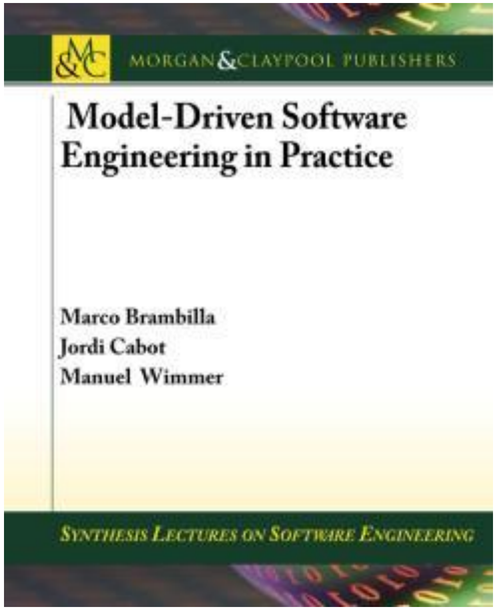
MORGAN & CLAYPOOL PUBLISHERS

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.



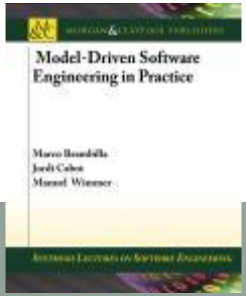
Content

- **Part A**
  - Introduction
  - Abstract Syntax

- Part B
  - Concrete Syntaxes
    - Graphical Concrete Syntax
    - Textual Concrete Syntax

- Summary

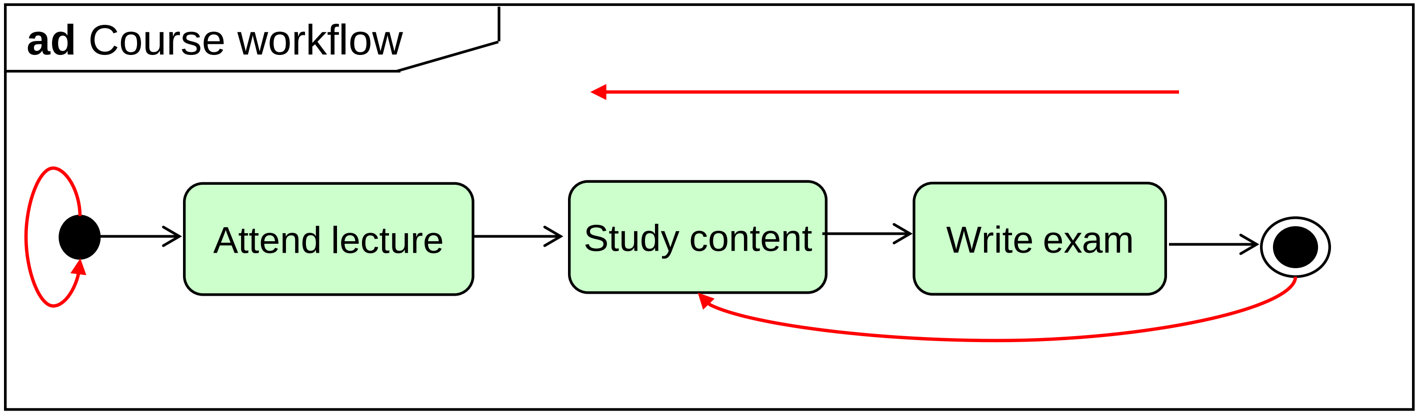
INTRODUCTION



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, …

Introduction

Overview

- What is a m
  - It is a model … mod … ges.
  - Based on a ob … tes and associations).
- What is a metam
  - Prefix "meta": an ope … applied to …
  - Examples: meta-discussion, meta-learning, ...
  - Classes, attributes and associations define the concepts and properties of modeling. Describe basic constraints.
- What is a meta-metamodeling?
  - Language how to build metamodels.
  - Capable of representing all valid models.
- Describe the **abstract syntax** of the languages they represent.

"Modeling how to model"

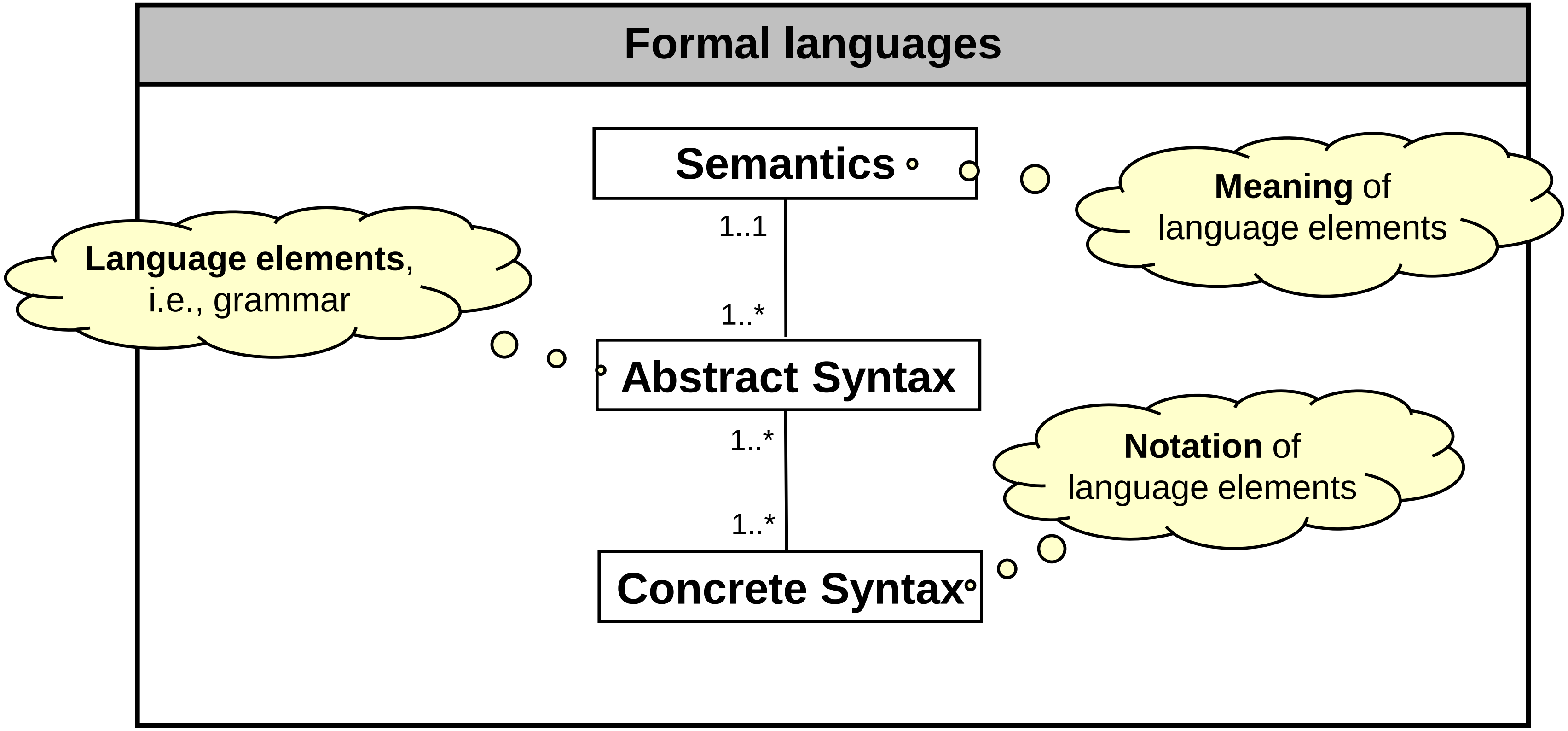
…scussing how to discuss.."

"Learning how to learn.."

Introduction

Anatomy of formal languages 1/2

- Languages have **goals** and **divergent** fields of applications, but still having a **common 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 (*Backus-Naur Form*), EBNF (*Extended Backus-Naur Form*), …

- Are used since 1960 for the **definition** of the **syntax** of **programming languages**
  - Remark: **abstract** and the **concrete** syntax are both defined

- EBNF (*Extended Backus-Naur Form*)
  - Code that **expresses the grammar** of a formal language.
  - Composed by **terminal symbols** and production **rules for non-terminals** that are restrictions on how terminal symbols can be combined into a sequence.

Programming languages

Overview

- 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

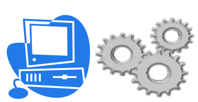
```
package mdse.book.example;
import java.util.*;
public class Student extends Person { … }
```

- 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 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
  - Structured representation of the data

- **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 | 0..1 | attributes

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)

Example	Description	Layer
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**

Introduction

What to expect from this lecture? (1/2)

1. **Metamodel-centric language design**

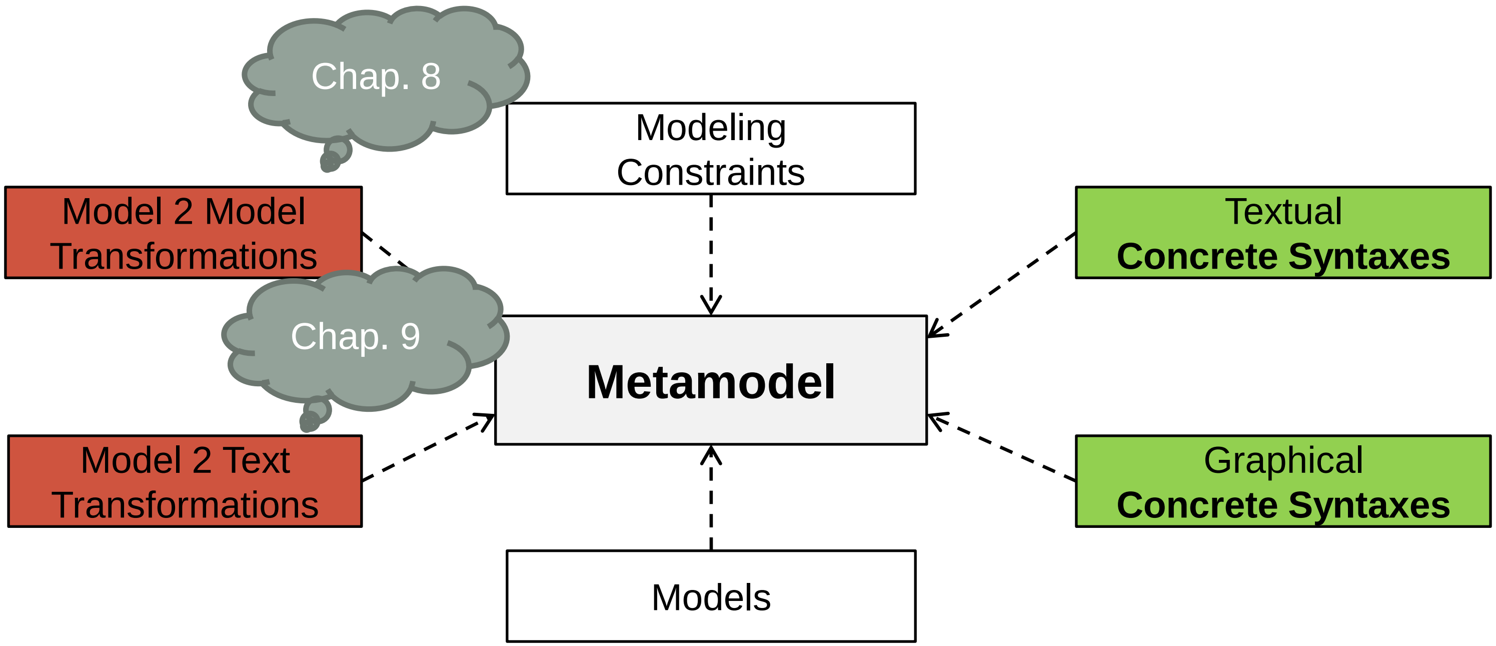
2. Define the **abstract syntax** of the modeling language with its restrictions

3. Define **concrete syntaxes** (textual and graphical) based on the **abstract syntax**

4. Show how the **resources of Eclipce** can be used to perform previous activities.

Introduction

What to expect from this lecture? (2/2)



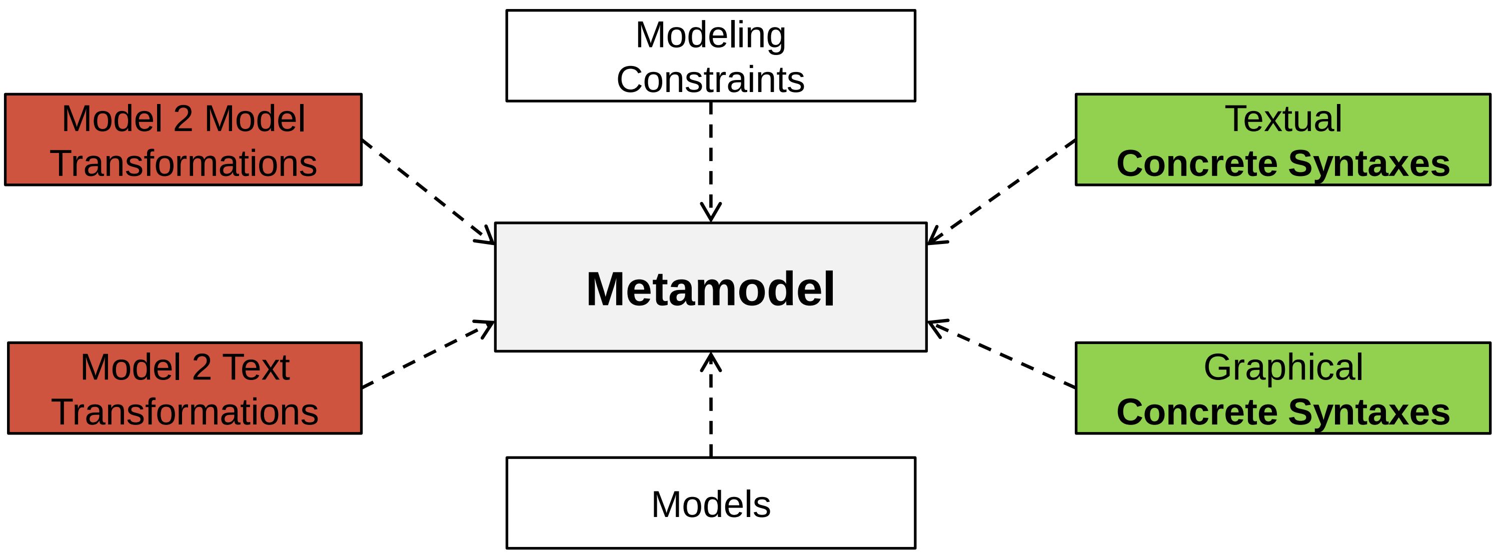
ABSTRACT SYNTAX

Introduction

Spirit and purpose of metamodeling 1/4

- **Metamodel-centric language design**

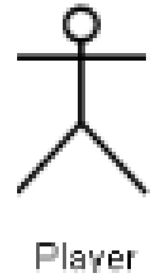
All aspects of language going beyond abstract syntax of a modeling language have in common that they are defined in terms of the metamodel



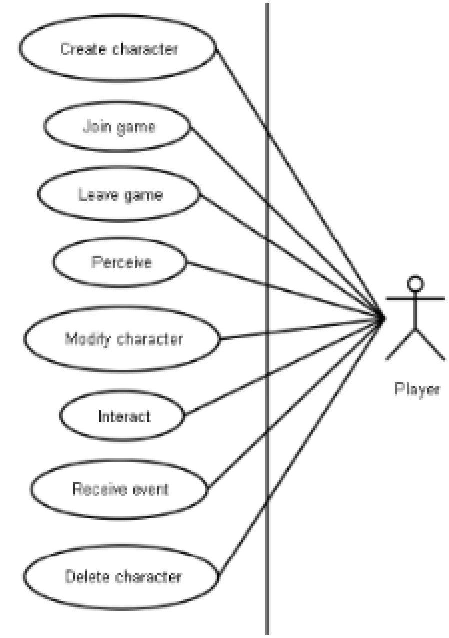
Introduction

Spirit and purpose of metamodeling 2/4

- Example
  - Actor has *properties* and *relationships*, this context can be understood as an **abstraction** of something (e.g. player)



  - When this actor is associated with a use case diagram, it receive a **semantic representation** in some contexts (e.g. join game or leave game), i.e. we give a **meaning** to it.
  - However, to create this use case diagram, we must *to follow* a **rules** defined to this kind of modeling language, its **metamodel**



Introduction

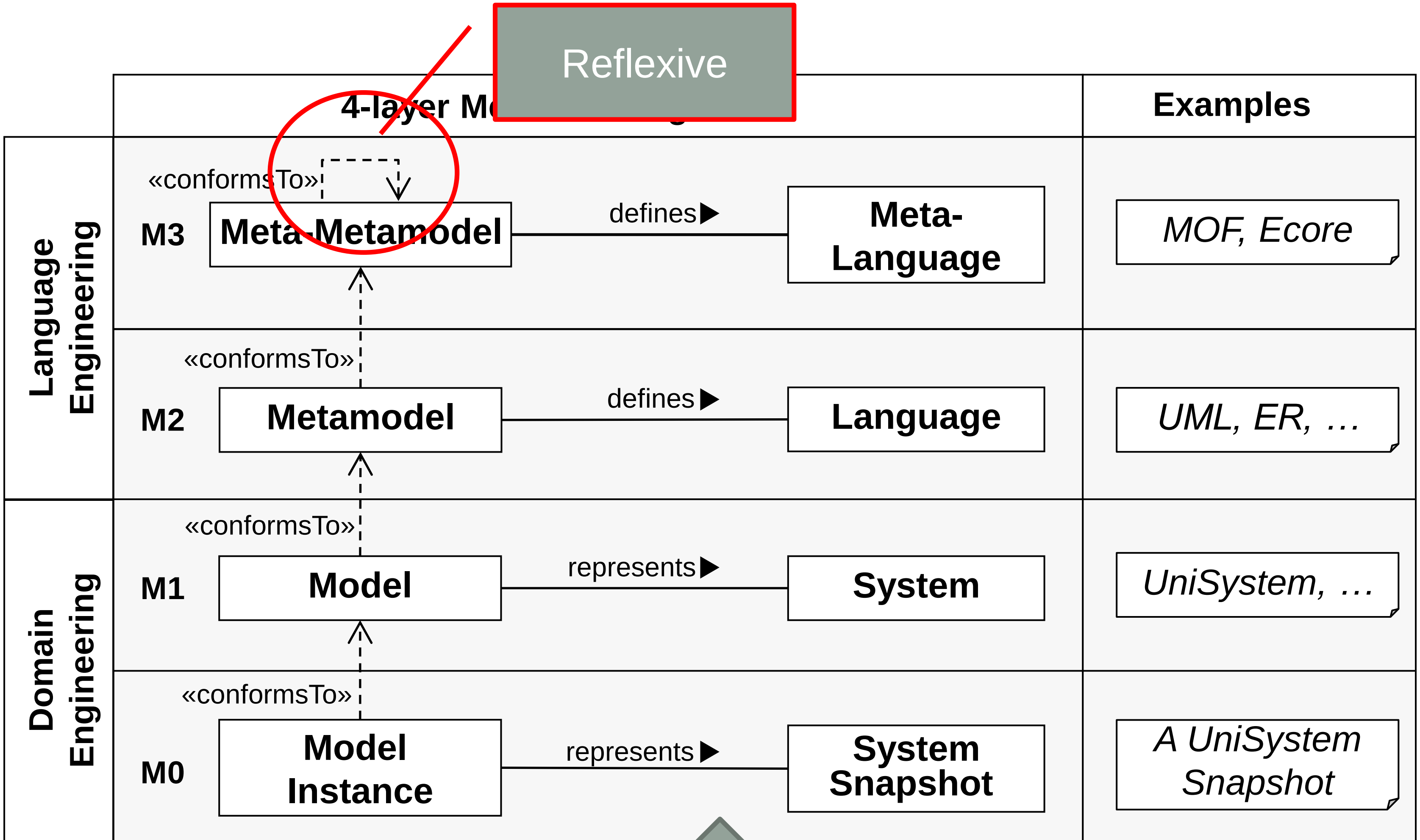
Spirit and purpose of metamodeling 3/4

- **Advantages** of metamodels
  - **Precise:** There is a formal definition of the language syntax which is processable by machines
  - **Accessible:** The **knowledge of UML class diagrams** is sufficient to read and understand
  - **Evolvable language definition:** Have an **accessible language definition** further contributes to an **easy adaptation of modeling languages**

- **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

Introduction

Spirit and purpose of metamodeling 4/4



M0: there are the instances of the domain concepts which represents real-word entities

MOF - Meta Object Facility

Introduction 1/2

- **OMG standard** for the **definition of metamodels**

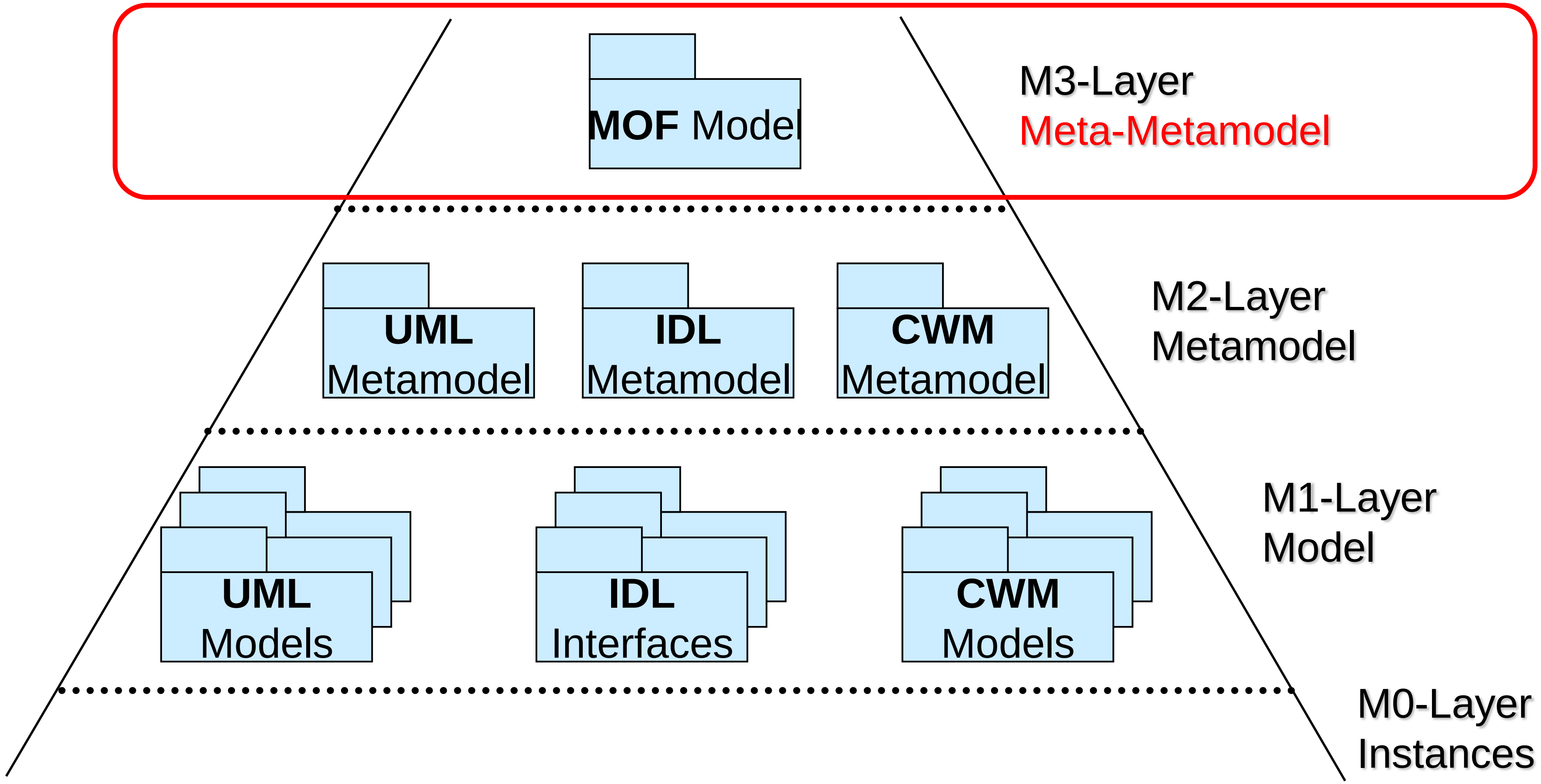
- MOF is an **object-orientated** modeling language
  - **Objects** are described by **classes**
  - **Intrinsic properties** of objects are defined as **attributes**
  - **Extrinsic properties** (links) between objects are defined as **associations**
  - **Packages** group classes

- 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 2/2

- OMG language definition stack



Why an additional language for M3

… isn‘t UML enough?

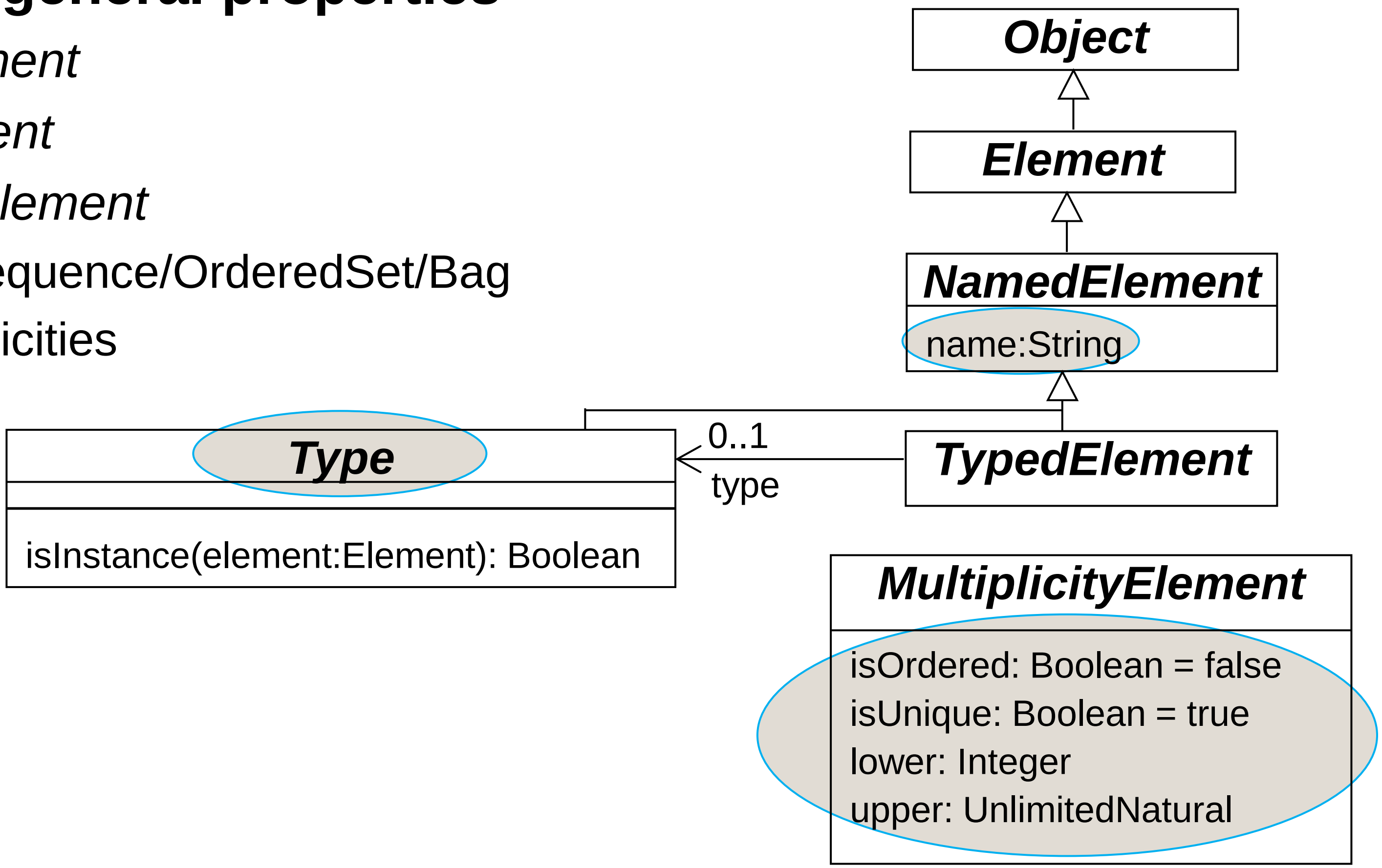
- **MOF** only a **subset** of **UML**
  - MOF is **similar** to the UML class diagram, but much more limited
  - No n-ary associations, no association classes, …
  - No overlapping inheritance, interfaces, dependencies, …
- Main differences result from the **field of application**
  - UML
    - Domain: **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

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

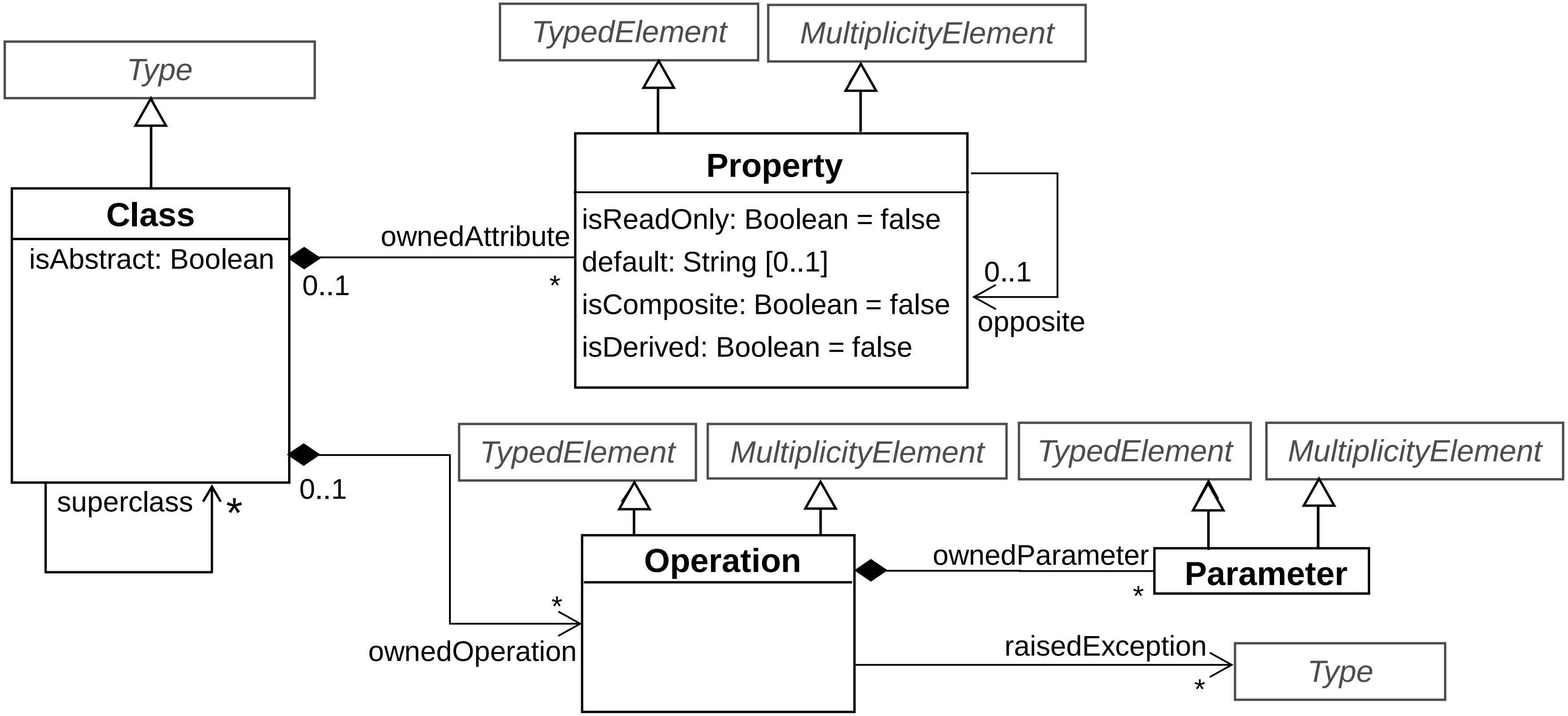
Taxonomy of abstract classes



MOF – Meta Object Facility

Language architecture of MOF 2.0

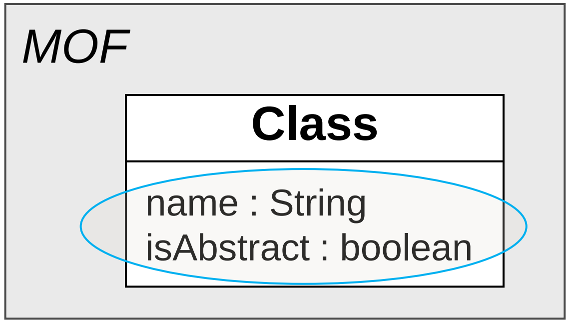
- **Core** of eMOF
  - Based on object-orientation
  - Classes, properties, operations, and parameters

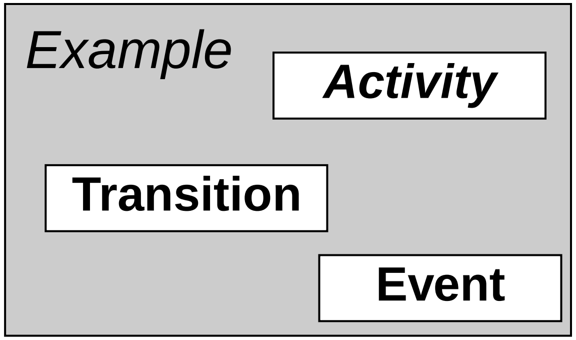


MOF – Meta Object Facility

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!

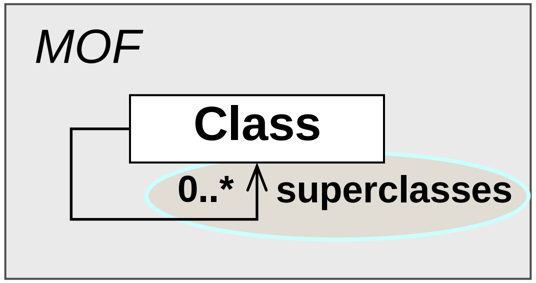


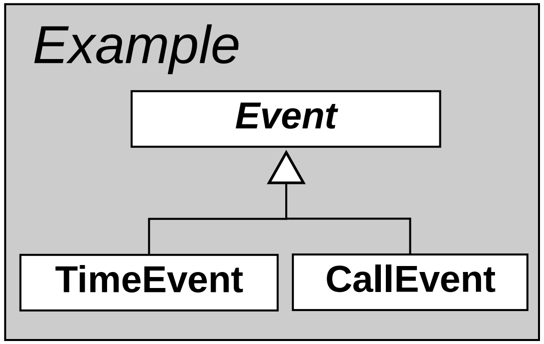


MOF – Meta Object Facility

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**

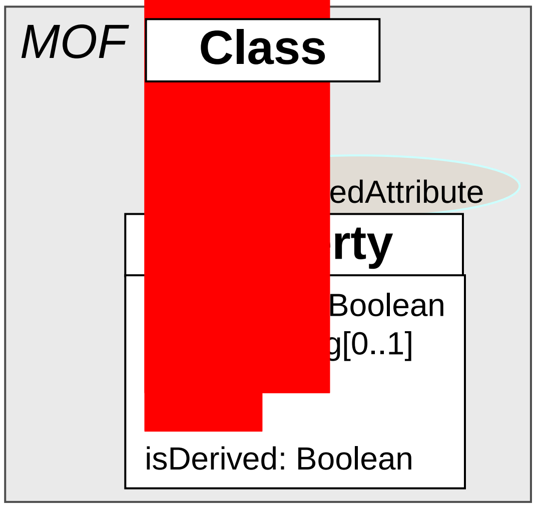


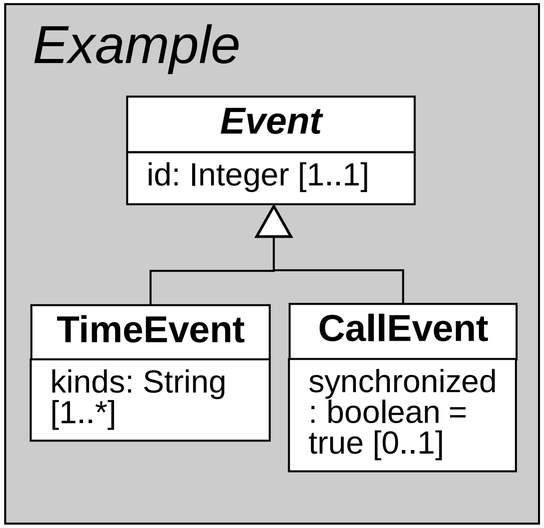


MOF – Meta Object Facility

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

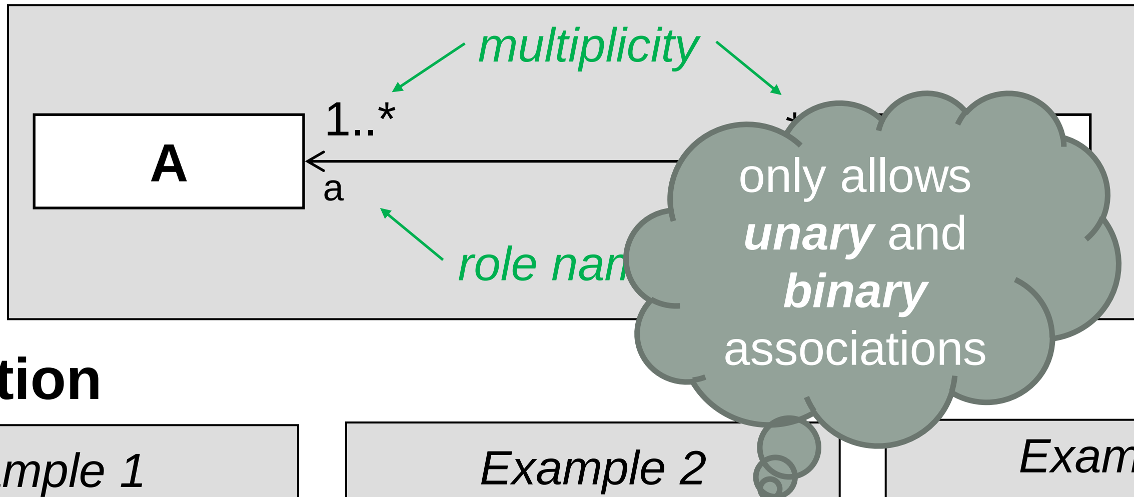
Associations

- 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

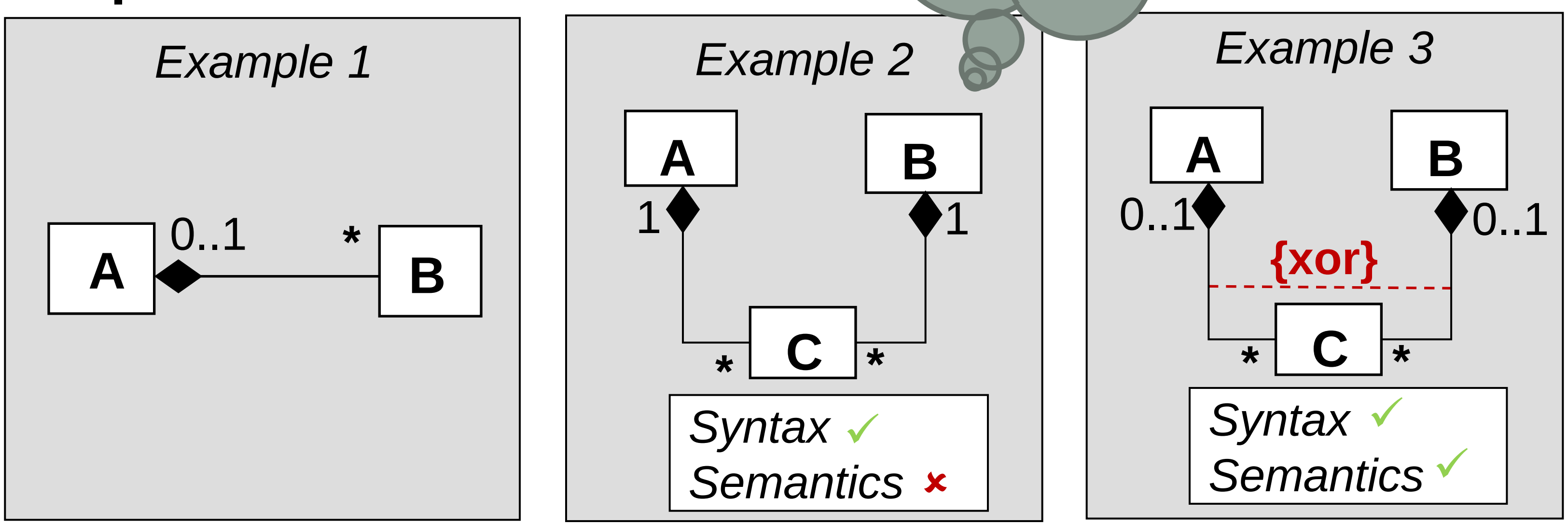
MOF – Meta Object Facility

Associations - Examples

- **Association**



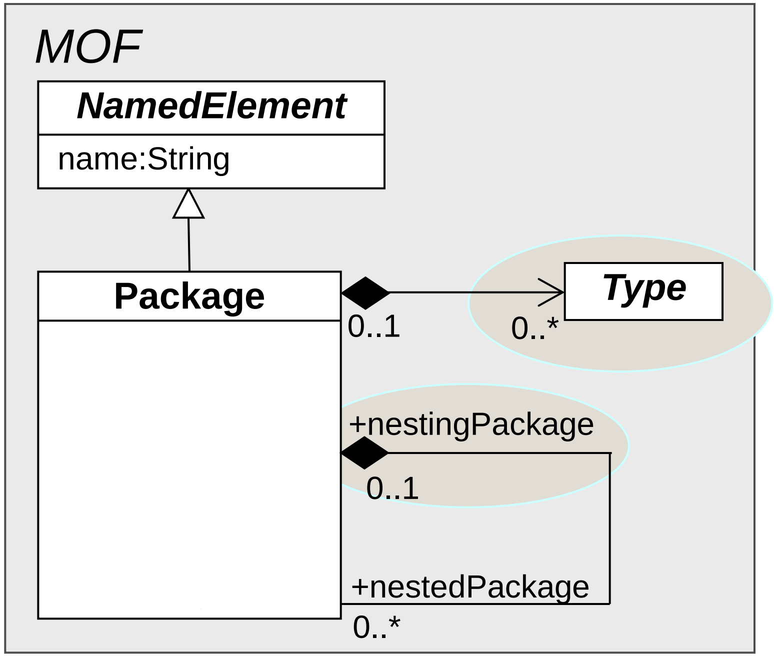
- **Composition**

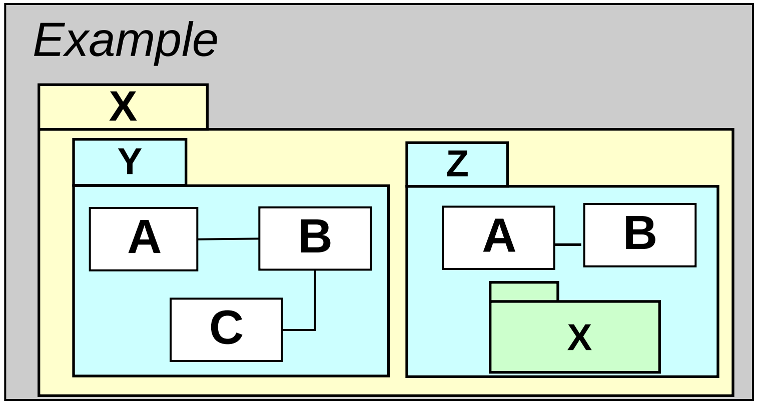


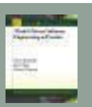
MOF – Meta Object Facility

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 by **one** package



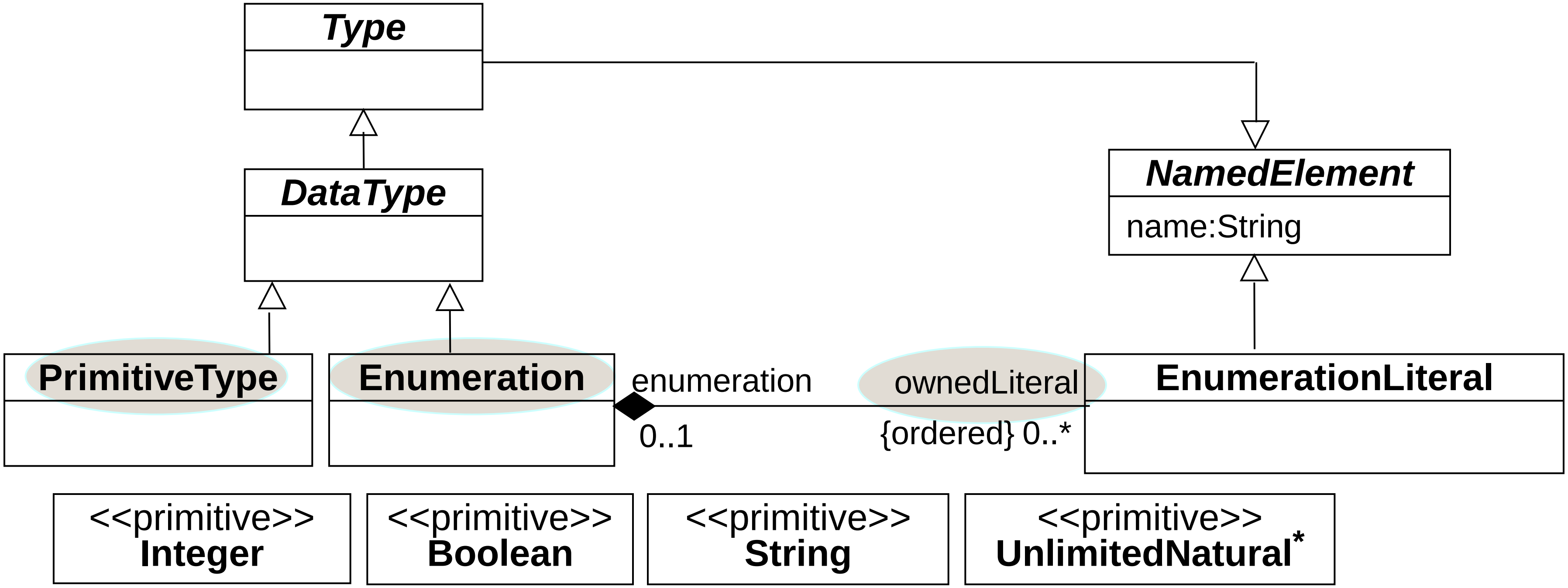




MOF – Meta Object Facility

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

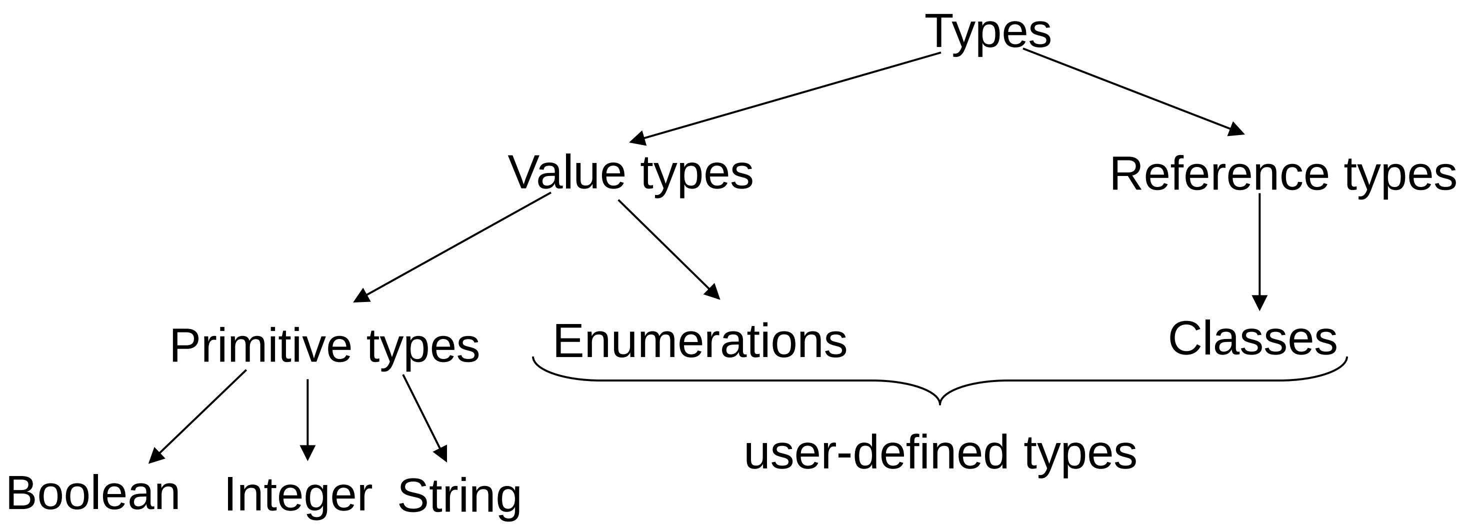


*) represents *unlimited number (asterisk)* – only for the definition of the **upper limits** of multiplicities

MOF – Meta Object Facility

Types 2/2

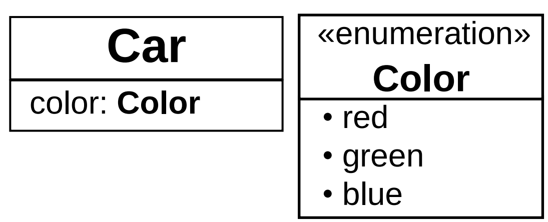
- Differentiation between ***value types*** and ***reference types***
  - Value types: contain a direct value (e.g., 123 or ‘x‘)
  - Reference types: contain a reference to an object



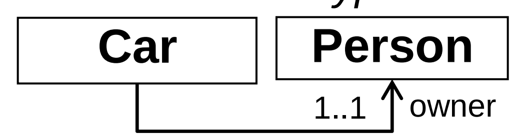
- **Examples**

Primitive types

Enumerations

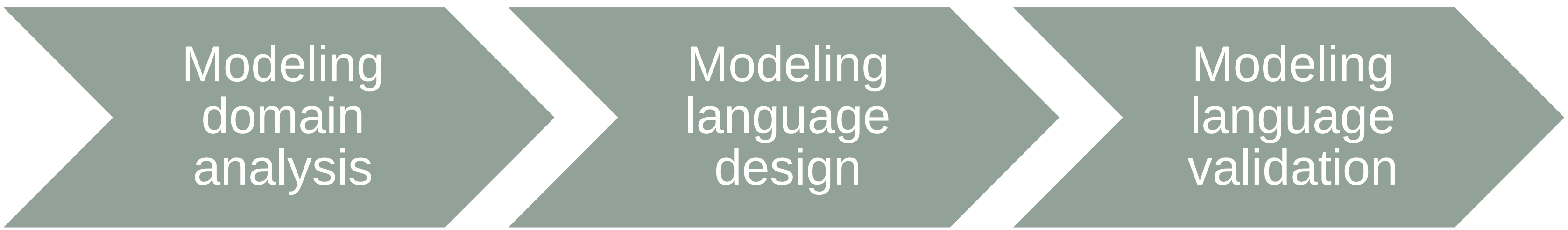


Reference types



Metamodel development process

Incremental and Iterative (1/2)



Identify *purpose*, *realiza-tion*, and *content* of the modeling language

Sketch reference modeling examples

Formalize modeling language by defining a metamodel

Formalize modeling constraints using OCL

Instantiate metamodel by modeling reference models

Collect feedback for next iteration

Metamodel development process

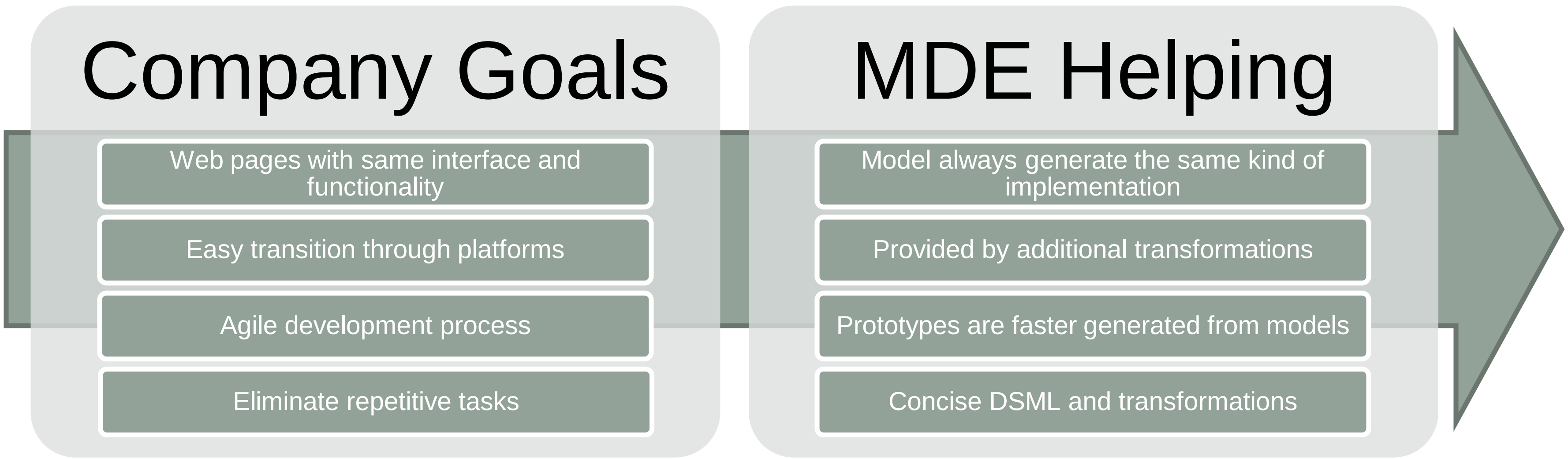
Incremental and Iterative (2/2)

- To get *feedback* from **domain experts**, a **concrete syntax** is also needed
- Both **syntactical concepts** are *encouraged* to be developed **together** in practical settings

Example

Context

- **sWML Example**
  - Company is **repeatedly** building simple Web applications, which all comprise **similar functionality**.
  - Web applications uses MVC pattern => Java (*Model*), JSF (*View*), Servlets (*Controller*) and Apache Tomcat (*Server*).
  - For each table, the Web applications implements a **simple CRUD**.



Metamodel development process

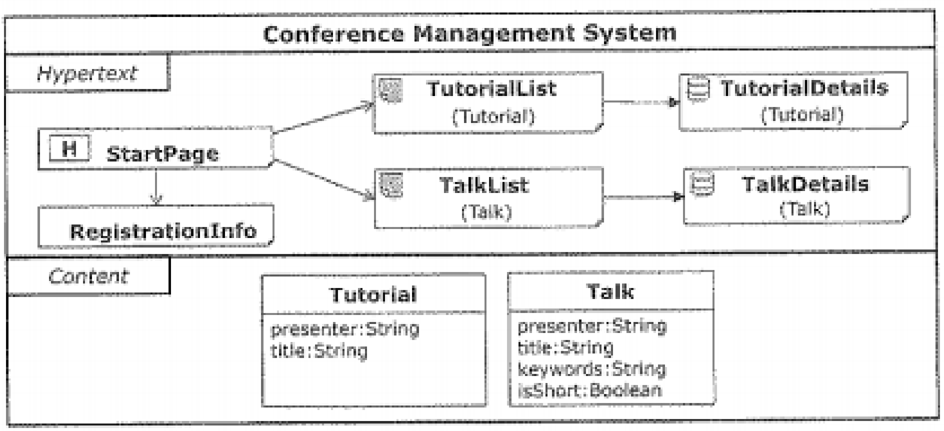
Incremental and Iterative

Modeling domain analysis → Modeling language design → Modeling language validation

Modeling domain analysis	Modeling language design	Modeling language validation
Identify *purpose*, *realization*, 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

Example: sWML

- **Several sources of information** must be exploited (use interviews, document analysis, …)
- Web application of a *conference management system*.

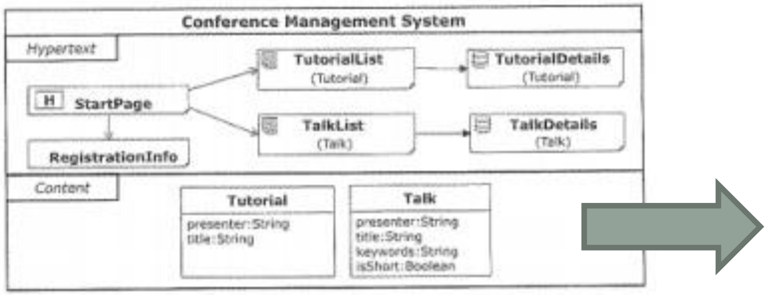


Purpose: Should modeling the content and hypertext layer.

Realization: a graphical syntax should be defined to allow the discussing with domain experts and also a textual syntax to allow the transition from model-driven to programming languages.

Example: sWML

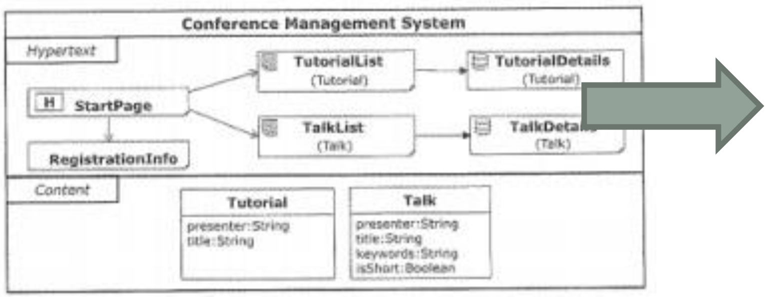
- **Content: *Content***



- **Not limit** for classes
- Classes must have a **unique** name and **multiple** attributes
- **Types**: *String, integer, Float, Boolean* and *Email*
- Must select one of its attributes as its **representative** attribute

Example: sWML

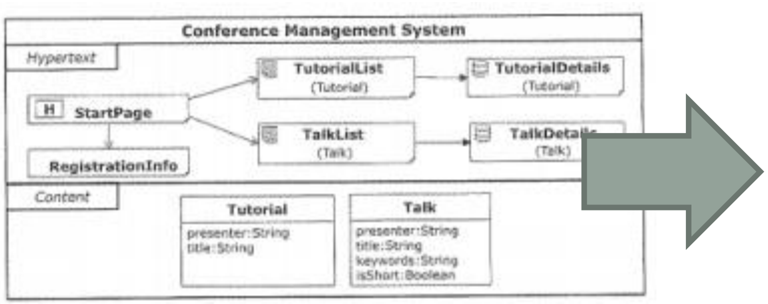
- **Content: *Hypertext***



- Different kinds of pages
- One identified **homepage**
- Subdivided into **static** and **dynamic** pages
- Dynamics are subdivided into ***details*** and ***index*** pages

Example: sWML

- **Content: *Links***



- Navigations between pages
- NCLinks: standard links
- Clinks: transport information

Metamodel development process

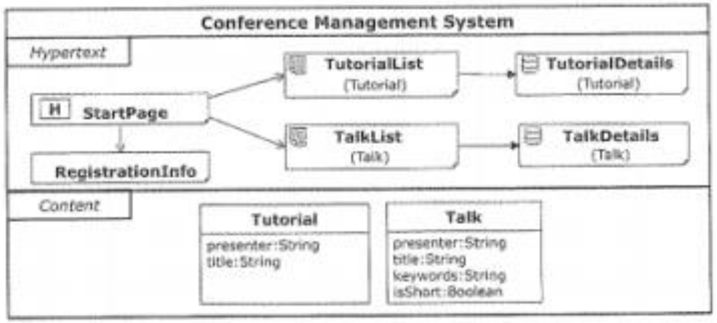
Incremental and Iterative

Modeling domain analysis	**Modeling language design**	Modeling language validation
Identify *purpose*, *realization*, 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

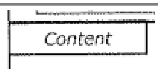
Example: sWML

Identification of the modeling concepts

Example model



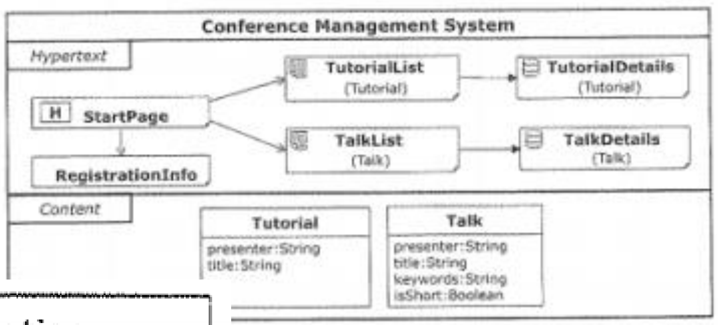
Notation table

Syntax	***Concept***
	Web Model
	Content Layer
	Class
presenter:String	Attribute
	Hypertext Layer
	Static Page
	Index Page
	Details Page
	NC Link
	C Link

Example: sWML

Determining the properties of the modeling concepts

Example model

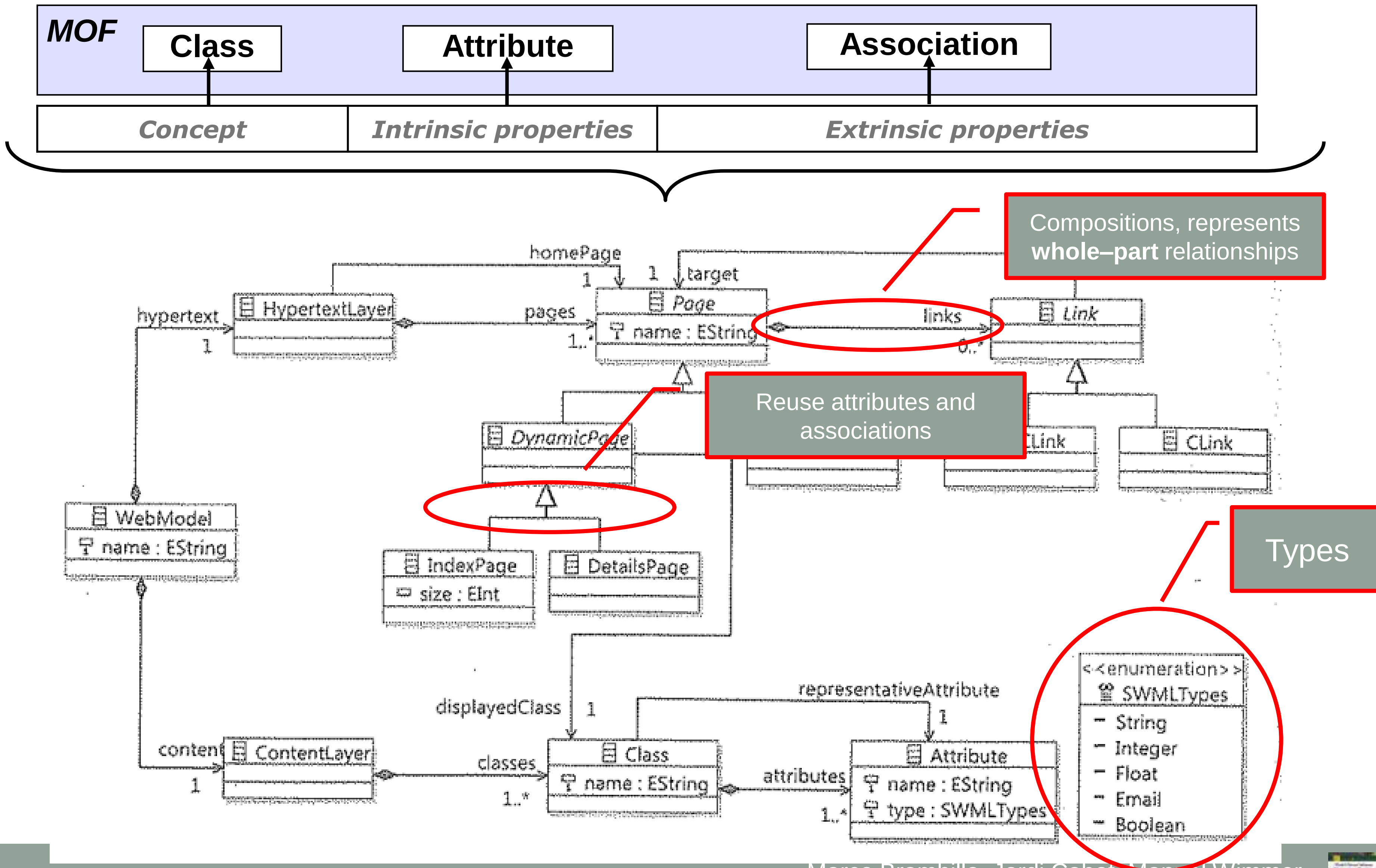


Modeling concept table

Concept	Intrinsic Properties	Extrinsic Properties
Web Model	name : String	One *Content Layer* One *Hypertext Layer*
Content Layer		Arbitrary number of *Classes*
Class	name : String	Arbitrary number of *Attributes* One representative *Attribute*
Attribute	name : String type : [String|Integer|Float|..]	
Hypertext Layer		Arbitrary number of *Pages* One *Page* defined as homepage
Static Page	name : String	Arbitrary number of *NCLinks*
Index Page	name : String size : Integer	Arbitrary number of *NCLinks* and *CLinks* One displayed *Class*
Details Page	name : String	Arbitrary number of *NCLinks* and *CLinks* One displayed *Class*
NC Link		One target *Page*
C Link		One target *Page*

Example: sWML

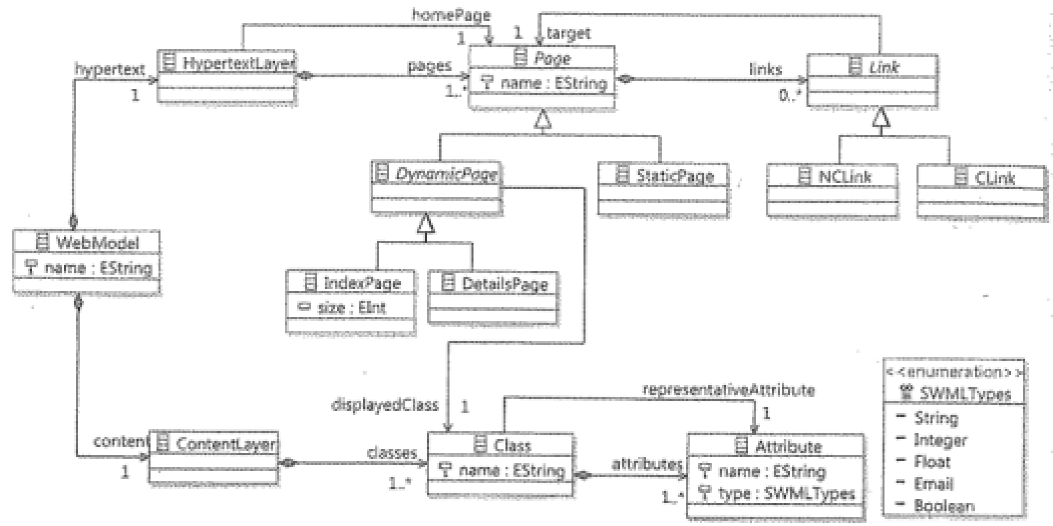
Object-oriented design of the language



Example: sWML

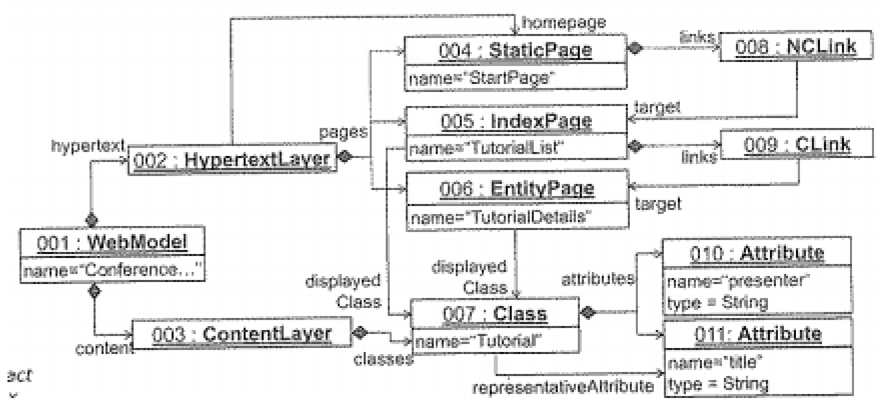
Overview

Metamodel

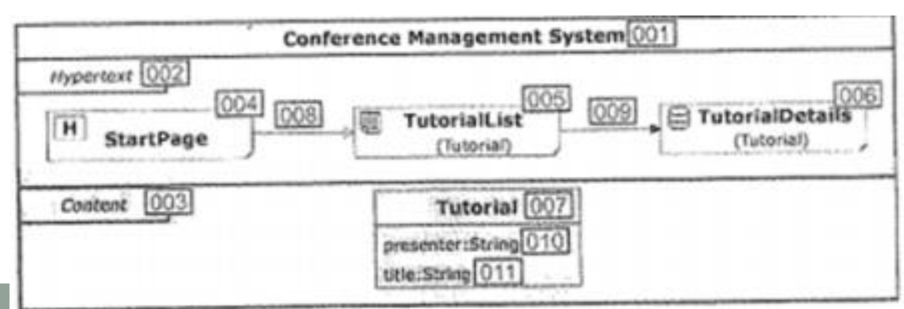


Model

Abstract syntax



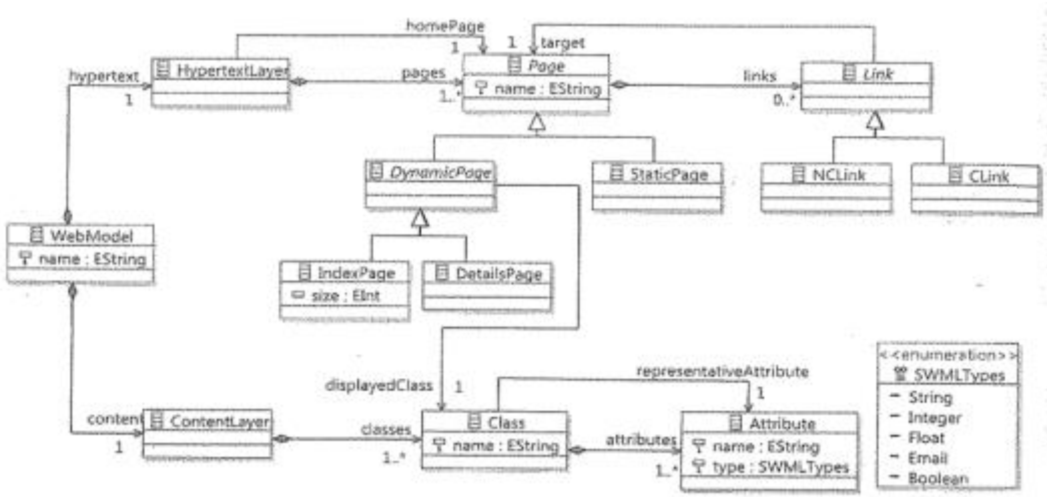
Concrete syntax



Example: sWML

Applying constraints

Metamodel



OCL Constraints

```
context ContextLayer
inv: self.classes -> forAll(x,y|x <> y, implies x.name <> y.name)

context Class
inv: self.attributes -> includes (self.representativeAttribute)

context Page
inv: not self.links -> select (l | l.oclIsTypeOf (NCLinks)) ->
exits (l|l.target = self)
```


Metamodel development process

Incremental and Iterative

Modeling domain analysis	Modeling language design	Modeling language validation
Identify *purpose*, *realization*, 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

Example: sWML

- Models are a **collection of objects**

- Use a **object diagrams** to instantiate class diagram, following the rules above:
  - Objects -> Classes
  - Values -> Attributes
  - Links -> Associations

Example: sWML

Instance of metamodel

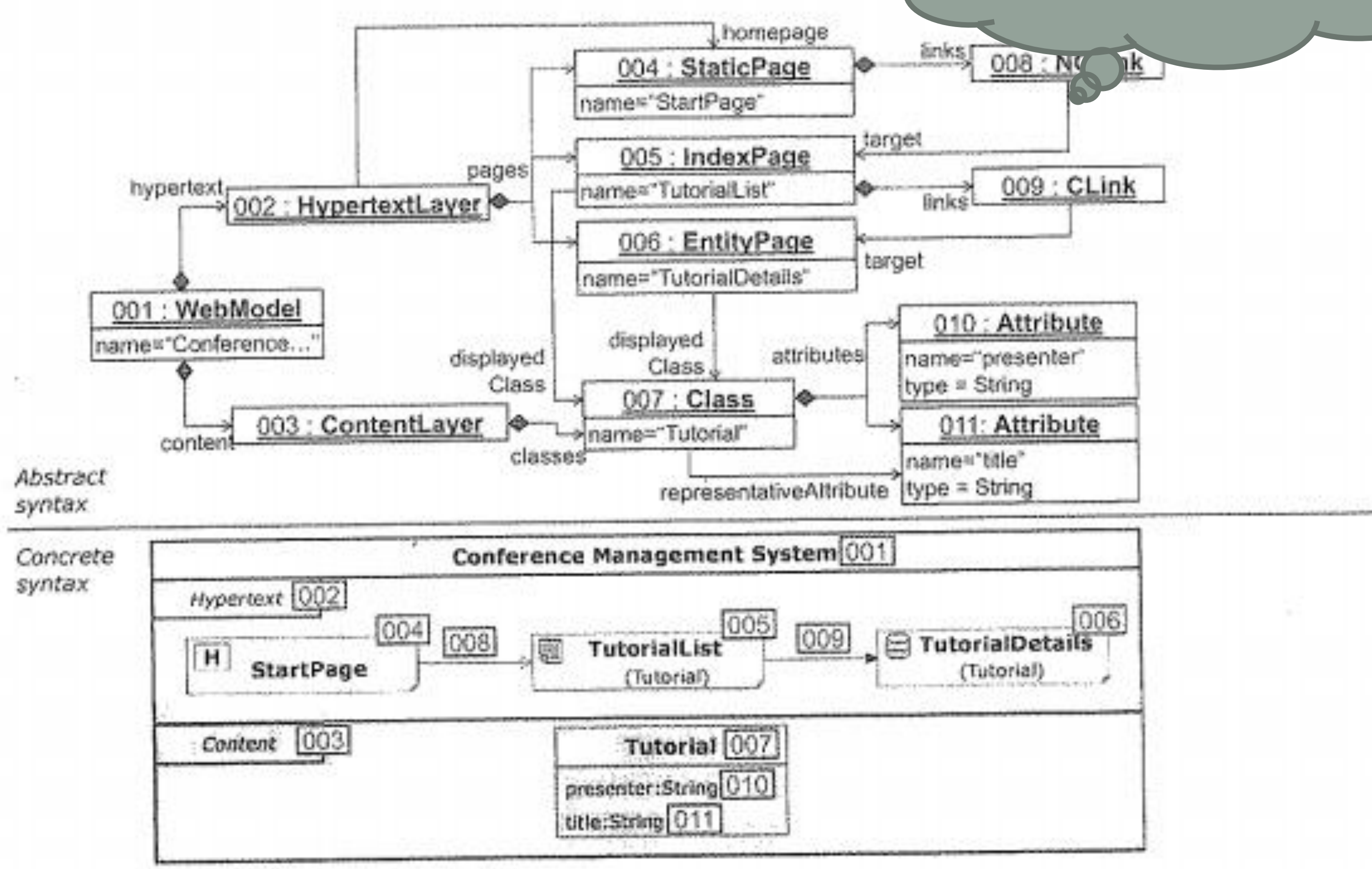
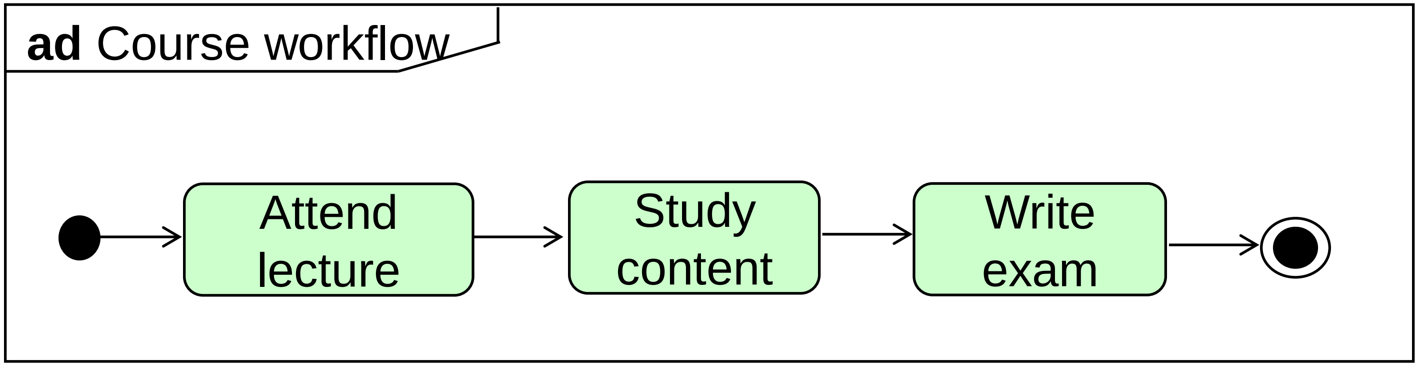


Figure 7.7: sWML model's abstract syntax.

Example 1/9

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



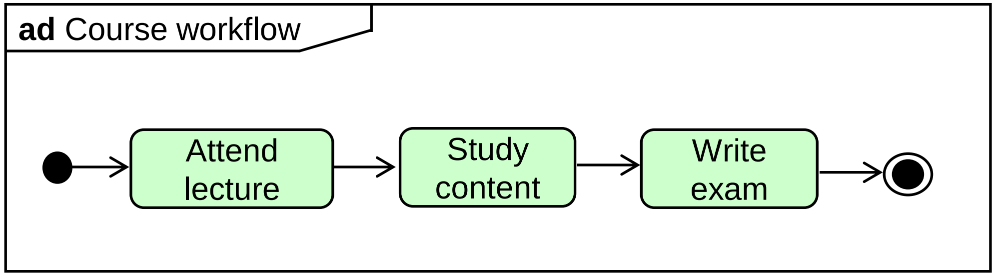
- Question: How does a possible metamodel to this language look like?

- Answer: apply metamodel development process!

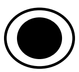
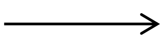
Example 2/9

Identification of the modeling concepts

Example model = Reference Model



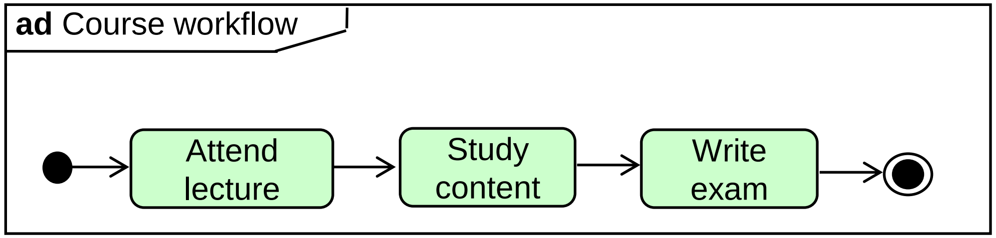
Notation table

Syntax	***Concept***
ad name	ActivityDiagram
	FinalNode
	InitialNode
name	Activity
	Transition

Example 3/9

Determining the properties of the modeling concepts

Example model

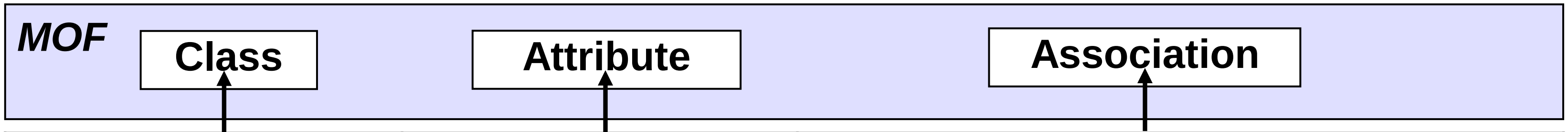


Modeling concept table

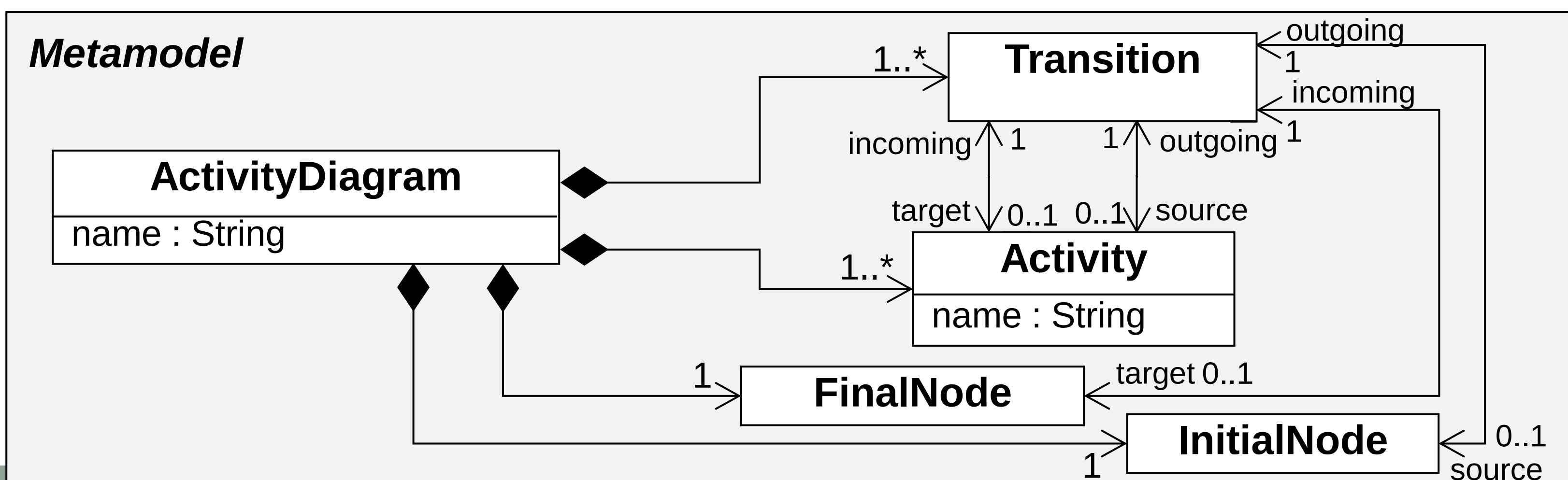
Concept	***Intrinsic properties***	***Extrinsic properties***
ActivityDiagram	Name	1 *InitialNode* 1 *FinalNode* Unlimited number of *Activities* and *Transitions*
FinalNode	-	Incoming *Transitions*
InitialNode	-	Outgoing *Transitions*
Activity	Name	Incoming and outgoing *Transitions*
Transition	-	Source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

Example 4/9

Object-oriented design of the language

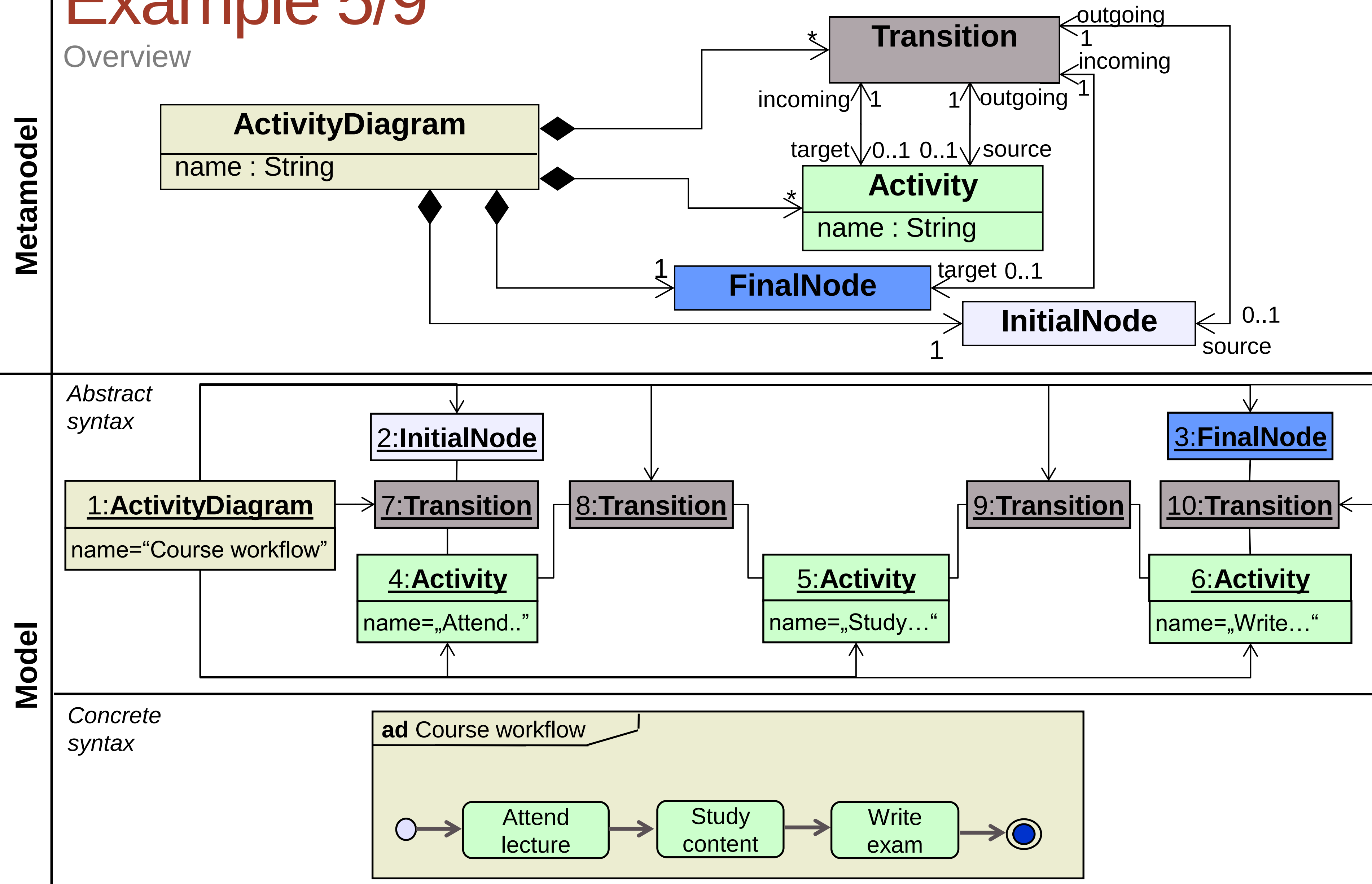


Concept	*Intrinsic properties*	*Extrinsic properties*
ActivityDiagram	Name	1 *InitialNode* 1 *FinalNode* Unlimited number of Activities and Transitions
FinalNode	-	Incoming *Transition*
InitialNode	-	Outgoing *Transition*
Activity	Name	Incoming and outgoing *Transition*
Transition	-	Source node and target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



Example 5/9

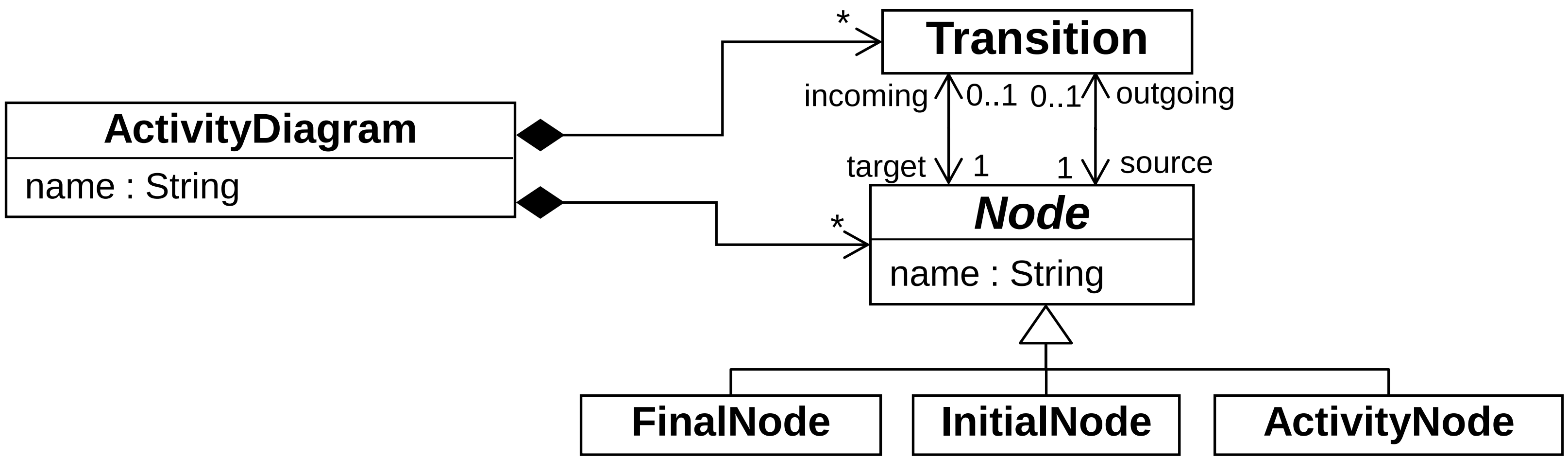
Overview



Example 6/9

Applying refactorings to metamodels

Metamodel



OCL Constraints

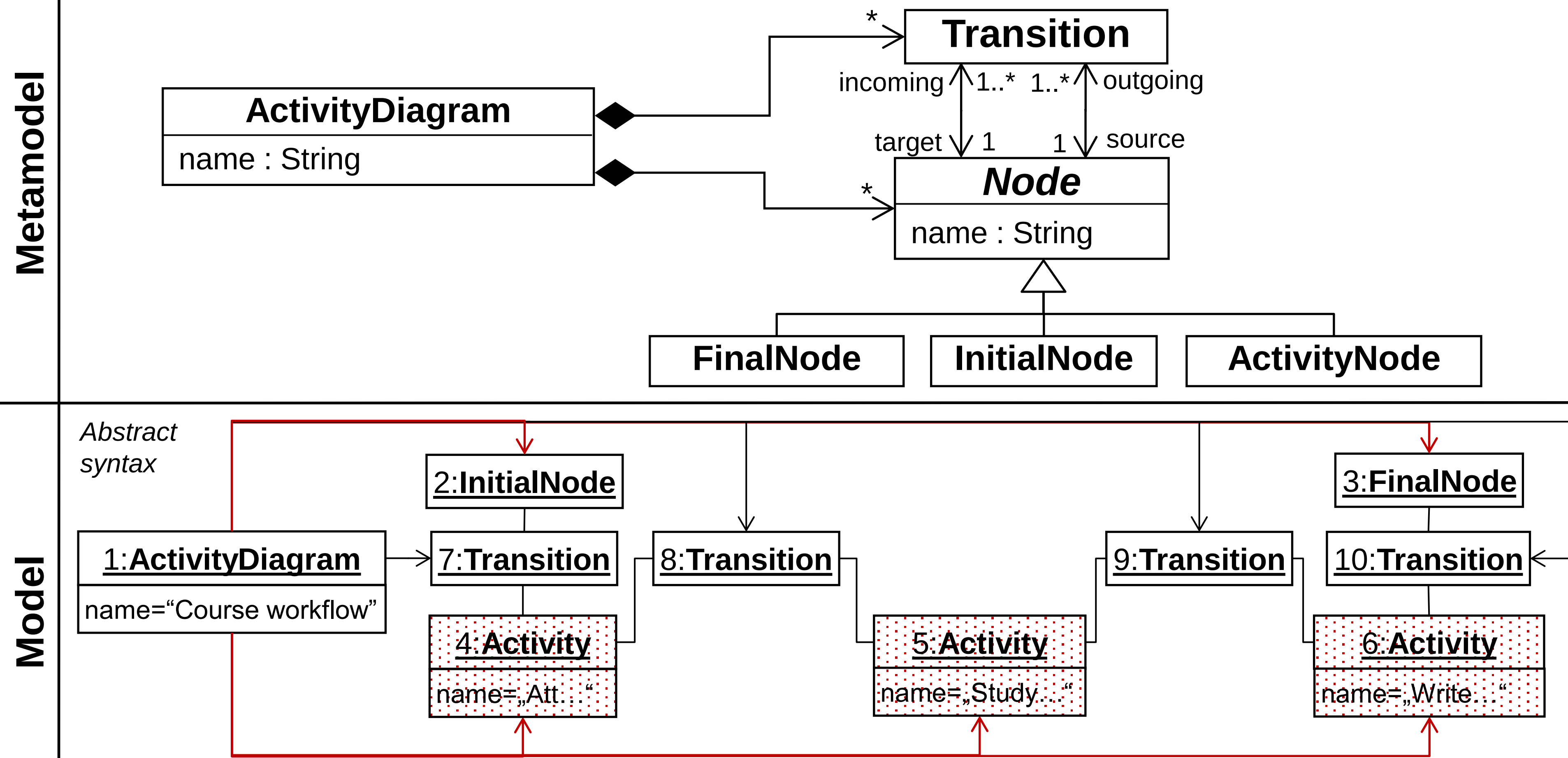
```
context ActivityDiagram
inv: self.transitions -> exists(t|t.isTypeOf(FinalNode))
inv: self.transitions -> exists(t|t.isTypeOf(InitialNode))

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



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
  - Models already exist and may become invalid

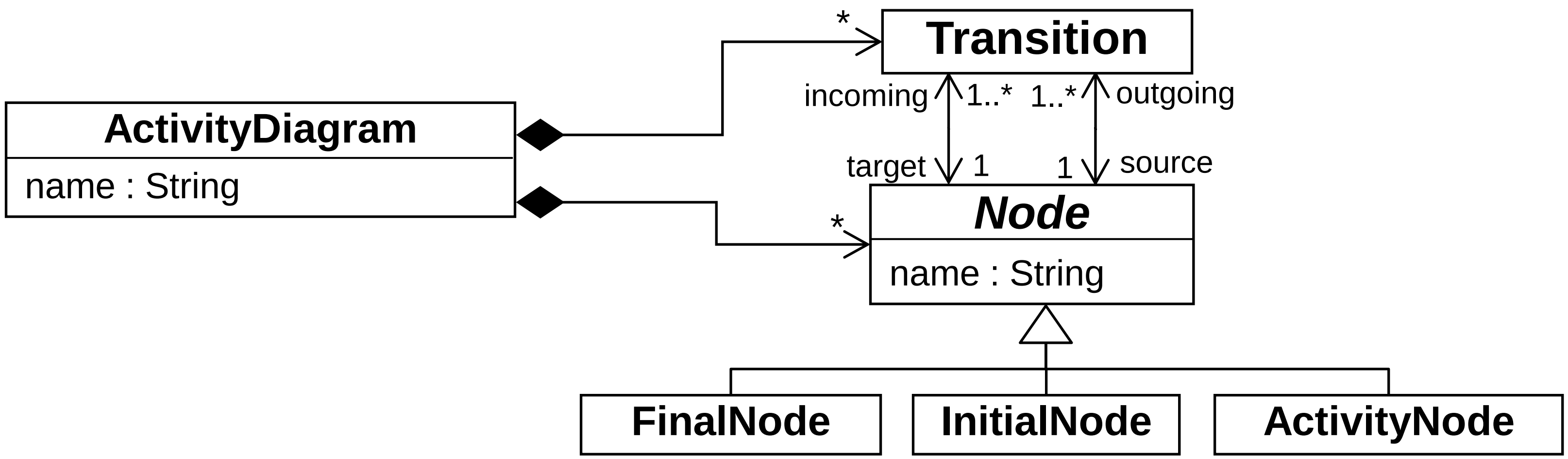
- **Changes** may **break** conformance relationships
  - Deletion and renamings of metamodel elements

- **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

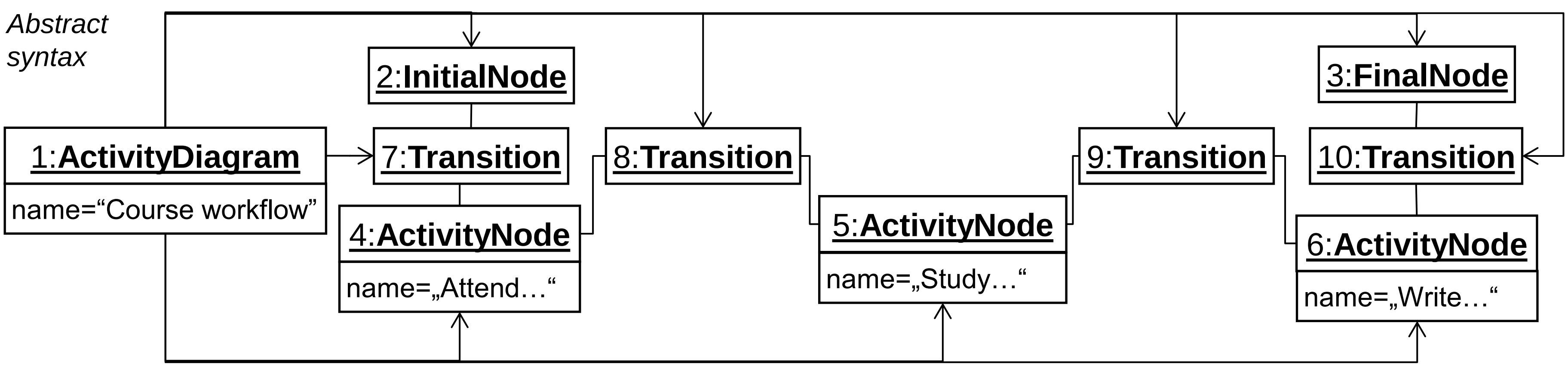
Example 9/9

Adapted model for new metamodel version

Metamodel



Model



More on this topic in Chapter 10!

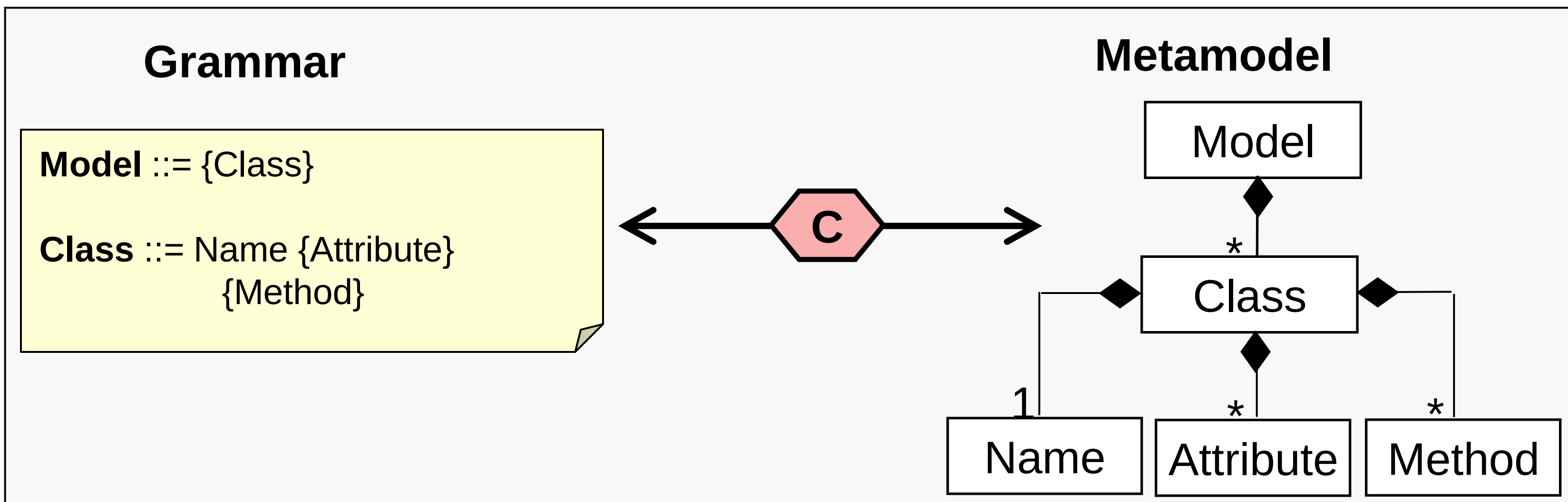
Excursus: Metamodeling – everything new? 1/2

Correspondence between EBNF and MOF

- **Mapping table** (excerpt)

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

- **Example**



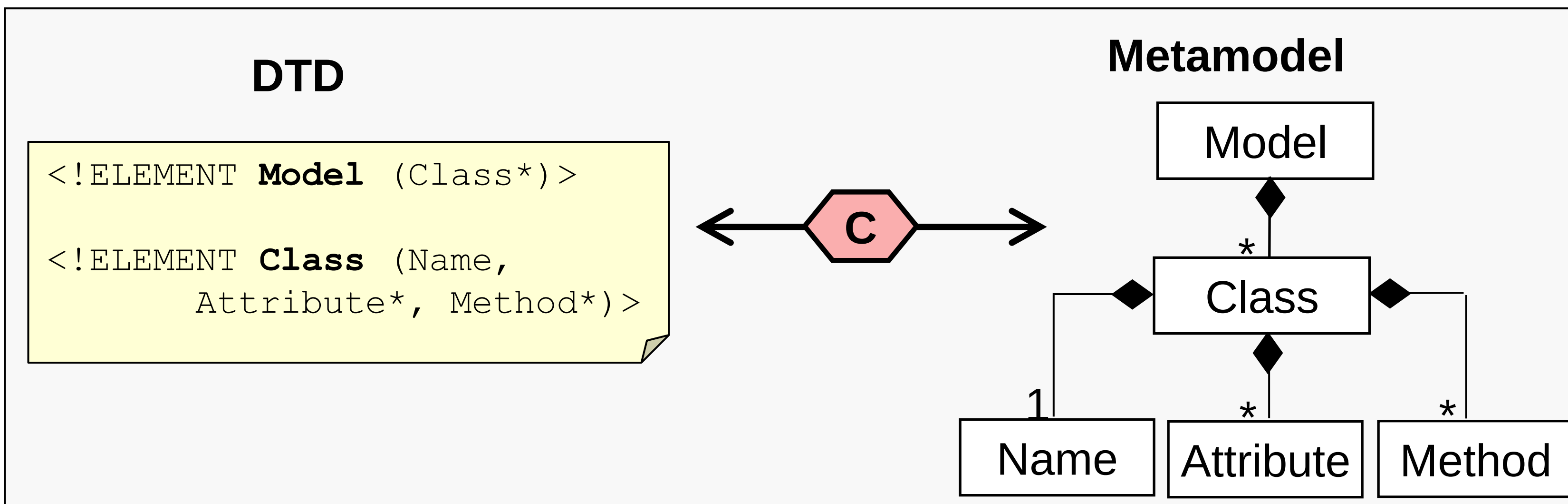
Excursus: Metamodeling – everything new? 2/2

Correspondence between DTD and MOF

- **Mapping table** (excerpt)

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

- **Example**



Ecore

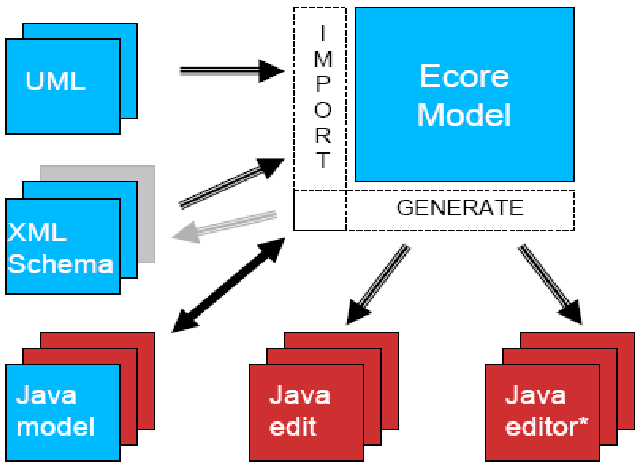
Introduction

- **Ecore** is the meta-metamodel of the Eclipse Modeling Frameworks (EMF)
  - www.eclipse.org/emf

- Ecore is a **Java**-based implementation of **eMOF**

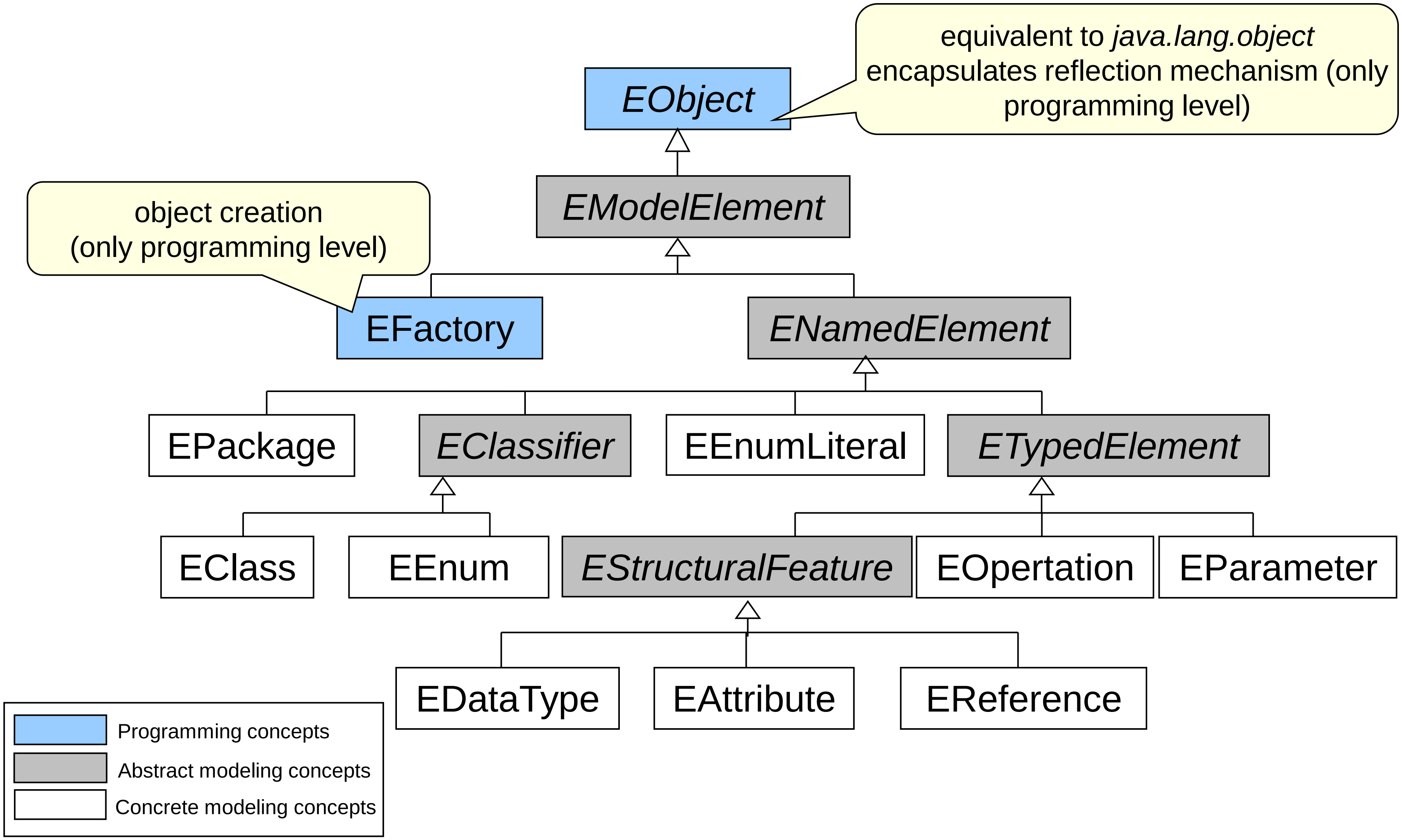
- **Aims** of Ecore
  - **Mapping eMOF** to **Java**

- **Aims** of EMF
  - Definition of modeling languages
  - Generation of model editors
  - UML/Java/XML integration framework



Ecore

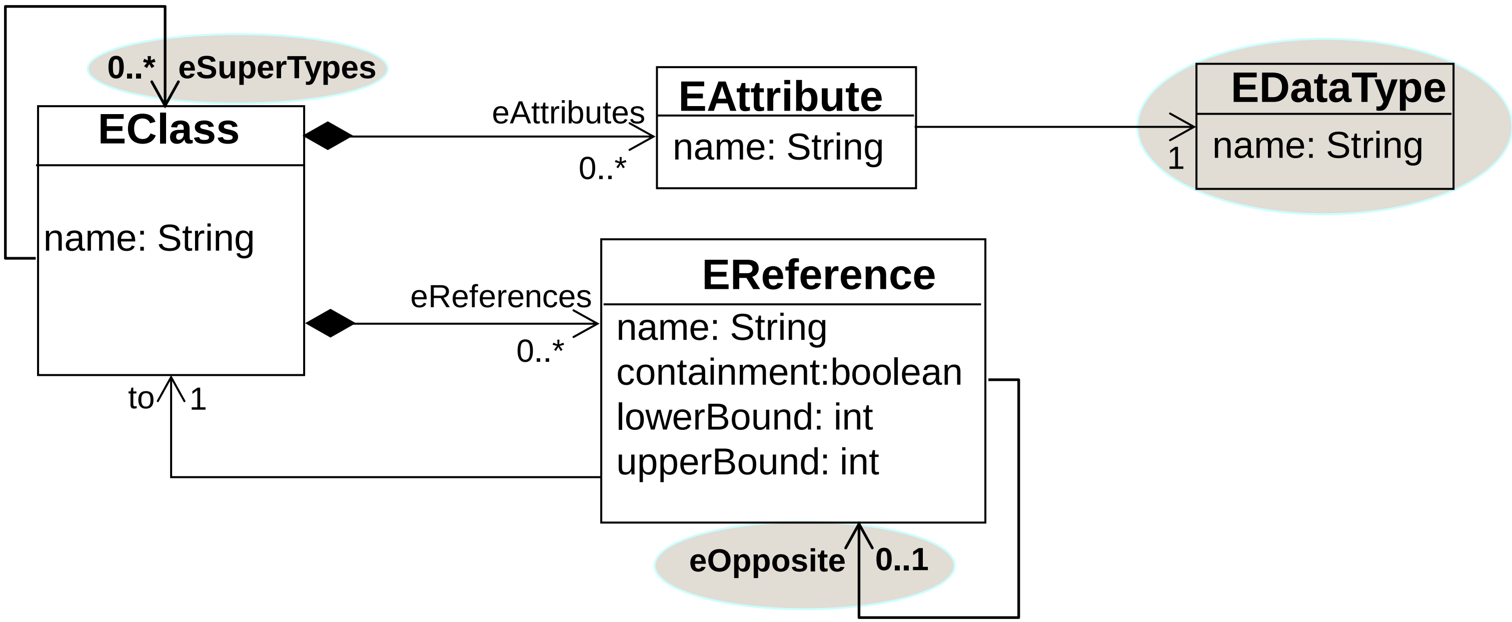
Taxonomy of the language concepts



Ecore

Core

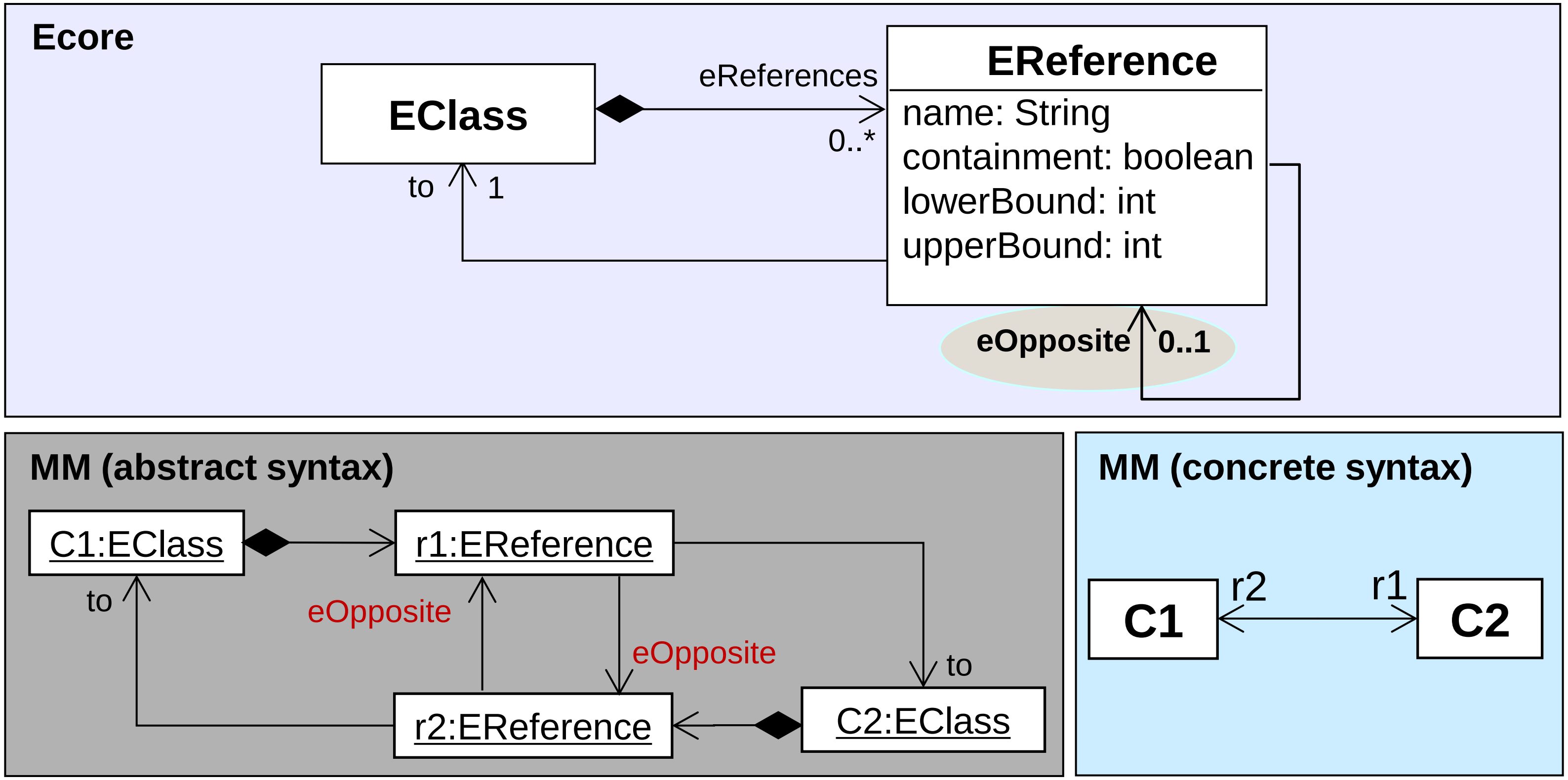
- Based on **object-orientation** (as eMOF)
  - Classes, references, attributes, inheritance, …
  - Binary *associations* are represented as **two references**
  - Data types are based on Java data types
  - Multiple inheritance is resolved by one „real“ inheritance and multiple implementation inheritance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



Ecore

Binary associations

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



Ecore

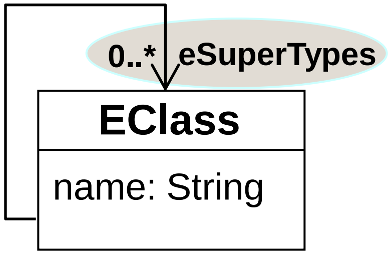
Data types

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

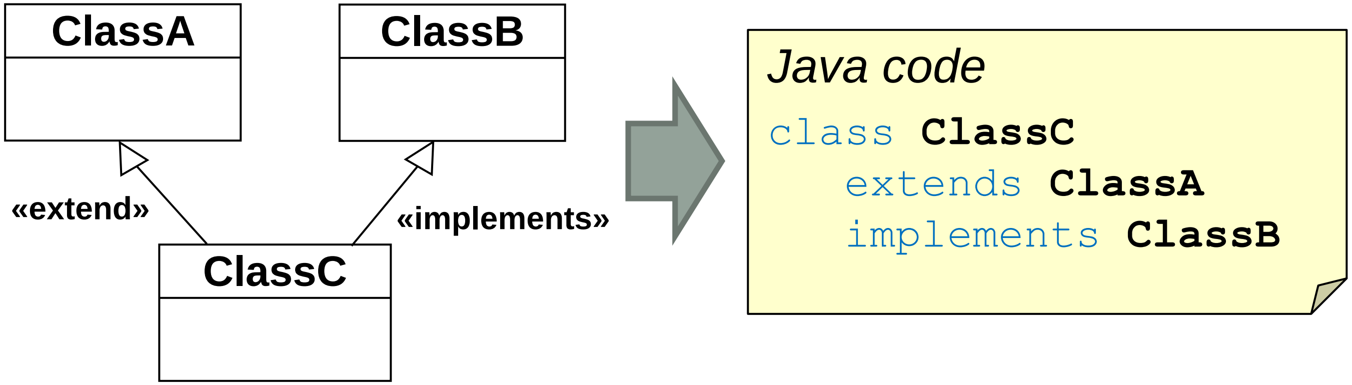
Ecore data type	Primitive type or class (Java)
EBoolean	boolean
EChar	char
EFloat	float
EString	java.lang.String
EBoolanObject	java.lang.Boolean
...	...

Ecore

Multiple inheritance



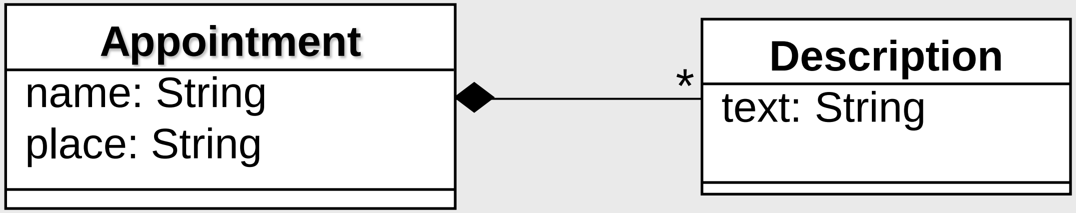
- Ecore supports **multiple inheritance**
  - Unlimited number of *eSuperTypes*
- Java supports **only single inheritance**
  - Multiple inheritance simulated by implementation of interfaces!
- Solution for Ecore2Java mapping
  - First inheritance relationship is used as „real" inheritance relationship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



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>
```


Eclipse Modeling Framework

What is EMF?

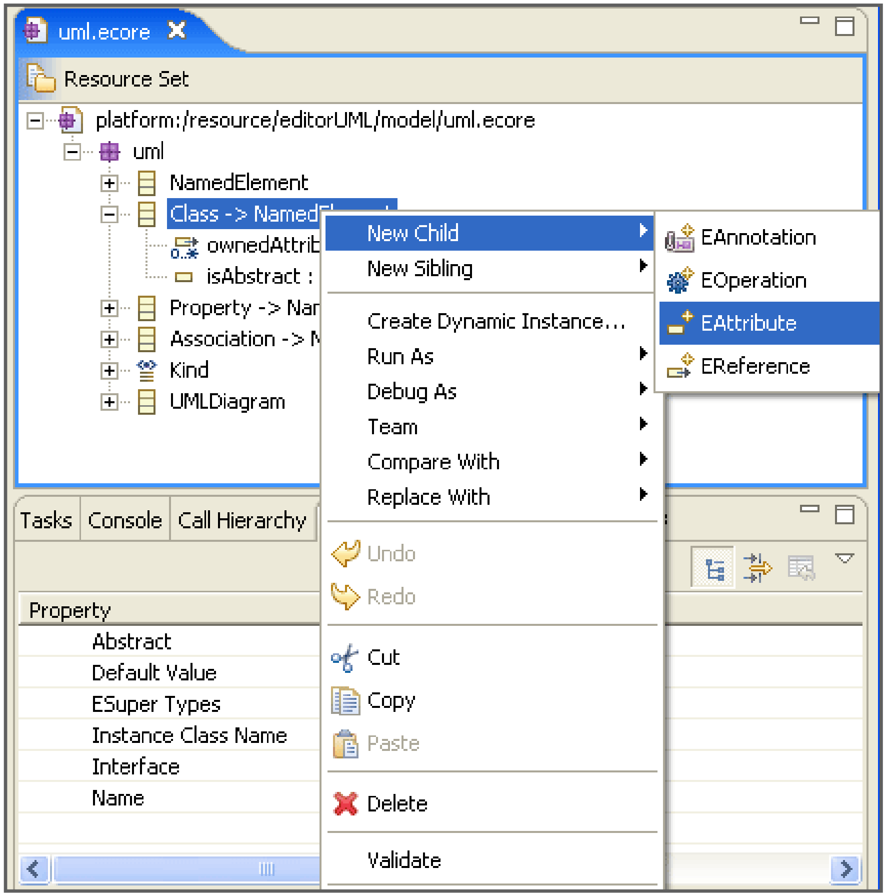
- **Pragmatic approach** to combine **modeling** and **programming**
  - **Straight-forward mapping rules** between Ecore and Java

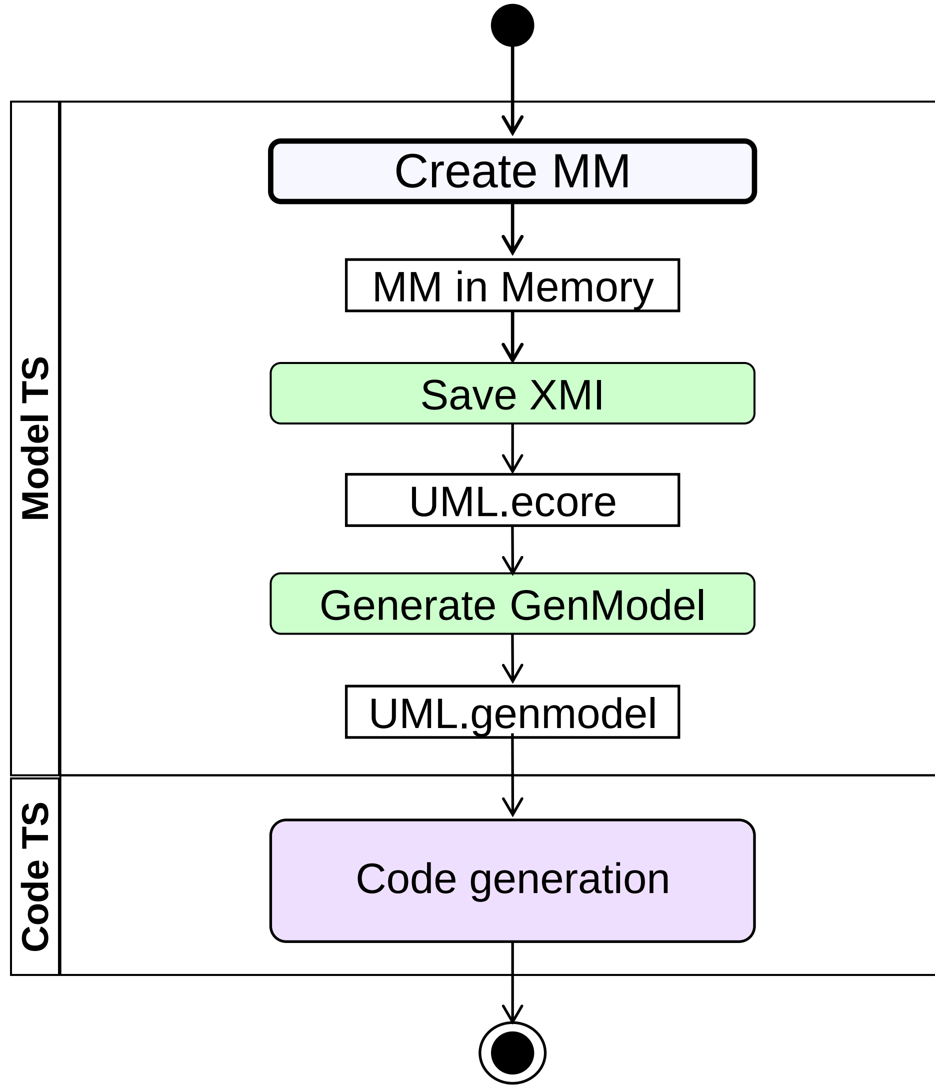
- EMF **facilitates automatic generation** of **different implementations** out of Ecore models
  - Java code, XML documents, XML Schemata

- **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)
  - …

Model editor generation process

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



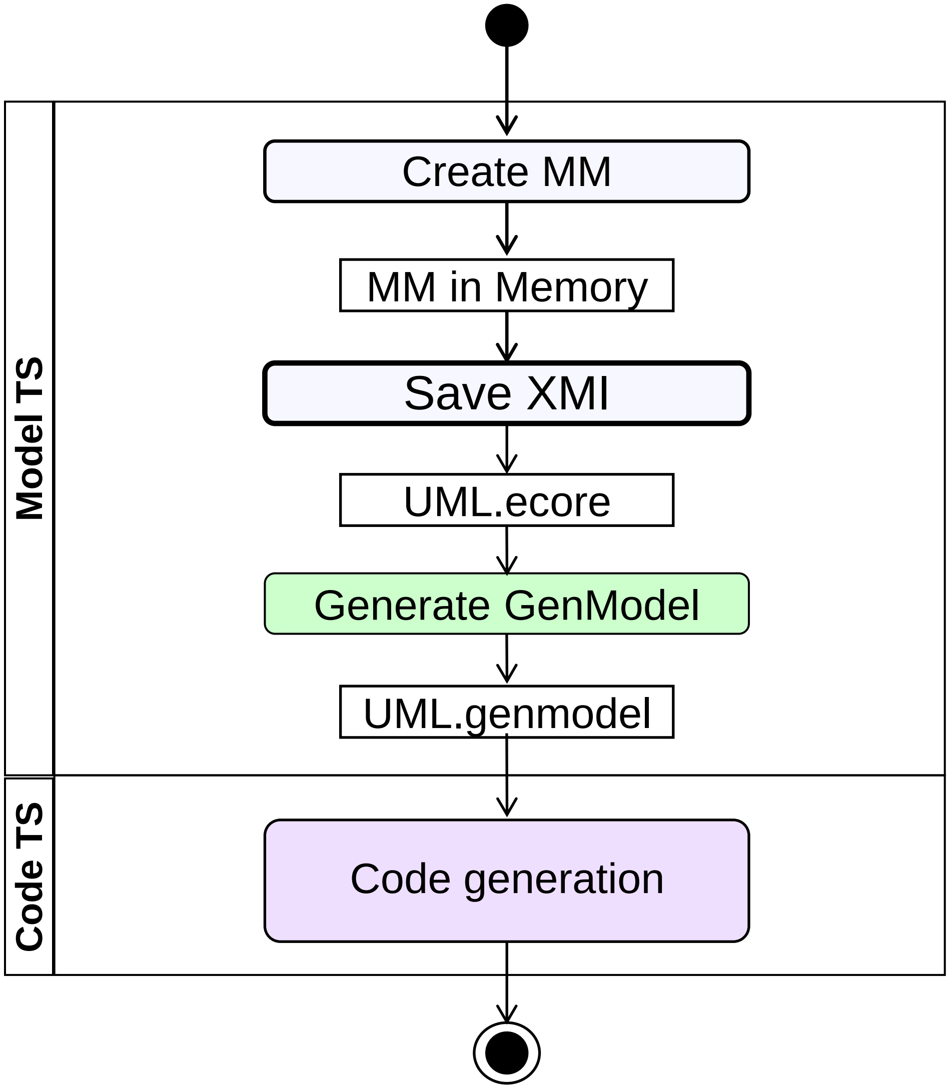


Model editor generation process

Step 2 – Save metamodel

UML.ecore

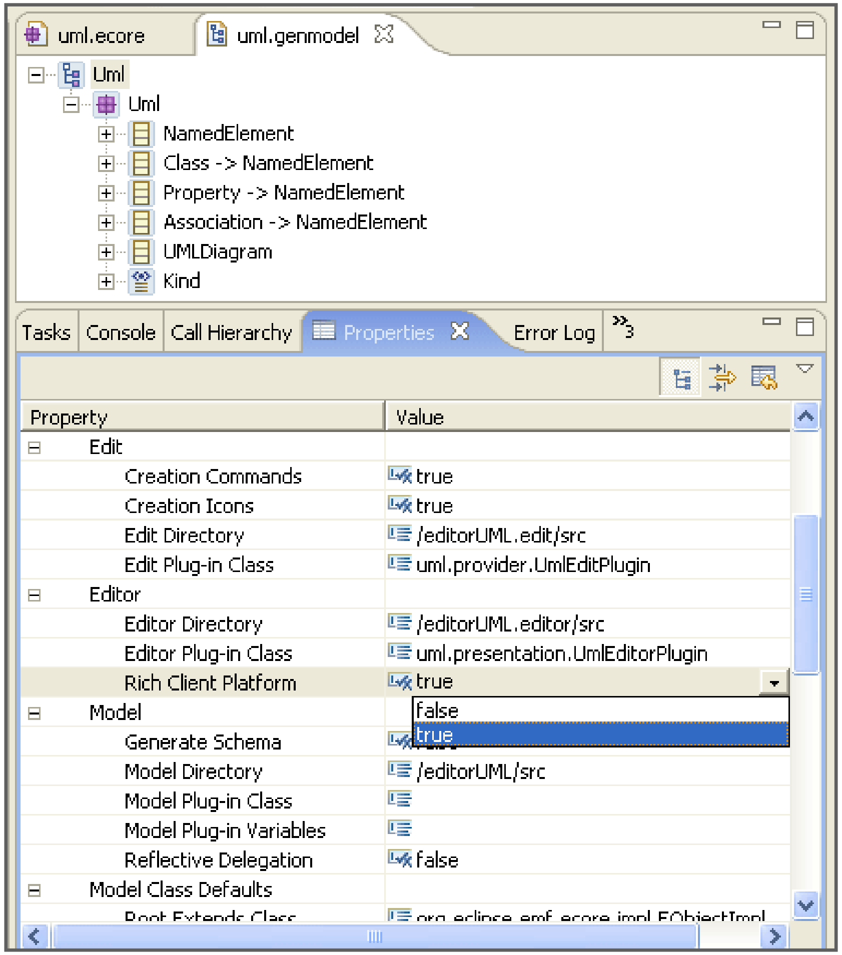
```
<?xml version="1.0" encoding="UTF-8"?>
<ecore:EPackage xmi:version="2.0"
   xmlns:xmi="http://www.omg.org/XMI"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-
           instance"
   xmlns:ecore="http://www.eclipse.org/emf/2002/
           Ecore"
   name="uml"
   nsURI="http://uml" nsPrefix="uml">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

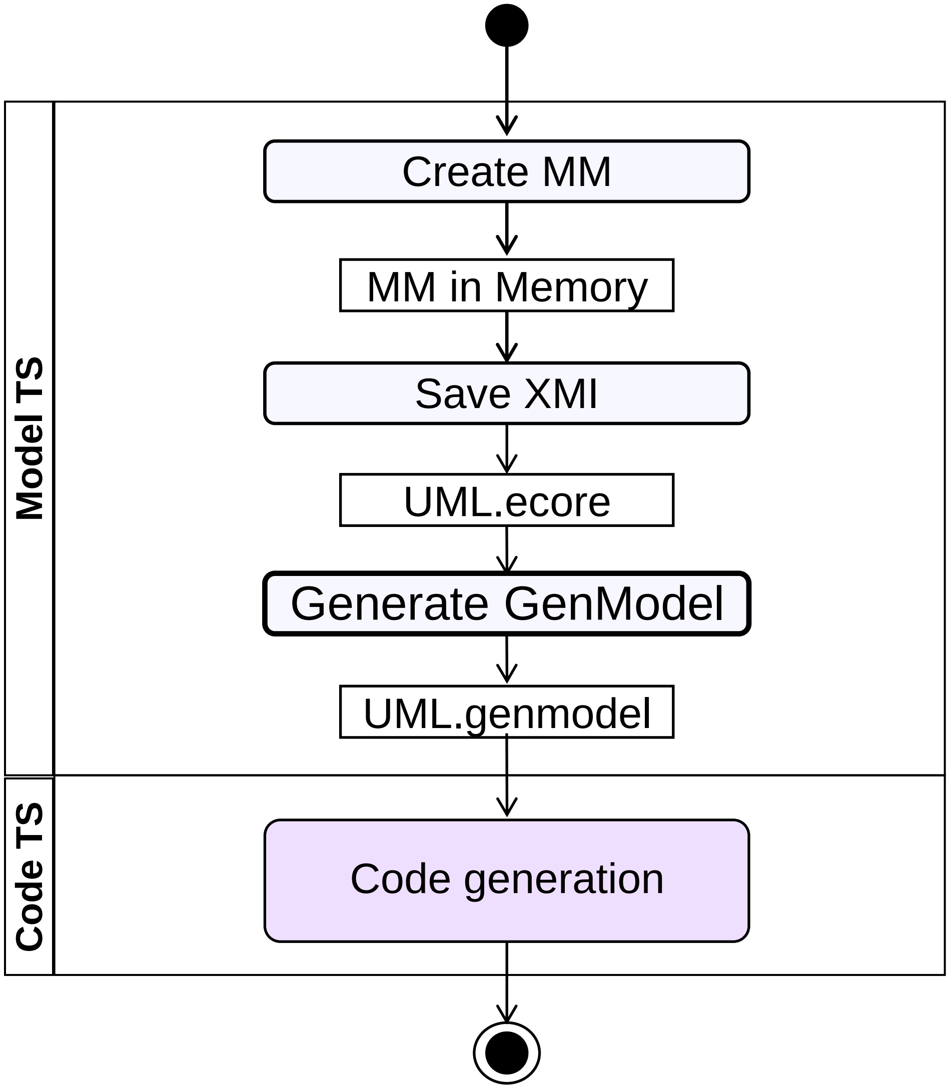


Model editor generation process

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:

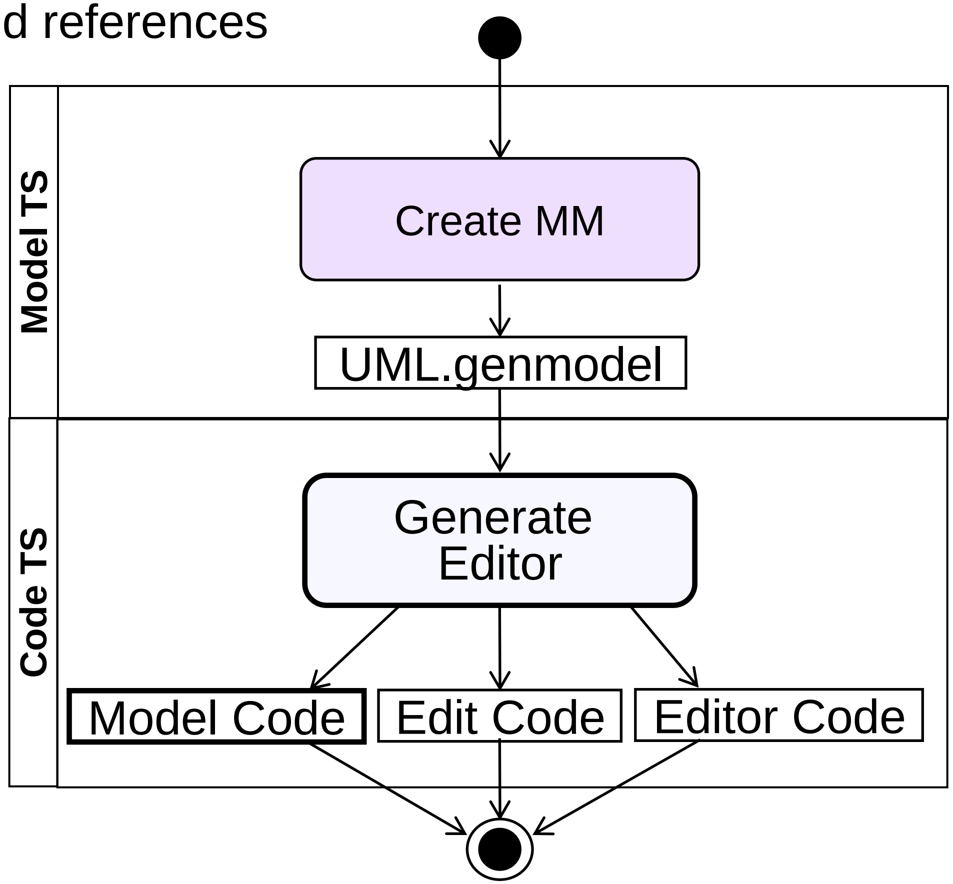
- **Interface:** Getter/setter for attributes and references

```
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;
   }
}
```

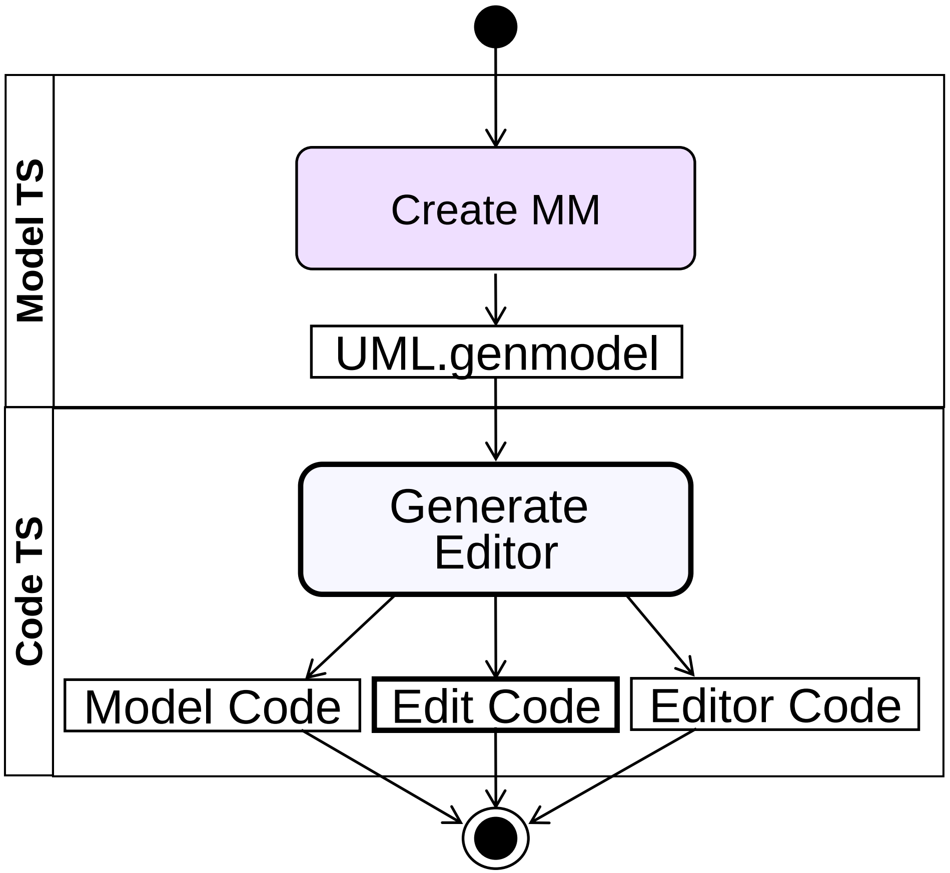
- **Factory** for the creation of model elements,
  for each Package one *Factory-Class* is created



Model editor generation process

Step 5 – Generate edit code

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

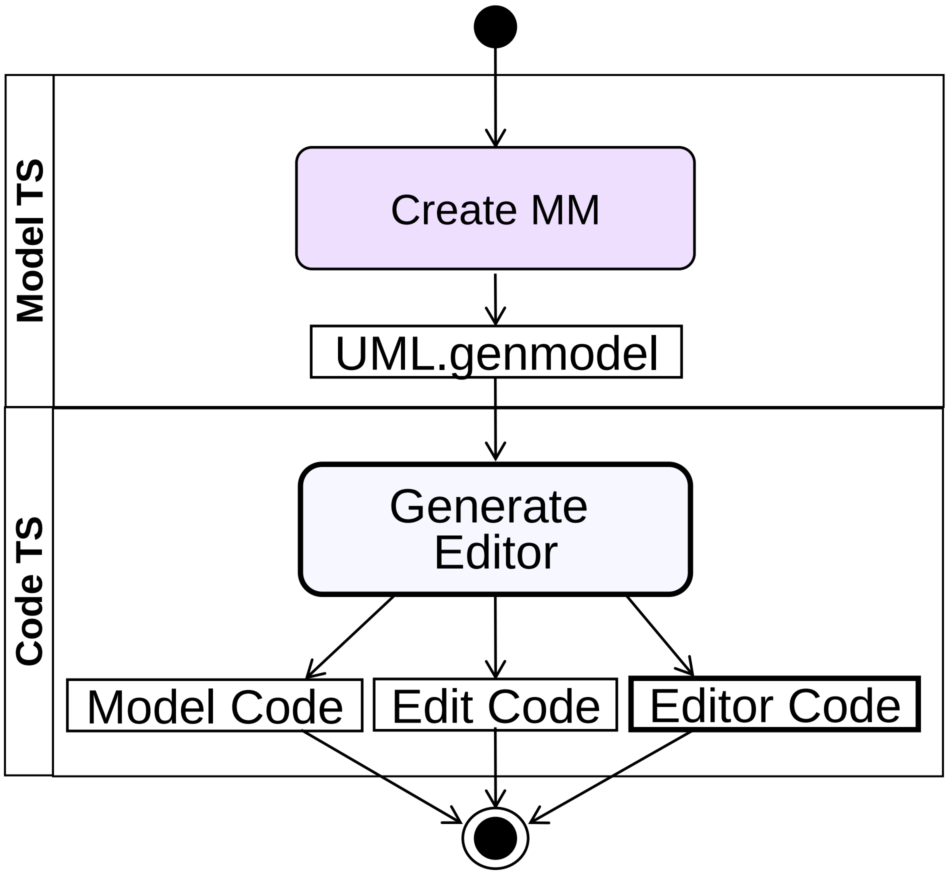


Model editor generation process

Step 6 – Generate editor code

- Editor as **Eclipse Plugin** or **RCP Application**

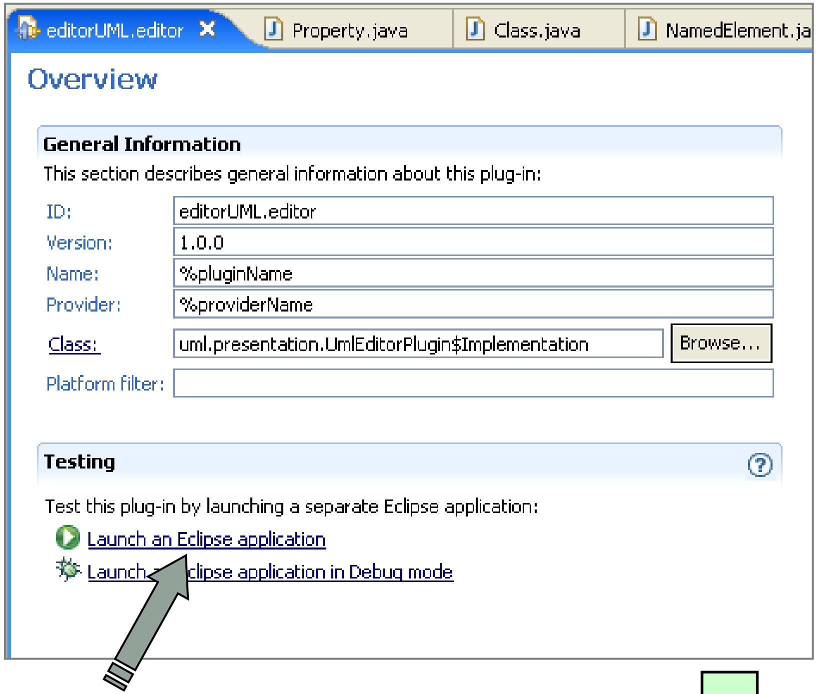
- Generated artifacts
  - *Model creation wizard*
  - *Editor*
  - *Action bar contributor*
  - *Advisor (RCP)*
  - *plugin.xml*
  - *plugin.properties*



Model editor generation process

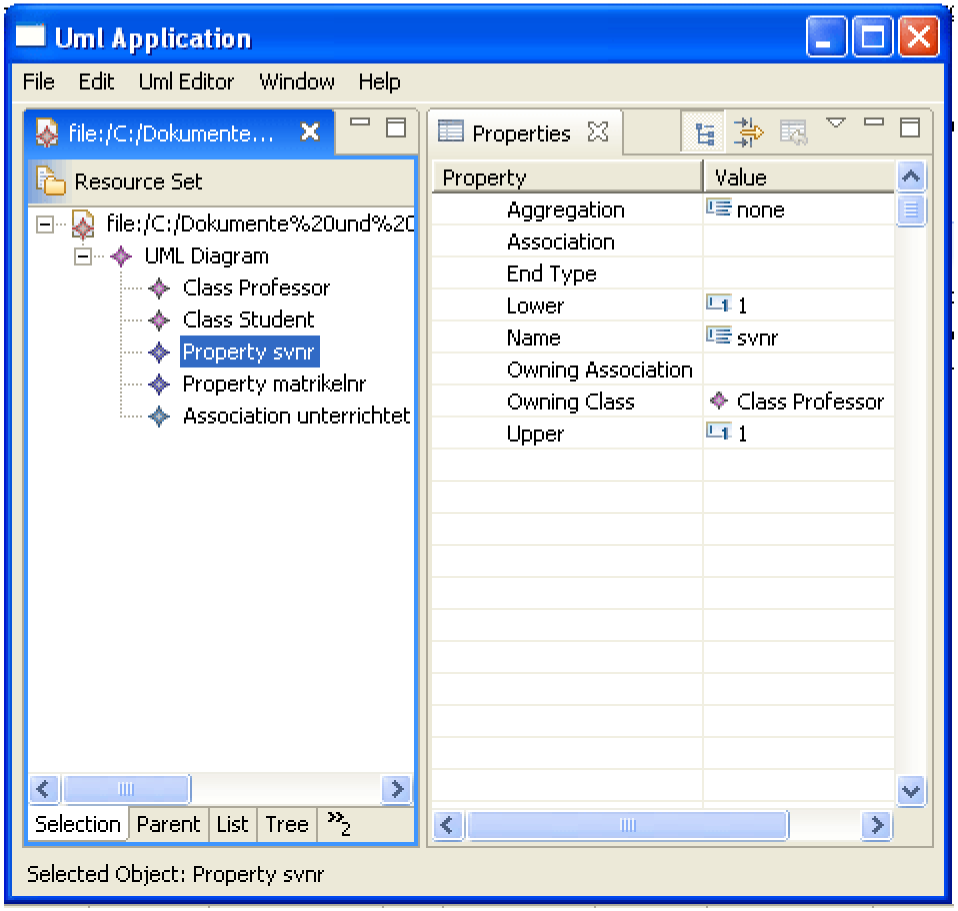
Start the modeling editor

Plugin.xml



Click here to start!

RCP Applikation



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

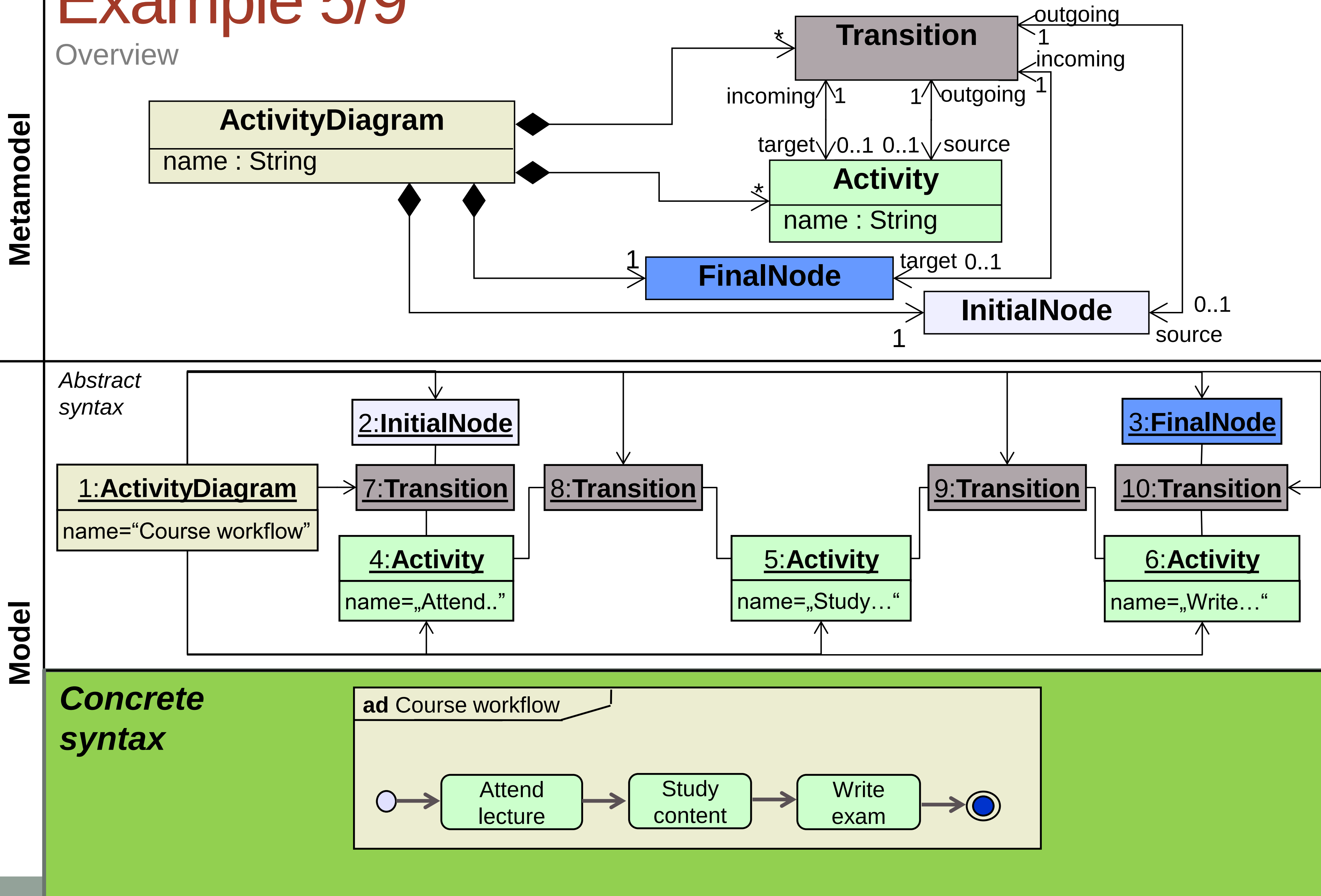
Content

- Part A
  - Introduction
  - Abstract Syntax
- **Part B**
  - Concrete Syntaxes
    - Graphical Concrete Syntax
    - Textual Concrete Syntax
- Summary

CONCRETE SYNTAX DEVELOPMENT

Example 5/9

Overview



Concrete syntax development

Overview

- Metamodels only define the abstract syntax, but not the concrete notation of the language.
  - i.e., graphical or textual elements used to render the model elements in modeling editors.
- Concrete syntax **improves** the **readability** of models
  - Abstract syntax not intended for humans!
- **One** abstract syntax may have **multiple** concrete syntaxes
  - Including textual and/or graphical
  - Mixing textual and graphical notations still a challenge!

Concrete syntax development

Overview

- For the concrete syntax definition there is currently **just one** OMG standard available → Diagram Definition (DD) specification.
- DD allows to define *graphical concrete syntaxes.*
- Concrete syntax in UML metamodel is only shown in so-called notation tables and by giving some examples.
- A more formal definition of the UML concrete syntax **is not given.**

Concrete syntax development

Benefits

- Having the concrete syntax formally defined allows the use of sophisticated techniques such as **automatic generation of editors** *to manipulate the artifacts in their concrete syntax.*
- There are several emerging frameworks which provide specific languages to describe the concrete syntax of a modeling language formally.
- These framework also allows the generation of editors for visualizing and manipulating models in their concrete syntax.

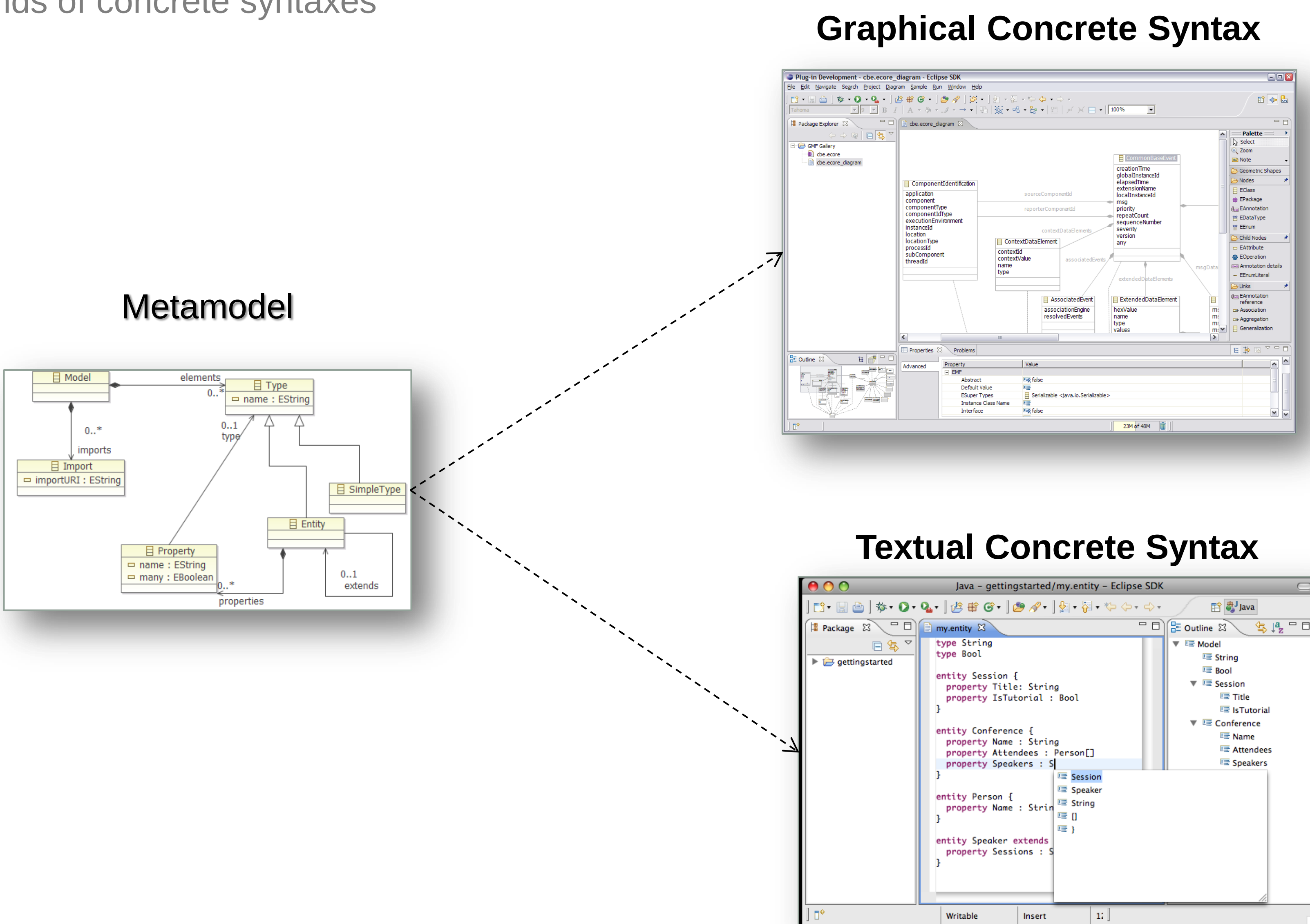
Concrete syntax development

Kinds of concrete syntaxes

GCS	TCS
Graphical Concrete Syntaxes	**Textual Concrete Syntaxes**
Encode information using spatial arrangements of graphical (and textual) elements.	Enconding information using sequences of characters in most programming languages.
Are two-dimensional representations.	One-dimensional representations.

Concrete syntax development

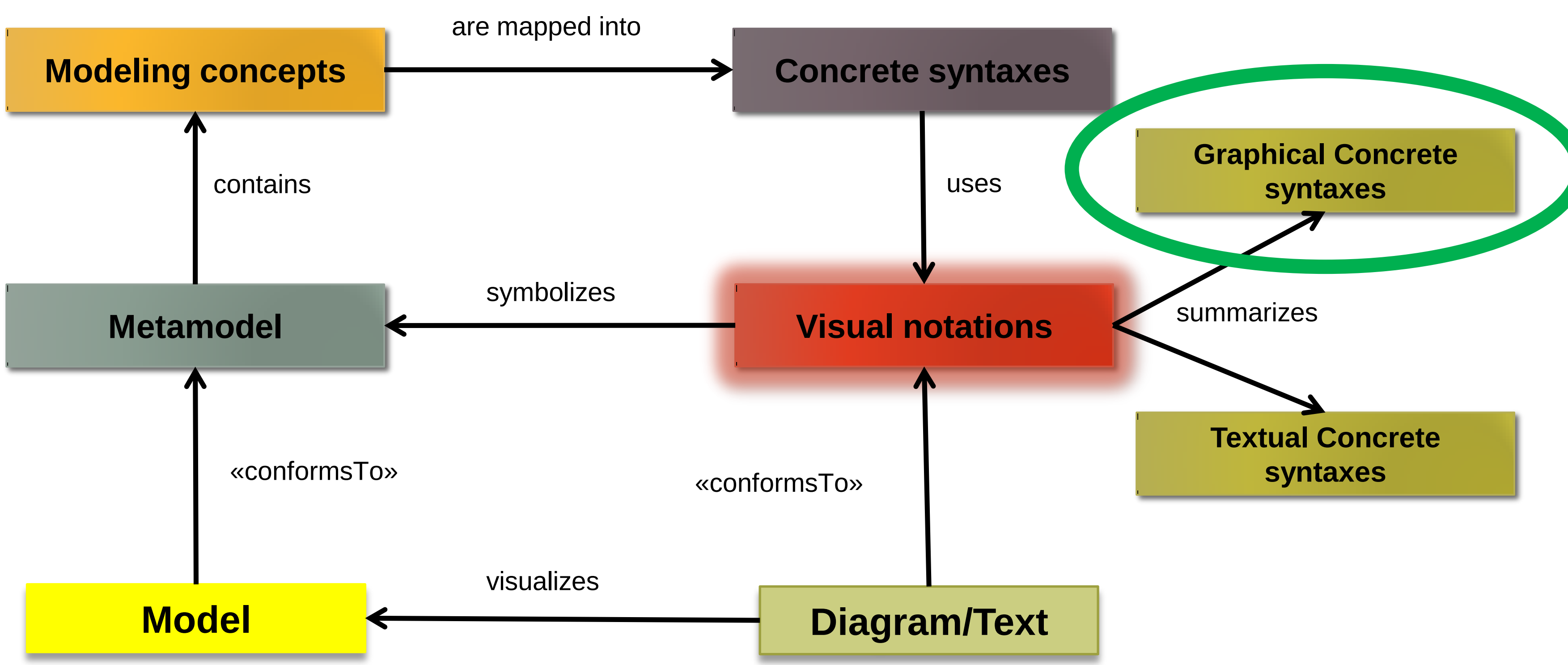
Kinds of concrete syntaxes



Concrete syntax development

Visual notations

- The **visual notation** of a model language is referred as **concrete syntax**
- **Visual notation** introduces symbols for modeling concepts.



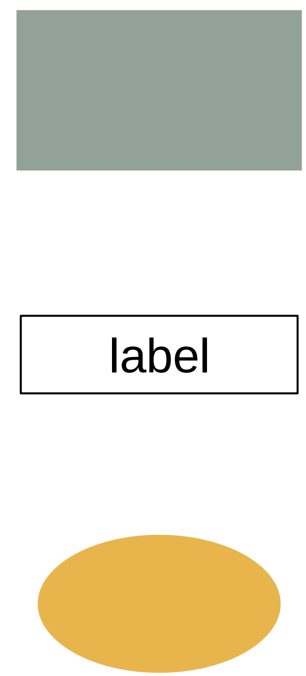
GRAPHICAL CONCRETE SYNTAX

Graphical concrete syntax

Anatomy of Graphical concrete syntax

A Graphical Concrete Syntax (GCS) consists of:

- **graphical symbols,**
  - e.g., rectangles, circles, …

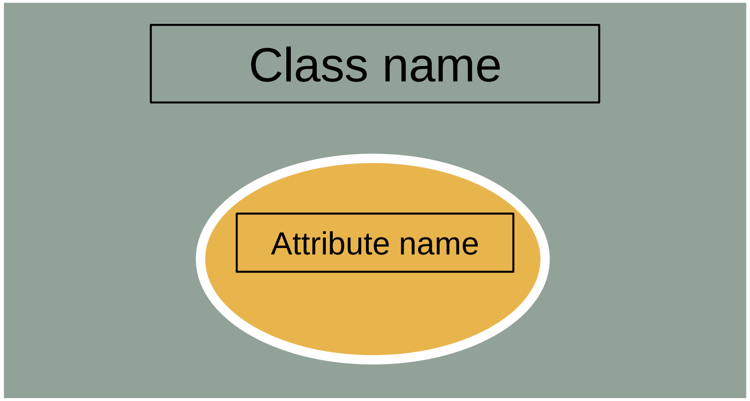


Graphical concrete syntax

Anatomy of Graphical concrete syntax

A Graphical Concrete Syntax (GCS) consists of:

- **graphical symbols,**
  - e.g., rectangles, circles, …
- **compositional rules**,
  - e.g., nesting of elements, …

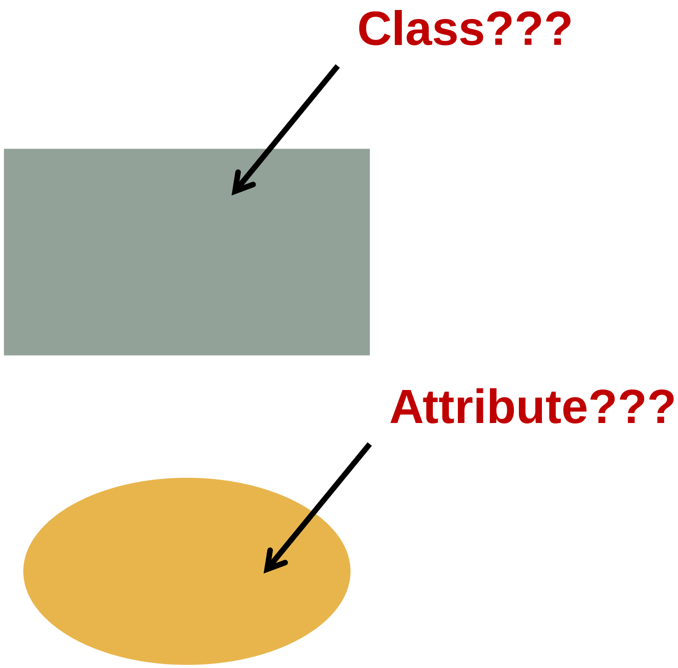


Graphical concrete syntax

Anatomy of Graphical concrete syntax

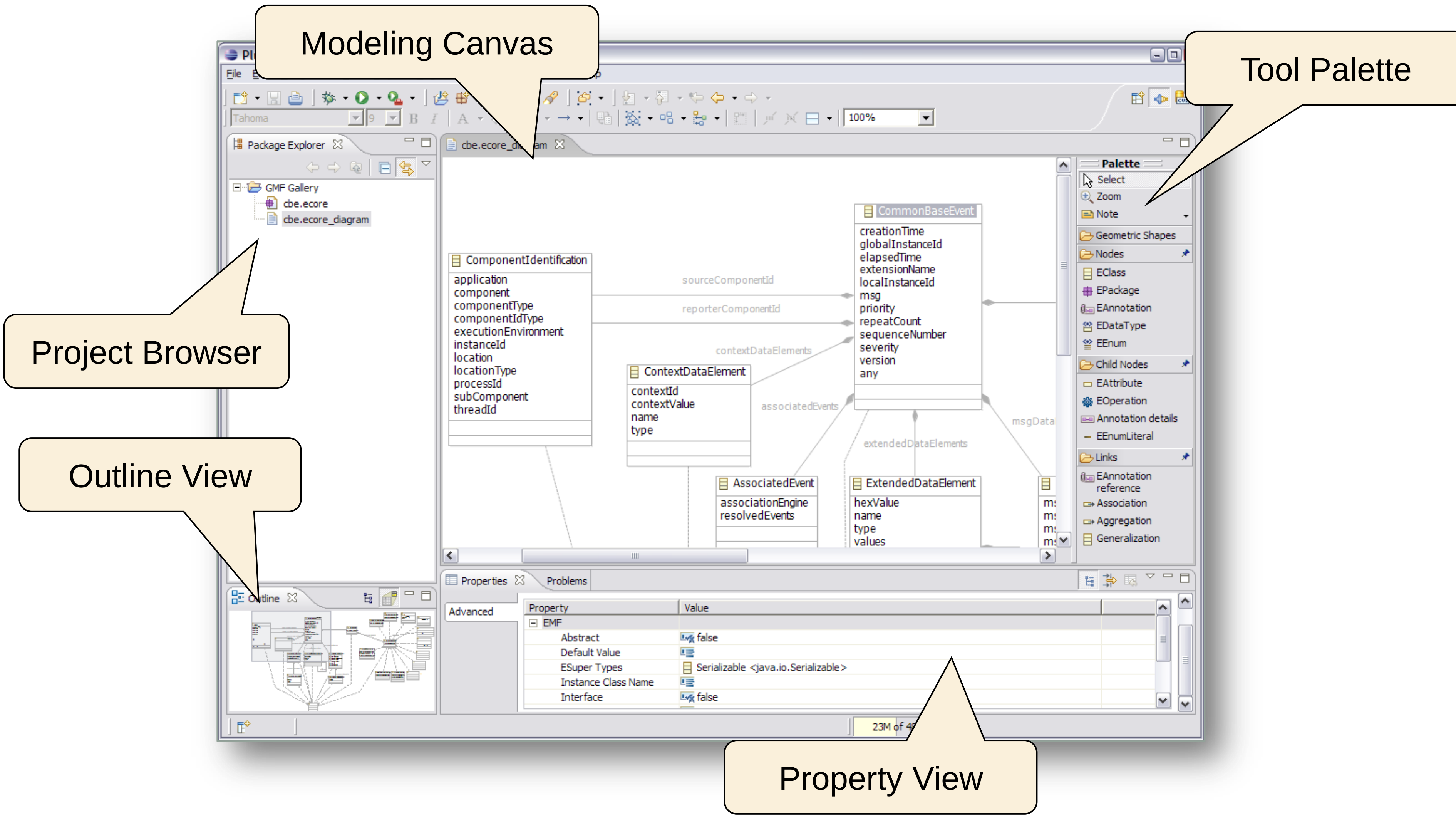
A Graphical Concrete Syntax (GCS) consists of:

- **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



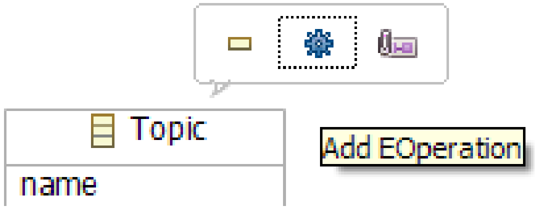
Graphical concrete syntax

Anatomy of Graphical concrete syntax

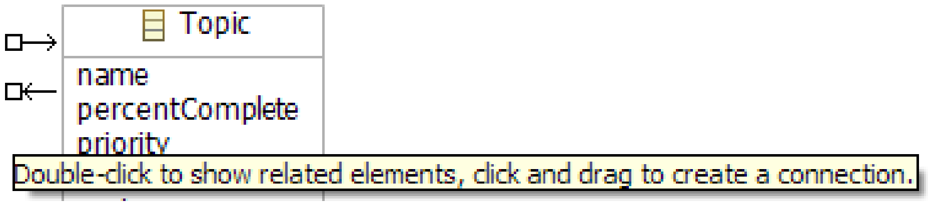


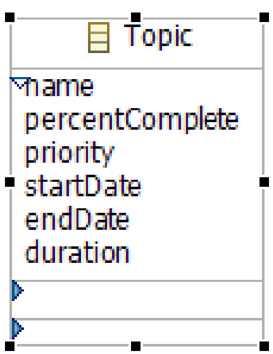
Graphical concrete syntax

Features of Graphical Modeling Editors

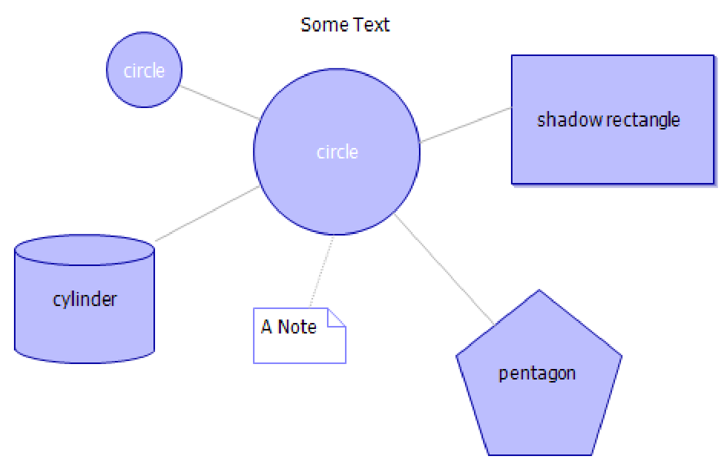


Connection Handles:





Geometrical Shapes:

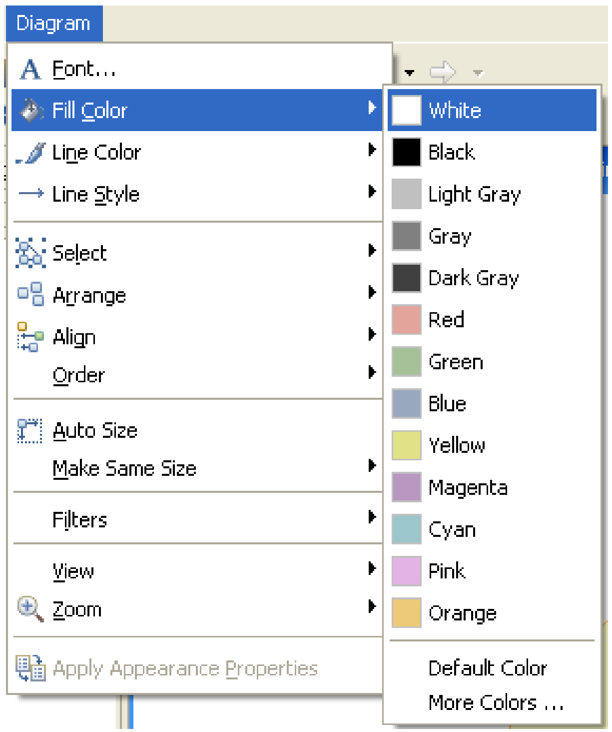


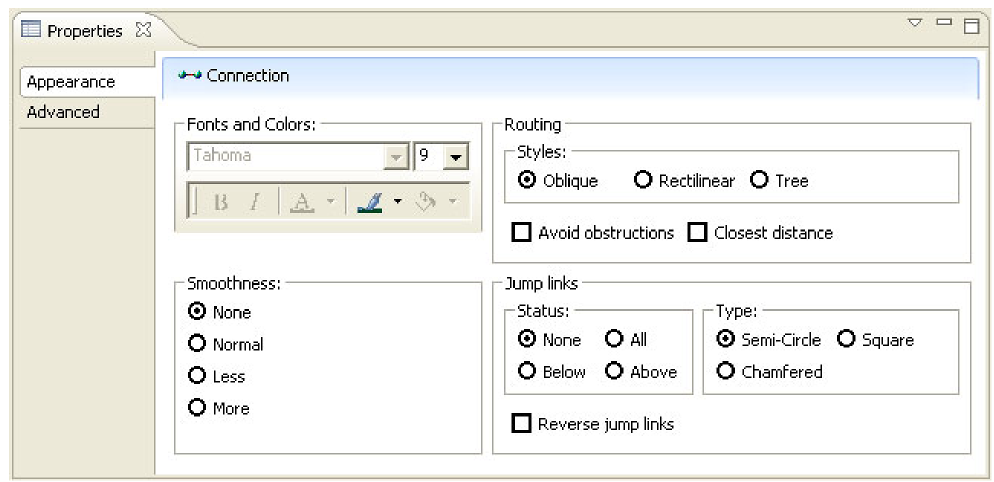
Graphical concrete syntax

Features of Graphical Modeling Editors

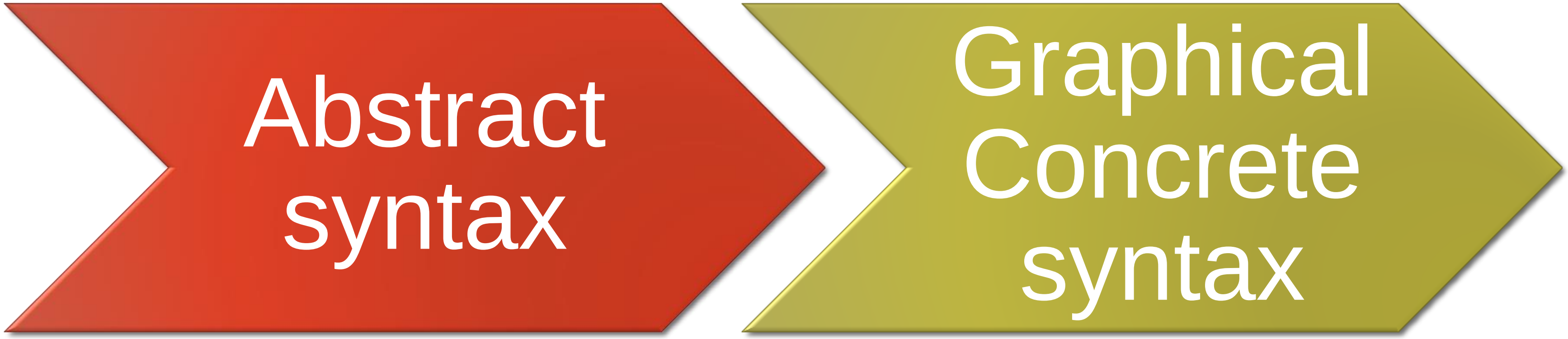
Toolbar:

Actions:





Graphical concrete syntax



Graphical concrete syntax

Approaches to GCS development

- Mapping-center GCS
- Annotation-center GCS
- API-center GCS

Graphical concrete syntax

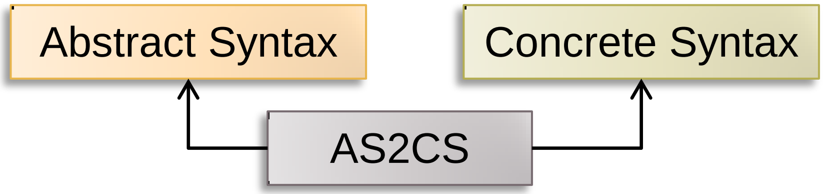
Approaches to GCS development

Mapping-center GCS

Annotation-center GCS

API-center GCS

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



This approach is followed by the

Graphical Modeling Framework (GMF)

Graphical concrete syntax

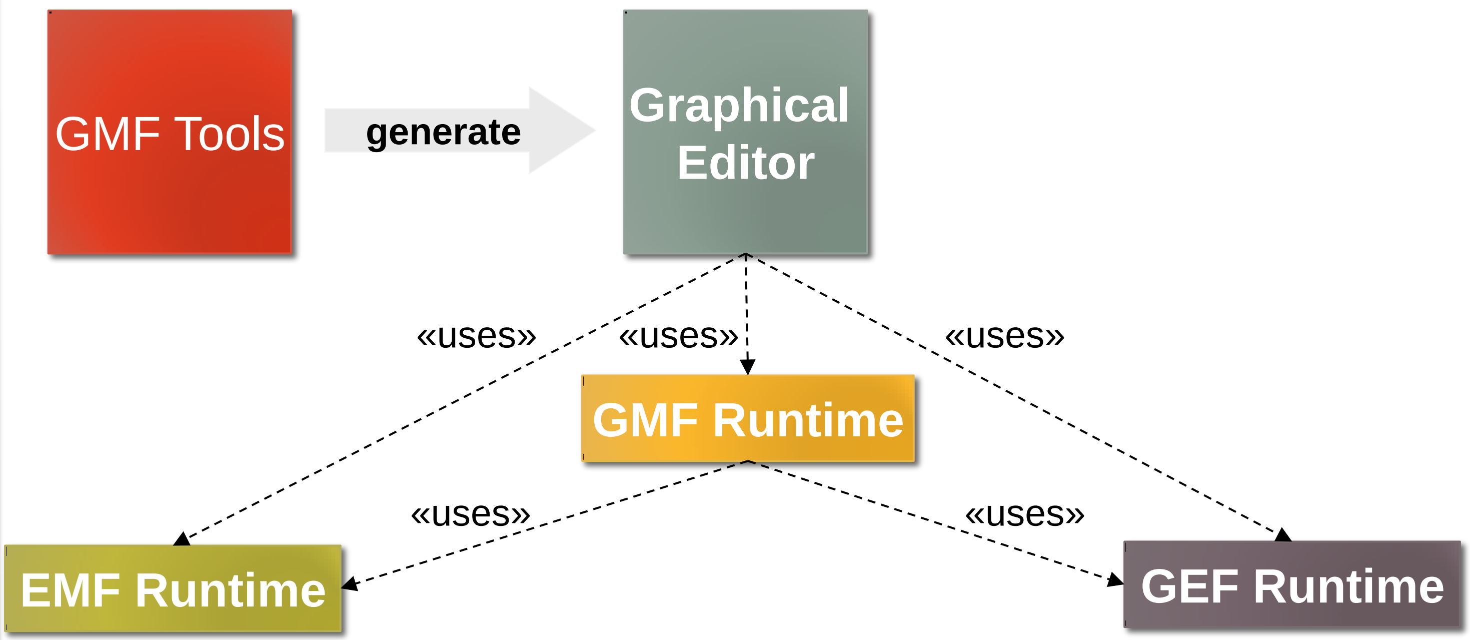
Approaches to GCS development

Mapping-center GCS

Annotation-center GCS

API-center GCS

*"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

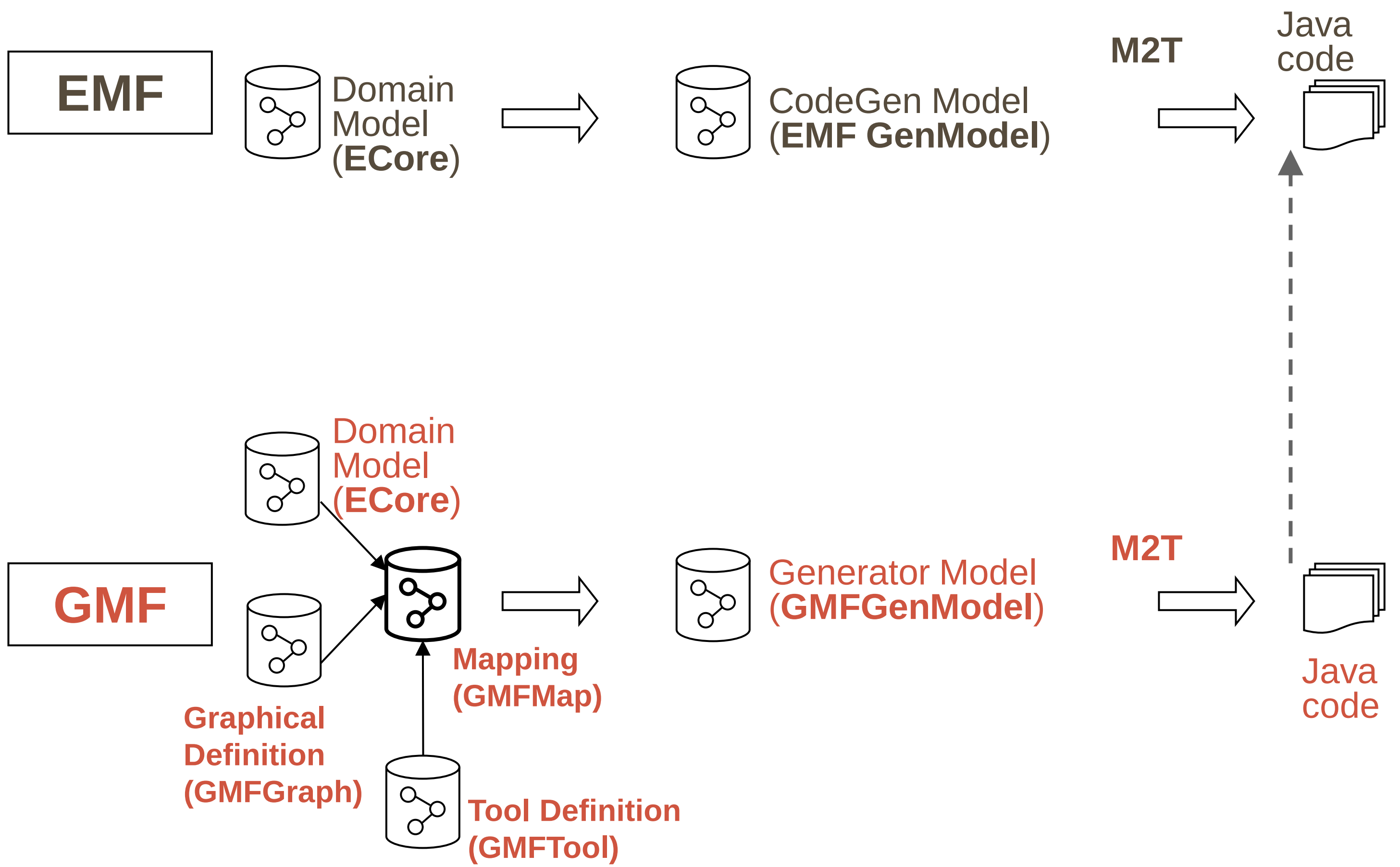


GMF is a DSML (Domain Specific Modeling Language)

Graphical concrete syntax

Approaches to GCS development

- **Mapping-center GCS**
- Annotation-center GCS
- API-center GCS



Graphical concrete syntax

Mapping between abstract syntax and concrete syntax elements

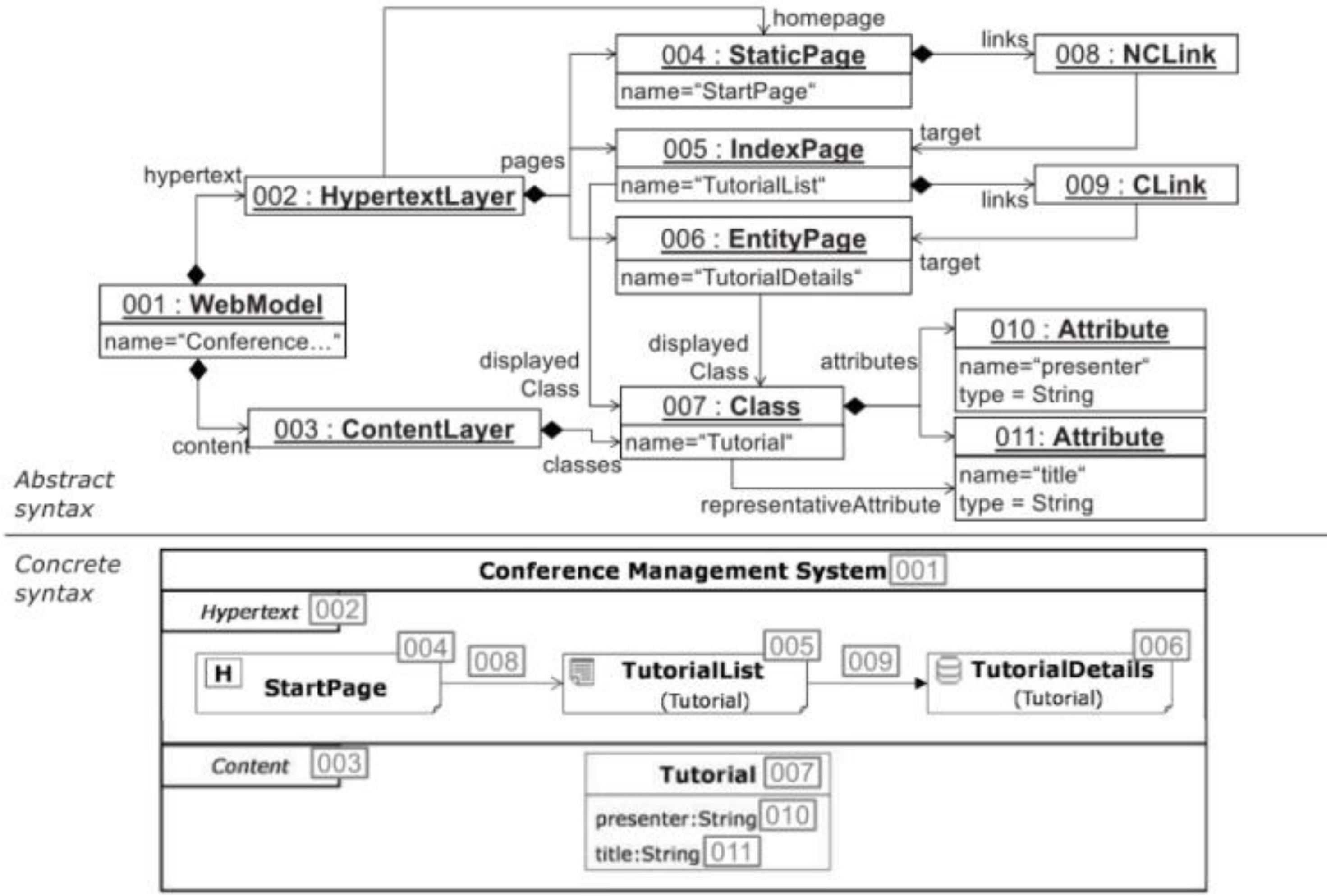
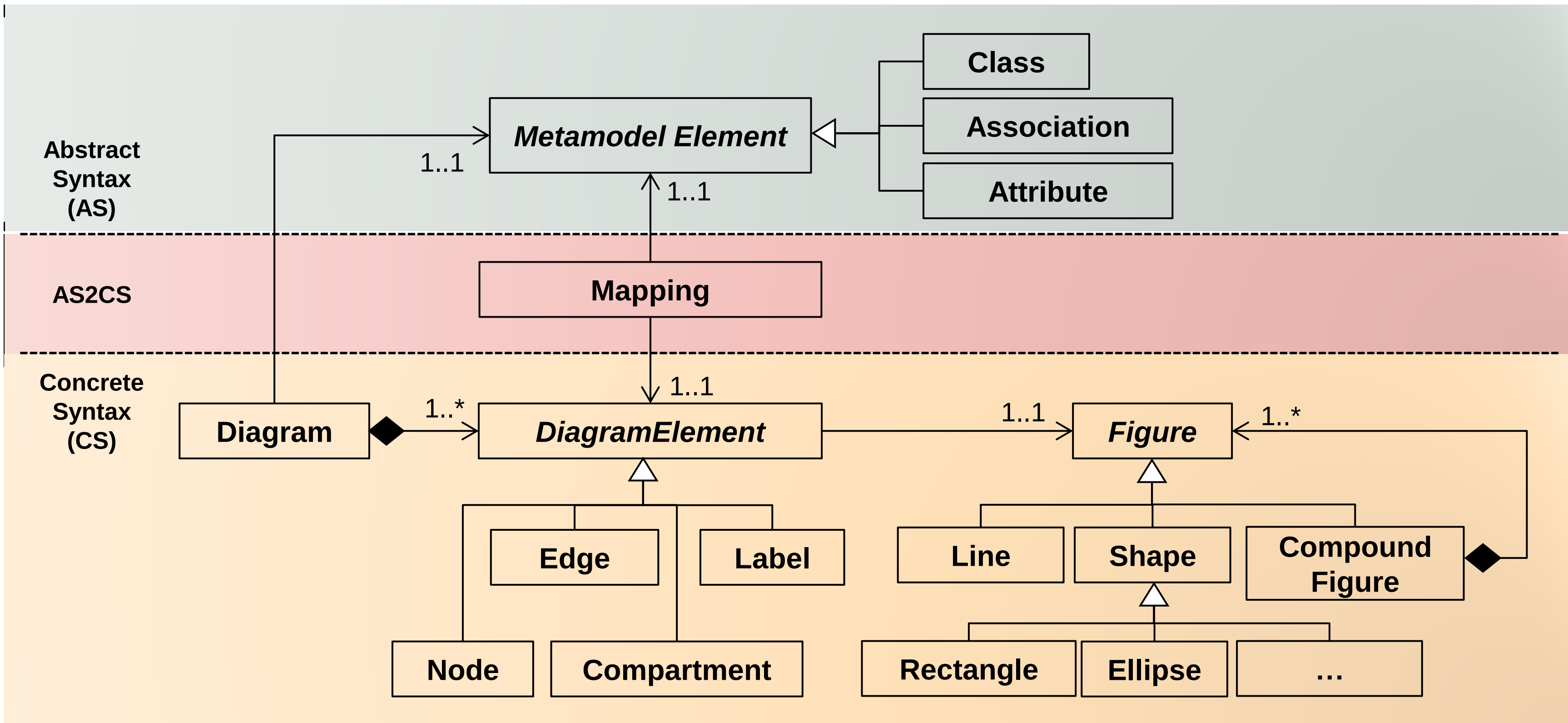


Figure 7.7: sWML model's abstract syntax. Page 91.

Graphical concrete syntax

Generic Metamodel for Graphical Concrete Syntax



Graphical concrete syntax

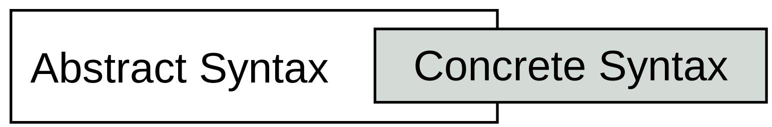
Approaches to GCS development

Mapping-center GCS

Annotation-center GCS

API-center GCS

- The metamodel is annotated with concrete syntax information.
- This approaches directly annotate the metamodel with information about how the elements are visualized.



This approach is supported by

EuGENia framework

Graphical concrete syntax

Approaches to GCS development

Mapping-center GCS

Annotation-center GCS

API-center GCS

- **EuGENia framework** allows to annotate an **Ecore-based metamodel** with GCS information by providing a high-level textual DSML.
- 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!!!

Graphical concrete syntax

Approaches to GCS development

Mapping-center GCS

Annotation-center GCS

API-center GCS

In **EuGENia framework** there are several annotations available for specifying the GCS for a given Ecore-based metamodel.

The main annotations are:

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

Compart-ment

- 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

Graphical concrete syntax

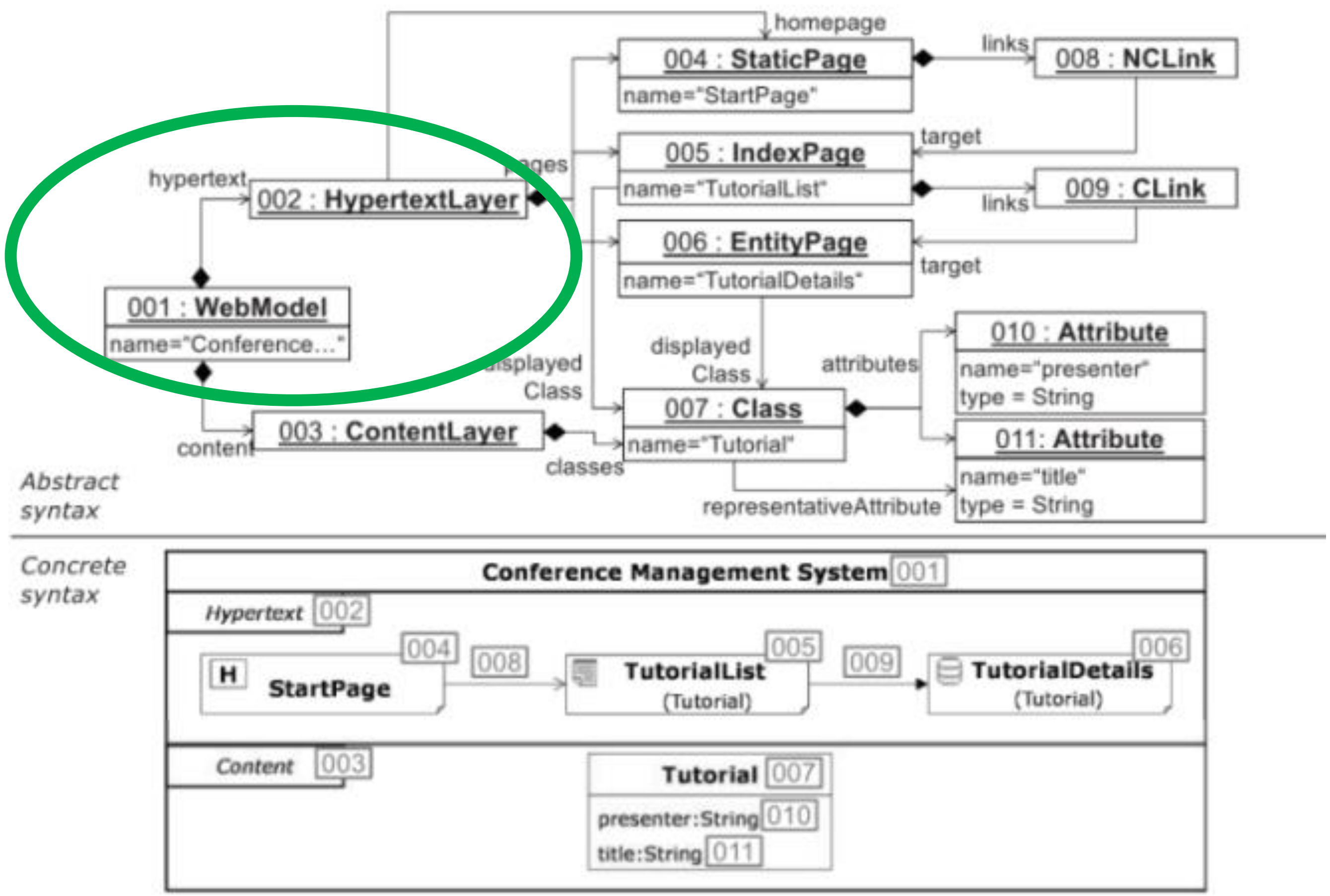
Approaches to GCS development

Mapping-center GCS

Annotation-center GCS

API-center GCS

Case study: Defining a GCS for sWML in EuGENia



Graphical concrete syntax

Approaches to GCS development

Mapping-center GCS

Annotation-center GCS

API-center GCS

Case study: Defining a GCS for sWML in EuGENia

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

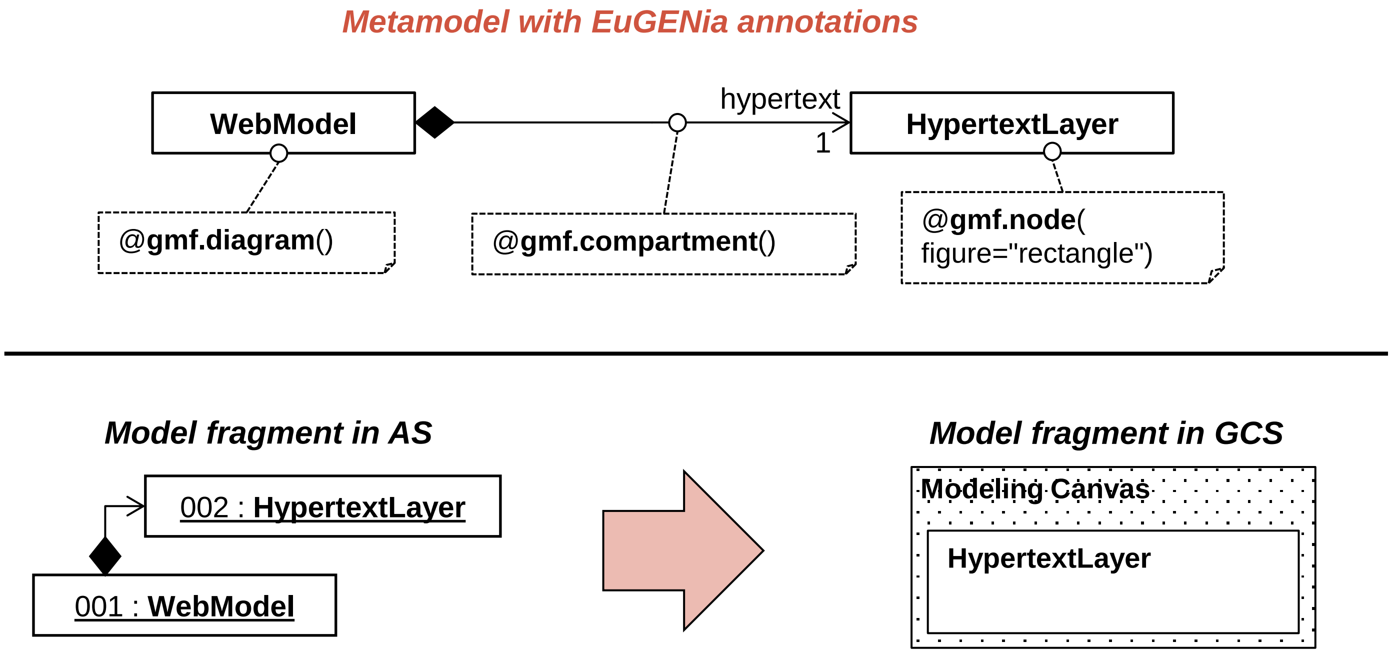


Fig. 7.11: GCS excerpt 1: Diagram, Compartment, and Node annotations.

Graphical concrete syntax

Approaches to GCS development

Mapping-center GCS

Annotation-center GCS

API-center GCS

Case study: Defining a GCS for sWML in EuGENia

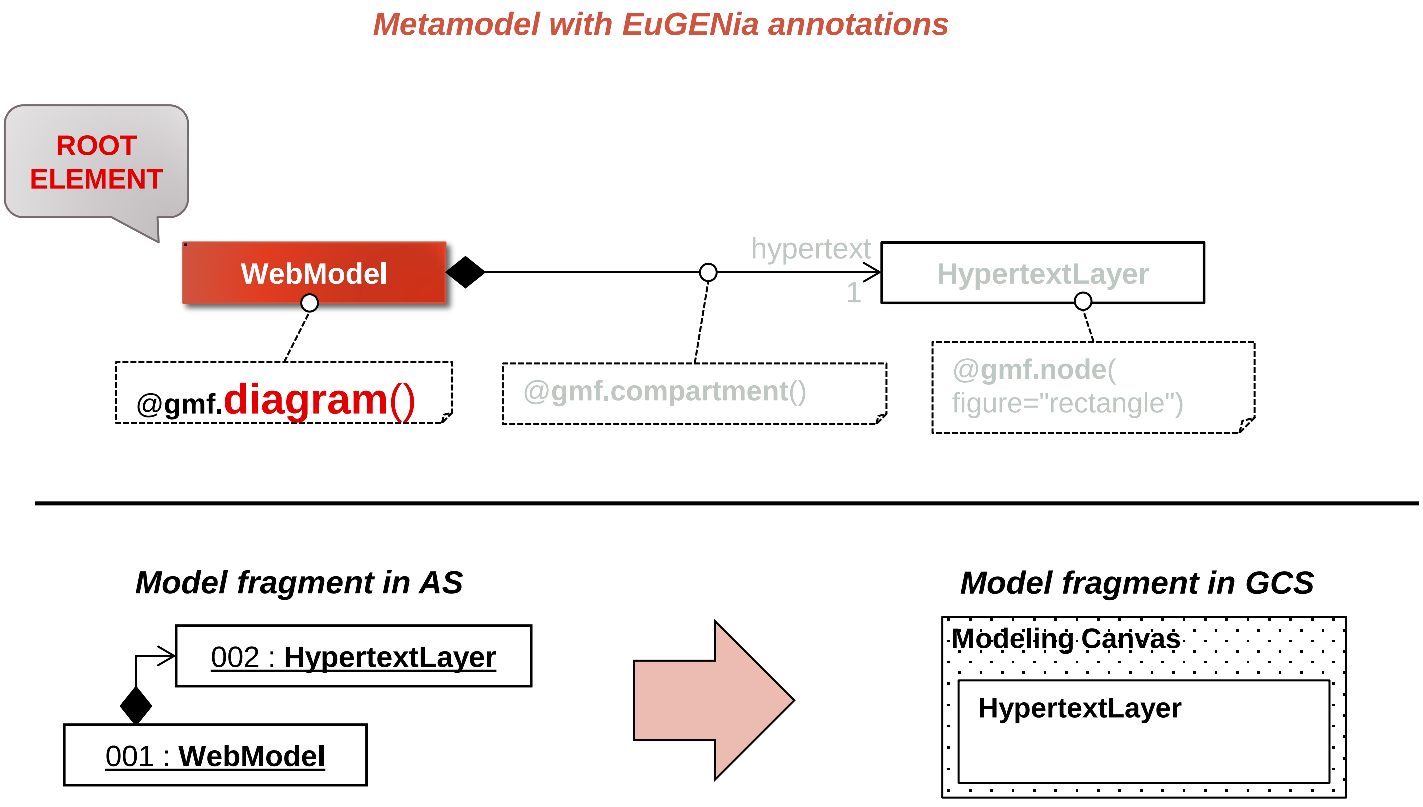


Fig. 7.11: GCS excerpt 1: Diagram, Compartment, and Node annotations.

Graphical concrete syntax

Approaches to GCS development

Mapping-center GCS

Annotation-center GCS

API-center GCS

Case study: Defining a GCS for sWML in EuGENia

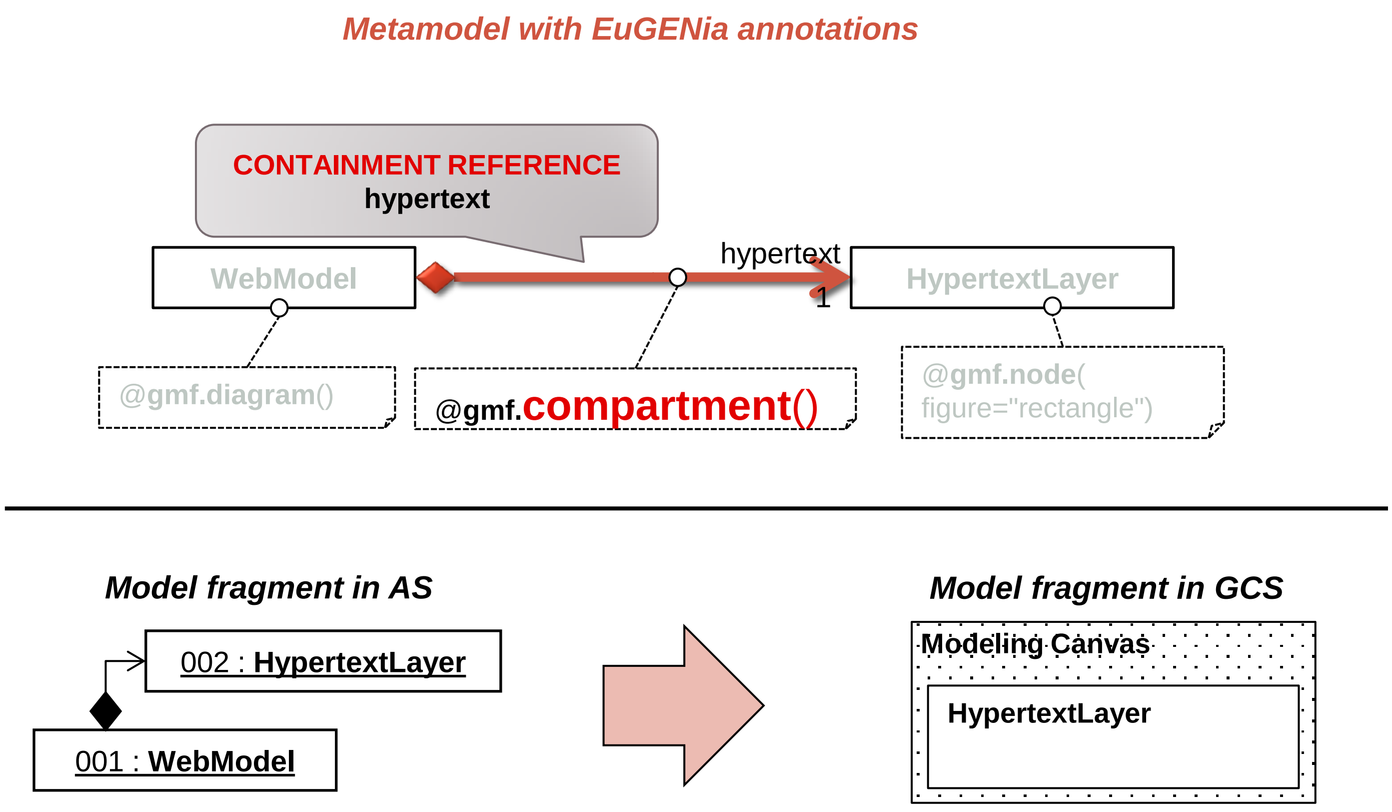


Fig. 7.11: GCS excerpt 1: Diagram, Compartment, and Node annotations.

Graphical concrete syntax

Approaches to GCS development

Mapping-center GCS

Annotation-center GCS

API-center GCS

Case study: Defining a GCS for sWML in EuGENia

Metamodel with EuGENia annotations

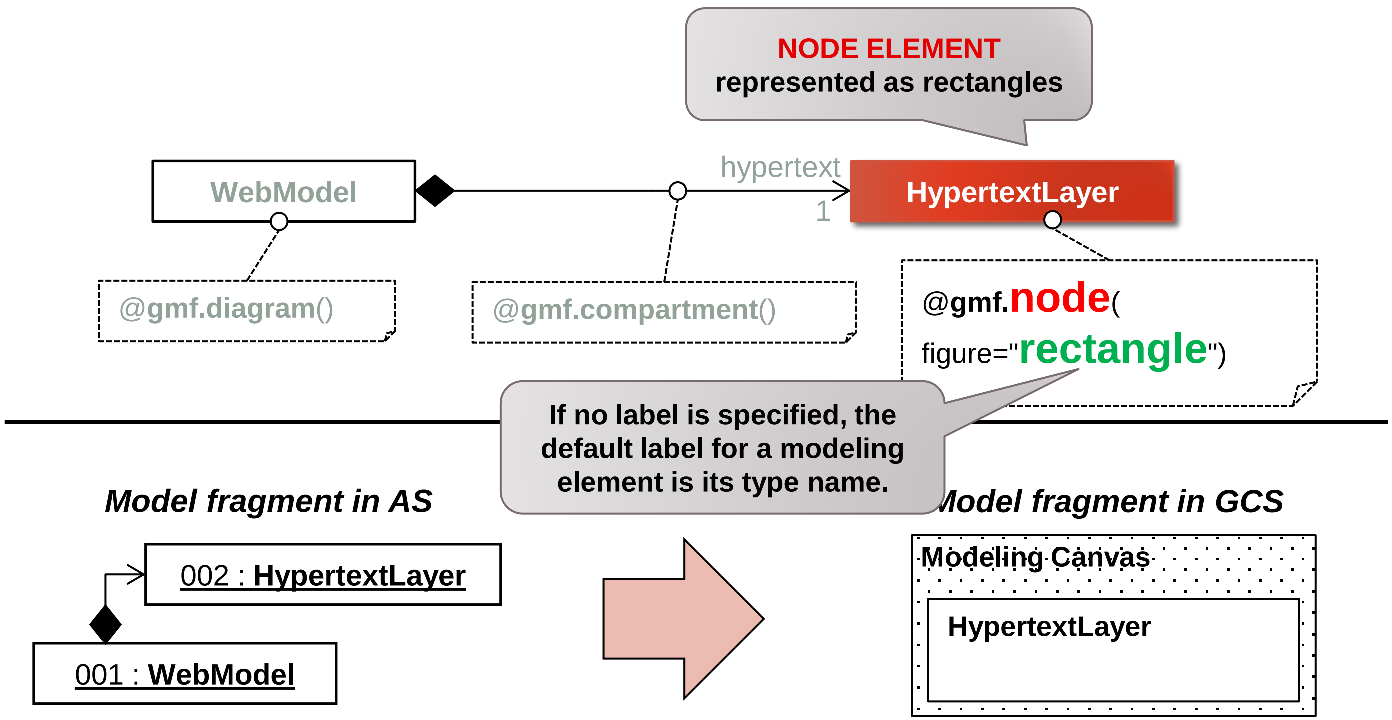


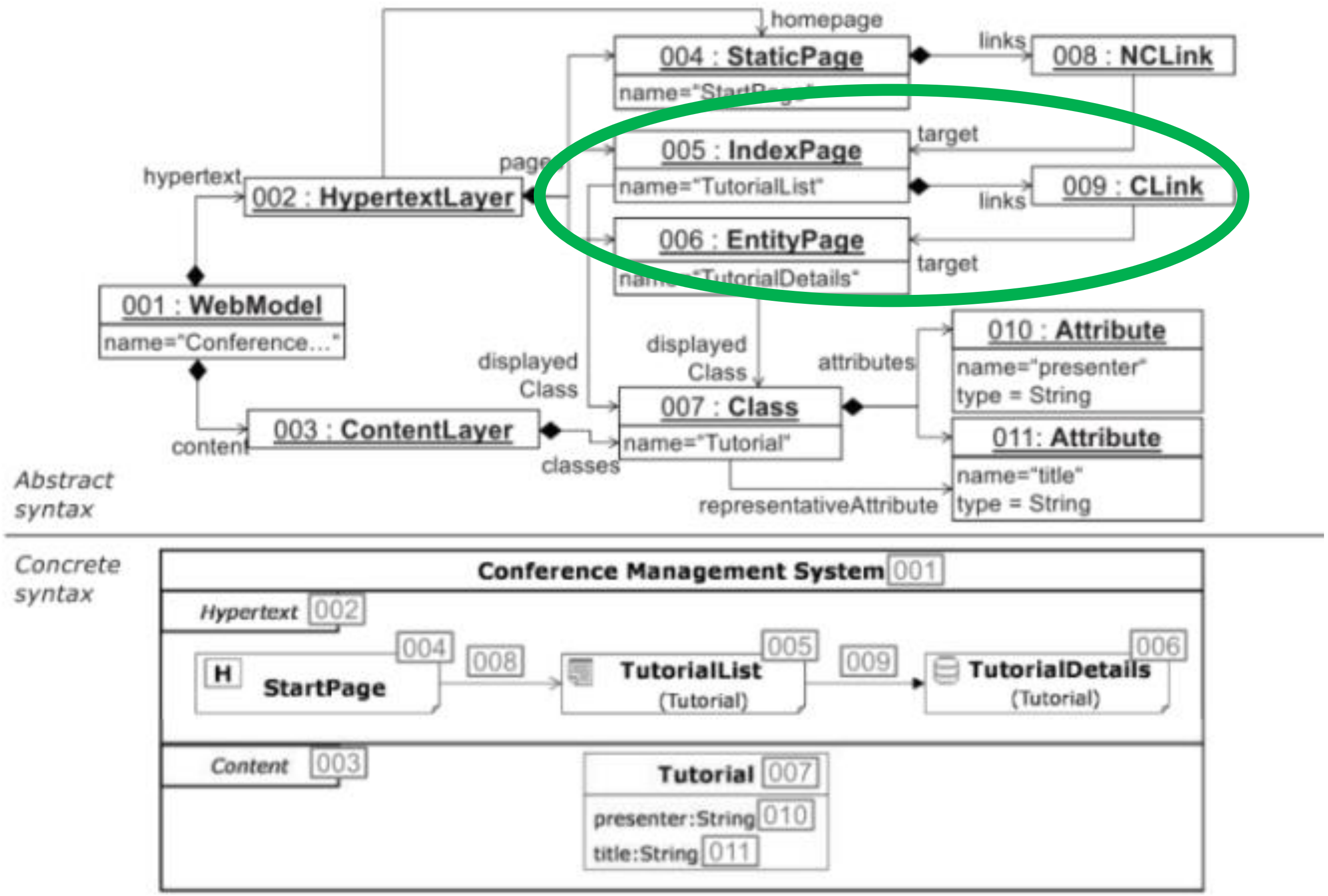
Fig. 7.11: GCS excerpt 1: Diagram, Compartment, and Node annotations.

Graphical concrete syntax

Approaches to GCS development

- Mapping-center GCS
- **Annotation-center GCS**
- API-center GCS

Case study: Defining a GCS for sWML in EuGENia



Graphical concrete syntax

Approaches to GCS development

Mapping-center GCS

Annotation-center GCS

API-center GCS

Case study: Defining a GCS for sWML in EuGENia

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

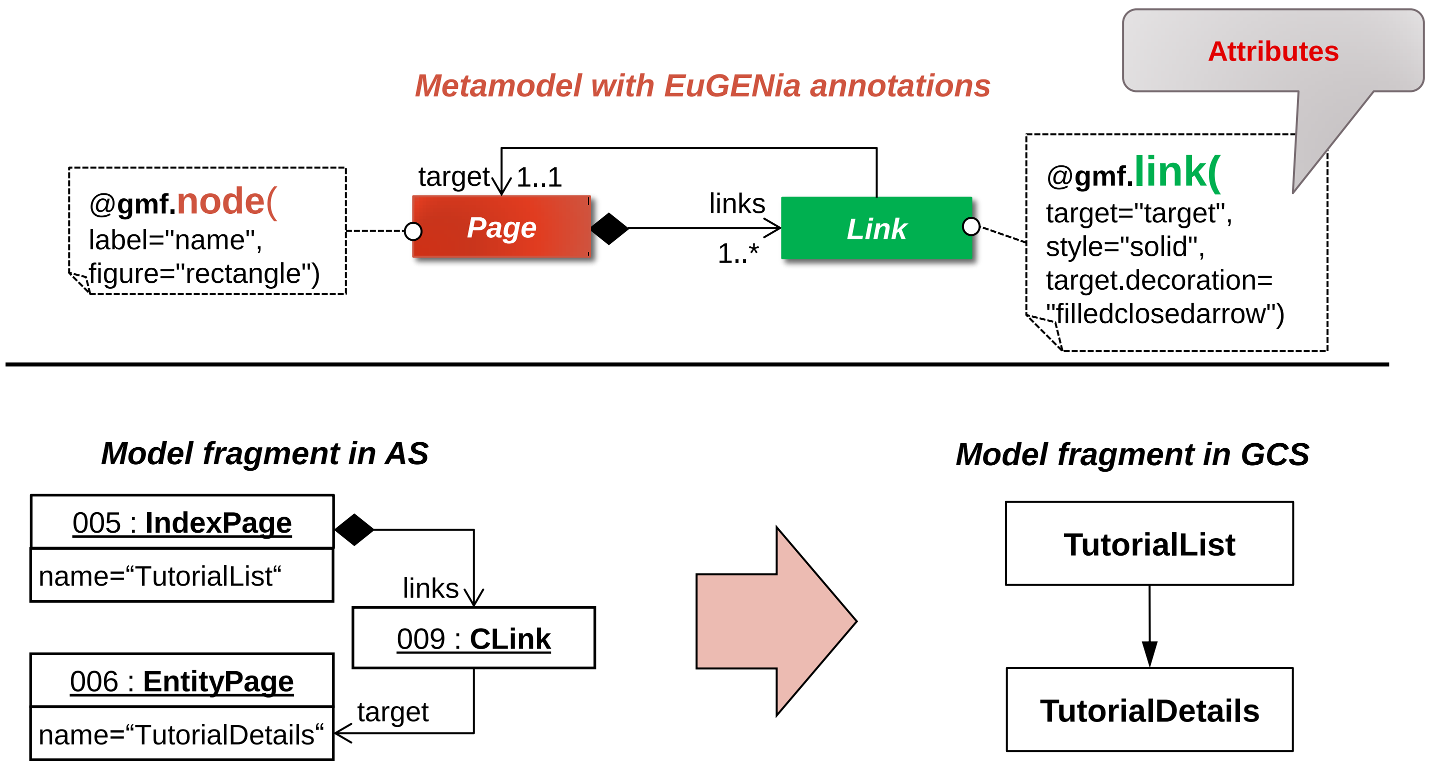


Fig. 7.12: GCS excerpt 2: Node and Link annotations.

Graphical concrete syntax

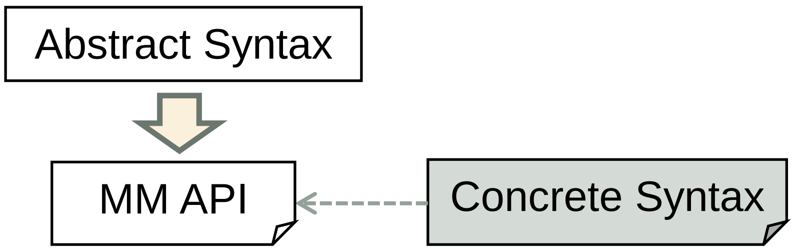
Approaches to GCS development

Mapping-center GCS

Annotation-center GCS

API-center GCS

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



This approach is supported by

Graphiti

Graphical concrete syntax

Approaches to GCS development

Mapping-center GCS

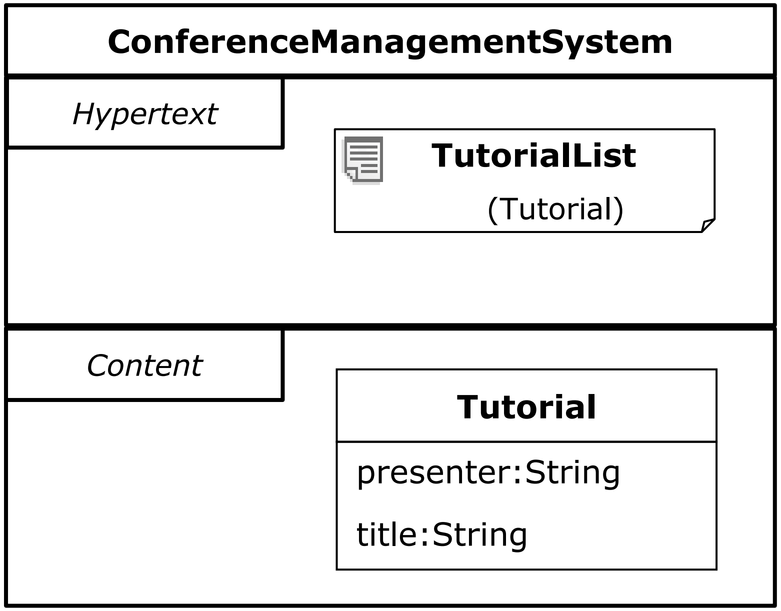
Annotation-center GCS

API-center GCS

- 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.
  - ***Link models*** establish the mapping between abstract and concrete syntax elements.
- DSL on top of Graphiti: Spray

Every GCS is transformable to a TCS

Example: sWML

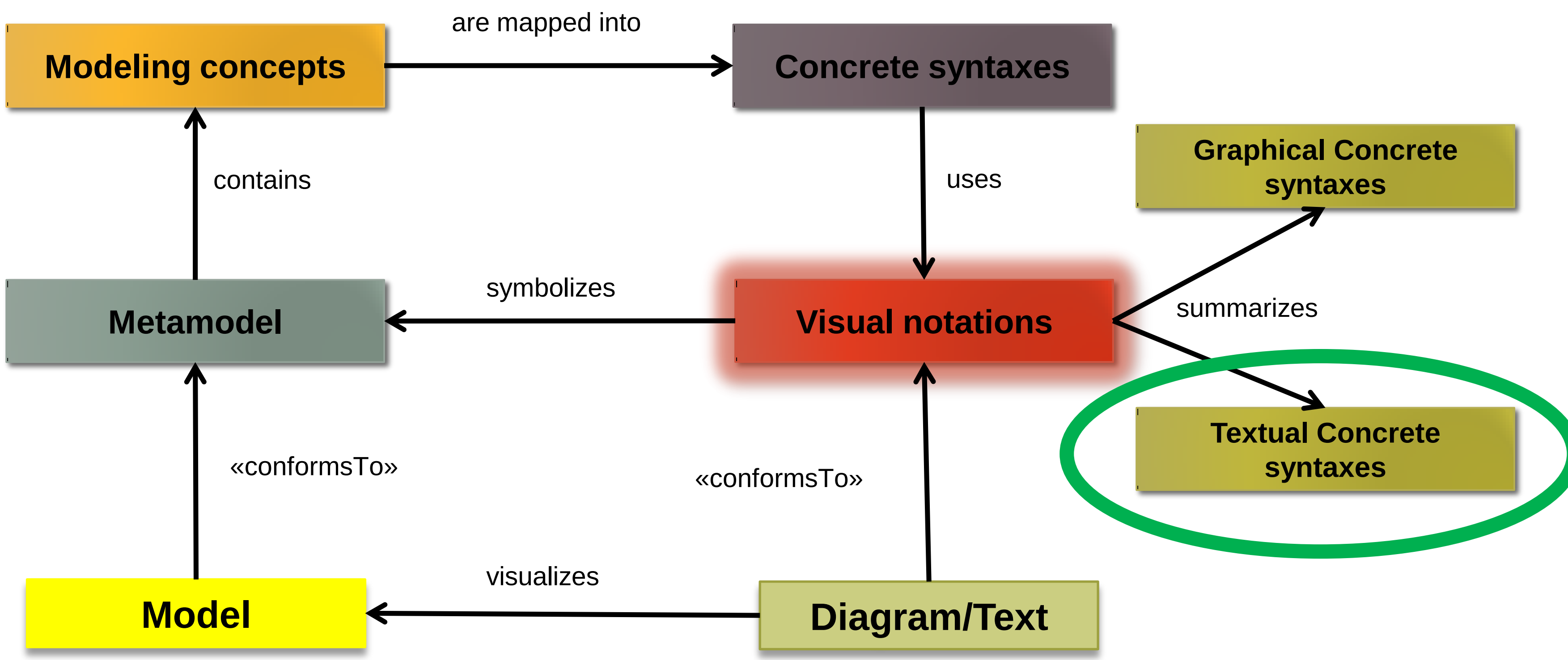



```
webapp ConferenceManagementSystem{

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

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

}
```


Concrete syntax development

Visual notations

- The **visual notation** of a model language is referred as **concrete syntax**
- **Visual notation** introduces symbols for modeling concepts.



TEXTUAL CONCRETE SYNTAX

Textual concrete syntax

Overview

- **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 concrete syntax

Overview

- **Assumption fundamental of textual specifications:**
  - Comprised text consistes of a sequence of characteres.
- **Not every** arbitrary sequence of characters represents a valid specification
- From a metamodel, **only a generic grammar may be derived** which allows the generic rendering of models textually as well as the parsing of text into models.
- In particular, **language-specific keywords** enhance the readability of textual specifications a lot.

Textual concrete syntax

Anatomy of textual languages

The following kinds of TCS elements can be identified:

Model information	TCS has to support model information stored in abstract syntaxes. (i.e., **name and type**)
Keywords	Are used for introducing the different model elements. (i.e., **reserved words**)
Scope borders	Special symbols, so-called scope borders, defines the borders of a model element. (i.e., **{ }**)
Separation characters	A special character is used for separating the entries of the list. (i.e., **;**)
Links	Identifiers have to be defined for elements which may be used to reference na element from another element by stating the identifier value. (i.e., **class names**)

Textual concrete syntax

Approaches to TCS development

- Metamodels **do not provide information** about the other kinds of TCS elements.

- For the definition of this TCS specific information, two approaches are currently available in MDE:

Generic TCS
- A textual syntax generically applicable for all kinds of models .
- The metamodel is sufficient to derive a TCS.

Language-specific TCS
- Resulting artifacts:
  - A metamodel for the abstract syntax.
  - TCS for the models.

Textual concrete syntax

Approaches to TCS development

Xtext tool

Language-specific TCS approaches:

Metamodel first

- 1. To define abstract syntax by means of a metamodel.
- 2. Textual syntax is defined based on the metamodel
- 3. To render each model elements into a text representation using a **text production rule**.

Grammar first

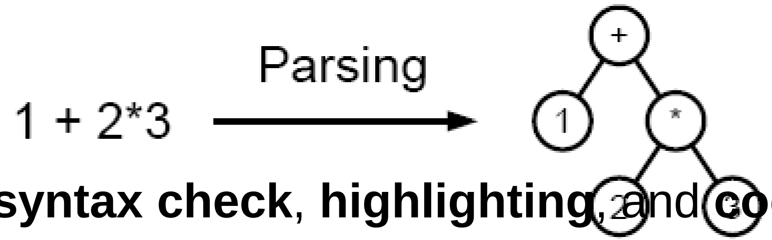
- 1. Start with the language definition developing the grammar defining the abstract and concrete syntax at once as a single specification.
- 2. The metamodel is automatically inferred from the grammar by dedicated metamodel derivation rules.

Textual concrete syntax

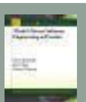
Using Xtext

Case study: Defining a TCS for sWML in Xtext

- **Xtext** is used for developing **textual domain specific languages**
- **Grammar** definition similar to **EBNF**, but with **additional features** inspired by **metamodeling**
- Creates **metam** ar definition



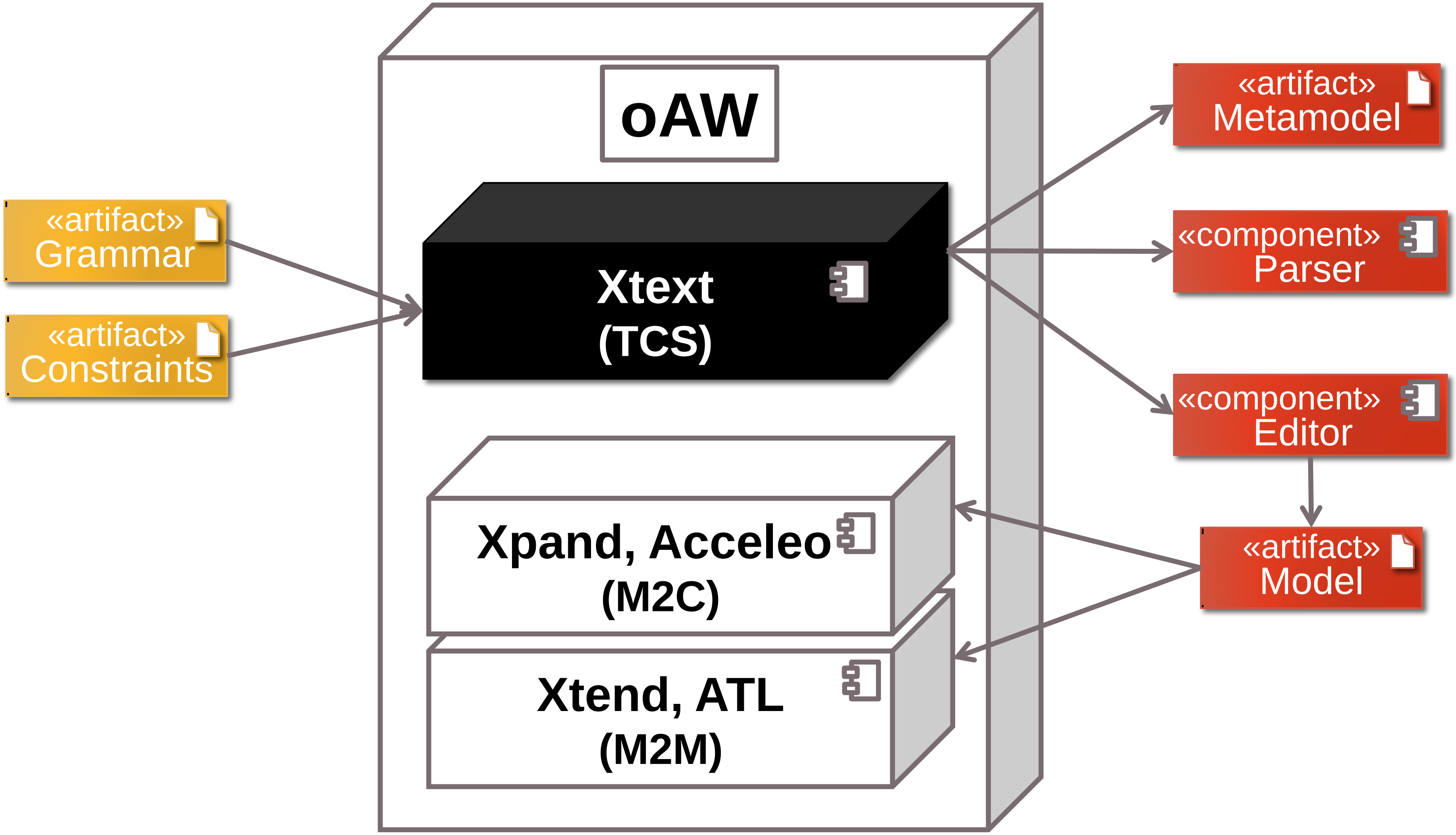
- Editor supports **syntax check**, **highlighting**, and **code completion**
- **Context-sensitive constraints** on the grammar described in OCL-like language



Textual concrete syntax

Using Xtext

Xtext architecture overview



Textual concrete syntax

Using Xtext

Xtext production rules

- **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

Textual concrete syntax

Using Xtext

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

Textual concrete syntax

Using Xtext

Xtext grammar

- **Type rules**
  - For each type rule a **class** is generated in the metamodel
  - Class name corresponds to rule name
  - Used to define modeling concepts

- **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

Textual concrete syntax

Using Xtext

Xtext grammar

Examples

- Assignment

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

- Cross References

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


Textual concrete syntax

Using Xtext

Xtext grammar

- **Enum rules**
  - Map Strings to enumeration literals
  - Are used for defining value enumerations

- **Examples**

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

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


Textual concrete syntax

Using Xtext

Case study: Defini… …ML in Xtext

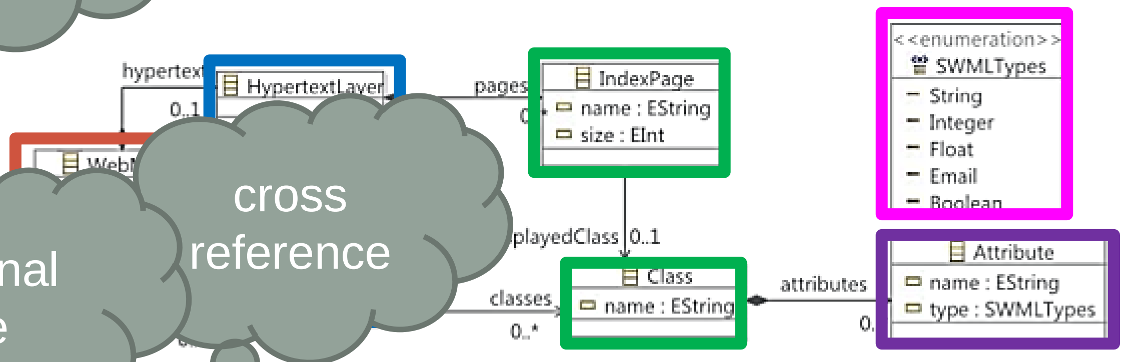
Containment reference

Terminal rule

cross reference

Enum rules

```
WebModel :
        'webapp' name=ID '{'
                hypertext=HypertextLayer
                content=ContentLayer
        '}' ;

        HypertextLayer :
                'hypertext

        '}' ;

        IndexPage :
                'index' name=I…            …playedClass=[Class] '['resultsPerPage']' '{' ... '}' ;

        terminal resultsPerPage returns ecore::EInt: ('10' | '20' | '30' ) ;


        ContentLAyer :
                'content {'
                        classes+=Class+
                '}' ;

        Class :
                'c…                    …+=Attribute+ '}' ;

        Attribute :
                'att…                  …LTypes ';' ;

        enum SWMLTypes :
                String | Integer | Float | Email | Boolean ;
```



SUMMARY

Summary

Final remarks (1/3)

- **Meta-modeling language**:
  - It is a modeling language for create modeling languages.
- **Metamodel**:
  - Modeling how to model.
- **Meta-meta model**:
  - Language for defining how to build metamodels.
- Meta models and meta-metamodels **only define the abstract syntaxes** of the languages.
  - Concrete syntaxes or semantics **are not covered** by them.

Summary

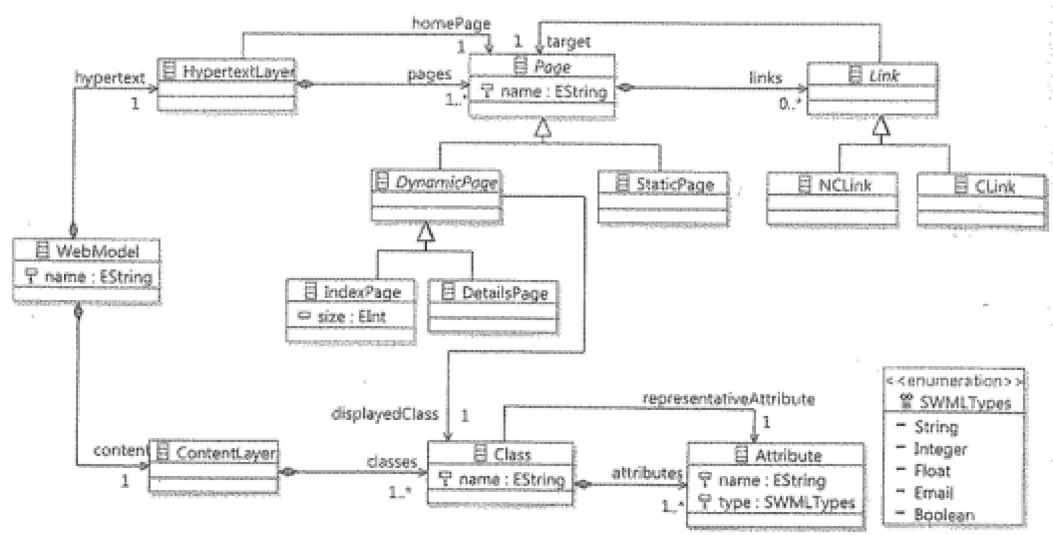
Final remarks (2/3)

- **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**

Summary

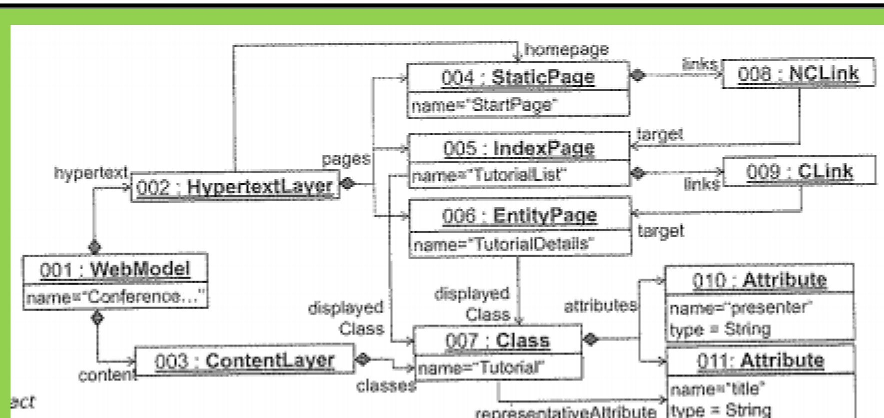
Final remarks (3/3)

Metamodel

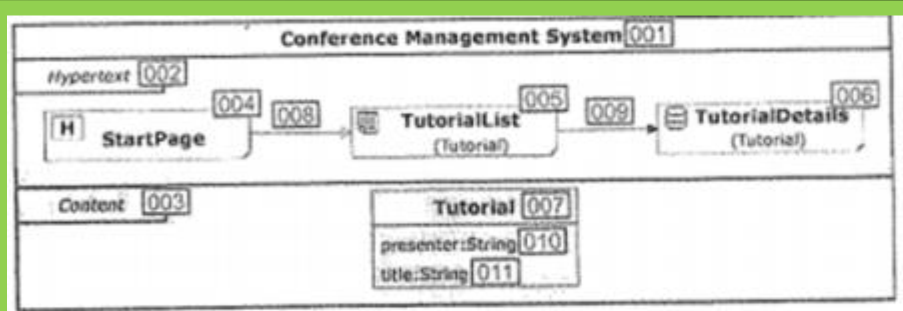


Model

Abstract syntax



Concrete syntax




```
webapp ConferenceManagementSystem{

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

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

}
```

OBRIGADO

GRACIAS

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