LEHRSTUHL 14
SOFTWARE ENGINEERING

Vorlesung (WS 2014/15)

Softwarekonstruktion

Prof. Dr. Jan Jürjens
TU Dortmund, Fakultät Informatik, Lehrstuhl XIV

Teil 1.1: Modellbasierte Softwareentwicklung

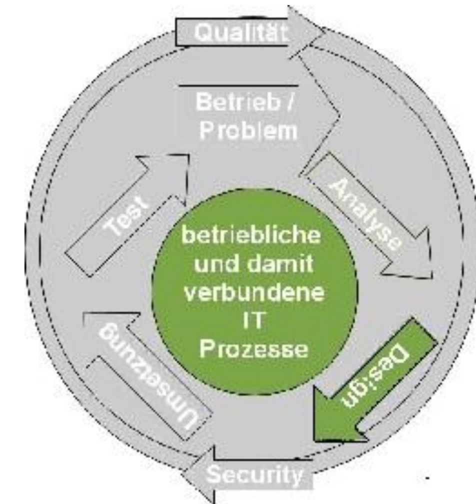
v. 27.10.2014

Modellgetriebene SW-Entwicklung

- Einführung
- Modellbasierte Softwareentwicklung
- OCL
- Ereignisgesteuerte Prozesskette (EPK)
- Petrinetze
- EMF

Qualitätsmanagement

Testen



Literatur (s. Vorlesungswebseite)**:**

- V. Gruhn: **MDA - Effektives Software-Engineering**. Kap. 2-5
- Bruegge & Dutoit: „Object-Oriented Software Engineering: Using UML, Patterns & Java“

Vorheriger Abschnitt:

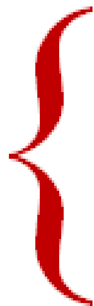
- Einführung in die modellbasierte Softwareentwicklung.

Dieser Abschnitt:

Fortgeschrittene Techniken der modellbasierten Softwareentwicklung, z.B.:

- Metamodellierung: Modell und Metamodell der UML
- UML-Erweiterungen
- Modelltransformation
- Design Patterns

1.1 Modellbasierte Software-entwicklung



- **Metamodellierung**
- UML-Erweiterungen
- Modelltransformation
- Design Patterns

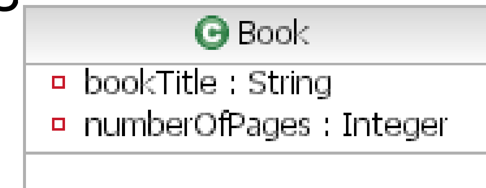
UML: Von informellem Modell zu Werkzeug

UML-Modelle: bislang

- **informell definiert** (textuelle Beschreibung)
- und **verwendet** (Skizzen auf Papier).

Aber: **UML kann mehr !**

- Codegenerierung
- Testfallgenerierung
- Modellvalidierung
- Modelltransformation
- …



Generiere Getters & Setters !

```
public String getBookTitle(){
  return bookTitle;
}
public void setBookTitle(String input){
  bookTitle = input;
}
...
```

Werkzeuge müssen UML „verstehen“ können.

➔ **Formelle Definition** der Notation.

Wie UML werkzeugkonform definieren ?

Wie UML-Notation **„formell" definieren** ?

➔ Verschiedene Möglichkeiten:
BNF-Grammatik, XML-Schema, …

Wie würde es jemand tun, der **nur UML kennt** ?

➔ **Zentrales Konzept** in UML identifizieren;
damit Menge der UML-Modelle definieren
(„**boot-strapping**").

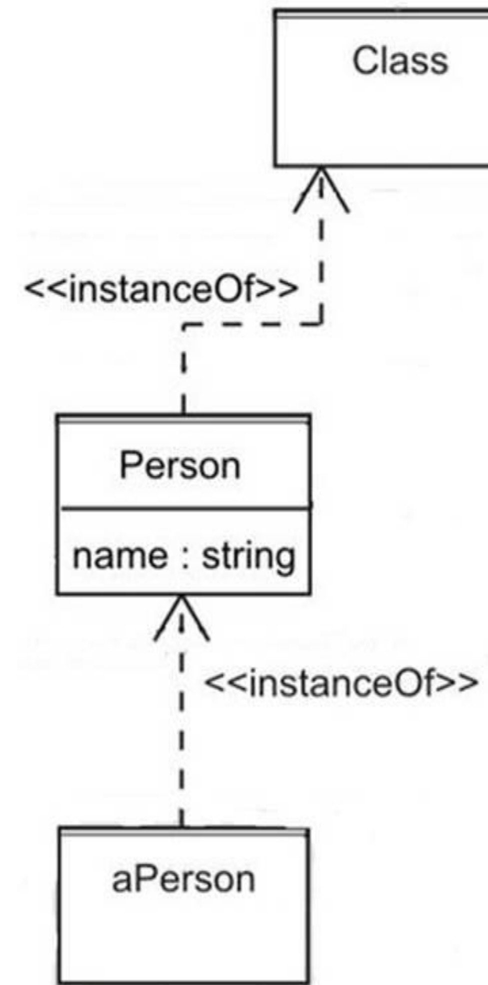
Zentrales Konzept in OO, um **Mengen von Entitäten** zu definieren ?

Zentrales OO-Konzept: Klasse-Instanz-Beziehung

Klasse *Person* im **UML-Modell** des modellierten **Systems**:
definiert Menge der Instanzen
(z.B. **Objekte** *aPerson*, *bPerson*, ….)

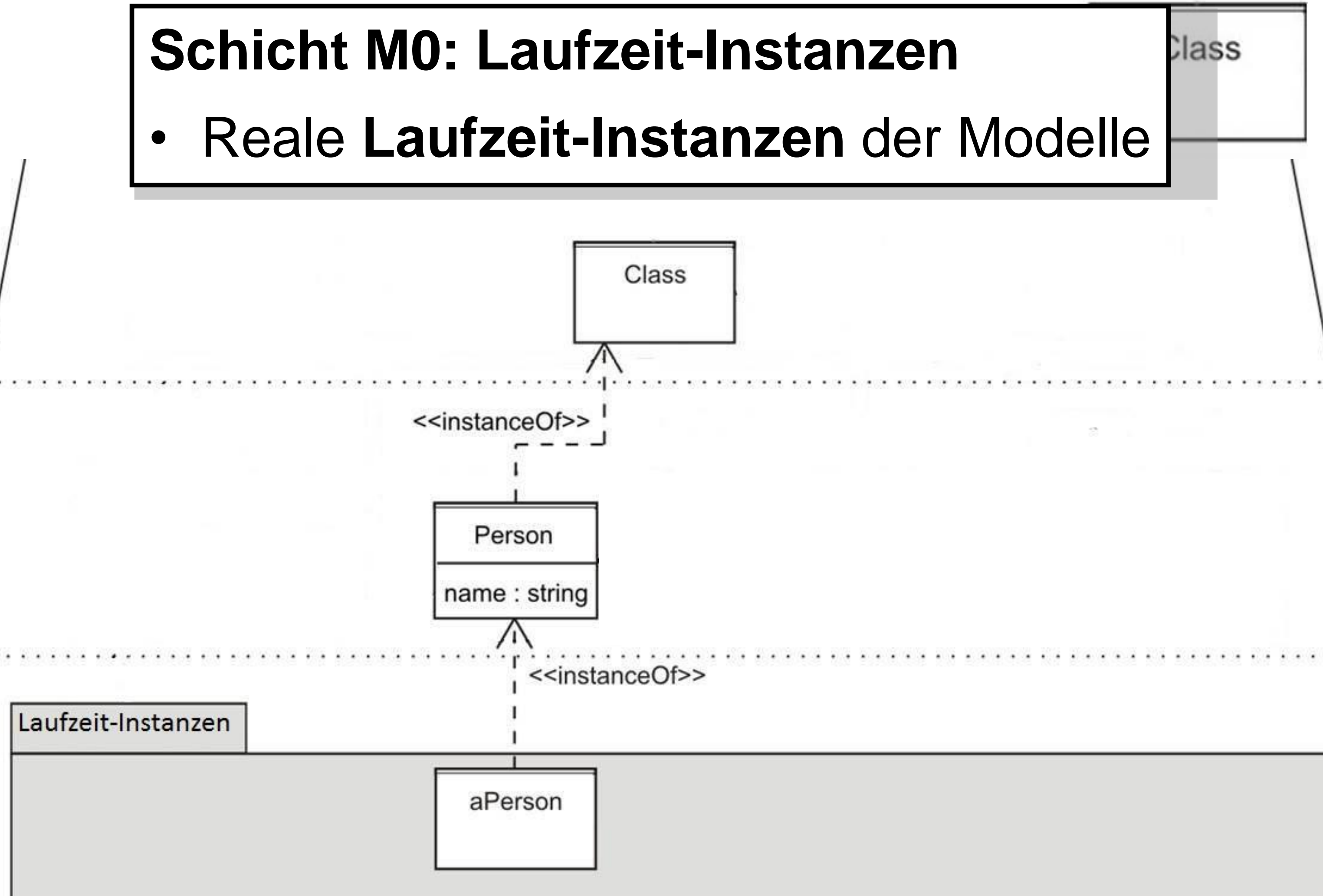
Metamodell: **Modell**, das **Notation** einer Modellierungsnotation **definiert.**

Klasse *Class* im **Definitions-Modell** der **UML** (“**Metamodell**”):
definiert Menge der **UML-Klassenmodelle**
(z.B. Klassenmodell *Person*).



Metamodellierungs-Hierarchie: Schicht M0

LEHRSTUHL 14
SOFTWARE ENGINEERING



Metamodellschicht (Schicht M2):

Metamodelle

Modellschicht (Schicht M1):

Modelle

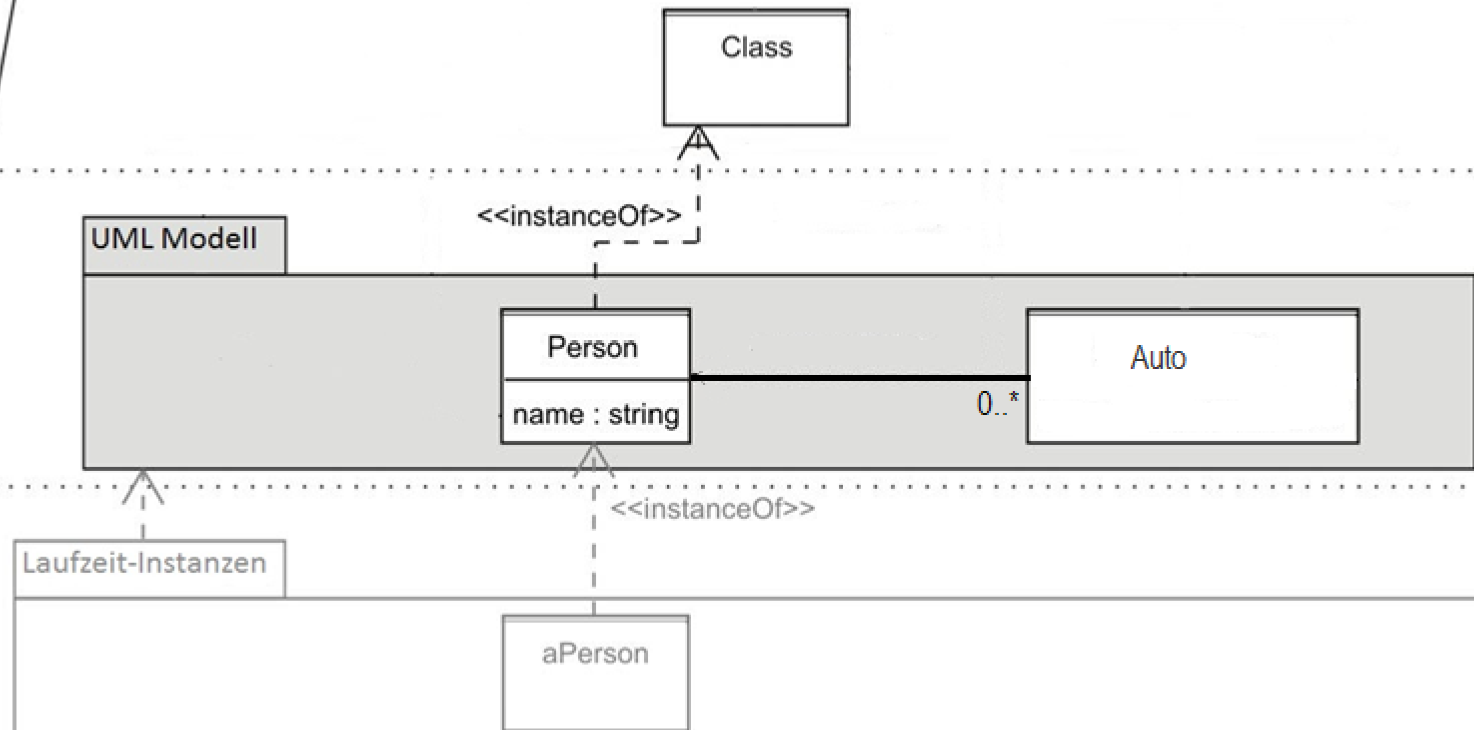
Informations-schicht (Schicht M0):

Instanzen

Metamodellierungs-Hierarchie: Schicht M1

Schicht M1: UML-Modell

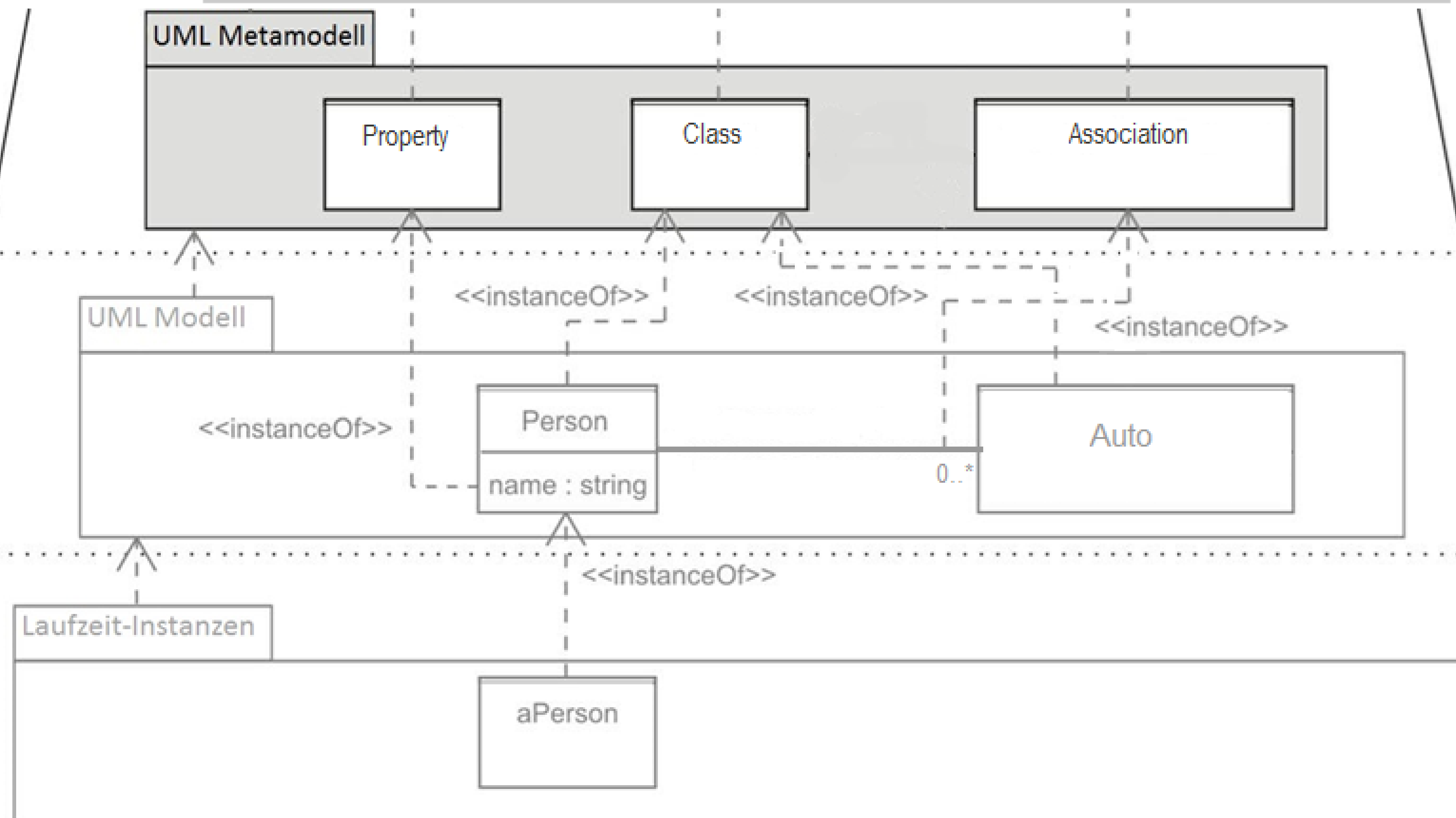
- **Instanziiert** aus **UML-Metamodell**.
- **Definiert** Menge **valider Laufzeit-Instanzen**.



Metamodellier… Schi…

Schicht M2: UML-Metamodellierung

- **Definiert** UML-Notation, d.h. Menge **valider UML-Modelle**.
- Insbesondere darin verwendete Konzepte wie *Klassen, Attribute, Assoziationen.*



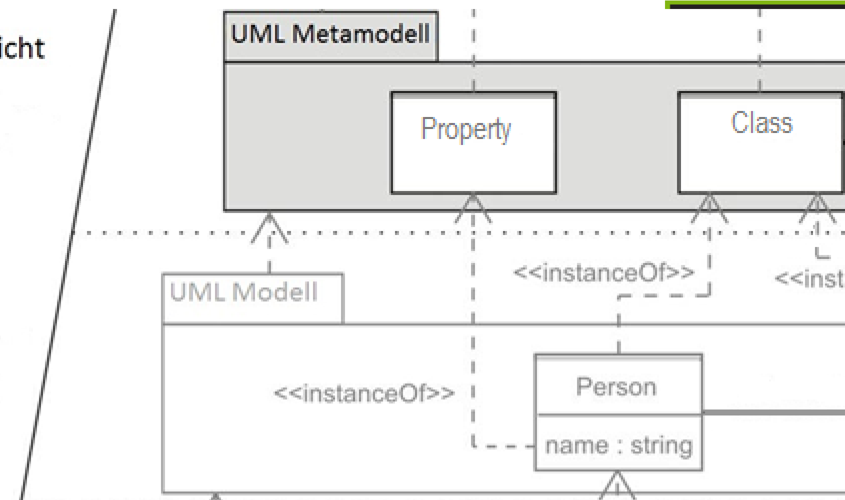
Beispiel

Metamodellschicht (Schicht M2):

Metamodelle

Modellschicht (Schicht M1):

Modelle



Klassendiagramm-Metamodell (vereinfacht)

Class — 0..1 ◆—— * Property 2..* —— 0..1 Association

Class 0..1 ◆—— * Operation

Auto

Farbe: color

+start()

Person —— 0..* Auto

<<instanceOf>>

<<instanceOf>>

<<instanceOf>>

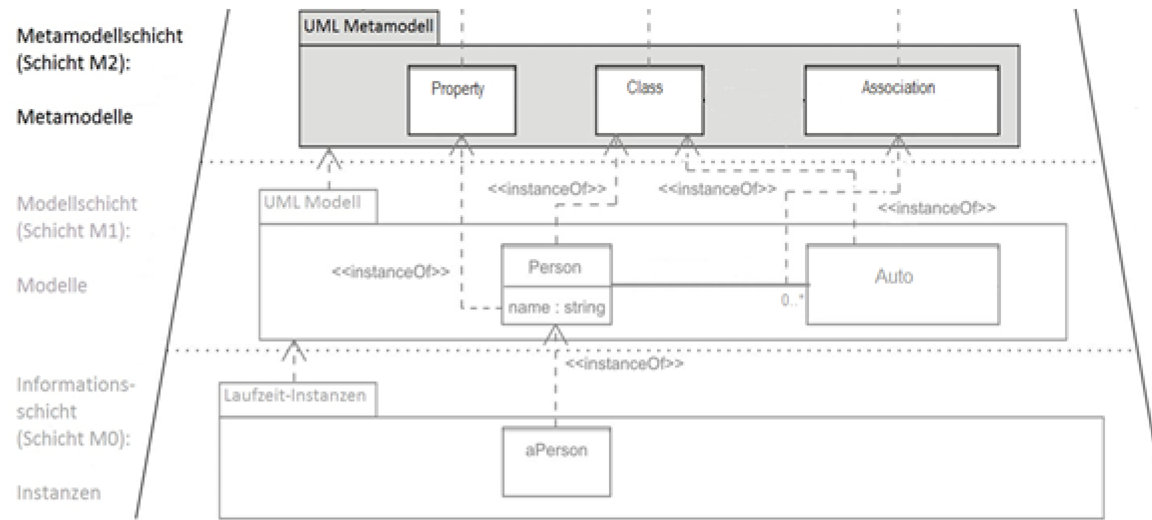
<<instanceOf>>

Sind wir jetzt fertig ?

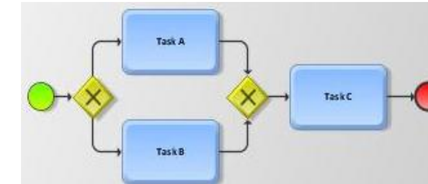
OK, Metamodelle sind cool – sind wir jetzt fertig ?

Könnten hier aufhören:

- **Klassendiagramme** grundlegendes Konzept,
- Übrige **UML-Syntax** damit **definieren**:



Metamodelle definieren ?



Aber: UML ist nicht alles !

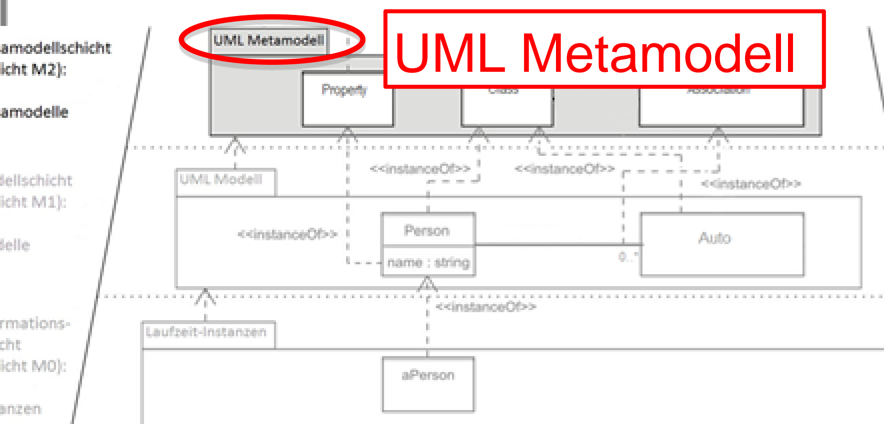
- Weitere Notationen: **eigene Metamodelle** (BPMN; Domain-Specific Languages (DSLs), …).

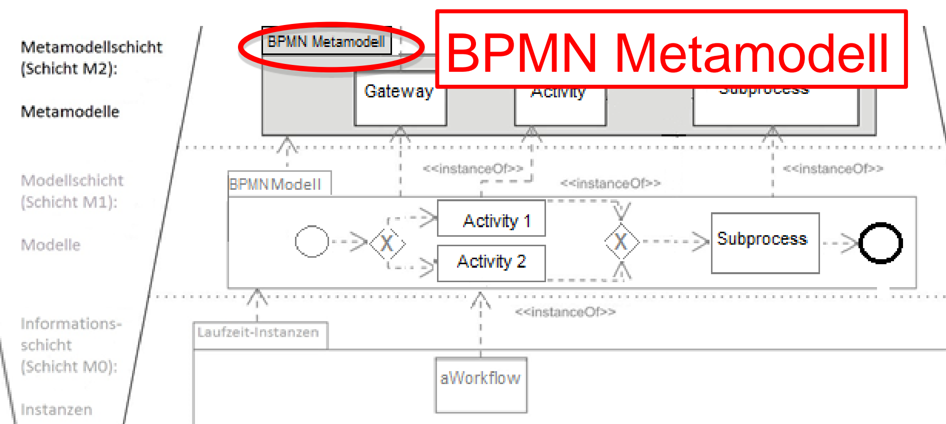
Gewünscht:

- **Wiederverwendbarkeit** der **Werkzeuge**
- **Austauschbarkeit** von **Modellen**

➔ Allgemeiner Ansatz zum **Metamodelle definieren.**

Wie ?





Metamodellierungs-Hierarchie: Schicht M3

Meta-Metamodellschicht (Schicht M3):

Meta-Metamodelle

MOF Modell

Class

<<instanceOf>>

<<instanceOf>>

<<instanceOf>>

Metamodellschicht (Schicht M2):

Metamodelle

UML Metamodell

Property

Class

classifier

Association

<<instanceOf>>

<<instanceOf>>

Modellschicht (Schicht

Modelle

Informa schicht (Schicht

Instanze

Schicht M3: Meta-Metamodelle

Definiert Menge **valider Metamodelle.**

Konkreter Ansatz der OMG: **Meta Object Facility (MOF).**

➔ Mit MOF definierte Modelle:

- Austauschbar mit XML Metadata Interchange (XMI)

Metamodellierungs-Hierarchie vs. UML

LEHRSTUHL 14
SOFTWARE ENGINEERING

Meta-Metamodellschicht (Schicht M3):

Meta-Metamodelle

MOF Modell

Class

<<instanceOf>> <<instanceOf>> <<instanceOf>>

Metamodellschicht (Schicht M2):

Metamodelle

UML Metamodell

Property

Class

classifier

Association

Modellschicht (Schicht M1):

Modelle

UML Modell

<<instanceOf>> <<instanceOf>> <<instanceOf>> <<instanceOf>>

Person

name : string

0..*

Auto

Informatio... schicht (Schicht M...

Instanzen

Endlich fertig: **Metamodelle** von UML, BPMN, DSLs, …

➔ mit MOF **definierbar**.

Hier: UML-spezifische Hierarchie

Diskussionsfrage: MDA Standards

Welche **Aussagen** passen zu den **Kernelementen der UML** ?

M3-Ebene („MOF“)

Definiert bekannte Diagrammarten (wie Klassen- / Aktivitätsdiagramm).

Definiert UML-Metamodell.

XMI

Format für Austausch der Modelle.

M2-Ebene

Diskussionsfrage: MDA Standards

Welche **Aussagen** passen zu den **Kernelementen der UML** ?

M3-Ebene („MOF“)

XMI

M2-Ebene

Definiert bekannte Diagrammarten (wie Klassen- / Aktivitätsdiagramm).

Definiert UML-Metamodell.

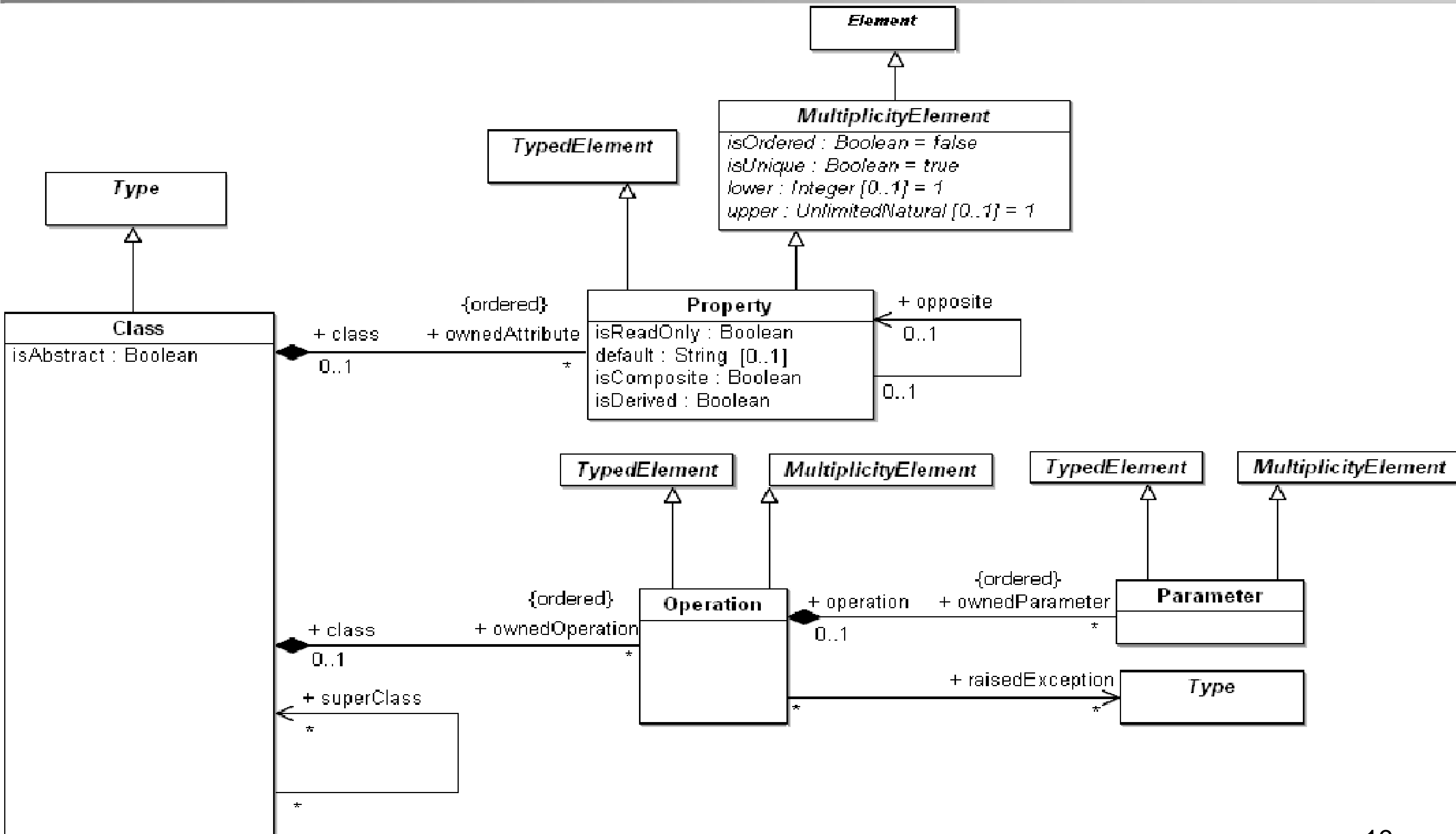
Format für Austausch der Modelle.

Beispiel MOF-Implementierung: Eclipse Modeling Framework (EMF)

	OMG	EMF
M3 (Metametamodel)	MOF	Ecore (Java classes)
M2 (Metamodel)	UML, CWM, ...	EMF model → EMF codegen → Java code for manipulation and default serialization of EMF models
M1 (Model)		EMF model

EMF: Framework für **Modellierung & Code-Generierung (inkl. Editor-Generierung)** mit „Essential MOF“.

UML Metamodell Beispiel
Core::Basic:ClassDiagram



Diskussionsfrage: Klassendiagramm vs. Metamodell

Wo finden sich die **rot markierten Elemente** aus dem **Klassendiagramm** im **Metamodell** wieder ?
(Zur Erinnerung: Attribut in UML-Metamodell als „Property“ definiert.)

Element

MultiplicityElement
isOrdered : Boolean = false
isUnique : Boolean = true
lower : Integer [0..1] = 1
upper : UnlimitedNatural [0..1] = 1

TypedElement

Type

Class
isAbstract : Boolean

{ordered}
+ class
+ ownedAttribute
0..1
*

Property
isReadOnly : Boolean
default : String [0..1]
isComposite : Boolean
isDerived : Boolean

+ opposite
0..1
0..1

TypedElement
MultiplicityElement
TypedElement
MultiplicityElement

{ordered}
+ class
+ ownedOperation
0..1
*

Operation

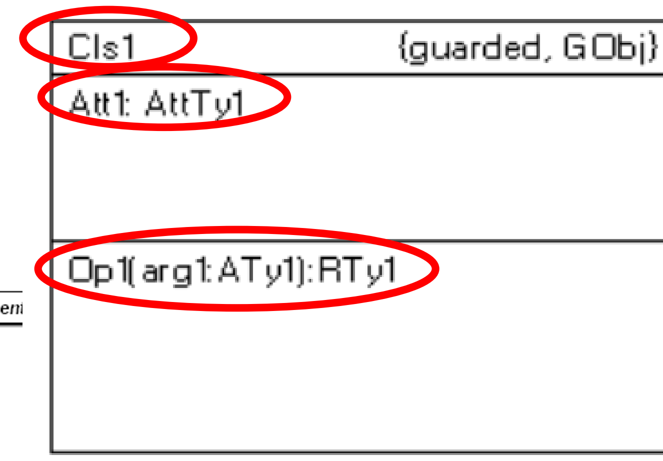
+ operation
{ordered}
+ ownedParameter
0..1
*

Parameter

+ raisedException
*
*

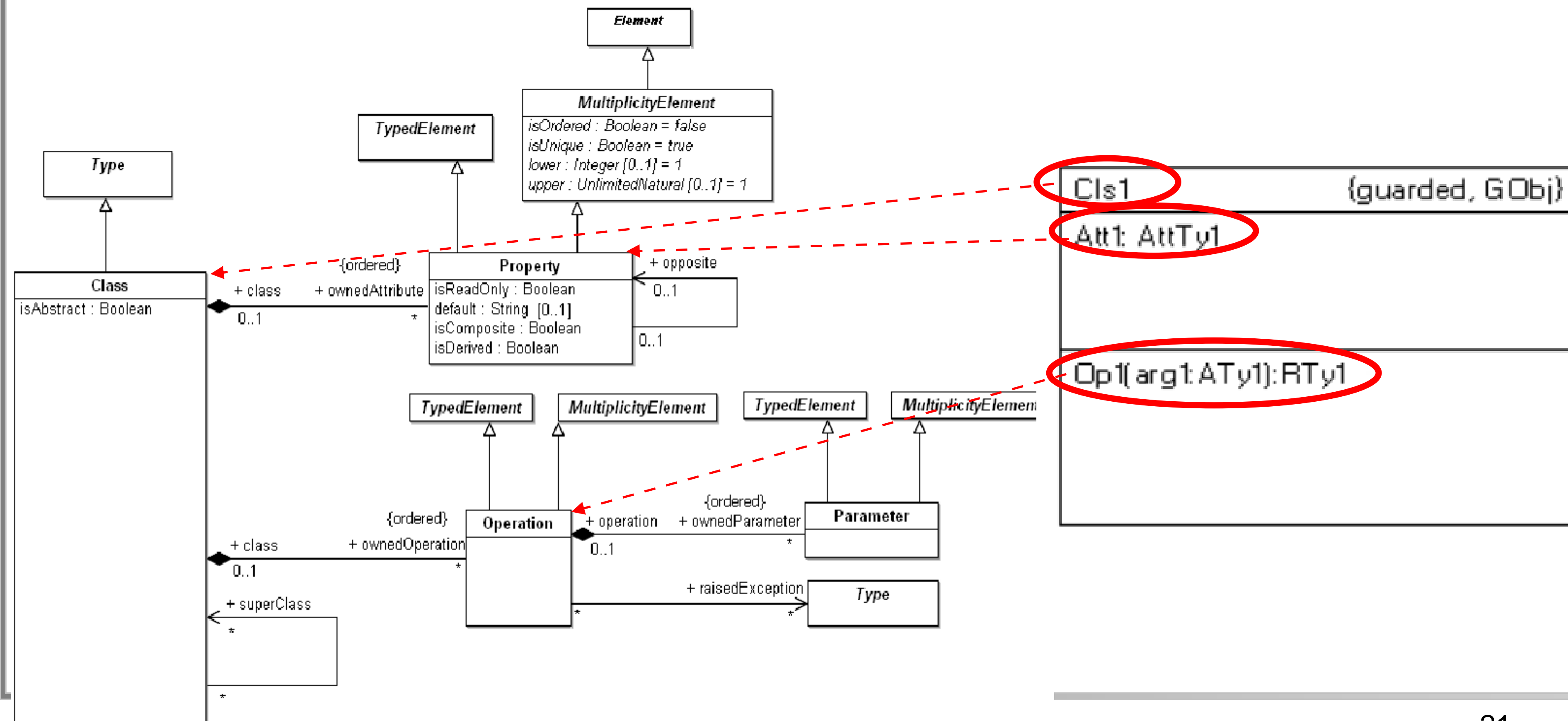
Type

+ superClass
*
*



Diskussionsfrage: Klassendiagramm vs. Metamodell

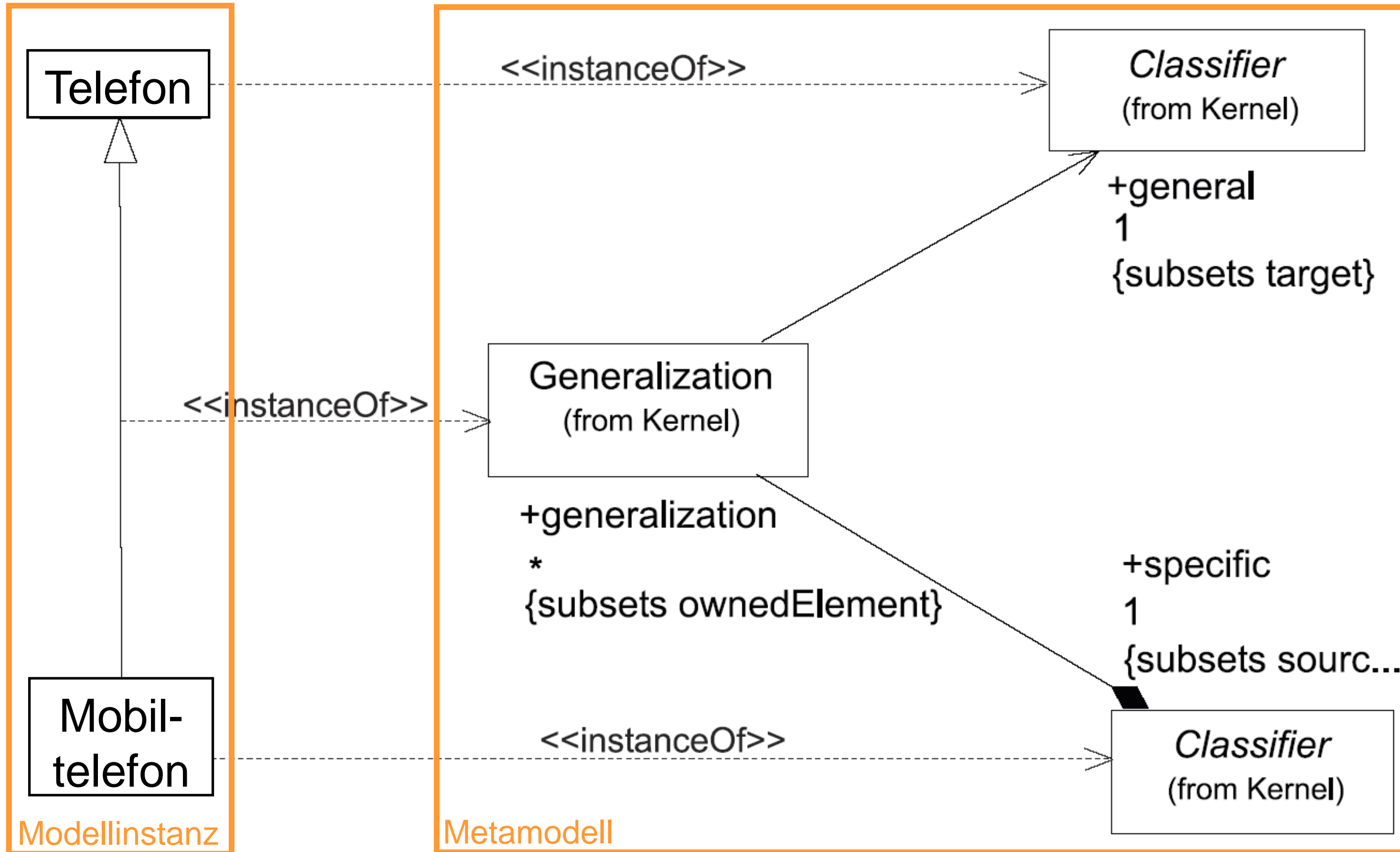
Wo finden sich die **rot markierten Elemente** aus dem **Klassendiagramm** im **Metamodell** wieder ?
(Zur Erinnerung: Attribut in UML-Metamodell als „Property“ definiert.)



Metamodell der UML
Beispiel: Generalisierung

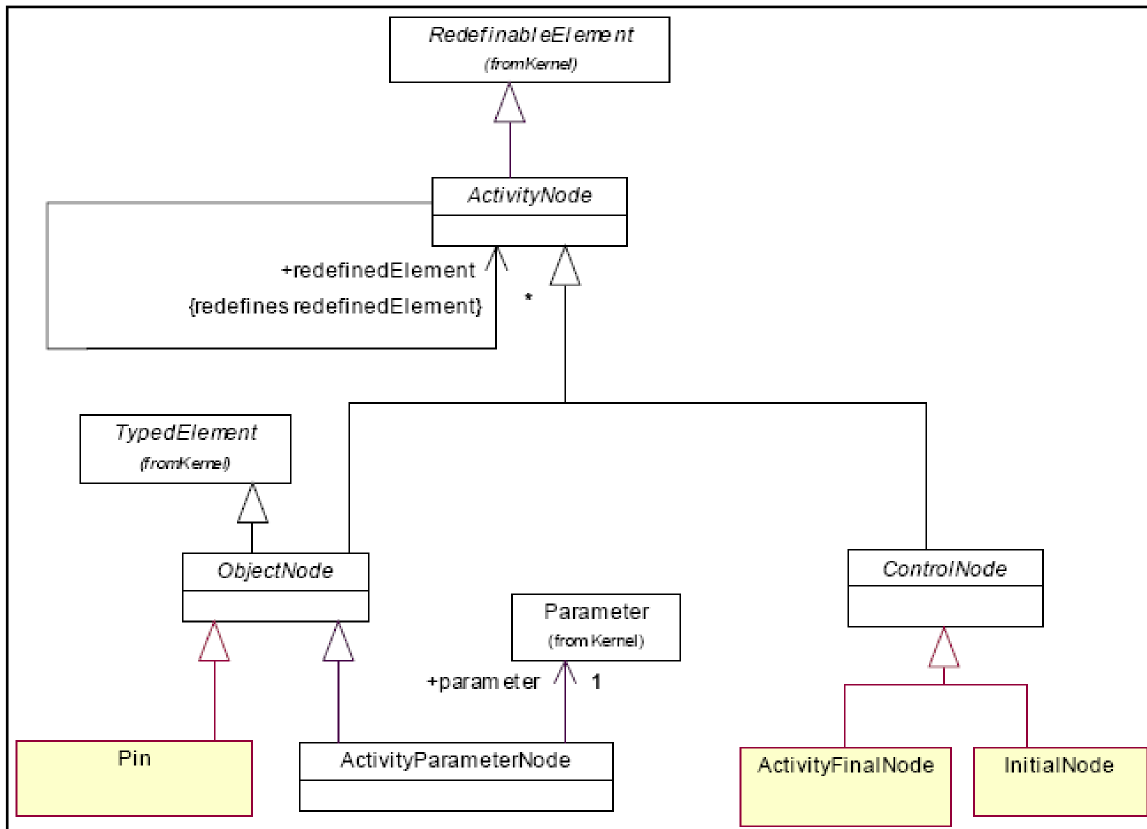
Beispiel einer Generalisierung:

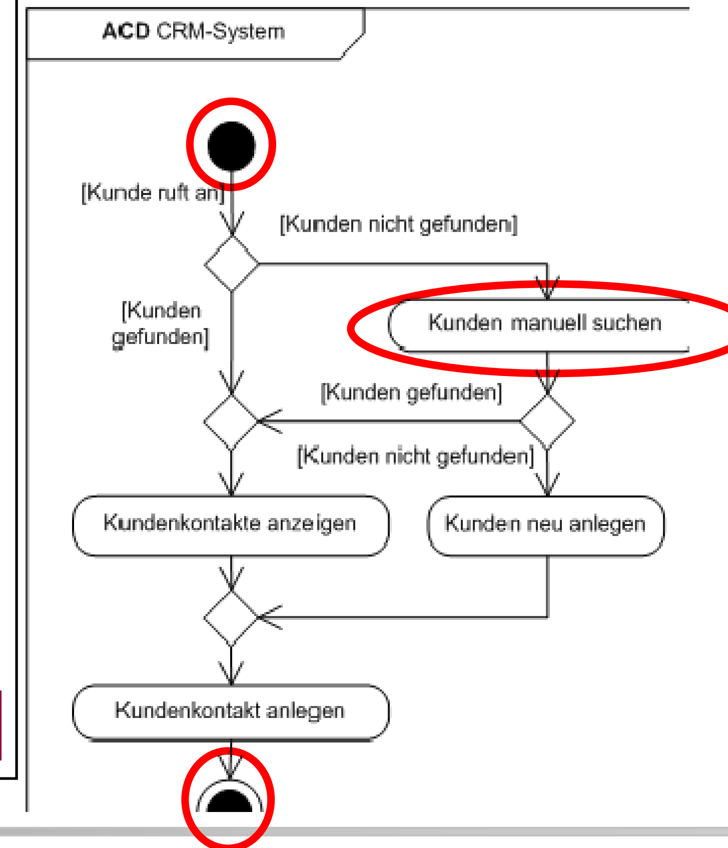
Allgemeine Definition der Generalisierung im UML-Metamodell:



Diskussionsfrage: Aktivitätsdiagramm vs. Metamodell

Wo finden sich die **rot markierten Elemente** aus dem **Aktivitätsdiagramm** im **Metamodell** wieder ?

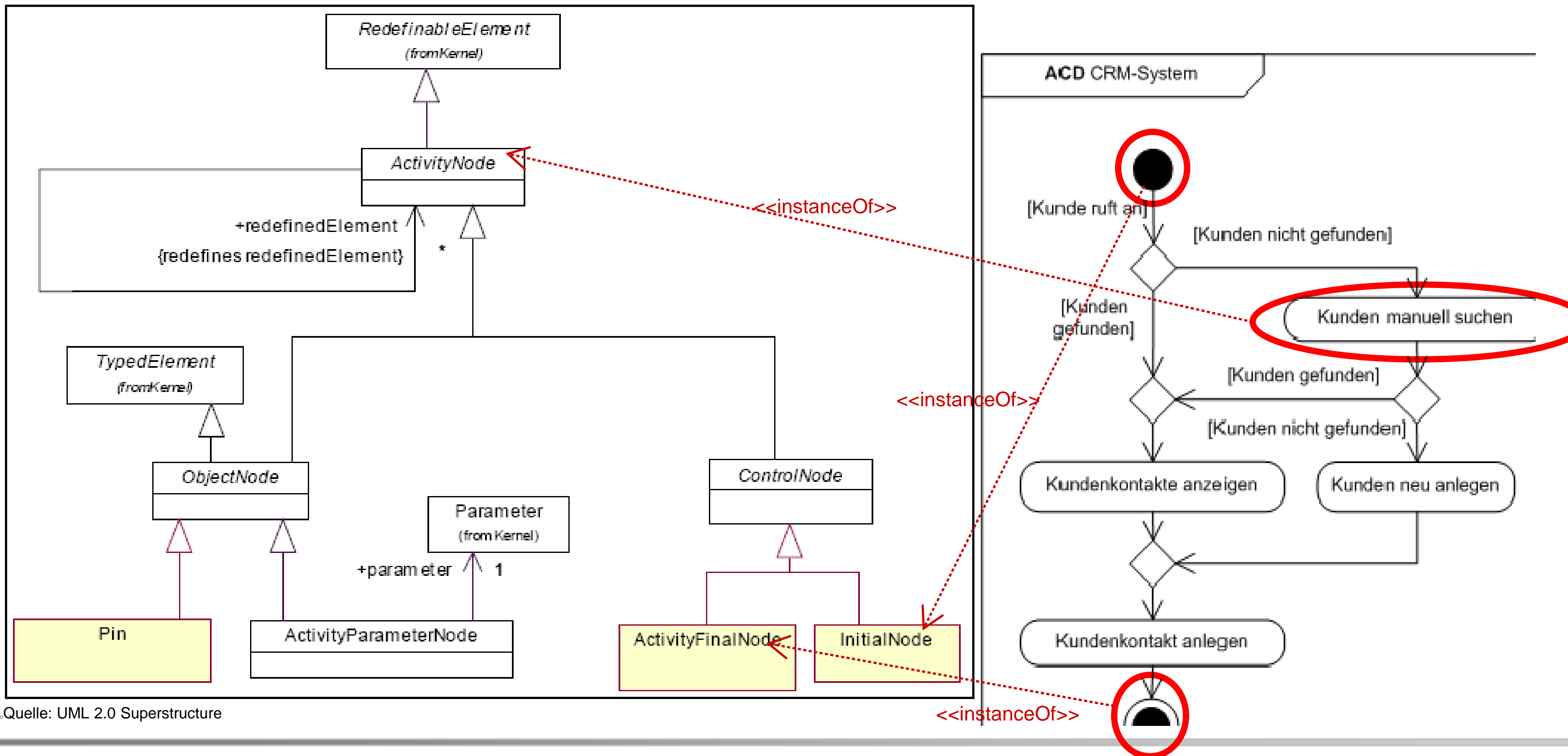




Quelle: UML 2.0 Superstructure

Diskussionsfrage: Aktivitätsdiagramm vs. Metamodell

Wo finden sich die **rot markierten Elemente** aus dem **Aktivitätsdiagramm** im **Metamodell** wieder ?



Quelle: UML 2.0 Superstructure

Vorteile von Metamodellierung

Vorteile im **Allgemeinen:**

- **Präzise** Definition von Modellierungsnotation
- **Mehrere** Notationen, **einheitlicher** Ansatz (UML, BPMN, DSLs, …)

Vorteile von **MOF:**

- **Wohlverstandene** Notation für Metamodelle (UML-Klassendiagramme)
- Weitgehende **Werkzeugunterstützung** (Modelltransformation, Codegenerierung etc)

Nachteile von Metamodellierung

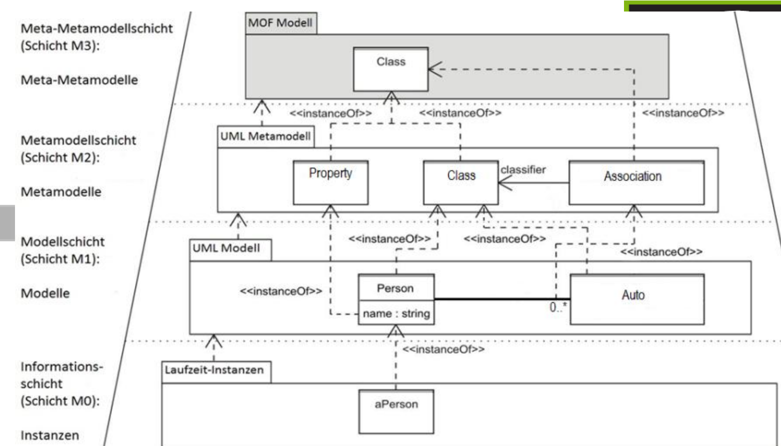
Nachteile im **Allgemeinen**:

- Erstellung Metamodell:
  Benötigt **Aufwand** und **Expertise**.

Nachteile von **MOF**:

- Dieselbe Notation (Klassendiagramme) auf verschiedenen Ebenen (inkl. M2, M3) evt. **verwirrend**.
- **Logisch zirkulär** (definiere Klassendiagramme mit Klassendiagrammen).

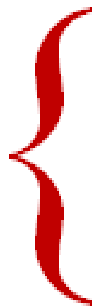
Zusammenfassung: Metamodellierung



Einführung Metamodellierung:

- **Metamodellierungshierarchie**
- Ausschnitt: Metamodell für **Klassendiagramm**
- **Werkzeugunterstützung** (EMF)
- **Vorteile / Nachteile**

1.1 Modellbasierte Software-entwicklung



- Metamodellierung
- **UML-Erweiterungen**
- Modelltransformation
- Design Pattern

Erweiterung der UML
Motivation

- **Fehlendes Fachwissen:** Großes und teures Problem bei Entwicklung und Wartung von IT-Systemen.
- **Lösungsansatz:** Aufnahme von fachlichen Konzepten in SW-Modell.
  - Dokumentiert **fachliche Zusammenhänge**.

    → Ermöglicht **Codegenerierung** bestimmter fachlicher Aspekte im Rahmen modellgetriebener SW-Entwicklung.

Erweiterung der UML
Problem

- UML abstrahiert von jeder **fachlichen Domäne.**
- Fachliche (oder spezielle technische) Konzepte in UML beschreiben.
- Definition der (Domänen-) Konzepte benötigt.
  - Erklärender Text zu einem Diagramm. → Ungeeignet.
  - Erweiterungen der UML-Notationselemente um fachliche Konzepte.

➔ UML-Erweiterung !

Erweiterung der UML
Lösungsmöglichkeiten

LEHRSTUHL 14
SOFTWARE ENGINEERING

Möglichkeiten der Erweiterung / Neudefinition einer Modellierungsnotation mittels Metamodellen:

A) Eigenes Metamodell „from scratch“:

- Aufwändig.

B) Direkte, beliebige **Erweiterung eines Metamodells:**

- Voraussetzung: Uneingeschränkter Zugriff auf Metamodell.
- Verletzt evt. existierende Semantik.

C) Erweiterung des UML-Metamodells durch **UML Profile**:

- Vorgesehene Schnittstelle zum UML-Metamodell.
- Lediglich Verfeinerung des UML-Metamodells.

➔ C) einfachste und daher meist beste Lösung; vgl. im Folgenden.

UML-Profil:

- Spezialisierung von Standard UML-Elementen zu konkreten Metatypen.
- Profil zu Modell hinzufügbar und im gesamten Modell verfügbar.
- Verschiedene Profile für verschiedene Anwendungsdomänen.
- Einige Profile vordefiniert und bei OMG verfügbar.

Definition eines Profils:

- Paket von Stereotypen und Tagged Values.
- Klassendiagramm definiert Beziehungen zwischen neuem Stereotyp und dem zu beschreibenden Element.

UML Profile
Erweiterungsmöglichkeiten

Stereotyp:

- **spezialisiert** Benutzung von Modellelementen <<*label*>>. (kann auch durch Symbol visualisiert werden, z.B. für <<subsystem>>).

Tagged value:

- **fügt** *{tag=value}* Paare zu stereotypisierten Elementen hinzu.

Constraint:

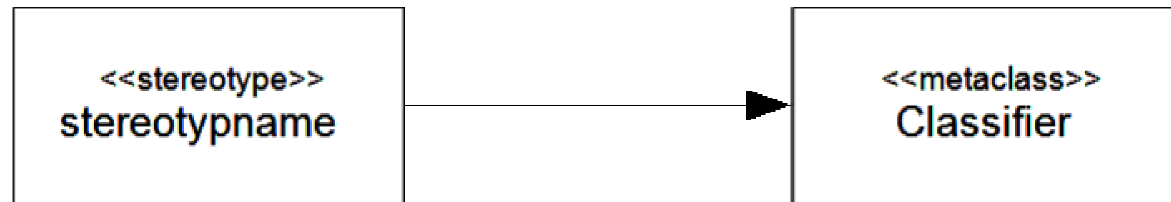
- **verfeinert** Semantik eines stereotypisierten Elements.
- z.B. mittels Object Constraint Language (OCL), s. Abschn. 1.2

Profil:

- **sammelt** obige Informationen.

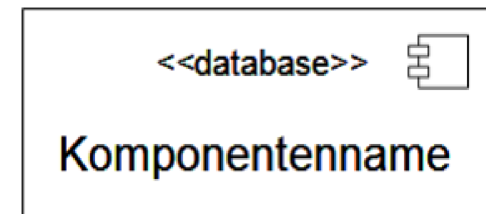
Definition von Stereotypen:
Definition neuer Stereotypen gemäß **Metamodell**:
(NB: Der Pfeil ist eine Spezialisierung.)

Definition des Stereotyp-Begriffs (**M3-Ebene**)



Definition eines konkreten Stereotypen (**M2-Ebene**):

Spätere Verwendung im Komponentendiagramm (**M1-Ebene**):



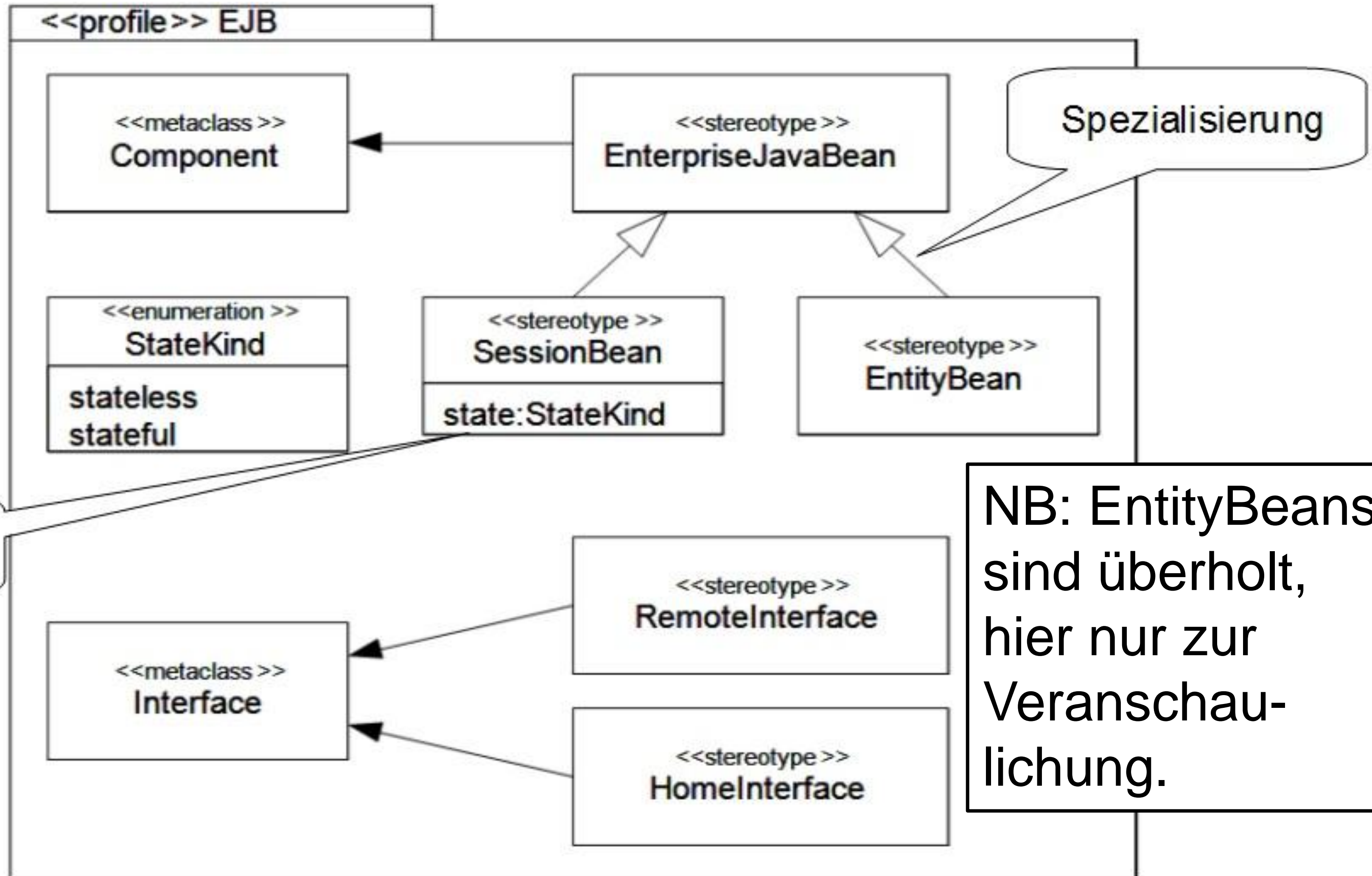
Tagged Values:

- Werte beschreiben **Eigenschaften eines Stereotypen** (fachliches Konzept).
- Tagged Values: **Name-Wert Paare.**
- **Einsatz** von Tagged Values im Modell:
  - **Kommentarfeld**, das mit Element verbunden ist oder
  - **Geschweifte Klammern** {Name = Wert} direkt in Element geschrieben.

Vordefinierte Stereotypen im UML-Standard (Auszug):

Name	**Language Unit**	**Applies to**
«document»	Deployments:: Artifacts	Artifact
«entity»	Components:: BasicComponents	Component
«executable»	Deployments:: Artifacts	Artifact
«file»	Deployments:: Artifacts	Artifact
«implement»	Components:: BasicComponents	Component
«library»	Deployments:: Artifacts	Artifact
«process»	Components:: BasicComponents	Component
«realization»	Classes::Kernel	Classifier
«service»	Components:: BasicComponents	Component
«source»	Deployments:: Artifacts	Artifact
«specification»	Classes::Kernel	Classifier
«subsystem»	Components:: BasicComponents	Component
«buildComponent»	Components:: BasicComponents	Component
«metamodel»	AuxilliaryConstructs:: Models	Model
«systemModel»	AuxilliaryConstructs:: Models	Model

Beispiel-UML-Profil „Enterprise Java Beans“: Metamodell

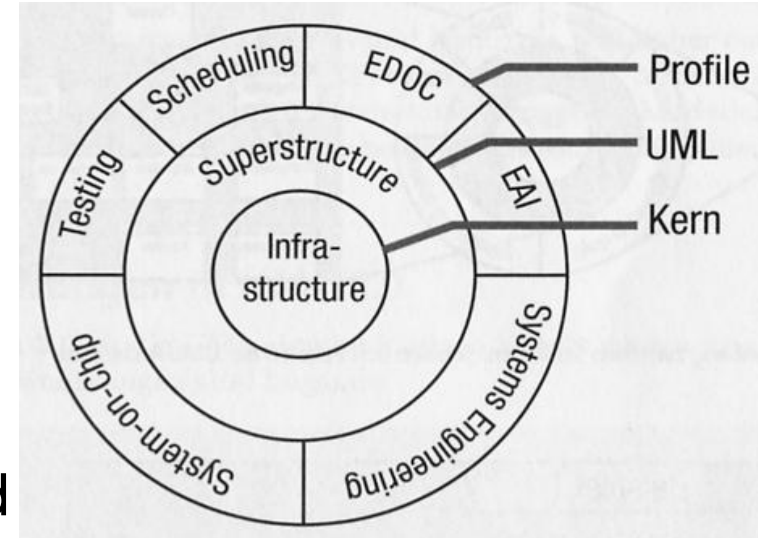


NB: EntityBeans sind überholt, hier nur zur Veranschau-lichung.

Standardisierte Profile der UML:

Allgemeine Profile für besondere Einsatzdomänen:

- Real-time UML, UMLsec.
- Enterprise Application Integration (EAI); Testing; Schedulability, Performance und Time Specification.



Profile für Implementierung in speziellen Programmiersprachen:

- C++, C#, Java

Profile für die spezielle Komponentenmodelle:

- JEE/EJB, COM, .NET, CCM (CORBA Component Model).

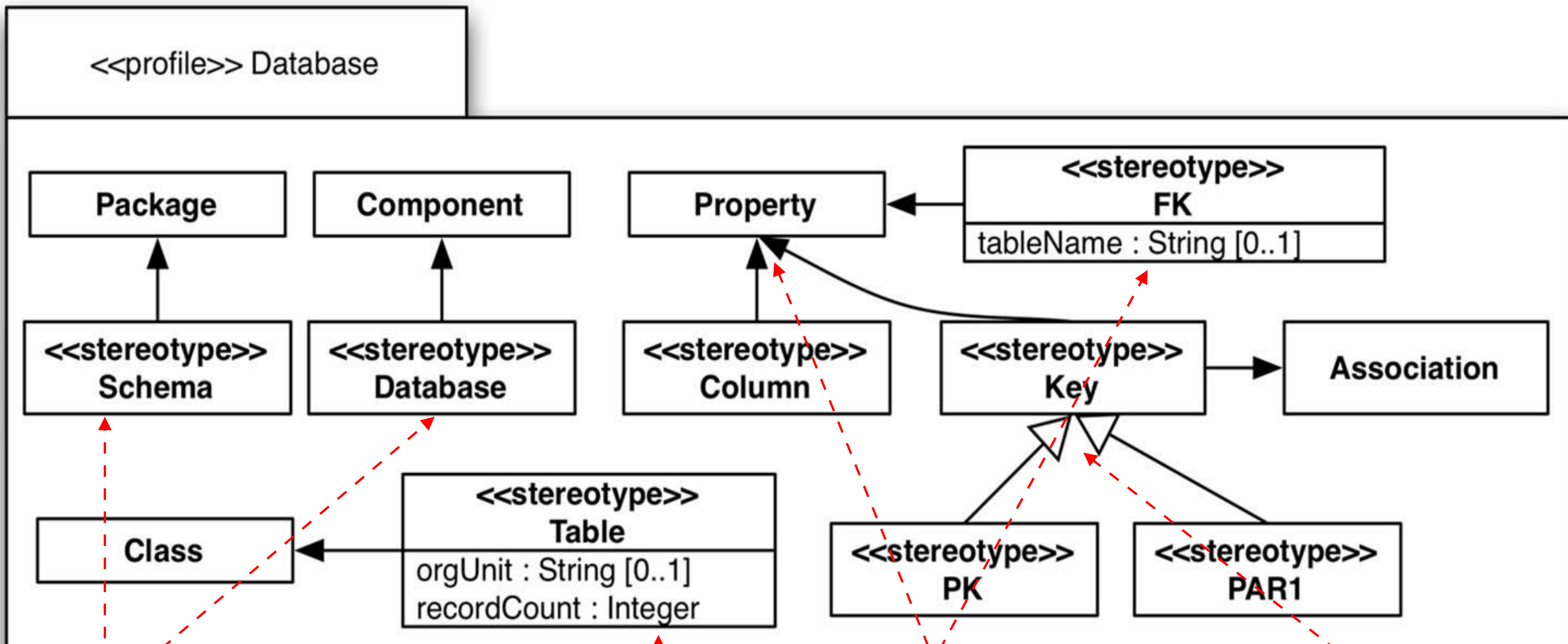
Beispiel: Profil für Datenbank

Grundlegende Konstrukte und ihre Darstellung:

- **Datenbank:** UML-Komponente mit Stereotyp <<Database>>.
- **Schema:** UML-Paket mit Stereotyp <<Schema>>.
- **Tabelle:** UML-Klasse mit Stereotyp <<Table>>.
- **Spalten:** Attribute der als <<Table>> markierten Klasse mit Stereotyp <<Column>> versehen.
- **Primärschlüssel:** Attribut markiert mit Stereotyp <<PK>> für Primärschlüssel und <<PAR1>> für Sekundärschlüssel (beide abgeleitet von <<Key>>).
- **Fremdschlüssel:** Attribut markiert mit Stereotyp <<FK>>.

Diskussionsfrage: Metamodell für technisches Datenmodell

Zu welchen Metamodellelementen gehören die Stereotyp-Definitionen ?



- **Datenbank:** UML-Komponente mit Stereotyp <<Database>>.
- **Schema:** UML-Paket mit Stereotyp <<Schema>>.
- **Tabelle:** UML-Klasse mit Stereotyp <<Table>>.
- **Spalten:** Attribute (Property) der als <<Table>> markierten Klasse; mit Stereotyp <<Column>> versehen.
- **Primärschlüssel:** Attribut markiert mit Stereotyp <<PK>> für Primärschlüssel und <<PAR1>> für (zusammengesetzte) Sekundärschlüssel (beide Stereotype abgeleitet von <<Key>>).
- **Fremdschlüssel:** Attribut markiert mit Stereotyp <<FK>>.

Beispiel technisches Datenmodell

<<Table>> Person
<<Column>> <<PK>> kdNr : Ganzzahl
<<Column>> name : Zeichenkette
<<Column>> vorname : Zeichenkette
<<Column>> adresse_hausNr : Ganzzahl
<<Column>> adresse_str : Zeichenkette
<<Column>> adresse_ort : Zeichenkette
<<Column>> adresse_plz : Zeichenkette

<<Table>> Auftrag
<<Column>> <<PK>> lfdNr : Ganzzahl
<<Column>> datum : Datum
<<Column>> <<PK>> <<FK>>
auftraggeber_kdNr : Ganzzahl

<<Table>> Lager
<<Column>> <<PK>> ort_hausNr : Ganzzahl
<<Column>> <<PK>> ort_strasse : Zeichenkette
<<Column>> <<PK>> ort_ort : Zeichenkette
<<Column>> <<PK>> ort_plz : Zeichenkette

<<Table>> Auftragslager
<<Column>> <<PK>> <<FK>> auftrag_lfdNr : Ganzzahl
<<Column>> <<PK>> <<FK>> auftrag_auftraggeber_kdNr : Ganzzahl
<<Column> <<PK>> <<FK>> lager_ort_hausNr : Ganzzahl
<<Column> <<PK>> <<FK>> lager_ort_strasse : Zeichenkette
<<Column>> <<PK>> <<FK>> lager_ort_ort : Zeichenkette
<<Column> <<PK>> <<FK>> lager_ort_plz : Zeichenkette

Beispiel für technisches Datenmodell mittels definiertem UML-Profil.

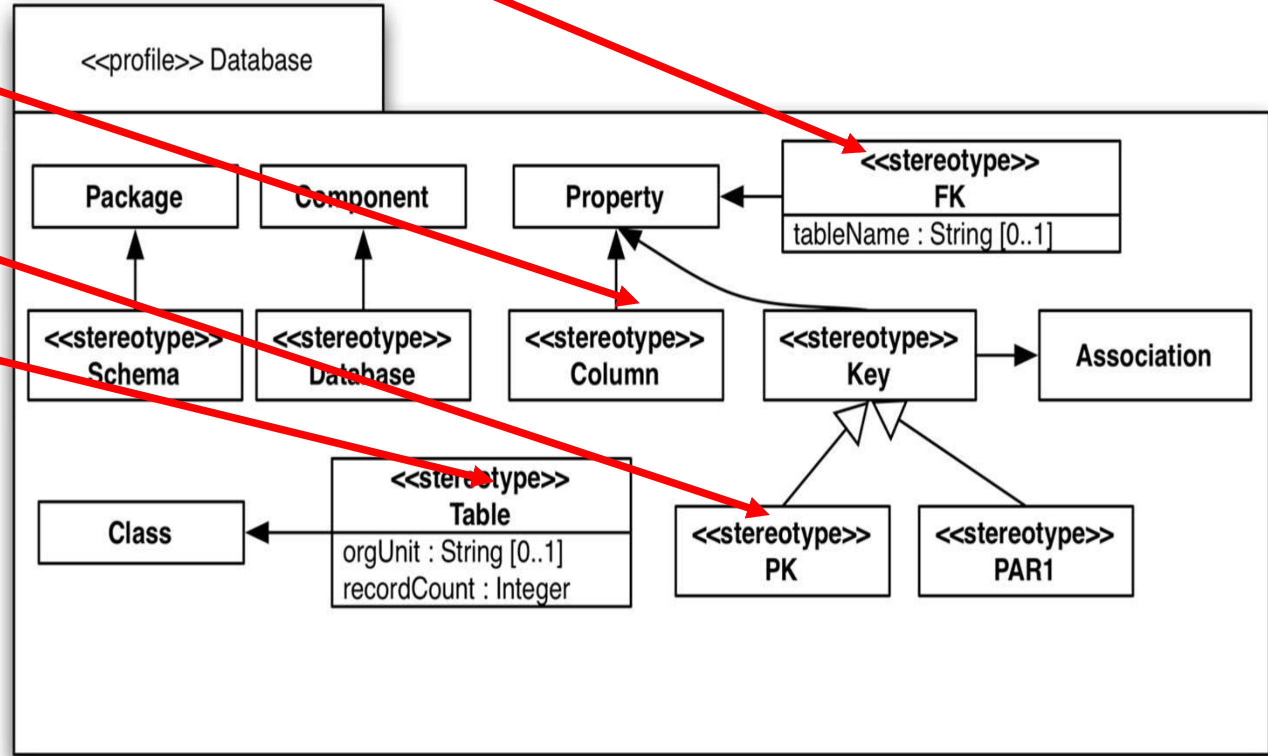
Beispiel technisches Datenmodell
Modell vs. Metamodell

<<Table>> Person
<<Column>> <<PK>> kdNr : Ganzzahl
<<Column>> name : Zeichenkette
<<Column>> vorname : Zeichenkette
<<Column>> adresse_hausNr : Ganzzahl
<<Column>> adresse_str : Zeichenkette
<<Column>> adresse_ort : Zeichenkette
<<Column>> adresse_plz : Zeichenkette

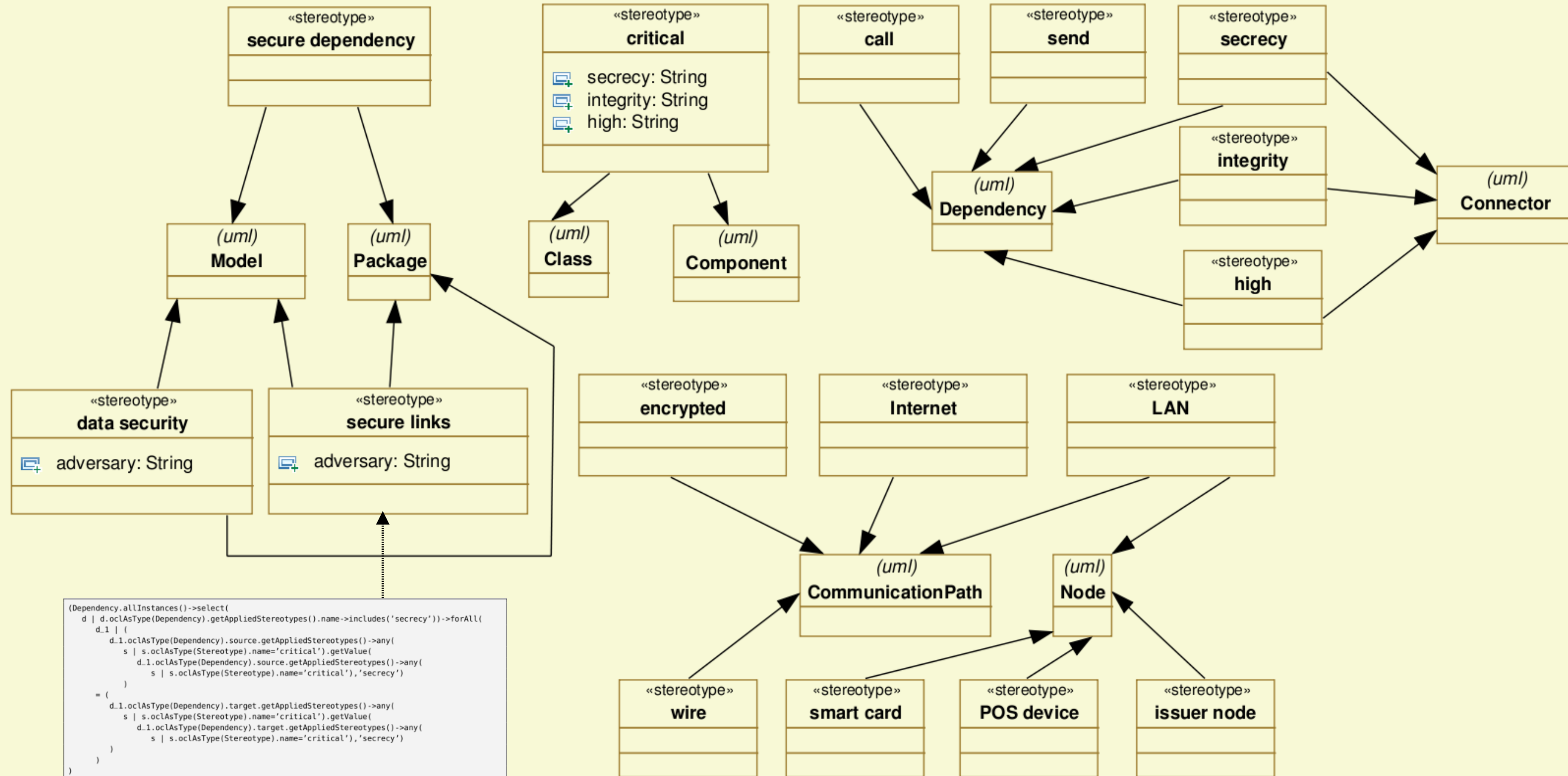
<<Table>> Auftrag
<<Column>> <<PK>> lfdNr : Ganzzahl
<<Column>> datum : Datum
<<Column>> <<PK>> <<FK>>
auftraggeber_kdNr : Ganzzahl

<<Table>> Lager
<<Column>> <<PK>> ort_hausNr : Ganzzahl
<<Column>> <<PK>> ort_strasse : Zeichenkette
<<Column>> <<PK>> ort_ort : Zeichenkette
<<Column>> <<PK>> ort_plz : Zeichenkette

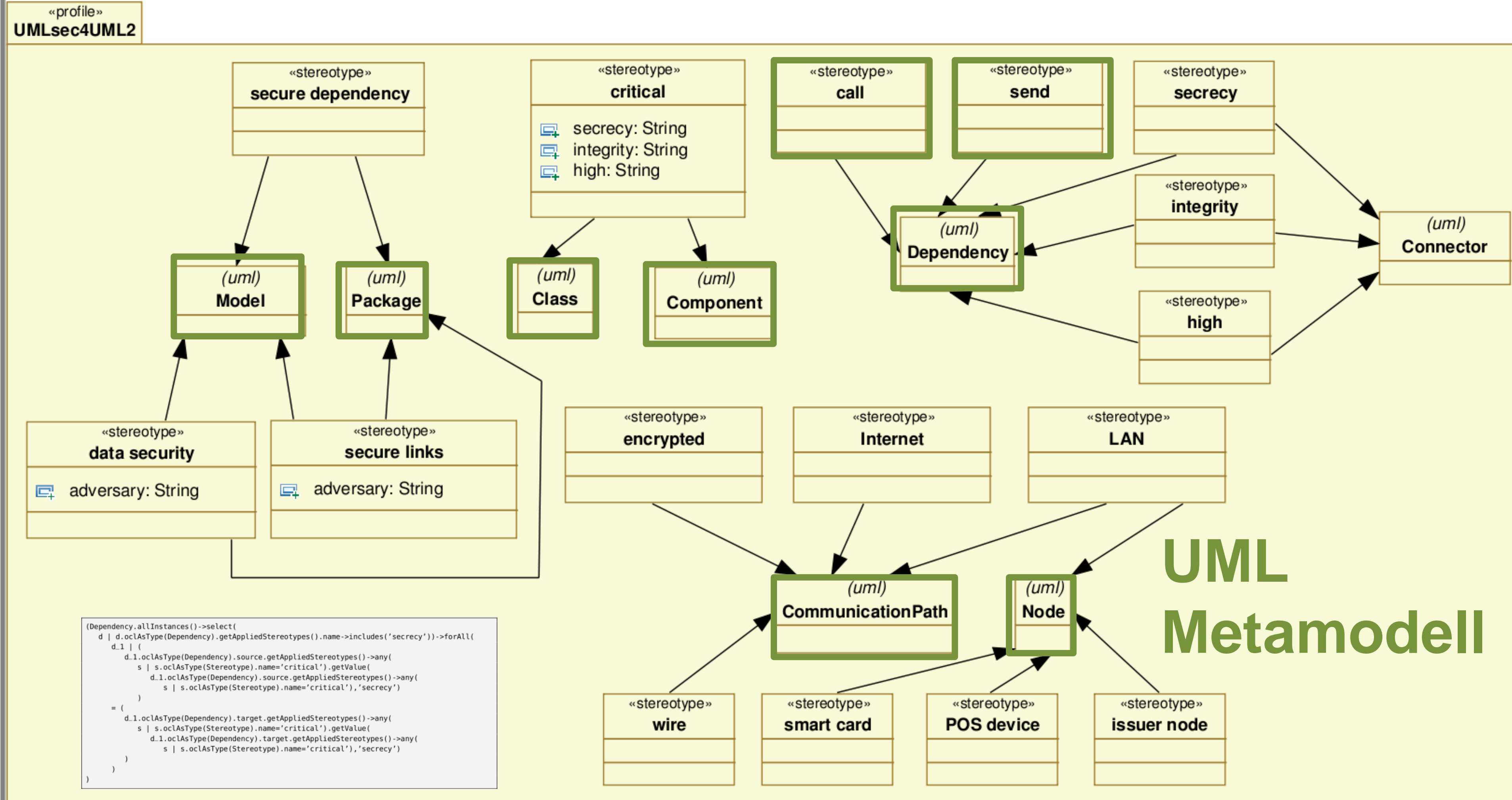
<<Table>> Auftragslager
<<Column>> <<PK>> <<FK>> auftrag_lfdNr : Ganz
<<Column>> <<PK>> <<FK>> auftrag_auftraggebe
<<Column> <<PK>> <<FK>> lager_ort_hausNr : Ga
<<Column> <<PK>> <<FK>> lager_ort_strasse : Ze
<<Column>> <<PK>> <<FK>> lager_ort_ort : Zeiche
<<Column> <<PK>> <<FK>> lager_ort_plz : Zeiche



Beispielprofil UMLsec



Beispielprofil UMLsec: UML-Metamodell




```
(Dependency.allInstances()->select(
   d | d.oclAsType(Dependency).getAppliedStereotypes().name->includes('secrecy'))->forAll(
      d_1 | (
         d_1.oclAsType(Dependency).source.getAppliedStereotypes()->any(
            s | s.oclAsType(Stereotype).name='critical').getValue(
               d_1.oclAsType(Dependency).source.getAppliedStereotypes()->any(
                  s | s.oclAsType(Stereotype).name='critical'),'secrecy')
            )
      = (
         d_1.oclAsType(Dependency).target.getAppliedStereotypes()->any(
            s | s.oclAsType(Stereotype).name='critical').getValue(
               d_1.oclAsType(Dependency).target.getAppliedStereotypes()->any(
                  s | s.oclAsType(Stereotype).name='critical'),'secrecy')
         )
      )
)
```

Beispielprofil UMLsec: UMLsec-Erweiterung

«profile»
UMLsec4UML2

«stereotype» **secure dependency**

«stereotype» **critical**
- secrecy: String
- integrity: String
- high: String

«stereotype» **call**

«stereotype» **send**

«stereotype» **secrecy**

«stereotype» **integrity**

«stereotype» **high**

(uml) **Dependency**

(uml) **Connector**

(uml) **Model**

(uml) **Package**

(uml) **Class**

(uml) **Component**

«stereotype» **data security**
- adversary: String

«stereotype» **secure links**
- adversary: String

«stereotype» **encrypted**

«stereotype» **Internet**

«stereotype» **LAN**

(uml) **CommunicationPath**

(uml) **Node**

UMLsec Profil

```
(Dependency.allInstances()->select(
   d | d.oclAsType(Dependency).getAppliedStereotypes().name->includes('secrecy'))->forAll(
      d_1 | (
         d_1.oclAsType(Dependency).source.getAppliedStereotypes()->any(
            s | s.oclAsType(Stereotype).name='critical').getValue(
               d_1.oclAsType(Dependency).source.getAppliedStereotypes()->any(
                  s | s.oclAsType(Stereotype).name='critical'),'secrecy')
            )
      = (
         d_1.oclAsType(Dependency).target.getAppliedStereotypes()->any(
            s | s.oclAsType(Stereotype).name='critical').getValue(
               d_1.oclAsType(Dependency).target.getAppliedStereotypes()->any(
                  s | s.oclAsType(Stereotype).name='critical'),'secrecy')
         )
      )
)
```

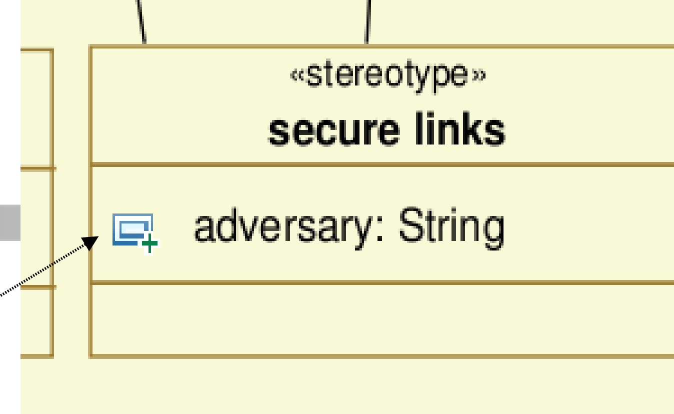
«stereotype» **wire**

«stereotype» **smart card**

«stereotype» **POS device**

«stereotype» **issuer node**

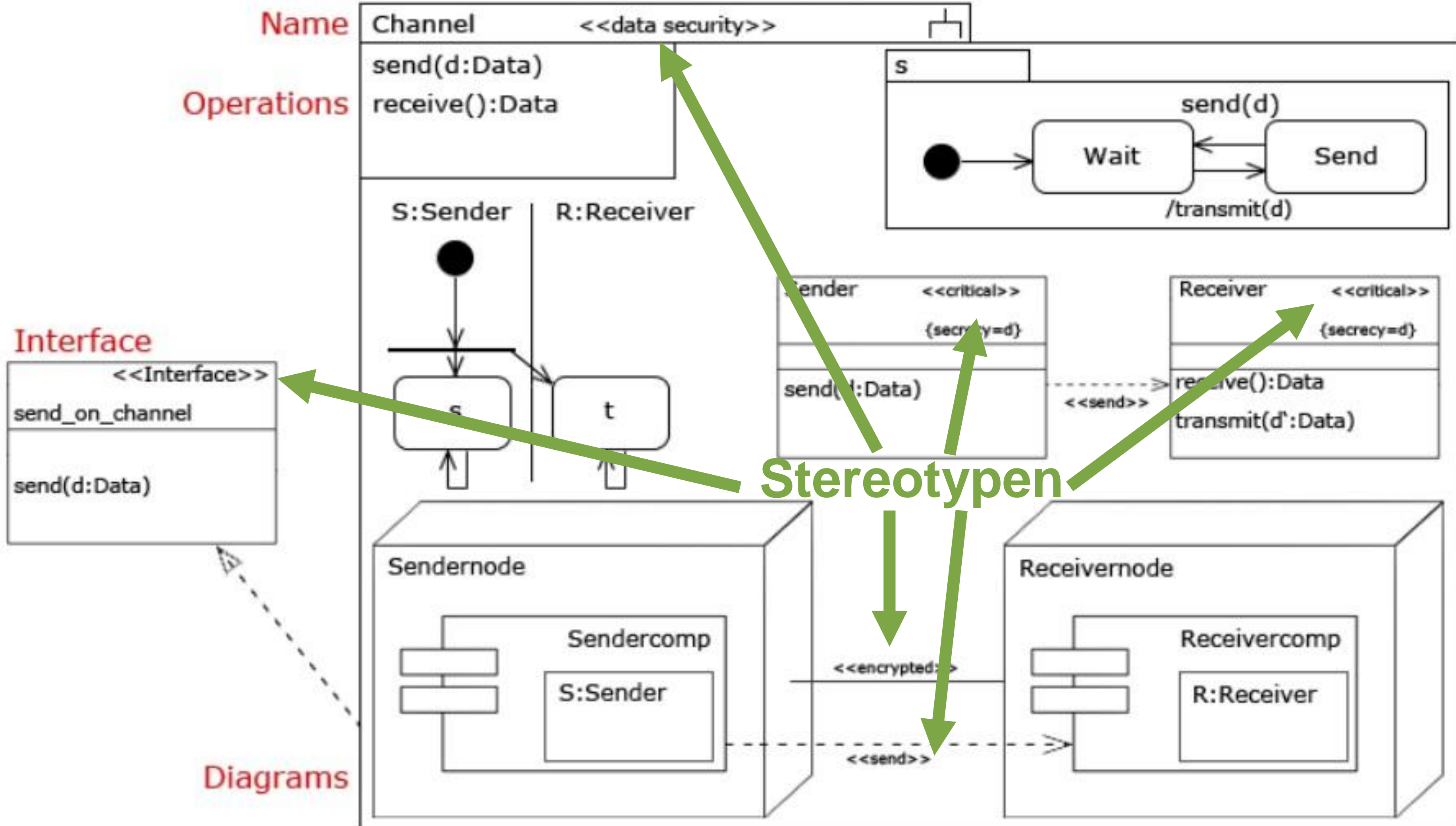
Beispiel-Stereotype <<secure links>>: OCL constraint



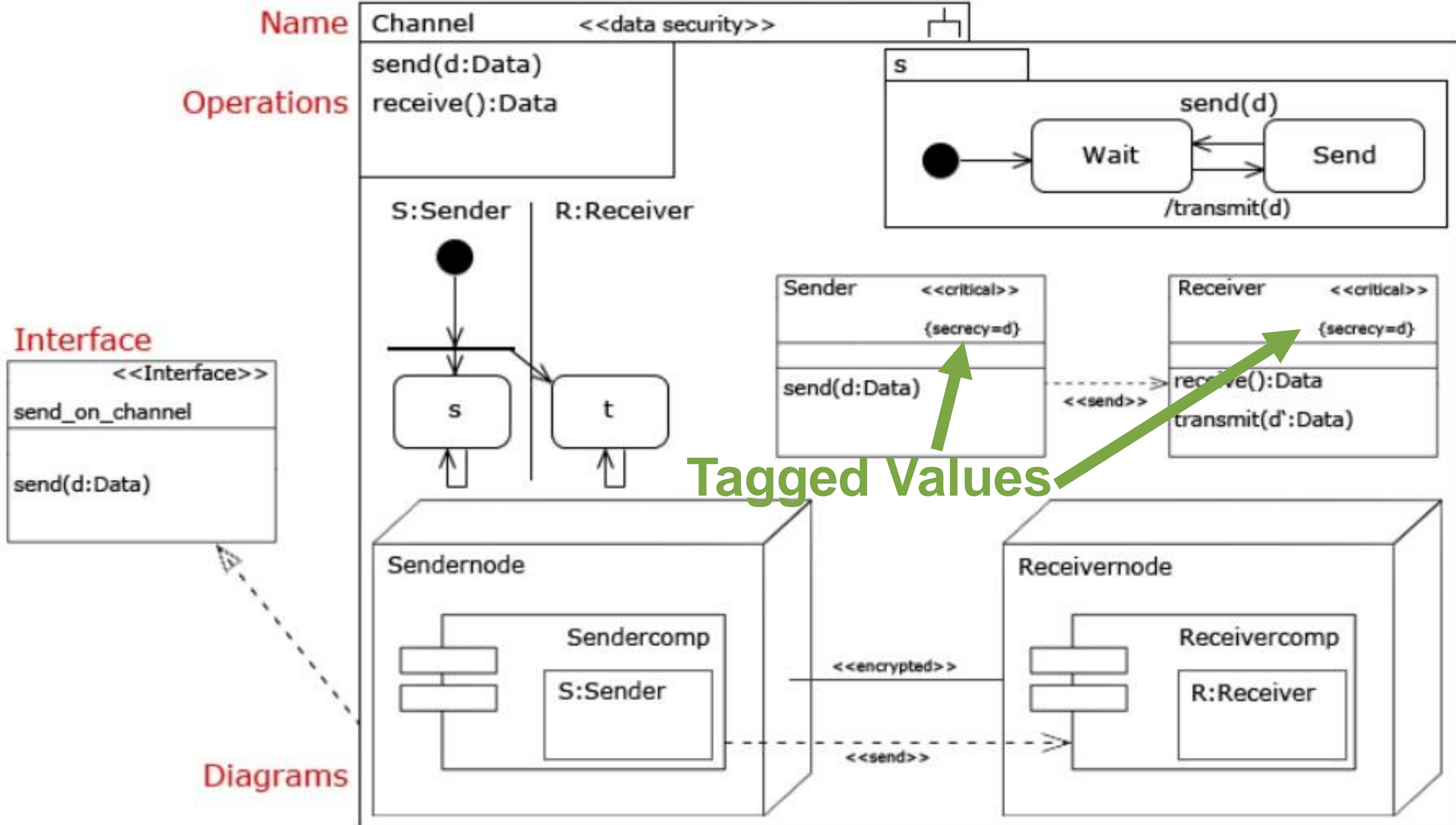
(Mehr Einzelheiten zu OCL in Abschnitt 1.2.)

```
(Dependency.allInstances()->select(
   d | d.oclAsType(Dependency).getAppliedStereotypes().name->includes('secrecy'))->forAll(
      d_1 | (
         d_1.oclAsType(Dependency).source.getAppliedStereotypes()->any(
            s | s.oclAsType(Stereotype).name='critical').getValue(
               d_1.oclAsType(Dependency).source.getAppliedStereotypes()->any(
                  s | s.oclAsType(Stereotype).name='critical'),'secrecy')
            )
      = (
         d_1.oclAsType(Dependency).target.getAppliedStereotypes()->any(
            s | s.oclAsType(Stereotype).name='critical').getValue(
               d_1.oclAsType(Dependency).target.getAppliedStereotypes()->any(
                  s | s.oclAsType(Stereotype).name='critical'),'secrecy')
         )
      )
)
```

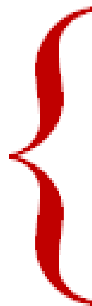
UMLsec Beispielmodell: Stereotypen



UMLsec Beispielmodell: Tagged Values



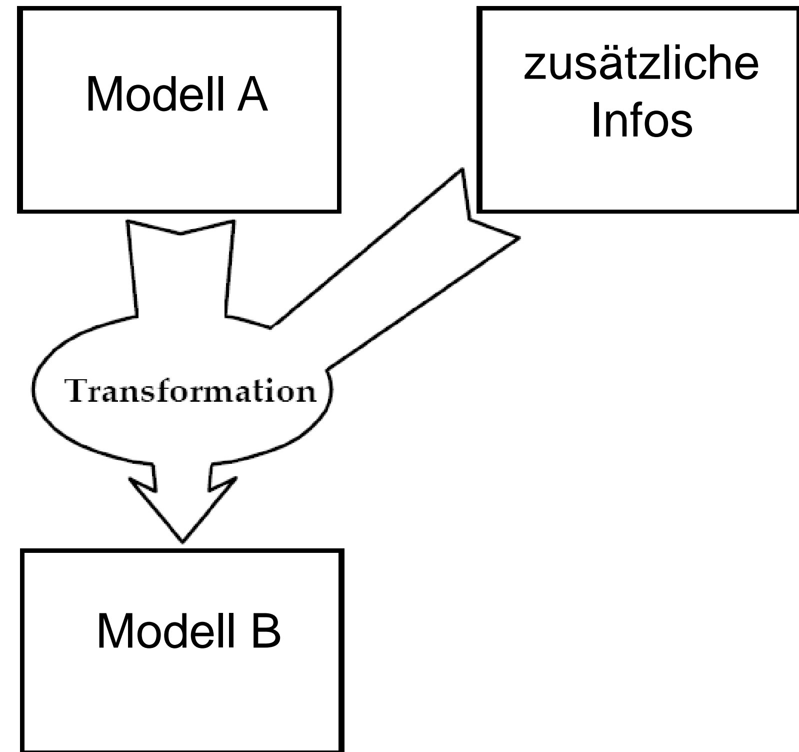
1.1 Modellbasierte Software-entwicklung



- Metamodellierung
- **Modelltransformation**
- Design Pattern

Modelltransformation

- **Berechenbare Abbildung** von Quellmodell in Zielmodell.
- Ziel: Modell A durch Anwendung einer Transformation und mittels und zusätzlicher Information **in Zielmodell B überführen.**
- Prinzipiell können **alle definier-baren Modelle** Quell- oder / und Zielmodell sein.
- **Viel Know-How im Generator.**



Modelltransformation

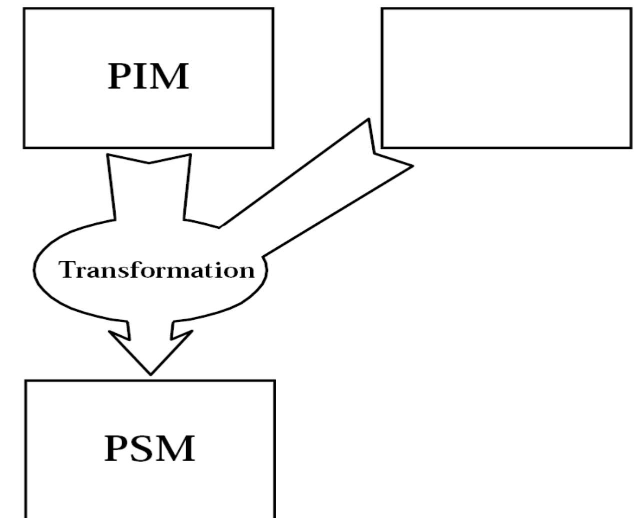
Horizontale und Vertikale Transformation

Horizontale Transformation:

- **Inhaltliche Weiterentwicklung** eines Modells durch Transformation.

Vertikale Transformation:

- Umwandlung eines Modelles einer Abstraktionsebene in Modell anderer Abstraktionsebene.
- z.B. Transformation eines Modells auf **technologiespezifischere Ebene.**
- z.B. Transformation in PSM: PIM und weitere Informationen notwendig.
- Kernstück des MDA-Ansatzes.

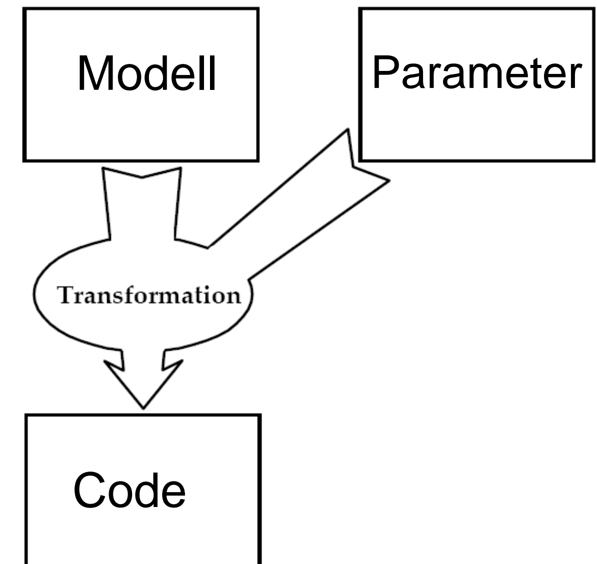


Modell-Modell Transformation:

- Überführung eines Modells in anderes Modell.
- Können auf verschiedenen Metamodellen basieren.

Modell-Code Transformation:

- Bezeichnet Überführung eines Modells in Quellcode.

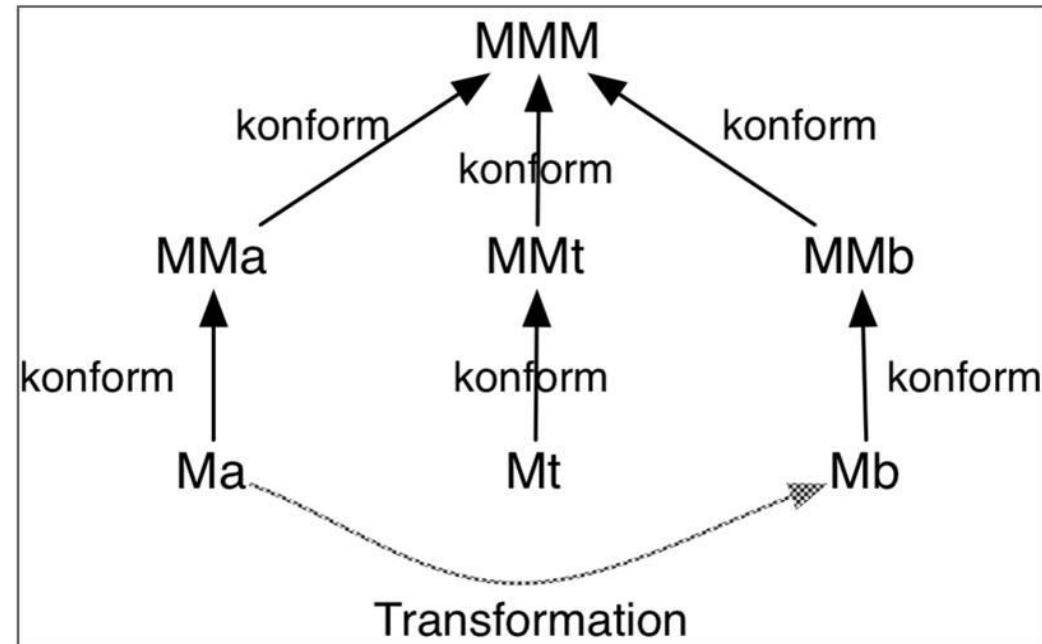


Unterscheidung von Modell-Code nicht trennscharf: Code kann als Art Modell aufgefasst werden.

Modelltransformation vs. Metamodelle

LEHRSTUHL 14 SOFTWARE ENGINEERING

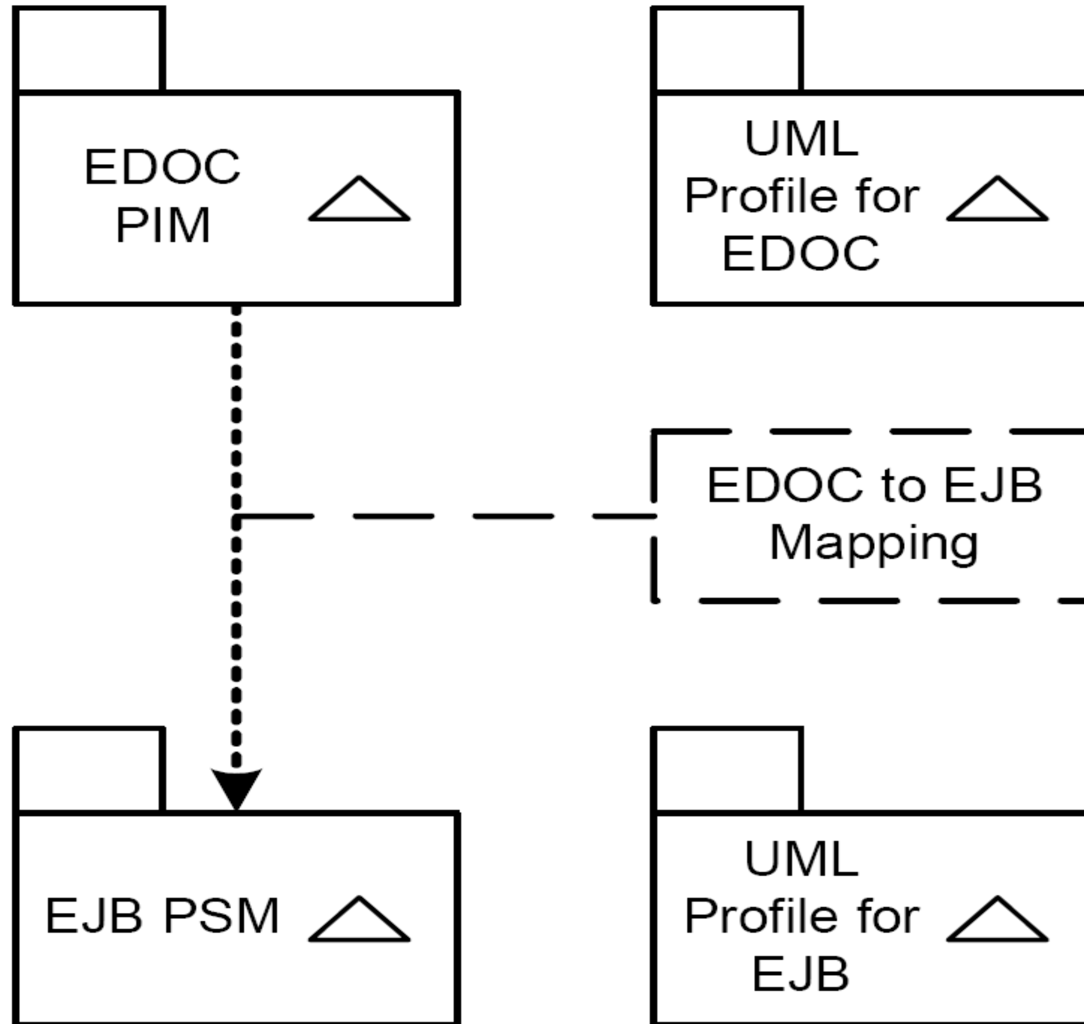
- **Dargestellte Modelle:** „Ma", „Mt" und „Mb": **Konform** zum jeweiligen Metamodell: „Mma", „MMt" und „Mmb.
- **Modelltransformation** von „Ma" nach „Mb": Auf Metamodellen definierte Abbildung.



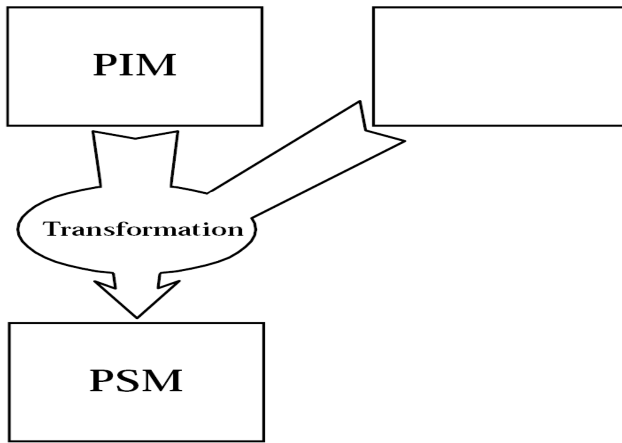
- Abbildung erlaubt, Instanzen von „MMa" in Instanzen von „MMb" zu transformieren.

Modelltransformation PIM-PSM vs. Metamodelle: Beispiel

LEHRSTUHL 14
SOFTWARE ENGINEERING

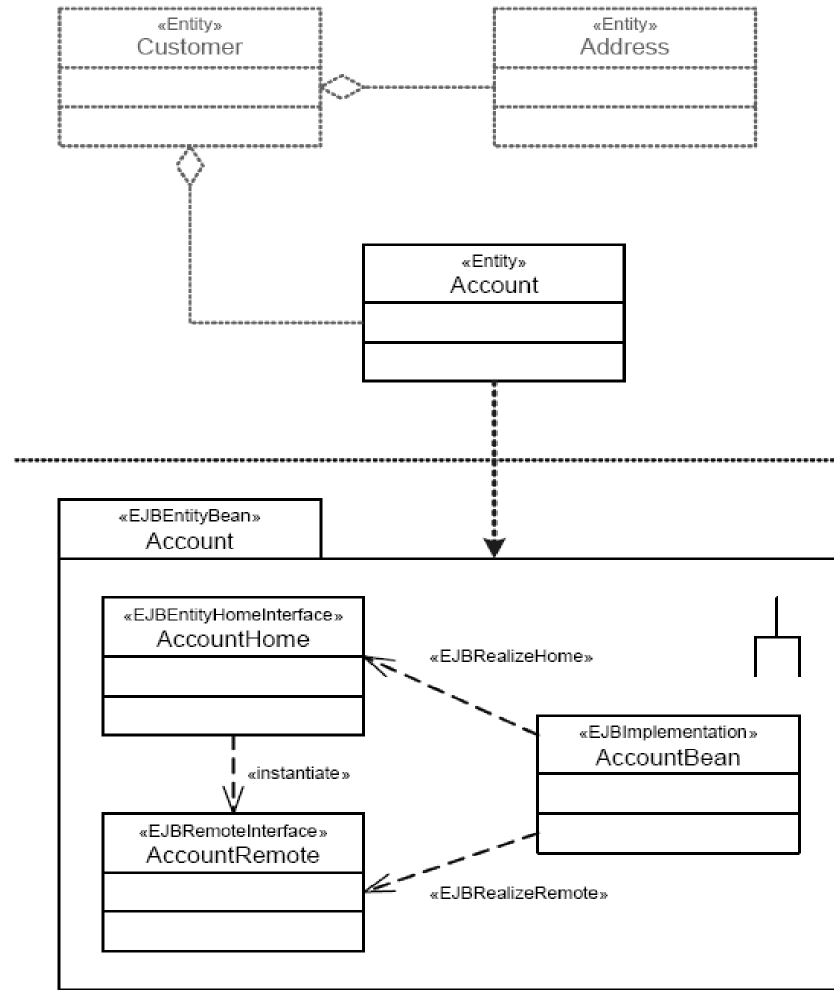


Transformation
Beispiel PIM → PSM



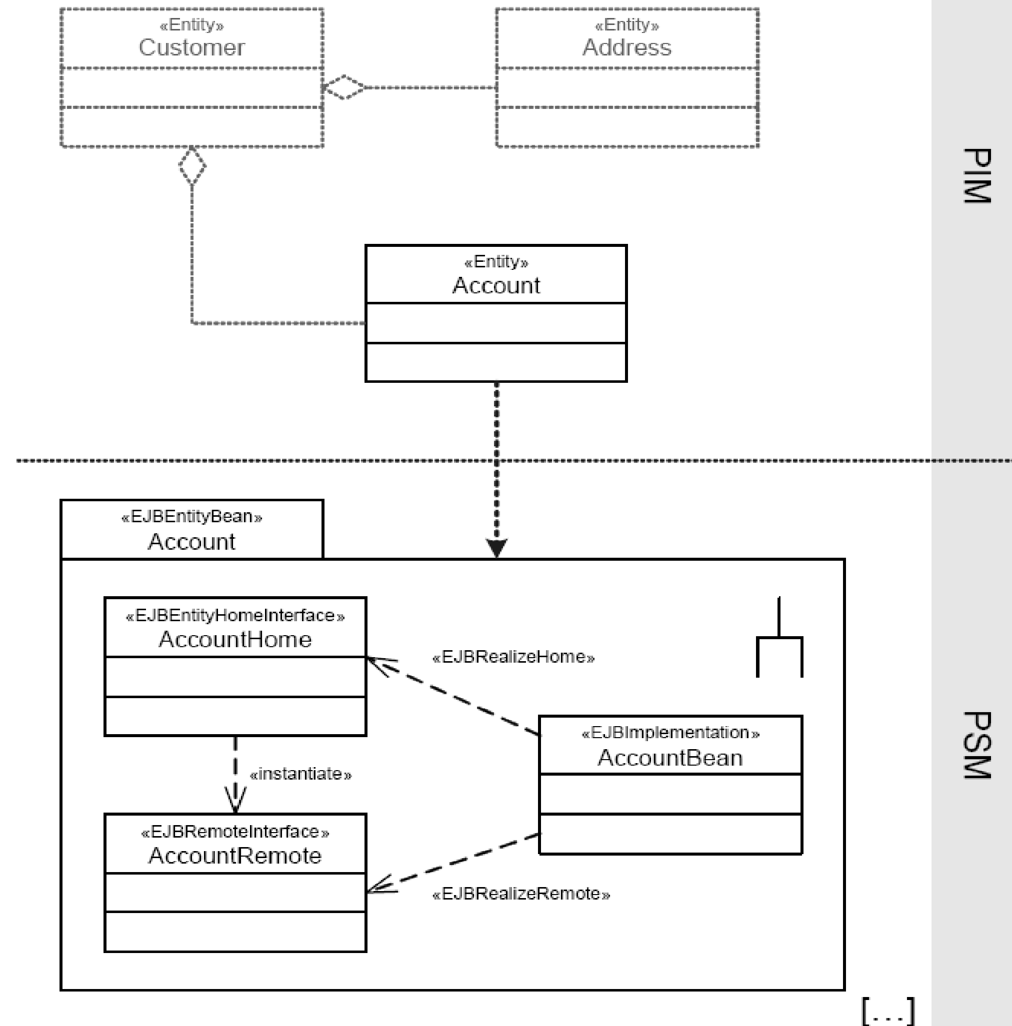
«Entity» der Klasse Account trägt **Semantik.**

➔ Führt zur Abbildung der Klasse in EntityBean des PSM.



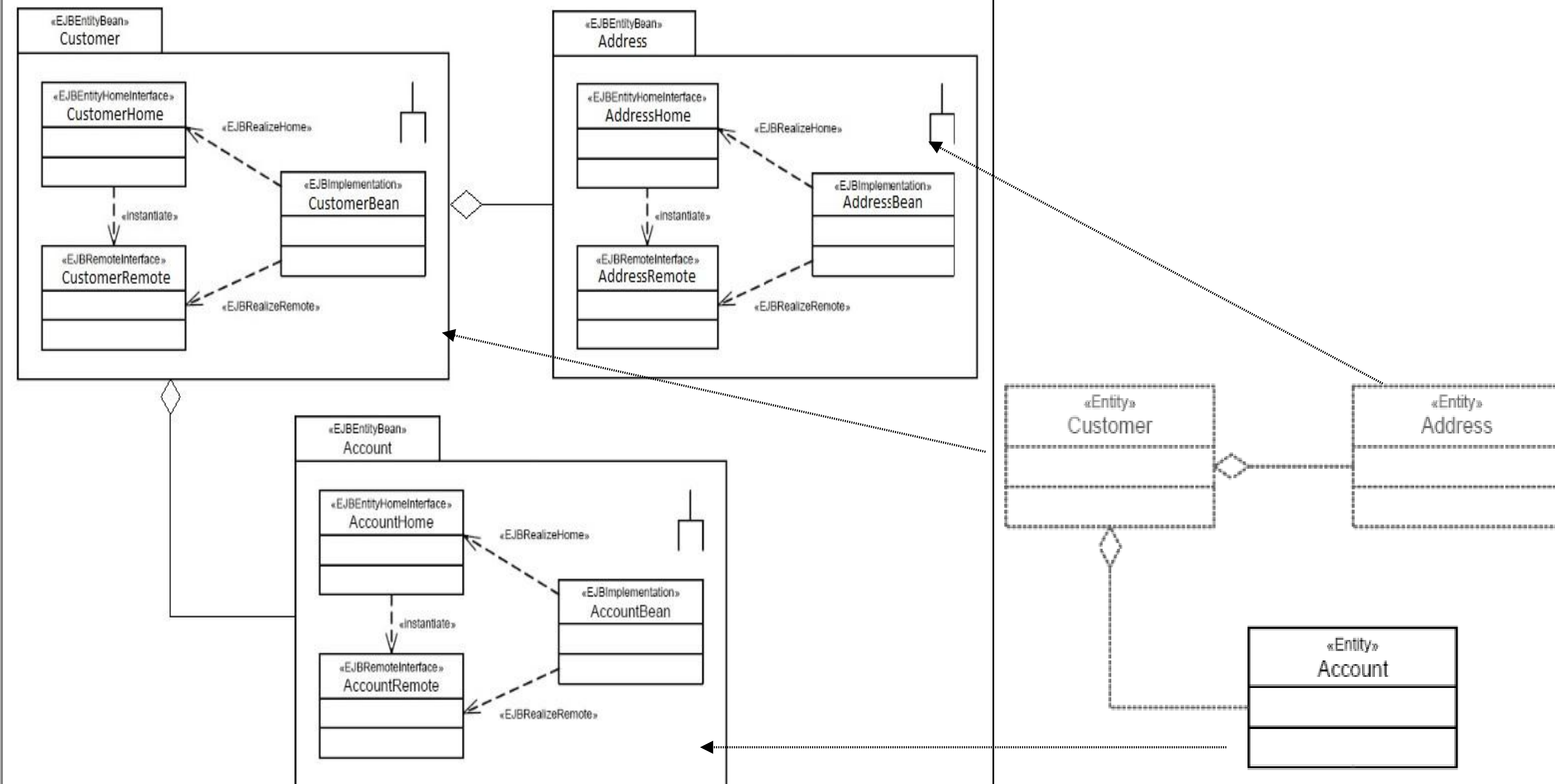
Diskussionsfrage
Beispiel PIM → PSM

Wie sieht das **PSM** vom gesamten PIM (inkl. **Customer** und **Address**) aus?



Diskussionsfrage
Beispiel PIM → PSM

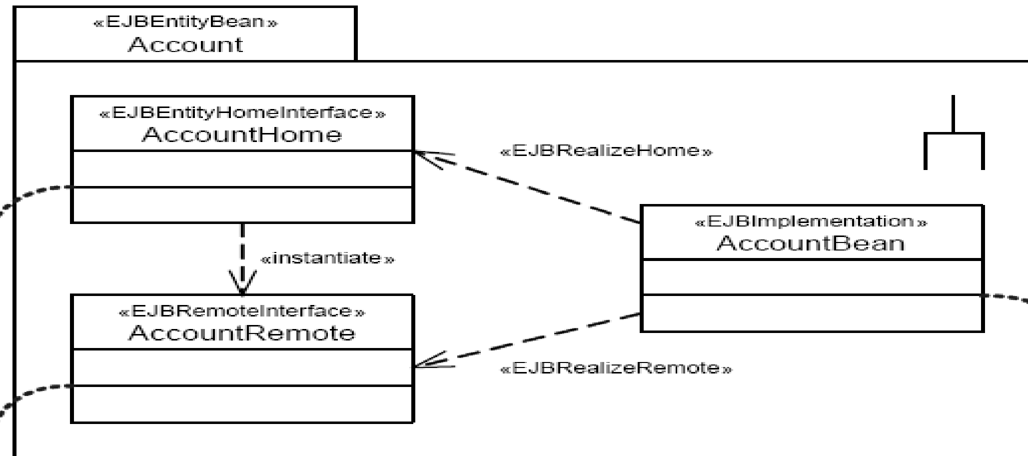
LEHRSTUHL 14
SOFTWARE ENGINEERING



Expandiert in **BeanClass** sowie **Remote-Interface** und HomeInterface:

- Wie Komponenten-modell verlangt.

[...]




```
AccountHome.java

import javax.ejb.EJBHome;
import javax.ejb.CreateException;
import javax.ejb.FinderException;

public interface Acco...
    extends EJBHome {

  AccountRemote creat...
      throws CreateEx...
  AccountRemote findB...
  [...]
}
```

```
AccountRemote.java

import javax.ejb....

public interface ...
    extends EJBOb...

  public String g...
  public void set...

  [...]
}
```

```
AccountBean.java

import javax.ejb.CreateException;
import javax.ejb.EntityBean;
import javax.ejb.EntityContext;

public abstract class AccountBean
    implements EntityBean {

  public abstract String getAccountNo();
  public abstract void setAccountNo(
       String accountNo);

  [...]
}
```

[...]

Modelltransformationssprachen:

- Mittel zur Beschreibung von Abbildungen (Transformationsregeln) von Instanzen eines Metamodells in anderes Metamodell.

Deklarative oder imperative Sprachkonstrukte.

Deklarative Transformationssprachen:

- Beschreibung der Transformationen durch Regeln.
  → Mit Vor- und Nachbedingungen spezifizierbar.
- Viele deklarative Transformationsansätze durch Graphentransformation realisierbar.

Imperative Transformationssprachen:

- Beschreibung der Transformation durch Sequenz von Aktionen.

QVT (Query View Transformation):

- Standard der OMG.

Besteht aus zwei Transformationssprachen:

- **QVT Relations:**
  - Deklaration.
  - Kann bidirektional und inkrementell transformieren.
- **QVT Operational Mapping:**
  - Imperativ.
  - Kann Relations verwenden.

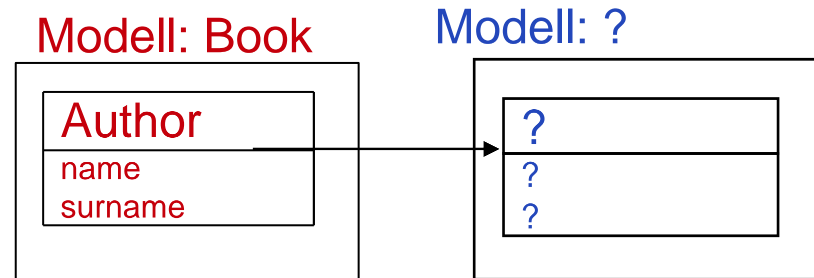
Metamodelle **mittels MOF beschreiben.**

Atlas Transformation Language (ATL):

- **Sprache des Eclipse-M2M-Projekts** (Modell-2-Modell):
  - Werkzeugunterstützung in Eclipse IDE (Debugger, Editor).
- Ursprung im INRIA (Franz. Forschungsinstitut).
- **Hybride Sprache** (deklarativ und imperativ).
- **Verwendet OCL** um Abfragen auf Modellen auszuführen.
  - Transformation besteht aus Satz von Regeln.
    → Überführen Elemente des Ausgangsmodells in Elemente des Zielmodells.

Beispiel ATL:
Resultat der Transformation von Book?

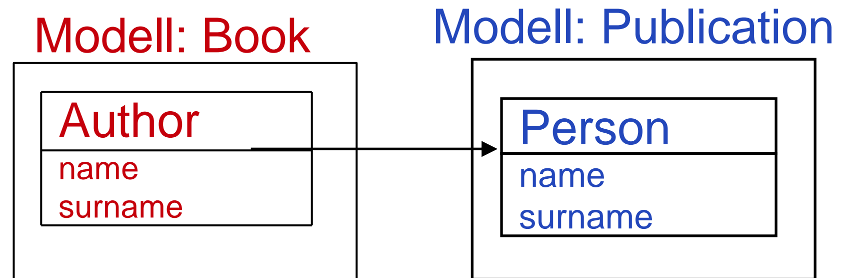
```
module Book2Publication;
   create OUT : Publication from IN : Book;
   rule Author {
    from
     a : MMAuthor!Author
    to
     p : MMPerson!Person (
      name <- a.name,
      surname <- a.surname
     )
   }
```



- Ausgangsmodell „IN“, konform zum Metamodell „Book“, in Modell „OUT“ (konform zu „Publication“) überführen.
- Regel überführt Elemente vom Typ „Author“ in Elemente vom Typ „Person“.

Beispiel ATL:
Resultat der Transformation von Book?

```
module Book2Publication;
   create OUT : Publication from IN : Book;
   rule Author {
    from
     a : MMAuthor!Author
    to
     p : MMPerson!Person (
      name <- a.name,
      surname <- a.surname
     )
   }
```



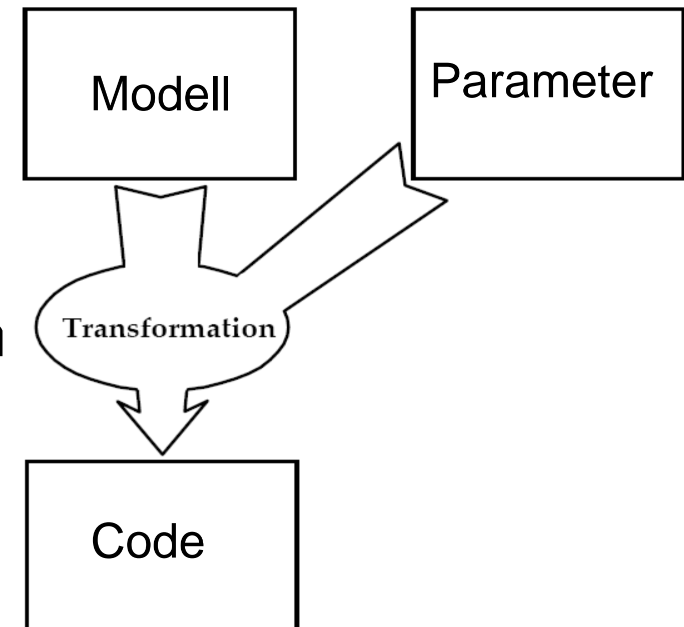
- Ausgangsmodell „IN“, konform zum Metamodell „Book“, in Modell „OUT“ (konform zu „Publication“) überführen.
- Regel überführt Elemente vom Typ „Author“ in Elemente vom Typ „Person“.

Codegenerierung
Ziel und Definition

Ziel: Generierung von Programmcode aus Modell.

Involviert Generator:

- Generator erzeugt **Programmcode** für spezifische Anwendungs- oder Programmklasse.
- Generator kapselt **generisches Programmmodell** (Klassen von Programmen).
- Konkret **erzeugter Code abhängig** von
  - Modell
  - Transformationslogik
  - Parameter



Grober Ablauf eines Generatorlaufs:

- Einlesen der Eingabespezifikation (bspw. UML, Modell, Text...).
- Einlesen der Parameter.
- Anwenden der **Transformationsregeln** auf Eingabespezifikation unter Berücksichtigung der Parameter.
- **Ausgabe von Programmcode.**

Phasen der Codegenerierung:

1. Programmierung eines Programmgenerators.
2. Parametrierung und Ergänzung eines Modells.
3. Parametrierter Aufruf des Generators und Erstellung des Programms.

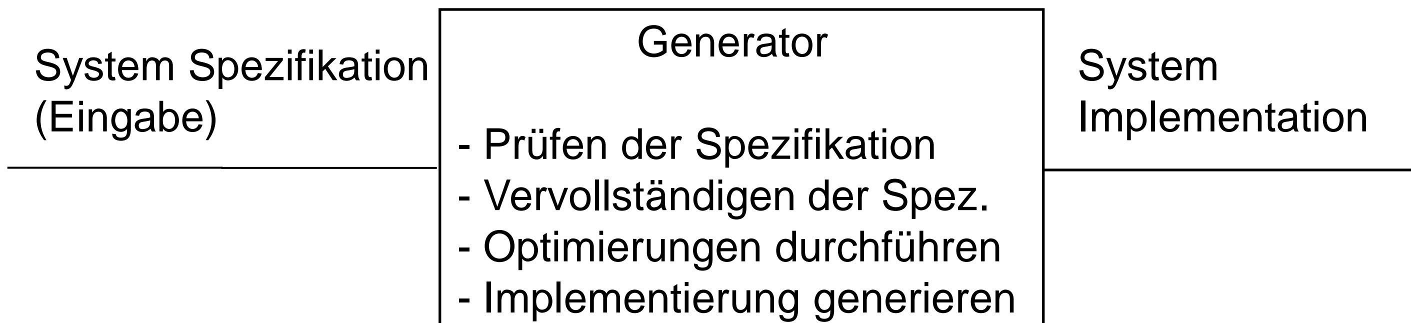
Ausgabe: Quellcode, Zwischencode, Binärcode.

Generatoren:

- Eingabe: abstrakte Beschreibung eines Software-Artefakts.
- Generieren dessen Implementation.

Software-Artefakt:

- Umfassendes Softwaresystem.
- Komponente.
- Klasse.
- Methode / Prozedur.



Codegenerierung
Vorteile

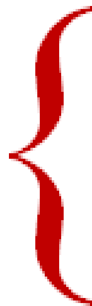
LEHRSTUHL 14
SOFTWARE ENGINEERING

- Codegenerierung arbeitet mit **variabilisiertem** Programmcode.
  → Programmcode mit Variationspunkten.
- Eindeutige Ausprägung durch Parametrierung des Programmcodes.
- Aufwand der Generatorentwicklung nicht trivial.
- **Eignung:** Für Lösungen mit entsprechend großer Zahl von Variationen in Praxis

Vorteil des Einsatzes:

- Gleichbleibende Qualität über alle Lösungen.
- Zentralisierter Wartungsaufwand.
- Erstellung mehrerer Lösungen in kurzer Zeit.

1.1 Modell-basierte Software-entwicklung

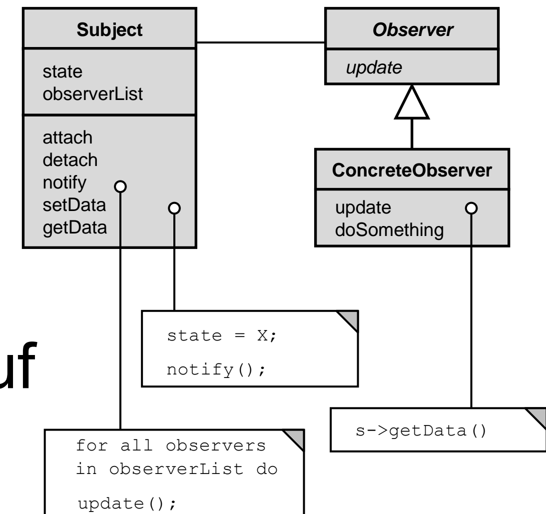


- Metamodellierung
- Modelltransformation
- **Design Pattern**

Motivation & Konzepte

OOD mehr als „Diagramme malen“ !

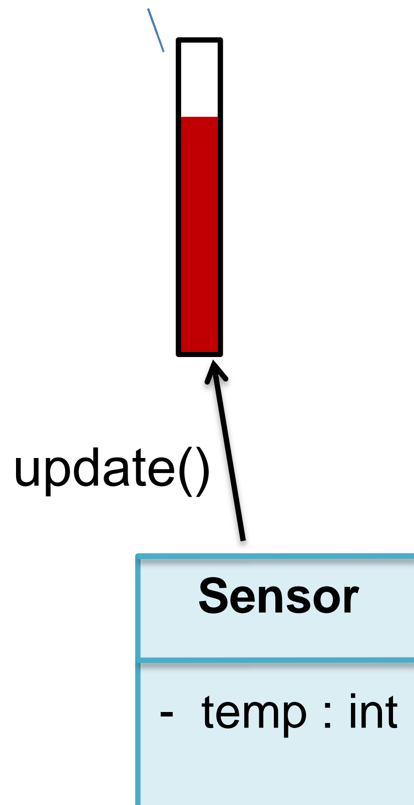
- Guter Designer: viel **Erfahrung**.
- Design-Probleme treten oft wiederholt auf
  ➔ Lösungen wiederverwenden !



Ziel: **Wissen** über **Design**-Lösungen:
festhalten, vermitteln, verwenden, erhalten !

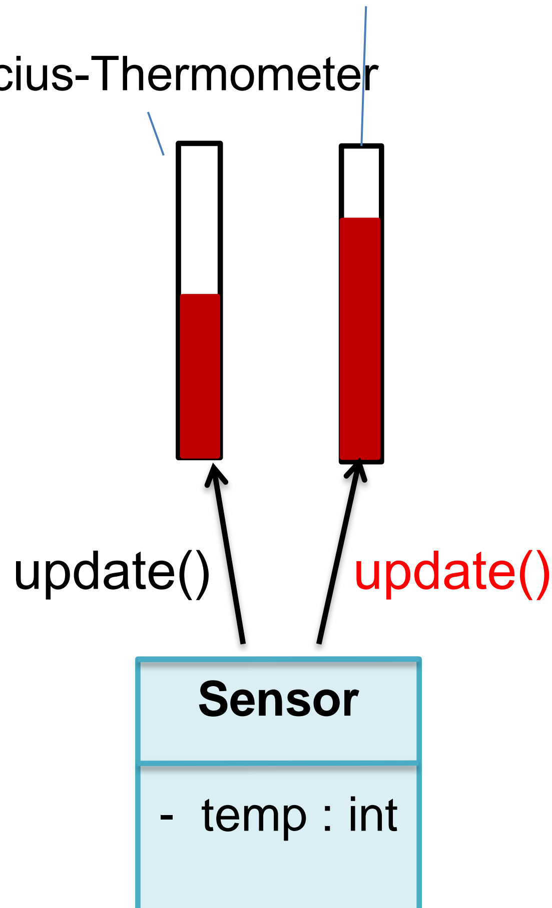
Daten mehrfach visualisieren

Thermometeranzeige **hinzufügen** - gutes Design ?




```
class Sensor {

    int  temp ;
    Thermometer  ct ;

    ...

    void change(int d) {
        temp = temp + d  ;
        ct.update(temp)  ;

    }
}
```

Daten mehrfach visualisieren



Ungünstig: Muss Sensor ändern !

➔ Wie geht es **besser ?**

```
class Sensor {

    int  temp ;
    Thermometer  ct ;
    Thermometer  ft ;
    ...

    void change(int d) {
        temp = temp + d  ;
        ct.update(temp)  ;
        ft.update(9*temp/5 + 32)
    }
}
```

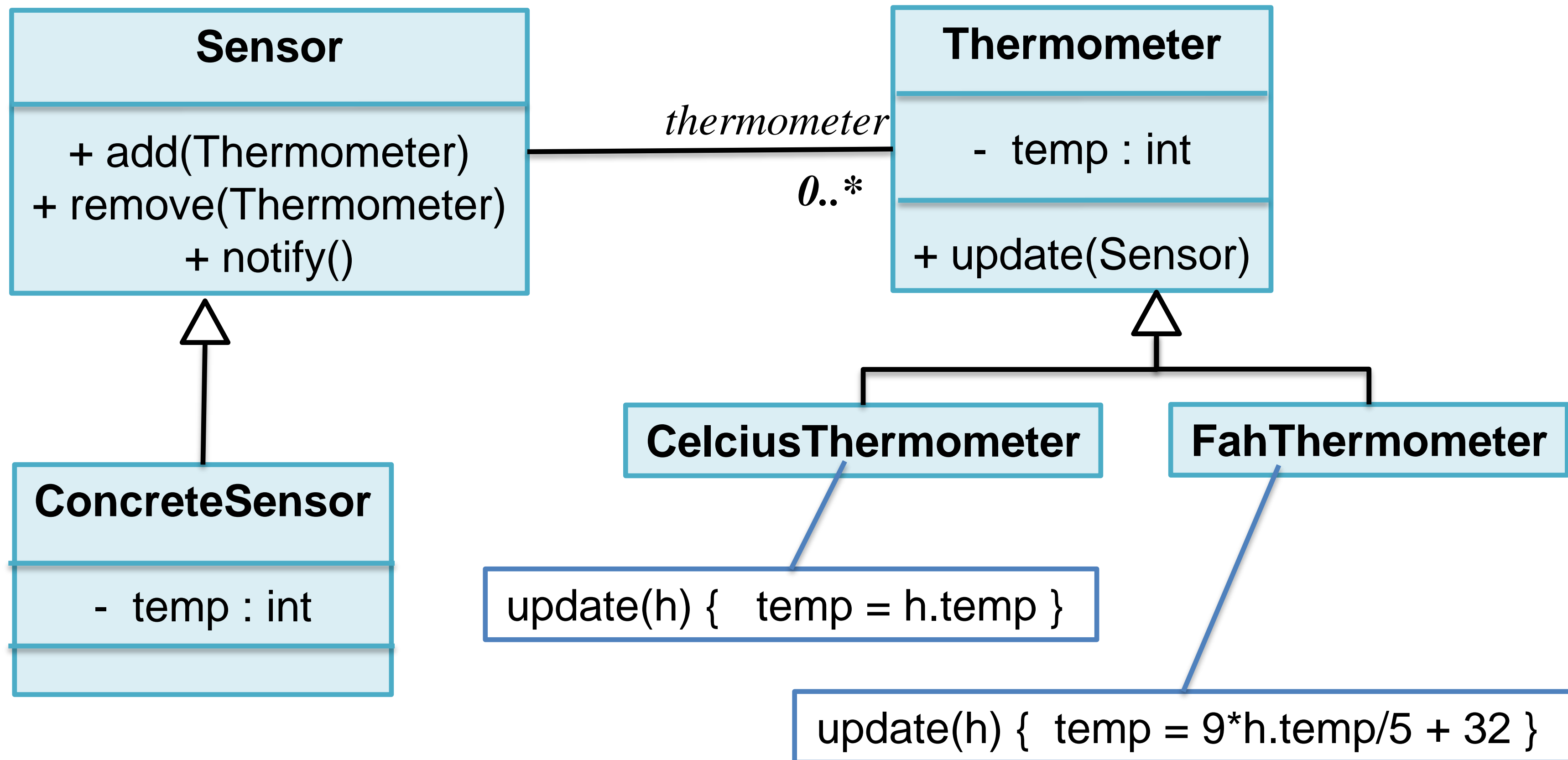
Daten mehrfach visualisieren: Lösung


```
class Sensor {
    private int  temp ;
    public List<Thermometer>  ts ;
    ...

    void change(int d) {
        temp = temp + d ;
        notify() ;
    }

    public void notify() {
        for (Thermometer t : ts)  t.update(this)
    }
}
```

Kann Thermometer hinzufügen / entfernen

Implementiert Temperatur-Umrechnung

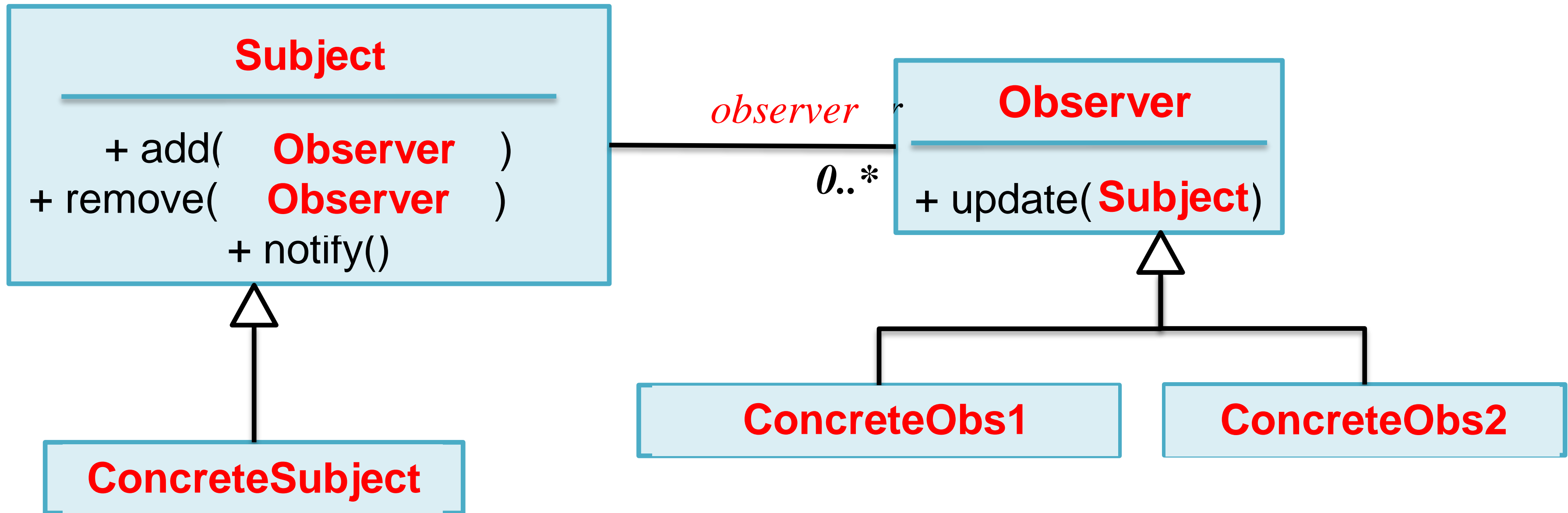
Daten mehrfach visualisieren: Designmodell



➔ Wie **verallgemeinern** zum **Wiederverwenden** ?

Wiederverwenden: Observer-Pattern

LEHRSTUHL 14 SOFTWARE ENGINEERING



Die vier Bestandteile eines Musters

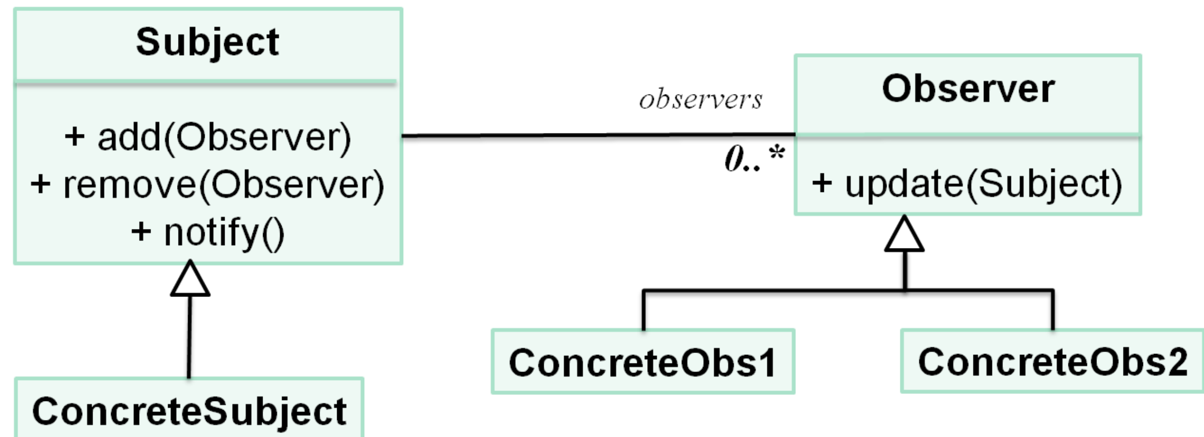
1. **Name**
2. **Problem**, inkl.:
   - Annahmen
   - Einflussfaktoren
3. **Lösung:**
   - bildliche und
   - sprachliche Beschreibung
4. **Anwendung** des Musters:
   - Konsequenzen
   - Trade-offs

2. Problem:

Zweck: 1-zu-n Beziehung zwischen Objekten (n dynamisch):
1-Objekt Zustandsänderung ➔ n-Objekte aktualisieren.

3. Lösung

- Struktur:



4. Observer - Anwendung

Konsequenzen

+ **Anpassbarkeit**: Beobachter: verschiedene Sichten auf Subjekt
+ **Modularität**: Subjekt / Beobachter: unabhängig änderbar
+ **Erweiterbarkeit**: beliebige Anzahl Beobachter hinzufügen
– **Update:** bei Kenntnis der Observer effizienter

Verwendungen

- **Smartphone event frameworks** (Symbian, Android, iPhone)
- **Publish / subscribe Middleware**
  (CORBA Notification Service, Java Message Service)
- **Mailing-Liste**
- **Java**: java.util.Observer, java.util.Observable

Überblick „Gang of Four“ Entwurfsmuster

Design Patterns
Elements of Reusable Object-Oriented Software
Erich Gamma
Richard Helm
Ralph Johnson
John Vlissides

Foreword by Grady Booch

		Zweck		
		Erzeugung	**Struktur**	**Verhalten**
Bereich	Klasse	Factory Method	Adapter (class)	Interpreter Template Method
	Objekt	Abstract Factory Builder Prototype Singleton	Adapter (object) Bridge Composite Decorator Flyweight Facade Proxy	Chain of Responsibility Command Iterator Mediator Memento Observer State Strategy Visitor

Vorteile von Mustern

Entwurfswissen festhalten:

- Design-Strukturen **explizit benennen**
- Design-**Entscheidungen** dokumentieren

Entwurf wiederverwenden:

- **Sprachen-** / **implementierungsunabhängig**
- Über **sprachspezifische** Einschränkungen hinaus
- Basis für **Automatisierung**

Empirische Studien (Tichy et al. 2001): Muster hilfreich.

Grenzen von Mustern

Kann Entwurfsoptionen einschränken.

Abstrahiert (bewusst) von Implementierung:

- **Implementierungsdetails** ungelöst
- **Manuelle Implementierung** mühsam, fehleranfällig

Lösung: nicht nur ***Design*** wiederverwenden.
➔ **Frameworks** !

Einführung Software Design Patterns:

- Ziele / grundlegende Elemente
- Erstes Beispiel: Observer Pattern
- Überblick "Gang of Four" Patterns
- Vorteile / Beschränkungen

Zusammenfassung und Ausblick

In diesem Abschnitt: Fortgeschrittene Konzepte zur „Modellbasierte Software-Entwicklung“. Wichtige Punkte:

- Metamodellierung
- UML-Erweiterungen
- Modelltransformationen
- Design Patterns

Im nächsten Kapitel: Object Constraint Language (OCL)

Anhang

Was Sie wiederholen sollten

Für diesen Abschnitt wiederholen:

- **UML-Klassendiagramme** (Generalisierung, Assoziationen, Komposition, Aggregation, Multiplizität,...)
- **UML-Aktivitätsdiagramme** (Aktivitäten, Aktionen, Verzweigungsknoten, Verbindungsknoten, Verteilungsknoten, Kontrollflüsse,...)

Hilfreich:

- Weitere **Struktur-** und **Verhaltensdiagramme** der UML (z.B. Objekt-, Sequenz-, Zustandsdiagramme)

Wichtig:

- Elemente dieser Diagramme genau kennen (→ relevant für zugehörige **Metamodelle**).

Metamodell:

- „meta“: „über“
- Modelle, die **Modelle beschreiben.**
- **Definition aller Elemente** der Modellierungssprache und ihrer Beziehungen untereinander.

Modell:	***Instanz:***
XML-Schema	XML-Datei
Grammatik	Programmiersprache: Java
Definition S/T-Netz	S/T-Netz Bestückungsroboter
UML-Klasse	Objekt
Metamodell	Modell

Metamodellierung

Metamodelle definieren **Modellelemente:**

- „Sprachdefinition“

Metamodelle für:

- Maschinenlesbarkeit.
- Validierung.
- Speicherung von Modellen (Repositories).
- Datenaustausch/Interoperabilität.
- Definition von Transformationen.

Modelle: Instanzen ihrer Metamodelle.

UML Spezifikation
Definition der Modellierungssprache

Bestandteile:

- **Abstrakte Syntax:**
  - Modellelemente.
- **Statische Semantik:**
  - Wohlgeformtheit.

- **Konkrete Syntax:**
  - Notation, ggf. graphische Symbole.
- **Dynamische Semantik:**
  - Verwendung, Bedeutung.

Informelle Beschreibung

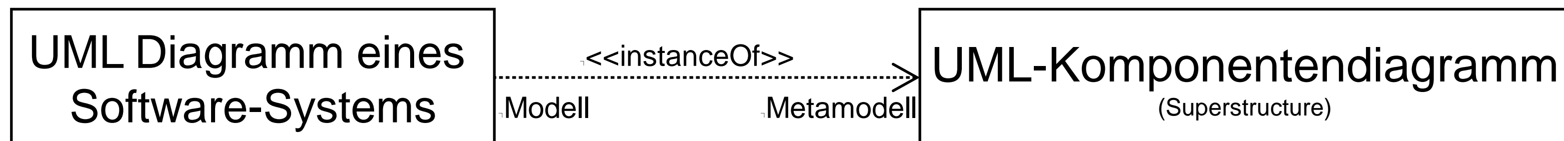
Model und Metamodell der UML

Modell:

- **Real existierendes System** als UML-Modell durch Anzahl von UML-Diagrammen beschrieben:
  - Klassendiagramm, Sequenzdiagramm, …

Metamodell:

- UML-Diagramme beinhalten **Notationselemente**:
  - Rechtecke, Pfeile, Balls, Sockets, Stereotypen, …
- **Definition** der UML-Diagramme als Metamodell:
  - UML Standard (UML 2.0 Superstructure).



Standards im Dunstkreis der MDA

Meta Object Facility (MOF):

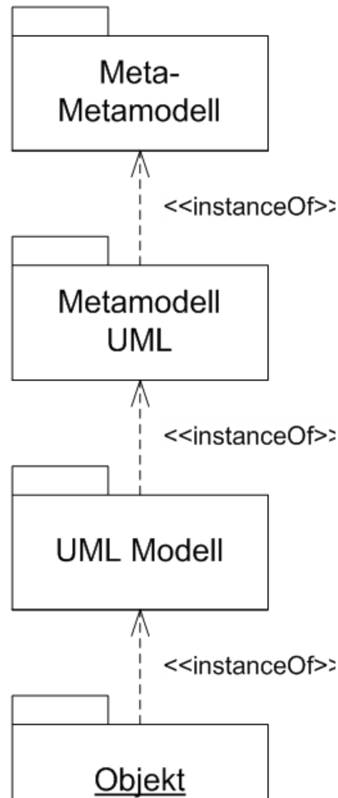
- Modellbasierte Sprache der OMG zur Definition von Metamodellen.
- Bsp.: Beschreibung der sämtlichen Standards des UML-2.x-Stacks.

Unified Modeling Language:

- Mittel Wahl zur Erstellung der Modelle innerhalb MDA.

XML Metadata Interchange (XMI):

- Definiert Abbildung der MOF auf XML.
- Ermöglicht standardisierten Austausch von beliebigen Meta-Modellen zwischen Tools.
  - z.B.: Transformatoren, Modellierungswerkzeugen, Codegeneratoren usw.
  - Grundvoraussetzungen zum Aufbau funktionierenden MDA-Infrastruktur.



UML Infrastructure: Definition der Elemente, mit der UML-Diagrammtypen spezifizierbar sind.
(UML-Meta-Metamodell, gegeben als Klassendiagramm (!))

UML Superstructure: Definition der UML-Diagrammtypen.
(UML-Metamodell, gegeben als Klassendiagramm (!))

Modell eines konkreten Systems.

Instanzen eines modellierten konkreten Systems.

Diskussionsfrage
Infrastructure /Superstructure: Ziele

Welche der beiden Ziele sind **Infrastructure** bzw. **Superstructure** zuzuordnen?

1. - Verbesserte architekturelle Angleichung zwischen **UML, MOF und XMI.**
   - Einheitliche, auf Nutzerebene verfügbare Erweiterungsmechanismen und Profile in einer zum **Metamodell** konsistenten Form.
2. - Direkte Unterstützung von Skalierbarkeit und Abkapselung für **Verhaltensmodellierung**.
   - Eindeutige **Definition der Semantik** von Relationen wie Generalisierung, Abhängig-keit und Assoziation.

Diskussionsfrage
Infrastructure /Superstructure: Ziele

Welche der beiden Ziele sind **Infrastructure** bzw. **Superstructure** zuzuordnen?

1. - Verbesserte architekturelle Angleichung zwischen **UML, MOF und XMI.**
   - Einheitliche, auf Nutzerebene verfügbare Erweiterungsmechanismen und Profile in einer zum **Metamodell** konsistenten Form.

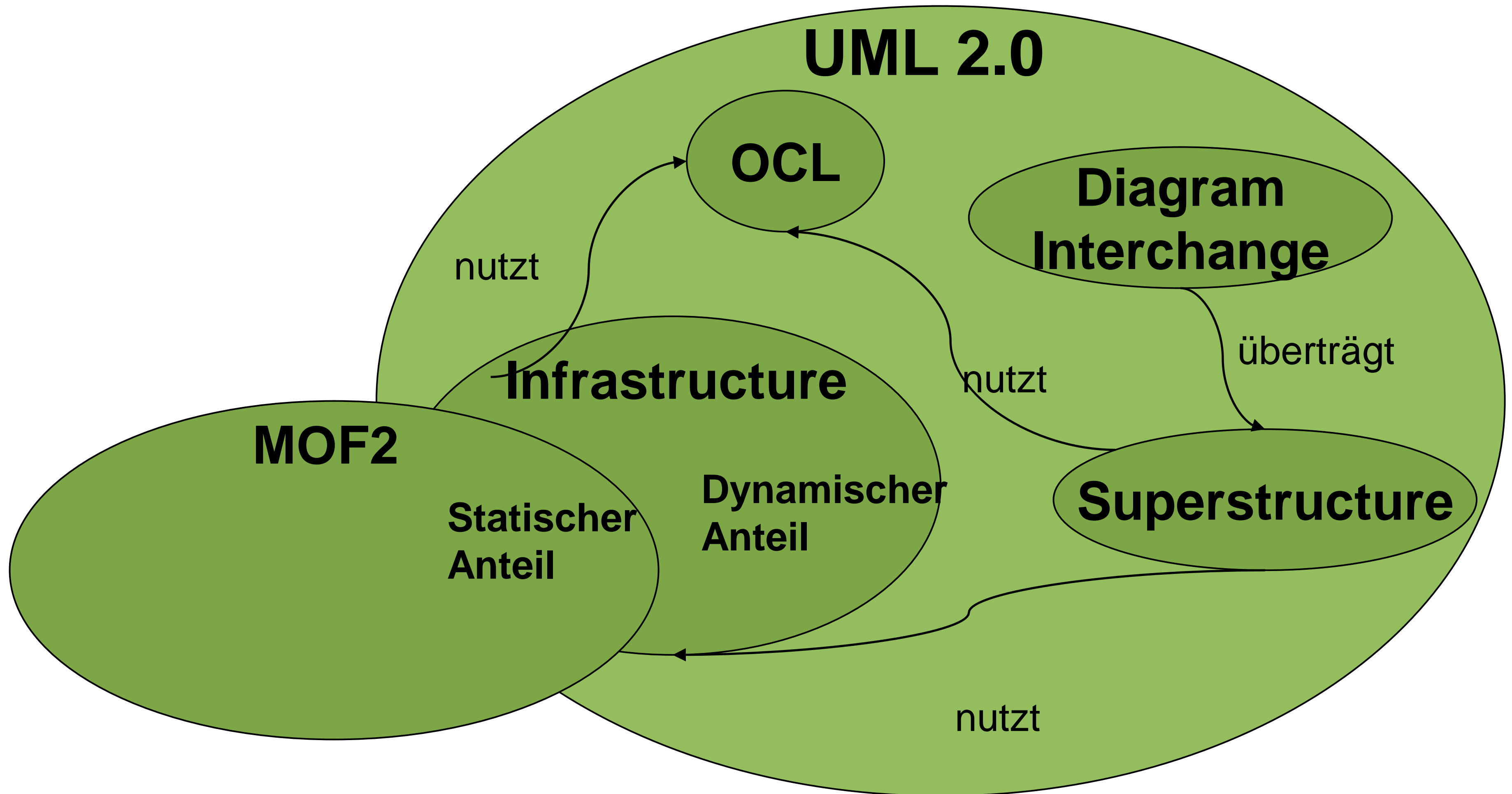
   Infrastructure

2. - Direkte Unterstützung von Skalierbarkeit und Abkapselung für **Verhaltensmodellierung**.
   - Eindeutige **Definition der Semantik** von Relationen wie Generalisierung, Abhängig-keit und Assoziation.

   Superstructure

LEHRSTUHL 14
SOFTWARE ENGINEERING

UML Spezifikation



Meta Ebenen: Beispiel

M3
MOF 2.0

Class

<<instanceOf>>
<<instanceOf>>
<<instanceOf>

M2
UML 2.0

Attribute
Class
classifier
Instance

<<instanceOf>>
<<instanceOf>>
<<instanceOf>>
<<instanceOf>>

M1
Benutzer-modell

Video
+title: String

<<snapshot>>

:aVideo
title="Casablanca"

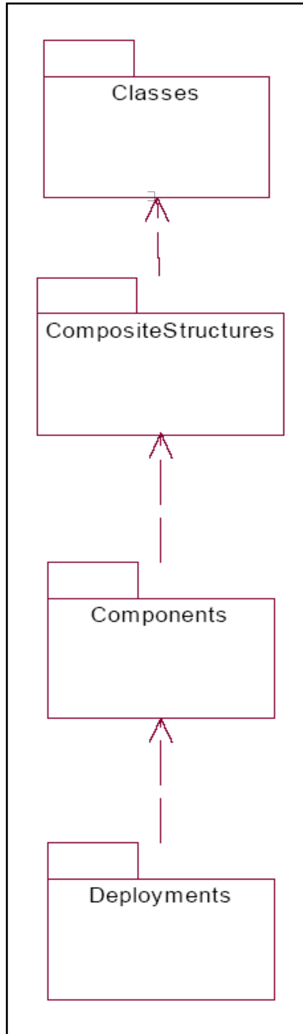
M0
Objekte der Realität

<<instanceOf>>

Aufbau des UML-Metamodells
Definitionen

- Definition der **Strukturdiagramme:**
  - Classes, Components, Composite Structures, Deployments.
- Definition der **Verhaltensdiagramme:**
  - Actions, Activities, Common Behaviors, Interactions, Use Cases, State Machines.
- Definition **zusätzlicher Konstrukte (Supplement):**
  - Hilfskonstrukte (Auxiliary Constructs):
    - Primitive Datentypen, Templates, Informationsflüsse.
  - UML-Profile.
- **Anhänge (Annexes):**
  - Reservierte Schlüsselwörter, Stereotypen, Profile, Tabellarische Darstellung von Diagrammen, Klassifikation von verwendeten Begriffen

Aufbau des UML-Metamodells
UML Strukturdiagramme



Aufgeteilt in verschiedene Pakete, die verschiedene Diagrammtypen repräsentieren:

= Klassendiagramm, Objektdiagramm, Paketdiagramm

= Kompositionsstrukturdiagramm

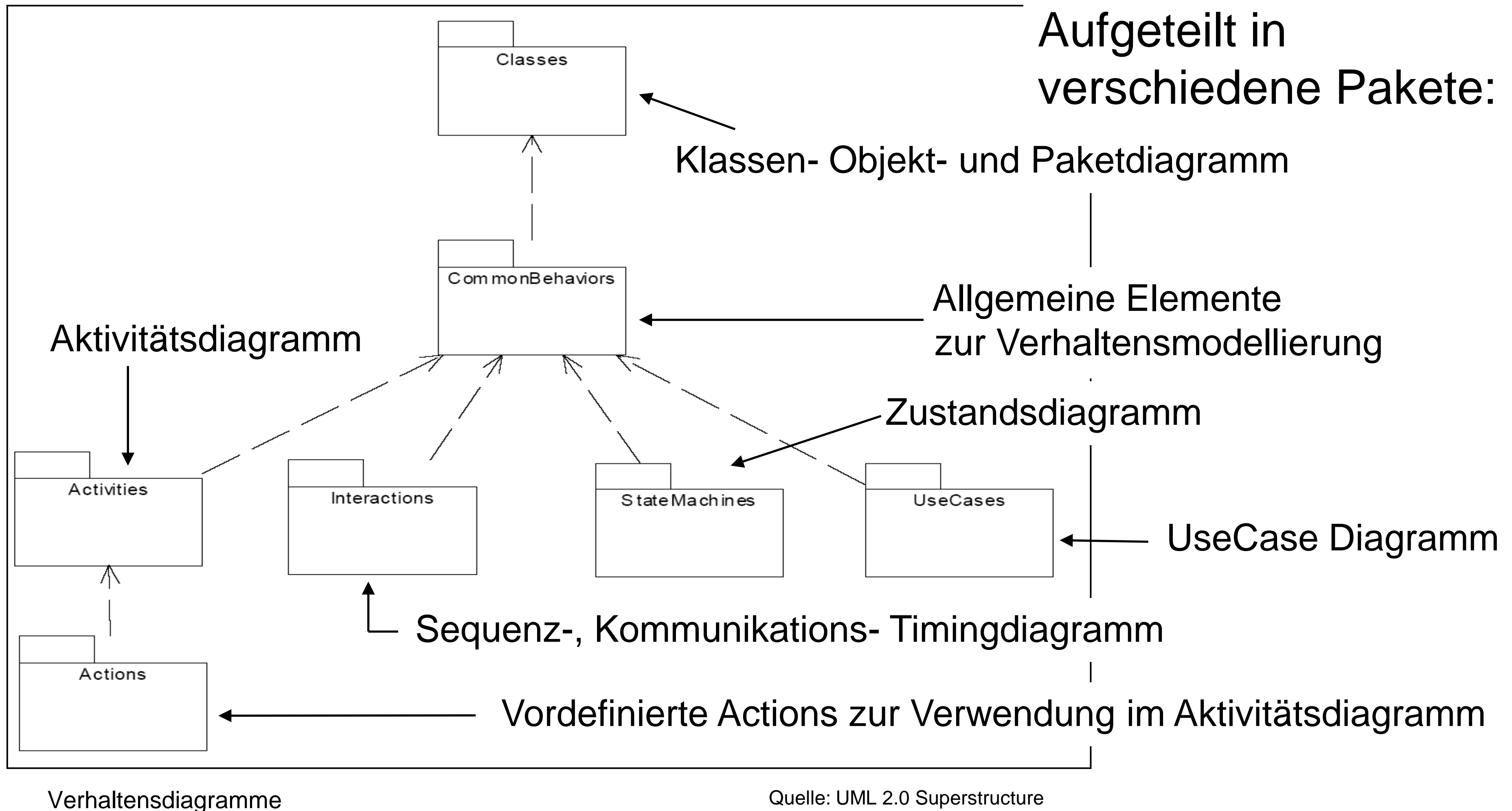
= Komponentendiagramm

= Deploymentdiagramm

Quelle: UML 2.0 Superstructure

Aufbau des UML-Metamodells
UML Verhaltensdiagramme

Aufgeteilt in verschiedene Pakete:



Verhaltensdiagramme

Quelle: UML 2.0 Superstructure

Definition der Semantik der Notationselemente der UML:

- Zu jedem Notationselement gibt es Text **„Semantics“.**

  → Definiert Verhalten des jeweiligen UML-Elementes.

- **Spielräume** zu einigen Elementen in Interpretation einräumen („Semantics Variation Points“).
- Siehe folgendes Beispiel der „Action“ im Aktivitätsdiagramm.
- Definition formaler Semantik: **Gegenstand vieler Forschungsprojekte** im UML-Umfeld.

Semantics

The sequencing of actions are controlled by control edges and object flow edges within activities, which carry control and object tokens respectively (see Activity). Alternatively, the sequencing of actions is controlled by structured nodes, or by a combination of structured nodes and edges. Except where noted, an action can only begin execution when all incoming control edges have tokens, and all input pins have object tokens. The action begins execution by taking tokens from its incoming control edges and input pins. When the execution of an action is complete, it offers tokens in its outgoing control edges and output pins, where they are accessible to other actions.

The steps of executing an action with control and data flow are as follows:

[1] An action execution is created when all its object flow and control flow prerequisites have been satisfied (implicit join). Exceptions to this are listed below. The flow prerequisite is satisfied when all of the input pins are offered tokens and accept them all at once, precluding them from being consumed by any other actions. This ensures that multiple action executions competing for tokens do not accept only some of the tokens they need to begin, causing deadlock as each execution waits for tokens that are already taken by others.

[2] An action execution consumes the input control and object tokens and removes them from the sources of control edges and from input pins. The action execution is now enabled and may begin execution. If multiple control tokens are available on a single edge, they are all consumed.

[3] An action continues executing until it has completed. Most actions operate only on their inputs. Some give access to a wider context, such as variables in the containing structured activity node, or the self object, which is the object owning the activity containing the executing action. The detailed semantic of execution an action and definition of completion depends on the particular subclass of action.

[4] When completed, an action execution offers object tokens on all its output pins and control tokens on all its outgoing control edges (implicit fork), and it terminates. Exceptions to this are listed below. The output tokens are now available to satisfy the control or object flow prerequisites for other action executions.

[5] After an action execution has terminated, its resources may be reclaimed by an implementation, but the details of resource management are not part of this specification and are properly part of an implementation profile.

Tefkat:

- Entwickelt an Universität von Queensland, Australien.
- **Open Source.**
- Deklarative Sprache.
- **Nicht bidirektional.**
- Werkzeugunterstützung in Eclipse IDE (Editor, Debugger).
- Metamodelle in Ecore (EMF) beschreiben.
- Beziehung zwischen Quell- und Zielmodellen über Regeln herstellen.
- Wiederverwendung von Code-Teilen über Definition von Pattern und Templates.
- **Sparsame Dokumentation.**