Model Driven Architecture Transformation : Model To Model & Model To Text

Zineb BOUGROUN

Plan

- Introduction
- Model to model : QVT
- Model to text : acceleo

Introduction

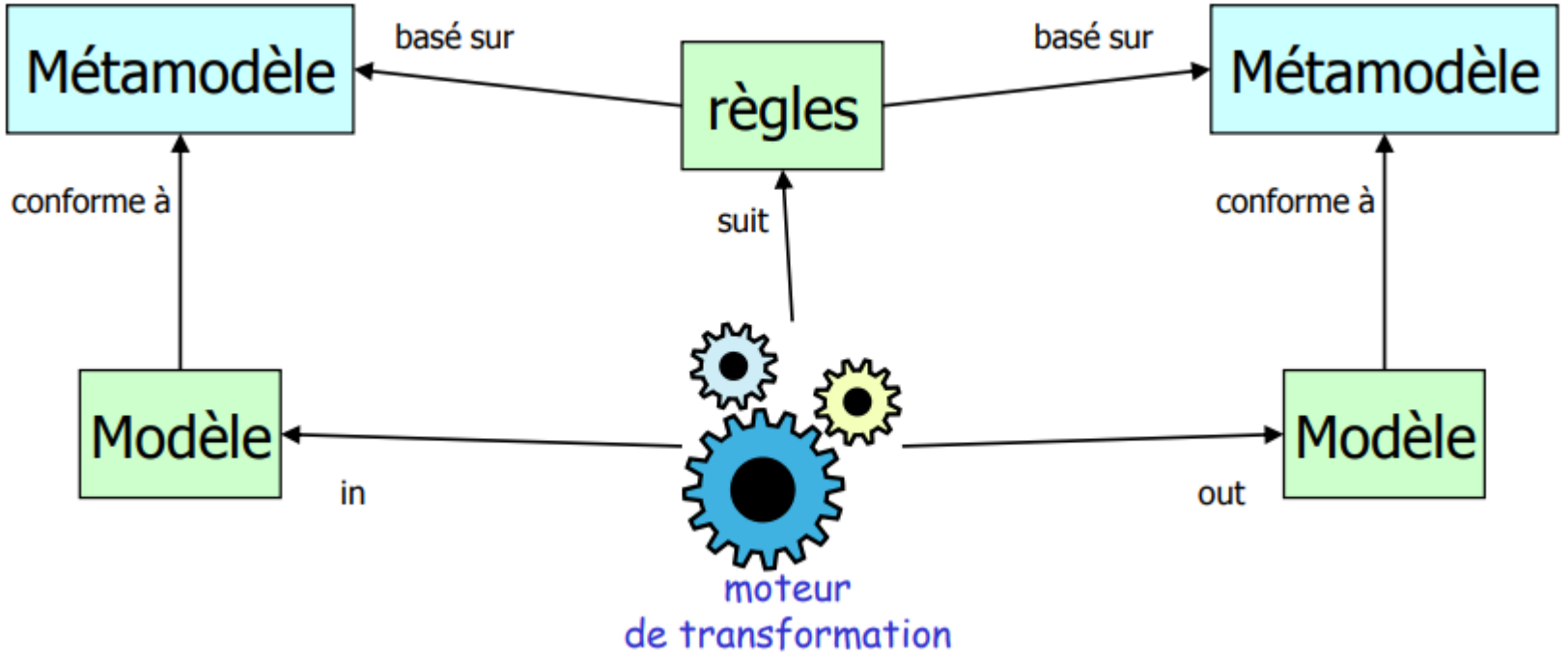
Définitions

- Les transformations de modèles sont le cœur des aspects de production de MDA
  - CIM vers PIM, PIM vers PSM, PSM vers code (sens inverse).
- Deux types de transformation
  - Model to model
  - Model to text
- Transformer un **modèle** source vers un **modèle** cible en se basant sur des règles de transformations.
- Une transformation de **modèles** permet de passer d'un **modèle** source décrit à un certain niveau d'abstraction à un modèle destination décrit éventuellement à un autre niveau d'abstraction.
- Une règle de transformation de **modèle** définit la manière dont un ensemble de concepts du méta modèle source est transformé en un ensemble de concepts du méta modèle destination.
  - Indépendance des règles vis-à-vis des instances du modèle

Model to Model

Transformation de modèle

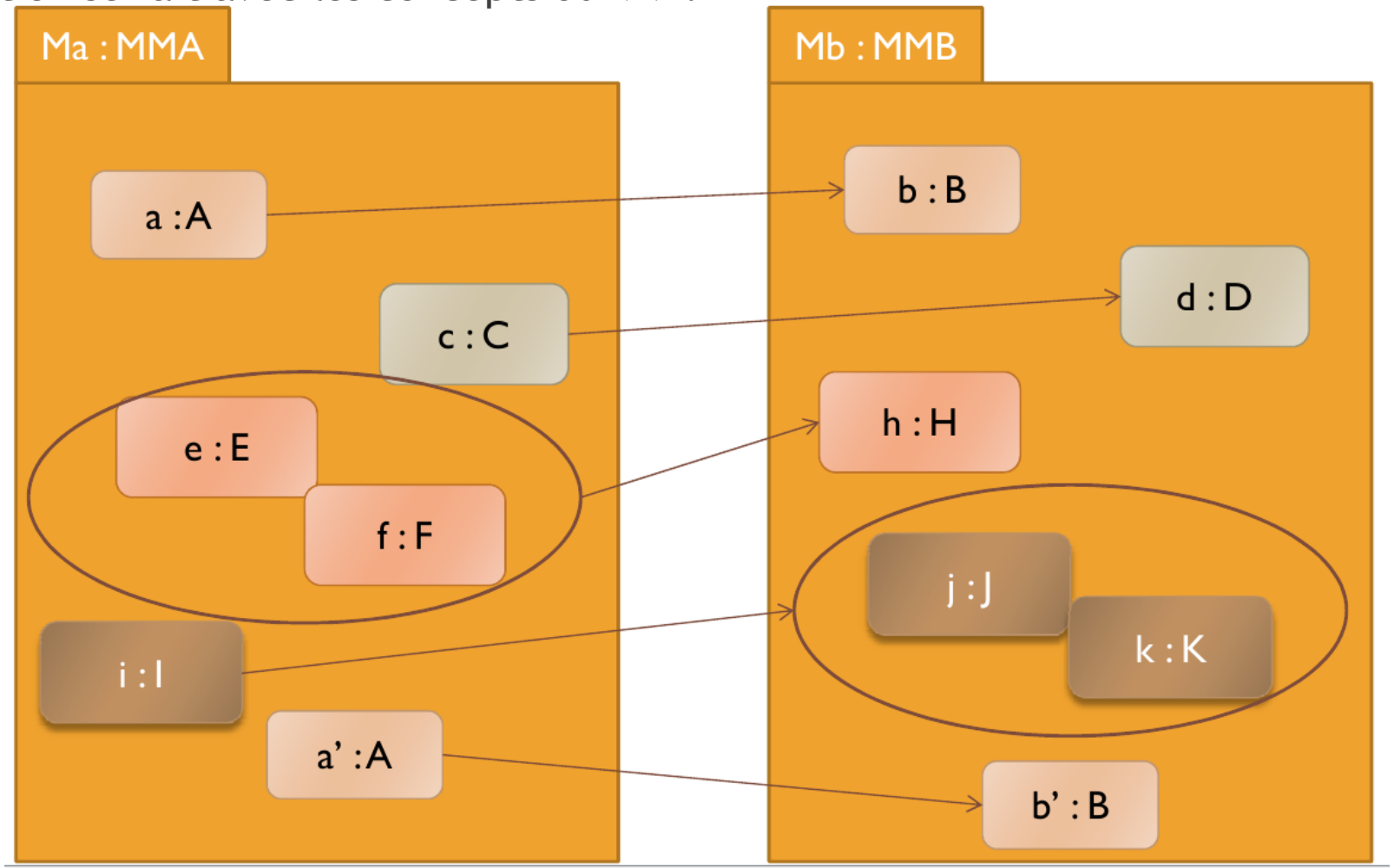
- Permet de transformer [1..*] modèles sources en [1..*] modèles cibles
  - Chaque modèle est conforme à un métamodèle
- agit sur les modèles
- ce conçoit sur les métamodèles !



Model to Model

Règle de transformation de modèle

- Une règle décrit comment transformer un concept (ou *) source vers un concept (ou *) cible
  - La description se fait avec les concepts du MM !



Model to Model

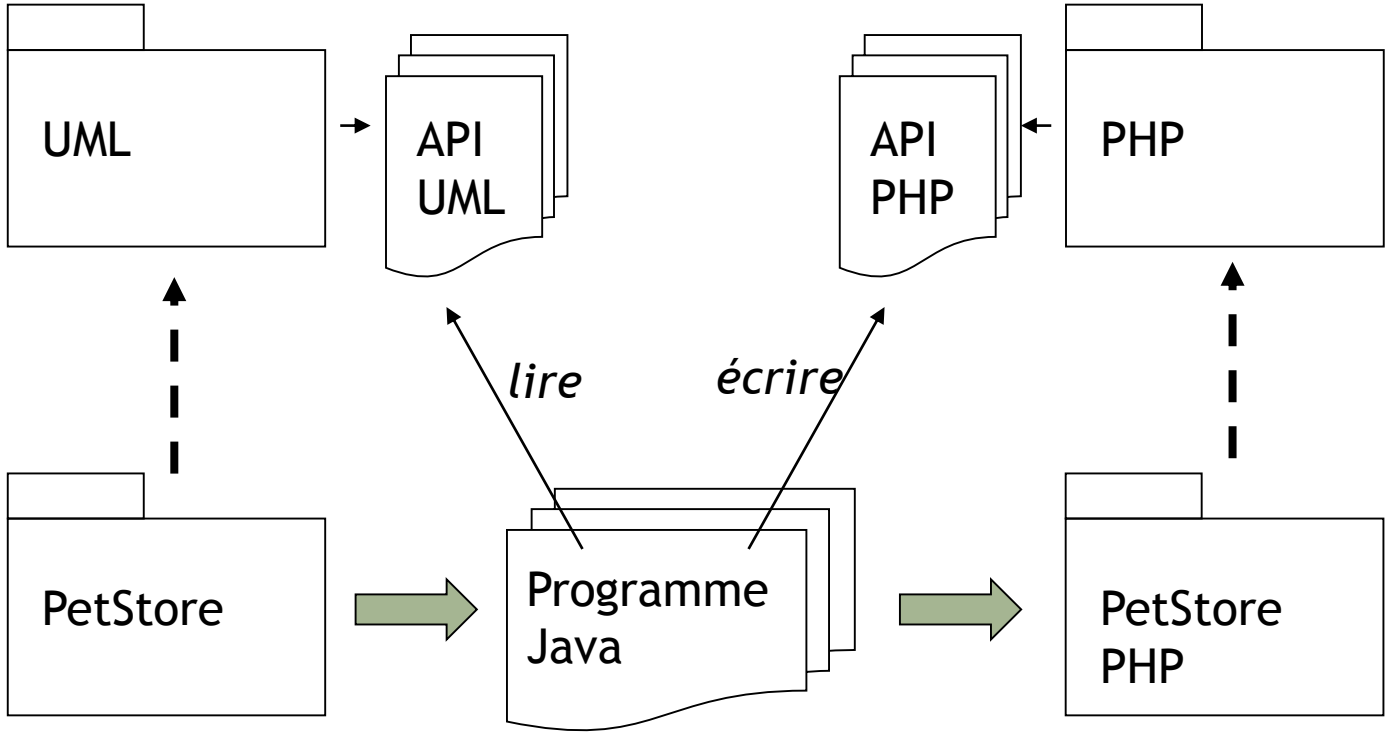
Approches de transformation de modèle

- Actuellement les transformations de modèles peuvent s’écrire selon trois approches
  - Programmation
  - Template
  - Modélisation

Model to Model

Approche par programmation

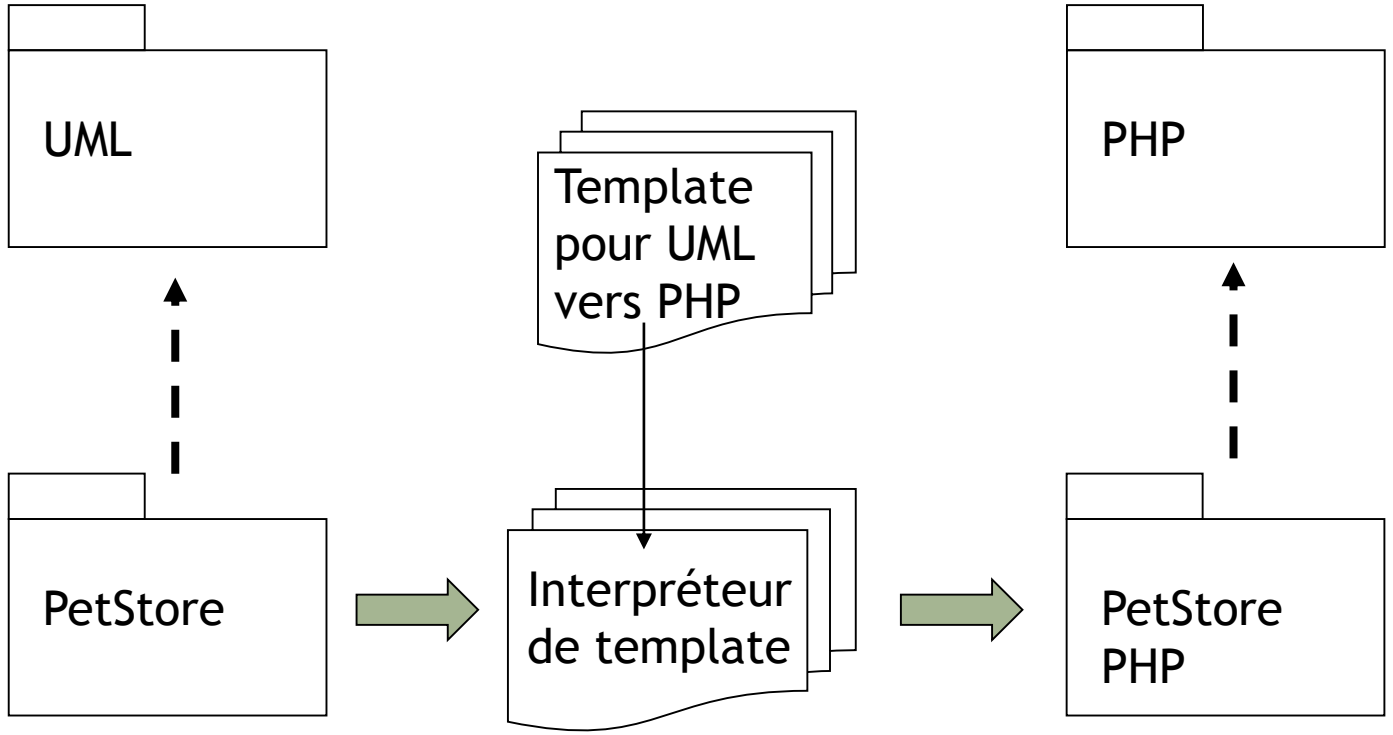
- Programmer une transformation de modèles de la même manière que l'on programme n'importe quelle application informatique
- La transformation est un programme qui utilise les API de manipulation des modèles



Model to Model

Approche par template

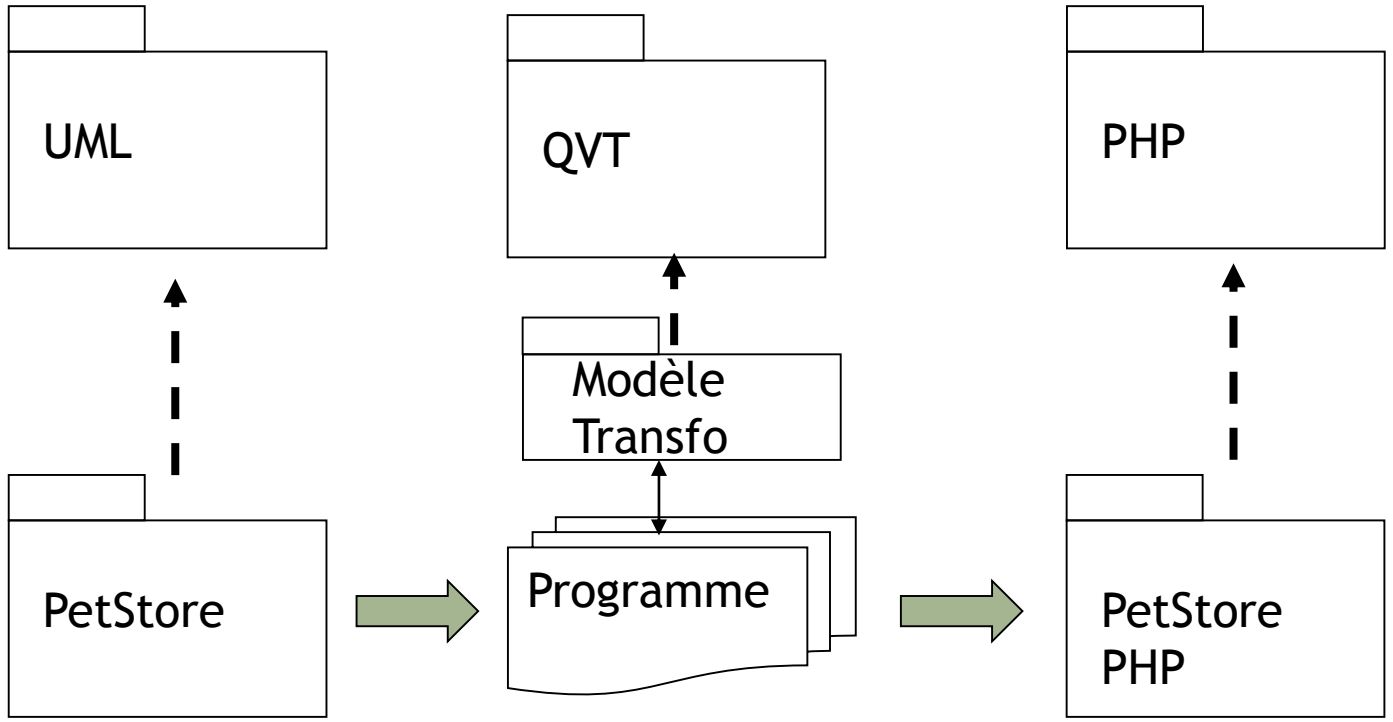
- La transformation est un template écrit dans un langage dédié.
- La transformation consiste à prendre un modèle template et remplacer les paramètres par les informations du modèles sources.
- Cette transformation nécessite un langage particulier pour définir les templates



Model to Model

Approche par modélisation

- La transformation est un modèle instance du métamodèle QVT.
- Par analogie à l'approche MDA, l'approche par modélisation modélise aussi les règles de transformations en se basant sur l'ingénierie dirigée par les modèles
- un modèle de transformations est structuré par son métamodèle, et les modèles sont indépendants des plateformes d'exécution



Query View Transformation : QVT

Définition

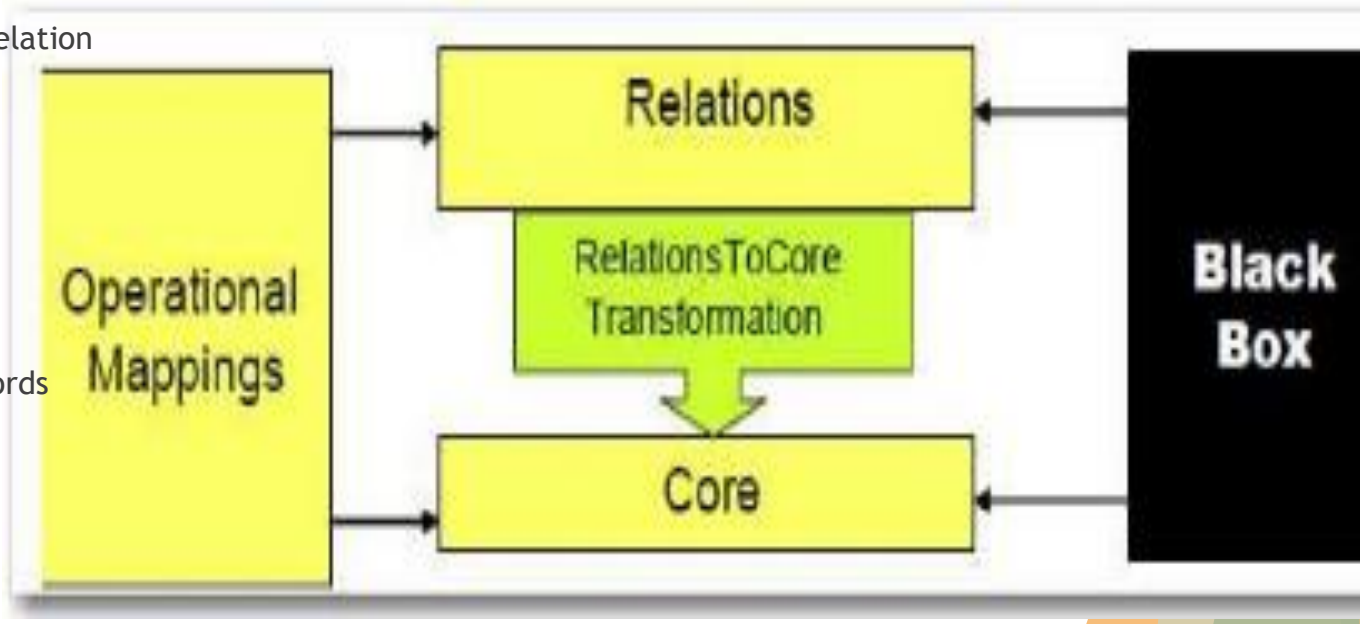
- Langage(s) de transformation et de manipulation de modèles normalisé par l'OMG
- Query/View/Transformation ou QVT
  - Query : sélectionner des éléments sur un modèle
    - Le langage utilisé pour cela est OCL légèrement modifié et étendu
    - Avec une syntaxe différente et simplifiée
  - View : une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

Query View Transformation : QVT

Langages et modes

3 langages/2 modes pour définir des transformations

- Mode déclaratif
  - Relation QVTr
    - Correspondances entre des ensembles/patrons d'éléments de 2 modèles
    - Langage de haut niveau
  - Core QVTc
    - Plus bas niveau, langage plus simple
    - Mais avec même pouvoir d'expression de transformations que relation
- Mode impératif
  - Mapping QVTo
    - 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



Query View Transformation : QVT

Principes de QVT

- **source, target** : métamodèles sources et cibles de la transformation
- **Query** : requêtes de la transformation (OCL)
- **Relation** : règles de correspondance entre source et cible
- **Mapping** : règles de construction (correspondances structurelles entre métamodèles)
  - Mapping un à un :
    - Un élément source correspond à un élément cible

    mapping MMa::A ⇒ MMb::B
  - Mapping un à plusieurs :
    - Un élément source correspond à plusieurs éléments cibles

    mapping MMa::I Set (MMb::J), MMb::K
  - Mapping plusieurs à un :
    - Plusieurs élément source sont transformés en un élément cible

    mapping MMa::E, MMa::F MMb::H

Query View Transformation : QVT

Une transformation

- Défini par un ensemble de relations comportant :
  - Domaine : ensemble d'éléments d'un modèle
  - Relation : contraintes sur les dépendances entre deux domaines : source et cible
- Une relation peut s'appliquer
  - Par principe quand on applique la transformation
  - En dépendance d'application d'une autre relation

Query View Transformation : QVT

Une transformation

- Transformation opérationnelle :
  - Défini le processus de conversion de {1..*} modèles source en {1..*} modèles cible.
  - Le scénario le plus typique :
    - Ma conforme au métamodèle MMa est converti en un modèle Mb conforme au métamodèle Mb.
    - Les méta-modèles impliqués dans la transformation se manifestent dans la signature de transformation
- signature de la transformation

```
transformation MM1ToMM2(in Source:MM1, out Target: MM2);
```

  - Exemple : transformation uml2mjava(in uml : umlMM, out mjava : mjavaMM)
- Un nom
- Input Model : in
- Output Model : out
- En paramètres:
  - les métamodèles utilisés et leur types transformation

Query View Transformation : QVT

Définition ModelType

- Il s'agit de référencer le ou les méta modèles à utiliser dans la transformation
- On utilise l'URI généré lorsqu'on enregistre le méta modèle dans Eclipse en .ecore

```
--Modeltype definition using the package namespace URI reference
modeltype MM1 uses "http://mm1/1.0";
```

- Ajouter des restrictions sur les modèles acceptés par la transformation
- La clause where répond à cette spécification

```
--Adding constraints to the defintion. In this case, the used instance of the meta-model
--should contain at least one Automaton.
modeltype MM1 uses "http://mm1/1.0" where
{ self.objectsOfType(Automaton)->size() >= 1 };
```

Query View Transformation : QVT

Syntaxe

- Commentaire
  - // It is a single line comment
  - -- it is another single line comment
  - /* It is a multi line comment */
- Affectation
  - myVar := true;
- Égalité
  - myVar = true;
- Différence
  - myVar != true;
  - myVar <> true;
- Concatenation
  - "a string" + " another string";
- Ajouter dans une liste
  - myList += aElement;

Query View Transformation : QVT

Syntaxe

- "."
  - Il est utilisé pour les objets, lors de l'appel des attributs ou l'appel des opération sur l'objet
  - Il a le même rôle que celui pour le langage Java
  - myColumn.operationOnColumn();
  - myColumn.type := "The type of the column";

- L'opérateur ' ->'
  - On peut accéder à chaque élément de la collection via l'opérateur ->
  - Il peut être vu comme un itérateur
  - myClass.attributes->map attribute2table();

  For each attribute the mapping named attribute2table is applied.

Query View Transformation : QVT

Syntaxe

- Les variables :
  - -- Typed and initialised variable

    var myStringVar : String := "an explicitly”;
  - -- Typed and non initialised variable

    var varNoInitializedAndExplicitelyTyped : Column;
  - -- Not typed but initialised variable

    var anotherStringVar := "an implicitly ";
  - -- Not typed or initialised variable

    var varNoInitializedAndNoTyped;

Query View Transformation : QVT

Syntaxe

- Conditions :

```
if (condition) then {
  /* Statement 1 */
} else {
  /* Statement 2 */
} endif;
```

Exemple

```
var a: String;
var b: Integer;
a := ( if (b=b) then "Logic" else "Miracle" endif; )
```

```
switch {
    case (condition1) /* Statement 1 */;
    case (condition2) /* Statement 2 */;
    else /* Statement 3 */;
};
```

Query View Transformation : QVT

Syntaxe

- Boucles :
  - While () loop or while()

```
while(condition) {

  /*Instruction*/
   break;
}
```

  - forEach

```
self.collection -> forEach(col | col.oclAsType(theCollection)) {....}
```

Query View Transformation : QVT

Syntaxe

- l'instantiation

```
TheObject := Object A {
-- l'instantiation
};
```

- Exemple :

```
key := object Key {
name := 'k_'+ self.name;
column := result.column[kind='primary'];
};
```

  - key est un nouvel élément crée :
    - Name est la concaténation de la chaine "k_" et le nom de l'élément transformé
    - La colonne en question est d type clé primaire

Query View Transformation : QVT

La fonction main()

- Le but de cette fonction est de définir des variables d'environnement et d'appeler la première cartographie (mapping).
- Les noms des méta-modèles peuvent être utilisés dans la fonction principale.

```
--Source represents the instance of the source meta-model. Source is a set of all
--elements. By using rootObjects, only the objects at the highest level are selected .
--Additionally, by applying the filter "[Model]" only the Model objects are selected .
main() {
  Source.rootObjects()[Model]->map toModel();
}
```

```
...
/* The transformation
*/
transformation uml2mjava(in uml : umlMM, out mjava : mjavaMM) {

/* The main operation. This is the entry point of the transformation.
*/
main() {
    uml.objectsOfType(Package)-> map package2package();
}
...
}
```

Query View Transformation : QVT

Mapping

- le mapping est le passage d'un élément du modèle d'entrée vers un ou plusieurs éléments du modèle de sortie.
- La déclaration de mapping

```
--Mapping declaration of "toModel()". Because both the input and output class is named
--"Model", --it has to be prefixed with the modeltype the objects belong to.
mapping MM1::Model::toModel() : MM2::Model { ... }
```

  - Le nom de mapping doit être unique
  - Il y a pas de possibilité de faire le polymorphisme
  - Exemple : Si on veut transformer une classe en une table :

```
mapping javaMM::Class::classToTable() : sqlMM::Table {
        …
}
```

Query View Transformation : QVT

Mapping

- La mapping avec condition sur le modèle source :

```
--Pre-condition: the Name of the Model object from the MM1 instance should
--start with an "M"
mapping MM1::Model::toModel() : MM2::Model
--The keyword "self" refers to the input object.
when { self.Name.startsWith("M"); }
{ ... }
```

Query View Transformation : QVT

Mapping

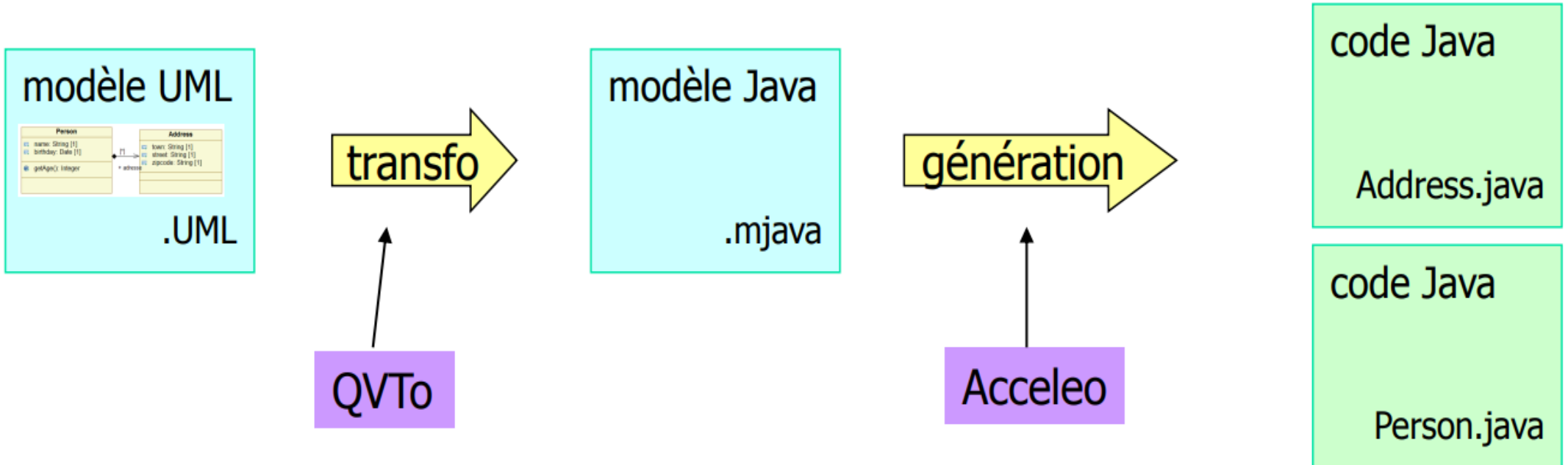
- Le corps du mapping contient trois partie:
  - Init : pour initialiser les paramètres et les variables.
  - Population : dans laquelle on développe la transformation
  - End: contient un code additionnelle qui doit s'executer avant la fin du mapping

```
init { log("Arrived in mapping toModel().");}

--Population section. This keyword can be ommitted.

--Copy the name from the input object to the output object
Name := self.Name;
--A model contains at least one Automaton, each one must be
--mapped after which it is assigned to the target model.
body += self.body->map toBody();

end { log("Leaving the mapping toModel().");}
```

Model to Text

Principe

- Génération de code
  - Modèle de conception ou de programmation
  - Traduction vers un langage cible
- Mise en œuvre
  - Analyse syntaxique du modèle
  - Génération de fichiers (.txt, .doc, .xmi, .html, .java …)



Model to Text

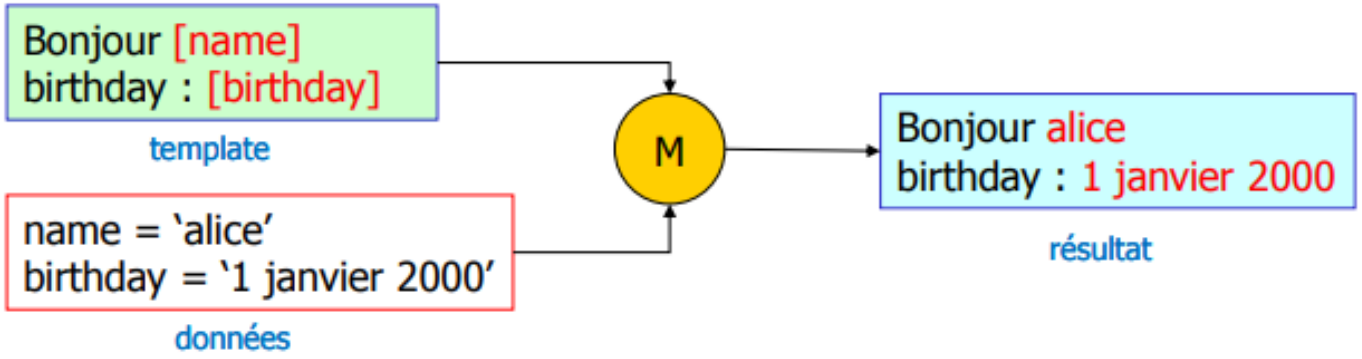
Techniques

- Basé sur des parseurs existants
  - XML/XSLT
- Basé sur des langages de programmation
  - Approches à base de visiteurs (Java)
  - API Java d'EMF
- Basé sur des templates de transformations
  - JET/Acceleo sous EMF
  - Velocity
  - …

Model to Text

Transformation par Template

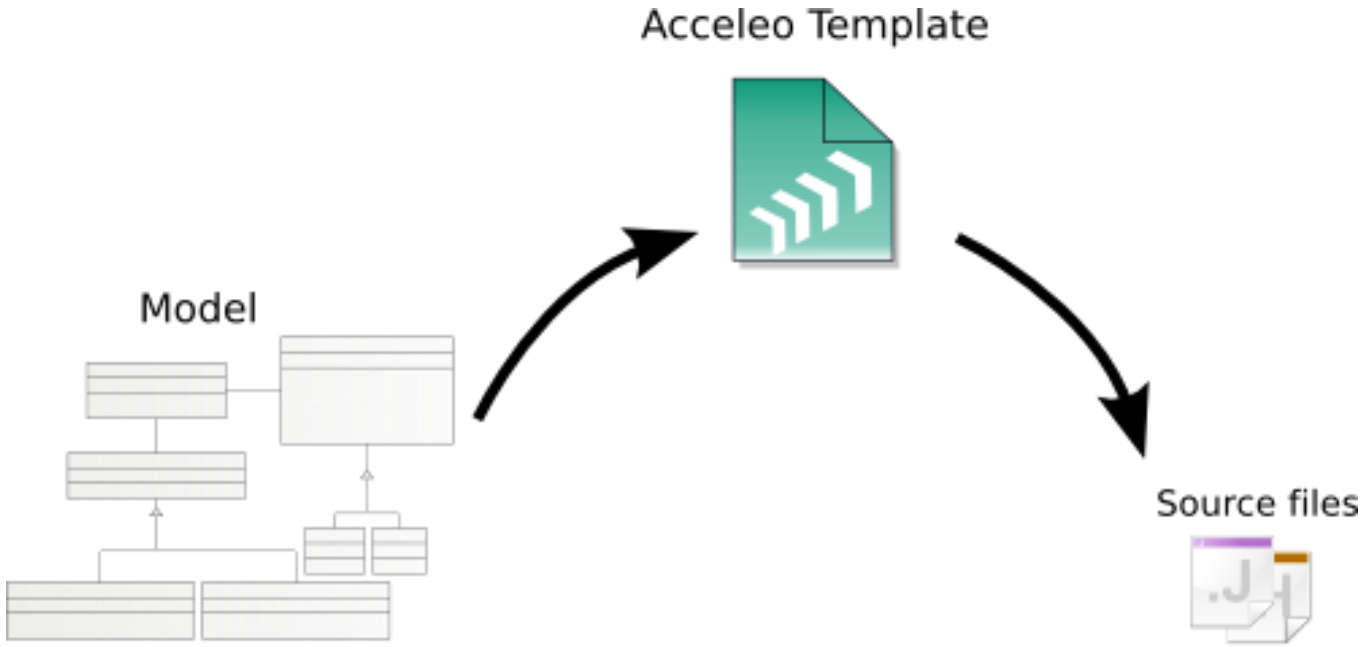
- Une template = une structure à base de textes à générer et de meta-données
- Structure proche de la structure du code généré : Plus intuitif
- Un templates sert à générer tout type de texte
  - xml, code, texte, …
- C'est une ébauche du texte à générer
  - contient des 'trous' à remplacer par les vrais valeurs
- On 'instancie' un template
  - on spécifie les valeurs à utiliser
  - on obtient le texte final



Acceleo

Présentation

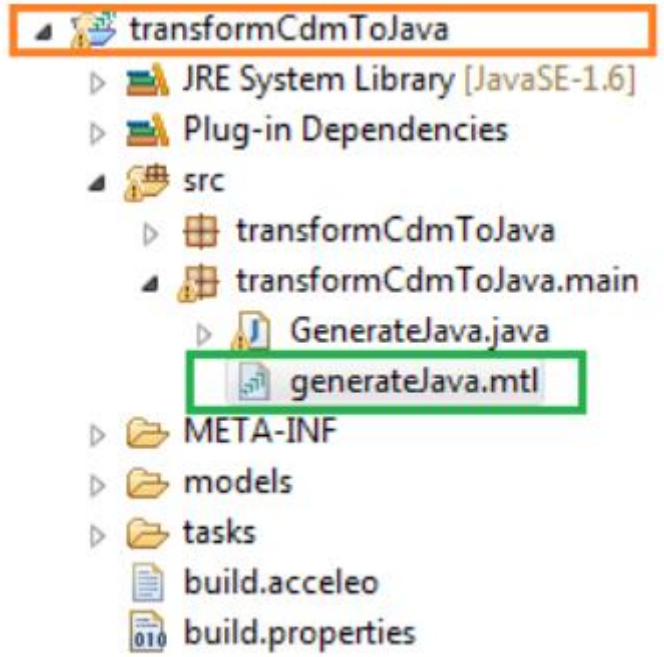
- **Acceleo** est un générateur de code source de la fondation Eclipse
- Il permet de mettre en œuvre l'approche MDA (Model driven architecture) pour réaliser des applications à partir de modèles basés sur EMF
- Il s'agit d'une implémentation de la norme de l'Object Management Group (OMG).
- pour les transformations de modèle vers texte (M2T) Model to Text.



Acceleo

Module

- Un module acceleo est un fichier .mtl qui contient :
  - Des templates pour générer le code contenant :
    - Texte constant à générer
    - Métadonnées et structures
    - Commentaires
    - Vérification de la syntaxe à l'écriture
  - Des queries pour extraire des informations dans les modèles
- Sert d'espace de nommage
- Le fichier doit commencer par
  - [module <nom_module>('URI_du_metamodel')]
  - [module generateJava('http://cdm/1,0')]



generate.mtl

```
[comment encoding = UTF-8 /]
[module generate('http:///mjava.ecore')/]

...
```

Acceleo

module

```
[comment encoding = UTF-8 /]
[**
 * The documentation of the module generate2.
 */]
[module generate2('http:///document.ecore')]


[**
 * The documentation of the template generateElement.
 * @param aDocument
 */]
[template public generateElement(aDocument : Document)]
[comment @main/]
[file (aDocument.name, false, 'UTF-8')]

[/file]
[/template]
```

Acceleo

Template

- Spécifie un template avec des trous
  - Permet de générer le texte
- Est délimité par
  - [template]...[/template]
- Contient le texte à générer et les endroits à remplacer par des données
  - [c.name/]
- A au moins un paramètre
  - élément du métamodèle qui est utilisé pour obtenir les données

generate.mtl

```
[comment encoding = UTF-8 /]
[module generate('http:///mjava.ecore')/]

[template public generateClass(c : Class)]
    name = [c.name/]
[/template]
```

Acceleo

File

- Tag à utiliser dans un template
- Permet de spécifier le fichier de sortie (dans lequel le texte est généré)
  - [file (< uri_expression>,< append_mode> , '<output_encoding >')] (...) [/file]
    - uri_expression
      - nom du fichier à générer
      - peut contenir un chemin d'accès – les répertoires sont alors créés
    - append_mode
      - optionnel – indique si le texte doit être remplacé ou ajouté
    - output_encoding
      - optionnel

```
[template public generateClass(aClass : Class)]
[file (aClass.name.toUpperFirst()+'.java', false, 'UTF-8')]
    public class [aClass.name.toUpperFirst()/] {
        [comment declaration des attributs/]
     }
```

```
[comment encoding = UTF-8 /]
[module generate('http:///mjava.ecore')/]

[template public generateClass(c : Class)]

[comment @main /]
[file (c.name, false, 'UTF-8')]
  name = [c.name/]
[/file]
[/template]
```

Acceleo

Exécution de template

- Un template peut être exécuté de deux façons:
  - soit par appel par le moteur
    - le moteur recherche dans le modèle tout les éléments du type requis par le paramètre du template
    - Pour chaque élément, le template est appelé
  - soit par appel explicite, à partir d'un autre template
    - équivalent à un appel de méthode

Acceleo

Exécution de template

- Le moteur execute:
  - les templates contenant un tag [file]
  - les templates contenant une annotation ▫ [comment @main /]

```
[module generate('http:///mjava.ecore')/]

[template public generateClass(c : Class)]

[comment @main /]
[file (c.name, false, 'UTF-8')]
class [c.name/]
[/file]
[/template]

[template public generateName(ele : NamedElement)]
[ele.name/]
[/template]
```

ici, seul le premier template est executé par le moteur

Acceleo

Exécution de template

- appel explicite
- Un template peut appeler d'autre template (~ appel de méthode)
  - l'appel se fait sur une instance du type du paramètre
    - [c.generateName()/]
  - ou en passant les paramètres
    - [generateName(c)/]

```
[template public generateClass(c : Class)]

[comment @main /]
[file (c.name, false, 'UTF-8')]

name = [c.generateName()/]

[/file]
[/template]

[template public generateName(ele : NamedElement)]
[ele.name/]
[/template]
```

Acceleo

Conditions : [if]

- Pour spécifier une partie conditionnel dans un [template]

```
[if (condition)]
    (...)
[/if]
```

```
[if (condition)]
    (...)
  [else]
    (...)
[/if]
```

```
[if (condition)]
    (...)
 [elseif (condition)]
    (...)
 [elseif (condition)]
    (...)
[/if]
```

Acceleo

Boucles [for]

```
[for (iterator : Type | expression)]
   (...)
[/for]
```

```
[for (expression)]
    (...)
[/for]
```

version light

Acceleo

Queries

- Une query est utilisé pour extraire de l'information du modèle
  - elle retourne généralement des objets du modèle
- Possède :
  - Visibilité : public, protected, private (comme la POO)
  - Nom de la query
  - Paramètres typés :
    - Type OCL (Boolean, String, Integer …)
    - Par le méta modèle
  - Type de retour
  - La query en elle-même
- [query visibilité nom_query(param : TypeParam) : TypeRetour=valeur]

```
[query private getAttribute(arg:Class): Set(Attribute) = arg.attributes/]

[query private getPropertiesName(arg:Class): Sequence(String) = arg.reverse.name/]
```

Acceleo

Queries use

```
[query public getPublicProperties(c : Classifier) : Set(Property)
  = c.properties->select(visibility = VisibilityKind.public)
/]
```

```
[template public main(c : Class) {javaName : String = c.genName();} ]

[comment @main /]
[file (c.name, false, 'UTF-8')]
[javaName/] {

/* Public properties */
  [for (p : Property | c.getPublicProperties())]
  public [p.genName()/];
    [/for]
}
[/file]

[/template]
```

Acceleo

- Écrire le template permettant de générer un squelette simple d’une classe. On utilise le métamodèle UML du diagramme de classe.