UML : concept objet et diagramme de classes

Achref El Mouelhi

Docteur de l'université d'Aix-Marseille
Chercheur en programmation par contrainte (IA)
Ingénieur en génie logiciel

elmouelhi.achref@gmail.com

UNIFIED
MODELING
LANGUAGE

Plan

Plan

UML

Diagramme de Classe

- Le diagramme de structure le plus important dans la réalisation d'un projet
- Utilisant le concept objet
- Pouvant être transformé, en respectant certaines règles, en MCD (le modèle conceptuel de données de Merise)

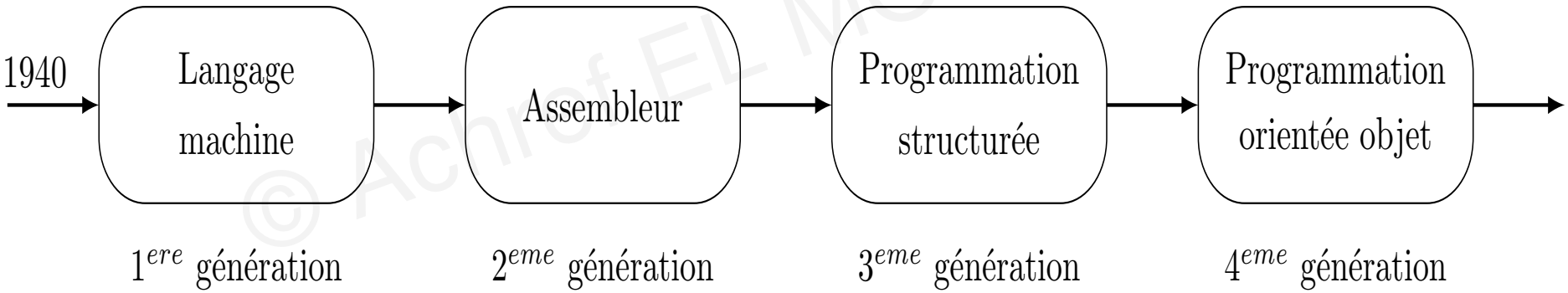
UML

Les éléments de base dans un diagramme de classe

- Classe
- Attribut
- Méthode
- Association
- Multiplicité

UML

Histoire de la programmation



UML

Langage machine

- Le langage natif d'un processeur (langage de bas niveau)
- Les traitements (les instructions) et les données (les variables) sont tous codés en binaire (suite des 0 et 1)

UML

Langage machine

- Le langage natif d'un processeur (langage de bas niveau)
- Les traitements (les instructions) et les données (les variables) sont tous codés en binaire (suite des 0 et 1)

Remarque

- La lisibilité est extrêmement difficile
- La réutilisation est quasi-impossible
- La notion de module n'existe pas

UML

L'assembleur

- Une réécriture plus lisible du langage machine
- Une instruction est formée de nom de l'opération (`mov`, `add`...) et d'une ou plusieurs opérandes (variables)
- Un programme assembleur est traduit en langage machine

UML

L'assembleur

- Une réécriture plus lisible du langage machine
- Une instruction est formée de nom de l'opération (`mov`, `add`...) et d'une ou plusieurs opérandes (variables)
- Un programme assembleur est traduit en langage machine

Remarque

- La lisibilité est légèrement meilleure mais reste difficile
- La réutilisation est toujours quasi-impossible
- La notion de module n'existe toujours pas

UML

La programmation structurée (ou procédurale)

- Un programme est composé d'un ensemble d'appel à des procédures réalisant chacune une tâche bien définie
- Une procédure peut appeler plusieurs procédures (y compris elle-même, on parle dans ce cas de récursivité)

UML

La programmation structurée (ou procédurale)

- Un programme est composé d'un ensemble d'appel à des procédures réalisant chacune une tâche bien définie
- Une procédure peut appeler plusieurs procédures (y compris elle-même, on parle dans ce cas de récursivité)

Remarque

- La lisibilité est considérablement améliorée
- Une procédure peut être appelée plusieurs fois par plusieurs procédures différentes (la réutilisation est nettement améliorée)
- Apparition de notion de module

UML

La programmation orientée objet

- Tout concept, idée, opération... est considéré comme un objet
- Un objet peut contenir certains autres objets et avoir des relations avec certains autres
- Chaque objet possède une structure interne et peut avoir plusieurs états différents

UML

La programmation orientée objet

- Tout concept, idée, opération... est considéré comme un objet
- Un objet peut contenir certains autres objets et avoir des relations avec certains autres
- Chaque objet possède une structure interne et peut avoir plusieurs états différents

Remarque

- Tout passe par des objets. Un programme correspond à un ensemble d'instanciation d'objets : facile à lire
- Un objet peut être instancié et utilisé par plusieurs autres objets (ce qui facilite la réutilisation des objets)
- Amélioration de la notion de module (Package)

UML

Remarque

- Certains compilateurs traduisent les codes soit en langage machine, soit en assembleur, soit en langage de programmation d'un niveau inférieur
- Certains compilateurs sont implémentés avec d'autres langages de programmation (par example la machine virtuelle de Java qui est implémentée en **C++**...)

UML

Pourquoi autant de génération ?

- (Plus) de problème de mémoire
- Des processeurs plus rapides
- Besoins différents
  - **COBOL** : pour les systèmes de gestion
  - **Java** : pour les problèmes complexes
  - **LISP** : pour les programmes d'intelligence artificielle
  - **PROLOG** : pour les problèmes de logique

Évolution de LOO

- **Simula** (1967)
- **Smalltalk** (1971 puis 1980)
- **C++** (1983)
- **Python** (1989)
- **Java** (1995)
- **C#** (2002)
- **PHP** (depuis la version 5 sortie en 2004)
- ...

Évolution de LOO

- **Simula** (1967)
- **Smalltalk** (1971 puis 1980)
- **C++** (1983)
- **Python** (1989)
- **Java** (1995)
- **C#** (2002)
- **PHP** (depuis la version 5 sortie en 2004)
- ...

Plusieurs langages de programmation non-objet ont proposé une nouvelle version orientée objet.

UML

Deux catégories de LOO

- Les langages à classes : **Java**, **C#**, **TypeScript**, **C++**, **Python**...
- Les langages à prototypes : **JavaScript**

UML

Pourquoi un LOO ?

- Supporté par tous les systèmes d'exploitation
- Adopté par **Microsoft**
- Adopté par les universitaires (en enseignement et en recherche)
- Adopté par la communauté open-source
- Adopté par les grandes entreprises
- ...

UML

Qu'est ce qu'un objet ?

- Une représentation miniature d'un objet réel
- Tout concept du monde réel est modélisé par un objet
- Un outil sur lequel se baser pour créer des choses qui n'existent pas encore
- Nous pouvons le considérer comme une boite noire : inaccessible directement qu'à travers une interface

UML

Un objet est une entité autonome

- véritable petit programme
- possède une durée de vie
- possède un état : un ensemble de valeurs
- accessible au monde extérieur via une interface

UML

Un objet est une entité autonome

- véritable petit programme
- possède une durée de vie
- possède un état : un ensemble de valeurs
- accessible au monde extérieur via une interface

Un programme est composé de plusieurs objets autonomes qui vont interagir entre eux

UML

De quoi est composé un objet ?

- un ensemble d'attributs (chaque attribut a un nom et une valeur) $\equiv$ données (qui définissent l'aspect statique de l'objet)
- un ensemble de méthodes (des fonctions, comme en programmation procédurale, dédiées à un type d'objet) $\equiv$ traitements (qui définissent l'aspect dynamique de l'objet)

UML

Les attributs

- un attribut, selon le contexte, peut être aussi appelé champ ou variable d'instance
- un attribut peut avoir un type (ce n'est pas une obligation, tout dépend du langage de programmation)
  - chaîne de caractère
  - entier
  - booléen
  - date
  - ...

UML

Remarque

- Un objet peut avoir comme attribut un autre objet
- Comme les fonctions en programmation procédurale, une méthode peut appeler plusieurs autres
- Contrairement à la programmation procédurale, les données et les traitements ne sont pas séparés

UML

Exemple : maVoiture

- Une voiture peut avoir comme attribut
  - numéro d'immatriculation (chaîne ou entier)
  - marque (chaîne)
  - modèle (chaîne ou entier)
  - puissance (entier)
  - couleur (chaîne)
  - ...
- elle peut avoir comme méthode
  - avancer
  - reculer
  - freiner
  - klaxonner
  - ...

Exemple : voiture

- numéro d'immatriculation : LY 069 ON
- marque : Peugeot
- modèle : 3008
- puissance : 8
- couleur : blanche

Exemple : voiture

- numéro d'immatriculation : LY 069 ON
- marque : Peugeot
- modèle : 3008
- puissance : 8
- couleur : blanche

Remarques

- L'ensemble de valeurs d'un objet définit son état

Exemple : voiture

- numéro d'immatriculation : LY 069 ON
- marque : Peugeot
- modèle : 3008
- puissance : 8
- couleur : blanche

Remarques

- L'ensemble de valeurs d'un objet définit son état
- Si je repeins la voiture précédente en noir (couleur = noir) alors **nous disons que l'objet a changé d'état**

Exemple : voiture

- numéro d'immatriculation : LY 069 ON
- marque : Peugeot
- modèle : 3008
- puissance : 8
- couleur : blanche

Remarques

- L'ensemble de valeurs d'un objet définit son état
- Si je repeins la voiture précédente en noir (couleur = noir) alors **nous disons que l'objet a changé d'état**
- La voiture peut donc changer d'état mais il s'agit toujours du même objet

UML

Attention

- Ne pas confondre objets **identiques** et objets **égaux**

UML

Attention

- Ne pas confondre objets **identiques** et objets **égaux**
- Deux objets sont égaux si, au moment de comparaison, leurs attributs respectifs ont les mêmes valeurs

UML

Attention

- Ne pas confondre objets **identiques** et objets **égaux**
- Deux objets sont égaux si, au moment de comparaison, leurs attributs respectifs ont les mêmes valeurs
- Deux objets sont identiques s'ils ont le même OID (Object IDentifier)

UML

Attention

- Ne pas confondre objets **identiques** et objets **égaux**
- Deux objets sont égaux si, au moment de comparaison, leurs attributs respectifs ont les mêmes valeurs
- Deux objets sont identiques s'ils ont le même OID (Object IDentifier)
- Si deux objets sont identiques alors ils sont aussi égaux

UML

Attention

- Ne pas confondre objets **identiques** et objets **égaux**
- Deux objets sont égaux si, au moment de comparaison, leurs attributs respectifs ont les mêmes valeurs
- Deux objets sont identiques s'ils ont le même OID (Object IDentifier)
- Si deux objets sont identiques alors ils sont aussi égaux
- La réciproque est bien évidemment fausse

UML

Exercice d'application

- Définissons un objet `monLivre` de type `Livre`

UML

Exercice d'application

- Définissons un objet `monLivre` de type `Livre`
- Déterminons ses principaux attributs ainsi que leurs valeurs

UML

Exercice d'application

- Définissons un objet `monLivre` de type `Livre`
- Déterminons ses principaux attributs ainsi que leurs valeurs
- Définissons ses méthodes de base

UML

Cycle de vie

- **Création :** se réalise via une méthode spécifique dite `constructeur`

UML

Cycle de vie

- **Création :** se réalise via une méthode spécifique dite `constructeur`
  - Pour certains LOO comme **Java**, **C++**, **C#**..., le constructeur porte le nom de la classe des objets

UML

Cycle de vie

- **Création :** se réalise via une méthode spécifique dite `constructeur`
  - Pour certains LOO comme **Java**, **C++**, **C#**..., le constructeur porte le nom de la classe des objets
- L'objet sera par la suite utilisé pour effectuer des tâches bien précises

UML

Cycle de vie

- **Création :** se réalise via une méthode spécifique dite `constructeur`
  - Pour certains LOO comme **Java**, **C++**, **C#**..., le constructeur porte le nom de la classe des objets
- L'objet sera par la suite utilisé pour effectuer des tâches bien précises
- **Destruction :** selon le LOO soit d'une façon implicite quand on ne fait plus référence à l'objet (comme en **Java, C#**...), soit d'une façon explicite en appelant une méthode bien spécifique `destructeur` (comme en **PHP**, **C++**...)

UML

Le garbage collector (ramasse miettes)

- C'est le destructeur d'objets en **Java**, **C#** et **Smalltalk**
- Il détruit automatiquement les objets qui ne sont plus utilisés
- Un objet n'est plus utilisé s'il n'est plus référencé
- Lorsque le garbage collector se déclenche, il détruit les objets non-utilisés

UML

Qu'est ce qu'une classe en POO ?

- Ça correspond à un plan, un moule, une usine...
- C'est une description abstraite d'un type d'objets
- Elle regroupe un ensemble d'objets ayant les mêmes propriétés statiques (attributs) et dynamiques (méthodes)

UML

Qu'est ce qu'une classe en POO ?

- Ça correspond à un plan, un moule, une usine...
- C'est une description abstraite d'un type d'objets
- Elle regroupe un ensemble d'objets ayant les mêmes propriétés statiques (attributs) et dynamiques (méthodes)

Qu'est ce que c'est la notion d'instance ?

- Une instance correspond à un objet créé à partir d'une classe (via le constructeur)

UML

Qu'est ce qu'une classe en POO ?

- Ça correspond à un plan, un moule, une usine...
- C'est une description abstraite d'un type d'objets
- Elle regroupe un ensemble d'objets ayant les mêmes propriétés statiques (attributs) et dynamiques (méthodes)

Qu'est ce que c'est la notion d'instance ?

- Une instance correspond à un objet créé à partir d'une classe (via le constructeur)
- L'instanciation : création d'un objet d'une classe

UML

Qu'est ce qu'une classe en POO ?

- Ça correspond à un plan, un moule, une usine...
- C'est une description abstraite d'un type d'objets
- Elle regroupe un ensemble d'objets ayant les mêmes propriétés statiques (attributs) et dynamiques (méthodes)

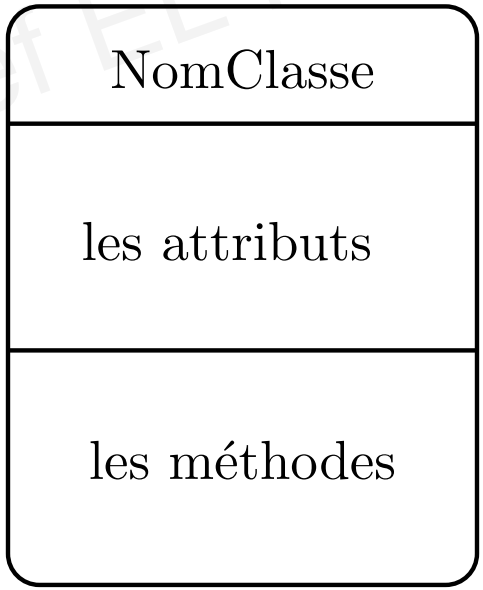
Qu'est ce que c'est la notion d'instance ?

- Une instance correspond à un objet créé à partir d'une classe (via le constructeur)
- L'instanciation : création d'un objet d'une classe
- instance $\equiv$ objet

Dans un digramme de classes **UML**

Une classe est représentée par un **classeur** contenant trois parties :

- première partie dédiée au nom de la classe
- seconde partie dédiée aux attributs
- dernière partie dédiée aux méthodes



UML

De quoi est composé une classe ?

- Attribut : visibilité + nom + type
  - LOO faiblement typé, comme **PHP**, **Pyhon**..., n'exigent pas un type pour les attributs
  - LOO fortement typé, comme **Java**, **C++**, **C#**..., exigent un type pour chaque attribut
- Méthode : nom + arguments + valeur de retour $\equiv$ signature : exactement comme les fonctions en procédurale

UML

Remarque

- Un attribut ou une méthode peut être de classe (static) ou d'instance
- Un attribut (ou une méthode) est dit static si sa valeur est partagée par tous les objets de la classe

UML

Une classe

- peut cacher son implémentation
- ne montre aux autres objets que ce qu'elle autorise : interface

UML

Une classe

- peut cacher son implémentation
- ne montre aux autres objets que ce qu'elle autorise : interface

Propriétés

- Les objets interagissent les uns avec les autres via les interfaces
- Ce qui n'est pas dans l'interface n'est ni visible ni accessible aux autres objets

UML

Une classe

- peut cacher son implémentation
- ne montre aux autres objets que ce qu'elle autorise : interface

Propriétés

- Les objets interagissent les uns avec les autres via les interfaces
- Ce qui n'est pas dans l'interface n'est ni visible ni accessible aux autres objets

La dissociation de l'interface et de l'implémentation : **encapsulation**

UML

L'encapsulation, pourquoi ?

- Protection des données
  - données inaccessibles de l'extérieur
  - accès uniquement via l'interface
  - possibilité de tracer tous les accès aux données
- focalisation sur les services rendus par les objets plutôt que sur leur structure

UML

L'encapsulation, comment ?

- En définissant des niveaux de visibilité
- Les méthodes comme les attributs sont concernés

UML

L'encapsulation, comment ?

- En définissant des niveaux de visibilité
- Les méthodes comme les attributs sont concernés

(Trois) niveaux de visibilités

- `public` : visible par tous les autres objets
- `protected` : visible par certains objets (à voir plus tard)
- `private` : visible seulement depuis l'intérieur de l'objet

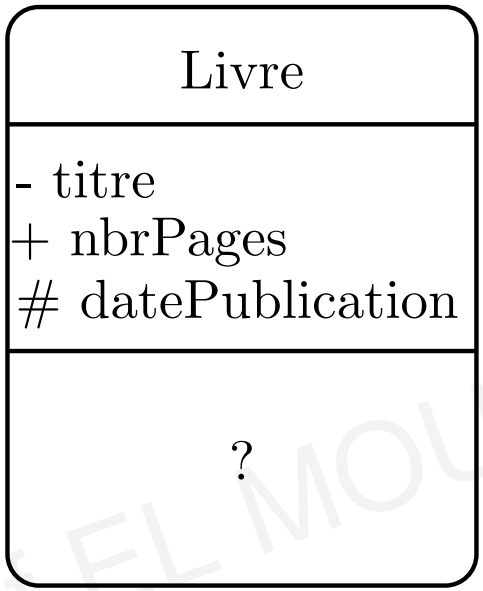
UML

Dans un diagramme de classes

On peut utiliser

- `+` : pour indiquer que la propriété a une visibilité `public`
- `-` : pour indiquer que la propriété a une visibilité `private`
- `#` : pour indiquer que la propriété a une visibilité `protected`

UML



Dans cet exemple

- `titre` a une visibilité `private`
- `nbrPages` a une visibilité `public`
- `datePublication` a une visibilité `protected`

UML

Certains langages, comme **Smalltalk** exige que :

- les attributs soient `private`
- les méthodes soient `public`

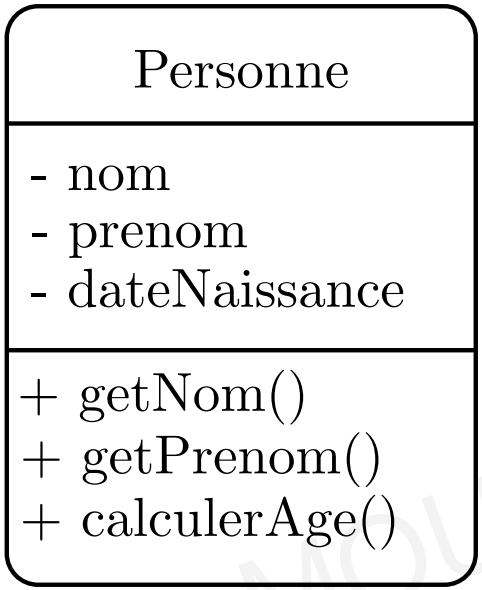
UML

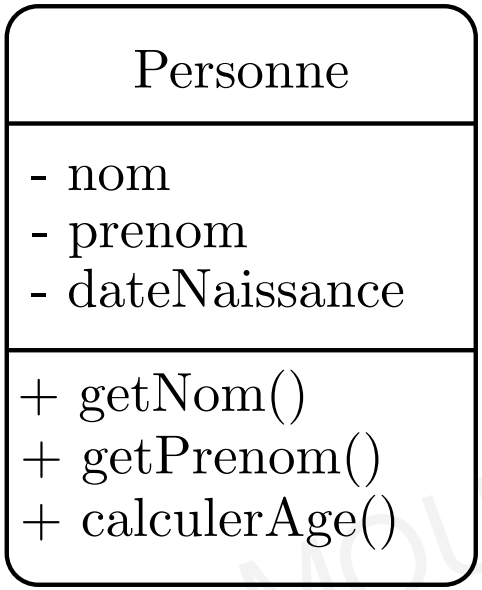
Certains langages, comme **Smalltalk** exige que :

- les attributs soient `private`
- les méthodes soient `public`

En **Java**, **PHP**, **C#**

C'est une recommandation





Encapsulation

- Seules les trois méthodes `getNom()`, `getPrenom()` et `calculerAge()` sont visibles par les autres objets
- Les autres objets ignorent l'existence de l'attribut `dateNaissance`

UML

Les méthodes publiques

- constituent l'interface de la classe
- sont accessibles de l'extérieur

Les méthodes privées

- sont déclenchées par le biais d'autres méthodes privées ou publiques
- ne sont pas accessibles depuis l'extérieur
- ne sont accessible qu'au développeur de la classe

UML

Question

Si les attributs sont toujours privés, comment modifier leurs valeurs ?

UML

Question

Si les attributs sont toujours privés, comment modifier leurs valeurs ?

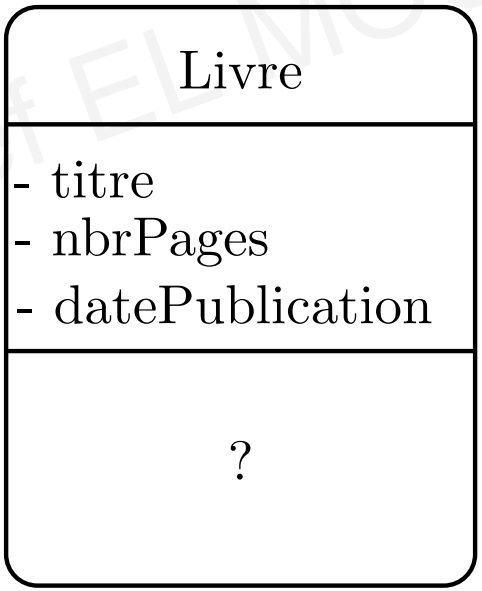
Réponse

- Les getters (accesseurs) pour récupérer la valeur (`getNomAttribut()`)
- Les setters (mutateurs) pour modifier la valeur (`setNomAttribut()`)

UML

Exercice

Définir les getters/setters de la classe `Livre`



UML

Comment créer un objet d'une classe ?

- en utilisant le constructeur de la classe
- en invoquant l'opérateur `new`

UML

Comment créer un objet d'une classe ?

- en utilisant le constructeur de la classe
- en invoquant l'opérateur `new`

Qu'est ce qu'un constructeur ?

- une méthode particulière et spécifique à chaque classe
- elle porte le nom de la classe et n'a pas de valeur de retour

UML

Exemple

- En **Java**, **C++**, **PHP**, **C#**, **TypeScript** : `new Personne();`
- En **Smalltalk** : `Personne new`

UML

Récapitulons

Une classe a :

- des attributs
- des méthodes dont
  - le constructeur
  - les getters
  - les setters

UML

Considérons la classe `Adresse` suivante

UML

Considérons la classe `Adresse` suivante

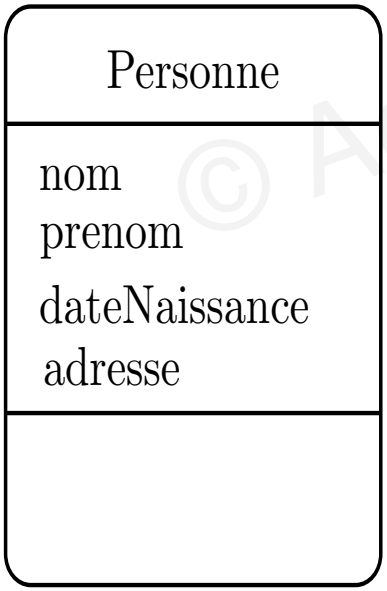
Hypothèse

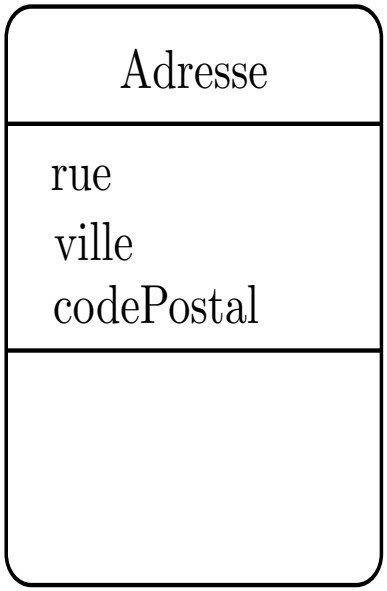
Supposant que chaque personne possède une adresse (objet créé à partir de la classe `Adresse`)

UML

Comment schématiser cela en **UML** ?

En ajoutant dans la classe `Personne` un attribut `adresse` de type `Adresse` ?

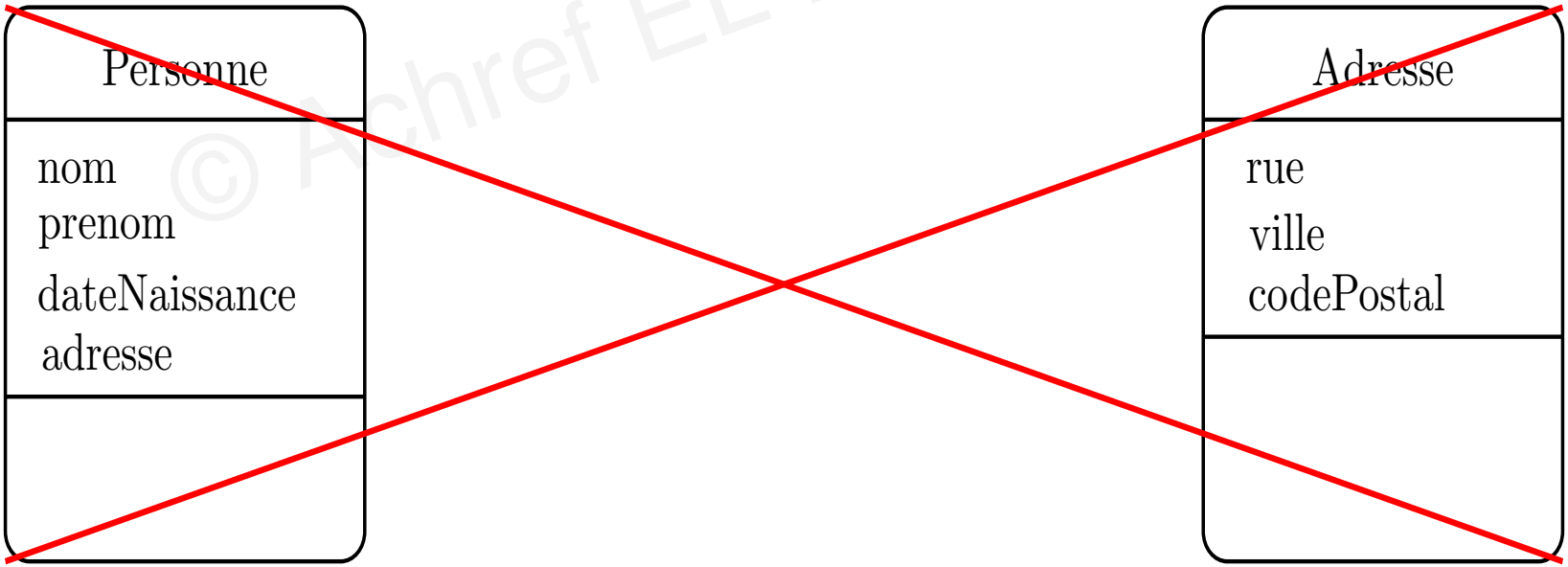




UML

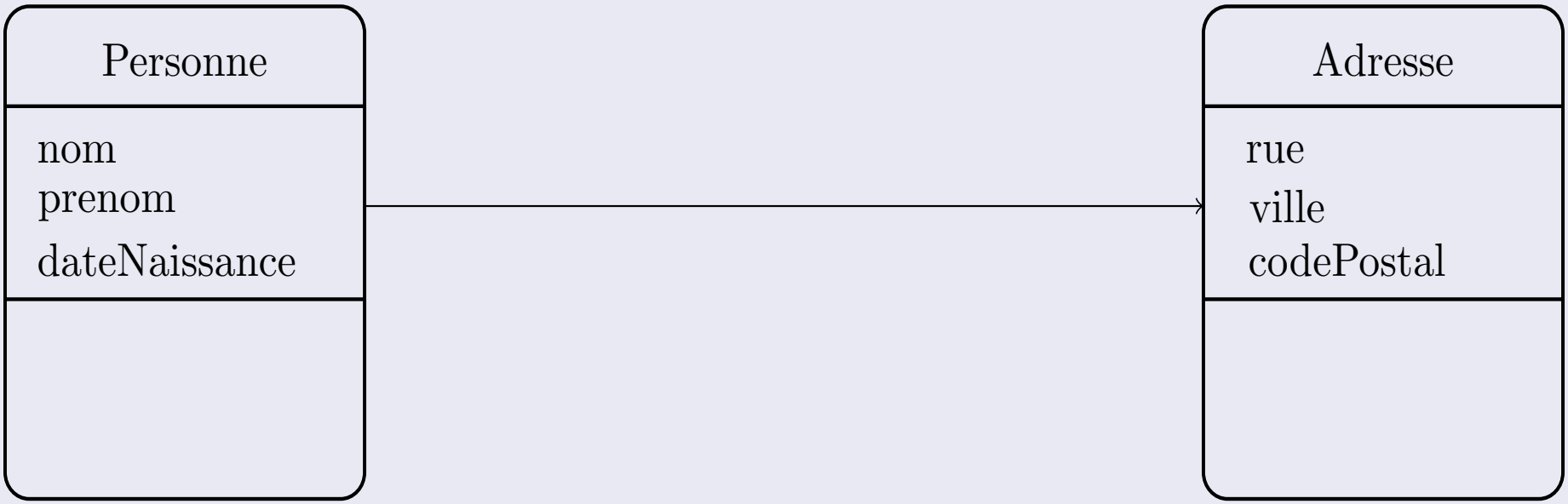
Comment schématiser cela en **UML** ?

En ajoutant dans la classe `Personne` un attribut `adresse` de type `Adresse` ?



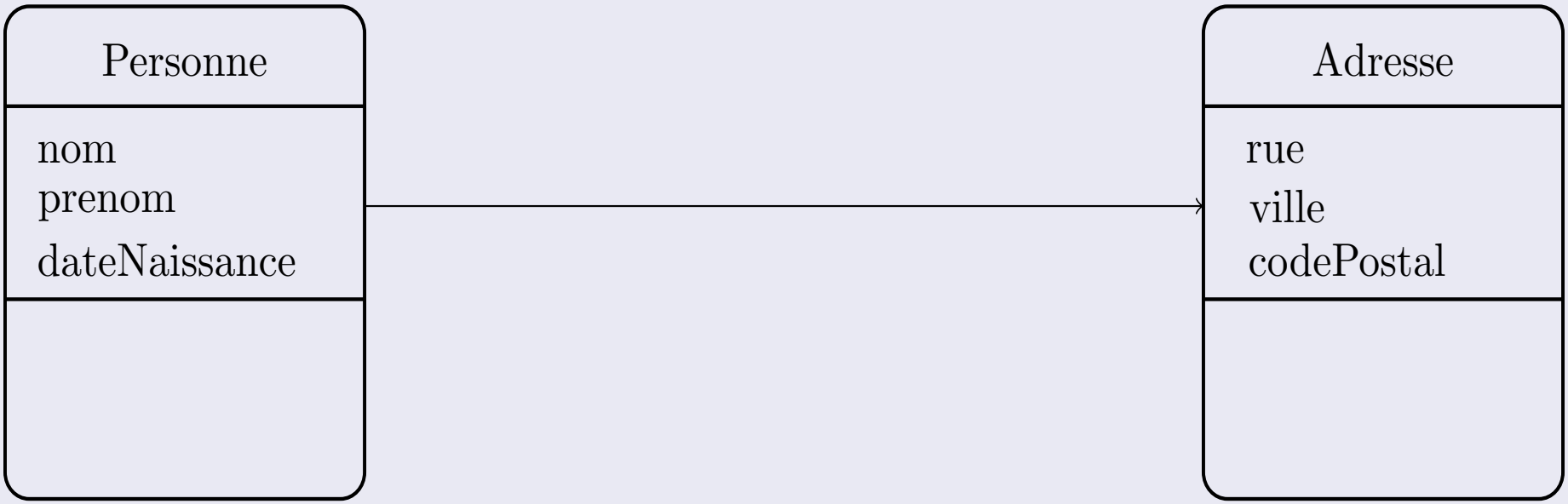
UML

En reliant les deux classes (sommets d'un graphe) par une arête



UML

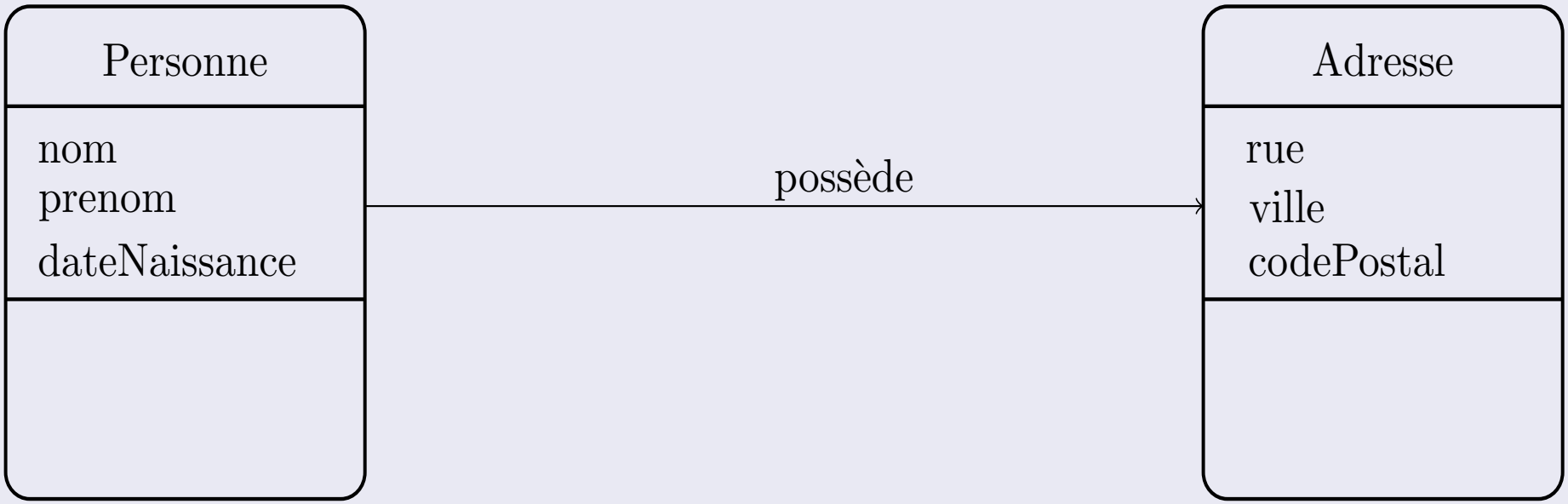
En reliant les deux classes (sommets d'un graphe) par une arête



Relation entre classes = association

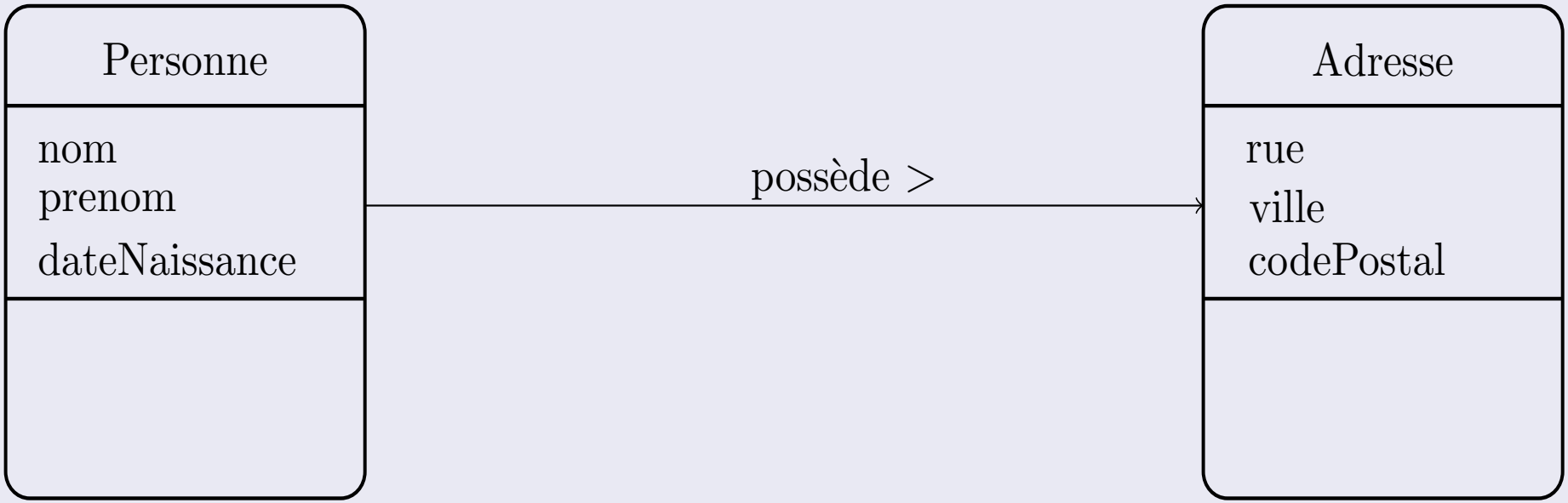
UML

L'association peut être nommée



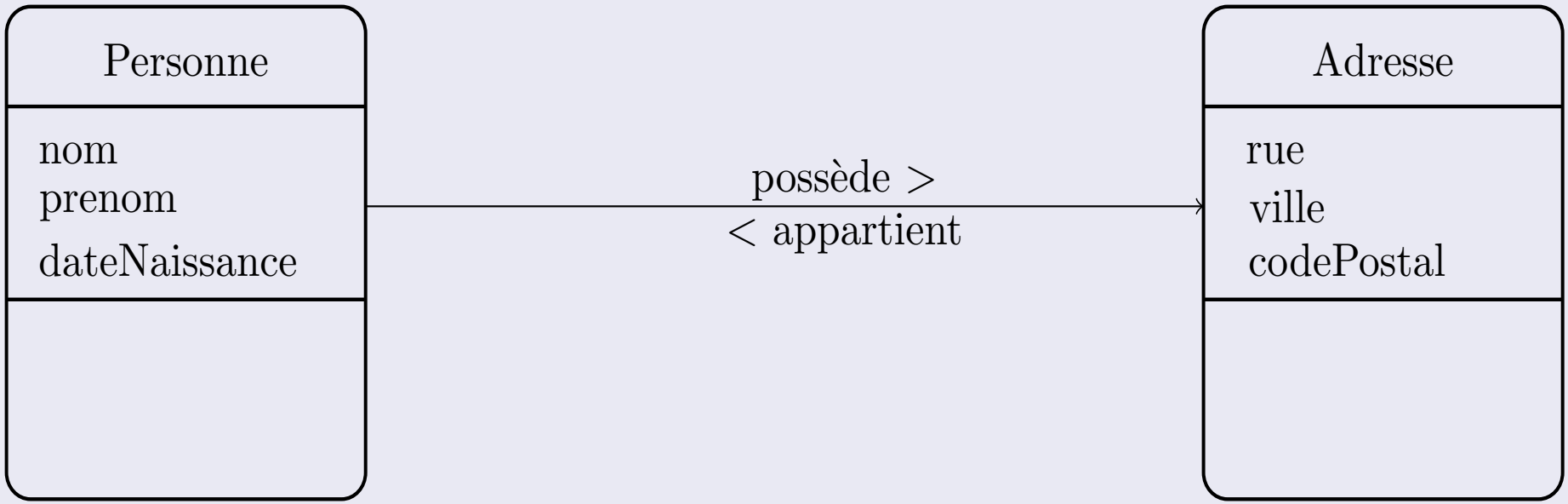
UML

On peut aussi indiquer le sens de la lecture



UML

On peut aussi ajouter deux noms à l'association : un pour chaque sens de lecture



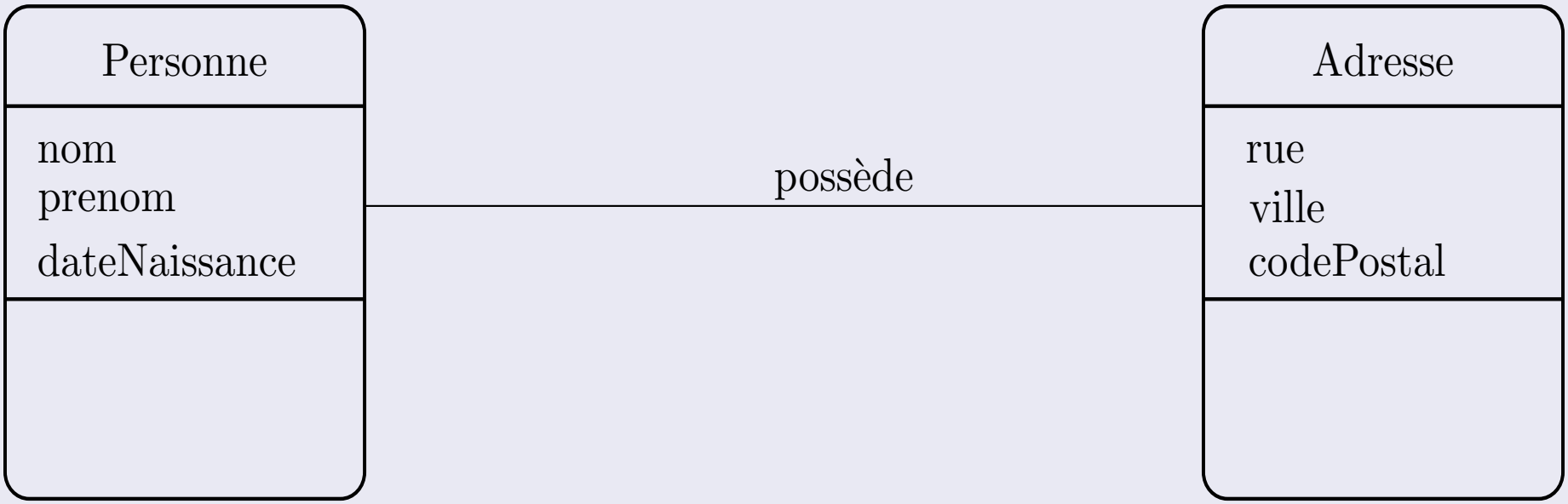
UML

Comment faire pour rendre la relation bidirectionnelle ?

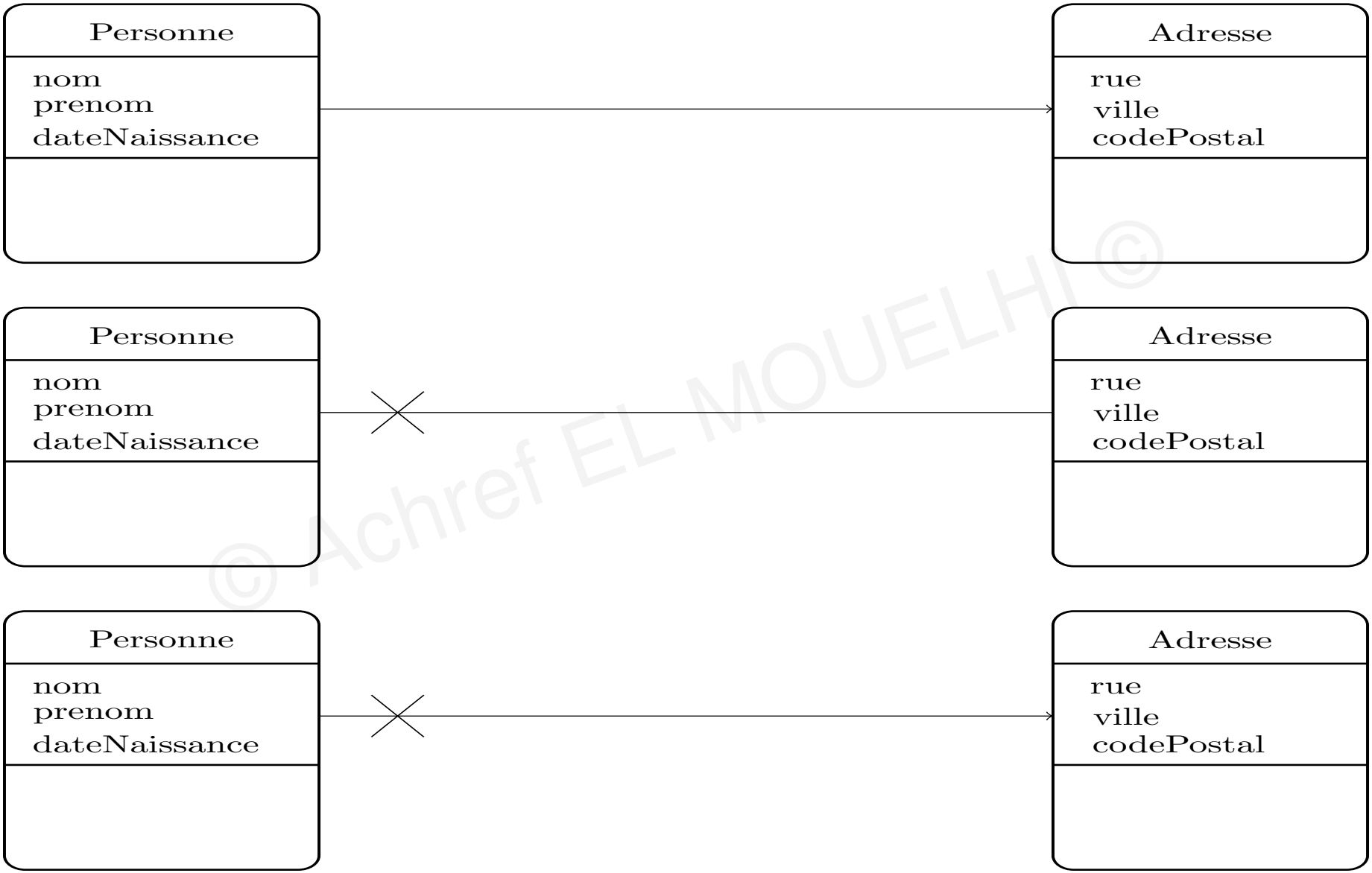
- Pour une personne, on peut connaître son adresse
- Et pour une adresse, on peut connaître son propriétaire

UML

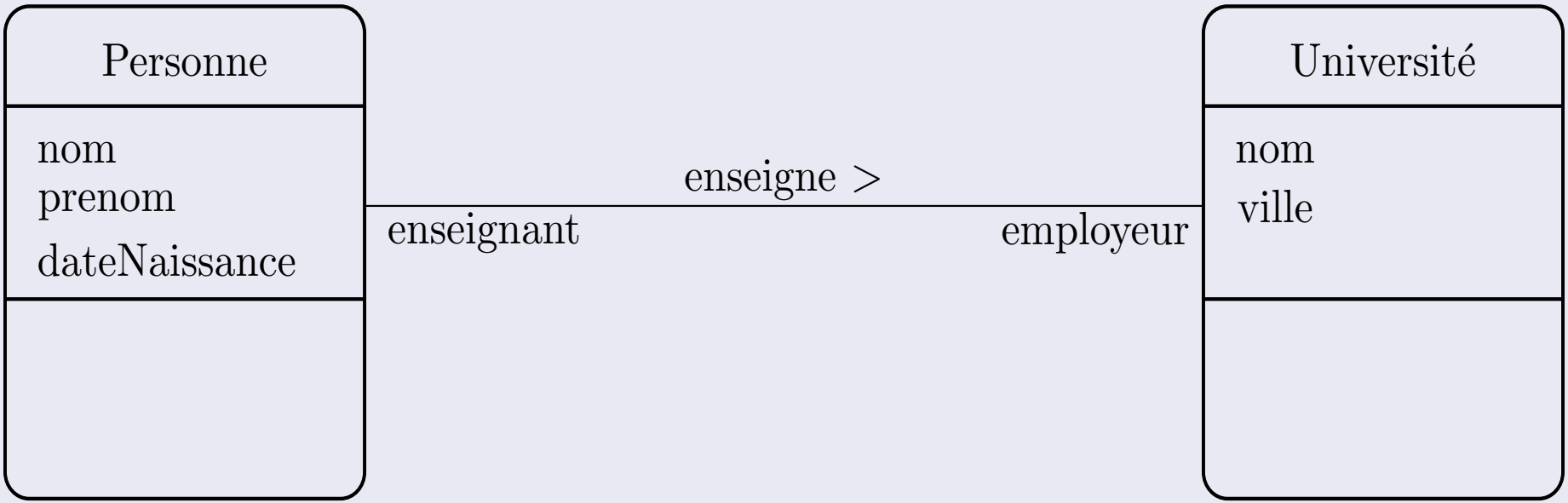
L'association est maintenant bidirectionnelle



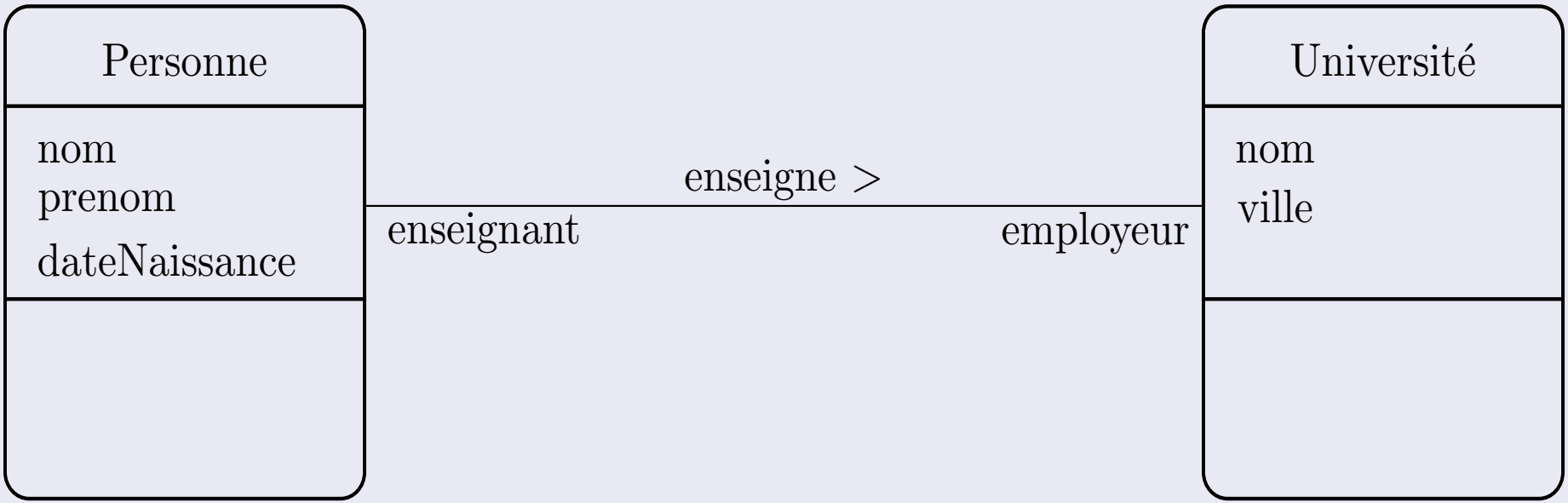
Les trois associations suivantes sont unidirectionnelles et équivalentes



On peut définir des rôles



On peut définir des rôles



Explication

- Le rôle d'une `Personne` dans une `Université` est `enseignant`
- Le rôle d'une `Université` pour une `Personne` est `employeur`

On peut définir une ou plusieurs associations entre deux classes

On peut définir une ou plusieurs associations entre deux classes

Deux associations possibles entre `Employé` et `Entreprise`

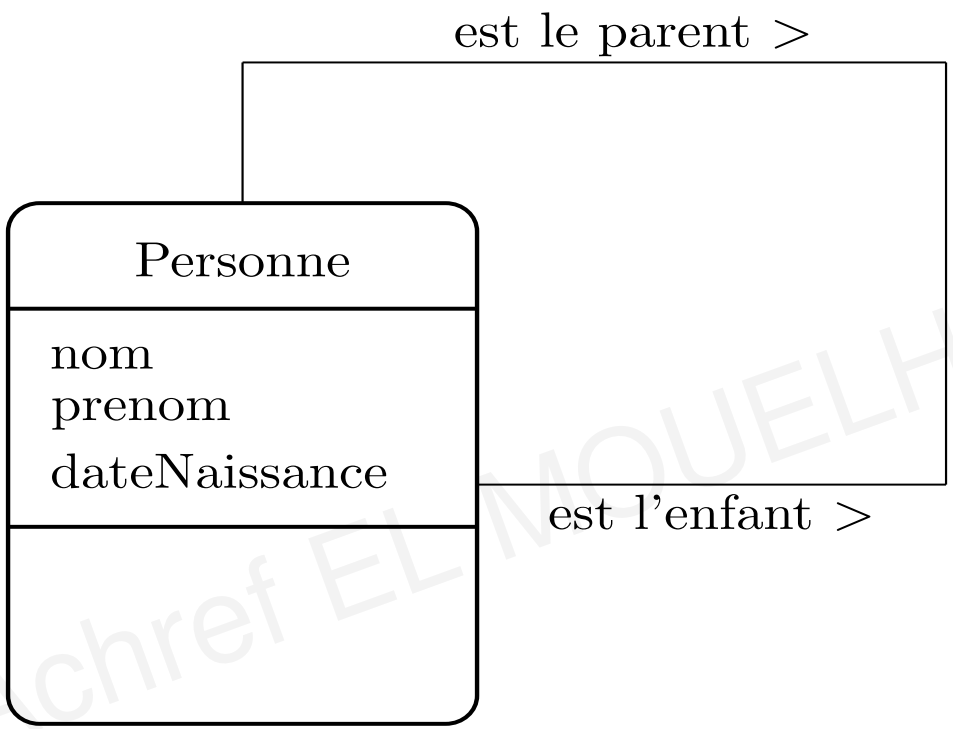
- Un `Employé` travaille dans une `Entreprise`
- Un `Employé` peut diriger une `Entreprise`

UML

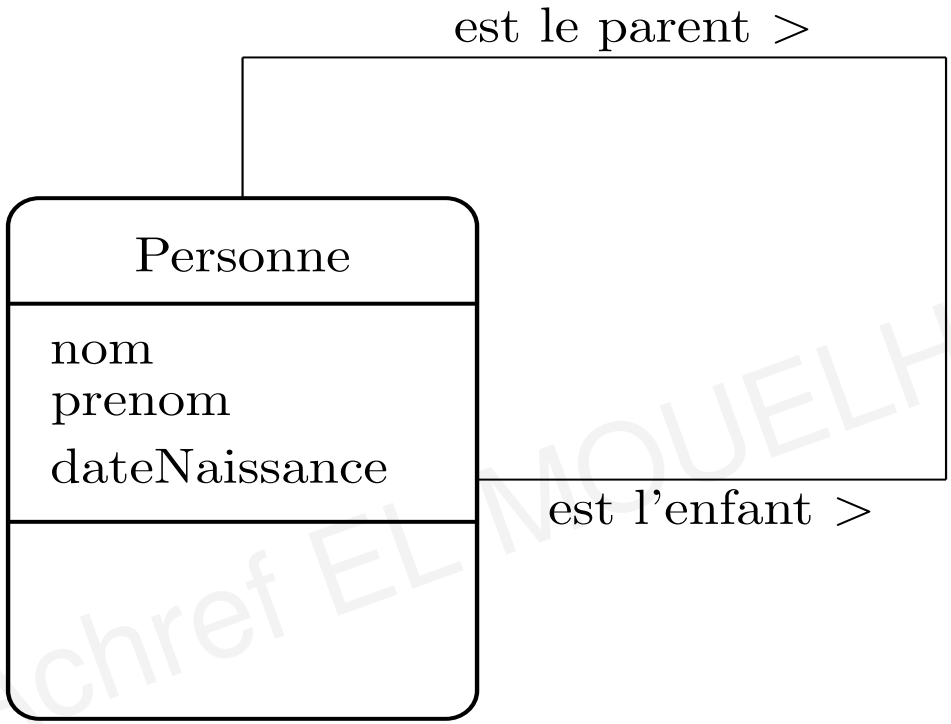
Remarque

Une association peut concerner une seule classe

Exemple d'auto-association



Exemple d'auto-association



Comment lire ça ?

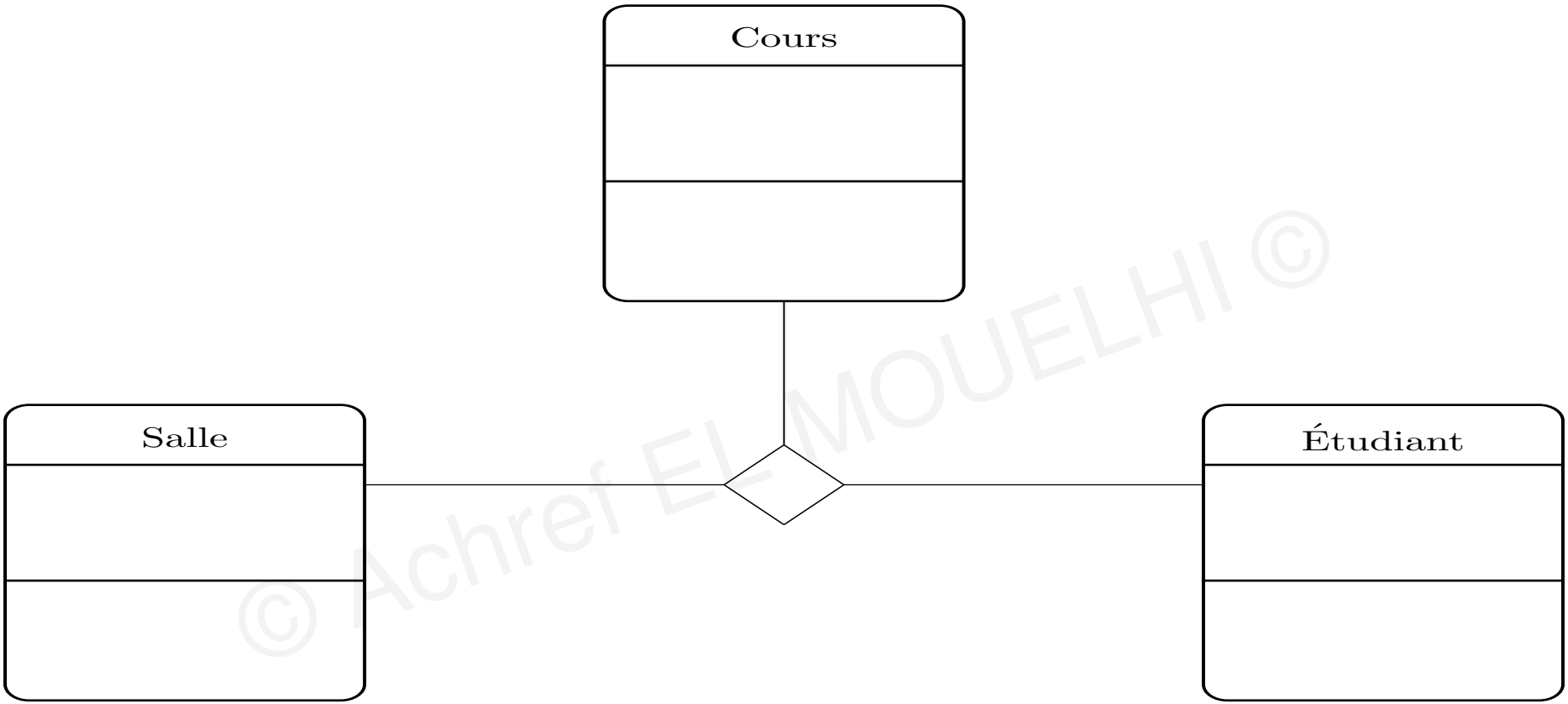
- Une personne est l'enfant d'une autre personne
- Une personne est le parent d'une autre personne

UML

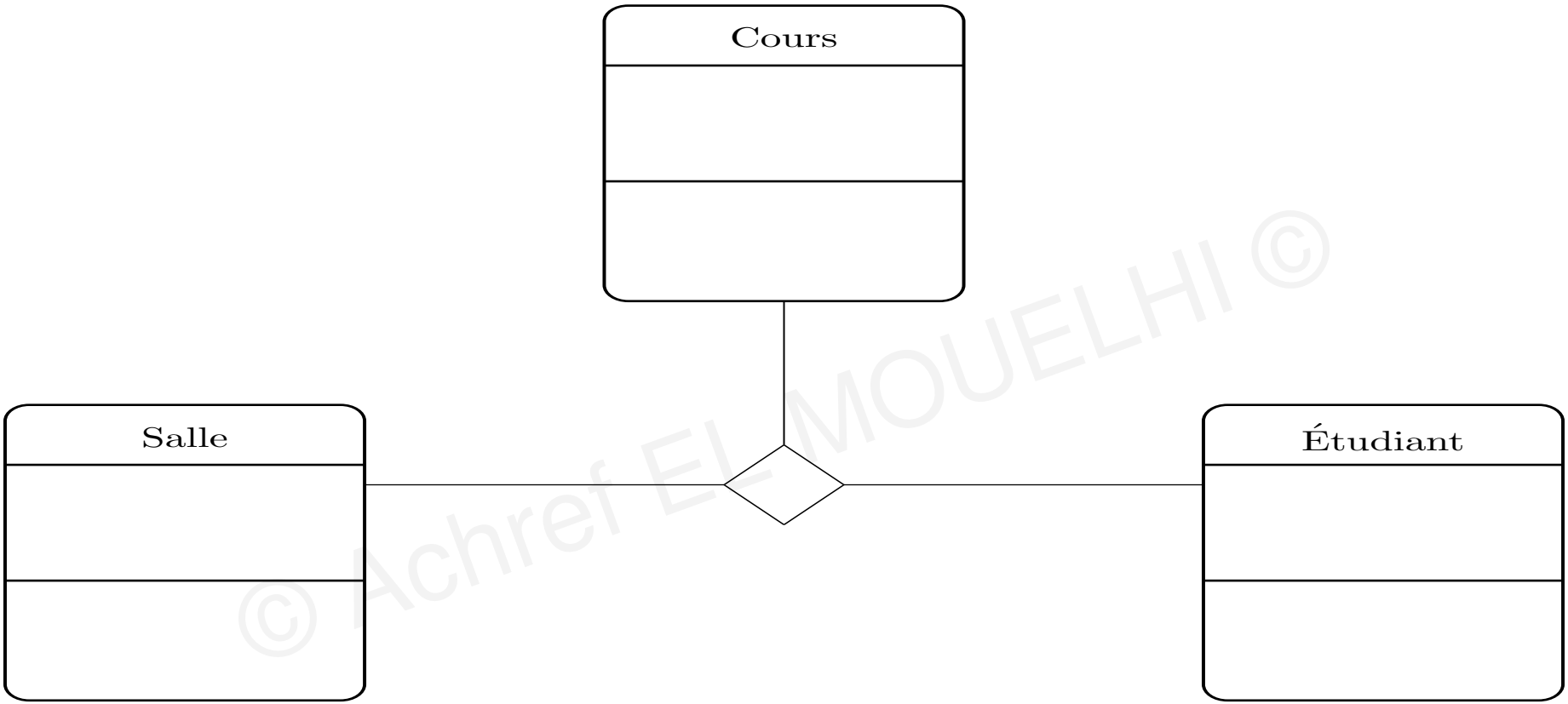
Remarque

- Toutes les associations, qu'on a vu, sont binaires
- C'est-à-dire, elles relient au plus deux classes
- On peut aussi utiliser des associations ternaires voire n-aires

Exemple d'association ternaire



Exemple d'association ternaire



Toute association non-binaire peut être transformée en un ensemble d'associations binaires.

UML

Quatre associations particulières

- héritage
- agrégation
- composition
- dépendance

UML

L'héritage, quand ?

- Lorsque deux ou plusieurs classes partagent plusieurs attributs (et méthodes)
- Lorsqu'une `Classe1` est **une sorte de** `Classe2`

UML

Exemple

- Un enseignant a un nom, un prénom, une date de naissance, un salaire et une date de recrutement

UML

Exemple

- Un enseignant a un nom, un prénom, une date de naissance, un salaire et une date de recrutement
- Un étudiant a aussi un nom, un prénom, une date de naissance et un niveau

UML

Exemple

- Un enseignant a un nom, un prénom, une date de naissance, un salaire et une date de recrutement
- Un étudiant a aussi un nom, un prénom, une date de naissance et un niveau
- Sémantiquement, enseignant et étudiant sont une sorte de personne

UML

Exemple

- Un enseignant a un nom, un prénom, une date de naissance, un salaire et une date de recrutement
- Un étudiant a aussi un nom, un prénom, une date de naissance et un niveau
- Sémantiquement, enseignant et étudiant sont une sorte de personne
- En plus, les deux partagent plusieurs attributs tels que nom, prénom et date de naissance

UML

Exemple

- Un enseignant a un nom, un prénom, une date de naissance, un salaire et une date de recrutement
- Un étudiant a aussi un nom, un prénom, une date de naissance et un niveau
- Sémantiquement, enseignant et étudiant sont une sorte de personne
- En plus, les deux partagent plusieurs attributs tels que nom, prénom et date de naissance
- Donc, on peut mettre en commun les attributs nom, prénom, date de naissance dans une classe `Personne`

UML

Exemple

- Un enseignant a un nom, un prénom, une date de naissance, un salaire et une date de recrutement
- Un étudiant a aussi un nom, un prénom, une date de naissance et un niveau
- Sémantiquement, enseignant et étudiant sont une sorte de personne
- En plus, les deux partagent plusieurs attributs tels que nom, prénom et date de naissance
- Donc, on peut mettre en commun les attributs nom, prénom, date de naissance dans une classe `Personne`
- Les classes `Étudiant` et `Enseignant` hériteront de la classe `Personne`

UML

L'héritage, pourquoi ?

Pour :

- réutiliser le code
- éviter la duplication de constituants (attributs, méthodes)

UML

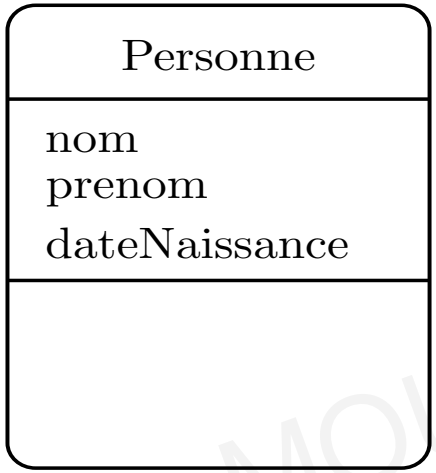
L'héritage, pourquoi ?

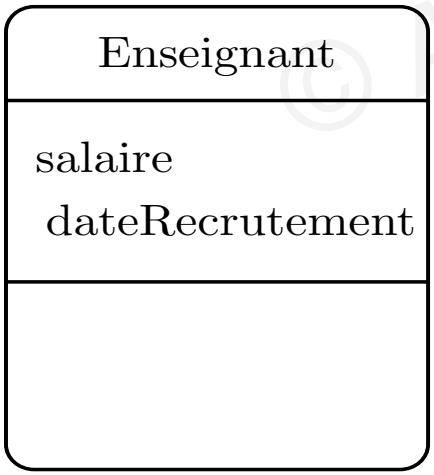
Pour :

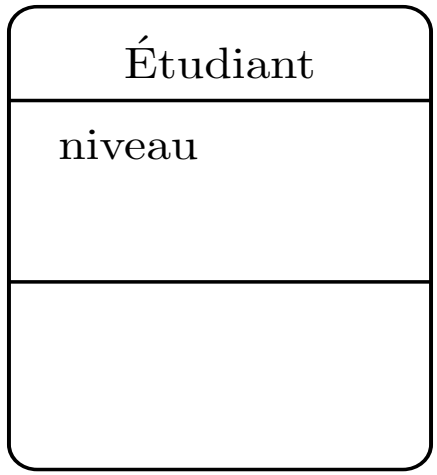
- réutiliser le code
- éviter la duplication de constituants (attributs, méthodes)

Comment faire l'héritage en **UML** ?

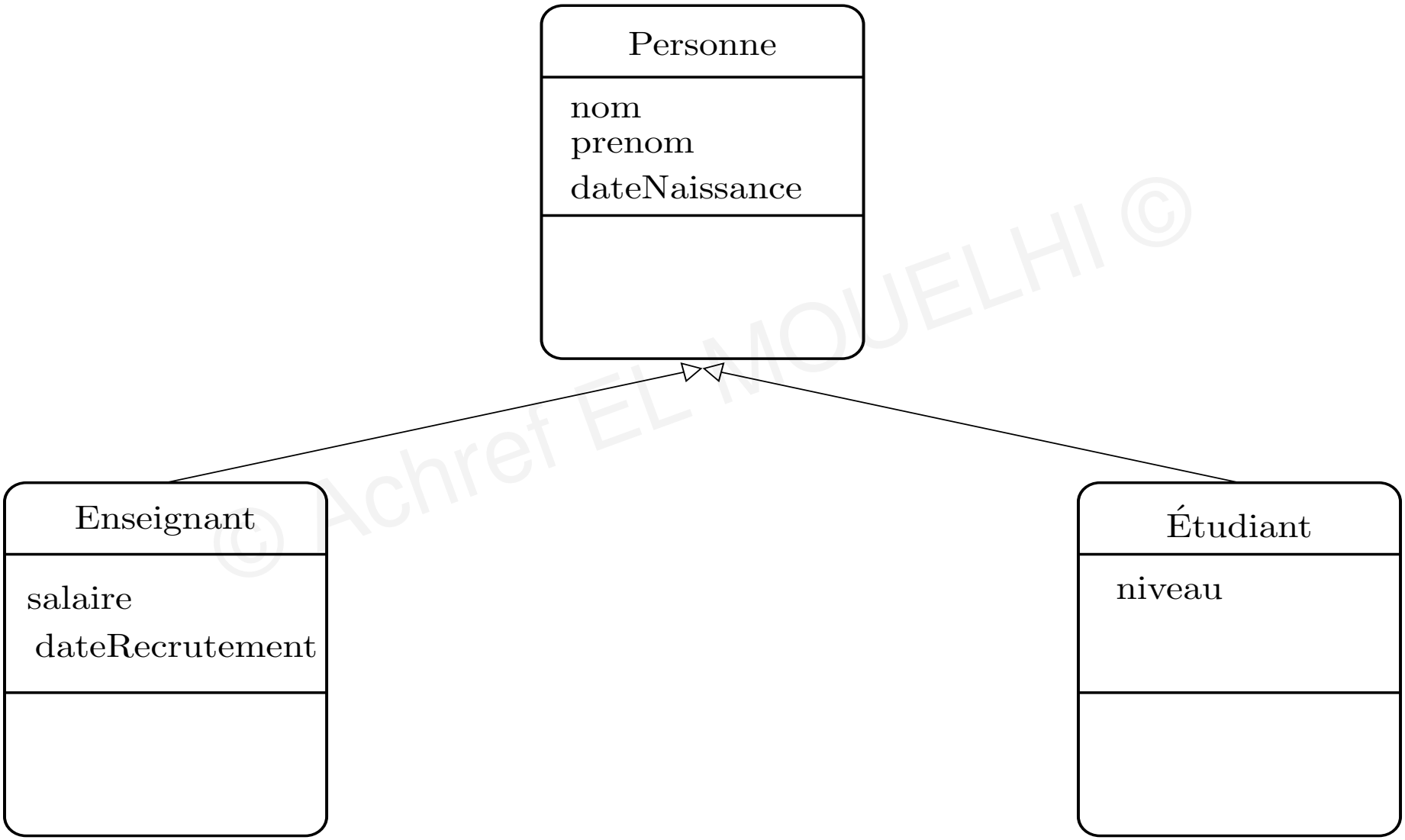
Étape 1 : Mettre les propriétés en commun (`nom`, `prenom` et `dateNaissance`) dans une nouvelle classe `Personne`







Étape 2 : Relier par une flèche les deux classes `Étudiant` et `Enseignant` à la classe `Personne`



UML

Terminologie

- La classe `Personne` est appelée : classe mère, super classe, classe parente, classe racine...
- Les classes `Étudiant` et `Enseignant` sont appelées : classes filles, sous-classes, classes dérivées...
- On dit aussi que les classes `Étudiant` et `Enseignant` **héritent** de la classe `Personne`

UML

L'héritage, pourquoi ?

- Représenter l'ordre naturel des classes (hiérarchie)
- Organiser sémantiquement et symboliquement les classes
- Spécifier ou généraliser une classe

UML

L'héritage, pourquoi ?

- Représenter l'ordre naturel des classes (hiérarchie)
- Organiser sémantiquement et symboliquement les classes
- Spécifier ou généraliser une classe

Qu'est ce que la généralisation ?

ajouter une classe au dessus d'une autre

UML

L'héritage, pourquoi ?

- Représenter l'ordre naturel des classes (hiérarchie)
- Organiser sémantiquement et symboliquement les classes
- Spécifier ou généraliser une classe

Qu'est ce que la généralisation ?

ajouter une classe au dessus d'une autre

Qu'est ce que la spécialisation ?

ajouter une classe au dessous d'une autre

UML

L'héritage : propriétés

- **Transitivité** : si A hérite de B et B hérite de C, alors A hérite de C
- **Non-réflexif** : une classe n'hérite pas d'elle même
- **Non-symétrique** : si A hérite de B, alors B n'hérite pas de A
- **Non-cyclique** : si A hérite de B et B hérite de C, alors C ne peut hériter de A

UML

Qu'est ce que l'héritage multiple ?

le fait d'hériter de plusieurs classes au même temps

UML

Qu'est ce que l'héritage multiple ?

le fait d'hériter de plusieurs classes au même temps

L'héritage multiple, est-il possible ?

- autorisé par certains LOO comme : **OCaml**, **Eiffel**...
- interdit par certains autres : **PHP**, **Java**, **C#**, **TypeScript**...

UML

Quel problème peut poser l'héritage multiple ?

Un problème de confusion (nommage) : si une classe A hérite de deux classes B et C qui ont un attribut ou une méthode en commun

UML

Quel problème peut poser l'héritage multiple ?

Un problème de confusion (nommage) : si une classe A hérite de deux classes B et C qui ont un attribut ou une méthode en commun

Les LOO qui interdisent l'héritage multiple proposent soit

- les interfaces (**PHP**, **Java**, **C#**, **TypeScript**...)
- les traits (**PHP**)

UML

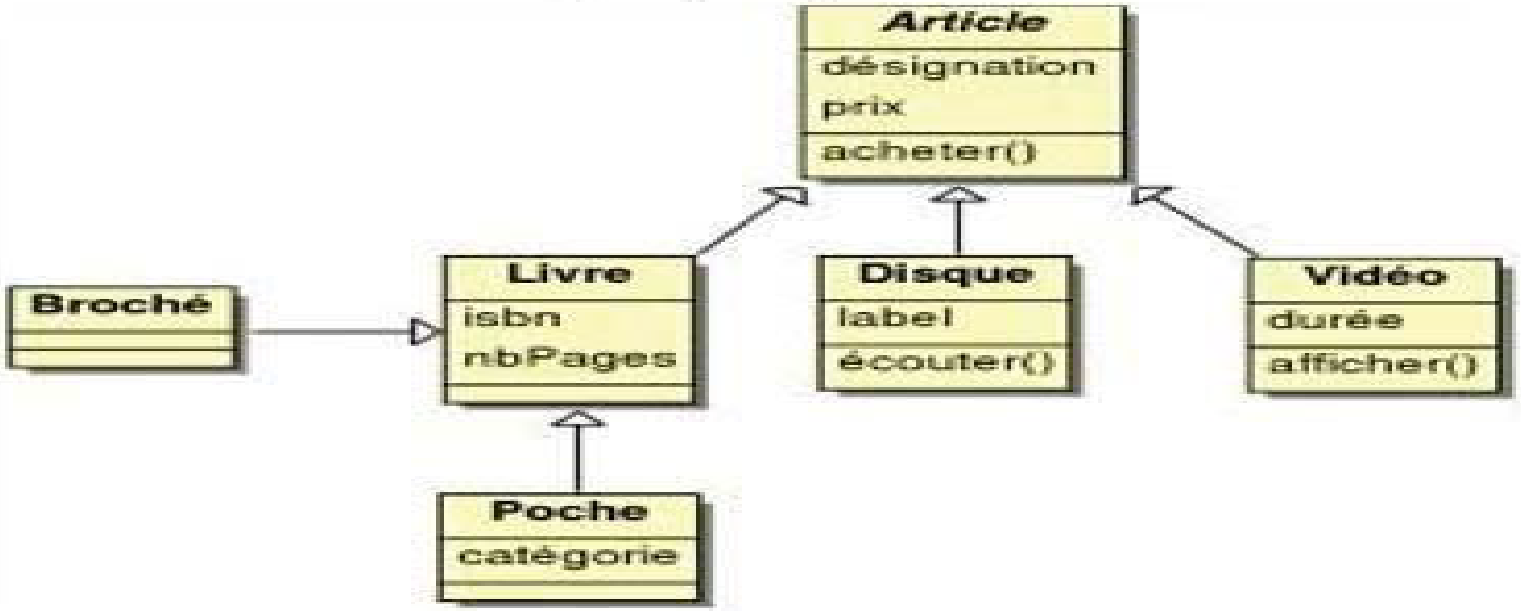
Exercice

- Une classe mère peut-elle accéder aux attributs de sa classe fille ?
- Une classe mère peut-elle avoir plusieurs classes filles ?
- Deux classes héritant la même classe peuvent-elles avoir un attribut qui n'est pas dans la classe mère ?
- Une classe fille peut-elle avoir comme uniques attributs et méthodes ceux de la classe mère ?

UML

Remarque

- Dans certains LOO comme **Java**, **C#**..., toutes les classes héritent d'une classe racine appelée `Object`
- `Object` offre plusieurs services tels que la conversion d'un objet en chaîne de caractères, le clonage...
- `Object` : cette classe racine n'existe pas en **PHP**...



Questions

- Existera t-il une méthode (autre que le constructeur) dans un objet de type `Poche` ?
- La classe `Broché` n'a pas d'attribut propre, pourrait-elle être implémentée ?
- Pourrait-on ajouter une relation d'héritage entre `Poche` et `Article` ?
- Un livre peut-il être `Broché` et de `Poche` ?

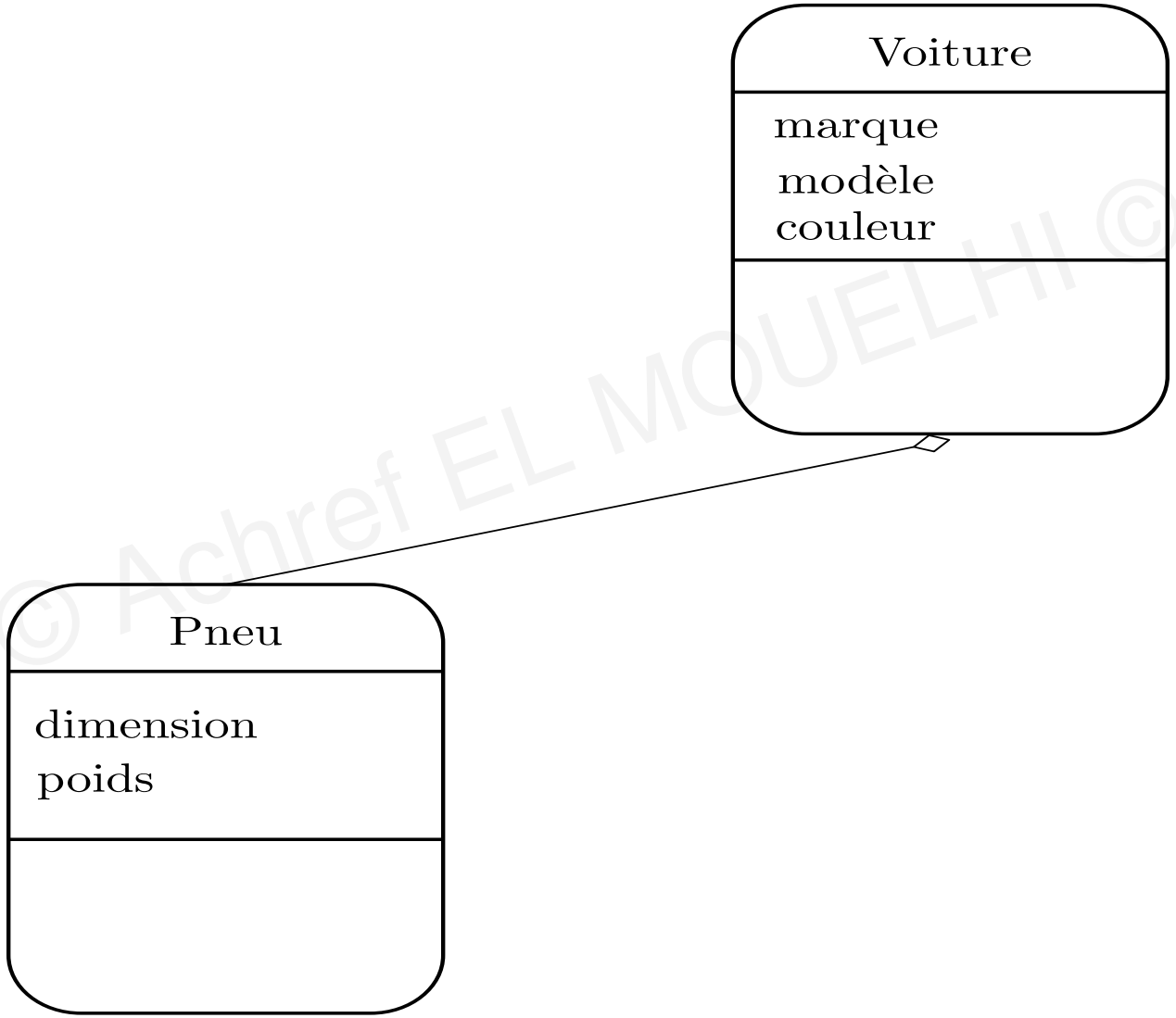
UML

L'agrégation

- C'est une association non-symétrique
- Modélisée par un losange vide coté agrégat
- Elle représente une relation de type ensemble/élément

UML

Exemple d'agrégation



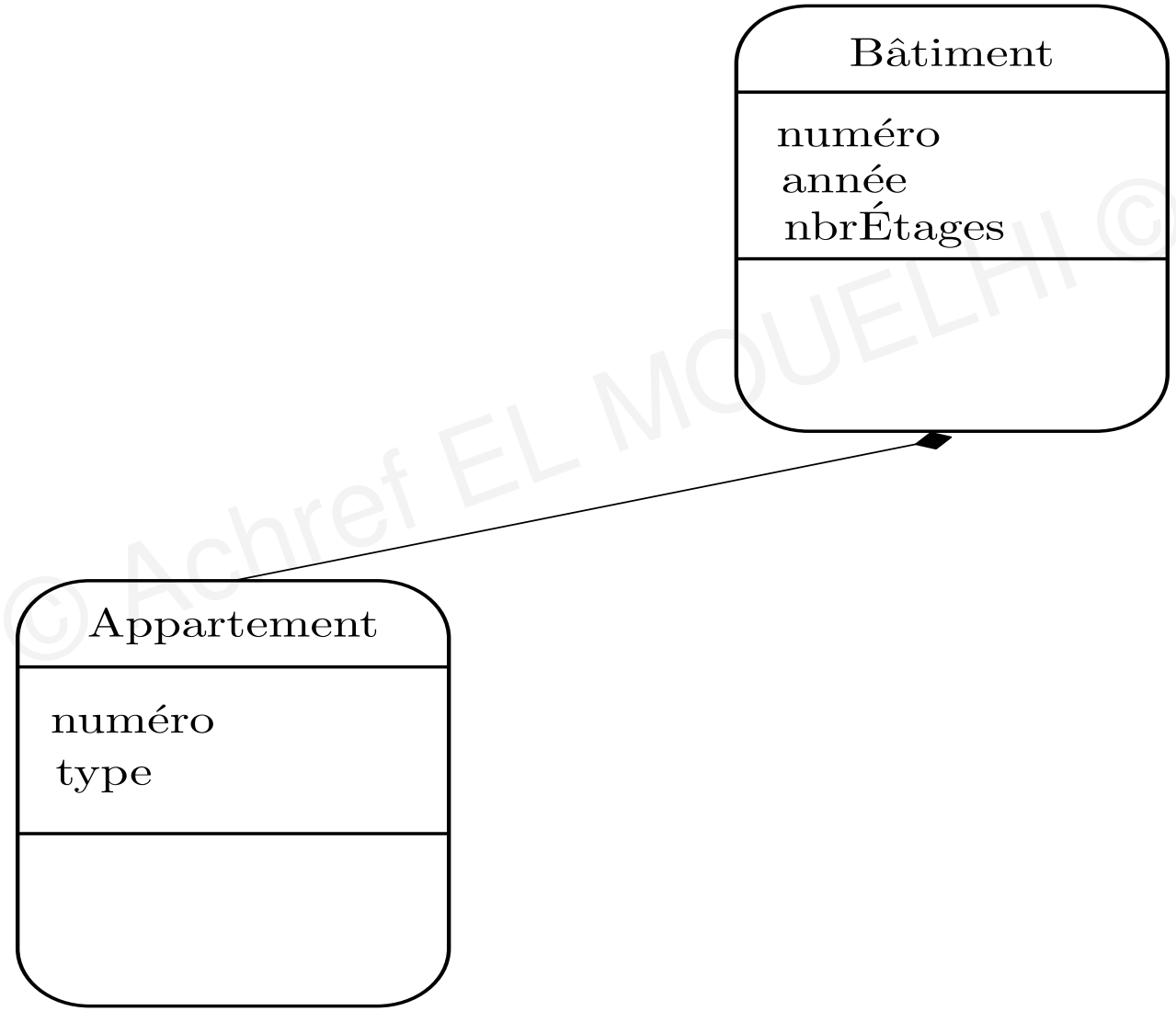
UML

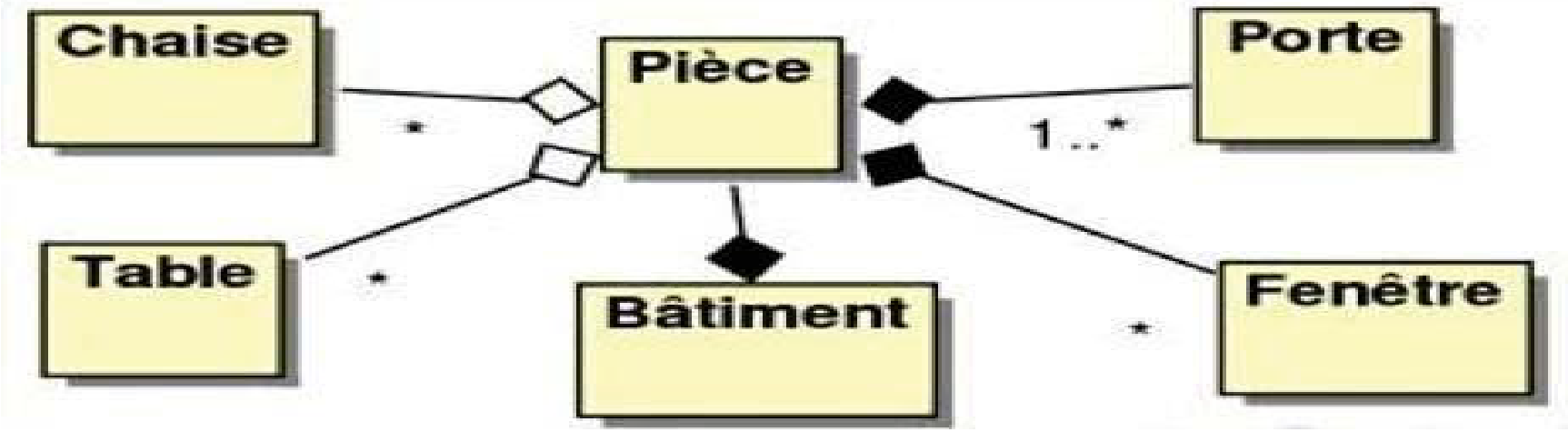
La composition

- C'est une agrégation forte modélisée par un losange noir coté agrégat
- L'élément n'existe pas sans l'agrégat (l'élément est détruit lorsque l'agrégat n'existe plus)
- L'élément ne peut être en relation qu'avec un seul agrégat

UML

Exemple de composition





Questions

- Si une pièce est détruite, les chaises associées devraient-elles également être détruites ?
- Si une pièce est détruite, les portes associées devraient-elles également être détruites ?
- Si un bâtiment est détruit, les portes associées à ses pièces devraient-elles également être détruites ?
- Si un bâtiment est détruit, les chaises associées à ses pièces devraient-elles également être détruites ?

UML

Exercice 2

Les classes suivantes sont liées par une agrégation ou une composition ?

- but et gardien de but
- voiture et moteur
- forêt et arbre
- paragraphe et ligne
- cinéma et salle
- salle et chaise
- enseignant et cours

UML

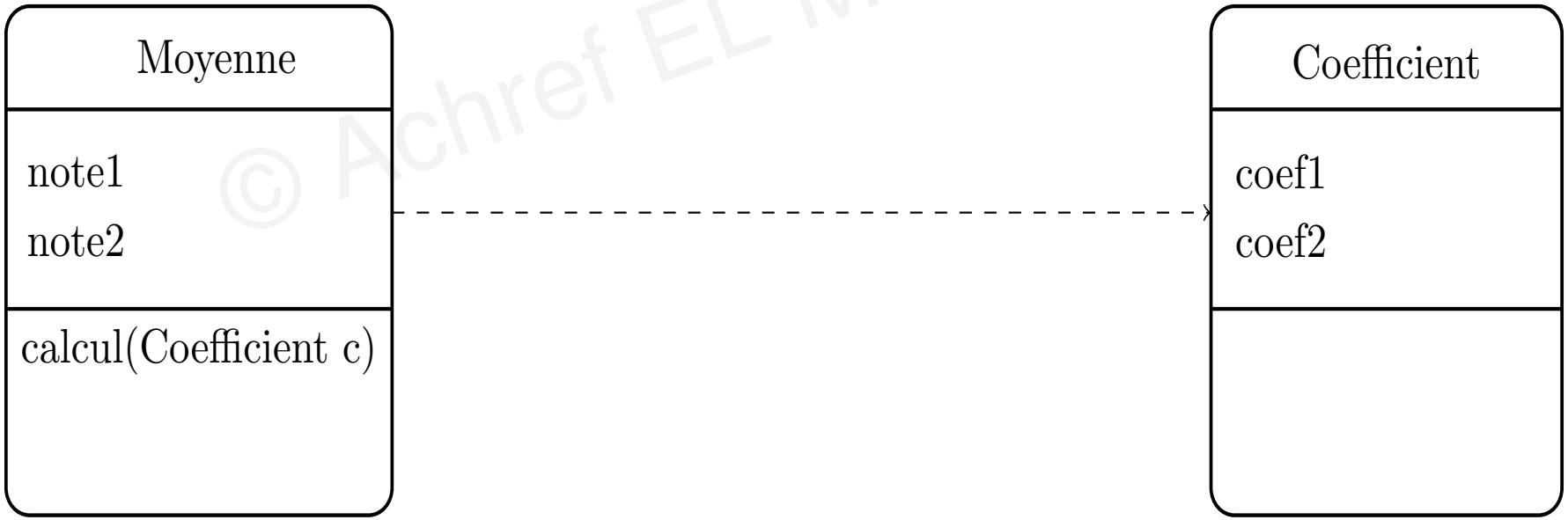
La dépendance

Une première classe utilise une deuxième sans que cette dernière soit un membre de la première

UML

La dépendance

Une première classe utilise une deuxième sans que cette dernière soit un membre de la première

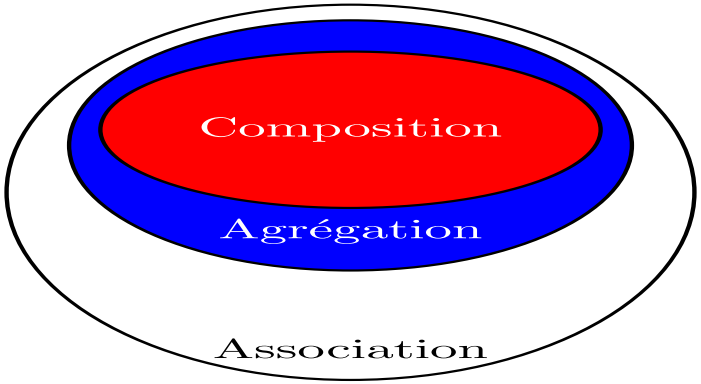


UML

Exercice

Identifier le type de relation (association, héritage, instanciation, composition, agrégation) qui relie chacun des couples suivants :

- Corps et Cœur
- Ford et ConstructeurAutomobile
- Tracteur et Véhicule
- Rectangle et Carré
- Homme et homme
- Entier et Nombre
- ChaîneDeCaractères et Caractère
- Homme et Humain



Pour récapituler

- Une association est une relation entre deux classes de même niveau conceptuel (aucune des deux n'est plus importante que l'autre) (**uses a**)
- Une agrégation est une relation ensemble/élément (**has a**)
- Une composition est une relation ensemble/élément tel que l'élément n'existe plus si l'ensemble est détruit (**owns a**)
- L'héritage est une relation entre deux classes dont une pouvant récupérer toutes les propriétés de la deuxième (**is a**)
- Une dépendance est une relation entre deux classes dont une propriété de la première fait référence la deuxième (**references a**)

UML

Définition

permet de définir le nombre minimum et maximum de relation que chaque objet de classe peut avoir avec un (ou plusieurs) objet d'une (ou plusieurs) autre classe

UML

Définition

permet de définir le nombre minimum et maximum de relation que chaque objet de classe peut avoir avec un (ou plusieurs) objet d'une (ou plusieurs) autre classe

Équivalent en MCD : cardinalité

UML

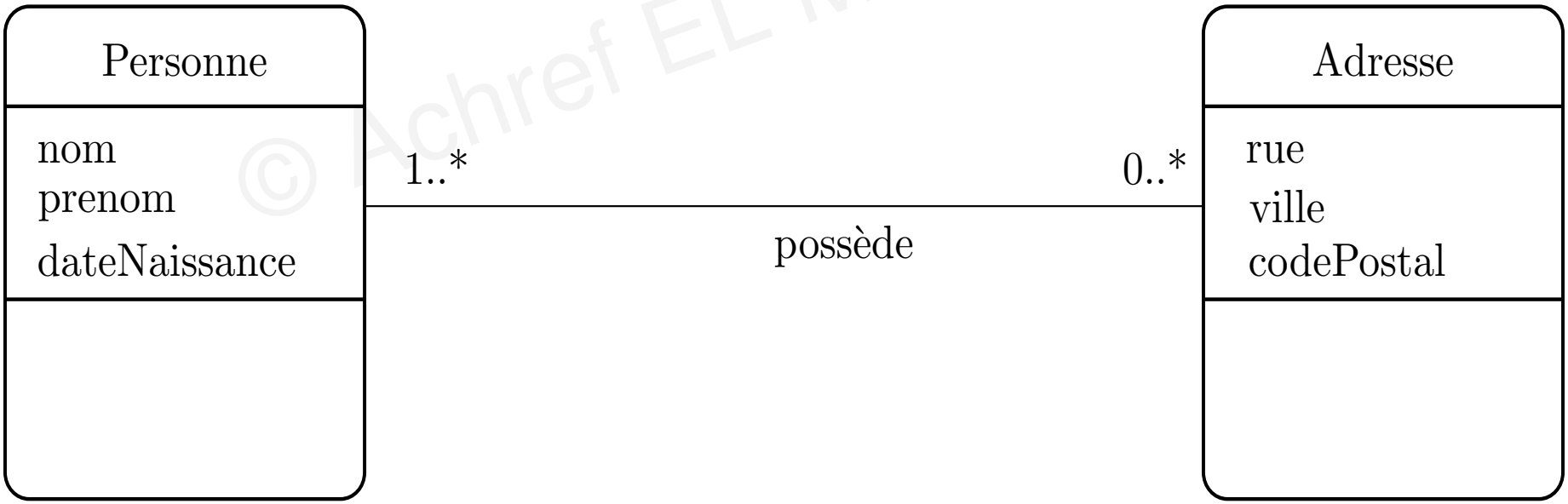
Six multiplicités possibles avec **UML**

- `0..1` : aucune ou au plus un objet
- `1` : exactement un seul objet [par défaut]
- `0..*` ou `*` : 0 ou plusieurs objets
- `1..*` : au moins un ou plusieurs objets
- $x..x$ ou x : exactement x objets
- $m..n$: Au moins m et au plus n objets

UML

Exemple

- Une personne peut avoir zéro ou plusieurs adresses
- Une adresse appartient à une ou plusieurs personnes



UML

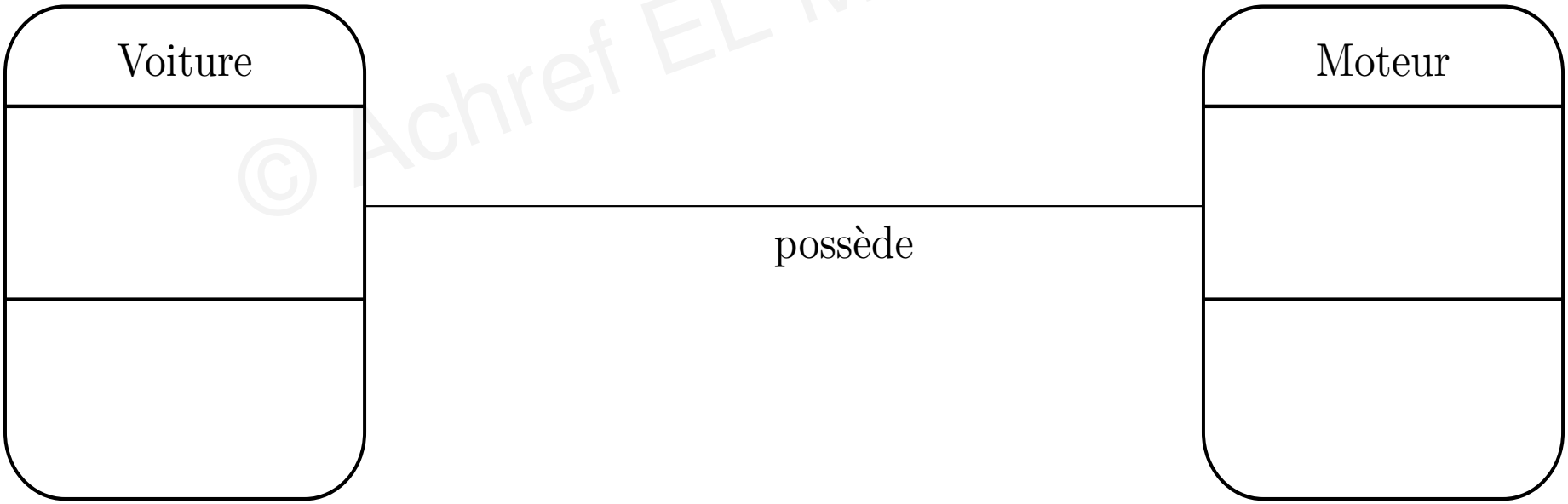
Exercice 1 : définir les multiplicités entre les deux classes suivantes

- Une voiture comporte plusieurs pneus
- Un pneu appartient à une seule voiture

UML

Exercice 2 : définir les multiplicités entre les deux classes suivantes

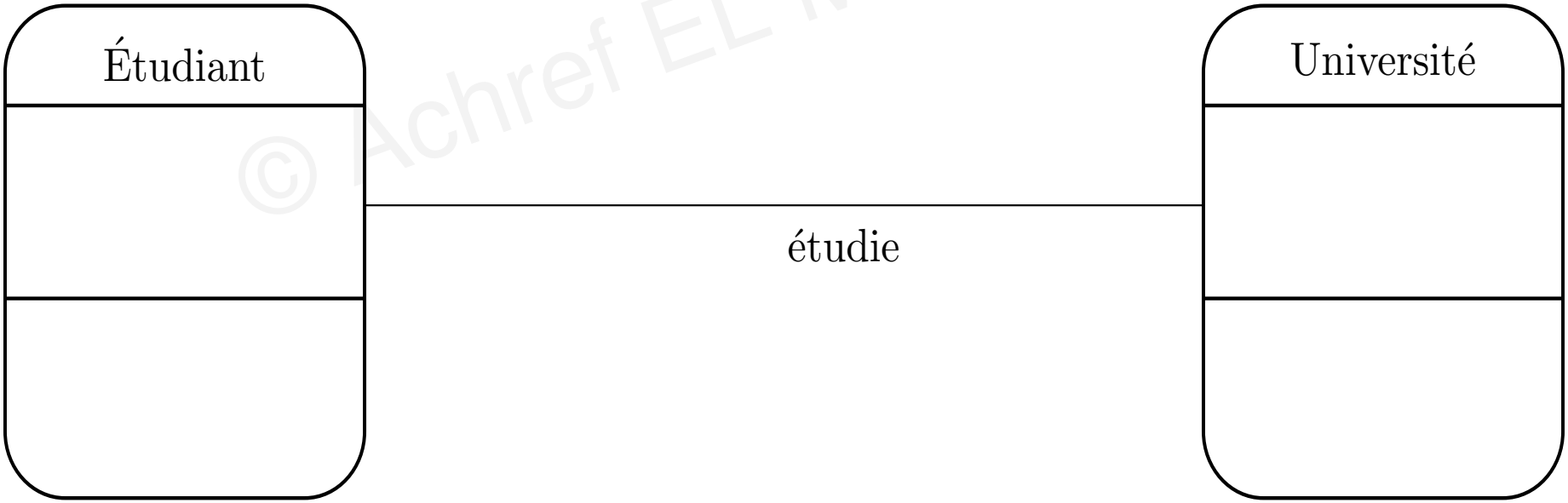
- Une voiture a un et un seul moteur
- Un moteur appartient à une seule voiture



UML

Exercice 3 : définir les multiplicités entre les deux classes suivantes

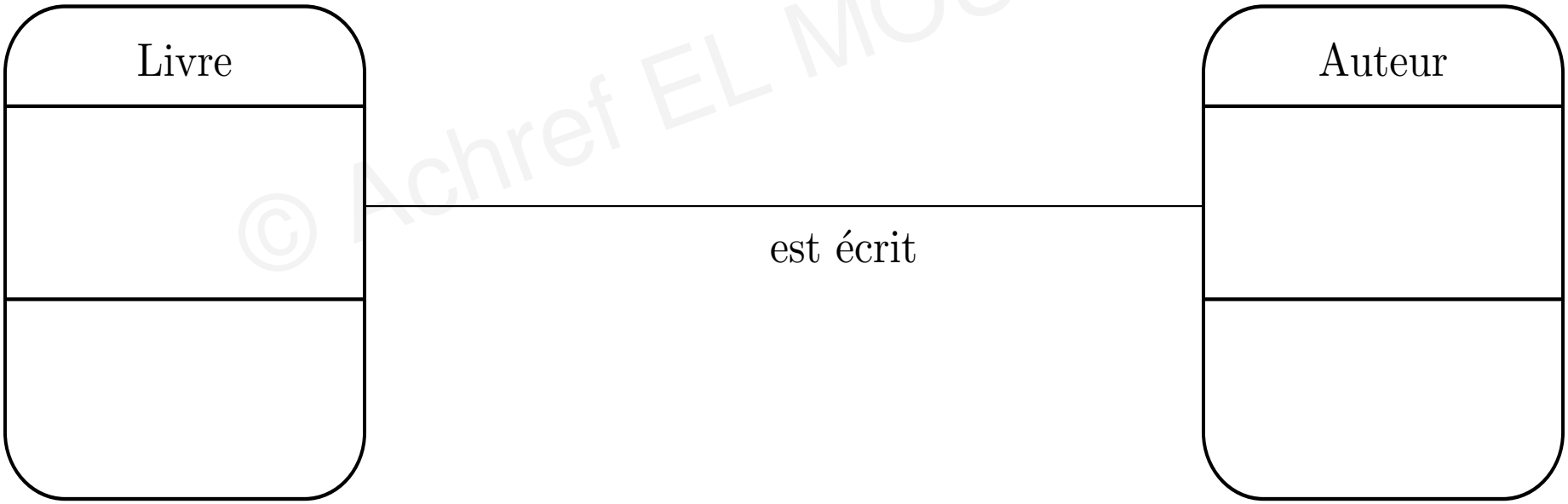
- Un étudiant est inscrit dans une et une seule université
- Une université a plusieurs étudiants



UML

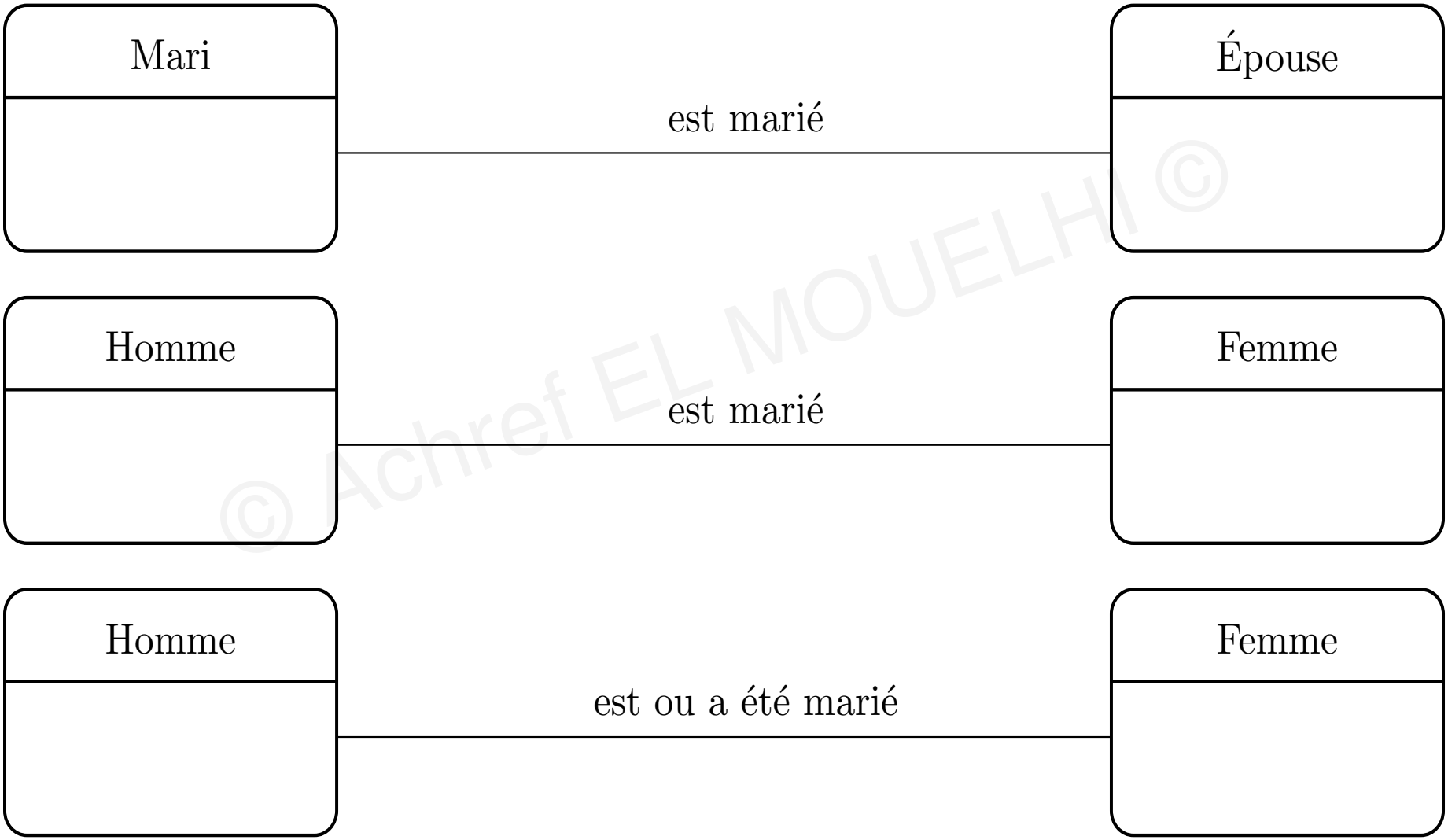
Exercice 4 : définir les multiplicités entre les deux classes suivantes

- Un livre est écrit par un ou plusieurs auteurs
- Un auteur a écrit un ou plusieurs livres



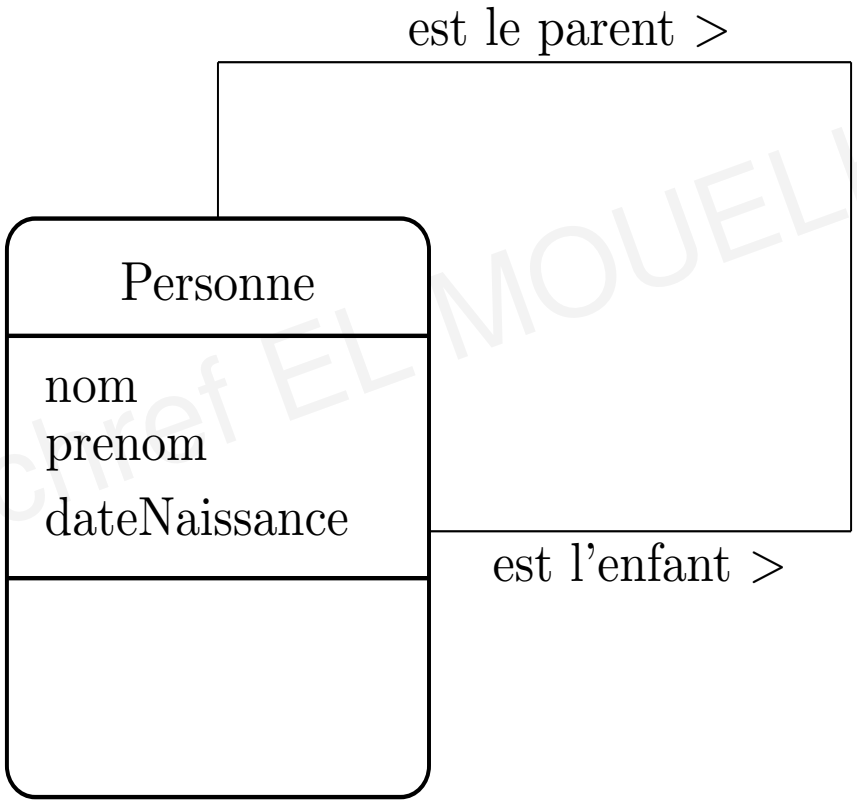
UML

Exercice 5 : déterminer les multiplicités des associations suivantes



UML

Exercice 6 : déterminer les multiplicités des associations suivantes



UML

Étant donné l'exemple suivant

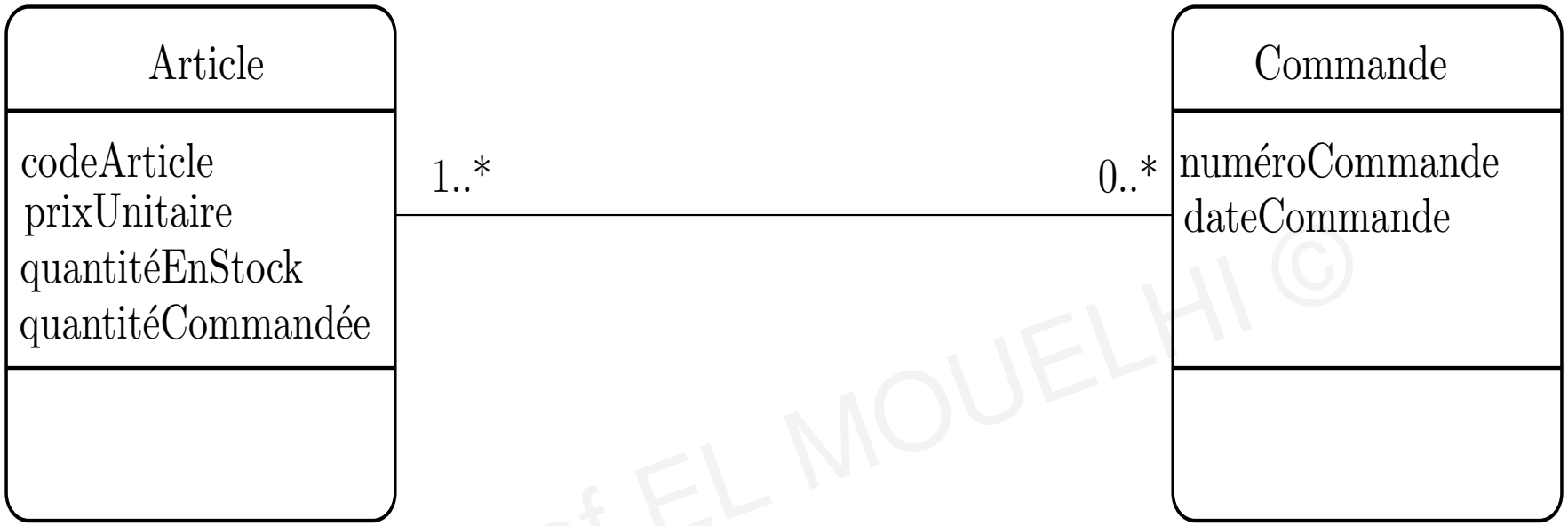
UML

Étant donné l'exemple suivant

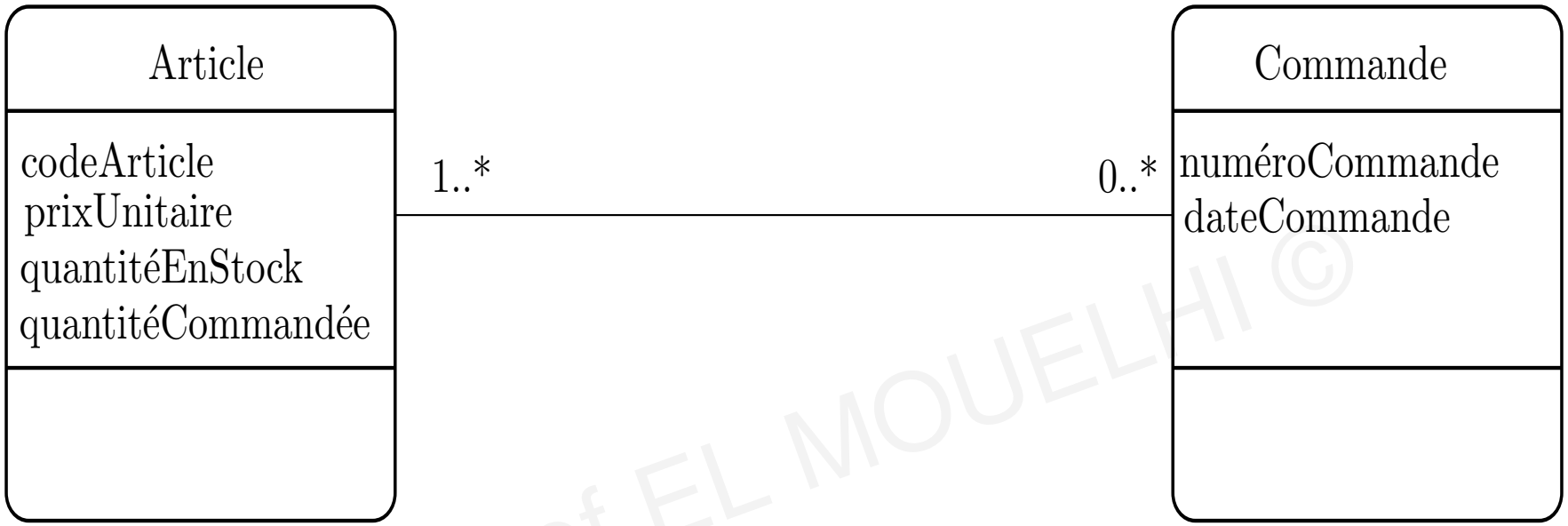
Question

Où peut-on placer la `quantitéCommandée` ?

Dans `Article` ?



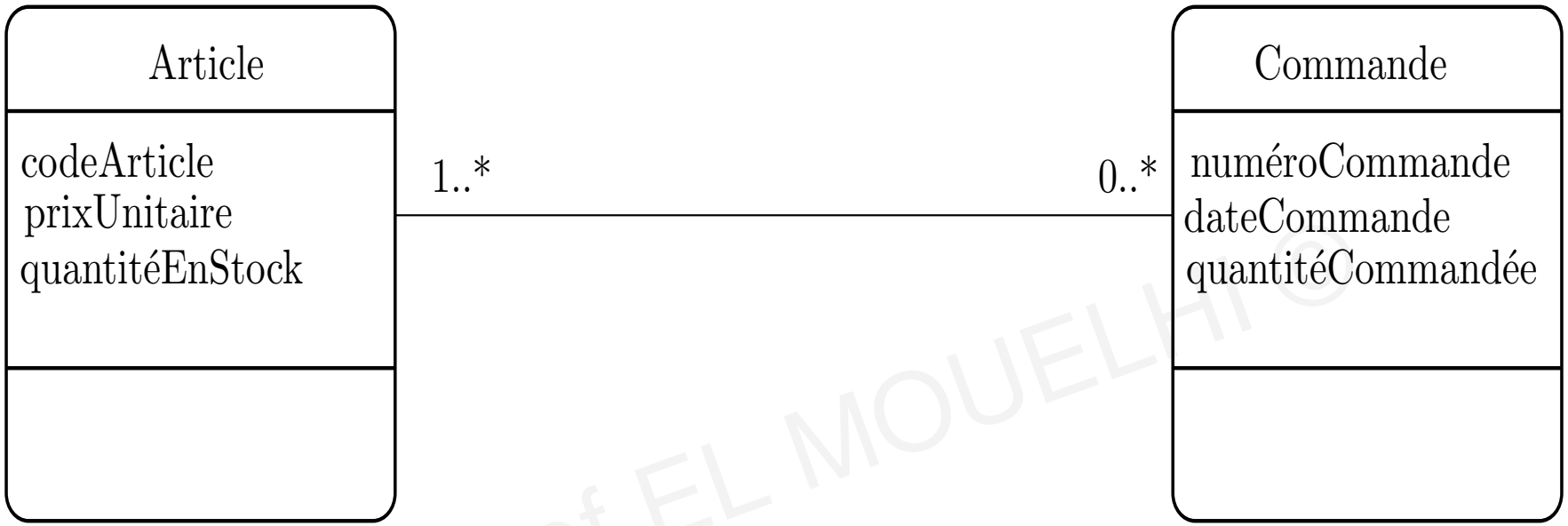
Dans `Article` ?



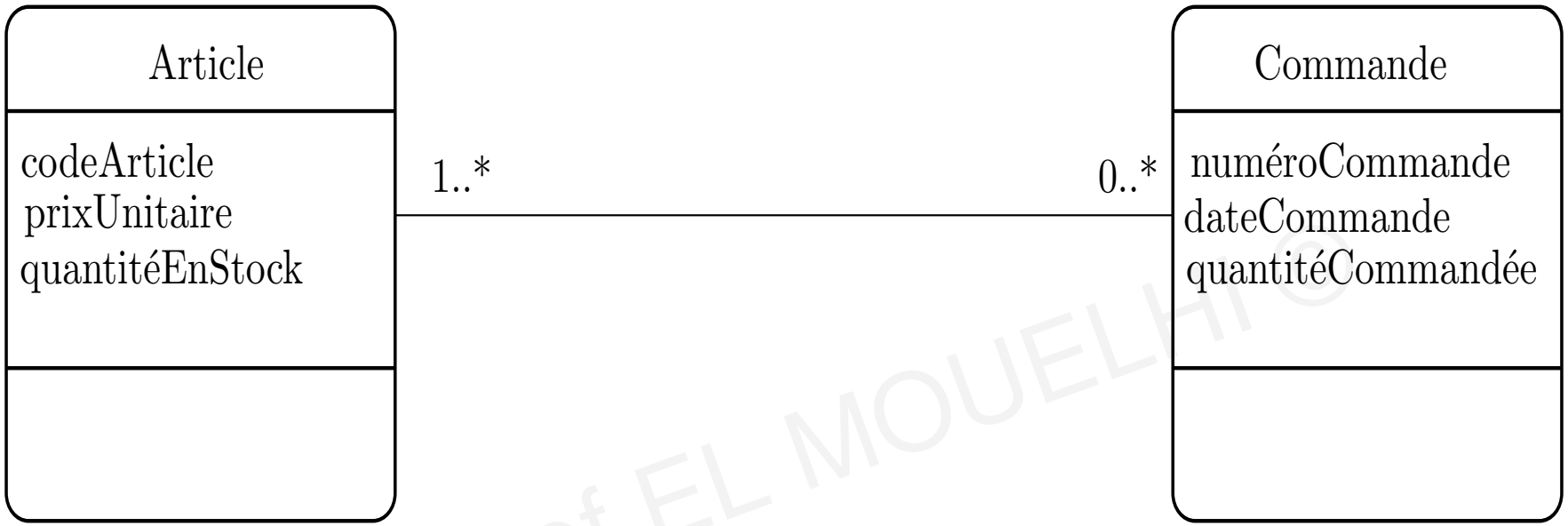
Impossible

Ainsi, une seule `quantitéCommandée` pour tous ceux qui commandent le même article.

Dans `Commande` **?**



Dans `Commande` ?



Impossible aussi

Ainsi, une seule `quantitéCommandée` pour tous les articles d'une même commande.

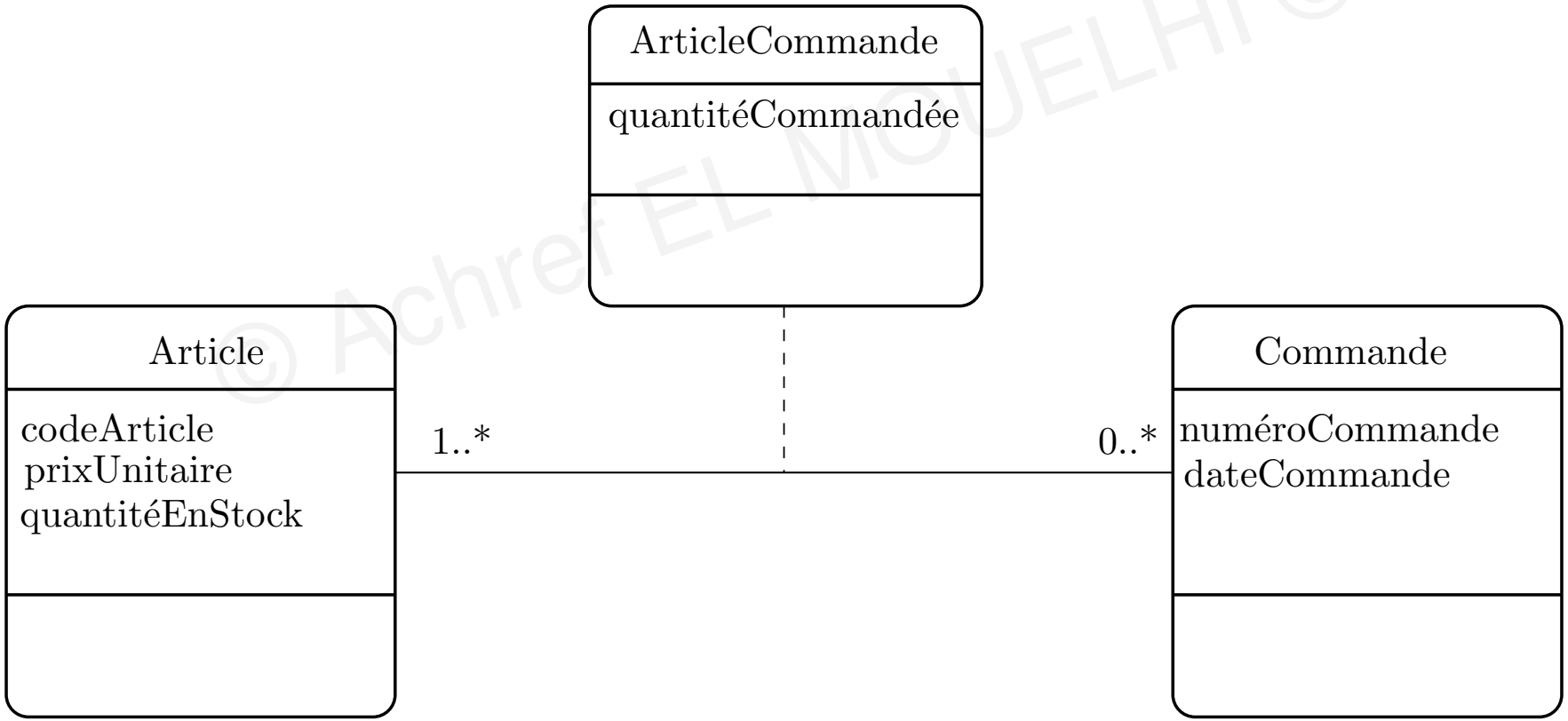
Conclusion

La `quantitéCommandée` ne peut être ni dans `Article` ni dans `Commande`.

Conclusion

La `quantitéCommandée` ne peut être ni dans `Article` ni dans `Commande`.

Solution : créer une classe d'association



UML

Définition

- En grec : prendre plusieurs formes
- En POO : possibilité de définir plusieurs méthodes avec le même nom
  - même si la même méthode existe dans la classe mère avec la même signature
  - même si une méthode avec le nom existe dans la classe mais avec une signature différente

UML

Surcharge (overload)

- Définir dans une classe plusieurs méthodes portant le même nom et avec des signatures différentes
- Autorisée en **Java** et **C#** mais pas par tous les LOO (comme **PHP**...)

UML

Surcharge (overload)

- Définir dans une classe plusieurs méthodes portant le même nom et avec des signatures différentes
- Autorisée en **Java** et **C#** mais pas par tous les LOO (comme **PHP**...)

Redéfinition

- Redéfinir une méthode, héritée, dans la classe fille
- Autorisée par tous les LOO (un des objectifs de l'héritage)
- La signature de la méthode redéfinie doit être identique à celle de la méthode héritée dans la classe mère

UML

Remarque

- Lorsqu'une méthode est appelée, le compilateur commence par regarder dans la classe de l'objet où la méthode a été appelée
- S'il la retrouve, il l'exécute
- Sinon il la repasse au niveau supérieure (classe mère directe)

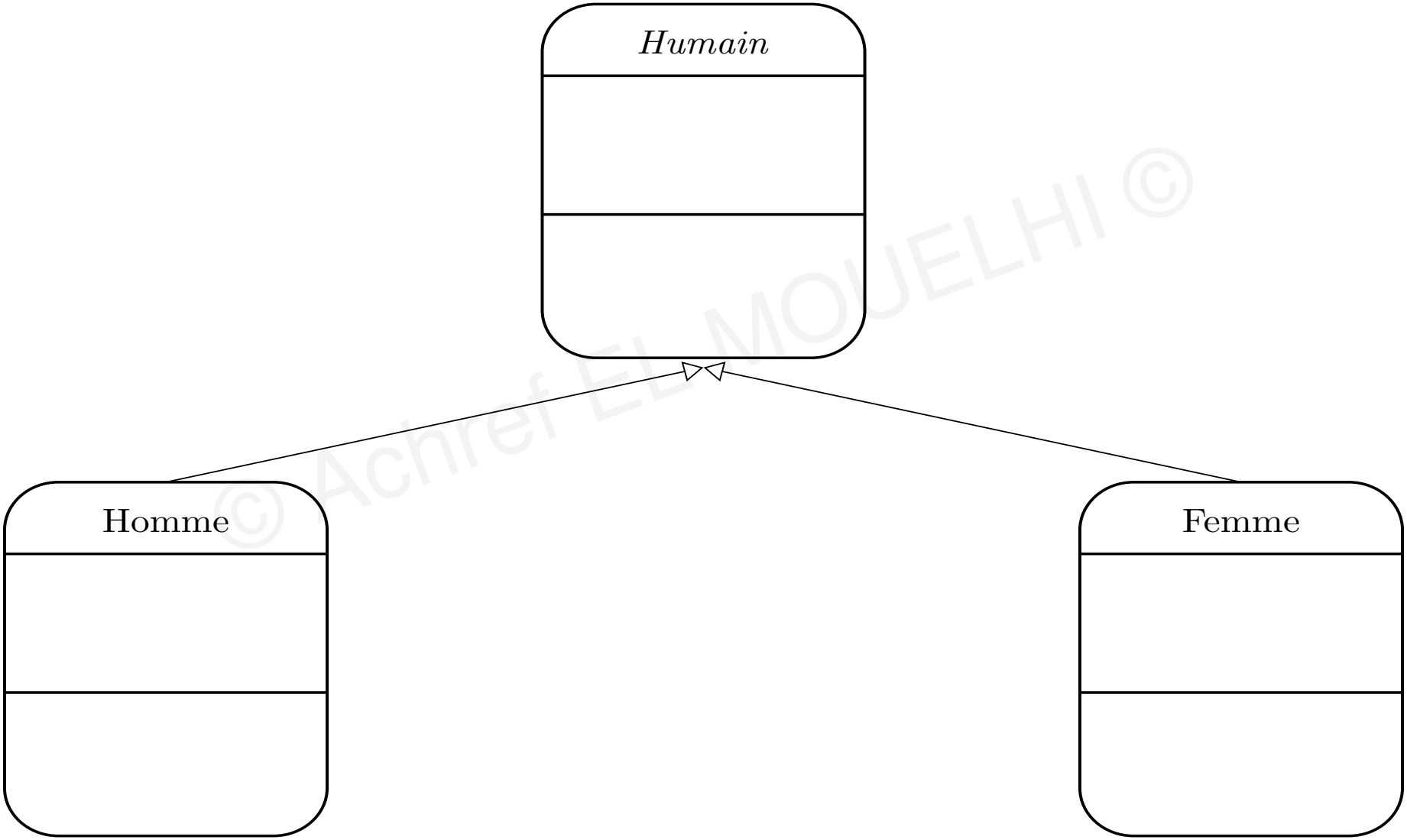
UML

Classe abstraite

- Classe qu'on ne peut instancier
- Par exemple, on sait qu'un être humain est soit homme soit femme. Donc, la classe `Humain` peut être déclarée abstraite.

UML

En UML, on écrit le nom d'une classe abstraite en *italique*



UML

Méthode abstraite

- C'est une méthode indéfinie (elle n'est pas implémentée)
- Si une classe contient une méthode abstraite, elle doit être aussi déclarée abstraite
- Si une classe hérite d'une classe abstraite, alors elle doit implémenter les méthodes abstraites de cette dernière

UML

Méthode abstraite

- C'est une méthode indéfinie (elle n'est pas implémentée)
- Si une classe contient une méthode abstraite, elle doit être aussi déclarée abstraite
- Si une classe hérite d'une classe abstraite, alors elle doit implémenter les méthodes abstraites de cette dernière

Remarque

Certains LOO, comme le **C++**, permettent de proposer une implémentation par défaut pour les méthodes abstraites

UML

Classe finale

Classe qu'on ne peut hériter

UML

Classe finale

Classe qu'on ne peut hériter

Méthode finale

Méthode que les classes filles ne peuvent redéfinir

UML

Exercice

- Une classe abstraite doit-elle avoir une ou plusieurs méthodes abstraites ?
- Une classe finale peut-elle avoir une méthode abstraite ?
- Une classe abstraite peut-elle avoir une méthode finale ?
- Une classe finale doit-elle avoir une méthode finale ?
- Une classe finale peut-elle hériter d'une classe abstraite ?
- Une classe peut-elle être finale et abstraite à la fois ?

UML

Une interface (un contrat)

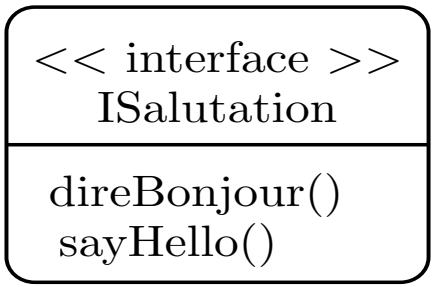
- Comme une classe 100% abstraite
- Définie par le mot clé `interface`
- Contenant uniquement toutes des méthodes abstraites

UML

Une interface (un contrat)

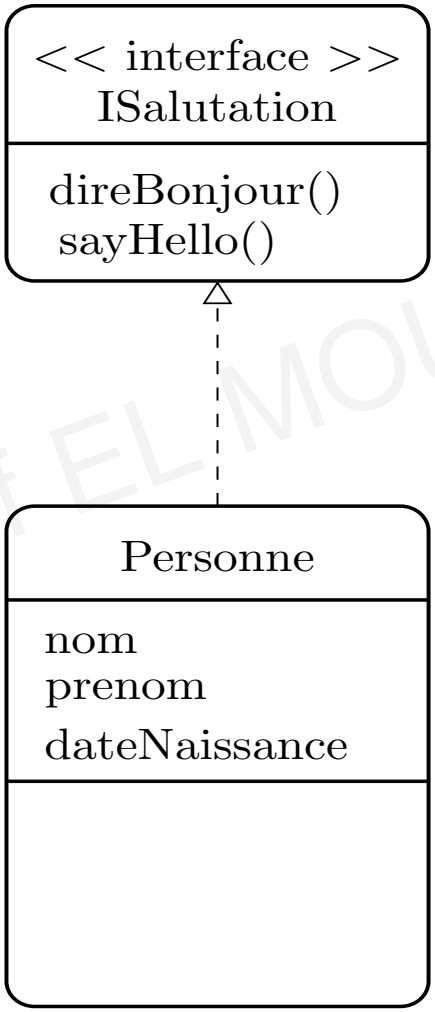
- Comme une classe 100% abstraite
- Définie par le mot clé `interface`
- Contenant uniquement toutes des méthodes abstraites

En UML, une interface est schématisée ainsi



UML

En UML, voici comment dire qu'une classe hérite d'une interface

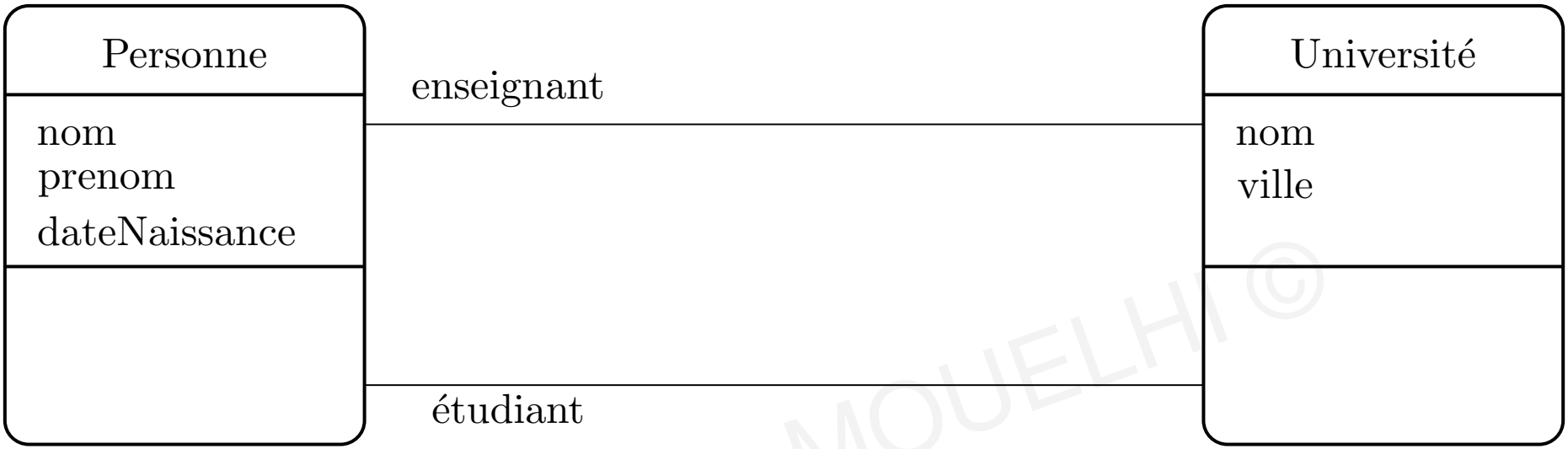


UML

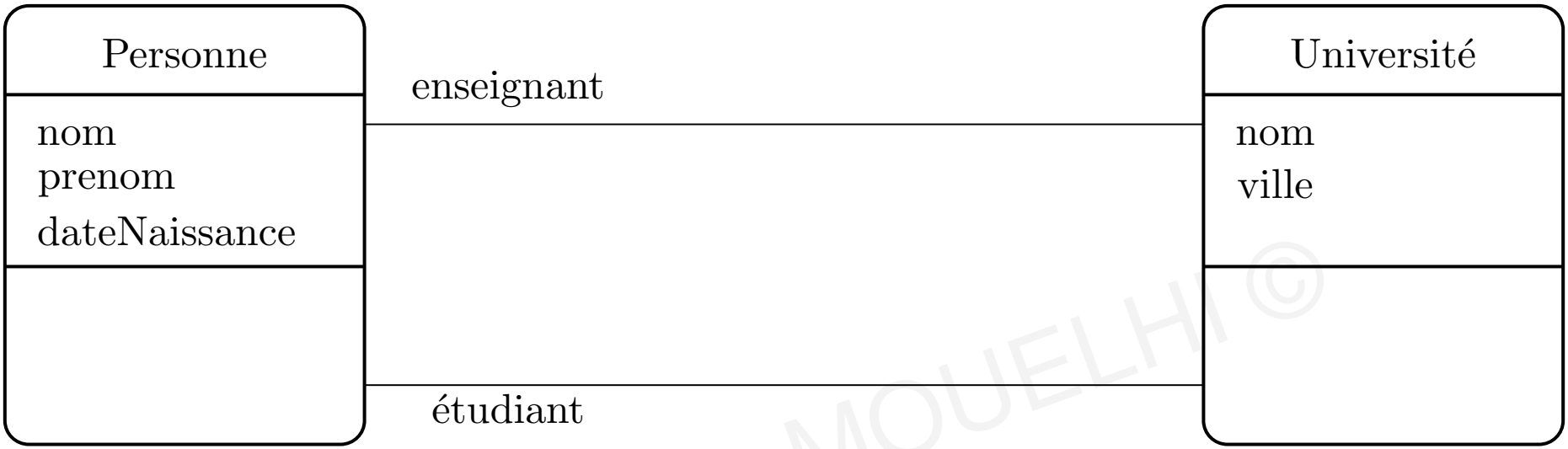
Remarques

- Une classe qui hérite d'une interface doit implémenter toutes ses méthodes
- Une classe peut hériter de plusieurs interfaces
- Une interface peut hériter d'autres interfaces

Étant donné l'exemple suivant



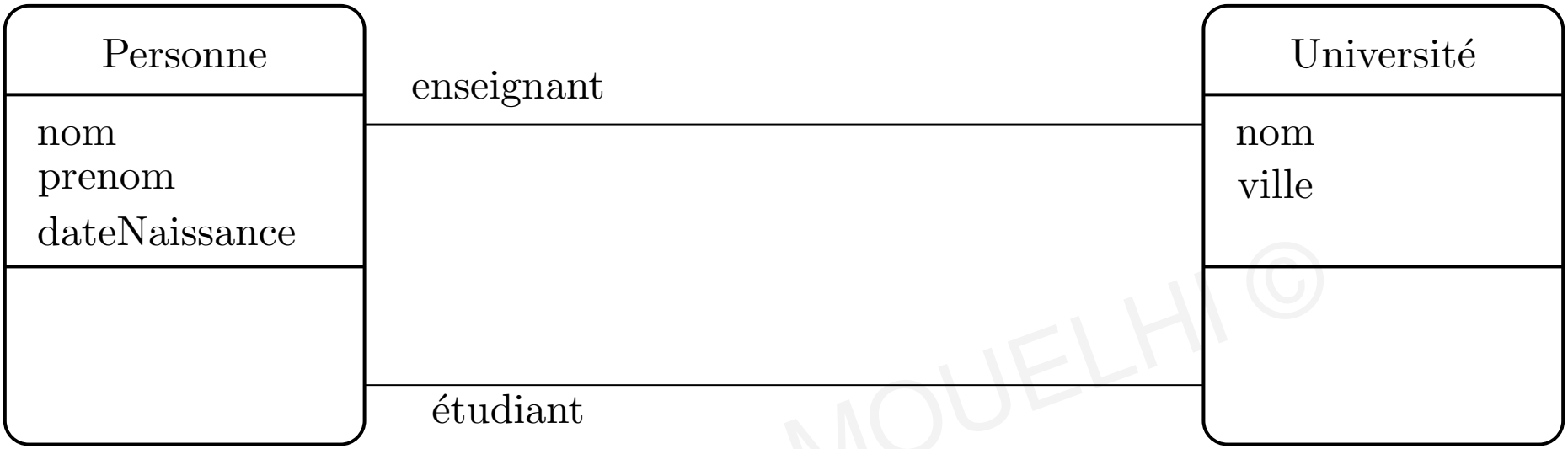
Étant donné l'exemple suivant



Question

Est-ce qu'une personne peut être à la fois Enseignant et étudiant ?

Étant donné l'exemple suivant



Question

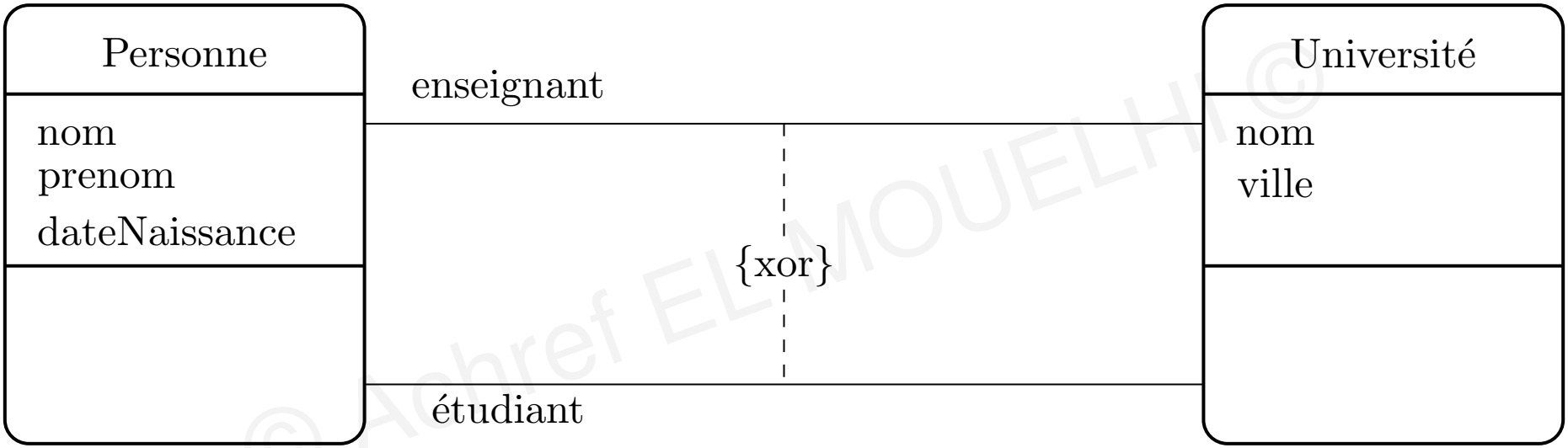
Est-ce qu'une personne peut être à la fois Enseignant et étudiant ?

Réponse

D'après ce diagramme de classes : oui

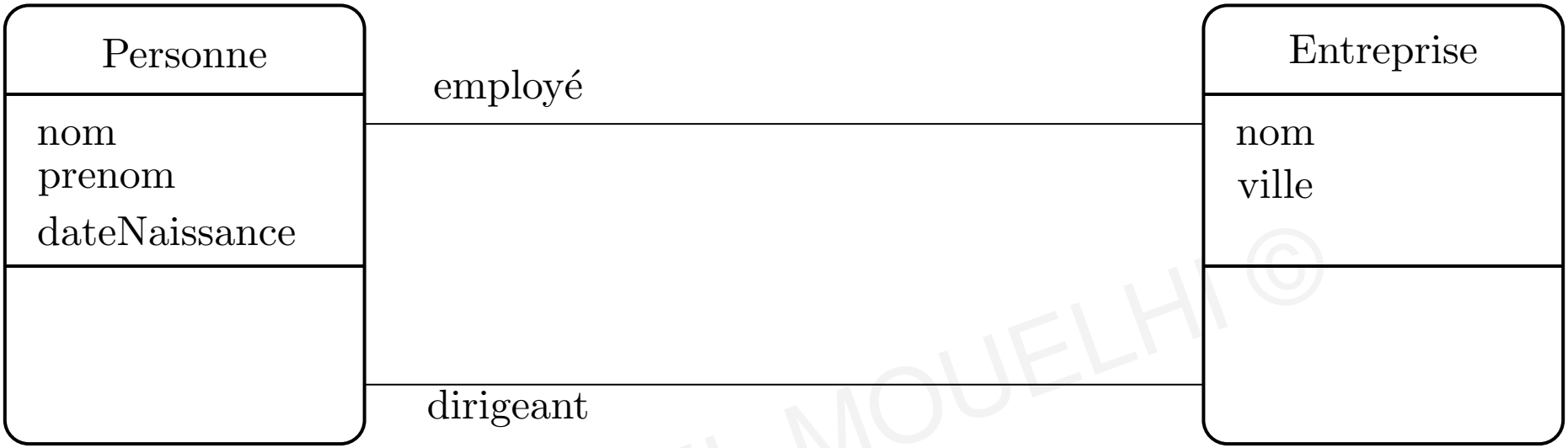
UML

Pour indiquer qu'une personne ne peut avoir les deux rôles à la fois, on rajoute une contrainte

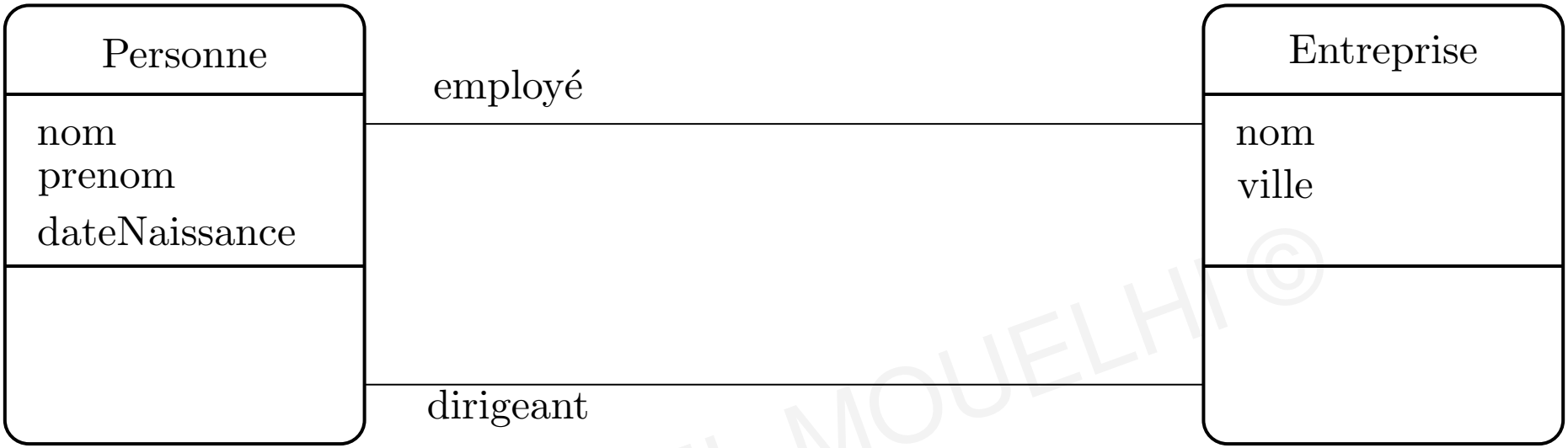


xor : eXclusive OR (OU exclusif)

Étant donné l'exemple suivant



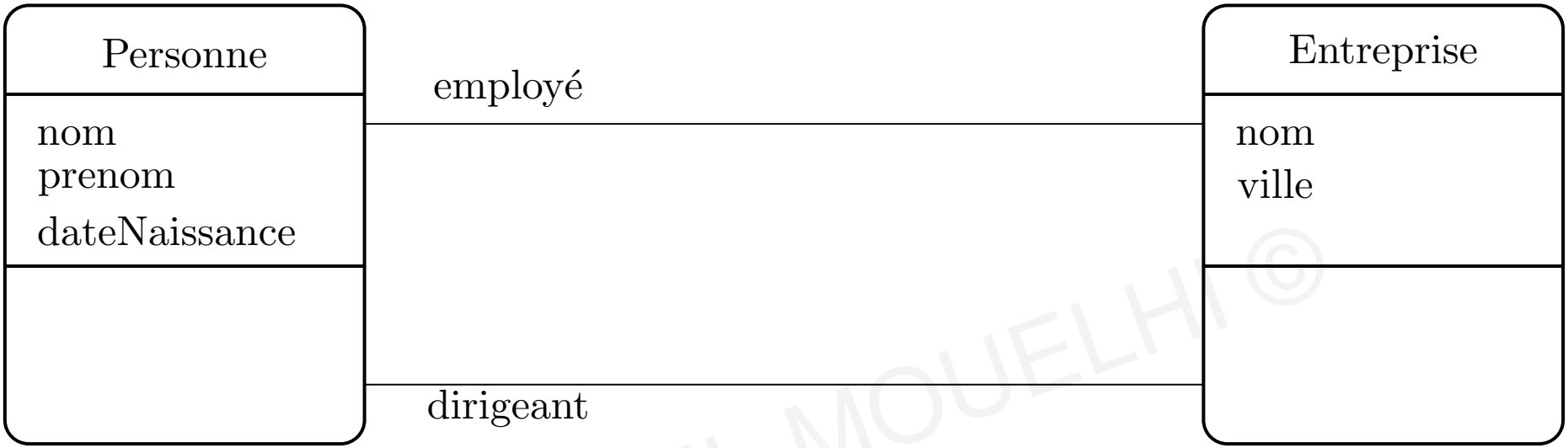
Étant donné l'exemple suivant



Question

Est-ce qu'une personne qui n'est pas employé dans cette entreprise peut la diriger ?

Étant donné l'exemple suivant



Question

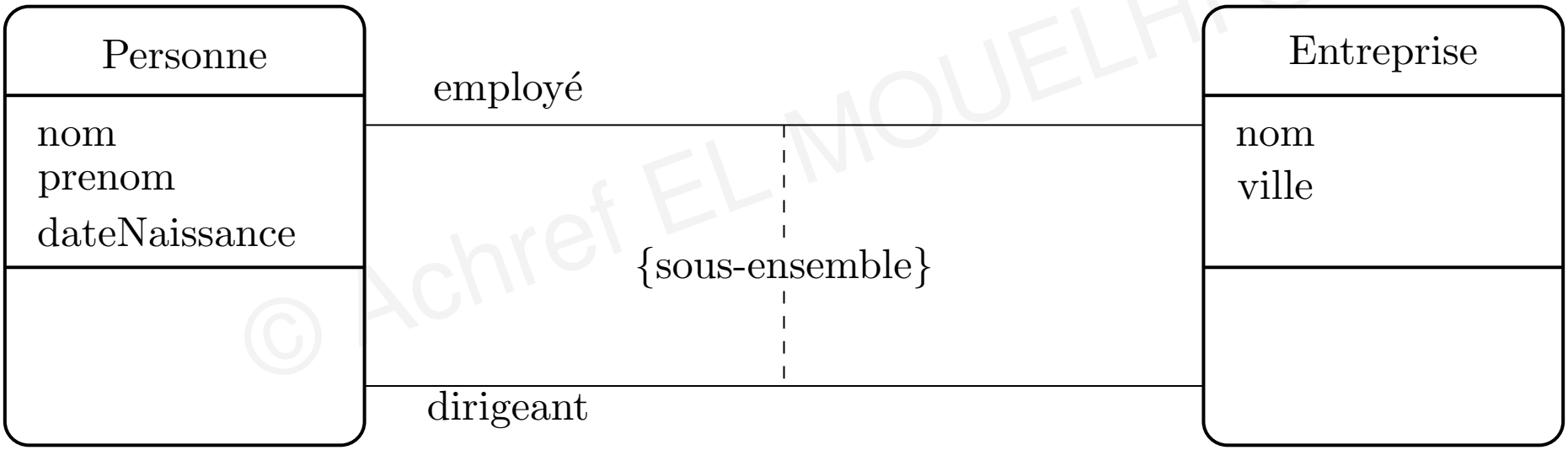
Est-ce qu'une personne qui n'est pas employé dans cette entreprise peut la diriger ?

Réponse

D'après ce diagramme de classes : oui

UML

Pour indiquer qu'une personne qui dirige une entreprise est forcément un de ses employés, on rajoute la contrainte suivante



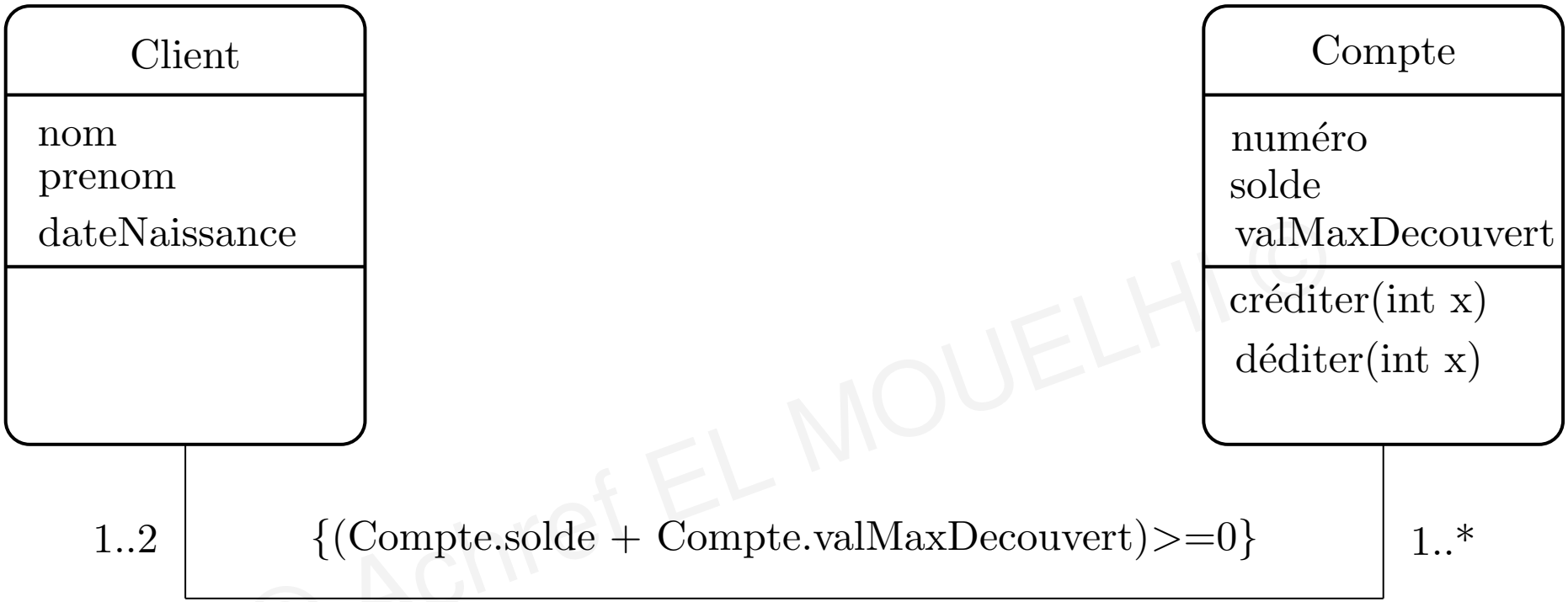
Étant donné l'exemple suivant

Étant donné l'exemple suivant

Question

Comment indiquer que le solde de chaque compte peut être négatif sans dépasser la valeur maximale de découvert ?

Il faut ajouter une contrainte



Remarque

Si on ajoute le nom d'association + les rôles, l'association deviendra trop dense et presque illisible.

Solution : utiliser **OCL**

- **O**bject **C**onstraint **L**anguage
- Initialement, un projet d'**IBM**
- Appartenant à **UML** la version 1.1
- Langage formel d'expression
- Permettant de définir des contraintes sur les différents diagrammes d'**UML**, et en particulier le diagramme de classes
- Basé sur la théorie des ensembles et la logique des prédicats
- Permettant principalement d'exprimer 2 types de 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

UML

Deux versions d'**OCL**

- **OCL 1** : intégré dans **UML 1.1**
- **OCL 2** : intégré dans **UML 2.0** et pouvant être généralisé sur d'autres modèles que ceux d'**UML**

UML

Avec **OCL**, les contraintes peuvent être définies un peu loin de l'association

Dans ce cas, il faut

- préciser le contexte
- définir la contrainte

```
context Compte
inv : solde + valMaxDecouvert >= 0
```

UML

Avec **OCL**, les contraintes peuvent être définies un peu loin de l'association

Dans ce cas, il faut

- préciser le contexte
- définir la contrainte

```
context Compte
inv : solde + valMaxDecouvert >= 0
```

Explication

- `context` : indique l'élément concerné par la contrainte
- `inv` (pour invariant) : exprime une contrainte sur un élément qui doit être respectée en permanence.

On peut définir plusieurs contraintes

```
context Compte
inv : solde + valMaxDecouvert >= 0

context Compte::débiter(int somme )
pre : somme > 0 and solde + valMaxDecouvert >= somme
post : solde = solde@pre - somme
```

On peut définir plusieurs contraintes

```
context Compte
inv : solde + valMaxDecouvert >= 0

context Compte::débiter(int somme )
pre : somme > 0 and solde + valMaxDecouvert >= somme
post : solde = solde@pre - somme
```

Explication

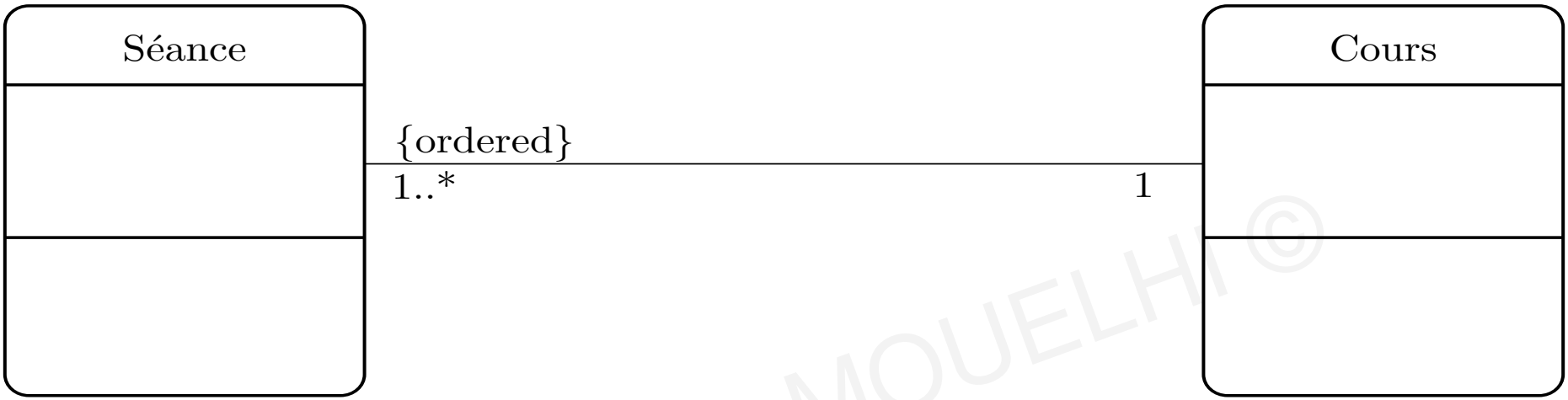
- `context` : indique l'élément concerné par la contrainte (ici la méthode `débiter()` de la classe `Compte`)
- `pre` : exprime une contrainte sur un élément qui doit être respectée pour que l'appel de la méthode soit valide
- `post` : exprime une contrainte sur un élément qui doit être respectée après l'appel de la méthode

UML

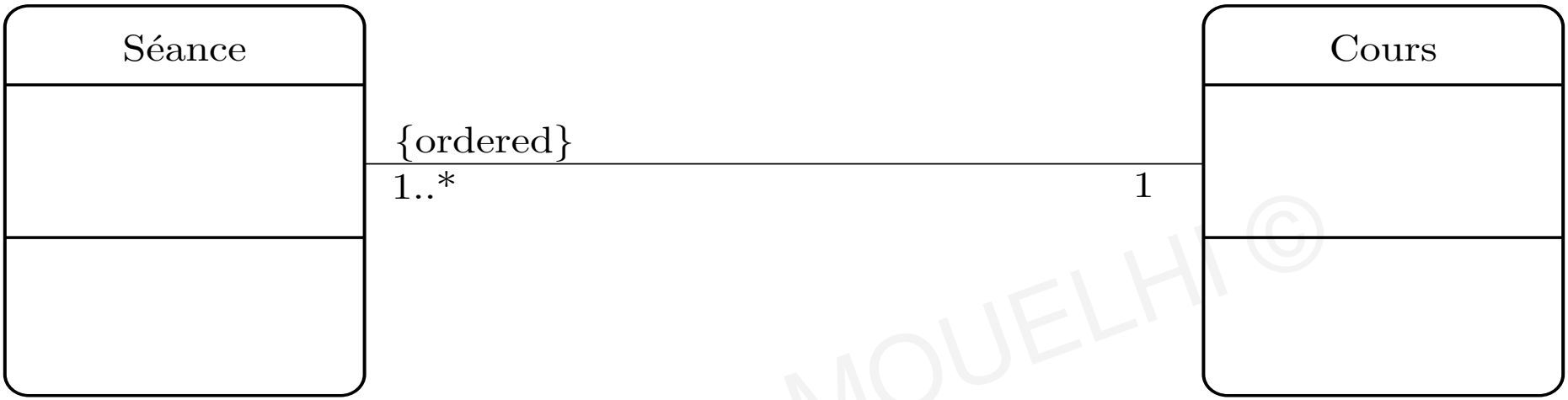
4 contraintes possibles pour les collections

- `Set` : ensemble mathématique sans doublons et sans ordre
- `Ordered` : ensemble mathématique sans doublons et avec ordre
- `Bag` : ensemble mathématique avec possibilité de doublons et sans ordre
- `Sequence` : ensemble mathématique avec possibilité de doublons et avec ordre

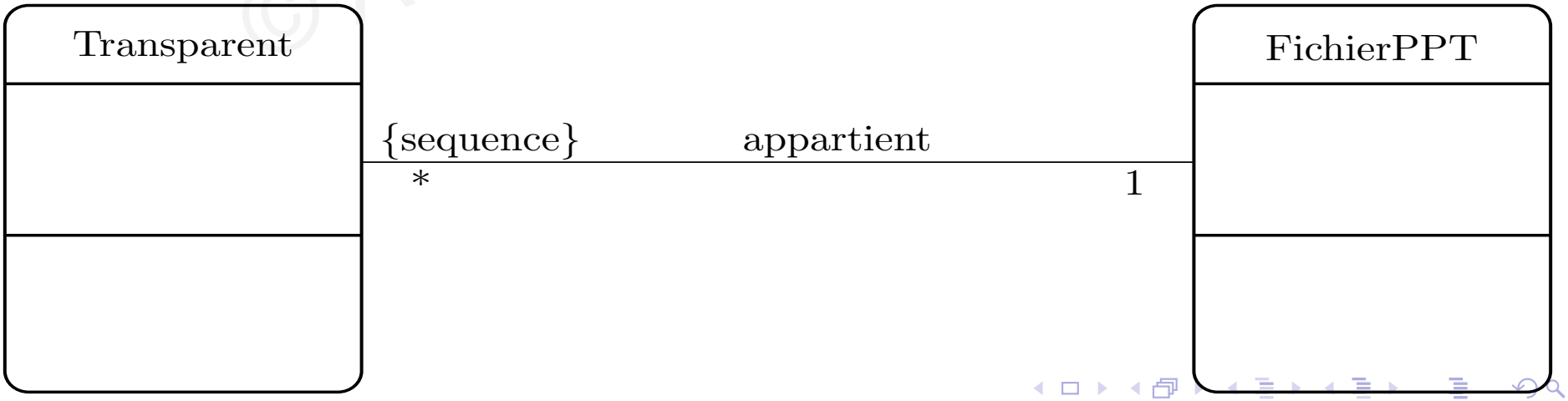
Exemple 1 : un cours est composé d'une séquence ordonnée (sans doublons) de séances



Exemple 1 : un cours est composé d'une séquence ordonnée (sans doublons) de séances



Exemple 2 : un fichierPPT contient une suite ordonnée (avec possibilité de doublons) de transparents



UML

Étapes

1. Préparer un dictionnaire de données
2. Identifier les classes et conserver les classes pertinentes
3. Identifier les associations et conserver les associations pertinentes
4. Identifier les attributs
5. Vérifier s'il est possible de simplifier en utilisant l'héritage
6. Itérer et affiner le diagramme

UML

Étape 1 : Préparer un dictionnaire de données

- Lire le texte
- Extraire tout nom ou verbe pouvant participer à notre système d'information
- Garder les synonymes tant qu'on n'a pas fini la lecture du texte
- Vérifier que la liste ne contient pas de doublons

UML

Étape 2 : Identifier les classes et conserver les classes pertinentes

- Éviter d'être trop sélectif
- Ne pas chercher l'héritage à cette étape
- Éliminer les classes redondantes, les synonymes, les classes vagues ou les classes sans lien avec le contexte

UML

Étape 3 : Identifier les associations et conserver les associations pertinentes

- Justifier l'existence d'un cycle car c'est souvent redondant
- Décomposer les associations n-aires en associations binaires
- Vérifier si les associations définies par rapport aux classes filles peuvent être remontées à la classe mère

UML

Étape 4 : Identifier les attributs

- Ne pas confondre attribut et classe
- Ne pas pousser la recherche des attributs à l'extrême
- Supprimer les synonymes
- Faire attention aux attributs de classe association
- Supprimer les attributs dérivés

Source : **UML 2 De l'apprentissage à la pratique** de **Laurent Audibert**

Les attributs dérivés peuvent être calculés à partir d'autres attributs et de formules de calcul. Lors de la conception, un attribut dérivé peut être utilisé comme marqueur jusqu'à ce que vous puissiez déterminer les règles à lui appliquer. Les attributs dérivés sont symbolisés par l'ajout d'un ≪ / ≫ devant leur nom.

UML

Étape 5 : Vérifier s'il est possible de simplifier en utilisant l'héritage

- Vérifier si certaines classes partagent certains attributs et/ou méthodes
- Éviter les raffinements excessifs

UML

Étape 6 : Itérer et affiner le diagramme

- Ne pas chercher un diagramme complet à la première passe
- Faire des itérations continuelles
- Vérifier le diagramme de classe après avoir fini les diagrammes d'états-transitions et de séquences
- Revenir sur le diagramme de classe après avoir fini les diagrammes d'états-transitions et de séquences
- Garder la possibilité de corriger des éventuelles anomalies du diagramme de classe pendant la phase de développement

UML

Citation 1 : **Jan van de Sneptscheut**

La différence entre la théorie et la pratique, c'est qu'en théorie, il n'y a pas de différence entre la théorie et la pratique, mais qu'en pratique, il y en a une.

Citation 2 : **Albert Einstein**

La théorie, c'est quand on sait tout et que rien ne fonctionne. La pratique, c'est quand tout fonctionne et que personne ne sait pourquoi. Ici, nous avons réuni théorie et pratique : Rien ne fonctionne... et personne ne sait pourquoi !