Softwarekonstruktion
im Wintersemester 2012 / 2013

3.2 Modellbasierte Softwareentwicklung
[inkl. Beiträge von Prof. Volker Gruhn]

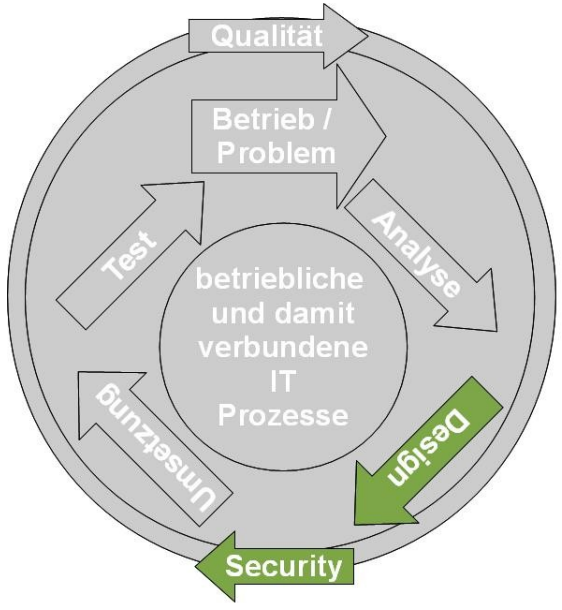
Prof. Dr. Jan Jürjens

TU Dortmund, Fakultät Informatik, Lehrstuhl XIV

v. 27.01.2013

Einordnung Modellbasierte Softwareentwicklung

- Qualitätsmanagement
- Testen
- Modellgetriebene SW-Entwicklung
  - OCL
  - Algebraische Spezifikation
  - Modellbasierte Softwareentwicklung



3.2 Modell-basierte Software-entwicklung

- Motivation & Einführung
- Semantik
- Semantik im UML Metamodell
- Modelltransformation

Motivation

Beispiele für die Zunahme der Komplexität von Software:

- Anstieg der Anzahl von Electronic Control Units (ECU) von der letzten zur neuen Mercedes S-Klasse: 64% von 45 ECUs auf bis zu 72 ECUs
- 1970 hatte die eingebettete Software eines Kfz ca. 100 LOC
  Heute: 1.000.000 LOC, bei Premiumfahrzeugen 10.000.000 LOC
- Länge und Gewicht des VW Phaeton Kabelbaums: 3960m, 64kg
- Beim Ausfall eines der Bremslichter beim
  US-New Beatle blockiert die gesamte Automatik

Motivation

Zunehmende Komplexität

Technische Komplexität
- Umfang der Datenmodelle
- Verteilte Implementierung
- Heterogenität der Infrastruktur (Kommunikationsmedien, Protokolle Betriebssysteme / Plattformen)

Funktionale Komplexität
- Umfang der Funktionalität
- Diversifizierung der Funktionalität
- Mensch-Maschine Schnittstelle

Entwicklungskomplexität
- Einflussnahme des Kunden
- Entwicklung in Zulieferketten (Integrationsaufwand)
- Qualitätsanforderungen
- Innovationsdruck

Qualität, Kosten, Termine

Qualität
- Nutzersicht: Funktionsvielfalt, Usability, Sicherheit, Performanz
- Entwicklungssicht: Wartbarkeit,
- Wiederverwendbarkeit

Kosten
- Entwicklungskosten
- Vermarktbarkeit
- Kosten im Gesamtlebenszyklus (Entwicklung, Inbetriebnahme, Wartung) Total Life Cycle Costs

Entwicklungszeit
- Time to Market
- Reaktionszeit bei Änderungen

Motivation

Ansätze, um die Komplexität zu beherrschen:

- Abstraktion
  - Ausblenden von Detailinformation
  - Einsatz von geeigneten SW-Modellen
- Strukturierung / Modularisierung
  - Gliederung und Aufteilung in klar abgegrenzte Unterstrukturen / Module
  - Partitionierung der Aufgaben (Dekomposition)
  - Einsatz von geeigneten SW-Modellen
- Methodik und Systematik
  - Verwendung bewährter Verfahren und Lösungsmuster
  - Systematisierung des Entwicklungsprozesses
  - Code Ebene: Design Pattern, Architektur: Referenzarchitekturen

Modellbasierte SW-Entwicklung

Kernideen der modellbasierten SW-Entwicklung (Auch: Modellbasiert, Modellzentriert, Model-driven, Model-based, MDA):

- Modelle sind zentrales Artefakt im SW-Prozess
  - Konsequente Nutzung von erster bis zur letzen Phase des Lebenszyklus zur letzten (Anforderung bis Wartung)
  - Vermeidung von Modellbrüchen im SW-Prozess
- Einsatz von Softwaremodellen mit fachlicher Semantik
  - Fachliche Anforderung von konkreter Technologie entkoppeln
  - Wiederverwendung fachlicher Aspekte

Modellbasierte SW-Entwickliung

Verwendung von Modellen:

- Generierung von Programmcode (Automatisierung)
- Dokumentation
  - Dokumentieren des Systems und seiner Bestandteile
  - Funktionsumfang, Einsatz, Umgebung
- Kommunikation
  - Projektkommunikation
  - Design- / Architekturentscheidung festhalten und diskutieren
- Spezifikation
  - Vorlage für Implementierung (Generierung von Software)
  - Vorlage für Tests
- Simulation
  - Validierung von Systemteilen
  - Frühes Feedback -> Auswirkung von Designentscheidungen

Modellbegriff und SW-Modelle

- Modell
  - Abbildung: Welt → Diskrete Struktur
  - „A model is a simplification of reality“ [BRJ01]
  - „A *model* is a *simplification of a system* built with an intended goal in mind. The model should be able to answer questions in place of the actual system.“ [Beziv01]

- Aspekte, die modelliert werden
  - Struktur
  - Beziehungen
  - Verhalten

Modellbegriff und SW-Modelle

- Modelle im SE-Engineering
  - Modellierungskonzepte
  - Grafische Modelle und Modelle auf Textbasis
  - Modellbasierte SW-Entwicklung bedeutet nicht zwangsläufig, dass mit der UML modelliert wird

- Beispiele für SW-Modelle
  - Algebren (s. Kap. 3.1), Petrinetze, XML-Schema, UML, EPK, Skizzen

SW-Modelle: XML-Schema (XSD)

- Modellierung strukturierter Textdaten
  - Technisches Modell von z.B. Schnittstellen, Datenbanken und Konfigurationsdateien
  - XML-Schema definiert Elemente und Attribute von XML-Dateien mit
    - einfachen Datentypen
    - komplexen Datentypen
    - Wertebereichen
    - Reihenfolge und Anzahl von Elementen
    - ein XML-Schema ist wiederum eine XML-Datei
  - XML-Dateien
    - Strukturierte Beschreibung
    - Kann von Menschen und Maschinen verstanden werden

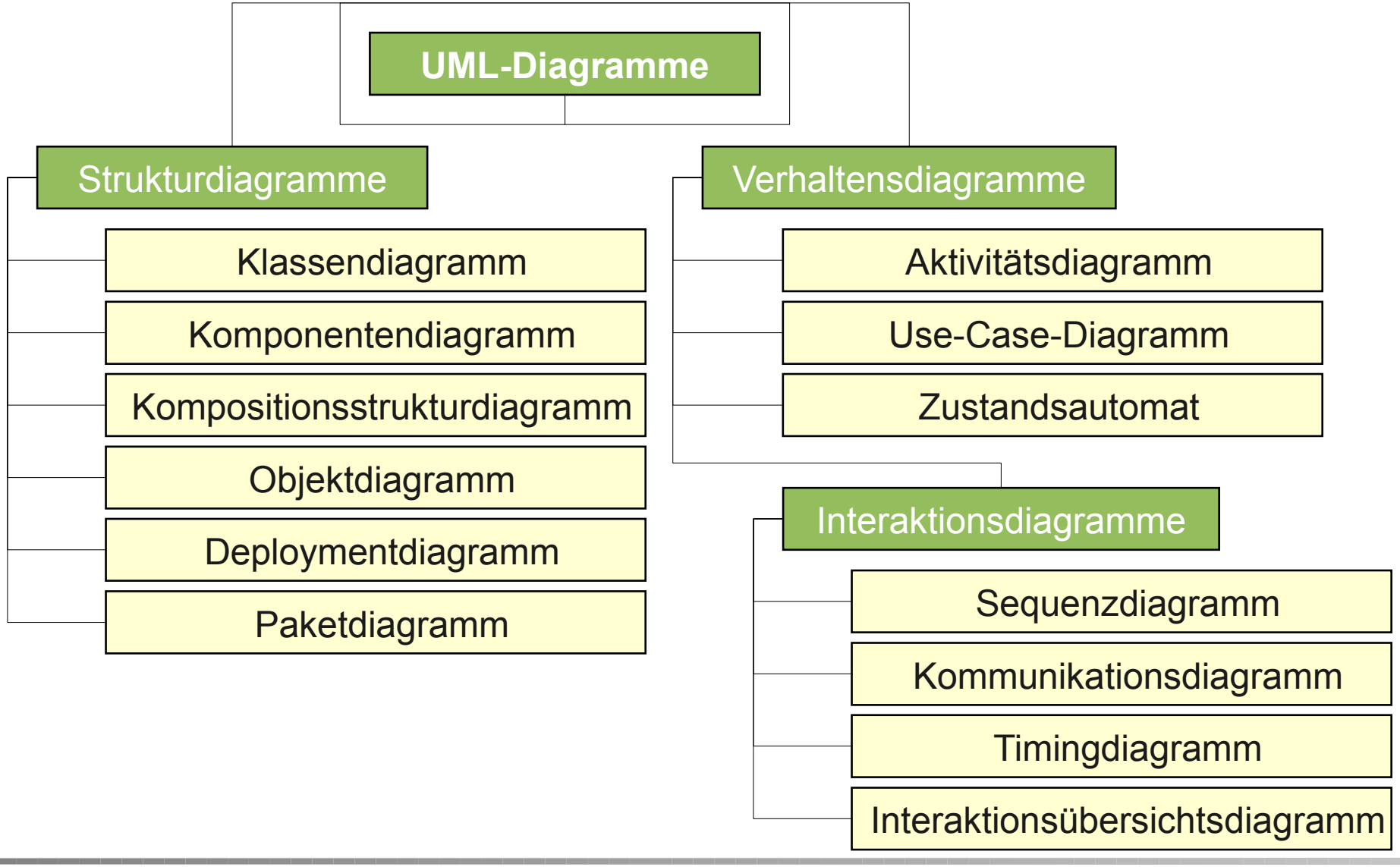
SW-Modelle: XSD Beispiel

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema" elementFormDefault="qualified" attributeFormDefault="unqualified">
      <xs:element name="auftrag">
         <xs:complexType>
            <xs:all>
               <xs:element name="kundennummer" type="xs:int"/>
               <xs:element name="auftragsdatum" type="xs:date"/>
               <xs:element name="ausführungsdatum" type="xs:date"/>
               <xs:element name="auftragsposition">
                  <xs:complexType><xs:sequence maxOccurs="unbounded"><xs:element
      ref="auftragspositiontype"/></xs:sequence></xs:complexType>
               </xs:element>
               <xs:element name="kundenanschrift">
                  <xs:complexType><xs:sequence><xs:element ref="adressetype"/></xs:sequence></xs:complexType>
               </xs:element>
               <xs:element name="rechnungsanschrift">
                  <xs:complexType><xs:sequence><xs:element ref="adressetype"/></xs:sequence></xs:complexType>
               </xs:element>
               <xs:element name="installationsanschrift">
                  <xs:complexType><xs:sequence maxOccurs="unbounded"><xs:element ref="adressetype"/></xs:sequence></xs:complexType>
               </xs:element>
            </xs:all>
         </xs:complexType>
      </xs:element>
      <xs:element name="auftragspositiontype">
         <xs:complexType>
            <xs:all>
               <xs:element name="beschreibung" type="xs:string"/>
               <xs:element name="unterposition">
                  <xs:complexType><xs:sequence maxOccurs="unbounded"><xs:element ref="unterpositiontype"/></xs:sequence></xs:complexType>
               </xs:element>
            </xs:all>
         </xs:complexType>
      </xs:element>
      <xs:element name="unterpositiontype">
         <xs:complexType><xs:all><xs:element name="beschreibung" type="xs:string"/></xs:all></xs:complexType>
      </xs:element>
      <xs:element name="adressetype">
         <xs:complexType><xs:all>
               <xs:element name="vorname" type="xs:string"/><xs:element name="nachname" type="xs:string"/>
               <xs:element name="strasse" type="xs:string"/><xs:element name="hausnummer" type="xs:int"/>
               <xs:element name="postleitzahl" type="xs:string"/> <xs:element name="stadt" type="xs:string"/>
         </xs:all></xs:complexType>
      </xs:element>
</xs:schema>
```

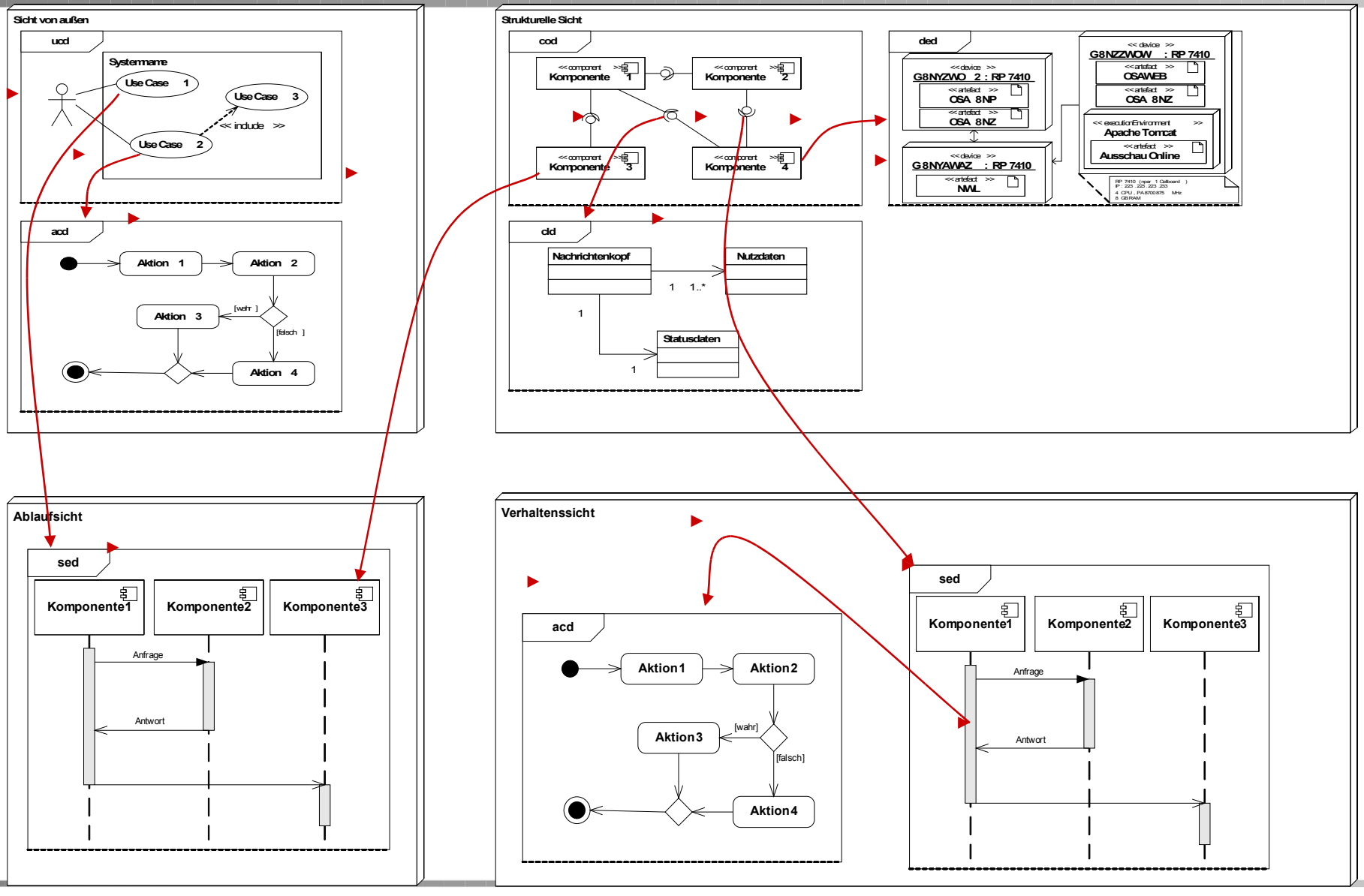
SW-Modelle: UML

- Unified Modeling Language (UML) [www.omg.org/uml]
  - Modellierung, Dokumentation, Spezifizierung und Visualisierung von komplexen Softwaresystemen (unabhängig vom Fachgebiet)
  - Standard der Object Management Group (OMG)
  - Stellt viele unterschiedliche Modellierungskonzepte auf einer einheitlichen Basis bereit
    - Insgesamt 13 verschiedene Diagrammtypen
  - Unterscheidung Modell / Diagramm
    - Ein UML-Modell wird durch eine Gruppe von Diagrammen repräsentiert
    - Ein Diagramm entspricht einer bestimmten Sicht auf das Modell, stellt oft nur einen Teil der im Modell enthaltenen Information dar
    - Modellelemente können mehrfach vorkommen

SW-Modelle: UML



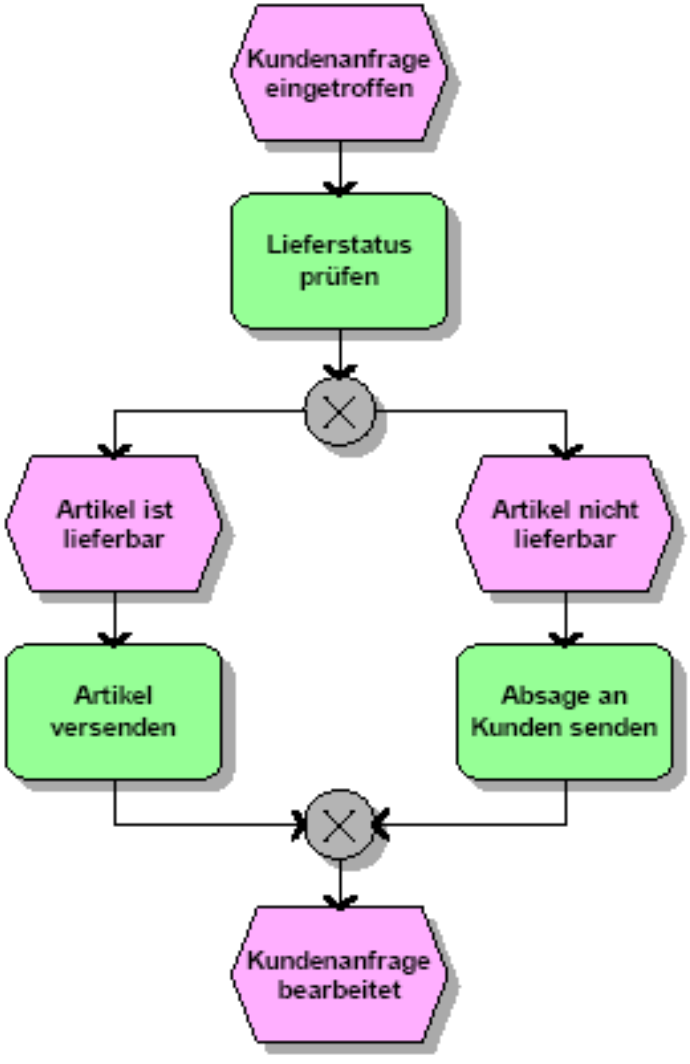
SW-Modelle: UML - Beispiel

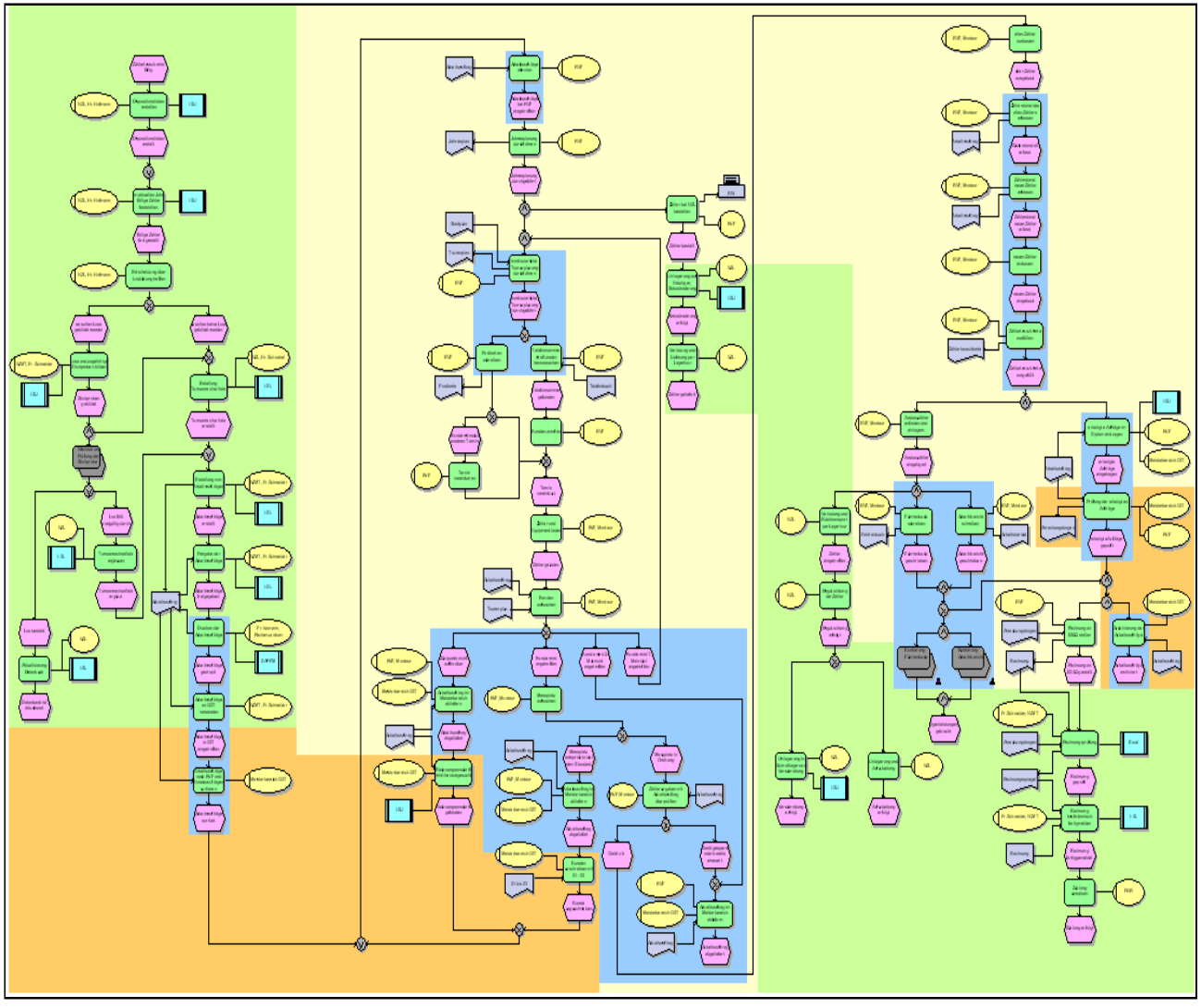


SWK-Modelle: EPK

- Ereignisgesteuerte Prozessketten (EPK)
  - Beschreibung von Geschäftsprozessen von Unternehmen
  - Symbole in EPKs
    - Ereignis: Betriebswirtschaftlicher Zustand
    - Funktion: Aktivität eines Geschäftsprozesses
    - Verknüpfungsoperatoren: AND, XOR, OR
    - Verbindung der Elemente per Kontrollfluss-Pfeil

SWK-Modelle: EPK - Beispiel

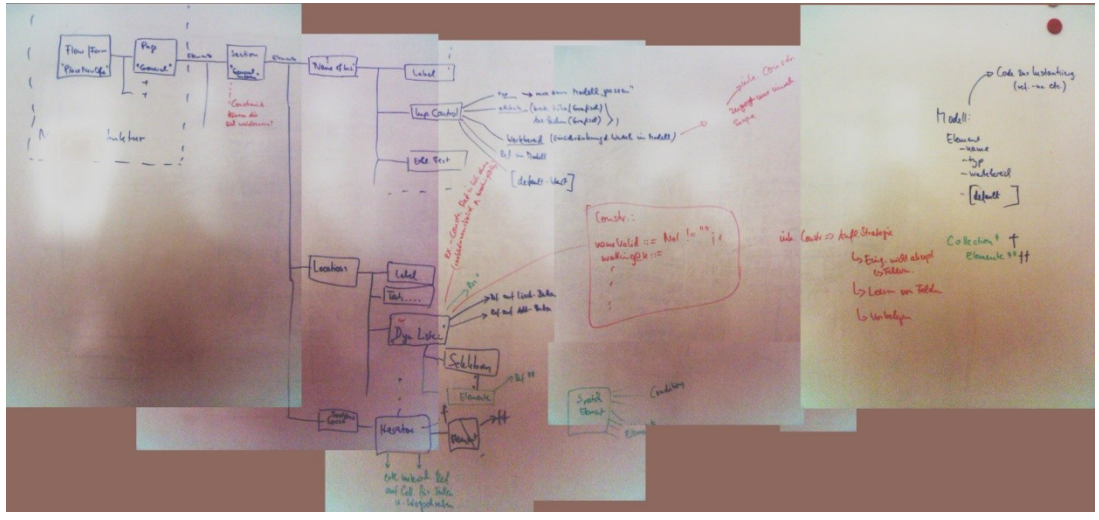


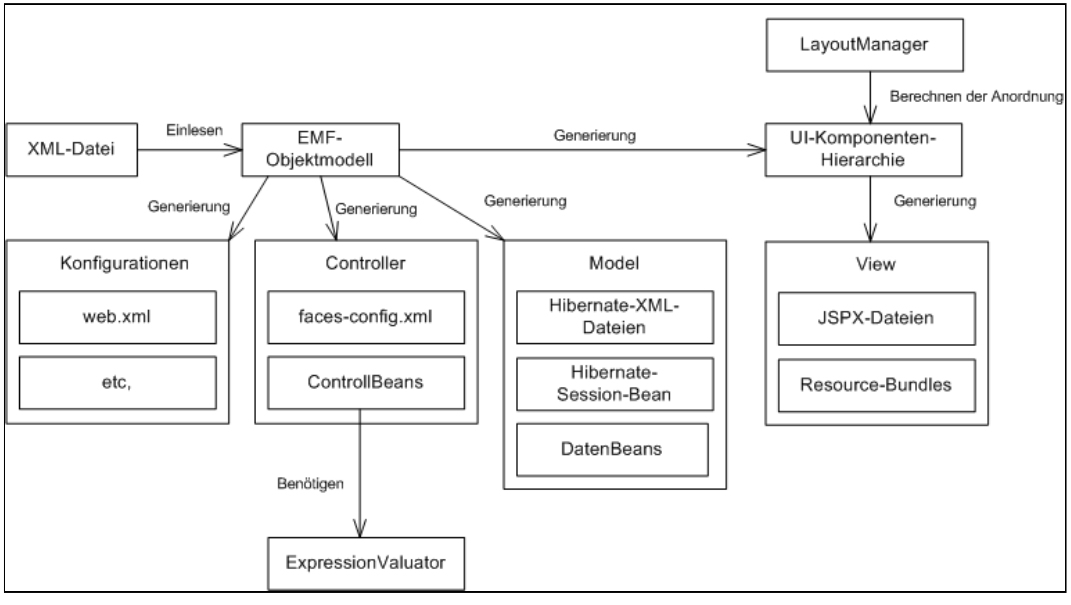


SWK-Modelle: Skizzen

- Skizzen
  - Per Hand (Papier, Flipchart, Whiteboard)
    oder mit Tool (Powerpoint, Visio, Photoshop) erstellte Grafiken
  - Verdeutlichen oft fachliche oder technische Zusammenhänge, aber in der Regel ohne klar definierte Semantik
    - Interpretation der Skizzen von Personen, die nicht an der Erstellung beteiligt waren nicht gewährleistet
  - Sind oft die wichtigsten Artefakte der Kommunikation innerhalb eines Projektteams

SWK-Modelle: Skizzen - Beispiel





3.2 Modell-basierte Software-entwicklung

- Motivation & Einführung
- **Semantik**
- Semantik im UML Metamodell
- Modelltransformation

Semantik von Modellen

- Syntax und Semantik von Modellen
  - Formales Modell
    - Syntax und Semantik sind auf mathematischer oder streng logischer Struktur definiert (Algebren, Petrinetze)
  - Semi-Formales Modell
    - Syntax ist präzise definiert.
    - Semantik ist nicht komplett formal definiert.
    - Wichtige Teile sind jedoch formal definiert.
  - Modell mit freier Semantik / Skizze
    - Syntax und Semantik sind nicht, oder nur durch natürliche Sprache definiert (Skizzen)

Semantik von Modellen

- Definition der Semantik von Modellen
  - Vorteile formaler Semantik
    - Struktur bzw. Verhalten des modellierten Systems ist eindeutig beschrieben
    - Struktur und Verhalten können automatisch analysiert werden
    - Fehler und Inkonsistenzen innerhalb eines Modells können automatisch gefunden werden
  - Nachteile formaler Semantik
    - Formale Modelle eines „echten“ Systems sind sehr komplex und unübersichtlich
    - Erstellung und Wartung formaler Modelle ist sehr aufwändig
    - Modellierer muss den Formalismus gut kennen

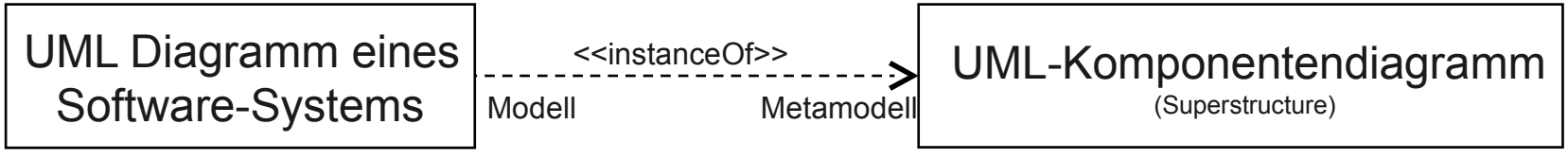
Model und Metamodell der UML

- Metamodell
  - „meta“ bedeutet soviel wie „über“
  - Metamodelle sind Modelle, die Modelle beschreiben
  - Definition aller Elemente einer 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

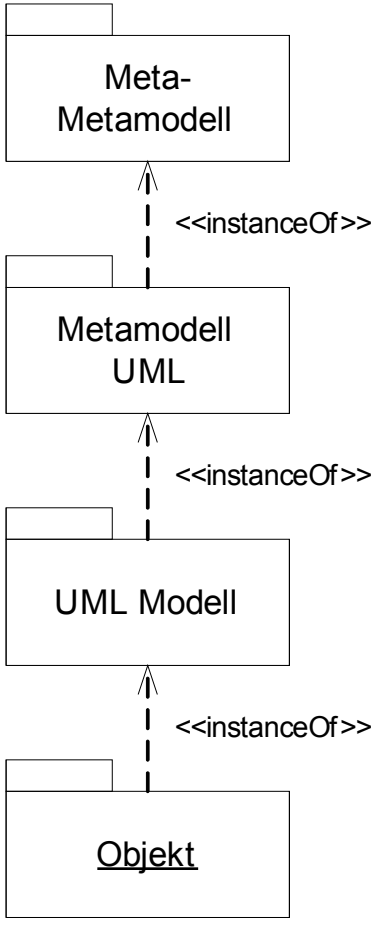
Model und Metamodell der UML

- Modell und Metamodell der UML
  - Real existierendes System wird als UML-Modell durch eine Anzahl von UML-Diagrammen beschrieben:
    - Klassendiagramm, Sequenzdiagramm, …
  - UML-Diagramme liefern Notationselemente, um reales System abstrakt zu beschreiben
    - Rechtecke, Pfeile, Balls, Sockets, Stereotypen, …
  - UML-Diagramme müssen selbst auch irgendwo definiert werden:
    - UML Standard (UML 2.0 Superstructure)



Modell und Metamodell der UML

Metamodellhierarchie der UML



UML Infrastructure: Definition der Elemente, mit der UML-Diagrammtypen spezifiziert werden können

UML Superstructure: Definition der 13 UML-Diagrammtypen

Modell eines konkreten Systems

Instanzen eines modellierten konkreten Systems

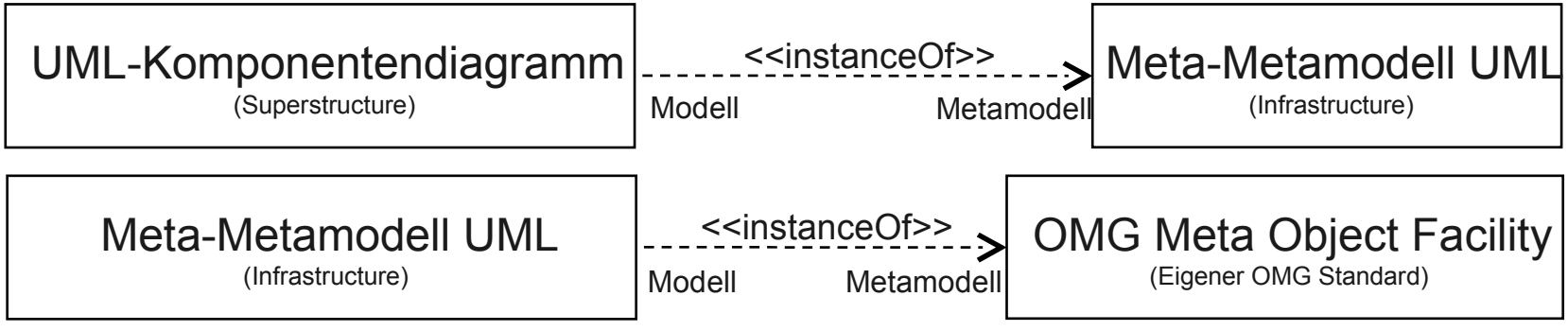
Modell und Metamodell der UML

- Infrastructure
  - Beschreibt das 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 können Elemente des Metaobjektes spezialisiert werden
- Beide zu erreichen auf der Webseite der OMG unter:

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

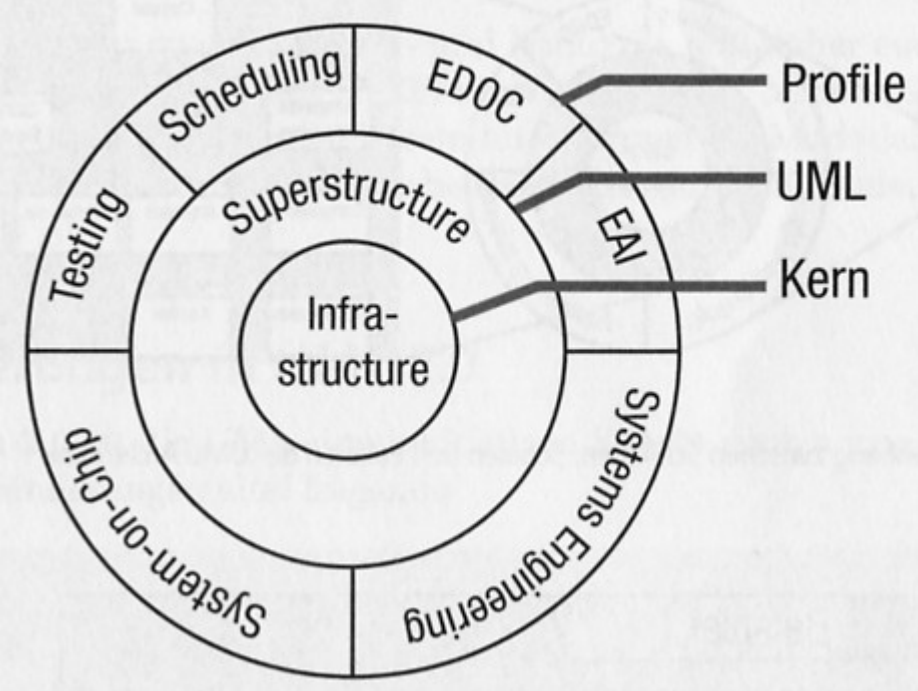
Modell und Metamodell der UML

- UML Infrastructure definiert ein Modell, von dem alle UML-Diagrammtypen eine Instanz bilden
- UML ist eine sich selbst beschreibende Sprache
  - UML Metamodell als Klassendiagramm beschrieben
- Grundlage des Aufbaus des UML Metamodells: Meta Object Facility
  - Einstufung der Modelle in die vier Ebenen



Modell und Metamodell der UML

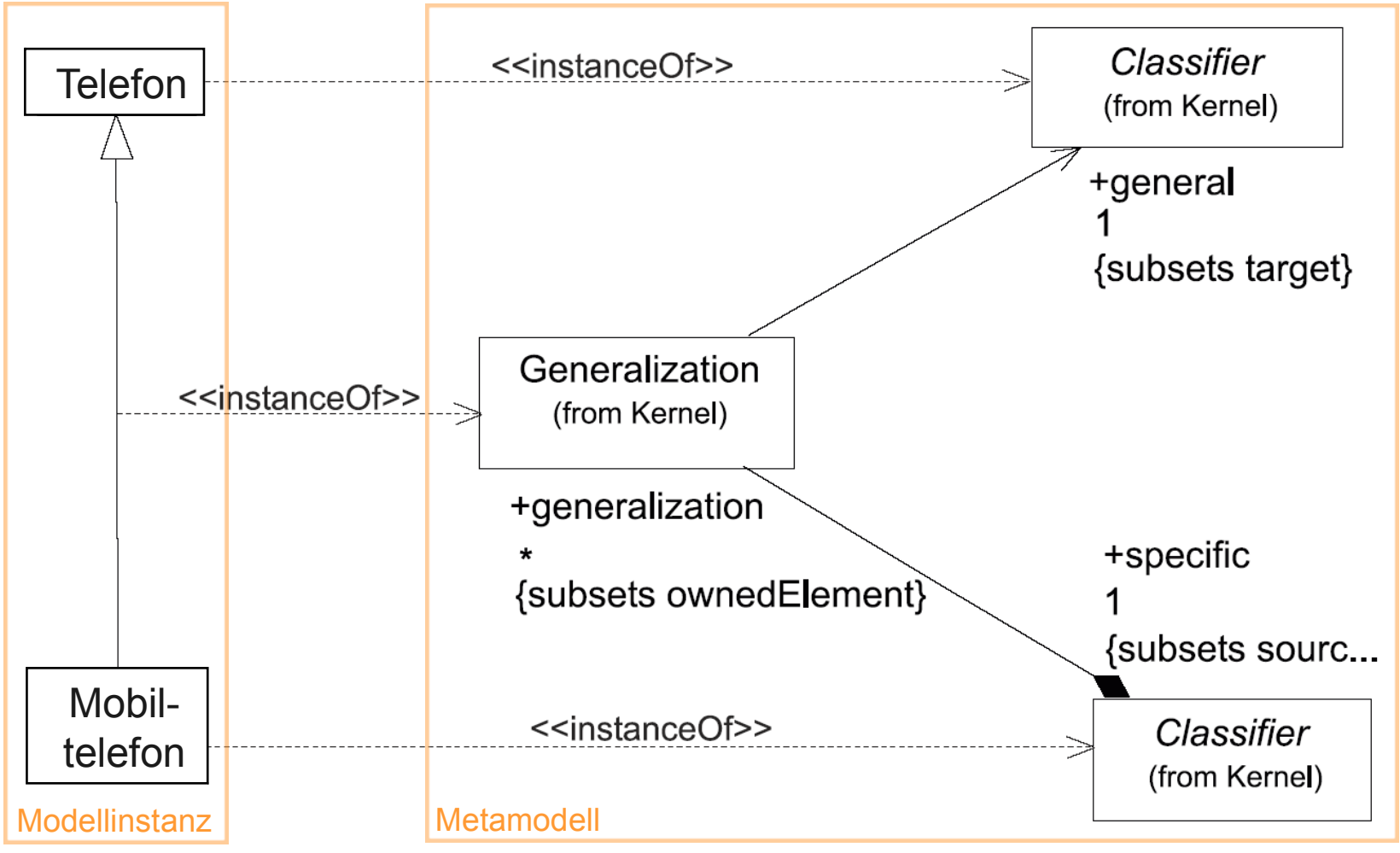
- Schichten-Architektur der UML
  - Kern: Infrastructure (Metamodel)
  - Eigentliche Sprachdefinition der UML: Superstructure
  - Optionale Erweiterungen: Profile



Quelle: UML für Studenten, Harald Störrle

Metamodell der UML - Beispiel

- Definition und Beispiel der Generalisierung



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

Aufbau des UML-Metamodells

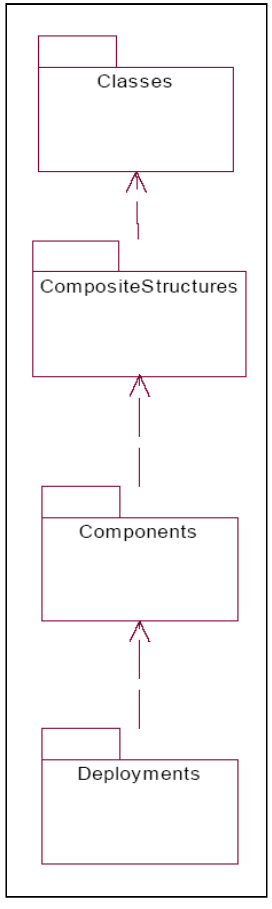
Aufbau des UML Metamodells

1. Definition der Strukturdiagramme
   - Classes (Klassen), Components (Komponenten), Composite Structures, Deployments
1. Definition der Verhaltensdiagramme
   - Actions, Activities, Common Behaviors, Interactions, Use Cases, State Machines
1. Definition zusätzlicher Konstrukte (Supplement)
   - Hilfskonstrukte (Auxilary Constructs)
     - Primitive Datentypen, Templates, Informationsflüsse, …
   - UML-Profile
1. 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 im Wesentlichen die verschiedenen Diagrammtypen repräsentieren:



= Klassendiagramm, Objektdiagramm, Paketdiagramm

= Kompositionsstrukturdiagramm

= Komponentendiagramm

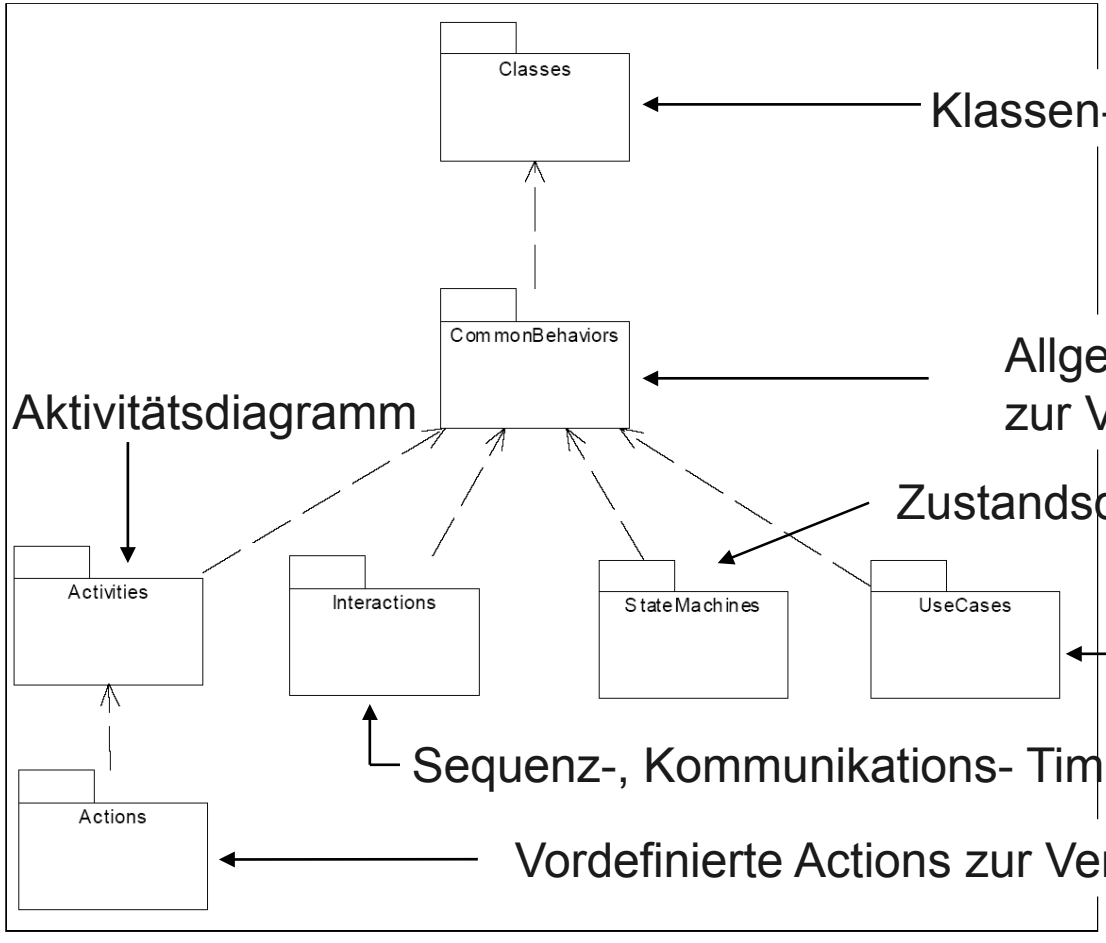
= Deploymentdiagramm

Quelle: UML 2.0 Superstructure

Aufbau des UML Metamodells

UML Verhaltensdiagramme

- Ebenfalls aufgeteilt in verschiedene Pakete



Klassen- Objekt- und Paketdiagramm

Allgemeine Elemente
zur Verhaltensmodellierung

Aktivitätsdiagramm

Zustandsdiagramm

UseCase Diagramm

Sequenz-, Kommunikations- Timingdiagramm

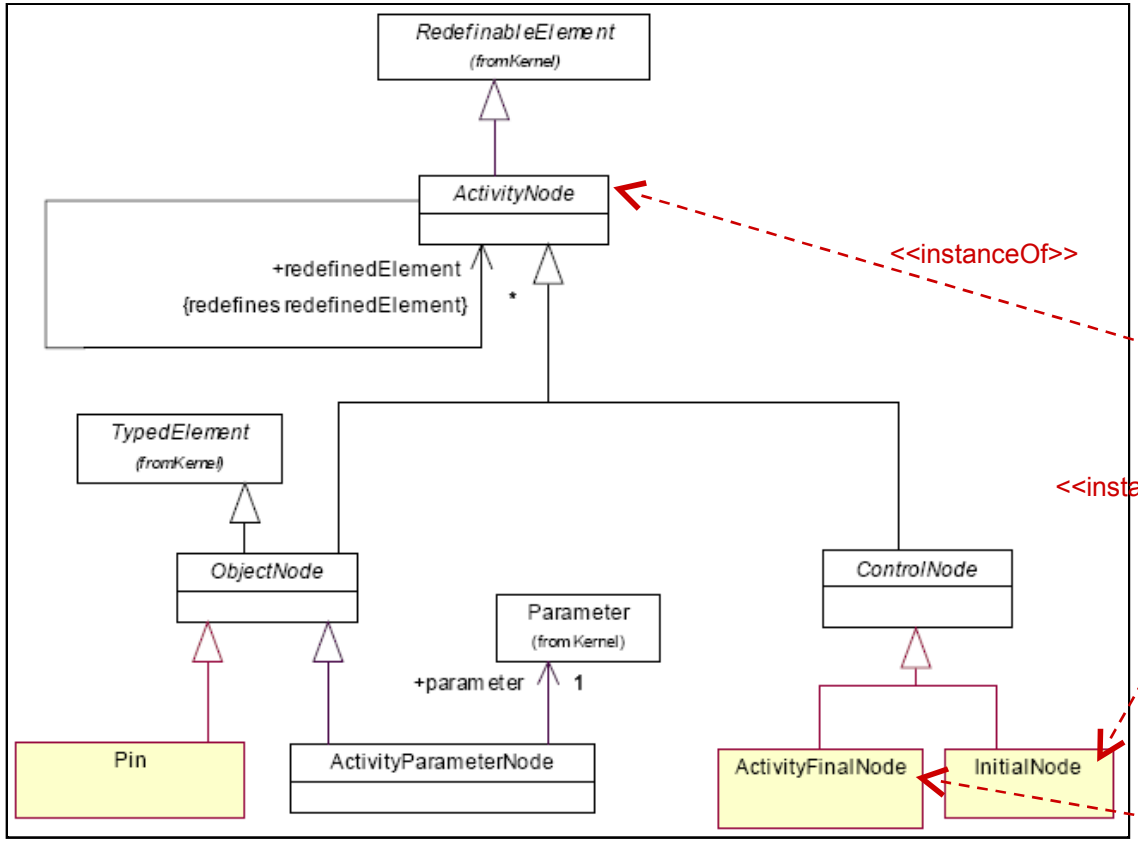
Vordefinierte Actions zur Verwendung im Aktivitätsdiagramm

Verhaltensdiagramme

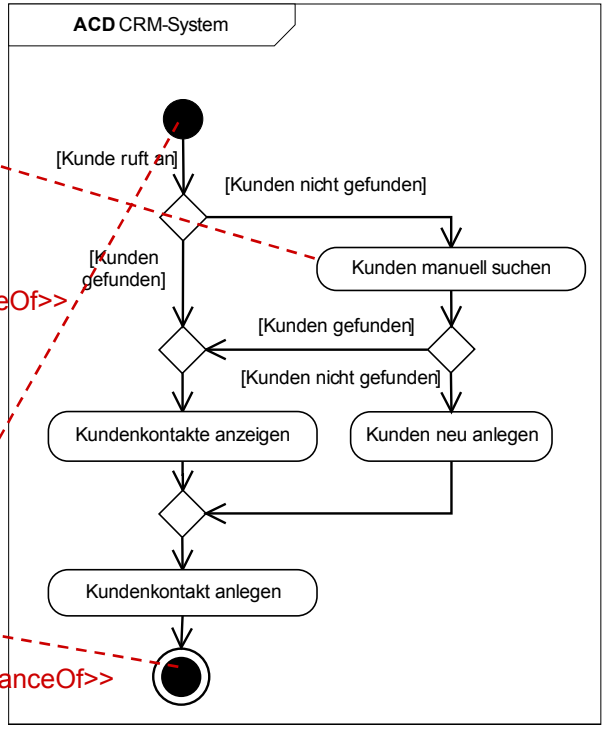
Quelle: UML 2.0 Superstructure

Aufbau des UML Metamodells

Beispiel: Definition der Elemente Aktivitätsdiagramm (Auszug)



Quelle: UML 2.0 Superstructure



3.2 Modell-basierte Software-entwicklung

- Motivation & Einführung
- Semantik
- **Semantik im UML Metamodell**
- Modelltransformation

Semantik im UML Metamodells

Definition der Semantik der Notationselemente der UML

- Zu jedem Notationselement gibt es einen Text „Semantics“
- Dieser Text definiert das Verhalten bzw. die Bedeutung des jeweiligen UML-Elementes
- Zu einigen Elementen werden Spielräume in der Interpretation eingeräumt („Semantics Variation Points“)
- Siehe folgendes Beispiel der „Action“ im Aktivitätsdiagramm
- Definition einer formalen Semantik (z.B. mit Hilfe von Petri-Netzen) ist Gegenstand vieler Forschungsprojekte im UML-Umfeld

Bsp. Action im Activity Diagram

12.3.2 Action (from CompleteActivities, FundamentalActivities, StructuredActivities)

Generalizations

- "ActivityNode (from BasicActivities, CompleteActivities, FundamentalActivities, IntermediateActivities, StructuredActivities)" on page 323.
- "ExecutableNode (from ExtraStructuredActivities, StructuredActivities)" on page 354.
- "Action (from BasicActions)" on page 230 *(merge increment)*.

Description

An action may have sets of incoming and outgoing activity edges that specify control flow and data flow from and to other nodes. An action will not begin execution until all of its input conditions are satisfied. The completion of the execution of an action may enable the execution of a set of successor nodes and actions that take their inputs from the outputs of the action.

Package CompleteActivities

In CompleteActivities, action is extended to have pre- and postconditions.

Attributes

No additional attributes

Associations

Package CompleteActivities

- localPrecondition : Constraint [0..*] Constraint that must be satisfied when execution is started.
- localPostcondition : Constraint [0..*] Constraint that must be satisfied when execution is completed.

Constraints

No additional constraints

Operations

[1] activity operates on Action. It returns the activity containing the action.

activity() : Activity;
activity = if self.Activity->size() > 0 then self.Activity else self.group.activity() endif

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.

See ValuePin and Parameter for exceptions to rule for starting action execution.

If a behavior is not reentrant, then no more than one execution of it will exist at any given time. An invocation of a non-reentrant behavior does not start the behavior when the behavior is already executing. In this case, tokens control tokens are discarded, and data tokens collect at the input pins of the invocation action, if their upper bound is greater than one, or upstream otherwise. An invocation of a reentrant behavior will start a new execution of the behavior with newly arrived tokens, even if the behavior is already executing from tokens arriving at the invocation earlier.

Package ExtraStructuredActivities

If an exception occurs during the execution of an action, the execution of the action is abandoned and no regular output is generated by this action. If the action has an exception handler, it receives the exception object as a token. If the action has no exception handler, the exception propagates to the enclosing node and so on until it is caught by one of them. If an exception propagates out of a nested node (action, structured activity node, or activity), all tokens in the nested node are terminated. The data describing an exception is represented as an object of any class.

Package CompleteActivities

Streaming allows an action execution to take inputs and provide outputs while it is executing. During one execution, the action may consume multiple tokens on each streaming input and produce multiple tokens on each streaming output. See Parameter.

Local preconditions and postconditions are constraints that should hold when the execution starts and completes, respectively. They hold only at the point in the flow that they are specified, not globally for other invocations of the behavior at other places in the flow or on other diagrams. Compare to pre and postconditions on Behavior (in Activities). See semantic variations below for their effect on flow.

Bsp. Action im Activity Diagram

Semantic Variation Points

Package CompleteActivities

How local pre- and postconditions are enforced is determined by the implementation. For example, violations may be detected at compile time or runtime. The effect may be an error that stops the execution or just a warning, and so on. Since local pre and postconditions are modeler-defined constraints, violations do not mean that the semantics of the invocation is undefined as far as UML goes. They only mean the model or execution trace does not conform to the modeler's intention (although in most cases this indicates a serious modeling error that calls into question the validity of the model).

See variations in ActivityEdge and ObjectNode.

Notation

Use of action and activity notation is optional. A textual notation may be used instead.

Actions are notated as round-cornered rectangles. The name of the action or other description of it may appear in the symbol. See children of action for refinements.

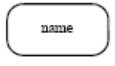


Figure 12.28 - Action

Package CompleteActivities

Local pre- and postconditions are shown as notes attached to the invocation with the keywords «localPrecondition» and «localPostcondition», respectively.

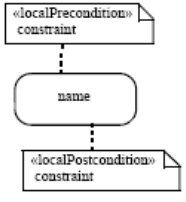


Figure 12.29 - Local pre- and postconditions

UML Superstructure Specification, v2.0 303

Erweiterung der UML

- Motivation für Erweiterungen der UML
  - Fehlendes Fachwissen ist großes (und teures) Problem bei der Entwicklung und Wartung von IT-Systemen
  - Lösungsansatz: Aufnahme von fachlichen Konzepten in SW-Modell
    - Dokumentiert fachliche Zusammenhänge
    - Ermöglicht im Rahmen der modellgetriebenen SW-Entwicklung die Codegenerierung bestimmter fachlicher Aspekte

Erweiterung der UML

- Problem
  - UML abstrahiert von jeder fachlichen Domäne
  - Fachliche (oder sehr spezielle technische) Konzepte sollen in UML beschrieben werden
  - Definition der (Domänen)-Konzepte benötigt
    1) Erklärender Text zu einem Diagramm → Ungeeignet
    2) Erweiterungen der UML-Notationselemente um fachliche Konzepte

- UML unterstützt mit sogenannten „Profilen“ die Anpassung auf Modellebene

UML-Profile

- UML-Profil
  - Spezialisierung von Standard UML-Elementen zu konkreten Metatypen
  - Ein Profil kann zu einem Modell hinzugefügt werden und steht dann im gesamten Modell zur Verfügung
  - Verschiedene Profile für verschiedene Anwendungsdomänen
  - Einige Profile sind bereits vordefiniert und bei der OMG verfügbar
- Definition eines Profils
  - Paket von Stereotypen und Tagged Values
  - Klassendiagramm definiert Beziehungen zwischen neuem Stereotyp und dem zu beschreibenden Element definiert
  - Für die Definition neuer Stereotypen ist eine Grundkenntnis des Metamodells erforderlich

UML-Profile - Beispiel

- Vordefinierte Stereotypen im UML-Standard (Auszug)

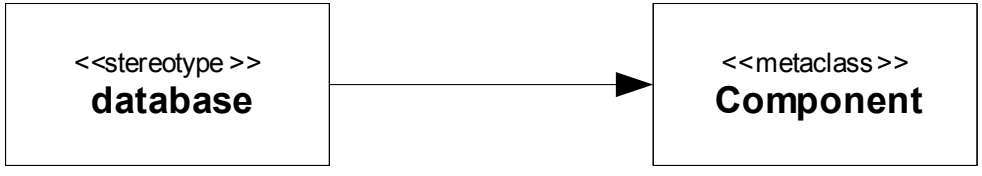
Name	Language Unit	Applies to
«document»	Deployments:: Artifacts	Artifact
«entity»	Components:: BasicComponents	Component
«executable»	Deployments:: Artifacts	Artifact
«file»	Deployments:: Artifacts	Artifact
«implement»	Components:: BasicComponents	Component
«library»	Deployments:: Artifacts	Artifact
«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 - Stereotypen

Definition von Stereotypen

- Für die Definition neuer Stereotypen sind Kenntnisse über das Metamodell erforderlich

- Konkretes Beispiel einer Definition:

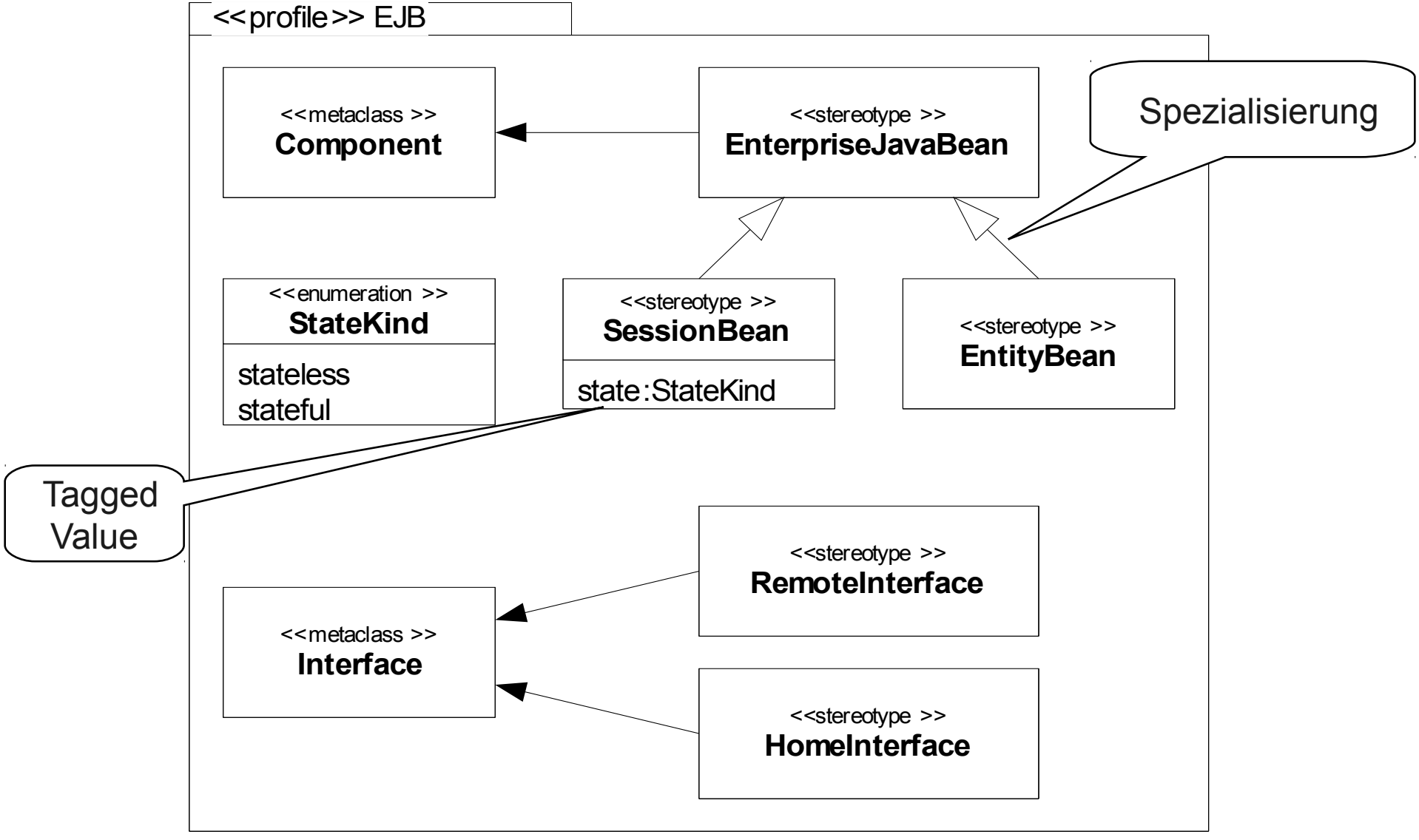


- Spätere Verwendung im Komponentendiagramm:

UML-Profile - Stereotypen

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

UML-Profile - Beispiel



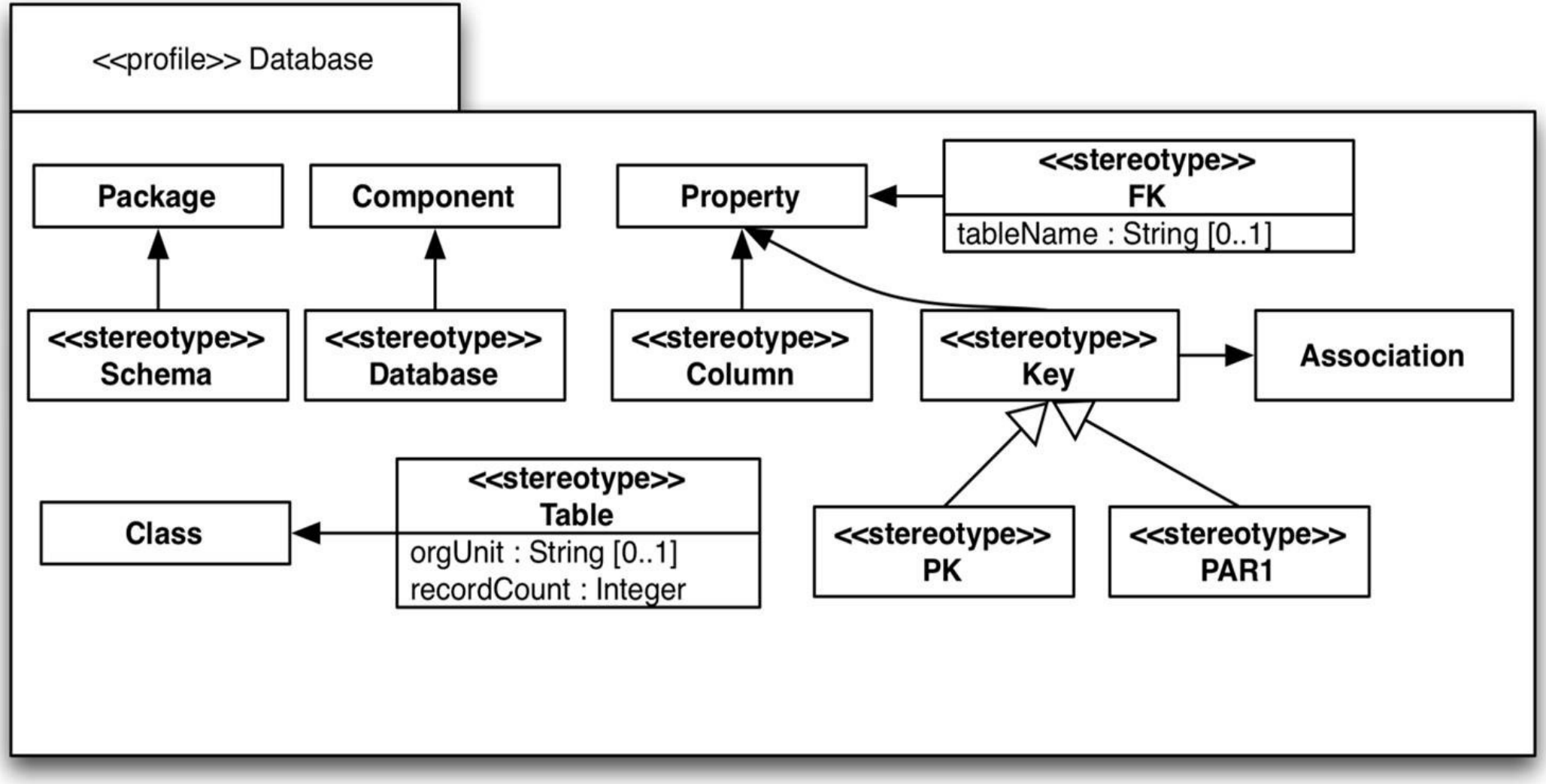
UML-Profile

- Semantik
  - Die Semantik eines Stereotyps wird durch einen Text „definiert“
  - Wichtig für die richtige Auswahl der zu verwendenden Stereotypen und für das Verständnis beim Lesen eines Diagramms
- Standardisierte Profile der UML
  - Allgemeine Profile für besondere Einsatzdomänen:
    - Real-time UML, UMLsec (s. Vorlesung MGSE im SS 2012)
    - 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:
  - Modellierung als UML Komponenten mit dem Stereotyp <<Database>>
  - Schema: Darstellung über UML Paket Notationselement mit dem Stereotyp <<Schema>>.
  - Tabelle: Darstellung als UML Klasse mit dem Stereotyp <<Table>>
  - Spalten: Darstellung als Attribute der als <<Table>> markierten Klasse. Die Attribute werden dem Stereotyp <<Column>> versehen.
  - Primärschlüssel: Darstellung über den Stereotyp <<PK>> für den Primärschlüssel und <<PAR1>> für (zusammengesetzte) Sekundärschlüssel
  - Fremdschlüssel: Darstellung als Attribut markiert mit dem Stereotyp <<FK>>

Beispiel technisches Datenmodell



Beispiel technisches Datenmodell

Zusammenfassung:

- Import der geerbten UML Metamodellklassen „Package“, „Component“, „Class“, „Property“ und „Association“ aus deren Paketen „Classes“ und „Components“
- Die Beziehungen mit ausgefüllten Spitzen sind „Extensions“. Sie verlaufen von Stereotyp zur Metaklasse aus dem zu ergänzenden Metamodell
- Stereotype können für eine Metaklasse optional oder zwingend sein.
- Hat ein Stereotyp Attribute muss im Modell beim entsprechenden Modellelement dieses Attribut belegt werden. (Bsp.: Stereotyp „FK“ Attribut „tableName“)

Beispiel technisches Datenmodell

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

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

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

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

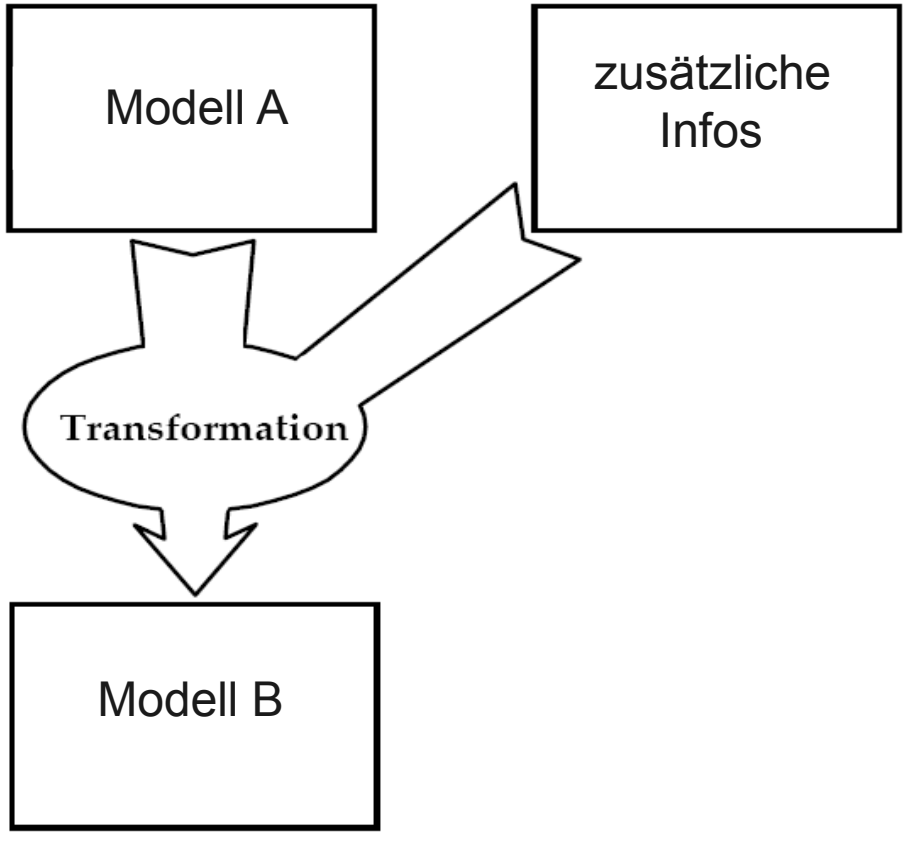
Beispiel für ein technisches Datenmodell modelliert mit dem definierten UML Profile

3.2 Modell-basierte Software-entwicklung

- Motivation & Einführung
- Semantik
- Semantik im UML Metamodell
- **Modelltransformation**

Modelltransformation

- Model A und zusätzliche Information werden durch die Anwendung einer Transformation in das Zielmodell B überführt
- Hintereinanderausführung mehrerer Transformationen sind möglich
- Prinzipiell könne alle Modelle die man definieren kann Quell- oder/und Zielmodell sein.
- Viel Know-How steckt im Generator



Quelle: MDA Guide Version 1.0.1

Modelltransformation

- Modelltransformation bezeichnet berechenbare Abbildungen eines Quellmodells in ein Zielmodell [BBG05].
- Ziel: Umwandlung eines Modells einer Abstraktionsebene in ein Modell einer anderen Abstraktionsebene
- Horizontale Transformation
  - Inhaltliche Weiterentwicklung eines Modells durch Transformation
- Vertikale Transformation
  - Transformation eines Modells auf eine technologiespezifischere Ebene

Modelltransformation

Einsatz von Modelltransformationen

- Modell-Modell Transformation
  - Bezeichnet die Überführung der Inhalte eines Modells in ein anderes Modell
  - Beide Modelle basieren oft auf verschiedenen Metamodellen

- Modell-Code Transformation
  - Bezeichnet die Überführung der Inhalte eines Modells in Quellcode

Unterscheidung von Modell-Modell- und Modell-Code-Transformationen nicht trennscharf, da Code auch nur ein Modell ist.

Modelltransformation

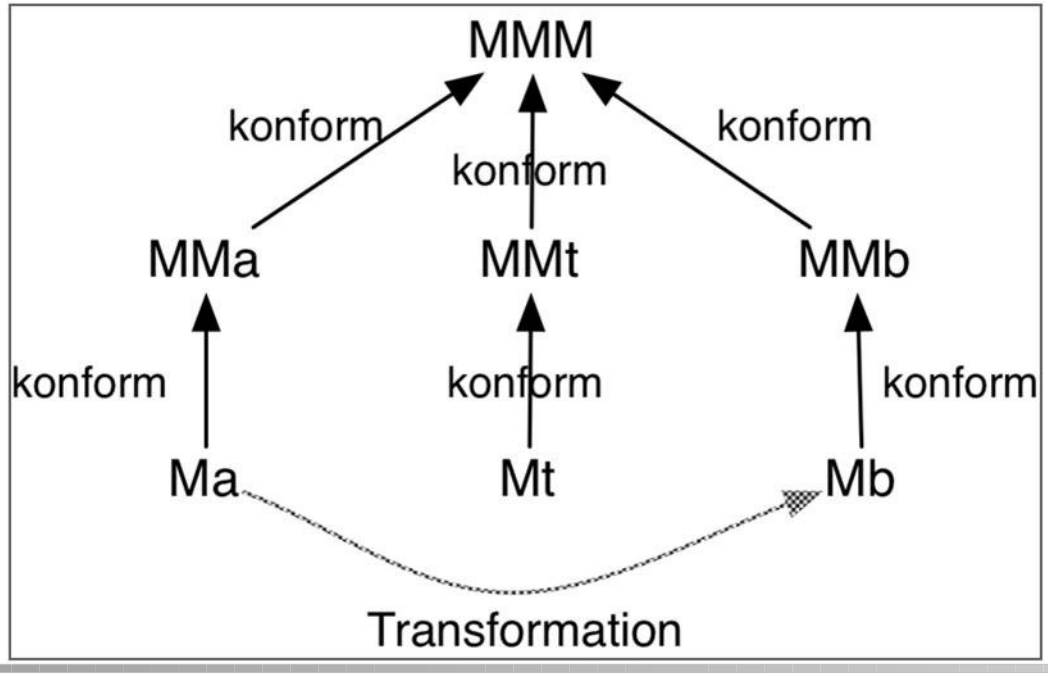
Einsatz von Modelltransformationen

- Modell-Modifikation
  - Anreicherung mit zusätzlicher Information oder Veränderung vorhandener Information eines Modells
  - Bsp.: In einem Zustandsautomaten erhält jeder Zustand eine Transition „Not-Aus“ zum Zustand „Aus“.

- Modell-Weaving
  - Verweben und Verlinken von Modellen
  - Ergebnis: Weaving-Modell drückt Beziehungen zwischen Modellen aus

Modelltransformation und Metamodell

- Die dargestellten Modelle „Ma“, „Mt“ und „Mb“ sind konform zum jeweiligen Metamodell „Mma“, „MMt“ und „MMb
- Eine Modelltransformation von „Ma“ nach „Mb“ ist eine auf den Metamodellen definierte Abbildung
- Die Abbildung erlaubt Instanzen von „MMa“ in Instanzen von „MMb“ zu transformieren



Modelltransformationssprachen

- Modelltransformationssprachen
  - Mittel zur Beschreibung von Abbildungen (Transformationsregeln) von Instanzen eines Metamodells in ein anderes Metamodell
  - Deklarative oder imperative Sprachkonstrukte

- Deklarative Transformationssprachen
  - Beschreibung der Transformationen durch Regeln, die mit entsprechenden Vor- und Nachbedingungen spezifiziert sind
  - Viele deklarative Transformationsansätze sind durch Graphentransformation realisierbar (wenn Modell als Graph)

- Imperative Transformationssprachen
  - Beschreibung der Transformation durch Sequenz von Aktionen

Modelltransformationssprachen

- QVT (Query View Transformation)
  - Standard der OMG
  - Besteht aus 2 Transformationssprachen:
    - QVT Relations
      - Deklarativ
      - Kann bidirektional und inkrementell transformieren
      - Eine der ersten Implementierungen ist ModelMorf
    - QVT Operational Mapping
      - Imperativ
      - Kann Relations verwenden
      - Implementierung: Smart QVT
  - Metamodelle müssen mittels der MOF beschrieben sein

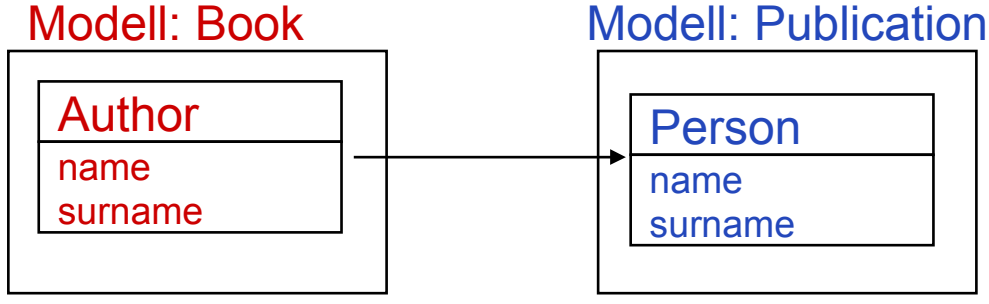
Modelltransformationssprachen

- 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 (sowohl deklarativ, als auch imperativ)
  - Verwendet OCL um Abfragen auf Modellen auszuführen
  - Transformation besteht aus einem Satz von Regeln, die Elemente des Ausgangsmodells in Elemente des Zielmodells überführen
  - Kann mit verschiedenen Arten von Metamodellen arbeiten

Modelltransformationssprachen

Beispiel ATL:

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



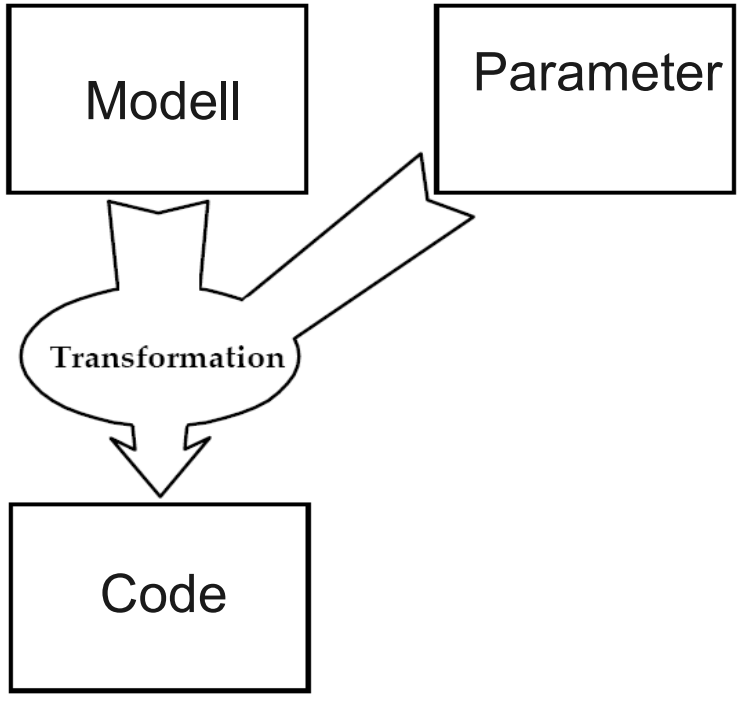
Das Ausgangsmodell „IN“, welches konform zum Metamodell „Book“ ist wird in Modell „OUT“ (konform zu „Publication“) überführt.
Die Regel überführt Elemente vom Typ „Author“ in Elemente vom Typ „Person“.

Modelltransformationssprachen

- Tefkat
  - Entwickelt an der Universität von Queensland, Australien
  - Open Source
  - Deklarative Sprache
  - Nicht bidirektional
  - Werkzeugunterstützung in Eclipse IDE (Editor, Debugger)
  - Metamodelle müssen in Ecore (EMF) beschrieben werden
  - Beziehung zwischen Quell- und Zielmodellen wird über Regeln hergestellt
  - Wiederverwendung von Code-Teilen über Definition von Pattern und Templates
  - Sparsame Dokumentation

Codegenerierung

- Ziel: Generierung von Programmcode aus einem Modell
- Involviert einen Generator
  - Der Generator erzeugt Programmcode für eine spezifische Anwendungs- oder Programmklasse
  - Generator kapselt ein generisches Programmmodell (Klassen von Programmen)
  - Konkret erzeugter Code abhängig von:
    Model
    Transformationslogik
    Parameter

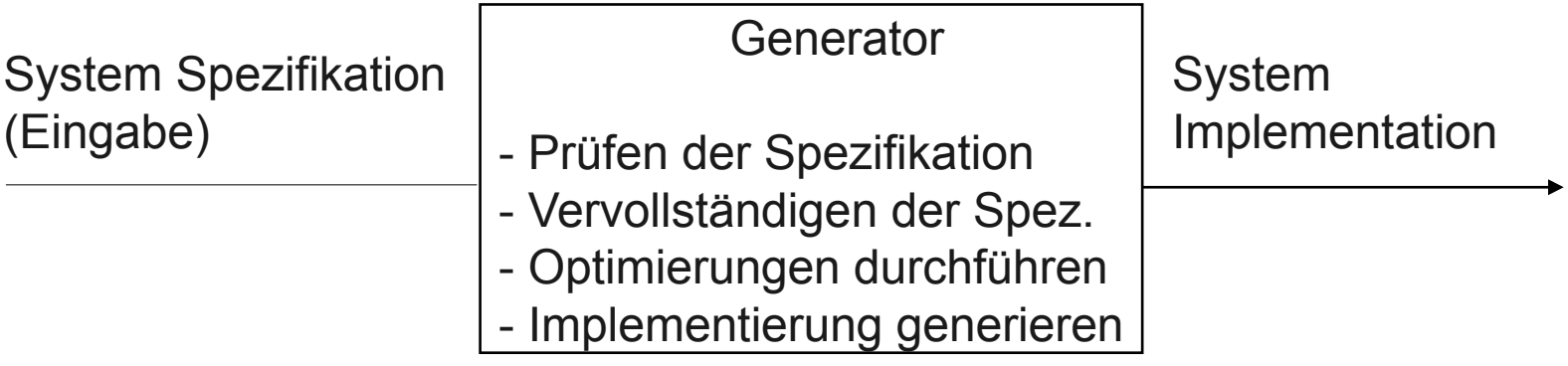


Codegenerierung

- Grober Ablauf eines Generatorlaufs
  - Einlesen der Eingabespezifikation (bspw. UML Modell, Text, etc)
  - Einlesen der Parameter
  - Anwenden der Transformationsregeln auf Eingabespezifikation unter Berücksichtigung der Parameter
  - Ausgabe von Programmcode
- Phasen der Codegenerierung nach [CKE00]
  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
  - Akzeptieren eine abstrakte Beschreibung eines Software-Artefakts als Eingabe und generieren dessen Implementation
- Software-Artefakt
  - Umfassendes Softwaresystem
  - Komponente
  - Klasse
  - Methode / Prozedur



Codegenerierung

- Codegenerierung arbeitet mit *variabilisiertem* Programmcode
  = Programmcode mit Variationspunkten
- Durch Parametrierung wird der Programmcode
  eindeutig ausgeprägt
- Aufwand der Generatorentwicklung nicht trivial
- Eignung: für Lösungen mit entsprechend großer Zahl von Variationen in der 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

Modellbasierte Szenarien

- Kommunikation mit Modellen
  - Analysemodelle: Fachexperte und SW-Architekt
  - Entwurfsmodelle: SW-Architekt und SW-Entwickler
  - Implementierungsmodelle: Dokumentation von Systemen
- Spezifizieren mit Modellen
  - Verfeinerung und Spezialisierung des Modells in der Entwurfsphase
  - Verbindliche Spezifikation zwischen Auftraggeber und IT-Firma
  - Vorlage zur Implementierung durch den SW-Entwickler
  - Generieren von Dokumenten aus Modell
- Testen mit Modellen
  - Ableiten von Integrations- und Abnahmetests aus fachlichem Modell
  - Automatische Generierung von technischen Testfällen (JUnit)

Modellbasierte Szenarien

- Simulieren mit Modellen
  - Simulation von Dynamik
  - Systemverhalten: z.B. nebenläufige Prozesse (Petri-Netz)
  - Dynamische Architekturen: Untersuchung von externen Einflüssen auf die Konfiguration dynamischer Architekturen (Projekt Mobco)

- Programmieren mit Modellen
  - Modellieren statt Programmieren: Generierung von Teilen des Programmcodes
  - Generatoren für konkrete technische Bereiche (DB-Schema aus XML oder EJB-Code)
  - Generatoren für komplette Schichten eines Systems (Persistenzschicht samt Konfigurationen und Datenobjekte)
  - Generatoren für spezielle aber schichtenübergreifende Bereiche eines Systems (UI, Validerung und Persistenz aus XML-Datei)
  - Generatoren für speziellen Projekzweck (z.B. Kapselung kundenspezifischer Besonderheiten, Produktlinien)

Zusammenfassung und Ausblick

Wir haben in diesem Kapitel eine Einführung und Überblick in das Thema modellbasierte Software-Entwicklung behandelt. Wichtige Punkte:

- Modellbegriff und SW-Modelle
- XML-Schema (XSD)
- Semantik von Modellen
- Modell und Metamodell der UML (Aufbau, Semantik)

Im nächsten Kapitel werden wir uns darauf aufbauend mit einem Teil der UML beschäftigen (nämlich der Object Constraint Language, OCL), der die Qualitätssicherung im Kontext der modell-basierten Software-Entwicklung unterstützt, indem Bedingungen an die Ausführung der modellierten System-Teile formuliert und analysiert werden können (vergleichbar mit der Verwendung von Assertions in Programmquellcode).

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