Business Informatics Group

Vienna University of Technology

Reusing Model Transformations across Heterogeneous Metamodels*

M. Wimmer, A. Kusel, W. Retschitzegger, **J. Schönböck,**
W. Schwinger, J. S. Cuadrado, E. Guerra, and J. de Lara

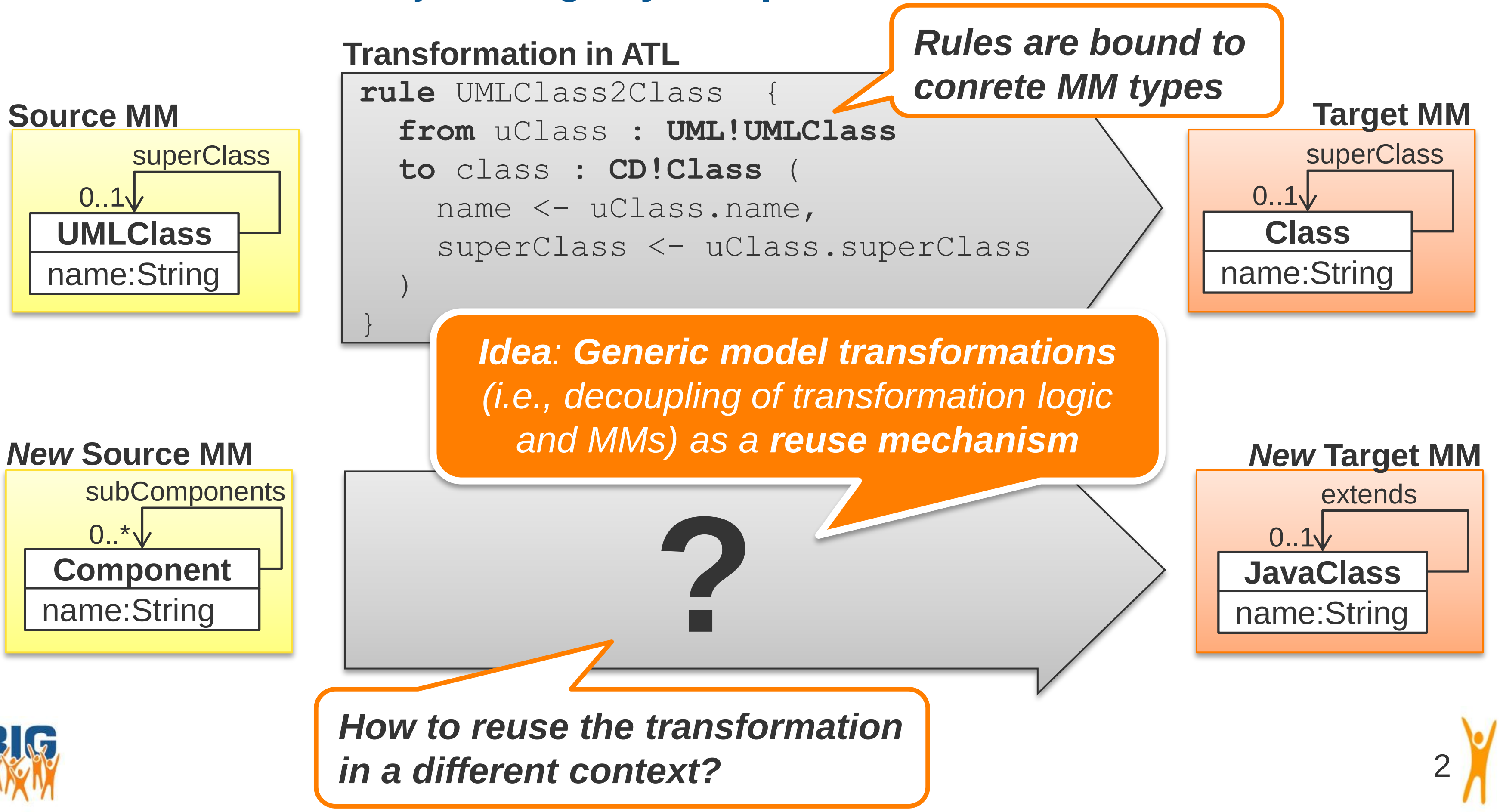
5th International Workshop on Multi-Paradigm Modeling (MPM 2011)

Johannes Schönböck
Business Informatics Group
Institute of Software Technology and Interactive Systems
Vienna University of Technology
Favoritenstraße 9-11/188-3, 1040 Vienna, Austria
phone: +43 (1) 58801-18804 (secretary), fax: +43 (1) 58801-18896
office@big.tuwien.ac.at, www.big.tuwien.ac.at

*This work has been partly funded by the Austrian Science Fund (FWF) under grant P21374-N13.

Motivation (1/2)

- **Model transformations** are **key** enablers for multi-paradigm modeling
- However: **little support** for **reusing transformations** in different contexts, since they are **tightly coupled** to **metamodels**



Transformation in ATL

```
rule UMLClass2Class  {
   from uClass : UML!UMLClass
   to class : CD!Class (
     name <- uClass.name,
     superClass <- uClass.superClass
   )
}
```

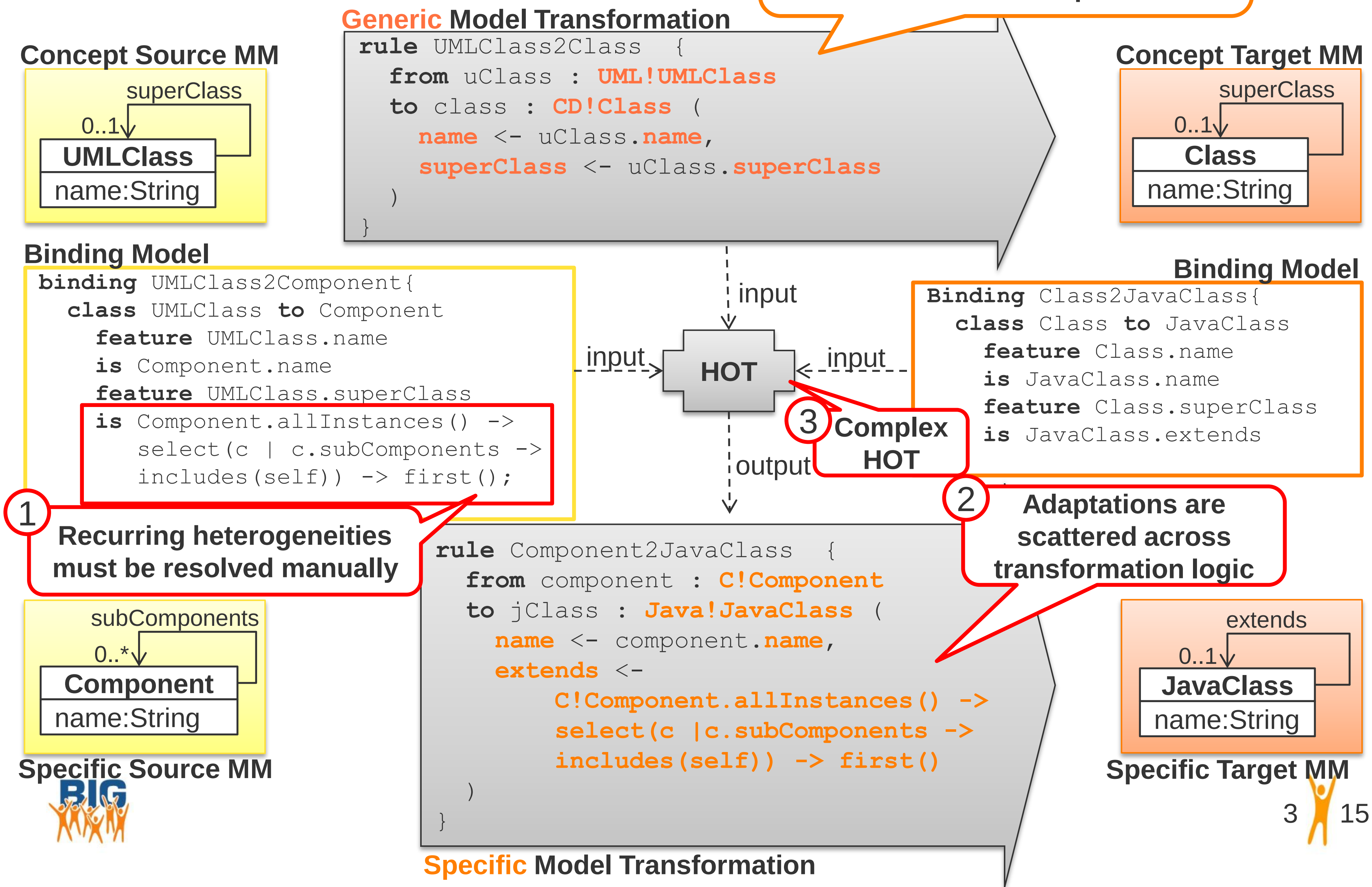
Rules are bound to conrete MM types

Idea: Generic model transformations *(i.e., decoupling of transformation logic and MMs) as a* ***reuse mechanism***

How to reuse the transformation in a different context?

Motivation (2/2)

Classes, attributes, and references of the concept MMs are variables that are bound to the specific MMs



Generic Model Transformation

```
rule UMLClass2Class  {
  from uClass : UML!UMLClass
  to class : CD!Class (
    name <- uClass.name,
    superClass <- uClass.superClass
  )
}
```

Binding Model

```
binding UMLClass2Component{
  class UMLClass to Component
    feature UMLClass.name
    is Component.name
    feature UMLClass.superClass
    is Component.allInstances() ->
       select(c | c.subComponents ->
       includes(self)) -> first();
```

Binding Model

```
Binding Class2JavaClass{
  class Class to JavaClass
    feature Class.name
    is JavaClass.name
    feature Class.superClass
    is JavaClass.extends
```

3 Complex HOT

1 Recurring heterogeneities must be resolved manually

2 Adaptations are scattered across transformation logic

```
rule Component2JavaClass  {
  from component : C!Component
  to jClass : Java!JavaClass (
    name <- component.name,
    extends <-
        C!Component.allInstances() ->
        select(c |c.subComponents ->
        includes(self)) -> first()
  )
}
```

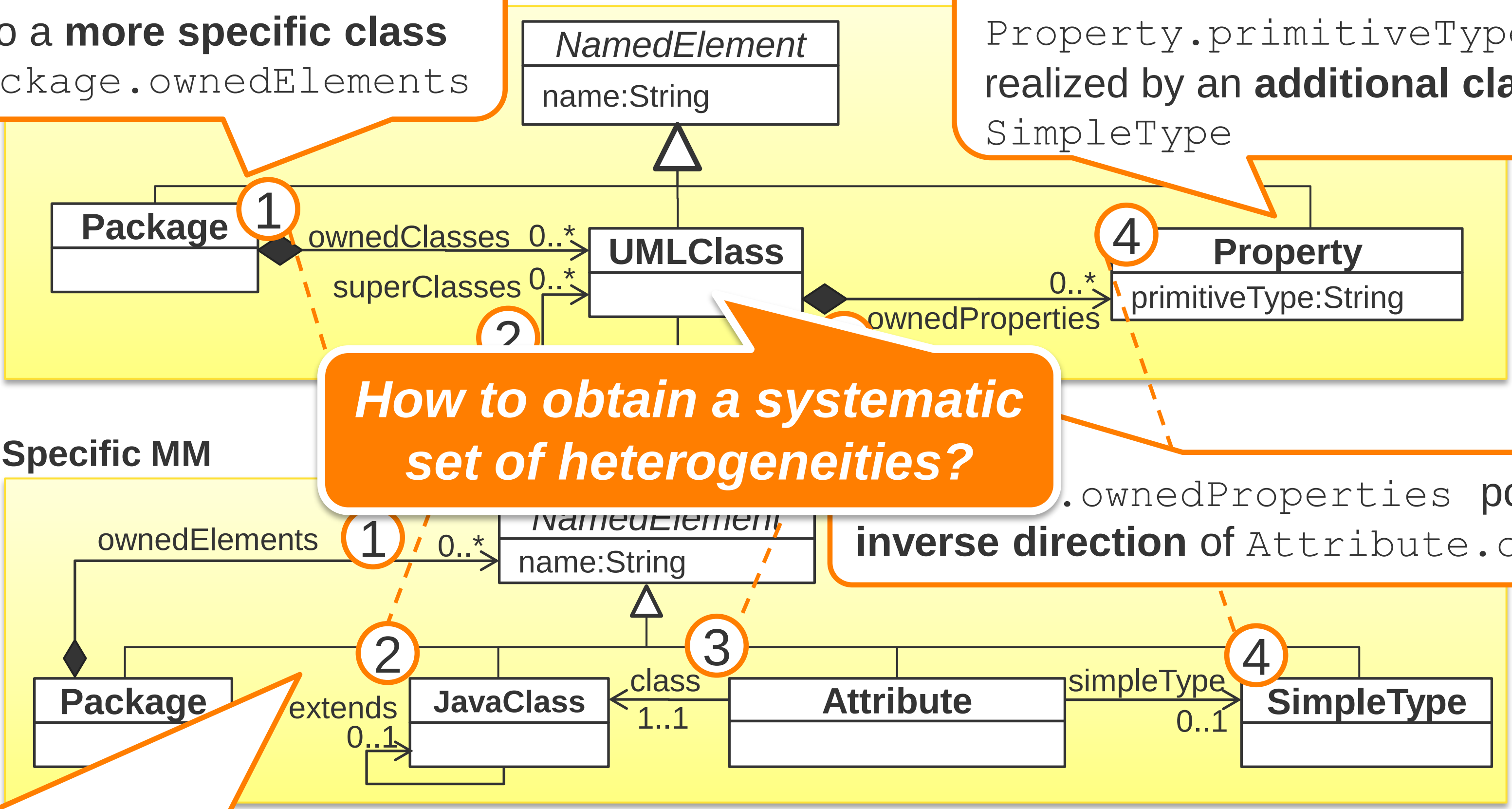
Specific Model Transformation

Approach

- **Problem 1: Recurring heterogeneities must be resolved manually**
  - For resolving common heterogeneities, a **library** of generic and composable **adapters** in t... form of **templates** is proposed
  - **Adapters** realize a **vi** **required features of** s established
  - **Selection** of adapters happens **automatically** by analyzing **bindings** of the binding model
- **Problem 2: Adaptations are scattered across transformation logic**
  - **Adapters** are realized by means of **helper functions**
  - Consequently, **adapters** are **added** to the transformation, but **not intermingled** with the transformation
- **Problem 3: Complex HOT**
  - **Templates** exist for adapters; these may be easily instantiated, **without** the need of **analyzing** and **rewriting** existing **transformation code**

Question 1:
What are common heterogeneities?

Exemplary Heterogeneities between MMs



1 `Package.ownedClasses` **refers** to a **more specific class** than `Package.ownedElements`

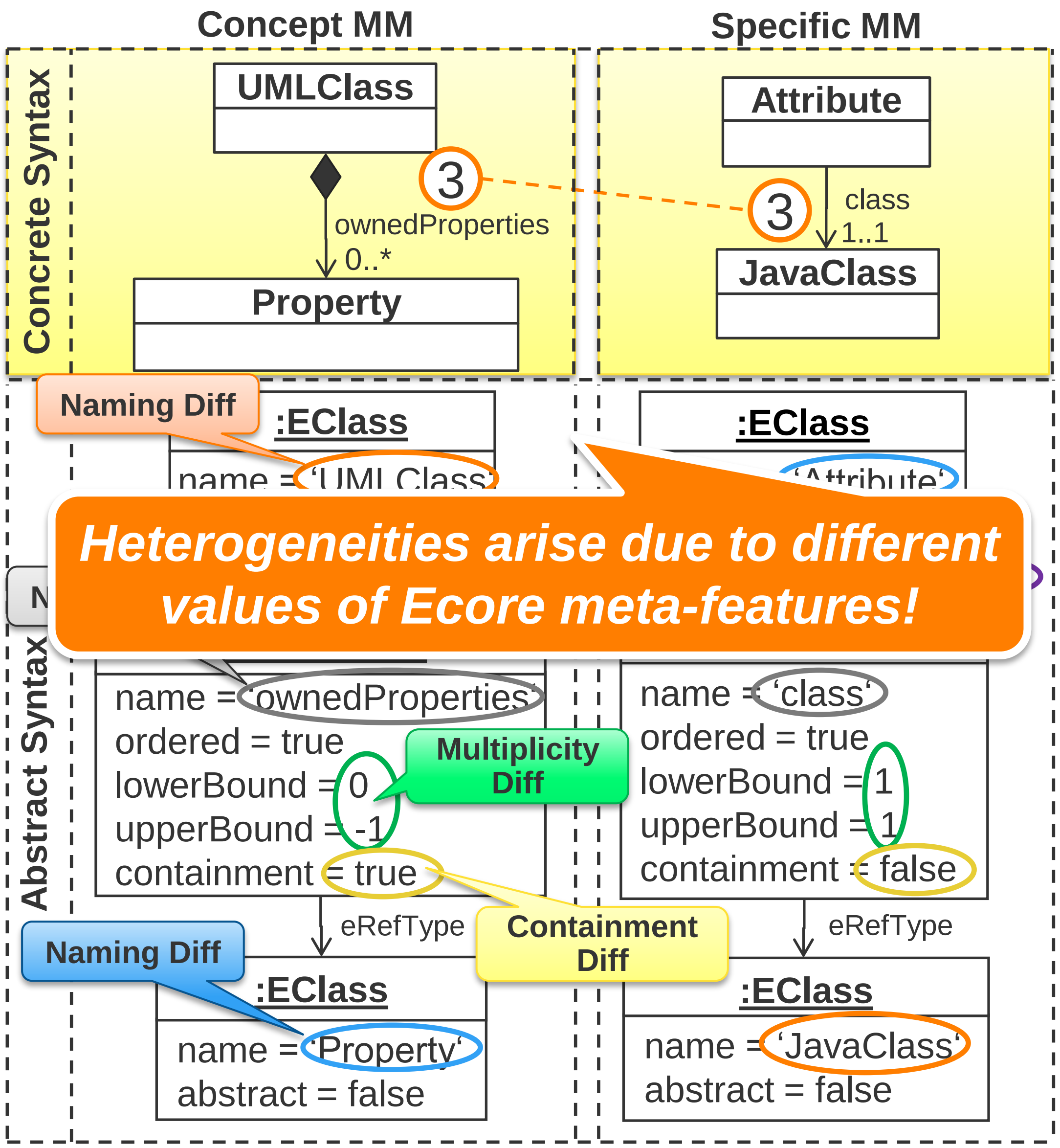
4 The **attribute** `Property.primitiveType` is realized by an **additional class** `SimpleType`

How to obtain a systematic set of heterogeneities?

`.ownedProperties` points to **inverse direction** of `Attribute.class`

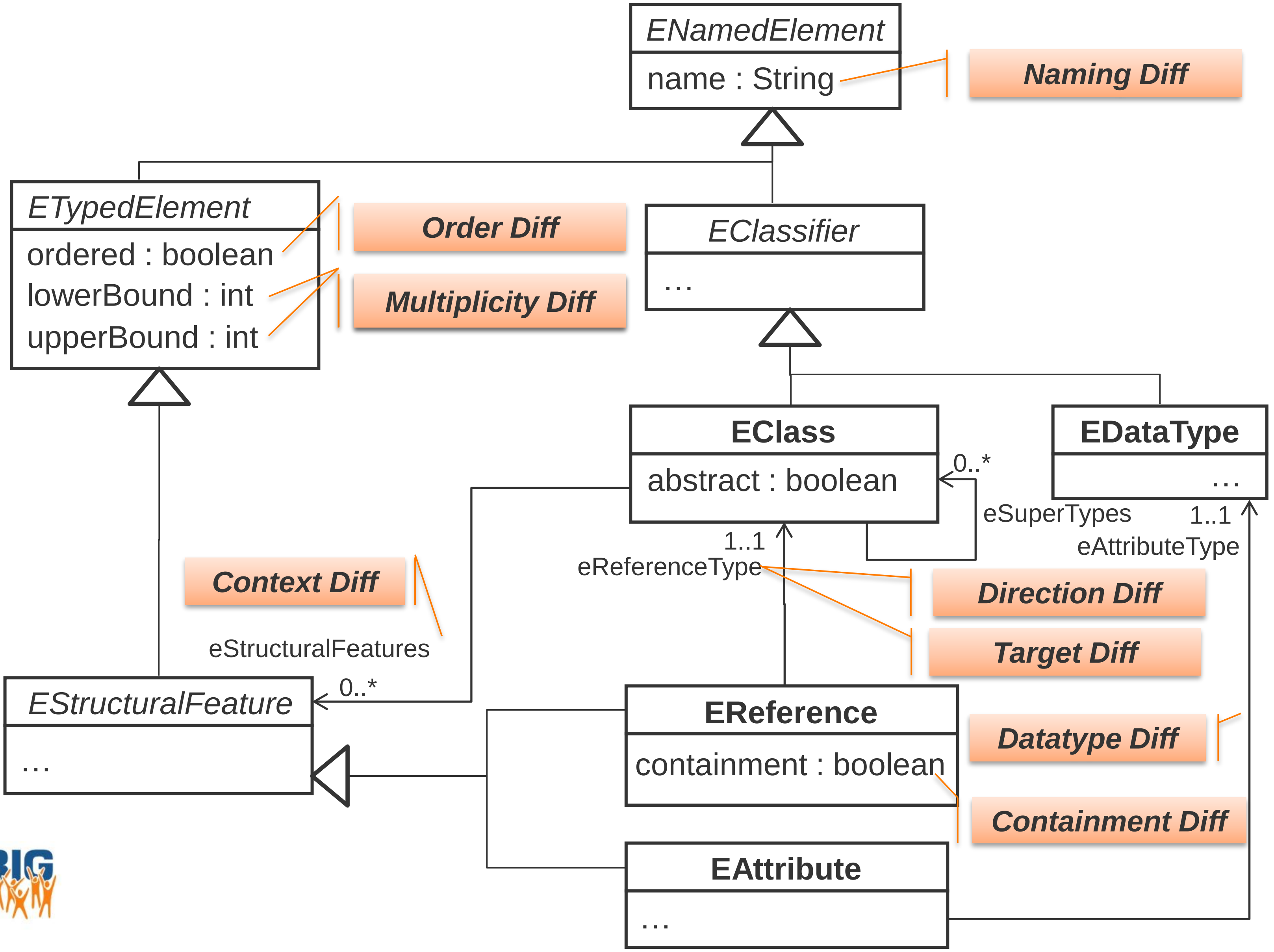
2 `UMLClass.superClasses` allows for **multiple inheritance,** whereas `JavaClass.extends` allows for **single inheritance,** only

Analysis of Heterogeneity #3



Heterogeneities arise due to different values of Ecore meta-features!

Systematic Set of Heterogeneities



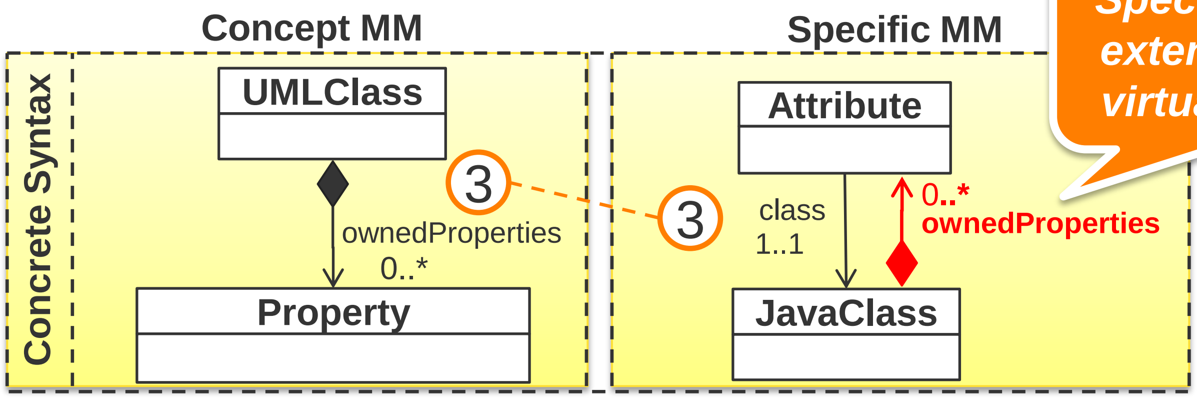
Approach

- **Problem 1: Recurring heterogeneities must be resolved manually**
  - For resolving common heterogeneities, a **library** of generic and composable **adapters** in the form of **templates** is proposed
  - **Adapters** realize **irtual view** on the specific MM, which **provide required featur** **ent MM** → a *subtype relationship* is establis
  - **Selecti** **gs** of the binding model

Question 2:
How to resolve the heterogenties by adapters?

- **Problem 2: Adaptations are scattered across transformation logic**
  - **Adapters** are realized by means of **helper functions**
  - Consequently, **adapters** are **added** to the transformation, but **not intermingled** with the transformation
- **Problem 3: Complex HOT**
  - **Templates** exist for adapters; these may be easily instantiated, **without** the need of **analyzing** and **rewriting** existing **transformation code**

Exemplary Adapter by Means of a Helper Function



Specific MM is extended by a virtual feature

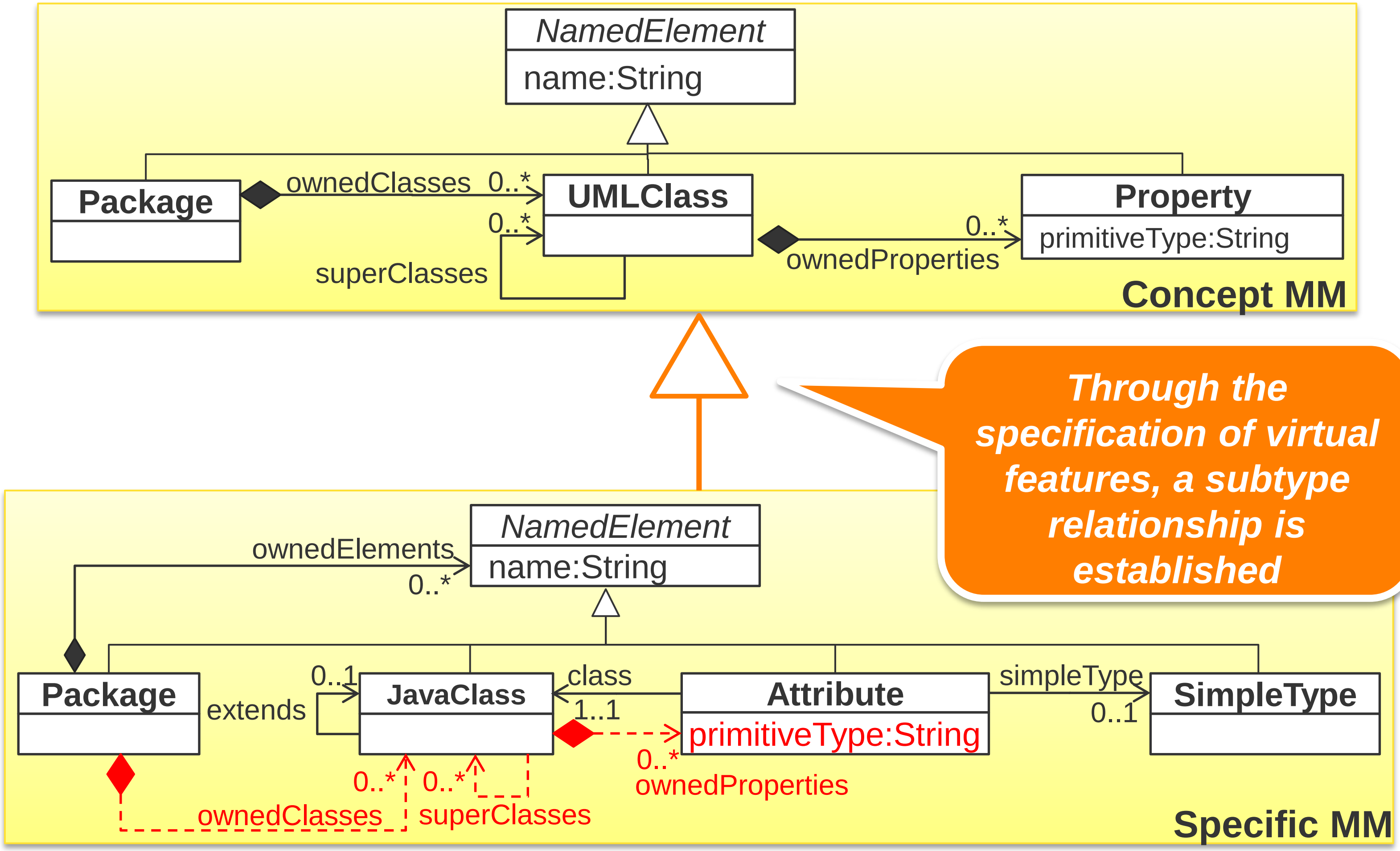
```
helper context Java!JavaClass
   Sequence(Java!Att   bute) =                     >
                                     select(a| a.class = self);
```

If a generic transformation accesses `JavaClass.ownedProperties` on the specific MM → an error arises!

Virtual feature is realized by a helper f nction

Now, the access to `JavaClass.ownedProperties` is enabled!

Subtype Relationship



Through the specification of virtual features, a subtype relationship is established

Approach

- **Problem 1: Recurring heterogeneities must be resolved manually**
  - For resolving common heterogeneities, a **library** of generic and composable **adapters** in the form of **templates** is proposed
  - **Adapters** realize a **virtual view** on the specific MM, which **provide required features of the concept MM** → a ***subtype* relationship** is established
  - **Selection** of adapters happens **automatically** by analyzing **bindings** of the binding model
- **Problem 2: Adaptations are scattered acr ormation logic**
  - **Adapters** are
  - Consequently **not intermingled** with the transformation
- **Problem 3: Complex HOT**
  - **Templates** exist for adapters; these may be easily instantiated, **without** the need of **analyzing** and **rewriting** existing **transformation code**

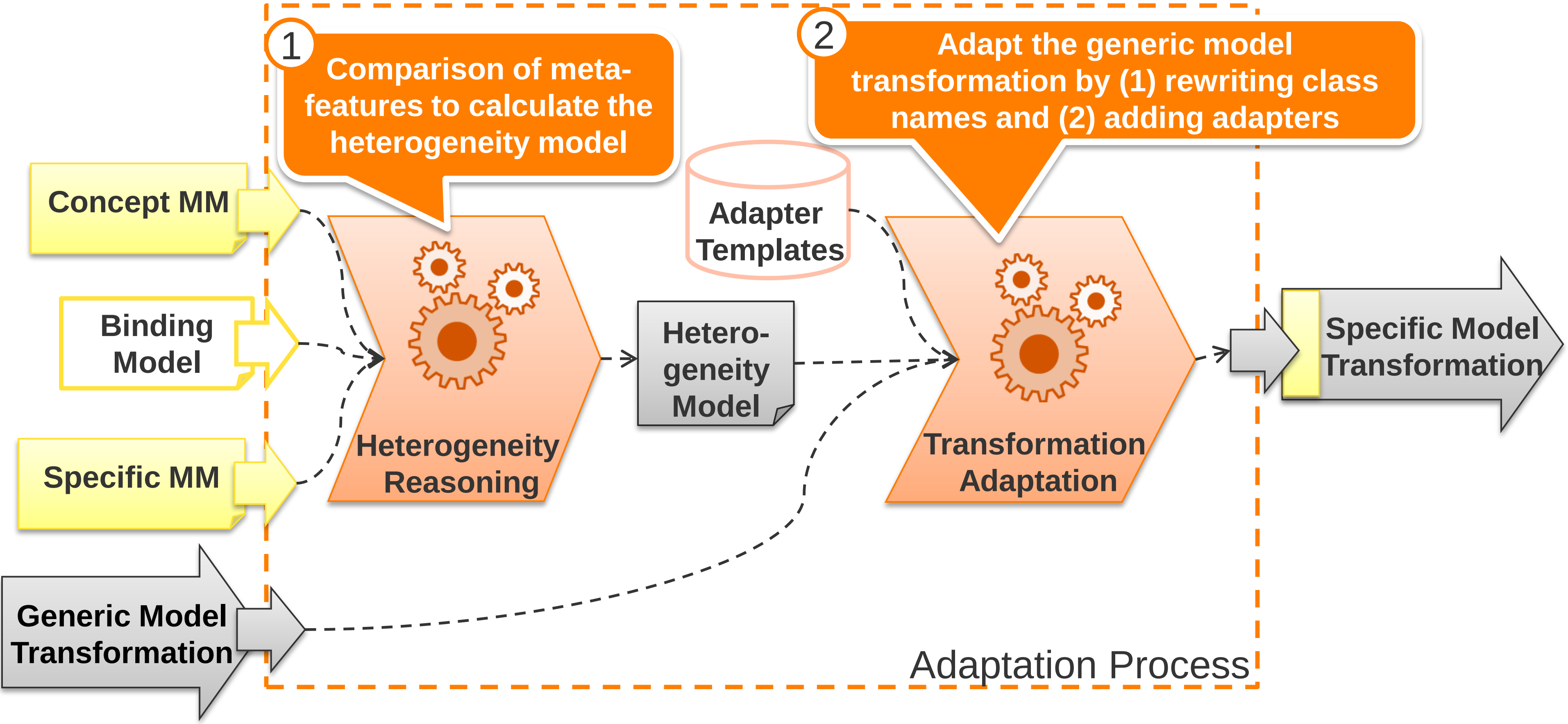
Question 3:
How does the binding model look like?

Binding Model

```
binding UML2Java{
  class Package to Package
    feature Package.ownedClasses ①
    is Package.ownedElements
  class UMLClass to JavaClass
    feature UMLClass.superClasses ②
    is JavaClass.extends
    feature UMLClass.ownedProperties ③
    is Attribute.class
  class Property to Attribute
    feature Property.primitiveType ④
    is Attribute.simpleType.name
}
```

1:1 bindings suffice!

Adaptation Process


1
Comparison of meta-features to calculate the heterogeneity model
2
Adapt the generic model transformation by (1) rewriting class names and (2) adding adapters
Concept MM
Binding Model
Specific MM
Generic Model Transformation
Heterogeneity Reasoning
Adapter Templates
Hetero-geneity Model
Transformation Adaptation
Specific Model Transformation
Adaptation Process

Template for Resolving Target Difference

Template

```
helper context <specificRef.owner> def : <conceptRef.name> :
<conceptRef.type.resolve> =
self.<specificRef.name> -> select(x|
x.oclIsKindOf(<conceptRef.type.resolve>));
```

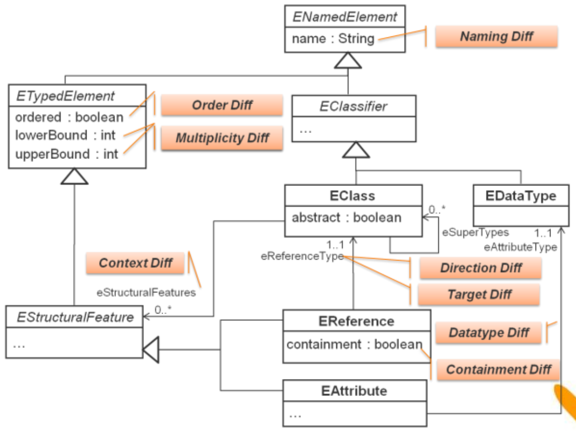
```
helper context Java!Package def : ownedClasses :
Sequence(Java!JavaClass) =
self.ownedElements -> select(x|
x.oclIsKindOf(Java!JavaClass));
```

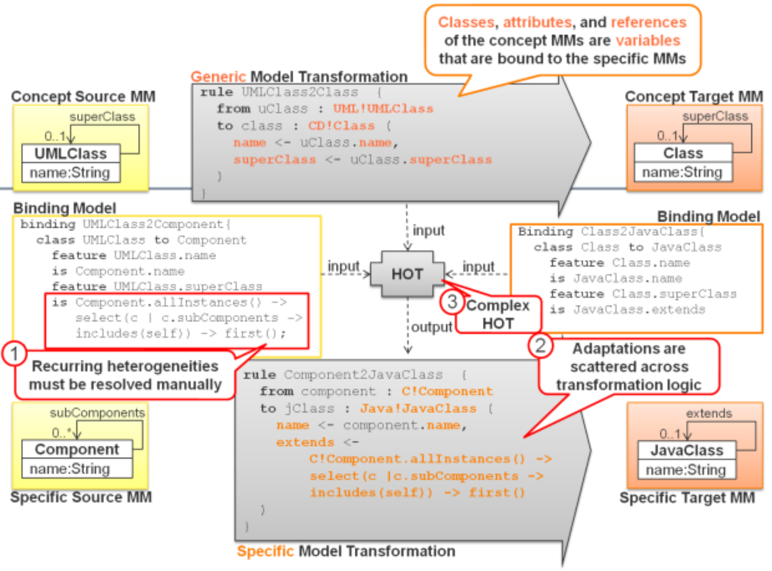
Exemplary Instantiation of Template

Future Work

- Handling Heterogeneities between Classes
  - So far, only differences between attributes and references have been considered
  - Definition of virtual classes by means of helper functions would be required to consider also differences between classes
- Reusing Transformations for Specific Target MMs
  - So far, only adaptations of the source MM have been performed
  - This is, since it is not possible to query the target model to provide virtual features
- Specialization of Constraints
  - Also constraints on the concept MMs have to be translated for specific MMs

Thank you for your attention!




Classes, attributes, and references of the concept MMs are variables that are bound to the specific MMs
Generic Model Transformation
Concept Source MM
superClass
0..1
UMLClass
name:String
Concept Target MM
superClass
0..1
Class
name:String
Binding Model
input
HOT
input
output
3 Complex HOT
1 Recurring heterogeneities must be resolved manually
2 Adaptations are scattered across transformation logic
subComponents
0..*
Component
name:String
Specific Source MM
extends
0..1
JavaClass
name:String
Specific Target MM
Specific Model Transformation

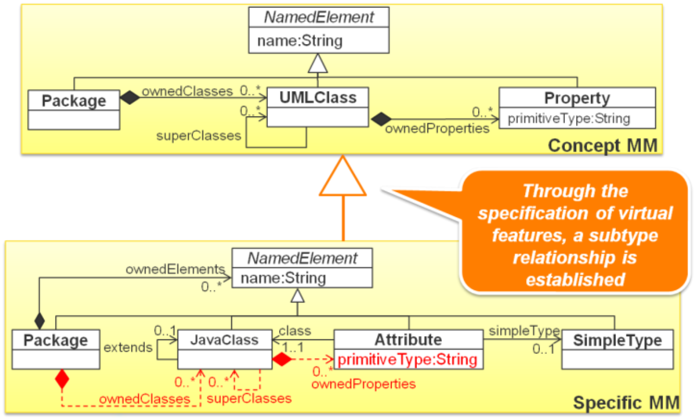
```
rule UMLClass2Class  {
  from uClass : UML!UMLClass
  to class : CD!Class (
    name <- uClass.name,
    superClass <- uClass.superClass
  )
}
```

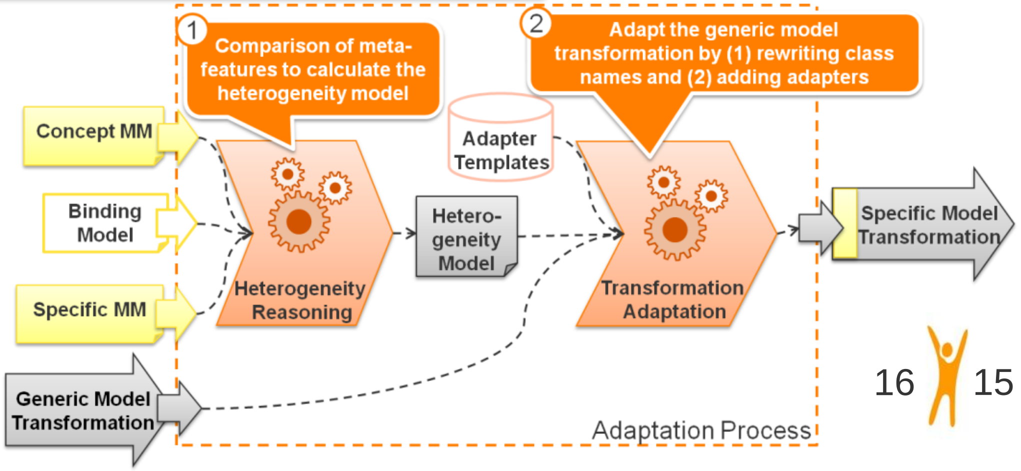
```
binding UMLClass2Component{
  class UMLClass to Component
    feature UMLClass.name
    is Component.name
    feature UMLClass.superClass
    is Component.allInstances() ->
       select(c | c.subComponents ->
       includes(self)) -> first();
```

```
Binding Class2JavaClass{
  class Class to JavaClass
    feature Class.name
    is JavaClass.name
    feature Class.superClass
    is JavaClass.extends
```

```
rule Component2JavaClass  {
  from component : C!Component
  to jClass : Java!JavaClass (
    name <- component.name,
    extends <-
        C!Component.allInstances() ->
        select(c |c.subComponents ->
        includes(self)) -> first()
  )
}
```

http://www.modeltransformation.net




1 Comparison of meta-features to calculate the heterogeneity model
2 Adapt the generic model transformation by (1) rewriting class names and (2) adding adapters
Concept MM
Binding Model
Specific MM
Generic Model Transformation
Heterogeneity Reasoning
Adapter Templates
Hetero-geneity Model
Transformation Adaptation
Specific Model Transformation
Adaptation Process