QVT Operational

MOF QVT: OMG's model-to-model transformation standard

- **QVT** stands for **Q**uery/**V**iews/**T**ransformations
  - OMG standard language for expressing *queries*, *views*, and *transformations* on MOF models

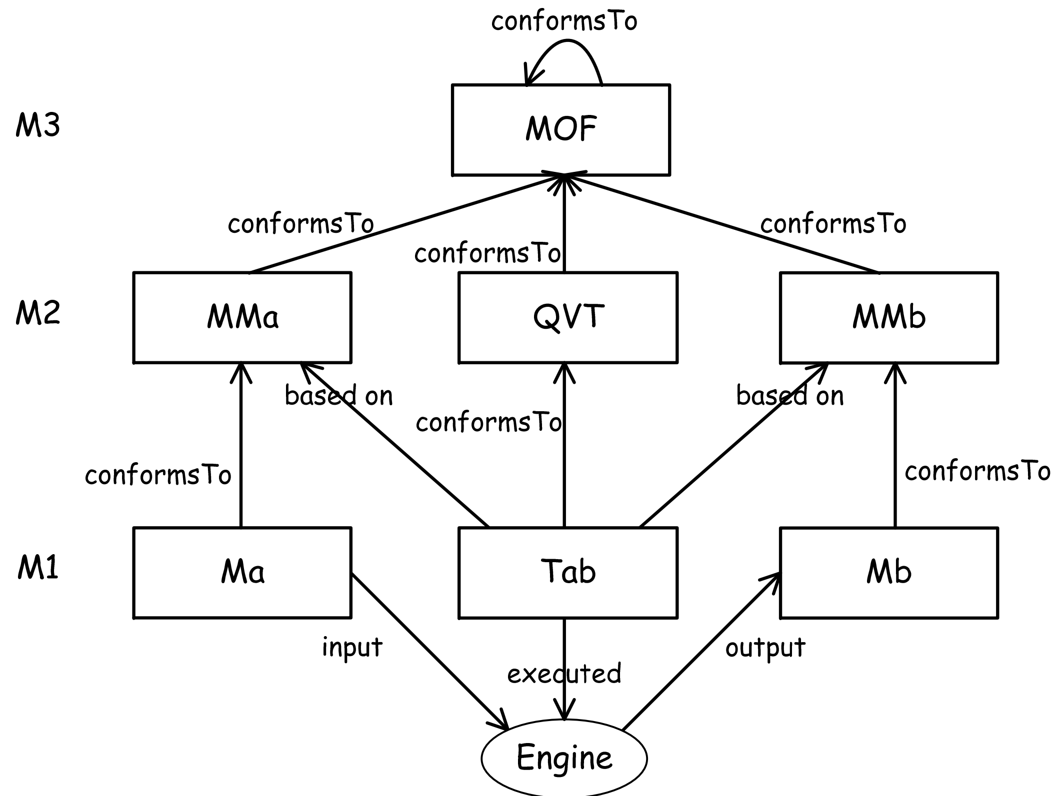
- OMG QVT Request for Proposals (QVT RFP, ad/02-04-10) issued in 2002
  - Seven initial submissions that converged to a common proposal
  - Current status (June, 2011): version 1.1, formal/11-01-01

http://www.omg.org/spec/QVT/1.0/

http://www.omg.org/spec/QVT/1.1/

MOF QVT context

- Abstract syntax of the language is defined as MOF 2.0 metamodel
  - Transformations (*Tab*) are defined on the base of MOF 2.0 metamodels (*MMa*, *MMb*)
  - Transformations are executed on instances of MOF 2.0 metamodels (*Ma*)



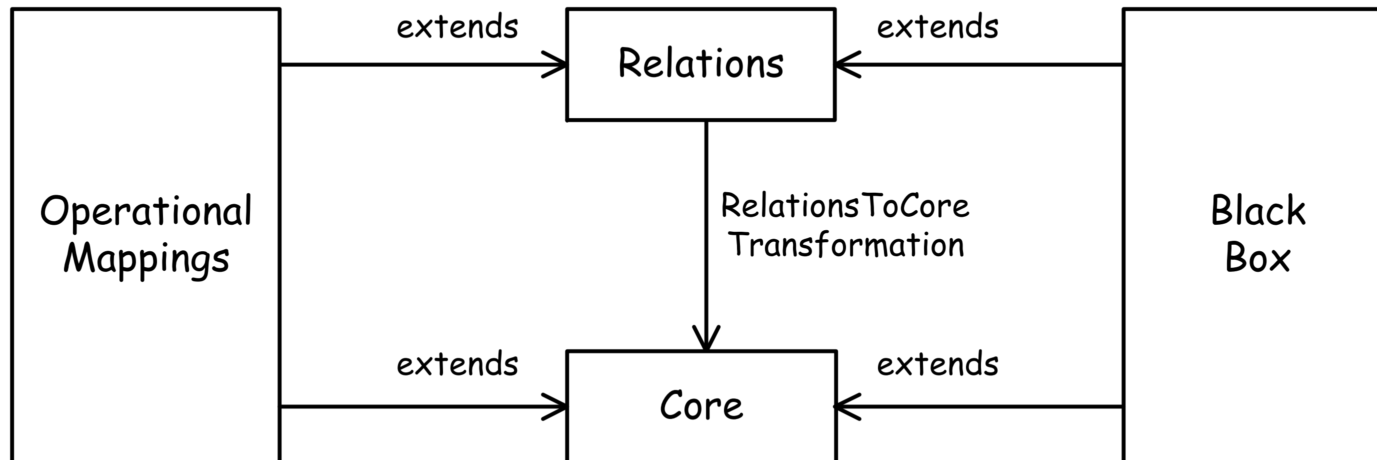
Requirements for MOF QVT language

- Some requirements formulated in the QVT RFP

Mandatory requirements	
Query language	Proposals shall define a language for querying models
Transformation language	Proposals shall define a language for transformation definitions
Abstract syntax	The abstract syntax of the QVT languages shall be described as MOF 2.0 metamodel
Paradigm	The transformation definition language shall be declarative
Input and output	All the mechanisms defined by proposals shall operate on models instances of MOF 2.0 metamodels
Optional requirements	
Directionality	Proposals may support transformation definitions that can be executed in two directions
Traceability	Proposals may support traceability between source and target model elements
Reusability	Proposals may support mechanisms for reuse of transformation definitions
Model update	Proposals may support execution of transformations that update an existing model

MOF QVT architecture

- Layered architecture with three transformation languages:
  - **Relations** (declarative)
  - Core (declarative, simpler than Relations)
  - **Operational Mappings** (imperative)
- Black Box is a mechanism for calling external programs during transformation execution
- QVT is a set of three languages that collectively provide a hybrid “language”.



Overview of Operational Mappings (OM)

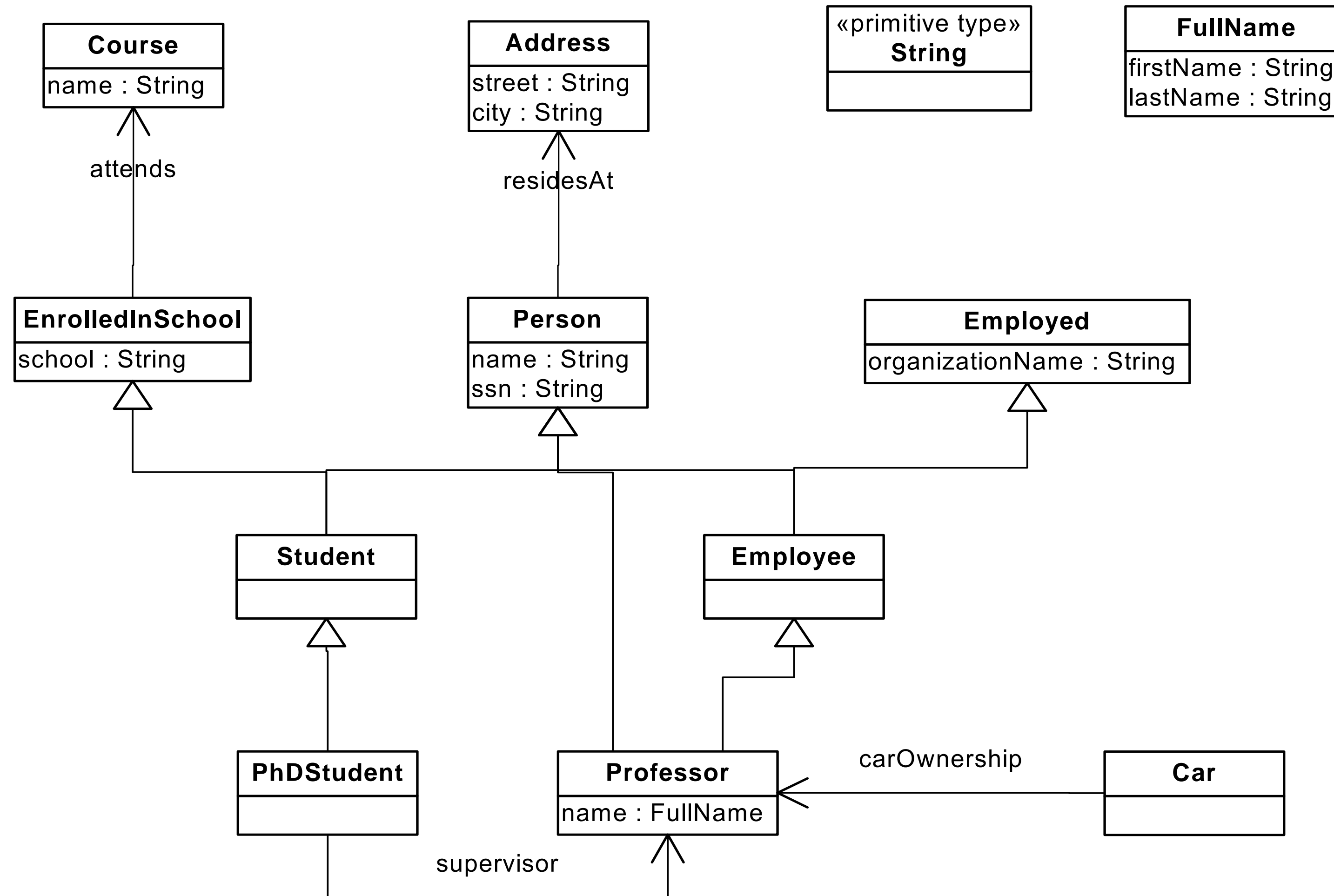
- Imperative transformation language that extends relations
- OM execution overview:
  - **Init**: code to be executed before the instantiation of the declared outputs.
  - **Instantiation** (internal): creates all output parameters that have a null value at the end of the initialization section
  - **Population**: code to populate the result parameters and the
  - **End**: code to be executed before exiting the operation. Automatic handling of traceability links
- Transformations are unidirectional
- Supported execution scenarios:
  - Model transformations
  - In-place update
- OM uses explicit internal scheduling, where the sequence of applying the transformation rules is specified within the transformation rules
- Updates have to be implemented in the model transformations

Flattening class hierarchies example

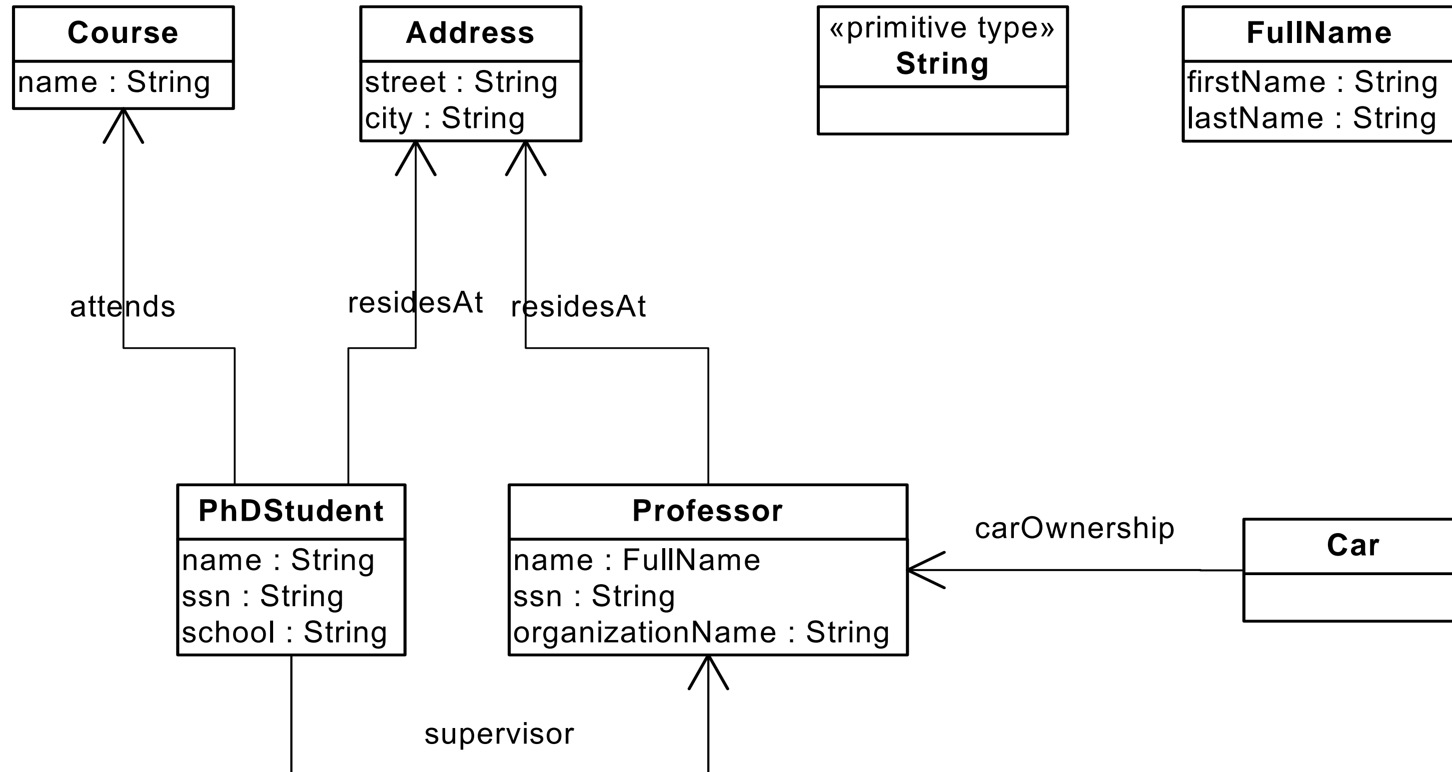
- Flattening UML class hierarchies: given a source UML model transform it to another UML model in which only the leaf classes (classes not extended by other classes) in inheritance hierarchies are kept.

- Rules:
  - Transform only the leaf classes in the source model
  - Include the inherited attributes and associations
  - Attributes with the same name override the inherited attributes
  - Copy the primitive types

Sample input model



Sample output model



OM language: Transformation program structure

```
transformation
flatten
(in hierarchical : UML,
 out flat : UML);
```

Signature: declares the transformation name and the source and target metamodels. **in** and **out** keywords indicate source and target model variables.

```
main() {
  …
}
…
```

Entry point: execution of the transformation starts here by executing the operations in the body of `main`

helper declarations

…

mapping operations declarations

Transformation elements: Transformation consists of mapping operations and helpers forming the transformation logic.

Mapping operations

- A mapping operation maps one or more source elements into one or more target elements
- Always unidirectional
- Selects source elements on the base of a type and a Boolean condition (guard)
- Executes operations in its body to create target elements
- May invoke other mapping operations and may be invoked
- Mapping operations may be related by inheritance, merging, and disjunction

General structure of mapping operations

```
mapping Type::operationName(((in|out|inout) pName : pType)*)
   : (rName : rType)+
  when {guardExpression}              pre-condition
  where {guardExpression} {           post-condition
  init {
     ...        init section contains code executed before the instantiation of the declared result
  }             elements
```

There exists an implicit instantiation section that creates all the output parameters not created in the init section. The trace links are created in the instantiation section.

```
  population {
    ...
  }
```

population section contains code that sets the values or the result and the parameters declared as `out` or `inout`. The `population` keyword may be skipped. The population section is the default section in the operation body.

```
  end {
  ...
  }
}
```

end section contains code executed before exiting the operation

Mapping operations: Example

- Rule for transforming leaf classes
  - selects only classes without subclasses, collects all the inherited properties and associations, creates new class in the target model

target type: instance created on call

```
mapping Class::copyLeafClass() : Class
  when {
    not hierarchical.allInstances(Generalization)->exists(g | g.general = self)
  } {
  name := self.name;
  ownedAttribute += self.ownedAttribute.
                      map copyOwnedProperty();
  ownedAttribute += (self.allFeatures()[Property] -
                       self.ownedAttribute).copyProperty(self);
  self.allFeatures()[Property]->select(p |
    not p.association.oclIsUndefined()).association.copyAssociation(self);
}
```

object on which mapping is called

call of another mapping

guard: mapping operation only executed for elements for which the guard expression evaluates to true

call of a helper

- Mappings only executed once
- Call of mappings with OCL-syntax (*collection*`->map` vs. *object*`.map`)

Helpers: Example

meta-model extension

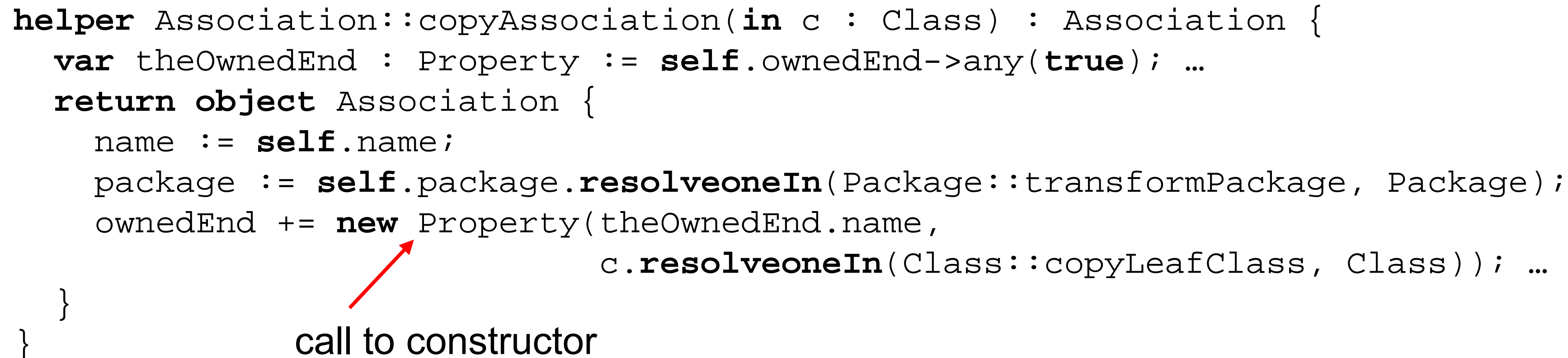
object creation and population

```
intermediate property Property::mappedTo : Set(Tuple(c : Class, p : Property));

helper Property::copyProperty(in c : Class) : Property {
  log('[Property] name = ' + self.name);
  var copy := object Property {
    name := self.name;
    type := self.type.map transformType();
  };
  self.mappedTo += Tuple{ c = c, p = copy };
  return copy;
}
```

Resolving object references

- The transformation engine maintains links among source and target model elements. These links are used for resolving object references from source to target model elements and back.
  - `resolveIn` is an operation that looks for model elements of a given type (`Class`) in the target model derived from a source element by applying a given rule (`copyLeafClass`).



```
helper Association::copyAssociation(in c : Class) : Association {
  var theOwnedEnd : Property := self.ownedEnd->any(true); …
  return object Association {
    name := self.name;
    package := self.package.resolveoneIn(Package::transformPackage, Package);
    ownedEnd += new Property(theOwnedEnd.name,
                             c.resolveoneIn(Class::copyLeafClass, Class)); …
  }
}
```


  - Variants: `resolve(`i `|` exp`)`, `resolveone(`i `|` exp`)`
  - `late resolve` for resolving **after** the transformation (in order of calls)

Mapping operations: Disjunction, inheritance, merging

```
mapping DataType::copyDataType() : DataType {
  name := self.name;
  ownedAttribute += self.ownedAttribute.map copyOwnedProperty();
}

mapping PrimitiveType::copyPrimitiveType() : PrimitiveType {
  init {
    result := self.deepclone().oclAsType(PrimitiveType);
  }
}

mapping Type::transformType() : Type
  disjuncts DataType::copyDataType,
            Class::copyLeafClass,
            PrimitiveType::copyPrimitiveType;
```

- Inherited rules executed after `init`
- Merged rules executed after `end`

Imperative OCL constructs

- More sophisticated control flow
  - `compute (`*v* `:` *T* `:=` *exp*`)` *body*
    - like `let … in`
  - `while (`*cond*`)` *body*
  - *coll*`->forEach (`*i* `|` *exp*`)` *body*
  - `break`, `continue`
  - `switch`-statement, exceptions