



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

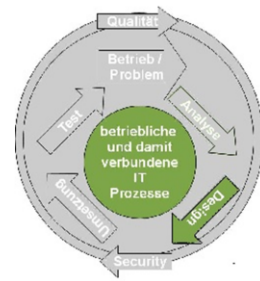
- Einführung
- **Modellbasierte Softwareentwicklung**
- OCL
- Ereignisgesteuerte Prozesskette (EPK)
- Petrinetze
- EMF

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

3



1.1 Modellbasierte Softwareentwicklung

Metamodellierung

UML-Erweiterungen

Modelltransformation

Design Patterns

4

Literatur:

V. Gruhn: **MDA - Effektives Software-Engineering**

<http://www.ub.tu-dortmund.de/katalog/titel/1223129>

Kapitel 2



UML-Modelle: bislang

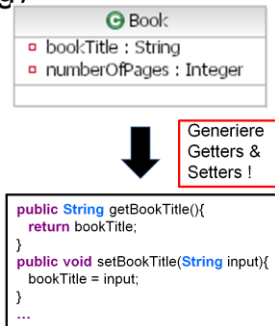
- **informell definiert** (textuelle Beschreibung)
- und **verwendet** (Skizzen auf Papier).

Aber: **UML kann mehr !**

- Codegenerierung
- Testfallgenerierung
- Modellvalidierung
- Modelltransformation
- ...

Werkzeuge müssen UML „verstehen“ können.

→ **Formelle Definition** der Notation.



5



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 ?

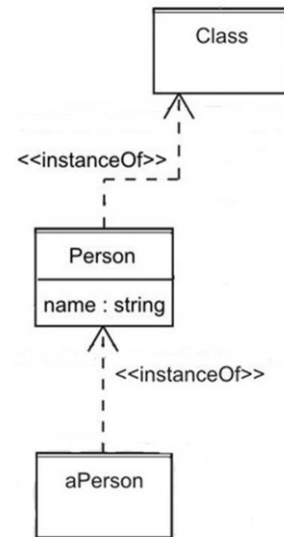


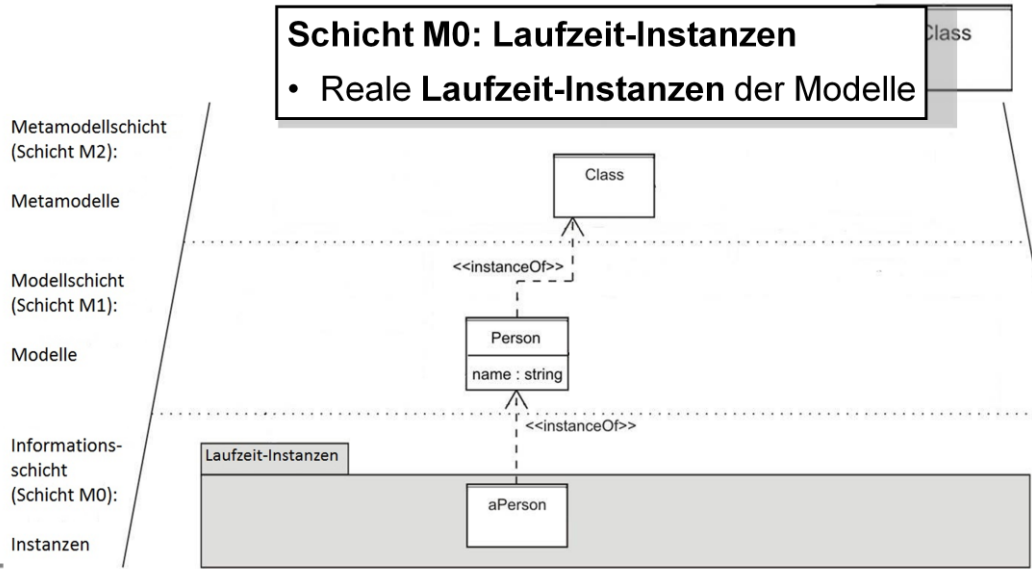
6

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*).

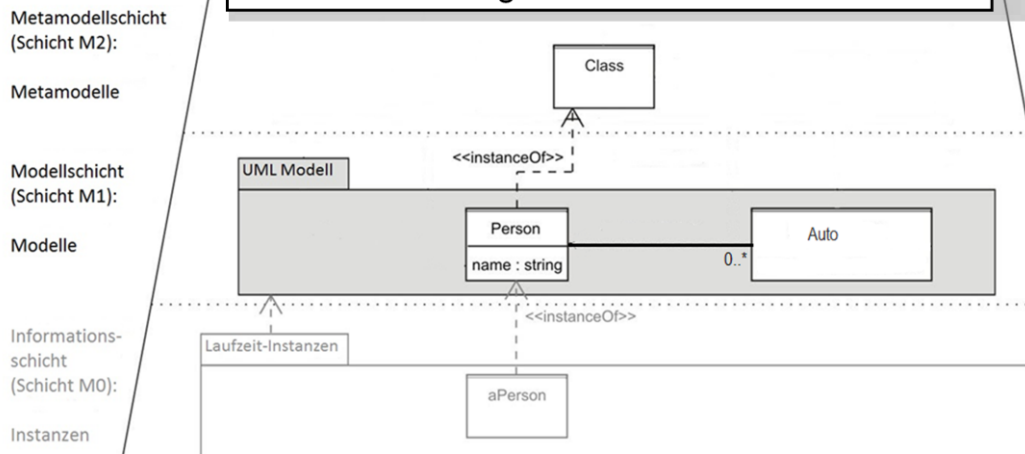




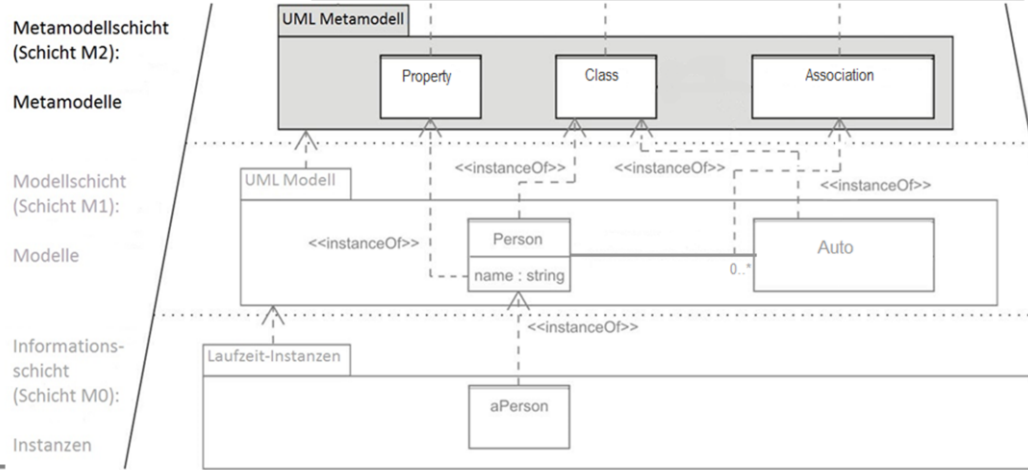


Schicht M1: UML-Modell

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



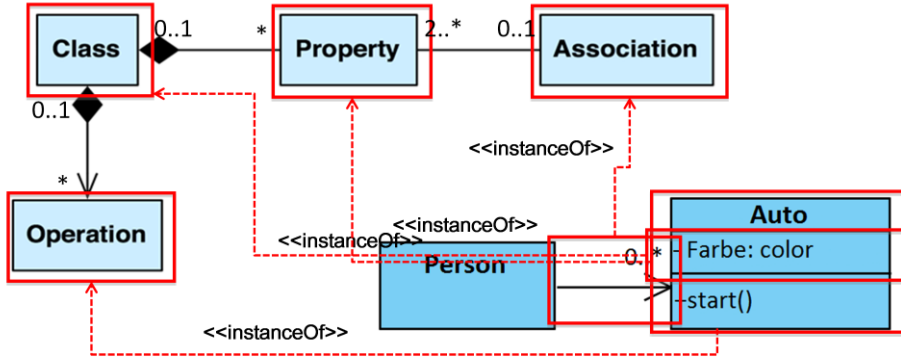
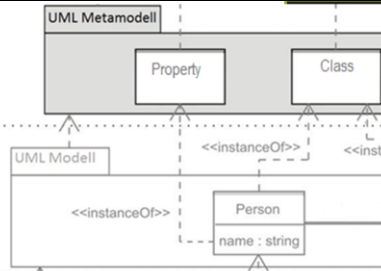
- Definiert UML-Notation, d.h. Menge valider UML-Modelle.
- Insbesondere darin verwendete Konzepte wie *Klassen*, *Attribute*, *Assoziationen*.



Beispiel

Klassendiagramm-Metamodell (vereinfacht)

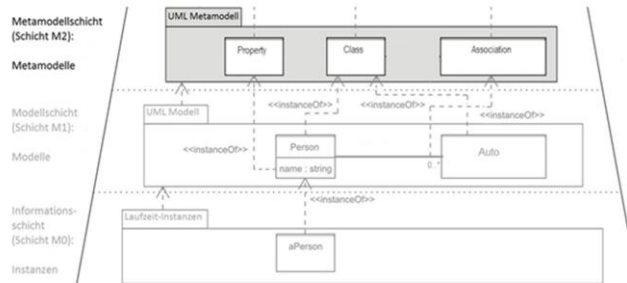
Metamodellschicht (Schicht M2):
Metamodelle
Modellschicht (Schicht M1):
Modelle



OK, Metamodelle sind cool – sind wir jetzt fertig ?

Könnten hier aufhören:

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





Aber: UML ist nicht alles !

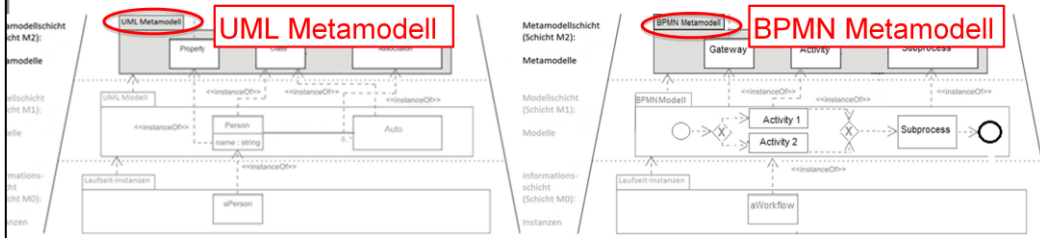
- Weitere Notationen: **eigene Metamodelle** (BPMN; Domain-Specific Languages (DSLs), ...).

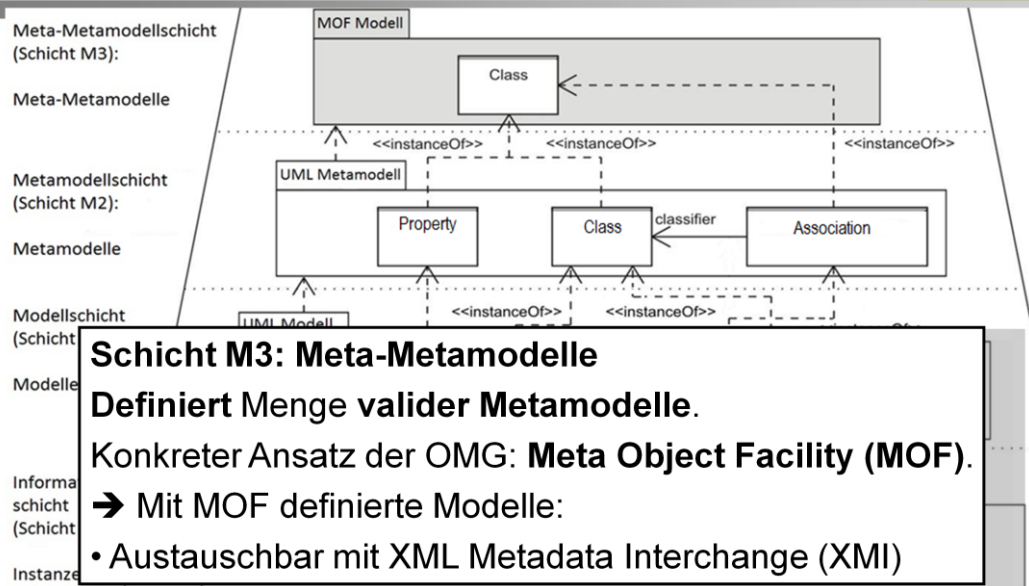


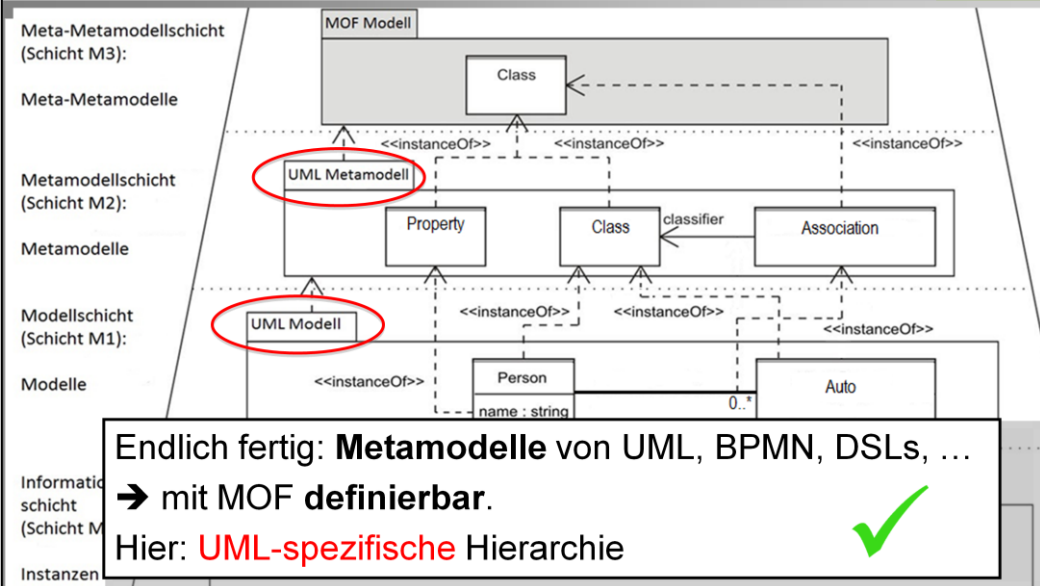
Gewünscht:

- **Wiederverwendbarkeit** der **Werkzeuge**
 - **Austauschbarkeit** von **Modellen**
- ➔ Allgemeiner Ansatz zum **Metamodelle definieren**.

Wie ?









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

M3-Ebene („MOF“)

Definiert bekannte Diagrammarten
(wie Klassen- / Aktivitätsdiagramm).

Definiert UML-Metamodell.

XMI

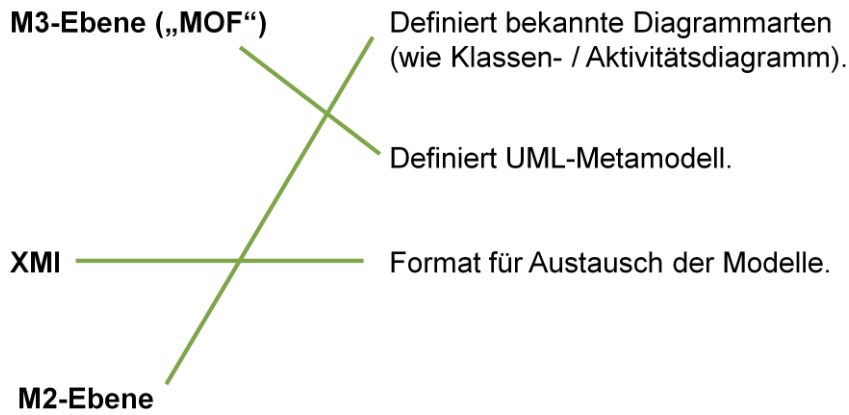
Format für Austausch der Modelle.

M2-Ebene

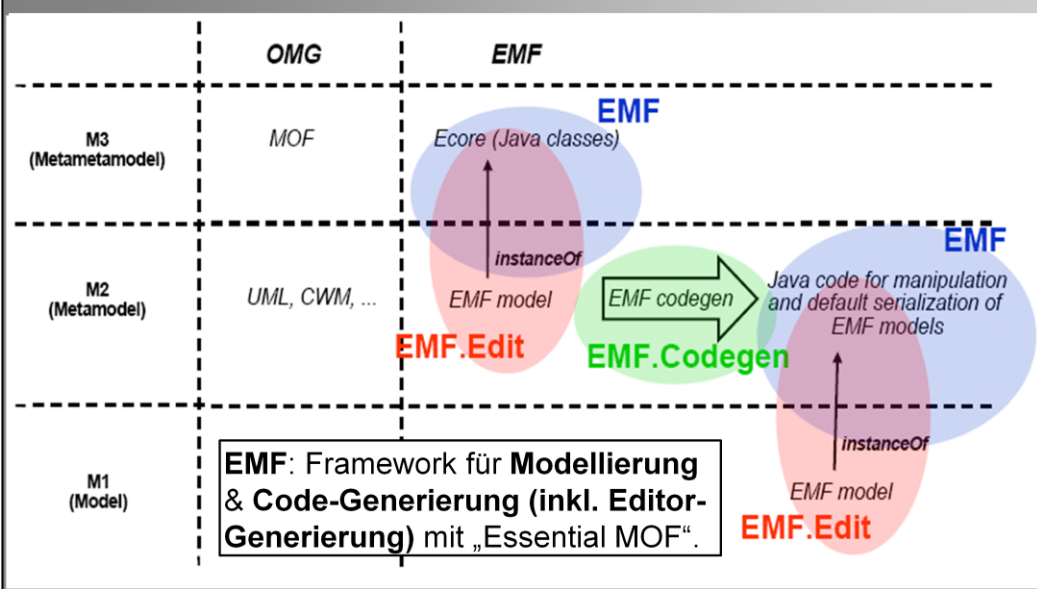
16

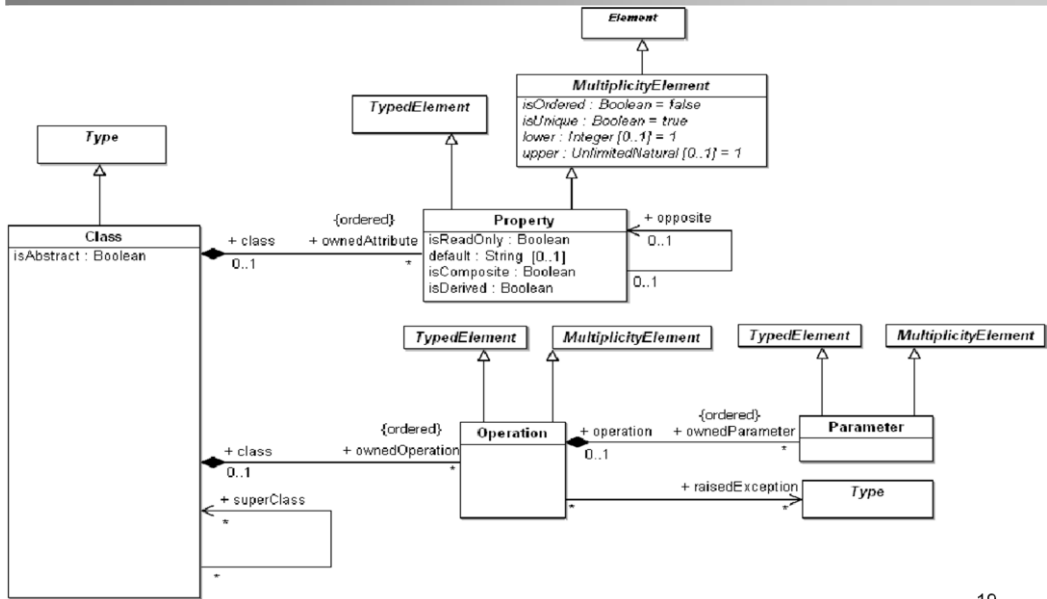


Welche **Aussagen** passen zu den **Kernelementen der UML** ?



17

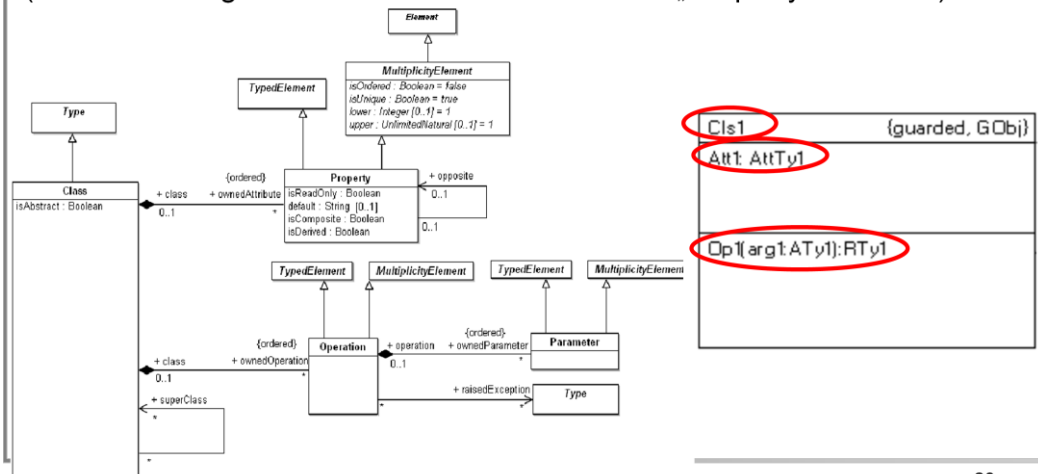






Wo finden sich die **rot markierten Elemente** aus dem **Klassendiagramm** im **Metamodell** wieder ?

(Zur Erinnerung: Attribut in UML-Metamodell als „Property“ definiert.)

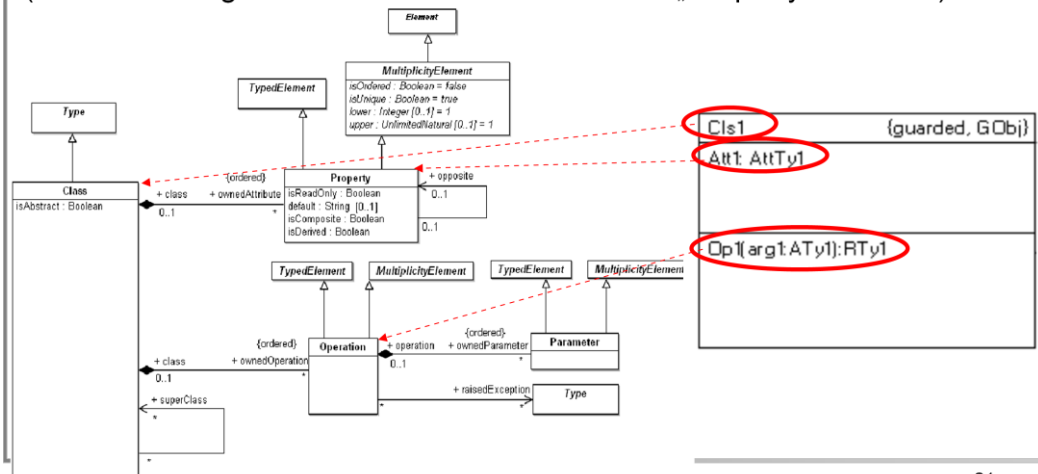


20

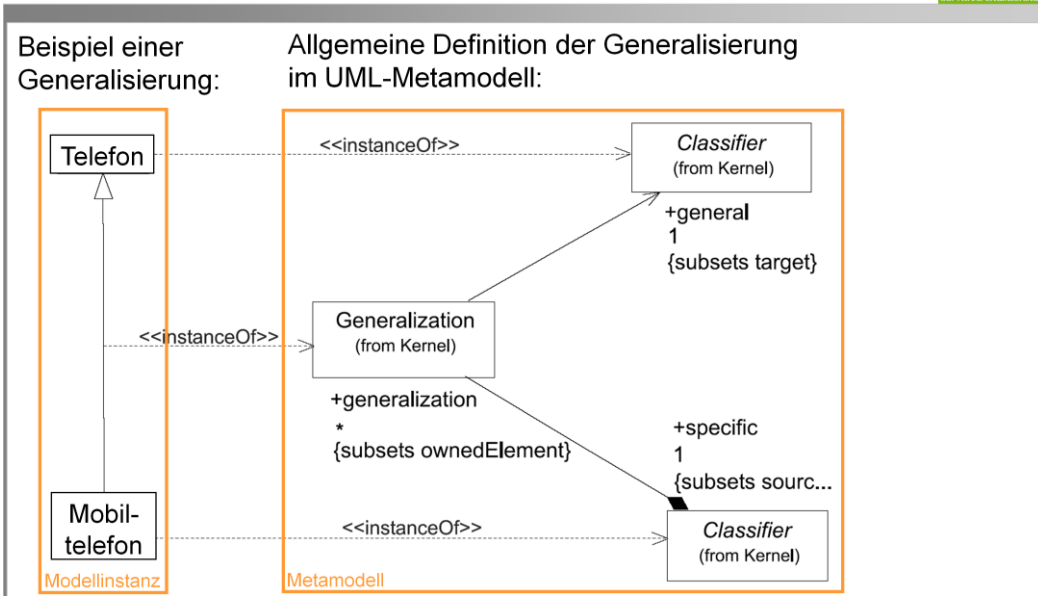


Wo finden sich die **rot markierten Elemente** aus dem **Klassendiagramm** im **Metamodell** wieder ?

(Zur Erinnerung: Attribut in UML-Metamodell als „Property“ definiert.)



21



22

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

Literatur:

V. Gruhn: **MDA - Effektives Software-Engineering**

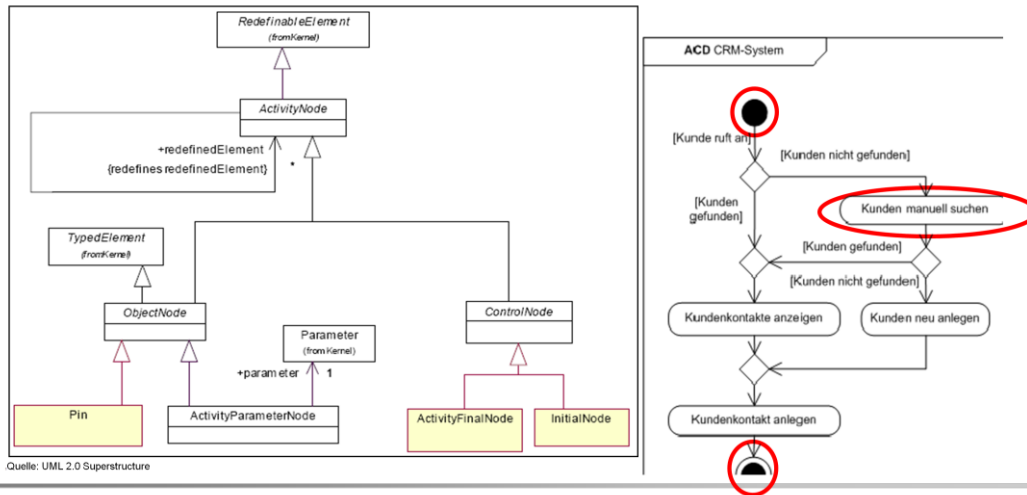
<http://www.ub.tu-dortmund.de/katalog/titel/1223129>

„Abschnitt 3.4 (S.98-101)

„Abbildung 3.20 (S.99)

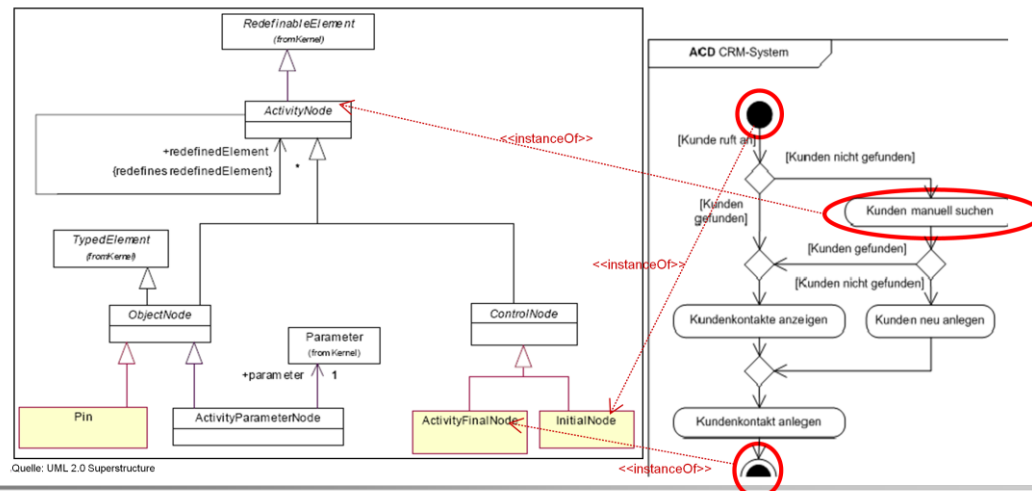


Wo finden sich die **rot markierten Elemente** aus dem **Aktivitätsdiagramm** im **Metamodell** wieder ?



23

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



Quelle: UML 2.0 Superstructure



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 im Allgemeinen:

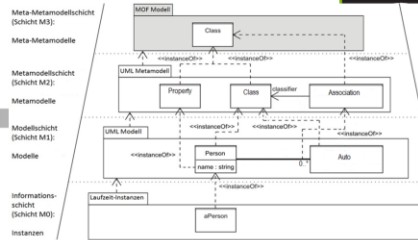
- Erstellung Metamodell:
Benötigt **Aufwand** und **Expertise**.



Nachteile von MOF:

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

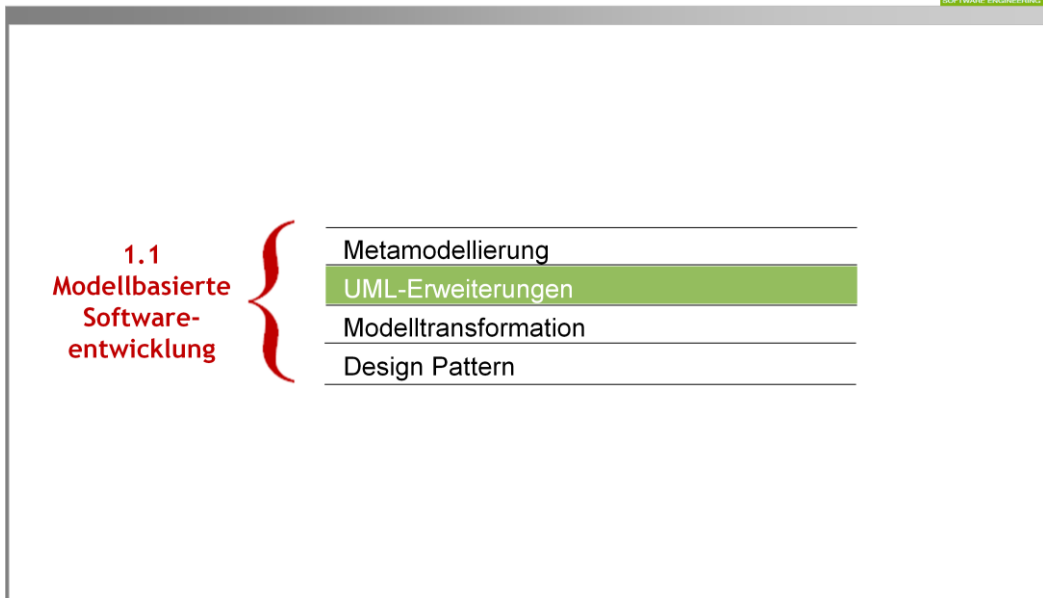
Zusammenfassung: Metamodellierung



Einführung Metamodellierung:

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





Literatur:

V. Gruhn: **MDA - Effektives Software-Engineering**

<http://www.ub.tu-dortmund.de/katalog/titel/1223129>

Kapitel 2



- **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.

29

Literatur:

V. Gruhn: **MDA - Effektives Software-Engineering**

<http://www.ub.tu-dortmund.de/katalog/titel/1223129>

Abschnitt 3.3.4 – UML-Profile (S.94-98)



- 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 !

30

Literatur:

V. Gruhn: **MDA - Effektives Software-Engineering**

<http://www.ub.tu-dortmund.de/katalog/titel/1223129>

Abschnitt 3.3.4 – UML-Profile (S.94-98)



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.

Literatur:

V. Gruhn: **MDA - Effektives Software-Engineering**

<http://www.ub.tu-dortmund.de/katalog/titel/1223129>

Abschnitt 3.3.4 – UML-Profile (S.94-98)



UML-Profil:

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

Definition eines Profils:

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

32

Literatur:

V. Gruhn: **MDA - Effektives Software-Engineering** <http://www.ub.tu-dortmund.de/katalog/titel/1223129>

Abschnitt 3.3.4 – UML-Profile (S.94-98)



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.

33

Literatur:

V. Gruhn: **MDA - Effektives Software-Engineering**

<http://www.ub.tu-dortmund.de/katalog/titel/1223129>

Abschnitt 3.3.4 (S.94-98)



Definition von Stereotypen:

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

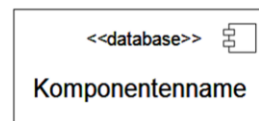
Definition des Stereotyp-
Begriffs (**M3-Ebene**)



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



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



34

Literatur:

V. Gruhn: **MDA - Effektives Software-Engineering**

<http://www.ub.tu-dortmund.de/katalog/titel/1223129>

Abschnitt 3.3.4 (S.94-98)



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.

35

Literatur:

V. Gruhn: **MDA - Effektives Software-Engineering**

<http://www.ub.tu-dortmund.de/katalog/titel/1223129>

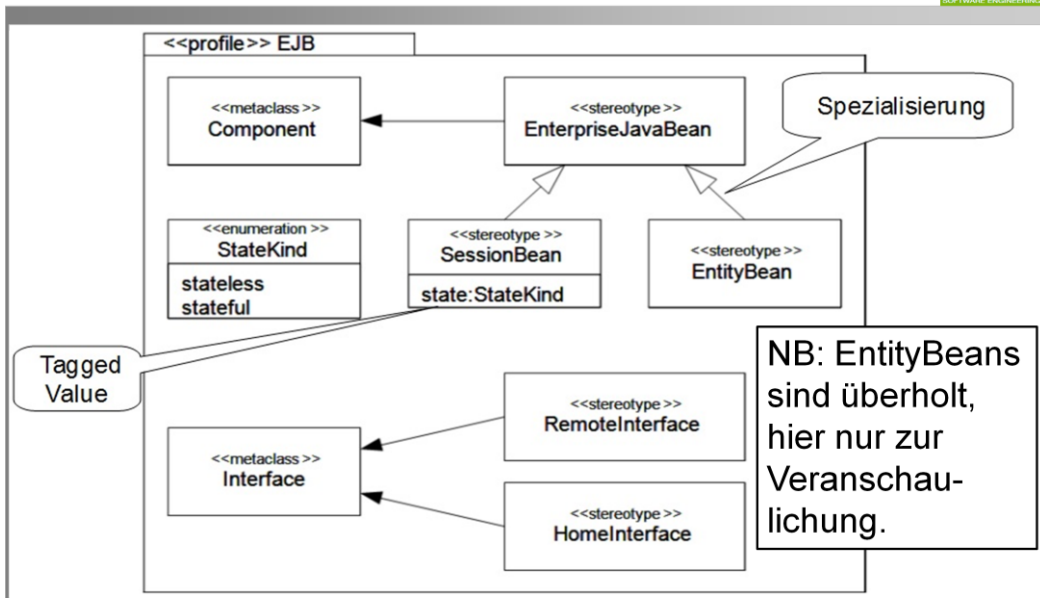
Abschnitt 3.3.4 (S.94-98)



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

36



37

Literatur:

V. Gruhn: **MDA - Effektives Software-Engineering**

<http://www.ub.tu-dortmund.de/katalog/titel/1223129>

Abschnitt 3.3.4 (S.94-98)

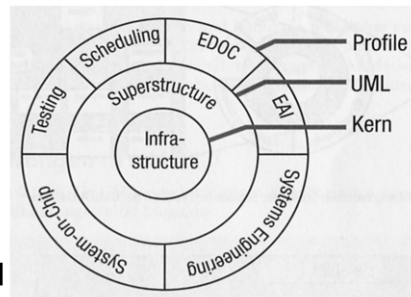
http://en.wikipedia.org/wiki/Entity_Bean :

In EJB 3.0, entity beans were superseded by the Java Persistence API (which was subsequently completely separated to its own spec as of EJB 3.1). Entity Beans have been marked as a candidate for pruning as of Java EE 6[1][2] and are therefore considered a deprecated technology.

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).

38

Literatur:

V. Gruhn: **MDA - Effektives Software-Engineering**

<http://www.ub.tu-dortmund.de/katalog/titel/1223129>

Abschnitt 3.3.4 (S.94-98)

Semantik:

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

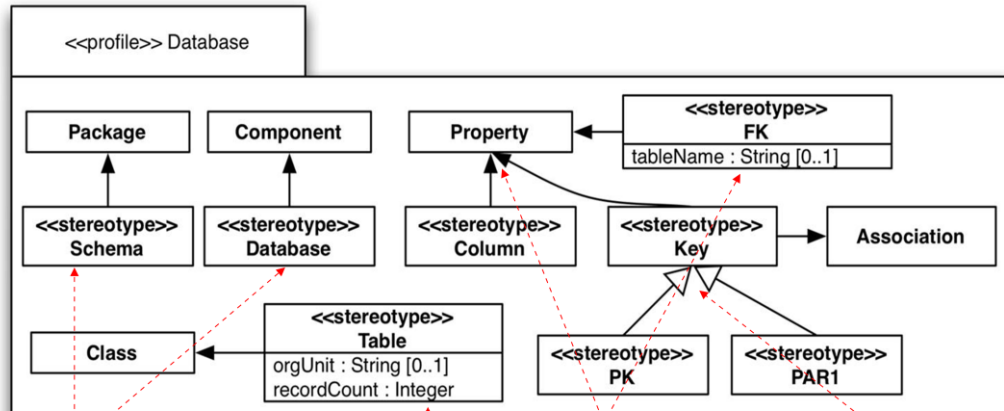
**Grundlegende Konstrukte und ihre Darstellung:**

- **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>>.

39



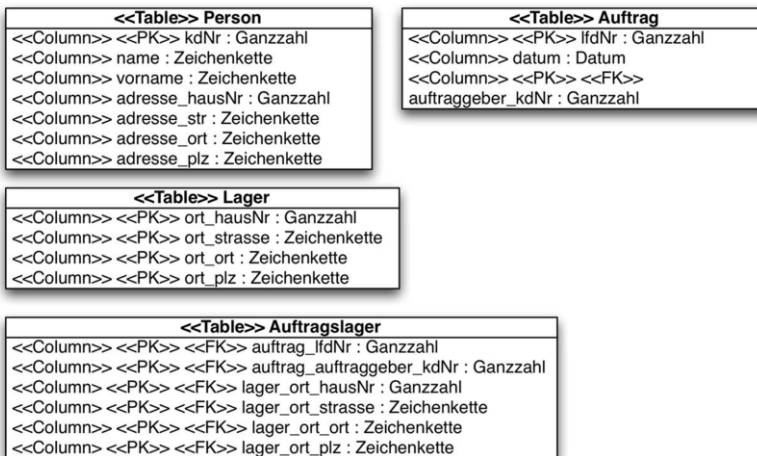
Zu welchen Metamodellelementen gehören die Stereotyp-Definitionen ?



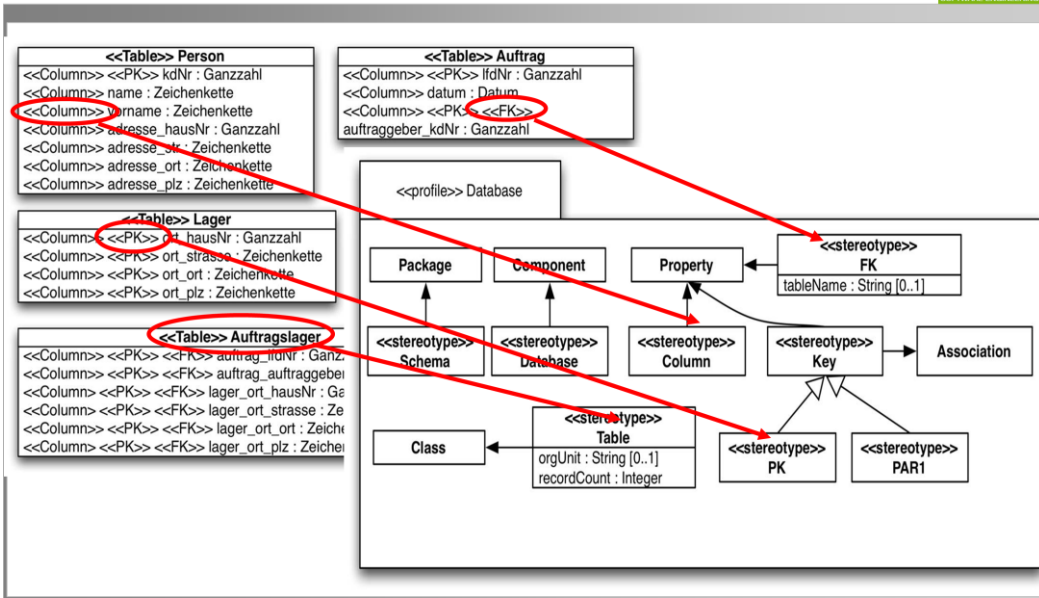
- **Datenbank:** UML-Komponente mit Stereotyp <<Database>>.
- **Schema:** UML-Paket mit Stereotyp <<Schema>>.
- **Tabelle:** UML-Klasse mit Stereotyp <<Table>>.
- **Spalten:** Attribute (Property) der als <<Table>> markierten Klasse; mit Stereotyp <<Column>> versehen.
- **Primärschlüssel:** Attribut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

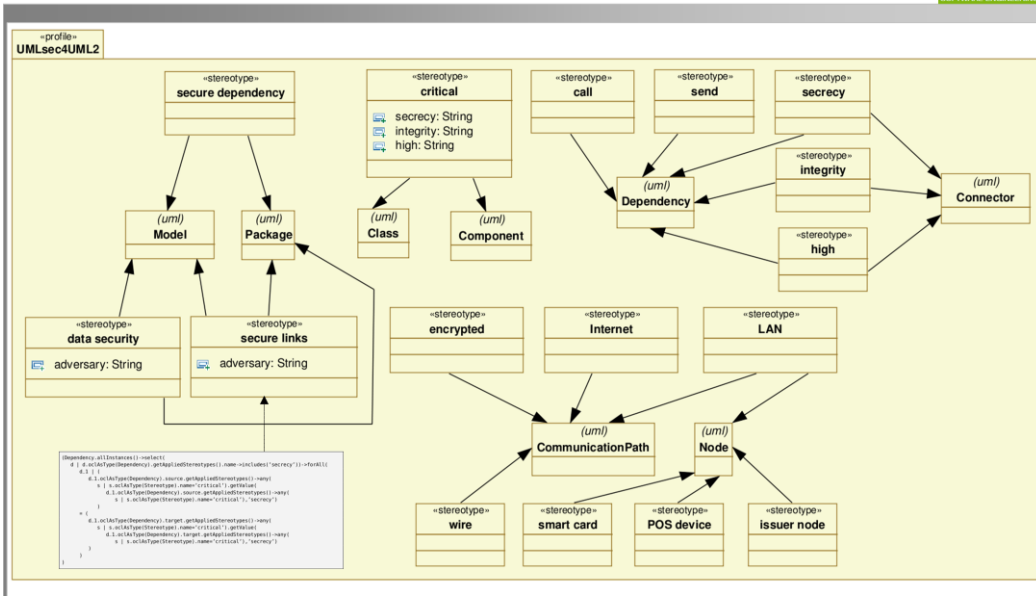
Zusammenfassung:

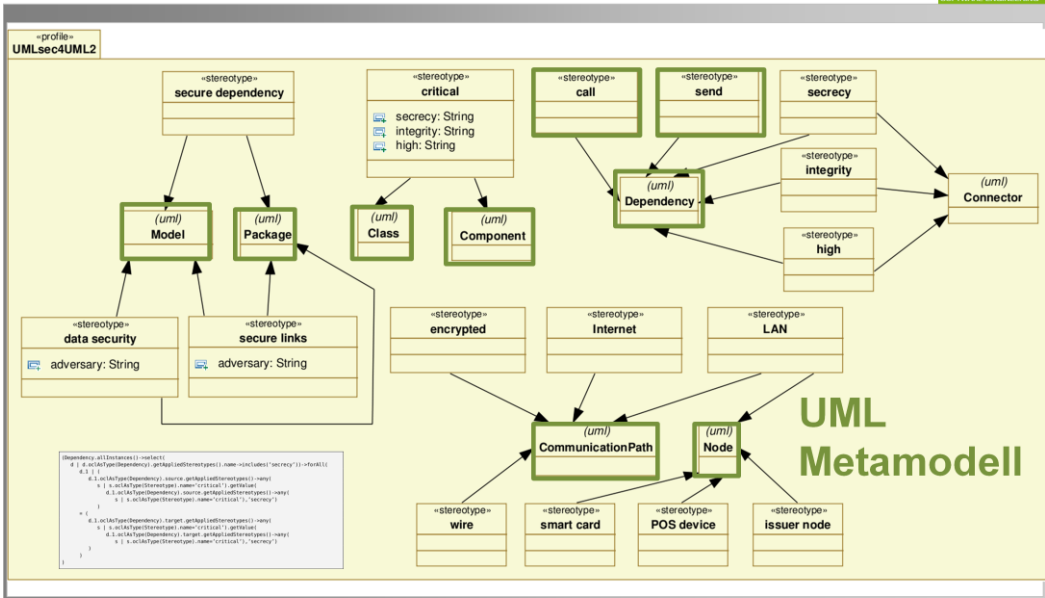
- **Import der geerbten UML-Metamodellklassen** „Package“, „Component“, „Class“, „Property“ und „Association“ aus deren Paketen „Classes“ und „Components“.
- **Beziehungen mit ausgefüllten Spitzen:** „Extensions“. → Verlaufen von Stereotyp zur Metaklasse im ursprünglichen Metamodell.
- Stereotype **optional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

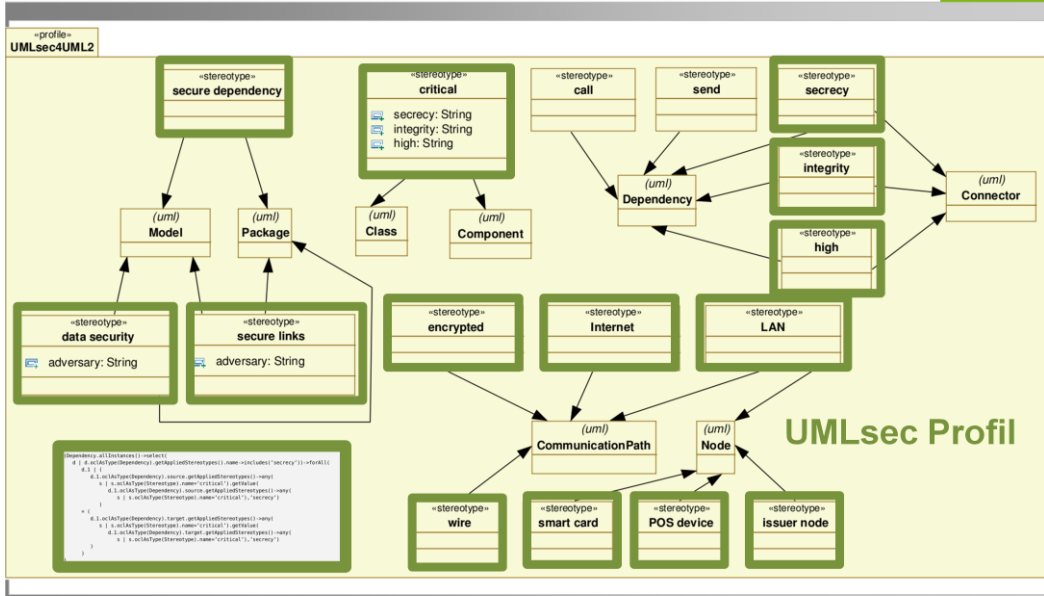


Beispiel für technisches Datenmodell mittels definiertem UML-Profil.







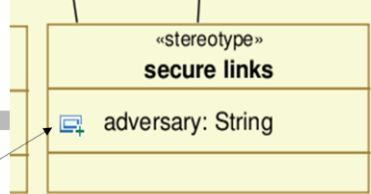


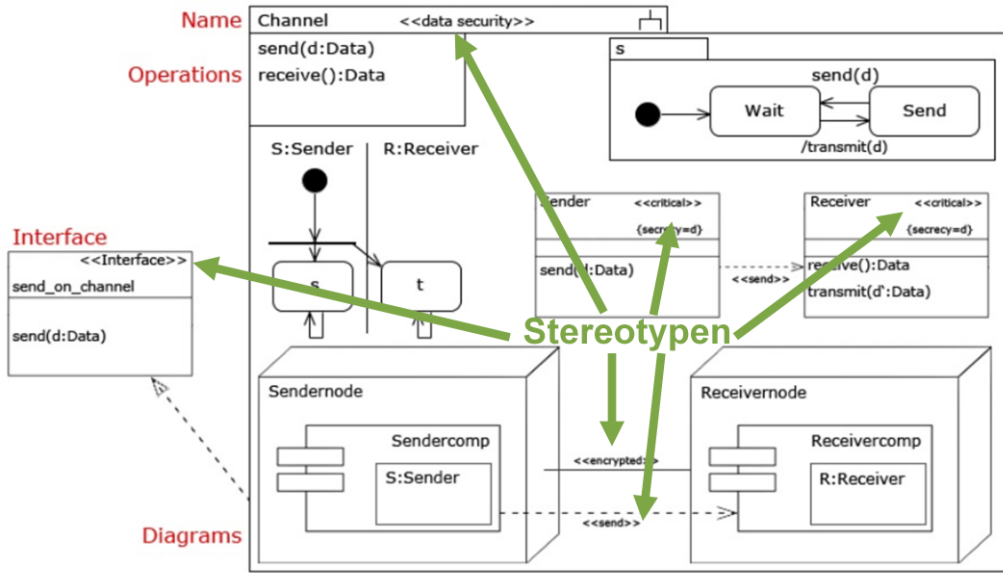
45

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

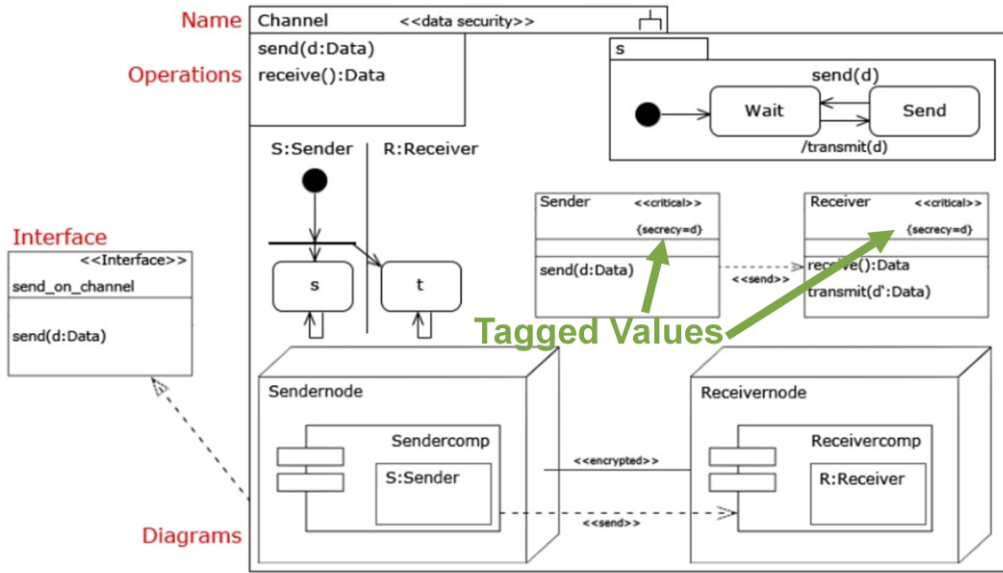
(Mehr Einzelheiten zu OCL in Abschnitt 1.2.)

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

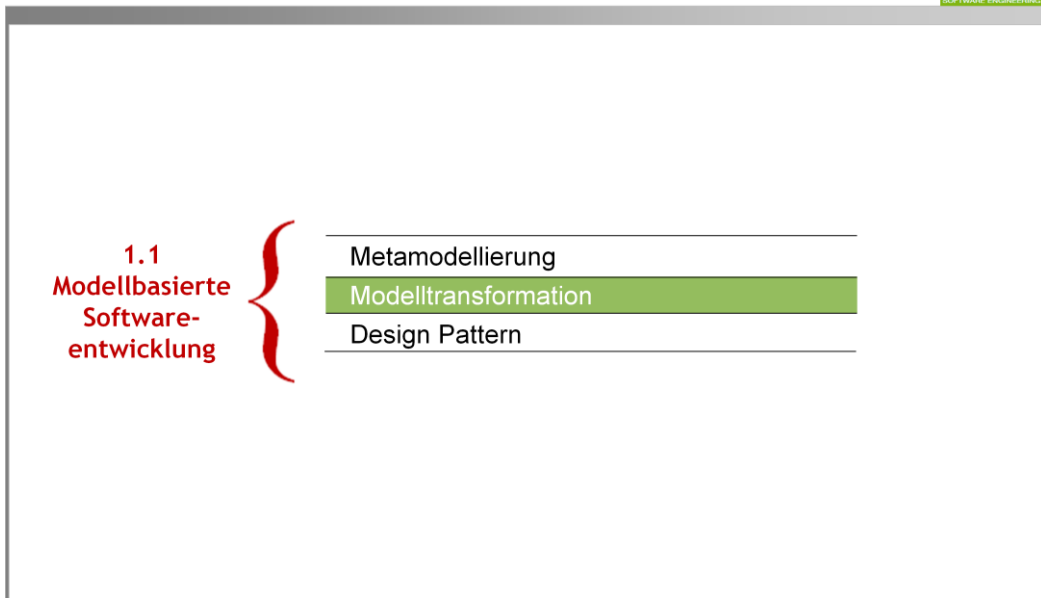




47



48



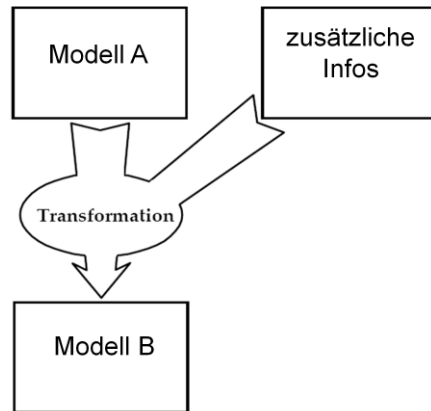
Literatur:

V. Gruhn: **MDA - Effektives Software-Engineering**

<http://www.ub.tu-dortmund.de/katalog/titel/1223129>

Kapitel 2

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



50

- **Hintereinanderausführung mehrerer Transformationen** möglich.
- **Spezifikation für Transformation** in Mapping Rules festhalten.
- **Mapping Rules** in Mapping gebündelt.

Literatur:

V. Gruhn: **MDA - Effektives Software-Engineering**

<http://www.ub.tu-dortmund.de/katalog/titel/1223129>

Abschnitt 5.1 (S.149-151)

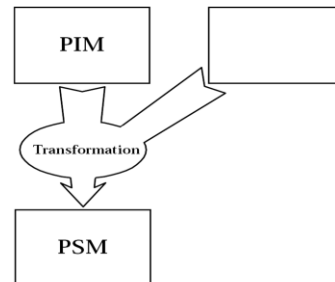
Quelle der Abbildung: MDA Guide Version 1.0.1

Horizontale Transformation:

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

Vertikale Transformation:

- Umwandlung eines Modells 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.



51

Literatur:

V. Gruhn: **MDA - Effektives Software-Engineering**

<http://www.ub.tu-dortmund.de/katalog/titel/1223129>

Abschnitt 5.2 (S.151-153)

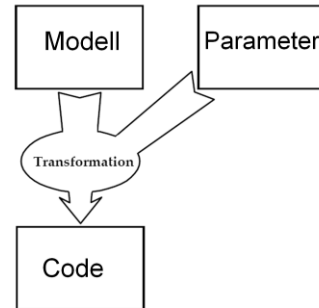
Modell-Modell Transformation:

- 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.



52

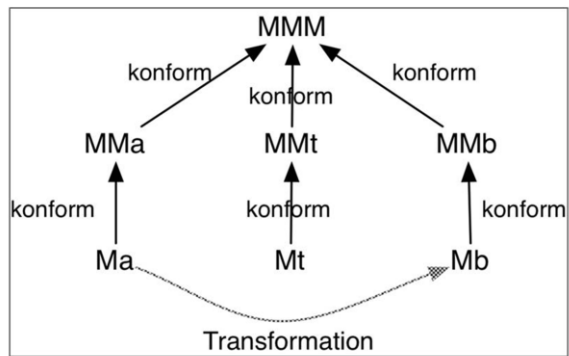
Literatur:

V. Gruhn: **MDA - Effektives Software-Engineering**

<http://www.ub.tu-dortmund.de/katalog/titel/1223129>

Abschnitt 5.2 (S.151-153)

- **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.



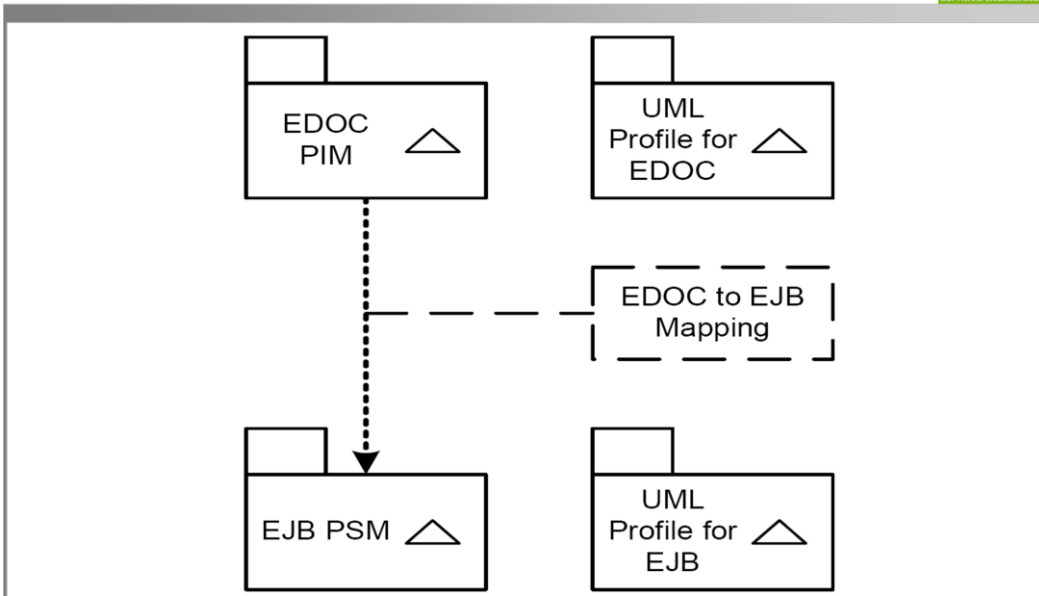
53

Literatur:

V. Gruhn: **MDA - Effektives Software-Engineering**

<http://www.ub.tu-dortmund.de/katalog/titel/1223129>

Abschnitt 5.2 (S.151-153)



54

Literatur:

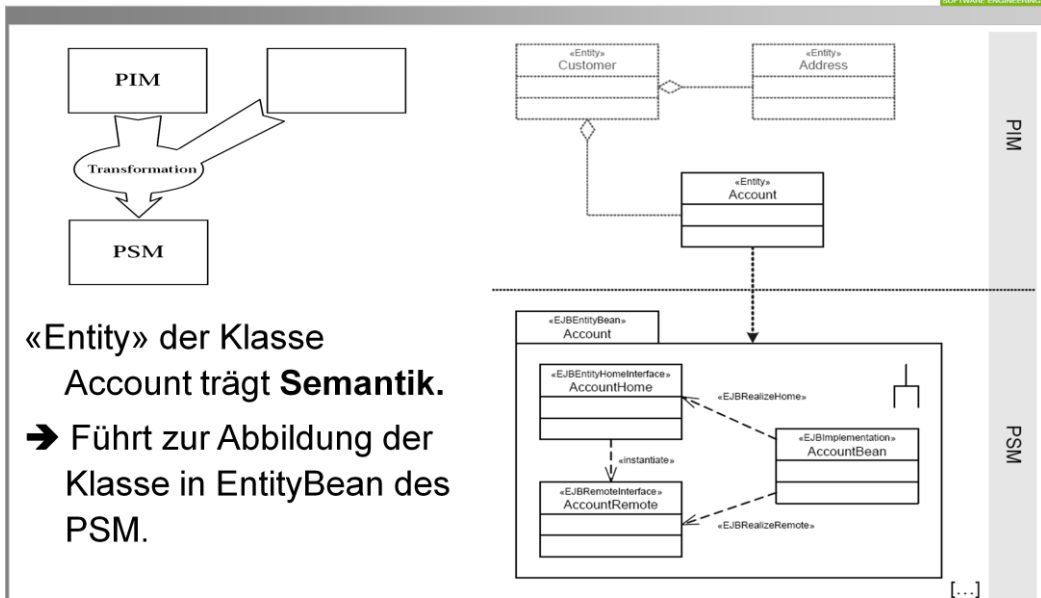
V. Gruhn: **MDA - Effektives Software-Engineering**

<http://www.ub.tu-dortmund.de/katalog/titel/1223129>

·Abschnitt 2.2.3 (S.25-31)

·Abbildung 2.8 (S.28)

·Abschnitt 5.5 (S.178-180)



55

Literatur:

V. Gruhn: **MDA - Effektives Software-Engineering**

<http://www.ub.tu-dortmund.de/katalog/titel/1223129>

Abschnitt 2.2.3 (S.25-31)

Abbildung 2.7 (S.27)

Abbildung 2.8 (S.28)

Abbildung 2.9 (S.29)

Abschnitt 5.5 (S.178-180)

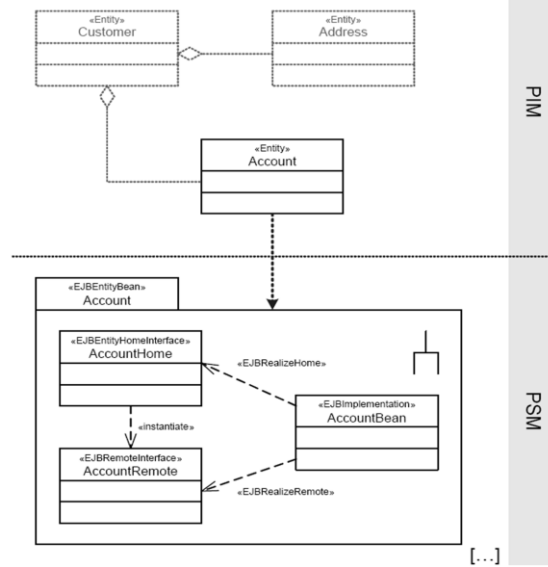
Object Management Group (OMG): *MDA Guide Version 1.0.1*.

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

PIM auf **höherer Abstraktionsebene** angesiedelt.

NB: „Gabelsymbol“ steht für <<subsystem>> (= Komponente mit eigenständigem Verhalten)

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



56

Literatur:

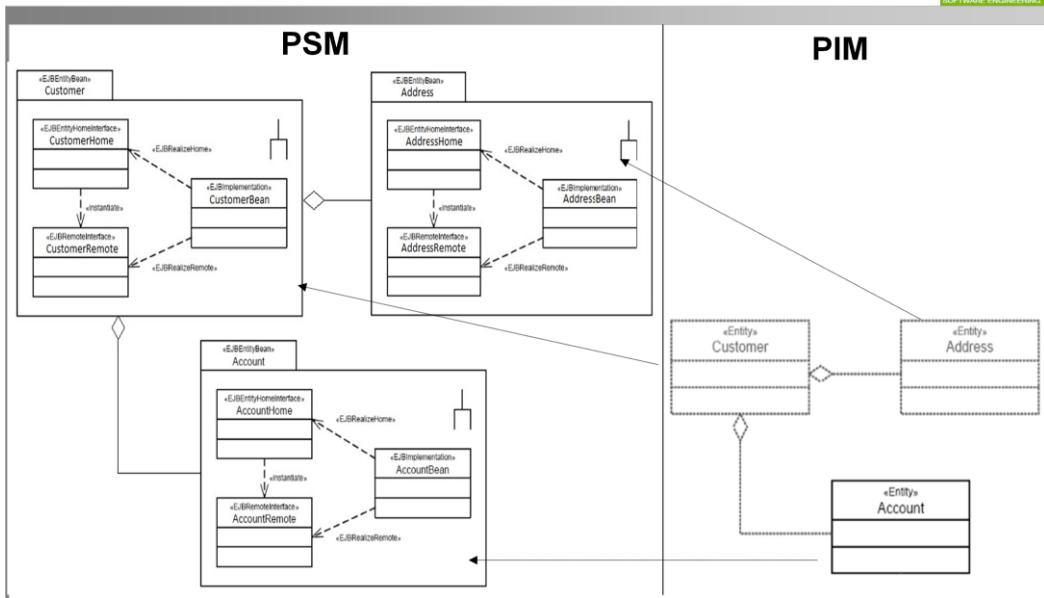
V. Gruhn: **MDA - Effektives Software-Engineering**

<http://www.ub.tu-dortmund.de/katalog/titel/1223129>

·Abschnitt 2.2.3 (S.25-31)

·Abbildung 2.9 (S.29)

·Abschnitt 5.5 (S.178-180)



57

Literatur:

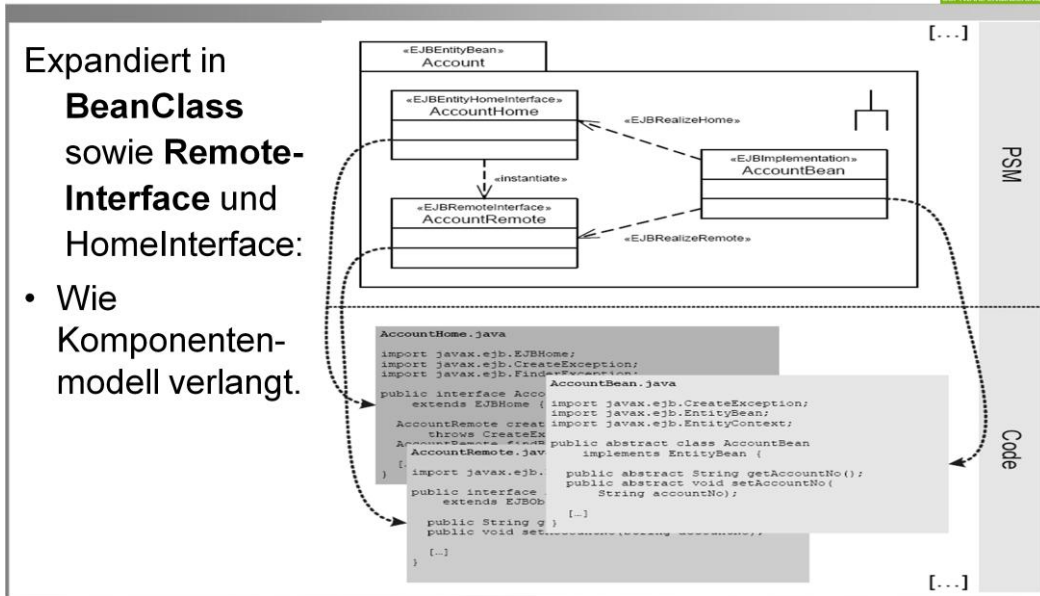
V. Gruhn: **MDA - Effektives Software-Engineering**

<http://www.ub.tu-dortmund.de/katalog/titel/1223129>

·Abschnitt 2.2.3 (S.25-31)

·Abbildung 2.9 (S.29)

·Abschnitt 5.5 (S.178-180)



58

Literatur:

V. Gruhn: **MDA - Effektives Software-Engineering**

<http://www.ub.tu-dortmund.de/katalog/titel/1223129>

·Abschnitt 2.2.3 (S.25-31)

·Abbildung 2.10 (S.30)

·Abschnitt 5.5 (S.178-180)

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



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.

59



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.**

60



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.

61

Literatur:

V. Gruhn: **MDA - Effektives Software-Engineering**

<http://www.ub.tu-dortmund.de/katalog/titel/1223129>

Abschnitt 5.3.3 (S.164-167)

ATL (S.167)

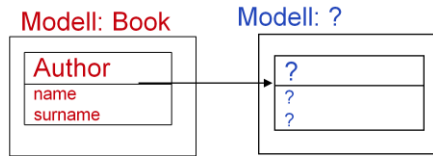


Resultat der Transformation von Book?

```

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“.

62

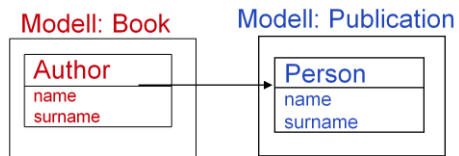


Resultat der Transformation von Book?

```

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

```



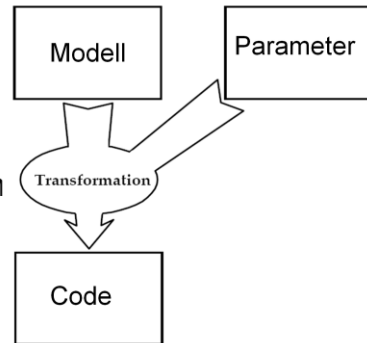
- Ausgangsmodell „IN“, konform zum Metamodell „Book“, in Modell „OUT“ (konform zu „Publication“) überführen.
- Regel überführt Elemente vom Typ „Author“ in Elemente vom Typ „Person“.

63

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



64



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.

65

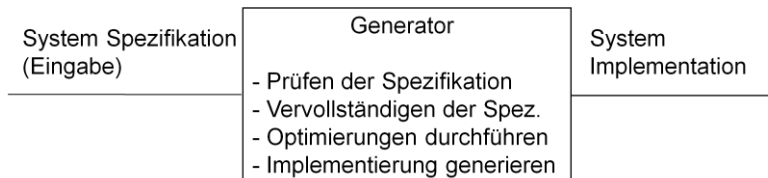


Generatoren:

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

Software-Artefakt:

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



66



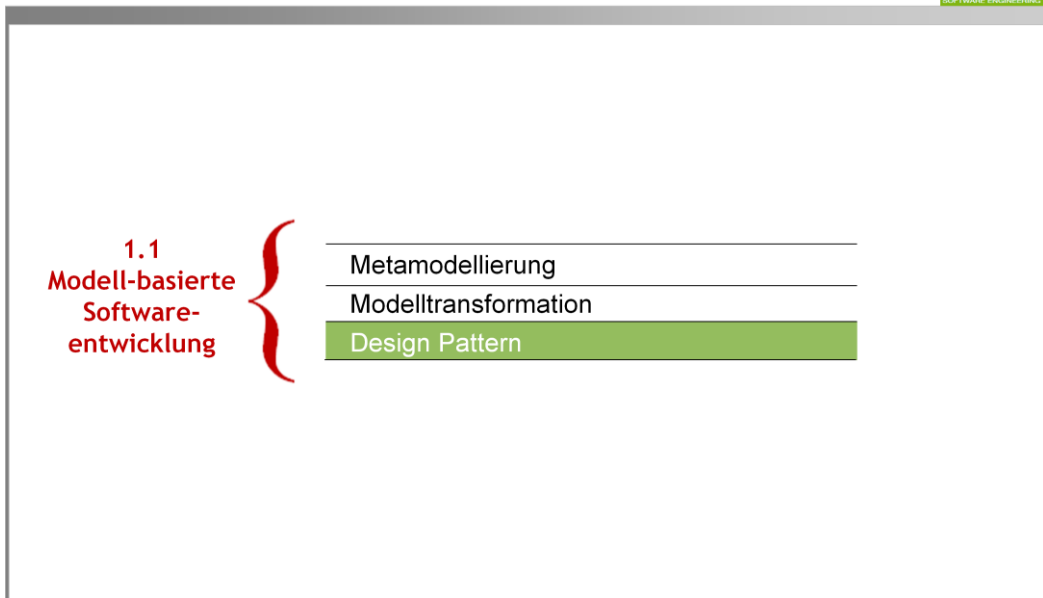
- Codegenerierung arbeitet mit **variabilisiertem** Programmcode.
→ Programmcode mit Variationspunkten.
- Eindeutige Ausprägung durch Parametrierung des Programmcodes.
- Aufwand der Generatorentwicklung nicht trivial.
- **Eignung:** Für Lösungen mit entsprechend großer Zahl von Variationen in Praxis

Vorteil des Einsatzes:

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

67

- **Eignung:** Für Lösungen mit entsprechend großer Zahl von Variationen in Praxis:
 - Technische Domänen: Hibernate, EJBs, Spring Beans, ...
 - Architekturschichten: Persistenzschicht.
 - Fachliche Variationen.



Literatur:

V. Gruhn: **MDA - Effektives Software-Engineering**

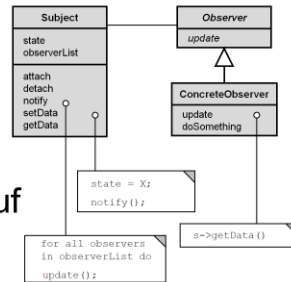
<http://www.ub.tu-dortmund.de/katalog/titel/1223129>

Kapitel 2



OOD mehr als „Diagramme malen“ !

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

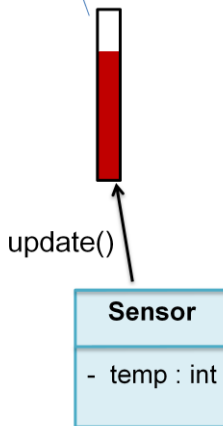


**Ziel: Wissen über Design-Lösungen:
festhalten, vermitteln, verwenden, erhalten !**



Thermometeranzeige hinzufügen - gutes Design ?

Celcius-Thermometer



```
class Sensor {
    int temp ;
    Thermometer ct ;

    ...

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



Fahrenheit-Thermometer

Celcius-Thermometer

Ungünstig: Muss Sensor ändern !
→ Wie geht es besser ?

```

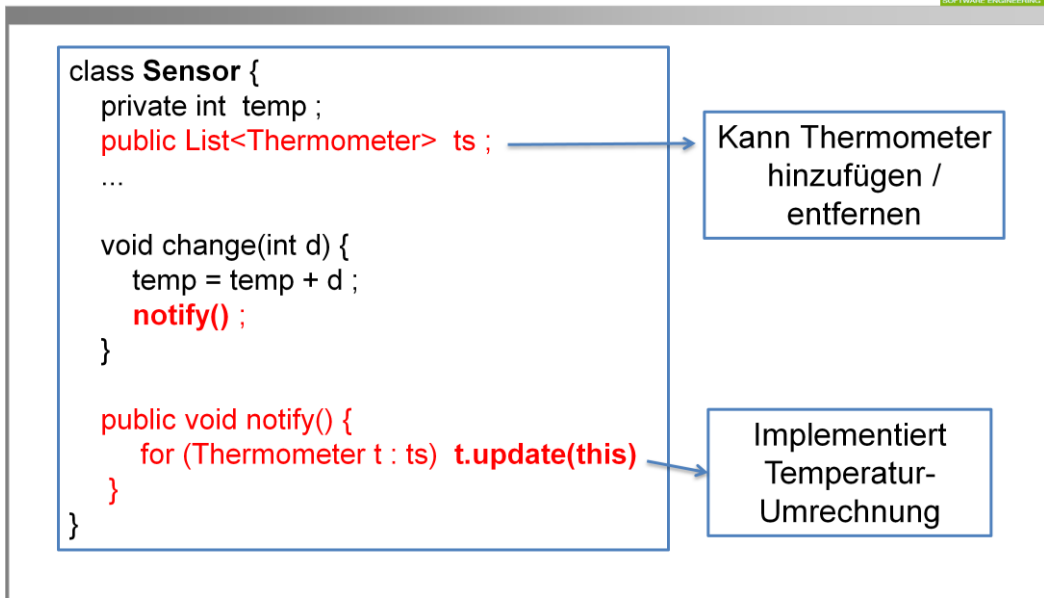
class Sensor {
    int temp ;
    Thermometer ct ;
    Thermometer ft ;
    ...
    void change(int d) {
        temp = temp + d ;
        ct.update(temp) ;
        ft.update(9*temp/5 + 32)
    }
}
    
```

Sensor
- temp : int

71

PROBLEM:

1. If we want to add more thermometers we have to change the code of Heater.
2. Calculation of temperature conversions is done in Heater, which is not logical.

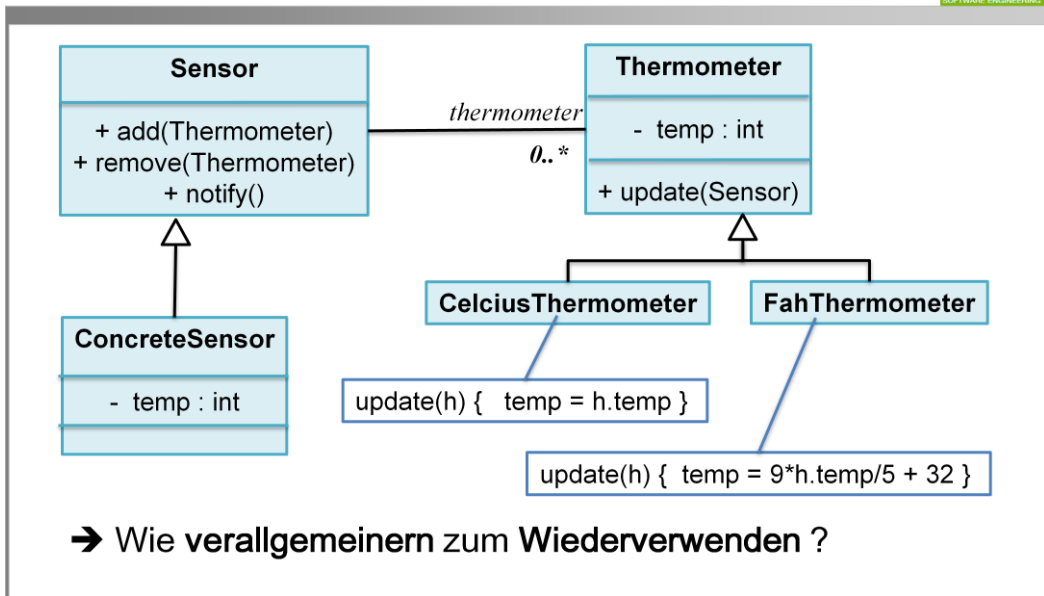


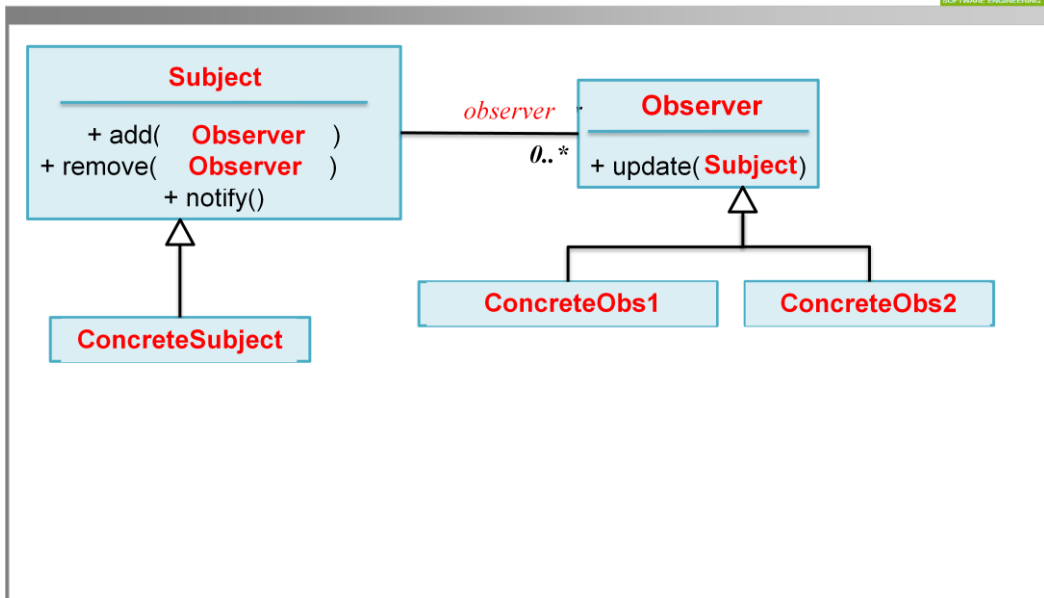
72

Call notify() to let the thermometers know that Heater's state has changed.

We put this in a separate method "notify" so that other methods in Heater (e.g. "reset") can use it too.

Now we can add/remove Thermometer at will. We can even add different kinds of thermometers!







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

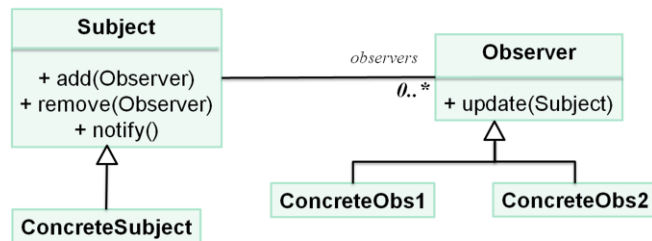


2. Problem:

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

3. Lösung

- Struktur:



ConcreteSubject notifies its observers whenever a change occurs that could make its observers state inconsistent with its own

After being informed of change, a ConcreteObserver queries the subject to reconcile its state with subjects.

Observer object that initiates change request postpones its update until it gets notification from subject. Notify() is not always called by subject. Can be called by an observer, or any other object.



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`



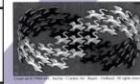
Überblick „Gang of Four“ Entwurfsmuster

Softwarekonzepte
WS 2014

Design Patterns

Elements of Reusable
Object-Oriented Software

Erich Gamma
Richard Helm
Ralph Johnson
John Vlissides



Foreword by Grady Booch

		<i>Zweck</i>		
		Erzeugung	Struktur	Verhalten
<i>Bereich</i>	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

78



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.



[http://ps.ipd.kit.edu/backend/index.php/veroeffentlichungen-
details_en/items/3602.html](http://ps.ipd.kit.edu/backend/index.php/veroeffentlichungen-details_en/items/3602.html)

A Controlled Experiment in Maintenance Comparing Design Patterns to
Simpler Solutions

Journal Article

Dec

2001

Authors

Lutz Prechelt, Barbara Unger, Walter F. Tichy, Peter Brössler, Lawrence G.
Votta

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





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

- Metamodellierung
- UML-Erweiterungen
- Modelltransformationen
- Design Patterns

Im nächsten Kapitel: Object Constraint Language (OCL)





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:

XML-Schema
Grammatik
Definition S/T-Netz
UML-Klasse
Metamodell

Instanz:

XML-Datei
Programmiersprache: Java
S/T-Netz Bestückungsroboter
Objekt
Modell

Literatur:

V. Gruhn: **MDA - Effektives Software-Engineering**

<http://www.ub.tu-dortmund.de/katalog/titel/1223129>

Abschnitt 3.3 (S.84)

Abschnitt 3.3.1 (S.84-87)



Metamodelle definieren **Modellelemente**:

- „Sprachdefinition“

Metamodelle für:

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

Modelle: Instanzen ihrer Metamodelle.

Literatur:

V. Gruhn: **MDA - Effektives Software-Engineering**

<http://www.ub.tu-dortmund.de/katalog/titel/1223129>

Abschnitt 3.3 (S.84)

Abschnitt 3.3.1 (S.84-87)



Bestandteile:

- **Abstrakte Syntax:**

- Modellelemente.

- **Statische Semantik:**

- Wohlgeformtheit.

- **Konkrete Syntax:**

- Notation, ggf. graphische Symbole.

- **Dynamische Semantik:**

- Verwendung, Bedeutung.

**Metamodell
+ Constraints (OCL)**

**Informelle
Beschreibung**

Literatur:

V. Gruhn: **MDA - Effektives Software-Engineering**

<http://www.ub.tu-dortmund.de/katalog/titel/1223129>

Abschnitt 3.1.2 – Kurzer Exkurs: Abstrakte Syntax vs.
Konkrete Syntax (S.68-71)

Abschnitt 3.2.3 (S.80-83)

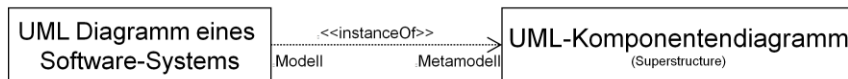


Modell:

- **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).



Literatur:

V. Gruhn: **MDA - Effektives Software-Engineering**

<http://www.ub.tu-dortmund.de/katalog/titel/1223129>

Abschnitt 3.2.3 (S.80-83)

Abbildung 3.10 (S.81)



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):

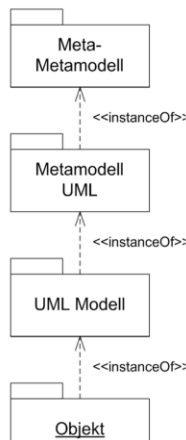
- Definiert Abbildung der MOF auf XML.
- Ermöglicht standardisierten Austausch von beliebigen Meta-Modellen zwischen Tools.
 - z.B.: Transformatoren, Modellierungswerkzeugen, Codegeneratoren usw.
 - Grundvoraussetzungen zum Aufbau funktionierender MDA-Infrastruktur.

Literatur:

V. Gruhn: **MDA - Effektives Software-Engineering**

<http://www.ub.tu-dortmund.de/katalog/titel/1223129>

Abschnitt 2.2.4 (S.31-32)



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.

Literatur:

V. Gruhn: **MDA - Effektives Software-Engineering**

<http://www.ub.tu-dortmund.de/katalog/titel/1223129>

Abschnitt 3.2.3 (S.80-83)

Abbildung 3.10 (S.81)



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

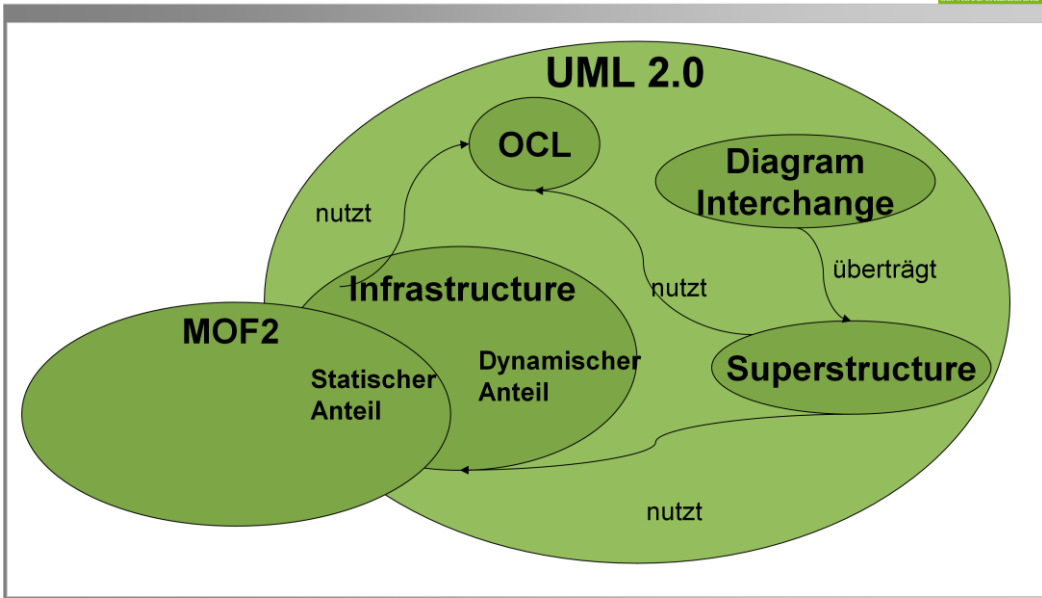
1. - Verbesserte architekturelle Angleichung zwischen **UML, MOF und XMI**.
 - Einheitliche, auf Nutzerebene verfügbare Erweiterungsmechanismen und Profile in einer zum **Metamodell** konsistenten Form.
2. - Direkte Unterstützung von Skalierbarkeit und Abkapselung für **Verhaltensmodellierung**.
 - Eindeutige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

91

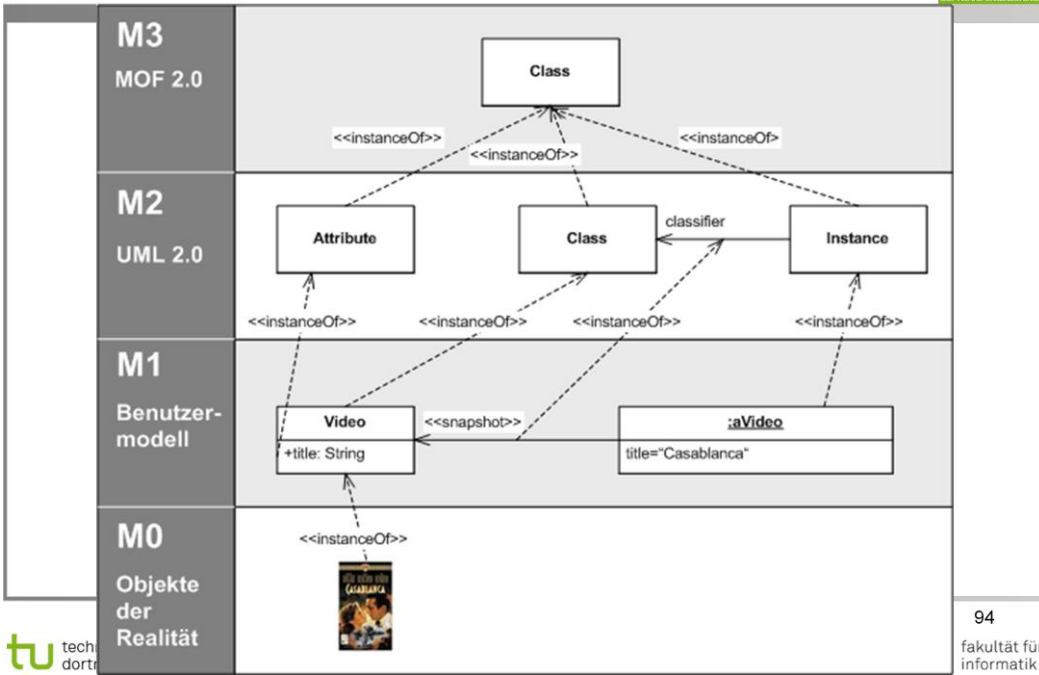


Welche der beiden Ziele sind **Infrastructure** bzw. **Superstructure** zuzuordnen?

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



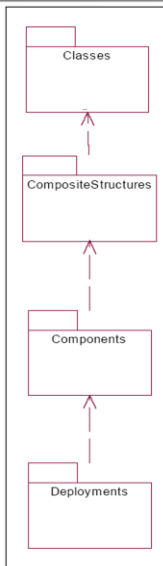
93





- 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

95



Aufgeteilt in verschiedene Pakete, die verschiedene Diagrammtypen repräsentieren:
= Klassendiagramm, Objektdiagramm, Paketdiagramm

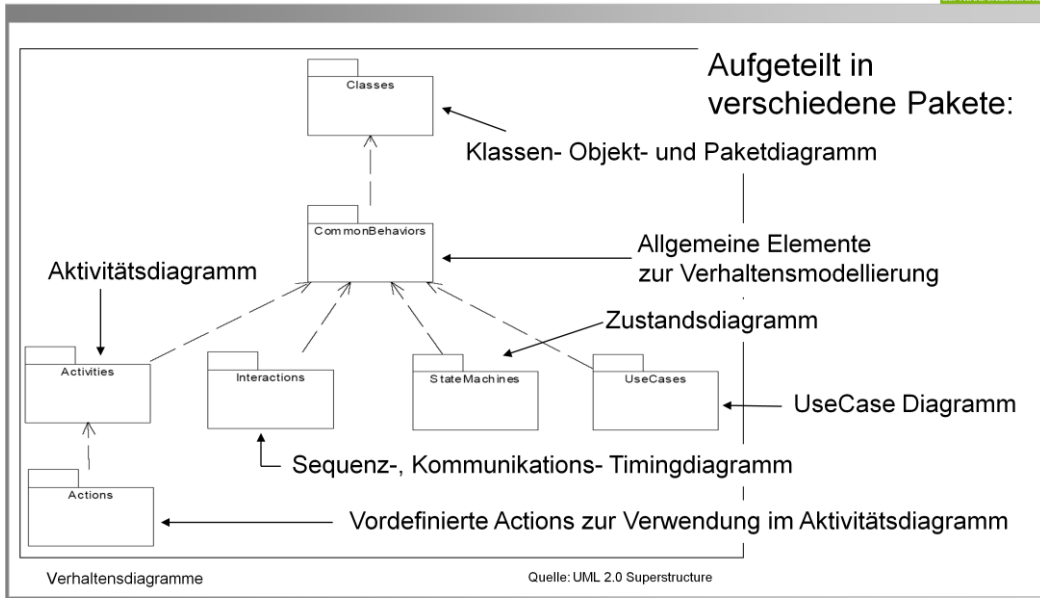
= Kompositionsstrukturdiagramm

= Komponentendiagramm

= Deploymentdiagramm

Quelle: UML 2.0 Superstructure

96



97



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.

98



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.

99



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.**