

**KM3:**  
**Kernel MetaMetaModel**

**Manual**  
**- *version 0.3* -**

**August 2005**

**by**  
***ATLAS group***  
***LINA & INRIA***  
***Nantes***

# Content

<b>1</b>	<b>Introduction .....</b>	<b>3</b>
<b>2</b>	<b>KM3 Structure.....</b>	<b>3</b>
2.1	<i>Metamodel .....</i>	<i>3</i>
2.2	<i>Constraints .....</i>	<i>4</i>
<b>3</b>	<b>KM3 Syntax .....</b>	<b>5</b>
3.1	<i>KM3 File .....</i>	<i>6</i>
3.2	<i>Package.....</i>	<i>6</i>
3.3	<i>Class.....</i>	<i>6</i>
3.4	<i>Multiplicities .....</i>	<i>6</i>
3.5	<i>Attribute .....</i>	<i>6</i>
3.6	<i>Reference .....</i>	<i>7</i>
3.7	<i>Enumerations .....</i>	<i>7</i>
3.8	<i>Primitive Data Types .....</i>	<i>7</i>
<b>4</b>	<b>Current Status &amp; Implementation .....</b>	<b>8</b>
<b>5</b>	<b>Summary .....</b>	<b>8</b>
<b>6</b>	<b>References .....</b>	<b>9</b>
<b>Appendix A:</b>	<b>XML Metamodel Example .....</b>	<b>10</b>
<b>Appendix B:</b>	<b>KM3 Metametamodel .....</b>	<b>11</b>

## 1 Introduction

KM3, the Kernel MetaMetaModel, provides a textual concrete syntax that eases the coding of metamodels. Its syntax is simple, straight forward and has some similarities with the Java notation. A *.km3* file can be transformed into a metamodel and serialized in XML [2]. Present tools support Eclipse EMF Ecore [1] and MOF 1.4 [4] (using Netbeans MDR).

KM3 abstract syntax (metamodel) is based on Ecore and eMOF 2.0 [3]. It is defined in KM3 in the file KM3.km3, which is given in Appendix B.

## 2 KM3 Structure

KM3 provides semantics for metamodel descriptions. Not surprisingly, KM3 resembles Ecore terminology and has the notion of package, class, attribute, reference and primitive data type. Section 2.1 is dedicated to the description of the structure KM3 metamodel, whereas Section 2.2 provides the list of non-structural constraints (e.g. those that are not directly encoded within the KM3 metamodel) that have to be fulfilled by KM3-defined metamodels.

### 2.1 Metamodel

Figure 1 provides a description of the KM3 metamodel. Its corresponding textual description in the KM3 format is also provided in Appendix B. Note that the KM3 metamodel conforms to its own semantics.

A KM3 **Metamodel** is composed of **Package** elements. A **Package** contains some abstract **ModelElement** entities (**TypedElement**, **Classifier**, **EnumLiteral** and **Package** entities, since a **Package** is itself a **ModelElement**). Each **ModelElement** is characterized by its *name*. Note that a **ModelElement** is not necessarily contained by a **Package**. A **ModelElement**, as well as a **Metamodel**, inherits from the abstract **LocatedElement** entity. This last defines a *location* attribute that aims to encode, in a string format, the location of the declaration of the corresponding element within the *.km3* source file.

A **Classifier** can be either an **Enumeration**, a **DataType** or a **Class**. An **Enumeration** is composed of **EnumLiteral** elements. Every **EnumLiteral** is contained by an **Enumeration**. The **DataType** element enables to define primitive data types. Finally, the **Class** element defines a boolean *isAbstract* attribute that enables to declare abstract classes. Moreover, a **Class** can be extended through direct *supertypes* (**Class** elements).

A **Class** is composed of a set of abstract **StructuralFeature** elements. A **StructuralFeature** inherits from the abstract **TypedElement** entity. This entity defines the *lower*, *upper*, *isOrdered* and *isUnique* attributes. The two first attributes defines the minimal and maximal cardinality of a **TypedElement**. The *isOrdered* and *isUnique* Boolean attributes encode the fact that the different instances of the **TypedElement** are respectively ordered and unique. A **TypedElement** obviously has a *type*, which corresponds to a **Classifier** element. Note that a **StructuralFeature** has a unique *owner* of type **Class**. This means that, although a **StructuralFeature** is also a **ModelElement**, it is contained by a **Class**, instead of a **Package**.

A **StructuralFeature** is either a **Reference** or an **Attribute**. The **Attribute** element enables to define attribute of a class. A **Reference** defines the boolean *isContainer* attribute that encodes the fact that the elements pointed by the reference are contained by this last. A **Reference** can also have an *opposite* reference.

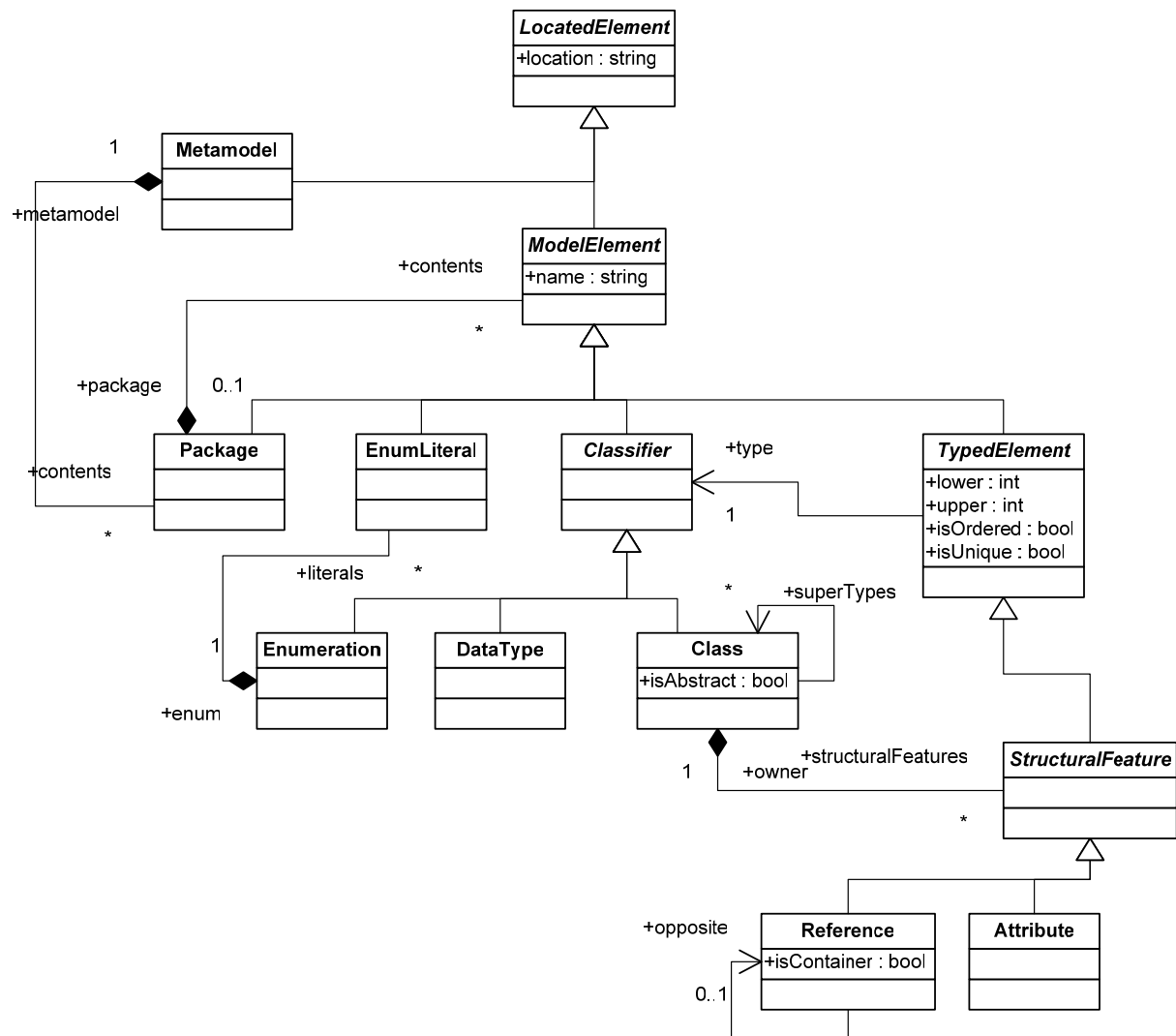


Figure 1. The KM3 metamodel

## 2.2 Constraints

Section 2.1 defines a number of structural constraints on KM3 metamodels. However, in the same way additional constraints can be specified on a MOF metamodel [3] by means of the OCL language [5], KM3 metamodels have to respect a set of non-structural additional constraints. Unlike MOF, KM3 does not provide a mean to express additional constraints (such as OCL constraints) in the same file that the metamodel definition. However, OCL constraints can be specified and implemented in ATL. We therefore provide here an informal list of the non-structural constraints that have to be respected by KM3 metamodels:

- A **Package** *name* has to be unique within its **Package**.
- A **Classifier** has to belong to a **Package** (it can not be defined independently).
- An **EnumLiteral** has to belong to a **Package** (it can not be defined independently).
- A **Classifier** *name* has to be unique within the **Package** it belongs to.

- A **Package** can only contain **Package** and **Classifier** elements through its *contents* reference.
- A **Class** is not allowed to be its own direct or indirect *supertype*. In other words, inheritance paths should define no cycle.
- The *name* of a **StructuralFeature** has to be unique in the **Class** it belongs to, as well as in the *supertypes* of this **Class**.
- The *opposite* **Reference** of a **Reference** must have an *opposite*.
- The *opposite* of the *opposite* of a **Reference** A must be A.
- The *type* of the *opposite* of a **Reference** has to be the *owner* of the **Reference**. This means that an opposite reference has to point (through the *type* reference, inherited from **TypedElement**) to the very same Class of the original **Reference** (the reference should not, for instance, point to a *supertype* of this class).
- The *lower* attribute of a **TypedElement** cannot be lower than 0.
- The *upper* attribute of a **TypedElement** has to be 1) unbounded or 2) greater or equal to 1.
- The *upper* attribute of a **TypedElement** cannot be lower than its *lower* attribute.
- The *isOrdered* attribute of a **TypedElement** cannot be true if the *upper* value is 1.
- The *type* of a **Reference** must be a **Class**.

A tool dedicated to the checking of these non-structural constraints, has been made available as an ATL (Atlas Transformation Language) transformation. The KM3 to Problem transformation therefore produces, from a KM3 metamodel, a Problem model that encodes all the non-structural constraints that are not fulfilled by the input KM3 metamodel [6]. This ATL transformation therefore provides an OCL-based expression of the constraints listed in this section.

### 3 KM3 Syntax

In this section, we will use a simple XML metamodel as an example (see Figure 2).

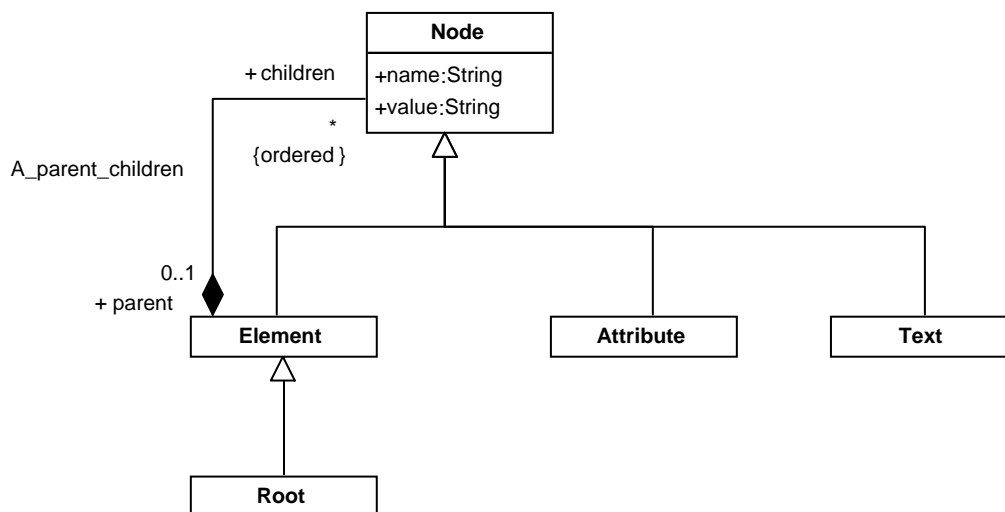



Figure 2. A simple XML metamodel

	ATL Documentations	
	KM3 User Manual	Date 23/08/2005

This XML metamodel is used to illustrate the different KM3 elements that will be explained in the following.

### 3.1 KM3 File

A KM3 file defines a set of packages. In each file there must be one package that has the same name as the file name (without extension) and there has to be one package that declares the primitive data types that are used in this file. Each KM3 file has the “.km3” extension.

Please note:

- Always check the correspondence of the naming of your file with the naming of your main package
- If a name within your model collides with a KM3 keyword (e.g. package, class) you have to put it in quotation marks "".
- The KM3 parser is case sensitive.
- Comments start with double minus "--".

### 3.2 Package

A package contains a set of classes that form a consistent model. The following code snippet shows the definition of the package “XML”:

```
package XML {
-- your class definitions go here
}
```

### 3.3 Class

A class consists of attributes and references. It can be extended from superclasses. The following code snippet shows the class “Element” and its superclass “Node”:

```
class Node {
-- attributes and references
}

class Element extends Node {
-- attributes and references
}
```

### 3.4 Multiplicities

Attributes and references have multiplicities. If not specified, the multiplicity is one. In general the multiplicity is specified with a lower and an upper value:

- [0-\*] or [\*] are used to declare multiplicity zero to many;
- [1-1] declares multiplicity one and is assumed if no multiplicity is declared;
- [1-\*] declares multiplicity one to many.

If the upper multiplicity value is greater than one, the attribute or the reference can be ordered. This is declared with the keyword “ordered”.

### 3.5 Attribute

An attribute has a name, a multiplicity and a type. The following code snippet shows the String attribute “name”:

```
attribute name : String;
```

### 3.6 Reference

A reference has a name, a multiplicity and a type. It may have an opposite reference and can be declared as being a container. The following code snippet shows the opposite references “parent” in the class Node and “children” in the class Element. While the class Node has either no or one “parent” reference of the type Element, the nodes reference in the class Element may have zero to n “children” of the type Node. Since XML elements contain sets of ordered Nodes, the element declaration is complemented as follows:

```
abstract class Node {
    reference parent[0-1] : Element oppositeOf children;
    -- further attributes and references
}

class Element extends Node {
    reference children[*] ordered container : Node oppositeOf parent;
}
```

### 3.7 Enumerations

In KM3 you can define your own enumeration types:

```
enumeration Color {
    literal green;
    literal blue;
    literal orange;
}
```

and use these for defining the attributes of your classes:

```
abstract class Node {
    attribute colorOfNode : Color;
    -- further attributes and references
}
```

Note that, according to the KM3 metamodel, enumerations have to be defined within packages.

### 3.8 Primitive Data Types


The primitive data types supported by KM3 are String, Boolean, Integer and Double. The KM3 engine has to know the primitive types that are used in a KM3 file. The recommended way of using primitive data types is to define an additional package:

```
package XML {
    -- your class definitions go here
}

package PrimitiveTypes {
    datatype Boolean;
    datatype Integer;
    datatype String;
}
```

Please note:

- If you forget the PrimitiveTypes package (and do not use an alternative way of importing them) your file cannot be translated to an XMI file (supposing that you use primitive types).

	ATL Documentations	
	KM3 User Manual	Date 23/08/2005

## 4 Current Status & Implementation


The reference definition of the KM3 metamodel, which is the one actually considered by KM3 dedicated tools, is provided in Appendix B. With respect to the graphical representation provided in Figure 1, this reference KM3 definition is not exact, since three entities (Classifier, TypedElement, and StructuralFeature) are not declared abstract although they should be.

Current reference implementation of the KM3 parser also suffers from another limitation: it currently manages a single namespace by metamodel. This implies that the name of a Classifier has to be unique in the whole metamodel, and not in its only Package. Future releases of the KM3 parser will correct this limitation.

## 5 Summary

KM3 is a convenience notation that has been defined in the ATLAS team to facilitate and speed up the creation and management of metamodels. This is not a notation proposed for normalization but just a handy tool defined for ATL users. Many conversion bridges between KM3 and standard notations such as XMI have been built.



	ATL Documentations	
	KM3 User Manual	Date 23/08/2005

## 6 References

- [1] Budinsky, F., Steinberg, D., Ellersick, R., Grose, T. Eclipse Modeling Framework. Addison Wesley Professional. ISBN: 0131425420, 2004.
- [2] OMG/XMI XML Model Interchange (XMI) OMG Document AD/98-10-05, October 1998.
- [3] OMG/MOF 2.0 Core Final Adopted Specification Document. ptc/03-10-04, 2004.
- [4] OMG/MOF Meta Object Facility (MOF). Version 1.4. formal/2002-04-03, 2002.
- [5] OMG/OCL Specification, ptc/03-10-14. October 2003. Available from [www.omg.org](http://www.omg.org).
- [6] KM3 to Problem ATL transformation. Documentation available at [http://dev.eclipse.org/viewcvs/indextech.cgi/~checkout~/gmt-home/subprojects/ATL/ATL\\_examples/KM32Problem/ExampleKM32Problem%5Bv00.01%5D.pdf](http://dev.eclipse.org/viewcvs/indextech.cgi/~checkout~/gmt-home/subprojects/ATL/ATL_examples/KM32Problem/ExampleKM32Problem%5Bv00.01%5D.pdf).

## Appendix A: XML Metamodel Example

The XML metamodel (see Figure 3) has been chosen to illustrate KM3.

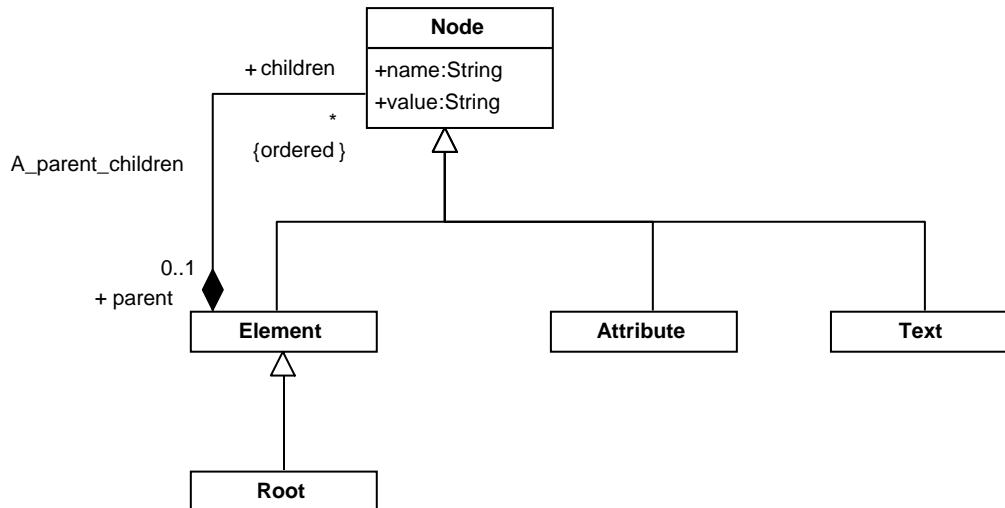


Figure 3. A simple XML metamodel

To obtain an Ecore-XML formatted file of the XML model (see Figure 3) the following KM3 code has been written.

```

package XML {
    abstract class Node {
        attribute name : String;
        attribute value : String;
        reference parent[0-1] : Element oppositeOf children;
    }

    class Attribute extends Node {
    }

    class TextNode extends Node {
    }

    class Element extends Node {
        reference children[*] ordered container : Node oppositeOf parent;
    }

    class Root extends Element {
    }
}

package PrimitiveTypes {
    datatype Boolean;
    datatype Integer;
    datatype String;
}
  
```

## Appendix B: KM3 Metamodel

The following .km3 file defines KM3:

```
package KM3 {
  abstract class LocatedElement {
    attribute location : String;
  }

  abstract class ModelElement extends LocatedElement {
    attribute name : String;
    reference "package" : Package oppositeOf contents;
  }

  class Classifier extends ModelElement {}

  class DataType extends Classifier {}

  class Enumeration extends Classifier {
    reference literals[*] ordered container : EnumLiteral oppositeOf enum;
  }

  class EnumLiteral extends ModelElement {
    reference enum : Enumeration oppositeOf literals;
  }

  class Class extends Classifier {
    attribute isAbstract : Boolean;
    reference supertypes[*] : Class;
    reference structuralFeatures[*] ordered container : StructuralFeature
oppositeOf owner;
  }

  class TypedElement extends ModelElement {
    attribute lower : Integer;
    attribute upper : Integer;
    attribute isOrdered : Boolean;
    attribute isUnique : Boolean;
    reference type : Classifier;
  }

  class StructuralFeature extends TypedElement {
    reference owner : Class oppositeOf structuralFeatures;
  }

  class Attribute extends StructuralFeature {}

  class Reference extends StructuralFeature {
    attribute isContainer : Boolean;
    reference opposite[0-1] : Reference;
  }

  class Package extends ModelElement {
    reference contents[*] ordered container : ModelElement oppositeOf
"package";
    reference metamodel : Metamodel oppositeOf contents;
  }
}
```

```
    }  
  
    class Metamodel extends LocatedElement {  
        reference contents[*] ordered container : Package oppositeOf  
metamodel;  
    }  
}  
  
package PrimitiveTypes {  
    datatype Boolean;  
    datatype Integer;  
    datatype String;  
}
```