

Generic Model Refactorings

**Naouel Moha, Vincent Mahé,
Olivier Barais, and Jean-Marc Jézéquel**

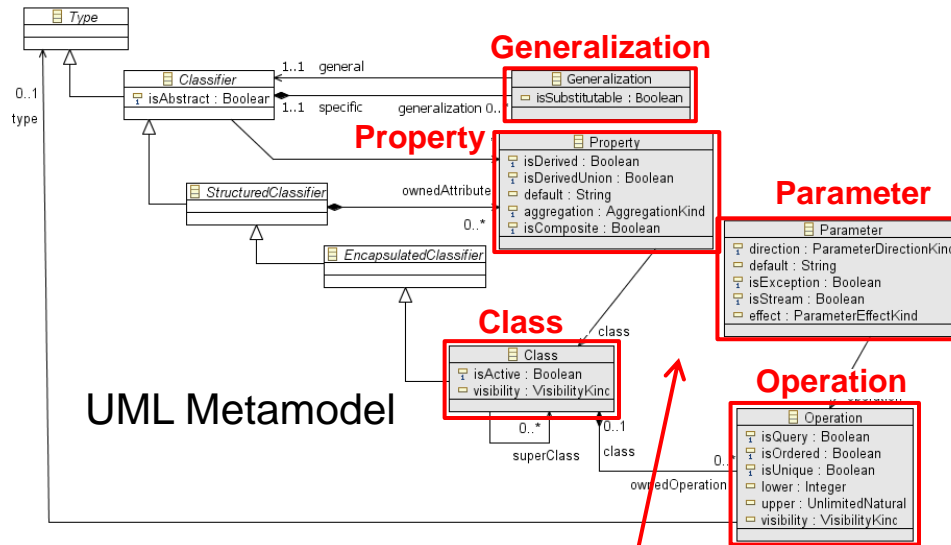
Triskell Team, INRIA Rennes – Bretagne Atlantique/IRISA, Université Rennes 1, France

MODELS'09

October 4 - 9, 2009



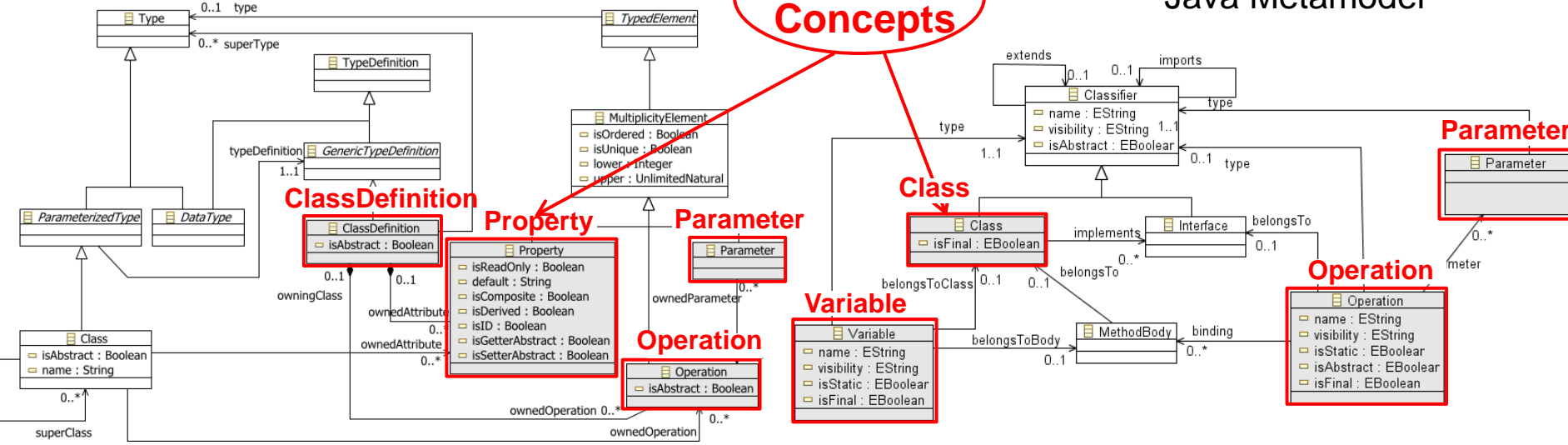
Context



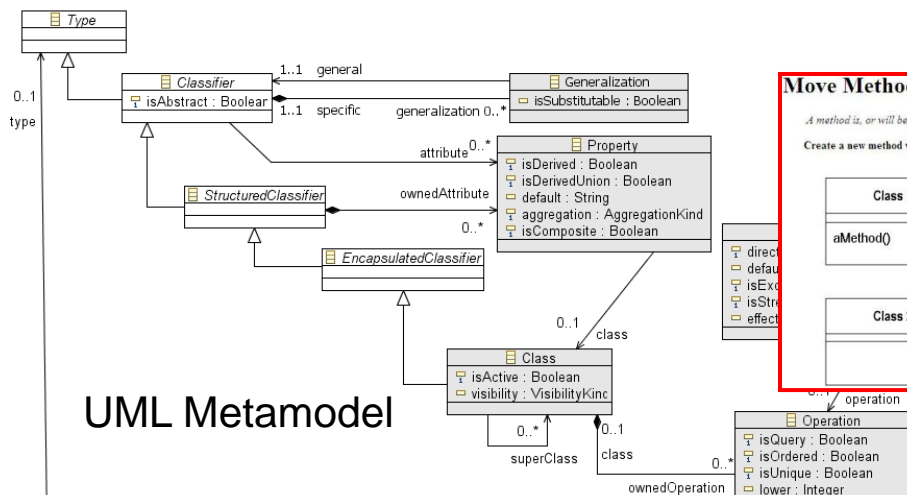
Common Concepts

MOF Metamodel

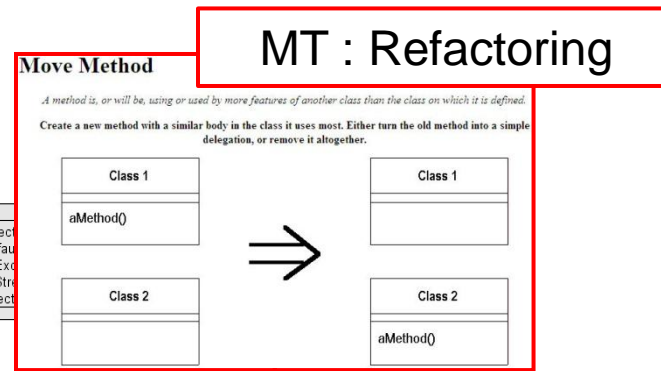
Java Metamodel



Context

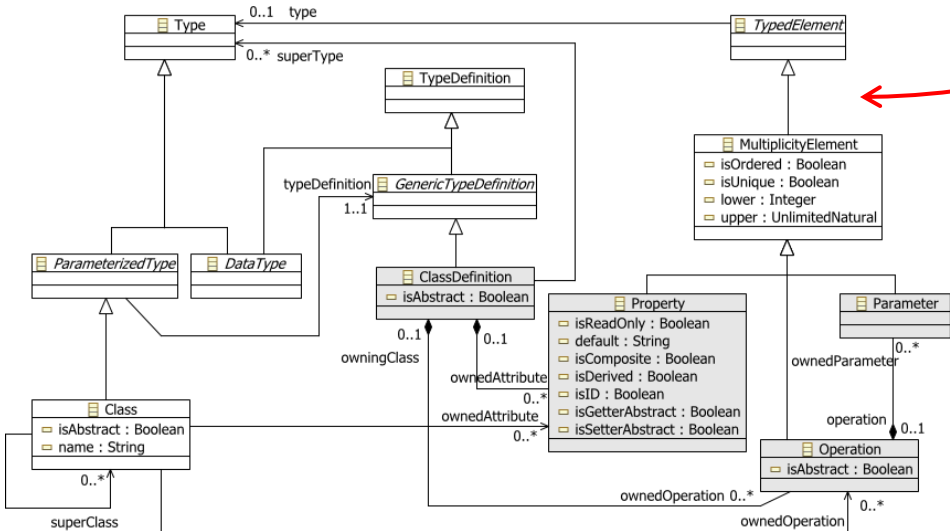


UML Metamodel

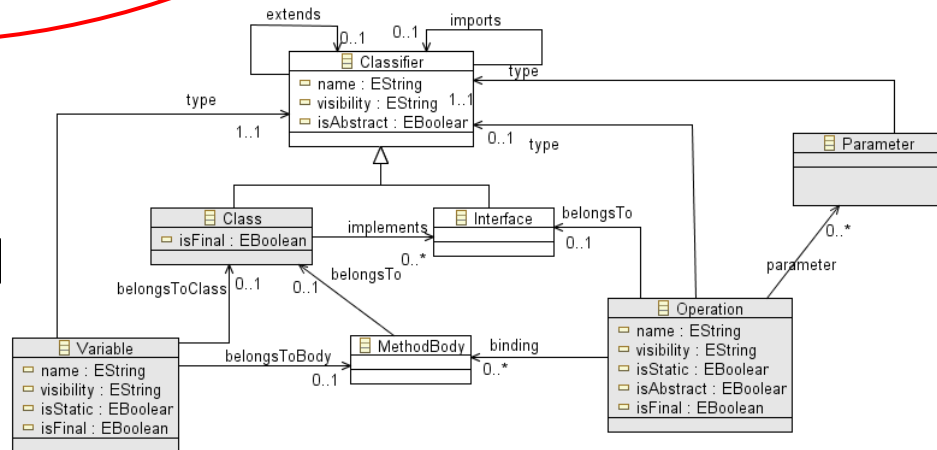


The metamodels may be structurally different

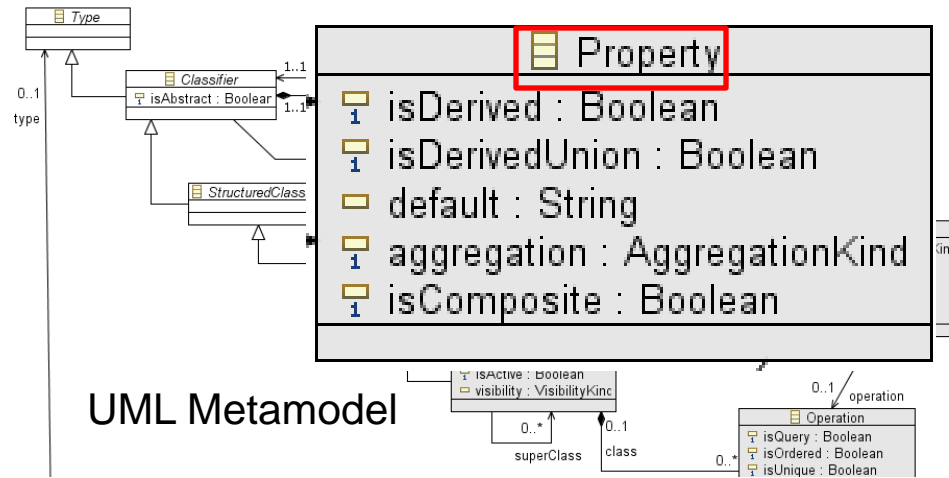
MOF Metamodel



Java Metamodel



Context

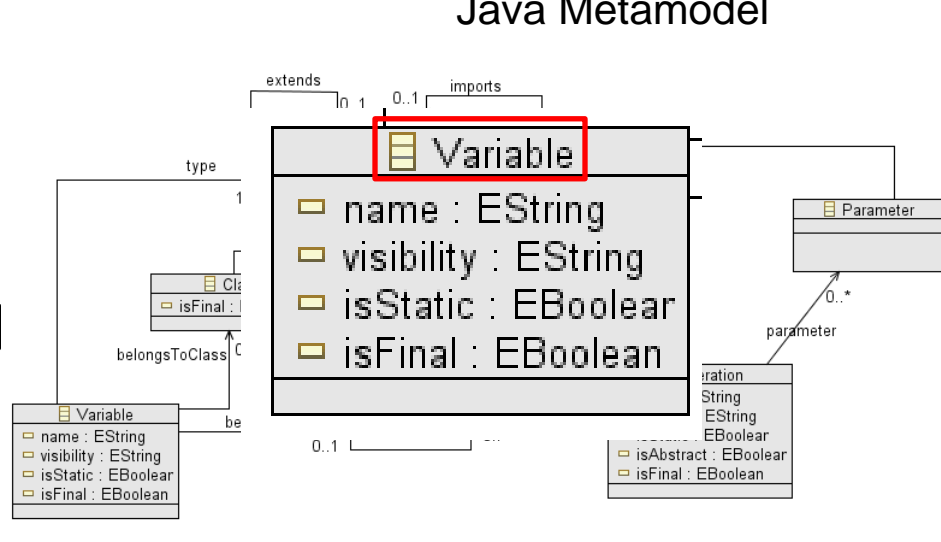
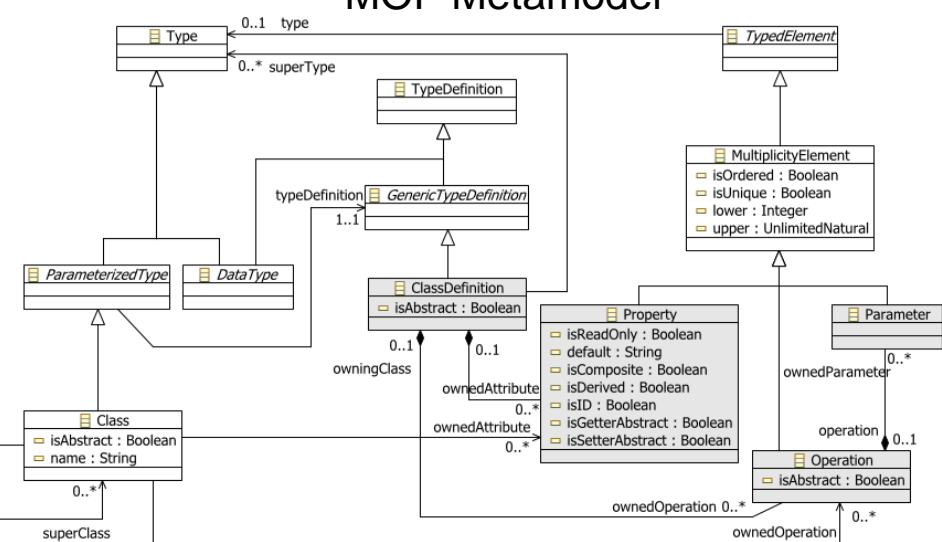


UML Metamodel

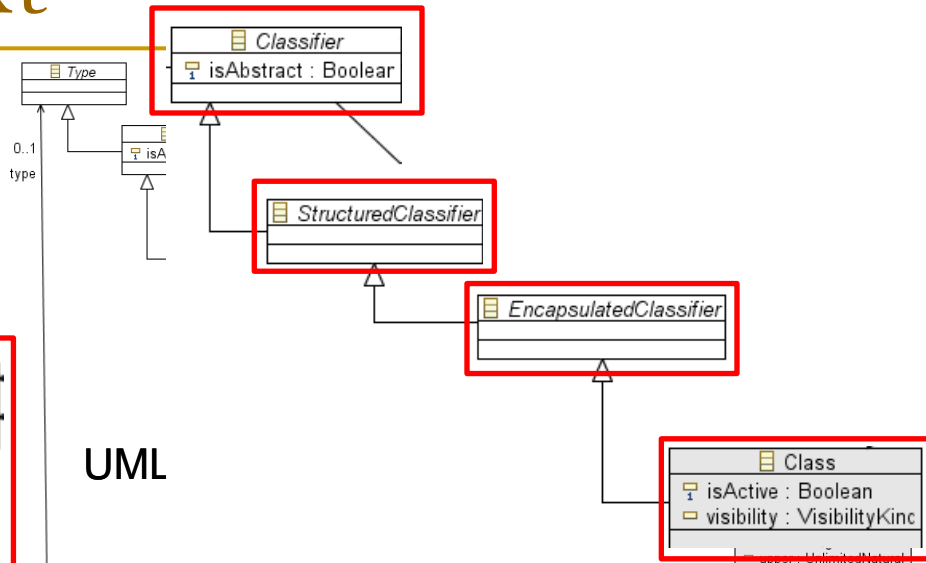
The metamodel elements may have different names

MOF Metamodel

Java Metamodel



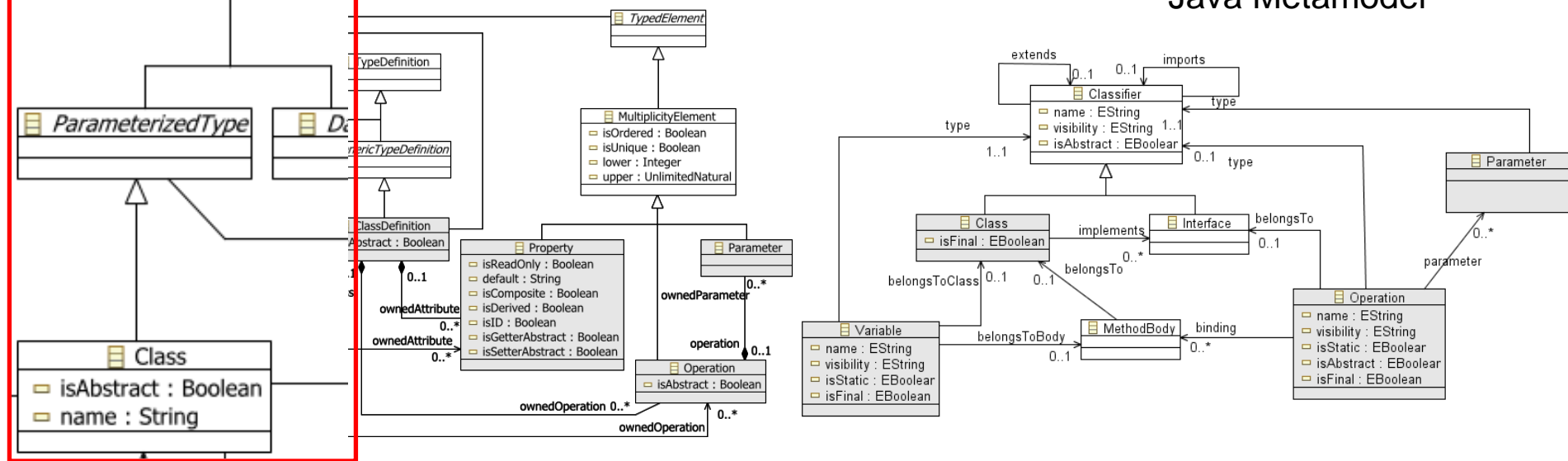
Context



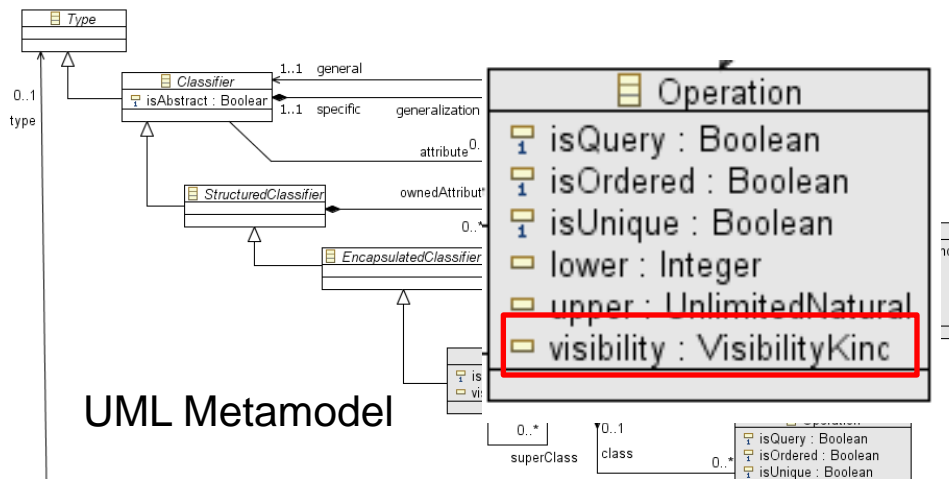
There may be additional or missing elements

UML Metamodel

Java Metamodel



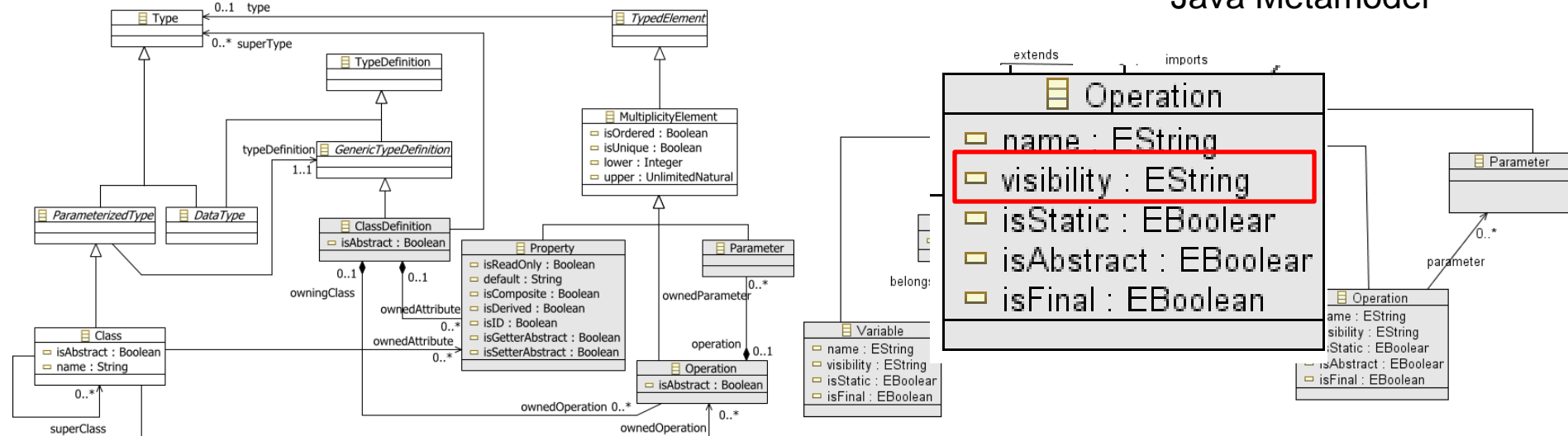
Context



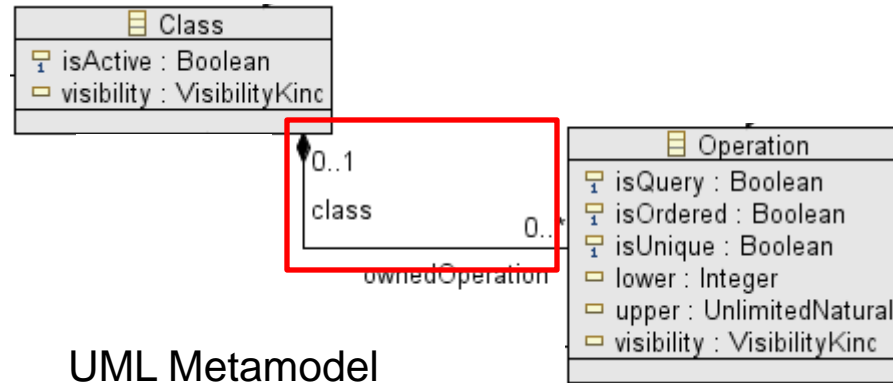
The types of elements may be different

MOF Metamodel

Java Metamodel



Context

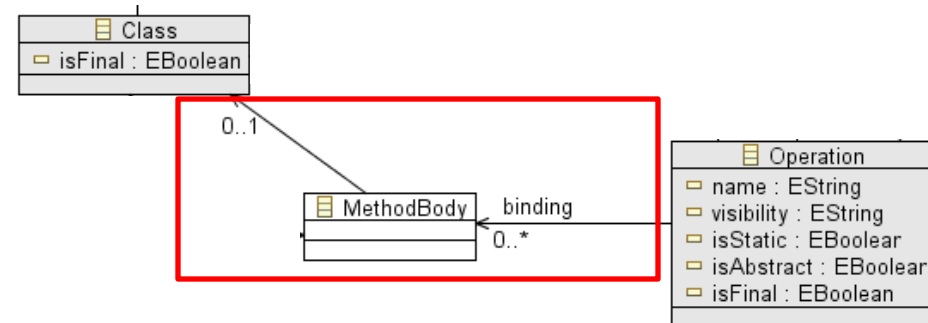
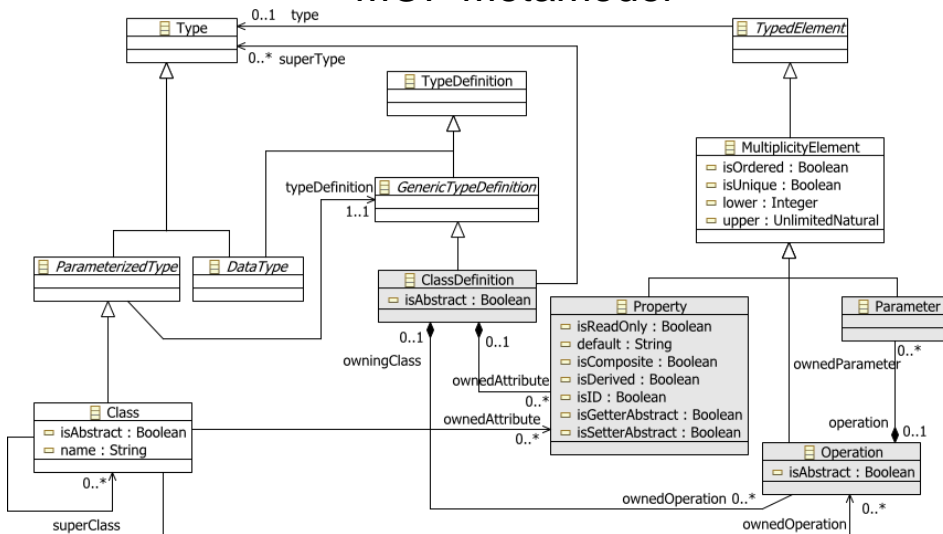


UML Metamodel

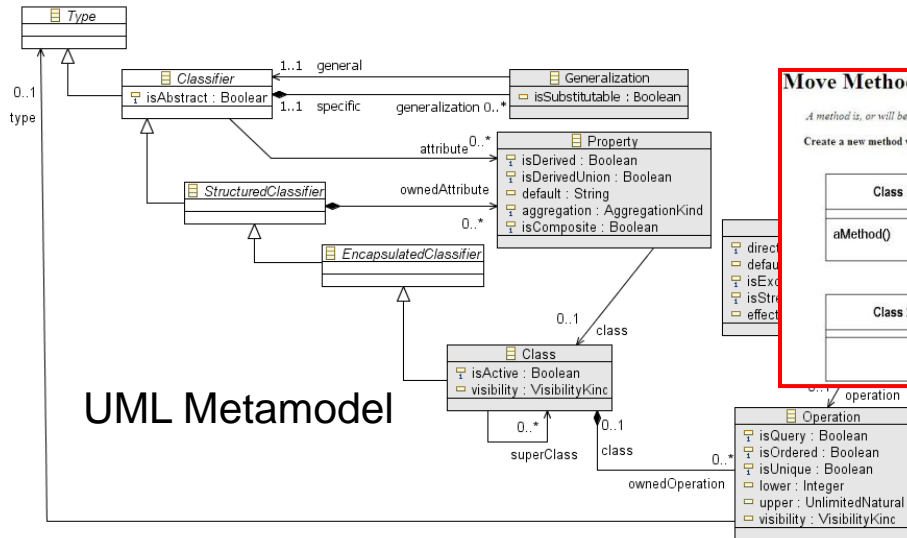
The way metamodel classes are linked together may be different

MOF Metamodel

Java Metamodel



Context

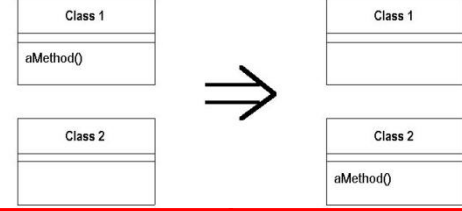


UML Metamodel

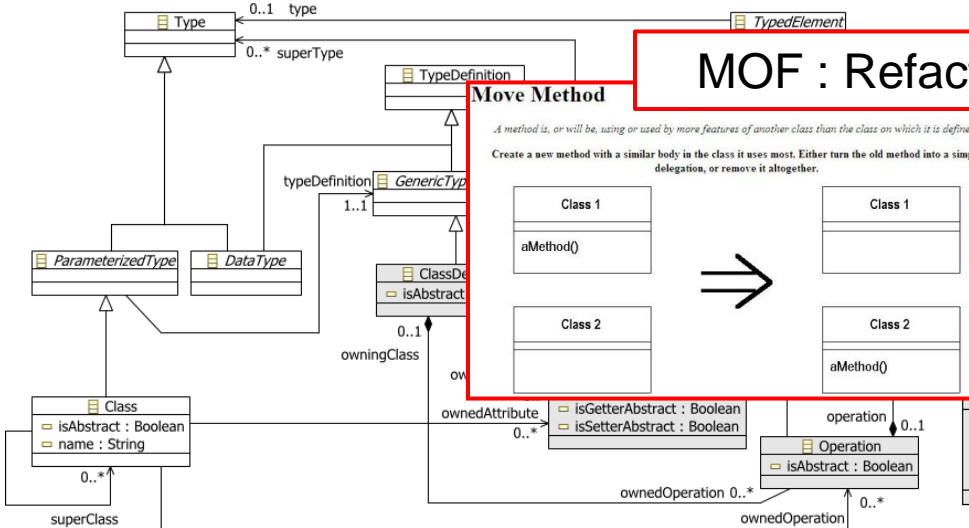
UML : Refactoring

Move Method

A method is, or will be, using or used by more features of another class than the class on which it is defined. Create a new method with a similar body in the class it uses most. Either turn the old method into a simple delegation, or remove it altogether.



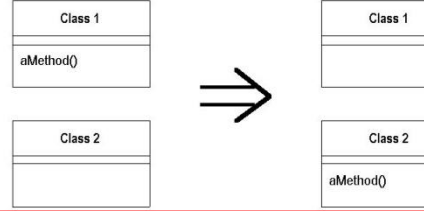
MOF Metamodel



MOF : Refactoring

Move Method

A method is, or will be, using or used by more features of another class than the class on which it is defined. Create a new method with a similar body in the class it uses most. Either turn the old method into a simple delegation, or remove it altogether.

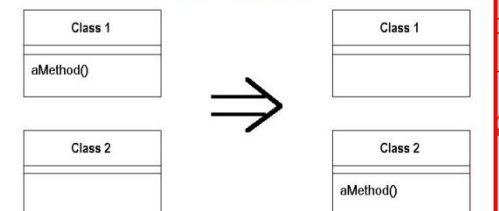


Java Metamodel

Java : Refactoring

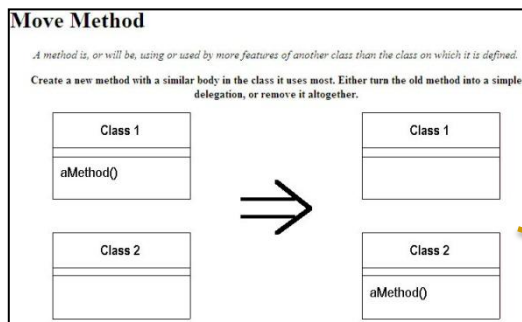
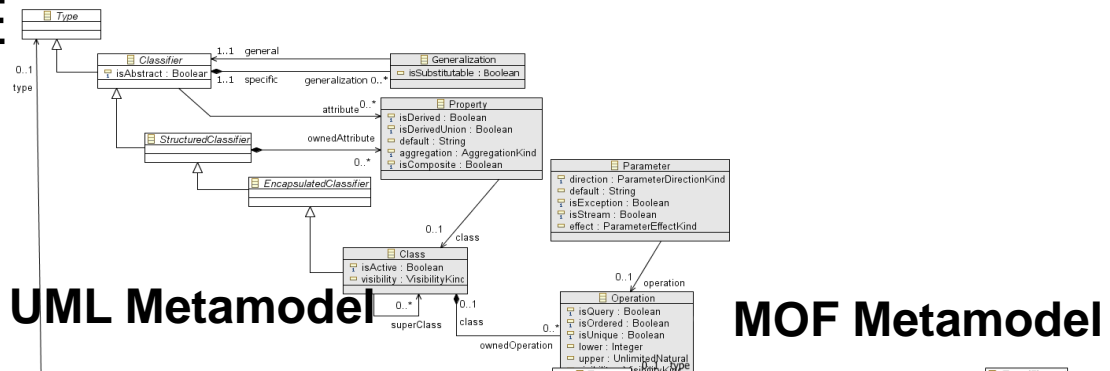
Move Method

A method is, or will be, using or used by more features of another class than the class on which it is defined. Create a new method with a similar body in the class it uses most. Either turn the old method into a simple delegation, or remove it altogether.

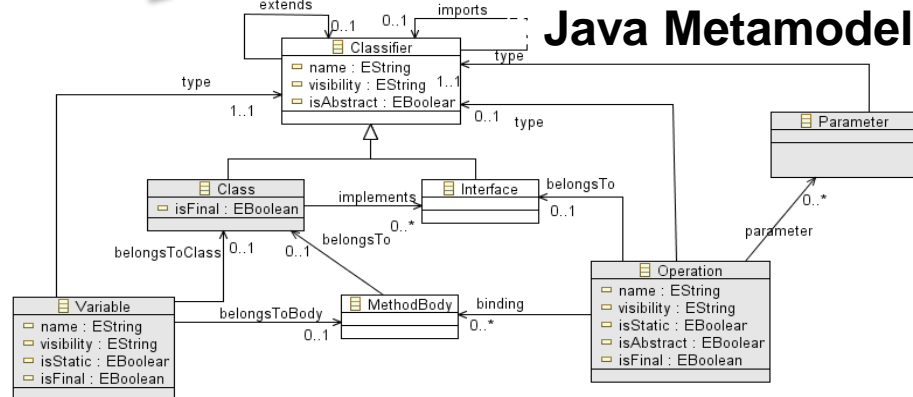


Goal

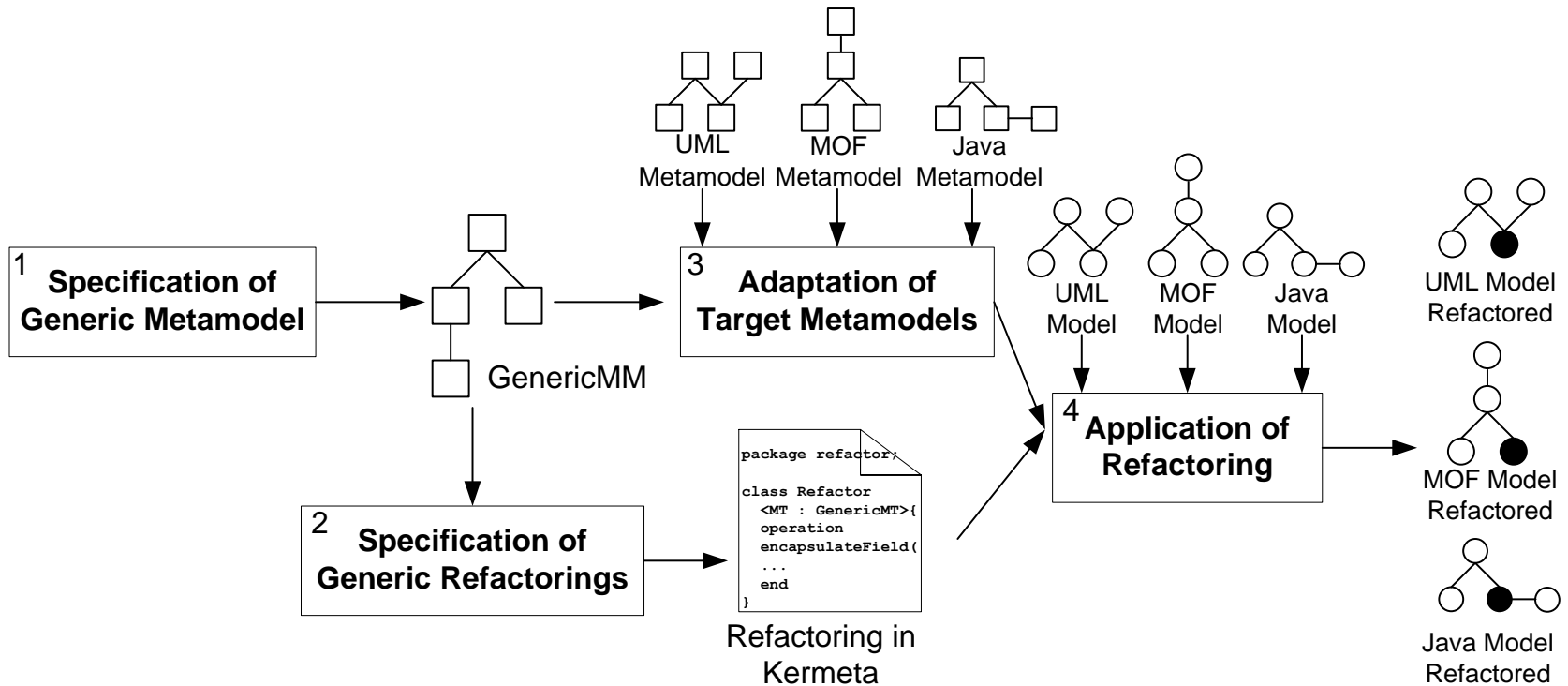
Code Reuse in MDE



Generic : Refactoring



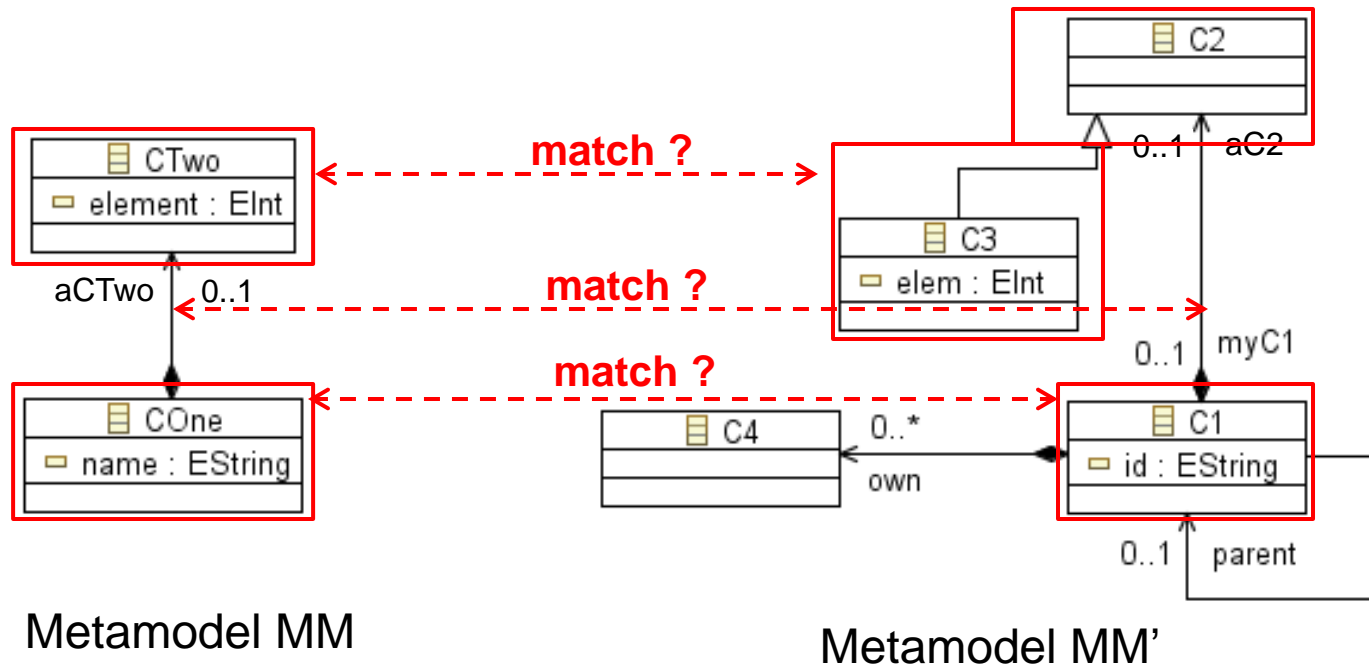
Approach



Based on **aspect weaving** and **model typing** [Steel 07, SoSyM]

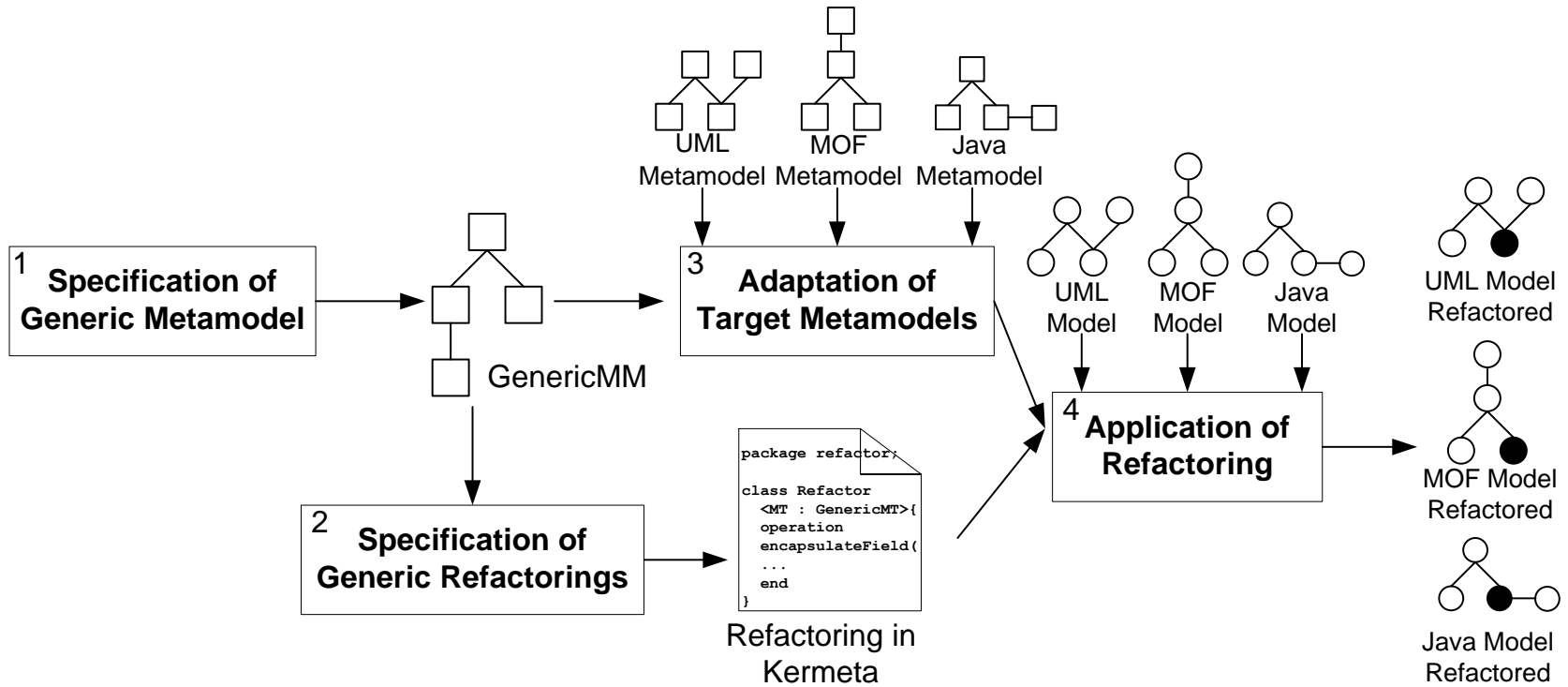
Model Typing

- Does the metamodel MM' match the metamodel MM ?

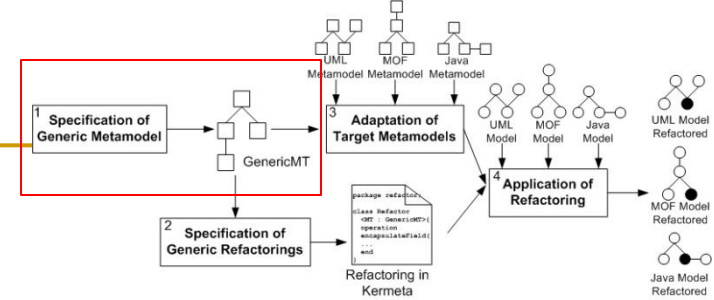


- Metamodel MM' **matches** another metamodel MM iff for each **class C** in MM, there is one and only one **corresponding class C'** in MM'
- Set of rules guarantees a **subtype relationship** btw the two MMs [Steel 07, SoSyM]
- If T works for MM, and if MM' is a model subtype of MM, then T works with MM'

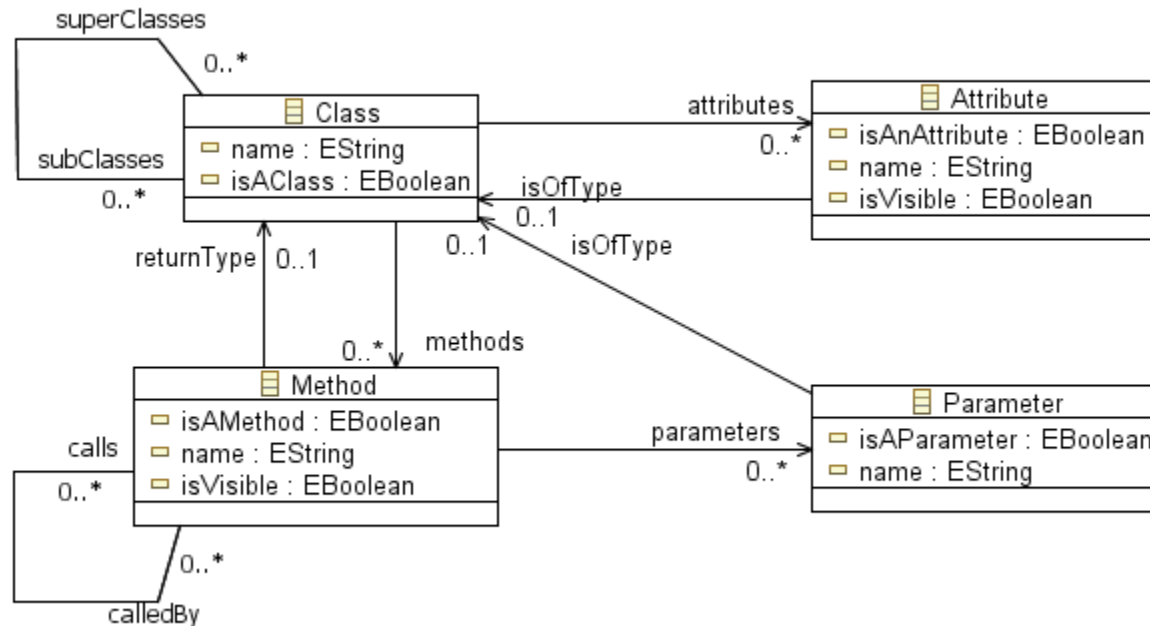
Approach



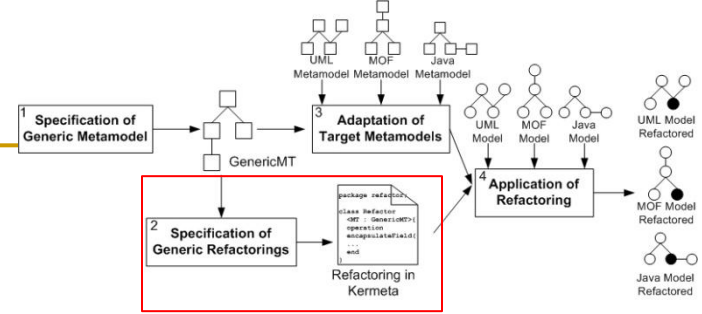
Approach



Step 1: Specification of Generic Metamodel



Approach

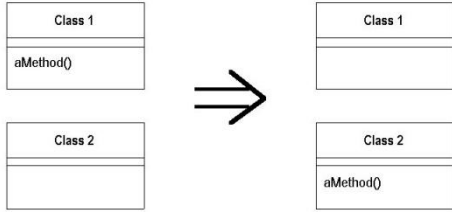


Step 2: Specification of Generic Refactorings

GenericMM : Refactoring

Move Method

A method is, or will be, using or used by more features of another class than the class on which it is defined.
Create a new method with a similar body in the class it uses most. Either turn the old method into a simple delegation, or remove it altogether.



```

package refactor;

class Refactor<MT : GenericMT>
{
    operation moveMethod(aMethod : MT::Operation,
        sourceClass : MT::Class,
        targetClass : MT::Class) : Void

    // Preconditions
    pre methodExists is
        do sourceClass.operations.exists( op |
            op.name == aMethod.name)
        end

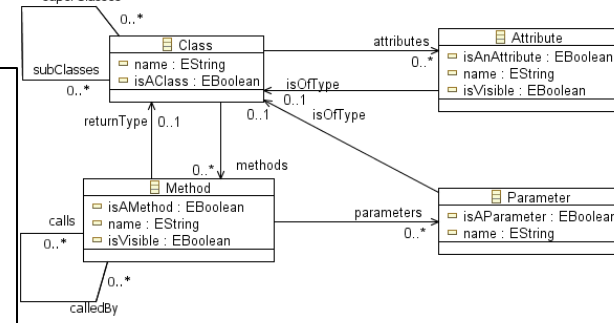
    pre notSameMethodInTarget is
        do
            not targetClass.operations.exists( op |
                op.name == aMethod.name
            )
        end

    // Operation body
    is do

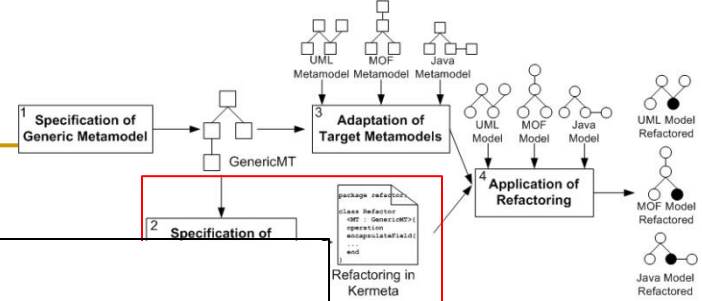
        targetClass.operations.add(aMethod)
        sourceClass.operations.remove(aMethod)

    end
}
    
```

Generic Metamodel



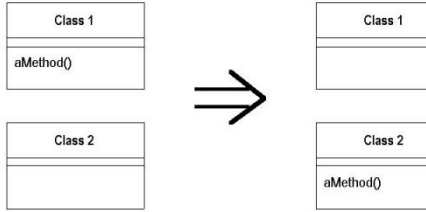
Approach



Step 2: S

GenericMM

Move Method
A method is, or will be, using or used by more features of another class than the class on which it is defined.
 Create a new method with a similar body in the class it uses most. Either turn the old method into a delegation, or remove it altogether.



```

package refactor;

class Refactor<MT : GenericMM>
{
    operation moveMethod(aMethod : MT::Operation,
                        sourceClass : MT::Class,
                        targetClass : MT::Class) : Void

    // Preconditions
    pre methodExists is
        do sourceClass.operations.exists( op |
            op.name == aMethod.name)
        end

    pre notSameMethodInTarget is
        do
            not targetClass.operations.exists( op |
                op.name == aMethod.name
            )
        end

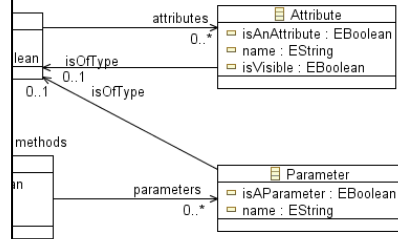
    // Operation body
    is do

        targetClass.operations.add(aMethod)
        sourceClass.operations.remove(aMethod)

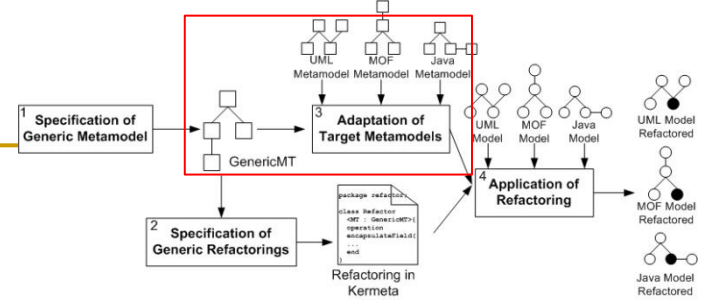
    end
}
    
```

Refactorings

Generic Metamodel

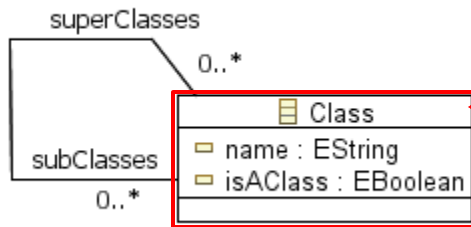


Approach



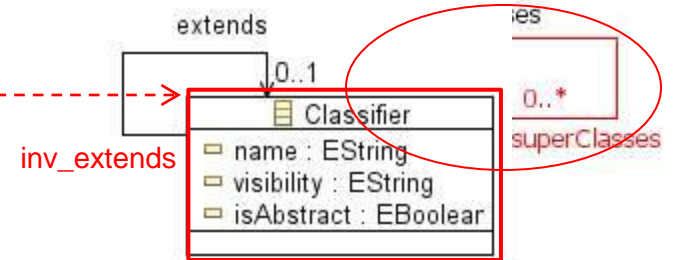
Step 3: Adaptation of Target Metamodels

Generic Metamodel

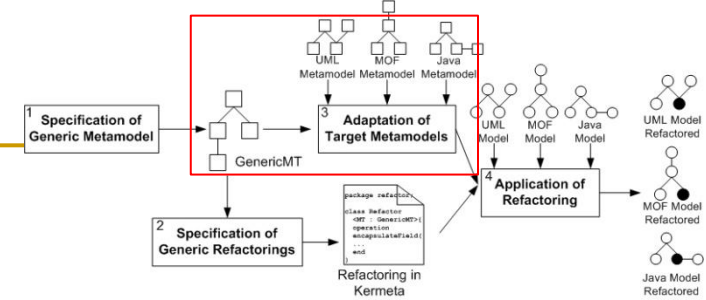


matches ? Yes!

Java Metamodel

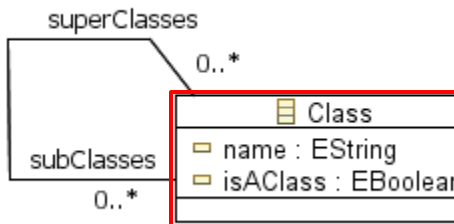


Approach



Step 3: Adaptation of Target Metamodels

Generic Metamodel



```
package java;

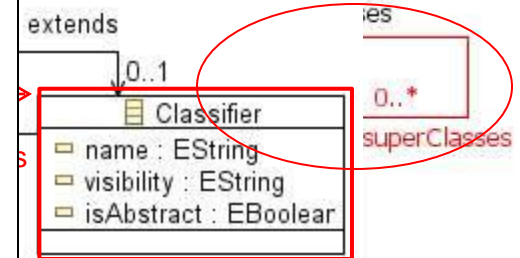
aspect class Classifier {
  reference inv_extends : Classifier[0..*]#extends
  reference extends : Classifier[0..1]#inv_extends
}

aspect class Class {

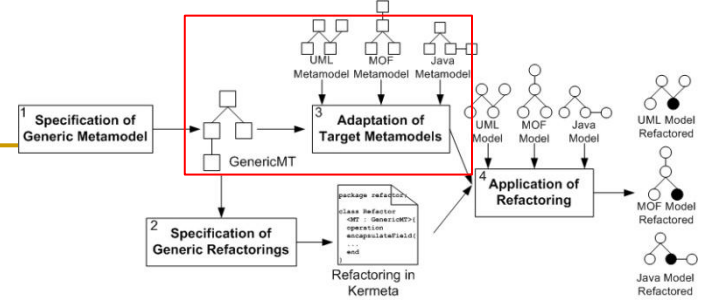
  property superClasses : Class[0..1]#subClasses
  getter is do
    var clazz : java::Class
    clazz ?= self.extends
  end

  property subClasses : Class[0..*]#superClasses
  getter is do
    result := OrderedSet<java::Class>.new
    self.inv_extends.each{subC |
      var clazz : java::Class
      clazz ?= subC
    }
  end
}
```

Java Metamodel

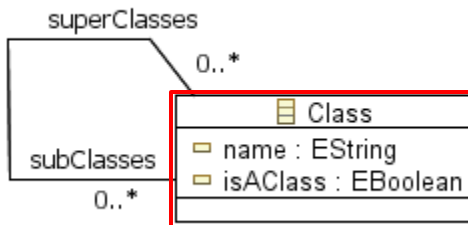


Approach

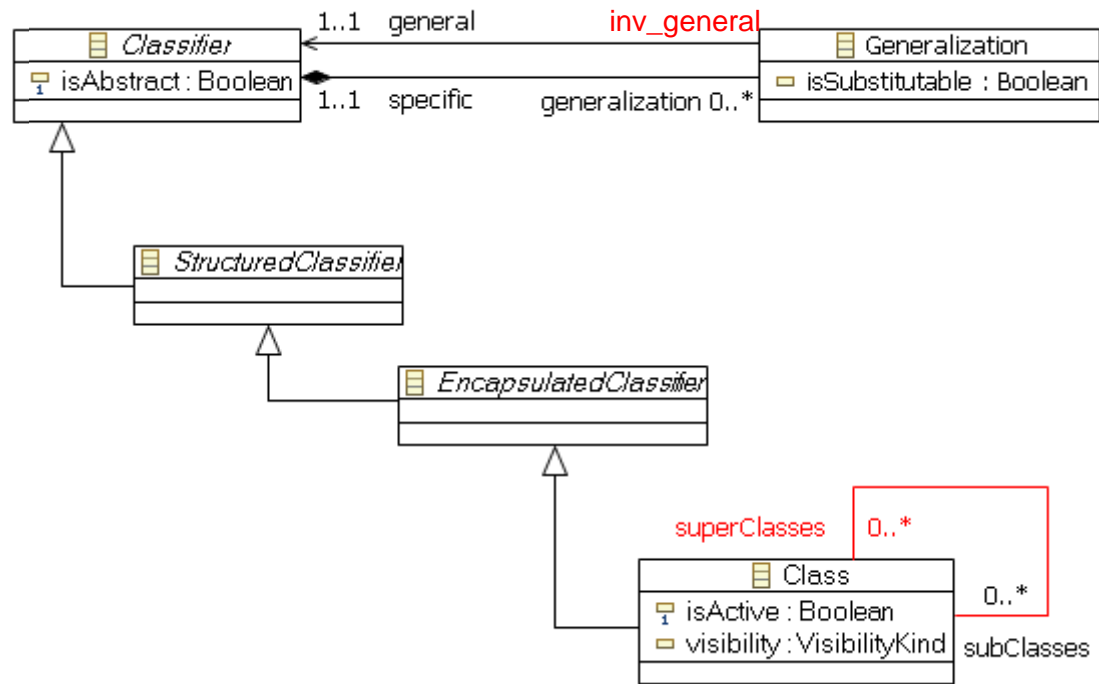


Step 3: Adaptation of Target Metamodels

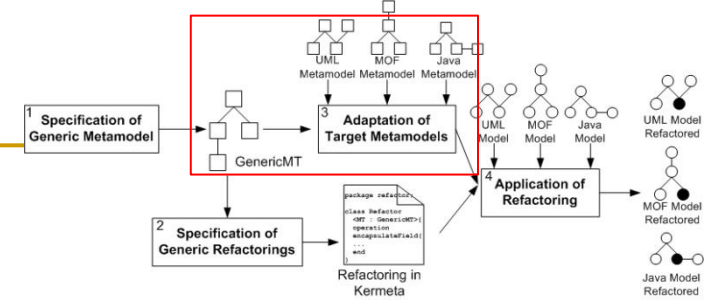
Generic Metamodel



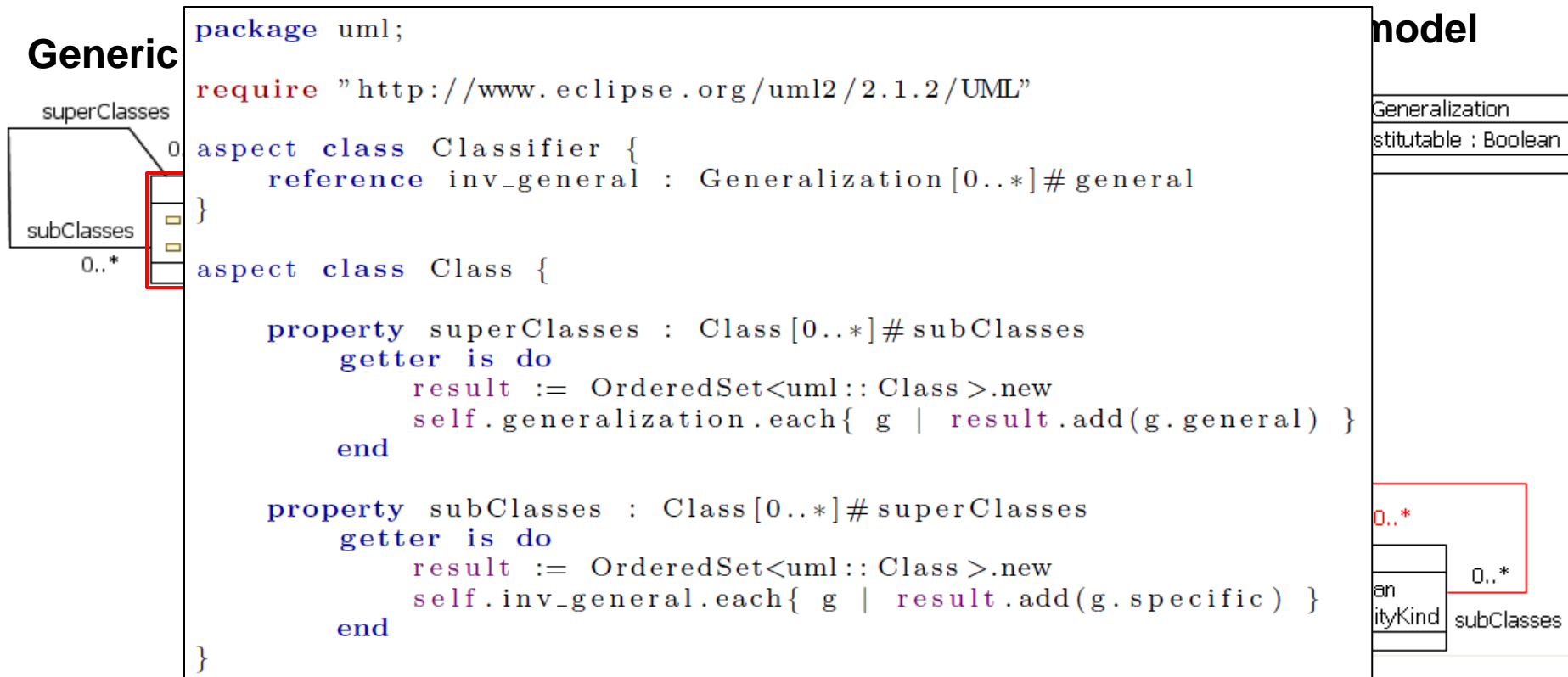
UML Metamodel



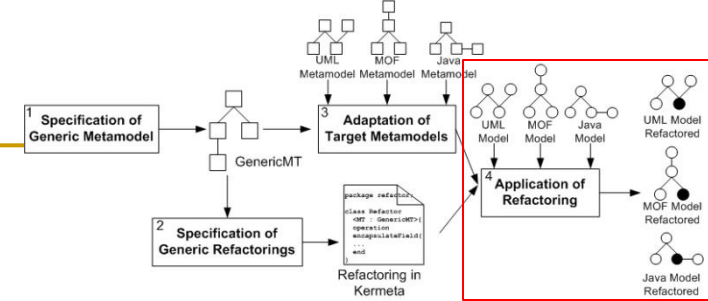
Approach



Step 3: Adaptation of Target Metamodels



Approach



Step 4: Application of Refactoring

```
package refactor;
class Refactor<MT : GenericMT> {
    operation pullUpMethod(
        source : MT::Class
```

Class Refactor <MT : GenericMM>

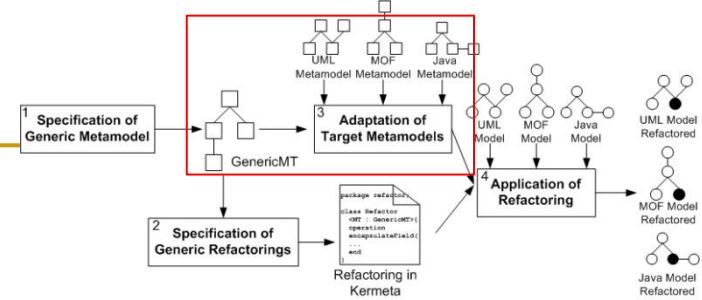
```
pre sameSignatureInOtherSubclasses is do
    target.subClasses.forAll{ sub |
        sub.methods.exists{ op |
            haveSameSignature(meth, op)
        }
    }
end
// Operation body
is do
    target.methods.add(meth)
    source.methods.remove(meth)
end
```

```
package refactor;
require "http://www.eclipse.org/uml2/2.1.2/UML"
class Main {
    operation main() : Void is do
        var rep : EMFRepository init EMFRepository.new
        var model : uml::Model
        model ?= rep.getResource("lan_application.uml").one
        var source : uml::Class init getClass("PrintServer")
        var target : uml::Class init getClass("Node")
        var meth : uml::Operation init getOperation("bill")
```

**Var refactor : Refactor::Refactor<uml::UmlMM>
refactor.moveMethod(meth, source, target)**

The Target metamodel is a subtype of the expected supertype GenericMM

Approach



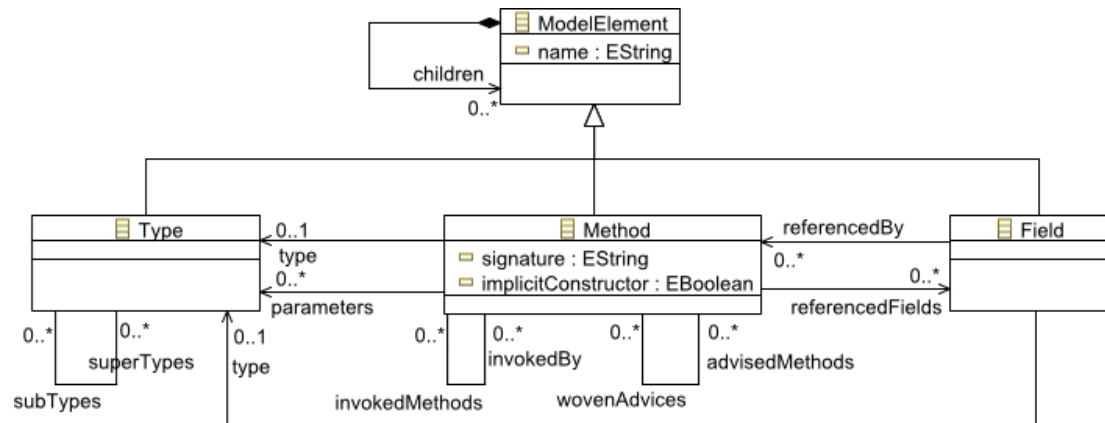
Discussions

- The adaptation is necessary because metamodels are structurally different
- The adaptation virtually modifies the structure of the target metamodel with additional elements
- Model typing guarantees the type conformance between the target metamodels and the generic metamodel

Case Study

Goal : Validate our approach on a concrete application

- 3 refactorings (Encapsulate Field, Move Method, and Pull Up Method)
- 3 different metamodels (Java, MOF, and UML)
- 4th metamodel



- ❑ **Ambiguous match** : 2 classes of the generic MM were unified in a same class
- ❑ **Limitation** : *each element in the generic MM should correspond to a distinct element in the target MM*

Conclusion

- Approach for **generic model refactorings**
 - **Model typing** and **weaving of aspects**
 - Writing adaptations can be more or less difficult
 - Once adaptation done, one can reuse all transfo
 - If new transfo added, it can be applied on all target metamodels
 - **Flexible reuse** of model transformations
 - **Generalisable** to other endogenous MTs

Conclusion

- AOM perspective : “Weaving refactoring aspects into metamodels”
- Future Work
 - Increase the repository of refactorings on other metamodels
 - Experiment on large scale applications such as Java programs
 - Experiment with other model transformations

Questions

Contact: moha@irisa.fr

<http://www.irisa.fr/triskell>

