

# ***Ingénierie des Modèles***

## ***Méta-modélisation***

Eric Cariou

Master Technologies de l'Internet 2<sup>ème</sup> année

*Université de Pau et des Pays de l'Adour  
UFR Sciences Pau – Département Informatique*

Eric.Cariou@univ-pau.fr

# *Introduction/Plan*

- ◆ But de la méta-modélisation
  - ◆ Définir des langages de modélisation ou des langages de manière générale
- ◆ Architecture MOF de l'OMG
  - ◆ 4 niveaux de (méta)modélisation
- ◆ Syntaxes abstraite et concrète
- ◆ Définition de méta-modèles
  - ◆ Profils UML
  - ◆ MOF
  - ◆ Ecore

# *Principales normes modélisation OMG*

- ◆ MOF : Meta-Object Facilities
  - ◆ Langage de définition de méta-modèles
- ◆ UML : Unified Modelling Language
  - ◆ Langage de modélisation
- ◆ CWM : Common Warehouse Metamodel
  - ◆ Modélisation ressources, données, gestion d'une entreprise
- ◆ OCL : Object Constraint Language
  - ◆ Langage de contraintes sur modèles
- ◆ XMI : XML Metadata Interchange
  - ◆ Standard pour échanges de modèles et méta-modèles entre outils

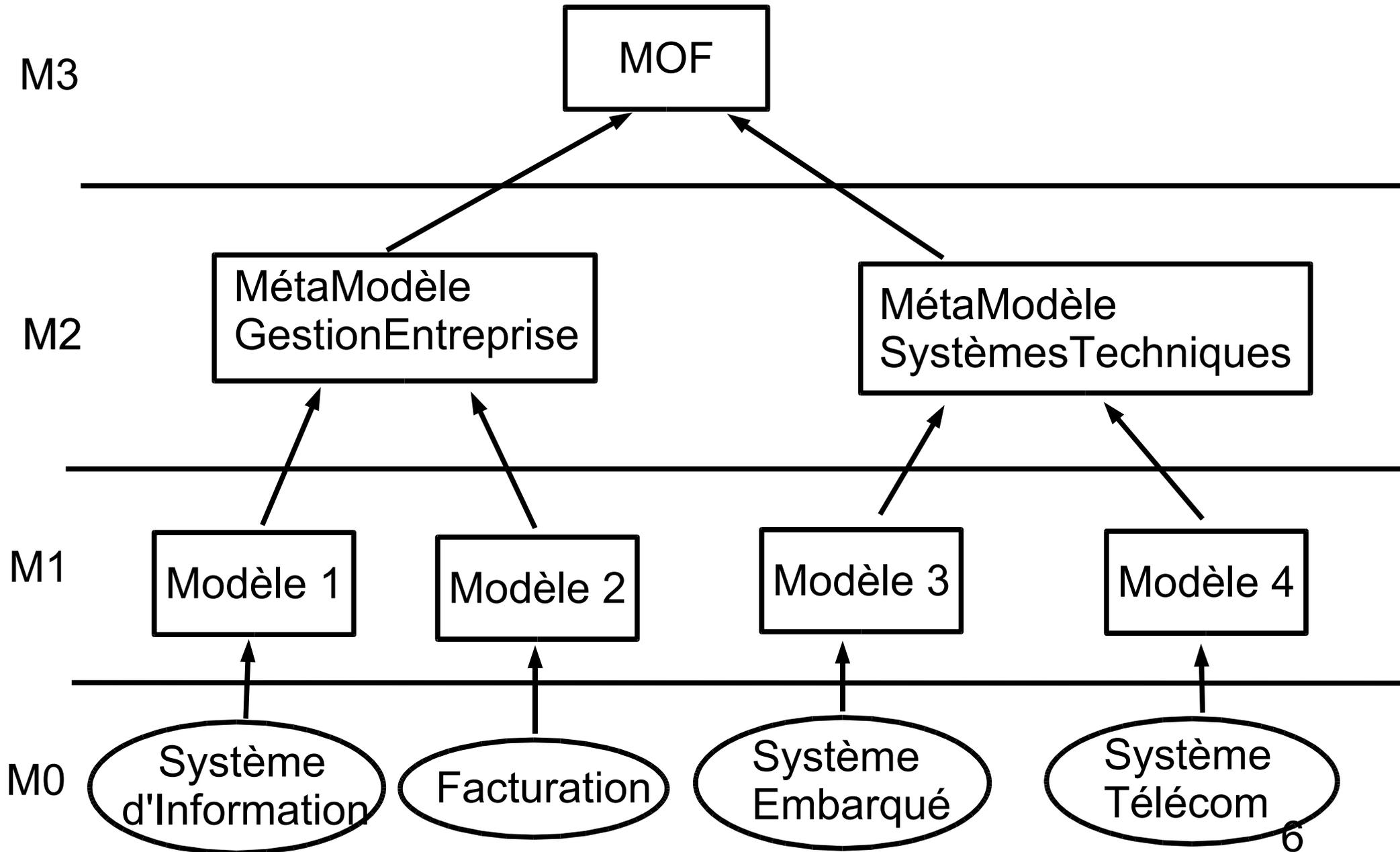
# *Normes OMG de modélisation*

- ◆ Plusieurs de ces normes concernent la définition et l'utilisation de méta-modèles
  - ◆ MOF : but de la norme
  - ◆ UML et CWM : peuvent être utilisés pour en définir
  - ◆ XMI : pour échange de (méta-)modèles entre outils
- ◆ MOF
  - ◆ C'est un méta-méta-modèle
    - ◆ Utilisé pour définir des méta-modèles
  - ◆ Définit les concepts de base d'un méta-modèle
    - ◆ Entité/classe, relation/association, type de données, référence, package ...
  - ◆ Le MOF peut définir le MOF

# *Hiérarchie de modélisation à 4 niveaux*

- ◆ L'OMG définit 4 niveaux de modélisation
  - ◆ M0 : système réel, système modélisé
  - ◆ M1 : modèle du système réel défini dans un certain langage
  - ◆ M2 : méta-modèle définissant ce langage
  - ◆ M3 : méta-méta-modèle définissant le méta-modèle
    - ◆ Le niveau M3 est le MOF
    - ◆ Dernier niveau, il est méta-circulaire : il peut se définir lui même
- ◆ Le MOF est – pour l'OMG – le méta-méta-modèle unique servant de base à la définition de tous les méta-modèles

# Hiérarchie de modélisation à 4 niveaux



# *Hiérarchie de modélisation à 4 niveaux*

- ◆ Hiérarchie à 4 niveaux existe en dehors du MOF et d'UML, dans d'autres espaces technologiques que celui de l'OMG
  - ◆ Langage de programmation
    - ◆ M0 : l'exécution d'un programme
    - ◆ M1 : le programme
    - ◆ M2 : la grammaire du langage dans lequel est écrit le programme
    - ◆ M3 : le concept de grammaire EBNF
  - ◆ XML
    - ◆ M0 : données du système
    - ◆ M1 : données modélisées en XML
    - ◆ M2 : DTD / Schema XML
    - ◆ M3 : le langage XML

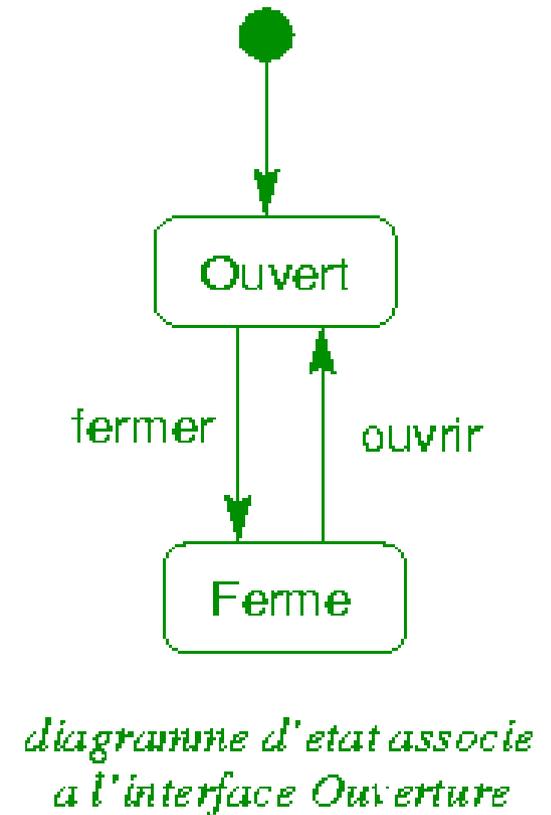
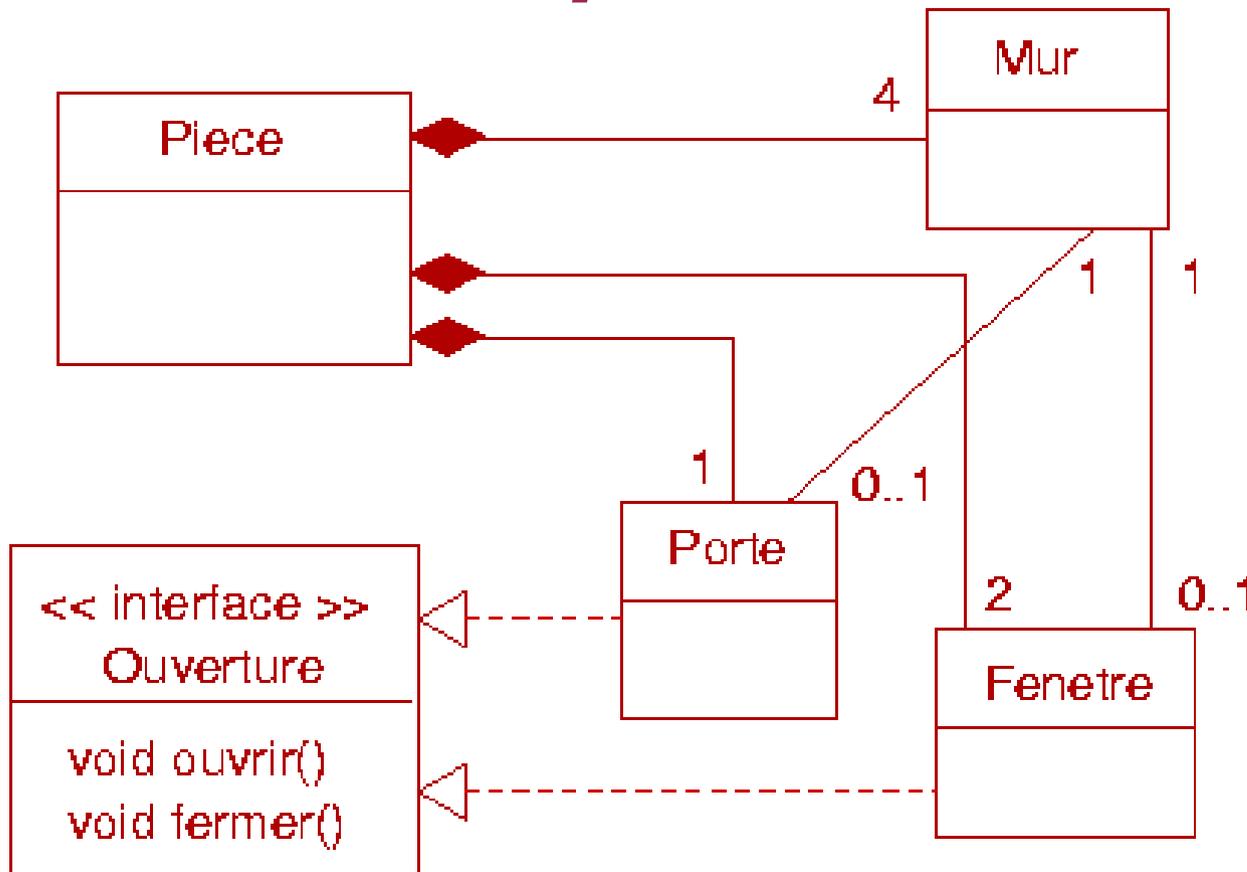
# *Méta-modélisation UML*

- ◆ Avec UML, on retrouve également les 4 niveaux
  - ◆ Mais avec le niveau M3 définissable en UML directement à la place du MOF
- ◆ Exemple de système réel à modéliser (niveau M0)
  - ◆ Une pièce possède 4 murs, 2 fenêtres et une porte
  - ◆ Un mur possède une porte ou une fenêtre mais pas les 2 à la fois
  - ◆ Deux actions sont associées à une porte ou une fenêtre : ouvrir et fermer
  - ◆ Si on ouvre une porte ou une fenêtre fermée, elle devient ouverte
  - ◆ Si on ferme une porte ou une fenêtre ouverte, elle devient fermée

# ***Méta-modélisation UML***

- ◆ Pour modéliser ce système, il faut définir 2 diagrammes UML : niveau M1
  - ◆ Un diagramme de classe pour représenter les relations entre les éléments (portes, murs, pièce)
  - ◆ Un diagramme d'état pour spécifier le comportement d'une porte ou d'une fenêtre (ouverte, fermée)
  - ◆ On peut abstraire le comportement des portes et des fenêtres en spécifiant les opérations d'ouverture fermeture dans une interface
    - ◆ Le diagramme d'état est associé à cette interface
  - ◆ Il faut également ajouter des contraintes OCL pour préciser les contraintes entre les éléments d'une pièce

# M1 : spécification du système

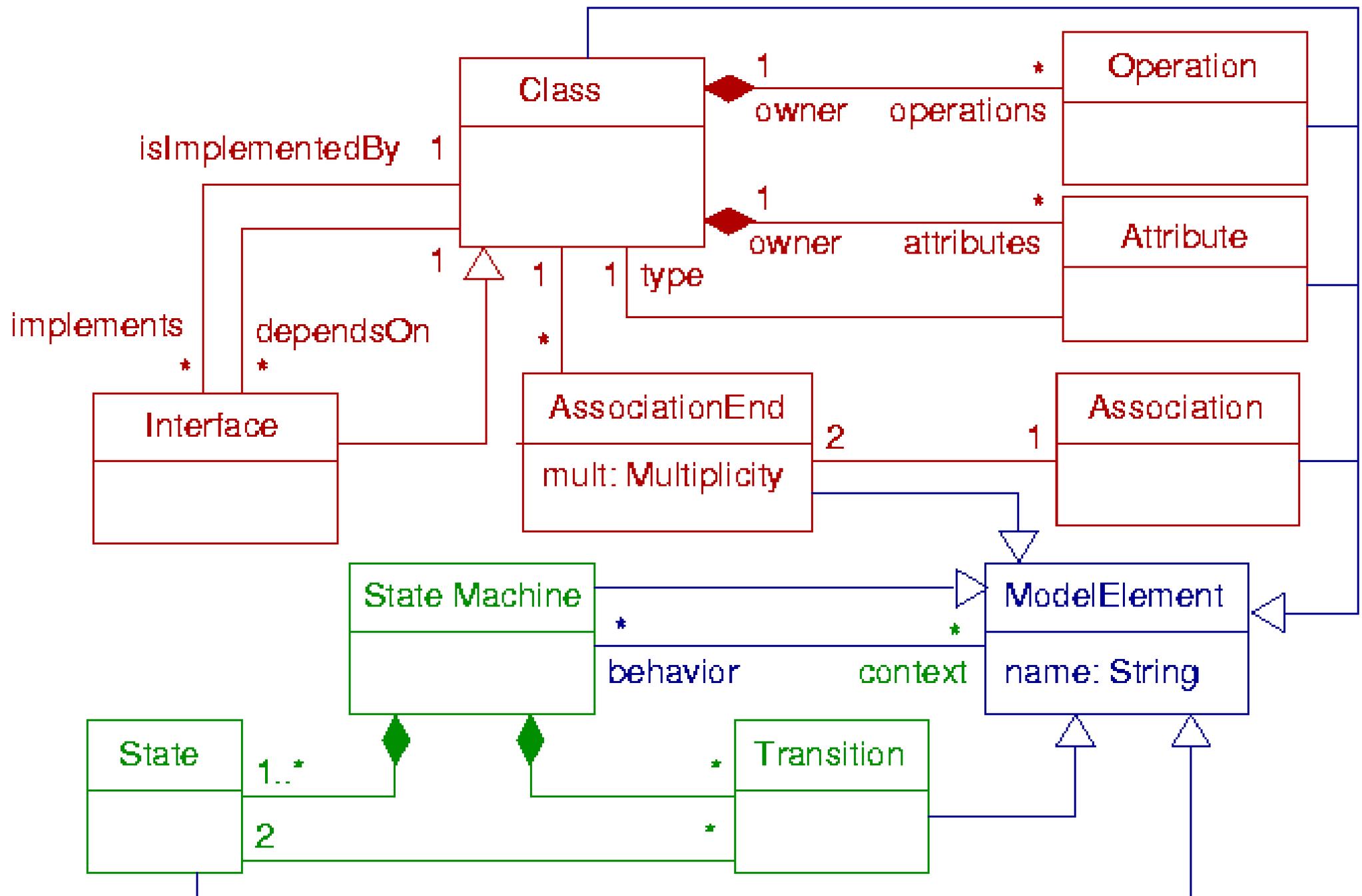


- ◆ **context** Mur inv: fenetre -> union(porte) -> size() <= 1  
-- un mur a soit une fenêtre soit une porte (soit rien)
- ◆ **context** Piece inv:  
mur.fenetre -> size() = 2 -- 2 murs de la pièce ont une fenêtre  
mur.porte -> size() = 1 -- 1 mur de la pièce a une porte

# *Méta-modélisation UML*

- ◆ Les 2 diagrammes de ce modèle de niveau M1 sont des diagrammes UML valides
- ◆ Les contraintes sur les éléments des diagrammes UML et leurs relations sont définies dans le méta-modèle UML : niveau M2
  - ◆ Un diagramme UML (de classes, d'états ...) doit être conforme au méta-modèle UML
- ◆ Méta-modèle UML
  - ◆ Diagramme de classe spécifiant la structure de tous types de diagrammes UML
    - ◆ Diagramme de classe car c'est le diagramme UML permettant de définir des éléments/concepts (via des classes) et leurs relations (via des associations)
  - ◆ Avec contraintes OCL pour spécification précise

# M2 : Méta-modèle UML (très simplifié)



# *M2 : Méta-modèle UML (très simplifié)*

## ◆ Contraintes OCL, quelques exemples

- ◆ **context** Interface **inv**: attributes -> isEmpty()

*Une interface est une classe sans attribut*

- ◆ **context** Class **inv**: attributes -> forAll ( a1, a2 | a1 <> a2 **implies** a1.name <> a2.name)

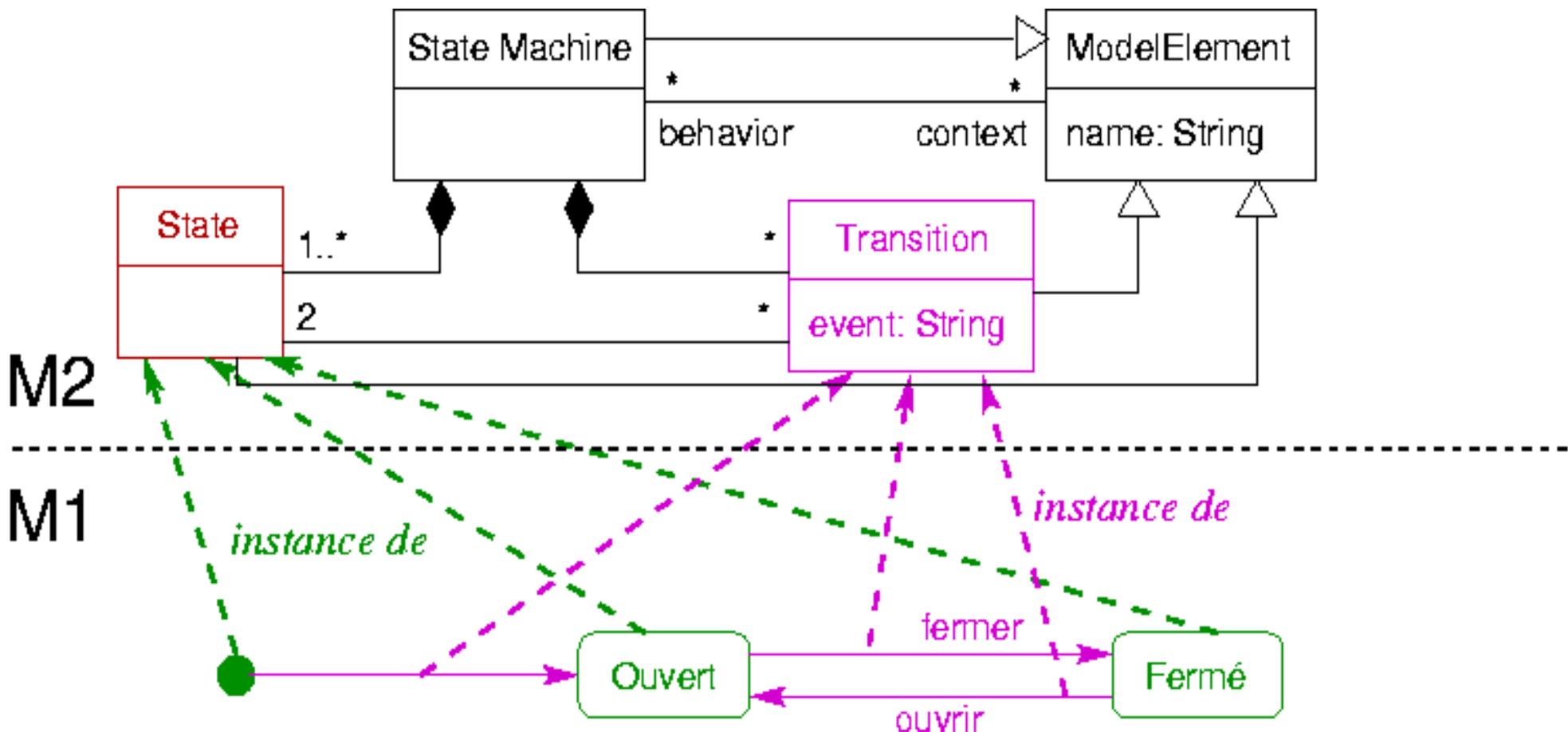
*2 attributs d'une même classe n'ont pas le même nom*

- ◆ **context** StateMachine **inv**: transition -> forAll ( t | self.state -> includesAll(t.state))

*Une transition d'un diagramme d'état connecte 2 états de ce diagramme d'état*

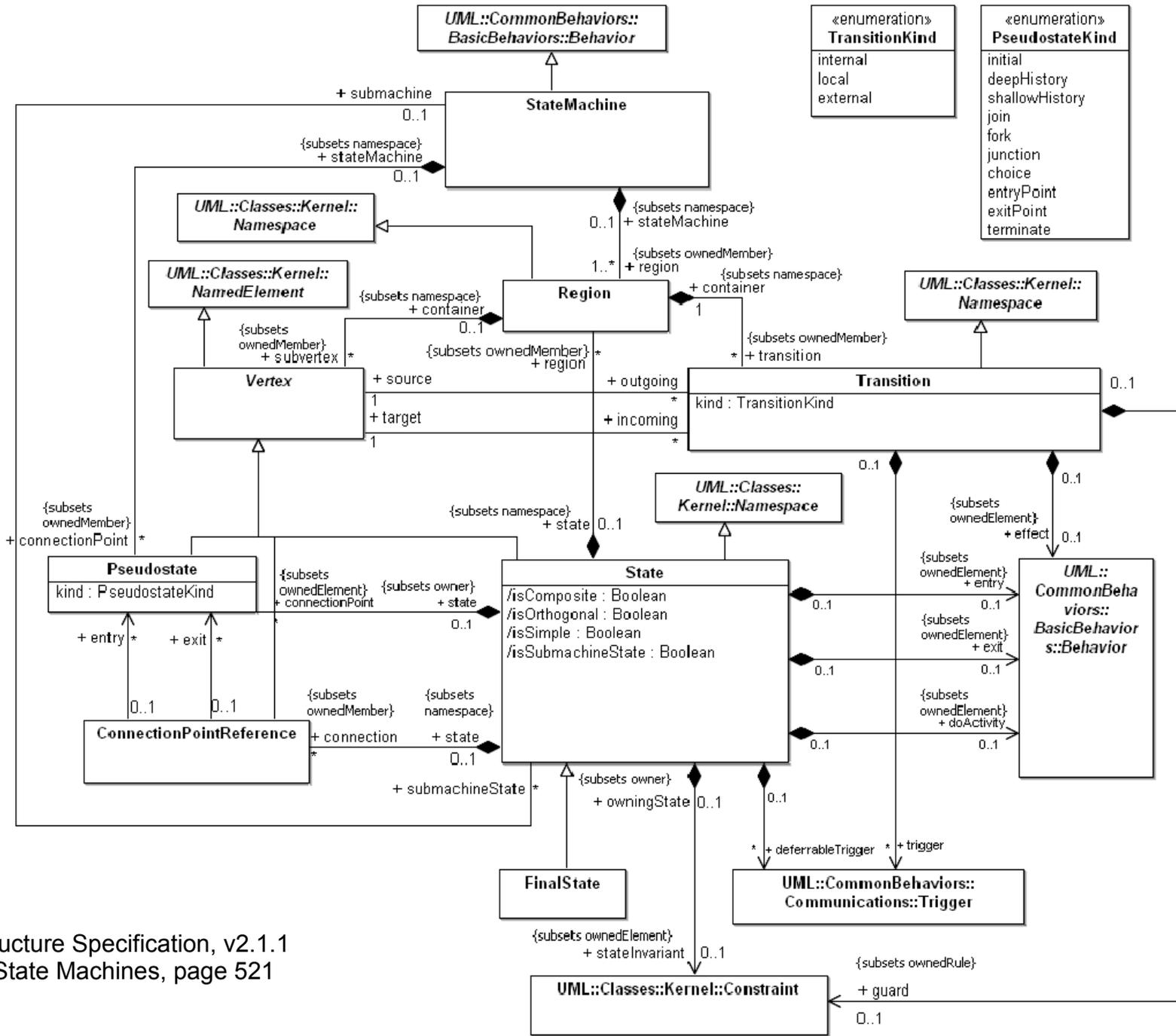
# Liens éléments modèle/méta-modèle

- ◆ Chaque élément du modèle
  - ◆ Est une « instance » d'un élément du méta-modèle (d'un méta-élément)
  - ◆ En respectant les contraintes définies dans le méta-modèle
- ◆ Exemple avec diagramme état



# Extrait méta-modèle UML 2.0

## ◆ Partie spécifiant les machines à états

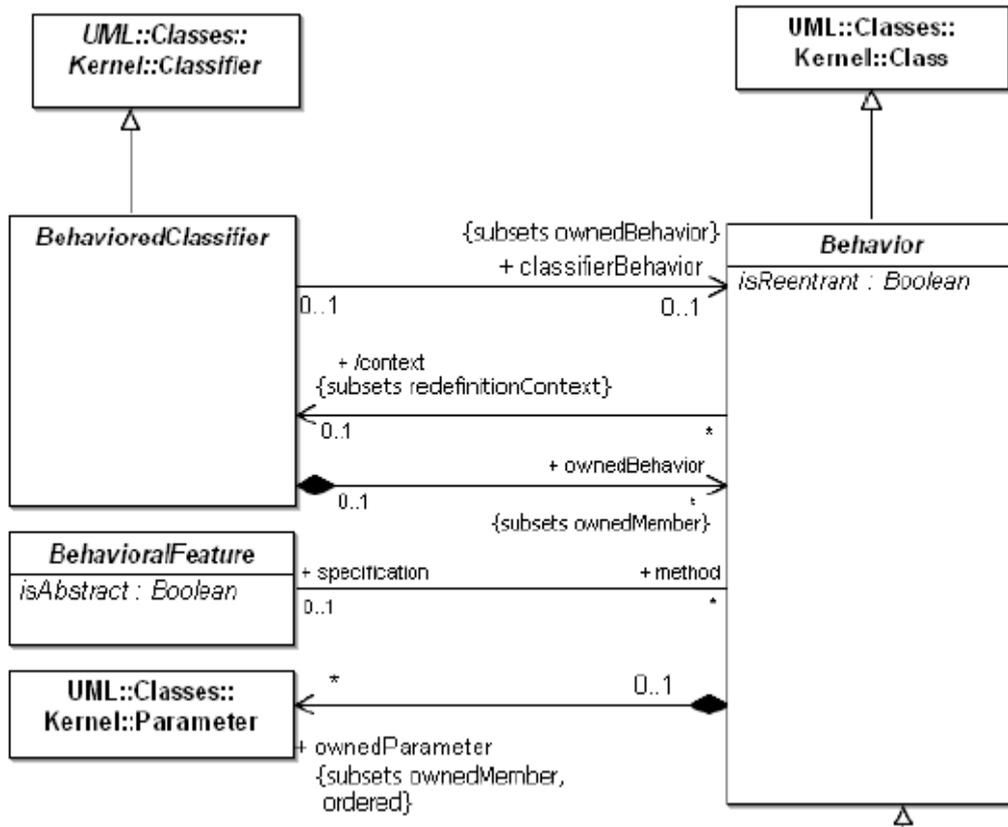


# Extrait méta-modèle UML 2.0

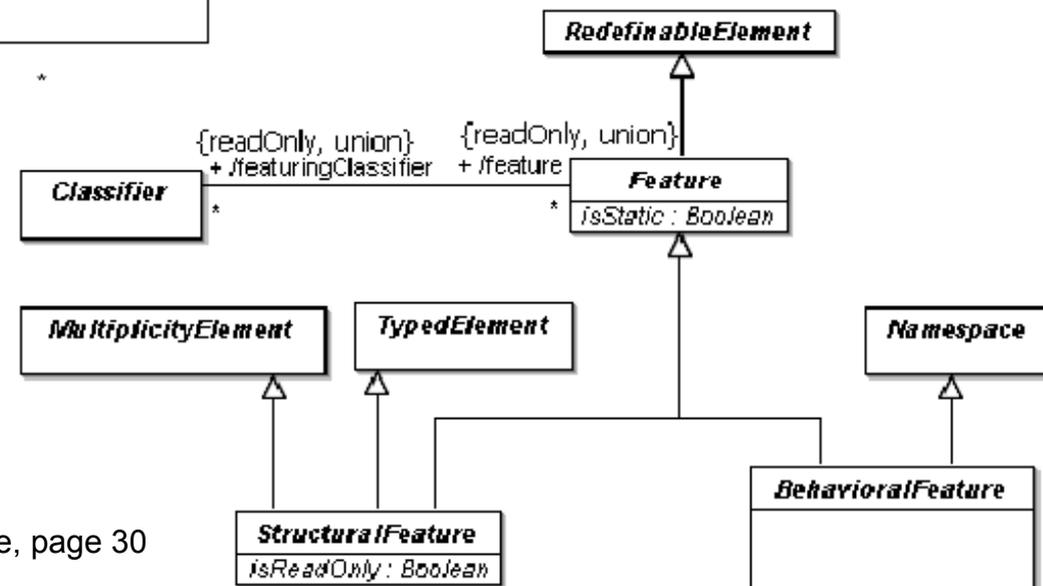
- ◆ Exemples de contraintes OCL pour la specification des machines à états
  - ◆ Invariants de la classe StateMachine
    - ◆ The classifier context of a state machine cannot be an interface.
      - ◆ `context->notEmpty() implies not context.oclIsKindOf(Interface)`
    - ◆ The context classifier of the method state machine of a behavioral feature must be the classifier that owns the behavioral feature.
      - ◆ `specification->notEmpty() implies (context->notEmpty() and specification->featuringClassifier->exists (c | c = context))`
    - ◆ The connection points of a state machine are pseudostates of kind entry point or exit point.
      - ◆ `connectionPoint->forall (c | c.kind = #entryPoint or c.kind = #exitPoint)`

# Extrait méta-modèle UML 2.0

## ◆ Pour comprendre les contraintes OCL



UML Superstructure Specification, v2.1.1  
Figure 13.6 - Common Behavior, page 424

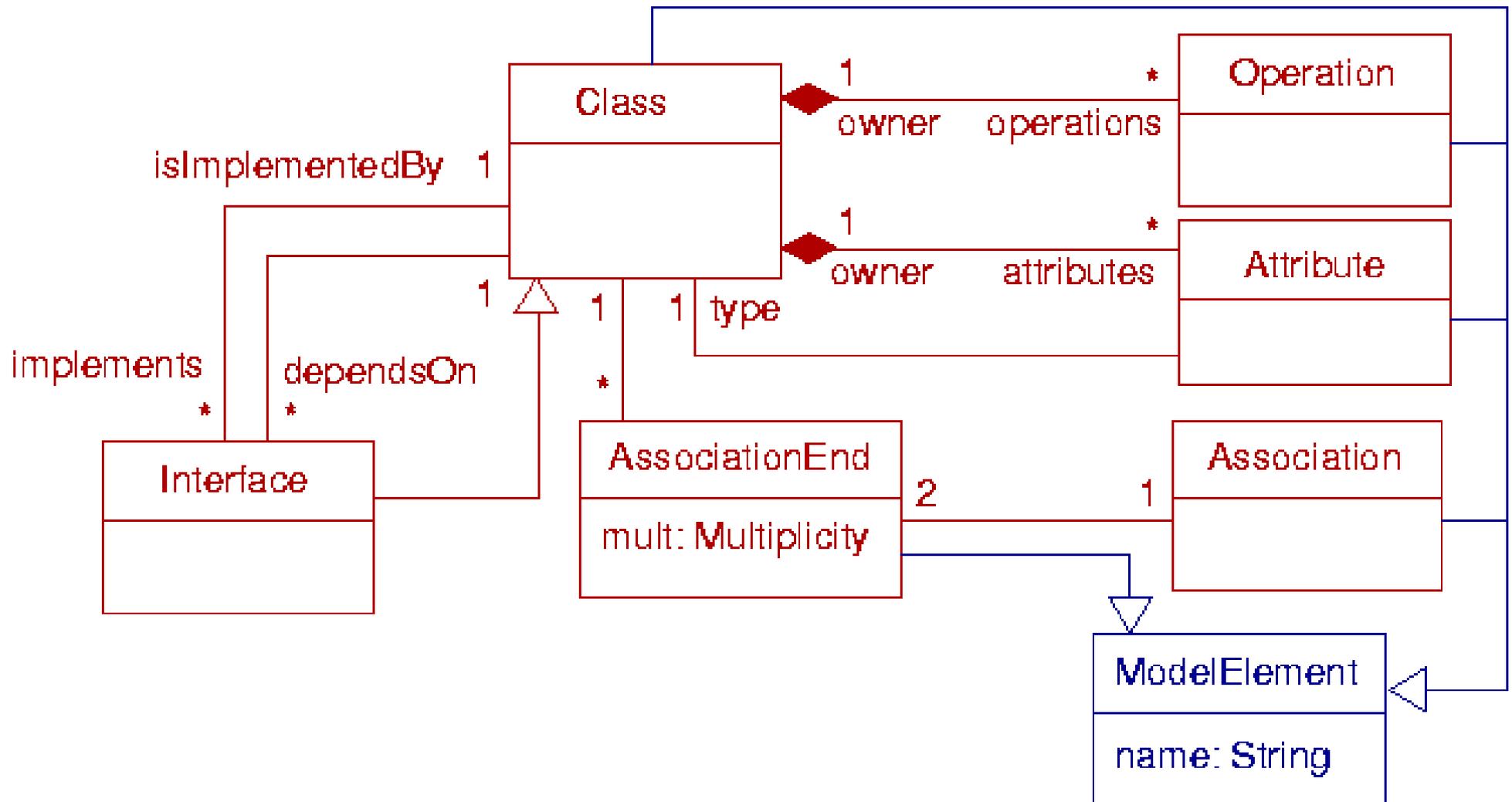


UML Superstructure Specification, v2.1.1  
Figure 7.10 - Features diagram of the Kernel package, page 30

# *Méta-modélisation UML*

- ◆ Le méta-modèle UML doit aussi être précisément défini
  - ◆ Il doit être conforme à un méta-modèle
  - ◆ C'est le méta-méta-modèle UML
    - ◆ Niveau M3
- ◆ Qu'est ce que le méta-modèle UML ?
  - ◆ Un diagramme de classe UML (avec contraintes OCL)
- ◆ Comment spécifier les contraintes d'un diagramme de classe UML ?
  - ◆ Via le méta-modèle UML
  - ◆ Ou plus précisément : via la partie du méta-modèle UML spécifiant les diagrammes de classes
- ◆ Méta-méta-modèle UML = copie partielle du méta-modèle UML

# M3 : Méta-méta-modèle UML (simplifié)



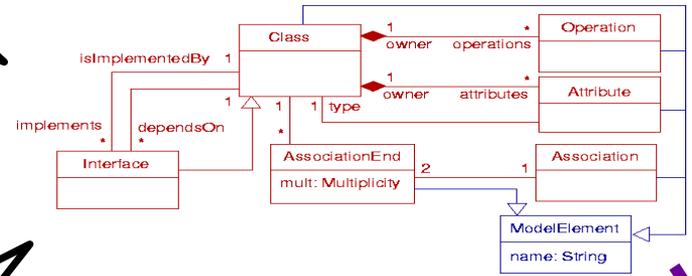
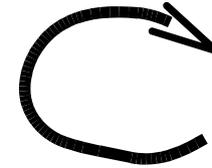
# ***Méta-modélisation UML***

- ◆ Méta-méta-modèle UML doit aussi être clairement défini
  - ◆ Il doit être conforme à un méta-modèle
  - ◆ Qu'est ce que le méta-méta-modèle UML ?
    - ◆ Un diagramme de classe UML
  - ◆ Comment spécifier les contraintes d'un diagramme de classe ?
    - ◆ Via la partie du méta-modèle UML spécifiant les diagrammes de classe
      - ◆ Cette partie est en fait le méta-méta-modèle UML
- ◆ Le méta-méta-modèle UML peut donc se définir lui même
  - ◆ Méta-circulaire
  - ◆ Pas besoin de niveau méta supplémentaire

# Hiérarchie de modélisation

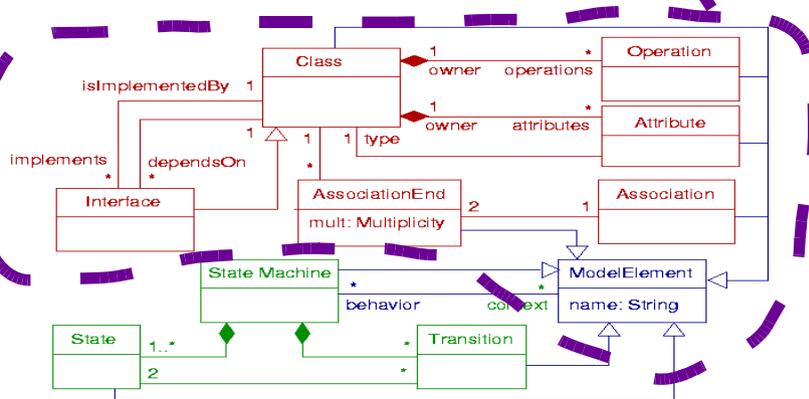
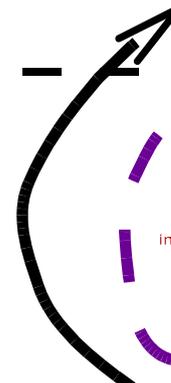
Niveau M3

conforme à



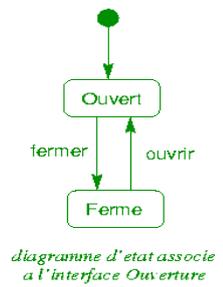
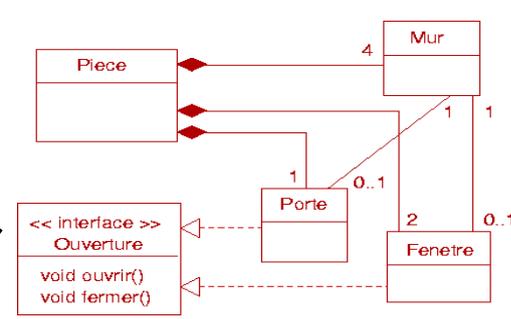
Niveau M2

conforme à



Niveau M1

conforme à



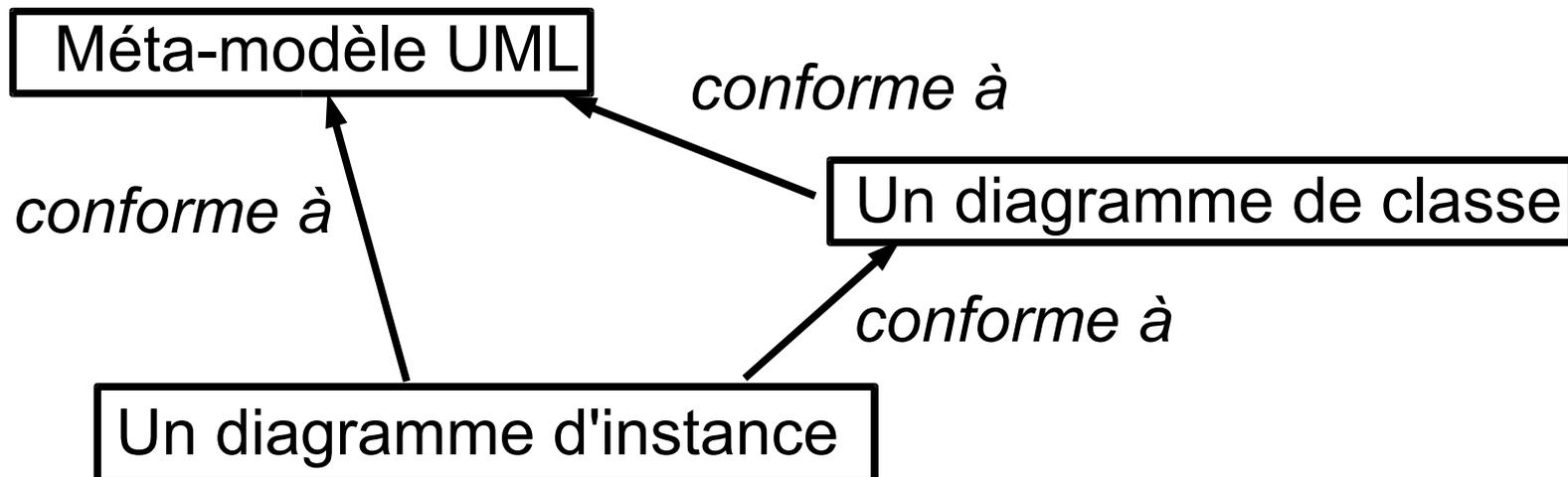
Niveau M0

conforme à



# Diagrammes d'instance UML

- ◆ Un diagramme d'instance est particulier car
  - ◆ Doit être conforme au méta-modèle UML
    - ◆ Qui définit la structure générale des diagrammes d'instances
  - ◆ Doit aussi être conforme à un diagramme de classe
    - ◆ Ce diagramme de classe est un méta-modèle pour le diagramme d'instance
  - ◆ Diagramme de classe est conforme également au méta-modèle UML



# *Hiérarchie de modélisation*

- ◆ Architecture à 4 niveaux
  - ◆ Conceptuellement pertinente
- ◆ En pratique
  - ◆ Certains niveaux sont difficiles à placer les uns par rapport aux autres
  - ◆ Cas du diagramme d'instance
    - ◆ Diagramme d'instance de l'application : niveau M1
    - ◆ Méta-modèle UML : niveau M2
    - ◆ Diagramme de classe de l'application : niveau M1 ou M2 ?
      - ◆ M1 normalement car modélise l'application et conforme au méta-modèle de niveau M2
      - ◆ Mais devrait être M2 rapport au diagramme d'instance qui est de niveau M1
- ◆ Conclusion
  - ◆ Pas toujours facile ni forcément pertinent de chercher à placer absolument les modèles à tel ou tel niveau
  - ◆ L'important est de savoir quel rôle (modèle / méta-modèle) joue un modèle dans une relation de conformité

# Syntaxe

- ◆ Un langage est défini par un méta-modèle
- ◆ Un langage possède une syntaxe
  - ◆ Définit comment représenter chaque type d'élément d'un modèle
    - ◆ Élément d'un modèle = instance d'un méta-élément
- ◆ Syntaxe textuelle

- ◆ Ensemble de mots-clé et de mots respectant des contraintes défini selon des règles précises

- ◆ Notions de syntaxe et de grammaire dans les langages

- ◆ Exemple pour langage Java :

```
public class MaClasse implements MonInterface { ... }
```

- ◆ Grammaire Java pour la déclaration d'une classe :

```
class_declaration ::= { modifier } "class" identifier  
[ "extends" class_name ] [ "implements" interface_name  
{ "," interface_name } ] "{" { field_declaration } "}"
```

# Syntaxe

## ◆ Syntaxe graphique

- ◆ Notation graphique, chaque type d'élément a une forme graphique particulière
- ◆ Exemple : relations entre classes/interfaces dans les diagrammes de classe UML

◆ Trait normal : association



◆ Flèche, trait pointillé : dépendance



◆ Flèche en forme de triangle, trait en pointillé : implémentation



◆ Flèche en forme de triangle, trait plein : spécialisation



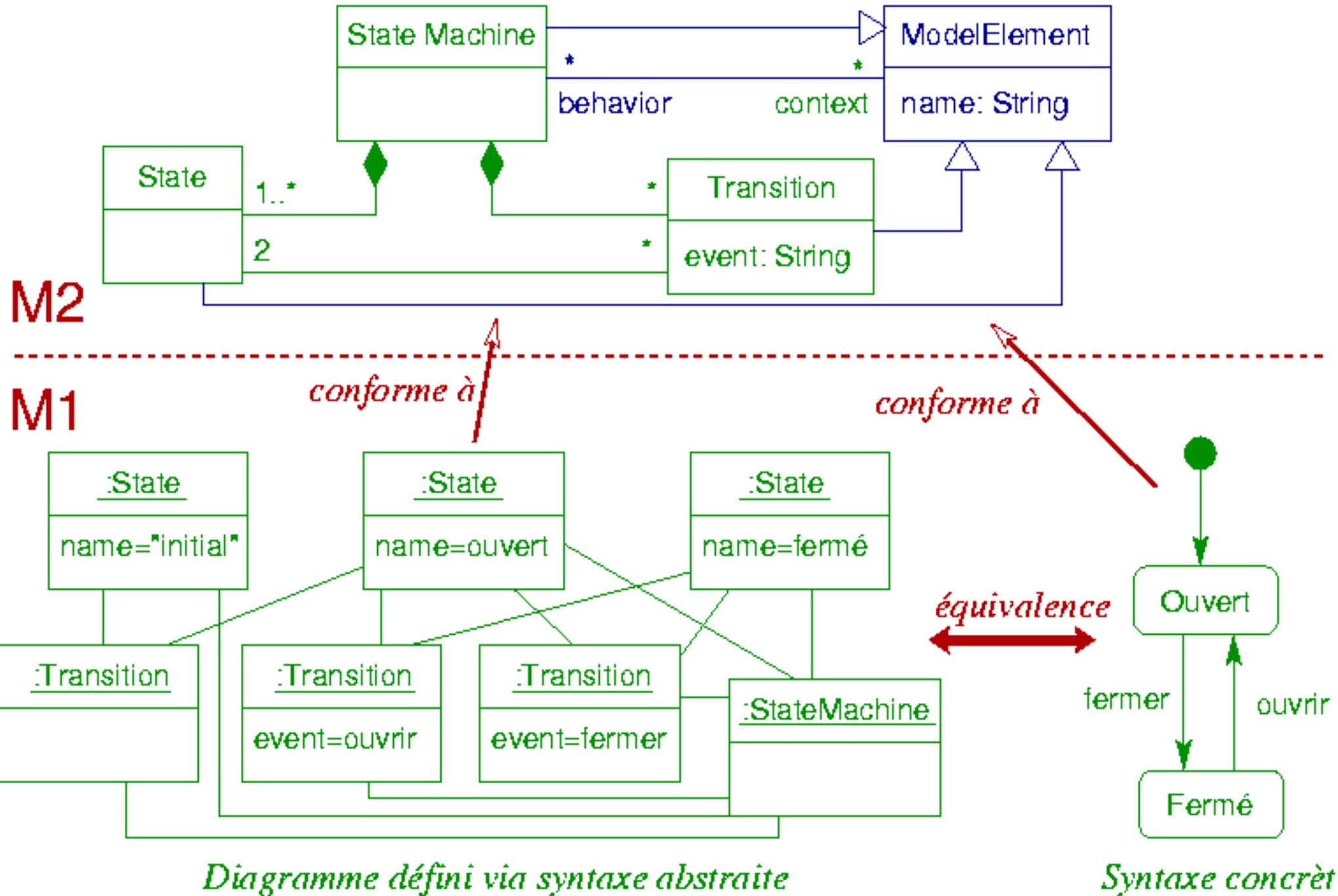
# Syntaxe

- ◆ Syntaxe abstraite/concrète
  - ◆ Abstraite
    - ◆ Les éléments et leurs relations sans une notation spécialisée
    - ◆ Correspond à ce qui est défini au niveau du méta-modèle, à des instances de méta-éléments
  - ◆ Concrète
    - ◆ Syntaxe graphique ou textuelle définie pour un type de modèle
    - ◆ Plusieurs syntaxes concrètes possibles pour une même syntaxe abstraite
- ◆ Un modèle peut être défini via n'importe quelle syntaxe
  - ◆ L'abstraite
  - ◆ Une des concrètes
- ◆ MOF : langage pour définir des méta-modèles
  - ◆ Pas de syntaxe concrète définie

# *Syntaxe*

- ◆ Exemple de la modélisation de la pièce
  - ◆ Syntaxe concrète
    - ◆ 2 diagrammes UML (classes et états) avec syntaxes graphiques spécifiques à ces types de diagrammes
  - ◆ Via la syntaxe abstraite
    - ◆ Diagramme d'instance (conforme au méta-modèle) précisant les instances particulières de classes, d'associations, d'états...
- ◆ Pour la partie diagramme d'états
  - ◆ Diagramme défini via syntaxe concrète : diagramme d'états de l'exemple
  - ◆ Diagramme défini via syntaxe abstraite : diagramme d'instances conforme au méta-modèle UML

# Syntaxe : exemple diagramme état



# *Définition de méta-modèles*

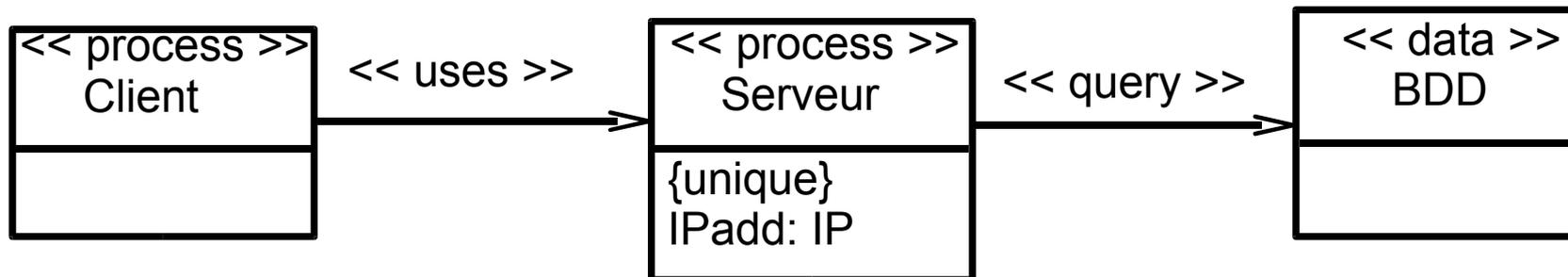
- ◆ But : définir un type de modèle avec tous ses types d'éléments et leurs contraintes de relation
- ◆ Plusieurs approches possibles
  - ◆ Définir un méta-modèle nouveau à partir de « rien », sans base de départ
    - ◆ On se basera alors sur un méta-méta-modèle existant comme MOF ou Ecore
  - ◆ Modifier un méta-modèle existant : ajout, suppression, modification d'éléments et des contraintes sur leurs relations
  - ◆ Spécialiser un méta-modèle existant en rajoutant des éléments et des contraintes (sans en enlever)
    - ◆ Correspond aux profils UML

# Profils UML

- ◆ Un profil est une spécialisation du méta-modèle UML
  - ◆ Ajouts de nouveaux types d'éléments
    - ◆ Et des contraintes sur leurs relations entre eux et avec les éléments d'UML
  - ◆ Ajouts de contraintes sur éléments existants d'UML
  - ◆ Ajouts de contraintes sur relations existantes entre les éléments d'UML
  - ◆ Aucune suppression de contraintes ou d'éléments
- ◆ Profil : mécanisme d'extension d'UML pour l'adapter à un contexte métier ou technique particulier
  - ◆ Profil pour composants EJB
  - ◆ Profil pour gestion bancaire
  - ◆ Profil pour architecture logicielle
  - ◆ ...

# Profils UML : éléments de spécialisation

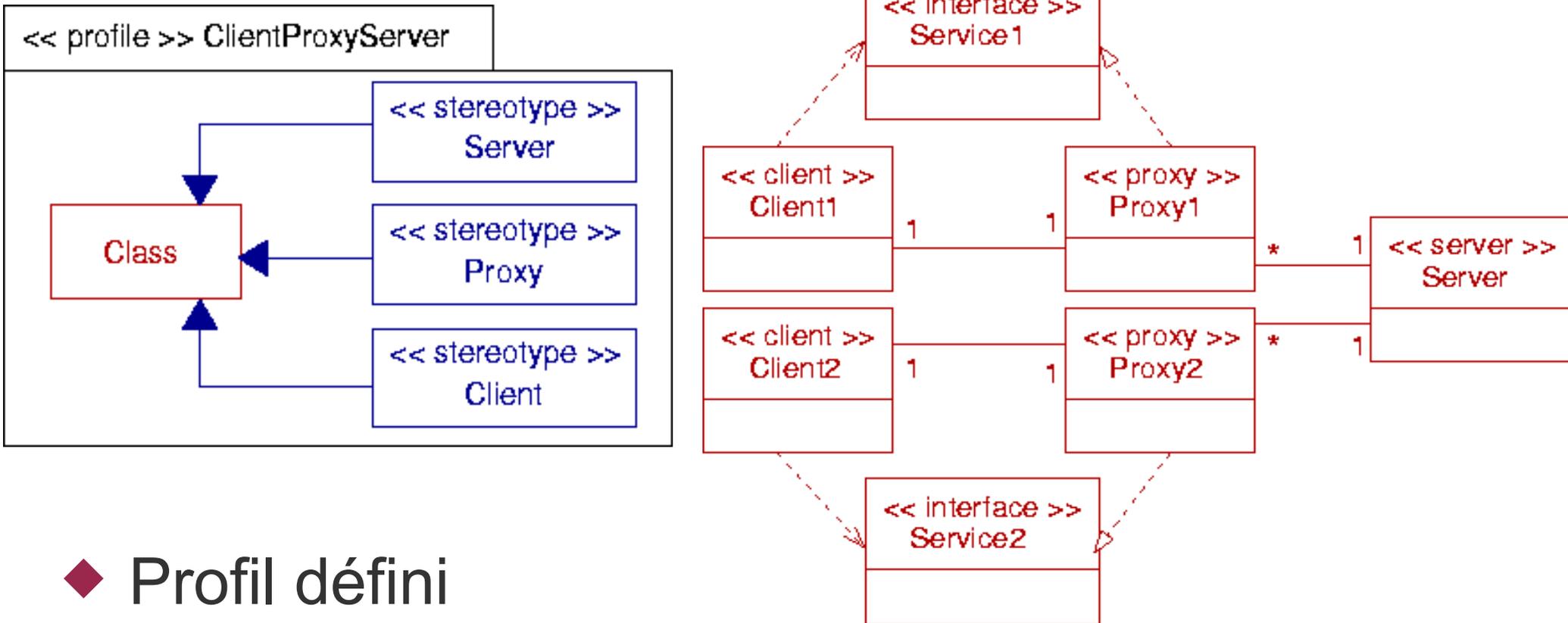
- ◆ Stéréotype
  - ◆ Extension, spécialisation d'un élément du méta-modèle
    - ◆ Classe, association, attribut, opération ...
    - ◆ Le nom d'un stéréotype est indiqué entre << ... >>
    - ◆ Il existe déjà des stéréotypes définis dans UML
      - ◆ << interface >> : une interface est un classifieur particulier (sans attribut)
- ◆ Tagged value (valeur marquée)
  - ◆ Pour marquer des attributs d'une classe pour préciser une contrainte ou un rôle particulier
    - ◆ Exemple : {unique} id: int



# Profils UML

- ◆ Un profil UML est composé de 3 types d'éléments
  - ◆ Des stéréotypes
  - ◆ Des tagged values
  - ◆ Des contraintes (exprimables en OCL)
    - ◆ Sur ces stéréotypes, tagged values
    - ◆ Sur des éléments du méta-modèle existant
    - ◆ Sur les relations entre les éléments
- ◆ Un profil UML est défini sous la forme d'un package stéréotypé << profile >>
- ◆ Exemple de profil : architecture logicielle
  - ◆ Des composants client et serveur
  - ◆ Un client est associé à un serveur via une interface de service par l'intermédiaire d'un proxy

# Exemple profil UML

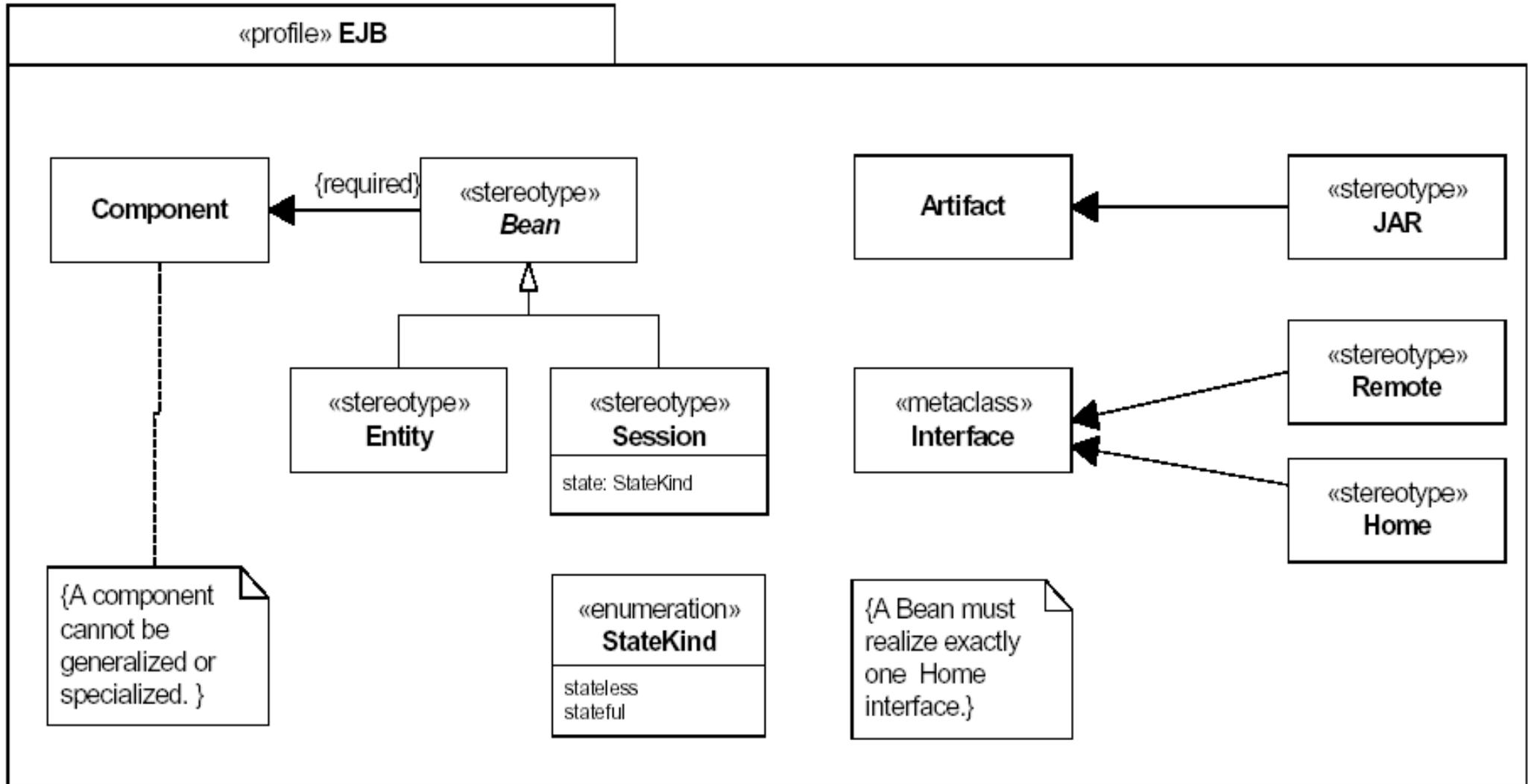


- ◆ Profil défini
  - ◆ nommé ClientProxyServer
  - ◆ Définit trois stéréotypes
    - ◆ Trois classes jouant un rôle particulier : extensions de la méta-classe Class du méta-modèle UML
      - ◆ Server, Proxy, Client

# Exemple profil UML

- ◆ Pour compléter le profil, ajout de contraintes OCL
  - ◆ Navigation sur le méta-modèle UML (simplifié) en considérant que la méta-classe Class a trois spécialisations (Server, Client, Proxy)
  - ◆ Un proxy associé à un client doit implémenter une des interfaces dont dépend le client et un proxy implémentant une interface d'un client doit avoir une association avec ce client
  - ◆ **context** Client **inv**:  
**let** proxies = self.associationEnd.association.associationEnd.-  
class -> select ( c | c.ocllsTypeOf(Proxy)) **in**  
**let** interfaces = self.dependsOn **in**  
interfaces -> forAll ( i | proxies.implements -> includes (i) **and**  
proxies -> forAll ( p | p.implements -> includes (i)  
**implies** p.hasClassRefWith(self)))
  - ◆ **context** Class **def**: hasClassRefWith(cl : Class) : Boolean =  
self.associationEnd.association.associationEnd.class  
-> exists ( c | c = cl )

# Autre exemple : profil EJB

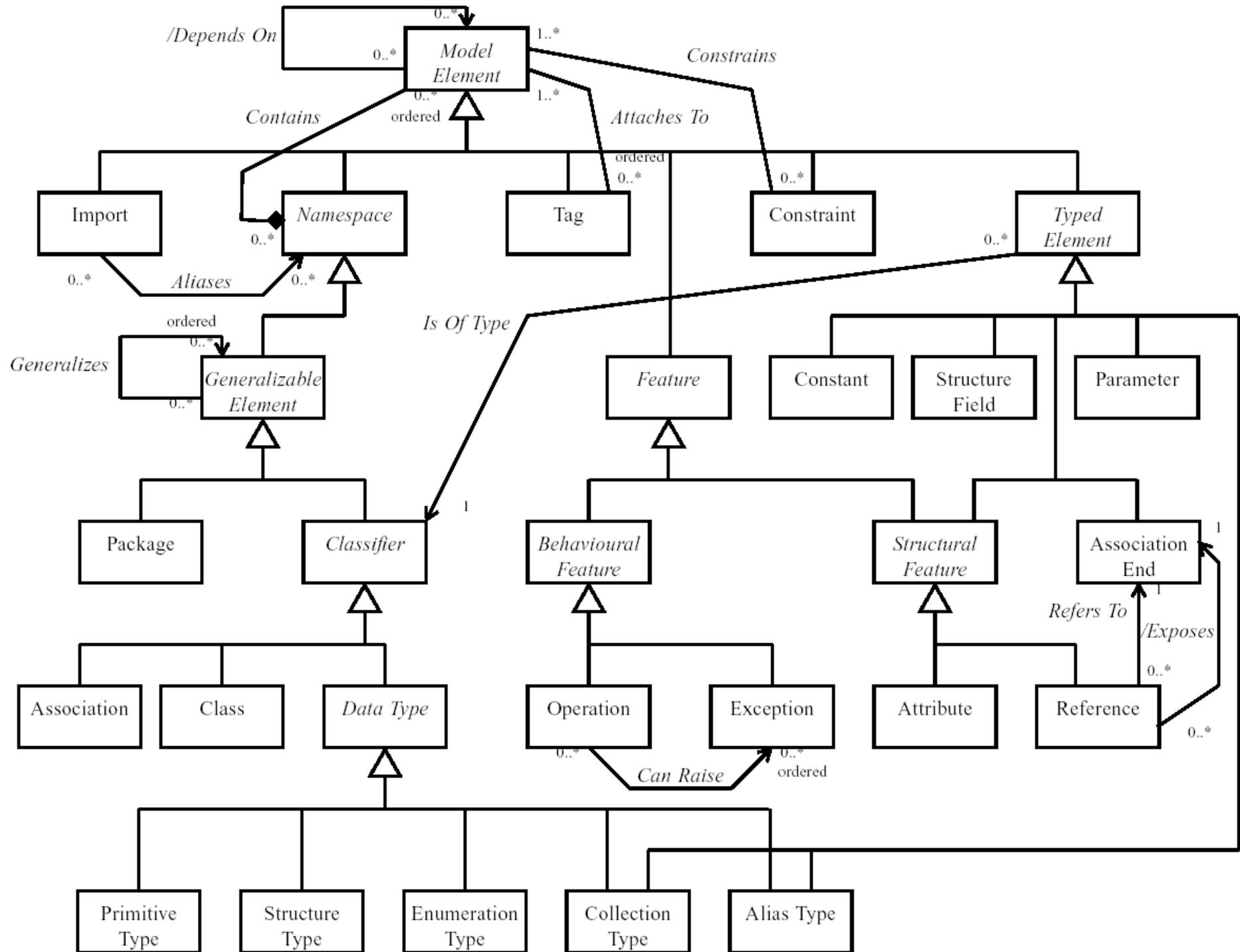


Source : norme UML 2.0

# *MOF*

- ◆ Meta Object Facilities (MOF) (version 2.0)
  - ◆ Méta-méta-modèle de référence pour l'OMG
  - ◆ Pas de syntaxe concrète
  - ◆ Aligné depuis la version 2.0 avec le « noyau » du méta-modèle d'UML 2.0
  - ◆ Décomposé en 2 parties
    - ◆ E-MOF : essential MOF
      - ◆ Méta-modèle noyau
    - ◆ C-MOF : complete MOF
      - ◆ Méta-modèle plus complet
    - ◆ E-MOF et C-MOF peuvent se définir mutuellement et chacun eux-mêmes
  - ◆ Définition d'un méta-modèle via le MOF
    - ◆ Méta-modèle = ensemble d'instances de méta-éléments du MOF associées entre elles

# MOF : méta-modèle (version 1.4)

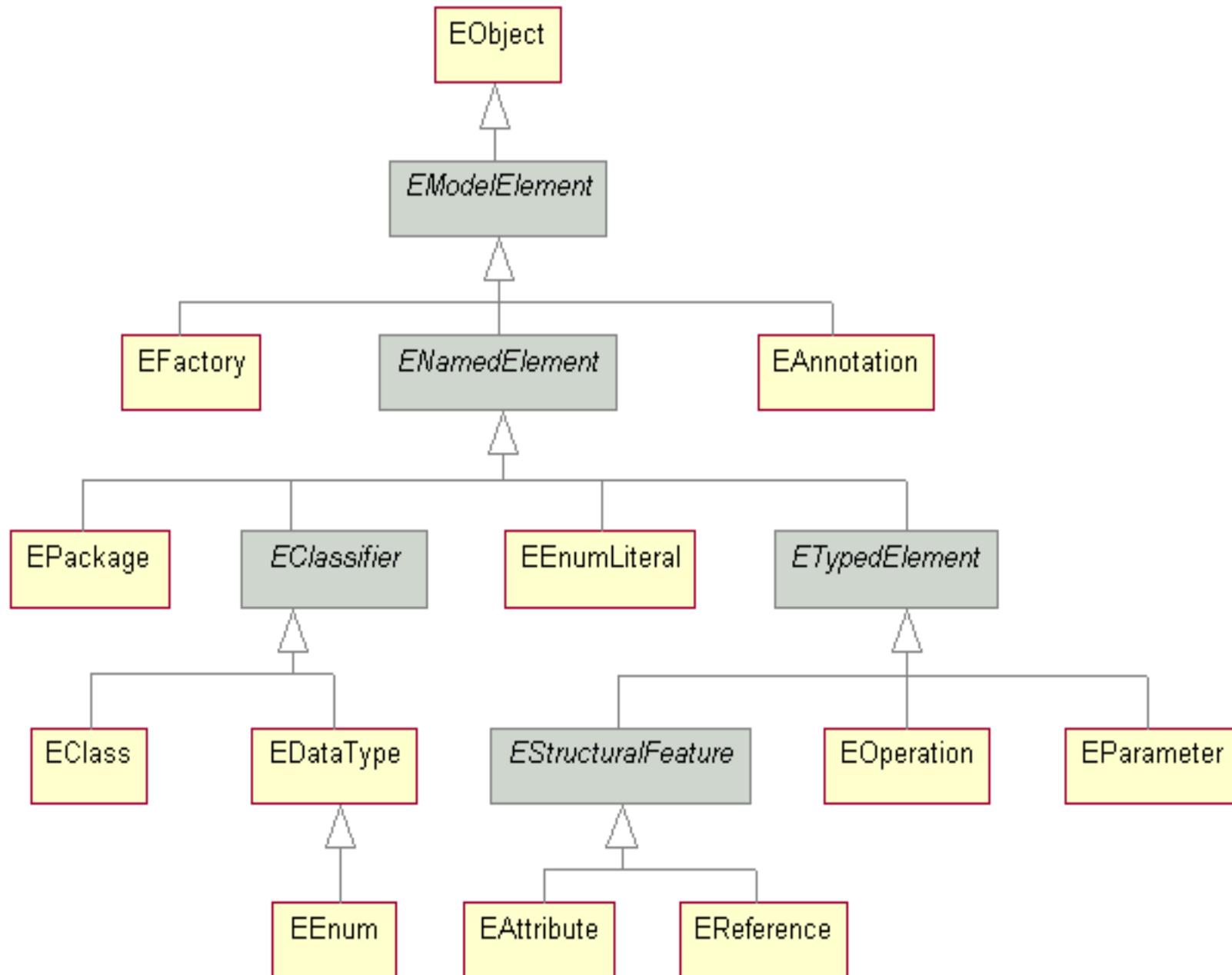


# *Ecore*

- ◆ Eclipse Modeling Framework (EMF)
  - ◆ Framework de modélisation intégré dans l'atelier de développement Eclipse (IBM) pour le langage Java
  - ◆ But : modéliser des programmes Java et travailler au niveau modèle en plus du code
  - ◆ Mais peut aussi servir à créer des modèles et des méta-modèles
- ◆ Ecore
  - ◆ Méta-modèle intégré dans EMF
  - ◆ Méta-modèle « minimal » aligné sur E-MOF

# Ecore

## ◆ Éléments du méta-modèle Ecore



# Ecore (détails MM)

