



# OCL – OBJECT CONSTRAINT LANGUAGE

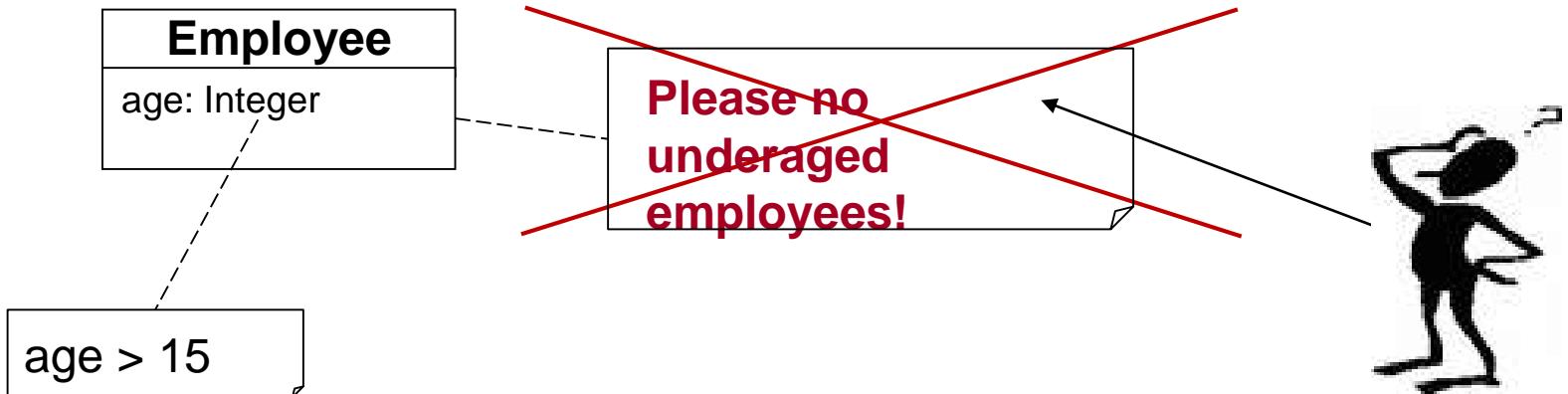


# Motivation

- Graphical modeling languages are generally not able to describe all facets of a problem description
  - » MOF, UML, ER, ...
- Special constraints are often (if at all) added to the diagrams in natural language
  - » Often ambiguous
  - » Cannot be validated automatically
  - » No automatic code generation
- Constraint definition also crucial in the definition of new modeling languages (DSLs).

## Motivation

- Example 1



e1:Employee
age = 19 ✓

e2:Employee
age = 31 ✓

e3:Employee
alter = 11 ✗

Additional question: How do I get all Employees younger than 30 years old?



# Motivation

- **Formal specification languages are the solution**
  - » Mostly based on set theory or predicate logic
  - » Requires good mathematical understanding
  - » Mostly used in the academic area, but hardly used in the industry
  - » Hard to learn and hard to apply
  - » Problems when to be used in big systems
- **Object Constraint Language (OCL): Combination of modeling language and formal specification language**
  - » Formal, precise, unique
  - » Intuitive syntax is key to large group of users
  - » No programming language (no algorithms, no technological APIs, ...)
  - » Tool support: parser, constraint checker, codegeneration,...



# OCL usage

- **Constraints in UML-models**
  - » Invariants for classes, interfaces, stereotypes, ...
  - » Pre- and postconditions for operations
  - » Guards for messages and state transition
  - » Specification of messages and signals
  - » Calculation of derived attributes and association ends
- **Constraints in meta models**
  - » Invariants for Meta model classes
  - » Rules for the definition of well-formedness of meta model
- **Query language for models**
  - » In analogy to SQL for DBMS, XPath and XQuery for XML
  - » Used in transformation languages



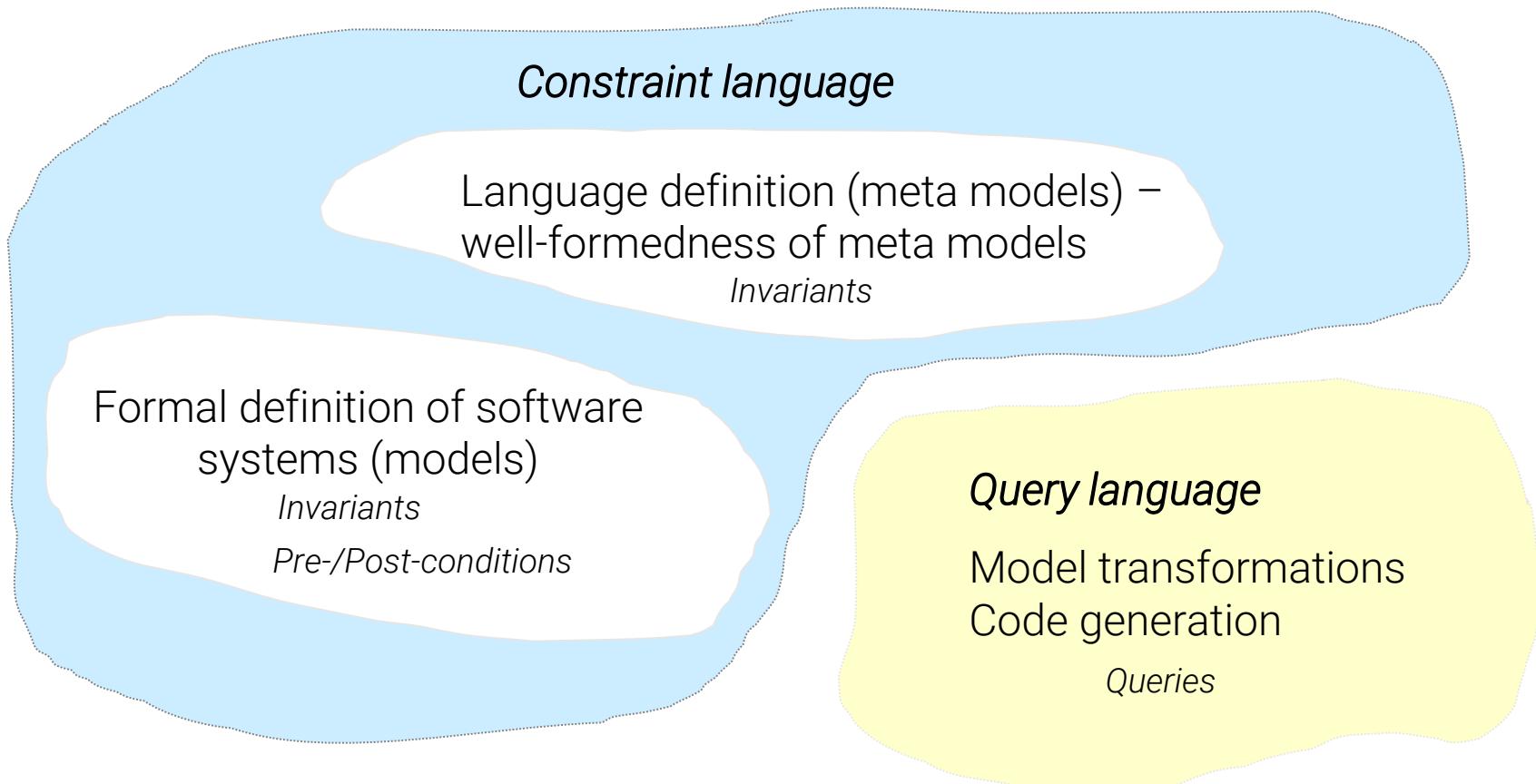
## OCL usage

- OCL field of application
  - » Invariants
  - » Pre-/Postconditions
  - » Query operations
  - » Initial values
  - » Derived attributes
  - » Attribute/operation definition
- context C inv: I
- context C::op() : T
- pre: P post: Q
- context C::op() : T body: e
- context C::p : T init: e
- context C::p : T derive: e
- context C def: p : T = e
- Caution: Side effects are not allowed!
  - » Operation C::getAtt : String body: att allowed in OCL
  - » Operation C::setAtt(arg) : T body: att = arg not allowed in OCL

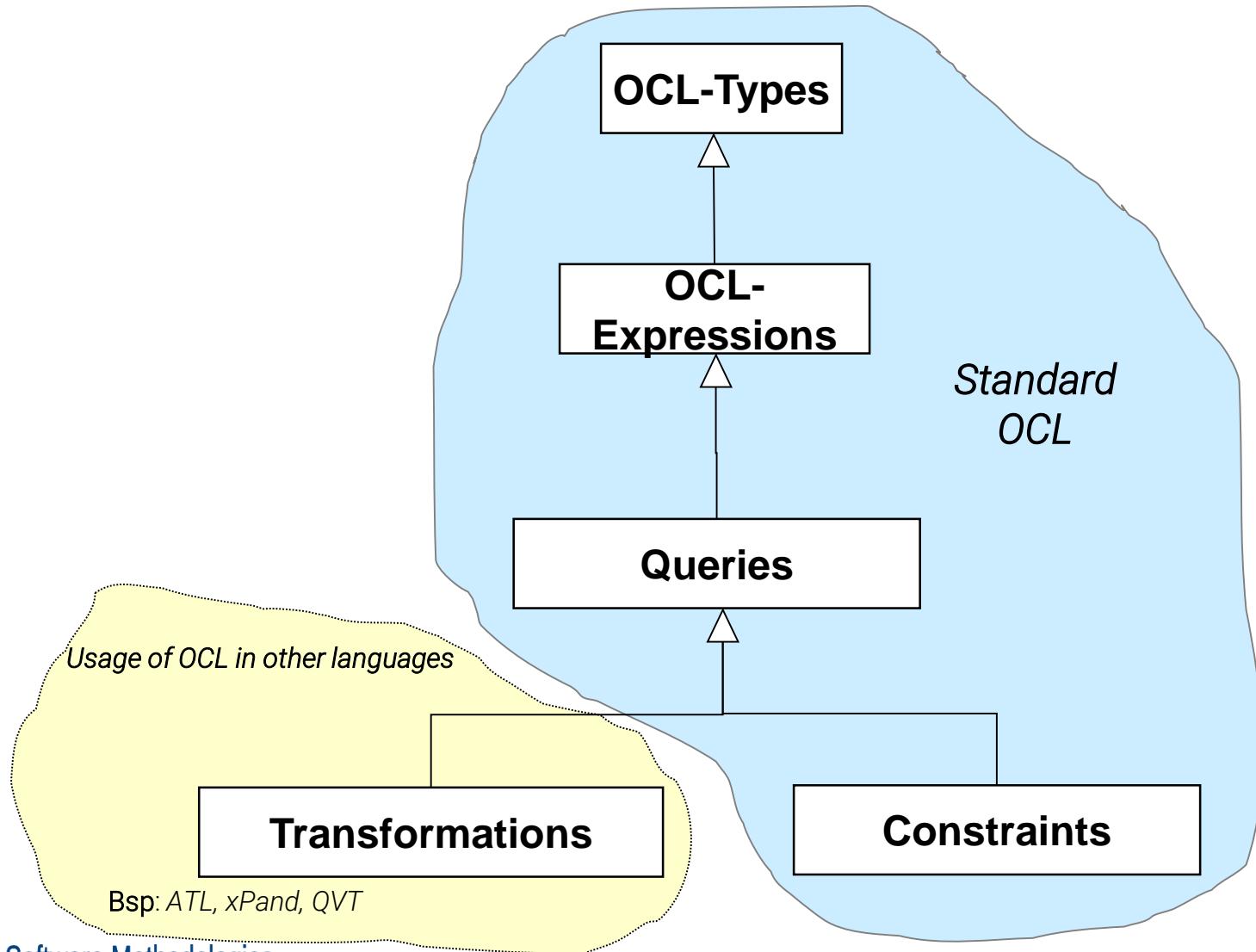


## OCL usage

- Field of application of OCL in model driven engineering



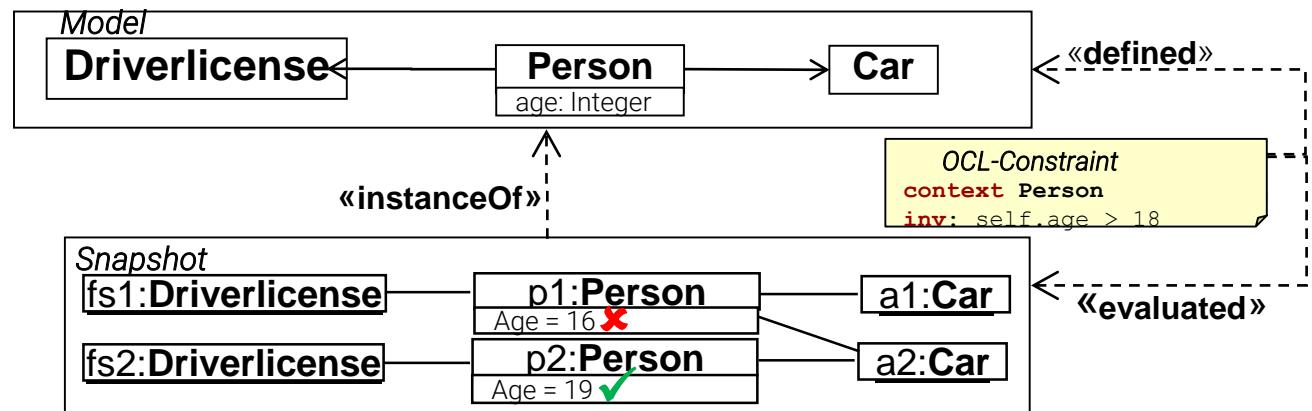
## OCL usage



# OCL usage

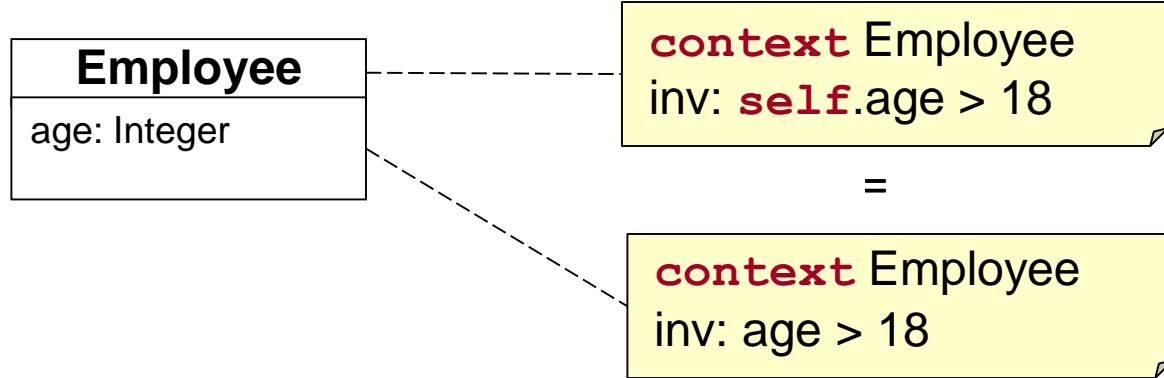
## How does OCL work?

- Constraints are defined on the modeling level
  - » Basis: Classes and their properties
- Information of the object graph are queried
  - » Represents system status, also called snapshot
- Analogy to XML query languages
  - » XPath/XQuery query XML-documents
  - » Scripts are based on XML-schema information
- Examples



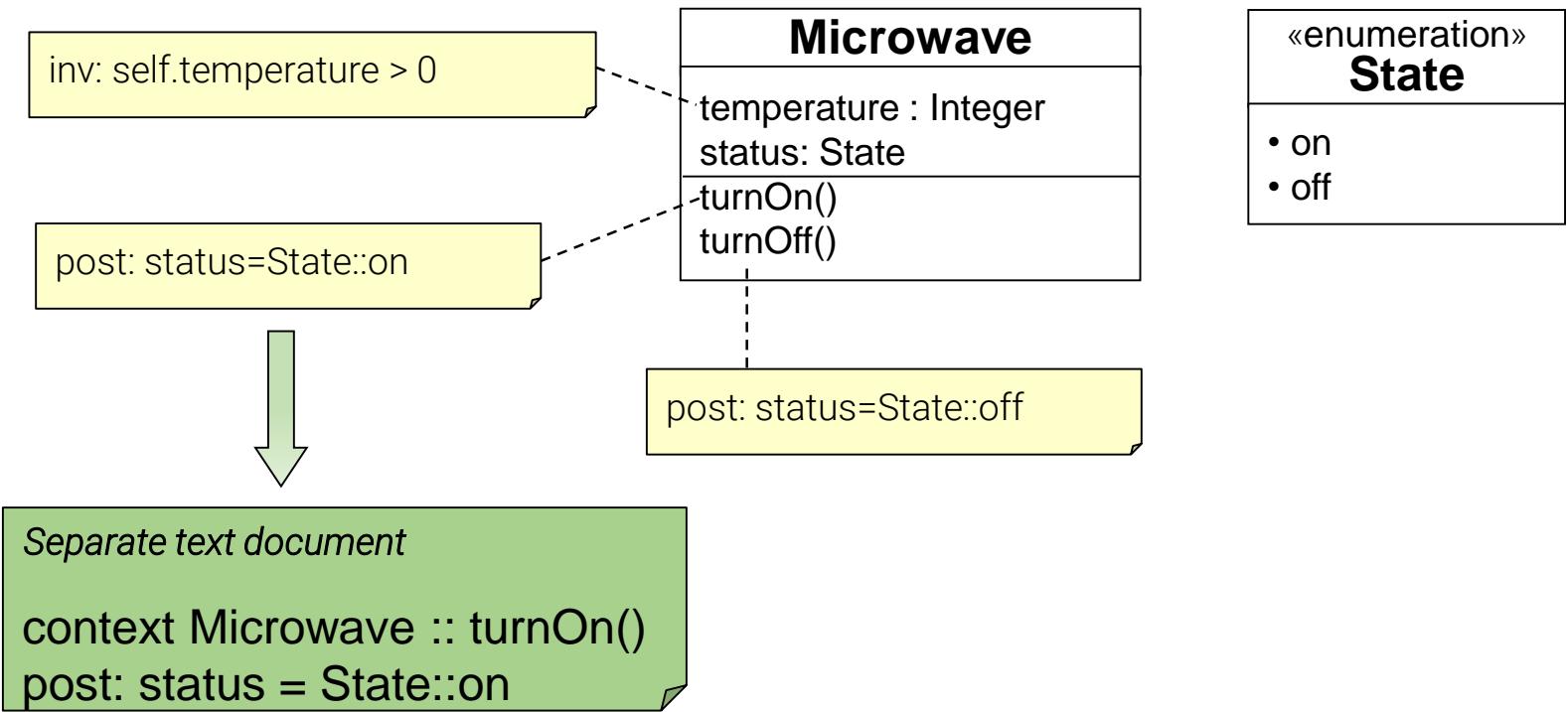
# Design of OCL

- Examples



# Design of OCL

- OCL can be specified in two different ways
  - » As a comment directly in the class diagram (context described by connection)
  - » Separate document file





# Types

- **OCL is a typed language**
  - » Each object, attribute, and result of an operation or navigation is assigned to a range of values (type)
- **Predefined types**
  - » Basic types
    - Simple types: Integer, Real, Boolean, String
    - OCL-specific types: AnyType, TupleType, InvalidType, ...
  - » Set-valued, parameterized Types
    - Abstract supertype: Collection( $T$ )
    - Set( $T$ ) – no duplicates
    - Bag( $T$ ) – duplicates allowed
    - Sequence( $T$ ) – Bag with ordered elements, association ends {ordered}
    - OrderedSet( $T$ ) – Set with ordered elements, association ends {ordered, unique}
- **Userdefined Types**
  - » Instances of Class in MOF and indirect instances of Classifier in UML are types
  - » EnumerationType – user defined set of values for defining constants

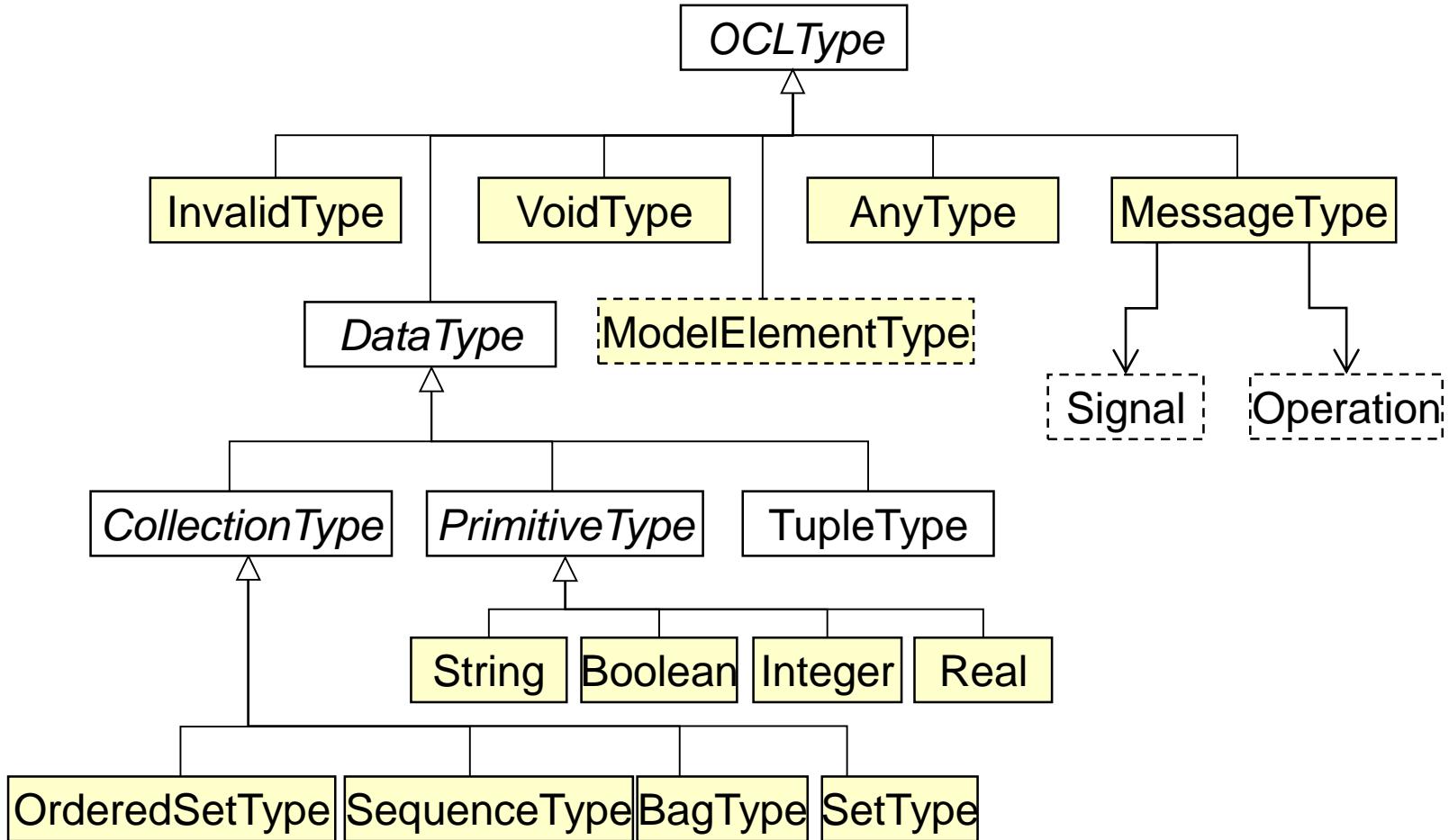


# Types

- Basic types
  - » true, false : Boolean
  - » -17, 0, 1, 2 : Integer
  - » -17.89, 0.01, 3.14 : Real
  - » "Hello World" : String
- Set-valued, parameterized types
  - » Set{ Set{1}, Set{2, 3} } : Set(Set(Integer))
  - » Bag{ 1, 2.0, 2, 3.0, 3.0, 3 } : Bag(Real)
  - » Tuple{ x = 5, y = false } : Tuple{x: Integer, y : Boolean}
- User defined types
  - » Passenger : Class, Flight : Class, Provider : Interface
  - » Status::started - enum Status {started, landed}

# Types

- OCL meta model (extract)





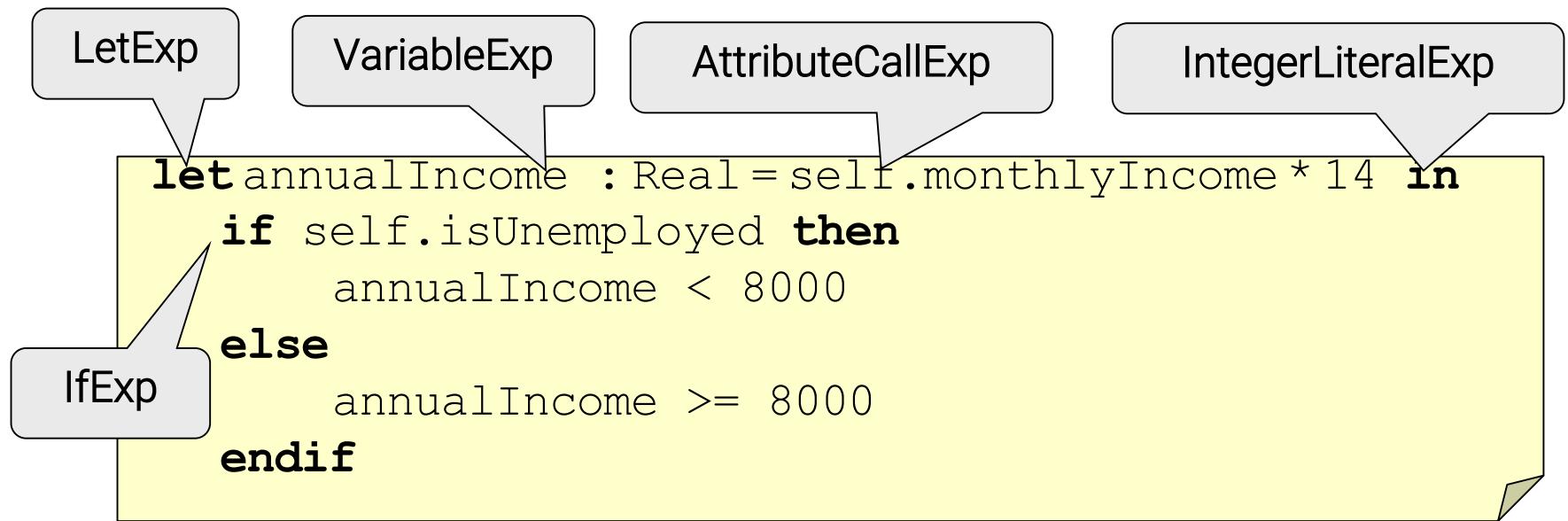
# Expressions

- **Each OCL expression is an indirect instance of `OCLExpression`**
  - » Calculated in certain environment – cf. context
  - » Each OCL expression has a typed return value
  - » OCL Constraint is an OCL expression with return value Boolean
- **Simple OCL expressions**
  - » LiteralExp, IfExp, LetExp, VariableExp, LoopExp
- **OCL expressions for querying model information**
  - » FeatureCallExp – abstract superclass
  - » AttributeCallExp – querying attributes
  - » AssociationEndCallExp – querying association ends
    - Using role names; if no role names are specified, lowercase class names have to be used (if unique)
  - » AssociationClassCallExp – querying association class (only in UML)
  - » OperationCallExp – Call of query operations
    - Calculate a value, but do not change the system state!



# Expressions

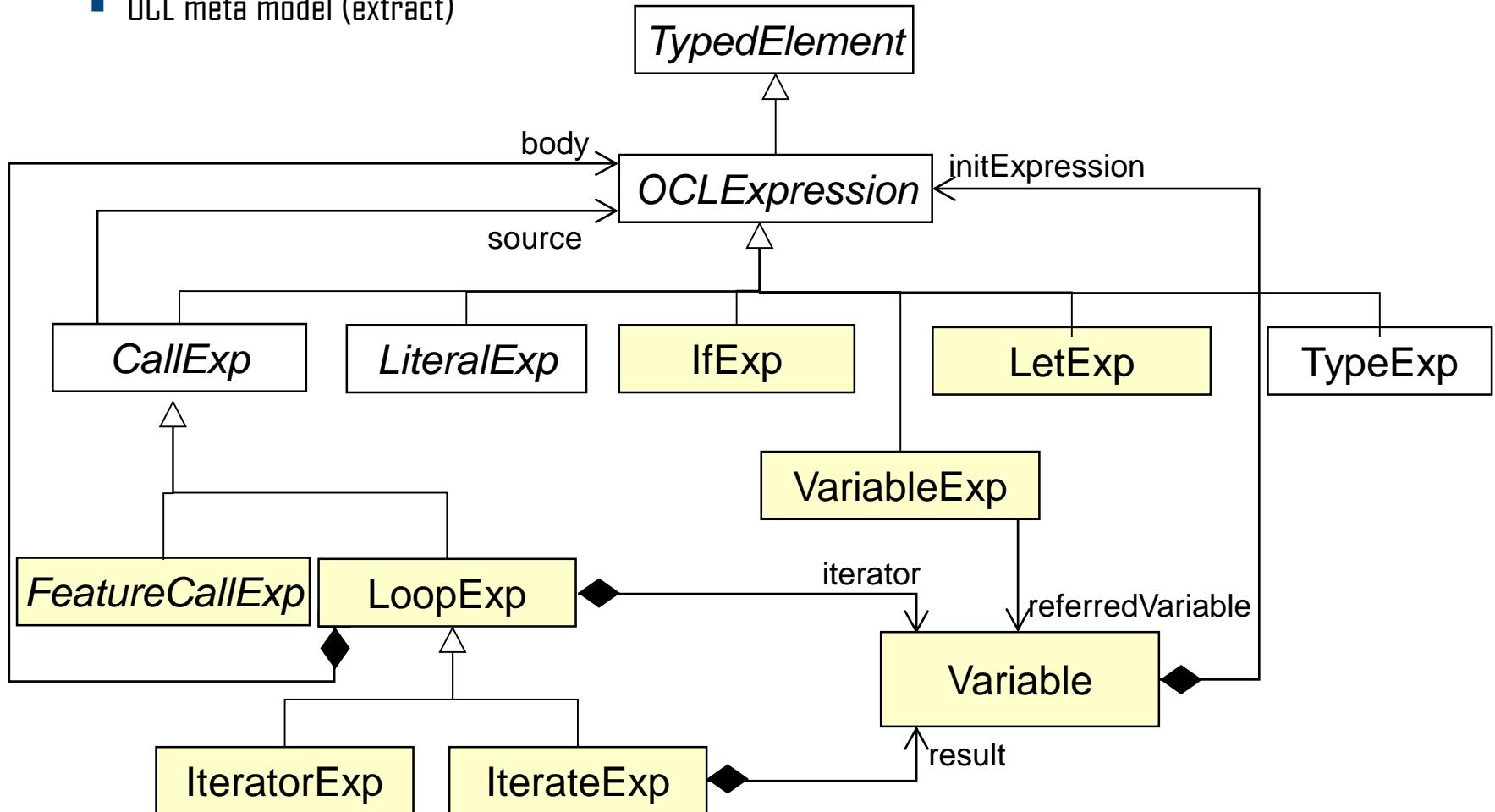
- Examples for LiteralExp, IfExp, VariableExp, AttributeCallExp



- Abstract syntax of DCL is described as meta model
- Mapping from abstract syntax to concrete syntax
  - » IfExp -> if Expression then Expression else Expression endif

# Expressions

- OCL meta model (extract)



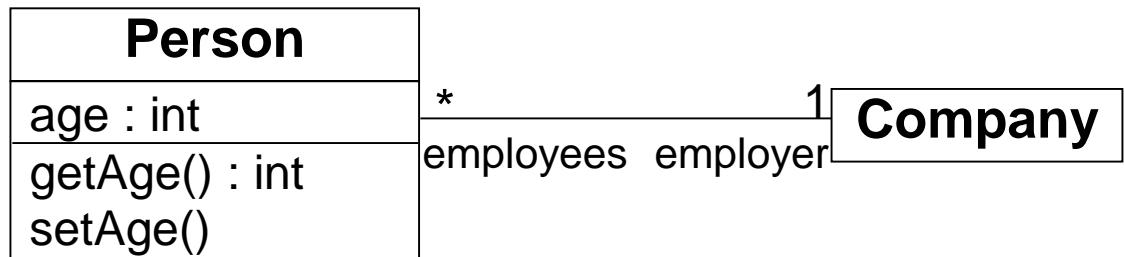
*LiteralExp*: CollectionLiteralExp, PrimitiveLiteralExp, TupleLiteralExp, EnumLiteralExp

170



## Query of model information

- Context instance
  - » context Person



- AttributeCallExp
  - » self.age : int

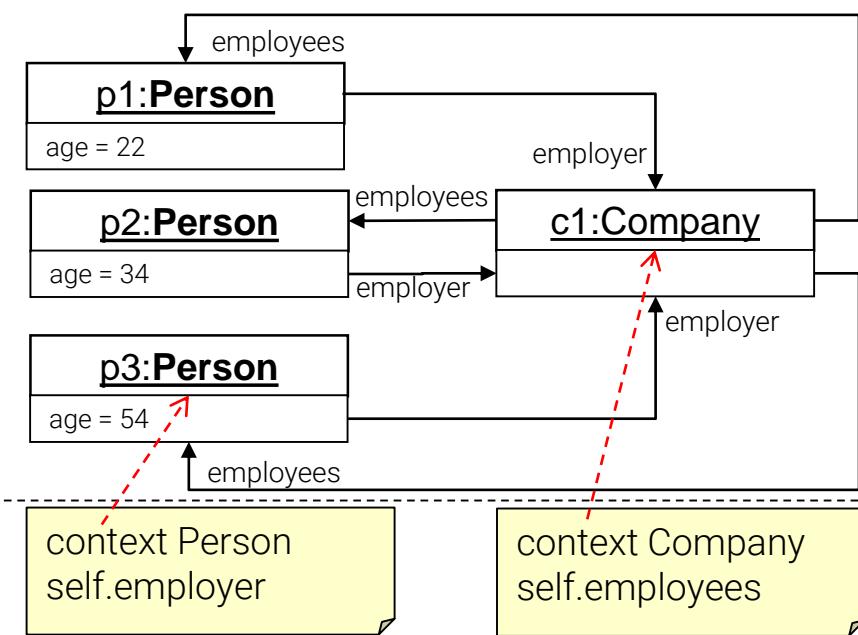
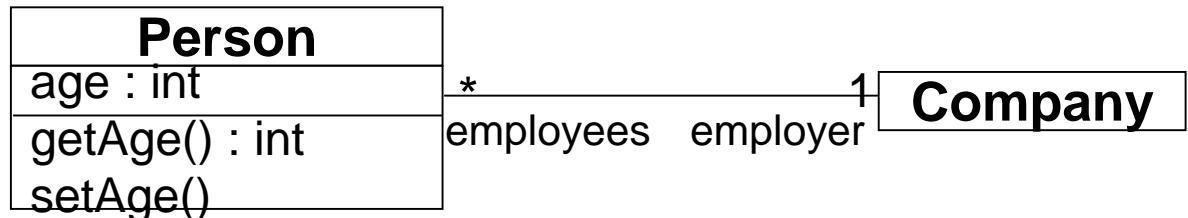
- OperationCallExp
  - » Operations must not have side effects
  - » Allowed: self.getAge() : int
  - » Not allowed: self.setAge()

- AssociationEndCallExp
  - » Navigate to the opposite association end using role names  
self.employer – Return value is of type Company
  - » Navigation often results into a set of objects – Example:  
context Company  
self.employees – Return value is of type Set (Person)



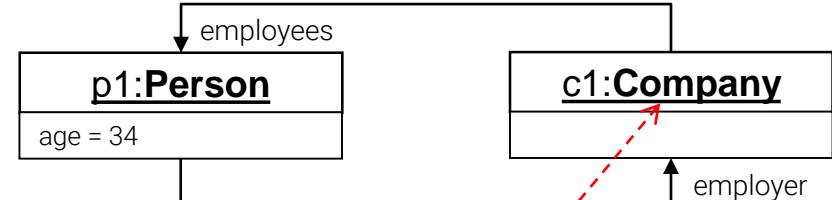
## Query of model information

- Example



**c1 : Company**

**Set{p1,p2,p3} :**  
**Set(Person)**

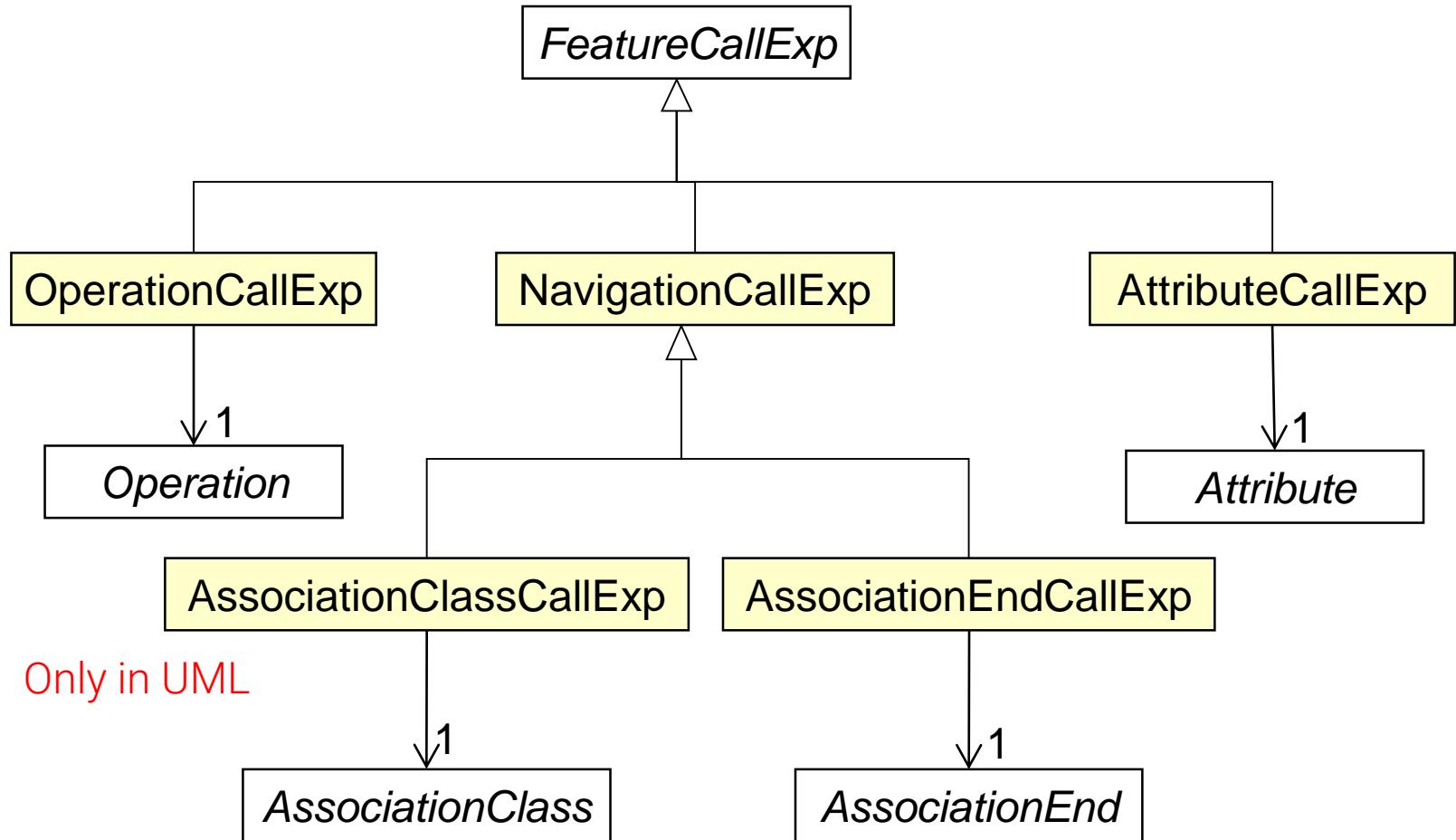


**context Company  
self.employees**

**Set{p1} :**  
**Set(Person)**

## Query of model information

- OCL meta model (extract)





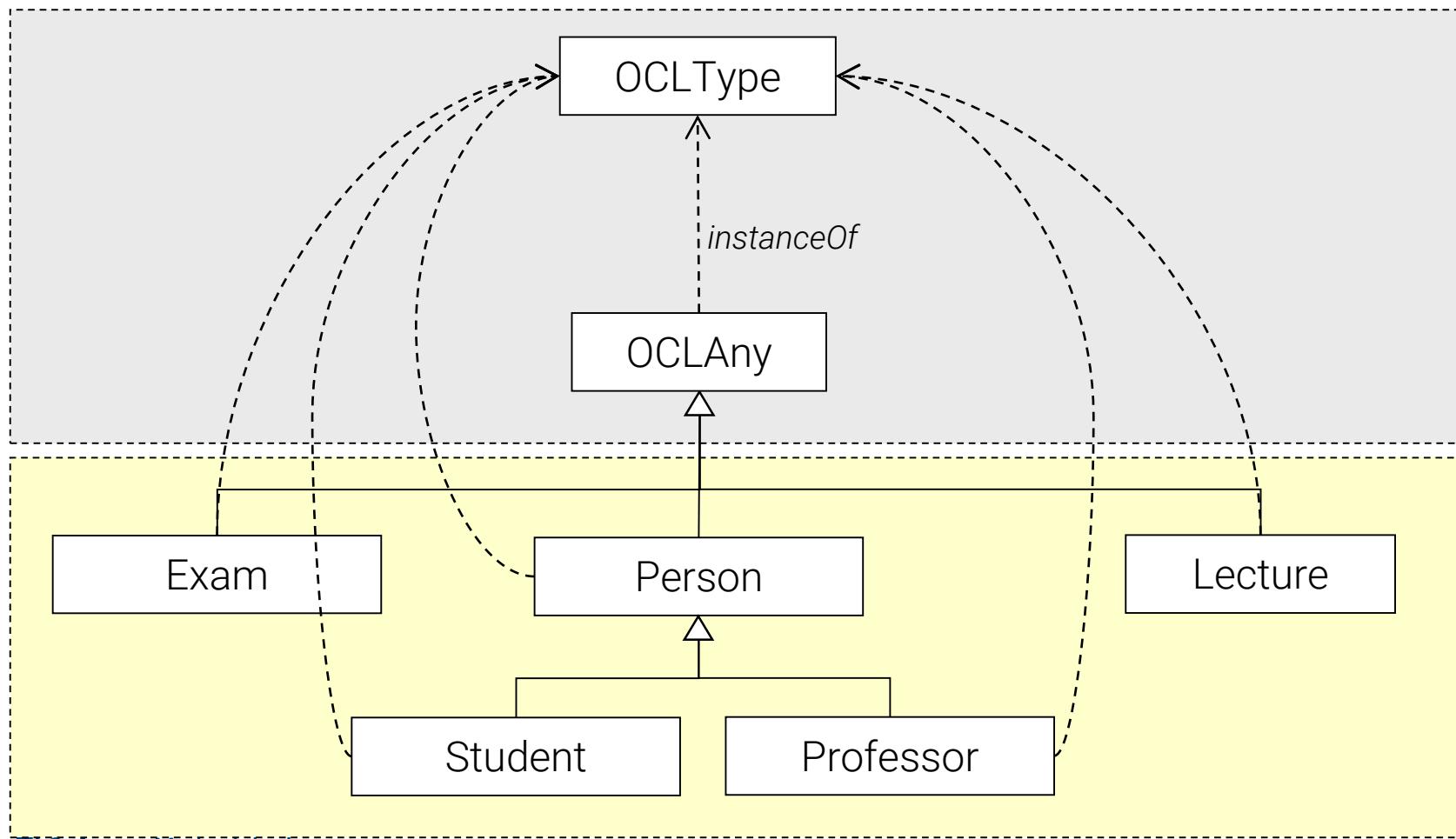
## OCL Library: Operations for OclAny

- **OclAny – Supertype of all other types in OCL**
  - » Operations are inherited by all other types
- **Operations of OclAny (extract)**
  - » Receiving object is denoted by *obj*

Operation	Explanation of result
$=(\text{obj2}:OclAny):\text{Boolean}$	True, if <i>obj2</i> and <i>obj</i> reference the same object
$\text{oclIsTypeOf}(\text{type}:OclType):\text{Boolean}$	True, if <i>type</i> is the type of <i>obj</i>
$\text{oclIsKindOf}(\text{type}:OclType):\text{Boolean}$	True, if <i>type</i> is a direct or indirect supertype or the type of <i>obj</i>
$\text{oclAsType}(\text{type}:OclType):\text{Type}$	The result is <i>obj</i> of type <i>type</i> , or <i>undefined</i> , if the current type of <i>obj</i> is not <i>type</i> or a direct or indirect subtype of it (casting)

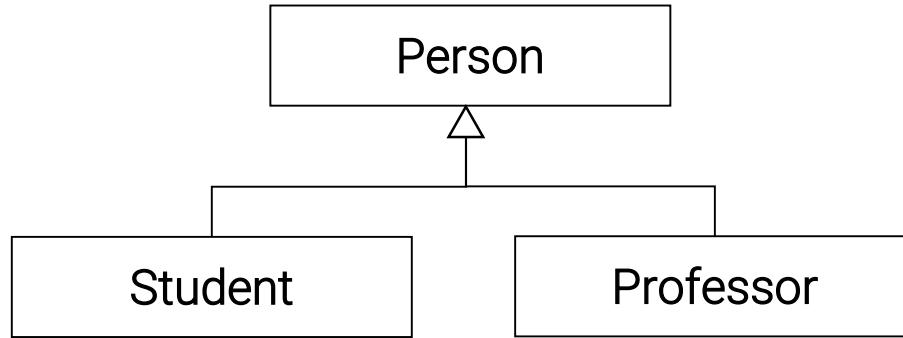
## Operations for OCLAny

- Predefined environment for model types



## Operations for OclAny

- `oclIsKindOf` vs. `oclIsTypeOf`



context **Person**

```

self.oclIsKindOf(Person) : true
self.oclIsTypeOf(Person) : true
self.oclIsKindOf(Student) : false
self.oclIsTypeOf(Student) : false
  
```

context **Student**

```

self.oclIsKindOf(Person) : true
self.oclIsTypeOf(Person) : false
self.oclIsKindOf(Student) : true
self.oclIsTypeOf(Student) : true
self.oclIsKindOf(Professor) : false
self.oclIsTypeOf(Professor) : false
  
```



# Operations for simple types

- Predefined simple types
  - » Integer {Z}
  - » Real {R}
  - » Boolean {true, false}
  - » String {ASCII, Unicode}
- Each simple type has predefined operations

---

Simple type	Predefined operations
Integer	$*$ , $+$ , $-$ , $/$ , $\text{abs}()$ , ...
Real	$*$ , $+$ , $-$ , $/$ , $\text{floor}()$ , ...
Boolean	and, or, xor, not, implies
String	$\text{concat}()$ , $\text{size}()$ , $\text{substring}()$ , ...



# Operations for simple types

## ■ Syntax

- » `v.operation(para1, para2, ...)`
  - Example: `"bla".concat("bla")`
- » Operations without brackets (Infix notation)
  - Example: `1 + 2, true and false`

---

Signature	Operation
$\text{Integer} \times \text{Integer} \rightarrow \text{Integer}$	$\{+, -, *\}$
$t1 \times t2 \rightarrow \text{Boolean}$	$\{\lt, \gt, \leq, \geq\}, t1, t2 \text{ typeOf } \{\text{Integer or Real}\}$
$\text{Boolean} \times \text{Boolean} \rightarrow \text{Boolean}$	$\{\text{and}, \text{or}, \text{xor}, \text{implies}\}$



# Operations for collections

- Collection is an abstract supertype for all set types
  - » Specification of the mutual operations
  - » Set, Bag, Sequence, OrderedSet inherit these operations
- Caution: Operations with a return value of a set-valued type create a new collection (no side effects)
- Syntax:  $v \rightarrow op(\dots)$  – Example:  $\{1, 2, 3\} \rightarrow size()$
- Operations of collections (extract)
  - » Receiving object is denoted by *coll*

Operation	Explanation of result
<i>size():Integer</i>	Number of elements in <i>coll</i>
<i>includes(obj:OclAny):Boolean</i>	True, if <i>obj</i> exists in <i>coll</i>
<i>isEmpty:Boolean</i>	True, if <i>coll</i> contains no elements
<i>sum:T</i>	Sum of all elements in <i>coll</i> Elements have to be of type Integer or Real



## Operations for collections

- Model operations vs. OCL operations



### OCL-Constraint

```
context Container  
inv: self.content -> first().isEmpty()
```

### Semantic

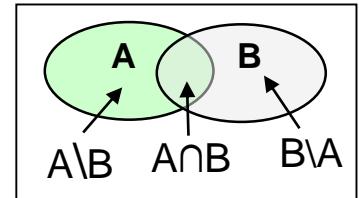
Operation *isEmpty()* always has to return true

```
context Container  
inv: self.content -> isEmpty()
```

Container instances must not contain bottles

# Operationen for Set/Bag

- Set and Bag define additional operations
  - » Generally based on theory of set concepts
- Operations of Set (extract)
  - » Receiving object is denoted by set



Operation	Explanation of result
<code>union(set2:Set(T)):Set(T)</code>	Union of set and set2
<code>intersection(set2:Set(T)):Set(T)</code>	Intersection of set and set2
<code>difference(set2:Set(T)):Set()</code>	Difference set; elements of set, which do not consist in set2
▪ Operations of Bag (extract)	
<del><code>symmetricDifference(bag2:Bag(T)):Bag(T)</code></del>	Set of all elements, which are either in set or in set2, but do not exist in both sets at the same time

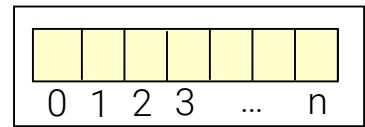
Operation	Explanation of result
<code>union(bag2:Bag(T)):Bag(T)</code>	Union of bag and bag2
<code>intersection(bag2:Bag(T)): Bag(T)</code>	Intersection of bag and bag2



# Operations for OrderedSet/Sequence

- OrderedSet and Sequences define additional operations
  - » Allow access or modification through an Index
- Operations of OrderedSet (extract)

~~» Receiving object is denoted by orderedSet~~



Operation	Explanation of result
<i>first:T</i>	First element of <i>orderedSet</i>
<i>last:T</i>	Last element of <i>orderedSet</i>
<i>at(i:Integer):T</i>	Element on index <i>i</i> of <i>orderedSet</i>
<i>subOrderedSet(lower:Integer, upper:Integer):OrderedSet(T)</i>	Subset of <i>orderedSet</i> , all elements of <i>orderedSet</i> including the element on position <i>lower</i> and the element on position <i>upper</i>
<i>insertAt(index:Integer,object:T) :OrderedSet(T)</i>	Result is a copy of the <i>orderedSet</i> , including the element <i>object</i> at the position <i>index</i>

- Operations of Sequence
  - » Analogous to the operations of OrderedSet



## Iterator-based operations

- OCL defines operations for Collections using Iterators
  - » Expression Package: LoopExp
  - » Projection of new Collections out of existing ones
  - » Compact declarative specification instead of imperative algorithms
- Predefined Operations
  - » select(exp) : Collection
  - » reject(exp) : Collection
  - » collect(exp) : Collection
  - » forAll(exp) : Boolean
  - » exists(exp) : Boolean
  - » isUnique(exp) : Boolean
- iterate(...) – Iterate over all elements of a Collection
  - » Generic operation
  - » Predefined operations are defined with iterate(...)



# Iterator-based operations

## Select-/Reject-Operation

- Select and Reject return subsets of collections
  - » Iterate over the complete collection and collect elements
- Select
  - » Result: Subset of collection, including elements where booleanExpr is true

### ■ Reject

- » Result: Subse

*collection -> select( v : Type | booleanExp(v) )*

*collection -> select( v | booleanExp(v) )*

*collection -> select( booleanExp )*

- » Just Syntactic Sugar, because each reject-Operation can be defined as a select-Operation with a negated expression

*collection-> reject(v : Type | booleanExp(v))*

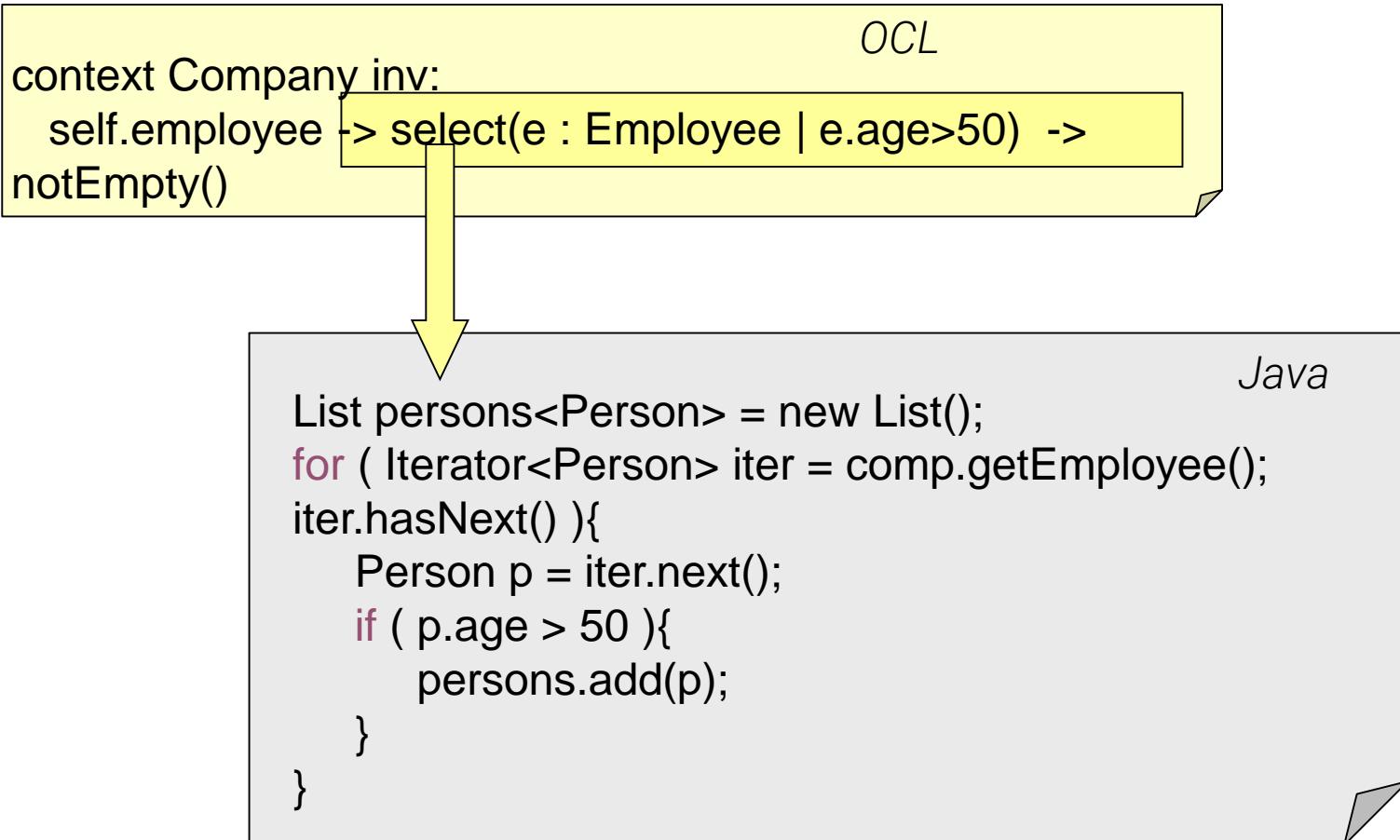
*collection-> select(v : Type | **not** (booleanExp(v)))*



# Iterator-based operations

## Semantic of the Select-Operation

- Select-/Reject-Operation





## Iterator-based operations

- Collect-Operation returns a new collection from an existing one. It collects the Properties of the objects and not the objects itself.
  - » Result of collect always Bag<T>.T defines the type of the property to be collected

```
collection -> collect( v : Type | exp(v) )  
collection -> collect( v | exp(v) )  
collection -> collect( exp )
```

- Example
  - » self.employees -> collect(age) – Return type: Bag(Integer)
- Short notation for collect
  - » self.employees.age



# Iterator-based operations

## Semantic of the Collect-Operator

OCL

```
context Company inv:  
    self.employee -> collect(birthdate) -> size() > 3
```

Java

```
List birthdate<Integer> = new List();  
for ( Iterator<Person> iter = comp.getEmployee();  
iter.hasNext() ){  
    birthdate.add(iter.next().getBirthdate()); }
```

OCL

```
context Company inv:  
    self.employee -> collect(birthdate) -> asSet()
```

**Bag**  
(with duplicates)

**Set**  
(without  
duplicates)



## Iterator-based operations

- ForAll checks, if all elements of a collection evaluate to true

```
collection -> forAll( v : Type | booleanExp(v) )
```

```
collection -> forAll( v | booleanExp(v) )
```

```
collection -> forAll( booleanExp )
```

Example: self.employees -> forall(age > 10)

- Nesting of forAll-Calls (Cartesian Product)

```
context Company inv:
```

```
self.employee->forAll (e1 | self.employee -> forAll (e2 |
e1 <> e2 implies e1.svnr <> e2.svnr))
```

```
context Company inv:
```

```
self.employee -> forAll (e1, e2 | e1 <> e2 implies e1.svnr <> e2.svnr))
```

» Beispiel: employees -> exists(e: Employee | e.isManager = true)



# Iterator-based operations

## Iterate-Operation

- Iterate is the generic form of all iterator-based operations

- Syntax

```
collection -> iterate( elem : Typ; acc : Typ =  
    <initExp> | exp(elem, acc) )
```

- Variable elem is a typed Iterator
- Variable acc is a typed Accumulator
  - Gets assigned initial value initExp
- exp(elem, acc) is a function to calculate acc

- Example

```
» collection -> collect( x : T | x.property )
```

-- semantically equivalent to:

```
» collection -> iterate( x : T; acc : T2 = Bag{} | acc -> including(x.property) )
```



# Iterator-based operations

## Semantic of the Iterate-Operator

- Semantic of the Iterate-Operator

OCL

collection -> iterate(x : T; acc : T2 = value | acc -> u(acc, x))

Java

```
iterate (coll : T, acc : T2 = value) {  
    acc=value;  
    for( Iterator<T> iter =  
        coll.getElements(); iter.hasNext(); ) {  
        T elem = iter.next();  
        acc = u(elem, acc);  
    }  
}
```

- Example
  - » Set{1, 2, 3} ->
  - » Result: 6



# Tool Support

- **Wishlist**
  - » Syntactic analysis: Editor support
  - » Validation of logical consistency (Unambiguous)
  - » Dynamic validation of invariants
  - » Dynamic validation of Pre-/Post-conditions
  - » Code generation and test automation
- **Today**
  - » UML-tools provide OCL-editors
  - » MDA-tools provide code generation of OCL-expressions
  - » Meta modeling platforms provide the opportunity to define OCL Constraints for meta models.
    - The editor should dynamically check constraints or restrict modeling, respectively.

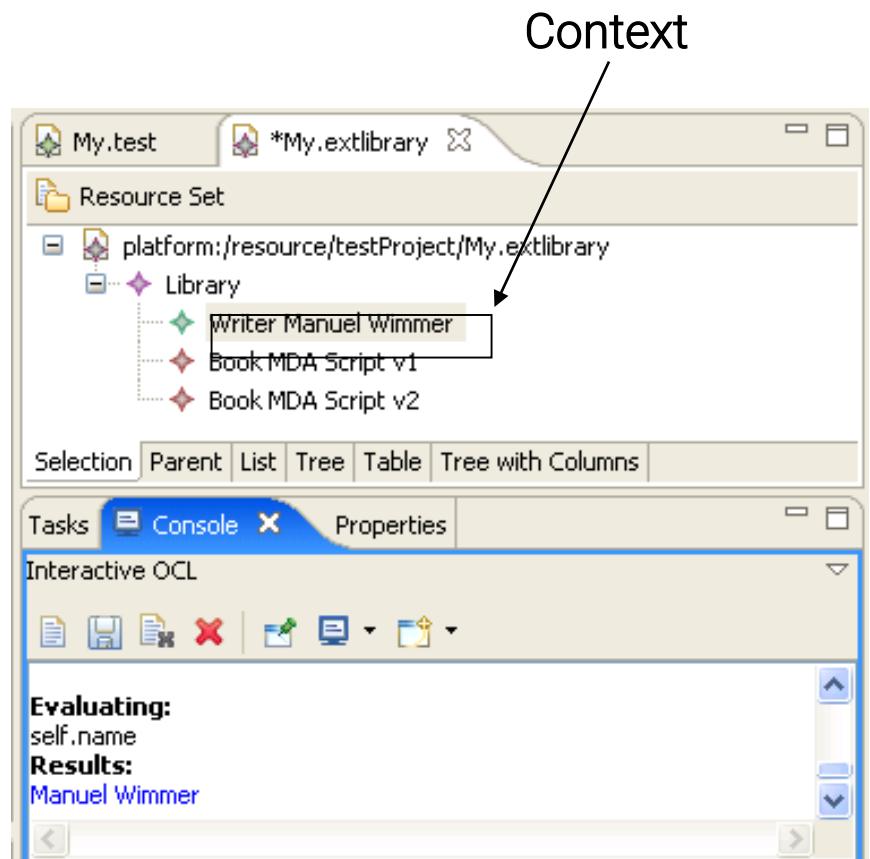


## OCL Tools

- Some OCL-parsers, which check the syntax of OCL-constraints and apply them to the models, are for free.
  - » IBM Parser
- Dresden OCL Toolkit 2.0
  - » Generation of Java code out of OCL-constraints
  - » Possible integration with ArgoUML
- OCL-frameworks are originated in the areas of EMF and the UML2 project of Eclipse
  - » Octopus
  - » Fraunhofer Toolkit
  - » OSLO
  - » EMFT OCL-Framework/Query-Framework

# OCL-Tools

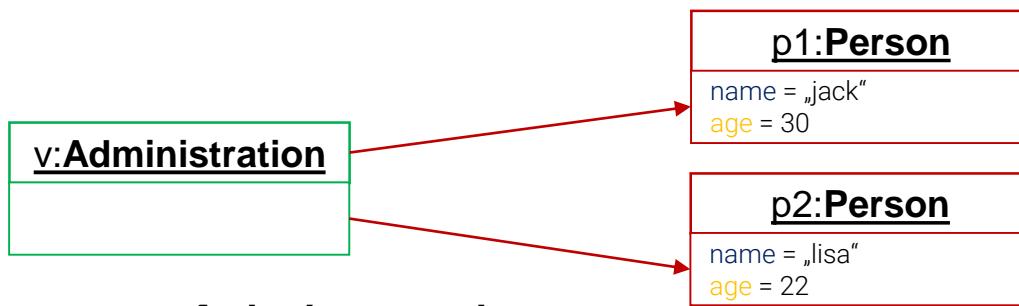
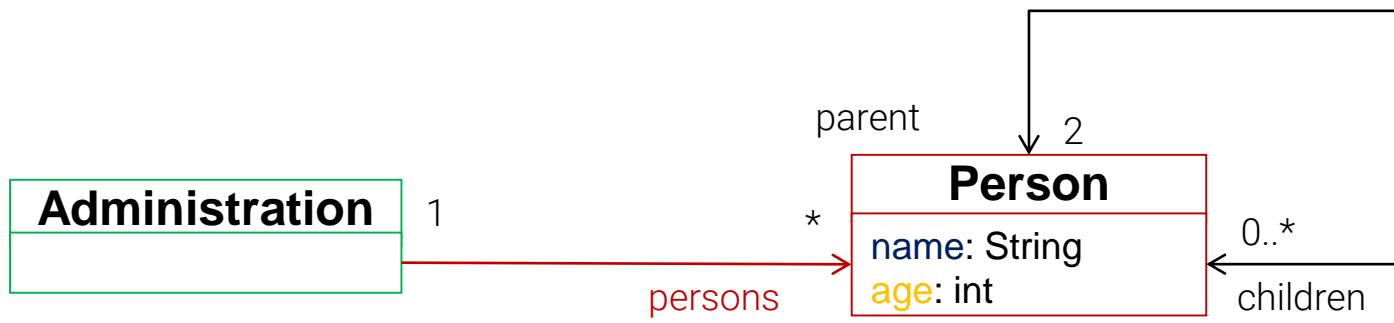
- EMFT OCL-Framework
  - » Based on EMF
  - » OCL-API – Enables the use of OCL in Java programs
  - » Interactive OCL Console – Enables the definition and evaluation of OCL-constraints
  
- EMFT Query-Framework
  - » Goal: SQL-like query of model information
  - » select exp from exp where oclExp



TUWEL: Interactive OCL Console  
Screencast

## Example 1: Navigation (1)

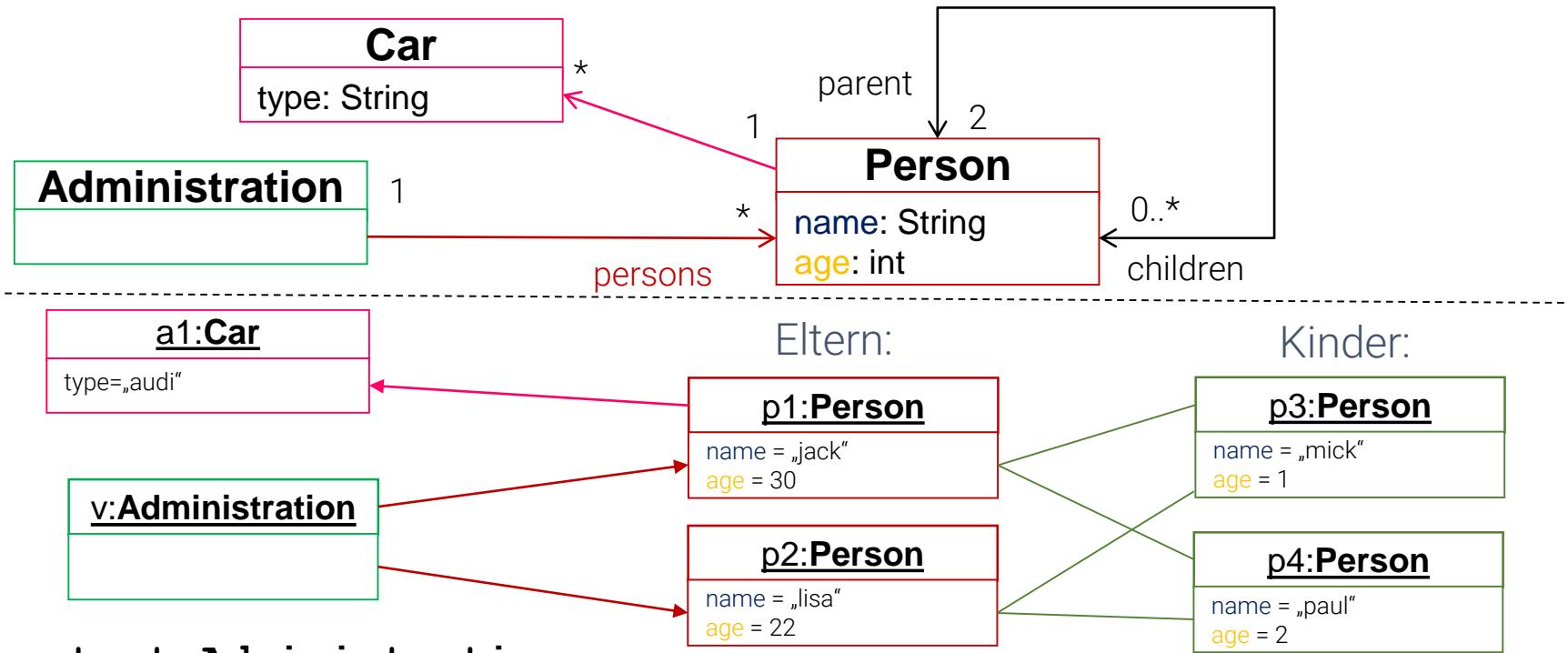
- `self.persons` → {Person p1, Person p2}
- `self.persons.name` → {jack, lisa}
- `self.persons.alter` → {30, 22}



**context Administration:**

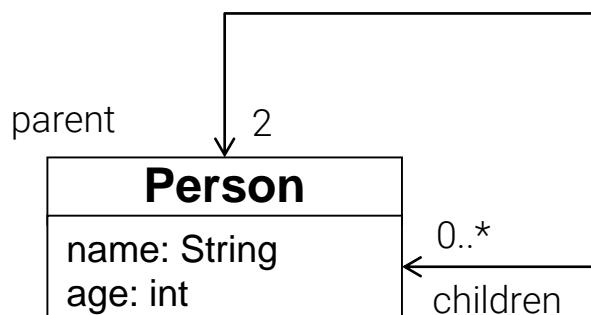
## Example 1: Navigation (2)

- `self.persons.children` → `{[{p3, p4}, {p3, p4}]}`
- `self.persons.children.parent` → `{[{p1, p2}, {p1, p2}], ...}`
- `self.persons.car.type` → `{"audi"}`

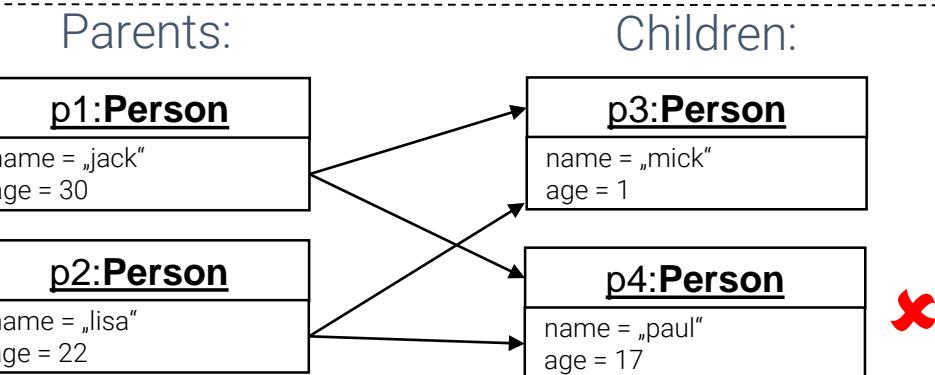


## Example 2: Invariant (I)

- context Person
- inv: self.children->forAll(k : Person | k.age < self.age-15)



**Constraint:** A child is at least 15 years younger than his parents.

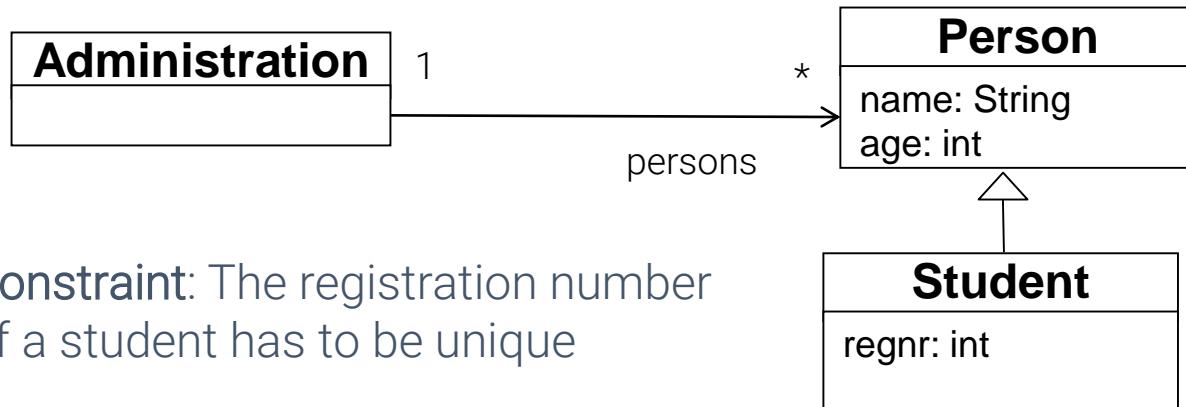


## Example 2: Invariant (2)

context Administration

inv uniqueRegnr :

`self.persons -> select(e : Person| e.oclIsTypeOf(Student))`  $\rightarrow \text{forAll}(e1 |$   
`self.persons -> select(e : Person | e.oclIsTypeOf(Student)) -> \text{forAll}(e2 |`  
`e1 <> e2 implies e1.ocIAstType(Student).regnr`  $\leftrightarrow$   
`e2.ocIAstType(Student).regnr))`



Constraint: The registration number of a student has to be unique

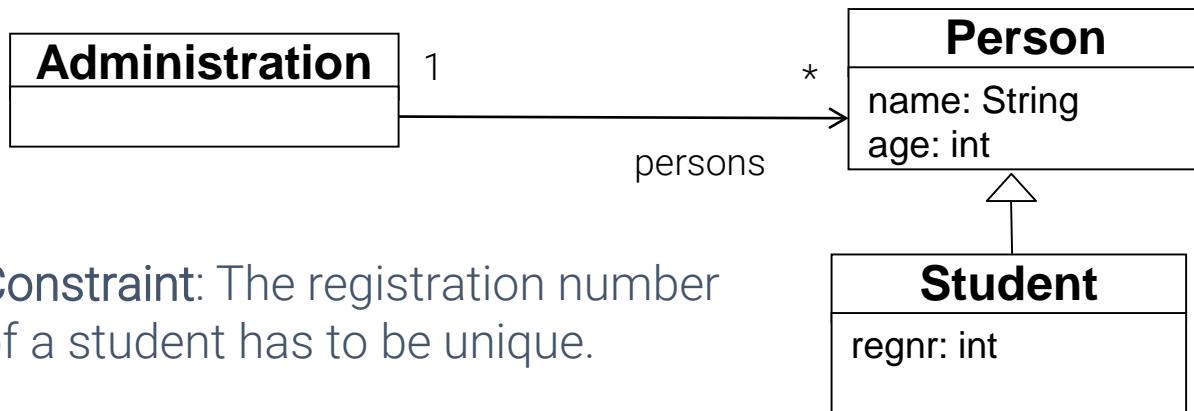


## Example 2: Invariant (2) cont.

context Administration

inv uniqueRegnr :

```
    self.persons -> select(e : Person| eoclIsTypeOf(Student)) -> forAll(e1, e1 | e1 <> e2  
implies                                e1.oclAsType(Student).regnr <> e2.oclAsType(Student).regnr)  
)
```

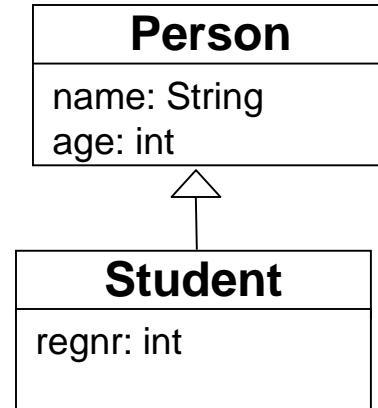


Constraint: The registration number of a student has to be unique.

## Example 2: Invariant (2) cont.

- context Student
- inv uniqueRegnr :
- `Student.allInstances() -> forAll(e1, e1 | e1 <> e2 implies e1oclAsType(Student).regnr <> e2.oclAsType(Student).regnr)`

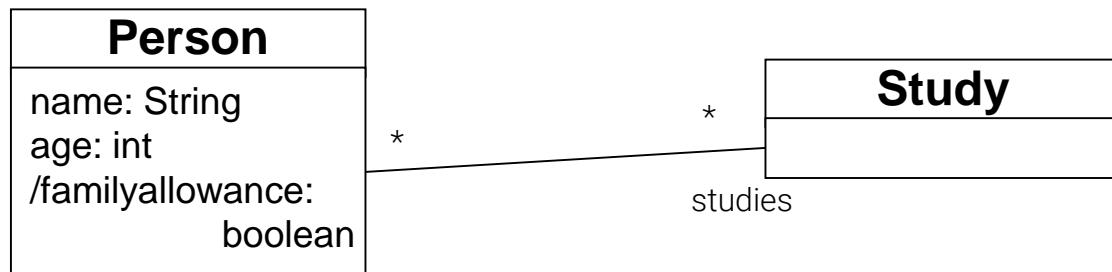
Constraint: The registration number of a student has to be unique.



## Example 3: Inherited attribute

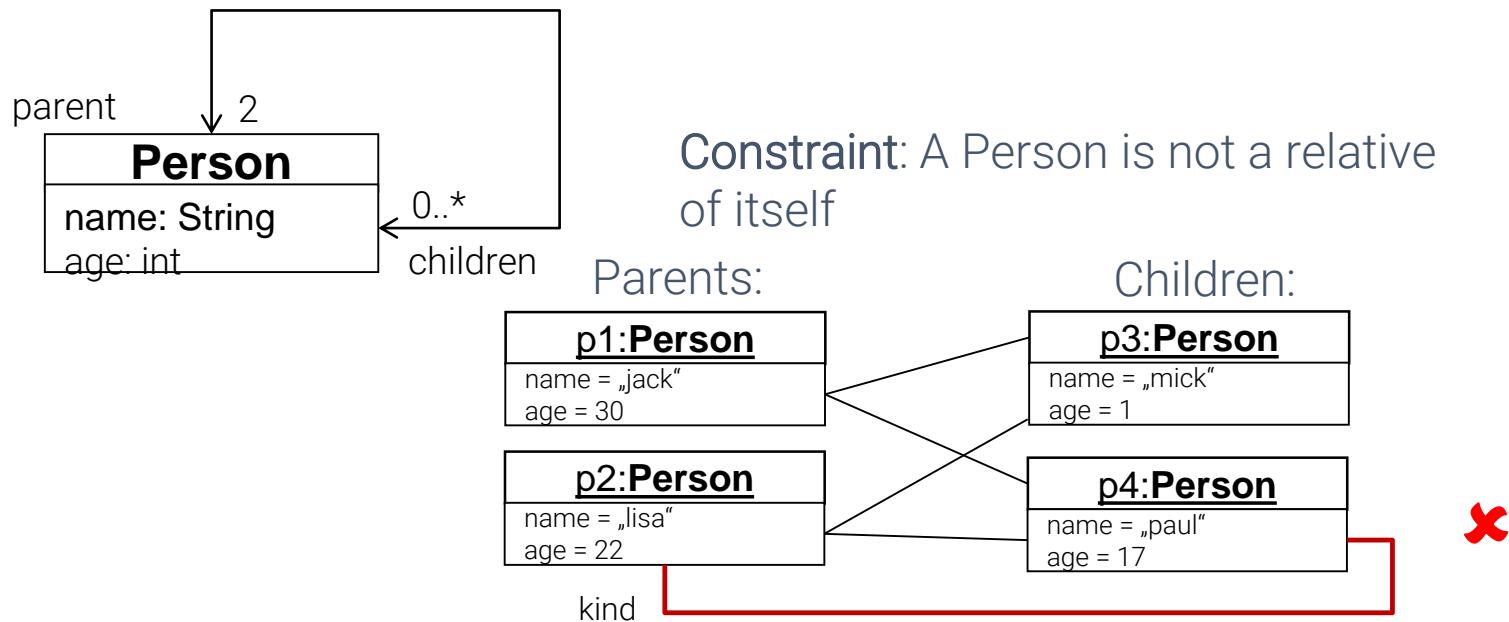
- context Person::familyallowance
- derive: self.age < 18 or
- (self.age < 27 and self.studies -> size() > 0)

A Person obtains family allowance, if he/she is younger than 18 years, or if he/she is studying and younger than 27 years old.



## Example 4: Definitions

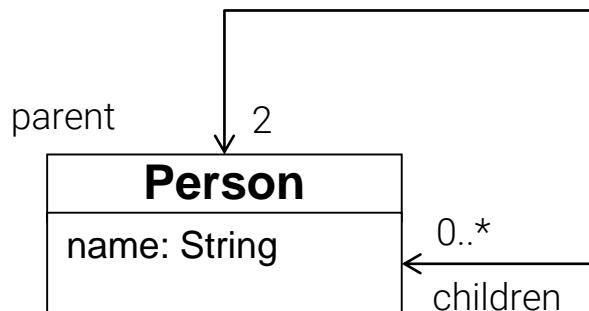
- context Person
- def: relative: Set(Person) = children-> union(relative)
- inv: self.relative -> excludes(self)



Assumption: Fixed-point semantic, otherwise if then else required

## Example 5: equivalent OCL-formulations (1)

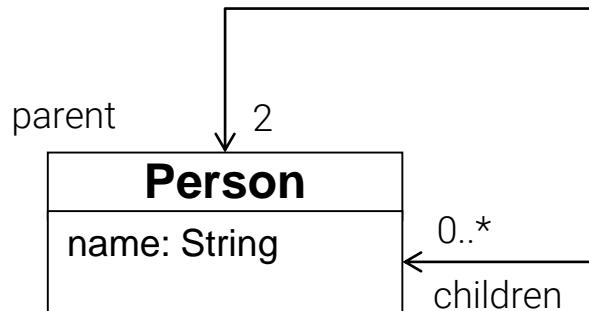
- $(\text{self}.children \rightarrow \text{select}(k \mid k = \text{self})) \rightarrow \text{size}() = 0$
- The Number of children for each person „self“, where the children are the person „self“, have to be 0.
  
- $(\text{self}.children \rightarrow \text{select}(k \mid k = \text{self})) \rightarrow \text{isEmpty}()$
- The set of children for each person „self“, where the children are the person „self“, has to be empty.



Constraint: A person is not its own child

## Example 5: equivalent OCL-formulations (2)

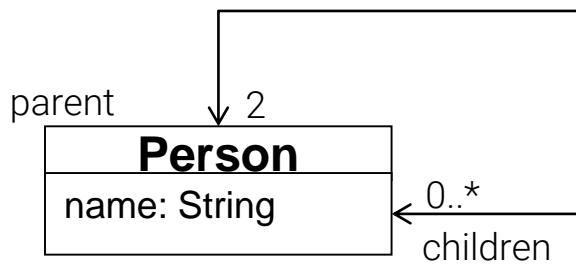
- `not self.children->includes(self)`
- It is not possible, that the set of children of each person „self“ contains the person „self“.
  
- `self.children->excludes(self)`
- The set of children of each person „self“ cannot contain „self“.



Constraint: A person is not its own child

## Example 5: equivalent OCL-formulations (3)

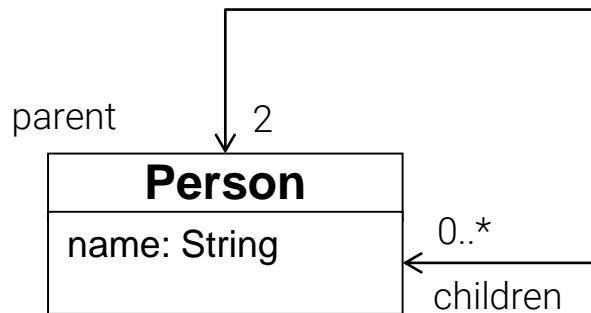
- `Set{self}->intersection(self.children)->isEmpty()`
- The intersection between the one element set, which only includes one person „self“ and the set of the children of „self“ has to be empty.
- `(self.children->reject(k | k <> self))->isEmpty()`
- The set of children for each person „self“, for whom does not apply, that they are not equal to the person „self“, has to be empty.



Constraint: A person is not its own child

## Example 5: equivalent OCL-formulations (4)

- `self.children->forAll(k | k <> self)`
- Each child of the person „self“ is not the person „self“.
  
- `not self.children->exists(k | k = self)`
- There is no child for each person „self“, which is the person „self“

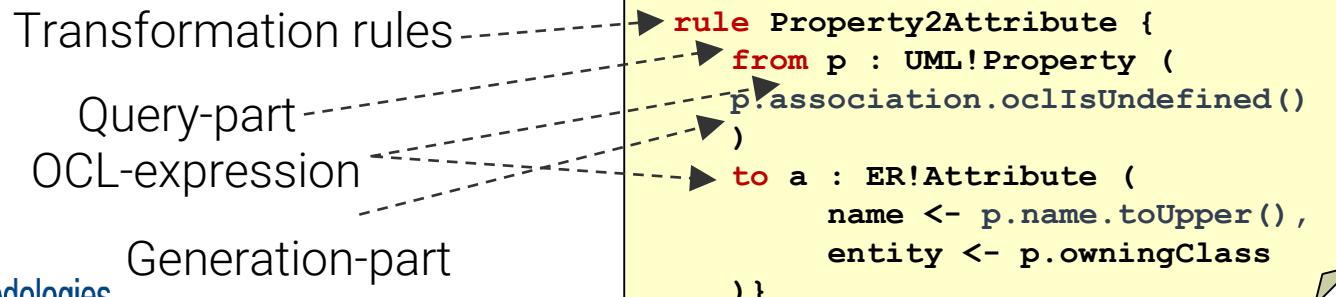


Constraint: A person is not its own child



## Outlook - Next unit: ATLAS Transformation Language (ATL)

- **Query-part (from) – What has to be transformed?**
  - » When selecting the relevant model elements
    - additionally to the type indication, constraints on attributes and association ends are required,
    - which are specified in OCL.
- **Generation-part(to) – What has to be created?**
  - » When creating the target structure
    - additionally to the type information, derived information are required,
    - which are calculated in OCL.





# References on OCL

## ■ Literature

- » Object Constraint Language Specification, Version 2.0
  - <http://www.omg.org/technology/documents/formal/ocl.htm>
- » Jos Warmer, Anneke Kleppe: The Object Constraint Language - Second Edition, Addison Wesley (2003)
- » Martin Hitz et al: UML@Work, d.punkt, 2. Auflage (2003)

## ■ Tools

- » OSLO - <http://oslo-project.berlios.de>
- » Octopus - <http://octopus.sourceforge.net>
- » Dresden OCL Toolkit - <http://dresden-ocl.sourceforge.net>
- » EMF OCL - <http://www.eclipse.org/modeling/mdt/?project=ocl>



# CHAPTER 6 -

## Developing your Own Modeling Language



# Overview

## 1 Introduction

- 1.1 Human Cognitive Processes
- 1.2 Models
- 1.3 Model Engineering

## 2 MDSE Principles

- 2.1 MDSE Basics
- 2.2 The MD\* Jungle of Acronyms
- 2.3 Modeling Languages and Metamodeling
- 2.4 Overview Considered Approaches
- 2.5 Tool Support
- 2.6 Criticisms of MDSE

## 3 MDSE Use Cases

- 3.1 MDSE applications
- 3.2 USE CASE1 – Model driven development
- 3.3 USE CASE2 – Systems interoperability
- 3.4 USE CASE3 – Model driven reverse engineering



# Overview

## 4 Model-Driven Architecture (MDA)

- 4.1 MDA Definitions and Assumptions
- 4.2 The Modeling Levels: CIM, PIM, PSM
- 4.3 Architecture-Driven Modernization

## 5 Modeling Languages at a Glance

- 5.1 Anatomy of Modeling Languages
- 5.2 General Purpose vs. Domain-Specific Modeling Languages
- 5.3 Overview of UML Diagrams
- 5.4 UML Behavioural or Dynamic Diagrams
- 5.5 UML Extensibility: The Middle Way Between GPL and DSL
- 5.6 Domain Specific Languages
- 5.7 Defining Modeling Constraints (OCL)



# Overview

## 6 Developing your Own Modeling Language

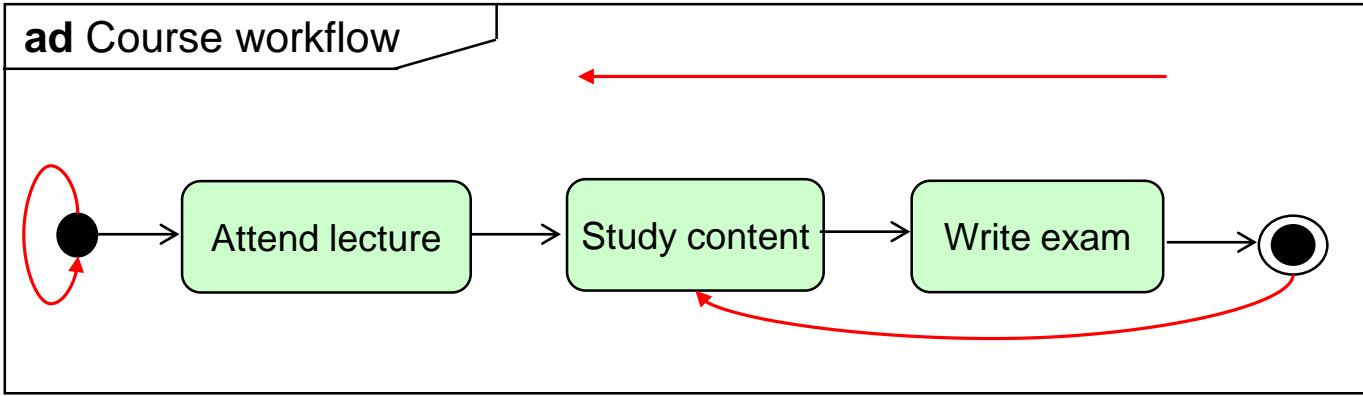
- 6.1 Metamodel-Centric Language Design
- 6.2 Programming Languages
- 6.3 Metamodel Development Process
- 6.4 MOF - Meta Object Facility
- 6.5 Example DSML: sWML
- 6.6 EMF and Ecore
- 6.7 Abstract Syntax Development
- 6.8 Graphical Concrete Syntax Development
- 6.9 Textual Concrete Syntax Development

## 7 Model-to-Model Transformations

- 7.1 Model Transformations and their Classification
- 7.2 Exogenous, Out-Place Transformations
- 7.3 Endogenous, In-Place Transformations
- 7.4 Mastering Model Transformations and their Rules

# Introduction

- Motivating example: a simple UML Activity diagram
  - Activity, Transition, InitialNode, FinalNode

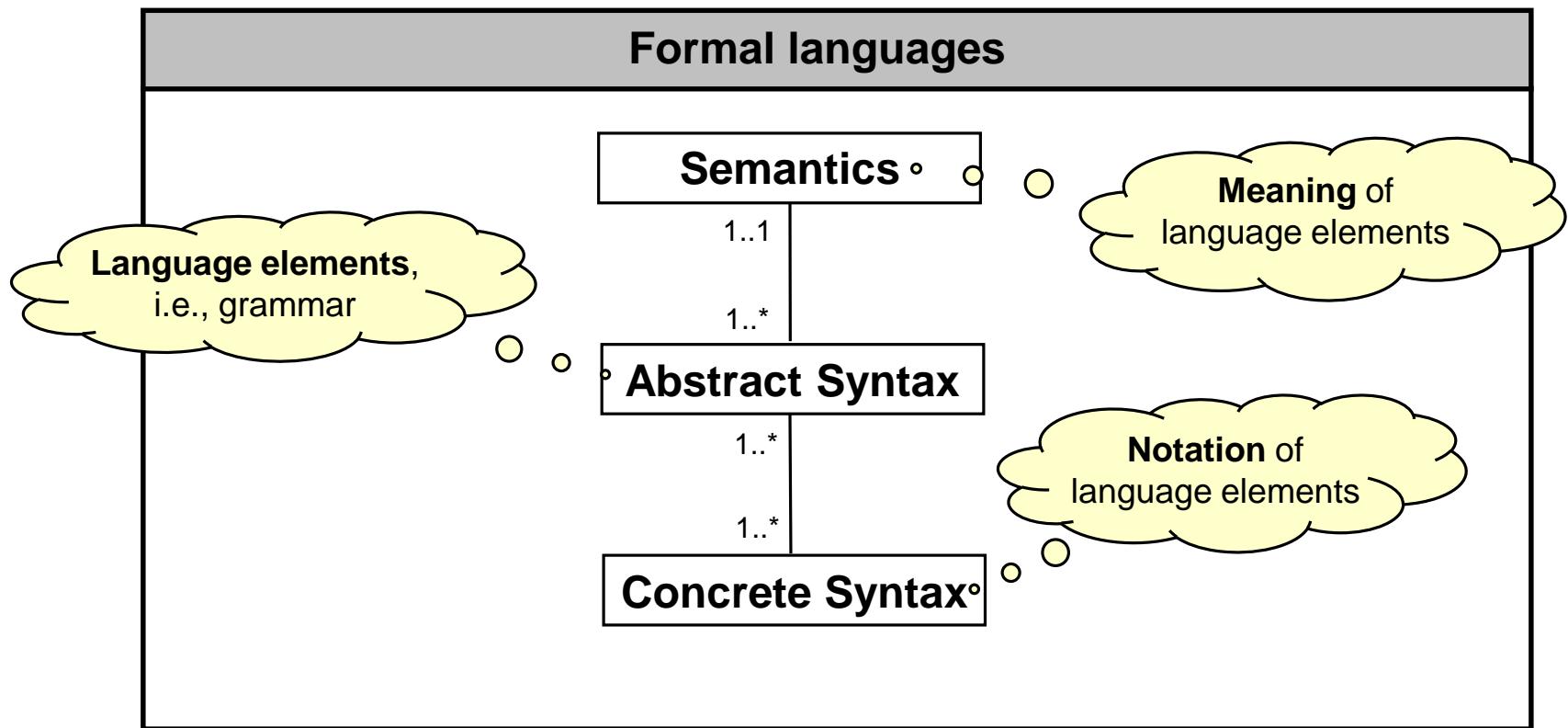


- Question: Is this UML Activity diagram valid?
- Answer: Check the UML metamodel!
  - Prefix „meta“: an operation is applied to itself
  - Further examples: meta-discussion, meta-learning, ...
- Aim of this lecture: Understand what is meant by the term „metamodel“ and how metamodels are defined.



# Introduction

Languages have divergent goals and fields of application, but still have a common definition framework





# Introduction

## ■ Main components

- » Abstract syntax: Language concepts and how these concepts can be combined (~ grammar)
  - It does neither define the notation nor the meaning of the concepts
- » Concrete syntax: Notation to illustrate the language concepts intuitively
  - Textual, graphical or a mixture of both
- » Semantics: Meaning of the language concepts
  - How language concepts are actually interpreted

## ■ Additional components

- » Extension of the language by new language concepts
  - Domain or technology specific extensions, e.g., see UML Profiles
- » Mapping to other languages, domains
  - Examples: UML2Java, UML2SetTheory, PetriNet2BPEL, ...
  - May act as translational semantic definition



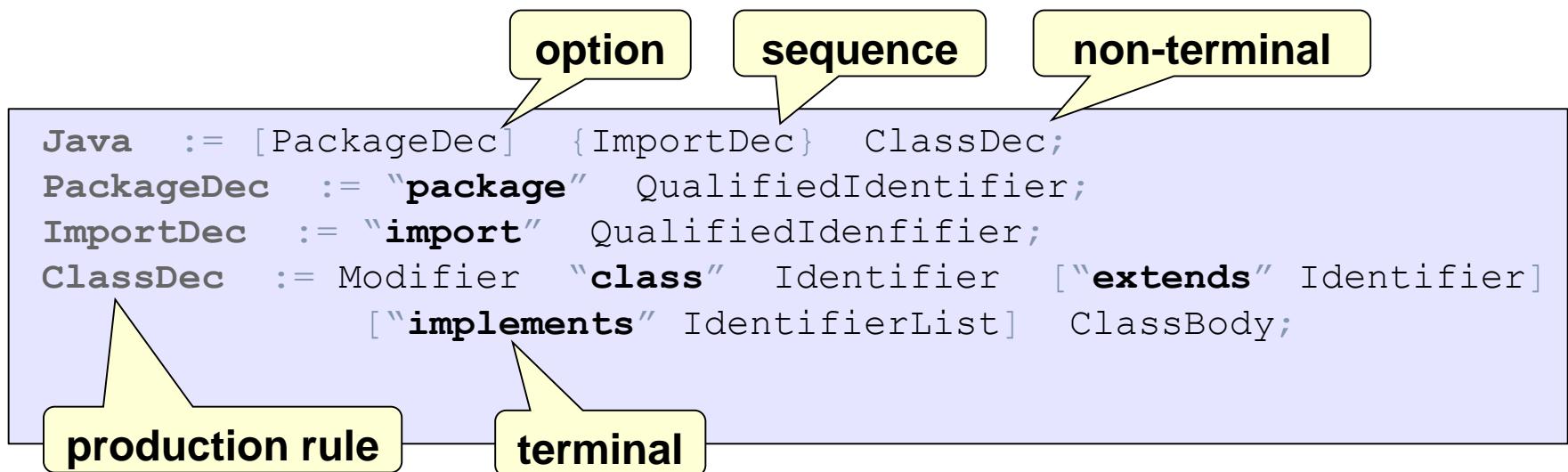
## Excursus: Meta-languages in the Past

- Formal languages have a long tradition in computer science
- First attempts: Transition from machine code instructions to high-level programming languages (Algol60)
- Major successes
  - » Programming languages such as Java, C++, C#, ...
  - » Declarative languages such as XML Schema, DTD, RDF, OWL, ...
- Excursus
  - » How are programming languages and XML-based languages defined?
  - » What can thereof be learned for defining modeling languages?



# Programming languages

- John Backus and Peter Naur invented formal languages for the definition of languages called meta-languages
- Examples for meta-languages: BNF, EBNF, ...
- Are used since 1960 for the definition of the syntax of programming languages
  - » Remark: abstract and the concrete syntax are both defined
- EBNF Example





# Programming languages

## Example: MiniJava

- Grammar

```
Java := [ PackageDec ] { ImportDec } ClassDec;
PackageDec := "package" QualifiedIdentifier;
ImportDec := "import" QualifiedIdentifier;
ClassDec := Modifier "class" Identifier ["extends" Identifier]
           ["implements" IdentifierList] ClassBody;
Modifier := "public" | "private" | "protected";
Identifier := {"a"-"z" | "A"-"Z" | "0"-"9"}
```

- Program

```
package mdse.book.example;
import java.util.*;
public class Student extends Person { ... }
```

- Validation: does the program conform to the grammar?

- Compiler: javac, gcc, ...
- Interpreter: Ruby, Python, ...



# Programming languages

- Four-layer architecture: Meta-architecture layers

```
EBNF := {rules};  
rules := Terminal | Non-Terminal | ...
```

**Definition of EBNF in  
EBNF – EBNF grammar  
(reflexive)**

M3-Layer

```
Java := [PackageDec]  
      {ImportDec} ClassDec;  
PackageDec := "package"  
QualifiedIdentifier; ...
```

**Definition of Java in  
EBNF – Java grammar**

M2-Layer

```
package big.tuwien.ac.at;  
public class Student  
    extends Person { ... }
```

**Program – Sentence  
conform to the grammar**

M1-Layer



**Execution of the  
program**

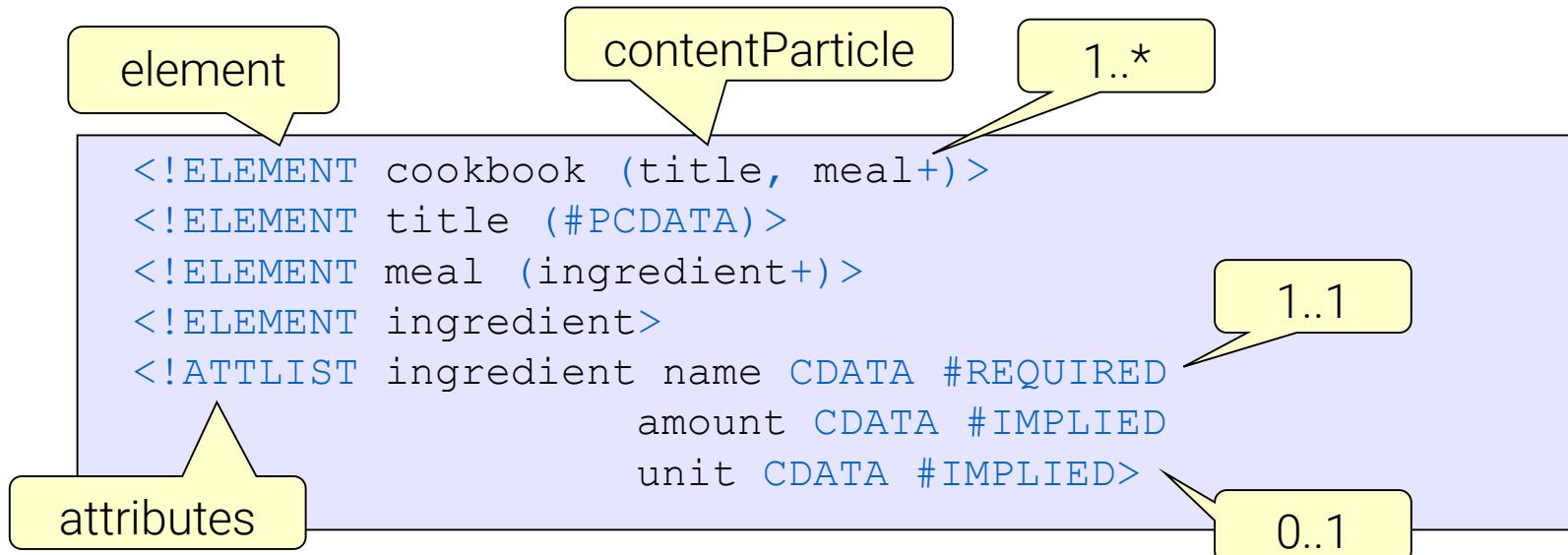
M0-Layer



# XML-based languages

## Overview

- XML files require specific structures to allow for a standardized and automated processing
- Examples for XML meta languages
  - » DTD, XML-Schema, Schematron
- Characteristics of XML files
  - » Well-formed (character level) vs. valid (grammar level)
- DTD Example





# XML-based languages

## DTD

- Example: Cookbook DTD

```
<!ELEMENT cookbook (title, meal+)>
<!ELEMENT title (#PCDATA)>
<!ELEMENT meal (ingredient+)>
<!ELEMENT ingredient>
<!ATTLIST ingredient name CDATA #REQUIRED
                    amount CDATA #IMPLIED
                    unit CDATA #IMPLIED>
```

```
<cookbook>
    <title>How to cook!</title>
    <meal name= „Spaghetti“ >
        <ingredient name = „Tomato“, amount=„300“ unit=„gramm“>
        <ingredient name = „Meat“, amount=„200“ unit=„gramm“> ...
    </meal>
</cookbook>
```



# XML-based languages: Five-layer architecture (was revised with XML-Schema)

- Meta-architecture layers

```
EBNF := {rules};  
rules := Terminal | Non-Terminal | ...
```

**Definition of EBNF  
in EBNF**

M4-Layer

```
ELEMENT := „<!ELEMENT “ Identifier „>“  
          ATTLIST;  
ATTLIST := „<!ATTLIST “ Identifier ...
```

**Definition of DTD  
in EBNF**

M3-Layer

```
<!ELEMENT javaProg (packageDec*,  
importDec*, classDec)>  
<!ELEMENT packageDec (#PCDATA)>
```

**Definition of Java in  
DTD – Grammar**

M2-Layer

```
<javaProg>  
  <packageDec>big.tuwien.ac.at</packageDec>  
  <classDec name=„Student“ extends=„Person“/>  
</javaProg>
```

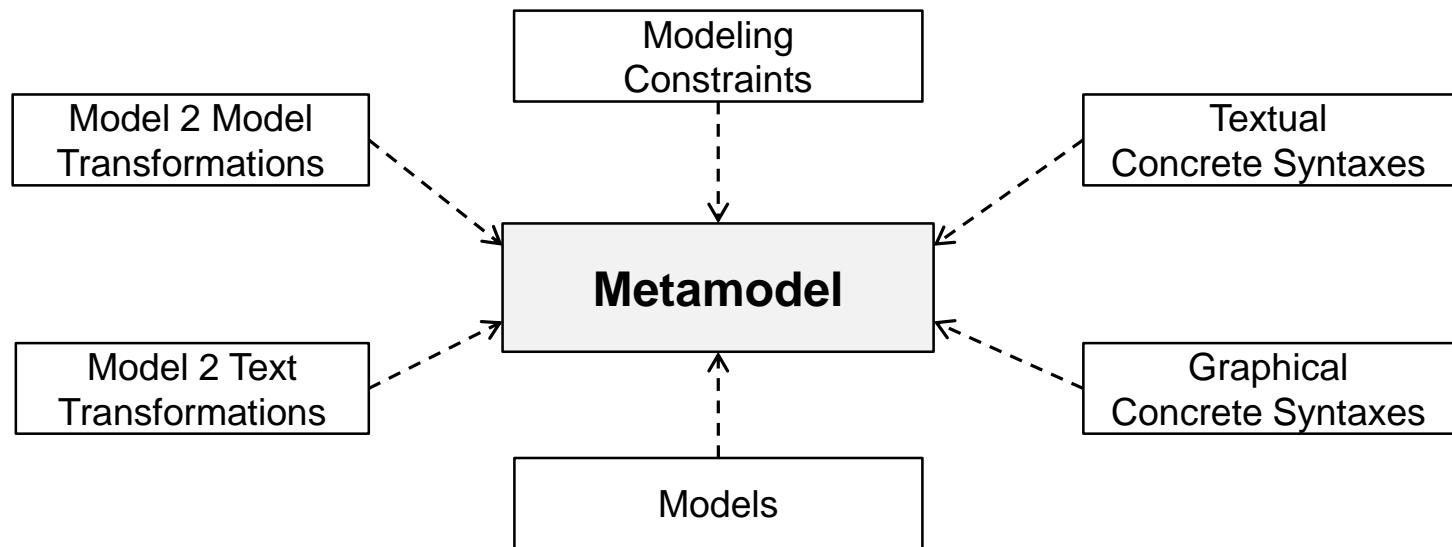
**XML –  
conform to the DTD**

M1-Layer

Concrete entities (e.g.: Student "Bill Gates")

M0-Layer

# Metamodel





# Metamodel

- **Advantages of metamodels**
  - » Precise, accessible, and evolvable language definition
- **Generalization on a higher level of abstraction  
by means of the meta-metamodel**
  - » Language concepts for the definition of metamodels
  - » MOF, with Ecore as its implementation, is considered as a universally accepted meta-metamodel
- **Metamodel-agnostic tool support**
  - » Common exchange format, model repositories, model editors, model validation and transformation frameworks, etc.



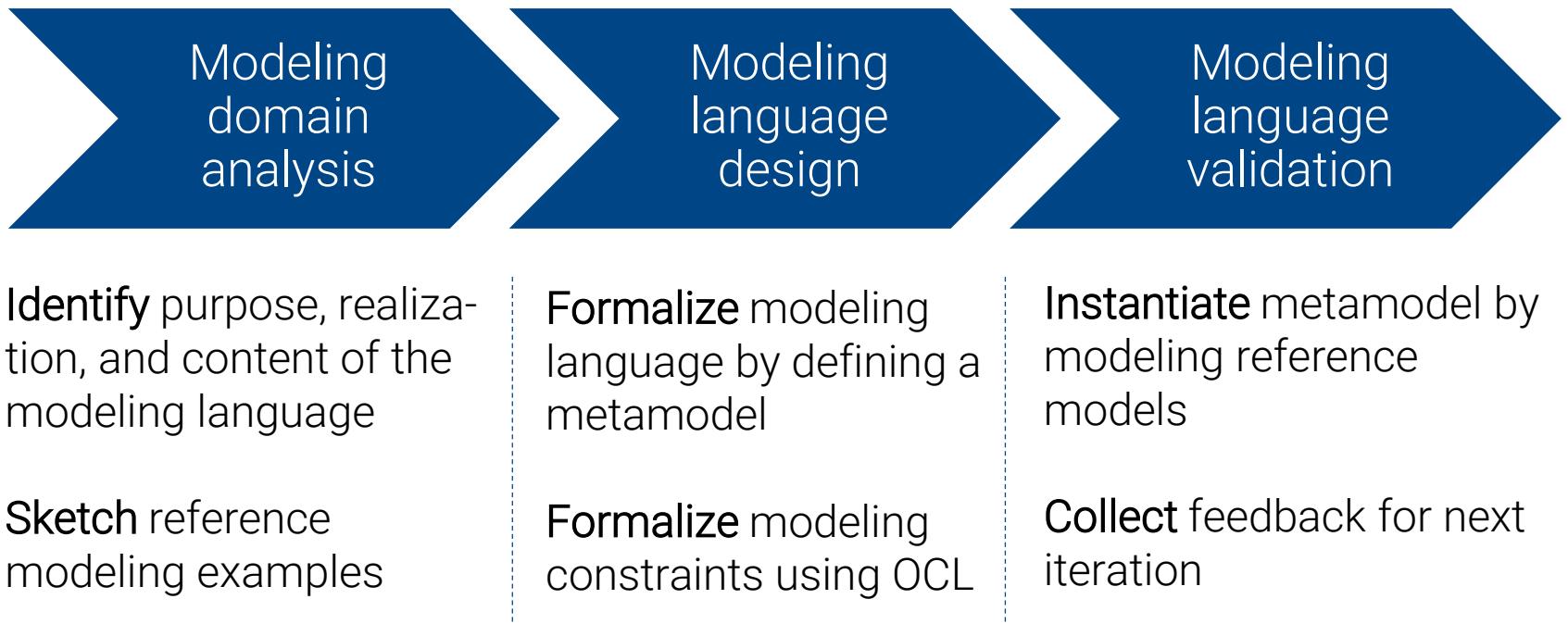
# Metamodel

4-layer Metamodeling Stack		Examples
Language Engineering	M3 <b>Meta-Metamodel</b>	MOF, Ecore
	«conformsTo» ↴	
	defines ➤ <b>Meta-Language</b>	
	↑	
Domain Engineering	M2 <b>Metamodel</b>	UML, ER, ...
	«conformsTo»	
	M1 <b>Model</b>	UniSystem, ...
	«conformsTo»	
	M0 <b>Model Instance</b>	A UniSystem Snapshot
	«conformsTo»	



# Metamodel development process

## Incremental and Iterative





# MOF - Meta Object Facility

- OMG standard for the definition of metamodels
- MOF is an object-orientated modeling language
  - » Objects are described by classes
  - » Intrinsic properties of objects are defined as attributes
  - » Extrinsic properties (links) between objects are defined as associations
  - » Packages group classes
- MOF itself is defined by MOF (reflexive) and divided into
  - » eMOF (essential MOF)
    - Simple language for the definition of metamodels
    - Target audience: metamodelers
  - » cMOF (complete MOF)
    - Extends eMOF
    - Supports management of meta-data via enhanced services (e.g. reflection)
    - Target audience: tool manufacturers



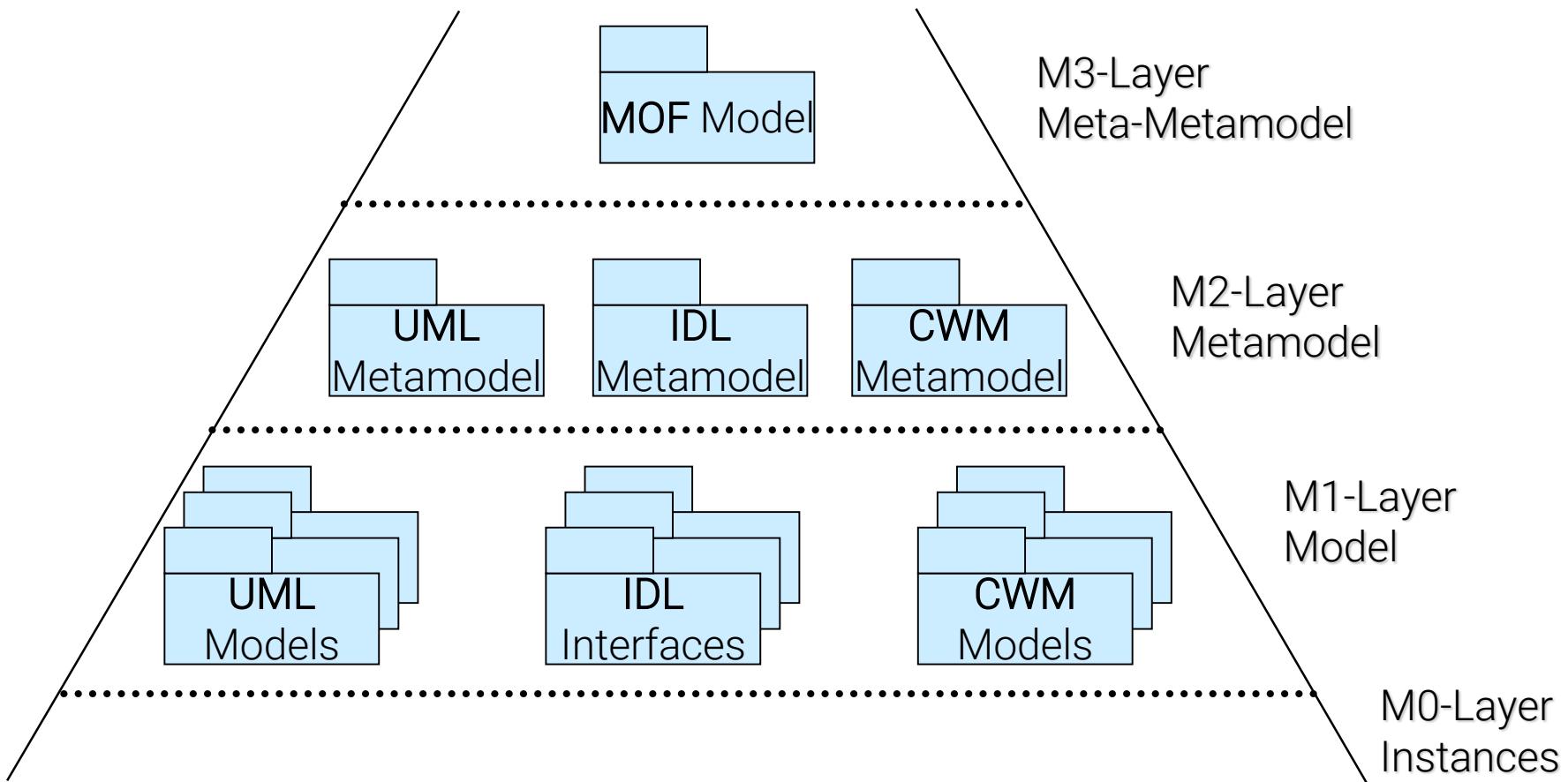
## MOF - Meta Object Facility

- Offers modeling infrastructure not only for MDA, but for MDE in general
  - » MDA dictates MOF as meta-metamodel
  - » UML, CWM and further OMG standards are conform to MOF
- Mapping rules for various technical platforms defined for MOF
  - » XML: XML Metadata Interchange (XMI)
  - » Java: Java Metadata Interfaces (JMI)
  - » CORBA: Interface Definition Language (IDL)



## MOF - Meta Object Facility

- OMG language definition stack



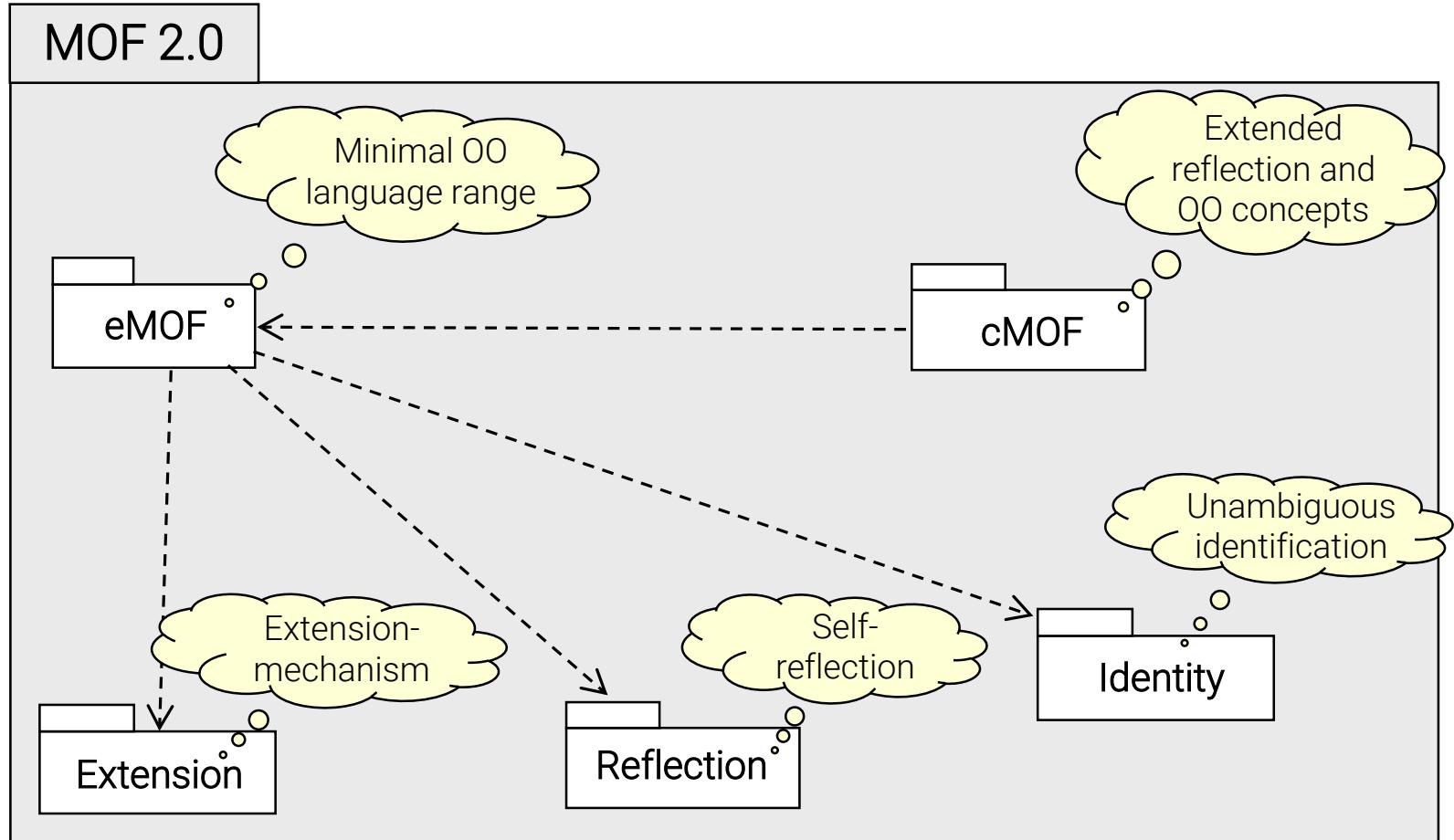


# Why an additional language for M3

- MOF only a subset of UML
  - » MOF is similar to the UML class diagram, but much more limited
  - » No n-ary associations, no association classes, ...
  - » No overlapping inheritance, interfaces, dependencies, ...
- Main differences result from the field of application
  - » UML
    - Domain: object-oriented modeling
    - Comprehensive modeling language for various software systems
    - Structural and behavioral modeling
    - Conceptual and implementation modeling
  - » MOF
    - Domain: metamodeling
    - Simple conceptual structural modeling language
- Conclusion
  - » MOF is a highly specialized DSML for metamodeling
  - » Core of UML and MOF (almost) identical

# MOF – Meta Object Facility

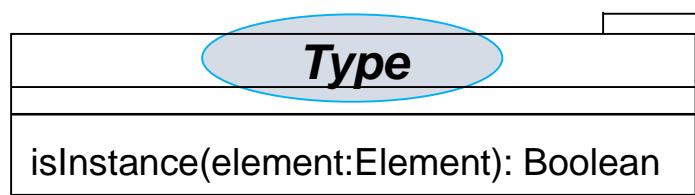
- Language architecture of MOF 2.0



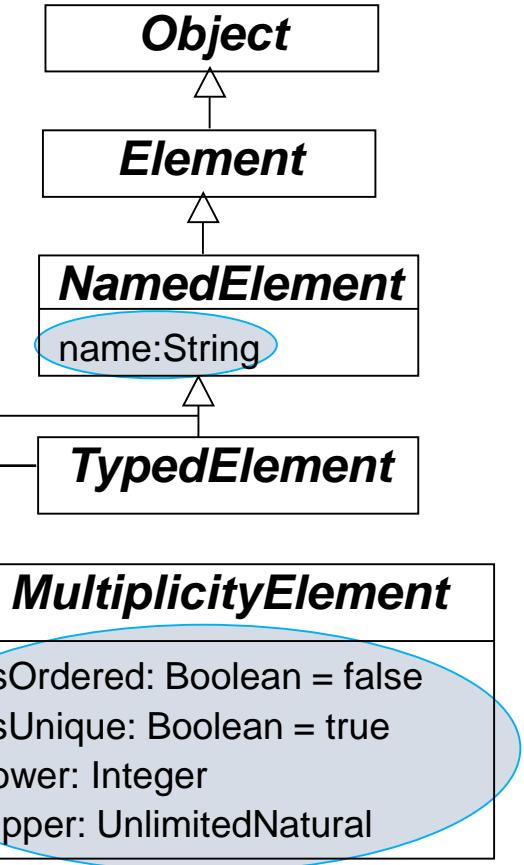


# MOF – Meta Object Facility

- Abstract classes of eMOF
- Definition of general properties
  - » NamedElement
  - » TypedElement
  - » MultiplicityElement
    - Set/Sequence/OrderedSet/Bag
    - Multiplicities



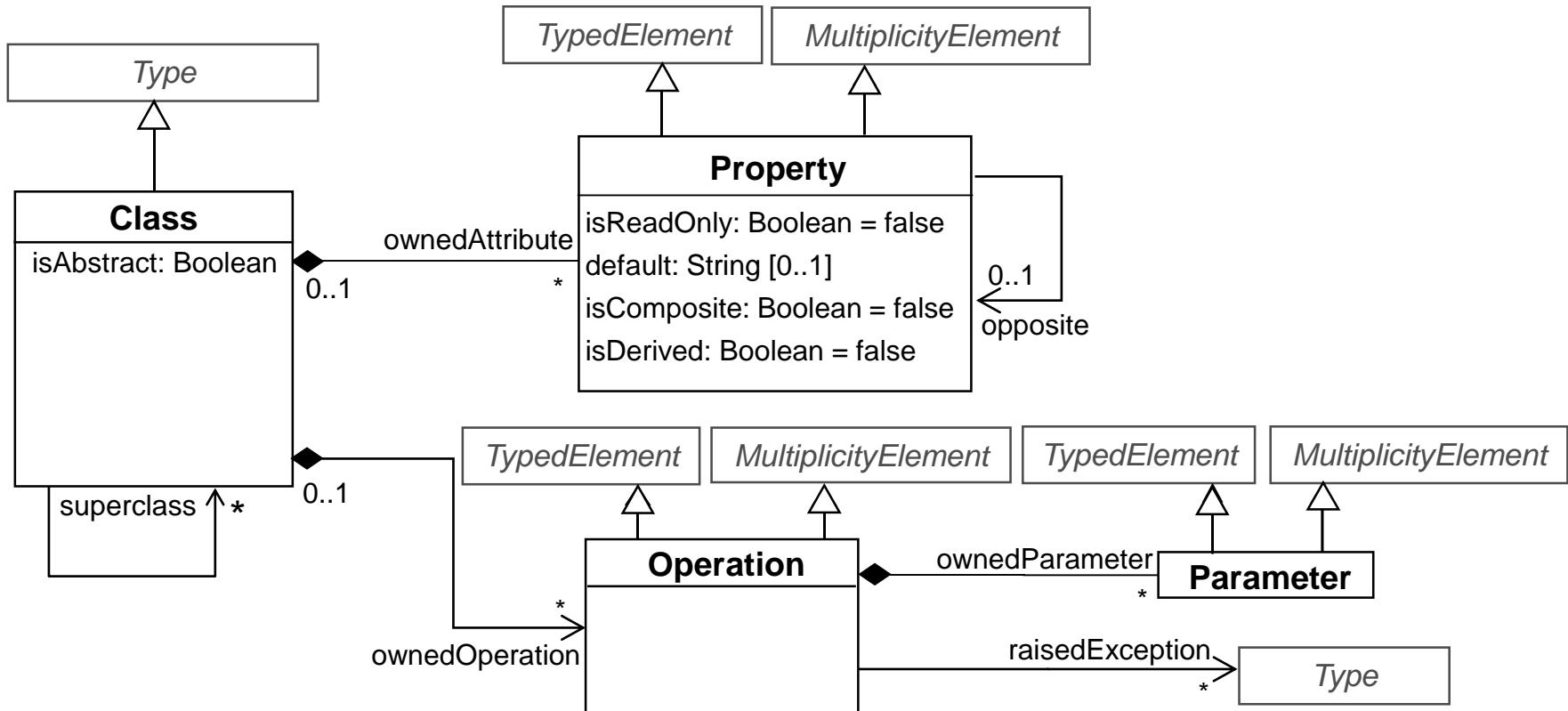
## Taxonomy of abstract classes





# MOF – Meta Object Facility

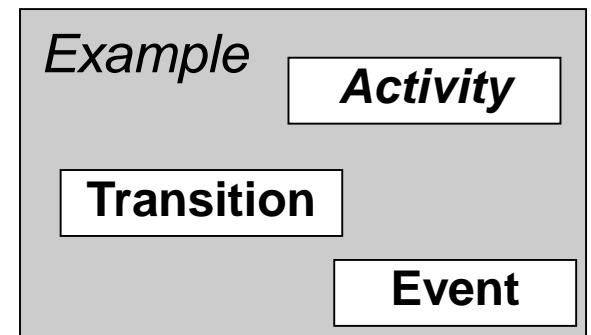
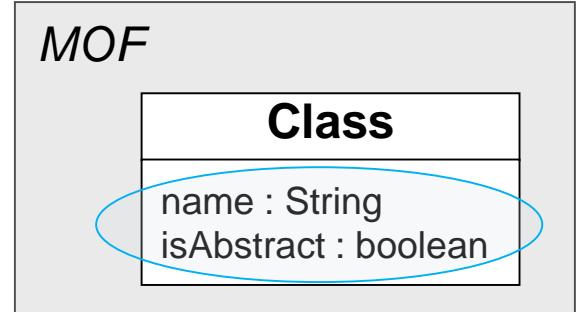
- Core of eMOF
  - Based on object-orientation
  - Classes, properties, operations, and parameters





# MOF – Meta Object Facility

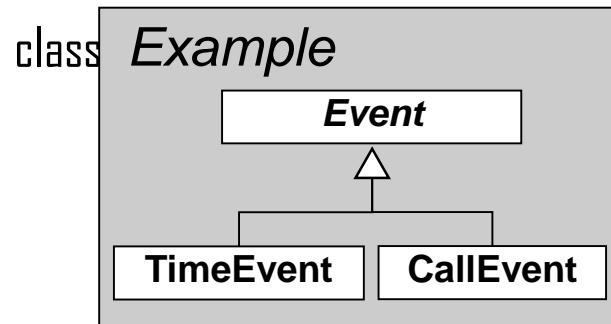
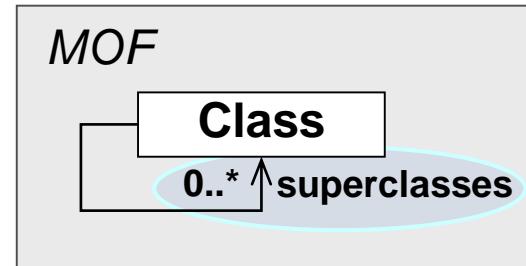
- A class specifies structure and behavior of a set of objects
  - » Intentional definition
  - » An unlimited number of instances (objects) of a class may be created
- A class has an unique name in its namespace
- Abstract classes cannot be instantiated!
  - » Only useful in inheritance hierarchies
  - » Used for »highlighting« of common features of a set of subclasses
- Concrete classes can be instantiated!





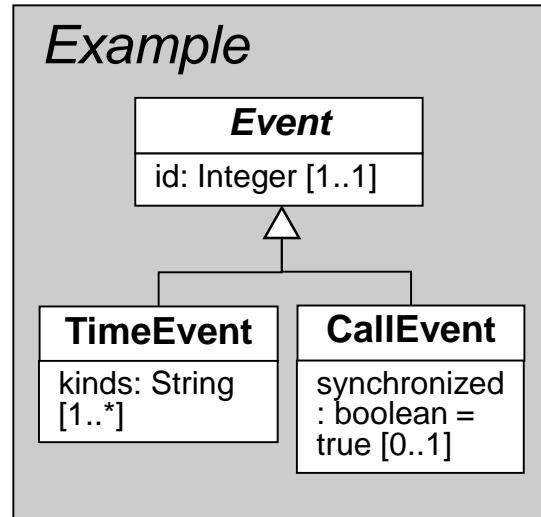
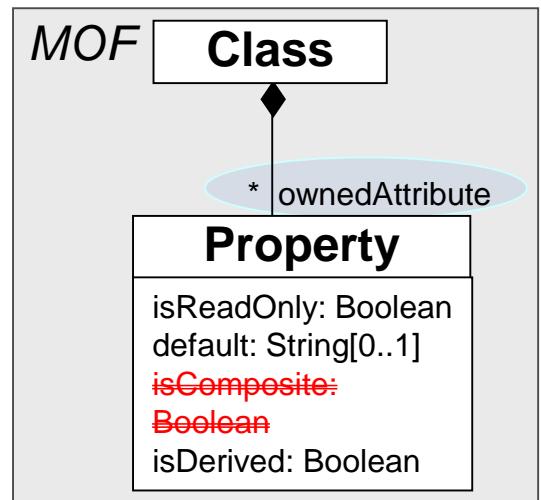
# MOF – Meta Object Facility

- Generalization: relationship between
  - » a specialized class (subclass) and
  - » a general class (superclass)
- Subclasses inherit properties of their super- properties
- Discriminator: „virtual“ attribute used for the classification
- Disjoint (non-overlapping) generalization
- Multiple inheritance



# MOF – Meta Object Facility

- Attributes describe inherent characteristics of classes
- Consist of a name and a type (obligatory)
- Multiplicity: how many values can be stored in an attribute slot (obligatory)
  - » Interval: upper and lower limit are natural numbers
  - » \* asterisk - also possible for upper limit (Semantics: unlimited number)
  - » 0..x means optional: null values are allowed
- Optional
  - » Default value
  - » Derived (calculated) attributes
  - » Changeable: isReadOnly = false
  - » isComposite is always true for attributes





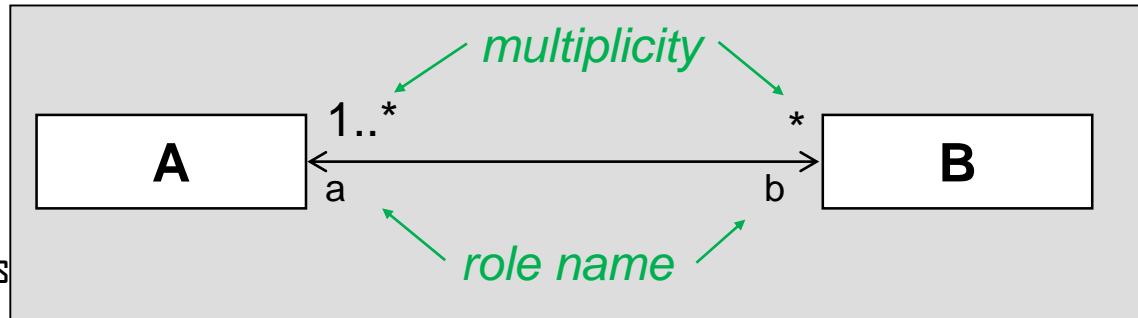
## MOF – Meta Object Facility

- An association describes the common structure of a set of relationships between objects
- MOF only allows unary and binary associations, i.e., defined between two classes
- Binary associations consist of two roles whereas each role has
  - » Role name
  - » Multiplicity limits the number of partner objects of an object
- Composition
  - » „part-whole“ relationship (also „part-of“ relationship)
  - » One part can be at most part of one composed object at one time
  - » Asymmetric and transitive
  - » Impact on Multiplicity: 1 or 0..1

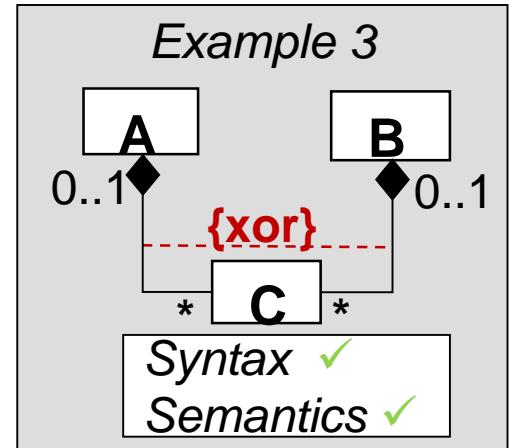
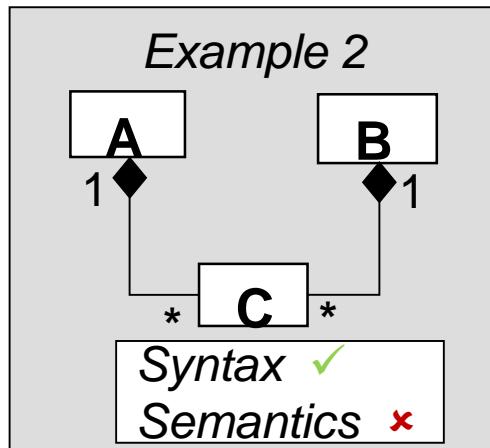
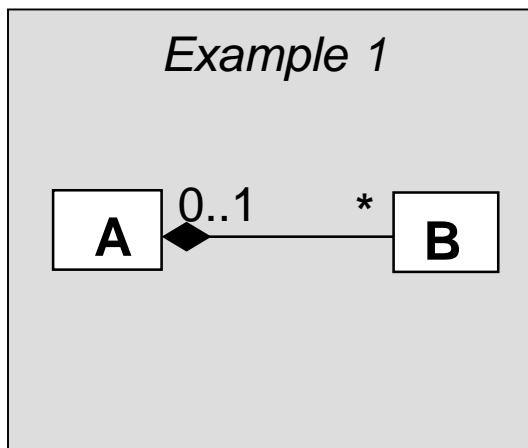


# MOF – Meta Object Facility

- Association

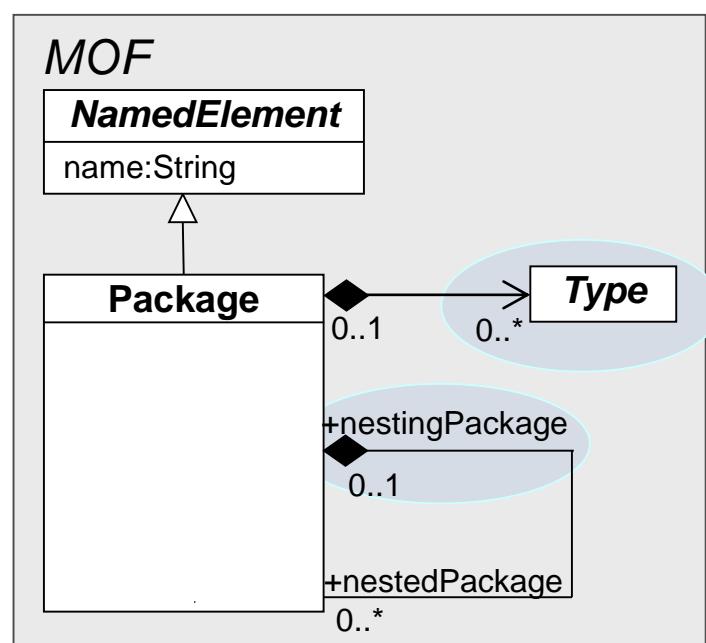


- Compos

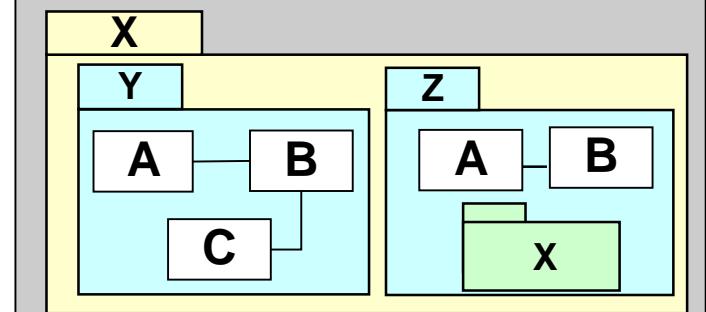


# MOF – Meta Object Facility

- Packages serve as a grouping mechanism
  - » Grouping of related types, i.e., classes, enumerations, and primitive types.
- Partitioning criteria
  - » Functional or information cohesion
- Packages form own namespace
  - » Usage of identical names in different parts of a metamodel
- Packages may be nested
  - » Hierarchical grouping
- Model elements are contained by one package



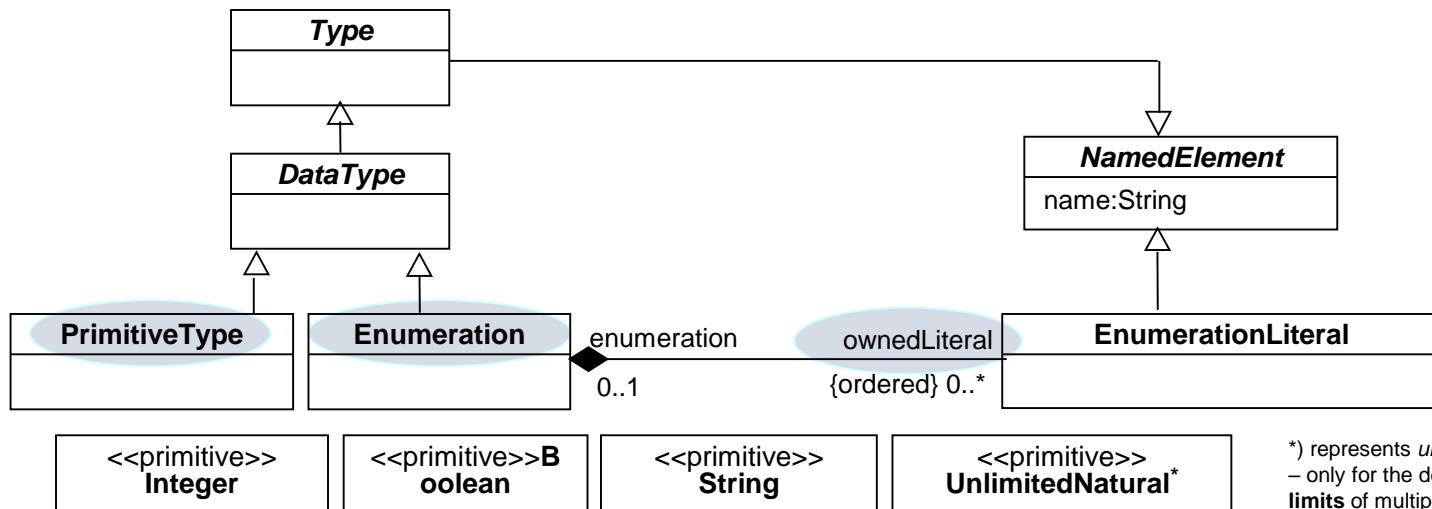
## Example





# MOF – Meta Object Facility

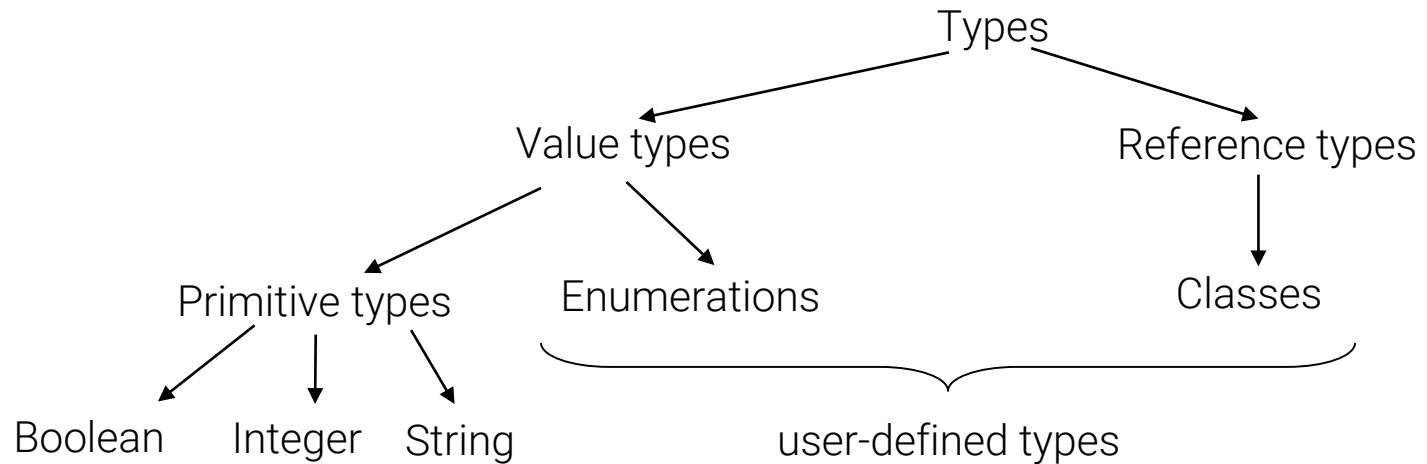
- Primitive data types: Predefined types for integers, character strings and Boolean values
- Enumerations: Enumeration types consisting of named constants
  - » Allowed values are defined in the course of the declaration
    - Example: enum Color {red, blue, green}
  - » Enumeration types can be used as data types for attributes



\*) represents *unlimited number (asterisk)*  
– only for the definition of the **upper limits** of multiplicities

# MOF – Meta Object Facility

- Differentiation between value types and reference types
  - » Value types: contain a direct value (e.g., 123 or 'x')
  - » Reference types: contain a reference to an object

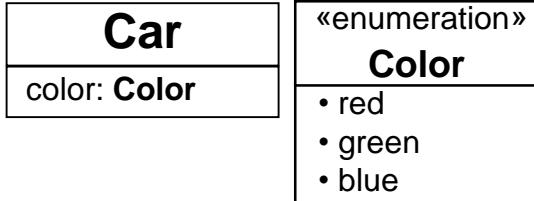


- Examples

Primitive types



Enumerations

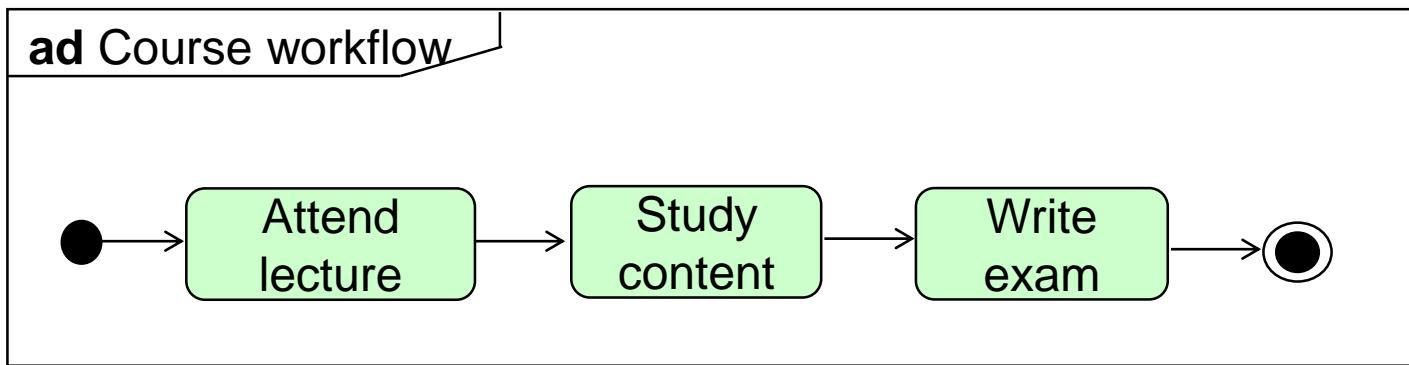


Reference types



## Example 1/9

- Activity diagram example
  - » Concepts: Activity, Transition, InitialNode, FinalNode
  - » Domain: Sequential linear processes



- Question: How does a possible metamodel to this language look like?
- Answer: apply metamodel development process!



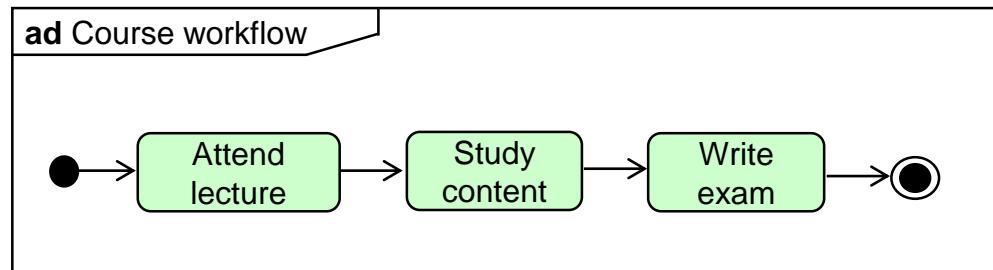
## Example 2/9

### Identification of the modeling concepts

Notation table

Syntax	Concept
	ActivityDiagram
	FinalNode
	InitialNode
	Activity
	Transition

Example model = Reference Model





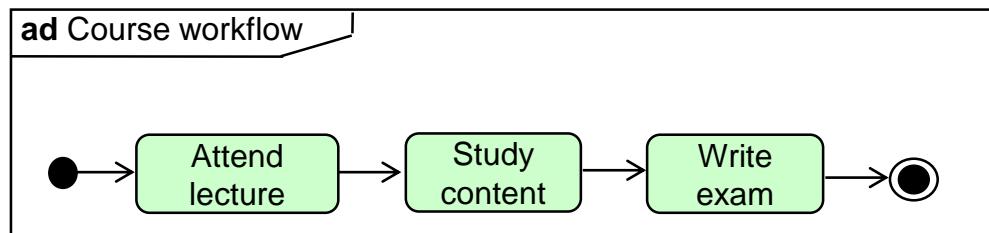
## Example 3/9

### Determining the properties of the modeling concepts

#### Modeling concept table

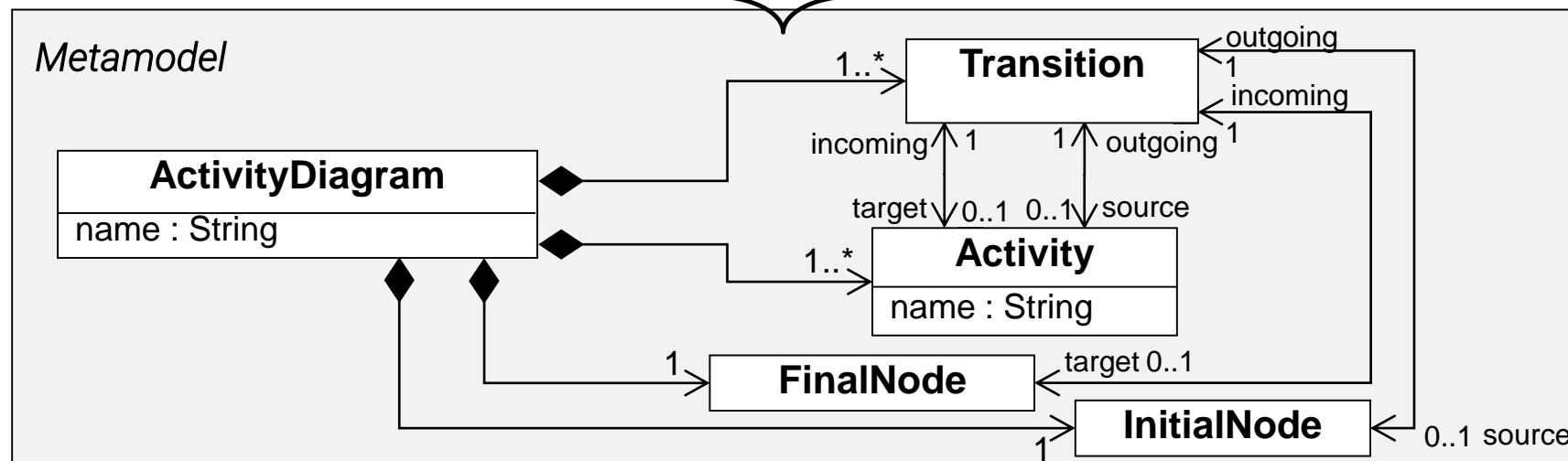
Concept	Intrinsic properties	Extrinsic properties
ActivityDiagram	Name	1 <i>InitialNode</i> 1 <i>FinalNode</i> Unlimited number of <i>Activities</i> and <i>Transitions</i>
FinalNode	-	Incoming <i>Transitions</i>
InitialNode	-	Outgoing <i>Transitions</i>
Activity	Name	Incoming and outgoing <i>Transitions</i>
Transition	-	Source node and target node Nodes: <i>InitialNode</i> , <i>FinalNode</i> , <i>Activity</i>

#### Example model



## Example 4/9 Object-oriented design of the language

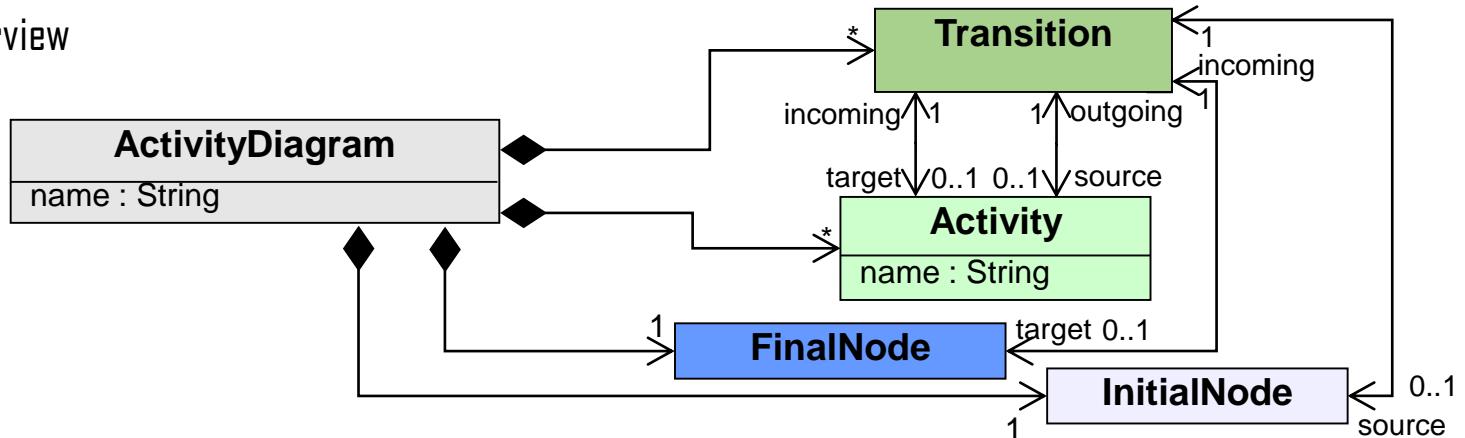
MOF	Class	Attribute	Association
Concept	Intrinsic properties	Extrinsic properties	
ActivityDiagram	Name	1 InitialNode 1 FinalNode Unlimited number of Activities and Transitions	
FinalNode	-	Incoming Transition	
InitialNode	-	Outgoing Transition	
Activity	Name	Incoming and outgoing Transition	
Transition	-	Source node and target node Nodes: InitialNode, FinalNode, Activity	



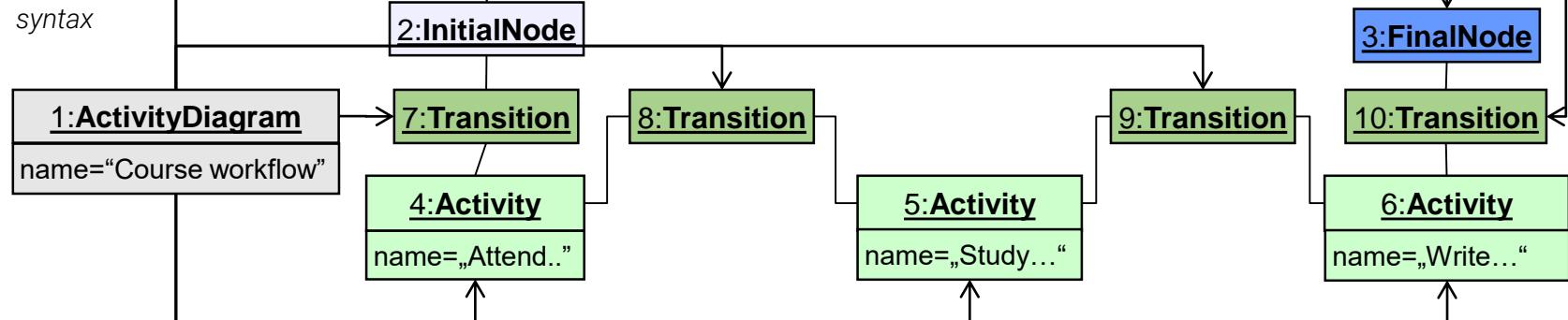
## Example 5/9

- Overview

Metamodel

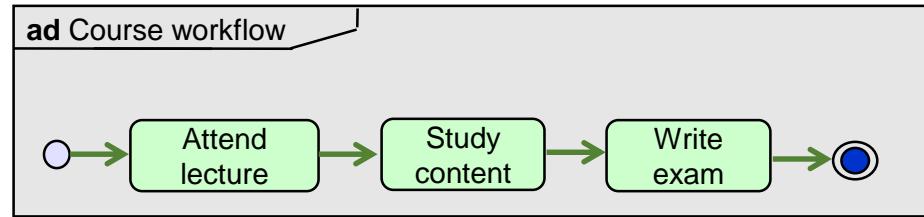


Abstract syntax



Model

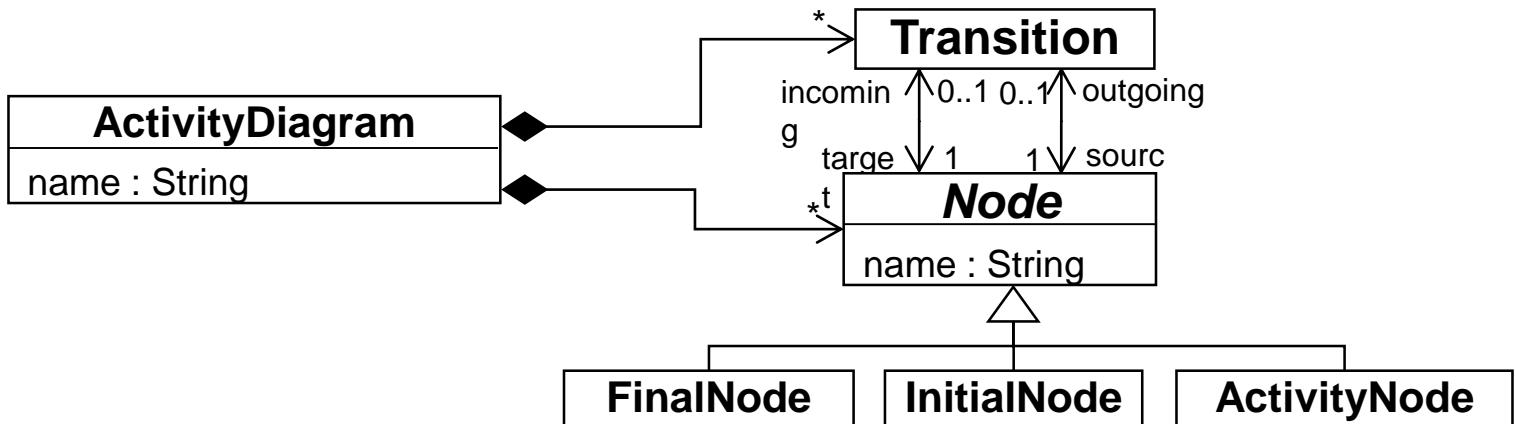
Concrete syntax



## Example 6/9

### Applying refactorings to metamodels

Metamodel



OCL Constraints

```

context ActivityDiagram
inv: self.transitions -> exists(t|t.isTypeOf(FinalNode))
inv: self.transitions -> exists(t|t.isTypeOf(InitialNode))

context FinalNode
inv: self.outgoing.isOclUndefined()

context InitialNode
inv: self.incoming.isOclUndefined()

context ActivityDiagram
inv: self.name <> '' and self.name <> OclUndefined ...
  
```

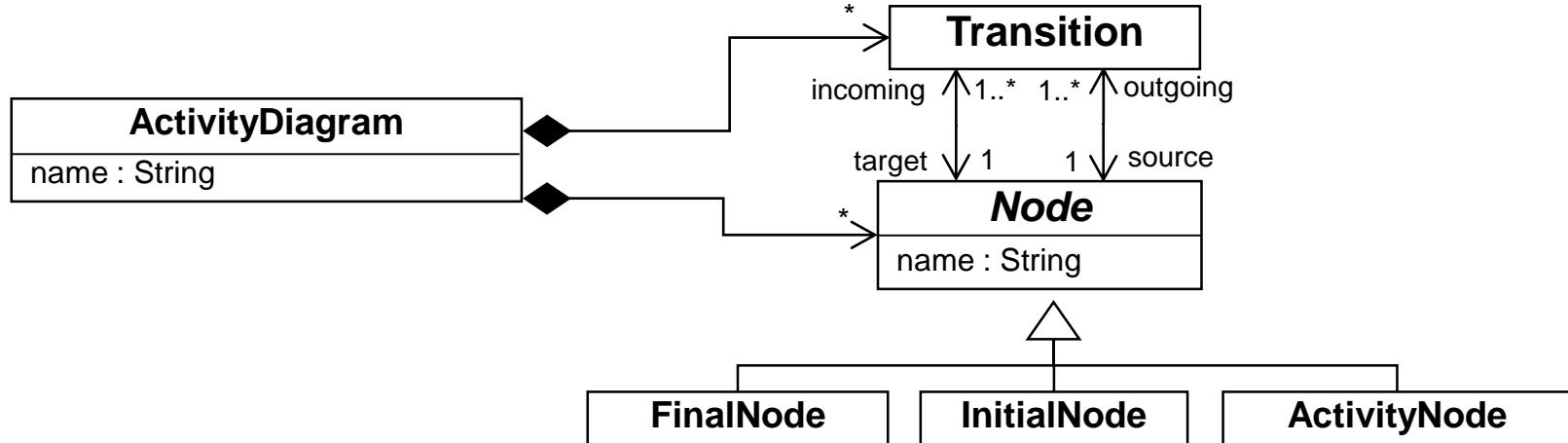
## Example 7/9

### Impact on existing models

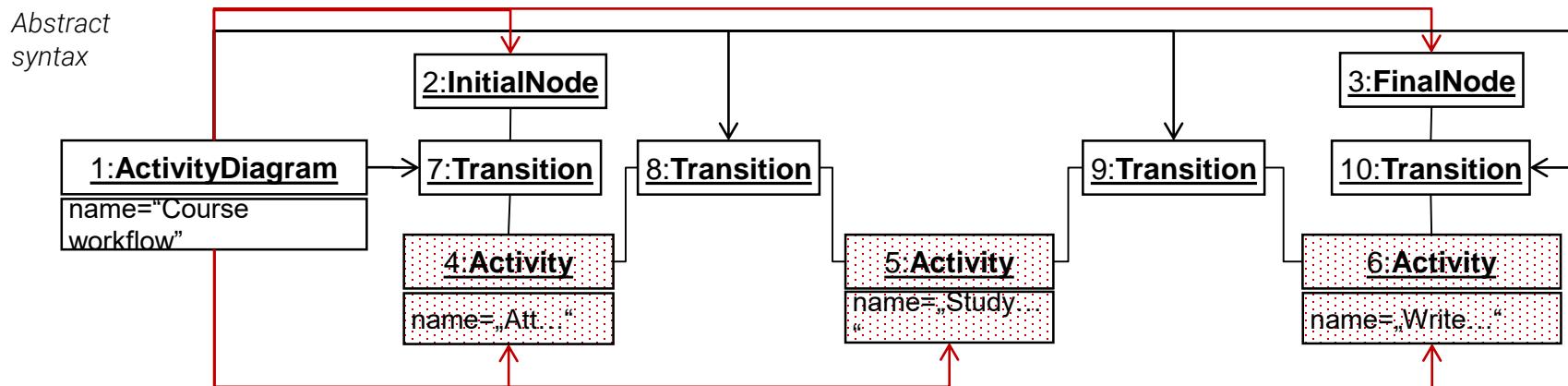
Changes:

- Deletion of class Activity
- Addition of class ActivityNode
- Deletion of redundant references

Metamodel



Model



Validation errors:

- ✗ Class Activity is unknown,
- ✗ Reference finalNode, initialNode, activity are unknown

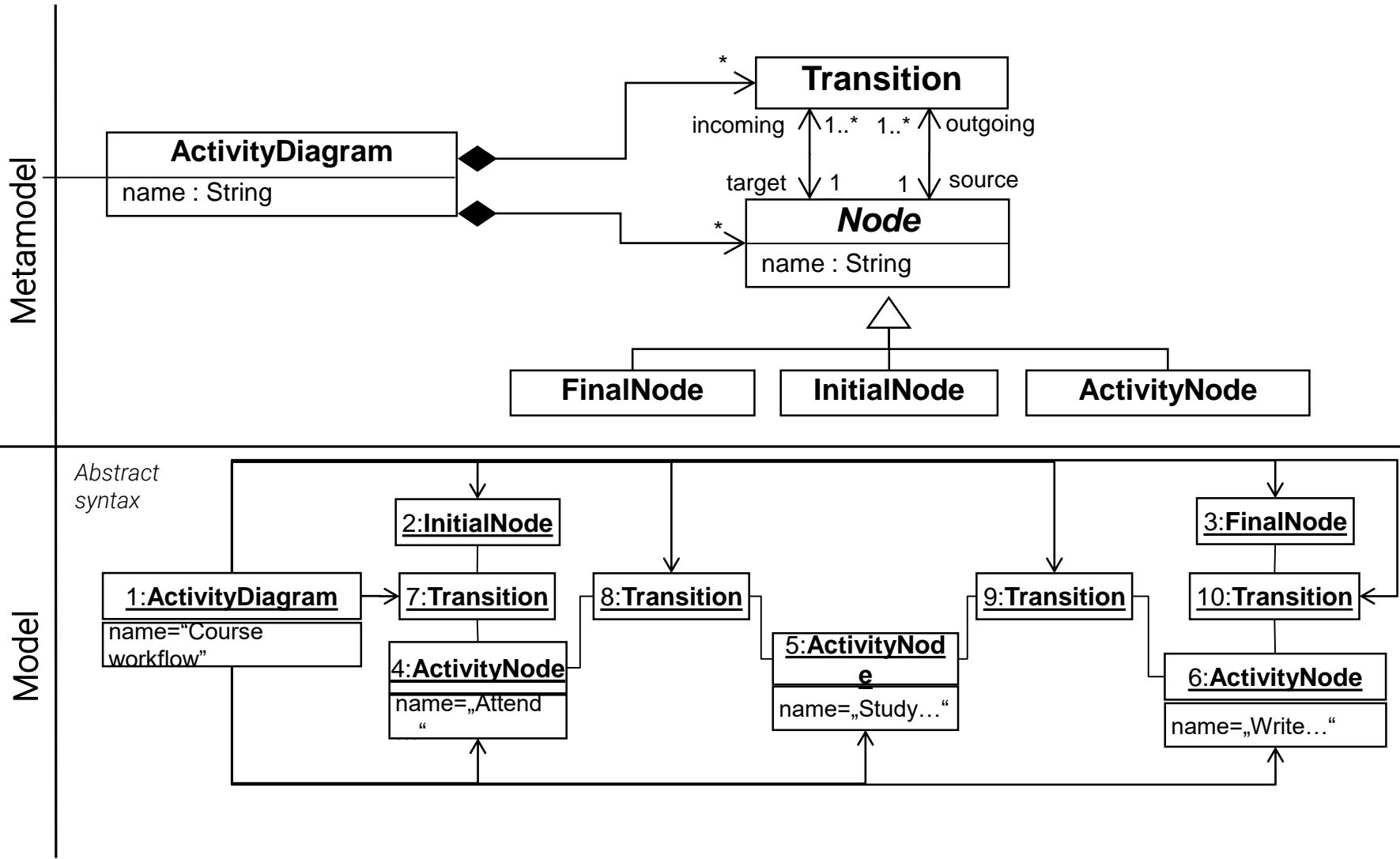


## Example 8/9

- **Model/metamodel co-evolution problem**
  - » Metamodel is changed
  - » Models already exist and may become invalid
- **Changes may break conformance relationships**
  - » Deletion and renamings of metamodel elements
- **Solution: Co-evolution rules for models coupled to metamodel changes**
  - » Example 1: Cast all Activity elements to ActivityNode elements
  - » Example 2: Cast all initialNode, finalNode, and activity links to node links

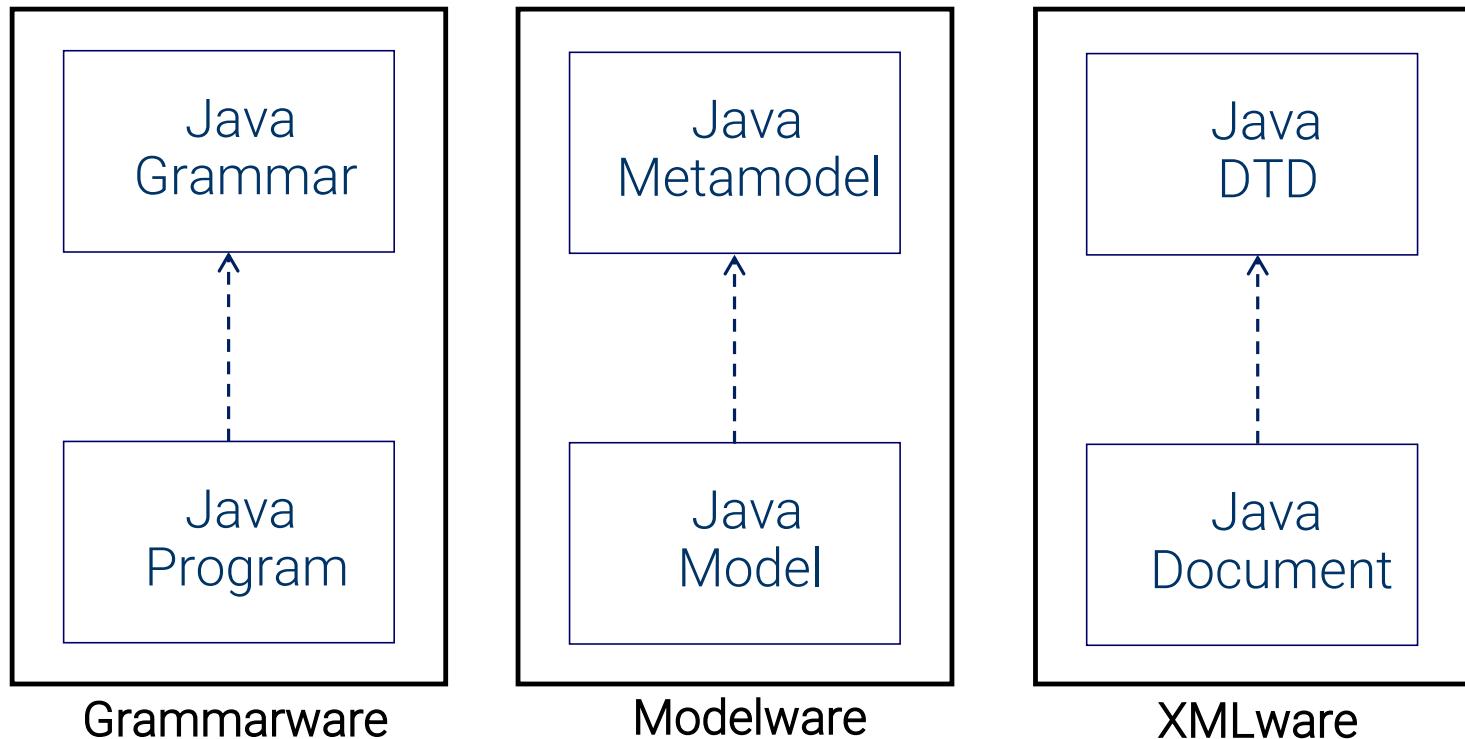
# Example 9/9

## Adapted model for new metamodel version



## Excursus: Metamodeling – everything new? I/3

- A language may be defined by meta-languages from different Technical Spaces (TS)
- Attention: Each TS has its (dis)advantages!



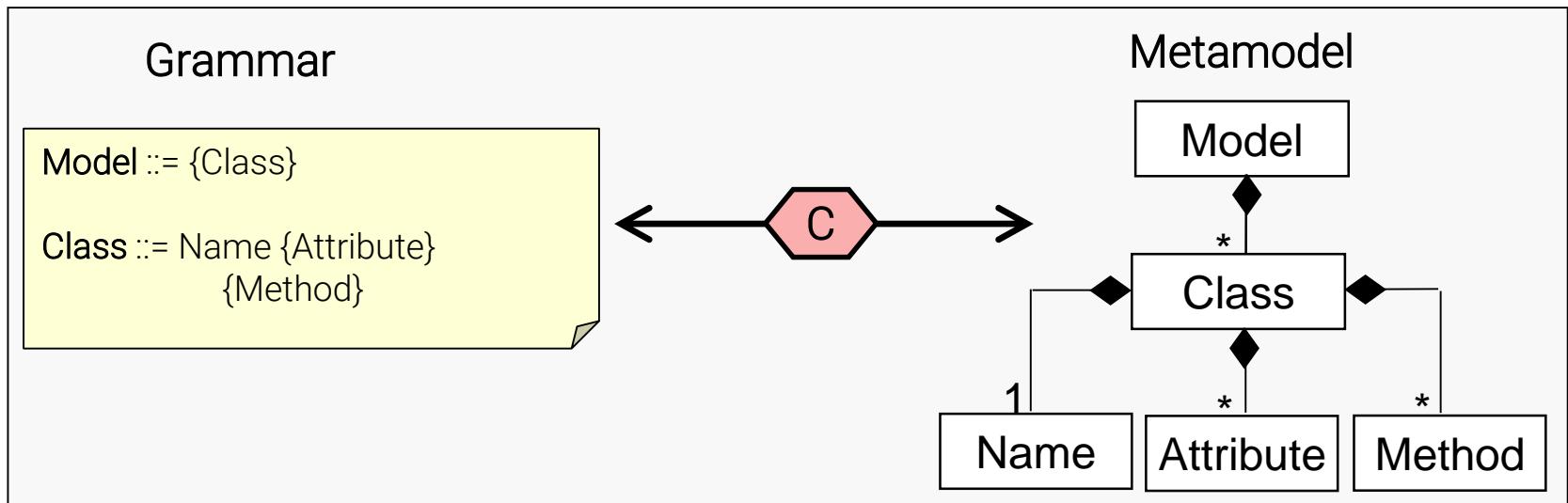
# Excursus: Metamodeling – everything new? 2/3

## Correspondence between EBNF and MOF

- Mapping table (excerpt)

EBNF	MOF
Production	Composition
Non-Terminal	Class
Sequence	Multiplicity: 0..*

- Example



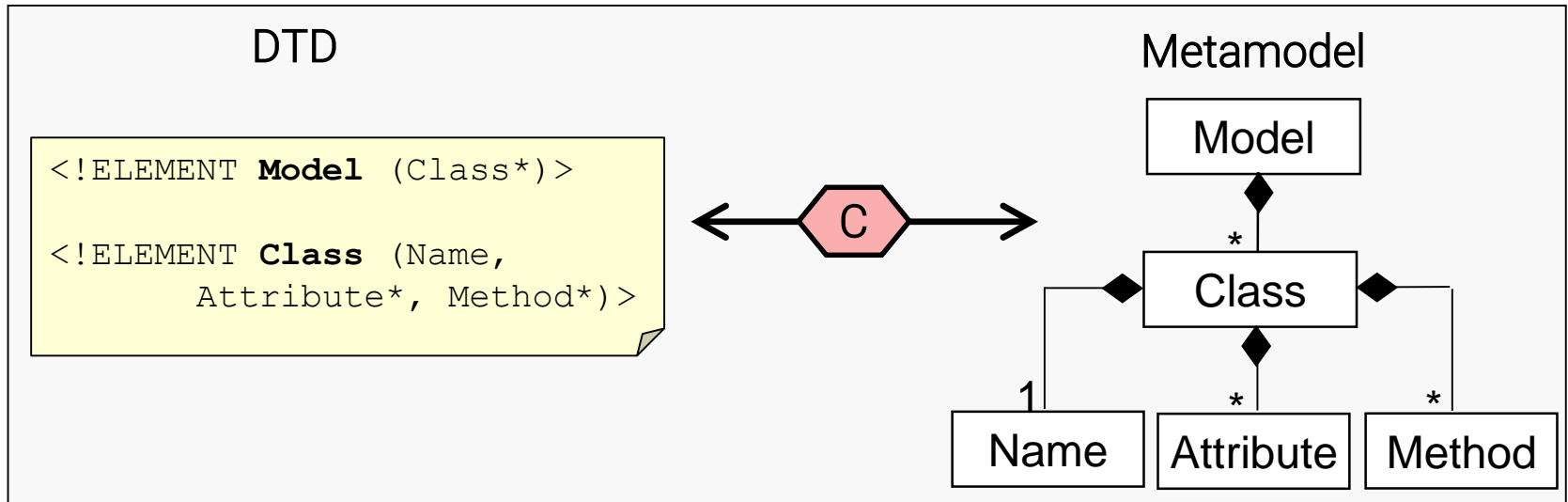
# Excursus: Metamodeling – everything new? 3/3

## Correspondence between DTD and MOF

- Mapping table (excerpt)

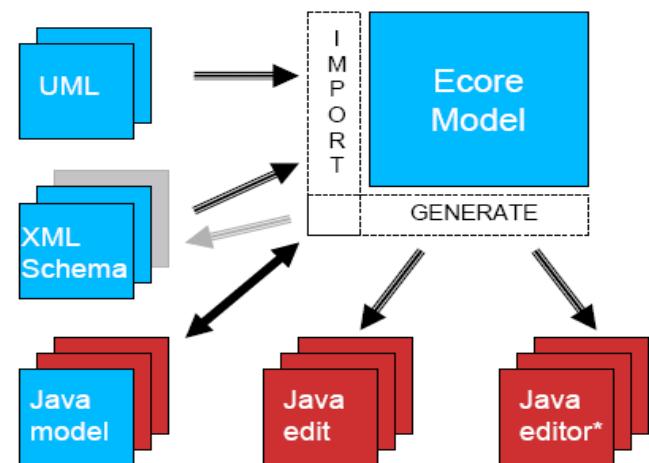
<i>DTD</i>	<i>MOF</i>
Item	Composition
Element	Class
Cardinality *	Multiplicity 0..*

- Example



# Ecore

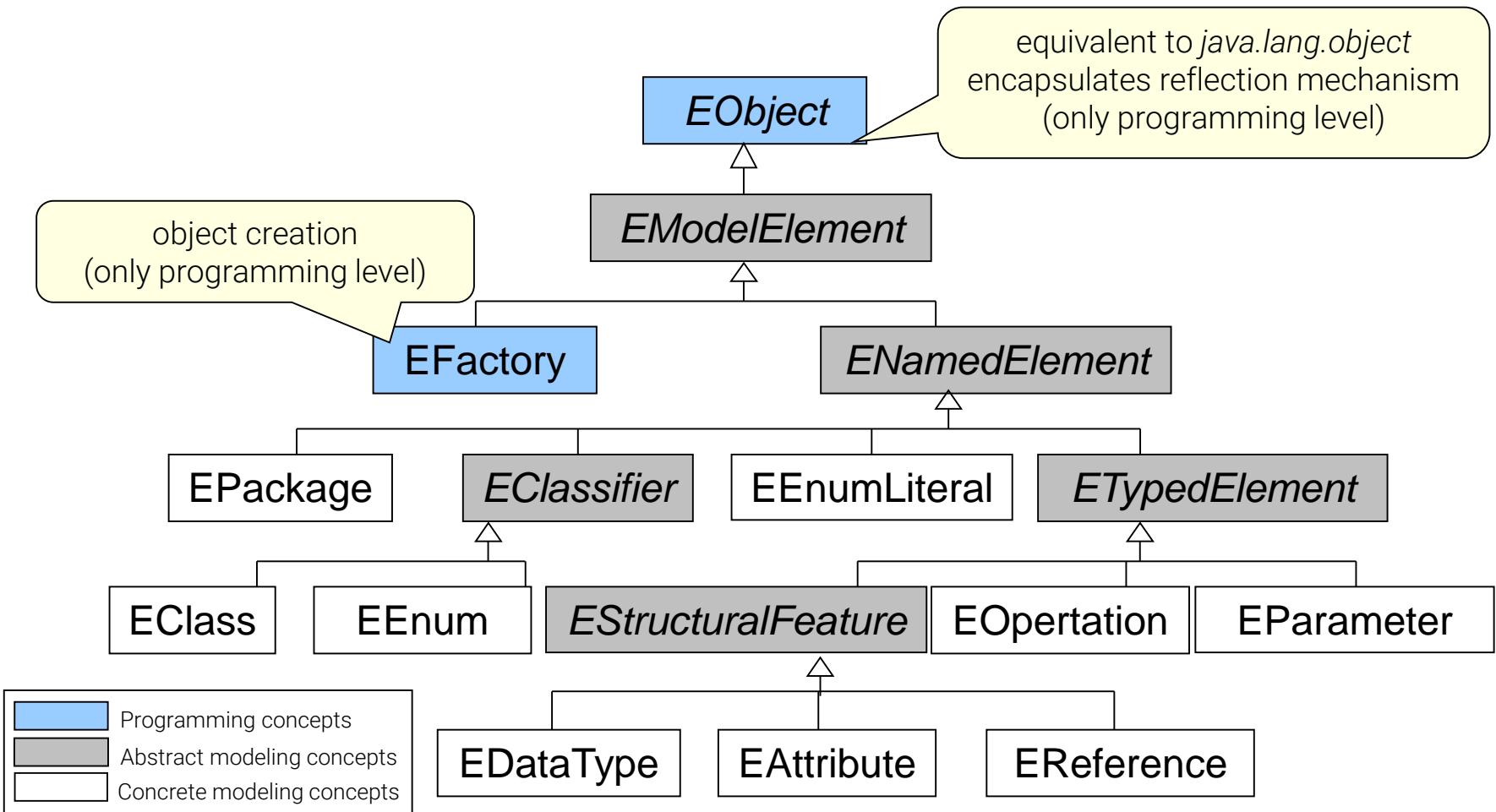
- Ecore is the meta-metamodel of the Eclipse Modeling Frameworks (EMF)
  - » [www.eclipse.org/emf](http://www.eclipse.org/emf)
- Ecore is a Java-based implementation of eMOF
- Aims of Ecore
  - » Mapping eMOF to Java
- Aims of EMF
  - » Definition of modeling languages
  - » Generation of model editors
  - » UML/Java/XML integration framework





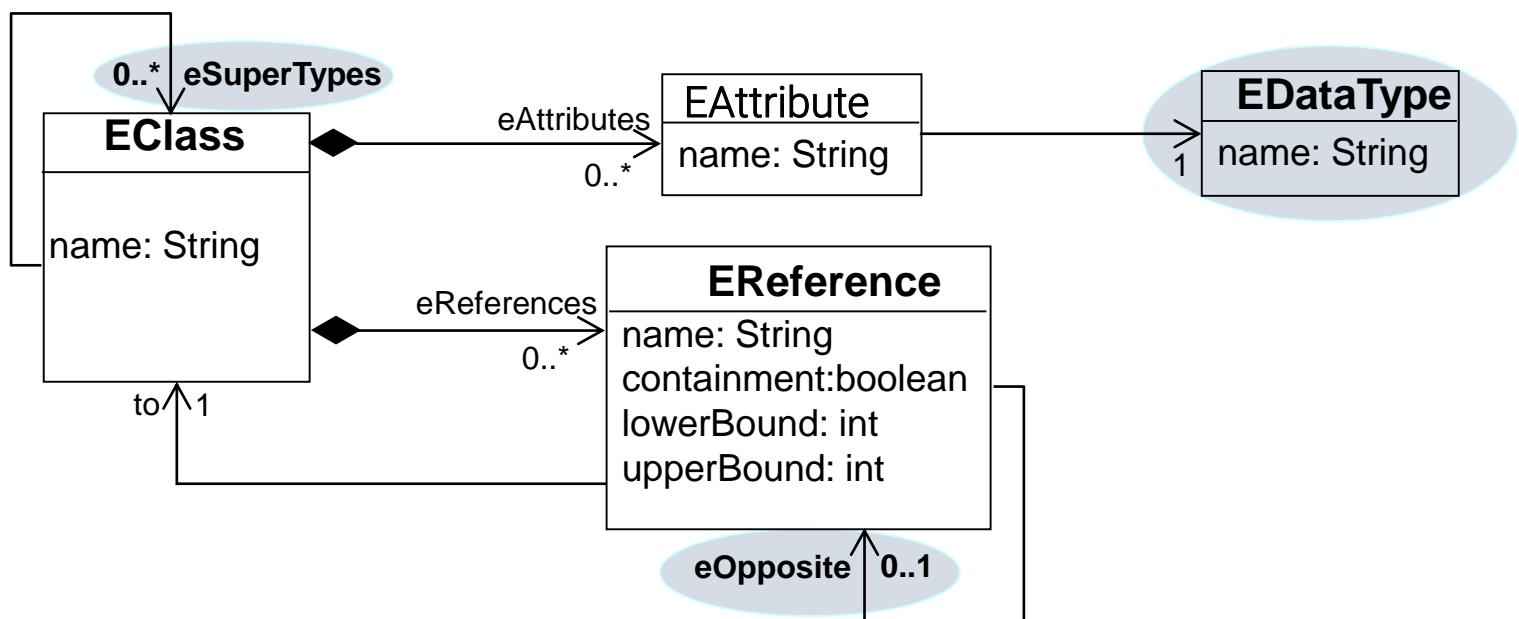
## Ecore

- Taxonomy of the language concepts



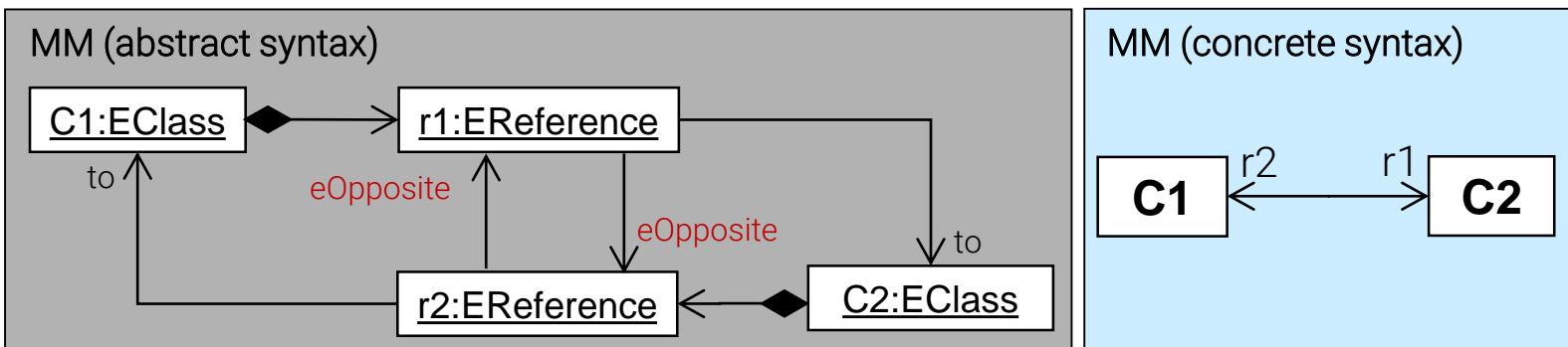
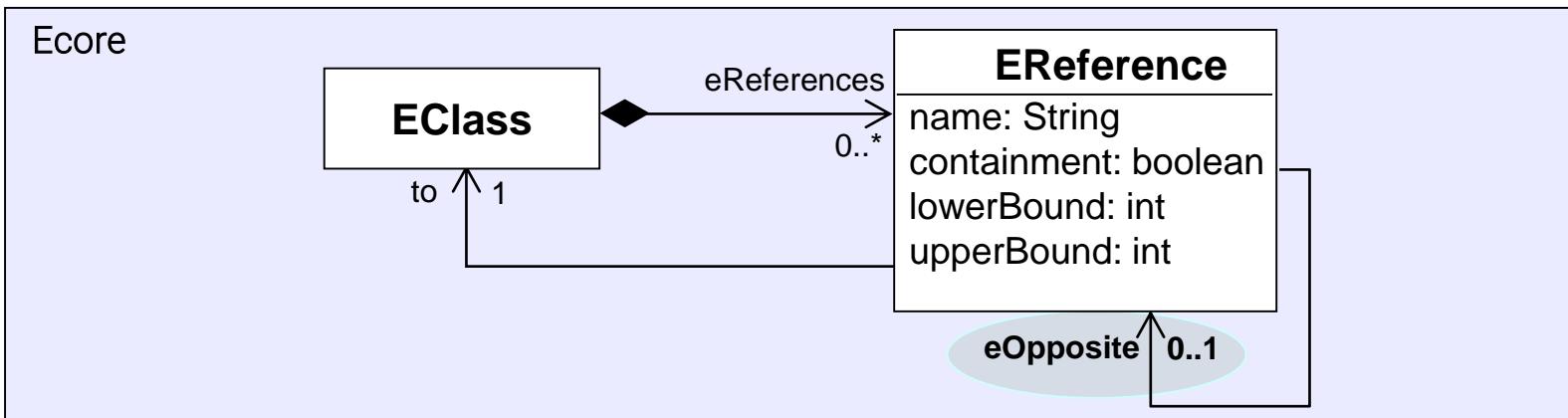
# Ecore

- Based on object-orientation (as eMDF)
  - » Classes, references, attributes, inheritance, ...
  - » Binary associations are represented as two references
  - » Data types are based on Java data types
  - » Multiple inheritance is resolved by one „real“ inheritance and multiple implementation inheritance relationships



# Ecore

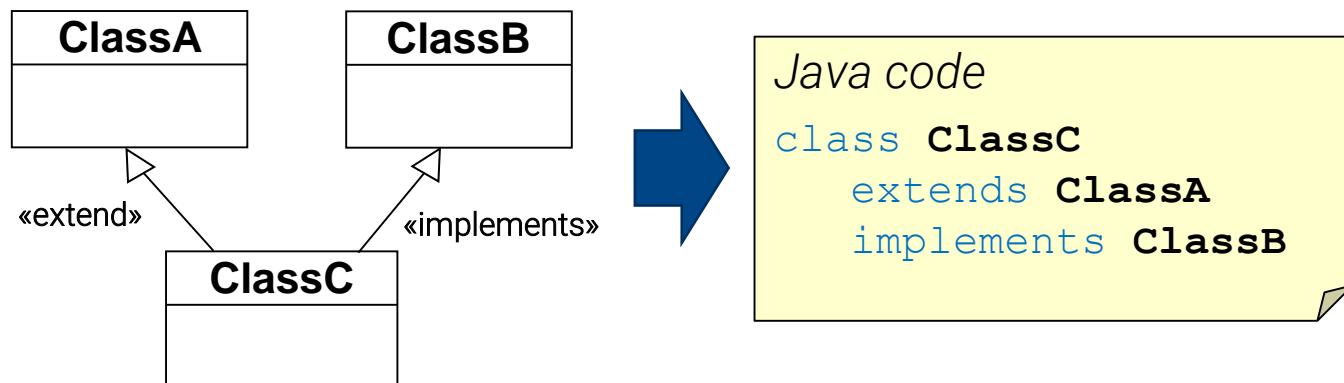
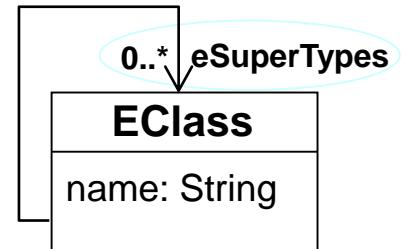
- A binary association demands for two references
  - » One per association end
  - » Both define the respective other one as eOpposite





# Ecore

- Ecore supports multiple inheritance
  - » Unlimited number of eSuperTypes
- Java supports only single inheritance
  - » Multiple inheritance simulated by implementation of interfaces!
- Solution for Ecore2Java mapping
  - » First inheritance relationship is used as „real“ inheritance relationship using «extend»
  - » All other inheritances are interpreted as specification inheritance «implements»





## Ecore - Data types

- List of Ecore data types (excerpt)
  - » Java-based data types
  - » Extendable through self-defined data types
    - Have to be implemented by Java classes

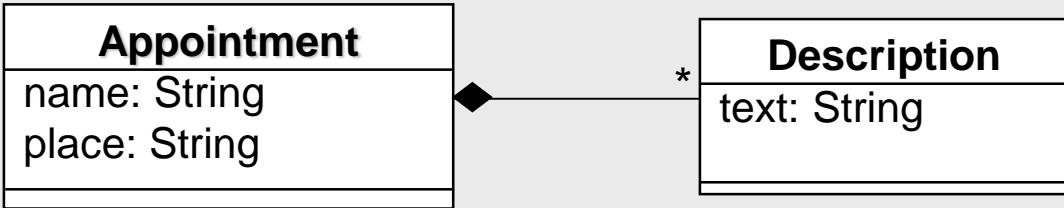
Ecore data type	Primitive type or class (Java)
EBoolean	boolean
EChar	char
EFloat	float
EString	java.lang.String
EBooleanObject	java.lang.Boolean
...	...



# Ecore

## Class diagram – Model TS

- Concrete syntax for Ecore models



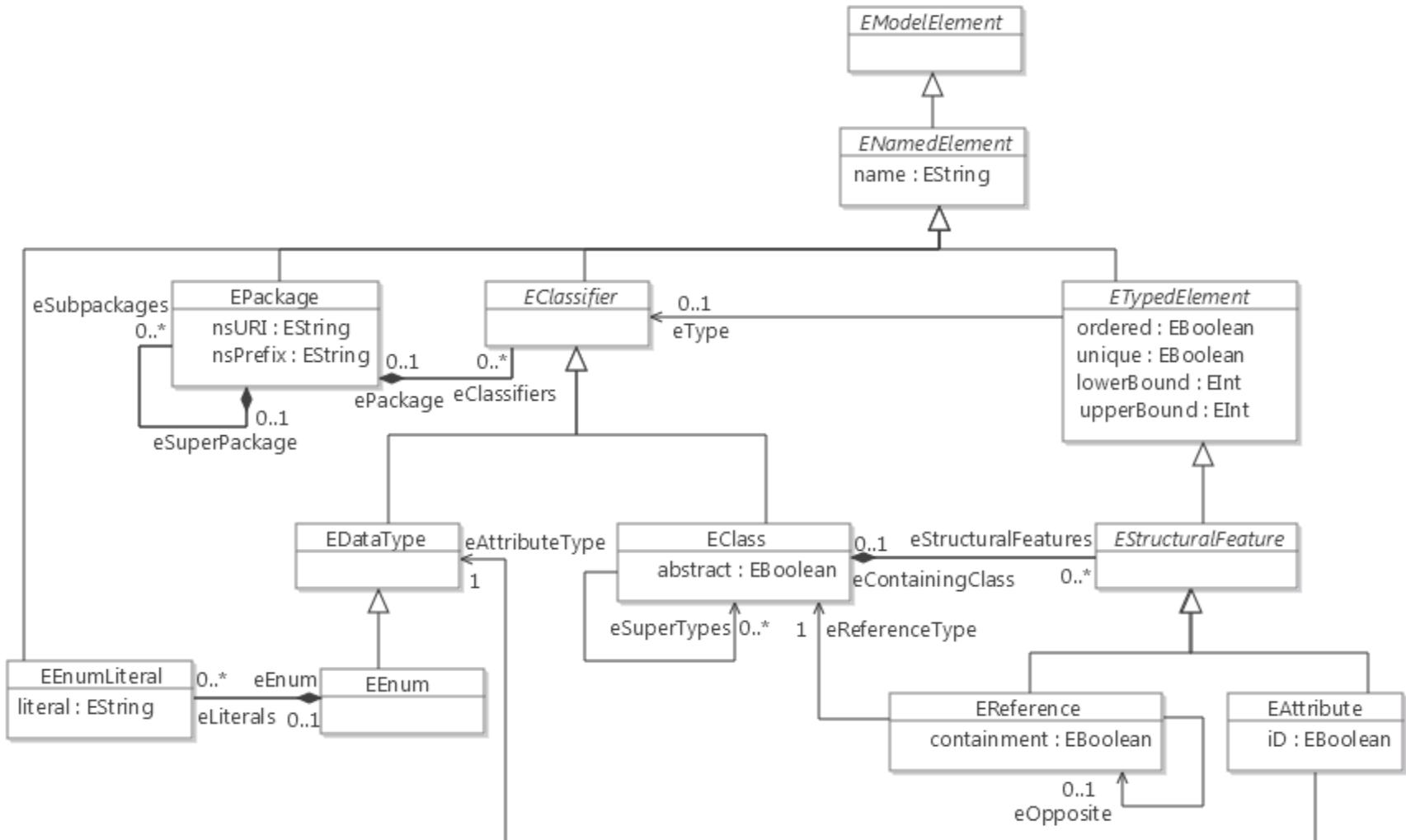
- Java Annotation

```
public interface Appointment{
    /* @model type="Description" containment="true" */
    List getDescription();
}
```

ANSWER

```
<xsd:complexType name="Appointment">
    <xsd:element name="description" type="Description"
        minOccurs="0" maxOccurs="unbounded" />
</xsd:complexType>
```

# Summary





# Eclipse Modeling Framework

## What is EMF?

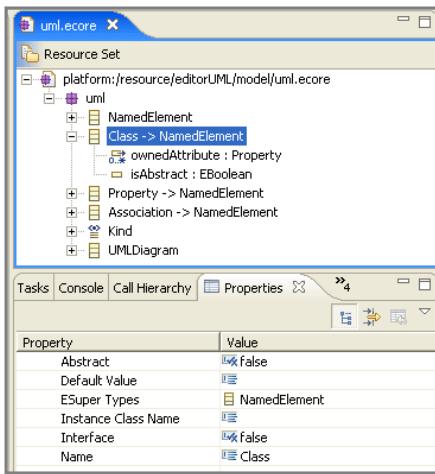
- Pragmatic approach to combine modeling and programming
  - » Straight-forward mapping rules between Ecore and Java
- EMF facilitates automatic generation of different implementations out of Ecore models
  - » Java code, XML documents, XML Schemata
- Multitude of Eclipse projects are based on EMF
  - » Graphical Editing Framework (GEF)
  - » Graphical Modeling Framework (GMF)
  - » Model to Model Transformation (M2M)
  - » Model to Text Transformation (M2T)
  - » ...

# Eclipse Modeling Framework

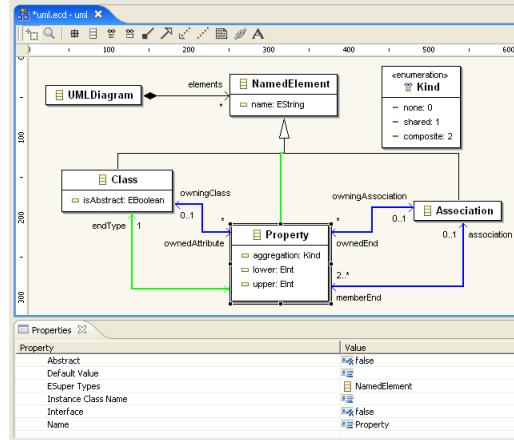
## Metamodeling Editors

- Creation of metamodels via
  - » Tree-based editors (abstract syntax)
    - Included in EMF
  - » UML-based editors (graphical concrete syntax)
    - e.g., included in Graphical Modeling Framework
  - » Text-based editors (textual concrete syntax)
    - e.g., KM3 and EMFatic
- All types allow for a semantically equivalent metamodeling

Tree-based editor



UML-based editor



Text-based editor

The screenshot shows the Eclipse Text-based editor interface. It displays a text-based representation of a metamodel in KM3 format. The code defines several classes: 'NamedElement', 'Schema', 'Table', and 'Column'. The 'NamedElement' class is abstract, with attributes for 'name' and 'Kind'. The 'Schema' class extends 'NamedElement' and has a reference to 'tables[]'. The 'Table' class extends 'NamedElement' and has attributes for 'columns[\*]' and an 'ordered container'. The 'Column' class extends 'NamedElement' and has an attribute for 'datatype'. There are also sections for 'PrimitiveType' and 'Boolean'.

```
uml.ecore
Resource Set
platform:/resource/editorUML/model/uml.ecore
uml
NamedElement
Class -> NamedElement
Property -> NamedElement
Association -> NamedElement
Kind
UMLDiagram

Properties
Abstract: false
Default Value: 
ESuper Types: NamedElement
Instance Class Name: 
Interface: false
Name: Class

UML-based editor
uml.ecore
Resource Set
platform:/resource/editorUML/model/uml.ecore
uml
NamedElement
Class -> NamedElement
Property -> NamedElement
Association -> NamedElement
Kind
UMLDiagram

Properties
Abstract: false
Default Value: 
ESuper Types: NamedElement
Instance Class Name: 
Interface: false
Name: Class

DatabaseSchema.km3
package DatabaseSchema {
    abstract class NamedElement {
        attribute name : String;
    }

    class Schema extends NamedElement {
        reference tables[*] container : Table;
    }

    class Table extends NamedElement {
        reference columns[*] ordered container : Column;
        operation drop() : Boolean;
    }

    class Column extends NamedElement {
        -- add more properties here
    }
}

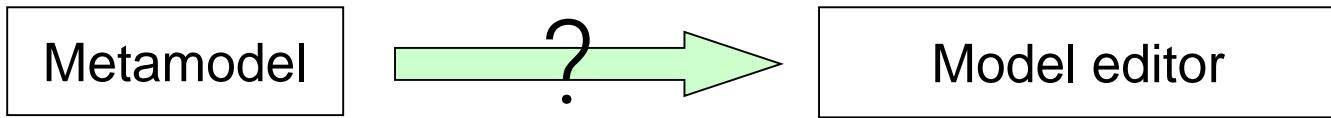
package PrimitiveType {
    datatype String;
    datatype Boolean;
}
```



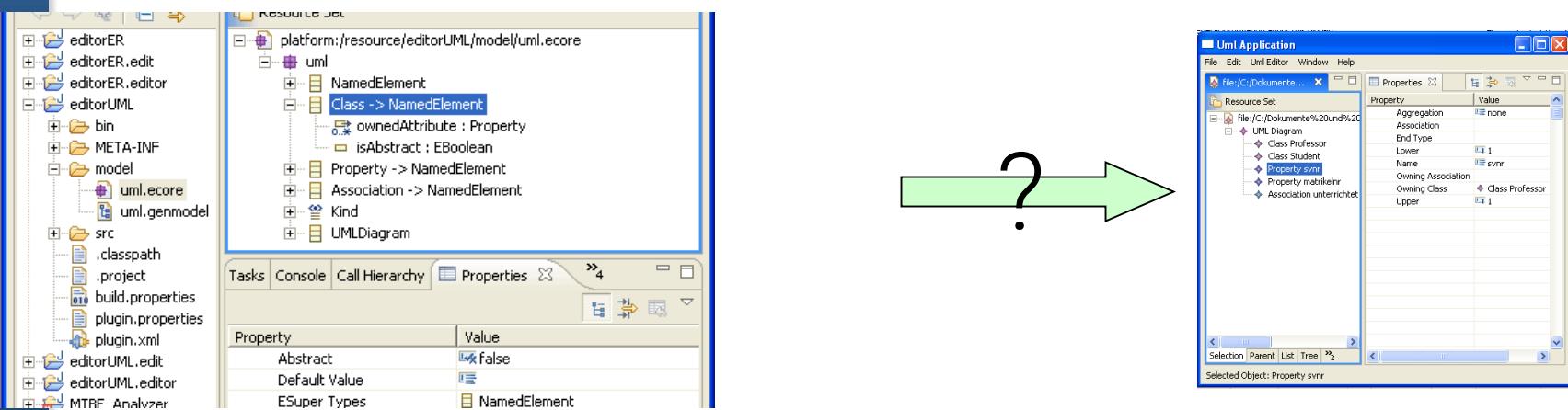
# Eclipse Modeling Framework

## Model editor generation process

- How can a model editor be created out of a metamodel?



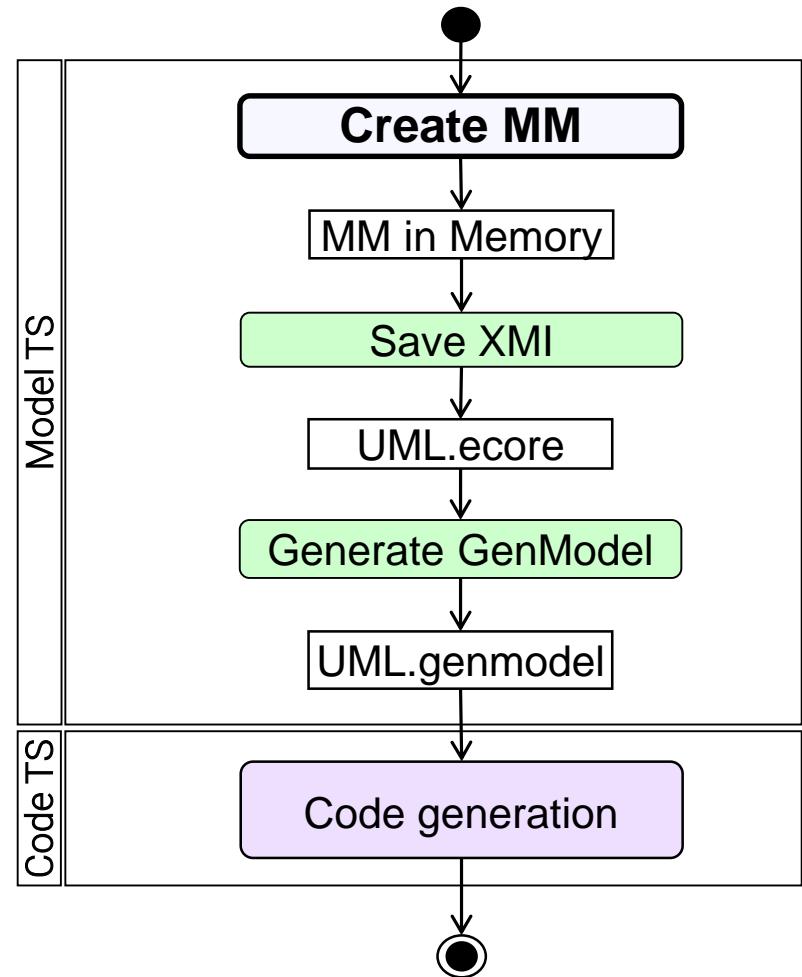
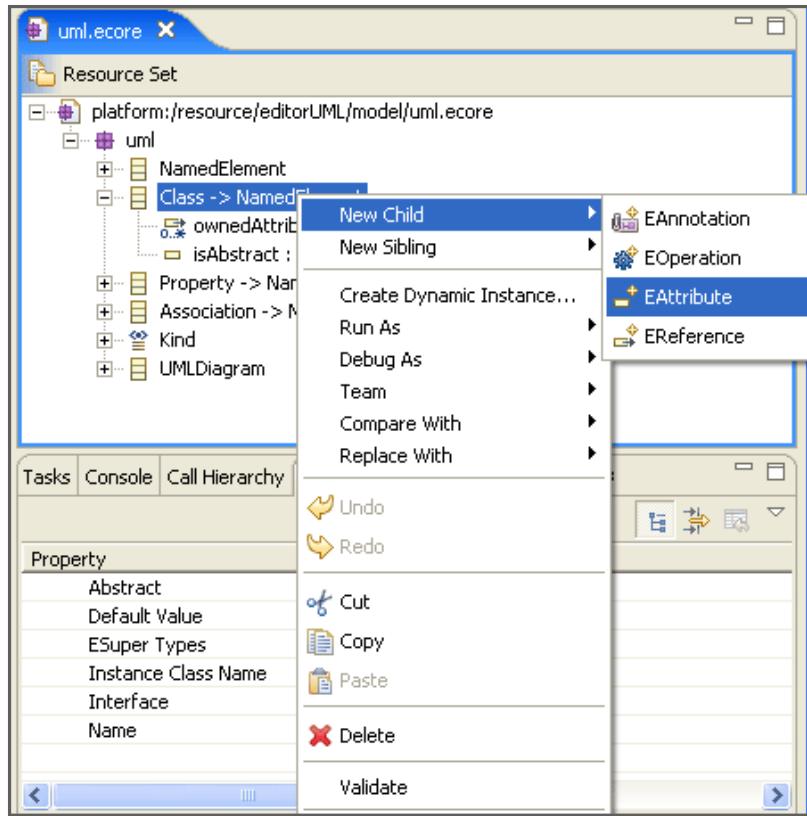
Example: MiniUML metamodel -> MiniUML model editor





# Model editor generation process

- Step 1 – Create metamodel (e.g., with tree editor)



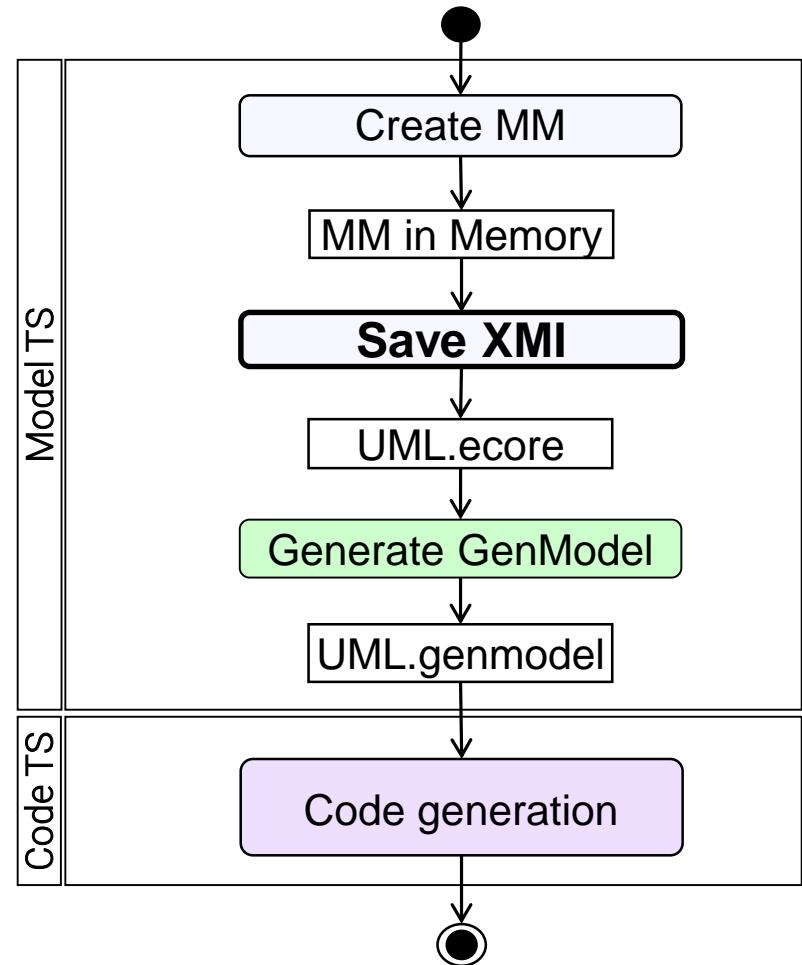


# Model editor generation process

## ■ Step 2 – Save metamodel

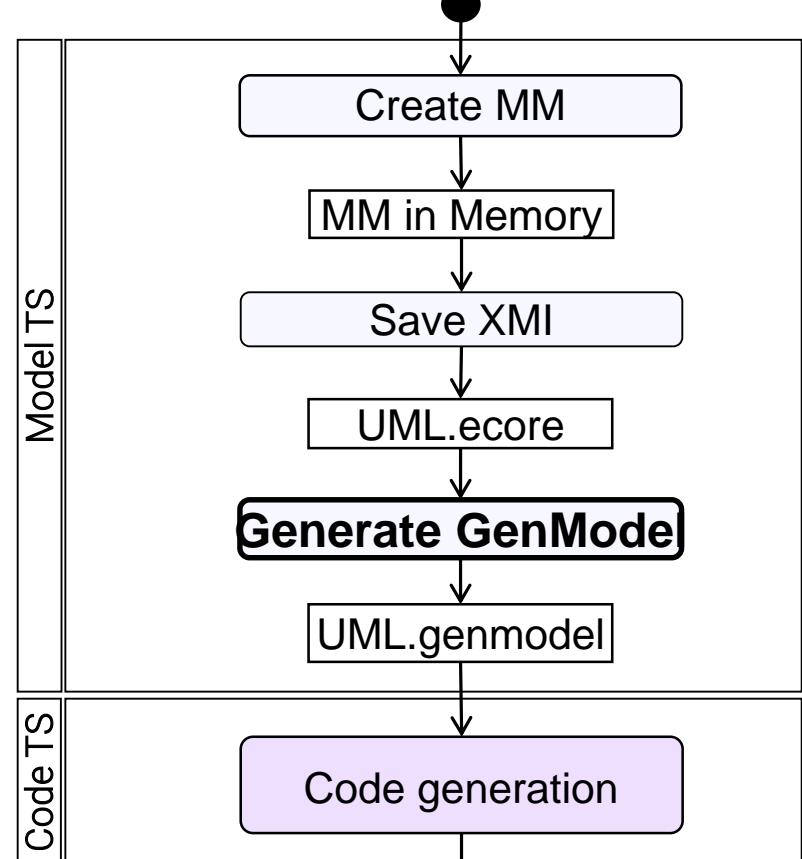
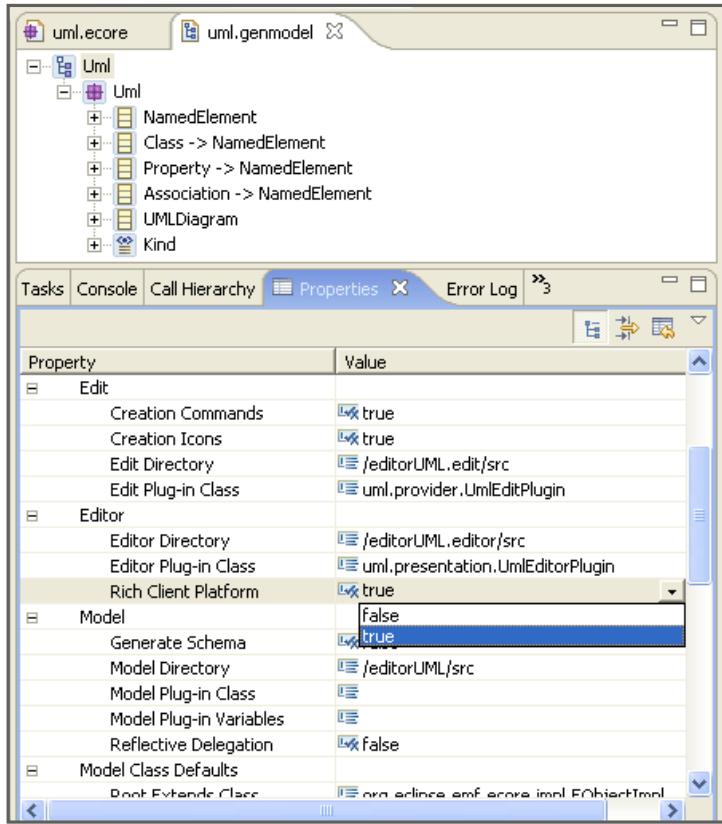
```
<?xml version="1.0" encoding="UTF-8"?>
<ecore:EPackage xmi:version="2.0"
  xmlns:xmi="http://www.omg.org/XMI"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-
    instance"
  xmlns:ecore="http://www.eclipse.org/emf/2002/
    Ecore"
  name="uml"
  nsURI="http://uml" nsPrefix="uml">
  <eClassifiers xsi:type="ecore:EClass"
    name="NamedElement">
    <eStructuralFeatures xsi:type="ecore:EAttribute"
      name="name" eType="ecore:EDataType
      http://www.eclipse.org/emf/2002/
        Ecore#//EString"/>
  </eClassifiers>
  <eClassifiers xsi:type="ecore:EClass" name="Class"
    eSuperTypes="#//NamedElement">
    <eStructuralFeatures xsi:type="ecore:EReference"
      name="ownedAttribute" upperBound="-1"
      eType="#//Property"
      eOpposite="#//Property/owningClass"/>
    <eStructuralFeatures xsi:type="ecore:EAttribute"
      name="isAbstract"
      eType="ecore:EDataType
      http://www.eclipse.org/emf/2002/
        Ecore#//EBoolean"/>
  </eClassifiers>
</ecore:EPackage>
```

UML.ecore



# Model editor generation process

- Step 3 - Generate GenModel



**GenModel** specifies properties for code generation



## Model editor generation process

- Step 4 – Generate model code

- For each meta-class we get:

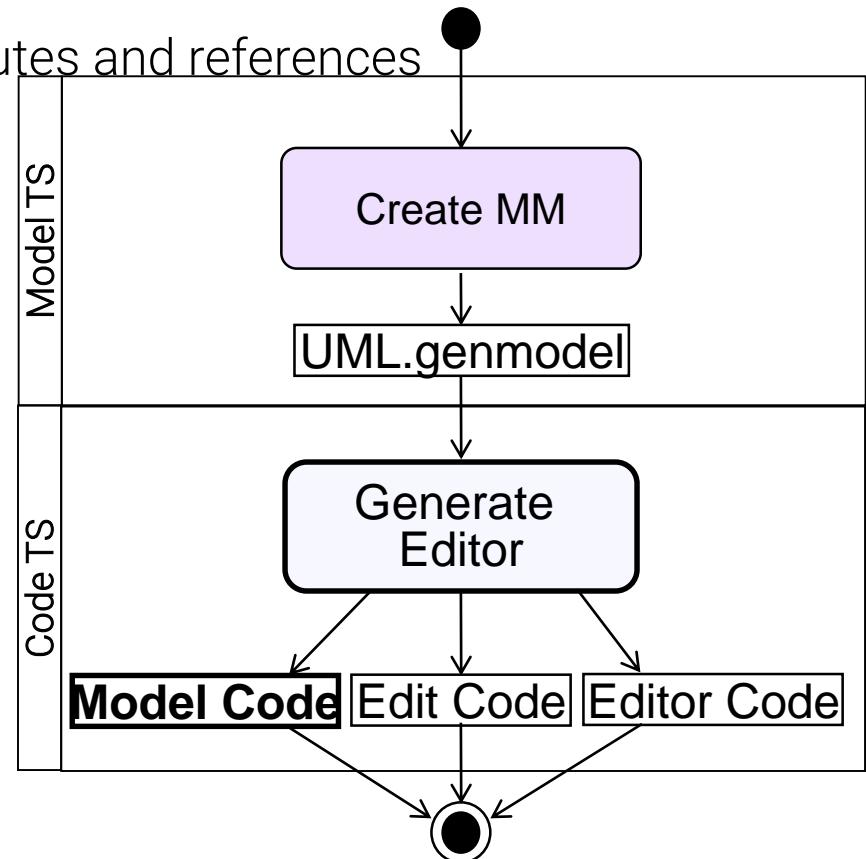
- » Interface: Getter/setter for attributes and references

```
public interface Class extends NamedElement{  
    EList getOwnedAttributes();  
    boolean isIsAbstract();  
    void setIsAbstract(boolean value);  
}
```

- » Implementation class:  
~~Getter/setter implemented~~

```
public class ClassImpl  
    extends NamedElementImpl implements Class{  
    public EList getOwnedAttributes() {  
        return ownedAttributes;  
    }  
    public void setIsAbstract(boolean  
        newIsAbstract) {  
        isAbstract = newIsAbstract;  
    }  
}
```

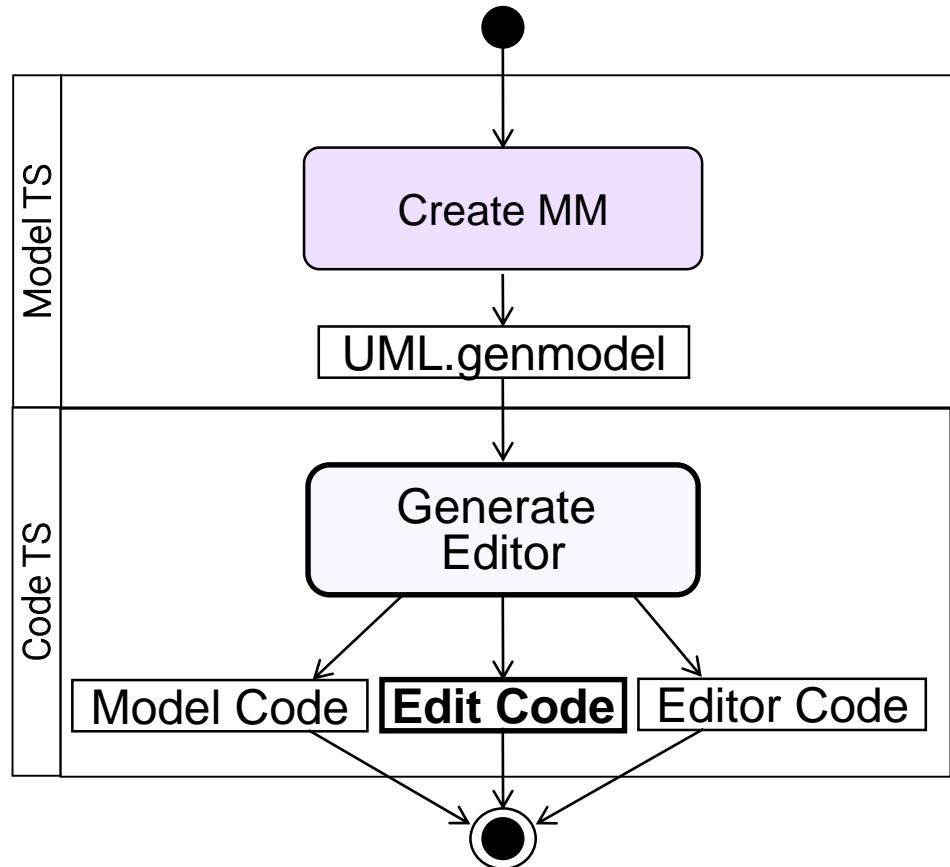
- Factory for the creation of model elements,  
for each Package one Factory-Class is created



## Model editor generation process

- Step 5 – Generate edit code

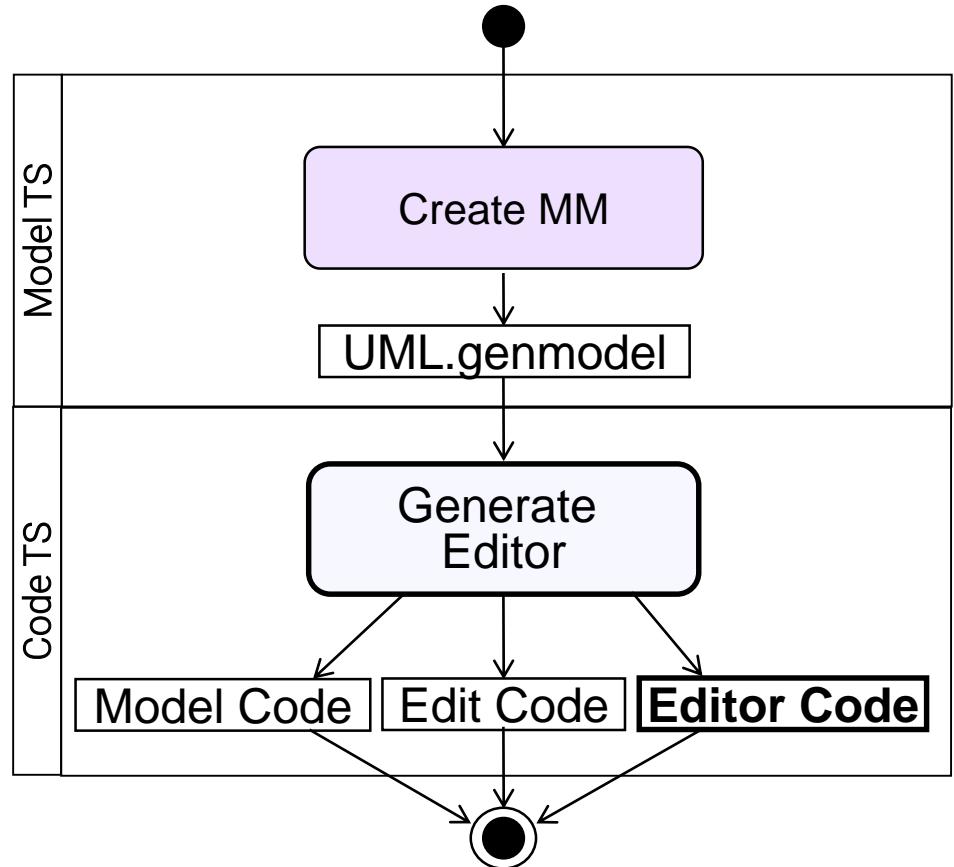
- UI independent editing support for models
- Generated artifacts
  - » TreeContentProvider
  - » LabelProvider
  - » PropertySource



## Model editor generation process

- Step 6 – Generate editor code

- Editor as Eclipse Plugin or RCP Application
- Generated artifacts
  - » Model creation wizard
  - » Editor
  - » Action bar contributor
  - » Advisor (RCP)
  - » plugin.xml
  - » plugin.properties





# Model editor generation process

- Start the modeling editor

**Plugin.xml**

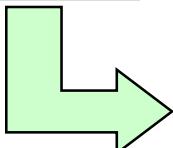
The screenshot shows the 'editorUML.editor' window with the following content:

**General Information**  
This section describes general information about this plug-in:

ID:	editorUML.editor
Version:	1.0.0
Name:	%pluginName
Provider:	%providerName
Class:	uml.presentation.UmlEditorPlugin\$Implementation
Platform filter:	

**Testing**  
Test this plug-in by launching a separate Eclipse application:  
[Launch an Eclipse application](#)  
[Launch an Eclipse application in Debug mode](#)

Click here to start!



**RCP Application**

The screenshot shows the 'Uml Application' RCP application window with the following content:

**Uml Application**

**Resource Set**

- file:/C:/Dokumente%20und%20
- UML Diagram
  - Class Professor
  - Class Student
  - Property svnr
  - Property matrikelnr
  - Association unterrichtet

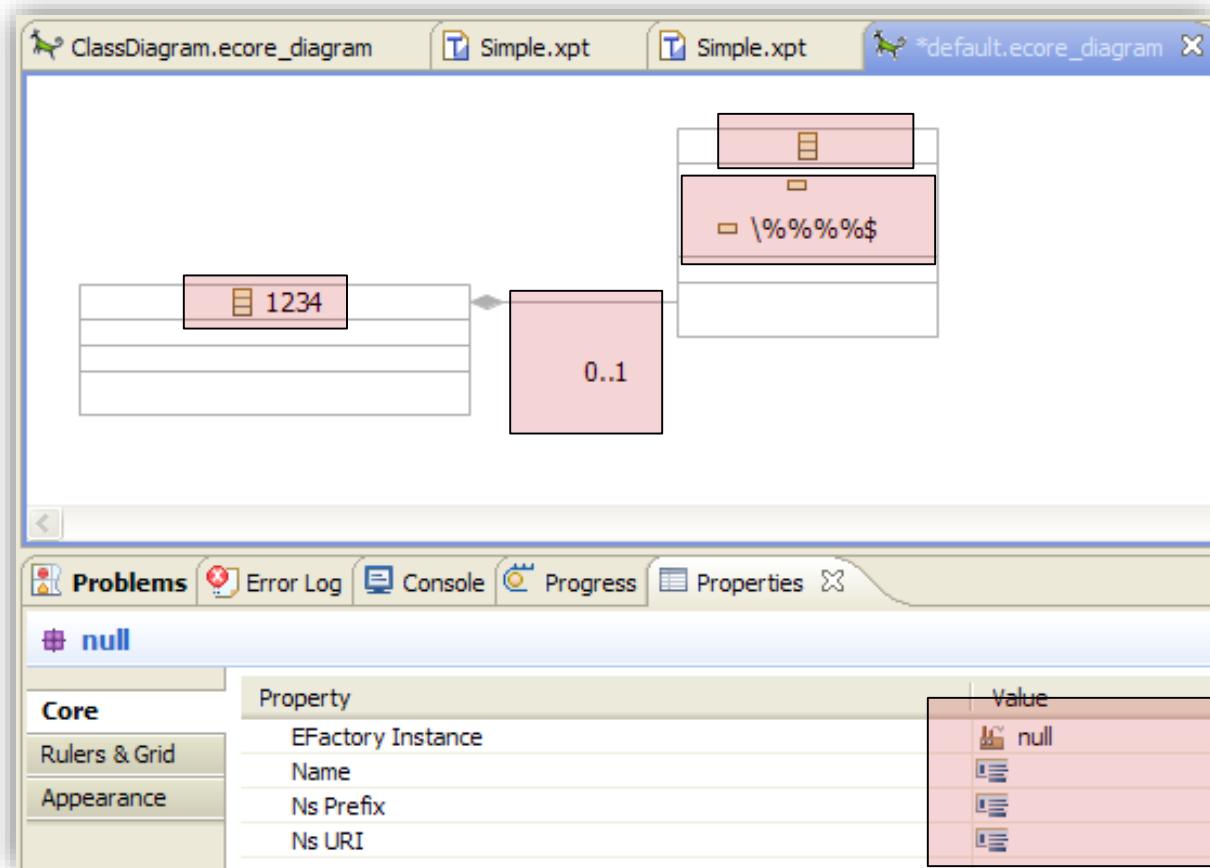
**Properties**

Property	Value
Aggregation	none
Association	
End Type	
Lower	L1
Name	svnr
Owning Association	
Owning Class	Class Professor
Upper	L1

Selected Object: Property svnr

## Metamodels are compiled to Java!

- A (meta)modeling mistake or an error of the EFM code generator?

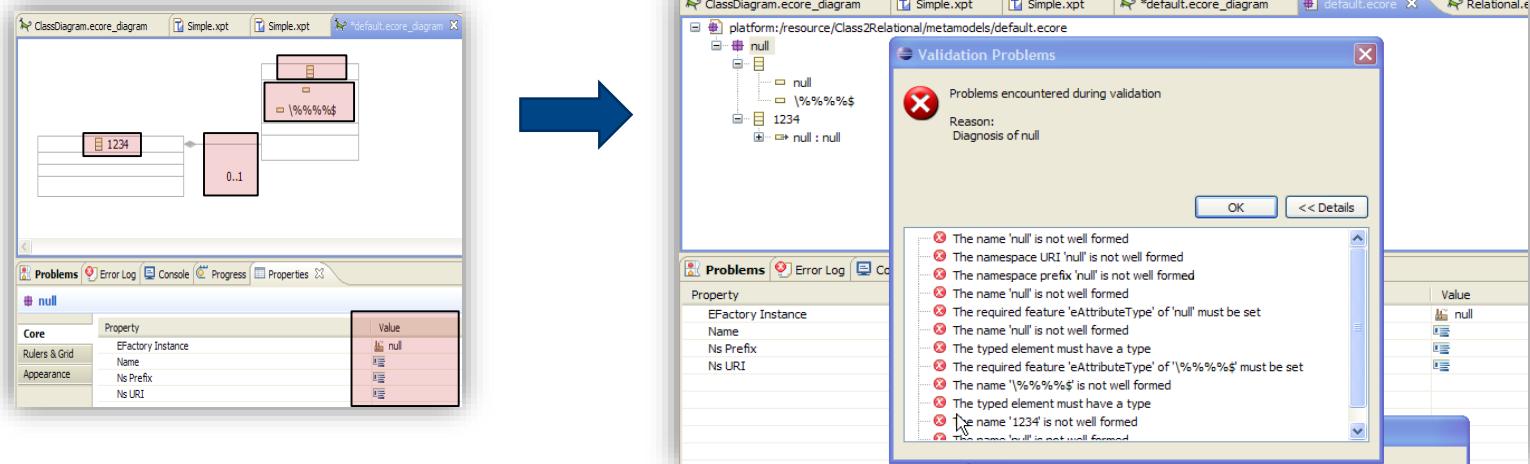




# Metamodels are compiled to Java!

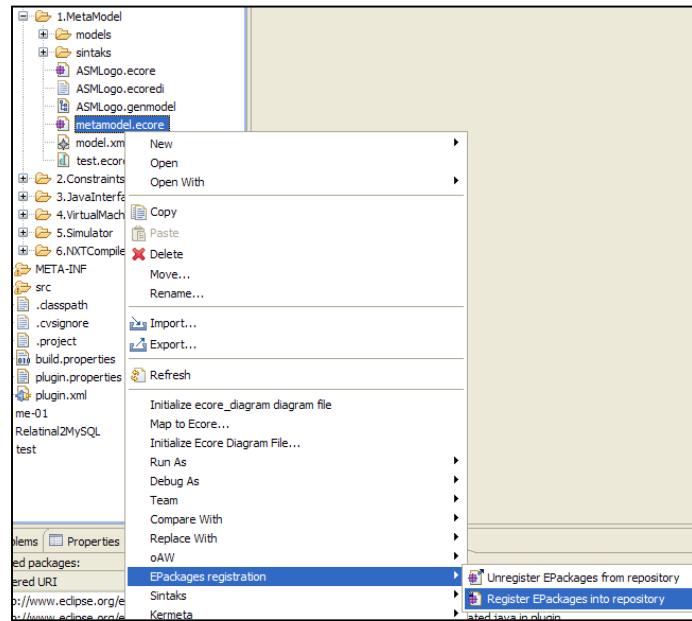
## ■ Attention

- » Only use valid Java identifier as names
  - No blanks, no digits at the beginning, no special characters, ...
- » NamedElements require a name
  - Classes, enumerations, attributes, references, packages
- » Attributes and references require a type
- » Always use the validation service prior to the code generation!!!

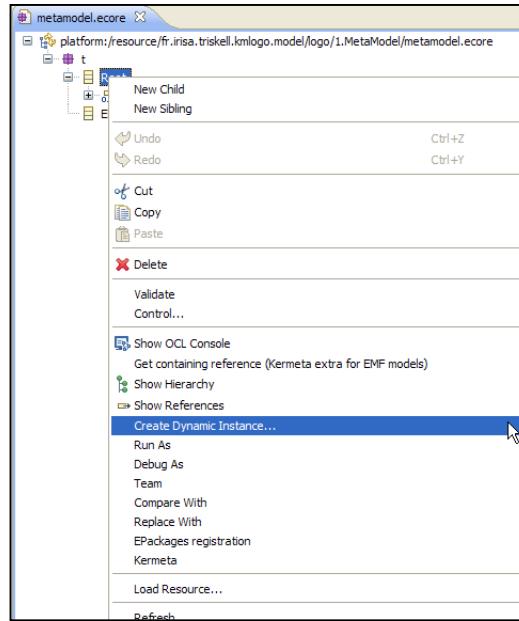


# Shortcut for Metamodel Instantiation Metamodel Registration, Dynamic Model Creation, Reflective Editor

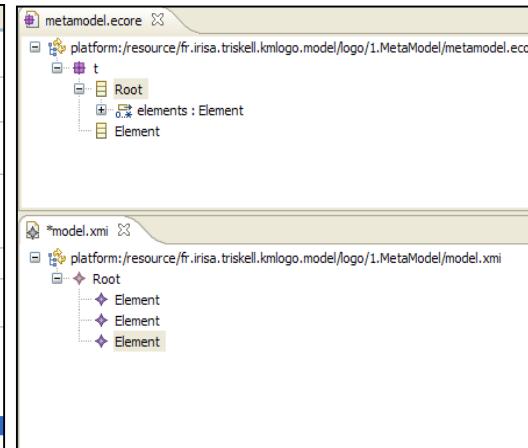
- Rapid testing by
  - » Registration of the metamodel
  - » Select root node (EClass) and create dynamic instance
  - » Visualization and manipulation by Reflective Model Editor



1) Register EPackages



2) Create Dynamic Instance



3) Use Reflective Editor



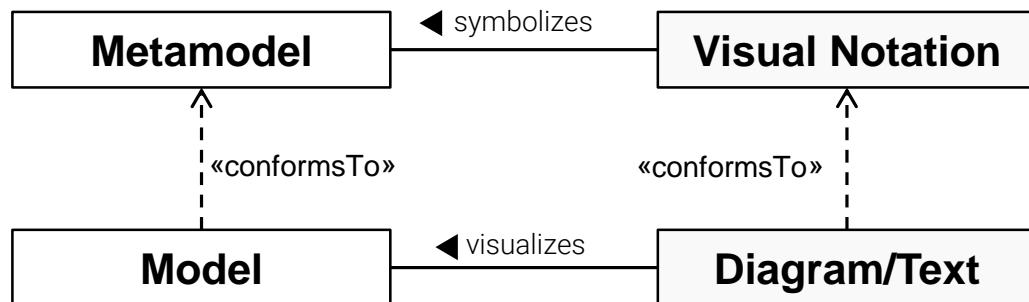
# OCL support for EMF

## Several Plugins available

- **Eclipse OCL Project**
  - » <http://www.eclipse.org/projects/project.php?id=modeling.mdt.ocl>
  - » Interactive OCL Console to query models
  - » Programming support: OCL API, Parser, ...
- **OCLinEcore**
  - » Attach OCL constraints by using EAnnotations to metamodel classes
  - » Generated modeling editors are aware of constraints
- **Dresden OCL**
  - » Alternative to Eclipse OCL
- **OCL influenced languages, but different syntax**
  - » Epsilon Validation Language residing in the Epsilon project
  - » Check Language residing in the oAW project

# Introduction concrete and abstract syntax

- The visual notation of a model language is referred as concrete syntax
- Formal definition of concrete syntax allows for automated generation of editors
- Several approaches and frameworks available for defining concrete syntax for model languages



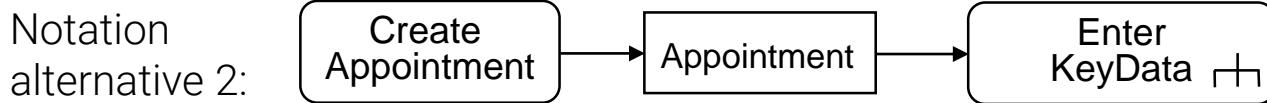
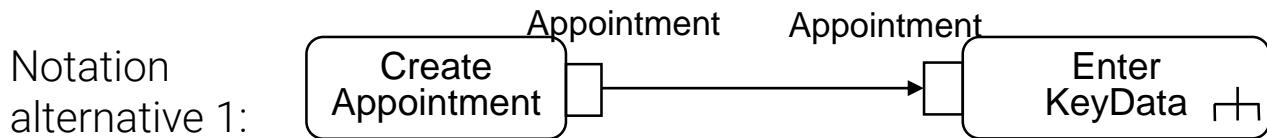
# Introduction concrete and abstract syntax

- Several languages have no formalized definition of their concrete syntax
- Example – Excerpt from the UML-Standard

<i>NODE TYPE</i>	<i>NOTATION</i>	<i>REFERENCE</i>
ForkNode		See ForkNode (from IntermediateActivities) on page -404.
InitialNode		See InitialNode (from BasicActivities) on page -406.
JoinNode		See “JoinNode (from CompleteActivities, IntermediateActivities)” on page 411.
MergeNode		See “MergeNode (from IntermediateActivities)” on page 416.

# Introduction concrete and abstract syntax

- Concrete syntax improves the readability of models
  - » Abstract syntax not intended for humans!
- One abstract syntax may have multiple concrete ones
  - » Including textual and/or graphical
  - » Mixing textual and graphical notations still a challenge!
- Example – Notation alternatives for the creation of an appointment



Notation alternative 3:

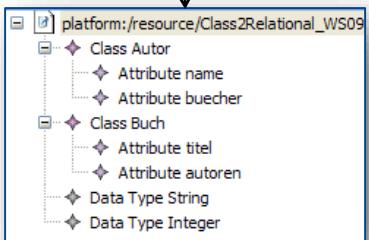
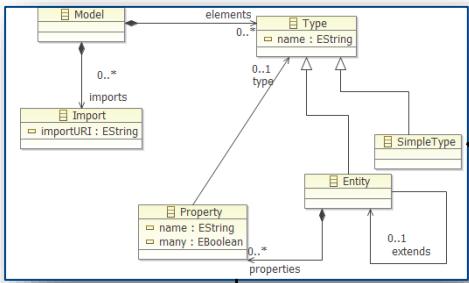
```
Appointment a;  
a = new Appointment;  
EnterKeyData (a);
```



# Introduction concrete and abstract syntax

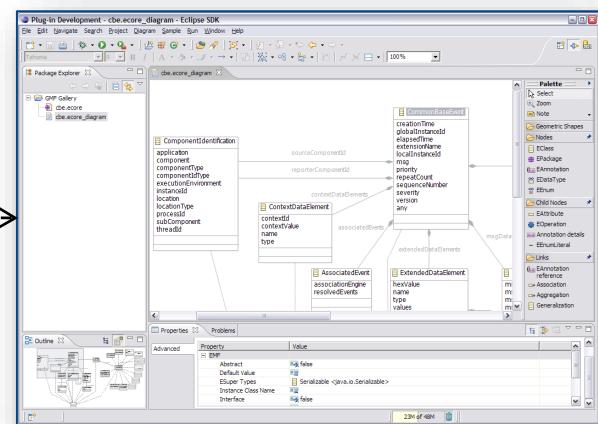
## ■ Concrete Syntaxes in Eclipse

### Ecore-based Metamodels



Generic tree-based EMF  
Editor

### Graphical Concrete Syntax



### Textual Concrete Syntax

```
platform:/resource/Class2Relational_WS09
  Class Autor
    Attribute name
    Attribute buecher
  Class Buch
    Attribute titel
    Attribute autoren
  Data Type String
  Data Type Integer
```

```
type String
type Bool

entity Session {
  property Title: String
  property IsTutorial : Bool
}

entity Conference {
  property Name : String
  property Attendees : Person[]
  property Speakers : Session[]
}

entity Person {
  property Name : String
}

entity Speaker extends Person
  property Sessions : Session[]
```

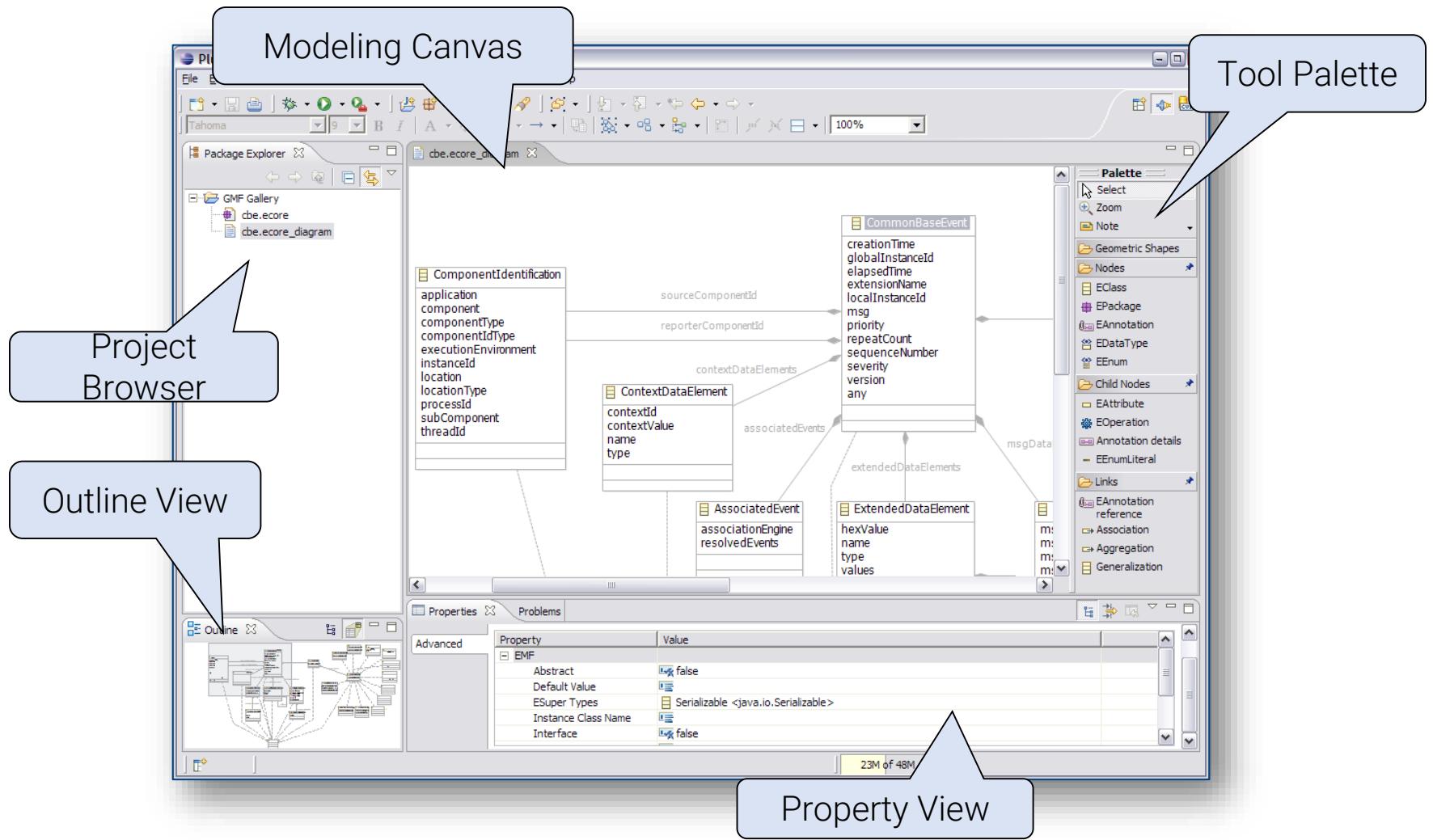


# Introduction concrete and abstract syntax

- A Graphical Concrete Syntax (GCS) consists of
  - » graphical symbols,
    - e.g., rectangles, circles, ...
  - » compositional rules,
    - e.g., nesting of elements, ...
  - » and mapping between graphical symbols and abstract syntax elements.
    - e.g., a class in the metamodel is visualized by a rectangle in the GCS



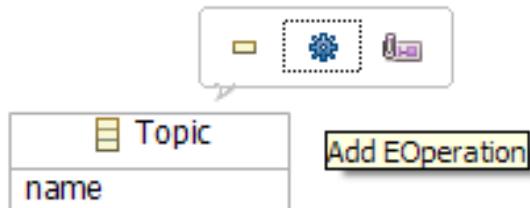
# Anatomy of Graphical Modeling Editors



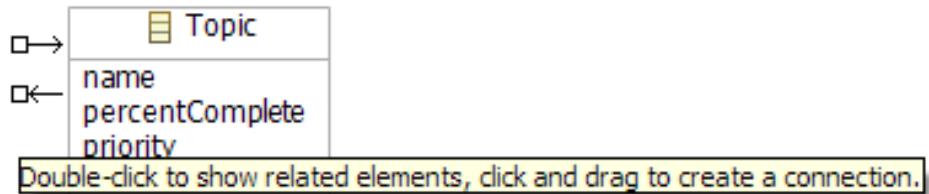


# Features of Graphical Modeling Editors

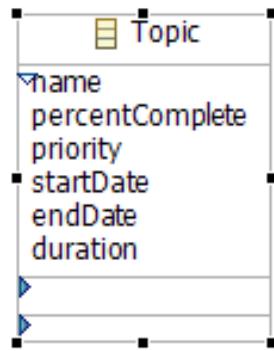
## Action Bars:



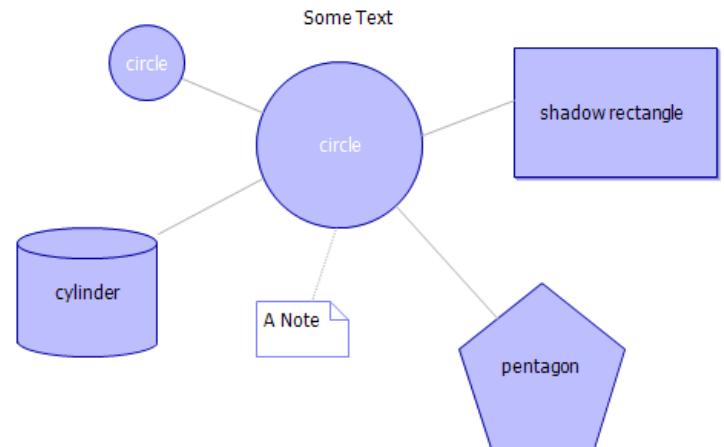
## Connection Handles:



## Collapsed Compartments:



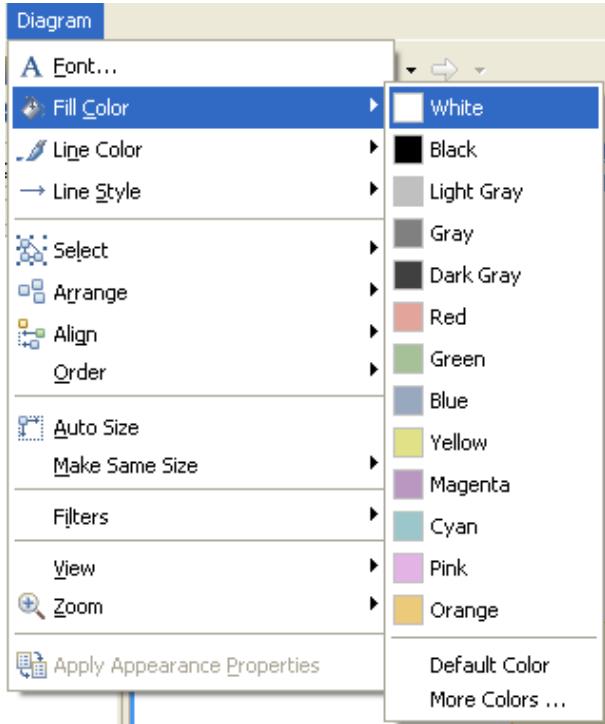
## Geometrical Shapes:





# Features of Graphical Modeling Editors

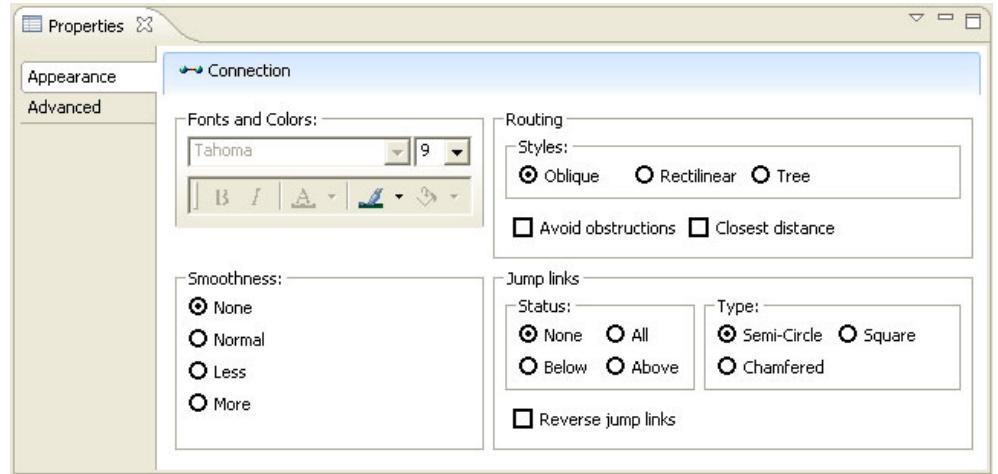
## Actions:



## Toolbar:

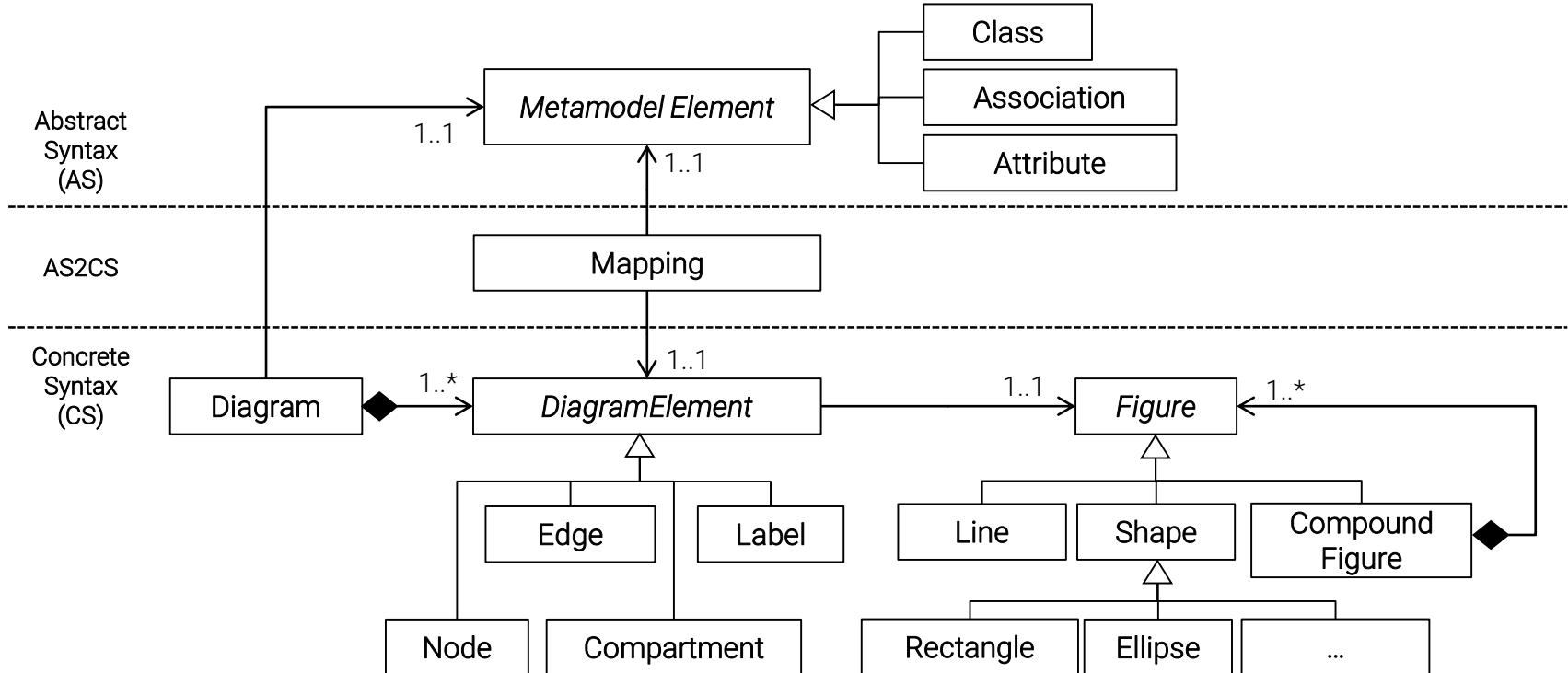


## Properties View:





# Generic Metamodel for GCS

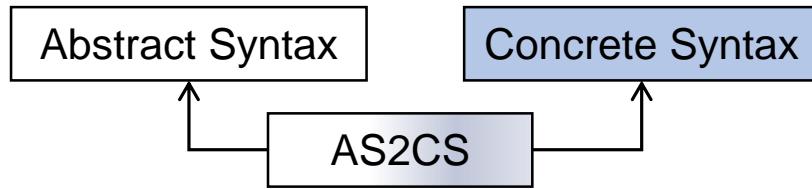




## GCS Approaches

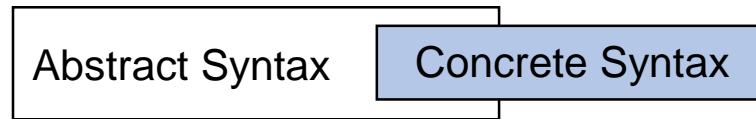
- **Mapping-based**

- » Explicit mapping model between abstract syntax, i.e., the metamodel, and concrete syntax



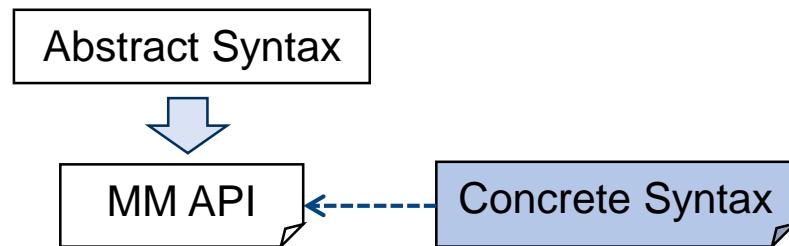
- **Annotation-based**

- » The metamodel is annotated with concrete syntax information



- **API-based**

- » Concrete syntax is described by a programming language using a dedicated API for graphical modeling editors

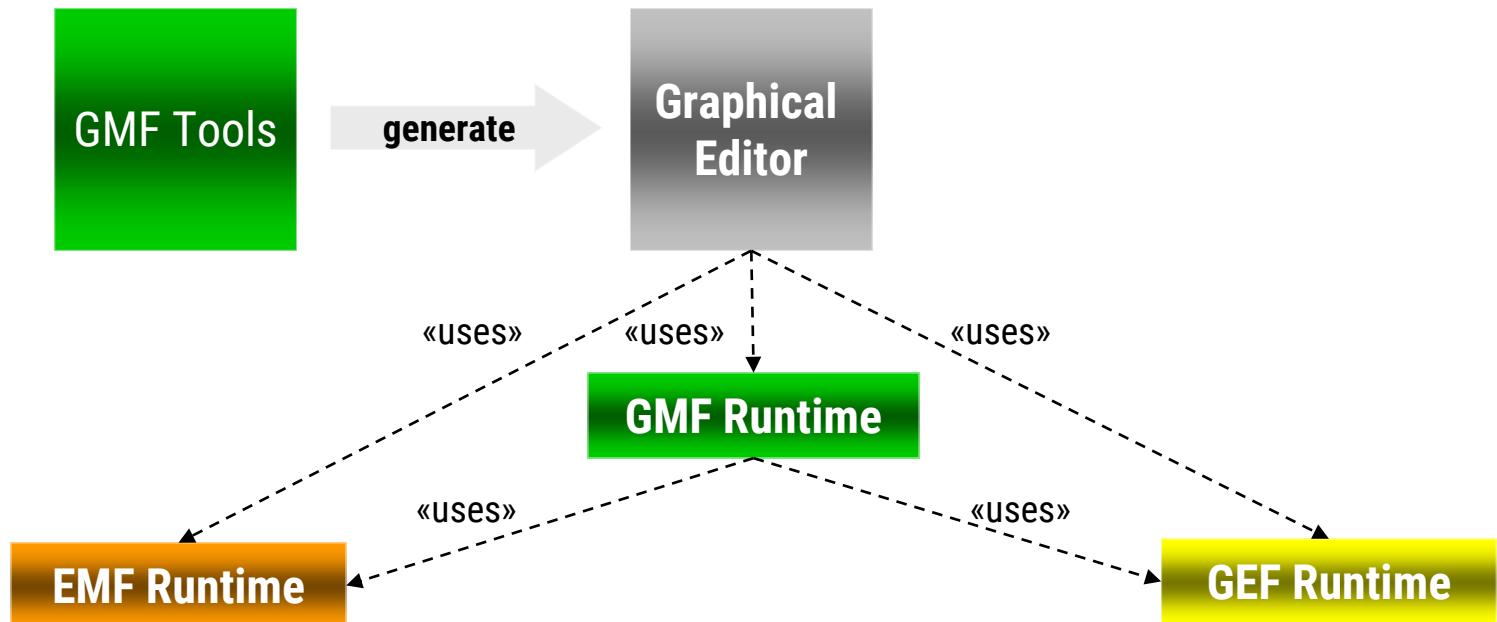




# Mapping-based Approach: GMF

## Basic Architecture of GMF

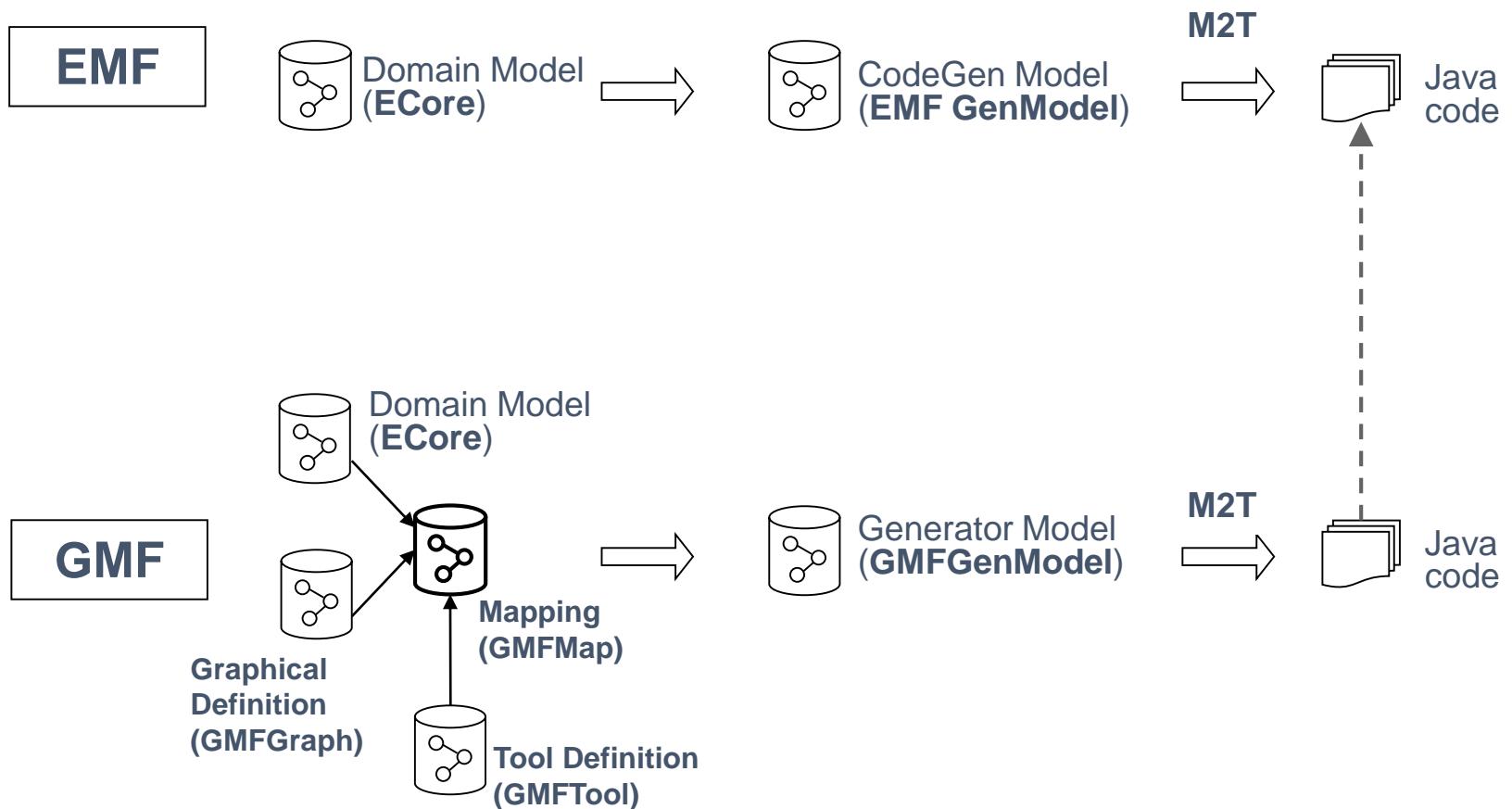
- "The Eclipse Graphical Modeling Framework (GMF) provides a generative component and runtime infrastructure for developing graphical editors based on EMF and GEF." - [www.eclipse.org/gmf](http://www.eclipse.org/gmf)





## Mapping-based Approach: GMF

- Tooling Component





## Annotation-based Approach: Eugenia

- Hosted in the Epsilon project
  - » Kick-starter for developing graphical modeling editors
  - » <http://www.eclipse.org/epsilon/doc/eugenia/>
- Ecore metamodels are annotated with GCS information
- From the annotated metamodels, a generator produces GMF models
- GMF generators are reused to produce the actual modeling editors
- Be aware:  
Application of MDE techniques for developing MDE tools!!!

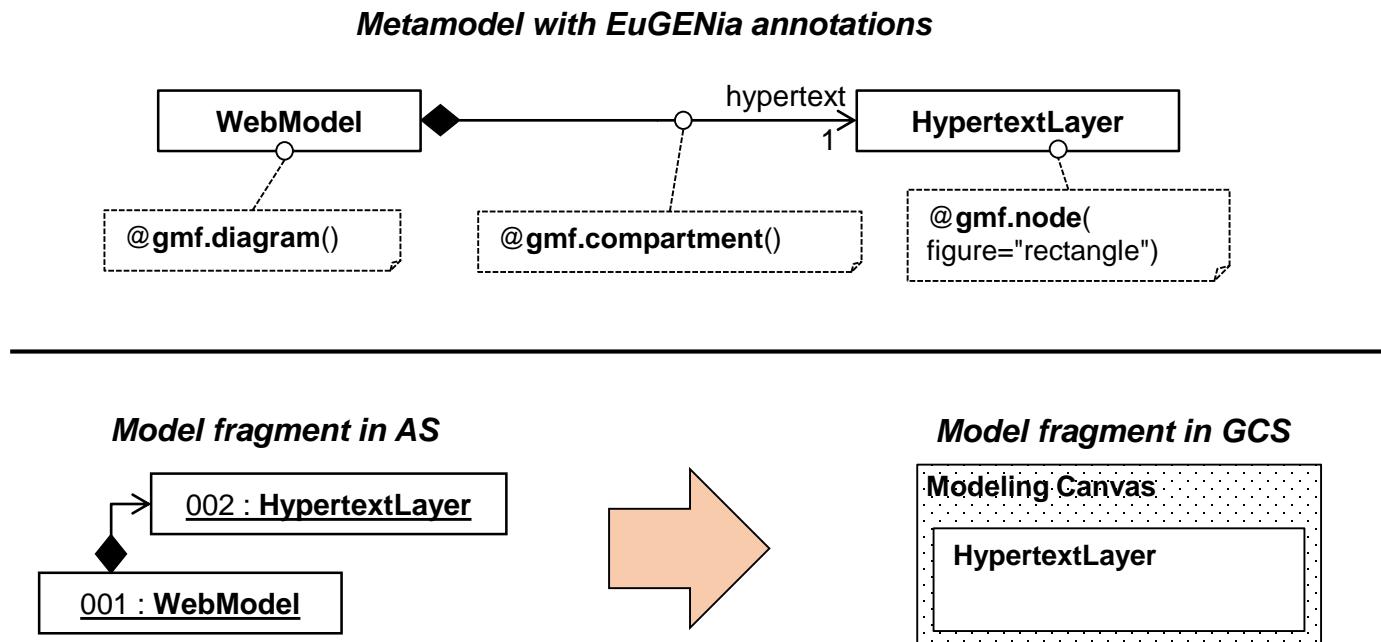


# Eugenia Annotations (Excerpt)

- **Diagram**
  - » For marking the root class of the metamodel that directly or transitively contains all other classes
  - » Represents the modeling canvas
- **Node**
  - » For marking classes that should be represented by nodes such as rectangles, circles, ...
- **Link**
  - » For marking references or classes that should be visualized as lines between two nodes
- **Compartment**
  - » For marking elements that may be nested in their containers directly
- **Label**
  - » For marking attributes that should be shown in the diagram representation of the models

## Eugenia Example #1

- HypertextLayer elements should be directly embeddable in the modeling canvas that represents WebModels

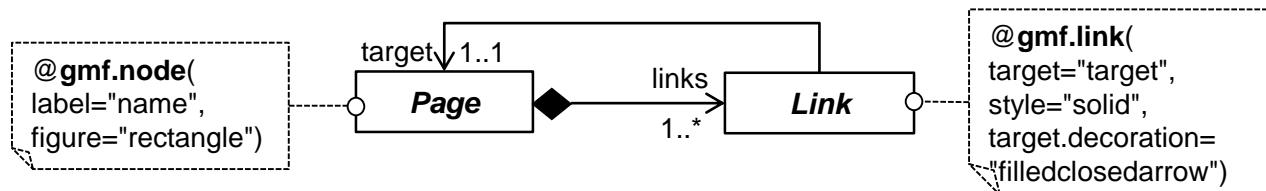




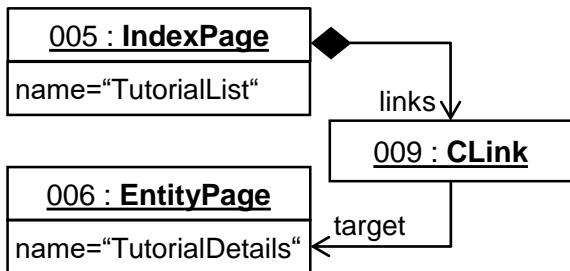
## Eugenia Example #2

- Pages should be displayed as rectangles and Links should be represented by a directed arrow between the rectangles

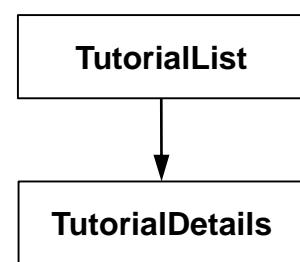
**Metamodel with EuGENia annotations**



**Model fragment in AS**



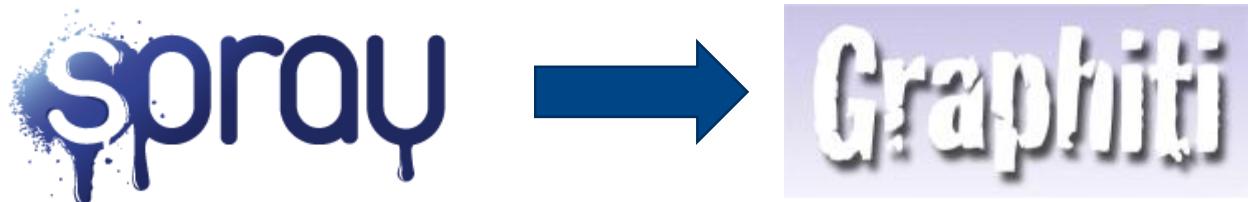
**Model fragment in GCS**





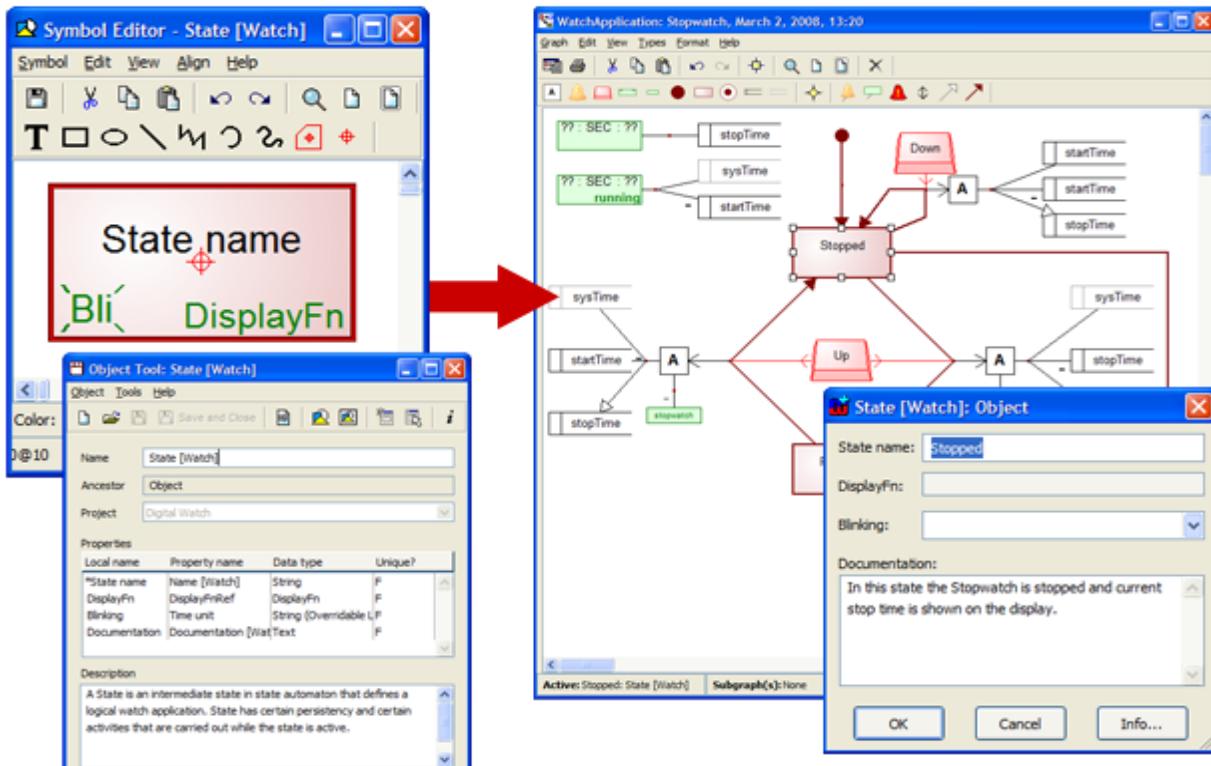
## API-based Approach: Graphiti

- Powerful programming framework for developing graphical modeling editors
- Base classes of Graphiti have to be extended to define concrete syntaxes of modeling languages
  - » Pictogram models describe the visualization and the hierarchy of concrete syntax elements (cf. .gmfgraph models of GMF)
  - » Link models establish the mapping between abstract and concrete syntax elements (cf. .gmfmap models of GMF)
- DSL on top of Graphiti: Spray



# Other Approaches outside Eclipse MetaEdit+

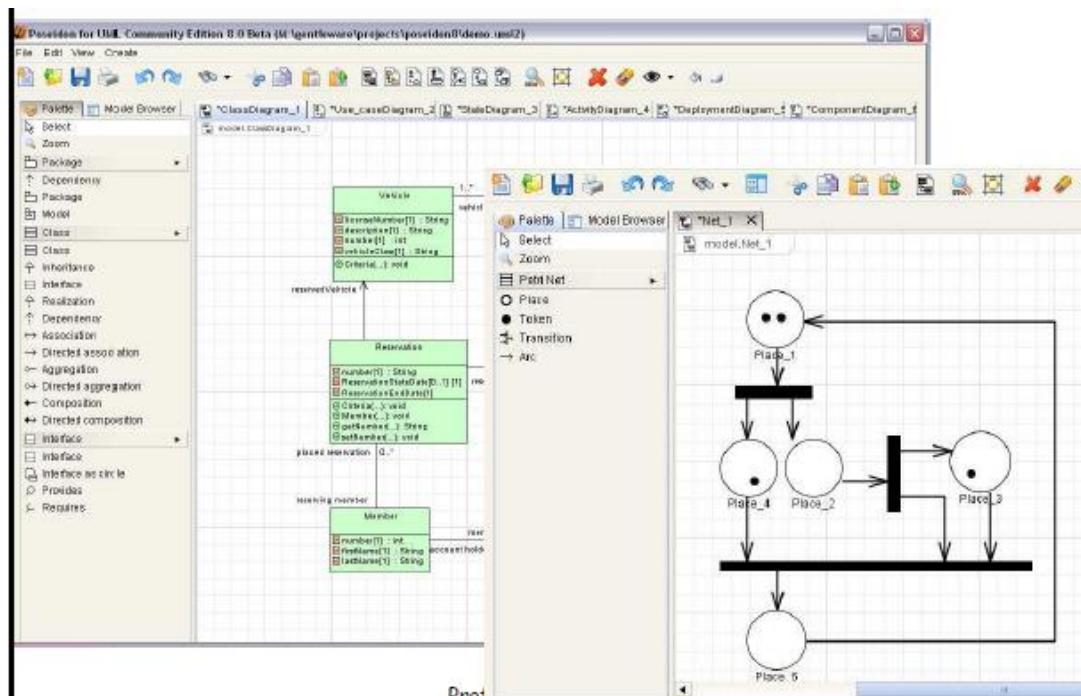
- Metamodeling tool outside Eclipse (commercial product)
- Graphical specification of figures in graphical editor
- Special tags to specify labels in the figures by querying the models





# Other approaches outside Eclipse Poseidon

- UML Tool
- Uses textual syntax to specify mappings, figures, etc.
  - » Based on Xtext
  - » Provides dedicated concrete syntax text editor





# Textual Modeling Languages

- Long tradition in software engineering
  - » General-purpose programming languages
  - » But also a multitude of domain-specific (programming) languages
    - Web engineering: HTML, CSS, Jquery, ...
    - Data engineering: SQL, XSLT, XQuery, Schematron, ...
    - Build and Deployment: ANT, MAVEN, Rake, Make, ...
- Developers are often used to textual languages
- Why not using textual concrete syntaxes for modeling languages?



# Textual Modeling Languages

- Textual languages defined either as internal or external languages
- Internal languages
  - » Embedded languages in existing host languages
  - » Explicit internal languages
    - Becoming mainstream through Ruby and Groovy
  - » Implicit internal languages
    - Fluent interfaces simulate languages in Java and C#
- External languages
  - » Have their own custom syntax
  - » Own parser to process them
  - » Own editor to build sentences
  - » Own compiler/interpreter for execution of sentences
  - » Many XML-based languages ended up as external languages
    - Not very user-friendly



# Textual Modeling Languages

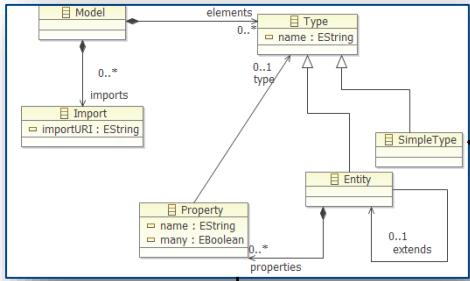
- Textual languages have specific strengths compared to graphical languages
  - » Scalability, pretty-printing, ...
- Compact and expressive syntax
  - » Productivity for experienced users
  - » Guidance by IDE support softens learning curve
- Configuration management/versioning
  - » Concurrent work on a model, especially with a version control system
  - » Diff, merge, search, replace, ...
  - » But be aware, some conflicts are hard to detect on the text level!
  - » Dedicated model versioning systems are emerging!



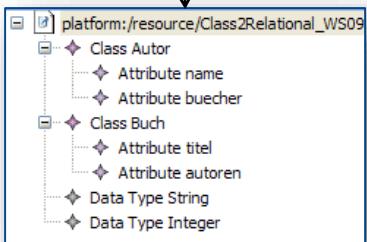
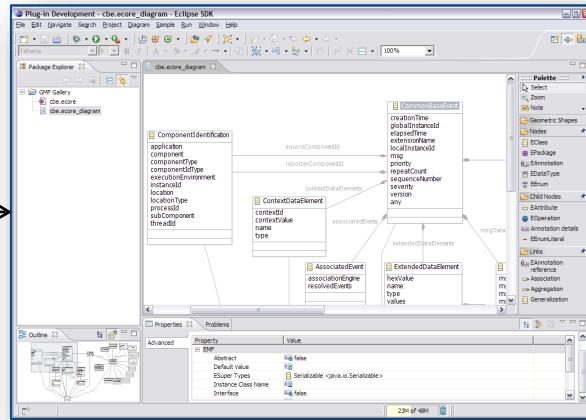
# Textual Concrete Syntax

## Concrete Syntaxes in Eclipse

### Ecore-based Metamodels



### Graphical Concrete Syntax



### Generic tree-based EMF Editor

### Textual Concrete Syntax

```
type String
type Bool

entity Session {
    property Title: String
    property IsTutorial : Bool
}

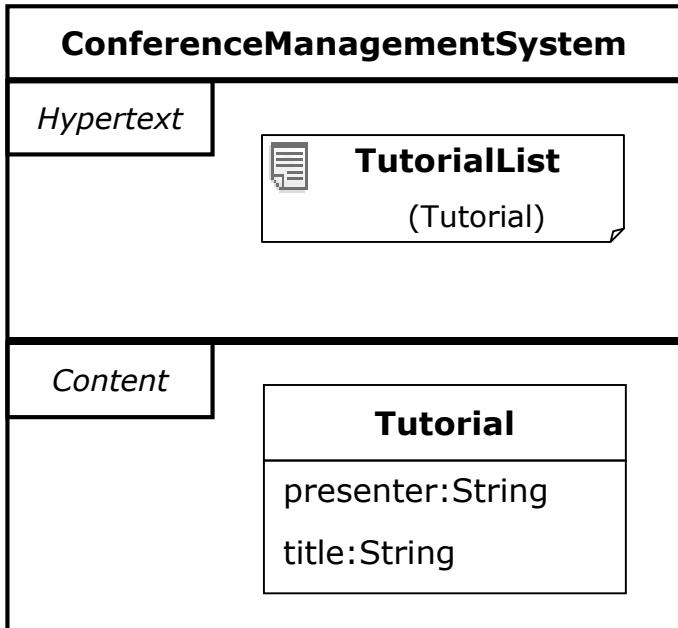
entity Conference {
    property Name : String
    property Attendees : Person[]
    property Speakers : Speaker[]
}

entity Person {
    property Name : String
}

entity Speaker extends Person
    property Sessions : Session[]
```

# Every GCS is transformable to a TCS

- Example: sWML



```

webapp ConferenceManagementSystem{

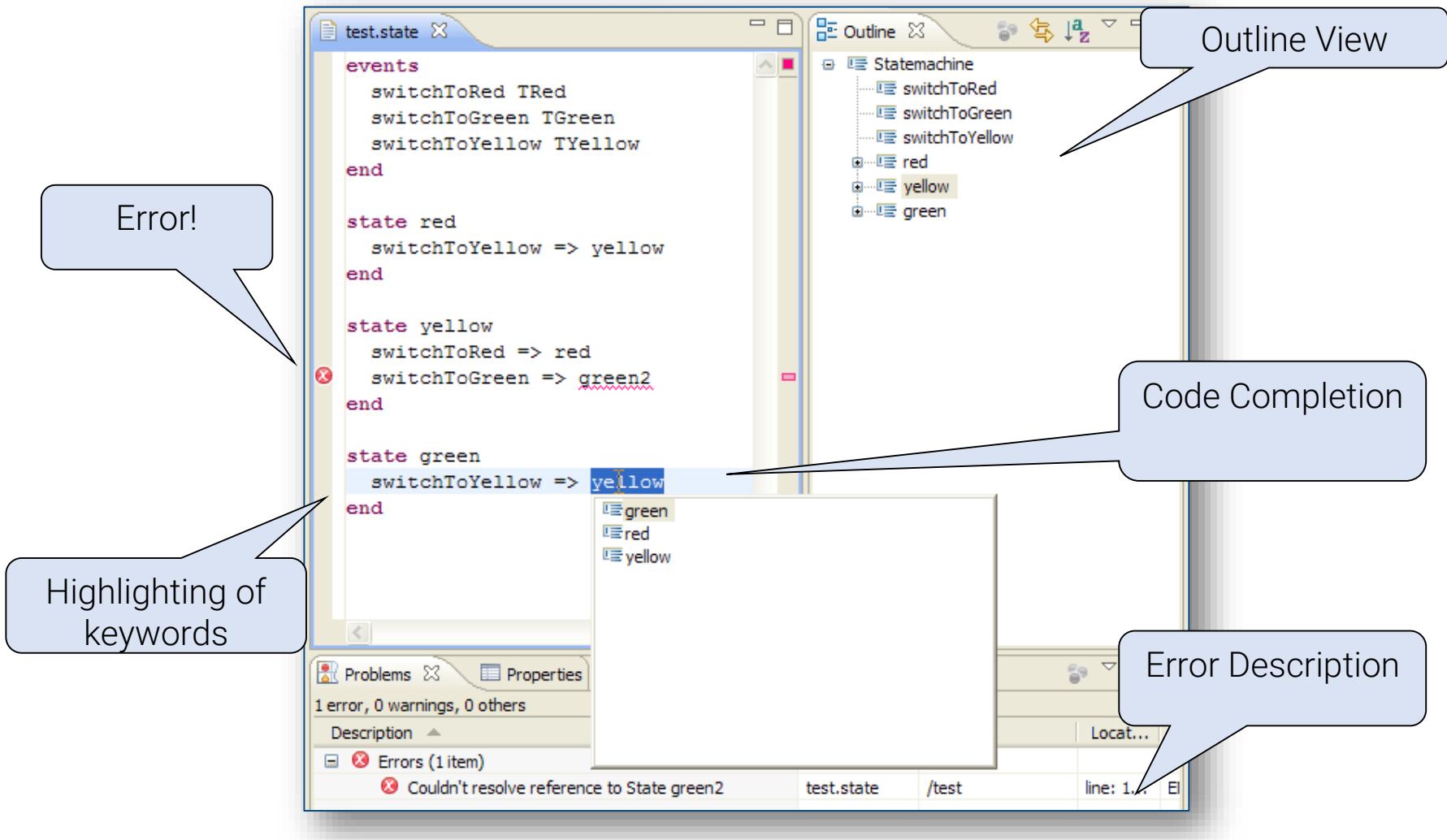
hypertext{
  index TutorialList shows Tutorial [10] {...}
}

content{
  class Tutorial {
    att presenter : String;
    att title : String;
  }
}
}
  
```

The code defines a **webapp** named **ConferenceManagementSystem**. It includes sections for **hypertext** and **content**. The **hypertext** section contains an **index** that shows a **TutorialList** containing up to 10 **Tutorial** objects. The **content** section defines a **Tutorial** class with attributes for **presenter** and **title**.



# Anatomy of Modern Text Editors





## Excursus: Textual Languages in the Past

- Extended Backus-Naur-Form (EBNF)
  - » Originally introduced by Niklaus Wirth to specify the syntax of Pascal
  - » In general, they can be used to specify a context-free grammar
  - » ISO Standard
- Fundamental assumption: A text consists of a sequence of terminal symbols (visible characters).
- EBNF specifies all valid terminal symbol sequences using production rules → grammar
- Production rules consist of a left side (name of the rule) and a right side (valid terminal symbol sequences)



# Textual Languages

## EBNF

- Production rules consist of
  - » Terminal
  - » NonTerminal
  - » Choice
  - » Optional
  - » Repetition
  - » Grouping
  - » Comment
  - » ...

Usage	Notation
definition	=
concatenation	,
termination	;
alternation	
option	[ ... ]
repetition	{ ... }
grouping	( ... )
terminal string	" ... "
terminal string	' ... '
comment	(* ... *)
special sequence	? ... ?
exception	-



# Textual Languages

## Entity DSL

- Example:

```
type String  
type Boolean
```

```
entity Conference {  
    property name : String  
    property attendees : Person[]  
    property speakers : Speaker[]  
}
```

```
entity Person {  
    property name : String  
}
```

```
entity Speaker extends Person {  
    ...  
}
```



# Textual Languages

## Entity DSL

- Sequence analysis

```
type String  
type Boolean
```

```
entity Conference {  
    property name : String  
    property attendees : Person[]  
    property speakers : Speaker[]  
}
```

```
entity Person {  
    property name : String  
}
```

```
entity Speaker extends Person {  
}
```

### Legend:

- **Keywords**
- **Scope borders**
- **Separation characters**
- **Reference**
- **Arbitrary character sequences**



# Textual Languages

## Entity DSL

- **EBNF Grammar**

Model := Type\*;

Type := SimpleType | Entity;

SimpleType := 'type' ID;

Entity := 'entity' ID ('extends' ID)? '{' Property\* '}';

Property := 'property' ID ':' ID ('[]')?;

ID := ('a'..'z'|'A'..'Z'|'\_') ('a'..'z'|'A'..'Z'|'|\_')|'0'..'9')\*;

# Textual Languages

## Entity DSL



### ■ EBNF vs. Ecore

Model := Type\*;

Type := SimpleType | Entity;

SimpleType := 'type' ID;

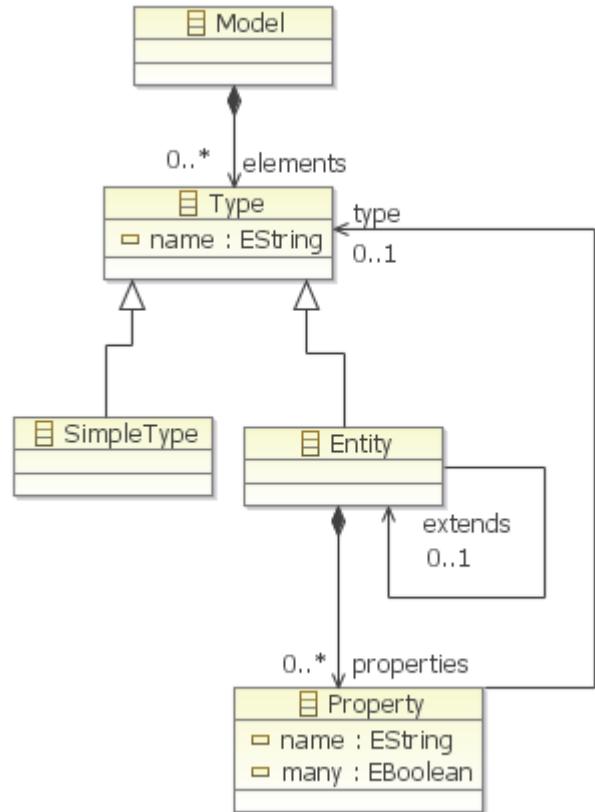
Entity := 'entity' ID ('extends' ID)? '{'

Property\* '}';

Property := 'property' ID ':' ID ('[]')?;

ID := ('a'..'z'|'A'..'Z'|'\_')

('a'..'z'|'A'..'Z'|'\_'|'0'..'9')\*;





# Textual Languages

## EBNF vs. Ecore

- **EBNF**
  - » Specifies concrete syntax
  - » Linear order of elements
  - » No reusability
  - » Only containment relationships
- **Ecore**
  - » Reusability by inheritance
  - » Non-containment and containment references
  - » Predefined data types and user-defined enumerations
  - » Specifies only abstract syntax
- **Conclusion**
  - » A meaningful EBNF cannot be generated from a metamodel and vice versa!
- **Challenge**
  - » How to overcome the gap between these two worlds?



# Textual Languages Solutions

## Generic Syntax

- » Like XML for serializing models
- » Advantage: Metamodel is sufficient, i.e., no concrete syntax definition is needed
- » Disadvantage: no syntactic sugar!
- » Protagonists: UML and XMI (OMG Standards)

## Language-specific Syntax

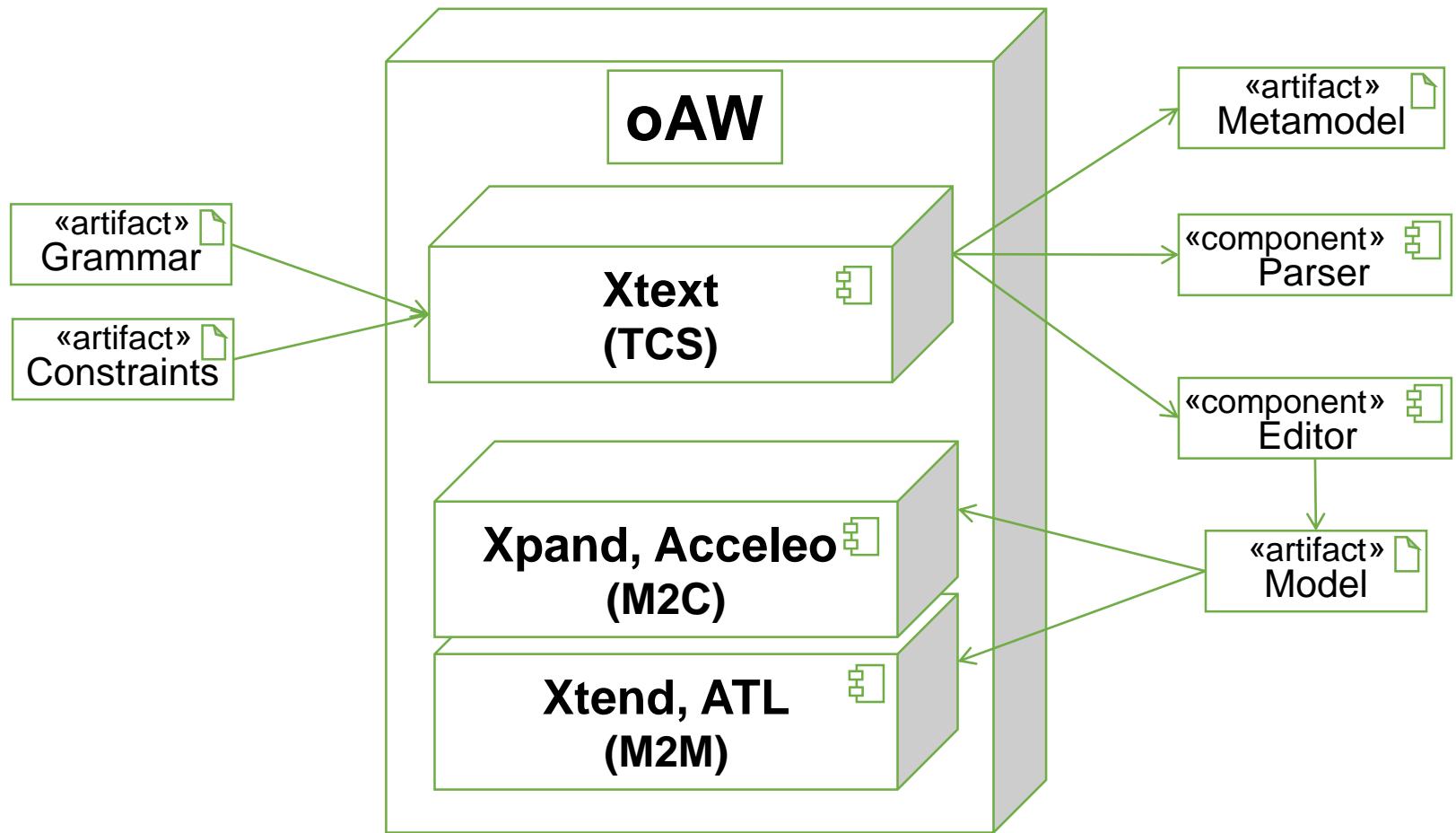
- Metamodel First!
  - » Step 1: Specify metamodel
  - » Step 2: Specify textual syntax
  - » For instance: TCS (Eclipse Plug-in)
- Grammar First!
  - » Step 1: Syntax is specified by a grammar (concrete syntax & abstract syntax)
  - » Step 2: Metamodel is derived from output of step 1, i.e., the grammar
  - » For instance: Xtext (Eclipse Plug-in)
    - Alternative process: take a metamodel and transform it to an initial Xtext grammar!



# Xtext Introduction

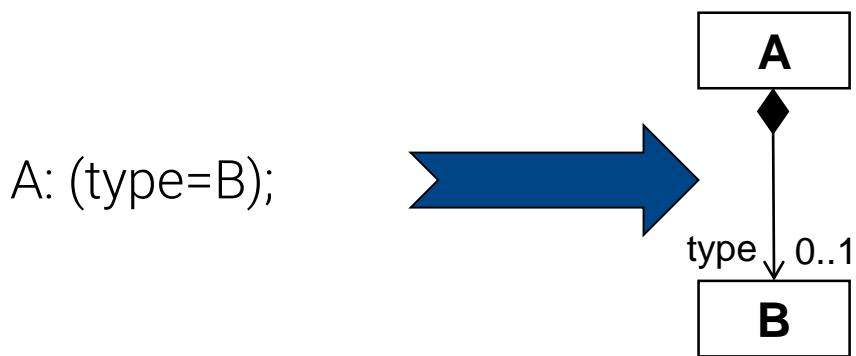
- Xtext is used for developing textual domain specific languages
- Grammar definition similar to EBNF, but with additional features inspired by metamodeling
- Creates metamodel, parser, and editor from grammar definition
- Editor supports syntax check, highlighting, and code completion
- Context-sensitive constraints on the grammar described in OCL-like language

# Xtext Introduction



# Xtext Grammar

- Xtext grammar is similar to EBNF
- But extended by
  - » Object-oriented concepts
  - » Information necessary to derive metamodels and modeling editors
- Example





# Xtext Grammar

- Terminal rules
  - » Similar to EBNF rules
  - » Return value is String by default
- EBNF expressions
  - » Cardinalities
    - ? = One or none; \* = Any; + = One or more
  - » Character Ranges    '0' .. '9'
  - » Wildcard                'f' . 'o'
  - » Until Token            '/\*' -> '\*/'
  - » Negated Token          '#' (! '#' ) \* '#'
- Predefined rules
  - » ID, String, Int, URI



# Xtext Grammar

- Examples

- terminal ID :

```
('^')? ('a'...'z' | 'A'...'Z' | '_') ('a'...'z' | 'A'...'Z' | '_' | '0'...'9')*;
```

- terminal INT returns.ecore::EInt :

```
('0'...'9')+;
```

- terminal ML\_COMMENT :

```
'/*' -> '*' / ' ';
```



# Xtext Grammar

- **Type rules**
  - » For each type rule a class is generated in the metamodel
  - » Class name corresponds to rule name
- **Type rules contain**
  - » Terminals -> Keywords
  - » Assignments -> Attributes or containment references
  - » Cross References -> Non Containment references
  - » ...
- **Assignment Operators**
  - » = for features with multiplicity 0..1
  - » += for features with multiplicity 0..\*
  - » ?= for Boolean features



# Xtext Grammar

Examples:

- Assignment

State :

```
'state' name=ID  
(transitions+=Transition)*  
'end';
```

- Cross References

Transition :

```
event=[Event] '=>' state=[State];
```



# Xtext Grammar

- **Enum rules**
  - » Map Strings to enumeration literals
- **Examples:**

```
enum ChangeKind :  
    ADD | MOVE | REMOVE  
;
```

```
enum ChangeKind :  
    ADD = 'add' | ADD = '+' |  
    MOVE = 'move' | MOVE = '->' |  
    REMOVE = 'remove' | REMOVE = '-'  
;
```

# Xtext Tooling

## ■ Xtext Grammar Definition

Grammar Name

Default Terminals (ID, STRING,...)

Metamodel URI

```
StateMachine.xtext
1grammar org.xtext.example.StateMachine with org.eclipse.xtext.common.Terminals
2
3generate stateMachine "http://www.xtext.org/example/StateMachine"
4
5Statemachine :
6    'events'
7    (events+=Event)+ 
8    'end'
9    ('resetEvents'
10    (resetEvents+=[Event])+ 
11    'end')?
12    'commands'
13    (commands+=Command)+ 
14    'end'
15    (states+=State)+;
```



# Xtext Tooling

## ■ Xtext Grammar Definition for State Machines

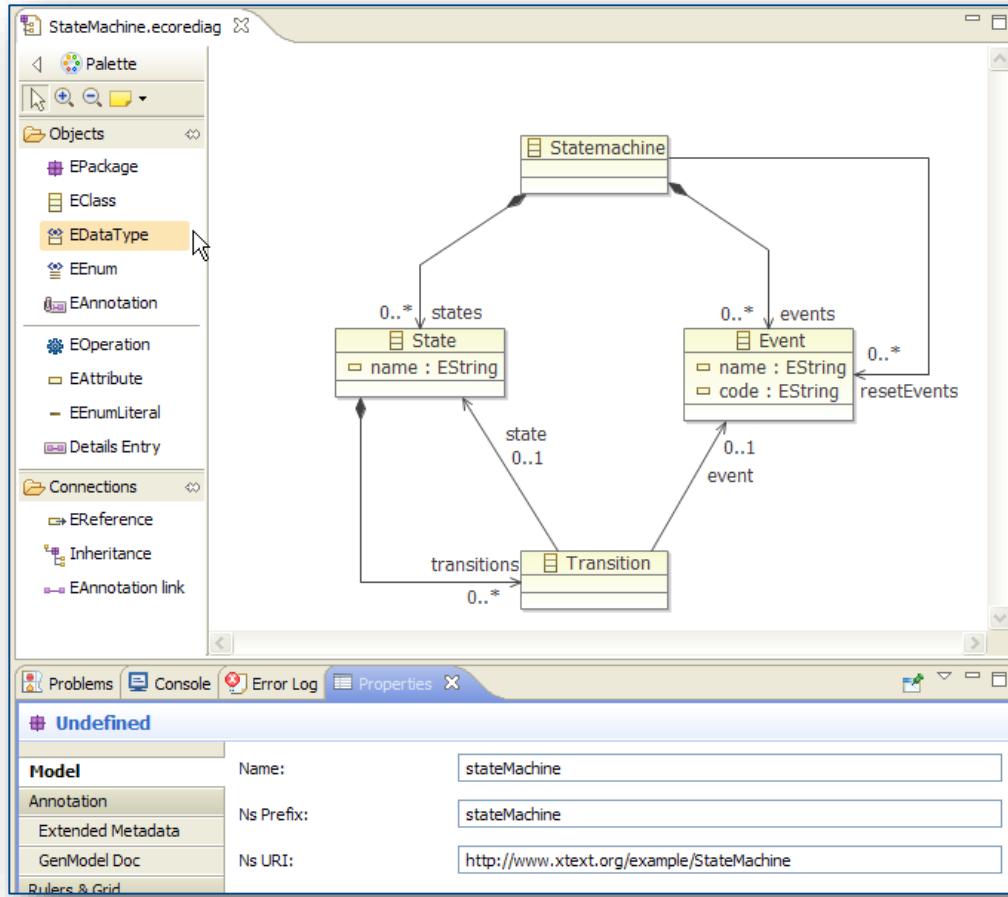
The screenshot shows a code editor window titled "StateMachine.xtext". The content is an Xtext grammar definition:

```
1grammar org.xtext.example.StateMachine with org.eclipse.xtext.common.Terminals
2
3generate stateMachine "http://www.xtext.org/example/StateMachine"
4
5Statemachine :
6    'events'
7        (events+=Event)+*
8    'end'
9    ('resetEvents'
10        (resetEvents+=[Event])+*
11    'end')?
12    (states+=State)+;
13
14Event :
15    name=ID code=ID;
16
17State :
18    'state' name=ID
19        (transitions+=Transition)*
20    'end';
21
22Transition :
23    event=[Event] '>' state=[State];
24
25
```



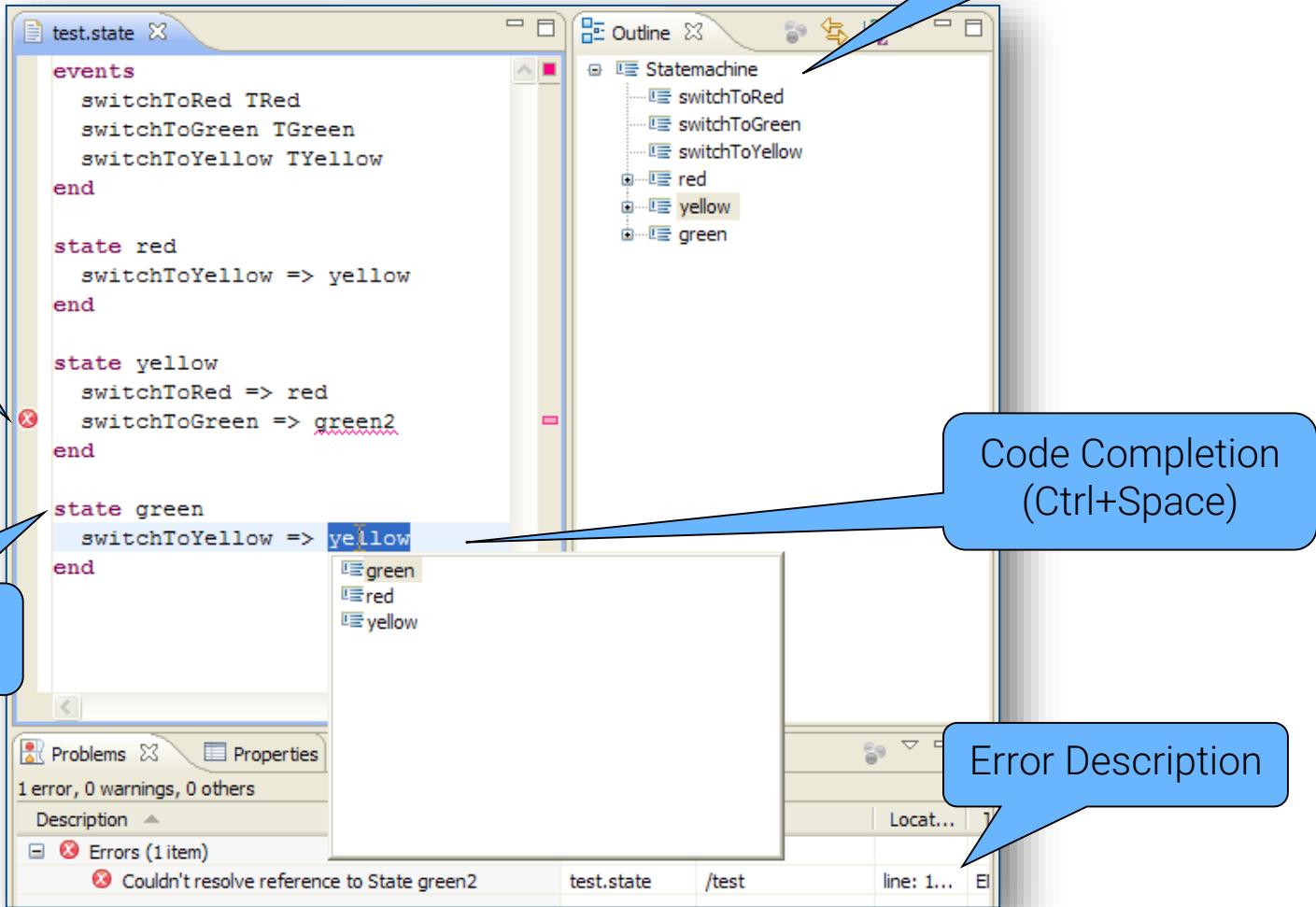
# Xtext Tooling

- Automatically generated Ecore-based Metamodel



# Xtext Tooling

## ■ Generated DSL Editor



# Example #1: Entity DSL

## Entity DSL Revisited



### Example Model

```
type String
type Bool

entity Conference {
    property name : String
    property attendees : Person[]
    property speakers : Speaker[]
}

entity Person {
    property name : String
}

entity Speaker extends Person {
    ...
}
```

### EBNF Grammar

```
Model := Type*;

Type := SimpleType | Entity;

SimpleType := 'type' ID;

Entity := 'entity' ID ('extends' ID)? '{'
    Property* '}';

Property := 'property' ID ':' ID ('[]')?;

ID := ('a'..'z'|'A'..'Z'|'_')
    ('a'..'z'|'A'..'Z'|_|'0'..'9')*;
```



# Example #1

## From EBNF to Xtext

### EBNF Grammar

```
Model := Type*;

Type := SimpleType | Entity;

SimpleType := 'type' ID;

Entity := 'entity' ID
  ('extends' ID)? '{'
    Property*
  }';

Property := 'property' ID ':'
  ID ('[]')?;

ID := ('a'..'z'|'A'..'Z'|'_')
  ('a'..'z'|'A'..'Z'|'_'|'0'..'9')*;
```

### Xtext Grammar

```
grammar MyDsl with
  org.eclipse.xtext.common.Terminals

generate myDsl "http://MyDsl"

Model : elements+=Type*;

Type: SimpleType | Entity;

SimpleType: 'type' name=ID;

Entity : 'entity' name=ID
  ('extends' extends=[Entity])? '{'
    properties+=Property*
  }';

Property: 'property' name=ID ':'
  type=[Type] (many?='[]')?;
```



# Example #1

## How to specify context sensitive constraints for textual DSLs?

### Xtext Grammar

```
grammar MyDsl with
org.eclipse.xtext.common.Terminals

generate myDsl "http://MyDsl"

Model : elements+=Type*;

Type: SimpleType | Entity;

SimpleType: 'type' name=ID;

Entity : 'entity' name=ID
  ('extends' extends=[Entity])? '{'
    properties+=Property*
  '}';
  
Property: 'property' name=ID ':'
  type=[Type] (many?='[]')?;
```

#### ■ Examples

- » Entity names must start with an Upper Case character
- » Entity names must be unique
- » Property names must be unique within one entity

#### ■ Answer

- » Use the same techniques as for metamodels!



## Example #1

### How to specify context sensitive constraints for textual DSLs?

- Examples
  - » Entity names must start with an Upper Case character
  - » Entity names must be unique within one model
  - » Property names must be unique within one entity
- Solution shown in Check language (similar to OCL)
  1. **context** myDsl::Entity  
**WARNING** "Name should start with a capital":  
name.toFirstUpper() == name;
  2. **context** myDsl::Entity  
**ERROR** "Name must be unique":  
(Model)this.eContainer().elements.name.  
select(e|e == this.name).size == 1;
  3. **context** myDsl::Property  
**ERROR** "Name must be unique":  
(Entity)this.eContainer().properties.name.  
select(p|p == this.name).size == 1;



# Example #1

## When to evaluate context sensitive constraints?

- Every edit operation for cheap constraints
- Every save operation for cheap to expensive constraints
- Every generation operation for very expensive constraints

The screenshot shows an IDE interface with a code editor, an outline view, and a problems view.

**Code Editor:** The file `test.mydsl` contains the following code:

```
type String

entity Student{
    property alter:String[]
}

entity Student{
    property name:String[]
    property name:String[]
}
```

There are four errors highlighted in red:

- Entityname must be unique (line 3)
- Entityname must be unique (line 8)
- Propertyname must be unique (line 10)
- Propertyname must be unique (line 9)

**Outline View:** Shows the model structure:

- Model
  - String
  - Student
  - Student

**Problems View:** Displays the error log:

Description	Resource	Path	Location	Type
Errors (4 items)				
Entityname must be unique	test.mydsl	/test	line: 3 /test/test.mydsl	EMF Problem
Entityname must be unique	test.mydsl	/test	line: 8 /test/test.mydsl	EMF Problem
Propertyname must be unique	test.mydsl	/test	line: 10 /test/test.mydsl	EMF Problem
Propertyname must be unique	test.mydsl	/test	line: 9 /test/test.mydsl	EMF Problem

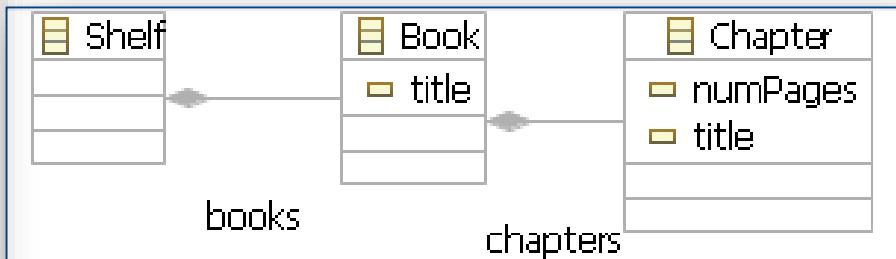
## Example #2: Bookshelf

- Edit „Bookshelf“ models in a text-based fashion
- **Given:** Example model as well as the metamodel
- **Asked:** Grammar, constraints, and editor for Bookshelf DSL

model.xml

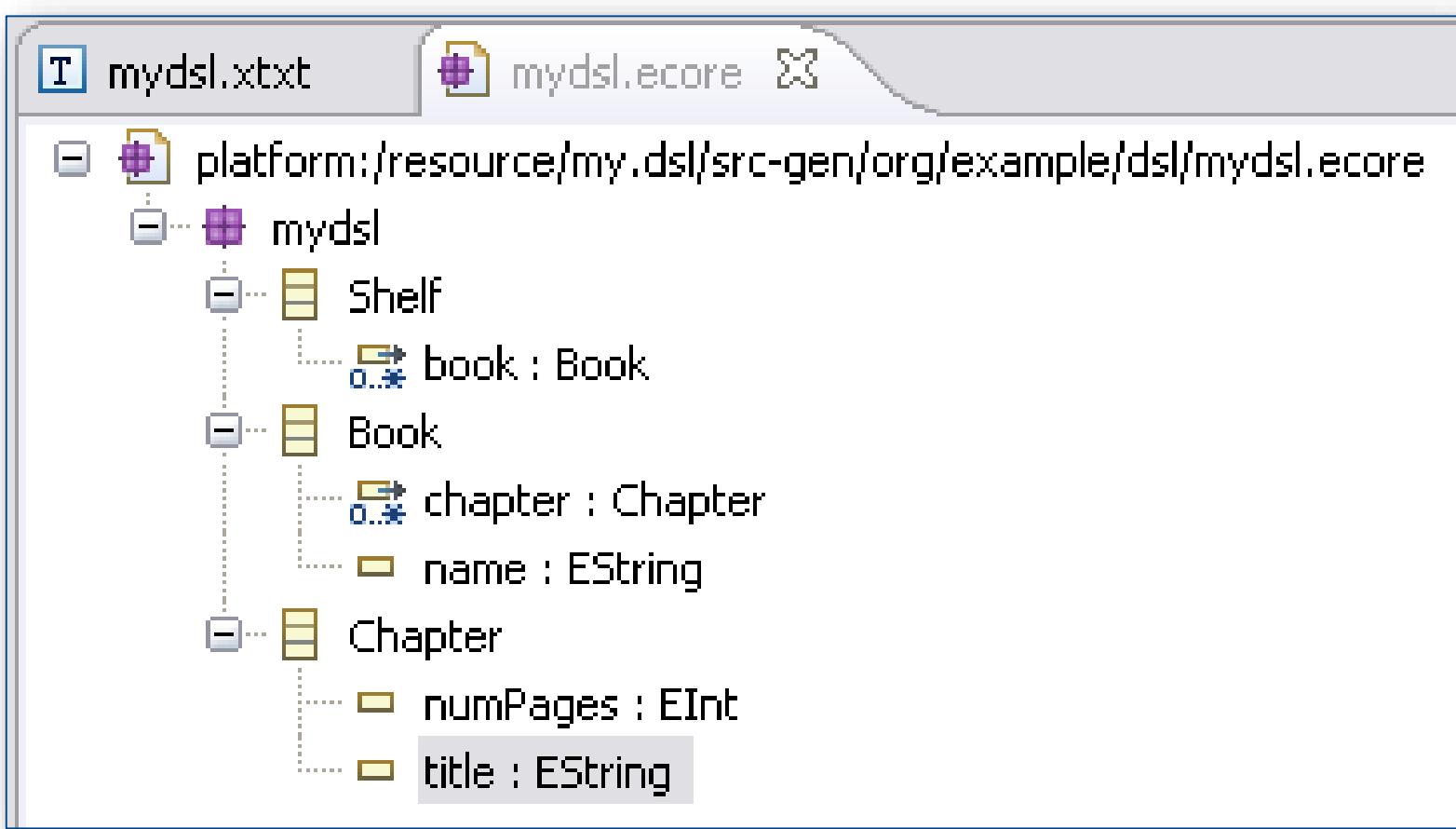
```

1<?xml version="1.0" encoding="ASCII"?>
2<bookshelf:Shelf xmi:version="2.0" xmlns:xmi="http://
3  <book name="LOTR">
4    <chapter title="chapter1" numPages="10"/>
5    <chapter title="chapter2" numPages="13"/>
6    <chapter title="chapter3" numPages="12"/>
7  </book>
8  <book name="RubyOnRailsInAction">
9    <chapter title="chapter1" numPages="10"/>
10 </book>
11 <book name="WickedCoolJava">
12   <chapter title="chapter1" numPages="13"/>
13   <chapter title="chapter2" numPages="11"/>
14 </book>
15</bookshelf:Shelf>
16
  
```



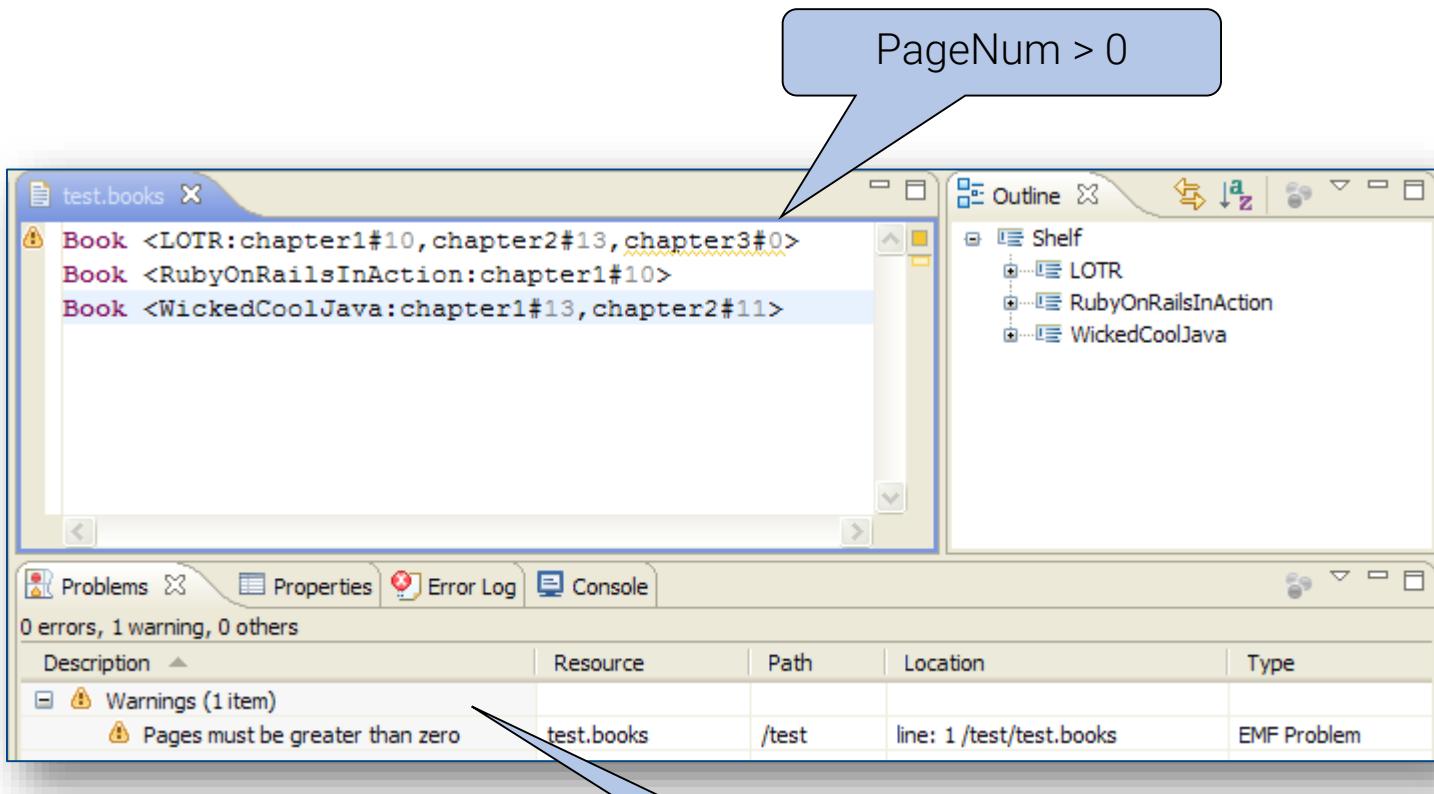


## Example #2: Metamodel Details





## Example #2: Editor





# CHAPTER 7 - Model-to-Model Transformations



# Overview

## 1 Introduction

- 1.1 Human Cognitive Processes
- 1.2 Models
- 1.3 Model Engineering

## 2 MDSE Principles

- 2.1 MDSE Basics
- 2.2 The MD\* Jungle of Acronyms
- 2.3 Modeling Languages and Metamodeling
- 2.4 Overview Considered Approaches
- 2.5 Tool Support
- 2.6 Criticisms of MDSE

## 3 MDSE Use Cases

- 3.1 MDSE applications
- 3.2 USE CASE1 – Model driven development
- 3.3 USE CASE2 – Systems interoperability
- 3.4 USE CASE3 – Model driven reverse engineering



# Overview

## 4 Model-Driven Architecture (MDA)

- 4.1 MDA Definitions and Assumptions
- 4.2 The Modeling Levels: CIM, PIM, PSM
- 4.3 Architecture-Driven Modernization

## 5 Modeling Languages at a Glance

- 5.1 Anatomy of Modeling Languages
- 5.2 General Purpose vs. Domain-Specific Modeling Languages
- 5.3 Overview of UML Diagrams
- 5.4 UML Behavioural or Dynamic Diagrams
- 5.5 UML Extensibility: The Middle Way Between GPL and DSL
- 5.6 Domain Specific Languages
- 5.7 Defining Modeling Constraints (OCL)



# Overview

## 6 Developing your Own Modeling Language

- 6.1 Metamodel-Centric Language Design
- 6.2 Programming Languages
- 6.3 Metamodel Development Process
- 6.4 MOF - Meta Object Facility
- 6.5 Example DSML: sWML
- 6.6 EMF and Ecore
- 6.7 Abstract Syntax Development
- 6.8 Graphical Concrete Syntax Development
- 6.9 Textual Concrete Syntax Development

## 7 Model-to-Model Transformations

- 7.1 Model Transformations and their Classification
- 7.2 Exogenous, Out-Place Transformations
- 7.3 Endogenous, In-Place Transformations
- 7.4 Mastering Model Transformations and their Rules

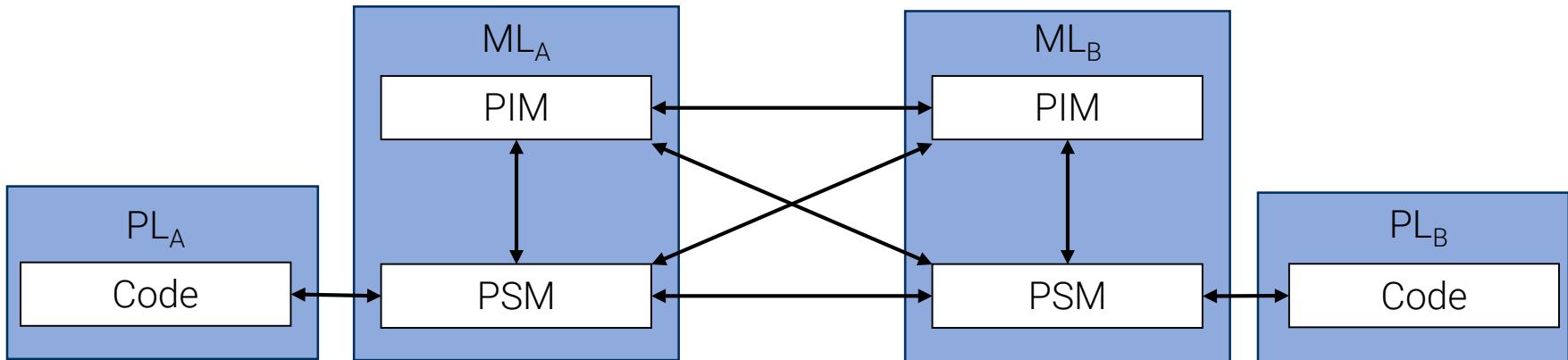
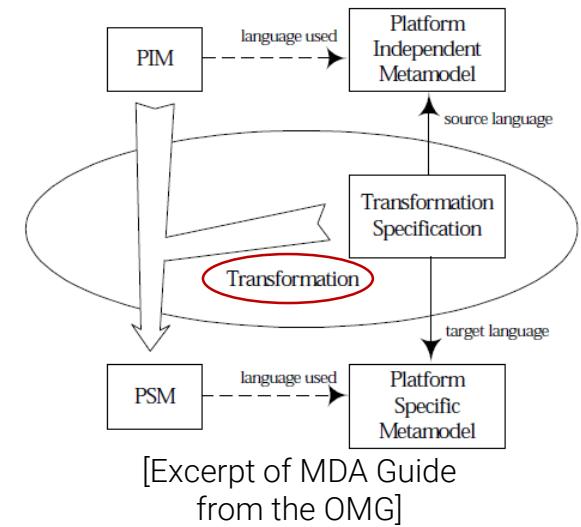


# 7.1 Model Transformations and their Classifications

# Motivation

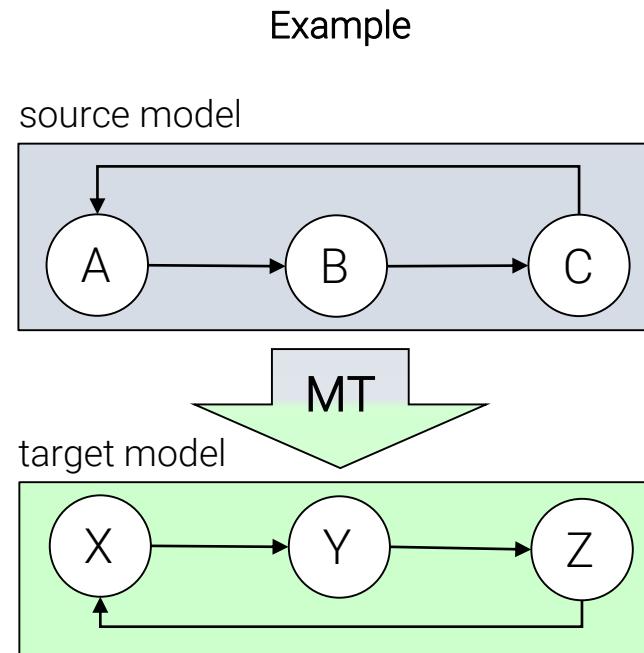
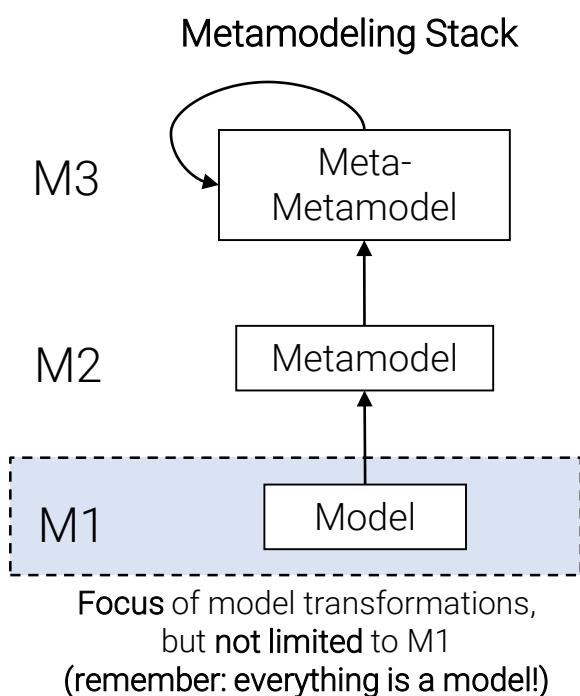
## Transformations are everywhere!

- Before MDE
  - » Program compilation, refactoring, migration, optimization, ...
- With MDE
  - » Transformations are key technologies!
  - » Every systematic manipulation of a model is a model transformation!
- Dimensions
  - » Horizontal vs. vertical
  - » Endogenous vs. exogenous
  - » Model-to-text vs. text-to-model vs. model-to-model



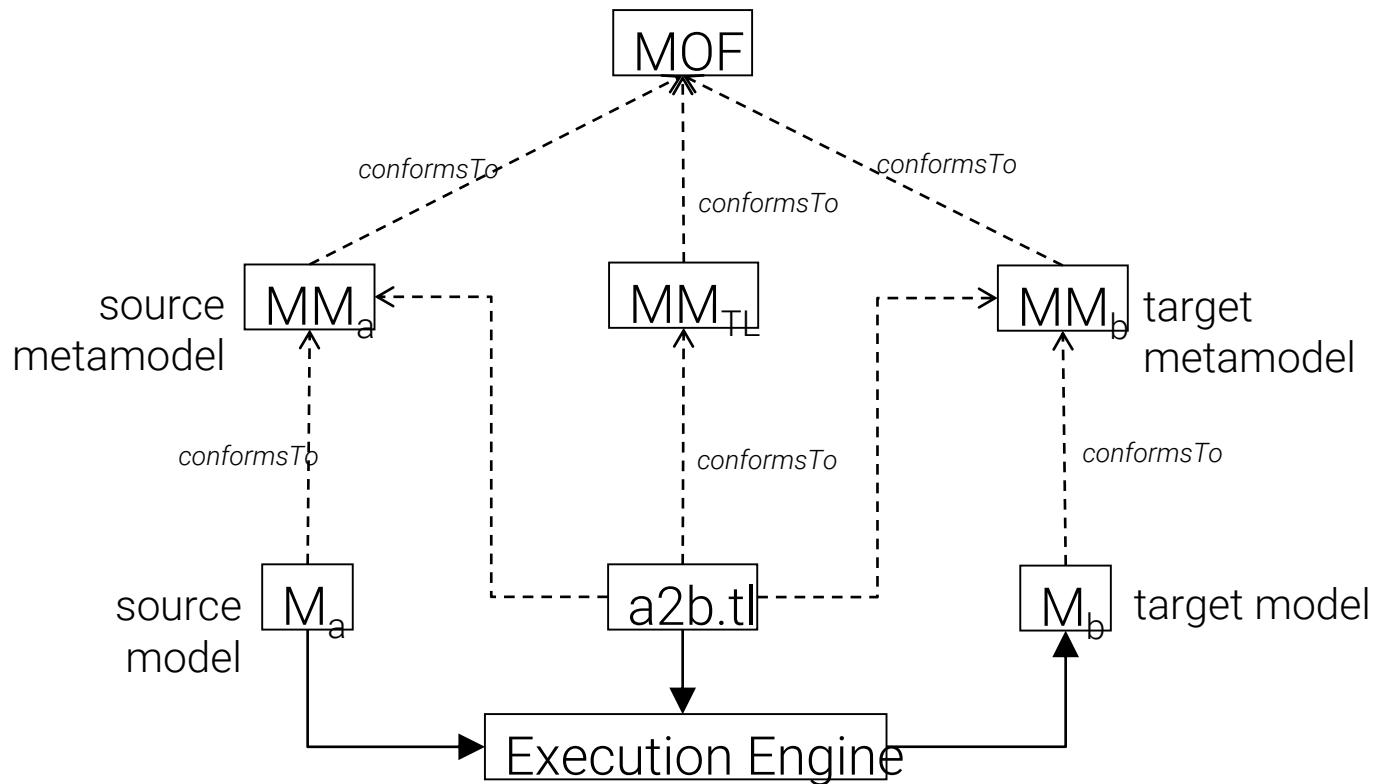
# Definitions

- A **model-to-model (M2M) transformation** is the automatic creation of target models from source models.
  - » 1-to-1 transformations
  - » 1-to-N, N-to-1, N-to-M transformations
  - » target model\* =  $T(\text{source model}^*)$



# Architecture

## Model-to-Model Transformation Pattern



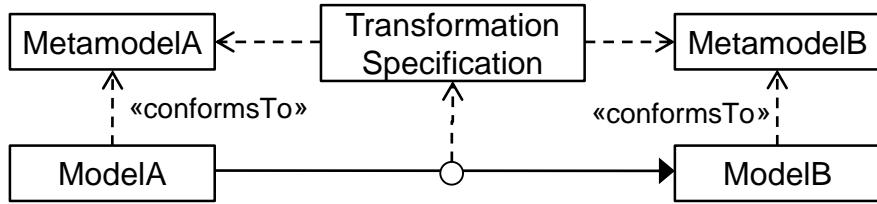
# Two Transformation Strategies

## Out-place vs. in-place transformations

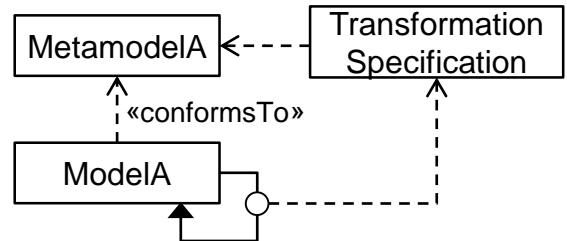
**Legend:**

- Transformation Execution
- - -> Dependency

Out-place Transformations build a new model from scratch



In-place Transformations change some parts in the model

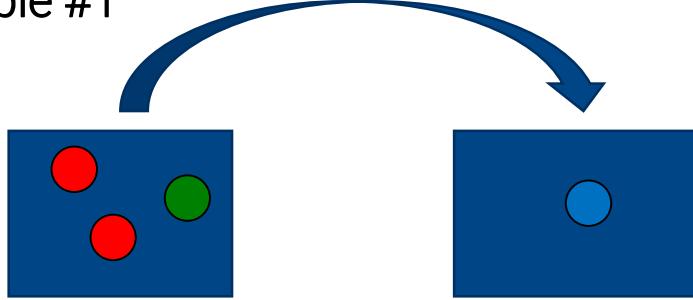




# Two Transformation Strategies

## Out-place vs. in-place transformations

Example #1



### Out-place Transformation

For each green element,  
create a blue element

### In-place Transformation

For each green element,  
create a blue element.

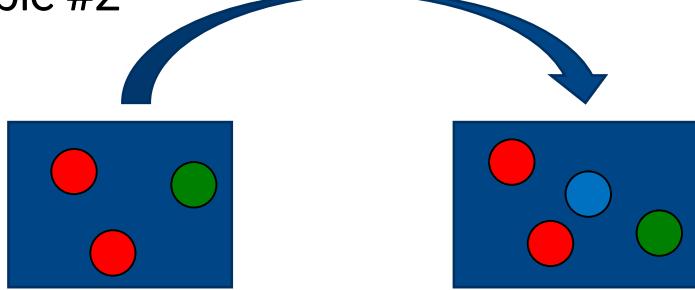
Delete all elements except blue  
ones



## Two Transformation Strategies

- Out-place vs. in-place transformations

Example #2



---

### Out-place Transformation

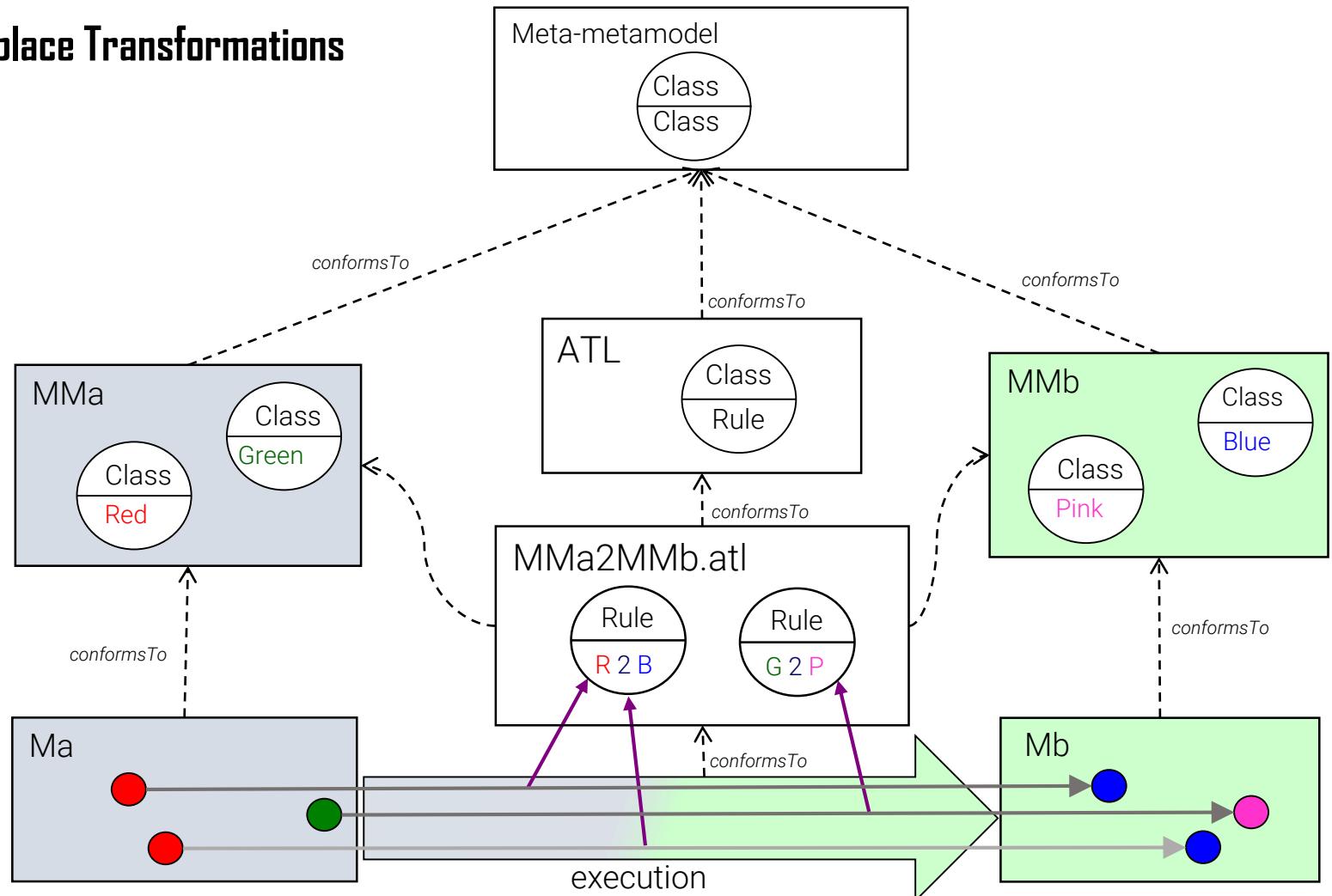
For each green element,  
create a blue element.  
For each green element,  
create a green element.  
For each red element, create  
a red element.

### In-place Transformation

For each green element,  
create a blue element.

# Architecture: Illustrative Example

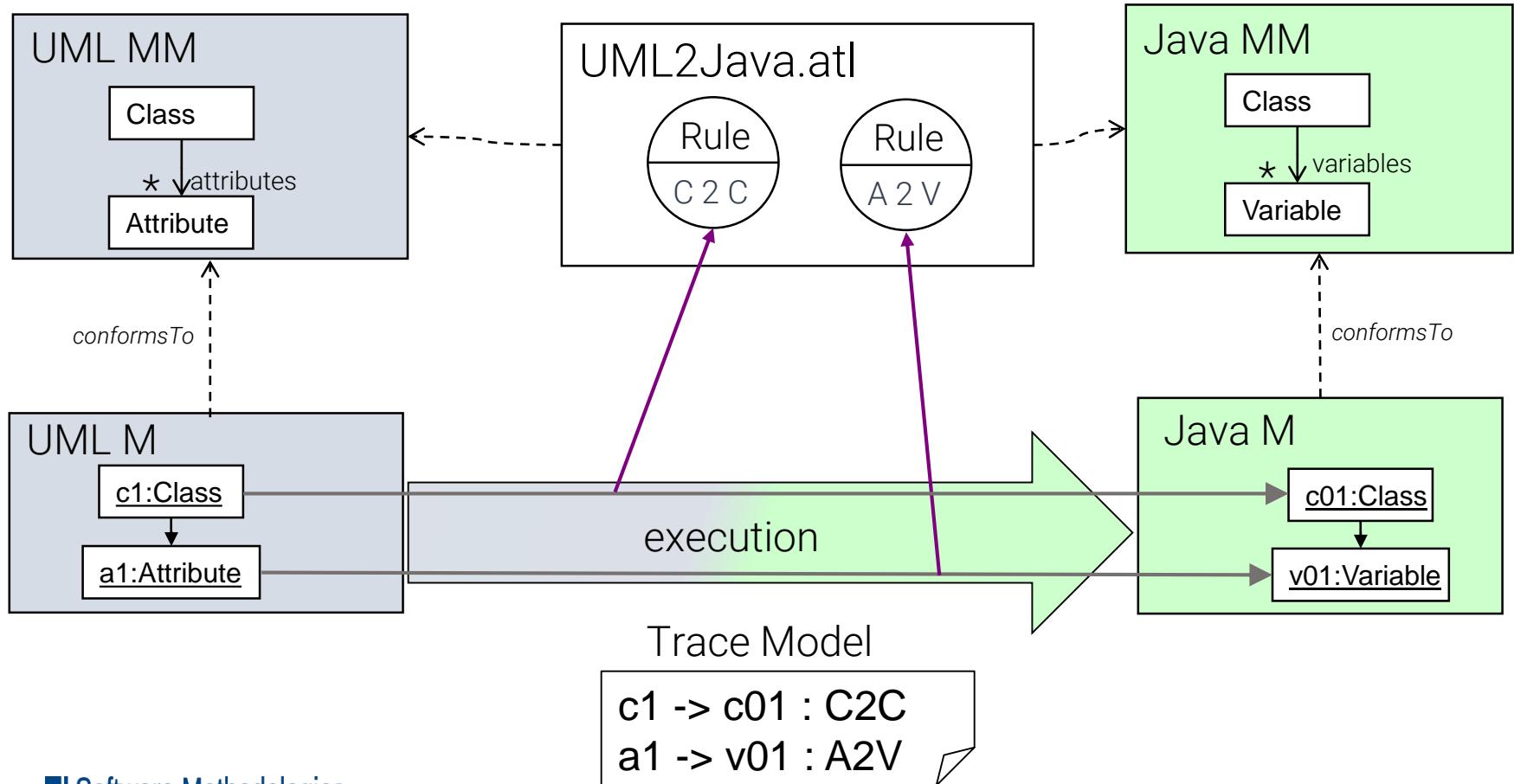
## Out-place Transformations





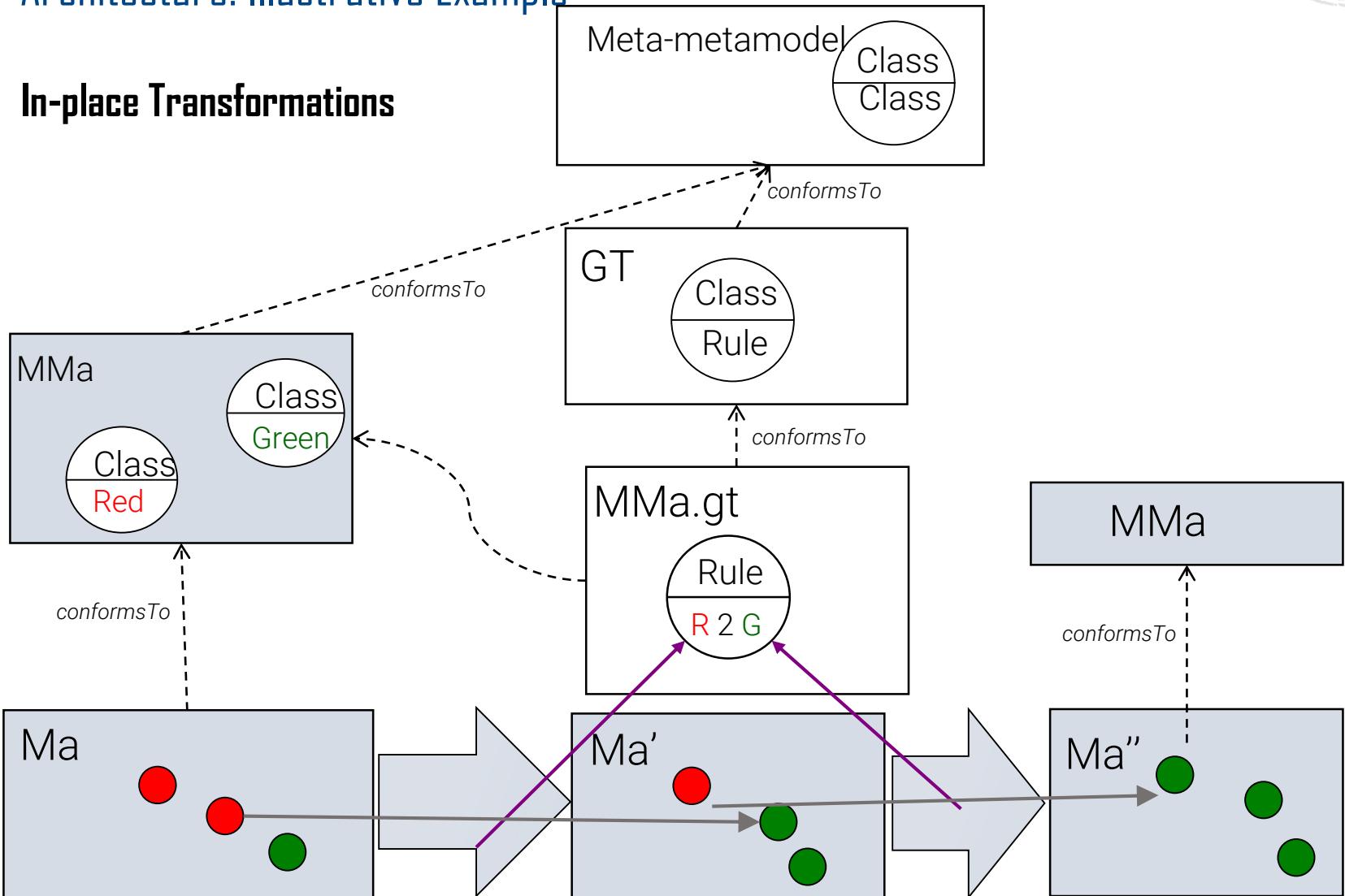
# Architecture: Concrete Example

## Out-place Transformations



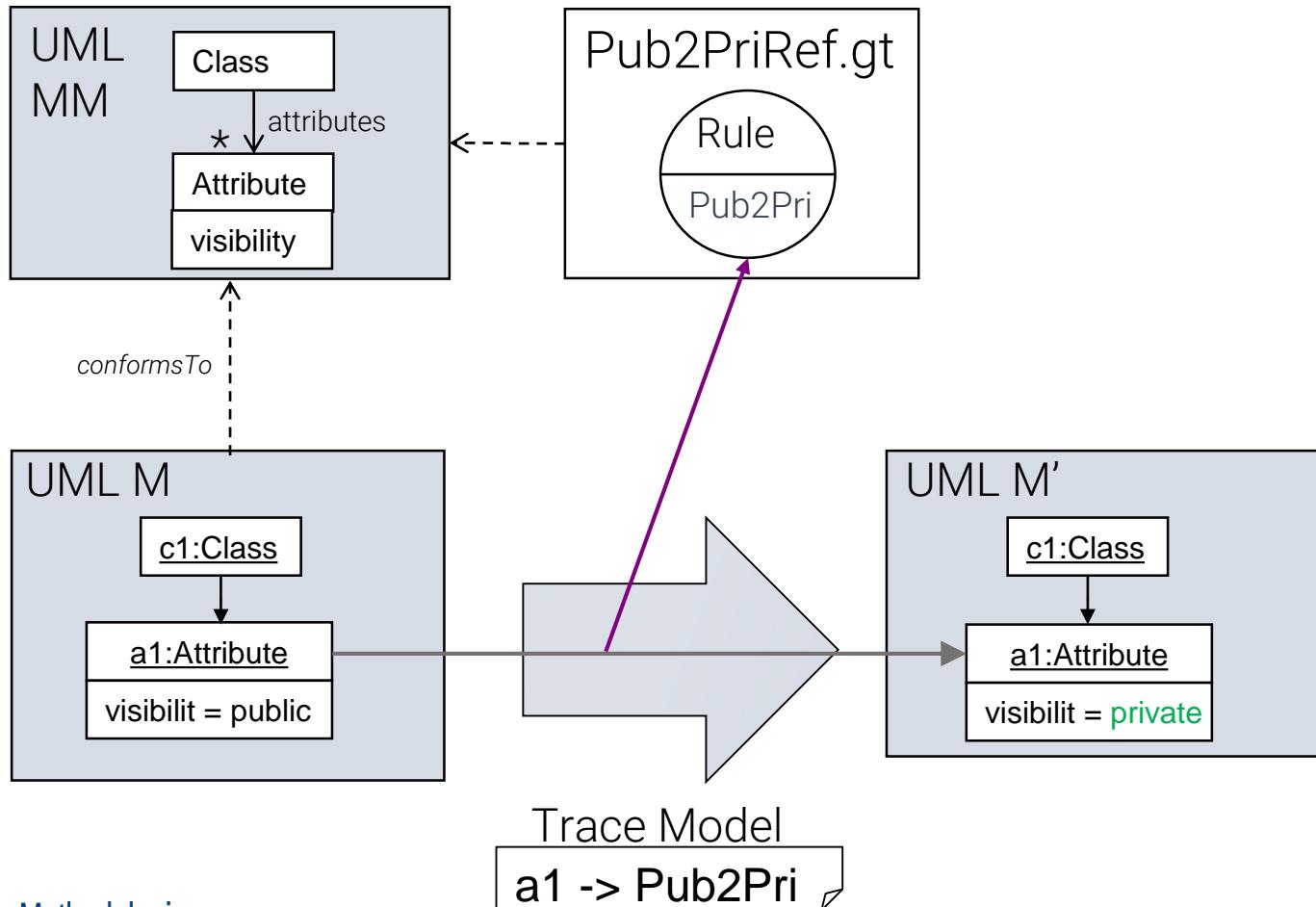
# Architecture: Illustrative Example

## In-place Transformations



# Architecture: Concrete Example

## In-place Transformations





## 7.2 Exogenous, Out-Place Transformations

Atlas transformation language



## ATL overview

- Source models and target models are distinct
  - » Source models are read-only
  - » Target models are write-only
- The language is a declarative-imperative hybrid
  - » Declarative part
    - Matched rules with automatic traceability support
    - Side-effect free query language: OCL
  - » Imperative part
    - Called/Lazy rules
    - Action blocks
    - Global variables via Helpers
- Recommended programming style: declarative



## ATL overview (continued)

- A **declarative rule specifies**
  - » A source pattern to be matched in the source models
  - » A target pattern to be created in the target models for each match during rule application
  - » An optional action block (i.e., a sequence of imperative statements)
  
- An **imperative rule is basically a procedure**
  - » It is called by its name
  - » It may take arguments
  - » It contains
    - A declarative target pattern
    - An optional action block

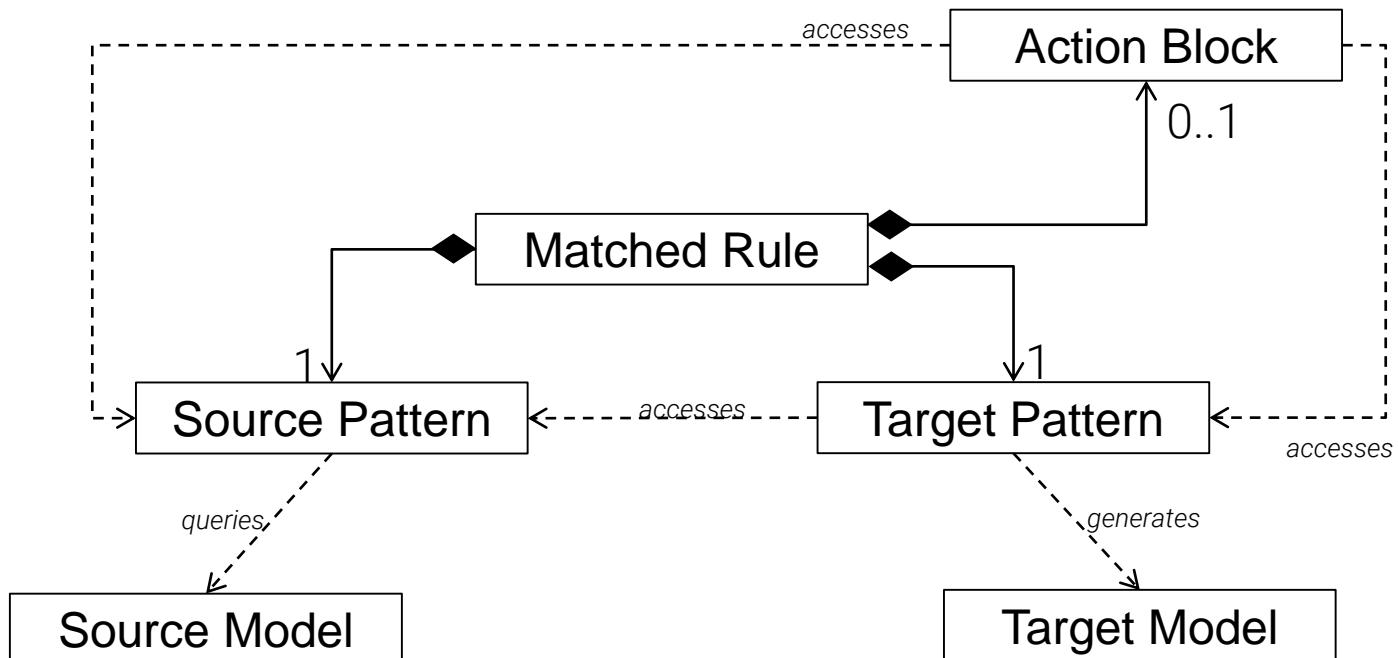


## ATL overview (continued)

- Applying a rule means
  - » Creating the specified target elements
  - » Initializing the properties of the newly created elements
- There are two types of rules concerning their application
  - » Matched rules are applied once for each match by the execution engine
    - A given set of elements may only be matched by one matched rule
  - » Called/Lazy rules are applied as many times as they are called from other rules

## Matched rules: Overview

- A Matched rule is composed of
  - » A source pattern
  - » A target pattern
  - » An optional action block





## Matched rules: source pattern

- The source pattern is composed of
  - » A set of labeled source pattern elements
  - » A source pattern element refers to a type contained in the source metamodels
  - » A guard (Boolean OCL expression) used to filter matches
- A match corresponds to a set of elements coming from the source models that
  - » Fulfill the types specified by the source pattern elements
  - » Satisfy the guard

```
rule Rule1{
    from
        v1 : SourceMM!Type1 (cond1)
    to
        v2 : TargetMM!Type1 (
            prop <- v1.prop
        )
}
```



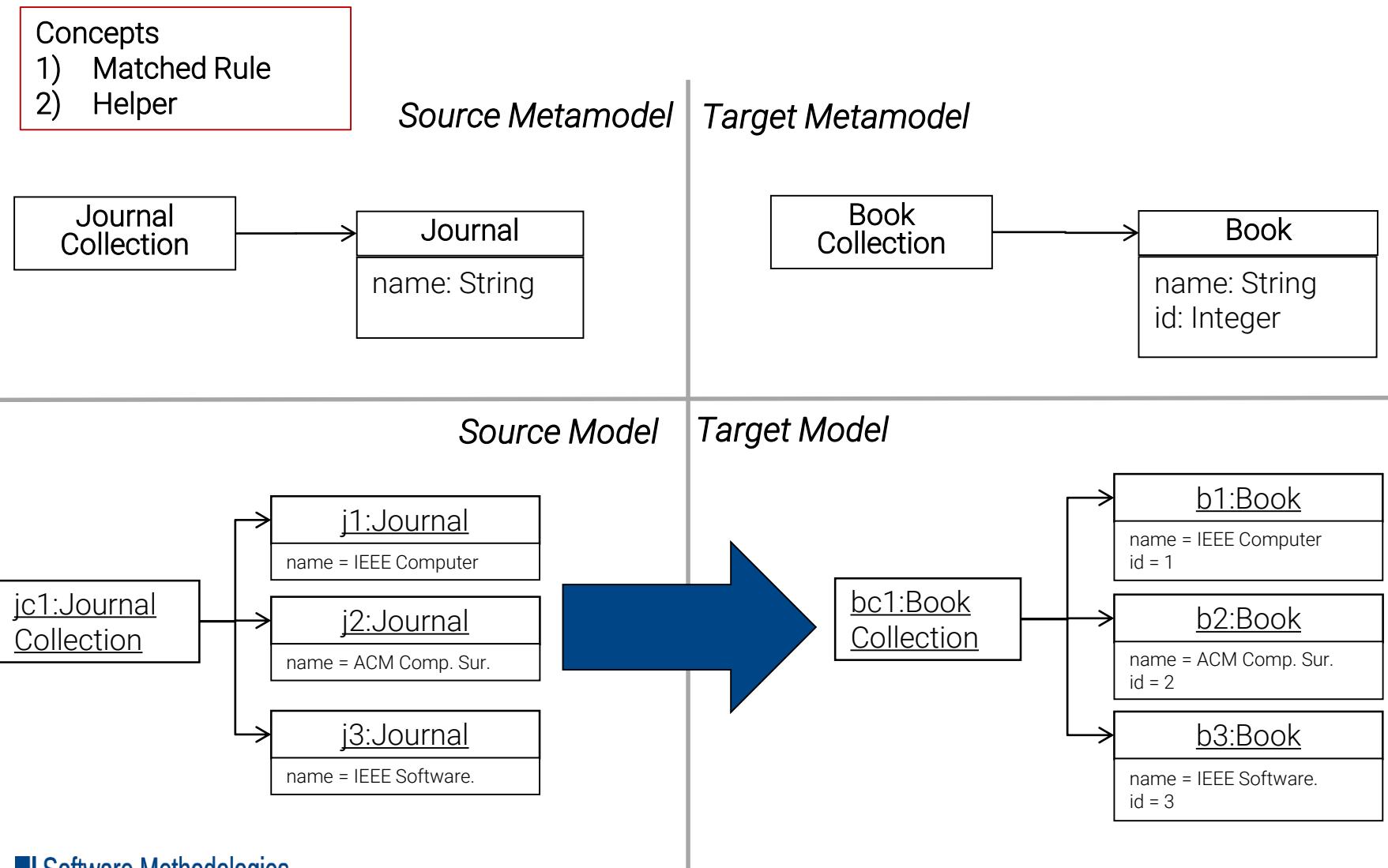
## Matched rules: target pattern

- The target pattern is composed of
  - » A set of labeled target pattern elements
  - » Each target pattern element
    - refers to a type in the target metamodels
    - contains a set of bindings
  - » A binding initializes a property of a target element using an OCL expression
- For each match, the target pattern is applied
  - » Elements are created in the target models
  - » Target elements are initialized by executing the bindings

```
rule Rule1{
    from
        v1 : SourceMM!Type1(cond1)
    to
        v2 : TargetMM!Type1 (
            prop <- v1.prop
        )
}
```



## Example #1 – Publication 2 Book





## Example #1

### Configuration of Source/Target Metamodels

- Header

```
module Publication2Book;  
create OUT : Book from IN : Publication;
```

- For code completion

- » --@path MM\_name =Path\_to\_metamodel\_definition
- » Activate code completion: Strg + space

## Example #1 Matched rules

- Example Publication 2 Book

```
module Publication2Book;
create OUT : Book from IN : Publication;

rule Collection2Collection {
    from jc : Publication!JournalCollection
    to bc : Book!BookCollection(
        books <- jc.journals
    )
}

rule Journal2Book {
    from j : Publication!Journal
    to b : Book!Book (
        name <- j.name
    )
}
```

Matched Rule

Header

Source Pattern

Target Pattern

Binding



## Example #1 Helpers

- Syntax

```
helper context Type def : Name(Par1 : Type, ...) :  
Type = EXP;
```

- Global Variable

```
helper def: id : Integer = 0;
```

- Global Operation

```
helper context Integer def : inc() : Integer = self  
+ 1;
```

- Calling a Helper

- » `thisModule.HelperName(...)` – for global variables/operations without context
- » `value.HelperName(...)` – for global variables/operations with context



## Example #1 Helpers

- Example Publication 2 Book

```
module Publication2Book;
create OUT : Book from IN : Publication;

helper def : id : Integer = 0;           ← Global Variable
helper context Integer def : inc() : Integer = self + 1;

rule Journal2Book {
    from
        j : Publication!Journal
    to
        b : Book!Book (
            name <- j.name
    )
    do {
        thisModule.id <- thisModule.id.inc();
        b.id <- thisModule.id;           ← Global Operation call
    }
}
```

Action Block

Global Operation

Global Operation call

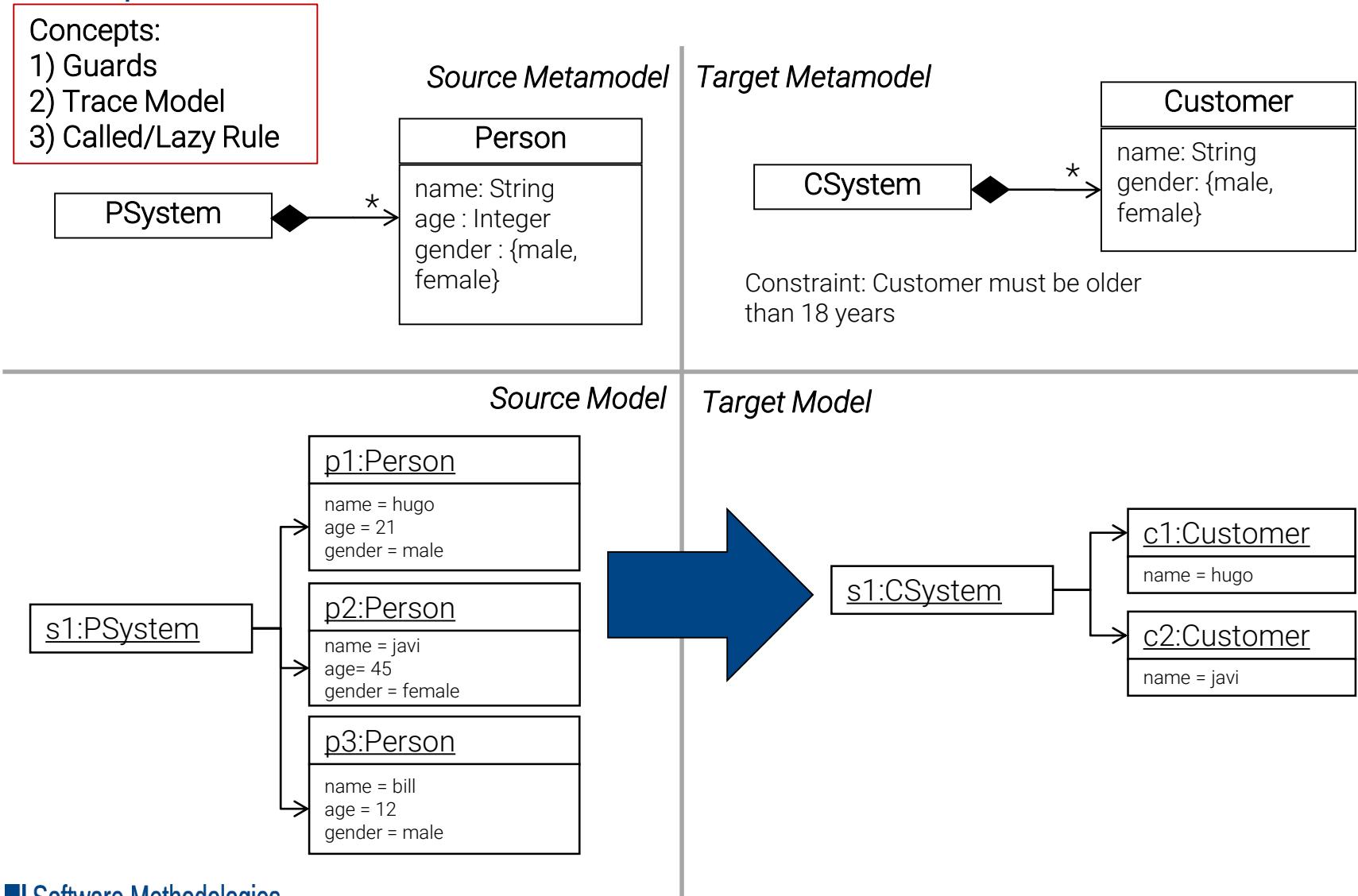
Module Instance for accessing global variables



## Example #2 – Person 2 Customer

Concepts:

- 1) Guards
- 2) Trace Model
- 3) Called/Lazy Rule





## Example #2

### Describing more complex correspondences and bindings

- Declarative Statements

- » If/Else, OCL Operations, Global Operations, ...
- » Application: Guard Condition and Feature Binding

- Example: IF/ELSE

```
if condition then  
    exp1  
else  
    exp2  
endif
```



## Example #2 Guards Conditions in Source Patterns (1/2)

- Example Person2Customer

```
rule Person2Customer {
    from
        p : Person!Person (p.age > 18)
    to
        c : Customer!Customer (
            name <- p.name
        )
}
```

Guard Condition



```
rule PSystem2CSYSTEM {
    from
        ps : Person!PSystem
    to
        cs : Customer!CSYSTEM (
            customer <- ps.person -> select(p | p.age > 18)
        )
}
```

Compute Subset for Feature Binding





## Example #2 Guards Conditions in Source Patterns (2/2)

- Example Person2Customer

```
rule Person2Customer {
    from
        p : Person!Person (p.age > 18)
    to
        c : Customer!Customer (
            name <- p.name
        )
}
```

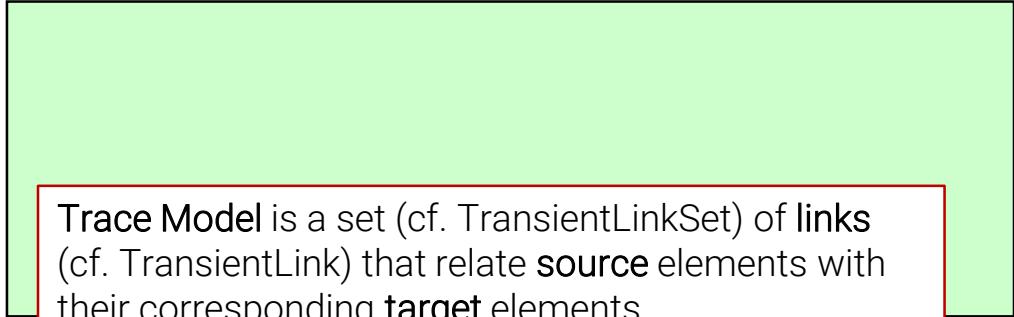
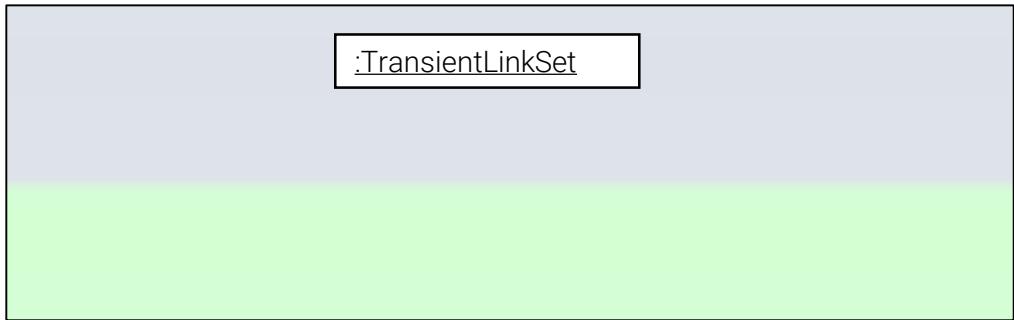
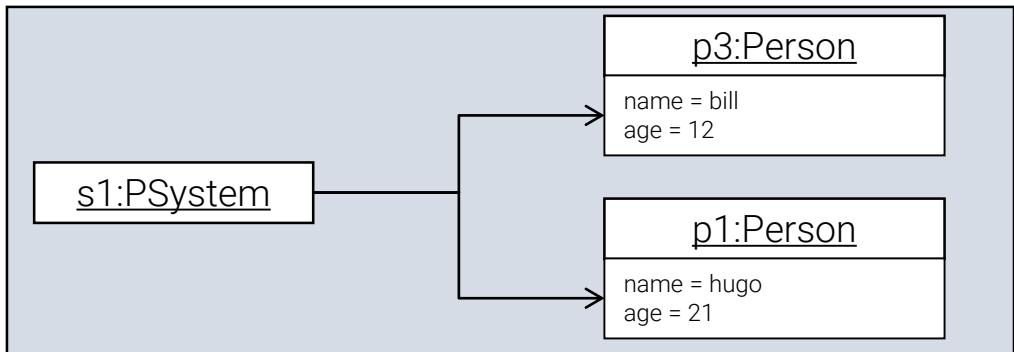
```
rule PSystem2CSYSTEM {
    from
        ps : Person!PSystem
    to
        cs : Customer!CSYSTEM (
            customer <- ps.person
        )
}
```

Subset for Binding is computed by ATL Engine ☺



## Example #2

### Implicit Trace Model – Phase I: Module Initialization Phase



```

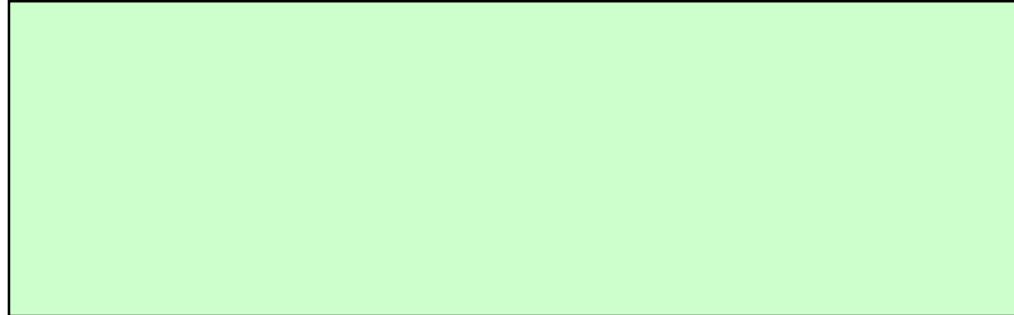
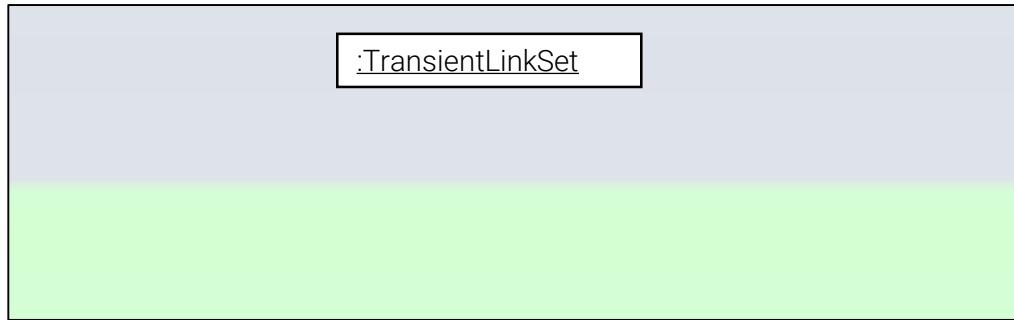
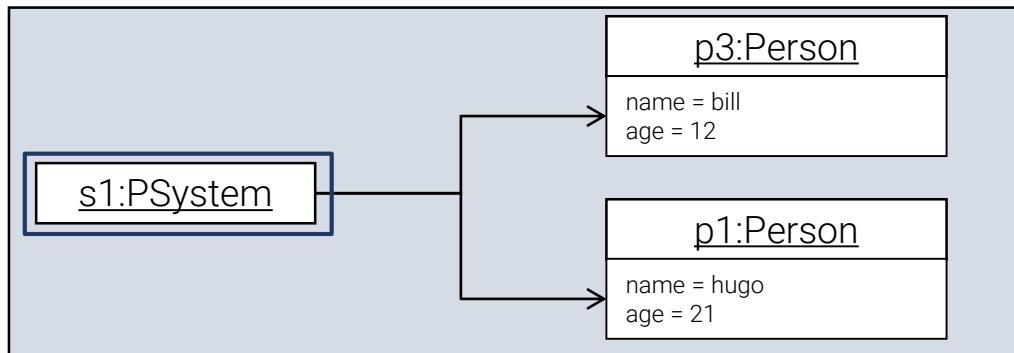
rule PSystem2CSystem {
  from
    ps : Person!PSystem
  to
    cs : Customer!CSystem (
      customer <- ps.person
    )
}

rule Person2Customer {
  from
    p : Person!Person(p.age > 18)
  to
    c : Customer!Customer (
      name <- p.name
    )
}
  
```



## Example #2

### Implicit Trace Model – Phase 2: Matching Phase



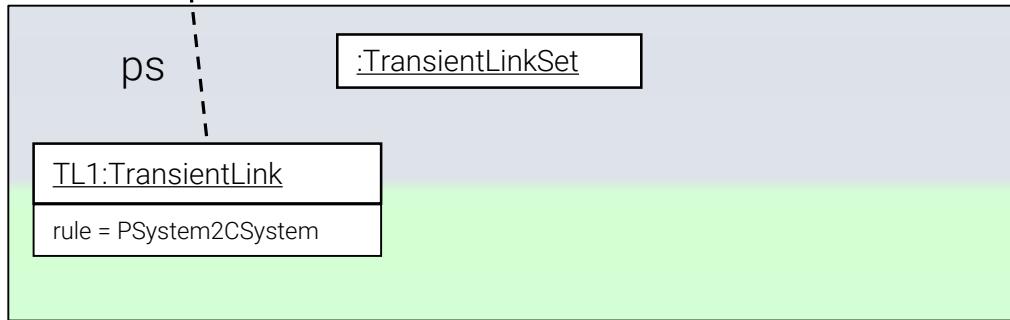
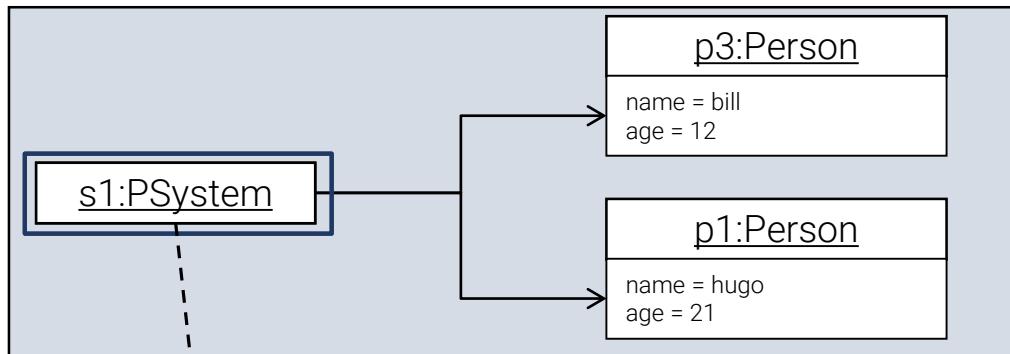
```
rule PSystem2CSystem {
    from
        ps : Person!PSystem
    to
        cs : Customer!CSystem (
            customer <- ps.person
        )
}

rule Person2Customer {
    from
        p : Person!Person(p.age > 18)
    to
        c : Customer!Customer (
            name <- p.name
        )
}
```



## Example #2

### Implicit Trace Model – Phase 2: Matching Phase



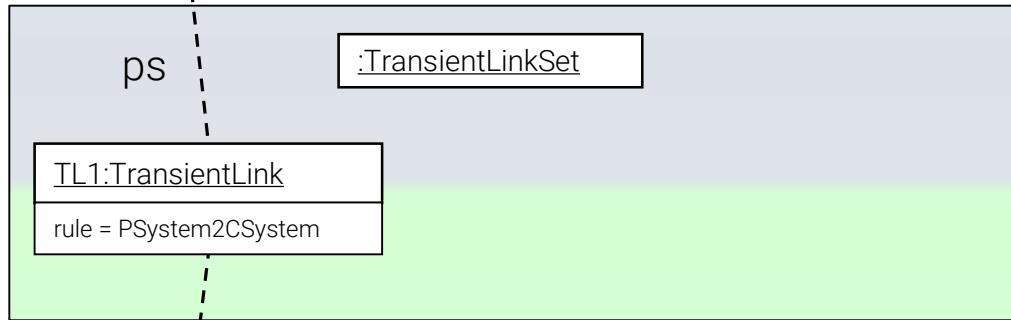
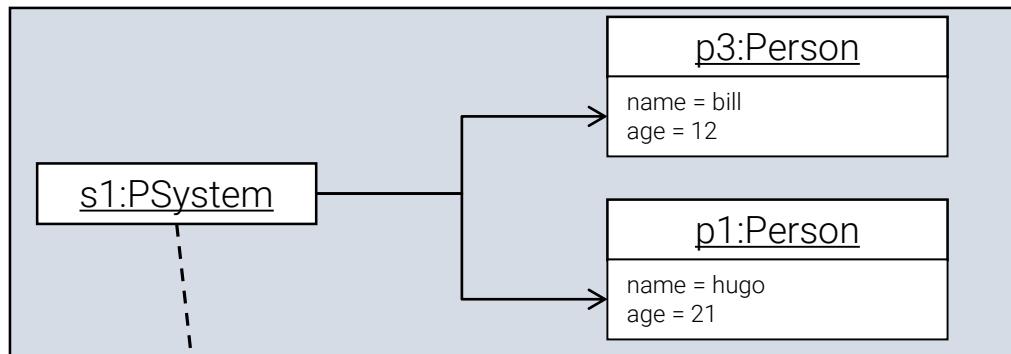
```
rule PSystem2CSystem {
    from
        ps : Person!Psystem
    to
        cs : Customer!CSystem (
            customer <- ps.person
        )
}
```

```
rule Person2Customer {
    from
        p : Person!Person(p.age > 18)
    to
        c : Customer!Customer (
            name <- p.name
        )
}
```



## Example #2

### Implicit Trace Model – Phase 2: Matching Phase



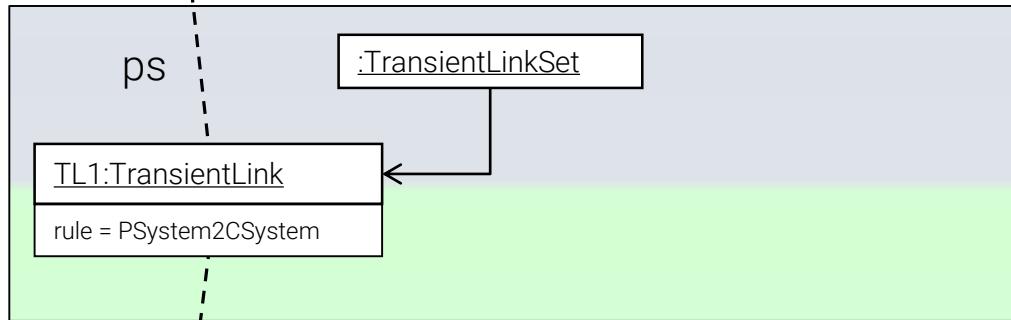
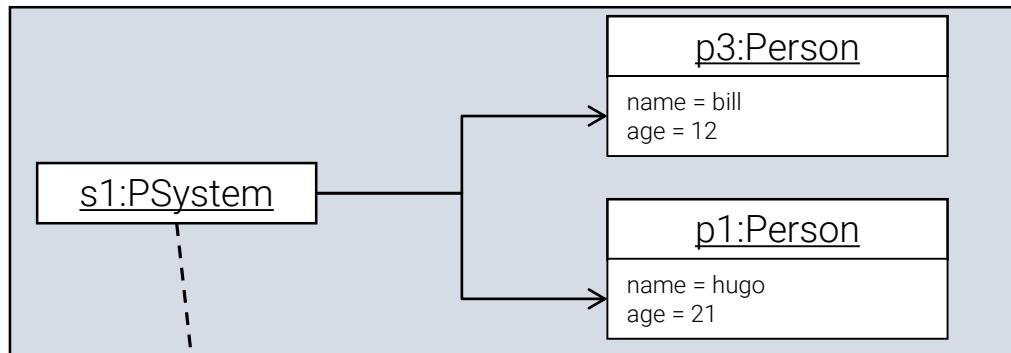
```
rule PSystem2CSystem {
    from
        ps : Person!Psystem
    to
        cs : Customer!CSystem (
            customer <- ps.person
        )
}

rule Person2Customer {
    from
        p : Person!Person(p.age > 18)
    to
        c : Customer!Customer (
            name <- p.name
        )
}
```



## Example #2

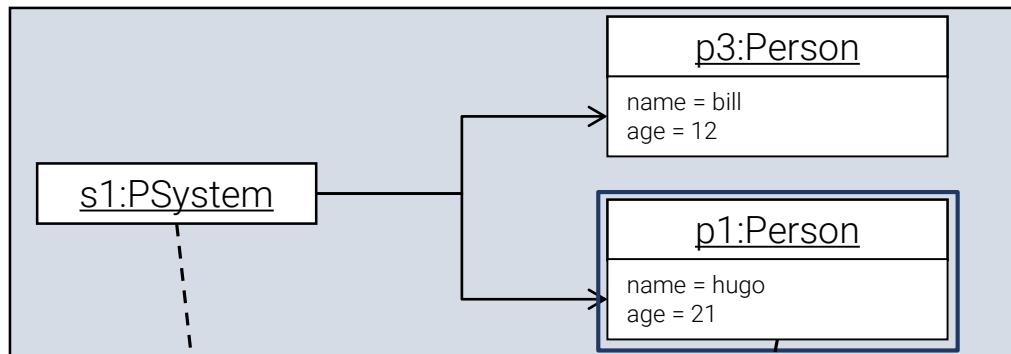
### Implicit Trace Model – Phase 2: Matching Phase



```
rule PSystem2CSystem {
    from
        ps : Person!Psystem
    to
        cs : Customer!CSystem (
            customer <- ps.person
        )
}

rule Person2Customer {
    from
        p : Person!Person(p.age > 18)
    to
        c : Customer!Customer (
            name <- p.name
        )
}
```

## Example #2 Implicit Trace Model – Phase 2: Matching Phase



```

rule PSysystem2CSystem {
  from
    ps : Person!Psystem
  to
    cs : Customer!CSystem (
      customer <- ps.person
    )
}

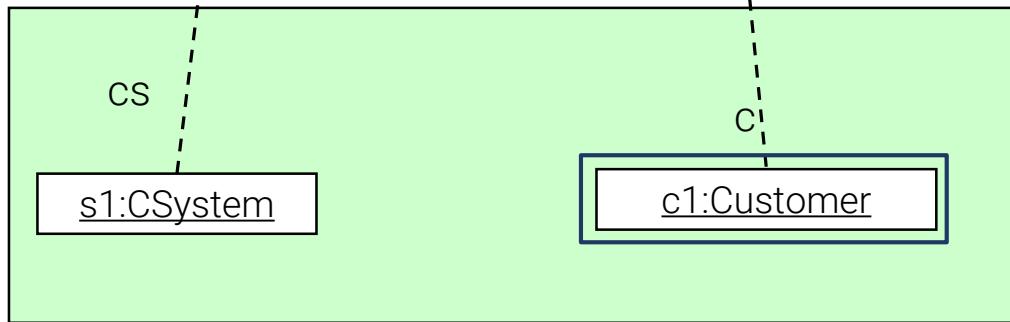
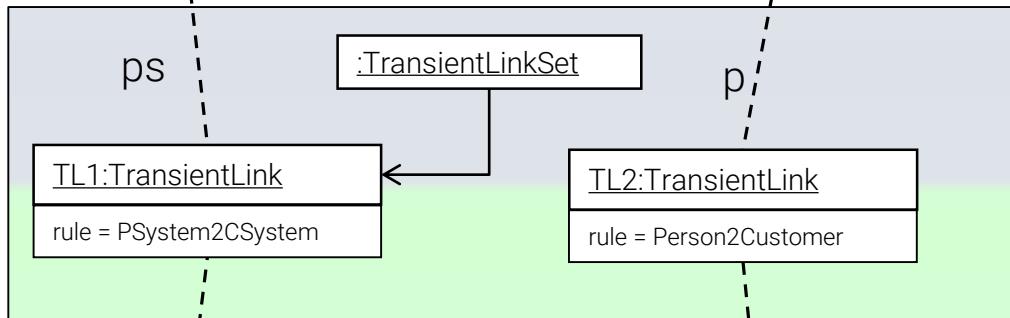
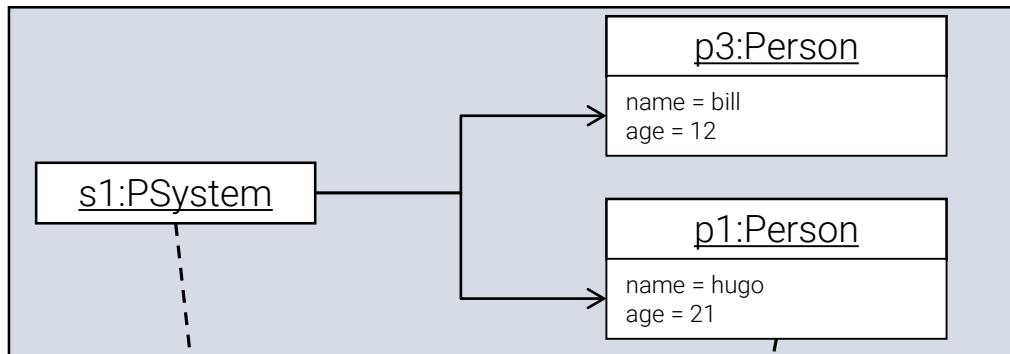
```

```

rule Person2Customer {
  from
    p : Person!Person(p.age > 18)
  to
    c : Customer!Customer (
      name <- p.name
    )
}

```

## Example #2 Implicit Trace Model – Phase 2: Matching Phase



```

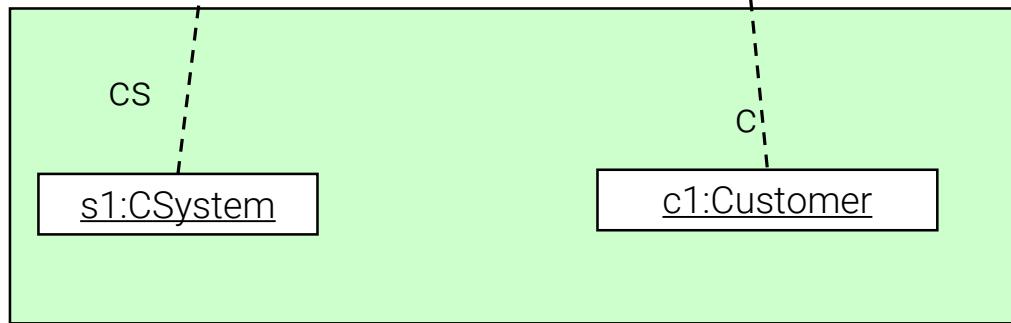
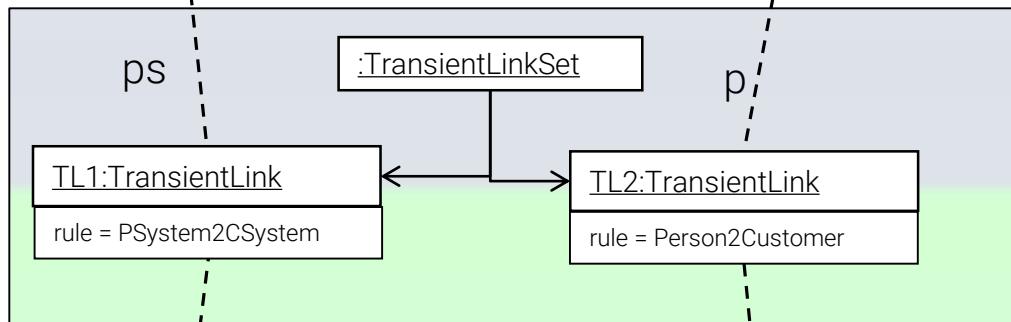
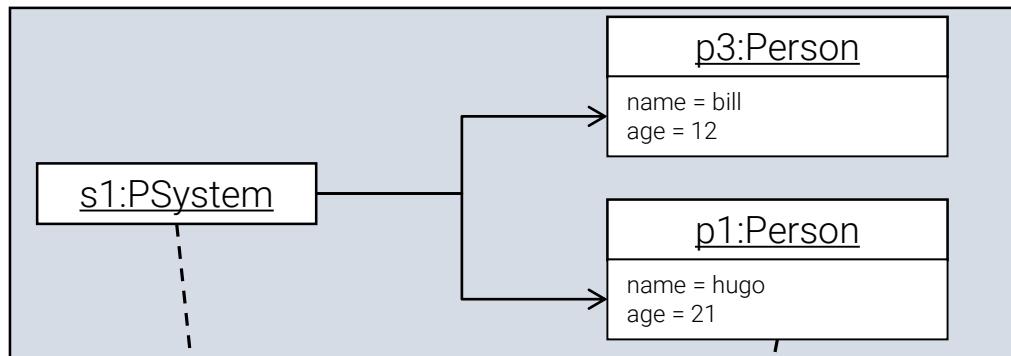
rule PSystem2CSystem {
  from
    ps : Person!Psystem
  to
    cs : Customer!CSystem (
      customer <- ps.person
    )
}

rule Person2Customer {
  from
    p : Person!Person(p.age > 18)
  to
    c : Customer!Customer (
      name <- p.name
    )
}

```



## Example #2 Implicit Trace Model – Phase 2: Matching Phase



```

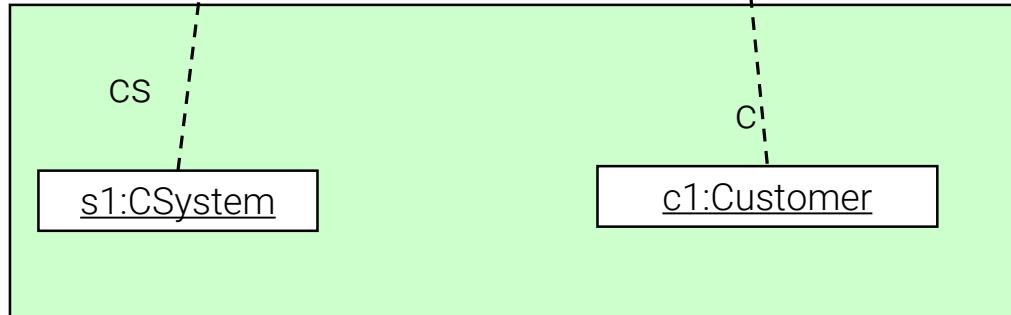
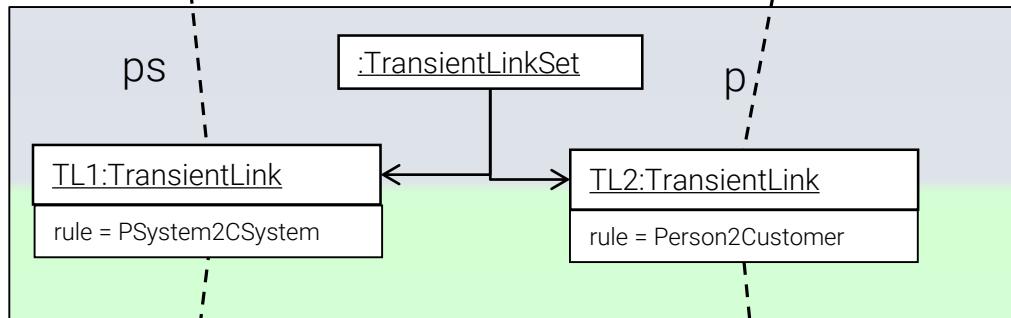
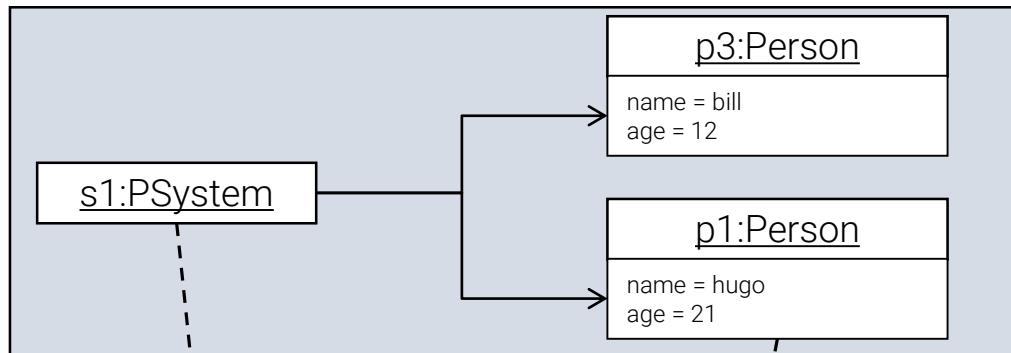
rule PSystem2CSystem {
  from
    ps : Person!Psystem
  to
    cs : Customer!CSystem (
      customer <- ps.person
    )
}

rule Person2Customer {
  from
    p : Person!Person(p.age > 18)
  to
    c : Customer!Customer (
      name <- p.name
    )
}

```

## Example #2

### Implicit Trace Model – Phase 3: Target Initialization Phase



```

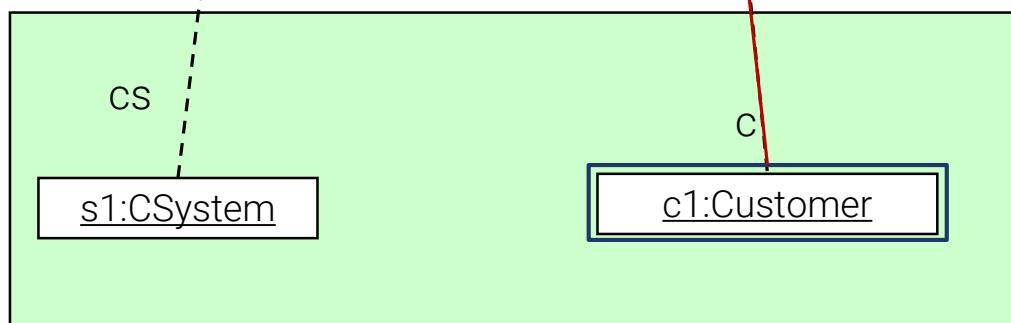
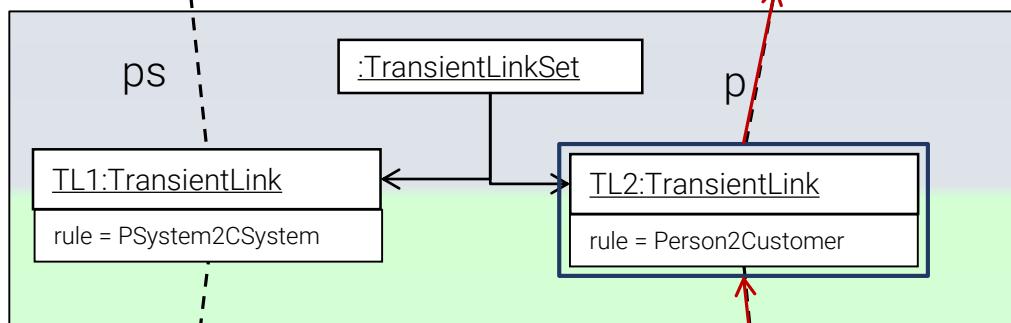
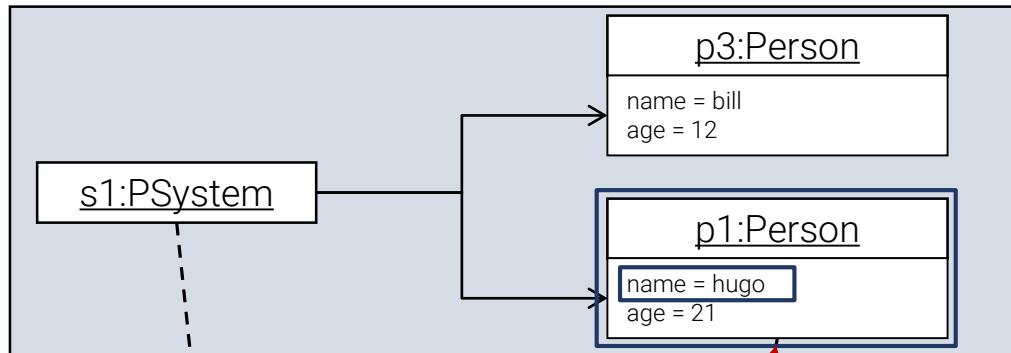
rule PSystem2CSysystem {
  from
    ps : Person!Psystem
  to
    cs : Customer!CSysystem (
      customer <- ps.person
    )
}

rule Person2Customer {
  from
    p : Person!Person(p.age > 18)
  to
    c : Customer!Customer (
      name <- p.name
    )
}
  
```



## Example #2

### Implicit Trace Model – Phase 3: Target Initialization Phase



```

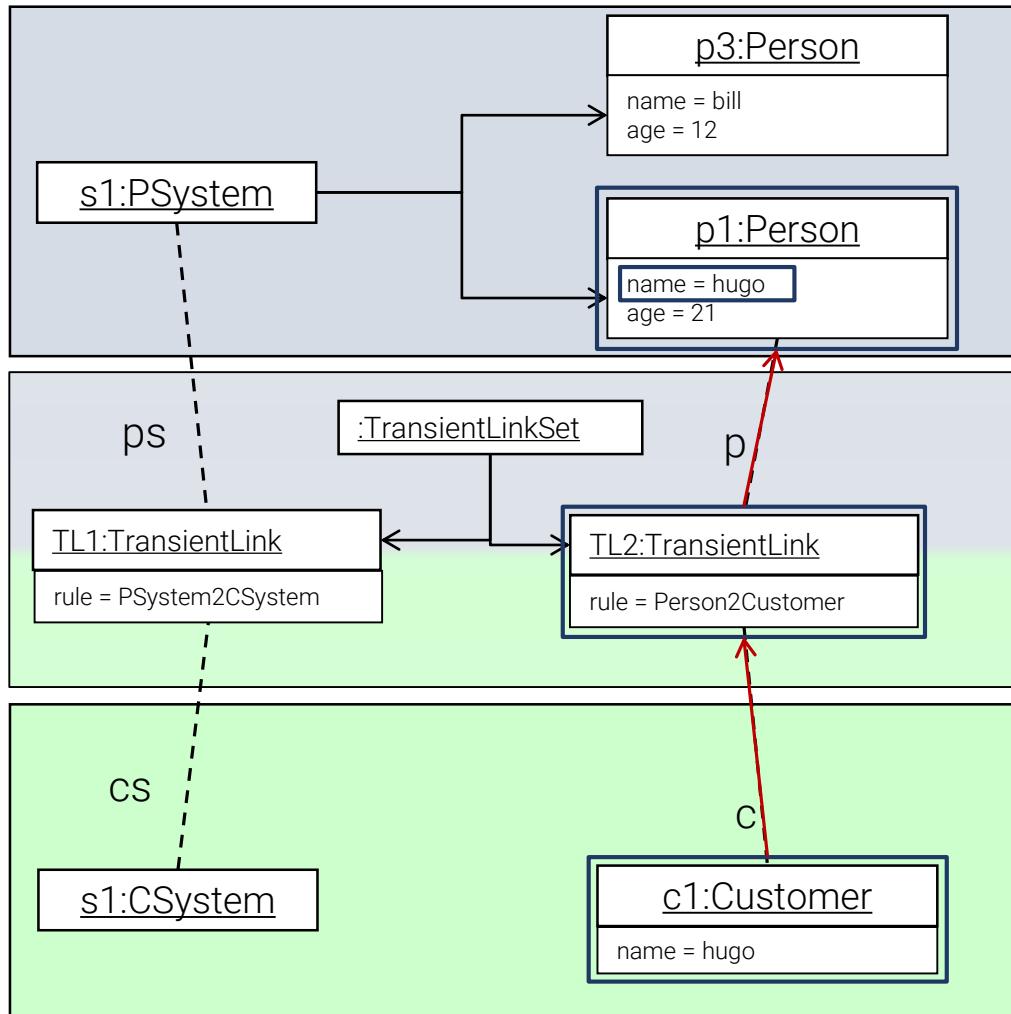
rule PSystem2CSystem {
  from
    ps : Person!Psystem
  to
    cs : Customer!CSystem (
      customer <- ps.person
    )
}

rule Person2Customer {
  from
    p : Person!Person(p.age > 18)
  to
    c : Customer!Customer (
      name <- p.name
    )
}

```

## Example #2

### Implicit Trace Model – Phase 3: Target Initialization Phase



```

rule PSystem2CSys {
  from
    ps : Person!Psystem
  to
    cs : Customer!CSystem (
      customer <- ps.person
    )
}

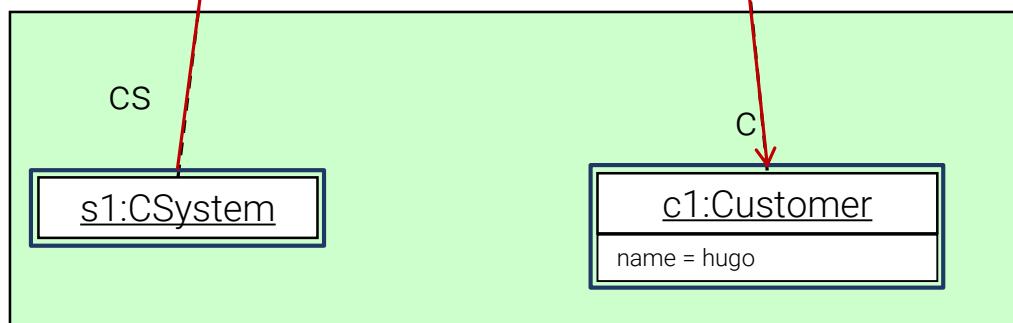
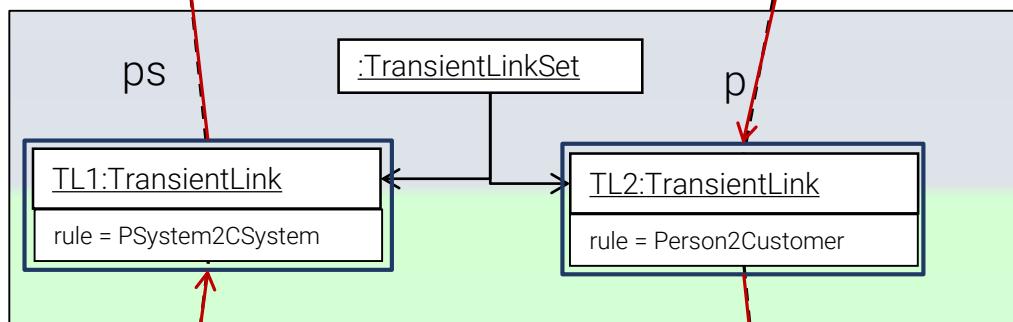
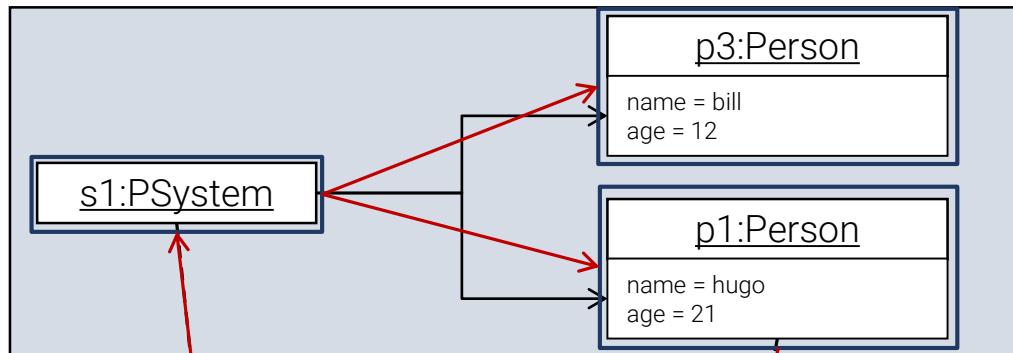
rule Person2Customer {
  from
    p : Person!Person(p.age > 18)
  to
    c : Customer!Customer (
      name <- p.name
    )
}

```



## Example #2

### Implicit Trace Model – Phase 3: Target Initialization Phase



```

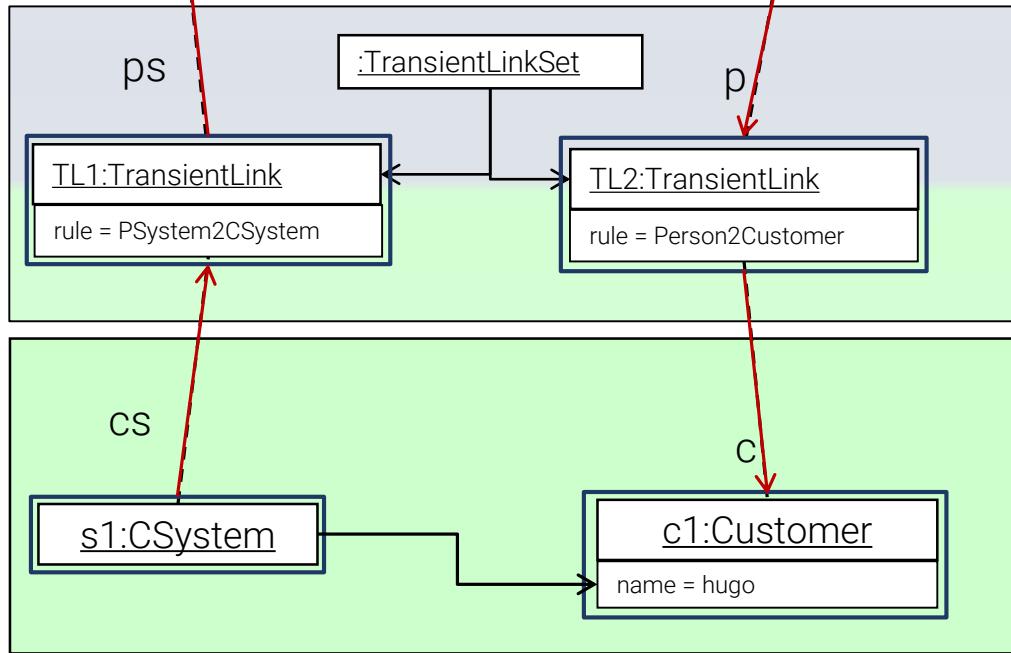
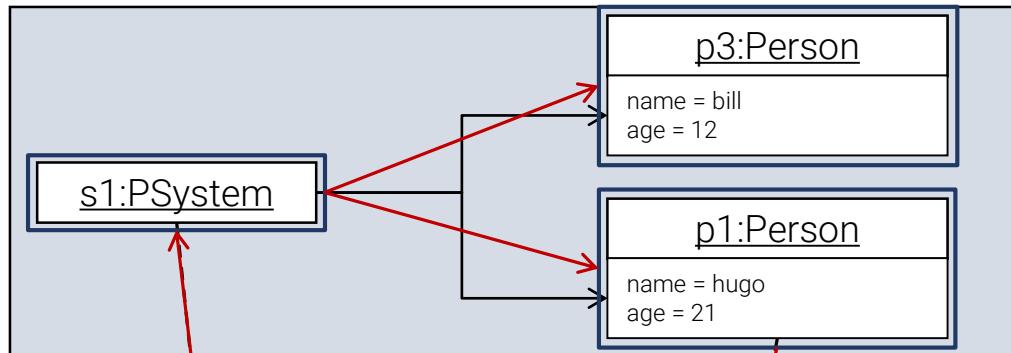
rule PSystem2CSystem {
  from
    ps : Person!Psystem
  to
    cs : Customer!CSystem (
      customer <- ps.person
    )
}

rule Person2Customer {
  from
    p : Person!Person(p.age > 18)
  to
    c : Customer!Customer (
      name <- p.name
    )
}

```

## Example #2

### Implicit Trace Model – Phase 3: Target Initialization Phase



```

rule PSystem2CSystem {
  from
    ps : Person!Psystem
  to
    cs : Customer!CSystem (
      customer <- ps.person
    )
}

rule Person2Customer {
  from
    p : Person!Person(p.age > 18)
  to
    c : Customer!Customer (
      name <- p.name
    )
}

```



# Transformation Execution Phases

## ■ **Module Initialization Phase**

- » Module variables (attribute helpers) and trace model are initialized
- » If an entry point called rule is defined, it is executed in this step

## ■ **Matching Phase**

- » Using the source patterns (**from**) of matched rules, elements are selected in the source model (each match has to be **unique**)
- » Via the target patterns (**to**) corresponding elements are created in the target model (for each match there are as much target elements created as target patterns are used)
- » Traceability information is stored

## ■ **Target Initialization Phase**

- » The elements in the target model are initialized based on the bindings (<-)
- » The resolveTemp function is evaluated, based on the traceability information
- » Imperative code (**do**) is executed, including possible calls to called rules



## Example #2

### Alternative Solution with Called Rule and Action Block (1/3)

- Imperative Statements in Action Blocks (`do`)

- IF/[ELSE]

```
if( aPerson.gender = #male )
    thisModule.men->including(aPerson);
else
    thisModule.women->including(aPerson);
```

- FOR

```
for( p in Person!Person.allInstances() ) {
    if(p.gender = #male)
        thisModule.men->including(p);
    else
        thisModule.women->including(p);
}
```



## Example #2

### Alternative Solution with Called Rule and Action Block (2/3)

```
rule PSystem2CSys { ← Matched Rule
  from
    ps : Person!Psystem
  to
    cs : Customer!CSys (
    )
  do {
    for( p in Person!Person.allInstances() ) {
      if(p.age > 18)
        cs.customer <- thisModule.NewCustomer(p.name,
                                                p.gender);
    }
  }
}
```

Annotations:

- Action Block**: Points to the **do** block.
- Explicit Query on Source Model**: Points to the **for** loop.
- Binding**: Points to the variable **p** in the **for** loop.
- Called Rule Call**: Points to the **NewCustomer** call.



## Example #2

### Alternative Solution with Called Rule and Action Block (3/3)

**Called Rule**

**rule** NewCustomer (name: String, gender: Person::Gender) {

**to** ← **Target Pattern**

        c : Customer!Customer (

            c.name <- name

        )

**do** { ← **Action Block**

        c.gender <- gender;

        c;

    }

}

Result of Called Rules is the last statement executed in the Action Block



## Example #2 Alternative Solution with Lazy Rule and Action Block (1/2)

```
rule PSystem2CSys {  
    from  
        ps : Person!PSystem  
    to  
        cs : Customer!CSys (  
    )  
    do {  
        for( p in Person!Person.allInstances() ) {  
            if(p.age > 18)  
                cs.customer <- thisModule.NewCustomer(p);  
        }  
    }  
}
```

Matched Rule

Action Block

Explicit Query on Source Model

Binding

Lazy Rule Call



## Example #2 Alternative Solution with Lazy Rule and Action Block (2/2)

Lazy Rule

```
lazy rule NewCustomer{
```

from

```
    p : Person!Person
```

to

```
    c : Customer!Customer (
```

```
        name <- p.name
```

```
        gender <- p.gender
```

```
)
```

```
}
```

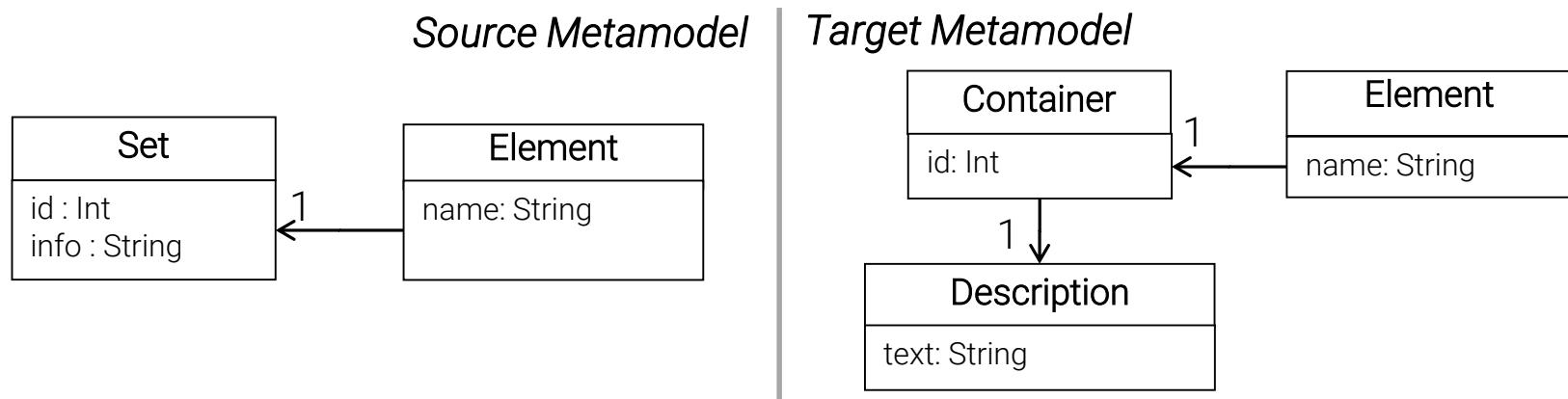
Source Pattern

Target Pattern

Result of Lazy Rules is the first specified target pattern element



## Example #3 Resolve Temp – Explicitly Querying Trace Models (1/4)

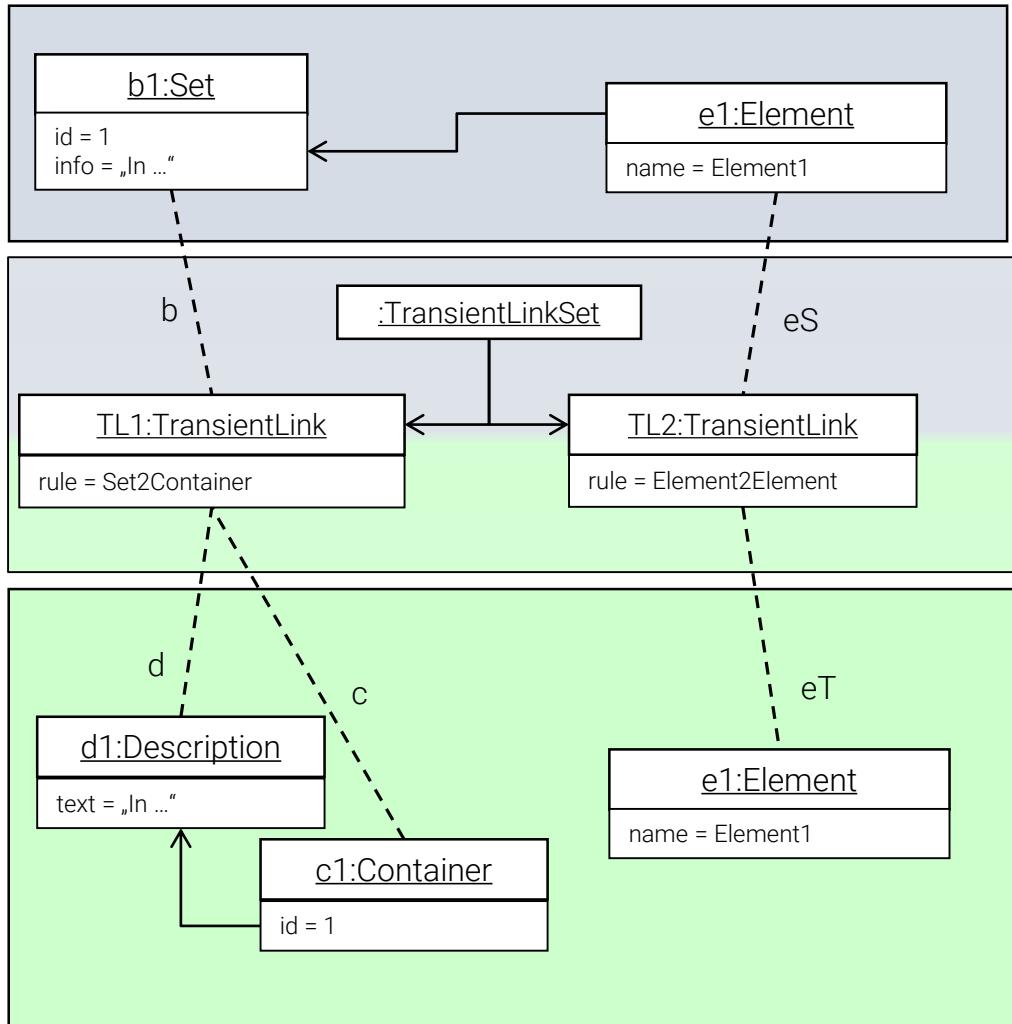


### Transformation

```
rule Set2Container {
    from
        b : source!Set
    to
        d : target!Description(
            text <- b.info
        ),
        c : target!Container(
            id <- b.id,
            description <- d
        )
}
```

```
rule Element2Element{
    from
        eS : source!Element
    to
        eT : target!Element (
            name <- eS.name,
            container <- thisModule.resolveTemp(
                eS.set, 'c'
            )
        )
}
```

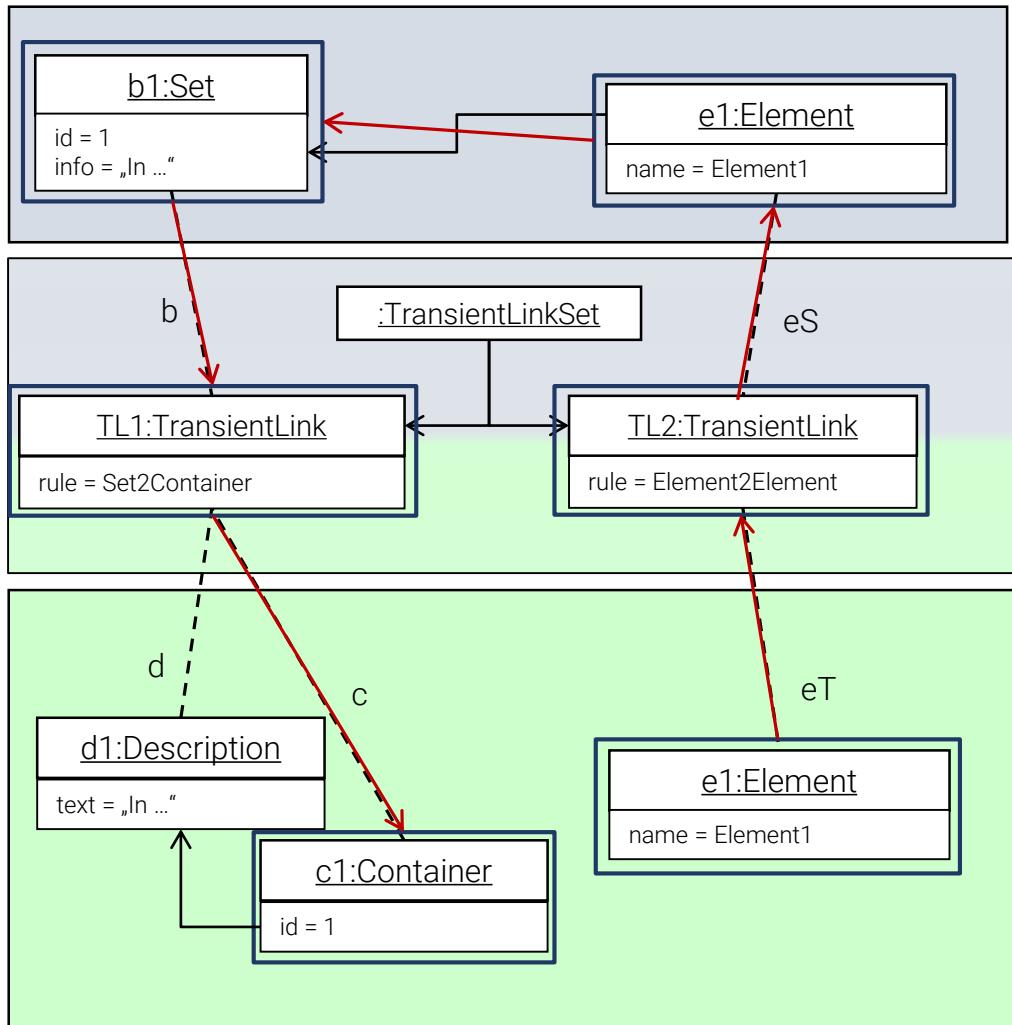
## Example #3 Resolve Temp – Explicitly Querying Trace Models (2/4)



```
rule Set2Container {
    from
        b : source!Set
    to
        d : target!Description(
            text <- b.info
        )
        c : target!Container(
            id <- b.id,
            description <- d
        )
}
```

```
rule Element2Element{
    from
        eS : source!Element
    to
        eT : target!Element (
            name <- eS.name,
            container <- thisModule.resolveTemp(
                eS.set, 'c'
            )
        )
}
```

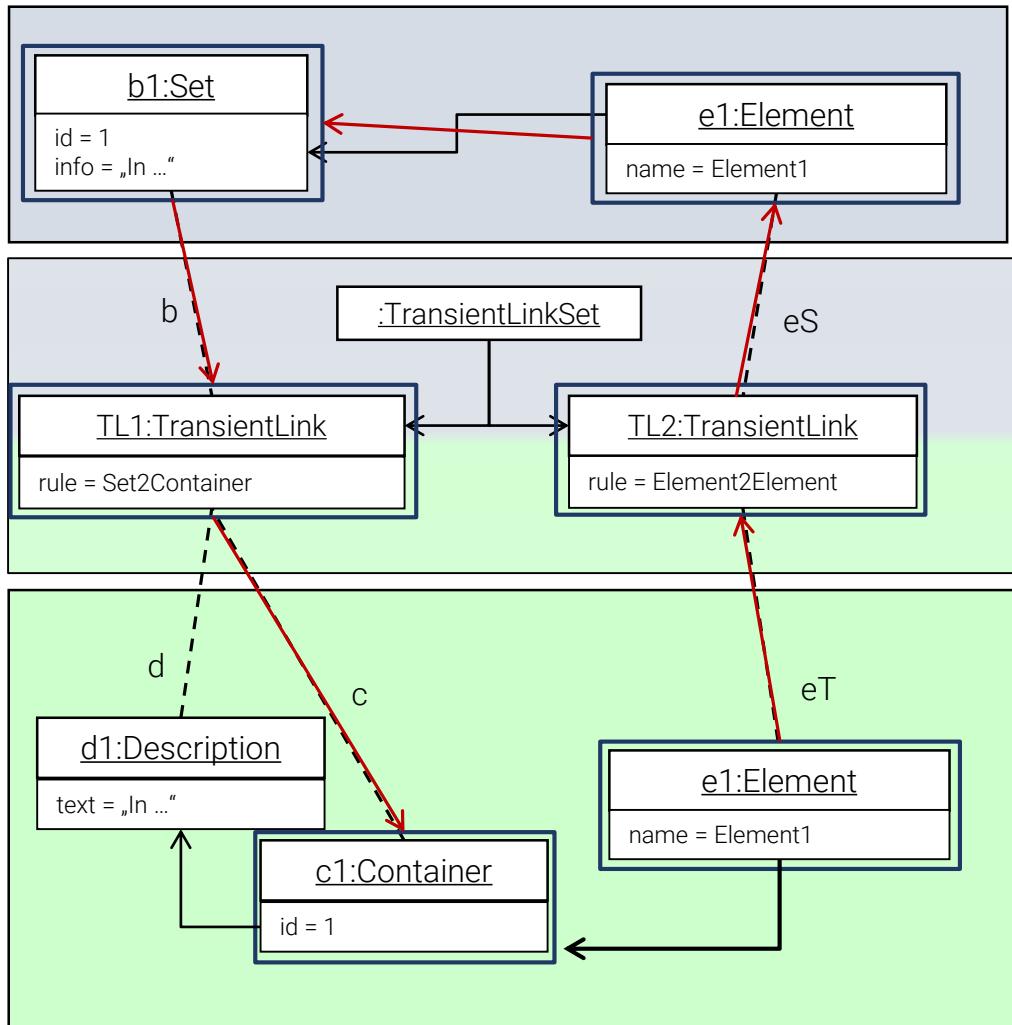
## Example #3 Resolve Temp – Explicitly Querying Trace Models (3/4)



```
rule Set2Container {
    from
        b : source!Set
    to
        d : target!Description(
            text <- b.info
        )
        c : target!Container(
            id <- b.id,
            description <- d
        )
}
```

```
rule Element2Element{
    from
        eS : source!Element
    to
        eT : target!Element (
            name <- eS.name,
            container <- thisModule.resolveTemp(
                eS.set, 'c'
            )
        )
}
```

## Example #3 Resolve Temp – Explicitly Querying Trace Models (4/4)



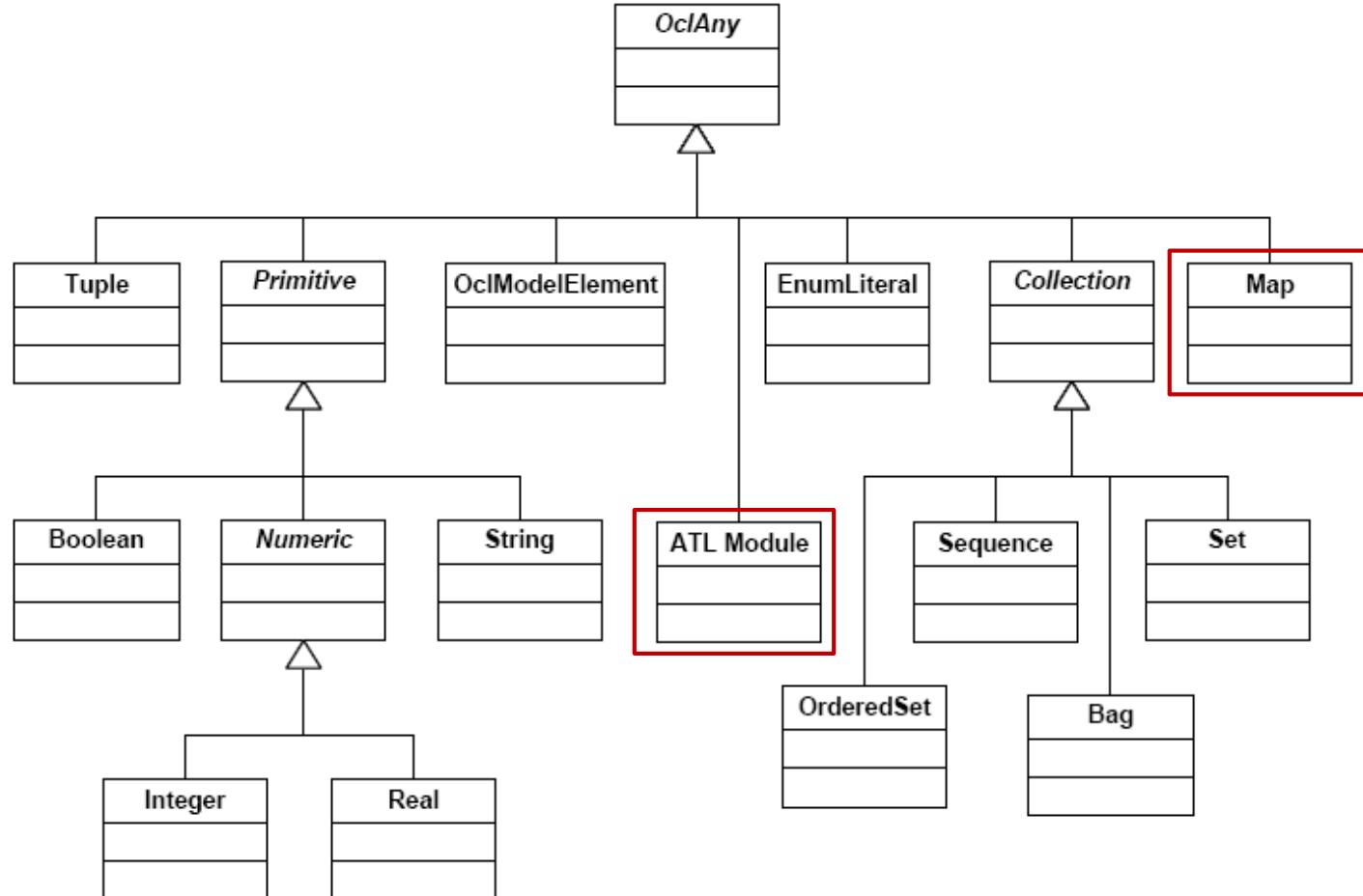
```
rule Set2Container {
    from
        b : source!Set
    to
        d : target!Description(
            text <- b.info
        )
        c : target!Container(
            id <- b.id,
            description <- d
        )
}
```

```
rule Element2Element{
    from
        eS : source!Element
    to
        eT : target!Element (
            name <- eS.name,
            container <- thisModule.resolveTemp(
                eS.set, 'c'
            )
        )
}
```



# ATL Data Types

- OCL Operations for each Type





## Rule inheritance

- Rule inheritance allows for reusing transformation rules
- A child rule may match a subset of what its parent rule matches
  - » All the bindings and filters of the parent still make sense for the child
- A child rule may specialize target elements of its parent rule
  - » Initialization of existing elements may be specialized
- A child rule may extend target elements of its parent rule
  - » New elements may be created
- A parent rule may be declared as abstract
  - » Then the rule is not executed, but only reused by its children
- Syntax

```
abstract rule R1 {  
    ...  
}  
rule R2 extends R1 {  
    ...  
}
```



# Rule inheritance

## Example #1

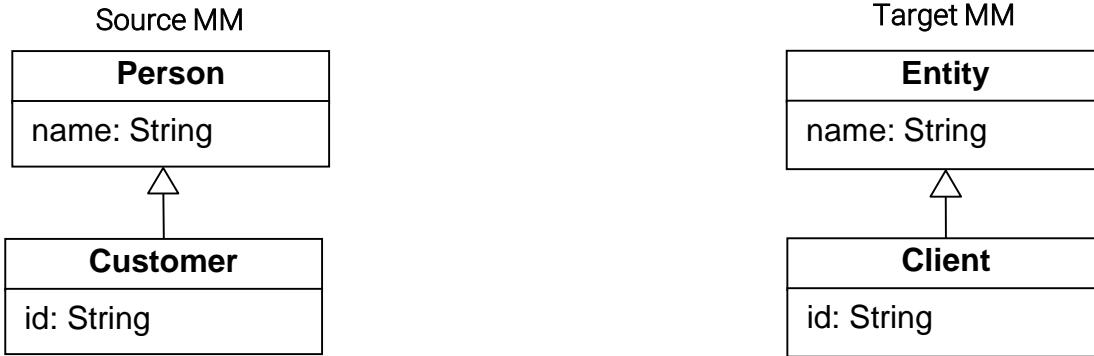


```
abstract rule Person2Entity {
    from
        p : source!Person
    to
        e : target!Entity(
            name <- p.name
        )
}
rule Customer2Client extends Person2Entity{
    from
        cu : source!Customer
    to
        cl : target!Client (
            id <- cu.id
        )
}
```



# Rule inheritance

## Example #2



```
rule Person2Entity {
    from
        p : source!Person (p.oclIsTypeOf(source!Person))
    to
        e : target!Entity(
            name <- p.name
        )
}
rule Customer2Client extends Person2Entity{
    from
        cu : source!Customer
    to
        cl : target!Client (
            id <- cu.id
        )
}
```

*A match has to be unique!*



## Debugging Hints

- Quick and Dirty: Make use of `.debug()`
- Proceed in tiny increments
- Immediately test changes
- Read Exception Trace
  - » Look top-down the stack trace for „ERROR“ to find meaningful message:

```
`` Check line number:  
***** BEGIN Stack Trace  
message: ERROR: could not find operation ChXXXXapter2TitleValue on Module having supertypes: [OclAny]
```

Line number

```
do_blocks can be used for temporary debug output  
A._applyBook2Line(1 : NTransientLink) : ??#32 14:25-14:76
```



## ATL in use

- ATL tools and documentation are available at <http://www.eclipse.org/atl>
  - » Execution engine
    - Virtual machine
    - ATL to byte code compiler
  - » Integrated Development Environment (IDE) for
    - Editor with syntax highlighting and outline
    - Execution support with launch configurations
    - Source-level debugger
  - » Documentation
    - Starter's guide
    - User manual
    - User guide
    - Basic examples



# Summary

- ATL is specialized in out-place model transformations
  - » Simple problems are generally solved easily
- ATL supports advanced features
  - » Complex OCL navigation, called rules, refining mode, rule inheritance, etc
  - » Many complex problems can be handled declaratively
- ATL has declarative and imperative features
  - » Any out-place transformation problem can be handled
- Further information
  - » Documentation: [http://wiki.eclipse.org/ATL/User\\_Guide\\_-\\_The\\_ATL\\_Language](http://wiki.eclipse.org/ATL/User_Guide_-_The_ATL_Language)
  - » Examples: [http://www.eclipse.org/m2m/atl/basicExamples\\_Patterns](http://www.eclipse.org/m2m/atl/basicExamples_Patterns)



# Alternatives to ATL

- **QVT**: Query-View-Transformation standard of the OMG
  - » Declarative QVT Relational language
  - » Imperative QVT Operational language
  - » Low-level QVT Core language (VM level)
- **TGG**: Triple Graph Grammars
  - » Correspondence graphs between metamodels
  - » Transform models in both directions, integrate and synchronize models
- **JTL**: Janus Transformation Language
  - » Strong focus on model synchronization by change propagating
- **ETL**: Epsilon Transformation Language
  - » Designated language in the Epsilon framework for out-place transformations
- **RubyTL**: Ruby Transformation Language
  - » Extension of the Ruby programming language
  - » Core concepts for out-place model transformations (extendable)
- Many more languages such as VIATRA, Tefkat, Kermeta, or SiTra, ...



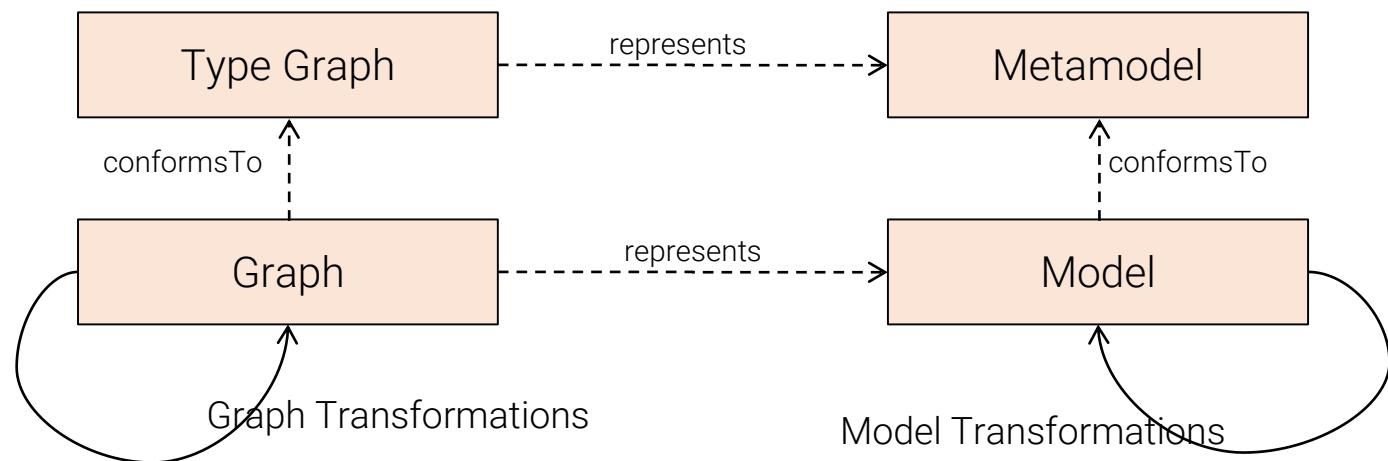
# 7.3 Endogenous, In-Place Transformations

Graph transformations

# Why graph transformations?

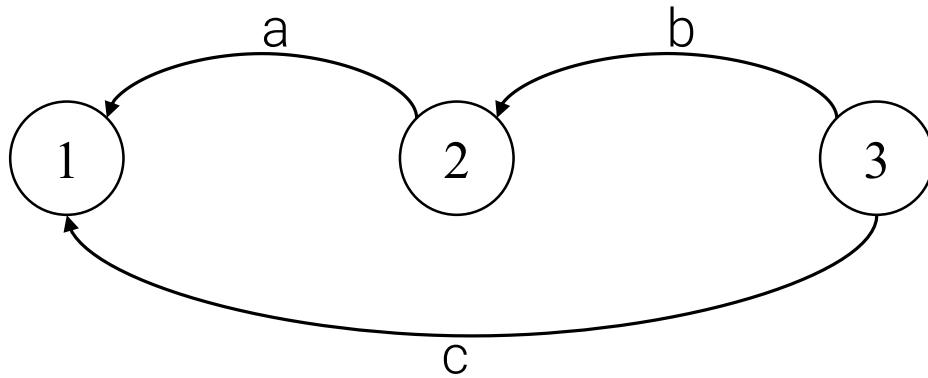
- Models are graphs
  - » Class diagram, Petri net, State machine, ...
- Type Graph:
  - » Generalization of graph elements
- Graph transformations
  - » Generalization of the graphs' evolutions

**Graph transformations are applicable for models!**



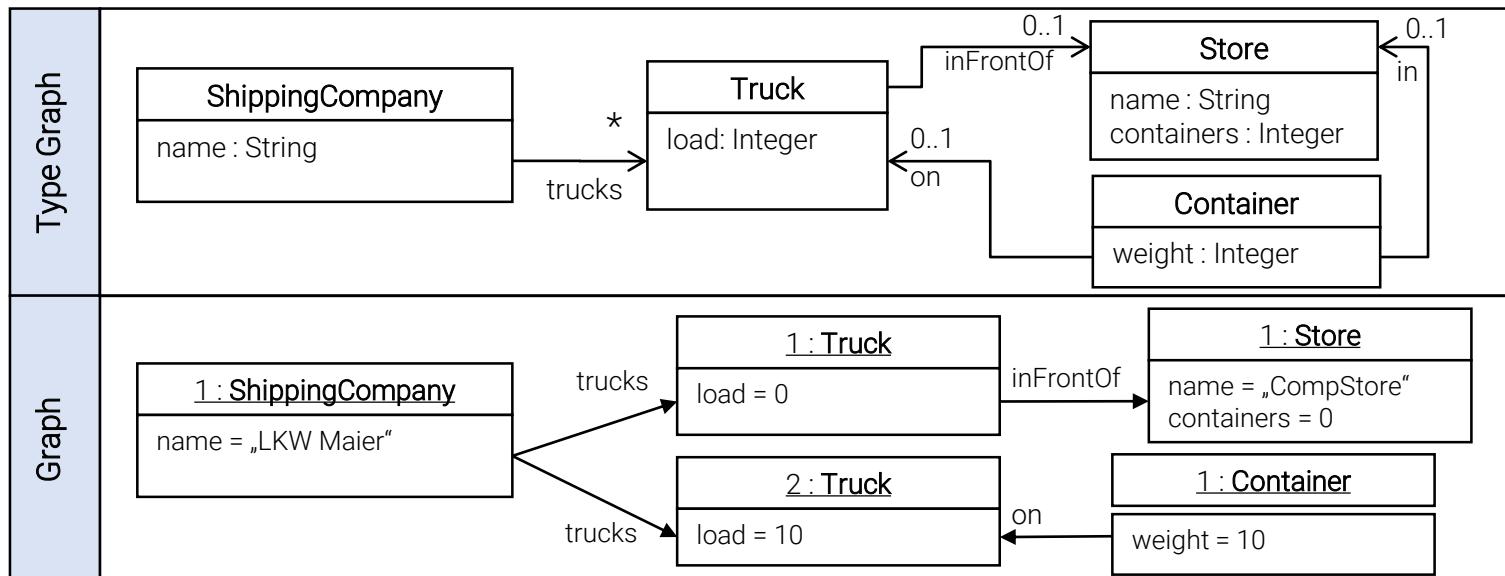
## Basics: Directed graph

- A directed graph  $G$  consists of two disjoint sets
  - » Vertices  $V$  (*Vertex*)
  - » Edges  $E$  (*Edge*)
- Each edge has a source vertex  $s$  and a target vertex  $t$
- Summarized:  $G = (V, E, s: E \rightarrow V, t: E \rightarrow V)$
- Example graph
  - »  $V = \{1, 2, 3\}$
  - »  $E = \{a, b, c\}$
  - »  $s(a) = 2$
  - »  $t(a) = 1$
  - »  $s(b) = 3$
  - »  $t(b) = 2$
  - » ...



# Typed attributed graph (1/3)

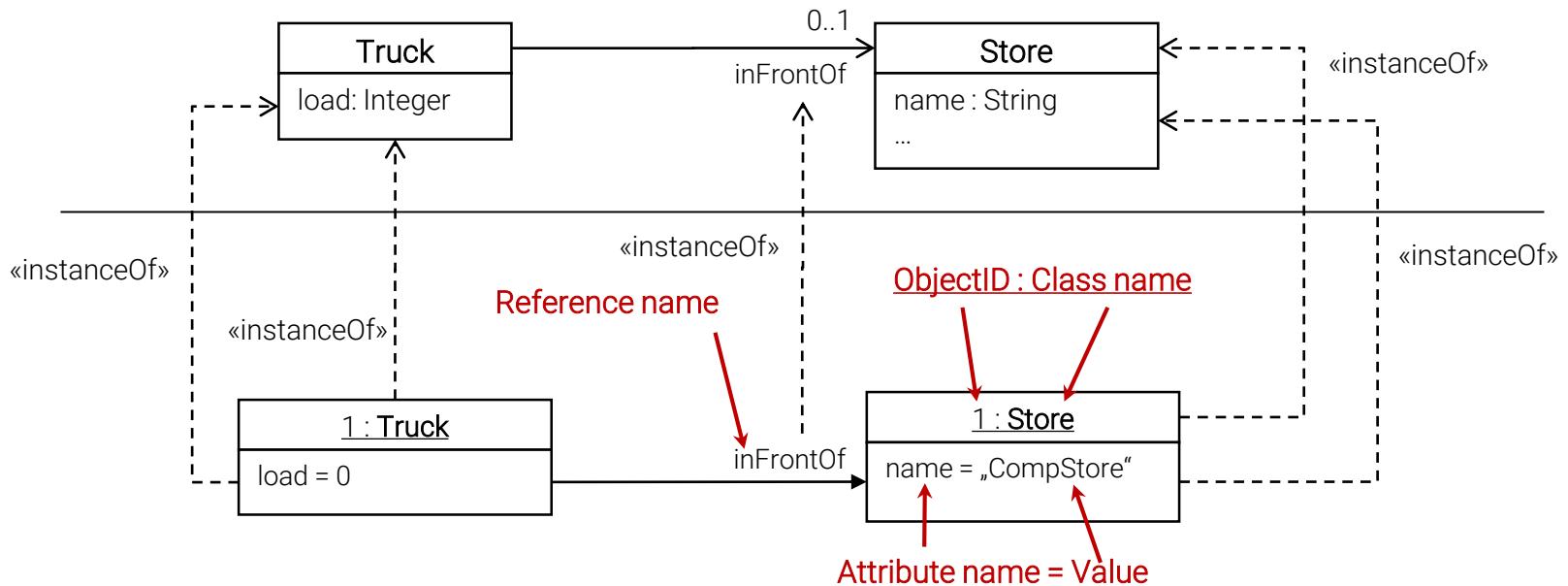
- To represent models further information is needed
  - » Typing: Each vertex and each edge has a type
  - » Attribution: Each vertex/edge has an arbitrary number of name/value pairs
- Notation for directed, typed and attributed graphs
  - » Graph is represented as an object diagram
  - » Type graph is represented as a class diagram
- Example



Example taken from: C. Ermel, M. Rudolf, and G. Taentzer, **The AGG approach: Language and environment**, *Handbook of Graph Grammars and Computing by Graph Transformation, Application, Languages and Tools*, volume 2, World Scientific, 1999.

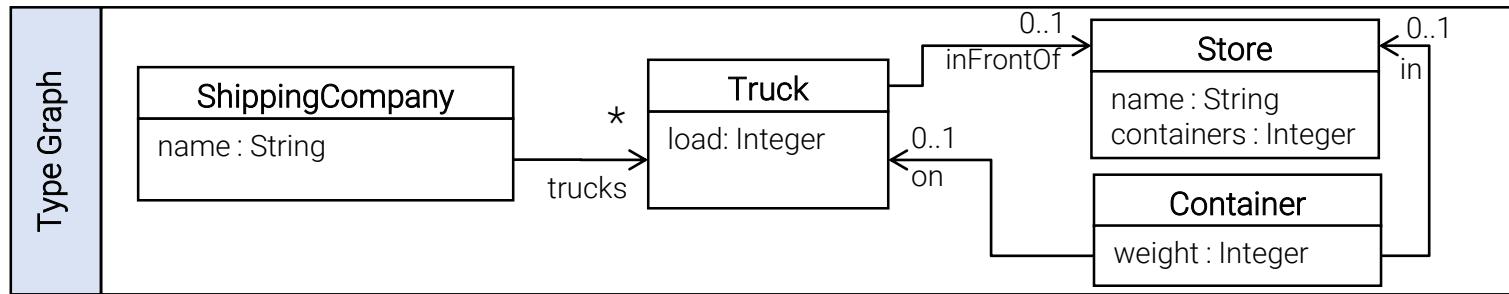
## Typed attributed graph (2/3)

- Object diagram
  - » Instance of class diagram
- Basic concepts
  - » Object : Instance of class
  - » Value: Instance of attribute
  - » Link: Instance of reference



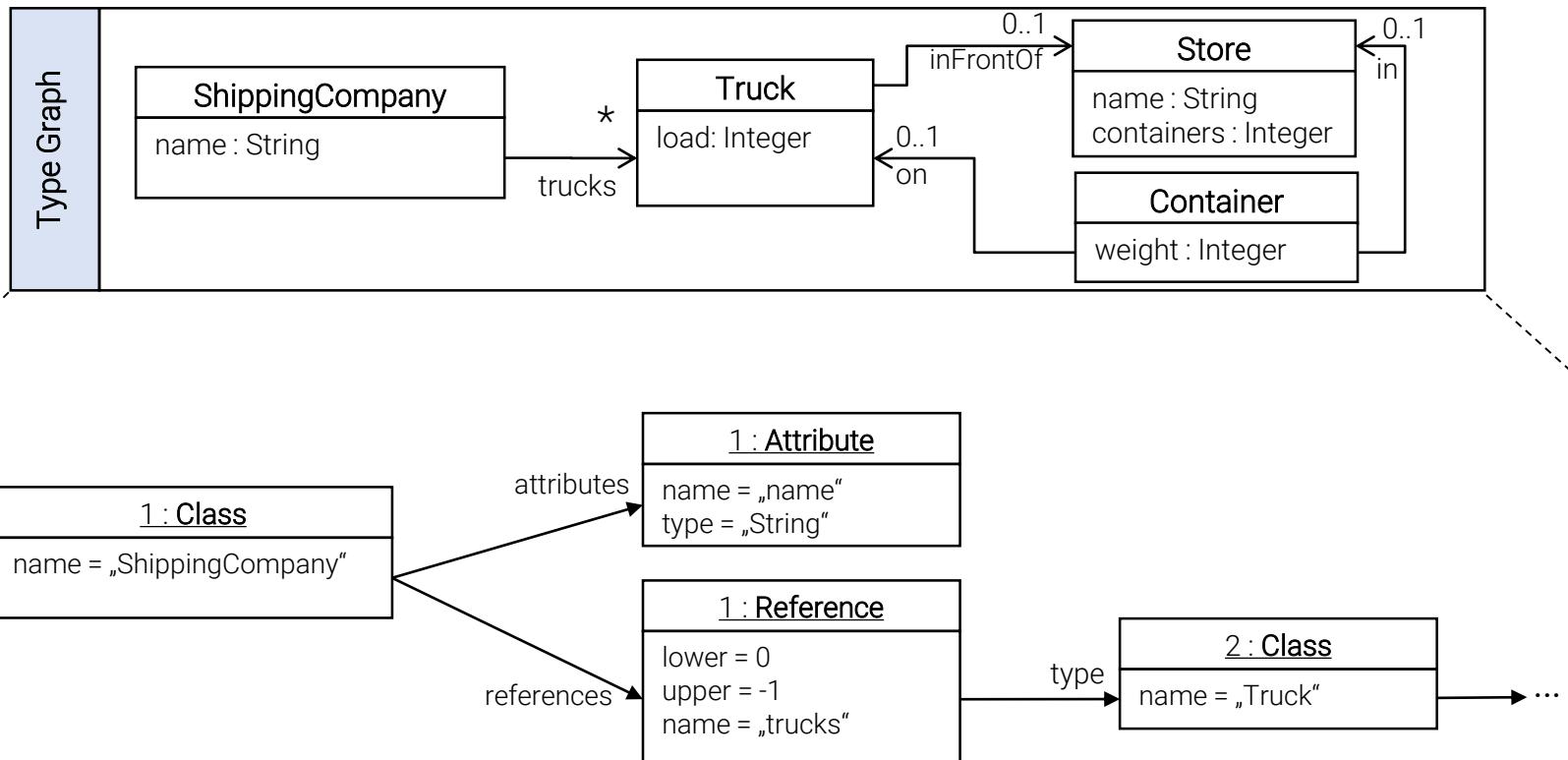
## Typed attributed graph (3/3)

- Question: How does the type graph look in pure graph shape (object diagram)?



## Typed attributed graph (3/3)

- Question: How does the type graph look in pure graph shape (object diagram)?



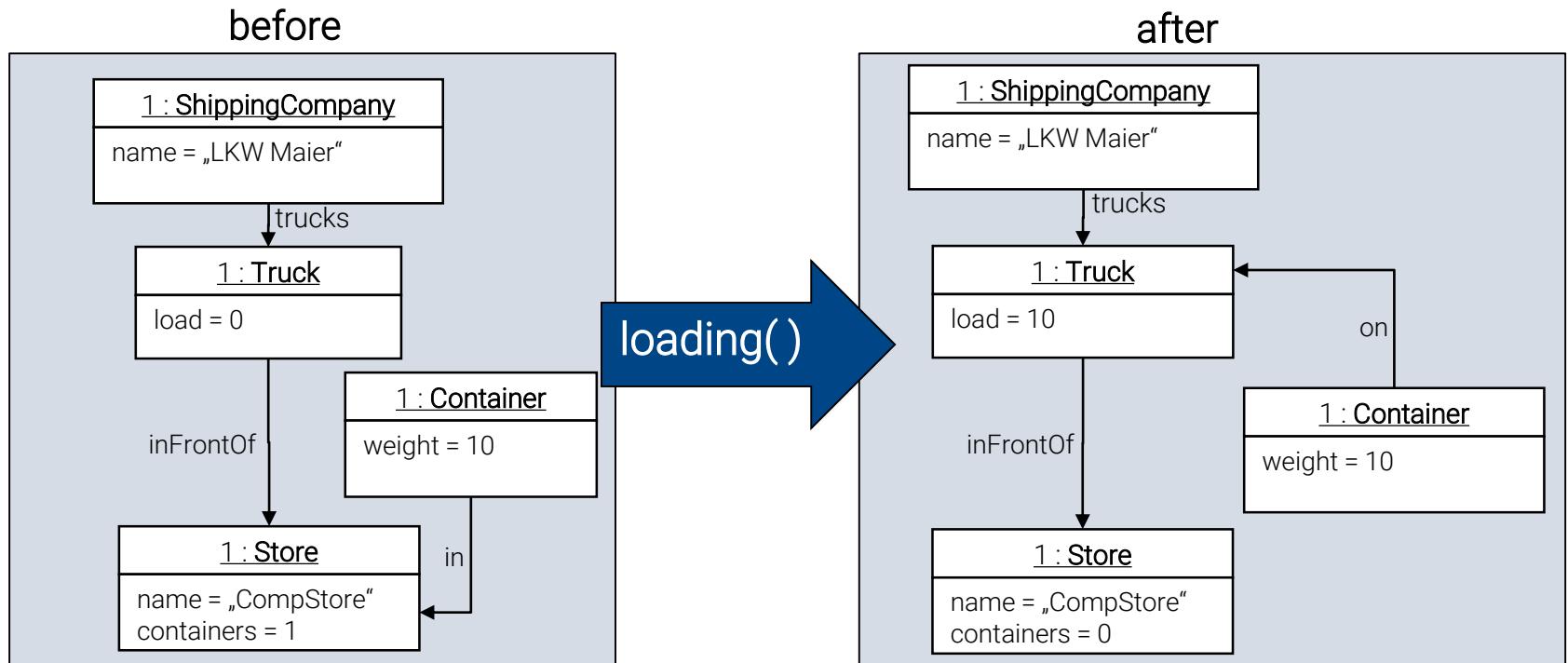


## Until now...

- ...we considered models as static entities
  - » A modeler creates a model using an editor – Done!
- But what is about dynamic model modifications?
- They are needed for
  - » Simulation
  - » Execution
  - » Animation
  - » Transformation
  - » Extension
  - » Improvement
  - » ...
- How can graphs be modified?
  - » Imperative: Java Program + Model API
  - » Declarative: Graph transformations by means of graph transformation rules

## Example

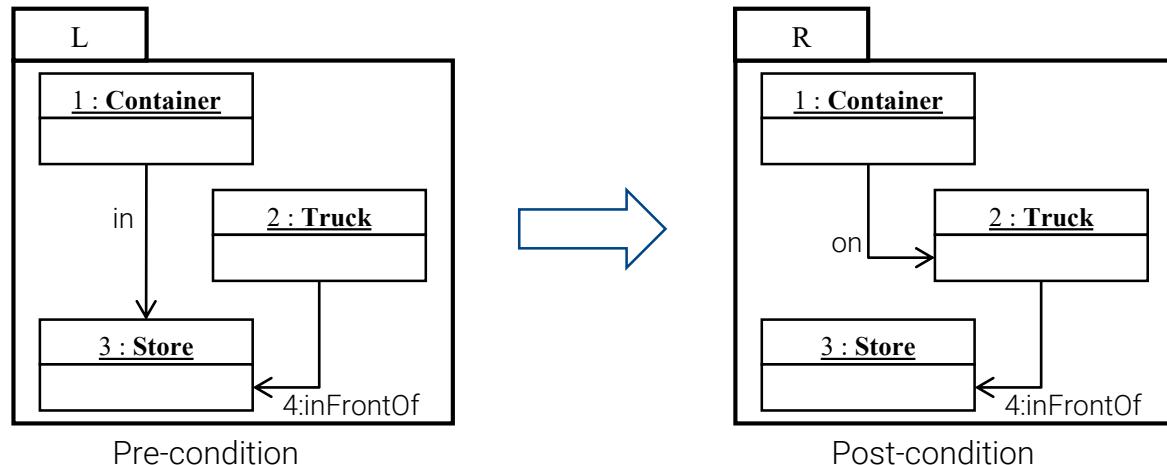
- Operation: Loading of Container onto a Truck



- How can this behaviour be described in a generic way?

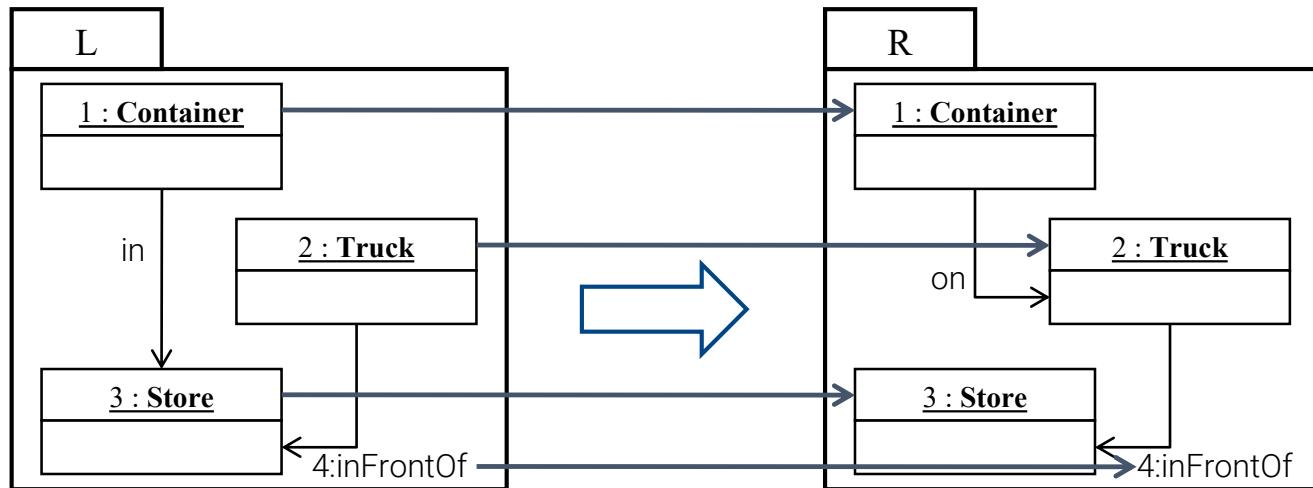
# Graph transformation rule

- A graph transformation rule  $\rho: L \rightarrow R$  is a structure preserving, partial mapping between two graphs
  - » L and R are two directed, typed and attributed graphs themselves
  - » Structure preserving, because vertices, edges and values may be preserved
  - » Partial, because vertices and edges may be added/deleted
- Example: Loading of Container onto a Truck



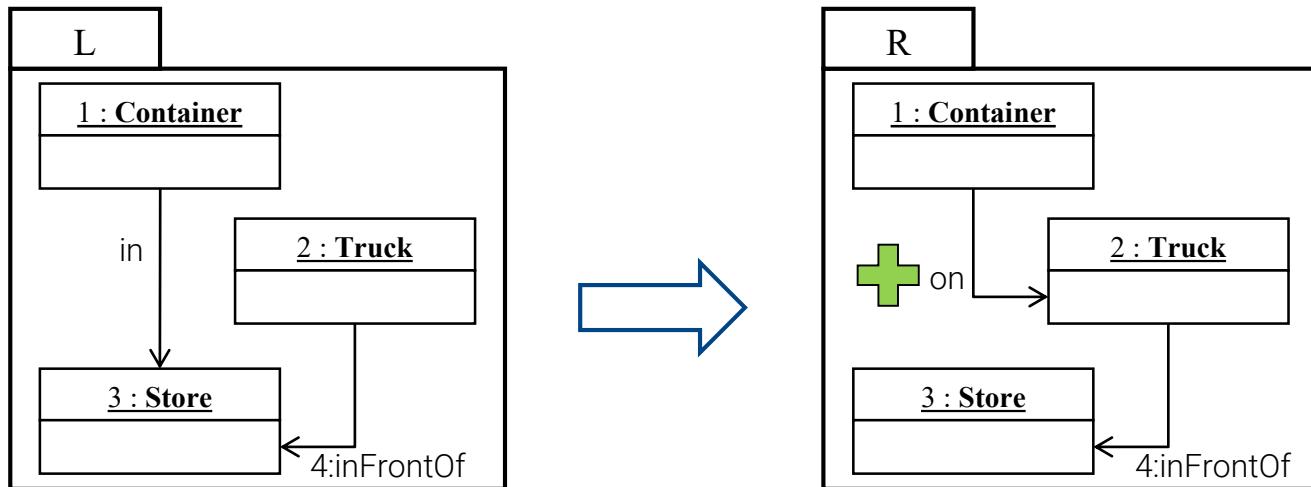
# Graph transformation rule

- Structure preserving
  - » All vertices and edges which are contained in the set  $L \cap R$
- Example



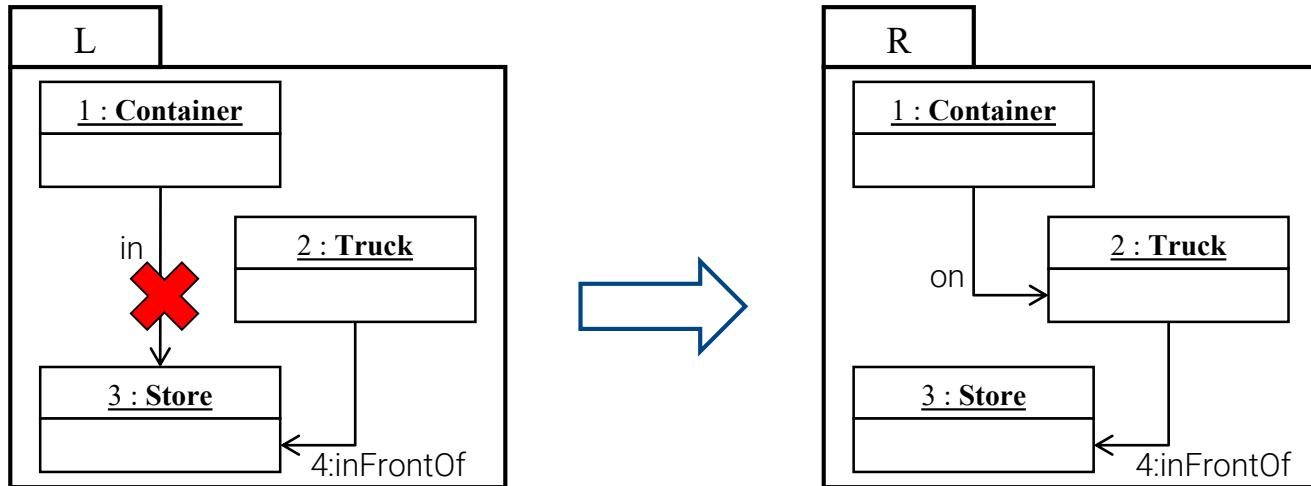
# Graph transformation rule

- Adding
  - » All vertices and edges which are contained in the set  $R \setminus L$
- Example



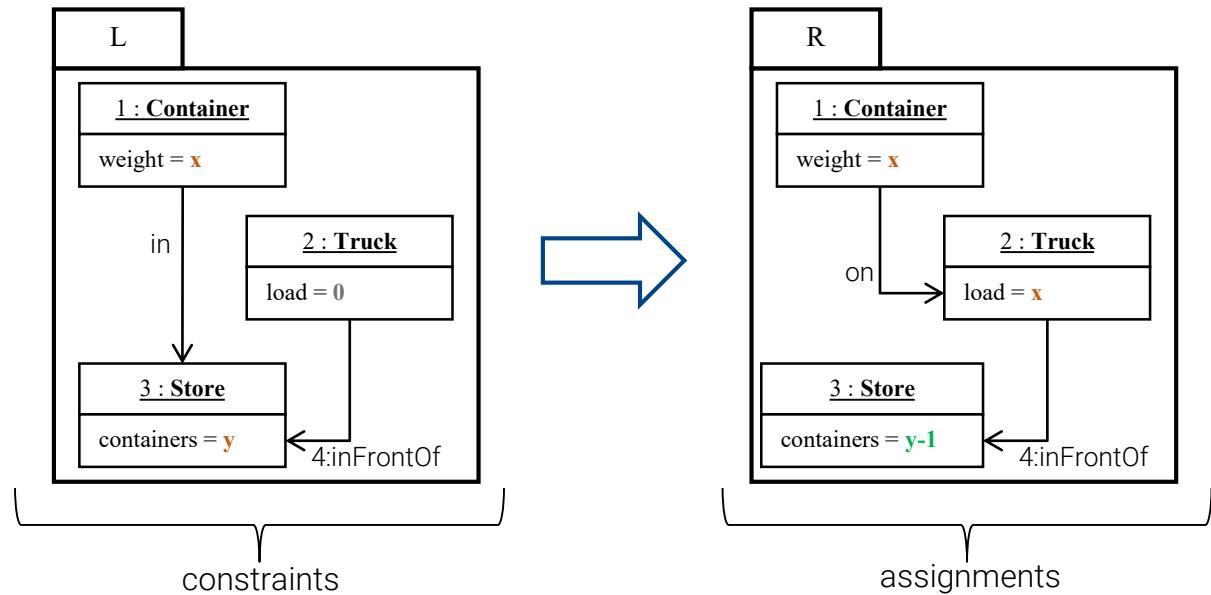
# Graph transformation rule

- Deleting
  - » All vertices and edges which are contained in the set  $L \setminus R$
- Example



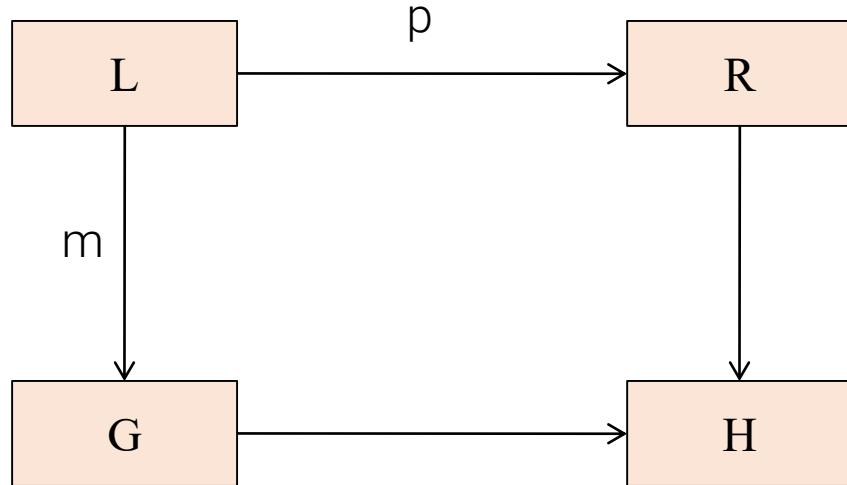
# Graph transformation rule

- Calculating – Make use of attributes
  - » Constants
  - » Variables
  - » Expressions (OCL & Co)
- Example



## Graph transformation

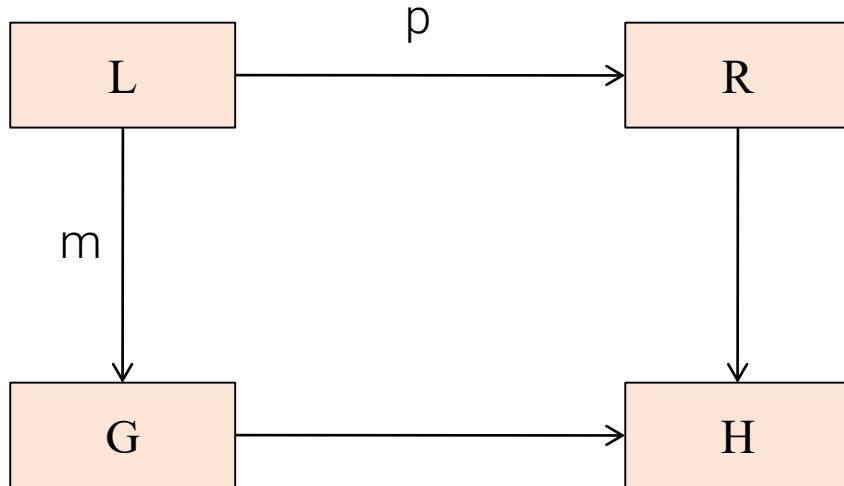
- A graph transformation  $t: \mathcal{G} \rightarrow \mathcal{H}$  is the result of the execution of a graph transformation rule  $p: L \rightarrow R$  in the context of  $G$ 
  - »  $t = (p, m)$  where  $m: L \rightarrow G$  is an injective graph morphism (**match**)



# Graph transformation

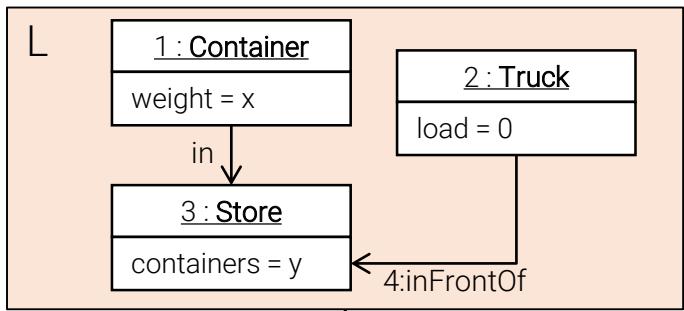
## Operational specification

- Prepare the transformation
  - » Select rule  $p: L \rightarrow R$
  - » Select match  $m: L \rightarrow G$
- Generate new graph  $H$  by
  - » Deletion of  $L \setminus R$
  - » Addition of  $R \setminus L$

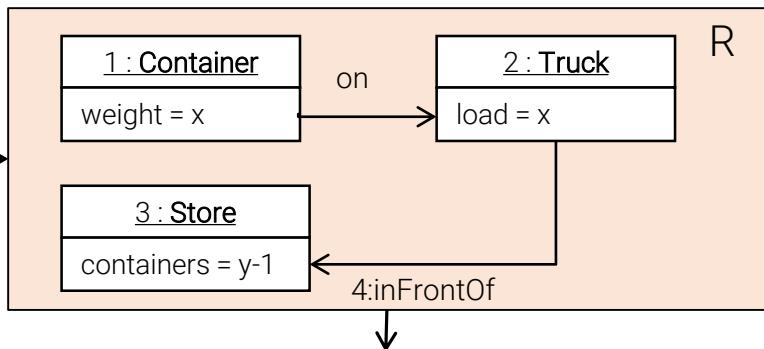


## Graph transformation example (1/2)

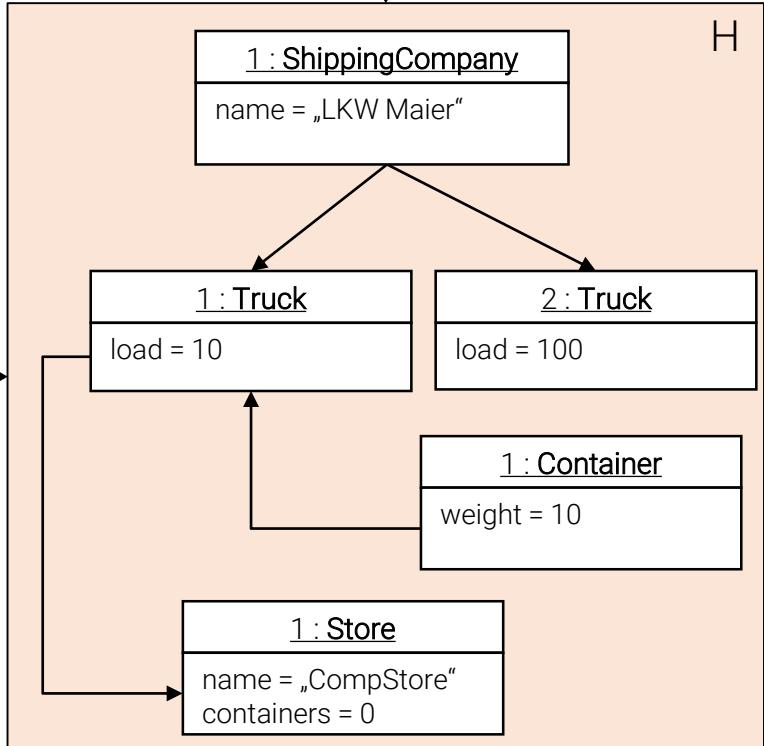
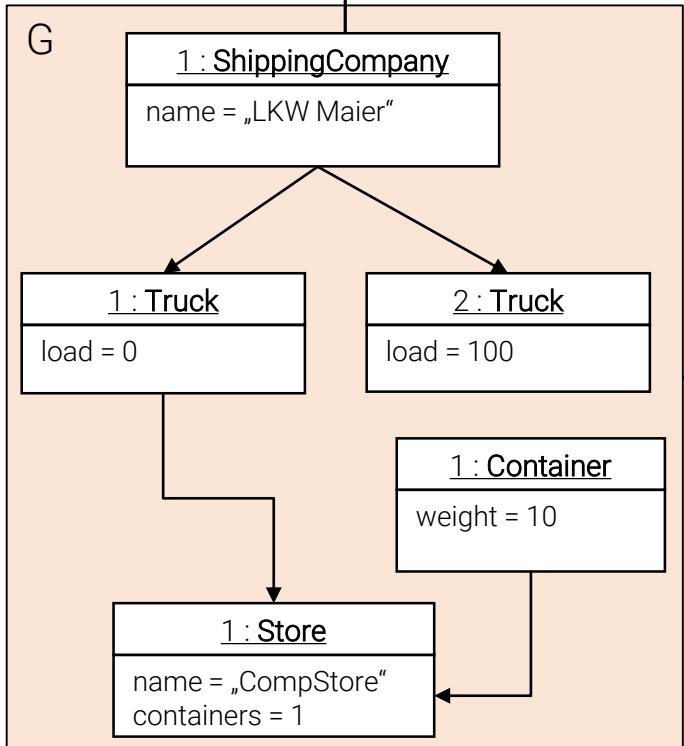
graph transformation rule



p

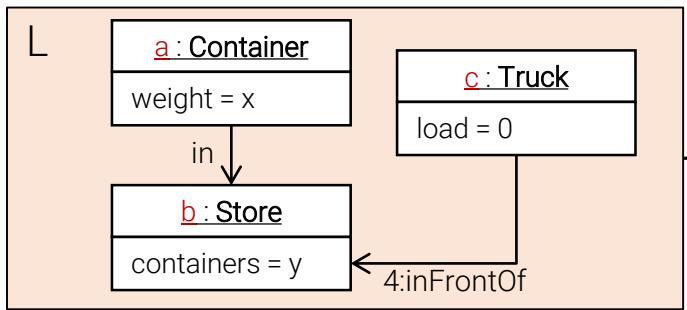


m

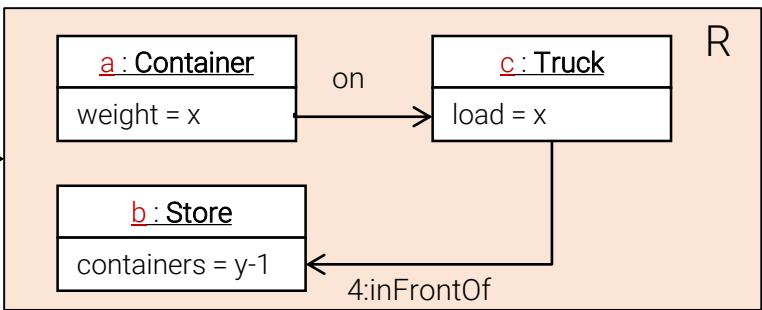


## Graph transformation example (2/2)

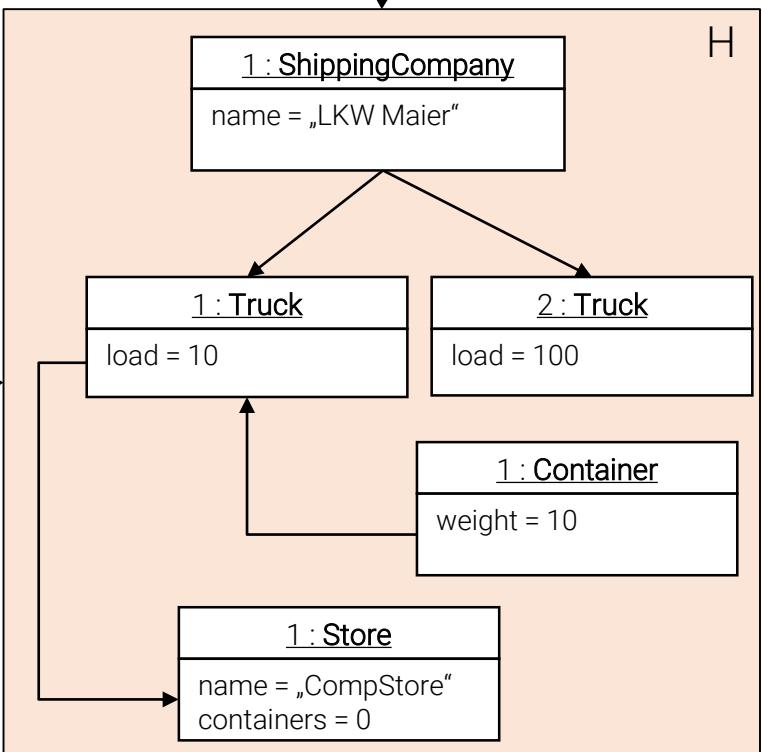
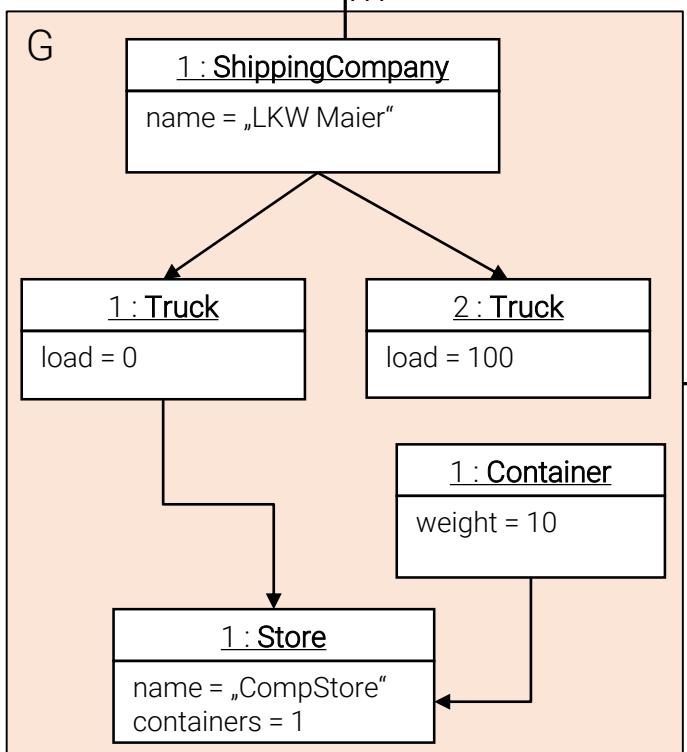
graph transformation rule



p

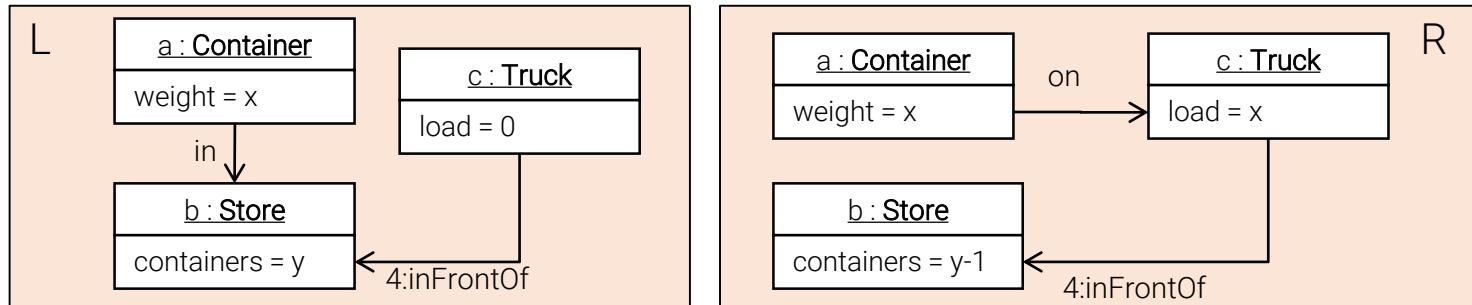


m



# Graph transformations in ATL

- Graph transformation



- ATL (Refining Mode)

```

rule Loading {
    from
        a : Container (a.in = b)
        b : Store
        c : Truck (c.inFrontOf = b AND c.load = 0)
    to
        _a : Container(weight <- a.weight, on <- _c)
        _b : Store(containers <- b.containers - 1)
        _c : Truck(load <- a.weight, inFrontOf <- _b)
}

```



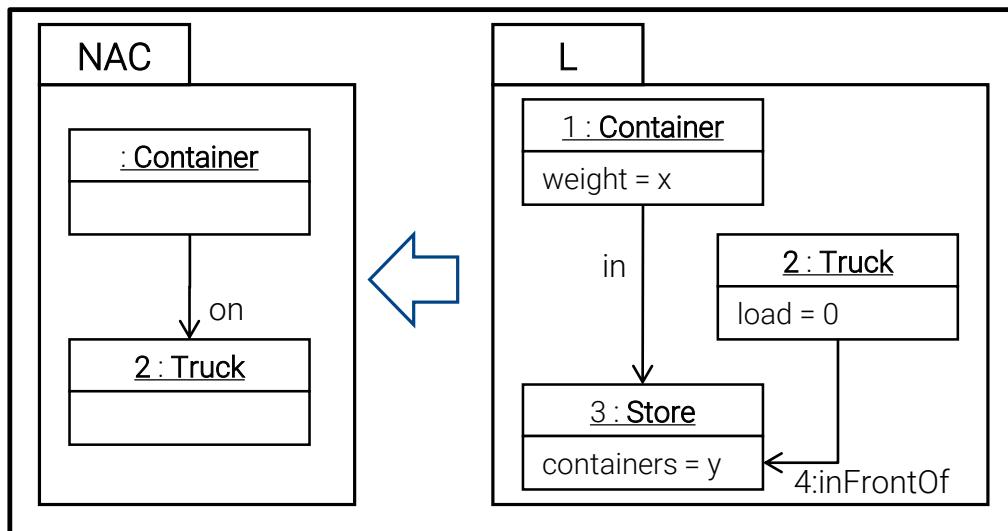
## Negative Application Condition (NAC)

- Left side of a rule specifies what must be existing to execute the rule
  - » Application Condition
- Often needed to describe what must not be existing
  - » Negative Application Condition (NAC)
- NAC is a graph which describes a forbidden sub graph structure
  - » Absence of specific vertices and edges must be granted
- Graph transformation rule is executed when NAC is not fulfilled

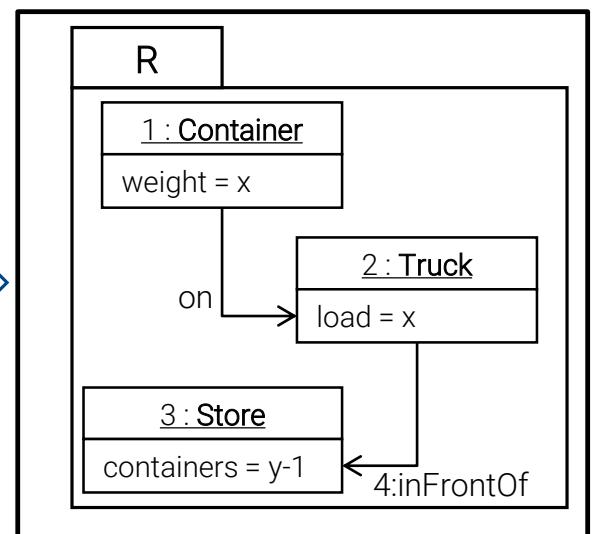
# Negative Application Condition (NAC)

- Example: A truck should only be loaded if there is no container on the truck

Advanced Pre-condition

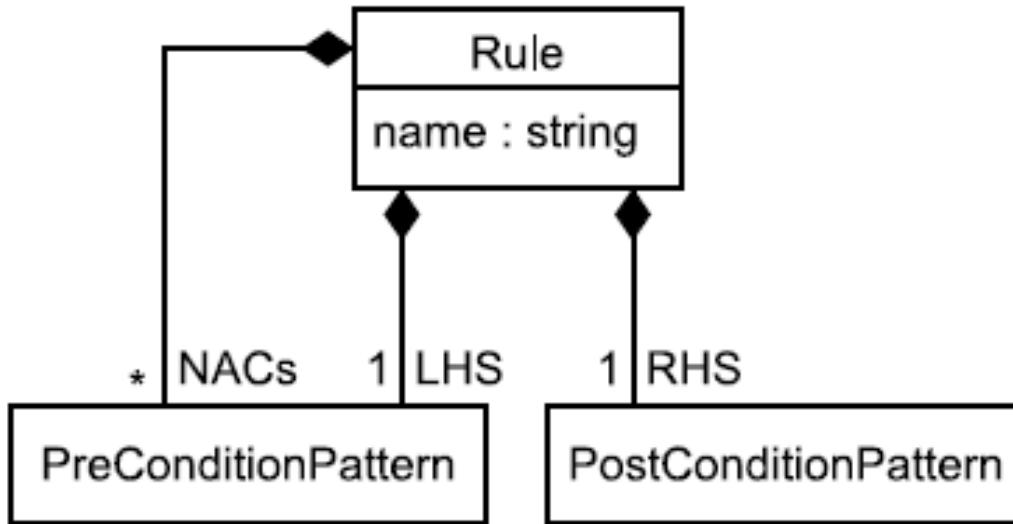


Post-condition



## Negative Application Condition (NAC)

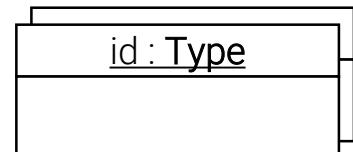
- Multiple NACs for one rule possible
- But: LHS and RHS must only be specified once





## Multi-Object

- The number of matching objects is not always known in advance
- Maximal set of objects which fulfill a certain condition
  - » Selection of all objects x from a set y which fulfill condition z
  - » In OCL:  $y \rightarrow \text{select}(x | z(x))$
- Notation: Multi-Object from UML 1.4
  - » A multi-object symbol maps to a collection of instances in which each instance conforms to a given type and conditions
  - » Example:  $\text{select}(x | x.\text{oclIsTypeOf}(\text{Type}))$





# Execution order of graph transformation rules

- A graph transformation system consists of a set of different graph transformation rules
- In which order are they being executed?
- Multiple procedures
  - » nondeterministic
  - » deterministic
    - Priorities
    - Programs (aka „programmable graph transformations“)
- Example – Specialized UML activity diagrams
  - » [failure]
  - » [success]



# Graph transformation Tools

- VIATRA
- PROGRES
- GrGen.NET
- VMTS
- MOMENT2
- EMT
- Fujaba
- AGG
- MoTif
- GROOVE
- MoTMoT
- ATL Refining Mode
- ...



## Summary

- Model transformations are graph transformations
  - » Type Graph = Metamodel
  - » Graph = Model
- Programmable graph transformations allow for the development of complex graph evolution scenarios
- Exogenous, out-place transformations can also be specified as graph transformations
  - » However more complex as with ATL
- Graph transformations become more and more relevant in practice
  - » Found their way to Eclipse!



# 7.4 Mastering Model Transformations and their Rules



# HOTs

- **Transformations can be regarded as models themselves**
  - » They are instances of a transformation metamodel!
  - » This uniformity allows reusing tools and methods defined for models
- Thus, a transformation model can itself be created or manipulated by transformations, by so-called Higher Order Transformations (HOTs)
  - » Transformations that take as input a model transformation and/or generate a model transformation as output
- **Examples**
  - » Refactoring support for transformations to improve their internal structure
  - » Adding a logging aspect to transformations
  - » ...



# Bi-directional Transformations

- Bi-directional model transformation languages
  - » do not impose a transformation direction when specifying a transformation
  - » allow for different execution modes such as transformation, integration, and synchronization
- Transformation mode is further divided into forward and backward transformation (source -> target -> source)
- Integration mode assumes to have source and target models given and checks if expected elements (that would be produced in the transformation mode) exist
- Synchronization mode updates the models in case the integration mode has reported unsatisfied correspondences
- Languages allowing for bi-directional transformations:
  - » JTL, TGG, QVT Relational, ...



# Lazy & Incremental Transformations

- Standard execution strategy for out-place transformations
  - » Read the complete input model
  - » Produce the output model from scratch by applying all matching transformation rules.
- Such executions are often referred as batch transformations.
- Two scenarios may benefit from alternative execution strategies
  - » An output model already exists from a previous transformation run for a given input model → incremental transformations
  - » Only a part of the output model is needed by a consumer → lazy transformations
- Experimental implementations for lazy and incremental transformations are available for ATL



# Transformation Chains

- Transformations may be complex processes
- Divide and Conquer
  - » Use different transformation steps to avoid having one monolithic transformation!
- Transformation chains are the technique of choice for modeling the orchestration of different model transformations
- Transformation chains are defined with orchestration languages
  - » Simplest form: sequential steps of transformations executions
  - » More complex forms: conditional branches, loops, and further control constructs
  - » Even HOMTs may produce dynamically transformations used by the chain
- Smaller transformations focusing on certain aspects allow for higher reusability



# CHAPTER 8 - Model-to-Text Transformations



# Overview

## 1 Introduction

- 1.1 Human Cognitive Processes
- 1.2 Models
- 1.3 Model Engineering

## 2 MDSE Principles

- 2.1 MDSE Basics
- 2.2 The MD\* Jungle of Acronyms
- 2.3 Modeling Languages and Metamodeling
- 2.4 Overview Considered Approaches
- 2.5 Tool Support
- 2.6 Criticisms of MDSE

## 3 MDSE Use Cases

- 3.1 MDSE applications
- 3.2 USE CASE1 – Model driven development
- 3.3 USE CASE2 – Systems interoperability
- 3.4 USE CASE3 – Model driven reverse engineering



# Overview

## 4 Model-Driven Architecture (MDA)

- 4.1 MDA Definitions and Assumptions
- 4.2 The Modeling Levels: CIM, PIM, PSM
- 4.3 Architecture-Driven Modernization

## 5 Modeling Languages at a Glance

- 5.1 Anatomy of Modeling Languages
- 5.2 General Purpose vs. Domain-Specific Modeling Languages
- 5.3 Overview of UML Diagrams
- 5.4 UML Behavioural or Dynamic Diagrams
- 5.5 UML Extensibility: The Middle Way Between GPL and DSL
- 5.6 Domain Specific Languages
- 5.7 Defining Modeling Constraints (OCL)



# Overview

## 6 Developing your Own Modeling Language

- 6.1 Metamodel-Centric Language Design
- 6.2 Programming Languages
- 6.3 Metamodel Development Process
- 6.4 MOF - Meta Object Facility
- 6.5 Example DSML: sWML
- 6.6 EMF and Ecore
- 6.7 Abstract Syntax Development
- 6.8 Graphical Concrete Syntax Development
- 6.9 Textual Concrete Syntax Development

## 7 Model-to-Model Transformations

- 7.1 Model Transformations and their Classification
- 7.2 Exogenous, Out-Place Transformations
- 7.3 Endogenous, In-Place Transformations
- 7.4 ATL Overview
- 7.5 Mastering Model Transformations and their Rules



# Overview

## 8 Model-to-Text Transformations

- 8.1 Basics of Model-Driven Code Generation
- 8.2 Code Generation Through Programming Languages
- 8.3 Code Generation Through M2T Transformation Languages
- 8.4 Mastering Code Generation
- 8.5 Excursus: Code Generation Through M2M Transformations and TCS

## 9 Analyzing Models (not included in the book)

- 11.1 Model Analysis
- 11.2 Tooling
- 11.3 Use Cases



# Introduction Terminology

- **Code generation**
  - » Wikipedia:  
„Code generation is the process by which a compiler’s code generator converts a syntactically-correct program into a series of instructions that can be executed by a machine.“
  - » Code Generation in Action (Herrington 2003):  
„Code generation is the technique of using or writing programs that write source code.“
- **Code generation (<http://en.wikipedia.org>)**
  - » Compiler Engineering: component of the synthesis phase
  - » Software Engineering: program to generate source code
- **Résumé: Term Code Generation is overloaded!**



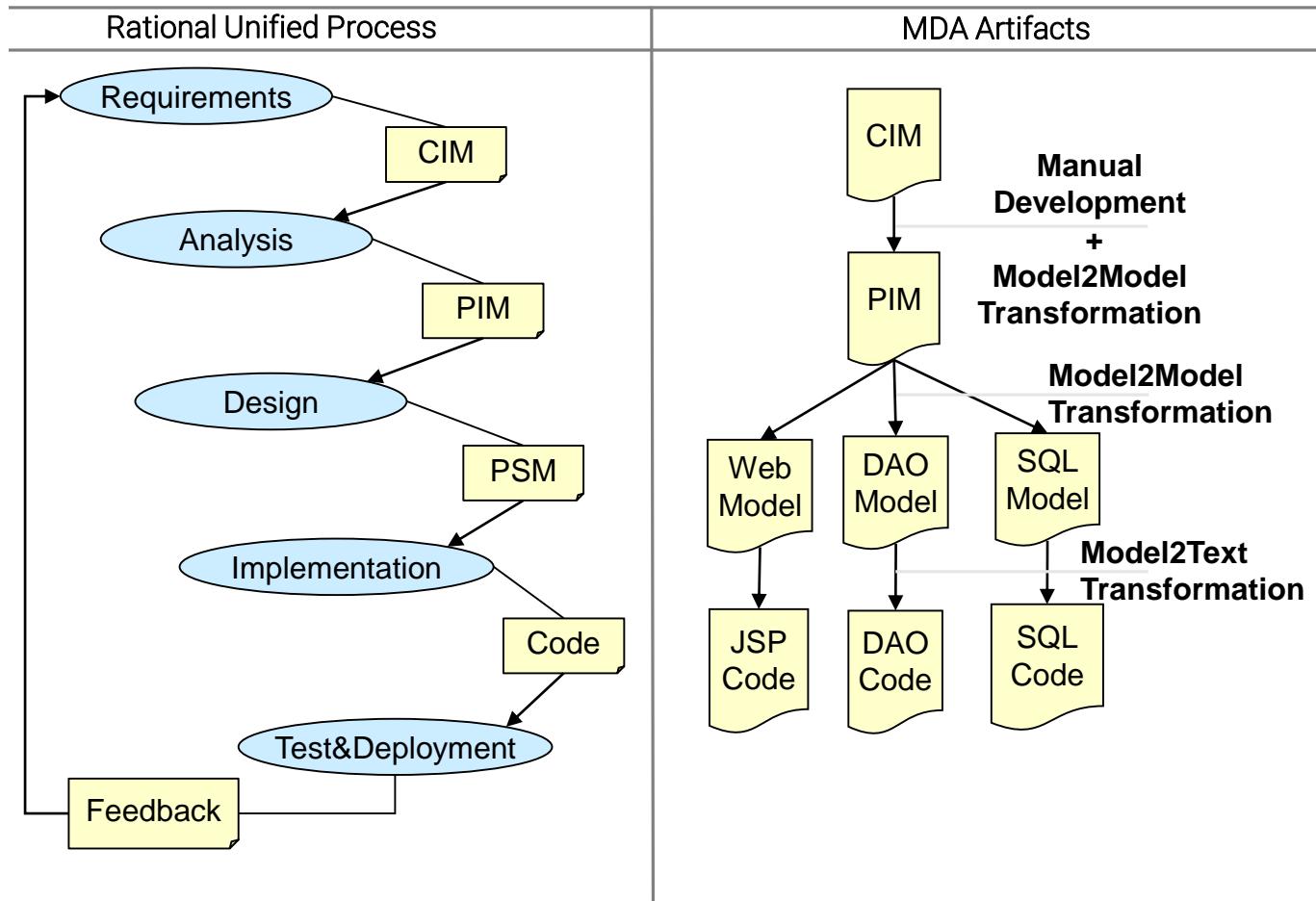
# Introduction

## Code Generation - Basic Questions

- **How much is generated?**
  - » Which parts can be automatically generated from models?
  - » Full or partial code generation?
- **What is generated?**
  - » Which kind of source code to generate?
  - » The less code to generate the better!
- **How to generate?**
  - » Which languages and tools to use for developing code generators?
  - » GPLs vs. DSLs

# Introduction

- Code Generation in MDA (just an example)





# Introduction

## What kind of code is generated?

- Model-to-Text, whereas text may be distinguished in
  - » Program code
  - » Documentation
  - » Test cases
  - » Model serialization (XMI)
- Direct translation to machine code possible, but inconvenient, error-prone and hard to optimize
  - » Reuse existing code generators
  - » Using existing functionality (frameworks, APIs, components)
  - » Motto: The less code to generate, the better!



# Introduction

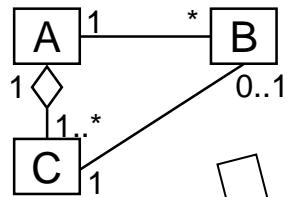
## Example: Platform for Web application development

- Example: developing a code generator for Web applications
- What options exist to be generated code?
  - » Dimensions of Web applications: Content, Hypertext, Presentation
  - » Programming languages: Java, C#, Ruby, PHP, ...
  - » Architectures: 2-layer, 3-layer, MVC, ActiveRecords, ...
  - » Frameworks: JSF, Spring, Struts, Hibernate, Ruby on Rails, ASP, ...
  - » Products: MySQL, Tomcat, WebLogic, ...
- Which combinations are appropriate?
  - » Experience gained in earlier projects
    - What has proven useful?
  - » Reference architectures

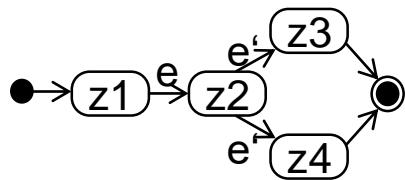
# Introduction

- What kind of code is generated?

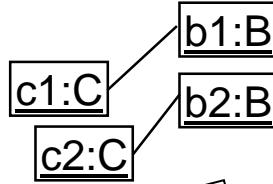
## Class diagrams



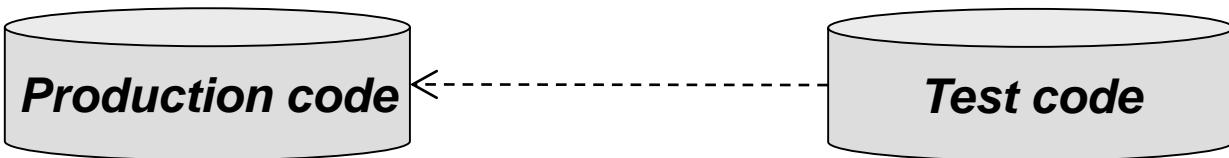
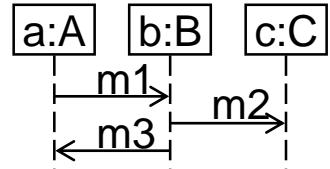
## State charts



## Object diagrams



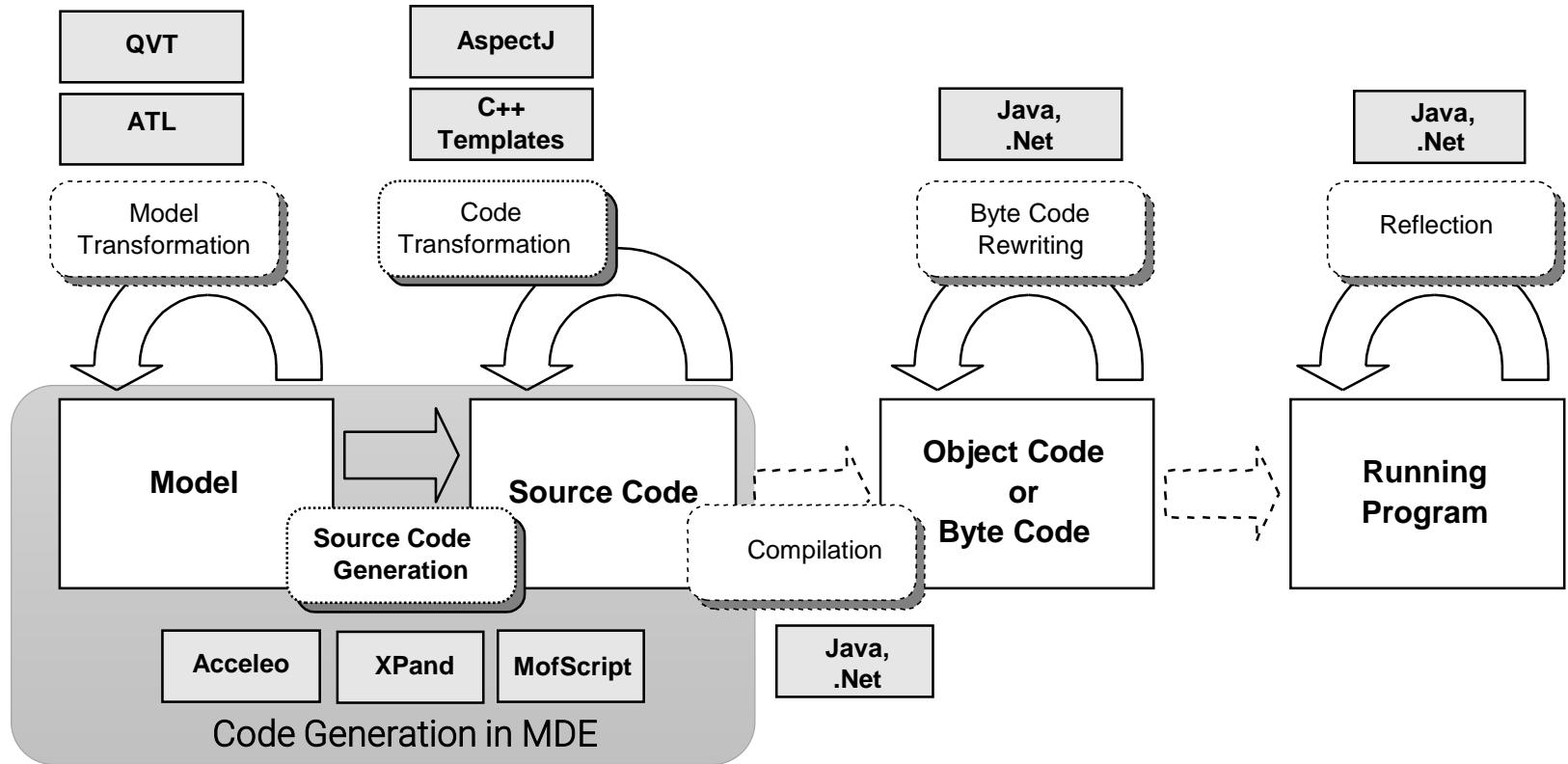
## Sequence diagrams



Picture based on Bernhard Rumpe: *Agile Modellierung mit UML*. Springer, 2012.

# Introduction

- Overview of generation techniques



Based on Markus Völter. A catalog of patterns for program generation. In *Proceedings of the 8th European Conference on Pattern Languages of Programs (EuroPLoP'03)*, pages 285–320, 2003.



# Introduction

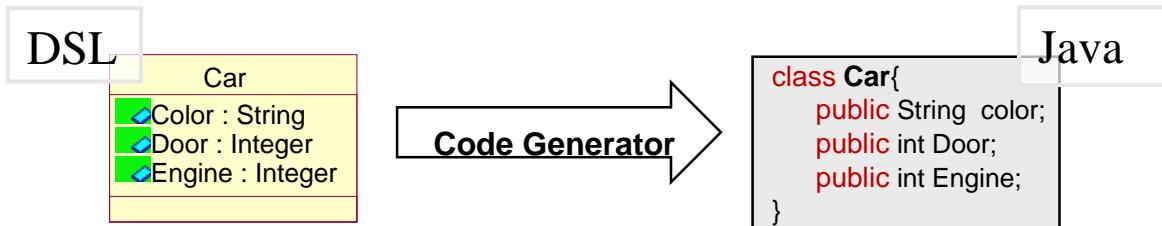
## Why code generation?

- **Code generation enables**
  - » Separation of application modeling and technical code
    - Increasing maintainability, extensibility, portability to new hardware, operating systems, and platforms
  - » Rapid prototyping
  - » Early and fast feedback due to demonstrations and test runs
- **Code generation enables to combine redundant code fragments in one source**
  - » Example: DDL, Hibernate, and Java Beans
    - may be specified in one Class Diagram

# Introduction

## Why code generation? – in contradiction to MDE? (1/2)

- Often no “real” model simulation possible
  - » UML environments mostly do not provide simulation features
    - However, they provide transparent transformation to C, C#, Java, ...
  - » UML Virtual Machines
    - Interpreter approach – spare code generation for certain platforms
    - Gets a new twist with fUML!
- Semantics of modeling languages, especially DSMLs, often defined by code generation





# Introduction

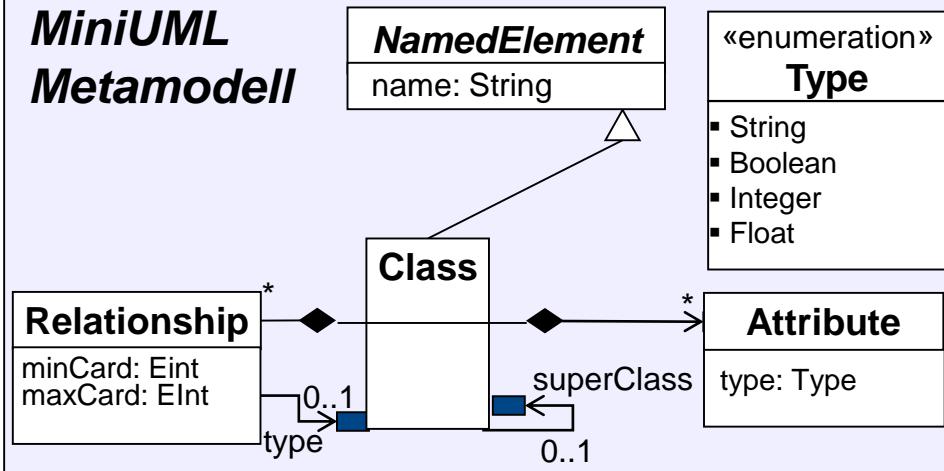
## Why code generation? – in contradiction to MDE? (2/2)

- Runtime environments are designed for programming languages
  - » Established frameworks available (Struts, Spring, Hibernate, ...)
  - » Systems depend on existing software (Web Services, DB)
  - » Extensions for code level often required (logging mechanism)
- Disadvantage: using models and code in parallel
  - » No single source of information – OUCH!
  - » Having the same information in two places may lead to inconsistencies, e.g., consider maintainability of systems



# Introduction

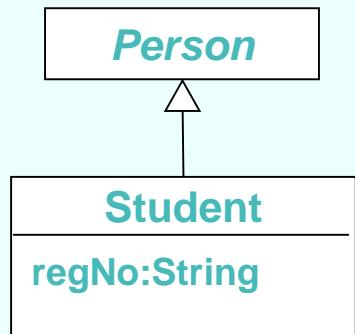
## MiniUML Metamodell



## MiniJava Grammar

```
ClassDec := Modifier "class" Identifier  
["extends" Identifier] ClassBody;  
  
AttributeDec := Modifier Type Identifier ";"  
  
MethodDec := Modifier ReturnType Identifier "("  
ParamList ")" "{" MethodBody "}"  
  
Identifier := {"a"--"z" | "A"--"Z" | "0"--"9"};
```

## MiniUML Modell



➤ **Model2Text**



## MiniJava Code

```
class Student extends Person{  
    private String regNo;  
    public void setRegNo (...) {  
        ...  
    }  
    public String getRegNo () {  
        ...  
    }  
}
```



# Programming languages

## Introduction – Code generation with Java based on EMF

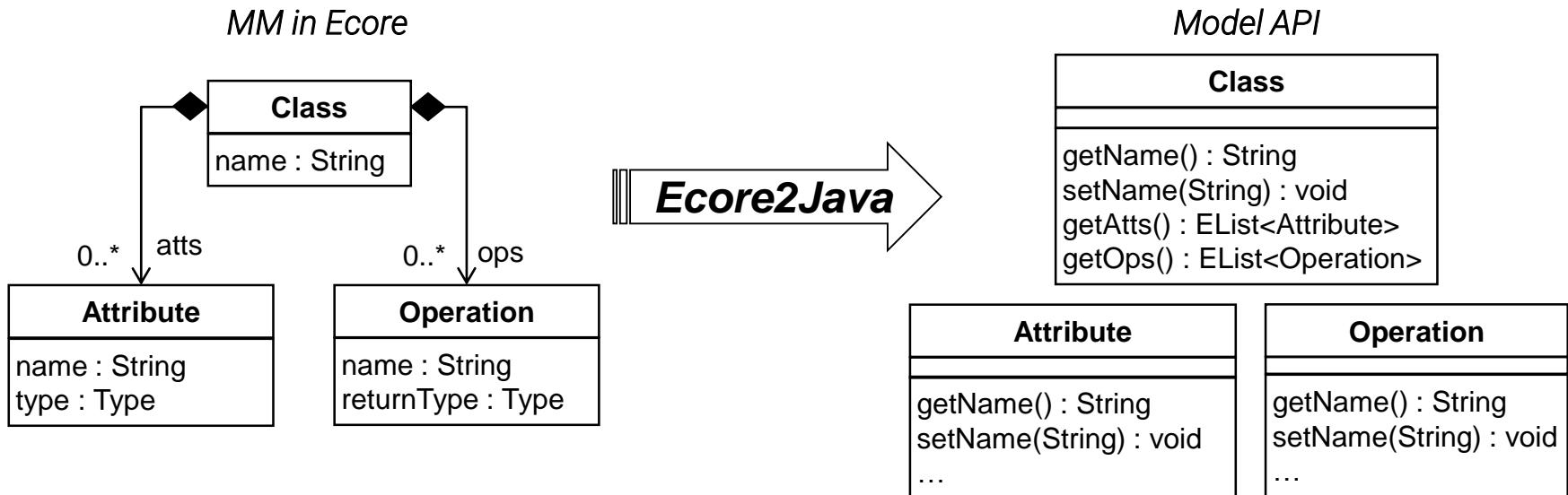
- Code generation may be realized using a traditional general purpose programming language, e.g., Java, C#, ...
- Models are de-serialized to an in-memory object graph
  - » Pre-defined XMI de-serializer provided by meta-modeling frameworks
  - » Out-of-the-box support in EMF
- Model API eases processing of models
  - » Generated automatically from metamodels
    - In EMF: .ecore -> .genmodel -> Java code
  - » If metamodel not available, you may use reflection



# Programming languages

## Model APIs for processing models

- Example: Ecore-based metamodel and automatically generated Java code (shown as UML Class Diagram)





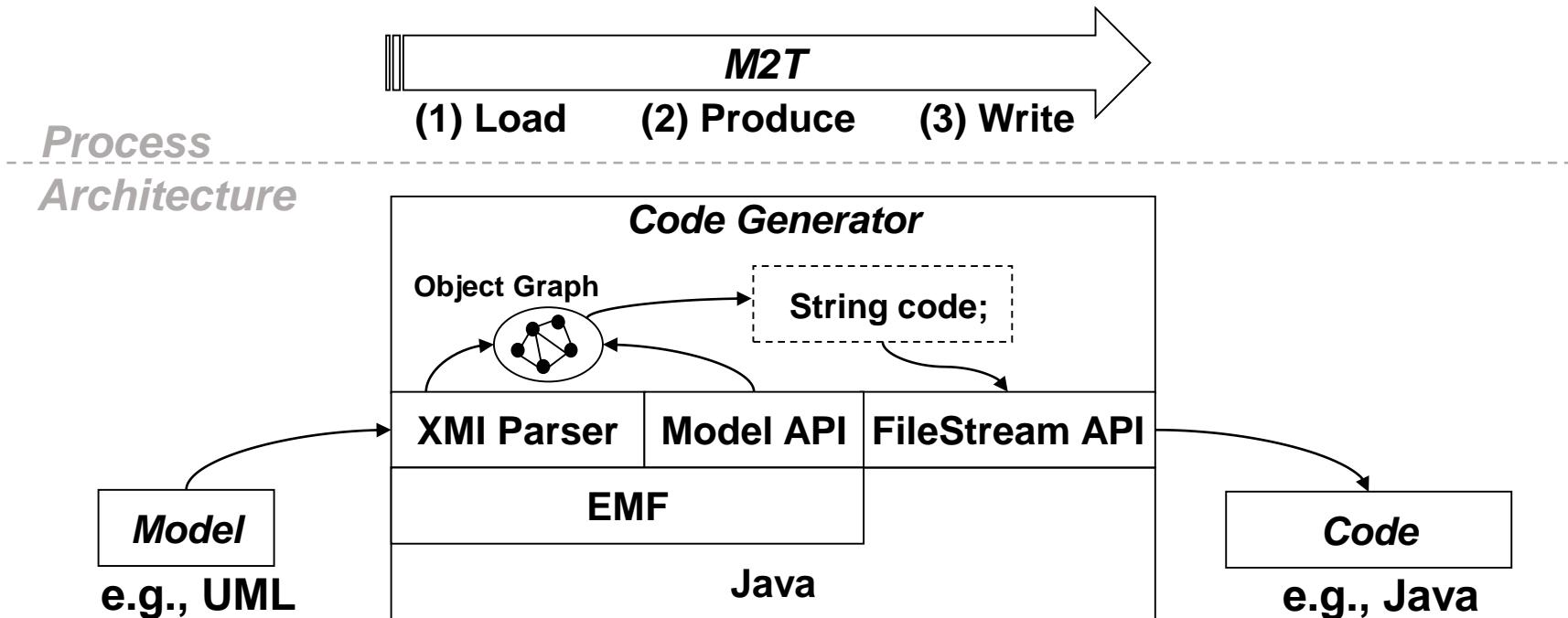
# Programming languages

## Code generation with Java: phases of code generation

- Load models
  - » Load XMI file into memory
- Process models and produce code
  - » Process models by traversing the model structure
  - » Use model information to produce code
  - » Save code into String variable
- Write code
  - » Persist String variable to a file using streams

# Programming languages

- Code generation with Java: Process and Architecture





# Programming languages

```
ResourceSet resourceSet = new ResourceSetImpl();  
Resource resource = resourceSet.getResource(URI.create("model.miniUML"));  
TreeIterator treeIter = resource.getAllContents();
```

(1) Load

```
while (treeIter.hasNext()) {  
    Object object = treeIter.next();  
    if (!object instanceof Class) continue;  
  
    Class cl = (Class) object;  
    String code = "class " + cl.getName() + "{";  
    // generate Constructor: code += ...  
    // generate Attributes: code += ...  
    // generate Methods: code += ...  
    code += "}";
```

Get all model elements

Query values via  
model API

(2) Produce

```
try {  
    FileOutputStream fos = new FileOutputStream(cl.getName() + ".java");  
    fos.write(code.getBytes());  
    fos.close();  
} catch (Exception e) {...}  
}
```

Create a file for  
each class

(3) Write



# Programming languages

- **Advantages**

- » No new languages have to be learned
- » No additional tool dependencies

- **Disadvantages**

- » Intermingled static/dynamic code
- » Non-graspable output structure
- » Lack of declarative query language
- » Lack of reusable base functionality



# M2T Transformation Languages... ...are Template based

- Templates are a well-established technique in software engineering
  - » Application domains: Text processing, Web Engineering, ...
  - » Example:

## *E-Mail Text*

Dear **Homer Simpson**,  
Congratulations! You have won ...

## *Template Text*

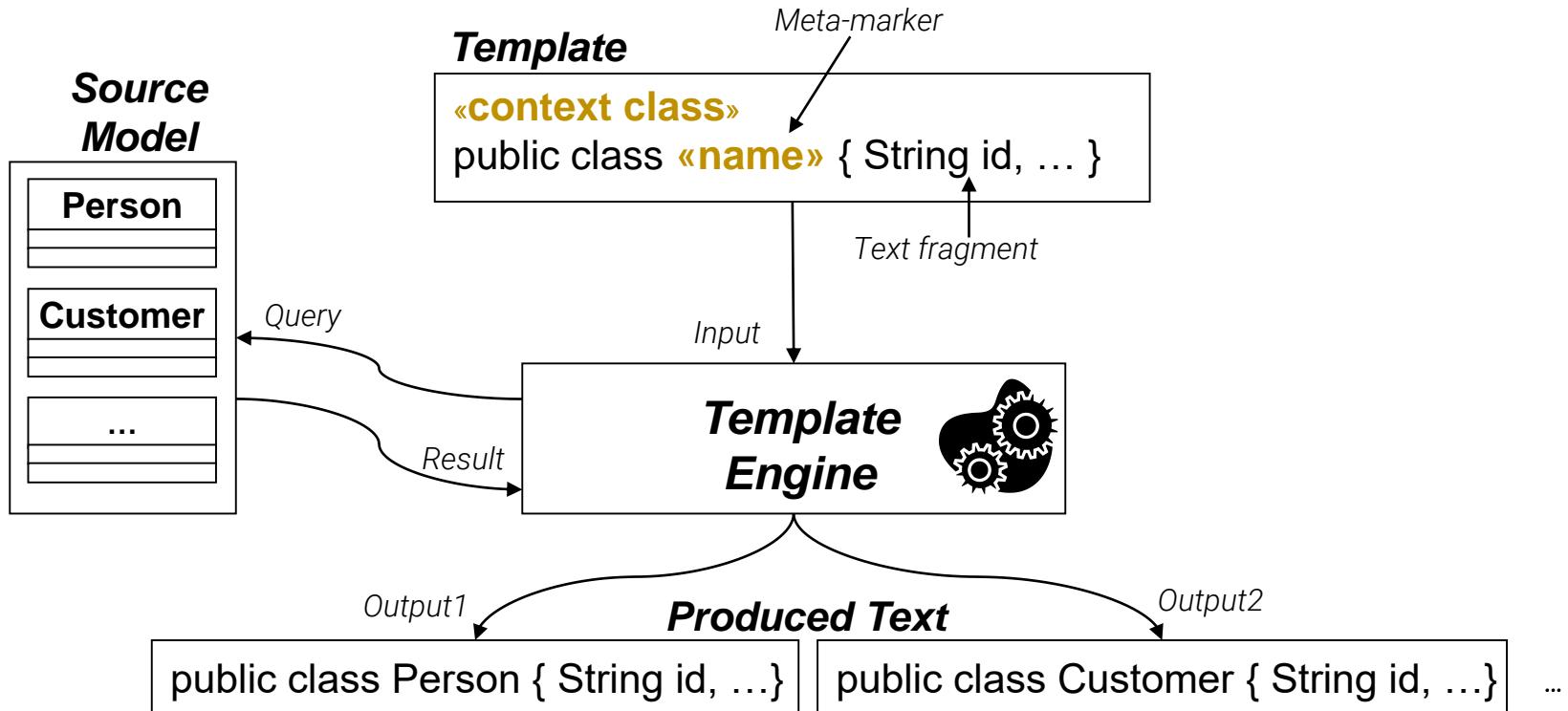
Dear «**firstName**» «**lastName**»,  
Congratulations! You have won ...

- Components of a template-based approach
  - » Templates
    - Text fragments and embedded meta-markers
  - » Meta-Markers query an additional data source
    - Have to be interpreted and evaluated in contrast to normal text fragments
    - Declarative model query: Query languages (OCL, XPath, SQL)
    - Imperative model query: Programming languages (Java, C#)
  - » Template Engine
    - Replaces meta-markers with data at runtime and produces output files

# M2T Transformation Languages

## Core Architecture

- Template-based approach at a glance





# M2T Transformation Languages

## Benefits

- **Separated static/dynamic code**
  - » Templates separate static code, i.e., normal text, from dynamic code that is described by meta-markers
- **Explicit output structure**
  - » Primary structure of the template is the output structure
  - » Computation logic is embedded in this structure
- **Declarative query language**
  - » OCL is employed to query the input models
- **Reusable base functionality**
  - » Support for reading in models, serialize text to files, ...



# M2T Transformation Languages Approaches

- A bunch of template languages for M2T transformation available
  - » JET, JET2
  - » Xpand Xtend
  - » MOFScript
  - » Acceleo
  - » XSLT
  - » ...



# Acceleo Introduction

- Acceleo is a mature implementation of the OMG M2T transformation standard
  - » Acceleo website: <http://www.eclipse.org/acceleo/>
  - » M2T Transformation standard: <http://www.omg.org/spec/MOFM2T>
- Template-based language
  - » Several meta-markers for useful for code generation available
- Powerful API supporting
  - » OCL
  - » String manipulation functions
  - » ...
- Powerful tooling supporting
  - » Editor, debugger, profiler, traceability between model and code, ...



# Acceleo Language Concepts

- **Module concept is provided**
  - » Imports the metamodels for the input models
  - » Act as container for templates
- **A template is always defined for a particular meta-class**
  - » Plus an optional pre-condition to filter instances
  - » Templates may call each other
  - » Templates may extended each other
  - » Templates contain text and provided meta-markers



# Acceleo Language Concepts

- Several meta-markers (called tags) are supported
- File Tag: To open and close files in which code is generated
- For/If Tag: Control constructs for defining loops and conditions
- Query Tag: Reusable helper functions
- Expression Tag: Compute values that are embedded in the output
- Protected Tag: Define areas that are not overridden by future generation runs

# Acceleo -Example

```
[module generateJavaClass('http://smvclm/1.0')]  
[query public getter(att : Attribute) : String = 'get'+att.name.toUpperFirst() /]  
[query public returnStatement(type: String) : String = if type = 'Boolean'  
    then 'return true;' else '...' endif /]  
[template public javaClass(aClass : Class)]  
[file (aClass.name.toUpperFirst()+'java', false, 'UTF-8')]
```

```
package entities;  
import java.io.Serializable;  
public class [aClass.name/] implements Serializable {  
    [for (att : Attribute | aClass.atts) separator ('\n')]  
        [javaAttribute(att)/]  
    [/for]  
    [for (op : Operation | aClass.ops) separator ('\n')]  
        [javaMethod(op)/]  
    [/for]  
}
```

```
[/file]  
[/template]  
...  
[Close output file]
```

The diagram illustrates the Acceleo template language with various annotations:

- Import metamodel (root package)**: Points to the first line of code: [module generateJavaClass('http://smvclm/1.0')].
- Query**: Points to the [query] blocks.
- Open output file**: Points to the [file] block.
- Template definition**: Points to the [template] block for the class definition.
- Meta class**: Points to the [meta] block for the class definition.
- Static Text**: Points to the static Java code within the template.
- Template Call**: Points to the [for] loop that calls the [javaAttribute] template.
- Loop**: Points to the [for] loop that iterates over operations.
- Protected Area**: Points to the protected area within the template definition.
- Expression**: Points to the expression within the template definition.

```
[template public javaAttribute(att : Attribute)]  
private [att.type/] [att.name/];
```

```
public [att.type/] [att.getter()/] {  
    return [att.name/];  
}
```

```
[/template]
```

```
[template public javaMethod(op : Operation)]  
public [op.type/] [op.name/] () {
```

```
// [protected (op.name)]  
// Fill in the operation implementation  
[returnStatement(op.type)/]  
// [/protected]
```

```
[/template]
```



# Acceleo Protected Areas

- Protected areas are not overridden by the next generator run
- They are marked by comments
- Their content is merged in the newly produced code
  - » If the right place cannot be found, warning is given!
- Example
  - » 

```
public boolean checkAvailability(){  
    // Start of user code checkAvailability  
    // Fill in the operation implementation here!  
    return true;  
    // End of user code  
}
```



## Abstracting Templates

- To ensure that generated code is accepted by developers (cf. Turing test for code-generation), familiar code should be generated
  - » Especially when only a partial code generation is possible!
- Abstract code generation templates from reference code to have known structure and coding guidelines considered
- Acceleo supports dedicated refactorings to transform code into templates
  - » E.g., substitute String with Expression Tag



## Generating step-by-step

- Divide code generation process into several steps
  - » Same applies as for M2M transformations!
- Transformation chains may use a mixture of M2M and M2T transformations
  - » To keep the gap between the models and the code short
- If code generators exists, try to produce their required input format with simpler M2M or M2T transformations
  - » E.g., code generator for flat state machines, transform composite state machines to flat ones and run existing code generator



## Separating transformation logic from text

- Separate complex transformation logic from text fragments
- Use queries or libraries that are imported to the M2T transformation
- By this, templates get more readable and maintainable
- Queries may be reused



## Mastering code layout

- Code layout is determined by the template layout
- Challenging to produce code layout when several control structures such as loops and conditionals are used in the template
  - » Special escape characters for line breaks used for enhancing the readability of the template are provided
- Alternative
  - » Use code beautifiers in a post-processing step
  - » Supported by Xpand for Java/XML out-of-the-box



## Be aware of model/code synch problems

- Protected areas help save code in succeeding generator runs
- Code contained in protected areas is not always automatically integrated in the newly generated code
  - » Assume a method is renamed on model level
  - » Where to place the code of the method implementation?
  - » Which identifier to use for identifying a protected area?
  - » Natural or artificial identifiers?
- Model refactorings may be replayed on the code level before the next generator run is started
  - » Code in protected areas may also reflect the refactorings!

## Code Generation by M2M + TCS

- Code Generation achievable through applying a M2M transformations to a programming language metamodel
- If a TCS is available for the programming language metamodel, the resulting model may be directly serialized into text
- Only recommended when
  - » programming language metamodel + TCS are already available
  - » fully code generation is possible

