The Meta-Object Facility (MOF)

Richard Paige
University of York, UK

July 2006

Context of this work

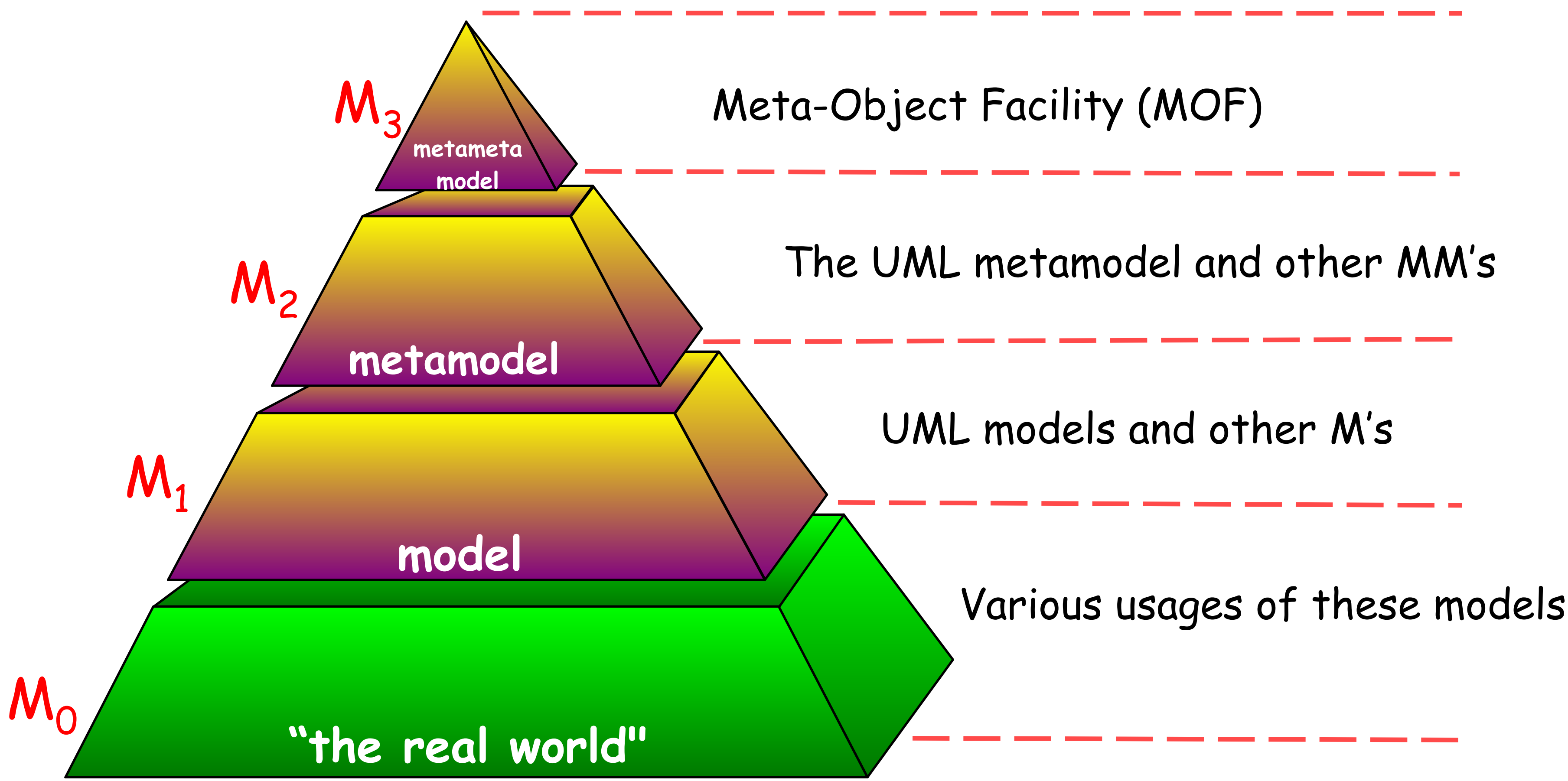
- The present courseware has been elaborated in the context of the MODELWARE European IST FP6 project (http://www.modelware-ist.org/).
- Co-funded by the European Commission, the MODELWARE project involves 19 partners from 8 European countries. MODELWARE aims to improve software productivity by capitalizing on techniques known as Model-Driven Development (MDD).
- To achieve the goal of large-scale adoption of these MDD techniques, MODELWARE promotes the idea of a collaborative development of courseware dedicated to this domain.
- The MDD courseware provided here with the status of open source software is produced under the EPL 1.0 license.

Intended Audience

- Have some experience with Model-Driven Development.
- Are aware of, but may not be familiar with, the relevant OMG/MDD standards.
- Are interested in learning more about language development and implementation.

Refresher

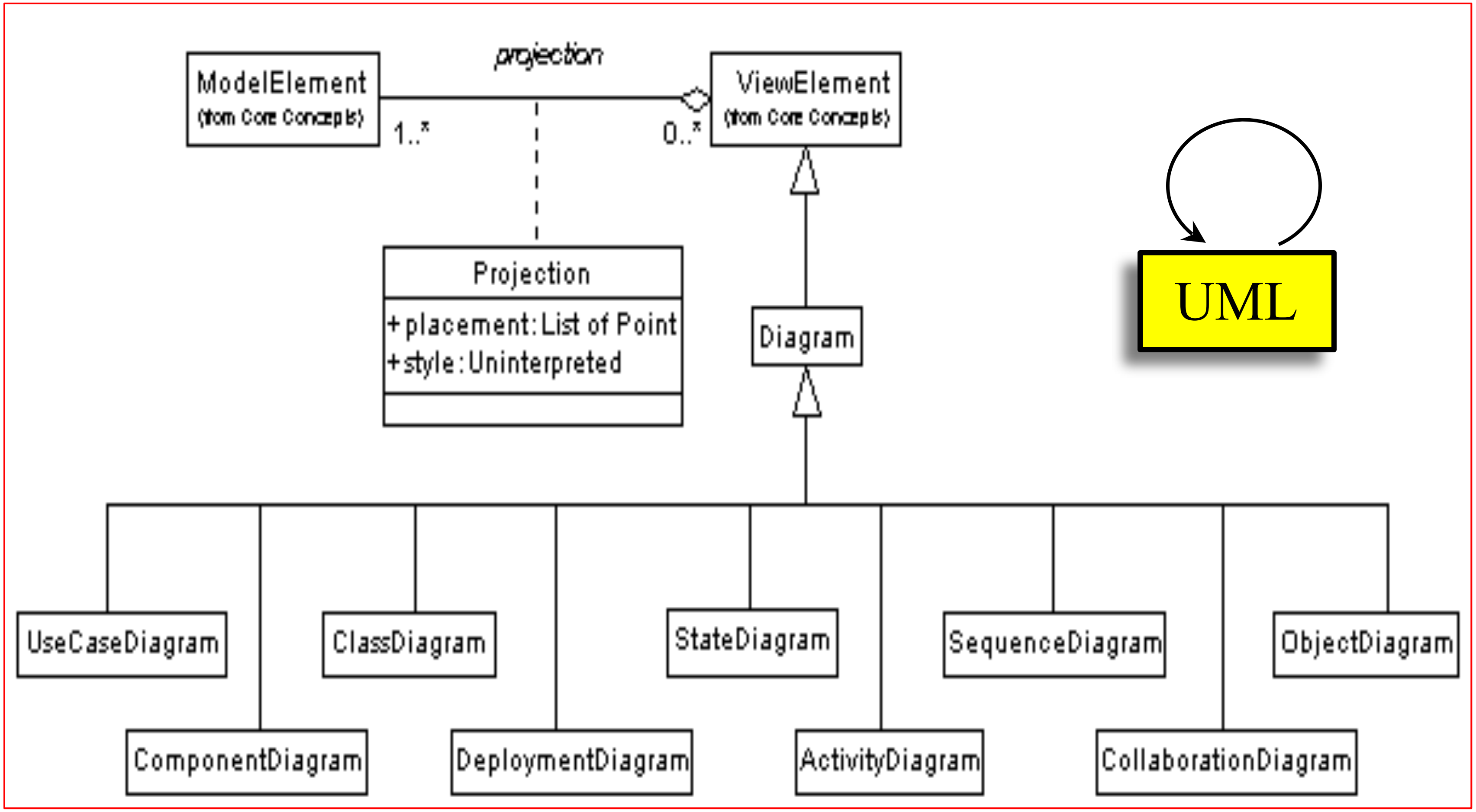
- Recall the OMG metamodel architecture.



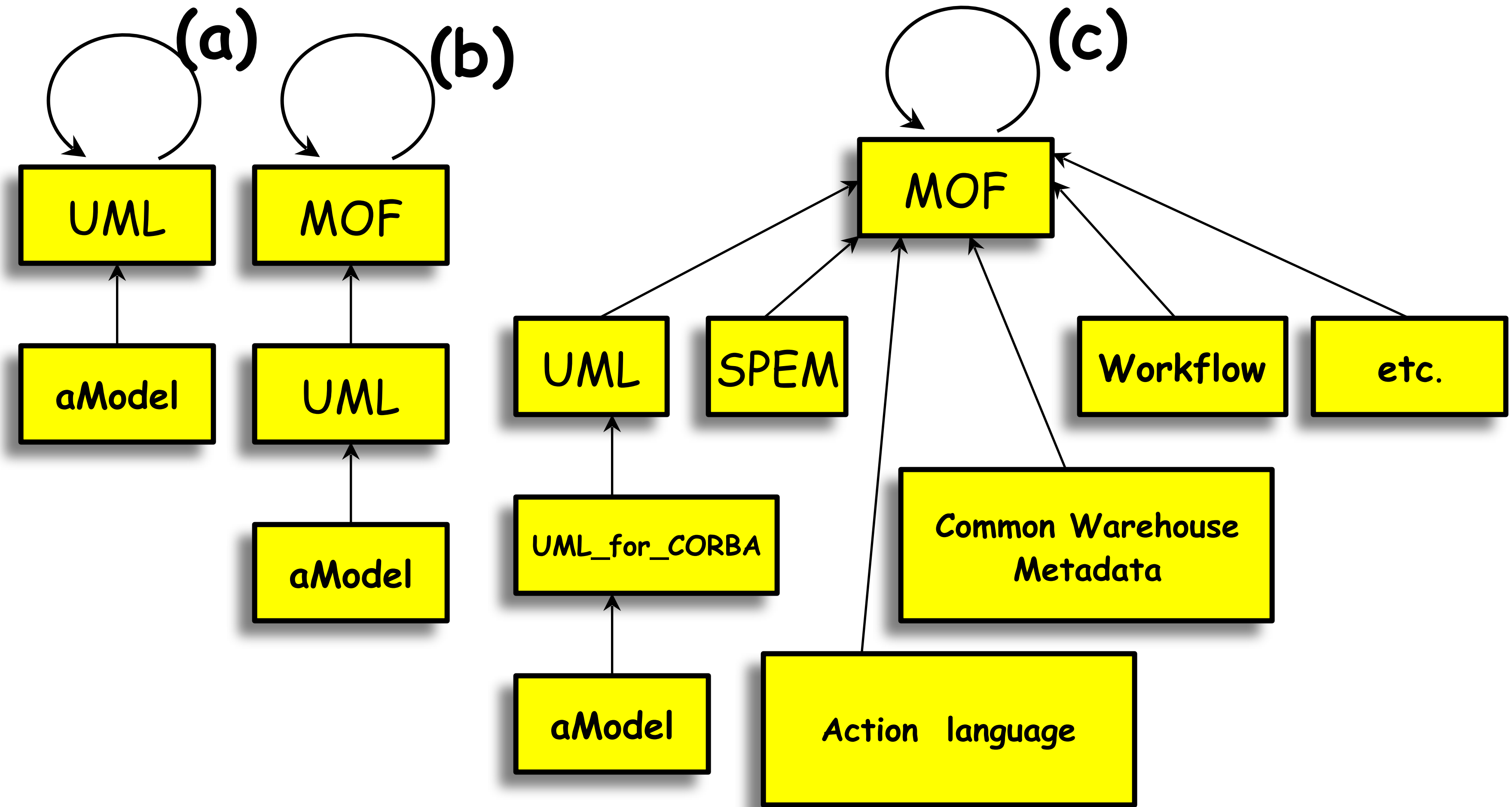
MOF

- MOF = Meta-Object Facility
- A metadata management framework.
- A language to be used for defining languages.
  - i.e., it is an OMG-standard metamodelling language.
  - The UML metamodel is defined in MOF.
- MOF 2.0 shares a common core with UML 2.0.
  - Simpler rules for modelling metadata.
  - Easier to map from/to MOF.
  - Broader tool support for metamodelling (i.e., any UML 2.0 tool can be used).
- How has MOF come to be?

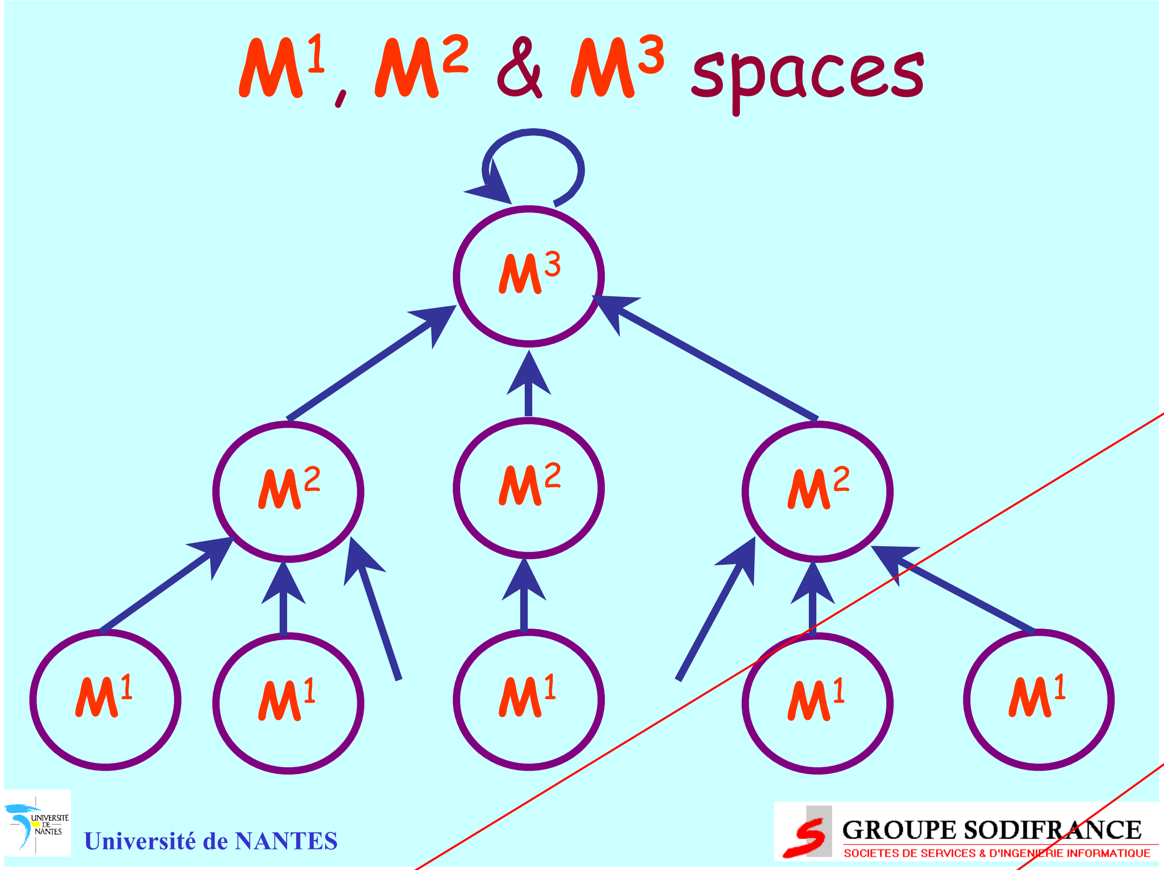
Fragments of a UML metamodel

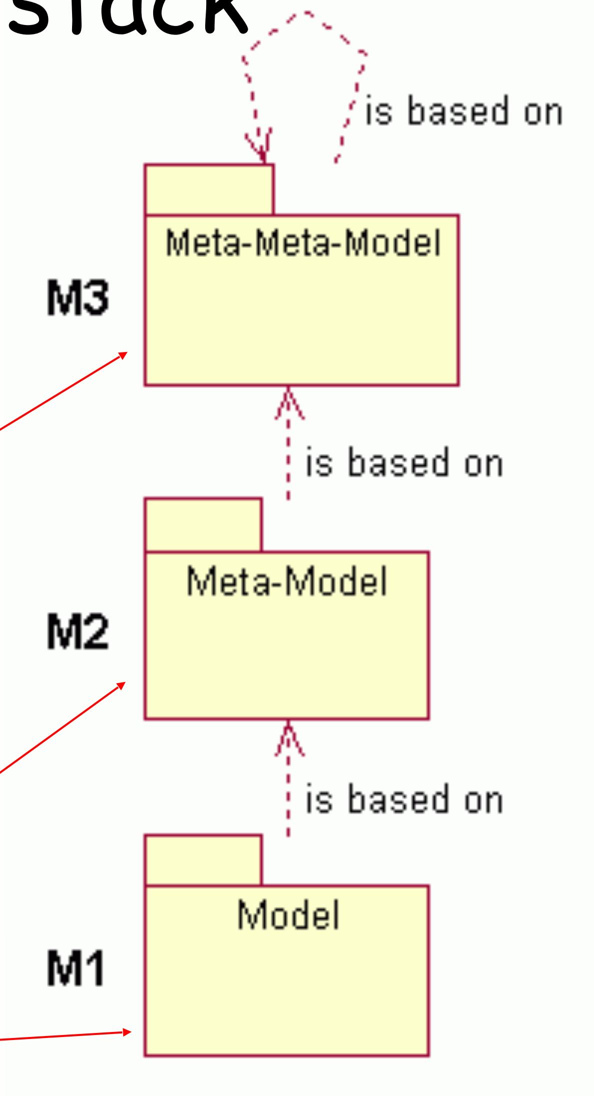


Stages in the Evolution of Languages at the OMG.



The MDA meta-model stack



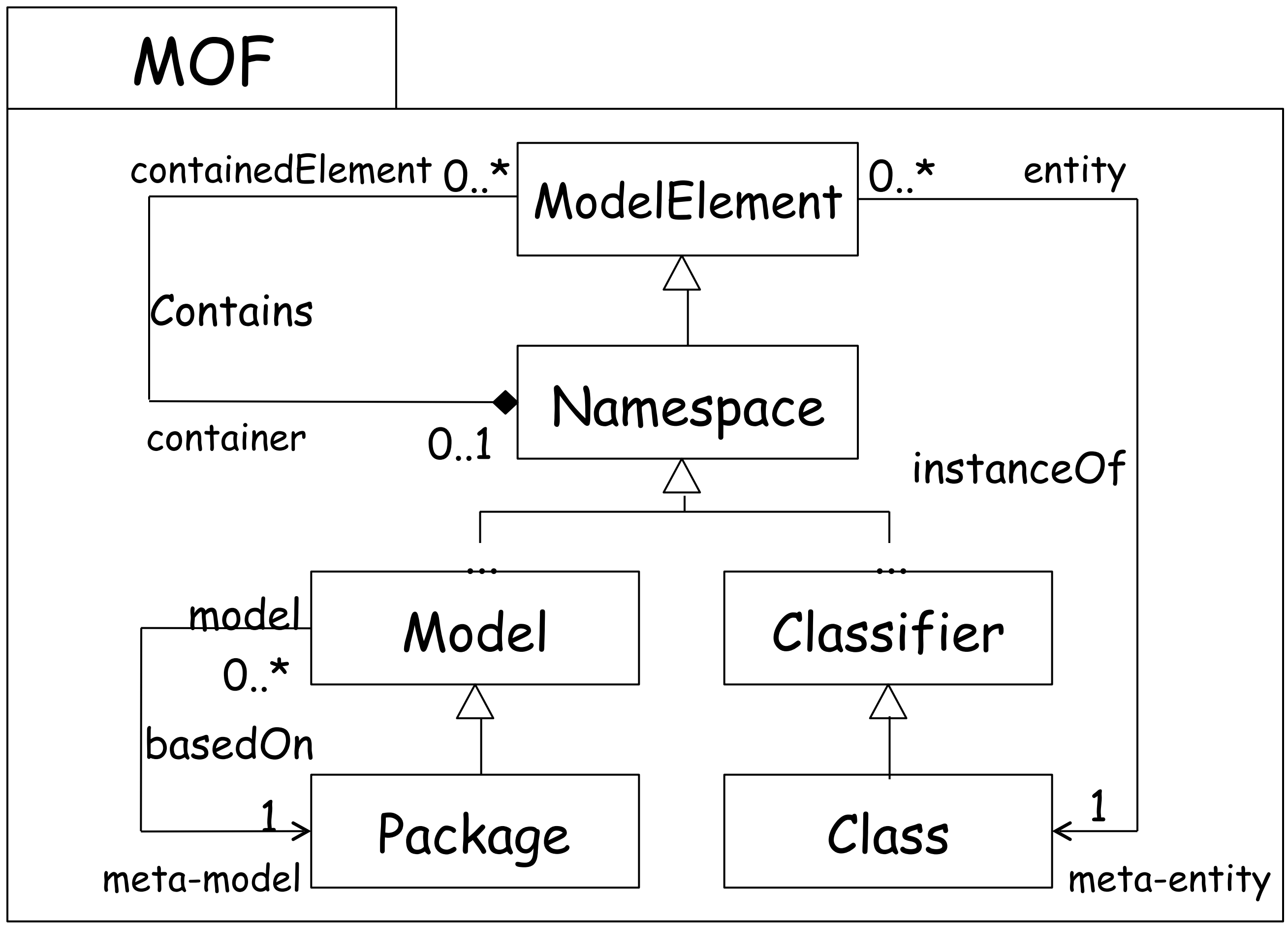


- One unique Meta-Meta-model (the MOF)
- An important library of compatible Meta-models
- Each of the models
  is defined in the language of its unique meta-model

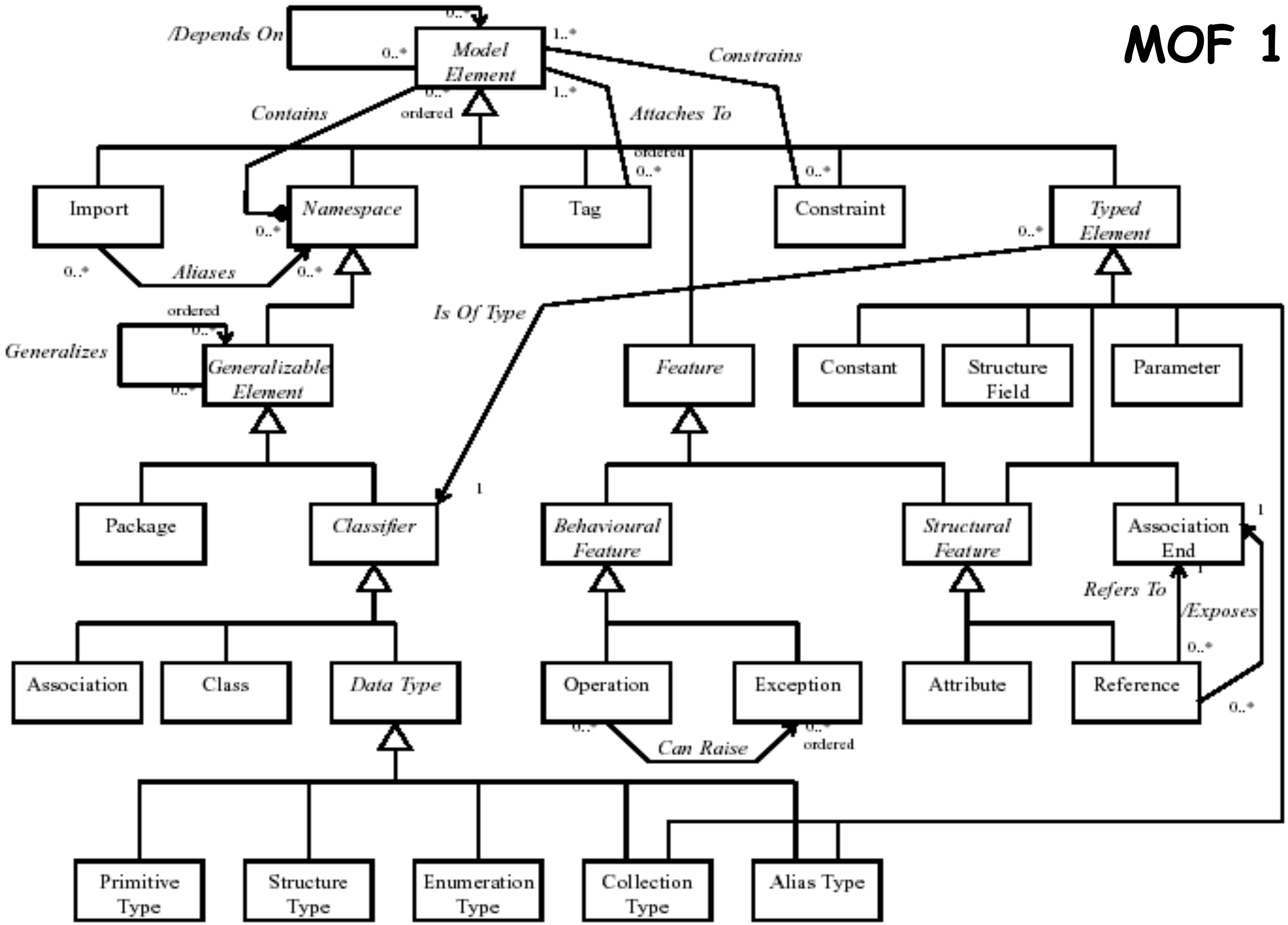
MOF Evolution

- MOF has evolved through several versions.
- MOF 1.x is the most widely supported by tools.
- MOF 2.0 is the current standard, and it has been substantially influenced by UML 2.0.
- MOF 2.0 is also critical in supporting transformations, e.g., QVT and Model-to-text.
- We will carefully clarify which version of MOF we are presenting.
  - Important lessons can be learned by considering each version.

Principal Diagram - MOF 1.x



MOF 1.x



/Depends On
0..*
0..*
Model Element
1..*
Constrains
Contains
0..*
ordered
1..*
Attaches To
ordered
0..*
0..*
Import
Namespace
Tag
Constraint
Typed Element
0..*
0..*
0..*
Aliases
0..*
ordered
0..*
Is Of Type
Generalizes
Generalizable Element
0..*
Feature
Constant
Structure Field
Parameter
1
Package
Classifier
Behavioural Feature
Structural Feature
Association End
1
1
Refers To
/Exposes
0..*
Association
Class
Data Type
Operation
Exception
Attribute
Reference
0..*
0..*
Can Raise
0..*
ordered
Primitive Type
Structure Type
Enumeration Type
Collection Type
Alias Type

MOF 1.x Key Abstract Classes

- **ModelElement** is the common base **Class** of all M3-level Classes.
  - Every **ModelElement** has a name
- **Namespace** is the base **Class** for all M3-level Classes that need to act as containers
- **GeneralizableElement** is the base **Class** for all M3-level Classes that support generalization (i.e., inheritance in OOP)
- **TypedElement** is the base **Class** for M3-level Classes such as **Attribute**, **Parameter**, and **Constant**
  - Their definition requires a type specification
- **Classifier** is the base **Class** for all M3-level Classes that (notionally) define types.
  - Examples of **Classifier** include **Class** and **DataType**

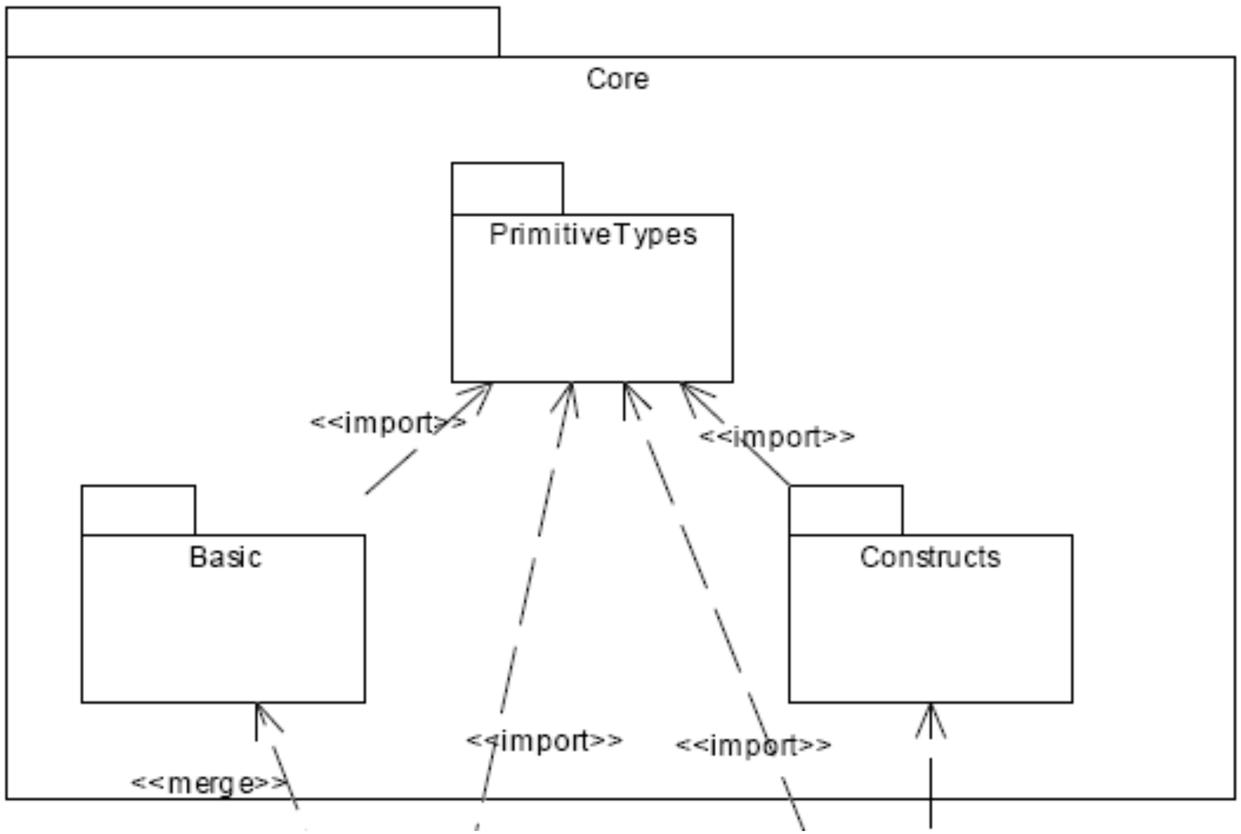
The MOF 1.x Model - Main Concrete Classes

- The key concrete classes (or meta-metaclasses) of MOF are as follows:
  - Class
  - Association
  - Exception (for defining abnormal behaviours)
  - Attribute
  - Constant
  - Constraint

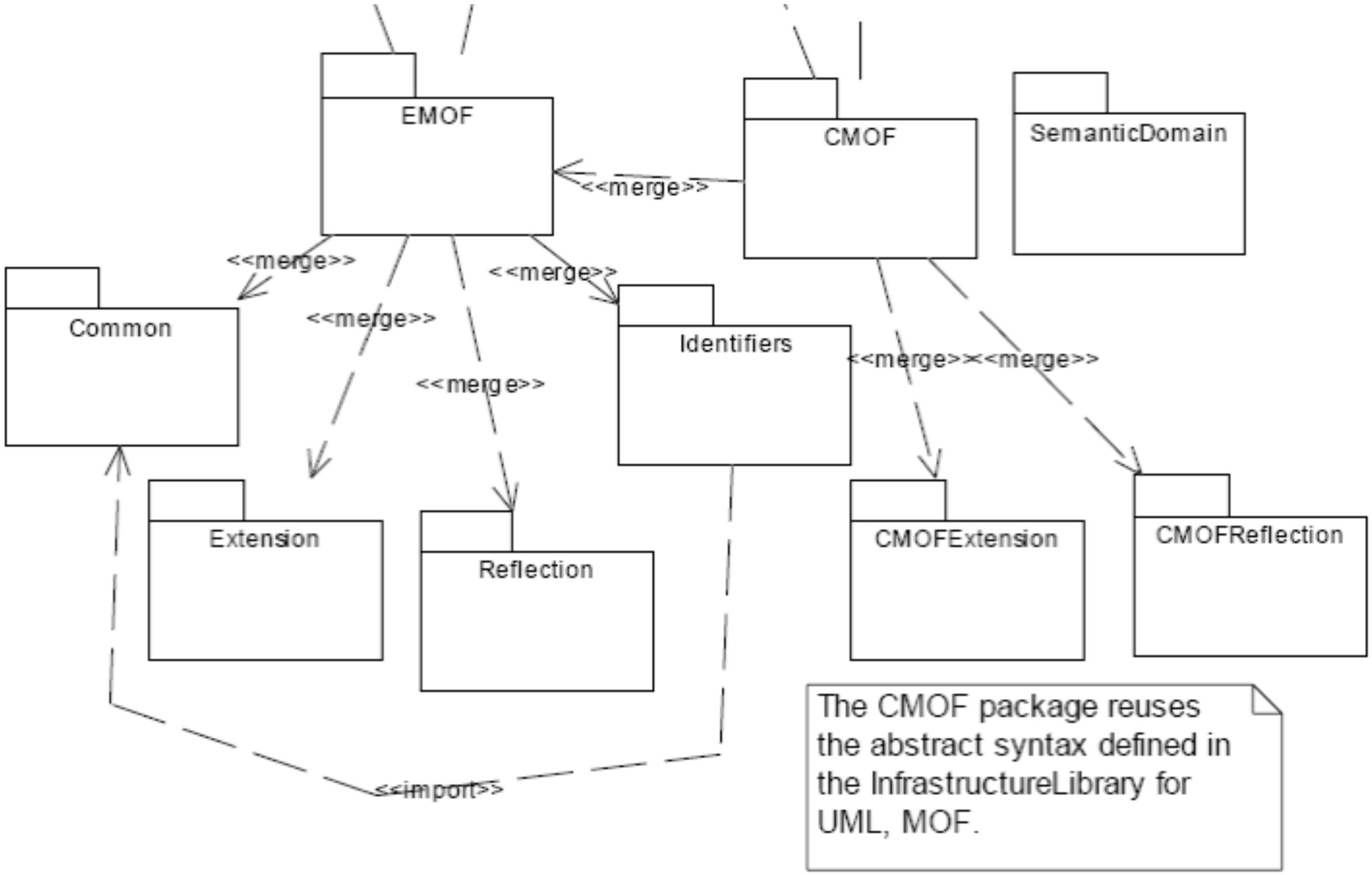
The MOF 1.x Model: Key associations

- **Contains:** relates a ModelElement to the Namespace that contains it
- **Generalizes:** relates a GeneralizableElement to its ancestors (superclass and subclass)
- **IsOfType:** relates a TypedElement to the Classifier that defines its type
  - An object is an instance of a class
- **DependsOn:** relates a ModelElement to others that its definition depends on
  - E.g. a package depends on another package

MOF 2.0 Relationships



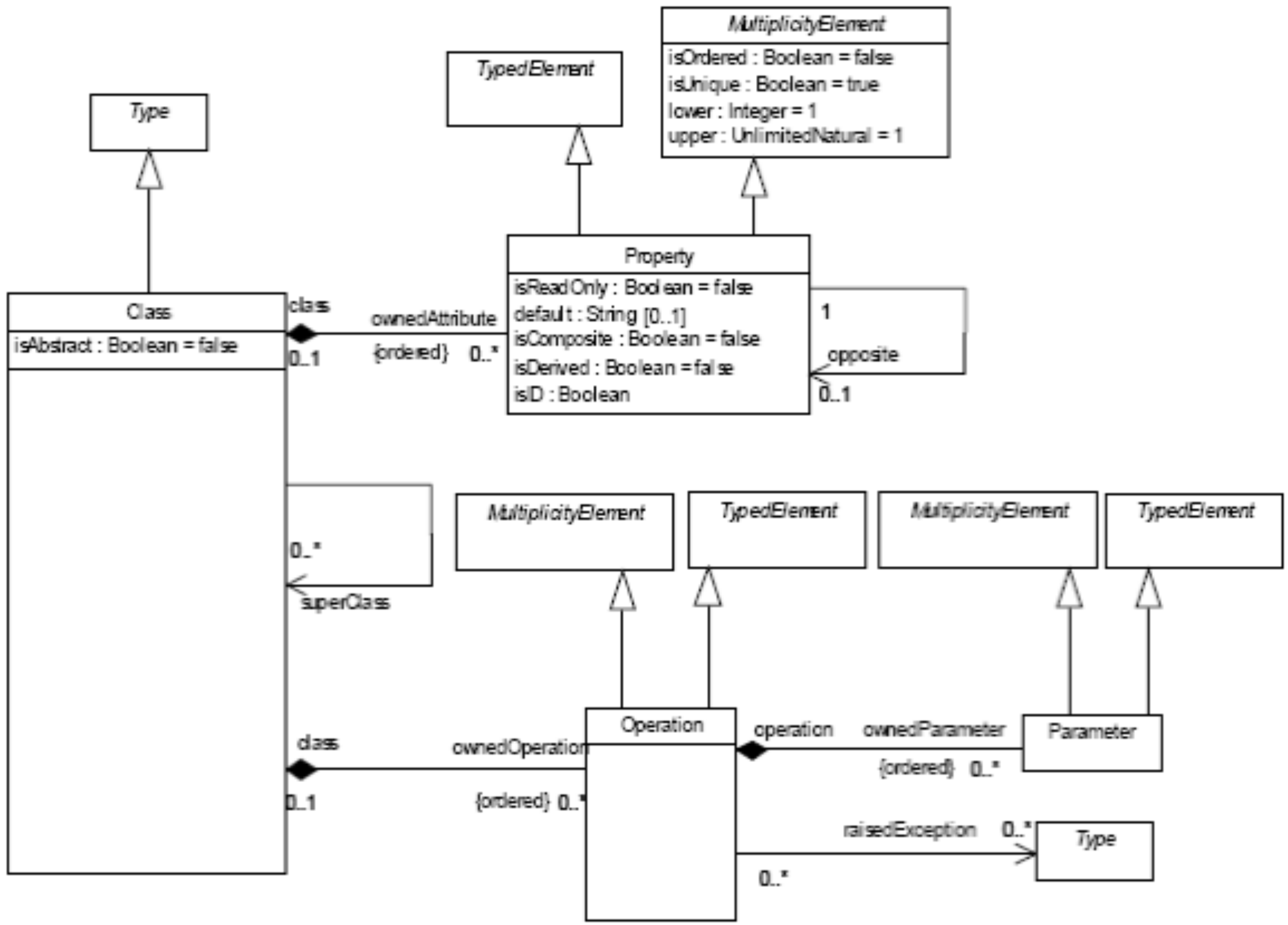
MOF 2.0 Relationships (II)



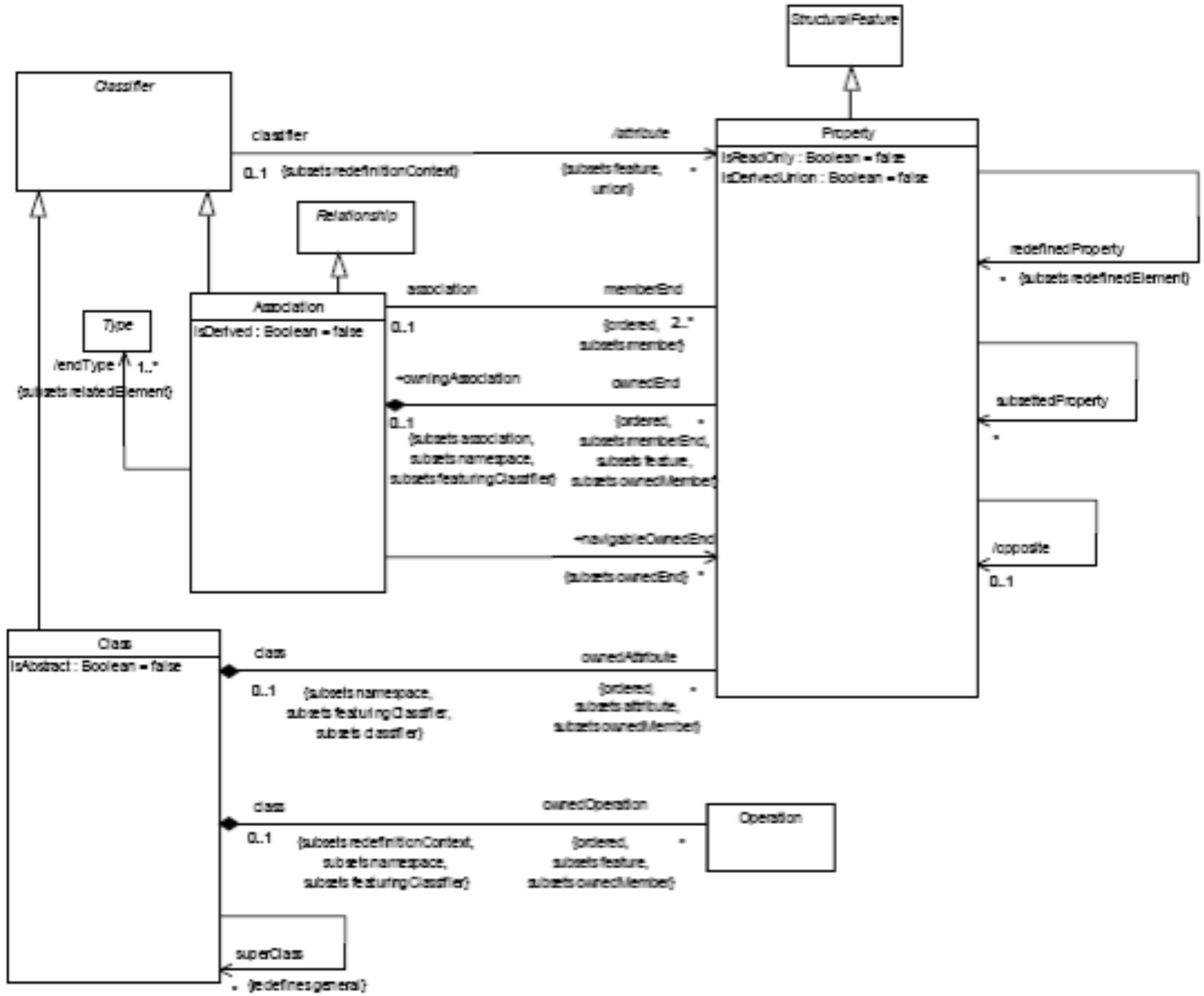
MOF 2.0 Structure

- MOF is separated into Essential MOF (EMOF) and Complete MOF (CMOF).
- EMOF corresponds to facilities found in OOP and XML.
  - Easy to map EMOF models to JMI, XMI, etc.
- CMOF is what is used to specify metamodels for languages such as UML 2.0.
  - It is built from EMOF and the core constructs of UML 2.0.
  - Really, both EMOF and CMOF are based on variants of UML 2.0.

EMOF Core Classes



CMOF Core Constructs



MOF Implementations

- Most widely known/used is EMF/ECore within Eclipse.
  - It is mostly compatible with MOF 1.x, and allows importing EMOF metamodels via XMI.
- The XMF-Mosaic tool from Xactium implements ExMOF (Executable MOF) which subsets and extends MOF 1.x.
- UML2MOF from Sun is a transformation from UML metamodels to MOF 1.x metamodels (with some bugs).
- Sun MDR implementation.
- Commercial implementations from Adaptive, Compuware, possibly MetaMatrix, MEGA, Unicorn.

Towards Tool Support

Why Should We Care about MDA?

1. It's not totally vaporware -- tools exist!
2. Programmers know that generating repeated code is eminently feasible.
   - MDA will pave the way for even more complex systems
   - The Generative Programming people have realised this for ages.
3. Smart people recognize many of the arguments against MDA were also used to oppose high-level languages vs. assembly language

MDD with EMF

- Contrary to most programmers' beliefs, modelling can be useful for more than just documentation
- Just about every program we write manipulates some data model
  - It might be defined using Java, UML, XML Schemas, or some other definition language
- EMF aims to extract this intrinsic "model" and generate some of the implementation code
  - Can be a tremendous productivity gain.
- EMF is one implementation of MOF (though it has differences).
  - We cannot claim that EMF = MOF!

EMF Model Definition

- Specification of an application's data
  - Object attributes
  - Relationships (associations) between objects
  - Operations available on each object
  - Simple constraints (e.g., multiplicity) on objects and relationships
- Essentially the Class Diagram subset of UML

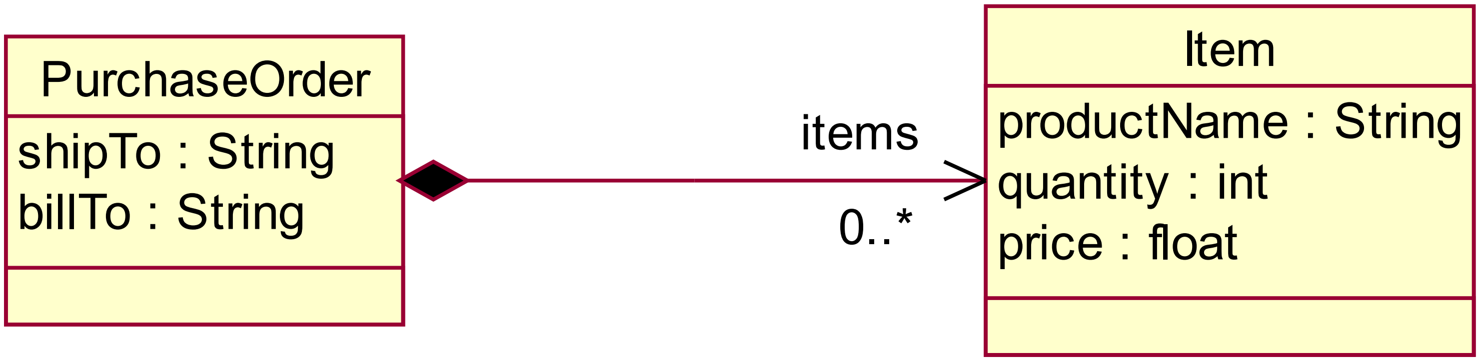
EMF Model Definition

- EMF models can be defined in (at least) three ways:
  1. Java interfaces
  2. UML Class Diagram
  3. XML Schema
- Choose the one matching your perspective or skills, and EMF can generate the others as well as the implementation code

EMF Model Definition
Java interfaces

```
public interface PurchaseOrder {
  String getShipTo();
  void setShipTo(String value);
  String getBillTo();
  void setBillTo(String value);
  List getItems(); // List of Item
}

public interface Item {
  String getProductName();
  void setProductName(String value);
  int getQuantity();
  void setQuantity(int value);
  float getPrice();
  void setPrice(float value);
}
```

EMF Model Definition - UML class diagrams



EMF Model Definition - XML

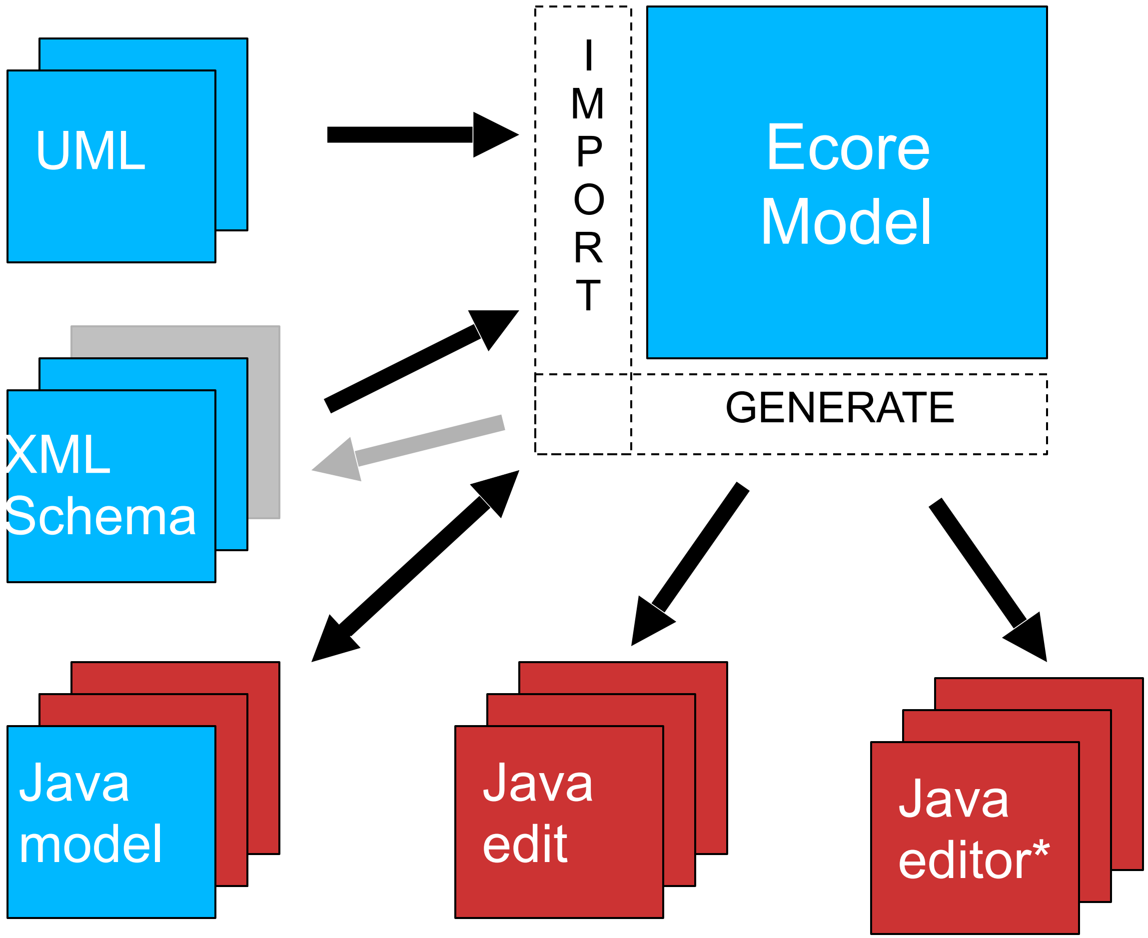
```
<xsd:complexType name="PurchaseOrder">
 <xsd:sequence>
  <xsd:element name="shipTo" type="xsd:string"/>
  <xsd:element name="billTo" type="xsd:string"/>
  <xsd:element name="items"  type="PO:Item"
               minOccurs="0" maxOccurs="unbounded"/>
 </xsd:sequence>
</xsd:complexType>

<xsd:complexType name="Item">
 <xsd:sequence>
  <xsd:element name="productName" type="xsd:string"/
>
  <xsd:element name="quantity" type="xsd:int"/>
  <xsd:element name="price" type="xsd:float"/>
 </xsd:sequence>
</xsd:complexType>
```

EMF Model Definition

Unifying Java, XML, and UML technologies

- All three forms provide the same information
  - Different visualization/representation
  - The application's "model" of the structure
- From a model definition, EMF can generate:
  - Java implementation code, including UI
  - XML Schemas
  - Eclipse projects and plug-ins

EMF Architecture
Model Import and Generation



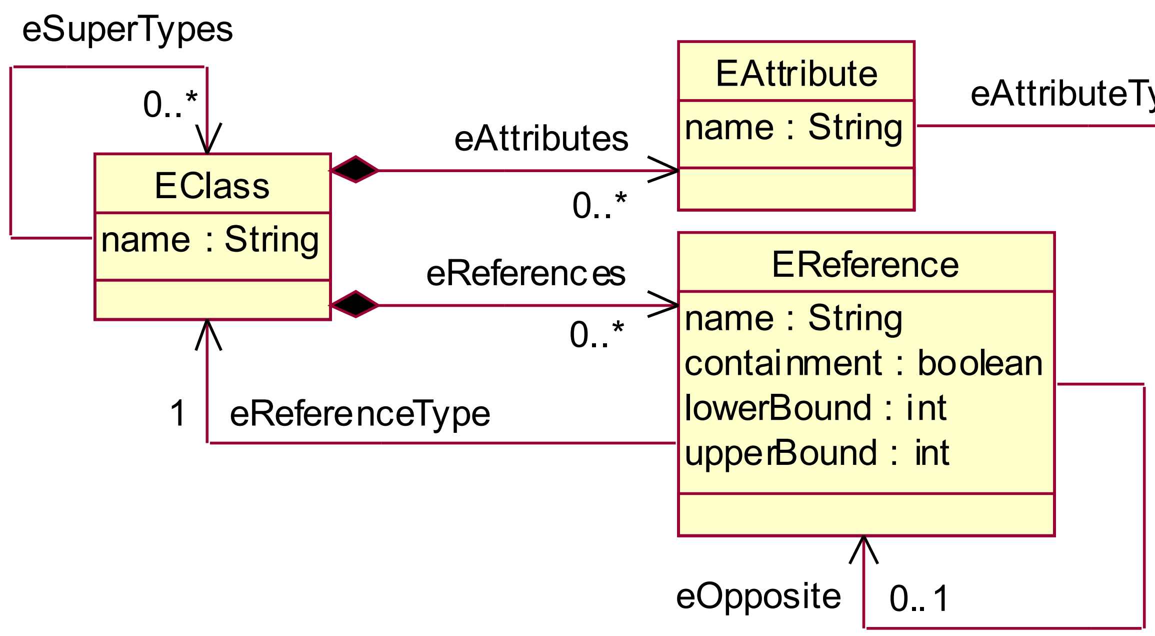
Generator features:

- Customizable JSP-like templates (JET)
- Command-line or integrated with Eclipse JDT
- Fully supports regeneration and merge

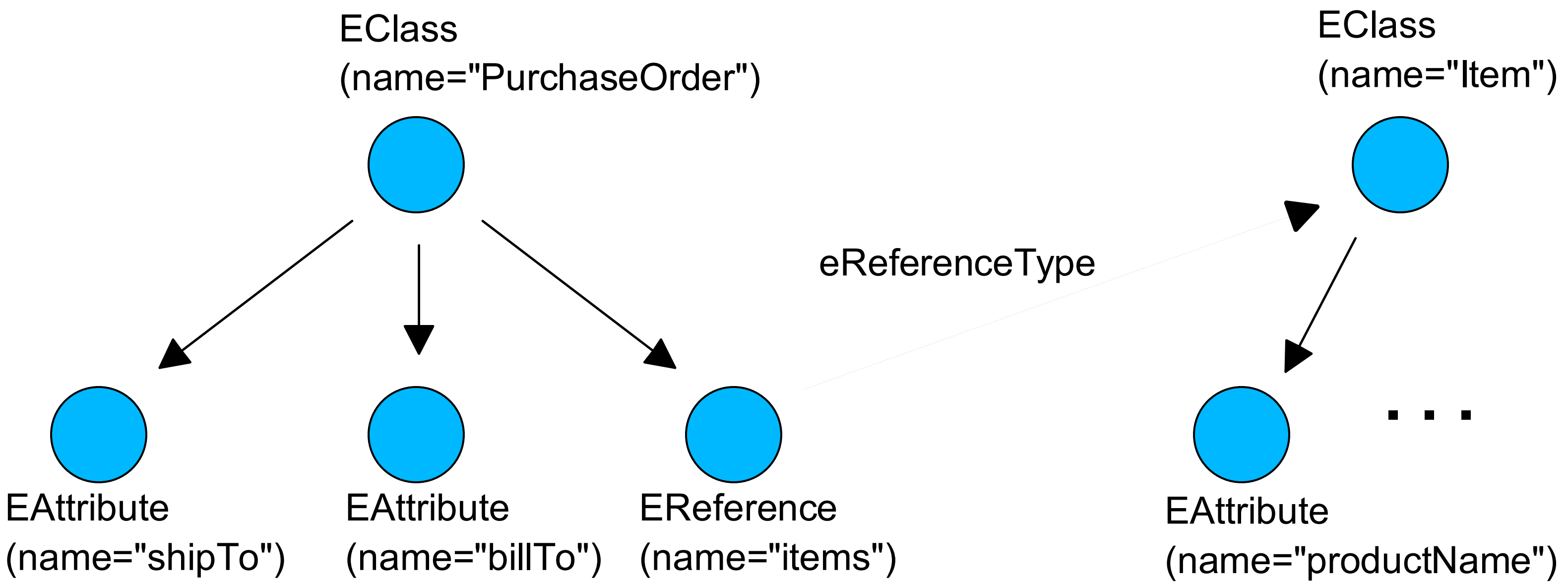
* requires Eclipse to run

EMF Architecture - Ecore

- Ecore is EMF's model of a model (metamodel)
  - Persistent representation is XMI



EMF Architecture - PurchaseOrder Ecore Model



EMF Architecture - PurchaseOrder Ecore XMI

```
<eClassifiers xsi:type="ecore:EClass"
  name="PurchaseOrder">
 <eReferences name="items" eType="#//Item"
   upperBound="-1" containment="true"/>
 <eAttributes name="shipTo"
   eType="ecore:EDataType http:...Ecore#//EString"/
>
 <eAttributes name="billTo"
   eType="ecore:EDataType http:...Ecore#//EString"/
>
</eClassifiers>
```

- Alternate serialization format is EMOF
  - Part of MOF 2.0 Standard as we saw earlier

EMF Dynamic Architecture

- Given an Ecore model, EMF also supports dynamic manipulation of instances
  - No generated code required
  - Dynamic implementation of reflective EObject API provides same runtime behavior as generated code
  - Also supports dynamic subclasses of generated classes
- All EMF model instances, whether generated or dynamic, are treated the same by the framework

EMF Architecture - Users

- IBM WebSphere/Rational product family
- Other Eclipse projects (XSD, UML2, VE, Hyades)
- ISV's (TogetherSoft, Ensemble, Versata, Omondo, and more)
- SDO reference implementation
- Large open source community

Code Generation - Feature Change

- Efficient notification from “set” methods
  - Observer Design Pattern

```
public String getShipTo() {
  return shipTo;
}

public void setShipTo(String newShipTo) {
  String oldShipTo = shipTo;
  shipTo = newShipTo;
  if (eNotificationRequired())
    eNotify(new ENotificationImpl(this, ... );
}
```

Code Generation

- All EMF classes implement interface EObject
- Provides an efficient API for manipulating objects reflectively
  - Used by the framework (e.g., generic serializer, copy utility, generic editing commands, etc.)
  - Also key to integrating tools and applications built using EMF

```
public interface EObject {
  Object eGet(EStructuralFeature f);
  void eSet(EStructuralFeature f, Object v);
  ...
}
```

Related Standards

- There is actually a family of standards related to MOF.
- MOF 2.0 Versioning:
  - for managing multiple, co-existing versions of metadata, and allowing inclusion in different systems in different configurations.
- MOF 2.0 Facility and Object Lifecycle:
  - Models object creation/deletion, move, comparison
  - Also models events that may be interesting.
- MOF 2.0 QVT.
- MOF Model-to-Text
- XMI.

MOF 2.0 Action Semantics

- What is Action Semantics?
- Current practice and limitations in capturing behaviour in MOF models
- MOF 2.0 Action Semantics
  - MOF AS Abstract syntax
  - Towards a MOF AS Concrete syntax
- Benefits, i.e., programmatic manipulation of models.
- Note: not a standard, evolving work, currently building a prototype implementation in Epsilon framework.

What is Action Semantics?

- Structural semantics capture the structural properties of a model
  - i.e., the model elements and their structural relationships
- Action semantics capture the behavior of a model
  - i.e., how the model behaves
- Actions semantics has been proposed for UML 2.0.
  - Variants appear in Executable UML work from Mellor et al.
- This has not addressed action semantics at the meta-metalevel, i.e., MOF 2.0.

Capturing behaviour in MOF

- In MOF models, behaviour is defined through operations
- OCL post-conditions can be used to define effects of the execution of an operation on the model
  - Post-conditions define the effects rather than how they are achieved
  - Allows flexibility in the implementation of the body of the operation

Limitations of post-conditions

- Cannot capture invocation of other operations
  - i.e., how do you say, in the post-condition, that another operation must be triggered?
  - This requires some notion of call semantics.
- Cannot capture algorithmic details necessary for efficiency.
  - e.g., you can specify that an operation sorts data, but how do you capture time bounds?
- Insufficient for simulation/execution
  - Only some post-conditions can actually be simulated (OCL in general is not fully executable).

MOF Action Semantics (AS)

- Extend MOF so that the we can capture **actions** performed
  - by invocation of operations
  - as response to model events
  - e.g. instance creation, attribute value update
- In order to achieve this we need
  - Abstract syntax
  - Concrete syntax (that implements the abstract syntax)

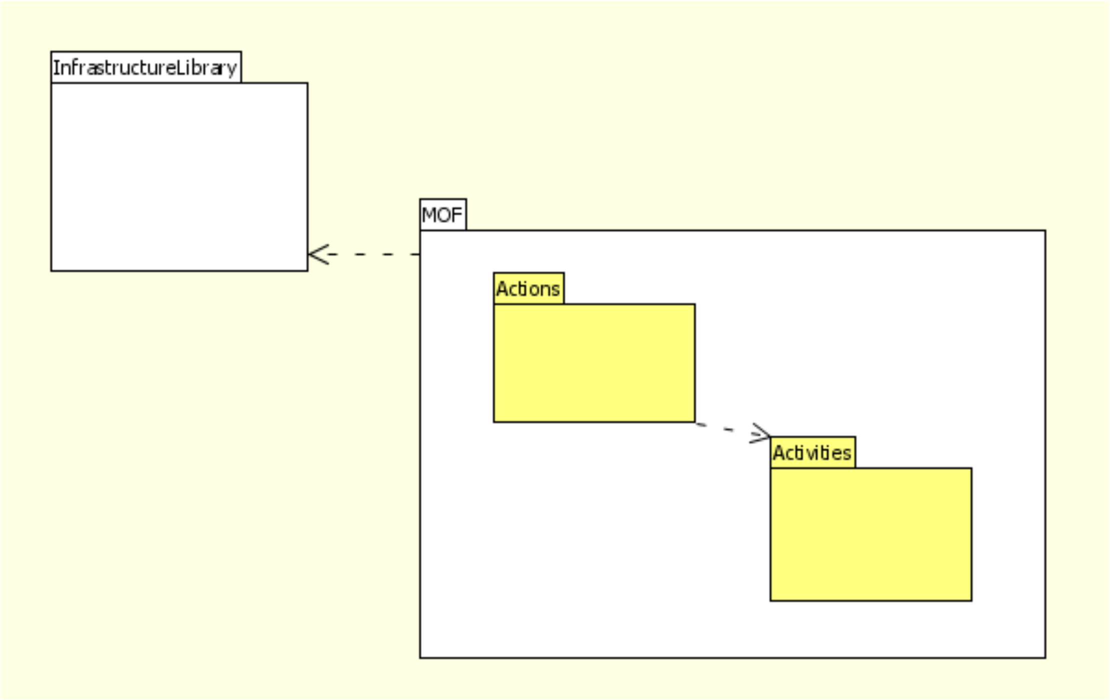
Actions

- Perform mathematical computations (Arithmetic, String, Boolean expressions)
- Control execution flow (if, for, while control structures etc)
- Create/Select/Delete object instances
- Read/Write instance attribute values
- Create/Delete relationships instances
- Navigate relationships
- Invoke other operations
    - cf., UML 2.0 Action Semantics

MOF AS Abstract Syntax

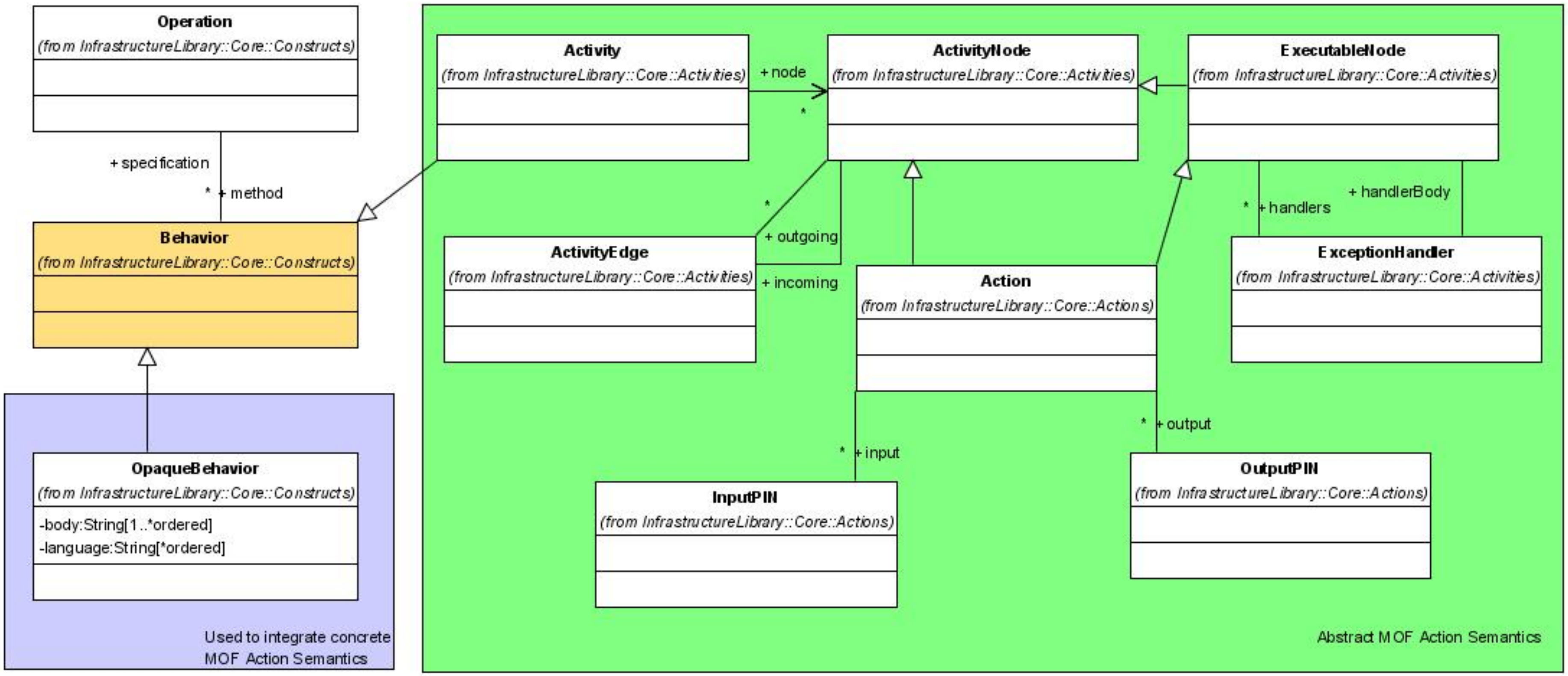
- Use the existing UML AS abstract syntax as a base
  - Port the "actions" and "activities" package of the "UML" package into the "MOF" package
  - Update the "operation" meta-class
  - Update ported meta-classes to match MOF modelling elements (instead of UML)
  - Remove classes that do not fit the MOF level of abstraction

Abstract Syntax: Package



- AS is a restriction of UML 2.0 AS.

Abstract Syntax



Abstract Syntax Details

- An Operation has multiple possible behaviours.
  - Activities are behaviours, and the activity graph is captured using ActivityNode and ActivityEdge.
  - A special kind of ActivityNode is an ExecutableNode, which may have a number of ExceptionHandlers, each of which also have ExecutableNodes.
- An Action is both an ActivityNode and an ExecutableNode.
  - Generalizations of Action will provide the computational behaviour needed to write action programs.
  - Finally, an Action has input and output PINs.
- Concrete syntax for the MOF action semantics is contained within the OpaqueBehavior.

AS Notes

- Possible to simplify this structure further by inferring the Activity graph (i.e., ActivityNode and ActivityEdge):
  - Actions know their precursor and successor, which can be used to implicitly extract the information encoded in nodes and edges.
  - This closely mimics trace semantics, as seen, for example in Communicating Sequential Processes.
- Computational behaviour is captured via generalization of the Action metaclass.
  - UML 2.0 contains approx 60 metaclasses for this.
  - We can add everything - trivially - but then MOF 2.0 + AS is 170 or so classes; is this worthwhile?

MOF AS Concrete Syntax

- Abstract AS is useful as foundation but insufficient.
- Need a concrete language
- We propose the use of a procedural (C-style) language like
  - Kabira Action Semantics, BridgePoint Object Action Language, KC Action Specification Language
- … but instead of proprietary model-querying expressions, integrate support for OCL statements
- No point creating a new language until UML 2.0 is stabilized.
  - However, we have developed the Epsilon Object Language which could be used for parts of this.

Benefits from MOF AS (1/2)

- Precise and executable meta-models
  - a metamodel enhanced with AS should be sufficient to drive a modelling tool
- Programmatic model manipulation
  - an executable language on top of MOF will allow programmatic manipulation of MOF-based models (e.g. UML models)

Programmatic model manipulation

- Task automation
  - e.g. a user can define that when an attribute is added into a UML class, a setter and getter operation are automatically added
- Intra-language transformations
  - perform intra-language transformations without having to define mapping rules for each element of the modelling language

Challenges

- MOF has gone through a major revision recently (MOF 2.0)
  - Consequently, it is doubtful that MOF can be changed again (soon) to include AS
  - Also MOF 2.0 is already 110+ classes; can we add 60 more for AS and get away with it?
- OMG should standardize a concrete AS language to facilitate interoperability between tools
  - debatable whether there is enough motivation for it

Transformations and Mappings
Uses of MOF in Practice

MDA in Practice

- There are three key techniques used in applying MDA in practice:
  - metamodelling (which is usually done by experts prior to systems development, using MOF-based languages);
  - modelling (done by systems engineers, using UML-based languages);
  - transformations between models (using QVT).
- Let's see an example of transformations.

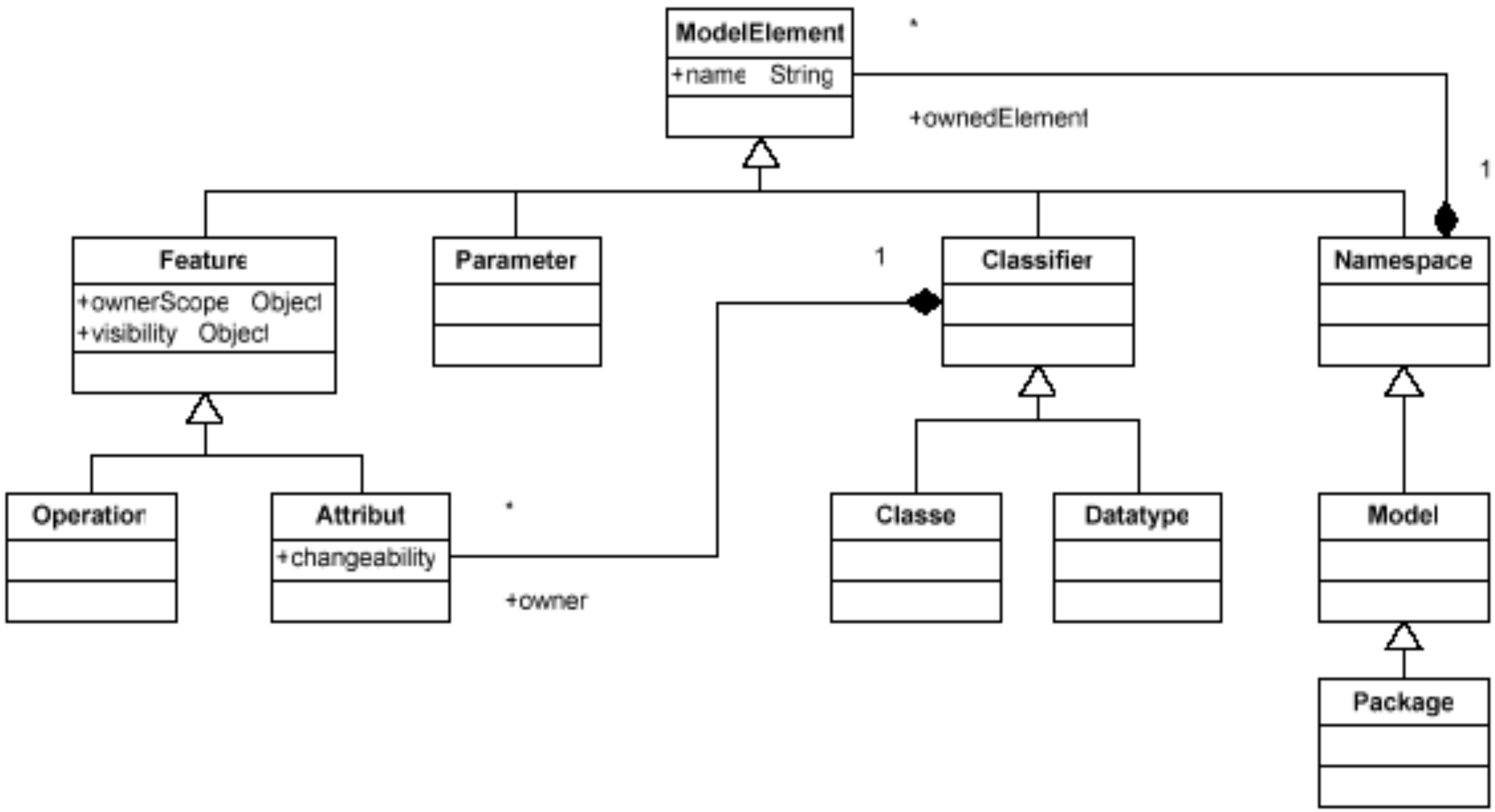
Example - Transformations with ATL

- ATL (Atlas Transformation Language)
- A declarative and imperative language for expressing model transformations.
- Transformations are expressed as a set of rules on metamodels.
  - Metamodel for source and target language.
- But transformations are themselves models, and have a metamodel.
- This means that you can define transformations on transformations!

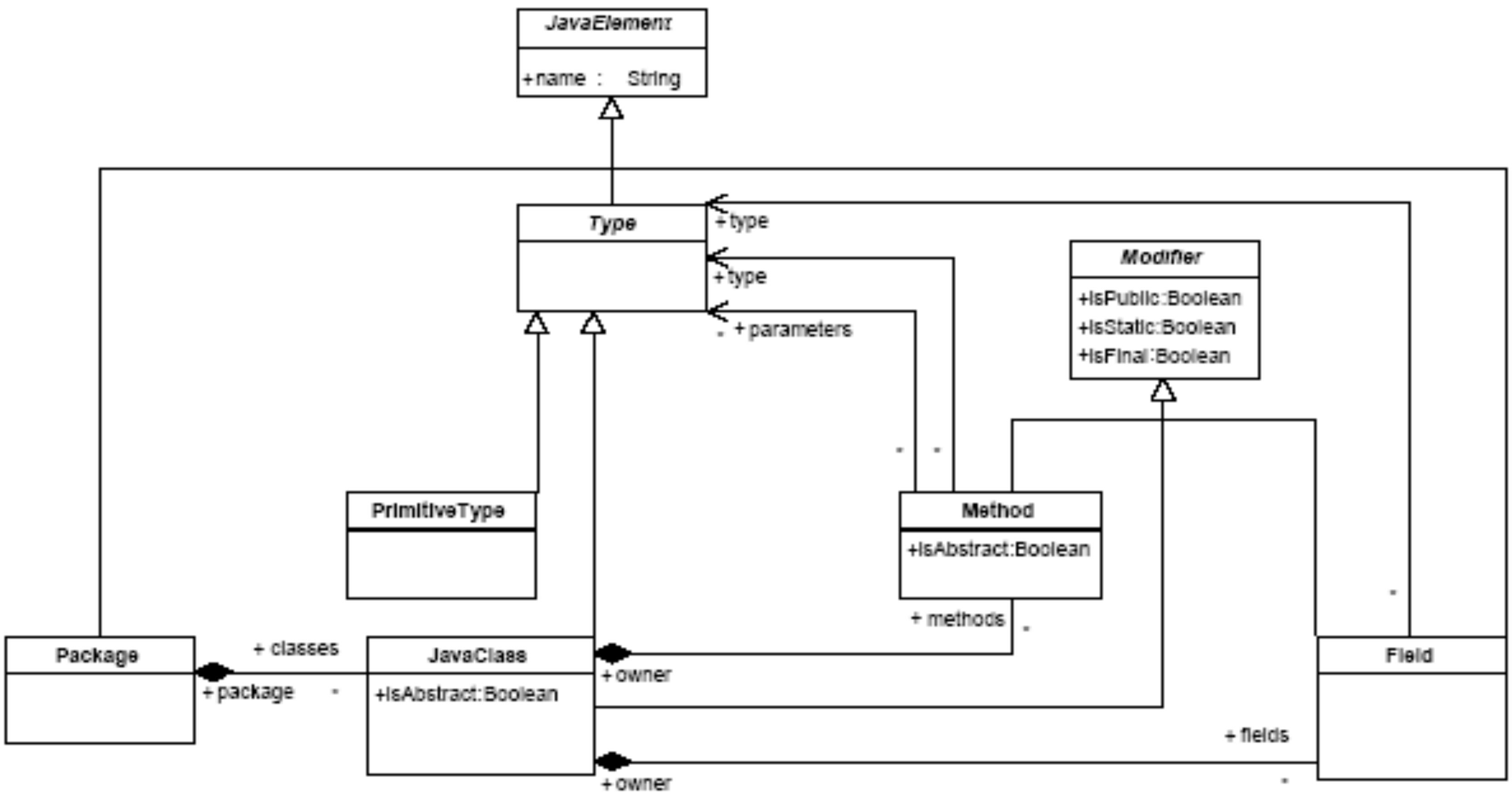
Example: UML to Java

- Transform a simple subset of UML into Java using ATL.
- Need a simple UML metamodel and a simple Java metamodel.
- Also need a set of transformation rules.

Source UML Metamodel



Target Java Metamodel



Rules (Informal)

- **For each UML Package instance, a Java Package instance has to be created.**
  - Their names have to correspond. However, in contrast to UML Packages which hold simple names, the Java Package name contains the full path information. The path separation is a point ".".
- **For each UML Class instance, a JavaClass instance has to be created.**
  - Their names have to correspond.
  - The Package reference and Modifiers have to correspond.
- **For each UML DataType instance, a Java PrimitiveType instance has to be created.**
  - Their names have to correspond.
  - The Package reference has to correspond.
- **For each UML Attribute instance, a Java Field instance has to be created.**
  - Their names, Types, and Modifiers have to correspond.
  - The Classes have to correspond.
- **For each UML Operation instance, a Java Method instance has to be created (similar to above)**

ATL Rules (Examples)

```
rule P2P {
	from e : UML!Package (e.oclIsTypeOf(UML!Package))
	to out : JAVA!Package (
		name <- e.getExtendedName()
	)
}

rule C2C {
	from e : UML!Class
	to out : JAVA!JavaClass (
		name <- e.name,
		isAbstract <- e.isAbstract,
		isPublic <- e.isPublic(),
		package <- e.namespace
	)
}
```

ATL Rules (Examples)

```
rule O2M {
	from e : UML!Operation
	to out : JAVA!Method (
		name <- e.name,
		isStatic <- e.isStatic(),
		isPublic <- e.isPublic(),
		owner <- e.owner,
		type <- e.parameter->select(x|x.kind=#pdk_return)->
				asSequence()->first().type,
		parameters <- e.parameter->select(x|x.kind<>#pdk_return)->
				asSequence()
	)
}
```

- Sometimes need to define “helpers” (intermediate functions) to simplify specifications.

Compositions

Model Compositions

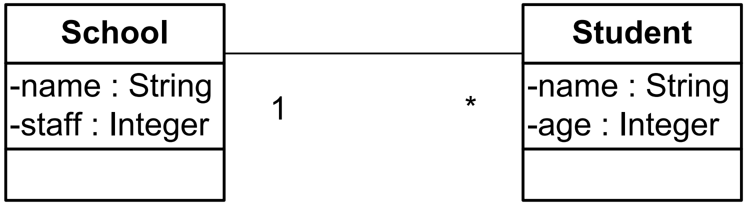
- Also (somewhat confusingly) called
  - model merging
  - model integration
  - model unification
- Basic idea: combining two (or more) distinct models into a single model.
- e.g., combining two UML class diagrams into a single class diagram.
- e.g., combining two or more XML schemas into a single XML schema.

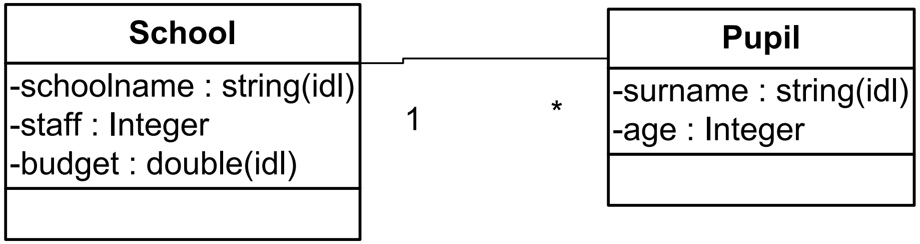
Why Is Composition Useful?

- To support teamwork.
  - Different individuals working on the same model at the same time.
  - Need to reconcile these different versions.
- To support the "MDA vision".
  - PIM + PDM leads to PSM.
- To support flexible styles of modelling.
  - e.g., adding exception modelling or traceability capacity to a system.
  - Construct a "traceability" metamodel or an "exception" metamodel and merge it with a system model.

Why Is Composition Hard?

- It's all about resolving inconsistencies.





Some Composition Issues

- How to identify model elements that match?
- How to identify model elements that conform (e.g., based on semantic properties)?
- How to deal with model elements for which no equivalent exists (e.g., extra attributes)?
- How to deal with clashes?
- Conclusion: It's impossible to automatically merge models.
  - A language is needed to describe when elements match, conform, clash, etc.

Epsilon Merging Language

- EML is one approach to merging models.
  - Developed here at York.
- There are others, e.g., Atlas Model Weaver, and the Glue Generator Tool.
- EML is more of a programmatic solution than AMW or GGT.
- Currently supports MOF 1.x (via MDR), EMF/EMOF, and XML-based metamodels, but there is no restriction as to repository/metamodelling framework.
- http://www.cs.york.ac.uk/~dkolovos/epsilon

EML Overview

- The Epsilon Merging Language (EML) is a language that supports the previously identified phases of model merging
- The EML uses a generic model management language, called EOL, as an infrastructure language.
  - EOL is like OCL, but it also supports model modification, and is not restricted to MOF-based languages.
- Therefore EML can be used to merge different types of models.

Phases of Model Merging

- Compare
  - Discover the corresponding concepts in the source models
- Conform
  - Resolve conflicts and align models to make them compatible for integration
- Merge
  - Merge common concepts of the source models and port non-matching concepts
- Restructure
  - Restructure the merged model so that it is semantically consistent

Structure of an EML Specification

- An EML specification consists of three kinds of rules:
  - Match rules
  - Merge rules
  - Transform rules
  - It also contains a pre and a post block that are executed before and after the merging respectively to perform tasks that are not pattern-based

Structure of Match Rules

- Each Match Rule has a unique name, and two meta-class names as parameters
- A Match Rule can potentially extend one or more other Match Rules and/or be declared as abstract
- It is composed of a **Guard**, a **Compare** and a **Conform** part and is executed for all pairs of instances of the two meta-classes in the source models
  - The **Guard** part is a constraint for the elements the rule applies to (i.e., a boolean expression)
  - The **Compare** part decides on whether the two instances match using a minimum set of criteria (side-effect free)
  - For matching instances, the **Conform** part decides on whether the instances fully conform with each other (side-effect free)
  - The scheduler executes compare rules, then conform rules.

Example Match Rules

```
abstract rule ModelElements
    match l : Left!ModelElement
    with r : Right!ModelElement
    extends Elements {

    compare {
        return l.name = r.name
            and l.namespace.matches(r.namespace);
    }
}

rule Classes
    match l : Left!Class
    with r : Right!Class
    extends ModelElements {

    conform {
        return l.isAbstract = r.isAbstract;
    }
}
```

```
rule StructuralFeatures
    match l : Left!StructuralFeature
    with r : Right!StructuralFeature
    extends ModelElements {

    compare {
        return l.owner.matches(r.owner);
    }

    conform {
        return l.type.matches(r.type);
    }
}
```

Categories of Model Elements

- After the execution of the match rules, 4 categories of model elements are identified:
  1. Elements that **match and conform** to elements of the opposite model
  2. Elements that **match but do not conform** to elements of the opposite model.
     - Existence of this category of elements triggers cancellation of the merging process.
  3. Elements that **do not match** with any elements of the opposite model
     - A transform rule is applied to port these elements to the target metamodel.
  4. Elements on which **no matching rule** has applied
     - Existence of this category of elements triggers warnings

After Matching…

- Elements of Category 1 (matching **and** conforming) will be merged with their match.
  - The specification of merging is defined in a **Merge Rule**
- Elements of Categories 3 and 4 (not matching) will be transformed into model elements compatible with the target metamodel.
  - The specification of transformation is defined in a **Transform Rule**
  - Additionally, elements in category 4 generate warnings (useful feedback in terms of whether or not a set of rules is "complete").

Structure of Merge Rules

- Each **Merge Rule** is defined using a unique name, two meta-class names as parameters and the meta-class of the model element that the rule creates in the target model
- It can extend other Merge Rules and/or be declared as abstract
- For all pairs of matching instances of the two meta-classes that satisfy the **Guard** of the rule, the rule is executed and an empty model element is created in the target model
- The contents of the newly created model element are defined by the body of the Merge Rule

Example Merge Rules

```
rule ModelElements
    merge l : Left!ModelElement
    with r : Right!ModelElement
    into m : Merged!ModelElement {

    m.name := l.name;
    m.namespace := l.namespace.equivalent();

}
```

```
rule Classes
    merge l : Left!Class
    with r : Right!Class
    into m : Merged!Class
    extends ModelElements {

    m.feature := l.feature.
                    includeAll(r.feature).
                    equivalent();
}
```

- The equivalent() operation returns the equivalent of the model element, on which it is applied, in the target model
- The equivalent of an element is the result of a **Merge Rule** if the element has a matching element in the opposite model; else it is the result of a **Transform Rule**

Structure of Transform Rules

- Each **Transform Rule** is defined using a unique name, a meta-classes, instances of which it can transform and a meta-class that declares the type of the target of the transformation
- Transform rules can also extend other Transform Rules and/or be declared as abstract
- For all instances of the meta-classes that have no matching elements in the opposite model, and for which the **Guard** is satisfied, the rule is executed and an empty model element (of the declared meta-class) is created
- The contents of the newly created element are defined by the body of the Transform Rule

Example Transform Rules

```
abstract rule ModelElement2ModelElement
    transform s : Uml!ModelElement
    to t : Merged!ModelElement {

    t.name := s.name;
    t.namespace := s.namespace.equivalent();
}

rule Class2Class
    transform s : Uml!Class
    to t : Merged!Class
    extends ModelElementToModelElement {

    t.feature := s.feature.equivalent();

}
```

- Note that Uml!Class refers to both instances of Left!Class and Right!Class since Left and Right have been declared to follow the Uml metamodel

Further Automating Model Merging

- EML makes it feasible to merge any pair of models
- However, writing the full merging specification by hand is not always practical. Useful information can be obtained from elsewhere
- For example in the case the source and the target models are of the same meta-model (e.g. all are UML models), merging and transformation rules can be inferred by the structure of the meta-model

Merging Strategies

- Inference of rules that are not explicit in the merging specification is performed by **Merging Strategies**.
- Each merging strategy defines two methods:
  - autoMerge(left:Object, right:Object) : Object
  - autoTransform(source:Object) : Object
- Each instance of the EML engine has a related MergingStrategy. In case it needs to match merge or transform specific elements for which no rule has been defined, it uses the behaviour defined in its MergingStrategy

The MOF/EMF Common Metamodel Strategy

- An example MergingStrategy we have implemented provides support for models of the same (either MOF or EMF) meta-model. Its functionality follows:
  - autoMerge
    - Can merge two instances of the same meta-class.
    - Creates a new instance of the meta-class in the target model.
    - For single-valued features of the meta-class it uses the values defined in the instance from the left model
    - For multi-valued features it uses the union of the values of the left and right instances
  - autoTransform
    - Creates a deep copy of the source model element in the target model

Overriding the Strategy behavior

- As we mentioned, the behavior defined in the Merging Strategy is invoked when no rule has been explicitly defined in the specification
- This always allows the developer to override the default behavior
- The use of the **auto** keyword in EML Merge and Transform rules also allows the developer to **complement** the strategy behavior
- By using the **auto** keyword, the engine first runs the strategy behavior and then the **explicit** behavior

Example of overriding behavior

```
auto rule ModelWithModel
    merge l : Left!Model
    with r : Right!Model
    into m : Merged!Model {

    m.name := l.name + ' and ' + r.name;
}
```

- The behavior of the strategy merges the two instances and since name is a single-valued feature, it uses the name of the left instance as the name of the merged instance
- The above displayed rule overrides this behavior and sets the name of the merged instance to left.name + 'and' + right.name