Model Transformations

Markus Völter
voelter@acm.org
www.voelter.de

www.mdsd-buch.de

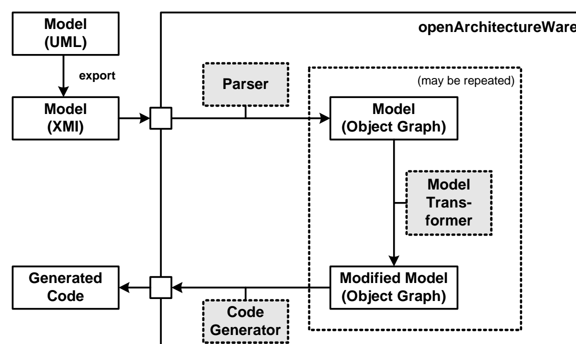
www.mdsd-book.org

Together with
Sven Efftinge
and Arno Haase

 © 2003 - 2006 Markus Völter

Transforming “in the Tool”

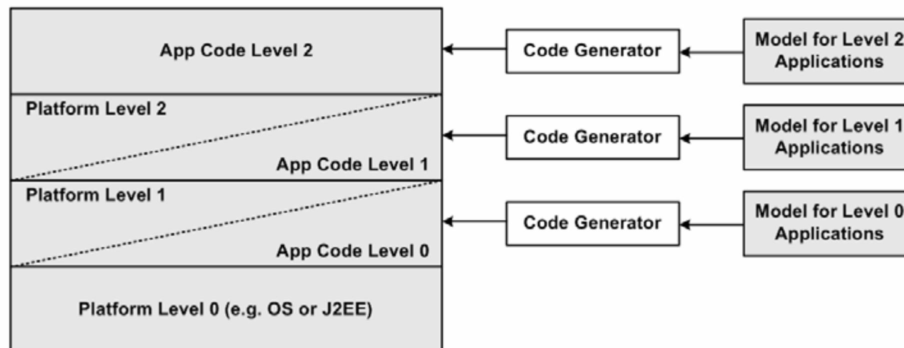
- Note that we consider an M2M transformation an “**implementation detail**” of the generator tool.
- We strongly **recommend not to try to modify intermediate models** – they serve as a means for modularizing generators only!
- Use a reference model together with model-specific constraints for testing the transformations
- Use **model weaving** to introduce additional data into the intermediate models



 © 2003 - 2006 Markus Völter

Layered MDSD: Platform Stacking

- The generated code of the lower layer **serves as the platform** for the next higher level
- A sequence of generation steps is used, whereas each of the generates code on which the next step builds



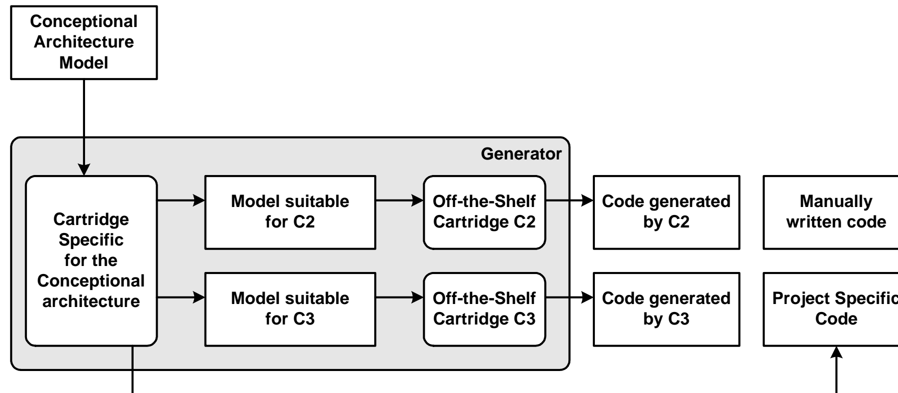
 © 2003 - 2006 Markus Völter

The Holy Grail: Cartridges

- Do you build **your own generator** for your specific architecture?
  - This is good, because it's tailored to *your* architecture
- Or do you want to (re)use **off-the-shelf cartridges** for certain standard technologies (such as J2EE, Hibernate, Spring)?
- You can do the **best of both worlds:**
  - Define applications using **your own metamodels** (architecture-centric, maybe functional ones on top) using Cascaded MDSD
  - **Transform** your models to input models for the off-the-shelf cartridges on the lower levels

 © 2003 - 2006 Markus Völter

The Holy Grail: Cartridges (II)



 © 2003 - 2006 Markus Völter

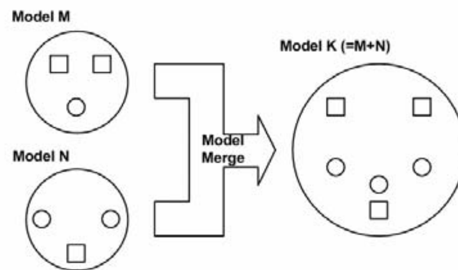
The Holy Grail: Cartridges (III)

- Cartridges are **not** just "3rd Party Generators downloadable from the Web".
- They need to be
  - Address only **one** aspect/concern/viewpoint/technology
  - **Minimally** redundant and maximally combinable
  - Cartridges must **expose the complete variability** of the technology/architecture they represent and not make any simplifying assumptions
    - These assumptions must be made by higher levels in the cascade – to cater for project/PLE specific aspects
  - The input for **cartridges is therefore usually created by a M2M transformation** from higher, more abstract and more all-encompassing levels
- Such systems are hardly available these days … hence the "Holy Grail" ☺

 © 2003 - 2006 Markus Völter

Model Merging

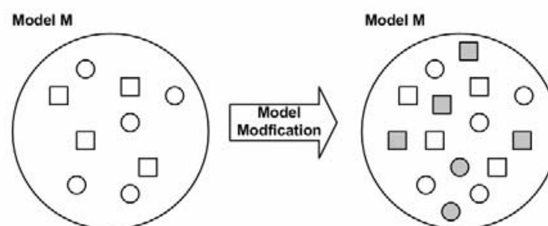
- Several models are **merged** with each other.



- Implications of model merging
  - Typically **easy to implement** (no actual transformation)
  - Meta models are obviously the same
  - Useful if models need to be **modularized** (team issues, performance, ...) and then put together for a complete build

Model Modifications

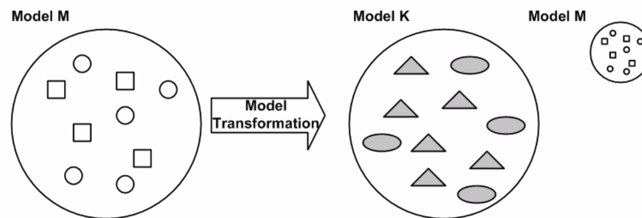
- An existing Model is **modified "in place".**



- Implications of model modification
  - An existing model is **enhanced at generation time**, adding elements (e.g. the index page of a web page)
  - The model is based on the same metamodel before and after the modification
  - little initial implementation overhead (e.g. using Java code)
  - tricky for bigger changes

Model Transformations

- A model is **transformed into another model**; the input model is left unchanged.

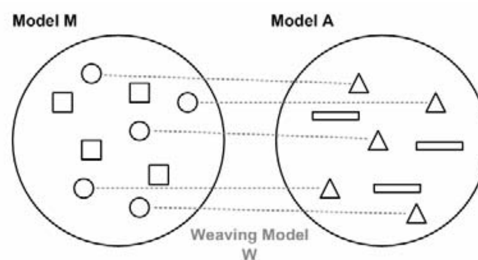


- Implications of model transformations
  - clean separation: **separate models, separate metamodels**
  - different domains can evolve independently
  - prerequisite for reusing "cartridges"
  - identical copy operations must be programmed explicitly
  - runtime and memory overhead

© 2003 - 2006 Markus Völter

Model Weaving/Linking

- Two (or more) models are **correlated with each other** in a well-defined way

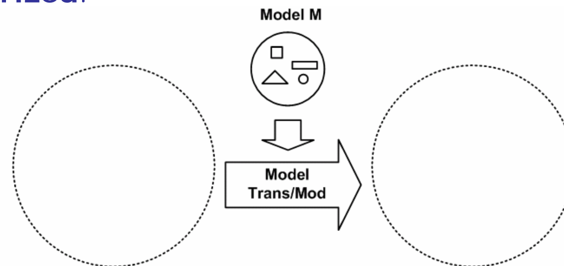


- Implications of model weaving
  - The **original models typically remain unchanged** (except for the weaving links)
  - A **weaving model (or spec) needs to be provided** – which is essentially a special kind of transformation rule
  - Most of today's M2M languages don't address this use case specifically – weavings must be programmed "imperatively" with the transformation language.

© 2003 - 2006 Markus Völter

Mixin Models

- The modification or transformation needs to be **parametrized**.



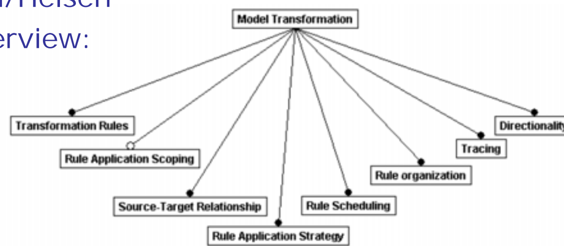
- Implications of mixin models
  - Aka Markup Models
  - Provide **additional (mark up) information** about how a given model should be processed in a modification of transformation
  - Obviously used **together with the other forms**

External Model Markings (AO-Modelling)

- In order to allow the transformation of a source model into a target model (or to generate code) it is sometimes necessary to **provide "support" information that is specific to the target meta model.**
  - Example: Entity Bean vs. Type Manager
- Adding these to the source model "**pollutes" the source model** with concepts specific to the target model.
- MDA proposes to add **"model markings",** but this currently supported well by only very few tools.
- Instead, we recommend keeping this information **outside of the model** (e.g. in an XML file); the transformation engine would use this auxiliary information when executing the transformations.

How to specify Transformations

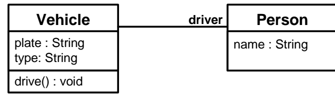
- There are various dimensions along we can distinguish M2M transformation approaches and languages:
  - **Conceptual Approach:** Procedural, Functional, Declarative/Relational
  - **Concrete Syntax:** Textual or Graphical
  - **Directionality:** Unidirectional vs. Bidirectional
  - Practical usability ☺
- A **paper** by Czarnecki/Helsen gives a very good overview: www.swen.uwaterloo.ca/~kczarnec/ECE750T7/czarnecki_helsen.pdf



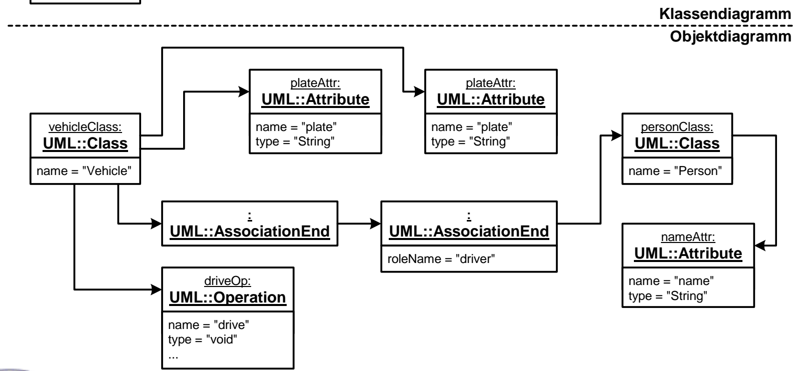
(Pseudo-)Java: Procedural/Textual/Unidirectional

```
function createEJBModel( Model source ): Model {
  Model target = new Model();
  foreach c:Class in source.classes {
    ImplementationClass implClass = new ImplementationClass();
    implClass.setName( c.getName()+"Bean" );
    target.addClass( implClass );
    Dependencies.define( implClass, c );
    RemoteInterface ri = new RemoteInterface ();
    // set name and add it to target model
    HomeInterface hi = new HomeInterface ();
    // set name and add it to target model
    foreach o:Operation in c.operations {
      ri.addOperation( new Operation( o.clone() ) );
      implClass.addOperation( new Operation( o.clone() ) );
    }
  }
  return target;
}
```

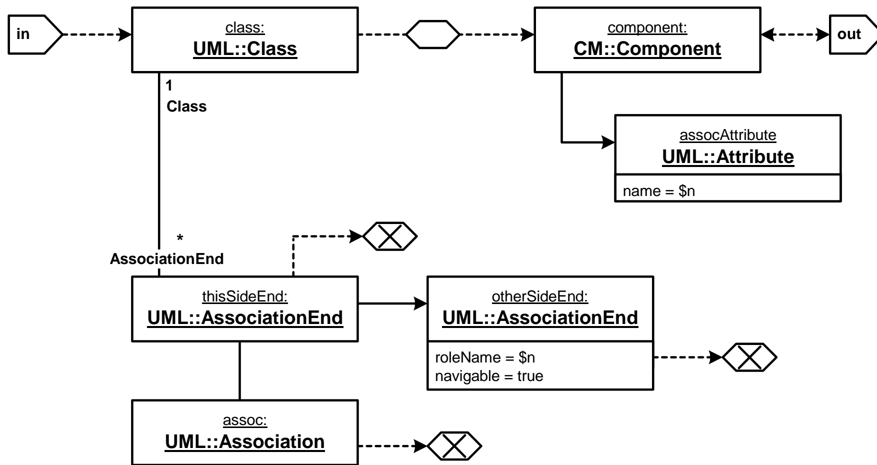
UMLX: Declarative/Graphical/Unidirectional II

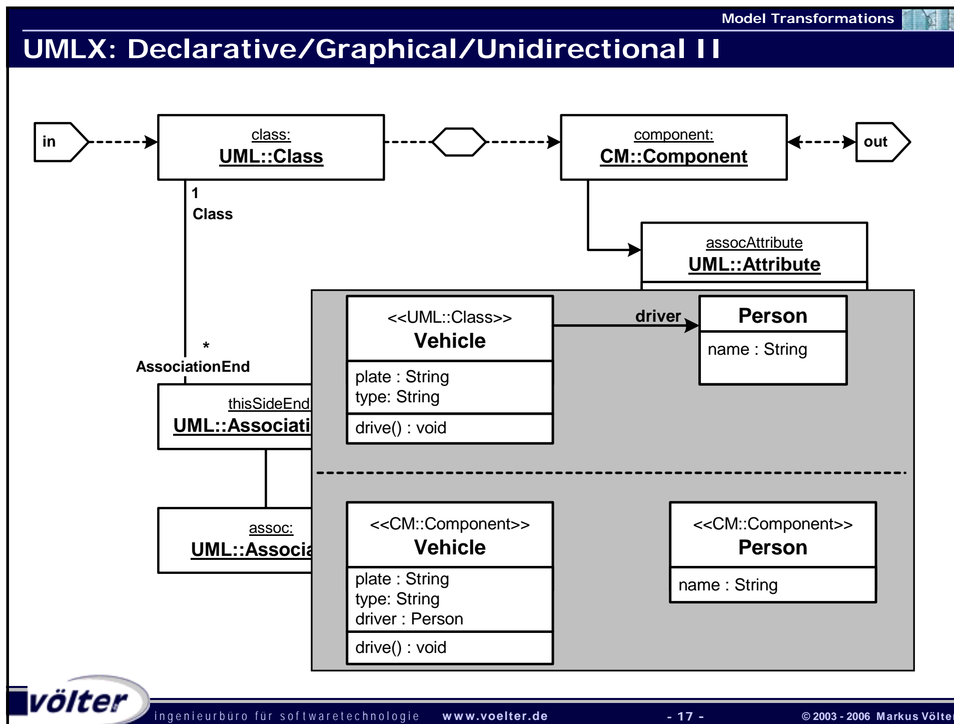


- **Precondition:**
  Representing a class diagram of metalevel n as an object diagram of metalevel n+1



UMLX: Declarative/Graphical/Unidirectional II





QVT Relational: Declarative/Textual/Bidirectional

```
top relation EntityKeyToTableKey {
  checkonly domain alma entity:Entity {
    key = entityKeyField:Field {}
  };
  enforce domain db table:Table {
    key = tableKey:Key {}
  };
  when {
    EntityToTable(entity, table
  }
  where {
    KeyRecordToKeyColumns(entit
  }
}
```

```
relation PhysicalQuantityTypeToColumn {
  pqName, pqUnit, fieldName : String;
  checkonly domain alma field:Field {
    name = fieldName,
    type = pq:PhysicalQuantityType {
      name = pqName,
      units = pqUnit
    }
  };
  enforce domain db table:Table {
    columns = column:Column {
      name = prefix + fieldName + '_as_' +
             pqName + '_in_' + pqUnit,
      type = AlmaPhysicalQuantityTypeToDbType(pq)
    }
  };
  primitive domain prefix:String;
}
```


© 2003 - 2006 Markus Völter

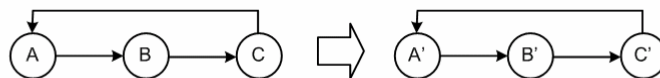
QVT Operational: Functional/Textual/Unidirectional

```
mapping DependentPart::part2table(in prefix : String) : Table
inherits fieldColumns {
  var dpTableName := prefix + recordName;
  name := dpTableName;
  columns := mainColumns +
             object Column {
               name := 'key_' + dpTableName;
               type := 'INTEGER';
               inKey := true;
             }
  end { self.parts->map part2columns(result, dpTableName + '_'); }
}
```

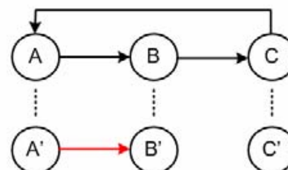
```
query PrimitiveType::convertPrimitiveType() : String =
  if self.name = "int" then 'INTEGER'
  else if self.name = "float" then 'FLOAT'
  else if self.name = "long" then 'BIGINT'
  else 'DOUBLE'
endif endif endif;
```


Keeping Track of the Current Transformation

- Consider trying to **duplicate** a model:



- **Problem:** When creating the connections, how do you know to which new element you had transformed the connection's source and target?



- Solution 1: **Keep track explicitly** in some kind of data structure (HashMap...)
- Solution 2: **Let the tool keep track** about these things: you can execute the mapping as often as you want, you'll always get the result of the first mapping

Traces, Impact Analysis and Partial Transformations

- Simple transformation tools will always transform **complete** source models into **complete** target models – partial transformations are not supported.
- This does not scale – especially in an interactive setting, where **various models need to be synchronized** in real time.
- To solve this problem, **trace information** can be used
  - A transformation automatically produces **a trace** as a side effect
  - If a subsequent change happens in the source model, the trace information can be used to **determine** which (partial) transformation must be reevaluated (Impact Analysis)
  - Currently, no transformation tools support this in a scalable way.

 © 2003 - 2006 Markus Völter

IDE Integration

- As usual, **IDE integration and tooling** is an important factor for any tool
  - Syntax highlighting
  - Metamodel awareness: Auto completion, type checker
  - Refactoring support
  - Debugger
  - Metamodels evolve: Transformations must be updated

 © 2003 - 2006 Markus Völter

A selection of existing implementations

- **3GL against Model** API (Java+EMF/MDR, C#+MDF)
  - straightforward for simple modifications
- **XSLT (BEWARE!)**
  - Works only XML-based models
  - severely limited scalability because there is no explicit metamodel
  - operations on XML level (often != meta model, very simple meta meta model, dynamically typed)
- **Eclipse GMT ATL**
  - Textual, used in practice, debugger
- **QVT**
  - standard of the OMG, implementations slowly appearing
- **Eclipse GMT oAW xTend**
  - Concise, practically proven, IDE support, no debugger (yet!)

ATL – Atlas Template Language

- ATL is a **subproject of Eclipse GMT**
  - Developed by INRIA in France
  - requires Eclipse as a runtime environment
  - Is based on the KM3 metamodel which can map to EMF, MDR and others
- **Scalability** as a primary focus
  - runs in its own virtual machine
  - strong support from both academia and industry
  - provides tool support
- combines **imperative and functional** aspects
  - conscious decision not to implement QVT

oAW / xTend

- openArchitectureware has a **lightweight transformation language**
  - Also part of Eclipse GMT
- Since release 4.1: xTend
  - integrated with the language for dynamic extension of the metamodel
  - very good tool support (debugger is in process)
  - open architecture: arbitrary metametamodels
  - Concise textual syntax

© 2003 - 2006 Markus Völter

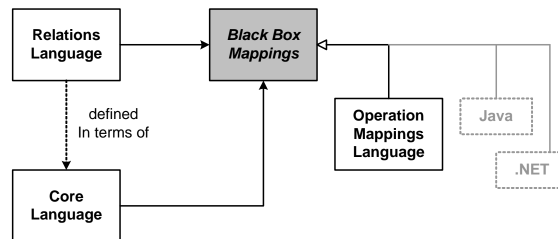
QVT – Query/View/Transformation

- QVT is the **OMG standard** for model transformations
  - CFP back in 2002 – very long process of standardisation
  - eight very different submissions
  - finalized this summer
- QVT is a standard, **not an implementation**
  - problem of interoperability
  - very wide scope
  - different interests of different groups, design by committee

© 2003 - 2006 Markus Völter

QVT – Query/View/Transformation II

- Since there's not much practical experience with M2M, **the OMG didn't really know what to standardize**.
- So they standardized a **bunch of languages** ☺
  - QVT Relational
  - QVT Operational
  - QVT Core
  - (+ Black box Mappings)

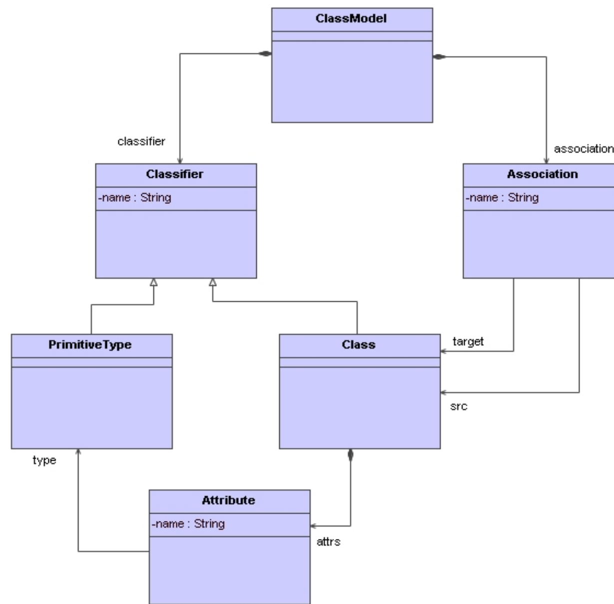


- The jury is still out wrt. the practical relevance of the various QVT languages.

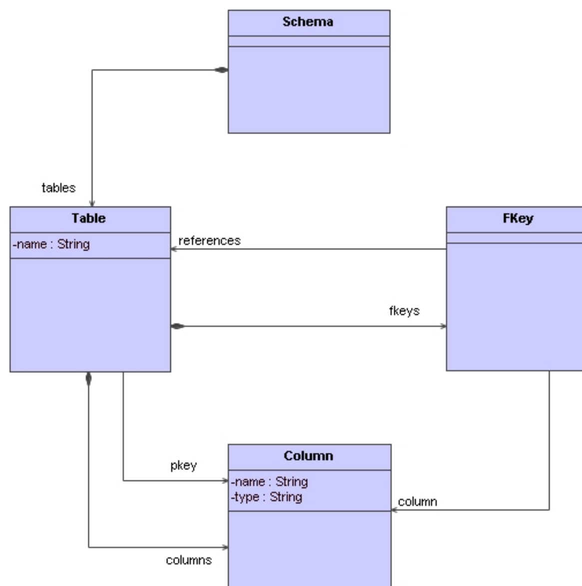
Model Transformations – a practical example

- **Example:** Simple Class Diagram -> Relational Database
- **Theory**
  - Looking at the two meta models
  - Thinking about the transformation in theory
- **Practice**
  - Doing it with QVT Relational (no implementation yet)
  - Doing it with QVT Operational (Together 2006)
  - Doing it with ATL (Eclipse/GMT/ATL)
  - Doing it with Xtend (oAW 4.1 - Eclipse/GMT/oAW)
- **Comparison**

The Simple Class Meta Model



The Relational Database Meta Model



Mapping

- In theory, the following **transformation steps** must be executed
  - ClassModel ⇒ Schema
  - Class ⇒ Table, Column (pkey)
  - Attribute ⇒ Column
  - Association ⇒ Foreignkey, Column
- In practice, ... let's see.

QVT Relations

- Bidirectional, declarative language
- The first line **declares a transformation** between two meta models (domains): *SimpleUML* and *SimpleRDBMS*

```
transformation umlRdbms (uml : SimpleUML,
                         rdbms : SimpleRDBMS) {
   relation PackageToSchema {
      domain uml p:Package {name=pn}
      domain rdbms s:Schema {name=pn}
   }
...
}
```

- The relation *PackageToSchema* describes how a *uml::Package* is mapped to an *rdbms::Schema*

QVT Relations II: Elements of a relation

```
relation ClassToTable {
   domain uml c:Class {
      namespace = p:Package {},
      kind = 'Persistent',
      name = cn
   }
   domain rdbms t:Table {
      schema = s:Schema {},
      name   = cn,
      column = cl:Column {
         name = cn+'_tid',
         type = 'NUMBER'},
      primaryKey = k:PrimaryKey {
         name = cn+'_pk',
         column = cl}
   } when {
      PackageToSchema(p, s);
   } where {
      AttributeToColumn(c, t);
   }
}
```

Object pattern / template for each domain

***when* clause** (Precondition)

***where* clause** (Postcondition, Implication)

 © 2003 - 2006 Markus Völter

QVT Relations III: Top and non-top relations

```
transformation umlRdbms (uml   : SimpleUML,
                         rdbms : SimpleRDBMS) {
   top relation PackageToSchema {…}
   top relation ClassToTable {…}
   relation AttributeToColumn {…}
}
```

- Execution of transformation requires that **all its top-level relations hold**
- Non-top relations are required to hold only if they are **invoked from the *where* clause** of another relation

 © 2003 - 2006 Markus Völter

QVT Relations IV: Check and Enforce

```
relation PackageToSchema {
    checkonly domain uml p:Package {name=pn}
    enforce domain rdbms s:Schema {name=pn}
}
```

- Domains can be *checkonly* or *enforce*.
  - **checkonly:** model cannot be changed
  - **enforce:** model will be modified
- In *enforce* domains, the transformation creates, deletes, and updates the elements in the target domain **until it corresponds to the source model and the given relations.**
- This allows for several **uses** of the engine:
  - **checkonly/checkonly:** diff engine, inconsistencies are reported
  - **checkonly/enforce:** forward transformation
  - **enforce/checkonly:** backward transformation

QVT Relations V: Keys

```
transformation umlRdbms (uml : SimpleUML,
                         rdbms : SimpleRDBMS) {
    key Table {schema, name};
    rule ...
```

- A relation may be **invoked in different where-clauses** (i.e. several times from within a transformation)
  - However, it is only executed once per key; upon subsequent invocation, the previous result is returned
- For **incremental transformations**, a certain model element might already exist but might require updating
- Here: we need to ensure that the same table is not created multiple times

QVT Relations VI: - Execution Scenarios

- **Check-only transformations** to verify that models are related in a specified way.
- Single direction transformations.
- Bi-directional transformations. (not supported in hybrid transformations)
- The ability to **establish relationships between pre-existing models**, whether developed manually, or though some other tool or mechanism.
- **Incremental updates** (in any direction) when one related model is changed after an initial execution.
- The ability to create as well as delete objects and values, while also being able to specify which objects and values must not be modified.

QVT Operational

```
transformation Simpleuml_To_Rdb;

metamodel 'http:///SimpleUML.ecore';
metamodel 'http:///rdb.ecore';

mapping main(in model: simpleuml::Model) : rdb::Model {
    name := model.name;
    schemas := package2schemas(model);
}
```

- Unidirectional, imperative language
- Top-level Constructs:
  - Mappings
  - Queries

- This example is based on the Together 2006 dialect of QVT Operational. It claims to be a QVT implementation, **but the language actually is different in many ways!**

QVT Operational II: Mappings

```
mapping package2schema(in pack: simpleuml::Package)
   : rdb::Schema when { pack.hasPersistentClasses() }
{
        name := pack.name;
        elements := pack.ownedElements->
           select(oclIsKindOf(simpleuml::Class))->
           collect(c |
              persistentClass2table(
                 c.oclAsType(simpleuml::Class)))->
                 asOrderedSet()
}
```

- **Implicit creation** of *rdb::Schema*
- *when* clause acts as a **guard** (mapping returns *null*, if guard evaluates to false)
- **Imperative** body (assignment statements)
- Non-declarative: Other mappings are explicitly called (`persistentClass2table(...)`)

QVT Operational III: Queries

```
query uml::Package::hasPersistentClasses() : Boolean {
   ownedElements->select(oclIsKindOf(uml::Class))->
      select(c |
      c.oclAsType(uml::Class).isPersistent())->
      size() > 0
}
```

- **Non-Invasive extensions** of existing meta classes
- **Functional** (body consists of one expression)
  - Expression language is OCL (as in QVT Relations)

QVT Operational IV: lateResolveByRule vs. Keys

```
mapping relationshipAttribute2foreignKey(
        in prop: uml::Property) :
        rdb::ForeignKey {
   name := 'FK'.concat(prop.name);
   reference := prop.type.
     lateResolveByRule('class2primaryKey',
                       rdb::PrimaryKey);
}
```

- Non-standard!
- Semantics not documented :-(
- In standard QVT it is done using **the key declaration** (a mapping is a relation with overwritten semantics)

```
key PrimaryKey {table};

mapping relationshipAttribute2foreignKey(
        in prop: uml::Property) :
        rdb::ForeignKey {...
```

© 2003 - 2006 Markus Völter

ATL (Atlas Transformation Language)

- Developed by INRIA Atlas Group
- Open Source @ Eclipse.org/GMT

```
module UML2RDBMS;
create OUT : SimpleRDBMS from IN : SimpleUML;

rule Package2Schema {
     from p : SimpleUML!Package
     to   s : SimpleRDBMS!Schema(
               name <- p.name,
               tables <- p.elements->
                 select(el|el.oclIsTypeOf(SimpleUML!Class))
             )
  }
```

- Statically typed (?)
- Unidirectional (*from -> to*)
- Based on OCL-like expression language
- Semi declarative

© 2003 - 2006 Markus Völter

ATL II: semi-declarative

```
rule Package2Schema {
    from p : SimpleUML!Package
    to   s : SimpleRDBMS!Schema(
               name<-p.name,
               tables<-p.elements->
                    select(e|e.oclIsTypeOf(SimpleUML!Class))
              )
}

rule Class2Table {
    from c : SimpleUML!Class
    to   t : SimpleRDBMS!Table(
              ...
}
```

- **Implicit invocation** of rules (something like type coercion in programming languages)

Kermeta

- Developed by IRISA (in Rennes, France)
- Open Source @ kermeta.org

```
class Class2RDBMS {
    reference class2table : Trace<Class, Table>
    reference fkeys : Collection<FKey>

    operation transform(inputModel : ClassModel) : RDBMSModel is do
        // Initialize the trace
        class2table := Trace<Class, Table>.new
        class2table.create
        fkeys := Set<FKey>.new
        result := RDBMSModel.new

        // Create tables
         getAllClasses(inputModel).select{ c | c.is_persistent }.each{ c |
            var table : Table init Table.new
            table.name := c.name
            class2table.storeTrace(c, table)
            result.table.add(table)
         }
        // Create columns
        getAllClasses(inputModel).select{ c | c.is_persistent }.each{ c |
            createColumns(class2table.getTargetElem(c), c)
        }
    end …
```

- **Imperative** (like Java with richer type system (EMOF), and closures)
- **No built in support for graph preservation** (A *Trace* class has to be used)

Kermeta II

```
class Main {
    operation main() : Void is do
        var inputModel : ClassModel init loadClassModel
        var tranfo : Class2RDBMS init Class2RDBMS.new
        var outputModel : RDBMSModel init
                          tranfo.transform(inputModel)

        var repository : EMFRepository init EMFRepository.new
        var resource : Resource init
          repository.createResource("../models/out.xmi",
                                "../metamodels/RDBMSMM.ecore")
       // Define a Root ...
       resource.instances.add(outputModel)

       resource.save()
    end
```

- Wants to be a **general purpose language** for processing models

 © 2003 - 2006 Markus Völter

Kermeta III

```
class ClassModel {
    attribute classifier : Classifier[0..*]
    attribute association : Association[0..*]

    operation classes() : Class[0..*] is do
       result := OrderedSet<Class>.new
       self.classifier.each{ c |
           if c.getMetaClass == Class then
               // cast the Classifier to Class
               var cls : Class cls ?= c
               result.add(cls)
           end
       }
    end
}
```

- No extension (or query) mechanism
- Operations can be implemented without escaping to a separate implementation language

 © 2003 - 2006 Markus Völter

Xtend

- Part of **openArchitectureWare**
- Open Source @ Eclipse.org/GMT

```
import ClassMM;
import RDBMSMM;

create Schema transform(ClassModel cm) :
  tables.addAll(cm.classifier.typeSelect(Class).table());
…
```

- **Functional** language (more or less ☺)
- **Statically typed**
- Dynamic polymorphism (multiple dispatch)
- Based on **OCL-like expression language** (with shorter syntax in most cases)
- Can work on many different meta-meta models (at the same time!)

 © 2003 - 2006 Markus Völter

Xtend II: Helpers/Extensions

```
associations(Class this) :
  eRootContainer.eAllContents.typeSelect(Association)
  .select(a|a.src==this);
```

- Return type is **derived** (can be specified optionally)
- Extensions can be invoked in member or function syntax
  `myClass.associations() == associations(myClass)`

 © 2003 - 2006 Markus Völter

Xtend III: *Create* extensions

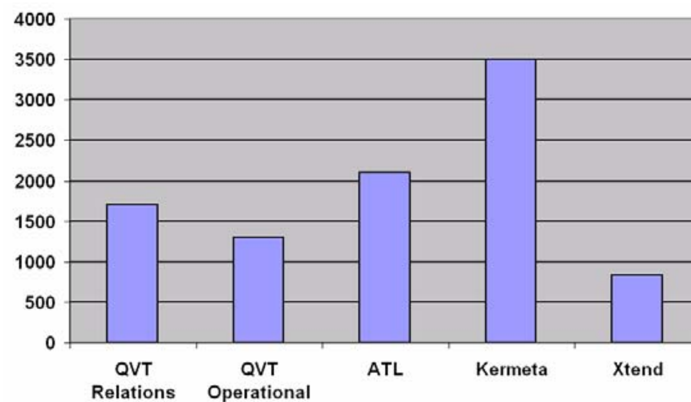
```
create Table table(Class c) :
   setName(c.name) ->
   pkey.add(c.primaryKey()) -> // both invocation
                               // of c.primaryKey()
   cols.add(c.primaryKey()) -> // return the *same* element
   cols.addAll(c.attrs.column()) ->
   cols.addAll(c.associations().fk_column())->
   fkeys.addAll(c.associations().fkey());

create Column primaryKey(Class c) :
   setName(c.name+"_ID")->
   setType("LONG");
```

- For each set of parameters a create extension is **executed only once**
  - subsequent invocations return the result of the original one
- **Automatic graph preservation**
- Recursion is handled implicitly and automatically

© 2003 - 2006 Markus Völter

Comparison: Amount of Code

- **Number of characters** needed to specify the transformation
  - may not be the only way to compare the languages, but certainly provides an indication.



© 2003 - 2006 Markus Völter

Comparison II: Query-by-Type

- Querying a typed **collection by type** (required regularly)
- OCL.dialect from Together 2006:

```
classifier->
        select(cls|cls.oclIsTypeOf(ClassMM::Class))->
        collect(cls|cls.oclAsType(ClassMM::Class))->asOrderedSet()
```

- Kermeta

```
result := OrderedSet<Class>.new
classifier.each{ c |
    if c.getMetaClass == Class then
        var cls : Class cls ?= c // cast the Classifier to Class
        result.add(cls)
    end}
```

- XTend:

```
classifier.typeSelect(Class)
```

Comparison III : Mechanisms to preserve graph structure

- **QVT dialect from Together**
  - Explicit call to untyped global functions (*lateResolve()*, etc.)
- **QVT standard**
  - Use of Keys
- **Xtend**
  - Implicit by use of *create* extensions
  - Parameter set constitutes the key
- **ATL**
  - Explicit call to global functions (resolveTemp())
- **Kermeta**
  - No support, manual work necessary

Conclusion : QVT (Relations)

- + Very powerful
- + OMG Standard
- - Complex
- - No implementations (therefore no tool support, yet)
- - Only for MOF based meta models
  (although we assume that implementations for "real world" meta meta models such as Ecore will become available)

Conclusion : QVT from Together 2006

- + Implementation available
- + tool support is ok
- - Not really QVT!
- - Works on generated and installed Ecore models, only
  (which makes incremental, iterative metamodel/transformation development a pain)

Conclusion : ATL

- + Implementation available
- + powerful
- + works on different meta meta models (e.g. Ecore, MDR, based on KM3 integration layer)
- - No standard
- - not statically typed?

© 2003 - 2006 Markus Völter

Conclusion : Kermeta

- + Implementation available
- + powerful
- + well thought out type system (real generics and closures)
- + good tool support
- - No standard
- - imperative and verbose syntax
- - no extension mechanism

© 2003 - 2006 Markus Völter

Conclusion: Xtend

- + automatic graph preservation
- + concise syntax
- + good tool support
- + works on many different meta meta models (ecore, uml2, java, etc.)
- - no standard

Conclusion

- Very short and (kind of) superficial walk-through
- Important aspects **we didn't talk about**
  - Available built-in types (Lists, Sets, Maps, Structs,...)
  - Black-box support (e.g. calling Java methods)
  - Runtime Footprint (e.g. Is execution from command line possible?)
  - Performance (compiled, interpreted)
  - Turn-around (compiled, interpreted)
  - Tool support in detail (code completion, debugger, refactoring, etc.)
  - Integration with other tools
  - ...