

Ingénierie des Modèles

Transformation de modèles

Eric Cariou

Master TIIL-A 2^{ème} année

Université de Bretagne Occidentale

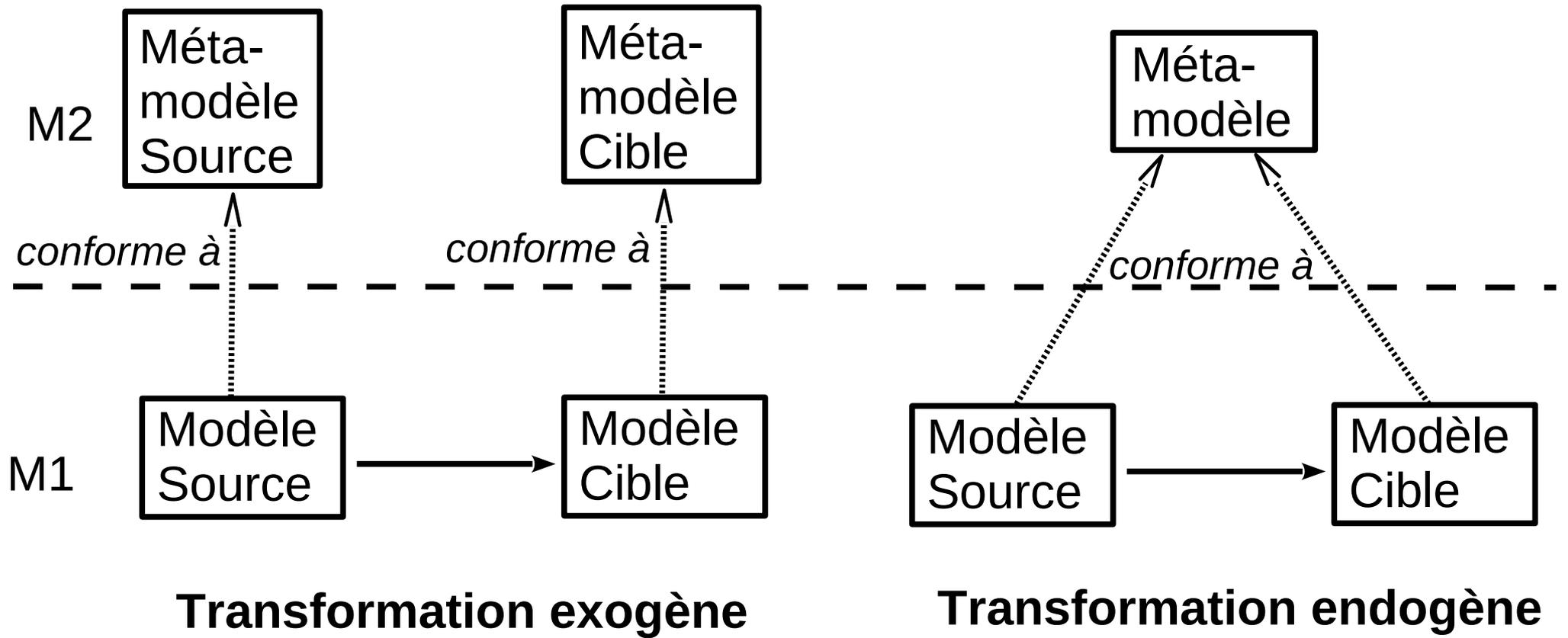
UFR Sciences & Techniques – Département Informatique

Eric.Cariou@univ-brest.fr

Transformations

- ◆ Une transformation est une opération qui
 - ◆ Prend en entrée des modèles (source) et fournit en sortie des modèles (cibles)
 - ◆ Généralement un seul modèle source et un seul modèle cible
- ◆ Transformation endogène
 - ◆ Modèles source et cible conformes au même méta-modèle
- ◆ Transformation exogène
 - ◆ Modèles source et cible conformes à des méta-modèles différents
- ◆ Transformations model to model (M2M) ou model to text (M2T)
- ◆ Exemples
 - ◆ M2T : d'un modèle UML vers un programme Java (génération de code)
 - ◆ M2M exogène : d'un diagramme UML vers un schéma de BDD
 - ◆ M2M endogène : d'un modèle UML vers un autre modèle UML

Transformations

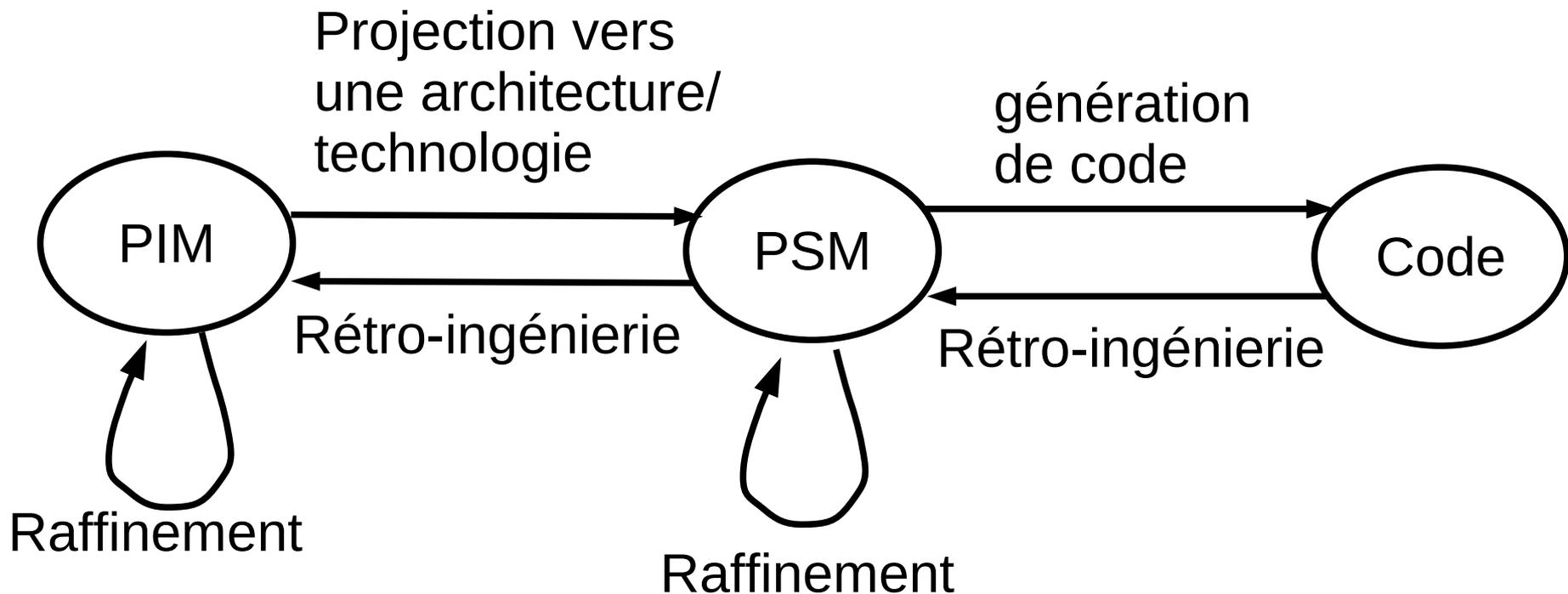


Model Driven Architecture

- ◆ Le MDA définit 2 principaux niveaux de modèles
 - ◆ PIM : Platform Independent Model
 - ◆ Modèle spécifiant une application indépendamment de la technologie de mise en oeuvre
 - ◆ Uniquement spécification de la partie métier d'une application
 - ◆ PSM : Platform Specific Model
 - ◆ Modèle spécifiant une application après projection sur une plate-forme technologique donnée

Model Driven Architecture

- ◆ Relation entre les niveaux de modèles



Niveaux de modèles

- ◆ Les niveaux PIM et PSM du MDA peuvent se généraliser dans tout espace technologique
- ◆ Modèles de niveau abstrait : indépendamment d'une plateforme de mise en œuvre, d'une technologie
- ◆ Modèles de niveau concret : par rapport à une plateforme, technologie de mise en œuvre
- ◆ Nécessité de modéliser une plateforme de mise en œuvre
 - ◆ PDM : Platform Deployment Model
 - ◆ Peu de choses sur ce sujet ...

Transformations en série

- ◆ Réalisation d'une application
 - ◆ Processus basé sur une série de transformations de modèles
- ◆ Exemple
 1. Modèle de l'application au niveau abstrait, avec un modèle de composant abstrait : modèle PIM
 2. Projection du modèle vers un modèle de composant EJB : modèle PSM
 3. Raffinement de ce modèle pour ajouter des détails d'implémentation : modèle PSM
 4. Génération du code de l'application modélisée vers la plateforme EJB

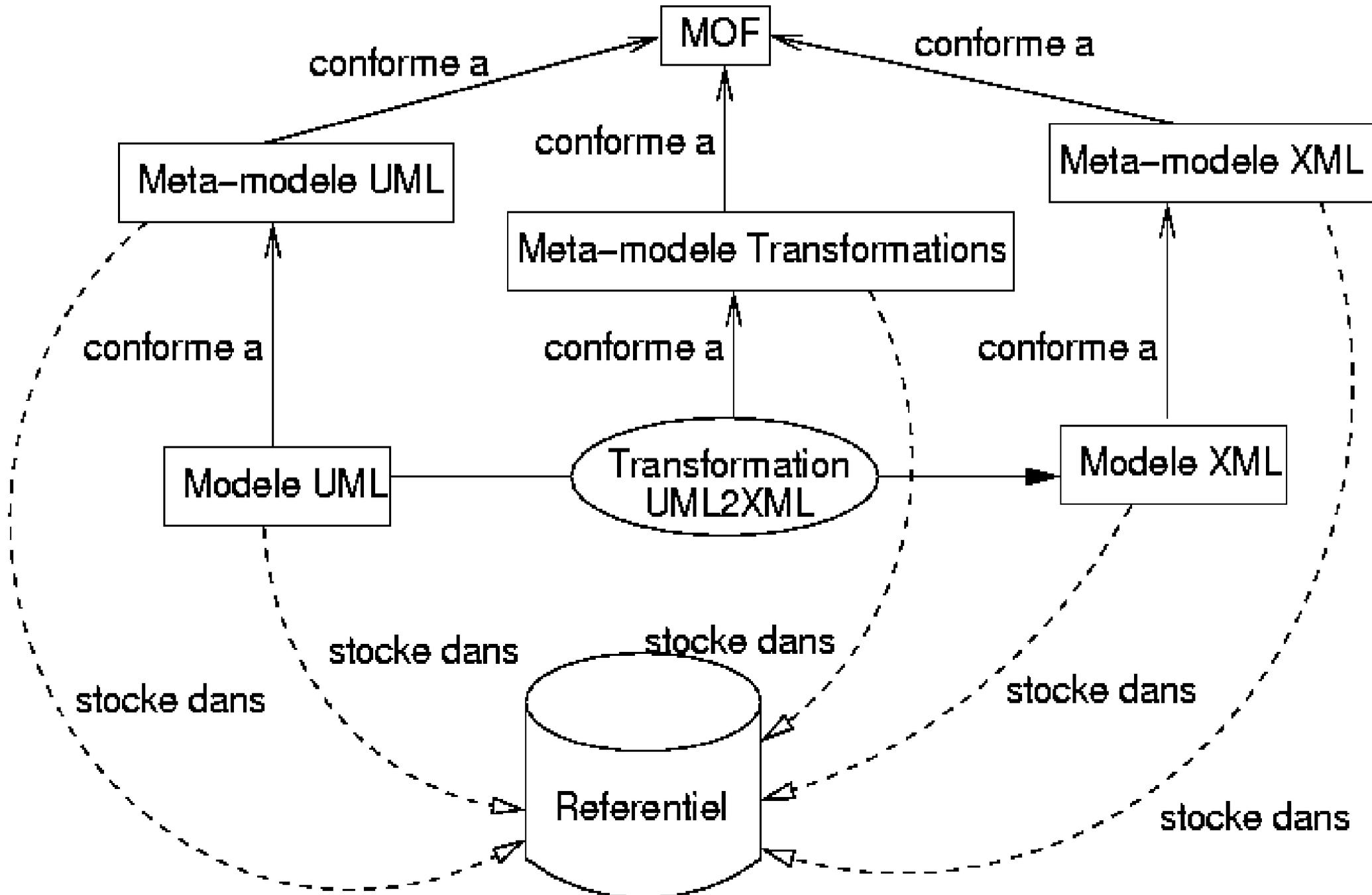
Autre vision des transformations

- ◆ Depuis longtemps on utilise des processus de développement automatisé et basé sur les transformations
 - ◆ Rien de totalement nouveau
 - ◆ Adaptation à un nouveau contexte
 - ◆ Exemple : compilation d'un programme C
 - ◆ Programme C : modèle abstrait
 - ◆ Transformation de ce programme sous une autre forme mais en restant à un niveau abstrait
 - ◆ Modélisation, représentation différente du programme C pour le manipuler : transformation en un modèle équivalent
 - ◆ Exemple : arbres décorés
 - ◆ Génération du code en langage machine
 - ◆ Avec optimisation pour une architecture de processeur donnée

Outils pour réaliser des transformations

- ◆ Outils de mise en œuvre
 - ◆ Exécution de transformations de modèles
 - ◆ Nécessité d'un langage de transformation
 - ◆ Qui pourra être défini via un méta-modèle de transformation
 - ◆ Les modèles doivent être manipulés, créés et enregistrés
 - ◆ Via un *repository* (dépôt, référentiel)
 - ◆ Doit pouvoir représenter la structure des modèles
 - ◆ Via des méta-modèles qui devront aussi être manipulés via les outils
 - ◆ On les stockera également dans un repository
- ◆ Il existe de nombreux outils ou qui sont en cours de développement (industriels et académiques)
 - ◆ Notamment plusieurs moteurs/langages de transformation

Modèles/méta-modèles/repository



Transformations : types d'outils

- ◆ Langage de programmation « standard »
 - ◆ Ex : Java
 - ◆ Pas forcément adapté pour tout
 - ◆ Sauf si interfaces spécifiques
 - ◆ Ex : JMI (Java Metadata Interface) ou framework Eclipse/EMF
- ◆ Langage dédié d'un atelier de génie logiciel
 - ◆ Ex : J dans Objecteering
 - ◆ Souvent propriétaire et inutilisable en dehors de l'AGL
- ◆ Langage lié à un domaine/espace technologique
 - ◆ Ex: XSLT dans le domaine XML, AWK pour fichiers texte ...
- ◆ Langage/outil dédié à la transformation de modèles
 - ◆ Ex : standard QVT de l'OMG, langage ATL
- ◆ Atelier de méta-modélisation avec langage d'action
 - ◆ Ex : Kermeta

Transformations : types d'outils

- ◆ 3 grandes familles de modèles et outils associés
 - ◆ Données sous forme de séquence
 - ◆ Ex : fichiers textes (AWK)
 - ◆ Données sous forme d'arbre
 - ◆ Ex : XML (XSLT)
 - ◆ Données sous forme de graphe
 - ◆ Ex : diagrammes UML
 - ◆ Outils
 - ◆ Transformateurs de graphes déjà existants
 - ◆ Nouveaux outils du MDE et des AGL (QVT, ATL, Kermeta ...)

Techniques de transformations

- ◆ 3 grandes catégories de techniques de transformation
 - ◆ Approche déclarative
 - ◆ Recherche de certains patrons (d'éléments et de leurs relations) dans le modèle source
 - ◆ Chaque patron trouvé est remplacé dans le modèle cible par une nouvelle structure d'élément
 - ◆ Écriture de la transformation « assez » simple mais ne permet pas toujours d'exprimer toutes les transformations facilement
 - ◆ Approche impérative
 - ◆ Proche des langages de programmation usuels
 - ◆ On parcourt le modèle source dans un certain ordre et on génère le modèle cible lors de ce parcours
 - ◆ Écriture transformation peut être plus lourde mais permet de toutes les définir, notamment les cas algorithmiquement complexes
 - ◆ Approche hybride : à la fois déclarative et impérative
 - ◆ La plupart des approches déclaratives offrent de l'impératif en complément car plus adapté dans certains cas

Exécution/spécification

- ◆ Problématiques d'exécution de transformations sont fondamentales
- ◆ Mais doit aussi être capable de spécifier des transformations
- ◆ Trois buts principaux
 - ◆ Spécification et documentation
 - ◆ Préciser ce que fait la transformation, dans quelles conditions on peut l'utiliser
 - ◆ Vérification, validation et test
 - ◆ S'assurer qu'un modèle peut bien être transformé ou bien est le résultat valide d'une transformation
 - ◆ Validation de l'enchaînement de transformation
 - ◆ Enchaînement de transformations est à la base de tout processus de développement basé sur le MDE

Spécification de transformation

- ◆ Mes travaux de recherche
 - ◆ Spécification de transformations via des contrats de transformations
 - ◆ Contrats : ensemble de contraintes sur un élément logiciel que s'engage à respecter l'élément (et l'utilisateur de l'élément)
 - ◆ Spécifie ce que ce fait l'élément sans détailler comment il fait (ce qui correspond au code)
 - ◆ Exemple du compte bancaire du cours sur OCL
 - ◆ **context** Compte : débiter(somme : int)
pre: somme > 0
post: solde = solde@pre - somme
 - ◆ L'opération débiter s'engage à respecter la post-condition si l'élément appelant l'opération respecte la pré-condition
 - ◆ Utilisation du langage OCL pour définir ces contrats

Exemple de transformation exogène :

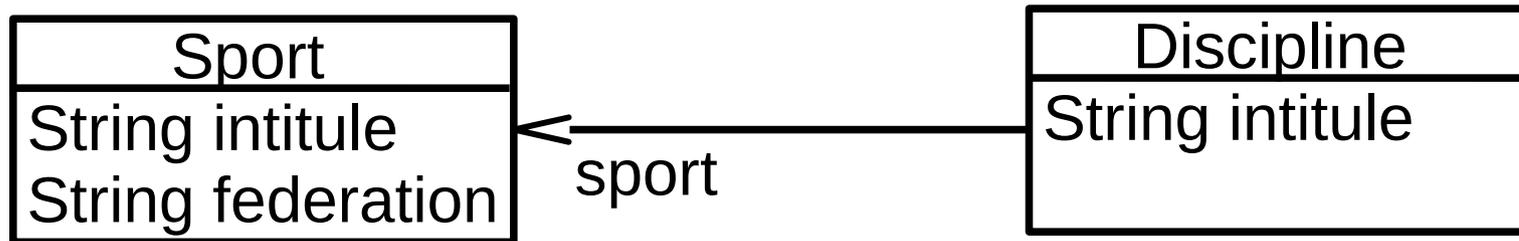
***d'un diagramme de classes
à un schéma de base de
données relationnelles***

Transformation exogène

- ◆ Deux méta-modèles (très simplifiés)
 - ◆ Diagramme de classes UML
 - ◆ Schéma de base de données relationnelles
 - ◆ Définis en Ecore
- ◆ Transformation
 - ◆ Chaque classe marquée comme persistante = une table
 - ◆ Un attribut d'une classe = une colonne d'une table
 - ◆ On rajoutera une colonne jouant le rôle de clé primaire et basée sur le nom des classes/tables
 - ◆ Une association entre classes = une jointure entre tables
 - ◆ Pas de prise en compte des cardinalités, pas de gestion des associations en * - * qui nécessitent une table de jointure
 - ◆ Définition de clés étrangères elles aussi colonnes dans les tables

UML vers SGBDR

- ◆ Exemple : le diagramme de classes



- ◆ Devient le schéma de BDD avec les tables
 - ◆ sport(number sport_tid, varchar intitule, varchar federation)
 - ◆ sport_tid : clé primaire
 - ◆ discipline(number discipline_tid, varchar intitule, number sport_tid)
 - ◆ discipline_tid : clé primaire
 - ◆ sport_tid : clé étrangère, jointure vers la table sport
- ◆ Trois implémentations
 - ◆ Partielle avec QVT : intérêt de la bidirectionnalité
 - ◆ Complète en Java/EMF : impératif
 - ◆ Complète en ATL : déclaratif

Query/View/Transformation

- ◆ Langage(s) de transformation et de manipulation de modèles normalisé par l'OMG
 - ◆ Query/View/Transformation ou QVT
 - ◆ Query : sélectionner des éléments sur un modèle
 - ◆ Le langage utilisé pour cela est OCL légèrement modifié et étendu
 - ◆ Avec une syntaxe différente et simplifiée
 - ◆ View : une vue est une sous-partie d'un modèle
 - ◆ Peut être définie via une query
 - ◆ Une vue est un modèle à part, avec éventuellement un méta-modèle restreint spécifique à cette vue
 - ◆ Transformation : transformation d'un modèle en un autre

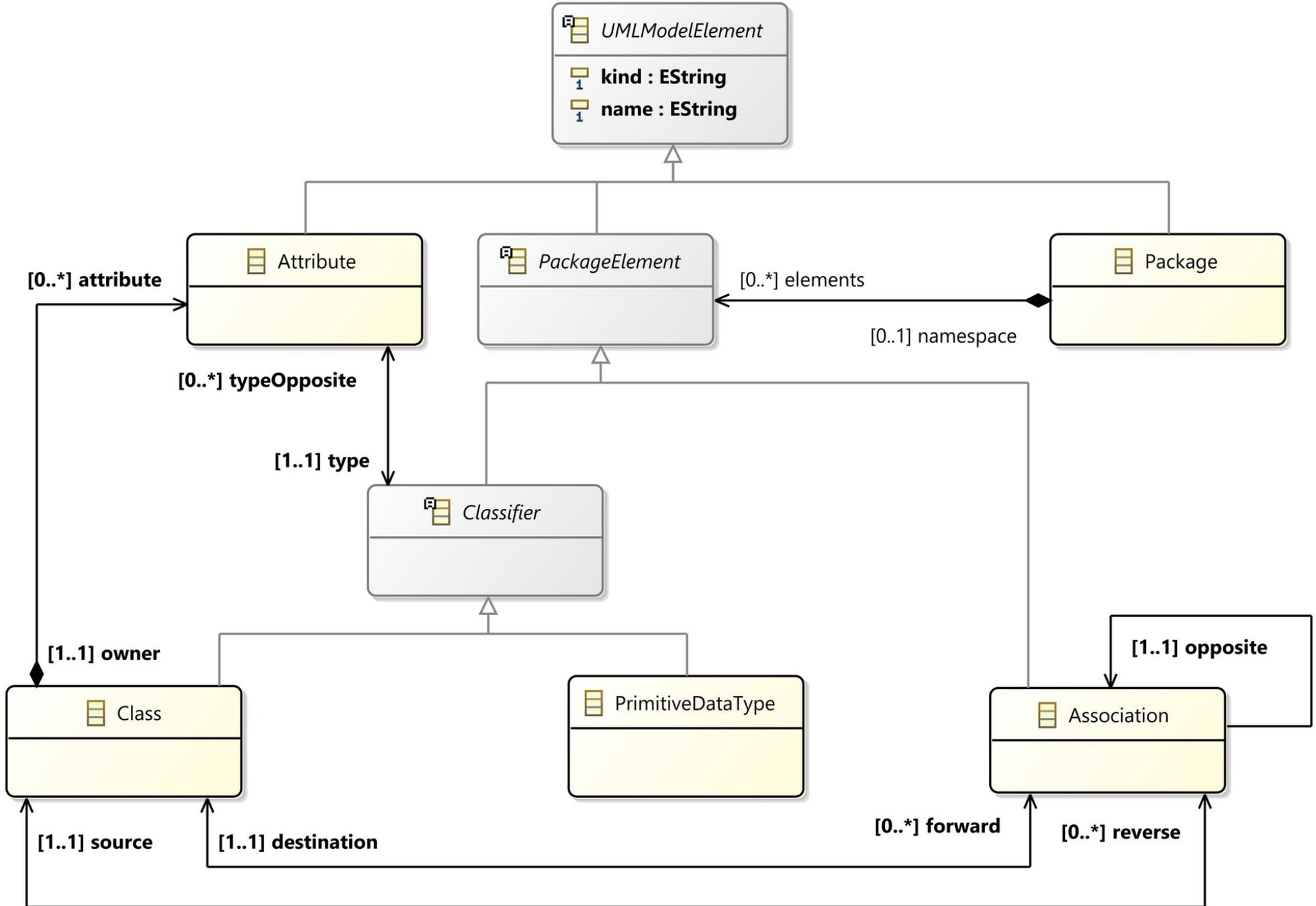
Langages de transformation dans QVT

- ◆ 3 langages/2 modes pour définir des transformations
 - ◆ Mode déclaratif
 - ◆ *Relation*
 - ◆ Correspondances entre des ensembles/patrons d'éléments de 2 modèles
 - ◆ Langage de haut niveau
 - ◆ *Core*
 - ◆ Plus bas niveau, langage plus simple
 - ◆ Mais avec même pouvoir d'expression de transformations que *relation*
 - ◆ Mode impératif
 - ◆ *Mapping*
 - ◆ Impératif, mise en oeuvre/raffinement d'une relation
 - ◆ Ajout de primitives déclaratives inspirées en partie d'OCL
 - ◆ Manipulation d'ensembles d'objets avec primitives à effet de bords
- ◆ Plusieurs syntaxes pour écriture de transformation selon les langages
 - ◆ Syntaxe textuelle
 - ◆ Syntaxe graphique

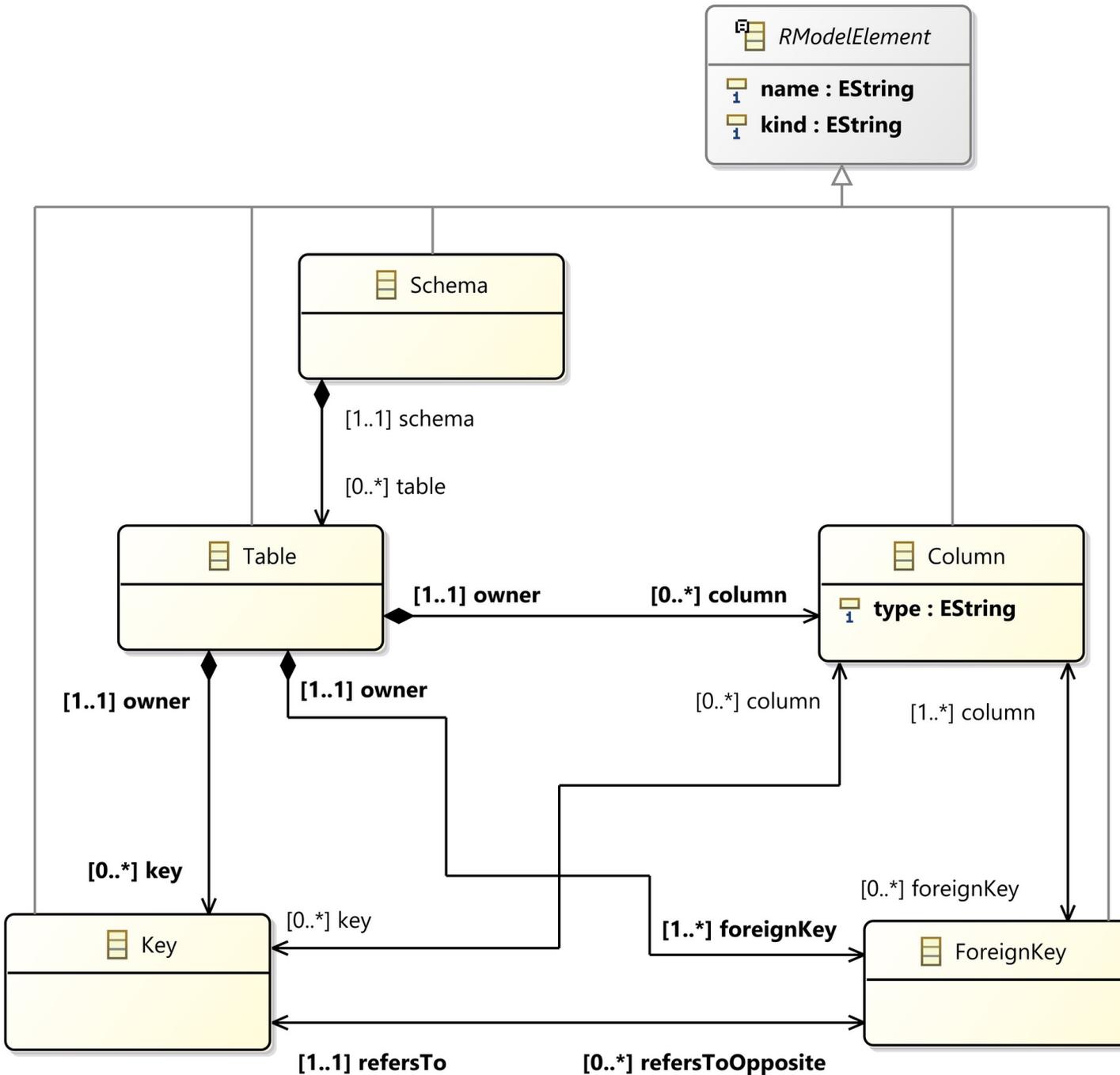
QVT : langage « relation »

- ◆ Une transformation est définie par un ensemble de relations
- ◆ Une relation fait intervenir 2 domaines
 - ◆ Domaine = ensemble d'éléments d'un modèle
 - ◆ Relation = contraintes sur dépendances entre éléments de 2 domaines
 - ◆ Domaine du modèle source
 - ◆ Domaine du modèle cible
- ◆ Une relation peut s'appliquer
 - ◆ Par principe quand on applique la transformation
 - ◆ En dépendance d'application d'une autre relation

Méta-modèle UML simplifié



Méta-modèle SGBDR simplifié



Exemple transformation

- ◆ Sur l'exemple précédent, voici les modèles
 - ◆ A gauche, le diagramme de classes
 - ◆ A droite, le schéma SQL correspondant

The screenshot shows the 'Sports.xmi' file in the IDE. The tree view displays the following structure:

- platform:/resource/BDD-CD/model/Sports.xmi
 - Package sports
 - Class Sport
 - Attribute intitule
 - Attribute Federation
 - Class Discipline
 - Attribute intitule
 - Primitive Data Type Integer
 - Primitive Data Type String
 - Association sport

The 'Association sport' element is selected. The Properties window below shows the following details:

Property	Value
Destination	Class Sport
Kind	
Name	sport
Namespace	Package sports
Opposite	
Source	Class Discipline

The screenshot shows the 'BDD.xmi' file in the IDE. The tree view displays the following structure:

- platform:/resource/BDD-CD/model/BDD.xmi
 - Schema sports
 - Table Sport
 - Column Sport_tid
 - Column intitule
 - Column Federation
 - Key Sport_pk
 - Table Discipline
 - Column Discipline_tid
 - Column intitule
 - Column Sport_tid
 - Key Discipline_pk
 - Foreign Key Sport_fk

The 'Foreign Key Sport_fk' element is selected. The Properties window below shows the following details:

Property	Value
Column	Column Sport_tid
Kind	
Name	Sport_fk
Owner	Table Discipline
Refers To	Key Sport_pk

Exemple : UML vers SGBDR en QVT

- ◆ But de la transformation
 - ◆ Modèle de données en UML vers équivalent schéma de données relationnel et inversement
 - ◆ transformation `umlToRdbms` (`uml : SimpleUML, rdbms : SimpleRDBMS`) {
 top relation PackageToSchema {...}
 top relation ClassToTable {...}
 relation AttributeToColumn {...}
 ...
}
- ◆ Etapes de la transformation
 - ◆ Chaque package UML correspond à un schéma de BDD, chaque classe persistante à une table, chaque attribut de classe à une colonne de table ...
- ◆ Relations marquées avec « top »
 - ◆ S'appliquent par principe
 - ◆ Les autres s'appliquent si dépendantes d'autres relations

Exemple : UML vers SGBDR en QVT

- ◆ top relation PackageToSchema

```
{
  domain uml p:Package {name=pn}
  domain rdbms s:Schema {name=pn}
}
```
- ◆ Pour chaque package UML, on a un schéma de données portant le même nom
 - ◆ Les attributs `name` correspondent à la même variable `pn`

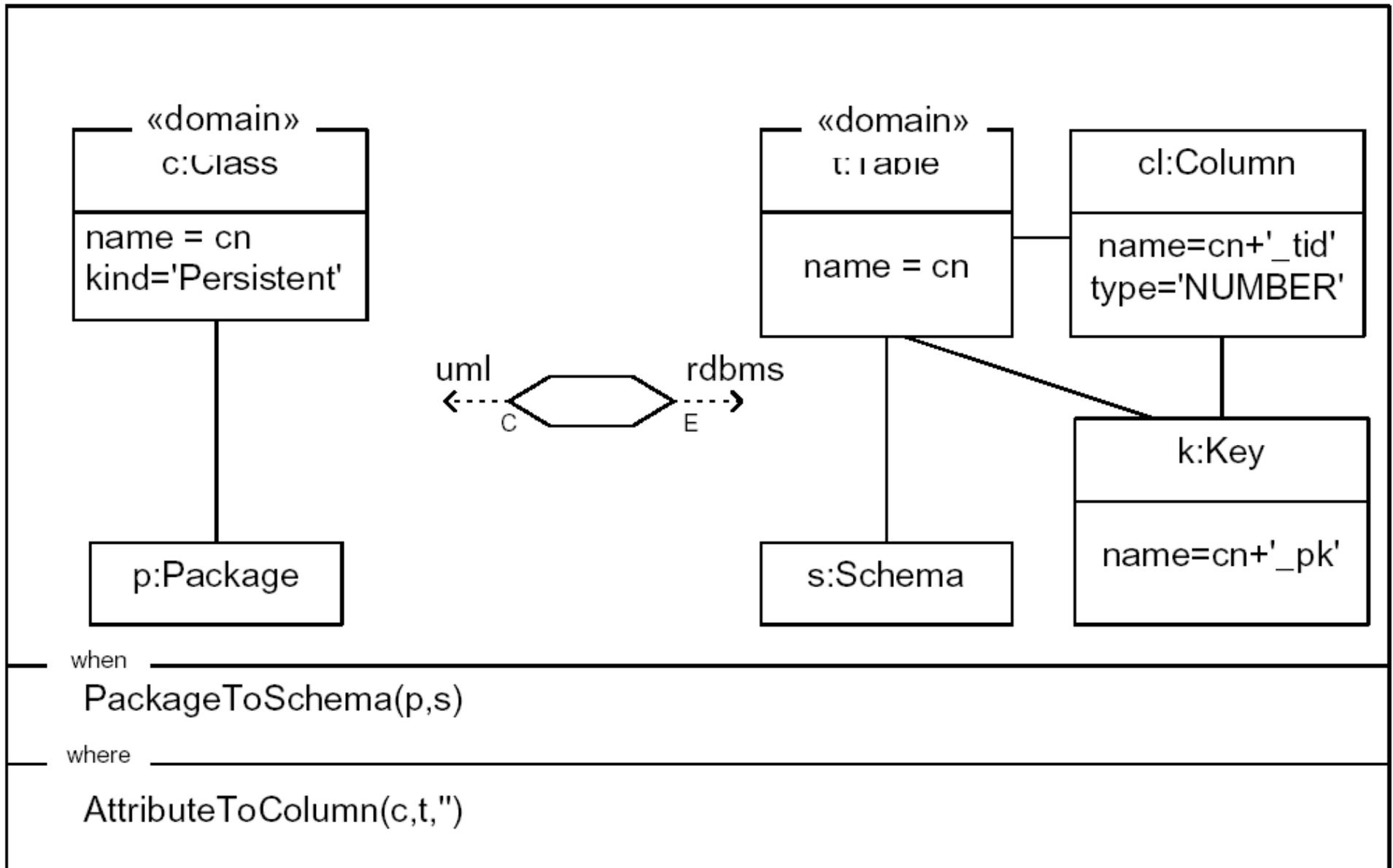
Exemple : UML vers SGBDR en QVT

```
top 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);
  }
}
```

- ◆ Pour chaque classe persistante
- ◆ On a une table avec
 - ◆ Le même nom
 - ◆ Une colonne pour l'identifiant avec nom formé à partir du nom de la classe
 - ◆ Une clé primaire avec nom formé à partir du nom de la classe
- ◆ Dépendances avec autres relations
 - ◆ « ClasseToTable » est appliquée quand on applique « PackageToSchema »
 - ◆ Et il faut appliquer aussi « AttributeToColumn » pour la classe et la table

Exemple : UML vers SGBDR en QVT

- ◆ Relation « ClassToTable », syntaxe graphique



Types de relations entre modèles

- ◆ Transformations définies par des relations
 - ◆ Correspondances/dépendances entre 2 modèles dans un sens comme dans l'autre
 - ◆ Spécification est bi-directionnelle par défaut
 - ◆ A l'exécution, on choisit une direction de transformation
 - ◆ Exécution est mono-directionnelle
 - ◆ Modèle cible peut exister ou pas à l'exécution
 - ◆ Sera alors complété, modifié ou créé selon les cas
- ◆ Possibilité de spécialiser une transformation/relation
 - ◆ Juste vérifier si le modèle cible est cohérent rapport au modèle source (checkonly)
 - ◆ Imposer que le modèle cible soit cohérent rapport au modèle source (enforced)

Types de relations entre modèles

◆ Exemple avec transformation UML/SGDBR

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

◆ Exécution dans le sens UML vers SGBDR

◆ Source = uml, cible = rdbms

◆ Le modèle cible comportera strictement un schéma pour chaque package du modèle source

◆ Création des schémas manquants

◆ Suppression des schémas existants mais ne correspondant pas

◆ Exécution dans le sens SGBDR vers UML

◆ Source = rdbms, cible = uml

◆ Vérifie seulement, en précisant les erreurs le cas échéant, que chaque schéma du modèle source correspond à un package du modèle cible (aucune modification du modèle cible)

Exemple : UML vers SGBDR en Java

```
public Schema toSGBD(SimpleUML.Package pack) {
    // création du schéma
    Schema schema = SimpleRDBMSFactory.eINSTANCE.createSchema();
    schema.setName(pack.getName());
    // parcours de l'ensemble des classes persistantes
    for (PackageElement elt : pack.getElements()) {
        if (elt instanceof SimpleUML.Class) {
            SimpleUML.Class cl = (SimpleUML.Class)elt;
            if (cl.getKind().equals("Persistent")) {
                // création de la table
                Table table = SimpleRDBMSFactory.eINSTANCE.createTable();
                table.setName(cl.getName());
                schema.getTable().add(table);
                // création de la colonne et de la clé primaire
                Column col = SimpleRDBMSFactory.eINSTANCE.createColumn();
                col.setName(cl.getName()+"_tid");
                col.setType("NUMBER");
                Key key = SimpleRDBMSFactory.eINSTANCE.createKey();
                key.setName(cl.getName()+"_pk");
                key.getColumn().add(col);
                table.getKey().add(key);
                table.getColumn().add(col);
            }
        }
    }
}
```

Exemple : UML vers SGBDR en Java

```
// ajout des colonnes de la table à partir des attributs
for (Attribute att : cl.getAttribute()) {
    col = SimpleRDBMSFactory.eINSTANCE.createColumn();
    col.setName(att.getName());
    if (att.getType().getName().equals("Integer"))
        col.setType("NUMBER");
    if (att.getType().getName().equals("Boolean"))
        col.setType("BOOLEAN");
    if (att.getType().getName().equals("String"))
        col.setType("VARCHAR");
    table.getColumn().add(col);
}
```

```
}
```

```
}
```

```
}
```

```
}
```

Exemple : UML vers SGBDR en Java

```
// parcours des associations pour créer les clés étrangères
for (PackageElement elt : pack.getElements()) {
    if (elt instanceof Association) {
        Association asso = (Association)elt;
        if (asso.getDestination().getKind().equals("Persistent")) {
            SimpleUML.Class source = asso.getSource();
            Table table = this.getTableByName(schema, source.getName());
            Column col = SimpleRDBMSFactory.eINSTANCE.createColumn();
            col.setName(asso.getDestination().getName()+"_tid");
            col.setType("NUMBER");
            ForeignKey key = SimpleRDBMSFactory.eINSTANCE.createForeignKey();
            key.setName(asso.getDestination().getName()+"_fk");
            key.setRefersTo(this.getTableByName(schema,
                asso.getDestination().getName()).getKey().get(0));
            key.getColumn().add(col);
            table.getColumn().add(col);
            table.getForeignKey().add(key);
        }
    }
}
return schema;
}
```

Exemple : UML vers SGBDR en Java

```
public Table getTableByName(Schema schema, String name) {
    Optional<Table> table = schema.getTable().stream().filter(t ->
        t.getName().equals(name)).findAny();
    if (table.isPresent()) return table.get();
    else return null;
}
```

◆ Analyse du code

- ◆ Assez verbeux mais pas de complexité particulière
 - ◆ On parcourt les classes persistantes, on crée les tables au fur et à mesure avec les colonnes correspondant aux attributs et la clé primaire
- ◆ Pour les associations, il faut juste retrouver avec la méthode `getTableByName` la table qui a le même nom que la classe pour référencer sa clé primaire
- ◆ Attention : bien faire la première boucle en entier avant la seconde
 - ◆ Doit être sûr que toutes les tables sont créées avant de traiter les associations qui vont utiliser ces tables
 - ◆ Le parcours explicite du modèle est un point important dans les transformations impératives

Langage de transformation ATL

- ◆ Langage déclaratif utilisant OCL
 - ◆ On déclare des règles qui associent un élément du modèle source à un ou plusieurs éléments générés dans le modèle cible
- ◆ Champ from
 - ◆ Sélectionne un élément du source par son type
 - ◆ Et un éventuel filtre écrit en OCL
- ◆ Champ to
 - ◆ Définit un ou plusieurs éléments du cible
 - ◆ Pour chacun, précise les valeurs de ses attributs et références
 - ◆ Avec l'affectation : <-
- ◆ Champ optionnel to
 - ◆ Partie impérative d'une transformation
- ◆ Peut écrire des fonctions utilitaires en OCL (helpers)

Exemple : UML vers SGBDR en ATL

```
-- nom de la transformation
module UMLtoBDD;

-- BDD : nom du méta-modèle coté SGBDR
-- CD : nom du méta-modèle coté UML
create OUT : BDD from IN : CD;

-- helper qui renvoie le type SQL du type UML d'un attribut
helper context CD!Attribute def: getSQLType() : String =
  if self.type.name = 'Integer' then 'NUMBER'
  else if self.type.name = 'Boolean' then 'BOOLEAN'
  else if self.type.name = 'String' then 'VARCHAR'
  else self.type.name
  endif endif endif;

-- règle qui pour chaque package du modèle source,
-- crée un schéma dans le modèle cible en positionnant son attribut name
rule SchemaToPackage {
from
  pack : CD!Package
to
  schema : BDD!Schema (
    name <- pack.name
  )
}
```

Exemple : UML vers SGBDR en ATL

```
-- pour chaque classe persistante, on crée une table
-- avec une clé primaire et sa colonne
rule ClassToTable {
from
  class : CD!Class (class.kind = 'Persistent')
to
  table : BDD!Table (
    name <- class.name,
    schema <- class.namespace
  ),
  columnKey : BDD!Column (
    name <- class.name + '_tid',
    type <- 'NUMBER',
    owner <- table
  ),
  primaryKey : BDD!Key (
    name <- class.name + '_pk',
    column <- Set { columnKey },
    owner <- table
  )
}
```

Exemple : UML vers SGBDR en ATL

```
-- Pour chaque attribut d'une classe, on crée une colonne
-- dans la table correspondant.
-- On filtre pour ne garder que les attributs des classes
-- persistantes sinon on créerait aussi des colonnes pour
-- des classes non persistantes.
rule AttributeToColumn {
from
  att : CD!Attribute (att.owner.kind = 'Persistent')
to
  column : BDD!Column (
    name <- att.name,
    owner <- att.owner,
    -- appel du helper défini dans le contexte de Attribute
    type <- att.getSQLType()
  )
}
```

Exemple : UML vers SGBDR en ATL

```
-- pour chaque association pointant vers une classe persistante,  
-- on crée la clé étrangère avec sa colonne et pointant vers la  
-- clé primaire de l'autre table  
rule AssociationToForeignKey {  
from  
  asso : CD!Association (asso.destination.kind = 'Persistent')  
to  
  columnKey : BDD!Column (  
    name <- asso.destination.name + '_tid',  
    type <- 'NUMBER',  
    owner <- asso.source  
  ),  
  foreignKey : BDD!ForeignKey (  
    name <- asso.destination.name + '_fk',  
    column <- Set { columnKey },  
    owner <- asso.source,  
    -- récupère la table pointée par la destination avec une expression  
    -- OCL puis affecte refersTo à la clé primaire de cette table  
    refersTo <- (BDD!Table.allInstances() -> any ( t |  
      t.name = asso.destination.name)).key -> first()  
  )  
}
```

Exemple : UML vers SGBDR en ATL

- ◆ Analyse du code
 - ◆ Beaucoup moins verbeux que la version Java
 - ◆ Tous les attributs/références sont publics, pas besoin de getters et de setters
 - ◆ Parcours du modèle réalisé par le moteur ATL
 - ◆ Applique les règles jusqu'à avoir traité tous les éléments
- ◆ Dans la règle `AttributeToColumn` pour la génération d'un élément de type `Column`, on a
 - ◆ `owner <- att.owner`
 - ◆ Littéralement : le propriétaire de la colonne est le propriétaire de l'attribut
 - ◆ C'est une classe du modèle source, ça ne semble pas avoir de sens
 - ◆ En pratique, le moteur ATL affecte comme propriétaire de la colonne, l'élément coté cible qui a été généré (dans une autre règle) à partir du propriétaire de l'attribut soit la table associée à cette classe

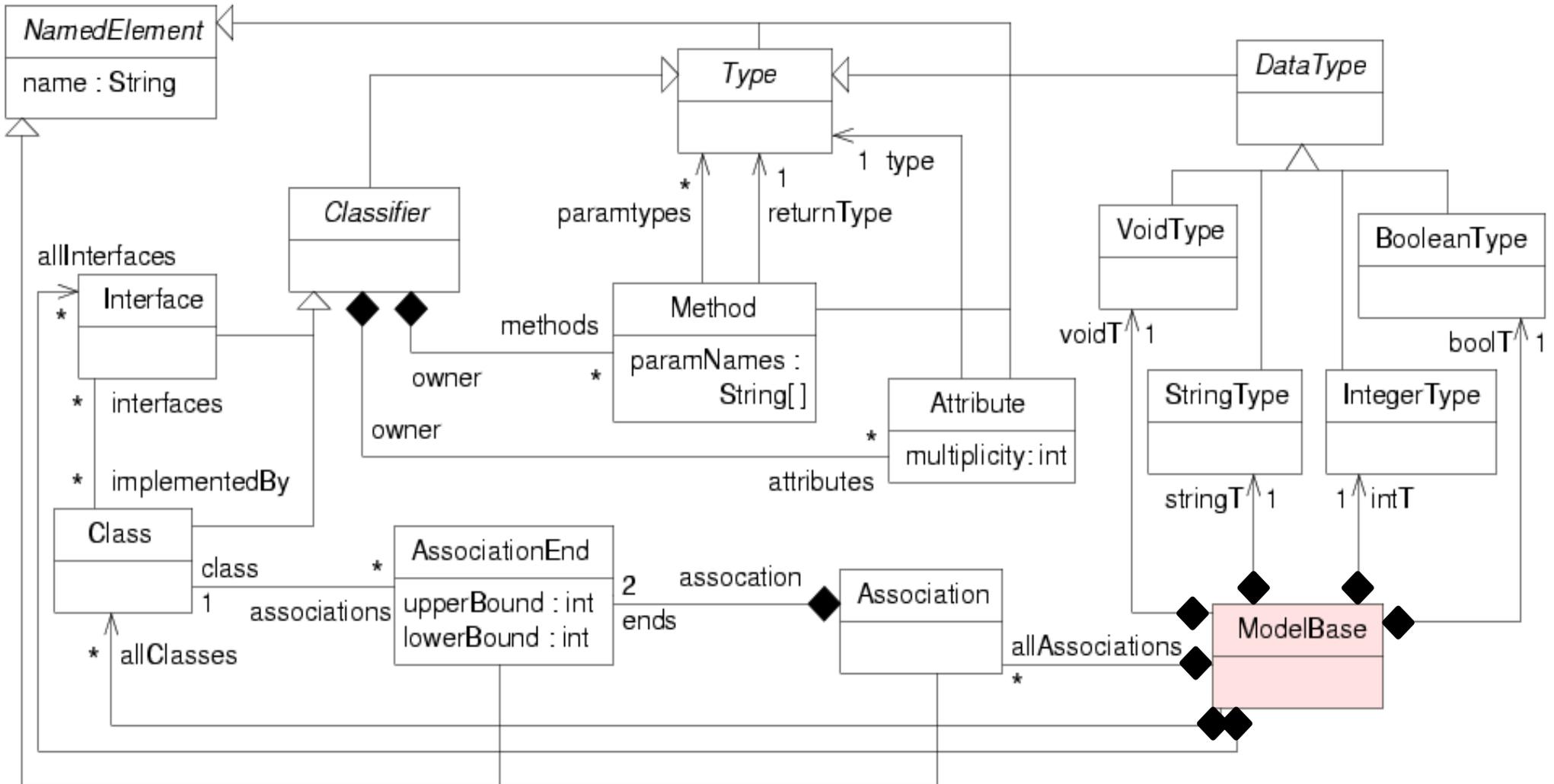
Exemple de transformation endogène :

***privatisation des attributs dans
un diagramme de classes***

Privatisation d'attributs

- ◆ Pour un diagramme de classe, passer les attributs d'une classe en privé et leur associer un « getter » et un « setter »
- ◆ On va traiter en entier cet exemple
 - ◆ Transformation en Java/EMF, style impératif
 - ◆ Transformation en ATL, style déclaratif
- ◆ Utilisation d'un méta-modèle simplifié de diagramme de classes
 - ◆ Méta-modèle UML trop complexe pour cet exemple
 - ◆ Mais plus complexe que pour l'exemple précédent : il manquait la définition des méthodes des classes

MM simplifié de diagramme de classes



- ◆ Note: les visibilités ne sont pas définies, on ne les gèrera pas pendant la transformation

MM simplifié de diagramme de classes

◆ Contraintes OCL pour compléter le méta-modèle

◆ Unicité des noms de type

```
context Type inv uniqueTypeNames:  
Type.allInstances() -> forall (t1, t2 |  
    t1 <> t2 implies t1.name <> t2.name)
```

◆ Une interface n'a pas d'attributs

```
context Interface inv noAttributesInInterface:  
attributes -> isEmpty()
```

◆ Une méthode à une liste de noms de paramètres et une liste de types de paramètres de même taille

```
context Method inv sameSizeParamsAndTypes:  
paramNames -> size() = paramTypes -> size()
```

◆ ...

Règles générales de la transformation

- ◆ Pour un attribut *att* de type *type*, la forme des accesseurs est
 - ◆ Getter : *type getAtt()*
 - ◆ Setter : *void setAtt(type xxx)*
 - ◆ Le nom de l'attribut est quelconque
- ◆ Règles de transformations
 - ◆ Pour chaque attribut de chaque classe
 - ◆ On ajoute, s'ils n'existaient pas déjà, un setter et un getter dans la classe qui possède l'attribut
 - ◆ Doit donc prévoir des fonctions de vérification de la présence d'un getter ou d'un setter

Transformation en Java/EMF

- ◆ Vérification si une méthode est un getter d'un attribut
- ◆

```
public boolean isGetter(Attribute att, Method met) {  
    if (met.getName().length() <= 4) return false;  
    String nomCherche =  
        "get"+att.getName().substring(0,1).toUpperCase()  
        +att.getName().substring(1, att.getName().length());  
    if (! met.getName().equals(nomCherche)) return false;  
    if (met.getParamTypes().size() != 0) return false;  
    if (met.getReturnType() != att.getType()) return false;  
    return true;  
}
```
- ◆ Nom « getAtt » (vérifie taille chaine > 3 sinon les substring plantent)
- ◆ Même type de retour que l'attribut
- ◆ Liste de paramètres vide

Transformation en Java/EMF

- ◆ Vérification si une méthode est un setter d'un attribut
- ◆

```
public boolean isSetter(Attribute att, Method met) {  
    if (met.getName().length() <= 4) return false;  
    String nomCherche =  
        "set"+att.getName().substring(0,1).toUpperCase()  
        +att.getName().substring(1, att.getName().length());  
    if (! met.getName().equals(nomCherche)) return false;  
    if (met.getParamTypes().size() != 1 ) return false;  
    if (met.getParamTypes().get(0) != att.getType()) return false;  
    if (met.getReturnType() != this.voidRef) return false;  
    return true;  
}
```
- ◆ Nom « setAtt »
- ◆ Un seul paramètre, du même type que l'attribut
 - ◆ Le nom du paramètre n'est pas important
- ◆ Type de retour est « void »
 - ◆ On aura dans la classe qui contient le code, ce type référencé dans l'attribut voidRef

Transformation en Java/EMF

- ◆ Méthodes pour créer un getter et un setter à un attribut
- ◆ Instancie la méthode, positionne son nom, son type de retour, l'éventuel paramètre pour un setter et son propriétaire est le propriétaire de l'attribut
- ◆

```
public void addGetter(Attribute att) {
    Method met = ClassDiagramFactory.eINSTANCE.createMethod();
    met.setName("get"+att.getName().substring(0,1).toUpperCase()
        +att.getName().substring(1, att.getName().length()));
    met.setReturnType(att.getType());
    met.setOwner(att.getOwner());
}

public void addSetter(Attribute att) {
    Method met = ClassDiagramFactory.eINSTANCE.createMethod();
    met.setName("set"+att.getName().substring(0,1).toUpperCase()
        +att.getName().substring(1, att.getName().length()));
    met.getParamTypes().add(att.getType());
    met.getParamNames().add("value");
    met.setReturnType(this.voidRef);
    met.setOwner(att.getOwner());
}
```

Transformation en Java/EMF

```
◆ public void addAccessors(ModelBase model) {  
    for (ClassDiagram.Class cl : model.getAllClasses()) {  
        for (Attribute att : cl.getAttributes()) {  
            if (cl.getMethods().stream().noneMatch(m -> isGetter(att,m)))  
                addGetter(att);  
            if (cl.getMethods().stream().noneMatch(m -> isSetter(att,m)))  
                addSetter(att);  
        }  
    }  
}
```

- ◆ La transformation implémente une double boucle
 - ◆ Traite chaque attribut de chaque classe
 - ◆ Si ne trouve pas dans les méthodes de la classe un getter ou un setter correspondant à l'attribut, on crée la méthode
- ◆ Le modèle en paramètre est directement modifié
 - ◆ Ce qui simplifie l'implémentation, on garde tout le reste du modèle à l'identique

Transformation en ATL

- ◆ Pour vérification des présences des getter et setter, on utilisera des helpers écrits en OCL

- ◆ **helper context** CD!Attribute **def:** hasGetter() : Boolean =
self.owner.methods -> exists (m |
 m.name = 'get' + self.name.firstToUpper() **and**
 m.paramNames -> isEmpty() **and**
 m.paramTypes -> isEmpty() **and**
 m.returnType = self.type
);

- ◆ **helper context** CD!Attribute **def:** hasSetter() : Boolean =
self.owner.methods -> exists (m |
 m.name = 'set' + self.name.firstToUpper() **and**
 m.paramNames -> size() = 1 **and**
 m.paramTypes -> includes(self.type) **and**
 m.returnType = thisModule.voidType
);

- ◆ Fonction pour gérer le premier caractère d'une chaîne en majuscule

```
helper context String def: firstToUpper() : String =  
self.substring(1, 1).toUpperCase() + self.substring(2, self.size());
```

- ◆ Référence sur le type void

```
helper def: voidType : CD!VoidType =  
CD!VoidType.allInstances() -> asSequence() -> first();
```

Transformation en ATL

- ◆ Point fondamental en ATL pour toute transformation
 - ◆ Un élément du source n'est traité que par au plus une règle
 - ◆ On ne peut donc pas avoir ici : une règle qui recopie l'attribut, une règle qui crée éventuellement le getter si l'attribut n'en a pas, une règle qui crée éventuellement le setter si l'attribut n'en a pas
 - ◆ Il y aurait alors plusieurs règles qui pourraient s'appliquer au même attribut
- ◆ Transformation ATL en mode raffinement
 - ◆ Uniquement dans le contexte d'une transformation endogène
 - ◆ Si un élément est référencé dans une règle, y compris indirectement et qu'il n'y a pas de règle explicite le concernant
 - ◆ Il est dupliqué automatiquement à l'identique dans le modèle cible
 - ◆ Évite d'avoir à écrire des règles pour recopier des éléments non modifiés par la transformation

Transformation en ATL

- ◆ Cinq règles pour notre transformation
 - ◆ Création d'une base de modèle identique
 - ◆ Le raffinement fait que toutes les classes, interfaces et associations de la base seront automatiquement dupliquées avec leur contenu
 - ◆ Pour chaque attribut, on a des règles qui créent l'attribut coté cible et les éventuels méthodes accesseurs manquantes
 - ◆ Selon qu'il possède déjà un getter ou un setter, 4 cas différents
 - ◆ Possède un getter et un setter (règle « hasAll »)
 - ◆ Possède un setter mais pas un getter (règle « hasSetter »)
 - ◆ Possède un getter mais pas un setter (règle « hasGetter »)
 - ◆ Ne possède ni l'un ni l'autre (règle « hasNothing »)

Transformation en ATL

```
◆ module AddAccessorRefining;  
create cible : CD refining source : CD;
```

```
... liste des helpers ...
```

```
rule duplicateModelBase {  
from  
  sourceBase : CD!ModelBase  
to  
  cibleBase : CD!ModelBase (  
    allClasses <- sourceBase.allClasses,  
    allInterfaces <- sourceBase.allInterfaces,  
    allAssociations <- sourceBase.allAssociations,  
    voidT <- sourceBase.voidT,  
    intT <- sourceBase.intT,  
    stringT <- sourceBase.stringT,  
    boolT <- sourceBase.boolT )  
}
```

```
rule attributeHasAll {  
from  
  attSource : CD!Attribute (  
    attSource.hasSetter() and attSource.hasGetter())  
to  
  attTarget : CD!Attribute (  
    name <- attSource.name,  
    owner <- attSource.owner,  
    type <- attSource.type,  
    multiplicity <- attSource.multiplicity )  
}
```

Transformation en ATL

- ◆ **rule** attributeHasSetter {
from
 attSource : CD!Attribute (
 attSource.hasSetter() **and not**(attSource.hasGetter())
)
to
 attTarget : CD!Attribute (
 name <- attSource.name,
 owner <- attSource.owner,
 type <- attSource.type,
 multiplicity <- attSource.multiplicity
),
 getter : CD!Method (
 name <- 'get' + attSource.name.firstToUpper(),
 returnType <- attTarget.type,
 owner <- attTarget.owner
)
}
- ◆ Pour un attribut du source qui a un setter mais pas de getter, 2 éléments sont créés coté cible
 - ◆ L'attribut équivalent
 - ◆ La méthode getter associée

Transformation en ATL

```
◆ rule attributeHasGetter {  
  from  
    attSource : CD!Attribute (  
      not(attSource.hasSetter()) and attSource.hasGetter()  
    )  
  to  
    attTarget : CD!Attribute (  
      name <- attSource.name,  
      owner <- attSource.owner,  
      type <- attSource.type,  
      multiplicity <- attSource.multiplicity  
    ),  
    setter : CD!Method (  
      name <- 'set' + attSource.name.firstToUpper(),  
      returnType <- thisModule.voidType,  
      owner <- attTarget.owner,  
      paramNames <- Set { 'value' },  
      paramTypes <- Set { attTarget.type }  
    )  
}
```

Transformation en ATL

```
◆ rule attributeHasNothing {  
  from  
    attSource : CD!Attribute (  
      not(attSource.hasSetter()) and not(attSource.hasGetter())  
    )  
  to  
    attTarget : CD!Attribute (  
      name <- attSource.name,  
      owner <- attSource.owner,  
      type <- attSource.type,  
      multiplicity <- attSource.multiplicity  
    ),  
    setter : CD!Method (  
      name <- 'set' + attSource.name.firstToUpper(),  
      returnType <- thisModule.voidType,  
      owner <- attTarget.owner,  
      paramNames <- Set { 'value' },  
      paramTypes <- Set { attTarget.type }  
    ),  
    getter : CD!Method (  
      name <- 'get' + attSource.name.firstToUpper(),  
      returnType <- attTarget.type,  
      owner <- attTarget.owner  
    )  
}
```