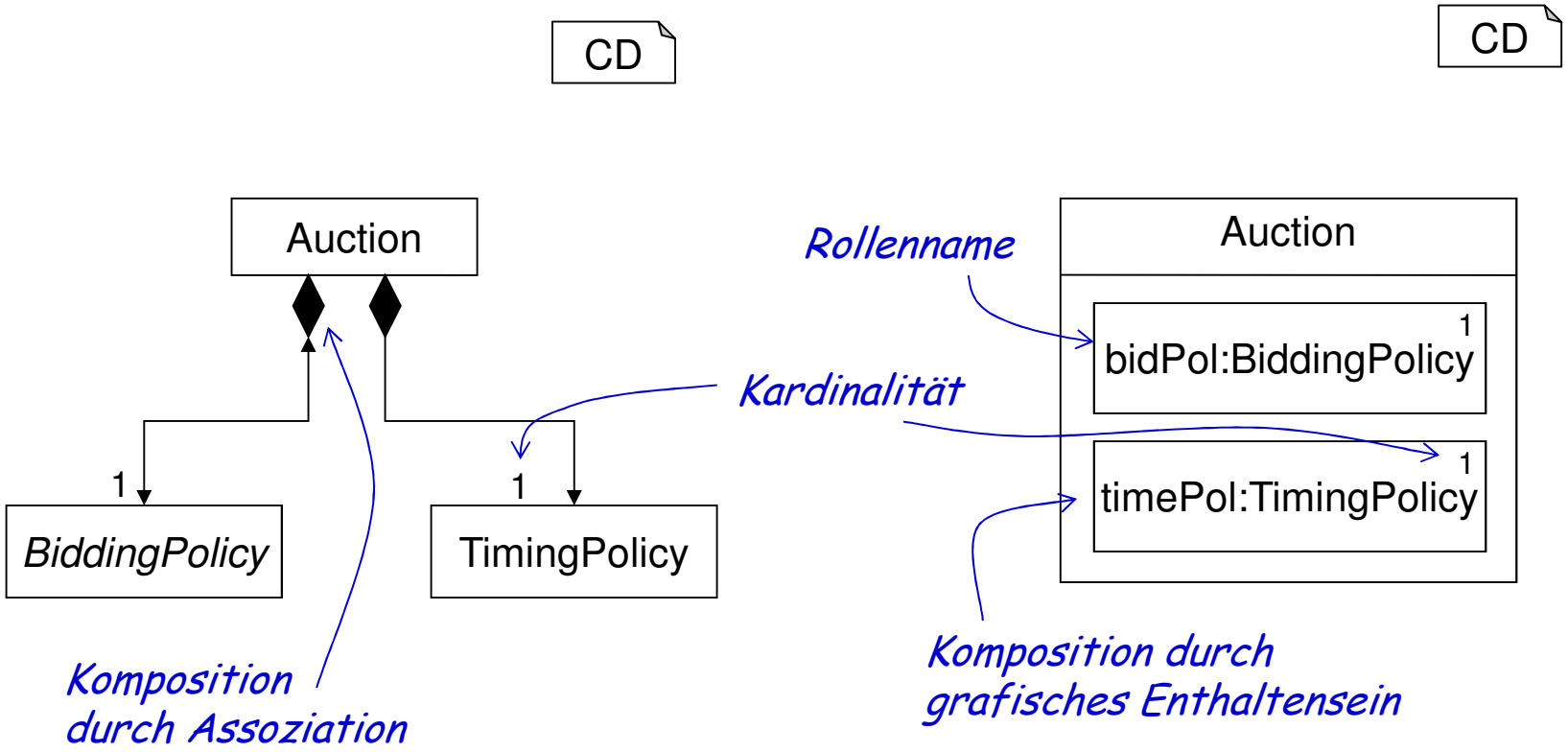
Einschränkungen bei Komposition

- Kontrolle der Teile durch Kompositum
  - Austausch der Teile höchstens mit Genehmigung des Kompositums (meist aber fixiert)
  - Lebenszyklus der Teile abhängig vom Kompositum
  - Oft auch: Zugang / Methodenaufrufe über das Kompositum
- Objekte können sich nicht selbst/gegenseitig enthalten
- Transitive Hülle des Enthaltenseins
  - aber: beachte verschiedene Formen
  - Beispiel: Hand, Rektor, RWTH?

- „Aggregation“ als schwache Form der Komposition mit weißer Raute: am besten nicht benutzen!

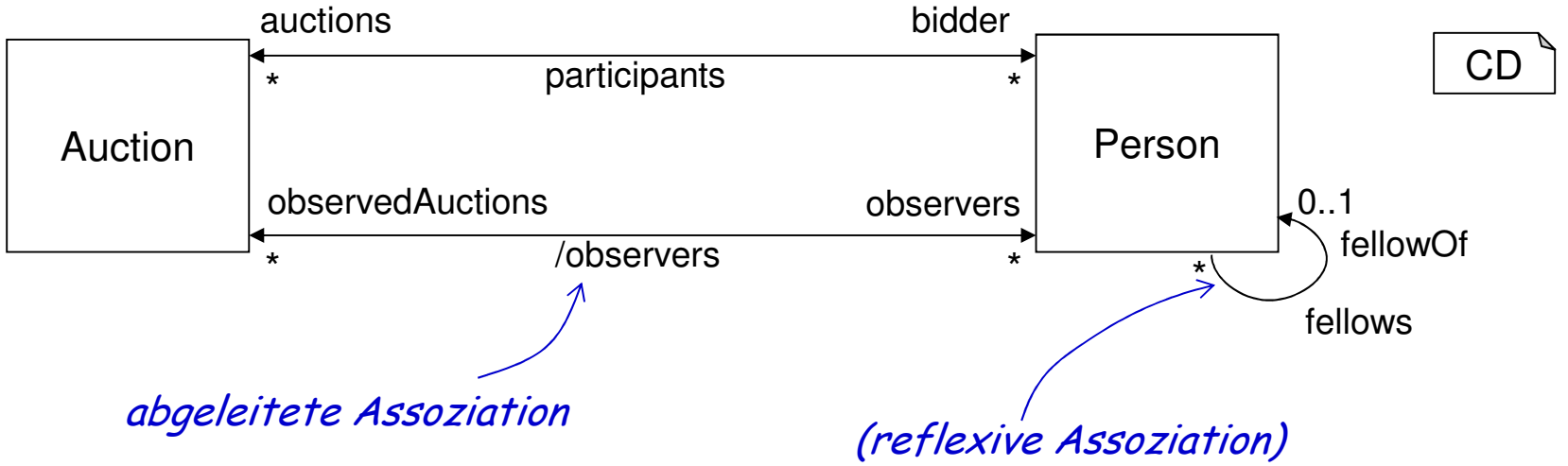
Darstellung von Komposition

- zwei (fast) identische Formen



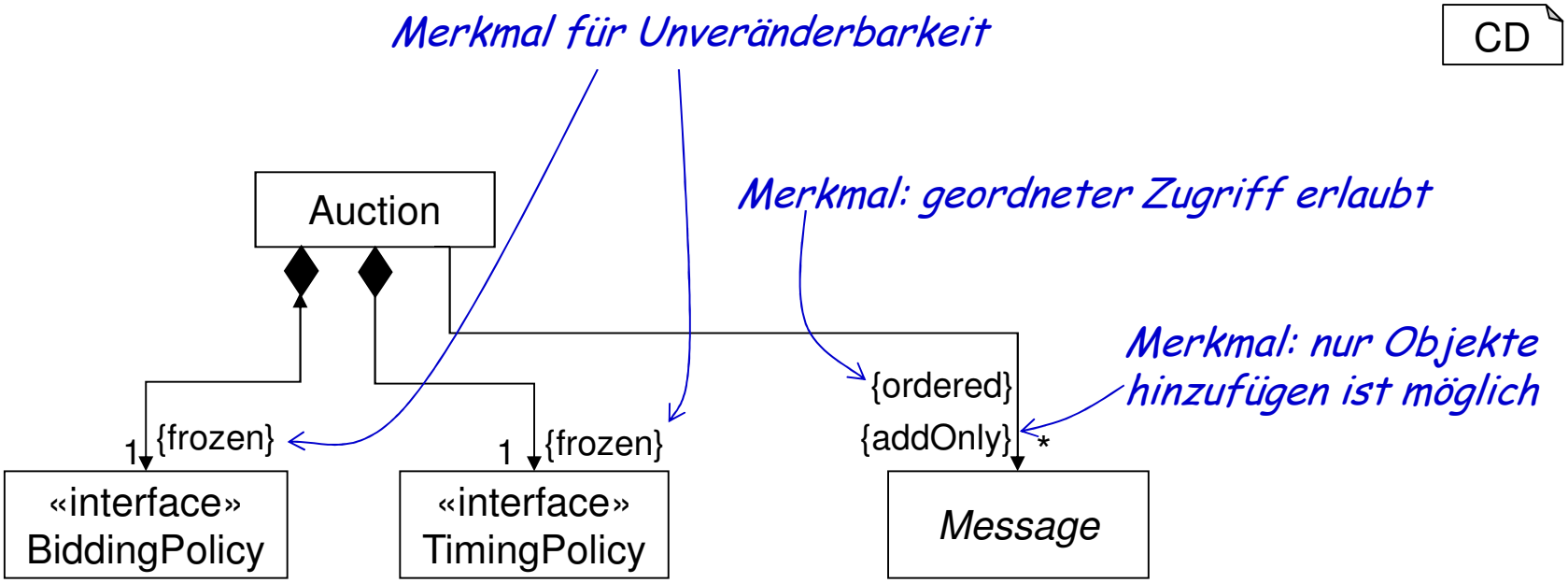
Abgeleitete Assoziation

- Abgeleitet heißt: Die Assoziation, kann durch eine andere berechnet werden.
- Beispiel: Eine Person darf eine Auktion beobachten, wenn sie Bieter oder Fellow eines Bieters ist.



- Eine Berechnungsvorschrift ist (in OCL fomuliert):
  - context Person p inv:
  - p.observedAuctions == p.fellowOf.auctions.union(p.auctions)

Merkmale für Assoziationen

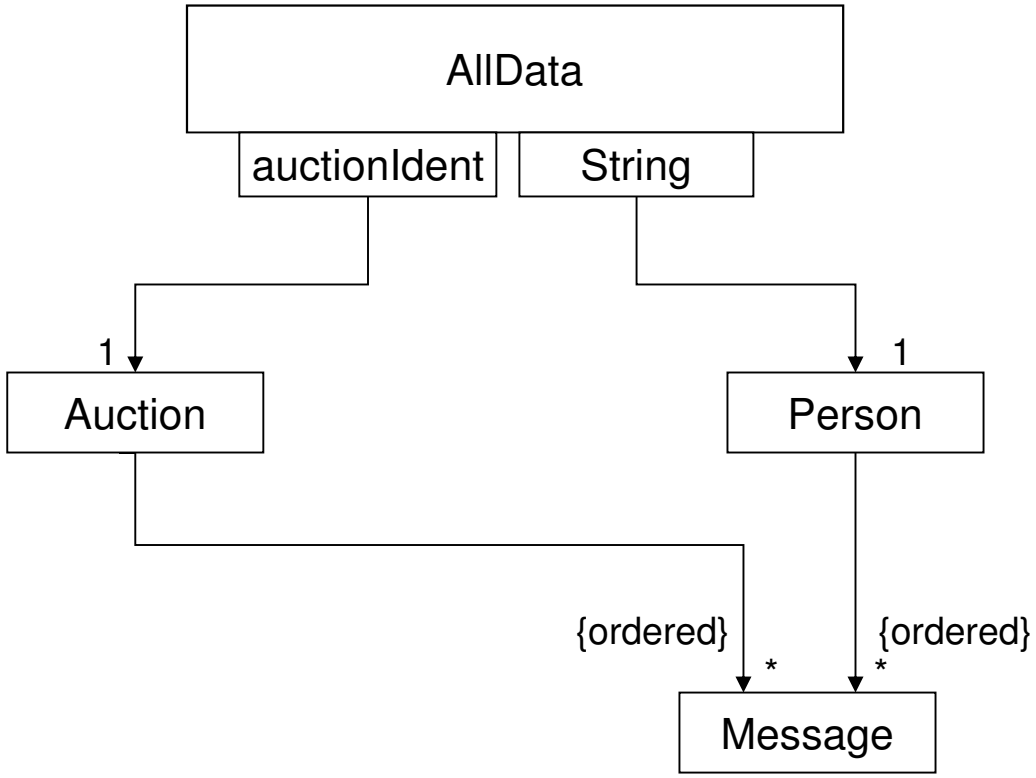


- Merkmale dienen der speziellen Realisierung:
  - {frozen}: Nach Initialisierung wird nicht geändert
  - {addOnly}: Nachrichten, die ausgegeben sind, bleiben erhalten
  - {ordered}: es gibt eine Reihenfolge

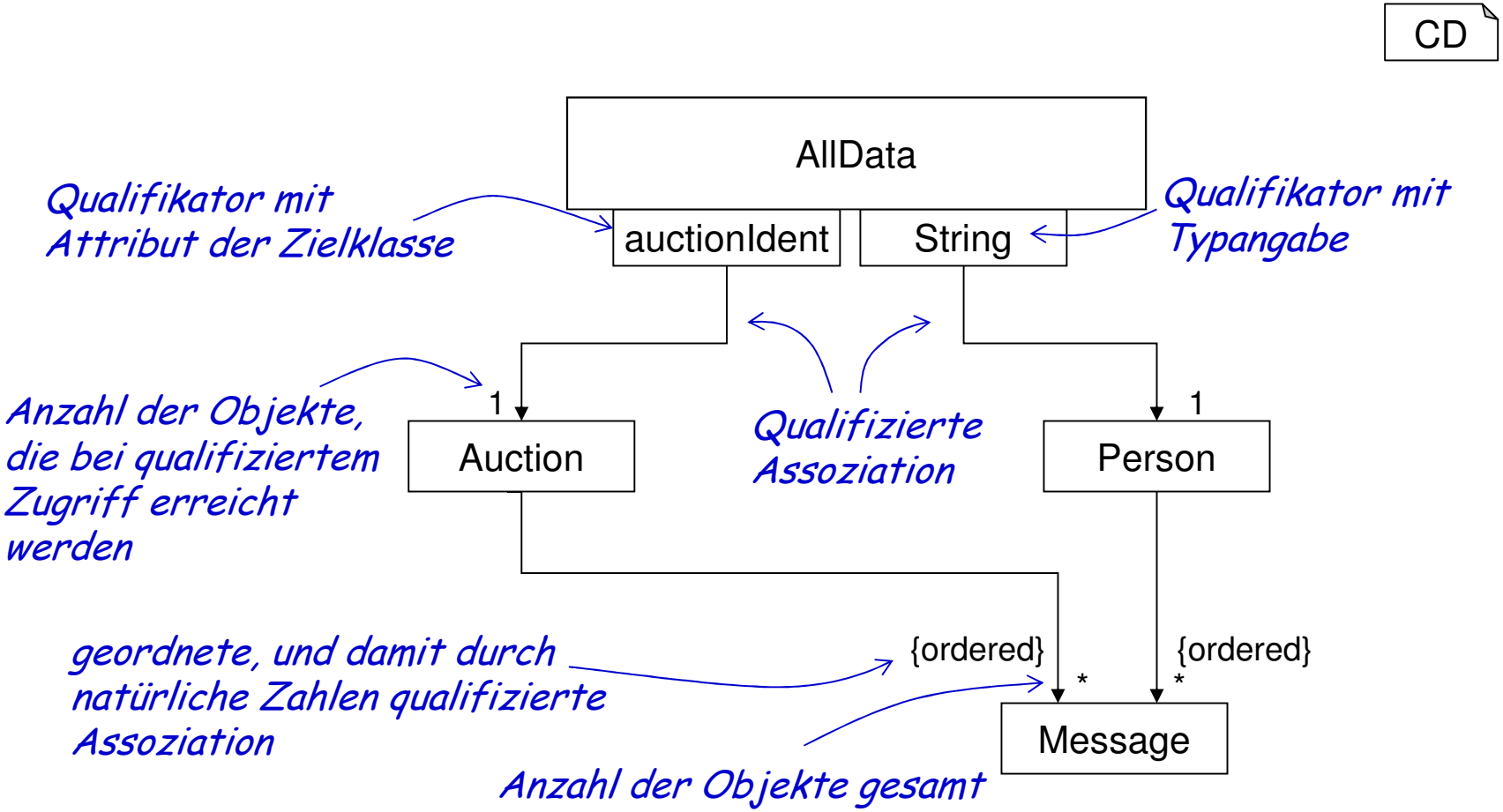
Qualifizierte Assoziation

Analyze

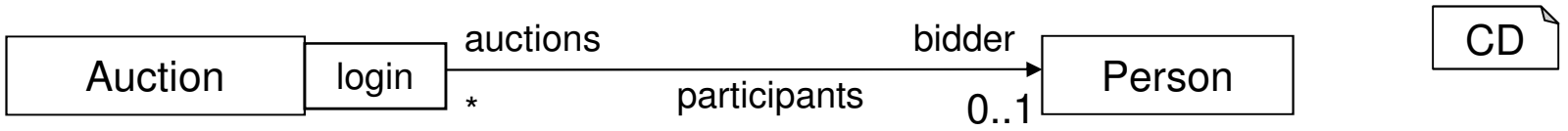
CD



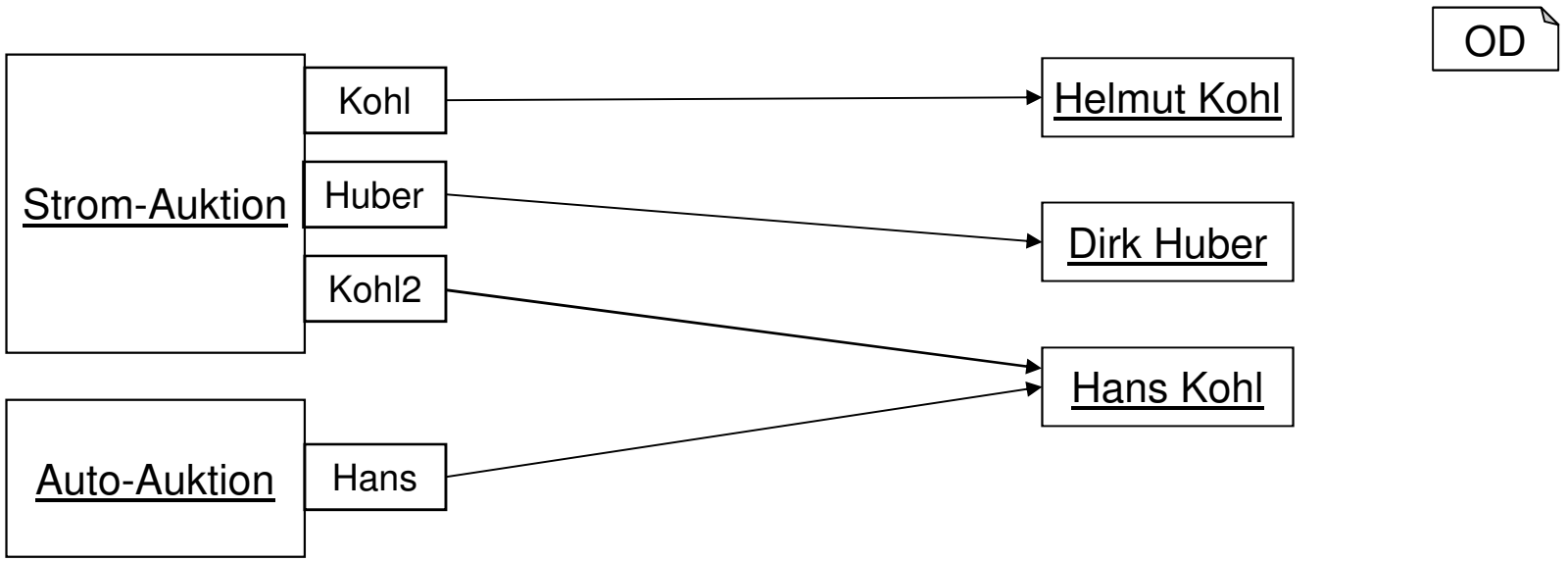
Qualifizierte Assoziation



Qualifizierte Assoziation und Links



- Qualifikator „login“ erlaubt einzelne Objekte zu selektieren
- Dasselbe Objekt kann in unterschiedlichen Auktionen durch verschiedenqualifiziert sein (unterschiedliche Logins)



Qualifizierte Assoziation

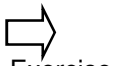
- Qualifizierte Assoziationen erlauben Selektion einzelner Objekte aus einer Menge mit Qualifikator
- Qualifikatoren können sein:
  - Zahlen-Intervall (0-..), das Reihenfolge anzeigt ({ordered})
  - Expliziter Identifikator (Attribut) des Zielobjekts (auctionIdent)
- Auch Komposition kann qualifiziert sein
- Qualifikation an beiden Enden möglich

- Qualifizierte Assoziation benötigt erweiterte Mechanismen zur Bearbeitung
  - Selektiver Zugriff, selektive Änderung

Zusammenfassung 2.3

- Klassendiagramme besitzen
  - Klassen mit Attributen und Methoden
  - Abstrakte Klassen, Interfaces
  - Vererbung, Interface-Erweiterung, - Implementierung
  - Assoziationen mit
    - Namen, Rollen, Kardinalitäten, Navigationsrichtungen
  - Varianten von Assoziationen:
    - Komposition
    - Qualifizierte Assoziation
    - Geordnete Assoziation
- Stereotypen und Merkmale (Tags) spezialisieren einzelne Modelelemente
- Nicht besprochene Erweiterungen:
  - Aggregation, Assoziationen mit >2 Partnern, Assoziationsattribute

Beispiel-Aufgabe - 1



- geg. Beschreibung, ges: Klassendiagramm:

  - Ein Fahrzeug besteht aus einer Karosserie, einem Bremssystem, zwei bis vier Sitzen und vier Rädern an den unterschiedlichen Positionen (vorneLinks, vorneRechts, hintenLinks, hintenRechts). Jedes Rad besteht aus einer Felge, einem Reifen, einer Scheibenbremse und fünf Schrauben.

Beispiel-Aufgabe - 2

- Es gilt außerdem:
  - Räder haben Durchmesser, Breite und Soll-Reifendruck.
  - Das Bremssystem ist mit allen Scheibenbremsen verbunden.
  - Fahrzeuge haben eine Typenbezeichnung, ein Datum der Erstzulassung und einen Besitzer.
  - Reifen haben eine Profiltiefe.

Ausblick in die Constraints mit OCL


Exercise

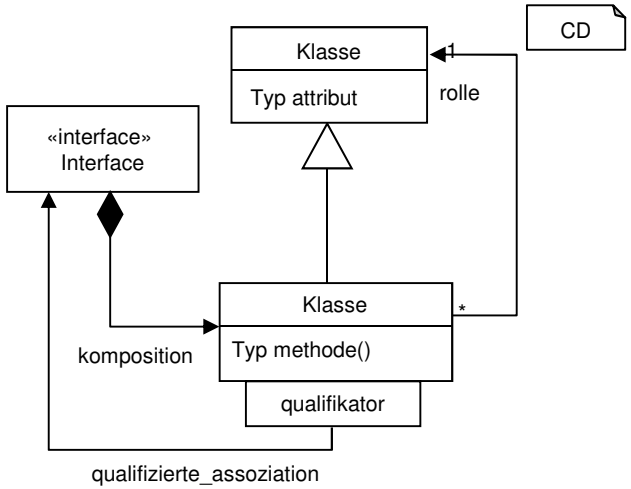
- Wie stellt man dar, dass
  - bei jedem Fahrzeug die beiden Reifen einer Seite jeweils denselben Soll-Reifendruck besitzen und
  - bei den Reifen vorne die Profiltiefe gleich ist.

SE SOFTWARE ENGINEERING

RWTH AACHEN

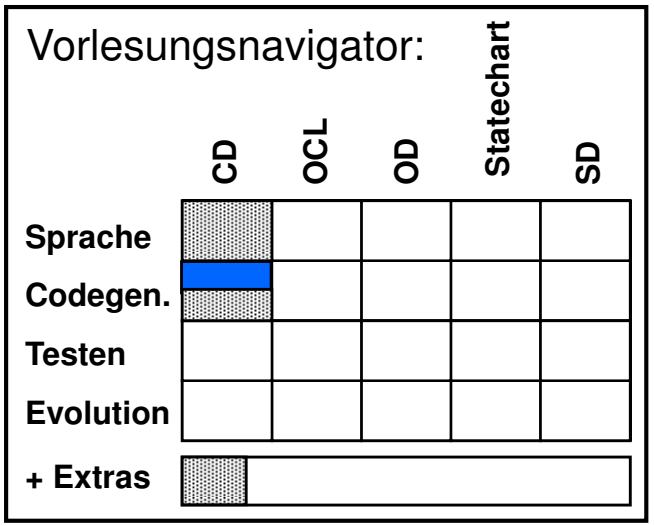
Modellbasierte Softwareentwicklung

- 2. Strukturmodellierung mit Klassendiagrammen
- 2.4. Codegenerierung Teil 2: Ausgesuchte Varianten der Vererbung und Assoziationen



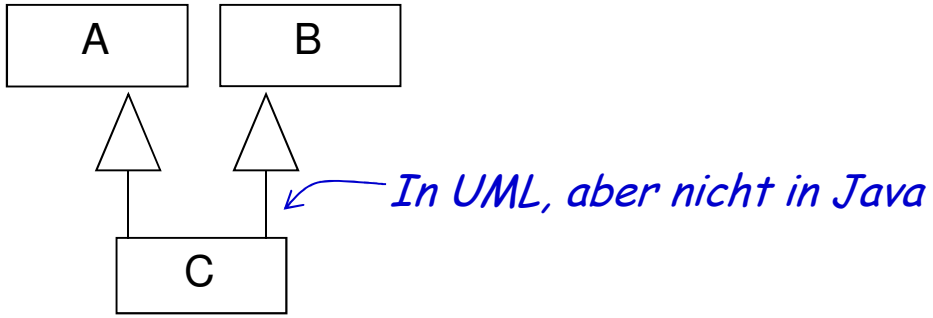
Prof. Dr. Bernhard Rumpe
Lehrstuhl für Software Engineering
RWTH Aachen

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



Vererbung

- Vererbung, Interface-Implementierung und Interface-Erweiterung werden direkt in Java abgebildet
- Problem: eine Klasse erbt von mehreren Vorgängern:



CD

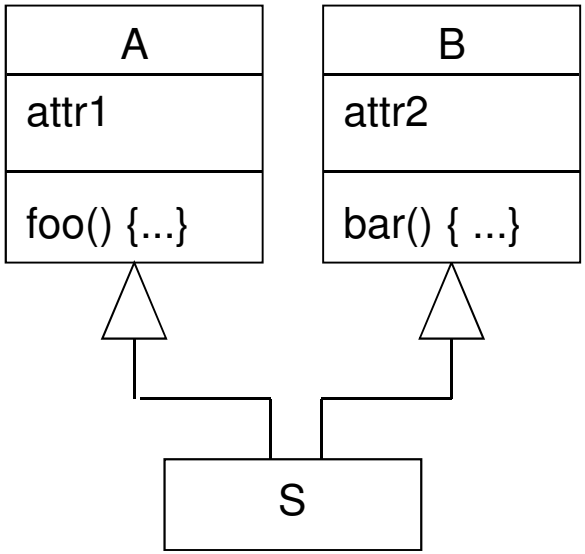
- Lösungen:
  - a. Eine Superklasse wird zum Interface umgebaut
  - b. Delegation statt Vererbung
  - c. Kombination aus beidem
- Auswahl der Lösung abhängig vom Kontext

Umbau von Mehrfachvererbung

Discuss

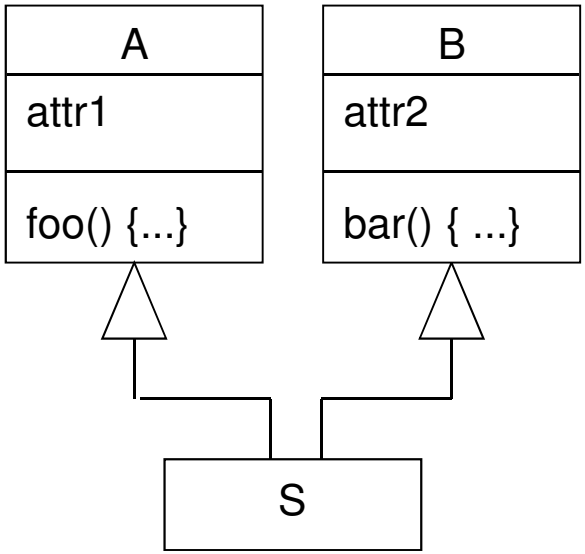
- Alt:

CD

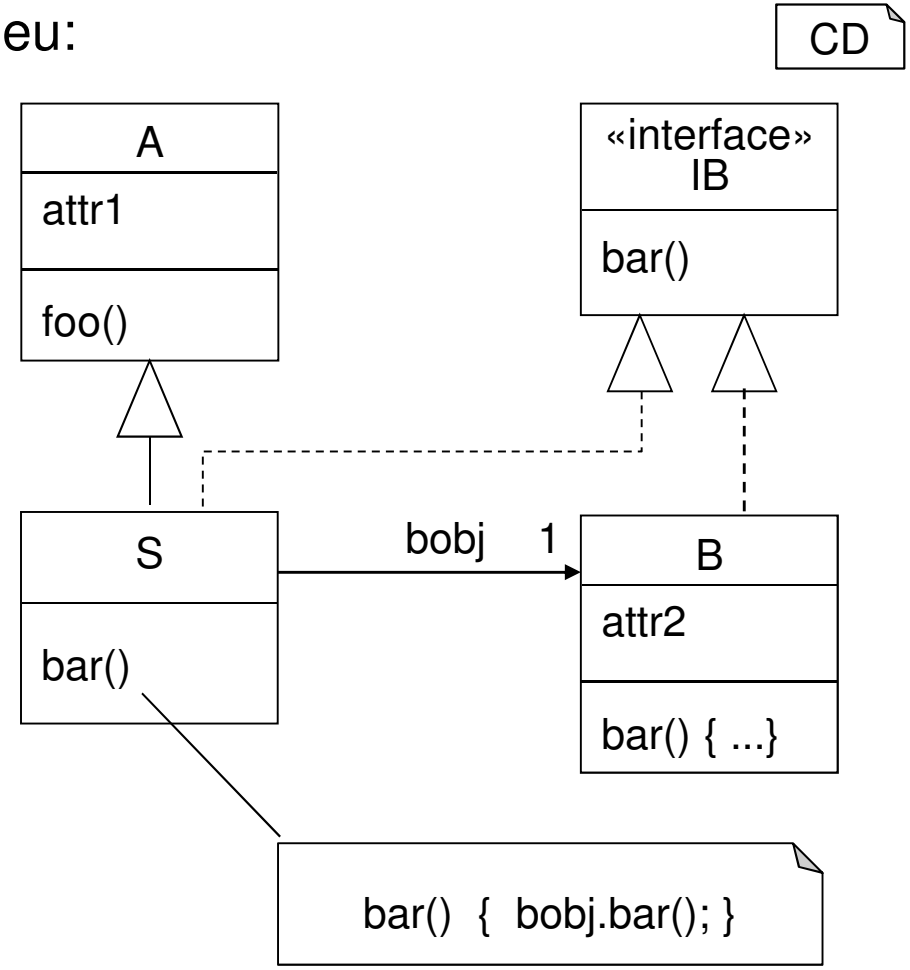


Umbau von Mehrfachvererbung

- Alt:



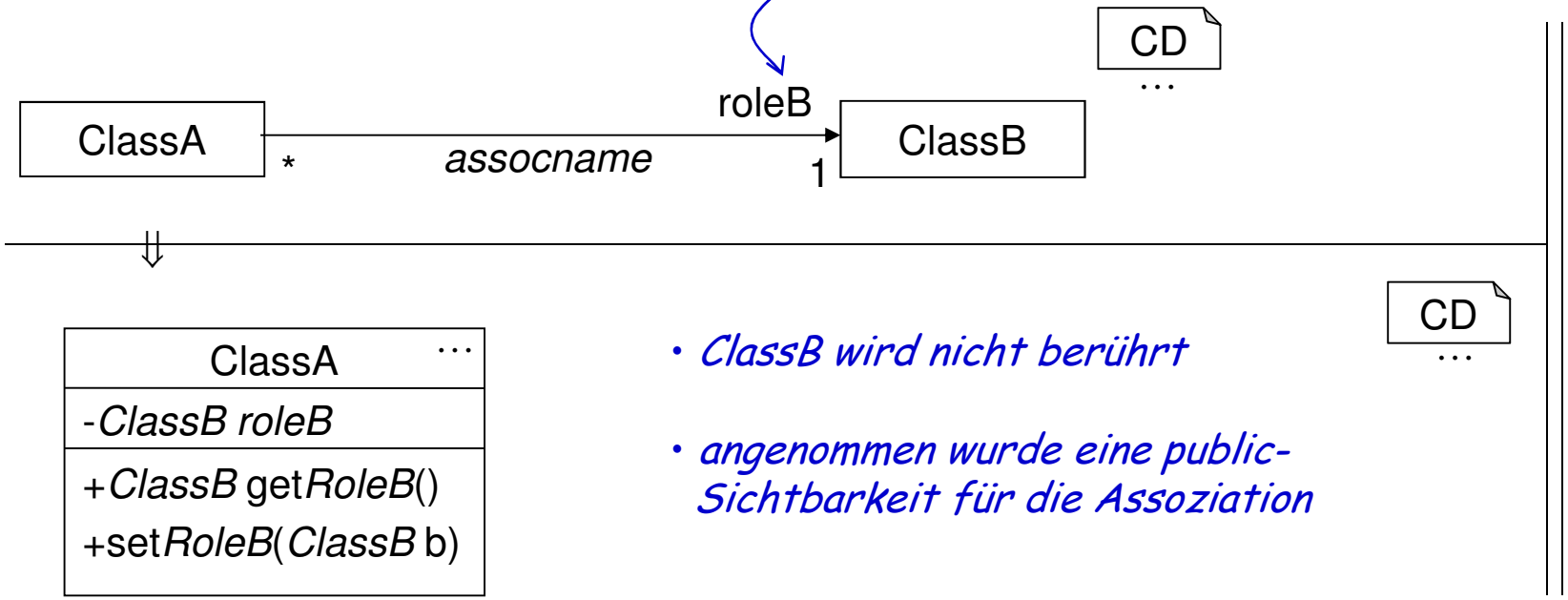
Neu:



- Probleme des Umbaus:
  - zwei Objekte enthalten den neuen Zustand verteilt: Komplexer

1-zu-*-Assoziation

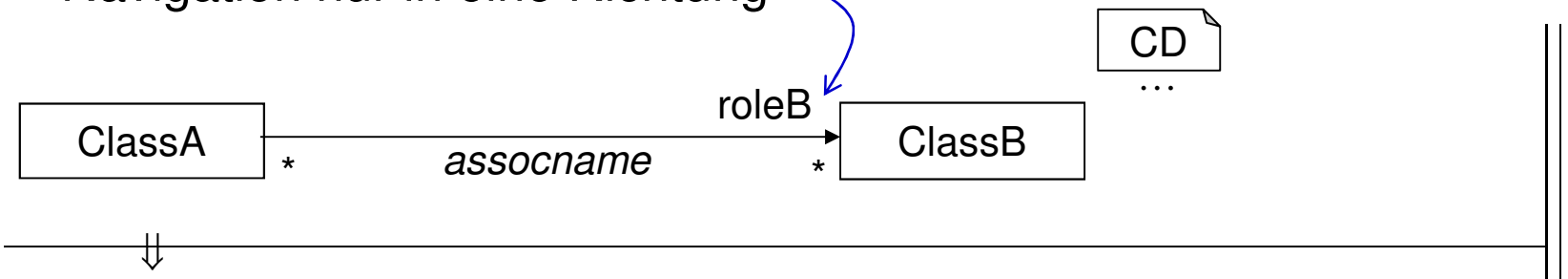
- Einfache Assoziation 1-zu-1 oder 1-zu-*
- Navigation nur in eine Richtung
- Codegenerierung beschrieben durch Transformation:

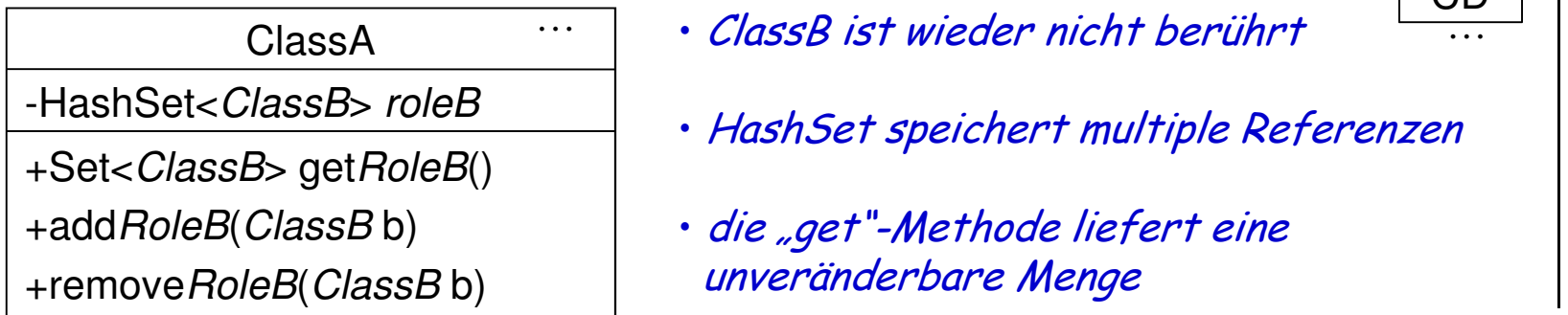


- *ClassB wird nicht berührt*
- *angenommen wurde eine public-Sichtbarkeit für die Assoziation*

-zu--Assoziation

- Navigation nur in eine Richtung

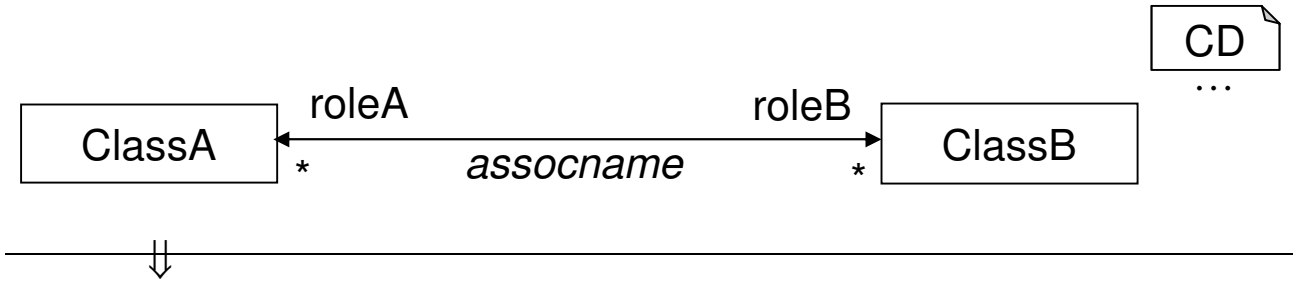


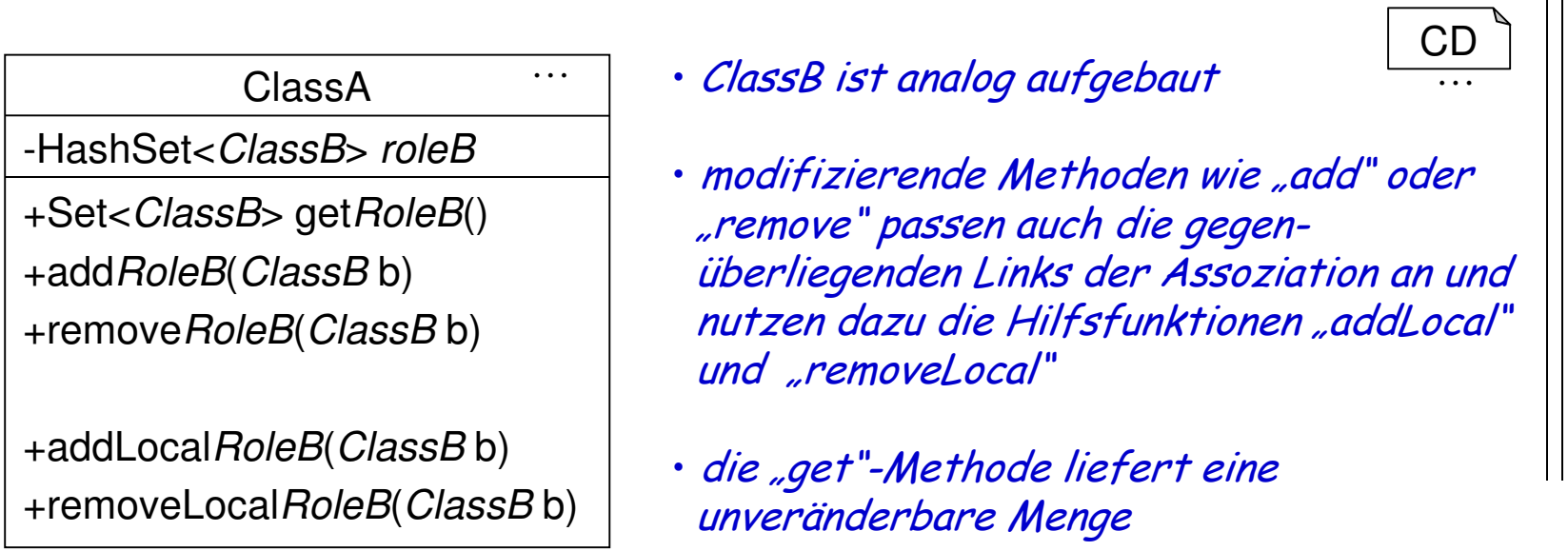


- *ClassB ist wieder nicht berührt*
- *HashSet speichert multiple Referenzen*
- *die „get"-Methode liefert eine unveränderbare Menge*
- *evtl. weitere Methoden wie Iteratoren*

-zu--Assoziation mit Navigation in beide Richtungen

- Umsetzung in der dezentralisierten Variante
- Im Prinzip Verwaltung der Assoziation auf beiden Seiten wie gehabt.
- Problem: Konsistenzhaltung erfordert zusätzliche Infrastruktur:

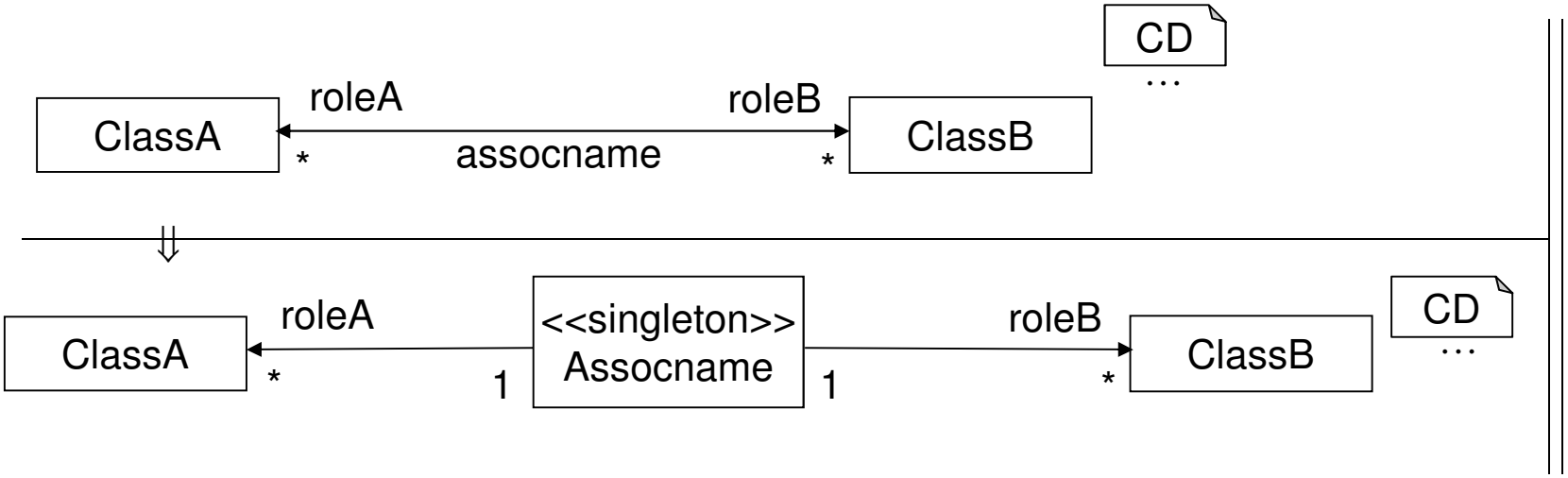




- *ClassB ist analog aufgebaut*
- *modifizierende Methoden wie „add" oder „remove" passen auch die gegen-überliegenden Links der Assoziation an und nutzen dazu die Hilfsfunktionen „addLocal" und „removeLocal"*
- *die „get"-Methode liefert eine unveränderbare Menge*

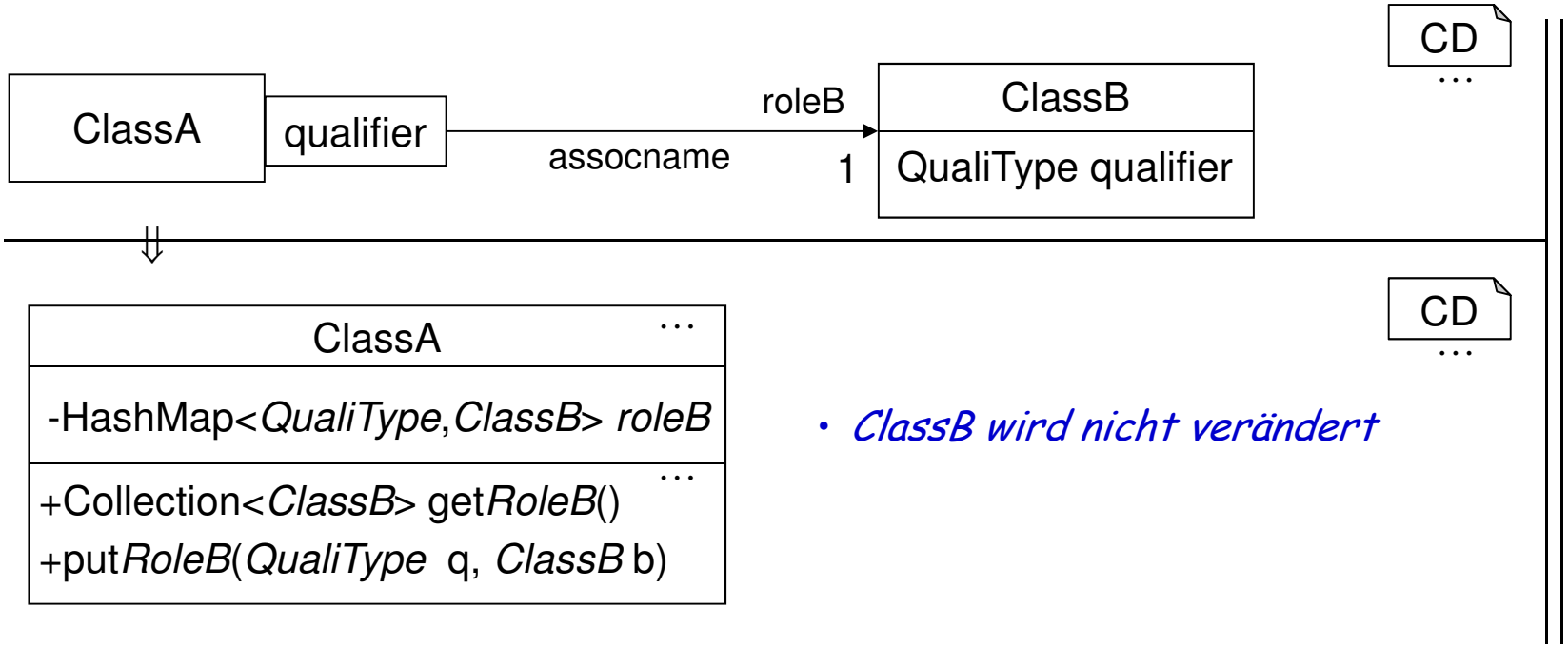
-zu--Assoziation mit Navigation in beide Richtungen

- Umsetzung in der zentralisierten Variante mit Singleton
- Einbau einer „Assoziationsklasse“, die zentral Links verwaltet
- Assocname verwendet intern eine in beide Richtungen navigierbare Relation
- Zugriff von ClassA bzw. ClassB erfolgt über zentrales Objekt
- aber: komplexere interne Verwaltungsstruktur



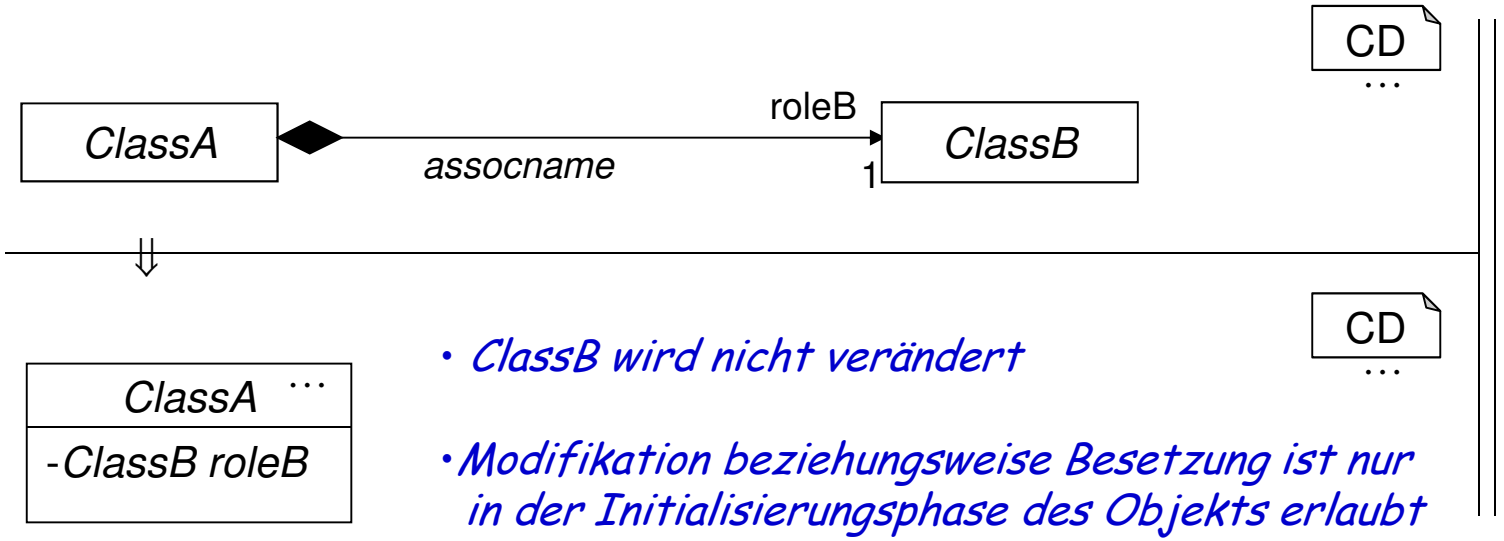
Qualifizierte Assoziation

- HashMap erlaubt die Realisierung eines Qualifikators
- Problem: Redundante Speicherung des Qualifikators in der HashMap und der Zielklasse
  - evtl. fordern, dass qualifier nicht verändert werden darf
- Zugriffsfunktionen und Modifikatoren können über die HashMap angeboten werden: Aber die HashMap nicht direkt herausgeben!



Komposition

- Komposition wie Assoziation behandeln
- Problem:
  - (Zeitliche) Abhängigkeit des Teilobjekts wird nicht realisiert
- Lösungsansätze:
  - Entwickler muss Komposition freiwillig „respektieren“
  - „Hilfestellung“ durch reduzierte Signatur



Zusammenfassung 2.4

- Dieser Abschnitt zeigte Codegenerierung aus Klassendiagrammen für verschiedene Konstellationen
  - Vielfalt syntaktischer Elemente: viele weitere Varianten
  - Manche Varianten sind in verschiedenen Kontexten optimal
  - Auswahl ist nicht trivial!
- Codegenerierung aus Klassendiagrammen ist automatisierbar
- aber: Gezeigte Umsetzungen können auch als Richtlinien für manuelle Umsetzung verstanden werden.

SE SOFTWARE ENGINEERING

RWTH AACHEN

Modellbasierte Softwareentwicklung

- 3. Object Constraint Language
- 3.1. Einführung

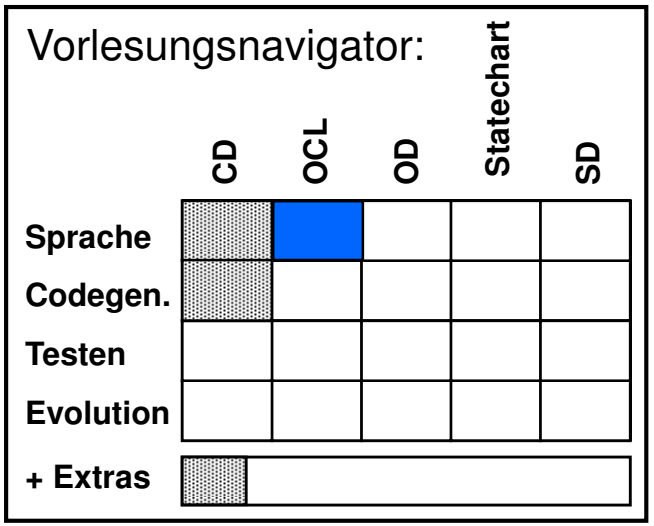
OCL

context Klasse inv:
    beschränkende Invariante

context Methode
    pre: Vorbedingung
    post: Nachbedingung

Prof. Dr. Bernhard Rumpe
Lehrstuhl für Software Engineering
RWTH Aachen

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



OCL – Einführendes Beispiel

- gegeben ist eine Klasse:

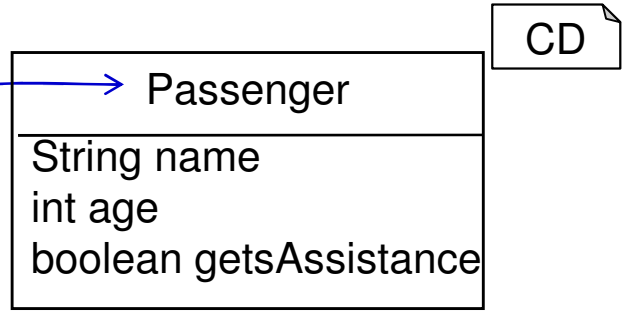
CD

Passenger
String name int age boolean getsAssistance

- Es soll nun zusätzlich gelten:
  - Passagiere sind mindestens ein Jahr alt
  - Sind Passagiere über 90 erhalten sie automatisch Unterstützung

OCL – Einführendes Beispiel

- gegeben ist eine Klasse:



- Es soll nun zusätzlich gelten:
  - Passagiere sind mindestens ein Jahr alt

context ist die Klasse

OCL

    - context Passenger inv:
      age >= 1

Invariante spricht über die Attribute der Objekte

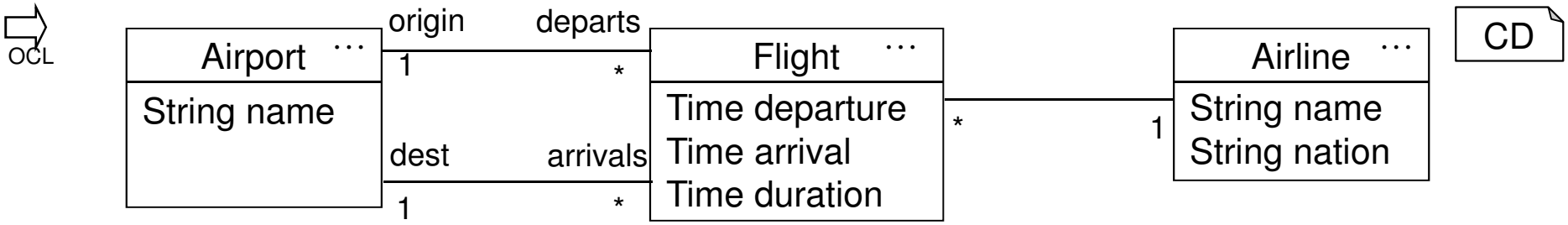
  - Sind Passagiere über 90 erhalten sie automatisch Unterstützug

Logik erlaubt komplexe Aussagen

OCL

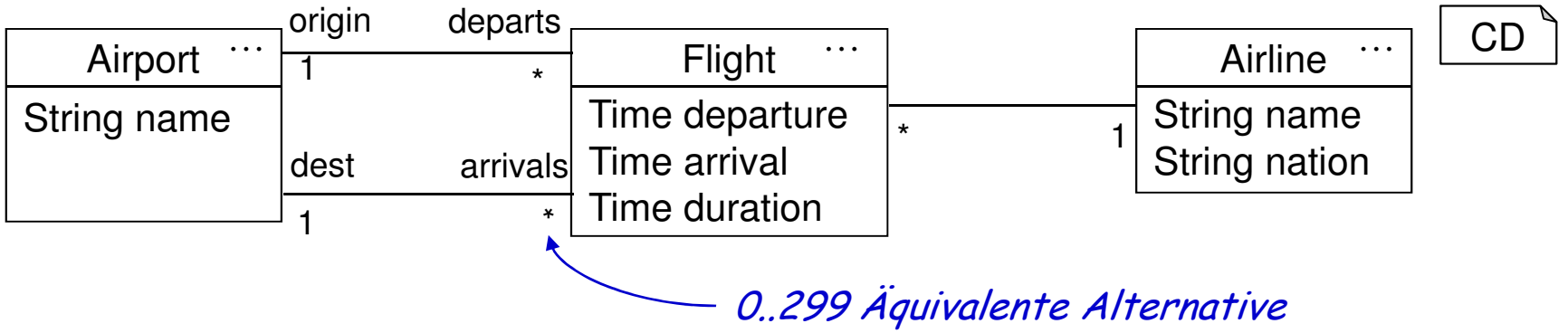
    - context Passenger inv:
      age >= 90 implies getsAssistance==true

OCL – Beispiel 2 (#Landungen)



- Weniger als 300 Landungen auf einem Flughafen:

OCL – Beispiel 2 (#Landungen)



- Weniger als 300 Landungen auf einem Flughafen:

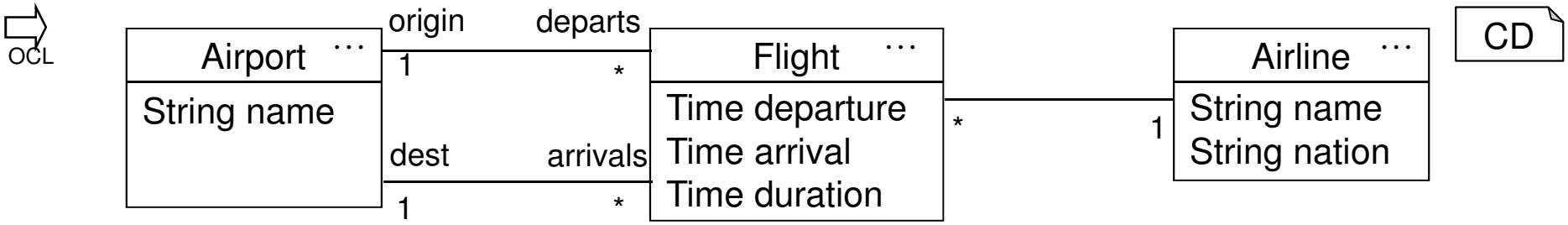
OCL

Explizite Benennung des Objekts:
Für alle ap vom Typ Airport gilt:

- context Airport ap inv:
  ap.arrivals.size < 300

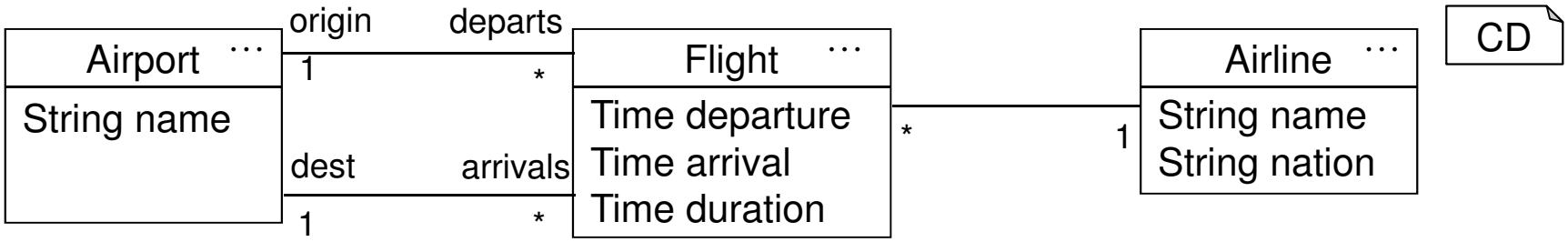
Navigation entlang einer Assoziation:
Liefert Menge von Objekten (Set(Flight))

OCL – Beispiel 3 (Schiphol)



- Alle Flüge der KLM starten in Amsterdam (Schiphol):

OCL – Beispiel 3 (Schiphol)



- Alle Flüge der KLM starten in Amsterdam (Schiphol):

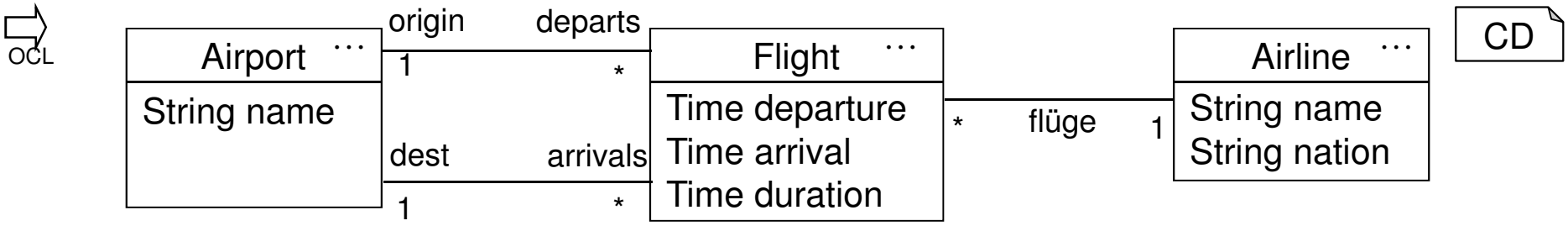
- context Airline al inv:
  al.name == “KLM” implies
  al.flight.origin.name == { “Schiphol” }

OCL

Menge (1 Element)

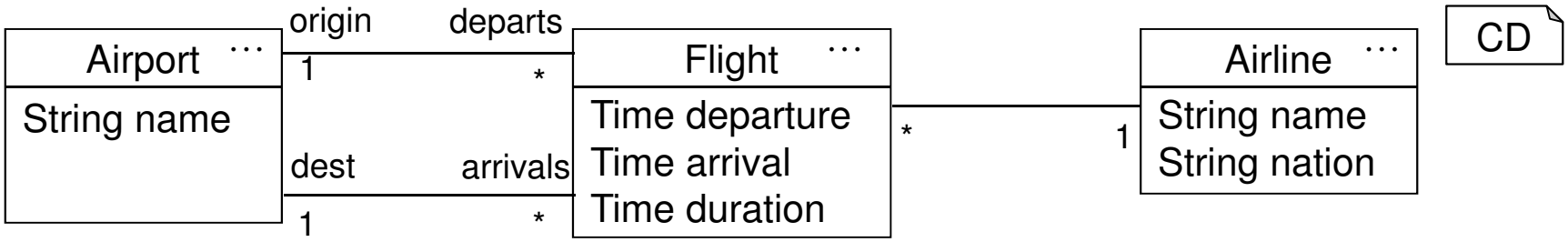
Navigationskette entlang Assoziationen:
Liefert Menge von Namen (Set(Flight))

OCL – Beispiel 4 (Start + Landung)



- Alle Flüge der KLM starten oder landen in Amsterdam (Schiphol):

OCL – Beispiel 4 (Start + Landung)



- Alle Flüge der KLM starten oder landen in Amsterdam (Schiphol):

OCL

- context Airline al inv:

```
    al.name == “KLM” implies
      forall fl in al.flight:
            fl.origin.name == “Schiphol” ||
            fl.dest.name   == “Schiphol”
```

Quantifier über Menge von Flügen

Object Constraint Language (OCL)

- OCL ist eine textuelle Spezifikationssprache
  - für Eigenschaften, die UML-Diagramme nicht abdecken
  - Invarianten, Vor-/Nachbedingungen, Wächter
- OCL einer First-Order Logik ähnlich, aber ausführbar.
  - Boolesche Operatoren, Quantoren
- Grundatentypen:
  - Boolean, Integer, Real, Char
  - Mengen und Sequenzen
- OCL wird im Kontext von UML-Diagrammen genutzt,
  - dort können Typen und Funktionen für OCL definiert werden
- In dieser Vorlesung:
  - Spezielle Fassung der OCL, die Java-Syntax nutzt

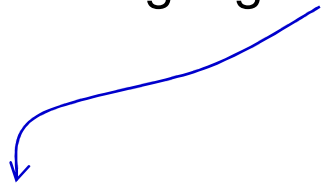
Begriffe zur OCL 1:

- Bedingung:
  - Eine Bedingung ist eine boolesche Aussage über ein System. Sie beschreibt eine Eigenschaft, die ein System oder ein Ergebnis besitzen soll.
  - Ihre Interpretation ergibt grundsätzlich einen der zwei Wahrheitswerte.

- Konsequenzen:
  - Eine Bedingungsauswertung kann nicht „abstürzen“.
    - Beispiel: 1/0 == 7 hat den Wahrheitswert false
  - Eine Bedingungsauswertung ist frei von Seiteneffekten
    - Einziges Ergebnis ist der berechnete Wert
  - Invariante nur bedingt „berechenbar “! Siehe dazu später!

Begriffe zur OCL 2:

- Kontext einer Bedingung:
  - Eine Bedingung ist in einen Kontext eingebettet, über den sie Aussagen macht.
  - Kontext ist definiert durch eine Menge von in der Bedingung verwendbaren Namen und ihren Signaturen. Dazu gehören Klassen-, Methoden- und Attributnamen des Modells sowie im Kontext einer Bedingung explizit eingeführte Variablen.



  - context Airport ap inv:
    ap.arrivals.size < 300

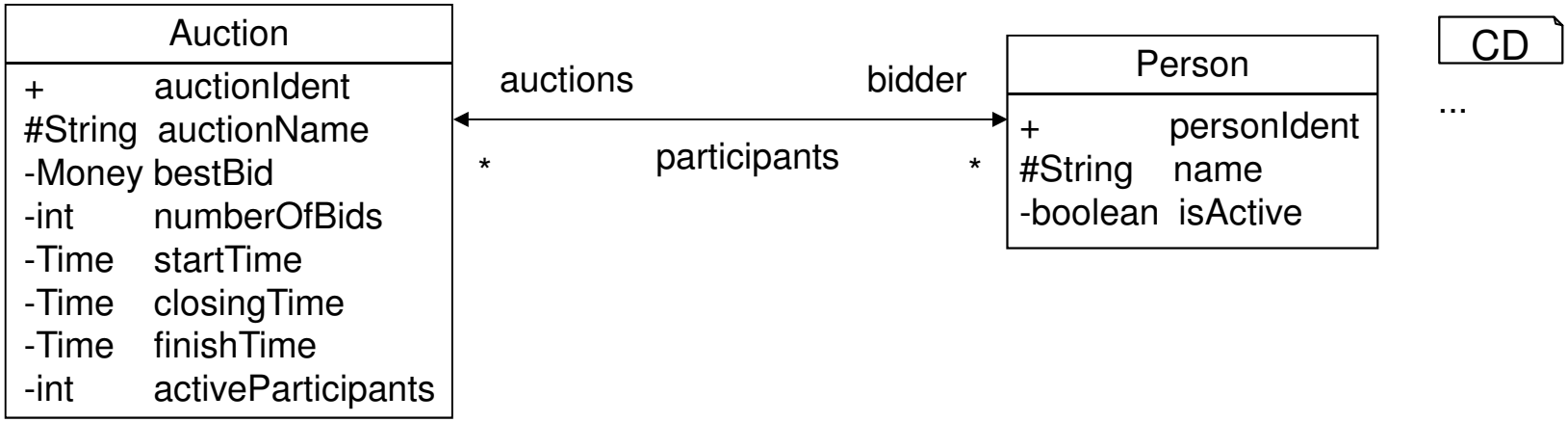
- Interpretation einer Bedingung an einer konkreten Objektstruktur. Die im Kontext eingeführten Variablen werden entsprechend mit Werten/Objekten belegt.

Begriffe zur OCL 3:

- Invariante:
  - beschreibt eine Eigenschaft, die in zu jedem (beobachteten) Zeitpunkt gilt.
  - Beobachtungszeitpunkte können eingeschränkt sein
  - Zeitlich begrenzte Verletzungen zum Beispiel während der Ausführung einer Methode zugelassen.

- Konsequenz:
  - Invarianten gelten vor allem dann, wenn die Objekte sich „in Ruhe“ befinden, also gerade keine Methoden darauf operieren.

Kontext „context“



- Kontext benennt neue Variable a:
  - context Auction a inv:

    a.startTime.lessThan(a.closingTime)

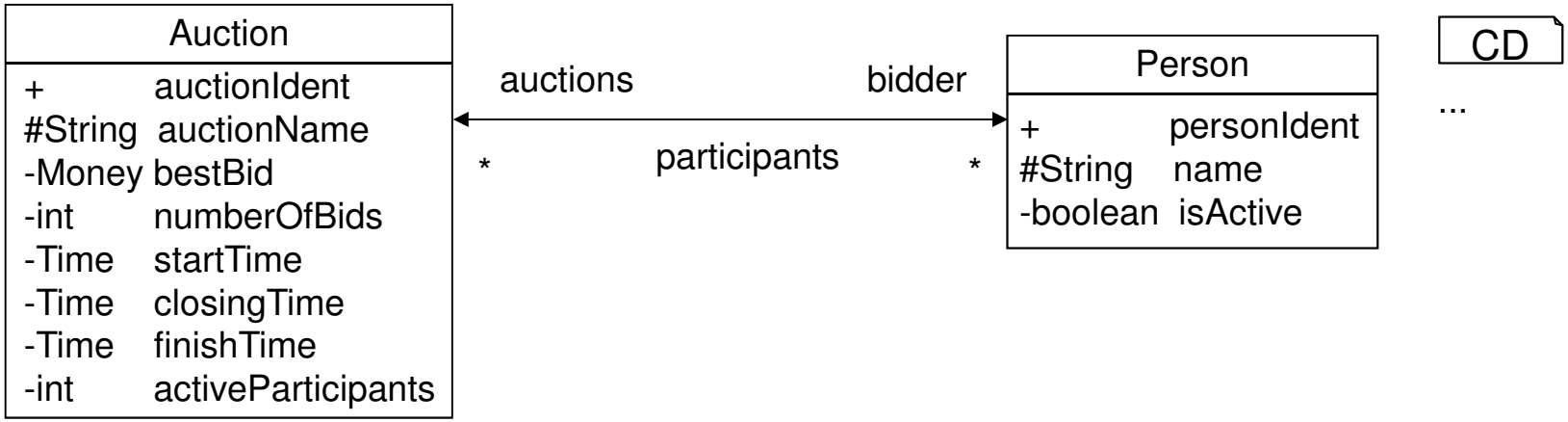
OCL

- Namen für die Bedingungen:
  - context Auction a inv Bidders1:

    a.activeParticipants <= a.bidder.size

OCL

Kontext „context“ ohne expliziten Namen



- Kontext benennt implizit neue Variable this:
  - context Auction inv:
    this.startTime.lessThan(this.closingTime)

OCL

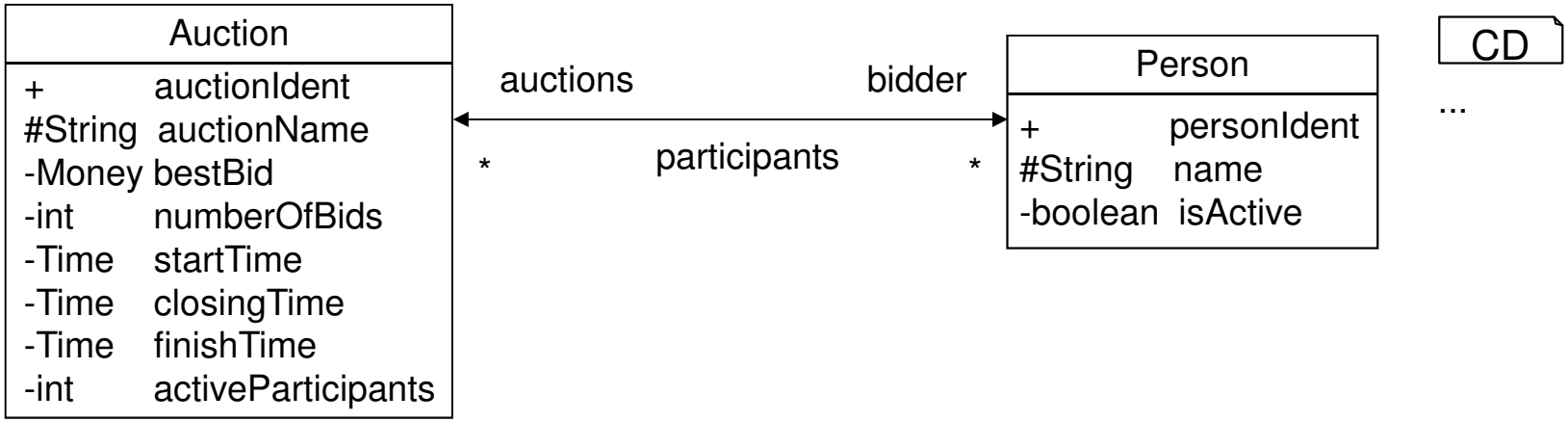
- äquivalent zu:
  - context Auction a inv:
    a.startTime.lessThan(a.closingTime)

OCL

- Kurzform verzichtet auf this (wie in Java):
  - context Auction inv:
    startTime.lessThan(closingTime)

OCL

Mengenkomprehension



- Mengen ähnlich zur Mathematik (wie in Gofer/Haskell):
- (Anzahl der aktiven Teilnehmer stimmt)
  - context Auction a inv:

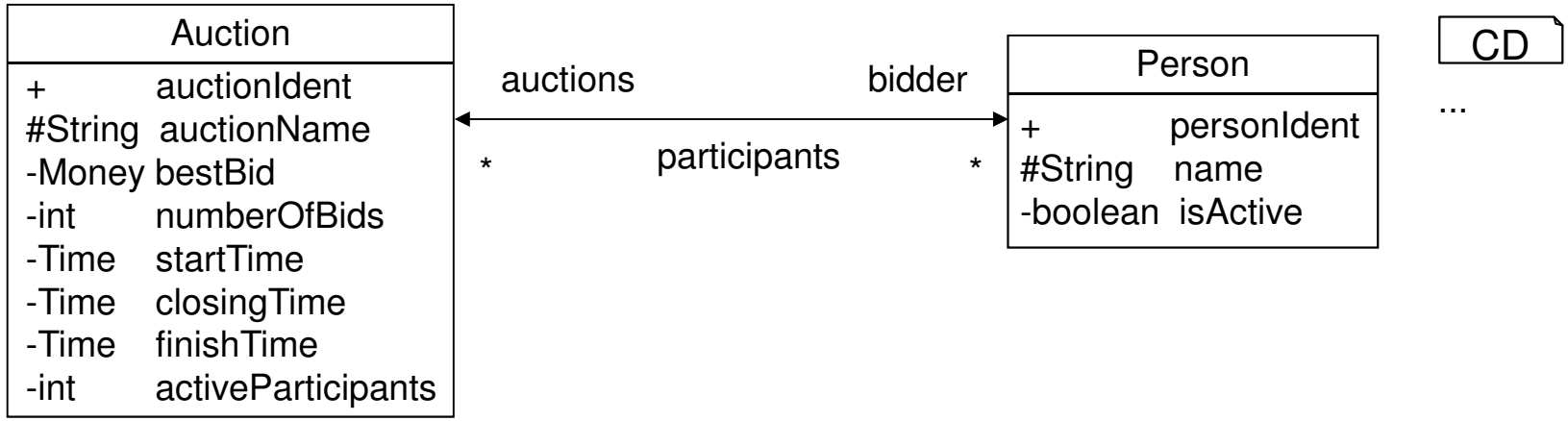
    a.activeParticipants == { p in a.bidder | p.isActive }.size

OCL

p ist aus der Menge von Bietern
p wird hier eingeführt mit dem Scope der Mengenkomprehension

Eigenschaft über p: selektiert eine Teilmenge

Lokale Variablen in OCL: let-Konstrukt



- Zwischenvereinbarungen:
- context Auction inv:

OCL

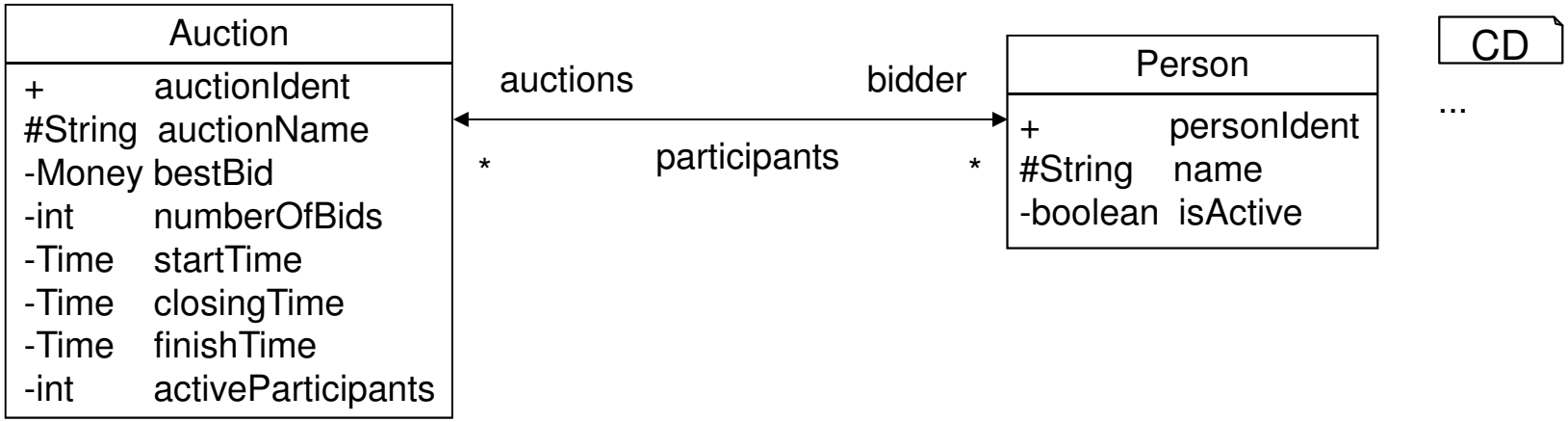
```
let min = startTime.lessThan(closingTime)
                    ? startTime : closingTime
in
  min == startTime
```

? : -- If-Then-Else

min wird hier als Variable eingeführt

und kann im Rumpf verwendet werden

Lokale Methoden in OCL: let-Konstrukt 2



- Zwischenvereinbarungen:
- context Auction a inv:

OCL

```
  let min(Time x, Time y) = x.lessThan(y) ? x : y
  in
    min(a.startTime, min(a.closingTime,a.finishTime)) == a.startTime
```

min wird hier als Operation mit Argumenten definiert

Fallunterscheidungen

- Fallunterscheidungen ergeben grundsätzlich Werte
  (OCL hat keine Anweisungen)
- Varianten:
  - **if** *Bedingung* **then** *Ausdruck1* **else** *Ausdruck2*
  - *Bedingung* **?** *Ausdruck1* **:** *Ausdruck2*
  - **typeif** *Variable* **instanceof** *Typ* **then** *Ausdruck1* **else** *Ausdruck2*

- Typeif ist eine typsichere Variante des Typecast für Variablen:

  - context Supertyp m inv:
    typeif m instanceof Subtyp then *(m hier als Subtyp bekannt)*
    else *(m hier nur als Supertyp)*

OCL

Grunddatentypen

- wie aus Java bekannt:
  - boolean, char, int, long, float, byte, short, double
- Entsprechende Operationen werden angeboten (+, ...)
  - Ausgeschlossen sind --, ++ etc. wegen Seiteneffekten
- String ist kein Grunddatentyp, sondern eine Klasse.
- Zusätzlich nutzen wir Datenstrukturen für Mengen und Listen:
  - Set<int>, List<String>, ...

Anhang: Liste aller OCL-Operatoren, Teil 1

Priorität	Operator	Assoziativität	Operanden, Bedeutung
14	@pre	links	Wert des Ausdrucks in Vorbedingung
	**	links	Transitive Hülle einer Assoziation
13	+, -, ~	rechts	Zahlen
	!	rechts	Boolean: Negation
	(type)	rechts	Typkonversion (Cast)
12	*, /, %	links	Zahlen
11	+, -	links	Zahlen, String (+)
10	<<, >>, >>>	links	Shifts
9	<, <=, >, >=	links	Vergleiche
	instanceof	links	Typvergleich
	in	links	Element von

Anhang: Liste aller OCL-Operatoren, Teil 2

Priorität	Operator	Assoziativität	Operanden, Bedeutung
8	==, !=	links	Vergleiche
7	&	links	Zahlen, Boolean: striktes und
6	\	links	Zahlen, Boolean: xor
5	|	links	Zahlen, Boolean: striktes oder
4	&&	links	Boolesche Logik: und
3	||	links	Boolesche Logik: oder
2,7	implies	links	Boolesche Logik: impliziert
2,3	<=>	links	Boolesche Logik: äquivalent
2	? :	rechts	Auswahlausdruck (if-then-else)

SE SOFTWARE ENGINEERING

RWTH AACHEN

Modellbasierte Softwareentwicklung

- 3. Object Constraint Language
- 3.2. Logik

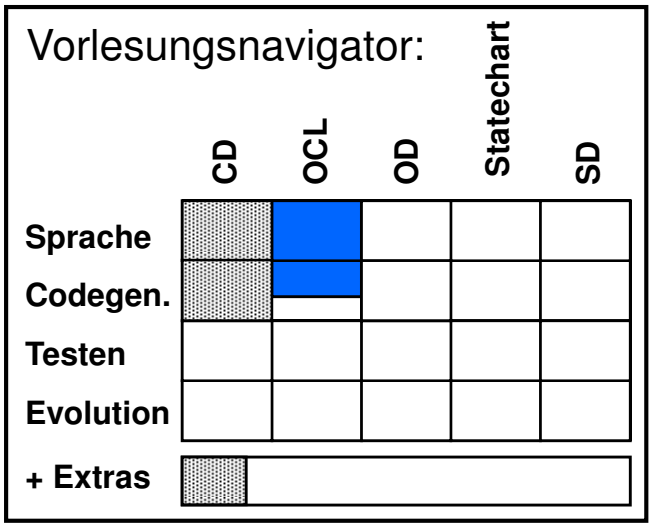
Prof. Dr. Bernhard Rumpe
Lehrstuhl für Software Engineering
RWTH Aachen

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

OCL

context Klasse inv:
    beschränkende Invariante

context Methode
pre: Vorbedingung
post: Nachbedingung



Logik in der OCL

TextAnim

- Boolesche Aussagen über Attribute und Assoziationen werden verknüpft mit
  - Logik-Operatoren: und, oder, genau-dann-wenn, impliziert, nicht
    &&, ||, <=>, implies, not
  - Quantoren: Es existiert, Für alle
    exists, forall
  - Vergleich: ==
- Boolesche Aussagen sind zweiwertig: **true** oder **false**
- In einer Implementierung können undefinierte Werte auftreten:
  - Programm stürzt ab
  - Keine Terminierung, z.B. unendliche Schleife
  - Ungültiger Wert
    (Referenz existiert nicht, Enumeration Out-of-Range)
- Einführung eines nur in der Semantik verwendeten Pseudo-Wertes „**undefined**“

Zwei-wertige Logik: Beispiel Konjunktion

- Wahrheitstabelle für die Konjunktion:

A && B	**true**	**false**
true		
false		

- Es gelten eine Reihe schöner Gesetze:
  - Assoziativität
  - Kommutativität
  - Involution

Drei-wertige Logik: Beispiel Konjunktion

write

- Wahrheitstabelle für die erweiterte Konjunktion:

Mathematischer Pseudowert

A && B	**true**	**false**	**undef**
true	**true**	**false**	
false	**false**	**false**	
undef			

- Welche Gesetze gelten dann noch?
  - Assoziativität (a && b) && c <=> a && (b && c)
  - Kommutativität a && b <=> b && a
  - Involution a && a <=> a

OCL

Varianten Drei-wertiger Logik: (b) Strikte Auswertung

A && B	**true**	**false**	**undef**
true	**true**	**false**	**undef**
false	**false**	**false**	**undef**
undef	**undef**	**undef**	**undef**

- Beispiele:
  - Pascal, strikte Auswertung von & in Java
- Vorteile:
  - implementierbar
  - Auswertungsreihenfolge egal, da assoziativ, kommutativ
- Nachteile:
  - immer beide Argumente auszuwerten
  - umständlich für die Logik, da drei Fälle

Varianten Drei-wertiger Logik: (c) Sequentielle Auswertung

A && B	**true**	**false**	**undef**
true	**true**	**false**	**undef**
false	**false**	**false**	**false**
undef	**undef**	**undef**	**undef**

- Beispiele:
  - && in C, C++, Java, ...
  - Abbruch von Befehlssequenzen in bash: svn up && make
- Vorteile:
  - leicht implementierbar
  - effizient: wenn links false, wird rechts nicht ausgewertet
- Nachteile:
  - nicht kommutativ
  - sehr umständlich für die Logik: weiter drei Fälle und die Booleschen Gesetze gelten nicht

Varianten Drei-wertiger Logik: (d) Kleene-Logik

A && B	**true**	**false**	**undef**
true	**true**	**false**	**undef**
false	**false**	**false**	**false**
undef	**undef**	**false**	**undef**

- Beispiele:
  - Keine (gängige) Programmiersprache
- Vorteile:
  - implementierbar
  - Booleschen Gesetze gelten: assoziativ, kommutativ, ...
- Nachteile:
  - beide Argumente parallel auszuwerten!
  - umständlich für die Logik, da weiter drei Fälle

Varianten Drei-wertiger Logik: (e) Lifting von undef

A && B	**true**	**false**	**undef**
true	**true**	**false**	**false**
false	**false**	**false**	**false**
undef	**false**	**false**	**false**

undef und false werden in der Logik identifiziert: Also nur zweiwertige Logik!

- Beispiele:
  - Verifikationswerkzeug Isabelle
- Vorteile:
  - einfache Gesetze und Beweisführung
  - einfache Formulierung von Eigenschaften
- Nachteile:
  - nicht vollständig auswertbar

Zweiwertige Semantik und Lifting

- Idee des Lifting:
  - Unterscheidung von Termen mit booleschen Werten mit drei Ergebnissen, wie
    - a==5, isOpen()
  - und Logik-Ausdrücken mit zwei Ergebnisswerten
- Lifting von „undef“ auf „false“ durch einen expliziten Operator λ
  - λ **true** == **true**
  - λ **false** == **false**
  - λ **undef** == **false**
- Aufbau von Logik-Ausdrücken unter Verwendung des Lifters λ :
  - λ(a==5) implies λ(isOpen())
  - λ(b==1/0)
- Lifter λ kann in der OCL syntaktisch erkannt werden und muss deshalb nicht explizit eingesetzt werden. Ein Logik-Ausdruck:
  - b==1/0

Implementierung des Lifting λ

- Problem: λ **undef** == **false** ist nicht implementierbar
- Praxis in Java zeigt „undef“ zumeist durch
  - 1) abnormale Fehler,
  - 2) unendliche Rekursion
  - 3) eher selten tritt Nichtterminierung durch Schleifen auf
- Fall 1&2 liefern Exceptions (zB Stack Overflow) und können abgefangen werden.
- Damit ist Lifting eines Ausdrucks x partiell implementierbar:

Implementierung des Lifting λ

- Problem: λ **undef** == **false** ist nicht implementierbar
- Praxis in Java zeigt „undef“ zumeist durch
  - 1) abnormale Fehler,
  - 2) unendliche Rekursion
  - 3) eher selten tritt Nichtterminierung durch Schleifen auf
- Fall 1&2 liefern Exceptions (zB Stack Overflow) und können abgefangen werden.
- Damit ist Lifting eines Ausdrucks x partiell implementierbar:

Java

```
boolean res;
try {
    res = x;                    // Auswertung von Ausdruck x
} catch(Exception e) {
    res = false;
}
```

Implementierung der Konjunktion &&

- Anwendung des Lifting bei (a && b):
  - boolean res;
    try {
      res = a; // *Auswertung Ausdruck a*
    } catch(Exception e) {
      res = false;
    }
    if(res) { // *Effizienz: b nur auswerten, wenn a wahr*
      try {
        res = b; // *Auswertung Ausdruck b*
      } catch(Exception e) {
        res = false
      }
    }

Java

- Bis auf Nichtterminierung ist Implementierung identisch mit der Semantik des Operators &&
- Analog lassen sich alle Logik-Operatoren (fast) implementieren.

Vergleiche mit ==

- Operatoren, die auf undefinierten Argumenten definierte Werte ergeben können heißen „nicht-strikt“

- Boolesche Operatoren, Fallunterscheidungen, und das let-Kontrukt sind nicht strikt:
  - if true then a else undef äquiv.: a
  - let a=1/0 in 3+7 äquiv.: 3+7

- Der Vergleichs-Operator == (sowie != und equals()) sind nach Konvention strikt:
  - (undef == undef) äquiv.: **undef**

Die Booleschen Operatoren: Definition durch Wahrheitstafeln

home

!a	
!true	
!false	
!undef	

a xor b	true	false	undef
true			
false			
undef			

a && b	true	false	undef
true			
false			
undef			

a || b	true	false	undef
true			
false			
undef			

a implies b	true	false	undef
true			
false			
undef			

a <=> b	true	false	undef
true			
false			
undef			

Anhang: Booleschen Operatoren

!a	
!true	false
!false	true
!undef	true

a xor b	true	false	undef
true	false	true	true
false	true	false	false
undef	true	false	false

a && b	true	false	undef
true	true	false	false
false	false	false	false
undef	false	false	false

a || b	true	false	undef
true	true	true	true
false	true	false	false
undef	true	false	false

a implies b	true	false	undef
true	true	false	false
false	true	true	true
undef	true	true	true

a <=> b	true	false	undef
true	true	false	false
false	false	true	true
undef	false	true	true

Anhang: Vergleich der Logiken anhand der Konjunktion

$a \wedge b$	true	false
true	true	false
false	false	false

(a) Klassische 2-wertige Logik

a & b	true	false	undef
true	true	false	undef
false	false	false	undef
undef	undef	undef	undef

(b) strikte Auswertung, wie Java-&

a and b	true	false	undef
true	true	false	undef
false	false	false	false
undef	undef	false	undef

(c) parallele Auswertung, Kleene-Logik

a && b	true	false	undef
true	true	false	undef
false	false	false	false
undef	undef	undef	undef

(d) sequentiell, wie Java-&&

$a \wedge b$	true	false	undef
true	true	false	false
false	false	false	false
undef	false	false	false

(e) Lifting: **undef** wird wie **false** verwendet

Anhang: Fallunterscheidung

- Fallunterscheidungen sind funktional:
- Es entsteht immer ein Wert: also ist der else-Teil notwendig.
- Zwei äquivalente Beschreibungsformen:

if then else		
if true	then a else b	a
if false	then a else b	b
if undef	then a else b	b

?:		
true	? a : b	a
false	? a : b	b
undef	? a : b	b

SE SOFTWARE ENGINEERING

RWTH AACHEN

Modellbasierte Softwareentwicklung

- 3. Object Constraint Language
- 3.3. Collections

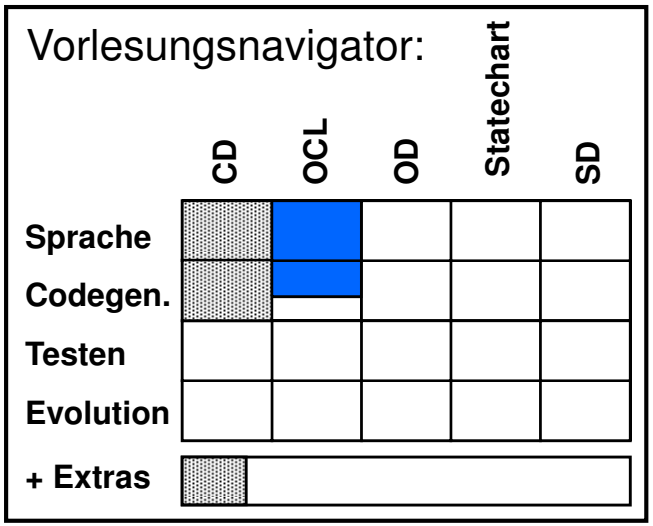
Prof. Dr. Bernhard Rumpe
Lehrstuhl für Software Engineering
RWTH Aachen

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

OCL

context Klasse inv:
beschränkende Invariante

context Methode
pre: Vorbedingung
post: Nachbedingung



Collections in OCL

- Besonders wichtig durch Navigation entlang von Assoziationen
- Java nutzt als Typisierung z.B. Set
- Die hier vorgestellte OCL beschreibt auch den Argumenttyp
  - Vorteil: Typsicherheit auch auf Argumentebene

- Set<X> stellt Mengen über Typ X dar

- List<X> stellt Listen dar:
  - Elemente über Index 0..(Länge-1) zugreifbar
  - Mehrfachvorkommen möglich

- Collection<X> ist Supertyp von Set<X> und List<X>
  - gemeinsames Interface der beiden Collections

Collections: Schachtelung, Subtyphierarchie

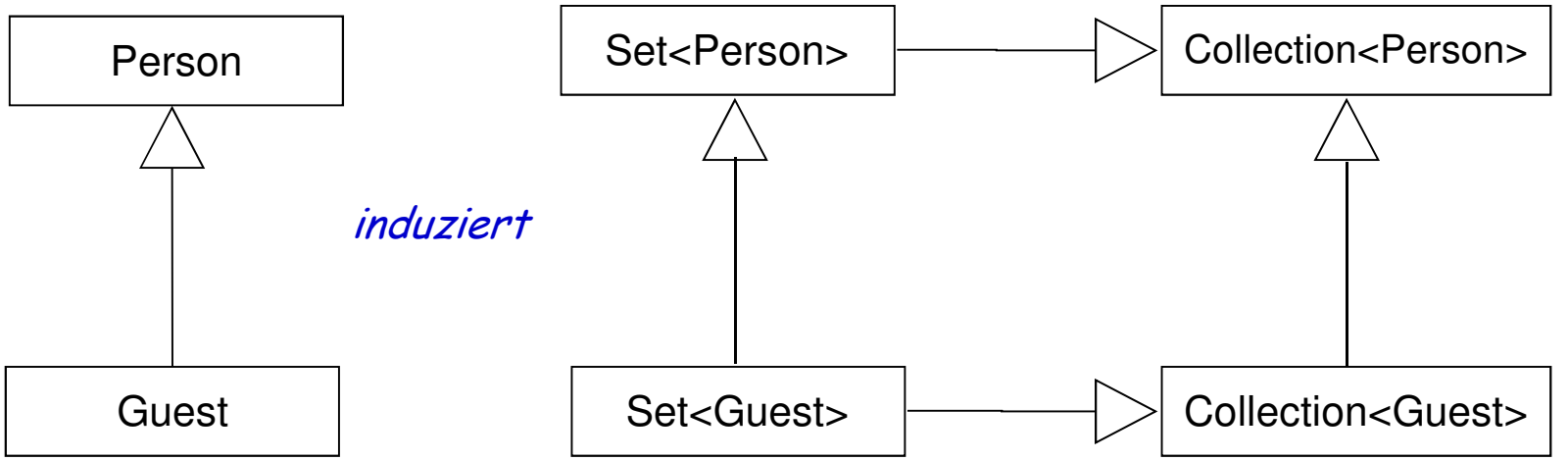
- Collectiontypen können geschachtelt werden:
- inv:

OCL

```
let Set<int>          si  = { 1, 3, 5 };
    Set<Set<int>>     ssi = { {}, {1}, {1,2}, si };
    List<Set<int>>    lsi = List{ {1}, si, {}, si }
in ...
```

- Subtyphierarchie wird durch Collection- und Elementtyp gebildet:

CD



Collection-Vergleiche

- Vergleich == auf Collections benötigt Vergleich auf den Elementen:
  - Elementvergleich ist == für Grunddatentypen
  - und equals() für Objekttypen
- Collections haben in OCL keine „Objektidentität“,
  - a==b vergleicht daher beide Collections auf inhaltliche Gleichheit
- Ist X ein Grunddatentyp oder selbst Collection, so gilt für Set<X>:
  - context Set<X> sa, Set<X> sb inv:
    sa==sb <=> (forall a in sa: exists b in sb: a==b) &&
    (forall b in sb: exists a in sa: a==b)
- Für Objekttypen X gilt:
  - context Set<X> sa, Set<X> sb inv:
    sa==sb <=> (forall a in sa: exists b in sb: a.equals(b)) &&
    (forall b in sb: exists a in sa: a.equals(b))
  - Methode equals() darf mit großer Vorsicht redefiniert werden
- Vergleich für Listen ist analog realisiert.

Aufschreibung und Typisierung

- Beispiele für Mengen:
  - Set{}, Set{ 2,3,5 }, Set{ “text“, “teile“ }
  - {}, { 2,3,5 }, {2}, {{2}}
  - Person
- Analog für Listen, jedoch mit List{...}

Ein Klassenname steht in OCL für die Extension: also die Menge aller derzeit existierenden Objekte

- Typisierung:
- Welchen Typ hat:
  - Set{ “text“, <Auction> a }
- Man könnte den gemeinsamen Supertyp Set<Object> verwenden.
- Praxis zeigt:
  - Methodisch besser: Set-Typ hängt nur vom ersten Argument ab
  - Ergebnis wäre Set<String> und Term ist fehlerhaft.
  - Deshalb explizite Typangabe:
    Set{ <Object>“text“, <Auction> a }

Mengen- und Listenkomprehension

- Kurzform für Mengen und Listen ganzer Zahlen und Characters:
  - Set{‘a‘..‘c‘} == {‘a‘, ‘b‘, ‘c‘}
  - List{-1..1,3..7,14} == List{-1,0,1, 3,4,5,6,7, 14}

OCL

- Umwandlung Menge und Liste: asSet und asList
  - Set{ *beschreibung* } == List{ *beschreibung* }.asSet
- Allgemeine Form der Komprehension
  - List{ *expr* | *beschreibung* }

Platzhalter für entsprechende Terme

Komprehension 1: der Generator

- Form der Charakterisierung:
  - v in *Liste/Menge*
- mit neuer Variable v, die über die Liste iteriert

- Beispiele
  - List{ x*x | x in List{1..5} } == List{1,4,9,16,25}

OCL

  - context Auction a inv: ...
    List{ m.time.asMsec() | Message m in a.message }

Komprehension 2: der Filter

- Form der Charakterisierung:
  - *condition*
- mit einer Booleschen Expression, die einen Teil selektiert

- Es gilt für die einfache Liste:
  - List{ *expr* | *condition* } ==
    if *condition* then List{ *expr* } else List{}

OCL

- Mächtig durch Kombination mit Generator:
  - List{ x*x | x in List{1..8}, !even(x) } == List{1,9,25,49}

OCL

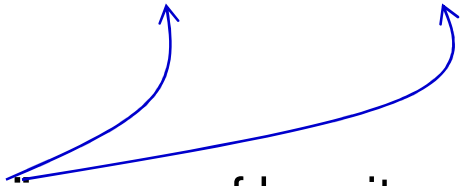
  - context Auction a inv:
    ... List{ m.time.asMsec()
    | m in a.message, m.time.lessThan(a.startTime) }

Komprehension 3: Zwischenergebnisse

- Lokale Variablen können als Zwischenergebnisse definiert werden
  - *v = expr*
  - *type v = expr*

OCL

- Beispiel:
  - List{ y | x in List{1..8}, int y = x*x, !even(y) } == List{1,9,25,49}



- Beschreibungselemente können auf bereits vorher definierte Elemente zurückgreifen

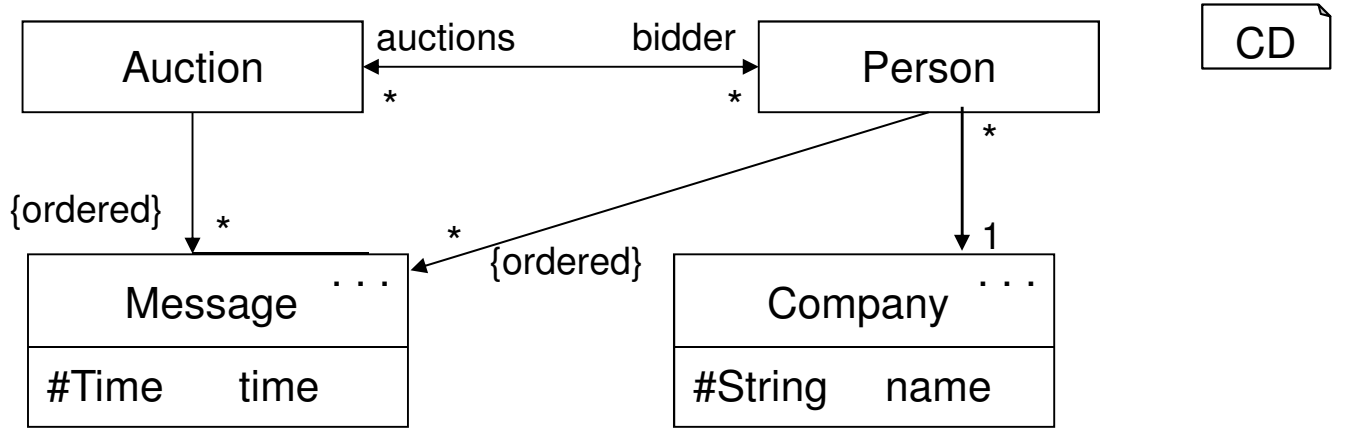
Komprehension: Abschluss

- Beschreibungsmächtigkeit durch Kombination der drei Formen ziemlich gut.
  - Leider bietet der UML 2 Standard diese Formen nicht an.
  - Sie wurden aus funktionalen Sprachen (Gofer, Haskell) entlehnt

```
List{ z+"?" |   x  in  List{"Spiel", "Feuer", "Flug"},
                y  in  List{"zeug", "platz"},
                String  z = x+y,
                z  !=  "Feuerplatz"                    }
 ==
```

OCL

Navigation in OCL - 1



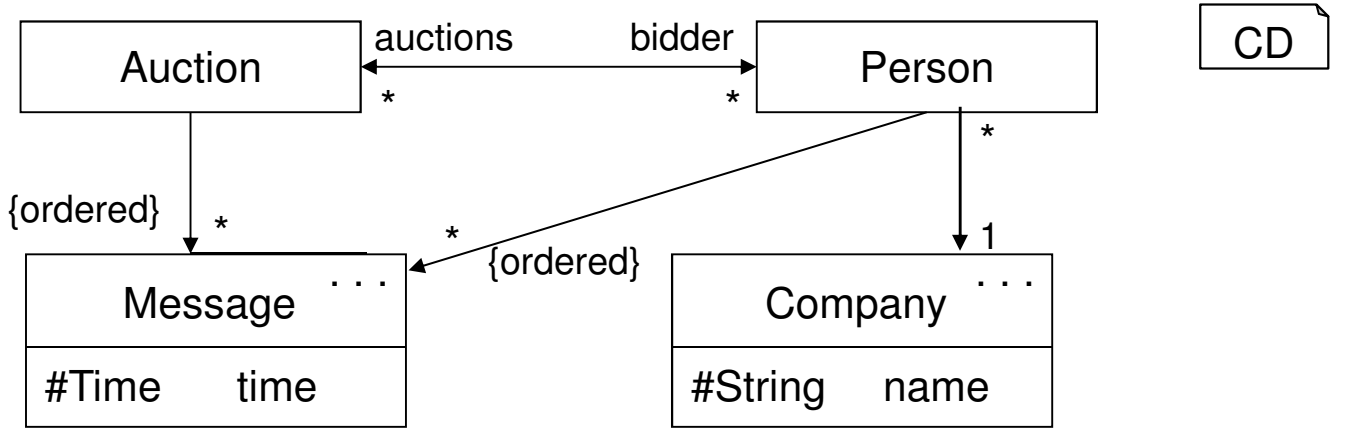
write

- Menge der Bieter der Auktion a:

OCL

- Menge der Namen beteiligter Firmen von a:
- Menge der Nachrichten von a:
- Menge der Personen einer Firma c:

Navigation in OCL - 2

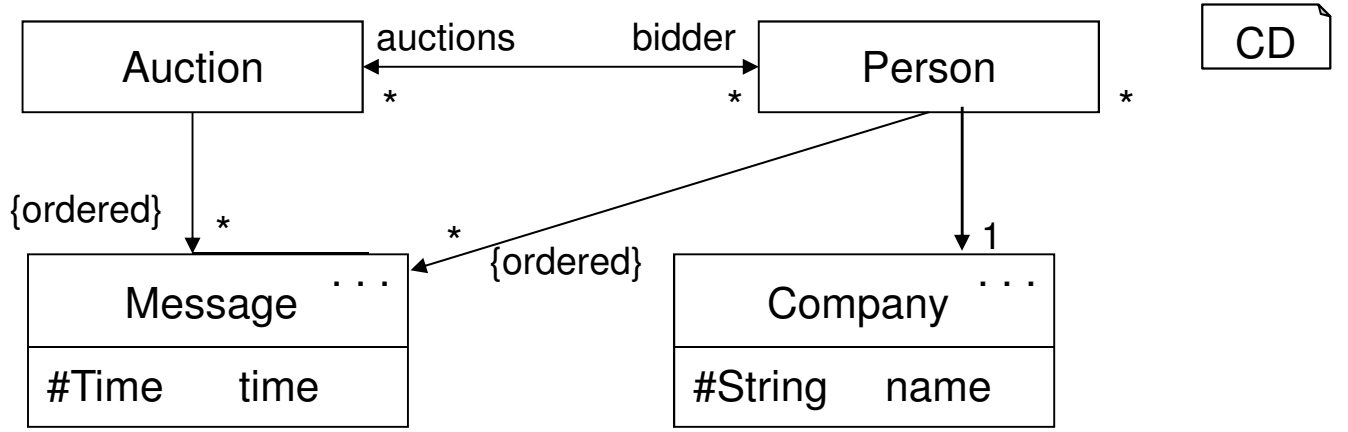


write

- Menge der Auktionen an denen die Firma c beteiligt ist:

OCL

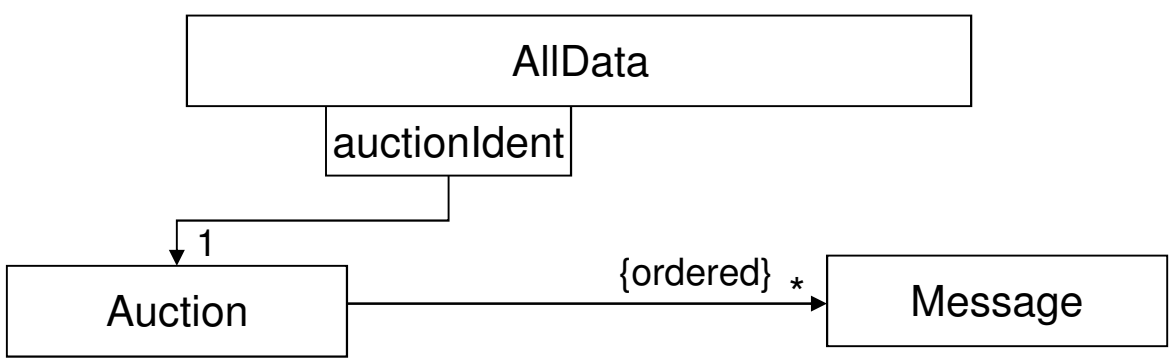
Flattening-Operator in der Navigation



OCL

- Menge der Auktionen an denen die Firma c beteiligt ist: c.person.auctions
  - c vom Typ Company
  - c.person vom Typ Set<Person>
  - c.person.auctions wäre nun vom Typ Set(Set(Auction))
- Aber automatisches Flatten entfernt eine Hierarchiestufe:
  - Operator „flatten“ wird bei Navigation überall implizit eingesetzt, wo eine Collection als Ausgangstyp steht
  - c.person.auctions == { p.auctions | p in c.person }.flatten

Qualifizierte Navigation



- Normale Navigation: ad.auction ergibt Set(Auction)
- Qualifizierte Navigation: ad.auction[ident] ergibt Auction
- Beispiel:
  - context AllData ad, Auction a inv:

    OCL

    ad.auction[a.auctionIdent] == a &&
    ad.auction[a.auctionIdent] in ad.auction;
- Bei {ordered}-Assoziationen ist der Index ganzzahlig ab 0.

  a.message[0] in WelcomeMess

  WelcomeMess.containsAll(Auction.message[0])

Menge der jeweils ersten Nachricht jeder Auktion

Quantoren

- forall und exists bilden Umsetzung der aus der Mathematik bekannten Quantoren.
- Beispiele:
  - forall a in Auction, p in Person, m in a.message:
    p in a.bidder implies m in p.message

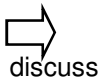
OCL

  - exists a in Auction:
    a.kategorie == “Uhr“

- Es gilt:
  - (exists *var* in *collExpr*: *expr*)
    <=> !(forall *var* in *collExpr*: *!expr*)

Eigenschaften des All-Quantors

- Dies kann erfüllt werden:
  - forall a in Auction: even(1/0)



- Generell gilt:
  - (forall x in Set{}: false) <=> true

- Context-Definition wirkt wie ein All-Quantor. Äquivalent sind:

  - context Auction a, Person p inv:
    forall m in a.message:
    p in a.bidder implies m in p.message

  - inv:
    forall a in Auction, p in Person, m in a.message:
    p in a.bidder implies m in p.message

Berechenbarkeit der Quantoren

- Quantoren unterscheiden die First-Order-Logik (FOL) von der Aussagenlogik
- Quantoren haben in der FOL einen unendlichen Suchraum
- Deswegen ist FOL nicht allgemein ausführbar.
- Beispiel:
  - inv:

    exists int a, b, c, n: n>2 && a^n == b^n + c^n

OCL

- Aber: objekt-wertige Quantoren in OCL werden über die Mengen der im Moment existenten Objekte interpretiert.
  - Diese Mengen sind endlich: daher die OCL-Quantoren „berechenbar“.

Spezialoperator „iterate“

- Ist ein Hilfskonstrukt zur Akkumulation eines Ergebnisses aus einer Collection.
  - iterate{ *elementVar* in *collectExpr*;
    Type accumulatorVar = initExpr :
    accumulatorVar = expr
    }

OCL

- Es simuliert Iteration durch eine Schleife:
  - Accumulator wird initial besetzt und dann Iteration neu berechnet.
  - Dabei iteriert *elementVar* über die Collection.
- Beispiel: Summe
  - iterate { elem in Auction;
    int acc = 0 :
    acc = acc+elem.numberOfBids
    }

OCL

- Wichtig: Collection ist eine Liste oder die Verarbeitung kommutativ & assoziativ, da sonst Ergebnis nicht eindeutig!

Spezialoperator „defined“:

- Selten ist es notwendig, über die Definiertheit eines Wertes zu sprechen:
  - defined(...)

OCL

- Anwendung zum Beispiel:
  - context Auction a inv:
    let Message mess = a.person[0]
    in
    defined(mess) implies ...

OCL

- Beispiel, es gilt:
  - inv:
    !defined(1/0)

OCL

Anhang: Mengenoperationen

- Für Mengen Set<X> stehen folgende an Java angelehnte Operationen zur Verfügung:

Signatur

- Set<X> add(X o); // *A∪{o}*
- Set<X> addAll(Collection<X> c); // *A∪c Vereinigung*
- boolean contains(X o); // *o∈A Schnittmenge*
- boolean containsAll(Collection<X> c); // *c⊆A ist Teilmenge?*
- int count(X o);
- boolean isEmpty;
- Set<X> remove(X o);
- Set<X> removeAll(Collection<X> c); // *A\c „ohne"*
- Set<X> retainAll(Collection<X> c); // *A∩B Schnittmenge*
- Set<X> symmetricDifference(Set<X> s); // *A\c ∪ c\A*
- int size;
- X flatten; // *X ist ein Collection-Typ*
- List<X> asList;

Anhang: Listenoperationen – Teil 1

- Für Listen List<X> stehen folgende an Java angelehnte Operationen zur Verfügung (Index 0 indiziert das erste Element):

Signatur

- List<X> add(X o); // *hinten anfügen*
- List<X> add(int index, X o); // *Index beginnt mit 0*
- List<X> prepend(X o); // *vorn anfügen*
- List<X> addAll(Collection<X> c);
- List<X> addAll(int index, Collection<X> c); // *Collection ab Index*
- boolean contains(X o);
- boolean containsAll(Collection<X> c);
- X get(int index);
- X first;
- X last;
- List<X> rest;

Verwendung als readOnly-Attribut: analog zu size. Dadurch werden Klammern überflüssig

Anhang: Listenoperationen - Teil 2

Signatur

- int indexOf(X o);
- int lastIndexOf(X o);
- boolean isEmpty;
- int count(X o);
- List<X> remove(X o);
- List<X> removeAtIndex(int index);
- List<X> removeAll(Collection<X> c);
- List<X> retainAll(Collection<X> c); // *Schnittmenge*
- List<X> set(int index, X o);
- int size;
- List<X> subList(int fromIndex, int toIndex);
- List<Y> flatten; // *X hat die Form Collection<Y>*
- Set<X> asSet;

Anhang: Eigenschaften von Listen

- Allgemeine Rechenregeln beschreiben die Eigenschaften der Listen
  - Nutzbar u.a. für Optimierungen oder zu Refactorisierung von Code
- Hier aber Eigenschaften der Listen an konkreten Beispielen zum einfacheren Verständnis:

OCL

```
List{0,1}                    != List{1,0};
List{0,1,1}                  != List{0,1};
List{0,1,2}.add(3)           == List{0,1,2,3};
List{'a','b','c'}.add(1,'d') == List{'a','d','b','c'};
List{0,1,2}.prepend(3)       == List{3,0,1,2};
List{0,1}.addAll(List{2,3})  == List{0,1,2,3};
List{0,1,2}.set(1,3)         == List{0,3,2};
List{0,1,2}.get(1)           == 1;
List{0,1,2}.first            == 0;
List{0,1,2}.last             == 2;
```

Anhang: Eigenschaften von Listen (ff.)

OCL

-
- List{0,1,2}.rest == List{1,2};
- List{0,1,2,1}.remove(1) == List{0,2};
- List{0,1,2,3}.removeAtIndex(1) == List{0,2,3};
- List{0,1,2,3,2,1}.removeAll(List{1,2}) == List{0,3};
- List{0..4}.subList(1,3) == List{1,2};
- List{0..4}.subList(3,3) == List{};

Anhang: Collection-Operationen

- Collection<X> hat als Supertyp die gemeinsame Signatur von Set<X> und List<X>:

Signatur

- Collection<X> add(X o);
- Collection<X> addAll(Collection<X> c);
- boolean contains(X o);
- boolean containsAll(Collection<X> c);
- boolean isEmpty;
- int count(X o);
- Collection<X> remove(X o);
- Collection<X> removeAll(Collection<X> c);
- Collection<X> retainAll(Collection<X> c);
- int size;
- Collection<Y> flatten;
- Set<X> asSet;
- List<X> asList;

Anhang: Umwandlung vs. Typkonversion

- Die Umwandlung von Mengen in Listen legen wir hier nicht explizit fest, fordern aber Eindeutigkeit:
  - forall s1 in Set<X>, s2 in Set<X>:
    s1 == s2 implies s1.asList == s2.asList

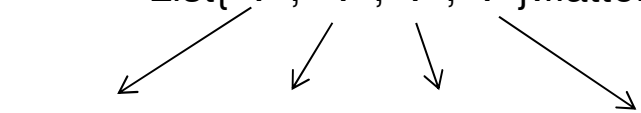
- Die Umwandlung asSet verändert ihr Argument, die Typkonversion (Set<X>) nur das „Aussehen“ des Arguments nach außen:
  - let Collection<int> ci = List{1,2,1};
    in
    ci.asSet == {1,2} &&
    ci.asList == List{1,2,1} &&
    ci.asSet.asList.size == 2 &&
    (List<int>) ci == List{1,2,1} &&
    !((Set<int>) ci == {1,2}) // *linke Seite ist undefiniert*

Anhang: Anwendung des Flattening-Operators

- Anwendungsformen des flatten-Operators:
  - Set<X> Set<Set<X>>.flatten;
  - List<X> Set<List<X>>.flatten;
  - Collection<X> Set<Collection<X>>.flatten;
  - List<X> List<Set<X>>.flatten;
  - List<X> List<List<X>>.flatten;
  - List<X> List<Collection<X>>.flatten;
  - Collection<X> Collection<Set<X>>.flatten;
  - List<X> Collection<List<X>>.flatten;
  - Collection<X> Collection<Collection<X>>.flatten;
- Faustformel:
  - List > Collection > Set:
  - Die stärkere Collection-Form zieht!

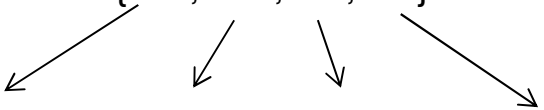
Anhang: Beispiele für Flattening

List{ . , . , . , . }.flatten == List{1 , 2 , 3 , 3 , 1 , 4 , 5 , 6}



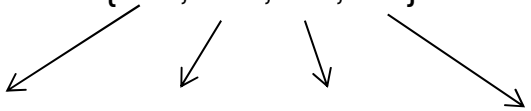
List {1,2,3} List {3} List {1,4,5} List {6}

Set{ . , . , . , . }.flatten == Set{1 , 2 , 3 , 4 , 5 , 6}



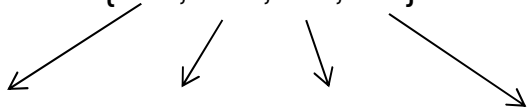
Set {1,2,3} Set {3} Set {1,4,5} Set {6}

List{ . , . , . , . }.flatten == List{1 , 2 , 3 , 3 , 1 , 4 , 5 , 6}
oder List{3 , 2 , 1 , 3 , 4 , 1 , 5 , 6}
oder eine andere der insgesamt $3!^2$=36 Permutationen



Set {1,2,3} Set {3} Set {1,4,5} Set {6}

Set{ . , . , . , . }.flatten == List{1 , 2 , 3 , 3 , 1 , 4 , 5 , 6}
oder List{6 , 3 , 1 , 2 , 3 , 1 , 4 , 5}
oder eine andere der insgesamt 4!=24 Permutationen



List {1,2,3} List {3} List {1,4,5} List {6}

SE SOFTWARE ENGINEERING

RWTH AACHEN

Modellbasierte Softwareentwicklung

- 3. Object Constraint Language
- 3.4. Queries, Methoden

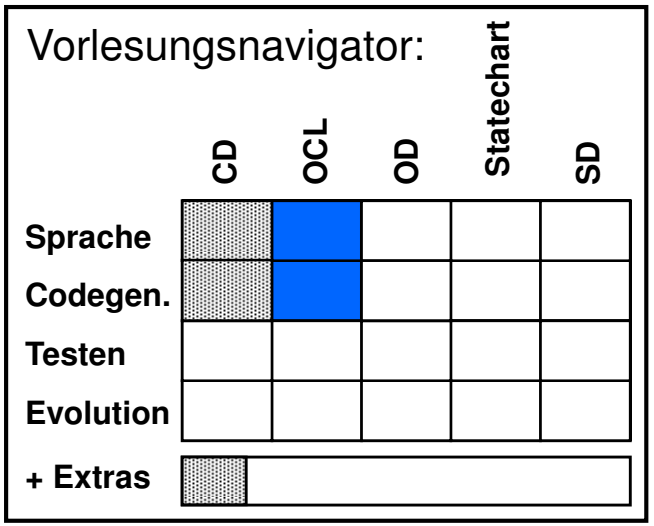
OCL

context Klasse inv:
beschränkende Invariante

context Methode
pre: Vorbedingung
post: Nachbedingung

Prof. Dr. Bernhard Rumpe
Lehrstuhl für Software Engineering
RWTH Aachen

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



Beziehungen zwischen OCL und Methoden

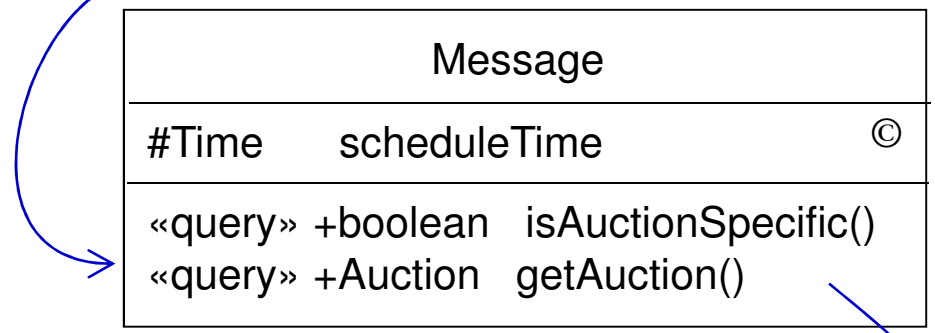
- Zwei grundsätzlich unterschiedliche Nutzungsformen der OCL für Methoden:

- 1) OCL-Aussagen können Methoden / Funktionen nutzen
  - seiteneffektfreie Queries aus dem Objektmodell, oder
  - Funktionen speziell für OCL definiert

- 2) OCL kann Vor-/Nachbedingungen von Methoden beschreiben

Queries

- Eine Query ist eine Methode des zugrunde liegenden Objektmodells. Eine Query ist seiteneffektfrei.
- Queries können mit dem Stereotyp «query» markiert werden.



CD

- context Auction a inv:
  forall p in a.bidder:
  a.message.asSet == { m in p.message |
  m.isAuctionSpecific() && m.getAuction()==a }

OCL

Seiteneffekte in Queries

- OCL wird zum Beispiel beim Testen eingesetzt:

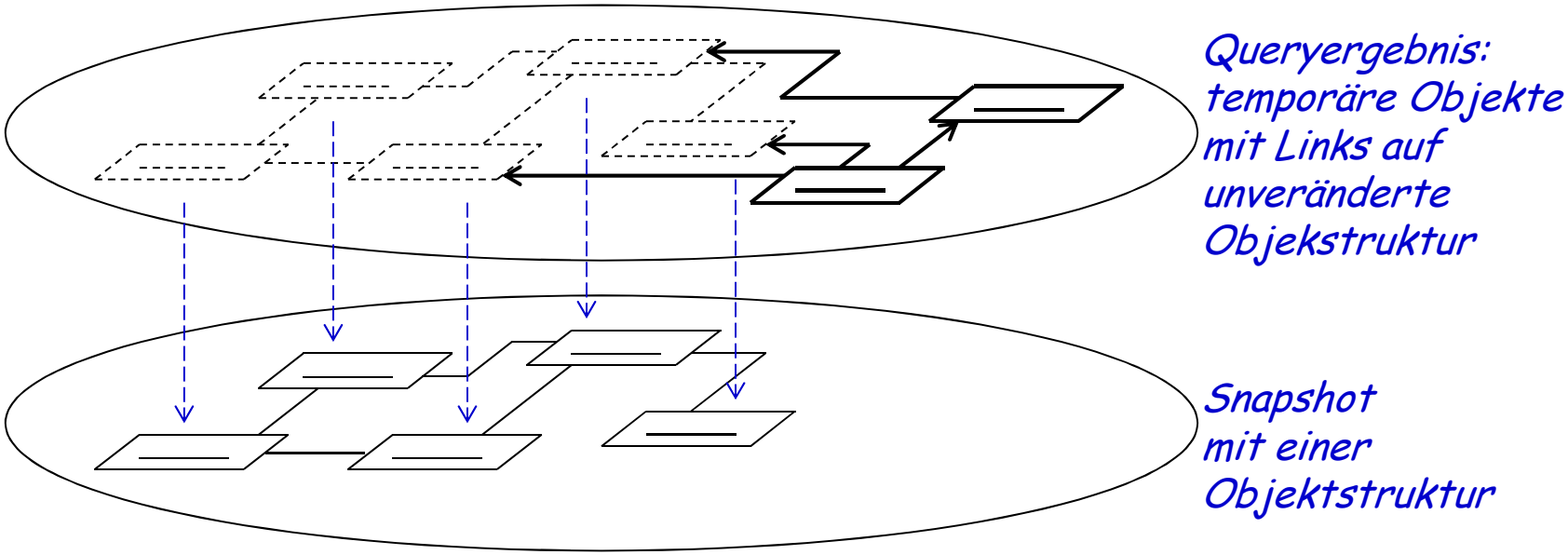
- Die Auswertung von OCL darf deshalb keine Veränderungen des Systemzustands bewirken:
  - keine Attribute dürfen verändert werden!

- Der Stereotyp «query» ist daher gleichzeitig ein Versprechen an den Nutzer und eine Verpflichtung an den Entwickler:
  - Queries dürfen in Subklassen überschrieben werden, aber müssen seiteneffektfrei bleiben
  - Queries dürfen nur andere Queries aufrufen

- Seiteneffektfreiheit kann automatisch geprüft werden.
  - Leider gibt es noch keine Sprache/Compiler, die dies unterstützt

Queries und Objekterzeugung

- Eine Query darf neue Objekte erzeugen und manipulieren
  - Beipiel: Aufbau einer Collection als Ergebnis der Query
- Alte Objektstruktur darf keine Kenntnis der neuen Objekte haben:



- Prüfung, ob eine Methode Query ist, erfordert eine Datenflussanalyse z.B. über den Konstruktor!

Implementierung von Queries in Java

- Java kennt keinen Stereotyp «query»
- Es ist guter Programmierstil, Queries mit
  - `get, is, has` oder `give` beginnen zu lassen
  - Allerdings besteht damit keine Sicherheit, dass es eine Query ist

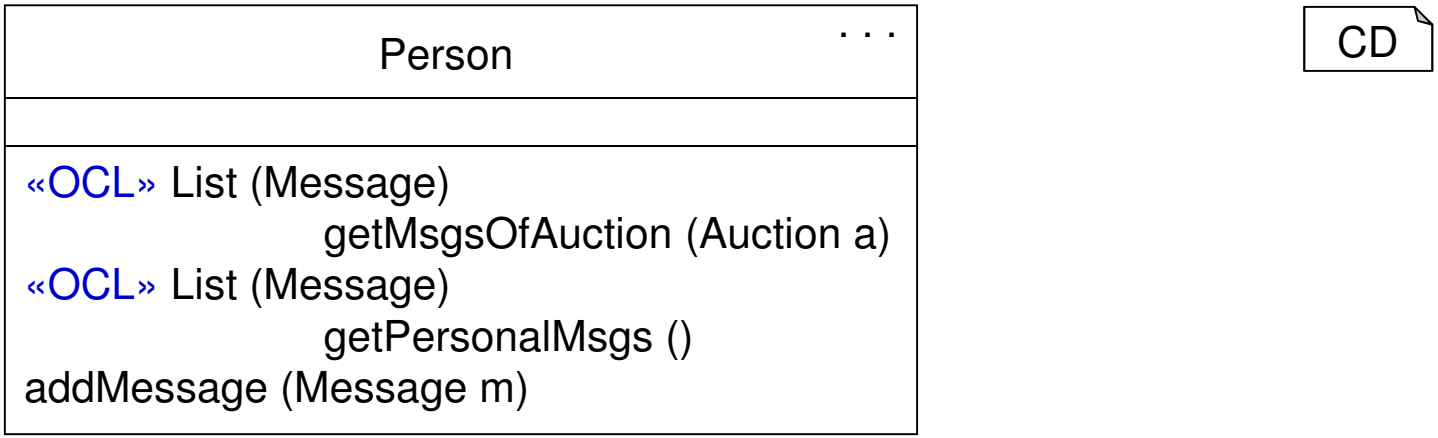
- Umsetzung von OCL in Java
  - a) pragmatisch: Man verlässt sich auf Seiteneffektfreiheit
  - b) konservativ: Man analysiert die genutzten Methoden auf Query-Fähigkeit

- Weiteres Problem: Terminierung, Korrektheit des Ergebnisses
  - Wir verwenden unseren try-catch-Ansatz für boolesche Queries und haben so eine (fast) korrekte Query-Realisierung

«OCL»-Methoden

- Oft ist die Definition wiederverwendbarer Abstraktionen für OCL sinnvoll:
  - Queries sind Teil des zugrunde liegenden Objektmodells
  - OCL erlaubt selbst keine Methodendefinition (außer in let-Konstrukten)

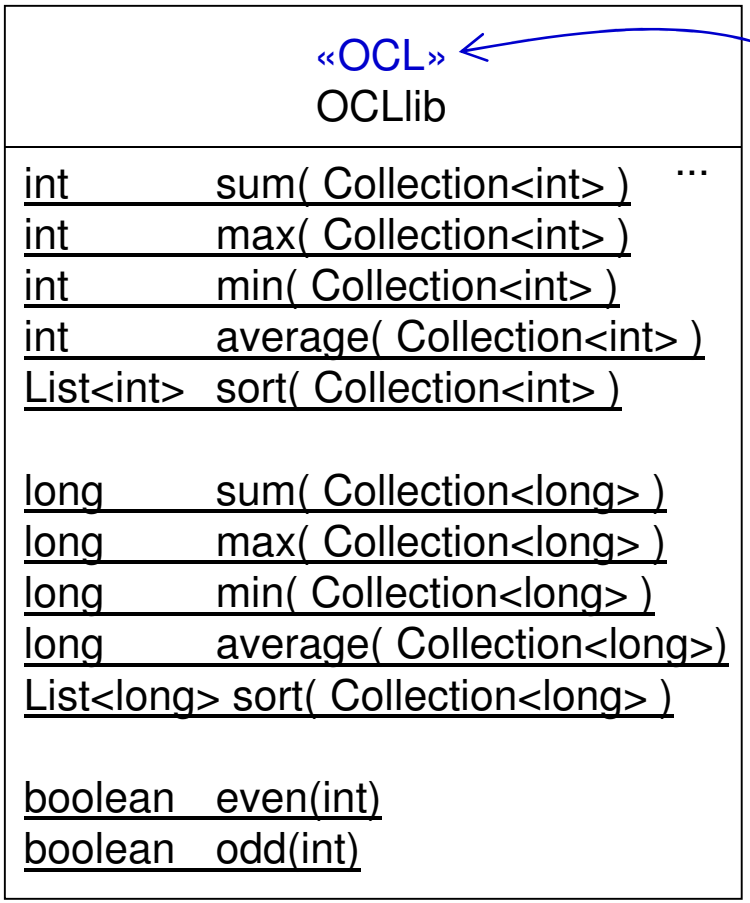
- Mit dem Stereotyp «OCL» gekennzeichnete Methoden sind wie Queries, die aber in einer Implementierung nicht eingebunden sind.
  - Sie können nur in OCL-Bedingungen eingesetzt werden.



Bibliothek von «OCL»-Methoden

- Sinnvoll ist auch eine Bibliothek an «OCL»-Methoden
- Beispiel für deren Inhalt:

CD



Stereotyp «OCL» auf Klasse angewandt: wirkt auf jede einzelne Methode

Statische Methoden: können jederzeit in OCL eingesetzt werden. Beispiel:

min(Auction.bidder.age) >= 18

Methodenspezifikation

- OCL kann eingesetzt werden zur Spezifikation des Effekts einer Methode:
  - Die Vorbedingung beschreibt, welche Bedingung gelten muss, damit die Methode korrekt arbeitet.
  - Die Nachbedingung beschreibt, welchen Effekt die Methode dann hat.

- Beispiel:

Kontext ist nun eine Methode

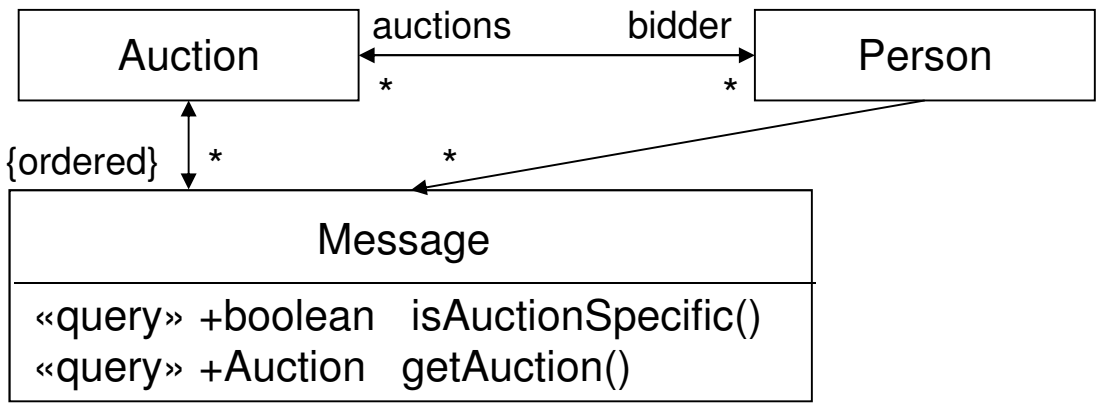
```
context boolean BidMessage.isAuctionSpecific()
pre: true
post: result == true
```

OCL

Es werden keine besonderen Anforderungen an den Zustand des Objekts und der Umgebung zum Zeitpunkt des Aufrufs gestellt

Ergebnis „result" ist in Subklasse Bidmessage immer true

Beispiel: Methode getAuction()



CD

- Beispiel:
- Die Methode Message.getAuction() liefert für auktions-spezifische Nachrichten die Auktion, zu der die Nachricht gehört:

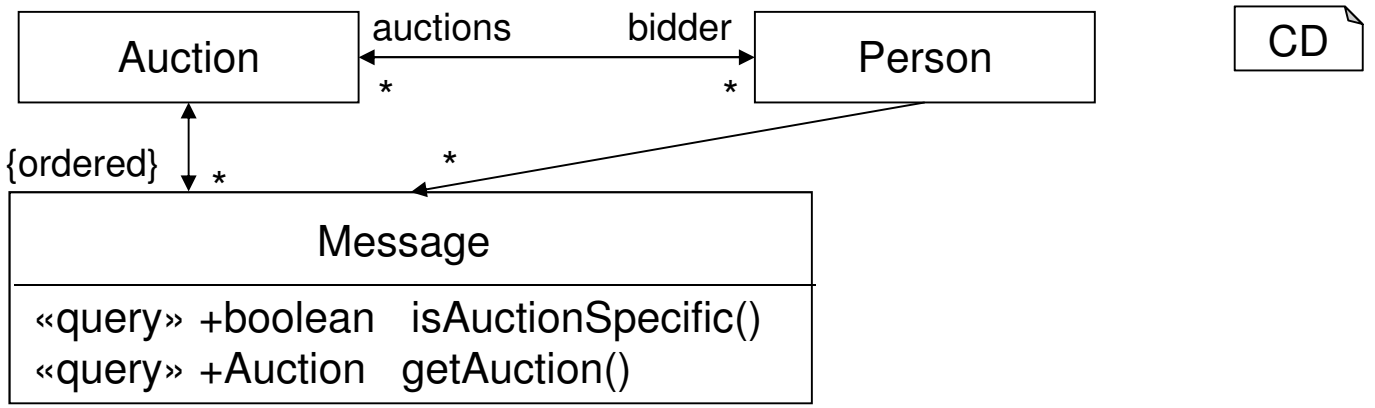
write

OCL

- context Auction Message.getAuction()
  pre:

  post:

Beispiel: Methode getAuction()



- Beispiel:
- Die Methode Message.getAuction() liefert für auktions-spezifische Nachrichten die Auktion, zu der die Nachricht gehört:

OCL

- context Auction Message.getAuction()
  pre: isAuctionSpecific()
  post: this in result.message

Queries können auch hier genutzt werden

„this" verweist auf das Objekt, das zur Methode gehört

Beispiel: Factory-Methode

- MessageFactory erzeugt verschiedene Arten von Nachrichten.
- Hier die Status-Nachricht mit drei Argumenten:

write

OCL

- context «static» StatusMessage
  MessageFactory.createStatusMessage(Time time,
  Auction auction, int newStatus)

  pre:
  post:

Beispiel: Factory-Methode

- MessageFactory erzeugt verschiedene Arten von Nachrichten.
- Hier die Status-Nachricht mit drei Argumenten:

OCL

```
context «static» StatusMessage
   MessageFactory.createStatusMessage(Time time,
                                      Auction auction, int newStatus)
pre:    true

post:   result.time      == time &&
        result.auction   == auction &&
        result.newStatus == newStatus
```

Attribute und Methodenparameter können genutzt werden

Konstruktoren

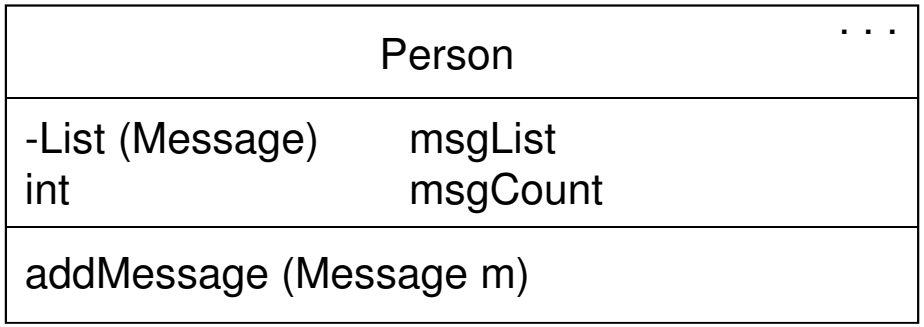
- Konstruktoren lassen sich wie Methoden spezifizieren:

  - context new Auction(BiddingPolicy p)
    pre: p != null
    post: biddingpolicy == p &&
    status == INITIAL &&
    messages.isEmpty;

- Im Konstruktor gilt immer this == result.
- Deshalb lassen sich Attribute mit
  - result.status, this.status oder nur status zugreifen.

@pre-Operator: Attributänderungen

CD



- addMessage fügt eine neue Nachricht hinzu, deren Timestamp neuer sein muss:

write

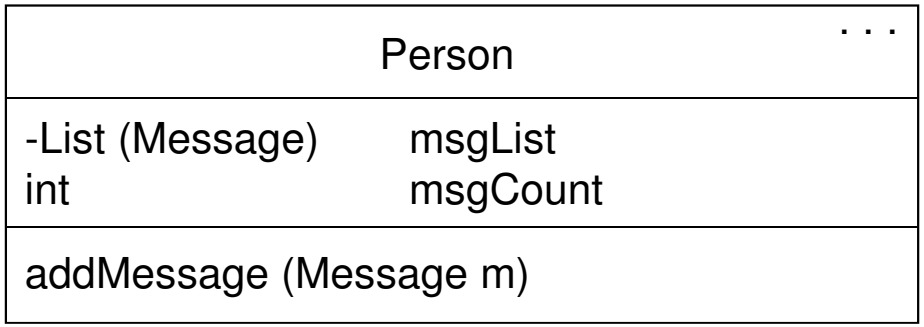
- context Person.addMessage(Message m)
  pre:

  post:

OCL

@pre-Operator: Attributänderungen

CD



- addMessage fügt eine neue Nachricht hinzu, deren Timestamp neuer sein muss:

OCL

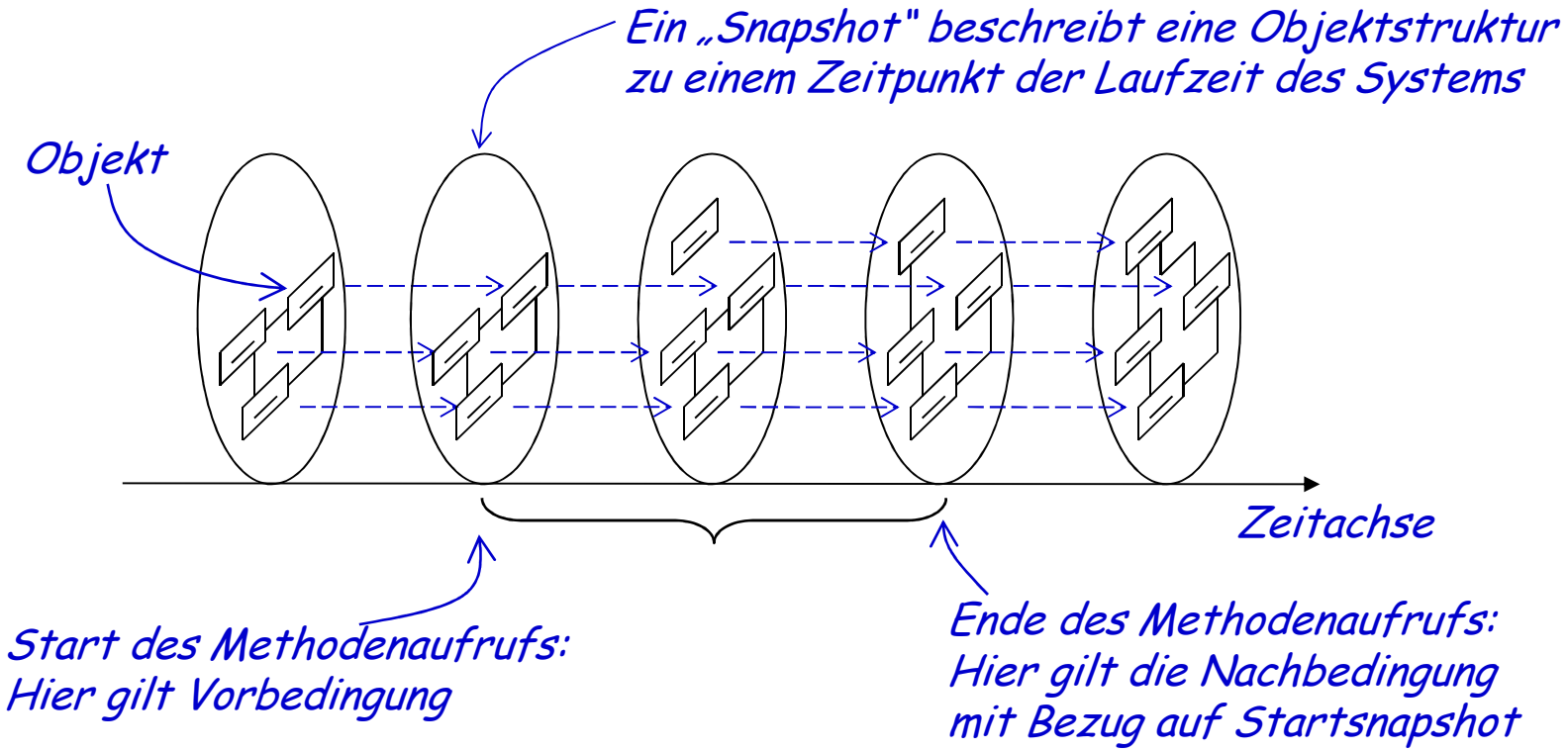
- context Person.addMessage(Message m)
  pre: forall x in msgList: m.time > x.time

  post: msgList == msgList@pre.add(m) &&
  msgCount == msgCount@pre +1

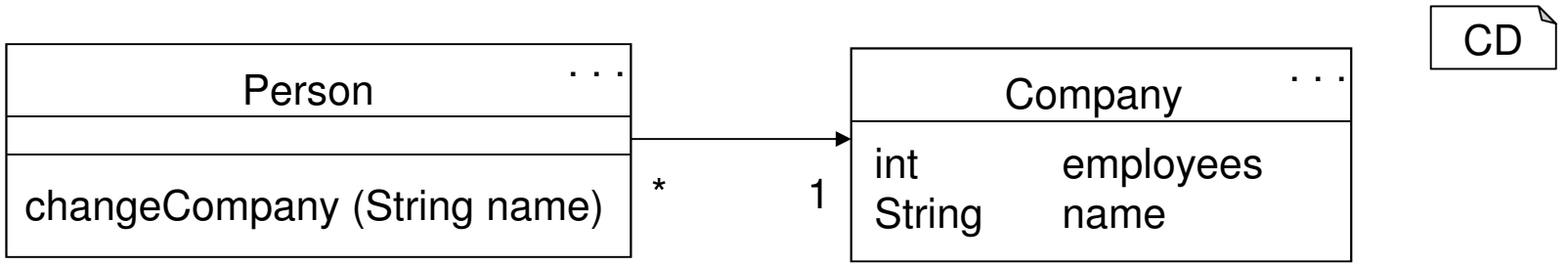
attribut@pre erlaubt den Zugriff auf den Zustand zum Methodenaufruf

Semantik der Methodenspezifikation

- Eine Invariante wird anhand einer Objektstruktur (Snapshot) interpretiert.
- Eine Methodenspezifikation anhand zweier Snapshots:
  - Startsnapshot für die Vorbedingung sowie
  - End- und Startsnapshot (!) für die Nachbedingung:



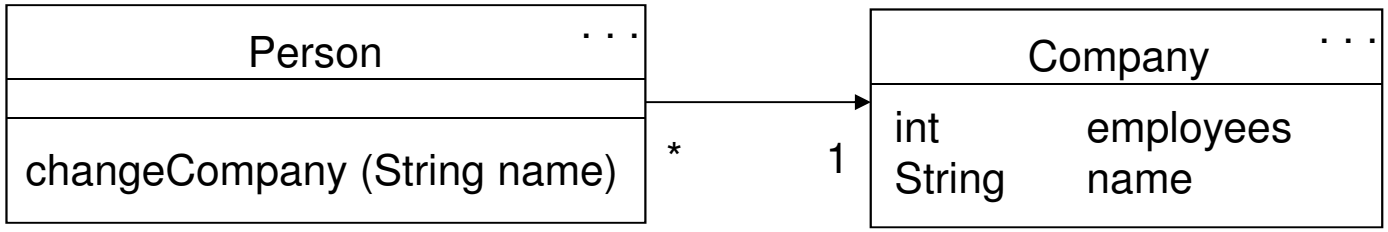
Komplexe Spezifikation



- Mit changeCompany kann eine Person das Unternehmen wechseln:
  - ggf. wird neues Unternehmen angelegt
  - Anzahl der Employees im alten und neuen Unternehmen wechseln!
- Komplexere Situation, deshalb zerlegen wir die Spezifikation in die Fälle:
  - Neues Unternehmen existiert bereits
  - Neues Unternehmen existiert noch nicht

Komplexe Spezifikation – Fall 1

CD



- Fall 1: Firmen-Objekt existiert bereits
- Nebenbedingung: Neue Firma != alter Firma

write

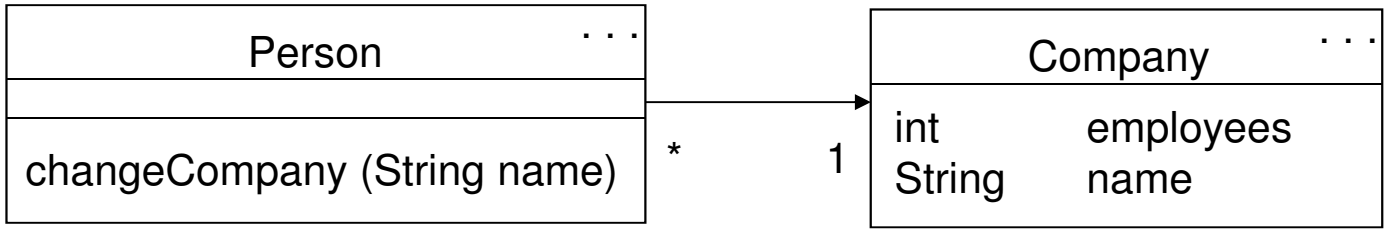
- context Person.changeCompany(String n)
  pre CC1pre: company.name != n &&
  exists Company co: co.name == n

  post CC1post:

OCL

Komplexe Spezifikation – Fall 1

CD



- Fall 1: Firmen-Objekt existiert bereits
- Nebenbedingung: Neue Firma != alter Firma

OCL

- context Person.changeCompany(String n)
  pre CC1pre: company.name != n &&
  exists Company co: co.name == n

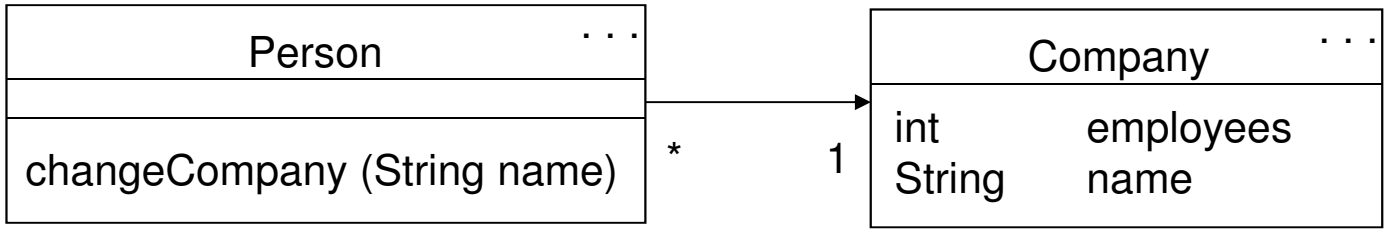
  post CC1post:
  company.name == n &&
  company.employees == company.employees@pre +1 &&
  company@pre.employees == company@pre.employees@pre -1

Vor-/Nachbedingungen mit Namen

Alte Firma, alter Mitarbeiterstand

Komplexe Spezifikation – Fall 2

CD



- Fall 2: Firmen-Objekt existiert noch nicht

write

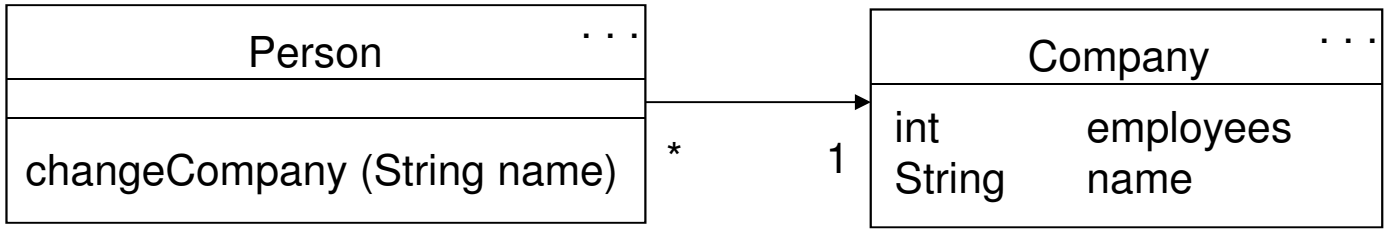
OCL

- context Person.changeCompany(String n)
  pre CC2pre: !exists Company co: co.name == n

  post CC2post:

Komplexe Spezifikation – Fall 2

CD



- Fall 2: Firmen-Objekt existiert noch nicht

OCL

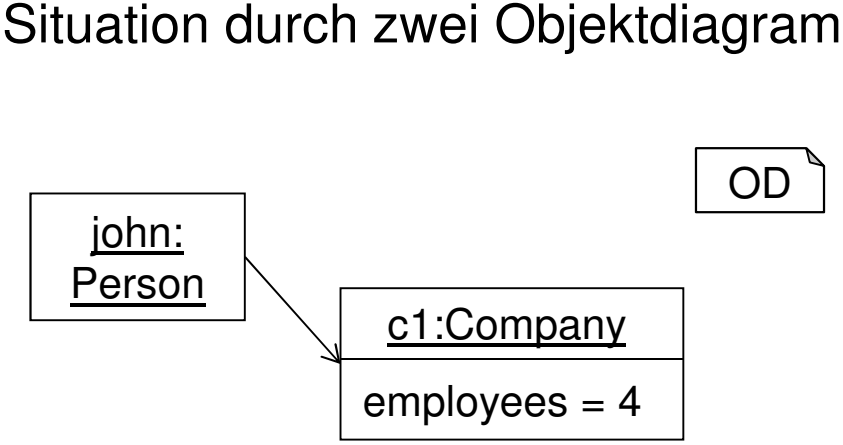
- context Person.changeCompany(String n)
  pre CC2pre: !exists Company co: co.name == n

  post CC2post:
  company.name == n &&
  company.employees == 1 &&
  company@pre.employees == company@pre.employees@pre -1
  && isnew(company)

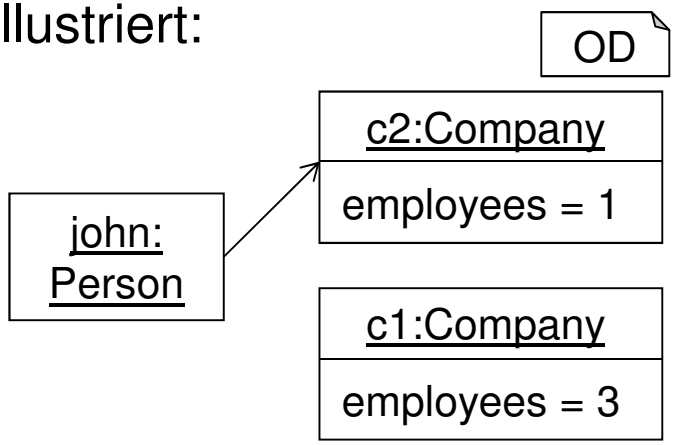
Operator isnew(.) beschreibt, dass ein Objekt erst erzeugt wurde

Im Detail: @pre in Nachbedingungen

- Situation durch zwei Objektdiagramme illustriert:



Vor Methodenausführung



Nach Methodenausführung

- John wechselt von c1 zur neu geschaffenen c2. Wie evaluieren folgende Ausdrücke in der Nachbedingung:
  - john.company.employees ==
  - john@pre.company.employees ==
  - john.company@pre.employees ==
  - john.company@pre.employees@pre ==
  - john.company.employees@pre ==

Im Detail: @pre in Nachbedingungen

- john.company.employees == 1
  - vollständig im Zustand nach dem Methodenaufruf evaluiert
- john@pre.company.employees == 1
  - Referenz auf das Objekt john ändert sich nicht: john==john@pre
- john.company@pre.employees == 3
  - company@pre greift auf den ursprünglichen Zustand des Objekts john zurück und evaluiert zu c1
  - Zugriff über employees erreicht den aktuellen Zustand von c1
- john.company@pre.employees@pre == 4
  - greift auf ursprüngliches Objekt c1 im ursprünglichen Zustand zu

Methodenspezifikation als Kontrakt

- Eingeführt von B. Meyer in der Programmiersprache Eiffel:
  - Kontrakt (Vertrag zwischen Aufrufer und Umgebung)
- „Wenn der Aufrufer Vorbedingung erfüllt, dann erfüllt die aufgerufene Methode die Nachbedingung“
  - plakativ: „Pre implies Post“

- Konsequenzen:
  - Vorbedingung ist eine Verpflichtung an die aufrufende Umgebung.
  - Nachbedingung eine Verpflichtung der aufgerufenen Methode.

- Einsatzformen:
  - Kontrakte erlaubt methodenlokale + kompositionale Verifikation
  - Kontrakte können für Tests eingesetzt werden
  - Kontrakte werden vererbt

Vorbedingung ist nicht erfüllt?

- Eine nicht erfüllte Vorbedingung erlaubt mehrere Interpretationen:
  - a) Programm sollte Fehler erkennen und abbrechen.
  - b) Programm sollte Fehler im Log vermerken und möglichst robust weiter machen.
  - c) Spezifikationssicht: Es ist nichts ausgesagt. Insbesondere muss die Nachbedingung nicht eingehalten werden.

- c) Ist aus Spezifikationssicht ideal: Sie erlaubt Komposition von Teilspezifikationen:
  - Beispiel: changeCompany
  - (CC1pre und CC2pre sind hier disjunkt.)
  - Gilt eine der beiden Vorbedingungen, so soll die jeweilige Nachbedingung gelten.
  - Würden beide gelten, so auch beide Nachbedingungen.

Komposition von Methodenspezifikationen

- context methode() und context methode()
  pre: Apre pre: Bpre
  post: Apost post: Bpost

OCL

- Komposition beider Bedingungen in der Form:
  - context methode()
    pre: Apre || Bpre
    post: (Apre‘ implies Apost) &&
    (Bpre‘ implies Bpost)

OCL

- ggf. sind Variablen anzupassen: Attribute in Apre‘ in der Nachbedingung sind mit @pre zu versehen.
- Die Komposition ist kommutativ und assoziativ und eignet sich für iterative Ergänzung der Spezifikationen.
- Bei expliziter Berechnung sind Vereinfachungen oft möglich

Vererbung von Methodenspezifikationen

- Die Methodenspezifikation einer Superklasse vererbt sich auf die Subklasse und kann dort spezialisiert werden:

  - context Sup.methode() und context Sub.methode()
    pre: Apre pre: Bpre
    post: Apost post: Bpost

    OCL

- Für die Superklasse Sup gilt die linke Spezifikation.
- Für die Subklasse Sub gilt die Komposition beider Spezifikationen.

Unvollständige Charakterisierungen

- Im Allgemeinen ist eine Methodenspezifikation unvollständig.

- Sie konzentriert sich auf die wesentlichen Elemente und überlässt es den Programmierern weitere Details zu klären
  - Prinzip des wohlwollenden Programmierers

- Ein böswilliger Programmierer könnte
  - unwillkürlich andere Objekte oder Attribute ändern,
  - weitere Objekte erzeugen.

- Für genauere Einschränkungen gibt es sog. „Frame-Regeln“:
  - nur die explizit erwähnten Attribute und Objekte dürfen modifiziert werden
  - Implizite Annahme: Der Rest bleibt unverändert, bzw. wird nur angepasst, wenn explizite Invarianten dazu zwingen.

Codegenerierung aus der OCL

- Viele Konstrukte der OCL sind systematisch implementierbar:
  - Nutzung der try-catch-Struktur zur Behandlung von Undefiniertheit
  - Set/Map/List lassen sich auf Java abbilden
  - Quantoren können durch Iteratoren realisiert werden
- Invarianten lassen sich ausführen und so in Tests einsetzen
  - Explizit festlegen, welche Invariante wo gilt:
    Ähnlich zu asserts in Java.
  - Effizienzüberlegungen:
    Invariante nur inkrementell testen, z.B. bei Objektänderung
  - Infrastruktur notwendig, um Objektänderungen zu beobachten
- Vorbedingungen sind wie Invarianten ausführbar
- Allerdings: Nachbedingungen benötigen Attributwerte aus der Startzeit!

Prüfender Code aus Nachbedingungen

- Nachbedingungen benötigen Attributwerte aus der Startzeit!
  - context Person.incAge()
    pre: true
    post: age == age@pre +1

OCL

- Die benötigten Startwerte sind aufzuheben.
- Eine Form mit lokalen Variablen statt @pre-Operator:
  - context Person.incAge()
    let ageOld = age
    pre: true
    post: age == ageOld +1

OCL

Die eingeführten Elemente des let-Operators können für beide Bedingungen genutzt werden

- let speichert hier alte Werte!
- Problem: ggf. muss ein ganzer Container doppelt verwaltet werden: Effizienzüberlegungen sind notwendig.

Konstruktiver Code aus Nachbedingungen

- Beispiel:
  - context Person.incAge()
    pre: true
    post: age == age@pre +1

OCL

- Aus vielen Nachbedingungen kann konstruktiv Code erzeugt werden:
  - class Person {
    void incAge() {
    assert true; // Vorbedingung
    age = age+1; // Nachbedingung
    } }

Java

- Jedoch nicht immer ist dies eindeutig, oder automatisiert lösbar:
  - context changeSomething()
    pre: true
    post: c*d==a*b

OCL

- Wie sind nun a, b, c oder d zu ändern? (Alles 0 setzen?)
- Prolog-artige Techniken helfen bei der Automatisierung.

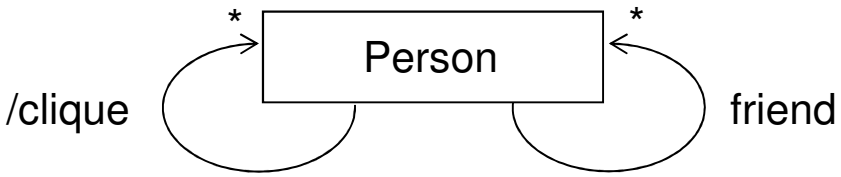
Zusammenfassung 3

- OCL ist eine textuelle Beschreibungssprache für Invarianten und Vor-/Nachbedingungen.
- OCL-Bedingungen gelten im Kontext von Klassen, etc.
- OCL besitzt keine eigenen Methodendefinitionen, sondern nutzt die des zugrunde liegenden Objektsystems.
- OCL bietet Mechanismen der Logik erster Stufe (FOL).
- Quantoren werden aber auf endlichen Objektmengen interpretiert und sind daher ausführbar.
- OCL erlaubt die kompakte Spezifikation von Invarianten, die oft graphisch nicht ausdrückbar sind.

Anhang: Transitive Hülle 1

- OCL ist eine First-Order-Logik.
- FOLs sind nicht in der Lage, Konzepte wie Termerzeugung oder das Induktionsprinzip der natürlichen Zahlen zu beschreiben.
- Die besonders häufig gebrauchte transitive Hülle über eine rekursive Assoziation ist in OCL (weil FOL!) nicht beschreibbar.
- Beispiel:

CD



- Es soll beschrieben werden, dass die abgeleitete Assoziation clique die transitive Hülle von friend darstellt:

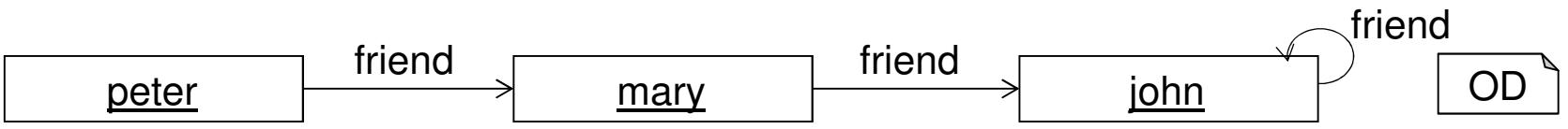
OCL

  - context Person inv TransitiveHuelle:
    clique == friend.addAll(friend.clique)

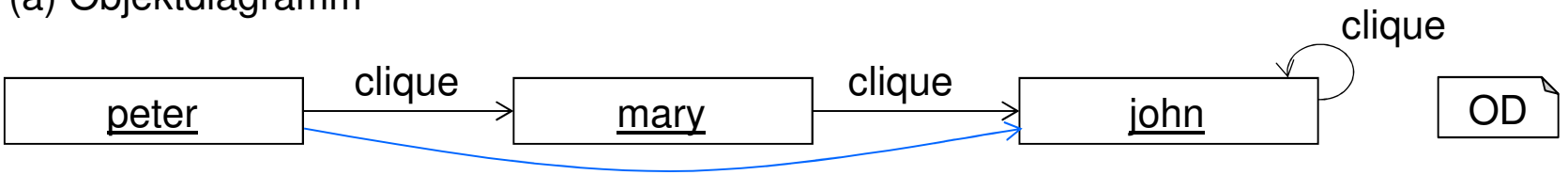
Anhang: Transitive Hülle 2

- context Person inv TransitiveHuelle:
  clique == friend.addAll(friend.clique)

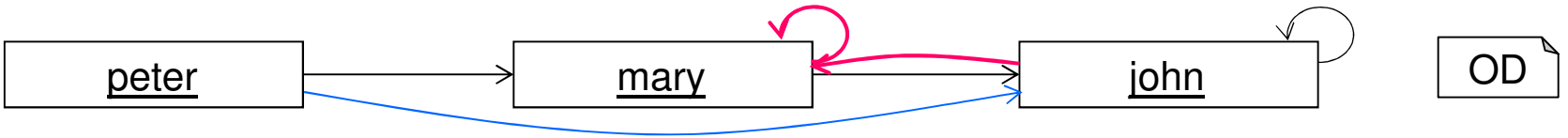
- beschreibt, dass clique transitiv ist und, dass sie friend enthält.
- dies ist nicht eindeutig, die transitive Hülle ist nur die kleinste:



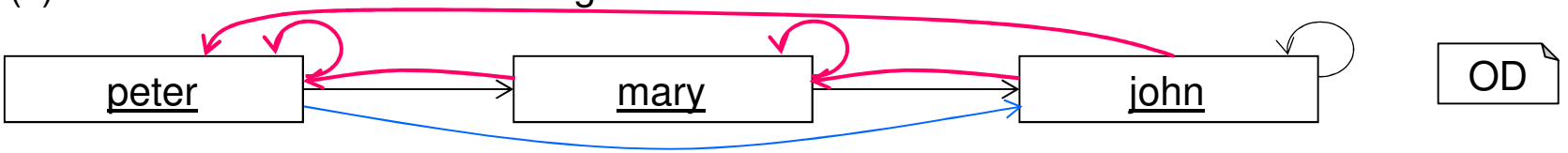
(a) Objektdiagramm



(b) die gewünschte transitive Hülle



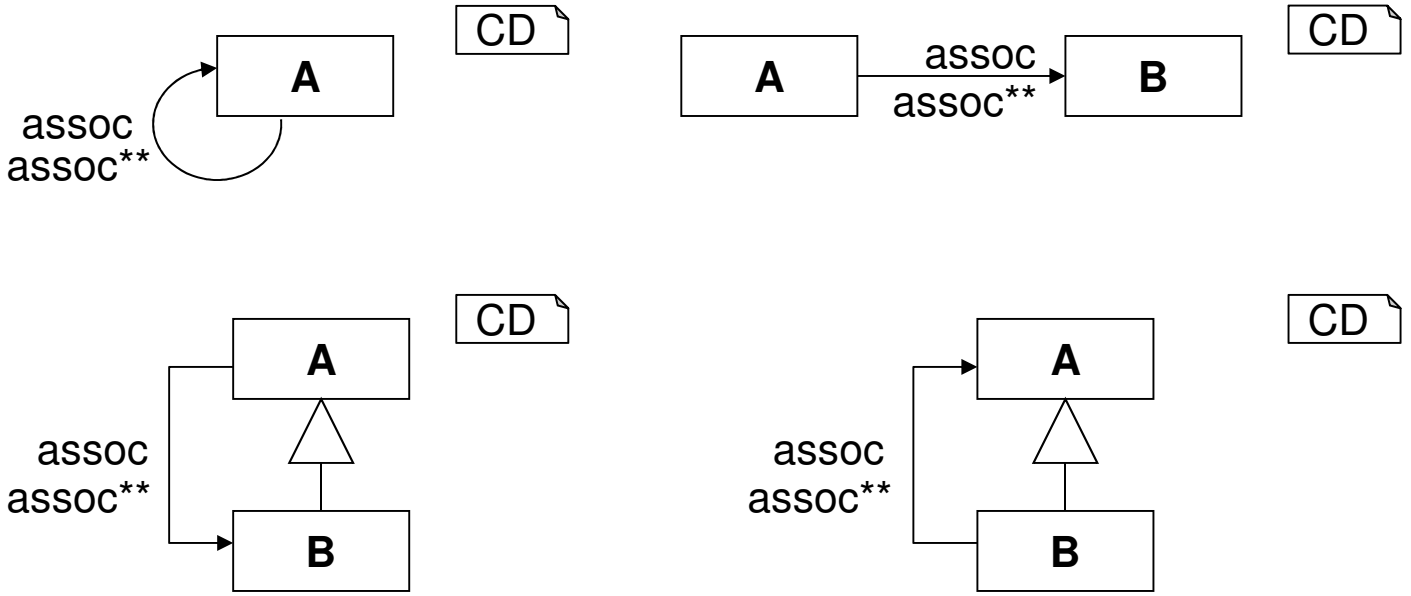
(c) eine weitere transitive Lösung



(d) die “maximale” Lösung

Anhang: Transitive Hülle 3

- Lösung: Nutzung eines expliziten Operators **, der die transitive Hülle einer Assoziation bildet:
  - context Person inv TransitiveHuelle:
    clique == friend**
- Typisierung der transitiven Hülle ist wie die der zugrunde liegenden Assoziation. Varianten:

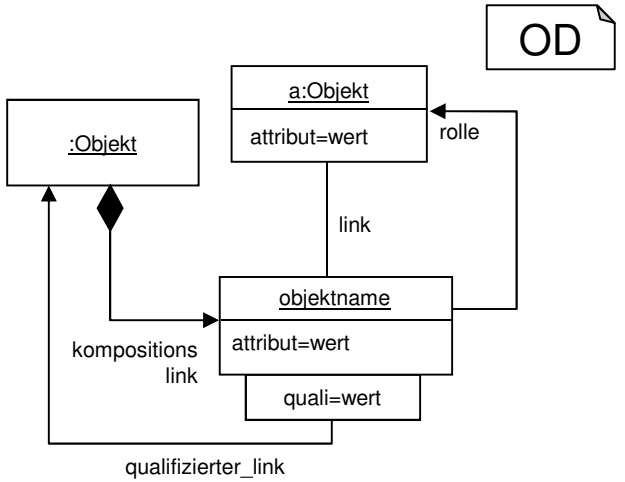


SE SOFTWARE ENGINEERING

RWTH AACHEN

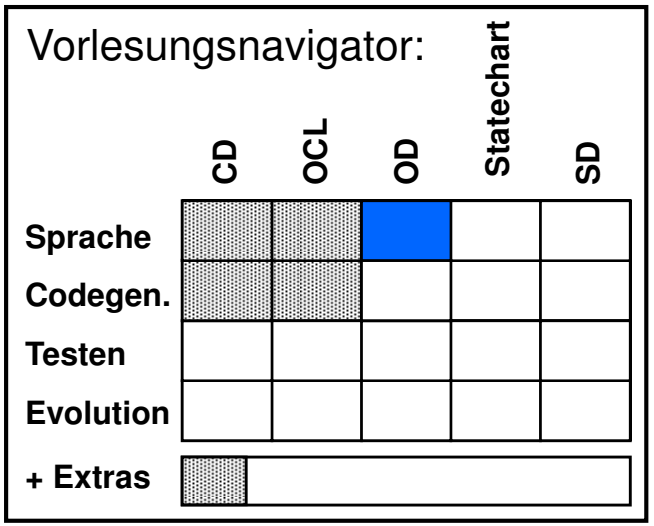
Modellbasierte Softwareentwicklung

- 4. Objektdiagramme
- 4.1. Sprache

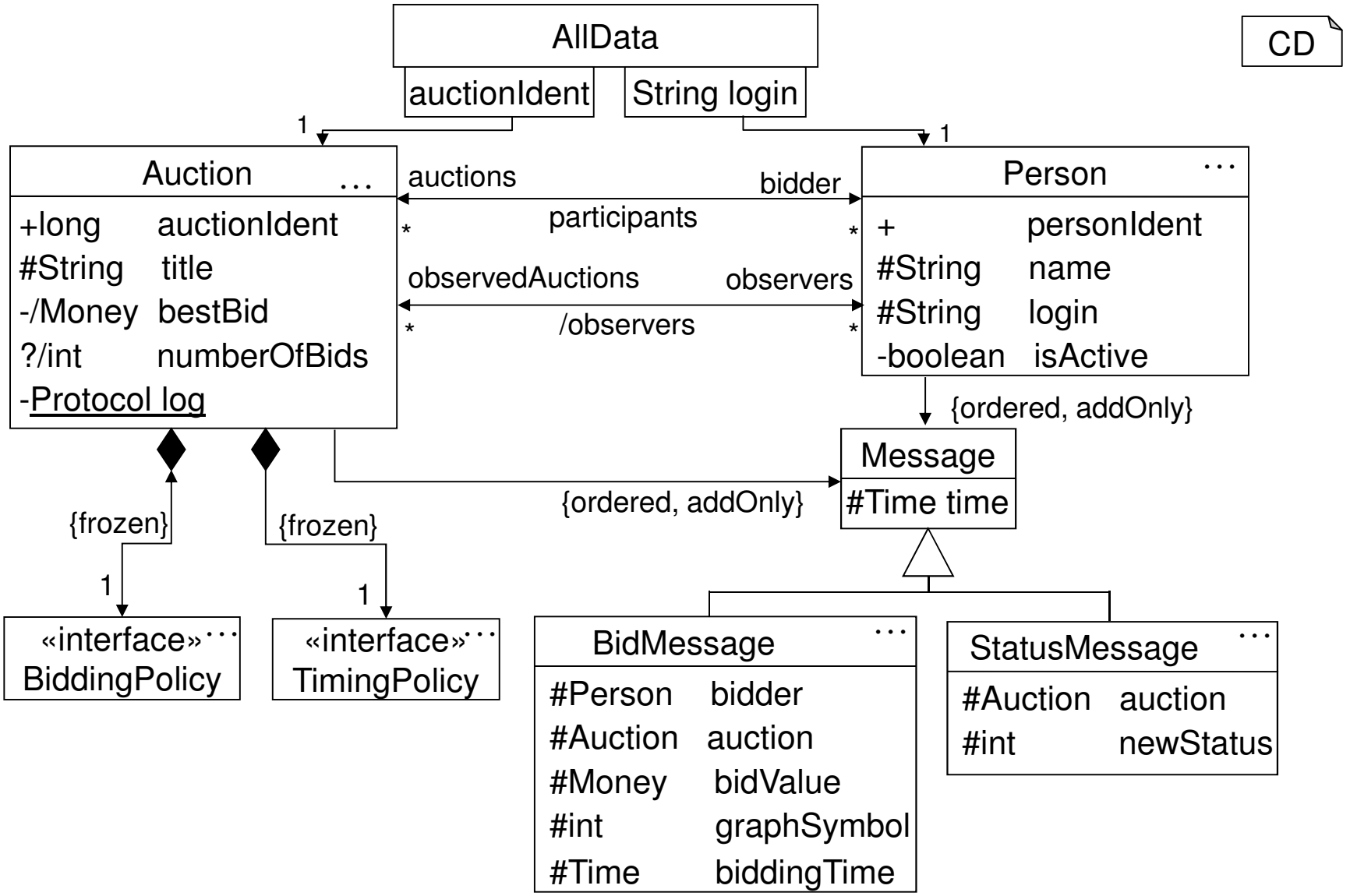


Prof. Dr. Bernhard Rumpe
Lehrstuhl für Software Engineering
RWTH Aachen

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



Zur Erinnerung: Klassendiagramm des Auktionssystems (Ausschnitt)



Objektdiagramm

- Ein Objektdiagramm zeigt eine konkrete Situation in einem Systemablauf:
  - Konkrete, benannte Objekte
  - Konkrete Attributwerte
  - Linkstruktur zwischen den Objekten

- Objektdiagramm zeigt einzelne, mögliche Situation
- vs. Klassendiagramm charakterisiert alle möglichen Situationen.

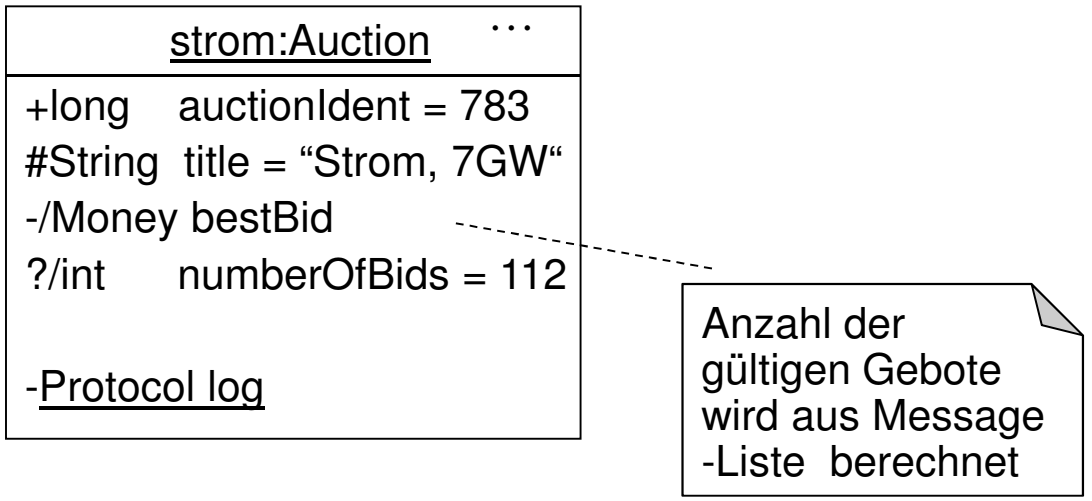
- Die gezeigte Situation eines Objektdiagramms kann gar nicht oder auch mehrfach auftreten.

- Einsatzformen:
  - Start oder Ende-Situation für einen Test
  - Unerwünschte Situation, ...

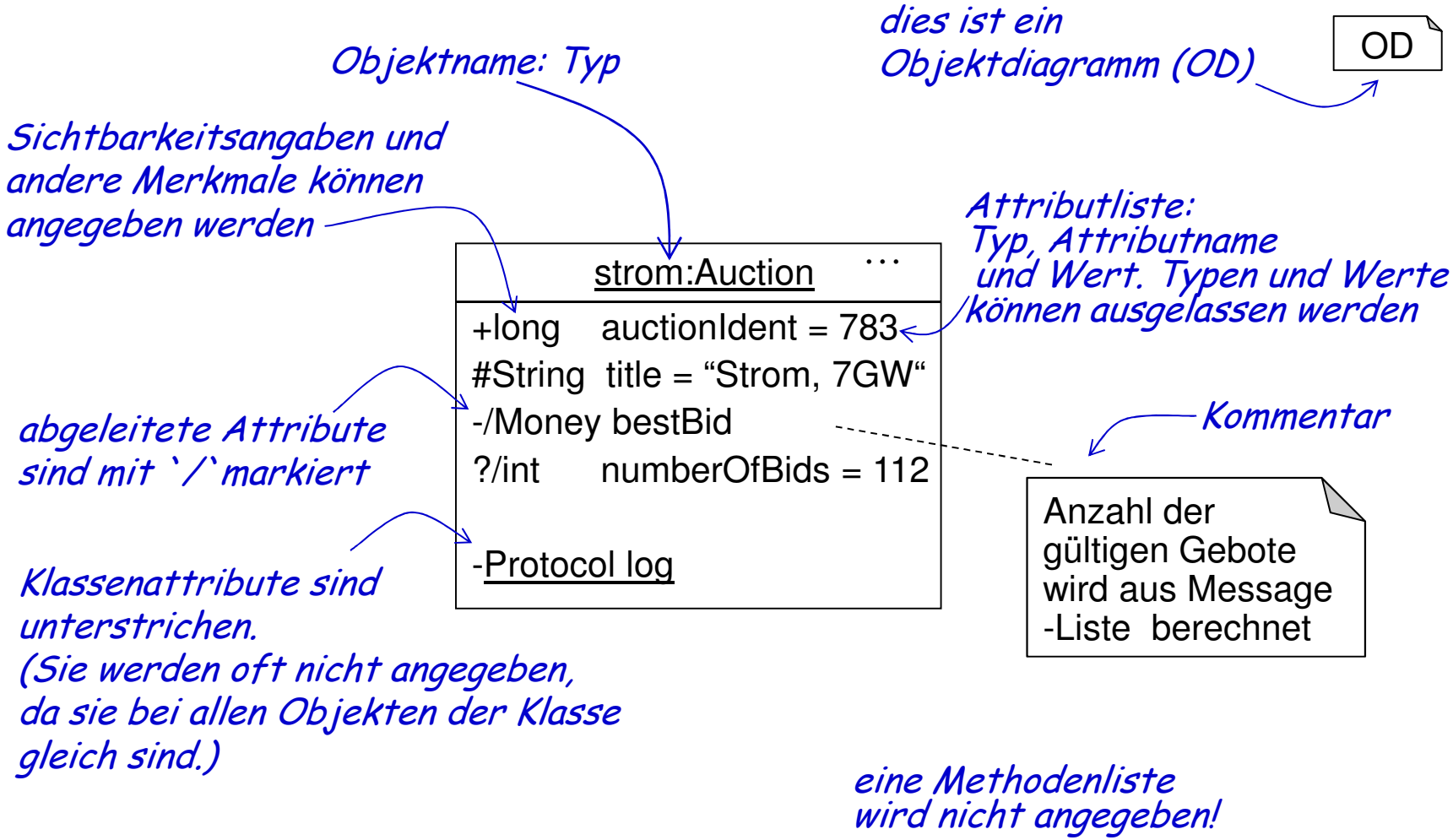
Beispiel: Einzelnes Objekt

discuss

OD



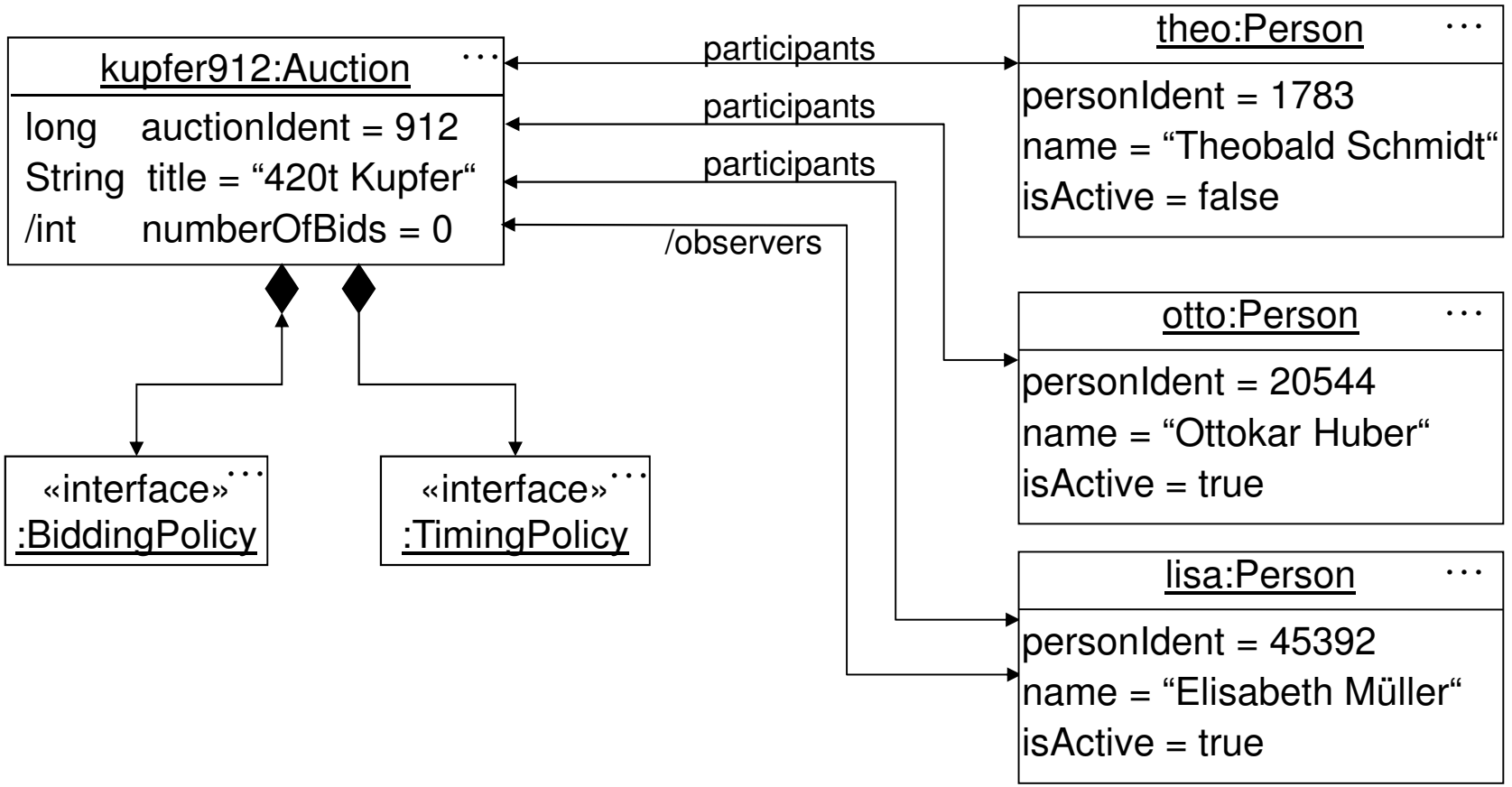
Beispiel: Einzelnes Objekt



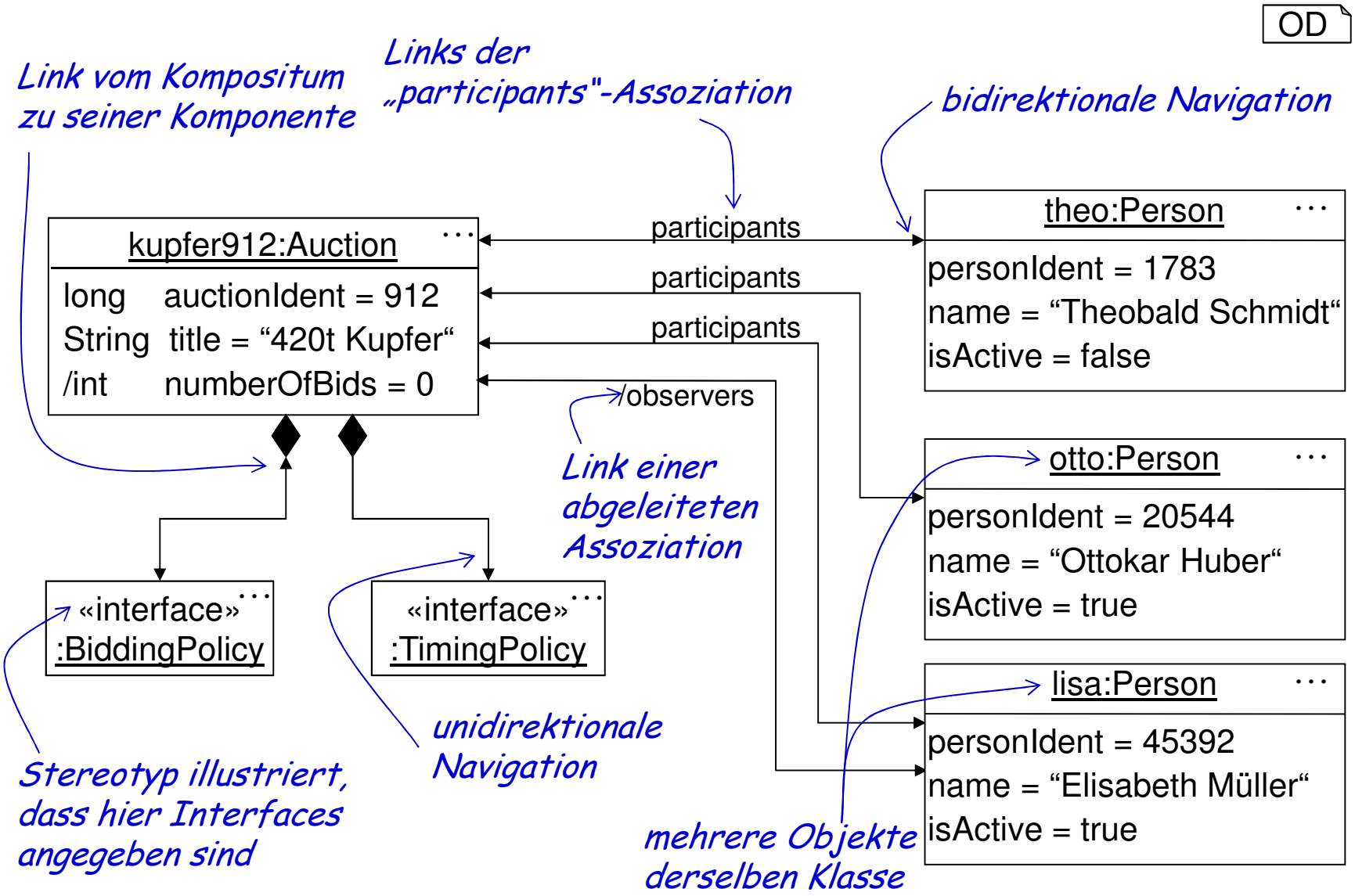
Beispiel: Linkstruktur

discuss

OD



Beispiel: Linkstruktur



Begriffsbestimmung

- Objekt
  - Objekt ist Instanz einer Klasse
  - Attribute haben einen Wert (oder sind ggf. uninitialisiert)
  - Objektdiagramm nutzt prototypische Objekte
  - Zwischen prototypischen Objekten und den echten Objekten des Systems besteht keine 1:1-Beziehung
- Objektname erlaubt die eindeutige Benennung des Objekts im Objektdiagramm
- Attribut beschreibt Zustandskomponente eines Objekts:
  - Immer: Attributname
  - Optional: Typ, Wert und Sichtbarkeit
- Abstrakte Objektdiagramme haben Variablennamen oder Ausdrücke statt konkreten Werten
- Link ist eine Ausprägung einer Assoziation zwischen zwei Objekten
  - Optional: Navigationsrichtung, Assoziations- und Rollennamen

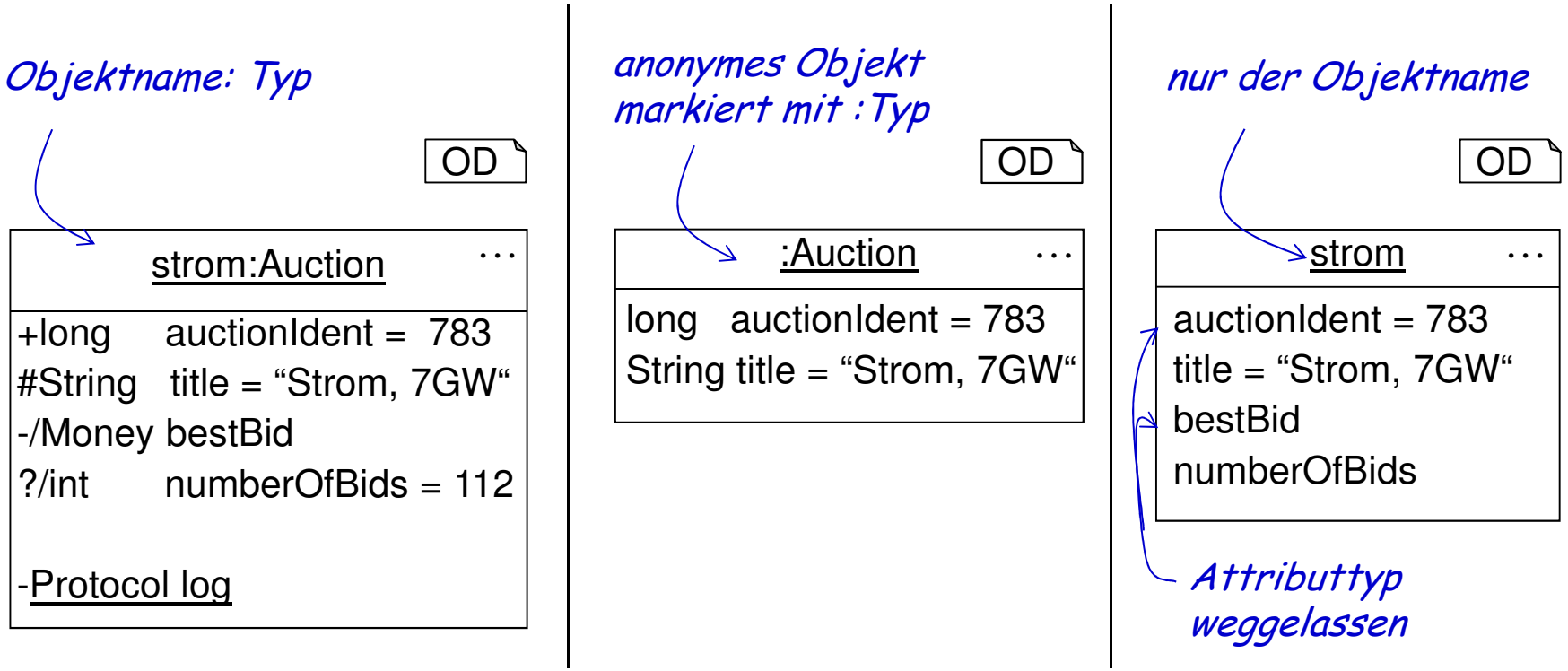
Aufgabe:

home

- Entwerfen sie ein OD, das folgendes charakterisiert (mit den wesentlichsten Beziehungen zwischen den beteiligten Elementen):
  - Ihre Familie mit Wohnorten
  - ein Flugzeug und seine technischen Geräte
  - eine (mehrteilige) Flugverbindung für den Gast „Wolfgang“ am 2.4.2004
- Ziel: Umgang mit OD (nicht inhaltlich perfekt, sondern syntaktisch korrekt)

Darstellung eines Objekts

- ... ist auf viele Weisen möglich:

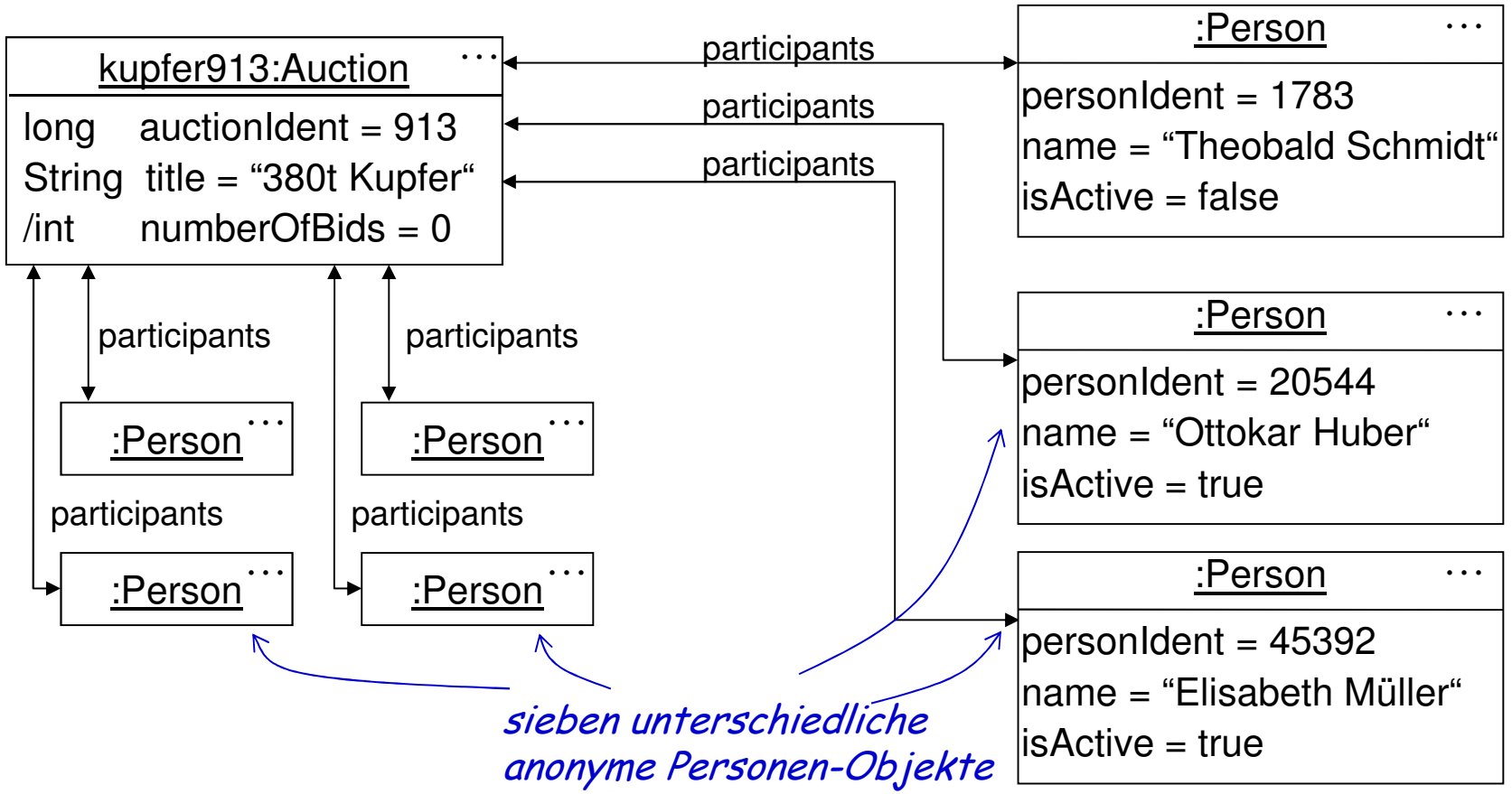


- Repräsentationsindikatoren „...“ und „©” zeigen ggf. Vollständigkeit der Attributlisten an.

Anonyme Objekte

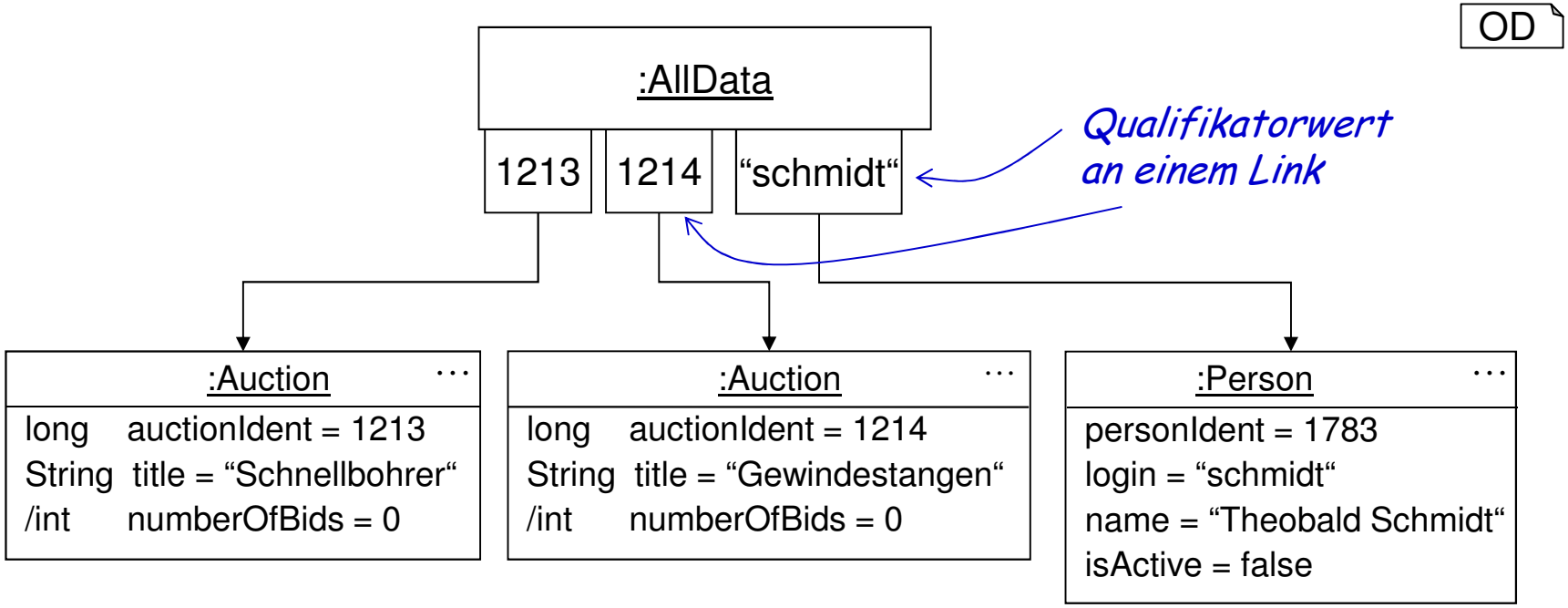
- Ein anonymes Objekt hat keinen expliziten Namen.
- Anonyme Objekte sind dennoch unterschiedlich:

OD



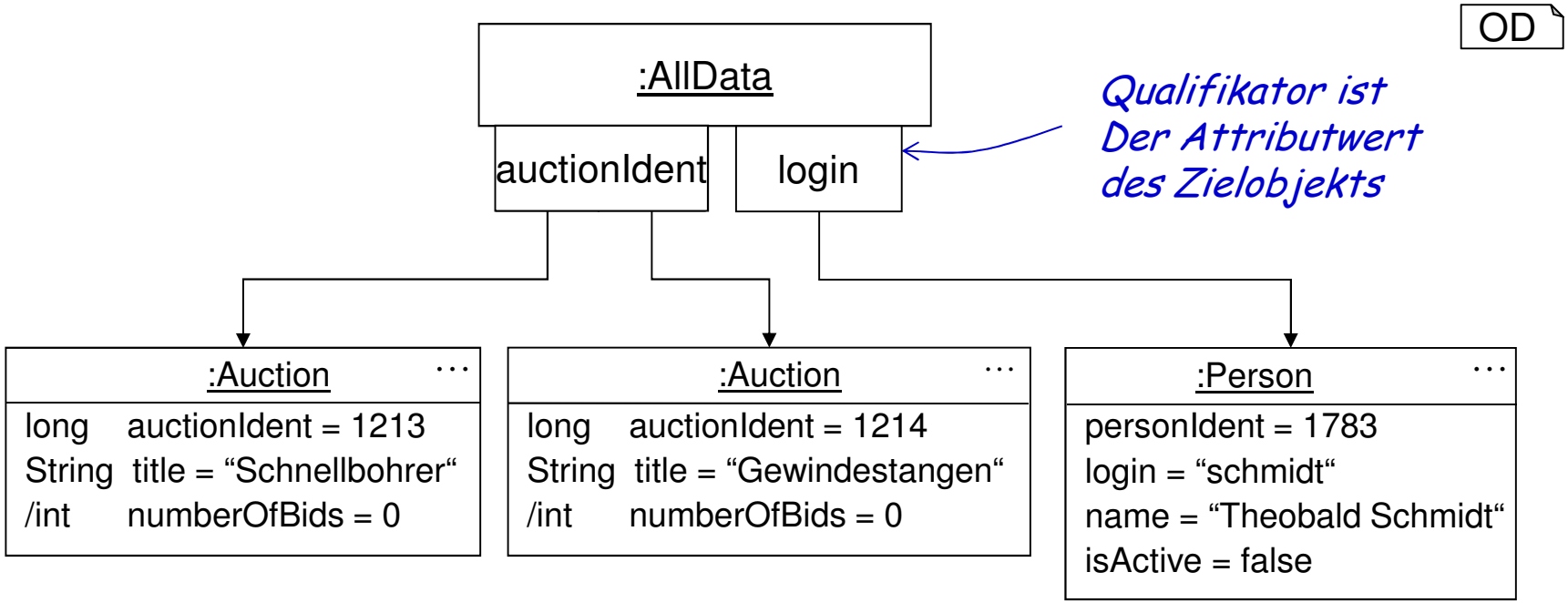
Qualifizierte Links

- ... gehören zu einer qualifizierten Assoziation.
- Sie enthalten normalerweise den konkreten Wert.
- Wenn Qualifikator bereits eindeutig die Assoziation festlegt, kann auf Assoziationsnamen verzichtet werden:



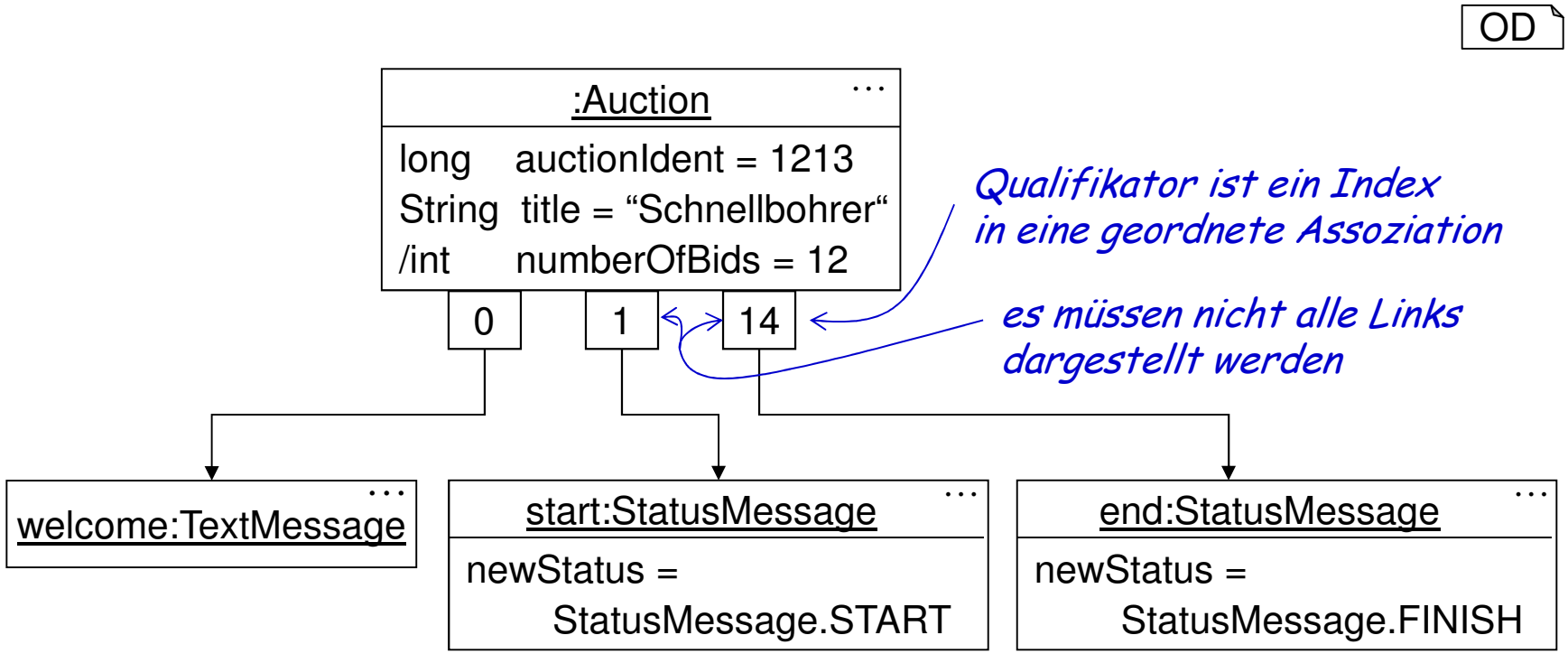
Qualifizierte Links ohne Wert

- Spezialfall: Statt konkretem Wert wird ein Attribut des Zielobjekts angegeben:



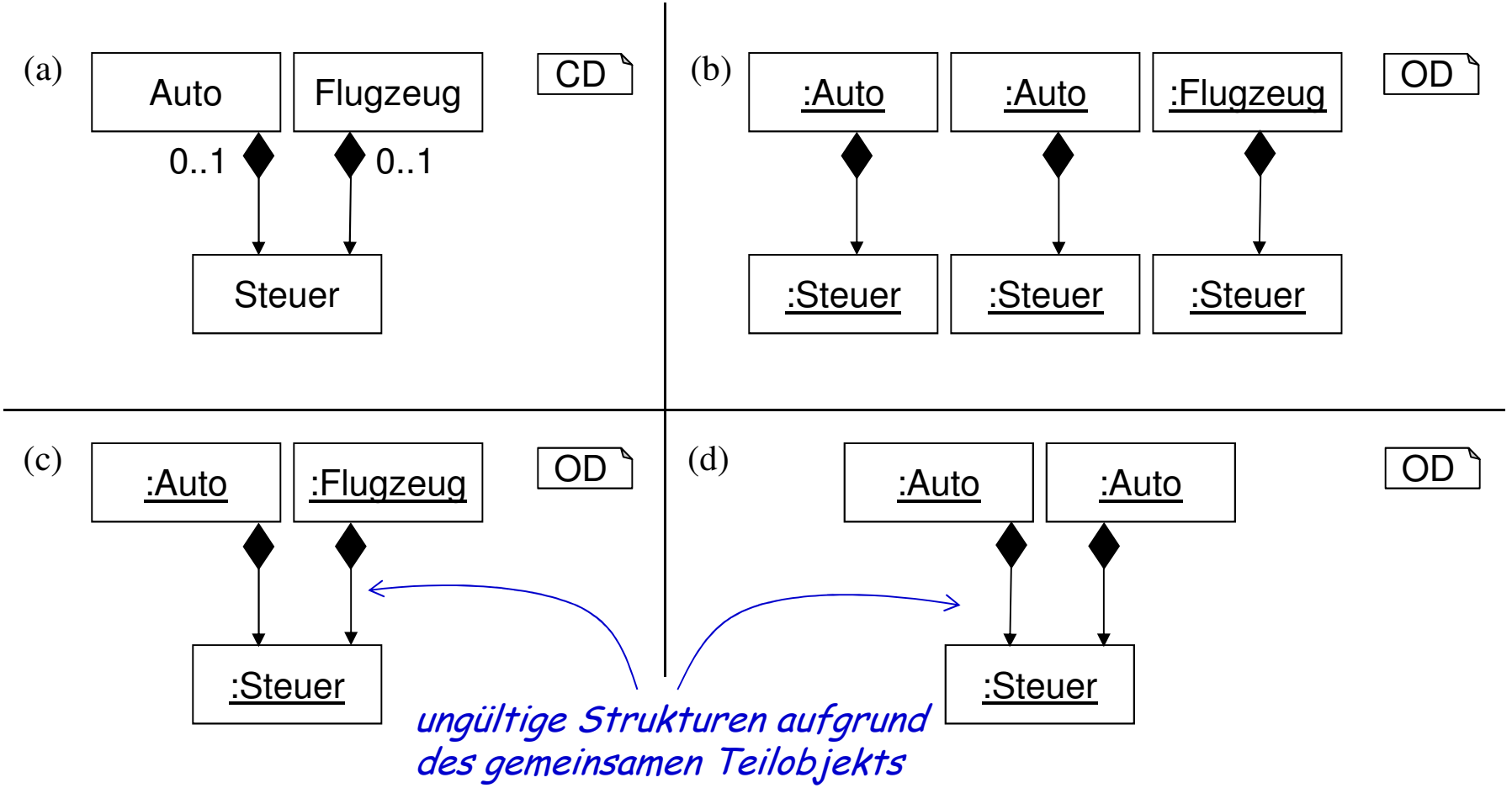
Links einer geordneten Assoziation

- ... enthalten ganze Zahlen als Qualifikator.
- Die Liste muss nicht vollständig sein.



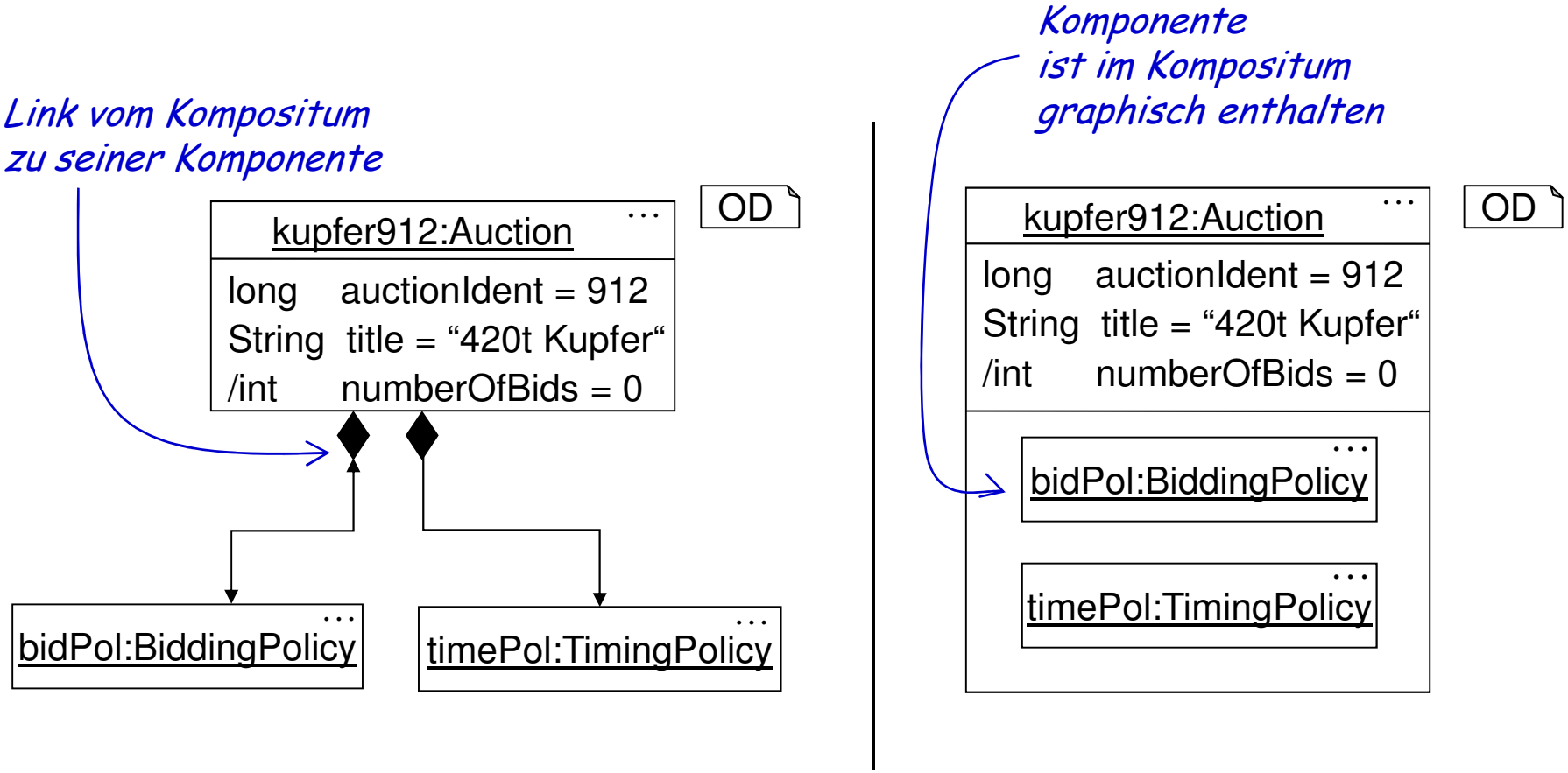
Komposition im Objektdiagramm

- Klassendiagramm (a) und Objektdiagramm (b) sind erlaubt
- (c) und (d) enthalten unzulässige Kompositionsstrukturen



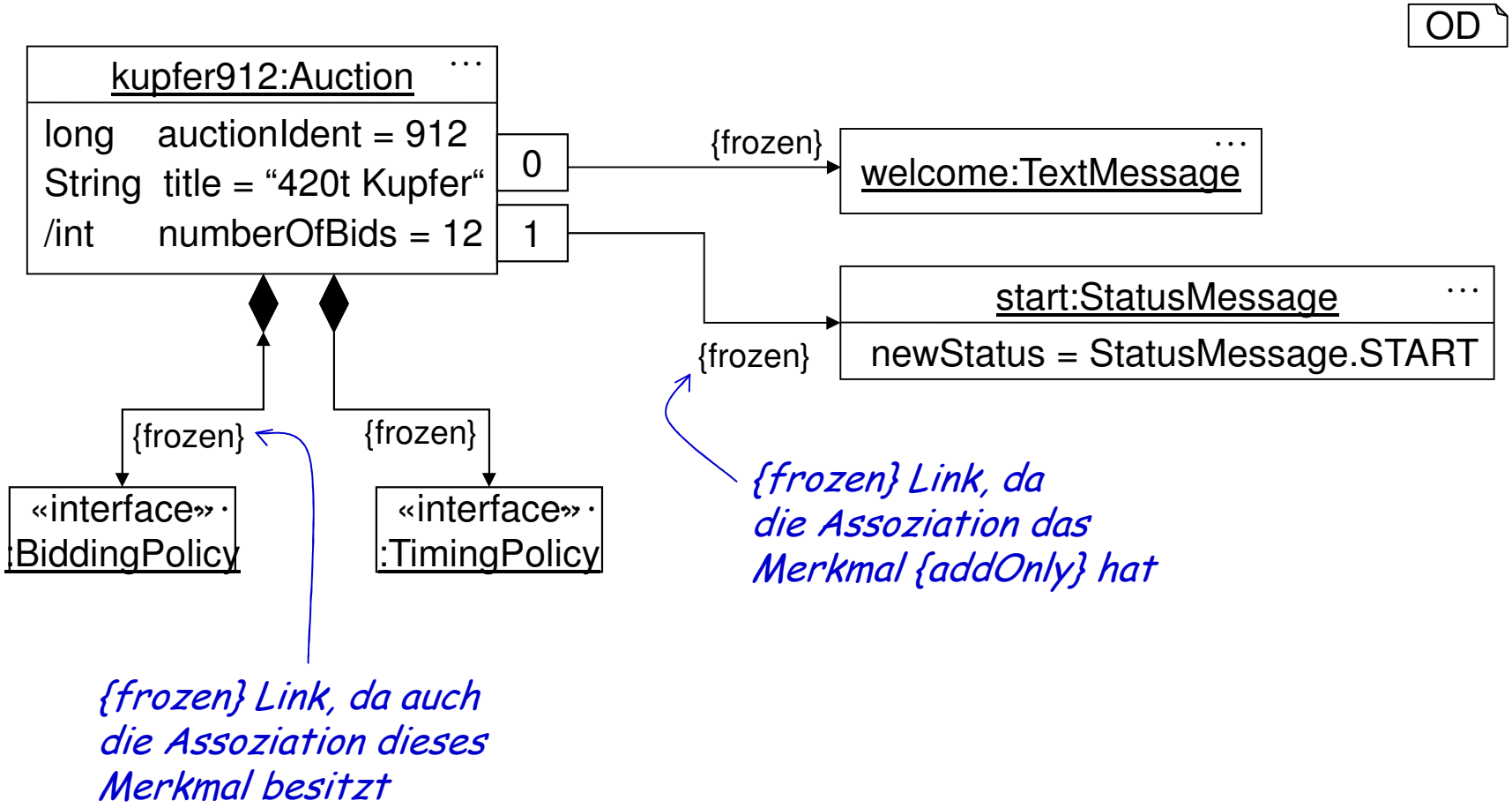
Alternative Darstellung der Komposition

- ... durch graphisches Enthaltensein.
- Schachtelung ist möglich.
- Beide Diagramme sind (bis auf Navigationsinformation) äquivalent:



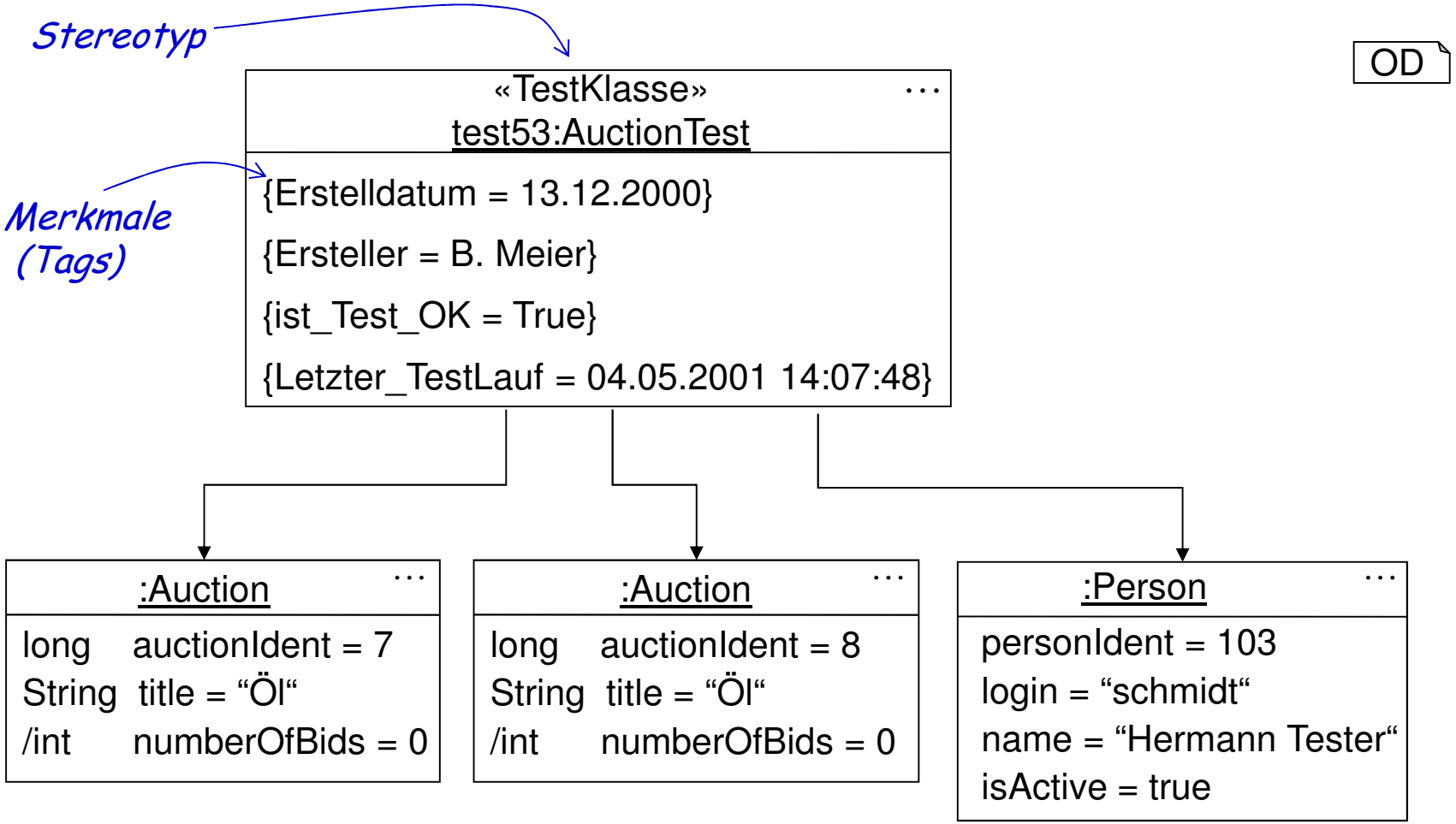
Merkmale in Objektdiagrammen

- Zum Beispiel {frozen} für Links:



Weitere Anwendung von Merkmalen

- Stereotypen wie im Klassendiagramm
- und detaillierende Information in Form von Merkmalen mit Werten:

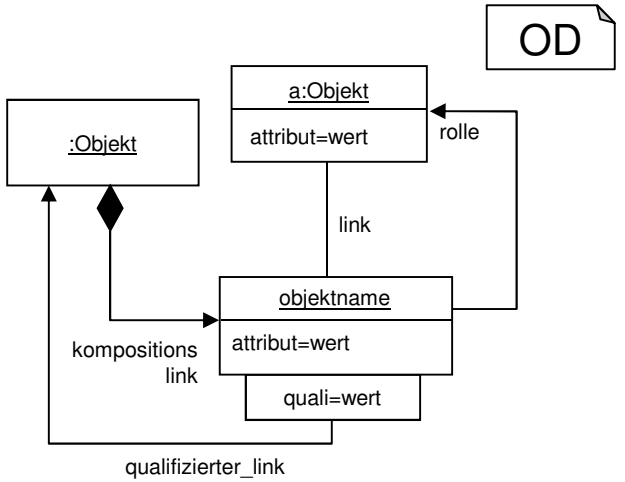


SE SOFTWARE ENGINEERING

RWTH AACHEN

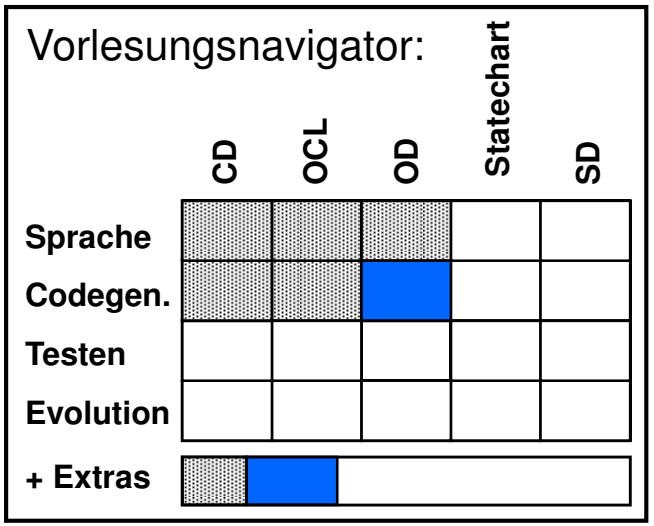
Modellbasierte Softwareentwicklung

- 4. Objektdiagramme
- 4.2. Bedeutung und Einsatz



Prof. Dr. Bernhard Rumpe
Lehrstuhl für Software Engineering
RWTH Aachen

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



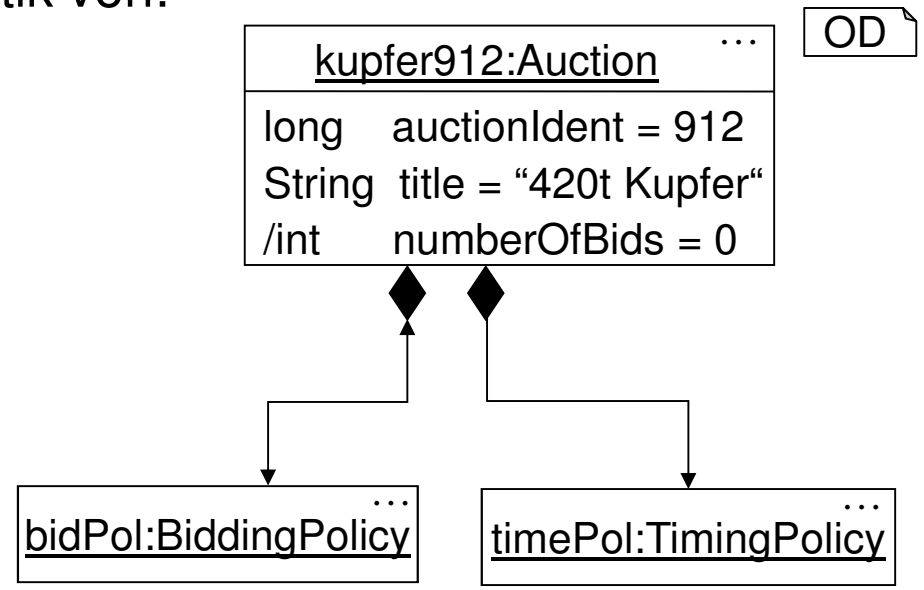
Semantik eines Objektdiagramms?

discuss

- Ein Objektdiagramm ist exemplarisch:
  - Welche Bedeutung hat ein solches Diagramm?

- Wo und wofür lassen sind Objektdiagramme anwenden?

- Was ist die Semantik von:



Semantik eines Objektdiagramms

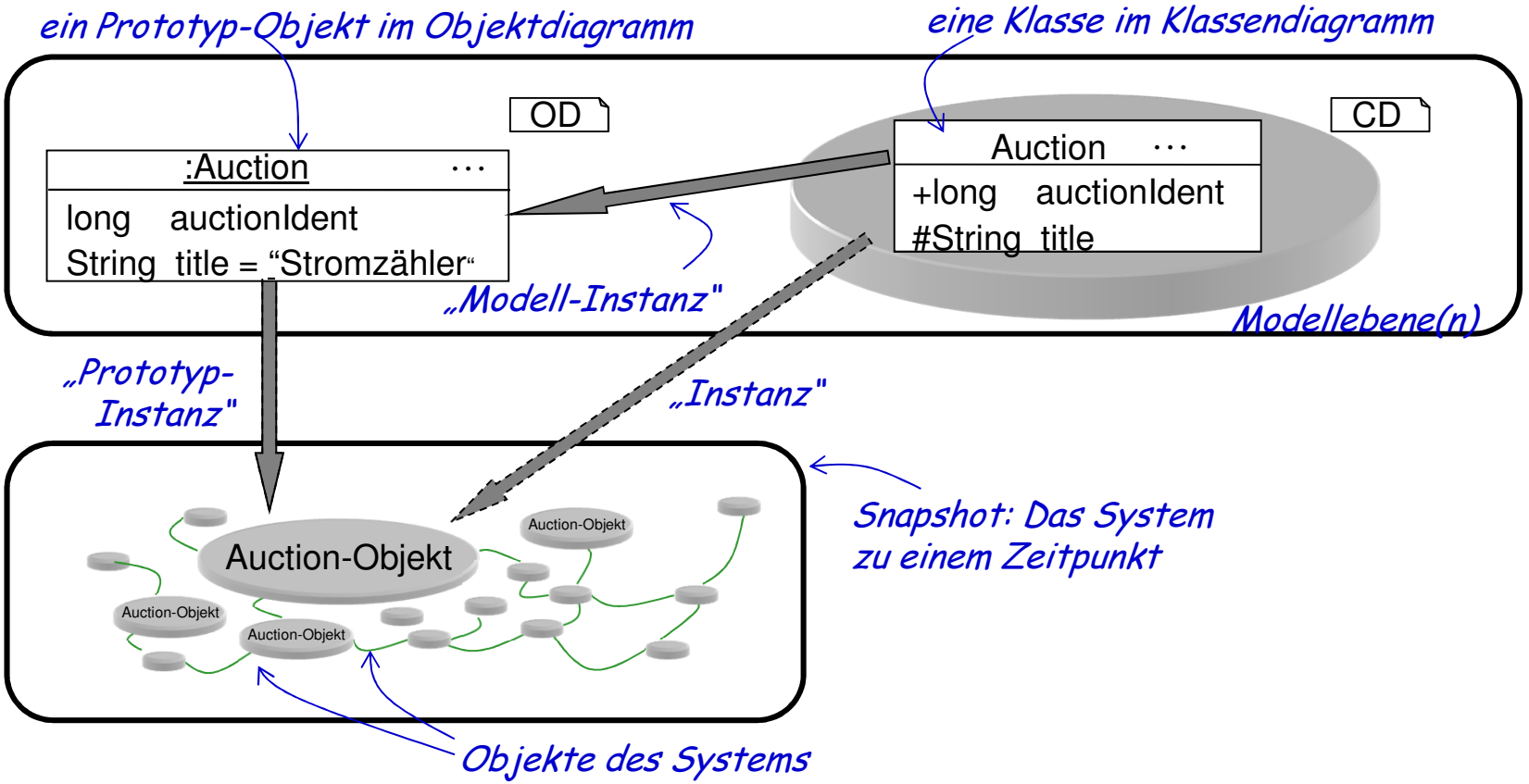
- Exemplarische Natur heißt:
  - Es kann mehr als eine Inkarnation des Diagramms geben
  - Es muss keine geben
  - Anzahl kann über die Zeit und verschiedene Systemläufe variieren

- Prototypische Objekte („Rechtecke“) im Diagramm nicht mit Objekten des Systems verwechseln:
  - Es herrscht keine 1:1-Beziehung.

- Aussagefähigkeit der Objektdiagramme ist sehr beschränkt.
- Es fehlen Möglichkeiten zu sagen
  - „Dieses OD gilt immer genau einmal.“
  - „Dieses OD gilt zu Beginn.“
  - „Dieses OD tritt nie ein.“
  - „Das unbesetzte Attribut x im OD liegt im Bereich [-5,5].“

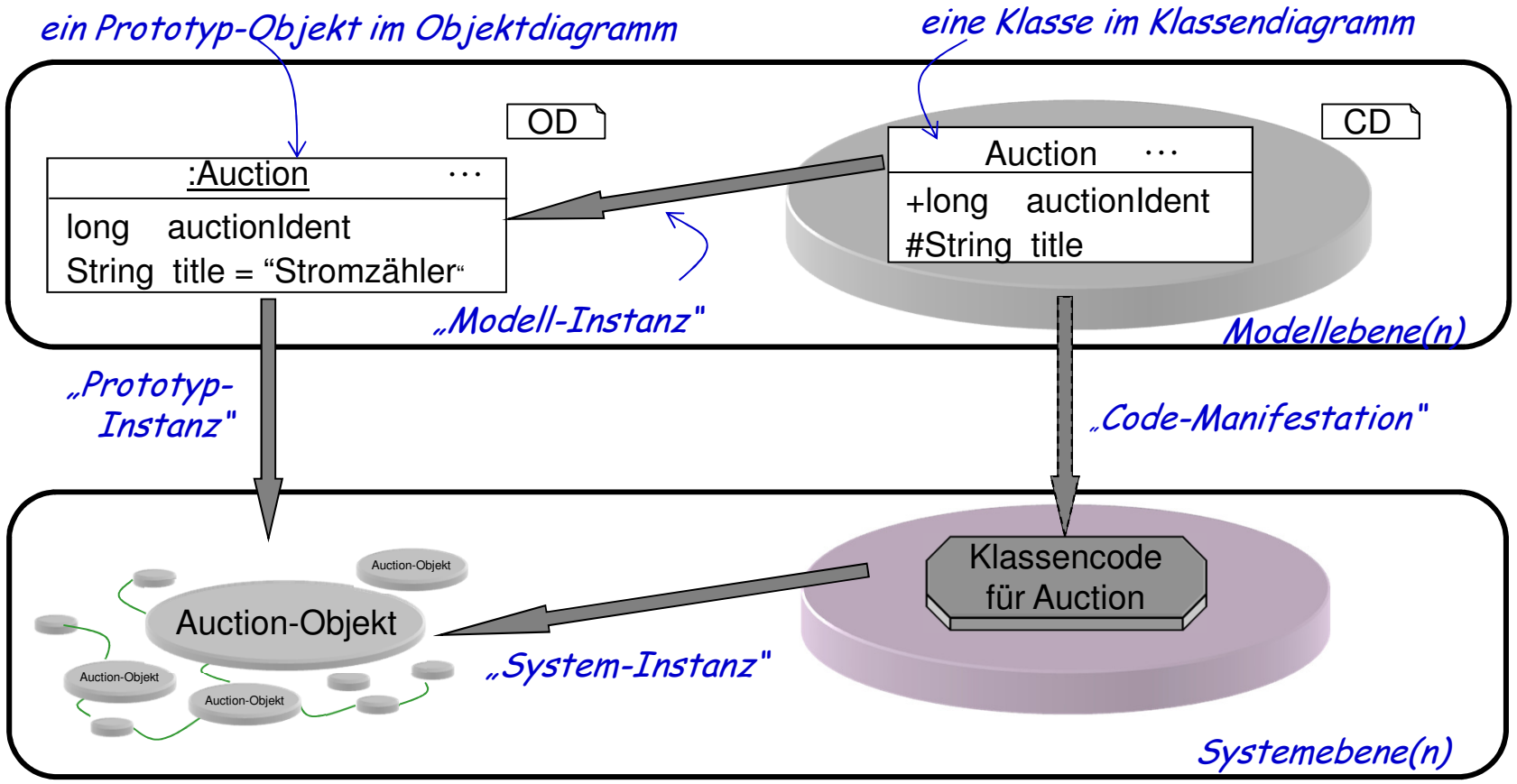
Modell-Instanz vs. Instanz im System

- OD und CD sind Teile der Modellebene: Obwohl zwischen Ihnen eine „Modell-Instanz“-Beziehung existiert.
- System-Objekte haben Beziehungen zu Prototyp-Objekten und Klassen: das sind aber drei unterschiedliche Arten von „Instanzen“



Modell-Instanz vs. Instanz im System: Mit „Klassencode“

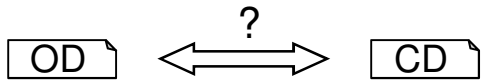
- Berücksichtigung der Existenz von Klassen zur Laufzeit im System führt zu folgender Erweiterung.
- „Klassencode“ ist der compilierte Teil einer Klasse des Modells.



Objekt- und Klassendiagramm

- Es gibt vielfältige Zusammenhänge und damit syntaktische Bedingungen:
  - Für genannte Objekte müssen Klassen existieren,
  - Attribute des OD müssen im CD definiert sein und denselben Typ haben,
  - Links müssen zu korrespondierenden Assoziationen passen,
  - Multiplizitäten sind einzuhalten,
  - etc.
- Repräsentationsindikatoren „...“ und „©” bei beiden Diagrammen erlauben Auslassungen bzw. erfordern Vollständigkeit.

- Spannende Fragen:



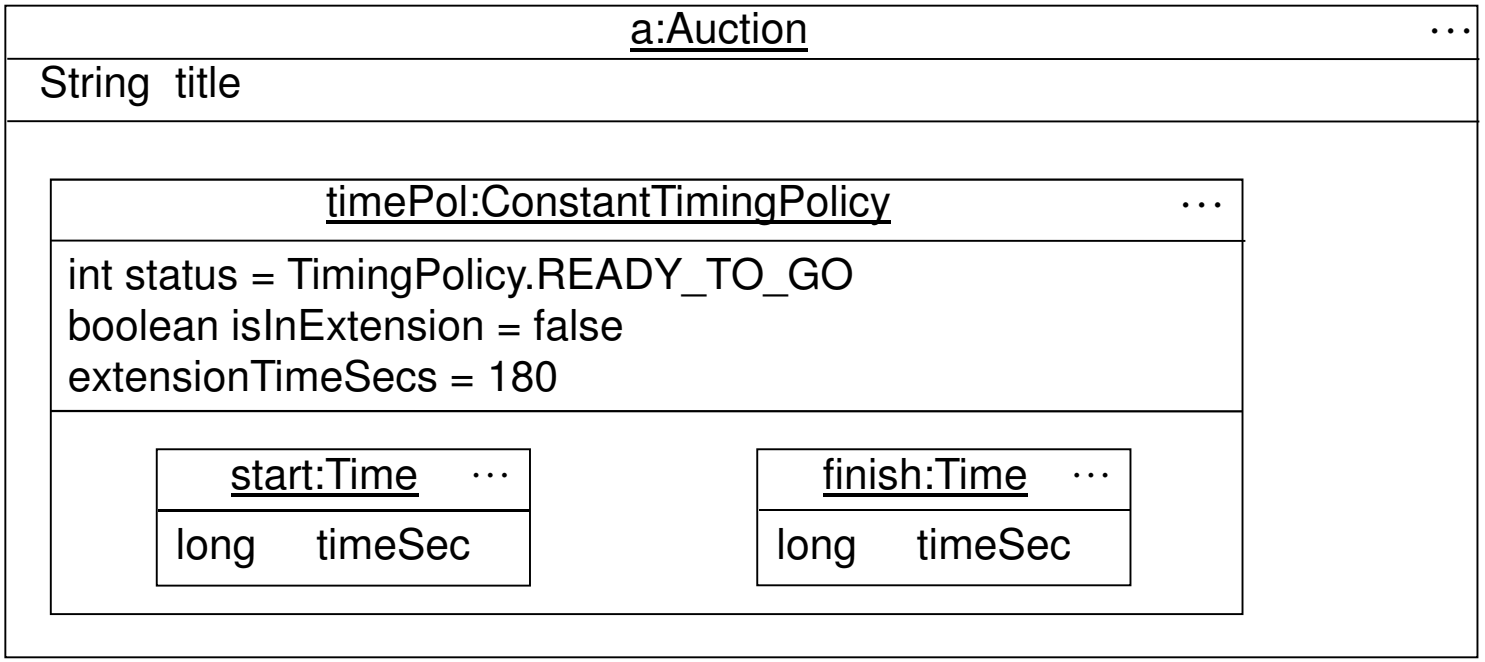
  - Welche Syntaxchecks sind nun exakt sinnvoll?
  - Lässt sich ein CD aus OD‘s ableiten?
  - Lassen sich OD‘s als Testbeispiele aus einem CD ableiten?

Prototypische Objekte

- Objektdiagramme können als Muster verstanden werden.
- Prototypen im OD als Vorbilder, Muster
- Unvollständige Objektbeschreibungen lassen Freiraum
- OCL erlaubt den Freiraum geeignet zu beschränken
- Beispiel:
  - start.timeSec + 2*60*60 <= finish.timeSec;

OCL

OD

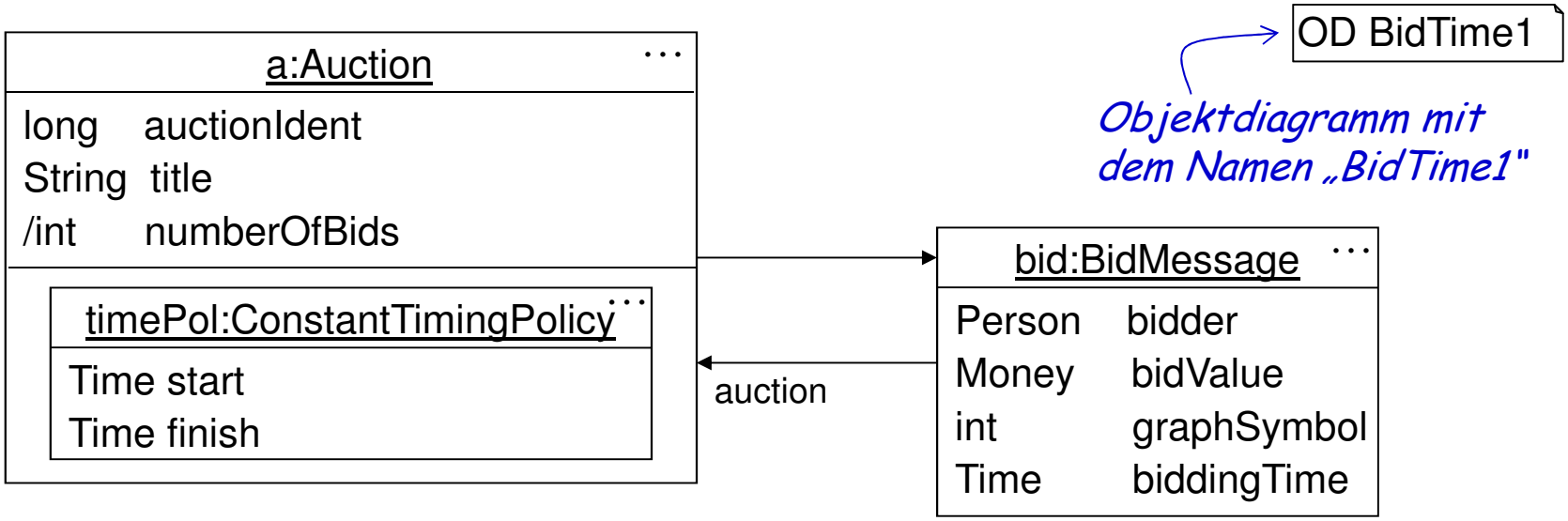


Einsatz von Objektdiagrammen

- Vielfältige Einsatzmöglichkeiten:
  - Exemplarische Situation zur Diskussion mit Kunden/Kollegen
  - Architekturbeschreibung statischer Anteile (Situation gilt immer)
  - Vorbedingung für einen Methodenaufruf
  - Nachbedingung für einen Methodenaufruf
  - Unerwünschte Situation
  - Ausgangssituation für einen Test
  - Sollsituation für einen Test

- Einsatzmöglichkeiten lassen sich beschreiben, indem Objektdiagramme mit OCL kombiniert werden:
  - Das Objektdiagramm als speziell notierte Aussage

OD als Aussage: Nutzung durch Namen



- Benannte Objektdiagramme werden als Aussage in die OCL einbezogen:
  - context Auction a, ConstantTimingPolicy timePol, BidMessage bid inv BidTime1A:

    OD.BidTime1 implies
    timePol.start.lessThan(bid.biddingTime) &&
    bid.biddingTime.lessThan(timePol.finish)

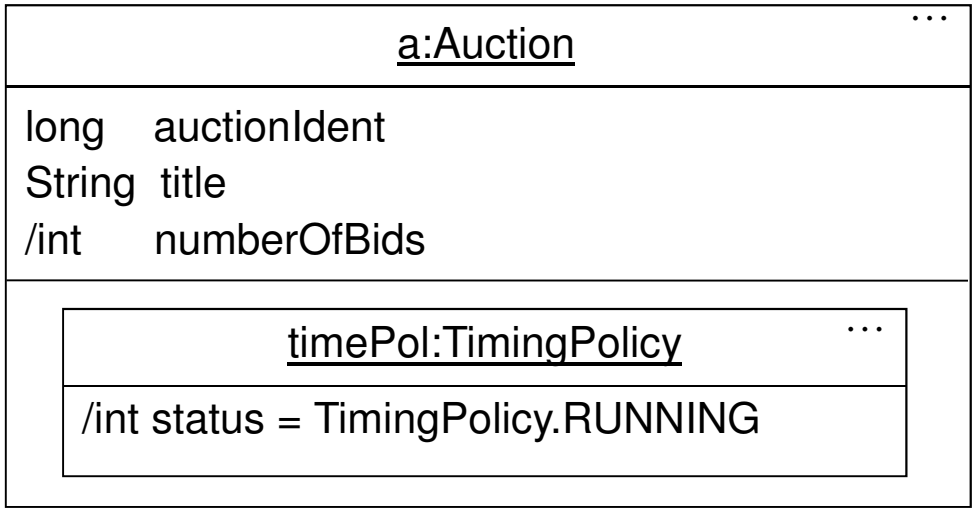
OCL

- Bindung der freien Namen des OD (a, bid, timePol) erfolgt in OCL

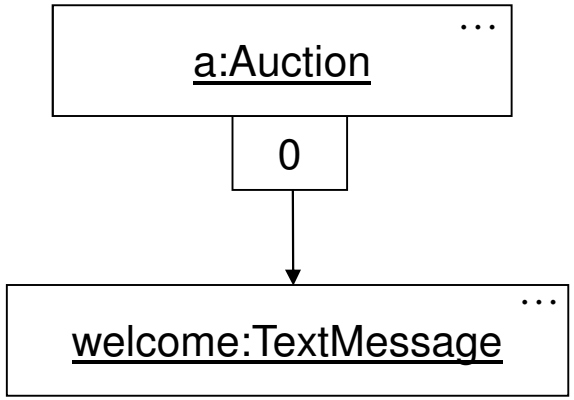
Logische Aussagen: Beispiel Implikation

- Wenn Situation 1A zutrifft, dann auch Situation 1B

OD Welcome1A



OD Welcome1B



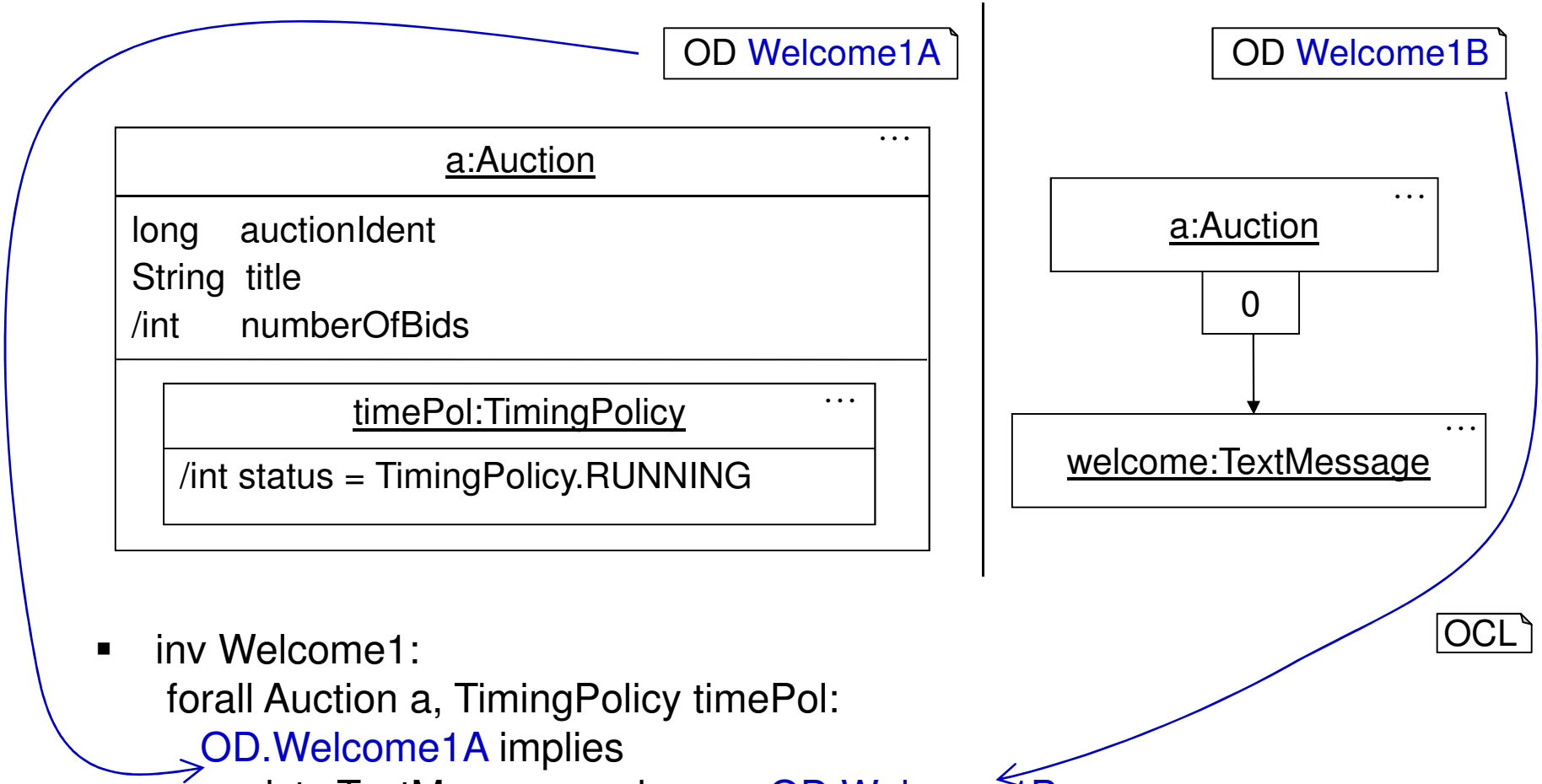
write

- inv Welcome1:

OCL

Logische Aussagen: Beispiel Implikation

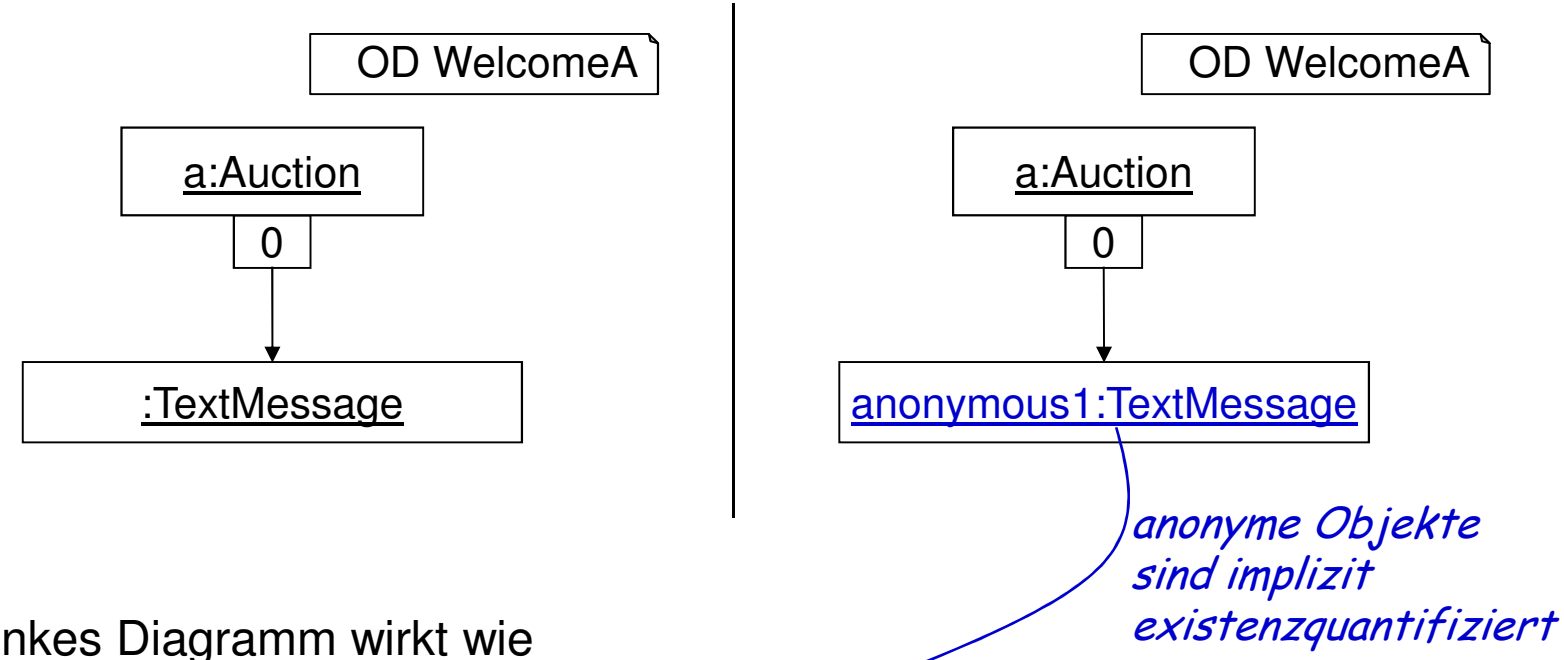
- Wenn Situation 1A zutrifft, dann auch Situation 1B



- inv Welcome1:
  forall Auction a, TimingPolicy timePol:
  OD.Welcome1A implies
  exists TextMessage welcome: OD.Welcome1B

Anonyme Objekte

- Ein anonymes Objekt wird behandelt wie eines mit einem eindeutigen, aber nach außen unbekannten Namen.
- Namensraum = Diagramm, Objekt ist existenzquantifiziert:



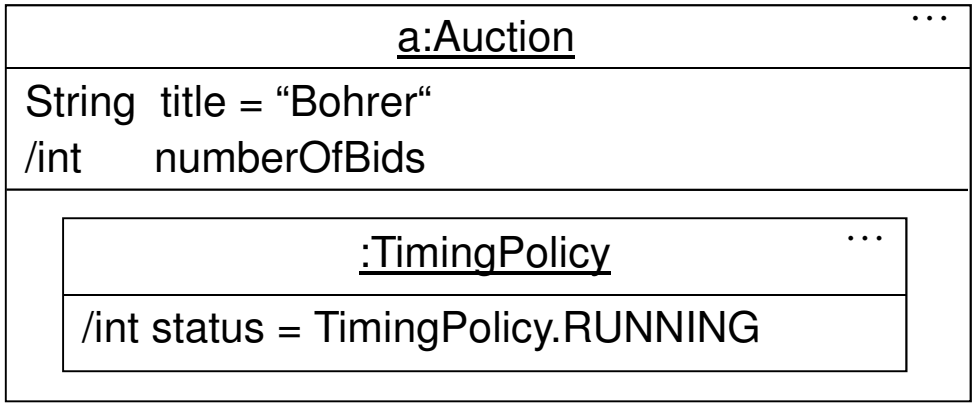
- Linkes Diagramm wirkt wie
  - ... exists TextMessage anonymous1: OD.WelcomeA
- Mehrere anonyme Objekte sind verschieden

Objektdiagramm als OCL-Aussage

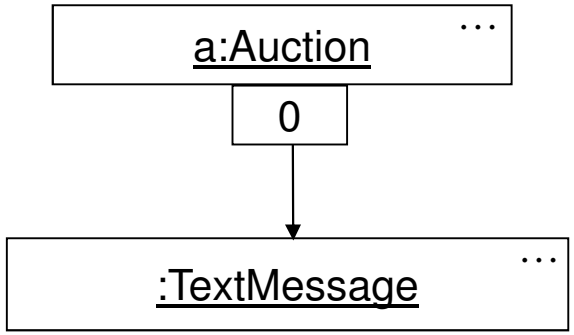
- inv Welcome2:
  forall Auction a: OD.Welcome2A implies OD.Welcome2B

OCL

OD Welcome2A



OD Welcome2B



- Jedes OD lässt sich in OCL expandieren. Obige Aussage wird zu:

write

- inv Welcome2:
  forall Auction a:

OCL

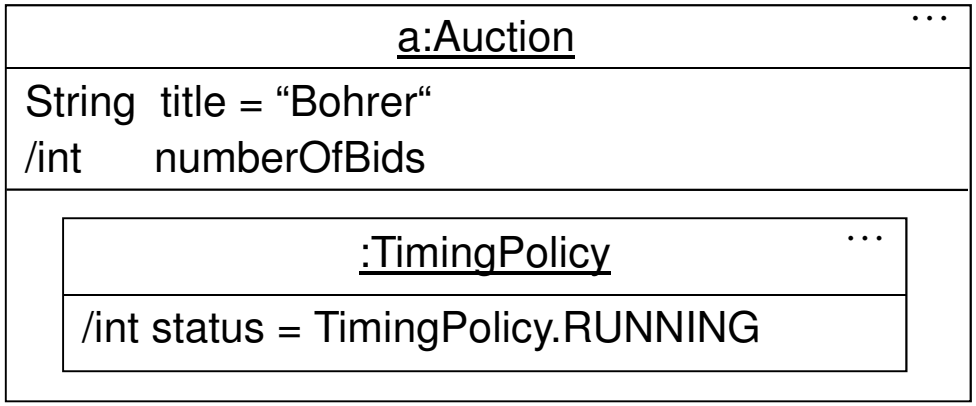
implies

Objektdiagramm als OCL-Aussage

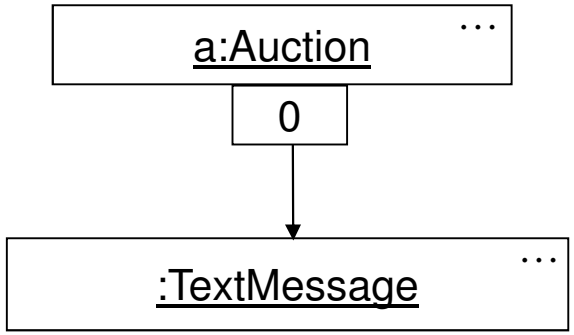
- inv Welcome2:
  forall Auction a: OD.Welcome2A implies OD.Welcome2B

OCL

OD Welcome1A



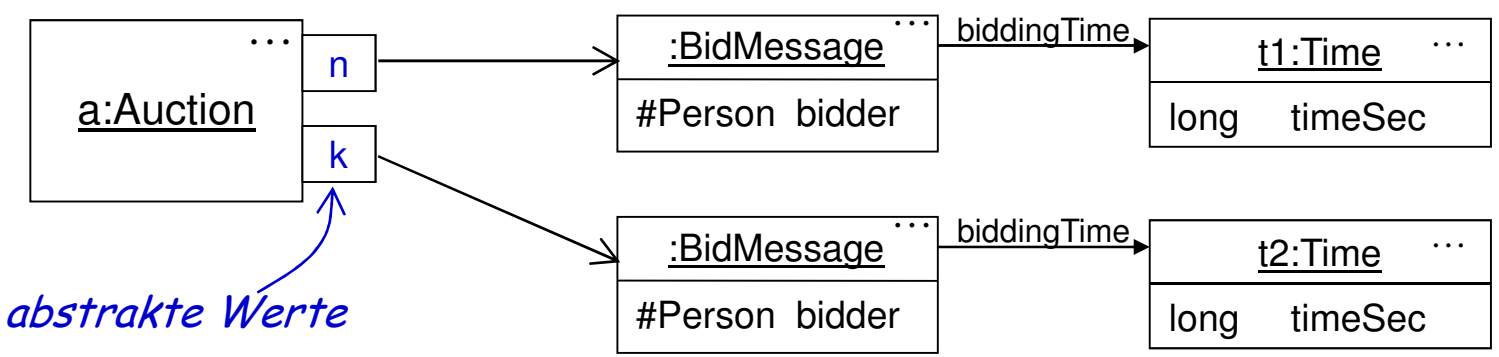
OD Welcome1B



- Jedes OD lässt sich in OCL expandieren. Obige Aussage wird zu:
- inv Welcome2:
  forall Auction a: (exists TiminingPolicy anon1:
  a.title == “Bohrer“ && a.timingPolicy == anon1 &&
  (Object)anon1 != a && anon1.status == TimingPolicy.RUNNING)
  implies (exists Textmessage anon2:
  (Object)anon2 != a && a.messages[0] == anon2)

OCL

Abstrakte Werte im OD

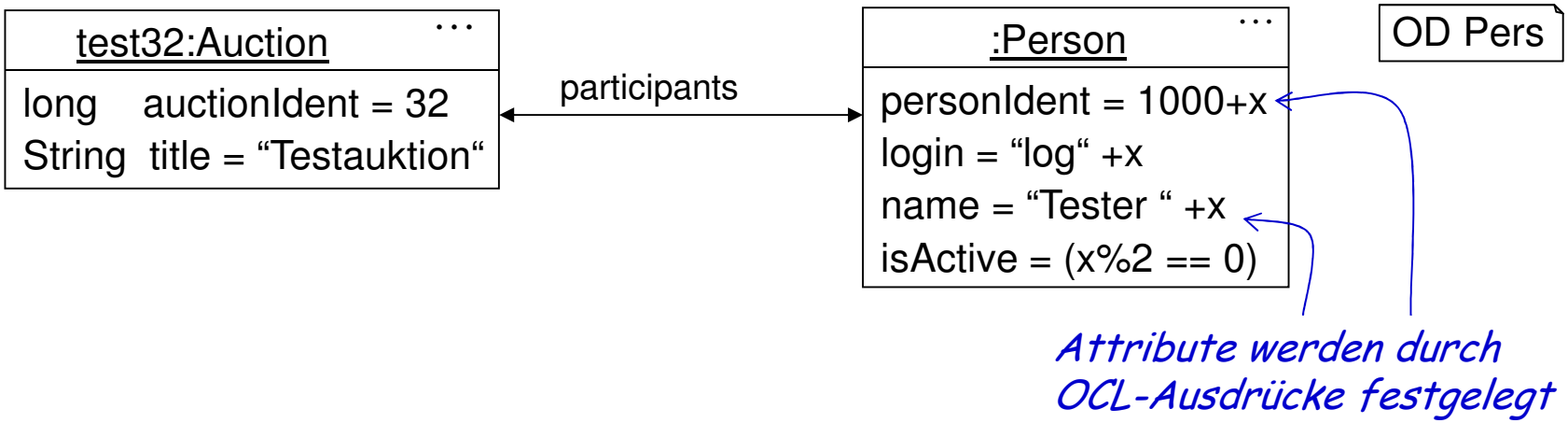


- „Abstrakte Werte“ werden durch Ausdrücke mit Variablen modelliert
- Aussage: Nachrichten sind in der Reihenfolge ihrer Entstehung in der Liste eingereiht.

OCL

- context Auction a, int n, k, Time t1, Time t2 inv:
  n<=k implies
  t1.timeSec <= t2.timeSec

Objektdiagramm als Vorlage



- OD beschreibt Teil einer Testsituation
- Hier wird der abstrakte Wert benutzt, um mehrere Inkarnationen des Diagramms zu charakterisieren:

  - context Auction test32 inv Test32:
    forall int x in {1..100}: OD.Pers

OCL

Methodischer Einsatz von ODs mit OCL

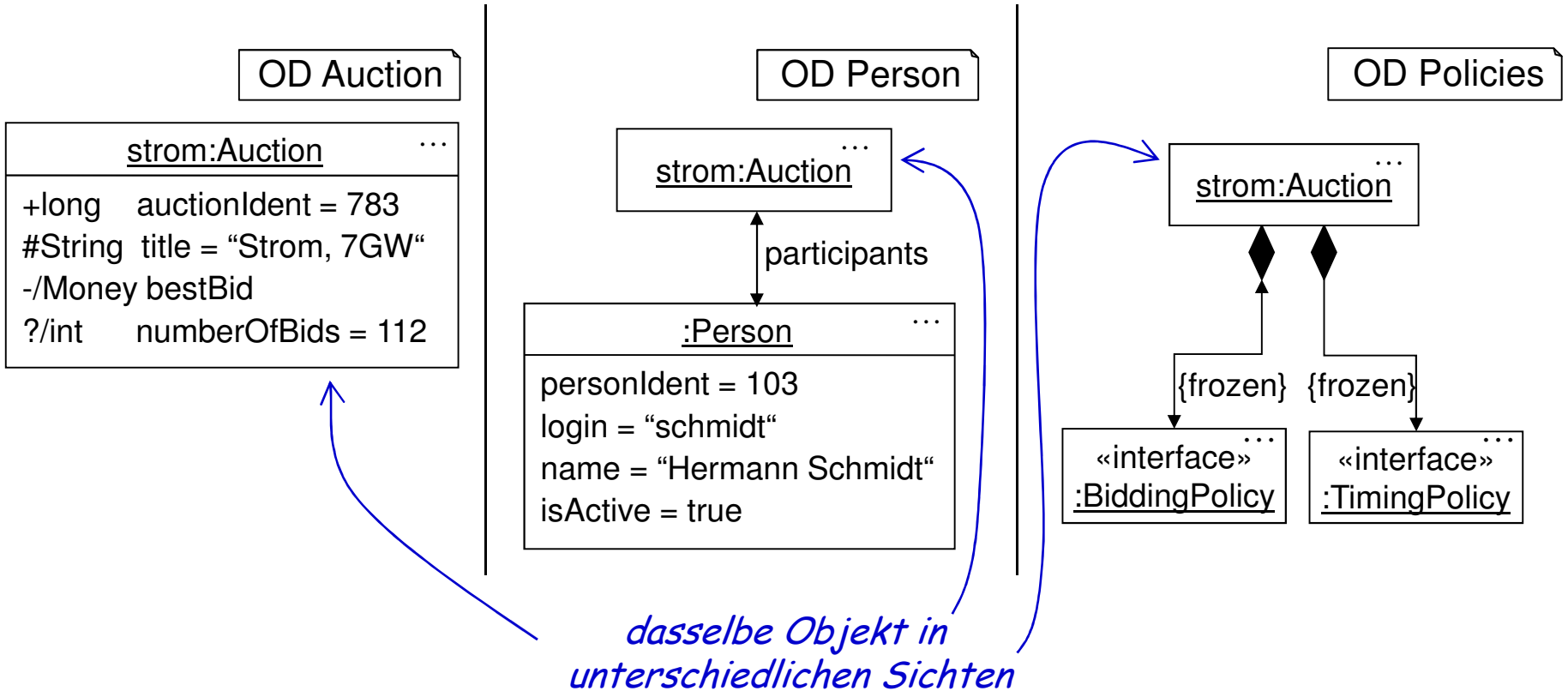
- Jedes OD ist in OCL umsetzbar
- OD beschreibt exemplarische Situation
- Variablen erlauben Abstraktion: „Abstrakte Werte“

- OCL erlaubt Charakterisierung der abstrakten Werte
- OCL beschreibt Beziehungen zwischen Attributen
- OCL erlaubt Kombination von Objektdiagrammen:
  - Komposition OD.A && OD.B
  - Unerwünschtes !OD.A
  - Alternativen OD.A || OD.B
  - Mehrfache Instanzen forall int x in {1..100}: OD.A
- Nutzung in
  - Methodenspezifikationen (Vor- und Nachbedingungen)
  - Beschreibung des Effekts von Konstruktoren / Modifikatoren

Komposition von OD's, Beispiel

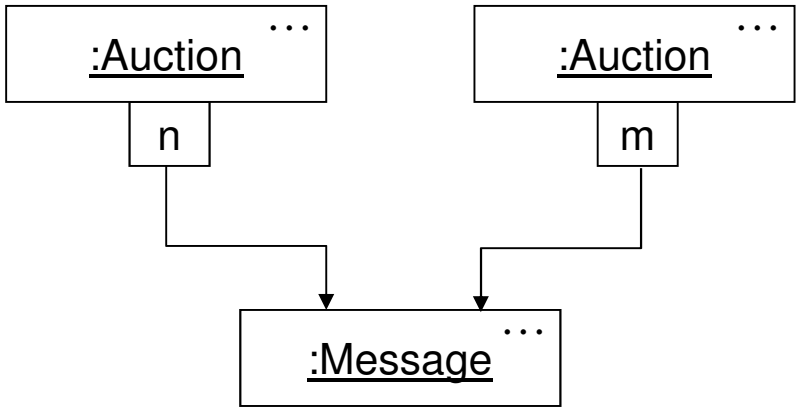
- „Zusammenkleben" von ODs an gemeinsamen Objekten:
- OD Auction && OD Person && OD Policies

OCL



Unerwünschte Situation als OD

- Folgende Situation tritt nie auf:



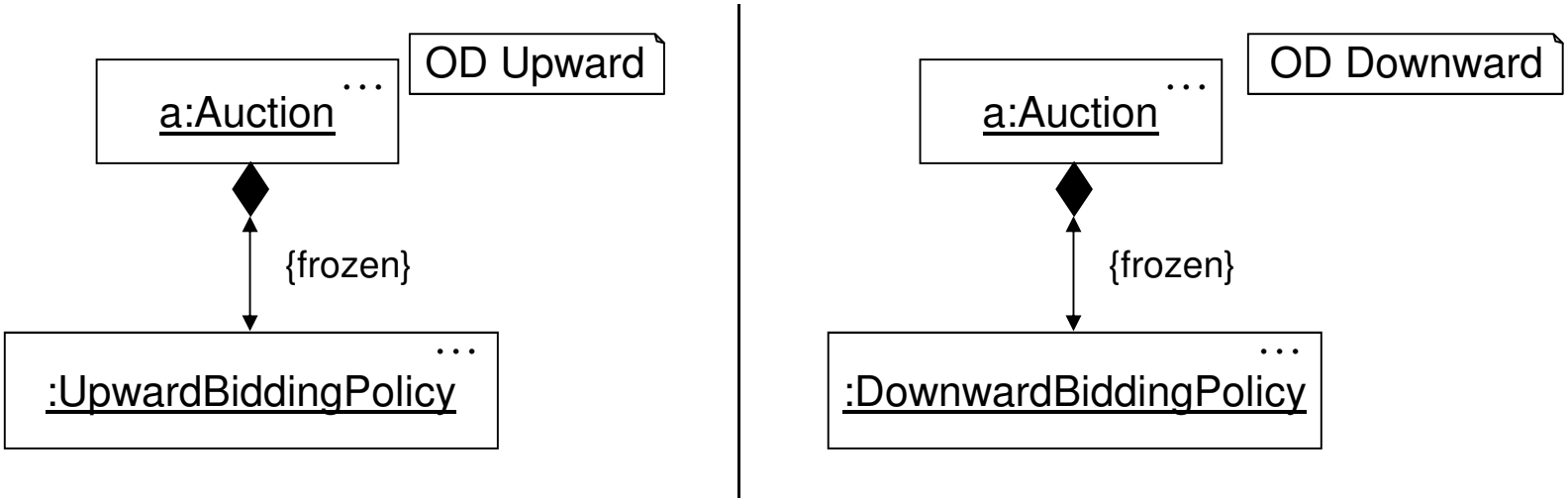
OD MsgIn2Auctions

- Das lässt sich in einer Invariante formulieren:
  - inv TwoAuctions:
    forall int n, int m: !OD.MsgIn2Auctions

OCL

Alternative OD‘s

- Eine von beiden Situationen tritt auf:

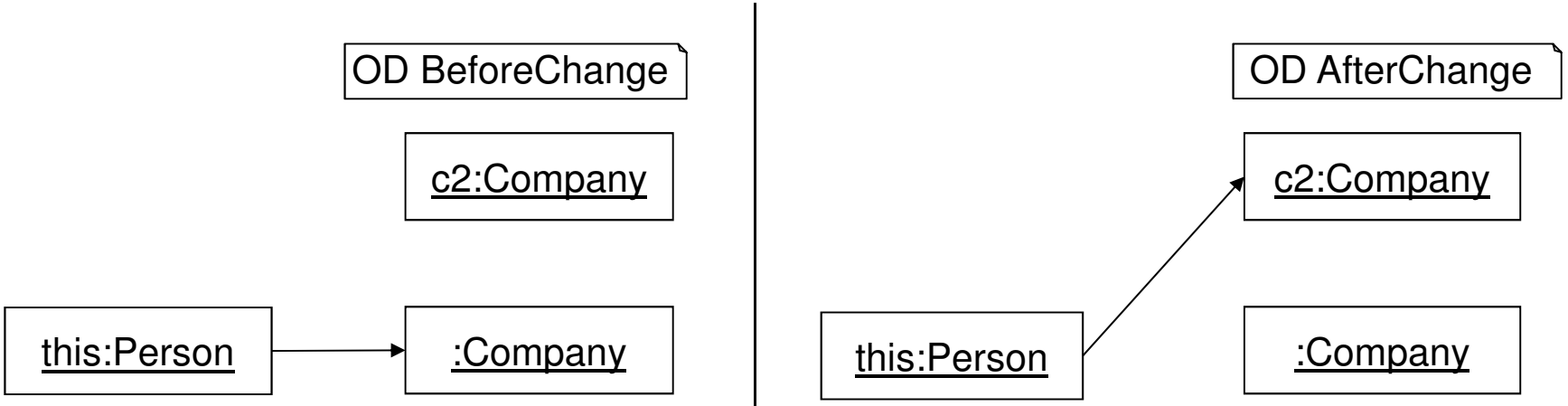


- inv:
  OD.Upward || OD.Downward

OCL

Methodenspezifikation mit Objektdiagrammen

- Wenn zunächst die linke Situation gilt, dann gilt nach Methodenaufruf die rechte:



Objektdiagramm: Zustand davor

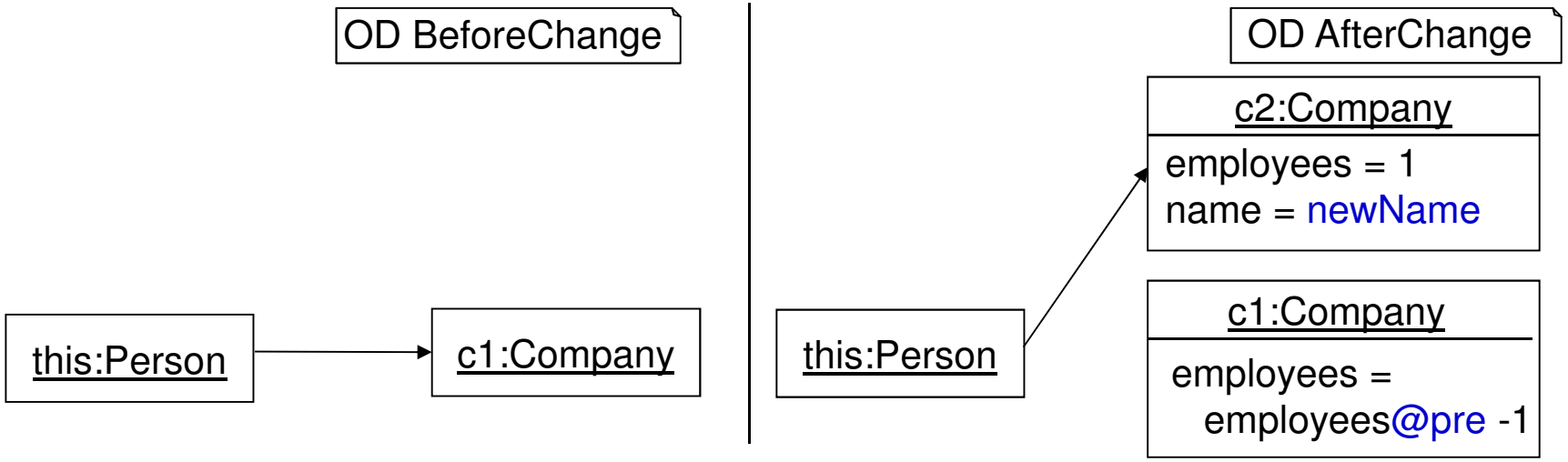
Objektdiagramm: Zustand danach

- c2 wird durch let gebunden, this durch den Methoden-Kontext:
  - context Person.changeCompany(String newName)
    let c2 = any { Company c | c.name==newName }
    pre: OD.BeforeChange
    post: OD.AfterChange

OCL

Objekterzeugung

- Rechts ist ein neues Objekt genannt.
- Inhalte der rechten Objekte beschreiben Verhalten der Methode:



Objektdiagramm: Zustand davor

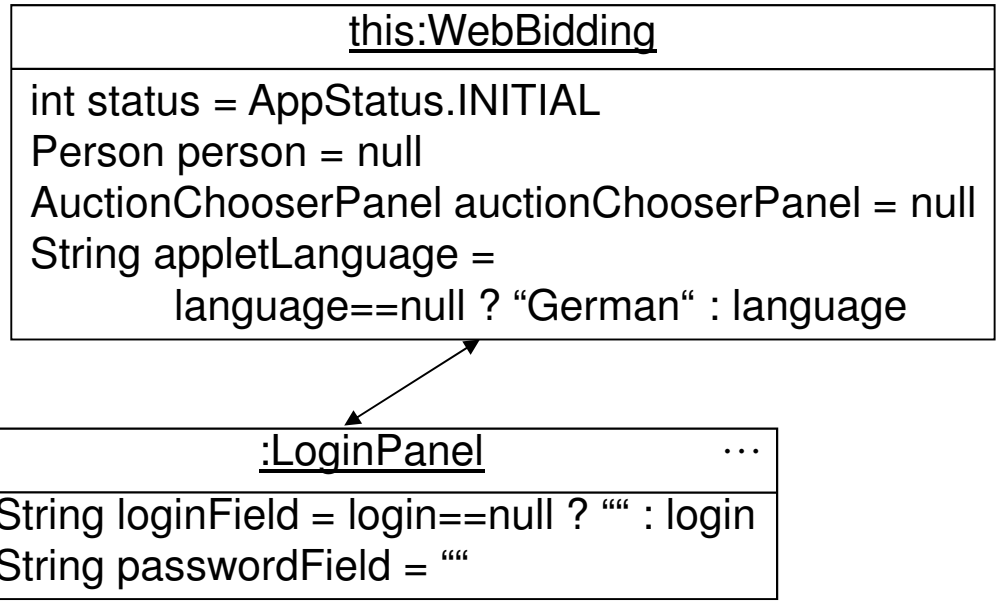
Objektdiagramm: Zustand danach

OCL

- context Person.changeCompany(String newName)
  let c1 = this.company
  pre: OD.BeforeChange
  post: exists Company c2: new(c2) && OD.AfterChange

Objektdiagramm im Code: Initialisierung

- Objektdiagramm als Template für Objektstrukturen
  - Es kann Code erzeugt werden, der das OD „instanziert“
- Beispiel:



OD WBInit

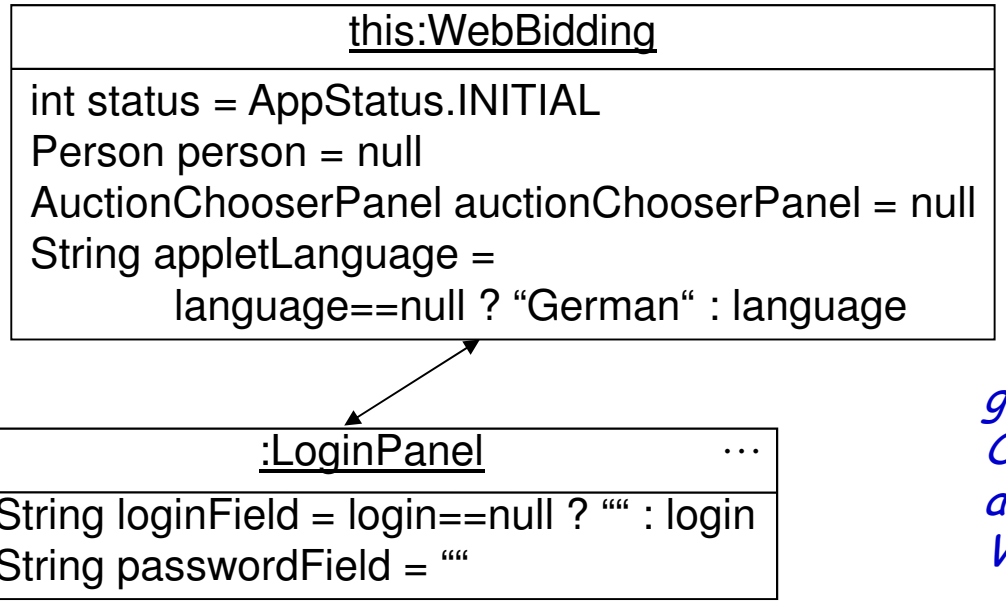
```
void wbInit(String login, String language)
        { this.status = AppStatus.INITIAL;
          this.person = null; ... loginPanel = new LoginPanel(); ... }
```

Java

- Notwendig: geeignete Konstruktoren, Zugriff auf Attribute, ...
- Typisch: Generatoranweisung beschreibt Namen der generierten Methode,

Objektdiagramm im Code: Erzeugung

- Objektdiagramm als Template für Objektstrukturen
  - Es kann Fabrik-Code erzeugt werden, der das OD „erzeugt“
- Beispiel:



OD WBInit

gleiches Objektdiagramm, aber andere Verwendung!

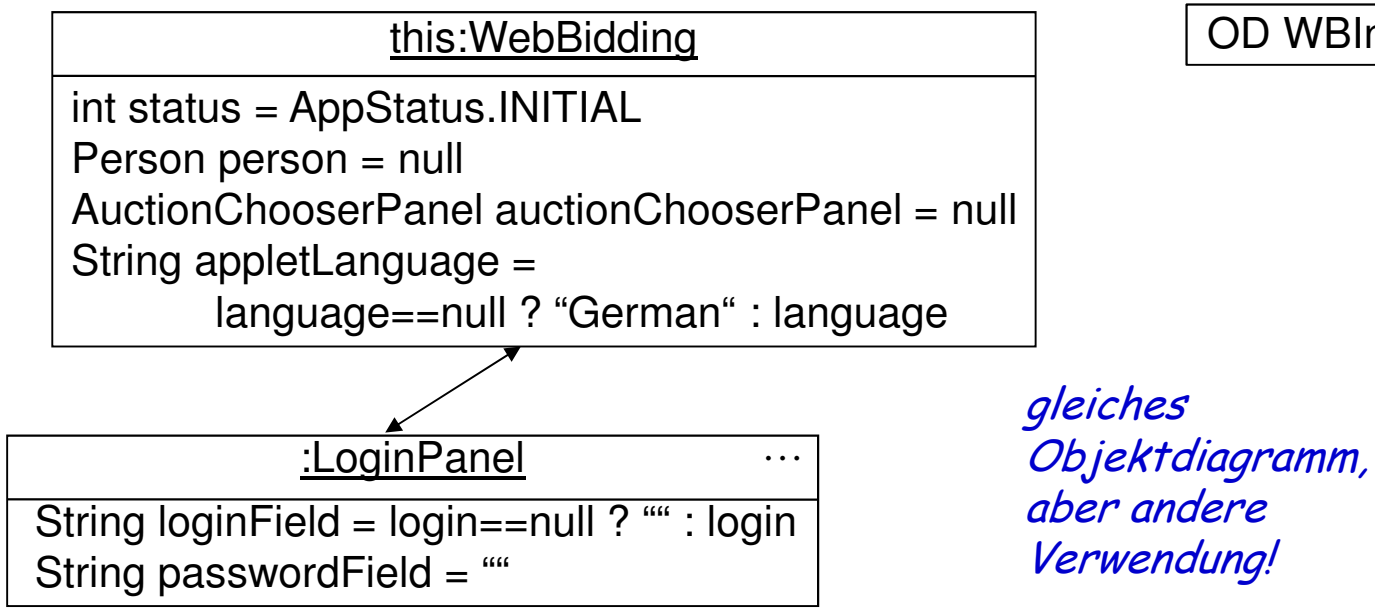
```
WebBidding wbInit(String login, String language)
      { WebBidding wb = new WebBdding();
        wb.status = AppStatus.INITIAL;
        wb.person = null; ... loginPanel = new LoginPanel(); ...
        return wb; }
```

Java

Objektdiagramm im Code: Prädikat

- Objektdiagramm als Prüf-Template
  - Es kann Code erzeugt werden, der die Erfüllung des OD prüft
- Beispiel:

OD WBInit



gleiches Objektdiagramm, aber andere Verwendung!

Java

```
boolean wbInitCheck(String login, String language)
        {   return this.status == AppStatus.INITIAL &&
                  this.person == null &&
                  (loginPanel.loginField == (login==null ? ““ : login)) && ... }
```

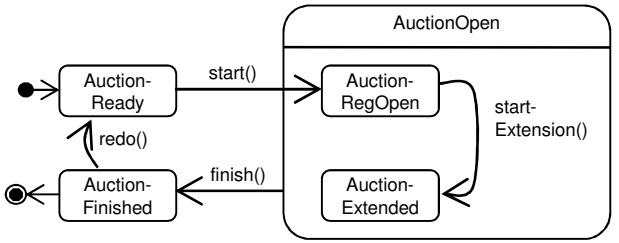
Zusammenfassung Objektdiagramme

- Objektdiagramme sind exemplarisch.

- Vielfältige Einsatzmöglichkeiten:
  - Exemplarische Situation zur Diskussion mit Kunden/Kollegen
  - Architekturbeschreibung statischer Anteile (Situation gilt immer)
  - Vorbedingung für einen Methodenaufruf
  - Nachbedingung für einen Methodenaufruf
  - Unerwünschte Situation
  - Ausgangssituation für einen Test
  - Sollsituation für einen Test

- Kombination der OD mit OCL erhöht Beschreibungsmächtigkeit
  - Komposition, Alternativen, Ausschluss, ...
- OD kann grundsätzlich in OCL übersetzt werden

Statechart

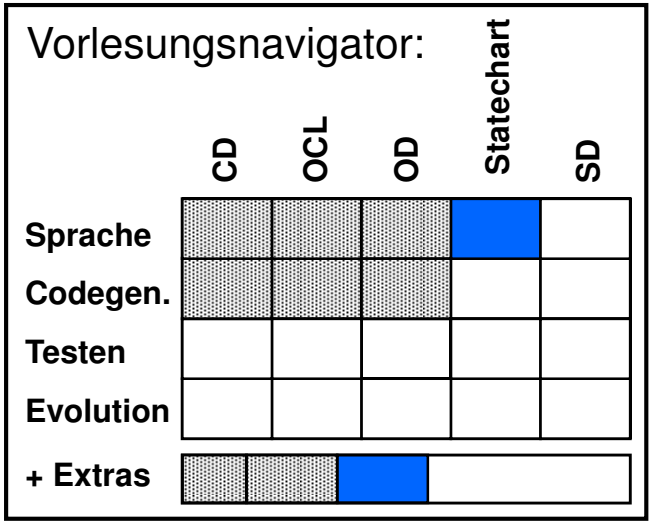


Modellbasierte Softwareentwicklung

- 5. Statecharts
- 5.1. Grundlagen: Automatentheorie / Verhalten

Prof. Dr. Bernhard Rumpe
Lehrstuhl für Software Engineering
RWTH Aachen

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



Statecharts

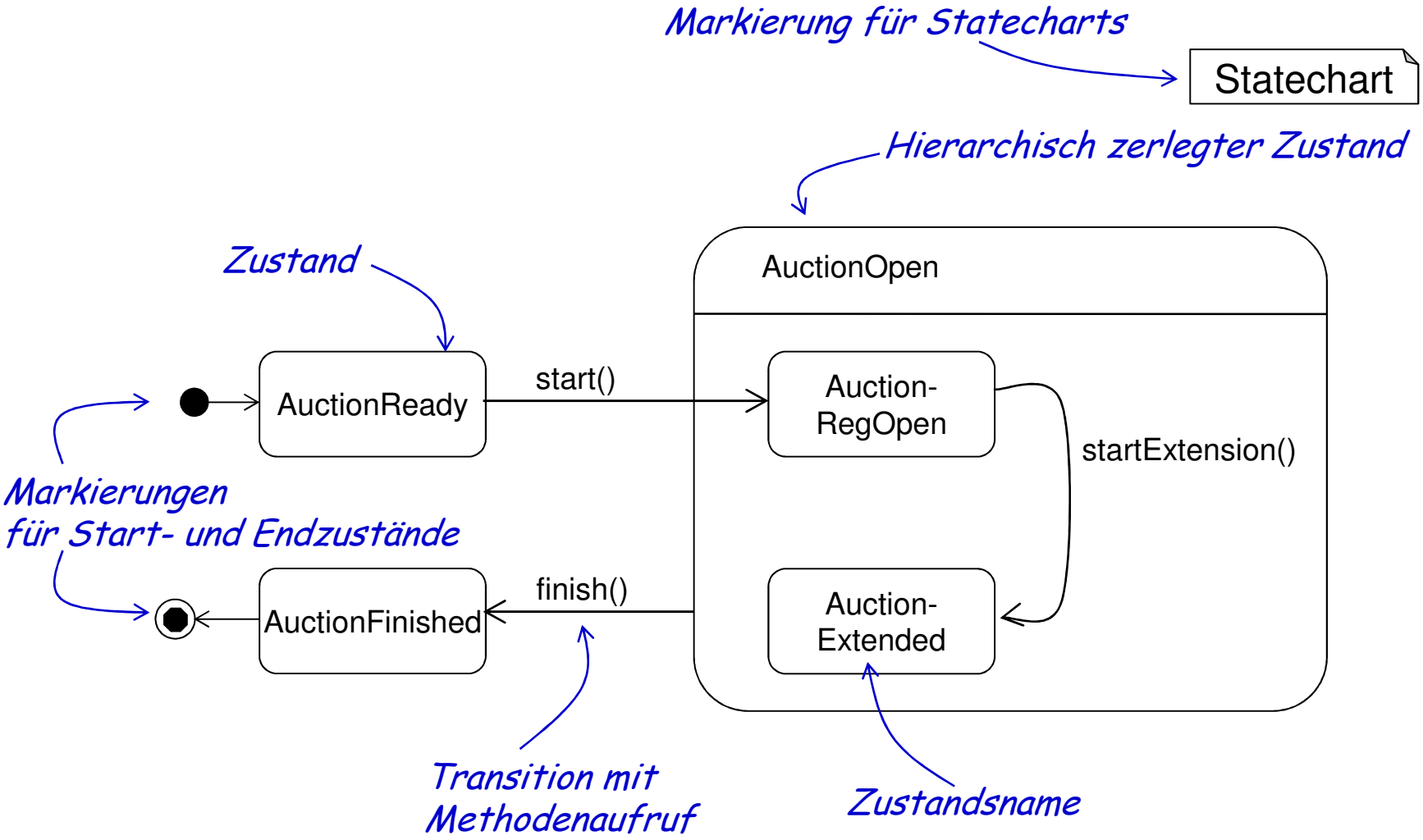
- Ziel ist die Beschreibung von Objektverhalten
- Annahmen:
  - Objekte haben im Zustand einen Daten- und einen Kontrollanteil
  - Kontrollanteil beschrieben durch endlichen Zustandsraum
  - Objektveränderungen sind Transitionen

- Statecharts erweitern Automatentheorie:
  - Hierarchische Zustände,
  - Aktionen in Transitionen und Zuständen, ...

- Historie:
  - Statecharts von David Harel, 1987 eingeführt
  - in viele Modellierungssprachen übernommen
  - viele Varianten entwickelt
  - von Beginn an Teil der UML

Beispiel-Statechart

- Einfaches Statechart für den „Ablauf“ einer Auktion:



Aufgabenvielfalt eines Statechart

- Darstellung des Lebenszyklus eines Objekts
- Implementierungsbeschreibung einer Methode
- Implementierungsbeschreibung des Verhaltens eines Objekts
- Abstrakte Anforderungsbeschreibung an den Zustandsraum eines Objekts
- Darstellung der Reihenfolge von erlaubten Eintreten von Stimuli (Aufrufreihenfolge)
- Charakterisierung der möglichen oder erlaubten Verhalten eines Objekts
- Bindeglied zwischen Zustands- und Verhaltensbeschreibung

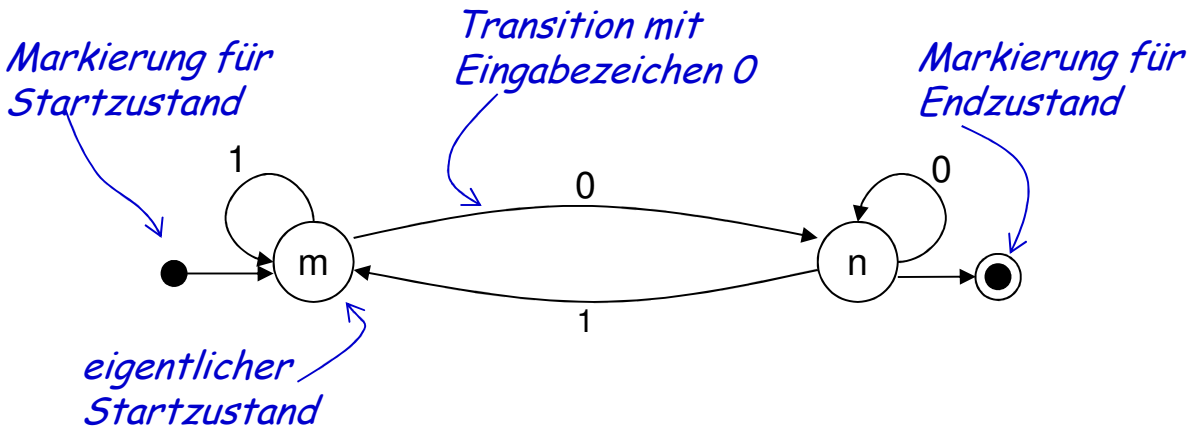
- Zum Besseren Verständnis dieser Aufgabenvielfalt zunächst einige Grundlagen der Automatentheorie

Automatentheorie

- Erkennender Automat (Z,E,t,S,F) hat
- (auch: nichtdeterministischer, alphabetischer Rabin-Scott Automat (RSA))
  - Menge von Zuständen Z
  - Eingabealphabet E
  - Menge von Startzuständen $S \subseteq Z$
  - Menge von Endzuständen $F \subseteq Z$
  - Transitionsrelation $t \subseteq Z \times E^{\varepsilon} \times Z$

- wobei:
  - ε das nicht vorhandene Eingabezeichen in spontanen Transitionen darstellt
  - $E^{\varepsilon} = E \cup \{\varepsilon\}$
  - Alle Mengen S, E, Z, F nicht leer und endlich.

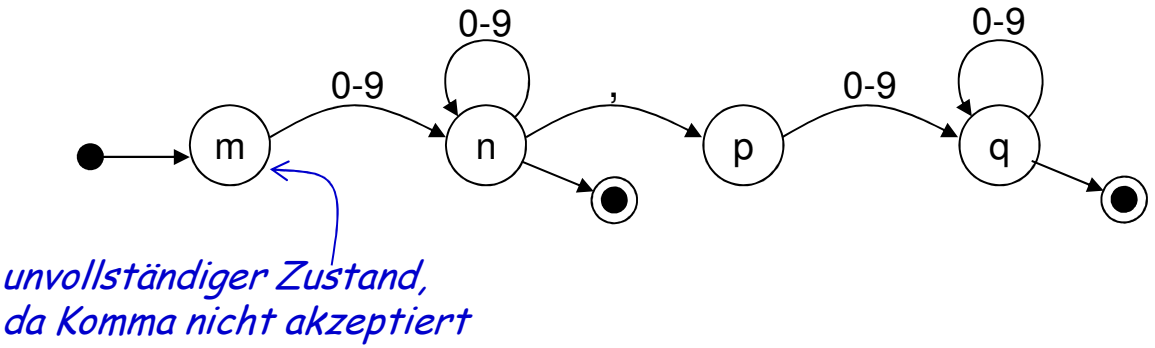
Darstellung erkennender Automat

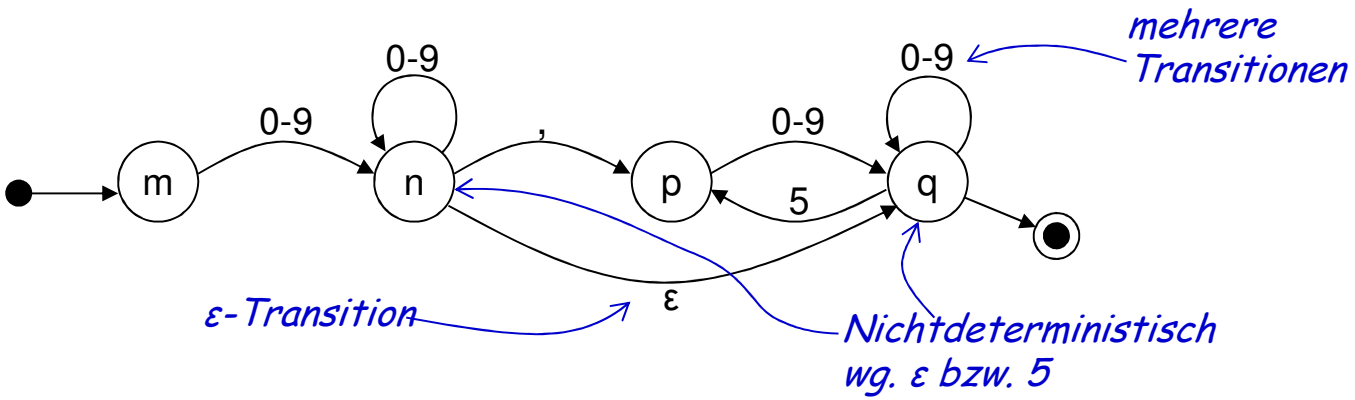


- Welche Sprache akzeptiert der Automat?

Beispiele erkennender Automaten

erkennende Automaten





Schaltbereitschaft, Semantik

- Eine Transition ist schaltbereit, wenn im Quellzustand das entsprechende Zeichen anliegt oder die Transition kein Eingabezeichen verlangt (also spontan ist).

- Semantik eines erkennenden Automaten ist die Menge an Eingaben (Wörter aus E), für die ein Weg von einem Startzustand zu einem Endzustand existiert.

- „Erkennung“ ist aber zu schwach für Verhaltensbeschreibung.
  - Deshalb Erweiterung der Automaten um „Ausgaben“
  - Mealy/Moore-Automaten!

Mealy-Automat

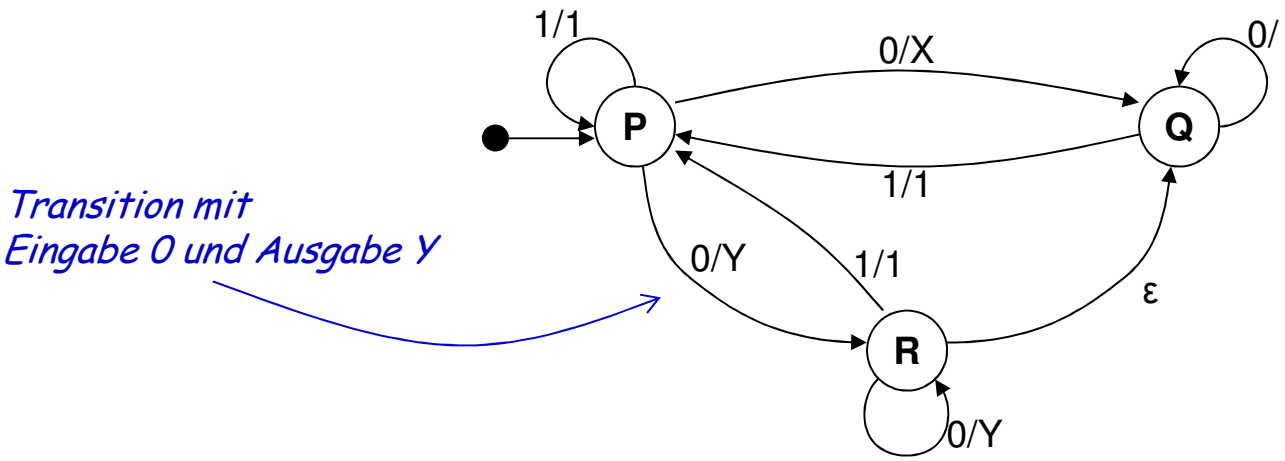
- Ein Mealy-Automat (Z,E,A,t,S,F)
  - beinhaltet erkennenden Automaten (Z,E,t,S,F),
  - Ausgabealphabet A
  - um Ausgabe erweiterte Transitionsrelation
    $t \subseteq Z \times E^{\varepsilon} \times Z \times A$

- Semantik des Mealy-Automat ist keine Menge (E*) die erkannt wird!

- Semantik des Mealy-Automat ist eine Relation zwischen Eingabe- und Ausgabe-Wörtern ($E^* \times A^*$):
  - das nach außen sichtbare „Verhalten“ des Automaten

Beispiel Mealy-Automat

Mealy-Automat



Beispieleingabe:	10001001
Transitionspfad:	**P** 1/1 **P** 0/X **Q** 0/ **Q** 0/ **Q** 1/1 **P** 0/X **Q** 0/ **Q** 1/1 **P**
Ausgabe:	1X1X1
Zustandsübergänge:	**P P Q Q Q P Q Q P**

Element der Semantik ist: (10001001 , 1X1X1)

Zustände werden als gekapselt angenommen
und bei der Semantik nicht berücksichtigt.

Theorie und Interpretation

- Mealy-Automaten bilden eine wohl-untersuchte Theorie
  - Deterministisch, Vervollständigung, Minimierung, Mächtigkeit, etc.

- Anwendung in der Praxis bedarf einer Interpretation der Theorie
  - Bezug zur modellierten Welt:
    - Was ist ein Zustand?
    - Was ein Eingabezeichen?
    - Was eine Ausgabe?

- Interpretationsspielräume:
  - Eine gute Theorie lässt sich auf viele Situationen der realen Welt anwenden.

Anwendung Automatentheorie in der OO-Modellierung

- Interpretationsmöglichkeiten:
  - Zustandsraum eines Objekts im allgemeinen unendlich vs. Endlicher Zustandsmenge im Mealy-Automat
  - Zustandsänderung durch Methodenaufruf, asynchrone Nachricht via Corba, Time-Out, ...?
  - Was sind Ausgaben?
  - Was ist eine spontane Transition bei OO?
  - Start-, Endzustände in OO?

Interpretation für Statecharts in UML/P

- Zustand des Automaten = Klasse von Zuständen des Objekts.

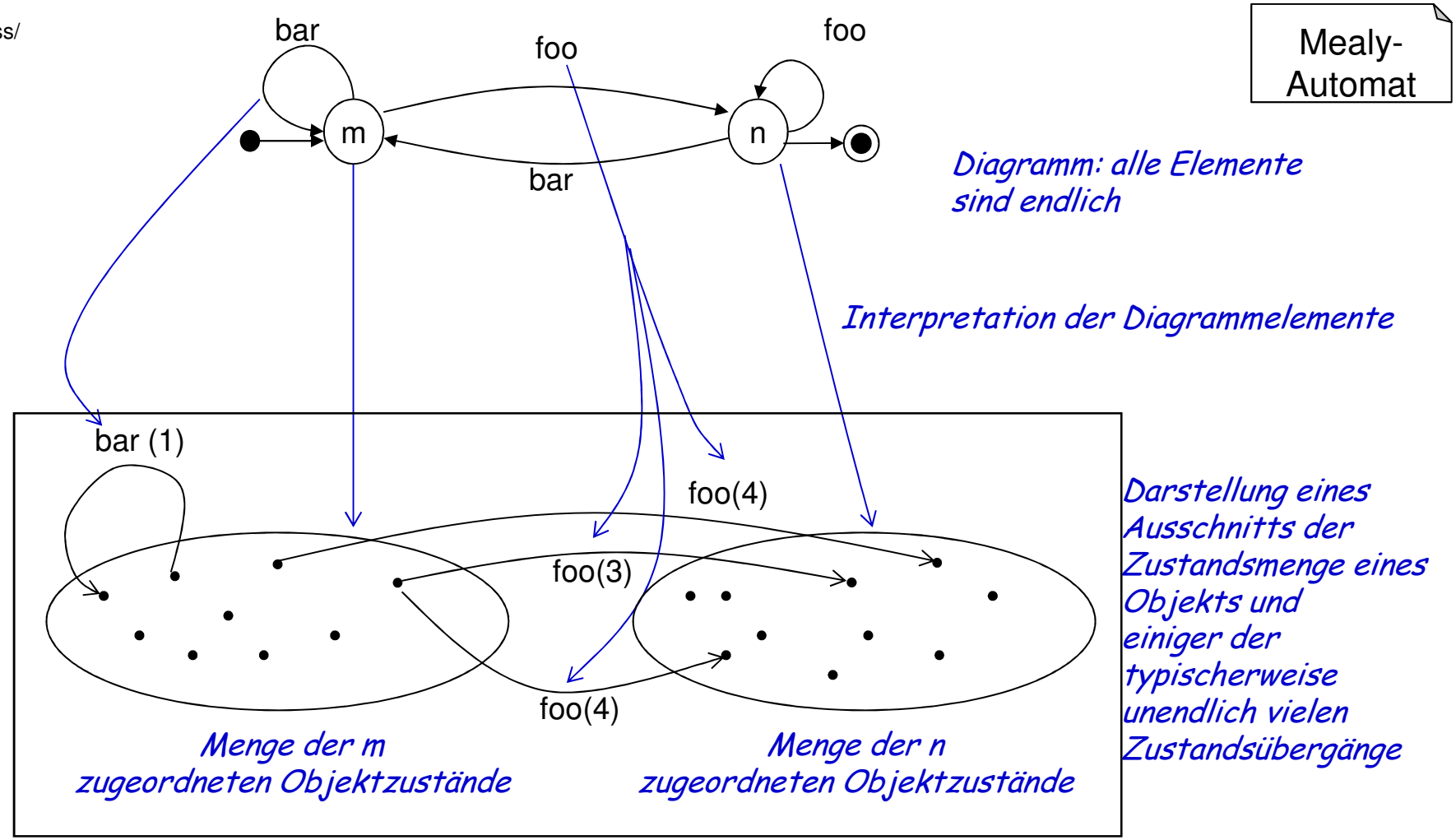
- Startzustand = Klasse von Objektzuständen, die unmittelbar nach Konstruktion (new ...) auftreten.
- Endzustand spielt keine Rolle, da in Java Garbage Collection Objekte „terminiert“

- Eingabezeichen = Methodenaufruf inklusive der Argumente
- Ausgabezeichen = Ausführung eines Methodenrumpfs:
  - Beinhaltet Attributänderungen, andere Methodenaufrufe
- Transition = Ausführung eines Methodenrumpfs.

- Unterscheidung Diagrammzustand und Objektzustand!

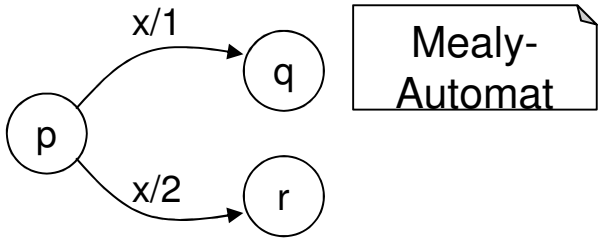
Beziehung Diagramm- und Objektzustände

Discuss/
Anim.



Nichtdeterminismus (N.Det.) im Automat

- Sind zwei Transitionen schaltbereit:
  so weiß der Nutzer des Objekts,
  eine der Transitionen wird genommen.

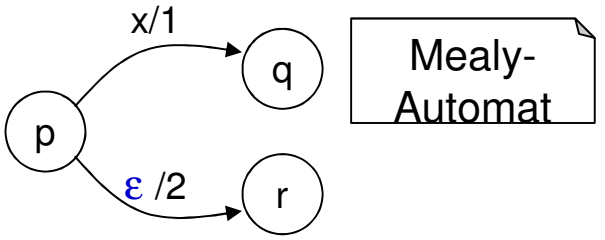


- Entscheidung darüber kann
  - (a) abhängig von Details des Zustands sein, die im Modell fehlen
    (= N.Det. durch Abstraktion: Unterspezifikation)
  - (b) dem Entwickler überlassen sein
    (= N.Det. als Entwurfsfreiheit : Unterspezifikation)
  - (c) zur Laufzeit geschehen (= N.Det. im System)
- Entscheidung kann dem Entwickler oder System überlassen sein
  - das macht für den Nutzer keinen Unterschied!

- Festlegung der Interpretation:
  - N.Det. des Automaten als Konzept zur Unterspezifikation!
- Prinzip der Unterspezifikation: Wo keine Aussage getroffen wurde, ist nichts bekannt!

ε - Transition

- ε - Transitionen sind „spontane“ Übergänge

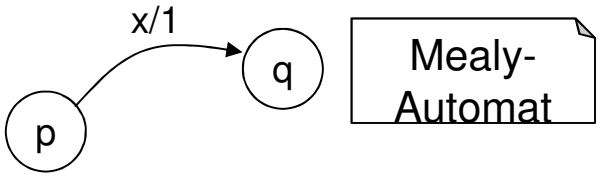


- Interpretationsmöglichkeiten:
  (1) Timer ist abgelaufen und verursacht Transition
  (2) Automat ist unvollständig: Nachricht, die zu dieser Transition führt, wurde nicht modelliert, aber Effekt durch Zustandswechsel sichtbar.
  (3) Die Transition ist Konsequenz einer vorhergehenden Transition und wird vom System automatisch ausgeführt.

- (1) erfordert Sprachmittel in der Programmiersprache
- (2) erlaubt Abstraktion, verhindert aber Codegenerierung
- (3) erlaubt lange Aktionen in Sequenzen, Verzweigungen und sogar Iteration eines Methodenrumpfs in Einzelschritte zu zerlegen: notationeller Komfort

Unvollständigkeit

- Im aktuellen Zustand „p“ ist keine Transition für Zeichen „y“ schaltbereit

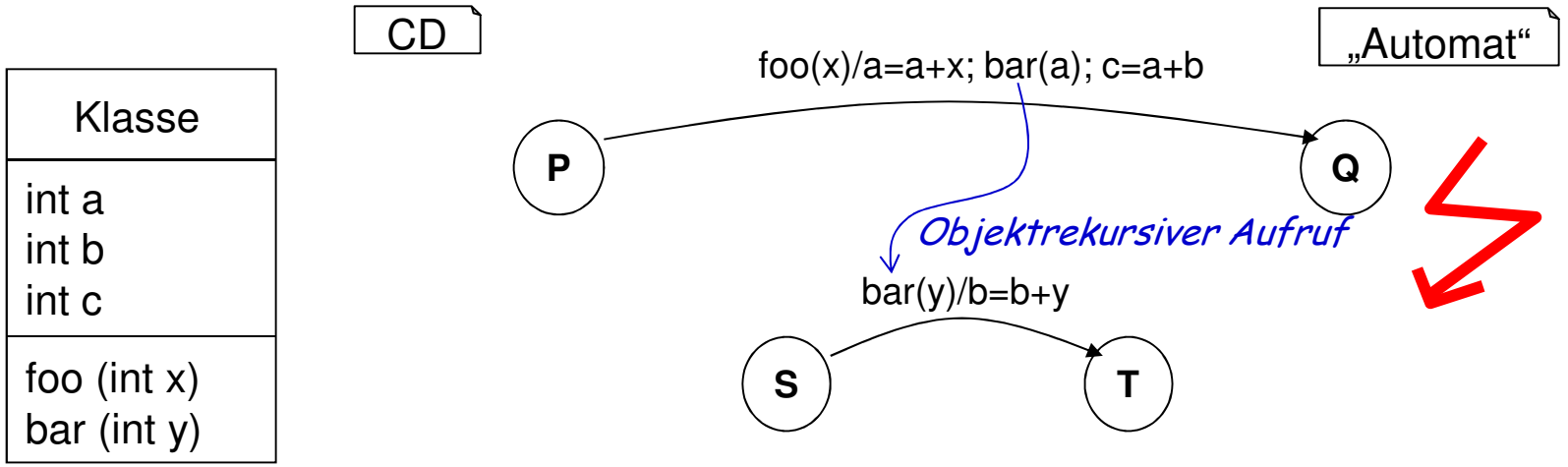


- Interpretationsmöglichkeiten für :
  1) Ignorieren: Keine Aktion ausführen, keinen Zustand ändern
  2) Chaos: Beliebige Reaktion erlaubt einen beliebigen Zustandsübergang und eine beliebige Aktion.
  3) Fehlerzustand wird eingenommen (und nur durch Rücksetznachricht verlassen).
  4) Fehlermeldung wie das Smalltalk „Message not understood“, aber keine Zustandsänderung

- 1, 3 und 4 sind für Implementierung geeignet: Codegenerator!
- Variante 2 für die Verwendung von Statecharts in der Spezifikation.
  - Chaos erlaubt die robuste Implementierung durch spätere Entwurfsentscheidungen (Hinzufügen von Transitionen)
  - Chaos = Nicht Wissen = Unterspezifikation

Beschreibungsmächtigkeit

- Implizite Annahmen bei Automaten:
  - Ankommende Stimuli werden sequentiell verarbeitet
  - Keine Parallelität im einzelnen Objekt bzw. der Transition
- Java erlaubt Parallelität und Rekursion (auf Methode oder Objekten)
- Statecharts der UML können Rekursion nicht adäquat darstellen.



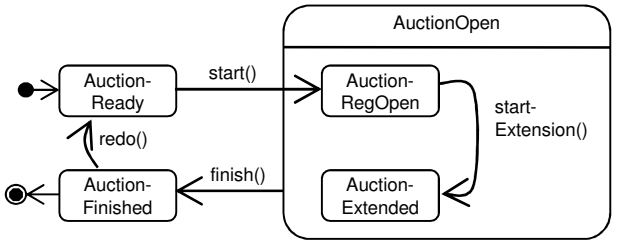
- Abhilfe: Java-Code ist synchronized und
- Annahme, dass intern aufgerufene Methoden (wie bar()) „Hilfsmethoden“ sind, die Zustände nicht nutzen oder beeinflussen!
  (Harel verbietet Objektrekursion sogar!)

SE SOFTWARE ENGINEERING

RWTH AACHEN

Statechart

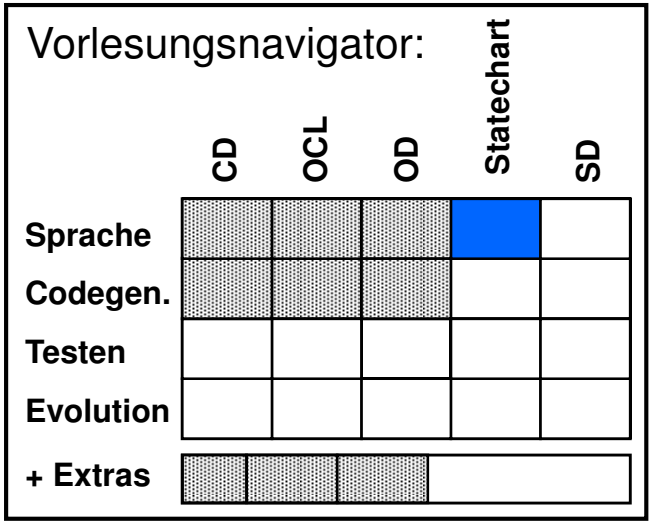
Modellbasierte Softwareentwicklung



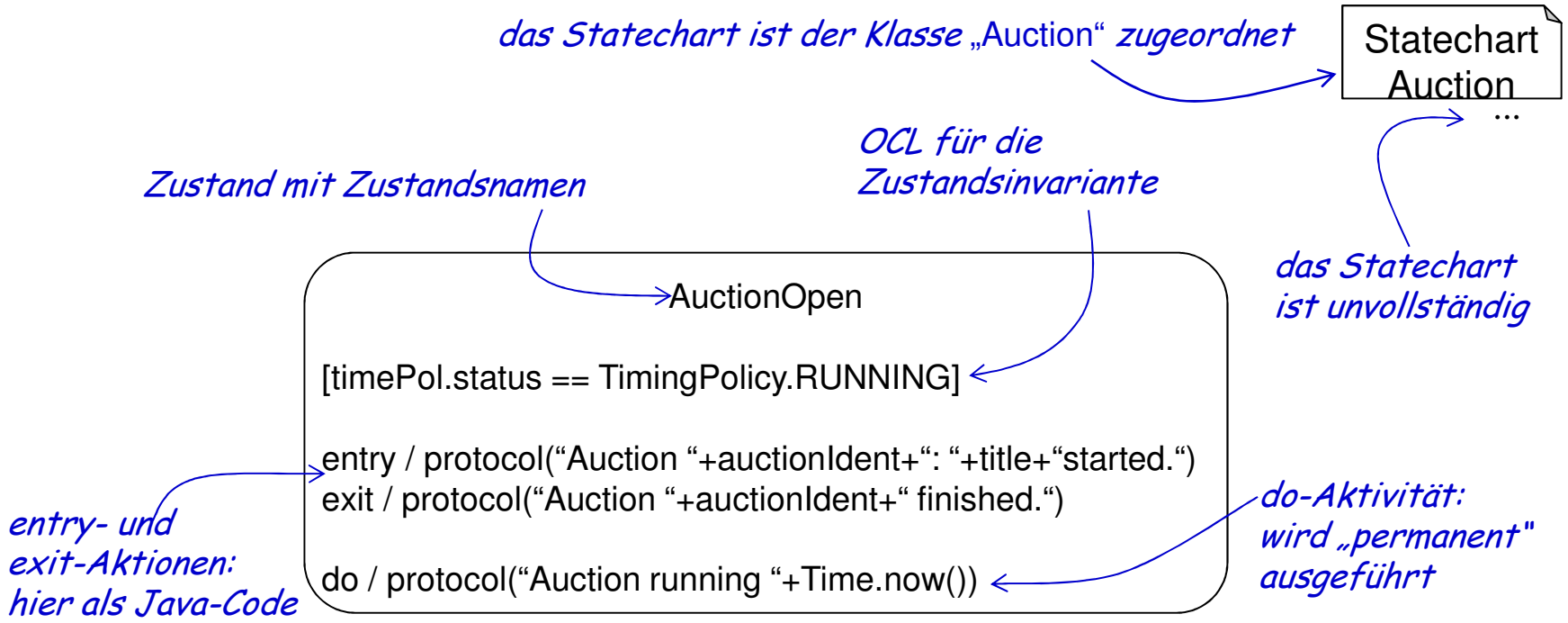
- 5. Statecharts
- 5.2. Zustände

Prof. Dr. Bernhard Rumpe
Lehrstuhl für Software Engineering
RWTH Aachen

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



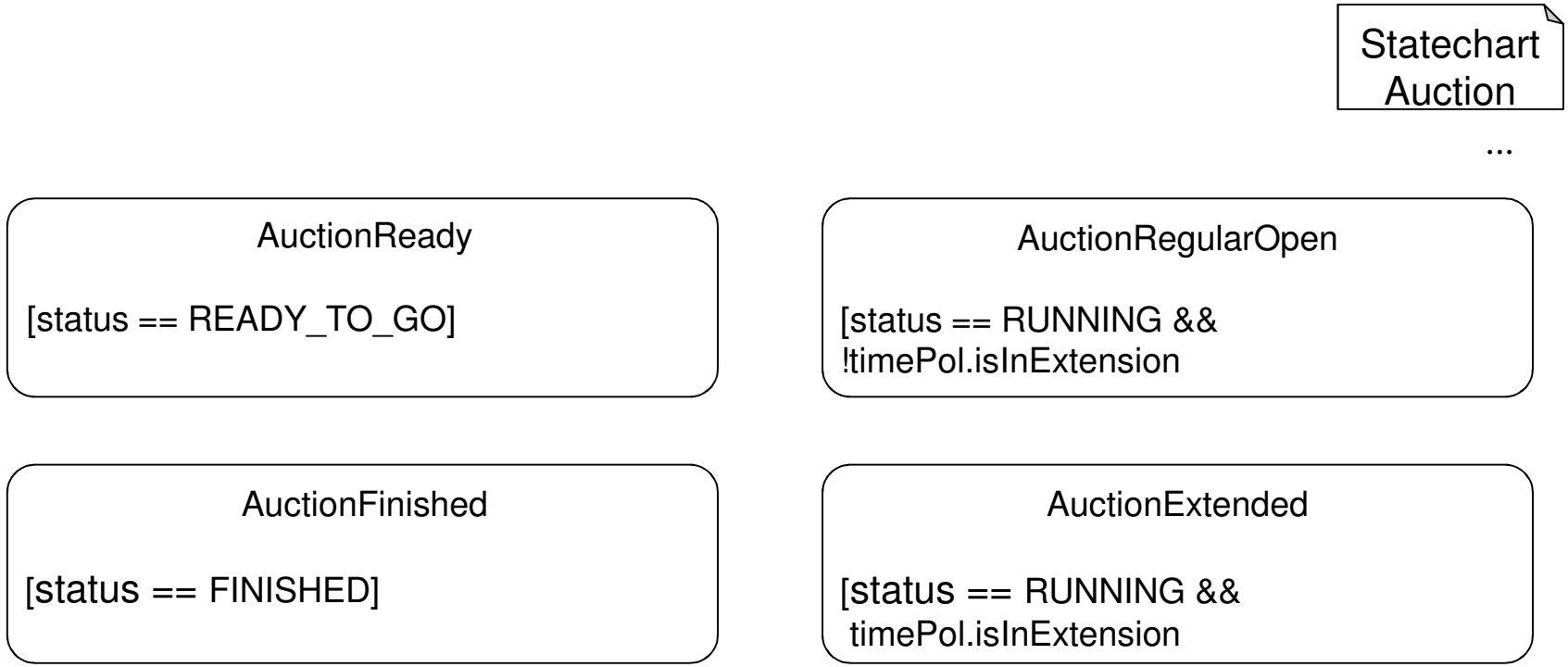
Zustände



- Zustände haben
  - Zustandsinvarianten
  - entry- / exit- und do-Aktionen
  - Subzustände (siehe später)

Zustandsinvarianten

- Zustandsinvariante formuliert in OCL über den Attributen des Objekts (und abhängiger Objekte)
- Verbindung zwischen Diagrammzustand und Objektzuständen



- Disjunktheit ist hier gegeben, aber nicht allgemein notwendig!

Daten- und Kontrollzustände

- Sind Zustandsinvarianten nicht disjunkt oder fehlen, so sind Automatenzustände „Kontrollzustände“
  - Bei Implementierung ist zusätzliches Attribut notwendig sie darzustellen
- Sind Invarianten disjunkt, ist das nicht notwendig: Automatenzustände können als „Datenzustände“ gesehen werden.
- Markierung der Zustände im Automat durch Stereotypen:

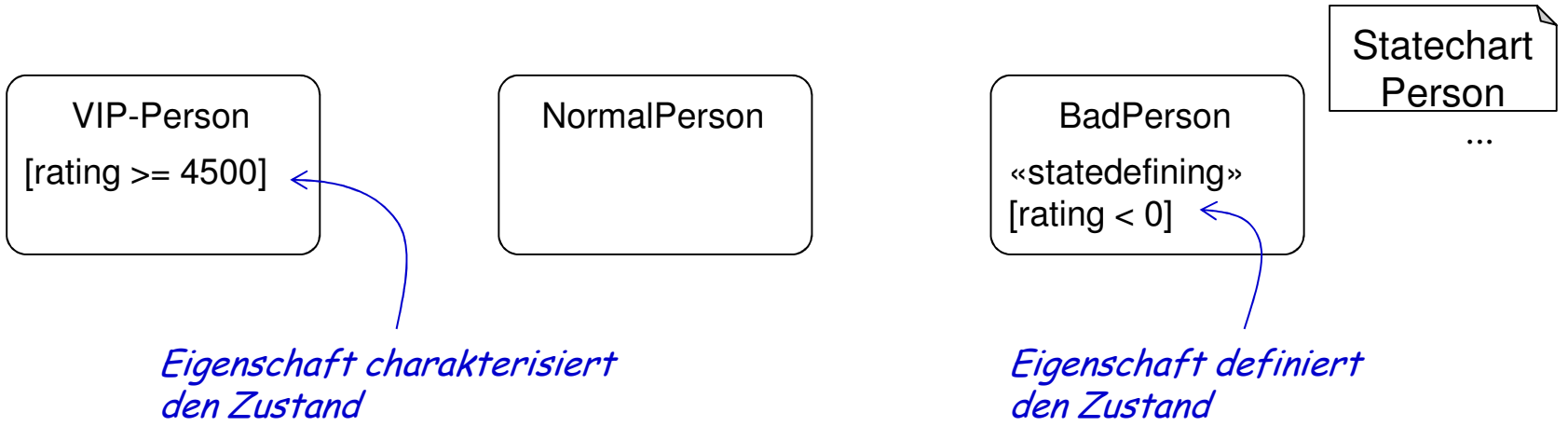
Statechart

«datastate»
Zustandsname

«controlstate»
Zustandsname

- Normalerweise nicht in einem Diagramm mischen!
- Umwandlung Kontroll- in Datenzustände z.B. durch Einführung eines Attributs
  - Vorbereitender Schritt für Codegenerierung, der auch automatisiert werden kann!

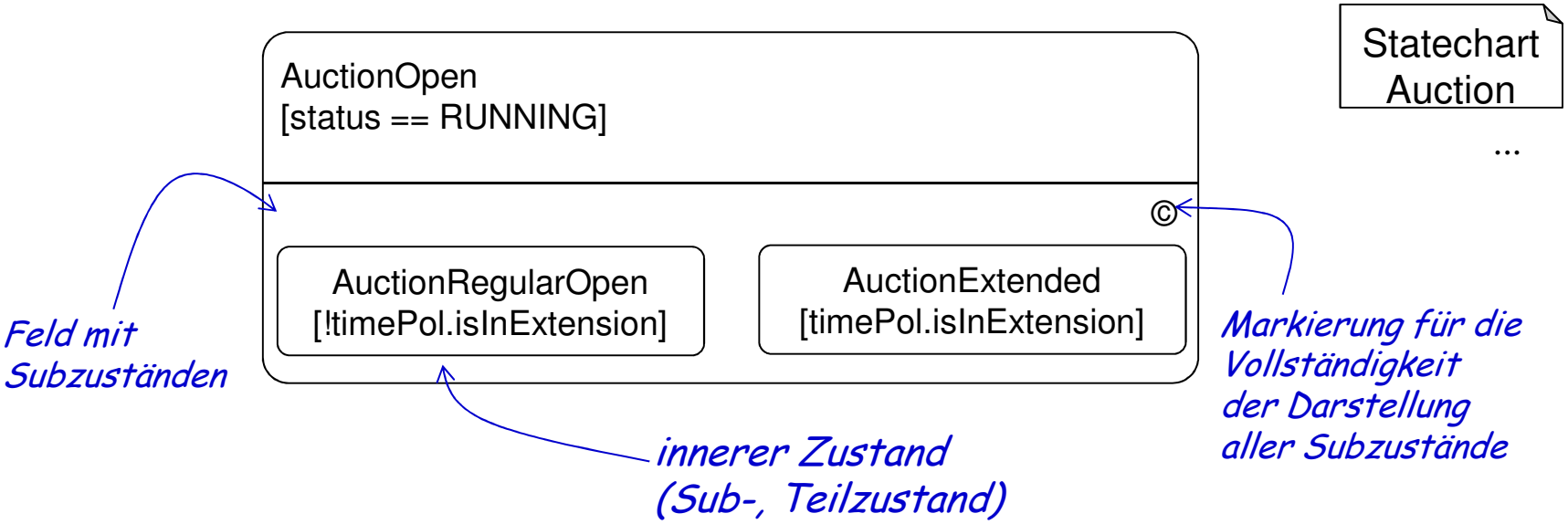
Invariante definiert den Datenzustand



- Normalerweise Invariante charakterisiert Objektzustände:
  - Personen mit rating >= 4500 können VIP sein (müssen nicht!)
- «statedefining»-Zustandsinvarianten definieren Zustand:
  - Personen sind im Zustand “BadPerson” genau dann, wenn rating < 0
- «statedefining»-Zustandsinvarianten müssen disjunkt sein.

Hierarchische Zustände

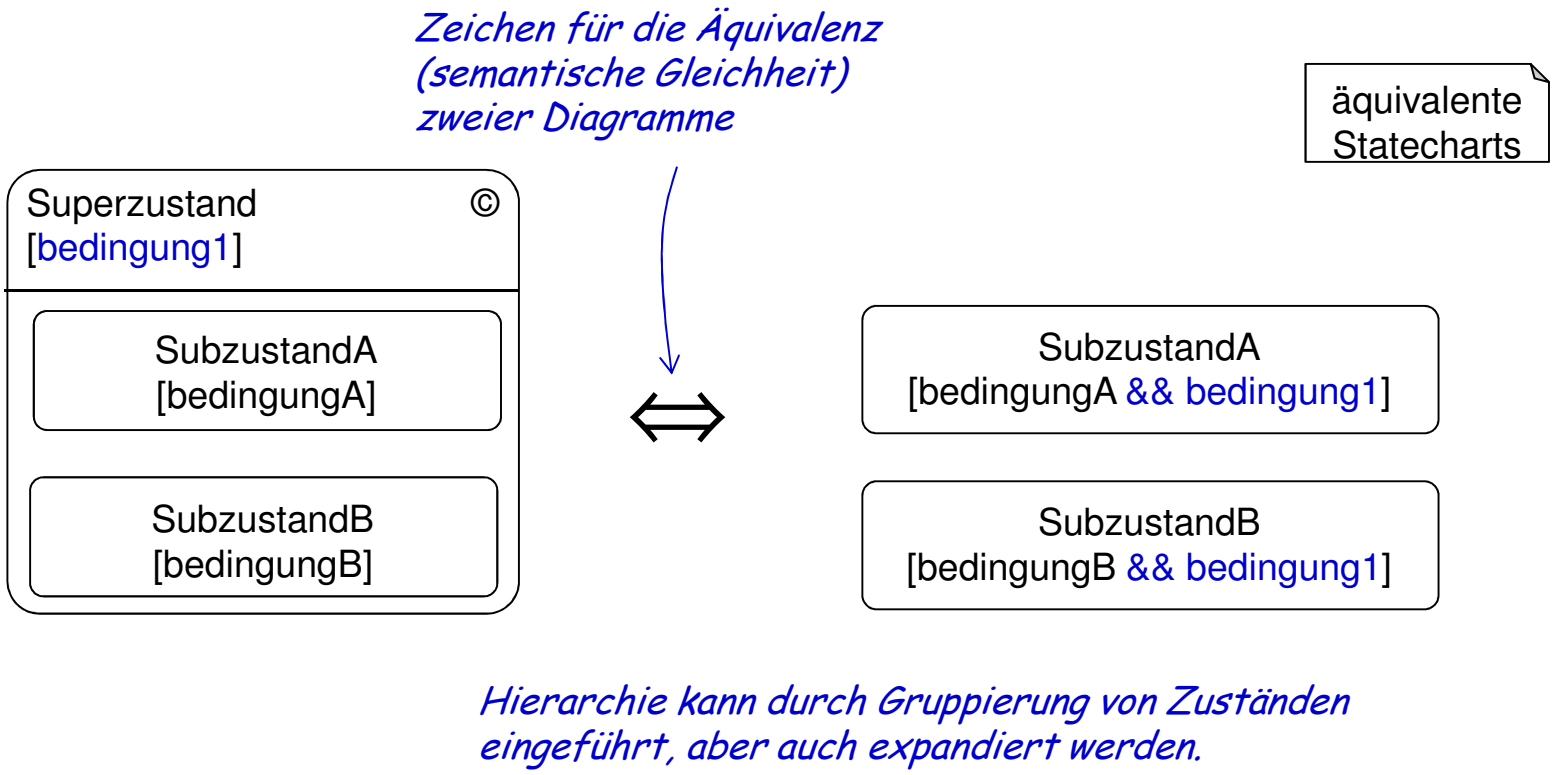
- Hierarchie zur strukturierten Darstellung von Zuständen
  - Subzustände mit gemeinsamen Merkmalen (Invarianten, Aktionen, Transitionen)



- UML/P bietet nur Oder-Dekomposition
  - „Oder-Dekomposition“ = Objekt ist genau in einem Subzustand
  - „Und“ würde ein Kreuzprodukt bedeuten und modelliert meist mehrere Objekte

Semantik hierarchischer Zustände

- Semantik-Erklärung durch Zurückführen auf bekannte Konzepte:
  - Transformation auf flache Zustände:



Start- und Endzustände

- Startzustand: So wird Objekt initialisiert
- Endzustand: Hier darf es Leben beenden. (vs. Garbage Collector?)
- Start- und Endzustand kann identisch sein

- Markierungen sind keine eigenen Zustände!
- Markierungen in Subzuständen haben andere Bedeutung
  (-> nächster Abschnitt!)

Begriffbildung für Zustände

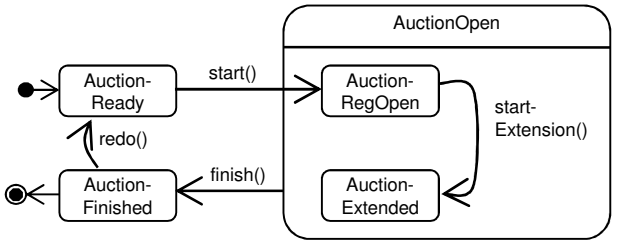
- Zustand (syn. Diagrammzustand)
  - repräsentiert eine Teilmenge der möglichen Objektzustände.
- Startzustand
  - markiert den Beginn des Lebenszyklus.
- Endzustand
  - beschreibt, dass das Objekt in diesem Zustand seine Pflicht erfüllt hat und nicht mehr gebraucht wird. Endzustände können wieder verlassen werden.
- Teilzustand (syn. Subzustand)
  - ist ein Teil eines hierarchisch geschachtelten Zustands.
- Zustandsinvariante
  - ist eine OCL-Bedingung, die für einen Diagrammzustand charakterisiert, welche Objektzustände ihm zugeordnet sind. Zustandsinvarianten verschiedener Zustände dürfen im Allgemeinen überlappen.
- (im später diskutierten Methodenstatechart werden Start-/Endzustände alternativ interpretiert)

SE SOFTWARE ENGINEERING

RWTH AACHEN

Statechart

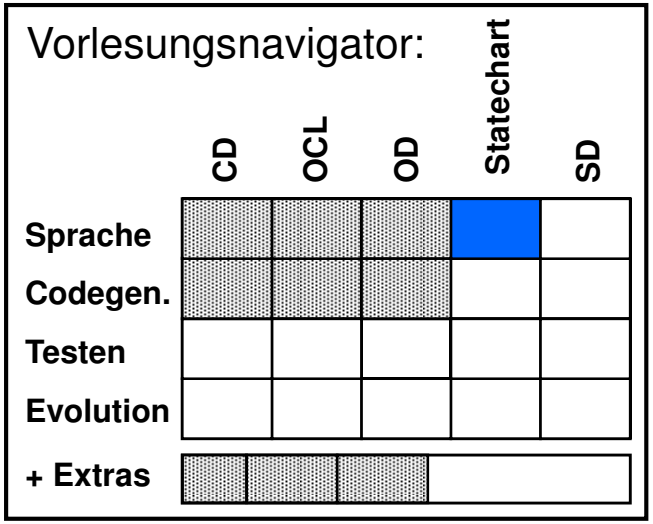
Modellbasierte Softwareentwicklung



- 5. Statecharts
- 5.3. Transitionen

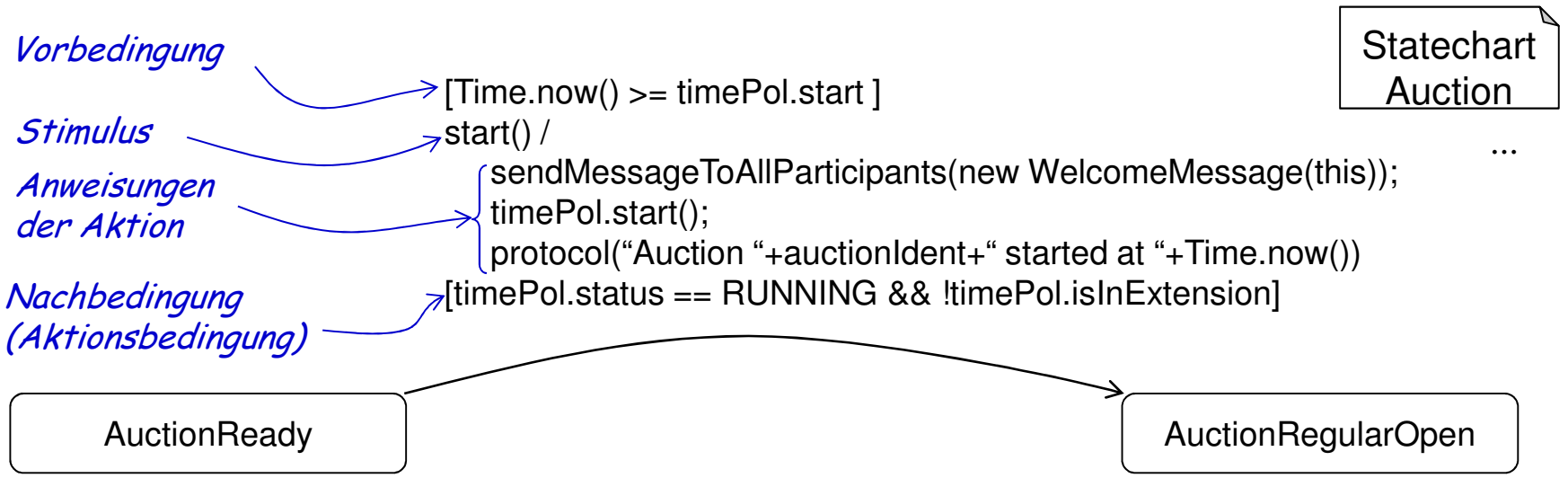
Prof. Dr. Bernhard Rumpe
Lehrstuhl für Software Engineering
RWTH Aachen

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



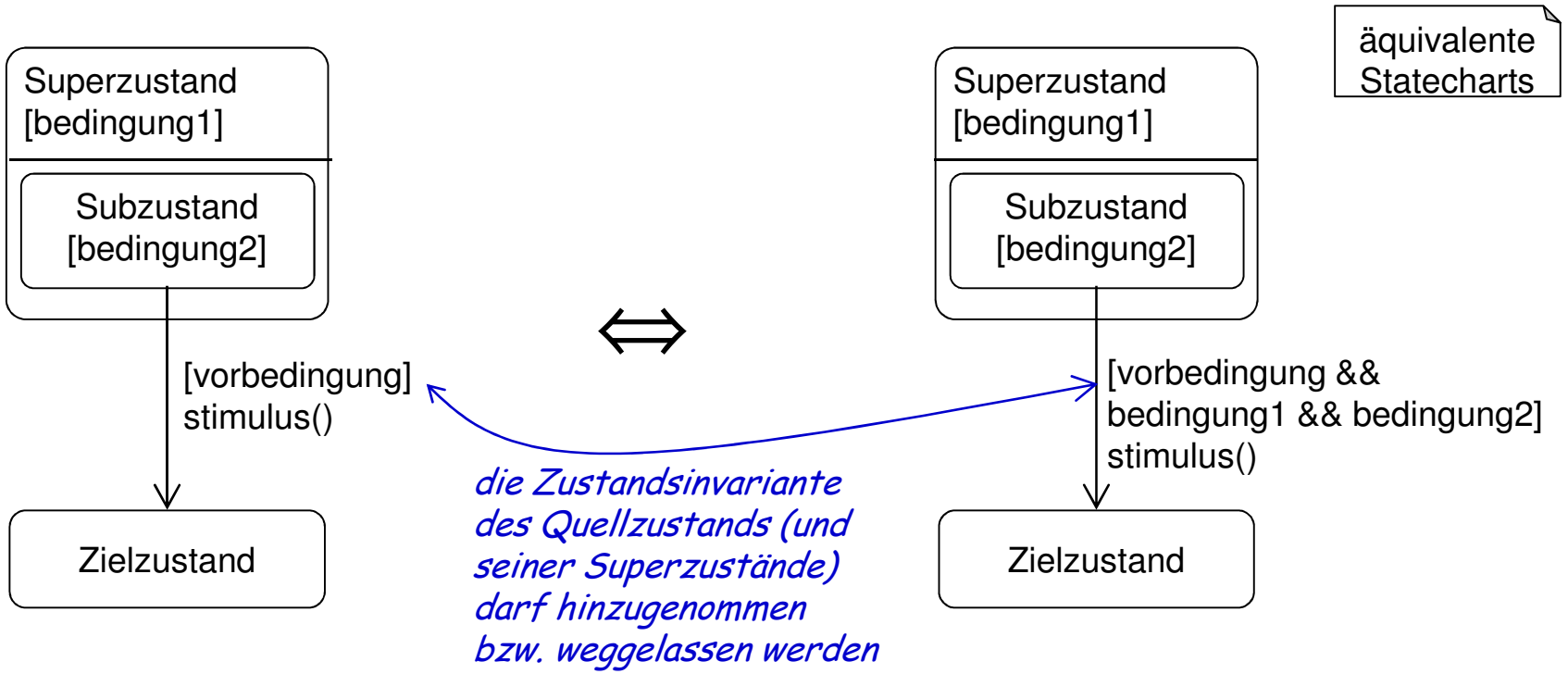
Transitionen

- Eine Transition beschreibt einen Ausschnitt eines Objektverhaltens
- Eine Transition besitzt
  - Quellzustand
  - Vorbedingung
  - Stimulus
  - Aktion (-> nächster Abschnitt)
  - Nachbedingung
  - Zielzustand



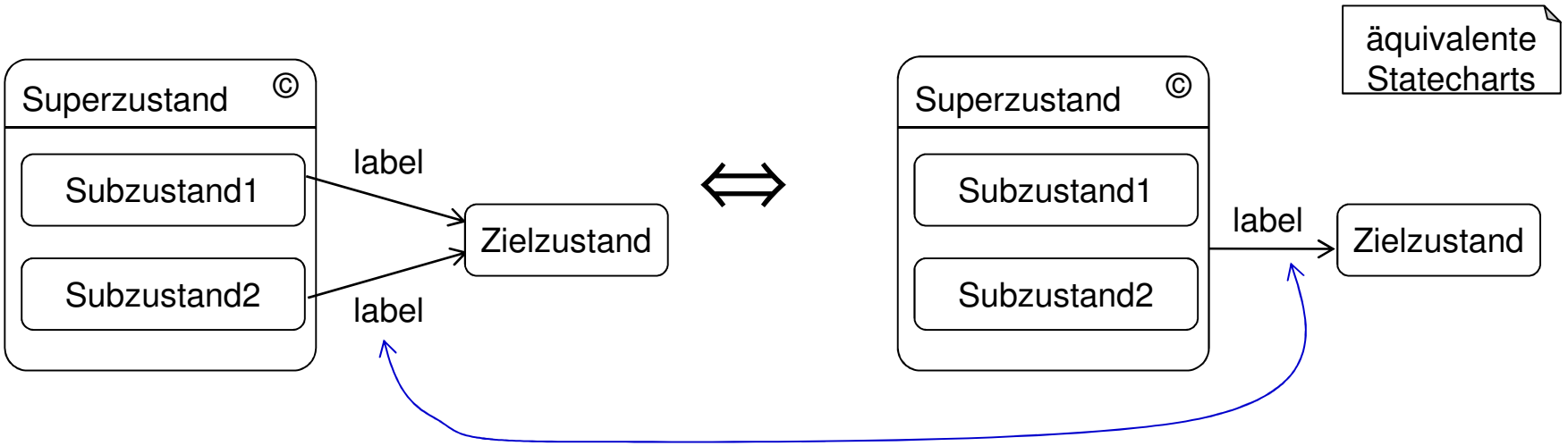
Vorbedingungen in Transitionen

- Vorbedingung der Transition und Zustandsinvariante bestimmen die Schaltbereitschaft
- Expansion oder Faktorisierung der Zustandsinvariante



Superzustand als Quelle

- Ist die Transitionsquelle ein Superzustand, so geht die Transition von jedem Subzustand aus:

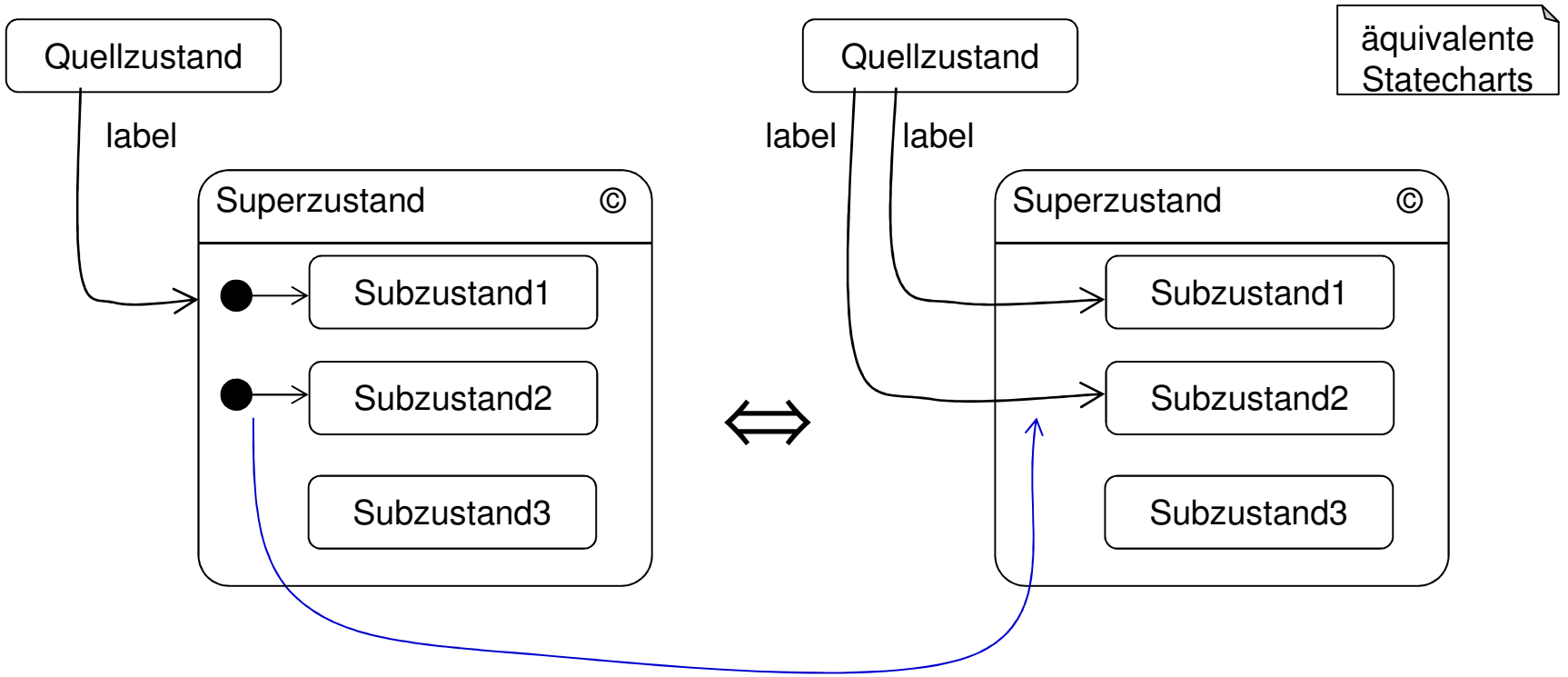


existieren gleichlautende Transitionen von allen Subzuständen, so können diese durch eine Transition vom Superzustand ersetzt werden, wenn die Liste der Subzustände vollständig ist (©)

- Sonderfall: Subzustände haben Start/Ende-Markierungen

Superzustand als Ziel

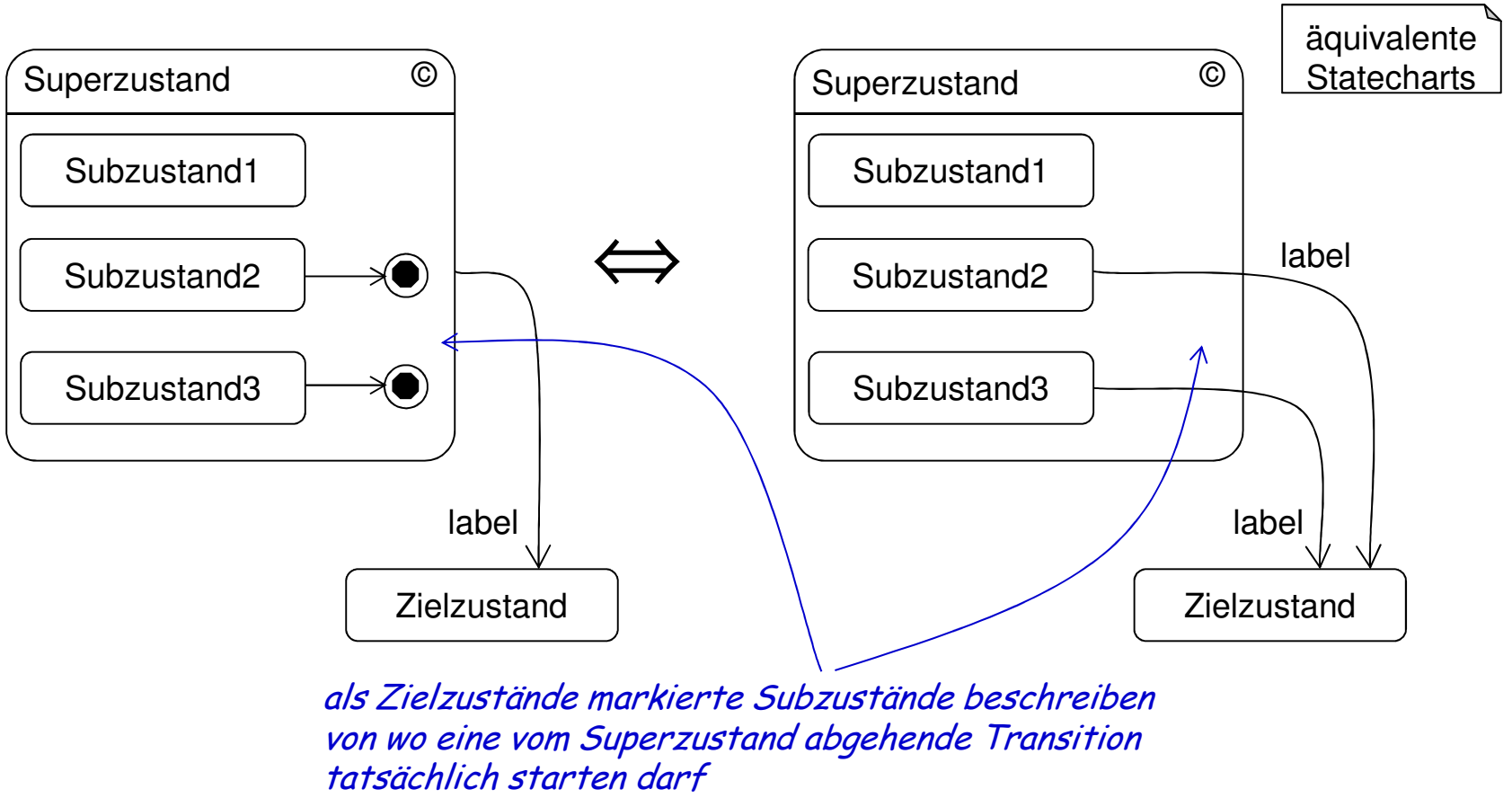
- Ist das Transitionsziel ein Superzustand, so geht die Transition zu jedem darin befindlichen Subzustand aus, der als Start markiert ist:



als Startzustände markierte Subzustände dienen dazu, das Ziel ankommender Transitionen zu präzisieren

Superzustand als Quelle (2)

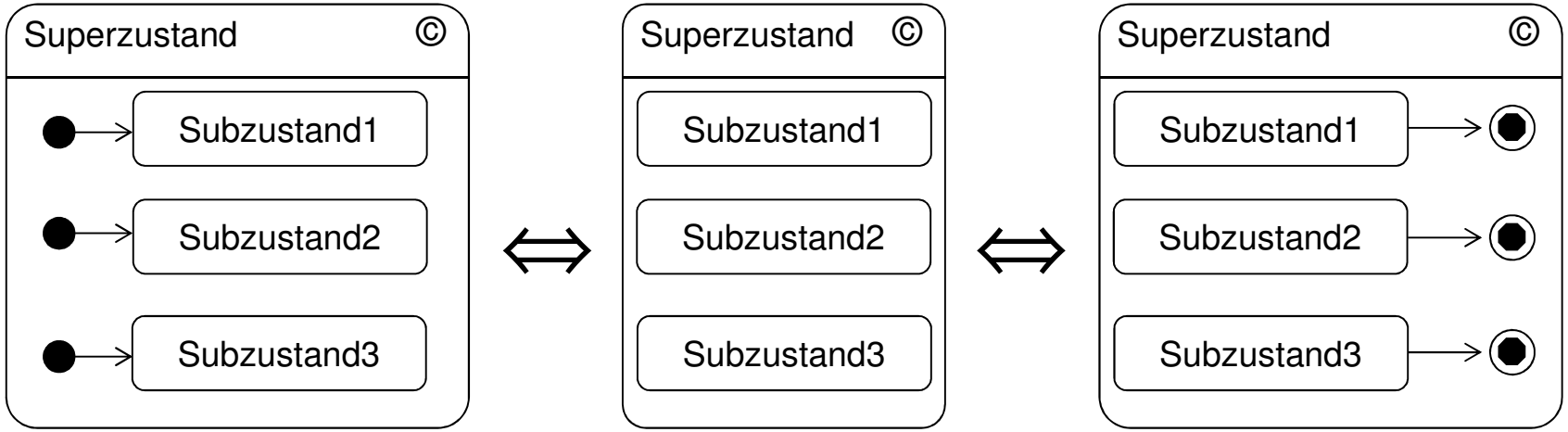
- Ist die Transitionsquelle ein Superzustand, so geht die Transition von jedem Subzustand aus, der als Ende markiert ist:



Fehlende Start- / Endzustände

- Ist kein Subzustand markiert, so gelten alle als implizit markiert:
- (Prinzip der Unterspezifikation: Wo keine Aussage getroffen wurde, ist nichts bekannt!)

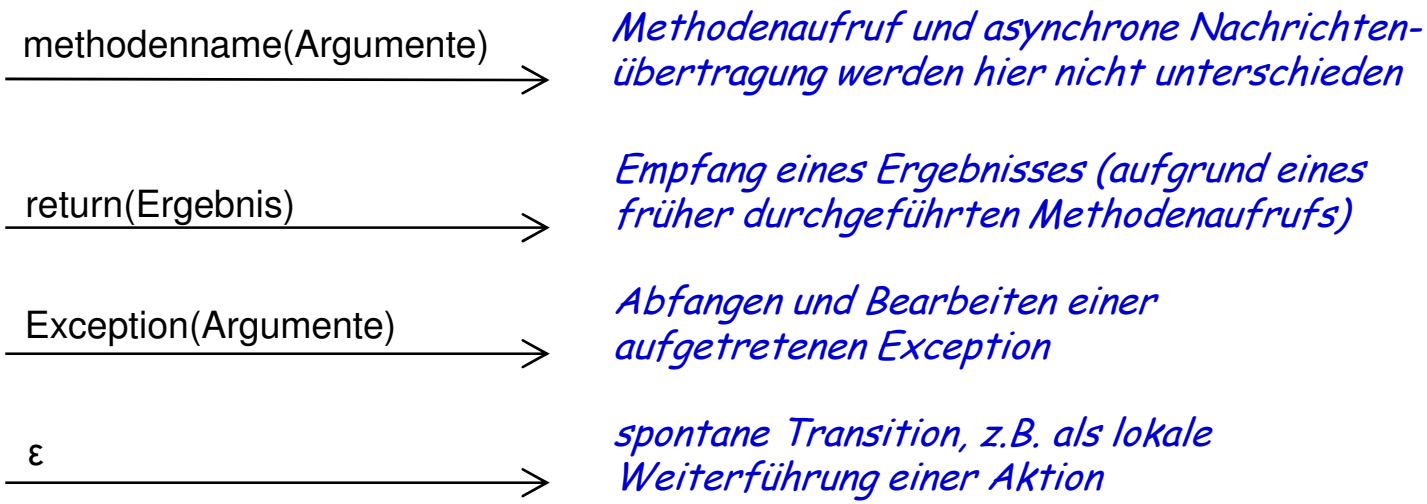
äquivalente Statecharts



- Mit den bisher angegebenen Transformationen lassen sich Transitionen immer auf innere Zustände umbauen
  - hierarchische Zustände werden dadurch überflüssig

Arten von Stimuli in Transitionen

- Allgemeine Varianten von Stimuli:
  - Nachricht wird empfangen (über einen Kommunikationsweg),
  - Methodenaufruf erfolgt,
  - Ergebnis eines Return-Statements zurückgegeben,
  - Exception wird abgefangen oder
  - Transition tritt spontan auf.
- Nachrichtenempfang wird meist via Methodenaufruf realisiert. Deshalb folgende Darstellung der Stimuli-Arten:

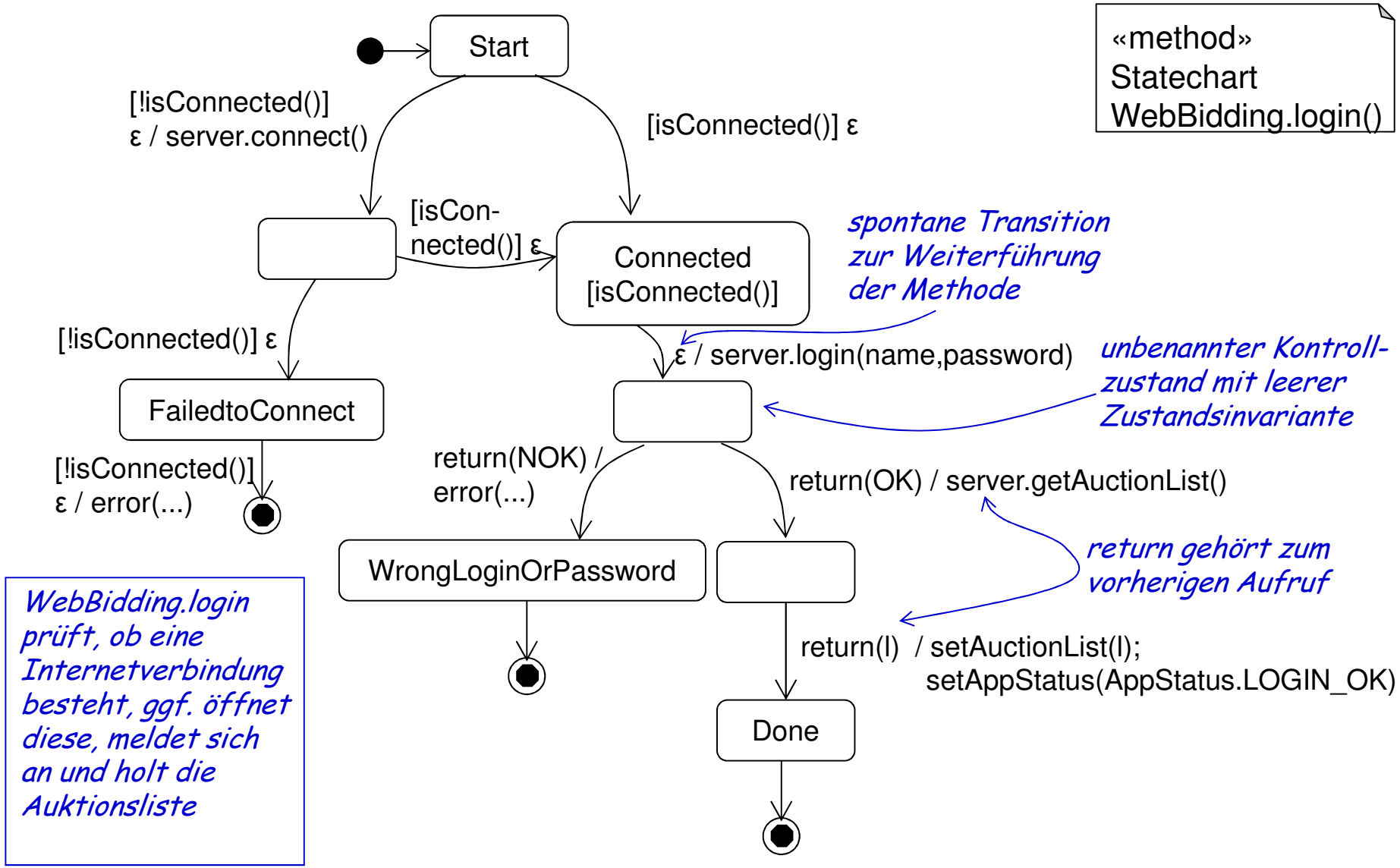


Methodenstatechart

- bisher: Statechart war einer Klasse zugeordnet
- Stimulus war ein Methodenaufruf,
  Aktion die Ausführung der Methode

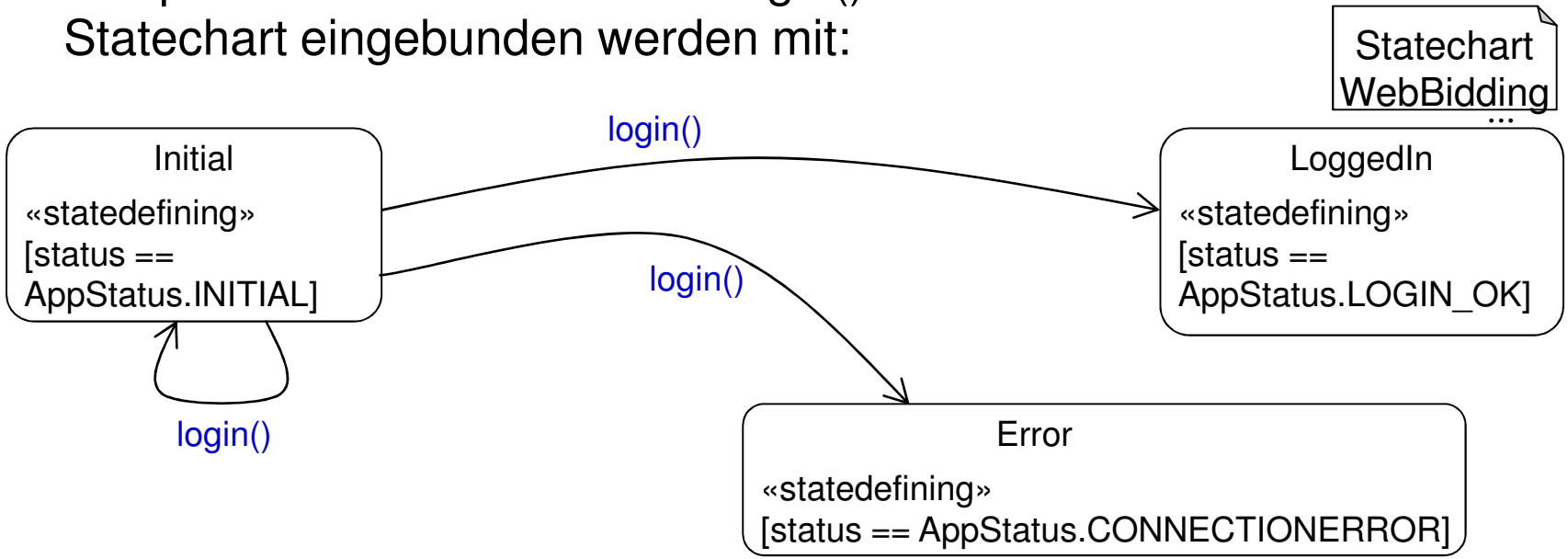
- Statecharts können auch einzelnen Methoden zugeordnet werden durch Stereotyp «method»:
  - Methodenstatechart beschreibt Kontrollfluss der Methode
  - Methodenstatechart besitzt nur Kontrollzustände («controlstate»), die deshalb nicht explizit anzugeben sind
  - Kontrollzustände entsprechen Programmzähler im Code

Beispiel Methodenstatechart



Kombination von Statecharts

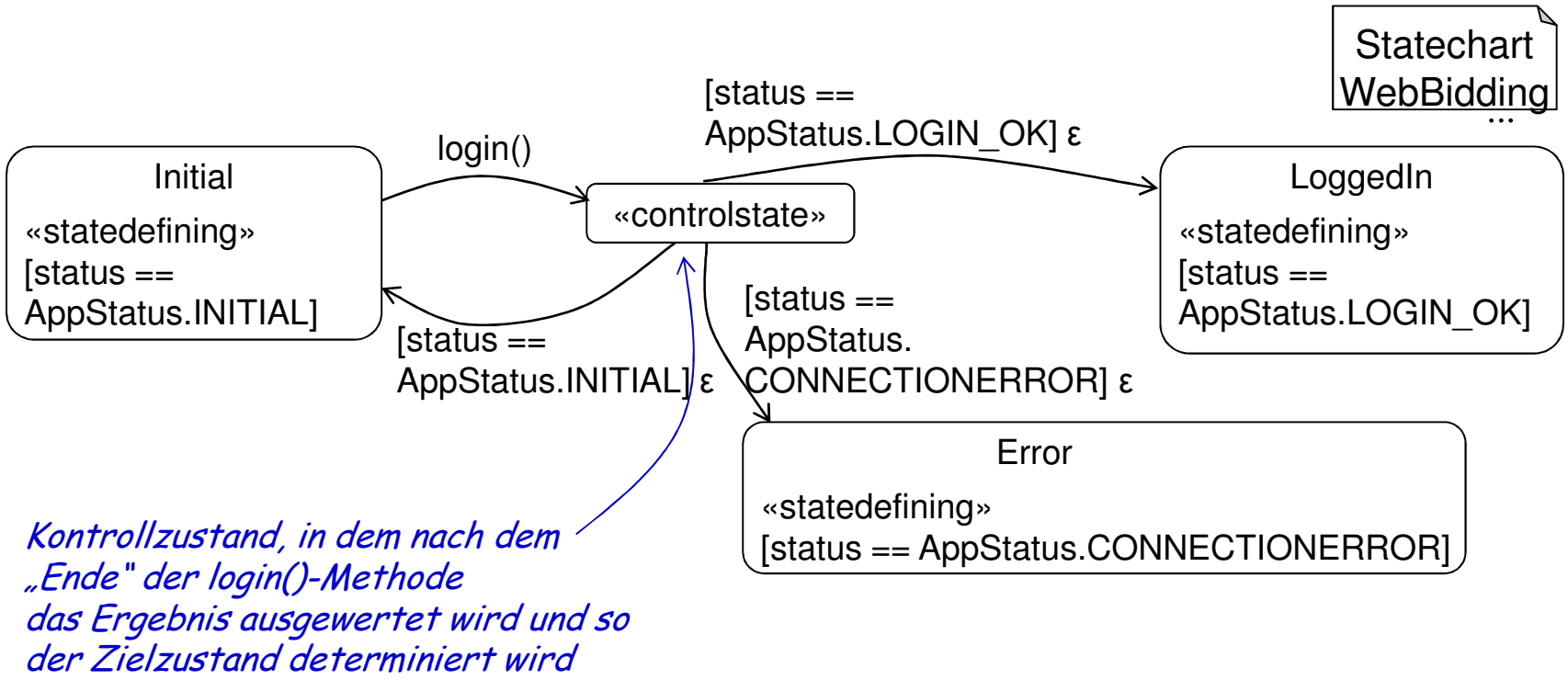
- Typische Vorgehensweise: Ein Top-Level-Statechart für die Klasse
- und einzelne Statecharts für komplexe Methoden.
- Beispiel: Vorher beschriebene login()-Methode kann im Klassen-Statechart eingebunden werden mit:



- Kompatibilität beider Statecharts gefordert: z.B. beim Zielzustand
- Meist kann nur ein Statechart konstruktiv eingesetzt werden, das andere dient dann zur Testfall-Bestimmung

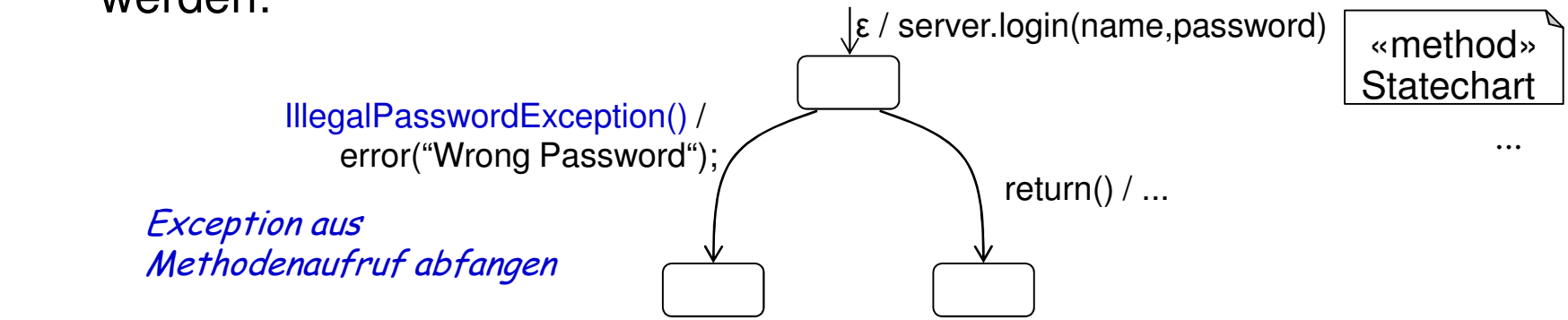
Komposition von Statecharts: Kontrollzustand zur Bestimmung des Zielzustands

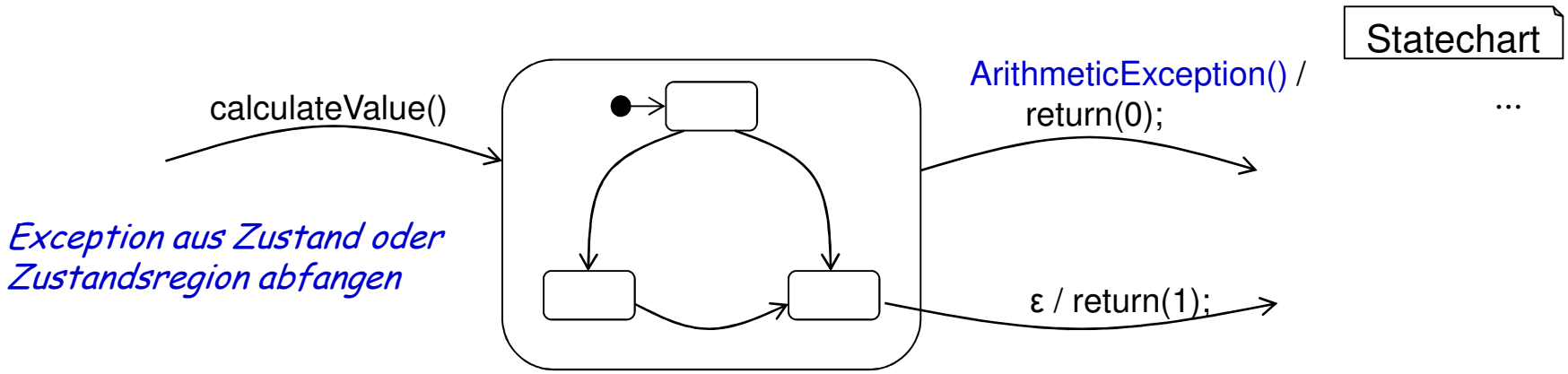
- Obiges Statechart kann verfeinert werden, um Zielzustand zu bestimmen:



Exception als Stimulus

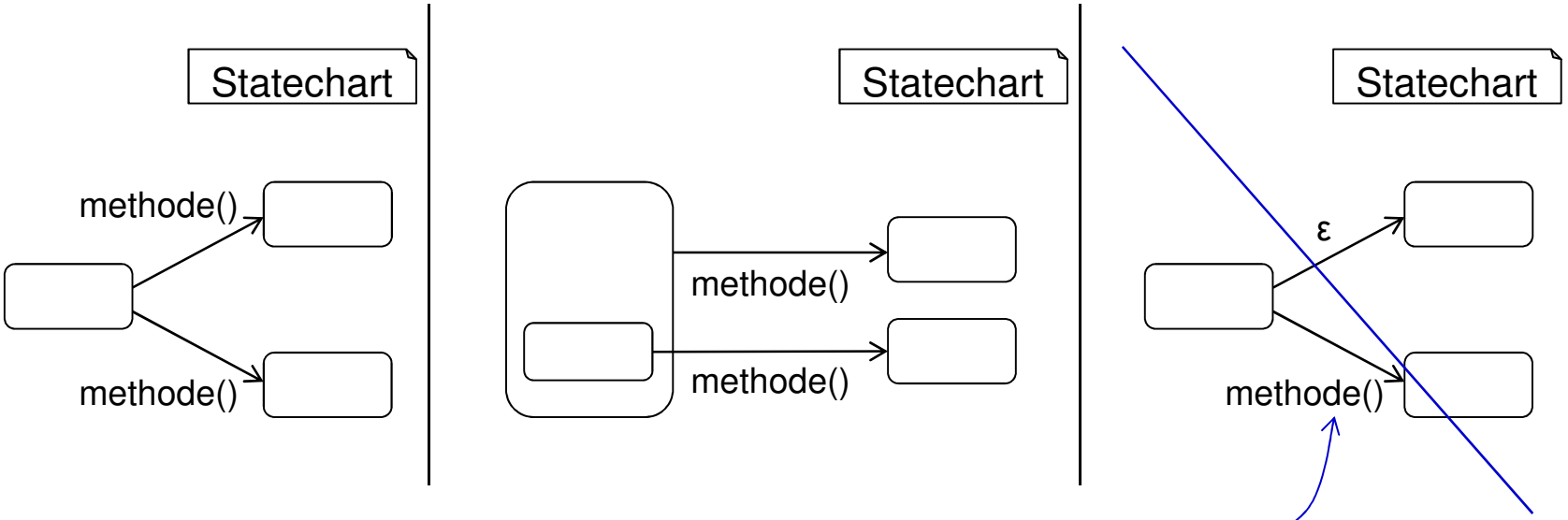
- Damit können anomale Terminierungen von Aufrufen abgefangen werden:





Überlappende Schaltbereitschaft

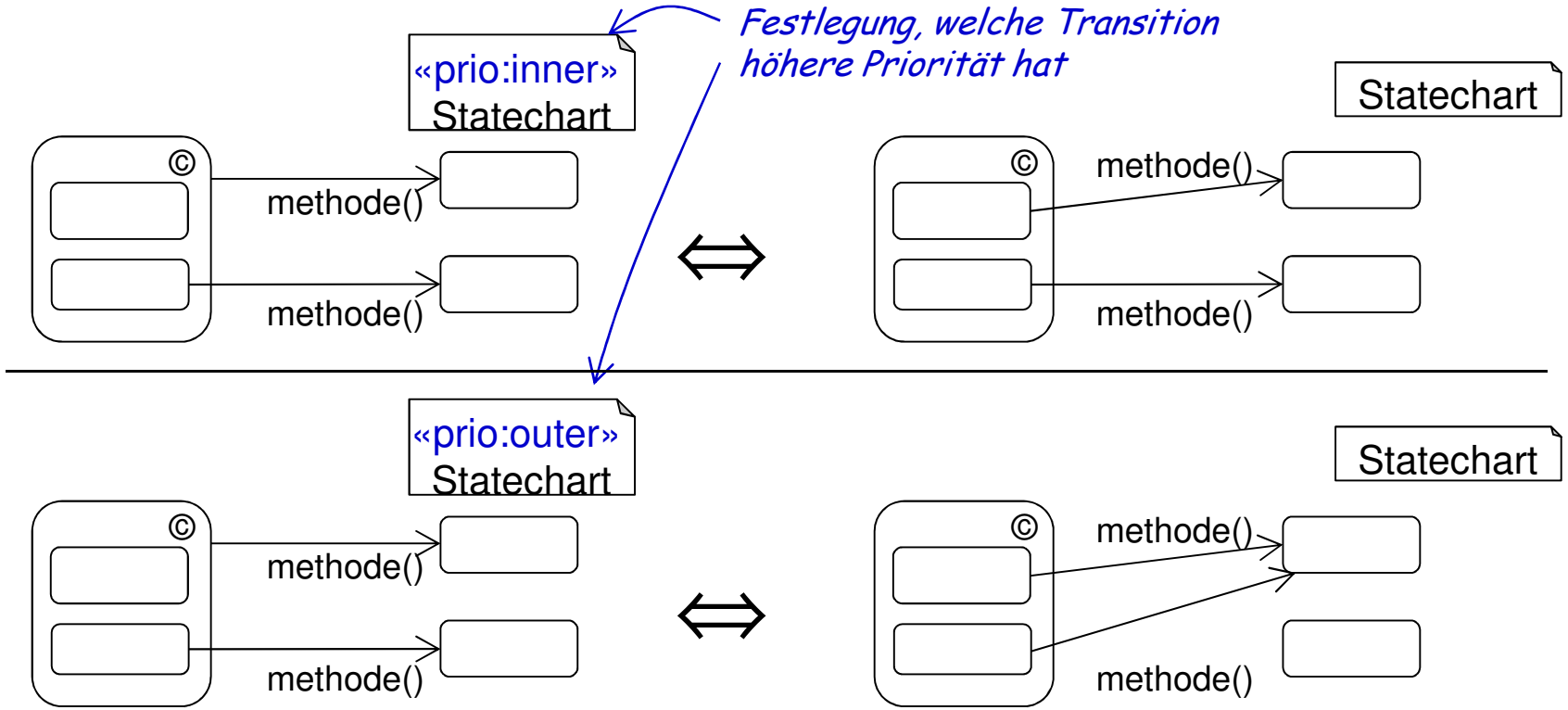
- = Nichtdeterminismus im Statechart
- = Interpretation durch Unterspezifikation:
  - Entwickler oder Implementierung darf auswählen
- Beispiele



Variante ist nicht erlaubt, da methodeninterne Weiterführung und neuer Methodenaufruf in einem Zustand gemischt sind

Priorisierung von Transitionen

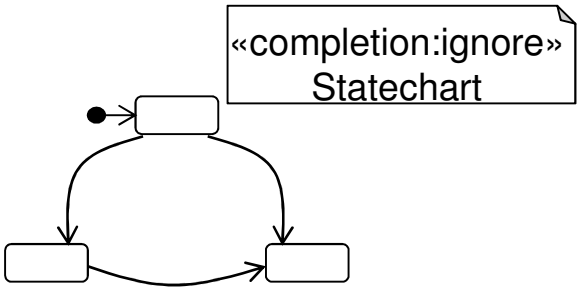
- Überlappung kann durch Priorisierung der Transitionen aufgelöst werden
- Statechart-Varianten priorisieren innere bzw. äußere Transitionen
- UML/P lässt Wahlfreiheit durch Stereotypen «prio:inner» und «prio:outer»

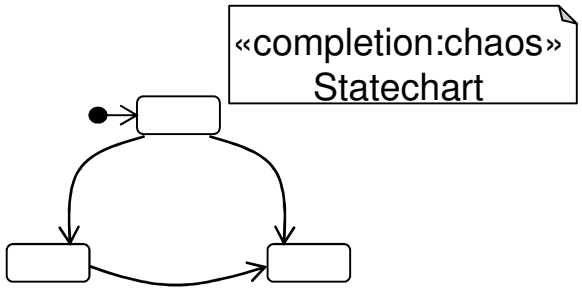


Unvollständiges Statechart

- Keine schaltbereite Transition vorhanden:
- Prinzip der Unterspezifikation: Wo keine Aussage getroffen wurde, ist nichts bekannt!
- Um eine Aussage zu treffen, können Stereotypen eingesetzt werden
  - «error» markiert Fehlerzustand, der dann angesprungen wird.
  - «exception» markiert einen Zustand, in dem auftretende Exceptions abgefangen werden

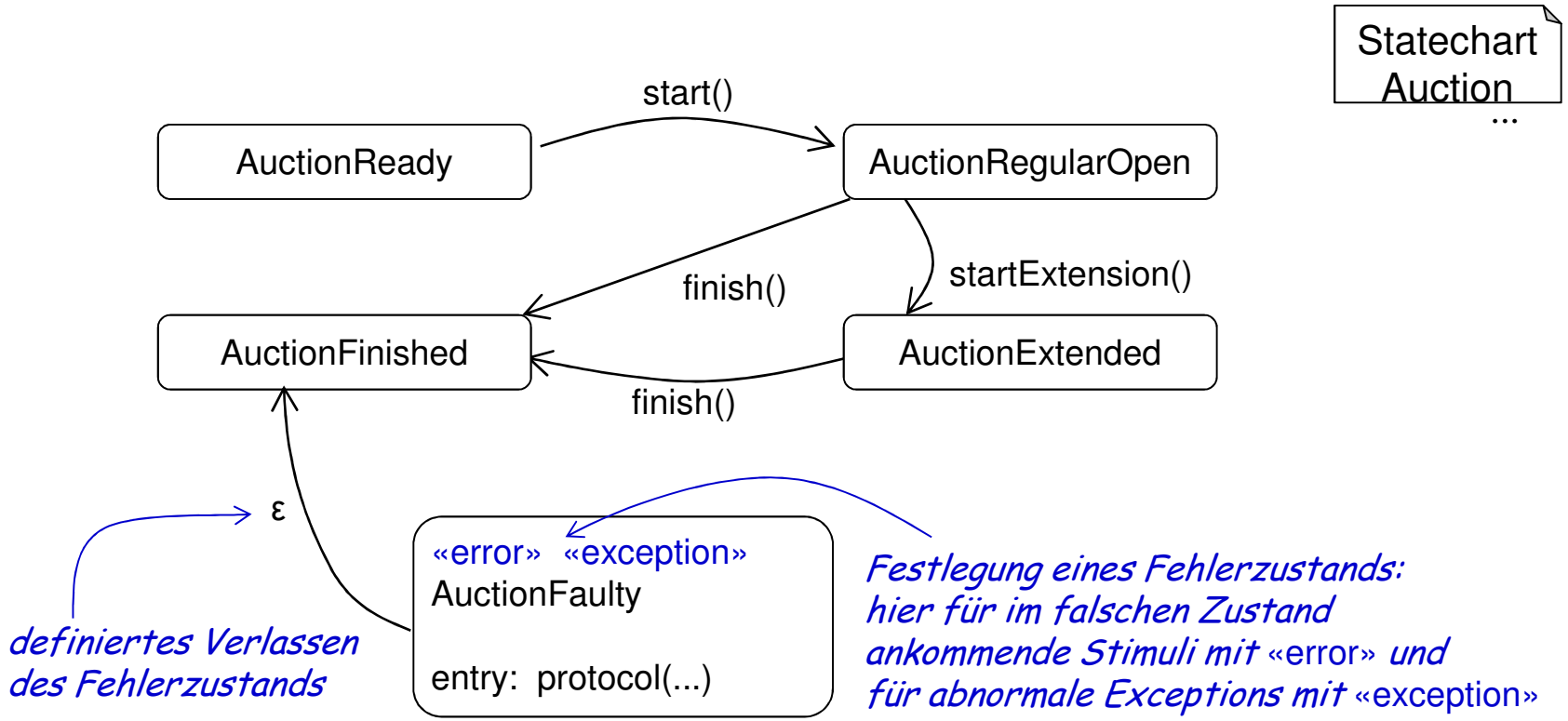
  - «completion:ignore» besagt, Aufruf wird ignoriert
  - «completion:chaos» besagt, Aufruf kann beliebig behandelt werden (Default)





Beispiel: Fehlervervollständigung

- Fehlervervollständigung durch Markierung eines expliziten Fehlerzustands mit «error»
- Exceptions können separat abgefangen werden: «exception»



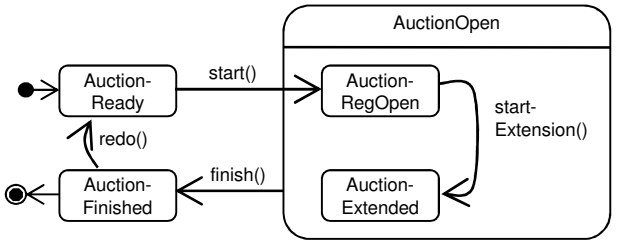
Begriffbildung für Transitionen

- Stimulus
  - von anderen Objekten verursacht und führt zur Ausführung einer Transition.
  - Stimulusarten: Methodenaufruf, RPC, Empfang asynchron versandter Nachricht oder Timeout.
- Transition
  - von Quellzustand in Zielzustand, beinhaltet einen Stimulus und eine Aktion als Reaktion. OCL-Bedingungen präzisieren die Transition.
- Schaltbereitschaft:
  - Transition ist genau dann schaltbereit, wenn Objekt im Quellzustand der Transition und Stimulus korrekt sind sowie Vorbedingung zutrifft.
  - Sind mehrere Transitionen schaltbereit, so ist das Statechart nichtdeterministisch und ausgewählte Transition ist nicht festgelegt.
- Vorbedingung der Transition:
  - OCL-Bedingung, die für die Attributwerte und den Stimulus erfüllt sein muss.
- Nachbedingung der Transition (syn. Aktionsbedingung):
  - OCL-Bedingung beschreibt Eigenschaften der Reaktion.

SE SOFTWARE ENGINEERING

RWTH AACHEN

Statechart

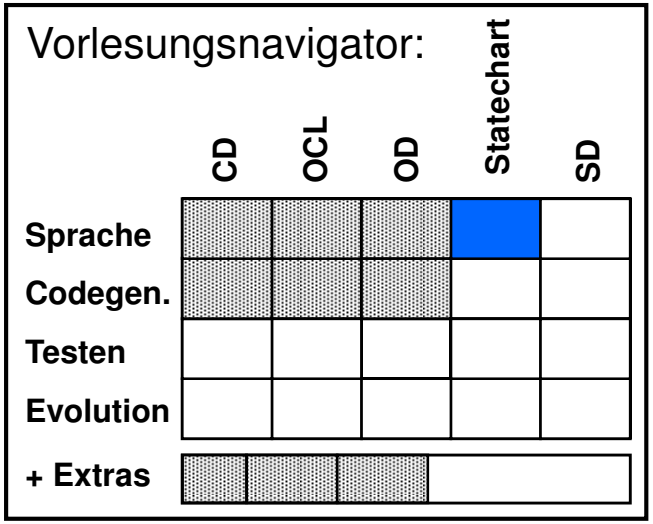


Modellbasierte Softwareentwicklung

- 5. Statecharts
- 5.4. Aktionen

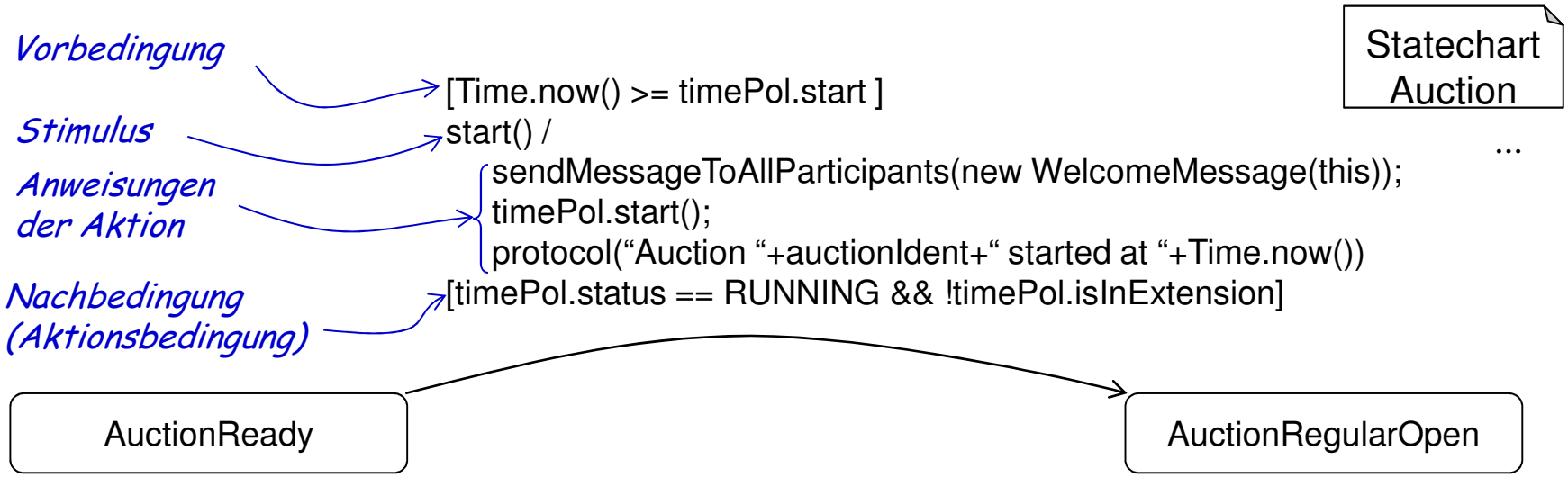
Prof. Dr. Bernhard Rumpe
Lehrstuhl für Software Engineering
RWTH Aachen

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

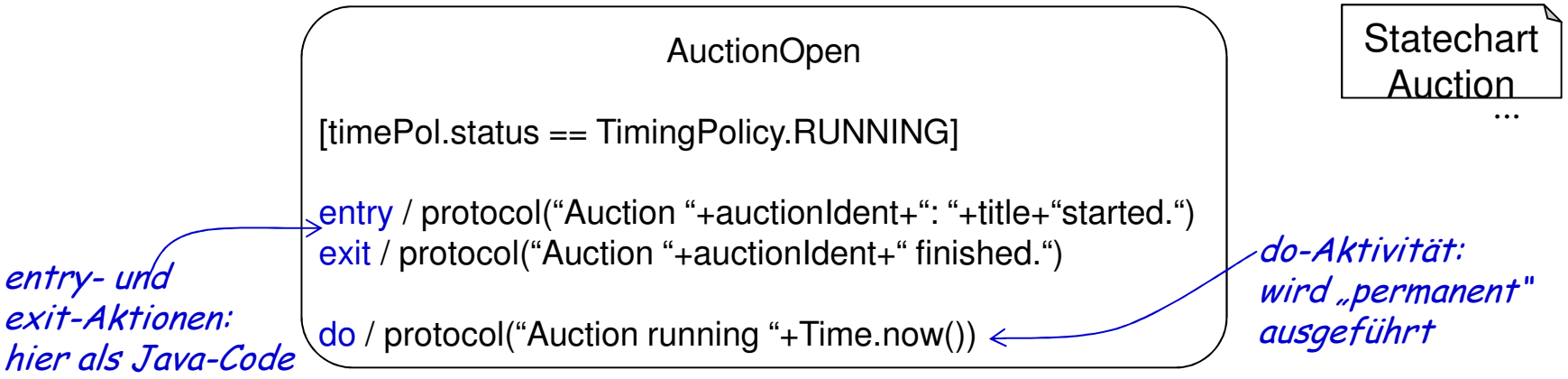


Aktion in Transitionen

- Ausgabe des Mealy-Automaten = Aktion im Statechart
  - Aktionen verändern Objektzustände
  - versenden Nachrichten / rufen Methoden auf
- Aktionsdarstellung in zwei Formen:
  - Operationell:
    - konkrete Anweisungen z.B. in Java
  - Beschreibend:
    - Aktionsbedingung = Nachbedingung der Transition
    - Effekt z.B. durch OCL festgelegt



Aktionen in Zuständen

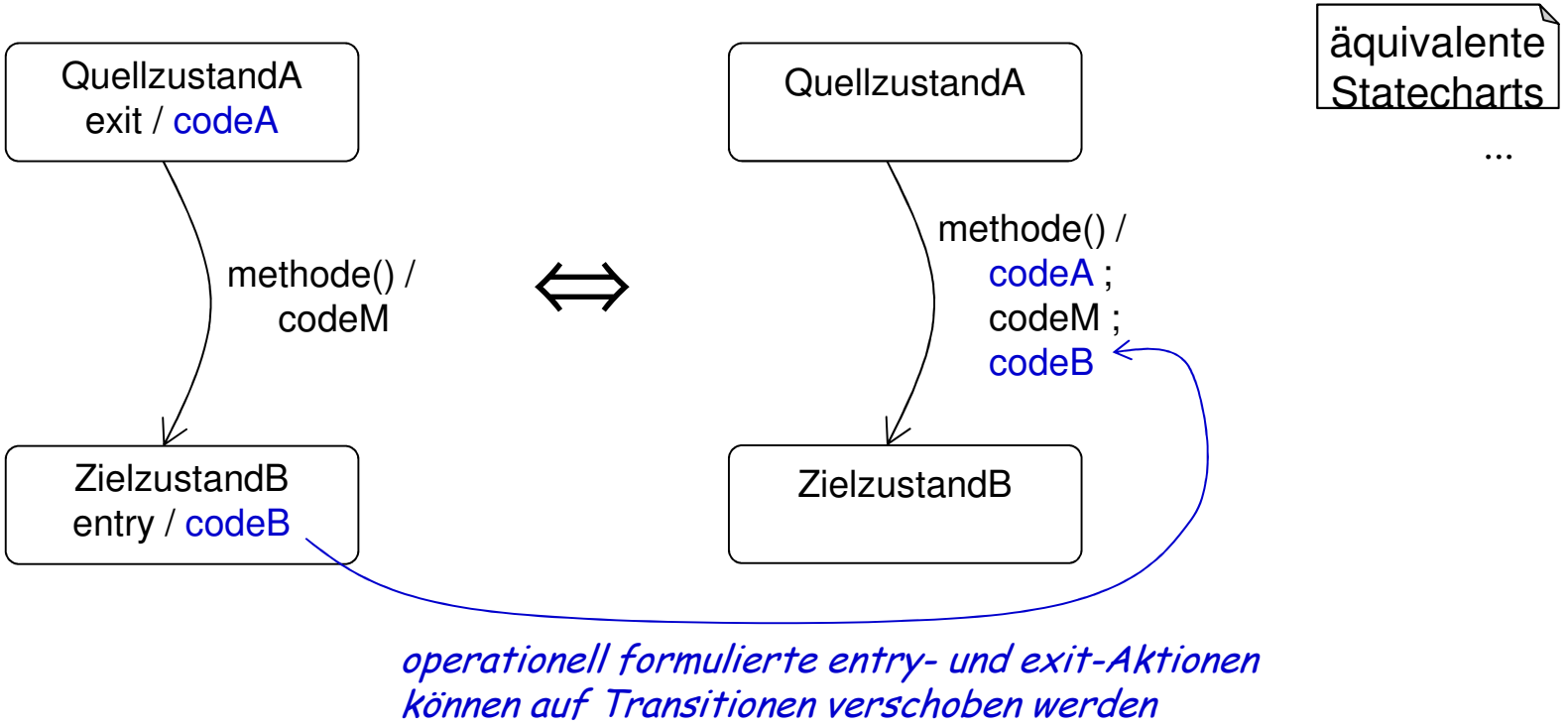


- entry- Aktion: Wird bei Betreten des Zustands ausgeführt
- exit-Aktion: beim Verlassen
- do-Aktivität: regelmäßig zwischendurch

- entry- und exit-Aktionen erweitern Statecharts zu Moore-Automaten mit zustandsbezogener Ausgabe

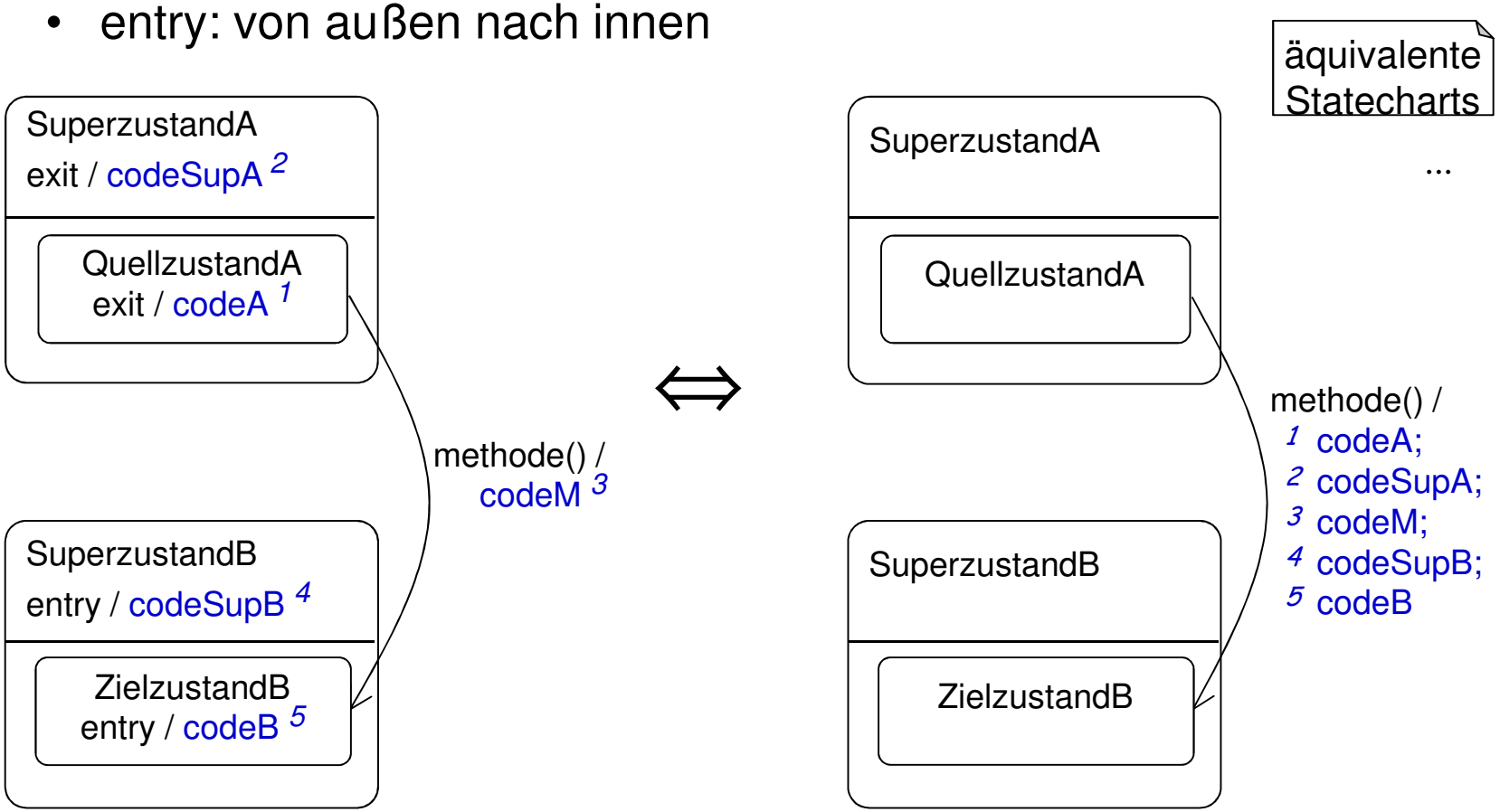
Semantik von entry-/exit-Aktionen

- Moore- können in Mealy-Automaten transformiert werden:
  - Verschieben der Zustands-Aktionen in angrenzende Transitionen
- Einfacher Fall für operationelle Aktionen:
  - Sequentielle Komposition (mit „ ; “)



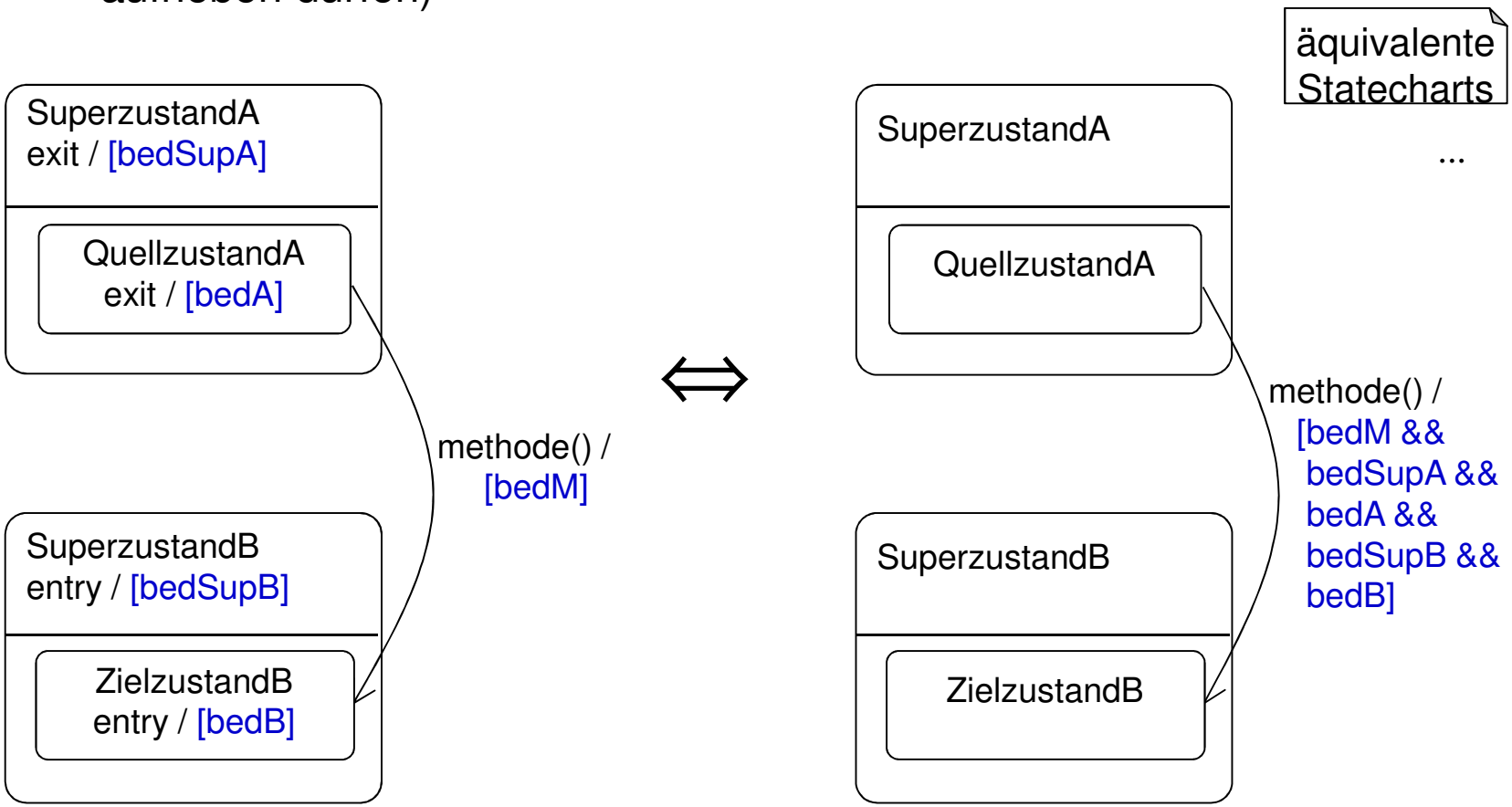
Zusammenspiel der entry-/ exit-Aktionen in der Hierarchie: operationell

- operationelle entry- und exit-Aktionen werden in der Reihenfolge des Verlassens und Betretens von Zuständen ausgeführt
  - exit: von innen nach außen
  - entry: von außen nach innen



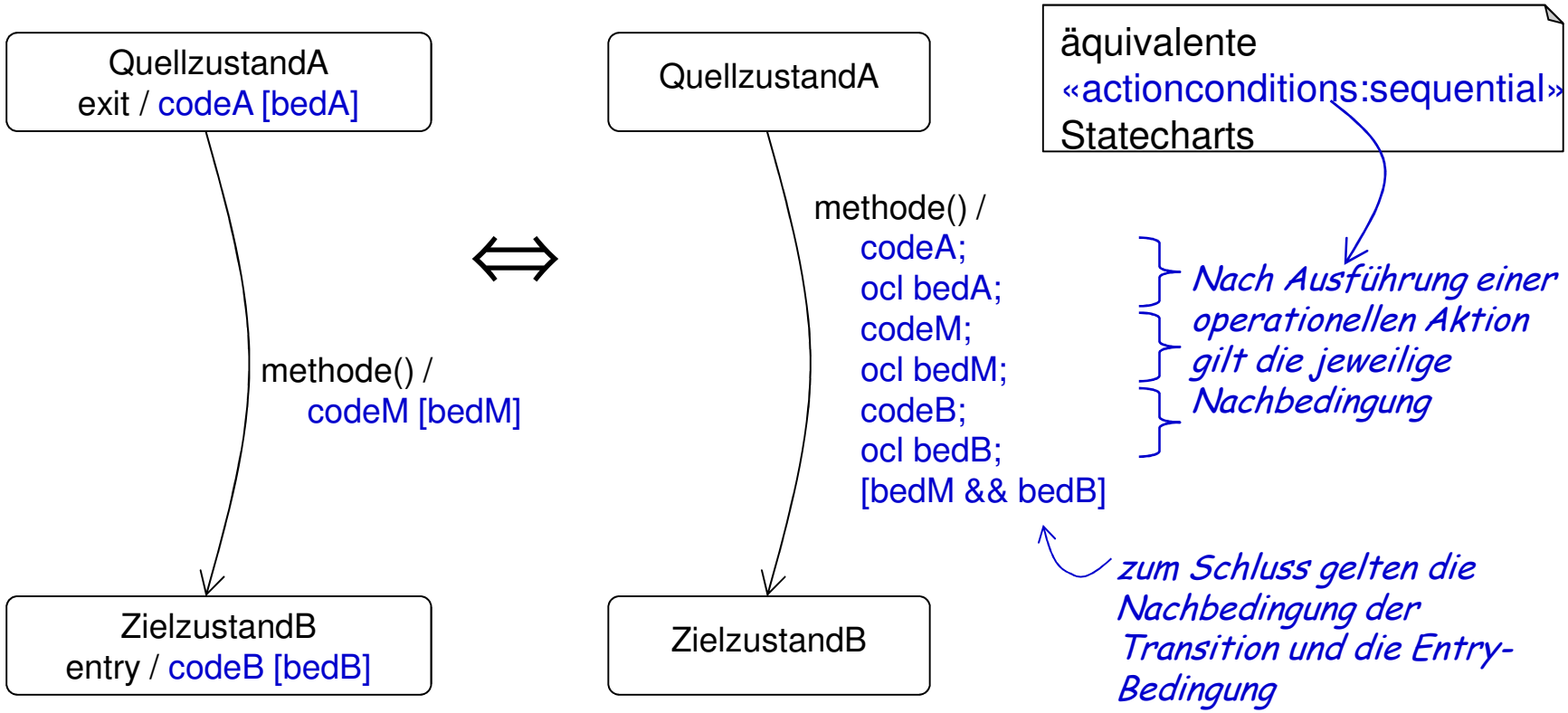
Zusammenspiel der entry- / exit-Aktionen in der Hierarchie: mit Bedingungen

- Entry- und exit-Aktionen als Bedingungen werden konjungiert (&&)
  - nach Transitionsausführen gelten alle Bedingungen gleichzeitig!
  - (anders als bei den operationellen Aktionen, die ihre Effekte gegenseitig aufheben dürfen)



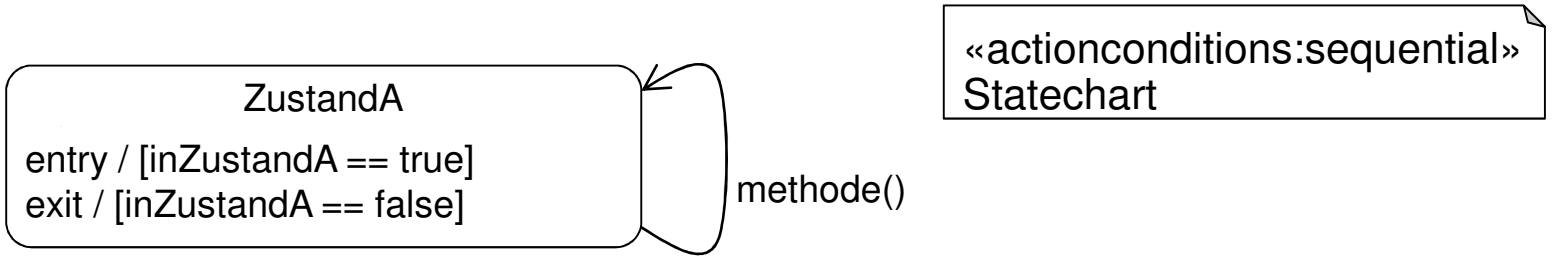
Zusammenspiel der entry- / exit-Aktionen in der Hierarchie: Mischform

- sind Bedingungen und Code-Aktionen angegeben, so ist klarzustellen, wann welche Bedingungen gelten.
- Stereotyp «actionconditions:sequential» führt zu abschnittsweiser Gültigkeit der Bedingungen



Anwendungsbeispiel «actionconditions:sequential»

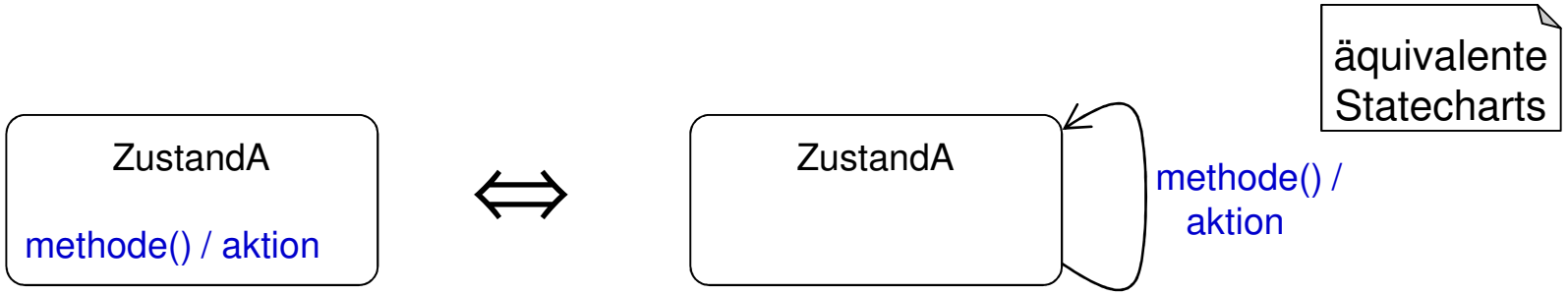
- Transitionsschleifen mit sich widersprechenden entry- / exit-Bedingungen können nur sequentiell aufgefasst werden:



- Entry- und exit-Aktion widersprechen sich:
  - Bei Nutzung einer Konjunktion (&&) wäre Nachbedingung false und damit die Transition nicht realisierbar!

Interne Transition

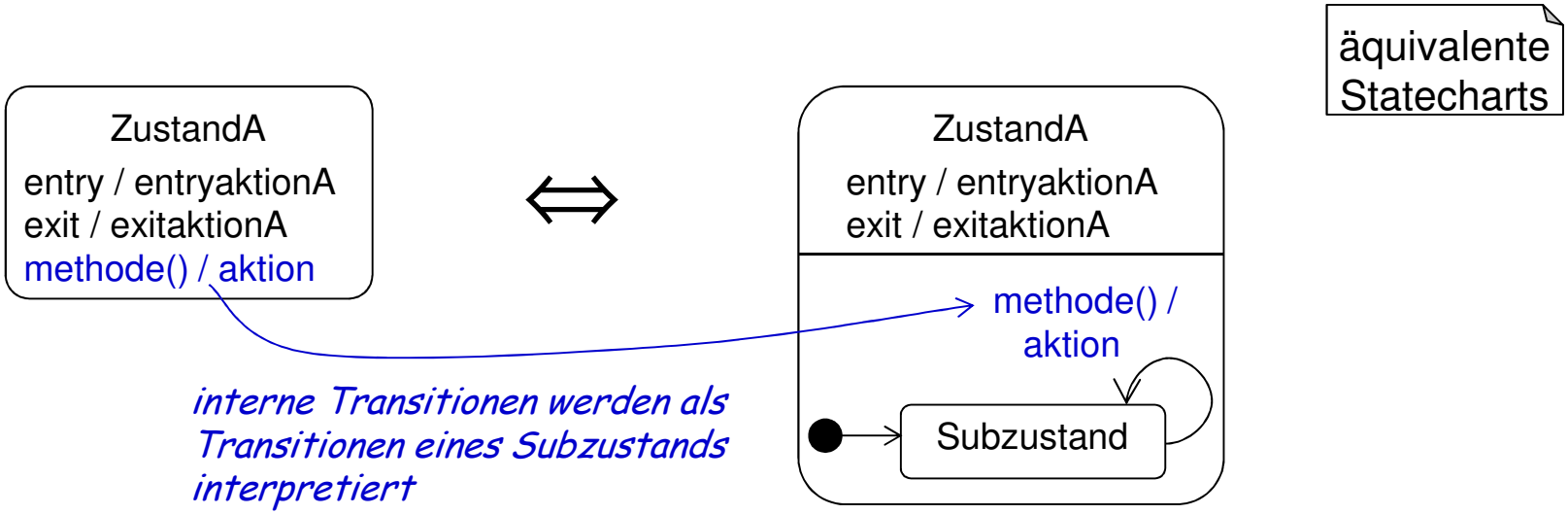
- Eine Transition kann in einem Zustand angegeben werden:
  - Interne Transitionen bilden eine alternative Darstellung für eine Schleife dieses Zustands:



- Bedingung: Zustand hat keine entry-/exit-Aktionen, denn
  - Entry- oder exit-Aktionen des Zustands werden links nicht ausgeführt, aber rechts schon.
  - Alternative?

Interne Transition

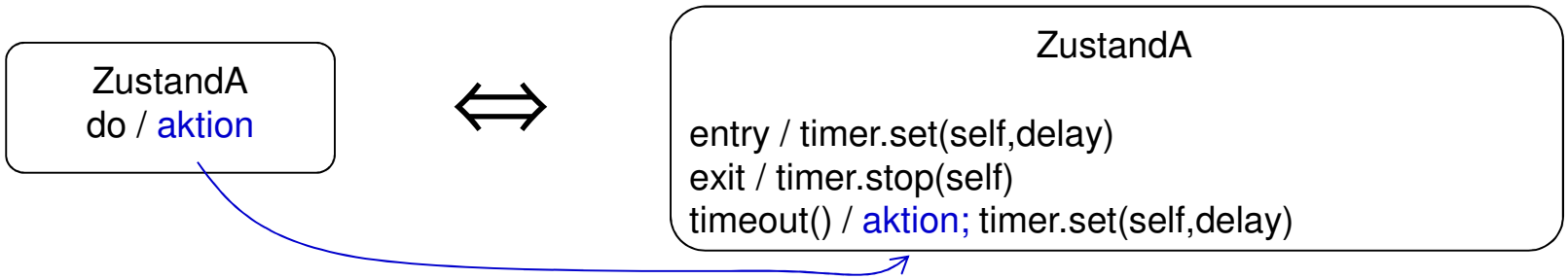
- Interne Transitionen sind formal Transitionen des (einzigen), dafür eingeführten Subzustands:
  - Dabei wird berücksichtigt, dass entry- / exit-Aktionen bei internen Transitionen nicht ausgeführt werden.



Do-Aktivität

- Regelmäßige Ausführung der do-Aktivität eines Zustands bedeutet
  - Externer zeitgesteuerter Mechanismus triggert die enthaltene Aktion regelmäßig
  - Vorschlag der Umsetzung durch Timer und interne Transition:

äquivalente Statecharts
...



eine do-Aktivität wird durch einen Timer regelmäßig ausgeführt

Begriffsbildung für Aktionen

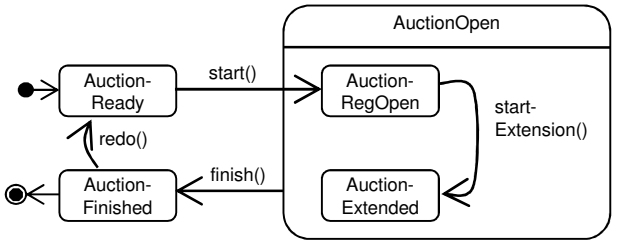
- Aktion
  - ist eine durch operationellen Code (z.B. Java) oder durch OCL-Bedingung beschriebene Veränderung des Zustands von Objekt und Umgebung.
- Entry-Aktion
  - gehört zu einem Zustand und wird bei dessen „Betreten“ ausgeführt bzw. die Bedingung etabliert.
- Exit-Aktion
  - gehört zu einem Zustand und wird bei dessen „Verlassen“ ausgeführt bzw. die Bedingung etabliert.
- Do-Aktivität
  - ist eine permanent andauernde Aktivität eines Zustands. Sie wird regelmäßig ausgeführt.
- Nichtdeterminismus im Statechart,
  - wenn in einer Situation mehrere alternative, schaltbereite Transitionen existieren. Verhalten des Objekts ist unterspezifiziert.

SE SOFTWARE ENGINEERING

RWTH AACHEN

Statechart

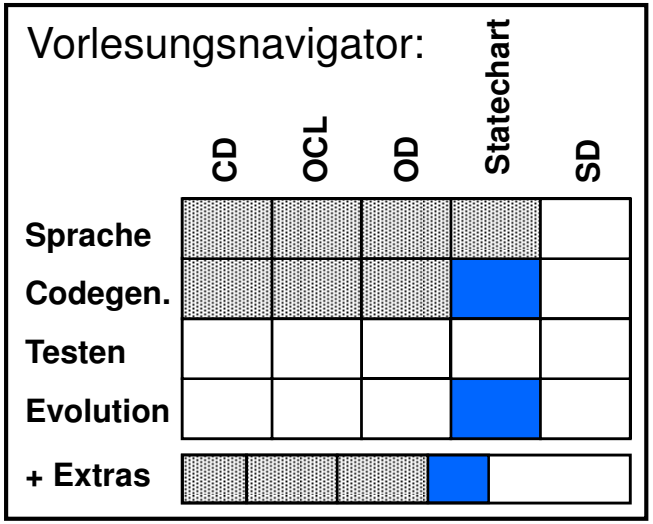
Modellbasierte Softwareentwicklung



- 5. Statecharts
- 5.5. Semantik, Codegenerierung, Transformation

Prof. Dr. Bernhard Rumpe
Lehrstuhl für Software Engineering
RWTH Aachen

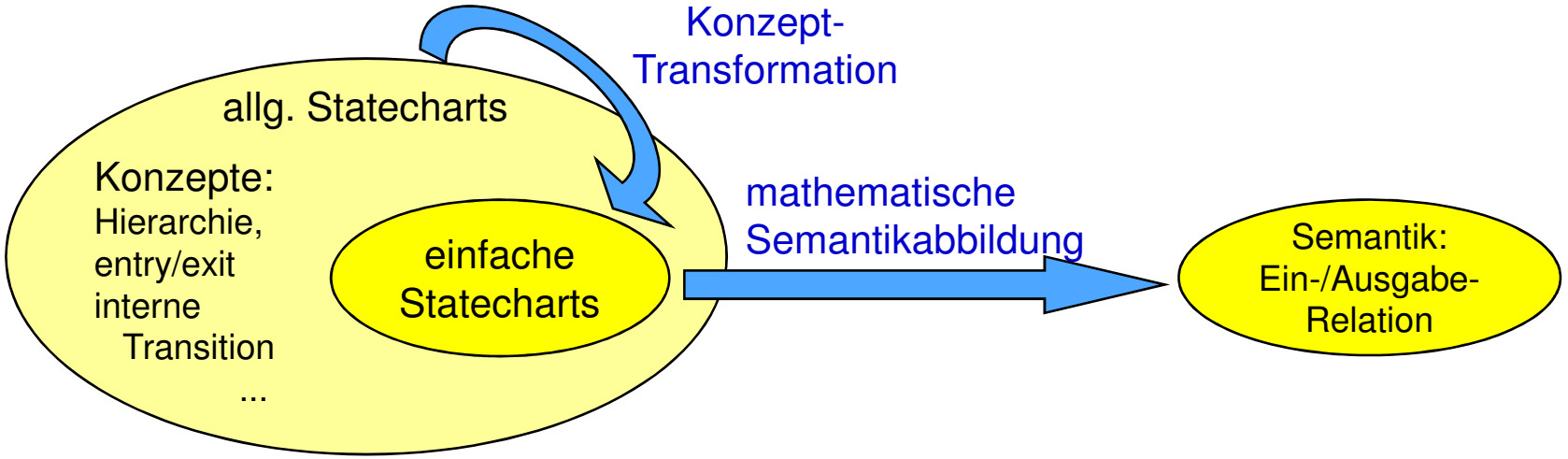
http://mbse.se-rwth.de/



Semantik - revisited

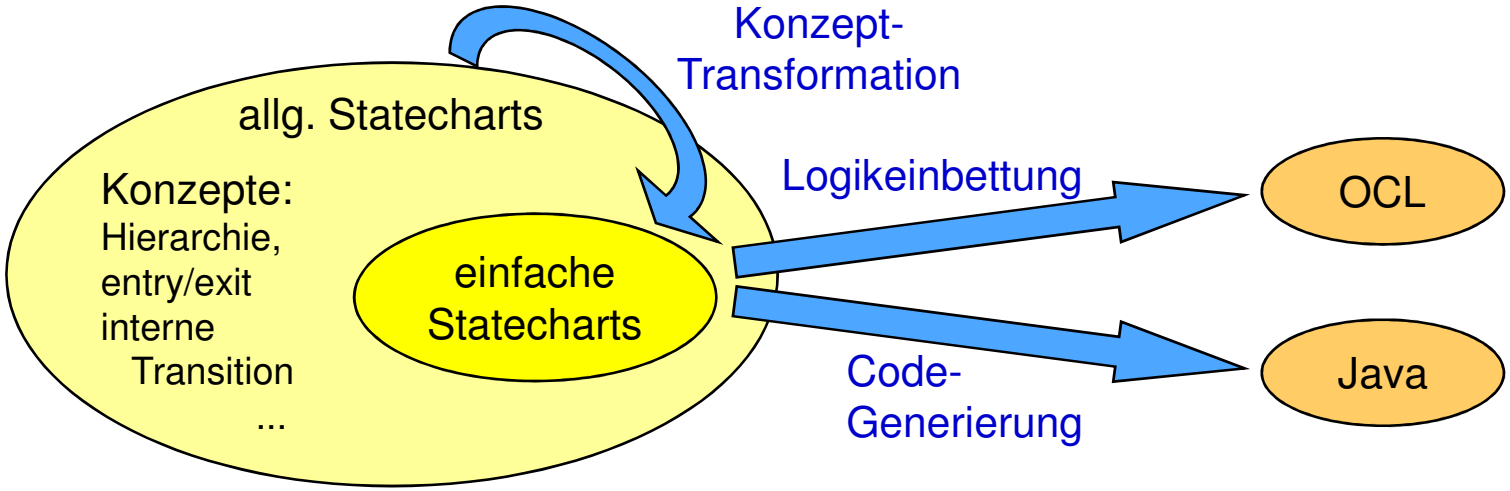
- Die Bedeutung eines Statechart wurde mehrstufig festgelegt:

1. Mealy-Automat:
   - Die Semantik eines Mealy-Automaten ist eine Relation zwischen Eingabe- und Ausgabe-Wörtern
   - Interpretation: Eingaben sind Methodenaufrufe, Ausgaben sind Aktionen
2. Zustandsinvarianten als präzisierendes Beschreibungsmittel
   - Verbindung zwischen Diagramm- und Objektzuständen
3. Erweiterung um Hierarchie, entry-/exit-Aktionen etc.
   - durch Transformation: Rückführung der Semantik auf einfacheren Teil-Dialekt der Statecharts



Transformationen von Statecharts

- Transformationen können komplexe Konzepte auf einfache reduzieren
- Anwendung bei
  - Semantikdefinition (wie vorher geschehen)
  - Generierung von Code
  - Optimierung von Statecharts (Zustandsminimierung, ...)
  - Abbildung der Statcharts in OCL-Bedingungen
- Transformation zur Codegenerierung ist analog zur Semantikdefinition, wichtig ist aber jetzt die schematische Ausführbarkeit:
  - unsere Konzepttransformationen leisten das



Vereinfachung von Statecharts durch Transformation

- Sammlung aus Transformationen bereits auf vorhergehenden Folien gegeben
  - Reihenfolge der Schritte ist noch festzulegen
- Die meisten Schritte sind automatisierbar
  - Entwurfsentscheidungen in manchen Fällen notwendig bzw. für die optimierte Umsetzung sinnvoll
  - Entscheidbarkeit bei OCL-Bedingungen nicht immer gegeben:
    - händisch prüfen oder Verifikations-Tool einsetzen?

- Optimierungsschritte sind in nachfolgendem Verfahren nur begrenzt enthalten.
- Ergebnis des Verfahrens: vereinfachtes Statechart ohne Hierarchie (flach), zustands-bezogene Aktionen.

Verfahren zur Vereinfachung von Statecharts: Schritte 1-9: Hierarchie entfernen

Nachfolgende Schritte wurden bei der Einführung der Konzepte erklärt:

1. Do-Aktivitäten eliminieren
2. Interne Transitionen zu echten Transitionen umformen
3. Zielzustände mit Subzuständen: Transitionen an Subzustände weiterleiten
4. Quellzustände mit Subzuständen: Analog Transitionen aus Subzuständen starten lassen
5. Wiederholung 3.-4. auf mehreren Hierarchieebenen bis Transitionen nur noch atomare Quell-, Zielzustände haben.
6. Exit-Aktionen der Aktion jeder verlassenden Transitionen hinzufügen und im Zustand entfernen.
7. Entry-Aktionen analog den ankommenden Transitionen hinzufügen.
8. Zustandsinvarianten von Superzuständen in die Subzustände aufnehmen.
9. Hierarchisch zergliederte Zustände entfernen.

Verfahren zur Vereinfachung von Statecharts: Schritt 10: Zustandsinvarianten verschärfen

10. Zustandsinvarianten verschärfen.

- Ausgangspunkt: $A \wedge B \neq \text{false}$

Z1 [A] Z2 [B]

- Ziel: Datenzustände erhalten
  durch Überführung in disjunkte Zustandsinvarianten
- Alternativen:
  - Invarianten mit weiteren Bedingungen konjugieren, bis disjunkt

Z1 [A && C] Z2 [B && !C] // C geeignet

  - Zustandsattribut („status“) einführen und in Invariante nutzen

Z1 [A && status==1] Z2 [B && status==2]

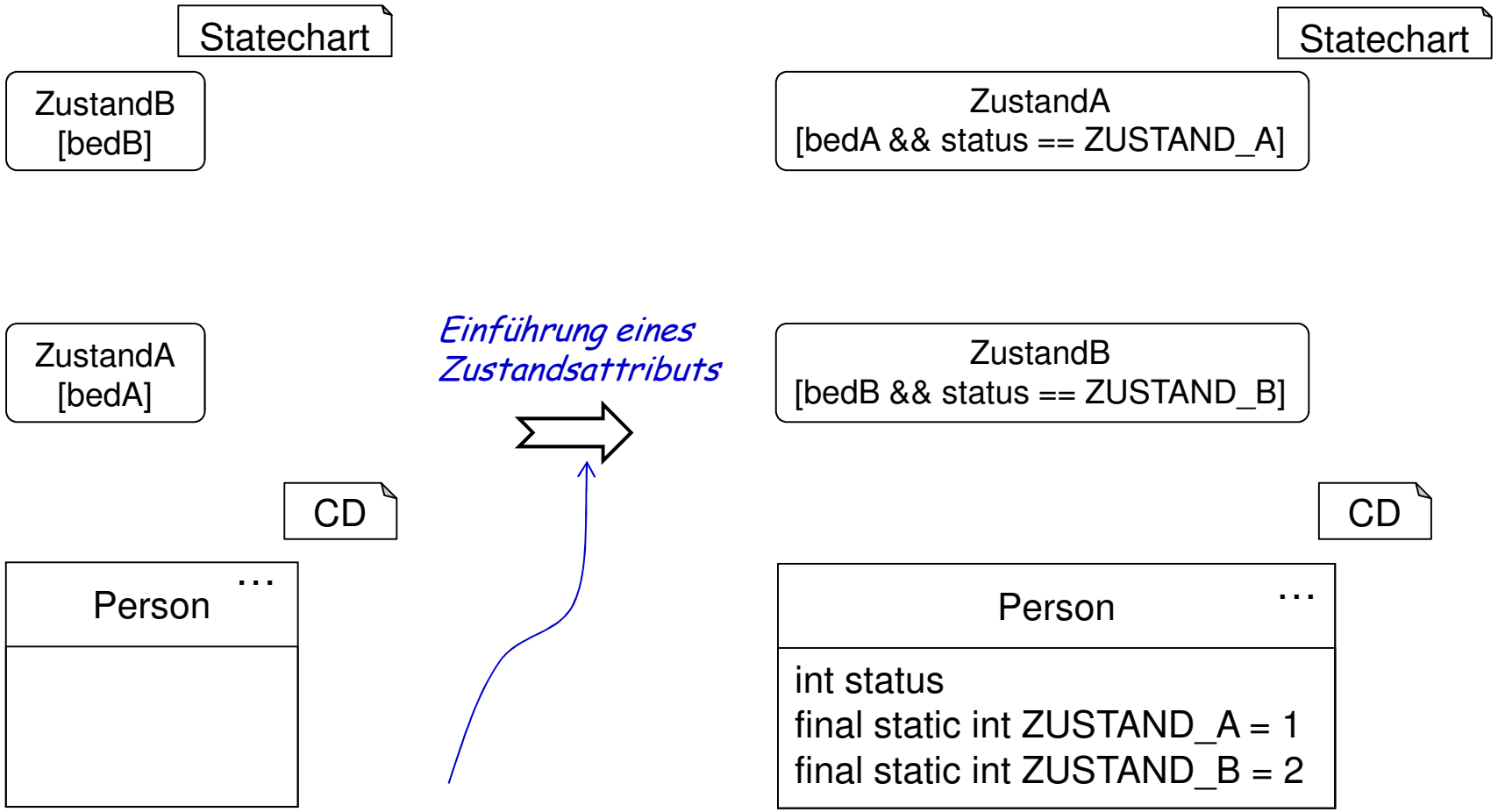
  - Schnittmenge aus einem Zustand herausnehmen

Z1 [A && !B] Z2 [B] oder

Z1 [A] Z2 [B && !A]

Beispiel für Schritt 10:

- z. B. verwendbar bei OCL-Umsetzung



dieser Pfeil weist auf den „generierenden" Aspekt der Transformation hin.

Verfahren zur Vereinfachung von Statecharts: Schritt 11-13: Zustandsinvarianten entfernen

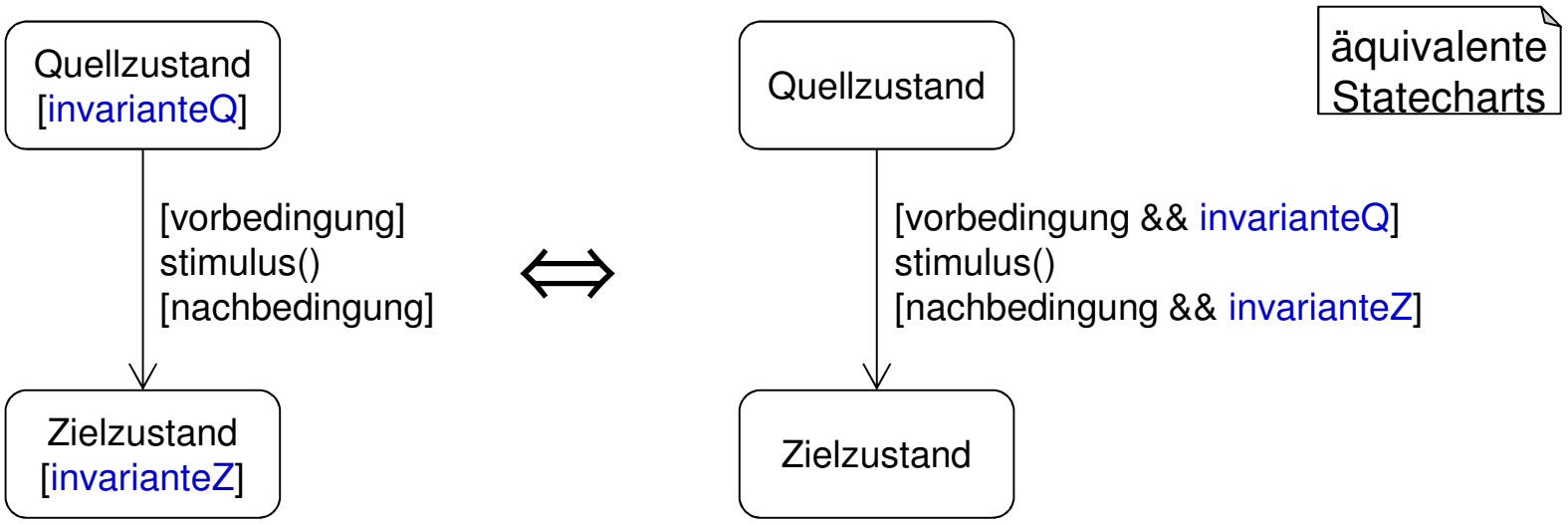
11. Zustandsinvarianten in die Vorbedingungen integrieren.

- Ziel:
  - Vorbedingungen von Transitionen enthalten alle Information

12. Zustandsinvarianten mit Aktionsbedingungen konjugieren.

- Ziel:
  - Aktionsbedingungen von Transitionen enthalten alle Information

13. Zustandsinvarianten entfernen.

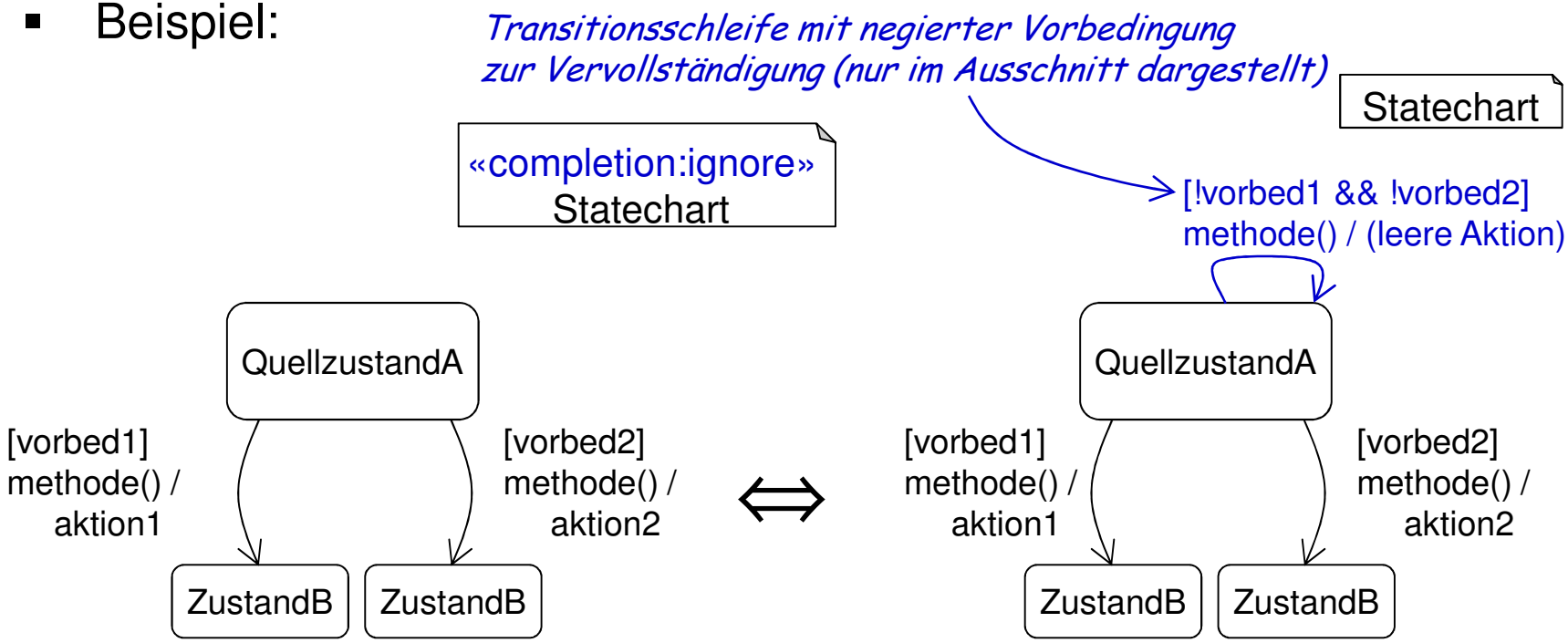


Verfahren zur Vereinfachung von Statecharts: Schritt 14: Vervollständigung

14. Vervollständigung des Statechart

- je nach Typ: «error», «exception», «completion:ignore»
- (nicht sinnvoll bei «completion:chaos»)

- Ziel: Stereotypen expandieren im Statechart
- (Diese Expansion ist nicht effizient für Codegenerierung!)
- Beispiel:



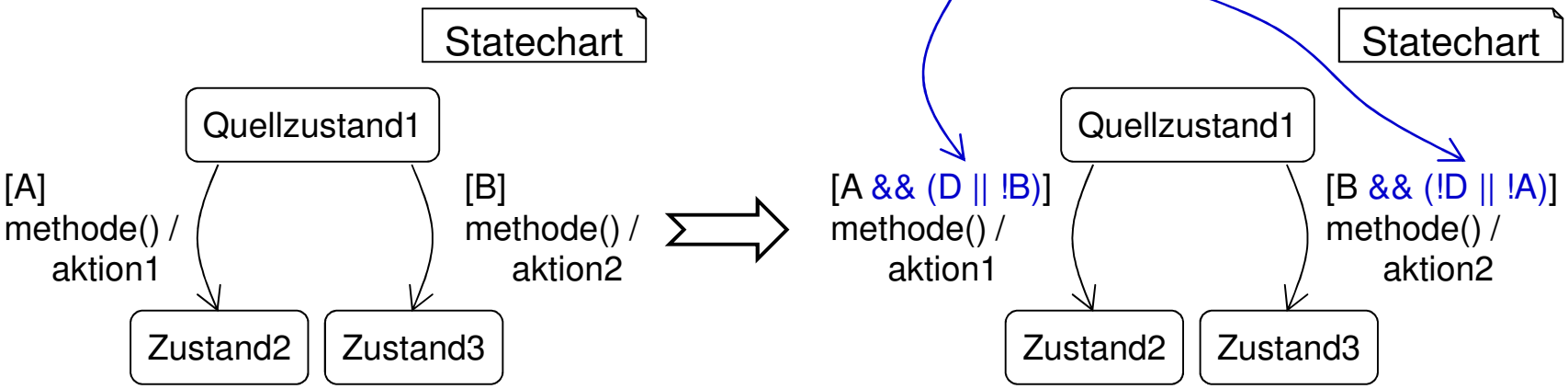
Verfahren zur Vereinfachung von Statecharts: Schritt 15: Nichtdeterminismus

15. Nichtdeterminismus der Transitionen reduzieren

- Einführung eines Diskriminators D

- Ziel: deterministisches Statechart
- Bei Codegenerierung gibt es effizientere Techniken: z.B. Reihenfolge der Prüfung von Vorbedingungen.
- Beispiel:

Nichtdeterminismus reduzieren, indem eine Diskriminatorbedingung D in normaler und negierter Form zu einem Paar überlappender Vorbedingungen hinzugefügt wird, D ist frei wählbar, Beispiel: (D==true) bedeutet 1 hat Priorität



Verfahren zur Vereinfachung von Statecharts: Schritt 16-17: Schaltbereitschaft, Erreichbarkeit

16. Transitionen ohne Schaltbereitschaft eliminieren

- Erkennbar an Vorbedingung == false

- Ziel: Durch die Transformationen sind viele Transitionen dupliziert und mit zusätzlichen Vorbedingungen versehen worden.
  - So entstehen leere Schaltbereiche: Diese Transitionen sind entfernbar!
  - Ideal: Bereits beim Transformieren die Schaltbereiche prüfen
  - Unentscheidbarkeit, wenn OCL-Bedingungen involviert

17. Nicht erreichbare Zustände eliminieren

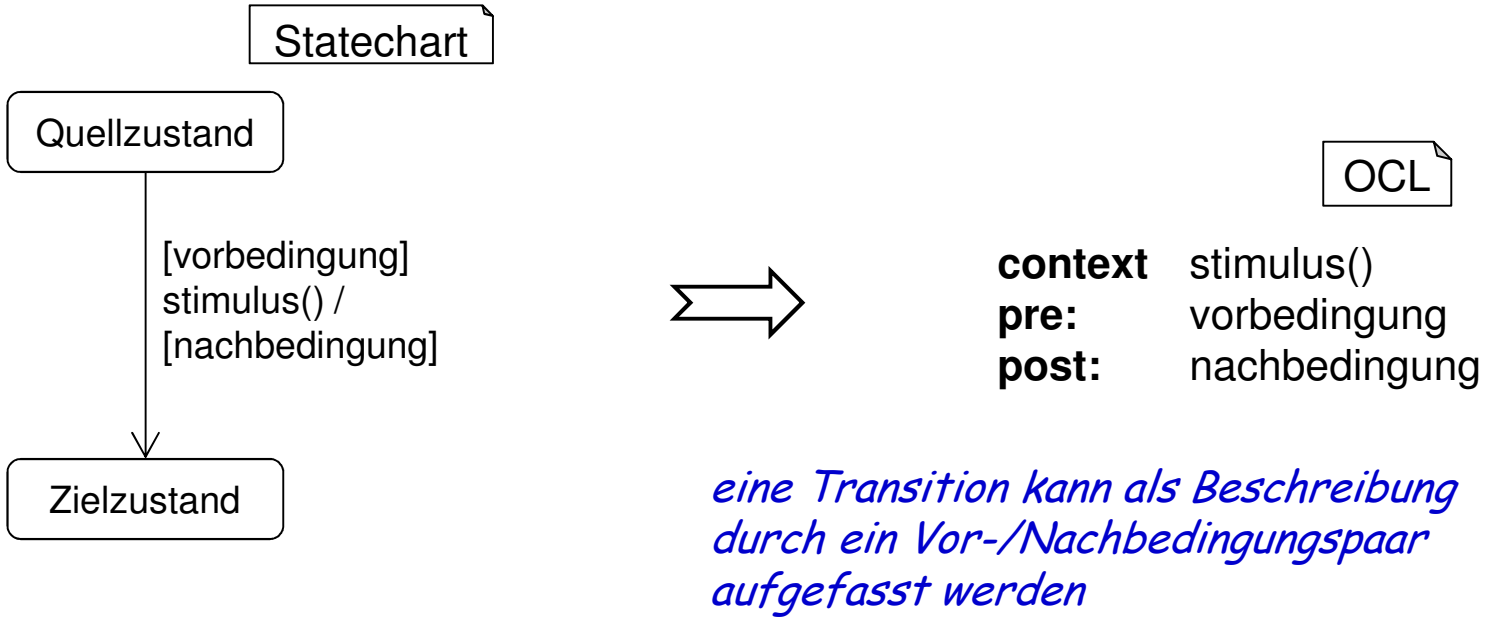
- Transitive Hülle über schaltbare Transitionen

- Letzte beiden Schritte sind Optimierungen.
- Gesamtergebnis: Eine wesentlich vereinfachte flache Form von Statecharts

Abbildung in die OCL

- Zur Durchführung von logischen Beweisen, oder zur Testfallgenerierung sinnvoll:
  - Transformation Statecharts in die OCL

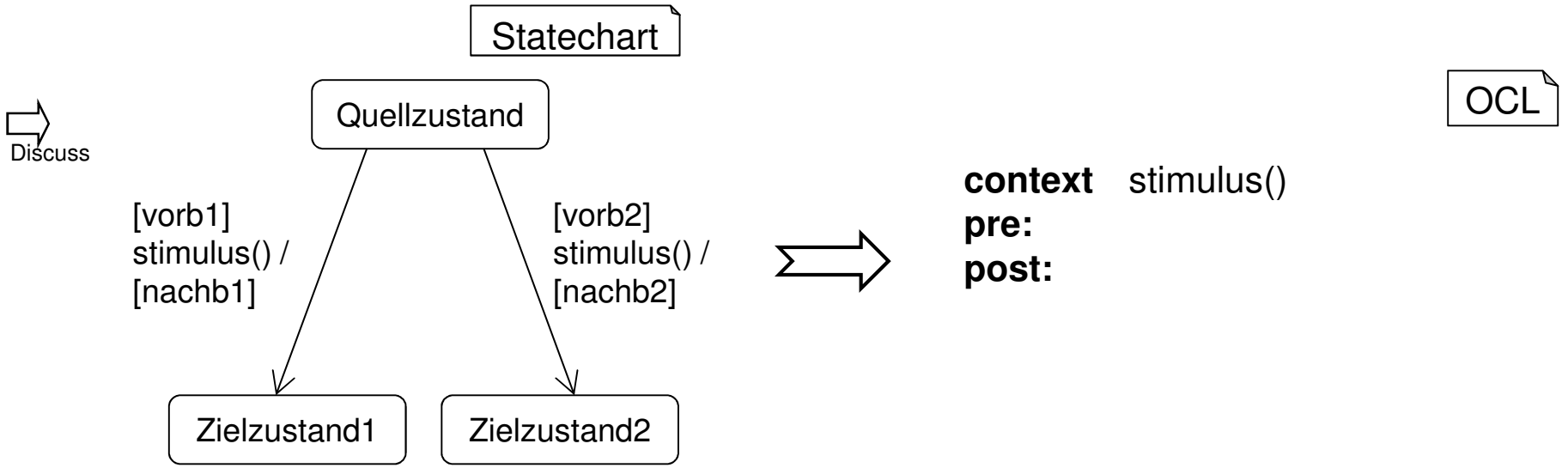
- Standardtransformation ausgehend vom vereinfachten Statechart:



eine Transition kann als Beschreibung durch ein Vor-/Nachbedingungspaar aufgefasst werden

Abbildung in die OCL - 2

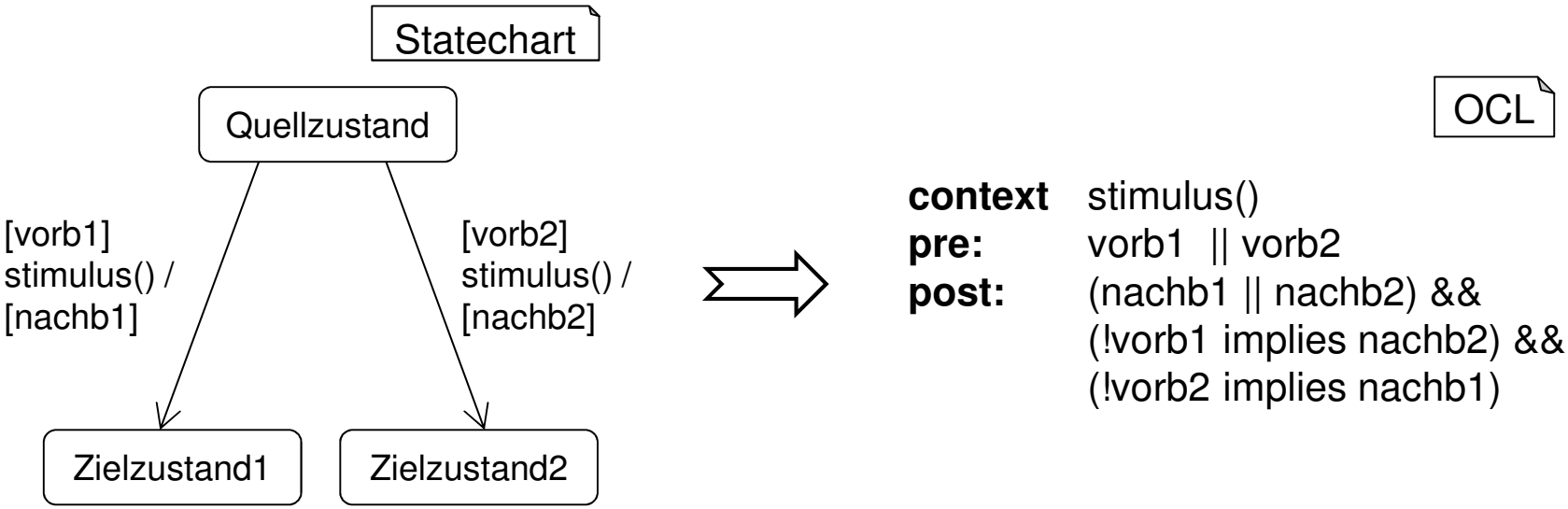
- Falls die Eliminierung von Unterspezifikation nicht gewünscht war:
- Überlappende Transitionen werden so übersetzt:
  (analog zur Kombination von OCL-Methodenspezifikationen)



zwei überlappende Transitionen werden als Beschreibung durch ein Vor-/Nachbedingungspaar aufgefasst1

Abbildung in die OCL - 2

- Falls die Eliminierung von Unterspezifikation nicht gewünscht war:
- Überlappende Transitionen werden so übersetzt:
  (analog zur Kombination von OCL-Methodenspezifikationen)



zwei überlappende Transitionen werden als Beschreibung durch ein Vor-/Nachbedingungspaar aufgefasst1

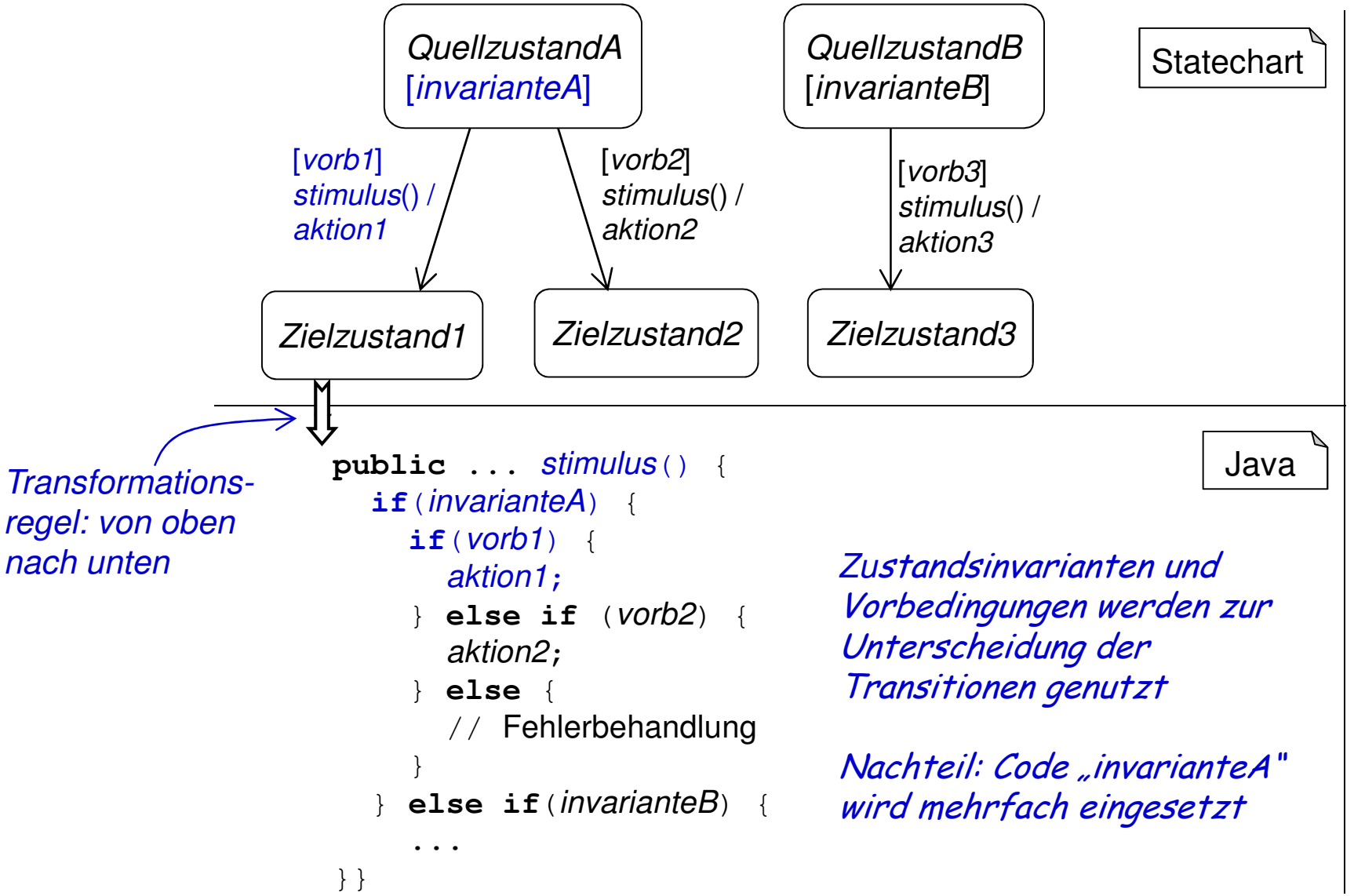
Codegenerierung

- Ausgangspunkt:
  - vereinfachte Statecharts (Zustandsinvarianten noch nicht expandiert)

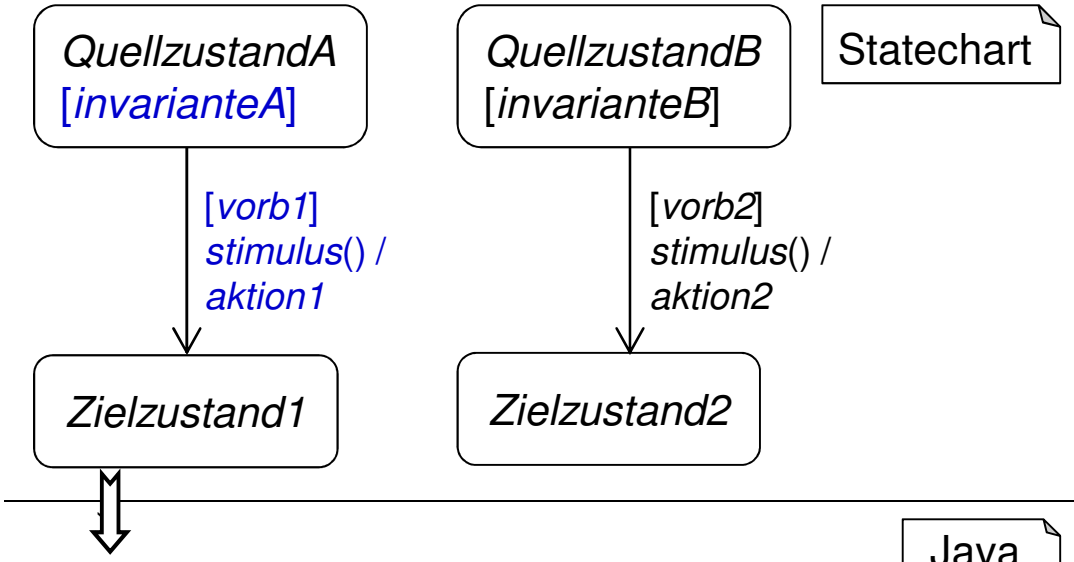
- Mehrere Varianten für Darstellung von Zuständen:
  - Explizites Zustandsattribut beschreibt Zustand, oder
  - Invarianten disjunkter Zustände als Prädikate, oder
  - State-Muster: Jedem Zustand ein eigenes Objekt zugeordnet

- Methoden-Statecharts werden nochmals anders behandelt.

Disjunkte Invarianten für Zustände



Auslagerung Zustandsinvarianten in eigene Prädikate



Java

```
public boolean invQuellzustandA() {
    return invarianteA;
}
public boolean invQuellzustandB() {
    return invarianteB;
}
public ... stimulus() {
  if(invQuellzustandA()) {
    ...
}
```

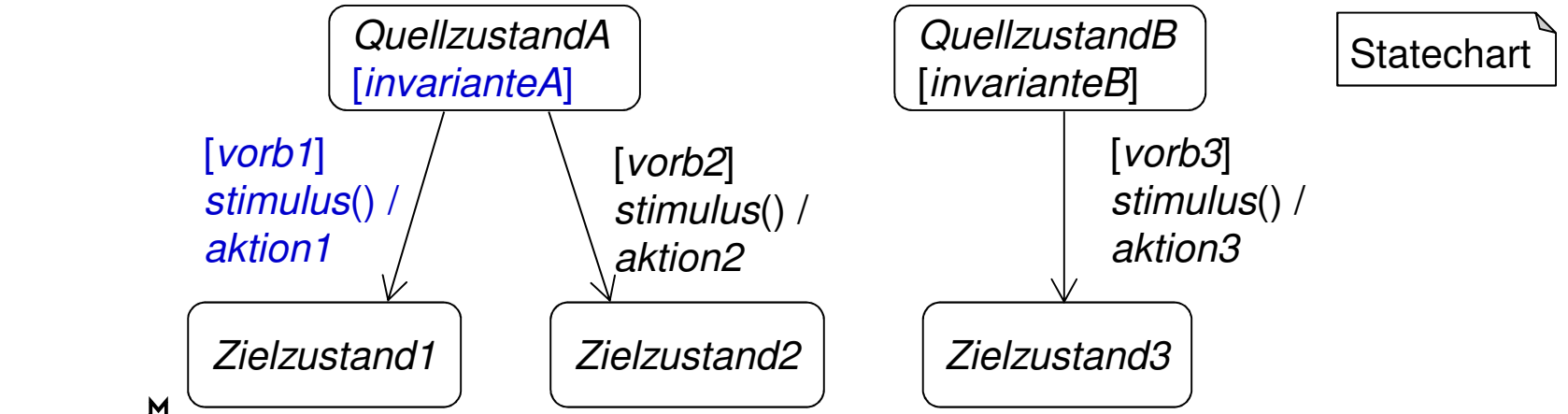
jeder Zustand wird zu einem Prädikat, das die Zustands-invariante evaluiert

Vorteil:
- „invarianteA“ nur einmal generiert.

Nachteil:
- „invarianteA“ kann komplex sein und zeitaufwendig zu berechnen

Besser:
- Zustandsattribut speichert aktuellen Zustand

Einführung eines Zustandsattributs




```
private int status;
final static int QUELLZUSTAND_A = 1;
final static int QUELLZUSTAND_B = 2;
final static int ZIELZUSTAND1 = 3; ...
```

der Diagrammzustand wird als Aufzählung gespeichert

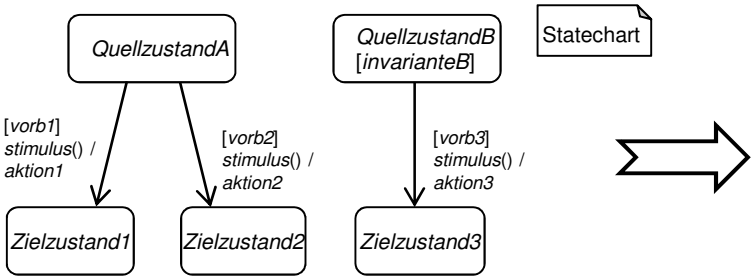
Java

```
public ... stimulus() {
  switch(status) {
    case QUELLZUSTAND_A:
      if(vorb1) {
        aktion1;
        status = ZIELZUSTAND1;
      } else if (vorb2) {
        aktion2;
        status = ZIELZUSTAND2;
      } ...
    break;
    case QUELLZUSTAND_B:
      ...
}}
```

Vorteil: Effizient
Nachteile: evtl. redundante Speicherung,
Konsistenz nicht gesichert:
(status==QUELLZUSTAND_A)
impliziert invarianteA

Nutzung der Invarianten für Tests

Ausgangsstatechart wie auf vorheriger Folie



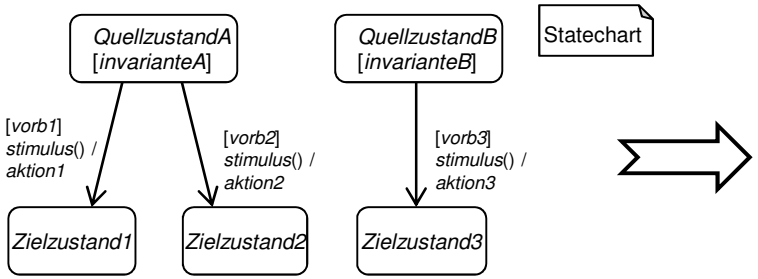
Java

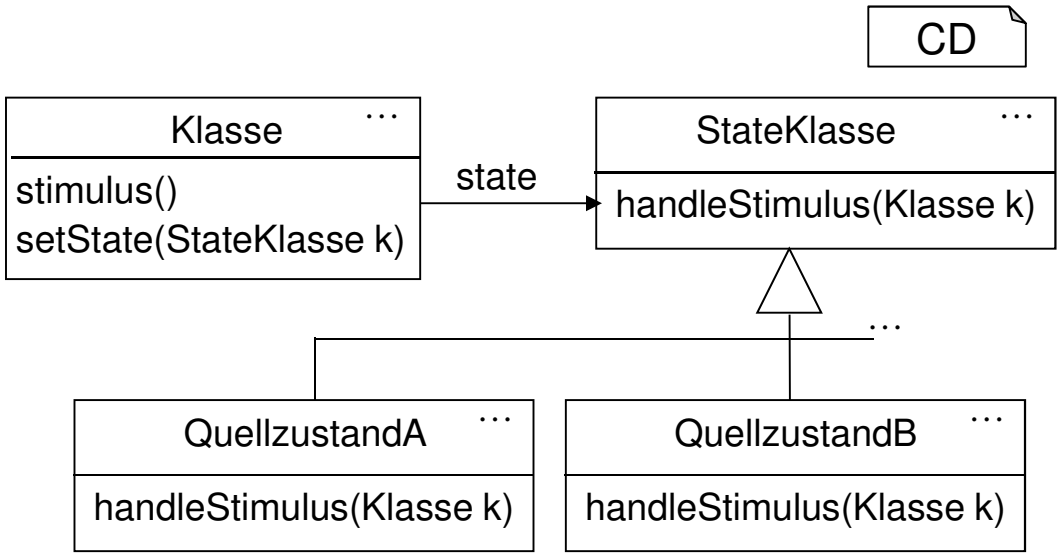
```
private int status;
final static int QUELLZUSTAND_A = 1;
final static int QUELLZUSTAND_B = 2;
final static int ZIELZUSTAND1 = 3;
...

public ... stimulus() {
  switch(status) {
    case QUELLZUSTAND_A:
      ocl invarianteA;
      if(vorb1) {
        aktion1;
        status = ZIELZUSTAND1;
      } else {
        ocl vorb2;
        aktion2;
        status = ZIELZUSTAND2;
      } ...
    break;
    case QUELLZUSTAND_B:
      ...
}}
```

Zustandsinvarianten und manche Vorbedingungen können in ocl-Aweisungen als Zusicherungen zu Testzwecken eingesetzt werden, wenn angenommen wird, dass das Diagramm vollständig ist

Entwurfsmuster: State (Gamma et.al. 1994)

Ausgangsstatechart wie auf vorheriger Folie





Java

```
class Klasse {
   QuellzustandA quellzustandA = ...
   Zielzustand1 zielzustand1 = ...

   public ... stimulus() {
      state.handleStimulus(this);
   }
}

class QuellzustandA {
   public .. handleStimulus(Klasse k)
{
   ocl invarianteA';
   if(vorb1) {
     aktion1';
     k.setState(k.zielzustand1);
   } else
   ...
}}
```

Vorteil: das State-Entwurfsmuster kann für zusätzliche Flexibilität verwendet werden
Nachteil: Overhead durch zusätzliche Objekte: je eines pro Zustand.

Vererbung von Statecharts

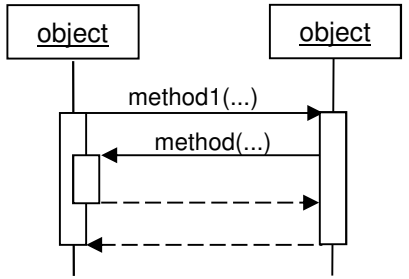
- Welche Aussage trifft ein Statechart der Superklasse für Objekte der Subklasse?
  - Unterschiedliche Meinungen (Harel, UML, Rumpe, ...)
- Formal:
  - Subklasse führt zu Verhaltensverfeinerung
  - Deshalb: Verfeinerung des durch Statechart spezifizierten Verhaltens kann gefordert werden.
  - Entsprechende Transformationsregeln existieren
- Pragmatisch:
  - Verhaltensverfeinerung durch Trafo.-Regeln zu starr
  - Besser Einsatz der Automaten zum Testen von Verhaltenskonformität.

Zusammenfassung Statecharts

- Statecharts sind eine Weiterentwicklung des Mealy-Automaten
- Statecharts bilden eine mächtige Form zur Definition von Verhalten auf Basis von Zuständen
- Die Kombination mit Codestücken für Aktionen, bzw. OCL für Bedingungen macht Statecharts beschreibungsvollständig und komfortabel.

- Eine Reihe von Variationen für Statecharts erlauben unterschiedliche Einsatzgebiete:
  - Methodenbeschreibungen
  - Lifecycles
  - Testfolgen
- in unterschiedlichen Phasen der Softwareentwicklung: Analyse, Design, Implementierung

SD

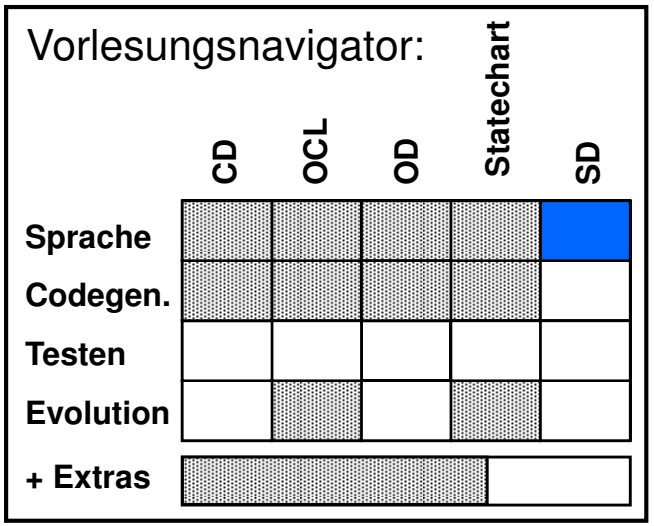


Modellbasierte Softwareentwicklung

- 6. Sequenzdiagramme
- 6.1. Konzepte, Syntax

Prof. Dr. Bernhard Rumpe
Lehrstuhl für Software Engineering
RWTH Aachen

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



Sequenzdiagramme (SD)

- Ziel:
  - Modellierung von exemplarischen Beobachtungen
  - Darstellung von Interaktionsmustern von Objekten
  - Zeitliche Reihenfolge von Aufrufen

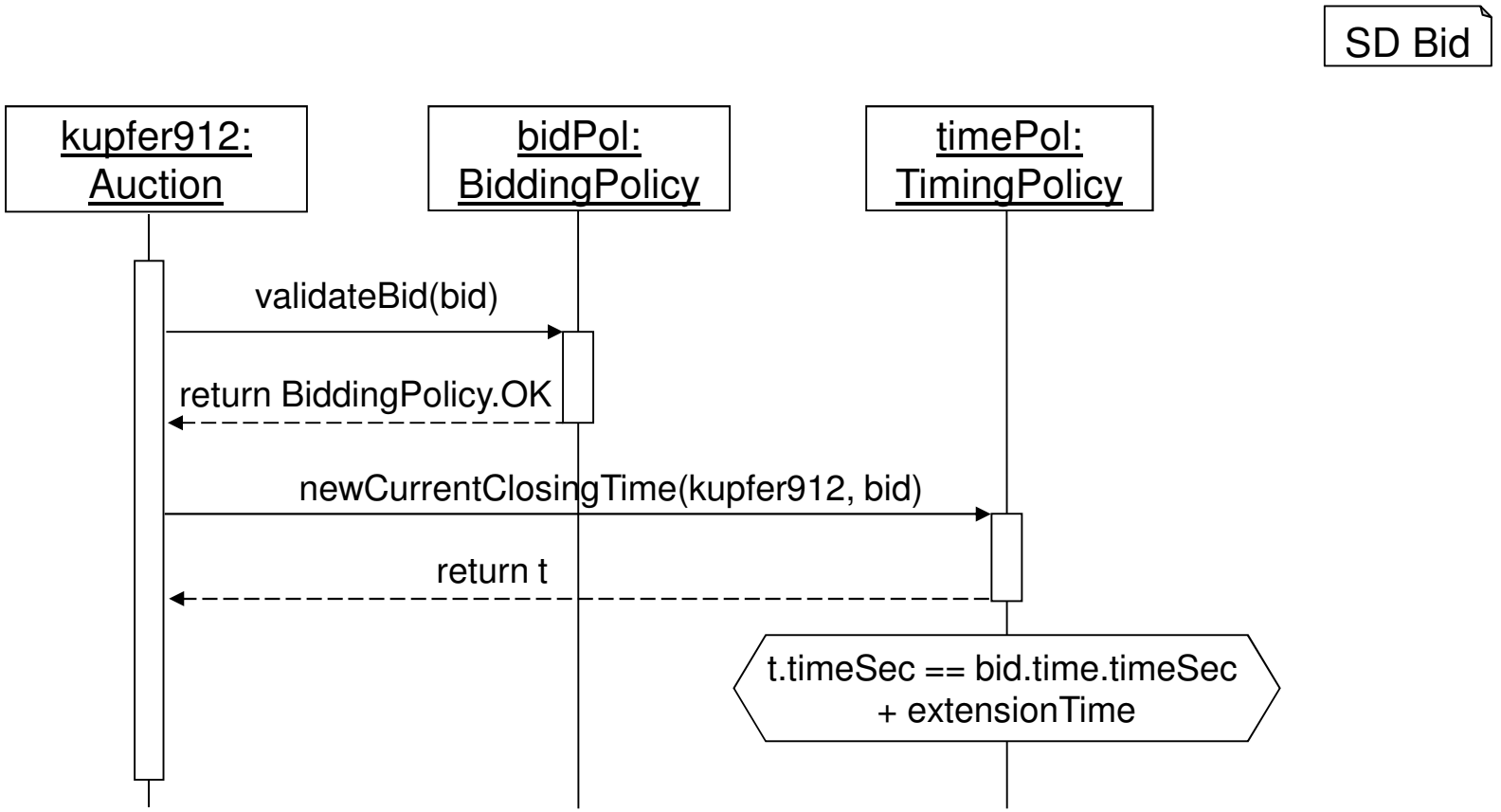
- Wesentlich ist
  - die Exemplarizität
  - der Fokus auf Interaktion

- Vergleich SD und Statechart: beides Verhaltensbeschreibungen

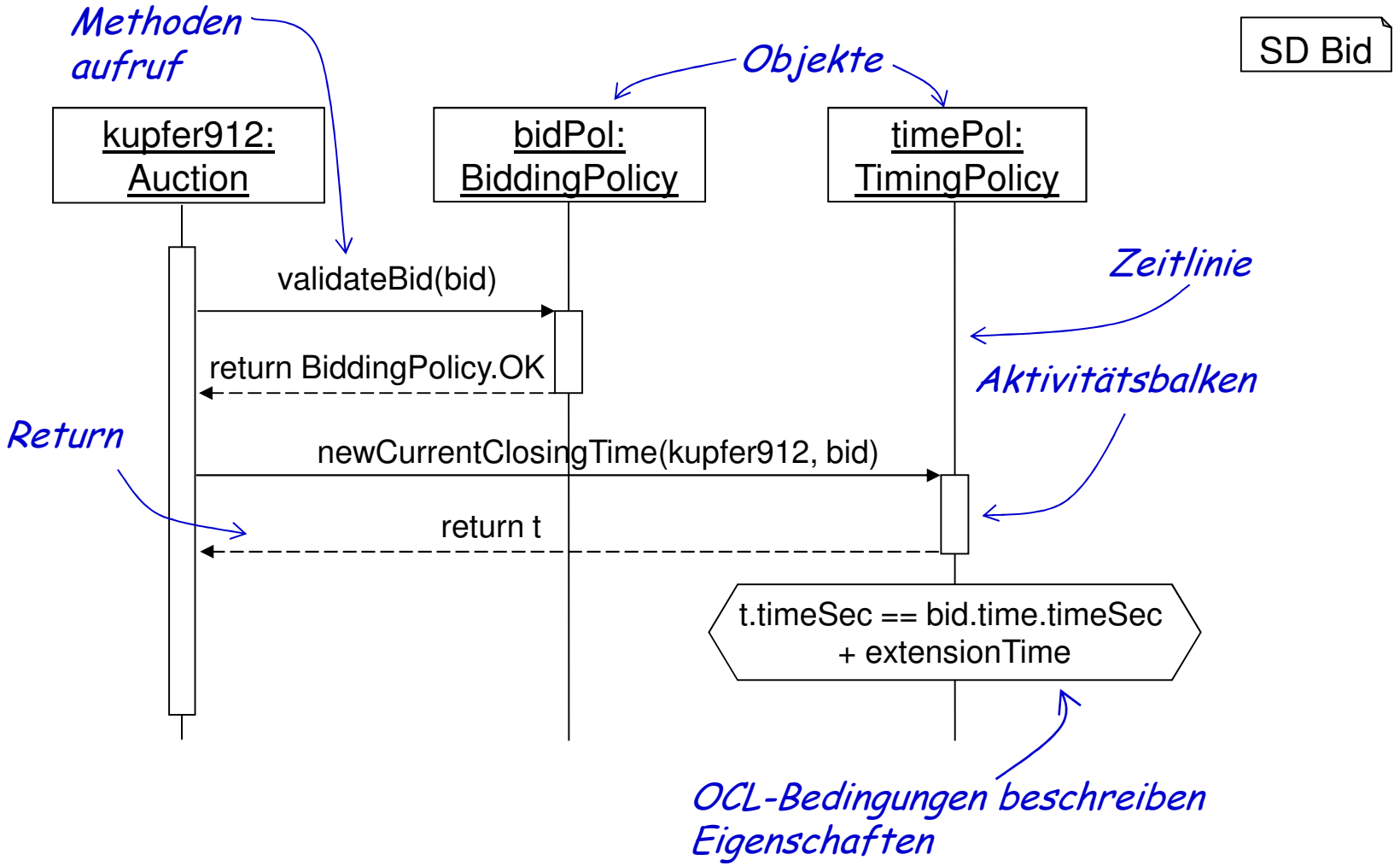
Sequenzdiagramme	Statecharts
Interaktion mehrerer Objekte	Verhalten eines Objekts
exemplarisch	vollständig
kein interner Zustand	zustandsbasiert

Beispiel: Sequenzdiagramm

Analyze

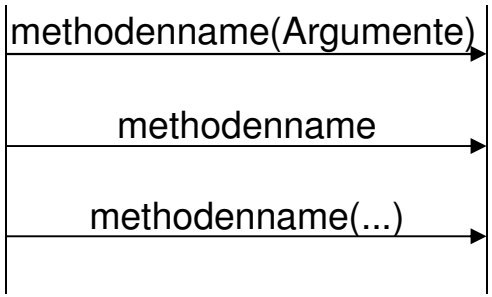


Beispiel: Sequenzdiagramm

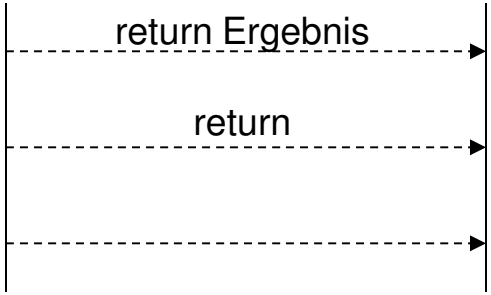


Interaktionsformen

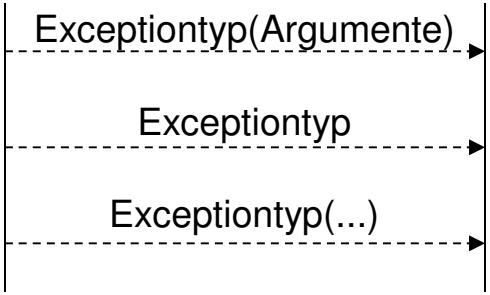
- Pfeilarten beschreiben drei Interaktionsformen
  - a) Methodenaufruf und asynchrone Nachrichtenübertragung (werden hier nicht unterschieden)
  - b) Ergebniss eines früheren Methodenaufrufs (Return)
  - c) Exception (als anormale Terminierung)
- Darstellungsformen:
  - a)



b)

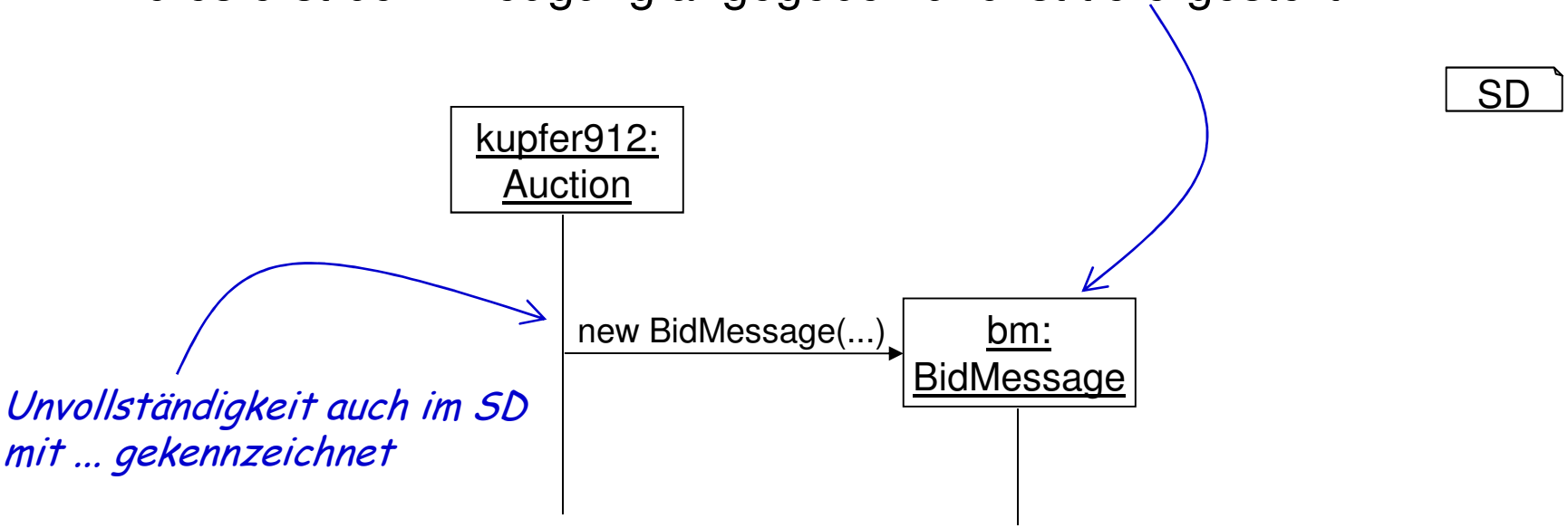


c)



Objekterzeugung mit Konstruktor

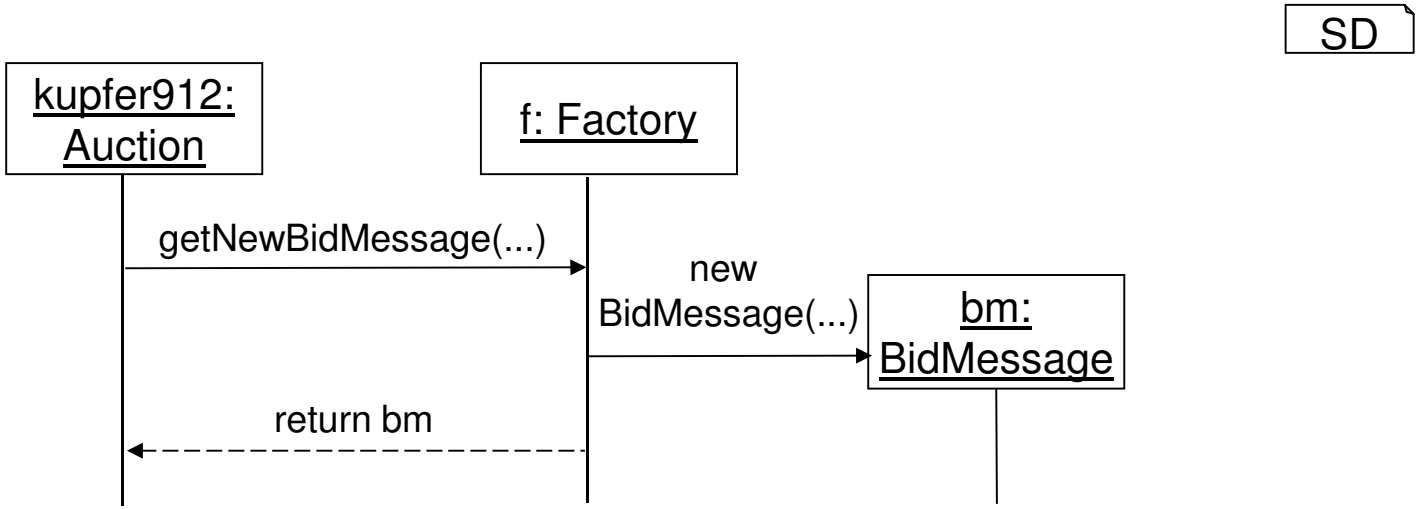
- Ist ein Objekt zum Beginn der Beobachtung noch nicht „lebendig, so wird es erst bei Erzeugung angegeben und ist tiefergestellt:



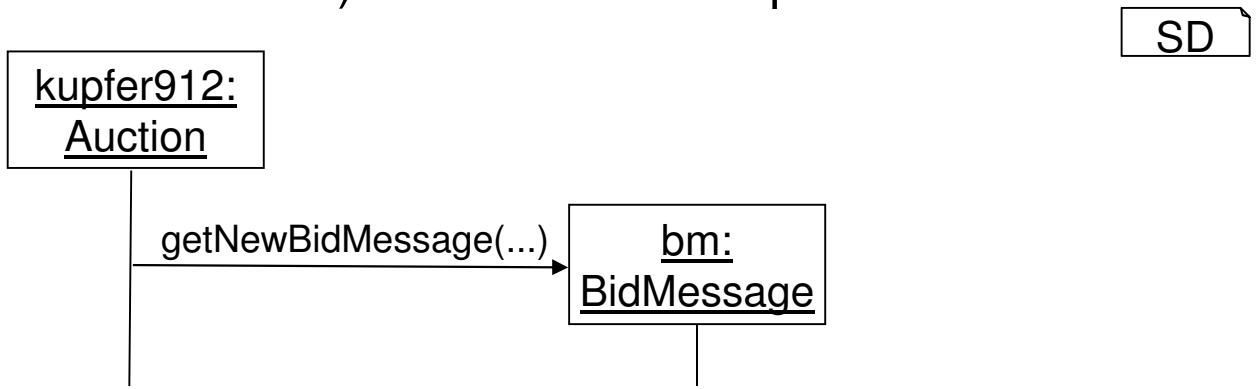
- Für C++-Abläufe existiert ein ähnliches Konstrukt zur Terminierung eines Objekts. Zielsprache Java benötigt dies nicht.

Objekterzeugung mit Factory

- Beispiel, wie ein Objekt über eine Factory erzeugt wird:

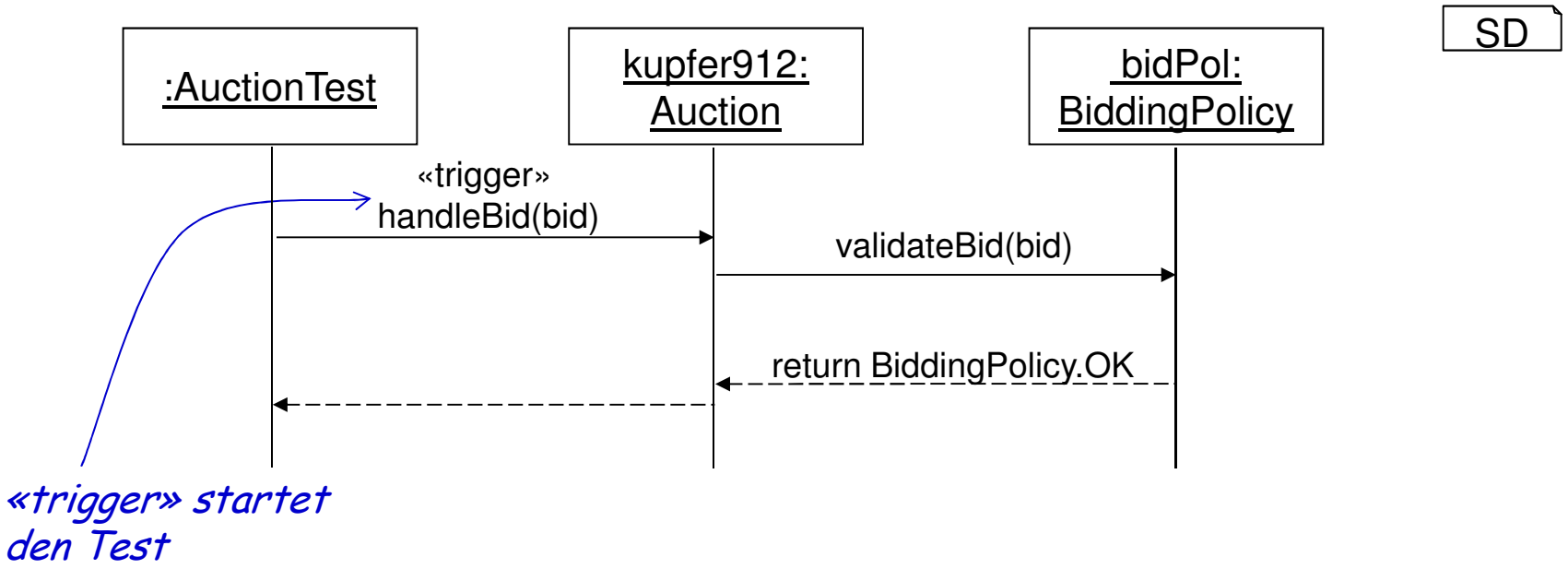


- Allerdings: Abstraktion von Factories sinnvoll, wenn die Semantik (also die Interpretation des SD) das zulässt. Beispiel ist dann:



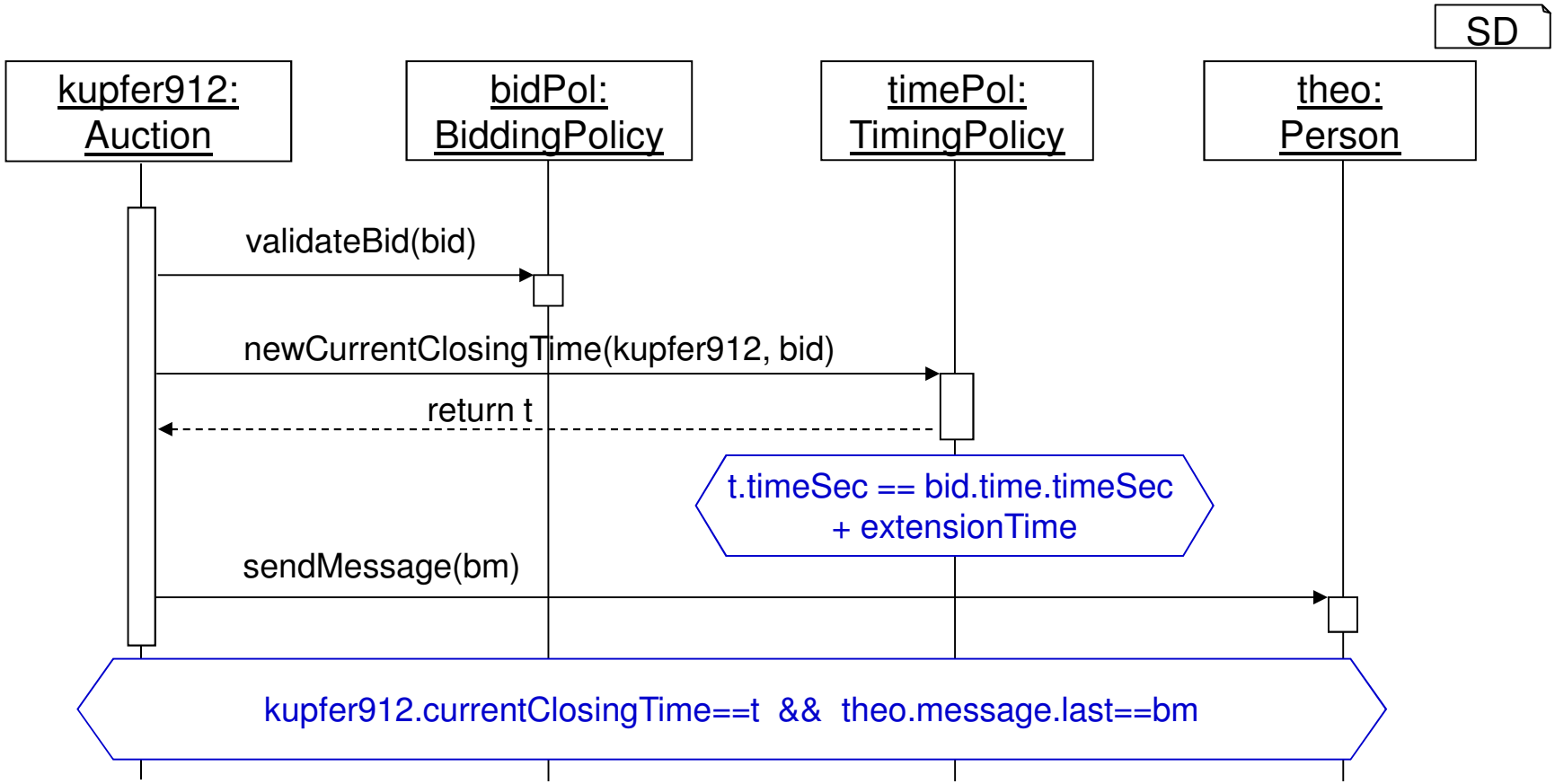
Stereotypen

- auch das SD erlaubt eigene Stereotypen und bietet vordefinierte.
- Beispiel:
  - «trigger» markiert den Aufruf, der die Interaktionen des Sequenzdiagramms auslöst
  - Einsatzgebiet, z. B. zur Modellierung von Tests:



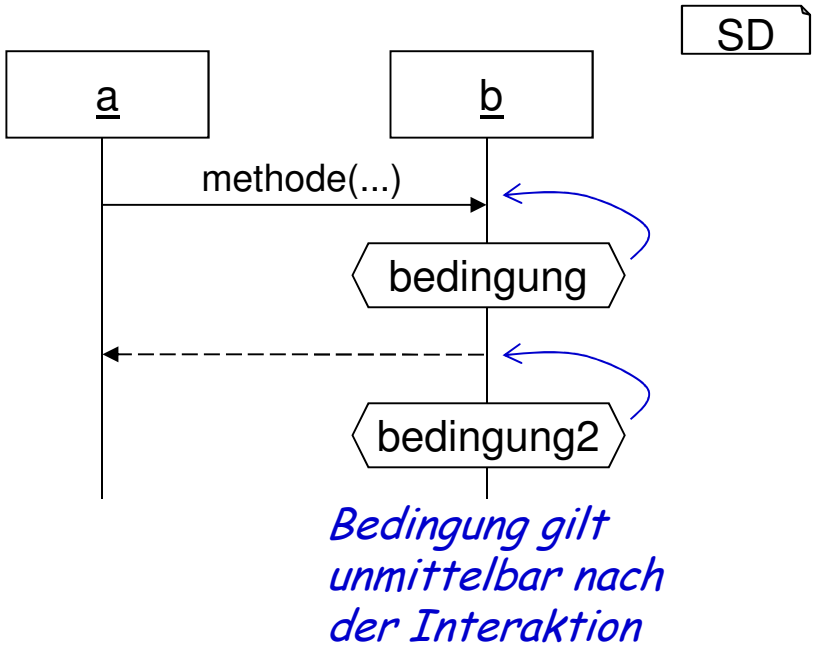
OCL-Bedingungen im Sequenzdiagramm

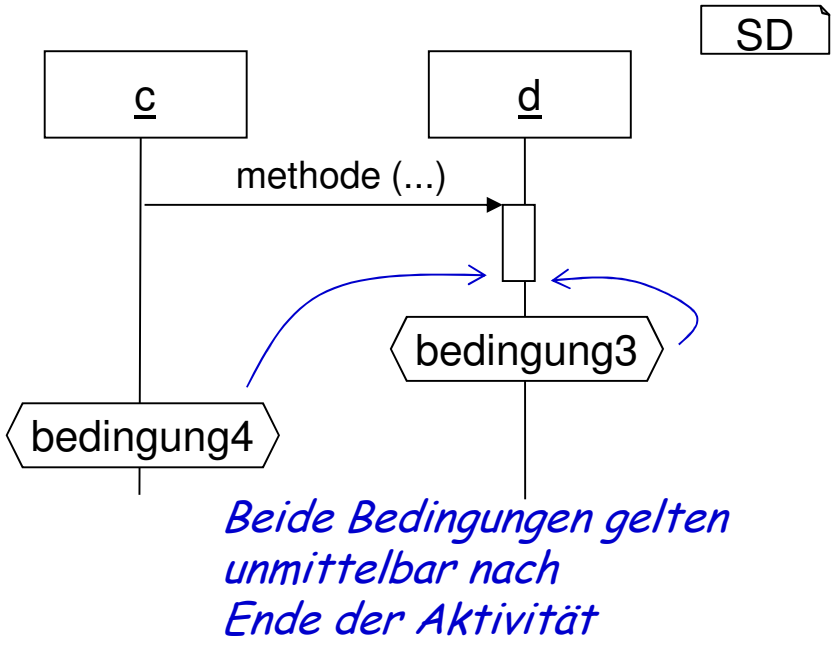
- OCL-Bedingung charakterisiert eine Eigenschaft, die mitten im Ablauf gelten soll:



OCL-Bedingungen im Sequenzdiagramm

- Präzise Gültigkeit der OCL-Bedingungen im Ablauf:

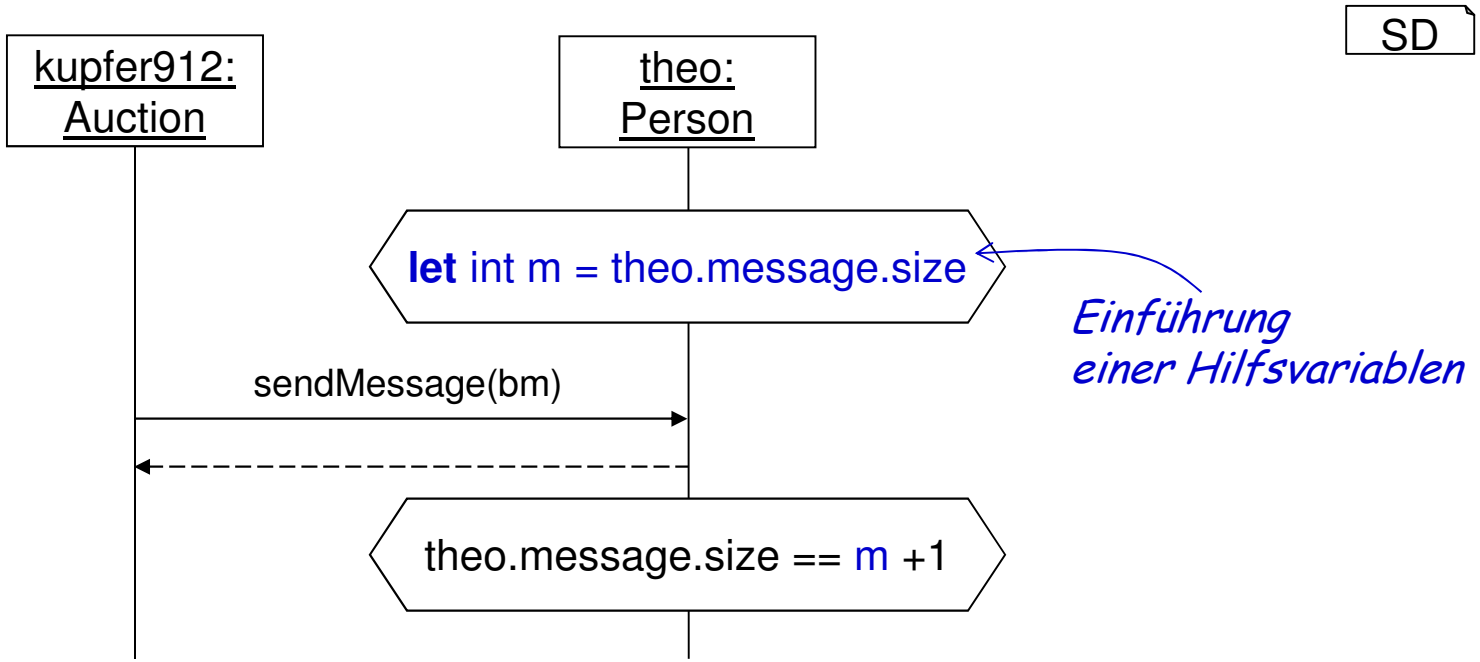




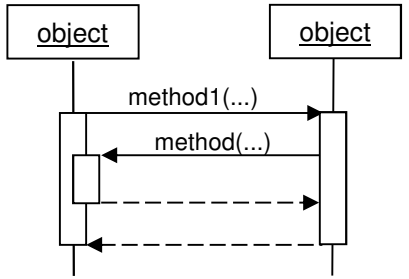
- In OCL verwendbare Variablennamen:
  - alle Objekte,
  - Attribute der Objekte, über deren Zeitlinie sie liegt (wenn eindeutig)
  - Argumente vorhergehender Methodenaufrufe

Hilfsvariablen in Sequenzdiagrammen (let)

- Analog der let-Variablen in Vor-/Nachbedingungen zur Wiederverwendung in späteren Bedingungen



SD

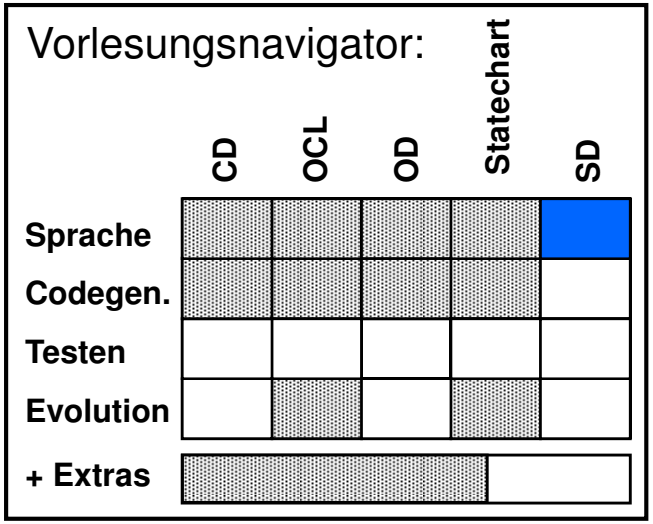


Modellbasierte Softwareentwicklung

- 6. Sequenzdiagramme
- 6.2. Semantik

Prof. Dr. Bernhard Rumpe
Lehrstuhl für Software Engineering
RWTH Aachen

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

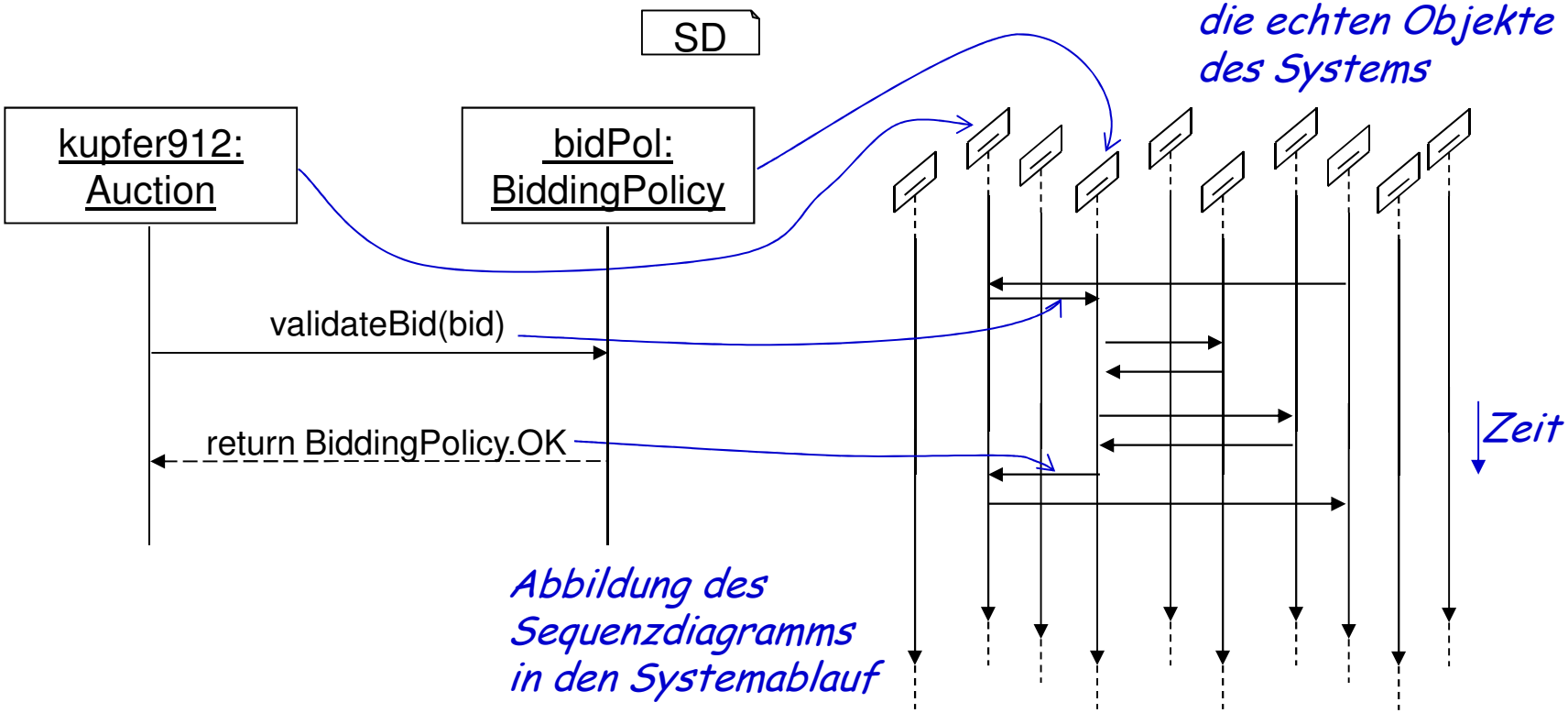


Exemplarizität und Unvollständigkeit

- Ein Sequenzdiagramm beschreibt einen Ausschnitt eines Ablaufs des System:
  - Die Objektmenge ist unvollständig
  - Argumente von Methodenaufrufen dürfen fehlen
  - Vor und nach dem gezeigten SD finden weitere Interaktionen statt.
    - Zwischendurch ebenfalls?
  - Der Ablauf kann mehrfach auftreten,
    - er kann zeitlich verschachtelt auftreten,
    - er kann auch gar nicht auftreten.
  - Welche Semantik hat nun ein Sequenzdiagramm?

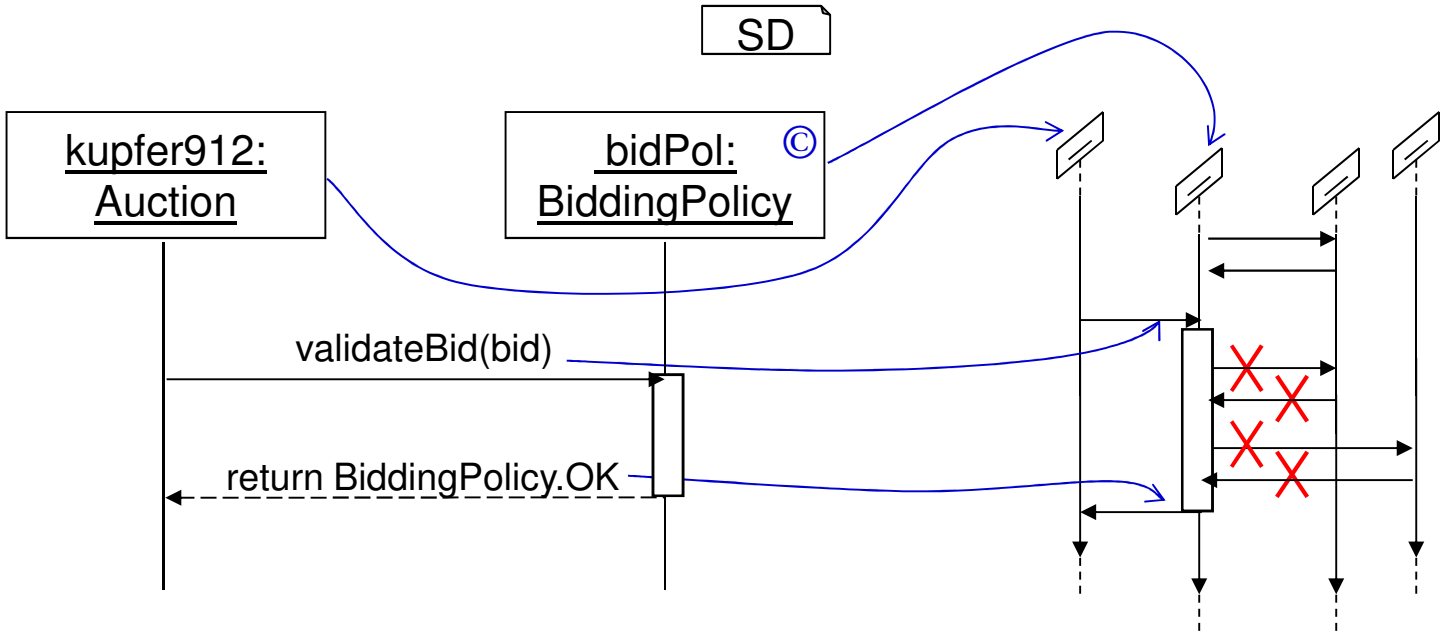
Präzise Bedeutung eines SD

- durch Abbildung der
  - prototypischen Objekte im SD auf echte Objekte des Systems
  - Interaktionen des SD auf echte Interaktionen im System
  - (präzise Definition siehe B. Rumpe: Modellierung mit UML)



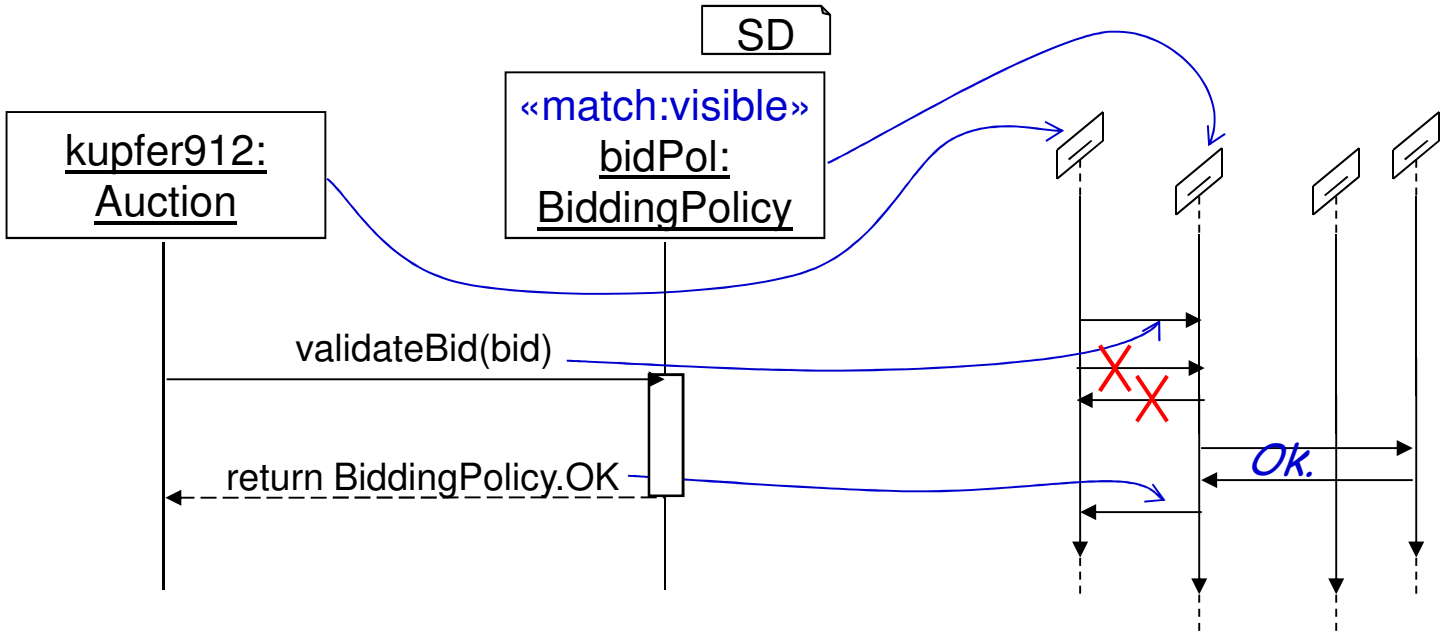
Vollständige Interaktionsliste

- Beispiel: Alle im Zeitraum der Beobachtung stattfindenden Interaktionen eines Objekts sind enthalten, d.h. es gibt keine weiteren.
- Markierung «match:complete» (kurz: ©) verbietet andere Interaktionen zwischendurch:



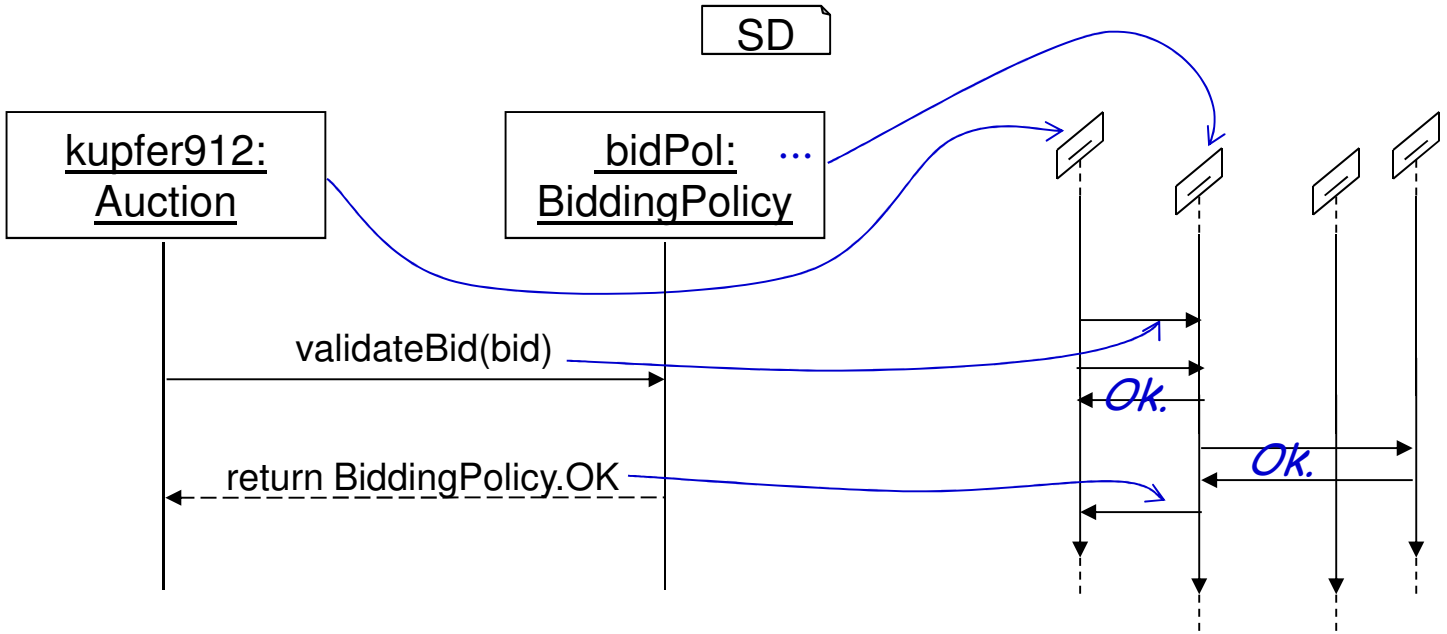
Fast vollständige Interaktionsliste

- Beispiel: Alle im Zeitraum der Beobachtung stattfindenden Interaktionen eines Objekts mit anderen sichtbaren Objekten sind enthalten, dh. es gab keine weiteren.
- Markierung «match:visible» verbietet andere Interaktionen zwischendurch mit sichtbaren Objekten:



Unvollständige Interaktionsliste

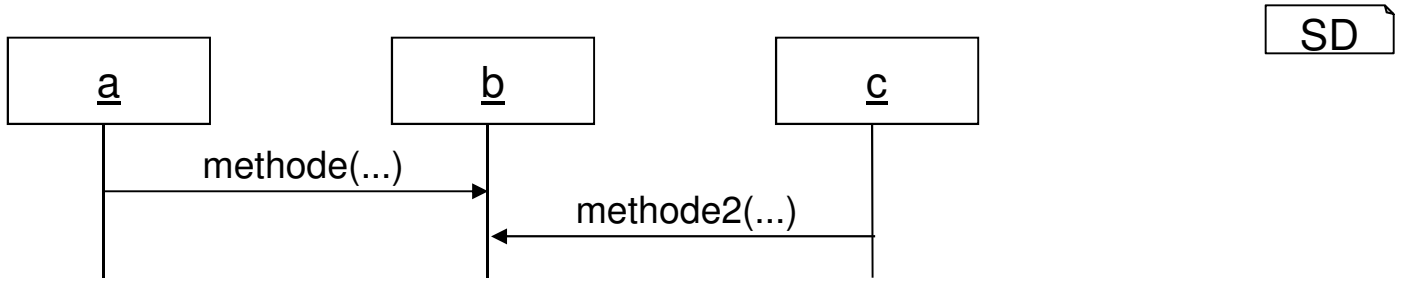
- Beispiel: Beliebige andere Interaktionen sind möglich
- Markierung «match:free» (kurz: ...) erlaubt alle anderen Interaktionen auch zwischendurch



Spezialfälle und deren Semantik ...

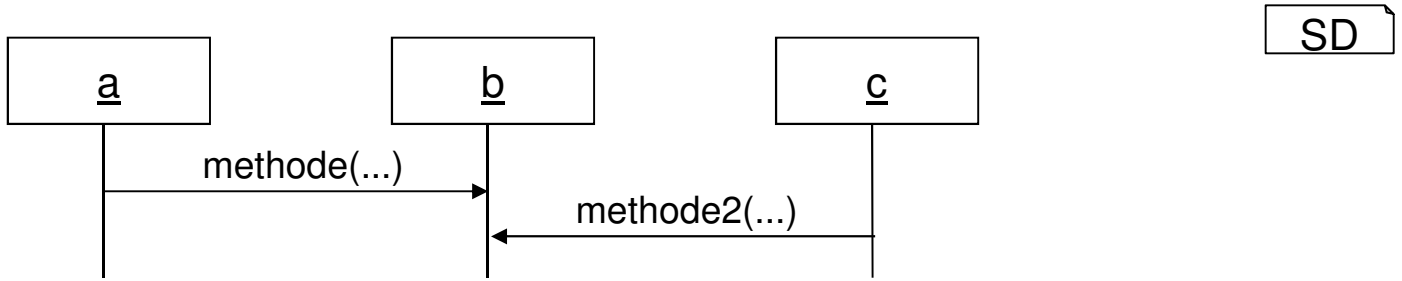
- Nicht-kausales SD ist ein SD, bei dem die Wirkzusammenhänge (Kausalität) nicht geklärt sind.
- Ist das SD sinnvoll? Was bedeutet dieses SD?

Analyze



Spezialfall: nicht-kausales SD

- Nicht kausales SD ist ein SD, bei dem die Wirkzusammenhänge (Kausalität) nicht geklärt ist.

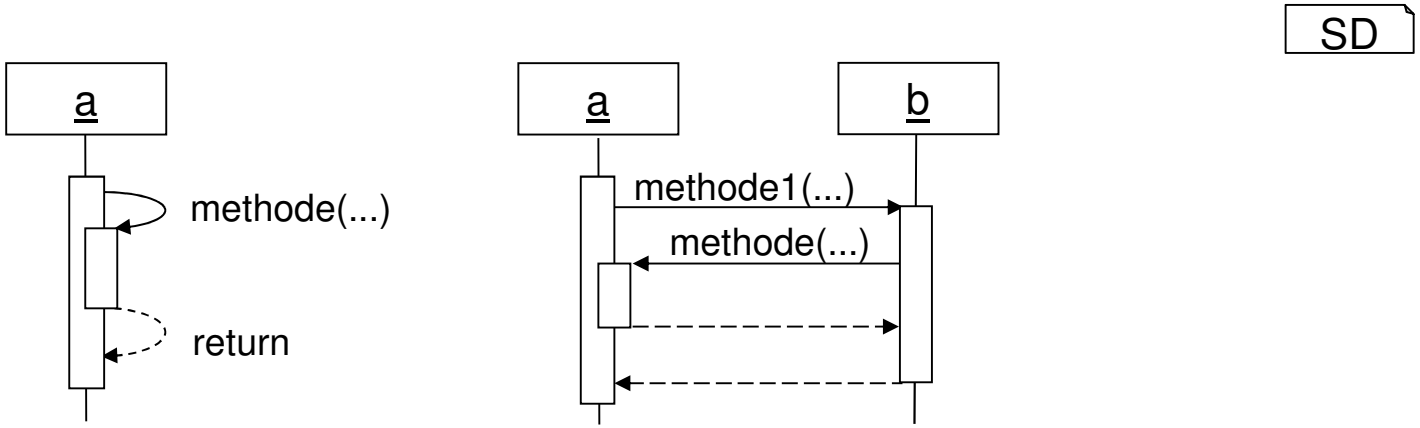


- Bedeutung:
  - nicht-kausale aber mögliche Beobachtung, zum Beispiel aufgrund eines nicht angegebenen Aufrufs von a nach c (oder von b nach c)
- SD ist nicht zur konstruktiven Codegenerierung (der Methodenrümpfe) verwendbar, da essentielle Information fehlt

Objektrekursion im SD

- Methodenrekursion: Die selbe Methode wird mit anderen Argumenten oder anderem Objekt wieder aufgerufen.
- Objektrekursion: Das selbe Objekt wird nochmals aufgerufen.

- Darstellung von Objektrekursion:

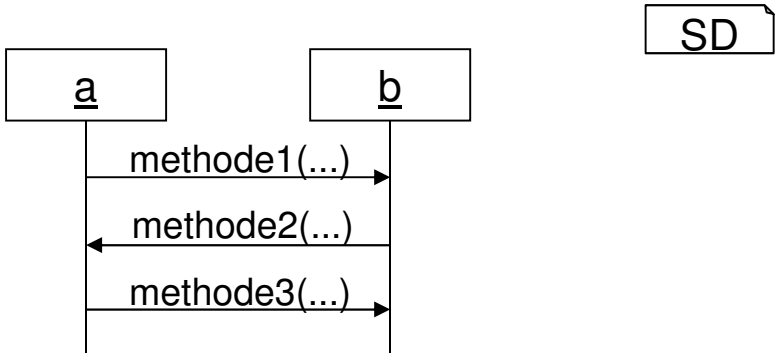


(a) Direkte Objektrekursion

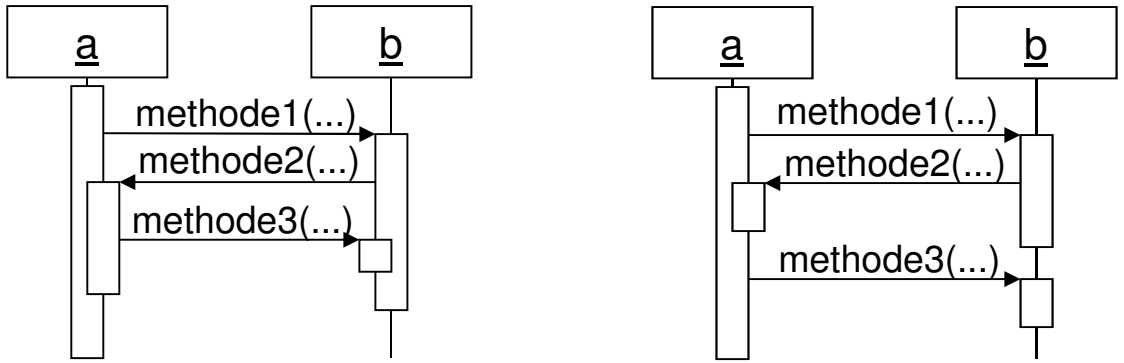
(b) Indirekte Objektrekursion

Mehrdeutigkeit

- Bedeutung dieses SD?



- SD ist als Beschreibung einer Beobachtung korrekt.
- Allerdings ist die Beobachtung unpräzise bzw. unvollständig (einige Details interessieren nicht)
- So gibt es mehrere Detaillierungen (hier durch Aktivitätsbalken dargestellt):



Methodischer Einsatz von SD

- Ein SD kann eine Beobachtung sein:
  - generiert aus einem Ablaufprotokoll für „Debugging“
  - Vorgegeben durch die Analyse

- Ein SD kann eine konstruktive Beschreibung eines notwendigen Ablaufs sein:
  - Voraussetzung: Eindeutiger Ablauf ohne Alternativen!
  - Da dies selten ist: konstruktive Codegenerierung aus SD wird nicht weiter betrachtet.

- Ein SD kann als Testtreiber verwendet werden:
  - «trigger» ist Initiator des Tests,
  - Rest ist eine Beobachtung

Zusammenspiel: SD und Statecharts

- Reihenfolgen manueller Erstellung:
  - SD als Analyse-naher Beschreibung aus denen Statcharts entwickelt werden
  - Statecharts werden analysiert durch Review konkreter Abläufe (SD)
    - Simulation der Statecharts (--> nahe an Codegenerierung und Ablaufanalyse)
  - SD und Statecharts werden als zwei Sichten des Systems unabhängig voneinander entwickelt und auf Konsistenz geprüft
    - durch entsprechende Vergleichstechniken (Signaturen, Aufrufreihenfolgen, etc.): Verwandt mit String-Erkennung endlicher Automaten
    - oder durch Codegenerierung aus Statecharts, Testfallgenerierung aus SD
- Viertiefende Literatur: Ingolf Krüger, LSC von D. Harel, et. al.

SE SOFTWARE ENGINEERING

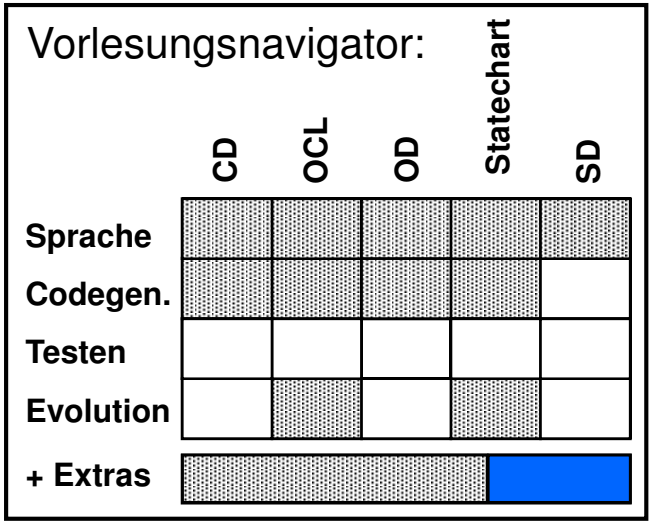
RWTH AACHEN

Modellbasierte Softwareentwicklung

- 7. Evolutionäre Methodik
- 7.1. Vorgehensmodell

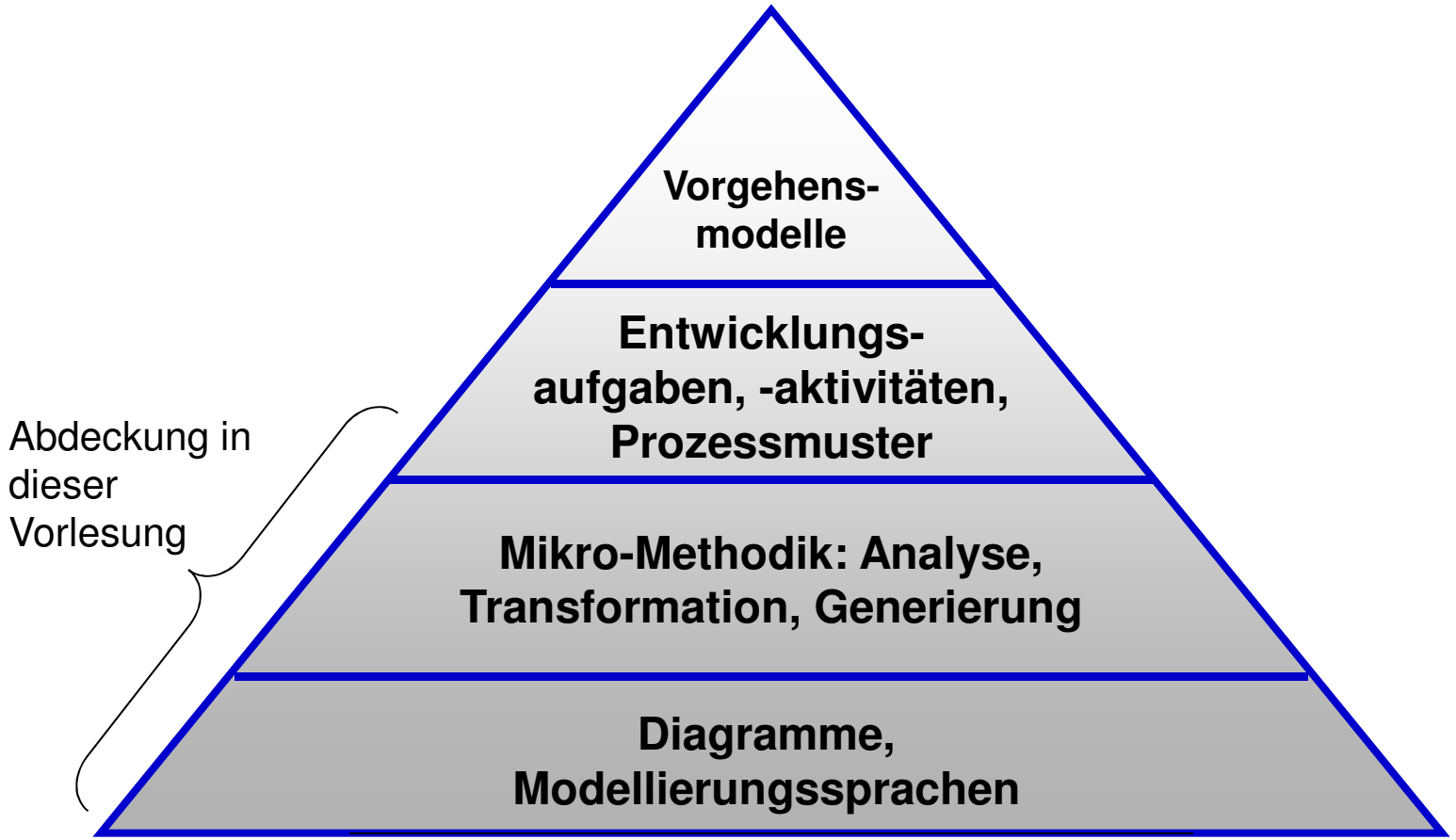
Prof. Dr. Bernhard Rumpe
Lehrstuhl für Software Engineering
RWTH Aachen

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



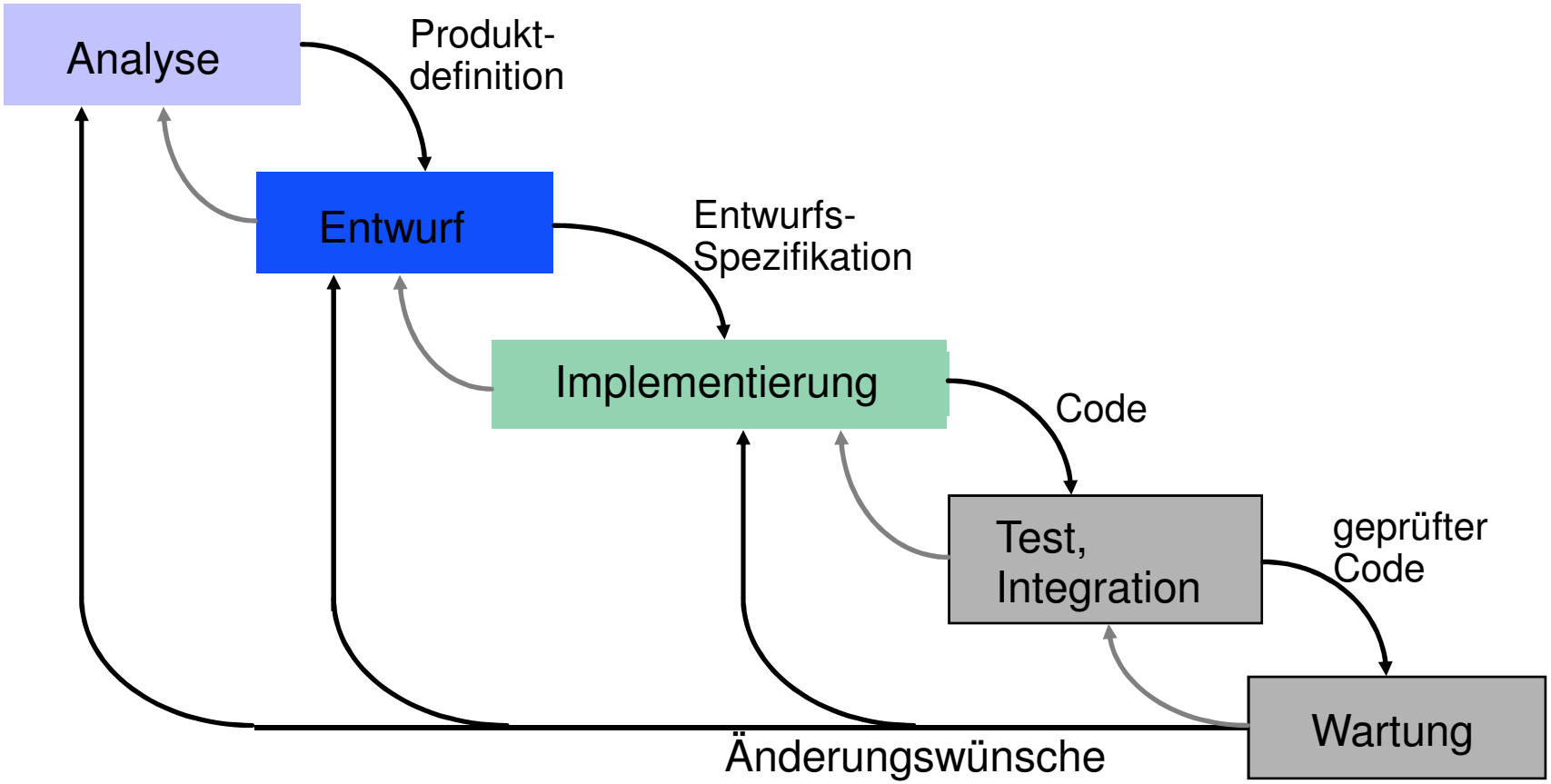
Grundlagen der Modellbildung

- Die Methodik-Pyramide:



Wasserfall-Modell

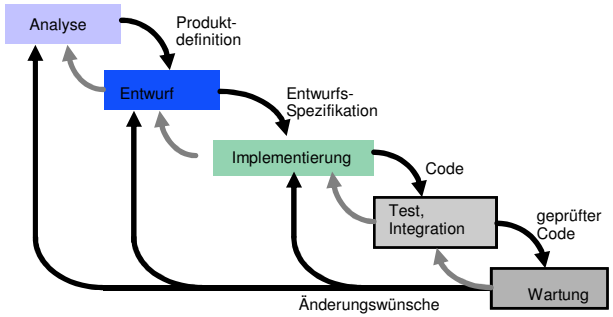
- ... als das der Urprototyp für Vorgehensmodelle:
- Alle anderen Modelle sind Derivate:



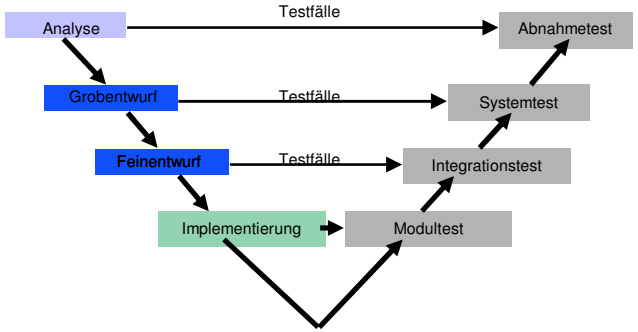
W. Royce (1970)

Variationen von Methoden

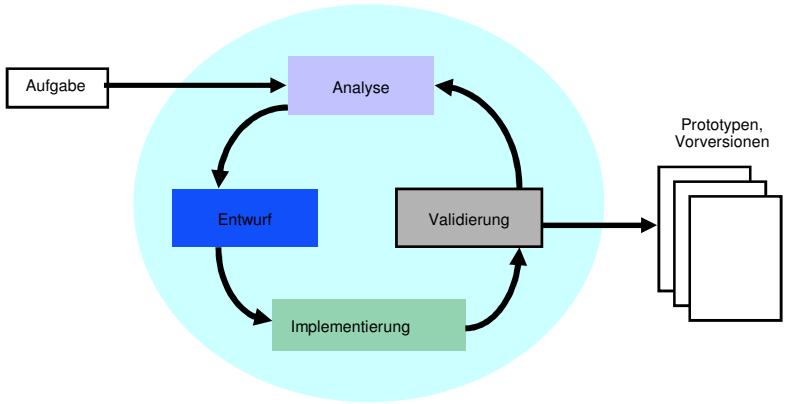
- Wasserfallmodell



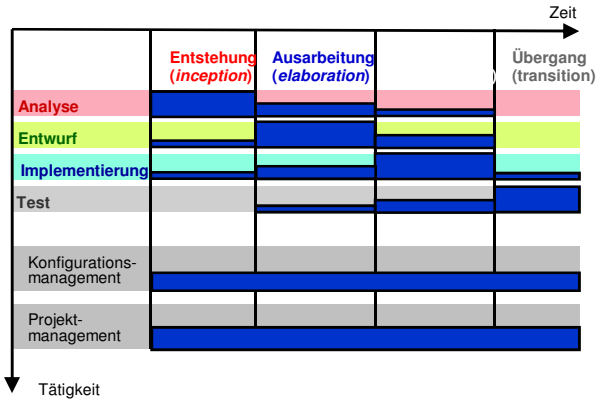
- V-Modell



- Evolutionäres/Inkrementelles Modell



- RUP



- Extreme Programming
- ... Details in Balzert: SE oder Vorlesung SE 2

eXtreme Programming (XP)

- Entwicklungsmethodik für kleinere Projekte
- Konsequente evolutionäre Entwicklung in sehr kleinen Inkrements
- Tests + Programmcode sind das Analyseergebnis, das Entwurfsdokument und die Dokumentation.
- Code wird permanent lauffähig gehalten
- Diszipliniertes und automatisiertes Testen als Qualitätssicherung
- Paar-Programmierung als QS-Maßnahme
- Refactoring zur evolutionären Weiterentwicklung
- Codierungsstandards

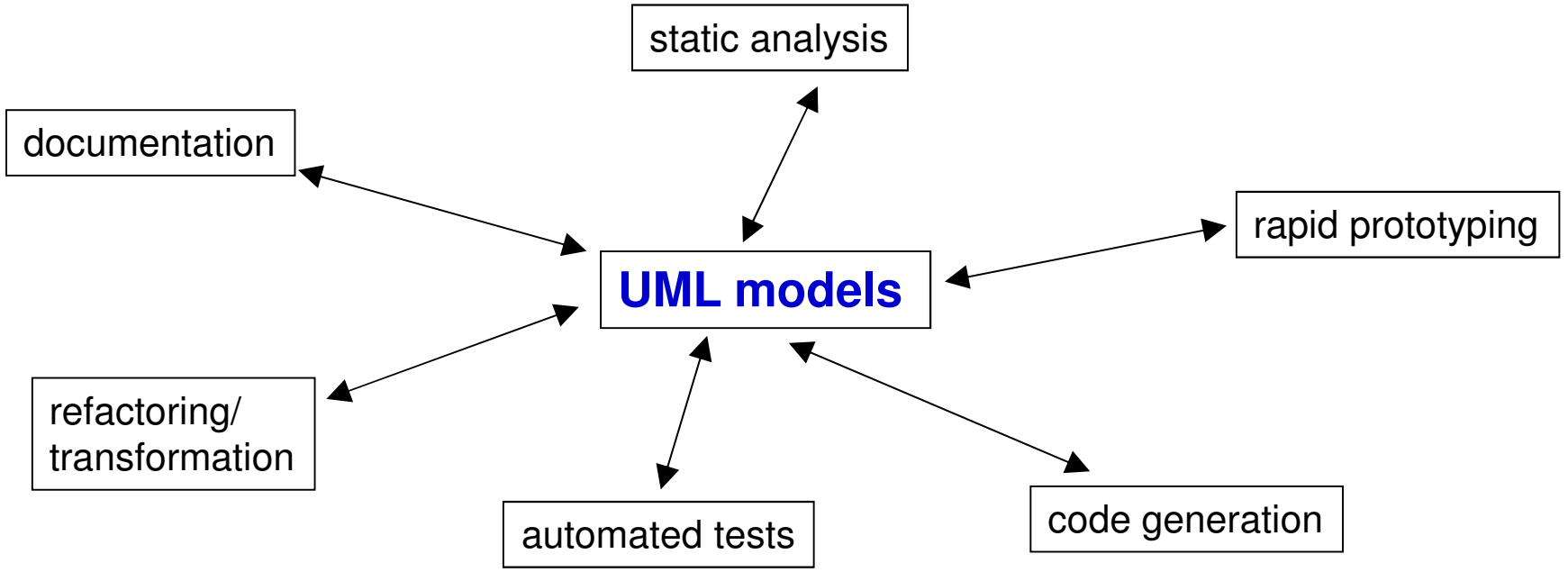
- Aber auch: Weglassen von traditionellen Elementen
  - kein explizites Design, ausführliche Dokumentation, Reviews

- „Test-First“-Ansatz
  - Zunächst Anwendertests definieren, dann den Code dazu entwickeln

Wdh. aus Kapitel 1

Modellbasierte Entwicklung mit der UML

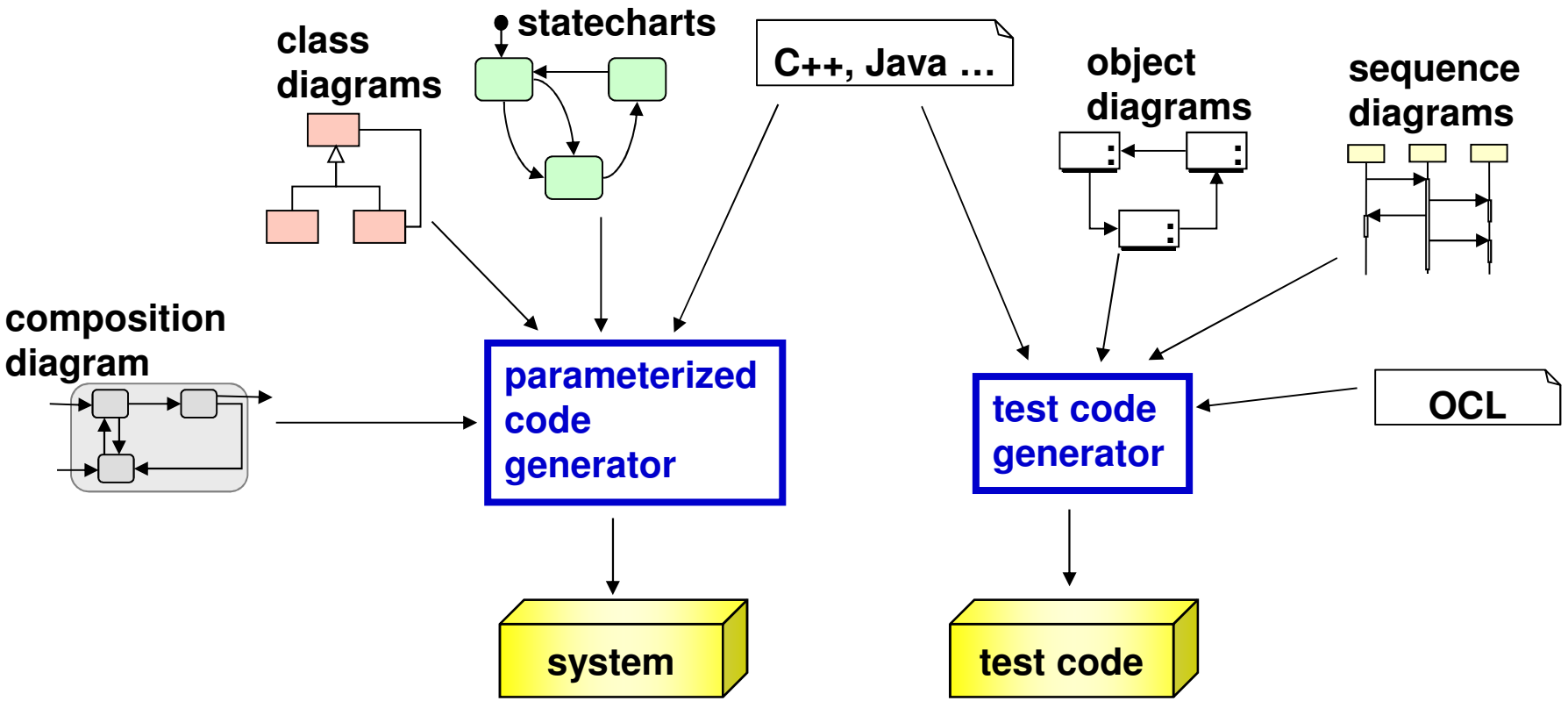
- Models as central notation



- UML serves as central notation for development of software
- UML is programming, test and modelling language at the same time

UML-basierte Modellierung

- UML + Code-Rümpfe erlauben Code & Test-Modellierung



⇨ Code- und Testmodelle prüfen gegenseitige Korrektheit

Kernelemente einer evolutionären Methodik

- Inkrementell Architektur, Funktionalität und Tests entwerfen
- Automatisierte Analysen:
  Typisierung, Datenfluss, Kontrollfluss, Testüberdeckung, ...
- Codegenerierung für System und automatische Tests

- Modell-basierte, automatisierte Tests
  - als Qualitätssicherung für Anwenderwünsche
  - für Regressionstests bei Änderungen

- Viele Releases mit eher kleinen Erweiterungen
- Häufige Simulation für den Kunden: Feedback
- Verfügbare Kunden/Domänenexperten

- Evolutionäre Transformationen zur inkrementellen Erweiterung

⇨ Methodik integriert Elemente moderner Entwicklungsmethoden:
Modell-basierung, „Evolution im Kleinen“, Test-Orientierung, Agilität

SE SOFTWARE ENGINEERING

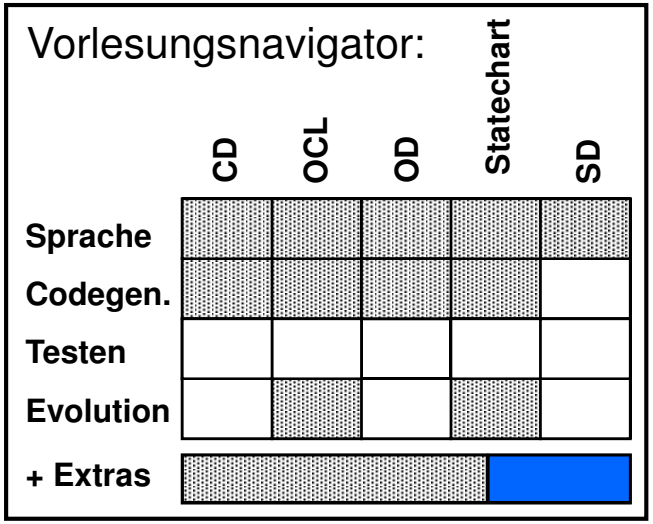
RWTH AACHEN

Modellbasierte Softwareentwicklung

- 7. Evolutionäre Methodik
- 7.2. Testen und Evolution

Prof. Dr. Bernhard Rumpe
Lehrstuhl für Software Engineering
RWTH Aachen

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



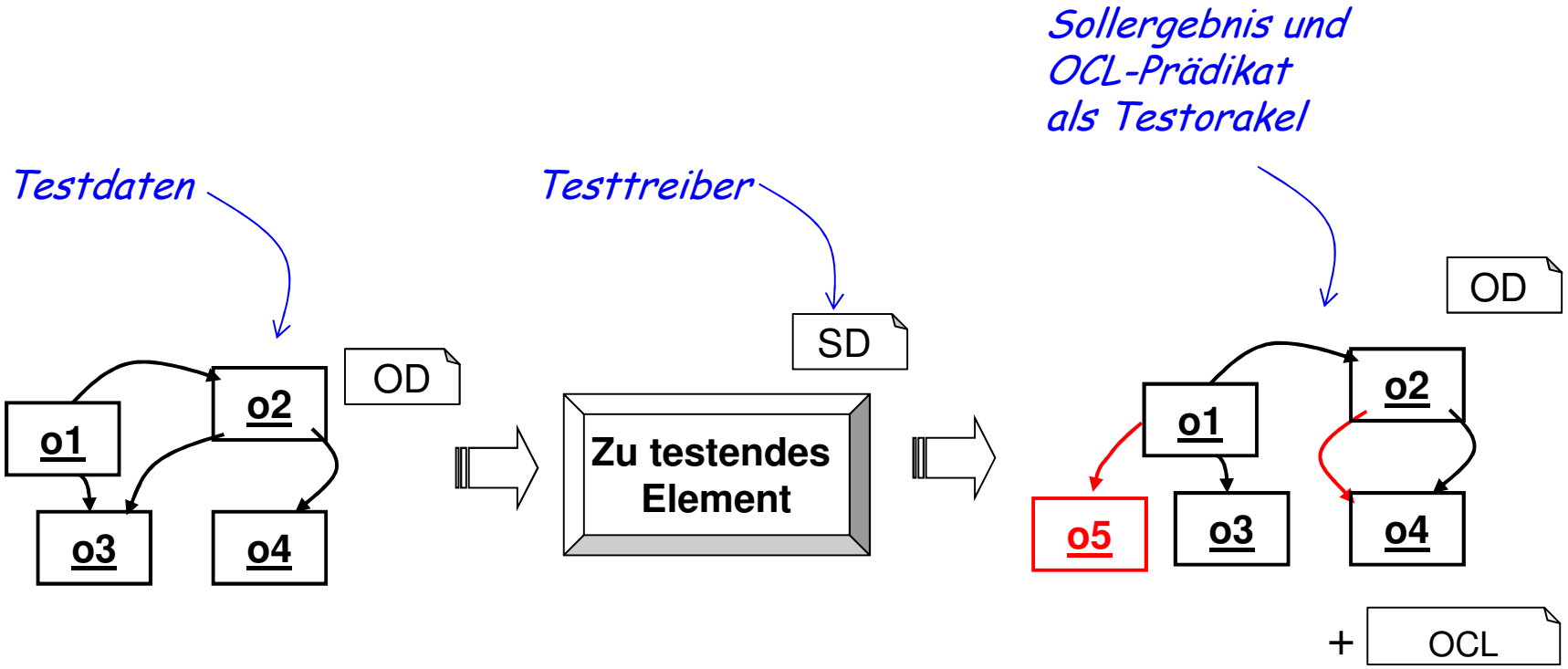
Tests

- Anforderungen an Tests:
  - Automatische Tests um Wiederholbarkeit zu sichern
    - Aufsetzen des Tests, Durchführung, Evaluation des Testergebnisses
  - Deterministische Tests mit determiniertem Ergebnis
  - Keine (bleibenden) Seiteneffekte
  - Effiziente Ausführung

- Ziele:
  - Demonstration der Qualität
  - Definition von Anforderungen noch zu realisierender Funktionalität
  - Grundlage für das Zutrauen in die Korrektheit des Codes
  - und Grundlage für die effektive Weiterentwicklung des Systems

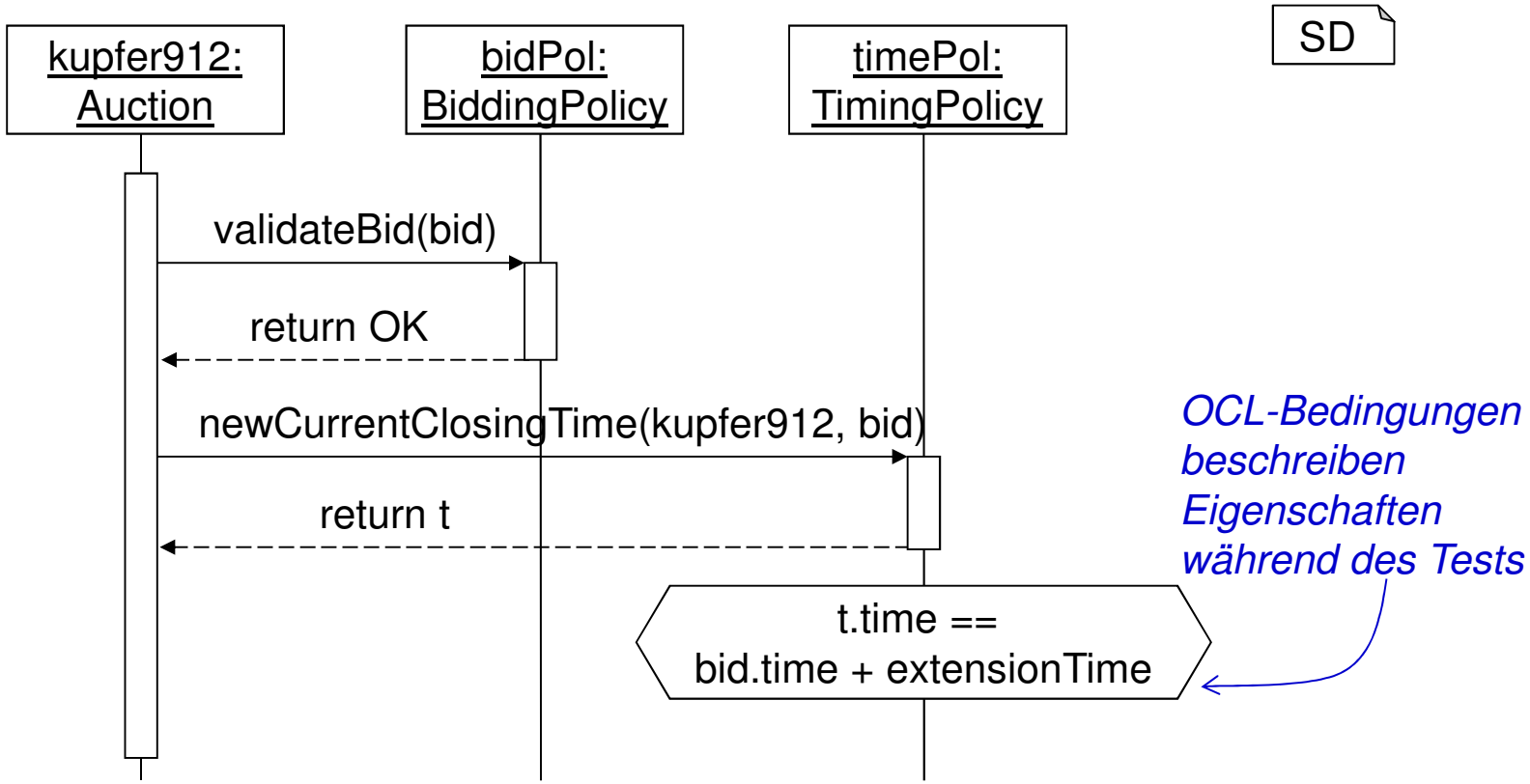
Testinfrastruktur (Einfache)

- Prinzip:
  - OD als Testdatensatz
  - weiteres OD und OCL als Orakel



Sequenzdiagramme (SD)

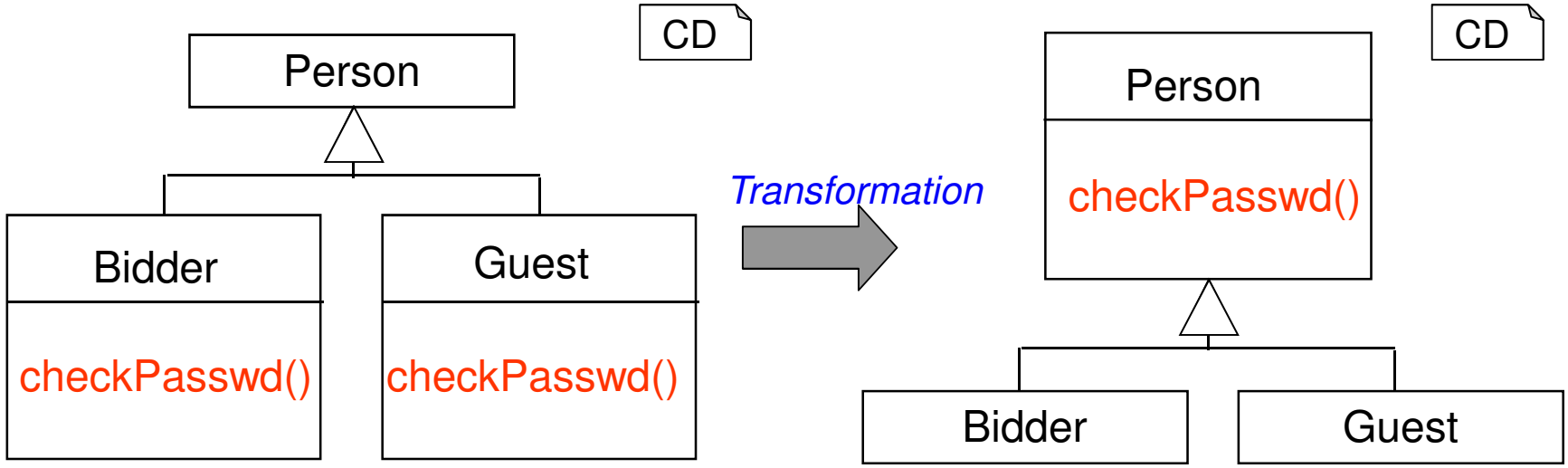
- lineare Struktur für exemplarische Beschreibung
- + integriertes OCL
- SD sind als Testprädikate oder –treiber einsetzbar
- SD können aus Statecharts (teilweise) generiert werden



Evolution von Modellen

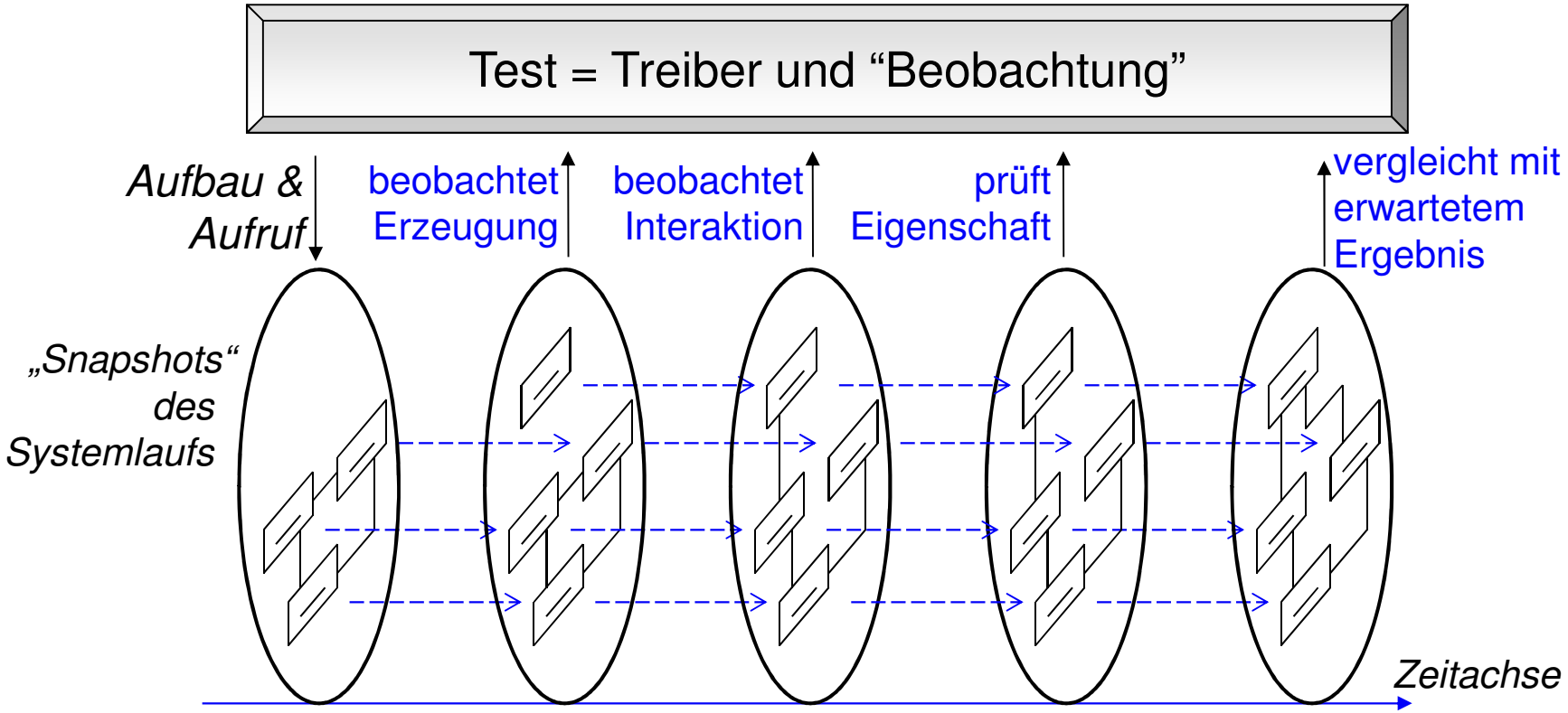
- Ziel ist die systematische Transformation eines Modells zur
- Verbesserung der Struktur/Architektur eines Systems unter
- Beibehaltung des beobachteten Verhaltens.

Beispiel: Methode aus zwei Subklassen generalisieren



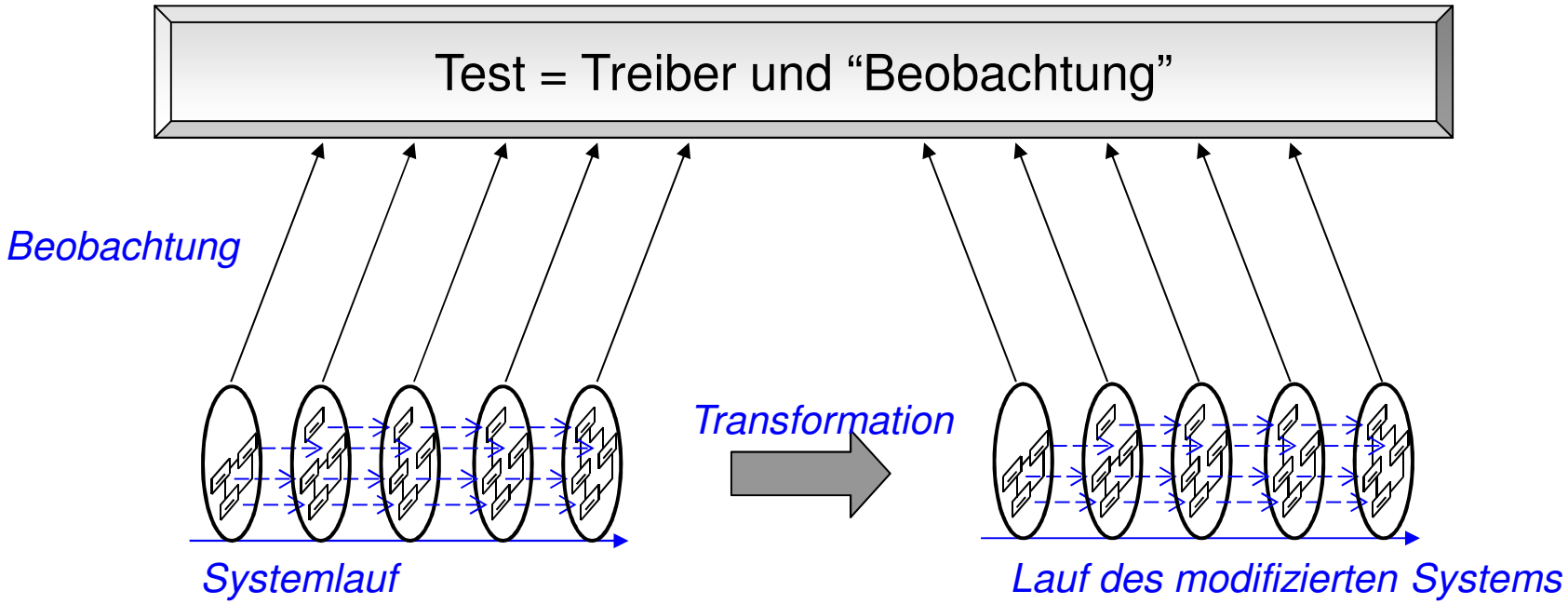
Tests sind Beobachtungen für Transformationen

- Tests beobachten Struktur und Verhalten:



Validierung von Transformationen

- Die Testbeobachtung bleibt unter der Transformation erhalten



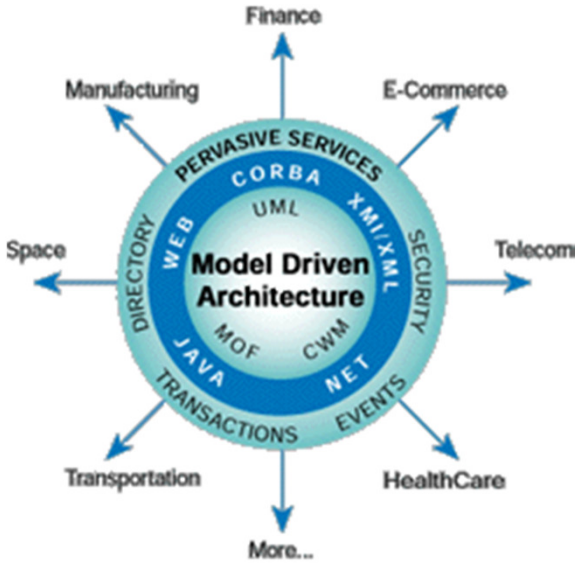
- Aber in der Praxis: oft ändern sich Strukturteile unter der Transformation
- Deshalb: Akzeptanztests auf geeignete Abstraktionen und fixierte Schnittstellen basieren

SE SOFTWARE ENGINEERING

RWTH AACHEN

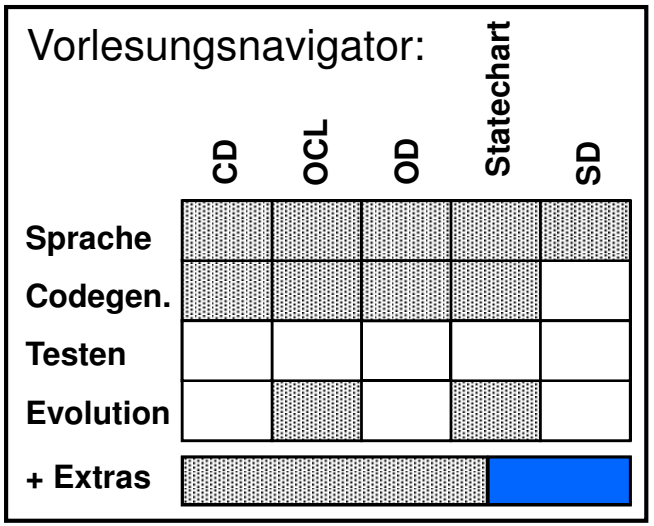
Modellbasierte Softwareentwicklung

- 7. Evolutionäre Methodik
- 7.3. Model Driven Architecture (MDA)



Prof. Dr. Bernhard Rumpe
Lehrstuhl für Software Engineering
RWTH Aachen

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



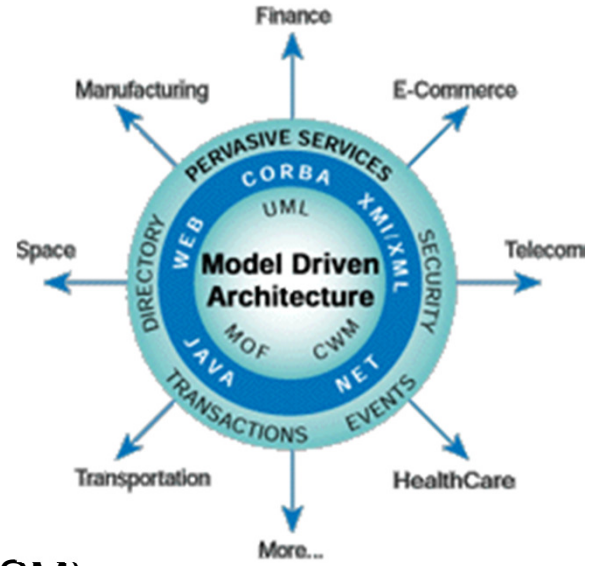
Object Management Group (OMG)

- Konsortium mehrer hundert Unternehmen
- Ziele: Korpus von de‘facto Standards
- Beispiele: Corba, UML - derzeit: MDA

- Aber auch domänenspezifische Standards:
  Health care, transportation, finance, ...
- OMG ist kein Standardisierungsgremium, aber mächtig!
- Mehr über die OMG: www.omg.org

Model Driven Architecture (MDA)

- ein weiterer Meilenstein der OMG



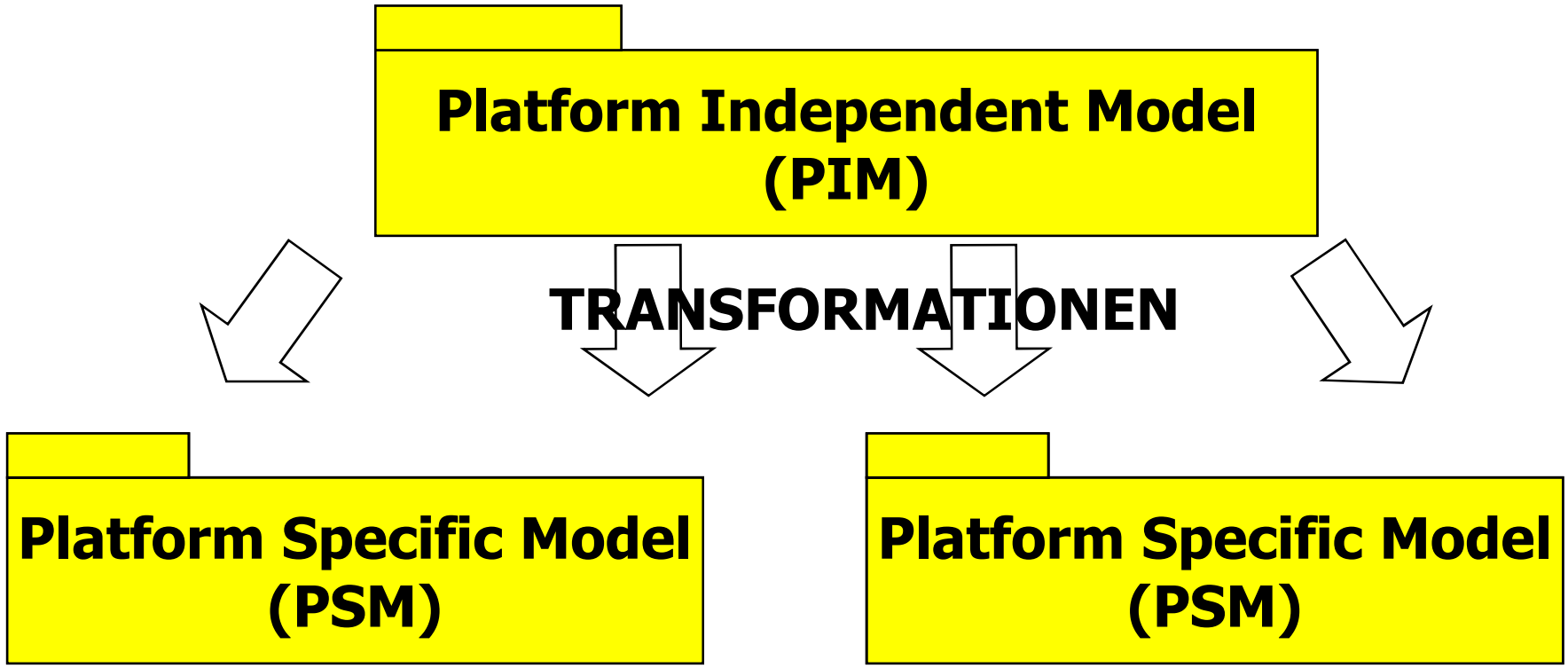
- Modelle sind Zentrum der Entwicklung
  - Middleware ist nicht mehr so wichtig.
- Fokus: plattformunabhängige Modelle
  - = Platform Independent Models (PIM)
  - Modelle also z.B. ohne Middleware-Details
- Sowie: Abstrakte plattformspezifische Modelle (PSM)
  - bei denen z.B. Middleware-Details integriert sind
- Ziele:
  - Kompakte, wiederverwendbare PIM-->PSM-Transformationen
  - PIM wird ebenfalls wiederverwendet, wenn Technologie wechselt!

Derzeit behandelte MDA Technologien

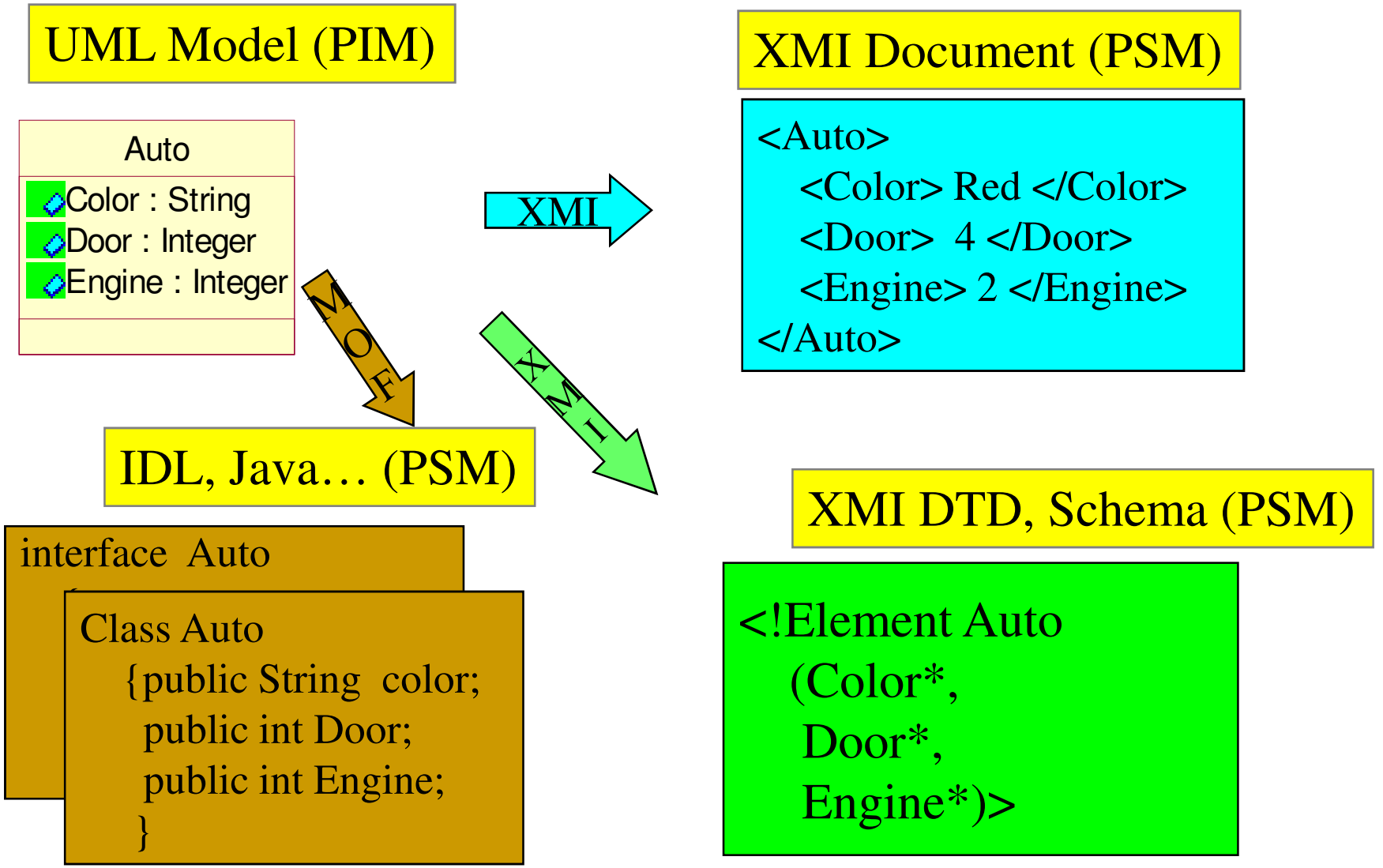
- Meta Object Facility (MOF)
- Unified Modeling Language (UML)
- XML Model Interchange (XMI)
- Common Warehouse Meta-model (CWM)
- Software Process Engineering Meta-model (SPEM)
- eine Reihe von UML-Profilen
- QVT – Query, View, Transformation:
  - 2003: Request for Proposals
  - 2008: Version 1.0
- . . .
- weitere Technologien werden integriert werden

Kern der MDA: PIM, PSM und Transformationen

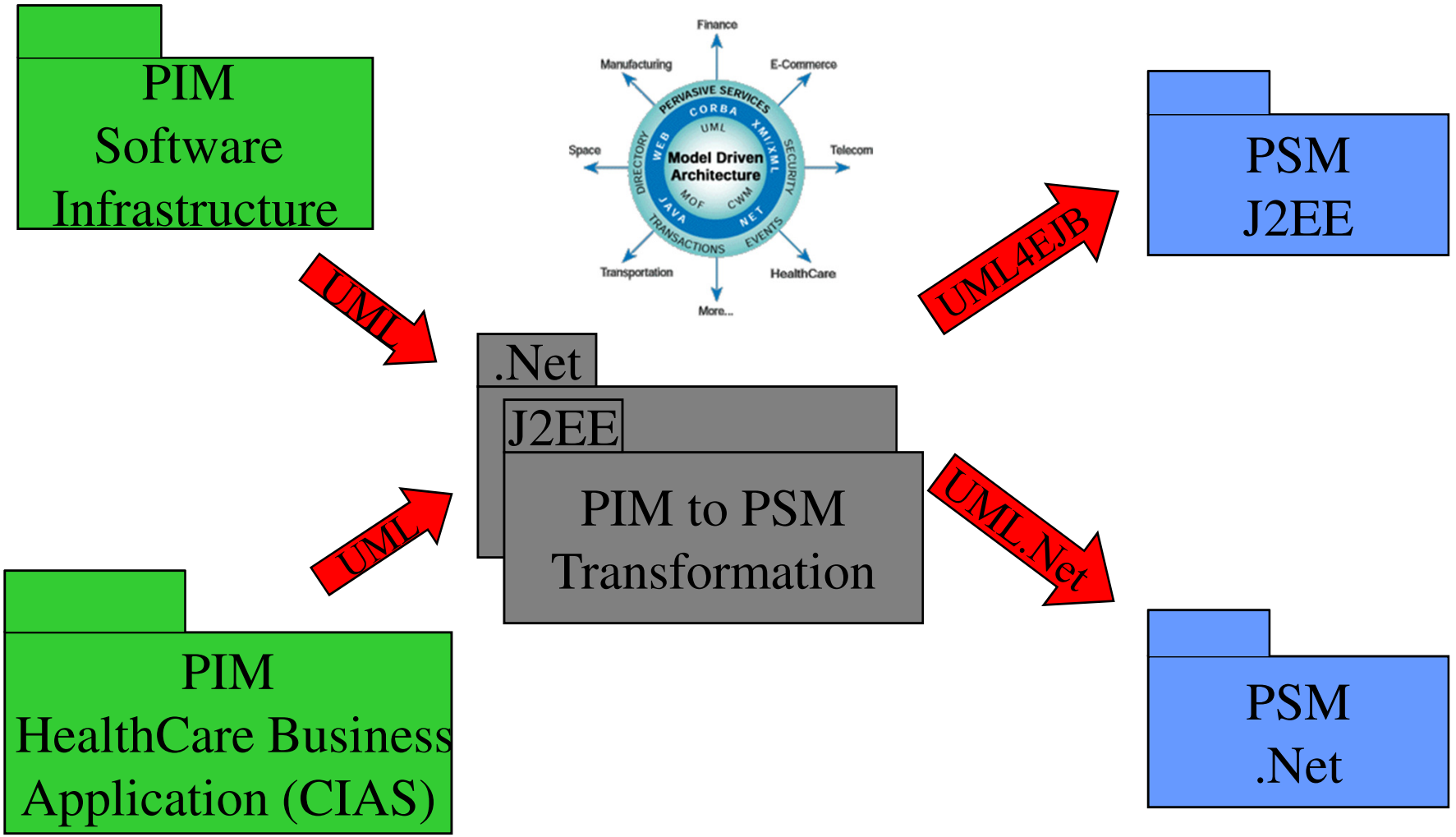
- Plattform Independent Model (PIM)
  - abstrakt, unabhängig von Details der Middleware etc.
- Plattform Specific Model (PSM)
  - Technologie-befrachtet, Plattform-Optimierungen, etc.



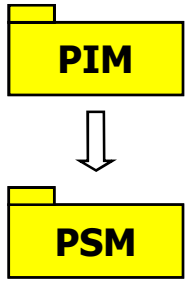
PIM --> PSM-Transformationen: MOF/XMI – basiertes Beispiel



PIM --> PSM-Transformationen: J2EE, .Net-basierte Beispiele



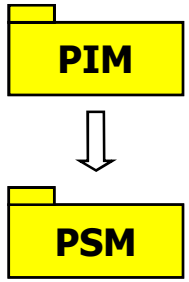
Ziele von MDA



- Wiederverwendung:
  - „Separation of concerns“ in die PIM und die Transformation erhöht Wiederverwendung:
  - PIM ist wiederverwendbar bei einer Transition zu neuer Technologie
  - Transformation ist wiederverwendbar für andere Applikationen auf der selben Plattform
- Produktivität:
  - wird besser durch Wiederverwendung
- Portabilität:
  - leichtere Portierbarkeit der technologieunabhängigen PIM
- Interoperabilität:
  - erzeugung mehrerer PSM für unterschiedliche Plattformen und Bridges erhöht die Interoperabilität
- Wartung und Dokumentation:
  - PIM eignet sich als Dokumentation genau so wie als Quelle von Evolution

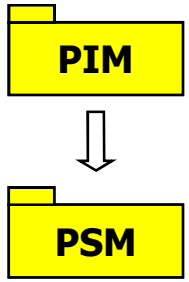
MDA: Aktueller Stand

- Es gibt sehr erfolgreiche MDA-Beispiele!



- MDA forciert die Standardisierung, obwohl die zugrundeliegenden Techniken nicht immer ausgereift sind.
- Schwierige Suche nach guten und effektiven Transformationssprachen
  - Metamodellierung, Graphtransformationen
- Werkzeuge entstehen
  - Meist manuell erzeugte, firmenspezifische Werkzeuge, XML-basiert
  - Werkzeughersteller bieten Standardtransformationen (Codegenerierung)
- Es gibt noch keinen Nachweis, dass MDA seine Versprechen erfüllen wird,
- aber erfolgreiche Beispiele und die Macht der OMG demonstrieren, dass MDA ernstzunehmen ist.

Wesentlicher Teil der MDA: Transformationen



- Viele Varianten von Transformationen:
  - bidirektional?
  - abstrahierend (vergesslich)?
  - detaillierend (Details hinzufügend)?
  - semantikerhaltend / verfeinernd / abstrahierend?
  - innerhalb oder zwischen Sprachen?

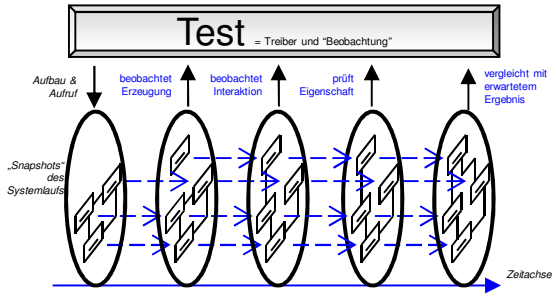
- Innerhalb einer Sprache sind Transitionen verwendbar für:
  - Verfeinerung / Abstraktion
  - oder Evolution

- Nachfolgend ein Beispiel:
  - Modellbasierte Evolution von Architekturen

SE SOFTWARE ENGINEERING

RWTH AACHEN

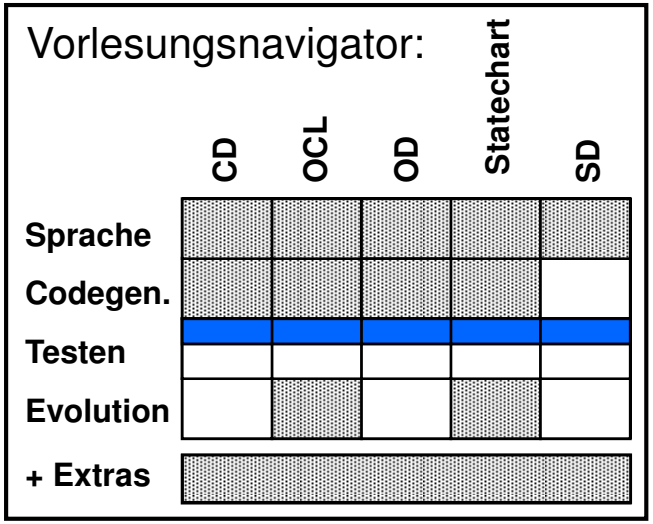
Modellbasierte Softwareentwicklung



- 8. Testen
- 8.1. Einführung

Prof. Dr. Bernhard Rumpe
Lehrstuhl für Software Engineering
RWTH Aachen

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



Was ist Testen?
Einige Definitionen ...

- Testen ist der Prozess, ein Programm mit der Absicht auszuführen, Fehler zu finden. (Myers: Testen ’79)

- Testen von Software ist die Ausführung der Softwareimplementierung auf Testdaten und die Untersuchung der Ergebnisse und des operationellen Verhaltens, um zu prüfen, dass die Software sich wie gefordert verhält. (Sommerville: Software Engineering ’01)

- Die Anwendung von Test-, Analyse- und Verifikationsverfahren dient im wesentlichen zur Überprüfung der Qualitätseigenschaften funktionale Korrektheit und Robustheit. (Liggesmeyer: ’90)

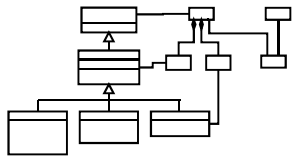
- Unter Testen versteht man den Prozeß des Planens, der Vorbereitung und der Messung, mit dem Ziel, die Merkmale eines IT-Systems festzustellen und den Unterschied zwischen dem aktuellen und dem erforderlichen Zustand nachzuweisen.

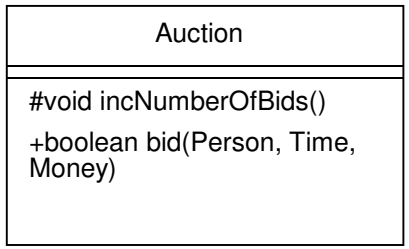
Charakteristika von Tests

- Ein Test lässt das zu testende System ablaufen.
- Tests sind idealerweise automatisiert.
- Ein automatisierter Test führt den Aufbau der Testdaten, den Test und die Prüfung des Testergebnisses selbständig durch. Der Erfolgsfall beziehungsweise das Scheitern werden durch den Testlauf erkannt und gemeldet.
- Eine Sammlung von Tests bildet selbst ein Softwaresystem, das gemeinsam mit dem zu prüfenden System abläuft.
- Ein Test ist exemplarisch.
- Ein Test ist wiederholbar und determiniert.
- Ein Test ist zielorientiert.
- Test kann bei einem modifizierten System exemplarisch die Verhaltensgleichheit mit dem Ursprungssystem nachweisen.

Testebenen

Testart	Wer erstellt den Test (oder führt ihn durch)?	Testling
Abnahmetest	Anwender, meistens interaktiv	Das installierte Produktionssystem
Systemtest („Akzeptanztest“)	Testteam mit Hilfe von Anwendern	Das instrumentierte Produktions-system in der Testumgebung
Subsystemtest („Integrationstest“)	Testteam, Entwickler	Subsystem
Klassentest („Modultest“, „Unittest“)	Entwickler, Testteam	Klasse
Methodentest	Entwickler	Methode





Effekte von automatisierten Tests

- Zutrauen der Entwickler in den eigenen Code sowie den Code von Kollegen signifikant höher.
- Erhöhtes Selbstvertrauen eines Entwicklers fremden Code anzupassen.
- Wissen über die Systemfunktionalität in wiederholbaren Tests gespeichert.
- Ausführliche Sammlung von Testfällen und Systemspezifikation sind zwei Modelle des Systems.
- Gescheiterter Test dokumentiert eine Fehlerbeschreibung.
- Testaufwand bleibt beschränkt. Interaktive Regressionstests würden den wiederholten Testaufwand zu sehr vergrößern.
- Ausführliche Testsammlung dokumentiert die Qualität des Systems dem Kunden gegenüber.

Begriffsbestimmung: Fehler

- Versagen (engl.: failure) ist die Unfähigkeit eines Systems oder einer Komponente eine geforderte Funktionalität in den spezifizierten Grenzen zu erbringen. Versagen manifestiert sich durch falsche Ausgaben, fehlerhafte Terminierung oder nicht eingehaltene Zeit- und Speicher-Rahmenbedingungen.
- Mangel (engl.: fault) ist ein fehlender oder falscher Code.
- Fehler (engl.: error) ist eine Aktion des Anwenders oder eines Systems der Umgebung, das ein Versagen herbeiführt.
- Auslassung (engl.: omission) ist das Fehlen von geforderter Funktionalität.
- Überraschung (engl.: surprise) ist Code, der keine geforderte Funktionalität unterstützt und daher nutzlos ist.

Begriffsbestimung: Test - 1

- Testobjekt = System im Test, zu testendes System, Testling, Prüfling
- Testverfahren: Vorgehensweise zur Erstellung und Durchführung von Tests.
- Testdaten (engl.: test point): konkreter Satz von Werten für die Eingabe eines Tests, inclusive Objektstruktur und zu testenden Objekten.
- Test-Sollergebnis: das erwartete Ergebnis eines Tests.
- Testfall (engl.: test case): Beschreibung des Zustands des zu testenden Systems und der Umgebung vor dem Test, den Testdaten und dem Test-Sollergebnis.

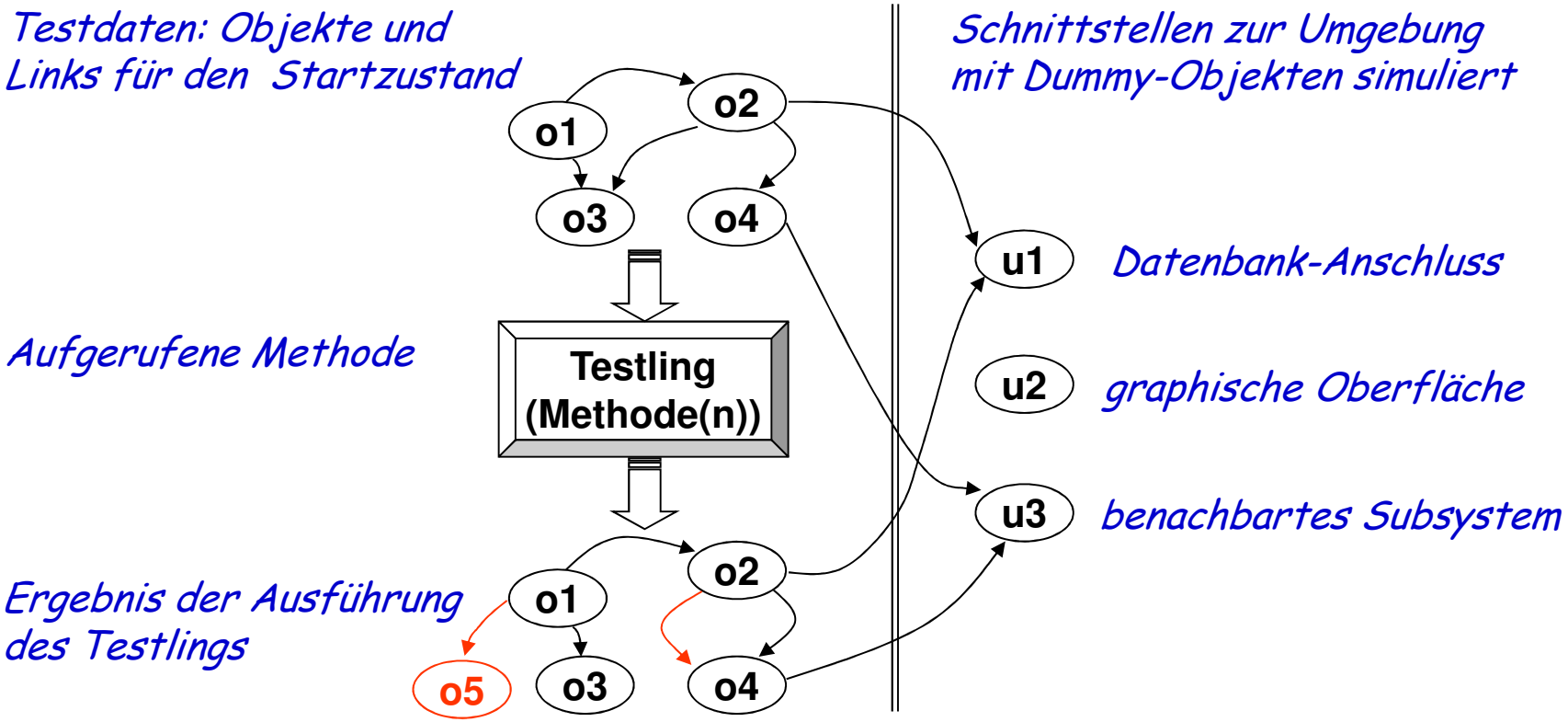
Begriffsbestimung: Test - 2

- Testsammlung (engl.: test suite): Menge von Testfällen.
- Testlauf (Testablauf, engl.: test run): Durchführung eines Tests mit tatsächlichen Ergebnissen (Test-Istergebnisse).
- Testtreiber organisiert den Testlauf vom Aufbau der Testdaten bis zur Prüfung des Testerfolgs.
- Testerfolg genau dann, wenn Istergebnis und Sollergebnis konform sind. Ansonsten ist der Test gescheitert.
- Testurteil (Verdikt): Aussage, ob Test erfolgreich war oder gescheitert ist.

UML als Test und Implementierungssprache

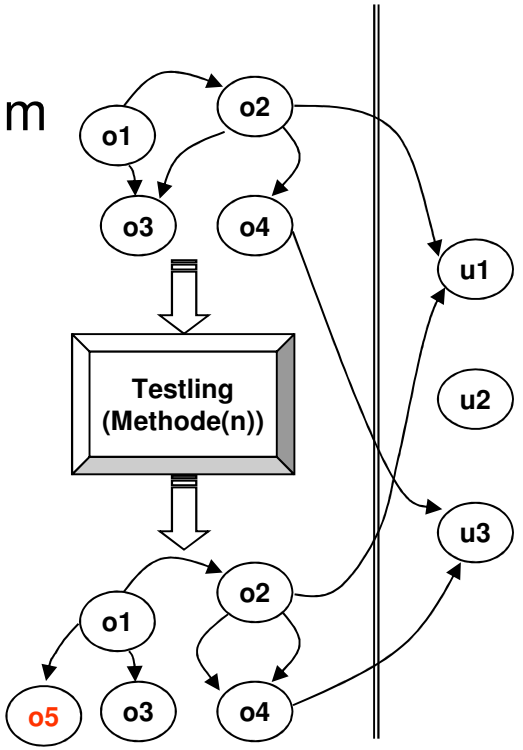
- Einsatzformen der UML
  - zur Codegenerierung
  - Testfalldefinition unter Nutzung von Diagrammen, z.B. SD, OD
  - Ableitung von (mehreren) Testfällen aus Diagrammen, z.B. Statecharts
  - Messung von Testfallüberdeckungen für Modelle, z.B. Statecharts
  - Fehlerquellen der UML bei Codeumsetzung prüfen, z.B. Multiplizitäten

Struktur eines Tests



Dummies, Mock-Objekte

- Kritisch:
  - Schnittstellen zur Umgebung des Testlings
- Diese werden mit Dummy-Objekten simuliert, um
  - (a) Seiteneffekte zu verhindern
  - (b) Zugriffe zu protokollieren
  - (c) vorbereitete Ergebnisse vorzuspiegeln
- Dummies ersetzen echte Umgebung:
  - Datenbank
  - Nachbarsysteme
  - GUI
  - aber auch: Nachbar-Objekte

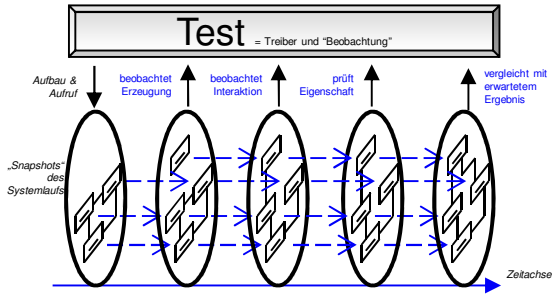


- Mock-Objekte sind Dummies mit etwas Intelligenz (z.B. Beantwortung von Methodenaufrufen)
- Standardverfahren: Dummy ist Objekt einer Subklasse mit (fast) leerer Implementierung

SE SOFTWARE ENGINEERING

RWTH AACHEN

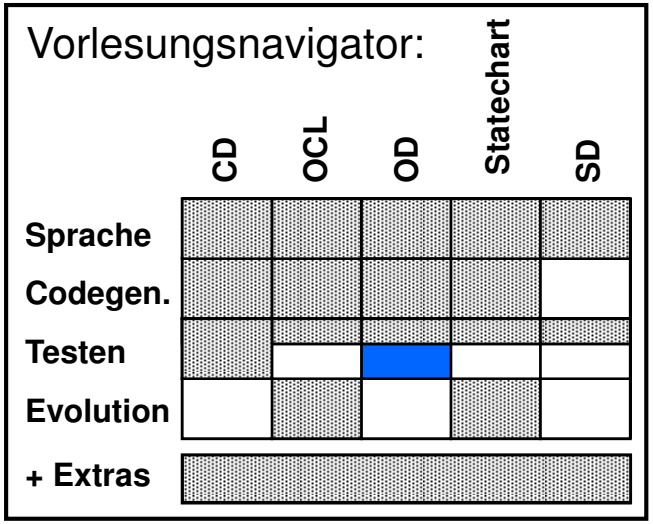
Modellbasierte Softwareentwicklung



- 8. Testen
- 8.2. Objektdiagramme modellieren Testdaten

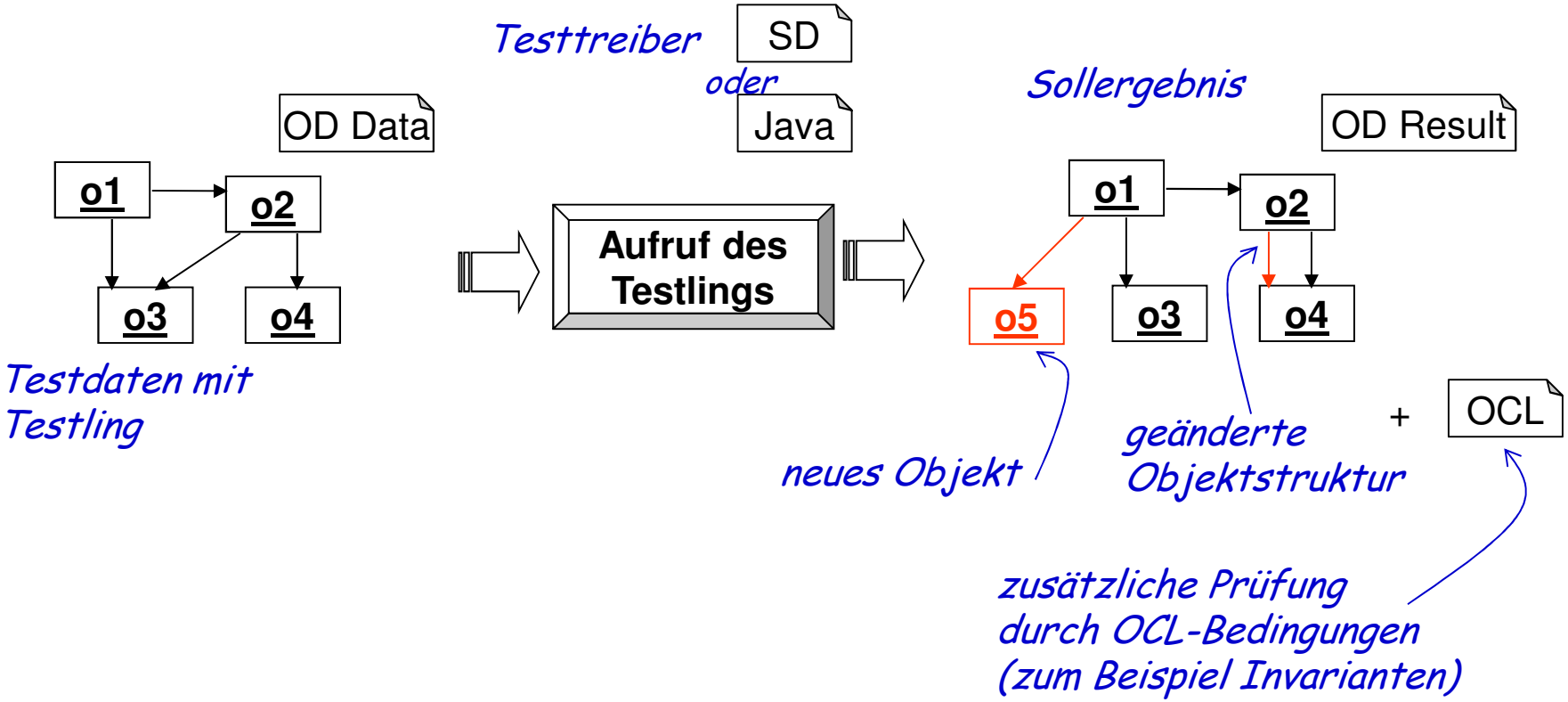
Prof. Dr. Bernhard Rumpe
Lehrstuhl für Software Engineering
RWTH Aachen

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



Modellierung eines Testfalls

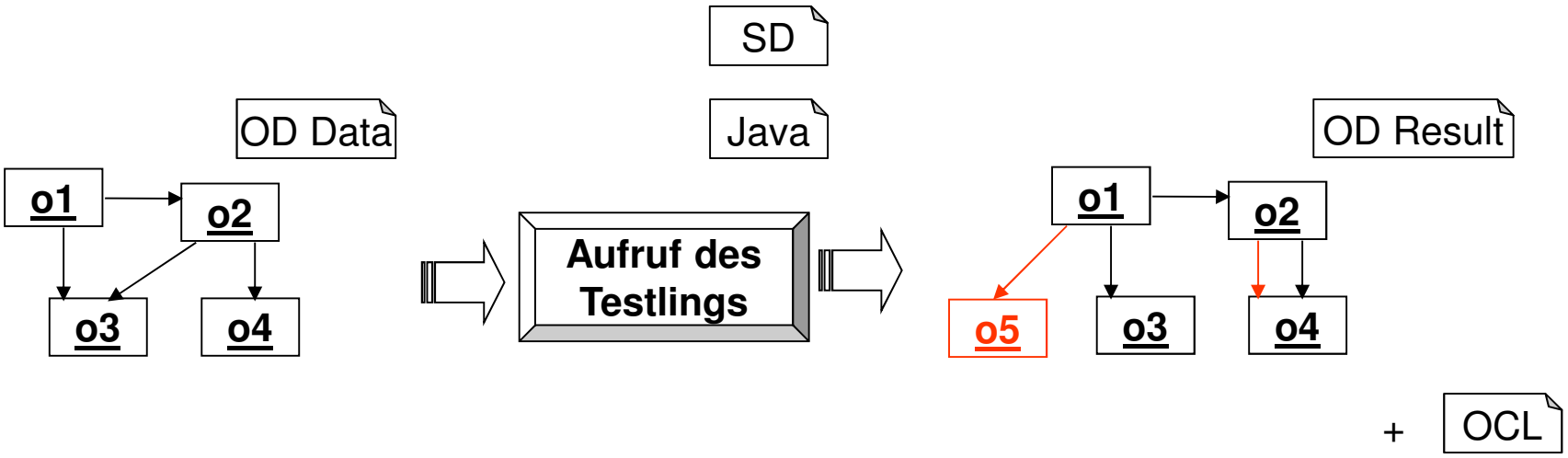
- unter Nutzung zweier Objektdiagramme, der OCL und eines Sequenzdiagramms (oder Java)
- Objektdiagramme modellieren Testdaten und Sollergebnis



Modellierung eines Testfalls – in der Praxis

- Testdaten oft relativ komplex, aber davon nur wenige notwendig. Wiederverwendbar mit Feintuning der Daten durch Java.
- Testtreiber oft sehr kompakt: einzelner Aufruf oder kurzes SD
- Ergebnis-OD braucht nur auf Unterschiede einzugehen und ist ebenfalls kompakt

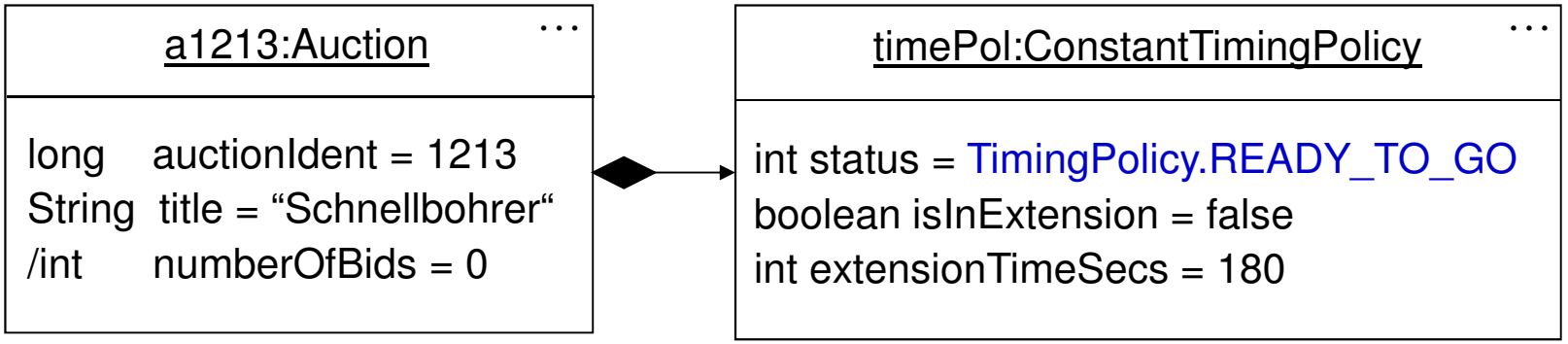
- Wiederverwendbarkeit einzelner Diagramme erlaubt effektive Testfalldefinition



Beispiel: Eröffnen einer Auktion - 1

- Ausgangssituation (vereinfacht):

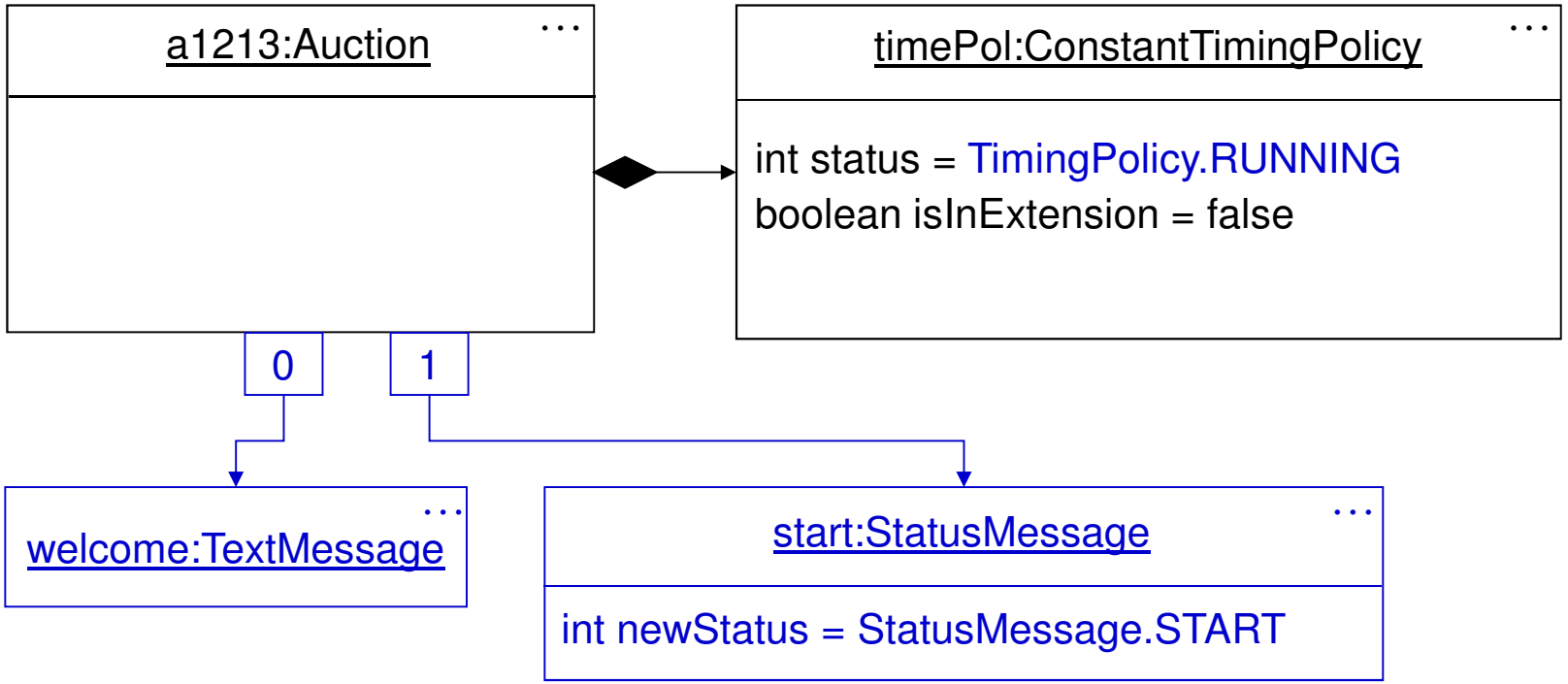
OD YetClosed
...



Beispiel: Eröffnen einer Auktion – 2

- Sollergebnis: Auktion läuft

OD Running



- und es soll gelten:
- context Auction a inv NoBidYet:
  { m in a1213.message | m instanceof BidMessage }.isEmpty

OCL

Der Test

- Die Beschreibung des Tests:

Test

  - testobject: Auction.start();
    testdata: OD YetClosed;
    driver: a1213.start();
    assert: OD Running;
    inv NoBidYet; inv Bidders1;

- der generierte Code

Java/P

  - testStart() {

```
Auction a1213 = setupYetClosed();   // Testdaten erzeugen
a1213.start();                      // Test ausführen
ocl isStructuredAsRunning(a1213);   // Sollergebnis erfüllt?
checkNoBidYet(a1213);               // Eigenschaft NoBidYet
checkBidders1(a1213);               // Invariante Bidders1
}
```

Zusätzliches Schlüsselwort ocl ist ähnlich dem assert in Java

- Die Diagramme und OCL werden wie besprochen umgesetzt.

Allgemeine Testfallstruktur

Aussehen eines Tests mit allen Elementen. Oft auch tabellarische Darstellung

test *Testobjekt* {

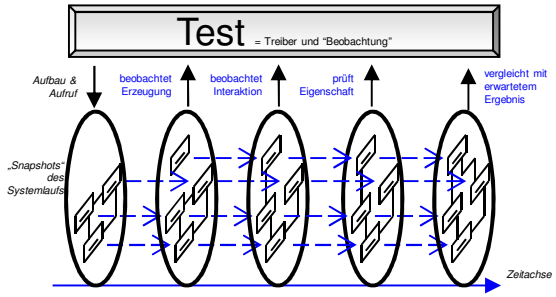
name:	*AuctionTest.testBid*
testdata:	*Objektdiagramme bereiten den Testdatensatz vor*
tune:	*Java-Code erlaubt Anpassung der Testdaten*
driver:	*Java-Methodenaufruf(e) oder Sequenzdiagramm*
methodspec:	*OCL-Methodenspezifikation geprüft bei Methodenaufruf*
interaction:	*Sequenzdiagramme als Ablaufbeschreibungen geprüft*
oracle:	*Java-Methodenaufruf oder Statechart produziert vergleichbare Orakelergebnisse*
comparator:	*Java-Code | OCL-Code vergleicht Ist- mit Sollergebnis*
statechart:	*Statechart + Anfangszustand und Zielzustände werden geprüft*
assert:	*Objektdiagramme | OCL-Bedingungen | Java-Prüfcode prüfen das Istergebnis*
cleanup:	*Java-Code räumt benutzte Ressourcen auf*

}

SE SOFTWARE ENGINEERING

RWTH AACHEN

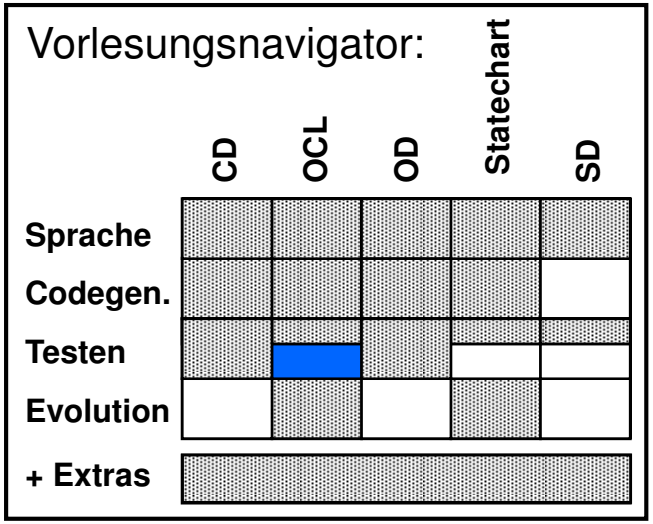
Modellbasierte Softwareentwicklung



- 8. Testen
- 8.3. OCL-Invarianten und Methodenspezifikationen

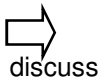
Prof. Dr. Bernhard Rumpe
Lehrstuhl für Software Engineering
RWTH Aachen

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



OCL-Invarianten dienen als Codeinstrumentierung

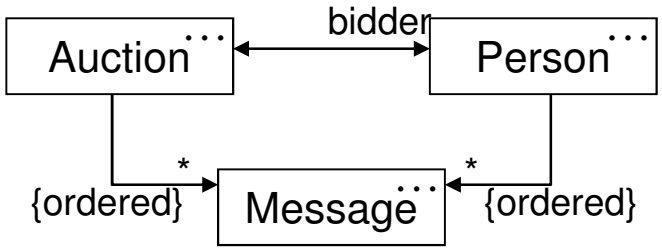
- Beispiel: Java-Methode, erweitert um Invarianten:



Java/P

```
class Auction {
  addMessage(Message m) {


    message.add(m);


    for(Iterator(Person) ip = bidder.iterator(); ip.hasNext(); ) {
      Person p = ip.next();
      p.receiveMessage(m);
    }

  }
}
```

CD



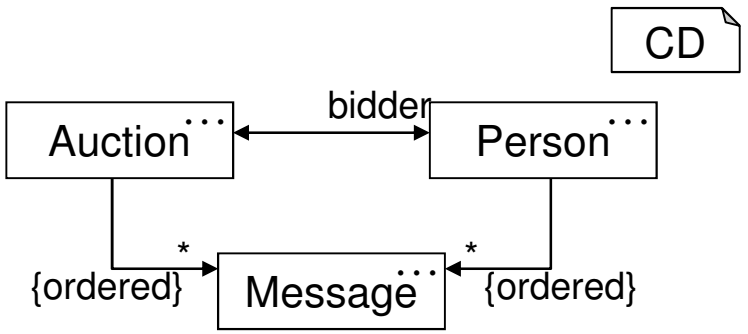
OCL-Invarianten dienen als Codeinstrumentierung

- Beispiel: Java-Methode, erweitert um Invarianten:

- Java/P

```
class Auction {
  addMessage(Message m) {
    ocl !this.message.contains(m);

    let int oldMessageSize = message.size;
    message.add(m);
    ocl message.size == oldMessageSize +1;

    for(Iterator(Person) ip = bidder.iterator(); ip.hasNext(); ) {
      Person p = ip.next();
      p.receiveMessage(m);
    }
    ocl forall p in bidder: m in p.message;
  }
}
```

CD



Codeinstrumentierung durch Invarianten

- ocl-Keyword ist ähnlich dem assert aus Java, erlaubt aber ocl-Bedingungen

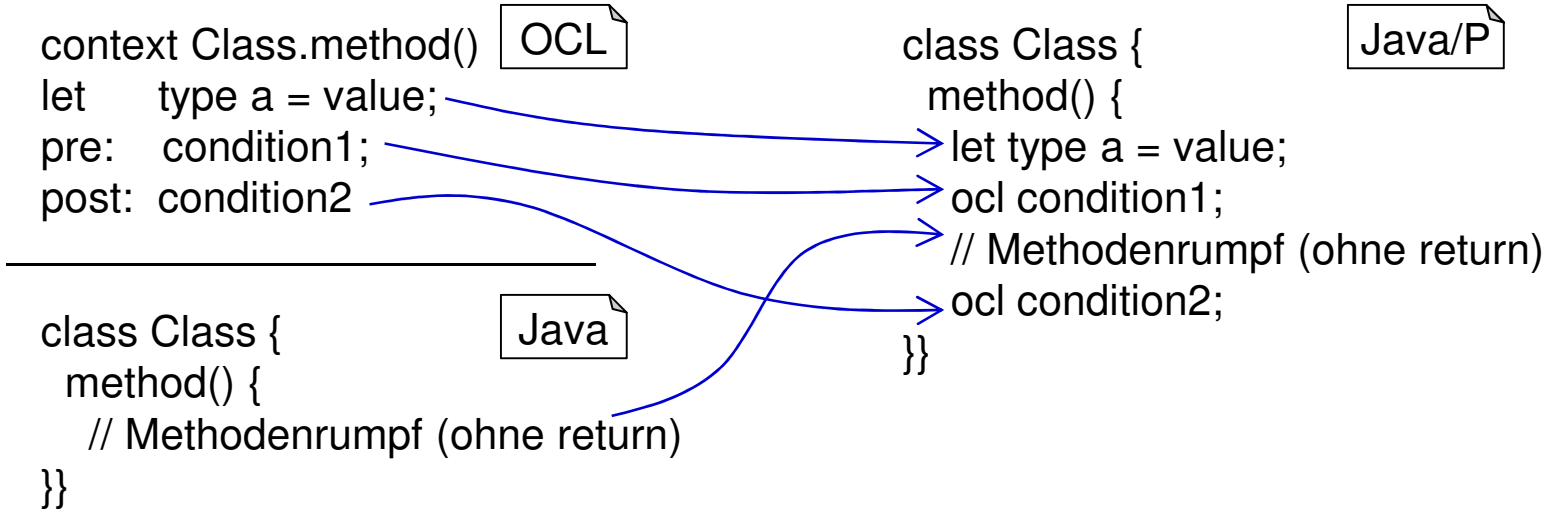
- Umsetzung durch Codegenerierung in
  - asserts (im normalen Code),
  - JUnit-Anweisungen (bei Tests) oder
  - Weglassen im Produktionscode

- Codeinstrumentierung ist besonders effektiv in Kombination mit vielen guten Tests!
  - Dadurch werden Invarianten intensiv getestet.

- Invarianten geben auch Hinweise, wo Tests durchgeführt werden sollten, z.B. für Grenzwerte bei Bedingungen

Methodenspezifikationen (Vor-/Nachbedingungen)

- Methodenspezifikationen können wie Invarianten genutzt werden.
- Codeinstrumentierung:

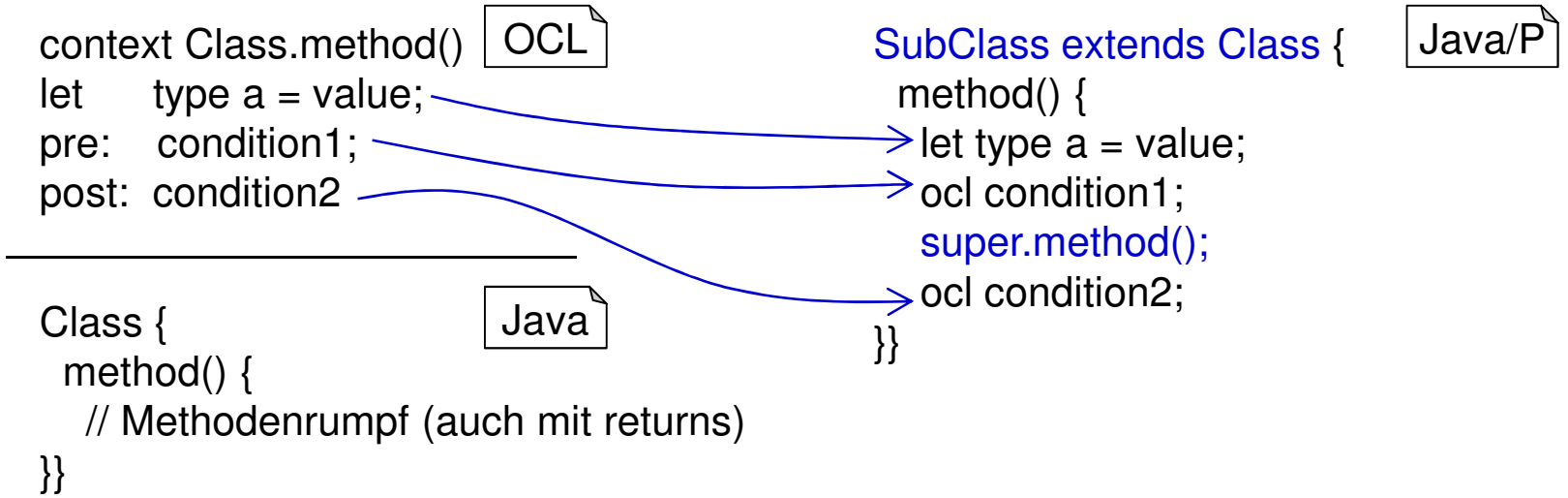


discuss

- Probleme:
  - Codeinstrumentierung evtl. nicht möglich, weil Quelle nicht verfügbar
  - Returns im Methodenrumpf sind gesondert zu behandeln

Methodenspezifikationen (Vor-/Nachbedingungen)

- Problembehebung:
  - Subklassenbildung benötigt keine Instrumentierung



- Zu beachten:
  - Überall Objekte der Subklasse verwenden,
    - Nutzung einer austauschbarer Factory (Entwurfsmuster)
  - keine statischen Methoden einsetzen.

Testfallbestimmung durch Methodenspezifikationen -1

- Grundidee:
  - Analyse einer Methodenspezifikation hilft bei der Entdeckung von verschiedenen zu testenden Fällen
- Beispiel:
  - context Person.changeCompany(String name)
    pre: company.name == name ||
    forall Company co: co.name != name

OCL

- Jede Klausel einer Disjunktion der Vorbedingung sollte als eigener Fall getestet werden:
  - company.name == name
  - forall Company co: co.name != name
- Weiterer Fall: Was passiert, wenn Vorbedingung nicht erfüllt ist:
  - company.name != name &&
    exists Company co: co.name == name

Testfallbestimmung durch Methodenspezifikationen -2

- Verfeinerung:
  - Heranziehen von Nachbedingungen
- Beispiel:

OCL

  - context int abs(int val)
    pre: true
    post: if (val>=0) then result==val else result==-val

- Jeder Fall der Nachbedingung sollte getestet sein.
  - Geeignete Testdaten sind zu suchen!
- Oft sind Randfälle und ihre Umgebung von Interesse:
- Klassische Randfälle: Leerer String, null, 0, leere Container.

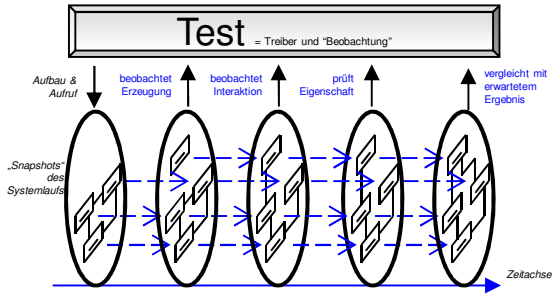
- In diesem Beispiel sind sinnvolle Testdaten: -n, -1, 0, 1, n (n groß)

- Mehr dazu in geeigneten Test-Veranstaltungen!

SE SOFTWARE ENGINEERING

RWTH AACHEN

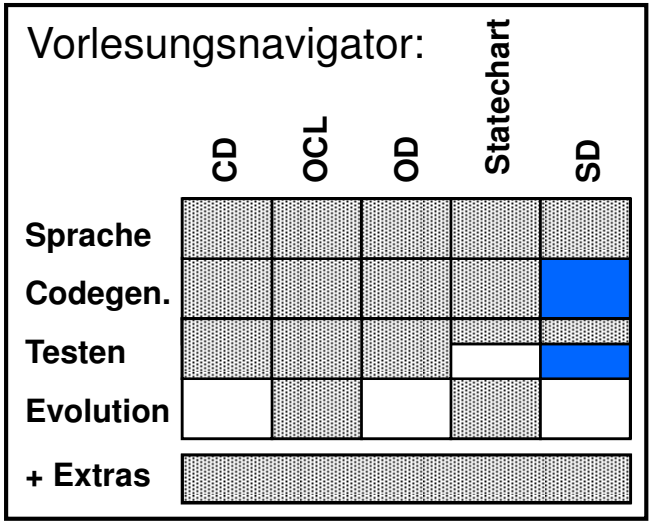
Modellbasierte Softwareentwicklung



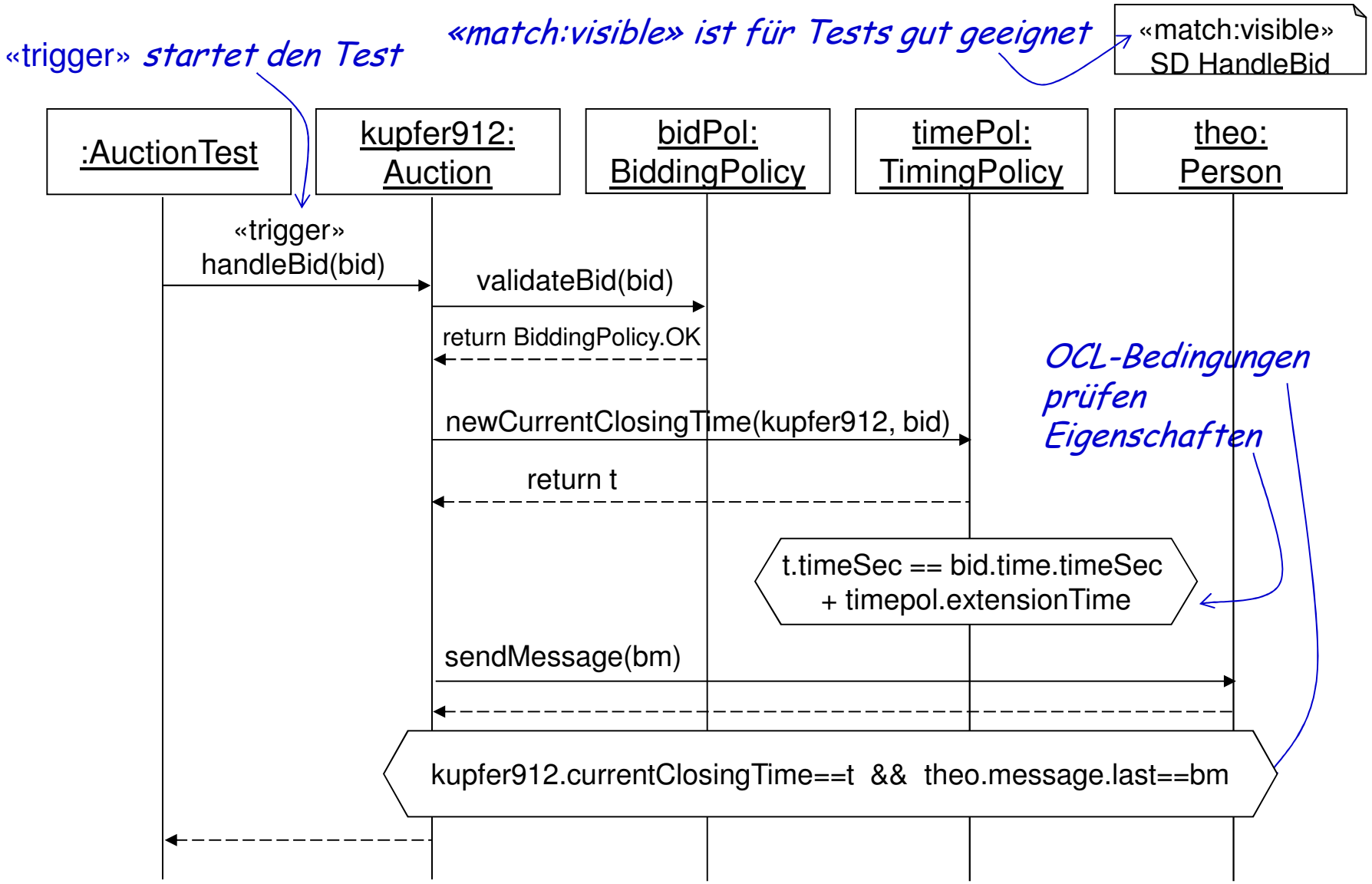
- 8. Testen
- 8.4. Sequenzdiagramme

Prof. Dr. Bernhard Rumpe
Lehrstuhl für Software Engineering
RWTH Aachen

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

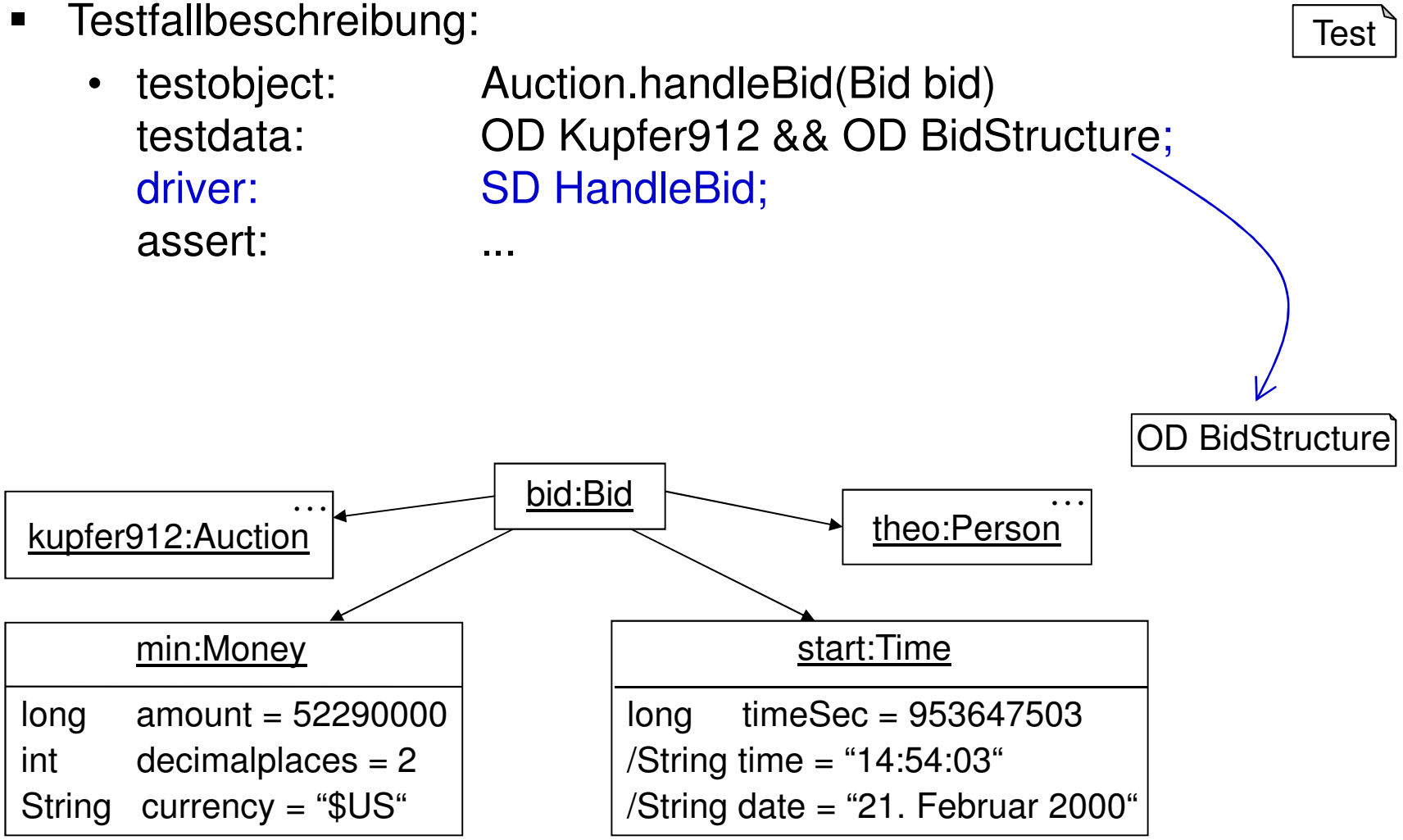


Sequenzdiagramm als Testfallbeschreibung

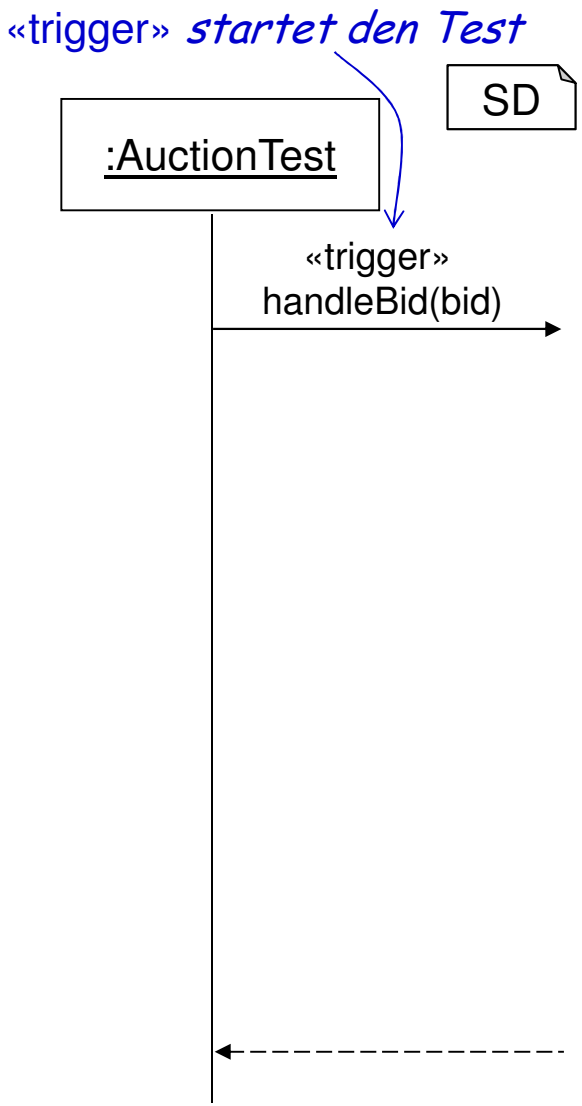


... zugehörige Testdaten

- Testfallbeschreibung:
  - testobject: Auction.handleBid(Bid bid)
    testdata: OD Kupfer912 && OD BidStructure;
    driver: SD HandleBid;
    assert: ...



Sequenzdiagramm als Testtreiber



- JUnit geeignet als Framework für Tests
- setUp beinhaltet Erzeugung der Objekte
- Trigger ist ein einfacher Aufruf

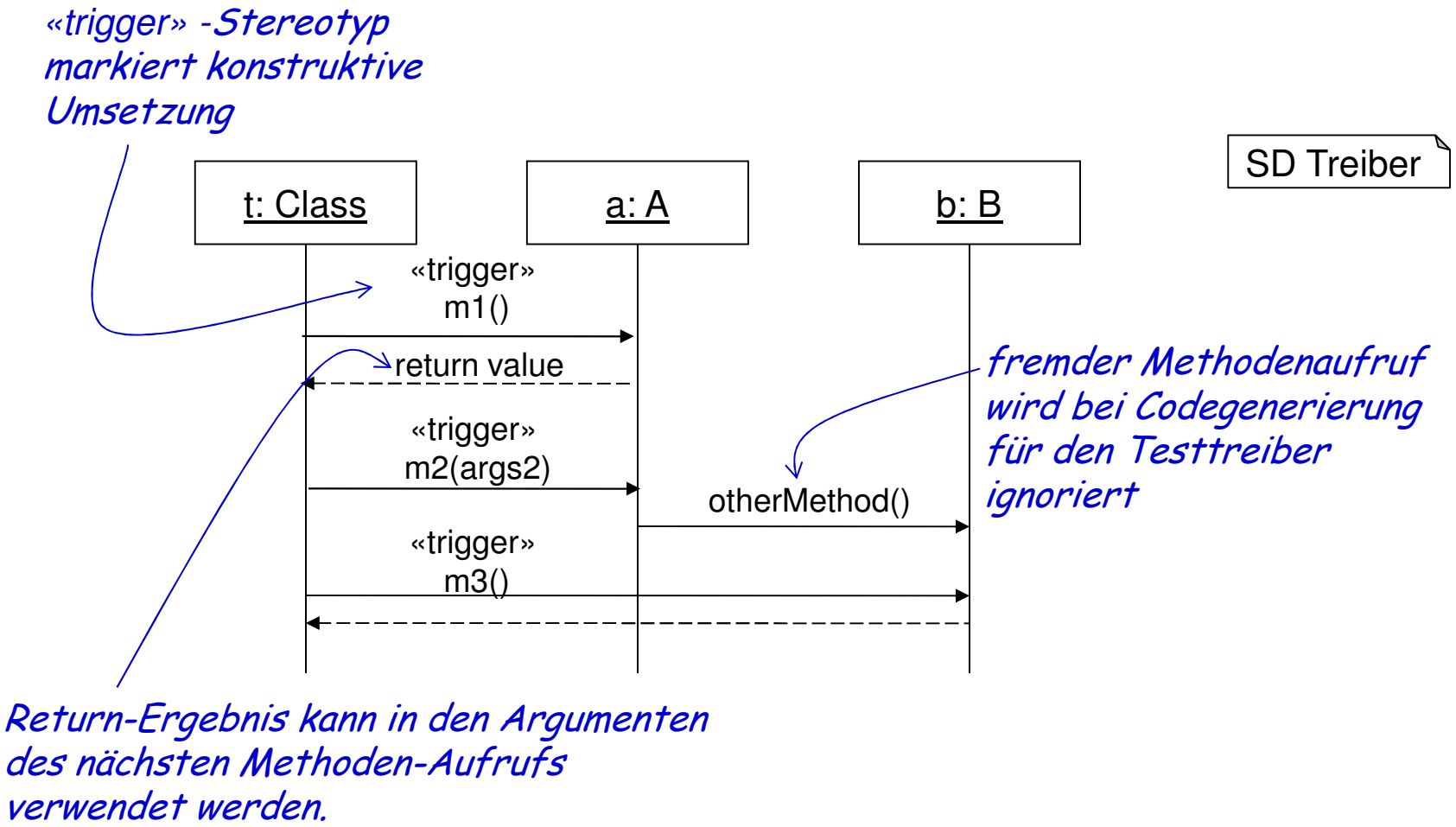
- mehr über JUnit unter: www.junit.org!

Java

```
import junit.framework.*;
public class AuctionTest extends TestCase {
    Auction kupfer912;
    Bid bid;
    public void testHandleBid() {
        setUp();
        kupfer912.handleBid(bid)
        // Asserts sind hier zu prüfen
        tearDown();
}}
```

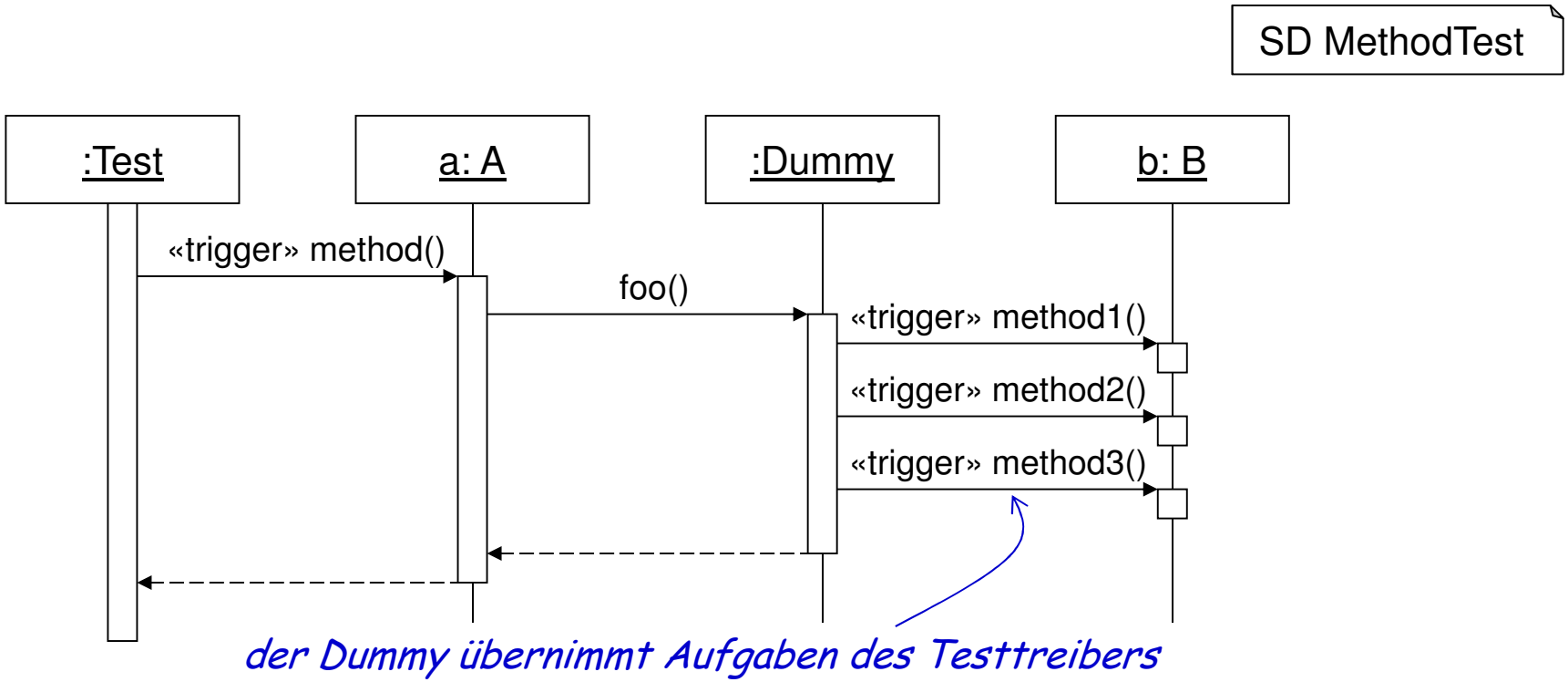
Komplexer Trigger -1

- Mehrere Trigger benötigen mehrere Methodenaufrufe



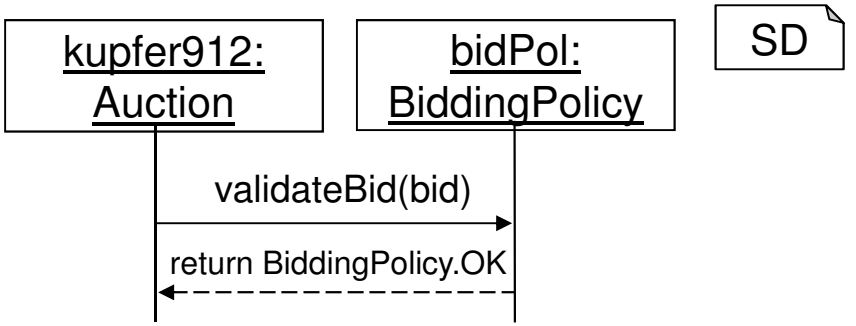
Komplexer Trigger -2

- Trigger kann über mehrere Objekte verteilt sein, wenn zwischendurch ein Dummy eingesetzt werden soll.
- Der Dummy kann aus dem SD generiert werden:



Sequenzdiagramm als Beobachtung

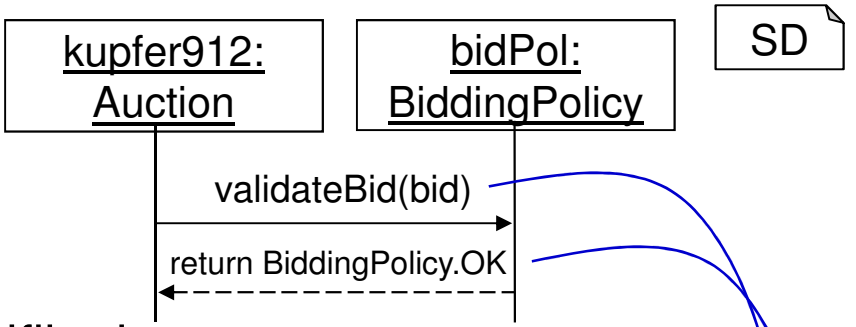
- Teil des Sequenzdiagramms ist zwischen den getesten Objekten zu beobachten:



- Technische Ansätze:
  - Reflection / Debugging API:
    Nicht standardisiertiert, nicht stabil, daher wenig nutzbar
  - Instrumentierung des Objectcodes
  - Instrumentierung des Quellcodes: wenn verfügbar
  - Instrumentierung durch Subklassen

Beobachtung von Aufrufen

- Teil des Sequenzdiagramms ist zwischen den getesten Objekten zu beobachten:



- Umsetzung wie bei Methodenspezifikation:
  - Instrumentation durch Subklasse
  - Verwendung eines Monitors für Reihenfolgen und Argumente:

Java

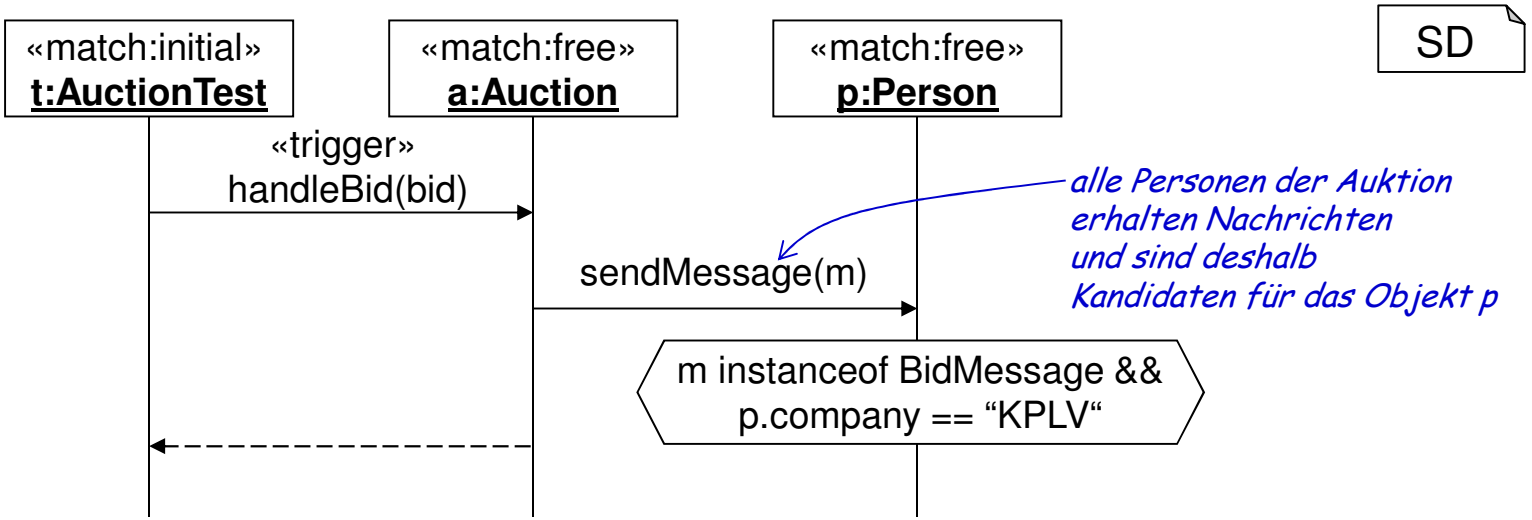
```
public class BiddingPolicyCheck extends BiddingPolicy {
    Monitor m;
    public BiddingPolicyCheck(Monitor m) { this.m = m; }

    public int validateBid(Bid bid) {
        m.callStarted(this, Monitor.ID_VALIDATE_BID,bid);
        int result = this.super(bid);
        m.callEnded(this, Monitor.ID_VALIDATE_BID,result);
        return result;
}}
```

Monitor zur Beobachtung

- Grundzüge des Monitors:

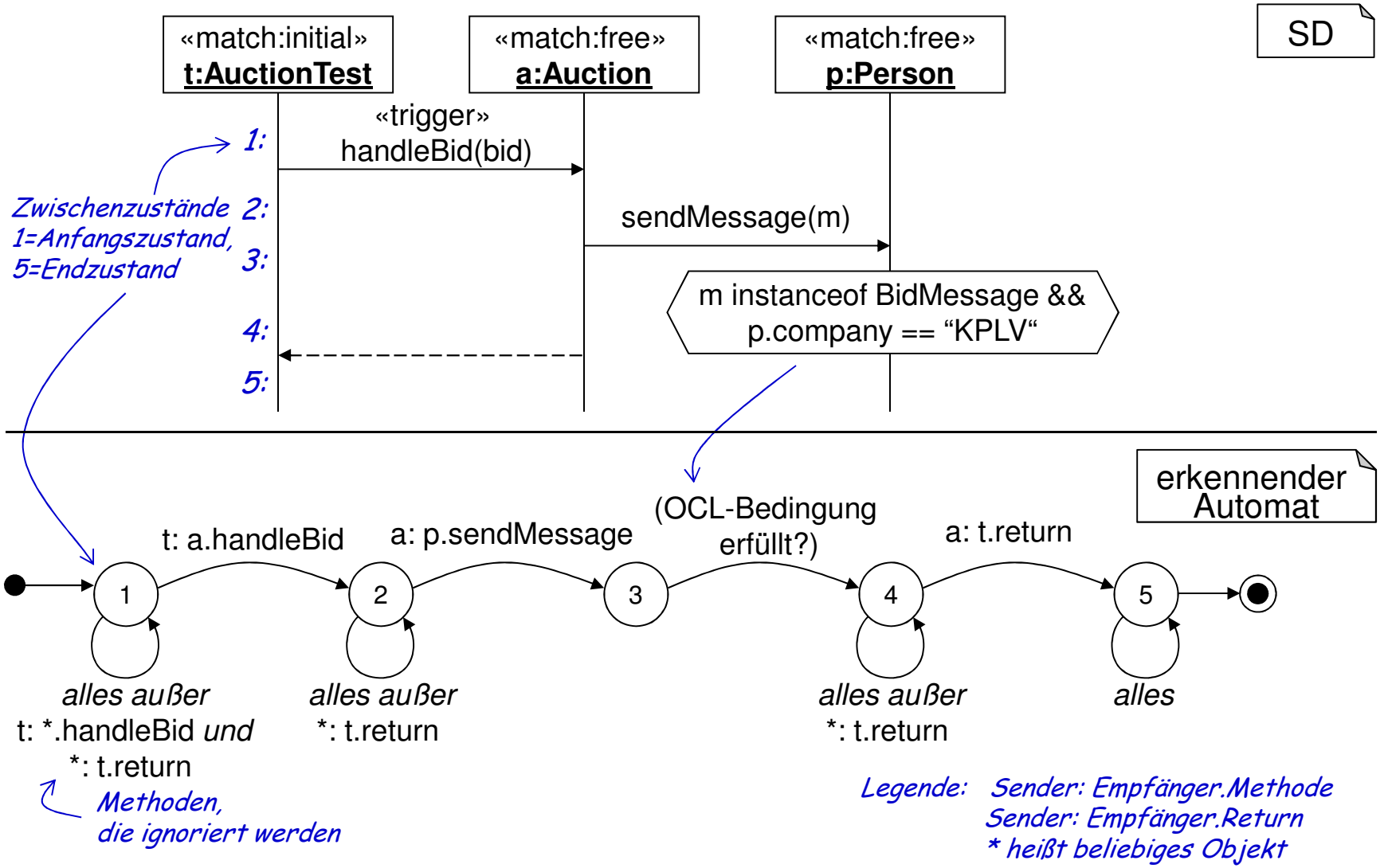
- Aufrufe und Returns werden beim Monitor angemeldet
  - Argumente sind: Aufrufer, Methoden-Identifikator, Argumente
    - callStarted(Object monitored, int methodID, Object arg1 ...)
- Geprüft werden
  - Reihenfolgen der Aufrufe / Returns
  - Korrektheit der Argumente
  - Erfüllung der Invarianten
- Stereotypen «match:*» beeinflussen erlaubte Beobachtungen:
  - «match:free» z.B. erlaubt, dass ein beobachtetes Objekt mit mehreren anderen in ähnlicher Weise kommuniziert
  - BTW: «match:initial» = Vollständige Beobachtung (complete) bis zur letzten angegeben Aktion jedes Stimulus-Typs, danach egal (free).

Reihenfolgeerkennung im Monitor

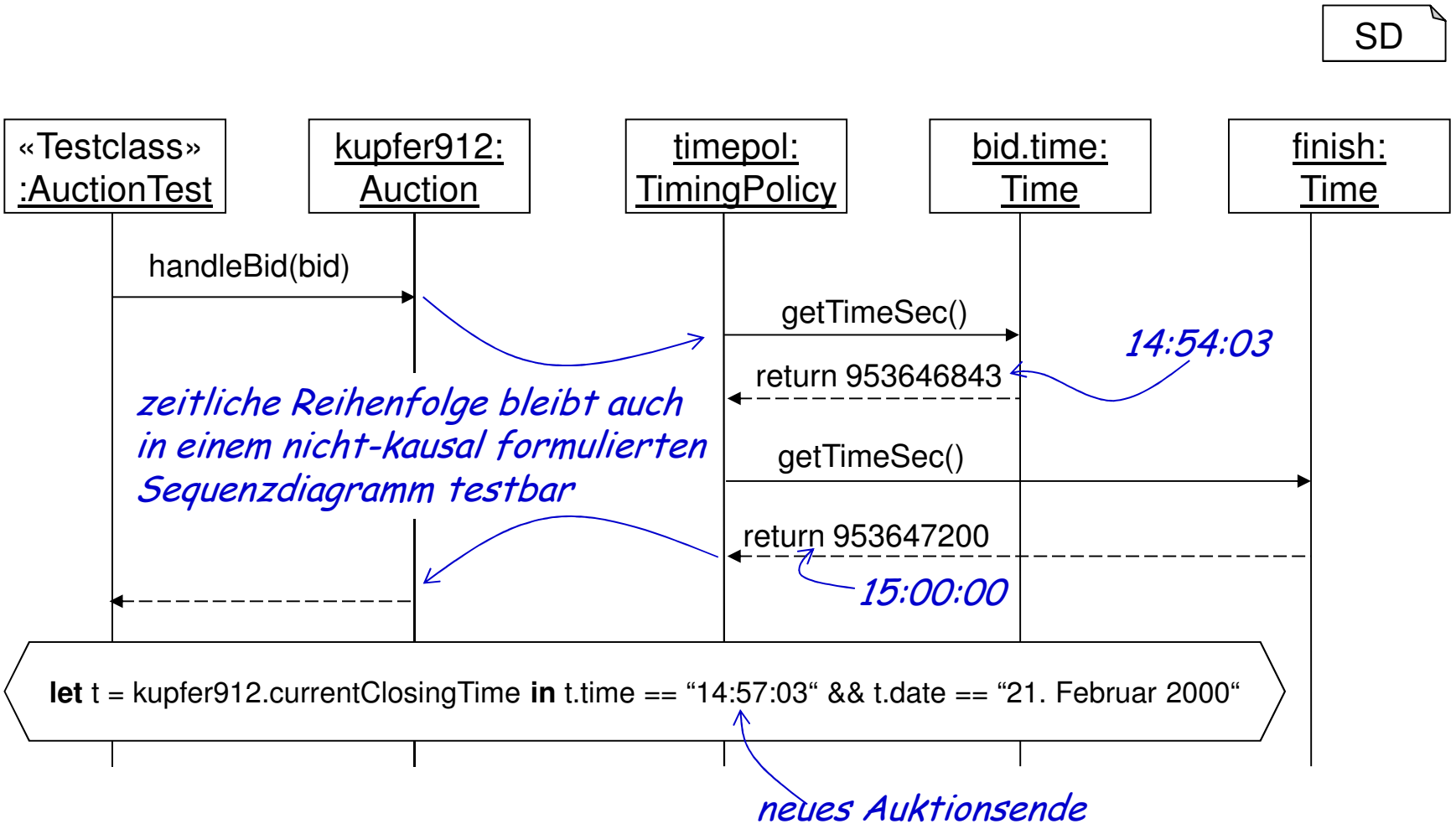


- Wegen «match:free» ist Objekt p nicht eindeutig
- OCL-Bedingung über p erfordert im Prinzip "Backtracking"
- Besser: Erkennung des Durchlaufs durch ein SD mit einem nicht-deterministischen Automaten
  - Effektiv durch übliche Transformation in deterministischen Automaten.

Erkennungsverfahren - animiert



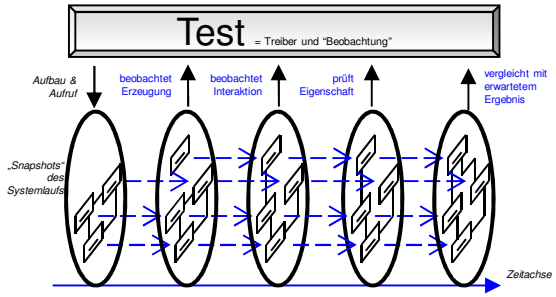
Nichtkausale Sequenzdiagramme



SE SOFTWARE ENGINEERING

RWTH AACHEN

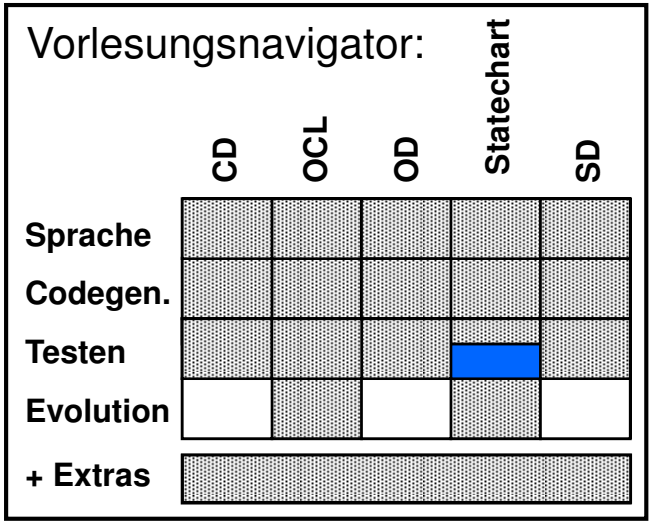
Modellbasierte Softwareentwicklung



- 8. Testen
- 8.5. Statecharts

Prof. Dr. Bernhard Rumpe
Lehrstuhl für Software Engineering
RWTH Aachen

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



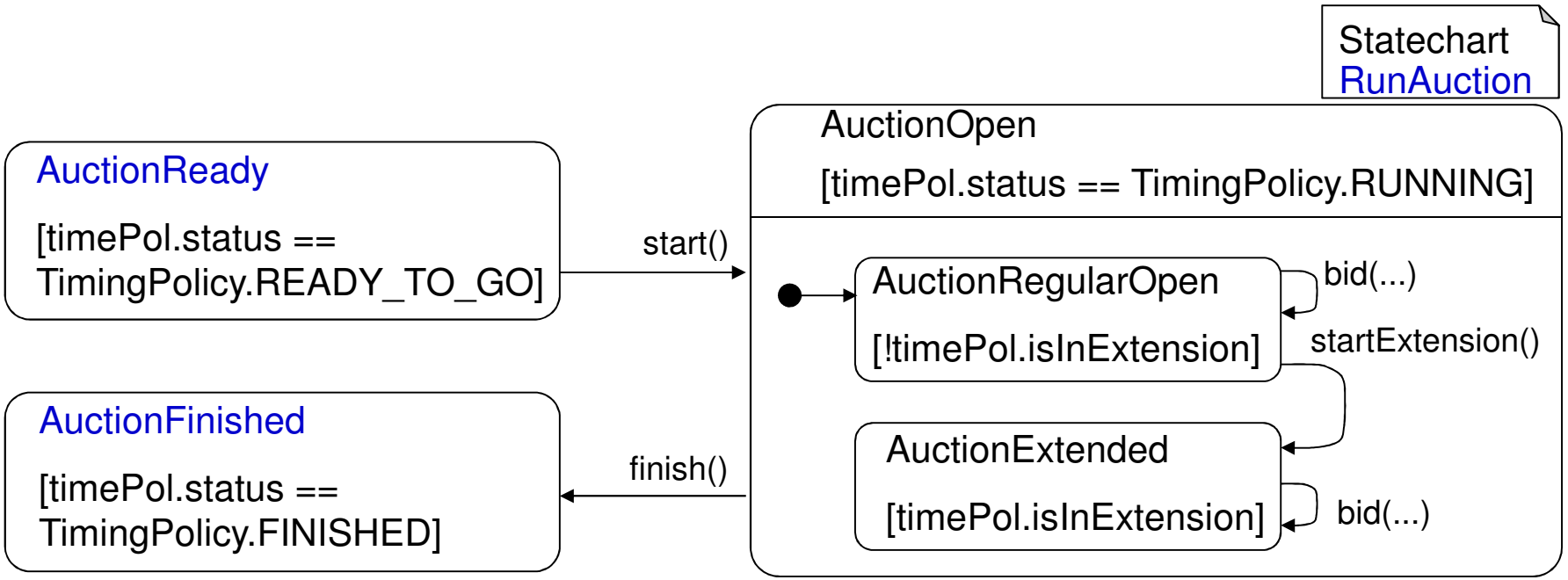
Einsatzgebiete für Statecharts

- Konstruktive Statecharts: zur Codegenerierung
  - typisch: Ausführbare Aktionen, hoher Detaillierungsgrad
  - (wurde bereits behandelt)

- Statecharts für Tests
  - typisch: Zustandsinvarianten, prädikative Nachbedingungen
  - (Prinzip: Umwandlung in OCL, Nutzung als Methoden-spezifikationen)

- Statecharts als Verhaltensbeschreibungen
  - typisch: wenig detailliert, unterspezifiziert (Transitionsauswahl)
  - Verwendung als Kontrolle der Zustandsübergänge im Testfall
  - oder als Generierungsvorlage für Testfälle

Statechart im Test als Ablaufprüfung



Test

```
driver:      auction.start();  auction.startExtension();  auction.finish();   ...
statechart:  auction    RunAuction from AuctionReady to { AuctionFinished }
```

beobachtetes Objekt — *Name des Statechart* — *Startzustand* — *Liste von Endzu-ständen* — *auction ist in einem (nicht abgebildeten) Objektdiagramm definiert*

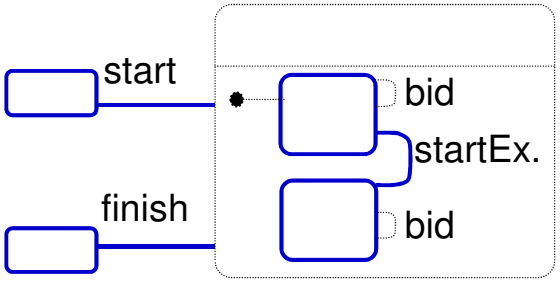
Testüberdeckung eines Statechart

- Zustandsüberdeckung:
  - Jeder Zustand wird von einem Test durchlaufen
- Transitionsüberdeckung:
  - Jede Transition wird von einem Test durchlaufen
- Pfadüberdeckung:
  - Jeder Pfad wird von einem Test durchlaufen
  - praktisch unmöglich, wenn eine Schleife enthalten ist:
- minimale Schleifenüberdeckung:
  - Schleifenlose Pfade + jede Scheife einmal durchlaufen

- Weitere Tests für jede der Alternativen in der Disjunktion in Vorbedingungen und Invarianten, ...
- Überdeckungen können mit einem Monitor gut gemessen werden. Aber:
  - Erzeugen von Testdaten für die Pfade ist schwierig
  - Pfadauswahl ist komplex (Minimale Menge von Pfaden?)
  - Manche Pfade sind nicht möglich, z.B. wegen Invarianten

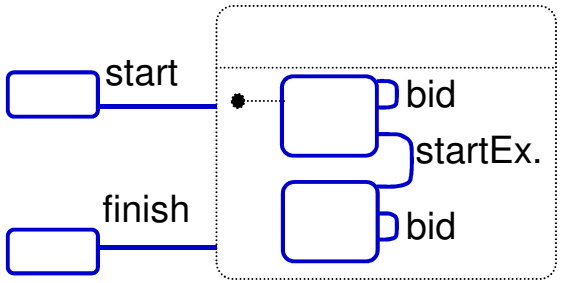
Beispiel: Testfälle für das Auction-Statechart

Zustandsüberdeckung benötigt einen Testfall
Eingabe: start; startExtension; finish

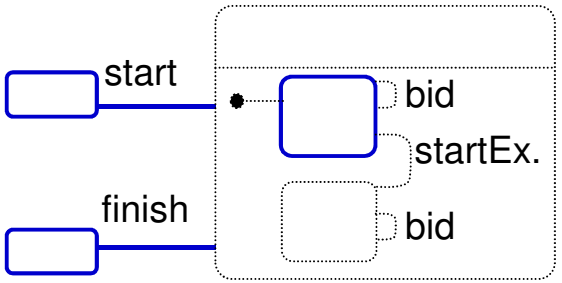


Transitionsüberdeckung und minimale Schleifenüberdeckung sind hier beide gleich.
Zwei Pfade reichen aus, sind aber auch notwendig, da finish aus beiden Subzuständen verlassen werden kann.

Eingabe: start; bid; startExtension; bid; finish



Eingabe: start; finish



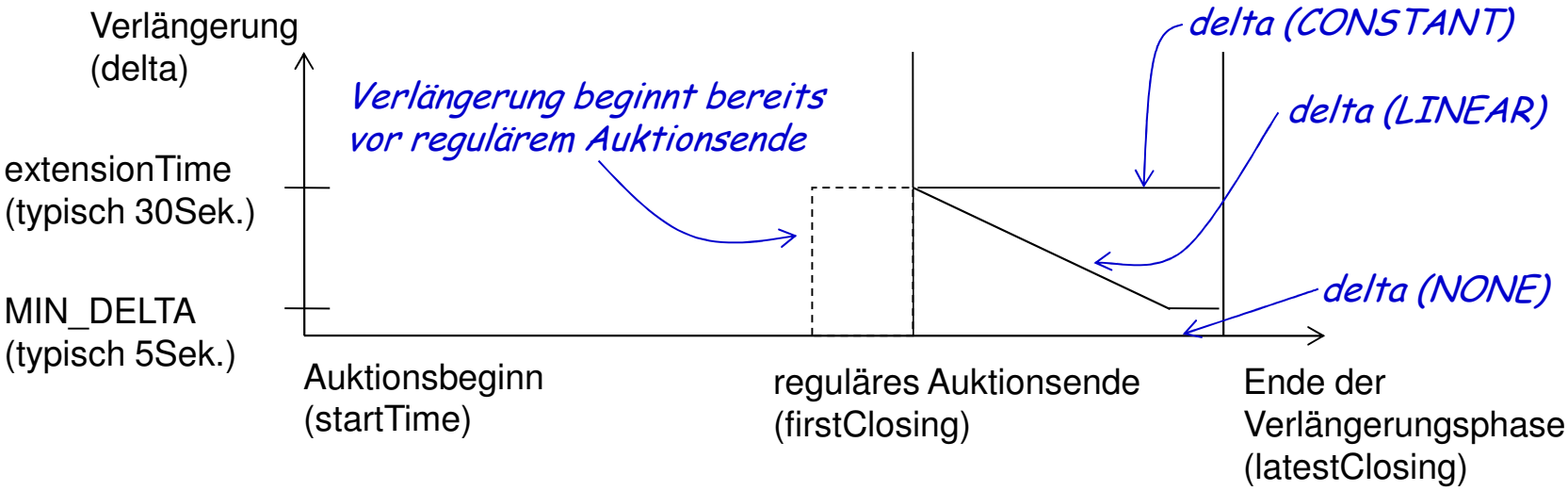
Pfadüberdeckung nicht möglich, wegen bid-Schleifen

start; bid;* startExtension; bid;* finish und start; bid;* finish

und deren Präfixe sind unterschiedliche Pfade.

Beispiel–Aufgabe: Policies zur Verlängerung einer Auktion

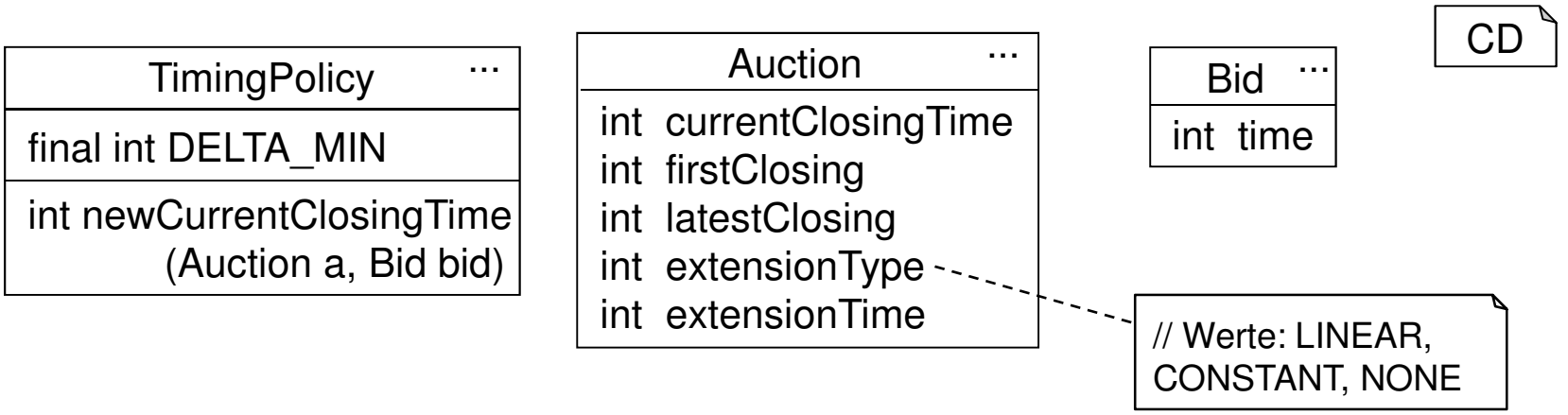
- Anforderungen: (1) Wird ein Gebot kurz vor Ende der Auktion abgegeben, so wird ggf. verlängert um bis zu *extensionTime* Sekunden. (2) Auktionen enden immer zwischen *firstClosing* und *latestClosing*.
- Drei Policies:
  - NONE: Es findet keine Verlängerung statt
  - CONSTANT: Die Verlängerung ist immer gleich.
  - LINEAR: Verlängerung linear abnehmend, mindestens MIN_DELTA.
- Der Graph veranschaulicht die gewährte Verlängerung delta:



Anhang

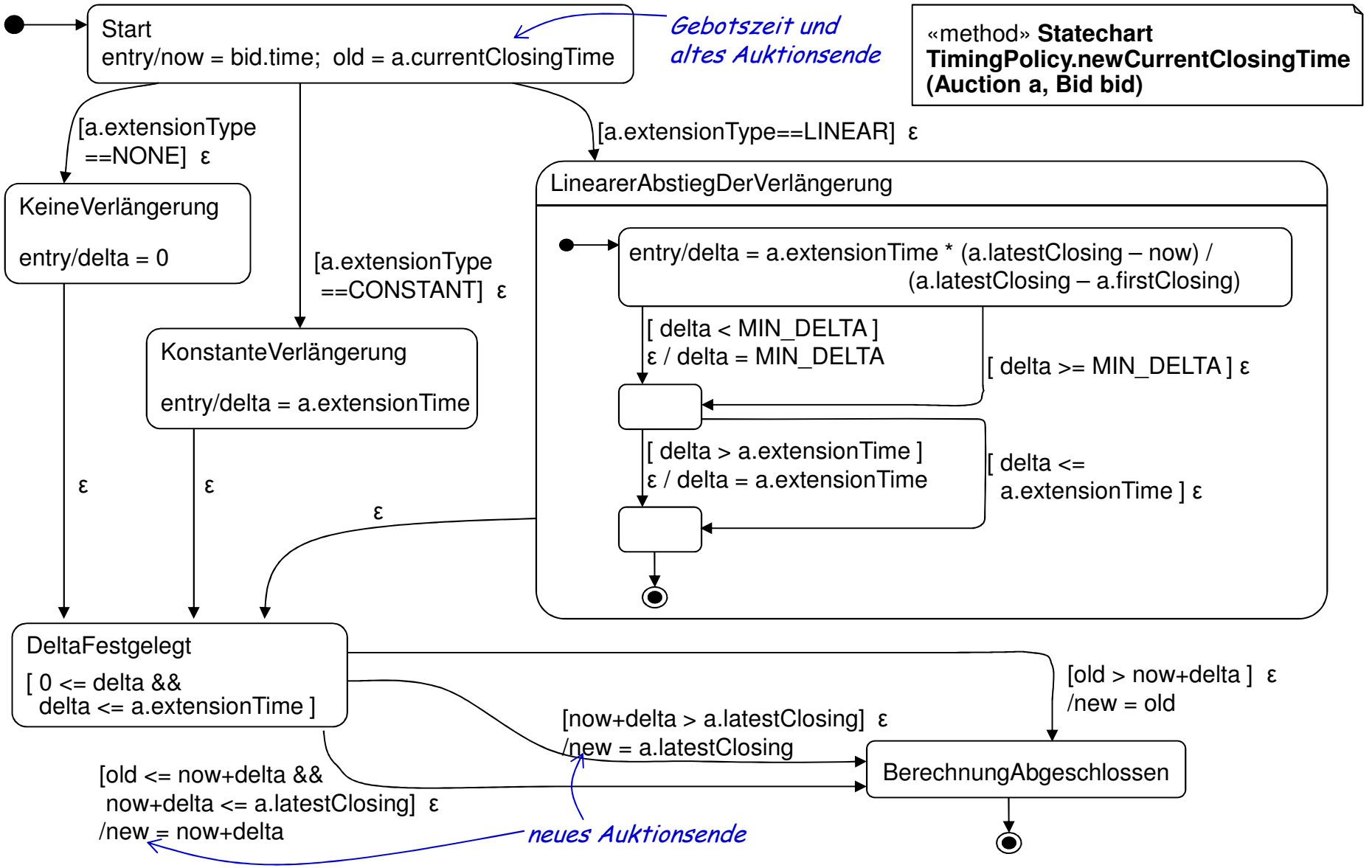
Aufgabe, Teil 1:

- 1) Implementieren Sie eine geeignete Methode newCurrentClosingTime, die in folgender Struktur die neue ClosingTime berechnet:



- 2) Überlegen Sie sich ein Statechart für die Methode das die Fälle und Varianten über den Kontrollfluß darstellt.
- 3) Identifizieren Sie je einen Satz von Pfaden, der eine Zustands-, Transitions- bzw. Pfadüberdeckung erreicht.
- 4) Entwickeln Sie für jeden Pfad einen Satz Testdaten.
- 5) Testen Sie Ihre Implementierung mit jedem Datensatz.

Ein Lösungsansatz/Orakel für newCurrentClosingTime

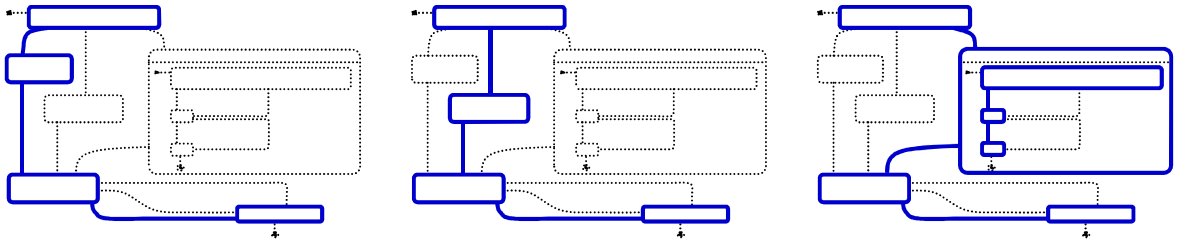


Aufgabe, Teil 2:

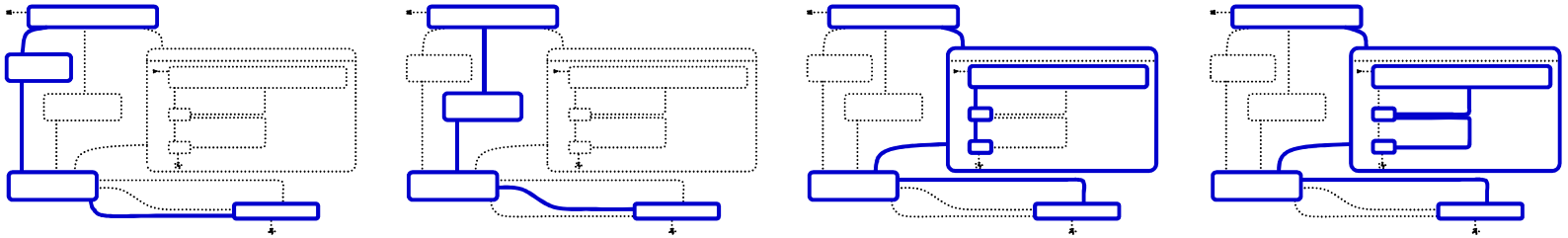
- 5) Identifizieren Sie für dieses Statechart je einen Satz von Pfaden, der eine Zustands-, Transitions- bzw. Pfadüberdeckung erreicht.
- 6) Entwickeln Sie für jeden Pfad einen Satz Testdaten.
- 7) Implementieren Sie das Statechart als Orakel
- 8) Testen Sie Ihre Implementierung mit jedem der Datensätze und vergleichen Sie das Ist-Ergebnismit dem Ergebnis des Orakels.

Lösungsansatz für Überdeckungen -1

Zustandsüberdeckung ist mit drei Pfaden möglich:

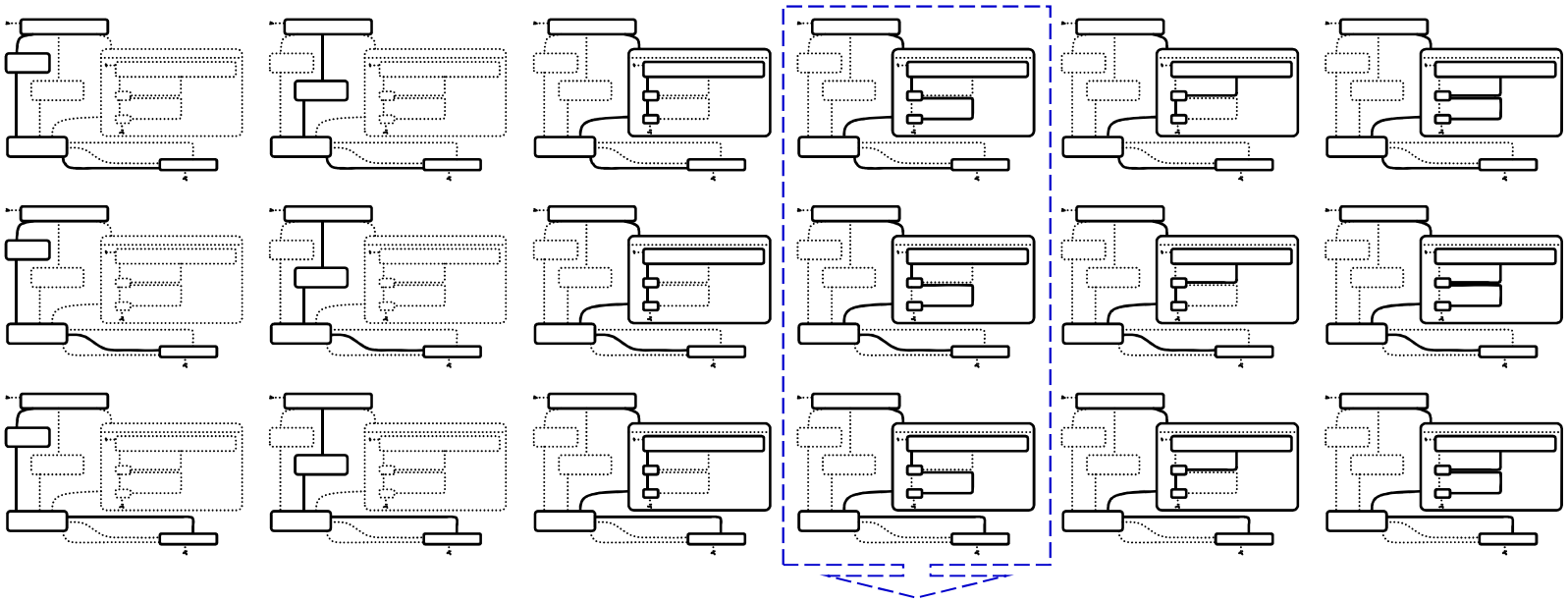


Transitionsüberdeckung ist damit noch nicht erreicht.
Es reichen aber vier Pfade:



Lösungsansatz für Überdeckungen -2

Pfadüberdeckung (identisch mit der minimalen Schleifenüberdeckung, da keine Schleife vorhanden ist; 18 Pfade):

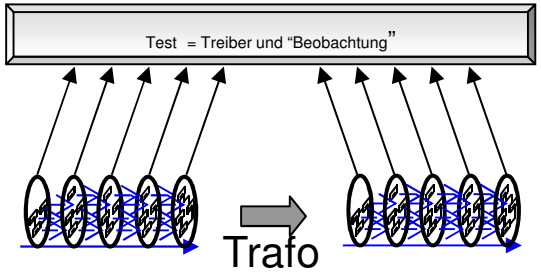


aufgrund von Invarianten im Algorithmus sind diese Pfade zwar erkennbar, aber nicht ausführbar und daher auch nicht für Tests zugänglich

SE SOFTWARE ENGINEERING

RWTH AACHEN

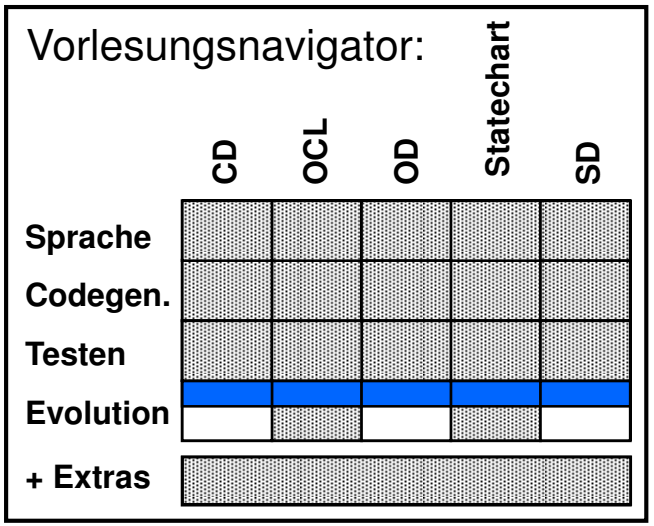
Modellbasierte Softwareentwicklung



- 9. Evolution durch Transformation
- 9.1. Grundlagen des Refactoring

Prof. Dr. Bernhard Rumpe
Lehrstuhl für Software Engineering
RWTH Aachen

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



Evolution

- Software muss permanent weiterentwickelt werden:
  - Neue Anforderungen
  - Geänderte Technologie
  - Neue Vernetzung mit Nachbarsystemen
  - Behebung von Fehlern

- Techniken für die Evolution von Legacy Systemen, z.B.:
  - Reverse Engineering: Gewinnung der ursprünglichen Entwurfsmodelle aus dem Quellcode (Objectcode)
  - Wrapping: Verpacken von Code einer alten Technologie (Cobol, Mainframe) in eine moderne Zugangsschicht (Java, Web)

- Evolution besteht meist aus einem oder wenigen großen Schritten (Transformationen) mit viel Fehlerrisiko

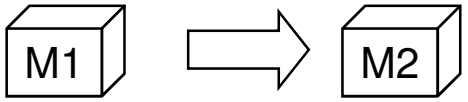
Evolution von Modellen

- Ziel:
  - Risikominimierung
  - Steigerung der Effektivität
- durch:
  - kleine Schritte durch systematische, überschaubare Transformationen
  - Nutzung von Architektur und Design in Form von Modellen.

- Voraussetzung für qualitätsgesicherte Modellevolution:
  - Codegeneratoren
  - Automatisierte Tests
  - Sammlung von Modelltransformationen

Modelltransformation

- Eine Modelltransformation ist ausführbare Abbildung eines gegebenen Modells in ein anderes.

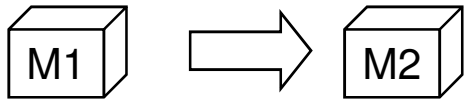


- Beispiele (für nichtevolutionäre Transformationen):
  - Abbildung von Klassendiagrammen nach Java
  - Abbildung von Klassendiagrammen nach SQL-Statements
  - Extraktion von Analysedaten

- Evolutionäre Transformationen:
  - Hinzufügen von get/set-Methoden
  - Verschieben eines Attributs
  - Zusammenlegen zweier Klassen
  - Minimalisierung von Statecharts

Modelltransformation

- Eine Modelltransformation ist eine zielgerichtete von einem Programm ausführbare Abbildung eines gegebenen Modells in ein anderes.



- Eigenschaften von Modelltransformationen:
  - bidirektional?
  - abstrahierend (vergessend)?
  - detaillierend (Details hinzufügend)?
  - semantikerhaltend / verfeinernd / abstrahierend?
  - innerhalb oder zwischen Sprachen?

- Innerhalb einer Sprache sind Transformationen verwendbar für:
  - Verfeinerung / Abstraktion
  - Evolution

Refactoring

- Spezialfall von Transformationen:
  - Fowler‘99 nutzt Refactoring auf Code-Ebene (Java)
  - Eingeführt wurde Refactoring von Opdyke/Johnson’ 92/93 für C++

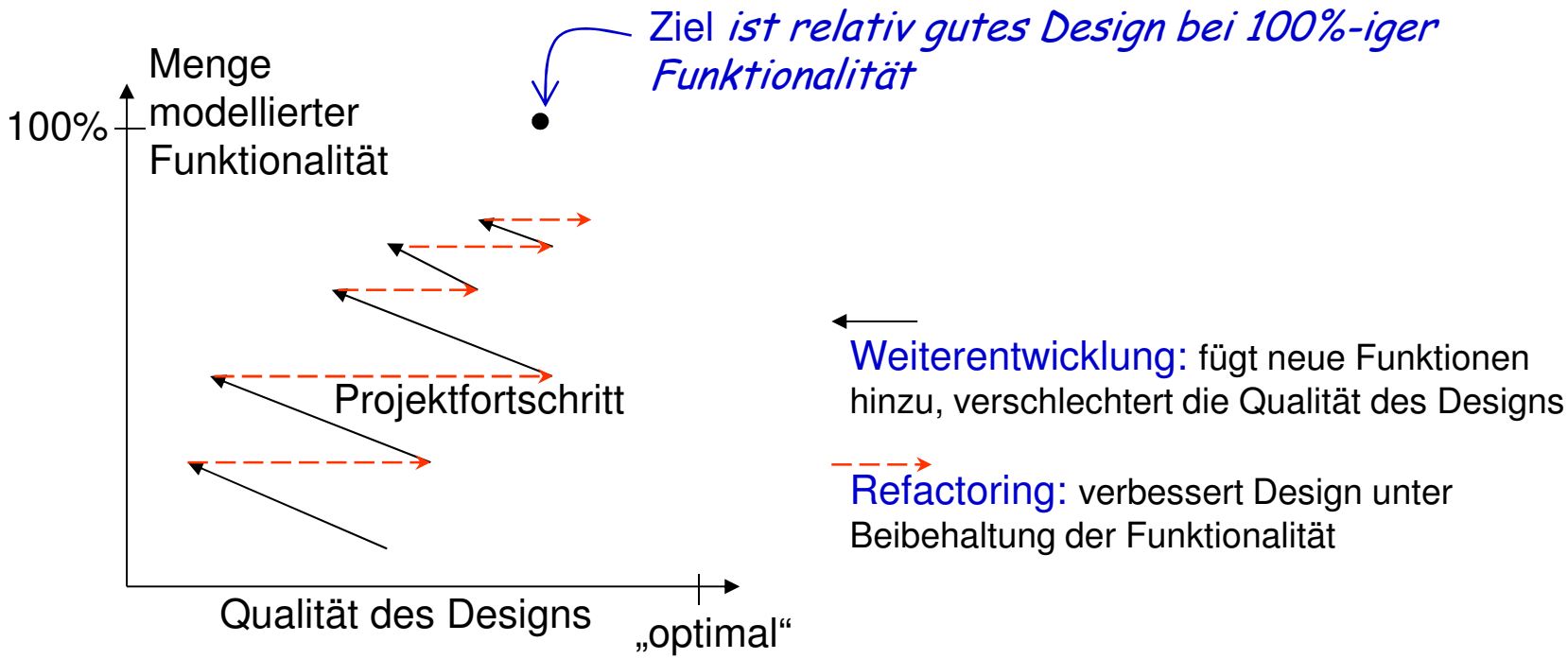
- Definition Refactoring [dt. Übersetzung von Fowler’99]:

  - Refaktorisierung:
    - Eine Änderung an der internen Struktur einer Software, um sie leichter verständlich zu machen und einfacher zu verändern, ohne ihr beobachtetes Verhalten zu ändern.

- Feststellung:
  - Refactoring von Modellen dient zur Evolution von Systemen.

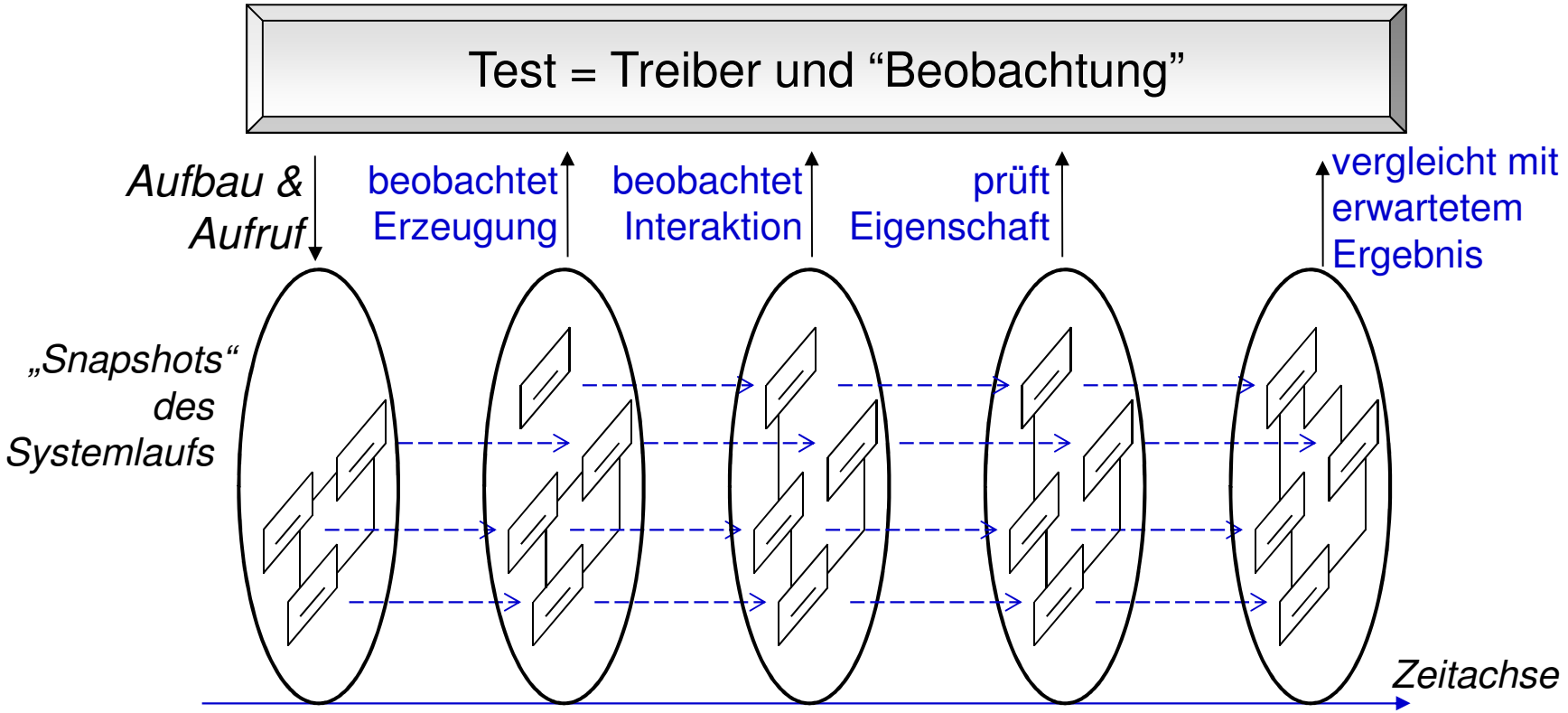
Methodik des Refactoring

- Scharfe Trennung der Aktivitäten: Refactoring und Erweiterung der Funktionalität
  - Refactoring dient nur der Design-Verbesserung
- Aber: zeitlich enge Verzahnung durch
  - „Model a little, refactor a little“ (frei nach XP)



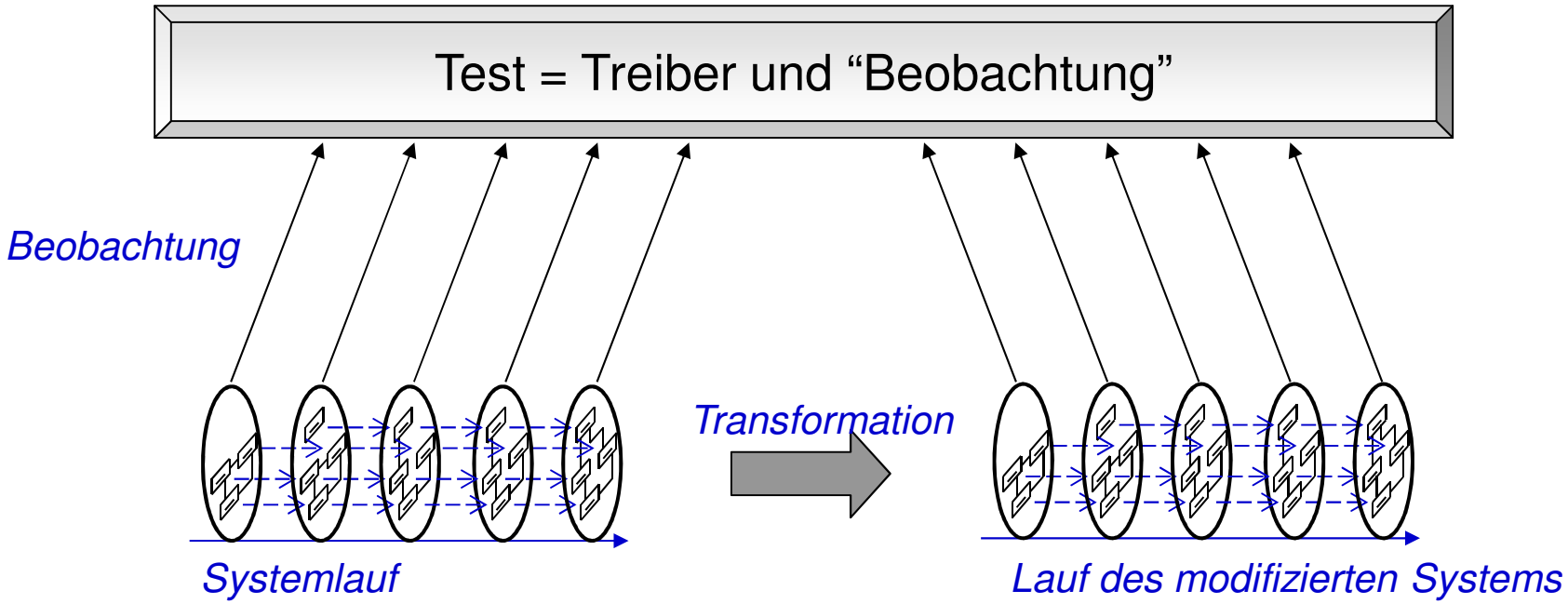
Tests sind Beobachtungen für Transformationen

- Tests beobachten Struktur und Verhalten:

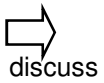


Validierung von Transformationen

- Die Testbeobachtung bleibt unter der Transformation erhalten



Refactoring von UML-Notationen



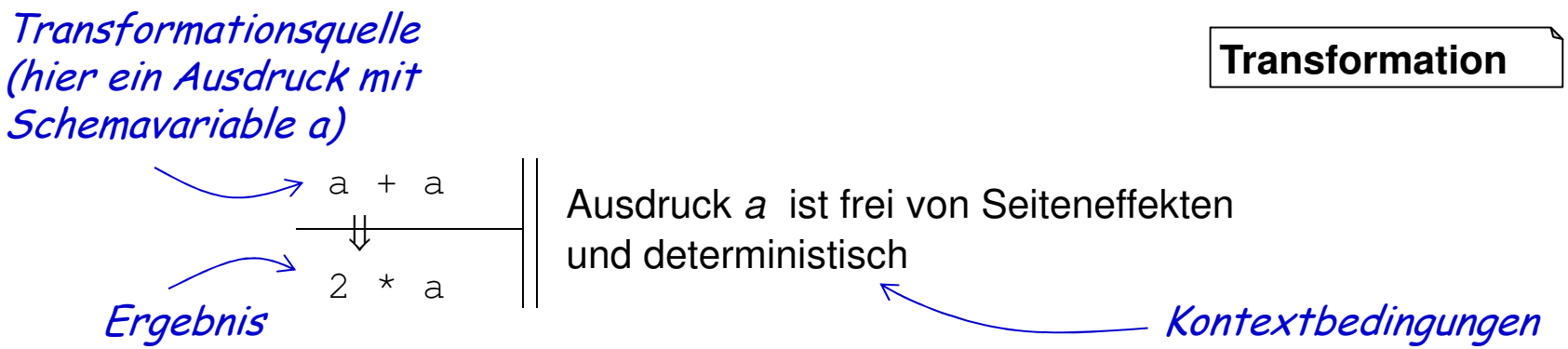
- Klassendiagramme
- Code
- Objektdiagramme
- OCL
- Statecharts
- Sequenzdiagramme

Refactoring von UML-Notationen

- Klassendiagramme
  - Architektur/Design-Verbesserung: sehr lohnend
- Code
  - siehe Refactoring-Literatur
- Objektdiagramme
  - notwendig im Kontext von CD-Transformationen,
    aber: unerforscht
- OCL
  - Logik besitzt ausgereifte Kalküle, Rechenregeln für Container, ...
- Statecharts
  - Transformationsregeln zur Vereinfachung von Statecharts
- Sequenzdiagramme
  - bisher noch keine Transformationstechniken dafür entwickelt.

Beispiel einer Transformation:

- für Java:



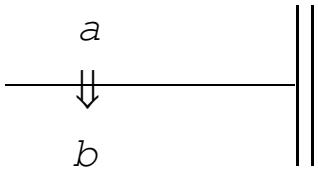
- Sowohl für Java als auch OCL steht die gesamte Algebra zur Verfügung, die dort als Gleichungen formuliert ist:
  - a-a == 0, a+b == b+a, x && true == x
- Datentypspezifische Transformationen, Beispiel für OCL:
  - List{a,b,c}.first == a
- Java hat meist Kontextbedingungen der Form:
  Ausdruck ist definiert | deterministisch | seiteneffektfrei.

Kontextbedingungen

- Neben allgemeingültigen Aussagen gibt es Transformationen, die nur unter Bedingungen gelten:

- Beispiel: Ersetzung von Gleichem:

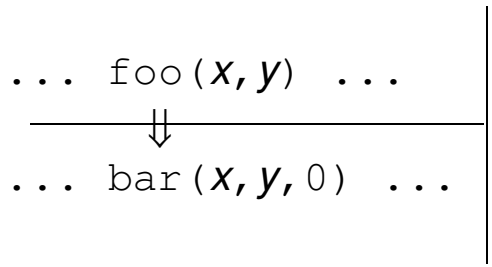
Transformation



1. Ausdrücke *a* und *b* sind frei von Seiteneffekten
2. *a* == *b*

- Beispiel: Ersetzung eines Methodenaufrufs:

Transformation

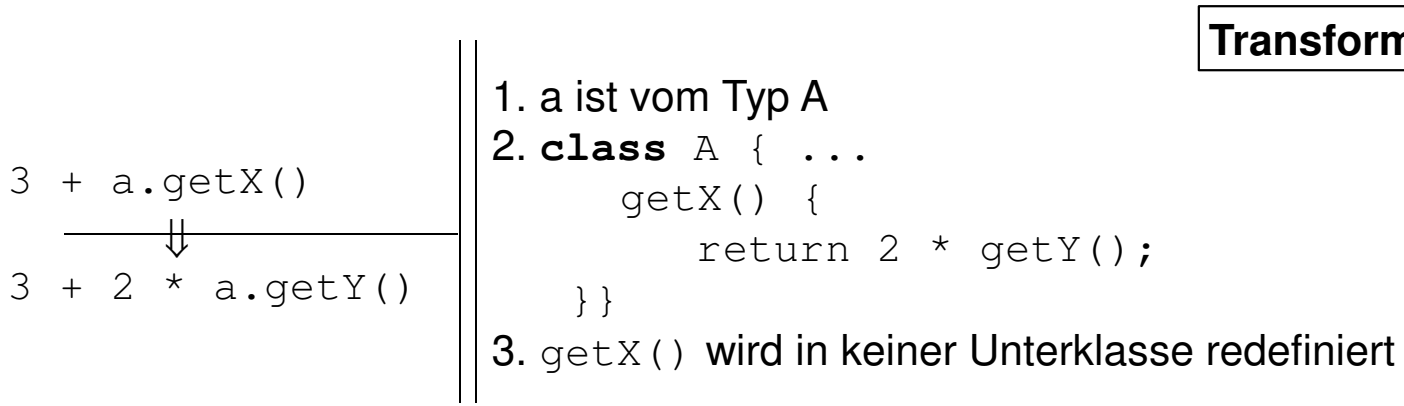


1. Ausdruck *x* hat Typ *A*, *y* hat Typ *B*
2. **inv:**

```
forall A a, B b:
  bar(a,b,0) == foo(a,b)
```

Kontextbedingungen

- Expansion einer Methode:
  - analog dem Compiler-Prinzip des Methoden-Inlining



- Meist sind auch noch allgemeinere, aber komplexere Kontextbedingungen möglich.
  - z.B. Redefinition ist hier möglich, aber nur in Grenzen
  - Prüfbarkeit der Kontextbedingungen?

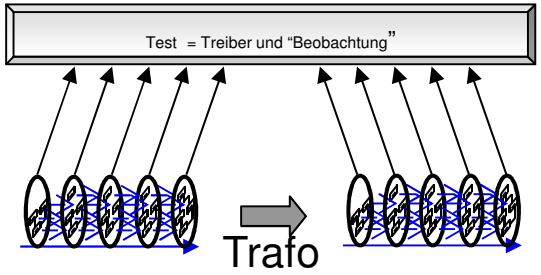
Prüfbarkeit von Kontextbedingungen

- Behandlung von Kontextbedingungen:
  - Automatische Prüfbarkeit an der Syntax:
    - Beispiele: Typisierung, Initialisierung von Variablen, ...
  - Semi-automatische Prüfbarkeit:
    - Beispiele: Model-Checking für Systemeigenschaften
  - Interaktive Verifikation:
    - Beispiele: Korrektheitsbeweise von Aussagen in First-Order-Logik
  - Test:
    - Beispiel: Überprüfung der Einhaltung von Invarianten zur Laufzeit
  - Manuelle Reviews:
    - Reviewer gibt sein „OK“.

SE SOFTWARE ENGINEERING

RWTH AACHEN

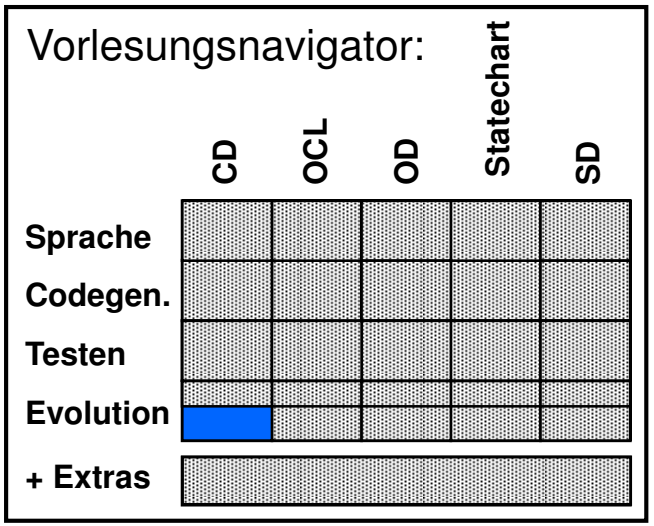
Modellbasierte Softwareentwicklung



- 9. Evolution durch Transformation
- 9.2. Refactoring von Klassendiagrammen

Prof. Dr. Bernhard Rumpe
Lehrstuhl für Software Engineering
RWTH Aachen

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



Quellen für Refactoring-Regeln

- Opdyke'93: 26 Grundregeln für C++:
  - Oft Löschen und Erzeugen von Programmelementen

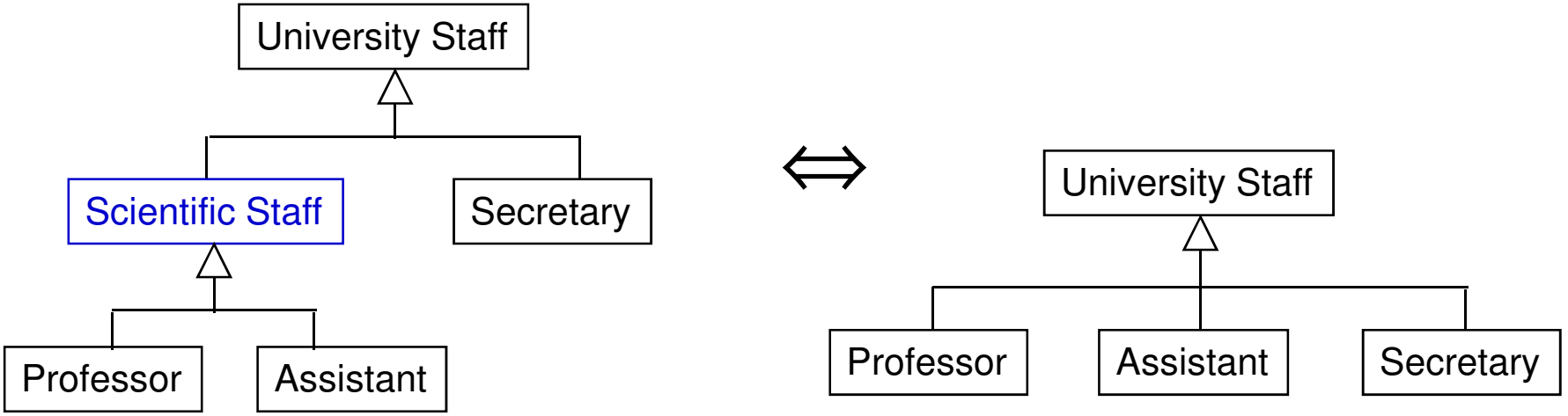
- Fowler'99: 72 Regeln für Java:
  - viele erklärt anhand von Klassendiagrammen
  - vier „Big Refactorings", Rest bearbeitet ein Vielzahl von Java-Elementen

Auszug der Liste der Refactorings (Fowler,99)

- Add Parameter
- Collapse Hierarchy
- Encapsulate Collection
- Extract Interface
- Extract Method
- Move Field (=Attribute)
- Move Method
- Pull Up Field
- Remove Middle Man
- Remove Parameter
- Rename Method
- Replace Array with Object
- Replace Conditional with Polymorphism
- Replace Delegation with Inheritance
- Replace Inheritance with Delegation
- Replace Error Code with Exception

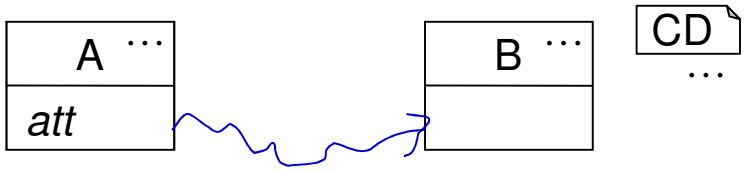
Refactoring “Collapse Hierarchy”

- Entfernen von Klassen in der Klassenhierarchie
- Die Regel kann in beide Richtungen angewendet werden
- Bei Entfernung: Vererbter Code ist in Subklassen zu verschieben
  - Sonderfall: Subklassen redefinieren Methode und rufen „super()“ auf
  - Sonderfall: Konstruktoren



Beispiel: Verschieben eines Attributs

- Attribut „att“ soll von Klasse A nach B verschoben werden



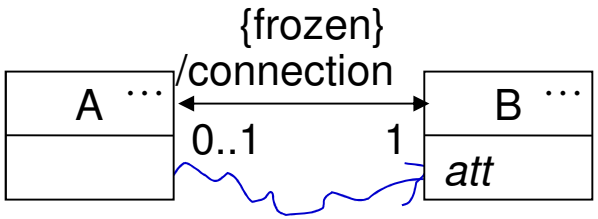
CD …

Code

```
// Code
a.att
```

a.exp ist der Navigationspfad von A nach B

⇓



CD …

OCL

```
context A a inv:
 a.connection == a.exp;
```

dies kann z.B. durch Tests geprüft werden

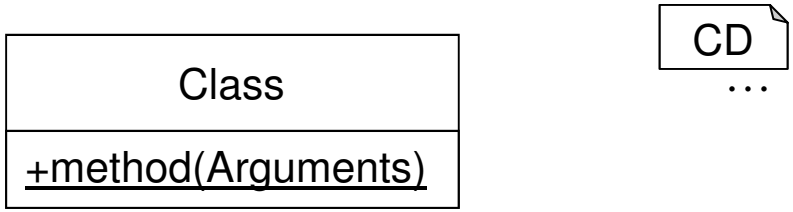
⇓

CD …

Code

```
// Code
a.exp.att
```

Refactoring: Einführung eines Testmusters

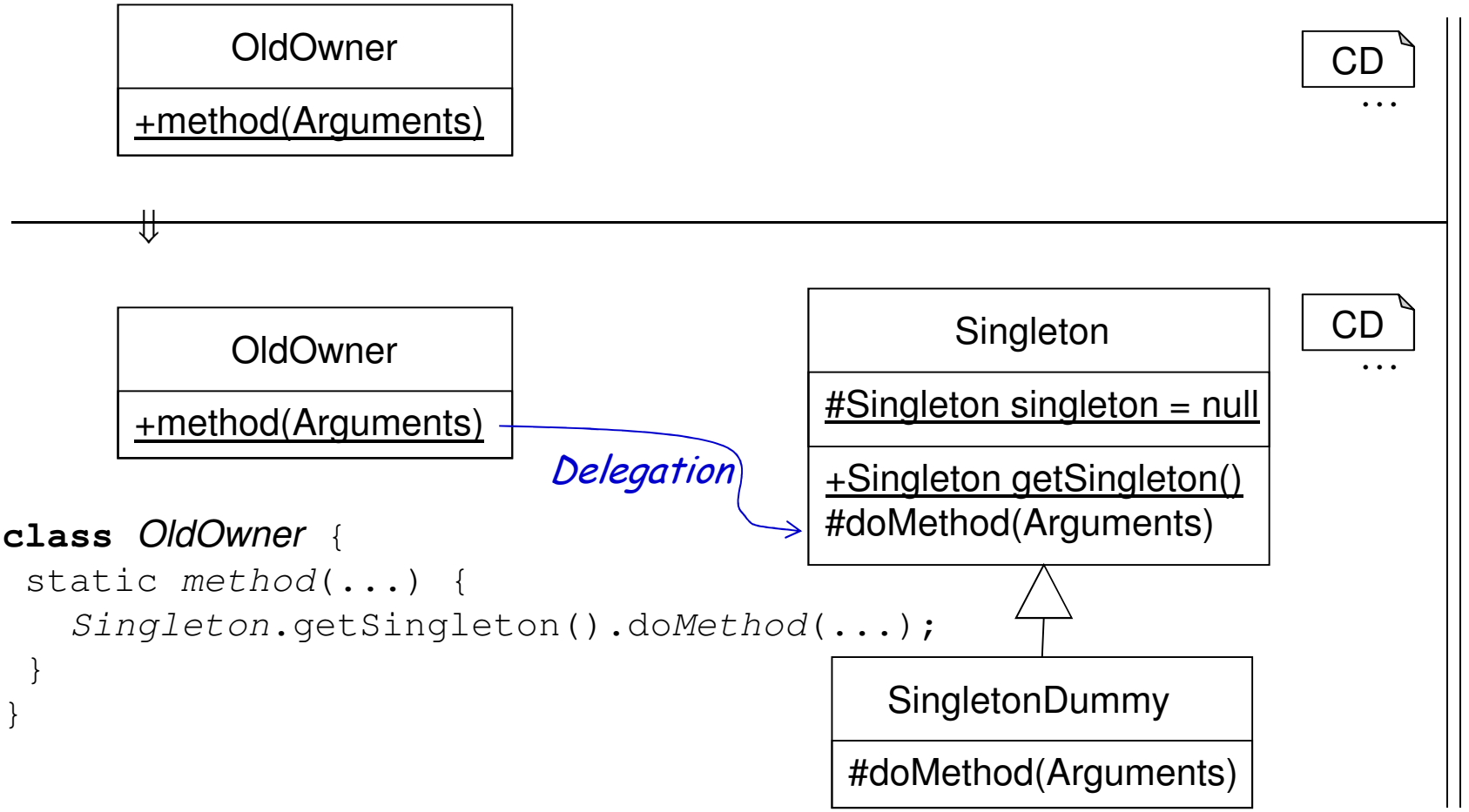


- Problem:
  - Klasse hat eine statische Methode
  - Methode hat Seiteneffekte
  - Struktur nicht für Tests geeignet!

- Lösung mit Transformation der Struktur in zwei Refactoring-Schritten
  - Delegation an Singleton-Objekt
  - Kapselung in Singleton

Ersetze statische Methode durch Singleton

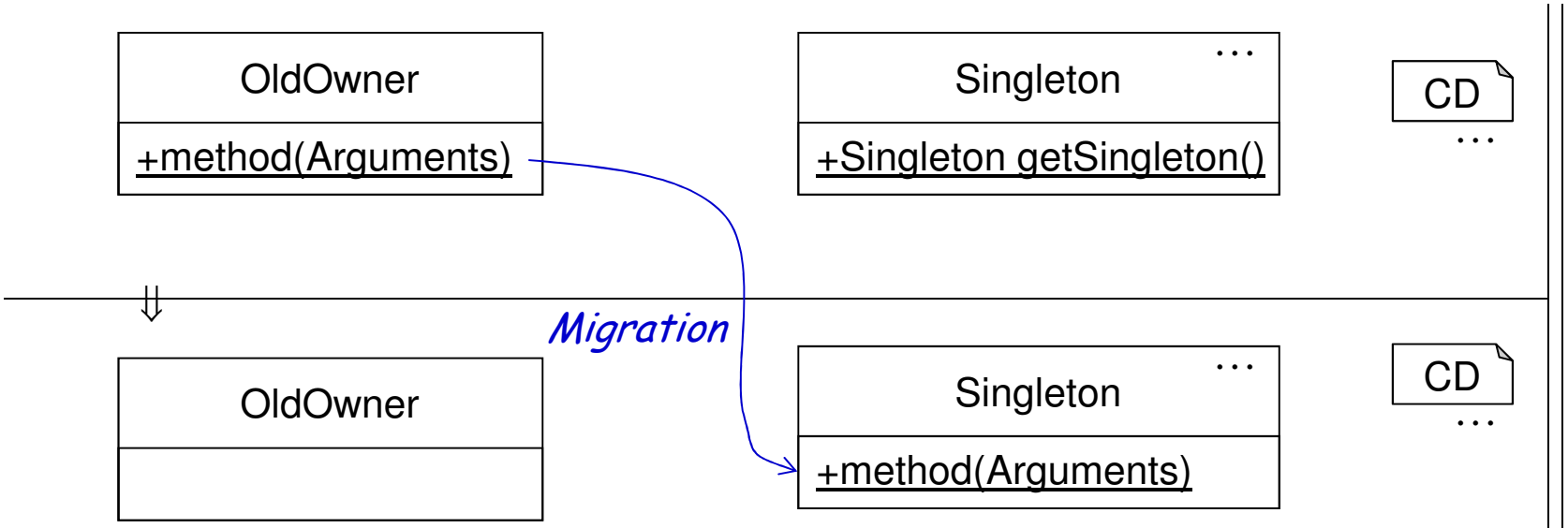
- Methode delegiert ihre Aufgabe an ein Singleton-Objekt
- SingletonDummy überschreibt die fragliche Methode




```
class OldOwner {
 static method(...) {
   Singleton.getSingleton().doMethod(...);
 }
}
```

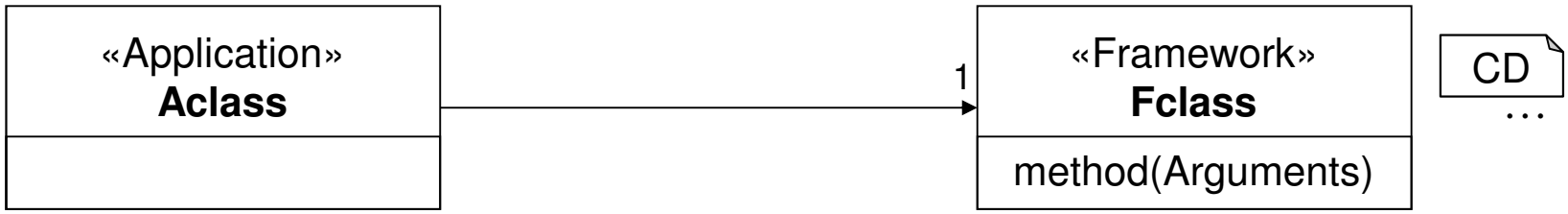
... und migriere statische Methode

- Kapselung des Aufrufmechanismus im Singleton




```
class Singleton {
  static method(...)
    if(singleton==null) // initialize Object;
    singleton.doMethod(...)
}}
```

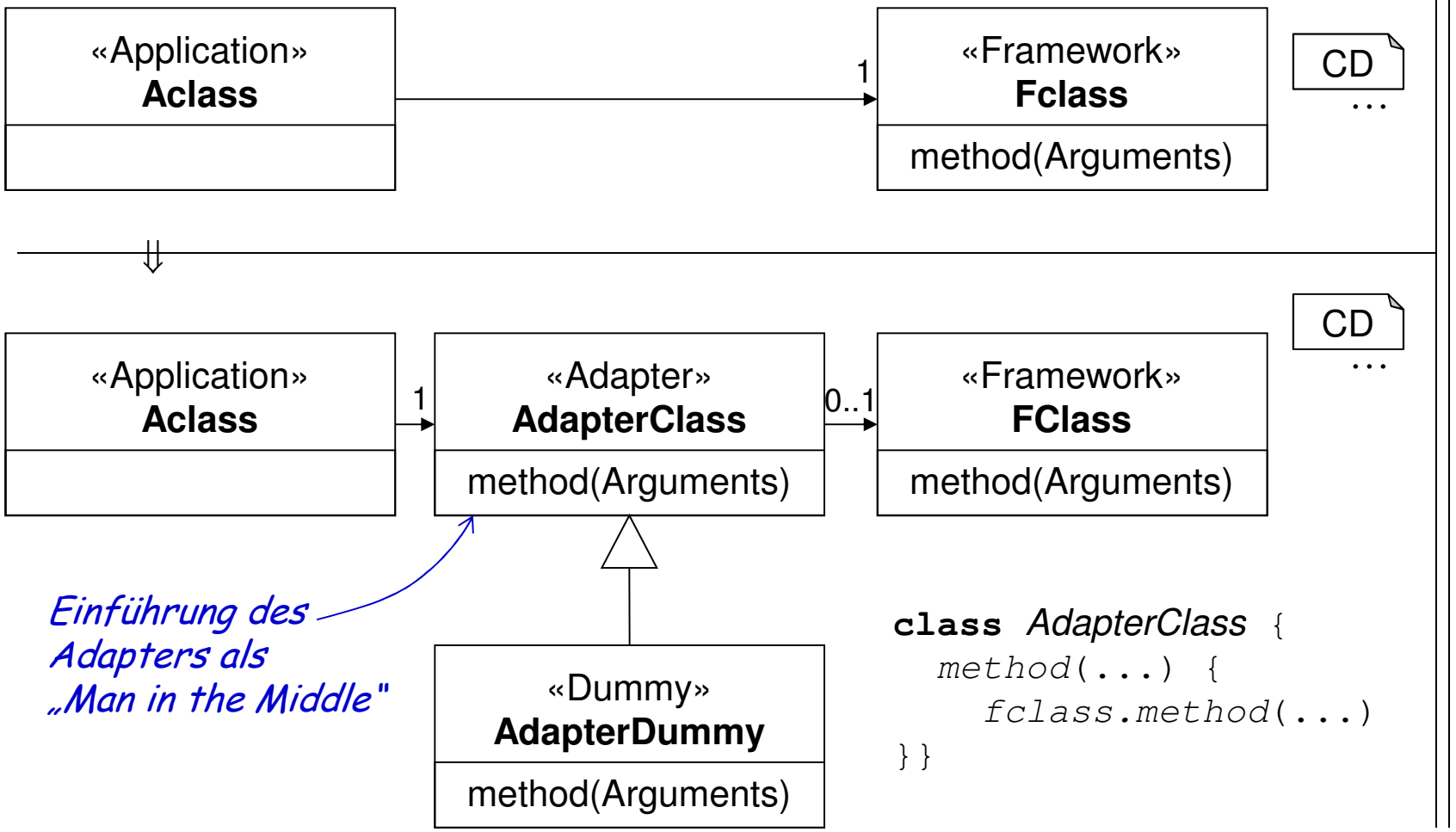
Refactoring: Entkopplung Applikation – Framework



- Problem:
  - Applikation nutzt ein Framework
  - Framework hat Seiteneffekte / DB, GUI, Web, ...
  - Nutzung des Frameworks für Tests ungeeignet

- Lösung:
  - Entkopplung durch Zwischenschaltung eines Adapters

Entkopplung mit Adapter



Große Refactorings

- sind komplexe Transformationen, die Planung erfordern
  - Idealerweise zerlegt in kleine systematische Schritte
- Beispiele:
  - Separate Domain from Representation (Fowler)
  - Convert prozedural design to OO (Fowler)
  - Entkopplung einer kompletten Applikation von einem Framework (GUI, Middleware, ...)
  - Komplexe Datenstrukturwechsel

Beispiel: Wechsel einer Datenstruktur

Vorgehensmuster:

1. alte Datenstruktur identifizieren
   hier: long durch Money ersetzen

SellItem ...
long valueInDM

2. Neue DS + Queries hinzufügen
   + compilieren & testen

SellItem ...
long valueInDM
Money value

3. Invarianten, um beide DS in Beziehung zu setzen:

```
context SellItem inv IV:
  valueInDM ==
        value.asDM()
```

4. Code für Besetzung neuer DS hinzufügen, wo immer die alte DS **verändert** wird
   + compilieren & testen

```
valueInDM = ...
value.set(...)
assert IV
```

5. Stellen mit **Nutzung** der alten DS anpassen + compilieren & testen

```
= ... valueInDM ...
        ⇓
= ... value.asDM() ...
```

6. Vereinfachen + compilieren & testen

7. Alte Datenstruktur entfernen
   + compilieren & testen

SellItem ...
Money value

Stand der Technik beim Refactoring

- Extreme Programming liefert die methodische Unterfütterung für Programmierung
- Eclipse, JUnit und Verwandte liefern Techniken zur Durchführung
  - Testfalldefinition, -ausführung, -management
  - Messung von „Code smells“ (Metriken)
  - Unterstützung für einfache Refactorings
  - Generierung von Dummies und Mock-Objekten für Tests
  - Zustand: Wird besser, aber noch viel zu tun!

- MDA liefert Methodik für modellbasierte Softwareentwicklung
  - Codegeneratoren
  - Transformationssprachen entstehen
  - Zustand: Werkzeuge sind zu langsam, ineffektiv! Noch viel zu tun!

SE SOFTWARE ENGINEERING

RWTH AACHEN

Modellbasierte Softwareentwicklung

- 10. Fin.

Danke für Ihre Aufmerksamkeit!

Prof. Dr. Bernhard Rumpe
Lehrstuhl für Software Engineering
RWTH Aachen

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