ModelType
generic refactoring usecase

Vincent MAHÉ

IRISA Lab / INRIA Rennes, France
Triskell Team

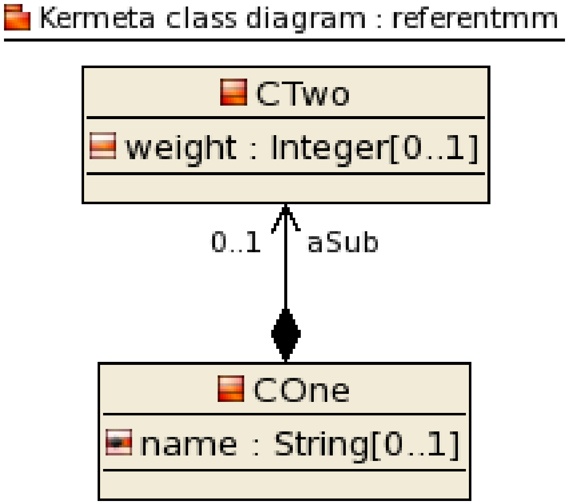
vmahe@irisa.fr

ModelType

- 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

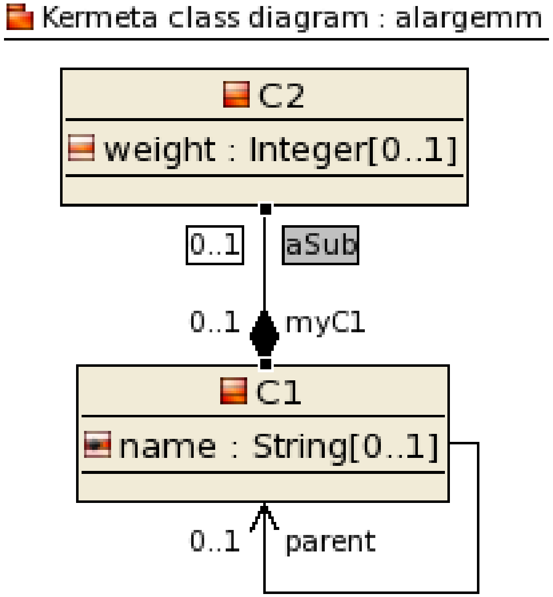
ModelType

We define a referent model and its model type



```
//// ReferentMT.kmt file ////
// same root as the .km file
package referentmm;

require kermeta
require "ReferentMM.km"

modeltype ReferentMT
{
   COne,
   CTwo
}
```

We want to find “it” in a larger model



```
//// ALargeMT.kmt file ////
package alargemm;

require kermeta
require "ALargeMM.km"

// we aim for it to correspond
// referent model type
modeltype ALargeMT
{
   C1,
   C2
}
```

ModelType

We write a program on ReferentMT

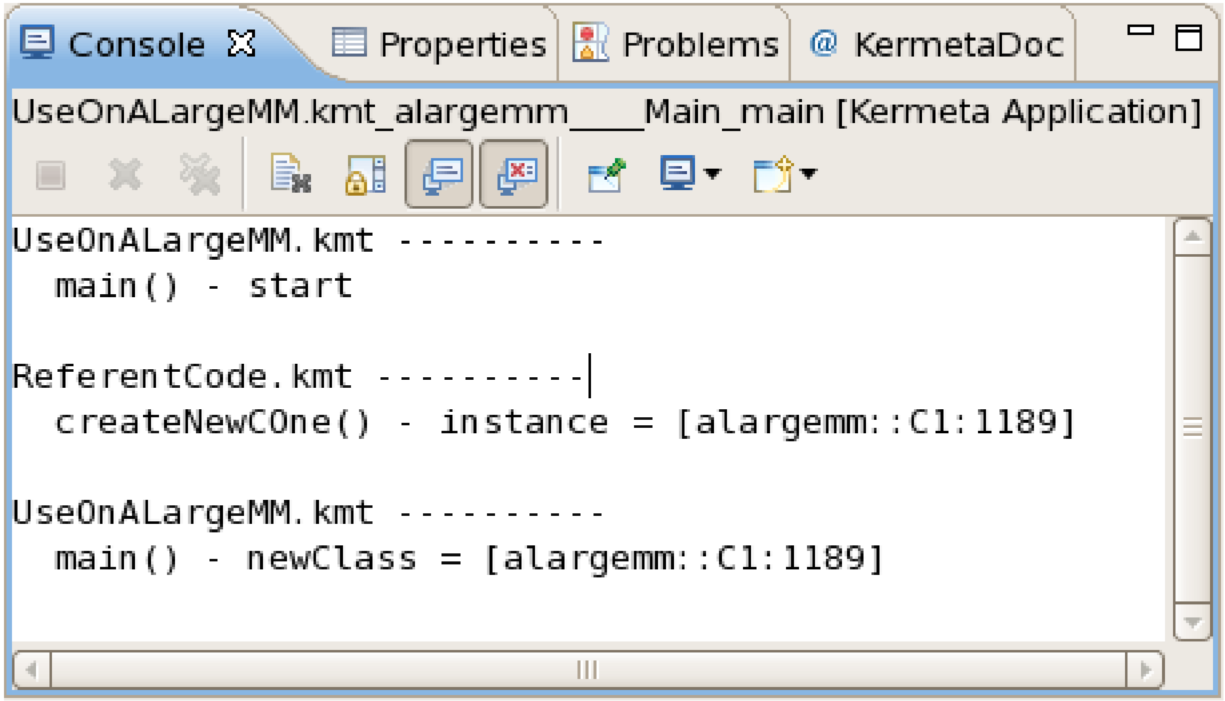
```
//// ReferentCode.kmt ////
package referentmm;

require kermeta
require "ReferentMT.kmt"

using kermeta::standard

// we define a generic class typed with ReferentMT
class Code<MT : ReferentMT>
{
 operation createNewCOne(name : String) : MT::COne is do
  // We are manipulating ReferentMM elements
  result := MT::COne.new
  result.name := name

  stdio.writeln("ReferentCode.kmt ----------")
  stdio.writeln("  createNewCOne() - instance = "
       + result.toString + "\n")
 end
}
```

We use it on ALargeMT

```
//// UseOnALargeMM.kmt ////
@mainClass "alargemm::Main"
@mainOperation "main"

package alargemm;

require kermeta
require "ALargeMT.kmt"
require "ReferentCode.kmt"

class Main
{
 operation main() : Void is do
  stdio.writeln("UseOnALargeMM.kmt ----------\n  main() - start\n")

  // we use referent code through targeted modeltype
  var code : referentmm::Code<alargemm::ALargeMT>
        init referentmm::Code<alargemm::ALargeMT>.new

  // we try to create a new C1 class using the referent code
  var newClass : alargemm::C1 init code.createNewCOne("MyC1Class")

  // we obtain an effective C1 class
  stdio.writeln("UseOnALargeMM.kmt ----------")
  stdio.writeln("  main() - newClass = " + newClass.toString)
 end
}
```

ModelType

Even the referent code manipulate the targeted metamodel elements



UseOnALargeMM.kmt_alargemm___Main_main [Kermeta Application]

```
UseOnALargeMM.kmt ----------
  main() - start

ReferentCode.kmt ----------
  createNewCOne() - instance = [alargemm::C1:1189]

UseOnALargeMM.kmt ----------
  main() - newClass = [alargemm::C1:1189]
```

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

Generic Refactoring Usecase

Our goal

- Define a library of generic refactorings
- Apply it on many similar metamodels
    - UML class diagrams
    - Kermeta program models
    - Java program models

A huge difficulty

- Find a modeltype that match all of them

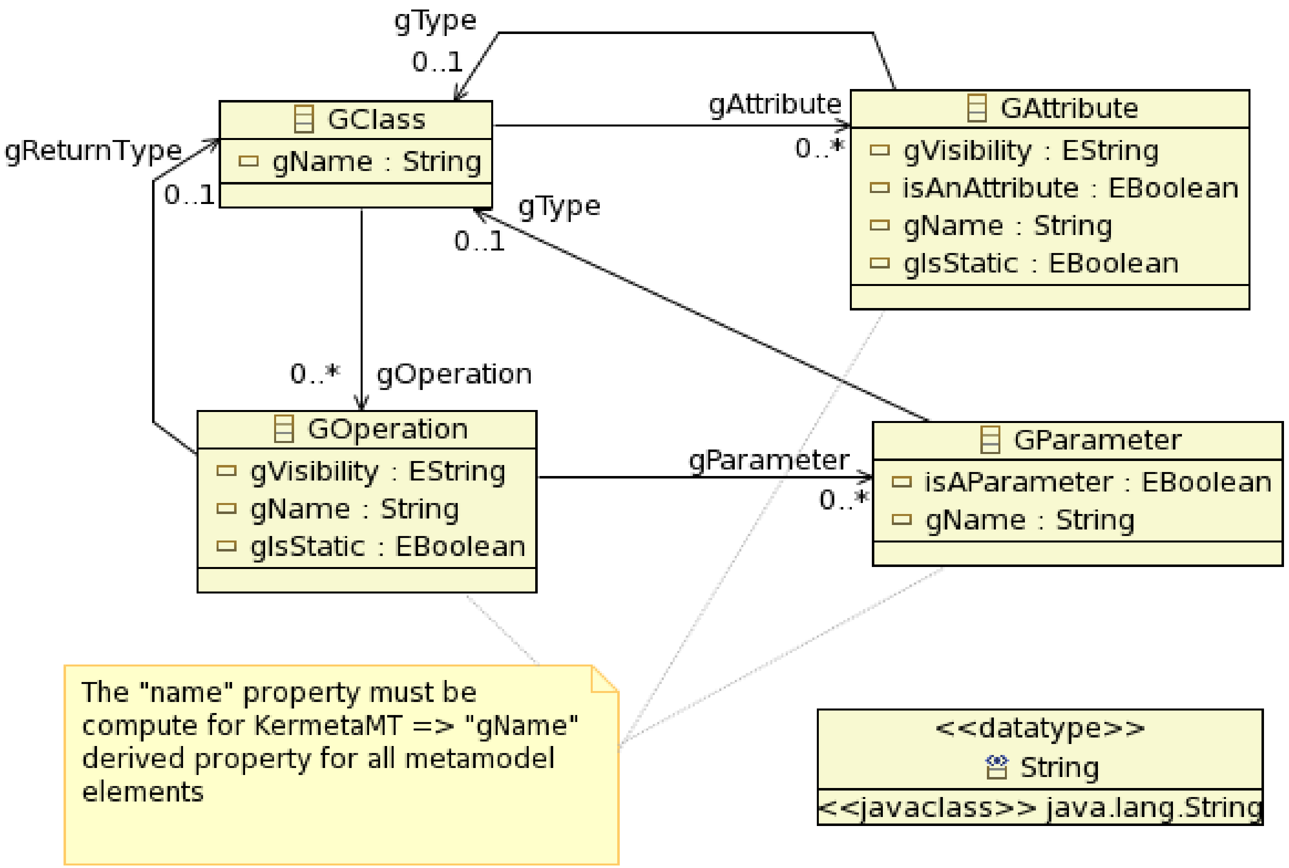
An effective solution

the **NonMatching Strategy**

NonMatching Strategy

The generic metamodel

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



GClass
- gName : String

GAttribute
- gVisibility : EString
- isAnAttribute : EBoolean
- gName : String
- gIsStatic : EBoolean

GOperation
- gVisibility : EString
- gName : String
- gIsStatic : EBoolean

GParameter
- isAParameter : EBoolean
- gName : String

<<datatype>>
String
<<javaclass>> java.lang.String

The "name" property must be compute for KermetaMT => "gName" derived property for all metamodel elements

NonMatching 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
}
```

NonMatching Strategy

We then adapt UML metamodel to add the generic elements through derived properties

```
// "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 ..]
```

NonMatching Strategy

We then adapt UML metamodel to add the generic elements through derived properties

```
// "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

NonMatching 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
}
```

NonMatching Strategy

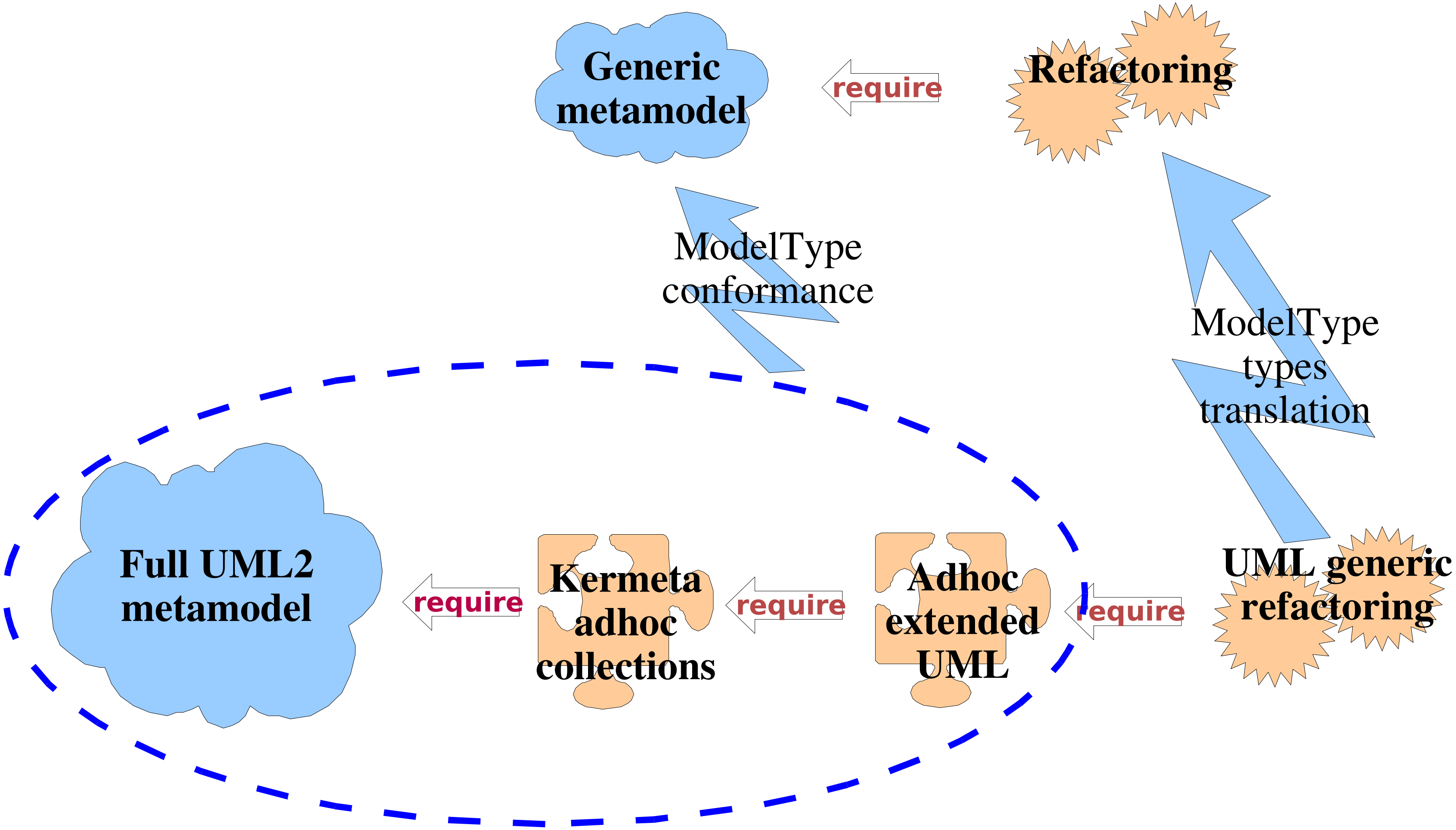
As we derive all generic features, we must include management of [0..*] multiplicities

We extend Kermeta collections to derived the generic references 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
}
```

NonMatching Strategy

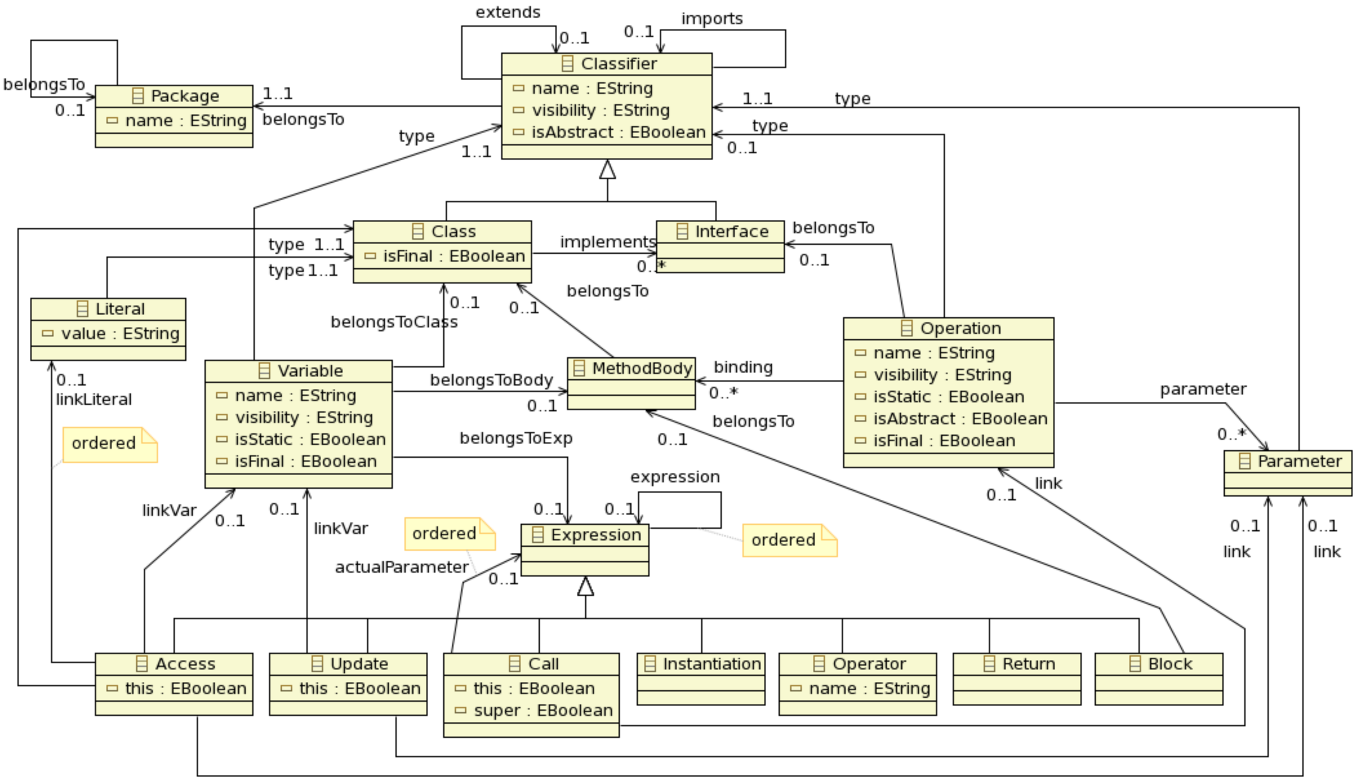
General scheme of the system



NonMatching Strategy

We want to refactor non UML models

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



NonMatching Strategy

We need

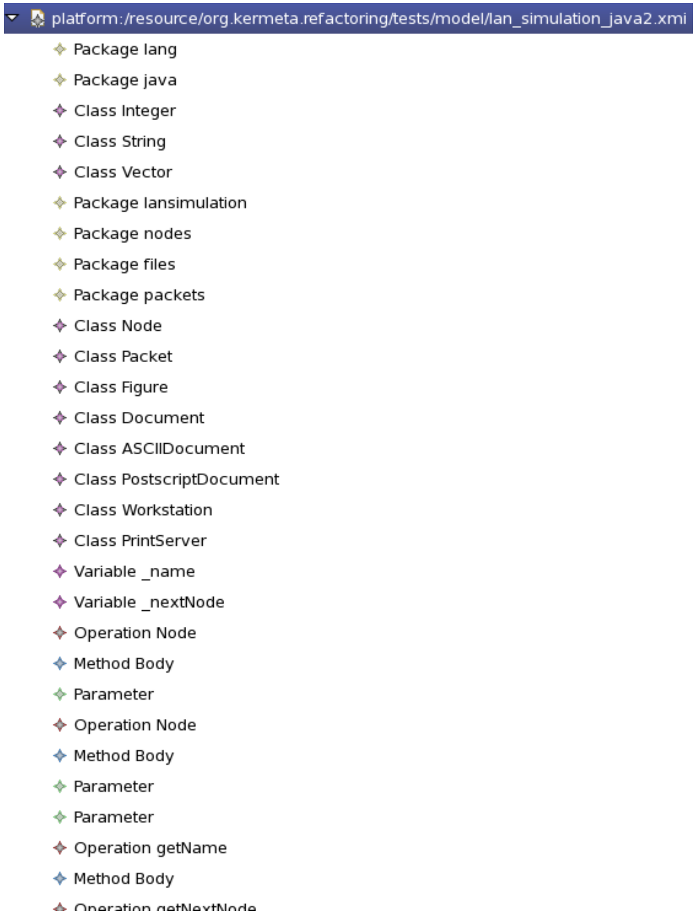
- Adhoc collections => JavaProgramHelper.kmt
- Derived properties => JavaProgramPlus.kmt
- ModelType => JavaProgramMT.kmt
- Launcher => JavaProgramGenericRefactoring.kmt

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

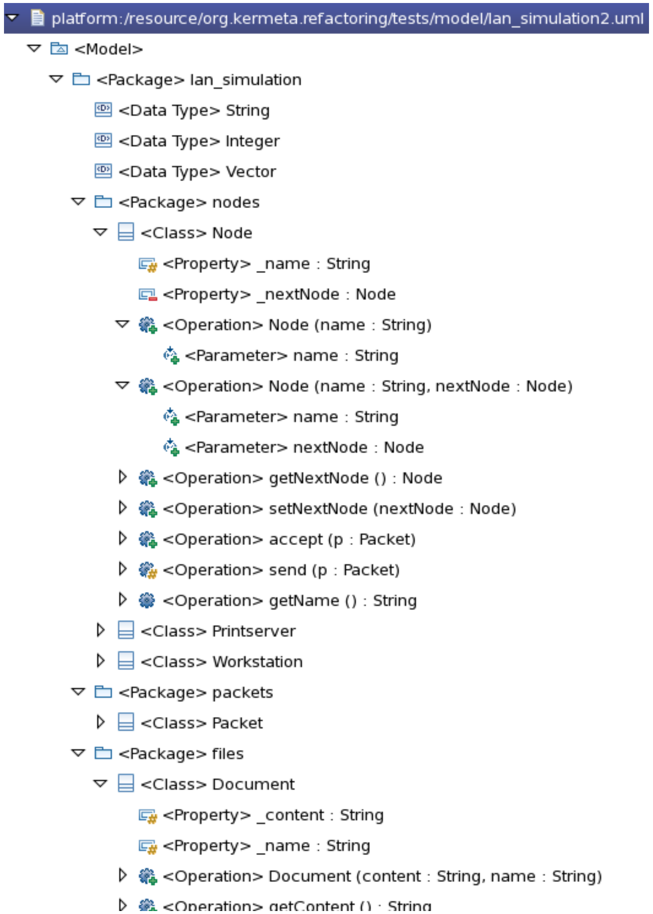
NonMatching Strategy

A flat model example



```
platform:/resource/org.kermeta.refactoring/tests/model/lan_simulation_java2.xmi
    Package lang
    Package java
    Class Integer
    Class String
    Class Vector
    Package lansimulation
    Package nodes
    Package files
    Package packets
    Class Node
    Class Packet
    Class Figure
    Class Document
    Class ASCIIDocument
    Class PostscriptDocument
    Class Workstation
    Class PrintServer
    Variable _name
    Variable _nextNode
    Operation Node
    Method Body
    Parameter
    Operation Node
    Method Body
    Parameter
    Parameter
    Operation getName
    Method Body
    Operation getNextNode
```

Similar UML model



```
platform:/resource/org.kermeta.refactoring/tests/model/lan_simulation2.uml
  <Model>
    <Package> lan_simulation
        <Data Type> String
        <Data Type> Integer
        <Data Type> Vector
      <Package> nodes
        <Class> Node
            <Property> _name : String
            <Property> _nextNode : Node
          <Operation> Node (name : String)
              <Parameter> name : String
          <Operation> Node (name : String, nextNode : Node)
              <Parameter> name : String
              <Parameter> nextNode : Node
          <Operation> getNextNode () : Node
          <Operation> setNextNode (nextNode : Node)
          <Operation> accept (p : Packet)
          <Operation> send (p : Packet)
          <Operation> getName () : String
        <Class> Printserver
        <Class> Workstation
      <Package> packets
        <Class> Packet
      <Package> files
        <Class> Document
            <Property> _content : String
            <Property> _name : String
          <Operation> Document (content : String, name : String)
          <Operation> getContent () : String
```

NonMatching Strategy

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
} }
```

NonMatching Strategy

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 ..]
}
```