

Ingredients

1	Class Modeling	1
2	Class Diagrams Primer	2
3	Meta-Modeling	6
4	Meta-Modeling Hierarchy	8

Reading: This talk is devoted mainly to a quick refresh of class modeling. Please use any sources you are aware of to read up about this.

The basic introduction to EMF is available in [1].

EMF tutorial for the current release of Eclipse can be found at <http://www.vogella.de/articles/EclipseEMF/article.html>

It is worth trying to look at the MOF specification from OMG. It is available at <http://www.omg.org/mof/>; follow the link at the bottom of the page to the current spec. Get an overview, and read enough fragments, as to get an impression of what is entailed by a standardisation of a modeling language.

Otherwise I recommend poking online (wikipedia, omg, emf websites) to get a brief understanding of what is EMF, ecore, MOF, XMI and relations between these concepts.

Meta-modeling hierarchy is described in section 6.2 of [3] (please read). This hierarchy can be found also in *UML 2.0 Infrastructure Specification* from OMG, however that spec is not for those of faint heart.

If you never been exposed to class diagrams, then *please read* [2].

1 Class Modeling

Unified Modeling Language (UML) provides a complete set of notations for modeling software systems. This includes requirements, architecture, types, structure and behaviours.

In this course we use only (or primarily) UML *class diagrams* and only for structural modeling. Our view of UML is thus clearly restricted. I also mix-in some non-standard EMF extensions.

△ Definition 1. A *class* is an abstraction that specifies attributes of a set of concept instances (objects) and their relations to other concepts (objects).

Observe that in the above we do not mean that a class is a programming language concept, existing in an implementation in a programming language. In this course we first and foremost will be using classes to model real-world concepts, or more precisely *domain-level* concepts. These are often much more abstract than what classes are, for example, in Java or C#. This big leap from implementation to a conceptual application domain is necessary in order to obtain productivity gains promised by MDD.

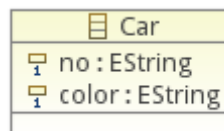
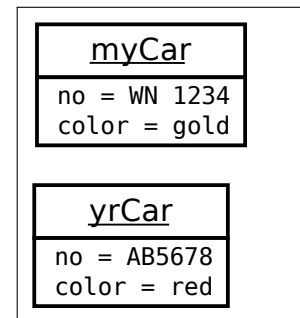
⚠ 2 Class Diagrams Primer

Both classes and objects are depicted by boxes with 3 compartments: names, attributes, and operations (we ignore the operations in the talk, but they are useful in some systems). When visualizing models, attributes and operations can be omitted if not essential.

The object diagram to the right shows us that there exist two cars with some attributes.

In the figure each object has two attributes. Object names (instance names) are underlined to distinguish them from classes.

In the simplest view classes are just types for objects. Our objects are of type Car:

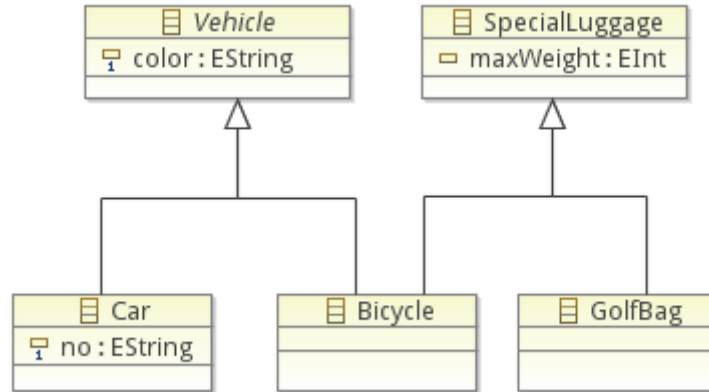


The above block represents class Car. The name is not underlined, and attributes have types, not values. They also have a *multiplicity* constraint (here simply “1”), which says that both attributes are mandatory for any instance of this class. By convention class names are capitalized, and instance names are not.

In order to make the discussion a bit more precise, we introduce a formalization of class diagrams in first order logics. For each class named C we will introduce a unary predicate of the same name $C(x)$, that holds for objects of type C . So for this example we have a predicate Car describing objects that conform to the type Car.

For simplicity of discussion we will ignore attributes in the logical formalization.

Generalization relation (also known as inheritance relation) says that instance sets of two classes are included: if class A generalizes class B, then each instance of B is also an instance of A.



every *Car* is a *Vehicle*, and so is every *Bicycle*. Similarly we have two kinds of special luggage. We sometimes say that a generalization relation expresses the *kind-of* relation ship (“a car is a kind of vehicle”).

We can again formalize the inheritance relation using first order logic formulae. If class *A* generalizes class *B*, then there holds an implication from the predicate representing the latter *B(x)* to the predicate representing the former, *A(x)*. For our example we have:

$$\forall x. \text{Car}(x) \rightarrow \text{Vehicle}(x) \quad (1)$$

$$\forall x. \text{Bicycle}(x) \rightarrow \text{Vehicle}(x) \quad (2)$$

and similarly for *Bicycle* and *GolfBag* with *SpecialLuggage*.

The above diagram shows an example of abstract class (*Vehicle*) – so class with no instances of its own (cursive title). We also show multiple inheritance: a bicycle is both a vehicle and a special luggage.

To formalize the notion of abstract classes we need to introduce a predicate representing instances of each class. For class *C* let's call the predicate $C^i(x)$. This is a stronger predicate than the predicate $C(x)$, which also covers instances of subclasses of *C*. Usually we have that for each class *C* the following holds:

$$\forall x. C^i(x) \rightarrow C(x) \quad (3)$$

but **not**

$$\forall x. C(x) \rightarrow C^i(x) \quad (4)$$

Now the class is abstract if it has no instances. So for example for the *Vehicle* class:

$$\forall x. \neg \text{Vehicle}^i(x) \quad (5)$$

All the diagrams presented above are actually ecore class diagrams (not UML class diagrams). Ecore is an implementation of MOF made within the Eclipse project, and MOF (which stands

for Meta-Object Facility) is a simple class diagramming language standardized by OMG, and used to define abstract syntax of UML languages, including the full blown UML class diagrams.

All the class diagrams above (and below) has been made using tools of the Eclipse Modeling language, and exported as models. In particular it means that EMF can process them automatically.

Already for such simple diagrams, we can generate editors of instances, or interactively derive instances and serialize them to xmi files (XMI is an OMG standard for model serialization).

NB. EMF also allows modeling of interfaces — this is done by adding an interface property to a class. The ecore diagram editor puts a little interface icon, next to the class name:



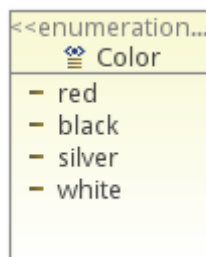
When EMF generates code interfaces are mapped to Java interfaces, while classes are mapped both to classes and interfaces. The latter is a simple pattern (a workaround, if you prefer) for Java's lack of multiple inheritance.

EMF provides simple types (for example **EString** used above), which are mapped to Java types during code generation.

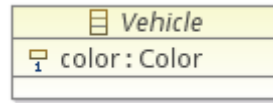
In class diagrams an attribute declaration can be followed by default value:

```
color :EString = 'red'
```

Enumerations often come useful in modeling, when we have a finite number of discrete simple values:



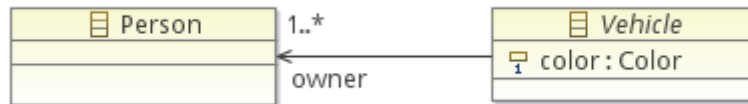
Enumerations can be used as types for attributes:



Now we only allow one of the four colors for vehicles.

It is also possible to introduce new basic types, by providing their Java implementations. Details in [1] (search for **EDataType** in the index).

An *association* represents a relation between instances of two classes. Note that association is qualitatively different than inheritance. We use associations to model all other kinds of relations between objects than *kind-of*.



The navigable name of the association is written on the “far end” — so `myCar.owner` gives the object representing the owner of `myCar`.

In the example the reference is also decorated with a multiplicity constraint `1..*`, meaning that a vehicle must have at least one owner. More than one owner are allowed (for modeling co-ownership). This also means that technically `myCar.owner` returns a collection, and not a single instance.

In EMF associations are unidirectional binary references. Unidirectional references can only be navigated in one direction. In UML references can be bidirectional and *n*-ary. For our purpose of meta-modeling, binary references are typically sufficient. Higher arity references can be always handled by creating an explicit class that will reify the association (similar to UML association classes). But, as already said, we rarely need it in language design.

Bidirectional references can be simulated using two unidirectional references. EMF allows to link two unidirectional references using the `EOpposite` property of the reference. In such case the generated code maintains links in both directions: whenever you add a link in one direction, the link in the other direction is created automatically. The mechanism is a bit complicated, and has shortcomings, so test well, when you rely on this. In particular, a reference cannot be `EOpposite` to itself,¹ and special care might be needed if you use references of multiplicity higher than 1.

¹This sounds a bit complicated, but in fact it appears in real domains for symmetric associations between objects of the same class. For example consider a class `Person` and unidirectional reference `marriedTo`. One way to model this in EMF would be to make the reference a bidirectional association, but this would require that it becomes an `EOpposite` of itself, which is not supported.

In first order logics, we can model references using binary predicates. For each reference r we introduce a predicate $r(x, y)$ relating the objects being linked. For instance for the owner reference above, we would introduce a predicate $\text{owner}(x, y)$. Associations (references) are typed so:

$$\forall x. \forall y. \text{owner}(x, y) \rightarrow \text{Vehicle}(x) \wedge \text{Person}(y) \quad (6)$$

Now, since the multiplicity constraint is 1..* on this reference we have additionally that:

$$\forall x. \exists y. \text{Vehicle}(x) \rightarrow \text{owner}(x, y) \quad (7)$$

Finally, references can be used to denote a *part-of* relation (in contrast to the *kind-of* relation of generalization). This is denoted using a black diamond on the owner side:



In this example we state that each vehicle contains 4 wheels as its integral part. This means that a vehicle instance without 4 wheels cannot exist (such an instance is not well-formed).

(Q. Incidentally, the example is silly, as it also means that every bike has 4 wheels. Do you know why?)

The black-diamond semantics also means that every object in such a relation can only have one owner — so there could not be cars sharing wheels. It also imposes a syntactic well-formedness rule: objects cannot be owners of themselves—effectively, directed subgraphs of any class diagram consisting of reference edges labelled with black diamonds have to be trees (or forests).

The black-diamond associations are interchangeable called “compositions”, “aggregations”, and “part-of relations”.

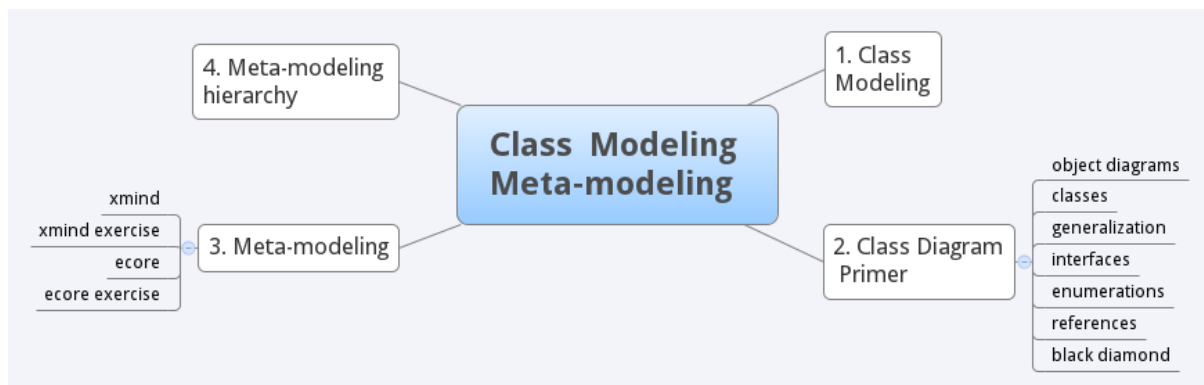
3 Meta-Modeling

In order to be able to process languages automatically we need to provide formal definitions of them. The most popular tools (EMF, etc) allow defining modeling languages (or at least defining their abstract syntax) using class diagrams, or very similar meta-modeling languages (like KM3). Then they can generate parsing and persistence code that treats models (mograms) in such defined languages as object diagrams.

⚠ **Definition 2.** *Meta-modeling* is the practice of modeling syntax of other languages using class diagrams. So meta-modeling is modeling of modeling languages, and a meta-model is a model of a modeling-language.

Figure 1 shows a meta-model of one of the classic examples—a mind-mapping language.

Small exercise: Take a piece of paper and draw an instance diagram representing the following mindmap:



Now that we know how to describe abstract syntax of languages using class diagrams, we can describe class diagrams themselves. It turns out that one can (retro-actively) provide an ecore model representing ... the ecore abstract syntax (and MOF model representing MOF syntax, and a UML model representing UML syntax). Figure 2 shows this meta-model.

⚠ This has actually sometimes led to confusion that some languages are defined in themselves, for example 'UML defined in UML' or 'ecore is defined in ecore'. This is not true — circular definitions of languages are not possible.

The practice of modeling the language in itself could better be called *bootstrapping*. Indeed, it is really akin to the practice of programming language designers, who tend to implement compilers for a new language in the language itself, as the first serious maturity test. For example your favourite java compiler is most likely implemented in java.

Of course, a bootstrapped language first needs a compiler or an interpreter implemented in another language (which already has a compiler or interpreter). Similarly for modeling languages: the first definition uses an existing language, or simply natural language description. The bootstrap-like self-definition comes later.

Another exercise: Try to take any of the tiny ecore models above and draw its abstract syntax as an instance of the ecore meta-model above. You need not to complete the exercise, but do enough, to understand the structure of the meta model. It makes sense to check out the simplified meta-model of ecore in chapter 2 of [1]. This model has only 4 classes, and makes it quite easy to understand the main idea behind representation of ecore in ecore.


4 Meta-Modeling Hierarchy

OMG (Object Management Group) organized models and languages in a hierarchy of abstraction layers, also known as the OMG modeling architecture. This is exemplified in Fig. 3 using our mind-map language. At the very top level we have the ecore language (M3) which allows describing class diagrams. Instances of this language are class diagrams at level M2. A class diagram describing an abstract syntax (the meta-model) of the mindmap language belongs here. Note the conformance relations between languages (and instance-of relations) between instances at level M2 and classes at level M3.

One level below, at M1, we have concrete models in the mindmap language, here shown using notation resembling object diagrams (so their abstract syntax is shown). The model at M1 describes mindmap notes of a concrete lecture. Again, note the conformance (and instance-of) arrows crossing the two layers, M1 and M2. At the bottom level of the hierarchy (M0) we have the physical world (or the domain) that is described by the models at M1. Here at M0 we find a concrete lecture, which has been abstracted by notes at M1.

Figure 4 shows the same architecture but using the UML as an example, instead of the mind-map language. Design of the UML has actually been the main rationale for organizing this architecture. Since then, it has proven very useful for understanding layers in design of domain specific languages, like our mindmap.

The UML hierarchy is a bit tricky. Note that the entire UML 2.0 is in M2 in the Figure. But UML 2.0 contains conformance as part of the language. It contains both class diagrams and object diagrams. The figure actually shows how UML tools are implemented to support this, using general language processing stack, like EMF. The instance-of relation between a Class and Instance becomes a regular reference (association) in the implementation—see the arrow labelled classifier in M2.

Various levels of meta-modeling can be set at various abstraction levels. For example a meta-model of ecore is very abstract. A meta-model of UML expressed in ecore is more concrete. A model of video-rental application expressed in UML is even more concrete. An instance of  that model, an actual data collection of videos is very concrete.

The final example (Fig. 5) shows the same architecture, as realized by W3C technology stack for structured data XML. At the top level we have the XML Schema Language that conforms to its specification in the XML Schema Language (again, after the language has been designed, it has been described in itself and the corresponding xsd file has been published). At level M2 we have XML Schema for concrete languages. Here, I use the XMI language as an example. At level M1 we have concrete XML files conforming to the schema of M2. In the example I use the mindmap.ecore file that conforms to the XMI schema for model representation in XML format.

The familiar XML stack has very similar aims to meta-modeling languages: describing structures and data in a standard manner. The main differences are that (1) XML documents are not really meant to be processed by humans, and (2) that XML processing stays largely on the level of strings or trees. The tools for processing models usually stay at a higher abstraction

level. As we will see in the later lectures, models are processed using languages that support standard object-oriented programming model.

In DSL based development, your DSL fits into level M2, replacing UML, and concrete models in the DSL are at level M1 — it depends on the concrete project whether they have further instances or not. Usually they do not.

References

- [1] Frank Budinsky, David Steinber, Ed Merks, Raymond Ellersick, and Timothy J. Groose. *Eclipse Modeling Framework*. Addison-Wesley, 2004.
- [2] Martin Fowler. *Uml Distilled: A Brief Guide to the Standard Object Modeling Language*. Addison-Wesley Professional, 2004.
- [3] Thomas Stahl and Markus Völter. *Model-Driven Software Development*. Wiley, 2005.

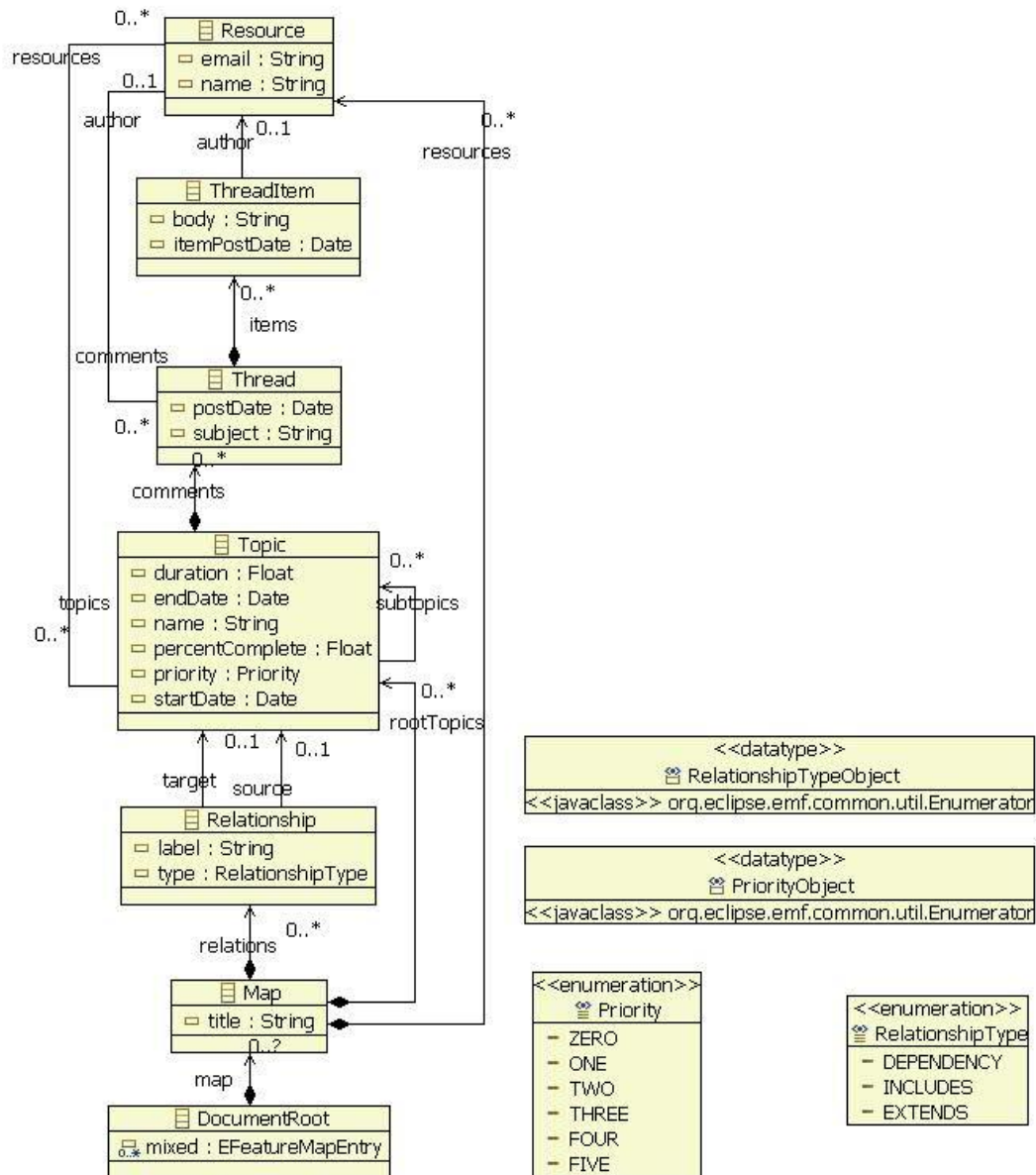


Figure 1: Meta-model of a mind-mapping language.

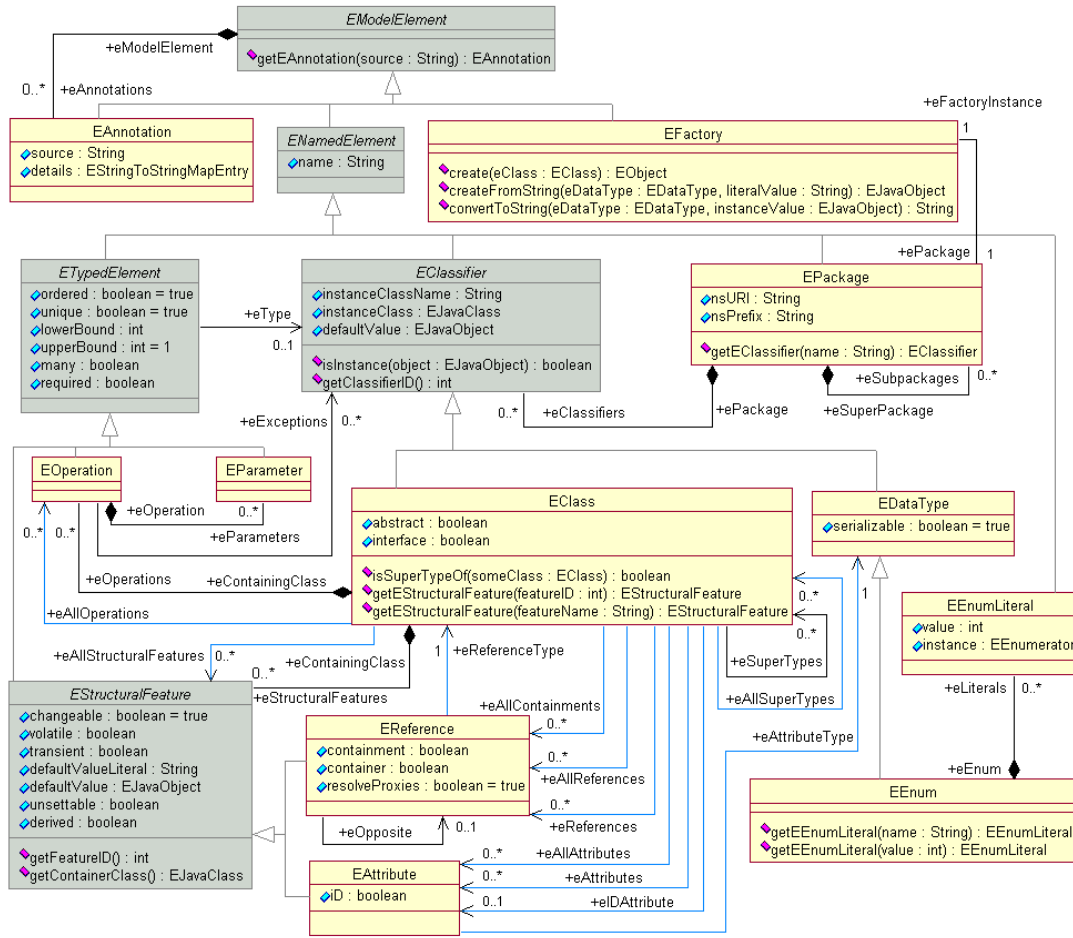


Figure 2: An.ecore meta-model of.ecore.

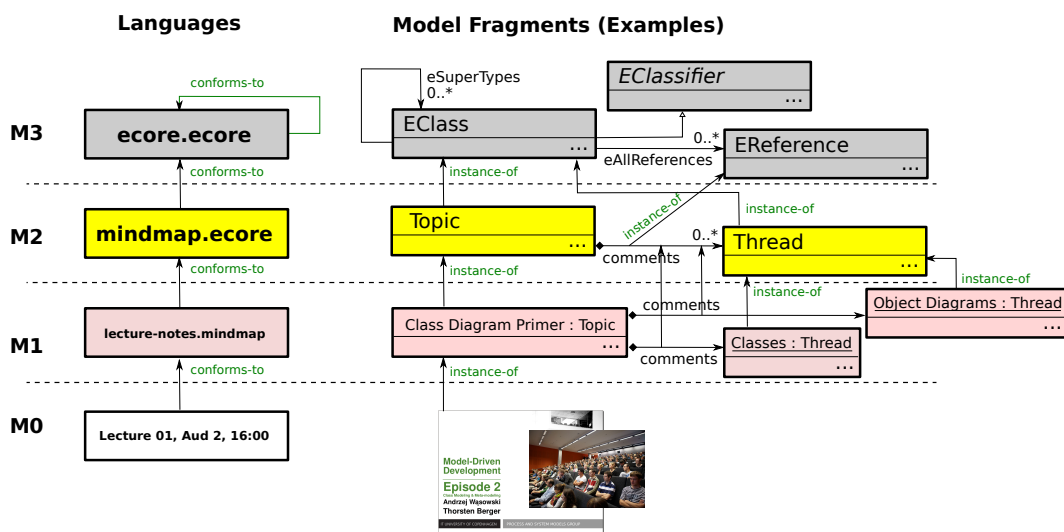


Figure 3: Metamodeling-hierarchy illustrated using the mindmap example language.

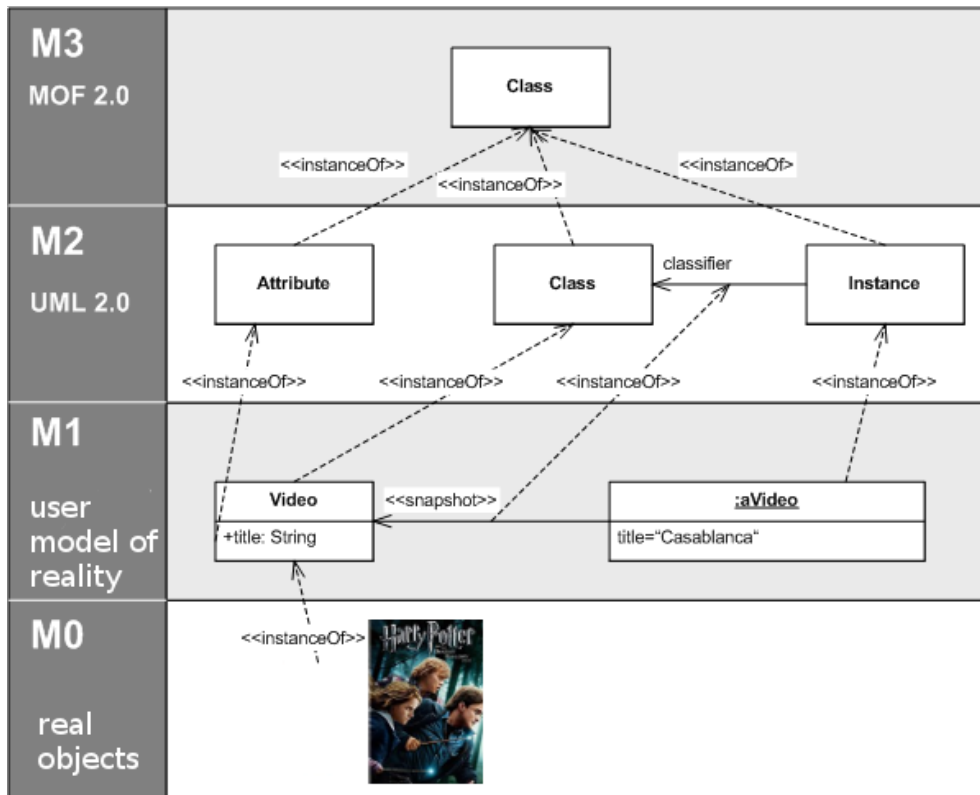


Figure 4: Metamodeling-hierarchy illustrated using UML (source: Wikipedia).

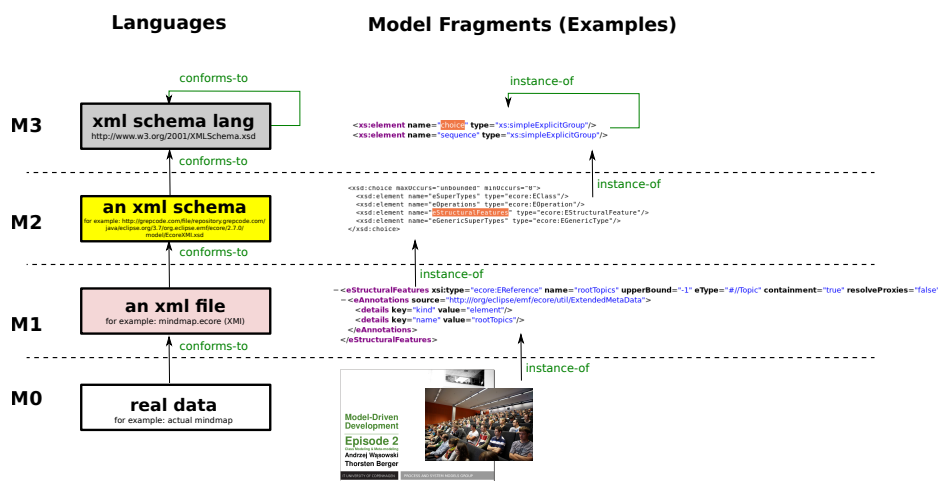


Figure 5: Metamodeling-hierarchy illustrated using XML technology stack.