MDA

Cours de 2e année ingénieur
Spécialisation « Génie Informatique »

fabien.romeo@fromeo.fr

Plan général

- Introduction (Séance 1)
  - Principes généraux
  - Modèles et méta-modèles
- Manipulation de modèles (Séances 2&3)
  - Exemples de méta-modélisation
  - Eclipse Modeling Framework
- Transformation de modèles (Séance 4&5)
  - Principes généraux
  - Règles de transformation par programmation

Principes généraux

Model-Driven Architecture

- Nouvelle méthode de génie logiciel (2000), initié par l'Object Management Group (ceux du standard UML)
- Méthode = Langage
  + Démarche
  + Outils
- A donné naissance au domaine plus général du Model-Driven Engineering (Ingénierie Dirigée par les Modèles)

Motivation 1

- Constante évolution des technologies
- Pour une entreprise, il est crucial de posséder la dernière technologie
  - Problème : pour la même application, il faut tout recommencer ($$$)
- Exemple :
  - Procédural (C, Cobol, …) → Objet
  - Objet (Java, C++, …) → Composant
  - Composant (Java EE, .Net, …) → Services
  - Services (WS-*, REST, OSGi, …) → Futur paradigme

Motivation 2

- A quoi sert la modélisation pour le logiciel, puisqu'il faudra toujours produire du code ?
  - A quoi servent les diagrammes UML ?
    - Pour être à la mode, il faut en faire
    - Pour remplir un rapport et montrer qu'on travaille sur des choses complexes
    - A rien, il n'y a pas besoin de diagrammes UML quand on fait de l'open source, la documentation c'est le code source

Motivation 3

- La miniaturisation et les avancées technologiques ont permis d'embarquer du logiciel dans des systèmes tout aussi nombreux que différents
  - Ex : ordinateurs, pda, téléphones, set-top boxes, automobile, domotique, …
- Comment développer une application logicielle pour qu'elle soit exécutable sur tous ces appareils ?

Intention du MDA

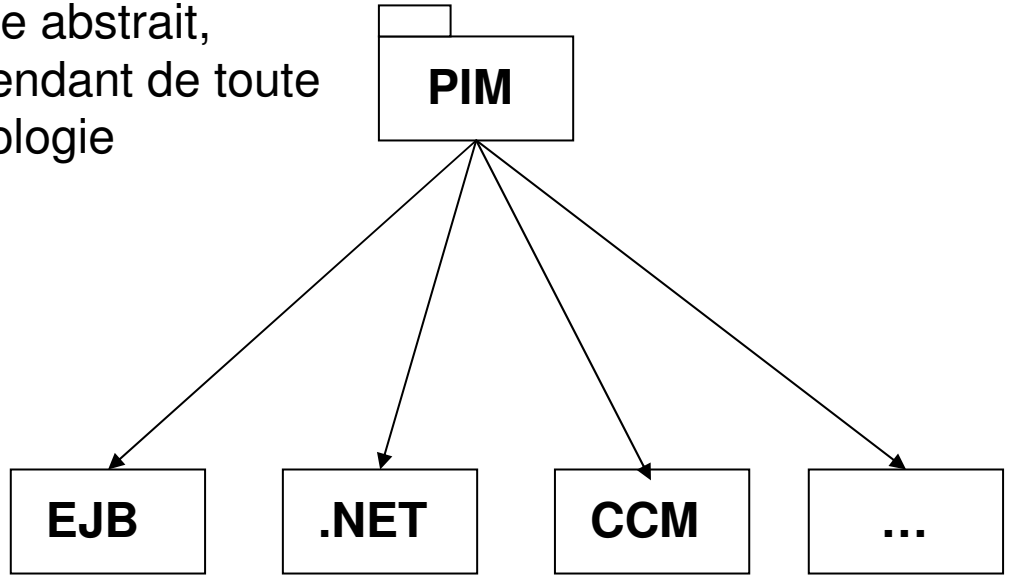
- Pérennité des savoir-faire
  - La durée de vie des modèles doit être plus longue que celle du code (modèles abstraits)
- Gains de productivité
  - L'utilisation des modèles ne doit plus être seulement contemplative mais productive (automatisation des opérations sur les modèles et génération de code)
- Prise en compte des plateformes d'exécution
  - La prise en compte des plateformes doit être explicitée par des modèles (modélisation des plateformes et lien avec les modèles des applications)

Modèles du MDA

- Le MDA définit 2 principaux niveaux de modèles d'application
  - PIM (Platform Independent Model)
    - Modèle spécifiant une application indépendamment de la technologie mise en œuvre
    - Modélise seulement la partie métier
  - PSM (Platform Specific Model)
    - Modèle spécifiant une application après projection sur une plateforme technologique donnée (définie par un PDM)
- Et un niveau pour les plateformes
  - PDM (Platform Description Model)
    - Modèle décrivant une plateforme de déploiement

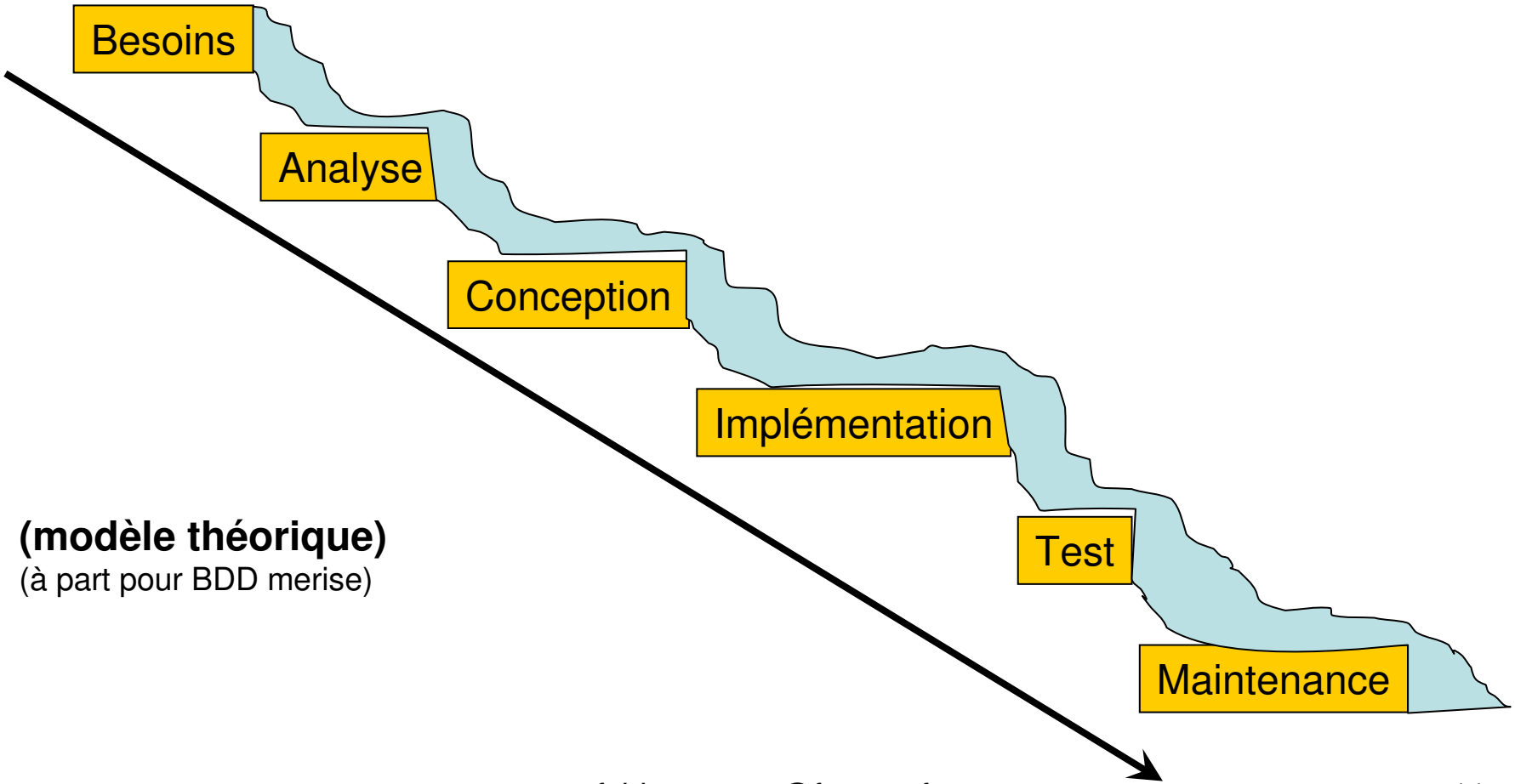
Model Once, Generate Everywhere

Modèle abstrait, indépendant de toute technologie



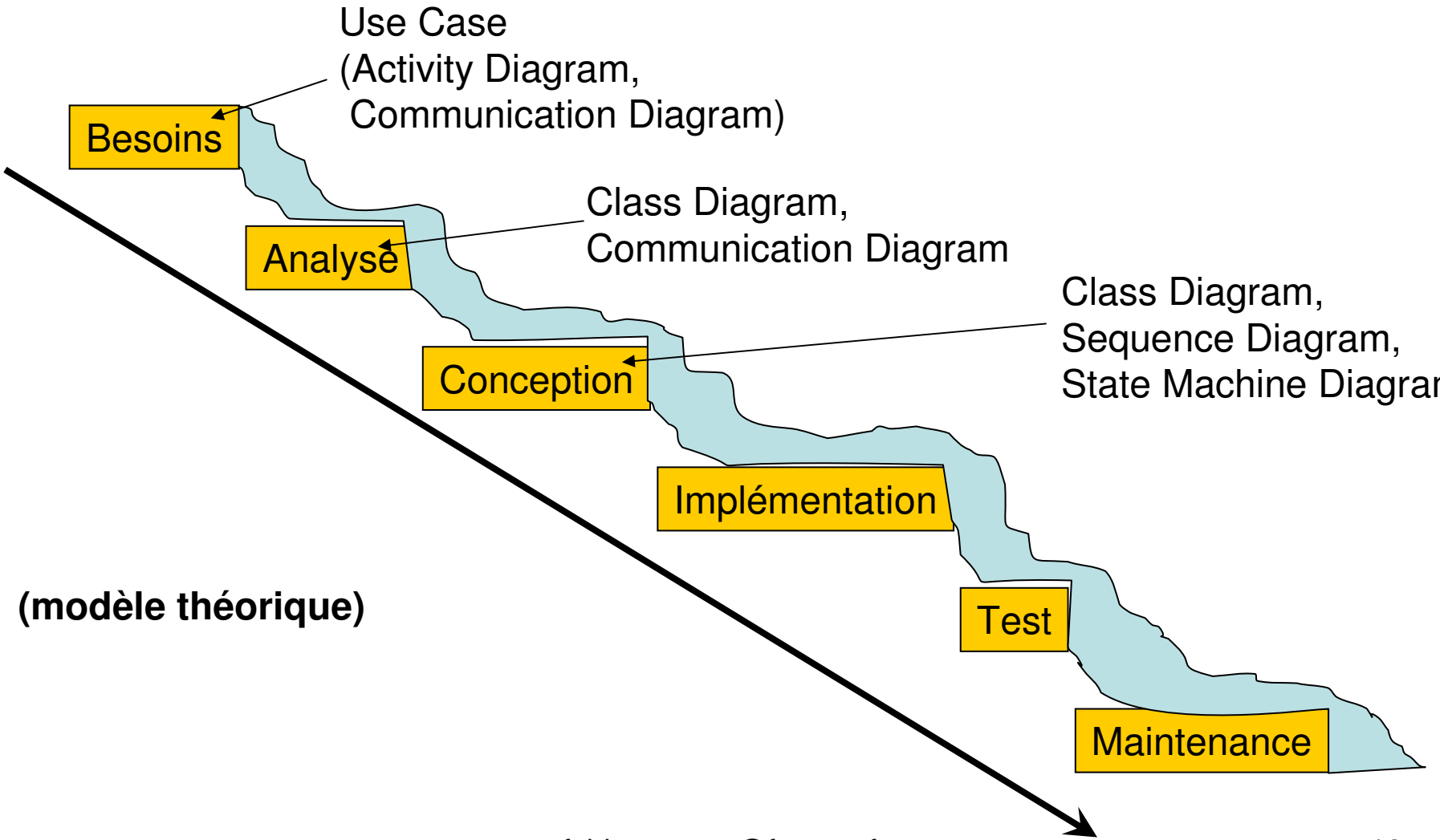
Génération de code pour plusieurs plateformes (même celles à venir)

Cycle de développement en cascade (rappel)



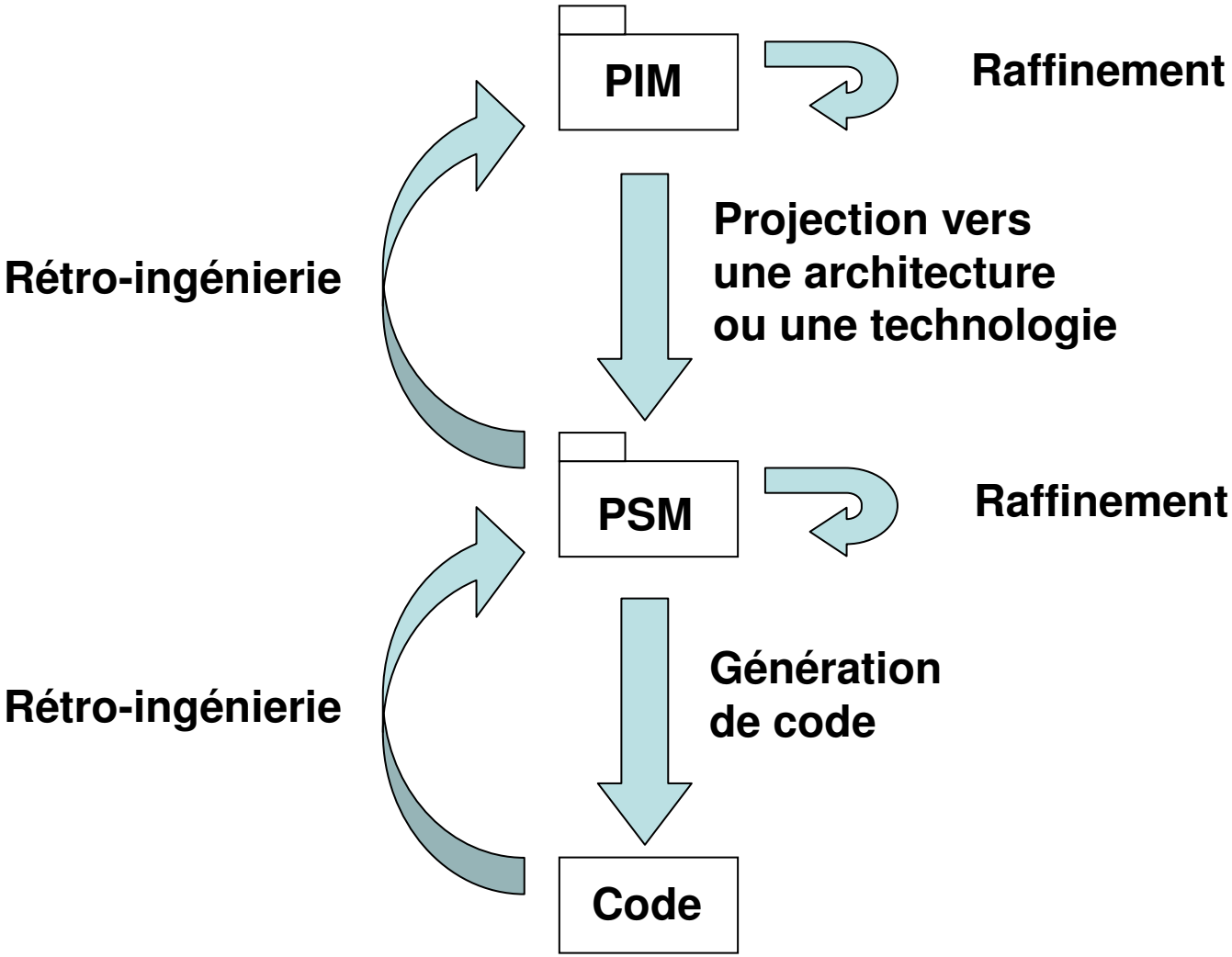
(modèle théorique)
(à part pour BDD merise)

Cycle de développement en cascade avec UML (rappel)

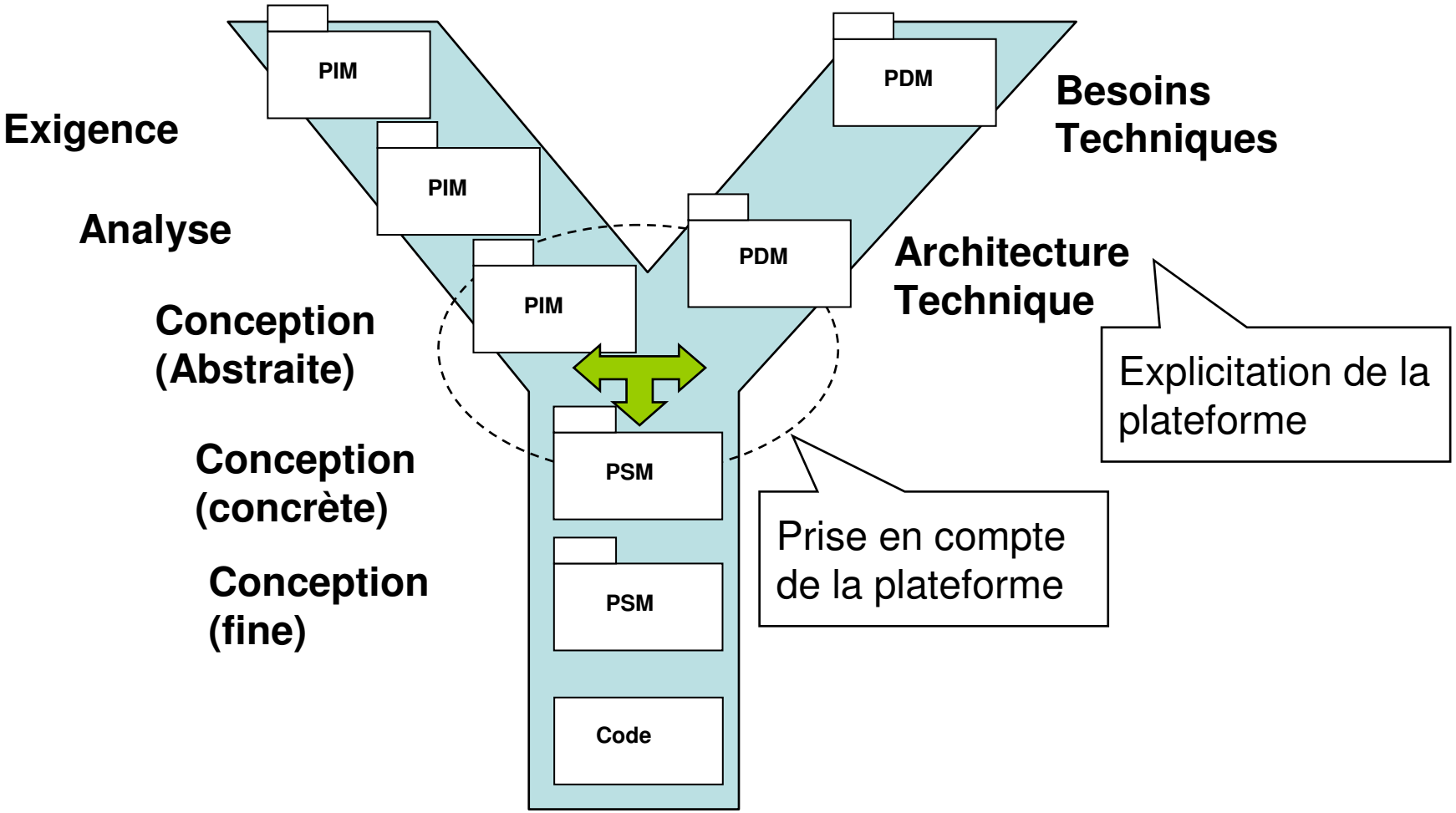


(modèle théorique)

Niveaux de modélisation

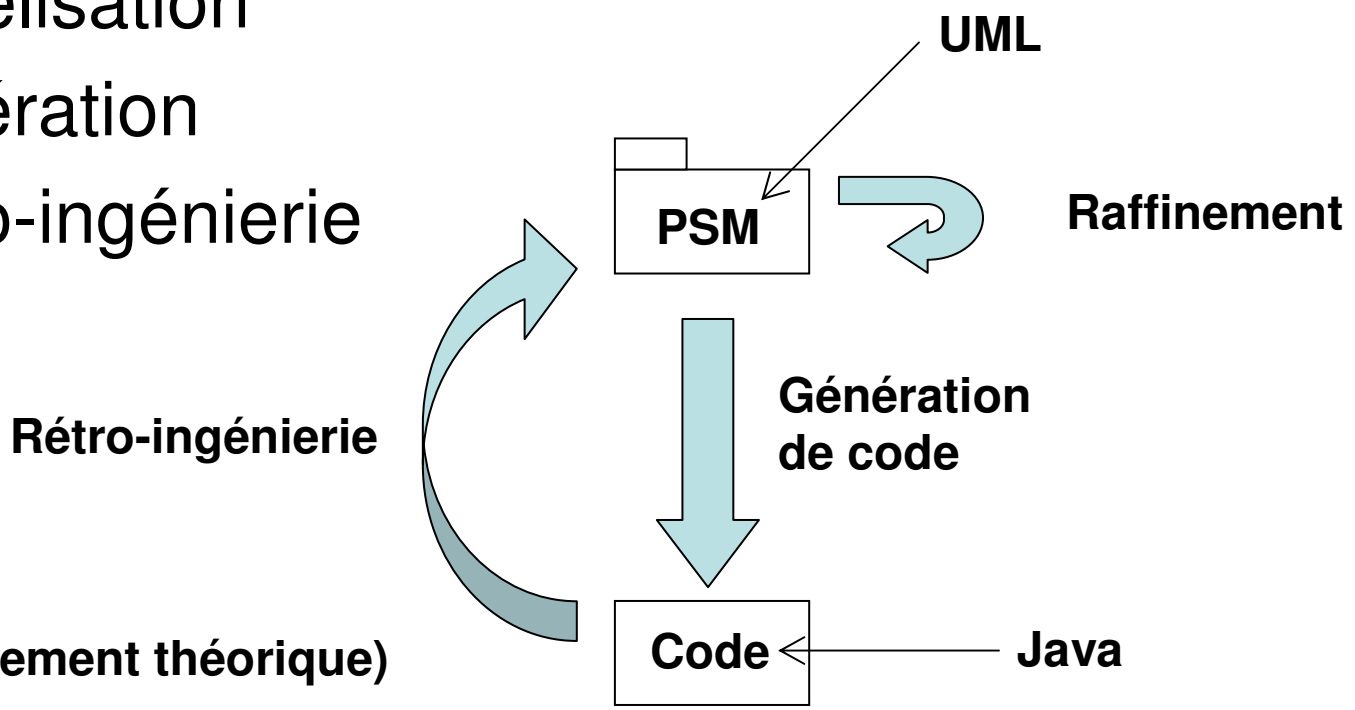


Cycle de développement et plateforme



Travaux pratiques (n°1)

- Utilisation de NetBeans pour pratiquer le cycle de développement du MDA
  - Modélisation
  - Génération
  - Rétro-ingénierie

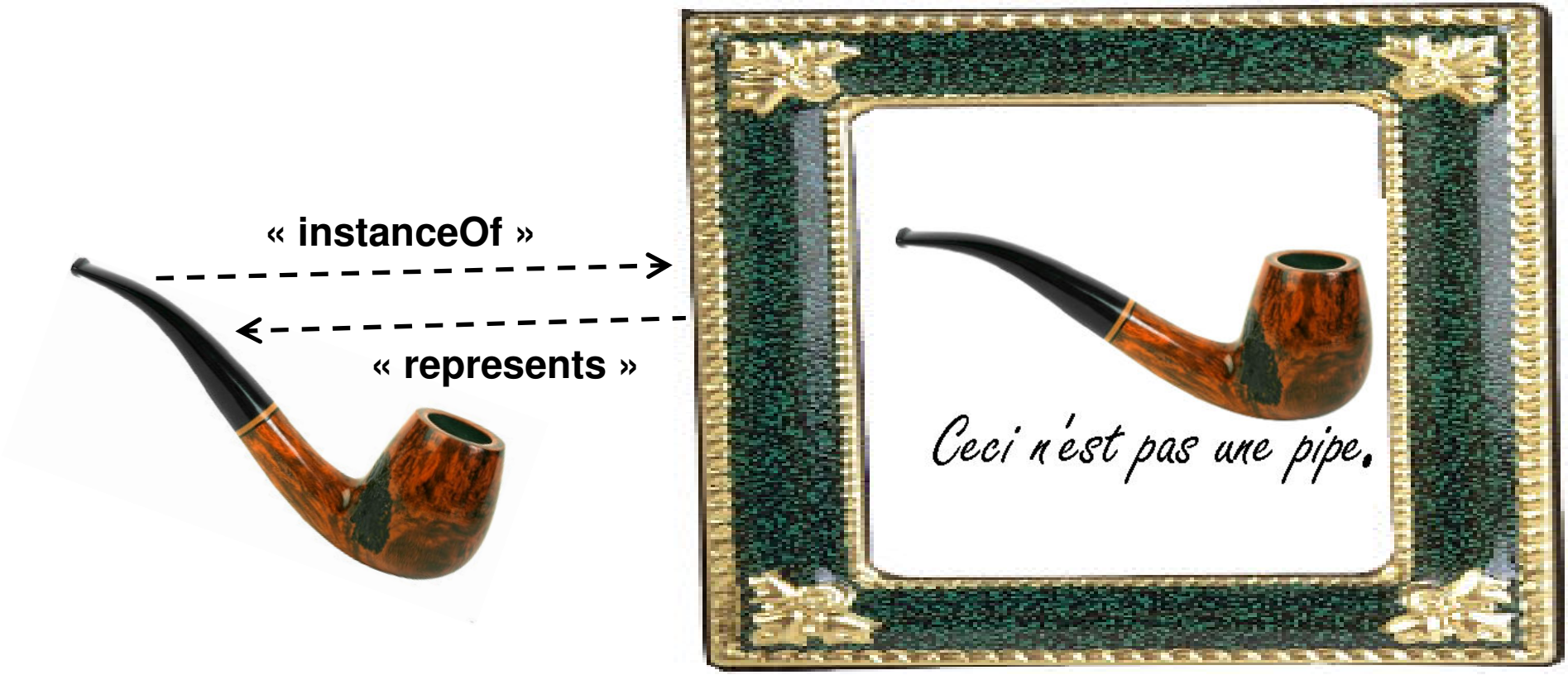


(modèle pas seulement théorique)

Modèles et méta-modèles

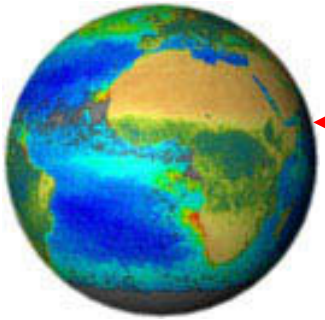
Système et modèle

- Un modèle est une abstraction d'un système physique, construit dans une intension particulière.



Autre exemple

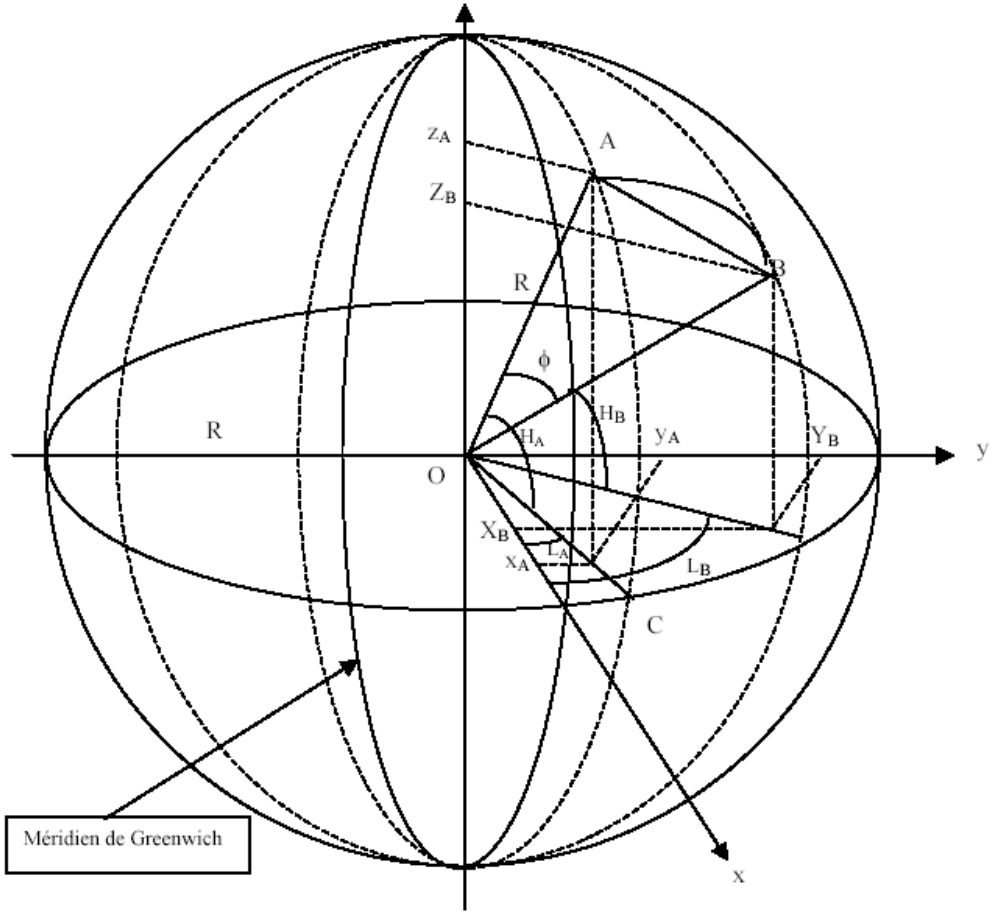
La mappemonde est un modèle de la terre



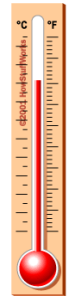
modelOf

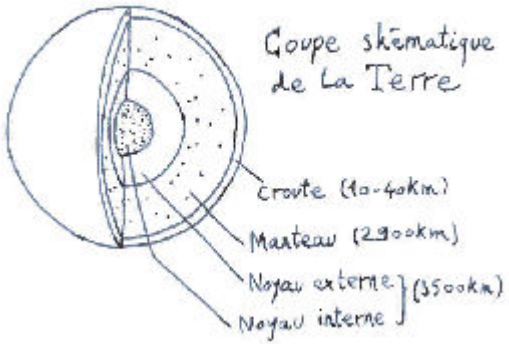
Autre exemple (suite)

- Permettant de poser certaines questions …

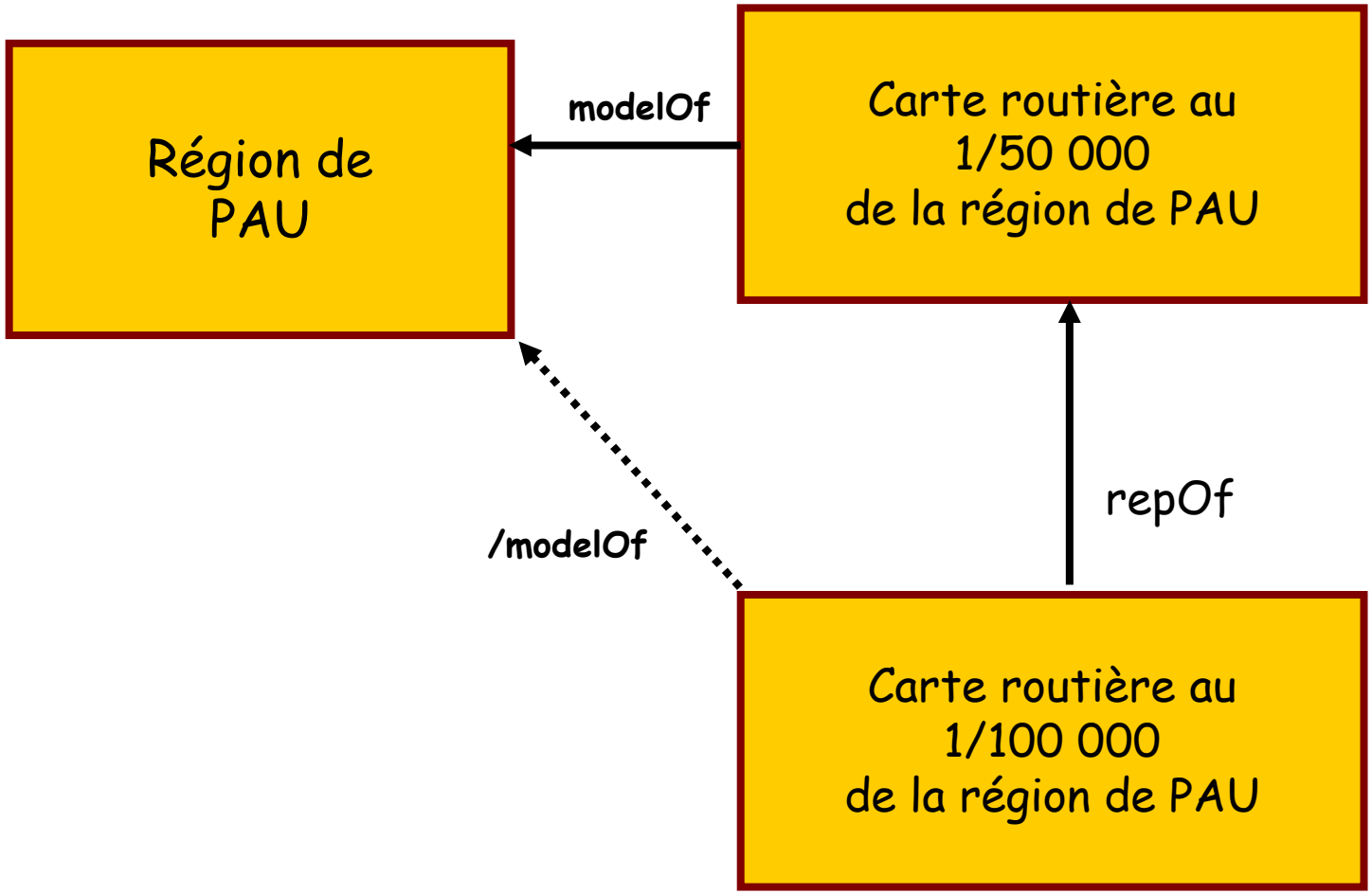


- mais pas d'autres.





Un modèle peut représenter un modèle

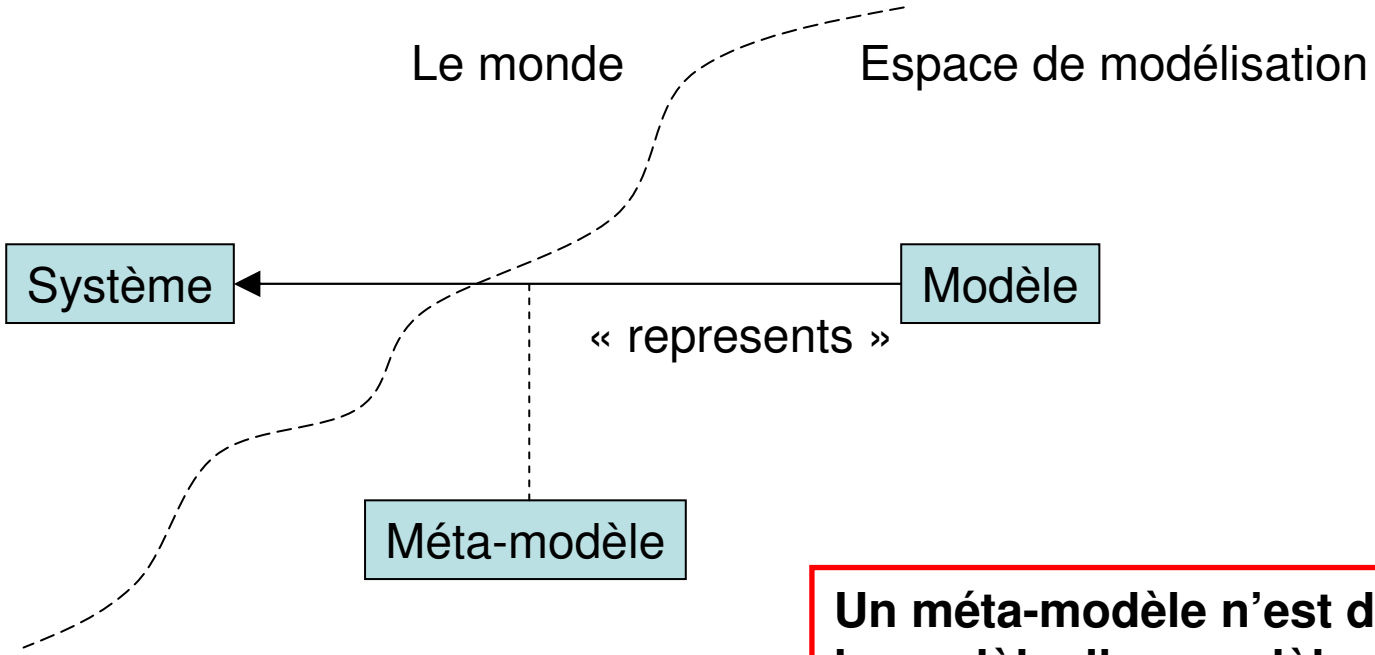


Méta-modèle

- L'approche MDA est basée sur les notions de modèle et de méta-modèle
- Le préfixe « méta » exprime l'auto-référence
- Méta-*
  - Méta-physique : la physique de la physique
  - Méta-classe : la classe des classes
    - java.lang.Class en Java
  - Méta-table : la table des tables
    - user_tables pour Oracle
  - Méta-modèle : le modèle d'un modèle ?

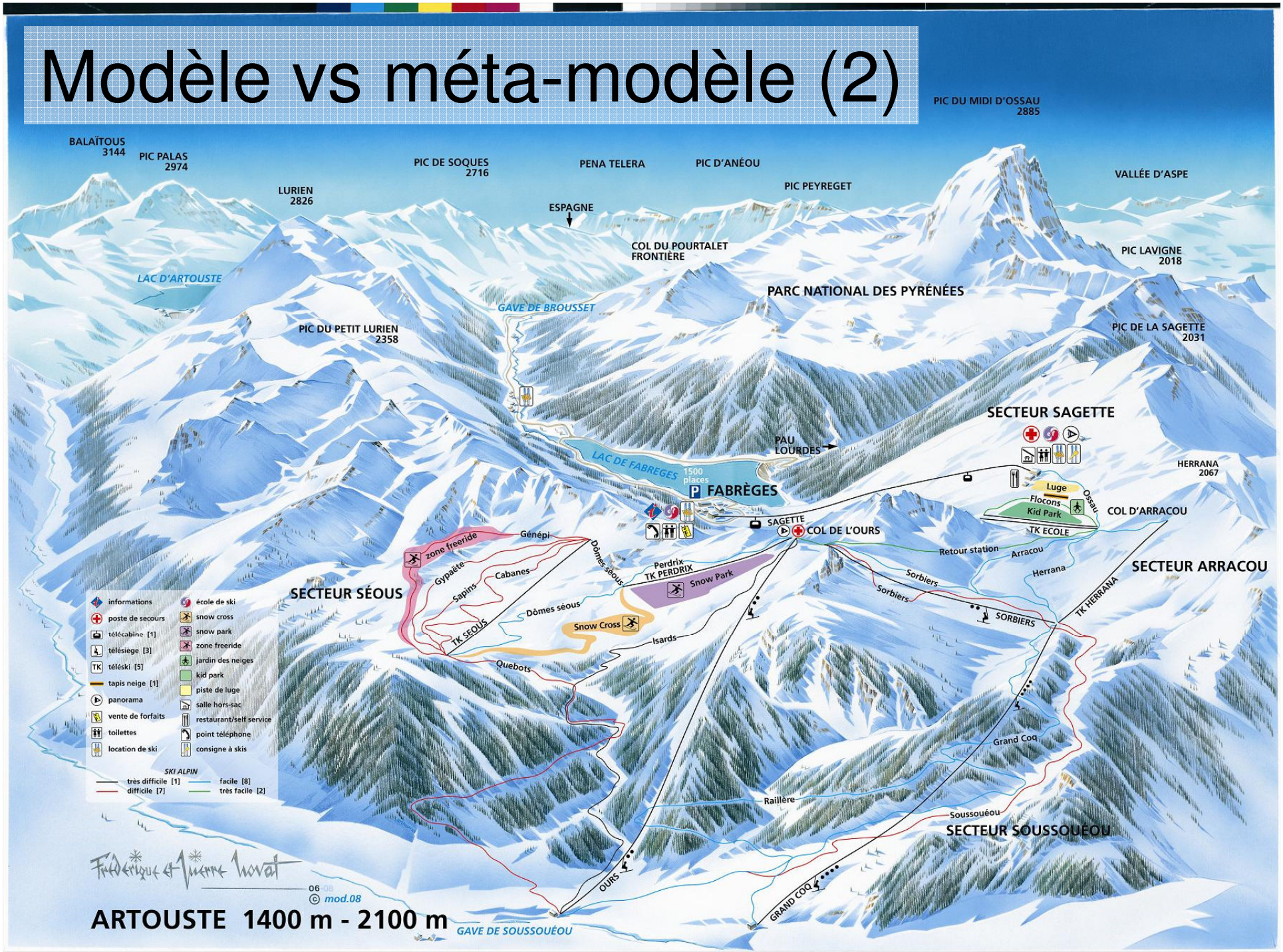
Modèle vs Méta-modèle

- Un méta-modèle définit la relation entre un modèle et un système

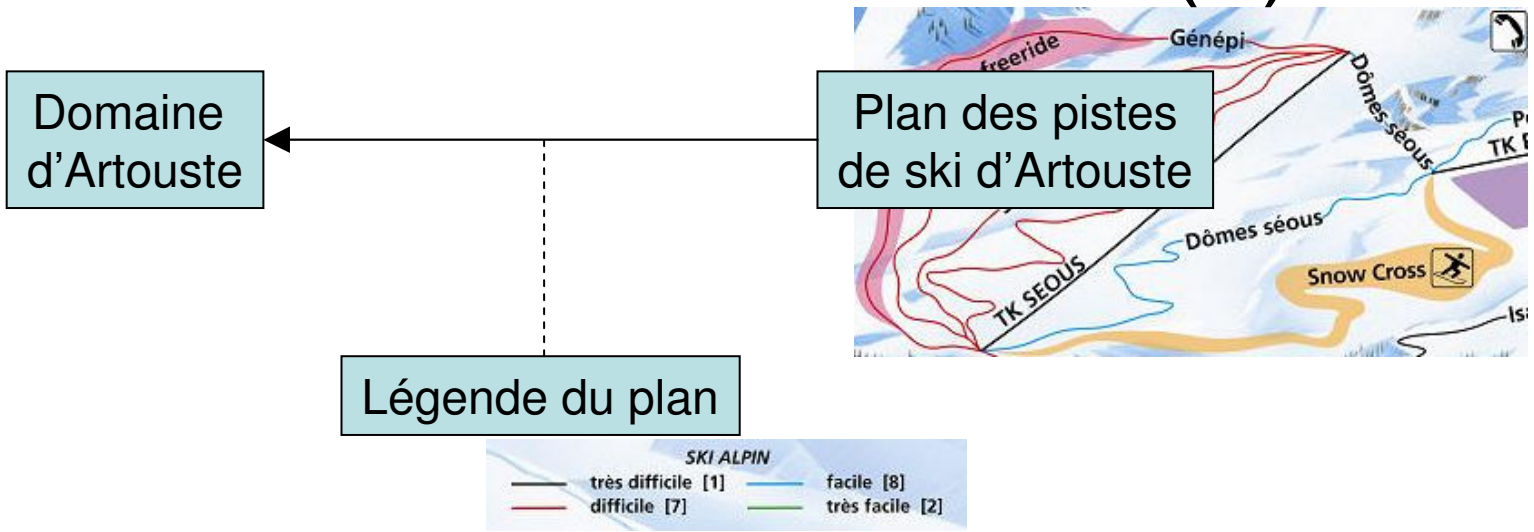


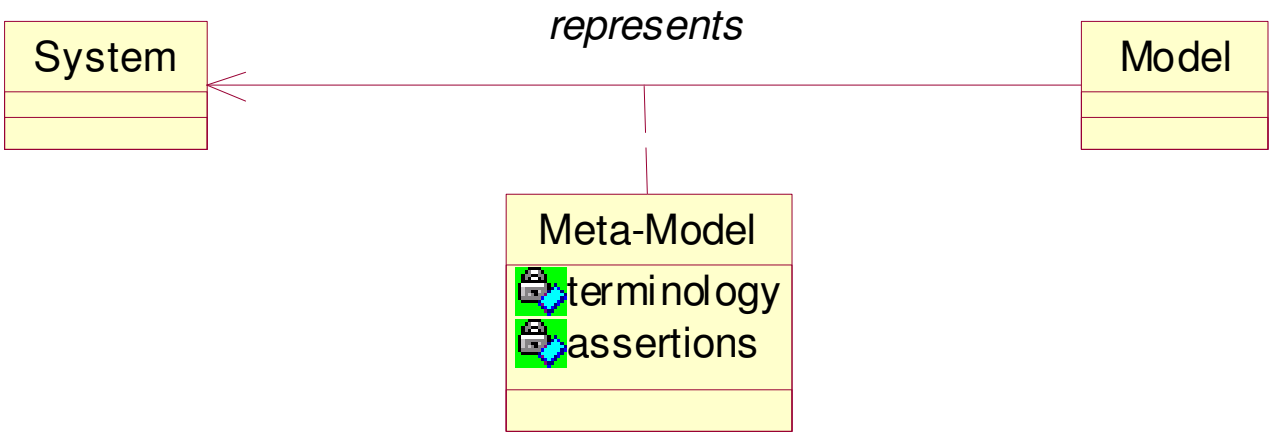
Un méta-modèle n'est donc pas, le modèle d'un modèle…

Modèle vs méta-modèle (2)



Modèle vs méta-modèle (3)

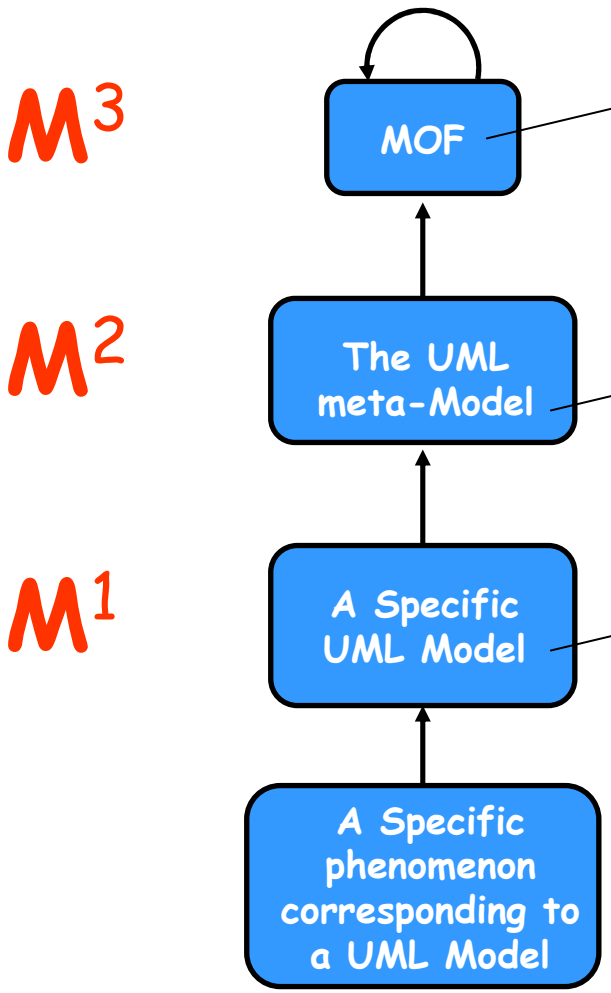




Méta-méta-méta-…

- Comment peut-on définir un méta-modèle ?
  - À l'aide d'un méta-méta-modèle !
- Et comment définir ce méta-méta-modèle ?
  - À l'aide d'un méta-méta-méta-modèle !
- Et comment définir ce méta-méta-méta-… ?
  - À l'aide d'un méta-méta-méta-…
- Le problème paraît sans fin …

Niveaux de modélisation dans UML

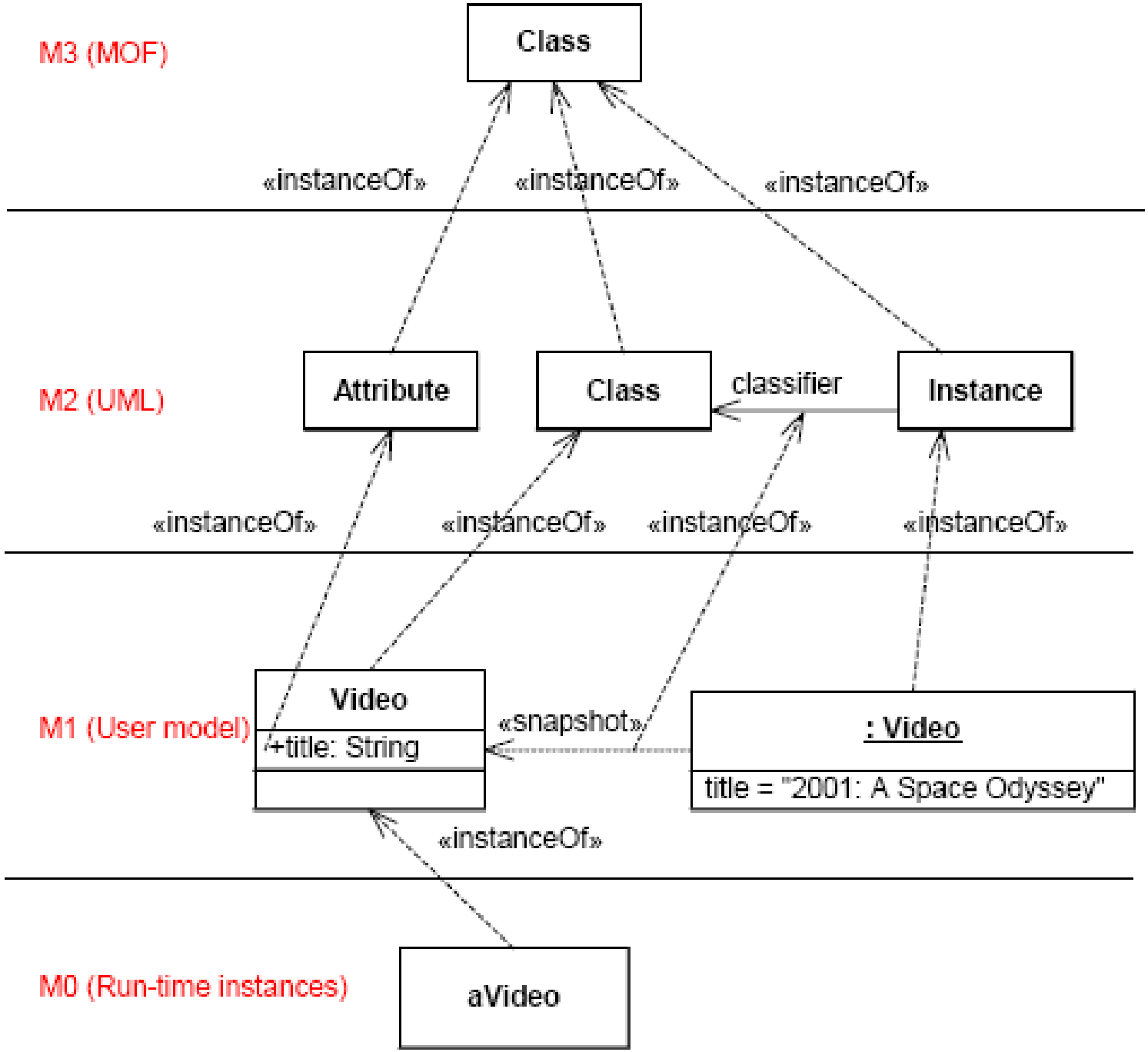


Définit les concepts d'un diagramme de classe sous la forme d'un diagramme de classe

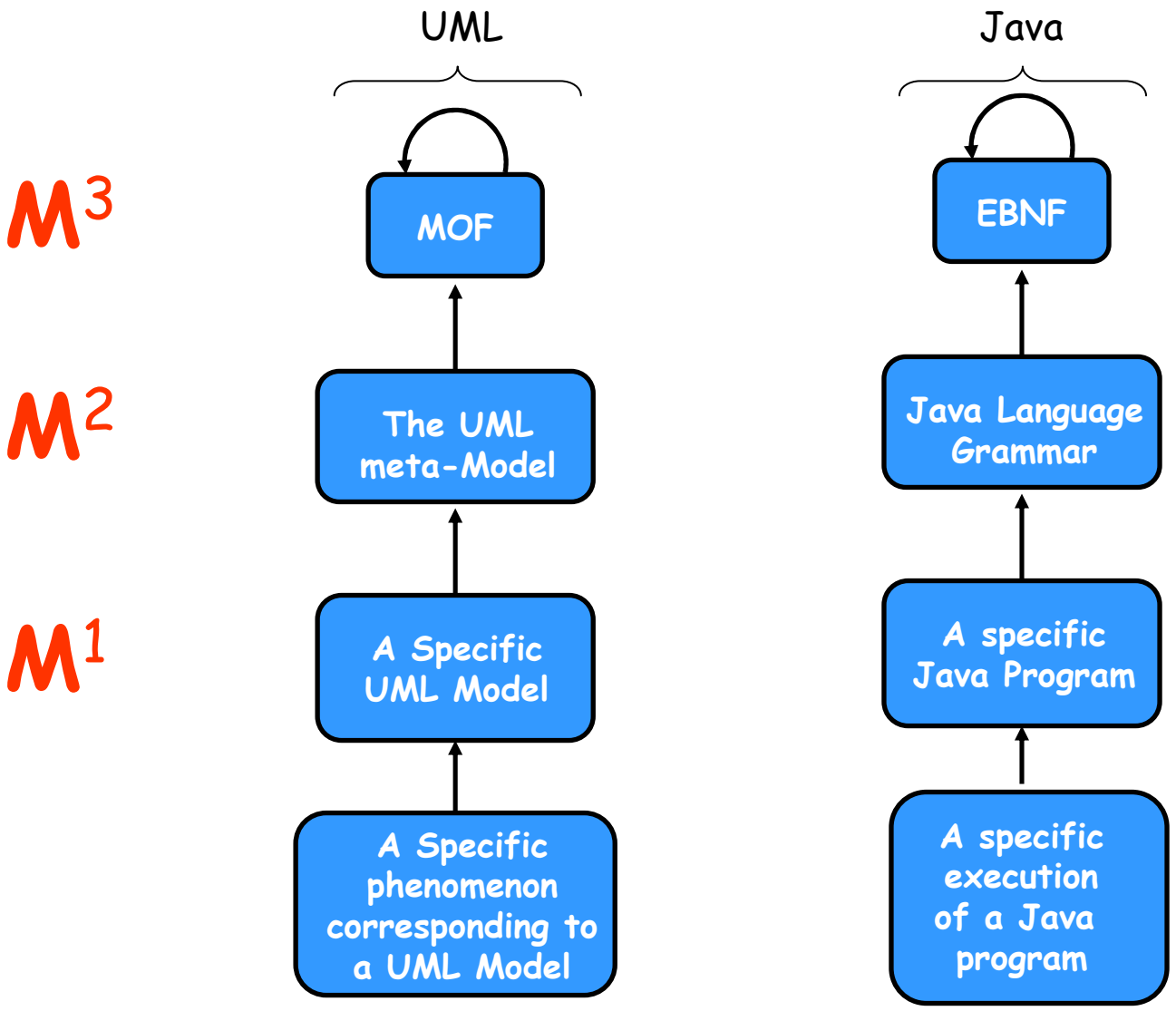
Définit les concepts des diagrammes UML sous la forme d'un diagramme de classe

Modélise le réel avec des diagrammes de classe, des machines à état, …

Les 4 niveaux de modélisation dans UML (exemple)



Niveaux de modèles : comparaison



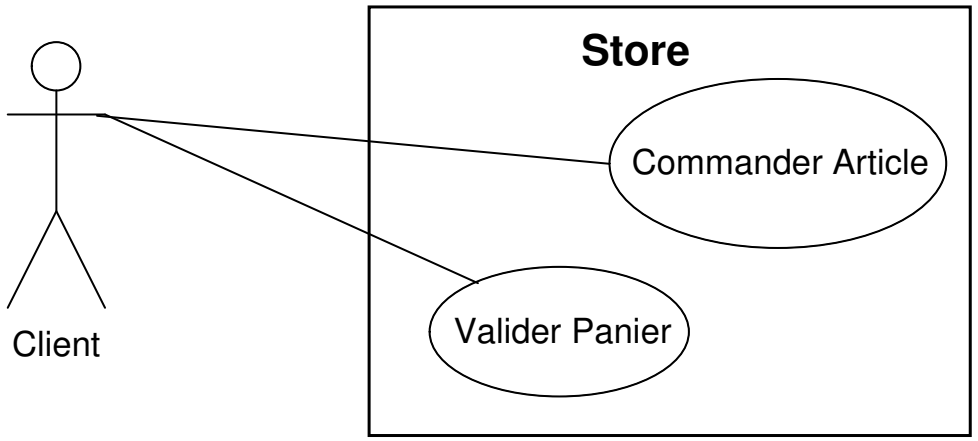
Exemples de méta-modèles

(méta-modèle d'UML allégé)

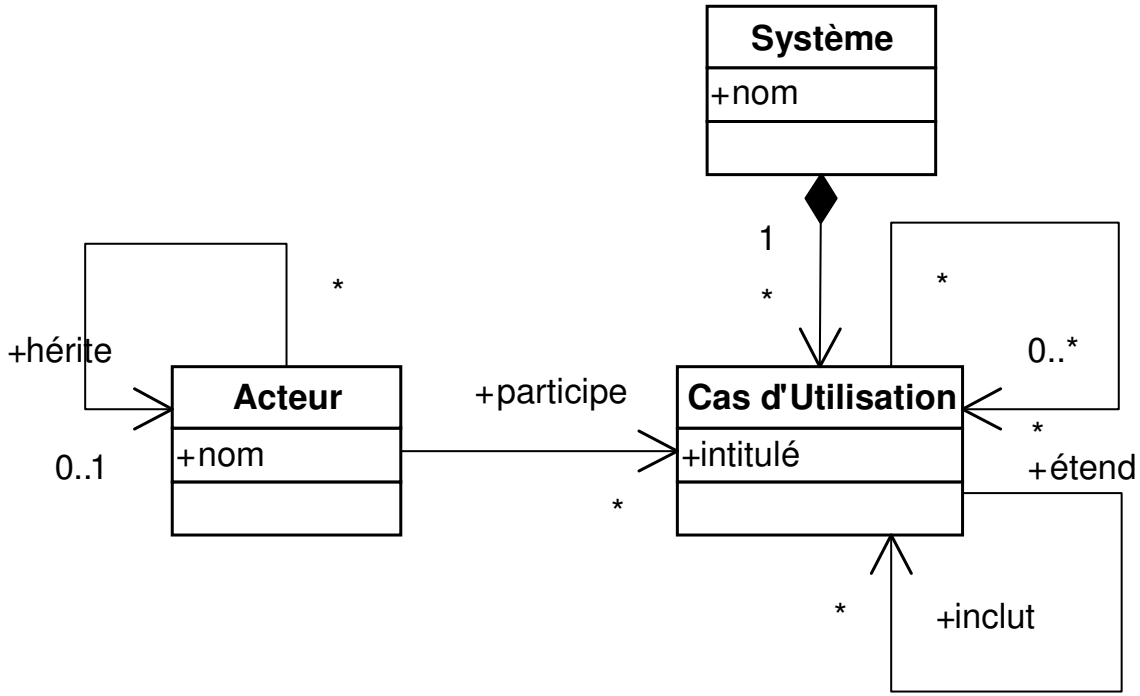
Cas d'utilisation : définition

- Un diagramme de cas d'utilisation contient des **acteurs**, un **système** et des **cas d'utilisation**
- Un acteur a un nom et est *relié* aux cas d'utilisation
- Un acteur peut *hériter* d'un autre acteur
- Un cas d'utilisation a un intitulé et peut *étendre* ou *inclure* un autre cas d'utilisation
- Le système a lui aussi un nom, et il inclut tous les cas d'utilisation

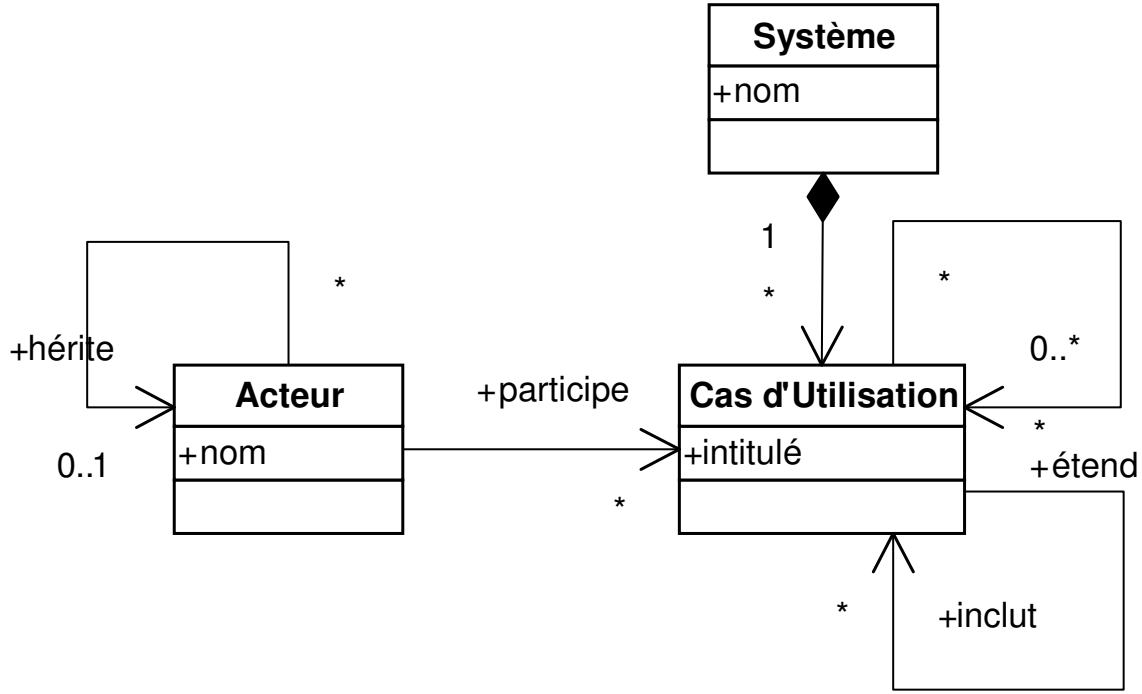
Cas d’utilisation : modèle (M1)

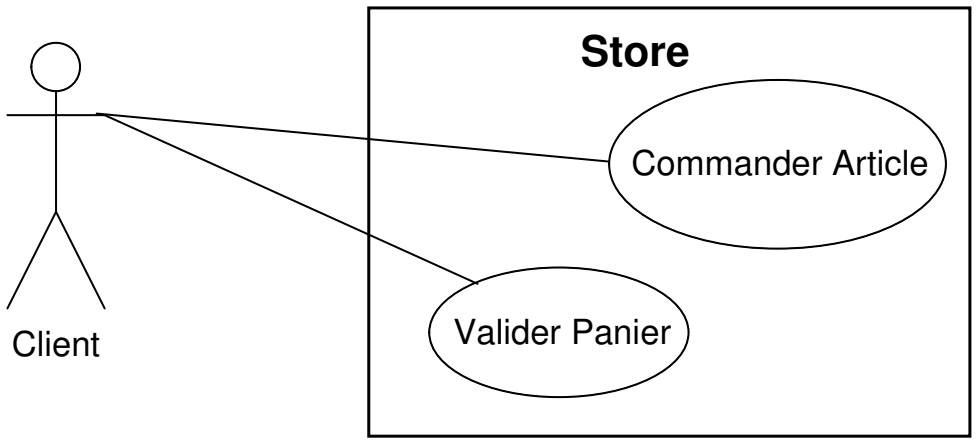


Cas d'utilisation : méta-modèle (M2)



Cas d'utilisation : correspondance M0 → M1





Cas d'utilisation : correspondance M0 → M1

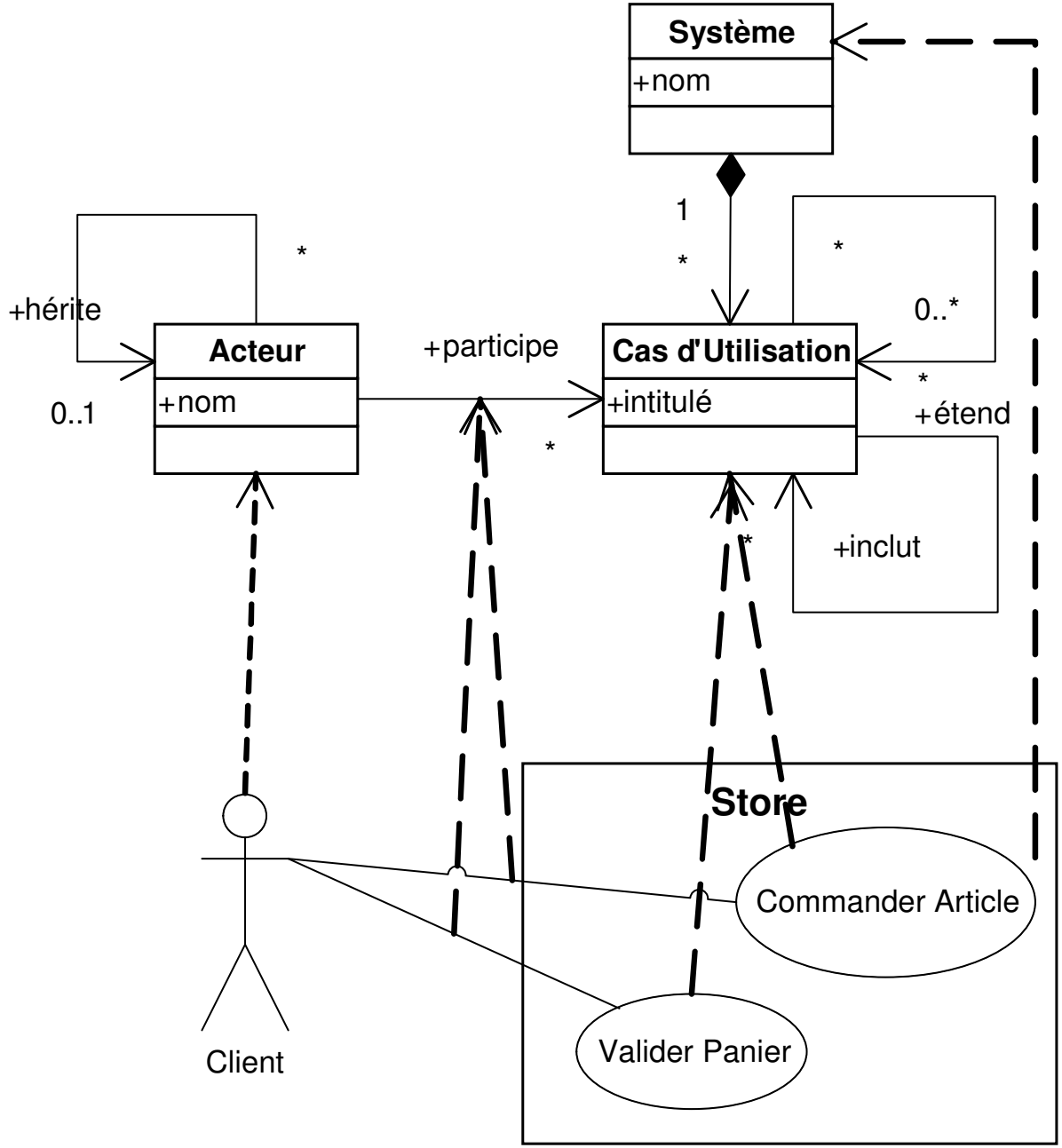


Diagramme de classes V1: définition

- Un diagramme de classes contient des **packages**
- Un packages a un nom et contient des **classes**. Un package peut *importer* un autre package
- Une classe a un nom et peut *contenir* des **attributs**. Une classe peut aussi *hériter* d'une autre classe
- Un attribut a un nom et une visibilité qui peut être soit public soit private. Un attribut *a* un **type** qui peut être soit un **type de base** (**string**, **integer**, **boolean**), soit une classe du diagramme

Diagramme de classes V1 : méta-modèle

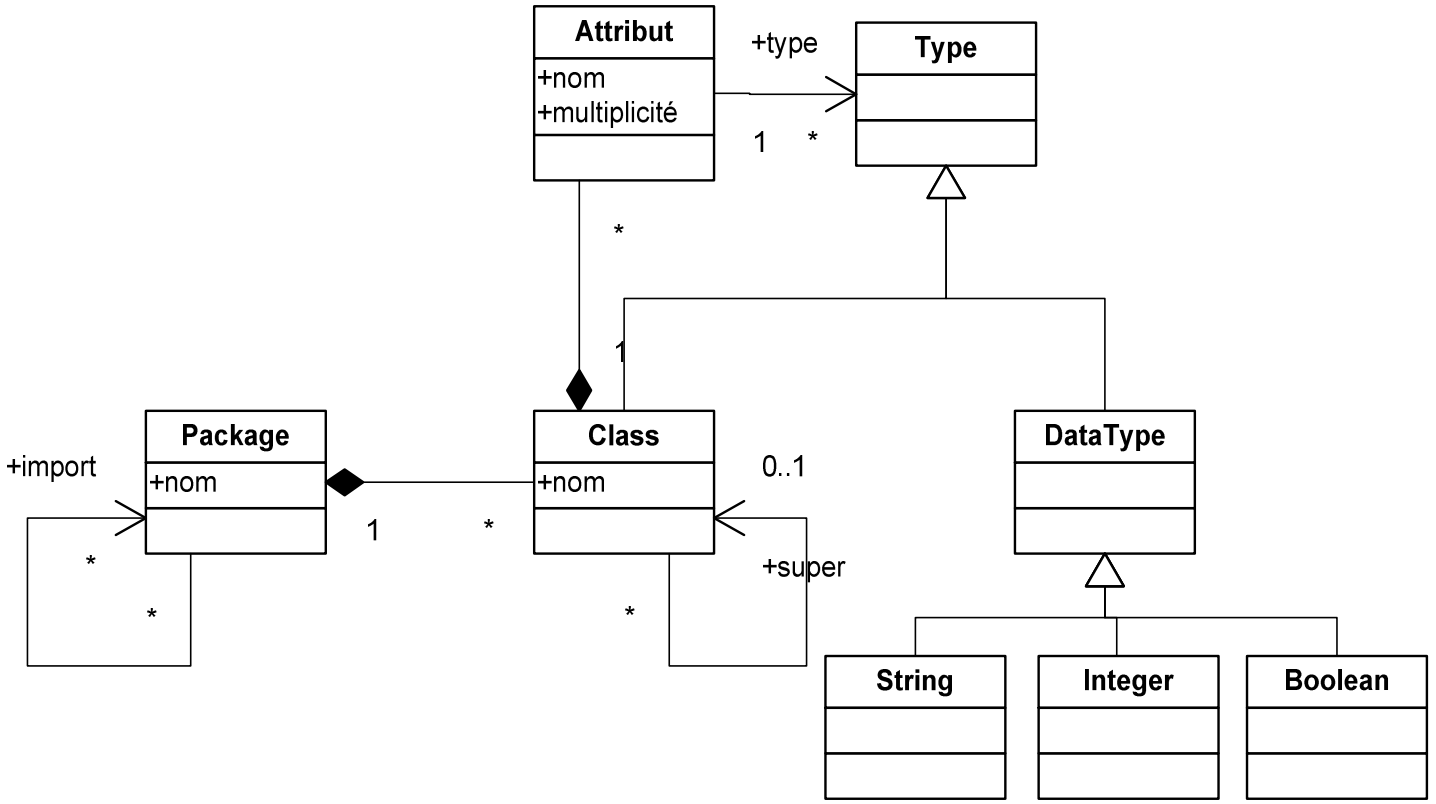
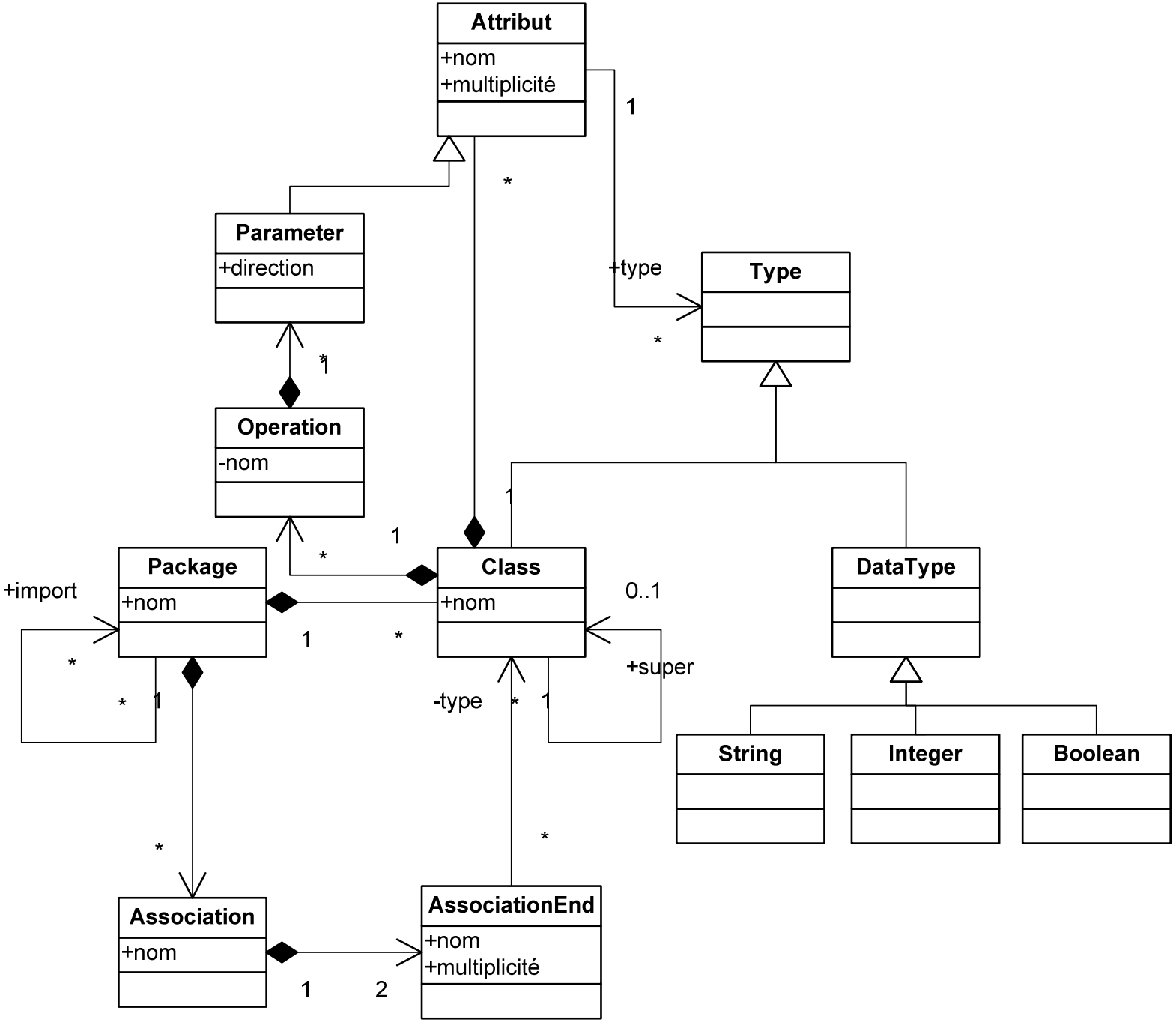


Diagramme de classes V2 : définition

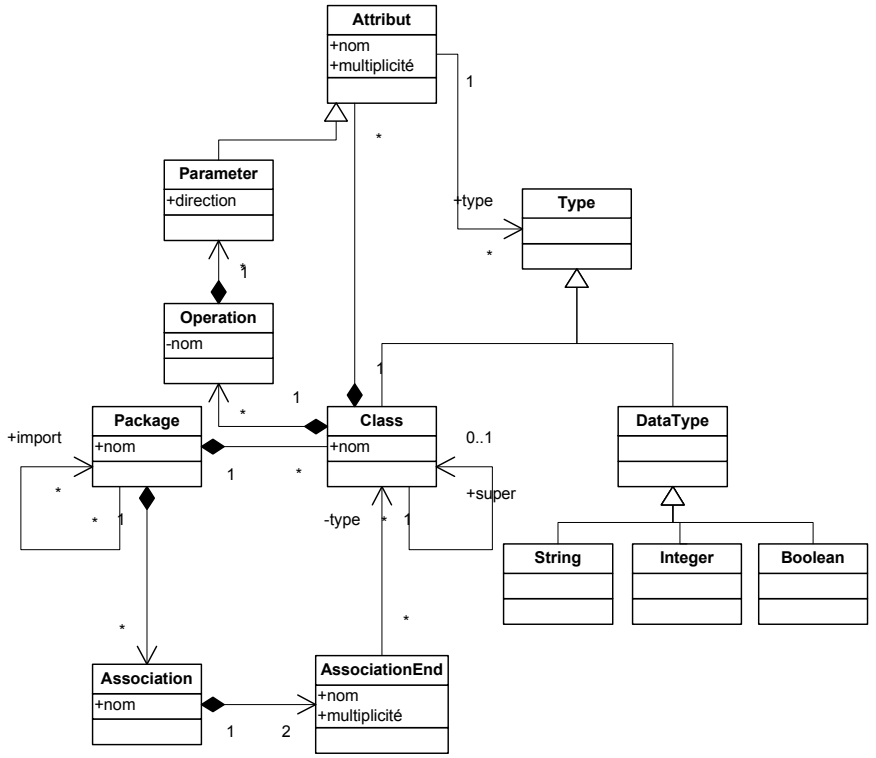
- Reprend la définition du diagramme de classes V1, plus:
- Une classe peut posséder 0 ou N **attributs**
- Un attribut possède un nom, une multiplicité et un type (classe ou type de donnée)
- Une classe peut contenir 0 ou N **opérations**
- Une opération possède un nom et peut contenir 0 ou N **paramètres**
- Un paramètre est un attribut qui possède en plus une direction (in, out, in/out)
- Une **association** définie une relation entre deux classes. Elle possède un nom et contient obligatoirement deux extrémités (**associationEnd**). Les extrêmités d'une association sont reliées aux classes dont l'association définie la relation
- Une extrêmité possède un nom et une multiplicité

Diagramme de classes V2 : Méta-modèle

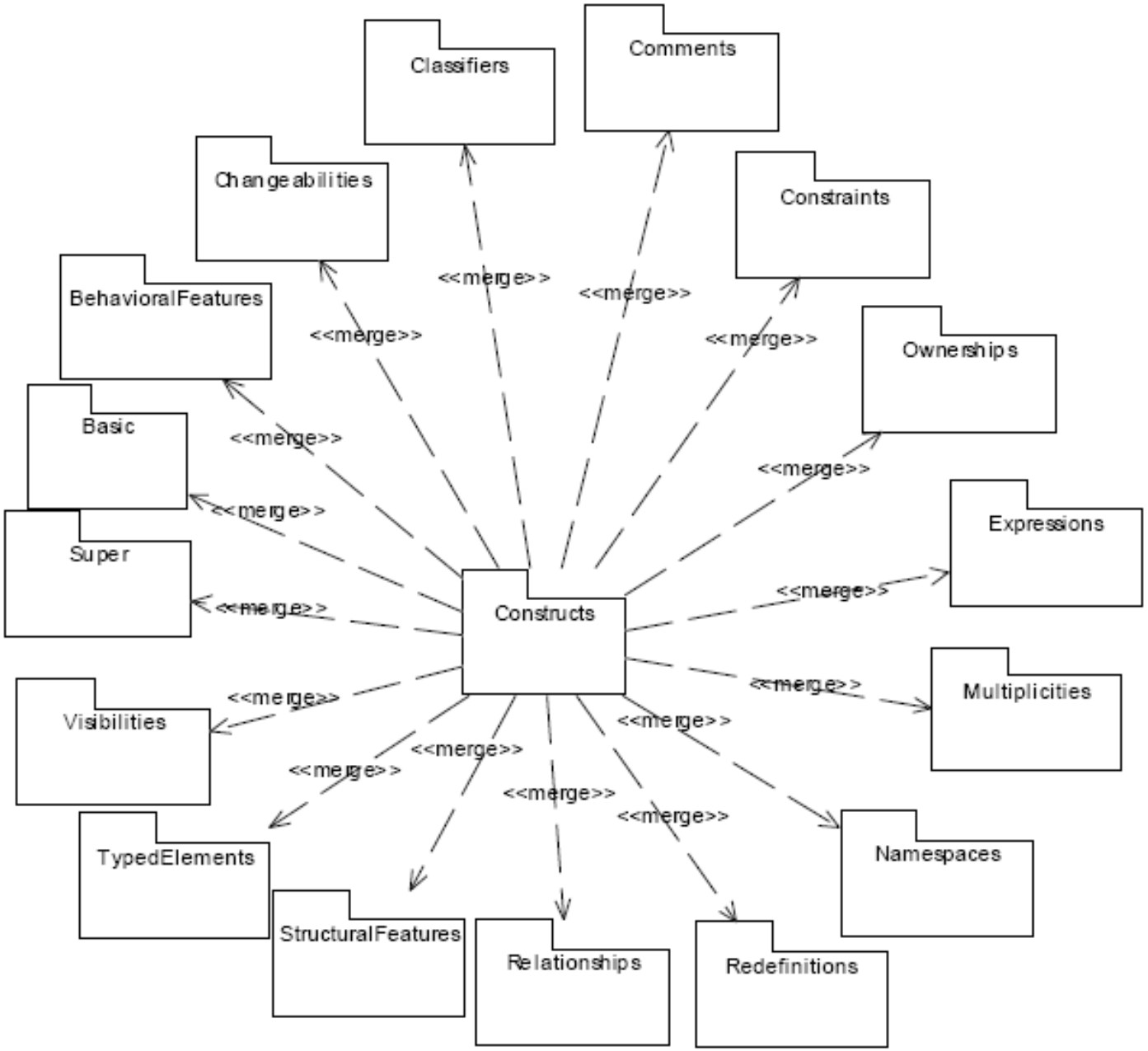


Le méta-méta-modèle MOF (M3)

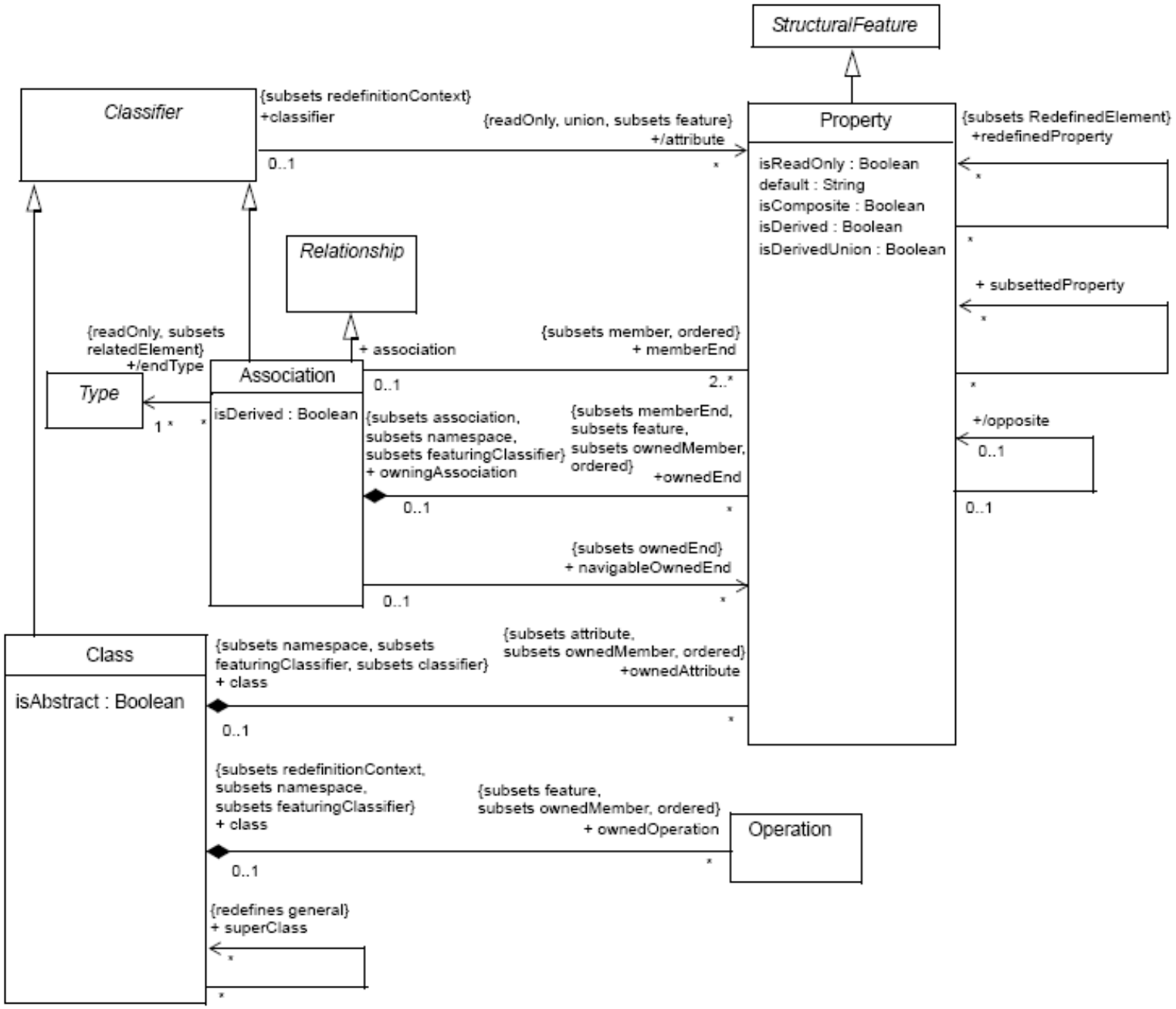
- Le MOF définit le langage permettant de définir des méta-modèles
- Les concepts du MOF sont les concepts de méta-classe, méta-attribut, méta-association, etc.
- MOF peut être défini à l'aide d'un diagramme de classe. Ce diagramme est le méta-méta-modèle
- Le méta-méta-modèle s'auto-définit.



Le méta-méta-modèle MOF : Décomposition en paquetages



Le méta-méta-modèle MOF : Partie du paquetage classifier



Eclipse Modeling Framework

(http://www.eclipse.org/emf)

EMF ?

- Eclipse Modeling Framework =
  Un environnement intégré, dédié à la modélisation
  - Définition de méta-modèles
  - Création de modèles satisfaisants les méta-modèles définis
- Un environnement de développement avec
  - génération automatique de code (squelettes de classes)
  - à partir de modèles
- Public ciblé
  - Destiné aux développeurs Java pour faciliter la construction d'applications à base de modèles structurés

Outils d'EMF

- Spécification de méta-modèles
- Générateur d'éditeurs de modèles
- Intégré à la plateforme Eclipse sous la forme de plugins
- Inter-opérabilité entre interfaces Java, UML et schéma XML (conversion de modèles)

Représentation des modèles (XMI)

Fonctionnement

- Le standard XMI (XML Metadata Interchange) permet le passage des modèles aux documents XML
- Il définit des règles permettant de construire des schéma XML à partir de méta-modèle
- Ainsi il est possible d'encoder un modèle dans un document XML

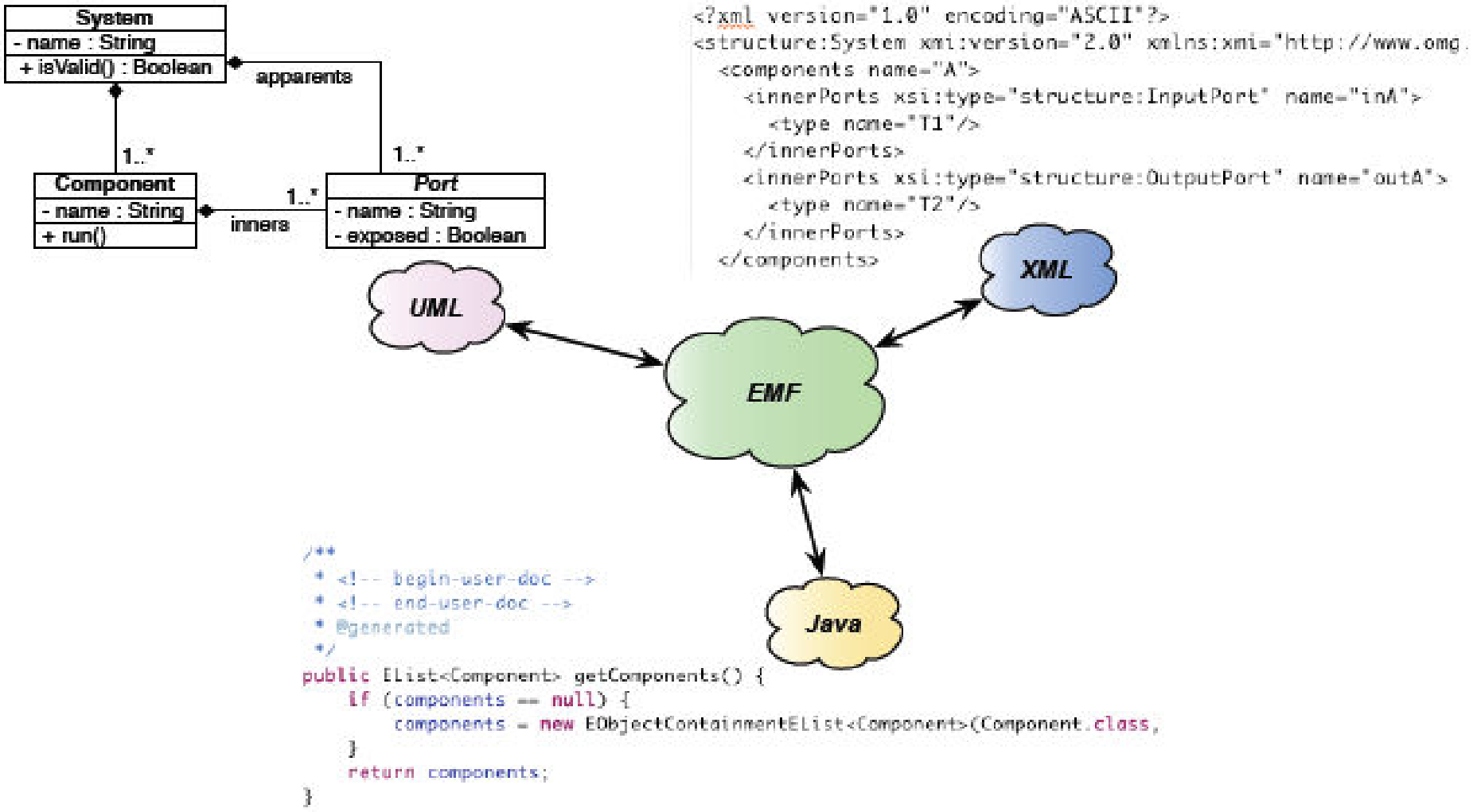
XMI et UML

- XMI se décline en 6 versions et UML se décline en 4 versions
  - Cela explique pourquoi l'échange de modèle UML en XMI pose quelques problèmes

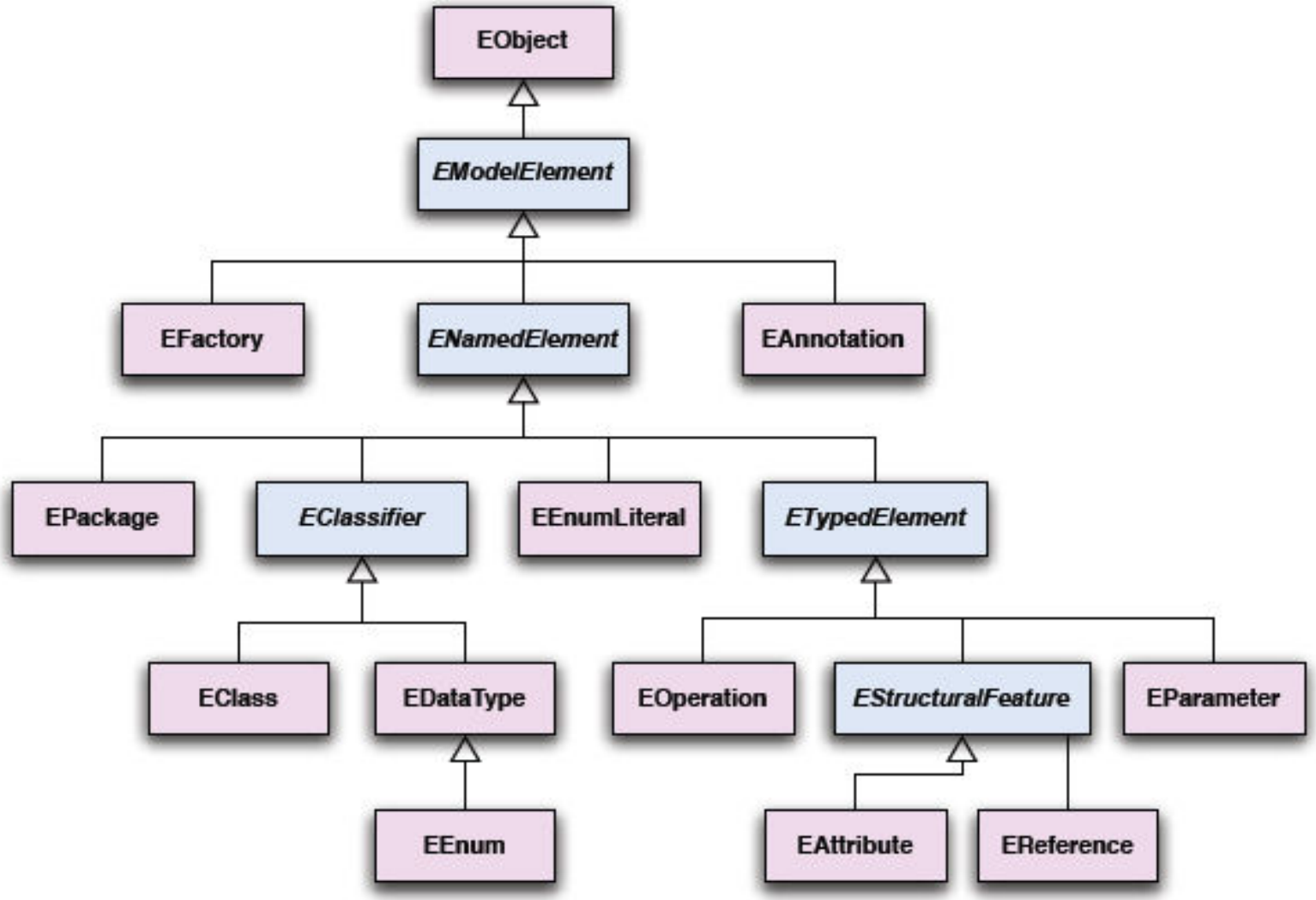
Manipulation de modèles

- La représentation des modèles en XMI souffre des lacunes du standard
- Besoin d'un autre format de représentation des modèles permettant leur manipulation dans les langages de programmation
- EMF (comme JMI) fournit un ensemble d'interfaces Java offrant les opérations nécessaires à la manipulation des modèles

EMF : lien entre des technologies

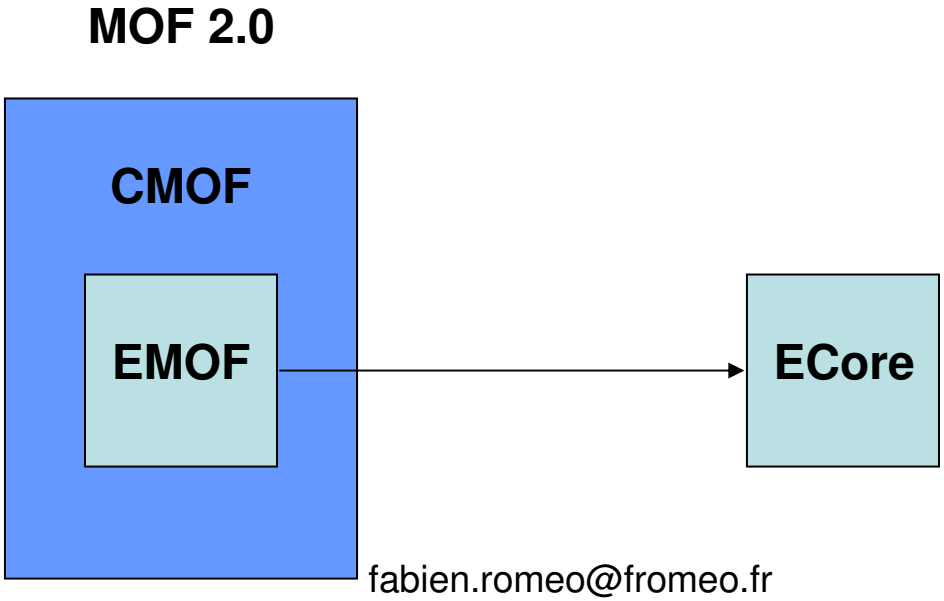


ECore : Le méta-méta-modèle d'EMF

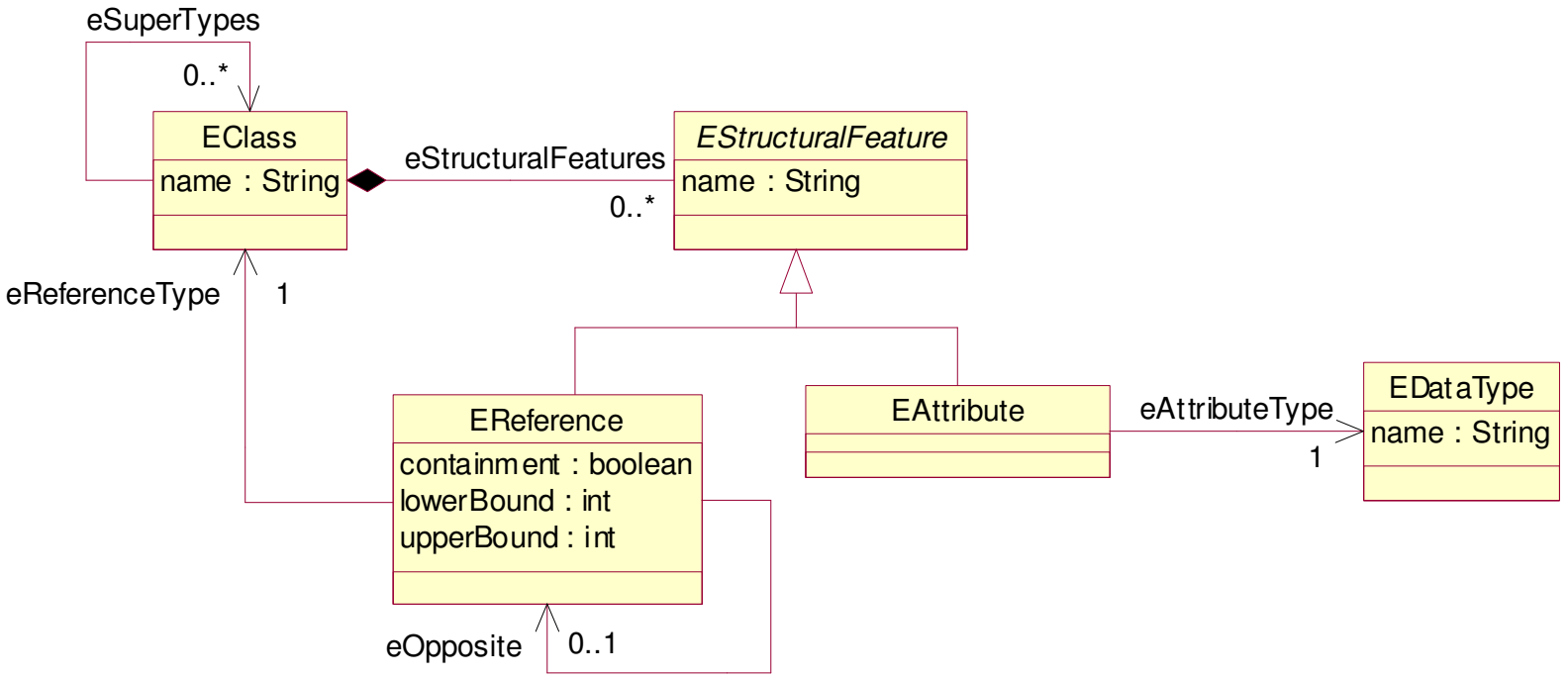


MOF vs. ECore

- EMOF (Essential MOF) sous-ensemble du MOF sans la notion de méta-association
- CMOF (Complete MOF) complète le CMOF avec les méta-associations

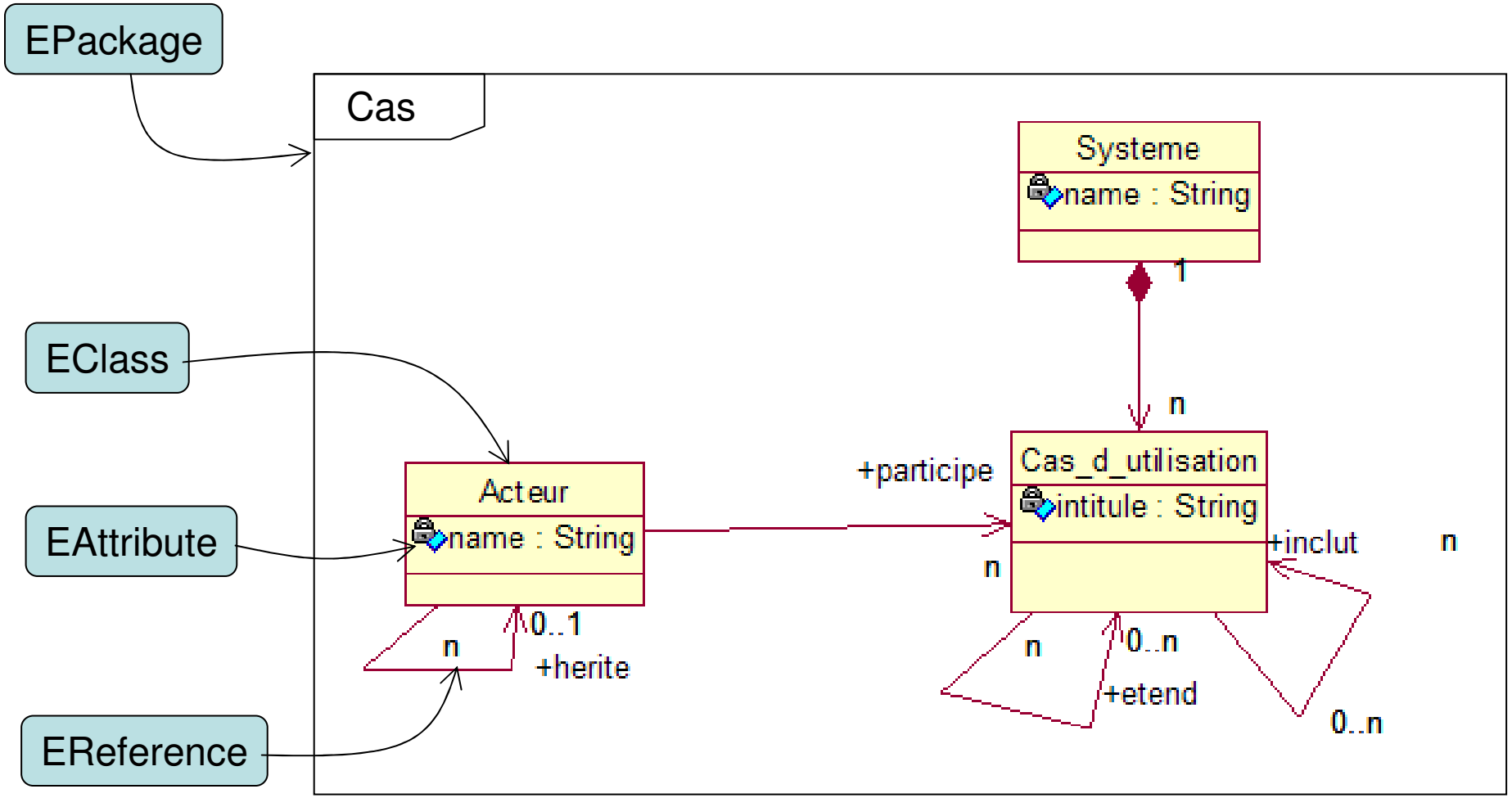


ECore : Extraits



(proche de notre méta-modèle UML V1)

Méta-modèle exemple sous EMF



API de manipulation de modèles

- Interfaces taylored
  - Interfaces adaptées à la manipulation d'un type de modèle particulier
- Interfaces réflectives
  - Interfaces utilisables sur tout type de modèle
  - Proposent des opérations indépendantes de la structure des modèles
  - Permettent d'obtenir des informations sur le méta-modèle d'un modèle
  - Permettent de connaître dynamiquement la structure d'un modèle

Interfaces réflectives d'EMF (1)

- EObject
  - Représente n'importe quel élément appartenant à un modèle ou un méta-modèle
  - Fournit la méthode eClass() qui permet d'obtenir l'EClass de l'élément
  - eGet() et eSet() permettent de lire et d'écrire les valeurs des propriétés de l'élément (attributs et références)
- EClass
  - Représente une méta-classe d'un méta-modèle
  - getEAttributes() et getEReferences() permettent d'obtenir la liste de tous les attributs et références contenus dans la méta-classe
  - getEStructuralFeature() permet d'obtenir une propriété (attribut ou référence) d'une EClass à partir de son nom

Interfaces réflectives d'EMF (2)

- EPackage
  - Représente le moyen d'accès à toutes les EClass définies dans un package
  - getEClassifier(String name) permet de récupérer la référence vers une EClass d'un métamodèle à partir de son nom
- EFactory
  - Représente le moyen de créer des instances des EClass définies dans un package
  - Fournit l'opération create() qui permet de créer une instance d'une EClass

Exemple interfaces réflectives

```
EFactory fact = //obtention de la référence
EPackage pa = //obtention de la référence
EClass acEC = (EClass)pa.getEClassifier("Acteur");
EObject ac = fact.create(acEC);
ac.eSet(acEC.getEStructuralFeature("nom"),"Client");
EClass cuEC = (EClass)pa.getEClassifier("casutil");
EObject cu1 = fact.create(cuEC);
cu1.eSet(cuEC.getEStructuralFeature("intitule"),"com
  mander panier");
EObject cu2 = fact.create(cuEC);
cu2.eSet(cuEC.getEStructuralFeature("intitule"),"val
  ider panier");
Collection parti =
  (Collection)ac.eGet(acEC.getEStructuralFeature("pa
  rticipe"));
parti.add(cu1); parti.add(cu2);
// …
```

Règles de génération d'interfaces taylored (1)

- Règles EClass
  - Une EClass donne lieu à la création d'une seule interface avec les caractéristiques :
    - A pour nom le nom de l'EClass
    - Offre des opérations de lecture et d'écriture pour chaque EAttribute de l'EClass
    - Offre des opérations de lecture et d'écriture pour chaque EReference de l'EClass
    - Hérite de l'interface réflective EObject

Règles de génération d'interfaces taylored (2)

- Règles EPackage
  - Une EClass donne lieu à la création de 2 interfaces : une interface Factory et une interface Package
  - Caractéristique de l'interface Factory :
    - A pour nom : nom_packageFactory
    - Offre une opération de création pour chaque EClass contenue dans l'EPackage
    - Hérite de l'interface réflective EFactory

Règles de génération d'interfaces taylored (3)

- Règles EPackage (suite)
  - Caractéristique de l'interface Package :
    - A pour nom : nom_packagePackage
    - Offre une opération de navigation pour chaque EClass contenue dans l'EPackage permettant d'obtenir l'EClass correspondante
    - Hérite de l'interface réflective EPackage

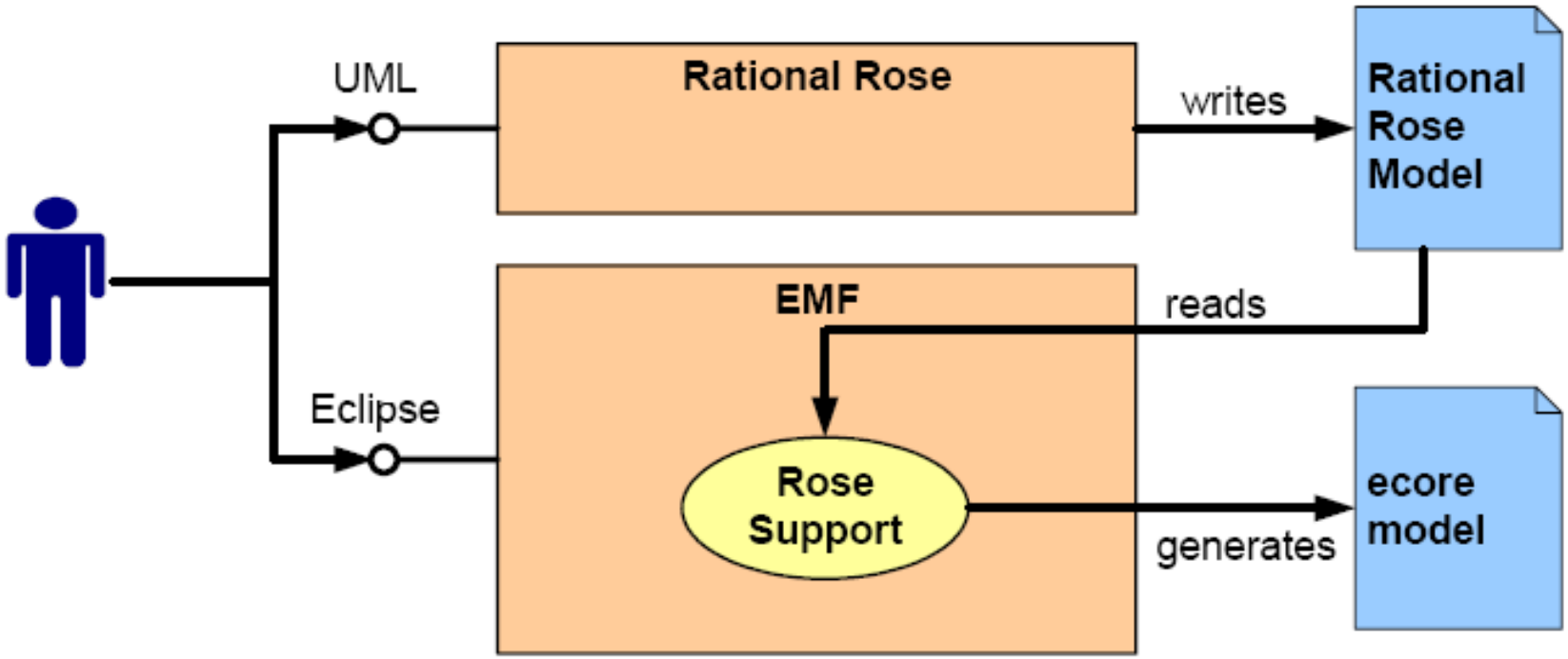
Exemple interfaces taylored

```
CasFactory fact = CasFactory.eINSTANCE;
Acteur ac = fact.createActeur();
ac.setNom("Client");
Cas_d_utilisation cau1 =
   fact.createCas_d_utilisation();
cau1.setIntitule("commander panier");
Cas_d_utilisation cau2 =
   fact.createCas_d_utilisation();
cau2.setIntitule("valider panier");
ac.getParticipe().add(cau1);
ac.getParticipe().add(cau2);
System sys = fact.createSysteme();
sys.setNom("Store");
sys.getCas().add(cau1);
sys.getCas().add(cau2);
```

Importation de modèles (1)

- UML
  - Rational Rose : fichiers .mdl
  - Eclipse UML2 (Omondo) : fichiers .uml2
- XML
  - XML shema moins expressif que ECore
  - Conversion des schémas XML en méta-modèles ECore
    - Un schéma → EPackage
    - Un type complexe → EClass
    - Un type simple → EDataType
    - Un attribut → EAttribut (pour un EDataType)
      ou EReference (pour un EClass)

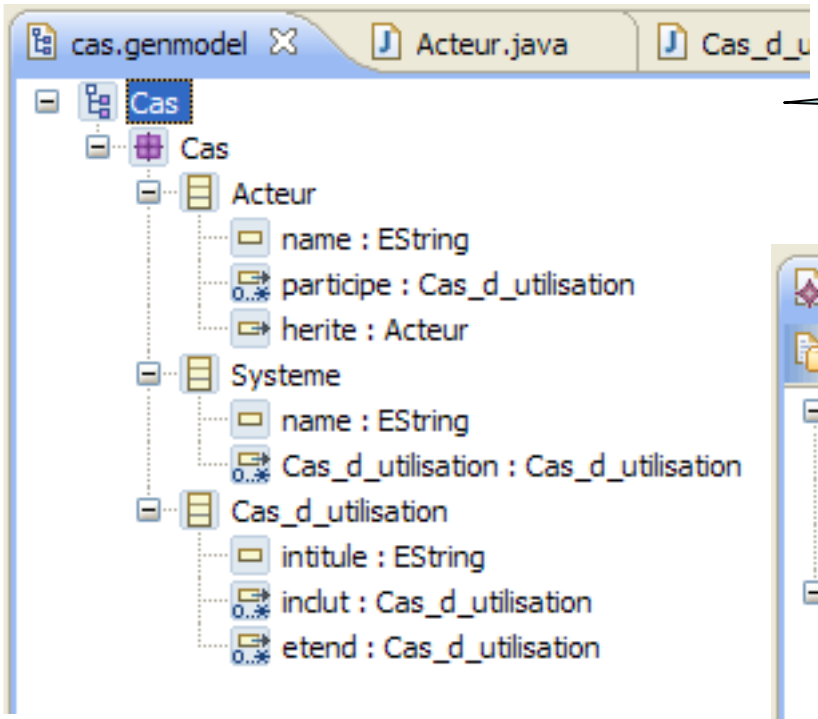
Import avec Rational Rose



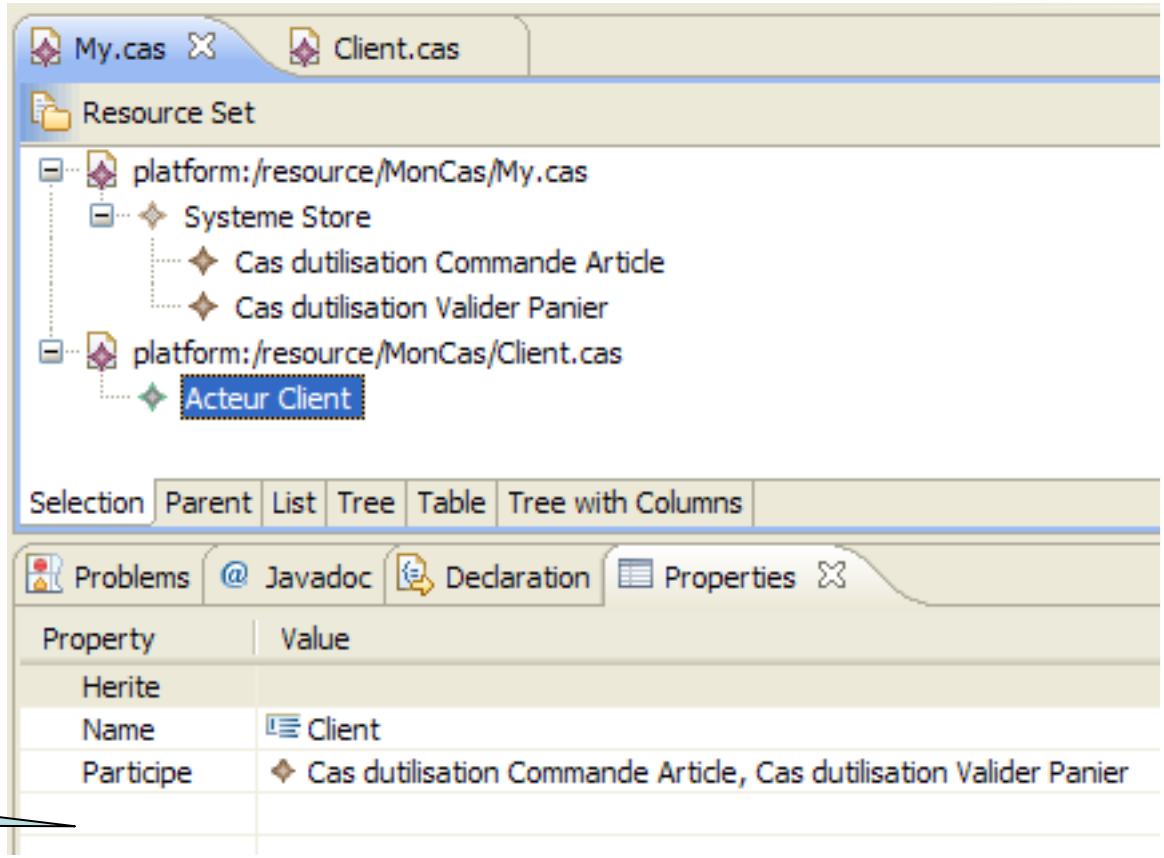
Importation de modèles (2)

- Java
  - Création d'un méta-modèle ECore à partir d'un modèle constitué d'interfaces Java annotées :
    - Hiérarchie des packages Java contenant les interfaces → hiérarchie de EPackage
    - Interface Java taguée @model → EClass
    - Les EReference, EAttribute sont générés pour chaque get*() d'une interface
    - Les EOperation sont générées pour n'importe quelle autre méthode
    - Ajout de propriétés @model [<property>='value'] (abstracness, interface, multiplicity, …)

Génération d'éditeur graphique



Méta-modèle des diagrammes de cas d'utilisations

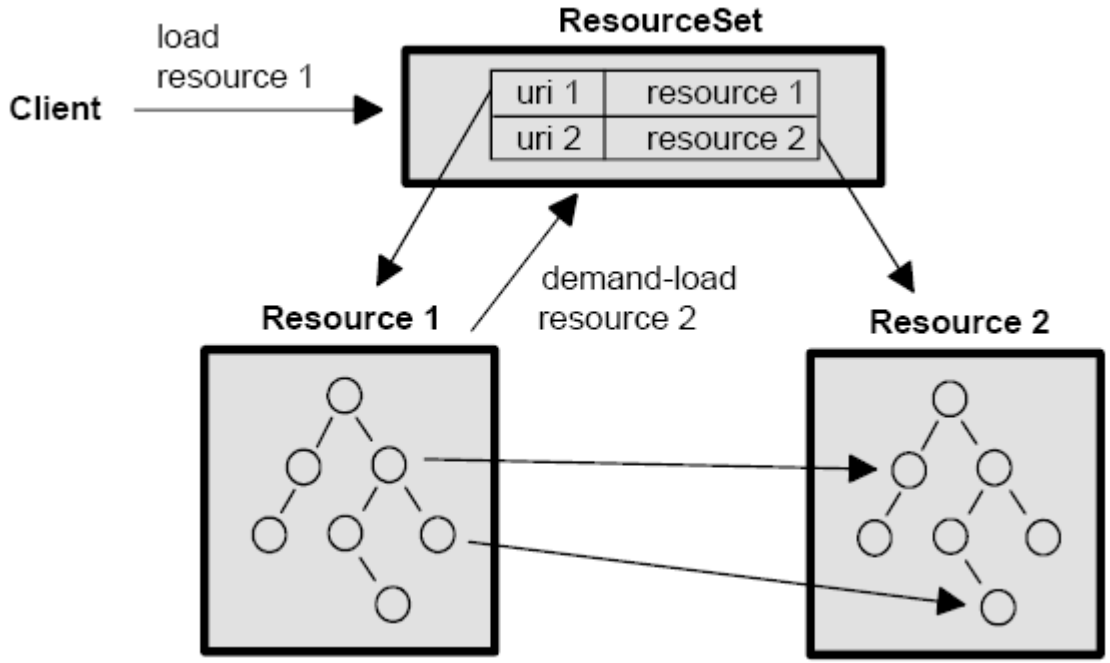


Editeur généré à partir du méta-modèle

Eclipse Modeling Framework (suite)

(http://www.eclipse.org/emf)

Persistance et Sérialisation



- Une donnée sérialisée = Une *Resource*
- Les *Resources* sont enregistrées dans un *ResourceSet*
- Chargement explicite d'une *Resource* par le client
- Chargement à la demande des autres *Resources* lors de la résolution des références

Chargement

```
// Create a resource set to hold the resources.
ResourceSet resourceSet = new ResourceSetImpl();

// Register the appropriate resource factory to handle all file extensions.
resourceSet.getResourceFactoryRegistry().getExtensionToFactoryMap().put
(Resource.Factory.Registry.DEFAULT_EXTENSION,
 new XMIResourceFactoryImpl());

// Demand load resource for this file.
Resource resource =
    resourceSet.getResource(URI.createFileURI(absolutePath), true);

// use reflective interfaces
for (EObject eObject : resource.getContents()) {...}

// or use taylored interfaces
Schema root = (Schema) resource.getContents().get(0);
for (Table t : root.getTables()) {...}
```

Sauvegarde

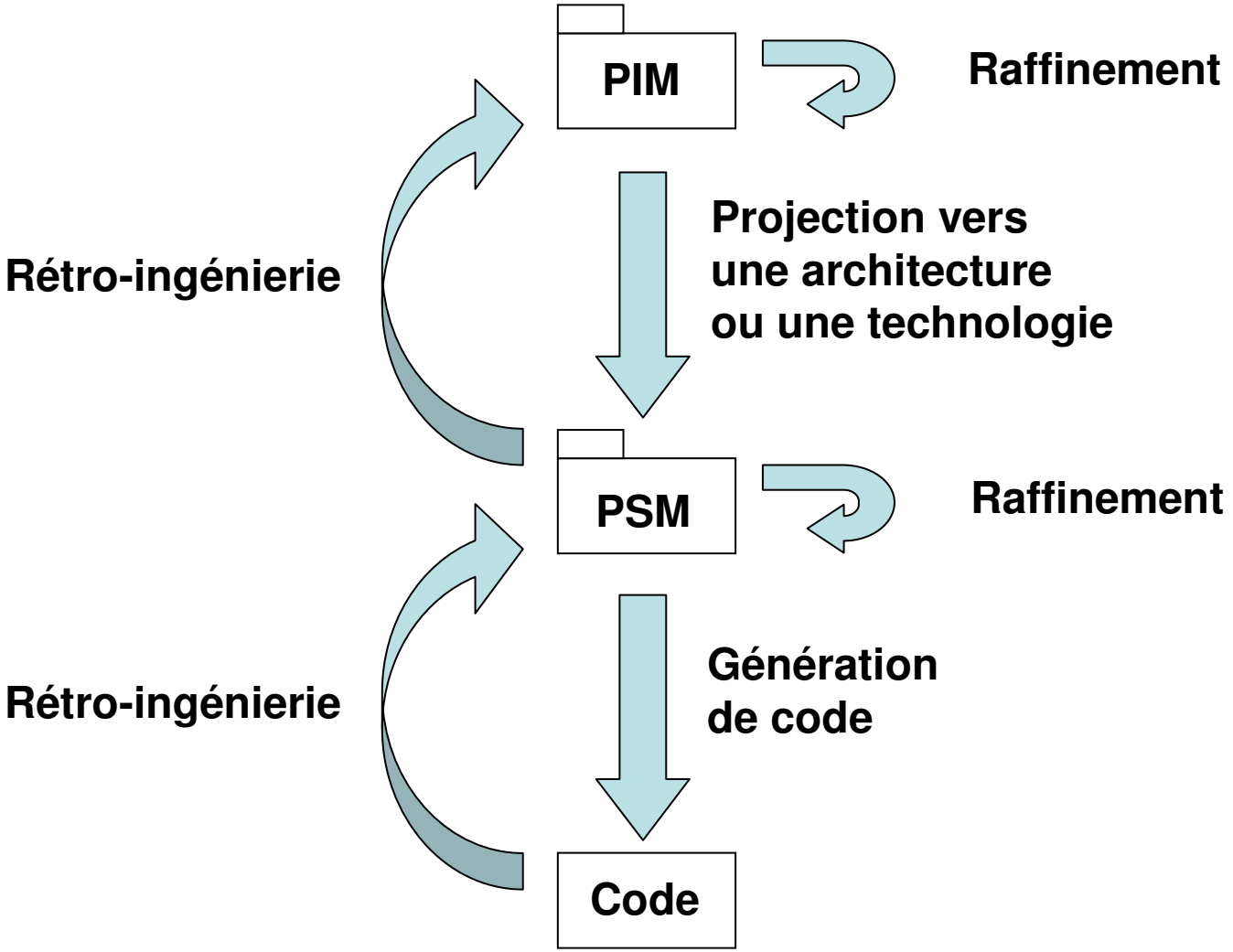
```
// Create a resource for this file.
URI fileURI = URI.createFileURI(absolutePath);
Resource resource = new XMIResourceFactoryImpl().createResource(fileURI);

// use reflective interfaces
EFactory fact = ...

// or use taylored interfaces
ShemaFactory fact = SchemaFactory.eINSTANCE()
Shema = fact.createSchema();
...

// save the resource
resource.getContents().add(shema);
try {
    resource.save(Collections.EMPTY_MAP);
} catch (IOException e) {
    e.printstackTrace();
}
```

Transformation de modèles

Introduction

- Les transformations de modèles sont le cœur des aspects de production de MDA
  - PIM vers PSM, PSM vers code (sens inverse).
- Les transformations de modèles sont basées sur les métamodèles
  - Toute classe UML se transforme en une classe Java
- Les constructeurs de plate-forme devraient produire les transformateurs permettant d'exploiter les plates-formes
  - UML vers EJB, UML vers .NET, UML vers
- Les sociétés devraient pouvoir adapter les transformateurs selon leurs propres expériences et leurs propres besoins
  - Ex: ne pas utiliser de composant EJB Entity!
- Actuellement les transformations de modèles peuvent s'écrire selon trois approches
  - Programmation, Template, Modélisation

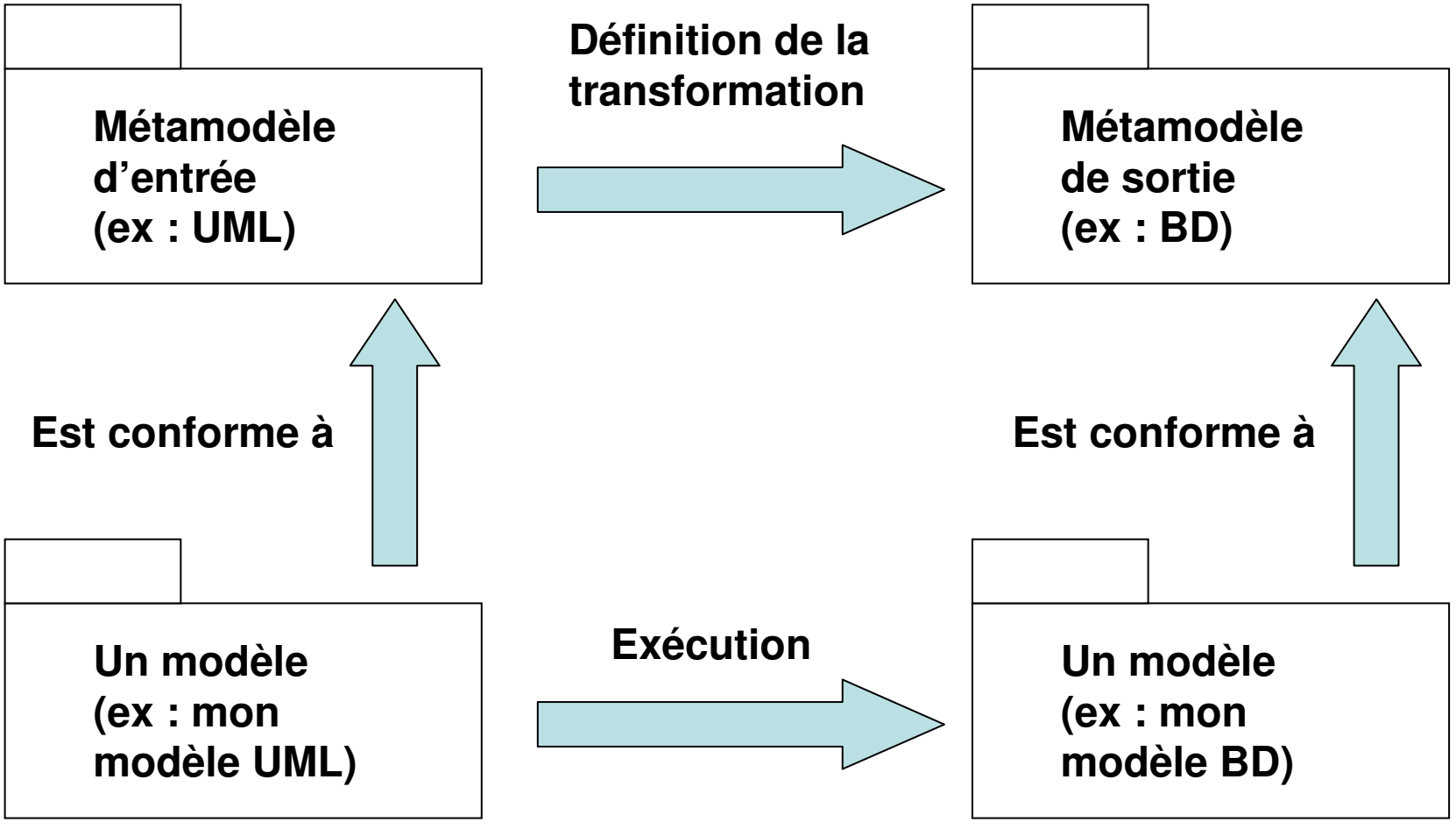
Principales transformations de MDA



Transformation et MDA

- Transformations de modèles PIM vers PIM
  - Raffinement de PIM pour améliorer la précision des informations
  - Ex : création de classes UML à partir d'un ensemble de diagrammes de séquences ou création de diagrammes d'états à partir de diagrammes de classes
  - Permet d'accélérer la production de PIM (outils)
- Transformations de modèles PIM vers PSM
  - Construction d'une bonne partie des PSM à partir des PIM
  - Garantissent la pérennité des modèles et leur productivité
- Transformations de modèles PSM vers code
  - Permettent de générer (la totalité) du code

Métamodèles et transformation de modèles

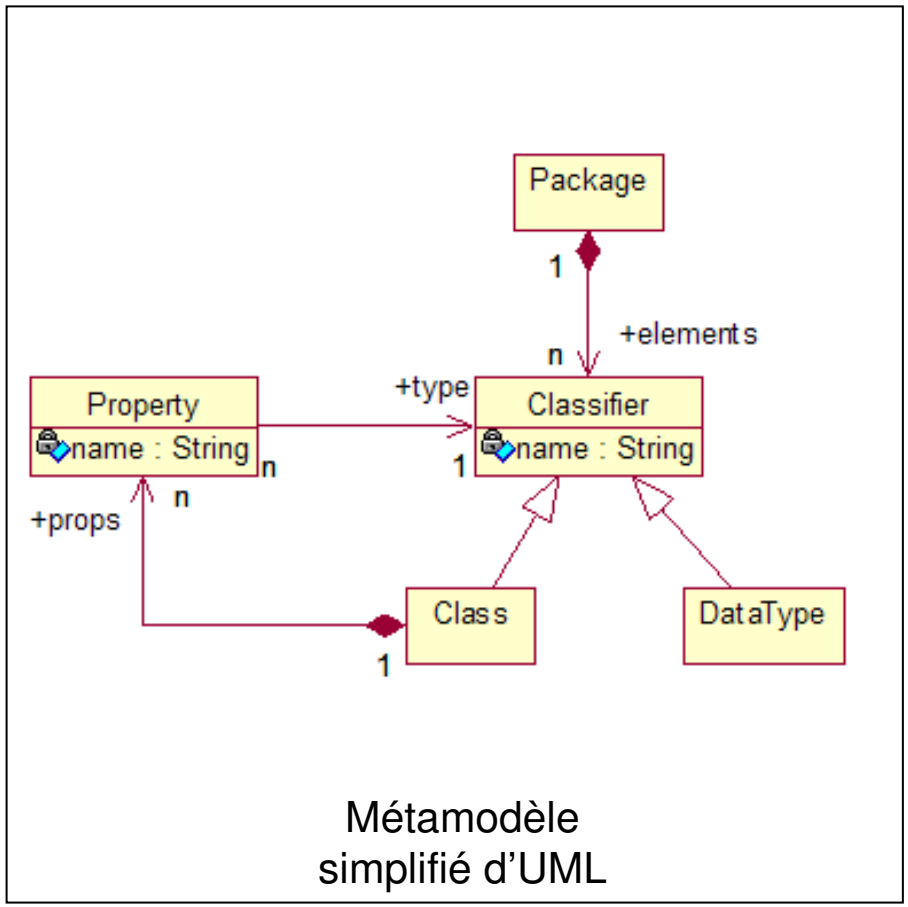


Spécification des règles de transformation

- Approche par programmation
  - Utilisation de langage objet pour programmer les transformations
  - Ex : manipulation de modèles en Java avec EMF
- Approche par template
  - Définition de canevas pour les modèles cibles paramétrés par des infos des modèles sources
  - Ex : JET (JSP-like) dans le framework Eclipse
- Approche par modélisation
  - Application des concepts de l'IDM aux transformations elles-mêmes
  - Ex : MOF2.0 QVT (Query,View,Transformation)

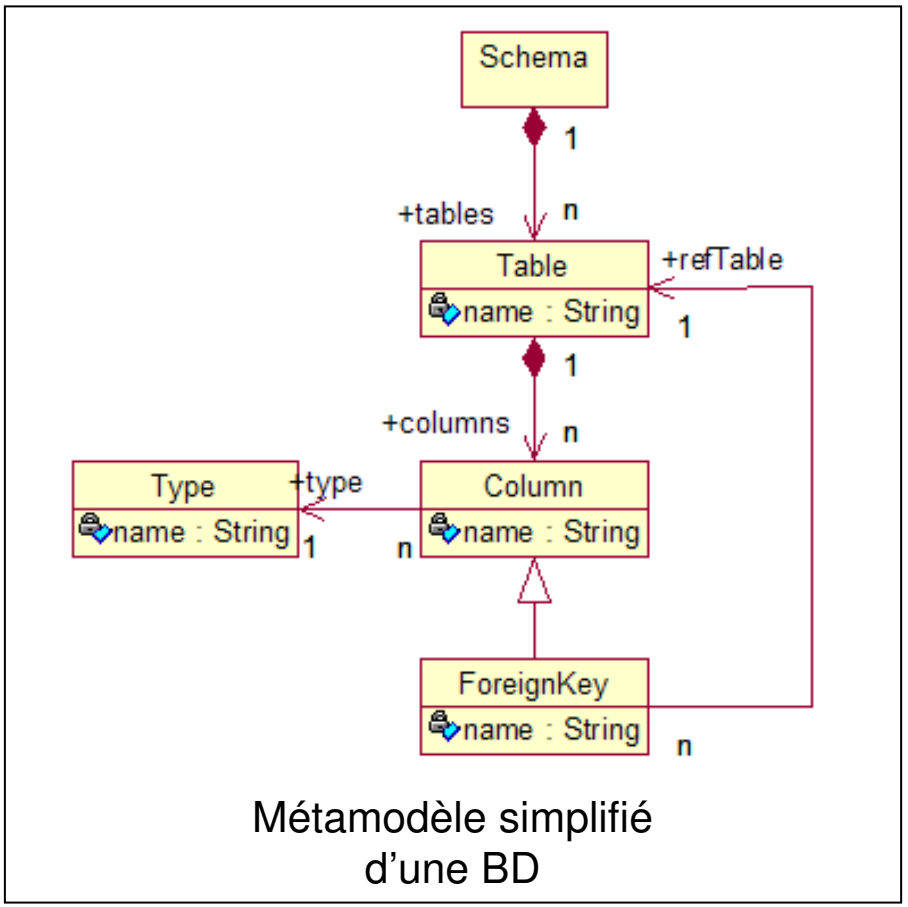
Approche par programmation : exemple de mise en œuvre

Source de la transformation



Métamodèle simplifié d'UML

Cible de la transformation



Métamodèle simplifié d'une BD

Définition des règles de la transformation UML vers BD

1. A tout package UML correspond un schéma
2. A toute classe UML du package correspond une table dans le schéma
3. A toute propriété UML d'une classe correspond une colonne dans la table
   - Si le type de la propriété est une classe UML, la colonne est une clé étrangère
   - Si le type de la propriété est un type de donnée UML, la colonne est du type correspondant
4. A tout type de donnée UML d'un package correspond un type de donnée dans le schéma

Elaboration des règles par programmation : règle 1

```
public void Package2Schema(Package p) {
   Schema sh =
   DatabaseFactory.eINSTANCE.createSchema();
   for (Classifier c : p.getElements()) {
       if (c instanceof Class) {
         sh.getTables().add(class2Table((Class) c));
       }
       else if (c instanceof DataType) {
         dataType2Type((DataType) c);
       }
   }
}
```

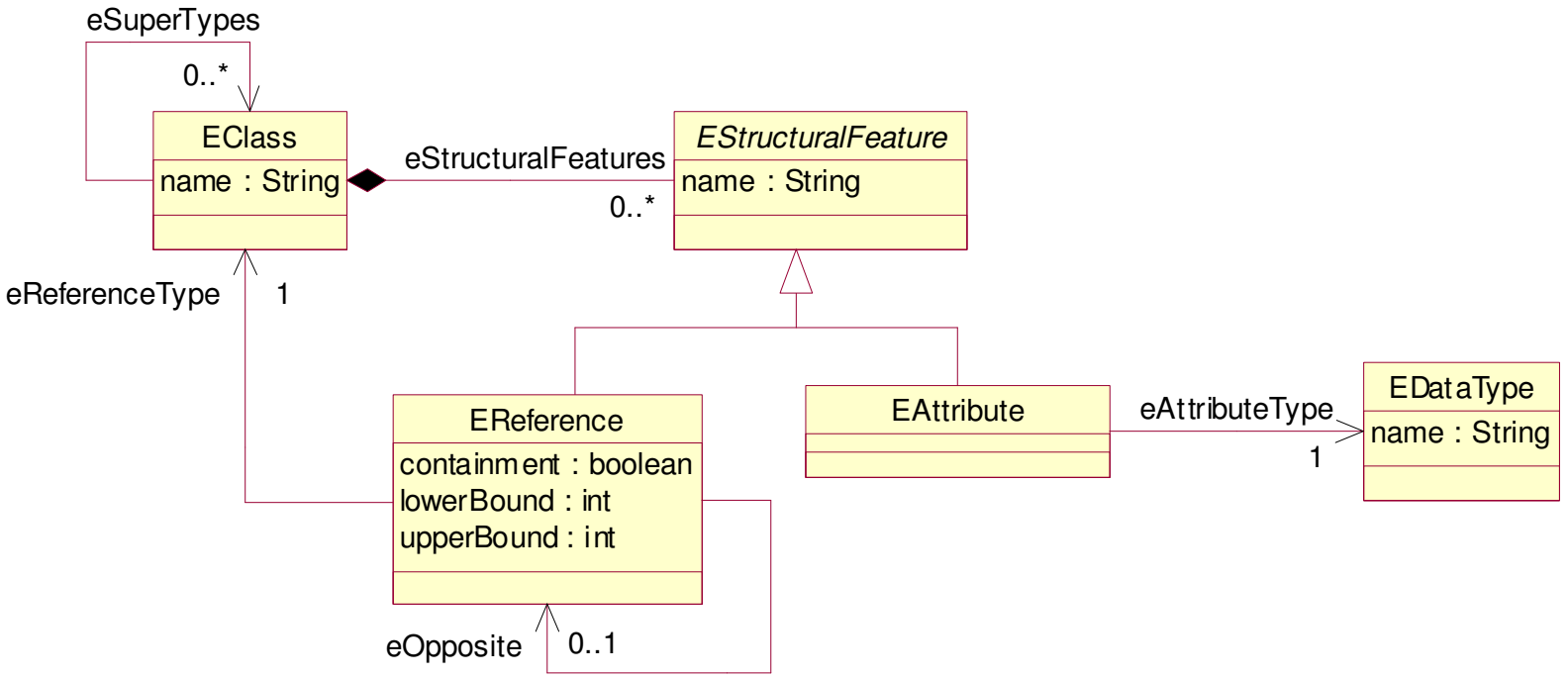
Elaboration des règles par programmation : règles 2, 3 et 4

```
public Table class2Table(Class c) {
    //TODO
}
public Column property2Column(Property p) {
    //TODO
}
public Type dataType2Type(DataType dt) {
    //TODO
}
```

Eclipse Modeling Framework (suite)

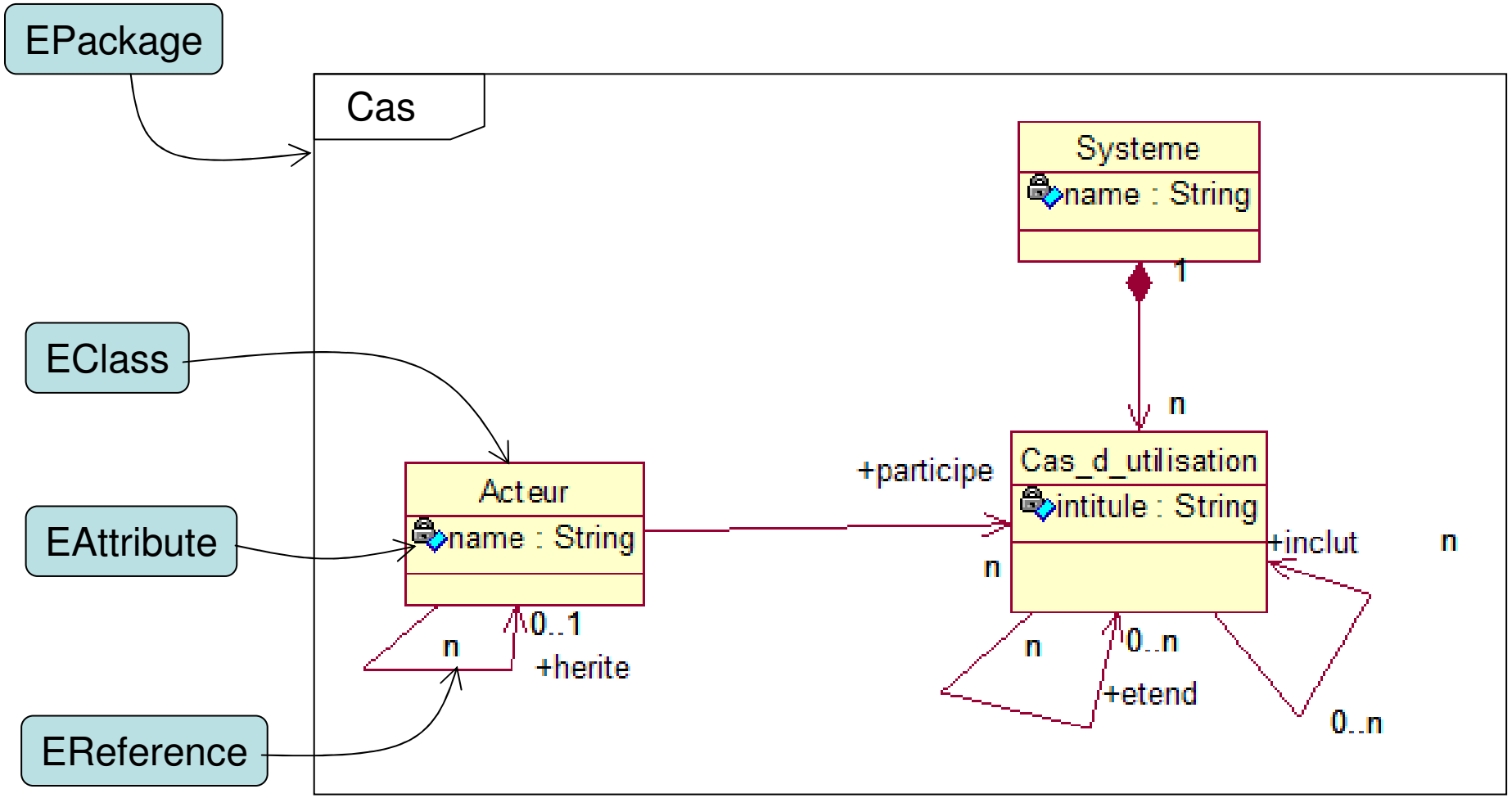
(http://www.eclipse.org/emf)

ECore : Extraits



(proche de notre méta-modèle UML V1)

Méta-modèle exemple sous EMF



API de manipulation de modèles

- Interfaces taylored
  - Interfaces adaptées à la manipulation d'un type de modèle particulier
- Interfaces réflectives
  - Interfaces utilisables sur tout type de modèle
  - Proposent des opérations indépendantes de la structure des modèles
  - Permettent d'obtenir des informations sur le méta-modèle d'un modèle
  - Permettent de connaître dynamiquement la structure d'un modèle

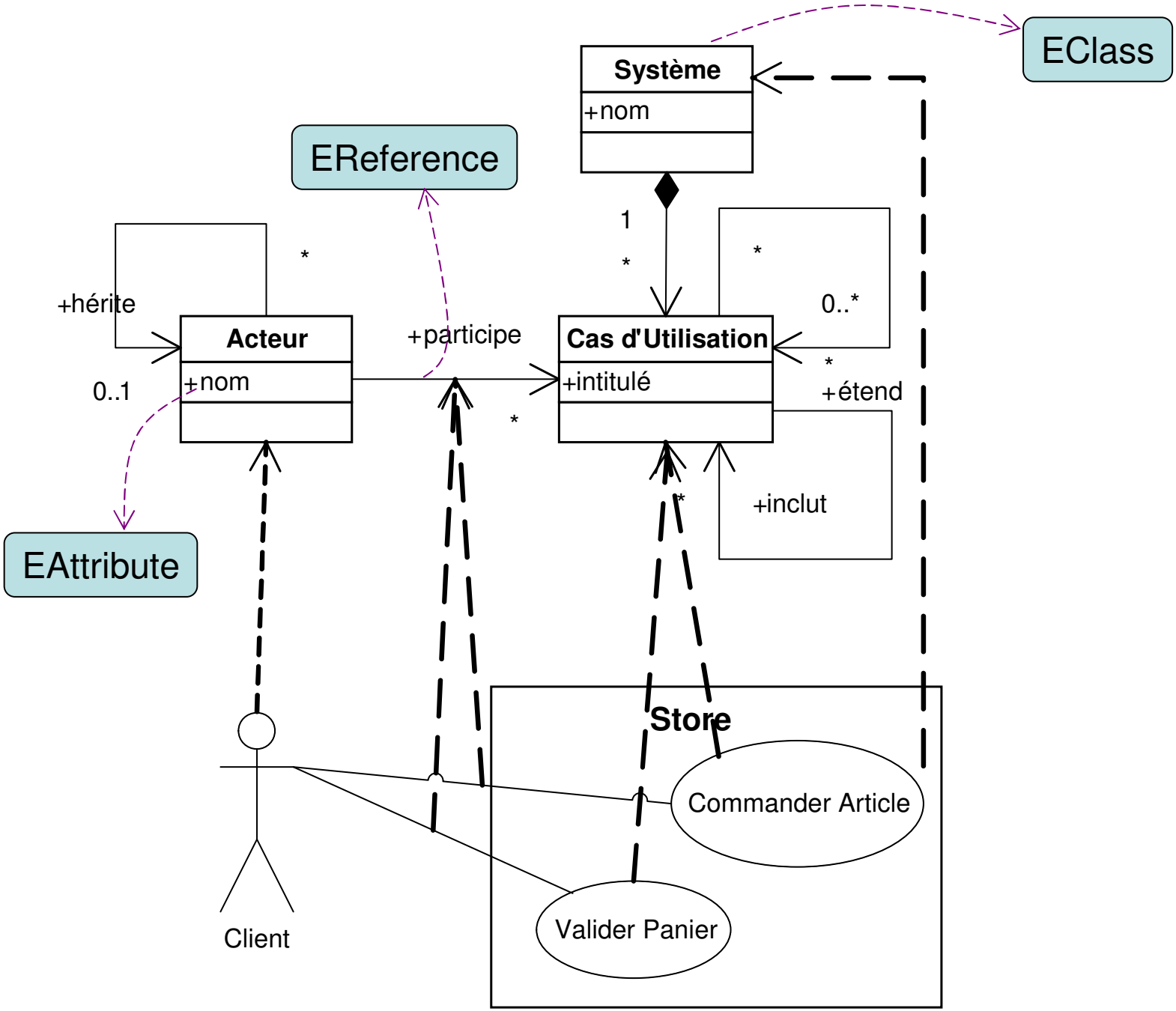
Interfaces réflectives d'EMF (1)

- EObject
  - Représente n'importe quel élément appartenant à un modèle ou un méta-modèle
  - Fournit la méthode eClass() qui permet d'obtenir l'EClass de l'élément
  - eGet() et eSet() permettent de lire et d'écrire les valeurs des propriétés de l'élément (attributs et références)
- EClass
  - Représente une méta-classe d'un méta-modèle
  - getEAttributes() et getEReferences() permettent d'obtenir la liste de tous les attributs et références contenus dans la méta-classe
  - getEStructuralFeature() permet d'obtenir une propriété (attribut ou référence) d'une EClass à partir de son nom

Interfaces réflectives d'EMF (2)

- EPackage
  - Représente le moyen d'accès à toutes les EClass définies dans un package
  - getEClassifier(String name) permet de récupérer la référence vers une EClass d'un métamodèle à partir de son nom
- EFactory
  - Représente le moyen de créer des instances des EClass définies dans un package
  - Fournit l'opération create() qui permet de créer une instance d'une EClass

Cas d'utilisation : correspondance M0→M1→ M2



Exemple interfaces réflectives

```
EFactory fact = //obtention de la référence
EPackage pa = //obtention de la référence
EClass acEC = (EClass)pa.getEClassifier("Acteur");
EObject ac = fact.create(acEC);
ac.eSet(acEC.getEStructuralFeature("nom"),"Client");
EClass cuEC = (EClass)pa.getEClassifier("casutil");
EObject cu1 = fact.create(cuEC);
cu1.eSet(cuEC.getEStructuralFeature("intitule"),"com
  mander panier");
EObject cu2 = fact.create(cuEC);
cu2.eSet(cuEC.getEStructuralFeature("intitule"),"val
  ider panier");
Collection parti =
  (Collection)ac.eGet(acEC.getEStructuralFeature("pa
  rticipe"));
parti.add(cu1); parti.add(cu2);
// …
```

Règles de génération d'interfaces taylored (1)

- Règles EClass
  - Une EClass donne lieu à la création d'une seule interface avec les caractéristiques :
    - A pour nom le nom de l'EClass
    - Offre des opérations de lecture et d'écriture pour chaque EAttribute de l'EClass
    - Offre des opérations de lecture et d'écriture pour chaque EReference de l'EClass
    - Hérite de l'interface réflective EObject

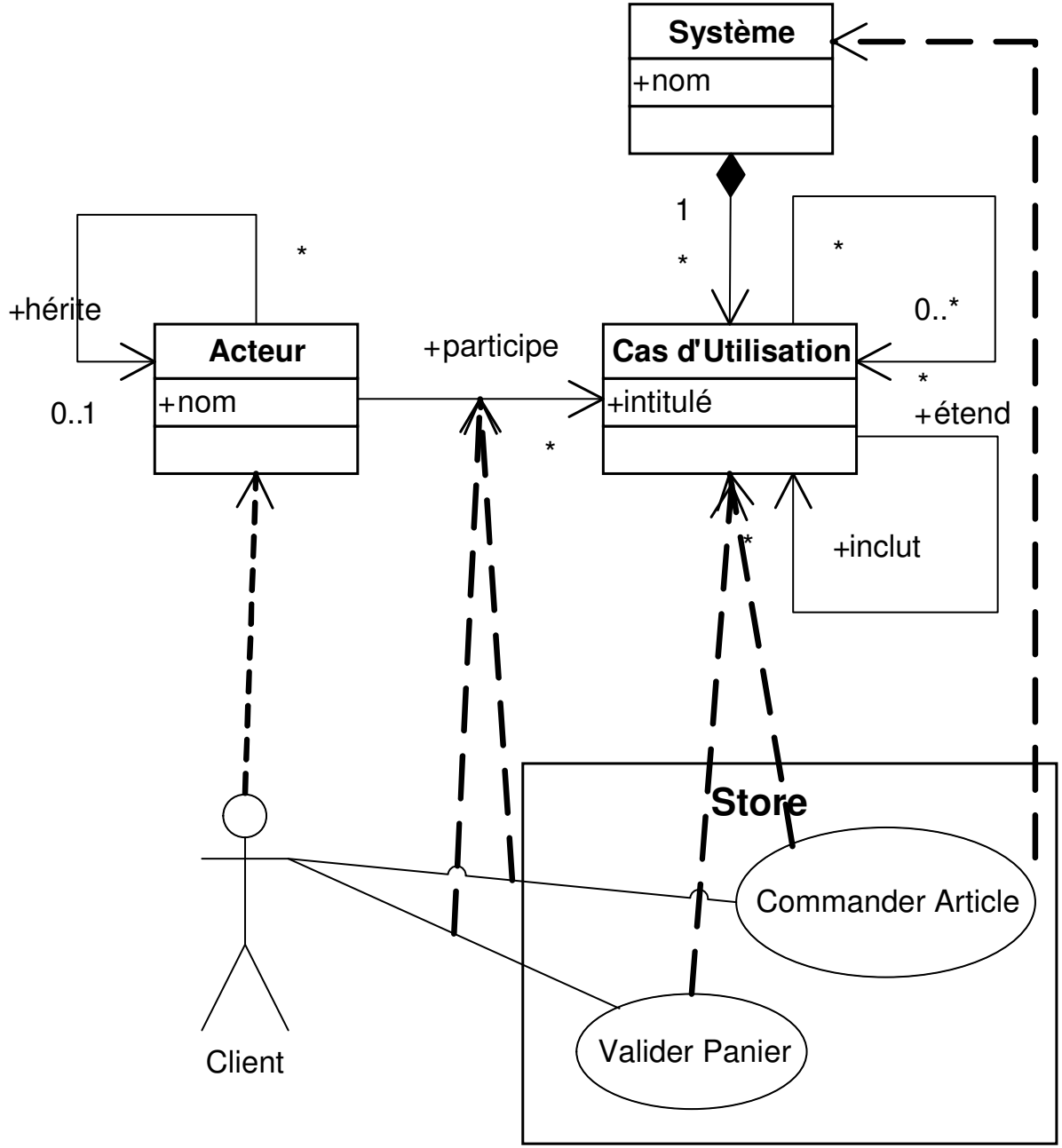
Règles de génération d'interfaces taylored (2)

- Règles EPackage
  - Une EClass donne lieu à la création de 2 interfaces : une interface Factory et une interface Package
  - Caractéristique de l'interface Factory :
    - A pour nom : nom_packageFactory
    - Offre une opération de création pour chaque EClass contenue dans l'EPackage
    - Hérite de l'interface réflective EFactory

Règles de génération d'interfaces taylored (3)

- Règles EPackage (suite)
  - Caractéristique de l'interface Package :
    - A pour nom : nom_packagePackage
    - Offre une opération de navigation pour chaque EClass contenue dans l'EPackage permettant d'obtenir l'EClass correspondante
    - Hérite de l'interface réflective EPackage

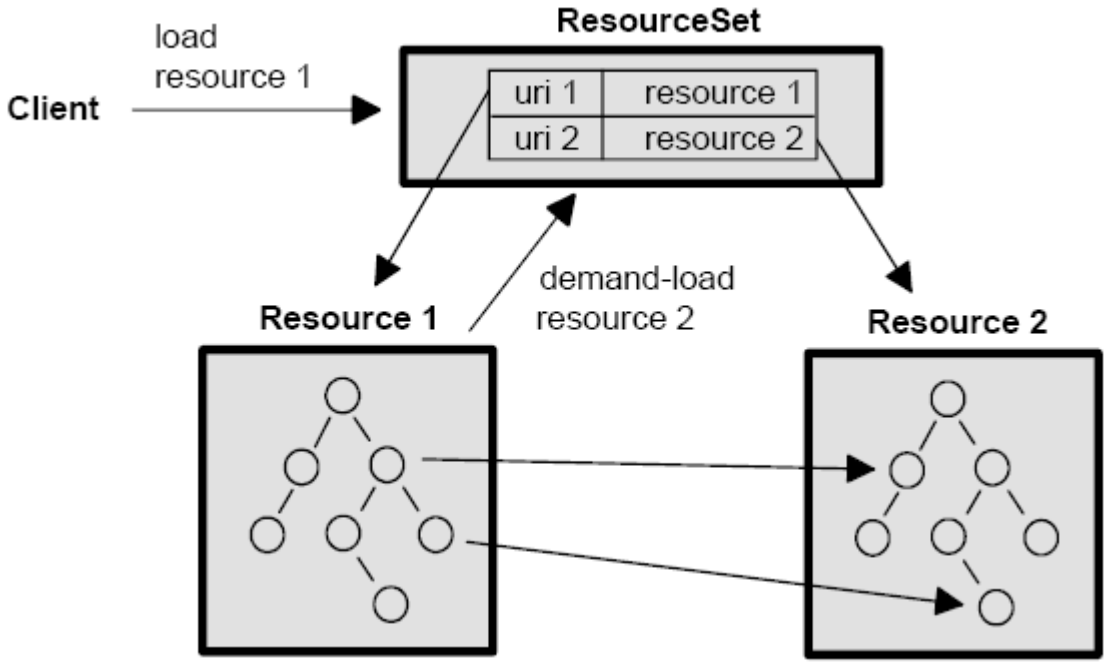
Cas d'utilisation : correspondance M0 → M1



Exemple interfaces taylored

```
CasFactory fact = CasFactory.eINSTANCE;
Acteur ac = fact.createActeur();
ac.setNom("Client");
Cas_d_utilisation cau1 =
   fact.createCas_d_utilisation();
cau1.setIntitule("commander panier");
Cas_d_utilisation cau2 =
   fact.createCas_d_utilisation();
cau2.setIntitule("valider panier");
ac.getParticipe().add(cau1);
ac.getParticipe().add(cau2);
System sys = fact.createSysteme();
sys.setNom("Store");
sys.getCas().add(cau1);
sys.getCas().add(cau2);
```

Persistance et Sérialisation



- Une donnée sérialisée = Une *Resource*
- Les *Resources* sont enregistrées dans un *ResourceSet*
- Chargement explicite d'une *Resource* par le client
- Chargement à la demande des autres *Resources* lors de la résolution des références

Chargement

```
// Create a resource set to hold the resources.
ResourceSet resourceSet = new ResourceSetImpl();

// Register the appropriate resource factory to handle all file extensions.
resourceSet.getResourceFactoryRegistry().getExtensionToFactoryMap().put
(Resource.Factory.Registry.DEFAULT_EXTENSION,
 new XMIResourceFactoryImpl());

// Demand load resource for this file.
Resource resource =
    resourceSet.getResource(URI.createFileURI(absolutePath), true);

// use reflective interfaces
for (EObject eObject : resource.getContents()) {...}

// or use taylored interfaces
Schema root = (Schema) resource.getContents().get(0);
for (Table t : root.getTables()) {...}
```

Sauvegarde

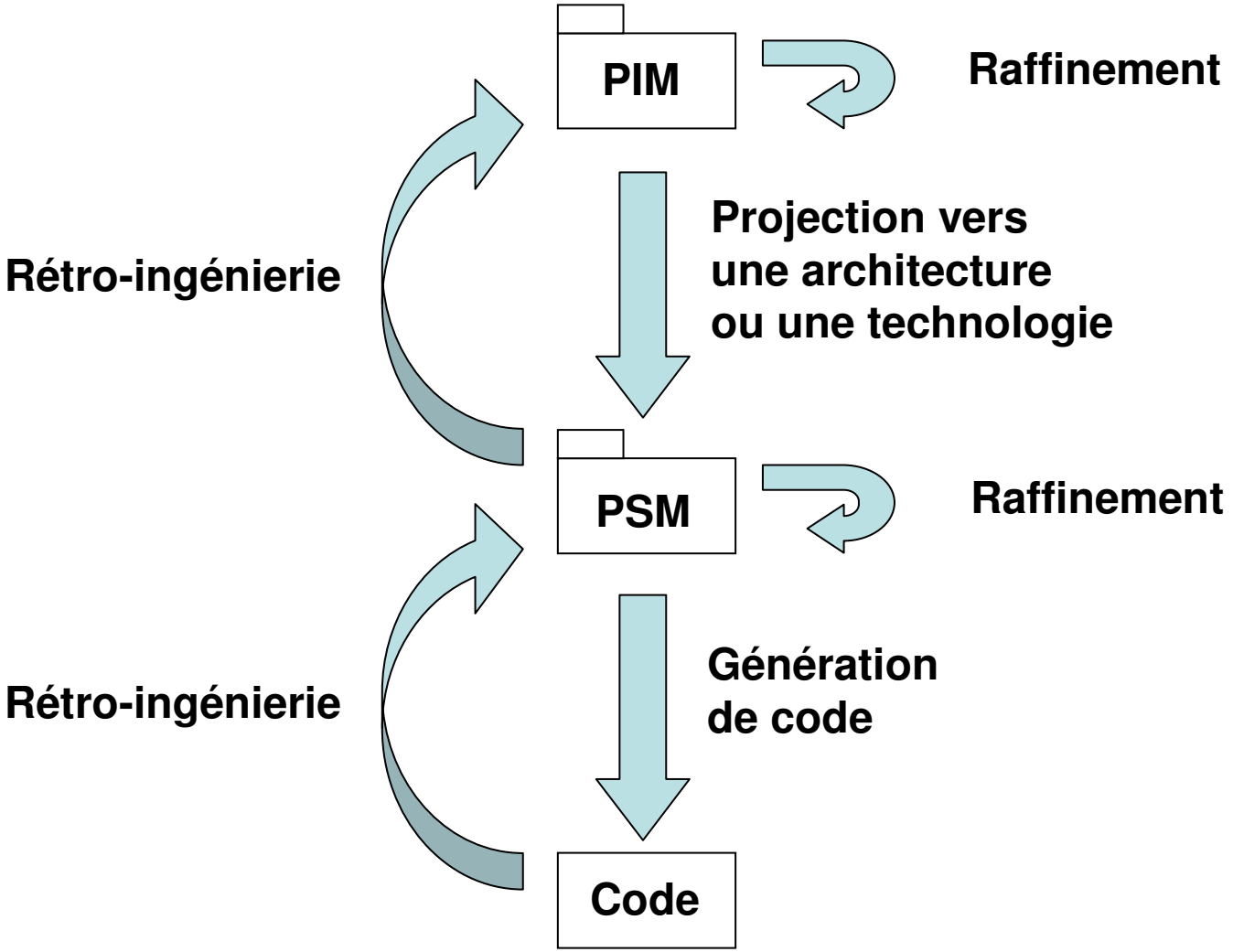
```
// Create a resource for this file.
URI fileURI = URI.createFileURI(absolutePath);
Resource resource = new XMIResourceFactoryImpl().createResource(fileURI);

// use reflective interfaces
EFactory fact = ...

// or use taylored interfaces
SchemaFactory fact = SchemaFactory.eINSTANCE()
Schema = fact.createSchema();
...

// save the resource
resource.getContents().add(schema);
try {
    resource.save(Collections.EMPTY_MAP);
} catch (IOException e) {
    e.printstackTrace();
}
```

Transformation de modèles

Introduction

- Les transformations de modèles sont le cœur des aspects de production de MDA
  - PIM vers PSM, PSM vers code (sens inverse).
- Les transformations de modèles sont basées sur les métamodèles
  - Toute classe UML se transforme en une classe Java
- Les constructeurs de plate-forme devraient produire les transformateurs permettant d'exploiter les plates-formes
  - UML vers EJB, UML vers .NET, UML vers
- Les sociétés devraient pouvoir adapter les transformateurs selon leurs propres expériences et leurs propres besoins
  - Ex: ne pas utiliser de composant EJB Entity!
- Actuellement les transformations de modèles peuvent s'écrire selon trois approches
  - Programmation, Template, Modélisation

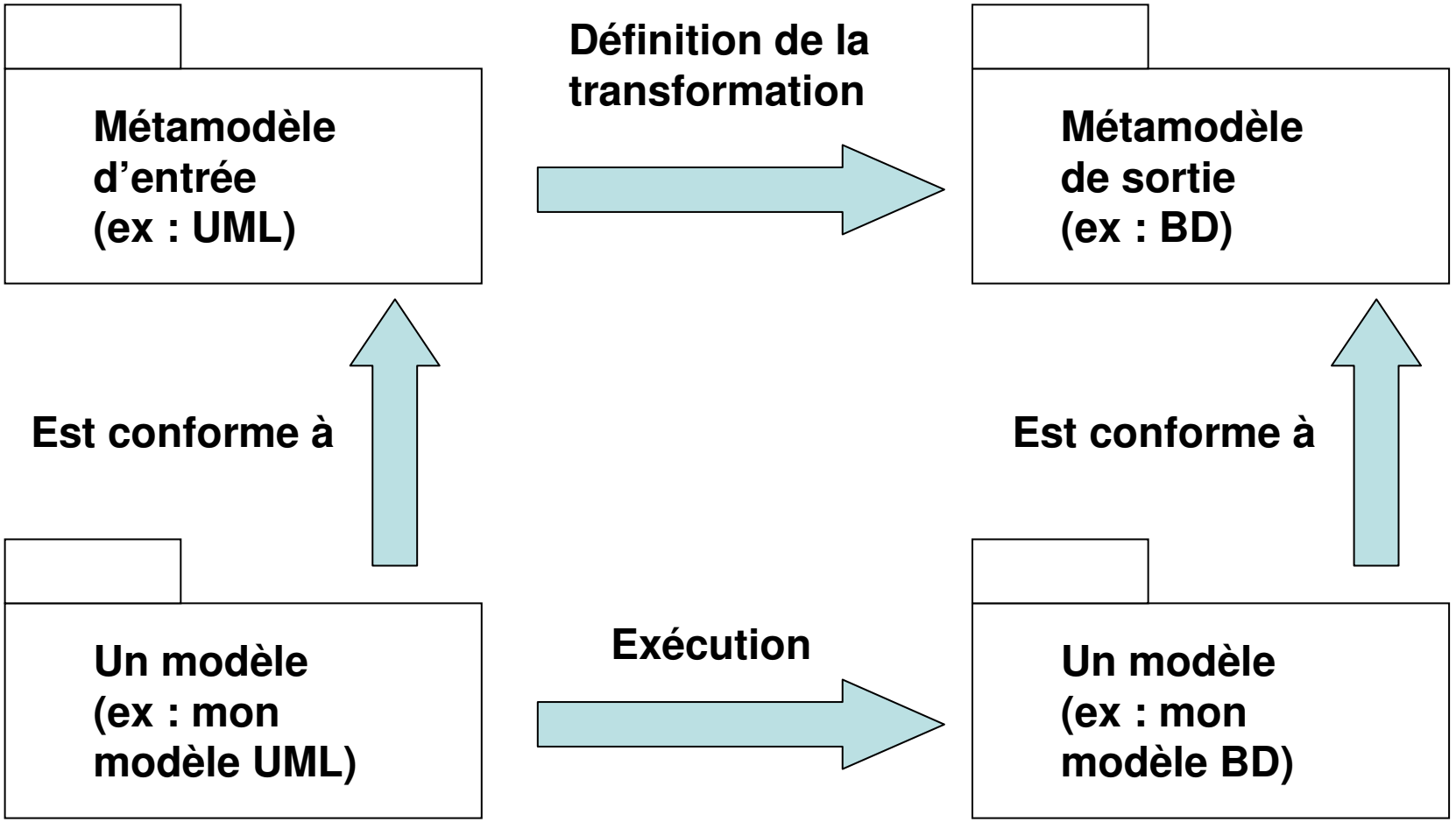
Principales transformations de MDA



Transformation et MDA

- Transformations de modèles PIM vers PIM
  - Raffinement de PIM pour améliorer la précision des informations
  - Ex : création de classes UML à partir d'un ensemble de diagrammes de séquences ou création de diagrammes d'états à partir de diagrammes de classes
  - Permet d'accélérer la production de PIM (outils)
- Transformations de modèles PIM vers PSM
  - Construction d'une bonne partie des PSM à partir des PIM
  - Garantissent la pérennité des modèles et leur productivité
- Transformations de modèles PSM vers code
  - Permettent de générer (la totalité) du code

Métamodèles et transformation de modèles

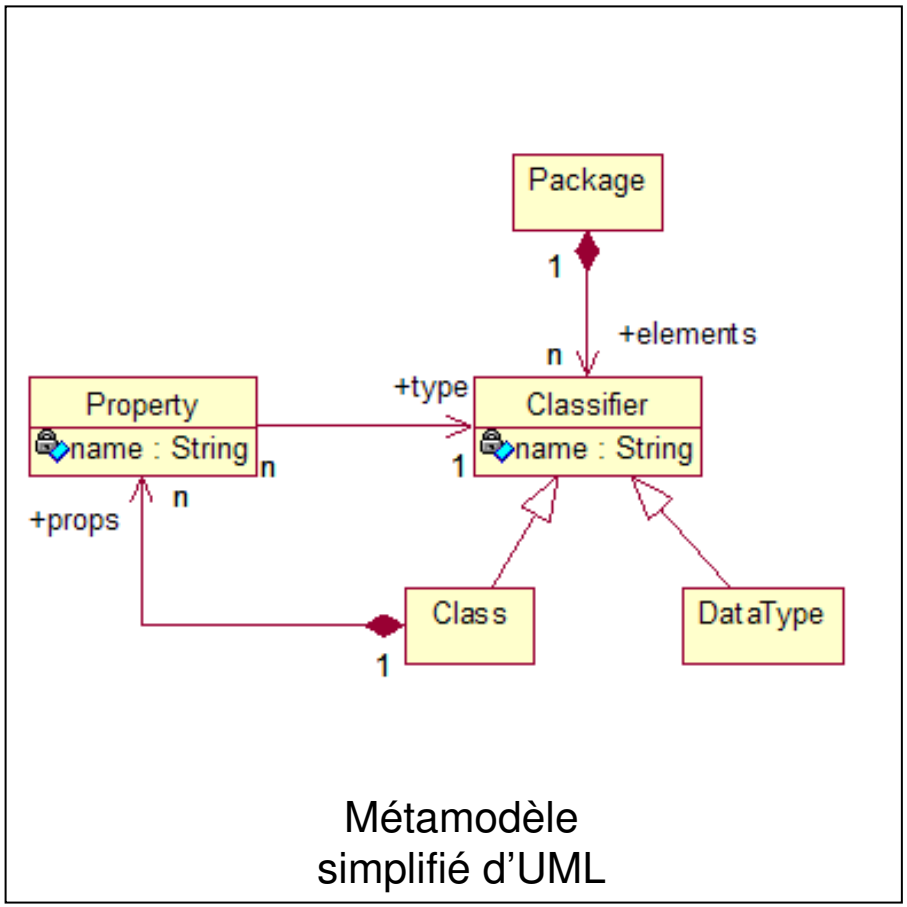


Spécification des règles de transformation

- Approche par programmation
  - Utilisation de langage objet pour programmer les transformations
  - Ex : manipulation de modèles en Java avec EMF
- Approche par template
  - Définition de canevas pour les modèles cibles paramétrés par des infos des modèles sources
  - Ex : JET (JSP-like) dans le framework Eclipse
- Approche par modélisation
  - Application des concepts de l'IDM aux transformations elles-mêmes
  - Ex : MOF2.0 QVT (Query,View,Transformation)

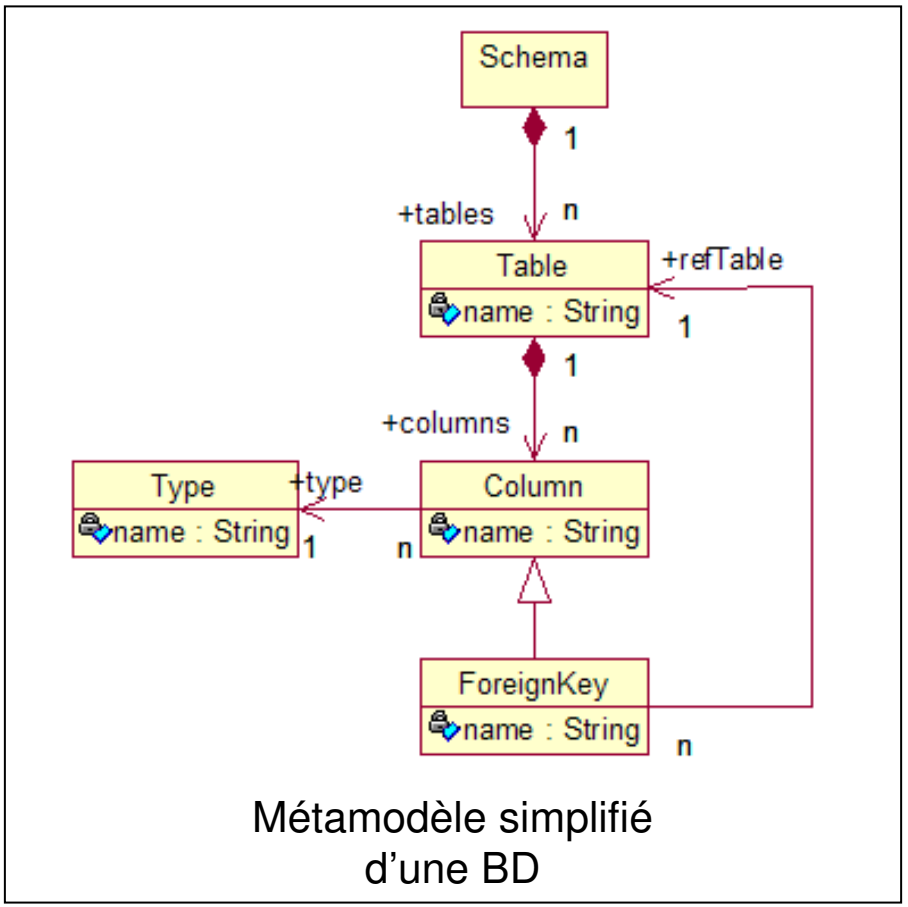
Approche par programmation : exemple de mise en œuvre

Source de la transformation



Métamodèle simplifié d'UML

Cible de la transformation



Métamodèle simplifié d'une BD

Définition des règles de la transformation UML vers BD

1. A tout package UML correspond un schéma
2. A toute classe UML du package correspond une table dans le schéma
3. A toute propriété UML d'une classe correspond une colonne dans la table
   - Si le type de la propriété est une classe UML, la colonne est une clé étrangère
   - Si le type de la propriété est un type de donnée UML, la colonne est du type correspondant
4. A tout type de donnée UML d'un package correspond un type de donnée dans le schéma

Exemple de modèle source (.xmi)

```
<?xml version="1.0" encoding="ASCII"?>
<simpleuml:Package xmi:version="2.0"
   xmlns:xmi="http://www.omg.org/XMI"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xmlns:simpleuml="http:///simpleuml.ecore"
   xsi:schemaLocation="http:///simpleuml.ecore simpleuml.ecore">
  <elements xsi:type="simpleuml:Class" name="Etudiant">
    <props name="nom" type="//@elements.2"/>
    <props name="prenom" type="//@elements.2"/>
    <props name="promo" type="//@elements.1"/>
  </elements>
  <elements xsi:type="simpleuml:Class" name="Promo">
    <props name="nom" type="//@elements.2"/>
  </elements>
  <elements xsi:type="simpleuml:DataType" name="String"/>
</simpleuml:Package>
```

Modèle transformé (.xmi)

```
<?xml version="1.0" encoding="ASCII"?>
<xmi:XMI xmi:version="2.0" xmlns:xmi="http://www.omg.org/XMI"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xmlns:database="http:///database.ecore">
  <database:Type name="String"/>
  <database:Schema>
    <tables name="Etudiant">
      <columns name="nom" type="/0"/>
      <columns name="prenom" type="/0"/>
      <columns xsi:type="database:ForeignKey" name="promo"
   refTable="/1/@tables.1"/>
    </tables>
    <tables name="Promo">
      <columns name="nom" type="/0"/>
    </tables>
  </database:Schema>
</xmi:XMI>
```

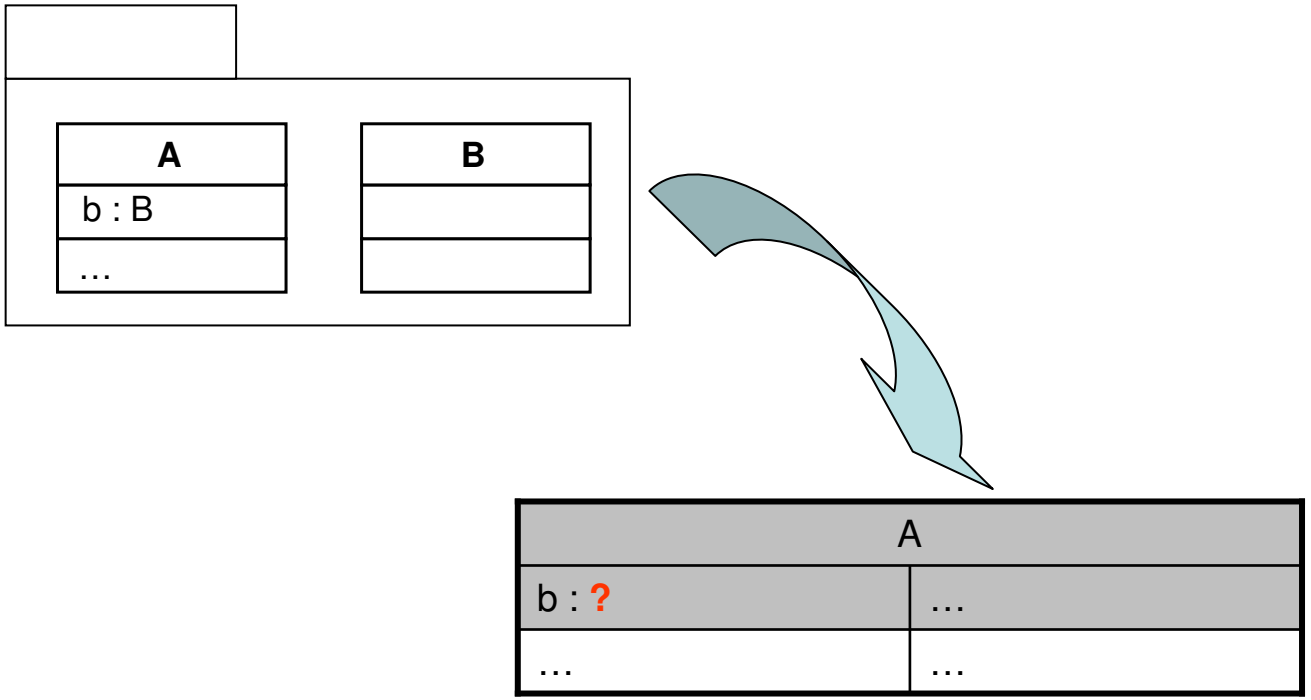
Elaboration des règles par programmation : règle 1

```
private void package2Schema(Package p) {
 Schema sh =
  DatabaseFactory.eINSTANCE.createSchema();

 for (Classifier c : p.getElements()) {
  if (c instanceof Class) {
   sh.getTables().add(class2Table((Class) c));
  }
  else if (c instanceof DataType) {
   resources.add(dataType2Type((DataType) c));
  }
 }
 resources.add(sh);
}
```

Elaboration des règles par programmation : règles 2, 3 et 4

```
public Table class2Table(Class c) {
    //TODO
}
public Column property2Column(Property p) {
    //TODO
}
public Type dataType2Type(DataType dt) {
    //TODO
}
```

Lien entre clé étrangère et table



Tant que la classe B n'a pas été transformée (`class2Table()`), il n'est pas possible de spécifier la table que cette clé référence

Elaboration des règles par programmation : gestion des références

```
private List<EObject> resources;
private Map<Class,Table> classTable;
private Map<Class,List<ForeignKey>> cleRef;
private HashMap<Classifier, List<Column>> colType;
private HashMap<DataType, Type> datatypeType;
```

Elaboration des règles par programmation : règle 2

```
private Table class2Table(Class clazz) {
 Table tab =
  DatabaseFactory.eINSTANCE.createTable();
 tab.setName(clazz.getName());
 for (Property p : clazz.getProps()) {
  tab.getColumns().add(property2Column(p));
 }
 classTable.put(clazz, tab);
 if(cleRef.containsKey(clazz)) {
  for(ForeignKey fk : cleRef.get(clazz)) {
   fk.setRefTable(tab);
  }
 }
 return tab;
}
```

Elaboration des règles par programmation : règle 3

```
private Column property2Column(Property p) {
 if(p.getType() instanceof Class) { //Class
  ForeignKey fk =
   DatabaseFactory.eINSTANCE.createForeignKey();
  fk.setName(p.getName());
  if(classTable.containsKey(p.getType())) {
   fk.setRefTable(classTable.get(p.getType()));
  } else {
   if (cleRef.containsKey(p.getType())) {
    cleRef.get(p.getType()).add(fk);
   } else {
    ArrayList<ForeignKey> al = new
   ArrayList<ForeignKey>();
    al.add(fk);
    cleRef.put((Class) p.getType(), al);
   }
  }
  return fk;
 } else { //DataType ...}
}
```

Elaboration des règles par programmation : règle 3 (suite)

```
private Column property2Column(Property p) {
 if(p.getType() instanceof Class) { //Class ...
 } else { //DataType
  Column col = DatabaseFactory.eINSTANCE.createColumn();
  col.setName(p.getName());
  if (datatypeType.containsKey(p.getType())) {
   col.setType(datatypeType.get(p.getType()));
  } else {
   if (colType.containsKey(p.getType())) {
    colType.get(p.getType()).add(col);
   } else {
    ArrayList<Column> al = new ArrayList<Column>();
    al.add(col);
    colType.put(p.getType(), al);
   }
  }
 return col;
 }
}
```

Elaboration des règles par programmation : règle 4

```
private Type dataType2Type(DataType dt) {
 Type t = DatabaseFactory.eINSTANCE.createType();
 t.setName(dt.getName());
 if (colType.containsKey(dt)) {
  for(Column col : colType.get(dt)) {
   col.setType(t);
  }
 }
 return t;
}
```

Elaboration des règles par programmation : sauvegarde

```
public void save(String fileURIstr) {
 URI fileURI = URI.createFileURI(fileURIstr);
 Resource resource = new
   XMIResourceFactoryImpl().createResource(fileURI);
 for(EObject eo : resources) {
  resource.getContents().add(eo);
 }

 try {
  resource.save(Collections.EMPTY_MAP);
 } catch (IOException e) {
  e.printStackTrace();
 }

}
```

Bibliographie

Livre

- **MDA en action : Ingénierie logicielle guidée par les modèles**
  - de Xavier Blanc

Cours

- Eric Cariou
  - http://www.univ-pau.fr/~ecariou/
- Jean Bézivin
  - http://www.aristote.asso.fr/Presentations/CEA-EDF-2003/Cours/JeanBezivin/IndexJeanBezivin.html
- Alban Gaignard
  - http://www.i3s.unice.fr/~gaignard