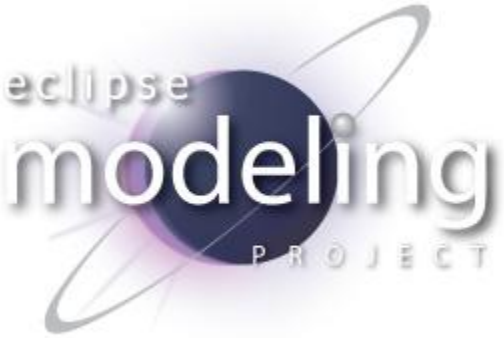
Practice of Model Transformation and Model Weaving in the Eclipse Modeling Project with ATL and AMW



Part 1
ATL: the ATLAS Transformation Language

Mikaël Barbero (1)
Marcos Didonet Del Fabro (1)
Juan M. Vara (2)

(1) ATLAS Group (INRIA & LINA), University of Nantes (France)
(2) Kybele Research Group, University Rey Juan Carlos of Madrid (Spain)

Prerequisites

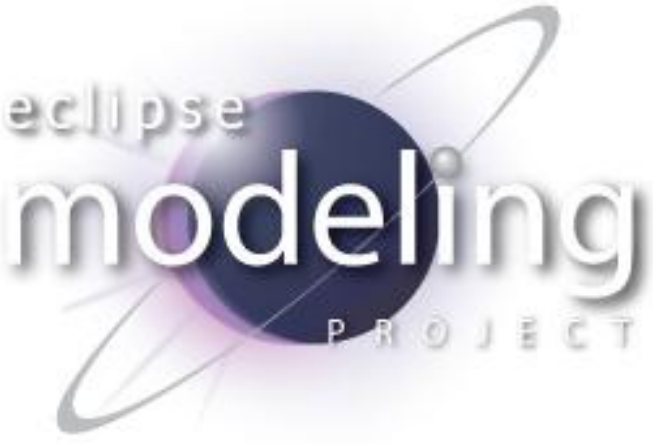
To be able to understand this lecture, a reader should be familiar with the following concepts, languages, and standards:

- Model Driven Engineering (MDE)
- The role of model transformations in MDE
- UML
- OCL
- MOF
- Basic programming concepts

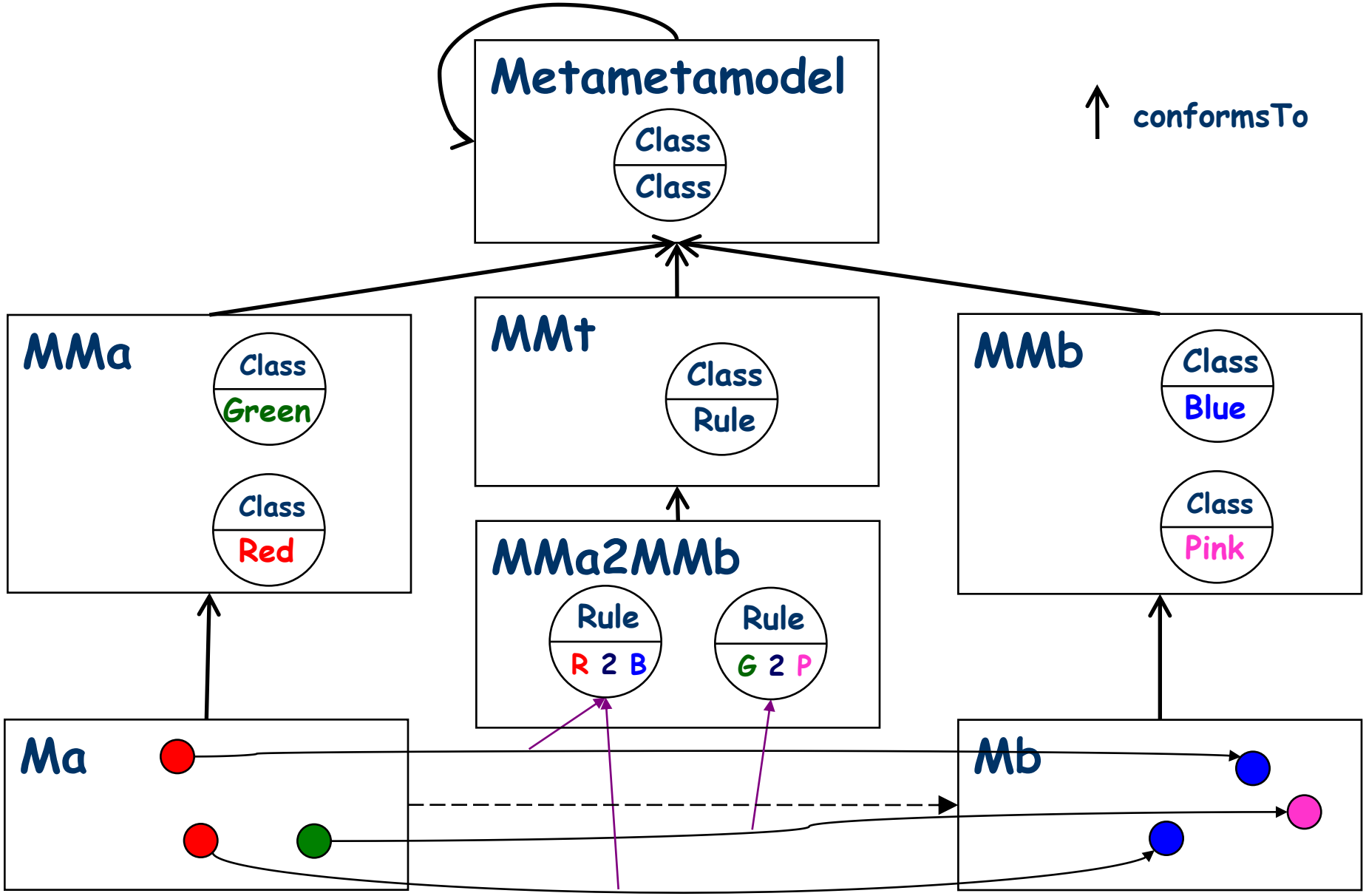
Contents

Introduction: ATL

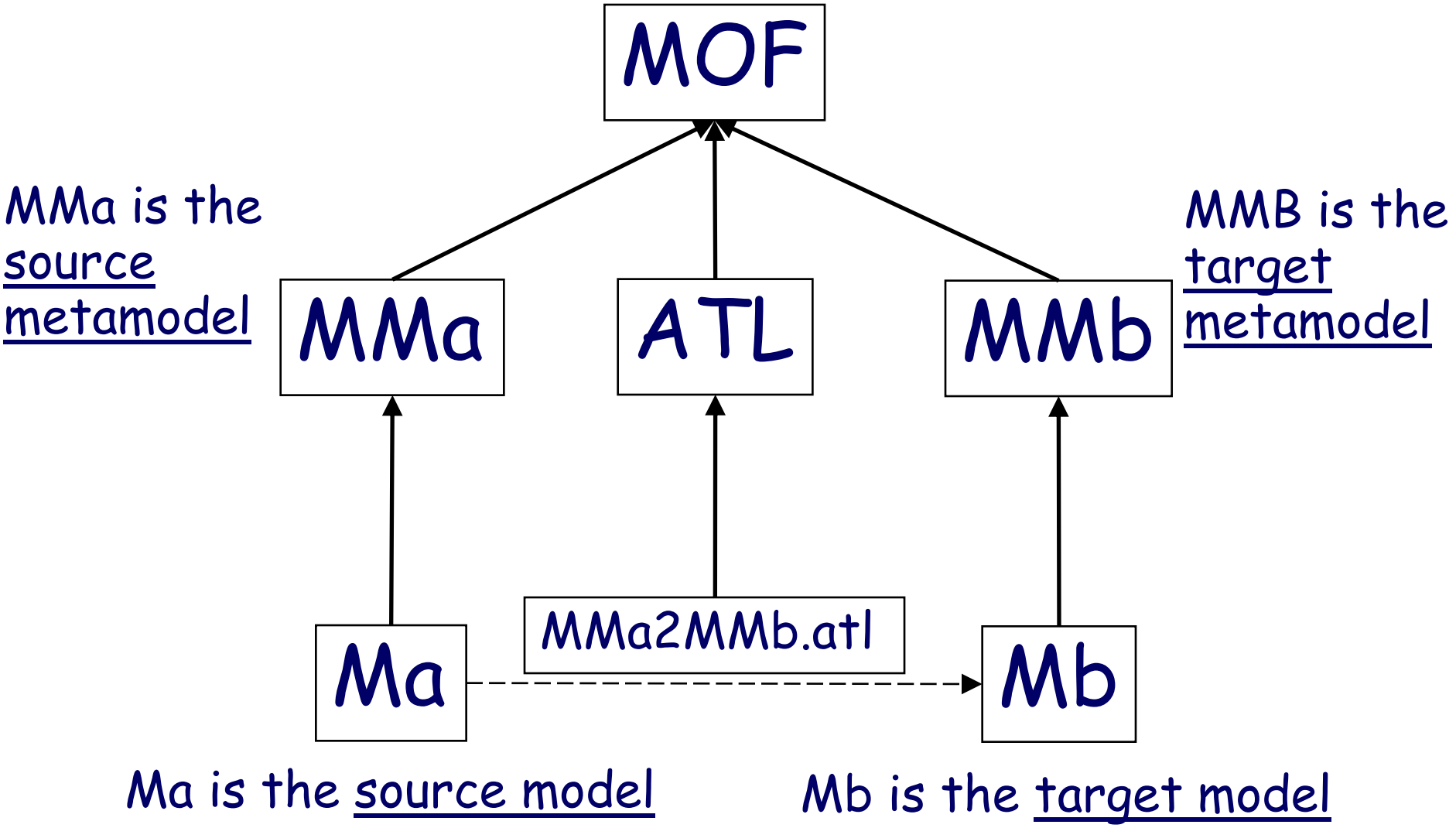
- ATL is a language and a Virtual Machine dedicated to model transformation
- ATL is a component of Eclipse Model-to-Model (M2M) project within the Eclipse Modeling Project (EMP)



Introduction: Model Transformation



Operational context of ATL



Contents

ATL in a nutshell

- Source models and target models are distinct:
  - Source models are read-only (they can only be navigated, not modified),
  - Target models are write-only (they cannot be navigated).
- The language is a declarative-imperative hybrid:
  - Declarative part:
    - Matched rules with automatic traceability support,
    - Side-effect free navigation (and query) language: OCL 2.0
  - Imperative part:
    - Called rules,
    - Action blocks.
- Recommended programming style: declarative

ATL overview (continued)

- A declarative rule specifies:
  - a source pattern to be matched in the source models,
  - a target pattern to be created in the target models for each match during rule application.
- An imperative rule is basically a procedure:
  - It is called by its name,
  - It may take arguments,
  - It can contain:
    - A declarative target pattern,
    - An action block (i.e. a sequence of statements),
    - Both.

ATL overview (continued)

- Applying a declarative rule means:
  - Creating the specified target elements,
  - Initializing the properties of the newly created elements.
- There are three types of declarative rules:
  - Standard rules that are applied once for each match,
    - A given set of elements may only be matched by one standard rule,
  - Lazy rules that are applied as many times for each match as it is referred to from other rules (possibly never for some matches),
  - Unique lazy rules that are applied at most once for each match and only if it is referred to from other rules.

Declarative rules: source pattern

- The source pattern is composed of:
  - A labeled set of types coming from the source metamodels,
  - A guard (Boolean expression) used to filter matches.
- A match corresponds to a set of elements coming from the source models that:
  - Are of the types specified in the source pattern (one element for each type),
  - Satisfy the guard.

Declarative rules: target pattern

- The target pattern is composed of:
  - A labeled set of types coming from the target metamodels,
  - For each element of this set, a set of bindings.
  - A binding specifies the initialization of a property of a target element using an expression.
- For each match, the target pattern is applied:
  - Elements are created in the target models (one for each type of the target pattern),
  - Target elements are initialized by executing the bindings:
    - First evaluating their value,
    - Then assigning this value to the corresponding property.

Execution order of declarative rules

- Declarative ATL frees the developer from specifying execution order:
  - The order in which rules are matched and applied is not specified.
    - Remark: the match of a lazy or unique lazy rules must be referred to before the rule is applied.
  - The order in which bindings are applied is not specified.
- The execution of declarative rules can however be kept deterministic:
  - The execution of a rule cannot change source models
    - ➔ It cannot change a match,
  - Target elements are not navigable
    - ➔ The execution of a binding cannot change the value of another.

Contents

Use Case 1

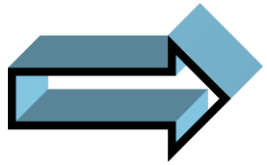
Families to Persons

Families to Persons: Overview

- Initially we have a text describing a list of families.
- We want to transform this list into another one describing a list of persons.

Transforming this …

…
Family: Simpsons
Father: Homer
Mother: Marge
Son: Bart
Daughter: Lisa
… other Families



… into this.

…
Mr. Homer Simpson
Mrs. Marge Simpson
Mr. Bart Simpson
Mrs. Lisa Simpson
… other Persons

Let's suppose these are not text, but models.

Input of the transformation is a model

Family: Simpson
  Father: Homer
  Mother: Marge
  Son: Bart
  Daughter: Lisa
  Daughter: Maggie
Family: Skywalker
  Father: Anakin
  Mother: Padme
  Son: Luke
  Daughter: Leia

This is the text.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<xmi:XMI xmi:version="2.0"
xmlns:xmi="http://www.omg.org/XMI"
xmlns="Families">
  <Family lastName="Skywalker">
    <father firstName="Anakin"/>
    <mother firstName="Padme"/>
    <sons firstName="Luke"/>
    <daughters firstName="Leia"/>
  </Family>
  <Family lastName="Simpsons">
    <father firstName="Homer"/>
    <mother firstName="Marge"/>
    <sons firstName="Bart"/>
    <daughters firstName="Lisa"/>
    <daughters firstName="Maggie"/>
  </Family>
</xmi:XMI>
```

This is the corresponding model.
It is expressed in XMI,
a standard way to represent models.

Output of the transformation should be a model

Mr. Anakin Skywalker
Mrs. Padme Skywalker
Mr. Luke Skywalker
Mrs. Leia Skywalker
Mr. Homer Simpson
Mrs. Marge Simpson
Mr. Bart Simpson
Mrs. Lisa Simpson
Mrs. Maggie Simpson

This is the text.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<xmi:XMI xmi:version="2.0"
xmlns:xmi="http://www.omg.org/XMI"
xmlns="Persons">
  <Man fullName="Anakin Skywalker"/>
  <Woman fullName="Padme Skywalker"/>
  <Man fullName="Luke Skywalker"/>
  <Woman fullName="Leia Skywalker"/>
  <Man fullName="Homer Simpsons"/>
  <Woman fullName="Marge Simpsons"/>
  <Man fullName="Bart Simpsons"/>
  <Woman fullName="Lisa Simpsons"/>
  <Woman fullName="Maggie Simpsons"/>
</xmi:XMI>
```

This is the corresponding model
(The corresponding XMI file is named "sample-Persons.ecore").

Each model conforms to a metamodel

conformsTo

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<xmi:XMI xmi:version="2.0"
xmlns:xmi="http://www.omg.org/XMI"
xmlns="Families">
  <Family lastName="Skywalker">
    <father firstName="Anakin"/>
    <mother firstName="Padme"/>
    <sons firstName="Luke"/>
    <daughters firstName="Leia"/>
  </Family>
  <Family lastName="Simpsons">
    <father firstName="Homer"/>
    <mother firstName="Marge"/>
    <sons firstName="Bart"/>
    <daughters firstName="Lisa"/>
    <daughters firstName="Maggie"/>
  </Family>
</xmi:XMI>
```

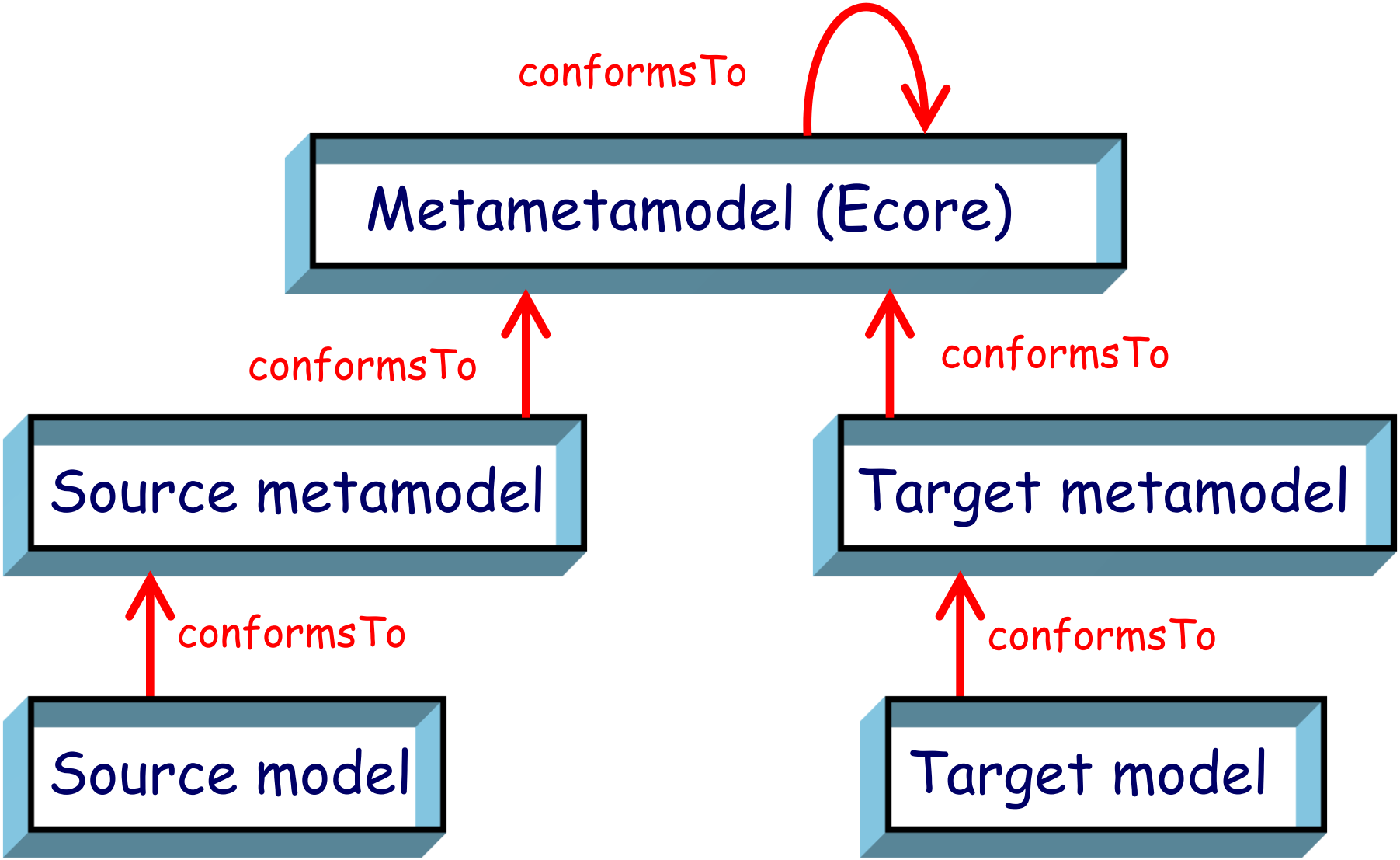
Source model
"sample-Families.xmi"

conformsTo

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<xmi:XMI xmi:version="2.0"
xmlns:xmi="http://www.omg.org/XMI"
xmlns="Persons">
  <Man fullName="Anakin Skywalker"/>
  <Woman fullName="Padme Skywalker"/>
  <Man fullName="Luke Skywalker"/>
  <Woman fullName="Leia Skywalker"/>
  <Man fullName="Homer Simpsons"/>
  <Woman fullName="Marge Simpsons"/>
  <Man fullName="Bart Simpsons"/>
  <Woman fullName="Lisa Simpsons"/>
  <Woman fullName="Maggie Simpsons"/>
</xmi:XMI>
```

Target model
"sample-Persons.xmi"

The general picture

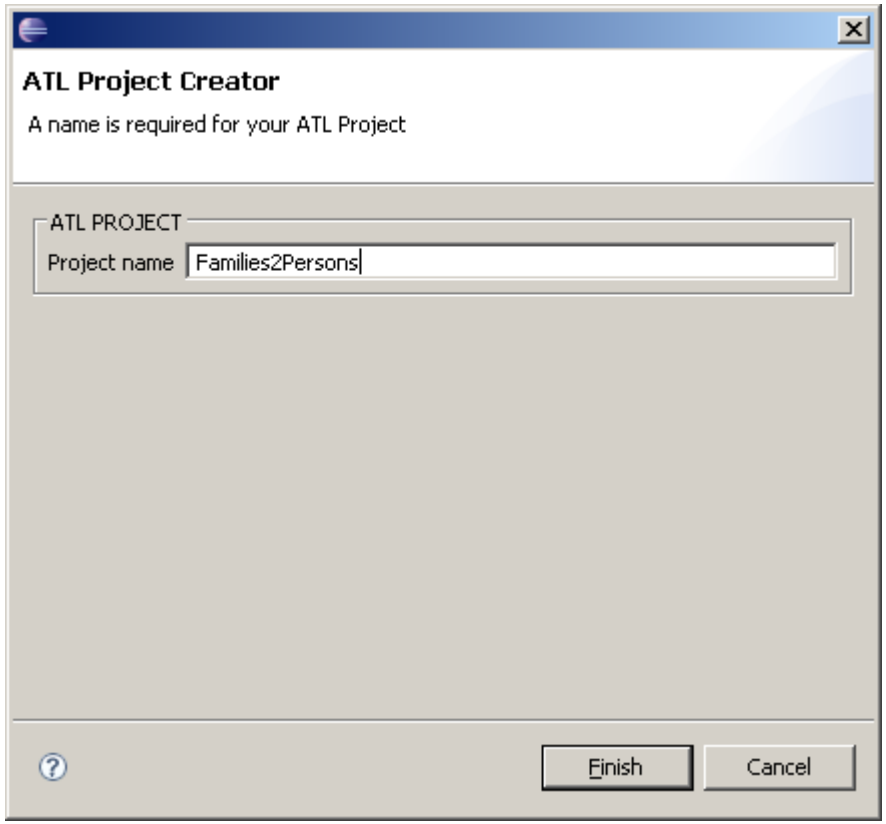


What we need to provide

- In order to achieve the transformation, we need to provide:
  1. A source metamodel in KM3 ("Families").
  2. A source model (in XMI) conforming to "Families".
  3. A target metamodel in KM3 ("Persons").
  4. A transformation model in ATL ("Families2Persons").
- When the ATL transformation is executed, we obtain:
  - A target model (in XMI) conforming to "Persons".

Families to Persons: project creation

- First we create an ATL project by using the ATL Project Wizard.



Definition of the source metamodel "Families"

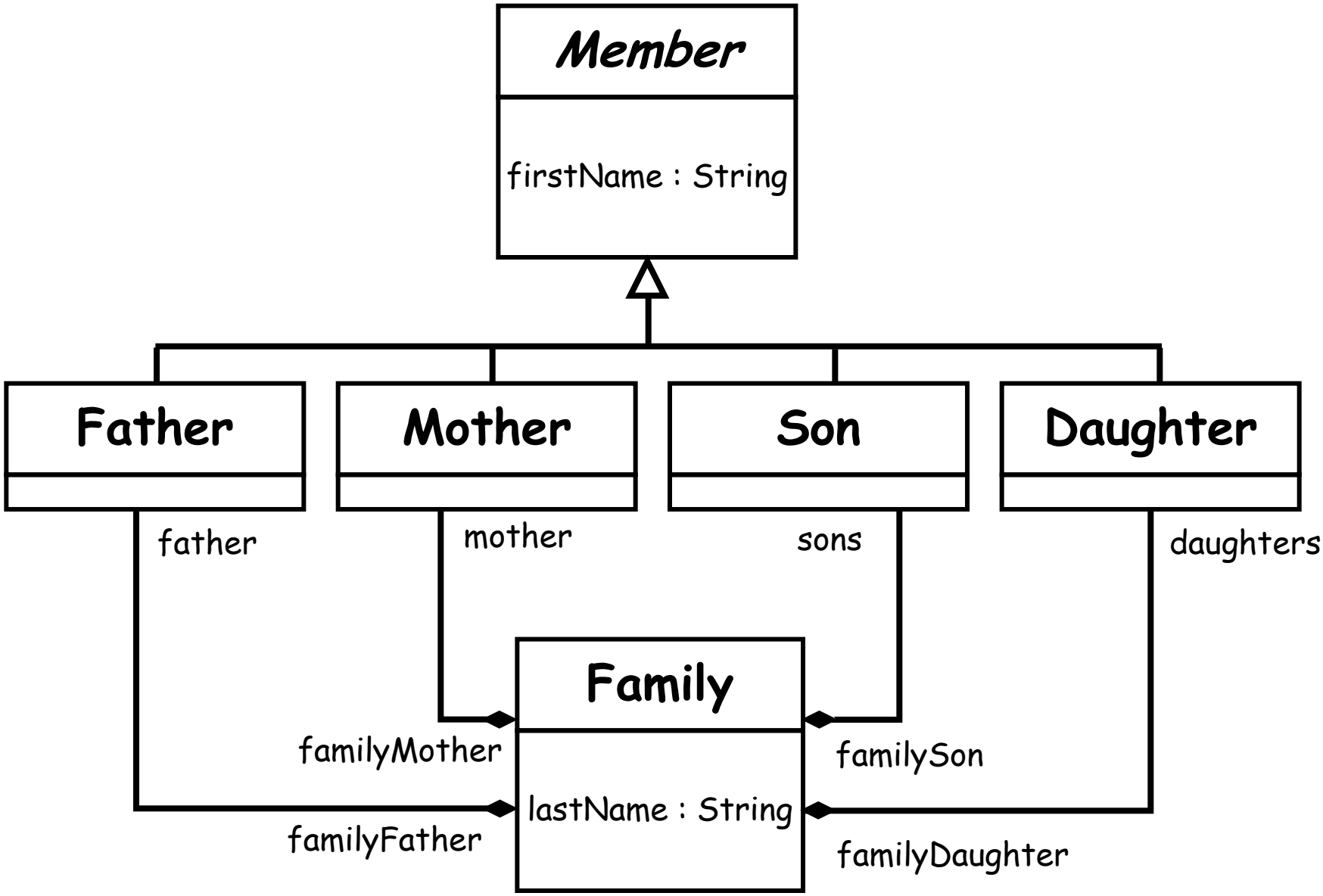
What is "Families":

- A collection of families.
- Each family has a lastName and is composed of members:
  - A father
  - A mother
  - Several sons
  - Several daughters
- Each family member has a first name.

Family: Simpson
- Father: Homer
- Mother: Marge
- Son: Bart
- Daughter: Lisa
- Daughter: Maggie

Family: Skywalker
- Father: Anakin
- Mother: Padme
- Son: Luke
- Daughter: Leia

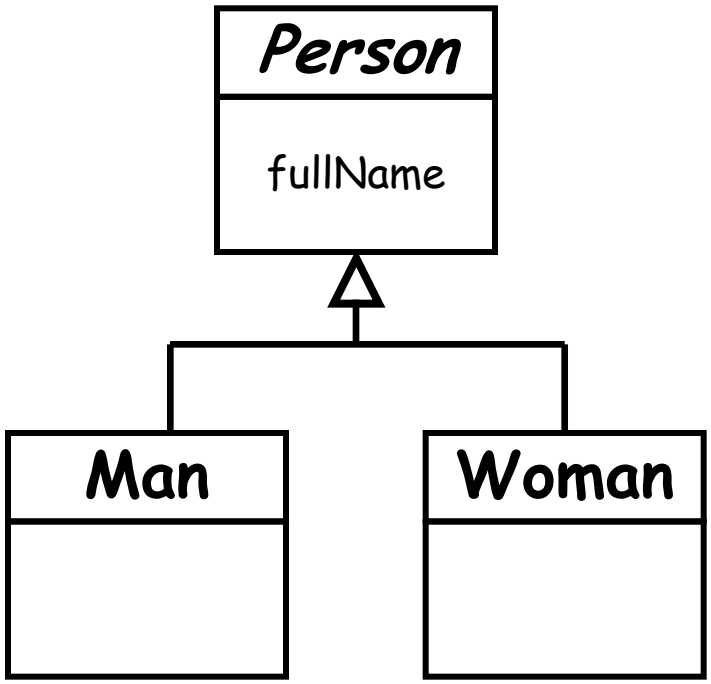
"Families" metamodel (visual presentation)



Demo

Writing a metamodel with KM3

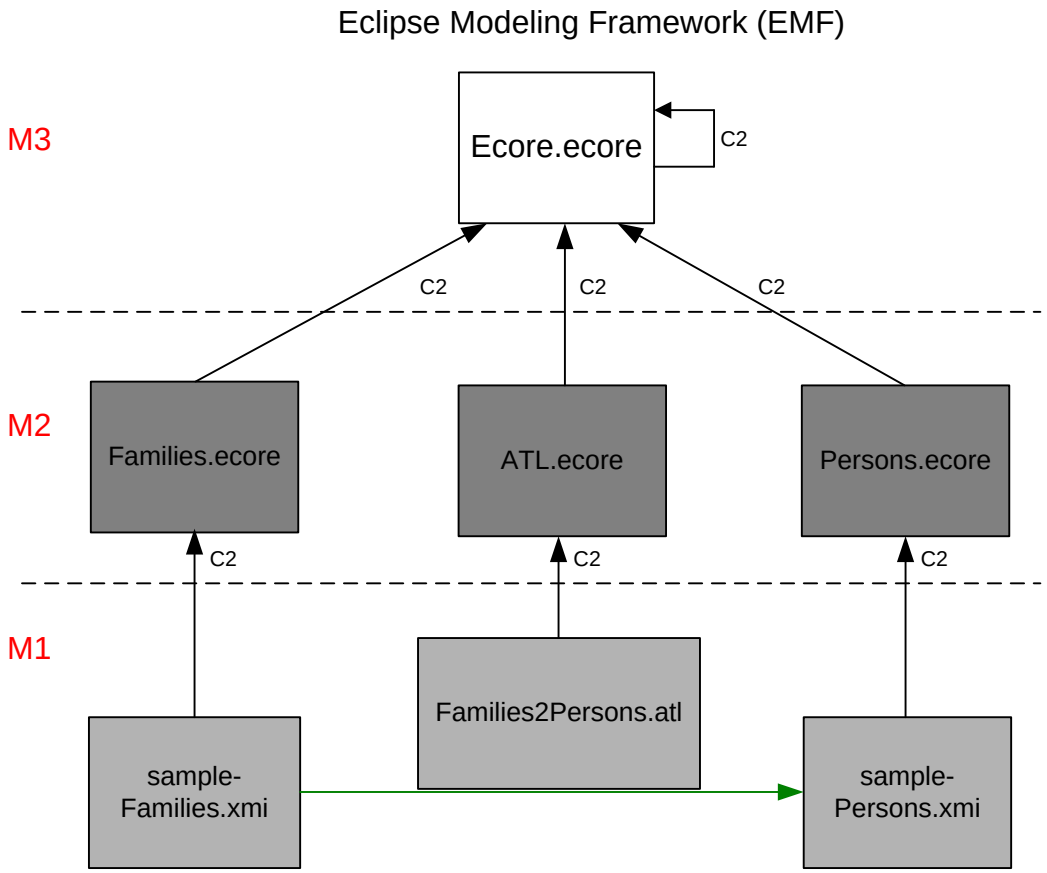
"Persons" metamodel (visual presentation)



Demo

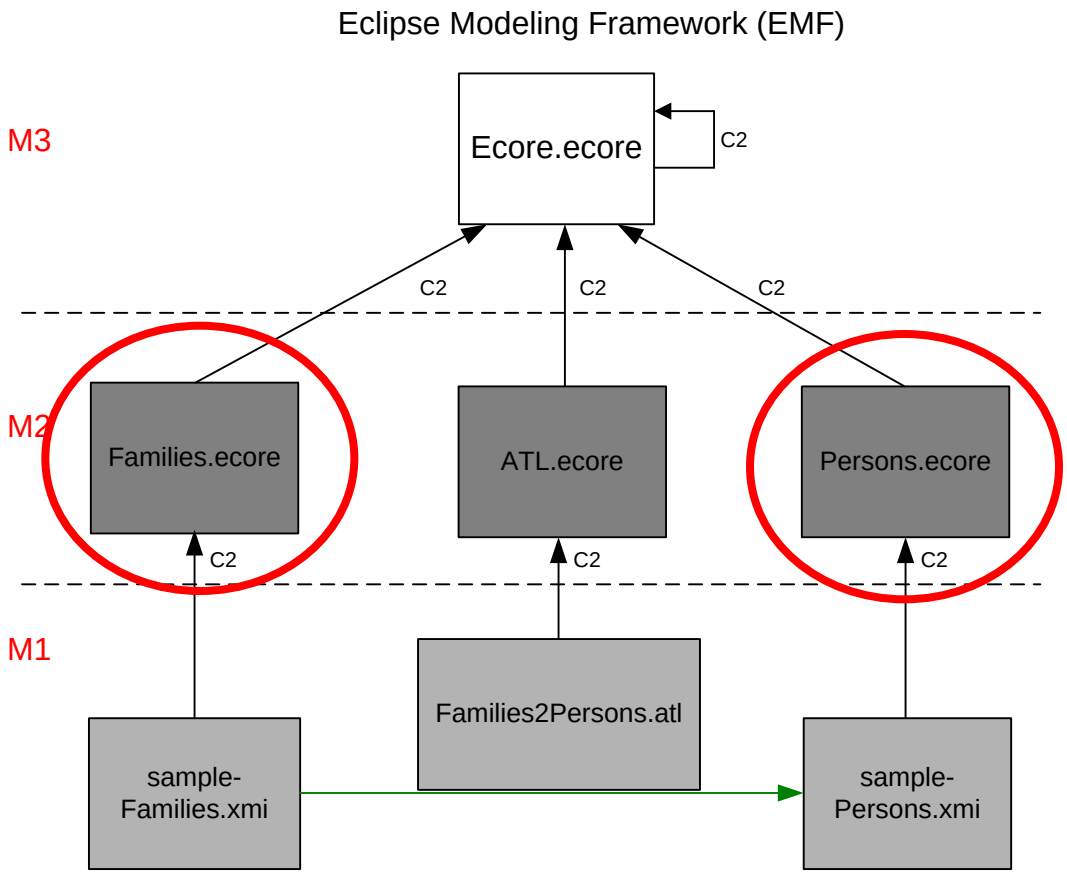
Writing a metamodel with KM3

The big picture



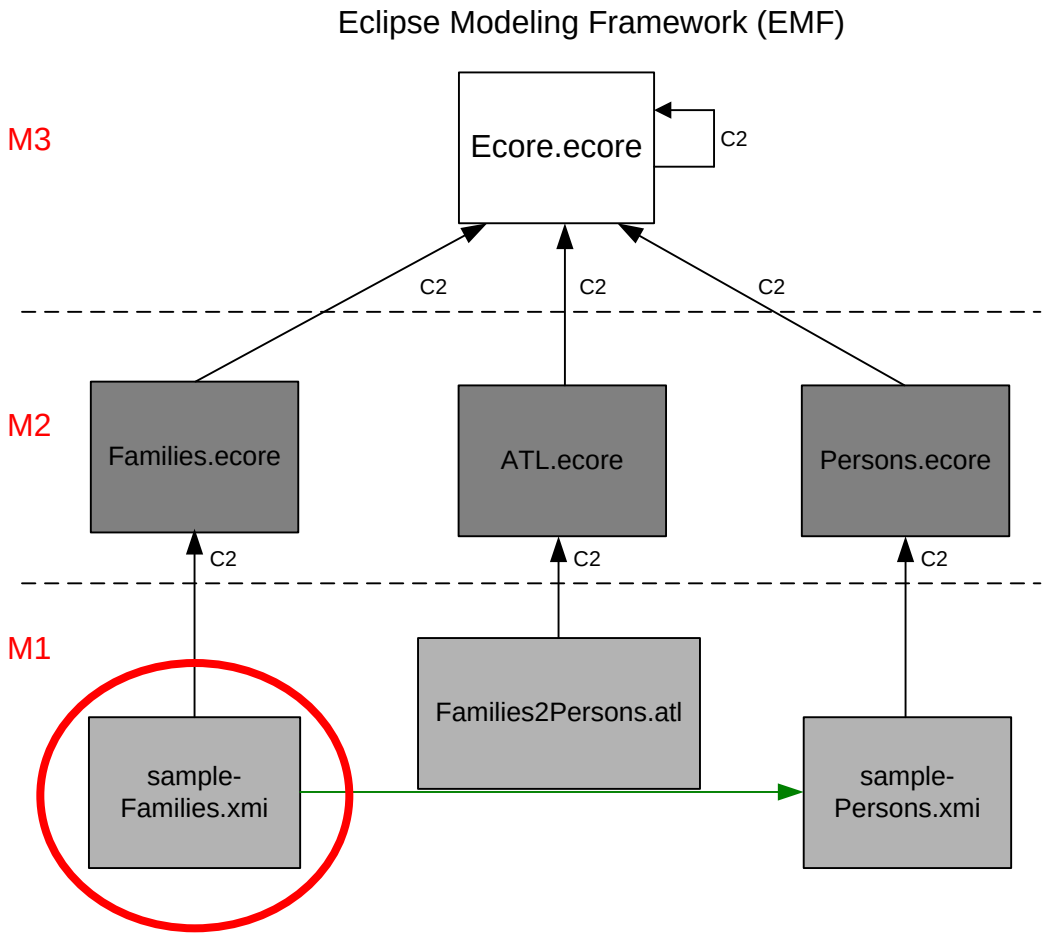
1. Our goal in this use case is to write the ATL transformation, stored in the "Families2Persons" file.
2. Prior to the execution of this transformation the resulting file "sample-Persons.xmi" does not exist. It is created by the transformation.
3. Before defining the transformation itself, we need to define the source and target metamodels ("Families.km3" and "Person.km3") and compile them to Ecore.
4. We take for granted that the definition of the ATL language is available (supposedly in the "ATL.ecore" file).
5. Similarly we take for granted that the environment provides the recursive definition of the metametamodel (supposedly in the "Ecore.ecore" file).

Families to Persons Architecture



1. *Families* and *Persons* metamodels have been created previously.
2. They have been written in the KM3 metamodel specification DSL (Domain Specific Language).
3. They have been compiled to Ecore

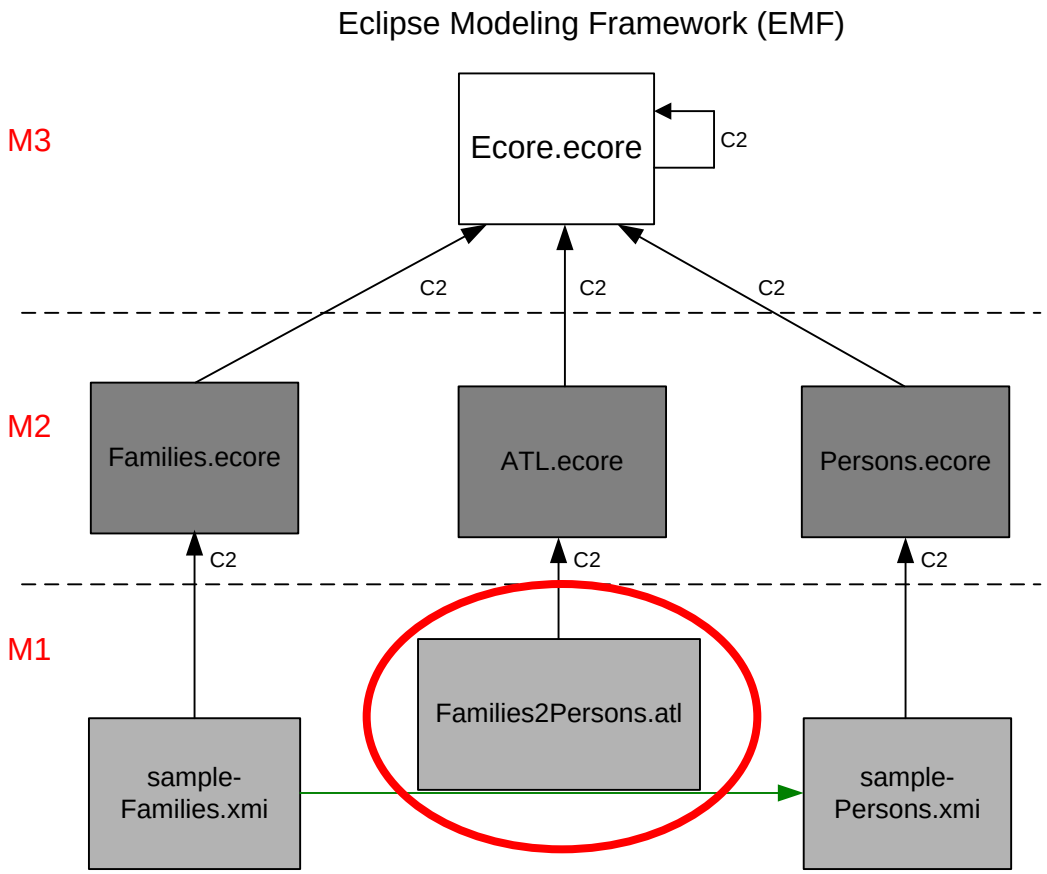
Families to Persons Architecture



1. The following file is the sample that we will use as source model in this use case:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<xmi:XMI xmi:version="2.0"
xmlns:xmi="http://www.omg.org/XMI"
xmlns="Families">
  <Family lastName="Skywalker">
    <father firstName="Anakin"/>
    <mother firstName="Padme"/>
    <sons firstName="Luke"/>
    <daughters firstName="Leia"/>
  </Family>
  <Family lastName="Simpsons">
    <father firstName="Homer"/>
    <mother firstName="Marge"/>
    <sons firstName="Bart"/>
    <daughters firstName="Lisa"/>
    <daughters firstName="Maggie"/>
  </Family>
</xmi:XMI>
```

Families to Persons Architecture



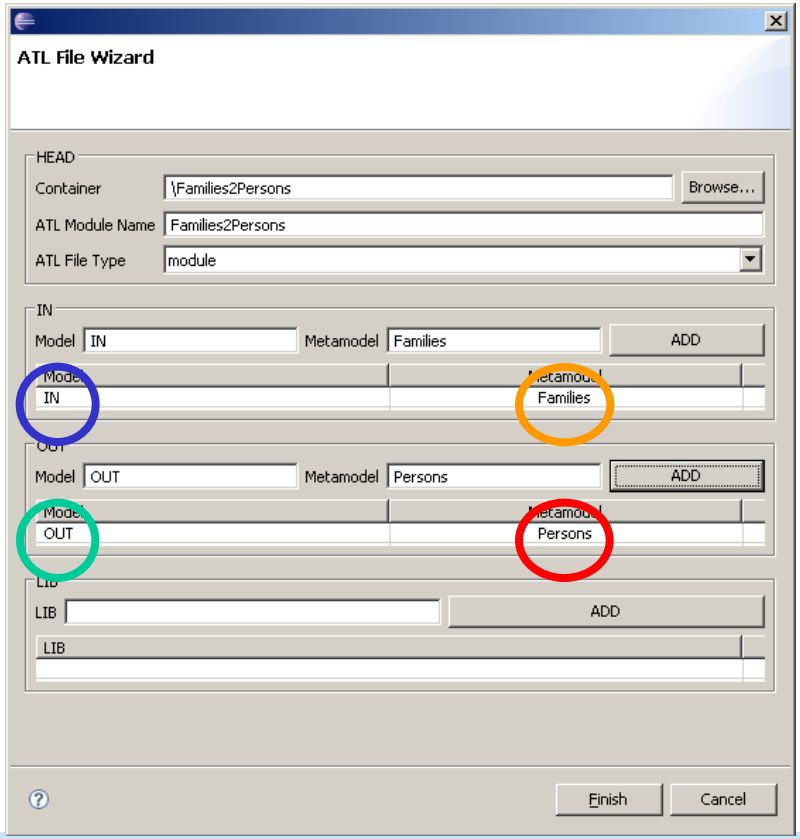
1. Now, let us start the creation of the ATL transformation *Families2Persons.atl*.
2. We suppose the ATL environment is already installed.

Families to Persons: ATL transformation creation

- Next we create the ATL transformation. To do this, we use the ATL File Wizard. This will generate automatically the header section.

IN:
Name of the source model in the transformation

OUT:
Name of the target model in the transformation



Families:
Name of the source metamodel in the transformation

Persons:
Name of the target metamodel in the transformation

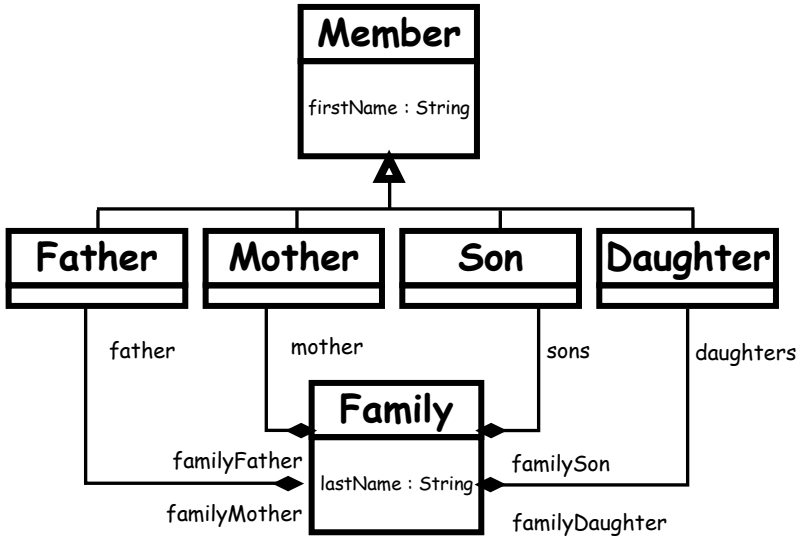
Families to Persons: header section

- The header section names the transformation module and names the variables corresponding to the source and target models ("IN" and "OUT") together with their metamodels ("Persons" and "Families") acting as types. The header section of "Families2Persons" is:

```
module Families2Persons;
create OUT : Persons from IN : Families;
```

Families to Persons: helper "isFemale()"

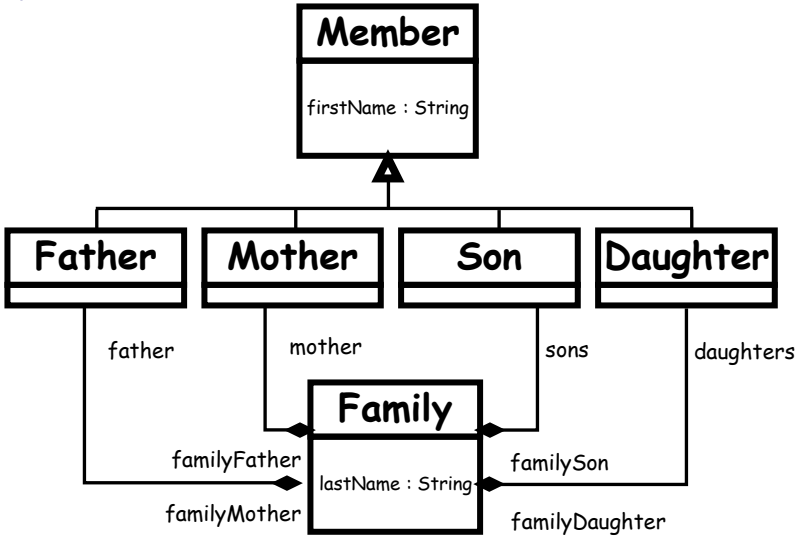
- A helper is an auxiliary function that computes a result needed in a rule.
- The following helper "isFemale()" computes the gender of the current member:




```
helper context Families!Member def: isFemale() : Boolean =
      if self.oclIsTypeOf(Families!Mother) or
            self.oclIsTypeOf(Families!Daughter) then
            true
      else
            false
      endif;
```

Families to Persons: helper "getName"

- The family name is not directly contained in class "Member". The following helper returns the family name by navigating the relation between "Family" and "Member":




```
helper context Families!Member def: getName() : String =
      if self.oclIsTypeOf(Families!Father) then
            'Mr. ' + self.firstName + ' ' + self.familyFather.lastName
      else
            if self.oclIsTypeOf(Families!Mother) then
                  'Mrs. ' + self.firstName + ' ' + self.familyMother.lastName
            else
                  if self.oclIsTypeOf(Families!Son) then
                        'Master ' + self.firstName + ' ' + self.familySon.lastName
                  else
                        'Miss ' + self.firstName + ' ' + self.familyDaughter.lastName
                  endif
            endif
      endif;
```

Families to Persons: writing the rules

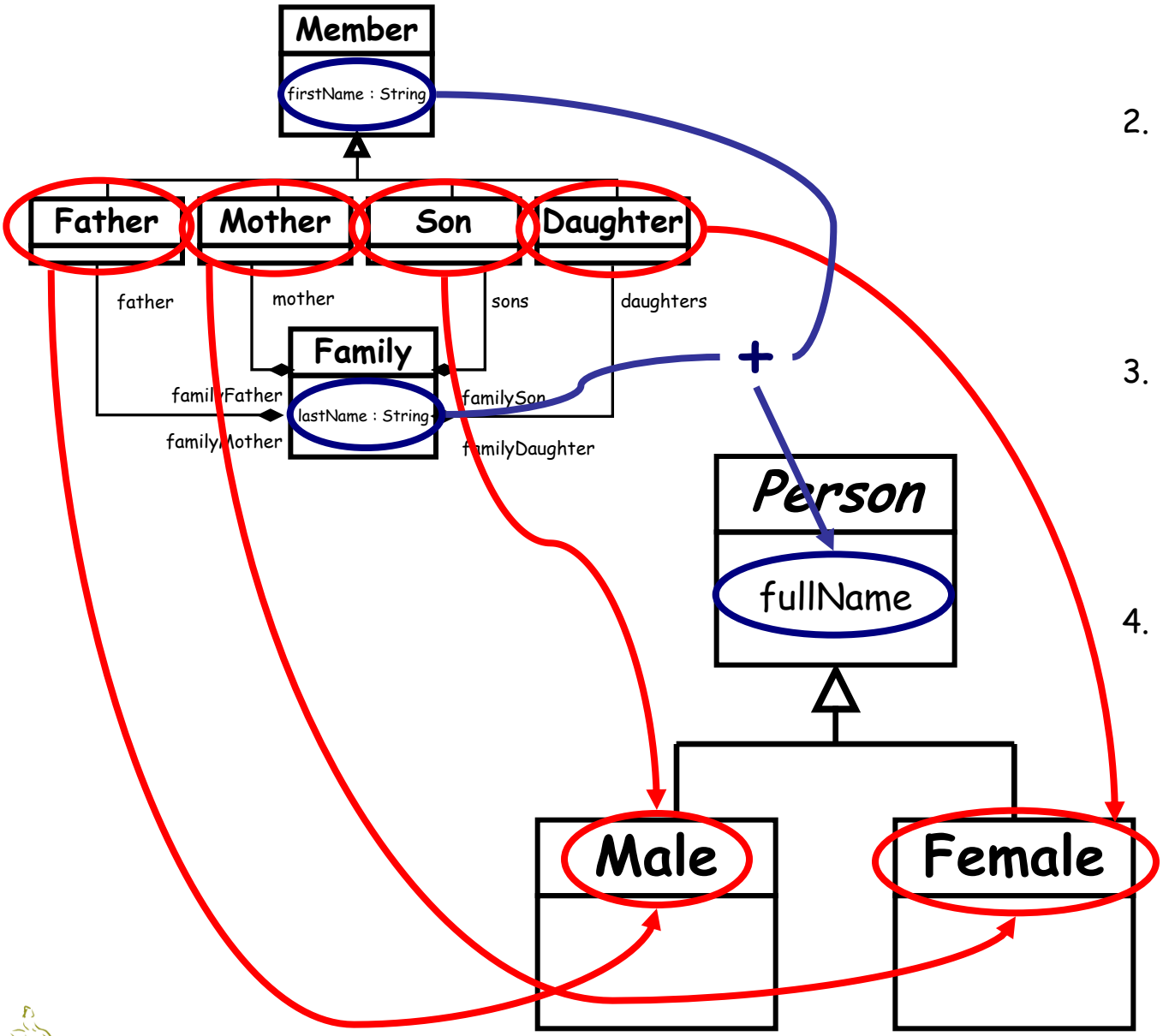
- After the helpers we now write the rules:
  - Member to Male

```
rule Member2Male {
      from
            m : Families!Member (not m.isFemale())
      to
            p : Persons!Male (
                  fullName <- m.getName()
            )
}
```

  - Member to Female

```
 rule Member2Female {
       from
             m : Families!Member (m.isFemale())
       to
             p : Persons!Male (
                   fullName <- m.getName()
             )
 }
```

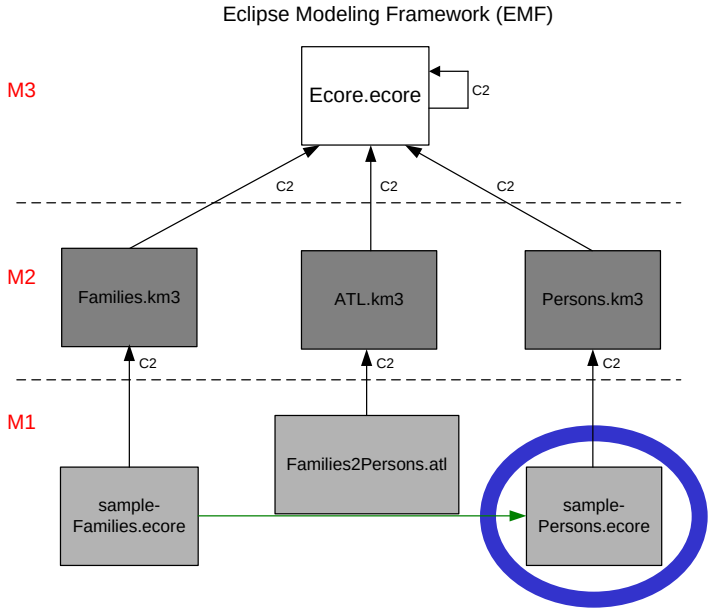
What do we want to do ?

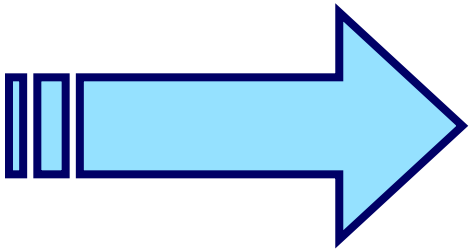


1. For each instance of the class "Member" in the IN model, create an instance in the OUT model.
2. If the original "Member" instance is a "father" or one of the "sons" of a given "Family", then we create an instance of the "Male" class in the OUT model.
3. If the original "Member" instance is a "mother" or one of the "daughters" of a given "Family", then we create an instance of the "Female" class in the OUT model.
4. In both cases, the "fullname" of the created instance is the concatenation of the Member "firstName" and of the Family "lastName", separated by a blank.

Families to Persons Architecture

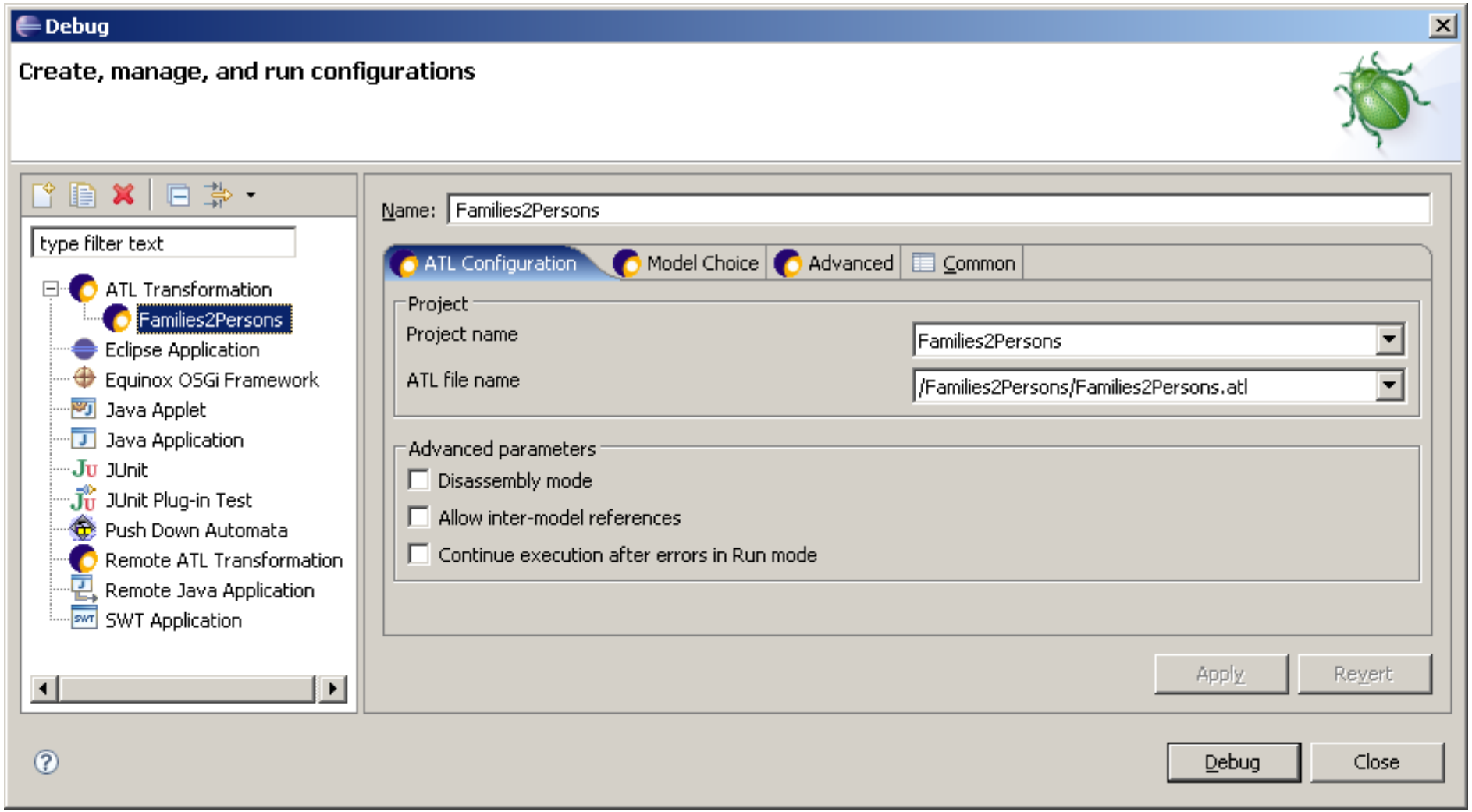
1. Once the ATL transformation "Families2Persons" is created, we can execute it to build the OUT model.





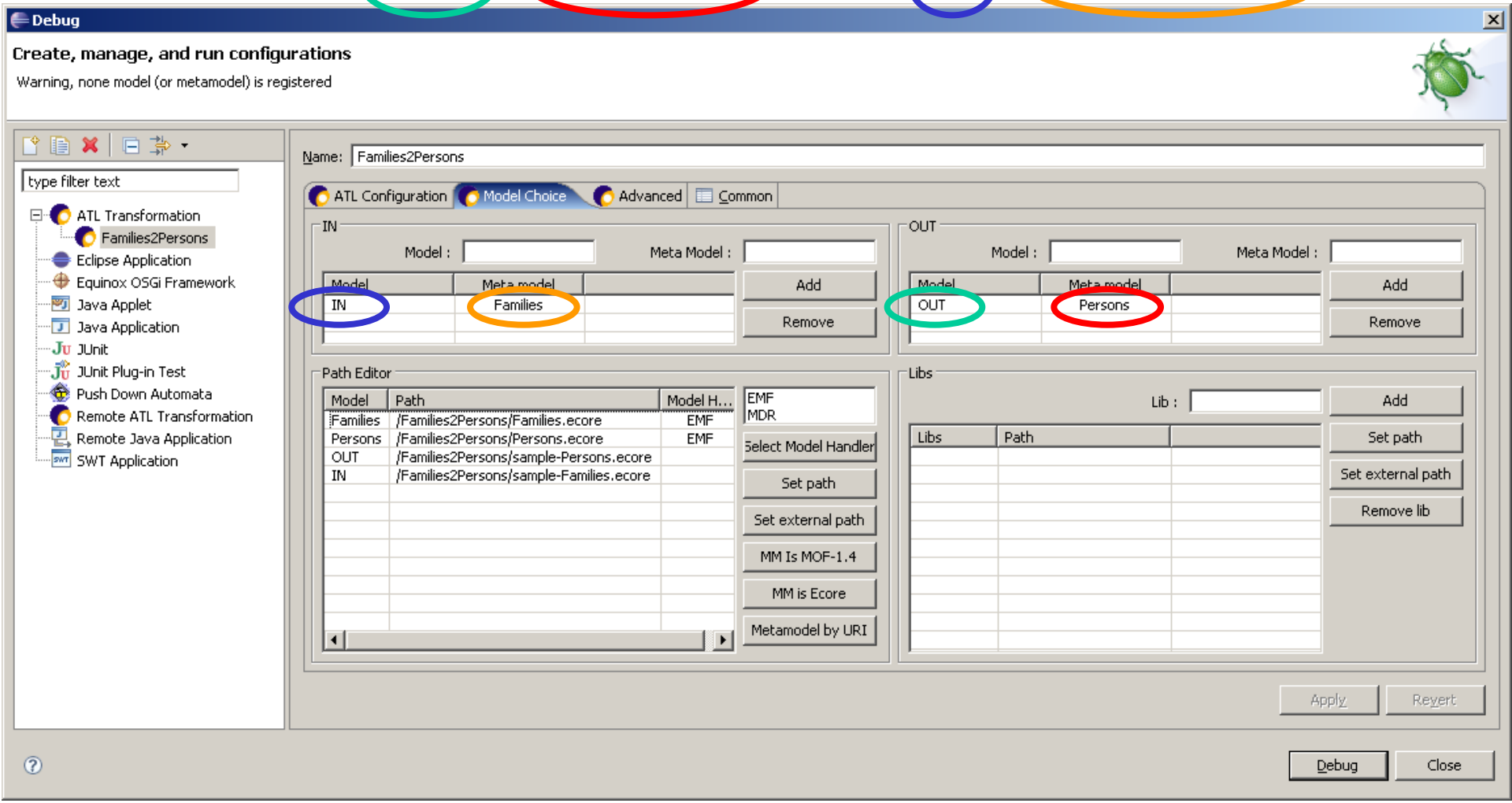
```
<?xml version="1.0" encoding="ISO-8859-1"?>
<xmi:XMI xmi:version="2.0"
    xmlns:xmi="http://www.omg.org/XMI"
xmlns="Persons">
  <Male fullName="Dylan Sailor"/>
  <Male fullName="Peter Sailor"/>
  <Male fullName="Brandon March"/>
  <Male fullName="Jim March"/>
  <Male fullName="David Sailor"/>
  <Female fullName="Jackie Sailor"/>
  <Female fullName="Brenda March"/>
  <Female fullName="Cindy March"/>
  <Female fullName="Kelly Sailor"/>
</xmi:XMI>
```

ATL Launch Configuration - 1



ATL Launch Configuration - 2

```
module Families2Persons;
create OUT : Persons from IN : Families;
```



Demo

Writing and launching your very first ATL transformation

Contents

Rule Inheritance

- Definition:
  - A matched rule R2 inherits from another matched rule R1:
    - R1 is the parent rule
    - R2 is the child rule, or subrule
  - A parent rule may be abstract (i.e. cannot be applied directly)
- Syntax:

```
abstract rule R1 {
  -- ...
}
rule R2 extends R1 {
  -- ...
}
```

- When to use:
  - When several rules share a common part

➔ code reuse

  - To specify polymorphic rules

➔ the parent rule specifies source and target element names and types of all its children

Rule Inheritance

- Usage rules:
  - A child rule matches a subset of what its parent rule matches:
    - The source pattern must have the same number of elements,
    - Each child source element must correspond to a unique parent source element,
    - Each child element type must conform to type of corresponding parent element (i.e. be the same or a subtype),
    - Guards are anded.
  - A child rule specializes target elements of its parent rule:
    - Target elements can be added in child rule,
    - Child target elements with corresponding parent elements (i.e. with the same variable name) can:
      - Have a different type (but must be a subtype of parent type),
      - Have more bindings,
      - Redefine bindings.
  - Only one child rule can match
    - There may be at most one *default* subrule (including parent rule if not asbtract): a rule without guard, which is matched by default.

Rule Inheritance

- Informal semantics:
  - Matching:
    1. Root rules (i.e. without parent) are matched,
    2. For each potential match, every subrule with a guard is tested,
    3. The one that matches, if any, is selected,
    4. If none matches, the *default** rule, if any, is selected,
    5. If selected rule is not a leaf (i.e. if it has subrules) then goto 2
    6. Target elements are created by using the most specific types.
  - Applying
    - Most specific bindings are used to initialize target elements (redefined bindings are not executed)

* *default* rule: subrule without guard or parent rule if not abstract

Demo

Using ATL Rule Inheritance

What we have learnt

- Writing metamodels with KM3 and compiling them to Ecore
- Writing a first ATL transformation
- Using rule inheritance
- Launching an ATL transformation

Contents

Use Case 2

Class to Relational

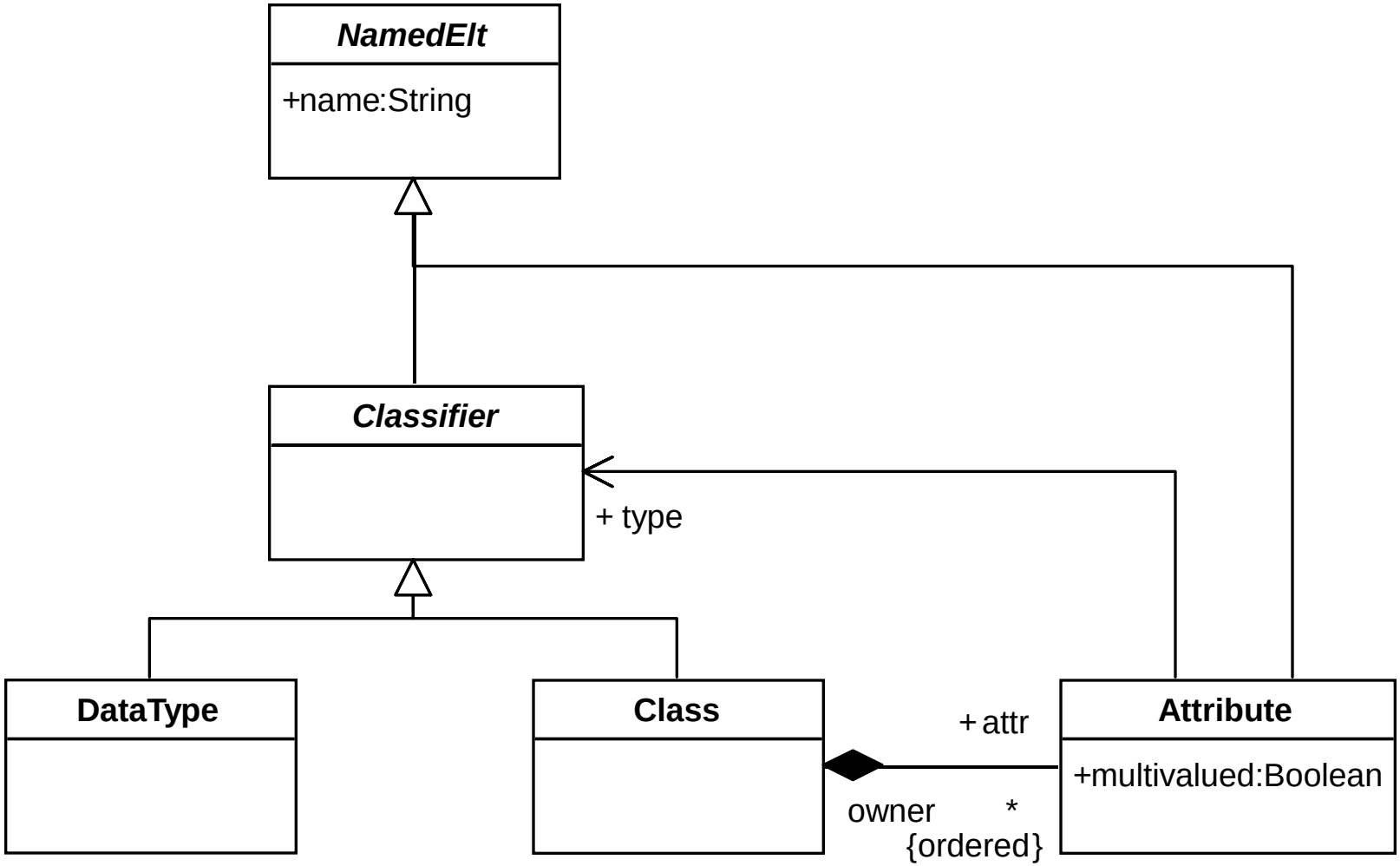
Families to Persons: Overview

- The source metamodel *Class* is a simplification of class diagrams.
- The target metamodel *Relational* is a simplification of the relational model.

➔ ATL declaration of the transformation:

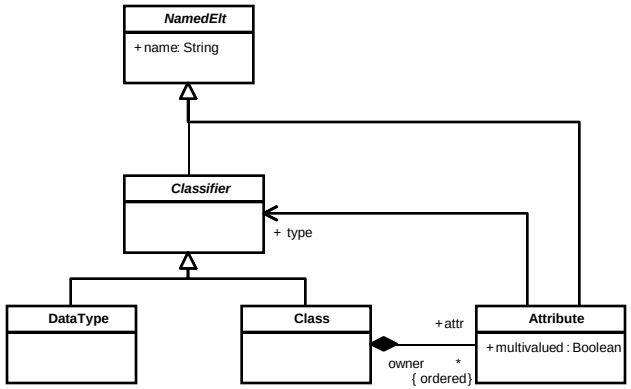
```
module Class2Relational;
create OUT : Relational from IN : Class;
```

"Class" metamodel



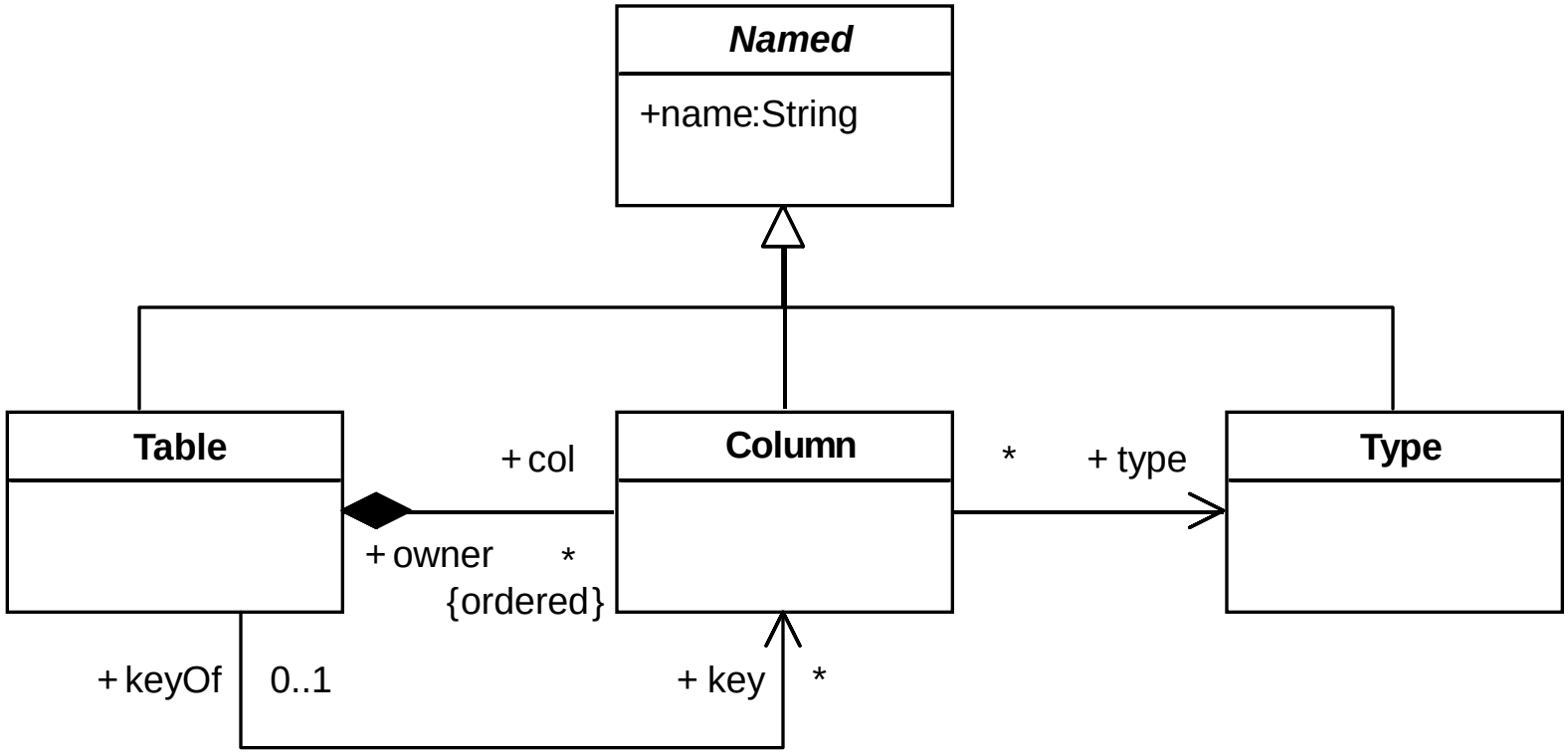
The Class Metamodel in KM3*

```
package Class {

    abstract class NamedElt {
        attribute name : String;
    }

    abstract class Classifier extends NamedElt {}

    class DataType extends Classifier {}

    class Class extends Classifier {
        reference attr[*] ordered container : Attribute oppositeOf owner;
    }

    class Attribute extends NamedElt {
        attribute multiValued : Boolean;
        reference type : Classifier;
        reference owner : Class oppositeOf attr;
    }
}
```

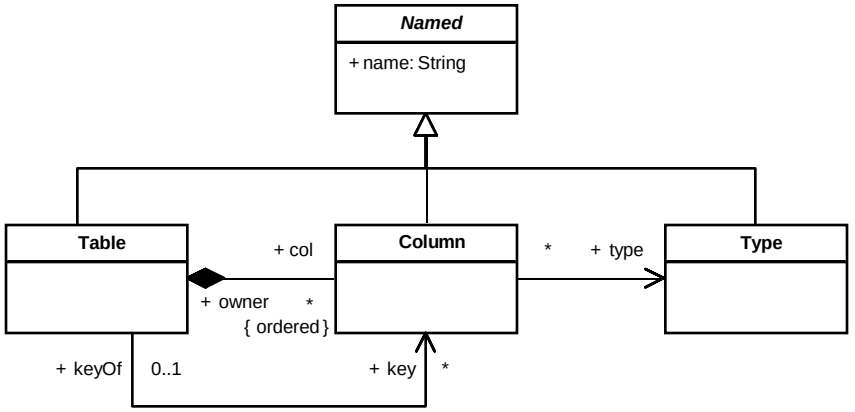


*For more information on KM3 see http://www.eclipse.org/gmt/am3/zoos/atlanticZoo/

"Relational" metamodel



The Relational Metamodel in KM3

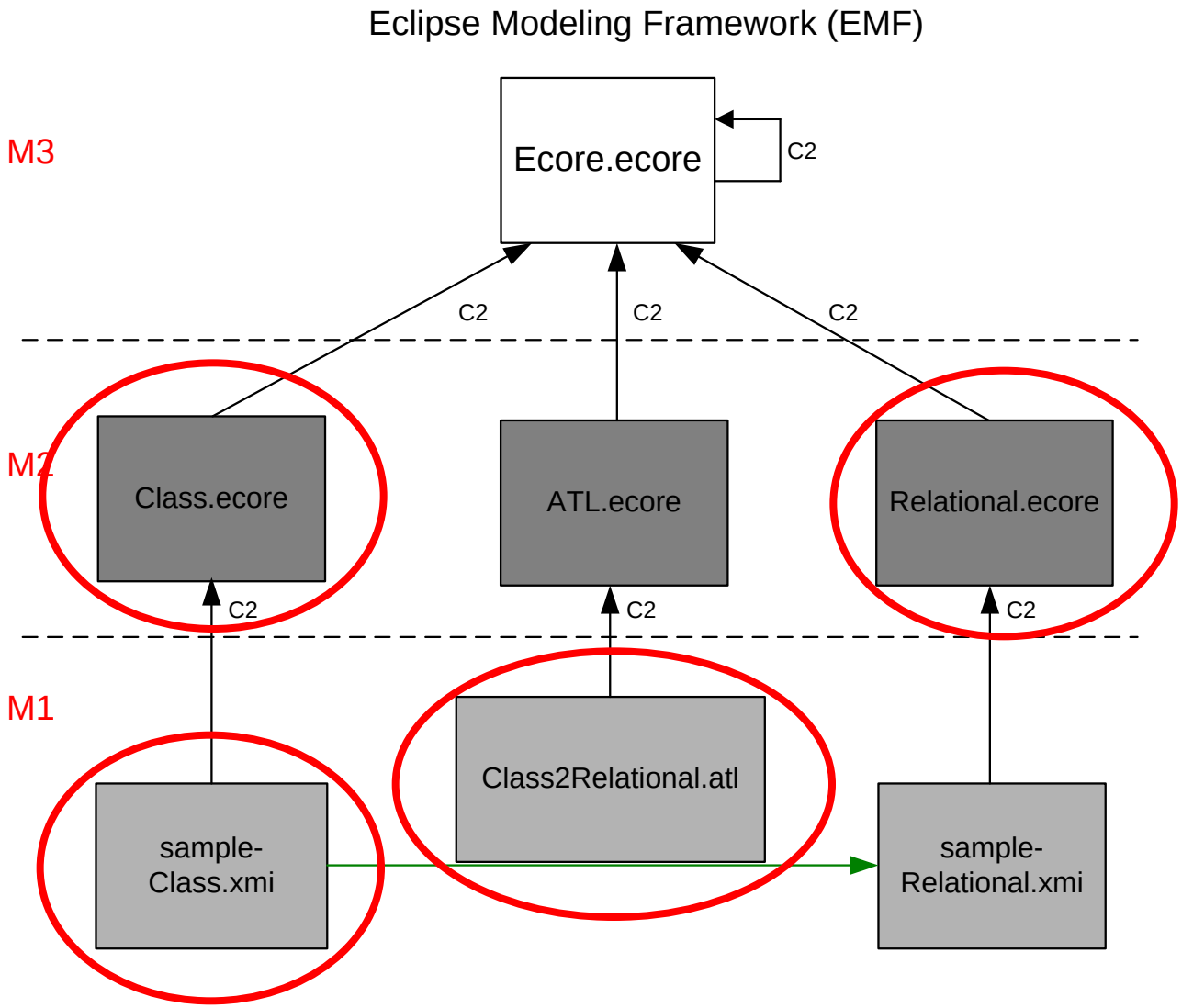



```
package Relational {

    abstract class Named {
        attribute name : String;
    }

    class Table extends Named {
        reference col[*] ordered container : Column oppositeOf owner;
        reference key[*] : Column oppositeOf keyOf;
    }

    class Column extends Named {
        reference owner : Table oppositeOf col;
        reference keyOf[0-1] : Table oppositeOf key;
        reference type : Type;
    }

    class Type extends Named {}
}
```

Demo

"Class" and "Relational" metamodel in KM3

The big picture



Rules specifications

- Class2Table: a Table is created for each Class
  - The name of the Table is the name of the Class
  - The columns of the table correspond to the single-valued attributes of the class
  - Each Table owns a key containing a unique identifier
- SingleValuedAttribute2Column: a Column is created for each single-valued Attribute
- MultiValuedAttribute2Column: a Table is created for each multi-valued Attribute, which contains two columns:
  - The identifier of the table created from the class owner of the Attribute
  - The value.

Example: Class to Relational, rule Class2Table

- A Table is created for each Class:

```
rule Class2Table {
   from                    -- source pattern
    c : Class!Class
   to                      -- target pattern
    t : Relational!Table
}
```

Example: Class to Relational, rule Class2Table

- The name of the Table is the name of the Class:

```
rule Class2Table {
   from
    c : Class!Class
   to
    t : Relational!Table (
        name <- c.name -- a simple binding
    )
}
```

Example: Class to Relational, rule Class2Table

- The columns of the table correspond to the single-valued attributes of the class:

```
rule Class2Table {
    from
     c : Class!Class
    to
     t : Relational!Table (
          name <- c.name,
          col <- c.attr->select(e |          -- a binding
                   not e.multiValued  -- using                    )
      -- complex navigation
     )
}
```

- Remark: attributes are automatically resolved into columns by automatic traceability support.

Example: Class to Relational, rule Class2Table

- Each Table owns a key containing a unique identifier:

```
rule Class2Table {
 from
      c : Class!Class
 to
      t : Relational!Table (
           name <- c.name,
           col <- c.attr->select(e |
                     not e.multiValued
                )->union(Sequence {key}),
           key <- Set {key}
      ),
      key : Relational!Column (   -- another target
           name <- 'Id'           -- pattern element
      )                           -- for the key
}
```

Example: Class to Relational, rule SingleValuedAttribute2Column

- A Column is created for each single-valued Attribute:

```
rule SingleValuedAttribute2Column {
   from   -- the guard is used for selection
    a : Class!Attribute (not a.multiValued)
   to
    c : Relational!Column (
        name <- a.name
    )
}
```

Example: Class to Relational, rule MultiValuedAttribute2Column

- A Table is created for each multi-valued Attribute, which contains two columns:
  - The identifier of the table created from the class owner of the Attribute
  - The value.

```
rule MultiValuedAttribute2Column {
    from
        a : Class!Attribute (a.multiValued)
    to
        t : Relational!Table (
            name <- a.owner.name + '_' + a.name,
            col <- Sequence {id, value}
        ),
        id : Relational!Column (
            name <- 'Id'
        ),
        value : Relational!Column (
            name <- a.name
        )
}
```

Demo

"Class2Relational"
ATL transformation

What we have learnt

- Using automatic traceability support in ATL
- Writing simple OCL for navigating models
- Writing helpers

ATL Resource page

- ATL Home page
  - http://www.eclipse.org/m2m/atl/
- ATL Documentation page
  - http://www.eclipse.org/m2m/atl/doc/
- ATL Newsgroup
  - news://news.eclipse.org/eclipse.modeling.m2m
- ATL Wiki
  - http://wiki.eclipse.org/index.php/ATL

End of part 1: Model Transformation with ATL

- Thanks
  - Questions?
  - Comments?

Summary

- We have presented here a "hello world" level basic ATL transformation as well as a more elaborated one.
- This is not a recommendation on how to program in ATL, just an initial example.
- Several questions have not been answered
  - Like how to transform a text into an XMI-encoded model.
  - Or how to transform the XMI-encoded result into text.
- For any further questions, see the documentation mentioned in the resource page (FAQ, Manual, Examples, etc.).

ATL Resource page

- ATL Home page
  - http://www.eclipse.org/m2m/atl/
- ATL Documentation page
  - http://www.eclipse.org/m2m/atl/doc/
- ATL Newsgroup
  - news://news.eclipse.org/eclipse.modeling.m2m
- ATL Wiki
  - http://wiki.eclipse.org/index.php/ATL