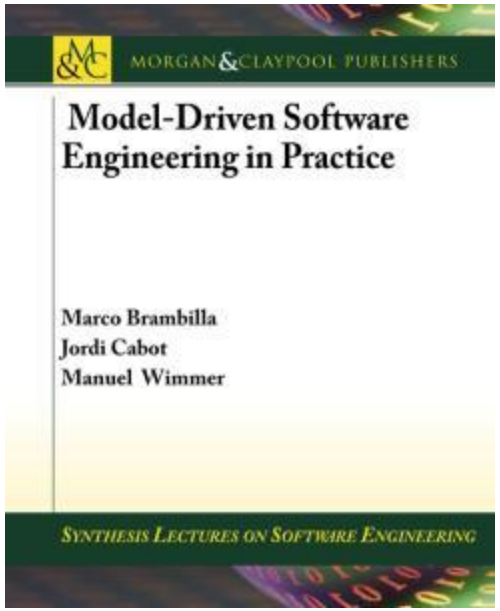
MORGAN & CLAYPOOL PUBLISHERS

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

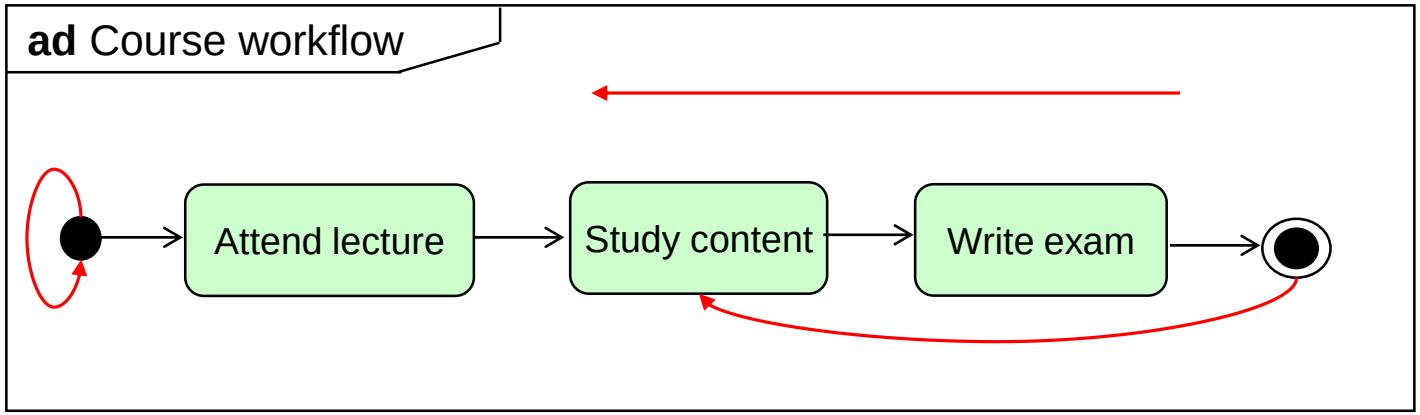
- Introduction
- Abstract Syntax
- Graphical Concrete Syntax
- Textual Concrete Syntax

INTRODUCTION

Introduction

What to expect from this lecture?

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



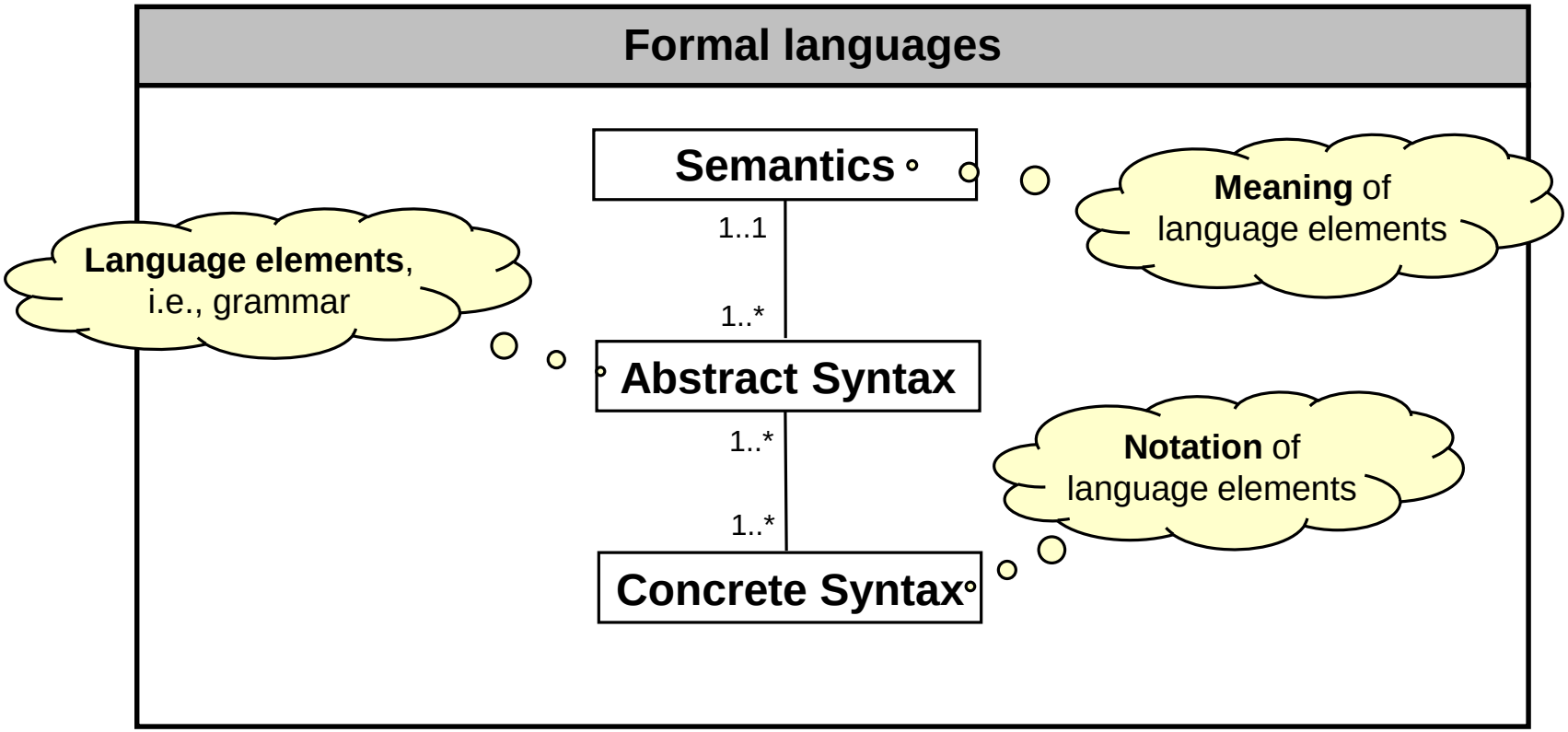
- **Question**: Is this UML Activity diagram **valid**?
- **Answer**: Check the **UML metamodel**!
  - Prefix „meta“: an operation is applied to itself
  - Further examples: meta-discussion, meta-learning, …

- **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**

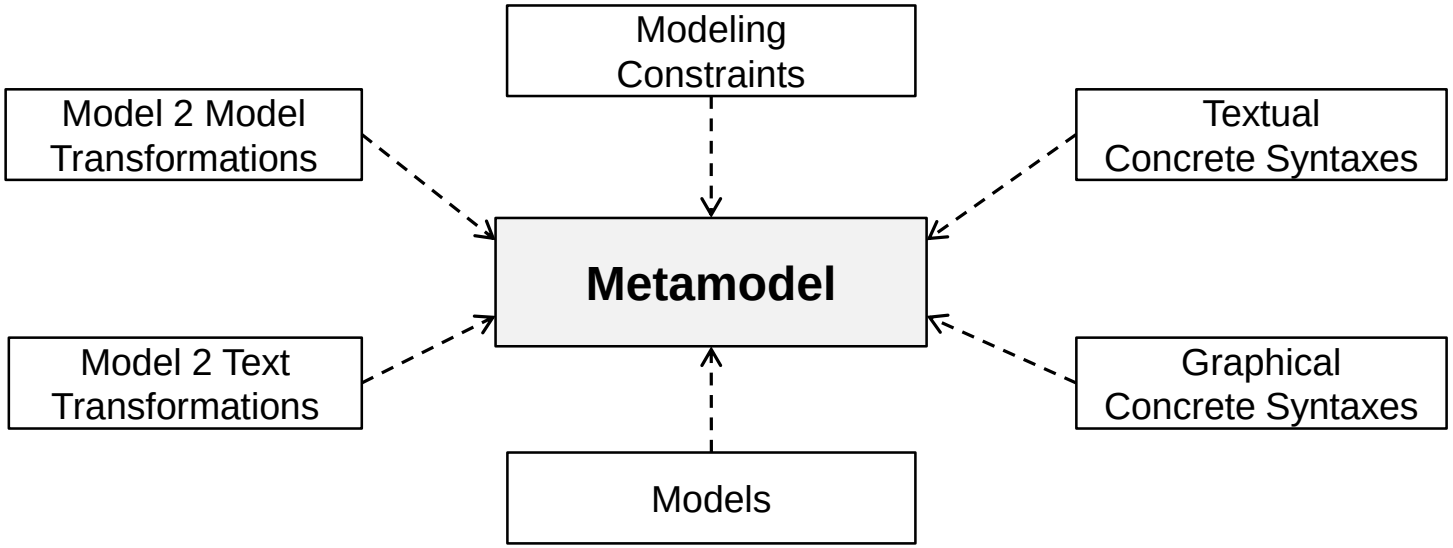
- **Additional components**
  - **Extension** of the language by new language concepts
    - Domain or technology specific extensions, e.g., see UML Profiles
  - **Mapping** to other languages, domains
    - Examples: UML2Java, UML2SetTheory, PetriNet2BPEL, …
    - May act as translational semantic definition

ABSTRACT SYNTAX

Introduction

Spirit and purpose of metamodeling 1/3

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



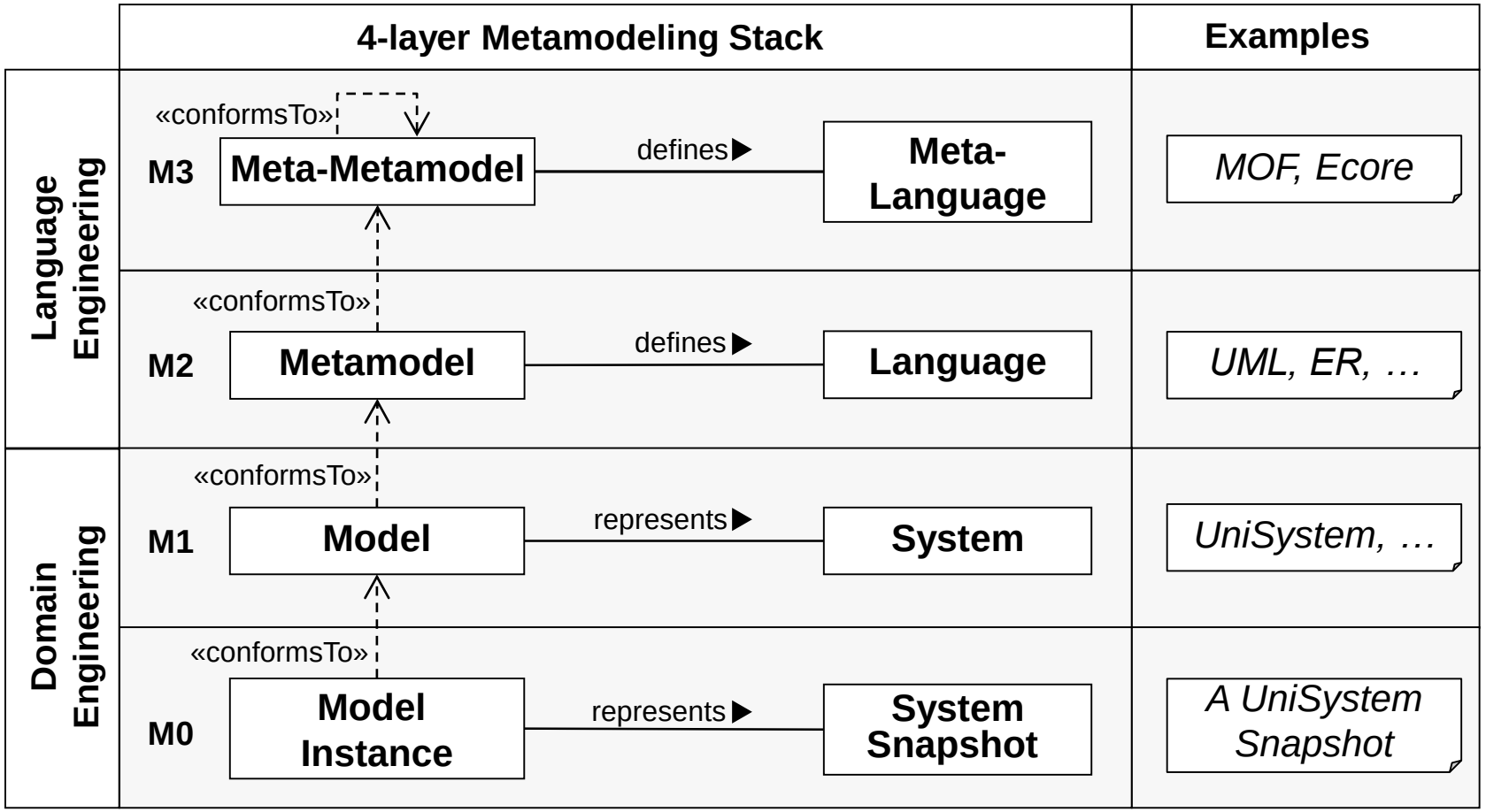
Introduction

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

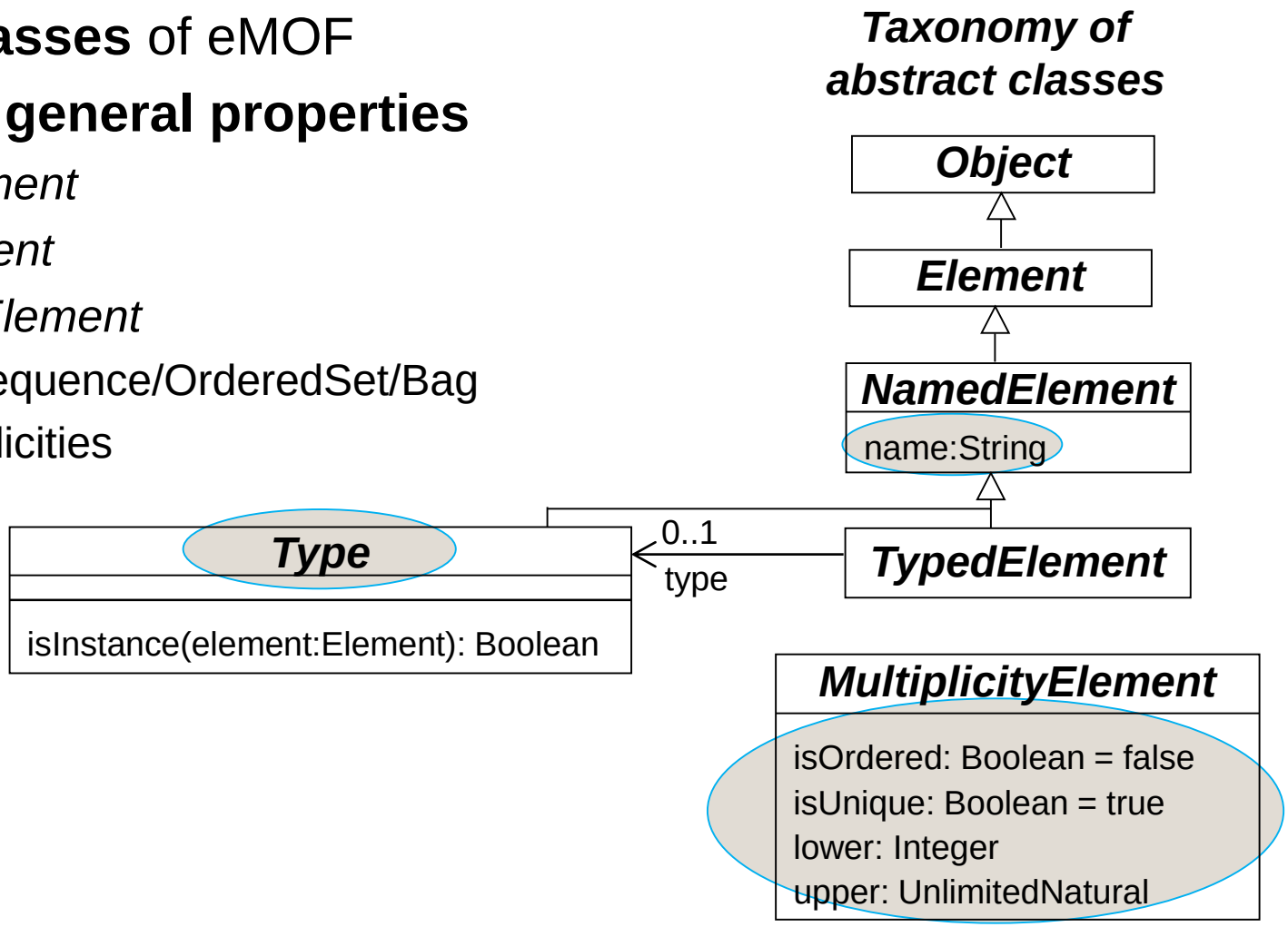
Spirit and purpose of metamodeling 3/3



MOF – Meta Object Facility

Language architecture of MOF 2.0

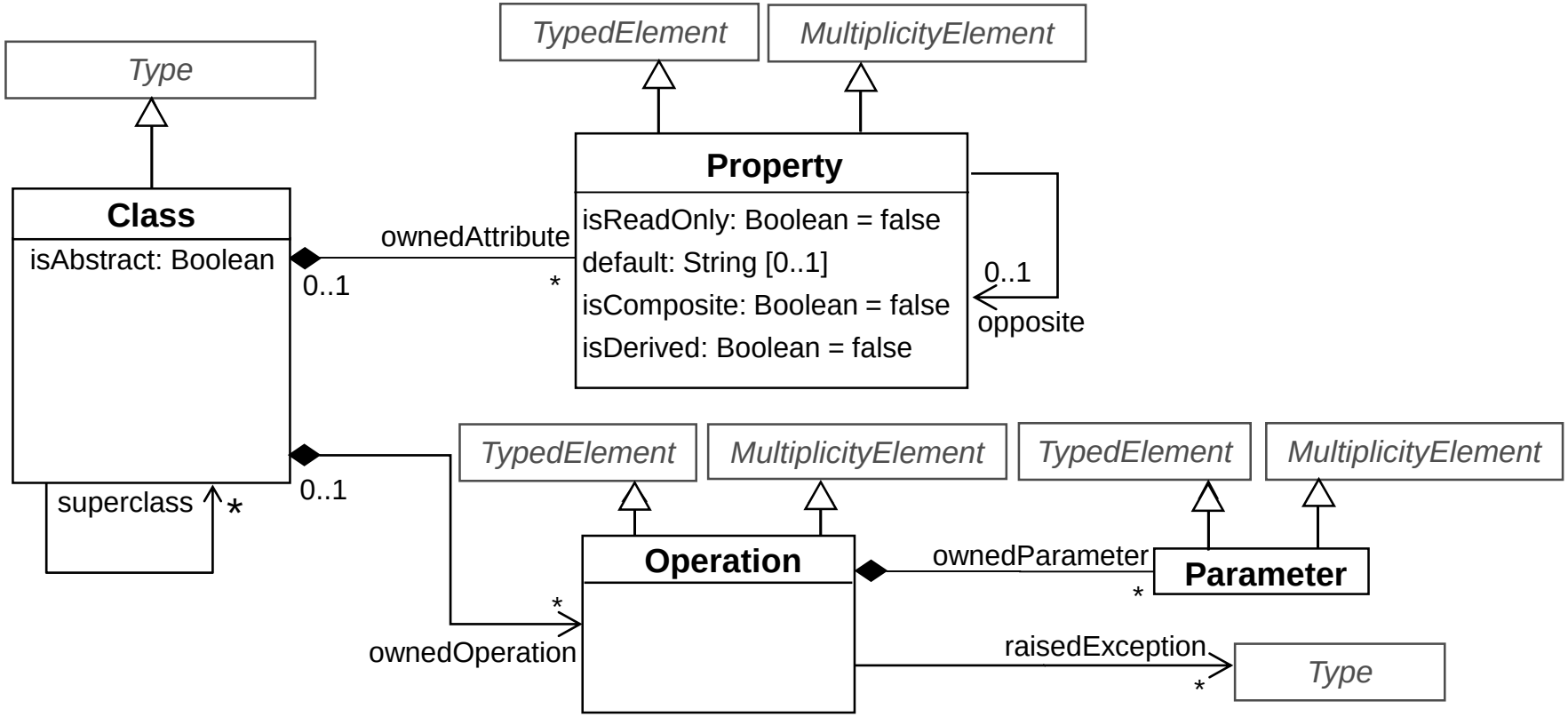
- **Abstract classes** of eMOF
- Definition of **general properties**
  - *NamedElement*
  - *TypedElement*
  - *MultiplicityElement*
    - Set/Sequence/OrderedSet/Bag
    - Multiplicities



MOF – Meta Object Facility

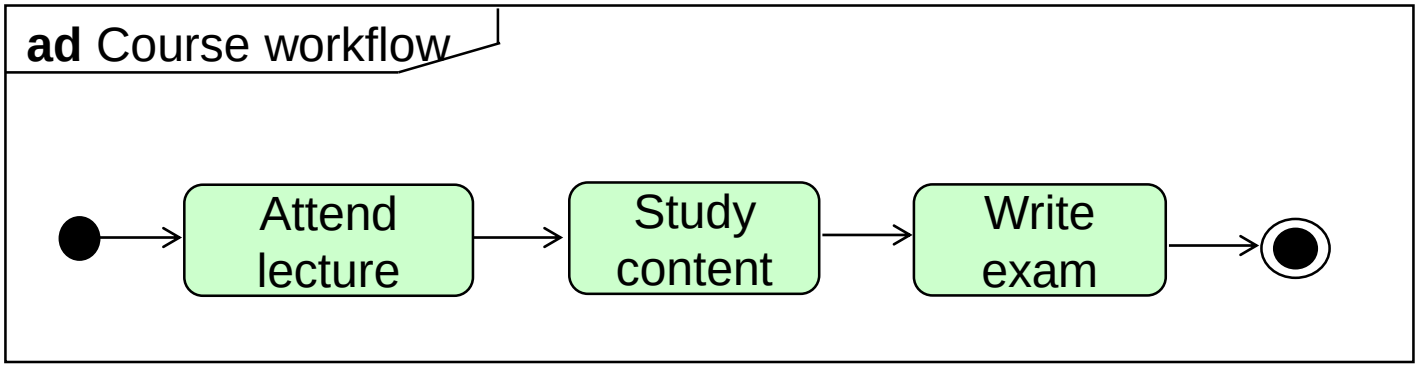
Language architecture of MOF 2.0

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



Example 1/9

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



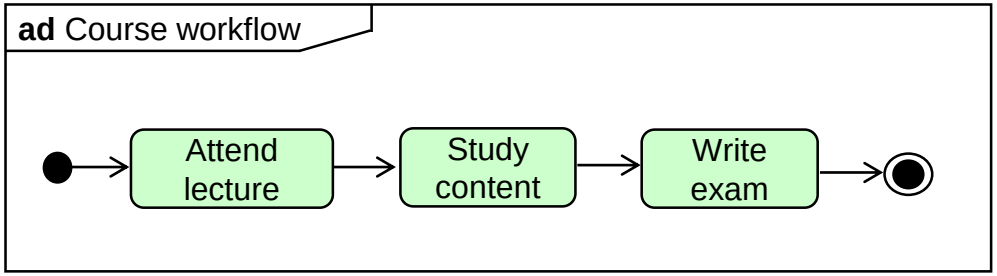
- Question: How does a possible metamodel to this language look like?

- Answer: apply metamodel development process!

Example 2/9

Identification of the modeling concepts

Example model = Reference Model



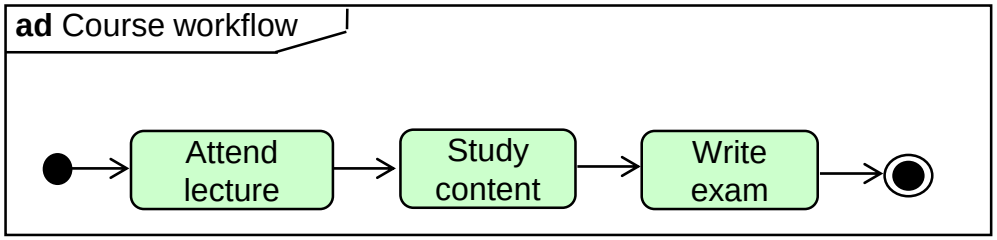
Notation table

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

Example 3/9

Determining the properties of the modeling concepts

Example model

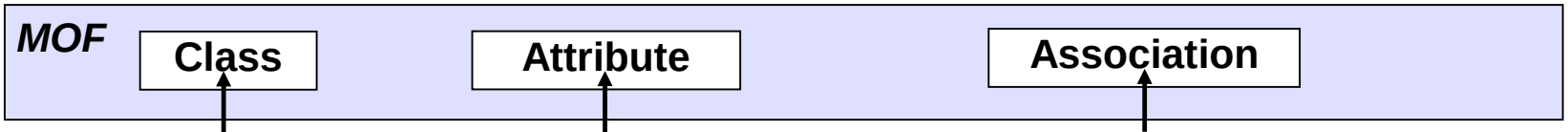


Modeling concept table

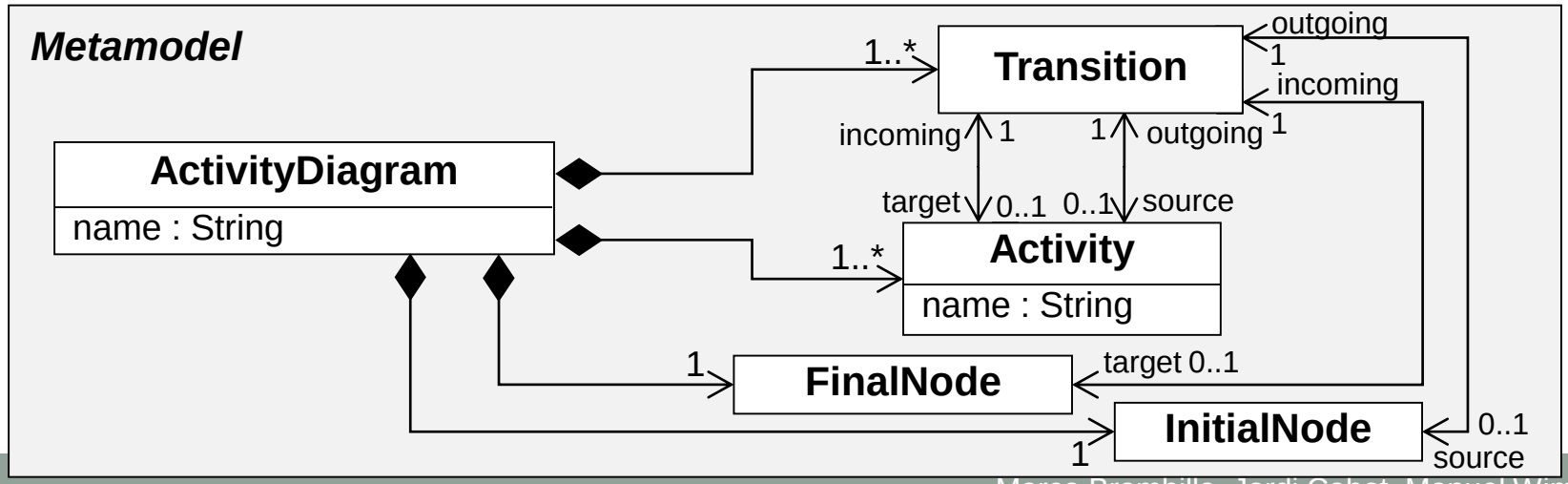
Concept	***Intrinsic properties***	***Extrinsic properties***
ActivityDiagram	Name	1 *InitialNode* 1 *FinalNode* Unlimited number of *Activities* and *Transitions*
FinalNode	-	Incoming *Transitions*
InitialNode	-	Outgoing *Transitions*
Activity	Name	Incoming and outgoing *Transitions*
Transition	-	Source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

Example 4/9

Object-oriented design of the language



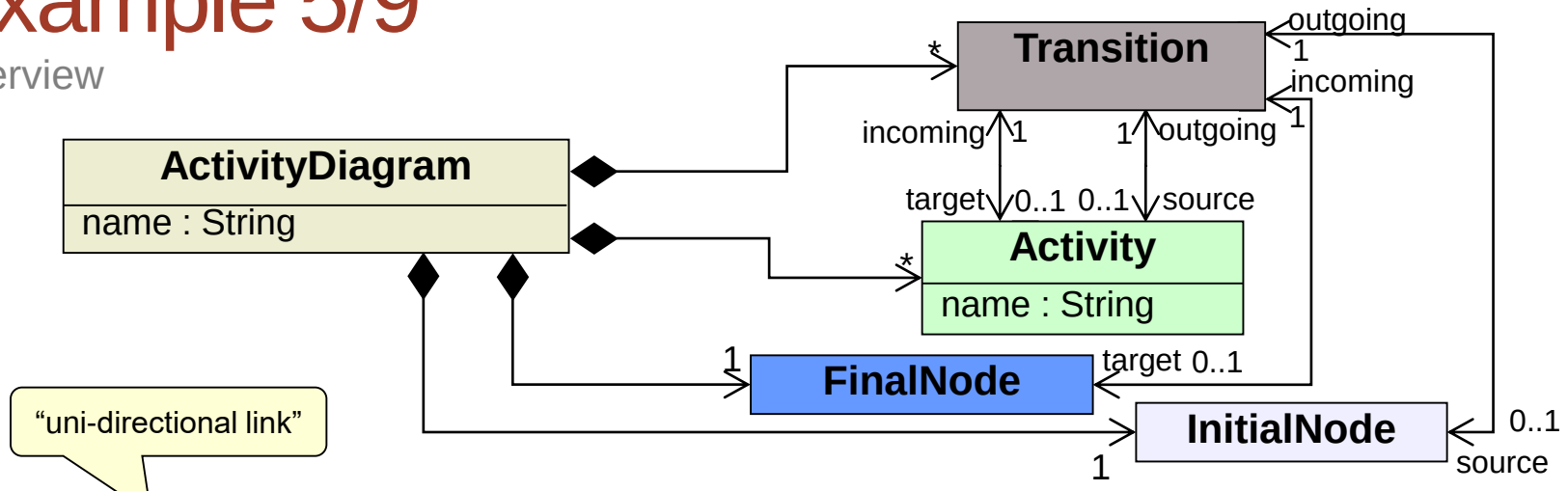
Concept	*Intrinsic properties*	*Extrinsic properties*
ActivityDiagram	Name	1 *InitialNode* 1 *FinalNode* Unlimited number of Activities and Transitions
FinalNode	-	Incoming *Transition*
InitialNode	-	Outgoing *Transition*
Activity	Name	Incoming and outgoing *Transition*
Transition	-	Source node and target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



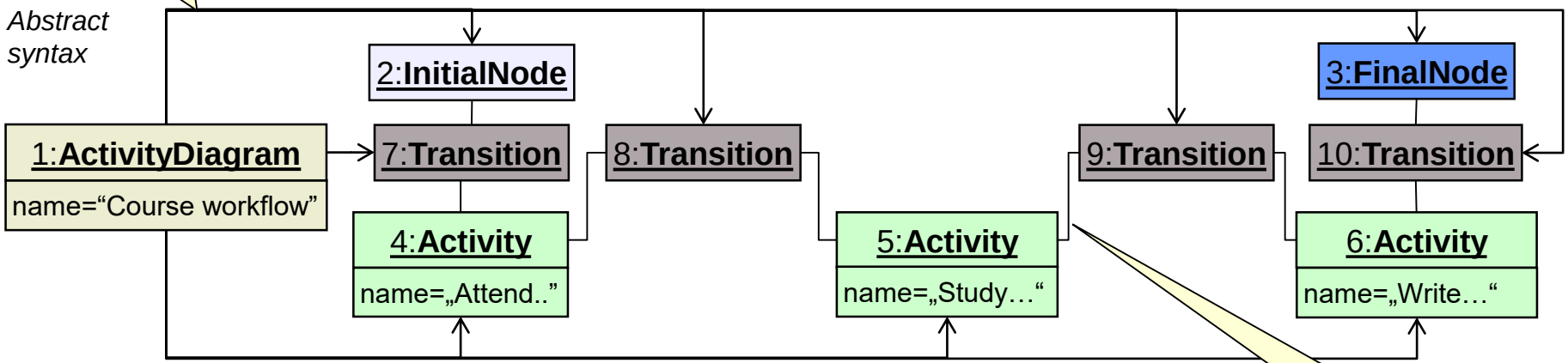
Example 5/9

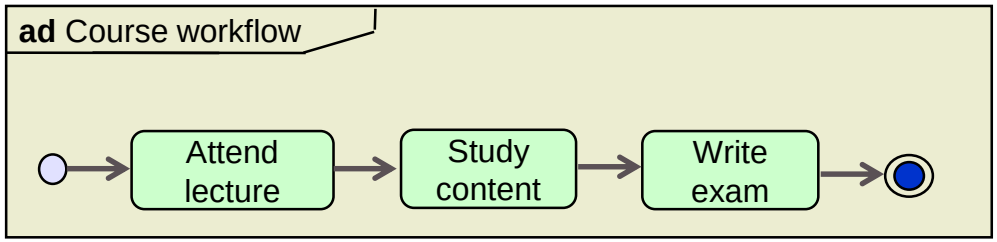
Overview

Metamodel



Model

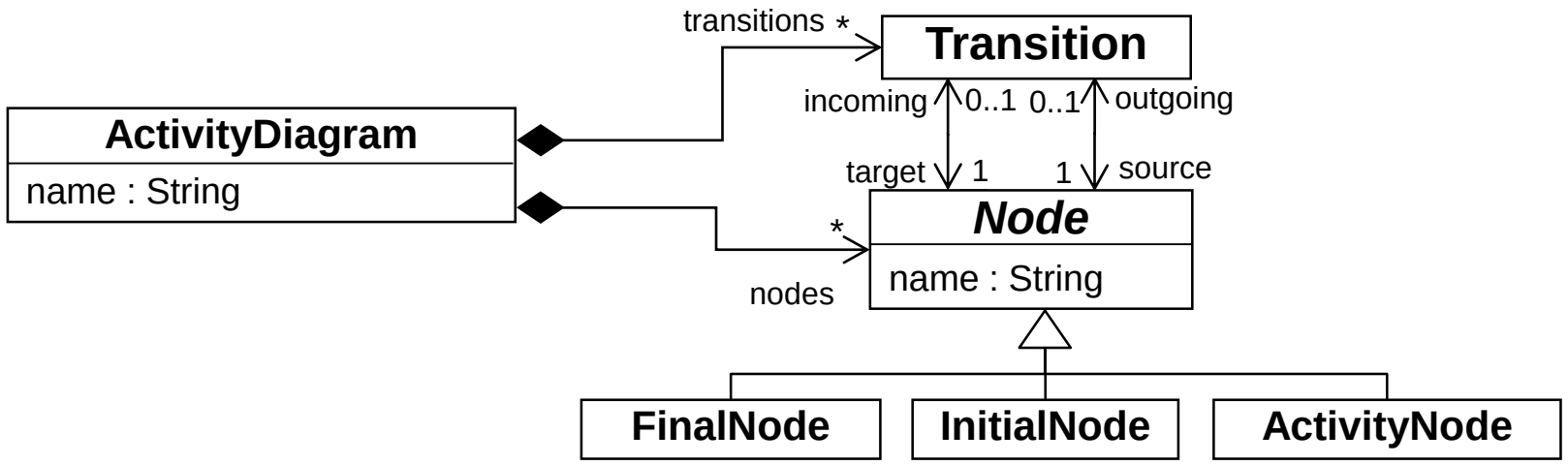




Example 6/9

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 …
```

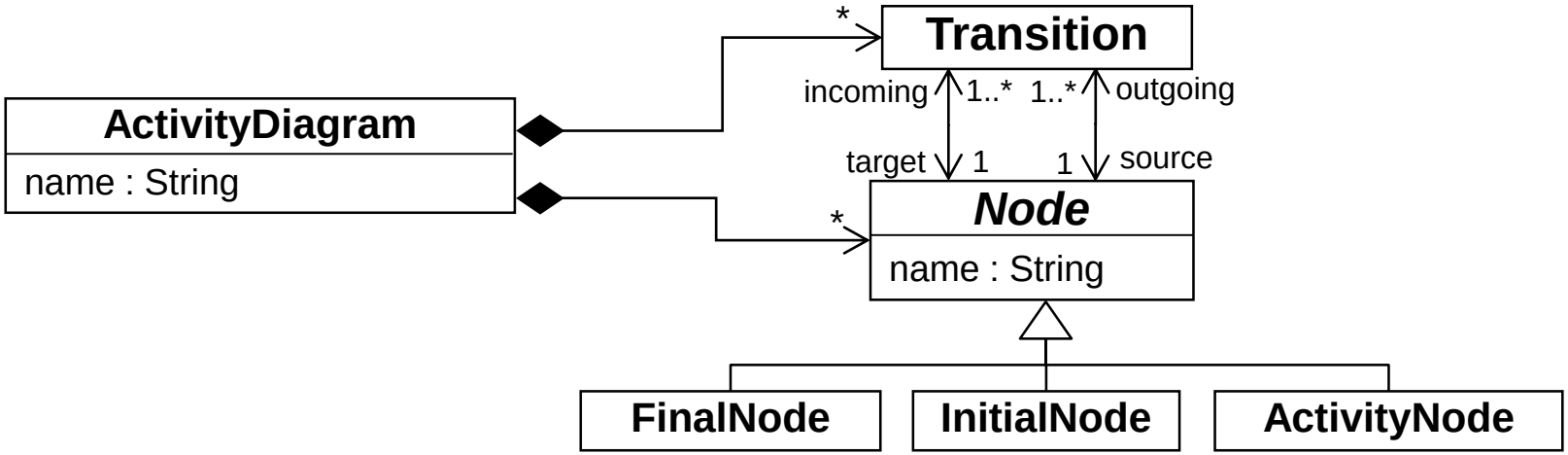

Example 7/9

Impact on existing models

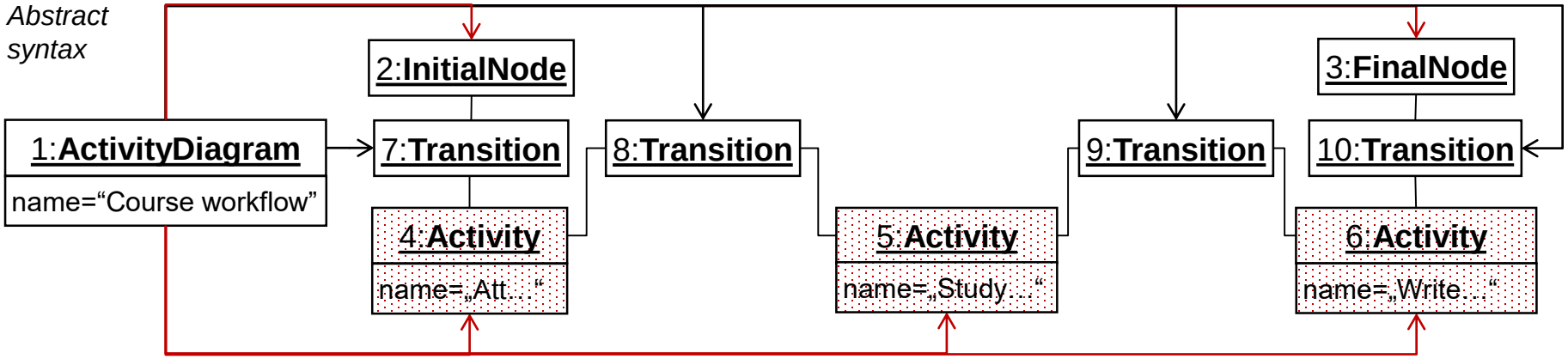
Changes:

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

Metamodel



Model



Validation errors:

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

Example 8/9

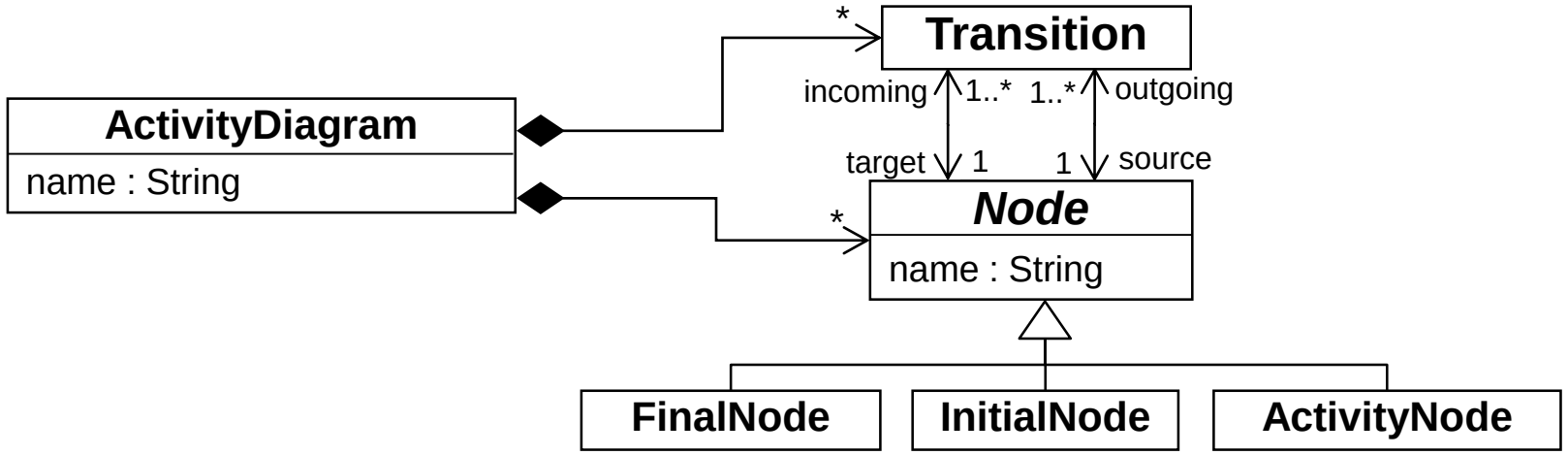
How to keep metamodels evolvable when models already exist

- **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

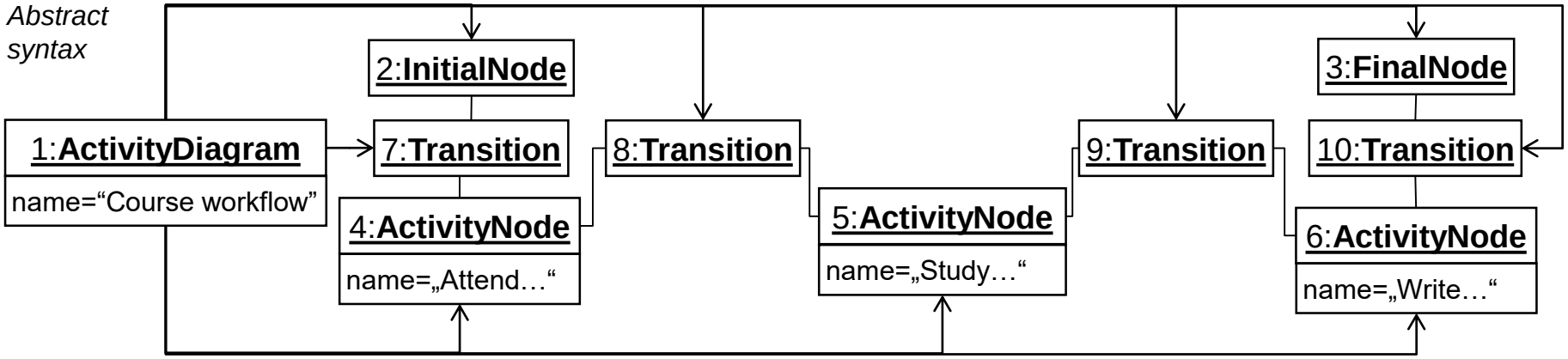
Example 9/9

Adapted model for new metamodel version

Metamodel



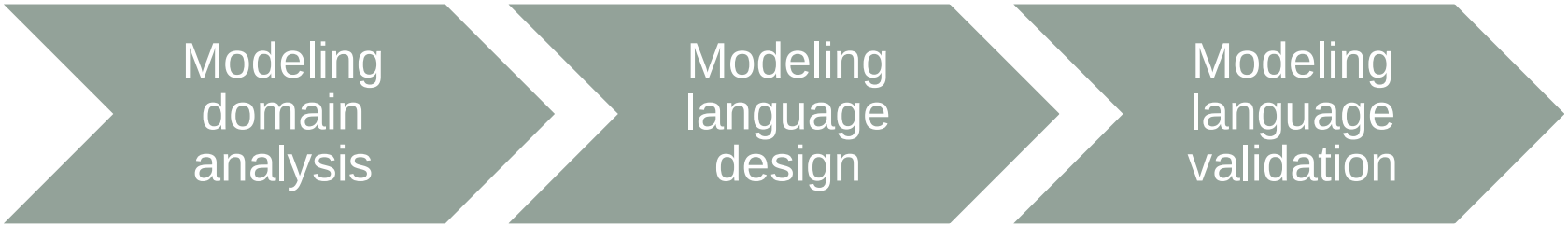
Model



More on this topic in Chapter 10!

Metamodel development process

Incremental and Iterative



Identify purpose, realization, and content of the modeling language

Sketch reference modeling examples

Formalize modeling language by defining a metamodel

Formalize modeling constraints using OCL

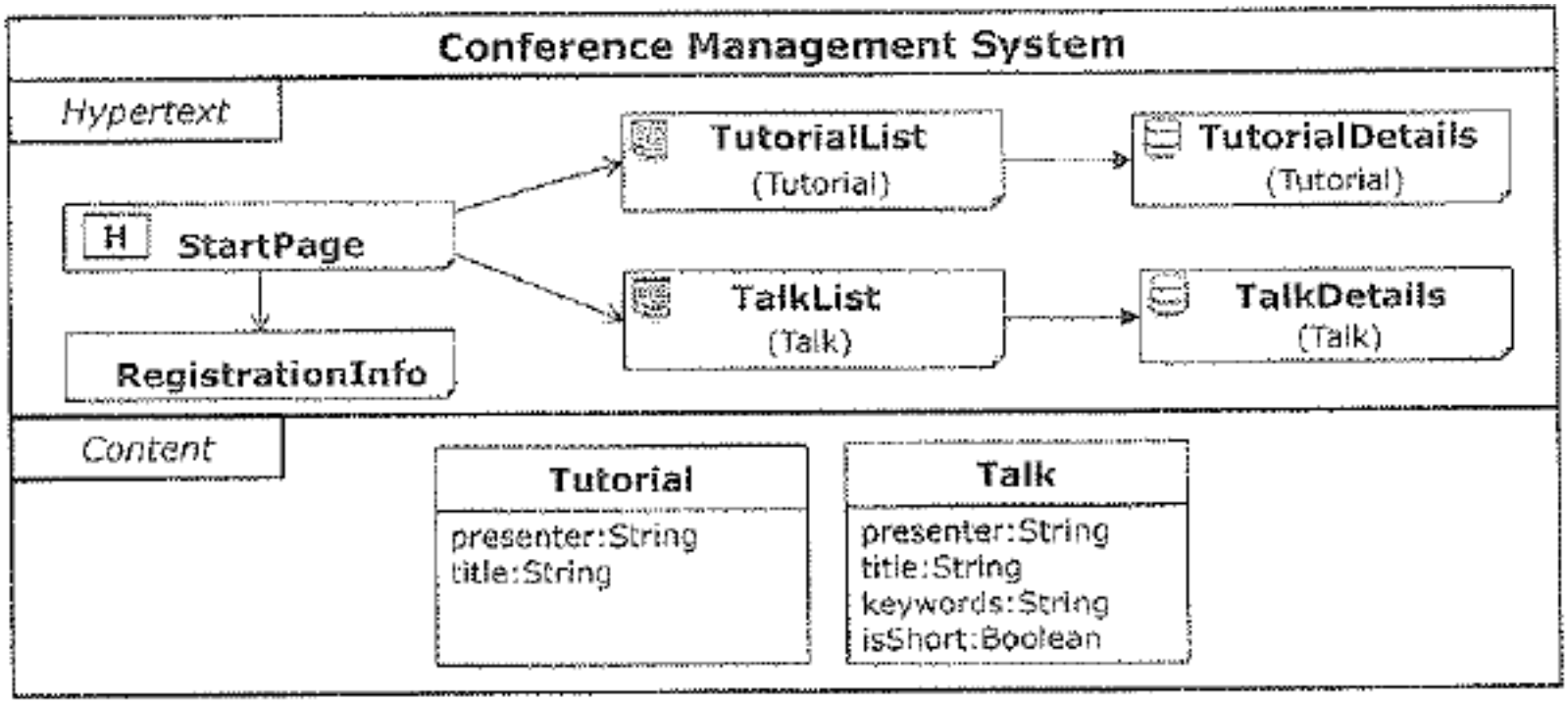
Instantiate metamodel by modeling reference models

Collect feedback for next iteration

DSML Example: SWML

- SWML: Simple Web Modeling Language
- STEP 1: MODELING DOMAIN ANALYSIS
- Language purpose: modeling the content layer and the hypertext layer of Web Applications
  - The content layer defines the schema of persistent data
  - The hypertext layer presentes the data to the user in the form of Web pages and defines the navigation between pages
- Language realization: support different SWML user types, a graphical syntax should be defined for easing communication with domain experts. A textual syntax should be provided for familiar developers
- Language content: a SWML consists of a content layer and a hypertext layer

DSML Example: SWML



Step 2 – Modeling Language Design: defining a metamodel

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*

Figure 7.4: Modeling concept table for sWML.

Step 2 – Modeling Language Design

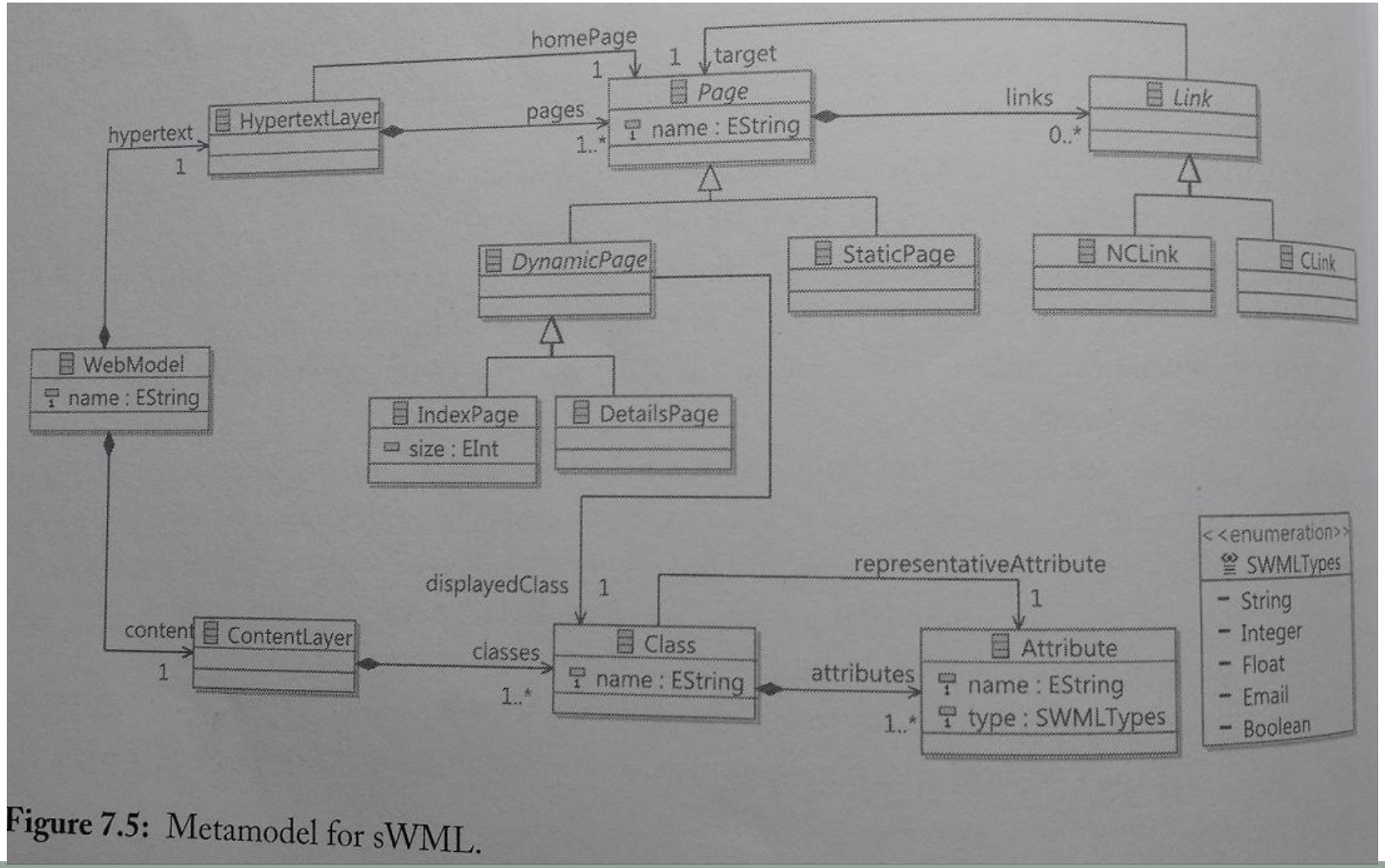


Figure 7.5: Metamodel for sWML.

Step 2 – Modeling Language Design - constraints in OCL

Rule 1: A *Class* must have a unique name within the *Content Layer* to avoid name clashes.

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

Rule 2: The representative *Attribute* of a *Class* must be taken out of the set of *Attributes* contained by the *Class*.

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

Rule 3: A page must not contain a non-contextual link to itself, because navigating such a link would result in exactly the same page as shown before the navigation.

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


Step 3 – Modeling Language Validation

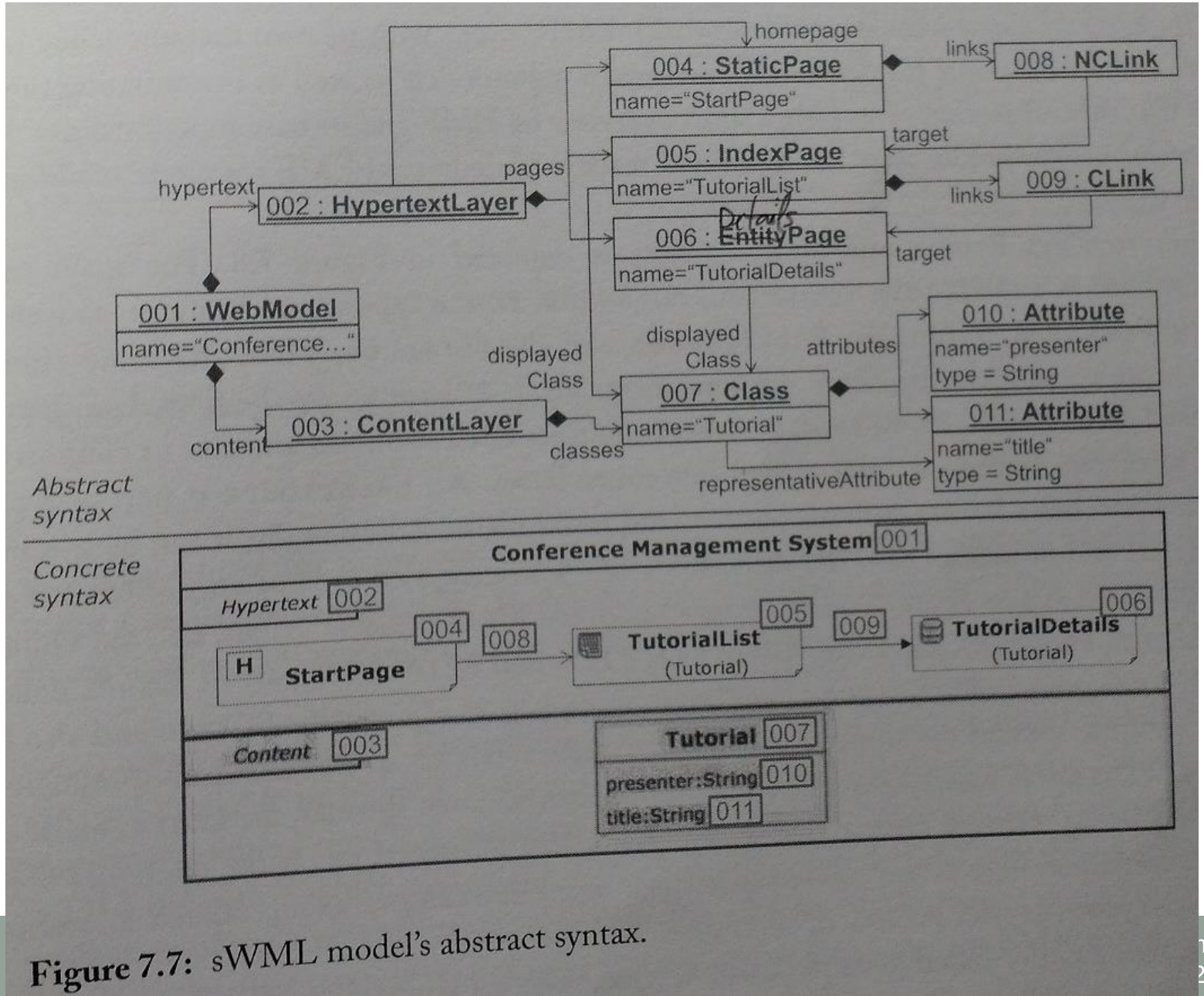
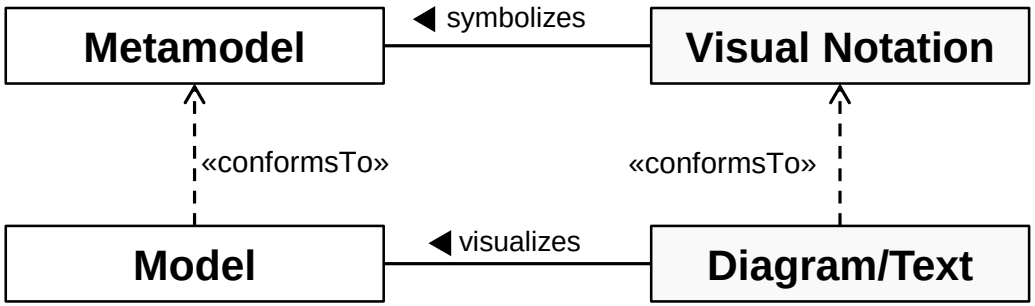


Figure 7.7: sWML model's abstract syntax.

GRAPHICAL CONCRETE SYNTAX

Introduction

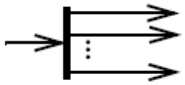
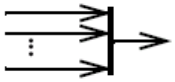
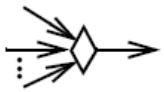
- The **visual notation** of a model language is referred as **concrete syntax**
- **Formal definition** of concrete syntax allows for **automated generation** of editors
- Several approaches and frameworks available for defining concrete syntax for model languages



Introduction

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

- Example – Excerpt from the UML-Standard

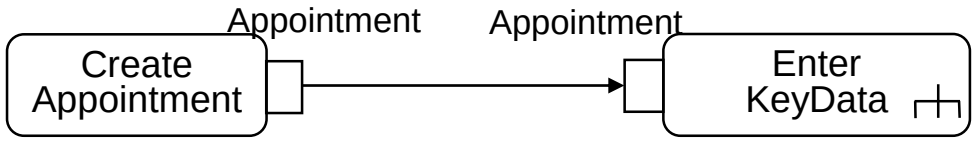
Node Type	*Notation*	*Reference*
ForkNode		See ForkNode (from IntermediateActivities) on page -404.
InitialNode		See InitialNode (from BasicActivities) on page -406.
JoinNode		See "JoinNode (from CompleteActivities, Interme-diateActivities)" on page 411.
MergeNode		See "MergeNode (from IntermediateActivities)" on page 416.

Introduction

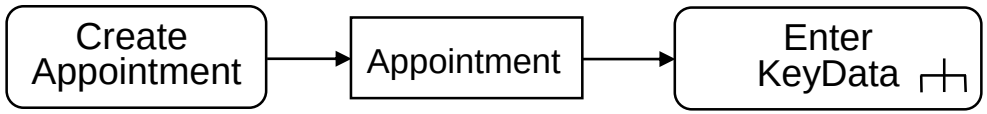
- Concrete syntax **improves** the **readability** of models
  - Abstract syntax not intended for humans!
- **One** abstract syntax may have **multiple** concrete ones
  - Including textual and/or graphical
  - Mixing textual and graphical notations still a challenge!

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

Notation alternative 1:



Notation alternative 2:



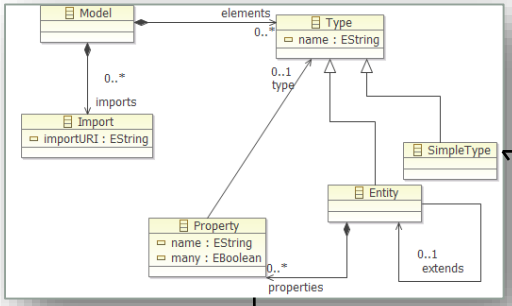
Notation alternative 3:

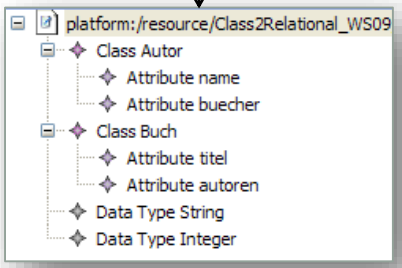
```
Appointment a;
a = new Appointment;
EnterKeyData (a);
```


Introduction

Concrete Syntaxes in Eclipse

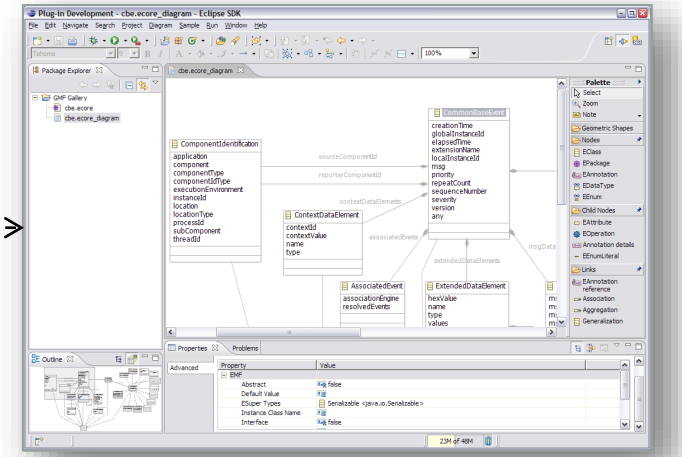
Ecore-based Metamodels



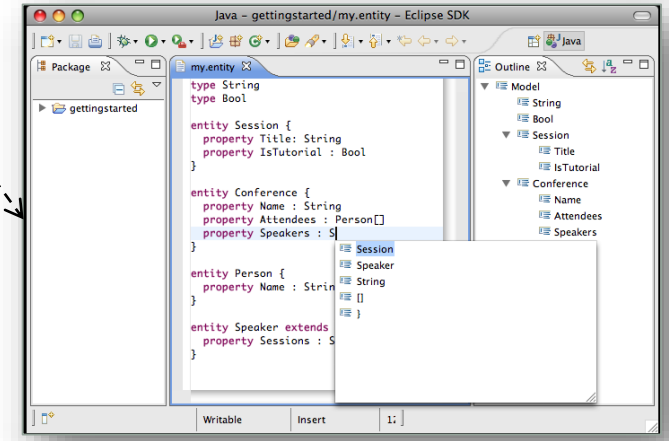


Generic tree-based EMF Editor

Graphical Concrete Syntax



Textual Concrete Syntax

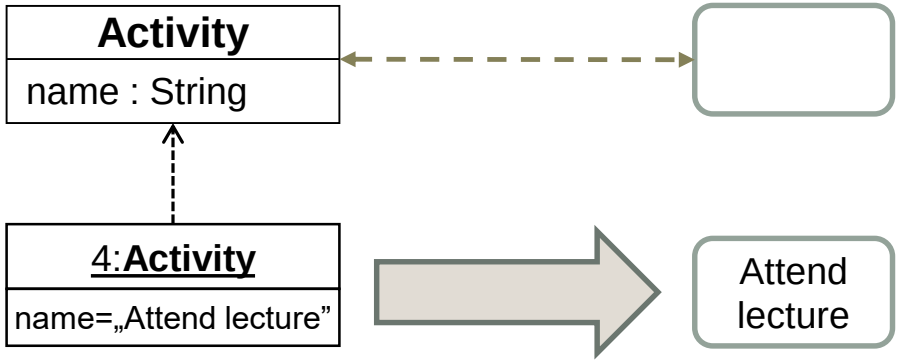


Anatomy of Graphical Concrete Syntaxes

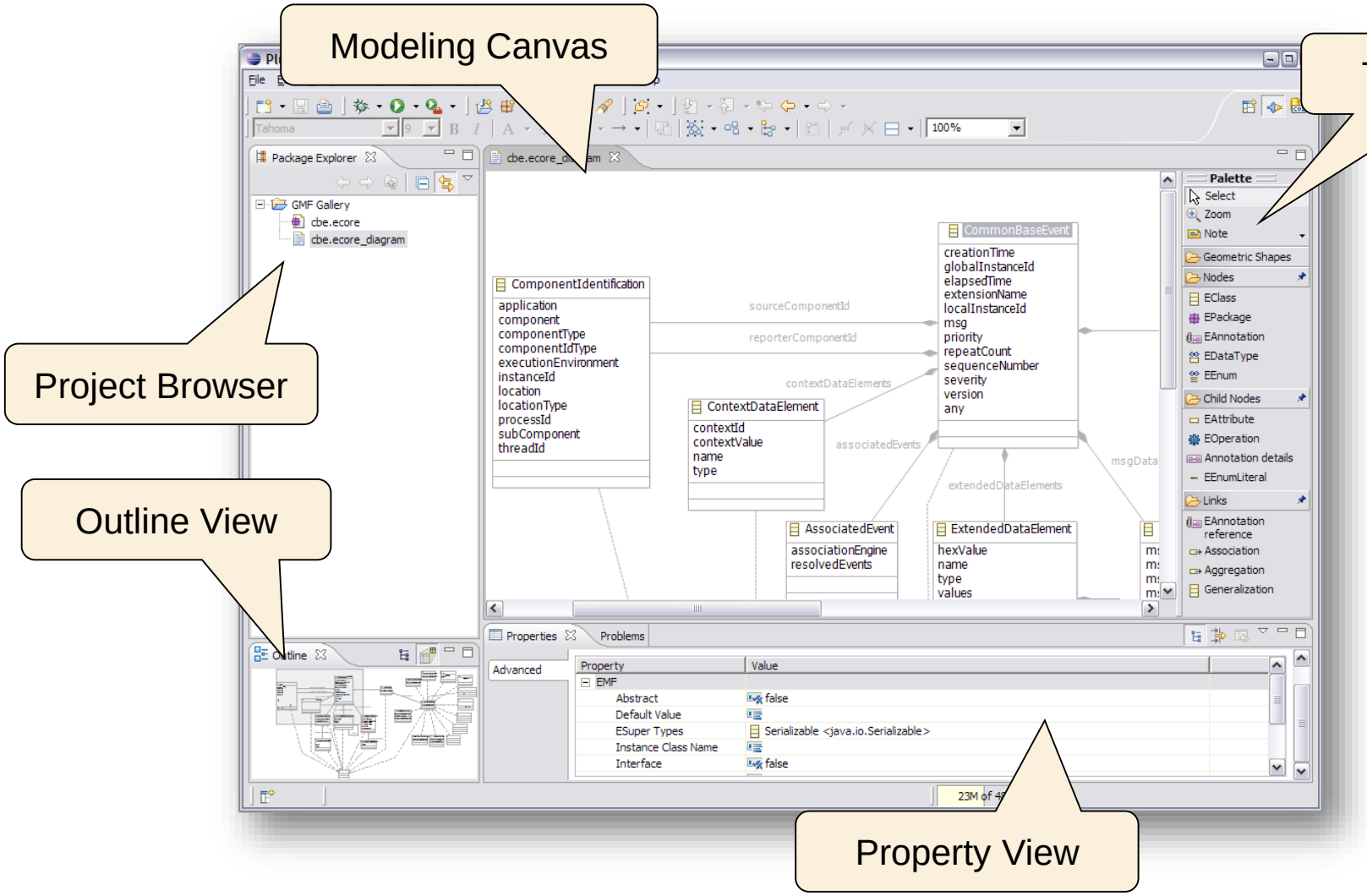
- A Graphical Concrete Syntax (GCS) consists of
  - **graphical symbols,**
    - e.g., rectangles, circles, …

  - **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

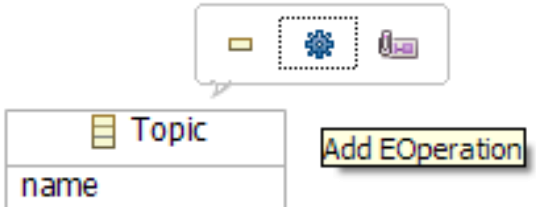


Anatomy of Graphical Modeling Editors

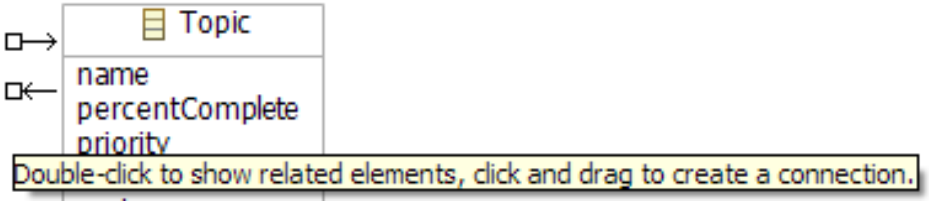


Features of Graphical Modeling Editors

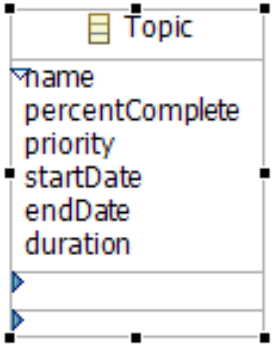
Action Bars:



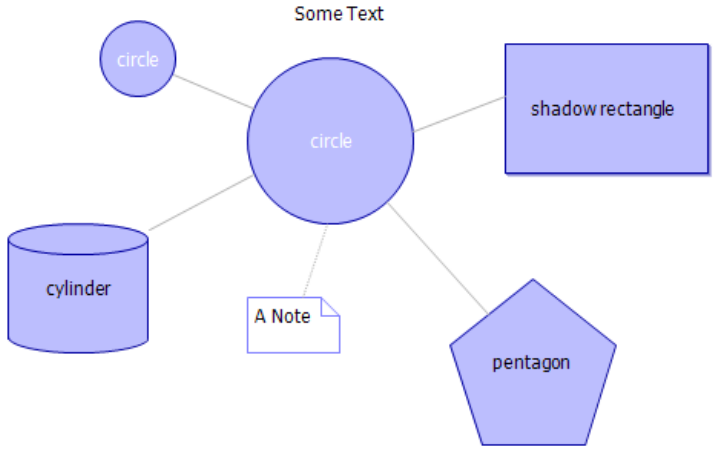
Connection Handles:



Collapsed Compartments:

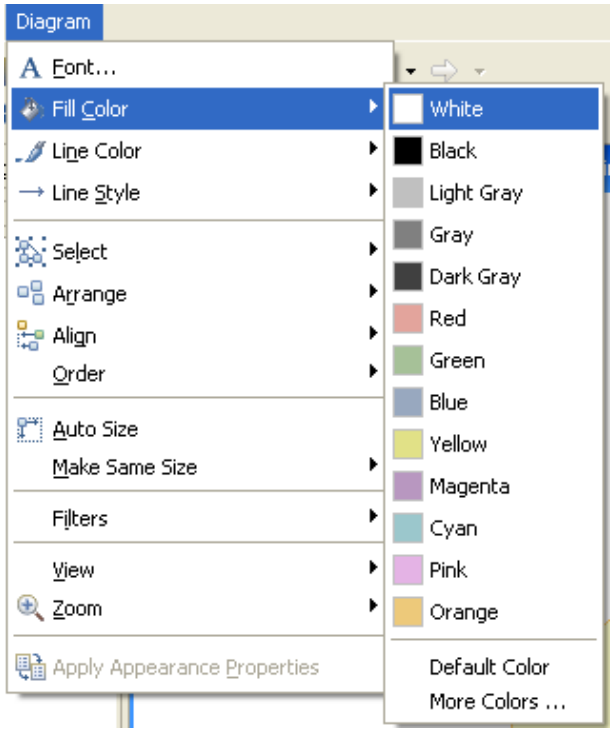


Geometrical Shapes:



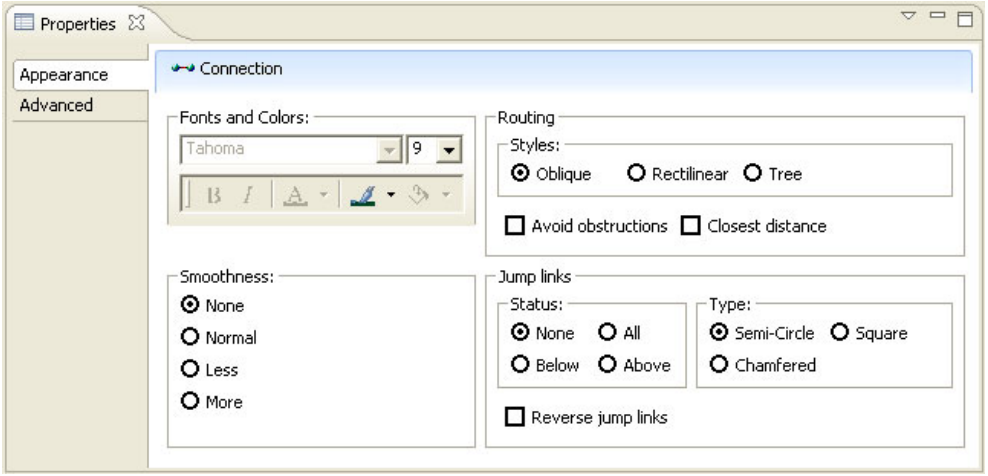
Features of Graphical Modeling Editors

Actions:

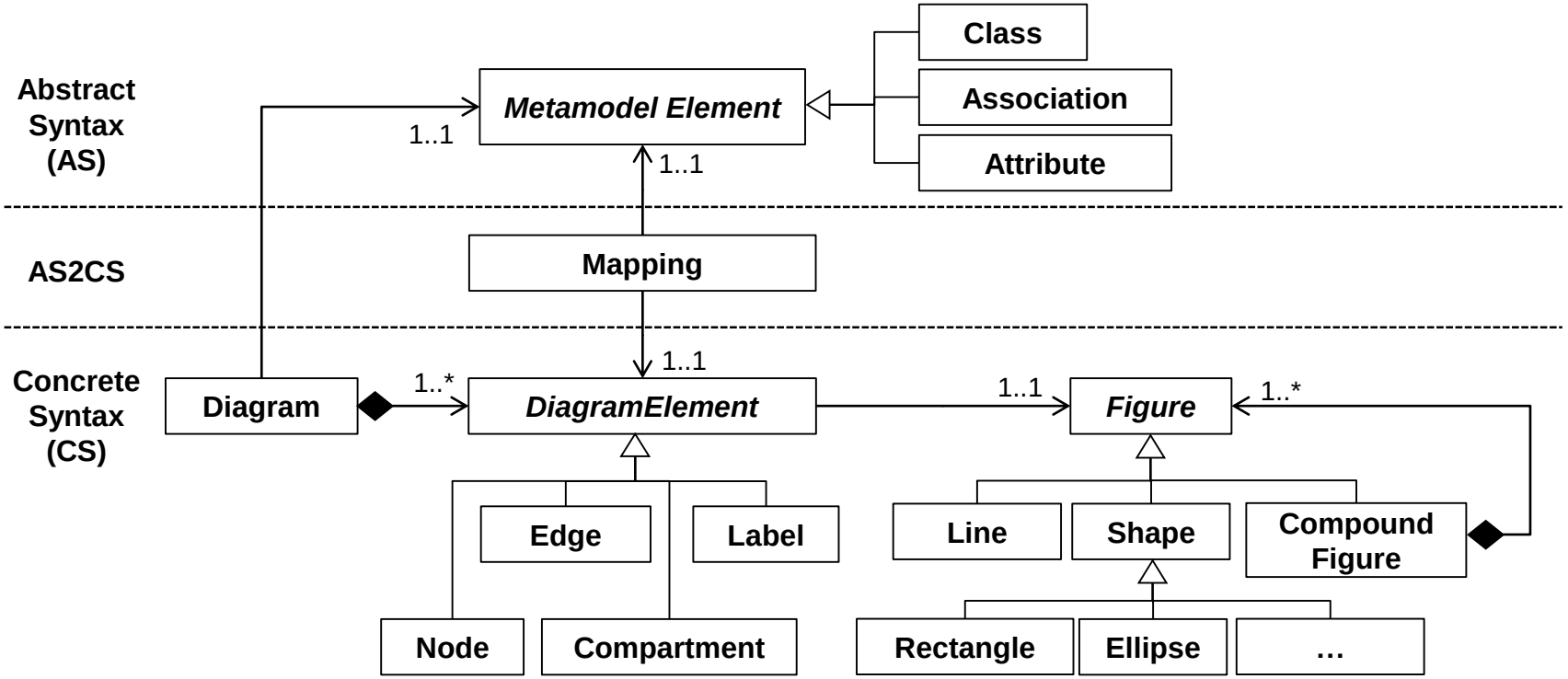


Toolbar:

Properties View:

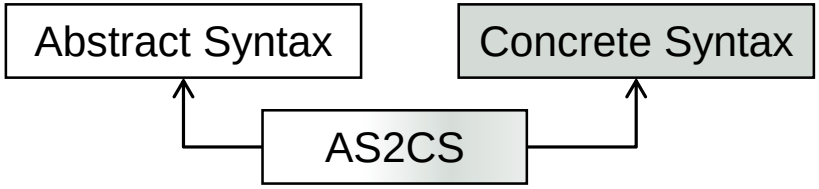


Generic Metamodel for GCS

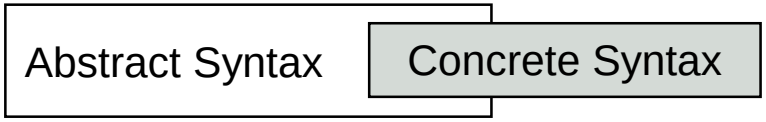


GCS Approaches

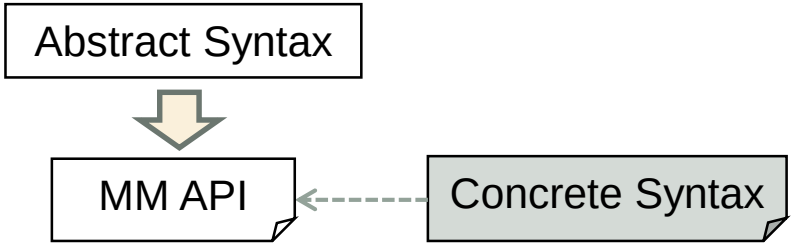
- **Mapping-based**
  - Explicit mapping model between abstract syntax, i.e., the metamodel, and concrete syntax



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



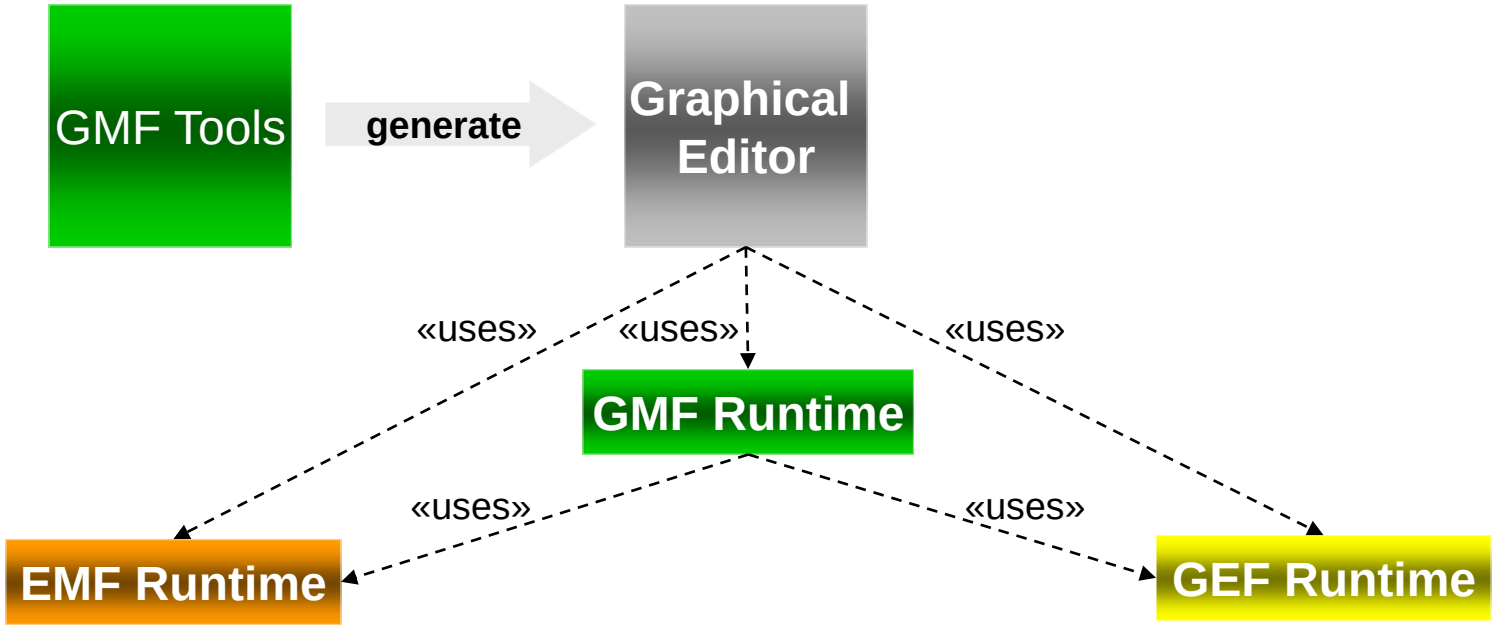
- **API-based**
  - Concrete syntax is described by a programming language using a dedicated API for graphical modeling editors



Mapping-based Approach: GMF

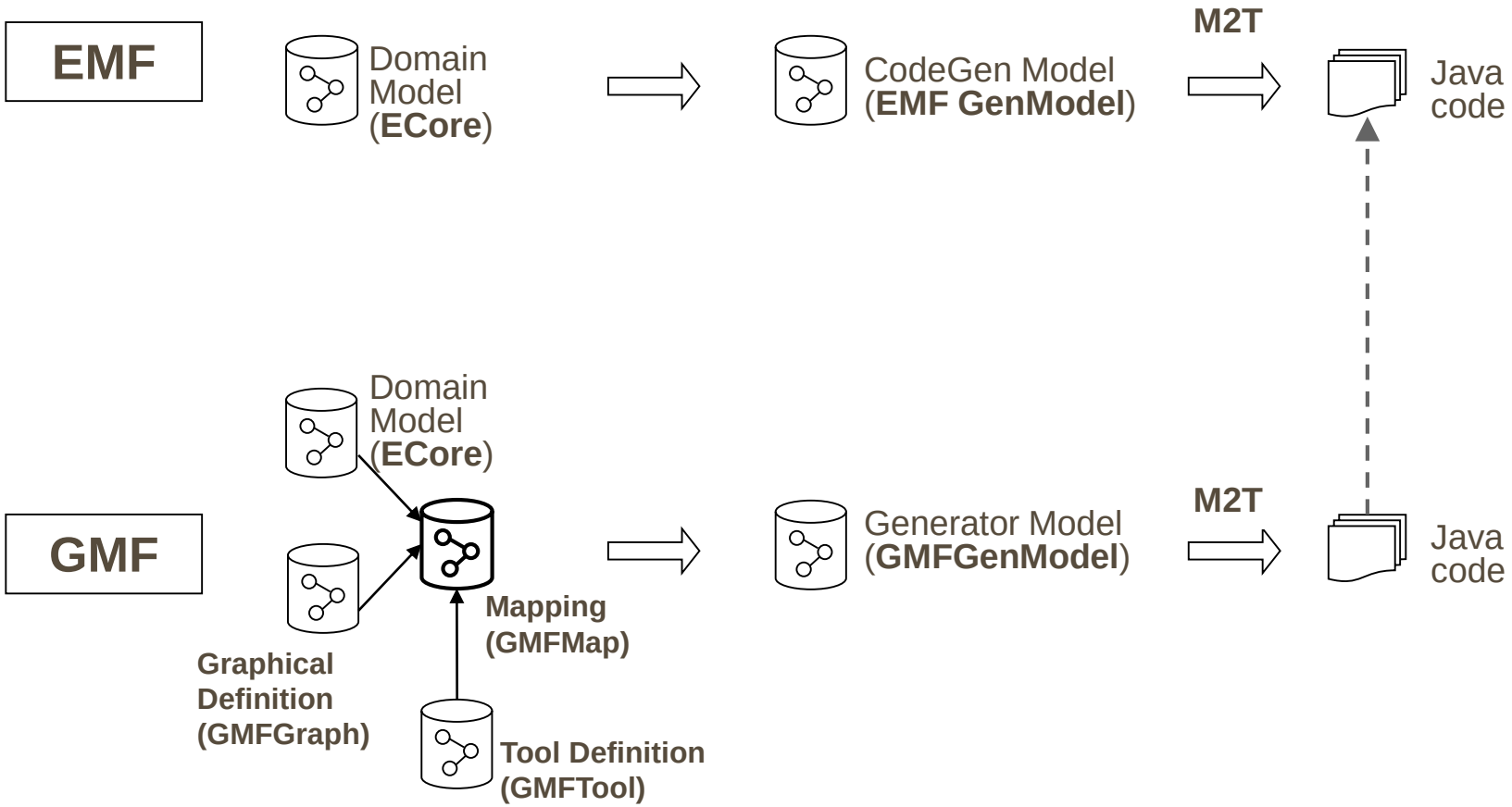
Basic Architecture of GMF

- *"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

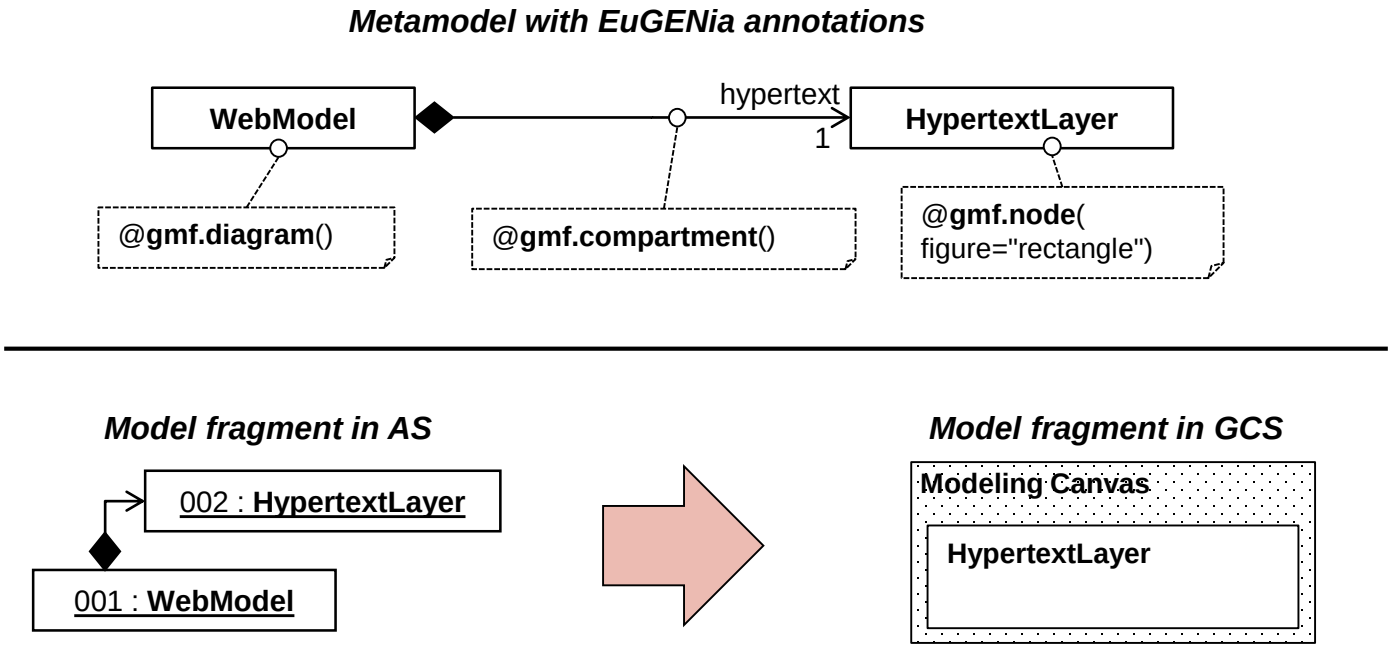
Be aware:
Application of MDE techniques for developing MDE tools!

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

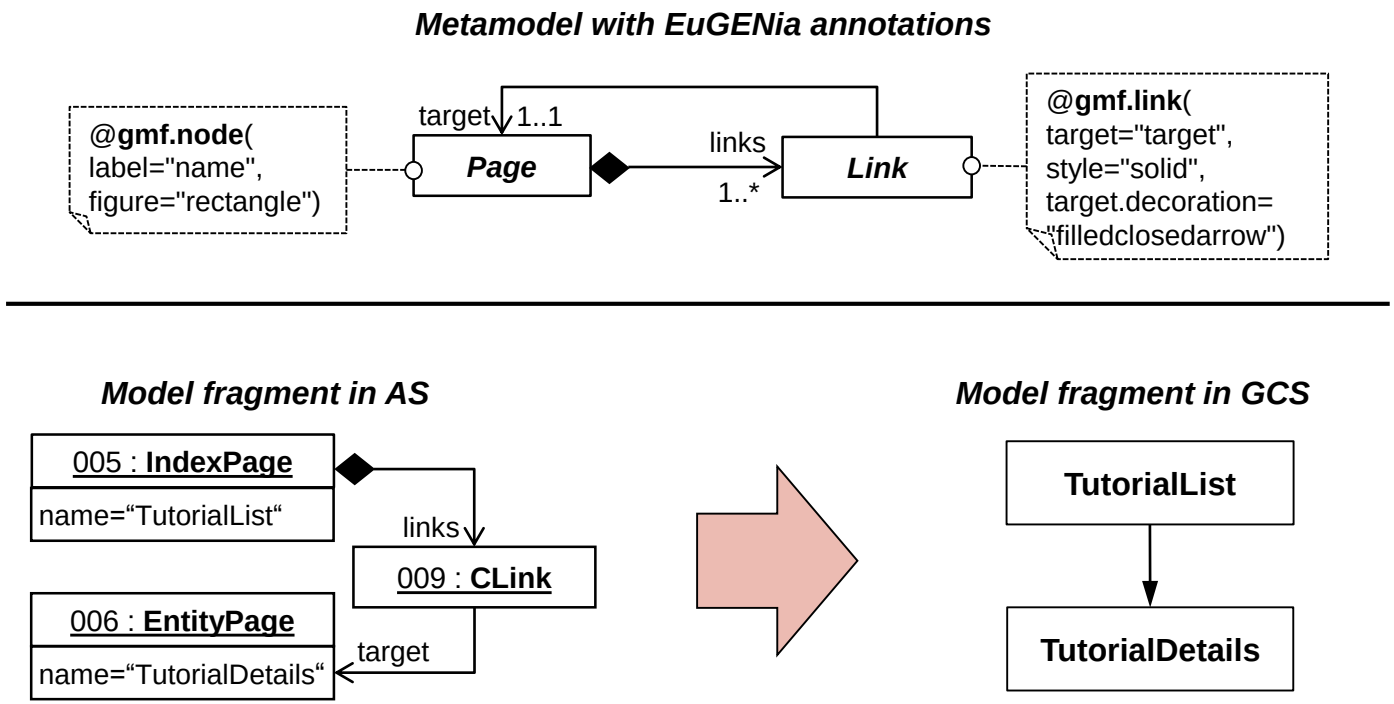
Eugenia Example #1

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



Eugenia Example #2

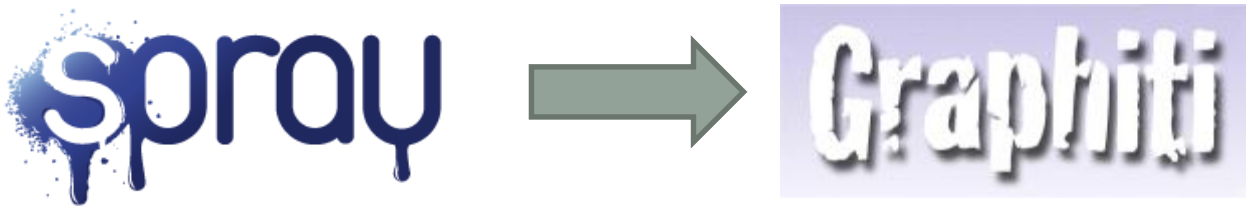
- *Pages* should be displayed as **rectangles** and *Links* should be represented by a directed **arrow** between the rectangles



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)

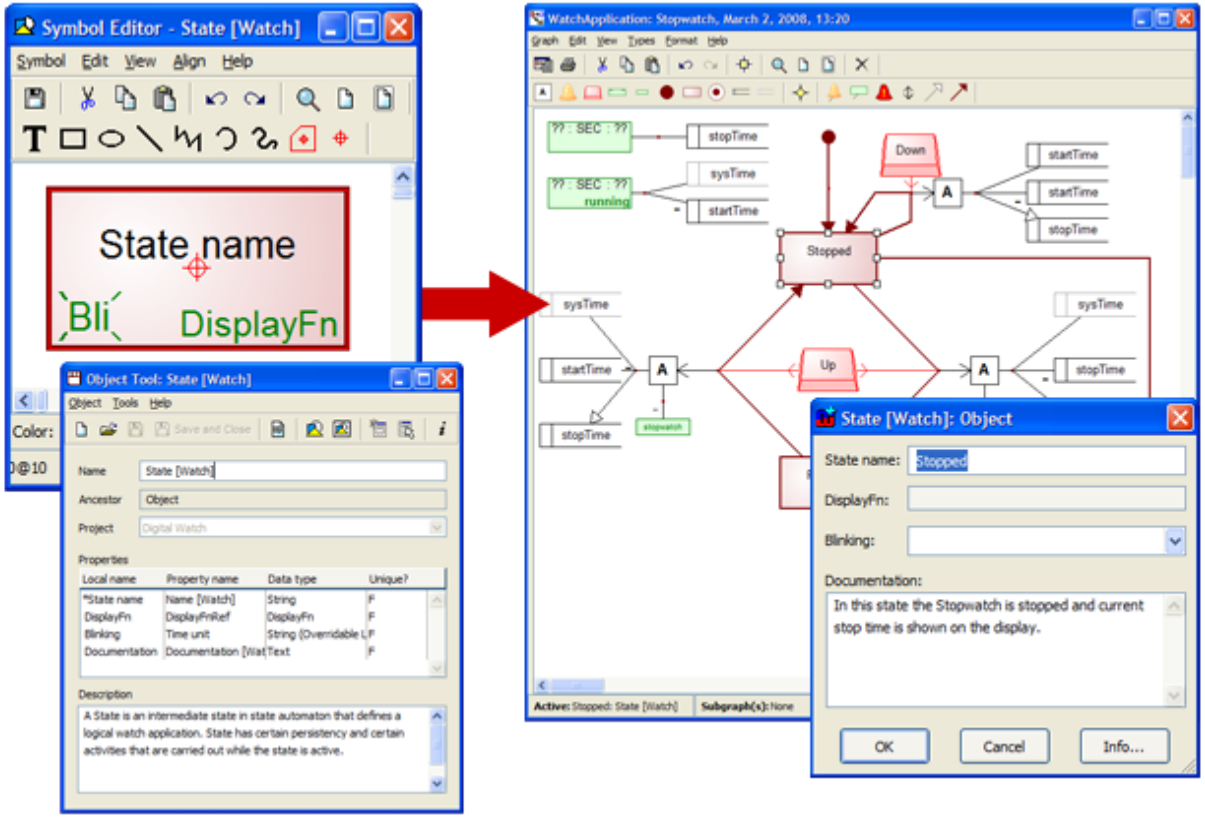
- DSL on top of Graphiti: Spray



Other Approaches outside Eclipse

MetaEdit+

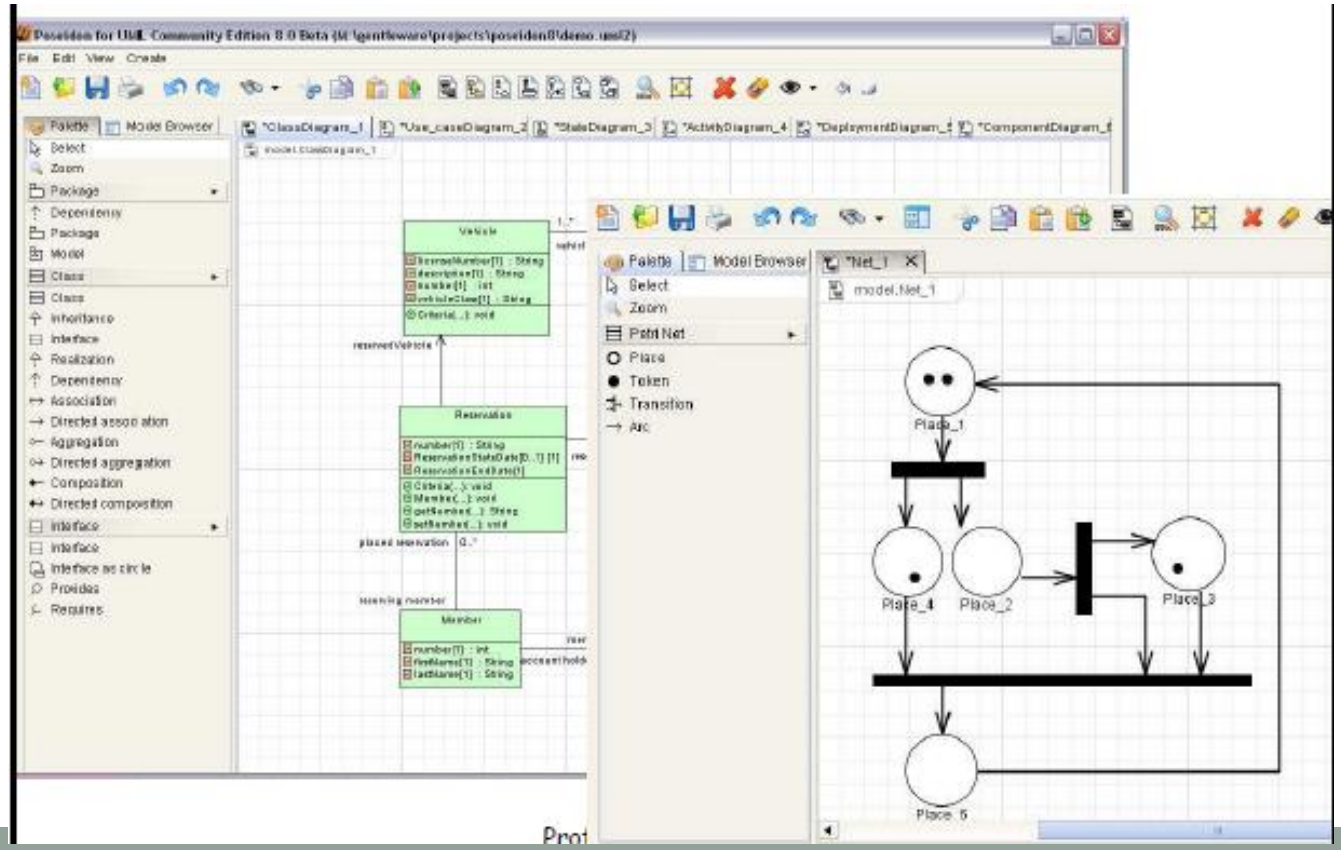
- Metamodeling tool outside Eclipse (commerical product)
- **Graphical specification of figures** in graphical editor
- Special tags to specify labels in the figures by querying the models



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

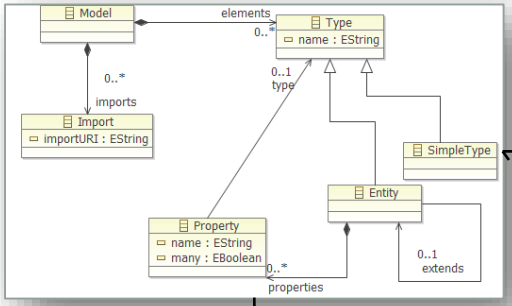
Textual Modeling Languages

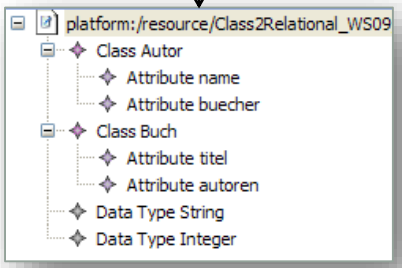
- **Textual languages** have **specific strengths** compared to **graphical languages**
  - Scalability, pretty-printing, …
- **Compact and expressive syntax**
  - Productivity for experienced users
  - Guidance by IDE support softens learning curve
- **Configuration management/versioning**
  - Concurrent 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

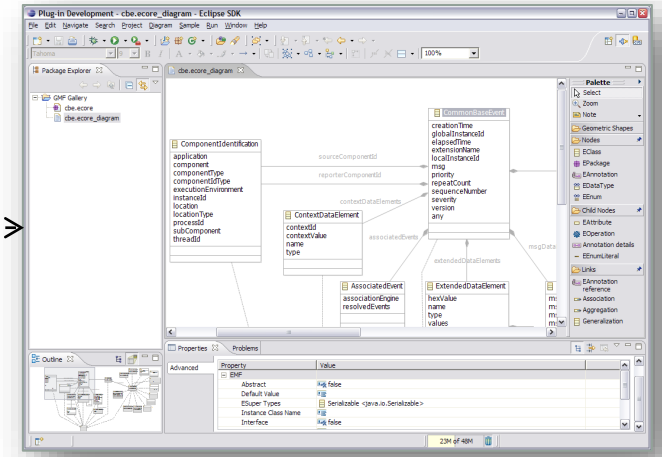
Ecore-based Metamodels



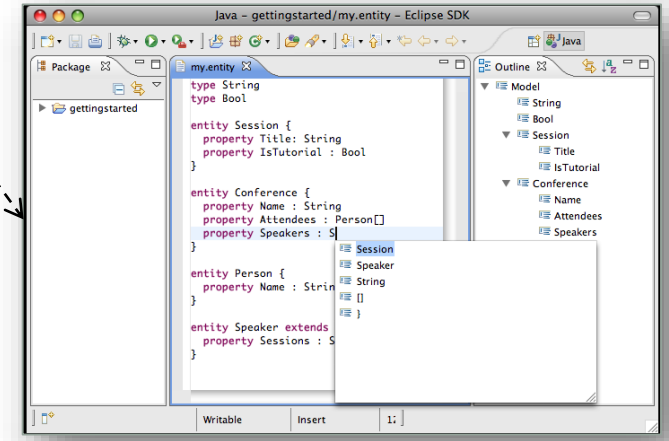


Generic tree-based EMF Editor

Graphical Concrete Syntax

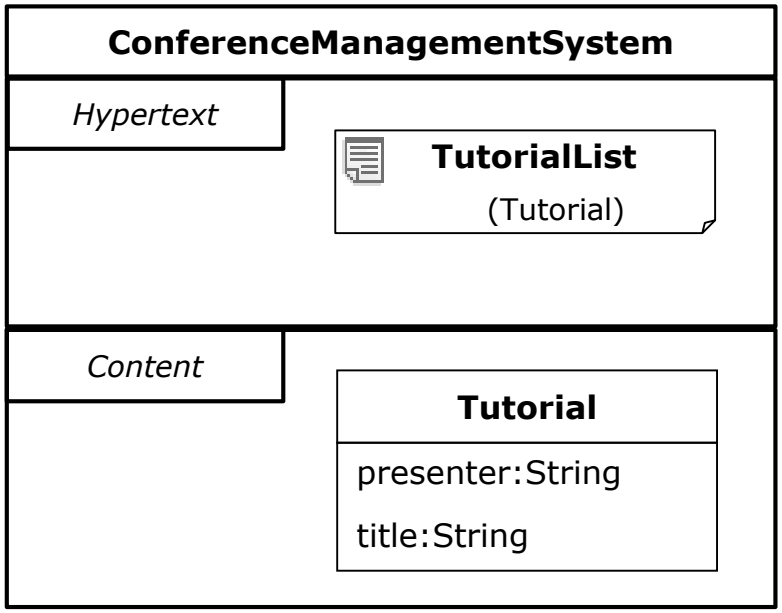


Textual Concrete Syntax



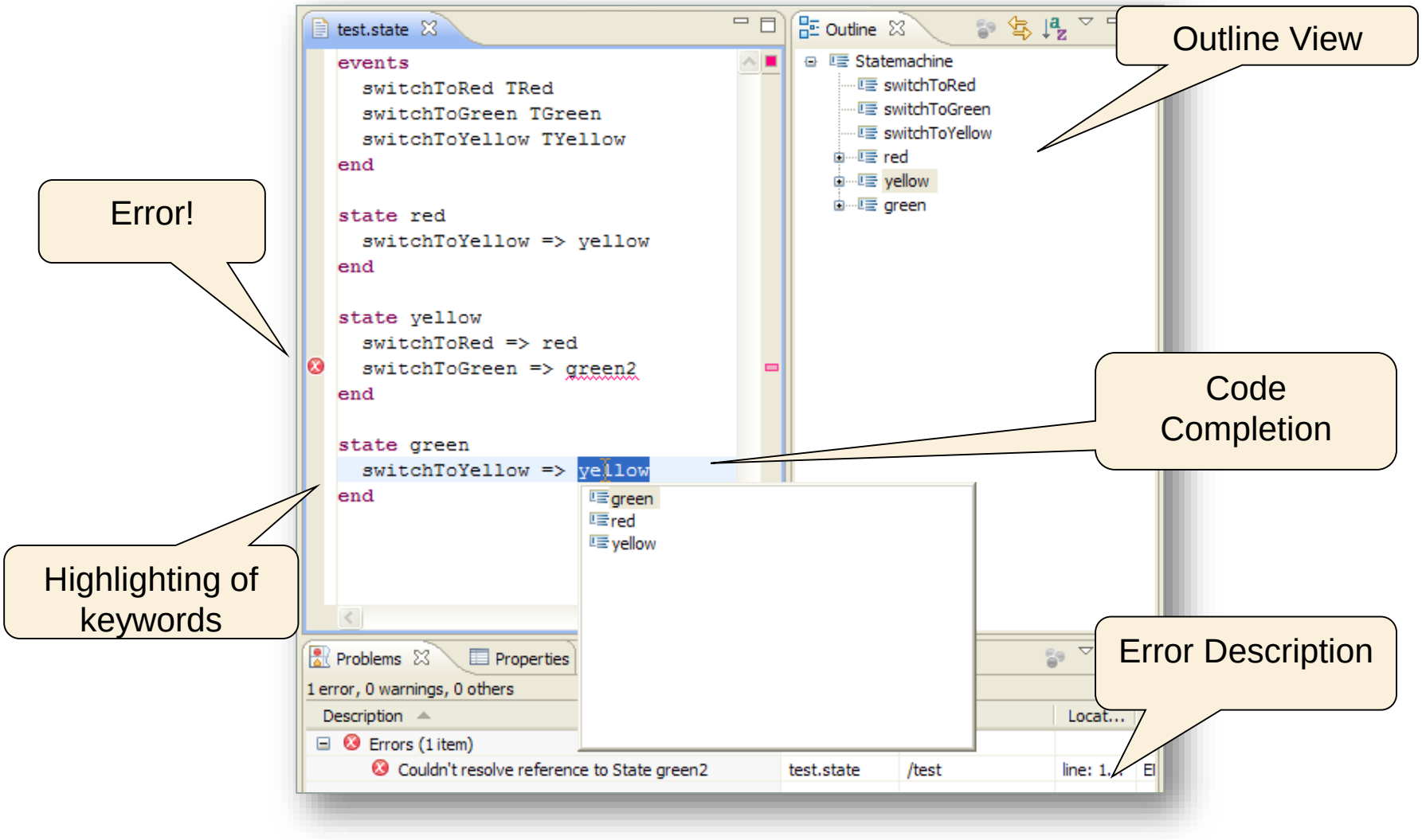
Every GCS is transformable to a TCS

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**

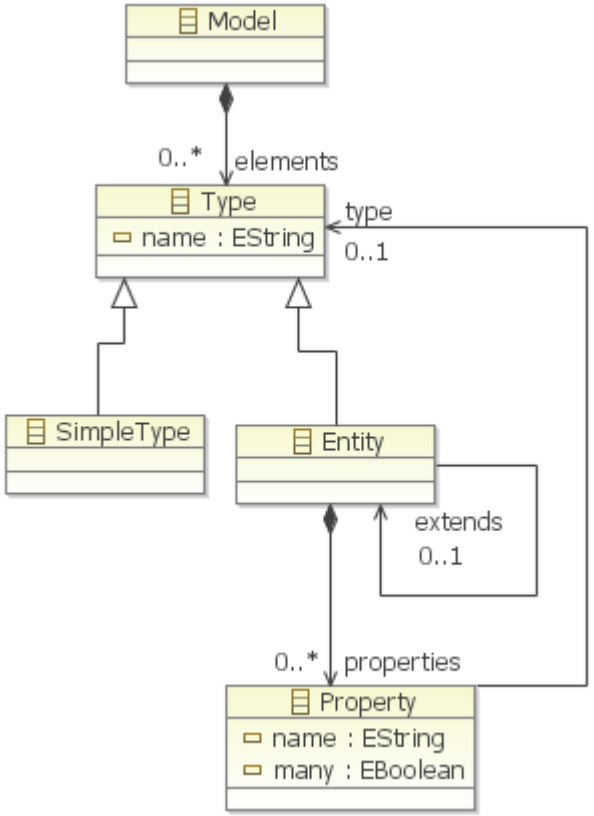
```
Model := Type*;

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 initial Xtext grammar!

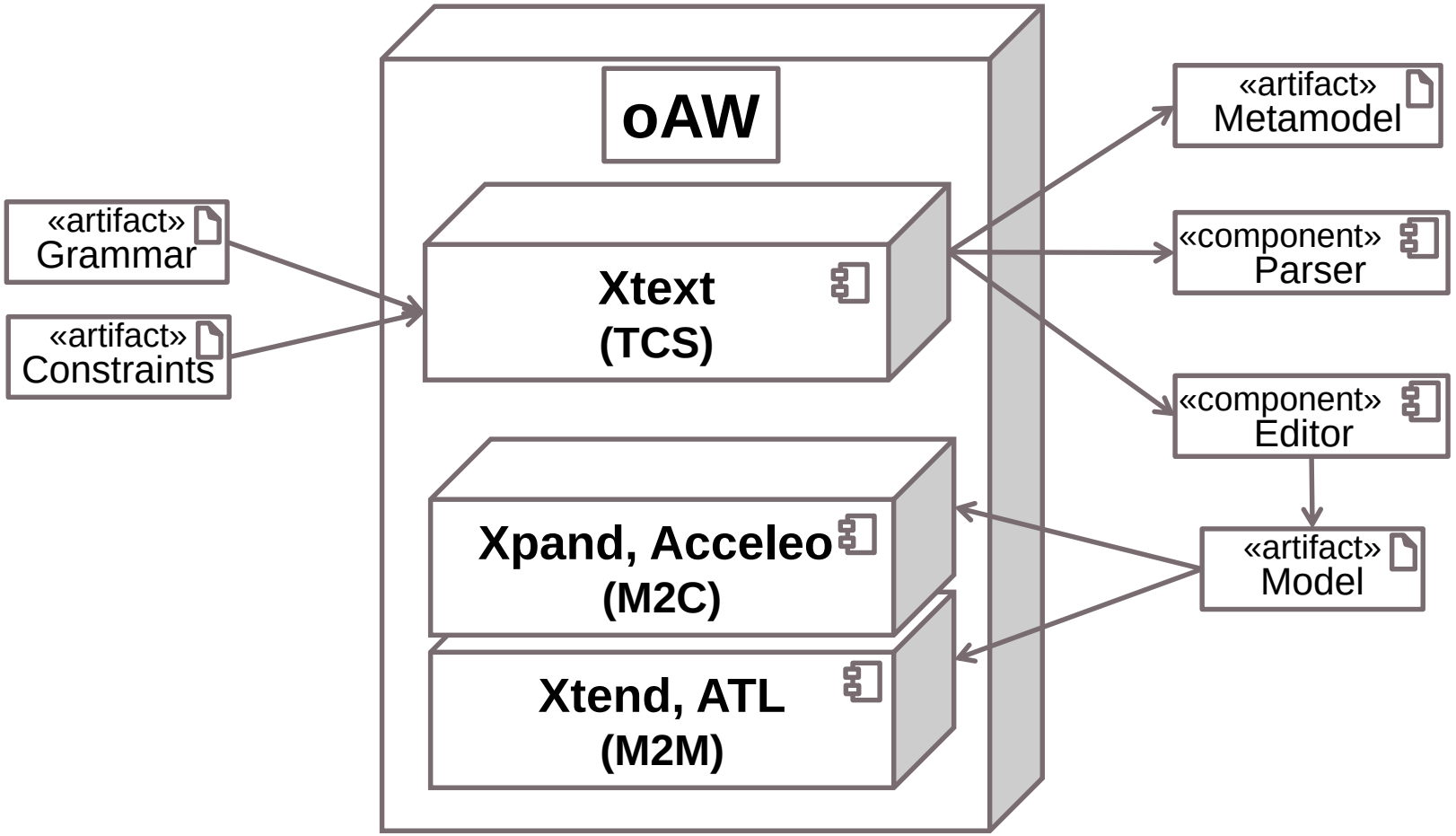
Xtext

Introduction

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

Xtext

Introduction



Xtext

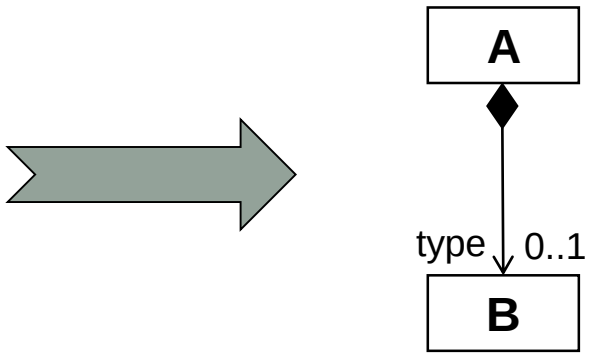
Grammar

- **Xtext** grammar **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

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**

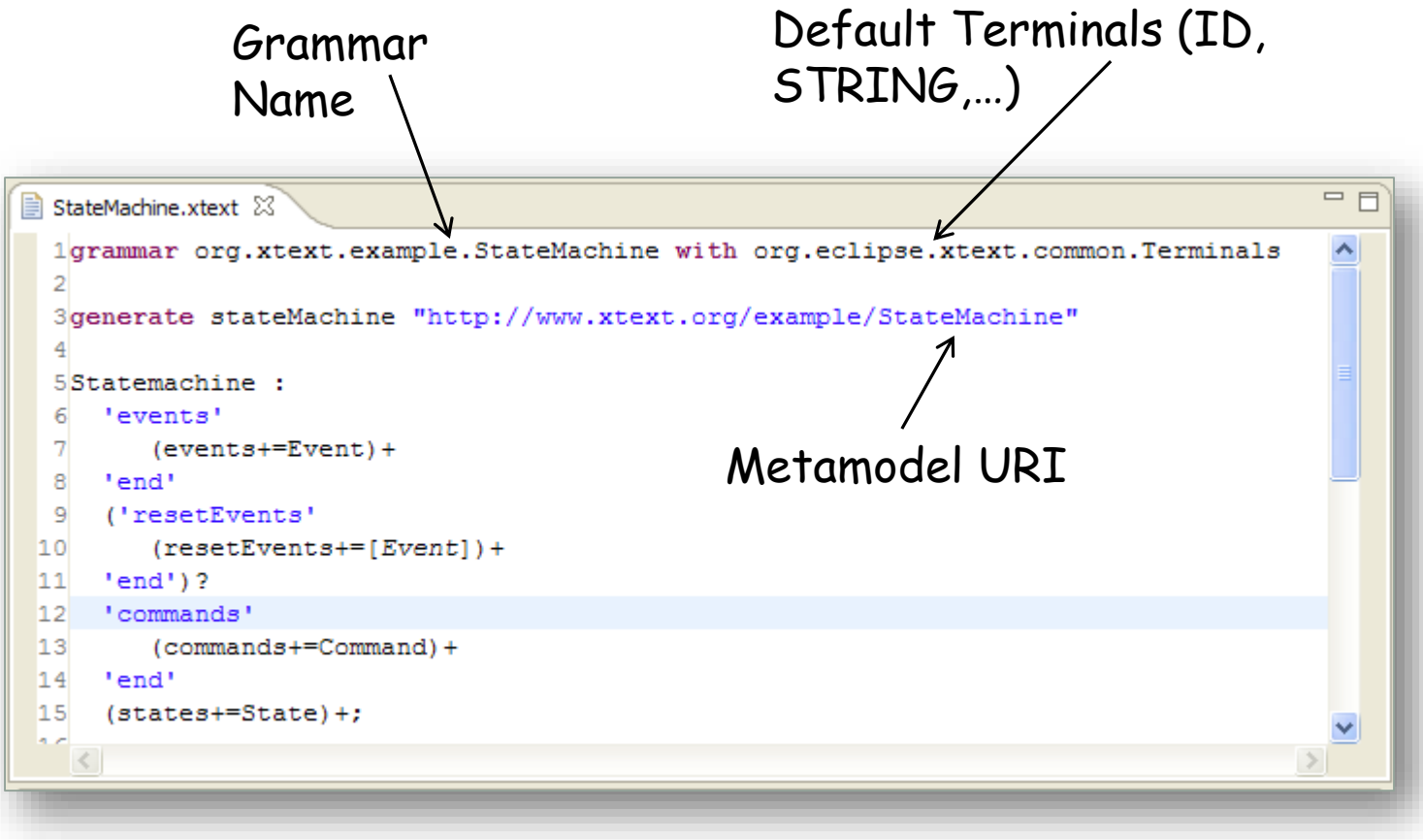
```
enum ChangeKind :
    ADD | MOVE | REMOVE
  ;
```

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


Xtext

Tooling

- **Xtext Grammar Definition**



Xtext

Tooling

- **Xtext Grammar Definition for State Machines**

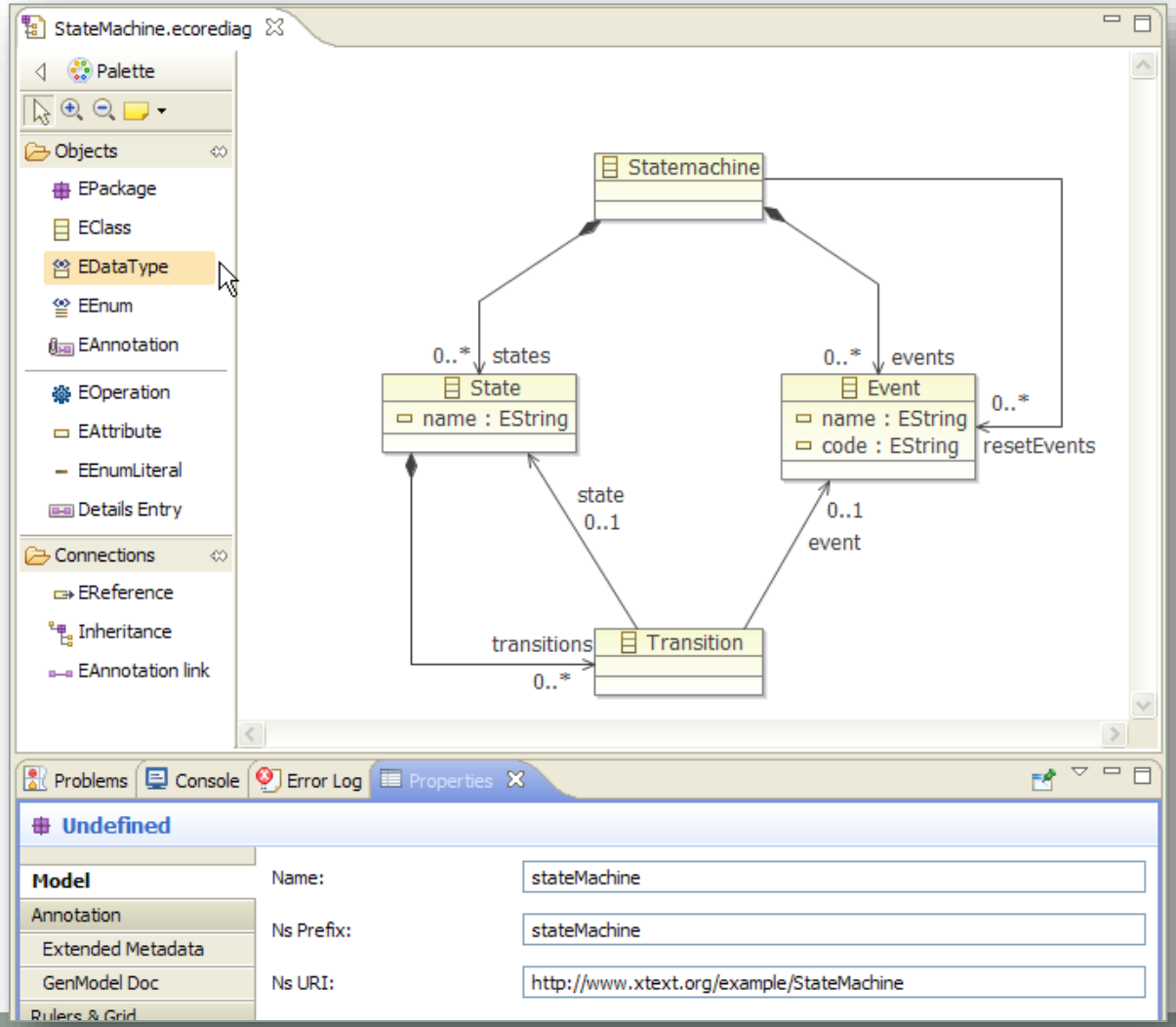
StateMachine.xtext

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


Xtext

Tooling

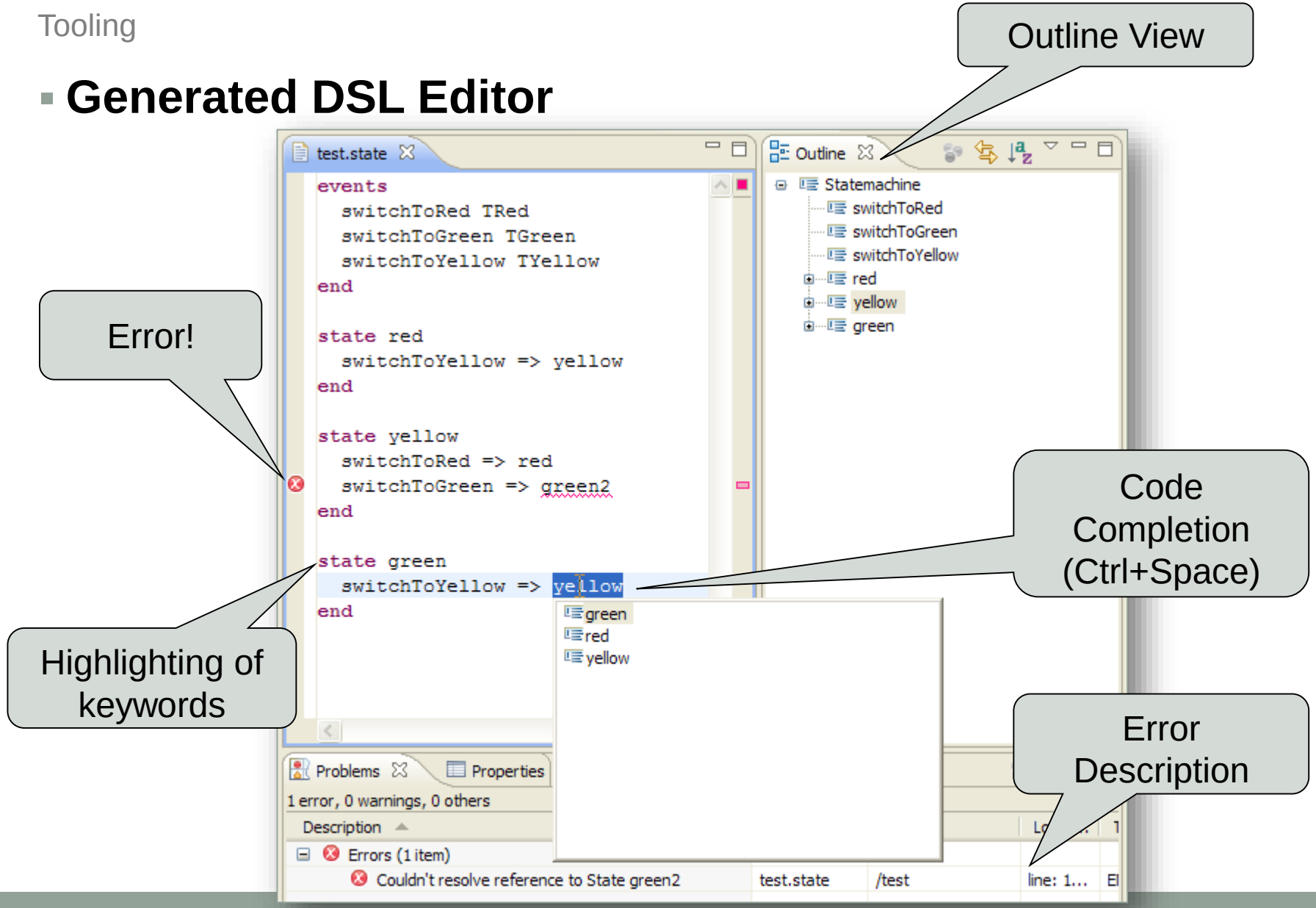
- **Automatically generated Ecore-based Metamodel**



Xtext

Tooling

- **Generated DSL Editor**



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