OpenEmbeDD lectures

Kermeta 1.3

ModelType for Generic Model Engineering

Contents

- Introduction
- Generic MM / transformation
- “Model Type”
- ModelType conformance toughness
- *NonMatching* strategy
- Apply refactoring to new metamodels
- Adaptation recurrent patterns
- … what is ModelType?

Introduction

What is the goal of this presentation?

- Present the Kermeta “ModelType” feature
- Use ModelType translation to build generic Model transformations:
  - Define the transformation on a generic metamodel
  - Define the ModelType on this generic metamodel
  - Declare corresponding ModelType on targetted metamodels
  - Complete those metamodels to adapt them to the ModelType, using Kermeta aspects
  - Run the transformation on instance models of those metamodels
- Discuss some theoretic concerns about ModelType

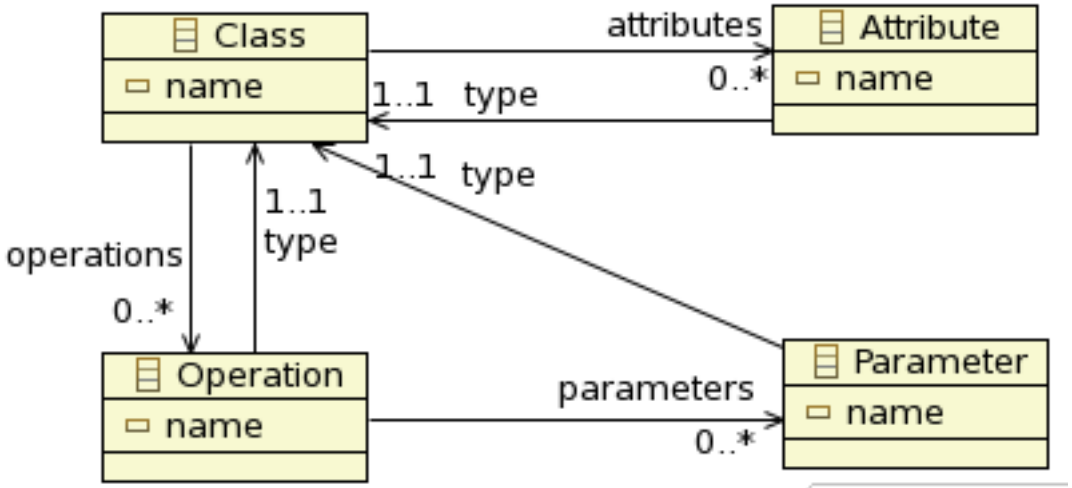
Generic transformation and its metamodel

Generic transformation

- **Our goal:**
  - Write once a refactoring using a generic metamodel
  - Easily apply it to equivalent real metamodels instances (like UML models,...)
- **It implies:**
  - Define the generic metamodel
  - Use it for typing in our refactoring
  - Use the code as-is on targeted metamodels
  - Easily adapt those metamodels to the generic one
  - Translate generic types in the corresponding types in the targeted metamodels
  - Typecheck statically and at execution time

Generic transformation

- **A expected transformation: setter refactoring**
  - A ***Class***
  - An ***Attribute*** of this class
  - The program adds a setter ***Operation*** on attribute with the ***type*** of attribute as ***Parameter***
  - The program adds a getter operation on attribute
- **First draft of a metamodel *GenericMM.ecore* about elements implied in transformation:**



Generic transformation

The refactoring code

```
package refactor;
require kermeta
require "GenericMT.kmt"

class Refactor<MT : GenericMT>
{
   operation encapsulateField(field : MT::Attribute,
                              fieldClass : MT::Class,
                              getterName : kermeta::standard::String,
                              setterName : kermeta::standard::String) : Void is do

     ///////// manage the setter /////////
     if not fieldClass.operations.exists{ op | op.name == setterName } then
       //  no setter so we must add it
       var op1 : MT::Operation init MT::Operation.new
       op1.name := setterName
       fieldClass.operations.add(op1)

       // it is a setter so we have input parameter)
       var par : MT::Parameter init MT::Parameter.new
       par.name := field.name
       par.type := field.type
       op1.parameters.add(par)
     end

     ///////// manage the getter /////////
     if not fieldClass.operations.exists{ op | op.name == getterName } then
       //  no getter so we must add it
       var op : MT::Operation init MT::Operation.new
       op.name := getterName
       fieldClass.operations.add(op)
       // it is a getter so we have a return type
       op.type := field.type
     end
   end
}
```

Model Type

Model Type

- **Jim STEEL PhD thesis**
- ***Type* = *set of values on which a set of operations can be performed successfully***
- ***Conformance* = *weakest substitutability relation that guarantees type safety***
- ***ModelType* = *a given metamodel as nominal input/output of a model processing program***
  - There could be metamodel variants which can conformed to this given metamodel
  - A program written on the ModelType can be executed on instances of ModelType conformed variants
  - Any metamodel (not a variant) which conforms to the ModelType can run the program on its instances

ModelType on generic Metamodel

“ModelType” is a simple declaration:

```
//// “GenericMT.kmt” file ////
// same root as the Ecore metamodel above
package genericmm;

require kermeta
require "GenericMM.ecore"

modeltype GenericMT
{
  Class,
  Attribute,
  Operation,
  Parameter
}
```

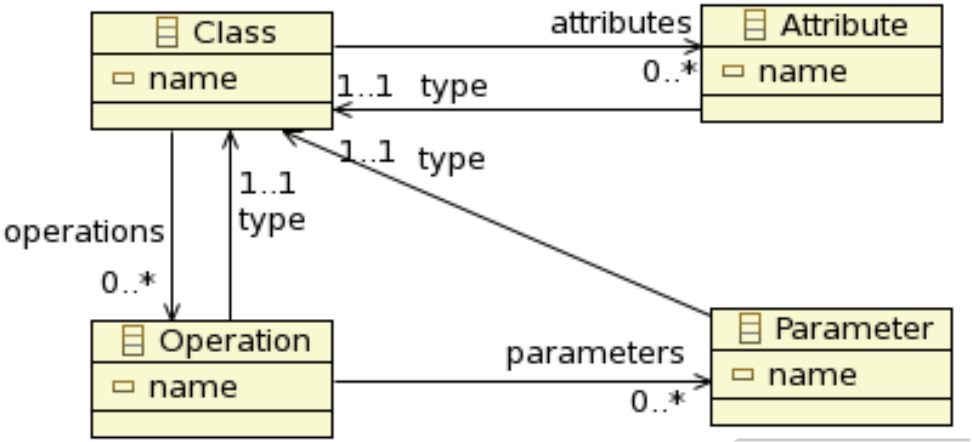
But it implies deeper concerns:

- The ModelType is based on the 4 declared elements
- The “`package genericmm;`” declaration links the ModelType with the root of the Ecore metamodel

ModelType on generic Metamodel

"ModelType" is a simple declaration:

```
//// "GenericMT.kmt" file ////
// same root as the Ecore metamodel above
package genericmm;

require kermeta
require "GenericMM.ecore"

modeltype GenericMT
{
  Class,
  Attribute,
  Operation,
  Parameter
}
```



But it implies deeper concerns:

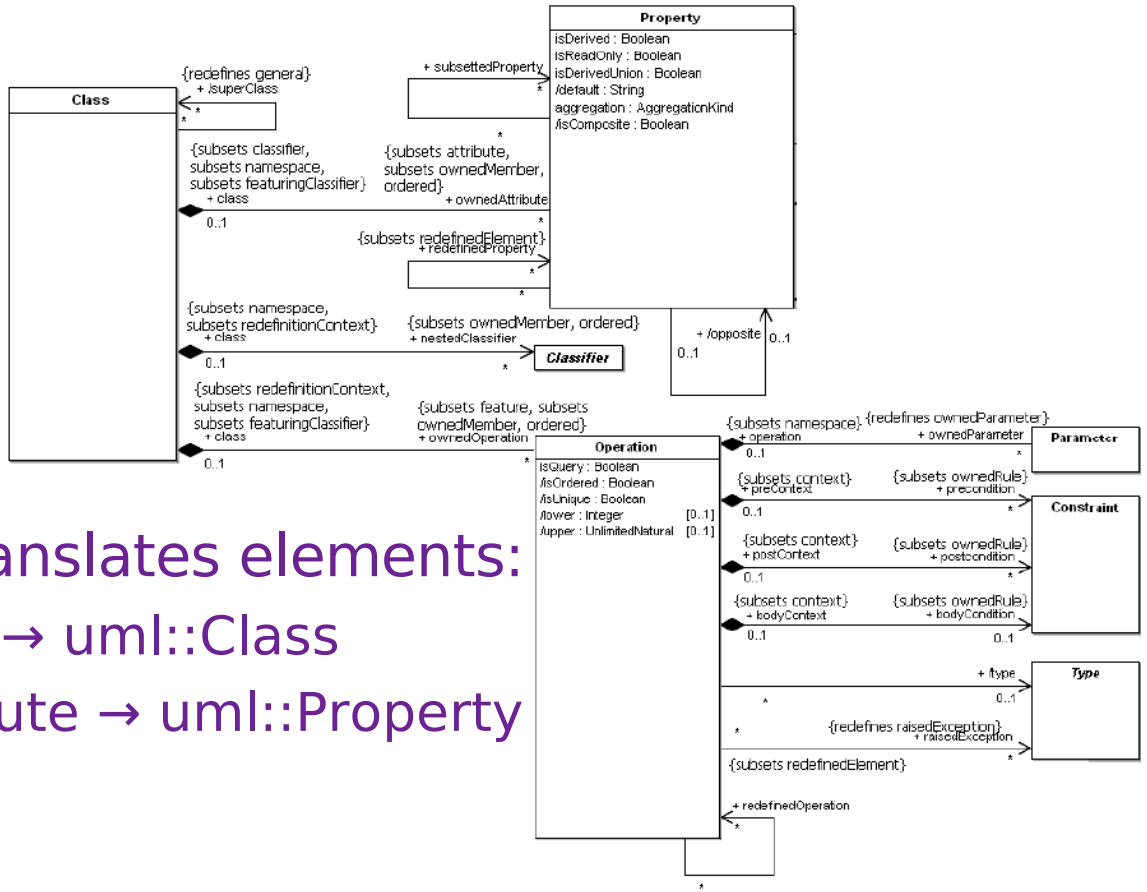
- The ModelType is based on the 4 declared elements
- The "`package genericmm;`" declaration links the ModelType with the root of the Ecore metamodel
- The ModelType includes all the features of each element as designed in the Ecore metamodel
- Details of relations in the Ecore are part of ModelType

Using ModelType with UML2

Applying ModelType on UML:

```
//// "UmlMT.kmt" file ////
// same root as the UML2 metamodel above
package uml;

require kermeta
require "http://www.eclipse.org/uml2/2.1.0/UML"

modeltype umlMT
{
  Class,
  Property,
  Operation,
  Parameter
}
```

UML equivalent of GenericMM



The typechecker translates elements:

- genericmm::Class → uml::Class
- genericmm::Attribute → uml::Property

Using ModelType with UML2

Using generic transformation method on UML

Call of the method with ModelType typing

```
@mainClass "refactor::Main"
@mainOperation "main"

package refactor;

require kermeta
require "../../metamodels/UmlMT.kmt"
require "GenericRefactor.kmt"

class Main
{
  operation main() : Void is do
    // initialization
    [....]

    var refactor : refactor::Refactor<uml::UmlMT> init refactor::Refactor<uml::UmlMT>.new

    // retrieving class and attribute to be refactored
    var node : uml::Class
    var nameField : uml::Property
    [....]

    // using the generic refactoring code
    refactor.encapsulateField(nameField, node, "getName", "setName")

    // we save the refactored UML model
    [....]
  end
}
```

ModelType conformance

The critical toughness of ModelType

ModelType conformance toughness

Targeted metamodels must comply to ModelType enough to typecheck

- Be similar is not sufficient, as ModelType is considered like any other Type in compiling domain
- The ModelType theory has defined rules of compliance between a top metamodel and variants
- The Kermeta typechecker implements the corresponding matching algorithm
- There is cycles between elements of a metamodel so the match of others elements may depend on an element with circularity
- Two similar elements of the targeted metamodel may compete for one element of generic metamodel, forbidding global match

ModelType conformance toughness

Matching rules (on Ecore properties)

req: generic metamodel element (required properties)

prov: targeted metamodel element (as provided)

- On **multiplicity**
  - req.upper = 1 *implies* prov.upper = 1
  - req.upper >= prov.upper
  - req.lower <= prov.lower
  - req.isOrdered *implies* prov.isOrdered
  - req.isUnique *implies* prov.isUnique
- On **EClass**
  - (*not* req.isAbstract) *implies* (*not* prov.isAbstract)
  - all req attributes *are matched* by prov attributes
  - all req operations *are matched* by prov operations

ModelType conformance toughness

- On **EProperty**
  - prov.name = req.name *(annotations are planned to weak it)*
  - prov multiplicity matches with req multiplicity
  - req.isReadOnly *implies* prov.isReadOnly
  - req.isComposite *implies* prov.isComposite
  - (req.opposite->isOclUndefined) *implies* (prov.opposite->isOclUndefined)
  - prov.opposite.name = req.opposite.name
- On **EOperation**
  - prov.name = req.name *(annotations are planned to weak it)*
  - prov multiplicity matches with req multiplicity
  - prov.ownedParameter.size = req.ownedParameter.size
  - all req parameters *are matched* by prov parameters

ModelType conformance toughness

As seen in previous slides, many features are verified by model typechecker

It is an AND statement

If one feature does not match, the ModelType does not match at all

The less restrictive the generic metamodel is, the easier targeted metamodels conform to it

- Do not use strict requirements which are not needed
  - Prefer Set to OrderedSet, [0..1] to [1..1],...
- Do not add elements which program does not use
  - There is no “Package” in our example

Ecore may implies restrictive default values

- Those properties should be relaxed as possible

ModelType conformance toughness

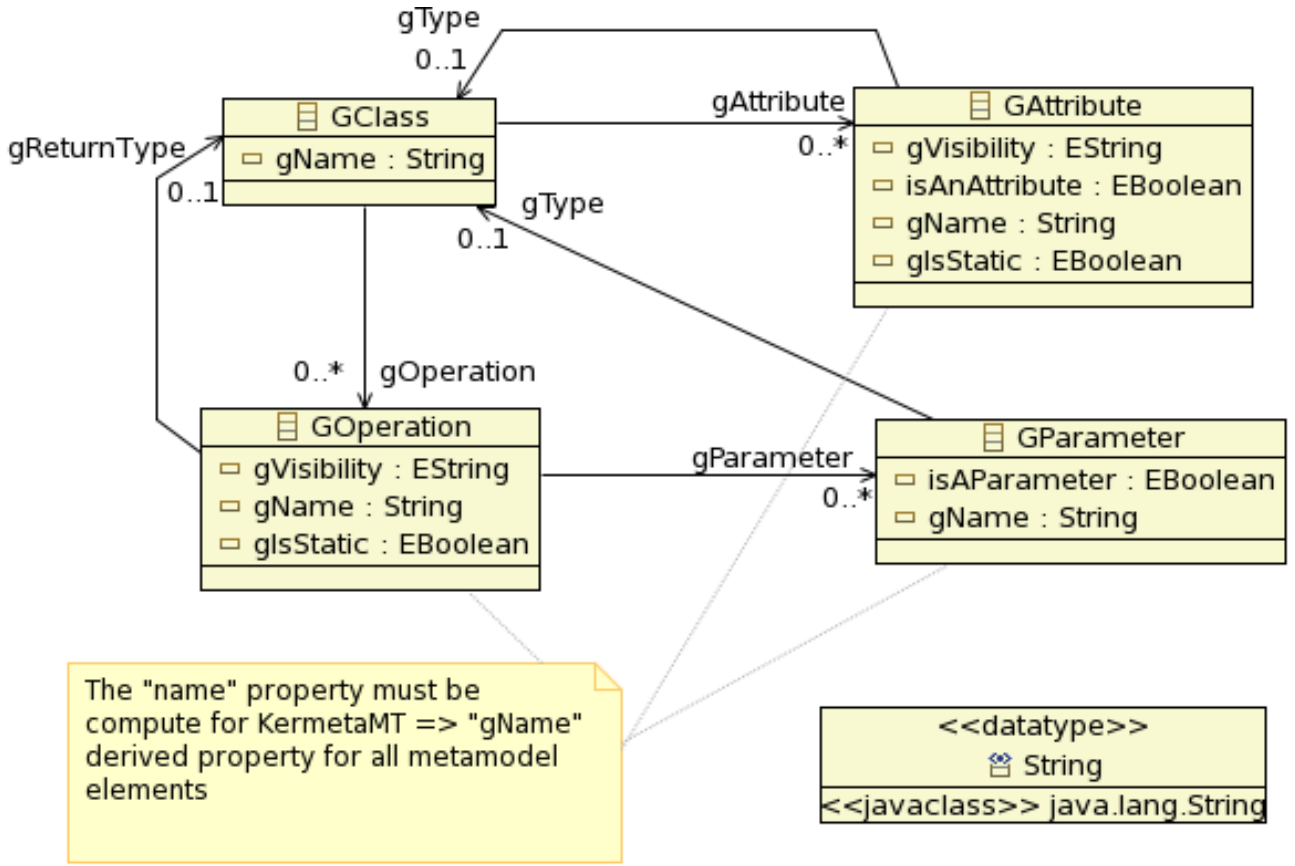
- **Main problem comes from name of attributes and operations of ModelType elements**
  - Generic metamodel elements' features must have same name as the targeted metamodel, which becomes impossible with multiple different targets
- **Sometimes the targeted metamodel semantic may also differ from the generic one**
  - It implies to derived the correct semantic in the targeted metamodel where it lacks
- **One possible strategy is to *use non matching names* in the generic metamodel**
  - Then we adapt the targeted metamodels with derived properties to conform it to the generic one
  - We generalize derived properties implementing generic

NonMatching strategy

NonMatch conformance strategy

The generic metamodel

obvious names are prefixed by a “g” (as generic) to avoid matching in original targeted metamodels



NonMatch conformance strategy

generic refactoring code

```
package refactor;
require kermeta
require "GenericMT.kmt"

class Refactor<MT : GenericMT>
{
  operation encapsulateField(field : MT::GAttribute,
                             fieldClass : MT::GClass,
                             getterName : kermeta::standard::String,
                             setterName : kermeta::standard::String) : Void is do

    ///////// manage the setter /////////
    if not fieldClass.gOperation.exists{ op | op.gName == setterName } then
      //  no setter so we must add it
      var op1 : MT::GOperation init MT::GOperation.new
      op1.gName := setterName
      fieldClass.gOperation.add(op1)

      // it is a setter so we have input parameter)
      var par : MT::GParameter init MT::GParameter.new
      par.gName := field.gName
      par.gType := field.gType
      op1.gParameter.add(par)
    end

    ///////// manage the getter /////////
    if not fieldClass.gOperation.exists{ op | op.gName == getterName } then
      //  no getter so we must add it
      var op : MT::GOperation init MT::GOperation.new
      op.gName := getterName
      fieldClass.gOperation.add(op)
      // it is a getter so we have a return type
      op.gType := field.gType
    end
  end
}
```

NonMatch conformance strategy

As we derive all generic features, we must include management of [0..*] multiplicities

We extend Kermeta collections to derived the references of generic metamodel with multiplicity > 1

```
// "UmlHelper.kmt" file

package kermeta;


require kermeta
require "http://www.eclipse.org/uml2/2.1.0/UML"

package standard {

  /** dedicated class for derived property on 'uml::Class' 'ownedOperation' attribute,
    because of its [0..*] multiplicity */
  aspect class ClassOperationsOSet<O : uml::Operation>
                          inherits kermeta::standard::OrderedSet<uml::Operation> {

    reference owner : uml::Class

    method add(element : uml::Operation) is do
    owner.ownedOperation.add(element)
    // we must maintain equivalence between real collection and the wrapping one
    super(element)
  end
}
```

Rem: we plan to make collections observable

NonMatch conformance strategy

We then add the generic metamodel elements through derived properties in UML

```
// "UmlPlus.kmt" file
package uml;

require "UmlHelper.kmt"

aspect class Class
{
  property gOperation : Operation[0..*]
    getter is do
      var coll : kermeta::standard::ClassOperationsOSet<Operation>
             init kermeta::standard::ClassOperationsOSet<Operation>.new
      coll.owner := self
      // we must duplicate data in the wrapping collection
      coll.addAll(self.ownedOperation)
      // we pass the wrapper as derived property value
      result := coll
    end

    property gAttribute : Property[0..*]
      [.. idem ..]
    end

  property gName : kermeta::standard::String
    getter is do
      result := self.name
    end

  property isAClass : kermeta::standard::Boolean
}
   [.. other properties ..]
```

managing multiplicity > 1

managing multiplicity = 1

managing similarity

NonMatch conformance strategy

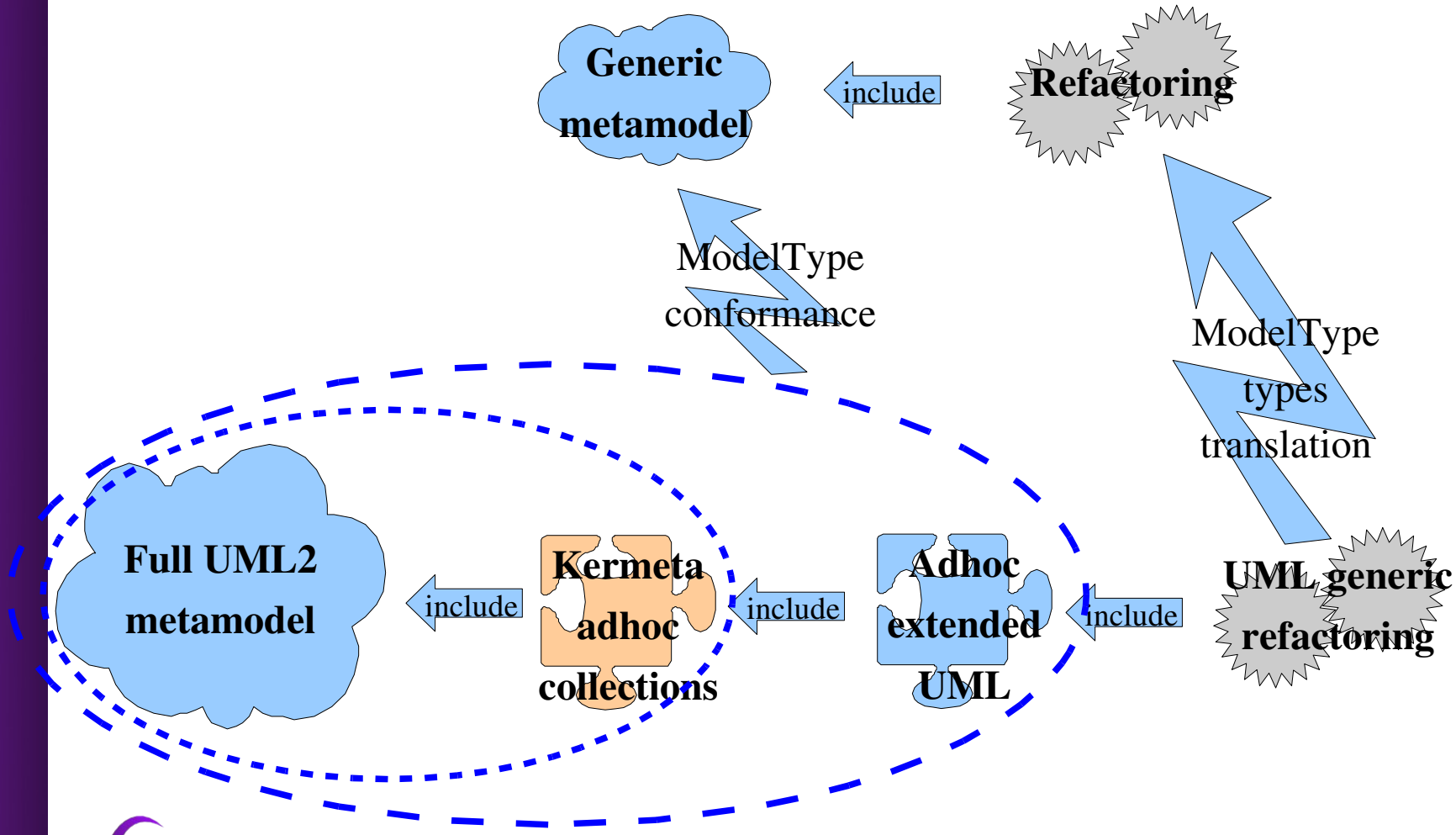
Final steps: ModelType + call of refactoring

```
// "UmlMT.kmt" file
package uml;


require kermeta
require "UmlPlus.kmt"

modeltype UmlMT
{
Class,
Property,
Operation,
Parameter
}
```

```
// "UmlGenericRefactoring.kmt" file
@mainClass "refactor::Main"
@mainOperation "main"

package refactor;

require kermeta
require "../../metamodels/UmlMT.kmt"
require "GenericRefactor.kmt"

class Main
{
  operation main() : Void is do
    // initialization
    [.. loading model ..]

    var node : uml::Class
    var nameField : uml::Property
    [.. retrieving elements ..]

    refactor.encapsulateField(nameField, node, "getName", "setName", false)

    // we save the refactored UML model
    [.. saving result ..]
  end
}
```

NonMatch conformance strategy

General scheme of the system

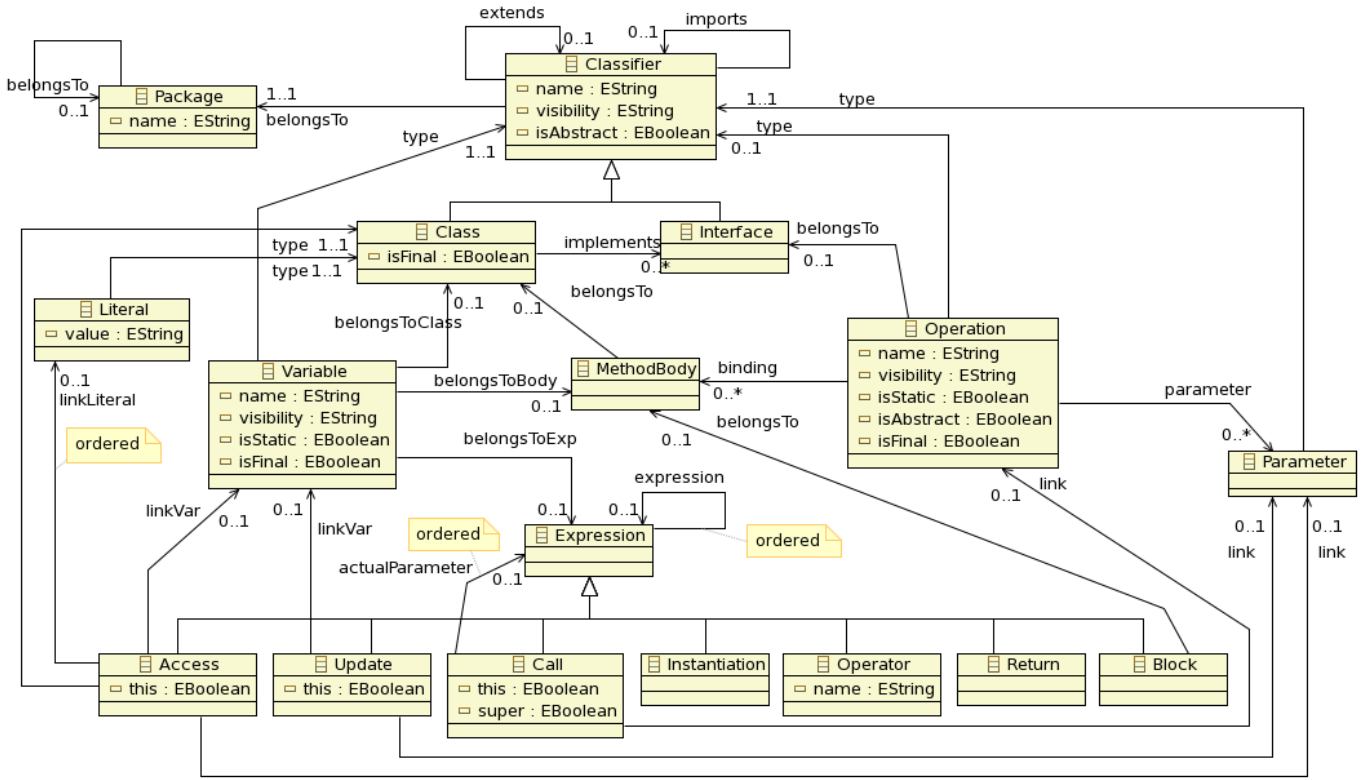


Apply to new metamodels

Apply to new metamodels

We want to refactor non UML models

- Example: java program models
- A given JavaProgram metamodel
- Different semantic
  - classes do not know operations
  - ...



Apply to new metamodels

We need

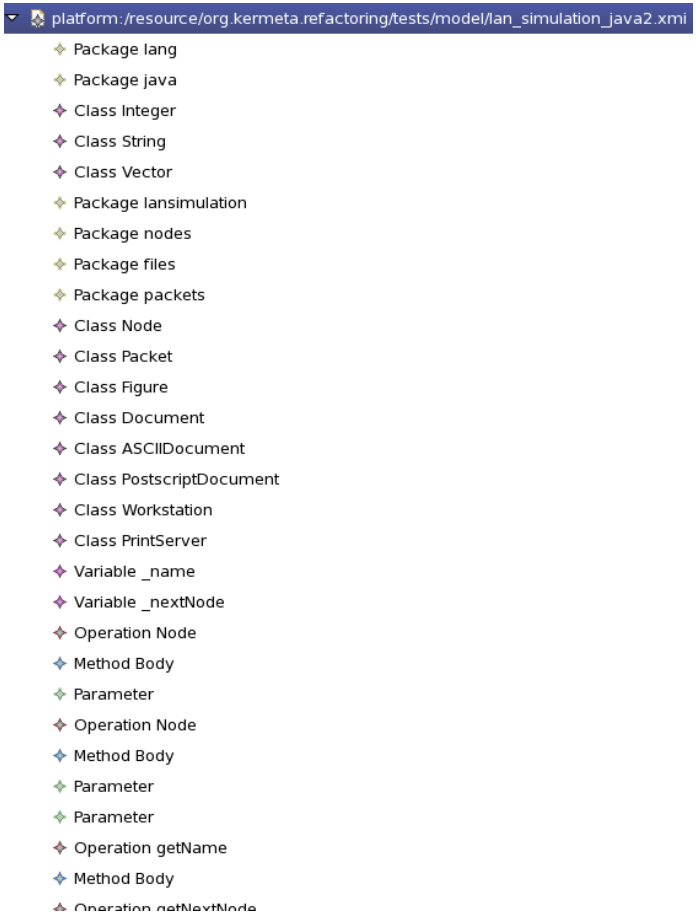
- Adhoc collections => JavaProgramHelper.kmt
- Derived properties => JavaProgramPlus.kmt
- ModelType => JavaProgramMT.kmt
- Launcher => JavaProgramGenericRefactoring.kmt

Main toughness: add lacking semantic

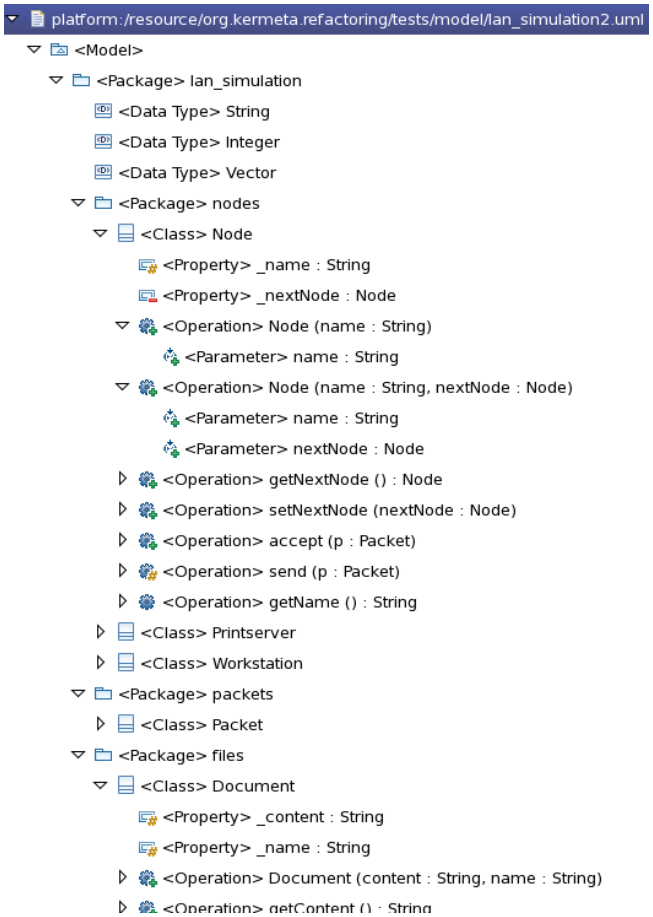
- Access to operations from class
  - => add opposites to metamodel at runtime: as it is not working currently for model loading, we replace them by adhoc computing in derived properties
- JavaProgram metamodel implies flat models (all model elements are stored at the resource root)
  - => manipulate the resource when adding elements

Apply to new metamodels

A flat model example



Similar UML model



Apply to new metamodels

Managing flat model structure

Adding a new element in model (adhoc collection)

```
// "JavaProgramHelper.kmt" file
package kermeta;

package standard {

  /** dedicated class for derived property on 'uml::Class' 'ownedOperation'
      attribute, because of its [0..*] multiplicity */
  aspect class ClassOperationsOSet<O : javaprogram::Operation> inherits
                kermeta::standard::OrderedSet<javaprogram::Operation> {

    reference owner : javaprogram::Class

    operation initialize(ownerColl : javaprogram::Operation[0..*]) is do
      self.addAll(ownerColl)
    end

    method add(element : javaprogram::Operation) is do
      // we must create a body if the operation have no body corresponding to the class
      var opBody : javaprogram::MethodBody init element.binding.detect{ body |
        body.belongsTo == owner or body.belongsTo.isVoid
      }
      if opBody == void then
        opBody := javaprogram::MethodBody.new
        element.binding.add(opBody)
        owner.containingResource.add(opBody)
        // we expect the operation is a new one and needs to be inserted in the resource
        owner.containingResource.add(element)
      end
      opBody.belongsTo := owner
      // we must maintain equivalence between real collection and the wrapping one
      super(element)
    end
  }
}
```

Apply to new metamodels

Managing flat model structure

Adding a new element in model (derived property)

```
// "JavaProgramPlus.kmt" file
package javaprogram;

require kermeta
require "JavaProgramHelper.kmt"

aspect class Class
{
  property gOperation : Operation[0..*]
    getter is do
      var coll : kermeta::standard::ClassOperationsOSet<Operation>
            init kermeta::standard::ClassOperationsOSet<Operation>.new
      coll.owner := self
      // we must duplicate data in the wrapping collection
      self.containingResource.each{ o |
        var op : Operation
        op ?= o
        if op != void then
          op.binding.each{ body |
            if body.belongsTo == self then
              coll.add(op)
            end
          }
        end
      }
      // we pass the wrapper as derived property value
      result := coll
    end
  [.. other derived properties ..]
}
```

Apply to new metamodels

Define ModelType and use it

```
// "JavaProgramMT.kmt" file
package javaprogram;

require kermeta
require "JavaProgramPlus.kmt"

modeltype JavaProgramMT
{
Class,
Variable,
Operation,
Parameter
}
```

```
// "JavaGenericRefactor.kmt" file
@mainClass "refactor::Main"
@mainOperation "main"

package refactor;

require "../../metamodels/JavaProgramMT.kmt"
require "GenericRefactor.kmt"

class Main
{
  operation main() : Void is do
    // initialization
    [.. loading model ..]

    var refactor : refactor::Refactor<javaprogram::JavaProgramMT>
            init refactor::Refactor<javaprogram::JavaProgramMT>.new

    var node : javaprogram::Class
    var nameField : javaprogram::Variable
    [.. retrieving elements ..]

    // MODEL TYPE use
    refactor.encapsulateField(nameField, node, "getName", "setName", false)

    [.. saving result..]
  end
}
```