Vorlesung (WS 2013/14)

Softwarekonstruktion

Prof. Dr. Jan Jürjens

TU Dortmund, Fakultät Informatik, Lehrstuhl XIV

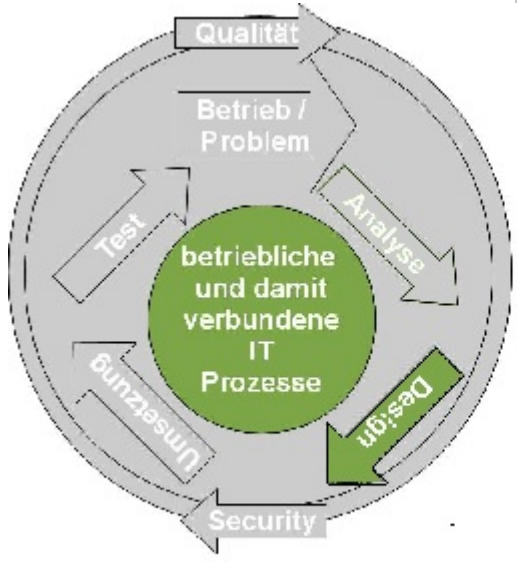
Teil 1.2: Modellbasierte Softwareentwicklung

v. 13.11.2013

Einordnung Modellbasierte Softwareentwicklung

LEHRSTUHL 14
SOFTWARE ENGINEERING

- Modellgetriebene SW-Entwicklung
  - Einführung
  - OCL
  - Modellbasierte Softwareentwicklung
  - EMF
- Qualitätsmanagement
- Testen
- Modellbasierte Sicherheit



Inkl. Beiträge von Prof. Summerville, St. Andrews University und Prof. Martin Glinz, Universität Zürich.

Literatur:

V. Gruhn: **MDA - Effektives Software-Engineering**. (s. Vorlesungswebseite)

- Kapitel 2-5

1.2 Modellbasierte Softwareentwicklung

1.2 Modell-basierte Software-entwicklung

- **MDA: Grundlagen und Konzepte**
- Metamodellierung
- Modelltransformation

Idee: Modellgetriebener Ansätze

- **Modelle zu Artefakten** erster Klasse:
  - Hinreichend zur Generierung eines Software-Systems.
- Gleich dem Übergang **von beschreibender zu vorschreibender Verwendung** von Modellen im SW-Entwicklungsprozess.
- **Nächsthöhere Abstraktionsstufe:**
  - Modellierungssprache bietet dichtere Notation für Teilmenge von Konzepten gegenüber klassischen Programmiersprachen.
    → Allein nicht ausreichend.
  - Geben Möglichkeit, domänenspezifische Sprachen selbst zu definieren.

- **Spezifikation der Softwarekomponente** unabhängig von technischer Umsetzung.
- **Verschiedene Abstraktionslevel** für Modelle:
  - **Plattform-unabhängig**, als konzeptionelle Basis des Ansatzes.
  - **Plattform-spezifisch.**
  - Dabei ist Plattform **relativ:** Spezifische Modelle der Ebene.
    → Unabhängige Vorlage für Modelle der nächsten Stufe und umgekehrt.
- Übergang von abstrakteren zu technologiespezifischeren Modellen.
  - **Vollautomatisiert** durch Verwendung von Transformationswerkzeugen.
  - **Beschreibung der Transformationen** in gesondert vorzuhaltenden Transformationsbeschreibungen.

Ziele der MDA
Konservierung der Fachlichkeit

- **Wertvolles Wissen über Kerngeschäftsprozesse** implizit in existierenden proprietären Altsystemen bzw. in Köpfen der Entwickler.
- **Plattform-unabhängige Modellierung** der unterliegenden Prozesse:
  - Pflege und Weiterentwicklung der Anwendungslogik.
  - Dokumentation und Evolution der Prozesse.
  - Integration dokumentierenden Modelle in Transformationskette hin zur Anwendung.

  → **Hilft gegen Degeneration von Dokumentation.**

Ziele der MDA
Portierbarkeit, Systemintegration & Interoperabilität

Portierbarkeit:

- **Systematische Abstraktion** von technischen Aspekten erleichtert:
  - Migration von Applikationen auf neue Versionen benutzter Technologien.
  - Portierung auf andere Zielumgebungen.
  - Anwendungsentwicklung für mehr als eine Zielplattform.

Systemintegration und Interoperabilität:

- **Trennung fachlicher Konzepte** von konkreter Repräsentation in Technologie existierender Systeme:
  - Erleichtert Bildung von Schnittstellen.
- **Verwendung offener Standards:**
  - Verringert Risiko von Vendor-Lock-Ins.
  - Hilft bestehende Anwendungen über Technologiezyklen zu retten.
- Erleichtert **Wiederverwendung** und steigert **Produktivität.**

Ziele der MDA
Effiziente Softwareentwicklung

Effiziente Softwareentwicklung:

Automatisierung der Anwendungserstellung:

- Beschleunigt Software-Prozesse.
- Verringert Aufwand.
- Steigert Software Qualität.
  - z.B. konsequente Verwendung von Referenzarchitekturen und Pattern-Replication.

Technologien werden gekapselt:

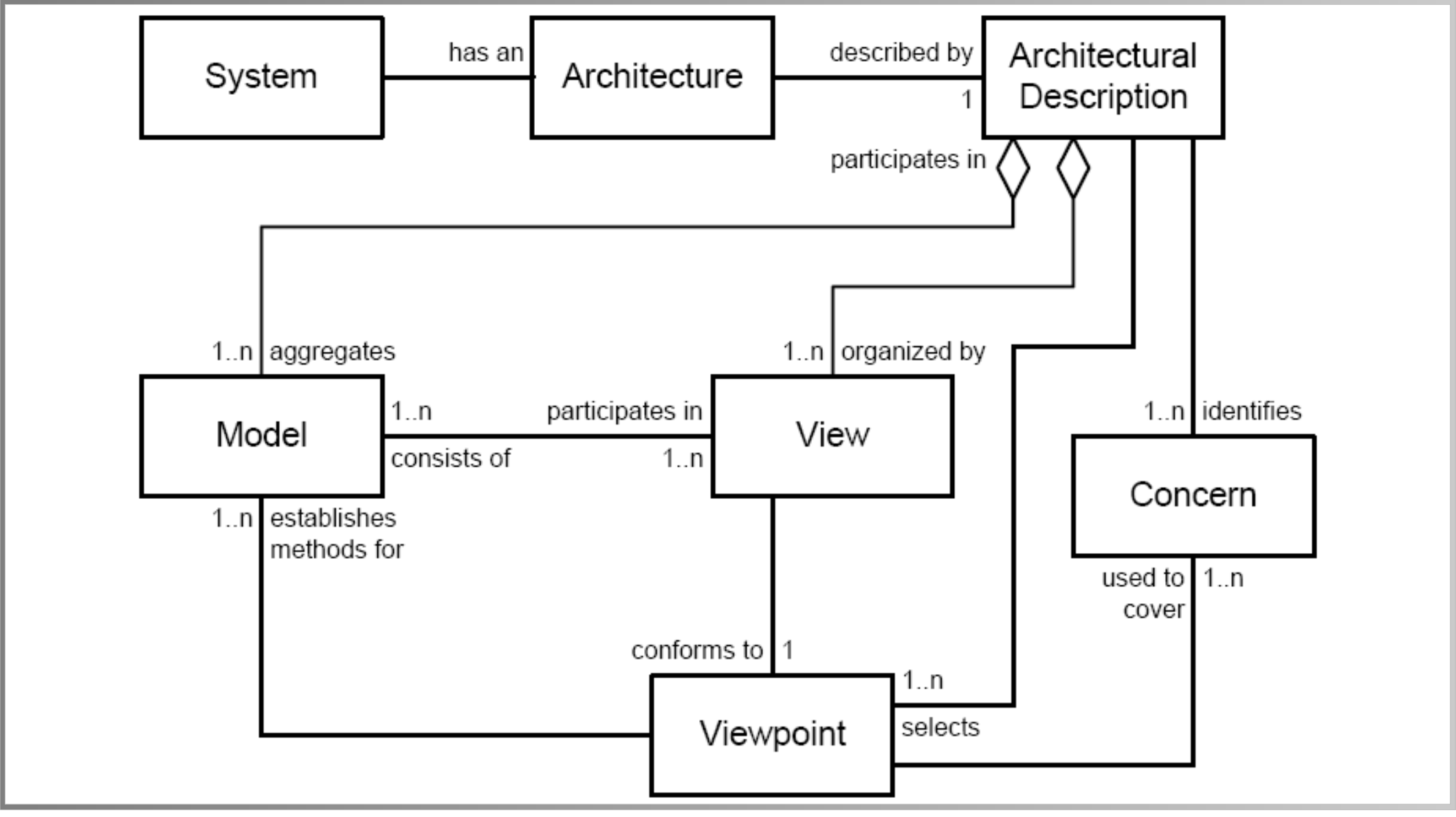
- Wissen von Experten in Form von **Transformationsvorschriften** multiplizierbar.
- Ermöglicht **optimale und gleichzeitige Nutzung** vorhandener Ressourcen.
- Beherrschung **systemimmanenter Komplexität.**
  - Prinzip: Teile-und-herrsche.

Ziele der MDA
Domänen-Orientierung

- **Domänenspezifische Modelle** und Wiederverwendung von Architekturmetamodelle:
  - Reduzierung der Time-to-Market ohne Budgets Erhöhung.
  - **Wissensvorsprung** innerhalb fachlichen Domänenwissens.
- **Fachlichkeit** innerhalb des Software-Entwicklungszyklus: Bedürfnis der Kunden:
  - **Hoher Flexibilität/Agilität** unterliegender Geschäftsprozesse.
  - **Ohne unnötige Beschränkung** durch technologische Vorgaben gerecht.

Terminologie IEEE1471

LEHRSTUHL 14
SOFTWARE ENGINEERING



Zentrale MDA-Begrifflichkeiten

Computation Independent Model (CIM):

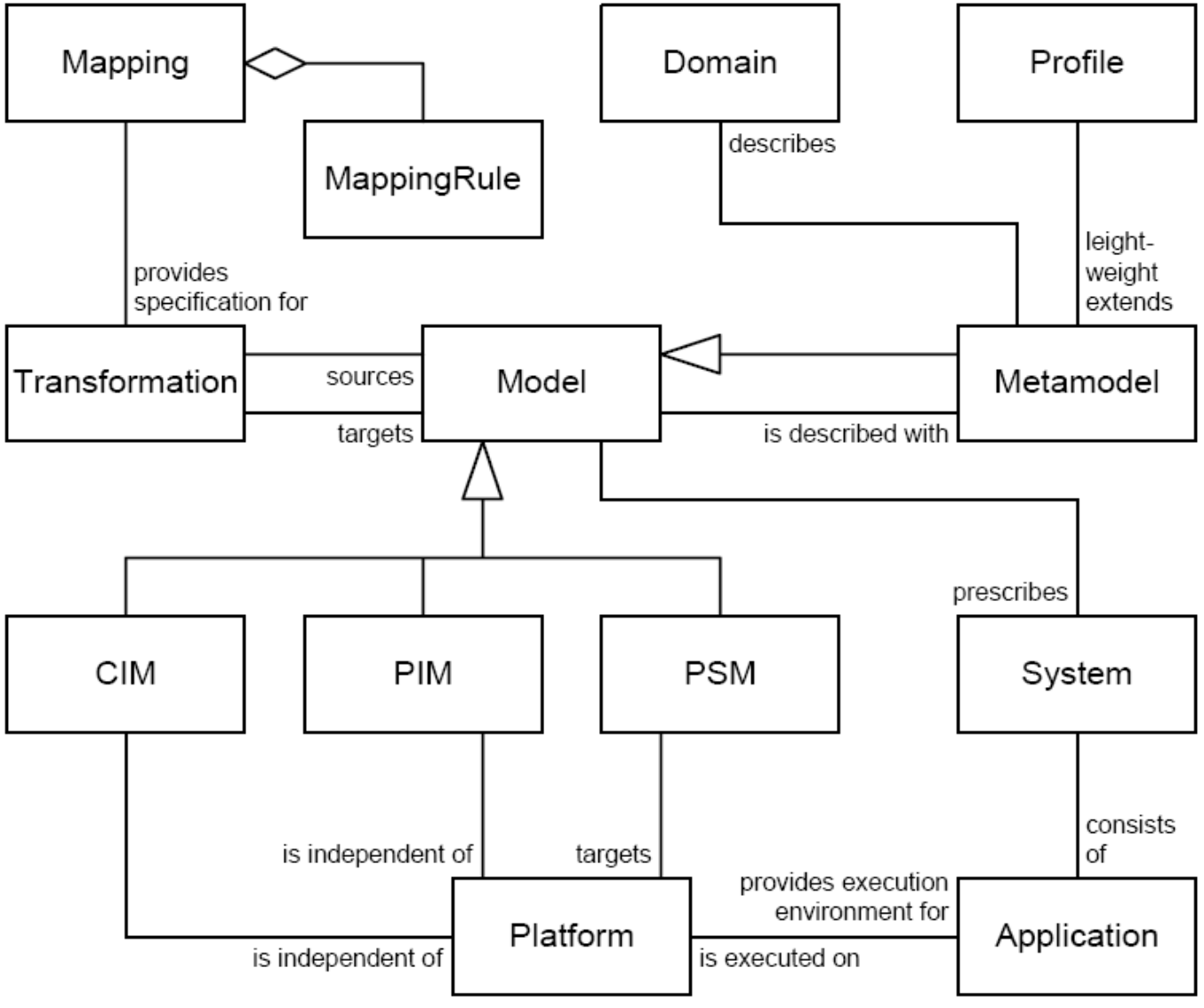
- **Anforderungen an System** und Umwelt.

Platform Independent Model (PIM):

- **Formale Struktur und Funktionalität** eines Systems.

Platform Specific Model (PSM):

- **PIM mit plattformabhängigen Informationen**.



Diskussionsfrage: MDA-Begrifflichkeiten

LEHRSTUHL 14
SOFTWARE ENGINEERING

- Welche **MDA-Begrifflichkeiten** sind zu folgenden Aussagen zuzuordnen?

Kann alle **Informationen umfassen**, die notwendig sind, um das System zu erzeugen und in Betrieb zu nehmen. — CIM

Beschreibt die **Nutzung des Systems**, seine Anforderungen an seine Umgebung, sowie den Nutzen, den die Umgebung aus diesem System zieht. — PIM

Spezifikation der Struktur und des Verhaltens des Systems unabhängig von den später zur Implementation einzusetzenden Hardware- & Softwareplattformen. — PSM

Diskussionsfrage: MDA-Begrifflichkeiten

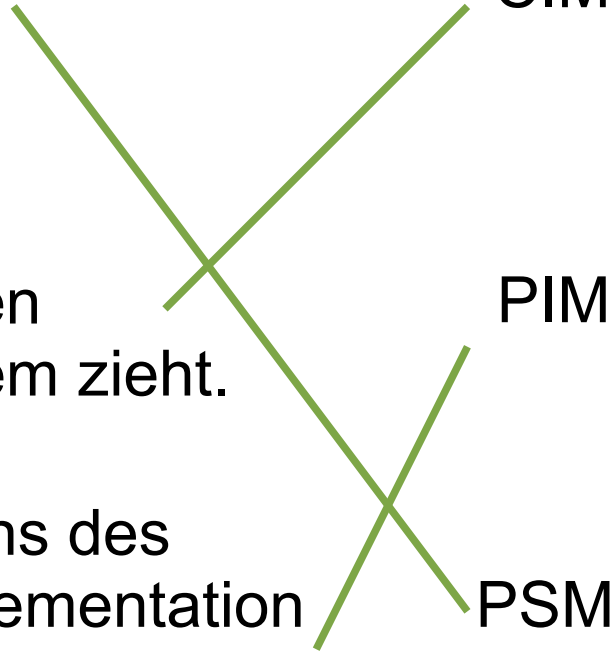
LEHRSTUHL 14
SOFTWARE ENGINEERING

- Welche **MDA-Begrifflichkeiten** sind zu folgenden Aussagen zuzuordnen?

Kann alle **Informationen umfassen**, die notwendig sind, um das System zu erzeugen und in Betrieb zu nehmen.

Beschreibt die **Nutzung des Systems**, seine Anforderungen an seine Umgebung, sowie den Nutzen, den die Umgebung aus diesem System zieht.

Spezifikation der Struktur und des Verhaltens des Systems unabhängig von den später zur Implementation einzusetzenden Hardware- & Softwareplattformen.



1.2 Modell-basierte Software-entwicklung

- MDA: Grundlagen und Konzepte
- **Metamodellierung**
- Modelltransformation

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.

Metamodell + Constraints (OCL)

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

Informelle Beschreibung

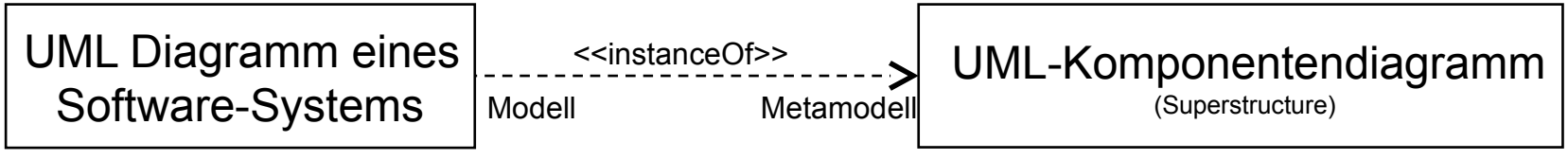
Model und Metamodell der UML

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:

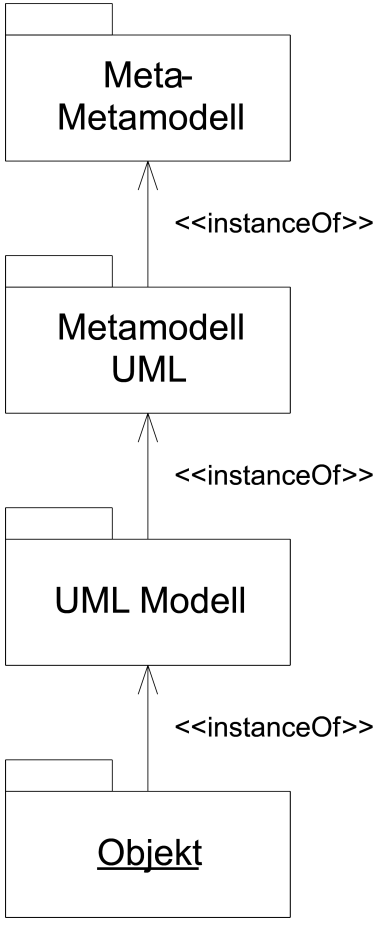
- Mittel Wahl zur Erstellung der Modelle innerhalb MDA.

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.

Modell und Metamodell der UML

Metamodellhierarchie



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, Abhängigkeit 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.

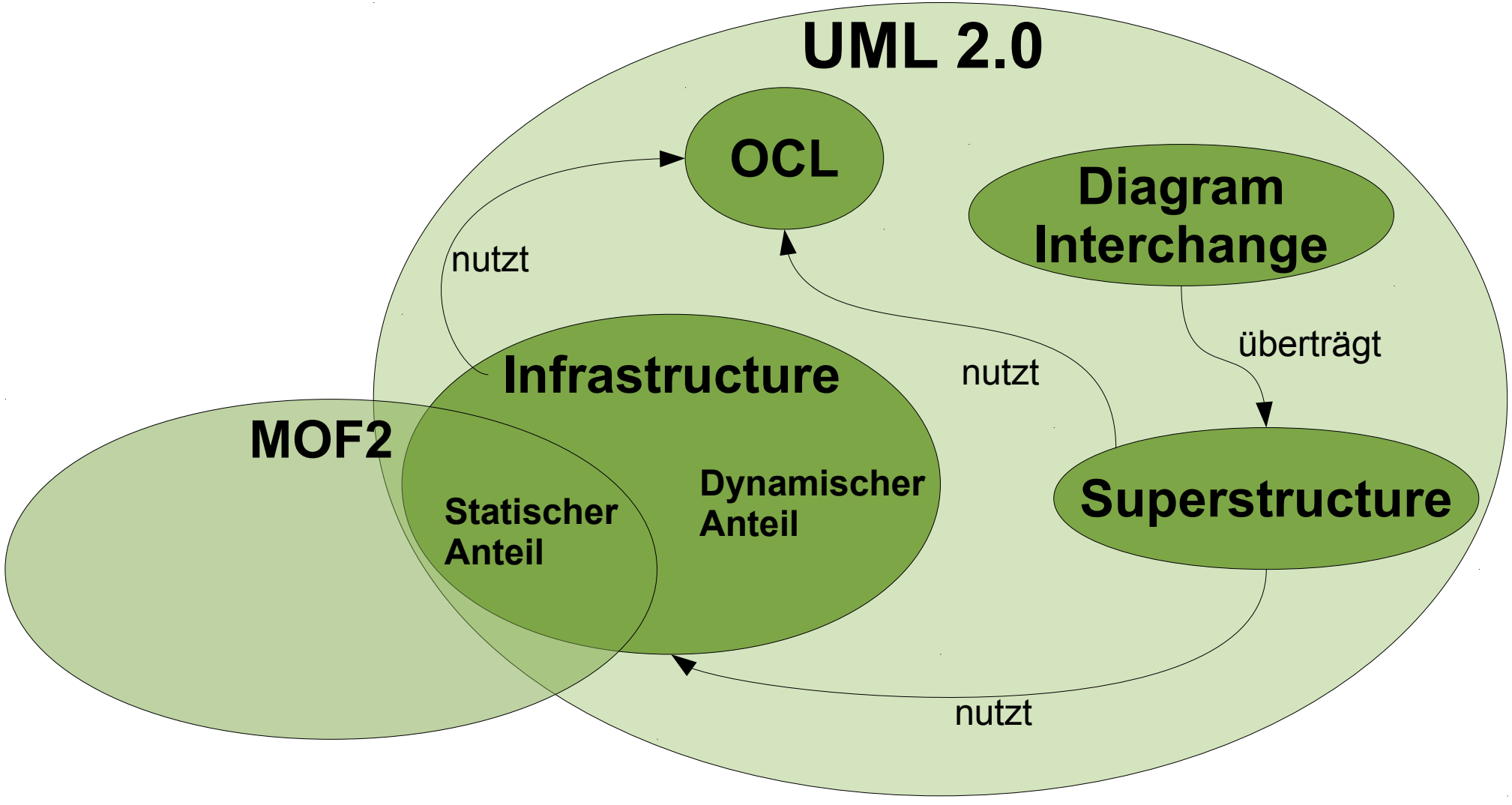
   Infrastructure

2. - Direkte Unterstützung von Skalierbarkeit und Abkapselung für **Verhaltensmodellierung**.

   - Eindeutige **Definition der Semantik** von Relationen wie Generalisierung, Abhängigkeit und Assoziation.

   Superstructure

UML Spezifikation



Diskussionsfrage: MDA Standards

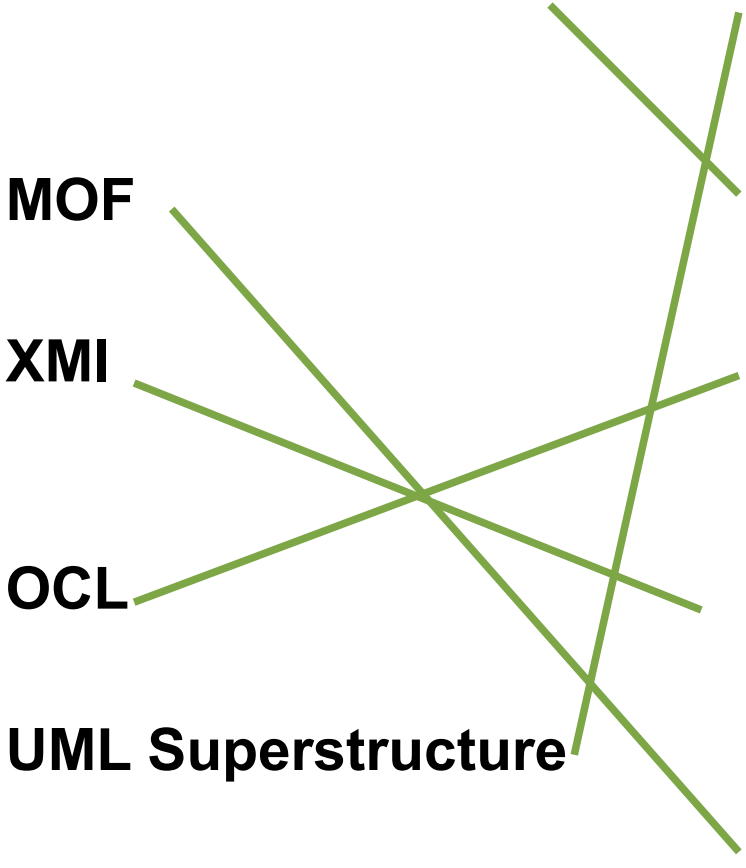
Welche **Aussagen** passen zu den **Kernelementen der UML** ?

UML Infrastructure	Definiert bekannte Diagrammarten (wie Klassen- / Aktivitätsdiagramm).
MOF	Bildet die Basis aller UML-Metamodelle.
XMI	Formale, textbasierte Sprache zur Präzisierung von Modellen.
OCL	Format für Austausch der Modelle.
UML Superstructure	Importiert die in der Infrastruktur definierte abstrakte Syntax und erweitert sie um Dienste.

Diskussionsfrage: MDA Standards

LEHRSTUHL 14
SOFTWARE ENGINEERING

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



UML Infrastructure

MOF

XMI

OCL

UML Superstructure

Definiert bekannte Diagrammarten (wie Klassen- / Aktivitätsdiagramm).

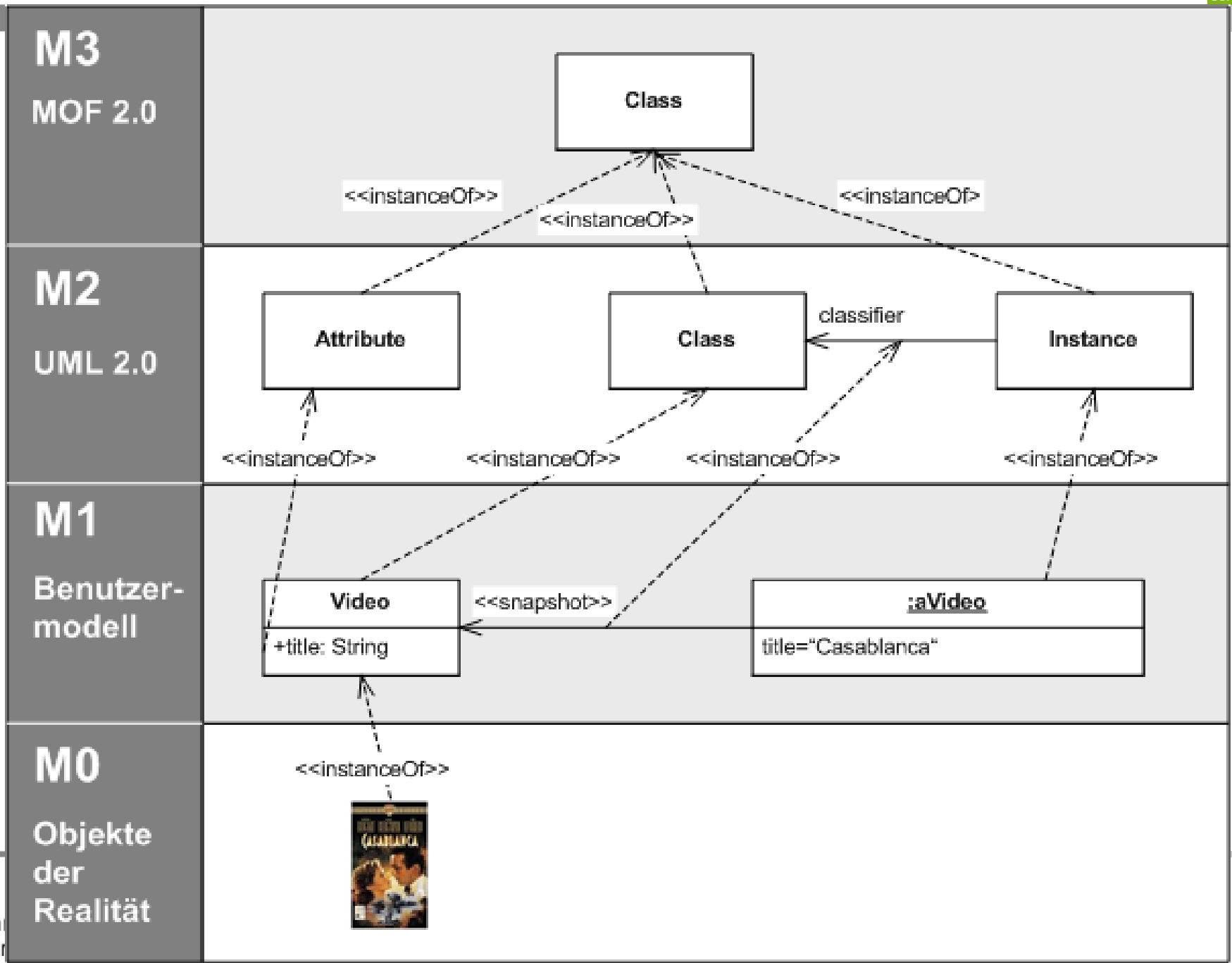
Bildet die Basis aller UML-Metamodelle.

Formale, textbasierte Sprache zur Präzisierung von Modellen.

Format für Austausch der Modelle.

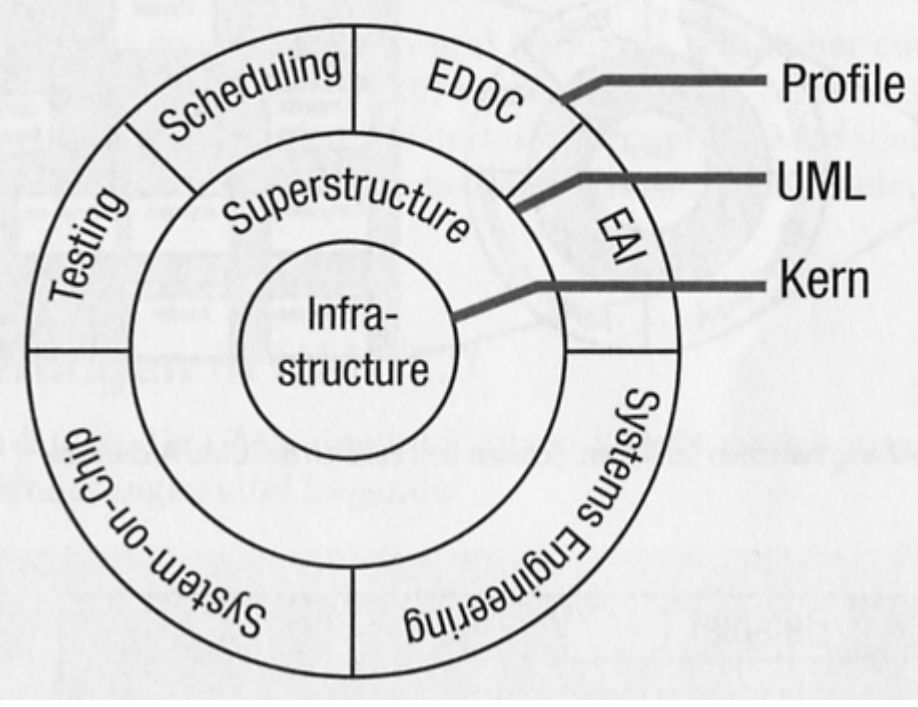
Importiert die in der Infrastruktur definierte abstrakte Syntax und erweitert sie um Dienste.

Meta Ebenen: Beispiel



Schichten-Architektur der UML:

- **Kern:** Infrastructure (Metamodel).
- Eigentliche Sprachdefinition der UML: Superstructure.
- Optionale Erweiterungen: Profile.



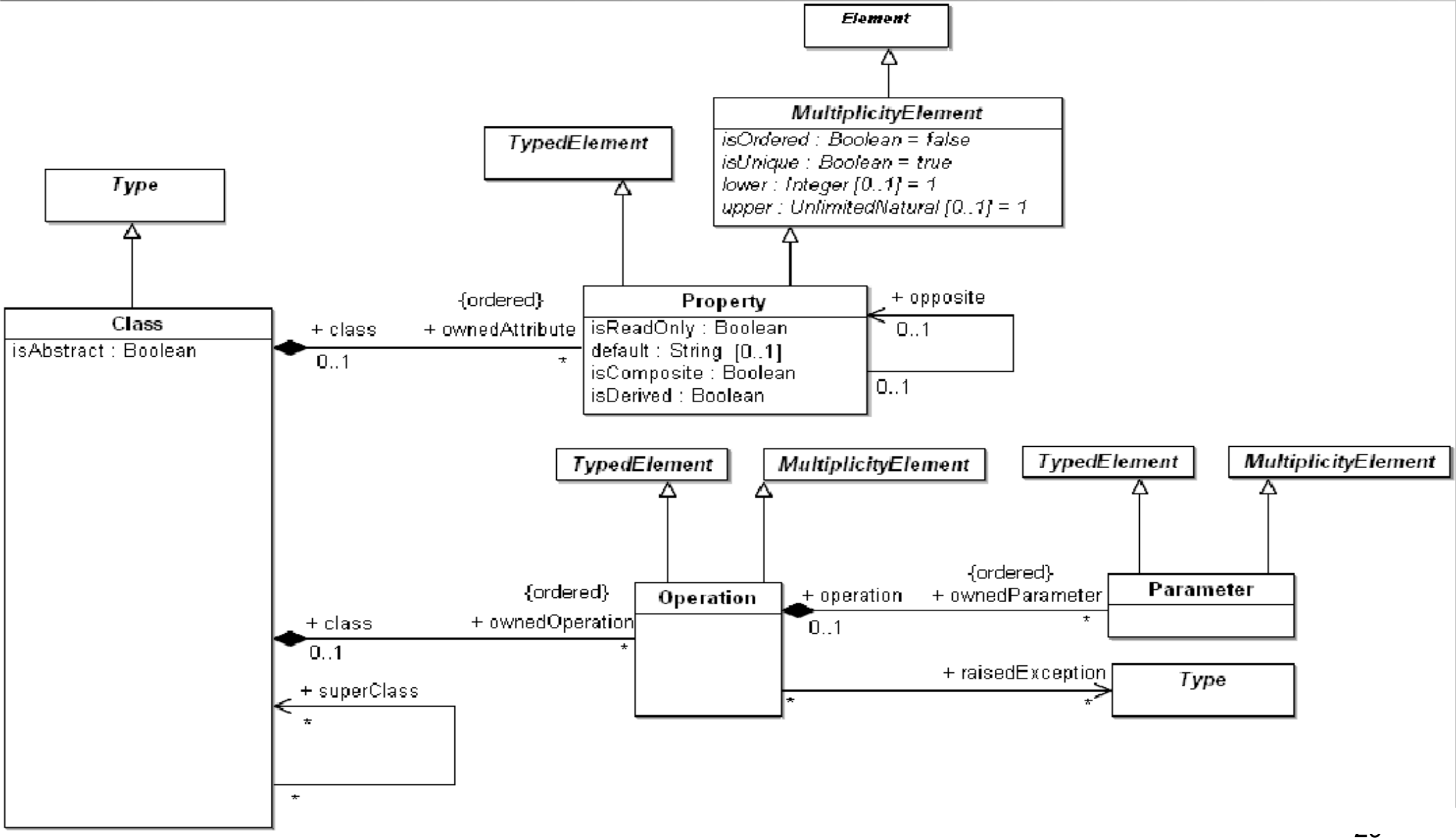
Quelle: UML für Studenten, Harald Störrle

Wege zum „eigenen“ Metamodell

Wege zum „eigenen“ Metamodell:

- **Eigenes Metamodell** „from scratch“:
  - Aufwändig.
- Direkte, beliebige **Erweiterung eines Metamodells:**
  - Voraussetzung: Uneingeschränkter Zugriff auf Metamodell.
  - Verletzt exisitierende Semantik.
- **Erweiterung des UML-Metamodells** durch:
  - UML Profile.
  - Vorgesehene Schnittstelle zum UML-Metamodell.
  - Lediglich Verfeinerung.

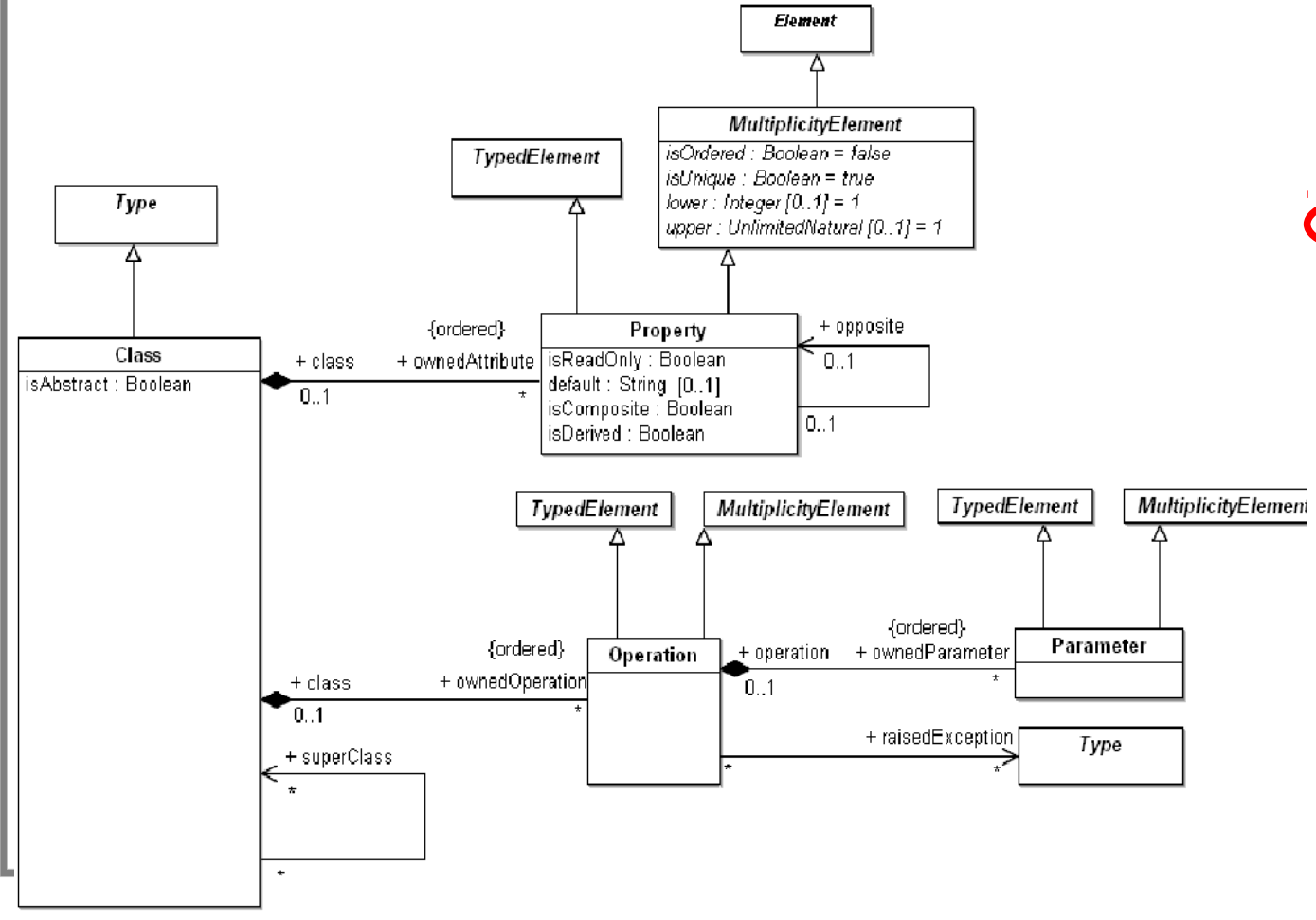
UML Metamodell Beispiel
Core::Basic:ClassDiagram

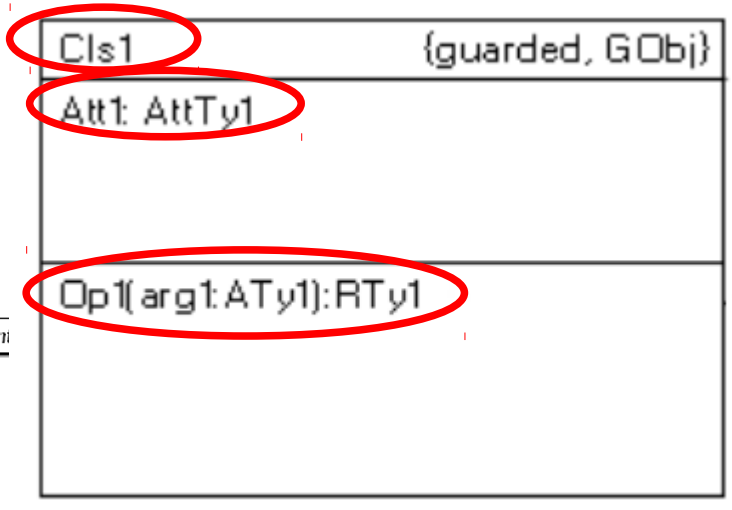


Diskussionsfrage: Klassendiagramm vs. Metamodell

LEHRSTUHL 14
SOFTWARE ENGINEERING

Wo finden sich die **rot markierten Elemente** aus dem **Klassendiagramm** im **Metamodell** wieder ?
(Tipp: Ein Attribut ist im UML-Metamodell eine „Property“ einer Klasse.)

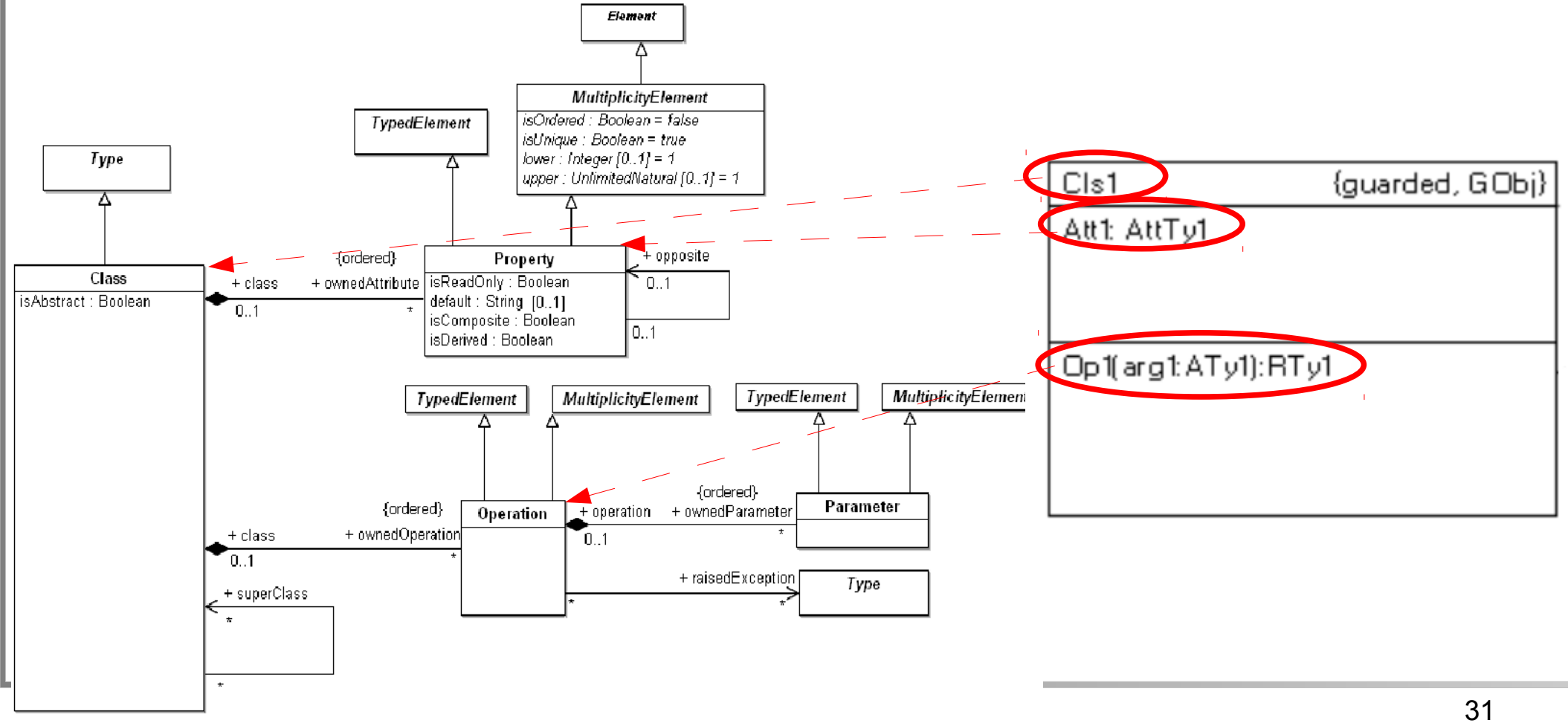




Diskussionsfrage: Klassendiagramm vs. Metamodell

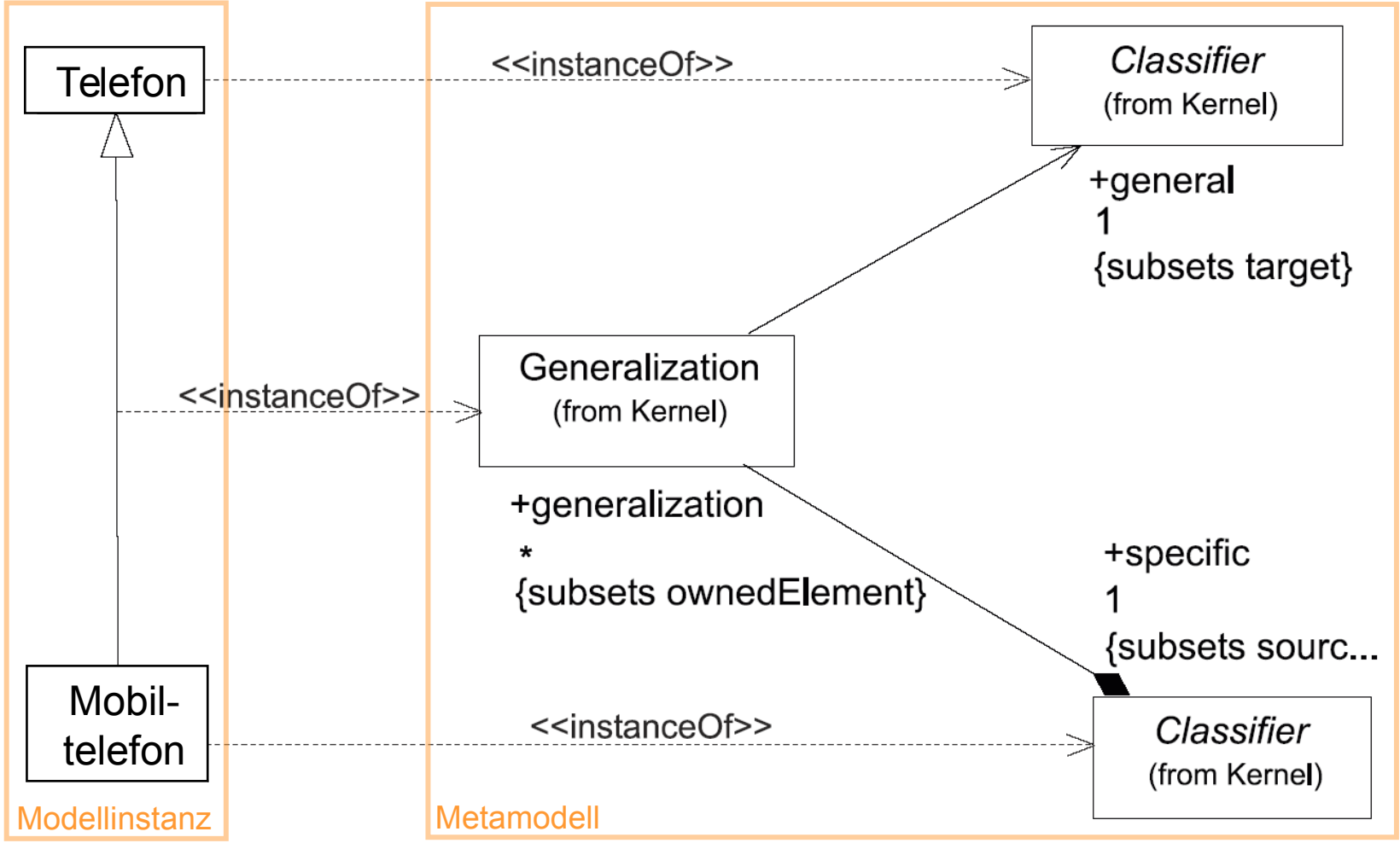
LEHRSTUHL 14
SOFTWARE ENGINEERING

Wo finden sich die **rot markierten Elemente** aus dem **Klassendiagramm** im **Metamodell** wieder ?
(Tipp: Ein Attribut ist im UML-Metamodell eine „Property“ einer Klasse.)



Metamodell der UML
Beispiel: Generalisierung

Definition und Beispiel der Generalisierung:

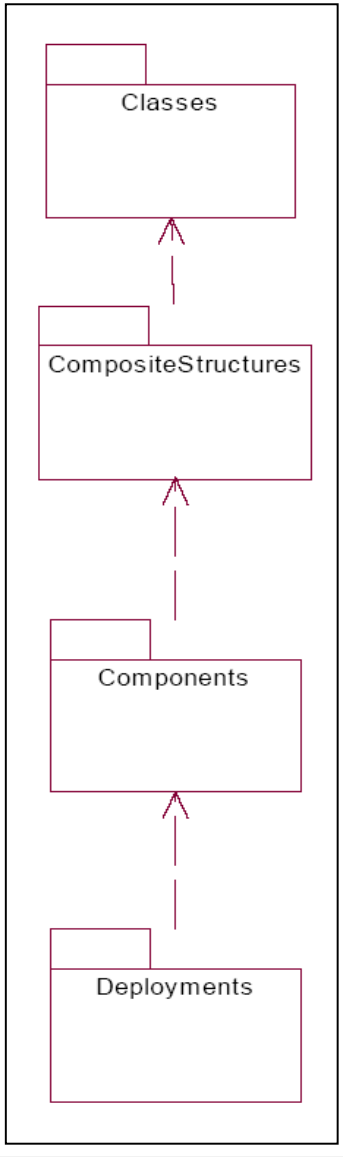


Quelle: Softwareentwicklung mit UML 2, M.Born, E.Holz, O.Kath

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

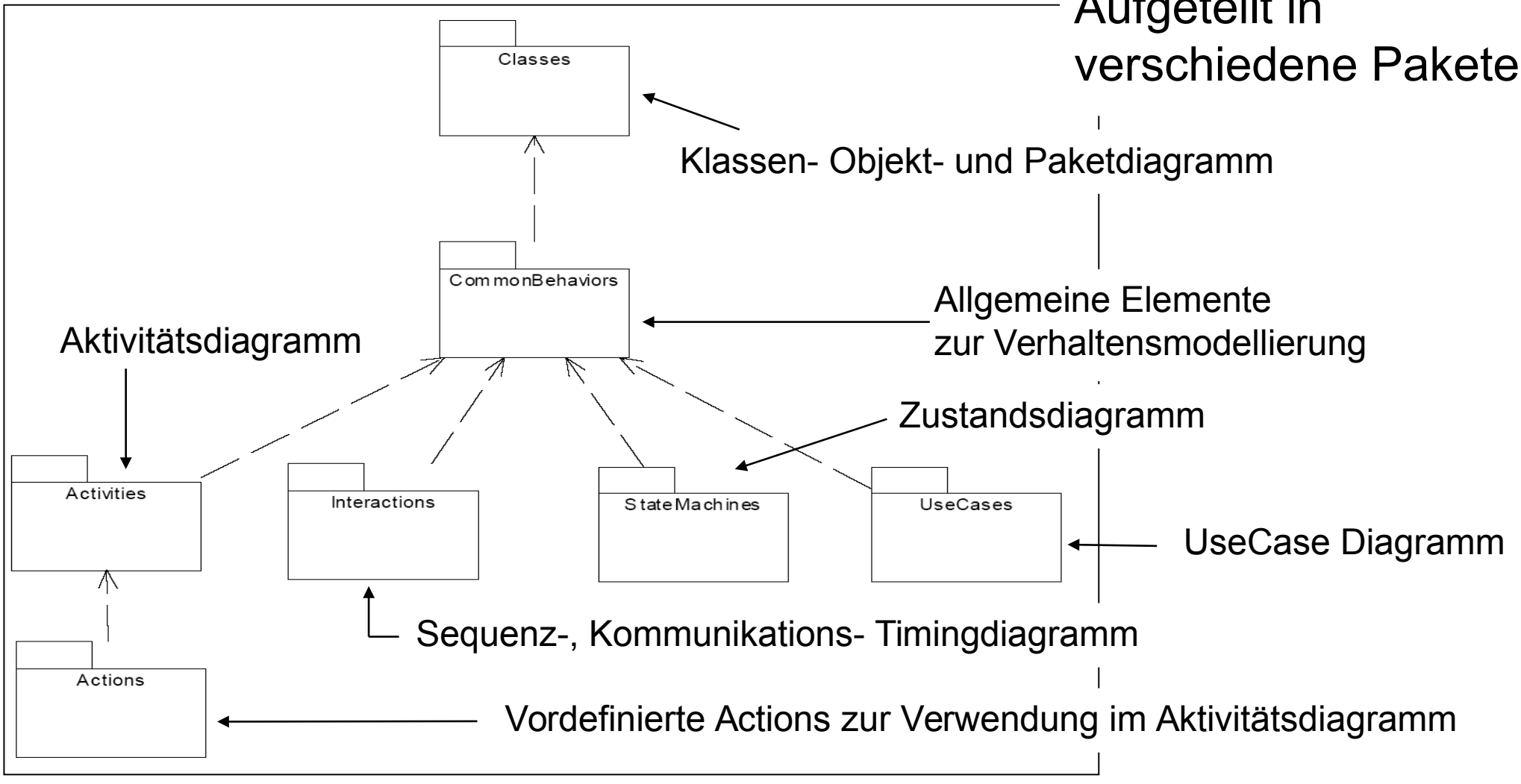
= Kompositionsstrukturdiagramm

= Komponentendiagramm

= Deploymentdiagramm

Quelle: UML 2.0 Superstructure

Aufbau des UML Metamodells
UML Verhaltensdiagramme

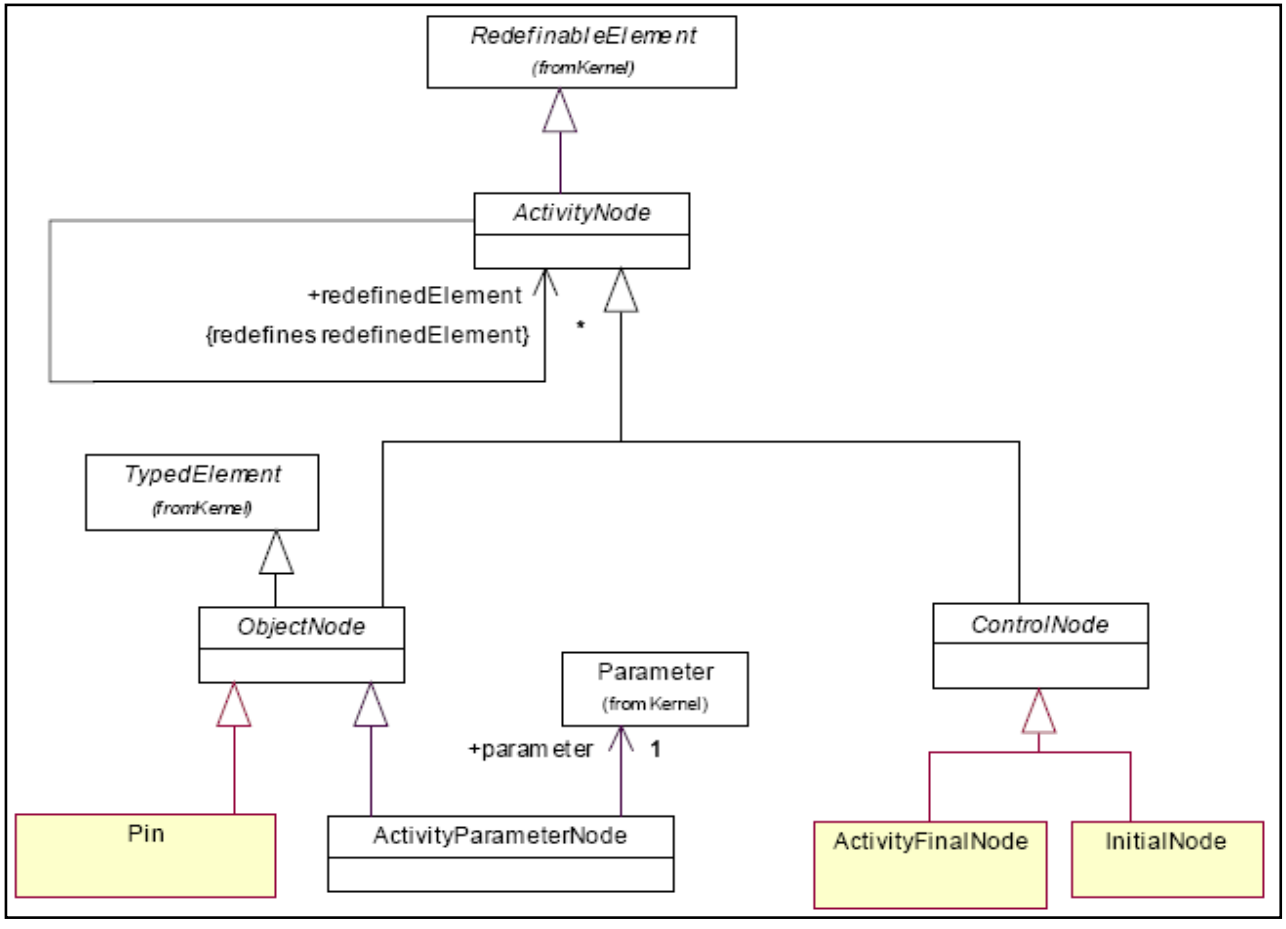


Verhaltensdiagramme

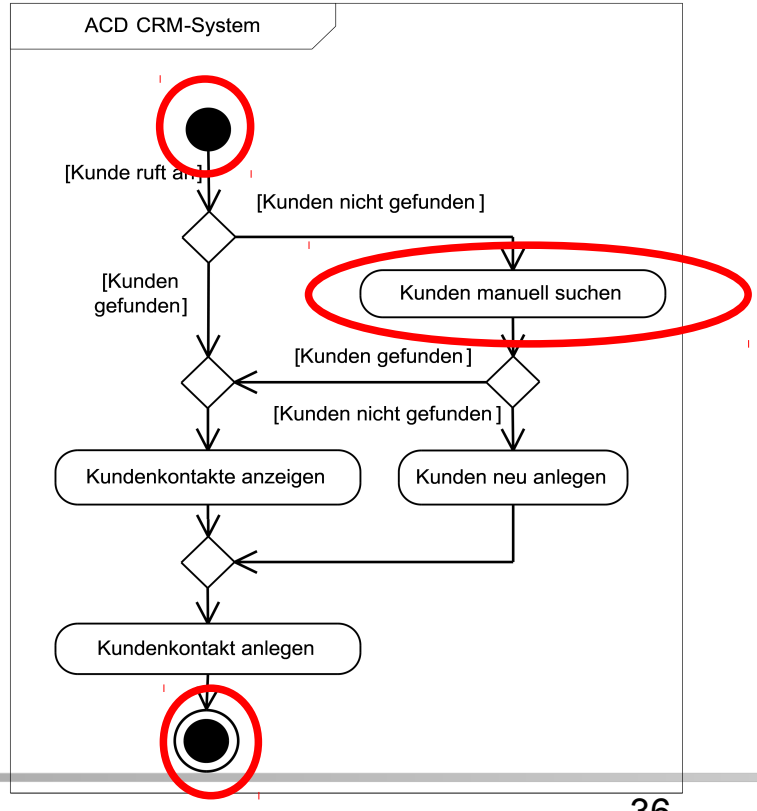
Quelle: UML 2.0 Superstructure

Diskussionsfrage: Aktivitätsdiagramm vs. Metamodell

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

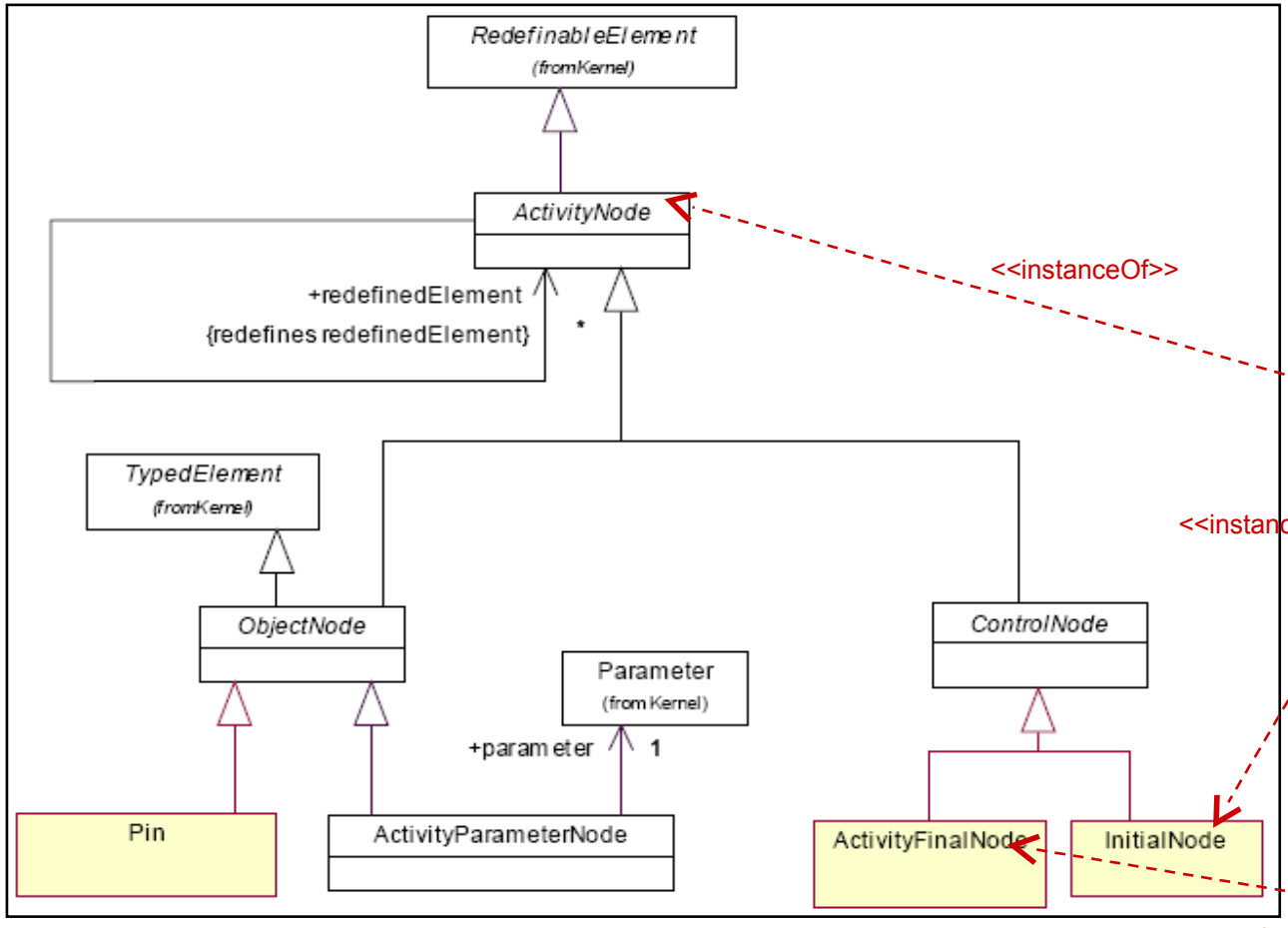


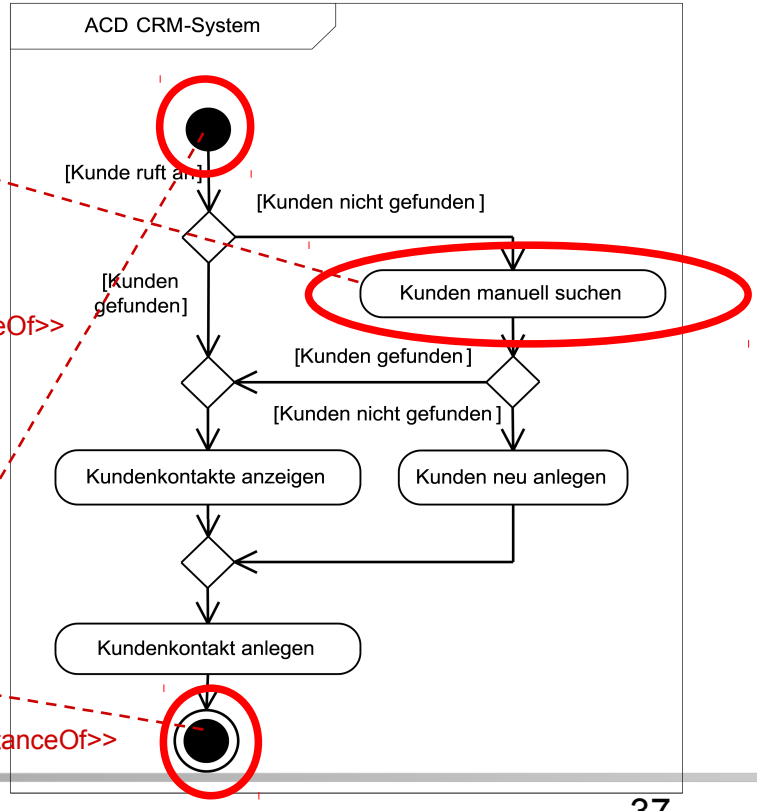
Quelle: UML 2.0 Superstructure



Diskussionsfrage: Aktivitätsdiagramm vs. Metamodell

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





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.

Semantik im UML Metamodell
Beispiel: Action im Activity Diagramm

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.

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** im UML beschreiben.
- Definition der (Domänen-) Konzepte benötigt.
  - Erklärender Text zu einem Diagramm. → Ungeeignet
  - Erweiterungen der UML-Notationselemente um fachliche Konzepte.

UML **unterstützt Anpassung** auf Modellebene mit sogenannten „Profilen“.

UML Profile
Definition

UML-Profil:

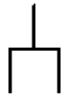
- Spezialisierung von Standard UML-Elementen zu konkreten Metatypen.
- Profil zu Modell hinzufügbar und im gesamten Modell verfügbar.
- Verschiedene Profile für verschiedene Anwendungsdomänen.
- Einige Profile vordefiniert und bei OMG verfügbar.

Definition eines Profils:

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

UML Profil
Erweiterungsmöglichkeiten

- Stereotypen:

  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.

- Profil:

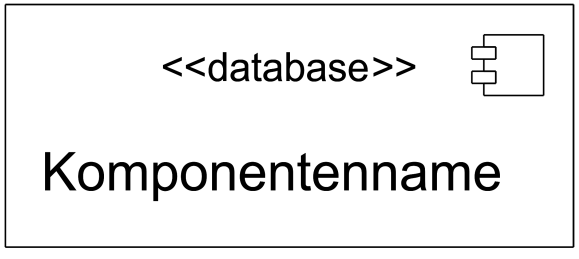
  sammelt obige Informationen.

Definition von Stereotypen:

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

Konkretes Beispiel
einer Definition:

Spätere Verwendung im
Komponentendiagramm:



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.

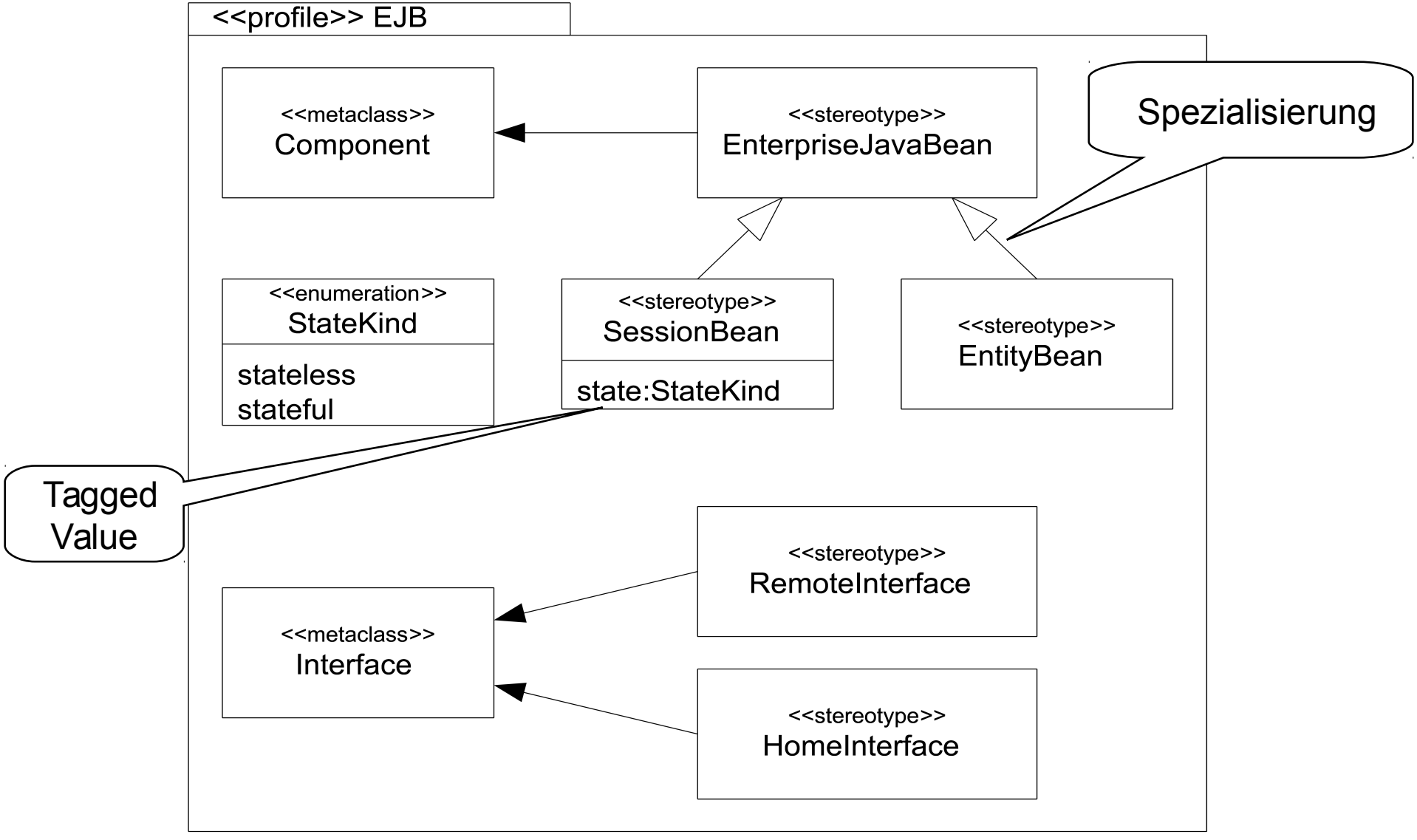
UML-Profile
Beispiel Stereotypen

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

UML-Profile

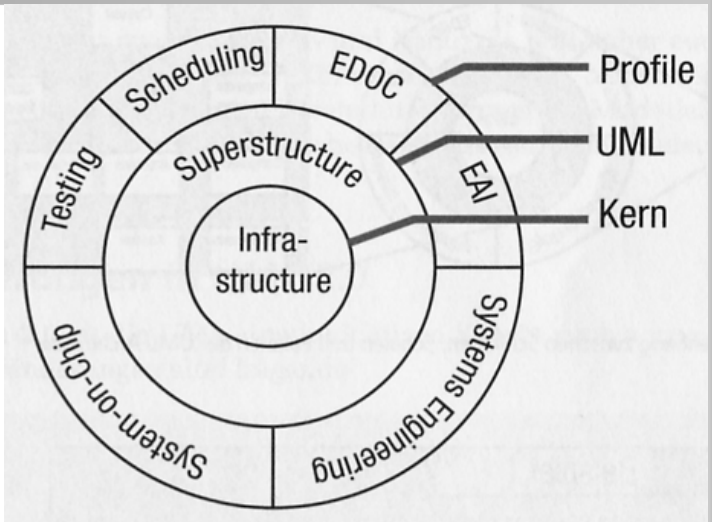
Beispiel-Profil: Enterprise Java Beans



UML-Profile
Semantik

Semantik:

- Durch Text, OCL constraints o.a. „definiert“.
- Wichtig für
  - richtige Auswahl der Stereotypen
  - Verständnis beim Lesen eines Diagramms.



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 technisches Datenmodell

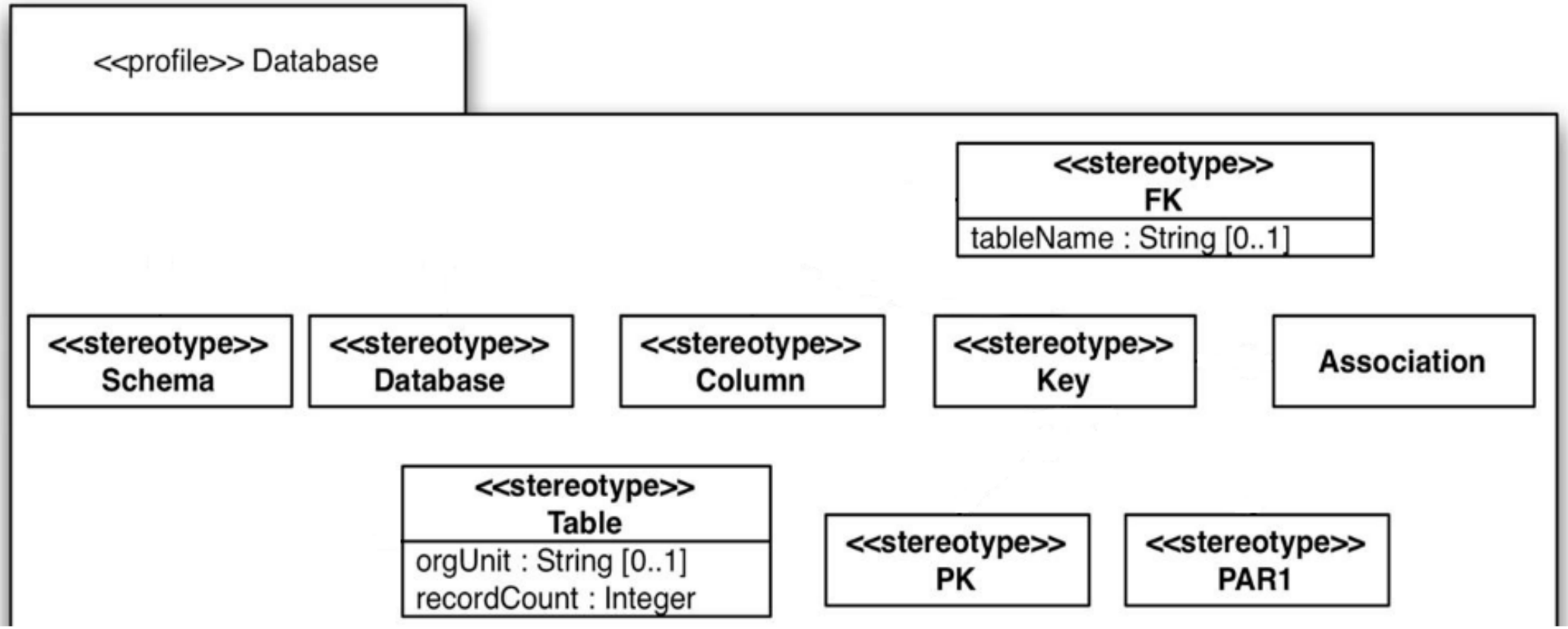
Grundlegende Konstrukte und ihre Darstellung:

- **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 (zusammengesetzte) Sekundärschlüssel (beide Stereotype abgeleitet von <<Key>>).
- **Fremdschlüssel:** Attribut markiert mit Stereotyp <<FK>>.

Diskussionsfrage:
Metamodell für technisches Datenmodell

LEHRSTUHL 14
SOFTWARE ENGINEERING

Zu welchen Metamodellelementen gehören die Stereotyp-Definitionen ?

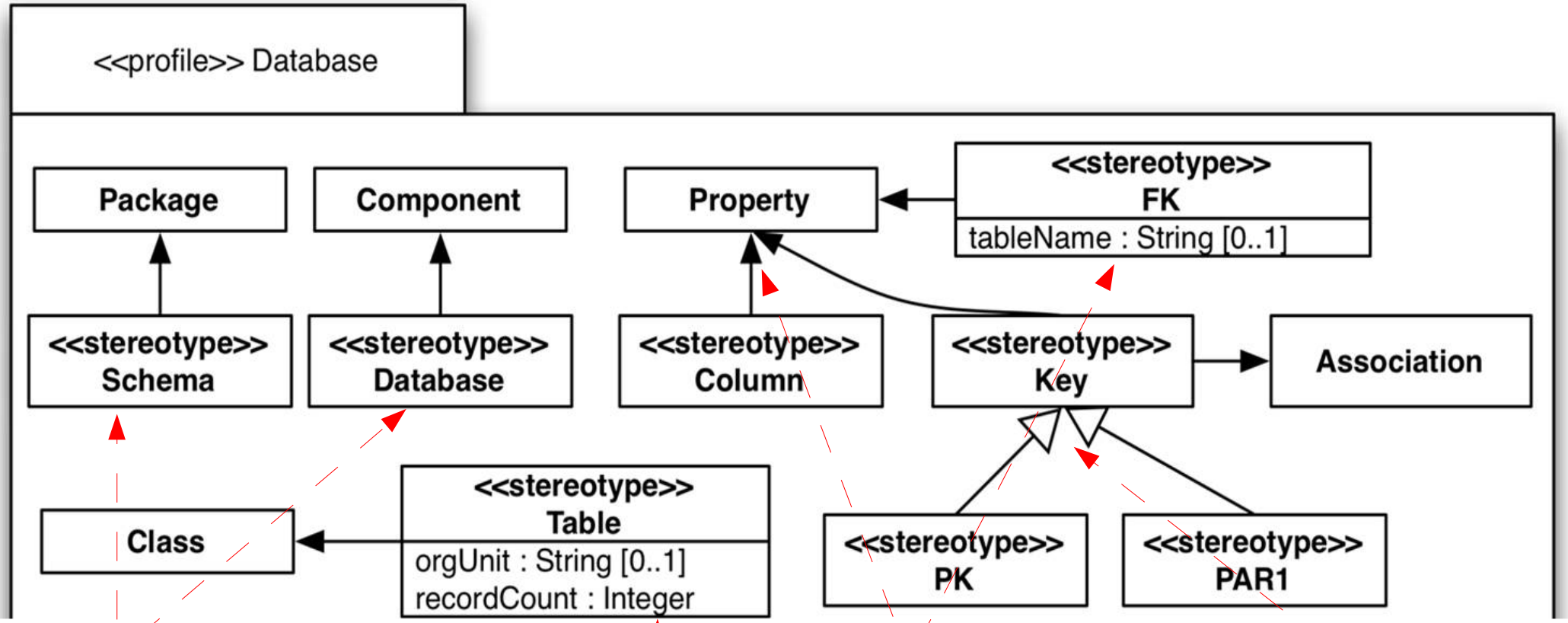


- **Datenbank:** UML-Komponente mit Stereotyp <<Database>>.
- **Schema:** UML-Paket mit Stereotyp <<Schema>>.
- **Tabelle:** UML-Klasse mit Stereotyp <<Table>>.
- **Spalten:** Attribute (Property) der als <<Table>> markierten Klasse; mit Stereotyp <<Column>> versehen.
- **Primärschlüssel:** Attribut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

Diskussionsfrage:
Metamodell für technisches Datenmodell

LEHRSTUHL 14
SOFTWARE ENGINEERING

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>>.

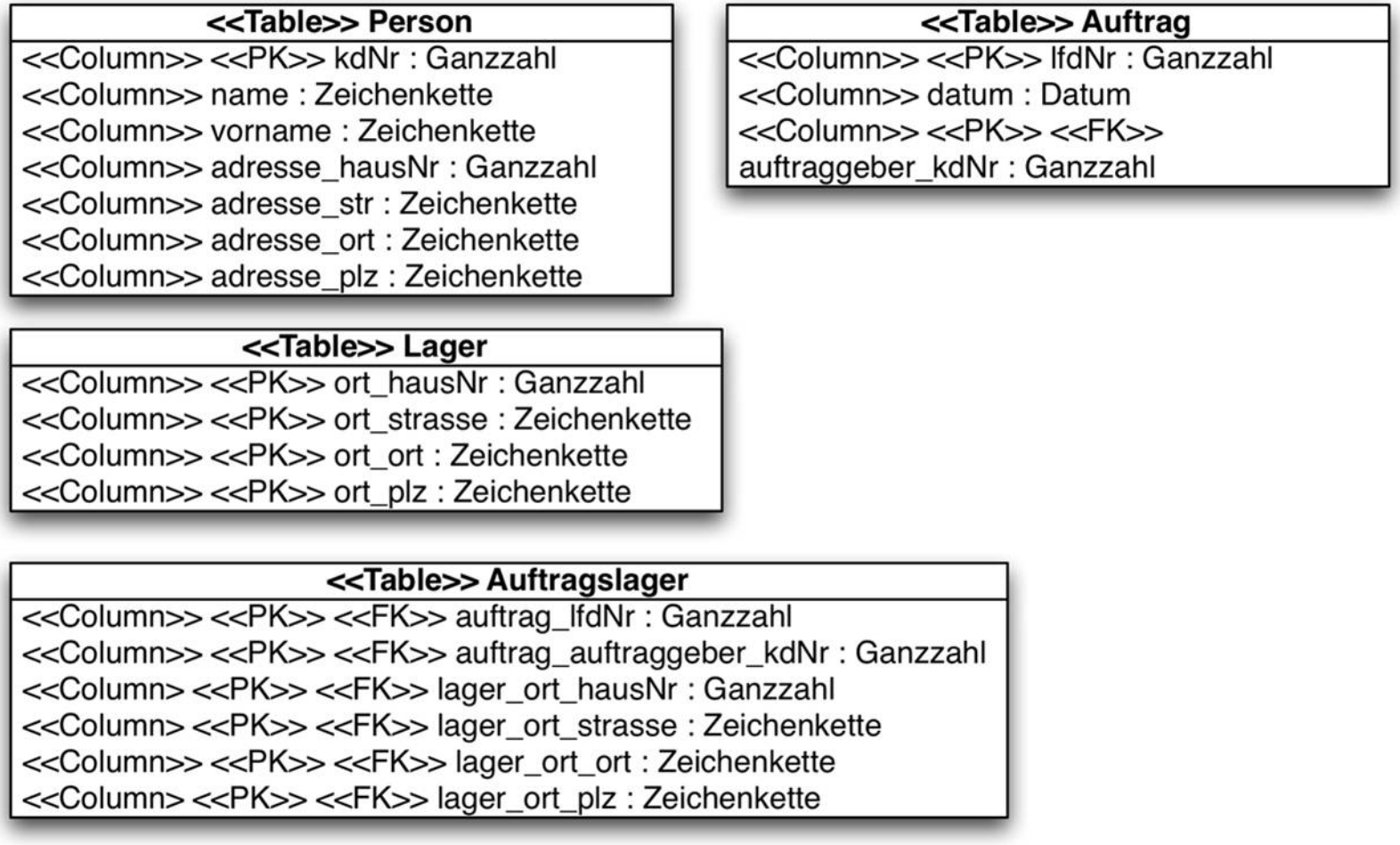
Beispiel technisches Datenmodell: Zusammenfassung

Zusammenfassung:

- **Import der geerbten UML-Metamodellklassen** „Package“, „Component“, „Class“, „Property“ und „Association“ aus deren Paketen „Classes“ und „Components“.
- **Beziehungen mit ausgefüllten Spitzen:** „Extensions“.
  → Verlaufen von Stereotyp zur Metaklasse aus zu ergänzendem Metamodell.
- Stereotype **optional oder zwingend** für Metaklasse.
- **Attribute** des Stereotyps im Modell beim entsprechenden Modellelement **belegen.**
  (Bsp.: Stereotyp „FK“ Attribut „tableName“)

Beispiel technisches Datenmodell

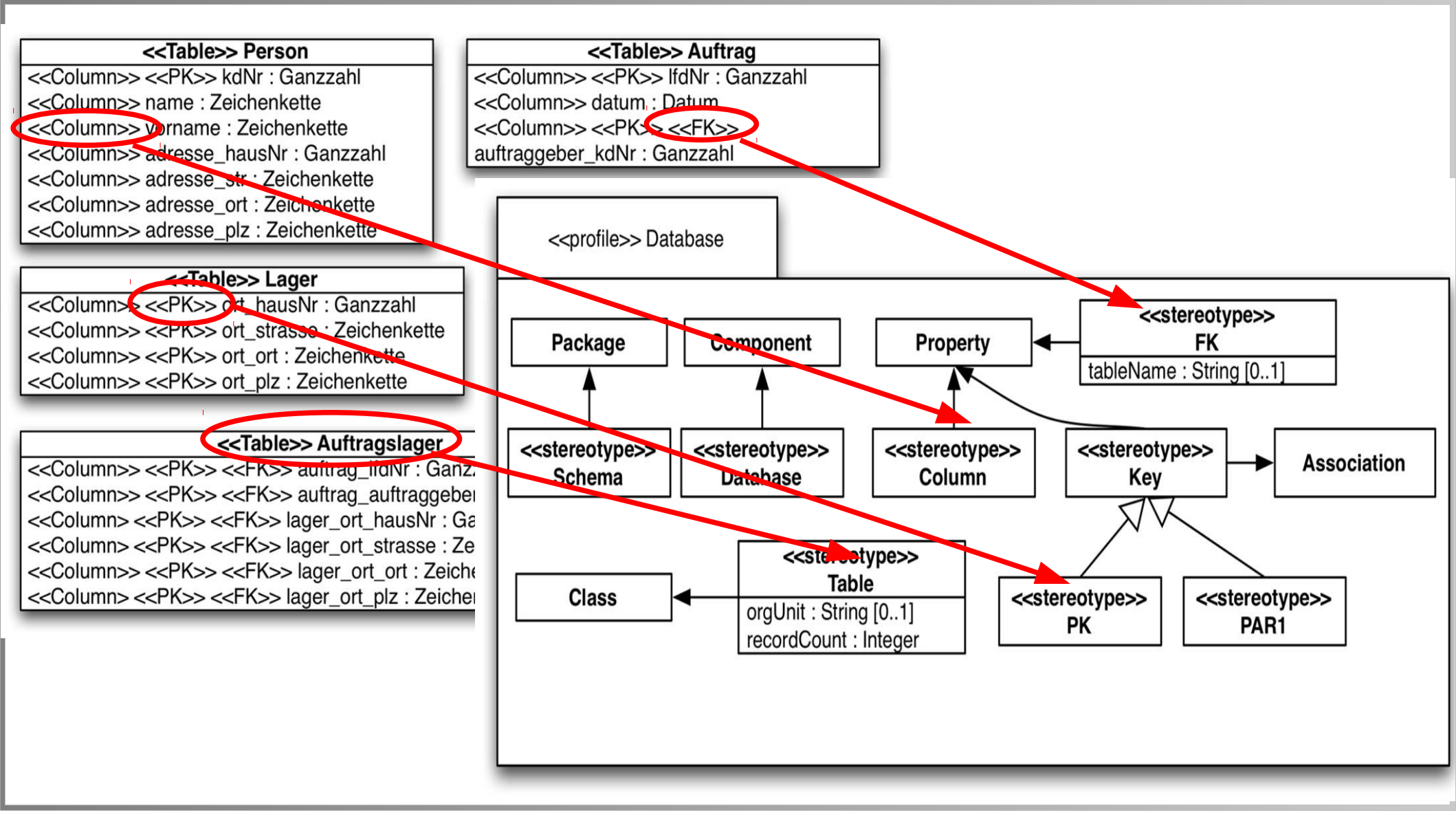
LEHRSTUHL 14
SOFTWARE ENGINEERING



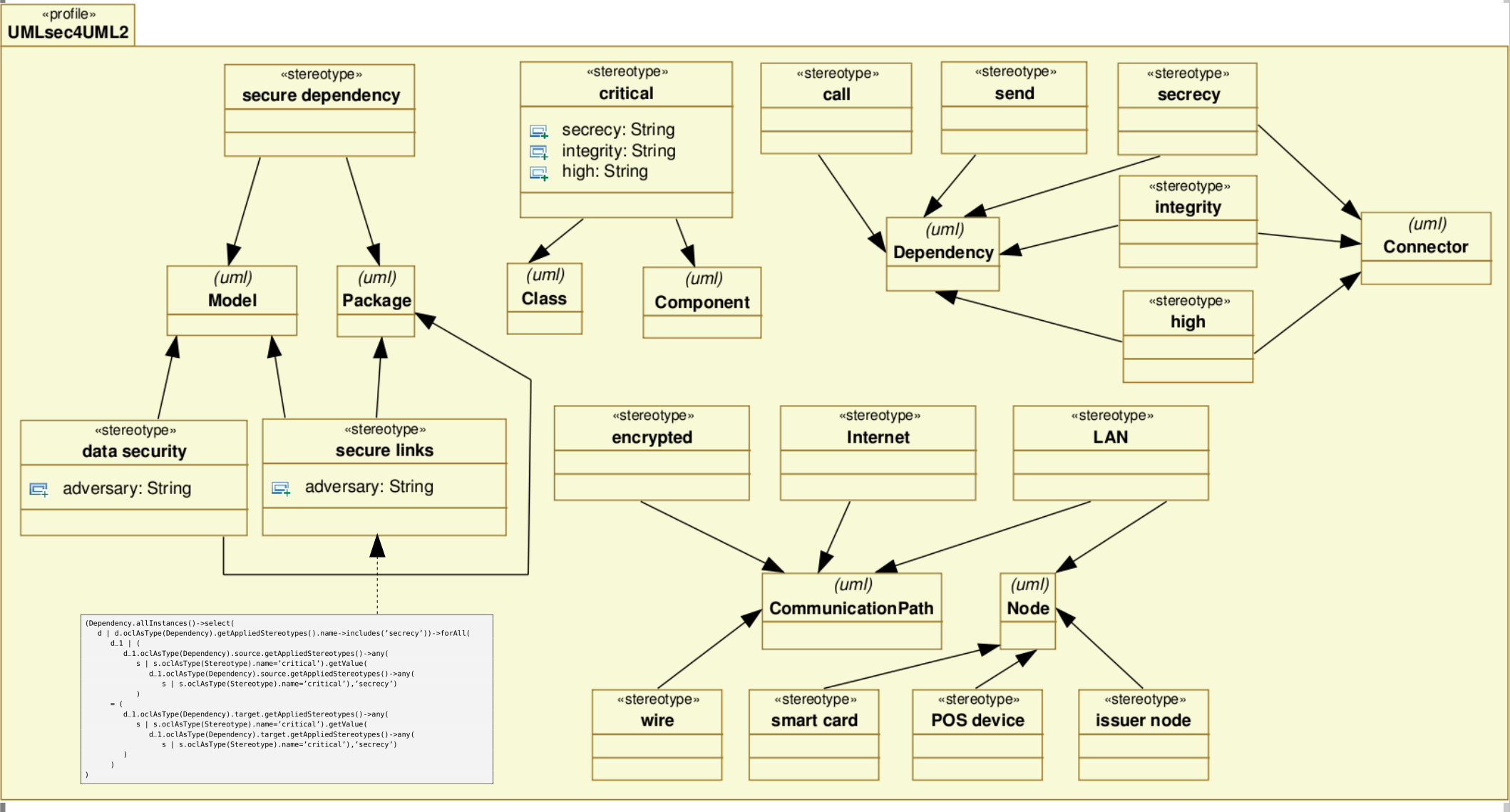
Beispiel für technisches Datenmodell; modelliert mit definiertem UML Profile.

Beispiel technisches Datenmodell: Modell vs. Metamodell

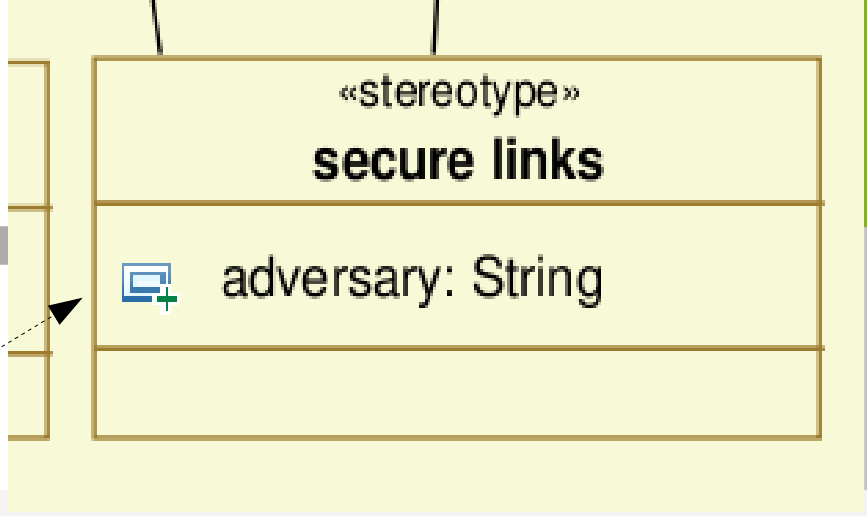
LEHRSTUHL 14
SOFTWARE ENGINEERING



Beispielprofil UMLsec



Stereotype <<secure links>>: OCL constraint

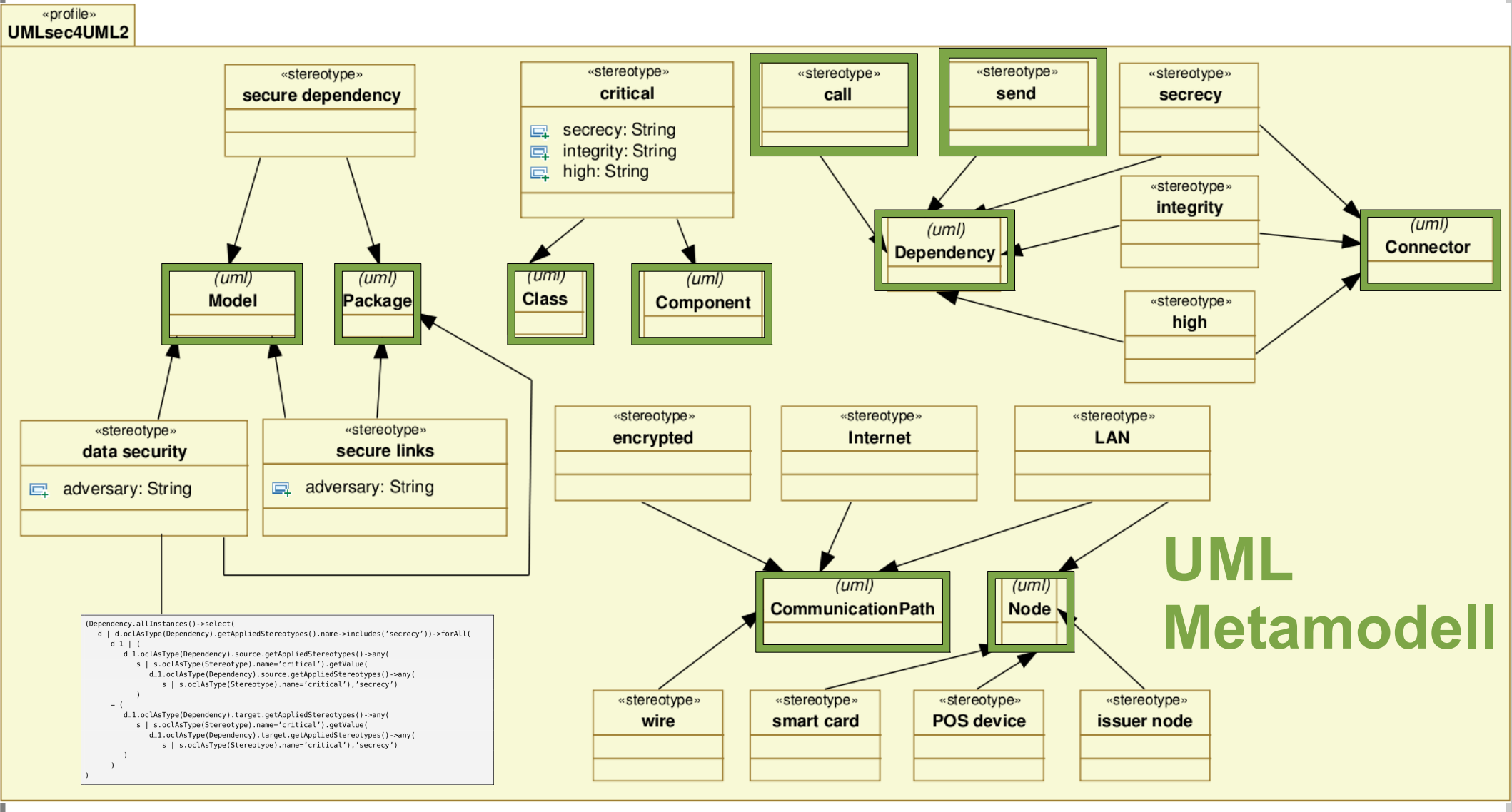


Mehr Einzelheiten dazu in Kap. 4 dieser Vorlesung.

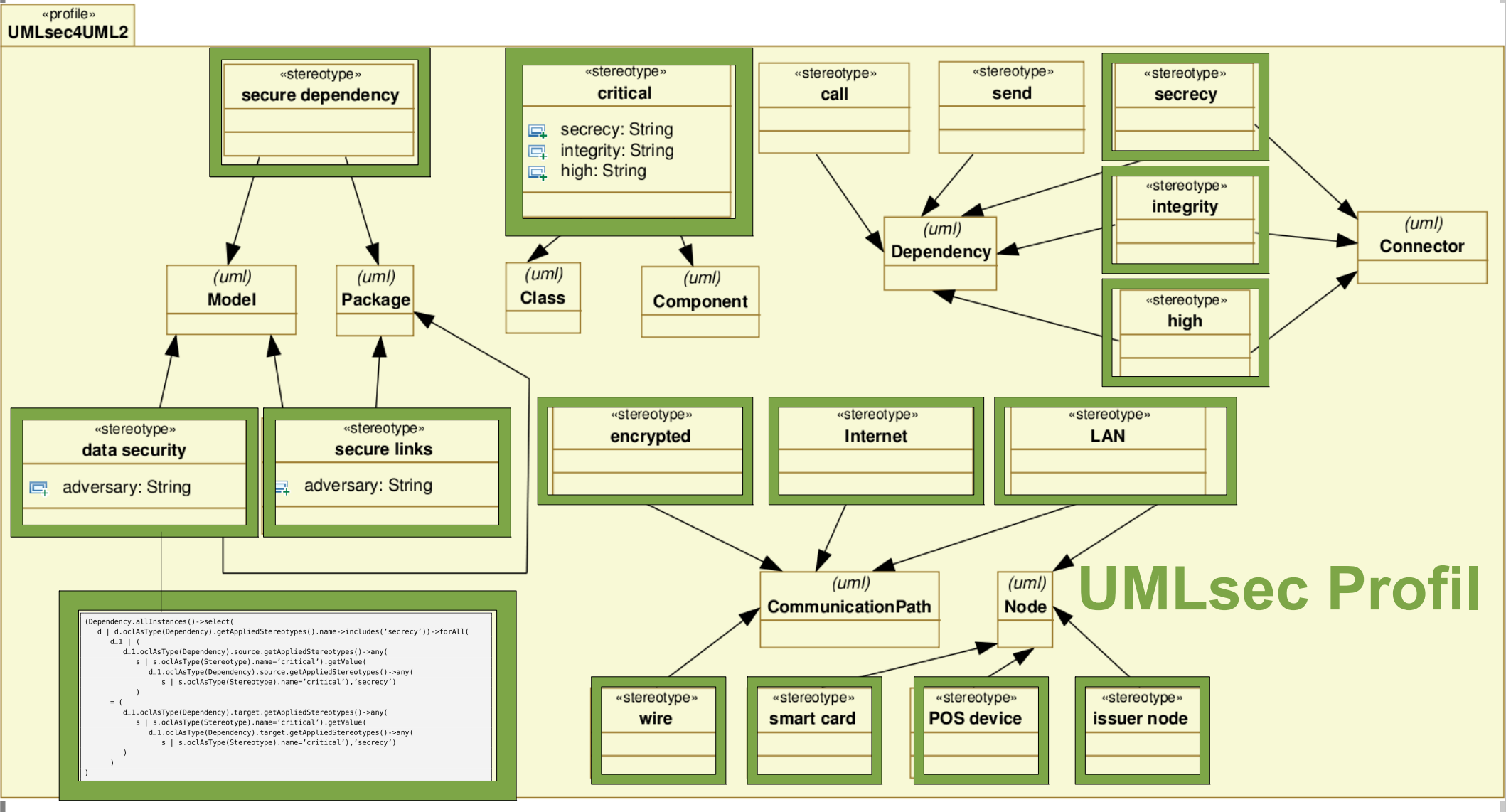
```
(Dependency.allInstances()->select(
   d | d.oclAsType(Dependency).getAppliedStereotypes().name->includes('secrecy'))->forAll(
      d_1 | (
         d_1.oclAsType(Dependency).source.getAppliedStereotypes()->any(
            s | s.oclAsType(Stereotype).name='critical').getValue(
               d_1.oclAsType(Dependency).source.getAppliedStereotypes()->any(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

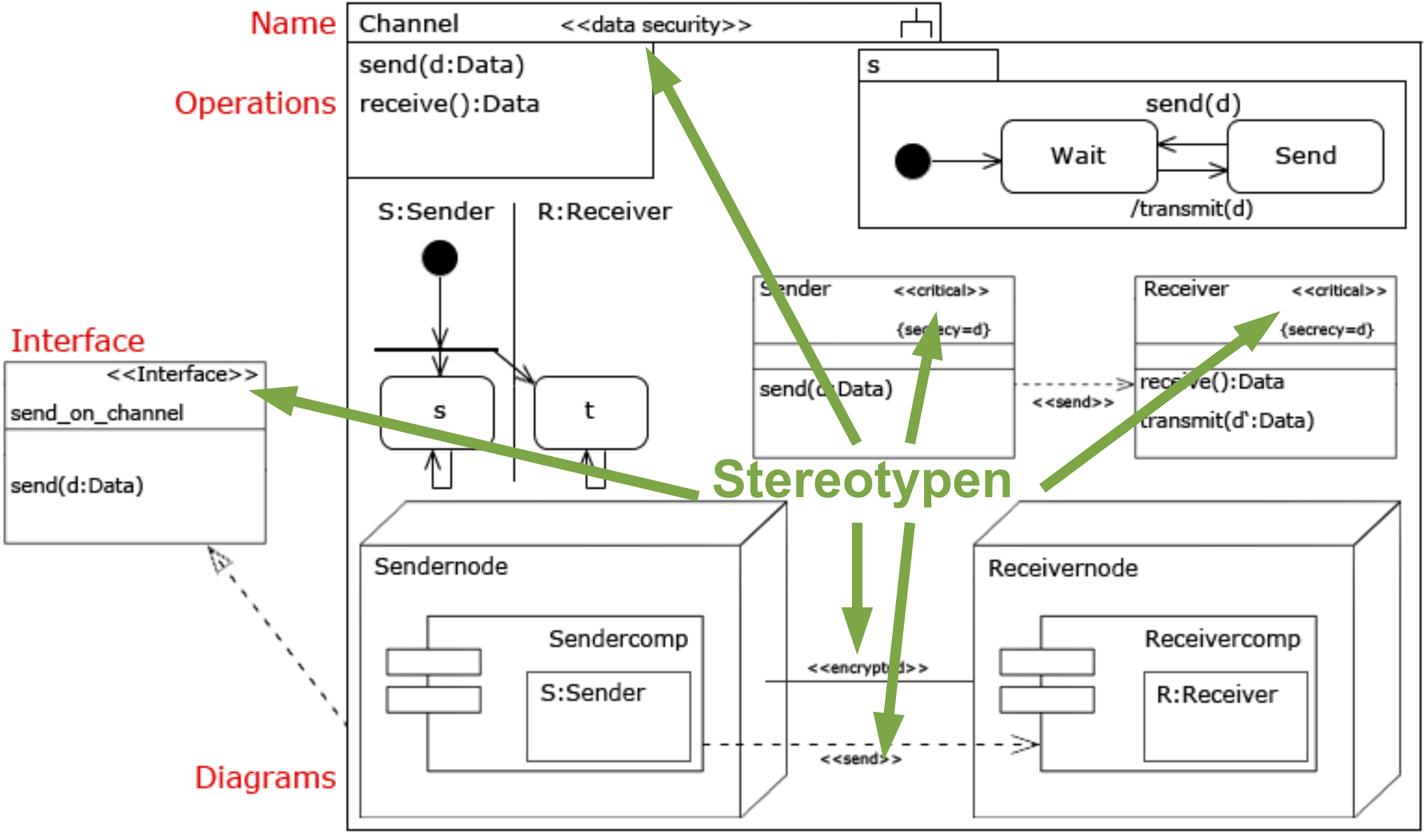
LEHRSTUHL 14
SOFTWARE ENGINEERING



Beispielprofil UMLsec: UMLsec-Profil



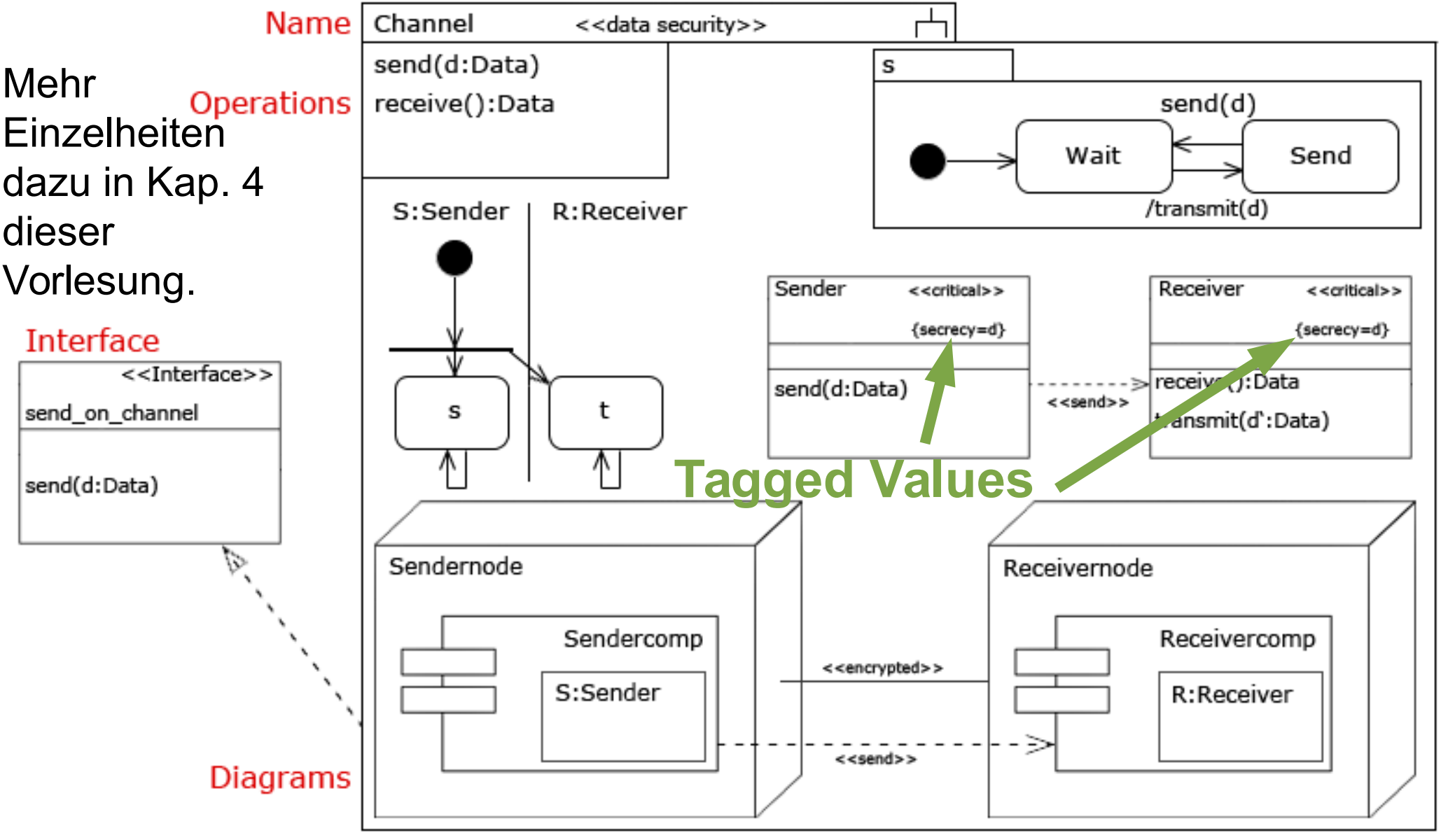
UMLsec Beispielmodell:
Stereotypen



UMLsec Beispielmodell: Tagged Values

LEHRSTUHL 14
SOFTWARE ENGINEERING

Mehr Einzelheiten dazu in Kap. 4 dieser Vorlesung.

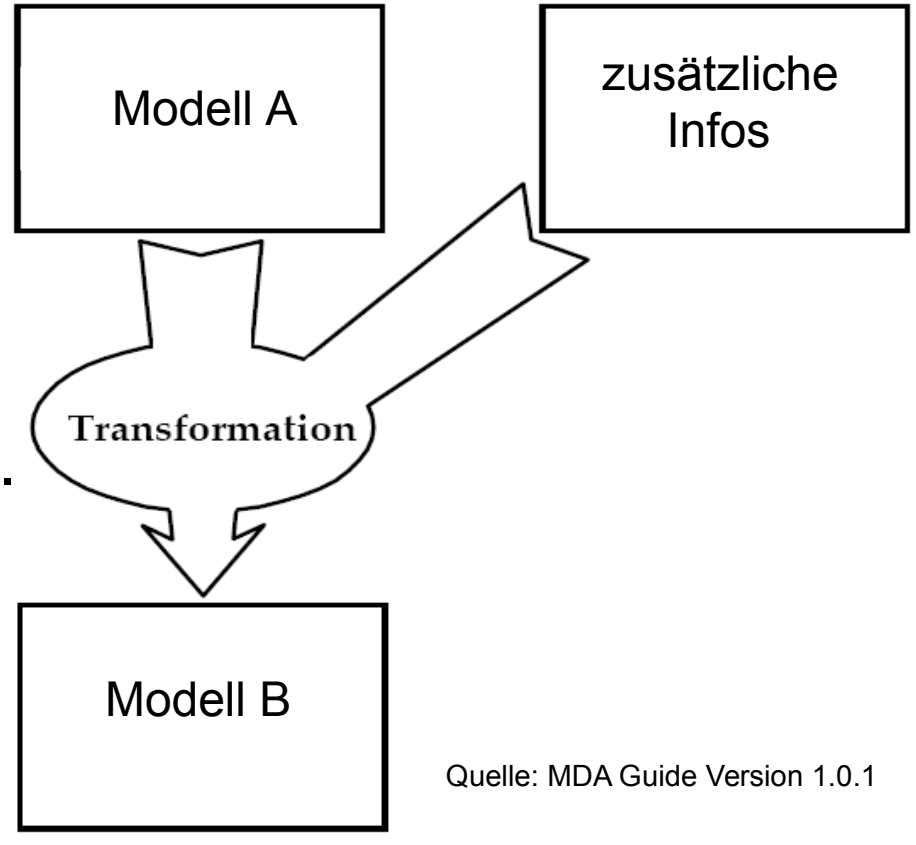


1.2 Modell-basierte Software-entwicklung

- MDA: Grundlagen und Konzepte
- Metamodellierung
- Modelltransformation

Modelltransformation

- Modell A und zusätzliche Information durch Anwendung einer Transformation **in Zielmodell B überführen**.
- **Hintereinanderausführung mehrerer Transformationen** möglich.
- Prinzipiell können alle definierbaren Modelle **Quell- oder / und Zielmodell** sein.
- **Viel Know-How im Generator.**

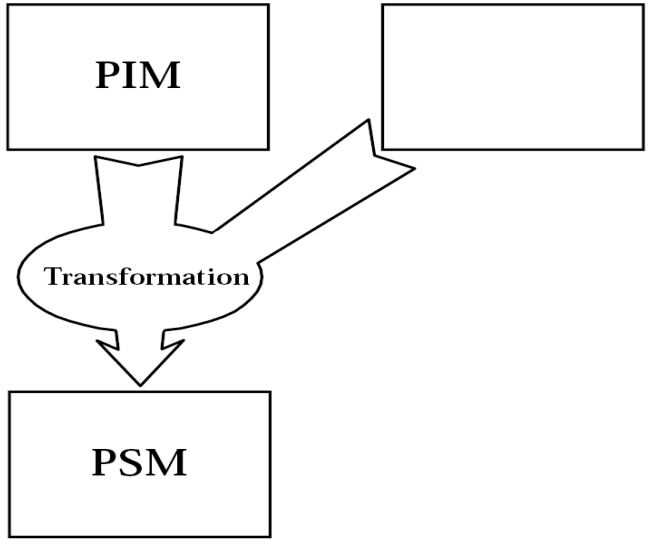


Quelle: MDA Guide Version 1.0.1

- **Spezifikation für Transformation** in Mapping Rules festhalten.
- **Mapping Rules** in Mapping gebündelt.
- Kernstück des MDA-Ansatzes.

Modelltransformation
Horizontale und Vertikale Transformation

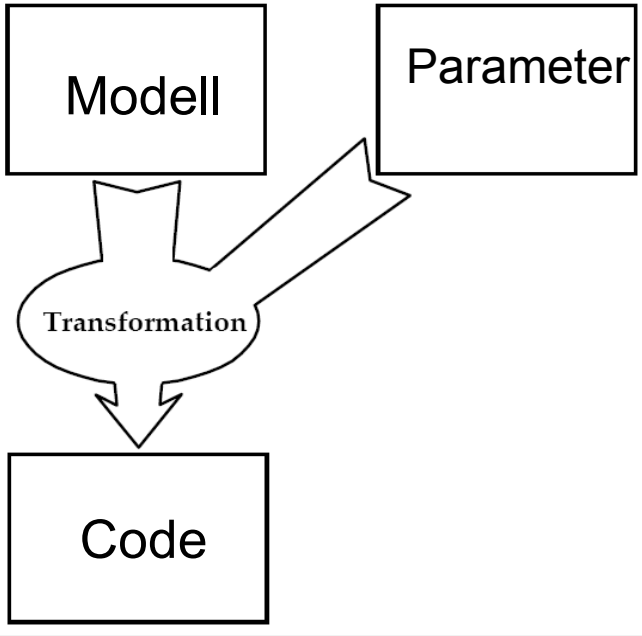
- Bezeichnet **berechenbare Abbildungen** eines Quellmodells in Zielmodell.
- **Ein mögliches Ziel:** Umwandlung eines Modells einer Abstraktionsebene in Modell anderer Abstraktionsebene.
- **Horizontale** Transformation:
  - **Inhaltliche Weiterentwicklung** eines Modells durch Transformation.
- **Vertikale** Transformation:
  - Transformation eines Modells auf **technologiespezifischere Ebene.**
  - Kernstück des MDA-Ansatzes.
  - Zur Transformation in PSM: PIM und „weitere Informationen“ notwendig.



Einsatz:

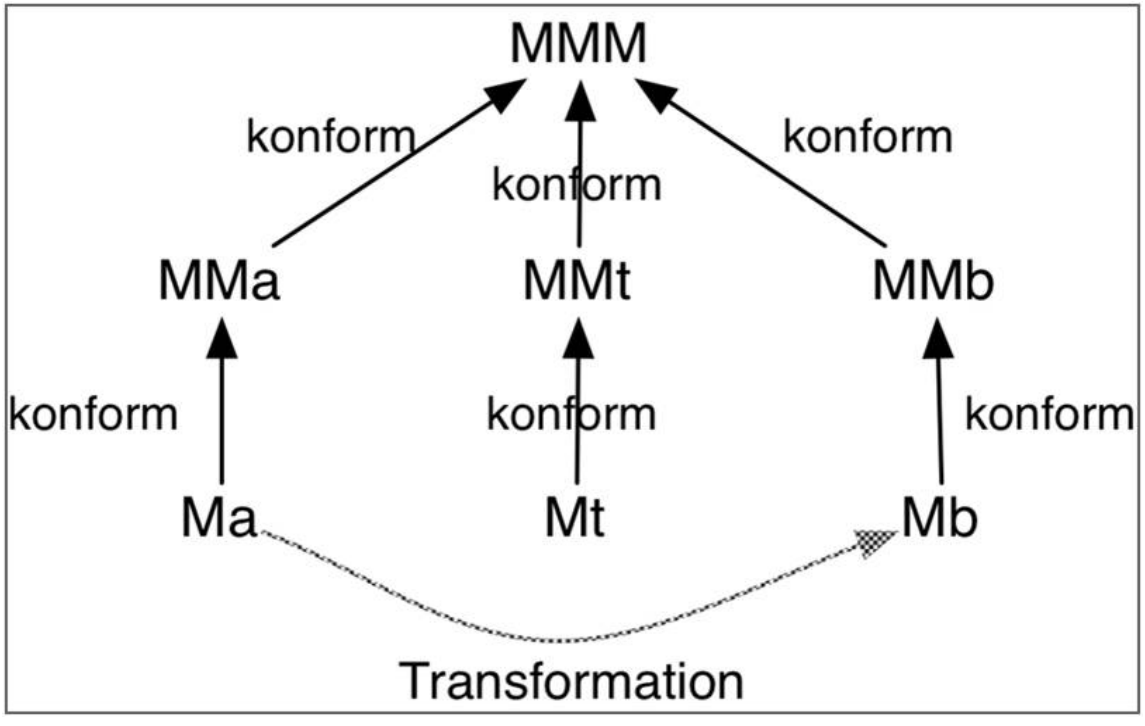
- **Modell-Modell** Transformation:
  - Bezeichnet Überführung der Inhalte eines Modells in anderes Modell.
  - Beide Modelle basieren auf verschiedenen Metamodellen.
- **Modell-Code** Transformation:
  - Bezeichnet Überführung der Inhalte eines Modells in Quellcode.

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



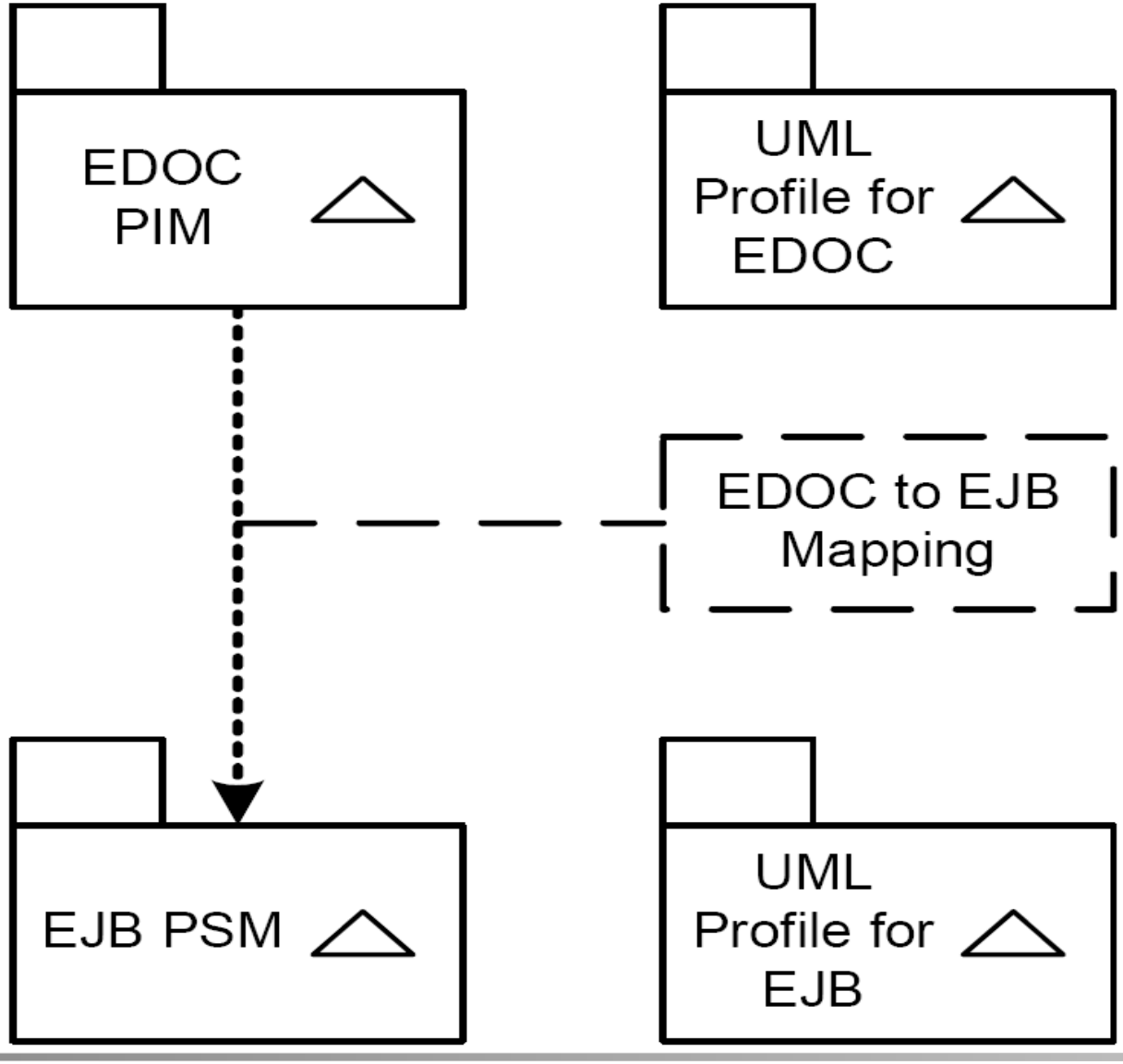
Modelltransformation vs. Metamodelle

- **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.

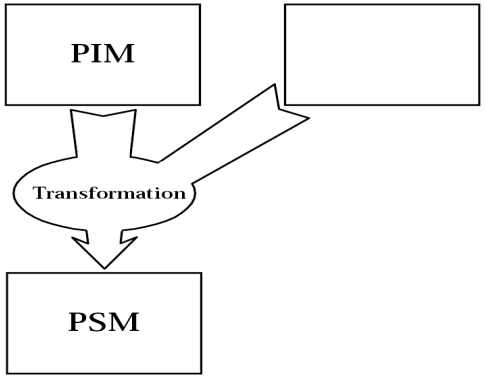


Modelltransformation PIM-PSM vs. Metamodelle: Beispiel

LEHRSTUHL 14
SOFTWARE ENGINEERING



Transformation
Beispiel PIM → PSM

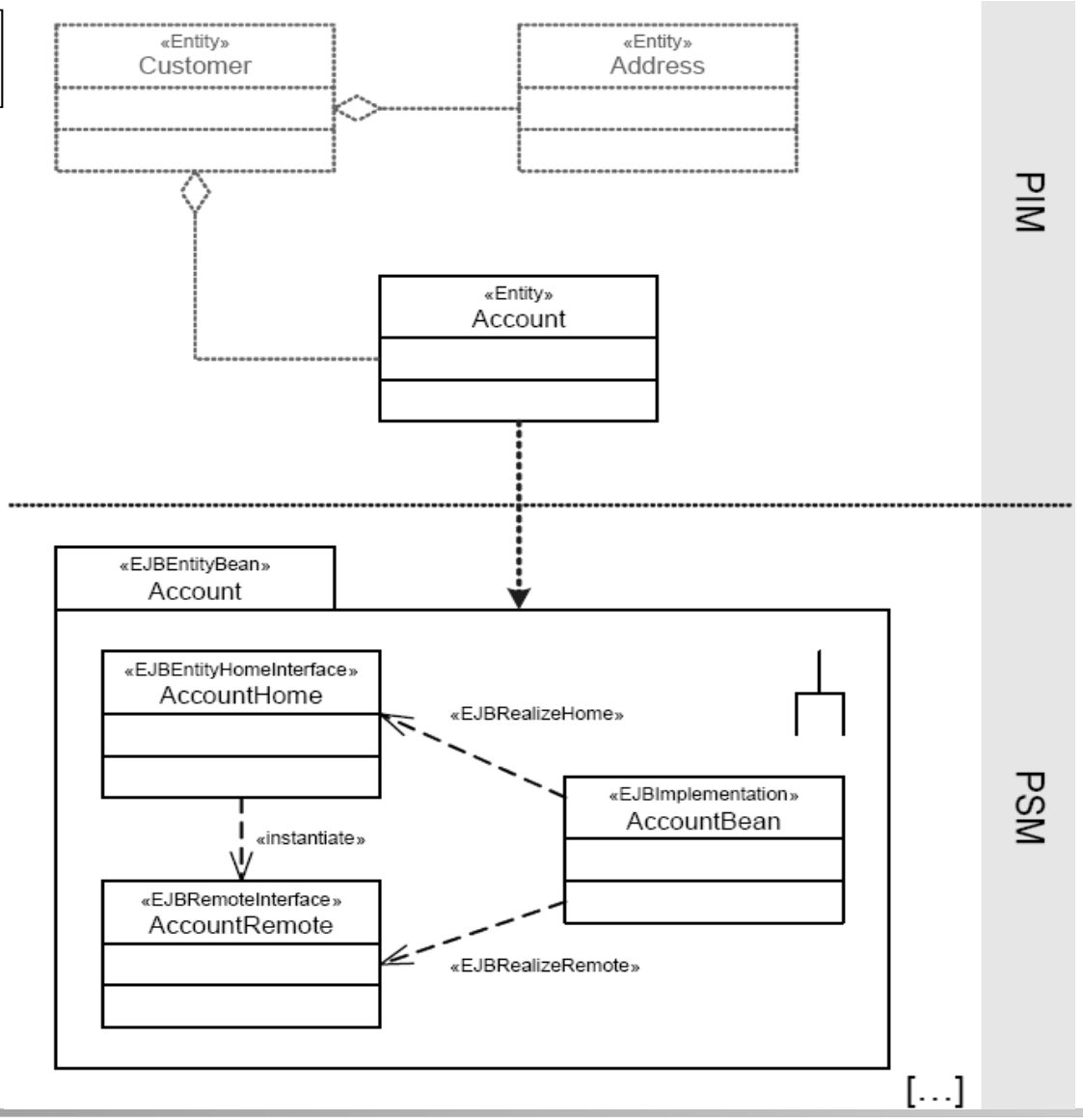


PIM auf **höherer Abstraktionsebene** angesiedelt.

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

- Führt zur Abbildung dieser Klasse in EntityBean des PSM.

NB: ⊓ steht für <<*subsystem*>> (= Komponente mit eigenständigem Verhalten)

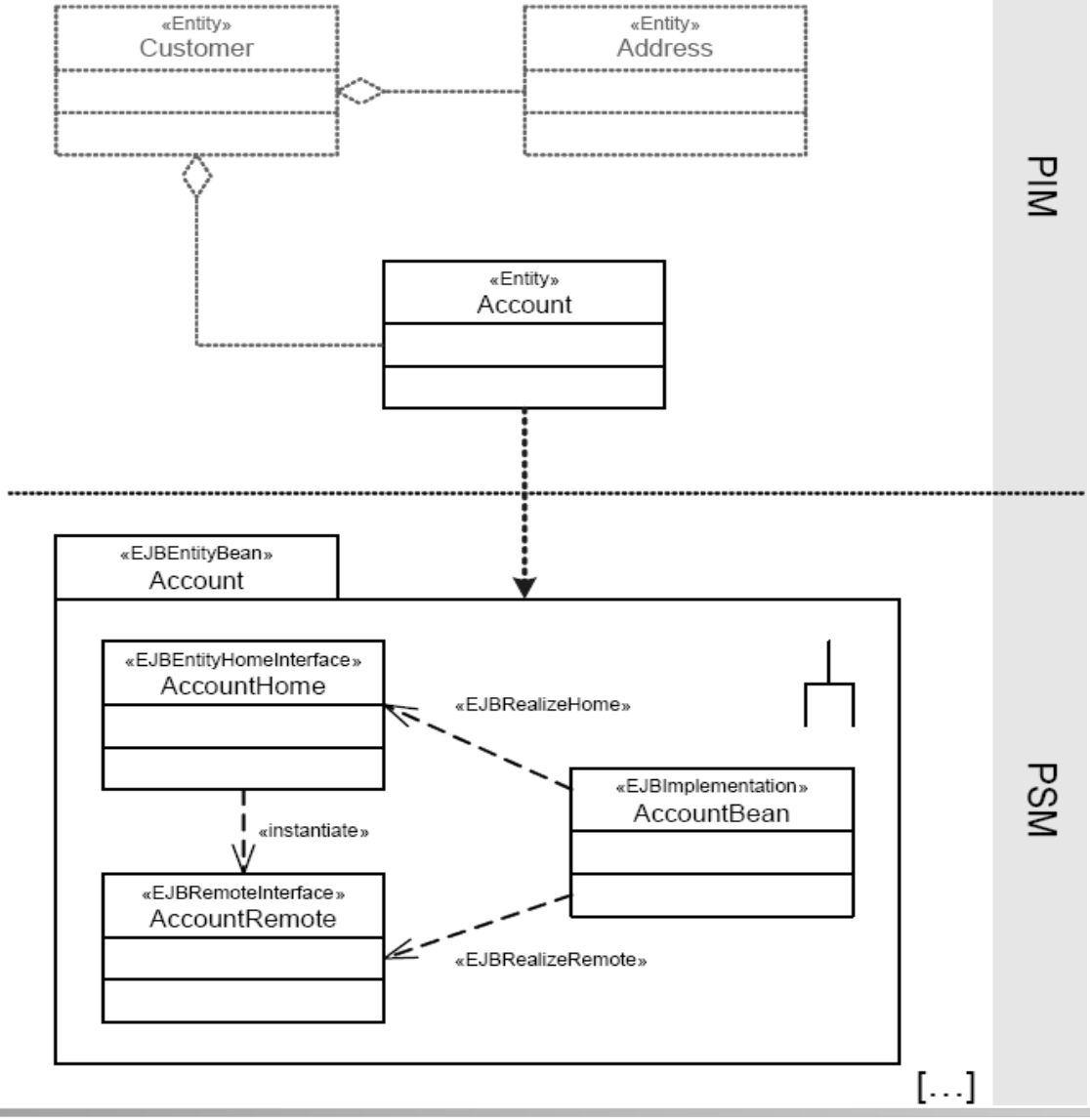


Object Management Group (OMG): *MDA Guide Version 1.0.1*.

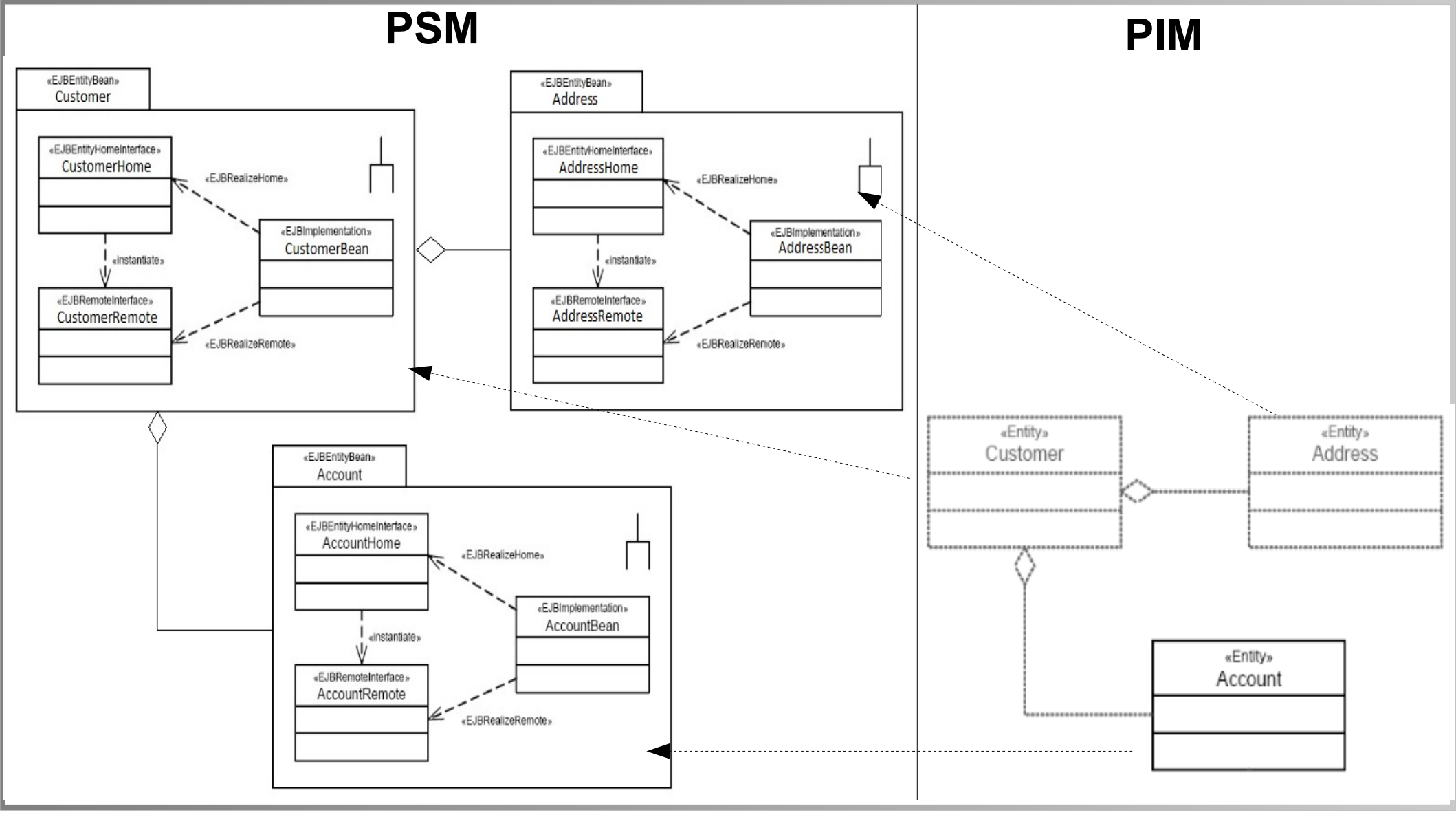
Gruhn, Pieper, Röttgers: MDA - Effektives Software-Engineering mit UML 2 und Eclipse, Springer-Verlag Berlin Heidelberg 2006 Abb. 2.09

Diskussionsfrage: Beispiel PIM → PSM

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



Diskussionsfrage: Beispiel PIM → PSM



Transformation Beispiel PSM → Code

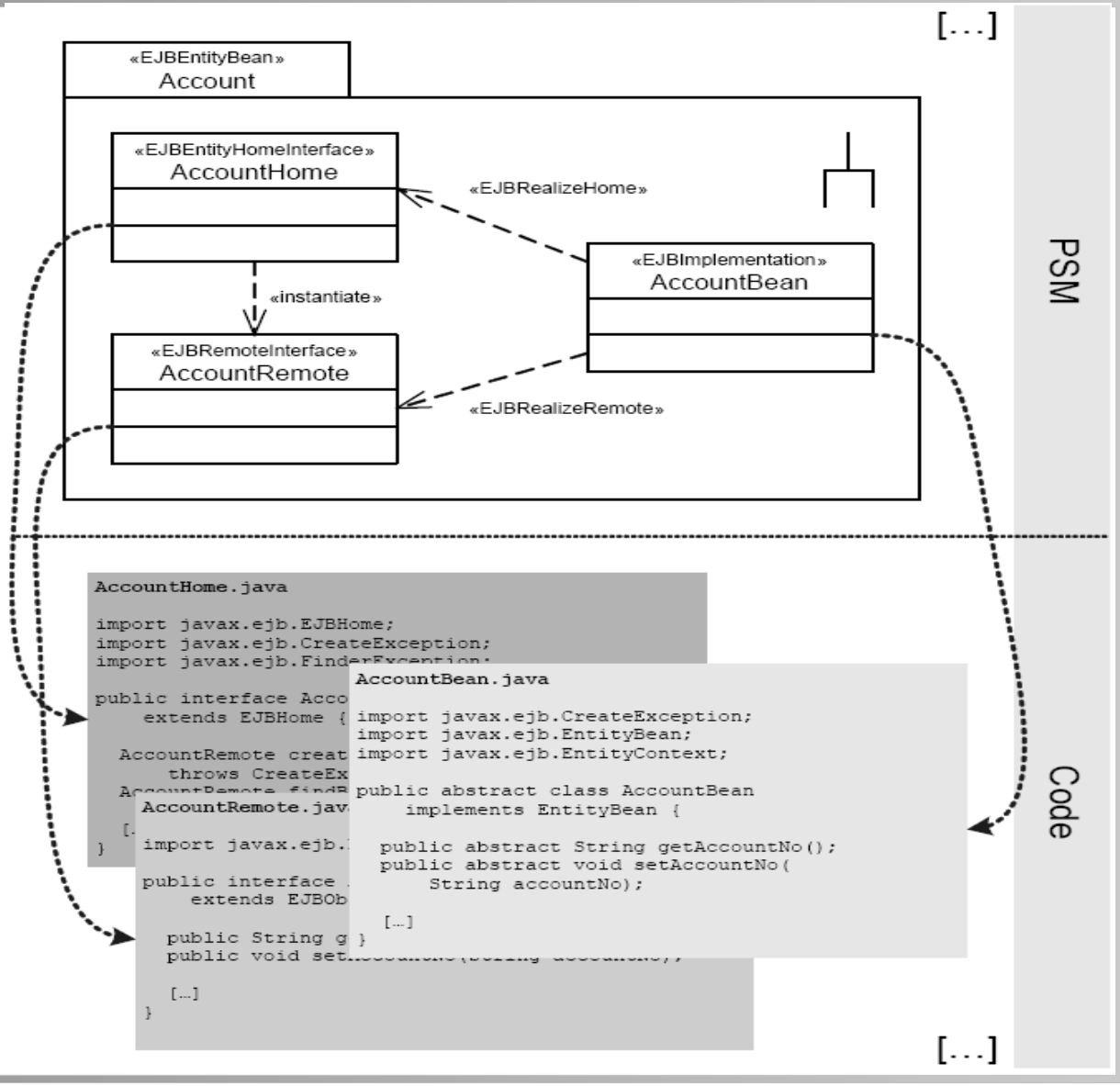
LEHRSTUHL 14
SOFTWARE ENGINEERING

Expandiert in **BeanClass** sowie **RemoteInterface** und HomeInterface:

- Wie Komponentenmodell verlangt.

Wegen Übersichtlichkeit nicht dargestellt (jedoch generiert):

- Operationen zum EJB-Lebens-zyklus (**ejbCreate, ejbActivate**, …).
- Deployment-Deskriptoren (ejb-jar.xml, ..., sowie hersteller-spezifische Deskriptoren).
- SQL-Skripte (z.B.: zum Aufsetzen des Datenbankschemas).
- Testskripte und ähnliches



Gruhn, Pieper, Röttgers: MDA - Effektives Software-Engineering mit UML 2 und Eclipse, Springer-Verlag Berlin Heidelberg 2006 Abb. 2.10

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.
  - Eine der ersten Implementierungen: ModelMorf.
- **QVT Operational Mapping:**
  - Imperativ.
  - Kann Relations verwenden.
  - Implementierung: Smart QVT.

Metamodelle **mittels MOF beschreiben.**

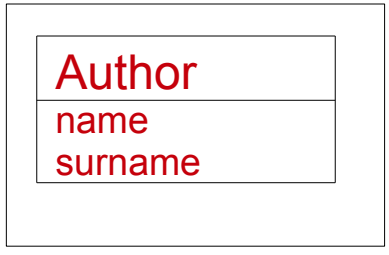
Atlas Transformation Language (ATL):

- **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.
- Arbeitet mit verschiedenen Arten von Metamodellen.

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
     )
   }
```

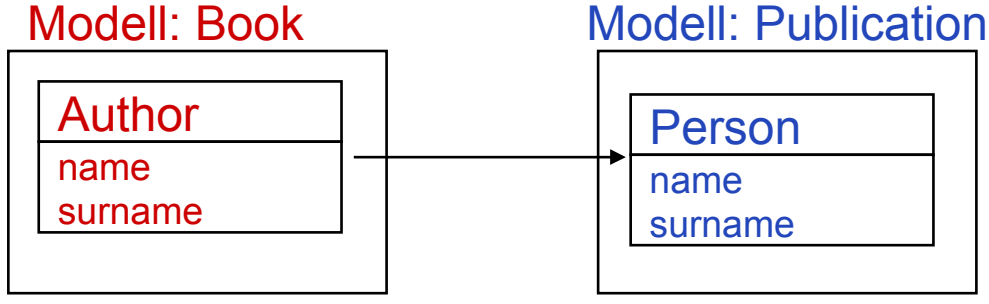
Modell: Book



- 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“.

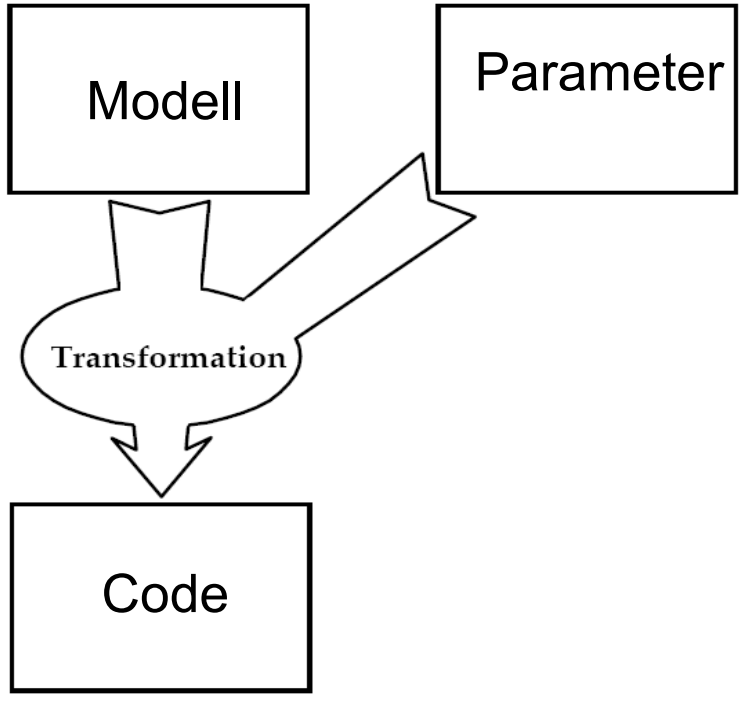
Tefkat:

- 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.**

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

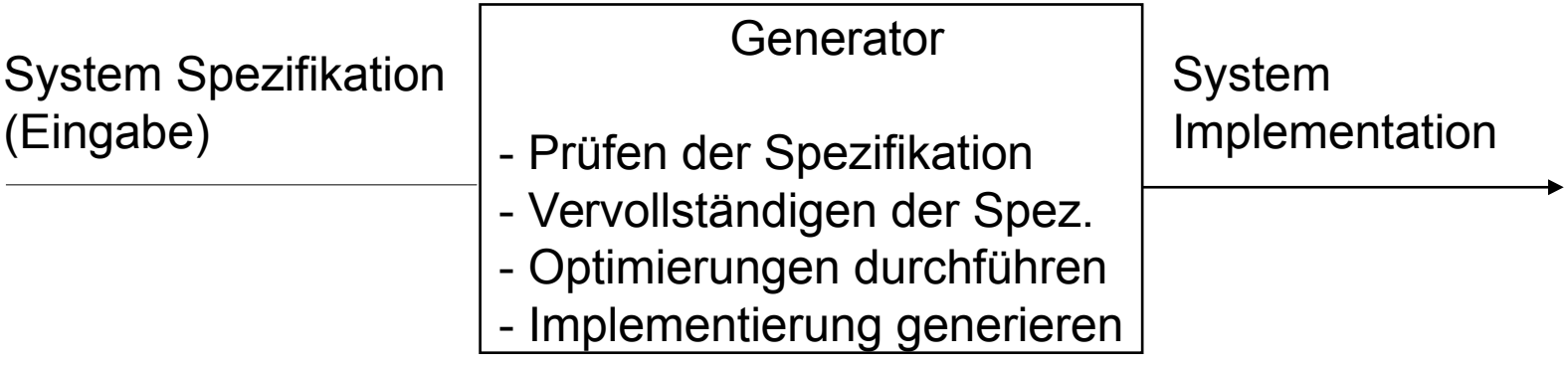


Codegenerierung
Ablauf und Phasen

- **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.

Codegenerierung
Generatoren und Software-Artefakt

- **Generatoren:**
  - Akzeptieren abstrakte Beschreibung eines Software-Artefakts als Eingabe.
  - Generieren dessen Implementation.
- **Software-Artefakt:**
  - Umfassendes Softwaresystem.
  - Komponente.
  - Klasse.
  - Methode / Prozedur.



Codegenerierung
Vorteile

- 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:
  - Technische Domänen: Hibernate, EJBs, Spring Beans, …
  - Architekturschichten: Persistenzschicht.
  - Fachliche Variationen.

Vorteil des Einsatzes:

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

Zusammenfassung und Ausblick

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

- MDA: Grundlagen und Konzepte
- Metamodellierung
- Modelltransformationen

Im nächsten Kapitel: Eclipse Modeling Foundation (EMF)

- Technische Grundlagen für UML-Werkzeuge und MDA.

Anhang
(Weitere Informationen und Beispiele)

Weiterführende Literatur

[BRJ01] The Unified Modeliung Language; Booch, Rumbaugh Jacobsen; Addison-Wesley; 2001

[Beziv01] Towards a Precise Definition of the OMG/MDA Framework. Bézivin, J. and O. Gerbé. ASE'01 - Automated Software Engineering. 2001. San Diego, USA.

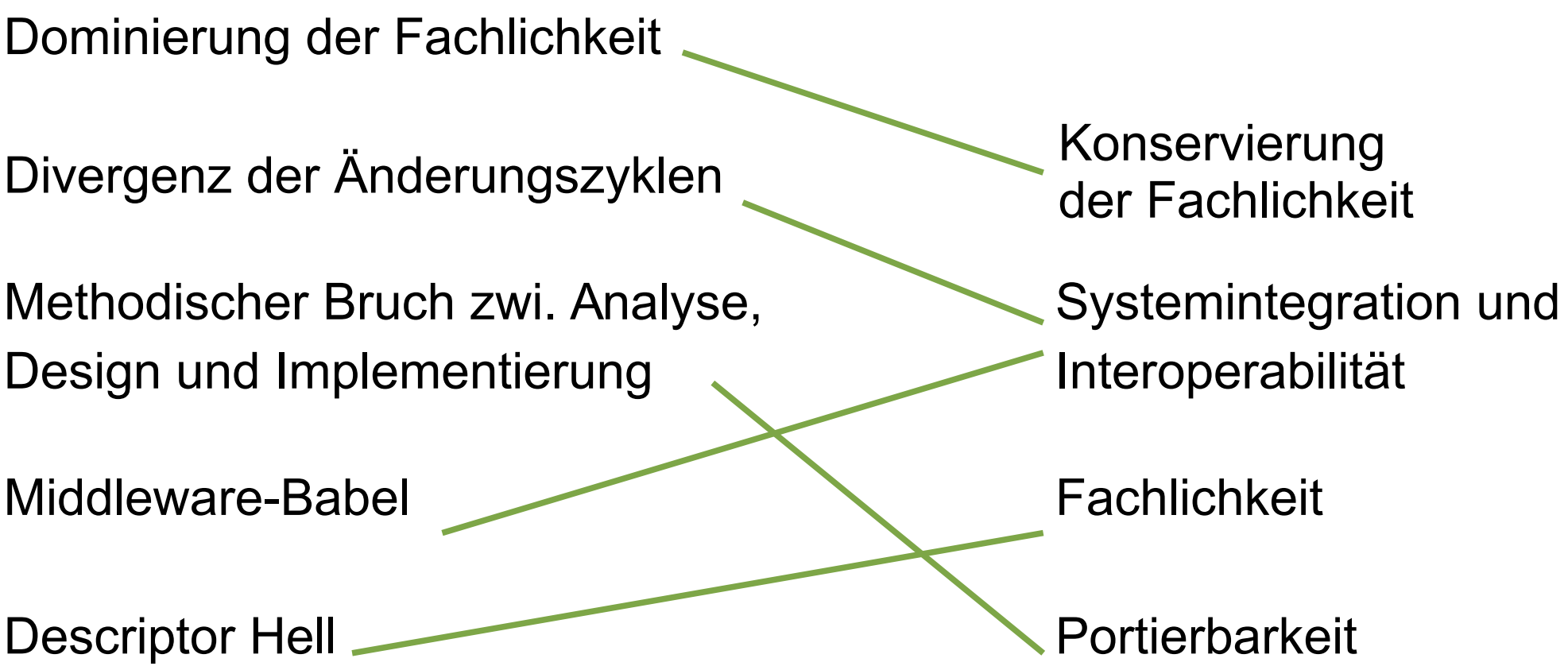
[BHK04] Softwareentwicklung mit UML2; M.Born, E. Holz, O.Kath; Addison-Wesely, 2004

[BBG05] Model-Driven Software Development; S.Beydeda, M.Book, V.Gruhn; Springer-Verlag 2005

[CKE00] Generative Programming: Methods, Tools, and Applications; K.Czarnecki, U.Eisenecker; Pearson 2000

Diskussionsfrage

- Welche **akuten Probleme** der Softwareentwicklung lassen sich mit den **Zielen der MDA** verbinden?



System:

- Aus Teilen **zusammengesetztes und strukturiertes Ganzes**.
- Hat Funktion und erfüllt Zweck.

Architecture:

- Organisation der Teile des Systems und deren Verhalten.
- **Beziehungen unter einzelnen Komponenten.**
  - Beziehungen an Bedingungen knüpfbar.
    - Erfüllung zur Erreichung des Systemzwecks.

Architectural Description:

- Besteht aus **Menge aller Artefakte:**
  - Notwendig zur Beschreibung einer Architektur.
- Typischerweise aus Modellen des Systems und unterstützender Dokumentation.

Terminologie IEEE1471
Concern und Model

Concern:

- **Spezifische Anforderungen** oder Bedingungen.

  → Erfüllen, um erfolgreiche Erfüllung des Systemzwecks zu gewährleisten.

- Als **Aspekt** übersetzt.

Model:

- Beschreibt oder **spezifiziert System** zu bestimmten Zweck.
- Erfasst **relevante Aspekte.**

  → Struktur, Verhalten, Funktion und Umwelt des Systems.

Viewpoint:

- Beschreiben Elemente und Konstrukte.

  → Ermöglichen **Erstellung des Modells** zur Beschreibung von Systemaspekten.
- **Regeln, Techniken bzw. Methoden** sowie Notationsmittel zur Erstellung eines Modells.

View:

- Kohärente **Menge von Modellen:**
  - Beschreibt bestimmte Aspekte eines Systems.
  - Verbirgt irrelevante Aspekte.
- Als Sicht auf System bezeichnet.
- Mit **Regeln und Notation** des entsprechenden Viewpoints dargestellt.

Metamodel:

- Modelle mit Menge von Elementen erstellbar.
- Enthält:
  - **Regeln** bei Erstellung des Modells beachten (abstrakte Syntax).
  - Bedeutung der Elemente und Elementkonstellationen (Semantik).

Profile:

- **Leichtgewichtige Erweiterung** eines Metamodells.
- Stellt **neue Modellelemente** zur Verfügung oder erweitert Semantik oder Syntax bestehender Elemente.
- **Verschärfen Bedingungen**, die an Elemente der Modelle gestellt werden.
  - Regeln eines **Design-by-Contracts** [DBC] unterordnen.

Zentrale MDA-Begrifflichkeiten
Domain und Application

Domain:

- Abgrenzbares, kohärentes Wissensgebiet.
- Im Metamodell: Konzepte einer Domäne in Bezug gesetzt und beschrieben.
- **Domänenspezifische Sprache:** Sprachdefinierender Charakter, Metamodell und Profil.

Application:

- Zu erstellendes Stück Software, umfasst **zu entwickelnde Funktionalität.**
- System aus einer oder mehreren Anwendungen zusammensetzbar.
  → Auf einer oder mehreren Plattformen ausführbar.

Plattform:

- **Ausführungsumgebung** für Anwendung.
- Zugriff auf Funktionalität über Schnittstellen. → Kenntnis über Implementierung nicht nötig.
- Bsp.: Plattformen im Bereich Betriebssysteme (Unix, Windows, OS/390).
- Bsp.: Programmiersprachen (C++, Java, C#).
- Bsp.: Middleware (CORBA, Java EE, .NET).
- Setzen in Art eines **Plattform-Stacks** aufeinander auf.
- Hardware eines Computers bildet Plattform für Betriebssystem.
- **Betriebssystem:** Plattform für einzelne Anwendungen usw.

Computation Independent Model (CIM):

- Liefert Sicht auf Gesamtsystem unabhängig von Implementierung.
- Modell im Vokabular seiner Domäne beschrieben.
- **Betont Anforderungen an System** und seine Umwelt.
- Synonyme: Business Model, Domain Model.

Platform Independent Model (PIM):

- Beschreibt **formale Struktur und Funktionalität** eines Systems.
- Unabhängig von implementierungstechnischen Details. → Abstrahiert von zugrunde liegenden Plattform.

Platform Specific Model (PSM):

- **PIM mit plattformabhängigen Informationen**.
- Zur Implementierung: Falls es alle Informationen zur Konstruktion und Betrieb eines Systems liefert.

  → Modelle auch direkt ausführbar (executable models).

- **Infrastructure:**
  - Beschreibt Meta-Metamodell der UML.
  - Liefert Elemente zur Definition der UML-Diagrammtypen.
- **Superstructure (= UML Metamodell):**
  - Definiert alle verfügbaren Diagrammtypen sowie Sonderkonstrukte (Profile, Templates,…).
  - Für jedes Diagramm Elemente des Metaobjektes spezialisierbar.
- Beide **erreichbar** auf Webseite der OMG unter:

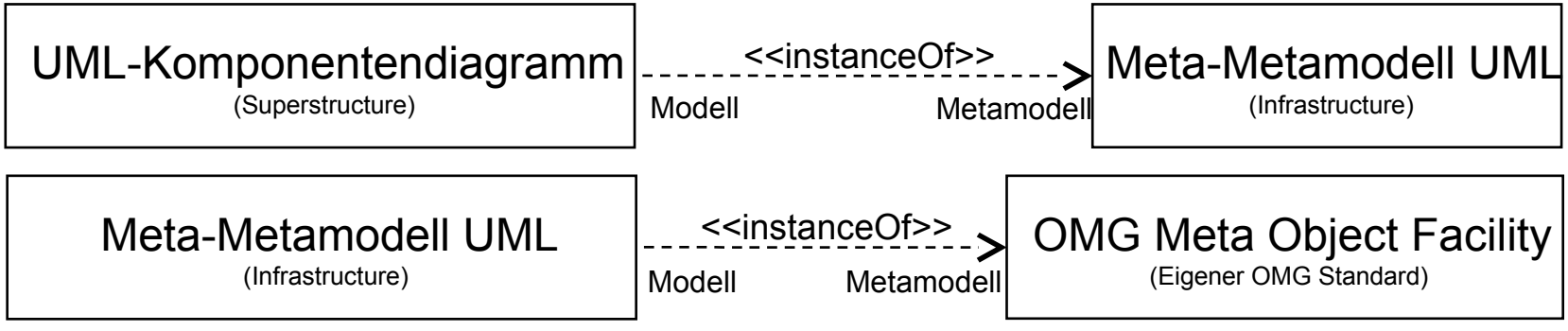
  http://www.omg.org/technology/documents/formal/uml.htm

Meta Ebenen: Definition

- **M0-Ebene:**
  - Konkret. Ausgeprägte Daten.
- **M1-Ebene:**
  - **Modelle.** Z.B. physikalische/logische Daten- /Prozessmodelle oder konkrete Ausprägungen von UML- bzw. Objekt-Modellen, welche Daten der M0-Ebene definieren.
- **M2-Ebene:**
  - **Meta-Modelle.** Definieren, wie Modelle aufgebaut und strukturiert sind. Z.B. definieren Sprachelemente wie Klassen, Assoziationen und Attribute der UML 2.0, wie konkrete UML-Modelle aufgebaut sein können.
- **M3-Ebene:**
  - **Meta-Meta-Modelle** (MOF-Ebene). Abstrakte Ebene, die zur Definition der M2-Ebene herangezogen wird. Definition der M3-Ebene erfolgt mit Mitteln der M3-Ebene. → Stellt Abschluss einer unendlichen Metaisierung dar.

Modell und Metamodell der UML

- **UML Infrastructure:** Definiert Modell, von dem alle UML-Diagrammtypen eine Instanz bilden.
- UML: Sich **selbst beschreibende Sprache.**
  - UML Metamodell als Klassendiagramm beschrieben.
- Grundlage des Aufbaus des UML Metamodells: **Meta Object Facility.**
  - Einstufung der Modelle in **vier Ebenen.**



Vorgaben der OMG

Konkrete Vorgehensanweisungen in MDA-Projekten vage.

Eher **Konzept-Charakter**, als Spezifikation.

OMG beschreibt einige **Basiskonzepte**[1][2].

- Im Folgenden die Wichtigsten.
- Vorab **Begriffsbestimmungen:**
  - Terminologie IEEE 1471.
  - Begriffe mit zentraler Rolle innerhalb MDA.

1 Object Management Group (OMG): *Model Driven Architecture – A Technical Perspective*. http://www.omg.org/cgi-bin/apps/doc?ormsc/01-07-01 (letzter Abruf Mai 2005)
2 Object Management Group (OMG): *MDA Guide Version 1.0.1*. http://www.omg.org/cgi-bin/doc?omg/03-06-01 (letzter Abruf Mai 2005).