Advanced Software Construction Techniques

2022/2023

University of Porto
Faculty of Engineering
Department of Informatics Engineering
Master in Informatics and Computing Engineering

Jácome Cunha (jacome@fe.up.pt)

Class 1
Presentation &
Introduction to MDSE

Settings

What Will You Learn?

- Model-driven engineering
    - Main topic of the course
- Re-engineering
- Model repair
    - Topic by **Nuno Macedo** (FEUP)
- Metaprogramming
    - Topic by **João Bispo** (FEUP)
- Low-code
    - Topic by probably OutSystems
- Parser combinators
    - Topic by **João Saraiva** (UMinho)
- Debugging and program repair (if time allows it)

How Will You Learn?

- **Flipped classroom**
  - **Note: you need to dedicate 123 hours this course** (apart from the classes)
  - **Divided by 13 weeks, > 9 hours every week!**
- Each week **you** will study a given topic
- During the first part of the following class we will have time to discuss doubts about the week's topic
- The rest of the class will be used to work in the project/exercises
- Course project-oriented

Evaluation

- C - participation in the classroom (10%)
- T - individual test (20%)
- P - group project (70%)

Abstract It Up

Abstraction

- **Abstraction** consists of the capability of finding the commonality in many different observations and thus generating a mental representation of the reality which is at the same time able to:
  - generalize specific features of real objects (generalization);
  - classify the objects into coherent clusters (classification); and
  - aggregate objects into more complex ones (aggregation).
- Natural behaviors of the human mind

From Abstraction to Models

- Abstraction is also widely applied in science and technology, where it is often referred to as modeling

- A **model** is a simplified or partial representation of reality

- Therefore, by definition, a model will never describe reality in its entirety

Models

- Models are recognized to implement at least two roles by applying abstraction:
    - **reduction** feature: models only reflect a (relevant) selection of the original's properties, so as to focus on the aspects of interest; and
    - **mapping** feature: models are based on an original individual, which is taken as a prototype of a category of individual and is abstracted and generalized to a model.

Models Everywhere

- “everything is a model” since nothing can be processed by the human mind without being “modeled”
- We all always create a mental model of reality
- Thus, you cannot avoid modeling
- When developing something, the developer must have in mind a model for their objective
- The model always exists, the only option is about its form: it may be mental (existing only in the designers’ heads) or explicit
- In this context of software, modeling is known as Model-Driven Software Engineering (MDSE)

The Basics

Algorithms + Data Structures = Programs

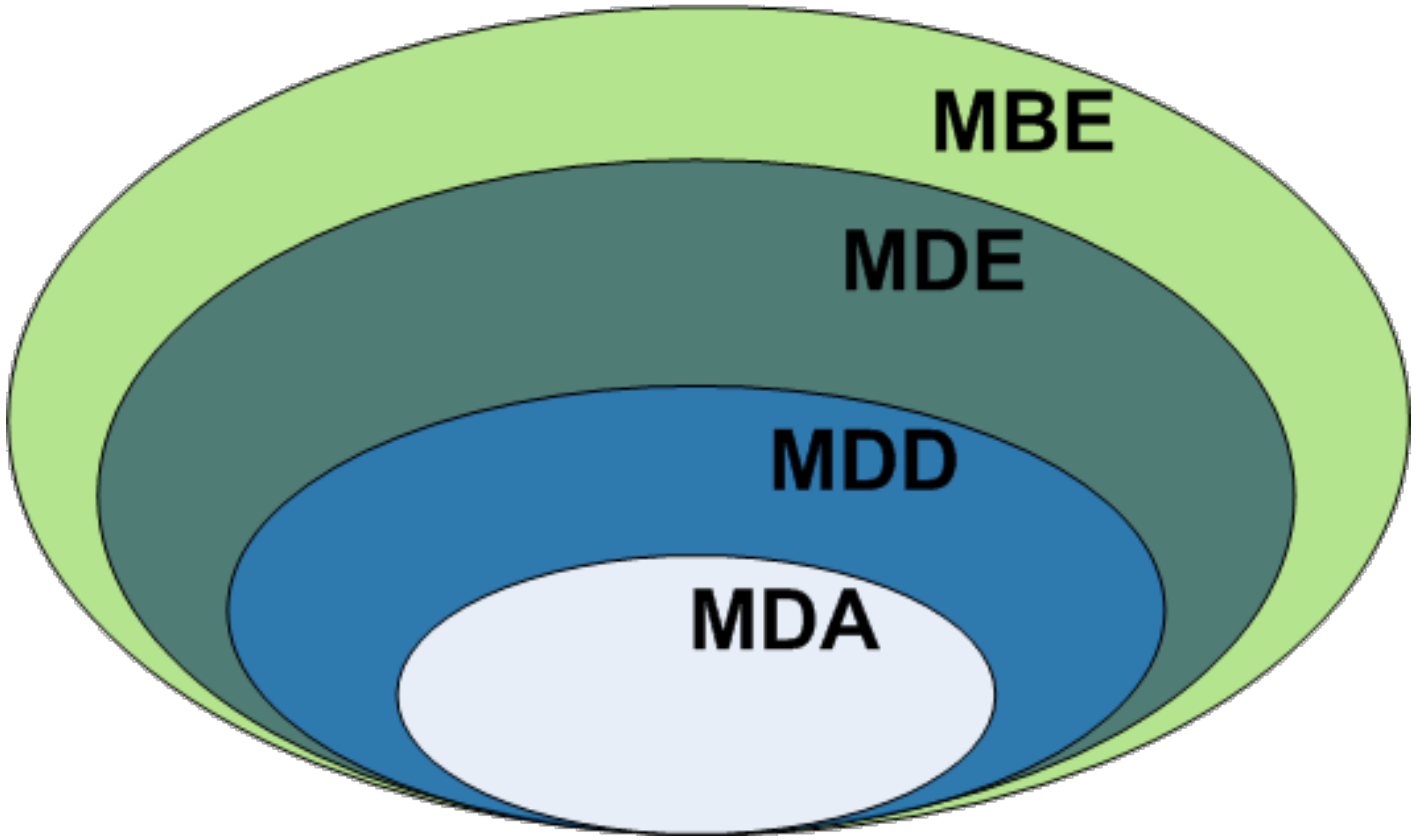
Models + Transformations = Software

MDSE Methodology

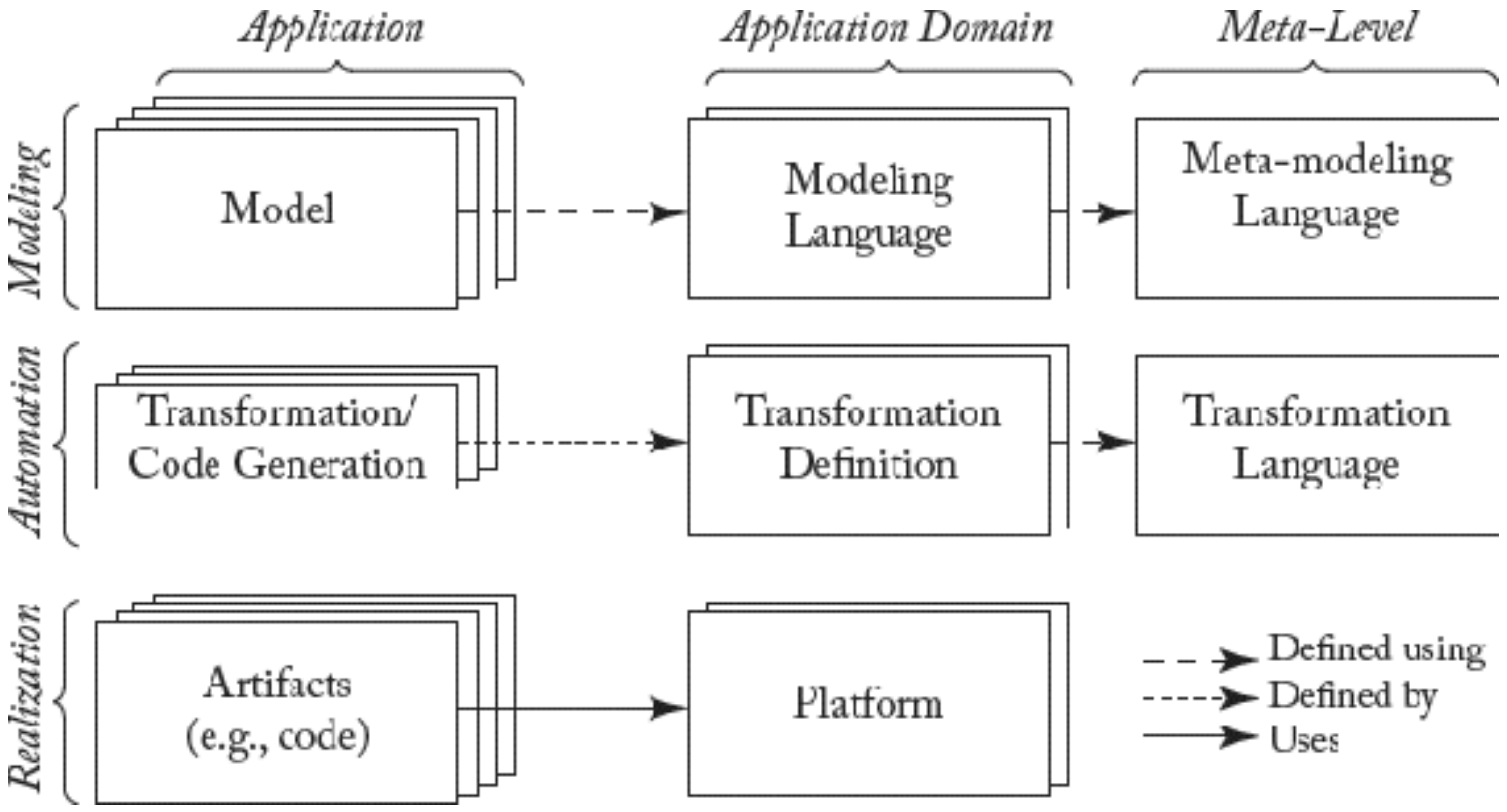
- Core concepts
    - Models, Transformations
- Notation
    - Many different *modeling languages* exists
- Process
    - Dependent on the kind of work to be done
- Tools
    - Software to manage the models and transformations

The Many Names

- Model-Driven Development (MDD)
  - models are the primary artifact of the development process
  - the implementation is (semi)automatically generated from the models
- Model-Driven Architecture (MDA)
  - particular vision of MDD proposed by the Object Management Group (OMG)
  - MDA can be regarded as a subset of MDD that uses OMG standards
- Model-Driven Engineering (MDE) is be a superset of MDD
  - not just development; encompasses other model-based tasks of a software engineering
  - E.g., the model-based evolution or the model-driven reverse engineering of a legacy system
- Model-Based Engineering (MBE) (or model-based development - MDD)
  - a softer version of MDE
  - MBE is a process in which software models play an important role although they are not necessarily the key artifacts of the development


MBE
MDE
MDD
MDA

MDSE: Top-Down Process

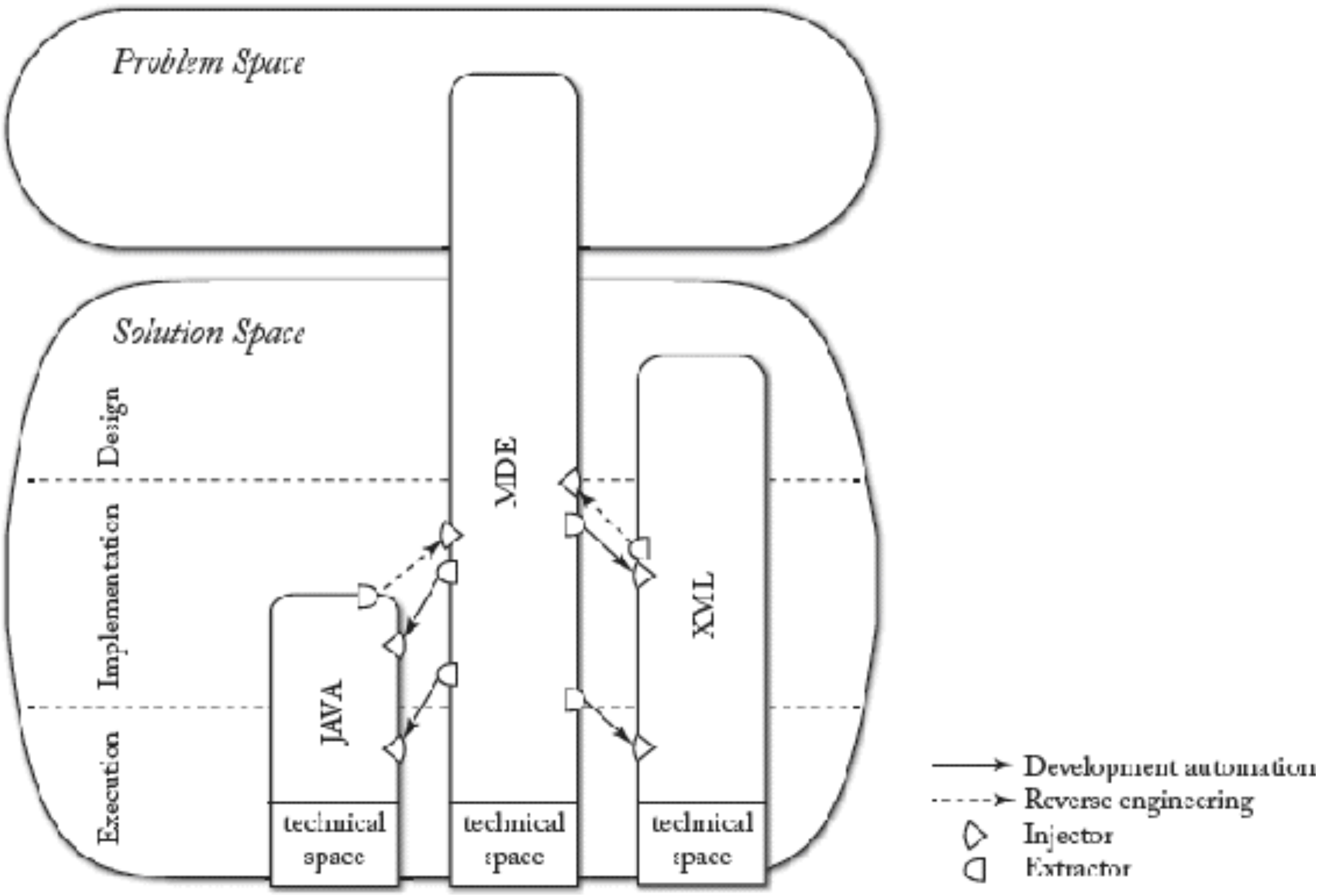


Problem vs. Solution

- **Problem space** (or **problem domain**): the field or area of expertise that needs to be examined to understand and define a problem
- **Domain models:** to define a common understanding of a field of interest, through the definition of its vocabulary and key concepts
- Domain model must not be defined with a thinking about design or implementation concerns
- **Solution space:** the set of choices at the design, implementation, and execution level performed to obtain a software application that solves the stated *problem* within the *problem domain*

Technical Spaces

- Technical spaces represent working contexts for the design, implementation, and execution of software applications
- These working contexts typically imply a binding to specific implementation technologies and languages
- A technical space can either span both the problem and solution domains or cover only one of these aspects
- During the software development process it is possible to move from one technical space to another (as represented by the arrows in the previous figure)


Problem Space
Solution Space
Design
Implementation
Execution
MDE
XML
JAVA
technical space
technical space
technical space
Development automation
Reverse engineering
Injector
Extractor

Languages

- Domain Specific Languages (DSLs): languages designed specifically for a certain domain, context, or company to ease the task of people that need to describe things in that domain
- For modeling languages, they are called Domain-Specific Modeling Language (DSML)
- E.g. HTML, SQL
- General-Purpose Modeling Languages (GPLs): represent tools that can be applied to any sector or domain for modeling purposes
- E.g. UML, Petri-nets

Different Levels of Abstraction

- Some models are more abstract than others

- Transformations can be defined for mapping a model specified at one level to a model specified at another level

Static and Dynamic

- Models can describe two main dimensions of a system: the static (or structural) part and the dynamic (or behavioral) part
- **Static models**: Focus on the static aspects of the system in terms of managed data and of structural shape and architecture of the system
- **Dynamic models**: Emphasize the dynamic behavior of the system by showing the execution sequence of actions and algorithms, the collaborations among system components, and the changes to the internal state of components and applications

Multi-View

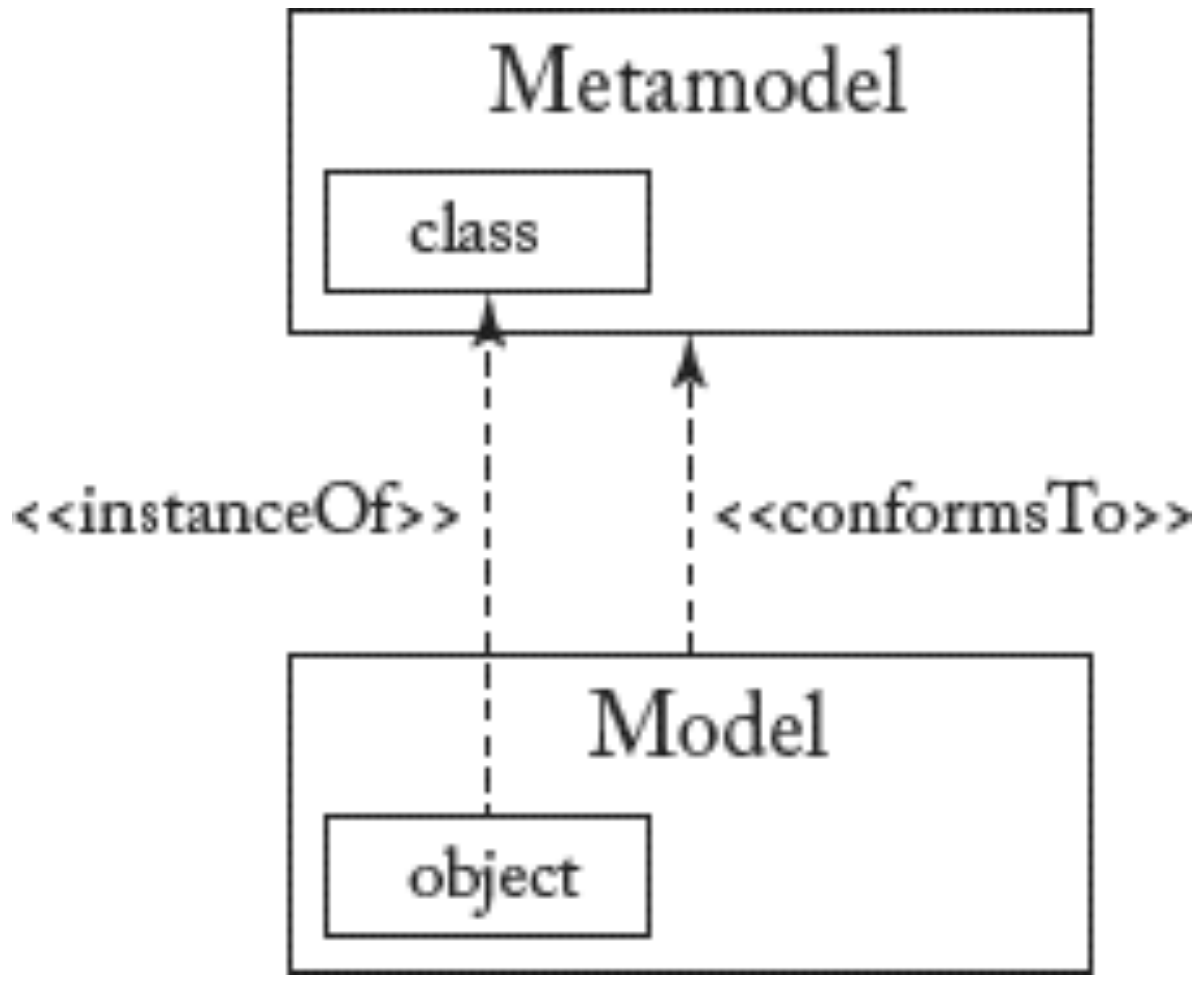
- A comprehensive view on the system should consider both static and dynamic aspects, preferably addressed separately but with the appropriate interconnections
- Multi-viewpoint modeling is one of the crucial principles of MDSE
- MDSE may lead to building various models describing the same solution
- Each model is focused on a different perspective and may use a different notation

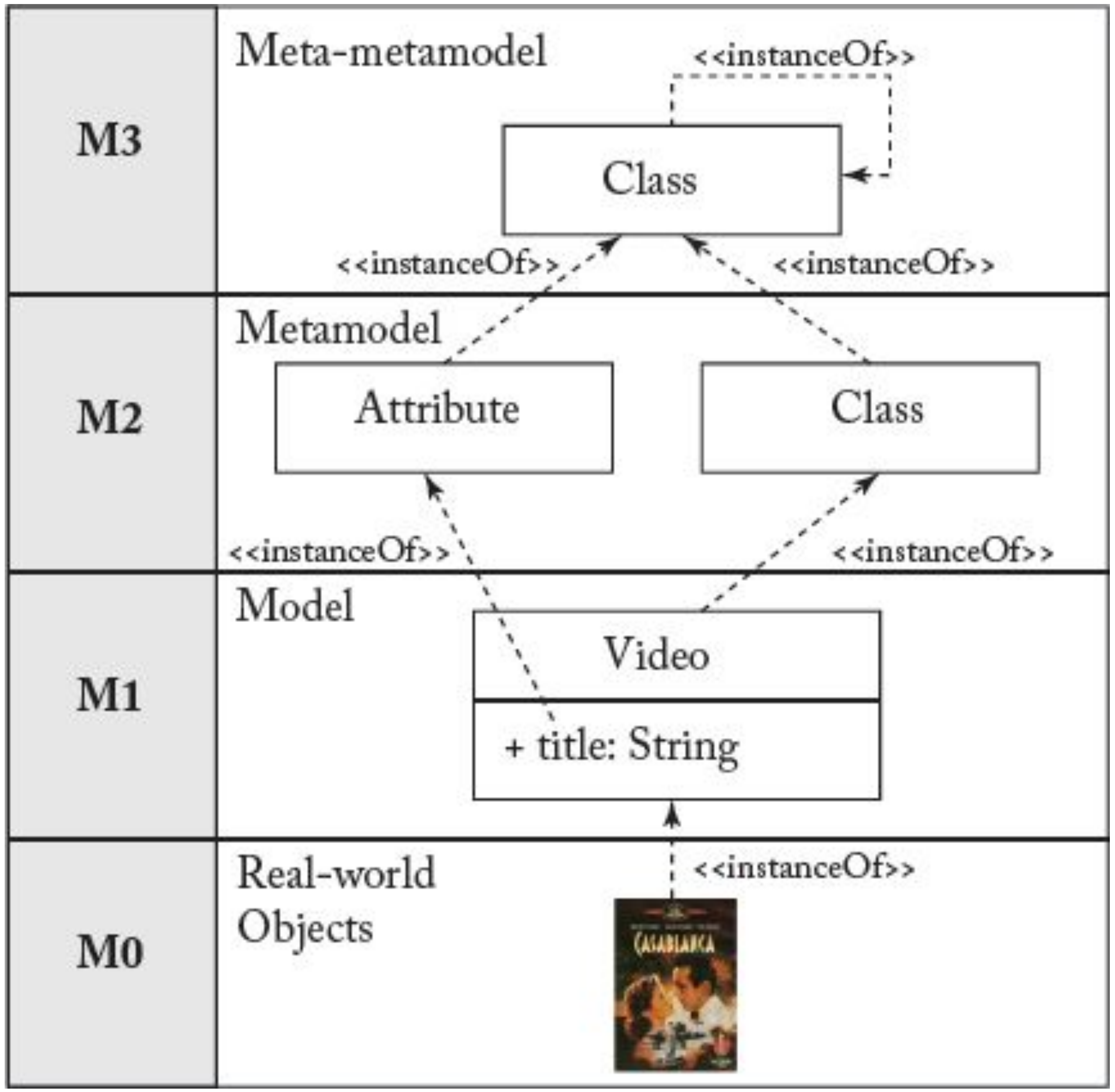
Metamodeling

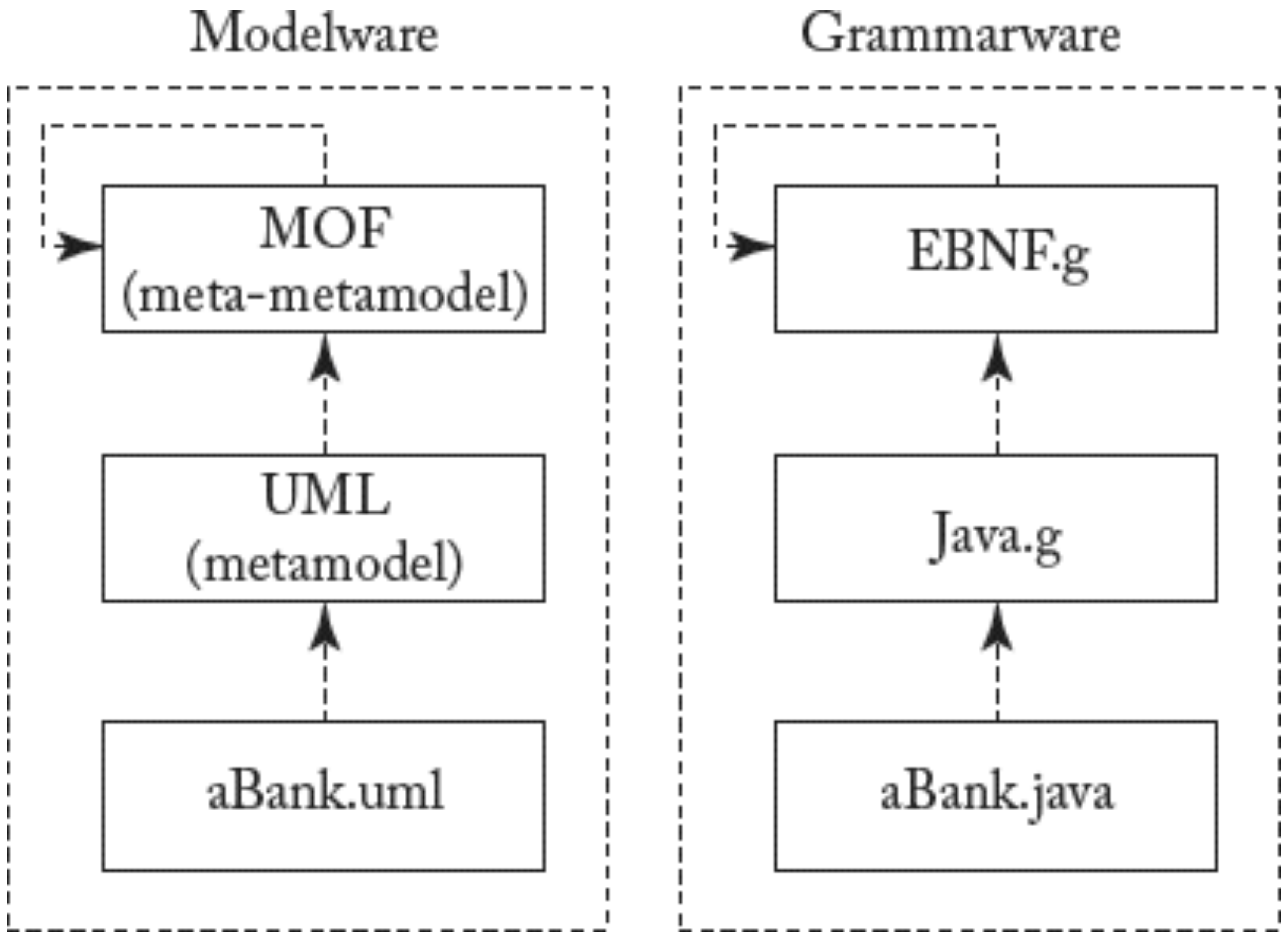
- **Metamodel**: yet another abstraction, highlighting properties of the model itself
- Metamodels basically constitute the definition of a modeling language (like grammars for PLs)
- They provide a way of describing the whole class of models that can be represented by that language
- One can define models of the reality, and then models that describe models (metamodels) and recursively models that describe metamodels (called meta-metamodels)

Metamodeling (cont.)

- While one could define infinite levels of metamodeling, meta-metamodels can be defined based on themselves
- Therefore it usually does not make sense to go beyond this level of abstraction
- We say that a model **conforms** to its metamodel in the way that a computer program conforms to the grammar of the programming language in which it is written.
- More specifically, we say that a model conforms to its metamodel when all its elements can be expressed as instances of the corresponding metamodel (meta)classes


Metamodel
class
<<instanceOf>>
<<conformsTo>>
Model
object


M3
Meta-metamodel
<<instanceOf>>
Class
<<instanceOf>>
<<instanceOf>>
M2
Metamodel
Attribute
Class
<<instanceOf>>
<<instanceOf>>
M1
Model
Video
+ title: String
M0
Real-world Objects
<<instanceOf>>
CASABLANCA

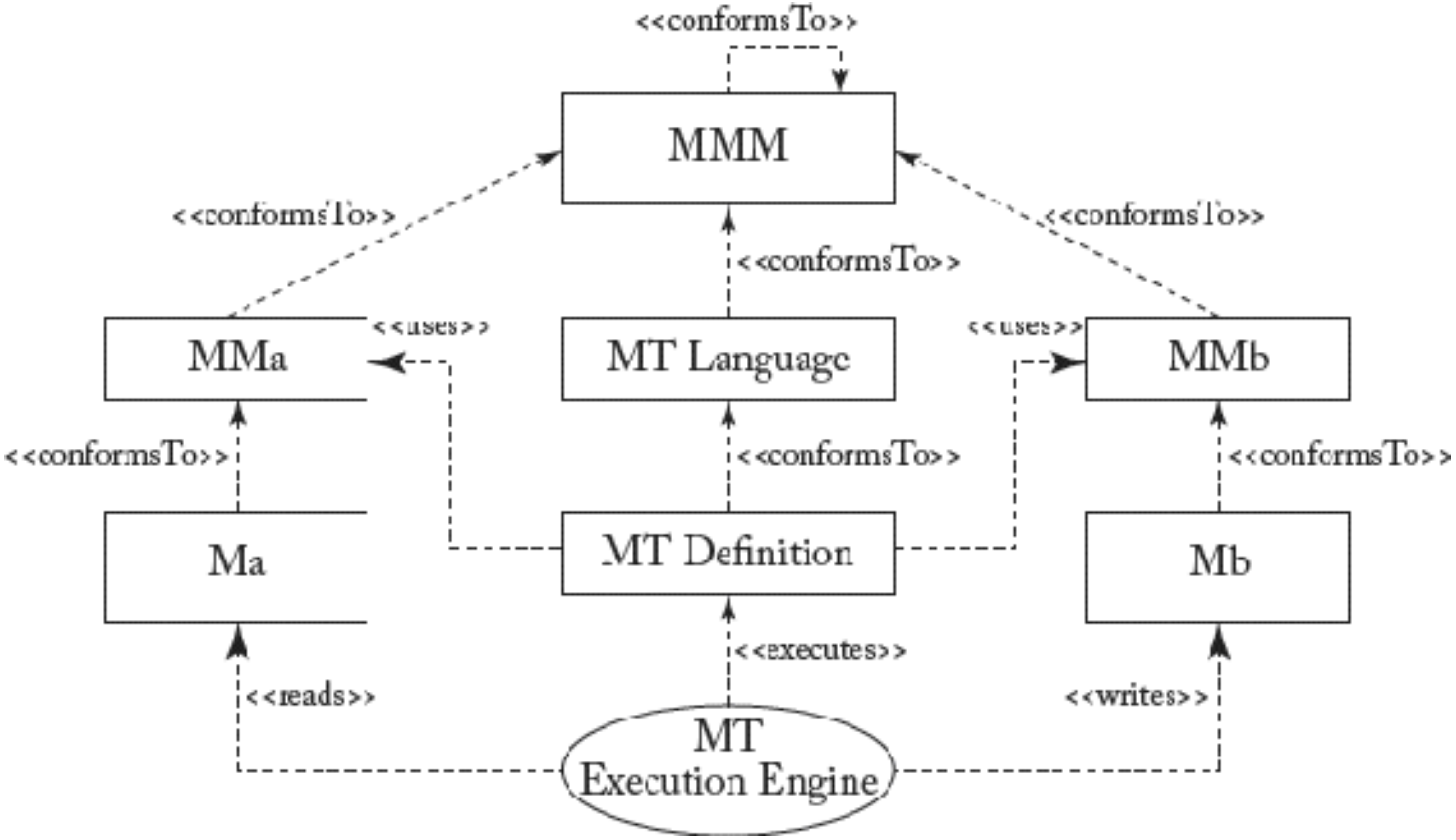

Modelware
MOF
(meta-metamodel)
UML
(metamodel)
aBank.uml
Grammarware
EBNF.g
Java.g
aBank.java

Transformations

- Transformations are defined at the metamodel level, and then applied at the model level (on models that conform to those metamodels)
- Performed between a *source* and a *target* model
- Transformation can be written manually
- Or can be defined as a refined specification of an existing one
- Transformations can be produced automatically out of some higher level mapping rules between models

Transformations (cont.)

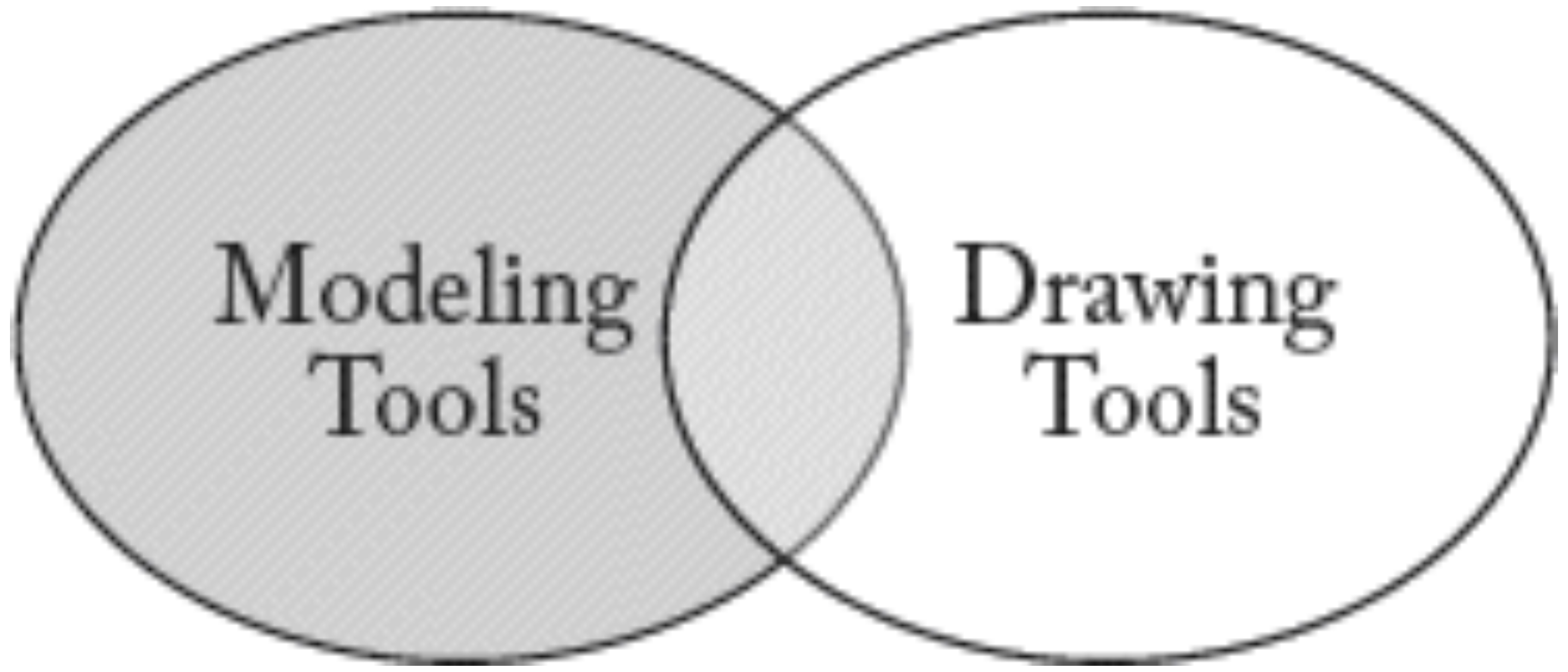
- Transformations can also be produced automatically out of some higher level mapping rules between models in two different ways:
  - 1. defining a mapping between elements of a model to elements of another one (*model mapping*)
  - 2. automating the generation of the actual transformation rules through a system that receives as input the two model definitions and the mapping between them and produces the transformations
- Transformations themselves can be seen as models, and managed as such, including their metamodeling (see next figure)



<<conformsTo>>
MMM
<<conformsTo>>
<<conformsTo>>
<<conformsTo>>
MMa
<<uses>>
MT Language
<<uses>>
MMb
<<conformsTo>>
<<conformsTo>>
<<conformsTo>>
Ma
MT Definition
Mb
<<executes>>
<<reads>>
<<writes>>
MT
Execution Engine

Tool Support

- Drawing tool ≠ modeling tool
- Only some tools are drawing and modeling tools at the same time



- A drawing tool can be only considered as a modeling tool if it “understands” the drawings
- It must understand what shapes represent (e.g., classes)
- This should at least be enough to validate a model, i.e., check that the model is a correct instance of its metamodel

Tool Support (cont.)

- Reasons to use modeling tools instead of drawing:

1. A modeling tool is able to export or manipulate the models using (external) access APIs
2. A modeling tool guarantees a minimum level of semantic meaning and model quality, because it grants alignment to some kind of metamodel
3. A modeling tool typically provides appropriate features for model transformations

TODO

- Read chapter 1 and 2 of the book (Model-Driven Software Engineering in Practice)
- What every developer should know about EMF
    - https://www.youtube.com/watch?v=0EkatYP7IK0
    - https://eclipsesource.com/blogs/tutorials/emf-tutorial/

Class 2
MDSE Use Cases

TODO

- Read chapter 1, 2, 3 of the book “Model-Driven Software Engineering in Practice”
- Building a tool for engineers based on EMF (using EMF to create tools for others)
    - https://www.youtube.com/watch?v=LBGUQ-B75yc

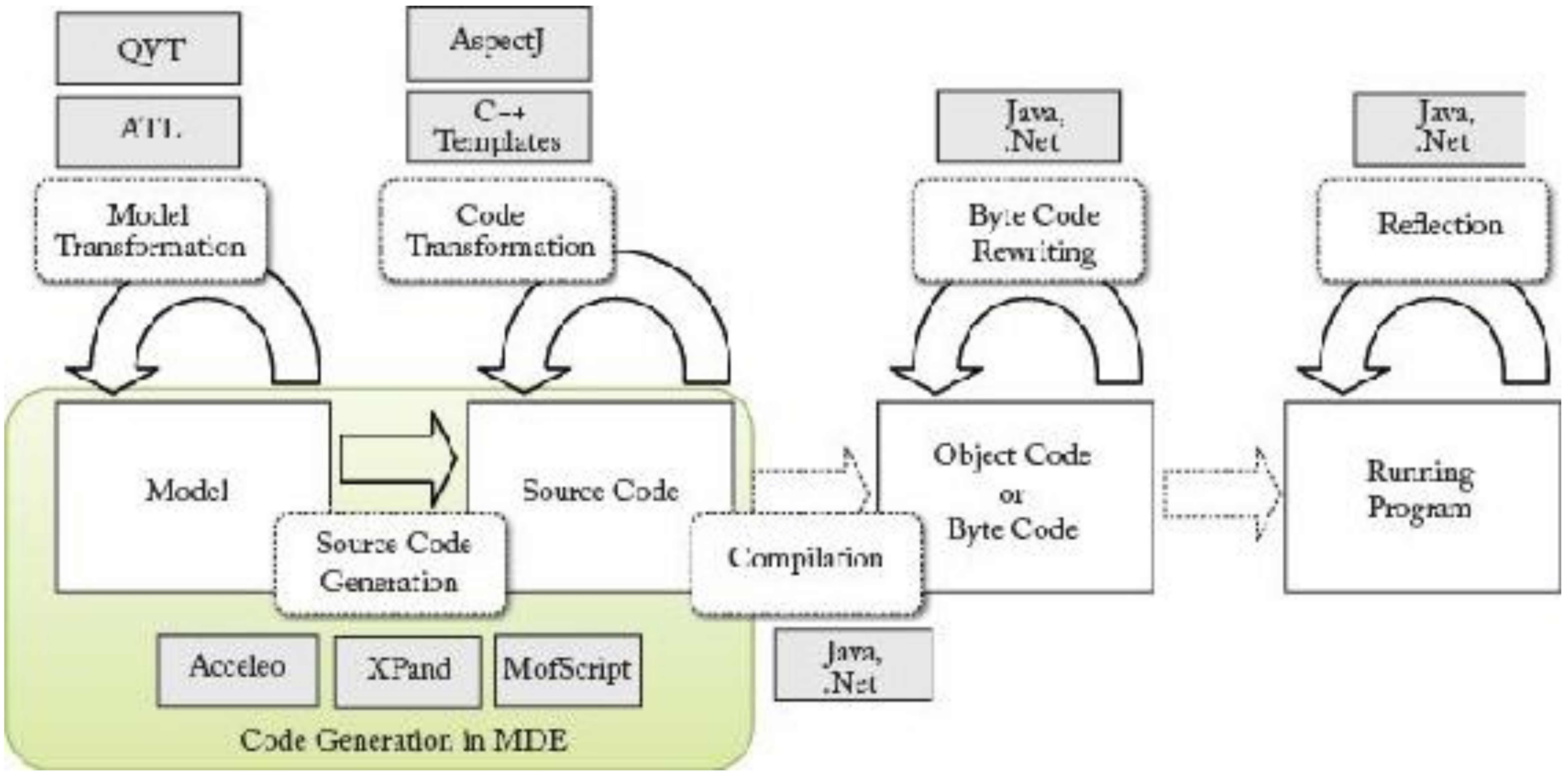
What are 3 possible use cases for MDSE?

1. Automating software development
2. System interoperability
3. Reverse engineering
4. Modeling the organization

Advantages Automating Software Development

- Increase of **communication** effectiveness between the stakeholders
- Increase in the **productivity** of the development team thanks to the (partial) automation of the development process
- Also, the automation reduces also the number of **defects** that could be inadvertently introduced by the developers

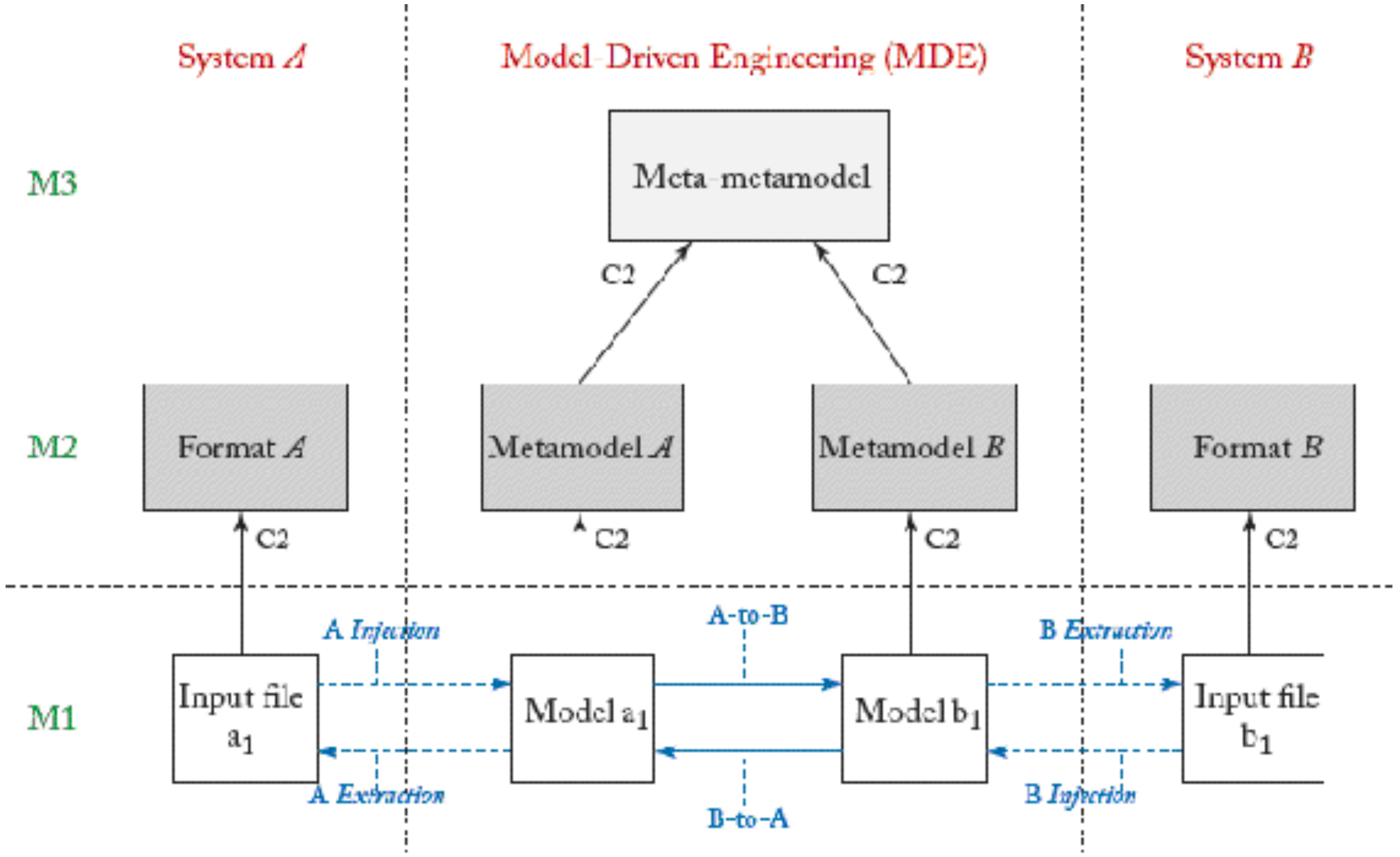
Overview


QVT
ATL
Model Transformation
AspectJ
C++ Templates
Code Transformation
Java, .Net
Byte Code Rewriting
Java, .Net
Reflection
Model
Source Code
Source Code Generation
Compilation
Object Code or Byte Code
Running Program
Acceleo
XPand
MofScript
Java, .Net
Code Generation in MDE

Advantages of Model-Driven Interoperability

- Making the bridges between systems **explicit**
- Systematic high-level **transformations** between different systems (e.g. using model-to-model transformations)

Overview

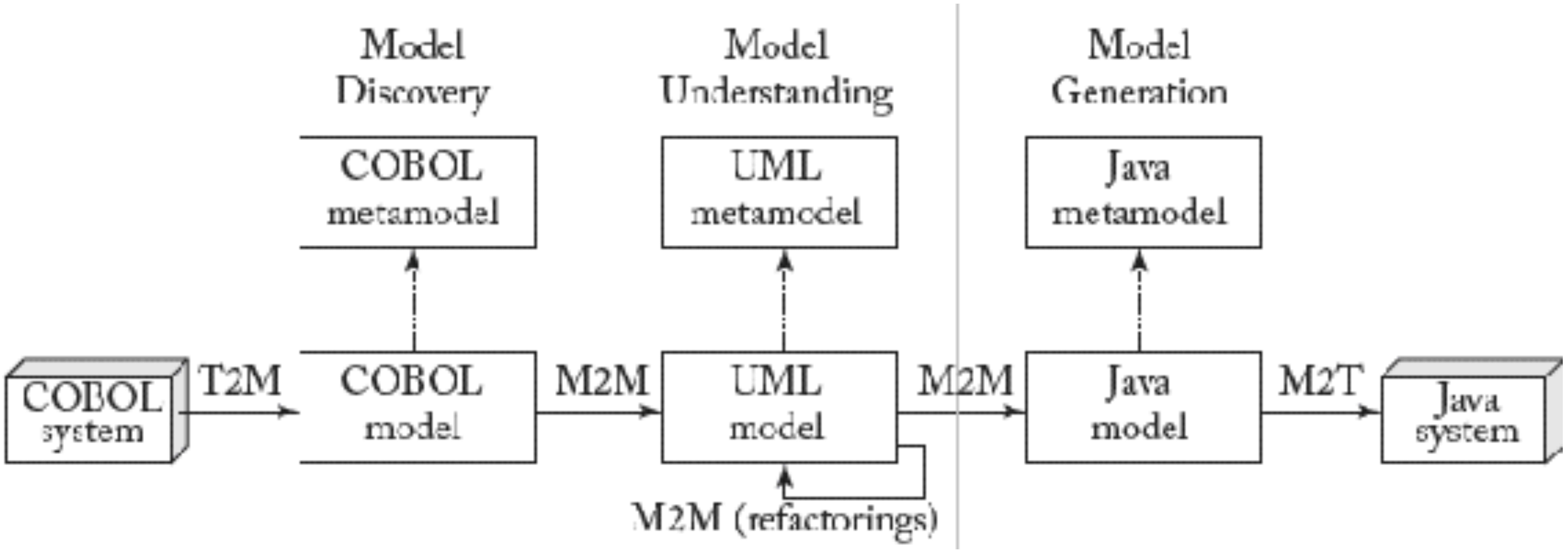


System A
Model-Driven Engineering (MDE)
System B
M3
Meta-metamodel
C2
C2
M2
Format A
Metamodel A
Metamodel B
Format B
C2
C2
C2
C2
A-to-B
A Injection
B Extraction
M1
Input file a1
Model a1
Model b1
Input file b1
A Extraction
B-to-A
B Injection

Advantages of Model-Driven Reverse Engineering

- Systematic and high-level **representation** of legacy system (instead of (old) code)
- Models can then be used to support **further evolution** (e.g. through code generation)

Example



Model Discovery
Model Understanding
Model Generation
COBOL metamodel
UML metamodel
Java metamodel
COBOL system
T2M
COBOL model
M2M
UML model
M2M
Java model
M2T
Java system
M2M (refactorings)

Advantages of Modeling the Organization

- **Explicit formal** business processes
- Allows to **analyze** the processes (performance, improvement, finding bottlenecks, …)
- Easier for people to **follow** and **manage** the processes

TODO in Class

Model It

- Create a model for an university
  - Universities have faculties (engineering, social sciences, etc.)
  - Faculties have degrees and professors
  - Degrees have courses
  - Courses have editions
  - Editions have students and professors
  - Students have grades in each edition’s course
  - …
- Fill free to include more information
- Add the necessary information to each entity (e.g. name, birthdate, credits, etc.)

Generate It

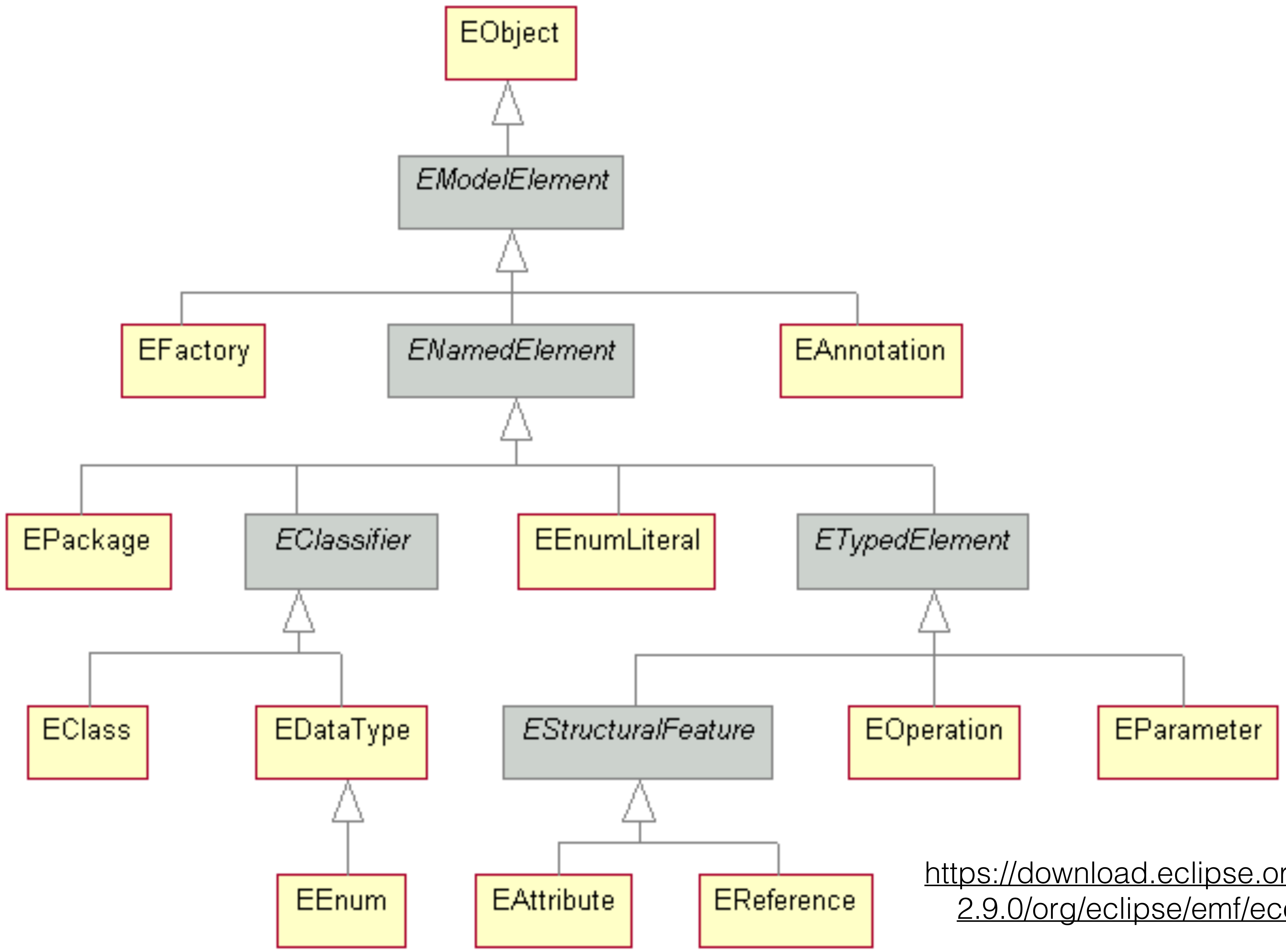
- Generate the code based on the model
  - Create a couple of tests to better understand the reflexive API
    - You can also check pages 40 to 43 and 52 and 53 (“2.5.3. The Reflective EObject API”) of the book “EMF: Eclipse Modeling Framework” (available at moodle)
- Generate the editor
- Run the editor and recreate (part) of UPorto
    - You can also check pages 116 to 119 of the book “EMF: Eclipse Modeling Framework” (available at moodle)
- Create a repository and share with me your results (**MANDATORY**)

TODO at Home

- Read chapter 5 of the book “EMF”

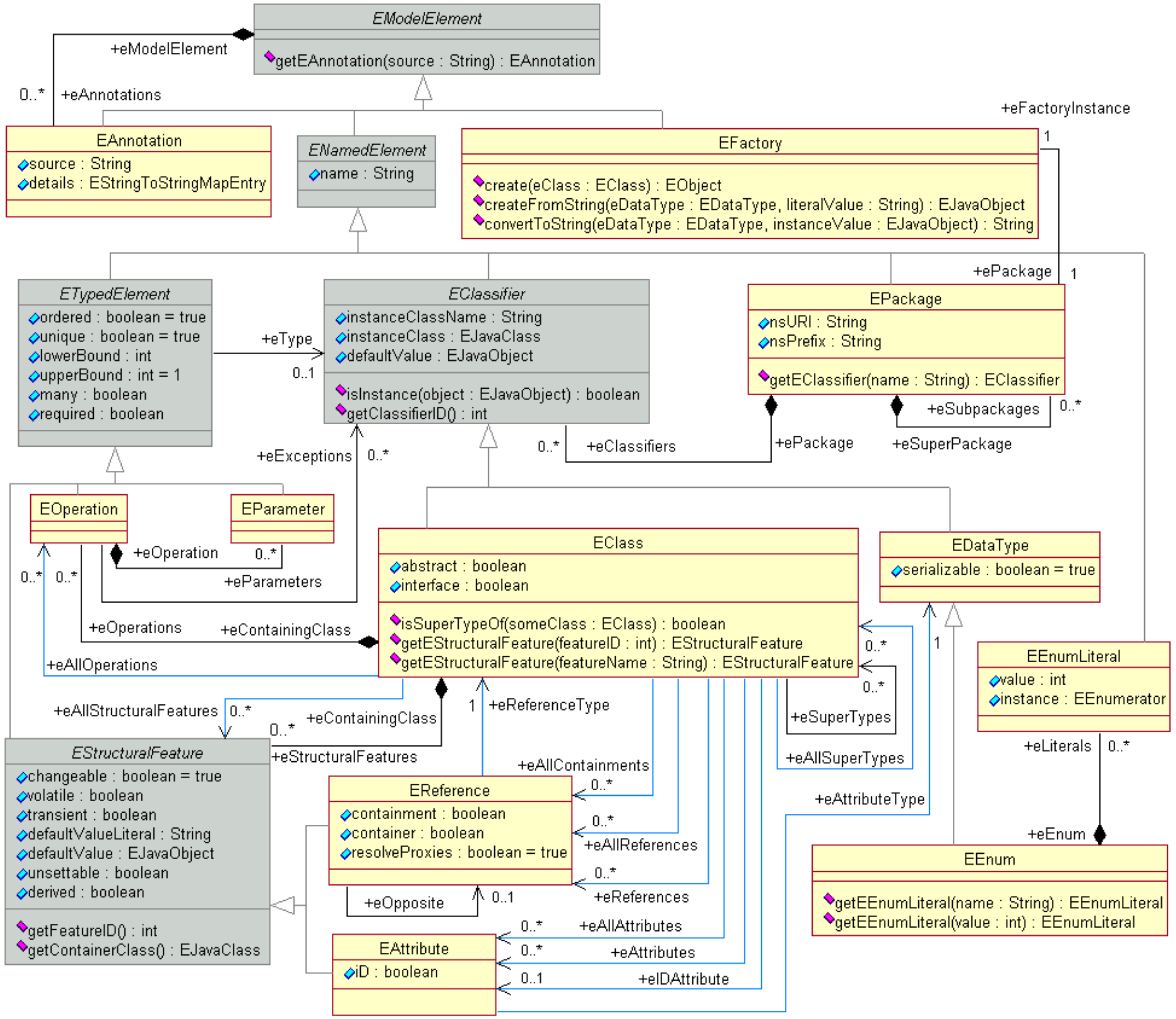
Class 3
Ecore Metamodel

Ecore Meta-Metamodel Components

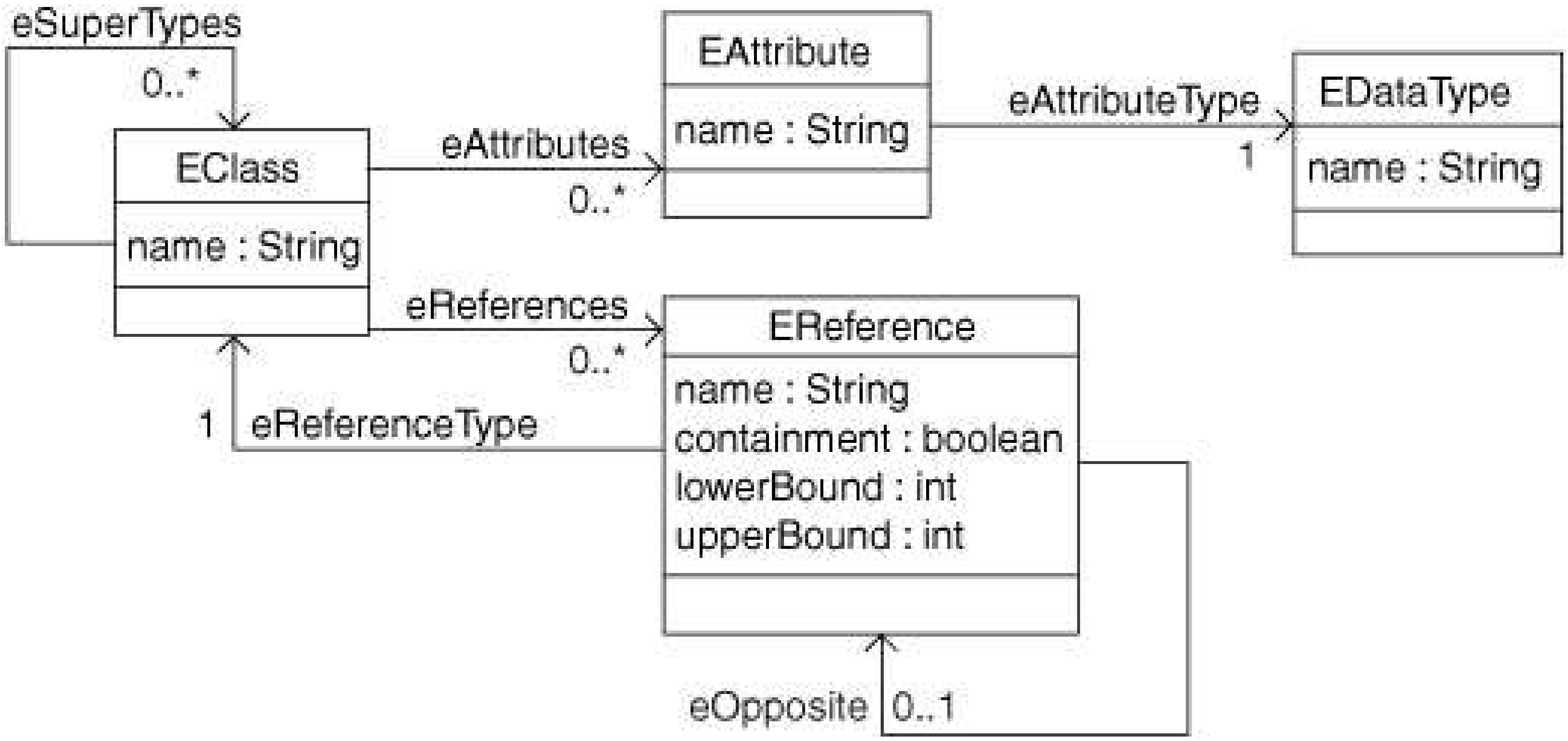


https://download.eclipse.org/modeling/emf/emf/javadoc/2.9.0/org/eclipse/emf/ecore/package-summary.html

Complete Ecore Meta-Metamodel



Ecore Kernel



Ecore Kernel Classes

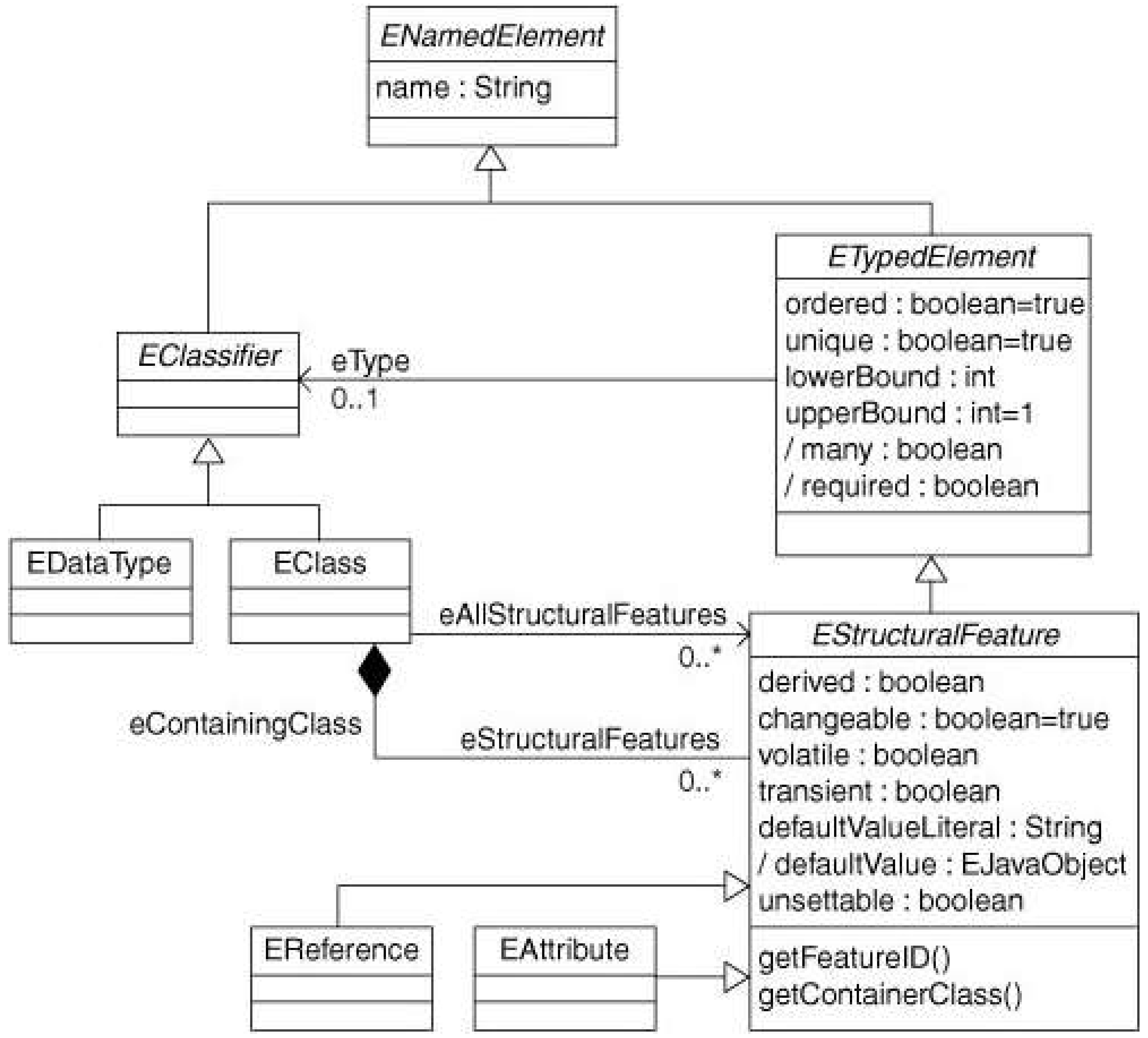
- **EClass** models classes themselves. Classes are identified by name and can have a number of attributes and references. To support inheritance, a class can refer to a number of other classes as its supertypes.
- **EAttribute** models attributes, the components of an object's data. They are identified by name, and they have a type.
- **EDataType** is used to represent simple types whose details are not modeled as classes. Instead, they are associated with a primitive or object type fully defined in Java. Data types are also identified by name, and they are used as the types of attributes.

Ecore Kernel Classes (cont.)

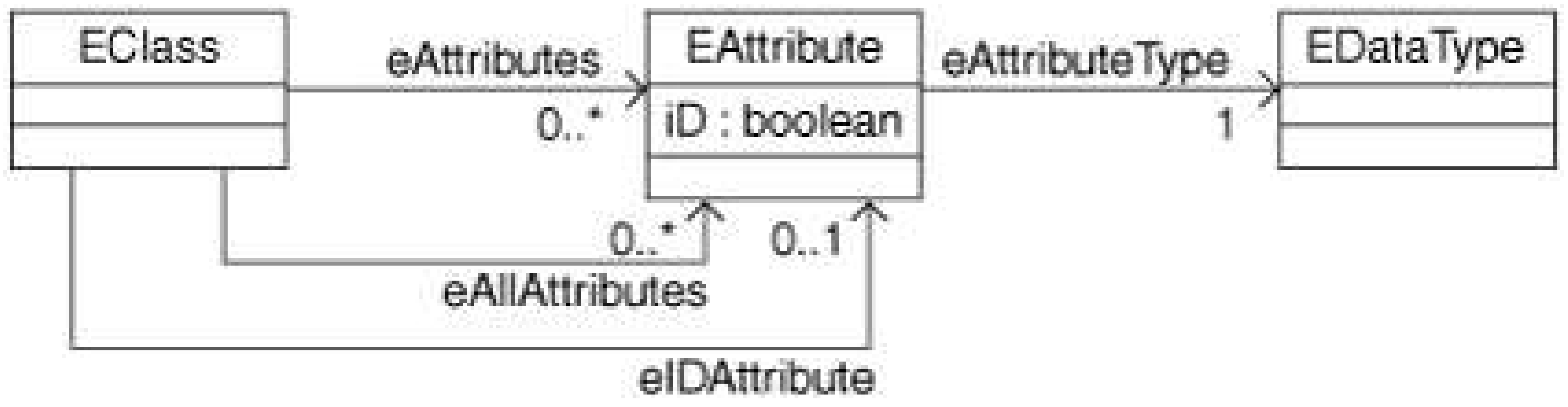
- **EReference** is used in modeling associations between classes; it models one end of such an association. Like attributes, references are identified by name and have a type. However, this type must be the EClass at the other end of the association (eReferenceType). If the association is navigable in the opposite direction, there will be another reference to represent this bidirectionality. A reference specifies lower and upper bounds on its multiplicity. Finally, a reference specifies whether it is being used to represent a stronger type of association, called containment.

Ecore Metamodel Expanded

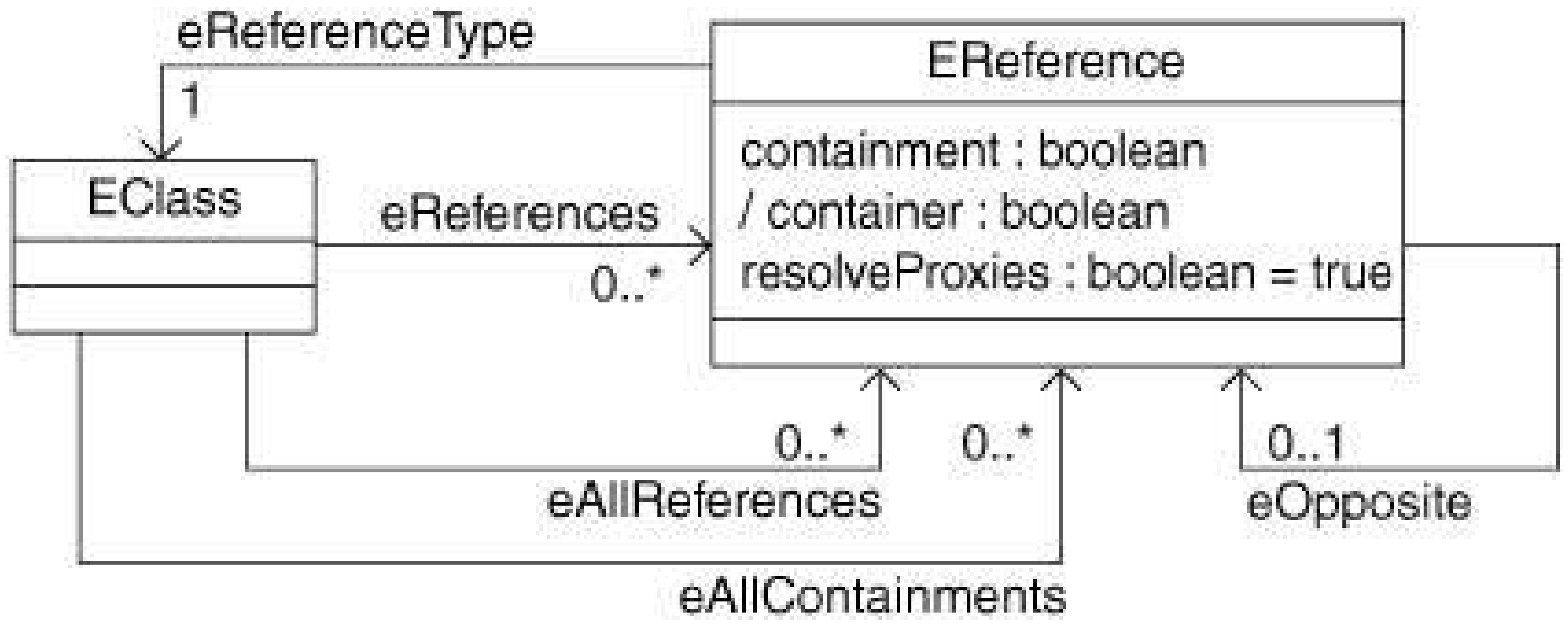
The previous model can be refined into this, factoring out some attributes.



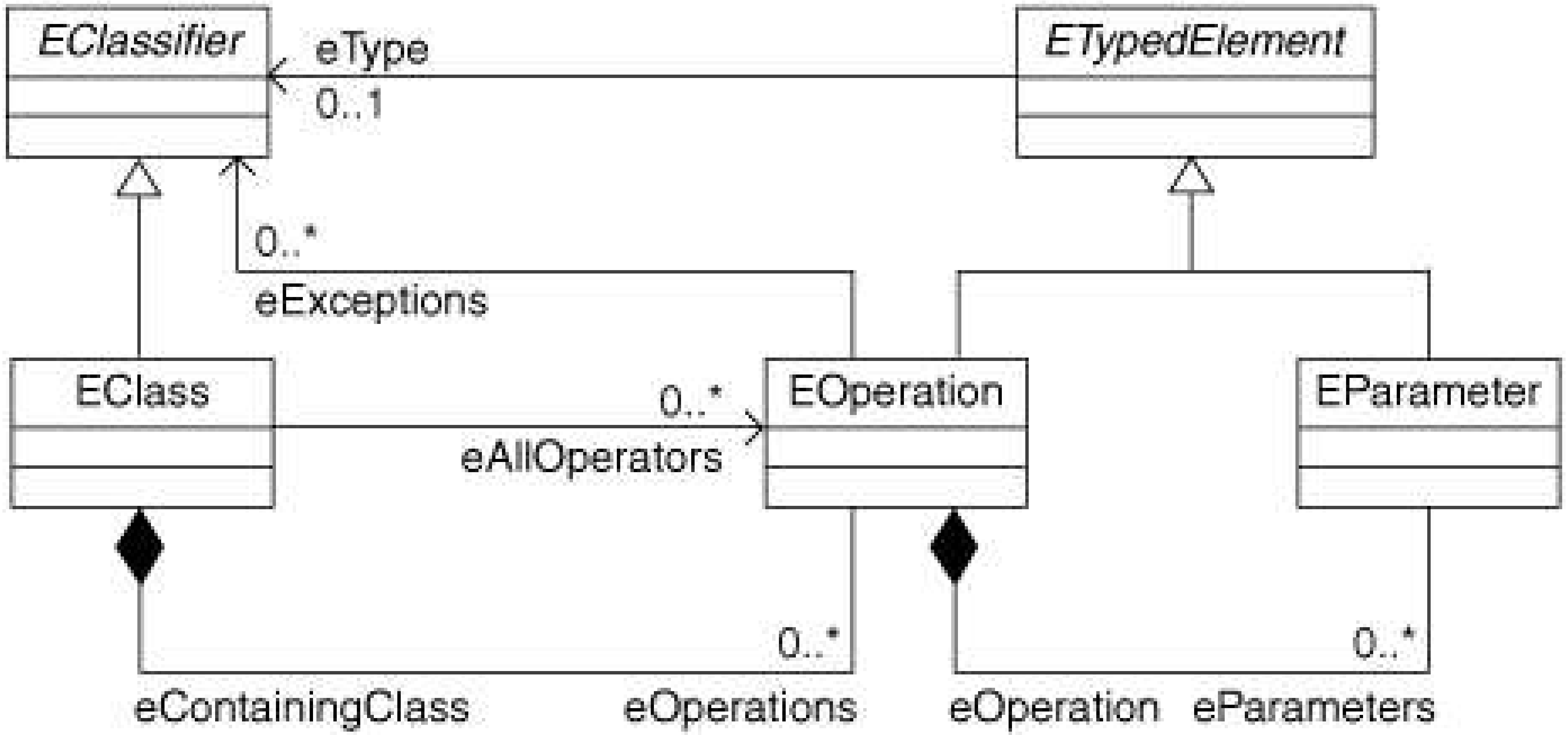
Ecore Attributes



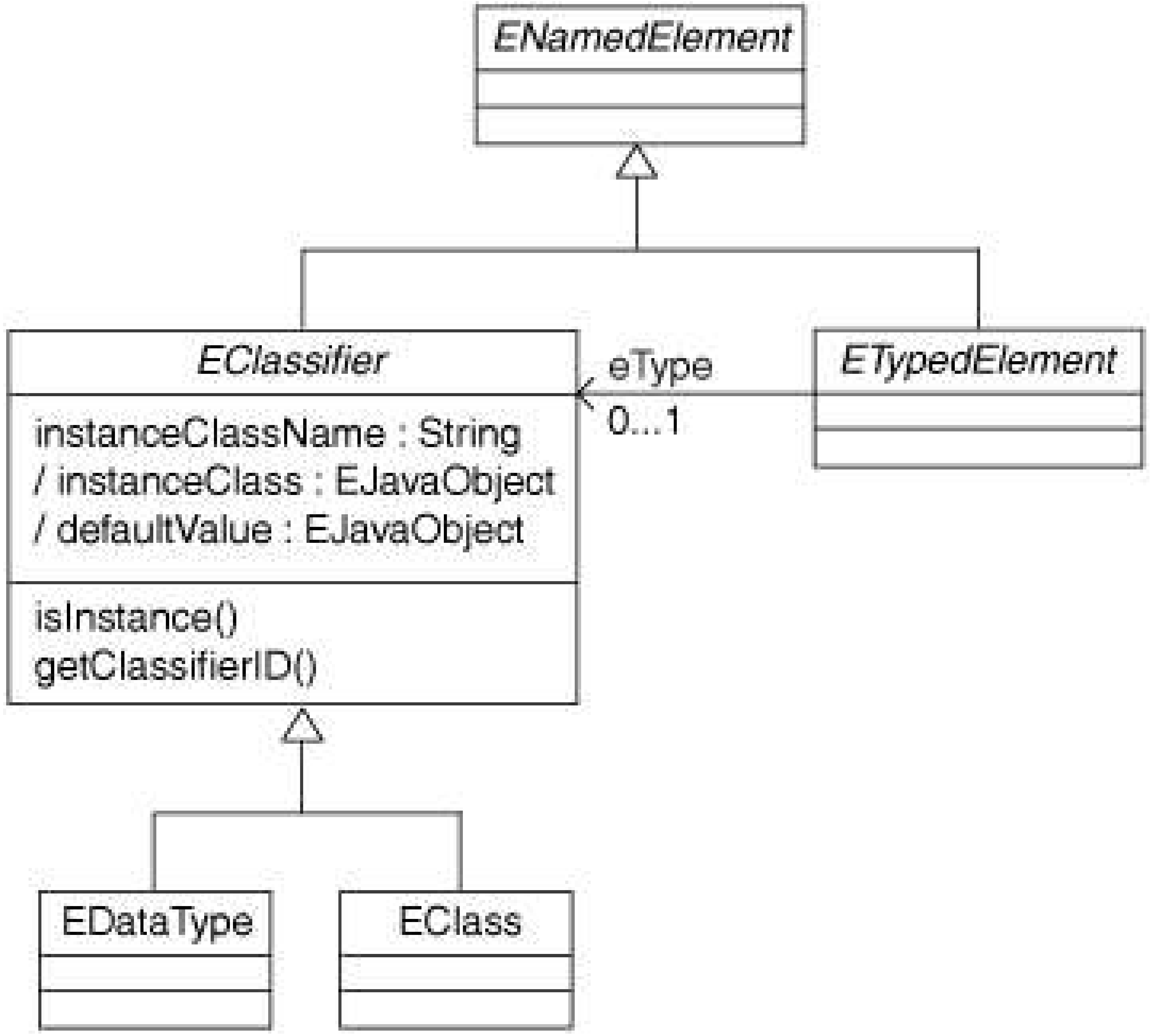
Ecore References



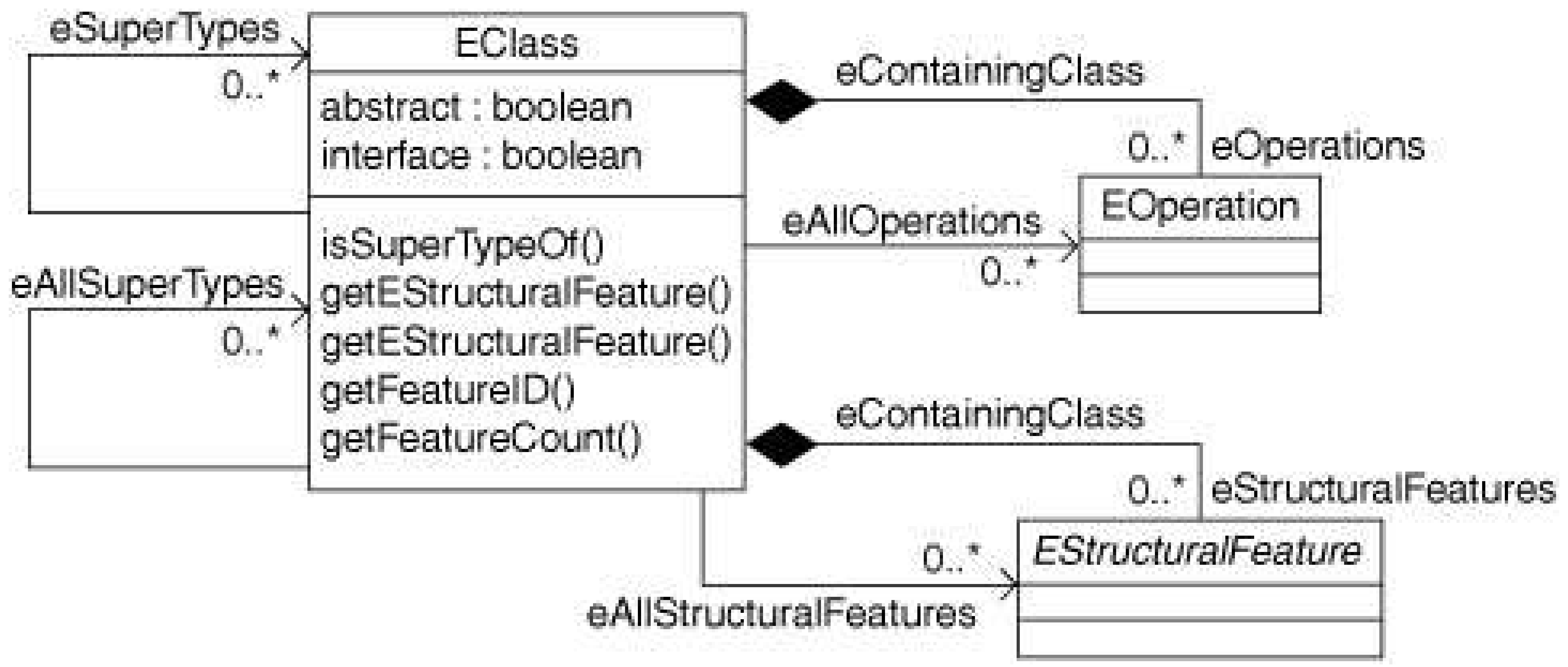
Ecore Operations and Parameters



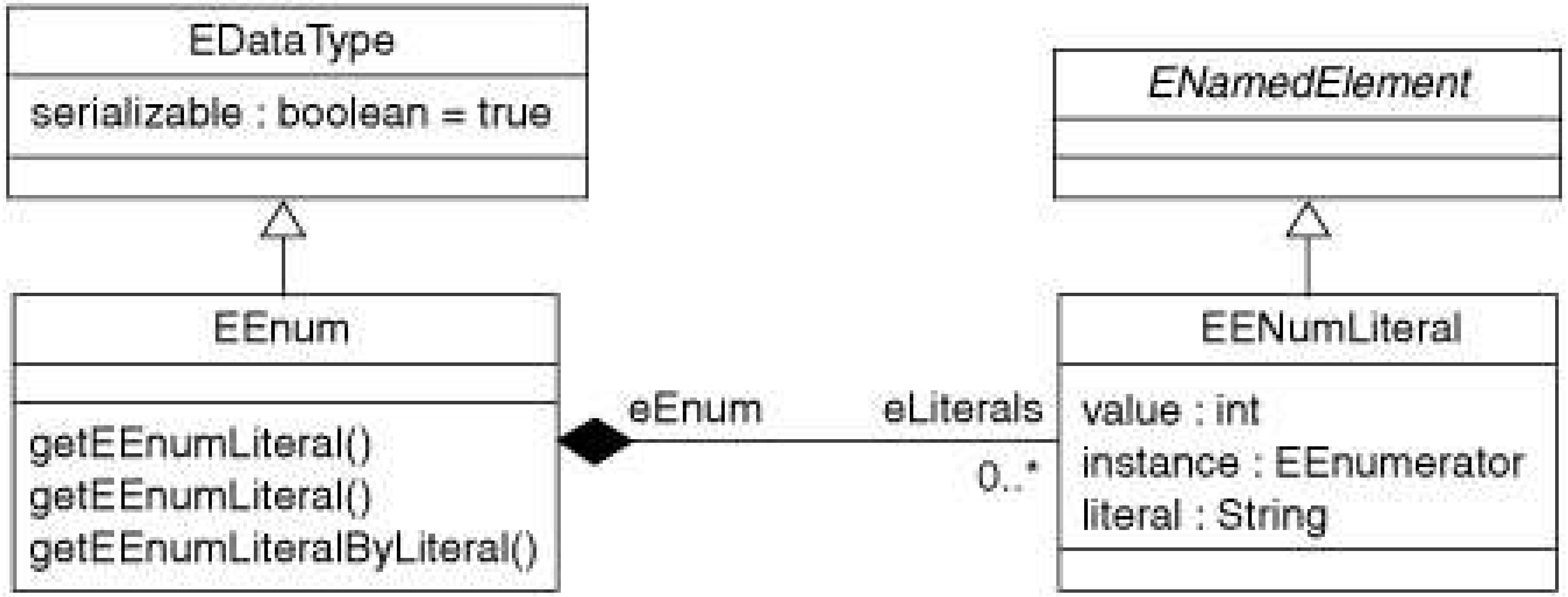
Ecore Classifiers



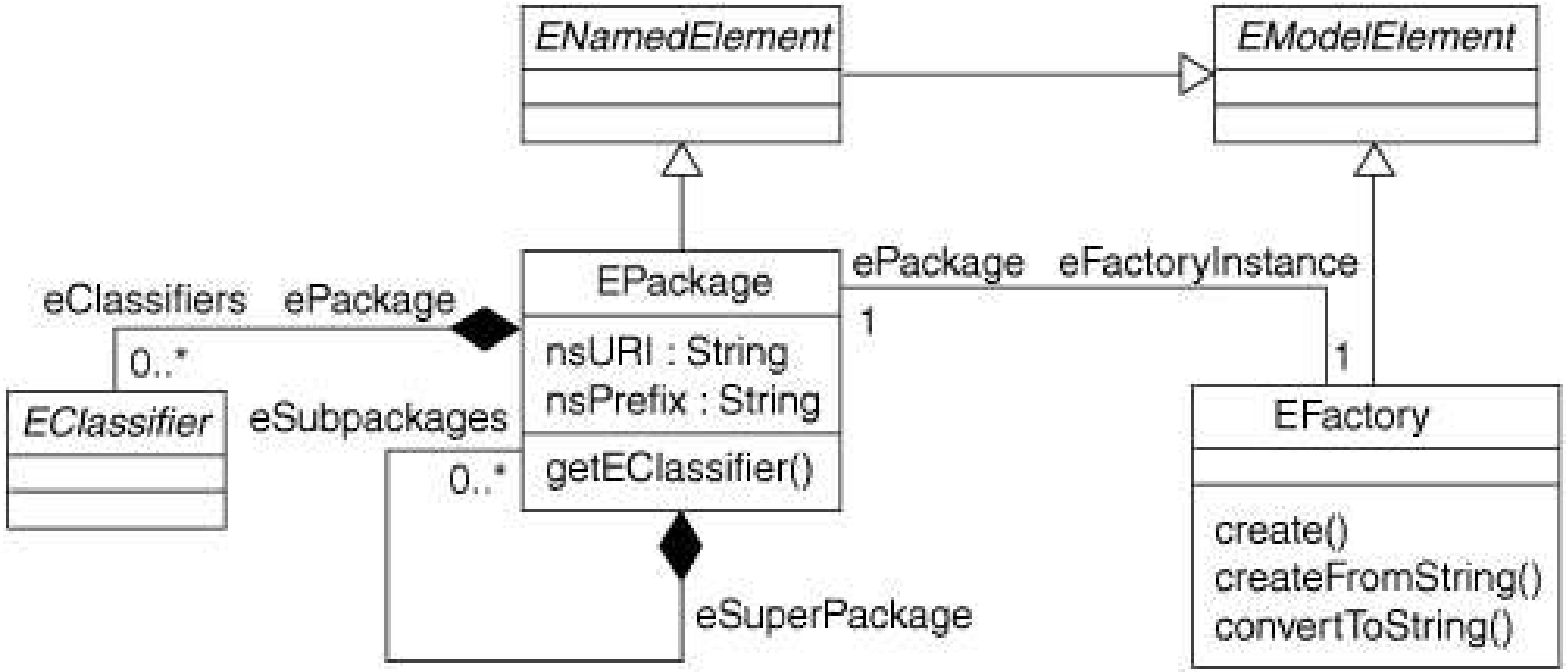
Ecore Classes



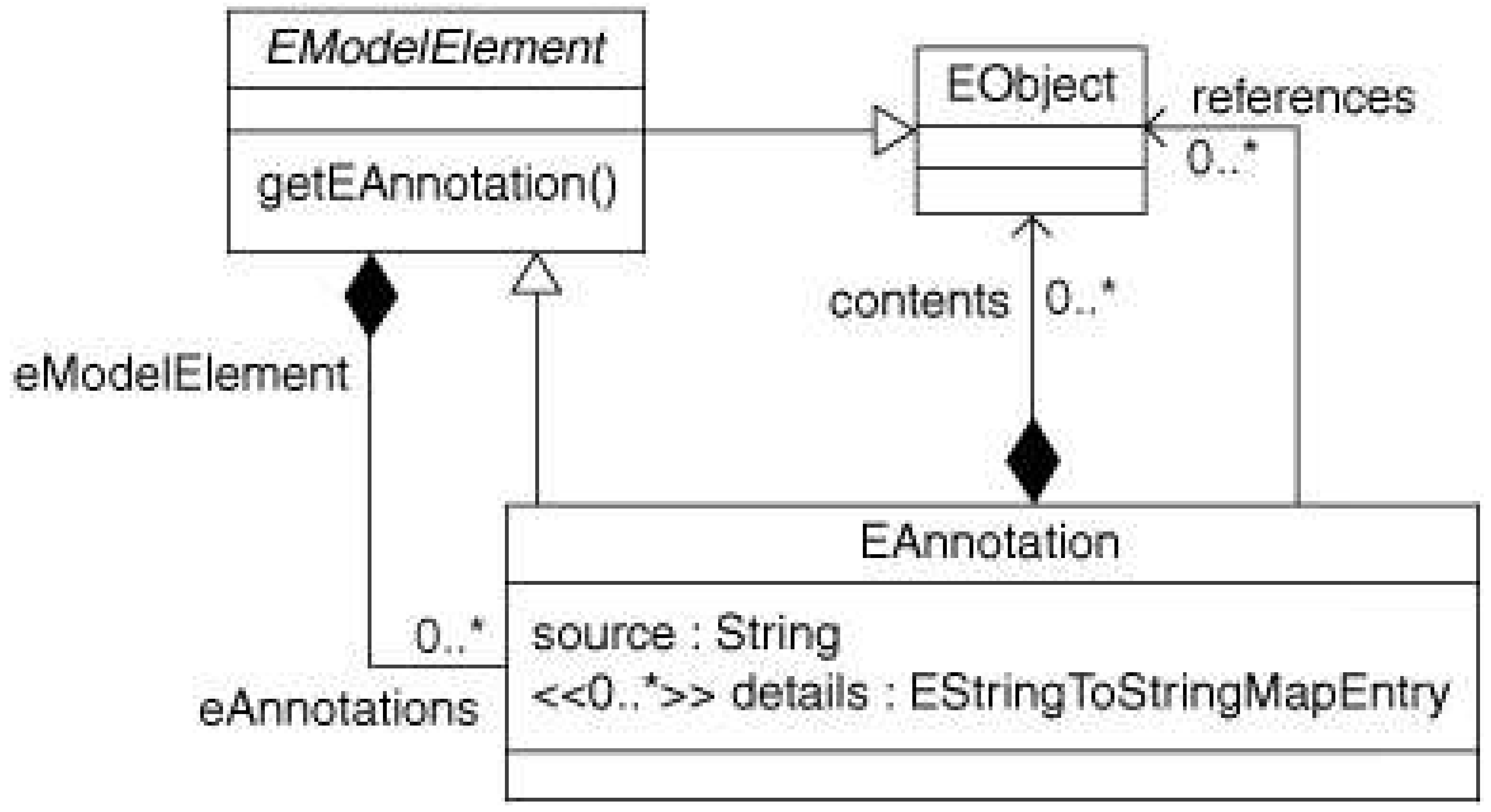
Ecore Data Types, Enumerated Types, and Literals



Ecore Packages and Factories



Ecore Annotations



TODO in Class

- Start working on the assignment (gather examples, create concepts, start modeling)
    - https://docs.google.com/document/d/13Qn3nrXPA2U76ntRHguJpwImVWdBECS_S4BraoTB9iw/edit?usp=sharing

TODO at Home

- Read sections 7.1, 7.2, 7.3, and 7.5.1 through 7.5.4 of the book “Model-Driven Software Engineering in Practice”

Class 4
Developing your Own Modeling Language

Abstract Syntax = Metamodel + Model Restrictions

Metamodel-Centric Language Design

- A grammar defines all valid sentences of a language
- A metamodel defines all valid models of a modeling language
- Metamodels containing classes, attributes, and associations define the modeling concepts and their properties (e.g. UML)
- Modeling constraints are only partially described (e.g. associations’ multiplicity)
- Additional (complex) constraints may be defined using a constraint language (e.g. OCL)
- metamodel + constraints = abstract syntax

Abstract Syntax Development

- Having an explicit metamodel conforming to a standardized meta-metamodel comes with several benefits:
  - **Precise language definition**: formal definition of the language's syntax which is processable by machines. For example, the metamodels may be used to check if models are valid instances.
  - **Accessible language definition**: if the metamodel is based on a well-known modeling language (e.g. UML class diagrams), the knowledge of such language is sufficient to read and understand the modeling language definitions in the form of meta-models.
  - **Evolvable language definition**: metamodels may be subject to modifications(e.g. the language may be extended with new subclasses for already existing metamodel classes). Having an accessible language definition further contributes to an easy adaptation of modeling languages based on metamodels.

- Generic tools, which are metamodel agnostic, may be developed based on the metametamodel level
- For example, current metamodeling frameworks (e.g. EMF) provide sophisticated reflection techniques to develop programs which are applicable to all instances of metamodels, for instance:
  - Exchange formats: based on the meta-metamodel, there is support to serialize/deserialize models into XML documents which are exchangeable between tools supporting the same meta-metamodel.
  - Model repositories: models may be stored to and retrieved from a model repository based on generic storage/loading functionalities.
  - Model editors: for modifying models, generic editors may be provided which are applicable on all models, irrespective of which modeling language is used.

Metamodel Development Process

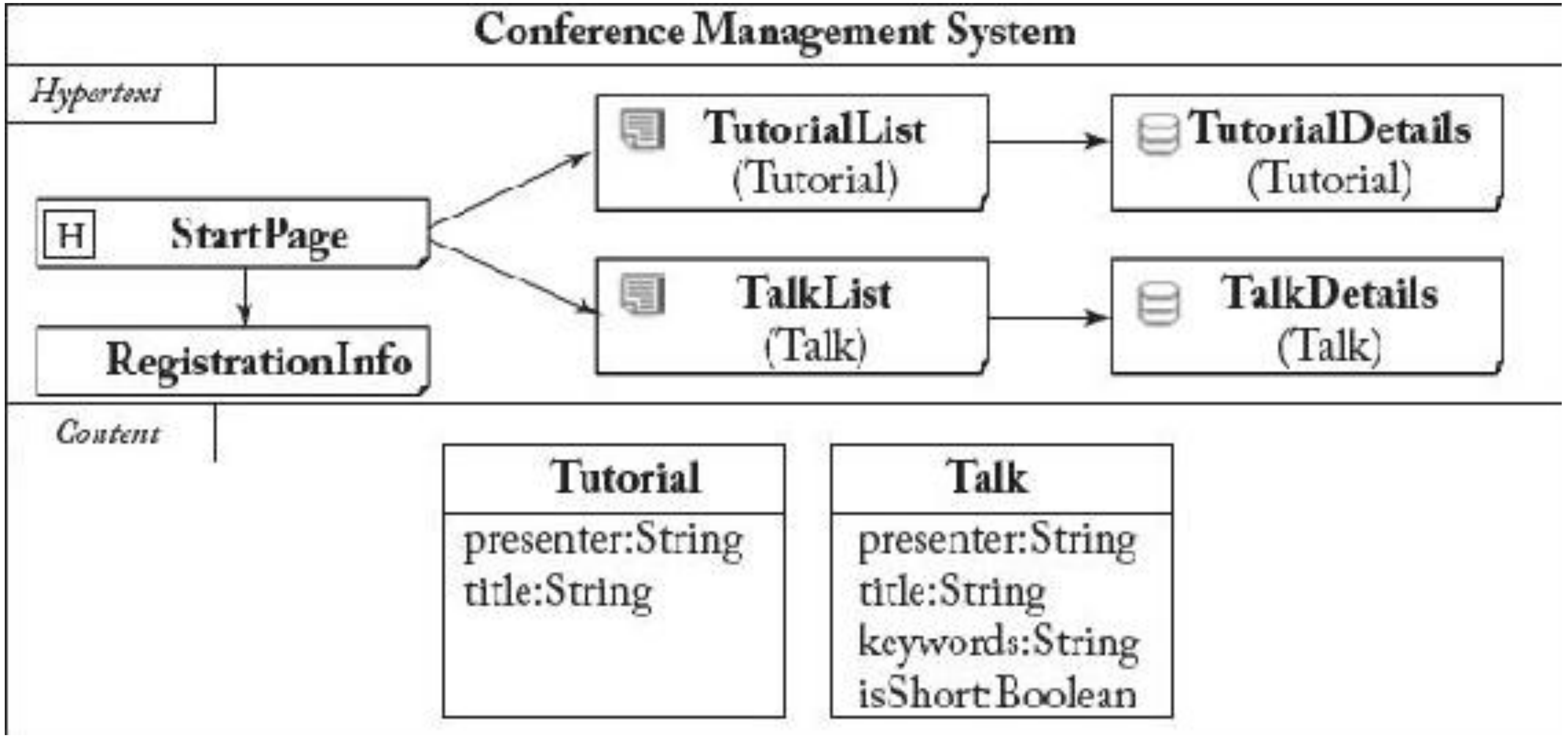
- **Step 1**: Modeling domain analysis: three aspects have to be considered: purpose, realization, and content of the language
  - The identification of modeling concepts and their properties (the content) is the most challenging aspect
  - For this purpose, the modeling domain that should be supported by the modeling language has to be analyzed
  - A pragmatic way to do this is to find several reference examples that should be expressible in the modeling language to be developed i.e. the requirements for the language are defined by example

Metamodel Development Process (cont.)

- **Step 2**: Modeling language design: a metamodeling language is used to formalize the modeling concepts by modeling the abstract syntax of the language and modeling constraints should be formalized by using OCL
  - The output of this step is a metamodel for the modeling language
- **Step 3**: Modeling language validation: the metamodel is instantiated by modeling the examples to validate the completeness and correctness of the metamodel
  - Other general principles of language design such as simplicity, consistency, scalability, and readability also have to be considered
  - The result of this step provides important feedback for the next iteration of the metamodel development process

An Example - sWML

- A possible sketch of a web app model



sWML Purpose

sWML should allow the modeling of the **content layer** and the **hypertext layer** of Web applications. The content layer defines the schema of the persistent data which is presented to the user by the hypertext layer in form of Web pages. In addition, the hypertext layer defines the navigation between Web pages and the interaction with the content layer, e.g., querying content from the database.

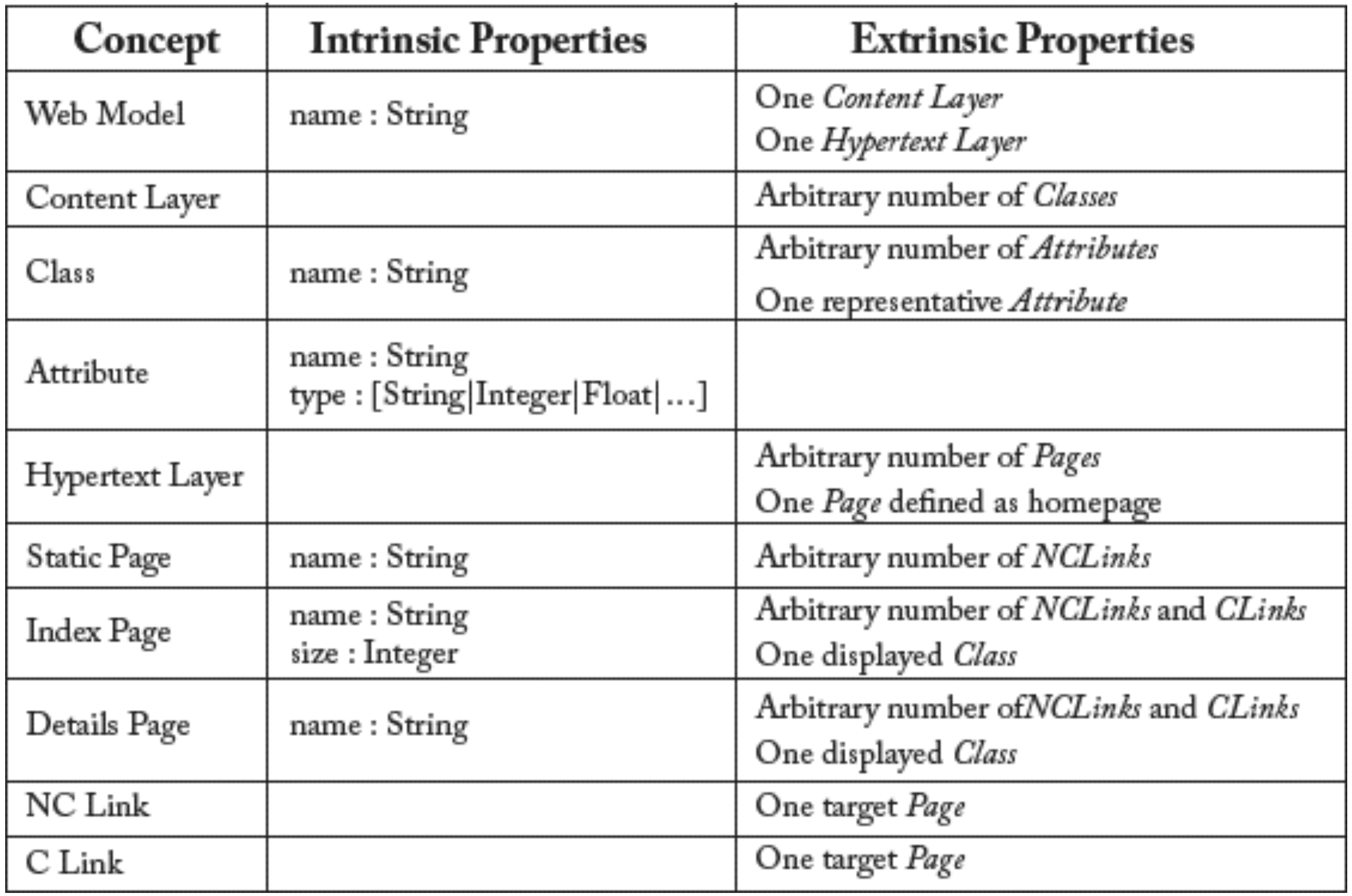
sWML Realization

To support different sWML user types, a **graphical syntax** should be defined which may be used for discussing with domain experts and for reasoning on the structure of the hypertext layer and the content layer. However, for making a smooth transition from the standard development process based on programming languages to the model-driven approach, additionally, a **textual syntax** should be provided for developers who are familiar and used to working with text-based languages.

sWML Content

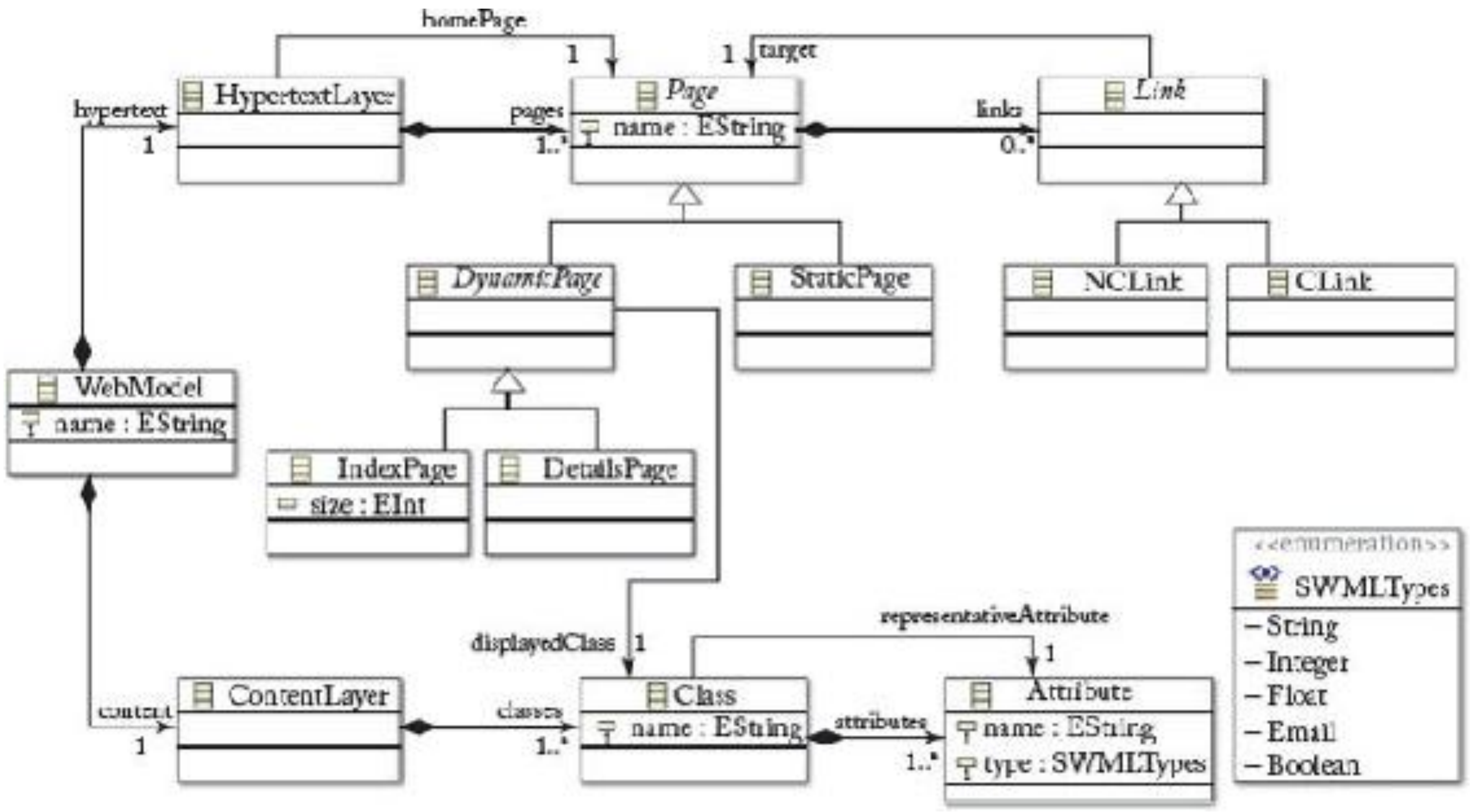
A sWML model consists of a content layer and a hypertext layer which reflects the previously mentioned purpose of the language.
A content model contains an unrestricted number of classes. Classes have a unique name (e.g., Tutorial) and multiple attributes (e.g., Tutorial.title). Attributes have an assigned name and type. For instance, permitted types are: String, Integer, Float, Boolean, and Email. For each class, one attribute has to be selected as the representative attribute which is not explicitly indicated in the concrete syntax to keep the syntax concise.
Hypertext models contain different kinds of pages, whereas each page has a name. Exactly one page is the homepage of the Web application. Later on, this page (and all directly linked pages of the homepage) can be accessed by all other pages of the Web application. Pages are subdivided into static and dynamic pages. Static pages represent only static content, e.g., a collection of useful links. In contrast, dynamic pages represent dynamically generated content coming from the database. Thus, a dynamic page always has a relationship to a class defining the type of the displayed instances. The relationship to the used classes is shown in parentheses under the name of the page. Dynamic pages are further subdivided into details pages and index pages having specific icons. Index pages show all instances of a class in terms of a list, e.g., a list consisting of all tutorials of the conference, showing only the representative attribute of the instances. In contrast, details pages always show exactly one instance, e.g., one tutorial, with all its attributes.
Links represent the navigation between pages by pointing from a source page to a target page. On the source page, the link is visualized to the user. Thus, a page usually knows its links pointing to other pages but it is unaware of incoming links. Two kinds of links are distinguished: (i) non-contextual links (NCLinks) are standard links, which do not transport any information, e.g., a link to a static page; and (ii) contextual links
(CLinks) transport information to the target page, e.g., to transfer data needed to compute the content of the target page, e.g., to select the instance shown by a details page.

sWML Concepts



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*

sWML Metamodel



Not Always Obvious

- For defining a concept, we have the following three options: define it as a class, attribute, or association
- Consider for instance the homepage concept in our example
- The first option (using a class) would result in an explicit class HomePage
  - However, if we would like to switch a homepage into a normal page, we have to delete the homepage, create a new page, and set all features using the previous values of the homepage for this new page
- The second option (using an attribute) would result in having a Boolean attribute in the class Page, e.g., called isHomepage
  - Thus, for each page we can dynamically decide if the page represents a homepage or not
  - We may need a modeling constraint which ensures that for each Web application, only one homepage exists

Not Always Obvious (cont.)

- Using the third option (an association), allows us to mark exactly one page as the homepage
  - Thus, we can dynamically change the homepage and do not need to add an additional well-formedness rule
- This discussion shows that, when deciding how to represent a modeling concept, one has to reason about the advantages and disadvantages of using classes, attributes, associations
- This decision has a major impact on the actual modeling possibilities influencing the modeling experience the users will have
- Thus, it is important to consider feedback from users to improve the modeling language by switching between different metamodeling patterns

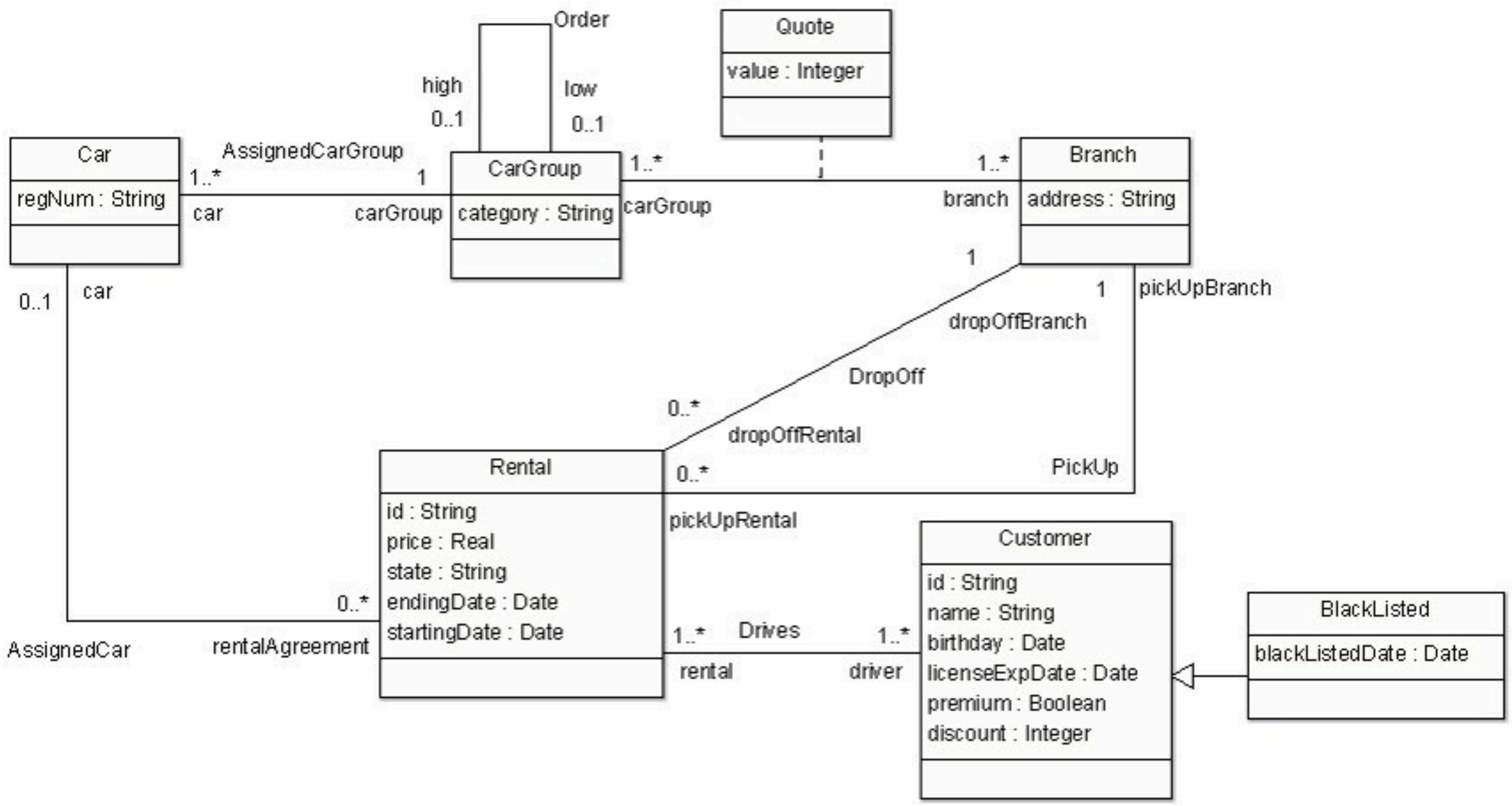
Modeling Constraints

- Several constraints may not be defined by current metamodeling languages
- Object Constraint Language (OCL) is employed to define additional constraints as so-called well-formedness rules
- These rules are implemented in OCL as additional invariants for the metamodel classes and have to hold for every model
- Thus, the constraints are defined on the metamodel and validated on the model level

TODO in Class

- Install OCL support in Eclipse
  - https://download.eclipse.org/modeling/mdt/ocl/updates/releases/latest
  - Choose “OCL Examples and Editors SDK”
- OCLinEcore tutorial
  - https://help.eclipse.org/latest/index.jsp?topic=%2Forg.eclipse.ocl.doc%2Fhelp%2FOCLinEcore.html
- Complete OCL tutorial
  - https://modeling-languages.com/ocl-tutorial/
  - https://modeling-languages.com/wp-content/uploads/2012/03/OCLChapter.pdf (full text)
- Continue working on the assignment (continue/finish modeling)

A Running Example



OCL Invariants

- Integrity constraints in OCL are represented as invariants defined in the context of a specific type
- This is the **context** type of the constraint
- Its body, the boolean condition to be checked, must be satisfied by all instances of the context type
- Example:

```
context Quote
inv QuoteOverZero:
self.value > 0
```

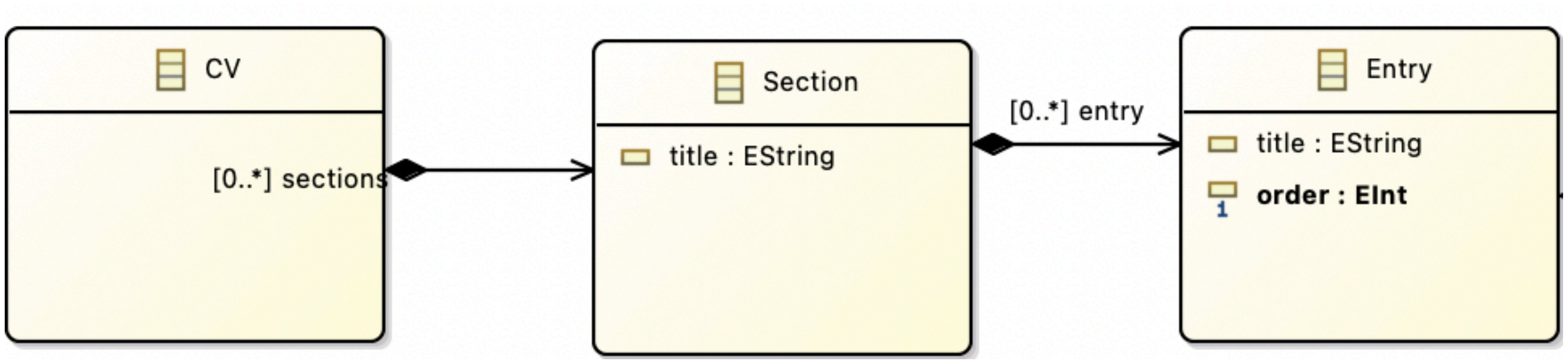
- `self` variable represents an arbitrary instance of Quote
- The dot notation is used to access the properties of the object

A More Complex Invariant

```
context BlackListed
inv NoRentalsBlackListed:
self.rental->forAll(r | r.startDate <
                        self.blackListedDate)
```

- In the context of BlackListed, we first retrieve all rentals (**navigation**)
- Then, we make sure that all of them (`forAll`) were created before the person was blacklisted (inequation)
- This is done by iterating on all related rentals and evaluating the date condition on each of them
- The `forAll` iterator returns true if and only if all elements of the input collection evaluate the condition to true

Another Example



Invariants in OCLinEcore

- Open the .ecore file with the OCLinEditor editor
- Inside a particular class:

```
invariant diffNames:
Section.allInstances()->forAll(s |
s <> self implies
s.title <> self.title);
```

Or

```
invariant diffNames:
Section.allInstances()->isUnique(title);
```

Another OCLinEcore Example

```
invariant select:

entry->select(e | e.order = 1)->size() = 1;
```

More Examples

- `invariant orderGreaterZero: order > 0;`
- `invariant orderUnique: Entry.allInstances()-> isUnique(order);`
- `invariant lowerName: self.title = self.title.toLowerCase();`
- `invariant diffNames: Section.allInstances()-> forAll(s | s <> self implies s.title <> self.title);`
- `invariant diffNameEntry: Entry.allInstances()-> forAll(e | e.title <> self.title);`
- `invariant size: self.entry->size() < 2;`
- `invariant select: self.entry-> select(e | e.order = 1)->size() = 1;`

TODO at Home

- Read more about OCL, if needed
- Continue working on the assignment (finish modeling/OCL)

Class 5
Developing your Own Modeling Language

TODO in Class

- Continue working on the assignment (finish modeling/OCL)

TODO at Home

- Finish assignment

Class 6
Developing Your Own Languages

Concrete Syntax

Visual & Textual

- Both are possible:
  - Graphical Concrete Syntaxes (GCS)
  - Textual Concrete Syntaxes (TCS)

GTS With Sirius

- We will use this framework to create a visual editor for the CV metamodel you have defined
- Tutorial: https://wiki.eclipse.org/Sirius/Tutorials/StarterTutorial#Start_creating_a_diagram_editor_with_Sirius
- Note:
  - in the step "Launch a new runtime from your Eclipse", you can ignore the VM argument;
  - the step "Install a sample model" and the remaining is done in the new eclipse instance;
  - the step "Start creating a diagram editor with Sirius" is the solution; you can use it to compare with yours;
  - the test step is done in the family sample model project;

Class 7
Model-To-Model
Transformations

Basics

- A M2M transformation is a program which takes one or more models as input to produce one or more models as output
- Most are one-to-one transformations, having one input model and one output model
- Example: the transformation of a class diagram into a relational model
- There are also situations where one-to-many, many-to-one, or even many-to-many transformations are required
- Example: a model merge scenario where the goal is to unify several class diagrams into one integrated view

Classification 1

- An **exogenous** transformation is between models from two different languages
- Example: the typical MDA scenario where a platform independent model, e.g., a UML model, is transformed to a platform specific model, e.g., a Java model
- **Endogenous** transformations are within models written with the same language
- Example: is model refactoring (similar as for code, a model may be subject to quality improvements which may be achieved by restructuring the models using a transformation)

Classification 2

- Exogenous model transformations are not only usable for **vertical** transformations (e.g. UML to Java)
- In those case the abstraction level of the input and output models are different
- Another usage is for **horizontal** transformations where the input and output models remain more or less on the same abstraction level
- Example: horizontal exogenous transformations are being used for realizing model exchange between different modeling tools, e.g., translating a UML class diagram to an ER diagram

Execution Paradigms

- There are **out-place** transformations for generating the output model from scratch
- Such transformations are especially suited to exogenous transformations
- There are **in-place** transformations for rewriting a model by creating, deleting, and updating elements in the input model
- This paradigm suits perfectly endogenous transformations such as refactorings

Tools

- https://projects.eclipse.org/projects/modeling.mmt
  - ATL
  - QVT
  - TGG
  - ETL

Class 8
Model-To-Text Transformations

Basics

- Used for automating several software engineering tasks such as the generation of code or documentation, task lists, etc.
- Actually, the main goal of model-driven software engineering is to get a running system out of its models
- M2T transformations are mostly concerned with code generation to achieve the transition from the model level to the code level
- Note one can also obtain test cases, deployment scripts, etc. from the models
- Formal code descriptions may also be derived which allows to analyze different properties of a system
- This is one of the major benefits of using models - they may be used constructively to derive the system as well as analytically to better explore or verify the properties of systems

Code Generation Through Programming Languages

Basics

- A code generator may be implemented as a program using the model API automatically generated from the metamodel to process the input models and print out code statements to a file using standard stream writers provided by APIs of the programming language used to implement the code generator
- It has some disadvantages

Disadvantages

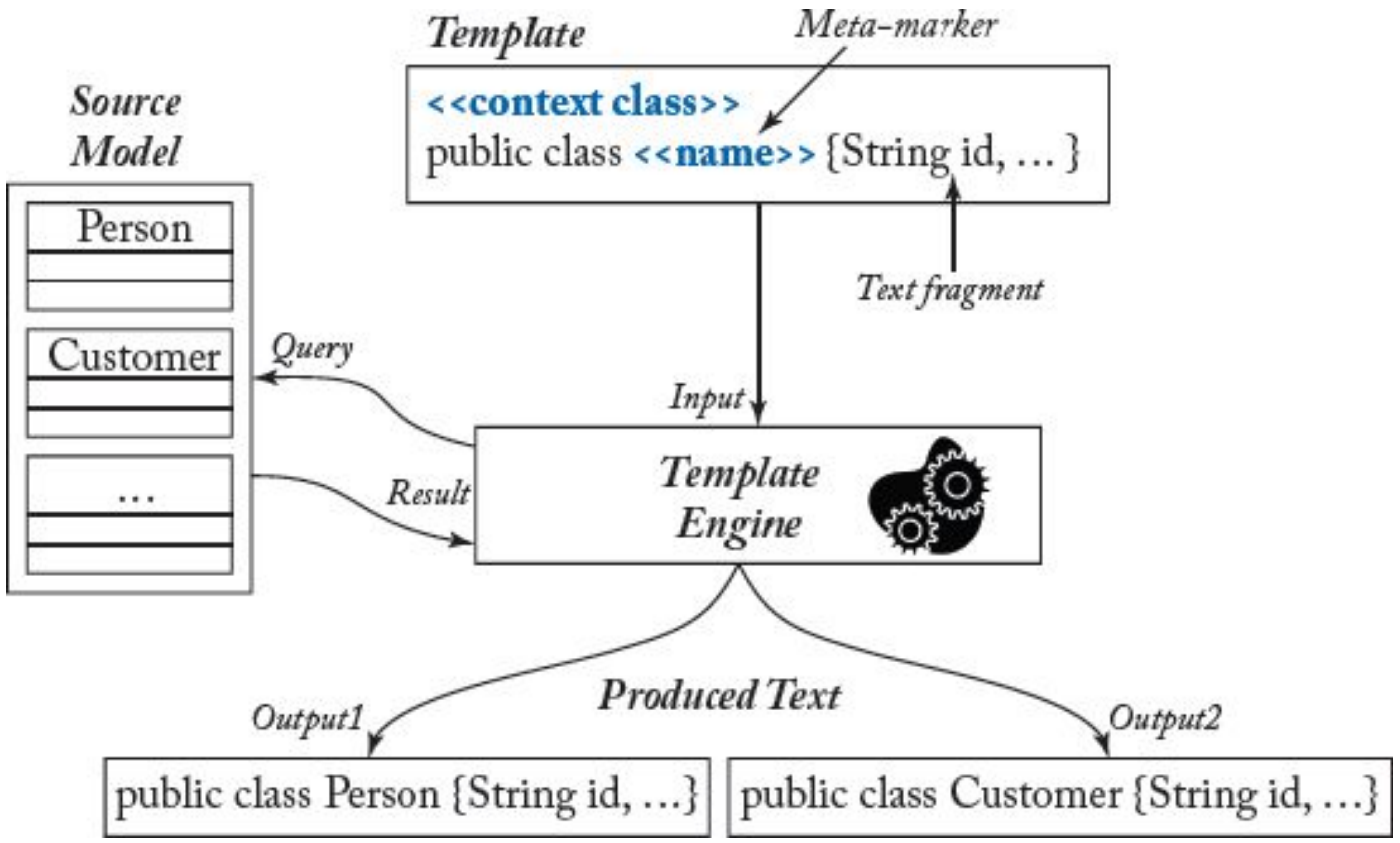
- There is **no separation of static code**, i.e., code that is generated in exactly the same way for every model element, e.g., the package definition, the imports, etc., **and dynamic code** which is derived from model information, e.g., class name, variable name
- The **structure** of the output is not easily graspable in the code generator specification
- There is **no declarative query language** for accessing model information available
  - Thus, many iterators, loops, and conditions as well as type casts unnecessarily lead to a huge amount of code
- **Missing reusable base functionality**: Code has to be developed for reading input models and persisting output code again and again for each code generator

Code Generation Through M2T Transformation Languages

Benefits (1)

- M2T transformation languages separate **static** and **dynamic** code by using a template-based approach
- A template can be seen as a kind of blueprint which defines static text elements shared by all artifacts as well as dynamic parts which have to be filled with information specific to each particular case
- A template contains:
  - **simple text fragments** for the static part
  - **meta-markers** for the dynamic part
- Meta-markers are placeholders and have to be interpreted by a template engine which processes the templates and queries additional data sources to produce the dynamic parts
- In M2T transformations, the additional data sources are models

Template-Based M2T



Benefits (2)

- Using templates allows us to explicitly represent the **structure of output text** within the template
- This is achieved by embedding the code for producing the dynamic parts of the output in the text representing the static part

Benefits (3)

- Within the meta-markers, we need to access the information stored in the models
- OCL is the choice to do this job in most M2M transformation languages
- Current M2T transformation languages also allow us to use OCL (or a dialect of OCL) for specifying meta-markers
- Other template languages not specifically tailored to models but supporting any kind of sources employ standard programming languages such as Java for specifying meta-markers

Benefits (4)

- Current M2T transformation languages come with tool support which allow us to directly read in models and to serialize text into files by just defining configuration files
- Thus, no tedious redefinition of model loading and text serializing has to be developed manually

Template-Based Languages and Technologies

- XSLT
- JET
- Xtend
- MOFScript
- Acceleo

Acceleo

- Practical relevance and mature tool support
- API supporting OCL
- Additional operations helpful for working with text-based documents in general, e.g., advanced functions for manipulating strings
- Powerful tooling such as an editor with syntax highlighting, error detection, code completion, refactoring, debugger, profiler
- Also, traceability API which allows us to trace model elements to the generated code and vice versa

How Acceleo Works

- It is necessary to create module to act as a container for templates
- The module also imports the metamodel definition for which the templates are defined
- This makes the template **aware** of the **metamodel** classes that can now be used as types in the template
- A template in Acceleo is always defined for a particular metamodel class
- In addition to the model element type, a pre-condition can be defined, e.g., to apply a template exclusively to model elements which are required to have a specific type as well as specific values

Acceleo Tags

- The Acceleo template language offers several meta-markers which are called tags:
- **Files**: there is a special file ***tag*** which is used to print the content created between the start and the end of the file tag for a given file
- **Control Structures**: There are tags for defining control structures such as loops (*for* tag) for iterating over collections of elements
- **Queries**: OCL queries can be defined (*query* tag). These queries can be called throughout the whole template and are used to factor out recurring code.

Acceleo Tags

- **Expressions**: There are general expressions for computing values in order to produce the dynamic parts of the output text
  - Expressions are also used to call other templates to include the code generated by the called templates in the code produced by the caller template
  - Calling other templates can be compared to method calls in Java
- **Protected Areas**: An important feature of M2T languages is to support projects where only partial code generation is possible
  - In particular, special support is needed to protect manually added code from file modifications in subsequent code generator runs
  - For this task, a special concept named protected areas has proven useful, which is supported by Acceleo via the *protected* tag
  - Protected areas are used to mark sections in the generated code that shall not be overridden again by subsequent generator runs
  - These sections typically contain manually written code

Example

```
[module generateJavaClass('http://smvcml /1.0')]
[query public getter(att : Attribute) : String = 'get '+att.name.toUpperFirst() /] [query public returnStatement(type:
String) : String = if type = 'Boolean'
then 'return true;' else '...' endif /] [template public javaClass(aClass : Class)]
[file (aClass.name.toUpperFirst ()+'.java', false, 'UTF-8')] package entities;
import java.io.Serializable;
public class [aClass.name/] implements Serializable {
[for (att : Attribute | aClass.atts) separator ('\n')] [javaAttribute(att)/]
[/for]
[for (op : Operation | aClass.ops) separator ('\n')] [javaMethod(op)/]
[/for]
} [/file]
[/template]
[template public javaAttribute(att : Attribute)]
  private [att.type/] [att.name/];
public [att.type/] [att.getter ()/]() { return [att.name/];
} ...
[/template]
[template public javaMethod(op : Operation)] public [op.type/] [op.name/]() {
// [protected (op.name)]
// Fill in the operation implementation here! [returnStatement(op.type)/]
// [/ protected] }
[/ template]
```

TODO

- Acceloo tutorial: https://wiki.eclipse.org/Acceleo/Getting_Started
- Implement the M2T transformation for the CV
- When implementing the M2T for the CV, when running the code generation, the wizard will ask for a CV file (Model)
- You can choose an xmi file with a CV
- To create one, you can open the ecore file, right hand side mouse click on the root of the model, and then choose Create Dynamic Instance and create an instance (we have done this before)