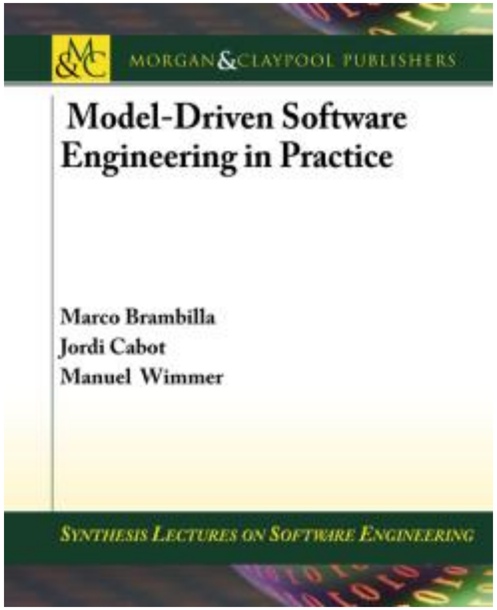
MORGAN & CLAYPOOL PUBLISHERS

Chapter 7

DEVELOPING YOUR OWN MODELING LANGUAGE

Teaching material for the book
Model-Driven Software Engineering in Practice
by Marco Brambilla, Jordi Cabot, Manuel Wimmer.
Morgan & Claypool, USA, 2012.



Content

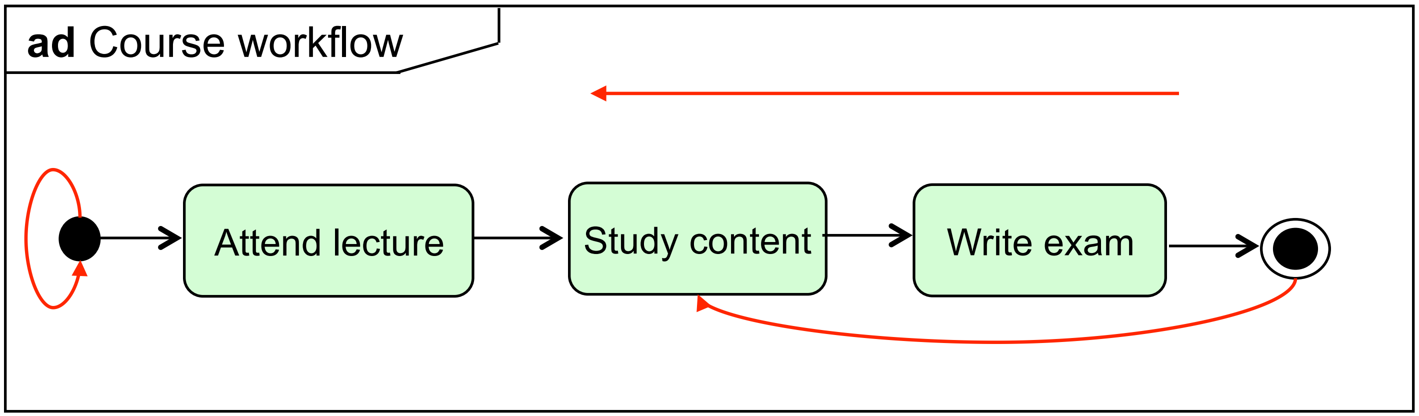
- Introduction
- Abstract Syntax
- Graphical Concrete Syntax
- Textual Concrete Syntax

INTRODUCTION

Introduction

What to expect from this lecture?

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



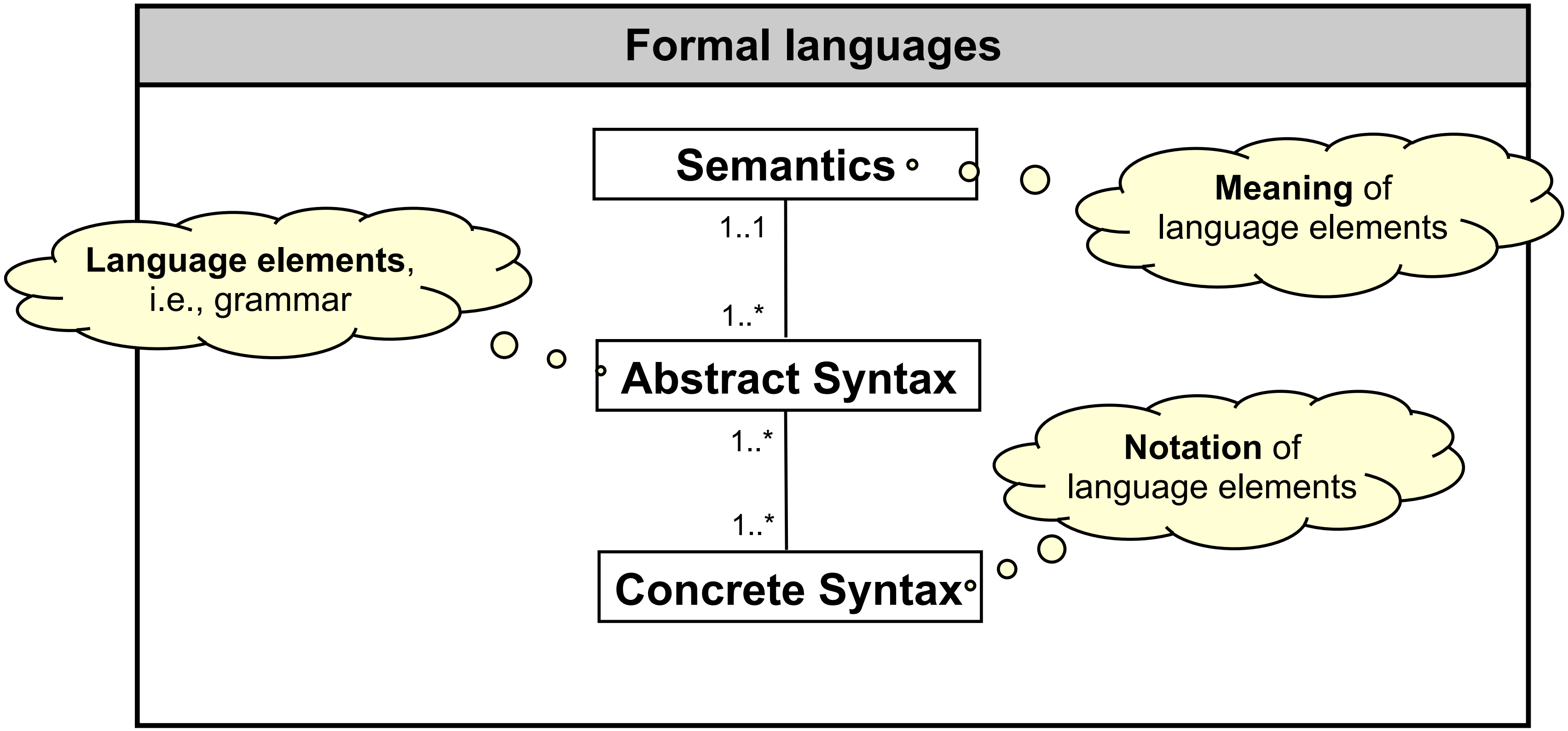
- **Question**: Is this UML Activity diagram **valid**?
- **Answer**: Check the **UML metamodel**!
  - Prefix „meta“: an operation is applied to itself
  - Further examples: meta-discussion, meta-learning, …

- **Aim of this lecture**: Understand **what** is meant by the term „metamodel“ and **how** metamodels are **defined**.

Introduction

Anatomy of formal languages 1/2

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



Introduction

Anatomy of formal languages 2/2

- **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

Or: Metamodeling – Old Wine in new Bottles?

- **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

Overview

- 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

option | sequence | non-terminal

```
Java        := [PackageDec]  {ImportDec}  ClassDec;
PackageDec    := "package"  QualifiedIdentifier;
ImportDec    := "import"  QualifiedIdenfifier;
ClassDec    := Modifier  "class"  Identifier  ["extends" Identifier]
                  ["implements" IdentifierList]  ClassBody;
```

production rule | terminal

Programming languages

Example: MiniJava

- Grammar

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

Meta-architecture layers

- Four-layer architecture

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

Definition of EBNF in EBNF – EBNF grammar (reflexive)

M3-Layer

Java := [PackageDec]
{ImportDec} ClassDec;
PackageDec := "**package**"
QualifiedIdentifier; …

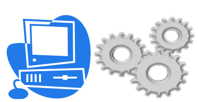
Definition of Java in EBNF – Java grammar

M2-Layer

```
package mdse.book.example;

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**

element

contentParticle

1..*

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

1..1

attributes

0..1

XML-based languages

Example: Cookbook DTD

- DTD

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

- XML

```
<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>
```

- Validation
    - XML Parser: Xerces, …

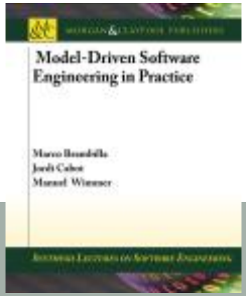
XML-based languages

Meta-architecture layers

- Five-layer architecture (was revised with XML-Schema)

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>**mdse.book.example**</packageDec> <classDec name=„**Student**“ extends=„**Person**“/> </javaProg>	**XML – conform to the DTD**	**M1-Layer**
Concrete entities (e.g.: Student “Bill Gates”)		**M0-Layer**

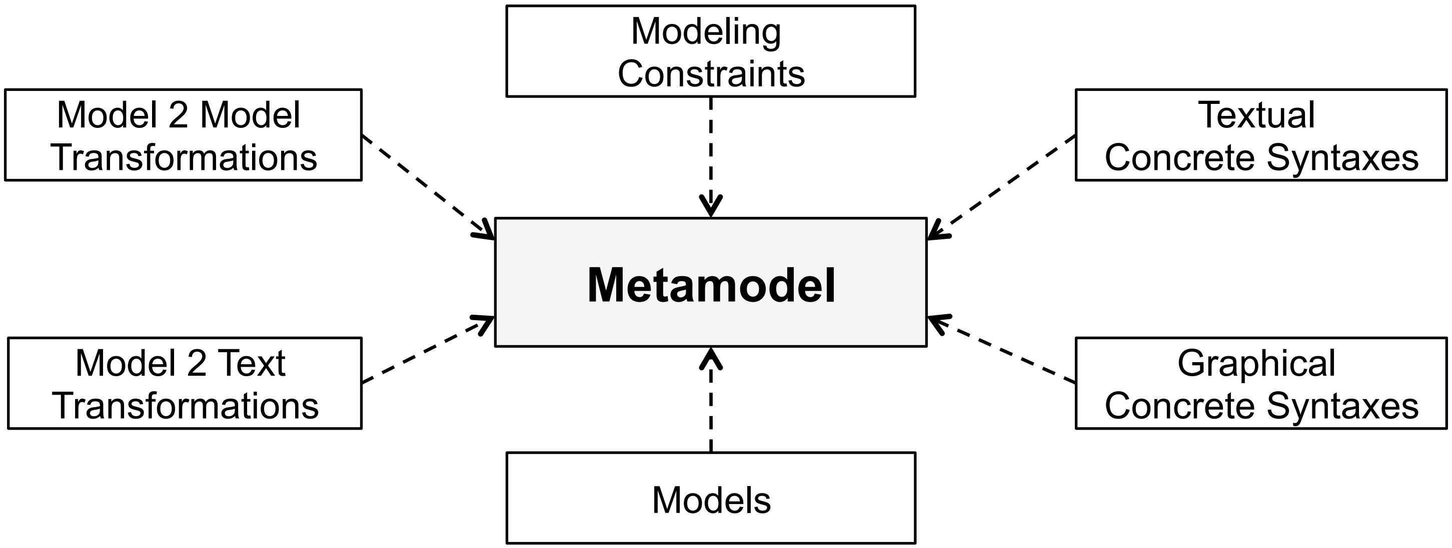
ABSTRACT SYNTAX



Introduction

Spirit and purpose of metamodeling 1/3

- **Metamodel-centric language design:**
  All language aspects base on the abstract syntax of the language defined by its metamodel



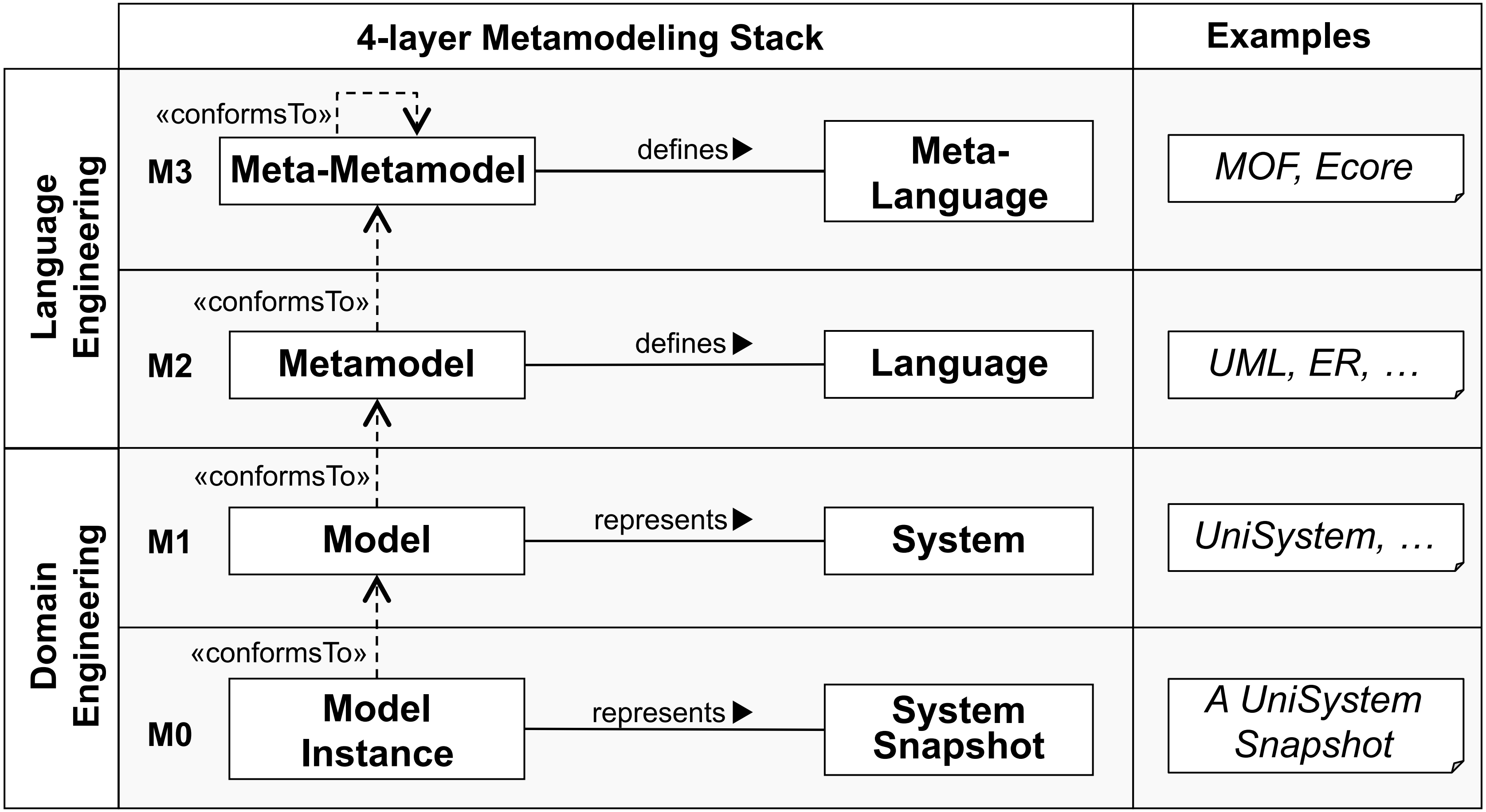
Introduction

Spirit and purpose of metamodeling 2/3

- **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.

Introduction

Spirit and purpose of metamodeling 3/3



Metamodel development process

Incremental and Iterative

Modeling domain analysis	Modeling language design	Modeling language validation
Identify purpose, realiza-tion, and content of the modeling language	**Formalize** modeling language by defining a metamodel	**Instantiate** metamodel by modeling reference models
Sketch reference modeling examples	**Formalize** modeling constraints using OCL	**Collect** feedback for next iteration

MOF - Meta Object Facility

Introduction 1/3

- **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

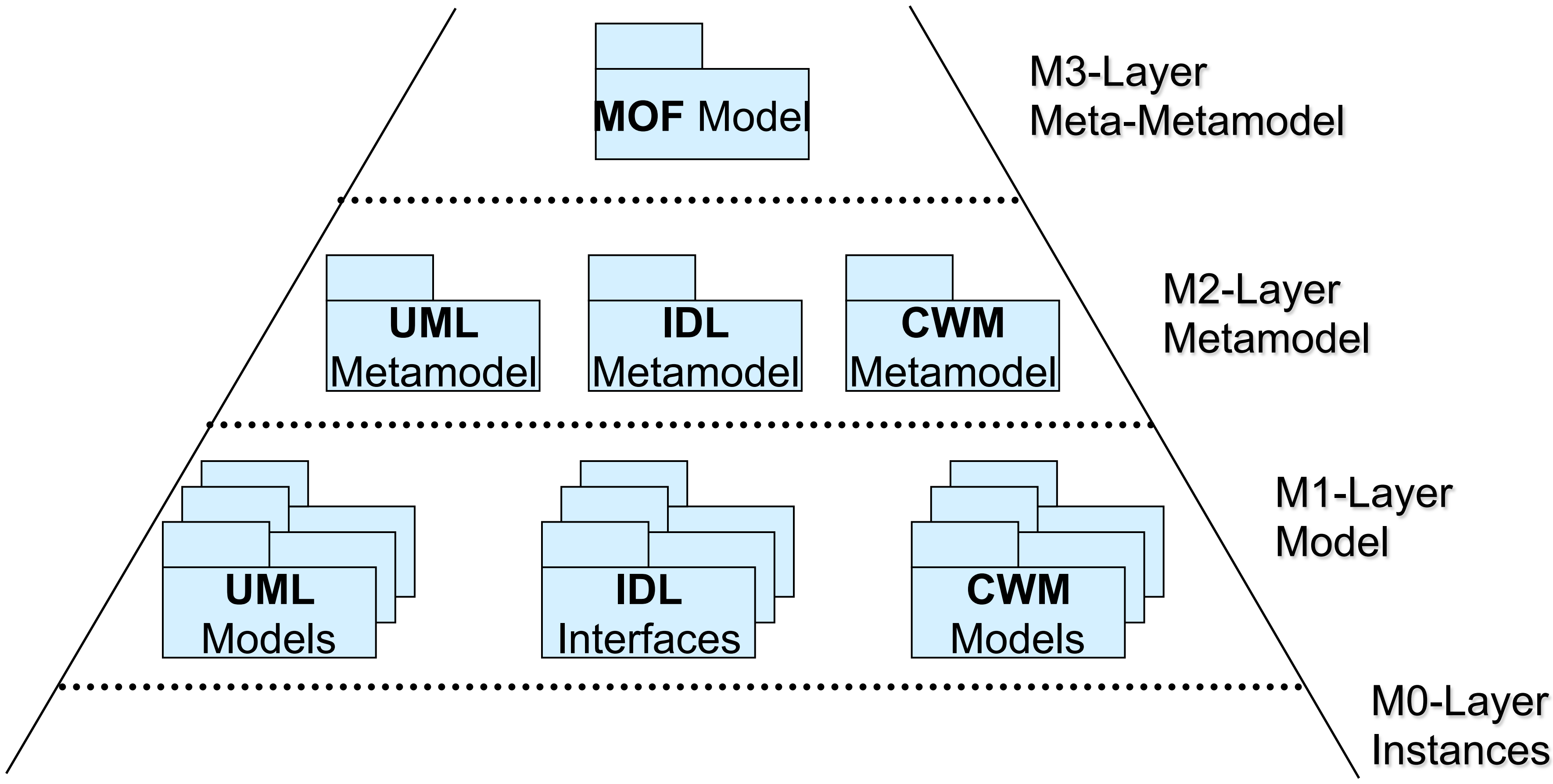
Introduction 2/3

- 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

Introduction 3/3

- OMG language definition stack



Why an additional language for M3

… isn‘t UML enough?

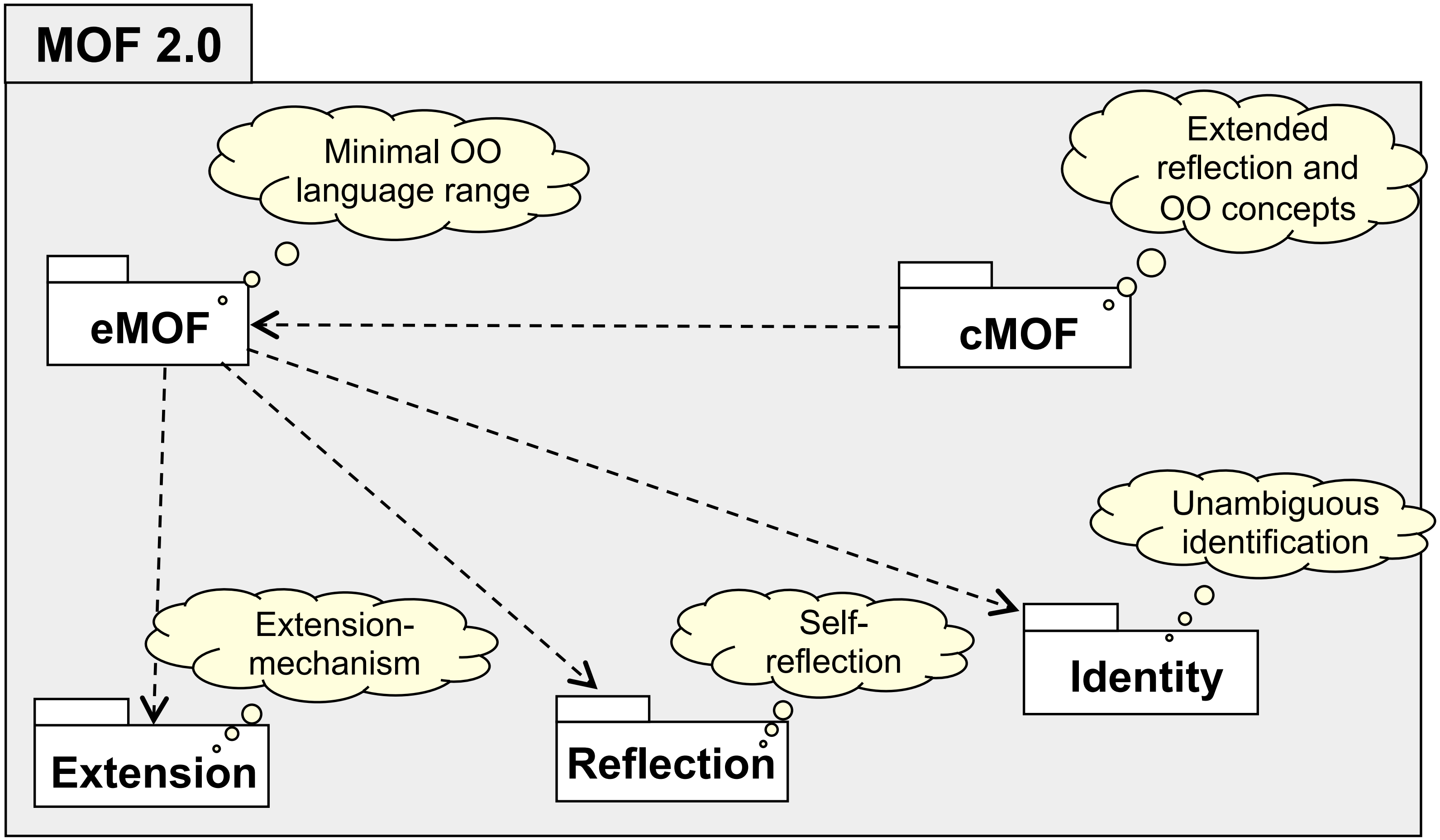
- **MOF** only a **subset** of **UML**
  - MOF is **similar** to the UML class diagram, but much more limited
  - No n-ary associations, no association classes, …
  - No overlapping inheritance, interfaces, dependencies, …

- 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

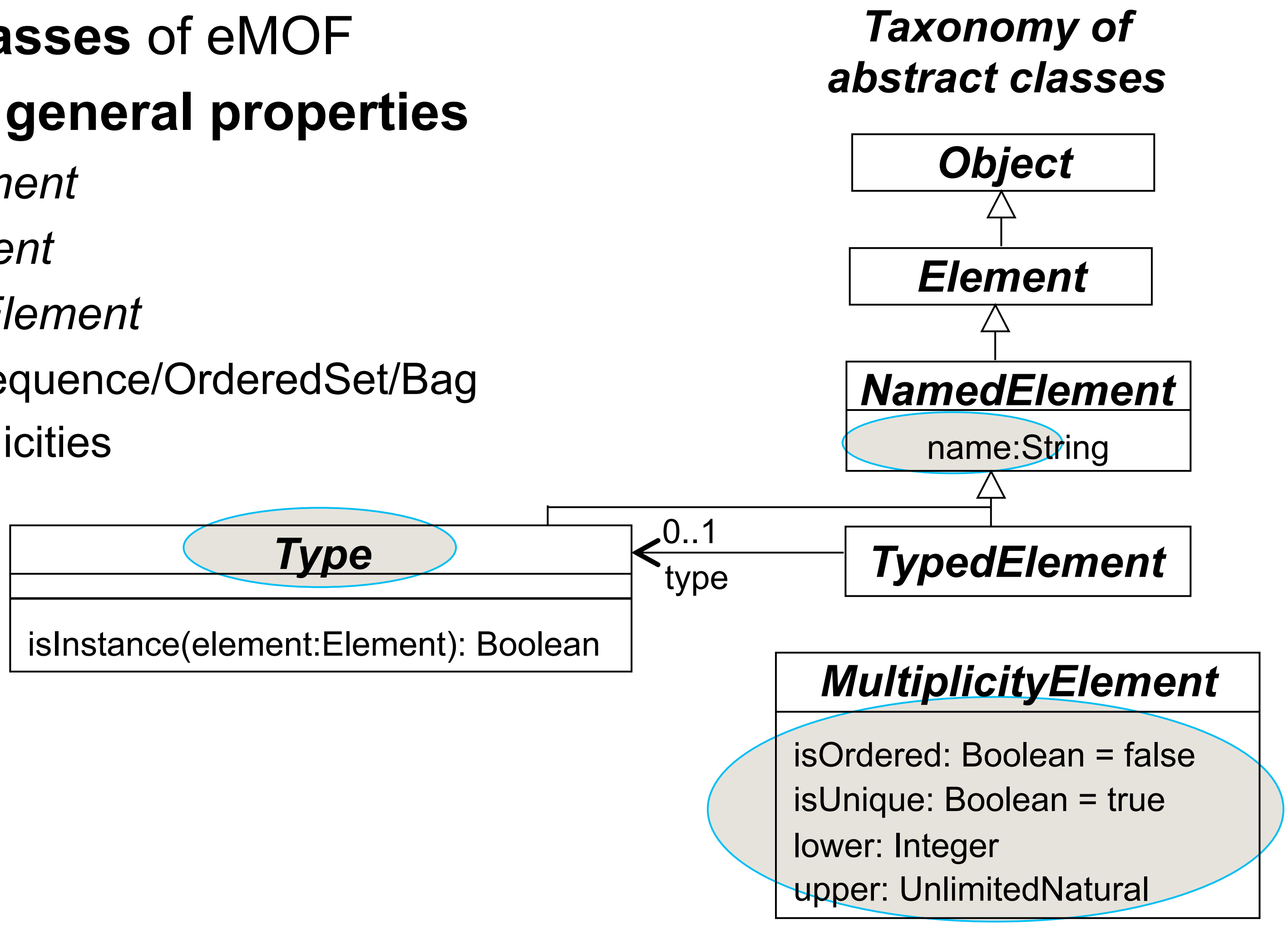
Language architecture of MOF 2.0



MOF – Meta Object Facility

Language architecture of MOF 2.0

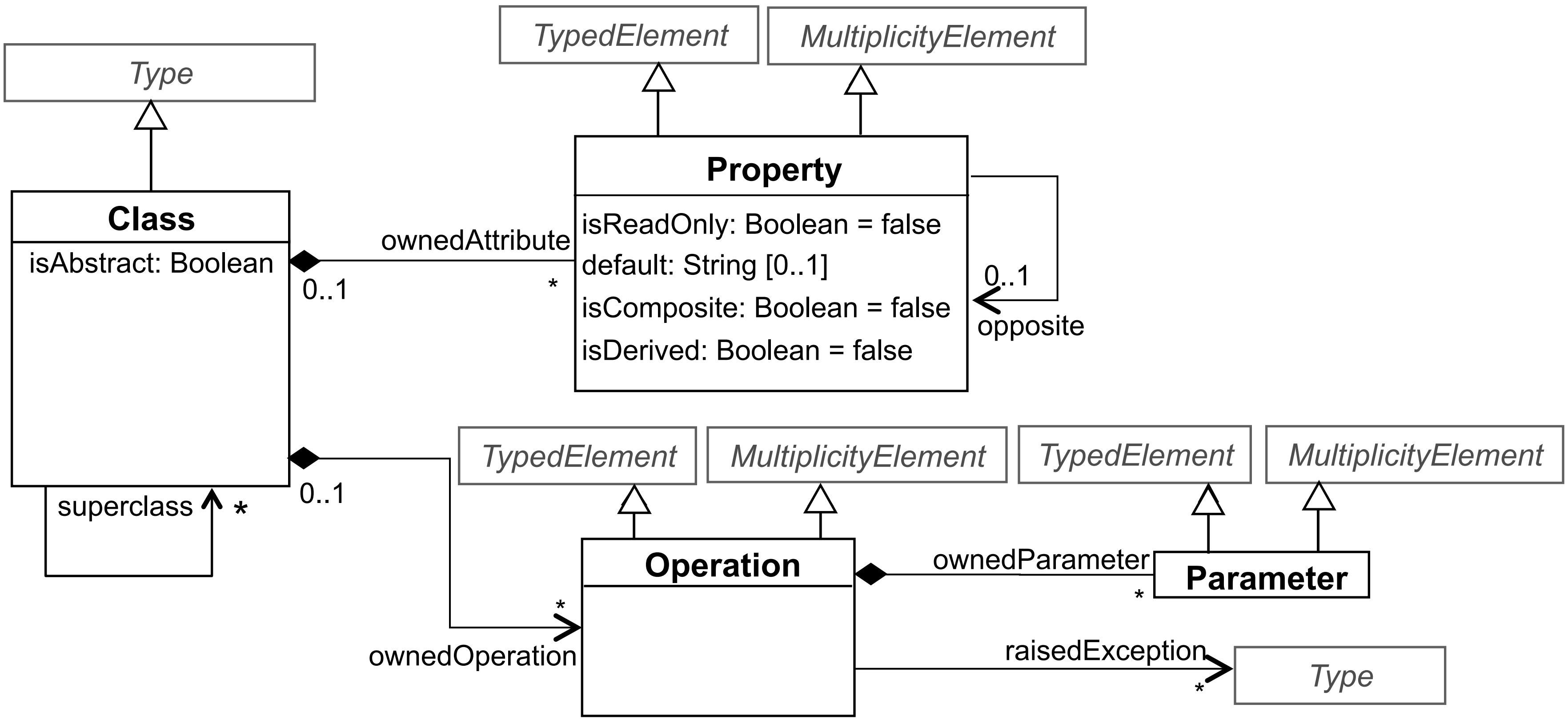
- **Abstract classes** of eMOF
- Definition of **general properties**
  - *NamedElement*
  - *TypedElement*
  - *MultiplicityElement*
    - Set/Sequence/OrderedSet/Bag
    - Multiplicities



MOF – Meta Object Facility

Language architecture of MOF 2.0

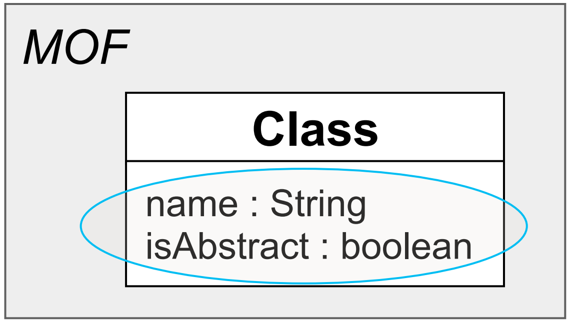
- **Core** of eMOF
  - Based on object-orientation
  - Classes, properties, operations, and parameters

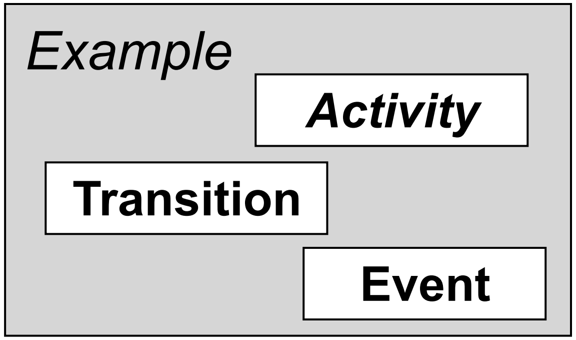


MOF – Meta Object Facility

Classes

- 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!

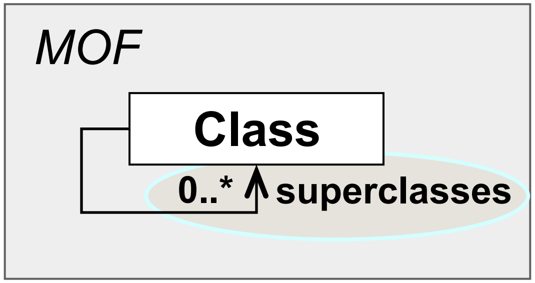


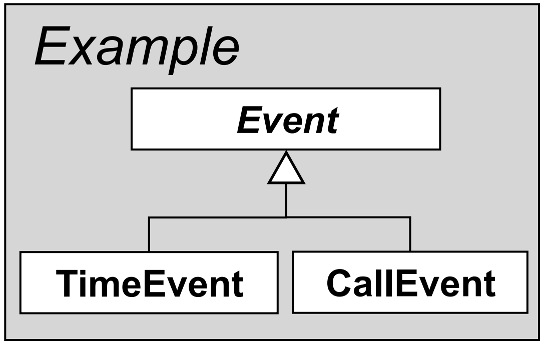


MOF – Meta Object Facility

Generalization

- **Generalization**: relationship between
  - a **specialized class** (*subclass)* and
  - a **general class** (*superclass*)
- Subclasses **inherit** properties of their superclasses and may add further properties
- Discriminator: „virtual" attribute used for the **classification**
- **Disjoint** (non-overlapping) generalization
- **Multiple inheritance**

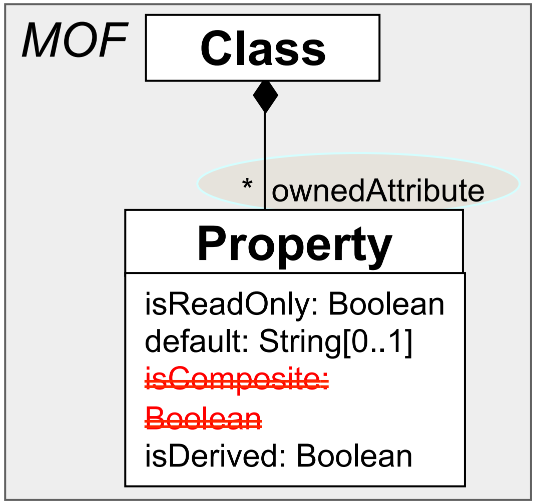


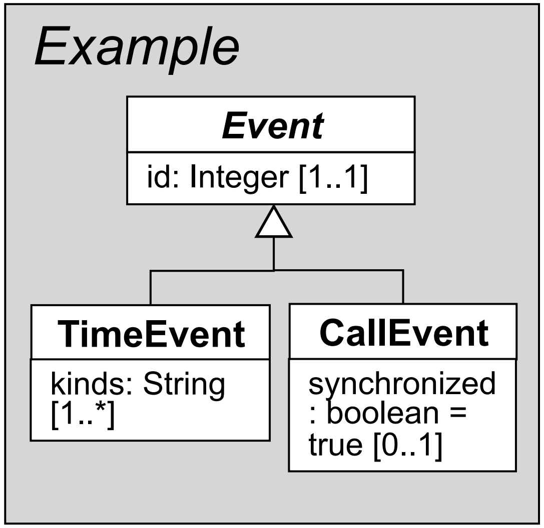


MOF – Meta Object Facility

Attributes

- **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

Associations

- 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

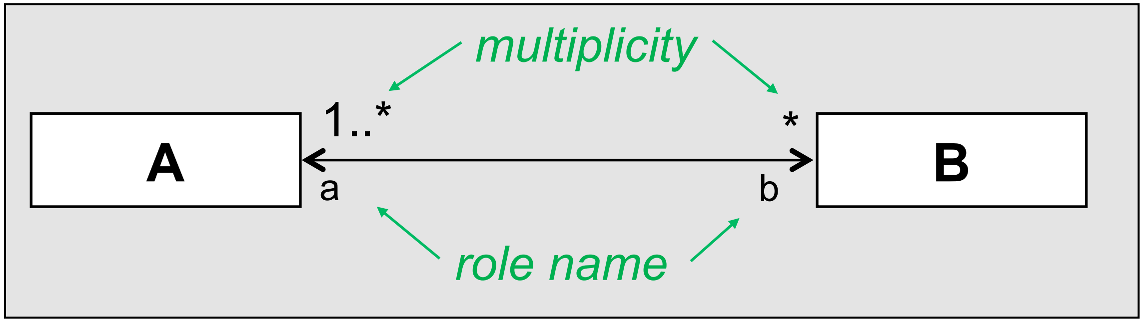
- **Binary associations** consist of **two roles** whereas each role has
  - **Role name**
  - **Multiplicity** limits **the number of** partner objects of an object

- **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

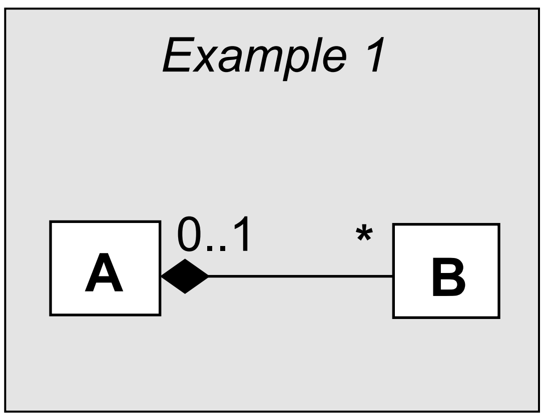
MOF – Meta Object Facility

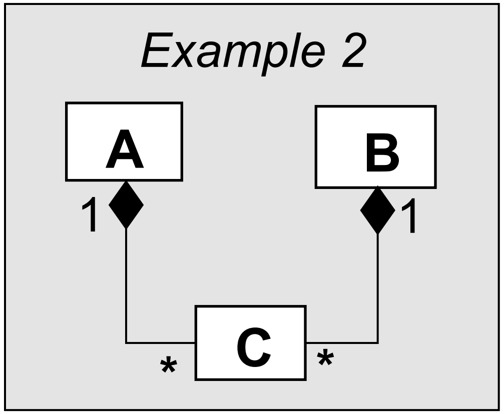
Associations - Examples

- **Association**

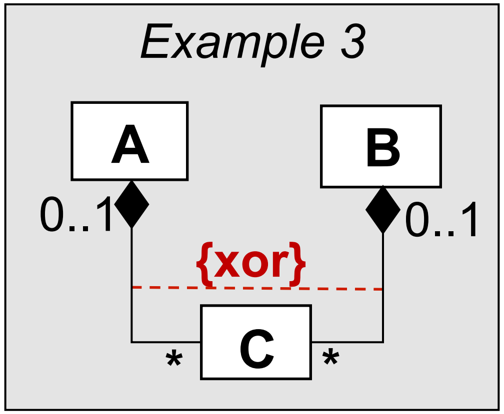


- **Composition**





Syntax ✓
Semantics ✗

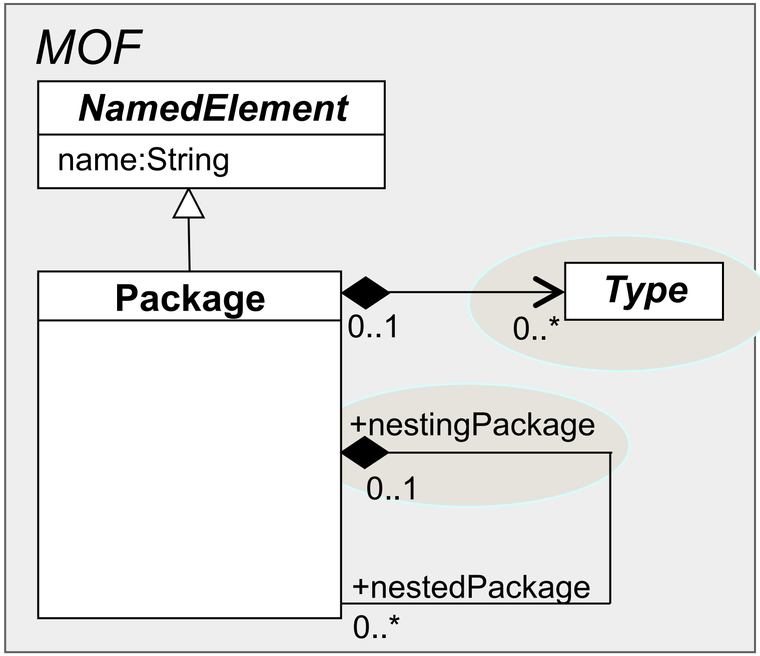


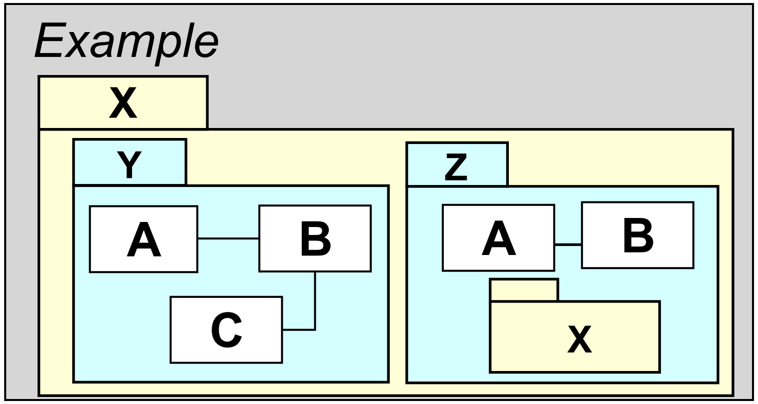
Syntax ✓
Semantics ✓

MOF – Meta Object Facility

Packages

- 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 in **one** package

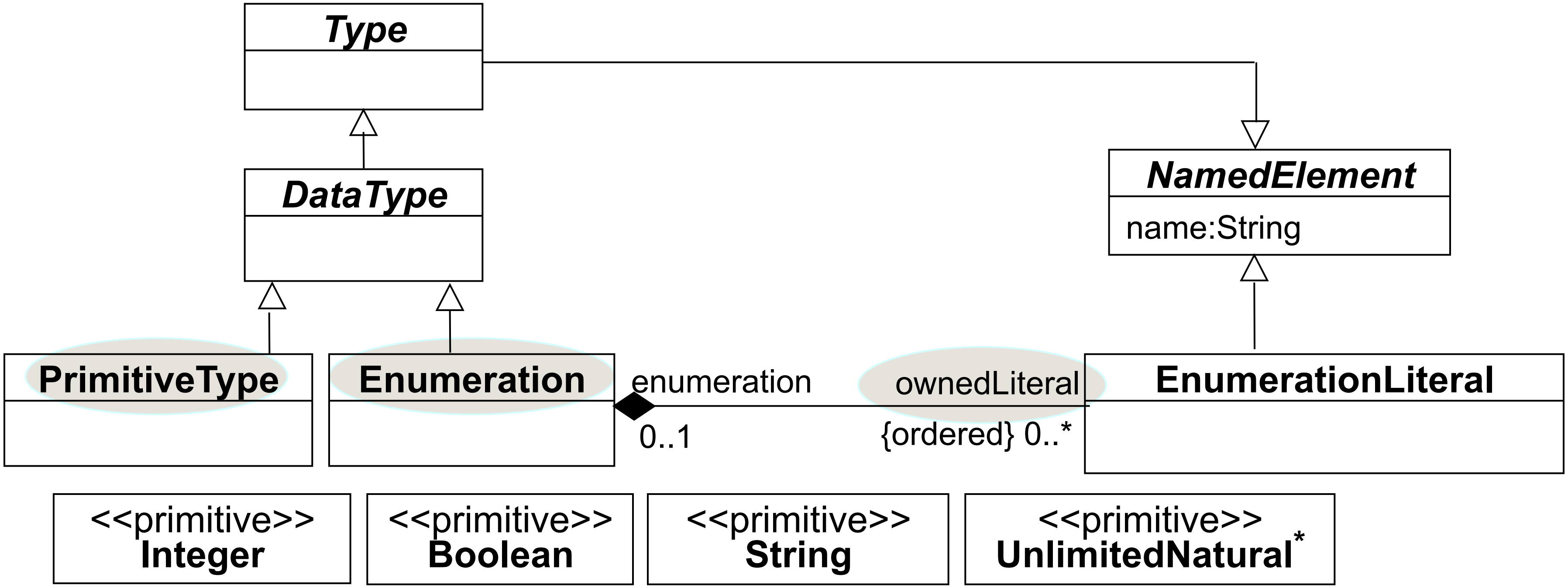




MOF – Meta Object Facility

Types 1/2

- **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

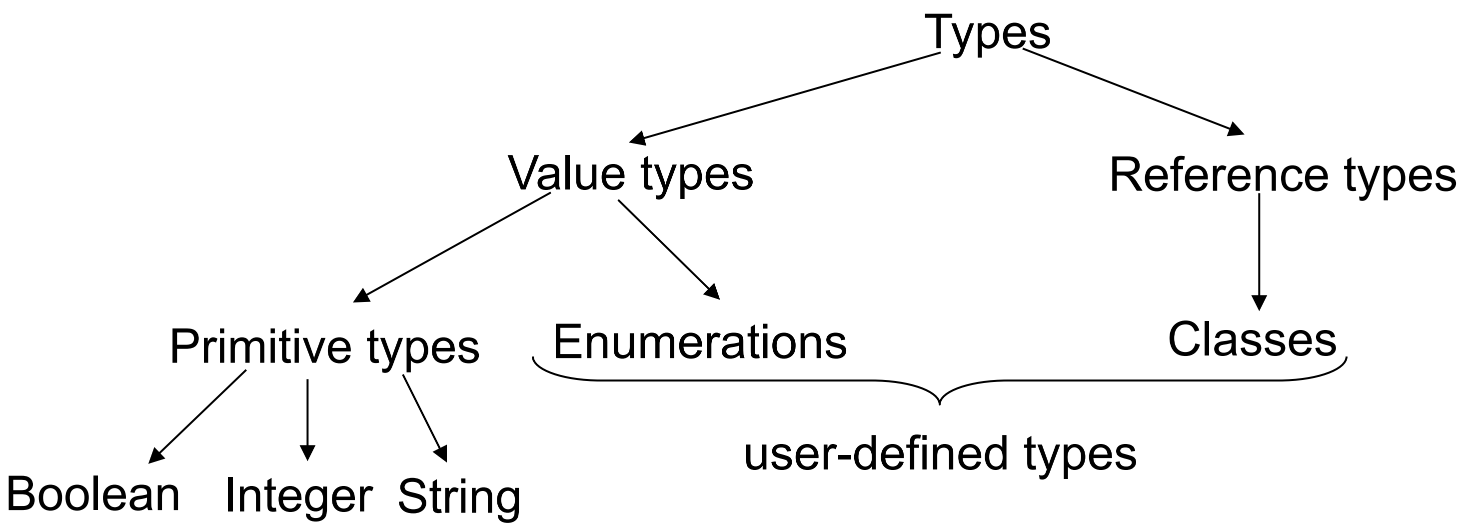


*) represents *unlimited number (asterisk)* – only for the definition of the **upper limits** of multiplicities

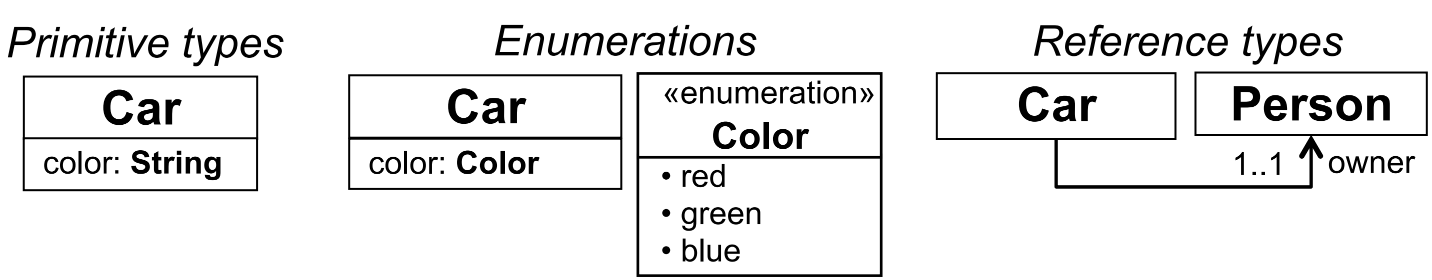
MOF – Meta Object Facility

Types 2/2

- 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

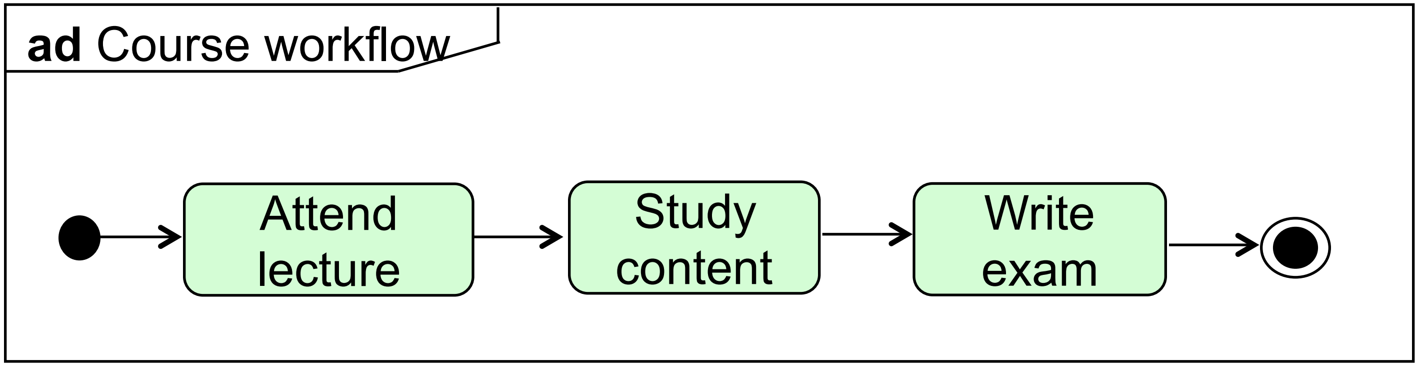


- **Examples**



Example 1/9

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



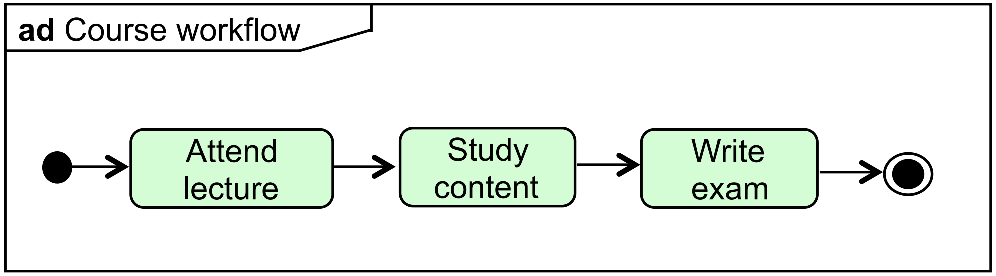
- Question: How does a possible metamodel to this language look like?

- Answer: apply metamodel development process!

Example 2/9

Identification of the modeling concepts

Example model = Reference Model



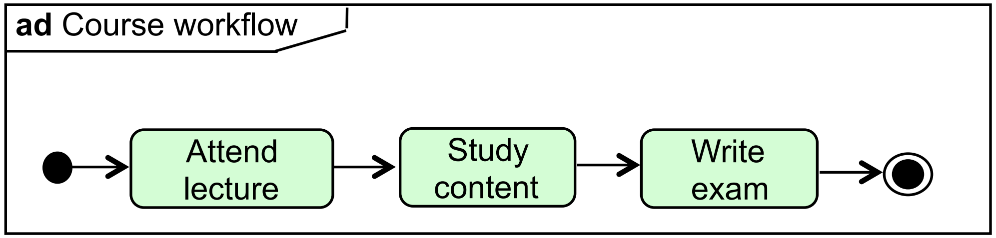
Notation table

Syntax	***Concept***
ad name	ActivityDiagram
◉	FinalNode
●	InitialNode
name	Activity
→	Transition

Example 3/9

Determining the properties of the modeling concepts

Example model

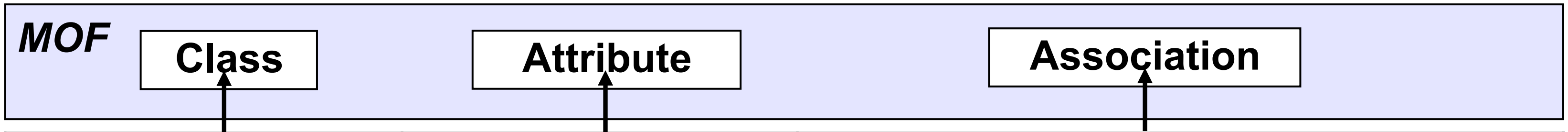


Modeling concept table

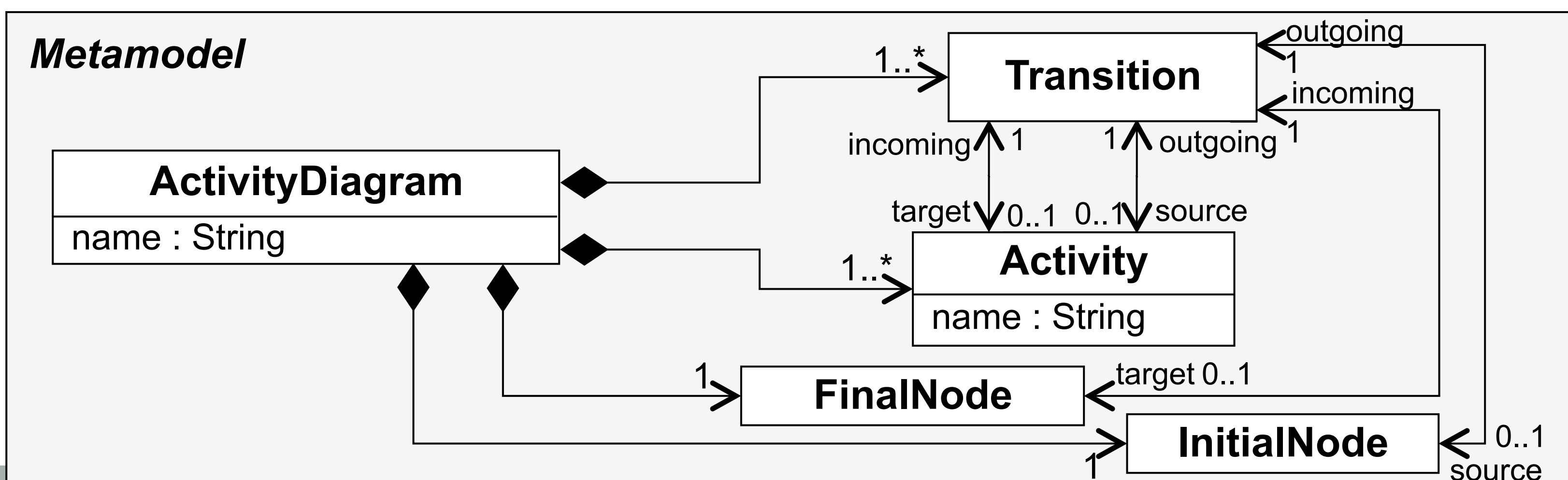
Concept	***Intrinsic properties***	***Extrinsic properties***
ActivityDiagram	Name	1 *InitialNode* 1 *FinalNode* Unlimited number of *Activities* and *Transitions*
FinalNode	-	Incoming *Transitions*
InitialNode	-	Outgoing *Transitions*
Activity	Name	Incoming and outgoing *Transitions*
Transition	-	Source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

Example 4/9

Object-oriented design of the language

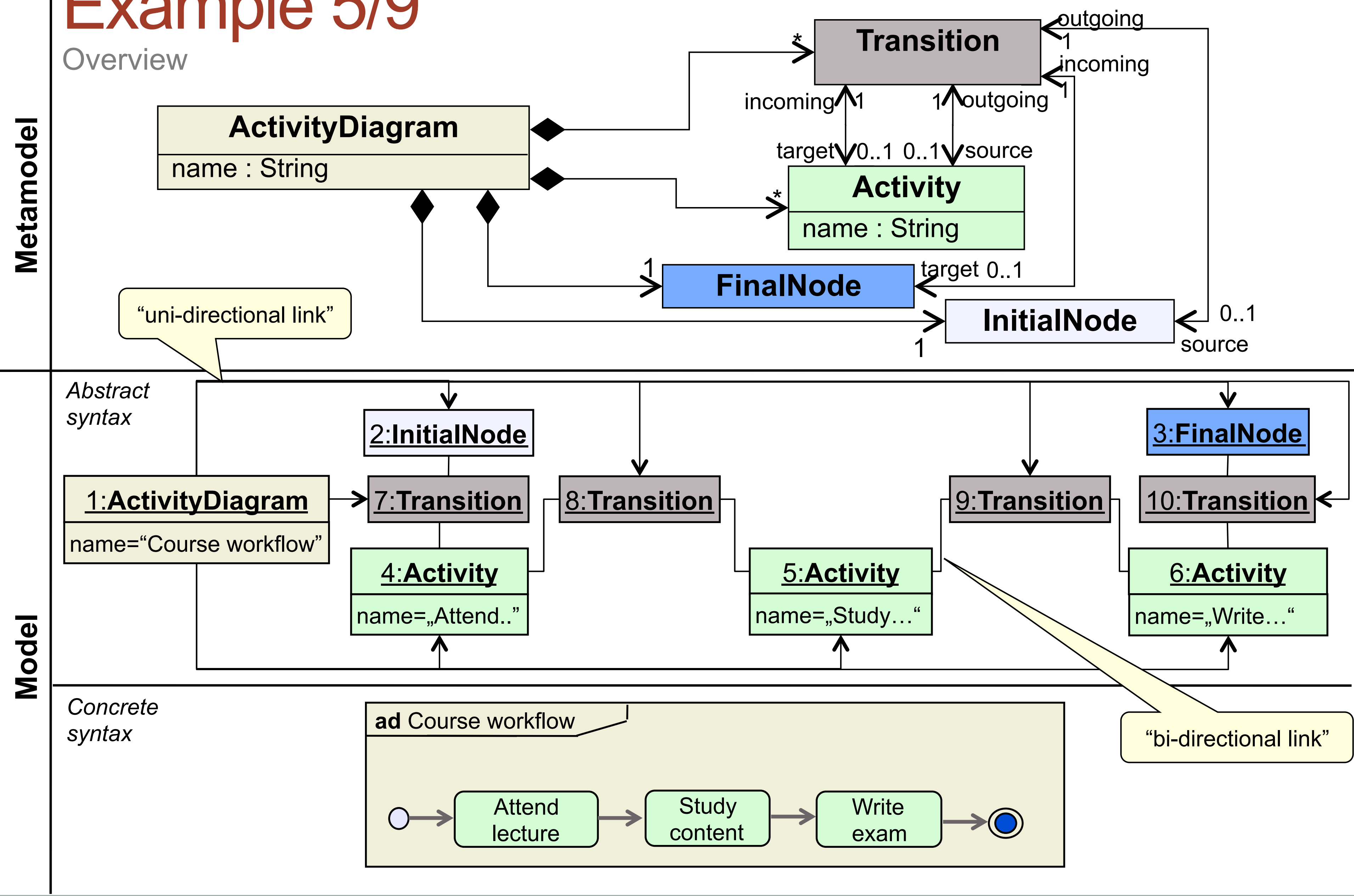


Concept	Intrinsic properties	Extrinsic properties
ActivityDiagram	Name	1 *InitialNode* 1 *FinalNode* Unlimited number of Activities and Transitions
FinalNode	-	Incoming *Transition*
InitialNode	-	Outgoing *Transition*
Activity	Name	Incoming and outgoing *Transition*
Transition	-	Source node and target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



Example 5/9

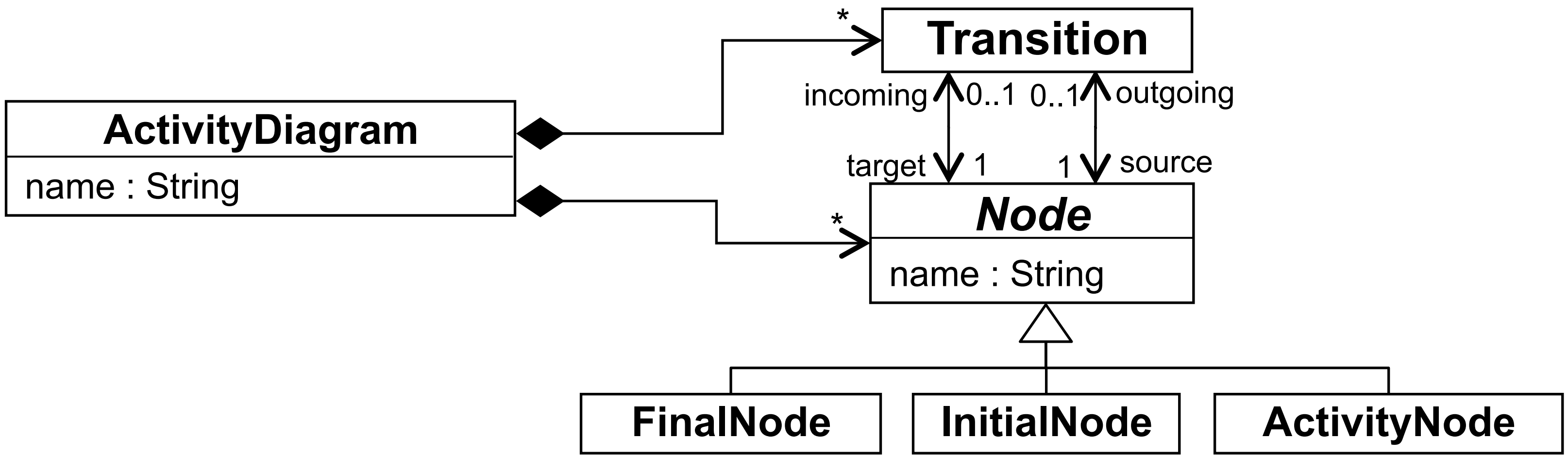
Overview



Example 6/9

Applying refactorings to metamodels

Metamodel



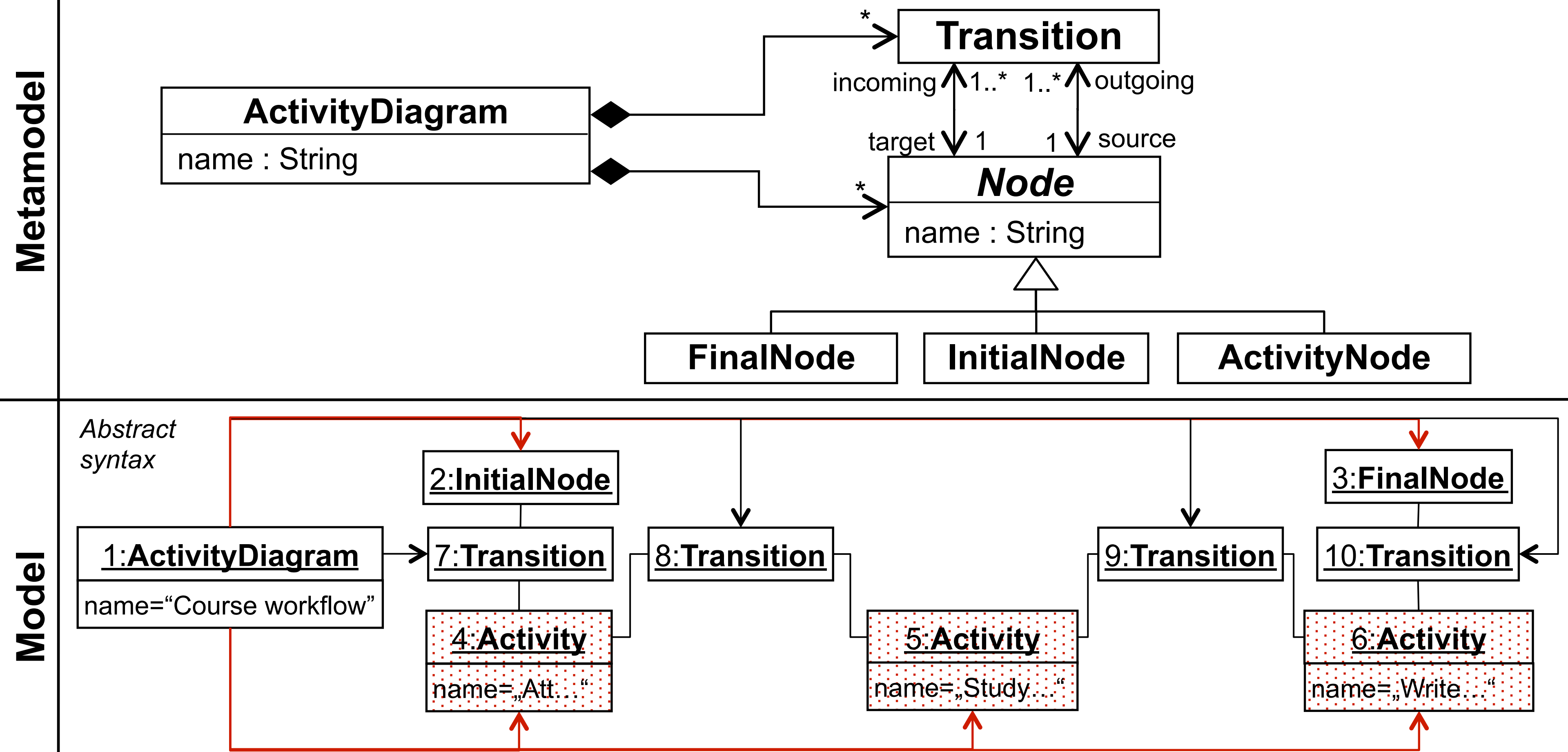
OCL Constraints

```
context ActivityDiagram
inv: self.nodes -> exists(n|n.isTypeOf(FinalNode))
inv: self.nodes -> exists(n|n.isTypeOf(InitialNode))

context FinalNode
inv: self.outgoing.oclIsUndefined()

context InitialNode
inv: self.incoming.oclIsUndefined()

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



Validation errors:
- Class Activity is unknown,
- Reference finalNode, initialNode, activity are unknown

Example 8/9

How to keep metamodels evolvable when models already exist

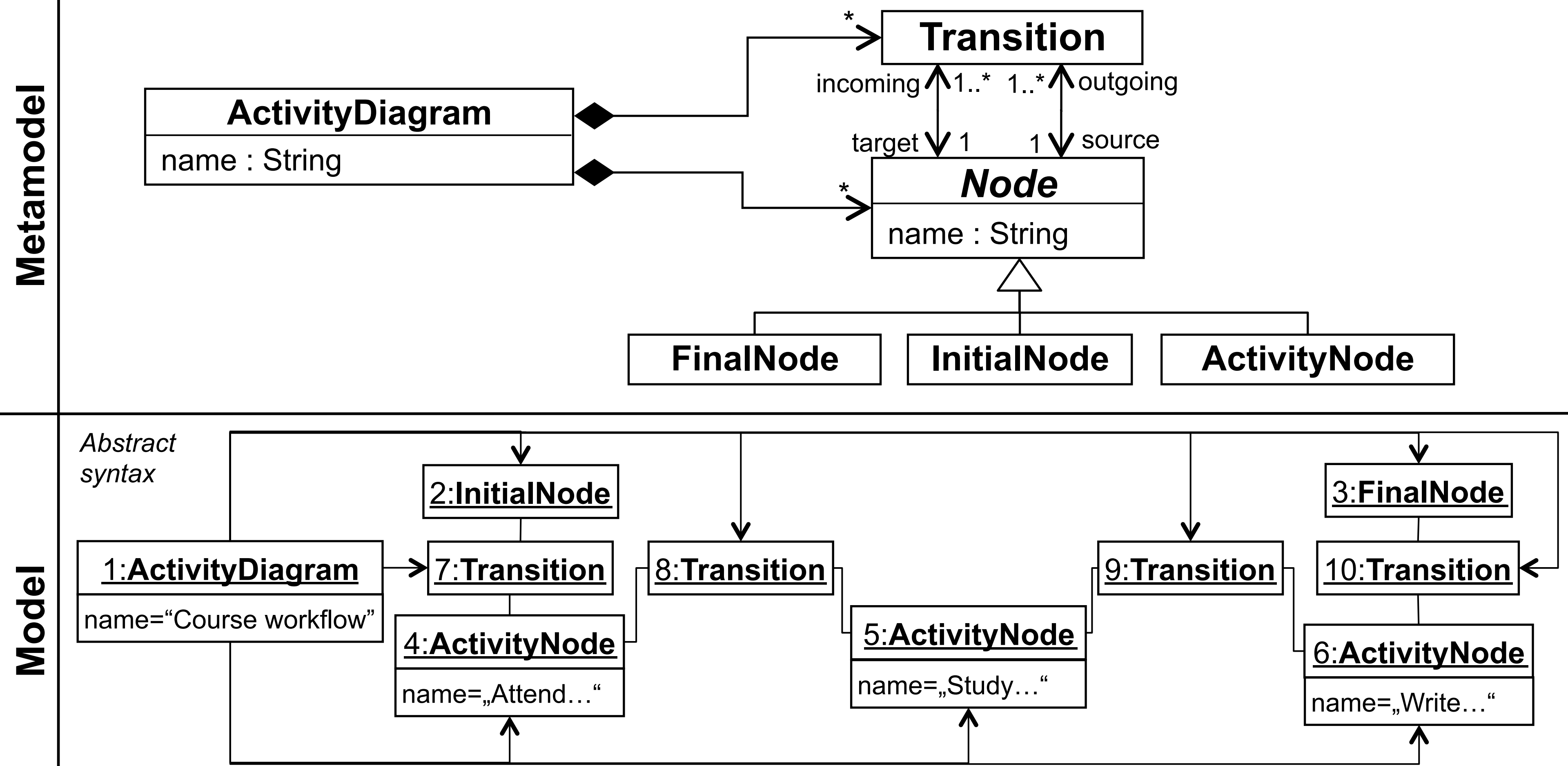
- **Model/metamodel co-evolution problem**
  - Metamodel is changed
  - Existing models eventually become invalid

- **Changes** may **break** conformance relationships
  - Deletions 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

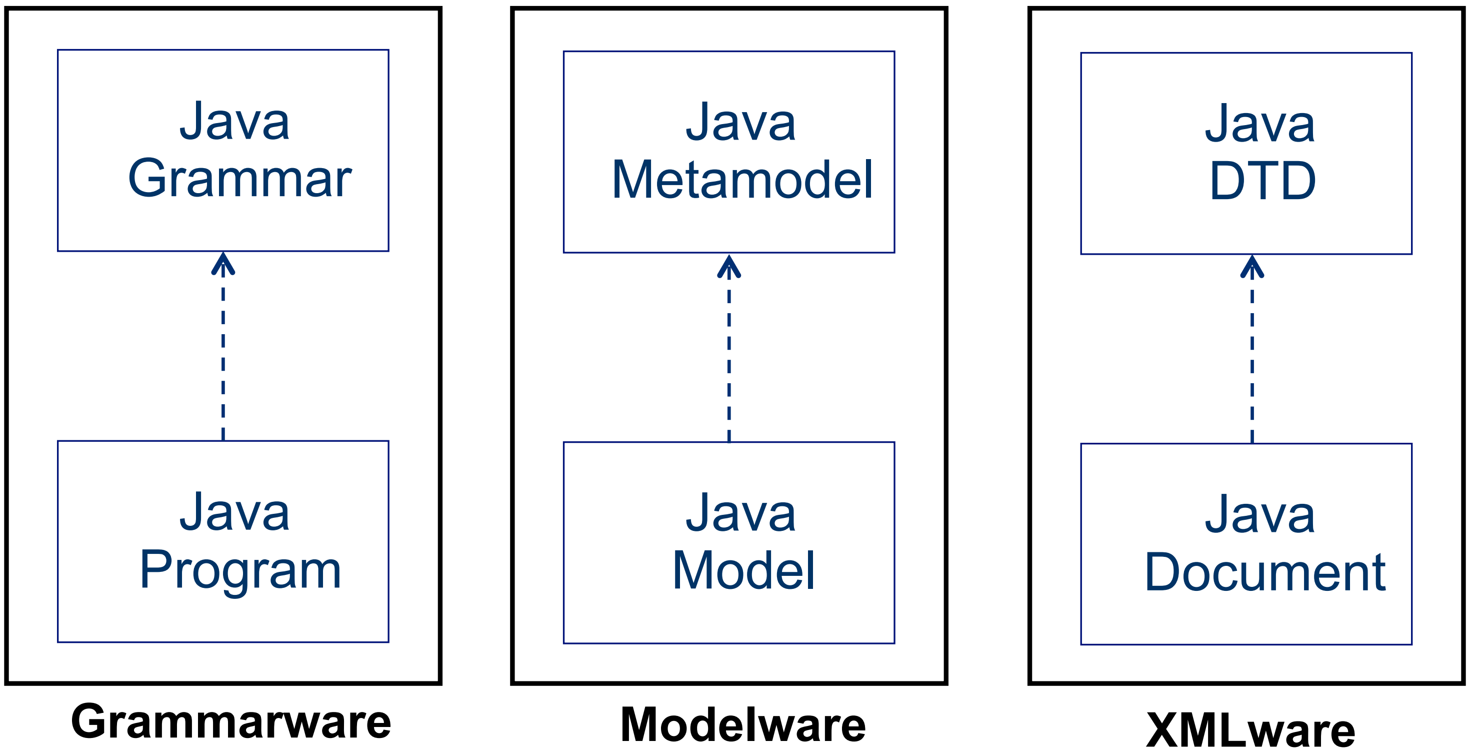
Adapted model for new metamodel version



More on this topic in Chapter 10!

Excursus: Metamodeling – everything new? 1/3

- 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..*

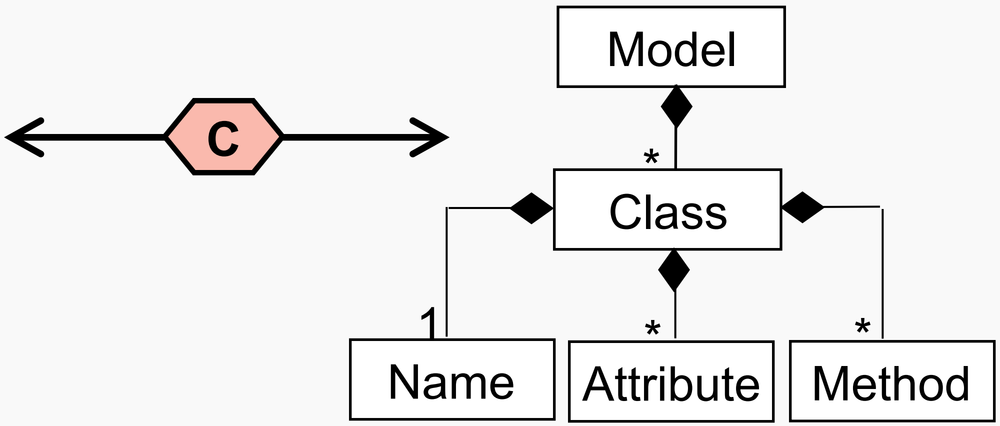
- **Example**

Grammar

Model ::= {Class}

Class ::= Name {Attribute}
{Method}

Metamodel



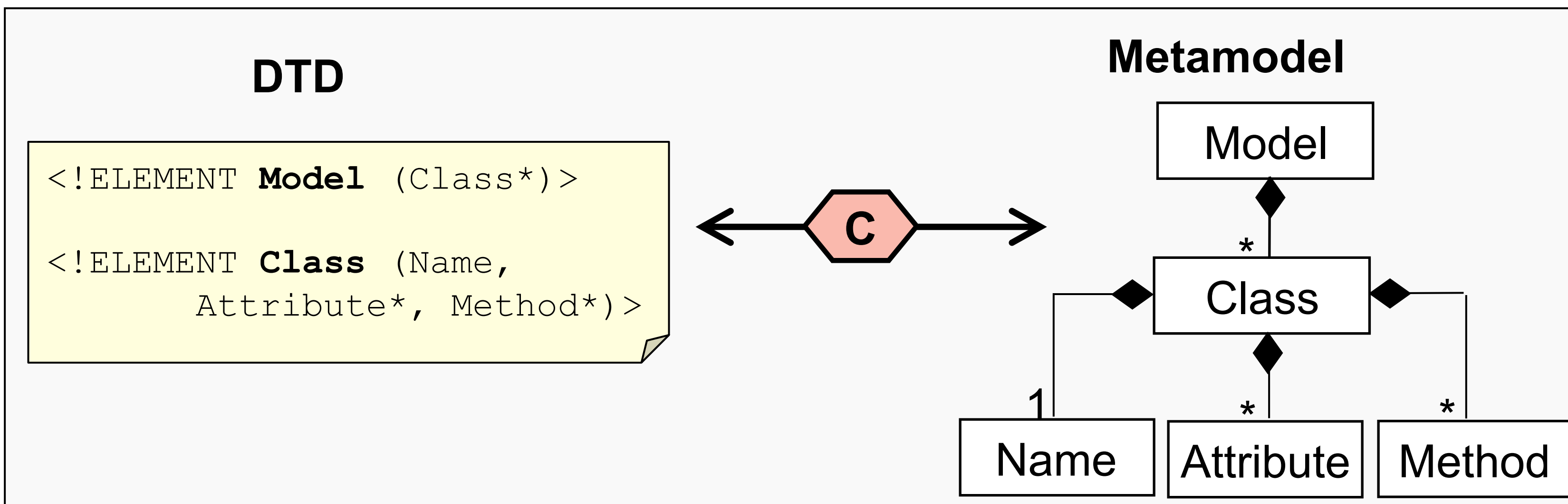
Excursus: Metamodeling – everything new? 3/3

Correspondence between DTD and MOF

- **Mapping table** (excerpt)

DTD	***MOF***
Item	Composition
Element	Class
Cardinality *	Multiplicity 0..*

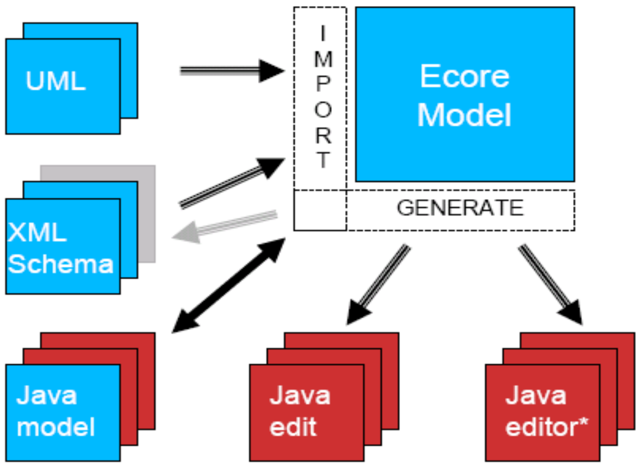
- **Example**



Ecore

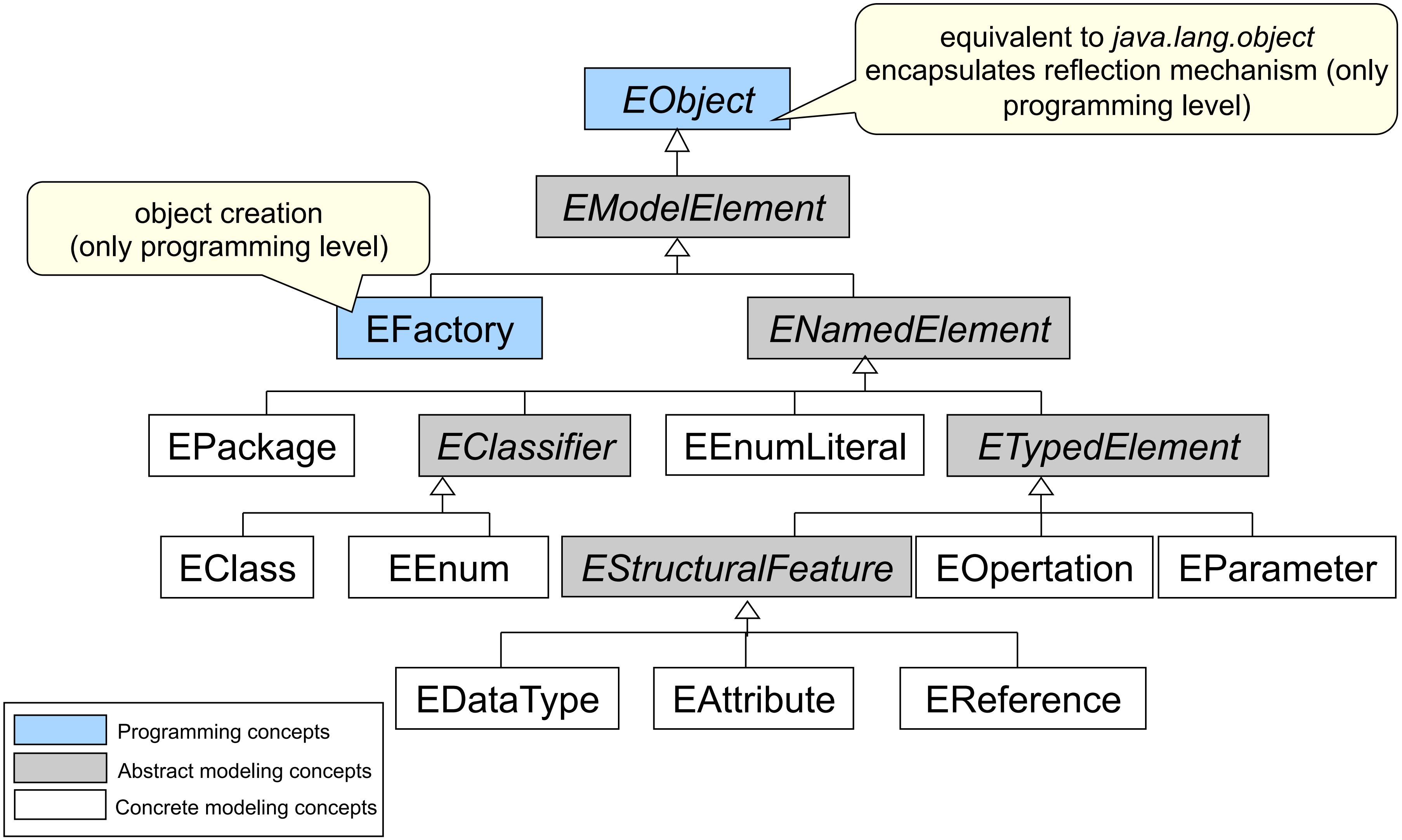
Introduction

- **Ecore** is the meta-metamodel of the Eclipse Modeling Frameworks (EMF)
  - 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

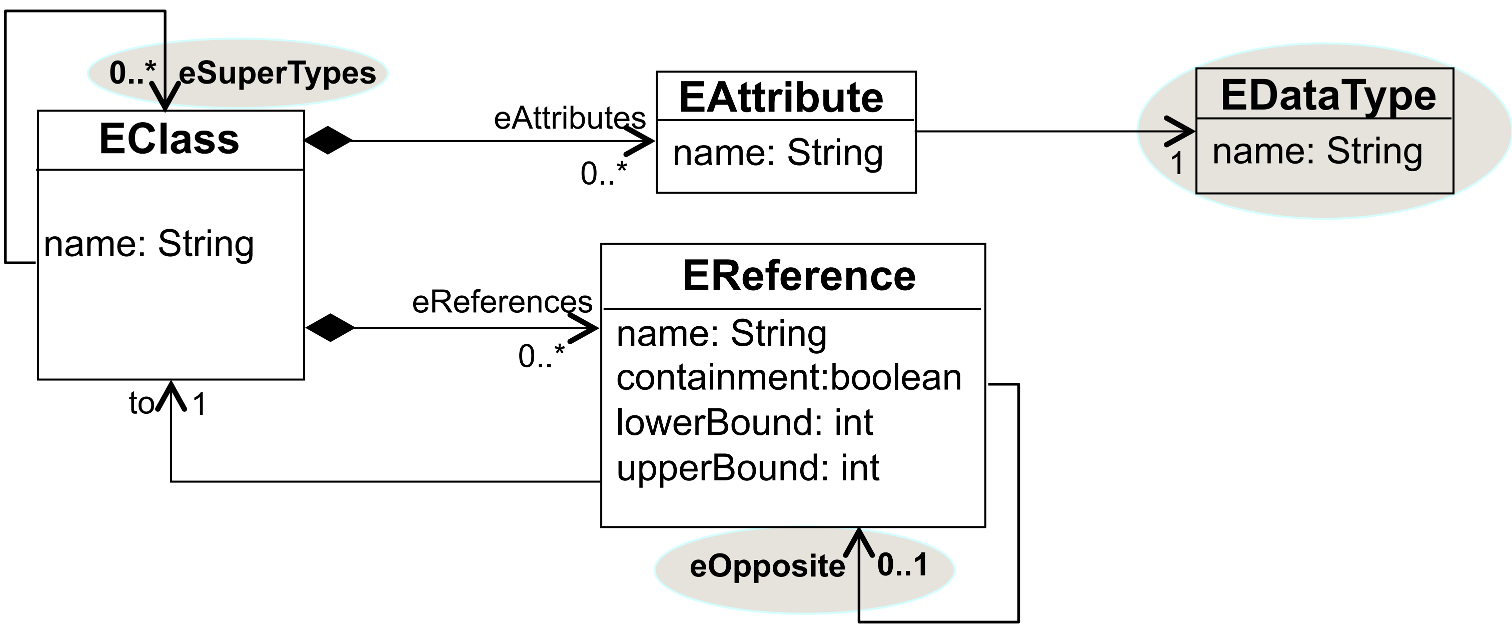
Taxonomy of the language concepts



Ecore

Core

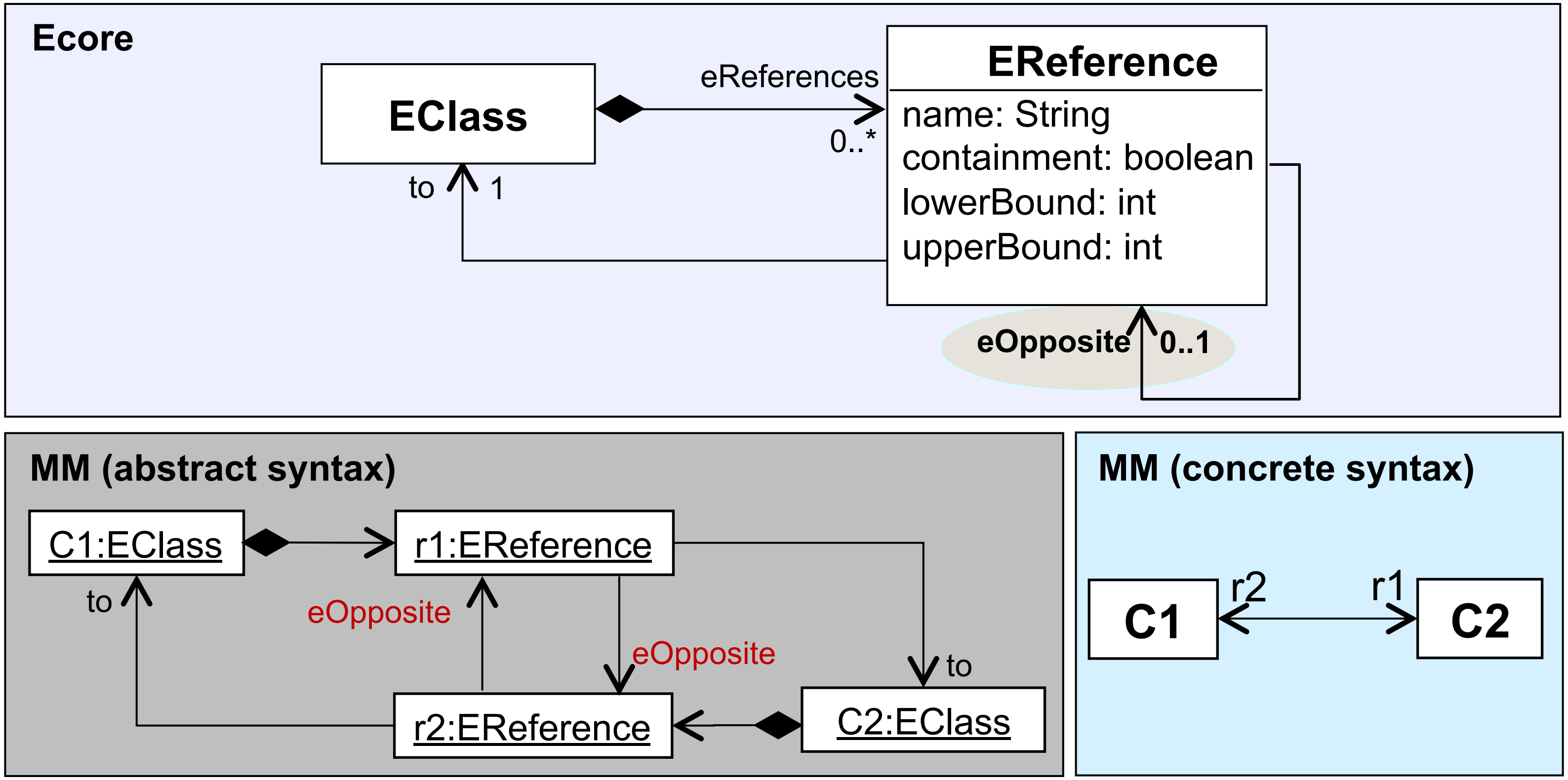
- Based on **object-orientation** (as eMOF)
  - Classes, references, attributes, inheritance, …
  - Binary *associations* are represented as **two references**
  - Data types are based on Java data types
  - Multiple inheritance is resolved by one „real“ inheritance and multiple implementation inheritance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



Ecore

Binary associations

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



Ecore

Data types

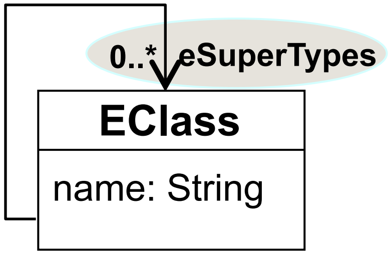
- List of Ecore data types (excerpt)
  - Java-based data types
  - **Extendable** through self-defined data types
    - Have to be implemented by Java classes

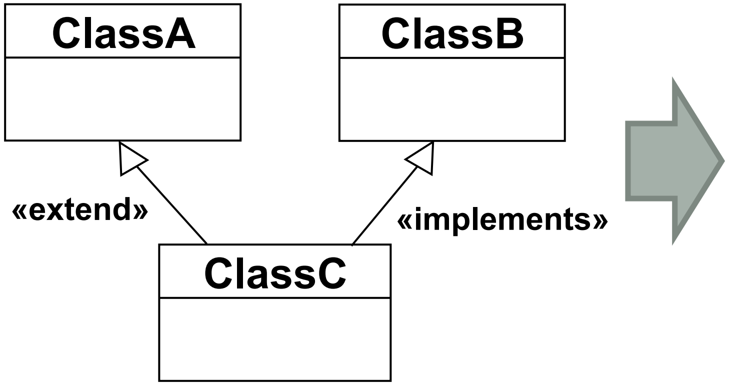
Ecore data type	**Primitive type or class (Java)**
EBoolean	boolean
EChar	char
EFloat	float
EString	java.lang.String
EBoolanObject	java.lang.Boolean
…	…

Ecore

Multiple inheritance

- 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»



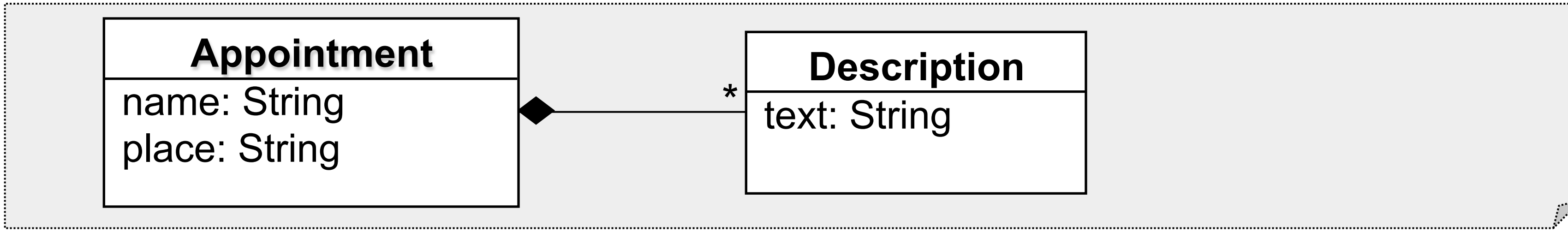



```
class ClassC
    extends ClassA
    implements ClassB
```

Ecore

Concrete syntax for Ecore models

- Class diagram – Model TS



- Annotated Java (Excerpt) – Program TS

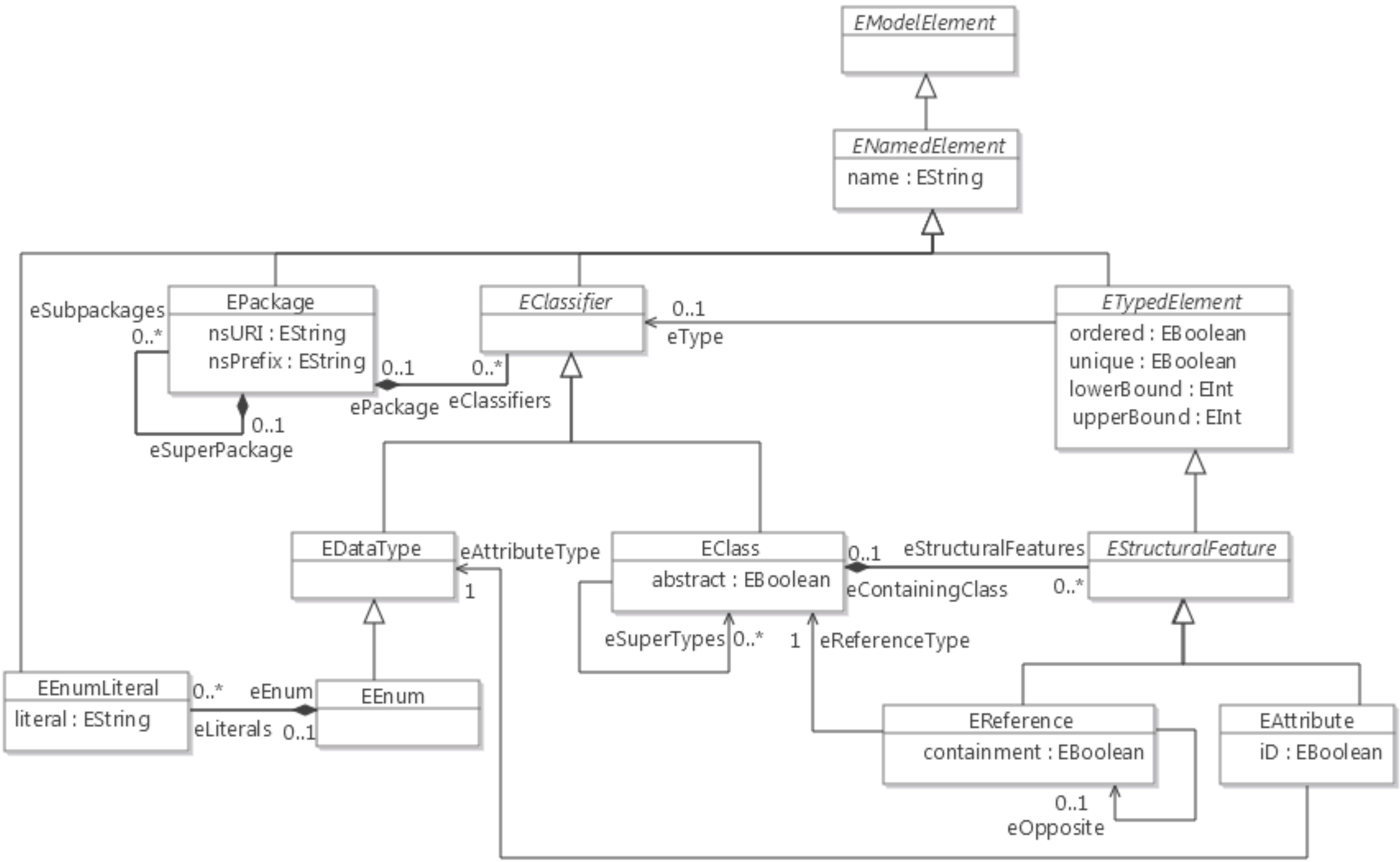
```
public interface Appointment{
    /* @model type="Description" containment="true" */
    List getDescription();
}
```

- XML (Excerpt) – Document TS

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

Summary

Ecore modeling elements at a glance



Eclipse Modeling Framework

What is EMF?

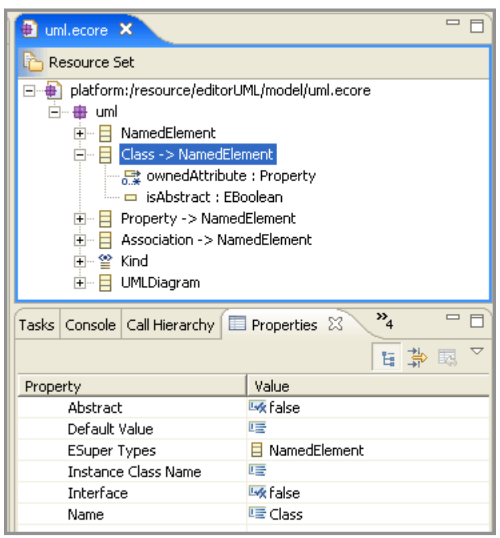
- **Pragmatic approach** to combine **modeling** and **programming**
  - **Straight-forward mapping rules** between Ecore and Java
- EMF **facilitates automatic generation** of **different implementations** out of Ecore models
  - Java code, XML documents, XML Schemata
- **Multitude** of **Eclipse projects** 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

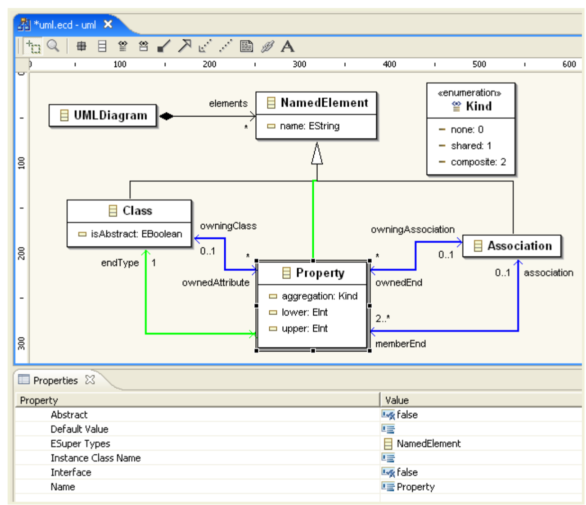
Metamodeling Editors

- **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**

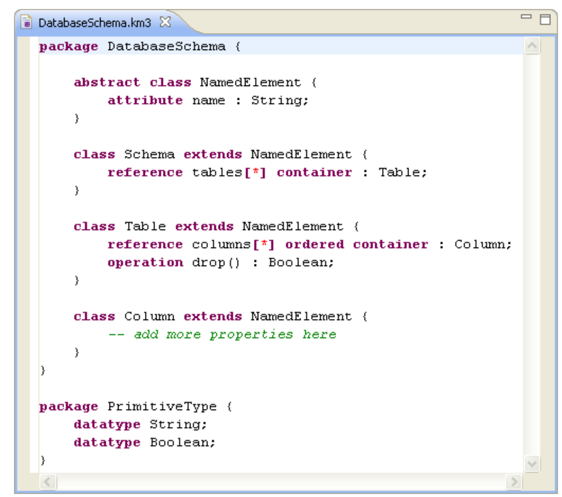
Tree-based editor



UML-based editor



Text-based editor



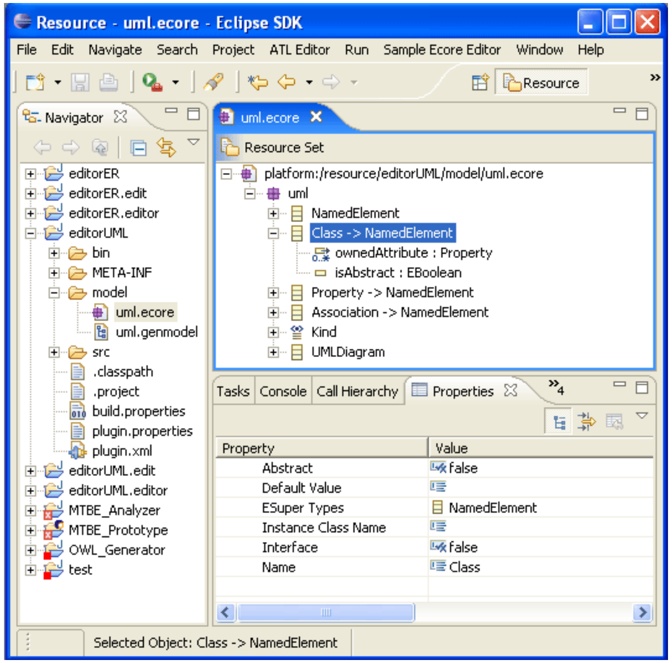
Eclipse Modeling Framework

Model editor generation process

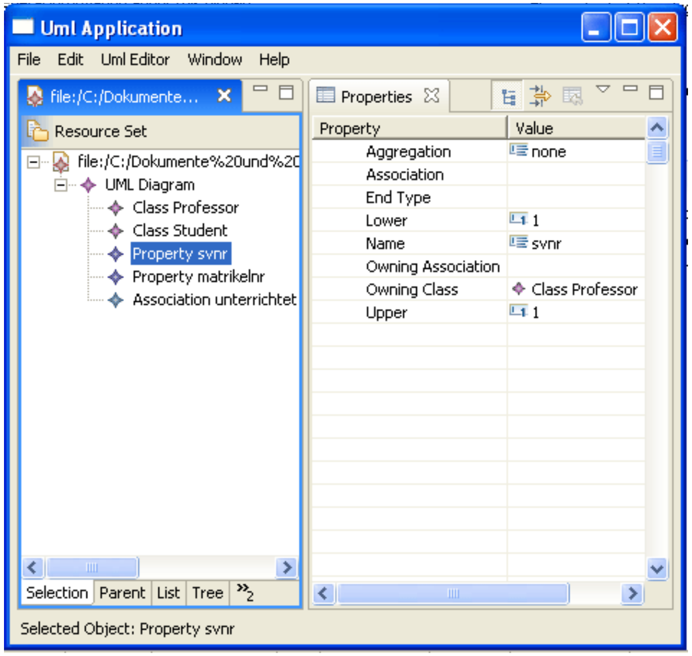
How can a **model editor** be created out of a **metamodel?**

Metamodel → ? → Model editor

Example: MiniUML metamodel -> MiniUML model editor

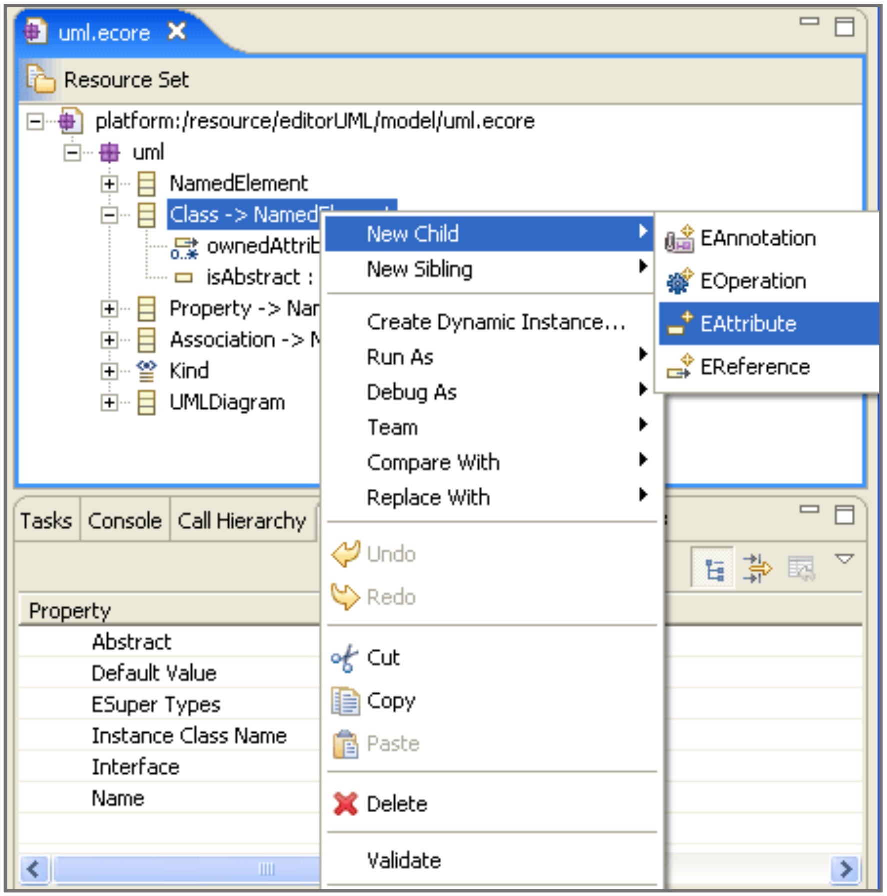


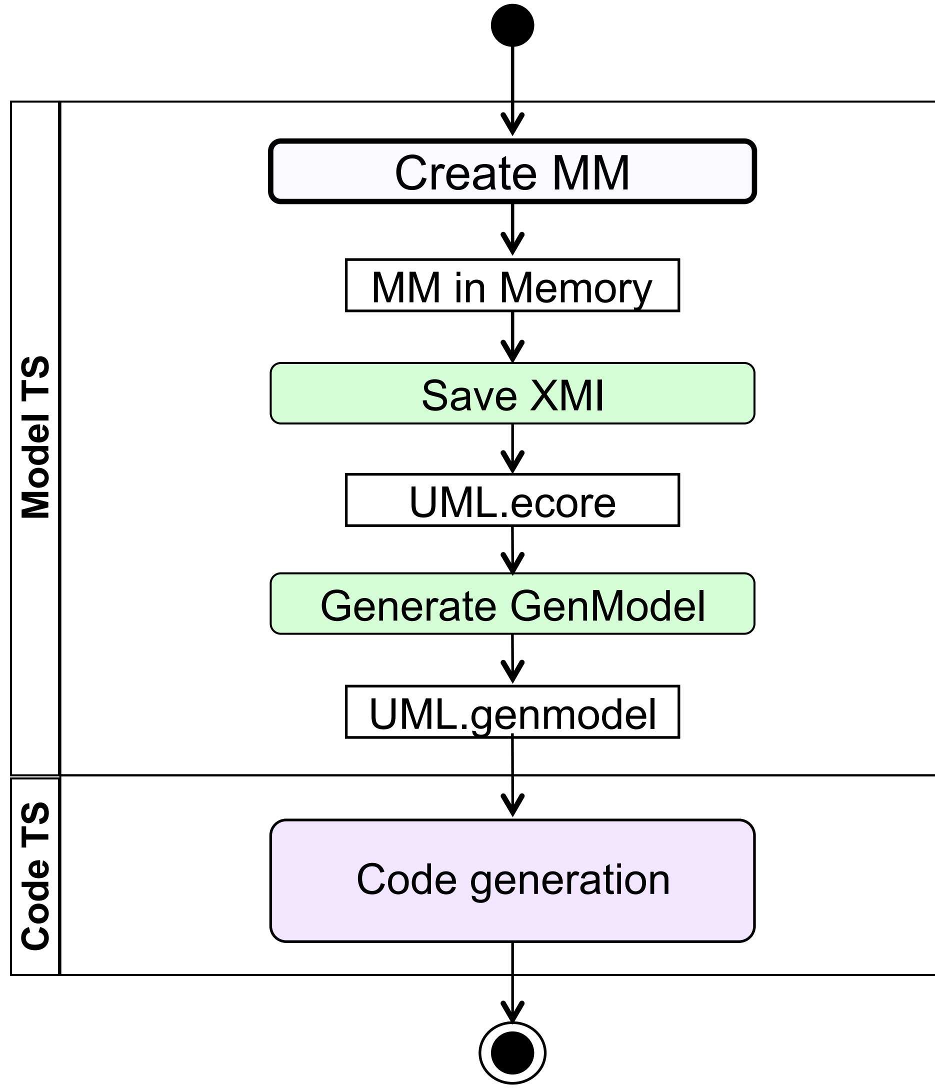
?



Model editor generation process

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



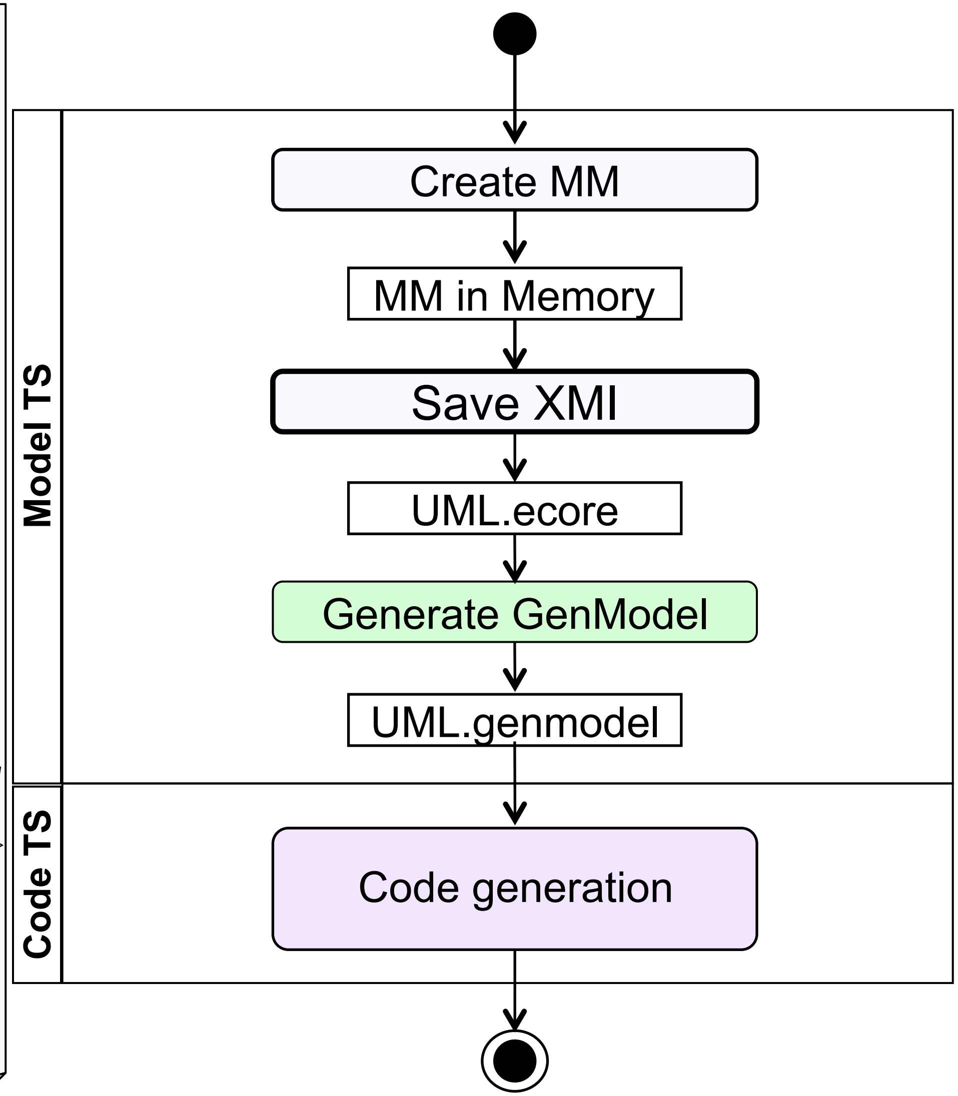


Model editor generation process

Step 2 – Save metamodel

UML.ecore

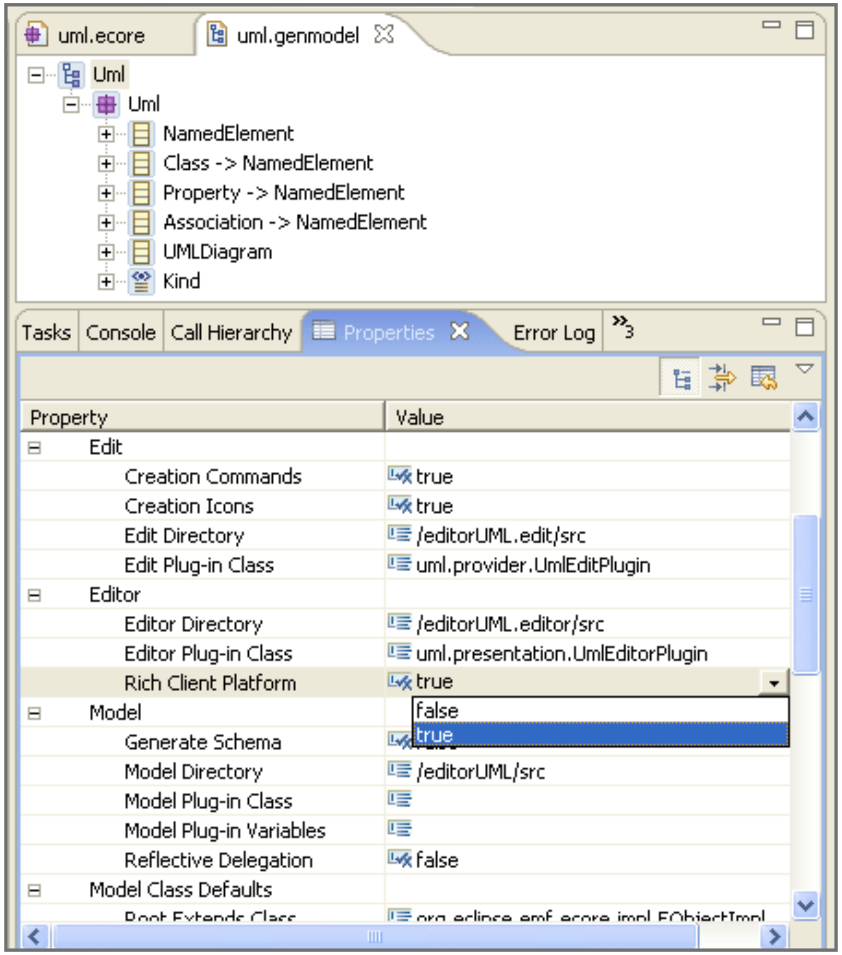
```xml
<?xml version="1.0" encoding="UTF-8"?>
<ecore:EPackage xmi:version="2.0"
   xmlns:xmi="http://www.omg.org/XMI"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-
           instance"
   xmlns:ecore="http://www.eclipse.org/emf/2002/
           Ecore"
   name="uml"
   nsURI="http://uml" nsPrefix="uml">
 <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>
```

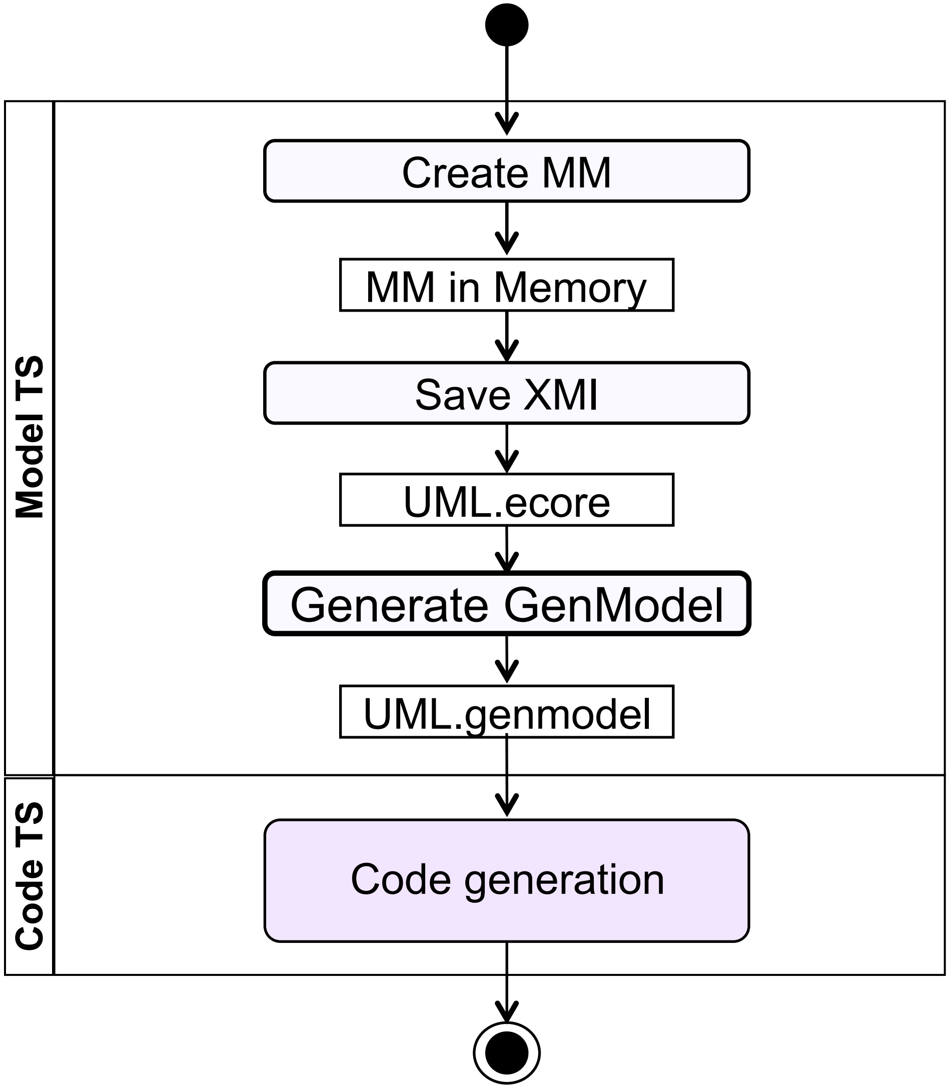


Model editor generation process

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:

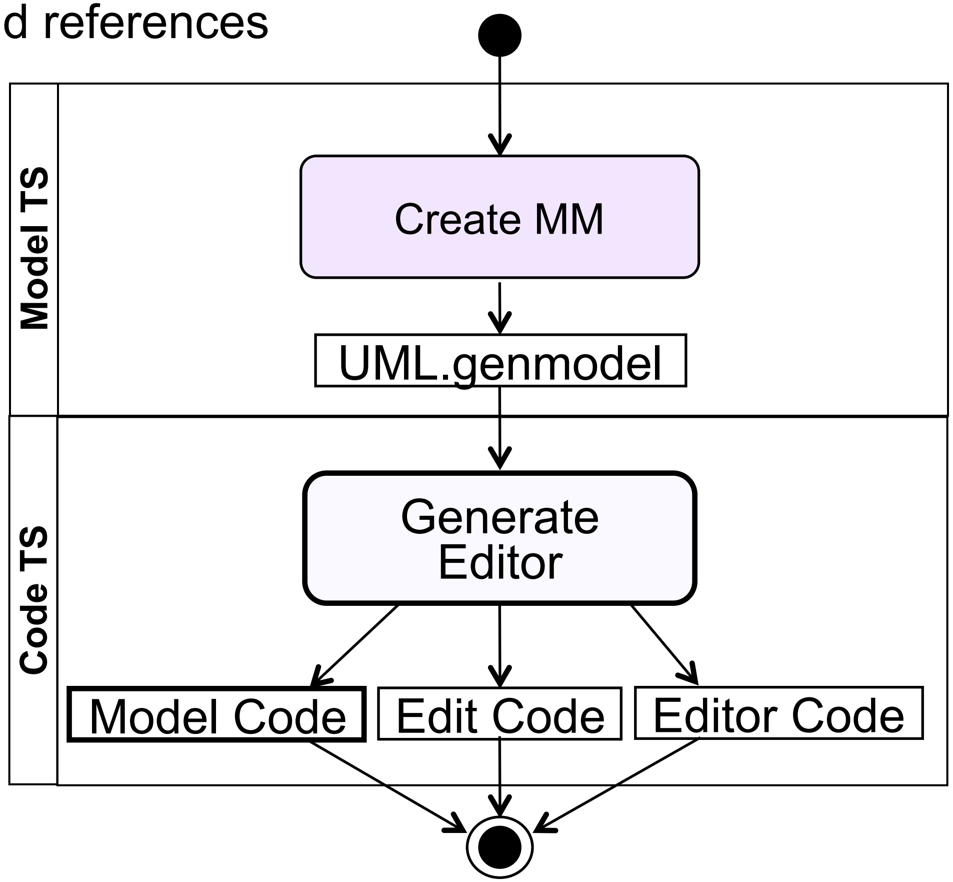
- **Interface:** Getter/setter for attributes and references

```
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

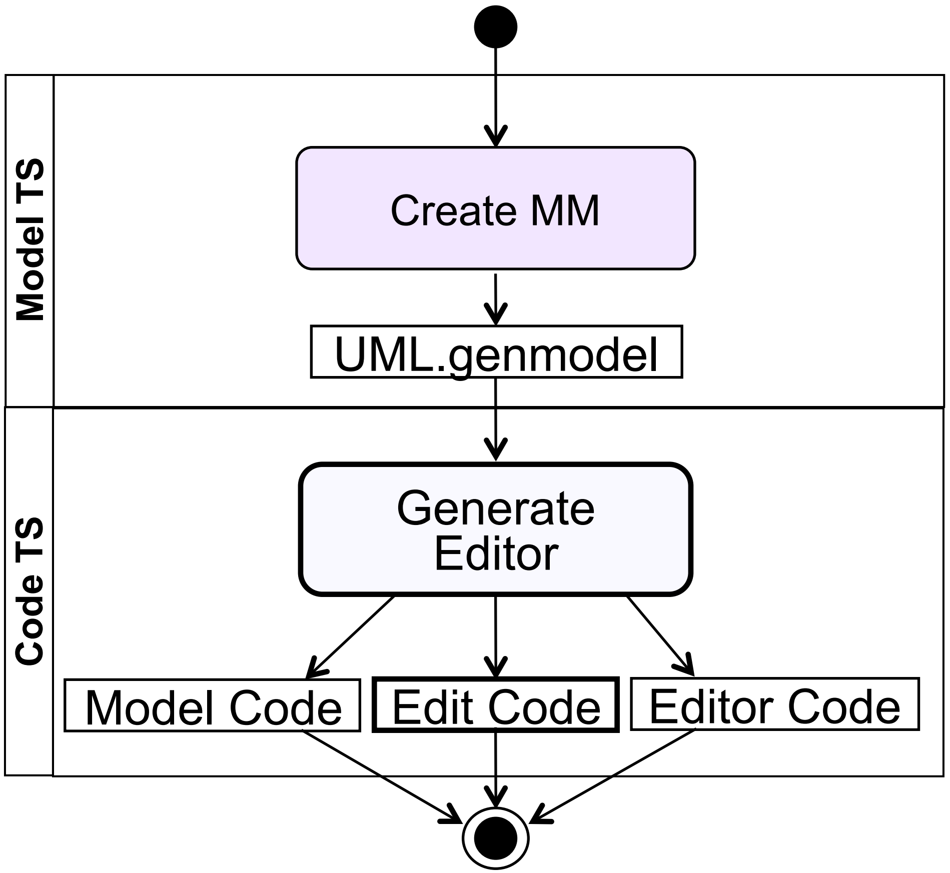


Model editor generation process

Step 5 – Generate edit code

- **UI independent** editing support for models

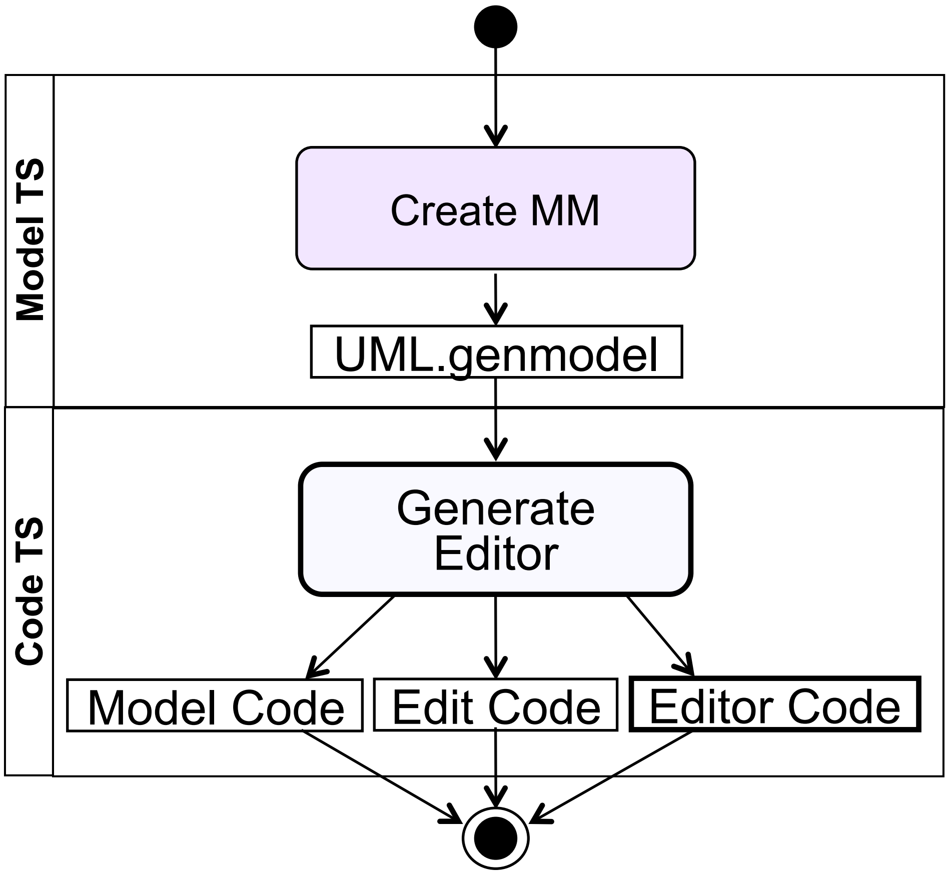
- Generated artifacts
  - *TreeContentProvider*
  - *LabelProvider*
  - *PropertySource*



Model editor generation process

Step 6 – Generate editor code

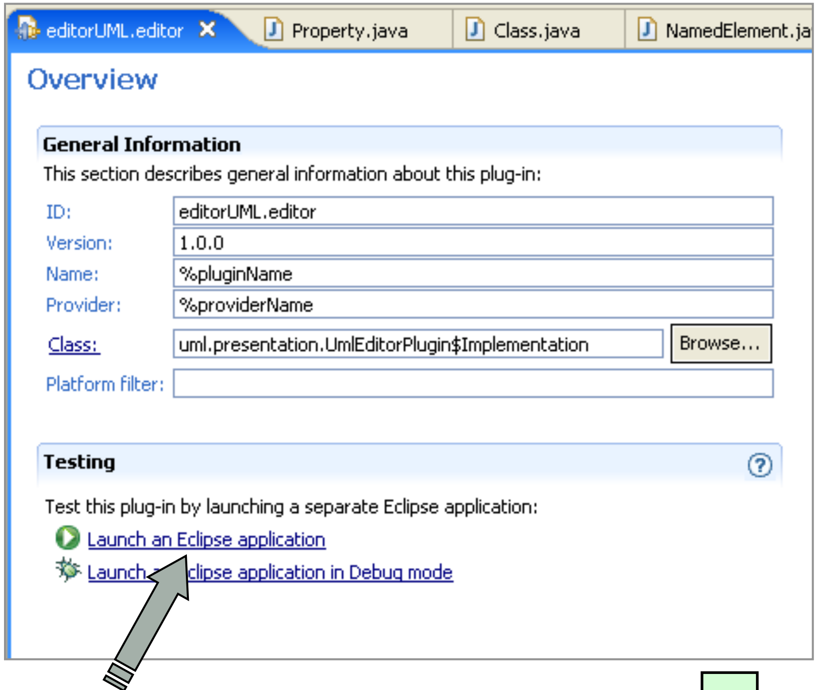
- Editor as **Eclipse Plugin** or **RCP Application**
- Generated artifacts
  - *Model creation wizard*
  - *Editor*
  - *Action bar contributor*
  - *Advisor (RCP)*
  - *plugin.xml*
  - *plugin.properties*



Model editor generation process

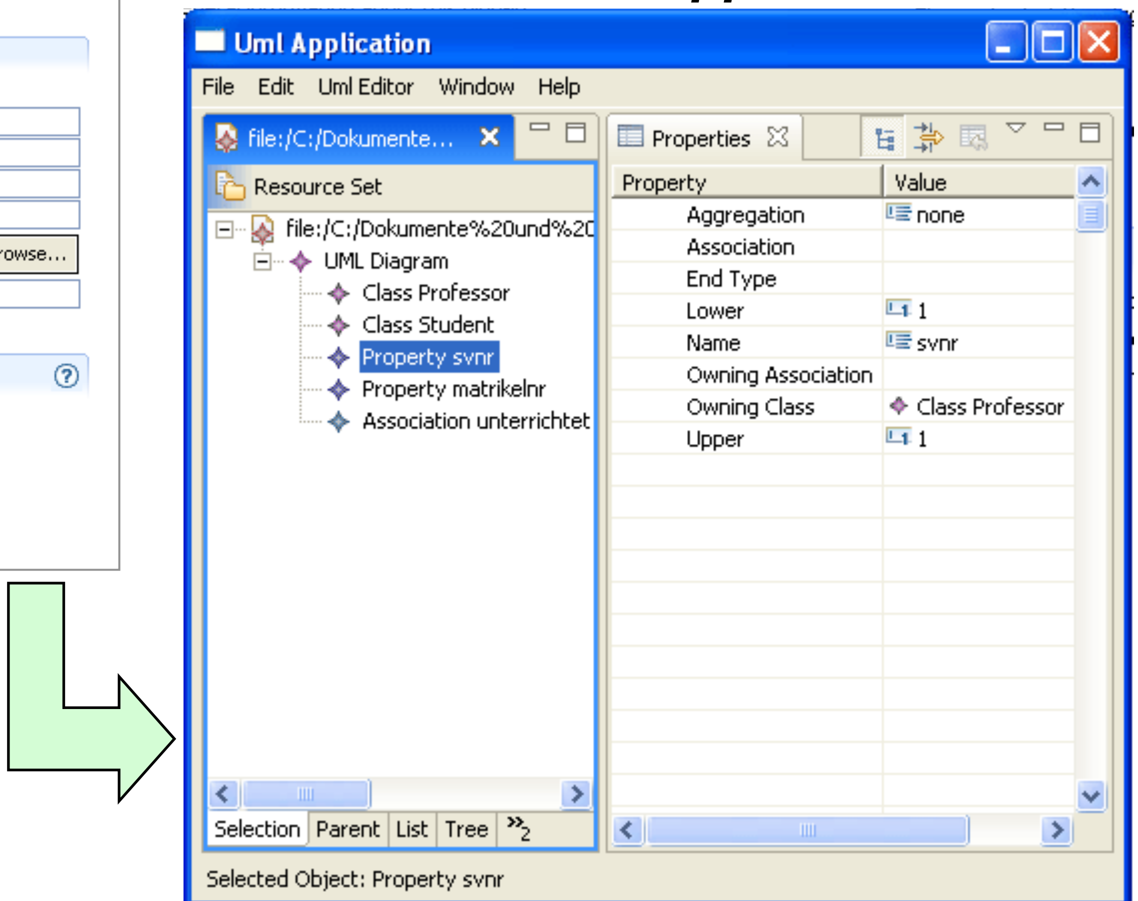
Start the modeling editor

Plugin.xml



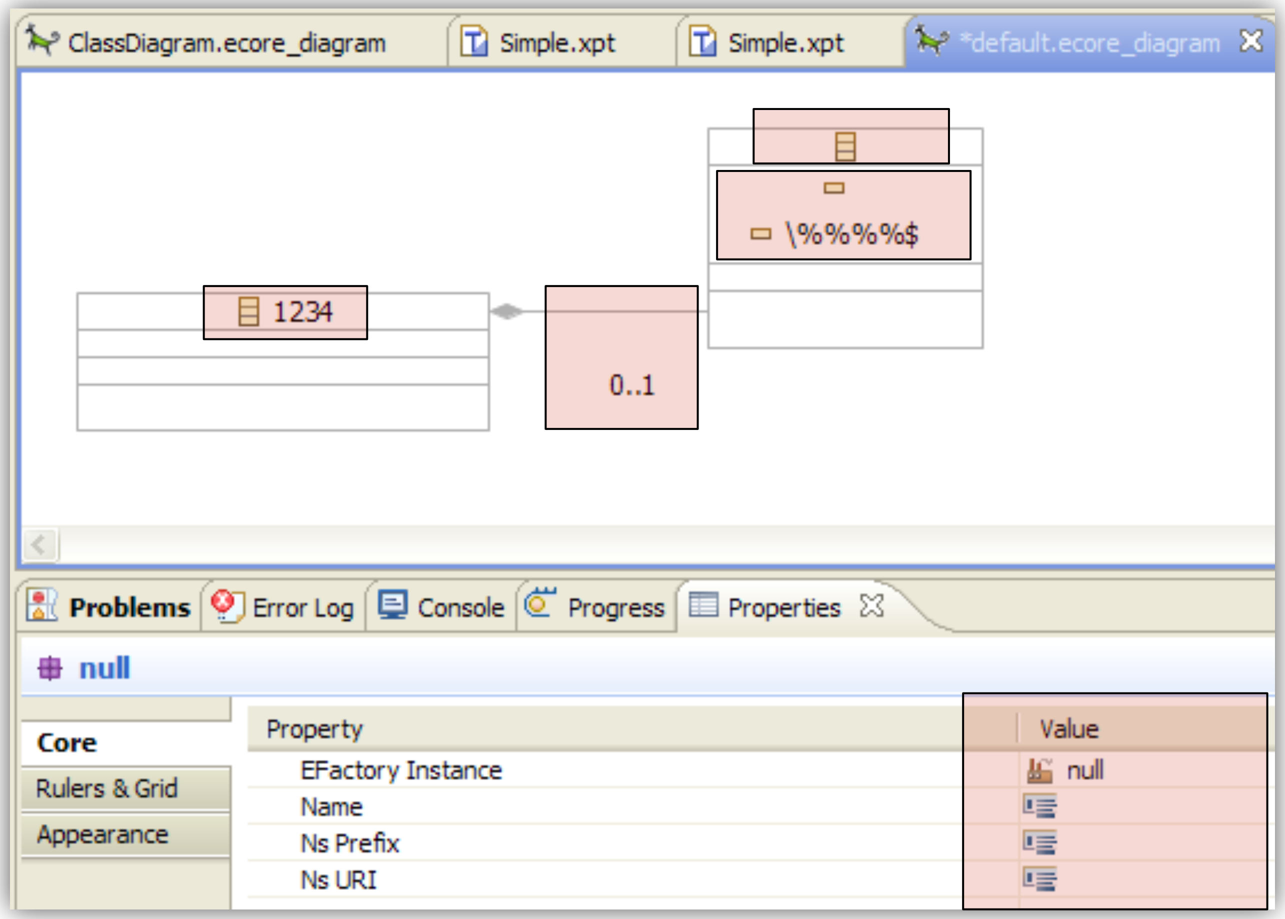
Click here to start!

RCP Application

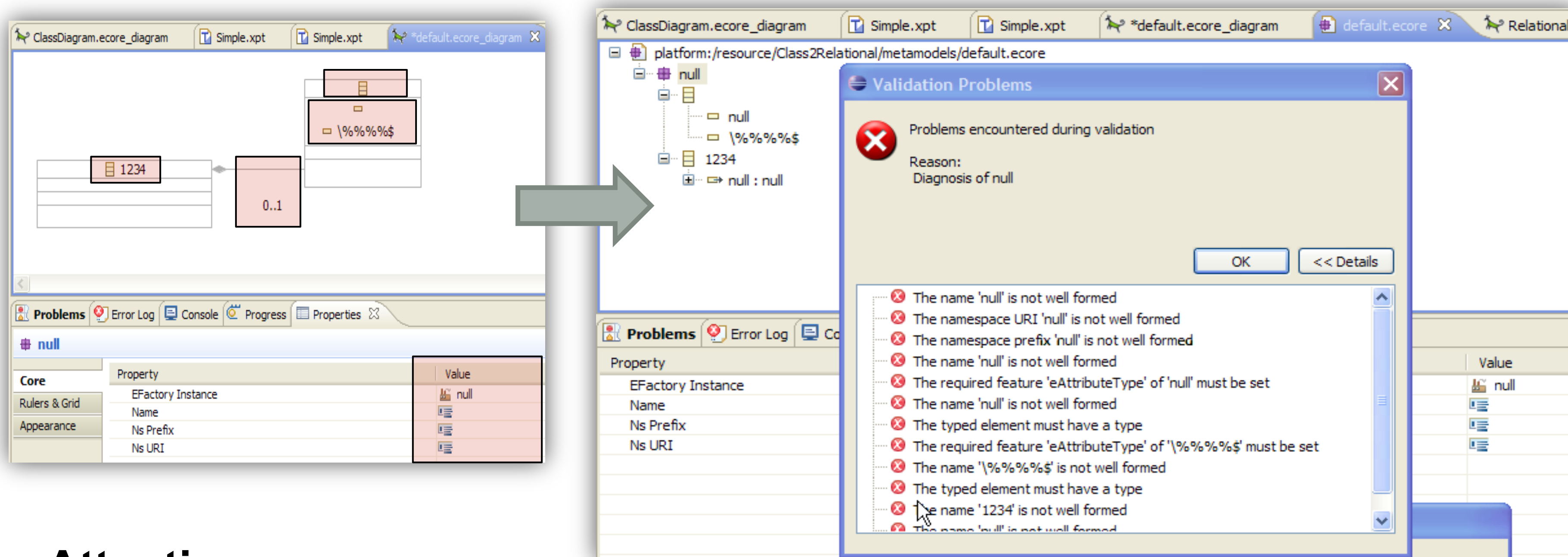


Metamodels are compiled to Java!

- Metamodeling mistake or error in EMF code generator?



Metamodels are compiled to Java!



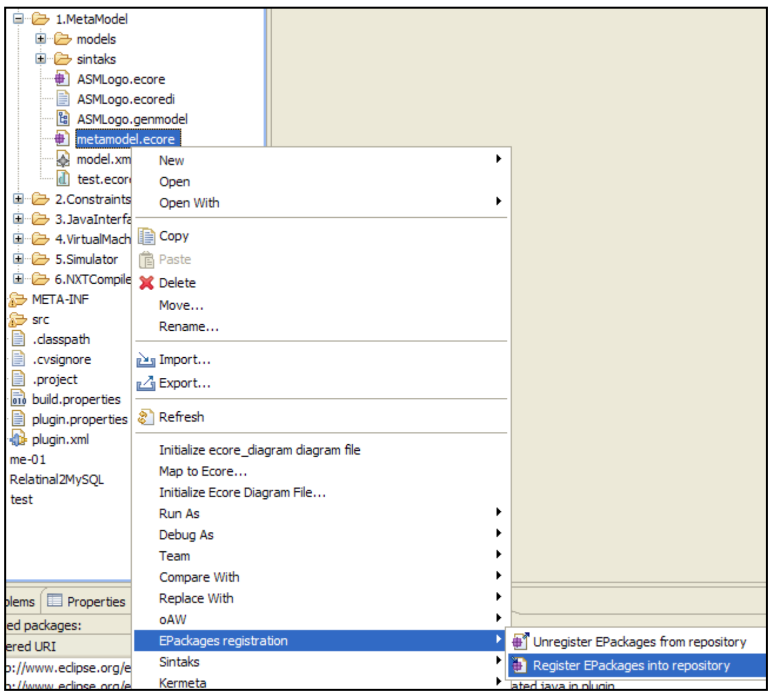
- **Attention**
  - Only use **valid Java identifier** as names
    - No blanks, no digits at the beginning, no special characters, …
  - **NamedElements** require a **name**
    - Classes, enumerations, attributes, **references**, packages
  - Attributes and references require a type
  - **Always**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

Shortcut for Metamodel Instantiation

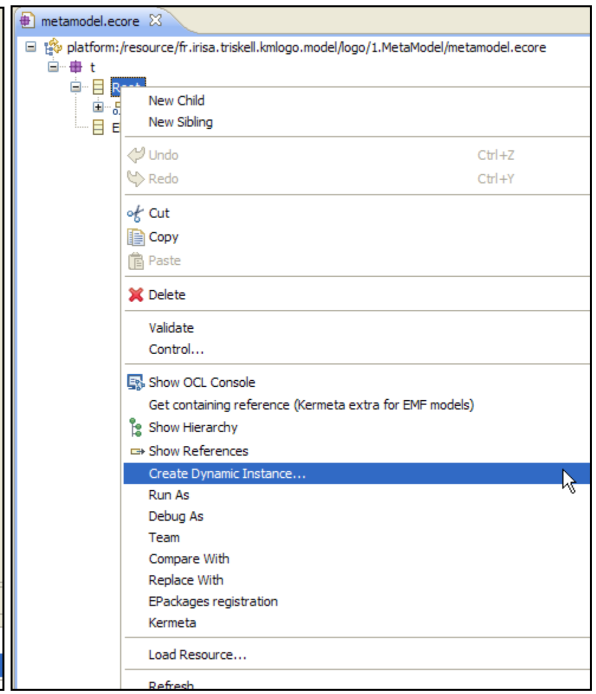
Metamodel Registration, Dynamic Model Creation, Reflective Editor

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

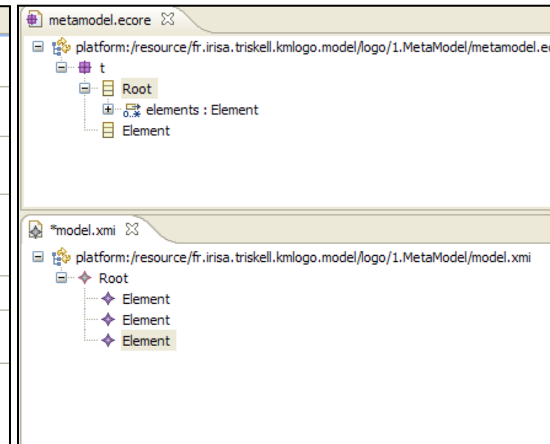
1) Register EPackages



2) Create Dynamic Instance



3) Use Reflective Editor

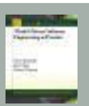


OCL support for EMF

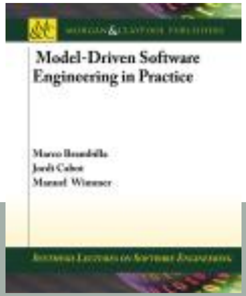
Several Plugins available

- **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

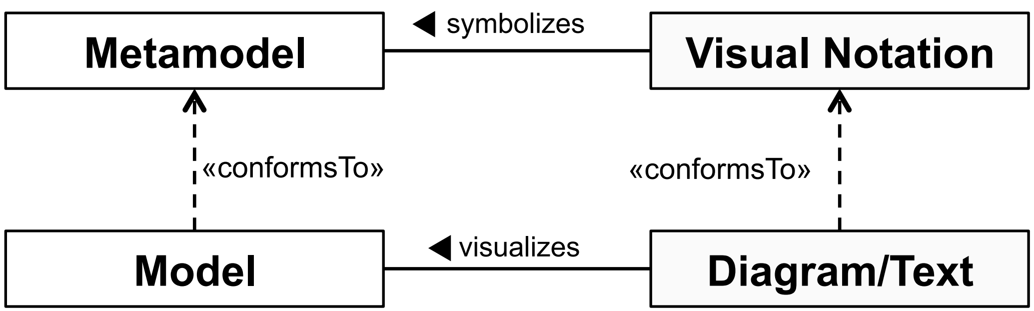


GRAPHICAL CONCRETE SYNTAX



Introduction

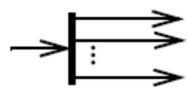
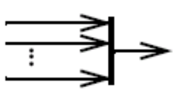
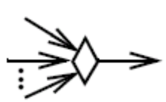
- The **visual notation** of a model language is referred as **concrete syntax**
- **Formal definition** of concrete syntax allows for **automated generation** of editors
- Several approaches and frameworks available for defining concrete syntax for model languages



Introduction

- Several languages have **no formalized definition** of their **concrete syntax**

- Example – Excerpt from the UML-Standard

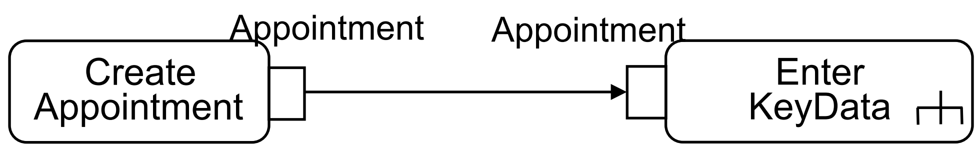
NODE TYPE	NOTATION	REFERENCE
ForkNode		See ForkNode (from IntermediateActivities) on page -404.
InitialNode		See InitialNode (from BasicActivities) on page -406.
JoinNode		See "JoinNode (from CompleteActivities, IntermediateActivities)" on page 411.
MergeNode		See "MergeNode (from IntermediateActivities)" on page 416.

Introduction

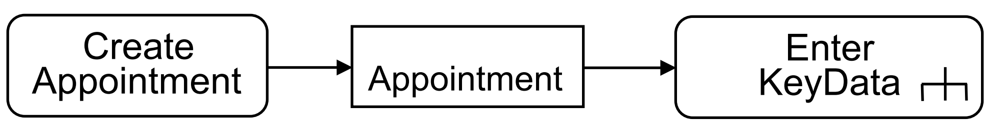
- Concrete syntax **improves** the **readability** of models
  - Abstract syntax not intended for humans!
- **One** abstract syntax may have **multiple** concrete ones
  - Including textual and/or graphical
  - Mixing textual and graphical notations still a challenge!

- **Example** – Notation alternatives for the creation of an appointment

Notation alternative 1:



Notation alternative 2:



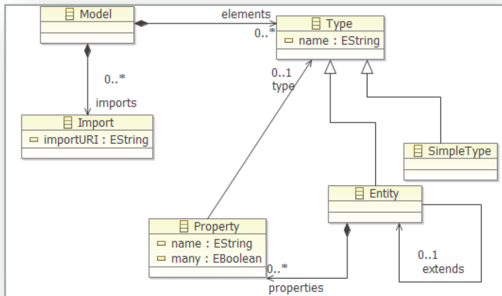
Notation alternative 3:

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

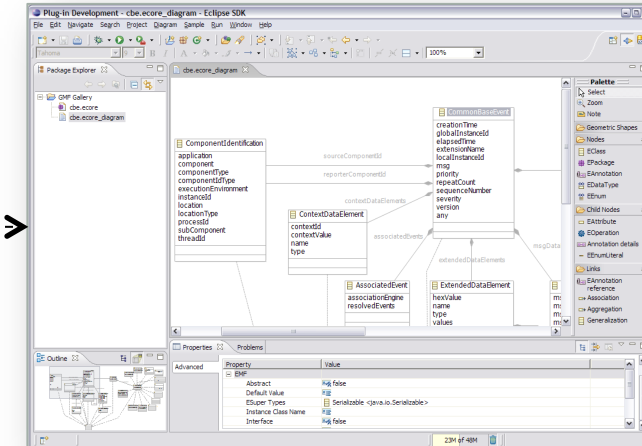
Introduction

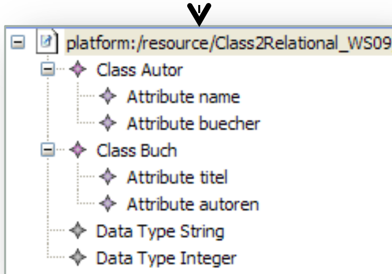
Concrete Syntaxes in Eclipse

Ecore-based Metamodels



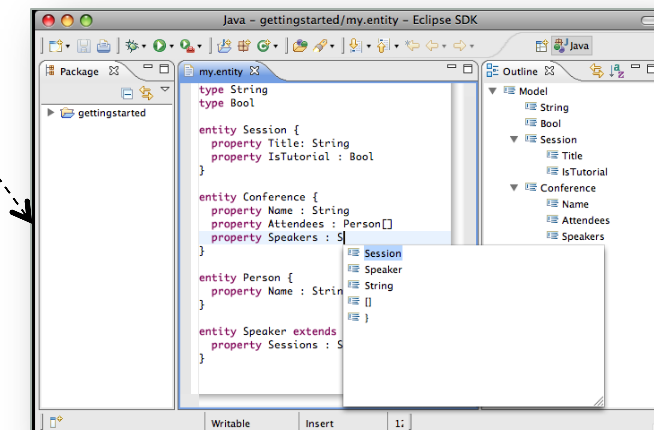
Graphical Concrete Syntax





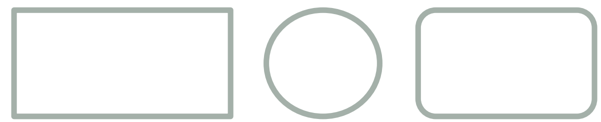
Generic tree-based EMF Editor

Textual Concrete Syntax

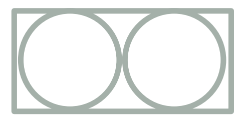


Anatomy of Graphical Concrete Syntaxes

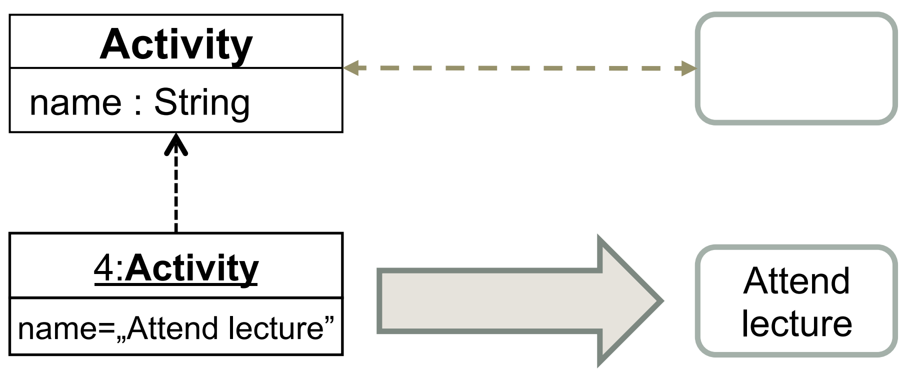
- A Graphical Concrete Syntax (GCS) consists of
  - **graphical symbols,**
    - e.g., rectangles, circles, …



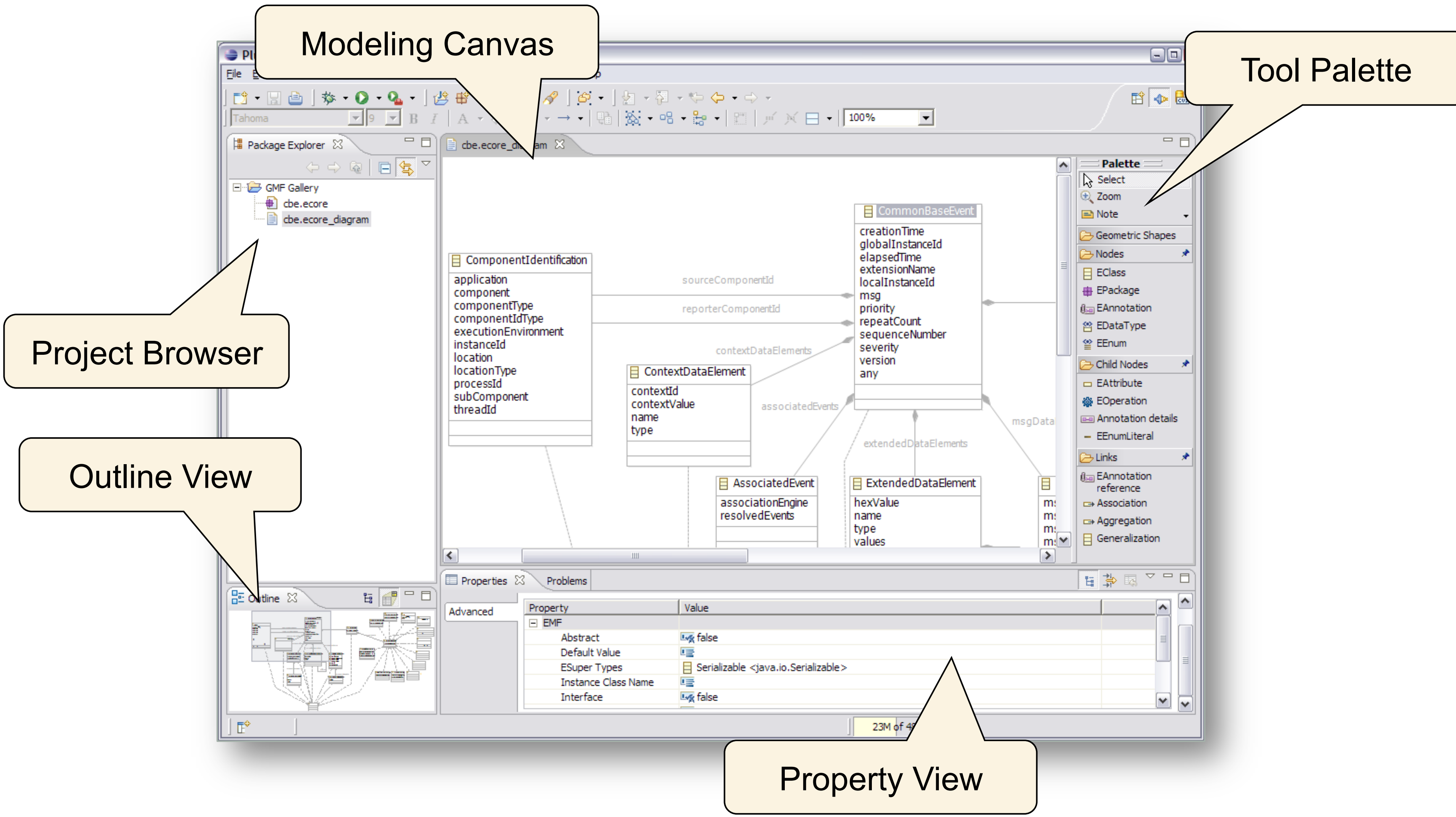
  - **compositional rules**,
    - e.g., nesting of elements, …



  - and **mapping** between **graphical symbols** and **abstract syntax elements**.
    - e.g., instances of a meta-class are visualized by rounded rectangles in the GCS

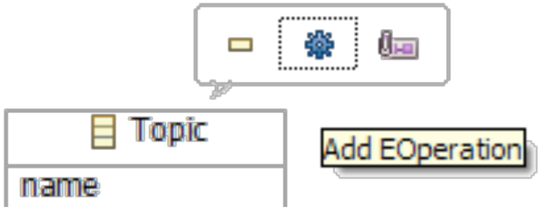


Anatomy of Graphical Modeling Editors

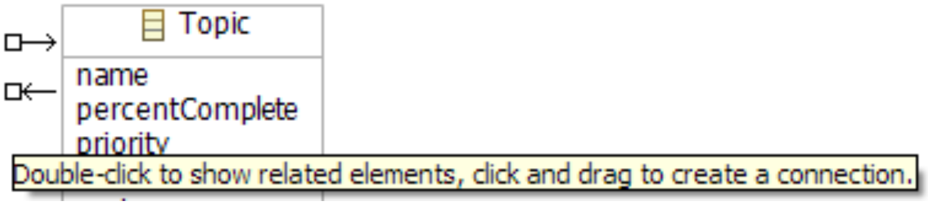


Features of Graphical Modeling Editors

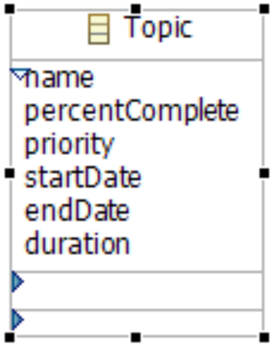
Action Bars:



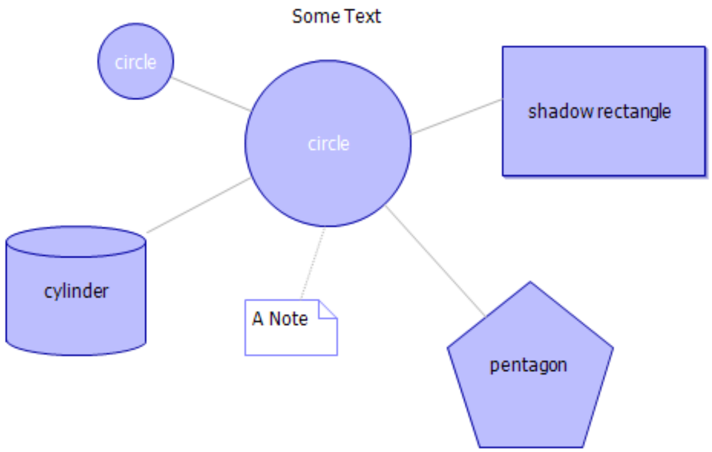
Connection Handles:



Collapsed Compartments:

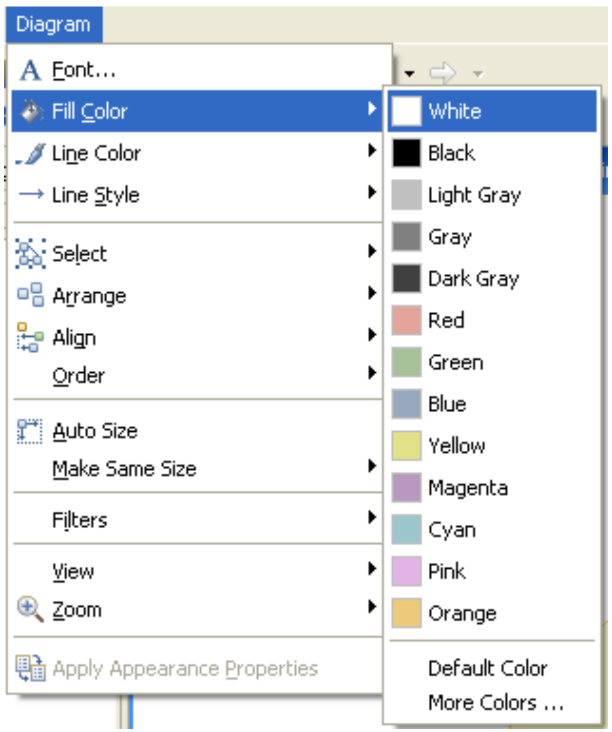


Geometrical Shapes:



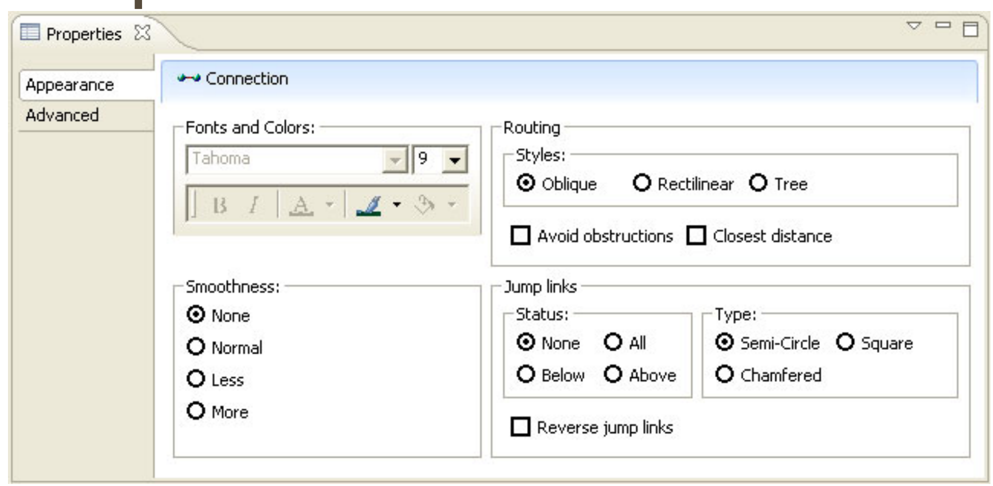
Features of Graphical Modeling Editors

Actions:

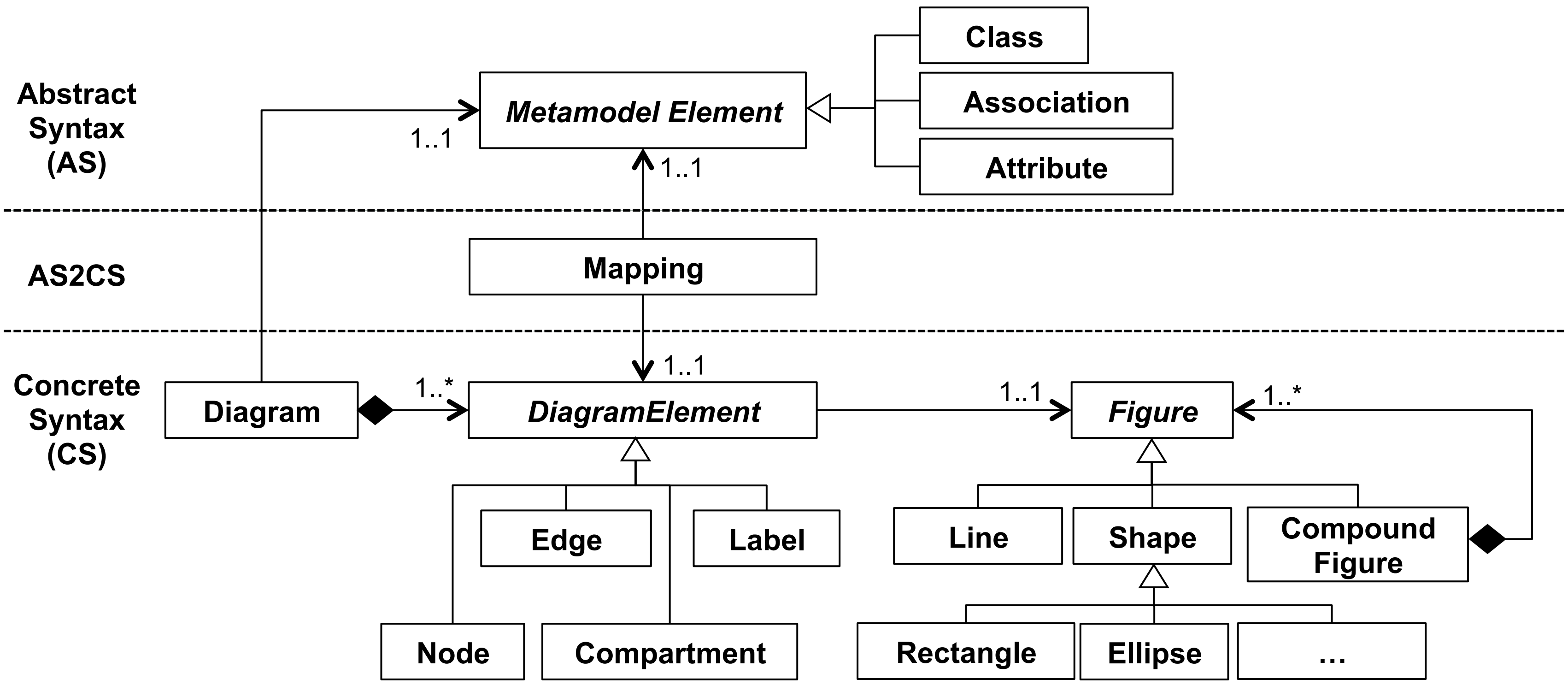


Toolbar:

Properties View:

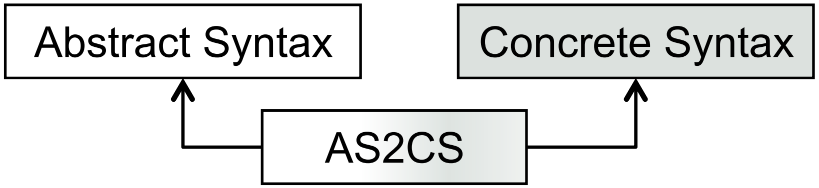


Generic Metamodel for GCS

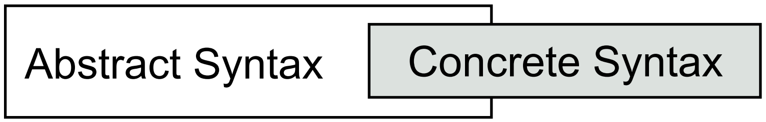


GCS Approaches

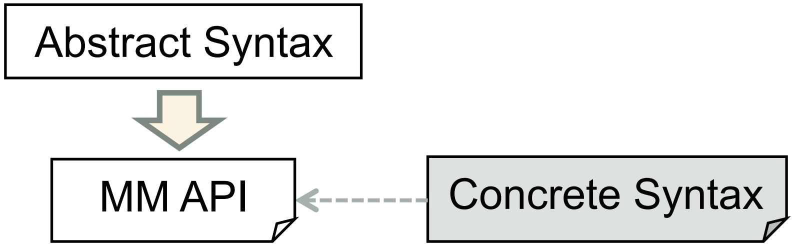
- **Mapping-based**
  - Explicit mapping model between abstract syntax, i.e., the metamodel, and concrete syntax



- **Annotation-based**
  - The metamodel is annotated with concrete syntax information



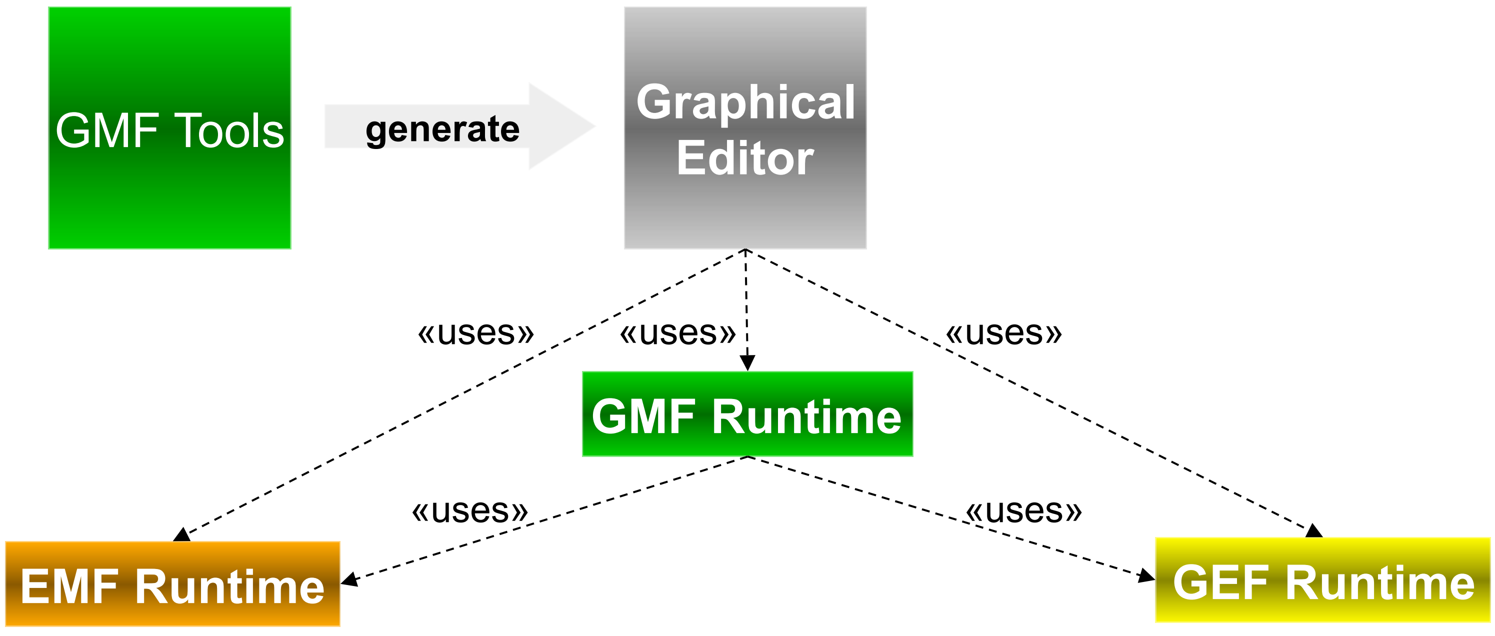
- **API-based**
  - Concrete syntax is described by a programming language using a dedicated API for graphical modeling editors



Mapping-based Approach: GMF

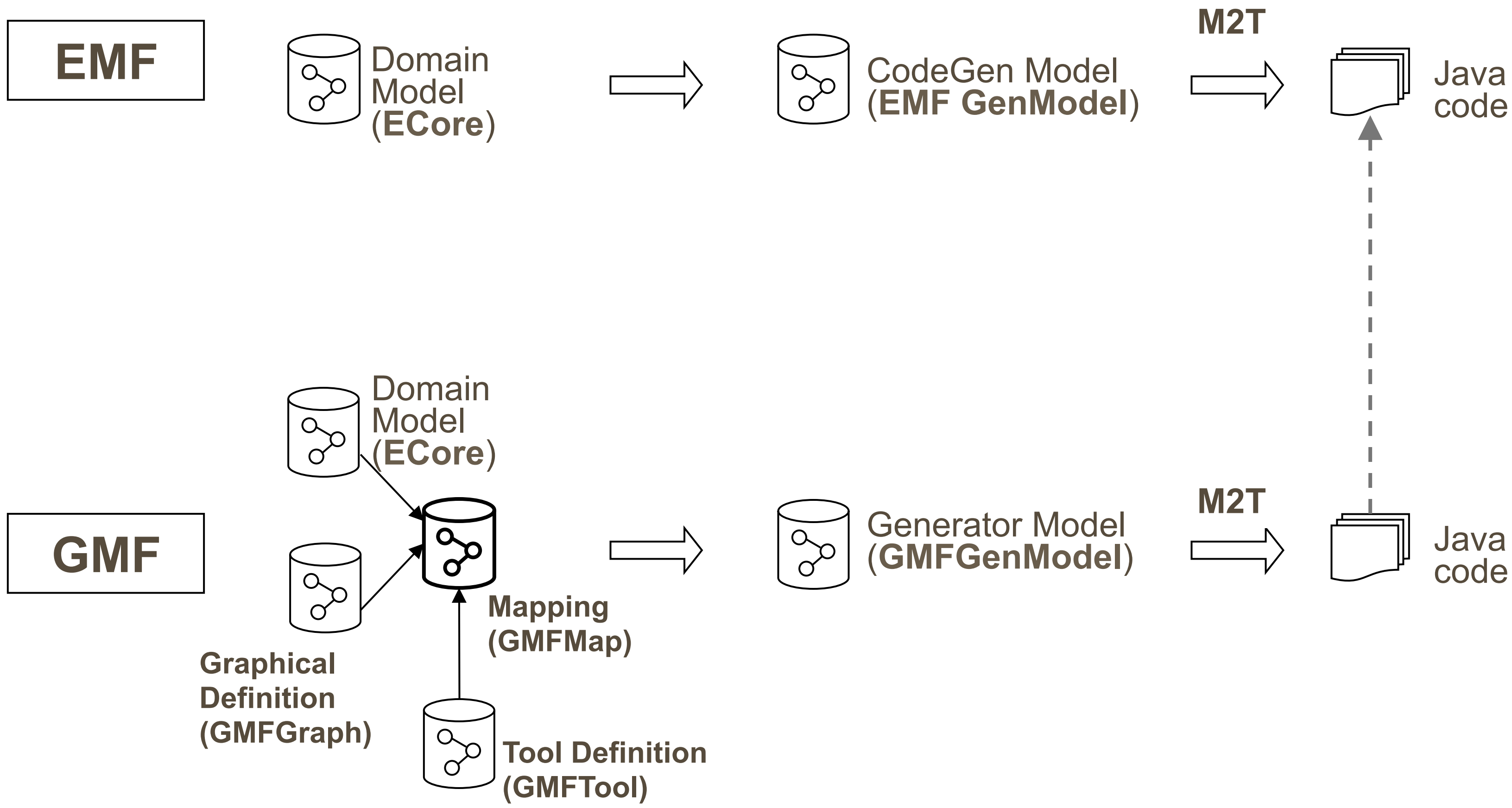
Basic Architecture of GMF

- *"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



Mapping-based Approach: GMF

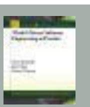
Tooling Component



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!

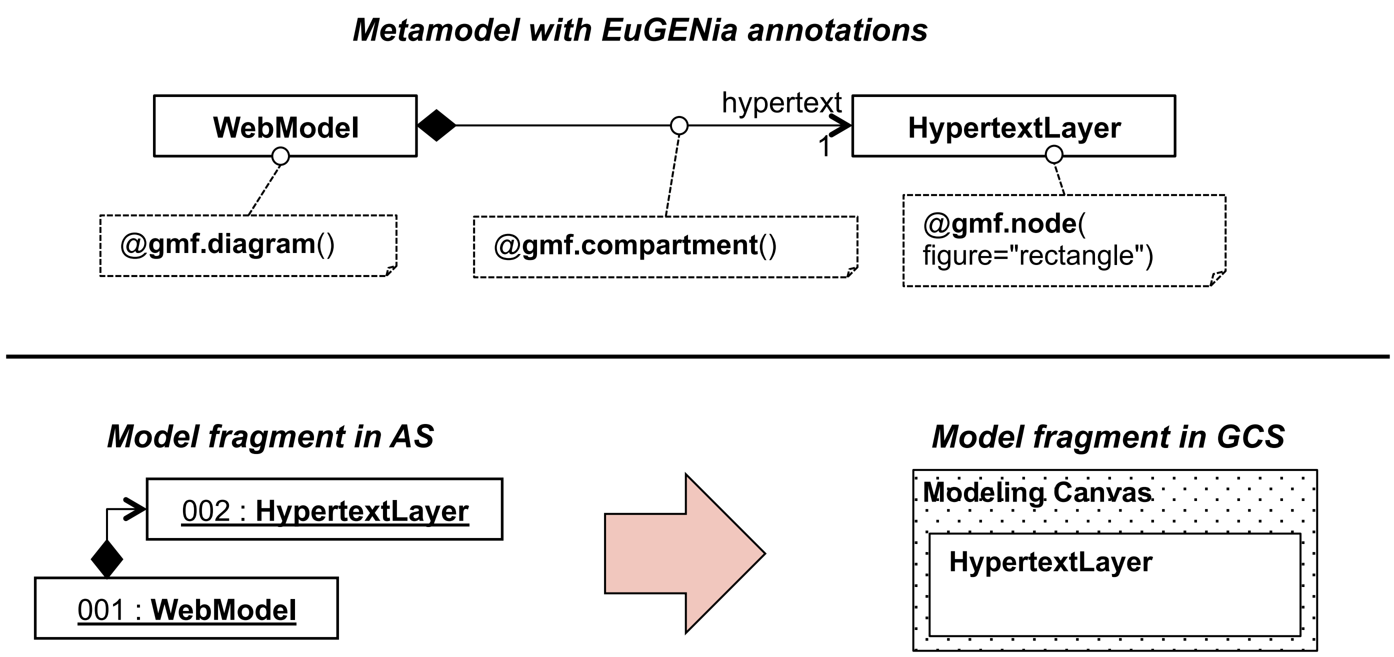


Eugenia Annotations (Excerpt)

- **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

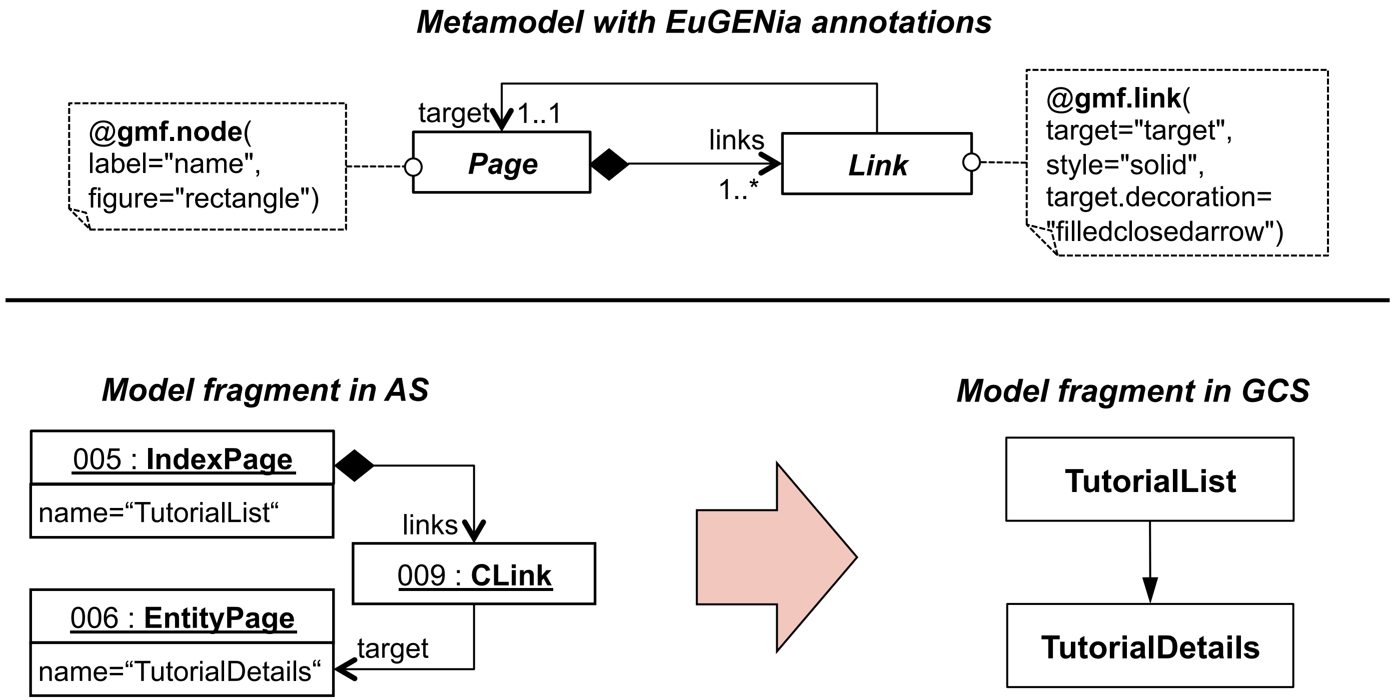
Eugenia Example #1

- *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



API-based Approach: Graphiti

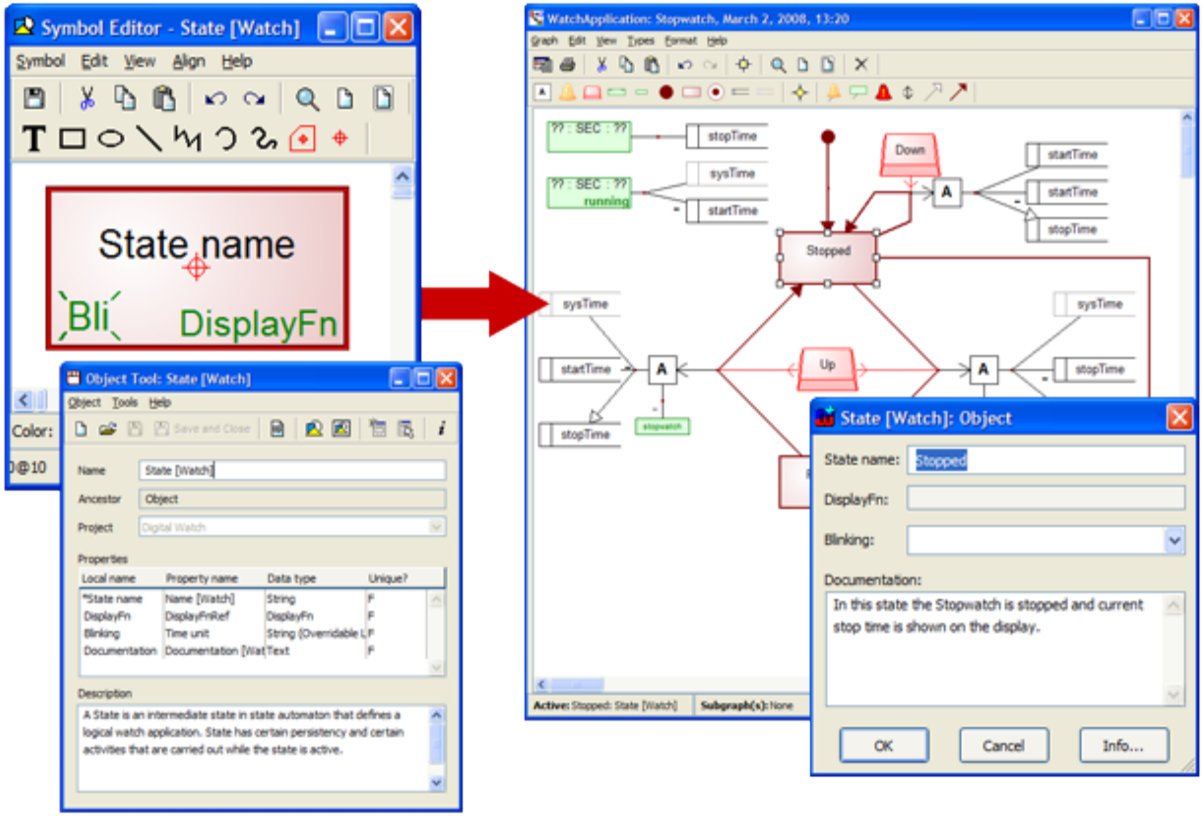
- Powerful **programming framework** for developing graphical modeling editors
- **Base classes** of Graphiti have to be **extended** to define concrete syntaxes of modeling languages
  - *Pictogram models* describe the visualization and the hierarchy of concrete syntax elements (cf. .gmfgraph models 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+

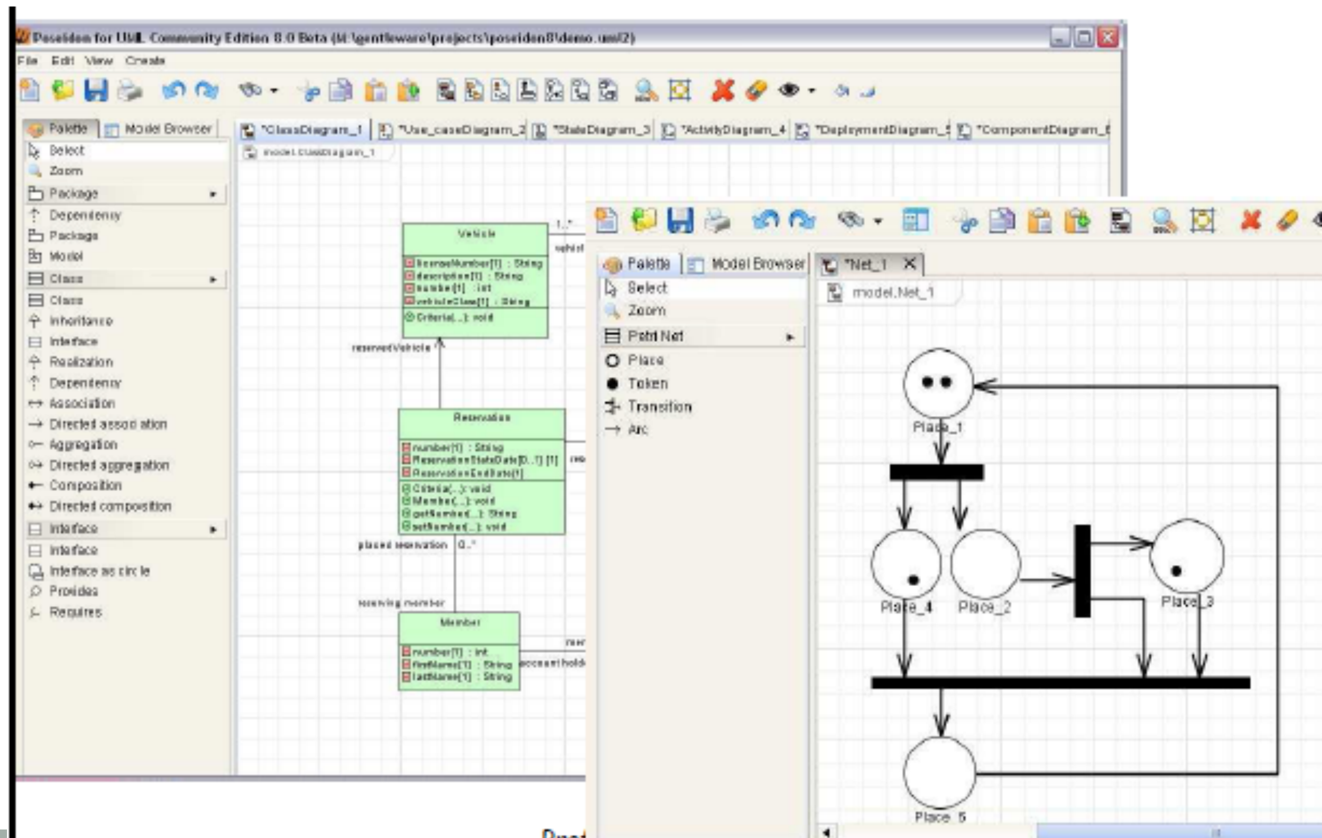
- Metamodeling tool outside Eclipse (commerical product)
- **Graphical specification of figures** in graphical editor
- Special tags to specify labels in the figures by querying the models



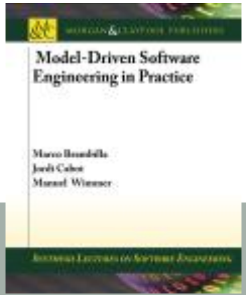
Other approaches outside Eclipse

Poseidon

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



TEXTUAL CONCRETE SYNTAX



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, …

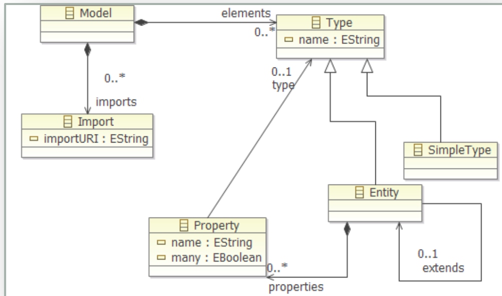
- **Compact and expressive syntax**
  - Productivity for experienced users
  - Guidance by IDE support softens learning curve

- **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!

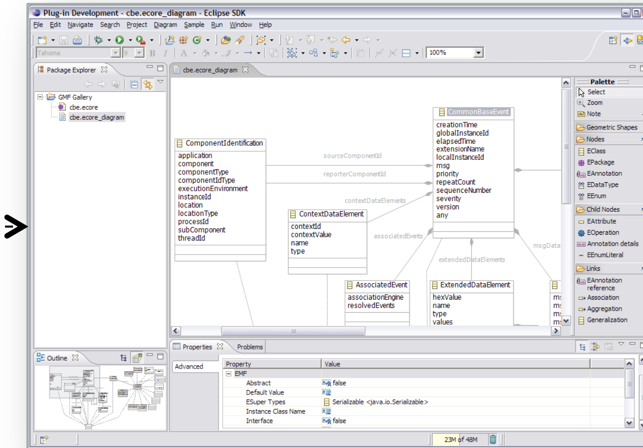
Textual Concrete Syntax

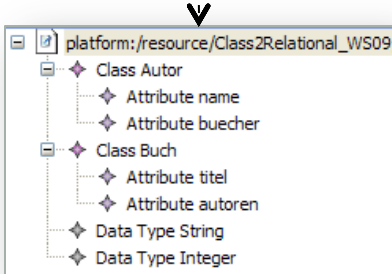
Concrete Syntaxes in Eclipse

Ecore-based Metamodels



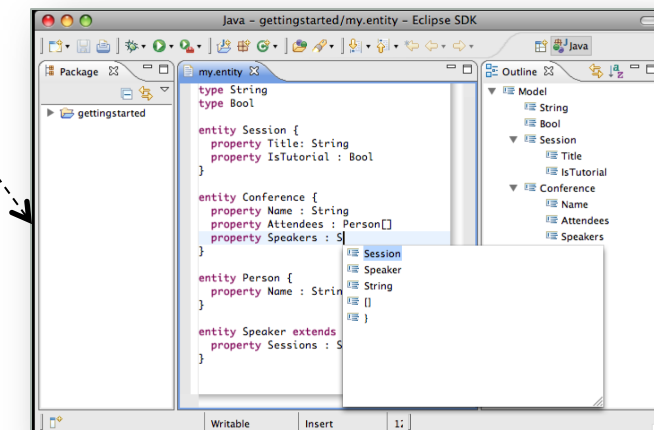
Graphical Concrete Syntax





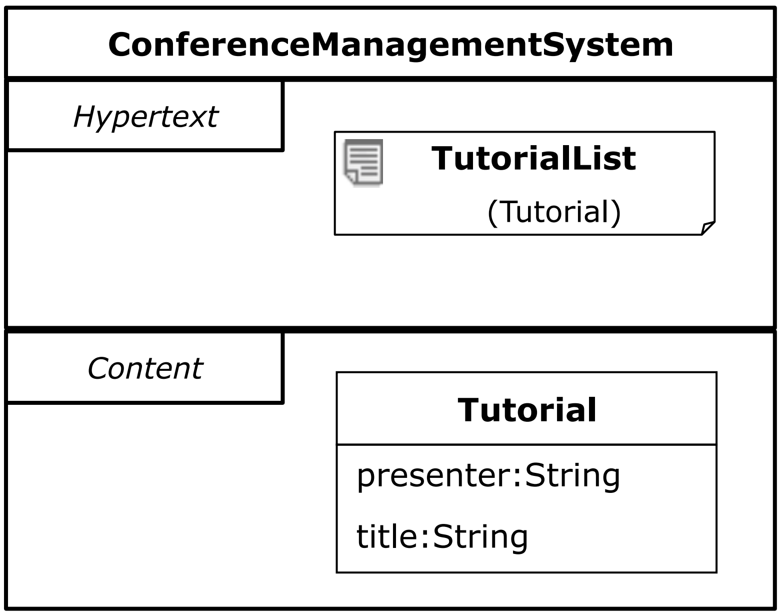
Generic tree-based EMF Editor

Textual Concrete Syntax



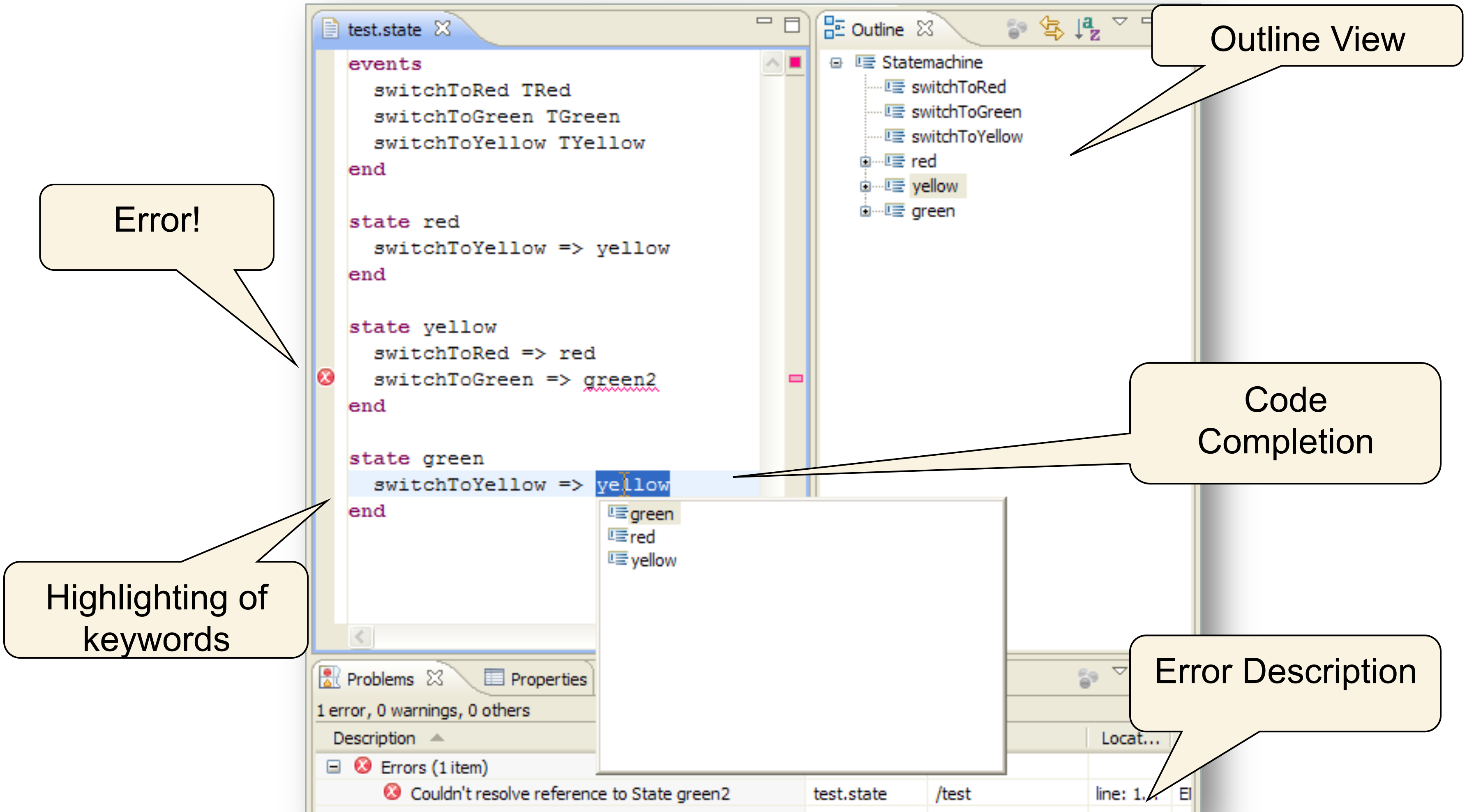
Every GCS is transformable to a TCS

Example: sWML




```
webapp ConferenceManagementSystem{

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

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

}
```

Anatomy of Modern Text Editors



Excursus: Textual Languages in the Past

Basics

- **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**

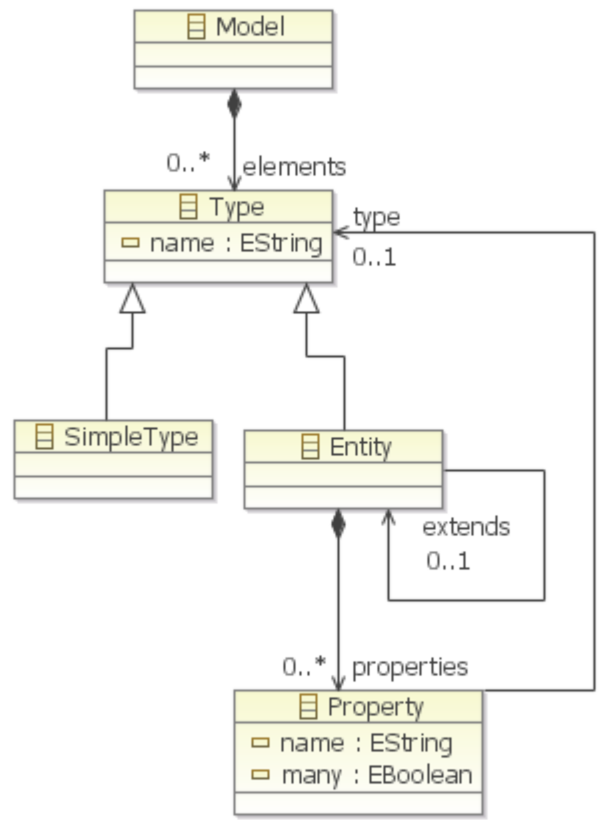
```
Model := Type*;

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: *HUTN* and *XMI* (OMG Standards)

Language-specific Syntax

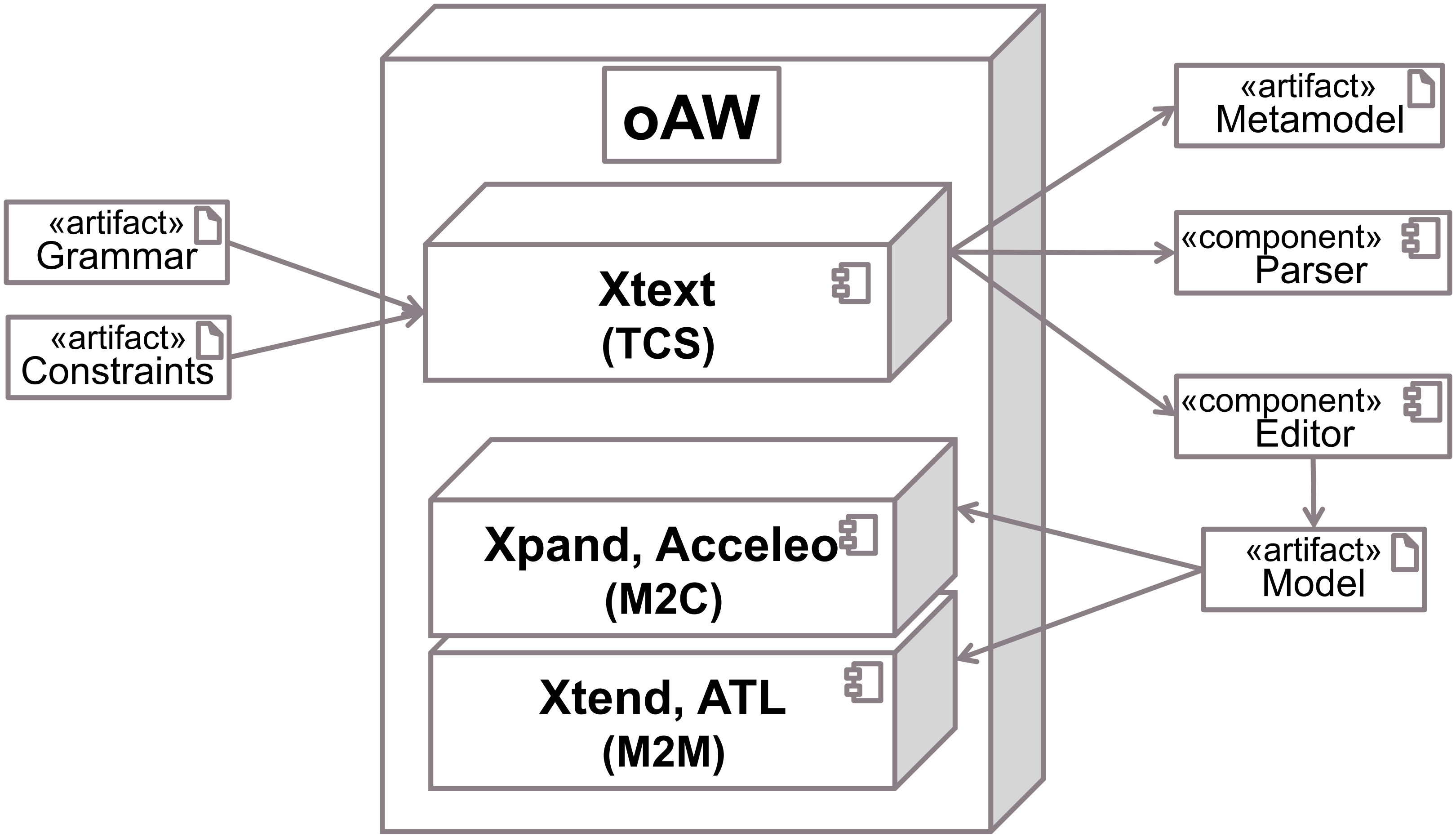
- *Metamodel First!*
  - Step 1: Specify metamodel
  - Step 2: Specify textual syntax
  - For instance: *TCS* (Eclipse Plug-in)
- *Grammar First!*
  - Step 1: Syntax is specified by a grammar (concrete syntax & abstract syntax)
  - Step 2: Metamodel is derived from output of step 1, i.e., the grammar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



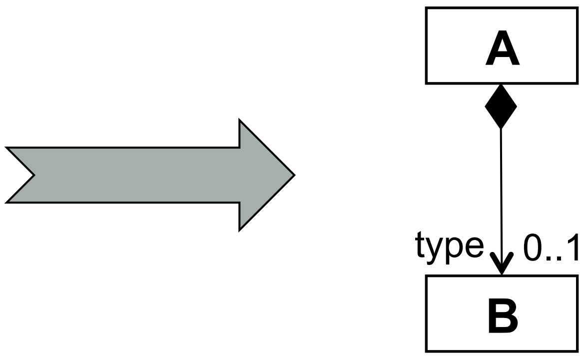
Xtext
Grammar

- **Xtext** grammar **similar** to **EBNF**

- But **extended** by
  - Object-oriented concepts
  - Information necessary to derive metamodels and modeling editors

- **Example**

A: (type=B);



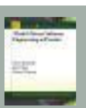
Xtext

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 -> *NonContainment 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,…)

StateMachine.xtext

```
grammar org.xtext.example.StateMachine with org.eclipse.xtext.common.Terminals

generate stateMachine "http://www.xtext.org/example/StateMachine"

Statemachine :
  'events'
     (events+=Event)+
  'end'
  ('resetEvents'
     (resetEvents+=[Event])+
  'end')?
  'commands'
     (commands+=Command)+
  'end'
  (states+=State)+;
```

Metamodel URI

Xtext

Tooling

- **Xtext Grammar Definition for State Machines**

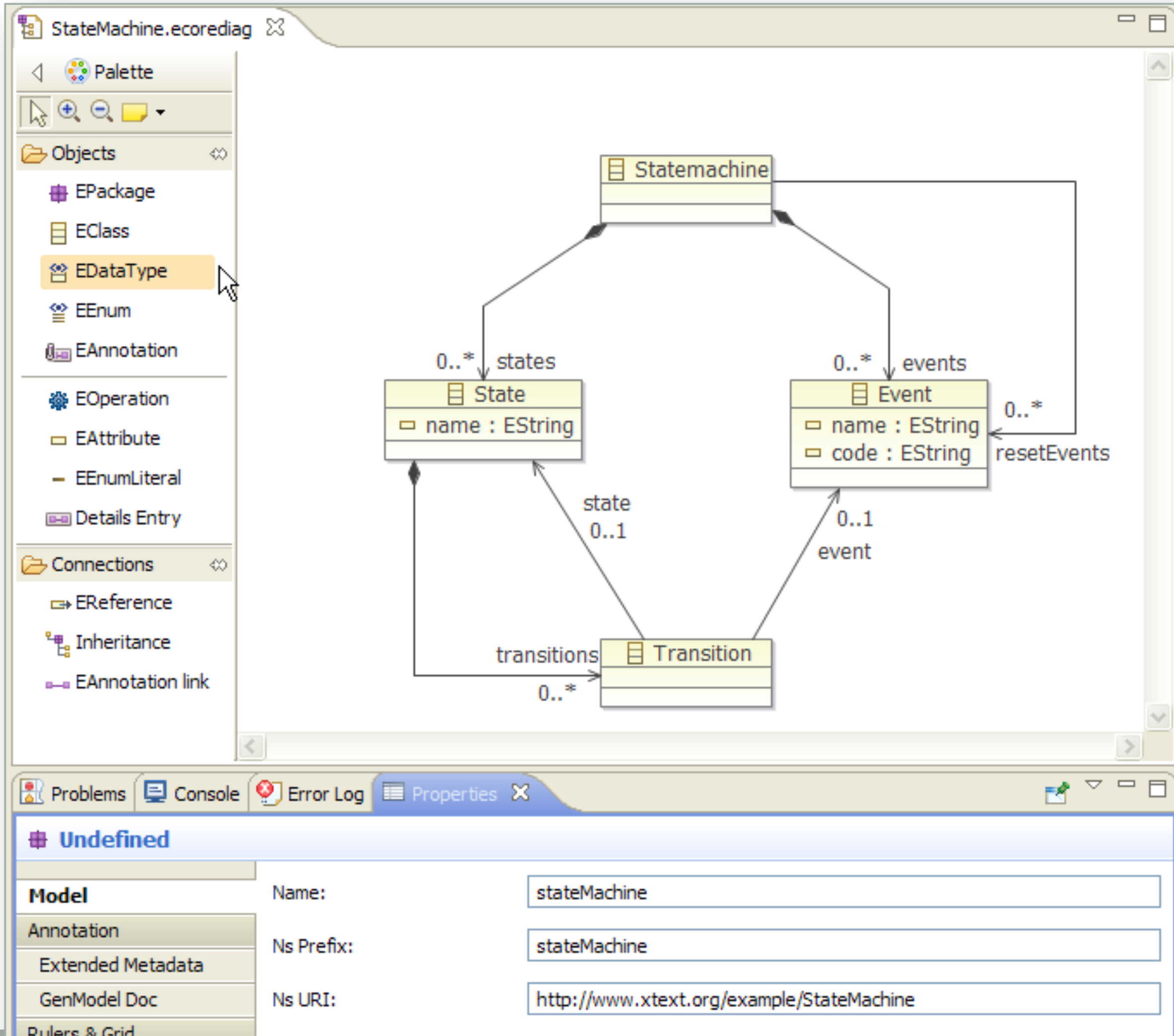
StateMachine.xtext

```
grammar org.xtext.example.StateMachine with org.eclipse.xtext.common.Terminals

generate stateMachine "http://www.xtext.org/example/StateMachine"

Statemachine :
  'events'
     (events+=Event)+
  'end'
  ('resetEvents'
     (resetEvents+=[Event])+
  'end')?
  (states+=State)+;

Event :
  name=ID code=ID;

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

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

Xtext

Tooling

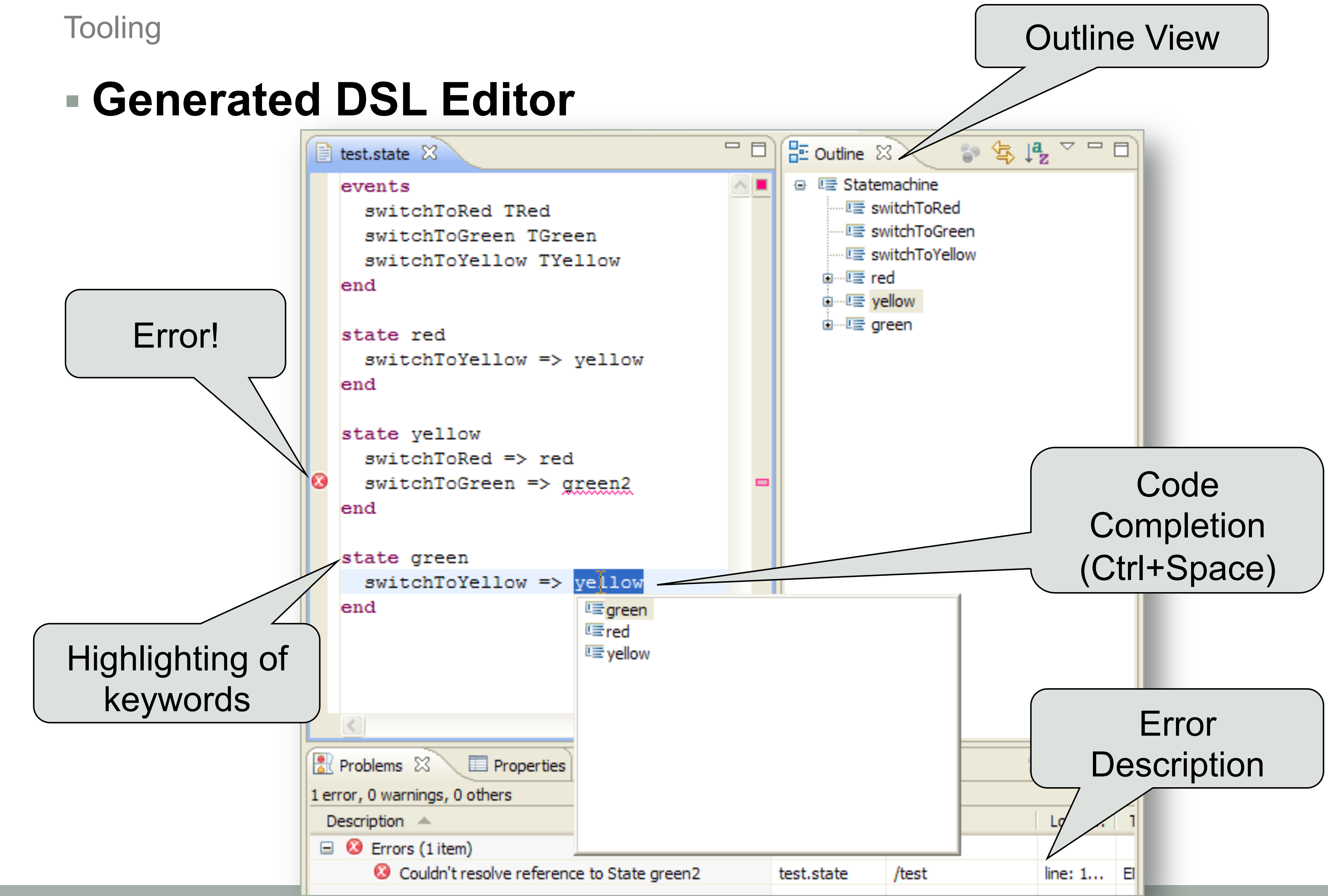
- **Automatically generated Ecore-based Metamodel**



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?

- **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!

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?

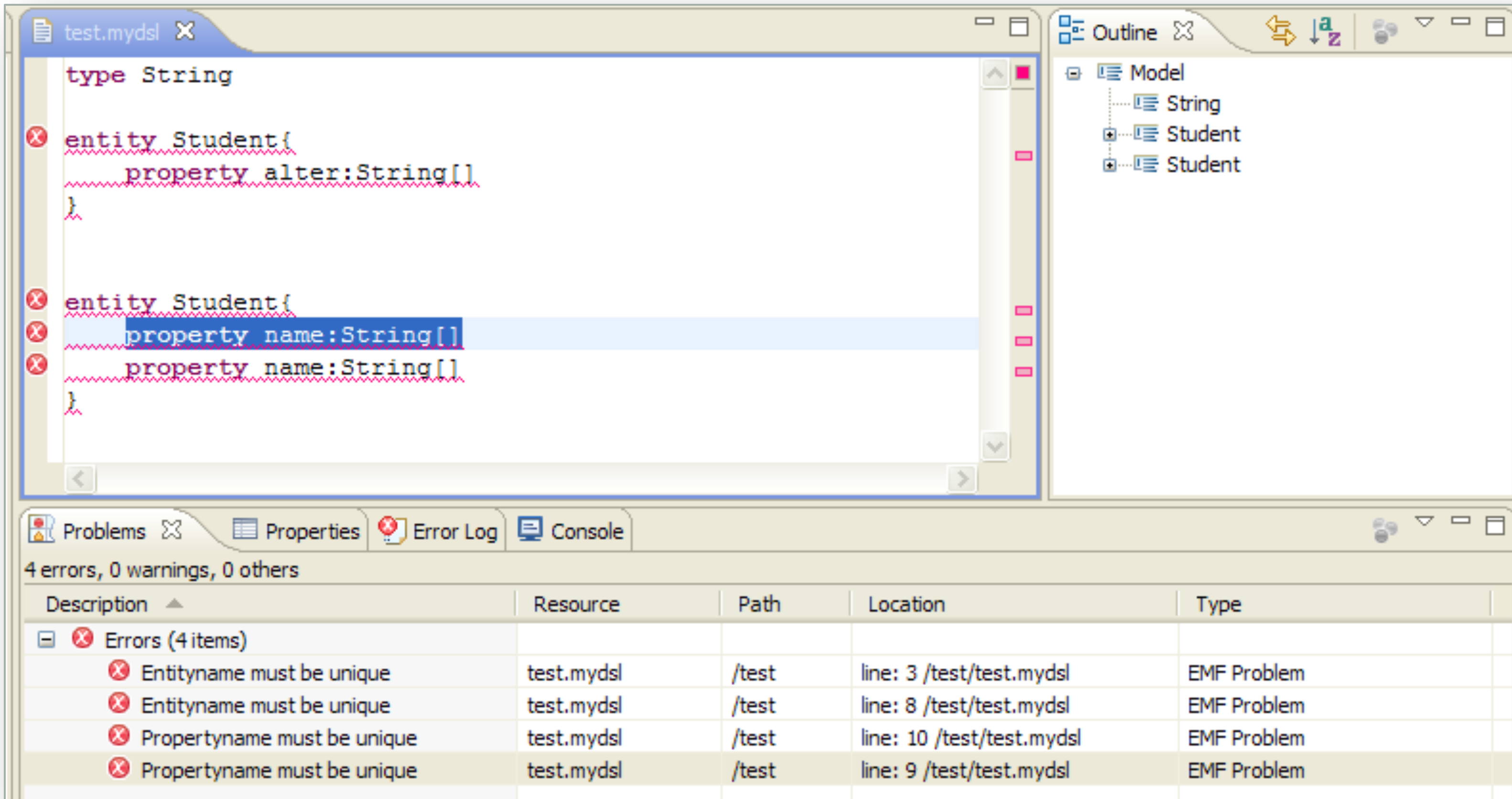
- Examples
  1. Entity names must start with an Upper Case character
  2. Entity names must be unique within one model
  3. Property names must be unique within one entity

- 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

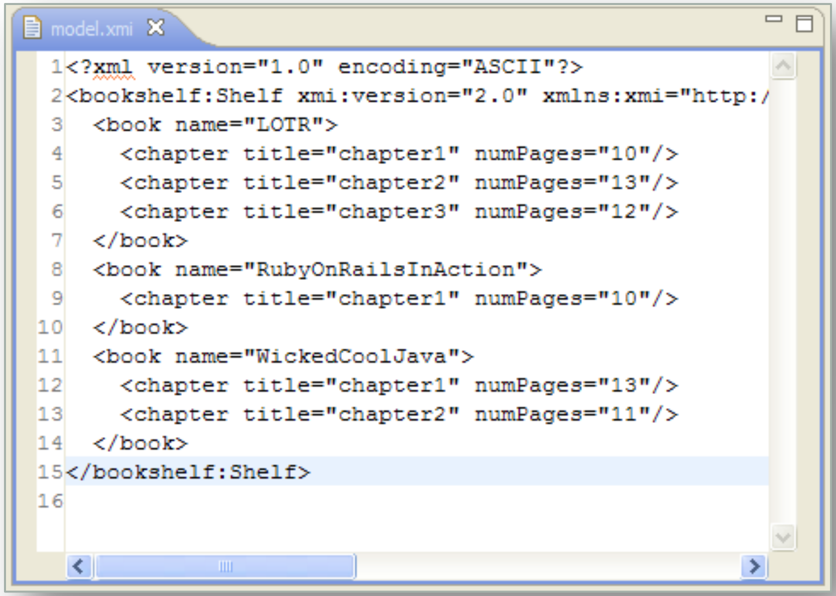
When to evaluate context sensitive constraints?

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

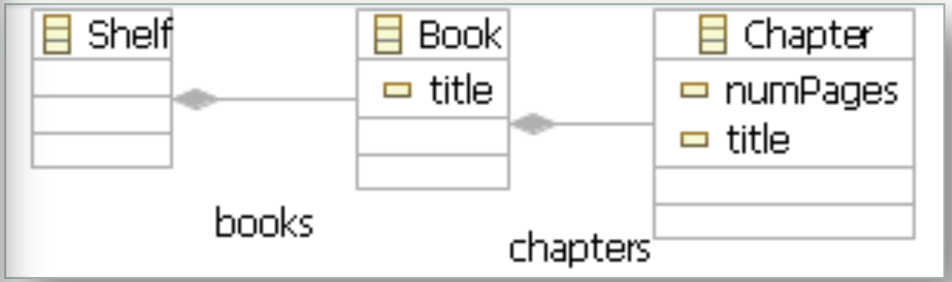


Example #2: Bookshelf (Homework)

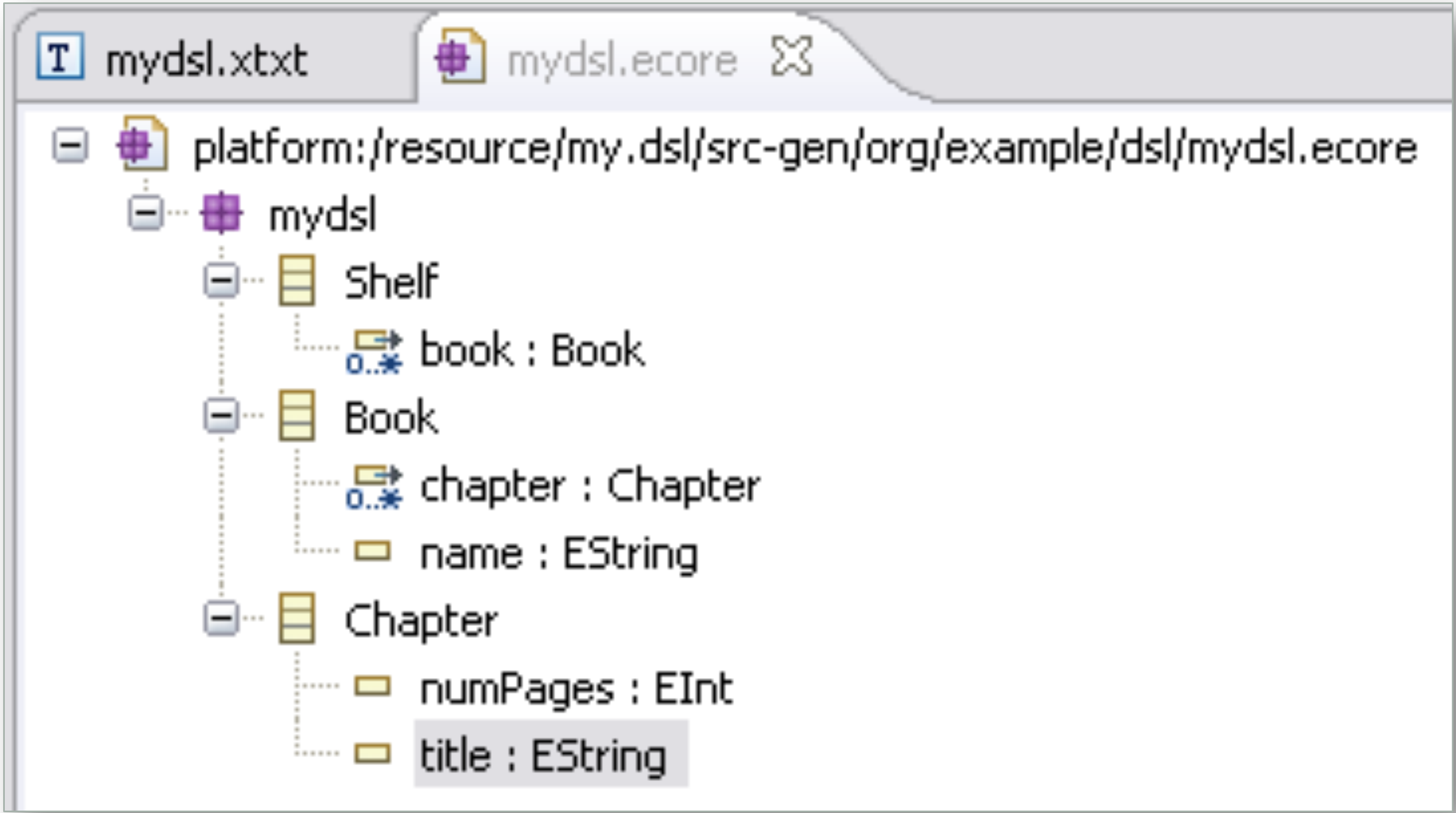
- Edit „Bookshelf“ models in a text-based fashion
- **Given**: Example model as well as the metamodel
- **Asked**: Grammar, constraints, and editor for Bookshelf DSL


model.xmi

```xml
<?xml version="1.0" encoding="ASCII"?>
<bookshelf:Shelf xmi:version="2.0" xmlns:xmi="http:/
  <book name="LOTR">
    <chapter title="chapter1" numPages="10"/>
    <chapter title="chapter2" numPages="13"/>
    <chapter title="chapter3" numPages="12"/>
  </book>
  <book name="RubyOnRailsInAction">
    <chapter title="chapter1" numPages="10"/>
  </book>
  <book name="WickedCoolJava">
    <chapter title="chapter1" numPages="13"/>
    <chapter title="chapter2" numPages="11"/>
  </book>
</bookshelf:Shelf>
```

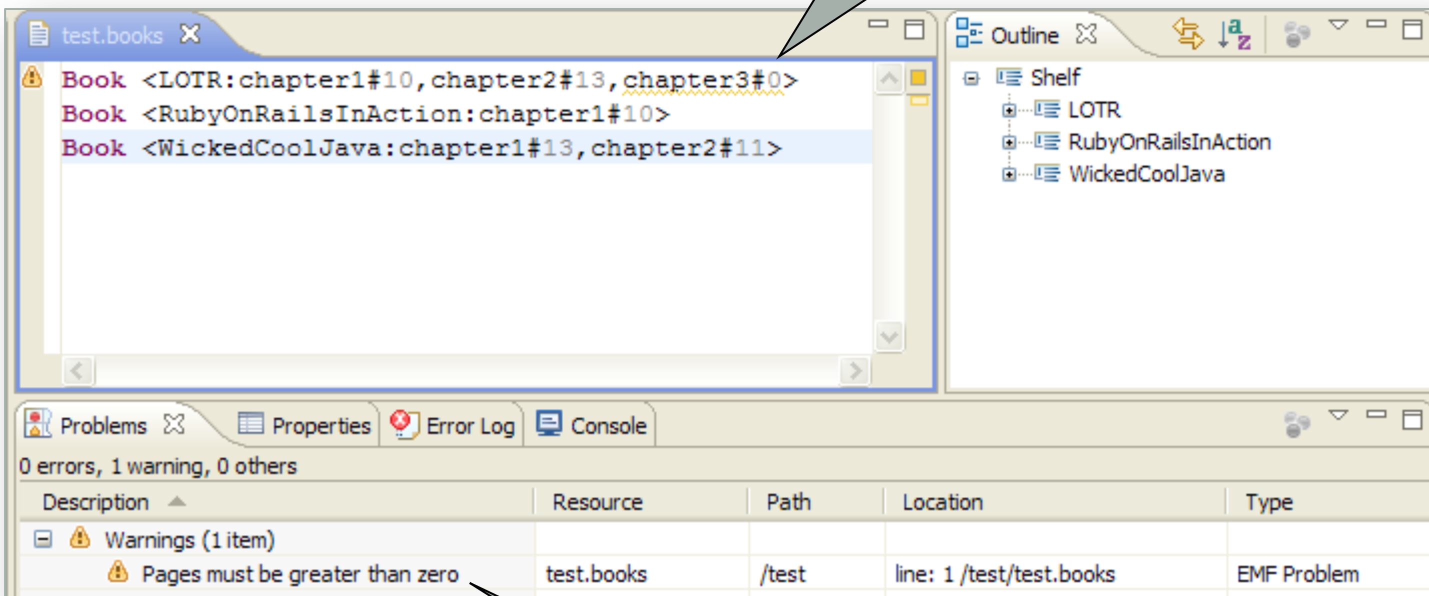


Example #2: Metamodel Details



Example #2: Editor

PageNum > 0



Give Warnings!

MORGAN & CLAYPOOL PUBLISHERS

MODEL-DRIVEN SOFTWARE ENGINEERING IN PRACTICE

Marco Brambilla,
Jordi Cabot,
Manuel Wimmer.
Morgan & Claypool, USA, 2012.

www.mdse-book.com
www.morganclaypool.com