MDE : **M**odel **D**riven **E**ngineering
IDM : **I**ngénierie **D**irigée par les **M**odèles

Y. LAKHRISSI
ENSA–USMBA–Fès

Plan du cours

- **Chapitre 1 :** Introduction et Contexte de l'IDM
- **Chapitre 2 : OCL** (Object Constrainte Language)
- **Chapitre 3 :** Méta-Modélisation
  - Notions de Modèle(M1) ; Méta-modèle(M2) ; Méta-méta-modèle(M3)
  - Extension du méta-modèle d'UML
  - TP 1
- **Chapitre 4 :** MDA (Model Driven Architecture)
  - Principes : CIM, PIM, PSM.
  - Outils MDA
- **Chapitre 5 :** Transformation de Modèles
  - Principe
  - Langages de transformation
  - TP2, TP3 et TP4

Chapitre I : Introduction

Contexte de l'IDM

Plan du chapitre I

- Chapitre I : Contexte de l'IDM

1 - État des lieux et constats
- Types de logiciel
- Nature du logiciel
- Critères de la qualité logicielle

2 - Génie Logiciel
- Objectifs
- Technologies de production logiciel au cours de l'histoire

3 - Limites du développement logiciel actuel
- Limites de la technologie Objet
- Les nouvelles exigences

4 - L'Ingénierie des Modèles : une nouvelle philosophie de production logiciel
- Définitions
- Vision générale des standards de l'OMG Object Management Group

Logiciel ?

- Définition :
  - Un **logiciel** ou une **application** est un ensemble de programmes, qui permet à un ordinateur ou à un système informatique d'assurer une tâche ou une fonctionnalité
  - Physiquement :
    - une structure d'informations Incluant programmes, données, documents, …
    - Un système d'objets, de processus, …
  - Le logiciel n'est pas seulement un ensemble de programmes, mais aussi la documentation pour :
    - l'installation,
    - l'utilisation,
    - le développement,
    - la maintenance.

Les catégories de logiciel (1/3)

- 1 – Les systèmes embarqués
  - Systèmes exécutés dans du matériel électronique isolé
    - machine à laver, télévision, lecteur DVD, téléphone mobile, magnétoscope, four à micro-ondes, réfrigérateur, lecteur MP3, ...
  - Difficile à modifier
- 2 - Les systèmes temps réel
  - Réaction immédiate requise
  - Systèmes de contrôle et de surveillance
  - Manipulent et contrôlent le matériel technique
  - Environnement contraignant
- 3 - Les systèmes de matériel
  - Systèmes d'exploitation, …

Les catégories de logiciel (2/3)

- 4 - Les systèmes de traitement de données
  - Ils stockent, recherchent, transforment et présentent l'information aux utilisateurs
  - Grandes quantités de données avec des corrélations complexes, enregistrées dans les bases de données
  - Largement utilisés en administration des affaires
  - Fiabilité des résultats
  - Sécurité dans l'accès aux données
- 5 - Les systèmes distribués
  - Synchronisent la transmission, assurent l'intégrité des données et la sécurité, ...
  - Technologies utilisées : CORBA, DOM/DCOM/.NET, SOAP, EJB, …

Les catégories de logiciel (3/3)

- 6 - Les systèmes d'entreprise
  - Décrivent les buts, les ressources, les règles et le travail réel dans une entreprise
- 7 - Générique
  - Vendu sur le marché
    - un tableur, un outil de base de données
    - un outil de traitement de texte
    - …
- 8 - Sur mesure
  - Pour un client spécifique
- 9 - …

La nature du logiciel (1/3)

- Le développement du logiciel est difficile à estimer
  - Il est difficile d'estimer l'effort de développement
  - Il est difficile d'estimer le temps,
  - Il est difficile d'estimer le coût final, …
- Le processus de développement est difficile à automatiser
  - L'industrie du logiciel exige beaucoup de main d'œuvre
- La qualité d'un logiciel n'est pas apparente
  - Même des informaticiens peu qualifiés peuvent arriver à bricoler quelque chose qui semble fonctionner
  - Egalement, les informaticiens qualifiés ne peuvent pas être sur de la qualité de leurs logiciels

La nature du logiciel (2/3)

- Un logiciel semble facile à modifier
  - La tentation est forte d'effectuer des changements rapides sans vraiment en mesurer la portée
- Un logiciel à une durée de vie
  - Il se détériore à mesure que des changements sont effectués
    - en raison de l'introduction d'erreurs
  - Les exigences changent
- Les logiciels mal conçus se détériorent rapidement
- La demande pour du logiciel est toujours croissante

La nature du logiciel (3/3)

- Raisons pour lesquelles le logiciel vieillit :
  - Maintenance / Modification
  - Inflexibilité dès le début
  - Documentation insuffisante ou inconsistante
  - Manque de modularité
  - Complexité, …
- Un logiciel est fiable s'il :
  - Répond aux spécifications
  - Ne produit jamais de résultat erroné : faute de raisonnement
  - Ne jamais entrer dans un état incohérent : oubli de cas particuliers

Les critères de qualité du logiciel

- **Validité** : Conformité d'un logiciel avec sa spécification
- **Robustesse** : Capacité à fonctionner même dans des conditions anormales
- **Extensibilité** : Facilité d'adaptation à des changements de spécifications
- **Réutilisabilité** : Capacité à être réutilisé en tout ou partie dans une nouvelle application
- **Compatibilité** : Facilité avec laquelle des composants logiciels peuvent être combinés
- **Efficacité** : Utilisation optimale des ressources matérielles
- **Portabilité** : Facilité de transfert dans différents environnements
- **Vérifiabilité** : Facilité à valider ; bien structuré et modulaire
- **Intégrité** : Aptitude à protéger les codes et les données
- **Ergonomie** : Facilité d'utilisation, Facilité d'apprentissage, bien documenté

Sommaire

- Chapitre I : Contexte de l'IDM

  - **1 - État des lieux et constats**
    - Types de logiciel
    - Nature du logiciel
    - Critères de la qualité logicielle

  - **2 - Génie Logiciel**
    - Objectifs
    - Technologies de production logiciel au cours de l'histoire

  - **3 - Limites du développement logiciel actuel**
    - Limites de la technologie Objet
    - Les nouvelles exigences

  - **4 - L'Ingénierie des Modèles : une nouvelle philosophie de production logiciel**
    - Définitions
    - Vision générale des standards de l'OMG Object Management Group

Le génie Logiciel ?

- Art de bien faire de bons Logiciel
  - Art : technique, créativité, esthétique, …
  - Bien faire : réussite, rentabilité, …
  - Bons : performance, fiabilité, robustesse, …
- Art de produire du logiciel
  - Art de spécifier, concevoir, réaliser, et de faire évoluer, avec des moyens (matériel et humain) et dans des délais raisonnables, des programmes, des documentations, et des procédures de qualité en vue d'utiliser un ordinateur pour résoudre certains problèmes.
- On parle également de processus de développement de logiciels et de gestion de projets
  - Gestion du personnel : Efforts
  - Gestion des ressources : Coûts
  - Aspects techniques : Conception & Réalisation
  - Contraintes de réalisation : Planification des taches et du temps

Le génie logiciel

- Les points communs des définitions
  - Travail de groupe et non d'un individu isolé
  - Besoins techniques et non-techniques
    - Connaissances informatiques
    - Capacité de communication
    - Gestion de projet

- Objectif du GL
  - Développer des logiciels considérés comme :
    - Logiciels fiables
    - Logiciels satisfaisant les besoins
    - Logiciels maintenables
    - Logiciels exploitables dans différents environnements
    - (cf. qualités du logiciel)

Le génie logiciel - Origine

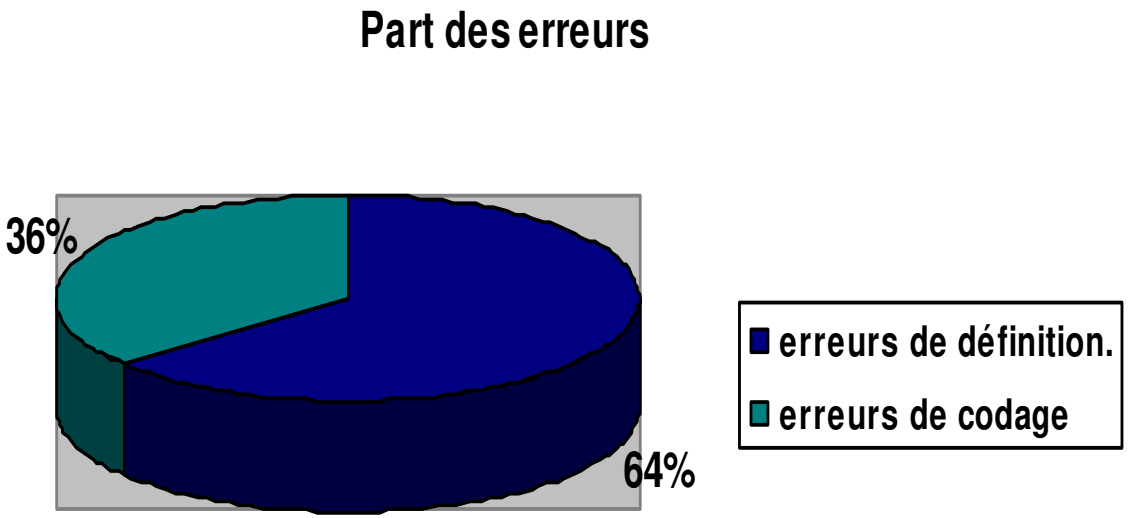
- Origine : Apparu en 1968, devant les insatisfactions générales en matière de logiciel :
  - Fiabilité douteuse
  - Dépassement des délais
  - Dépassement des coûts
  - Erreurs résiduelles persistantes
  - Sensibilité aux erreurs humaines, aux pannes matérielles
  - Difficultés de conversion, de mise en œuvre
  - Difficultés d'évolution

  ...
  - Complexité croissante du logiciel
  - Coût croissant lié en grande partie à la maintenance

  ...
  - Criticité des secteurs d'activité

Quelques statistiques

- Étude du gouvernement américain en 1979
  - Logiciels payés mais jamais livrés 40%
  - Logiciels livrés mais jamais utilisés 30%
  - Logiciels utilisés après modification 25%
  - Logiciels utilisés tel quel 5%



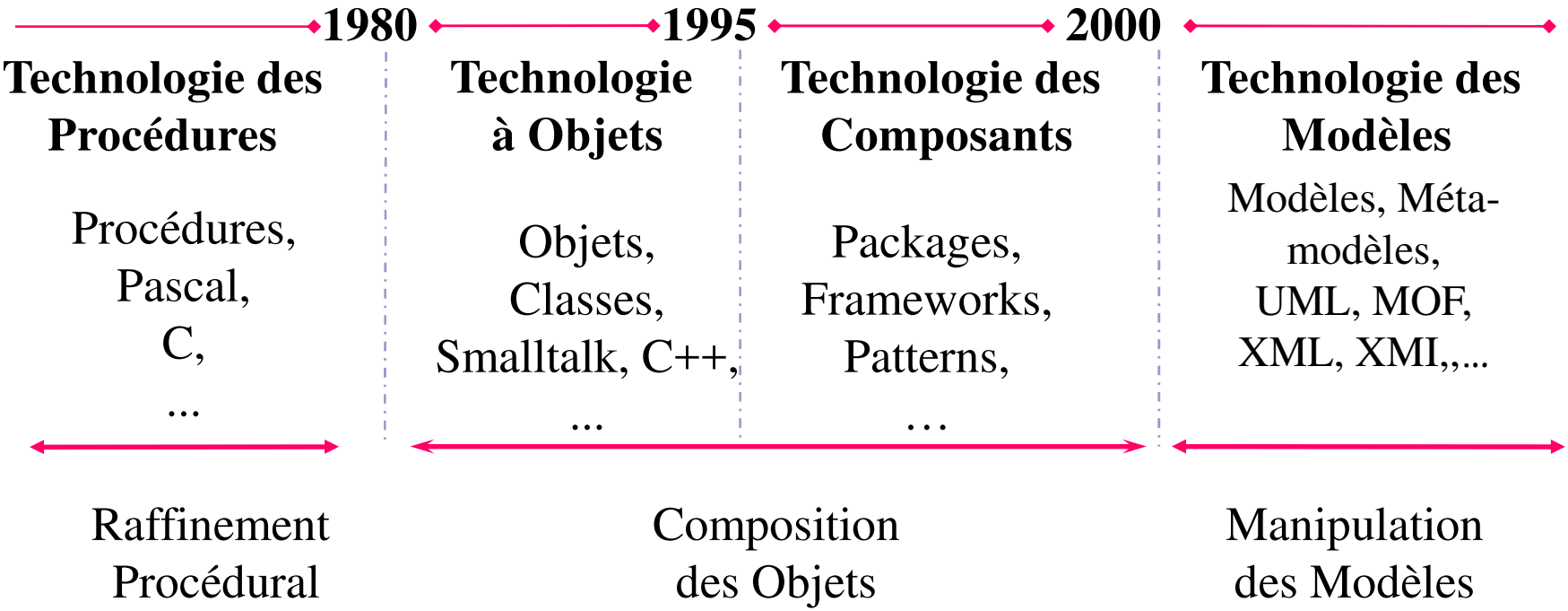
Évolution des technologies

- Les années 60
  - Pas de méthodes de développement de logiciels
  - Programmation assembleur
- Les années 70
  - Décomposition fonctionnelle
  - Méthodes SADT, ...
  - Programmation structurelle / modulaire
- Les années 80
  - Echec des méthodes purement fonctionnelles
  - Modélisation (Jackson)
  - Programmation Objet (Meyer)

Évolution des technologies

- Les années 90
  - Systèmes distribués et déploiement
  - Composants et lignes de produits
  - Propriétés non-fonctionnelles
  - Installation/Exécution
  - Méthodes et langages de conception : UML, …
- Les années 2000
  - Intégration logicielle omniprésente
  - Accélération des changements technologiques
  - La technologie objet commence à vieillir
  - Quelles solutions ?
  - Ingénierie des modèles !

Évolution des technologies



Abandon de la Technologie Objets pour la Technologie Modèles (J. Bézivin)

Sommaire

- Chapitre I : Contexte de l'IDM

  - **1 - État des lieux et constats**
    - Types de logiciel
    - Nature du logiciel
    - Critères de la qualité logicielle

  - **2 - Génie Logiciel**
    - Objectifs
    - Technologies de production logiciel au cours de l'histoire

  - **3 - Limites du développement logiciel actuel**
    - Limites de la technologie Objet
    - Les nouvelles exigences

  - **4 - L'Ingénierie des Modèles : une nouvelle philosophie de production logiciel**
    - Définitions
    - Vision générale des standards de l'OMG Object Management Group

Limites de la programmation objet (1/3)

- Les systèmes deviennent plus complexes
- Volume croissant
  - Des données
  - Du code
- Évolutivité croissante
  - De la partie métier (mondialisation, concentration, restructuration, …)
  - De la partie plate-forme d'exécution
- Hétérogénéité croissante
  - Des langages et des paradigmes
  - Des supports de données et des protocoles d'accès
  - Des systèmes et des plates-formes
  - Des technologies
- Le rythme d'arrivée des nouvelles technologies s'accélère
- Ce rythme ne se ralentira pas
- Les vieilles technologies ne meurent pas, elles se cachent

Limites de la programmation objet (2/3)

- Spécifications mouvantes
  - Evolution des besoins, de la législation, des groupements de sociétés, des technologies, …)
- Besoin important d'Intégration des applications existantes
- Nécessité d'une forte expertise des domaines métiers
  - Transports, médecine, finances, gestion, ….
- Intégration de propriétés non-fonctionnelles complexes (sécurité, persistance, traçabilité, …)
- Multiplicité des points de vue sur le logiciels
  - Expression des besoins, analyse, conception, implantation, validation, évolution, maintenance, clients, …
- La complexité a atteint un tel niveau qu'il est hors de portée d'un seul individu d'avoir une **vision globale** sur un système en évolution

Limites de la programmation objet (3/3)

- Absence de vision globale de l'application
  - les principaux concepts sont définis au niveau d'un objet individuel
  - pas de notion de description globale de l'architecture
- Difficulté d'évolution
  - conséquence de l'absence de vision globale
- Absence de services extra-fonctionnels
  - les services doivent être réalisés "à la main" (sécurité, etc.)
- Conclusion
  - **charge importante pour le programmeur**
  - **incidence sur la qualité de l'application**

Patrons et Composants

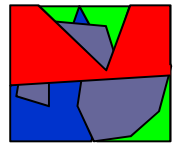
- Les patrons de conception et les composants ne sont ils pas une solution à la crise de la technologie objet ?

- Les patrons de conception
  - Encapsulent des savoir faire d'experts
  - Traitent autant du problème que des solutions
  - Traitent d'aspects non fonctionnels
  - Des collaborations paramétrées

- Les composants
  - Les composants permettent la construction d'applications par composition de briques de base configurables
  - Un composant est un module logiciel autonome
    - unité de déploiement (installation sur différentes platesformes)
    - unité de composition (combinaison avec d'autres composants)

Bilan

- Les nouvelles Exigences :
  - Besoins fonctionnels en constante augmentation
  - Besoins techniques complexes et évolutifs
  - Multiples acteurs, …
- Conséquence :
  - des composants complexes et hétérogènes …



- Symptômes :
  - **Dispersion :** une fonctionnalité est distribuée dans tout le système, et non pas placée dans une unité bien identifiée
  - **Couplage des fonctionnalités :** le cas où une unité contient des éléments provenant de différentes préoccupations

Réutilisation

Évolution

Traçabilité

Sommaire

- Chapitre I : Contexte de l'IDM

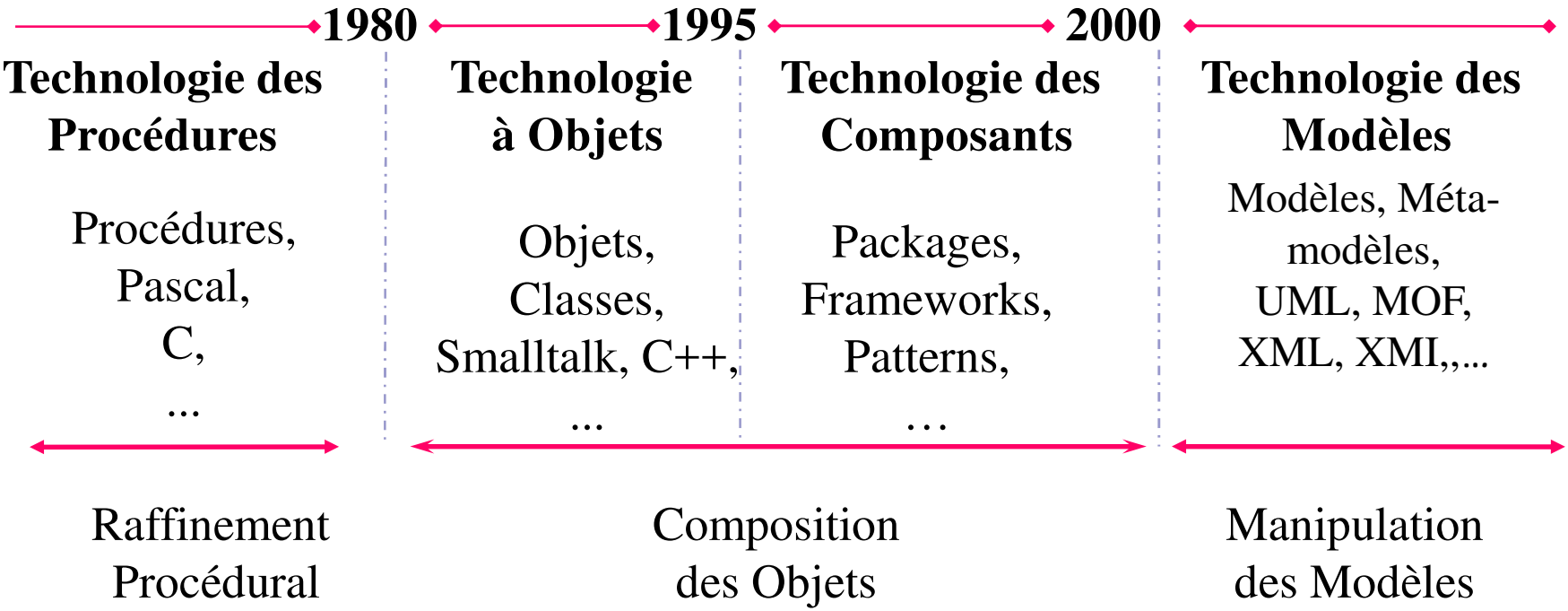
  - **1 - État des lieux et constats**
    - Types de logiciel
    - Nature du logiciel
    - Critères de la qualité logicielle

  - **2 - Génie Logiciel**
    - Objectifs
    - Technologies de production logiciel au cours de l'histoire

  - **3 - Limites du développement logiciel actuel**
    - Limites de la technologie Objet
    - Les nouvelles exigences

  - **4 - L'Ingénierie des Modèles : une nouvelle philosophie de production logiciel**
    - Définitions
    - Vision générale des standards de l'OMG Object Management Group

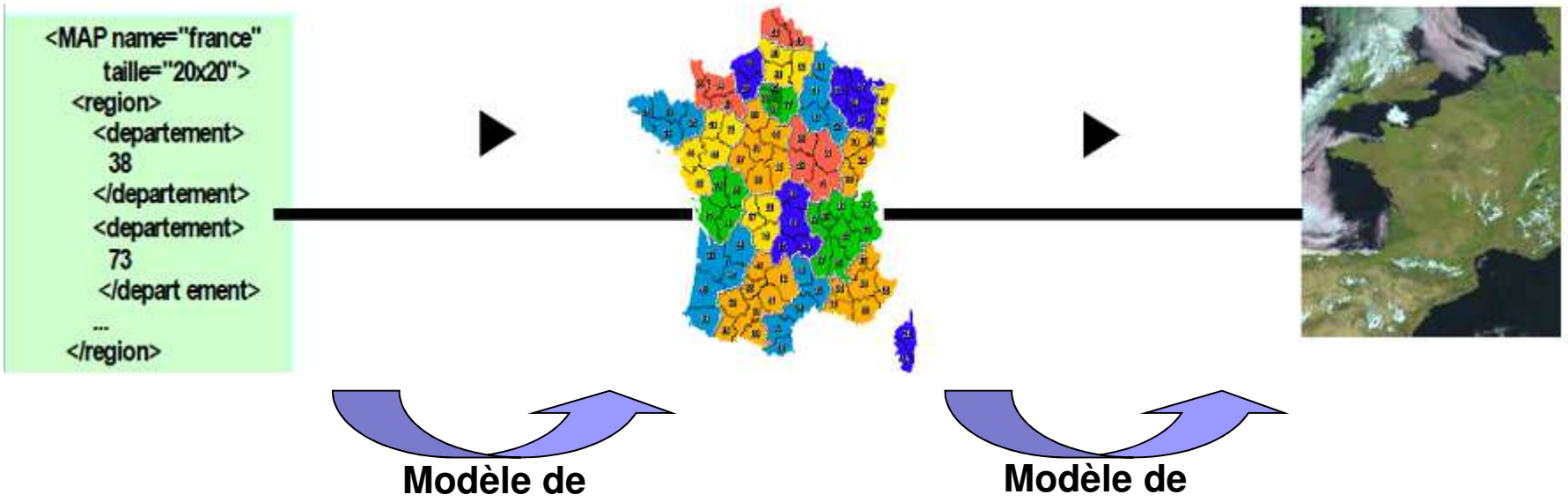
Évolution des technologies



Abandon de la Technologie Objets pour la Technologie Modèles (J. Bézivin)

Modèles ?

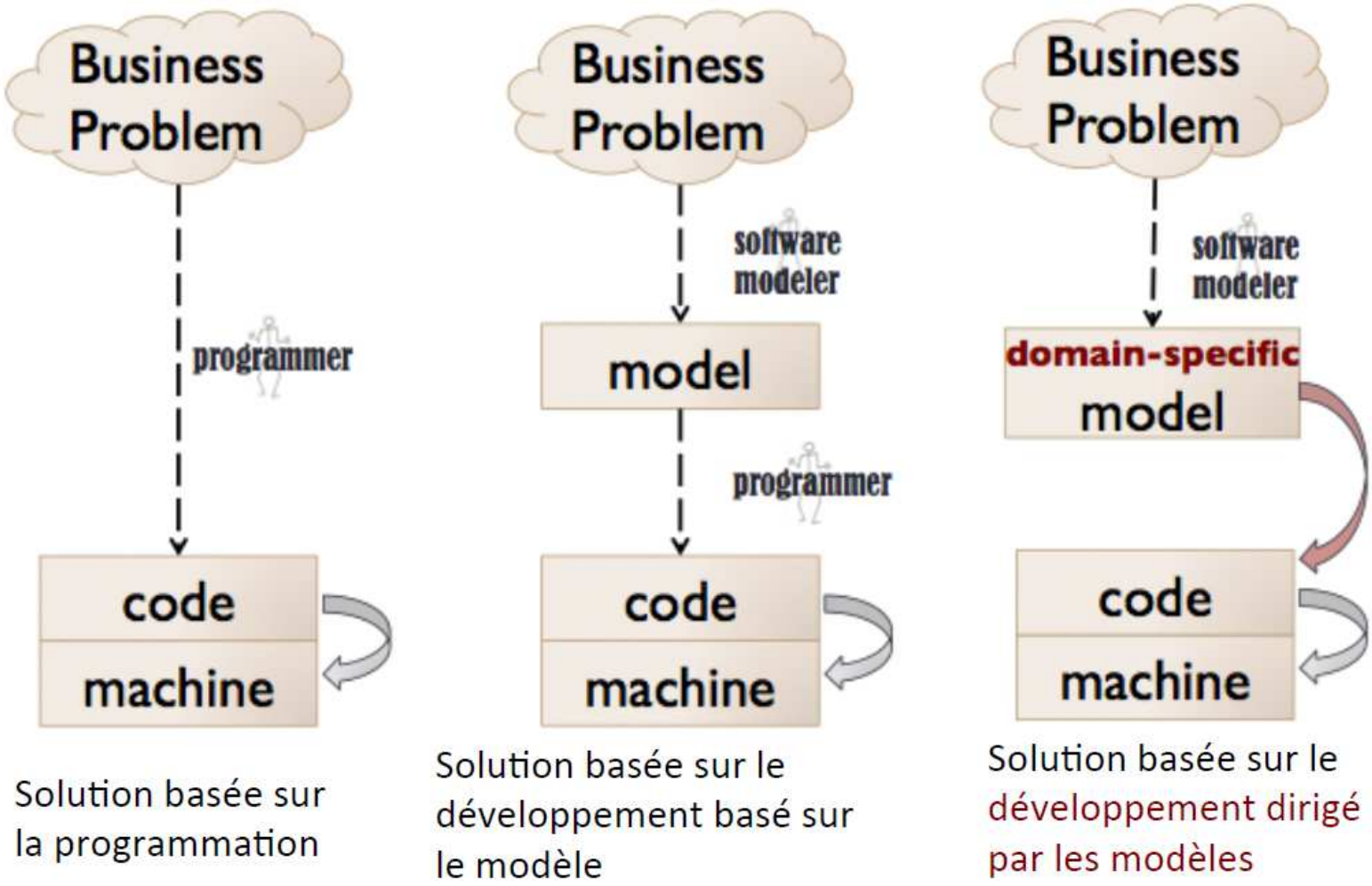
- Définition :
  - Un modèle est une abstraction de quelque chose de réel qui permet de comprendre avant de construire,
  - Le modèle simplifie la gestion de la complexité en offrant des points de vue et niveaux d'abstractions + ou - détaillés selon les besoins



Modèles

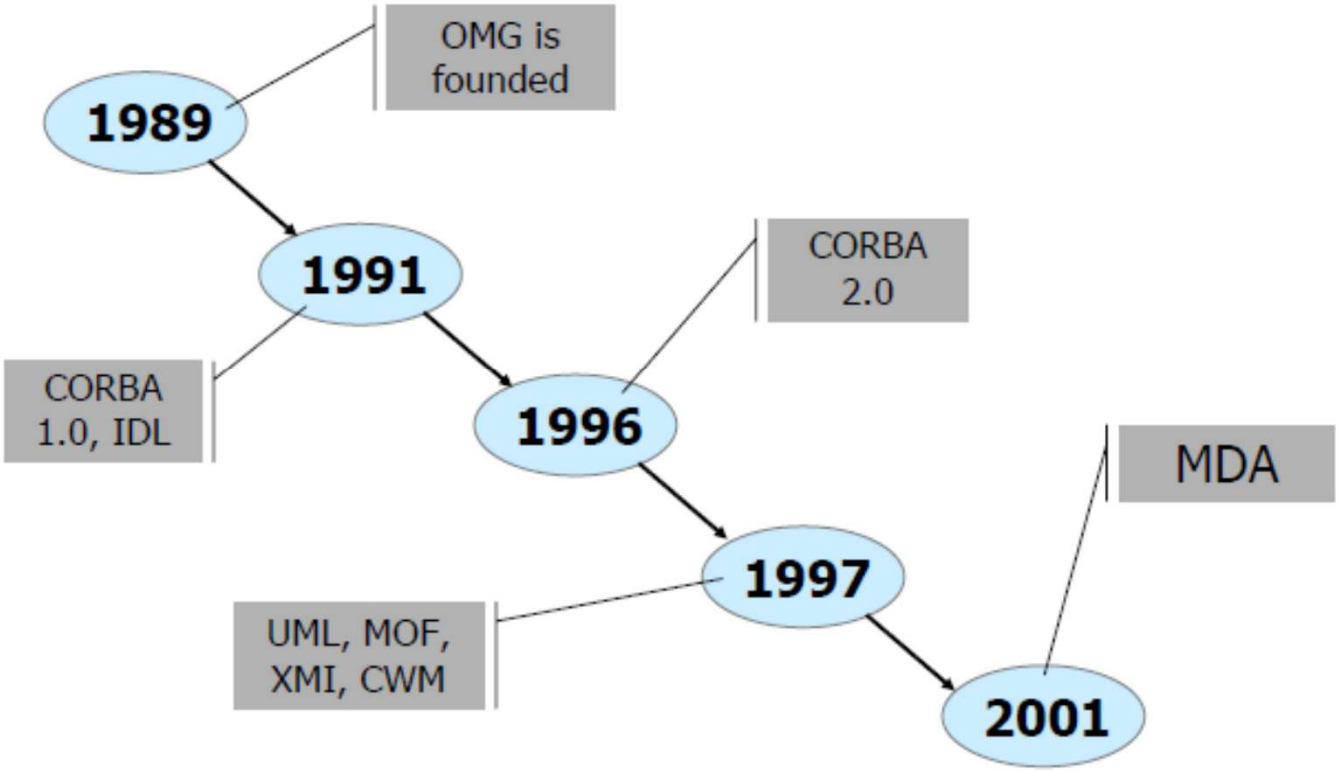
- La construction et la maintenance d'applications nécessitent de multiples langages (différents niveaux d'abstraction, différents intervenants),
- L'ingénierie des modèles consiste à définir les techniques nécessaires à la définition et à la mise en production de ces différents langages,
- UML est un langage de modélisation qui entre intégralement dans l'ingénierie des modèles,
- Il est important de savoir maitriser la diversité des langages de modélisation,
- Il est plus qu'important de rendre ces langages de modélisation productifs.

Pourquoi la méta-modélisation



OMG - Historique

- l'OMG, consortium de plus 1000 entreprises, initie la démarche MDA.



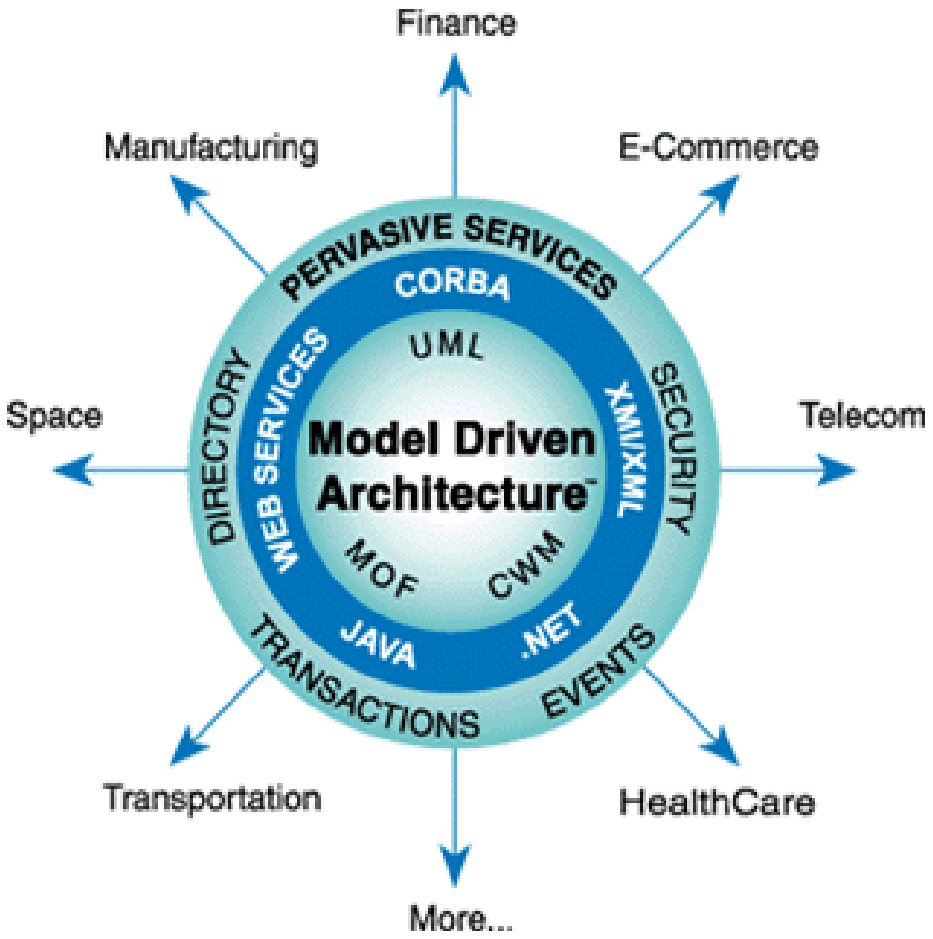
OMG : Object Management Group

- Parlons les mêmes langages et partageons les mêmes technologies !
  - → Les standard de l'OMG : UML, XMI, CWM, CORBA, IDL, …
- Expression de contraintes sur les modèles UML
  - → OCL
- Adaptabilité, extension d'UML
  - → Méta-modélisation
  - → Profils UML
- Interopérabilité
  - → MDA (CIM, PIM, PSM)
  - → QVT
- Réutilisation des solutions
  - → Composants, l'architecture CORBA

MDA : Vers l'interopérabilité

- Il n'y aura pas de consensus sur
  - Les plates-formes "hardware"
  - Les « operating systems »
  - Les protocoles network
  - Les langages de programmation
- Une solution : Se mettre d'accord à un niveau plus haut ! … MDA
- Considérer les services de modélisation
  - Une transformation est un service de modélisation au même titre que l'exécution de modèles ou la génération de tests
  - Interopérabilité de services

OMG - MDA


Finance
Manufacturing
E-Commerce
Space
Telecom
Transportation
HealthCare
More...
PERVASIVE SERVICES
CORBA
UML
DIRECTORY
WEB SERVICES
XMI/XML
SECURITY
Model Driven Architecture
MOF
CWM
JAVA
.NET
TRANSACTIONS
EVENTS

L’approche MDA

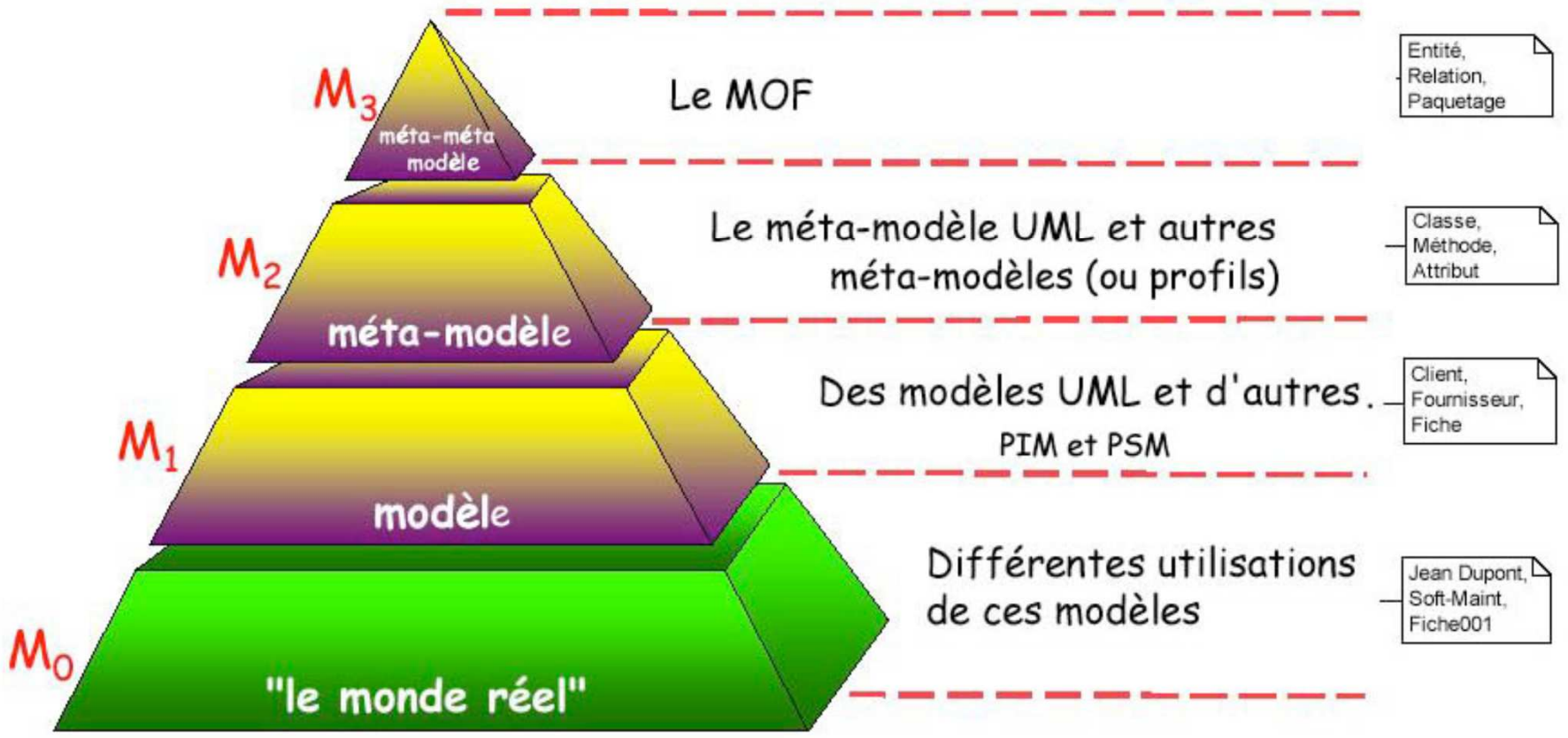
- Description
  - Le MDA est l'outil qui permet à une industrie de décrire ses fonctions indépendamment des implémentations
  - Penser l'application au niveau du modèle et laisser le soin de l'implémentation aux outils
  - Interopérabilité au niveau des modèles : Il s'agit d'avoir la possibilité d'écrire et de faire évoluer le modèle en fonction de l'organisation métier de l’application et non plus par les plate-formes.
    - Au niveau de l'organisation : PIM (Plateform Independant Model)
    - Au niveau des plate-formes : PSM (Plateform Specific Model).

L'approche MDA

- Une application complète de MDA : un PIM et un ou plusieurs PSM
  - Langage de description du MDA : UML.
  - L'application sera ensuite implémentée sur un large éventail de systèmes.
  - Les PSM peuvent communiquer entre eux en faisant intervenir plusieurs plate-formes pour échanger des données (CORBA par exemple)

- Avantages du MDA :
  - une architecture basée sur MDA est prête pour les évolutions technologiques.
  - plus grande facilité d'intégration des applications et des systèmes autour d'une architecture partagée.
  - une interopérabilité plus large permettant de ne pas être lié à une plate-forme.

Architecture à quatre niveaux



Chapitre II : OCL

Object Constraint Language

Contraintes UML

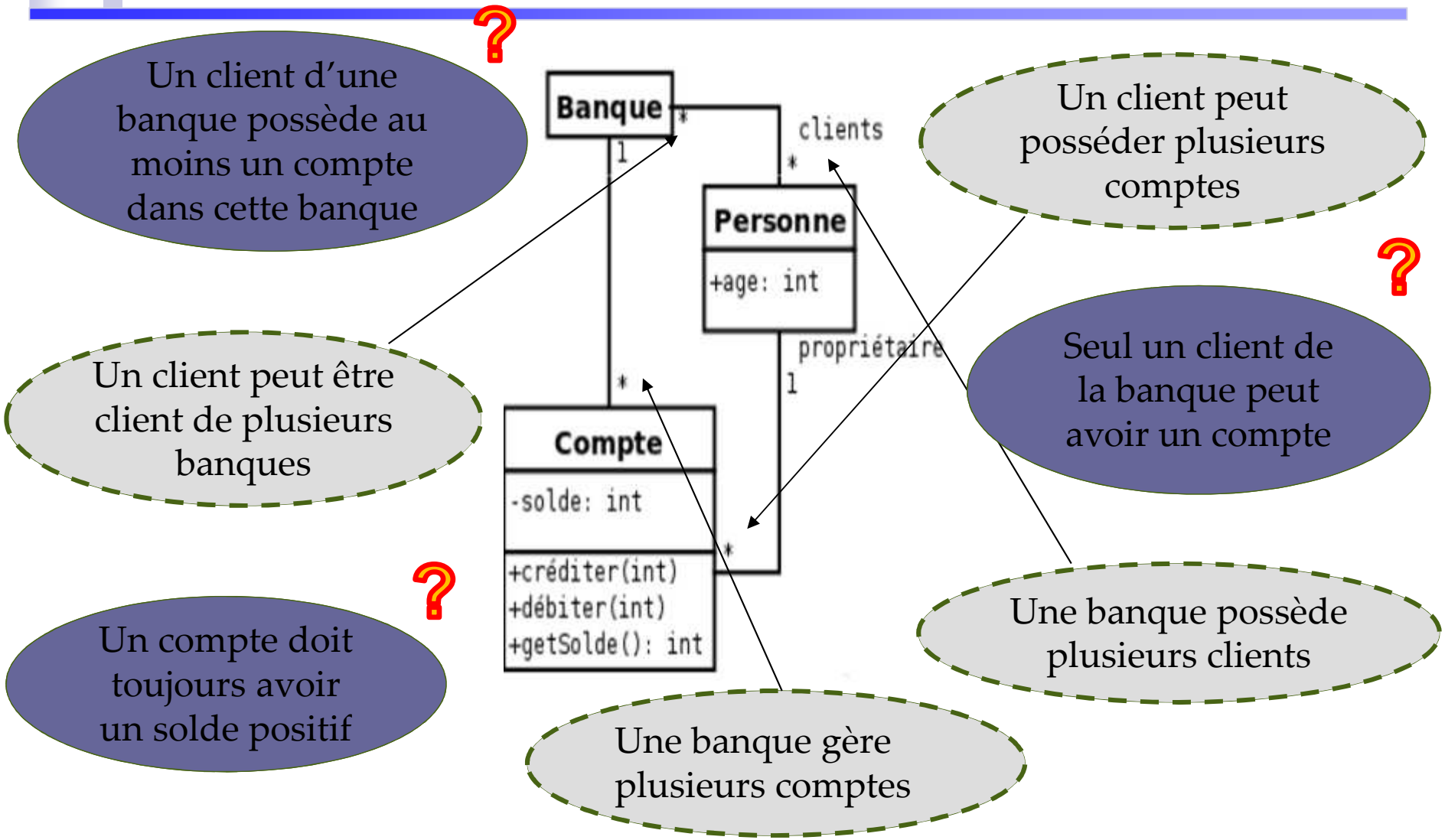
- Avec UML, on peut exprimer certaines formes de contraintes :
  - Contraintes de type : typage des propriétés, …
  - Contraintes structurelles :
    - les attributs dans les classes,
    - les différents types de relations entre classes (généralisation, association, agrégation, composition, dépendance),
    - la cardinalité et la navigabilité des propriétés structurelles,
    - …
  - Contraintes diverses : les contraintes de visibilité, les méthodes et classes abstraites, …

- Dans la pratique, toutes ces contraintes sont très utiles mais se révèlent insuffisantes.

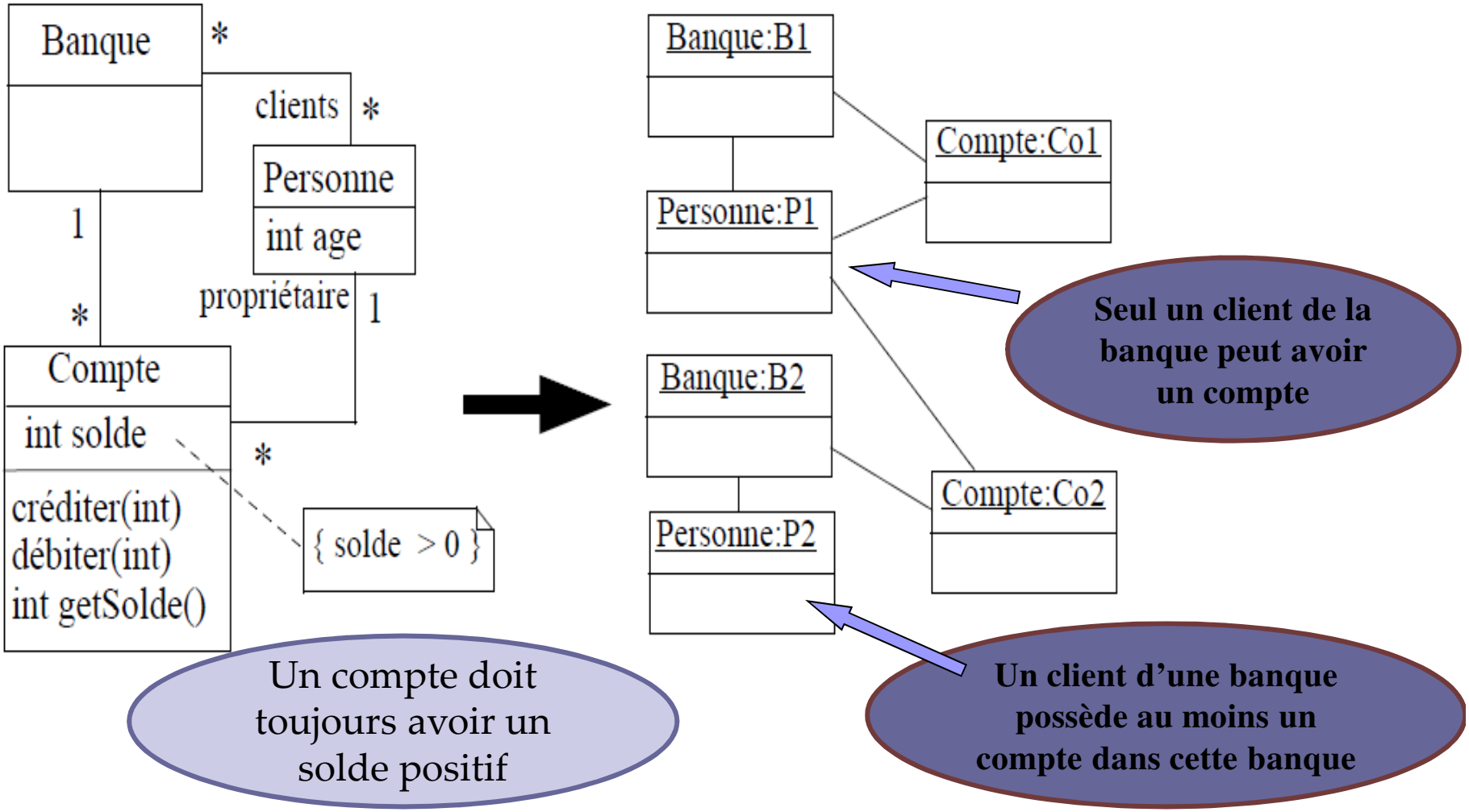
Exemple : Spécifications

- Développer un diagramme de classes qui respecte les contraintes suivantes :
  - Une personne peut posséder plusieurs comptes bancaires
  - Un compte appartient à un seul client
  - Une banque gère un ensemble de clients
  - Une banque gère un ensemble de comptes
  - Une personne peut être client dans plusieurs banques
  - Un compte appartient à une seule banque

Exemple : Spécifications



Validité du diagramme de classe à travers le diagramme d'objets



OMG - OCL
(Object Constraint Language)

- Pour spécifier complètement une application :
  - Diagrammes UML seuls sont généralement insuffisants
  - Nécessité de rajouter des contraintes
- Comment exprimer ces contraintes ?
  - Langue naturelle, mais manque de précision → compréhension pouvant être ambigüe
  - Langage formel avec sémantique précise
- OCL : *Object Constraint Language*
  - Un langage formel pour l'expression de contraintes, standardisé par l'OMG.
  - Permet d'ajouter des descriptions
    - Précises
    - Non Ambigües
    - Evite les désavantages des langages formels

OCL
(Object Constraint Language)

- Description
  - Développé d'abord en 1995 par IBM, puis combiné à UML en 1997 ;
  - Version 2.0 conforme à UML 2 et au MOF 2.0
  - Langage de contraintes orienté-objet
  - S'applique sur les diagrammes UML
  - Il se veut simple à écrire ainsi qu'à comprendre.
  - OCL se veut un langage formel permettant de décrire des contraintes de façon déterministe.
  - Langage formel (mais simple à utiliser) avec une syntaxe, une grammaire, une sémantique (manipulable par un outil)
  - OCL est purement un langage interrogatif : en aucun cas il ne peut modifier le modèle auquel il se rapporte.
  - On parle de langage sans effet de bord, ou *side-effect free*.

OCL : Utilisation

- OCL permet principalement d'exprimer des contraintes sur l'état d'un objet ou d'un ensemble d'objets :
  - Des invariants qui doivent être respectés en permanence
  - Des pré et post-conditions pour une opération :
    - Précondition : doit être vérifiée avant l'exécution
    - Postcondition : doit être vérifiée après l'exécution
  - Gardes sur transitions de diagrammes d'états ou de messages de diagrammes de séquence/collaboration
  - Des ensembles d'objets destinataires pour un envoi de message
  - Des attributs dérivés
  - Des stéréotypes
  - ...

OCL : Contexte d'une contrainte

- Contexte
  - Une expression OCL est toujours définie dans un contexte qui définit sa portée .
  - Le contexte permet d'associer une contrainte à n'importe quel élément du modèle : package, classe, interface, composant, opération, attribut …
  - Un contexte peut aussi dénoter un sous-élément comme une opération ou un attribut.
  - Dans un contexte, le mot-clé **self** dénote l'objet courant:
    - 'self' est implicite dans toute expression OCL
    - Équivalent à`this' en Java ou c++

- Syntaxe :
  - **context** <élément>

OCL : Contexte d'une contrainte

- Pour faire référence à un élément de type opération d'une classe C, il faut utiliser les :: comme séparateur (comme C::op).

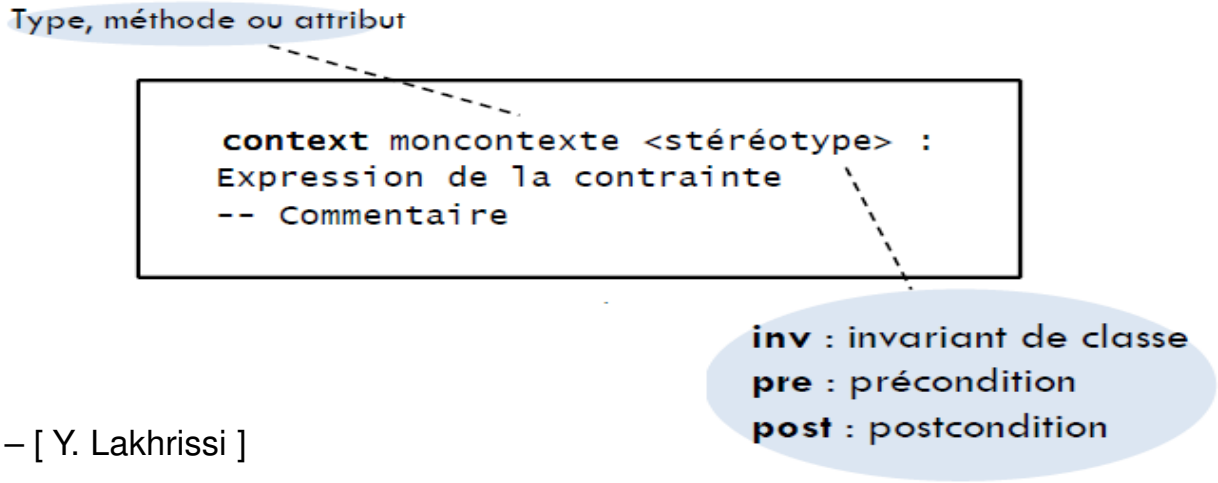
- Syntaxe pour préciser l'opération :
  - ***context ma_classe::mon_op(liste_param) : type_retour***

- Exemple :
  - Le contexte est la classe Compte :
    - ***context Compte***
    - L'expression OCL s'applique à la classe Compte, c'est-à-dire à toutes les instances de cette classe
  - Le contexte est l'opération getSolde() de la classe Compte:
    - ***context Compte::getSolde():integer***

Contrainte

- Une contrainte est la restriction d'une ou plusieurs valeurs d'un modèle UML

- Plusieurs types de contraintes :
  - **Invariant de Classe**
    - une contrainte qui doit être satisfaite par toutes les instances de la classe
  - **Precondition d'une opération**
    - une contrainte qui doit *être vérifiée* avant l' execution de l' opération
  - **Postcondition d'une opération**
    - une contrainte qui *doit être vérifiée* après l' execution de l' opération



Contrainte : Exemple(1/3)

-- Pour toutes les instances de la classe Compte, l'attribut solde doit toujours être positif

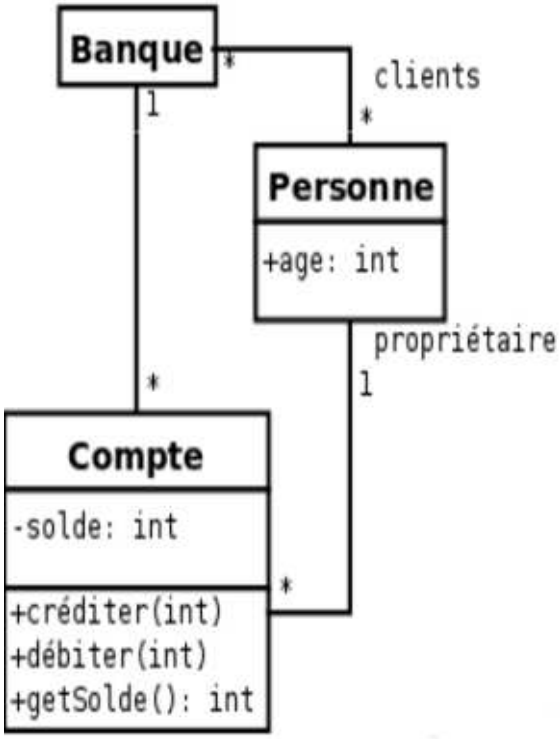
context Compte
inv: solde > 0

-- Le propriétaire d'un compte doit avoir l'age>18

context Compte
inv: self.proprietaire.age>18

--La somme à débiter doit être positive (méthode débiter)

context Compte::débiter(somme : Integer)
pre: somme > 0
post: solde = solde@**pre** – somme



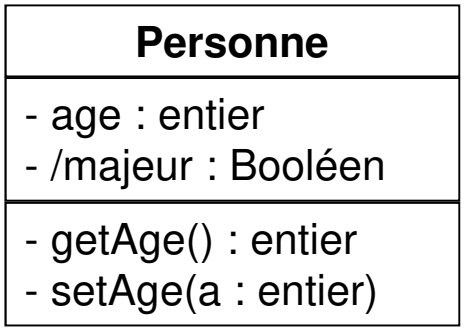
Remarque :

Dans la postcondition, deux éléments particuliers sont utilisables :

- Attribut **result** : référence la valeur retournée par l'opération
- mon_attribut@ **pre** : référence la valeur de mon_attribut avant l'appel de l'opération

Contrainte : Exemple(2/3)

context Personne::setAge(a : integer)
pre: (a <= 140)
and (a >=0)
and (a >= age)
post: age = a



context Personne **inv** ageBorné:
(age <= 140) and (age >=0)
-- l'âge ne peut dépasser 140 ans

Une contrainte peut être nommée par un label

context p : Personne **inv**:
(p.age <= 140) and (p.age >=0)

Un nom formel peut être donné à l'objet à partir duquel part l'évaluation

Contrainte : Exemple(3/3)

- Exemple 1 :
  - Le solde et la somme à débiter doivent être positifs pour que l'appel de l'opération « debiter » soit valide.
  - Après l'exécution de l'opération, l'attribut solde doit avoir pour valeur la différence de sa valeur avant l'appel et de la somme passée en paramètre.

context Compte::débiter(somme : Integer)
pre : somme > 0 and solde>somme
post : solde = solde@pre – somme

- Exemple 2 :
  - Le résultat retourné par la méthode getSolde doit être le solde courant

context Compte::getSolde() : Integer
post : result = solde

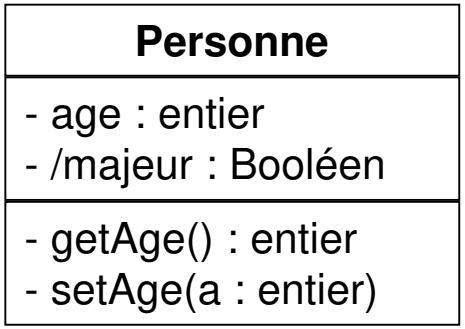
- Attention !
  - On ne décrit pas comment l'opération est réalisée mais des contraintes sur l'état avant et après son exécution.

Contrainte : init et derive

- ***init* : indique la valeur initiale d'un attribut**
- ***derive* : indique la valeur dérivée d'un attribut**
- Exemple :
  - L'âge initial de toute Personne est égal à 0
  - L'attribut dérivé « majeur » égal à True automatiquement si l'âge est >=18

context Personne::age : integer
Init : 0

context Personne::majeur : Boolean
Derive : age>=18



Contrainte : Bilan

- Une expression est définie sur un contexte qui identifie :
- ***une cible*** : l'élément du modèle sur lequel porte l'expression OCL

T	Type (Classifier : Interface, Classe...)	**context** Employee
M	Opération/Méthode	**context** Employee::raiseWage(inc: Int)
A	Attribut ou extrémité d'association	**context** Employee::job : Job

- ***le rôle*** : indique la signification de cette expression (pré, post, invariant...) et donc contraint sa cible et son évaluation.

rôle	cible	signification	évaluation
inv	T	invariant	toujours vraie
pre	M	précondition	avant tout appel de M
post	M	postcondition	après tout appel de M
body	M	résultat d'une requête	appel de M
init	A	valeur initiale de A	création
derive	A	valeur de A	utilisation de A
def	T	définir une méthode ou un attribut	

Contrainte : Bilan

- Dans une expression OCL, on peut utiliser :
  - Types de base: String, Boolean, Integer, Real.
  - Éléments du modèle UML
    - attributs,
    - classes
    - operations
  - Les associations du modèle UML
    - Y compris les noms de rôles
    - Conseil : utiliser des rôles dans UML pour simplifier OCL

Accès aux objets, navigation(1/2)

- Dans une contrainte OCL associée à un objet, on peut :
  - Accéder à l'état interne de cet objet (ses attributs)
  - Naviguer dans le diagramme : accéder de manière transitive à tous les objets (et leur état) avec qui il est en relation

- Nommage des éléments dans une expression OCL :
  - Attributs ou paramètres d'une opération : utilise leur nom directement
  - Objet(s) en association : utilise le nom de la classe associée (en minuscule) ou le nom du rôle d'association du coté de cette classe
  - Si cardinalité de 1 pour une association : référence un objet
  - Si cardinalité > 1 : référence une collection d'objets

Accès aux objets, navigation(2/2)

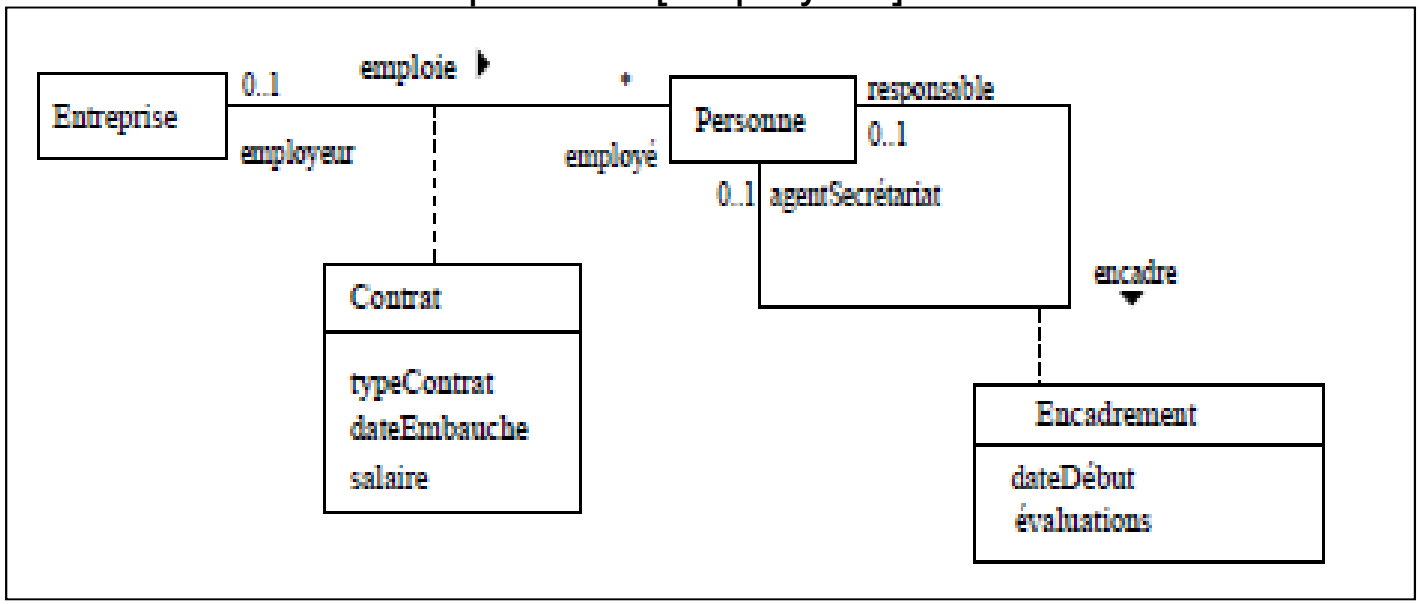
- Exemples, dans contexte de la classe Compte
  - **solde** : attribut référencé directement
  - **banque** : objet de la classe Banque (référencé via le nom de la classe) associé au compte
  - **propriétaire** : objet de la classe Personne (référence via le nom de rôle d'association) associée au compte

```
--Le propriétaire d'un compte doit avoir plus de 18 ans
context Compte
inv: propriétaire.age >= 18
```

  - **banque.clients** ➔ ensemble des clients de la banque
  - **banque.clients.age** ➔ ensemble des âges de tous les clients de la banque

Navigation vers les classes association

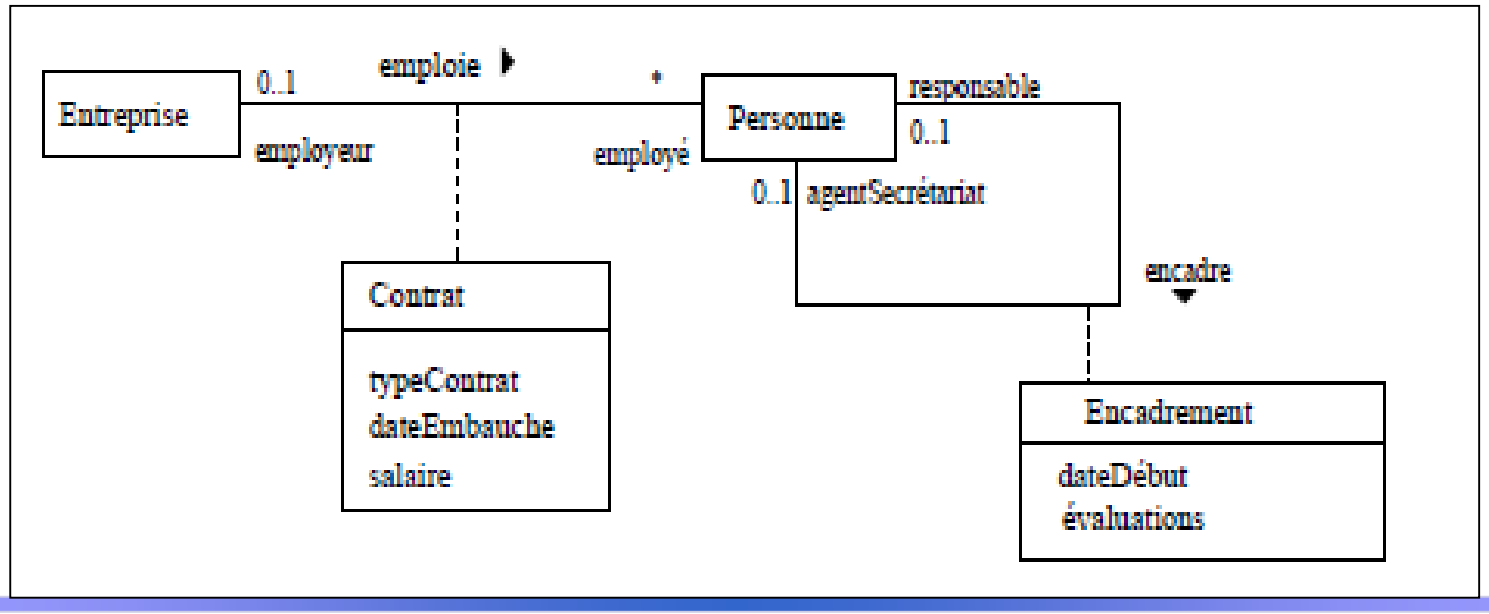
- Pour naviguer vers une classe association, on utilise le nom de la classe(en mettant le premier caractère en minuscule).
- Exemple : context p :Personne inv :
  p.contrat.salaire >= 0
- Ou encore, on précise le nom de rôle opposé (c'est même obligatoire pour une association réflexive) : context p :Personne inv :
  p.contrat[employeur].salaire >= 0



Navigation vers les classes association (Exercice)

- Exprimez les contraintes suivantes:

1. L'âge du responsable est sup de l'âge de l'agent
2. le salaire d'un agent de secrétariat est inférieur à celui de son responsable;
3. un agent de secrétariat a un type de contrat "agentAdministratif" ;
4. un agent de secrétariat a une date d'embauche antérieure à la date de début de l'encadrement (on suppose que les dates sont des entiers) ;

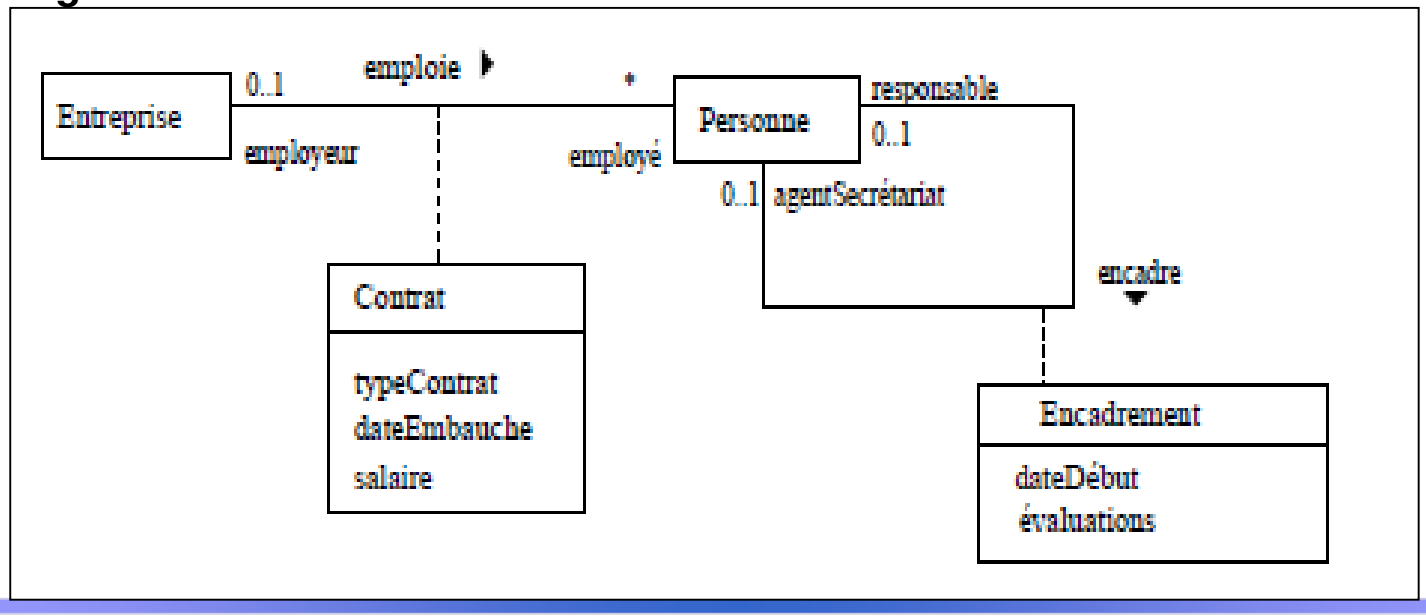


Navigation vers les classes association (Solution)

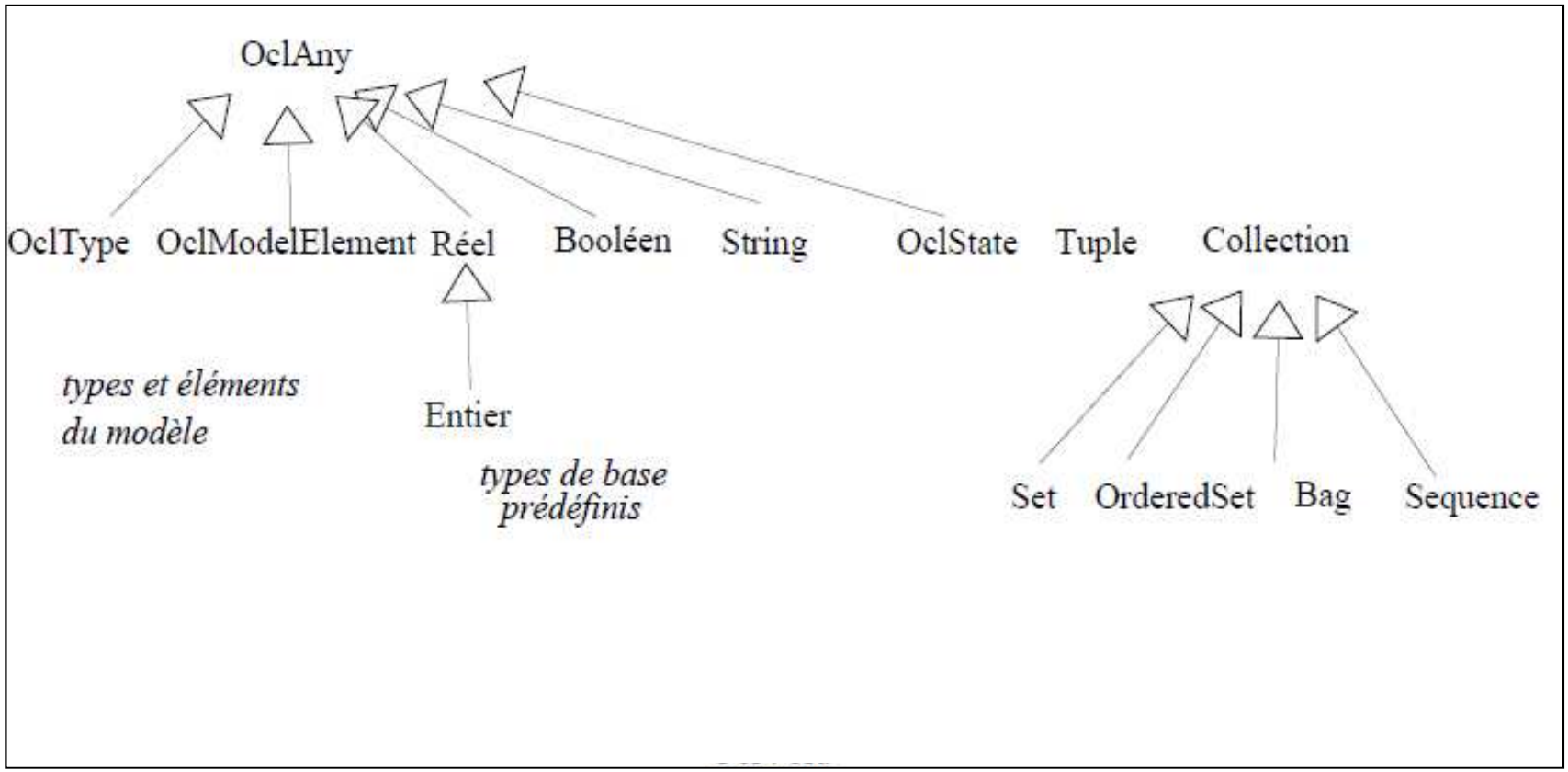
2. **context e :Encadrement inv :**
   e.responsable.contrat.salaire >= e.agentSecrétariat.contrat.salaire

3. ***context e :Encadrement inv :***
   e.agentSecrétariat.contrat.typeContrat='agentAdministratif'

4. **context e :Encadrement inv :**
   e.agentSecrétariat.contrat.dateEmbauche <= e.dateDebut



Types OCL


OclAny
OclType
OclModelElement
Réel
Booléen
String
OclState
Tuple
Collection
types et éléments du modèle
Entier
types de base prédéfinis
Set
OrderedSet
Bag
Sequence

Types OCL : Types de base

- Integer
  - 1, -2, 145
  - Opérateur : = <> + - * / abs div mod < > <= >=
- Real
  - 1.5, -123.4
  - Opérateur : = <> + - * / abs floor round < > <= >=
- String
  - 'bonjour'
  - s.concat(t), s.size(), s.toLower(), s.toUpper(), substring(,),replaceAll(,)…

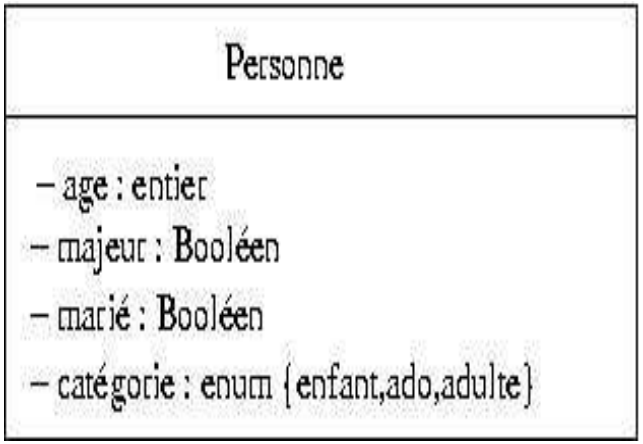
Types OCL : Types de base

- Booléen
  - Les constantes s'écrivent : true, false
  - Les opérateurs sont les suivants :
    - **= or xor and not**
    - b1 **implies** b2
    - **if** b **then** expression1 **else** expression2 **endif**

Exemple :

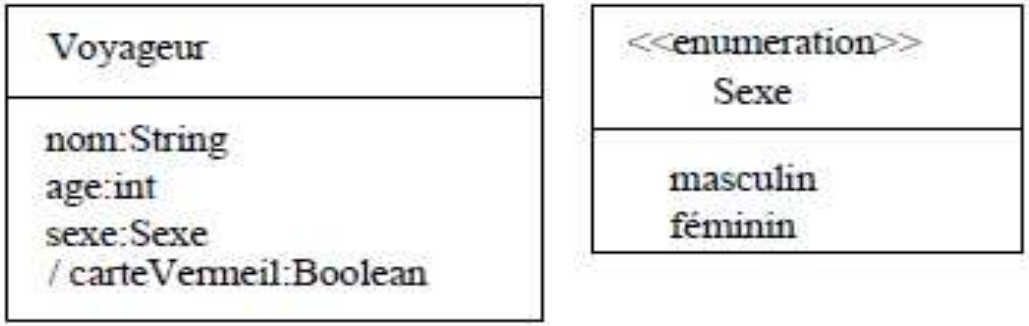
```
context Personne inv :
    marié implies majeur

context Personne inv :
   if age >=18 then
            majeur=vrai
   else majeur=faux
   endif
```



Types OCL : Types de base

- Exemple
  - Ecrivez pour le diagramme ci dessous la contrainte qui caractérise l'attribut dérivé carteVermeil. Un voyageur a droit à la carte vermeil si c'est une femme de plus de 60 ans ou un homme de plus de 65 ans.



Solution:

context Voyageur inv :
carteVermeil = ((age >=65) or ((sexe=Sexe::féminin) and (age >=60)))

Cette contrainte peut également s'écrire avec derive.
context Voyageur::carte:Boolean derive:
((age >=65) or ((sexe=Sexe::féminin) and (age >=60)))

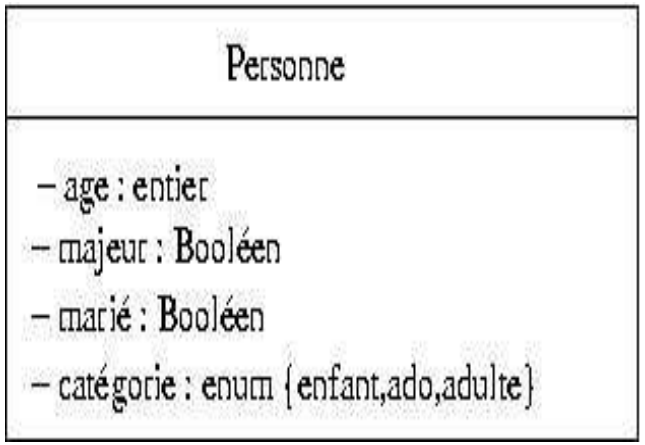
Types : Règle de précédence

- Ordre de précédence pour les opérateurs/primitives :
  - @pre
  - . et ->
  - not et -
  - * et /
  - + et -
  - if then else endif
  - >, <, <= et >=
  - = et <>
  - and, or et xor
  - Implies
- Les parenthèses permettent de changer cet ordre

Types OCL : Type énuméré

- Leur syntaxe est la suivante :
  - ***<nom_type_enuméré>::valeur***
- Le nom du type est déduit de l'attribut de déclaration
  - catégorie => Catégorie
- Exemple :

context Personne inv:
if age <=12
then categorie = Categorie::enfant
else if age <=18
then categorie = Categorie::ado
else categorie = Categorie::adulte
endif
endif

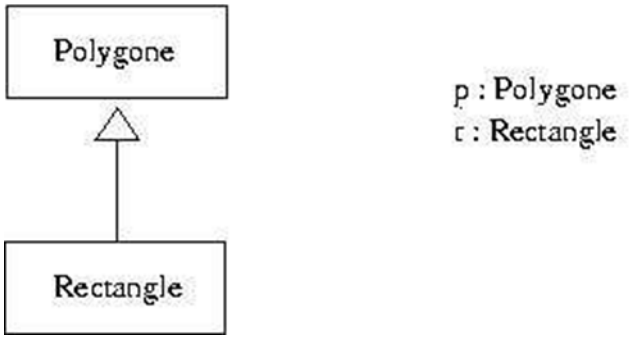


Types OCL : Type de modèle

- Les types des modèles utilisables en OCL sont les classifieurs, donc:
  - les classes, les interfaces et les associations
- Manipulation de la relation de spécialisation:
  - oclAsType(t) (l'objet est casté en t)
  - oclIsTypeOf(t) (vrai si t est supertype direct)
  - oclIsKindOf(t) (vrai si t est supertype indirect)

Exemple :

p = r
p.oclAsType(Rectangle)
r.oclIsTypeOf(Rectangle) (vrai)
r.oclIsKindOf(Polygone) (vrai)



Types OCL : Type de modèle

- Exercice :

  En supposant l'existence d'un attribut hauteur dans la classe Rectangle et d'une méthode hauteur() :Réel dans Polygone, écrivez une contrainte dans Polygone disant que le résultat de hauteur() :Réel vaut hauteur pour les polygones qui sont des rectangles, sinon 0.

- Solution 1:

```
context Polygone::hauteur()  : Real
post :
if self.oclIsTypeOf(Rectangle)
        then   result=self.hauteur
else  result=0
endif
```

- Solution plus générale:

```
context Polygone::hauteur() : Real
post :
if self.oclIsKindOf(Rectangle)
        then   result=self.oclAsType(Rectangle).hauteur
else  result=0
endif
```

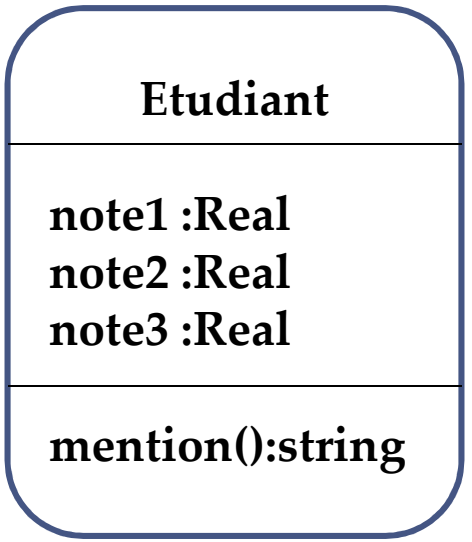
Types OCL : Type de modèle

- OclModelElement :
  - énumération des éléments du modèle
- OclType :
  - énumération des types du modèle
- OclAny :
  - tout type autre que Collection
- OclState :
  - pour les diagrammes d'états

Définition de variables et d'opérations

- Syntaxe pour définir une nouvelle variable :
  - **let** variable : type **=** expression1 **in** expression2
- Exercice
  - soit la classe **Etudiant**, disposant de 3 notes et munie d'une opération **mention** qui retourne la mention de l'étudiant sous forme d'une chaîne de caractères. Ecrivez les contraintes postcondition sur l'opération mention() en utilisant **let** pour calculer la moyenne.

```
context Etudiant::mention() : String
post :
let moyenne : Real =(note1+note2+note3)/3      in
result=
if (moyenne >= 16) then
            'très bien'
else if (moyenne >= 14) then
            'bien'
     else  'moins bien'
     endif
endif
```



Définition de variables et d'opérations

- Si on veut définir une variable(opération) utilisable dans plusieurs contraintes de la classe, on peut utiliser la construction def.
  - Syntaxe : **def** : <déclaration> = <requête>

Exemple :

context Personne
def : ageCorrect(a :Real) :Boolean = a>=0 and a<=140

Exp1- context Personne inv :
ageCorrect(age) - - l'âge ne peut depasser 140 ans

Exp2- context Personne::setAge(a :integer)
pre : ageCorrect(a) and (a >= age)

Let et def

- The Let expression allows a variable to be used in one OCL expression. To enable reuse of variables/operations over multiple OCL expressions one can use a Constraint with the stereotype «definition»,

- context Person
  - def: income : Integer = self.job.salary->sum()
  - def: nickname : String = 'Little Red Rooster'
  - def: hasTitle(t : String) : Boolean = self.job->exists(title = t)

Keywords

- and
- body
- context
- def
- derive
- else
- endif
- endpackage
- false
- if
- implies
- in
- init
- or
- package
- post
- pre
- self
- static
- then
- true
- xor
- inv
- invalid
- let
- not
- null
- Bag
- Boolean
- Collection
- Integer
- OclAny
- OclInvalid
- OclMessage
- OclVoid
- OrderedSet
- Real
- Sequence
- Set
- String
- Tuple
- UnlimitedNatural

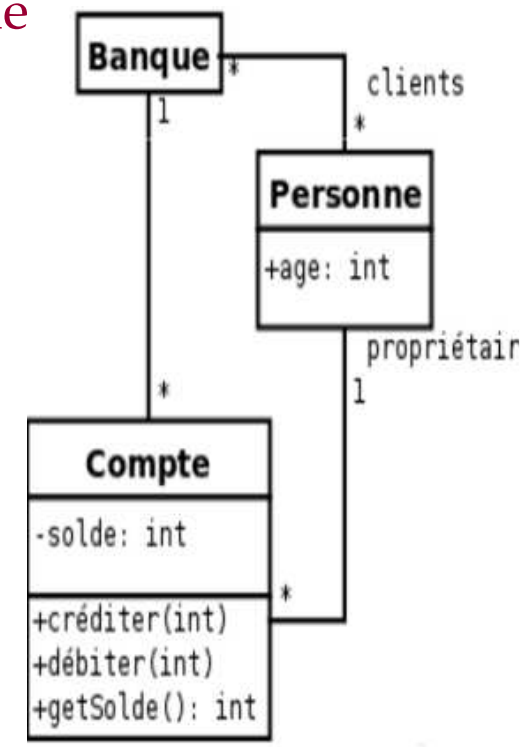
Définition de variables et d'opérations (Exercice)

- Utilisez le **Let ou Def** pour écrire une contrainte dans le contexte Personne qui spécifie qu'une personne majeure doit avoir de l'argent.
- sum() : fait la somme de tous les objets de l'ensemble

context Personne **inv** :
let argent : integer = compte.solde -> sum()
in age >= 18 **implies** argent > 0

context Personne **def** : argent() : integer =
compte.solde -> sum()

context Personne **inv** :
age >= 18 **implies** argent() > 0



Types OCL : Le type Collection

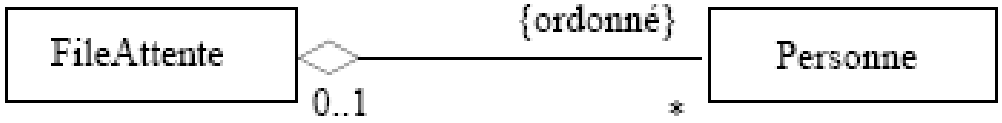
- Dans les expressions de navigation (a.b), le type du résultat dépend de la cardinalité de l'association :
  - Cardinalités simples : 0 ou 1, rend un objet.
  - Cardinalités multiples, rend une collection

Types OCL : Le type Collection

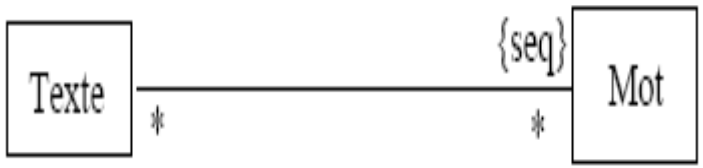
- Les types suivants sont fournies pour les collections, tous sont des sous-types d'une classe abstraite Collection:
  - Set : La notion mathématique d'ensemble (pas de doublons, pas d'ordre) => Exemple : { 1, 4, 3, 5 }
  - OrderedSet : La notion mathématique d'ensemble ordonné =>Exemple : { 2, 4, 6, 8 }
  - Bag : La notion mathématique de famille (doublons possibles, pas d'ordre) => Exemple : { 1, 4, 1, 3, 5, 4 }
  - Sequence : La notion mathématique de famille, et les éléments sont ordonnés => Exemple : { 1, 1, 3, 4, 4, 5 }
- Remarque : il est possible de définir des collections de collections,=> Exemple : Set { 2, 4, Set {6, 8 }}

Types OCL : Le type Collection
Exemple : résultats de navigation

- **OrderedSet** :obtenu en naviguant vers une extremité d'association munie de la contrainte « ordonné »
  - Exemple : l'expression **self.personne** dans le contexte **FileAttente**



- **Sequence** :obtenu en naviguant vers une extrémité d'association munie de la contrainte **seq**,
  - Exemple: l'expression **self.mot** dans le contexte **Texte**.



Opérations sur les collections(1/2)

- Appel : Les opérations sur une collection sont mentionnées avec ->
  - **Syntaxe d'utilisation** : **objetOuCollection -> primitive**

- OCL propose un ensemble de primitives utilisables sur les ensembles :
  - **size() : integer** : retourne le nombre d'éléments de l'ensemble
  - **count(unObjet :T) : integer** : donne le nombre d'occurrences de unObjet.
  - **sum() : T** : Addition de tous les éléments de la collection (qui doivent supporter une opération + associative)
  - **isEmpty() : Boolean** : retourne vrai si l'ensemble est vide
  - **notEmpty() : Boolean** : retourne vrai si l'ensemble n'est pas vide
  - **includes(obj: T) : Boolean** : vrai si l'ensemble inclut l'objet obj
  - **excludes(obj:T) : Boolean** : vrai si l'ensemble n'inclut pas l'objet obj

Opérations sur les collections(2/2)

- **forAll(uneExpression :OclExpression):Boolean** :Vaut vrai ssi une expression est vraie pour tous les éléments de la collection.
- **exists(uneExpression):Boolean :**Vrai ssi au moins un élément de la collection satisfait uneExpression
- **isUnique(uneExpression):Boolean** : Vrai ssi une expression s'évalue avec une valeur distincte pour chaque élément de la collection.
- **one(uneExpression):Booleen** :Vrai ssi un et un seul élément de la collection vérifie uneExpression.
- **any(uneExpression):T** :Retourne n'importe quel élément de la collection qui vérifie uneExpression.
- **sortedBy(uneExpression):Séquence** :Retourne une séquence contenant les éléments de la collection tries par ordre croissant suivant le critère uneExpression
- **select(expression):Séquence** rend la sous collection d'objets satisfaisant l'expression
- **reject(expression):Séquence** rejette de la collection les objets satisfaisant l'expression

select et reject

- **Select(expression):Séquence** rend la sous collection d'objets satisfaisant l'expression
- **context** Company **inv**:
  - self.employee->select(age > 50)->notEmpty()
  - self.employee->select(p | p.age > 50)->notEmpty()
  - self.employee.select(p : Person | p.age > 50)->notEmpty()
- **reject(expression):Séquence** rend la sous collection d'objets ne satisfaisant pas l'expression
- As an example, specify that the collection of all the employees who are not married is empty:
- context Company inv:
  - self.employee->reject(isMarried)->isEmpty()

Collect Operation

- the select and reject operations always result in a sub-collection of the original collection. When we want to specify a collection that is derived from some other collection, but which contains different objects from the original collection, we can use a *collect* operation.
- An example: specify the collection of *birthDates* for all employees in the context of a company:
  - self.employee->collect(birthDate)
  - self.employee->collect(person | person.birthDate)
  - self.employee->collect(person : Person | person.birthDate)
- The Bag resulting from the *collect* operation always has the same size as the original collection.
- It is possible to make a Set from the Bag, by using the asSet property on the Bag. The following expression results in the Set of different *birthDates* from all employees of a Company: self.employee->collect(birthDate)->asSet()

forAll operation

- The forAll operation in OCL allows specifying a Boolean expression, which must hold for all objects in a collection:
- The result is true if the boolean-expression-with-v is true for all elements of collection. If the boolean-expression-with-v is false for one or more v in collection, then the complete expression evaluates to false.
- For example, in the context of a company:
- context Company
  - inv: self.employee->forAll(age <= 65)
  - inv: self.employee->forAll(p | p.age <= 65)
  - inv: self.employee->forAll(p : Person | p.age <= 65)
- **context** Company **inv**:
  - self.employee->forAll(e1, e2 : Person | e1 <> e2 implies e1.forename <> e2.forename)

exists operation

- The *exists* operation in OCL allows you to specify a Boolean expression that must hold for at least one object in a collection:
  - **context** Company **inv**: self.employee->exists(forename = 'Jack')
  - **context** Company **inv**: self.employee->exists(p | p.forename = 'Jack')
  - **context** Company **inv**: self.employee->exists(p : Person | p.forename = 'Jack')

Iterate Operation

- The *iterate* operation is slightly more complicated, but is very generic.
- The operations *reject, select, forAll, exists, collect* can all be described in terms of *iterate*.
- An accumulation builds one value by iterating over a collection.
- collection->iterate(elem : Type; acc : Type = <expression> | expression-with-elem-and-acc)

Exercice 1

- Il n'existe pas de clients de la banque dont l'âge est inférieur à 18 ans (not : prend la négation d'une expression)

  context Banque inv:

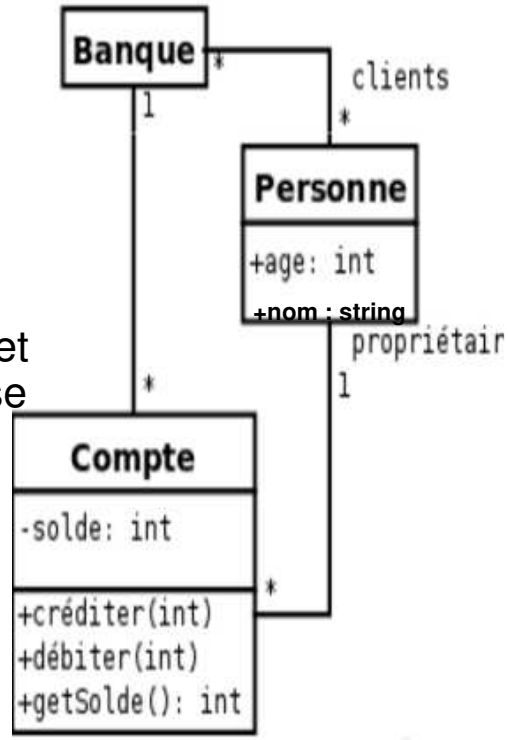
  not(clients -> exists (age < 18))

- Il n'existe pas deux instances de la classe Personne pour lesquelles l'attribut nom a la même valeur : deux personnes différentes ont un nom différent

  context Personne inv:

  Personne.allInstances() -> forAll(p1, p2 | p1 <> p2 implies p1.nom <> p2.nom)

  - Remarque : allInstances() : primitive s'appliquant sur une classe (et non pas un objet) et retournant toutes les instances de cette classe



- Si une personne possède au moins un compte bancaire, alors elle est cliente d'au moins une banque

  context Personne inv:

  compte -> notEmpty() implies banque -> notEmpty()

- Le propriétaire d'un compte doit être client de la banque

  context Compte inv :

  banque.clients -> includes (propriétaire)

Exercice 2

1- Developpez la fonction chercherClient applicable dans le contexte d'une Banque, elle reçoit une personne en paramètre et retourne un boolean resultat de la recherche de la personne dans la liste des clients

2-Ecrire la post condition d'une méthode salariésTriés dans le contexte de la classe **Entreprise**. Cette méthode retourne les employés ordonnés suivant leur salaire.

3-Ecrire la post condition d'une méthode "retraités" dans le contexte d'une **Entreprise** qui retourne les employés âgés de plus de 60 ans.

Solution:

1- context *Entreprise : : salariesTries():OrderedSet(Personne)*
post *: result = self.employe->sortedBy(p \| p.contrat.salaire)*

2- context *Entreprise :: retaités(): sequence(Personne)*
post *: result = self.employé->select(p :Personne\|p.age>=60)*

Exercice 3
Itération : forAll()

- Exemple1: Dans une entreprise, tous les employés aient plus de 16 ans.

  context e :Entreprise **inv :**

  e.employe->forAll(emp :Personne | emp.age >= 16)

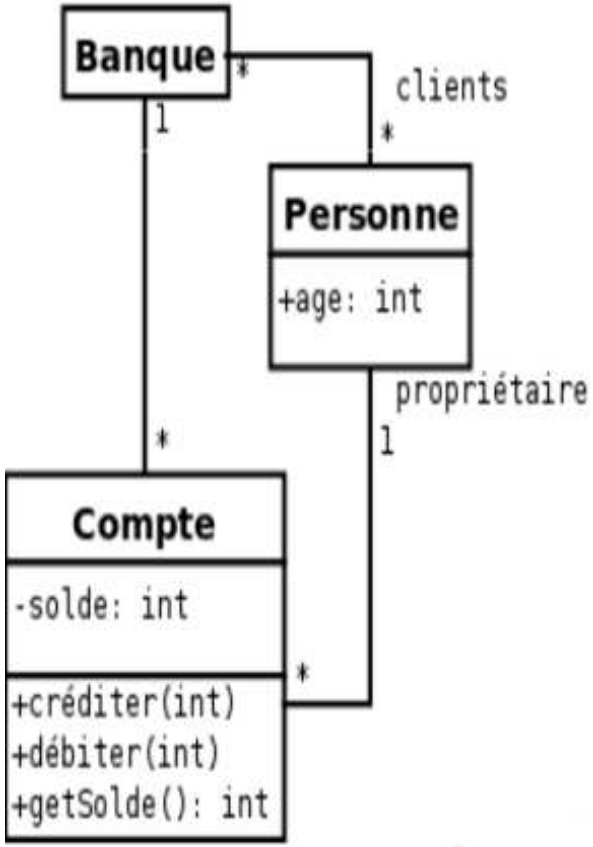
- Exemple 2: En supposant que la classe Personne dispose d'un attribut nom, on peut admettre que deux employés différents n'ont jamais le même nom.

  context e :Entreprise **inv :**

  e.employe->forAll(e1,e2 :Personne | e1 <> e2 **implies** e1.nom <> e2.nom)

Exercice 4

- En se basant sur ce diagramme UML, écrire les contraintes OCL suivantes :
  - Un solde ne doit pas dépasser 10.000.000
  - Pour les clients moins de 20ans le solde ne doit pas dépasser 100.000
  - La valeur maximale à débiter est inférieure à 10.000
  - Le nombre de client d'une banque doit être supérieur à 500
  - Les clients dont le solde est supérieur à 1.000.000 peuvent débiter jusqu'à 100.000 sinon jusqu'à 10.000



Exercice 5

- Modéliser le système suivant :
  - Une famille est composée de plusieurs personnes, au minimum 2 (les parents).
  - Les enfants portent le même nom que leur père.
  - L'âge des parents est supérieur strictement à l'âge de leur enfants.
  - La date du mariage des parents est antérieure de la date de naissance des enfants.
  - Les enfants doivent posséder des prénoms différents
  - Le mariage doit réunir deux personnes de sexe différent.
  - Il est interdit que les membres d'une même famille puissent se marier entre eux (à l'exclusion du mariage des parents).
  - Jusqu'à l'âge de 16ans, un enfant ne peut pas habiter en dehors du logement des parents

Exercice 6

- Modéliser le système suivant :

1. Une personne peut posséder plusieurs voitures
2. Chaque voiture a une marque, une date de mise en service et possède quatre pneus
3. Les pneus ont une marque. Deux marques sont possibles : GG et PP
4. Les voitures de marque "GCar" doivent avoir des pneus de marque "GG"
5. Les voitures de marque "Pcar" doivent avoir des pneus arrière de marque "PP"
6. Les voitures de plus de 20ans doivent se mettre hors service
7. Une voiture dont l'émission du CO2 dépasse la valeur 10 est mise hors service
8. Seules les personnes de plus de 30ans peuvent posséder des voitures de marque "PCar"
9. L'accélération est refusée si la vitesse de la voiture dépasse 140km/h
10. En cas de panne un signal d'alarme est déclenché pendant 1min

Chapitre III : Méta-Modélisation

1. **Notions de Modèle ; Méta-modèle ; Méta-méta-modèle**

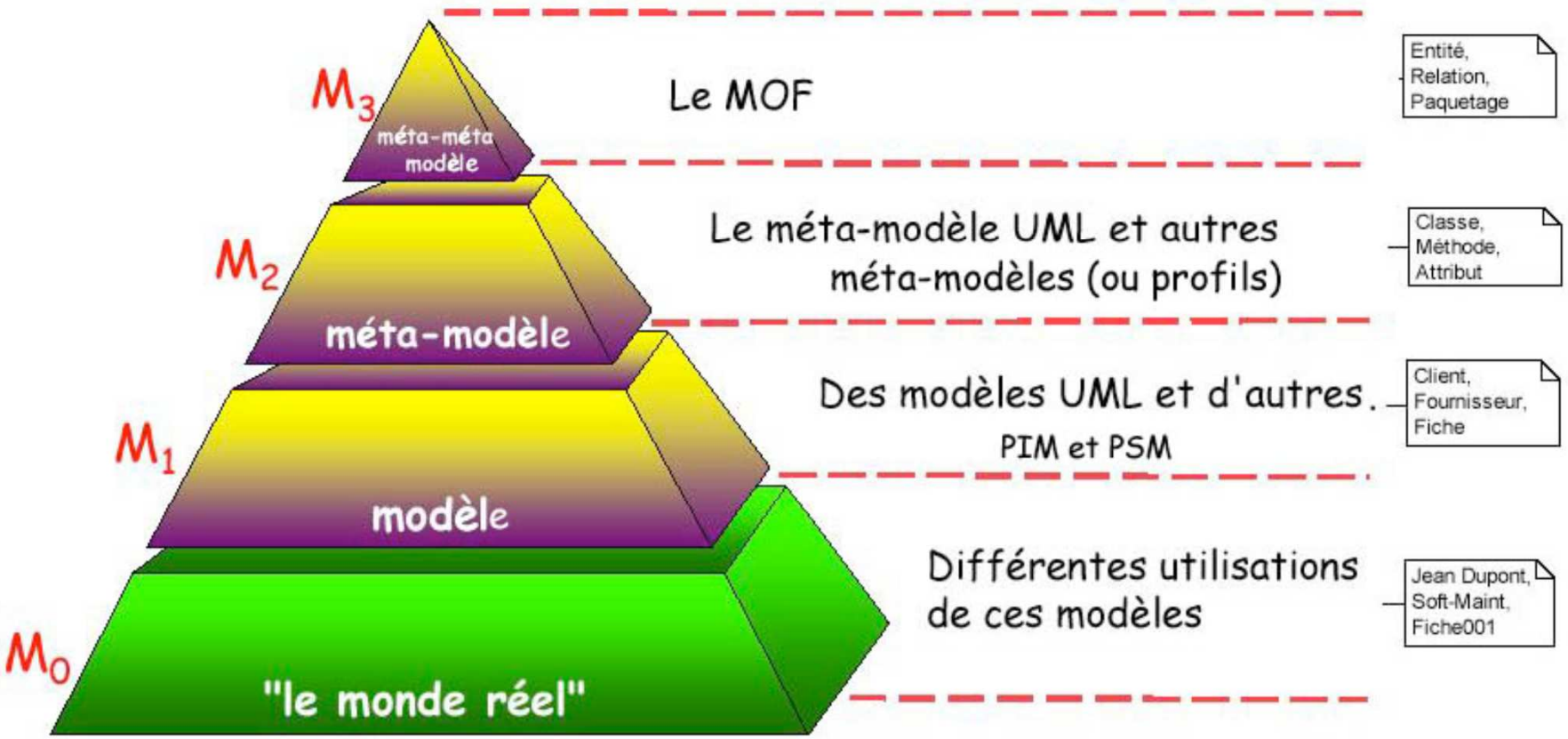
2. **Extension de UML / Profils UML**

Notion de métamodèle

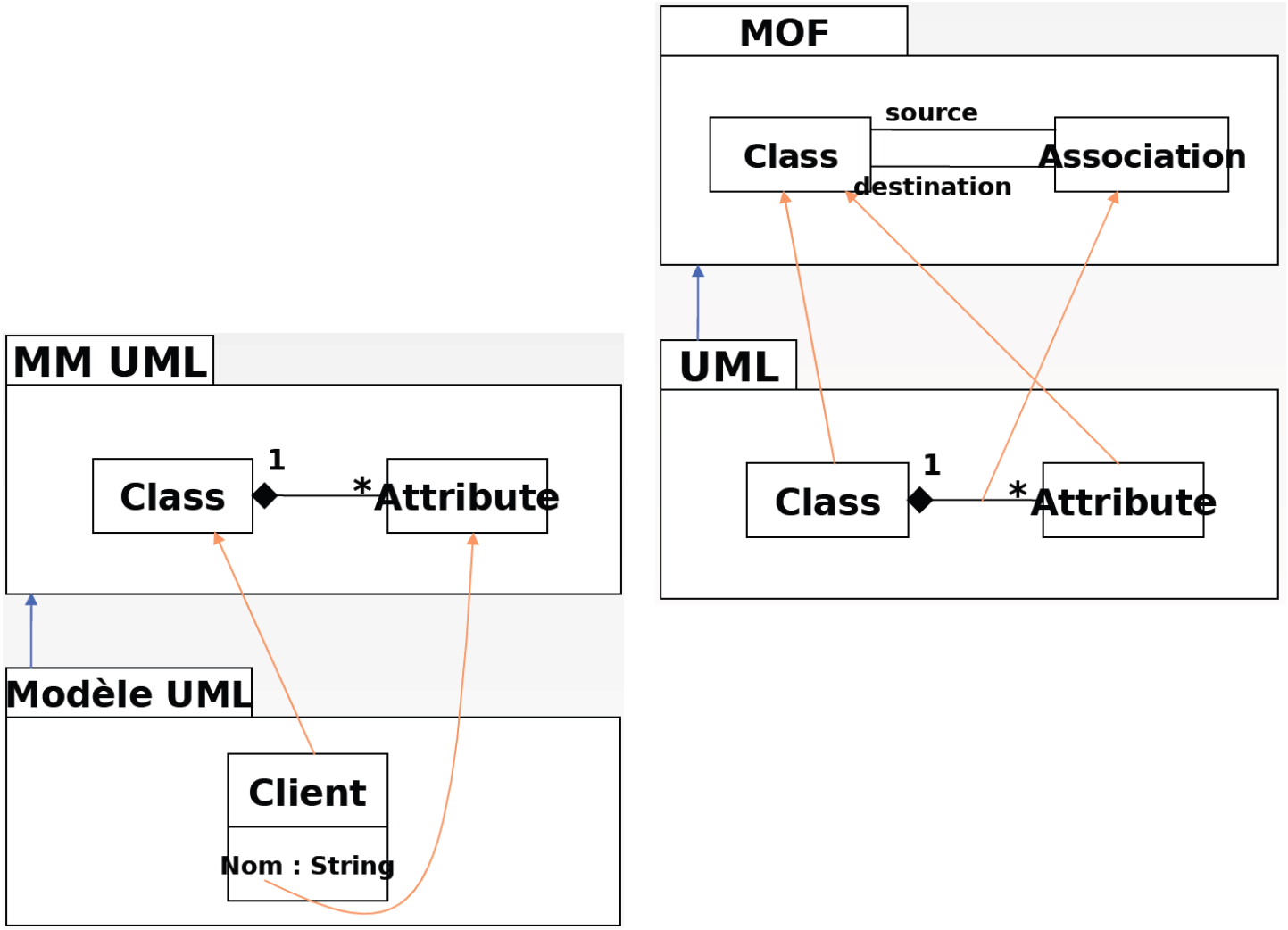
- Tout modèle doit respecter la structure définie par son métamodèle.
  - Les métamodèles fournissent la définition des entités d'un modèle, ainsi que les propriétés de leurs connexions et de leurs règles de cohérence. MOF les représente sous forme de *diagrammes de classes.*

- Métamodèle et sémantique d'un modèle !
  - Métamodèle et sémantique d'un modèle sont deux choses différentes. Un métamodèle définit la structure d'un ensemble de modèles mais fournit peu de précision sur leur sémantique.
  - Définir qu'un modèle UML contient des packages, qui, eux-mêmes, contiennent des classes, ne renseigne en rien sur la signification des concepts de package et de classe.

Architecture à quatre niveaux



Modèle-Métamodèle-Métamétamodèle

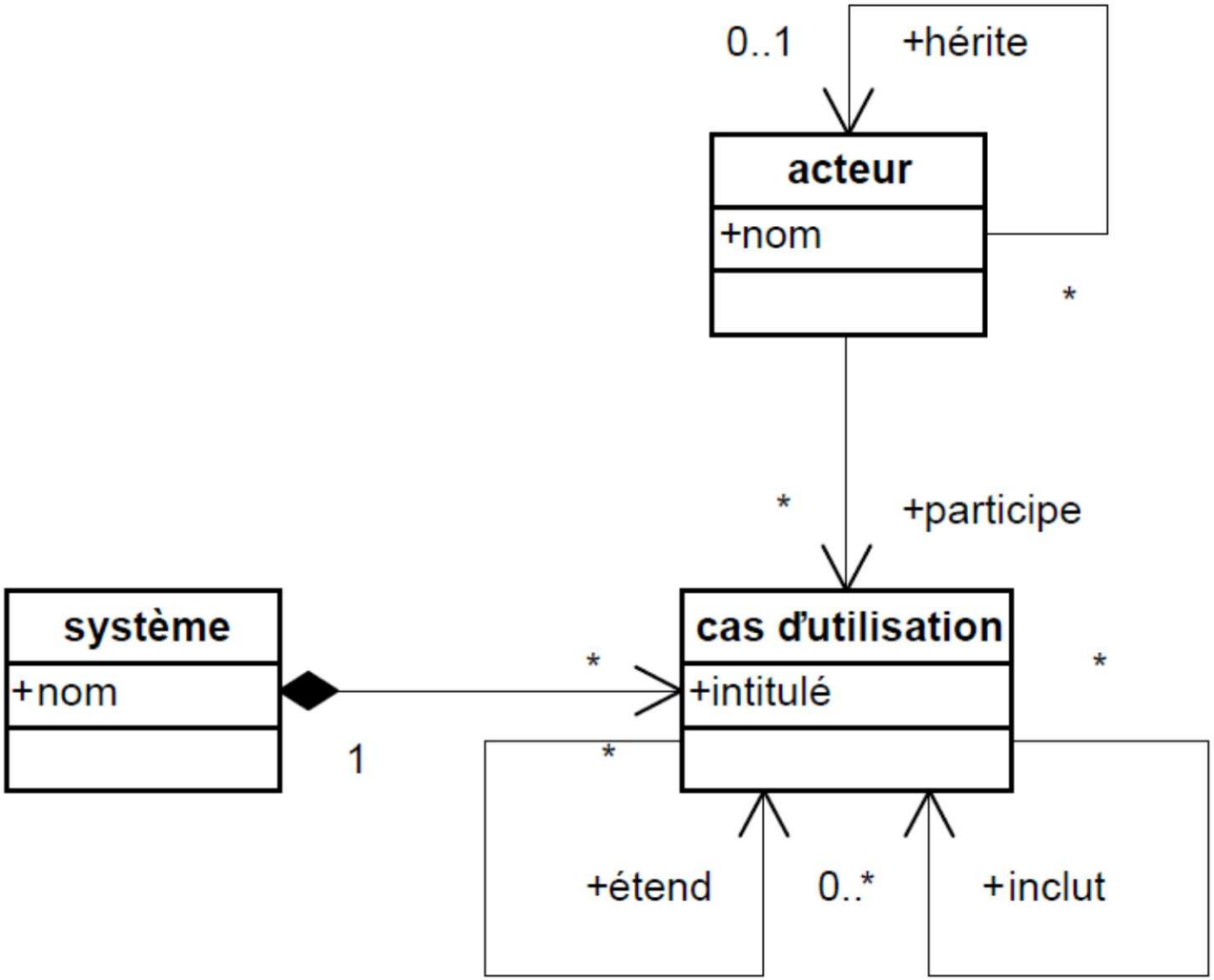


Exercice1 (1/4)

Métamodèle du diagramme des Use Case

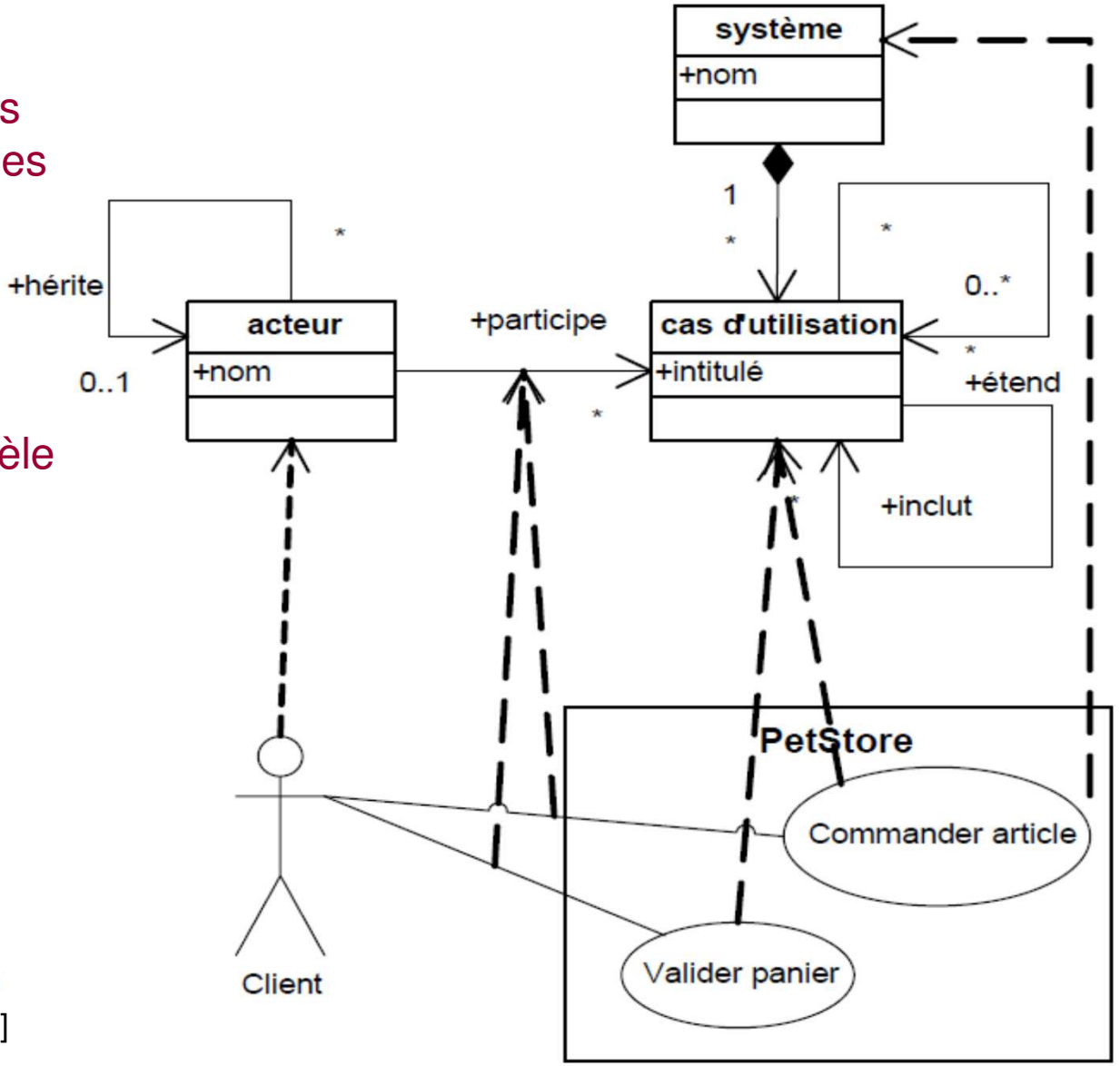
- Essayons de développer un métamodèle pour le diagramme des cas d'utilisation (version allégée bien sur !)
  - L'objectif n'est pas d'expliquer comment réaliser des diagrammes de cas d'utilisation.
  - L'objectif est de modéliser les éléments que nous utilisons pour réaliser un diagramme de cas d'utilisation.
  - Le résultat de cette modélisation = métamodèle des diagrammes de cas d'utilisation.
- Voici les exigences : (NB : cette définition n'est pas standard)
  - 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.

Exercice1 (2/4)



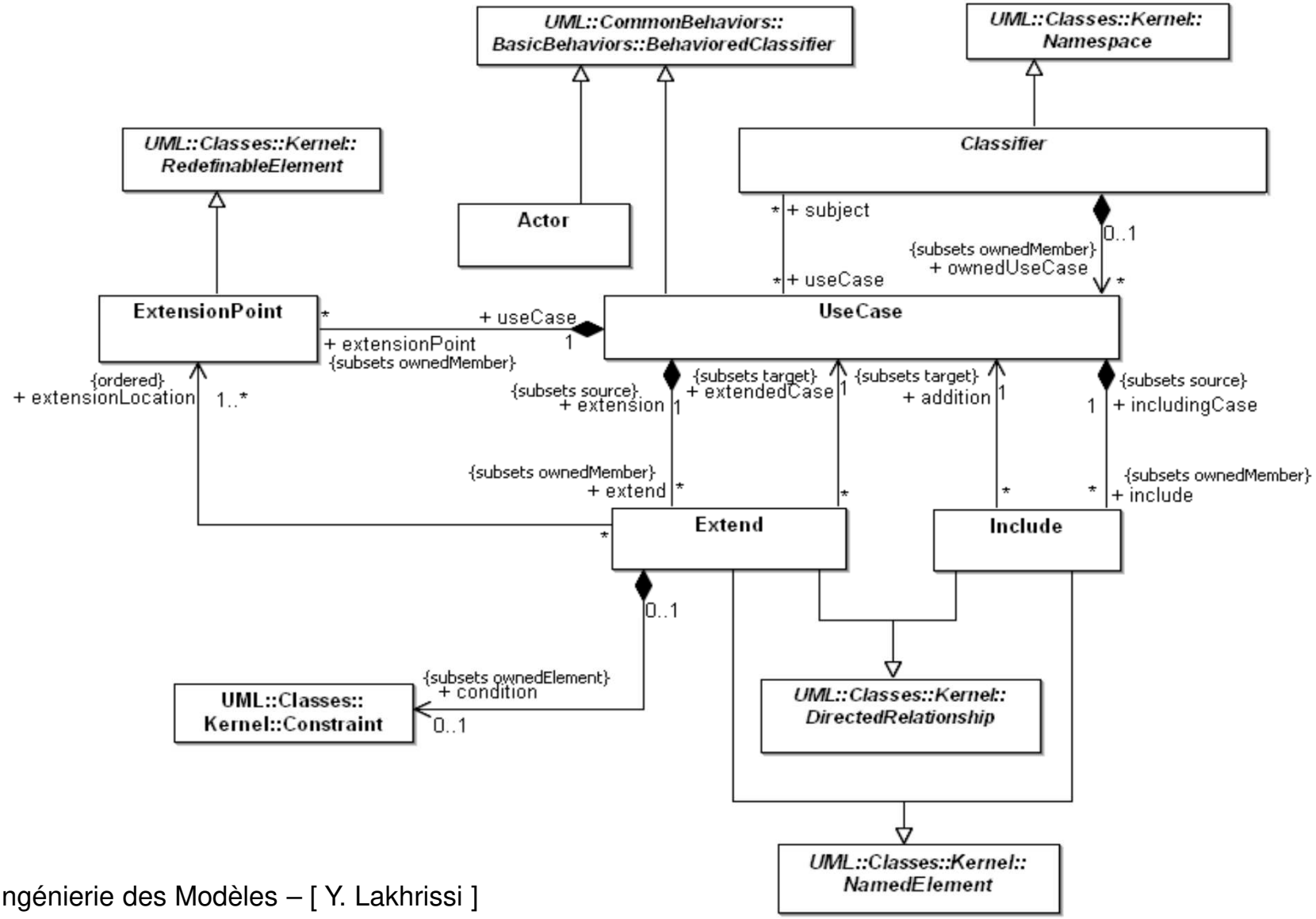
Exercice1 (3/4)

- Les flèches en pointillés peuvent être considérées comme des relations d'*instanciation.*
- On dit alors qu'un modèle est l'instance de son métamodèle.



Exercice1 (4/4)

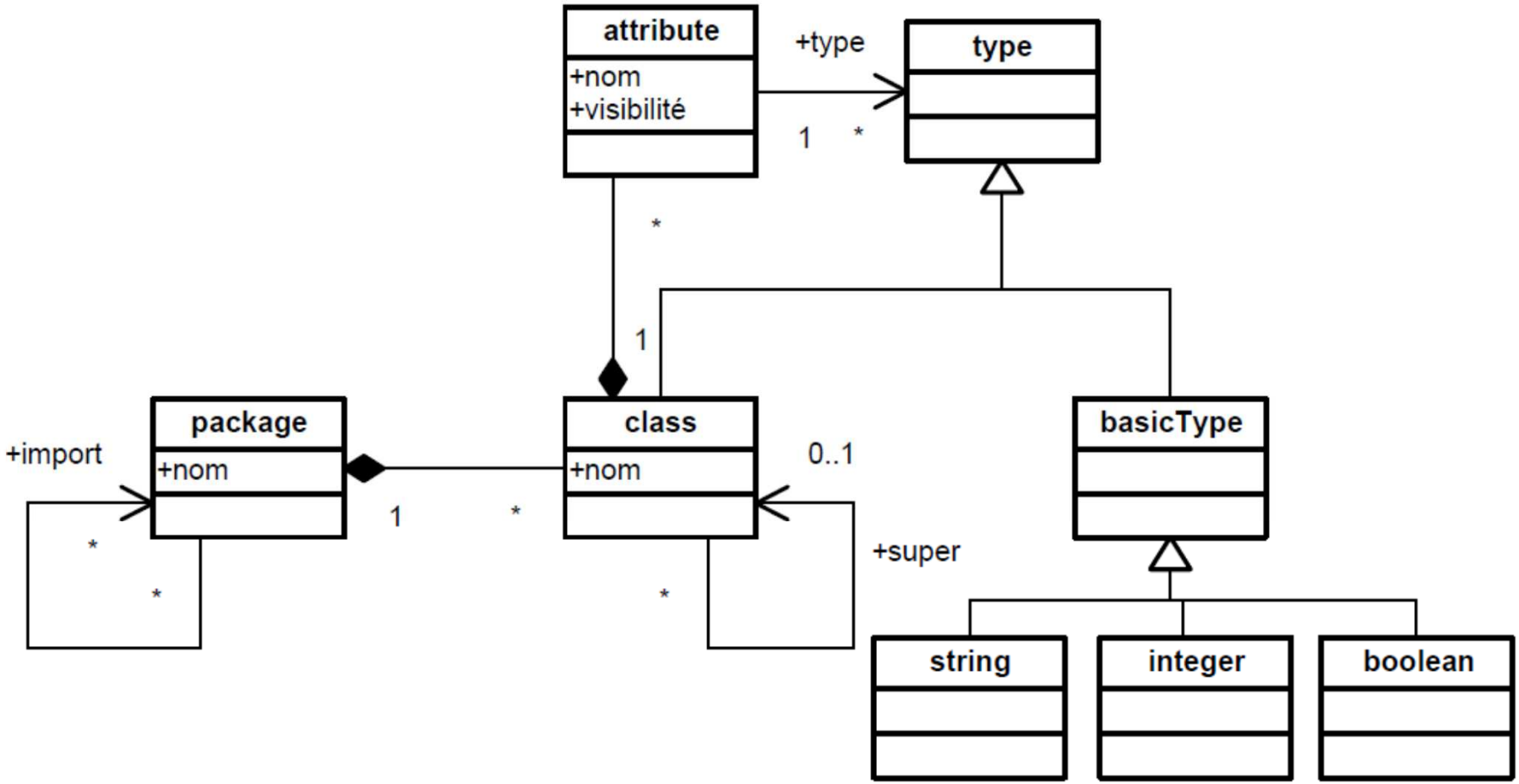
une partie de la spécification de l'OMG



Exercice 2 (1/3)
Métamodèle de diagramme de classes

- L'information qui nous intéresse est la suivante :
  - Un diagramme de classes contient des **packages.**
  - Un package a un nom et contient des **classes.**
  - Un package peut *importer* un autre package.
  - Une classe a un nom et peut *contenir* des **attributs et des méthodes.**
  - Une méthode peut avoir des paramètres et elle a un type de retour.
  - 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.

Exercice 2 (2/3)
Proposition du MM



Exercice 2 (3/3)

- Télécharger la spécification officielle « UML superStructure » à partir du site de l'OMG
- Vérifier les relations entre les Méta-classes : Classes, Property, Opération
- Étudier les contraintes appliquées sur ces méta-classes

Exercice 3

- Télécharger la spécification officielle « UML superStructure » à partir du site de l'OMG
- Référez vous à la section traitant les machines à états
- Décortiquez le méta-modèle traité dans cette section, et expliquez la signification des éléments suivants :
  - Region,
  - Vertex,
  - PseudoState,
  - Guard,
  - Trigger

TD

- Développez les contraintes suivantes :
  - Une classe doit posséder au moins un attribut
  - Si une classe possède un attribut privé, elle doit posséder également la méthode « get » de cet attribut. Cette méthode get doit etre public

Chapitre III : Méta-Modélisation

1. **Notions de Modèle ; Méta-modèle ; Méta-méta-modèle**
2. **Extension de UML / Profils UML**

UML et Profil

- UML permet de modéliser les applications Orientées Objet
  - Les modèles sont souvent utilisés pour Réfléchir, Définir la structure gros grain, Documenter
  - Ils ne permettent aucun gain significatif
  - Par nature, un modèle UML ne peut pas être productif
    - Indépendance des langages, sémantique trop générale
  - Il faut donc spécialiser UML pour être productif
    - UML pour CORBA, UML pour EJB, UML pour RT, …

UML et Profil

- UML permet-il de modéliser les applications non-Objet ?
- Il serait intéressant de disposer d'autres langages spécifiques à certains domaines tout en ayant la même productivité que celle des modèles UML
  - Langage pour modéliser les échanges entre applications
  - Langage pour modéliser les processus
  - Langage pour modéliser les éxigences
  - Langage pour modéliser les flux bancaires
  - …
- Grâce aux profils, il est possible de modéliser autre chose que des applications orientées objet (Ex: SQL)
- => Un modèle profilé est toujours un modèle UML

OMG et Profil

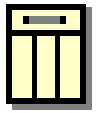
- Pour permettre l'adaptation d'UML à d'autres domaines et pour préciser la signification de cette adaptation, l'OMG a standardisé le concept de *profil UML.*
- Un profil est un ensemble de techniques et de mécanismes permettant d'adapter UML à un domaine particulier.
- Cette adaptation est dynamique, c'est-à-dire qu'elle ne modifie en rien le métamodèle UML et qu'elle peut se faire sur n'importe quel modèle UML.
- Les profils UML mettent en jeu le concept central de *stéréotype.*
- Un stéréotype est une sorte d'étiquette nommée que l'on peut coller sur n'importe quel élément d'un modèle UML. Lorsqu'un stéréotype est collé sur un élément d'un modèle, le nom du stéréotype définit la nouvelle signification de l'élément.

Profil UML

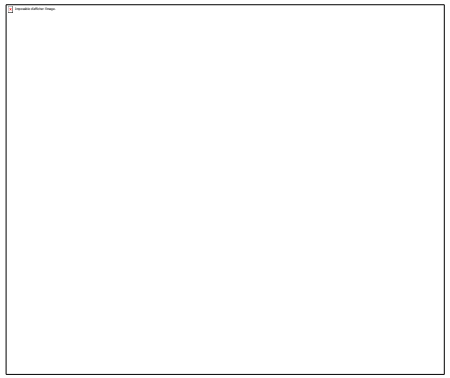
- LES STÉRÉOTYPES :
  - Ajout de nouveaux éléments de modélisation dans le contexte métier ou technique
  - Exemples : « interface », « entity_bean », …
  - Il est possible de stéréotyper tout concept UML (Classe, Attribut, Association, Use Case)
- LES TAGGED VALUES :
  - Annotation des éléments de modélisation
  - Exemples : {virtual}, {primary key}, …
  - Il est possible d'associer des tagged values à tout concept UML (Classe, Attribut, Association, Use Case)
- LES CONTRAINTES :
  - Préciser les conditions d'emploi des éléments du modèle
  - Il est possible d'associer des contraintes à tout concept UML (Classe, Attribut, Association, Use Case)

Exemple de Profil

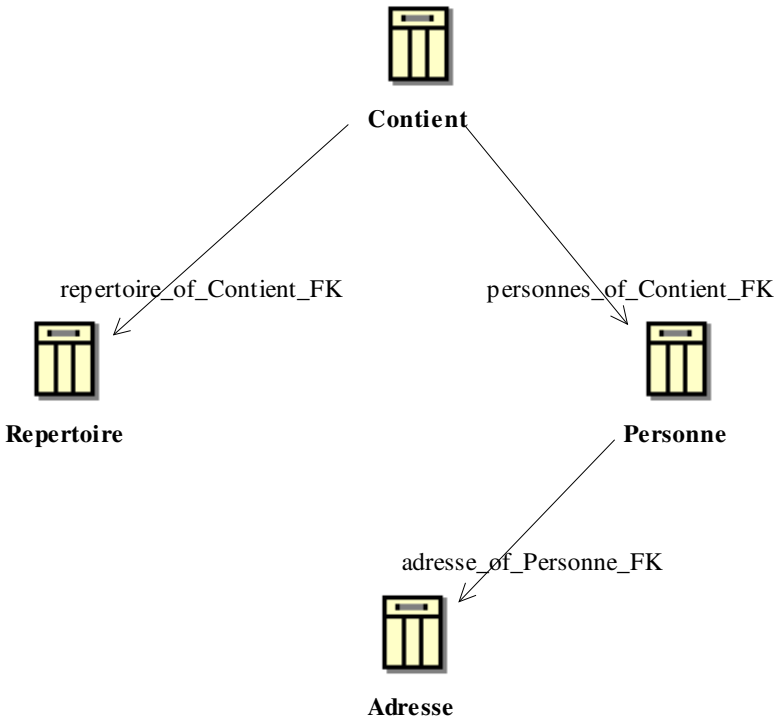
- SQL
  - Stereotype
    - Table, ForeignKey, PrimayKey
  - Contrainte
    - Une table ne peut avoir deux PrimaryKey, …
- EJB
  - Stereotype
    - EJBBean, EJBHomeInterface, …
  - Tagged-Value
    - Type de l'EJBBean (entity, session)
  - Contrainte
    - Un EJBBean doit avoir une EJBHomeInterface
- Table RDB

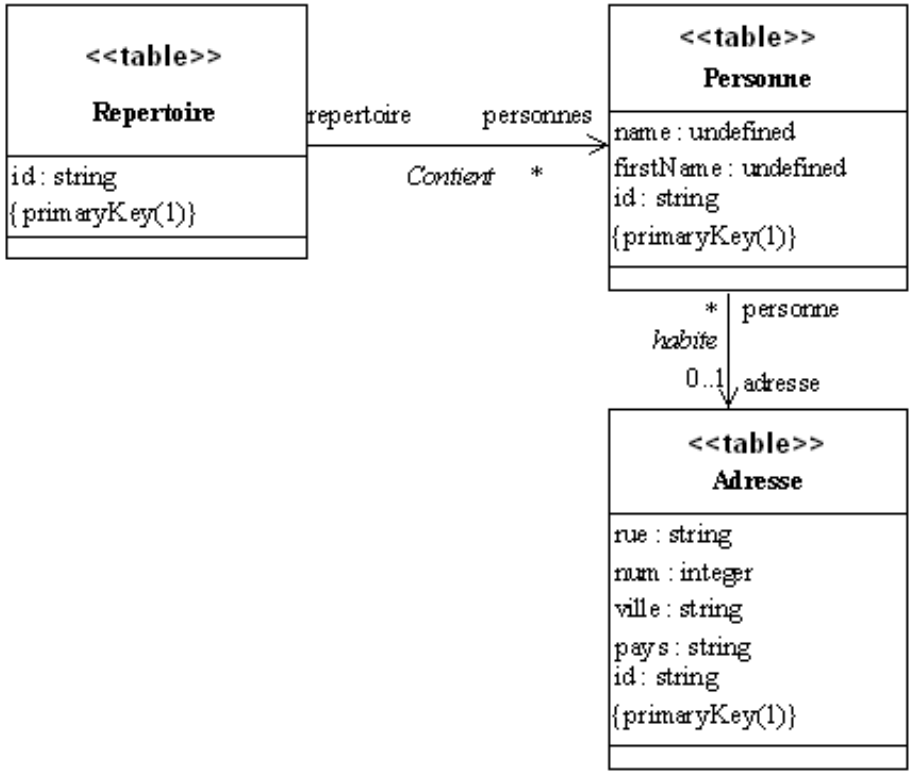


Repertoire



Transformation vers Profil





Mise en œuvre : Papyrus

- Papyrus (www.papyrusuml.org)
  - Éditeur UML open source
  - Relativement complet et aligné sur les spécifications officielles
  - Permet le développement de profil perso

- Papyrus permet
  - Création d'un projet... choix model ou profil...
  - Lors de la modification du profil, il faut le reappliquer sur le modèle
  - L'onglet propriété est très important !! Il contient lui-même plusieurs volet dont un nommé “profile”
  - Attention à la différence “delete from diagram” et “delete from model”
  - Faire des copies de sauvegarde régulièrement

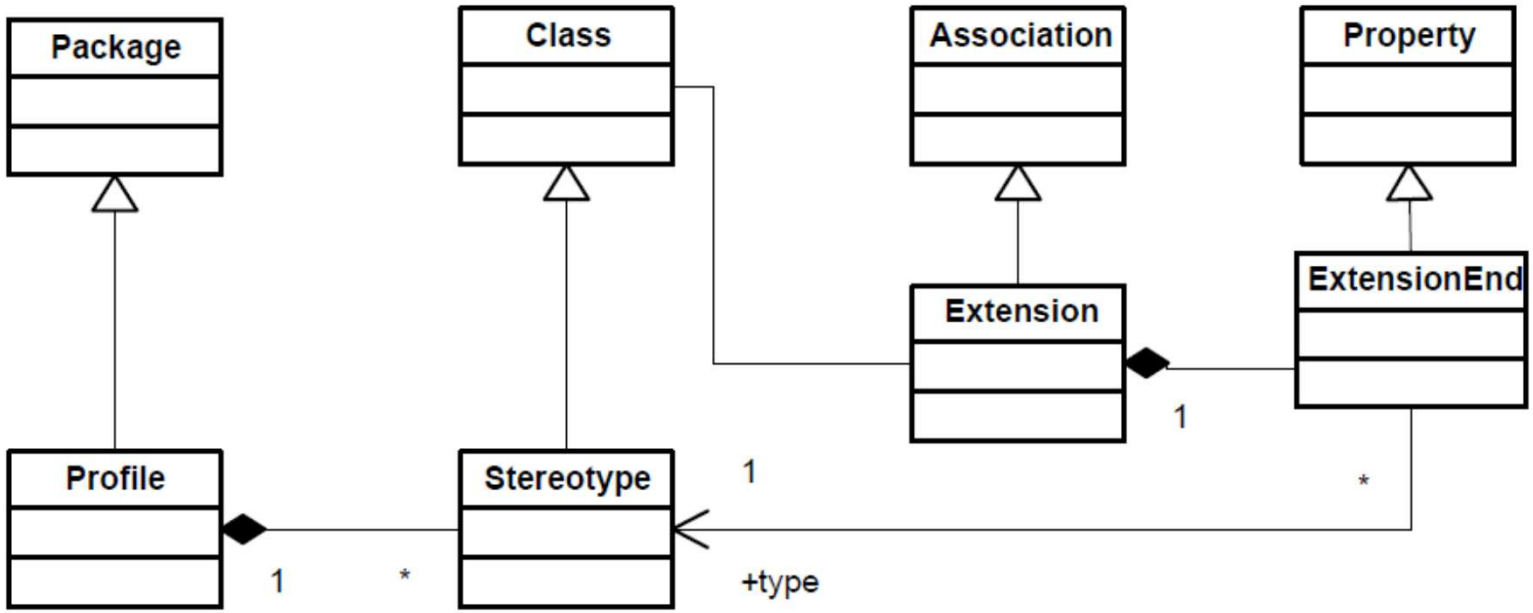
TP 1 : Se familiariser avec la création et l'utilisation de profil

- On souhaite développer un profil pour représenter quelques concepts de manipulation des bases de données relationnelles, tels que :
  - La notion de table,
  - La notion de clé primaire,
  - La notion de clés étrangère
  - La relation entre les tables

- A faire :
  - Créer un profil appelé BDR_Profil,
  - Créer un stéréotype pour chaque concept,
  - Sauvegarder et définir votre profil
  - Appliquer le profil à une application de test.

Exercice 2 (1/2)
MétaModèle des profils dans UML

- Développer un Métamodèle pour représenter la notion de profil dans UML



Exercice 2 (2/2)
MétaModèle des profils dans UML

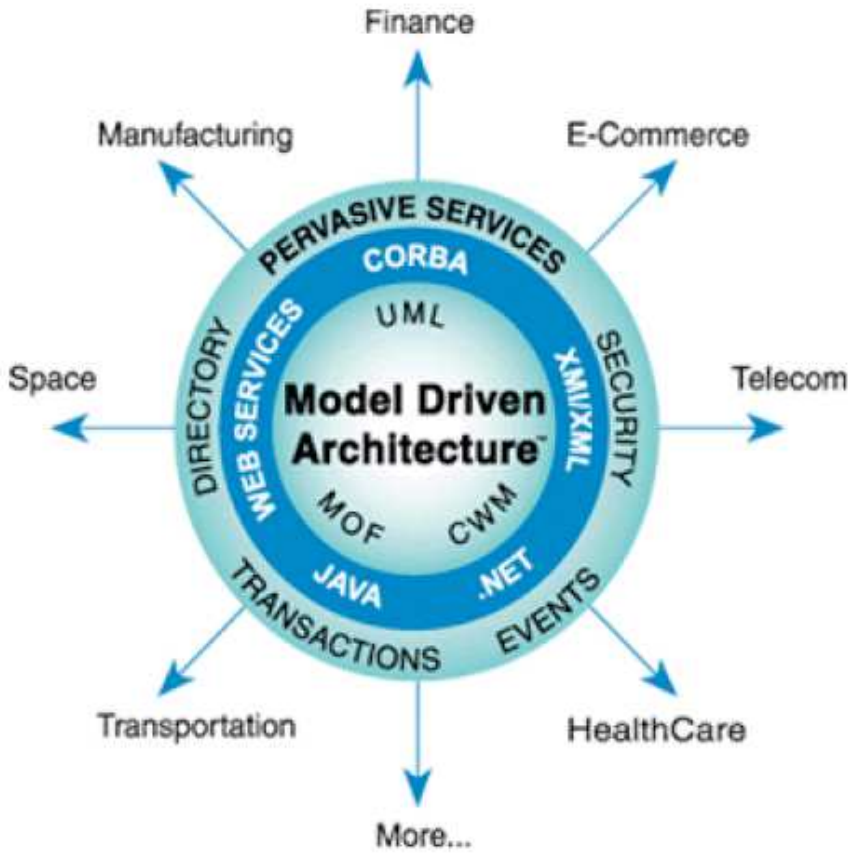
- la métaclasse Profile
  - hérite de la métaclasse Package.
  - Cela signifie que les définitions de nouveaux profils sont maintenant considérées comme étant des modèles UML et plus précisément comme des packages.
- La métaclasse Stereotype
  - hérite de son côté de la métaclasse Class.
  - Cela signifie que les définitions de stéréotypes sont considérées comme étant des classes UML. Le nom de la classe correspond au nom du stéréotype.
- la métaclasse Extension,
  - hérite de la métaclasse Association,
- métaclasse ExtensionEnd,
  - hérite de la métaclasse Property.
  - Cela signifie qu'un stéréotype apporte une extension de signification à une classe.

Chapitre IV

MDA & Transformation de modèles

Architecture MDA

- **L'architecture MDA de l'OMG repose sur 4 couches :**
  - **La couche modélisation**
    - UML : pour décrire le modèle
    - MOF : spécifie la structure et la syntaxe de la description des modèles
    - CWM : standard permettant d'échanger des méta-données
  - **La couche technologies**
    - XMI : pour décrire en XML les modèles
    - Les standards technologiques actuels (Java, .Net, …) pour l'implémentation
  - **La couche Services**
  - **La couche métier**

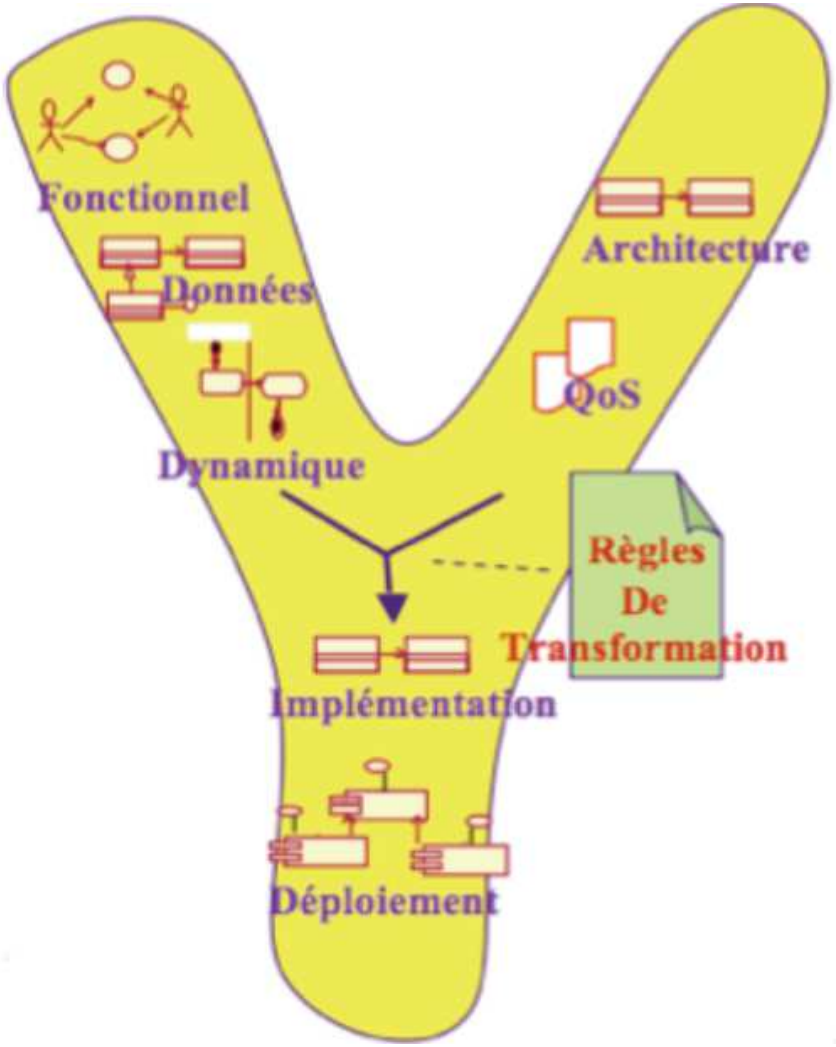


MDA : objectifs

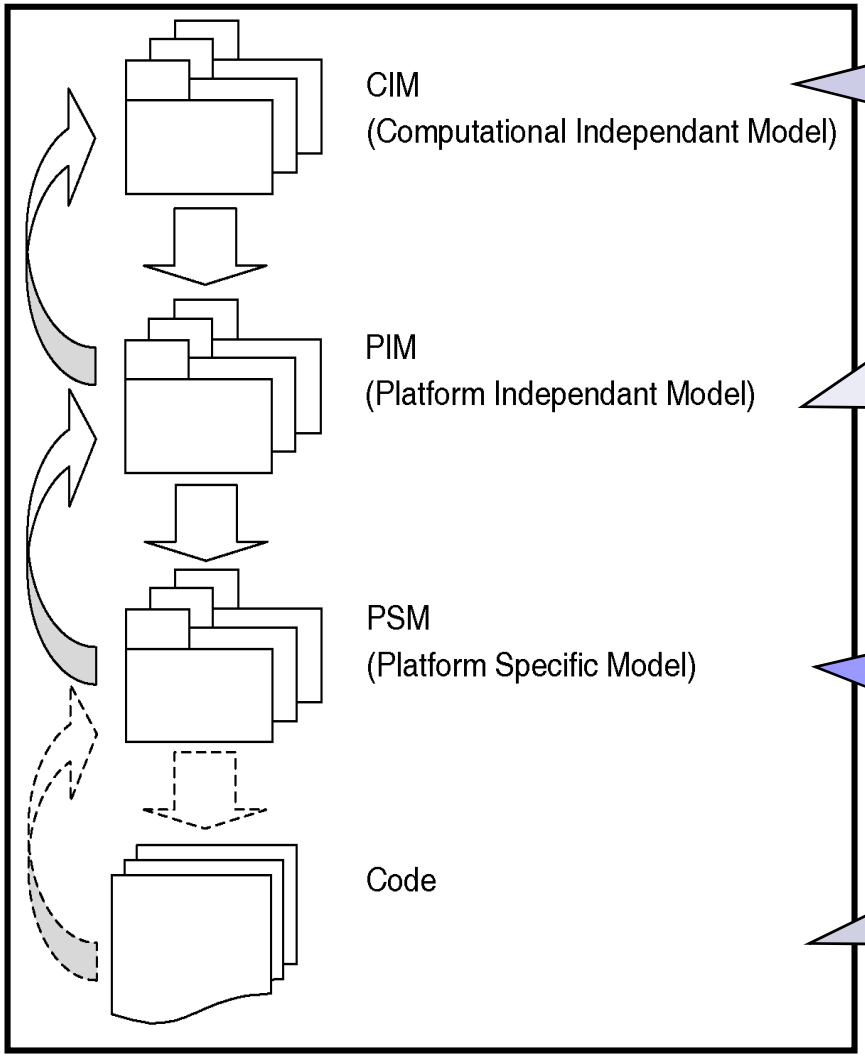
- Pérennité des savoir-faire
  - L'ambition du MDA est de faire en sorte que les modèles aient une durée de vie supérieure au code.
  - L'objectif est donc de fournir des langages de modélisation supportant différents niveaux d'abstraction.
- Gains de productivité
  - MDA vise à apporter des gains de productivité.
  - L'objectif est donc de faciliter la création d'opérations de production sur les modèles
- Prise en compte des plates-formes d'exécution
  - MDA veut rendre explicite la prise en compte des plates-formes d'exécution dans le cycle de vie des applications.
  - L'objectif est donc de construire des langages permettant de modéliser les plates-formes et de lier ces modèles aux modèles des applications.

Un processus dirigé par les modèles

Automatiser au mieux le processus de production vers le code



MDA : Principe



Modèle d'exigences
- Représente l'application dans son environnement.
- Aucune considération informatique n'apparait

Modèle d'analyse/conception abstraite
- Rattaché à un paradigme informatique
- Indépendant d'une plateforme de réalisation précise
- Représente l'architecture de l'application.

Modèles dépendants des plateformes
- Version modélisée du code
- Représente la construction de l'application.

Code de l'application et fichiers de configuration.

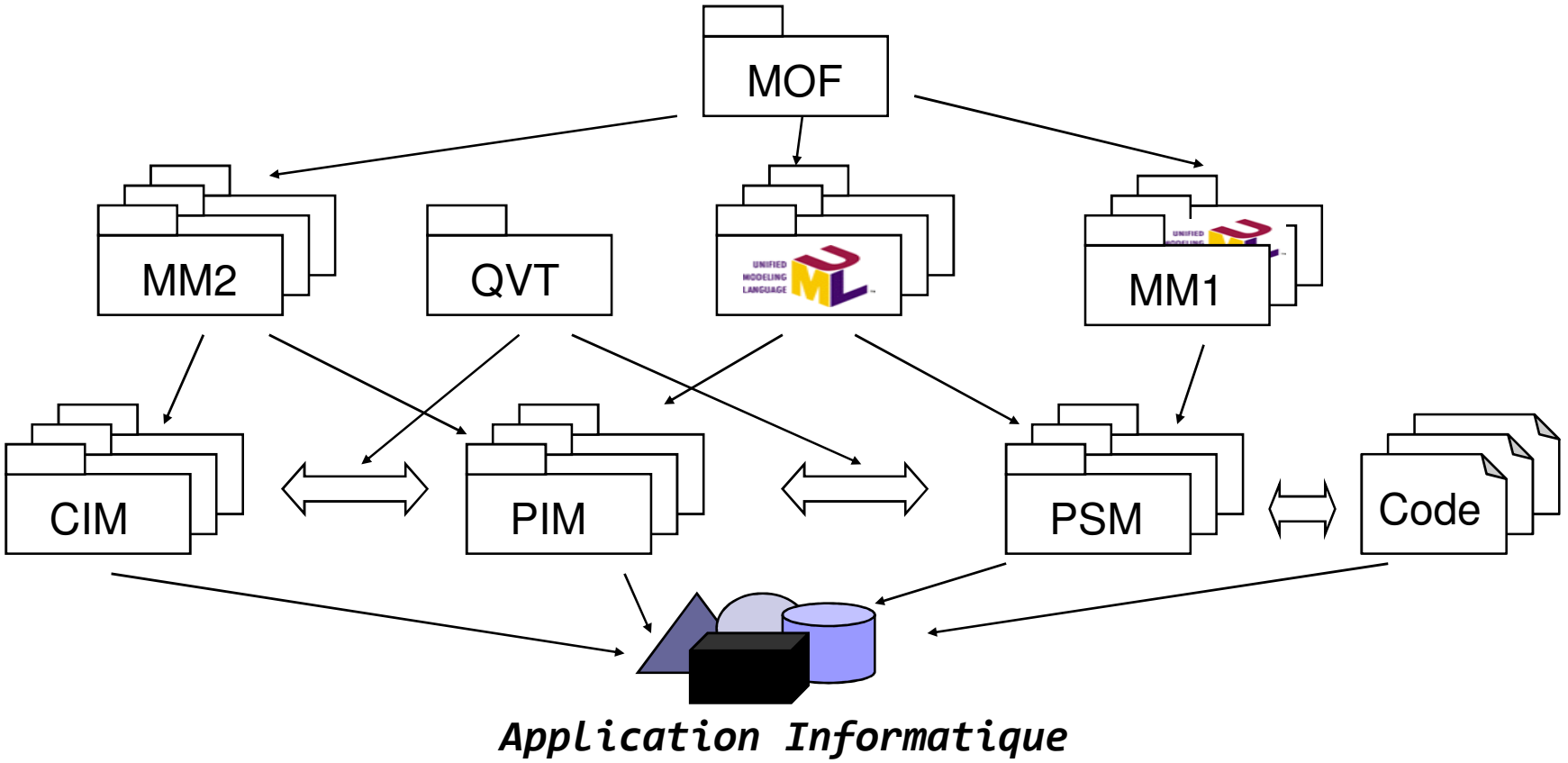
MDA : Réalisation d'une application

- Réalisation d'une application est un processus basé sur une série de transformations de modèles

- Exemple
  - Modèle de l'application au niveau abstrait, avec un modèle de composant abstrait (Composant UML2) : modèle PIM
  - Projection du modèle vers un modèle de composant EJB : modèle PSM
  - Raffinement de ce modèle pour ajouter des détails d'implémentation : toujours un modèle PSM
  - Génération du code de l'application vers la plateforme EJB

MDA : Ambition

Objectif : Tenter une interopérabilité par les modèles



Exemple de Transformation

- PIM → PSM
  - motif de passage d'une classe UML à une classe Java
  - prise en compte de l'héritage multiple (C++, C# , Java)
  - prise en compte des technologies disponibles

- CONCLUSION :
  - L'Ingénierie Dirigée par les Modèles repose sur la méta-modélisation ET sur les transformations.

Transformation de modèles

Transformation de modèles

- Un modèle de transformation vise à transformer des modèles source en modèles cible.

- Définition d'une Transformation
  - Ensemble de règles
  - Correspondances entre éléments du modèle source et éléments du modèle cible

- Les transformations de modèles sont le cœur des aspects de production de MDA
  - CIM vers PIM, PIM vers PSM, PSM vers code.

- Les grandes familles de langages de transformation (impératif, déclaratif et hybride)

Transformation de modèles - Principe

- 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.

- Les modèles source et destination **sont conformes à leur méta-modèle** respectif et le passage de l'un à l'autre est décrit par des règles de transformation.

- Les règles sont exécutées sur les modèles source afin de générer les modèles destination.

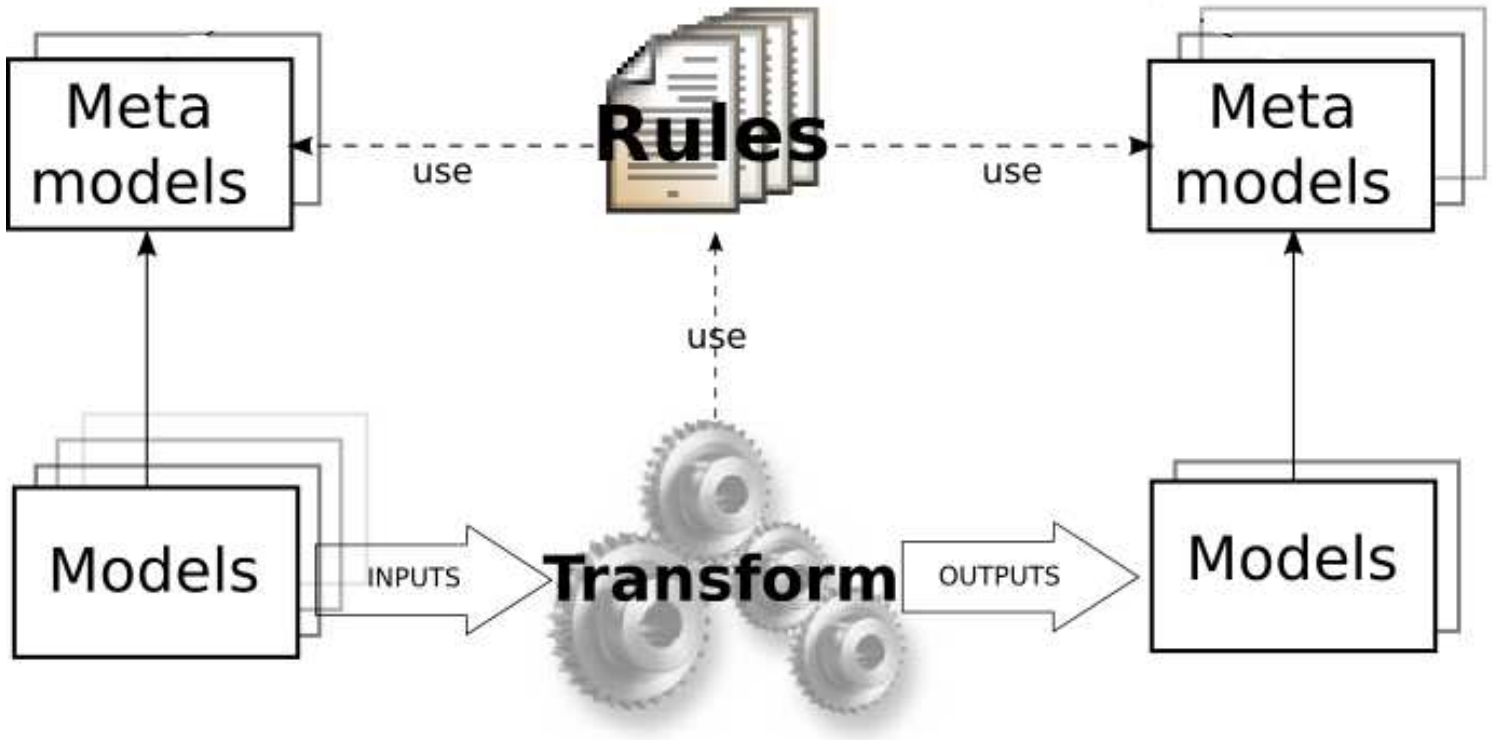
Moyens et standards

- Langages de (méta) modélisation
  - MOF, UML, OCL, XMI (XML Metadata Interchange)
- Un langage de transformation unifié
  - MOF 2.0 QVT
- Des prototypes issus de la recherche
  - Kermeta (INRIA, Rennes) : Impératif
  - ATL (INRIA, Nantes) : semi-Déclaratif
  - OpenQVT (FT R&D)
- Eclipse : Java + EMF
  - Eclipse Modeling Framework

Quels langages de transfo de modèles ?

- Purement déclaratifs (ex : Tefkat)
  - Un ensemble de règles faisant correspondre un motif du modèle source à un motif du modèle cible
  - Un moteur d'application des règles, non géré par le programmeur de transformation de modèles
- Semi déclaratifs (ex : ATL)
  - Un ensemble de règles, avec un motif du modèle source et des instructions impératives à effectuer sur le modèle cible quand le motif source est détecté
- Impératifs (ex : Kermeta, Java EMF)
  - Pas de notion de règle

Transformation de modèles - Principe



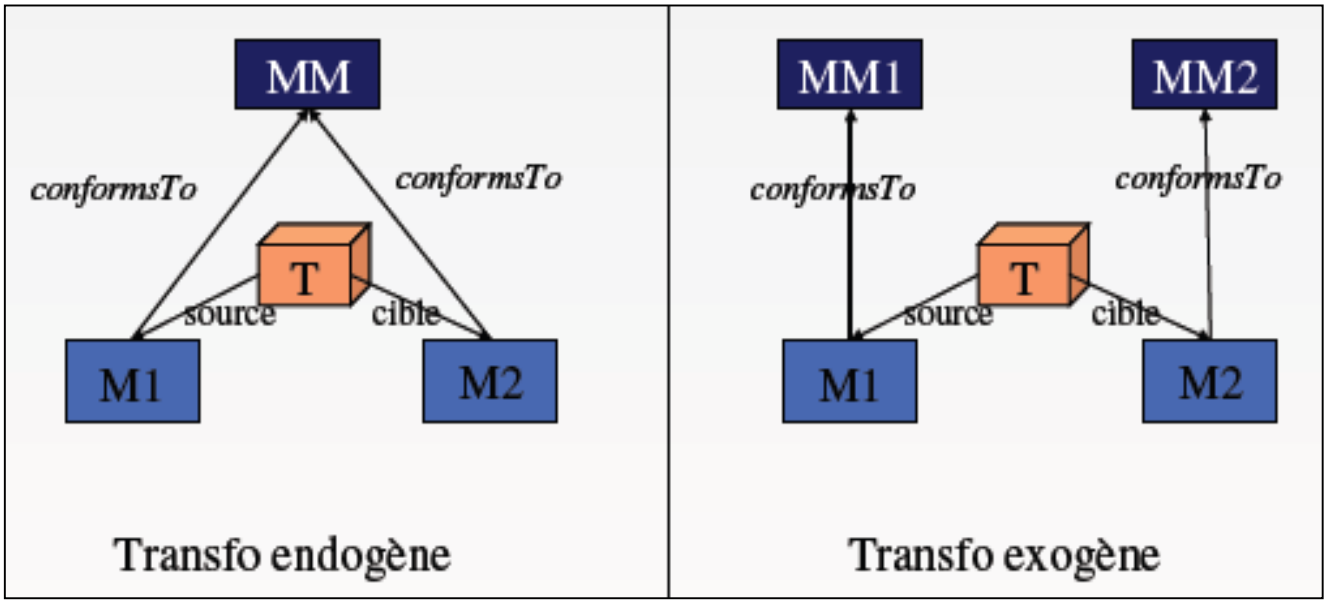
Exécution d'une règles de transformation

- Définition
  - Une règle de transformation définit la manière dont un ensemble de concepts du méta-modèle source est transformé en un ensemble de concepts du méta-modèle destination.

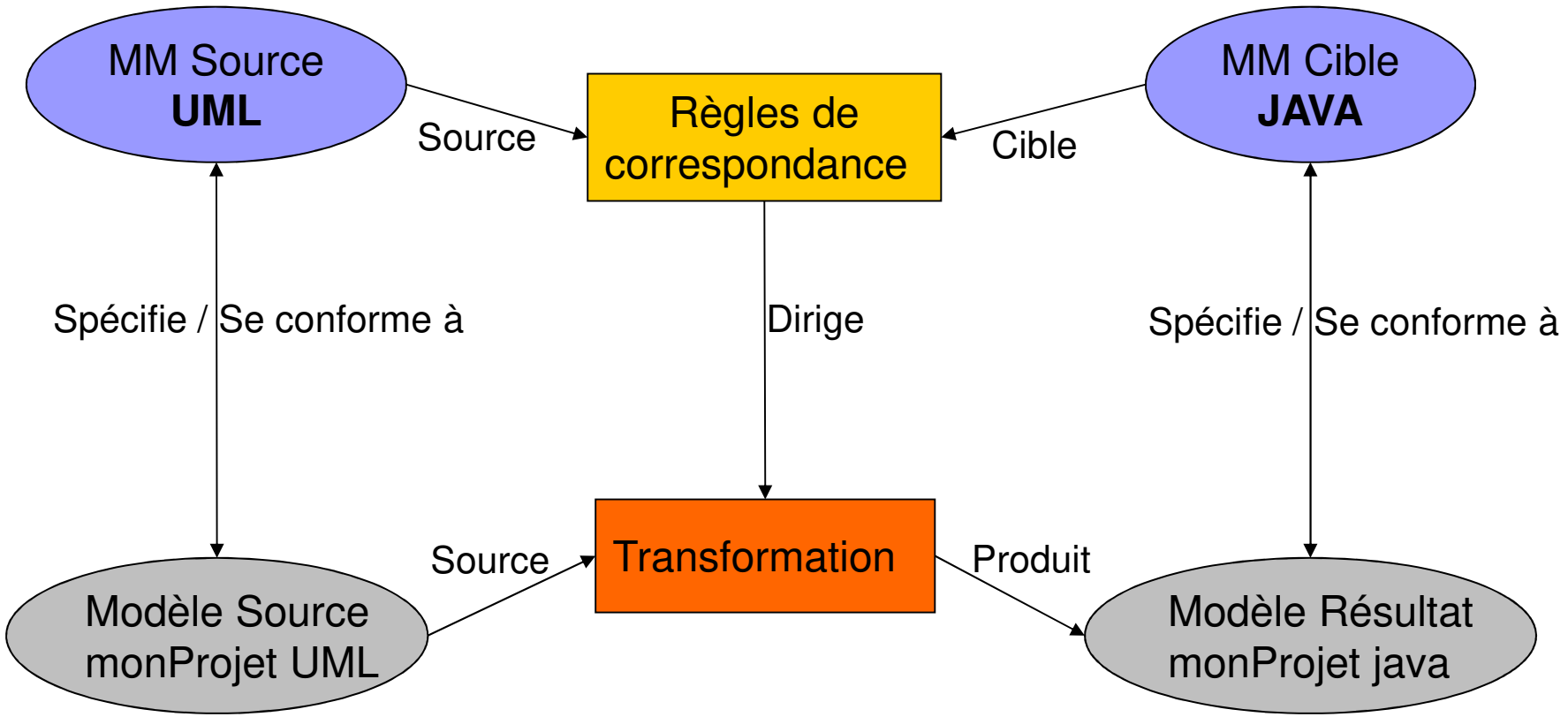
- Trois étapes :
  - **La *vérification* de la condition :** Analyse du modèle source de manière à détecter la présence d'un ensemble de concepts qui correspond à ce que la règle « attend » en entrée.
  - ***L'exécution* de la règle :** Elle a lieu pour chaque ensemble de concepts qui valide la condition.
  - **La *création* :** Génération d'un ensemble de concepts dans le modèle de sortie dont les champs sont remplis par les variables affectées durant l'exécution.

Transformations endogènes et exogènes

- Transformations endogènes,
  - Dans le même espace technologique
  - Les modèles sources et cible sont conformes au même meta-modèle
- Transformations exogènes
  - Entre 2 espaces technologiques différents
  - Les modèles source et cible sont conformes a des meta-modèles différents

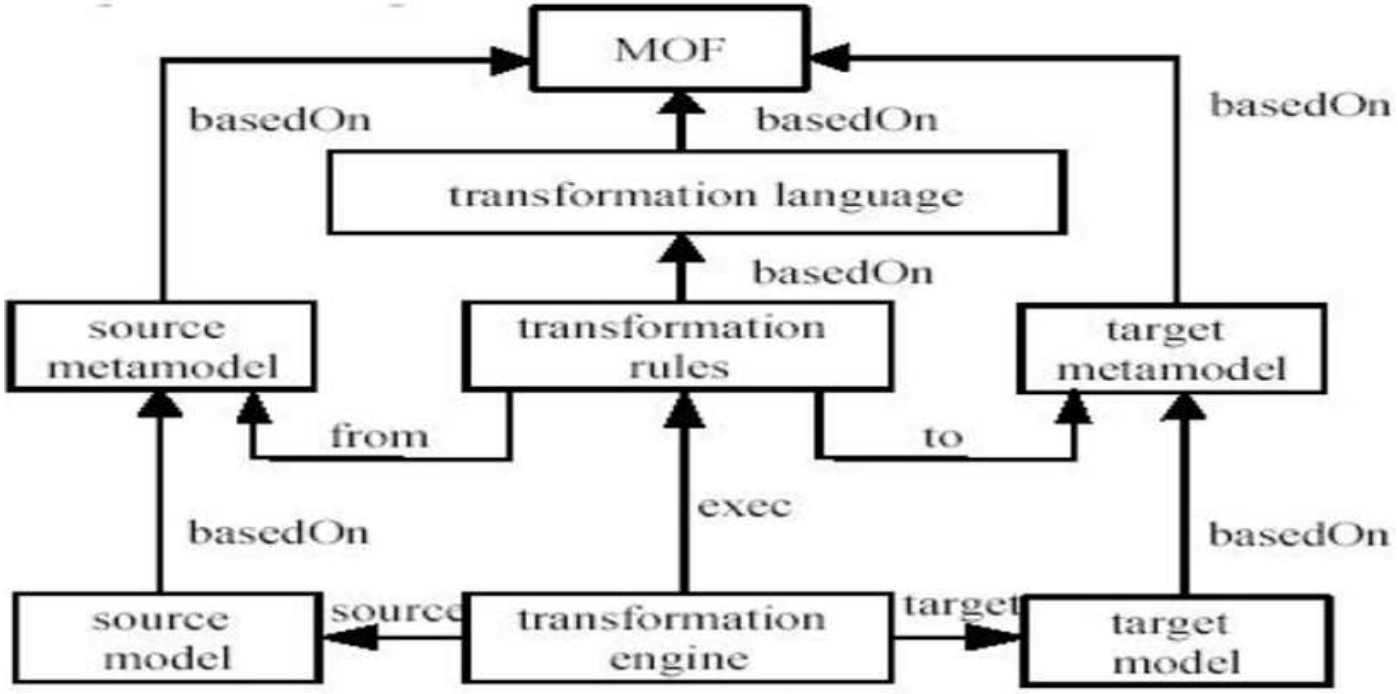


Transformation de modèles - Exemple



ATL
(ATLAS Transformation Language)

- Langage de transformation de modèles développé par l'INRIA
- Disponible en logiciel libre sous le projets Eclipse GMT
- Un des langages les plus proches de la norme QVT de l'OMG0



Les transformations de modèles - ATL

ATL
(ATLAS Transformation Language)

- Langage hybride (déclaratif et impératif)
- Pour accéder aux éléments d'un modèle , ATL utilise des requêtes sous formes d'expressions OCL.
- La définition d'un modèle de transformation se fait dans un fichier portant l'extension «.atl ». Ce fichier est composé de trois parties :
  - L'entête (Header)
  - Les fonctions (Helpers)
  - Les règles de transformation (Rules)

ATL : Structure d'une transformation
Header

- Le Header contient :
  - le nom de la transformation
  - le méta-modèle d'entrée
  - le méta-modèle de sortie
  - les librairies importées

- L'entête du fichier se déclare de la manière suivante :

Module <nom du module> ;
create OUT : <méta-modèle cible> **from IN** : <méta-modèle source> ;
Uses <librairies>;

ATL : Structure d'une transformation
Rules

- Décrivent la transformation d'un élément du méta-modèle source en un élément du méta-modèle cible.
- Contiennent des expressions OCL.
- Se décomposent en 2 parties :
  - from : indique les éléments du modèle source à transformer
  - to : contient les expressions de transformation
- Syntaxe :

```
rule <nom de la règle> {
from
<pattern source>
to
<un ou plusieurs patterns cibles>
}
```

ATL : Structure d'une transformation
Helper

- Eviter la redondance de code
- Définir des variables globales
- Les helpers ATL peuvent éventuellement prendre des paramètres et ont un type de retour ; définies à l'aide d'expressions OCL
- Syntaxe :

Helper def : Nom(<paramètre> : <type>) **:** <type retour> = <expressions OCL> ;

Génération de code

- Requete (Query) :
  - Transformation modèle vers texte
  - Une requête (**query**) est une transformation d'un modèle vers un type primitif
  - Comme un module, une requête peut définir des helpers et des attributs

Langage de requête ATL

- Les caracteres sont numerotes de 1 a size()
- Operations importantes :
  - **concat**(s : String) : ou +
  - **substring**(lower : Integer, upper : Integer)
  - **toInteger**() ,**toReal**()
  - **toUpper**() , **toLower**()
  - **toSequence**() : (of char)
  - **startsWith**(s : String) :
  - **indexOf**(s : String) :
  - **lastIndexOf**(s : String) :
  - **split** (regexp : String) :
  - **replaceAll**(c1 : String, c2 : String) :
  - **writeTo**(fileName : String) :
  - **println**() : ecrire la chaine dans la console d'Eclipse