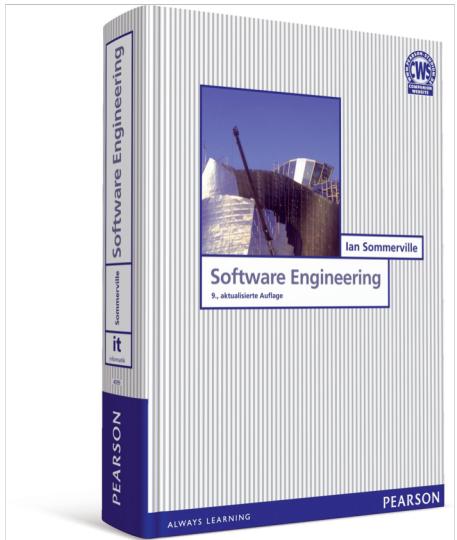


Teil 3.1:

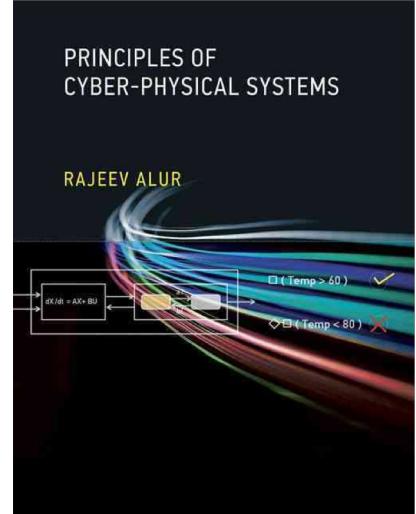
Domänenspezifische Sprachen /

Modellierung von Verhalten

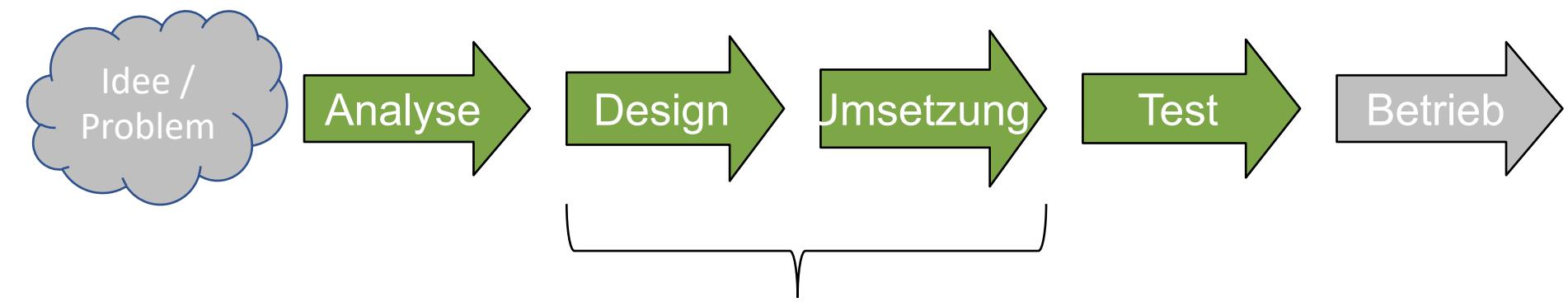
Literatur



- Ian Sommerville, Software Engineering, 9. überarbeitete Ausgabe, Pearson 2012.
- Stahl und Völter, Modellgetriebene Softwareentwicklung, dpunkt.verlag GmbH, 2005.
- Alur, Principles of Cyber-Physical Systems, 2015.



Einordnung

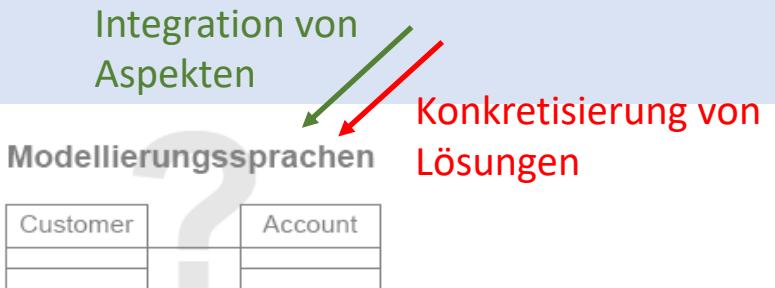


Agenda

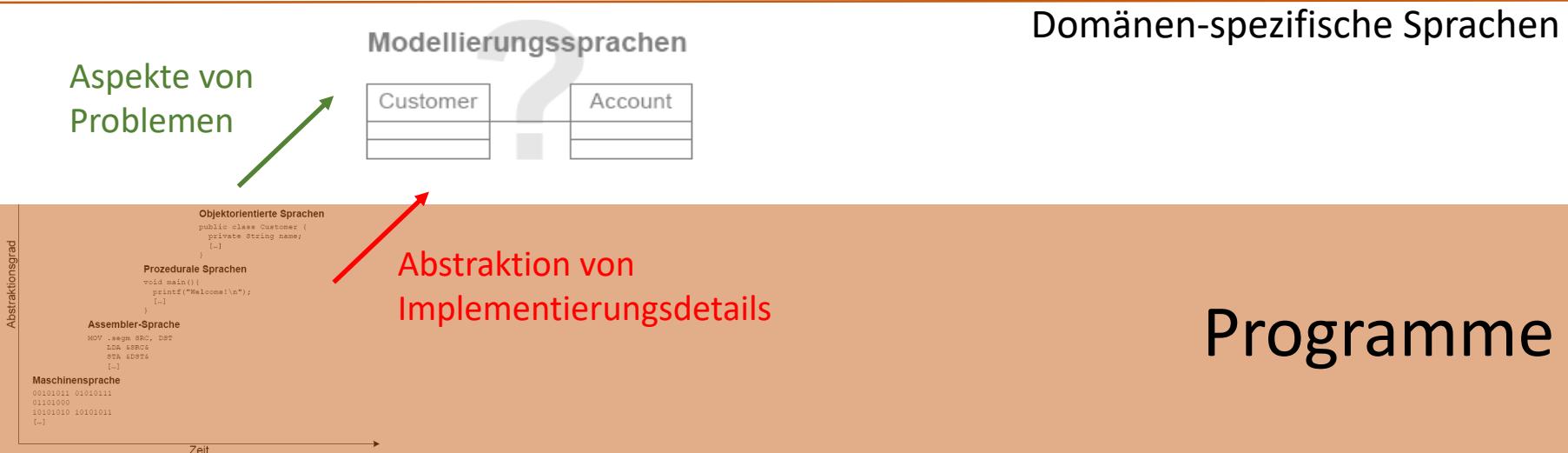
- Kurze Wiederholung: Meta-Modellierung
- Modellierung von Verhalten
 - Blockschaltdiagramme / Boolesche Logik
 - Erweiterte Endliche Automaten
- Domänenspezifische Sprachen
 - Beispiel: EMF
 - Code Generierung

Von Anforderungen zu Programmen

Anforderungen



Modellgetriebene Software-Entwicklung



Metamodellierung

Meta-Modellierung = Erstellung einer Modellierungssprache wird zum Gegenstandsbereich

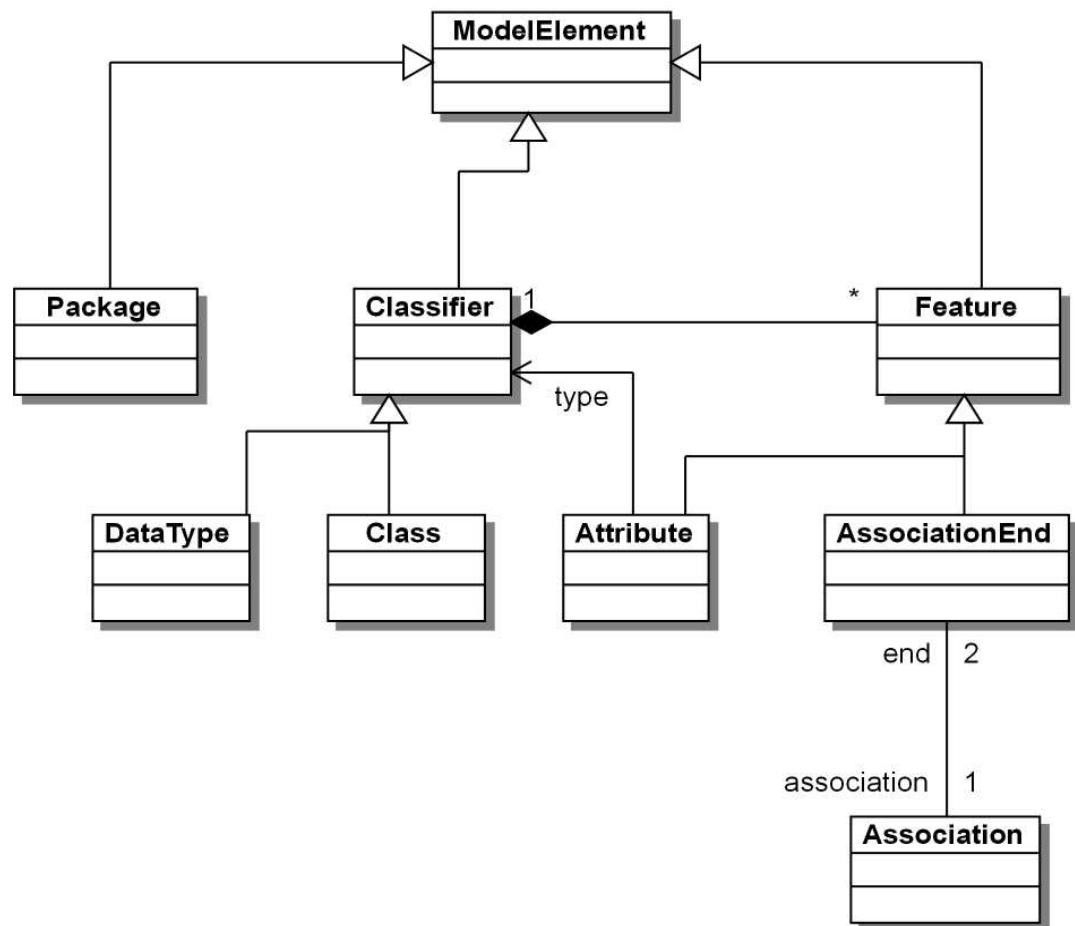
Bestandteile:

- Syntax C (konkret) und Syntax A (abstrakt)
- Semantik S
- Syntaktische Abbildung $M_s : A \rightarrow C$ (**bestimmt, wie man etwas beschreibt**)
- Semantische Abbildung $M_c : A \rightarrow S$ (**bestimmt Übersetzung**)

Semantik legt Bedeutung von Symbolen und deren Kombinationen fest.
Bedeutung definiert in begrenztem Bereich (semantische Domäne)

Meta Object Facility (MOF)

MOF Metamodel (vereinfacht)



Selbstreferenzierendes
Metamodell =
Metamodell
beschreibt sich selbst

Standard MOF 2.4.2

- ISO/IEC 19508:2014
Information
technology - Meta
Object Facility (MOF)

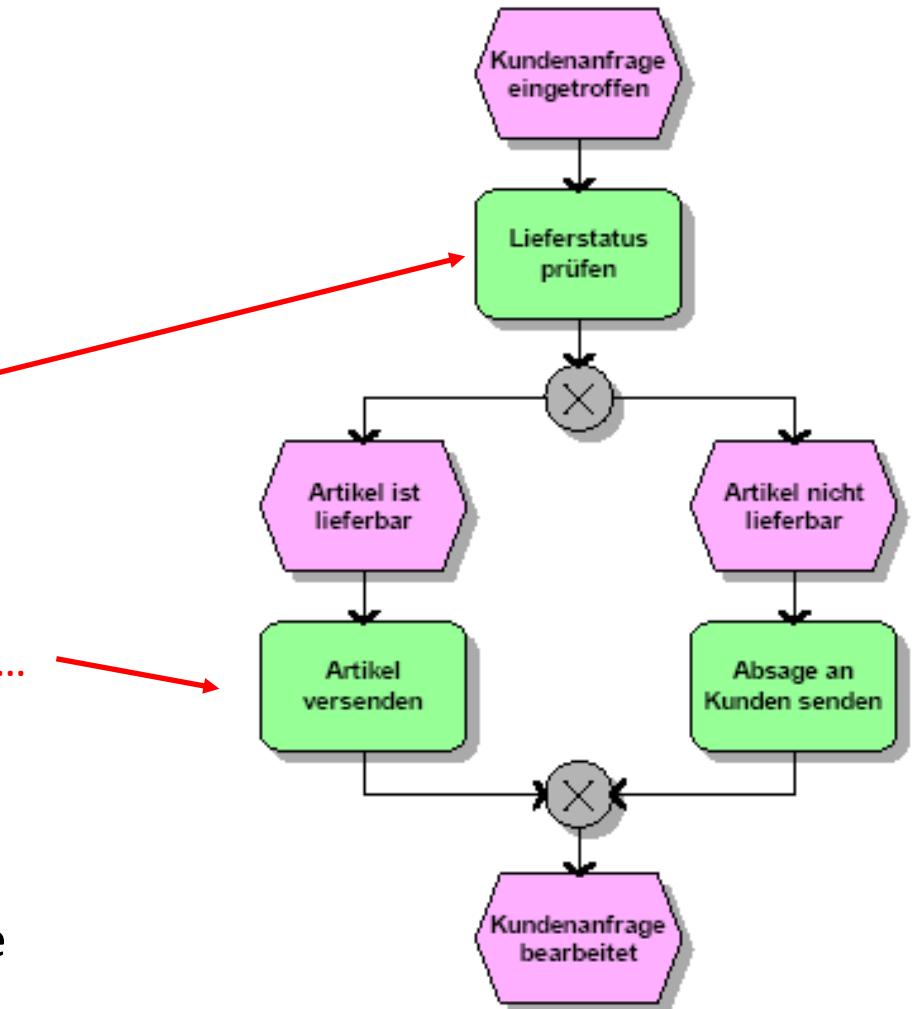
Beispiel: Event-Gesteuerte Prozessketten

Beispiel: Event-Gesteuerte Prozessketten

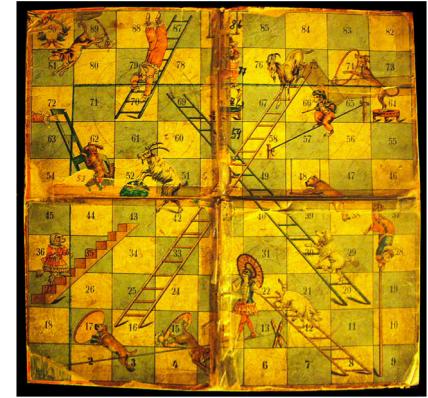
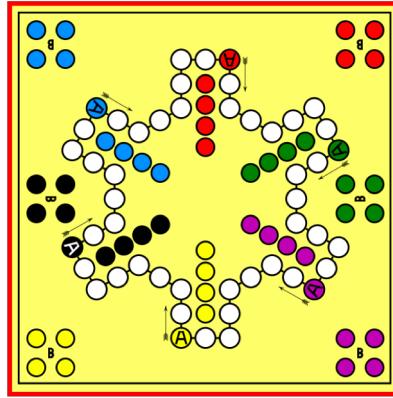
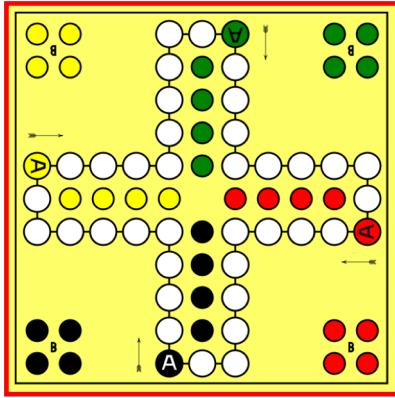
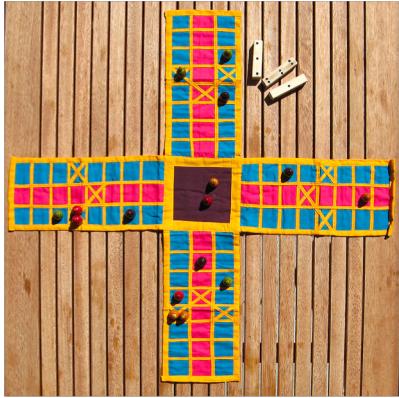
Anfrage in mehreren Datenbanken

Artikel aus Lager holen,
verpacken, Versandadresse laden, ...

- Modellierungssprache Prozesse
- Einzelne Schritte könnten komplexe Implementierungen haben



Beispiel-Domäne: Mensch Ärger Dich Nicht Varianten



- Verschiedene „Mensch Ärger Dich Nicht“ ähnliche Spiele oder Varianten
- Automatische Übersetzung / Ausführung der Spielbretter

Foto: Micha L. Rieser
(<https://commons.wikimedia.org/wiki/File:Pachisi-real.jpg>)

Modell-Hierarchie (Spielbrett)

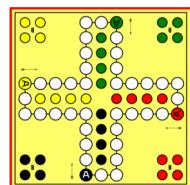
Konkrete Syntax:
DSL für Metamodell

Konkrete Syntax:
Domänenspezifische Sprache

Brettbeschreibung in DSL

Modellierung von
Elementen und Beziehungen

Elemente / Beziehungen
für Spielbretter



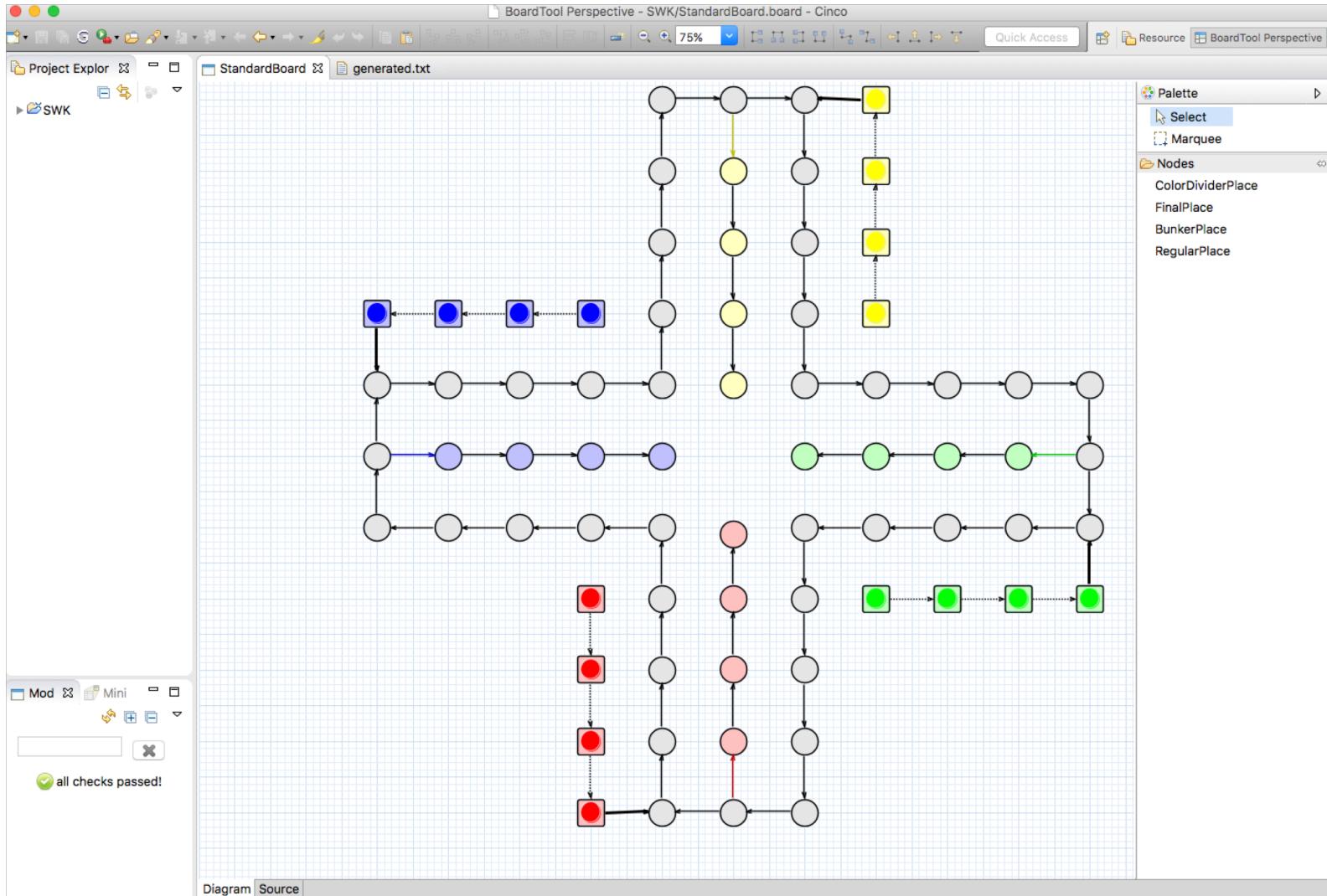
M3: Metametamodell

M2: Metamodell

**M1: Modell des
Spielbretts**

M0: Konkretes Spiel

Modell in konkreter Syntax des Metamodells



Metamodell in konkreter Syntax des Metametamodells

Modellelemente

Beziehungen

```
abstract node Place {
    incomingEdges(Transition[0,*])
}

@style(regularPlace)
node ColorDividerPlace extends Place {
    outgoingEdges (Transition[2,2], RegularTransition[1,1], ColorTransition[1,1])
}

abstract node ColoredPlace extends Place {
    attr Color as placeColor := NONE
}

@style(finalPlace)
node FinalPlace extends ColoredPlace {
    outgoingEdges(RegularTransition[0,1])
}

@style(bunkerPlace)
node BunkerPlace extends ColoredPlace {
    outgoingEdges({AutoTransition,OutOfBunkerTransition}[0,1])
}

@style(regularPlace)
node RegularPlace extends Place {
    outgoingEdges(RegularTransition[1,1])
}

| abstract edge Transition {
}

@style(simpleArrow)
edge RegularTransition extends Transition {
}

@style(outOfBunkerTransition)
edge OutOfBunkerTransition extends Transition {
}

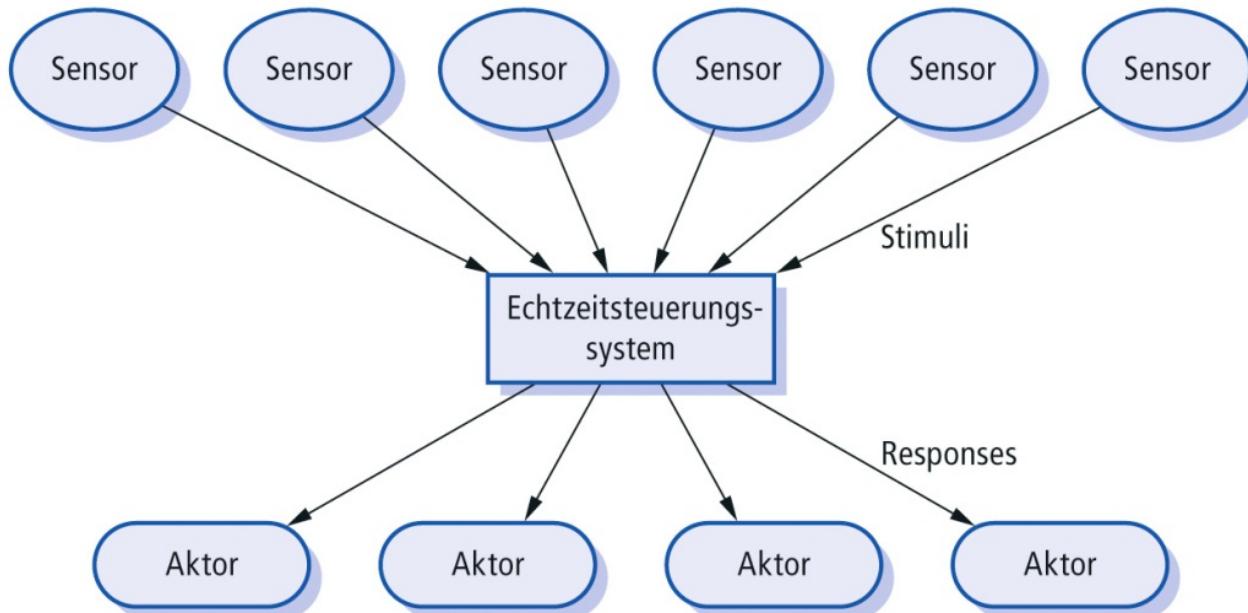
@style(autoTransition)
edge AutoTransition extends Transition {
}

@style(colorTransition)
edge ColorTransition extends Transition {
    attr Color as guard
}
```

Agenda

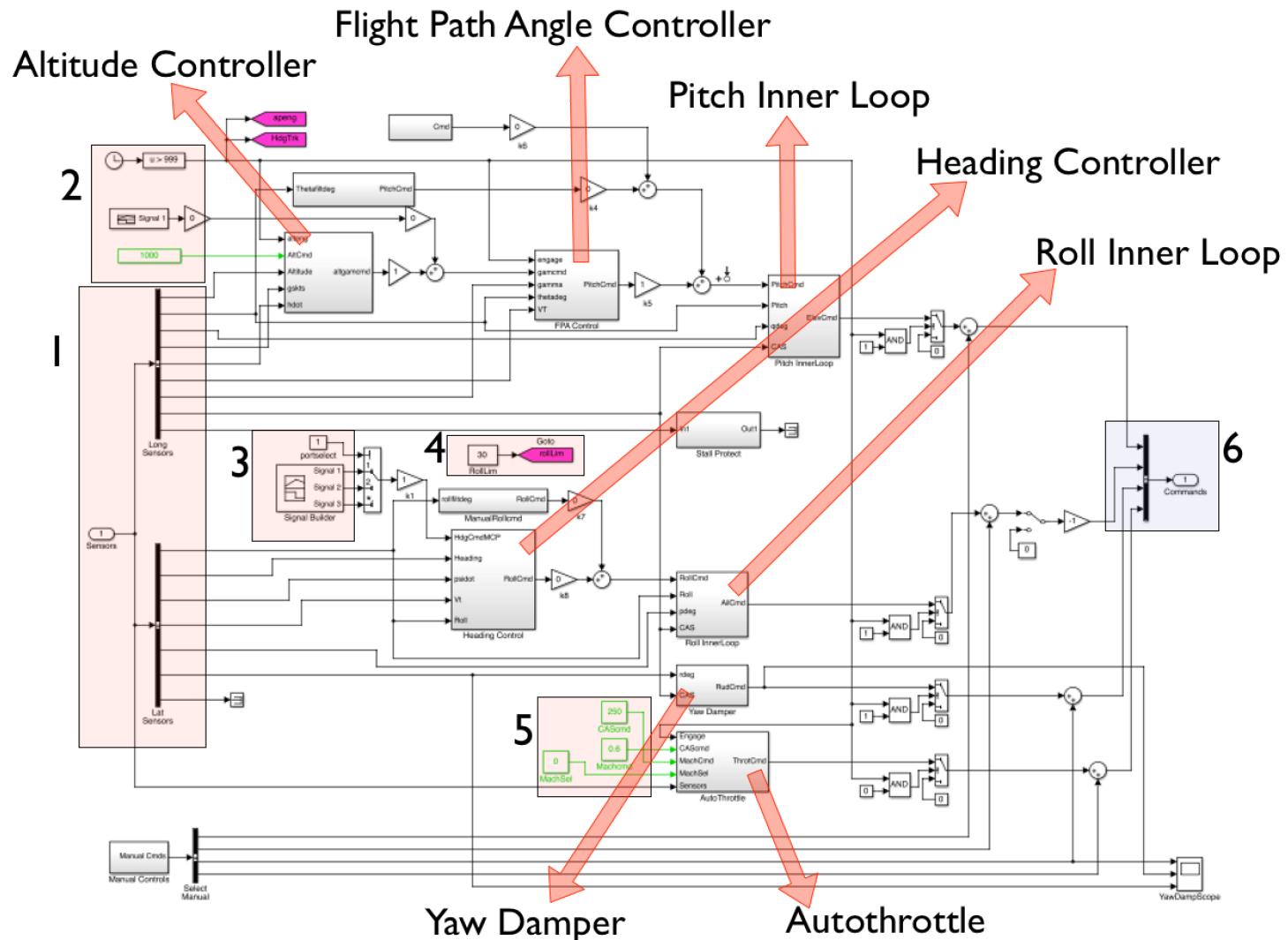
- Kurze Wiederholung: Meta-Modellierung
- Modellierung von Verhalten
 - Blockschaltdiagramme / Boolesche Logik
 - Erweiterte Endliche Automaten
- Domänenspezifische Sprachen
 - Beispiel: EMF
 - Code Generierung

Eingebettete Reaktive Systeme

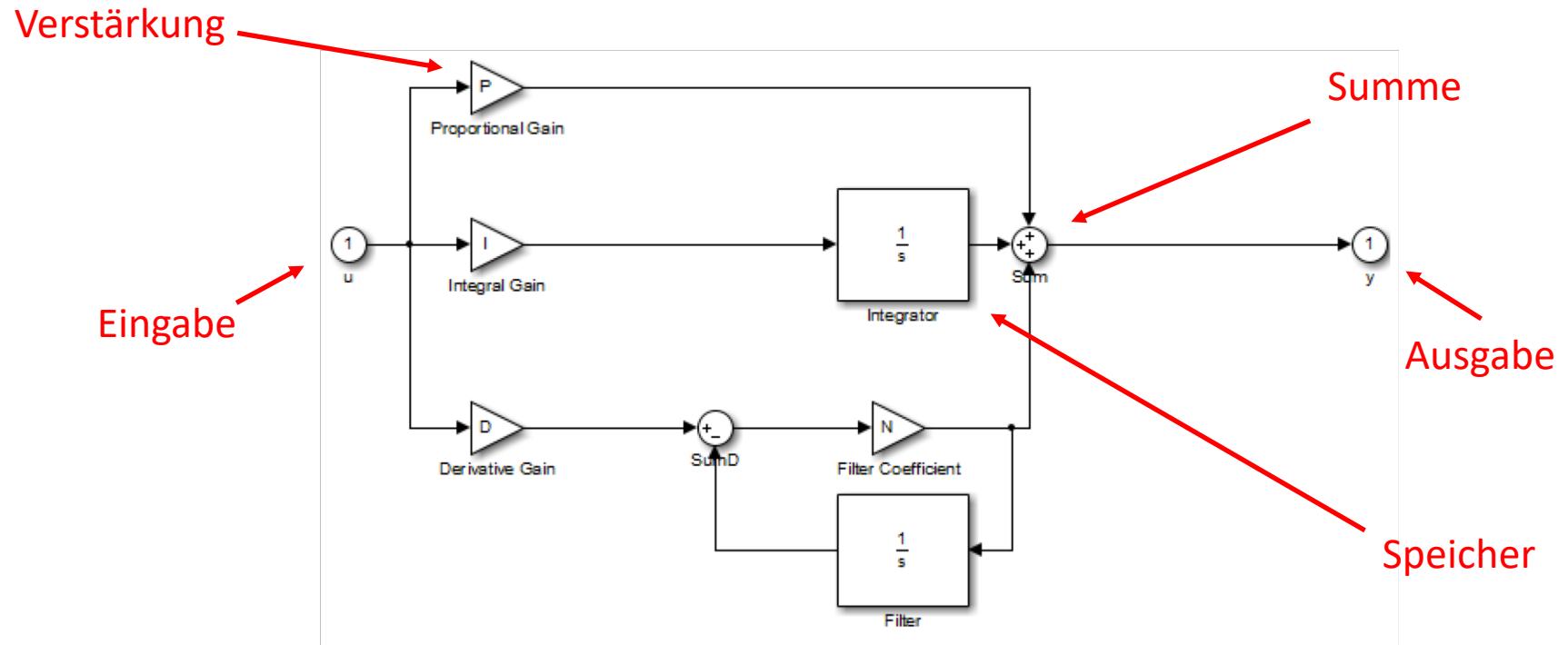


- Reagieren auf Eingaben der Umgebung
- Event-basiert oder Zeit-basiert
- Domäne in der Modellierungssprachen verbreitet sind
- Domäne in der oft Ingenieure an Software beteiligt sind

Beispiel: Autopilot im Flugzeug



Regelungstechnische Systeme



Blockschaltdiagramm:

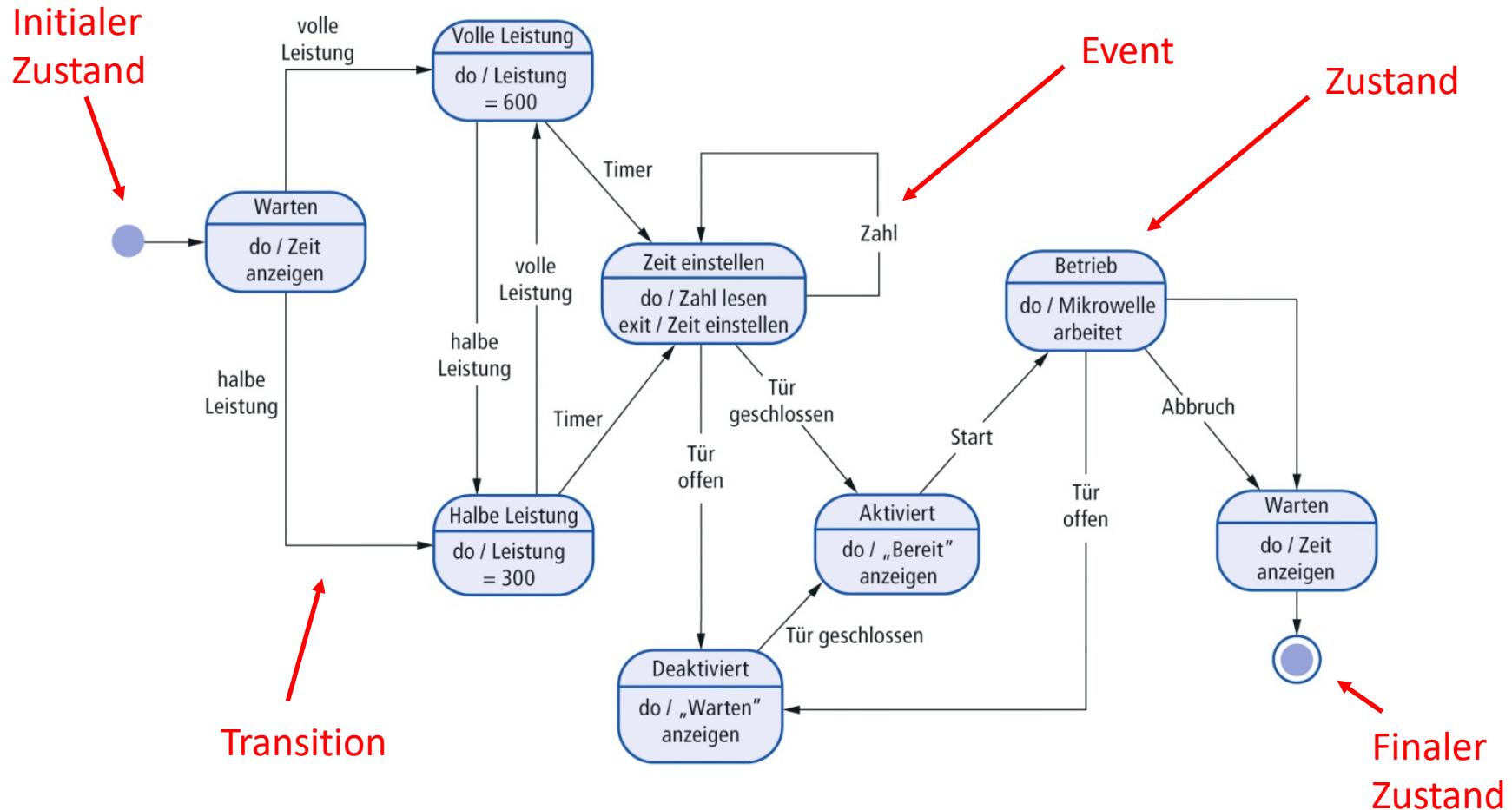
- Kanten: Datenfluss
- Boxen etc.: Operationen auf Daten
- U.a. Arithmetische und Boolesche Signale

Schaltbild: PID Regler

Ausführungssemantik

- Arbeiten auf Strömen von Datenwerten
- Aufruf oft periodisch (z.B. alle 20 Millisekunden)
- Bei jedem Aufruf werden aus Speicher und Eingaben Ausgaben berechnet
- Beispiel: Regelung der Geschwindigkeit im Auto

Beispiel: Event-gesteuertes System



Gemischtes Beispiel: Alarmanlage

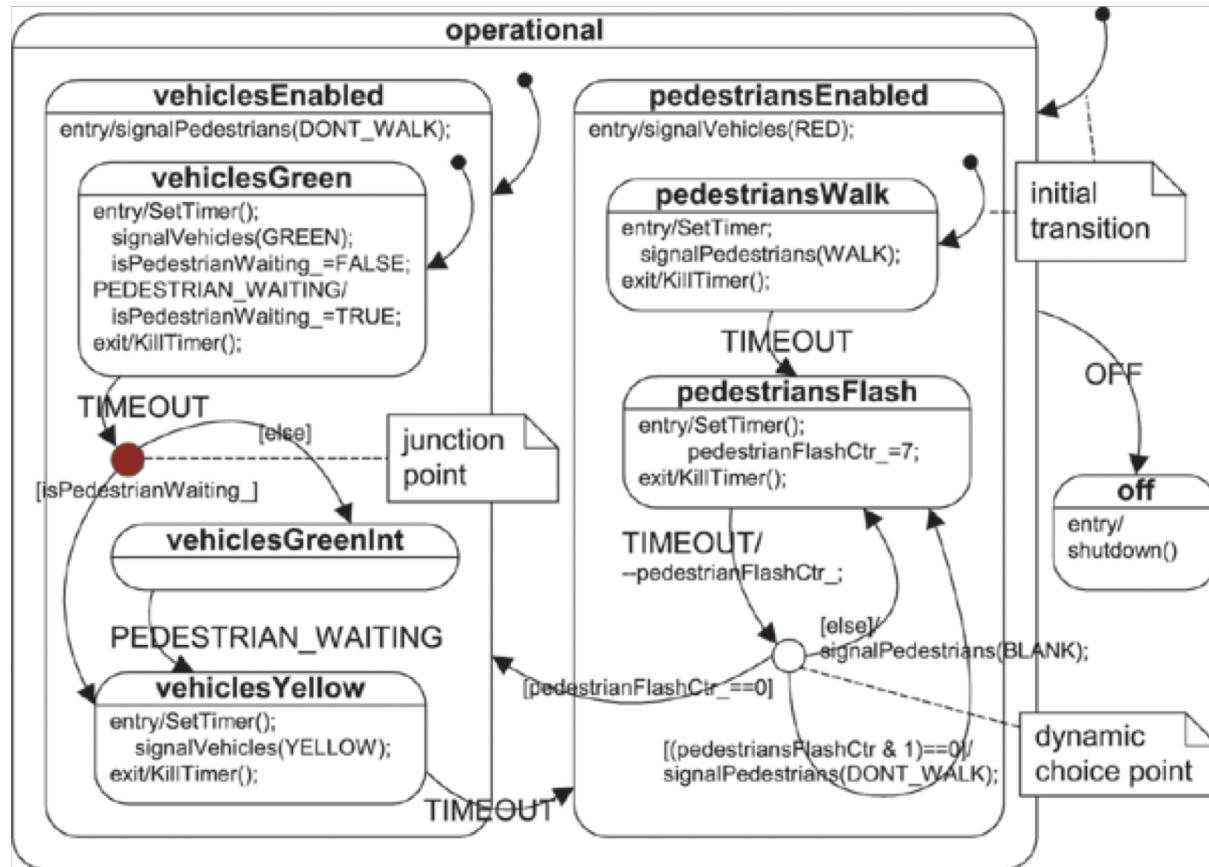
Stimulus	Response
Ein Sensor positiv	Löst Alarm aus; schaltet die Lampen in der Nähe des positiven Sensors ein.
Zwei oder mehr Sensoren positiv	Löst Alarm aus; schaltet die Lampen in der Nähe der positiven Sensoren ein; ruft die Polizei zu dem Ort des potenziellen Einbruchs.
Spannungsabfall zwischen 10% und 20%	Schaltet die Reservebatterie ein; führt ein Stromversorgungstest durch.
Spannungsabfall von mehr als 20%	Schaltet die Reservebatterie ein; löst Alarm aus; ruft die Polizei; führt ein Stromversorgungstest durch.
Ausfall der Stromversorgung	Ruft den Servicetechniker.
Sensorausfall	Ruft den Servicetechniker.
Konsolenpanikschalter positiv	Löst Alarm aus; schaltet die Lampen um die Konsole ein; ruft Polizei.
Alarm löschen	Schaltet alle aktiven Alarme aus; schaltet alle Lampen aus, die eingeschaltet wurden.

Diskussion

- Eingeschränkte Aspekte des Verhaltens (vgl. DSLs!)
 - Blockschaltdiagramme: Datenfluss
 - Automaten: Kontrollfluss

- ⇒ Semantik mathematisch definierbar
- ⇒ Wichtig in sicherheitskritischen Domänen
- ⇒ Code-Generierung möglich

Negativbeispiel: UML State Chart



Es gibt bis heute keine akzeptierte vollständige Festlegung der Semantik von State Charts

Blockschaltdiagramme / Erweiterte Endliche Automaten

Siehe Extra Foliensatz ...

Zusammenfassung

- Modellierungssprachen für Verhalten haben sich im Bereich eingebetteter Systeme durchgesetzt
- Modellierung von Aspekten (Kontrollfluss / Datenfluss)
- Herausforderung: Saubere Definition von Semantik

Blockschaltdiagramme

- Ohne Speicher: Abbildung auf logische Ausdrücke mit Variablen
- Semantik: Auswertung von Ausdrücken unter Belegung von Variablen

Erweiterte endliche Automaten

- Kombination von Ausdrücken und endlichen Automaten
- Semantik: Transitionssysteme

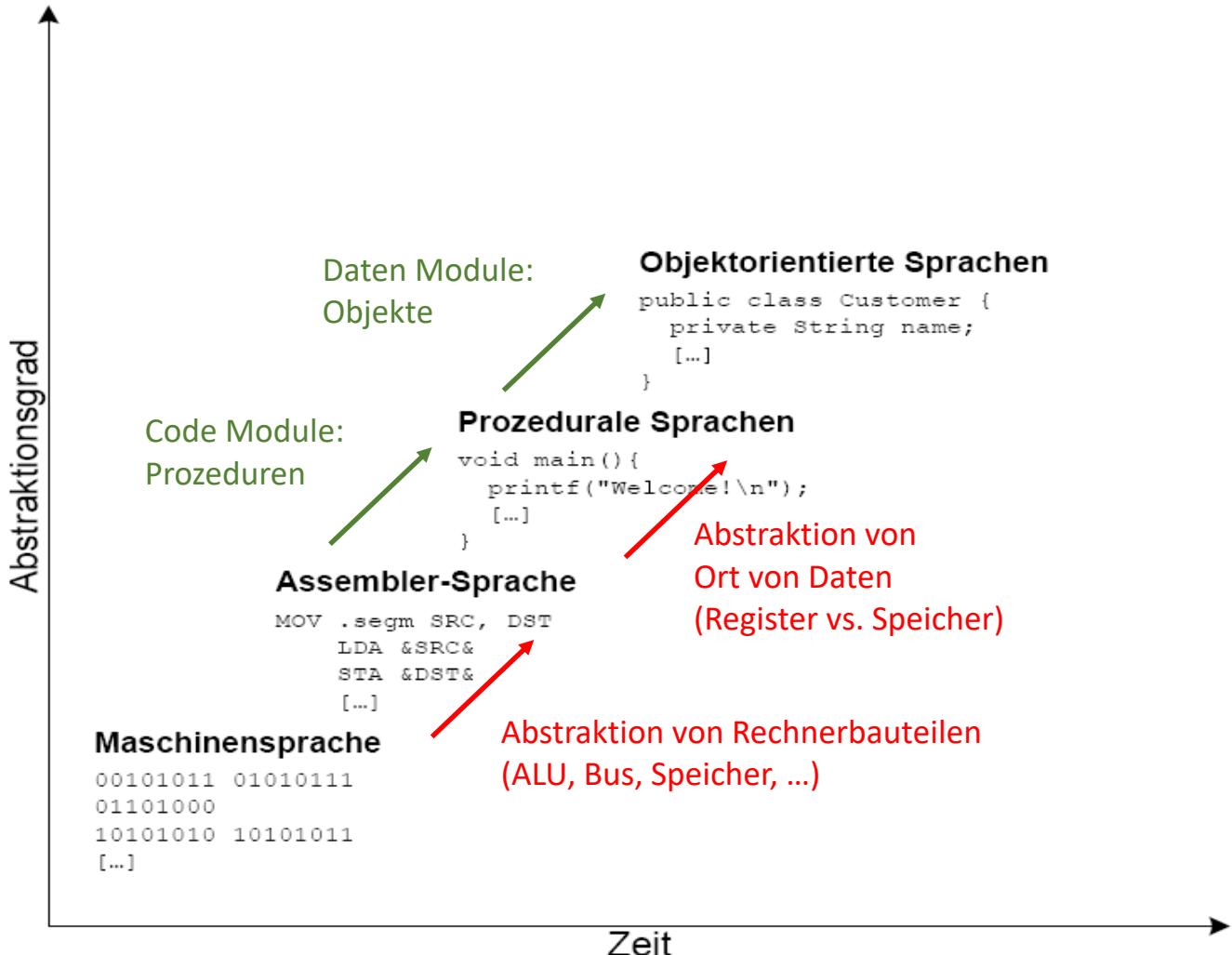
Agenda

- Kurze Wiederholung: Meta-Modellierung
- Modellierung von Verhalten
 - Blockschaltdiagramme / Boolesche Logik
 - Erweiterte Endliche Automaten
- Domänenspezifische Sprachen
 - Beispiel: EMF
 - Code Generierung

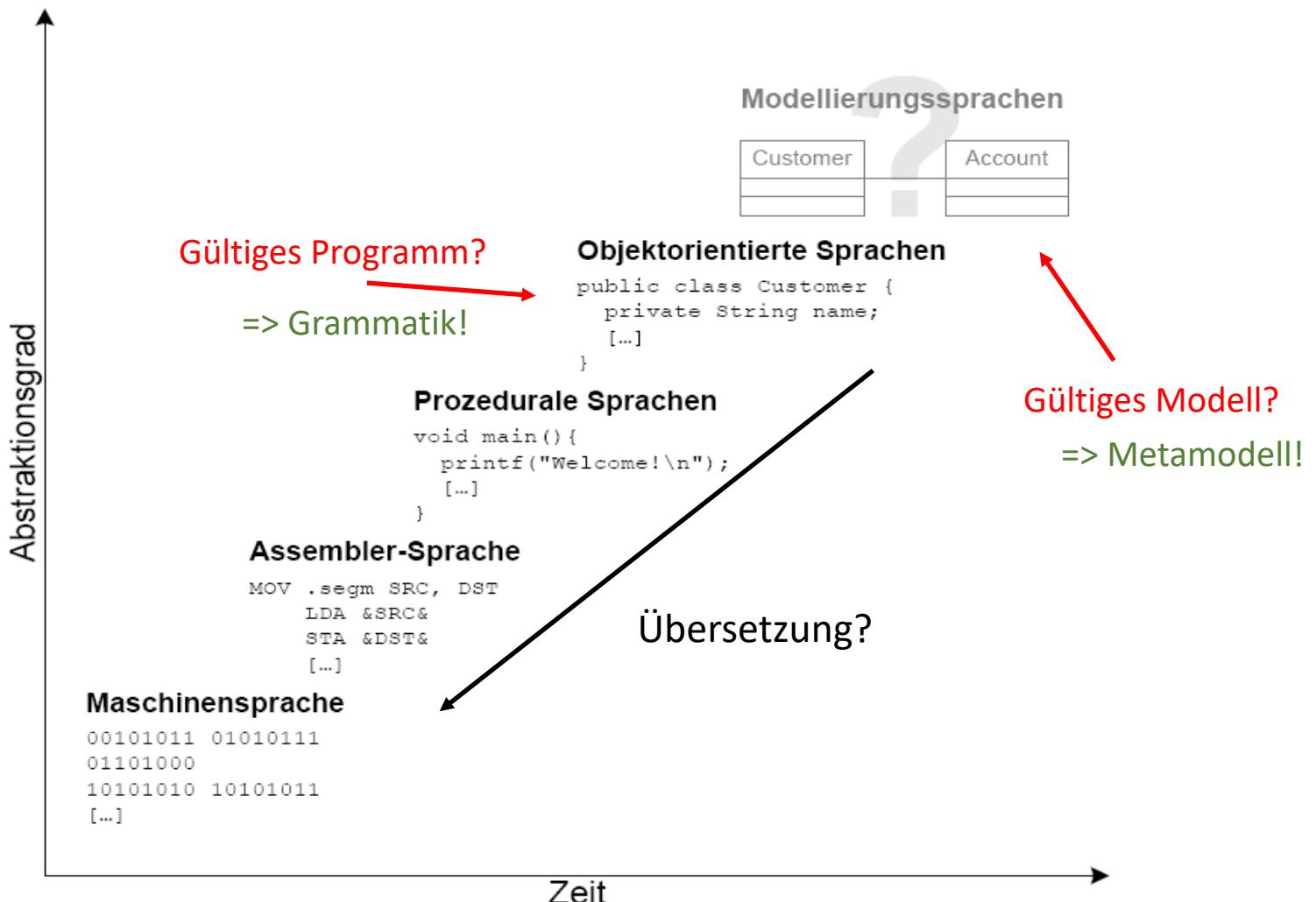
- Problem:
Komplexitäts-
Beherrschung

- Lösung:

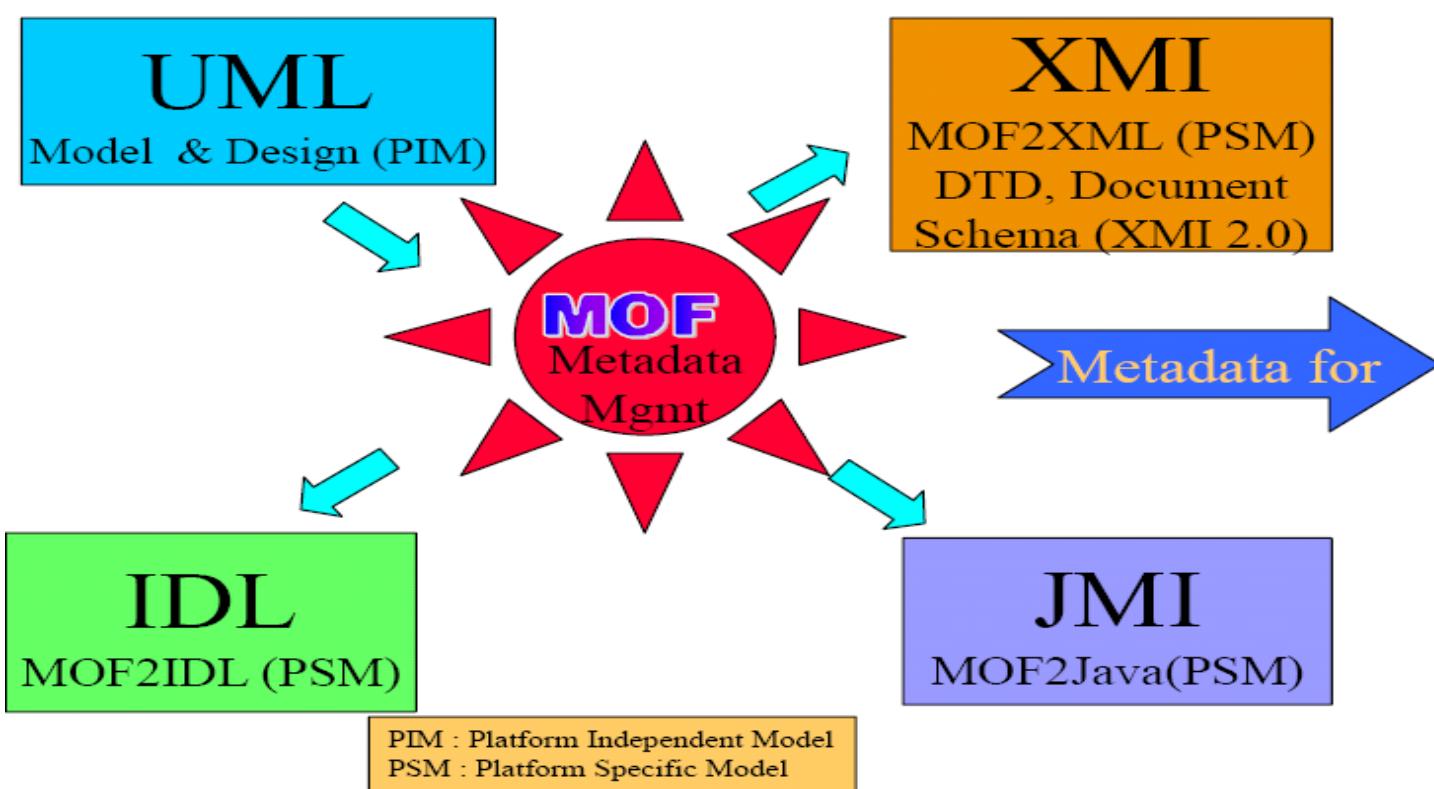
Abstraktion
und
Modularisierung



Analogie Programmiersprachen / Meta-Modellierung



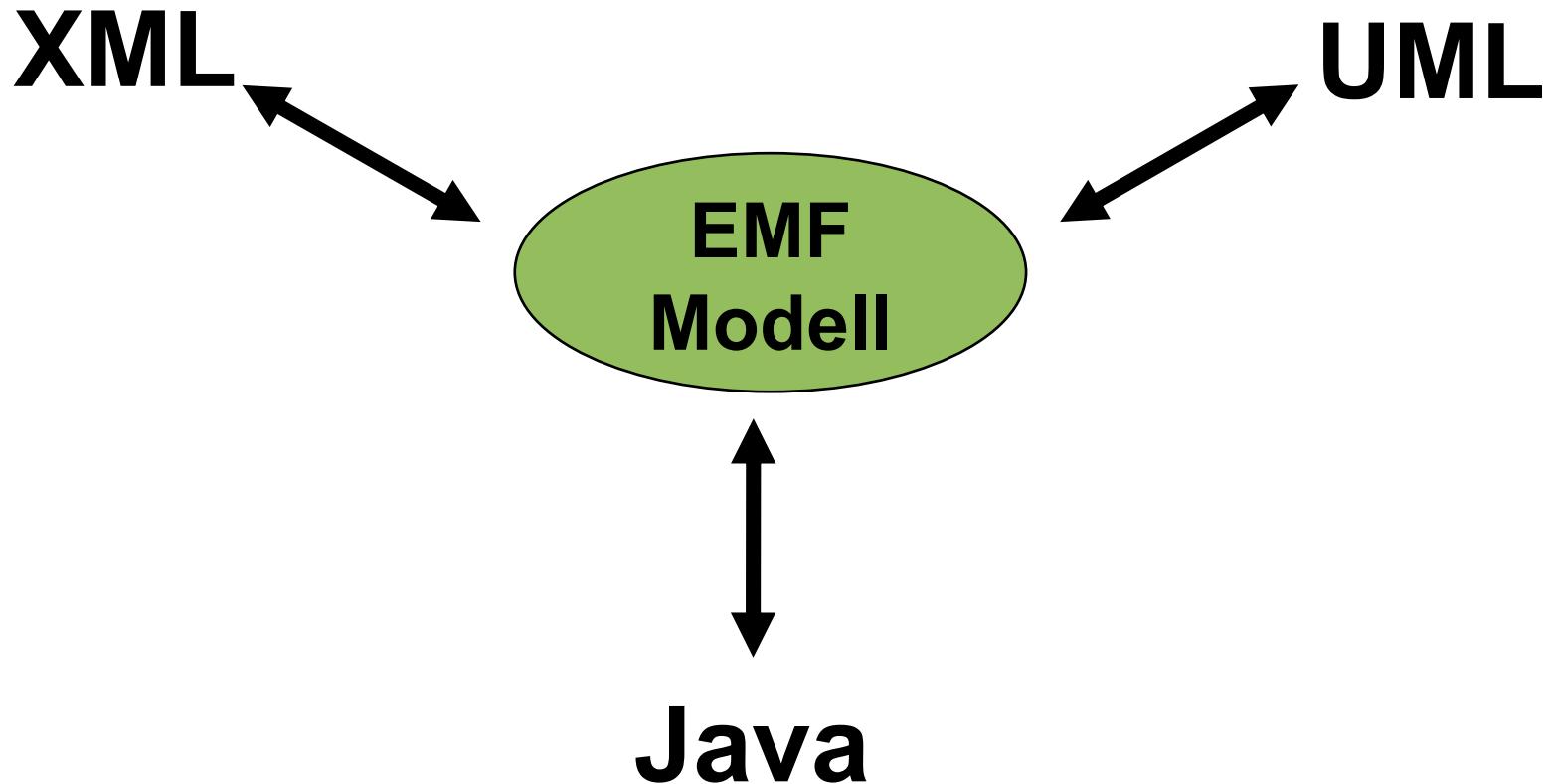
Motivation



- Wie Metamodelle möglichst einfach in MDA-Tools umsetzen ?
 → Insbes. graphische Darstellung von Modellen auf Basis von Metamodellen, Codegenerierung aus Modellen.

EMF - Modellimport

- Metamodelle aus Java-Klassen, UML-Diagrammen und XML-Dateien importierbar.



EMF – Überblick

• **EMF.EMOF:**

- Teil der MOF 2.0-Spezifikation (Essential MOF).

EMF.Ecore: Core EMF-Framework beinhaltet Meta-Model:

- **Um Modelle zu beschreiben.**
- **Laufzeitunterstützung** für Modelle inkl. Benachrichtigung bei Änderungen,
- **Persistenzunterstützung** durch Standard XML-Serialisierung,
- **API** um EMF-Modelle generisch zu verändern.

EMF.Edit:

- Generische und wiederverwendbare Klassen, um **Editoren** für EMF-Modelle zu erstellen.

EMF.Codegen:

- EMF Code-Generierungsframework: kann den für einen Editor für EMF-Modelle benötigten Code generieren.

Essential MOF (EMOF)

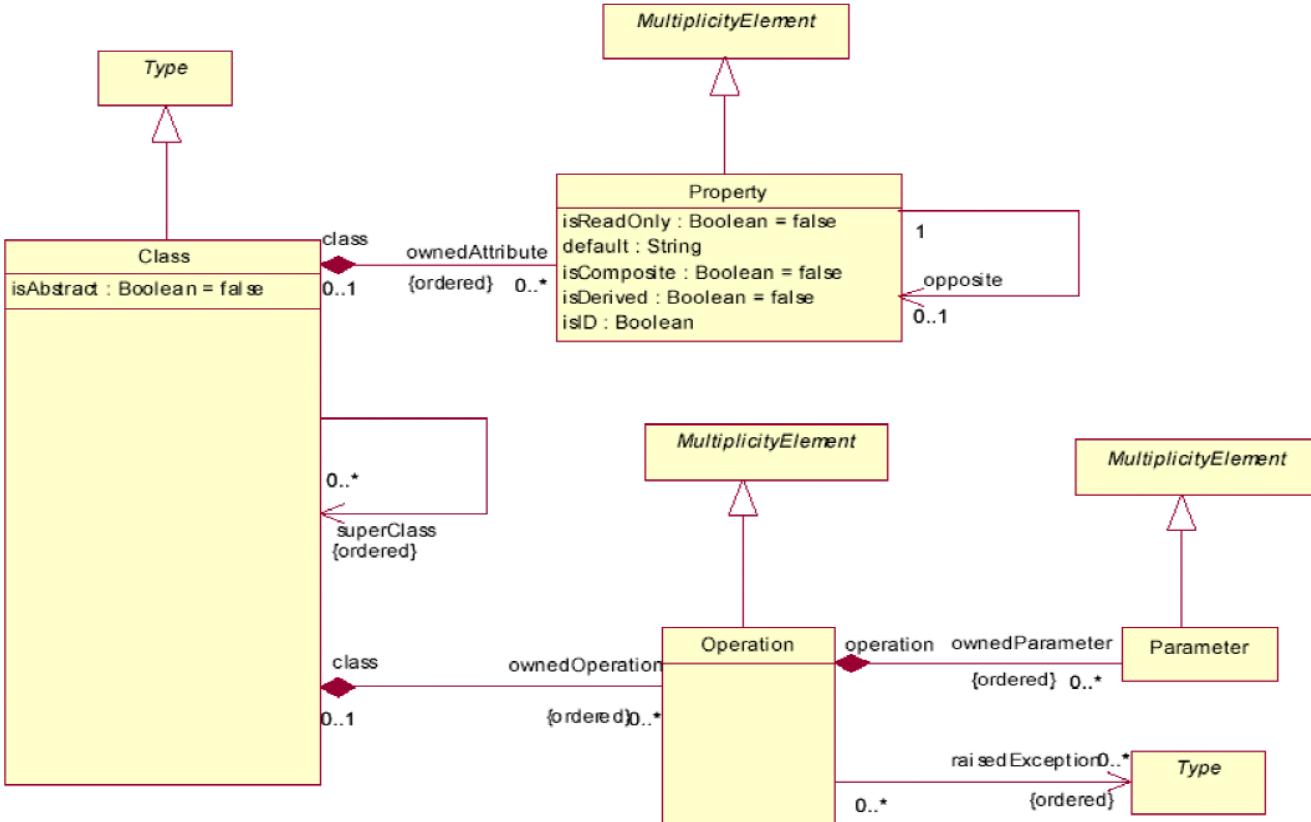
- **EMOF:**

- Teil von MOF 2.0
- Zur **Definition von einfachen Metamodellen.**
- Nutzt OO-Konzepte.

MOF 2.0 verwendet UML 2.0-Klassen-Diagramme.

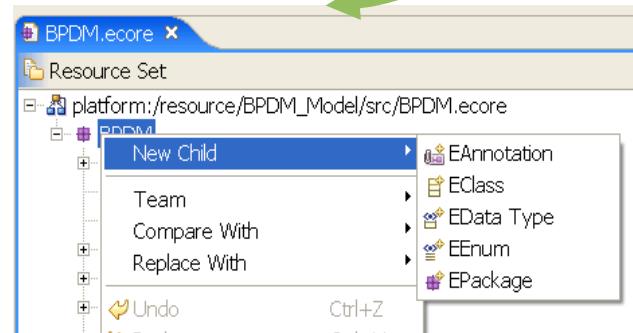
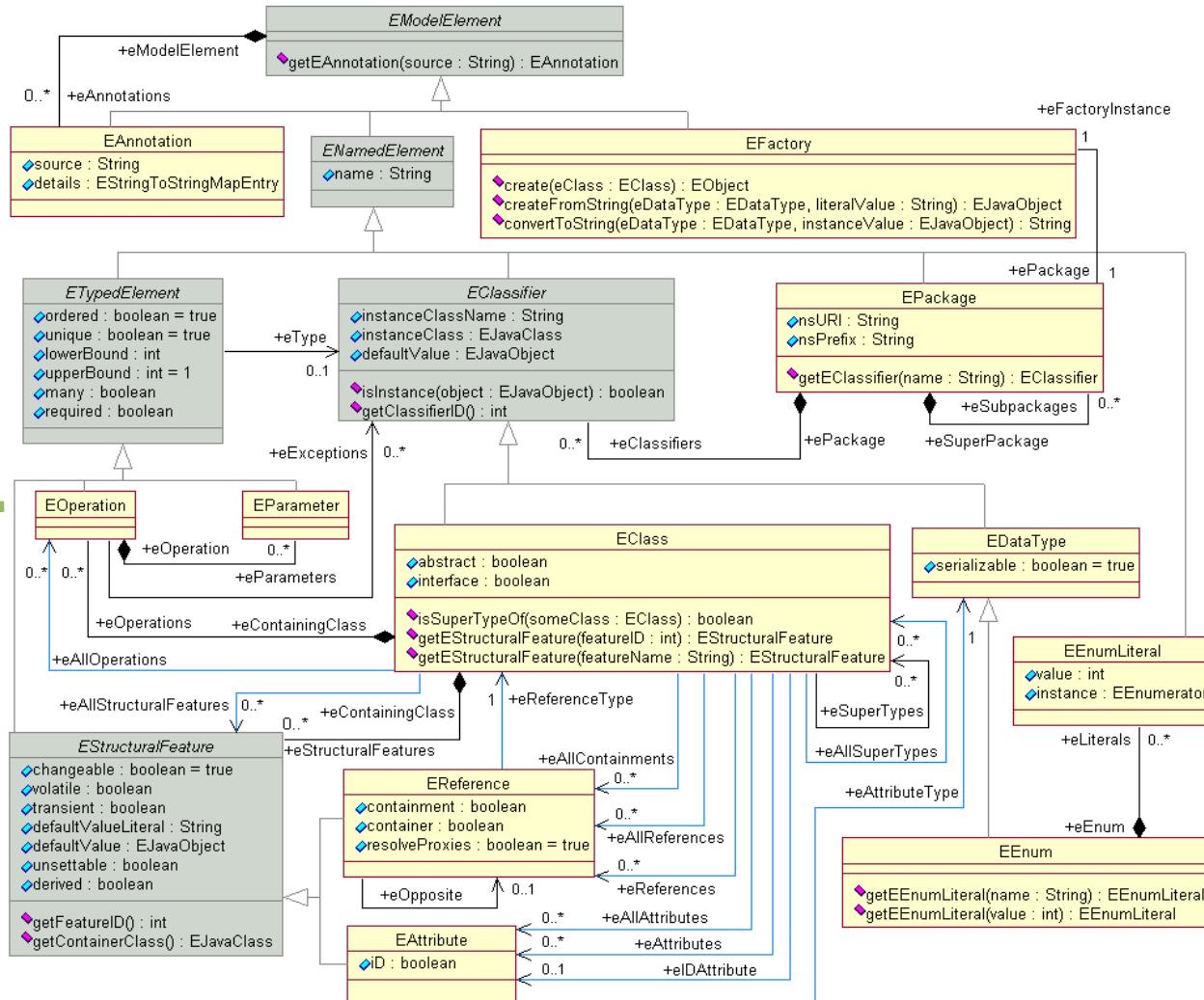
- **Metamodell mit UML-Tools erstellbar.**
- **MOF 2.0 definiert Complete MOF (CMOF) mit zusätzlichen Eigenschaften.**

Beispiel: vereinfachtes Metamodell für Klassendiagramme

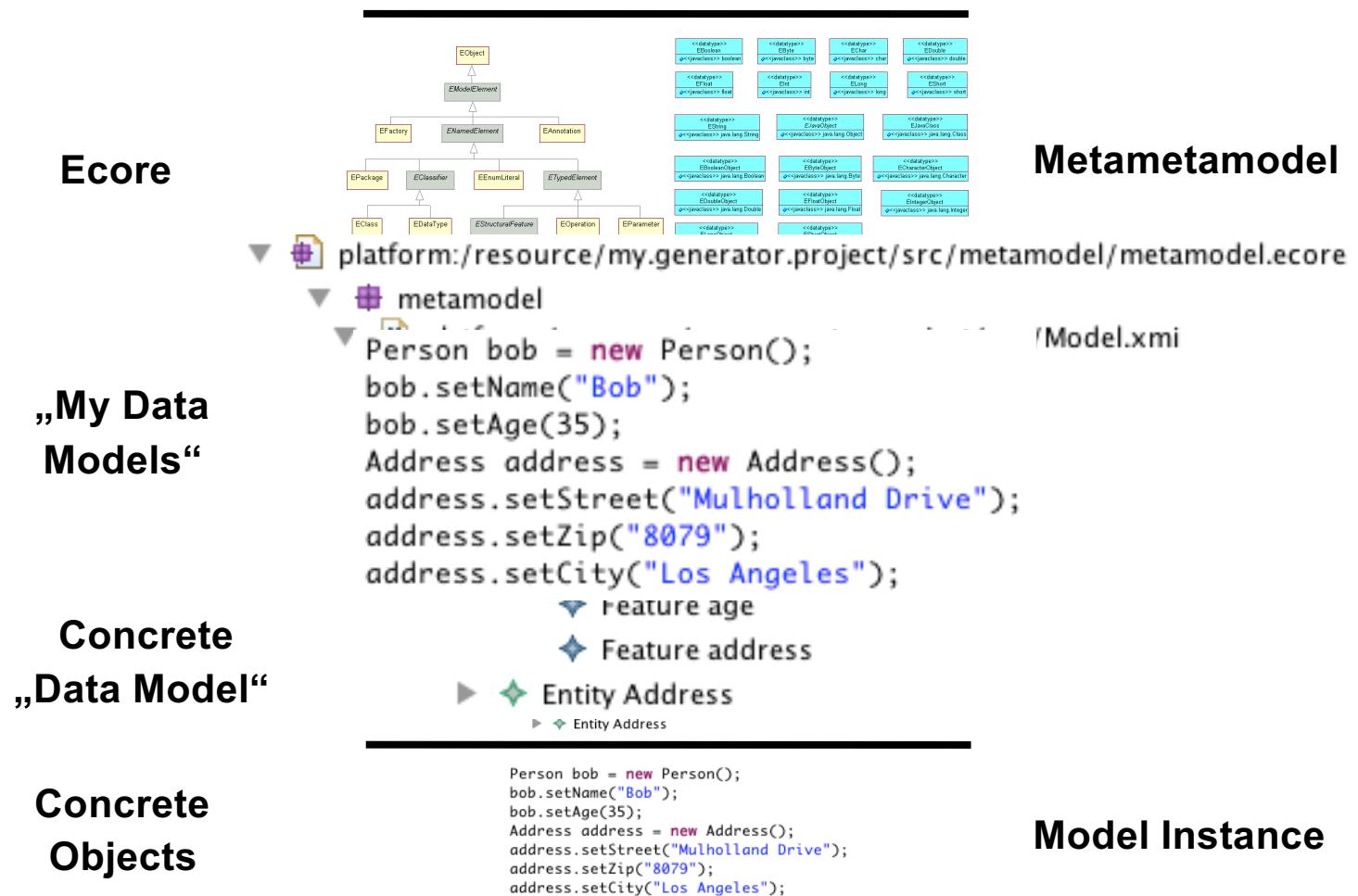


Ecore – Das Kern-Metamodell für EMF

• verwendet für Metamodellierung



Metamodeling im Eclipse Modeling Framework



Eclipse Modeling Framework

- “*a modeling framework and code generation facility for building tools and other applications based on a structured data model*” [EMF]
- part of the Eclipse Modeling Project [EMP]
- pragmatic bottom-up approach
- uses simple model specifications to generate:
 - corresponding Java classes
 - adapter classes for viewing and manipulating models
 - basic tree view editors
 - XML schemas

EMF Models

- EMF models are specified in XMI (XML Metadata Interchange)
 - OMG standard
 - interchange format for data whose metamodel is expressable in Meta-Object Facility (MOF)
 - EMF models contain:
 - simple object definitions
 - object attributes and operations
 - relationships between objects
 - simple conditions on objects and relationships (e.g. multiplicity)
- essentially: the class diagram portion of UML

EMF Model Concepts

- classes, abstract classes and interfaces

Class

AbstractClass

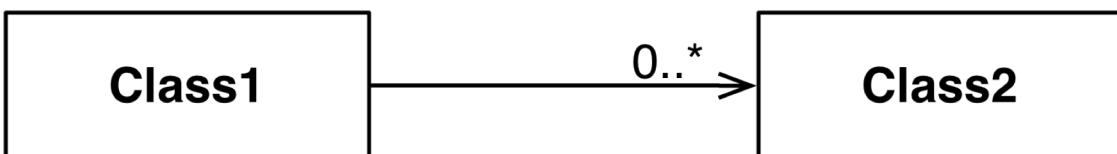
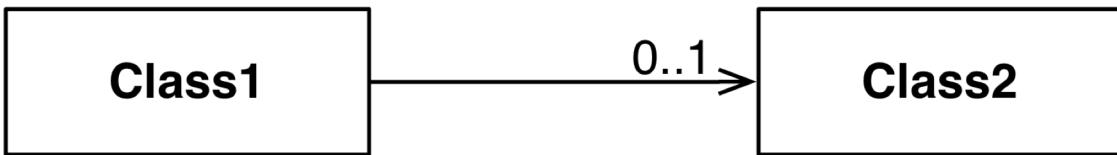
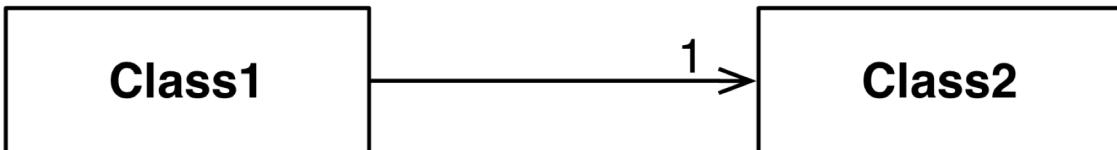
<<interface>>
Name

- operations and attributes

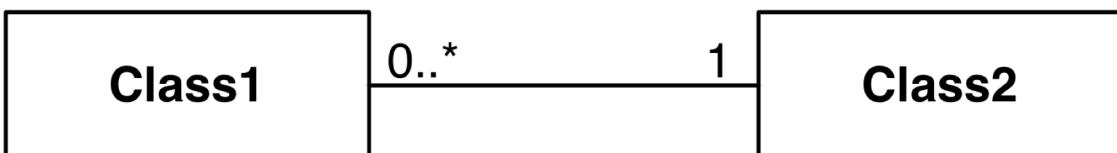
Person
attribute : type
<i><<0..*>></i> anotherattribute : type
Operation (arg1: type) : return type

EMF Model Concepts (2)

- one-way associations

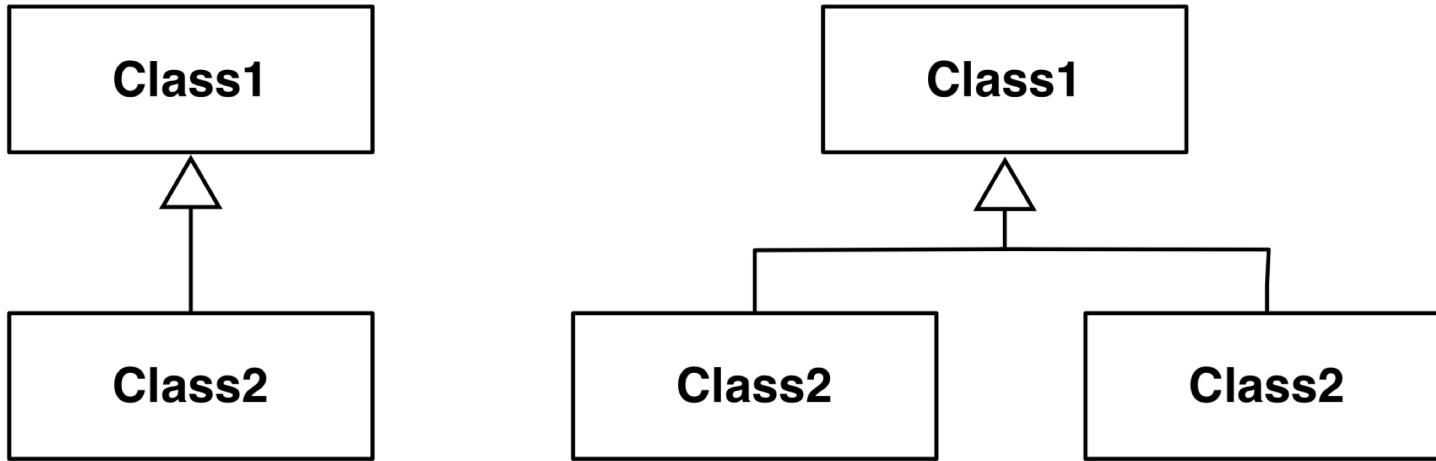


- bidirectional associations

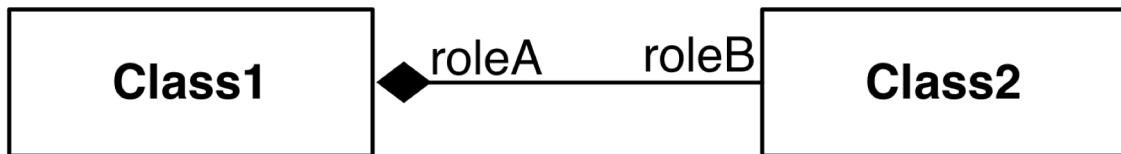


EMF Model Concepts (3)

- inheritance



- containment associations

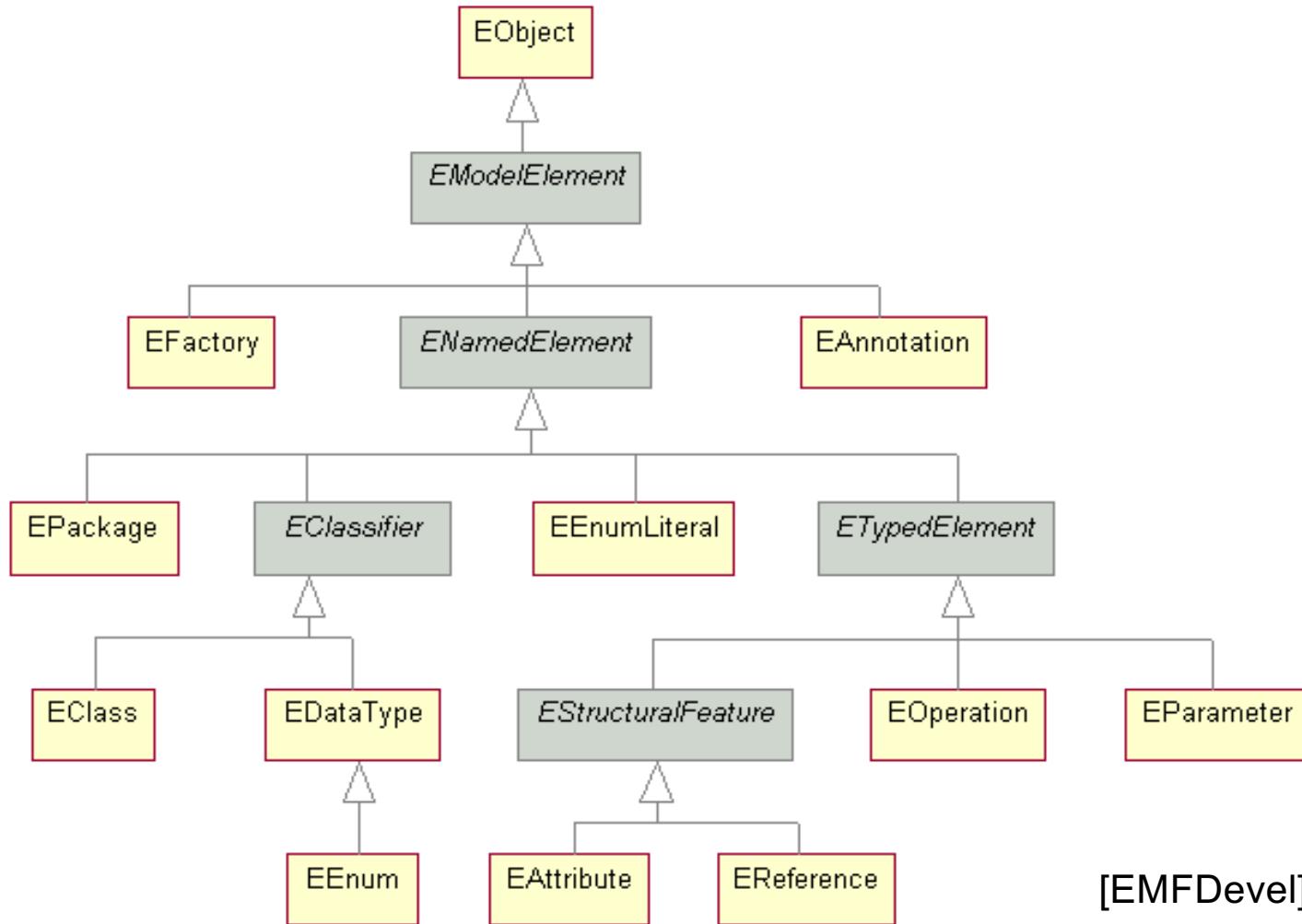


- and also enumerations and data types

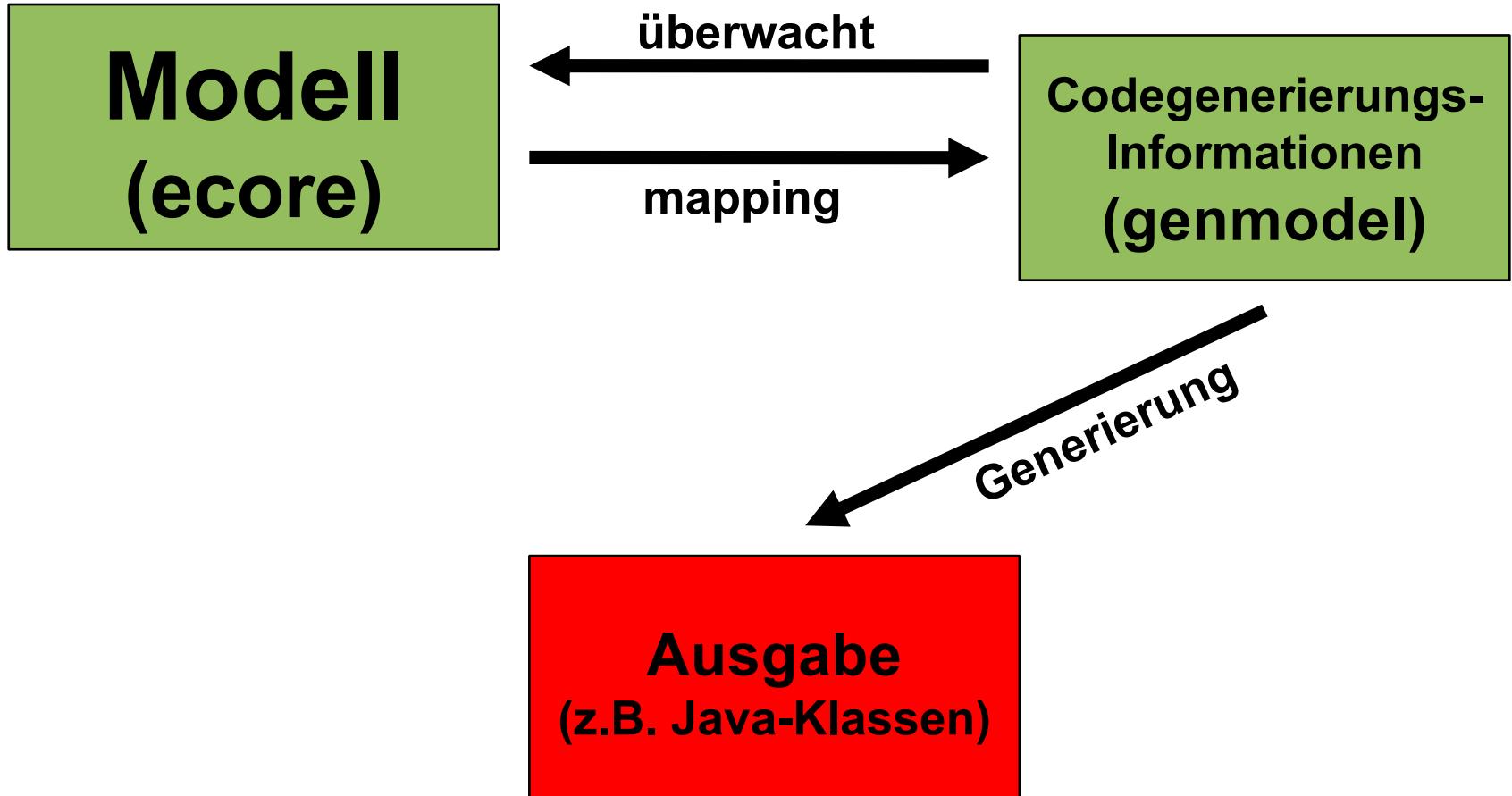
Ecore

- EMF model concepts are described by Ecore
- Ecore = EMF's metamodel
- very similar to OMG's EMOF (Essential Meta-Object Facility, a lightweight version of MOF)
- Ecore is reflexive, i.e. it can be used to describe itself
 - Ecore metamodel is an Ecore model
 - Ecore is just another EMF model
- stored as XMI (.ecore file)

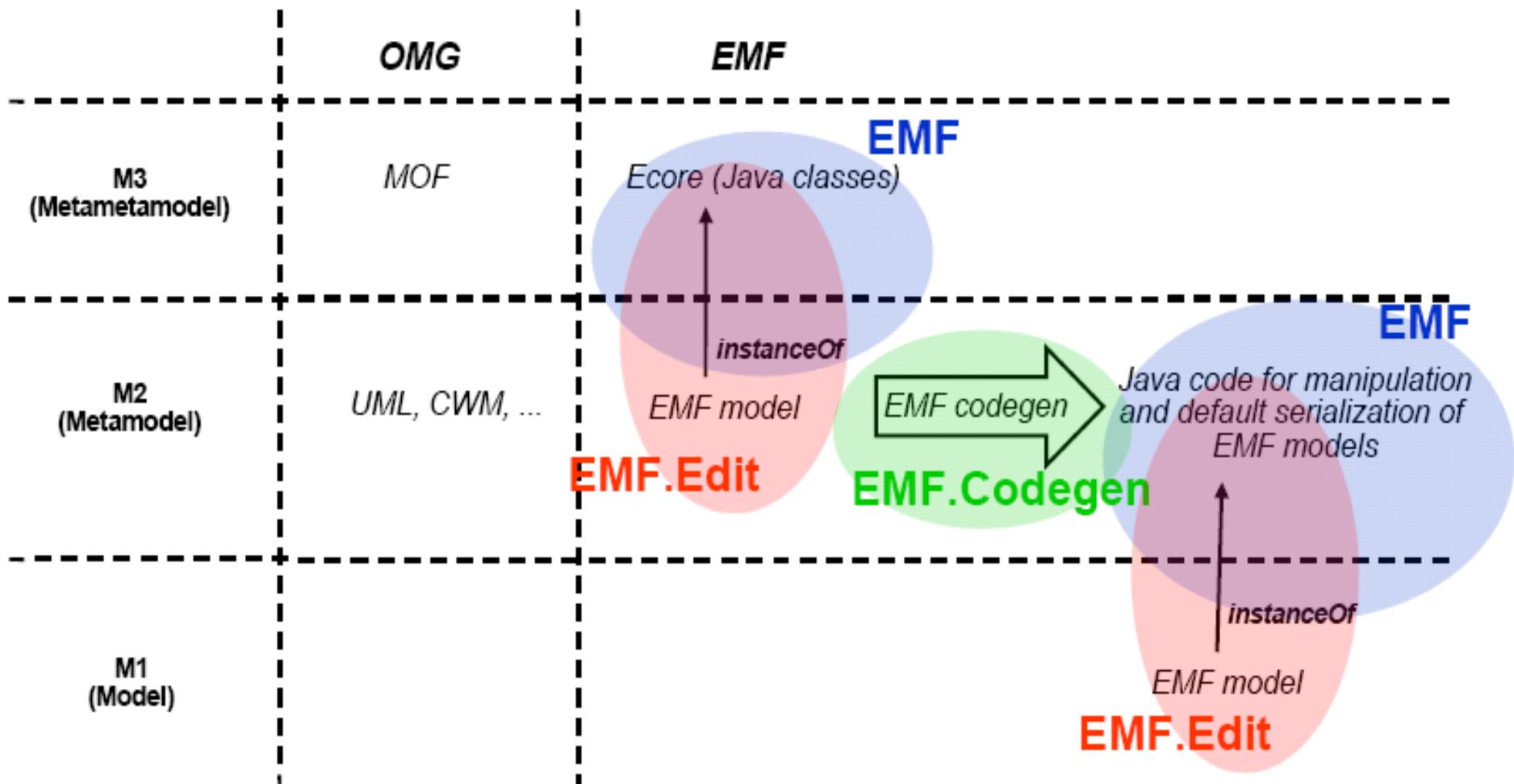
Ecore Modeling Concepts



Abhängigkeit Modell- und Generierungsdateien

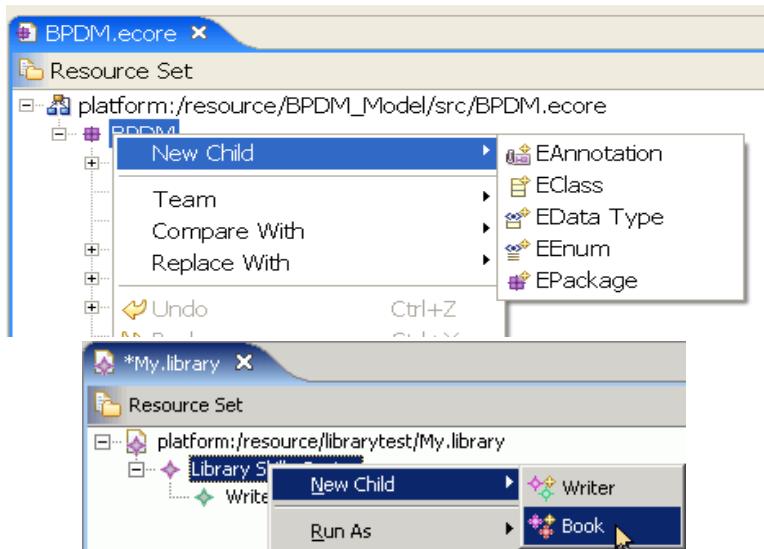


EMF – Überblick über Edit und Codegen

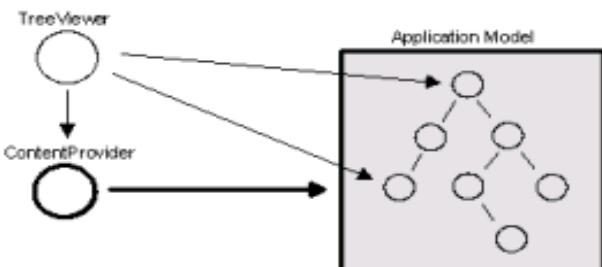


EMF.Edit – EMF.Codegen

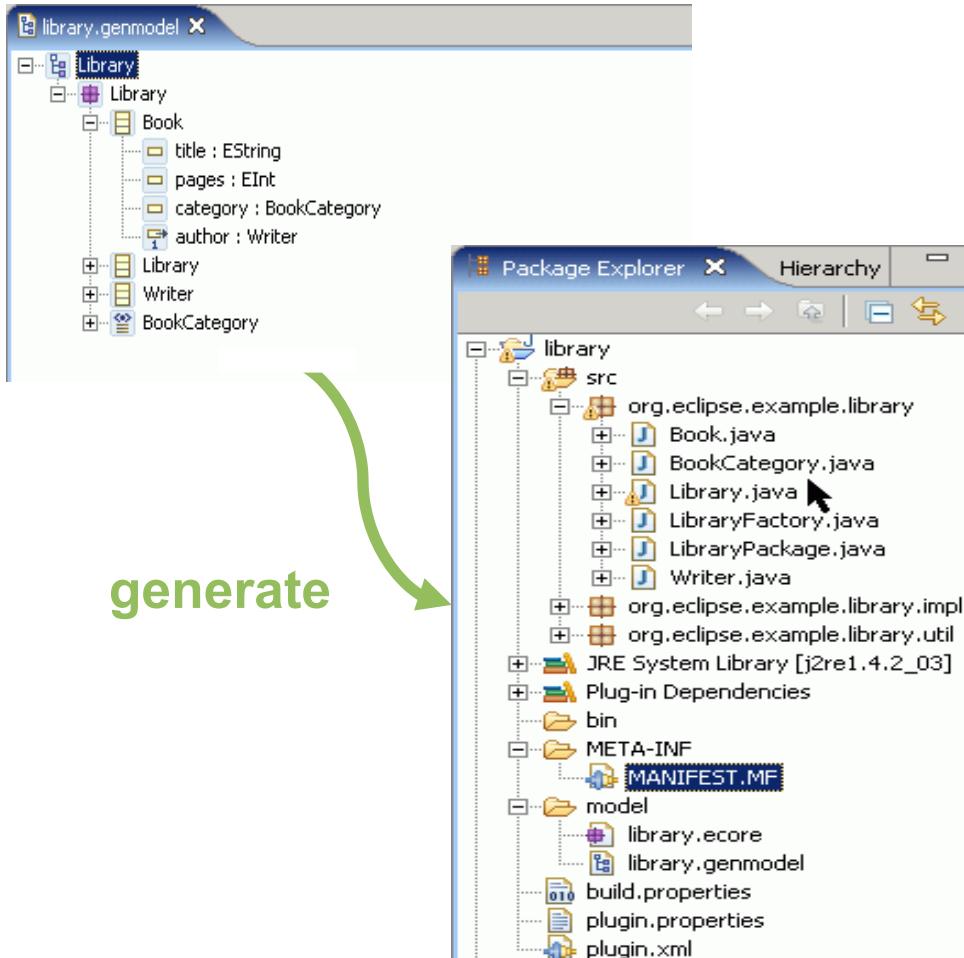
- EMF.Edit
- Modellierungseditor



Content Provider,
etc.



EMF.Codegen



EMF Zusammenfassung

• UML:

- EMF Ecore beschäftigt sich mit **Klassenmodellierungsaspekten** der UML.
- UML 2.0 Metamodell: In EMF Ecore implementiert.

MOF:

- Meta-Object Facility definiert konkrete Untermenge von UML.
→ Beschreibung der **Modellierungskonzepte** innerhalb Repository.
- Vergleichbar mit Ecore.
- Ecore vermeidet einige komplexe Elemente von MOF.
→ Fokus auf Tool-Integration als Management von Metadaten-Repositories.

XMI (*XML Metadata Interchange*):

- Zur **Serialisierung von Modellen**.
- Verwendung von EMF-Modell und Ecore selbst.

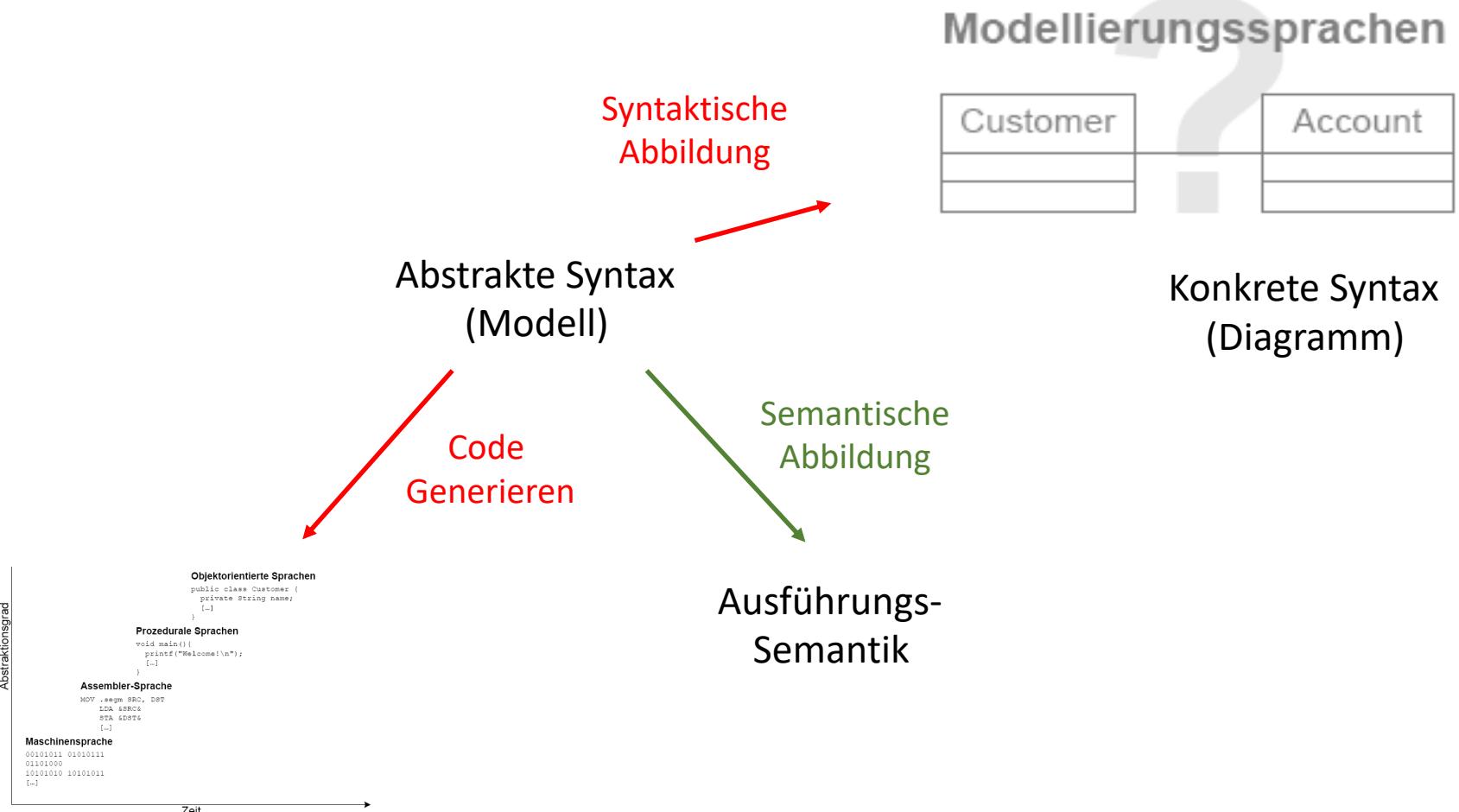
MDA:

- EMF unterstützt Hauptkonzept der MDA.
→ **Modelle für Entwicklung / Generierung** (nicht nur Dokumentation).

Agenda

- Kurze Wiederholung: Meta-Modellierung
- Modellierung von Verhalten
 - Blockschaltdiagramme / Boolesche Logik
 - Erweiterte Endliche Automaten
- Domänenspezifische Sprachen
 - Beispiel: EMF
 - **Code Generierung**

Von Modellen zu Code



Von Modellen zu Code



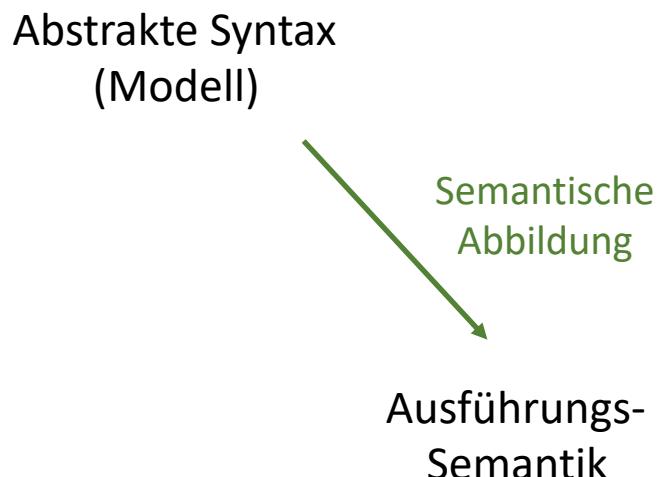
Beispiele:

- Grafische Syntax erweiterter endlicher Automaten vs. Formale Definition
- Abstrakte / Konkrete Syntax von EPKs (an der Tafel)

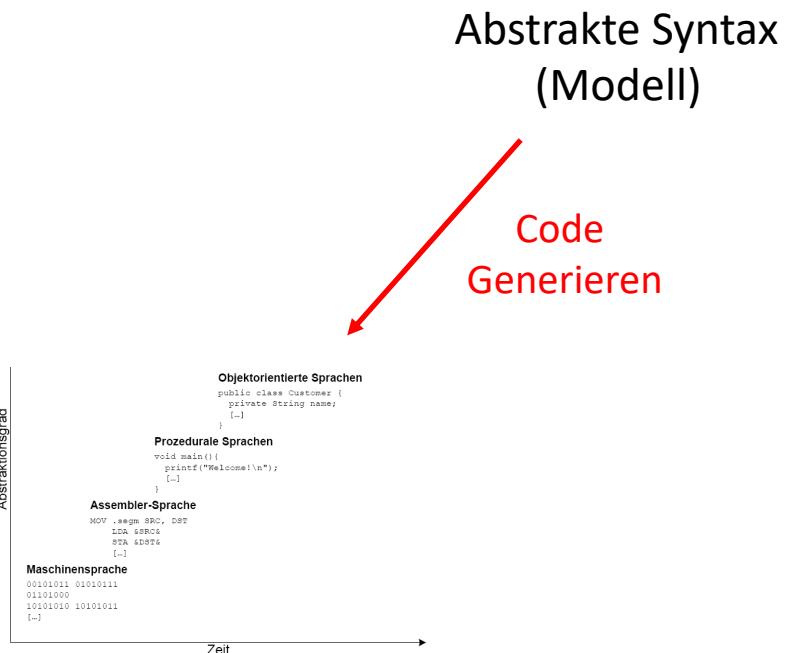
Von Modellen zu Code

Beispiel:

- Semantik erweiterter endlicher Automaten (Ausführungen)



Von Modellen zu Code

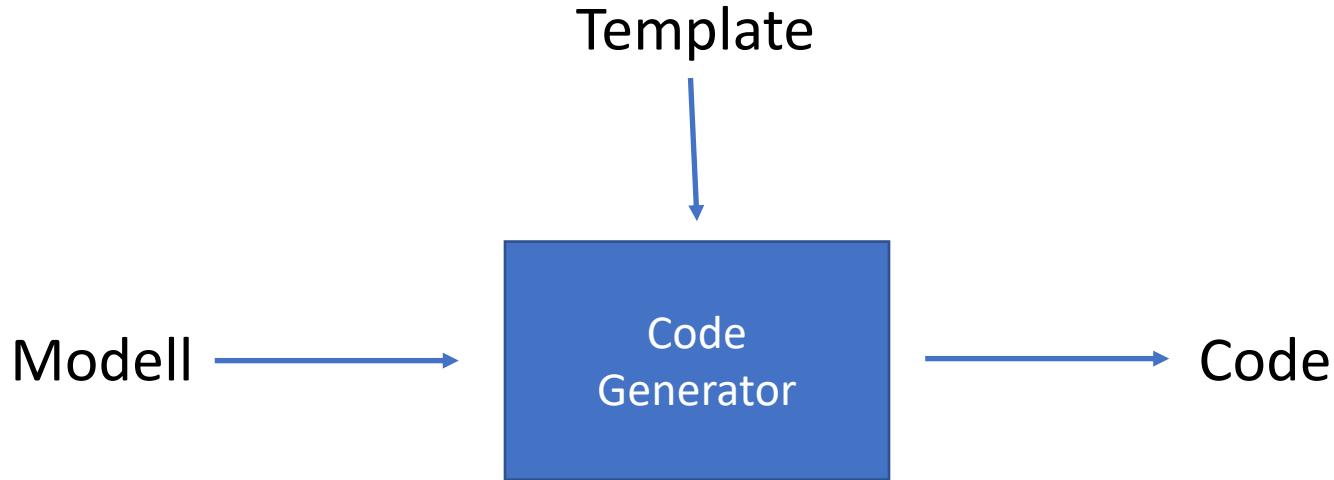


- Oft nicht formal spezifiziert
- Manchmal: Referenzcompiler = definiert Semantik
- Sonst:
 - Äquivalenz zu Semantik per Testen
 - Beweise über Transformationsschritte

Code Generierung

- Hier: Template-basierte Code Generierung am Beispiel erweiterter Endlicher Automaten
- Andere Verfahren in: VUC, Übersetzerbau, ...

Template-basierte Codegenerierung



Template:

- Vorlage die mit Modellinformationen ausgefüllt wird

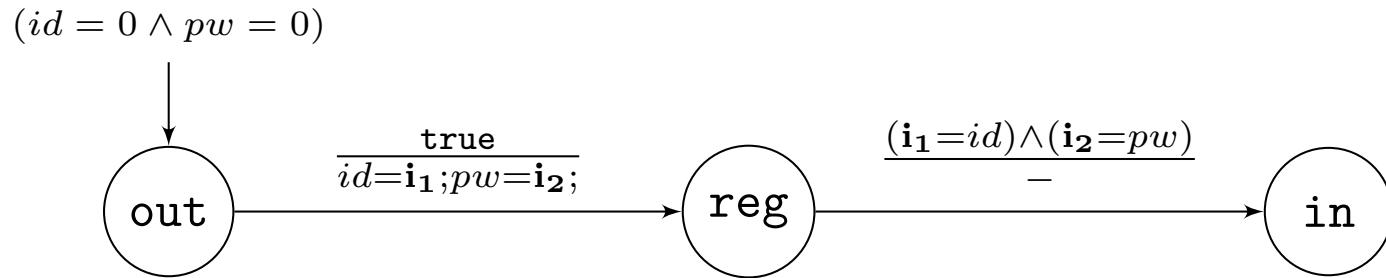
Verschiedene Templates für verschiedene Zielplattformen

Code Generator:

- Interpretiert Template Sprache und Modell

Relativ Generisch

Beispiel: Erweiterte endliche Automaten



- Konkrete Syntax: Automaten Diagramm
- Abstrakte Syntax: Definition
- Semantik: Ausführungen

Semantisches Funktional

Für erweiterten endlichen Automaten $A = \langle I, Q, X, (q_0, v_0), T, \text{State}, L \rangle$ sei

mit

$$\llbracket A \rrbracket : V_I^* \mapsto \{0, 1\}$$

Wir wollen Code generieren, der einen Boolean zurückgibt

$$\llbracket A \rrbracket(\vec{i}) = \begin{cases} 1 & \text{wenn } \vec{i} \text{ eine Ausführung in } A \text{ hat,} \\ 0 & \text{sonst.} \end{cases}$$

für $\vec{i} \in V_I^*$

Im Allgemeinen Fkt. Eingabe -> Ausgabe

Templates

Für $L(q_0) = \text{lbl}$

```
class Automaton {
    enum State { ... };
    State q=State.lbl;
    <<vars>>
    boolean step(Input i) {
        if (q==null) return false;
        switch (q) {
            <<block>>
        }
        return true;
    }
}
```

$$A = \langle I, Q, X, (q_0, v_0), T, \text{State}, L \rangle$$

Verweis auf
weitereTemplates

Templates

$$A = \langle I, Q, X, (q_0, v_0), T, \text{State}, L \rangle$$

<<vars>> Für $x \in X$ von Typ Nat und $n = v_0(x)$

```
int x = n;
```

(Boolesche Variablen analog)

<<block>> Für q mit $L(q) = \text{lbl}$

```
case State.lbl:
  <<trans>> ←
  q = null;
  break;
```

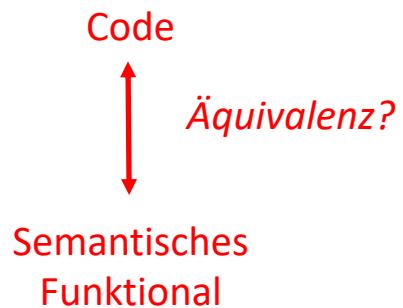
<<trans>> Für $t = (q, g, u, q')$ mit $L(q') = \text{lbl2}$

```
if (g.holdsFor(i, this)) {
  eval(u,i); // parallel Update
  q = State.lbl2;
  break;
}
```

Semantik

$$A = \langle I, Q, X, (q_0, v_0), T, \text{State}, L \rangle$$

```
public static boolean semofA(Automaton A, Input ... ivec) {
    boolean ret;
    for (Input i : ivec) {
        ret = a.step(i);
    }
    return ret;
}
```



$$\llbracket A \rrbracket : V_I^* \mapsto \{0, 1\}$$

mit

$$\llbracket A \rrbracket(\vec{i}) = \begin{cases} 1 & \text{wenn } \vec{i} \text{ eine Ausführung in } A \text{ hat,} \\ 0 & \text{sonst.} \end{cases}$$

für $\vec{i} \in V_I^*$

Code Generator in Xtend



- Statisch getypte Programmiersprache
- Unterstützt das Java Typsystem
 - `byte, short, int, long, float, double, boolean, char`
 - Arrays, Enums, Annotations, Generics
 - Java Klassen und Interfaces auf dem Classpath
- Xtend-Code aus Java aufrufbar und umgekehrt
- Umfassende Kompatibilität der Sprachen
- Einfachere Syntax, weniger Boilerplate Code

Field Access & Method Invocation

- Einfache Namensnennung kann verweisen auf
 - Lokales Feld
 - Variable
 - Parameter
 - Methode ohne Argumente (denn leere Klammern () sind optional)
- Existiert kein Feld / Methode, wird auf Getter verwiesen
- Setter können über Assignments aufgerufen werden

Xtend

```
user.name = 'Foo'          // user.setName('Foo')
```

Extension-Methoden

»Xtend

- Lokale Extension-Methoden sind implizit definiert
- Xtend erweitert Java-Typen über Extension-Methoden
 - IterableExtensions
 - CollectionExtensions
 - ListExtensions
 - MapExtensions

Xtend

```
map.filter[...]  
elements.mapValues[...]  
::
```

Template Expressions



Xtend

- Lesbare String-Konkatenation, nützlich für Generatoren
 - Startet / endet mit: `'''`
 - Expression bzw. Code-Block in „Guillemets“: `« »`

Xtend

```
def toHTML(String content) {
    <html>
        <body>
            <content>
        </body>
    </html>
}
```

Template Expressions



- Lesbare String-Konkatenation, nützlich für Generatoren
- Startet / endet mit: «»
- Expression bzw. Code-Block in „Guillemets“: « »
- Kommentare in Templates mit: «««

Xtend

```
def toHTML(String content) «»
  <html>
    <body>
      ««« This is not visible in the result. Nor is this: «content»
    </body>
  </html>
  «»
```

Template Expressions

»Xtend

- Bedingungen in Templates mit: **IF** - **ELSE** - **ENDIF**

Xtend

```
def toHTML(Paragraph p) ***\n<html>\n  <body>\n    «IF p.headline != null»\n      <h1>«p.headline»</h1>\n    «ELSE»\n      <div>ERROR: Headline is null!</h1>\n    «ENDIF»\n    <p>\n      «p.text»\n    </p>\n  </body>\n</html>\n***
```

Template Expressions



- Schleifen in Templates mit: **FOR** - **ENDFOR**
- Einfügungen mit: **BEFORE**, **SEPARATOR**, **AFTER**

Xtend

```
def toHTML(List<Paragraph> list) ''''  
  <html>  
    <body>  
      «FOR p:list BEFORE '<div>' SEPARATOR '</div><div>' AFTER '</div>'»  
        «IF p.headline != null»  
          <h1>«p.headline»</h1>  
        «ENDIF»  
        <p>  
          «p.text»  
        </p>  
      «ENDFOR»  
    </body>  
  </html>  
'''
```

Zusammenfassung

- Code Generierung:
 - Von Modellen zu ausführbarem Code
 - Template-basiert:
 - Ausfüllbare Vorlagen
 - Möglichst Generischer Code Generator
- Semantische Korrektheit
 - Fast nie bewiesen (Tests oder Referenz-Compiler)
 - Beweis über Semantik-erhaltende Transformationen