

# Diagrammes de classes



*Isabelle BLASQUEZ*  
*@iblasquez*

*Janvier 2022*



**Isabelle BLASQUEZ**



[@iblasquez](https://twitter.com/iblasquez)

**Enseignement : Génie Logiciel**

**Recherche : Développement logiciel agile**



ICSTUG #IUTAgile



CodeWeek. 

**#Software  
Craftsmanship**

 **IUSEOMIX** LIMOUSIN

# L'orienté **objet**, quelle **Classe** ! Et quel régal ...



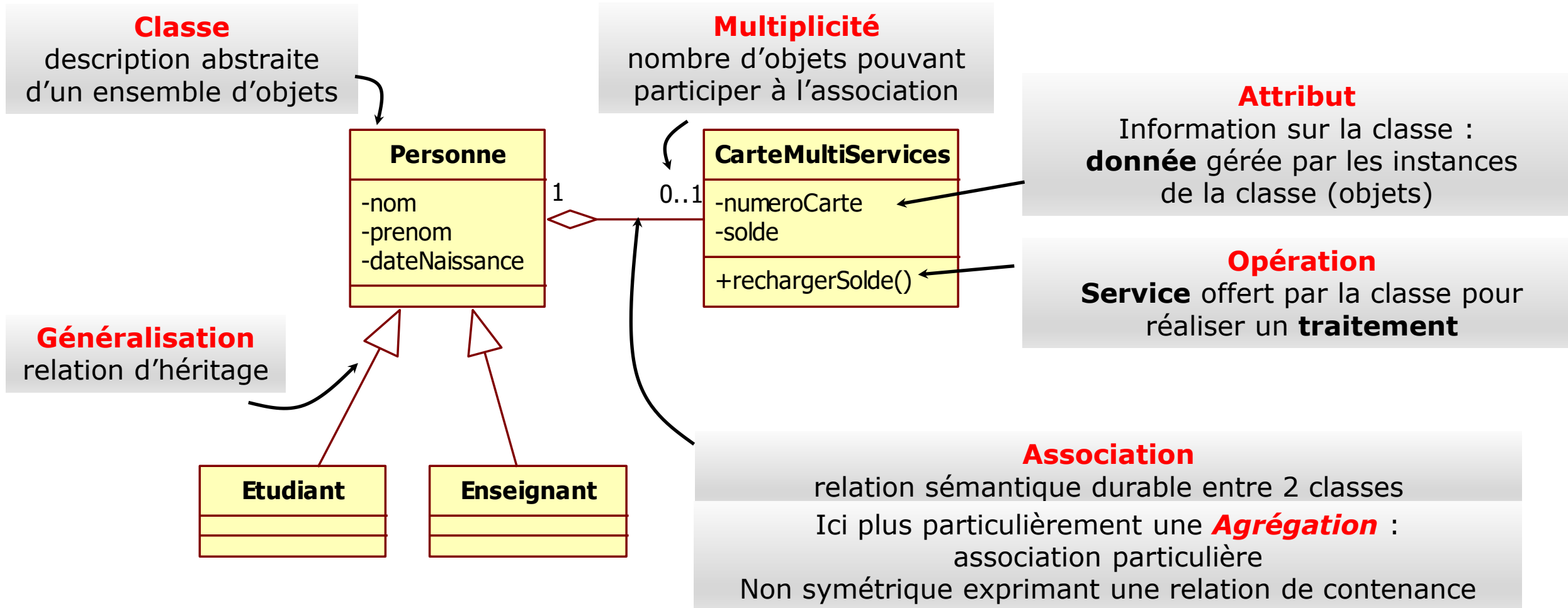
**Classe** : Patron pour créer des objets,  
réserver de l'espace mémoire

**Objet**  
(ou instance de classe)



# Représentation d'un diagramme de classes simplifié

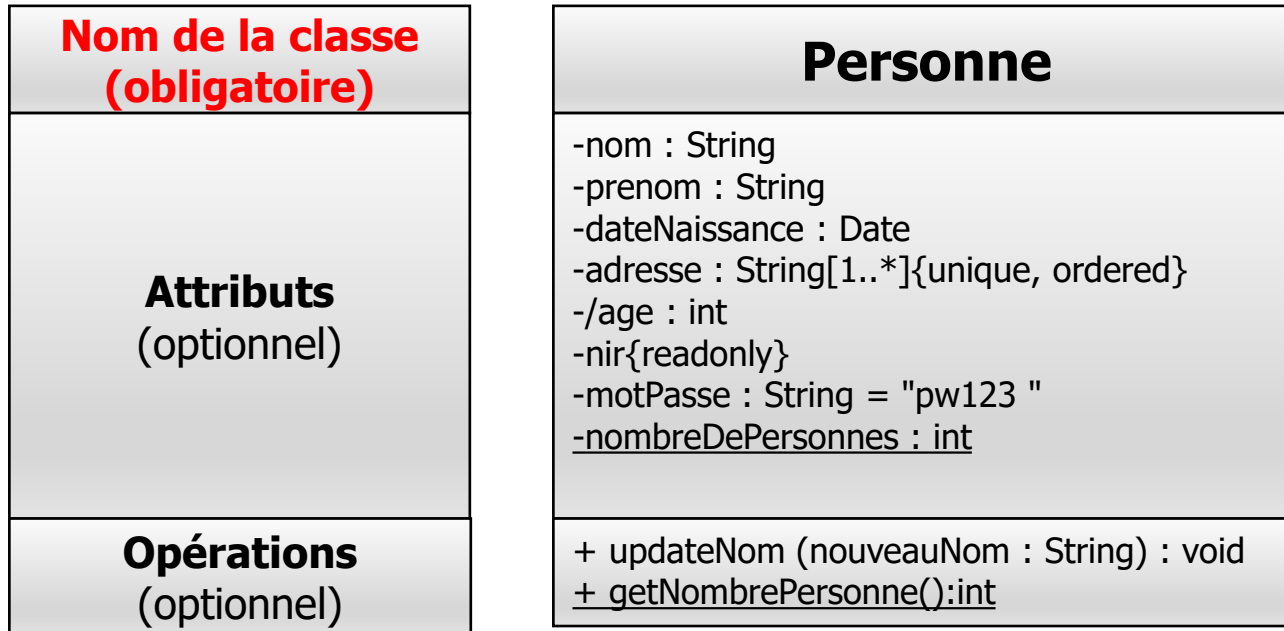
Le diagramme de classes fournit une vue statique du système.  
Il montre les **classes** (données + traitement) et leurs **relations**.



Le diagramme de classes fournit une **grande gamme de notations**...

# Classe

# Formalisme UML pour une classe



- Par convention :**
- le **nom de la classe** commencera par une **majuscule**
  - le **nom des attributs** et des **opérations** commencera par une **minuscule**

Syntaxe pour les attributs :

[visibilité] **nom** [: type] [multiplicité] = [valeurInitiale] [{propriété}]

Syntaxe pour les opérations :

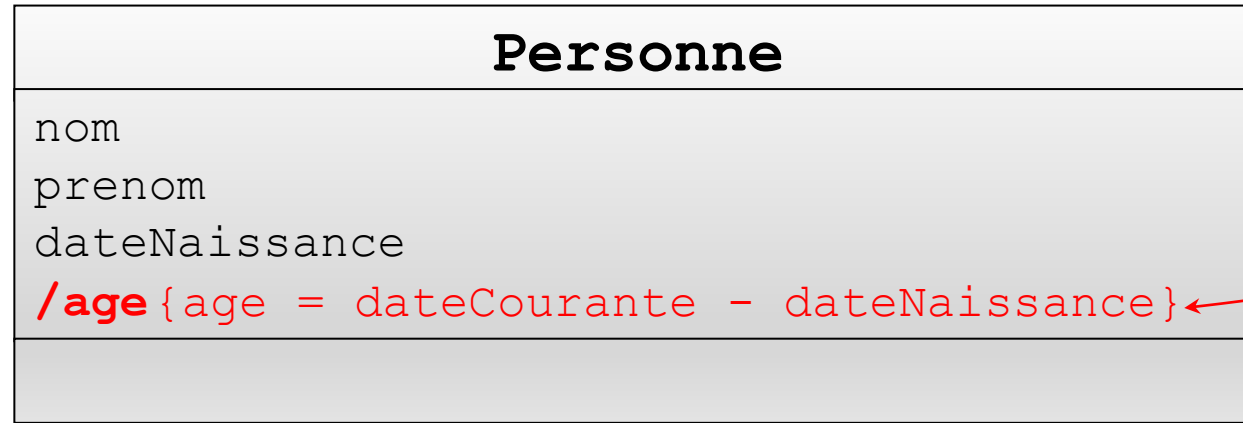
[visibilité] **nom** [(listeParametres)] [: typeRetour] [{propriété}]

public (+)  
private (-)  
package (rien)  
protected (#)

{readOnly}  
{unique}  
{non-unique}  
{ordered}  
{unordered}  
...

# Attribut dérivé et Attribut de classe : Exemples

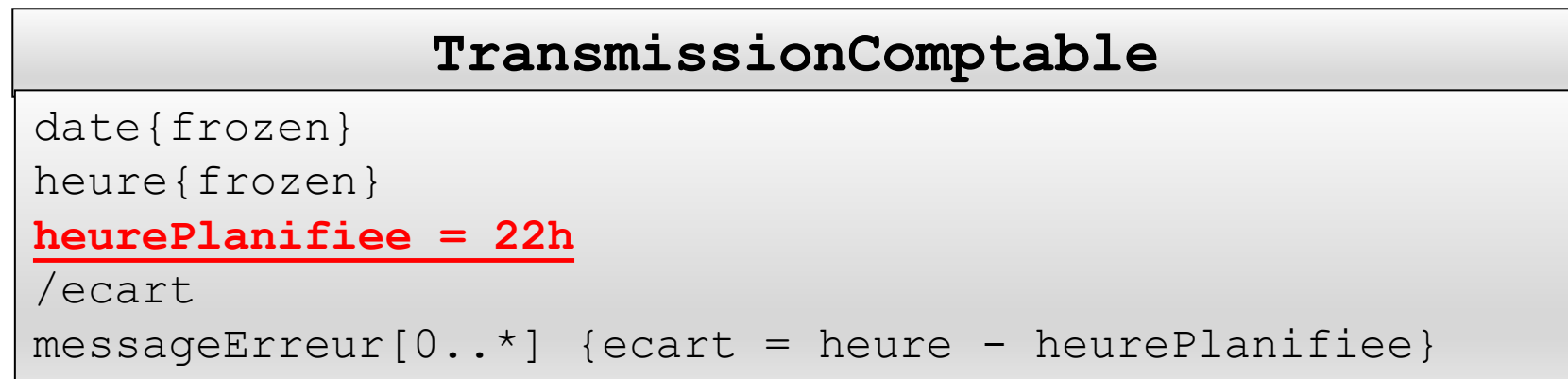
## Attribut dérivé



Contrainte

En analyse, un attribut dérivé ne précise pas encore ce qui doit être calculé par rapport à ce qui doit être stocké : ce sera un choix de conception.

## Attribut de classe



L'heure de la transmission est identique pour tous les objets de la classe.

# Association

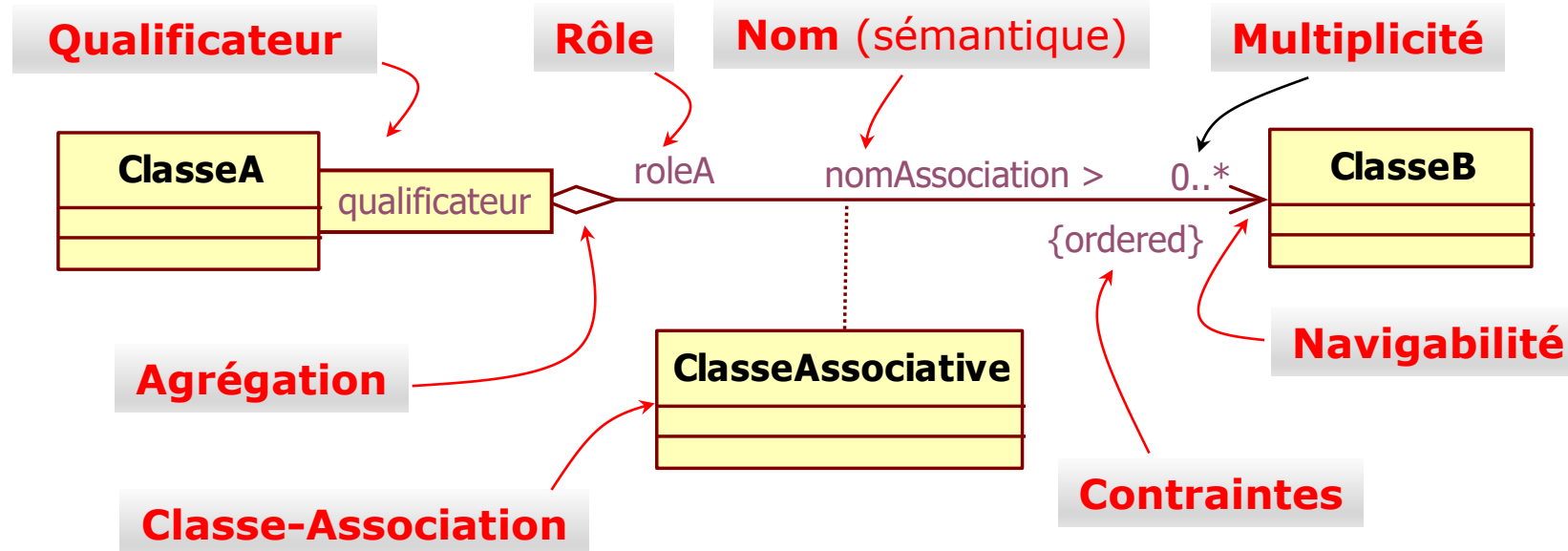


# Présentation l'association au sens UML

Une **association** est une **relation structurelle entre classes**.  
C'est une **connexion bi-directionnelle** représentée par un **trait continu**.



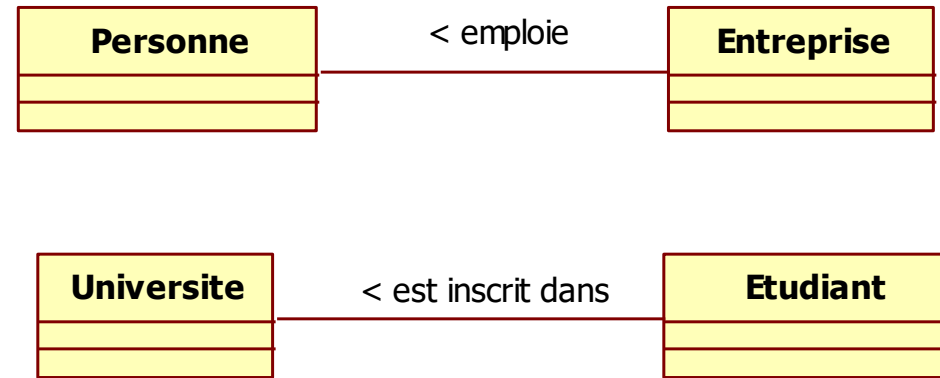
UML propose un ensemble complet de notations afin de détailler très précisément une association. En voici un extrait :



*Remarque : Les associations ne doivent pas forcément reprendre toutes ces notations à chaque fois, seules celles qui sont pertinentes !*

# Nom et Rôle d'une association

**Nom** d'une association (documentaire)



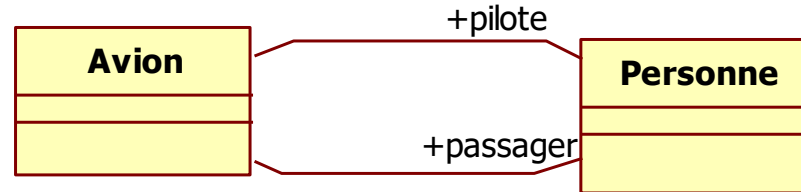
**Rôle** (structurel)



La **Personne** voit l'**Entreprise** comme son **employeur**  
et l'**Entreprise** voit la **Personne** **employé**

# Associations Multiples : prudence !

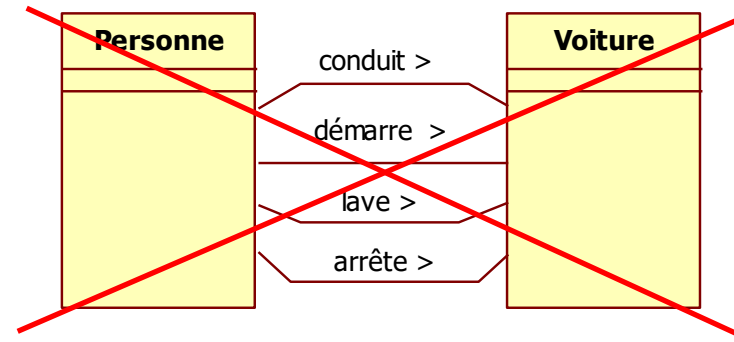
Possibilité de poser **plusieurs associations** entre 2 classes



*Mais la présence d'un grand nombre d'associations entre classes peut paraître suspecte.*



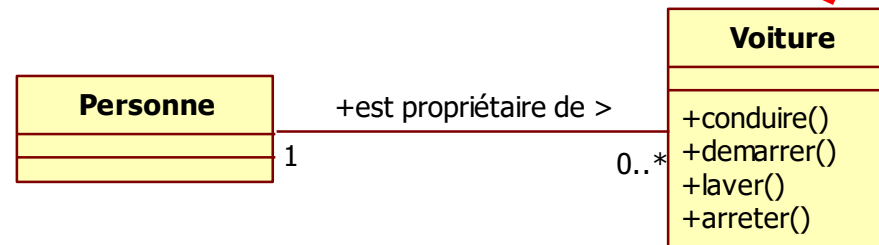
**Fort couplage à éviter !**



**NON**



**opération  $\neq$  association**



**OUI ! ! !**

# Multiplicité en UML

La **multiplicité** indique à combien d'instances de la classe située à cette extrémité, **1** instance de la classe située à l'autre extrémité est liée.



<b>1</b>	Un et un seul (multiplicité <b>par défaut</b> ) ( $\Rightarrow$ <b>exactement 1</b> )
<b>0..1</b>	Zéro ou un ( $\Rightarrow$ <b>au plus 1</b> )
<b>*</b> ou <b>0..*</b> ou <b>0..n</b>	De zéros à plusieurs ( $\Rightarrow$ <b>aucun, 1 ou plusieurs</b> )
<b>1..*</b> ou <b>1..n</b>	De un à plusieurs ( $\Rightarrow$ <b>au moins 1</b> )
<b>n</b>	Exactement <b>n</b> (entier naturel $> 0$ )
<b>m..n</b>	De <b>m</b> à <b>n</b> (entiers naturels $m \geq n$ )

# Multiplicité d'une Association

Exemple : Soit la modélisation suivante. Que signifie-t-elle ?



**Le choix de la multiplicité a un impact sur l'interprétation du modèle**



*Le seul changement de la multiplicité côté Entreprise change l'interprétation de la modélisation du problème !*

# Illustration d'un diagramme de classes simple

*... Pour mettre en application les notions vues précédemment, modélisons à l'aide d'un diagramme de classes le problème suivant ...*

1. Une université regroupe des personnes.
2. Dans une université, on trouve :
  - des étudiants qui sont inscrits à l'université.
  - des enseignants qui sont affectés à l'université. Dans ce cas, l'université est considérée comme leur employeur.
3. Un étudiant n'est inscrit qu'à une et une seule université.
4. Un enseignant peut être affecté à une université ou à aucune.
5. Dans une université, il peut y avoir de 0 à n étudiants.
6. Dans une université, il peut y avoir de 0 à n enseignants.

# Modélisation pas à pas de «A l'Université» (1/4)



1. Une **université** regroupe des **personnes**.



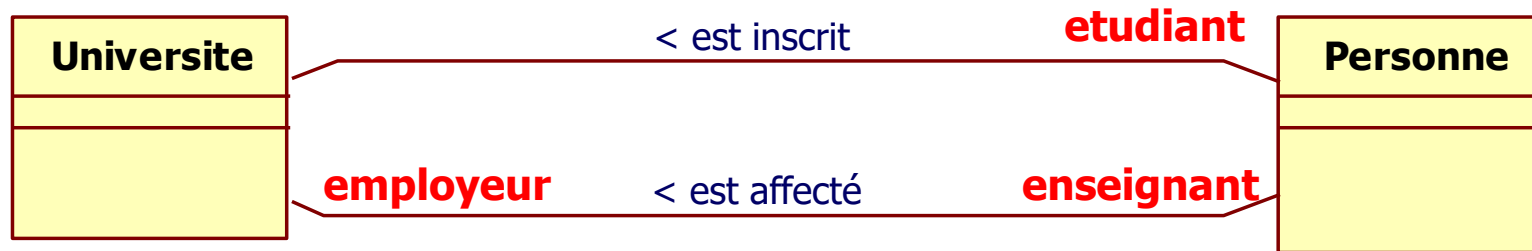
⇒ 2 concepts : **université** et **personne**

⇒ 1 association : **regroupe**

# Modélisation pas à pas de «A l'Université» (2/4)



2. Dans une université, on trouve :
- des étudiants qui sont inscrits à l'université.
  - des enseignants qui sont affectés à l'université. Dans ce cas, l'université est considérée comme leur employeur.

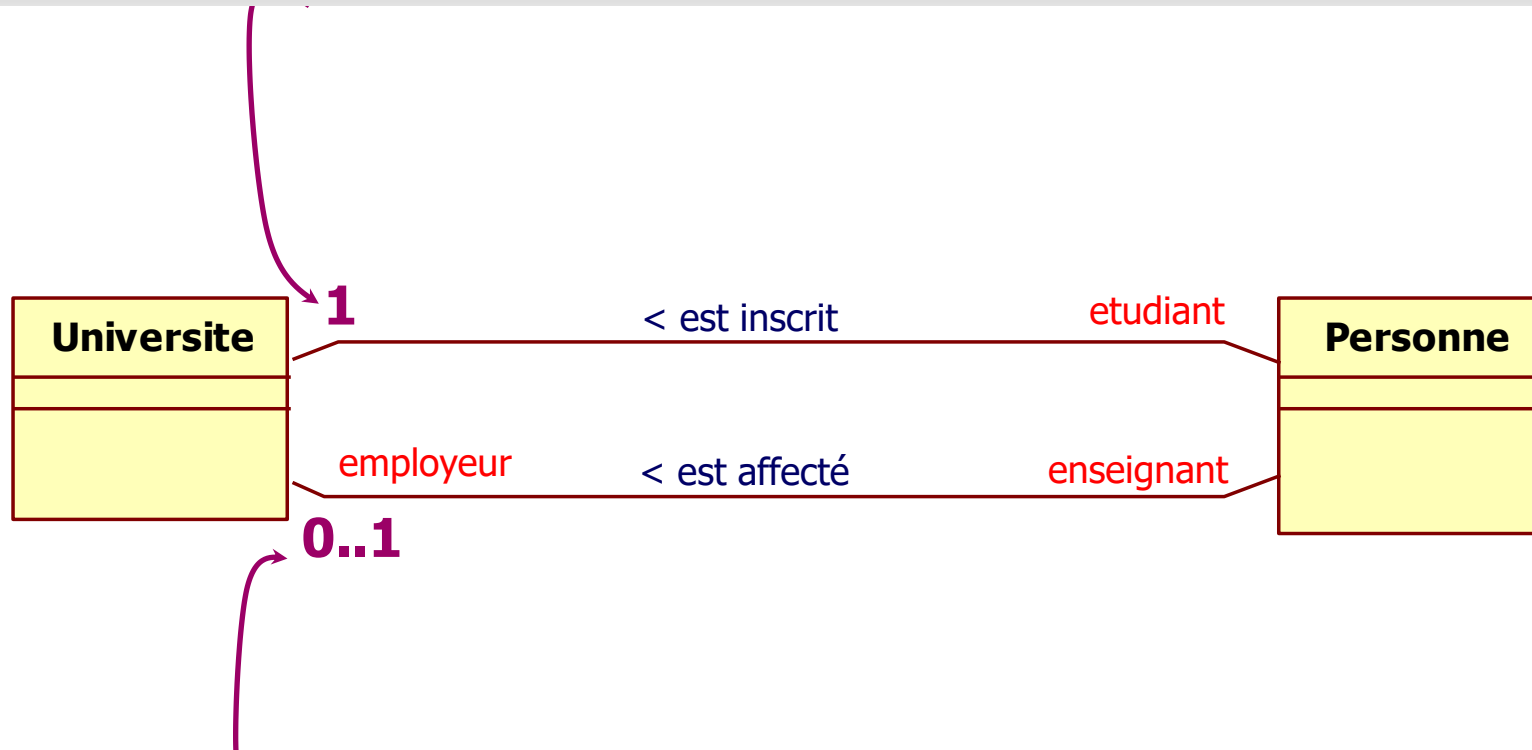




# Modélisation pas à pas de «A l'Université» (3/4)



3. Un étudiant n'est inscrit qu'à une et une seule université.



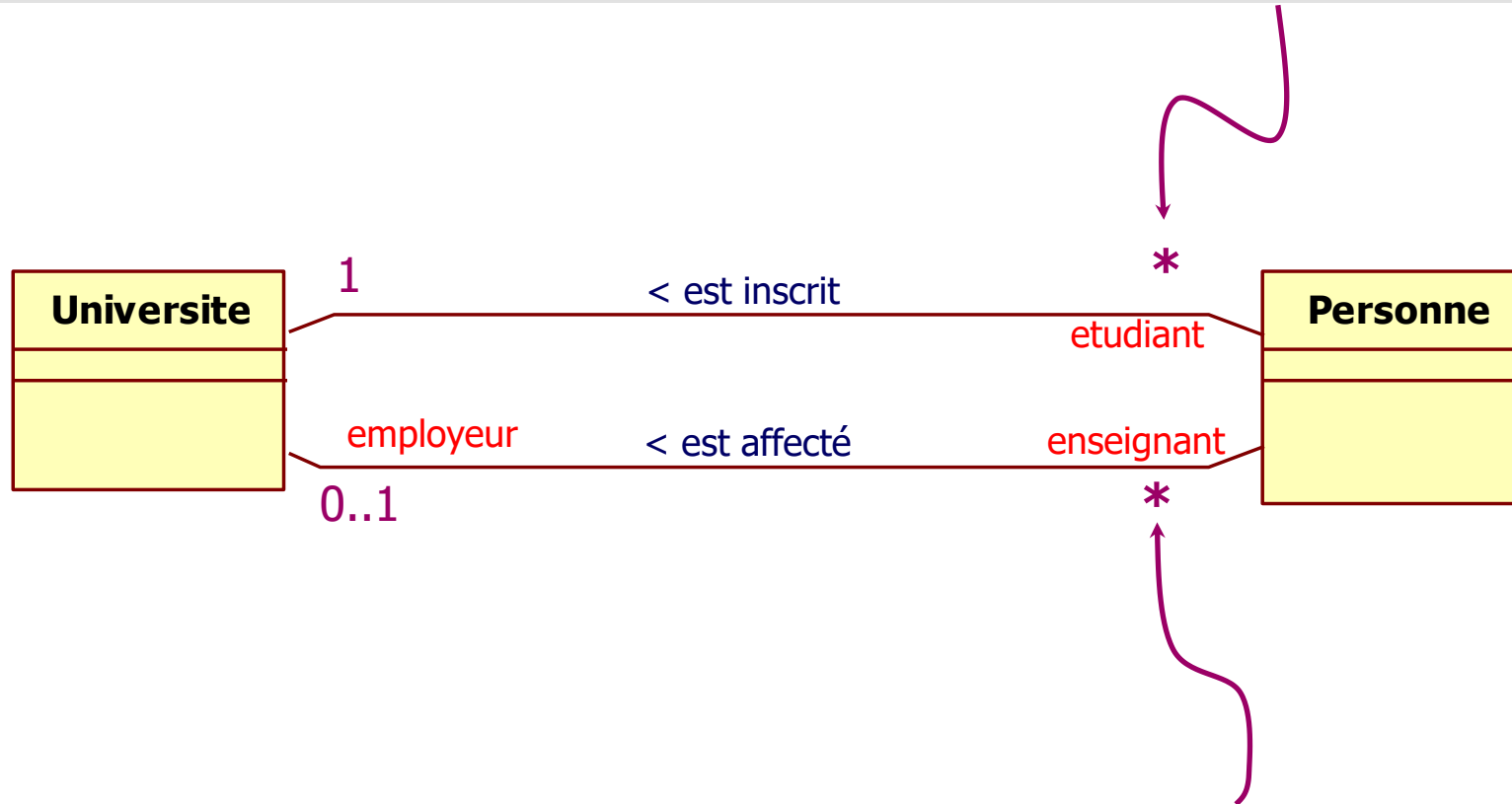
4. Un enseignant peut être affecté à une université ou à aucune.

**... Enfin la multiplicité apparaît ...**

# Modélisation pas à pas de «A l'Université» (4/4)



5. Dans une université, il peut y avoir de 0 à n étudiants.



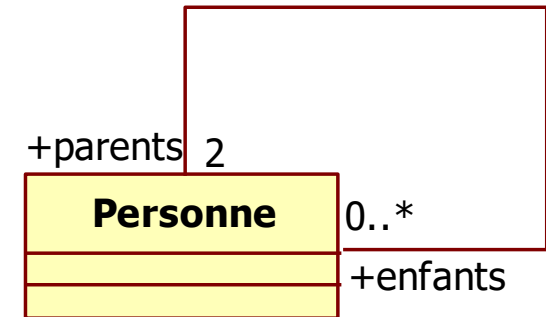
6. Dans une université, il peut y avoir de 0 à n enseignants.

# Association Réflexive

Une **association réflexive** est une association qui permet de **relier une classe à elle-même**

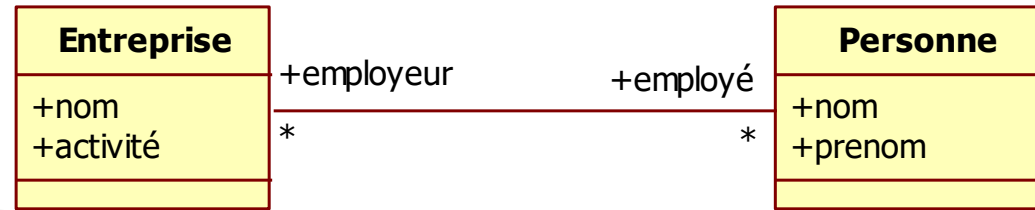
L'indication du **rôle** est **essentiel** à la clarté du diagramme

Exemple : Toute personne possède 2 parents et de 0 à plusieurs enfants



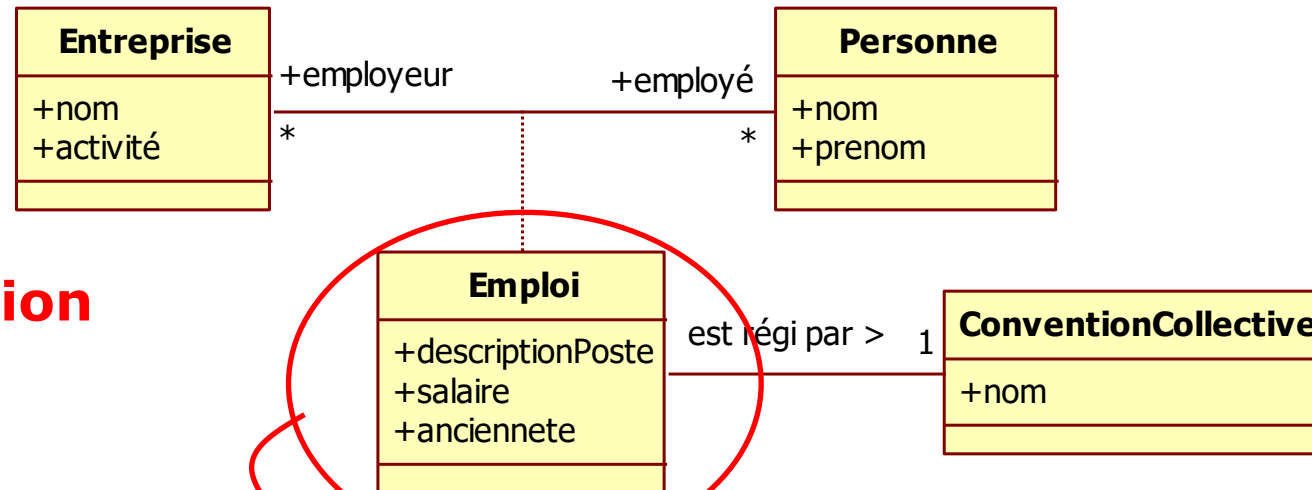
# Classe Associative : Exemple introductif

## Problème :



où placer la description du poste, le salaire, l'ancienneté ?

... Il faut valoriser ces attributs pour chaque *couple d'instance* (*Entreprise-Personne*) et pas simplement pour un objet ...

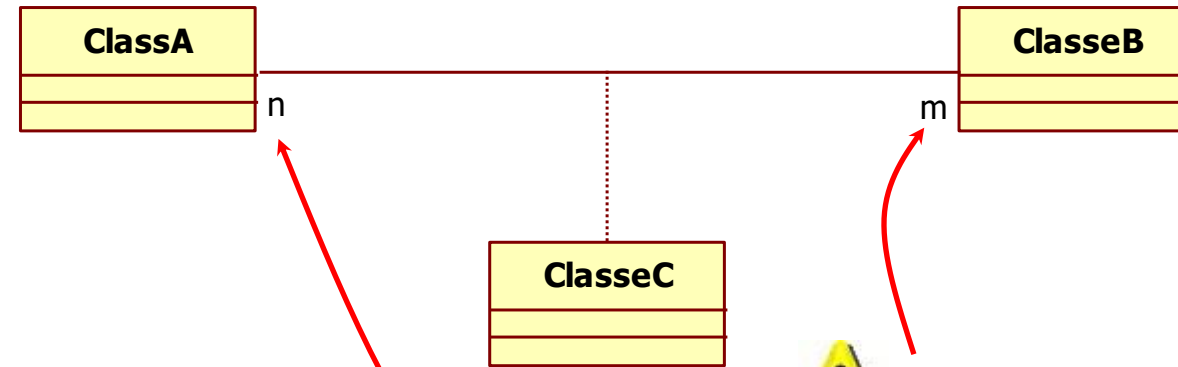


**1 seule classe-association est permise sur une association**

- ⇒ la classe `Emploi` est appelée **classe associative** (association et classe)
- ⇒ Elle contient des attributs et des méthodes et peut établir des relations avec d'autres classes

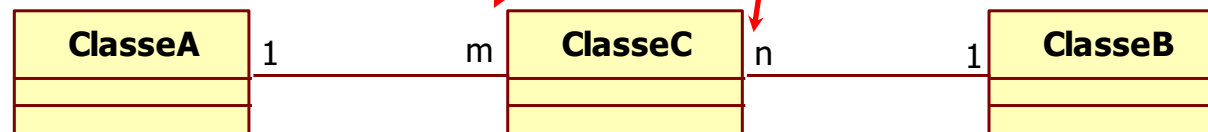
# Classe Associative en conception

## Phase Analyse



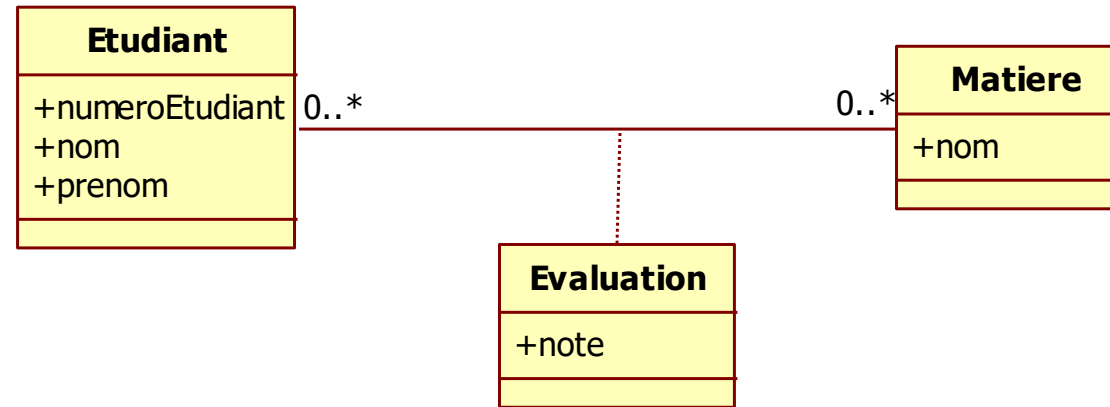
*à l'emplacement des multiplicités*

## Phase Conception

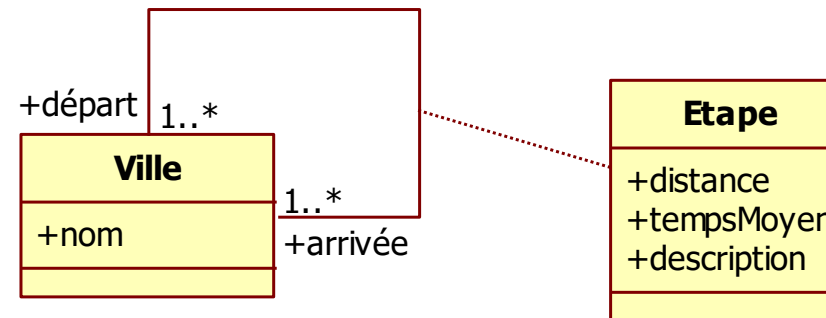


# Classe Associative: Exemples

Pour chaque matière, chaque étudiant est évalué et obtient au final une note

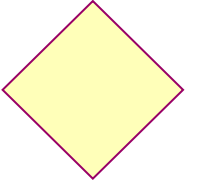


Une **étape** du Tour de France décrit une relation entre deux **villes**.



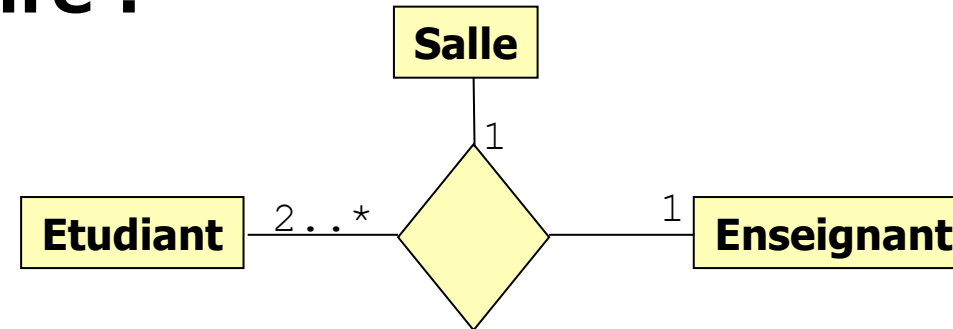
# Associations n-aires (arité des associations)

Une association qui relie **n classes** est une **association n-aires**

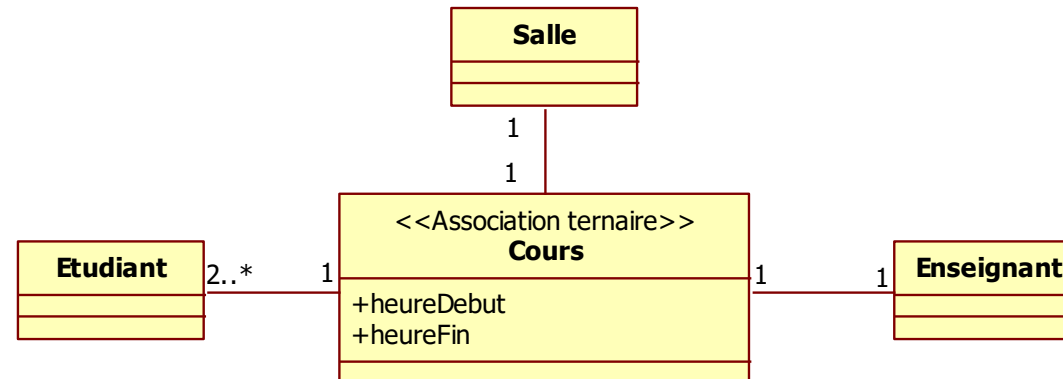


## Exemples d'associations ternaire :

→ Association ternaire modélisant un cours



→ Association ternaire faisant apparaître la classe-associative **Cours**



# Agrégation

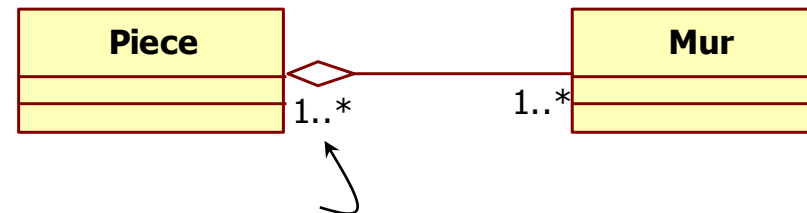
**Une agrégation** est un cas particulier d'association **non symétrique** exprimant une **relation de contenance** (agrégat/agrégé, tout/partie, composé/composant)



Remarque : *une agrégation n'a pas besoin d'être nommée : implicitement elle signifie **contient***

## Exemple d'agrégation :

Une pièce contient des murs



*1..\* car 1 mur peut être commun à 2 pièces ou plus ...*

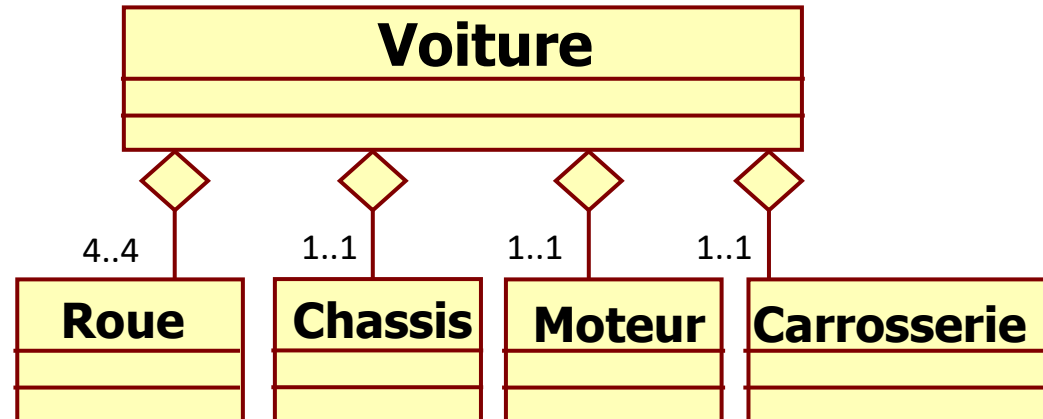


# Agrégation : Exemple



**Dans une agrégation, la contrainte d'intégrité fonctionnelle est **faible****

- la partie (composant) peut exister sans l'agrégat (composé)
- si le composé (agrégat) disparaît, le composant (partie) peut continuer d'exister



# Composition

**Une composition** est une **agrégation plus forte**  
(tout/partie ou composé/composant) qui implique :

- *qu'un composant ne peut appartenir qu'à un seul composé à la fois*  
(multiplicité maximale à **1** du côté du composé  $\Rightarrow$  **agrégation non partagée**) } (1)
- *que la destruction du composé entraîne la destruction de tous ses composants :*  
*le cycle de vie des composants est fortement lié à celui du composé* } (2)



## Exemple de composition :



Un répertoire *est composé* de fichiers

- Un fichier appartient à un et un seul répertoire  
 $\Rightarrow$  (1) est vérifié !
- La destruction du répertoire entraîne la destruction de tous les fichiers qu'il contient  
 $\Rightarrow$  (2) est vérifié !

# Navigabilité d'une Association : Définition

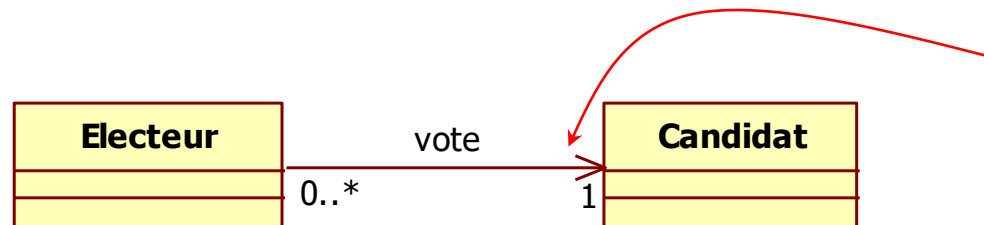
Par défaut, une association est **BIDIRECTIONNELLE**  
c-a-d navigable dans les deux sens : \_\_\_\_\_

**La navigabilité** indique s'il est possible de traverser une association.

Remarque :  
Ces 3 représentations sont équivalentes



## Exemple :



**La flèche** sur l'association restreint  
la navigabilité à un seul sens  
=> plus d'ambiguïté pour lire l'association ...

Un `Electeur` connaît un `Candidat`

Un `Candidat` ne connaît aucun `Electeur`

⇒ pas de navigation possible entre `Candidat` et `Electeur`

# Impact de la navigabilité sur le code

La navigabilité peut avoir **un impact sur l'implémentation**, en suggérant qu'un objet source mémorise une référence directe aux objets cible (référence qui portera le nom du rôle)

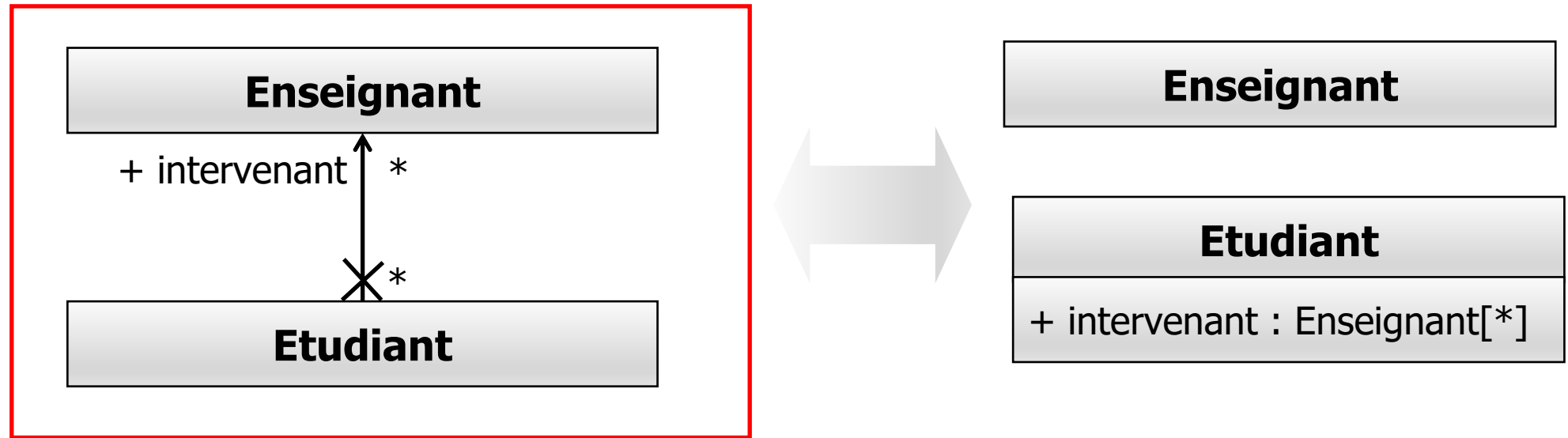


```
public class MotDePasse {  
  
    // rien.. car la classe MotDePasse  
    // ne connaît pas l'existence  
    // de la classe Utilisateur  
}
```

```
public class Utilisateur {  
    private MotDePasse cle;  
  
    public setMotDePasse(MotDePasse motDePasse);  
    public MotDePasse getMotDePasse();  
}
```

# Association & Navigabilité vs Attribut :

**Préférable**  
*de faire apparaître  
l'association  
et sa navigabilité  
dans la  
modélisation*



*... ce qui donnerait  
au niveau du code ...*

```
public class Enseignant {...}

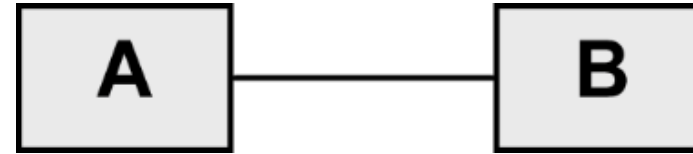
public class Etudiant{
    public Enseignant[] intervenant;
    ...
}
```

# Navigabilité : standard UML vs Bonne Pratique

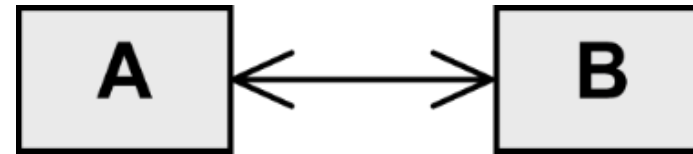
## Standard UML

## Bonne Pratique

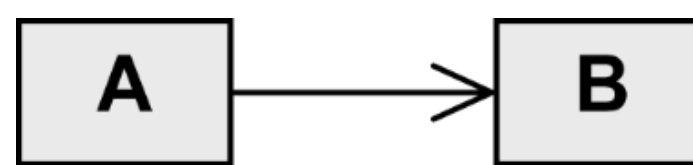
La navigation entre A et B  
est indéterminée



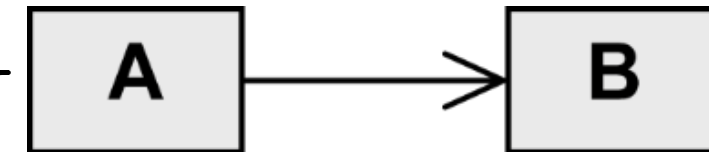
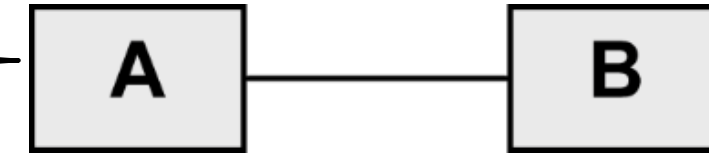
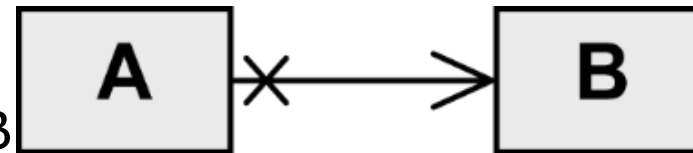
L'association entre les classes A et B  
est navigable de A vers B  
et de B vers A



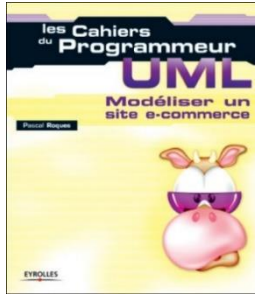
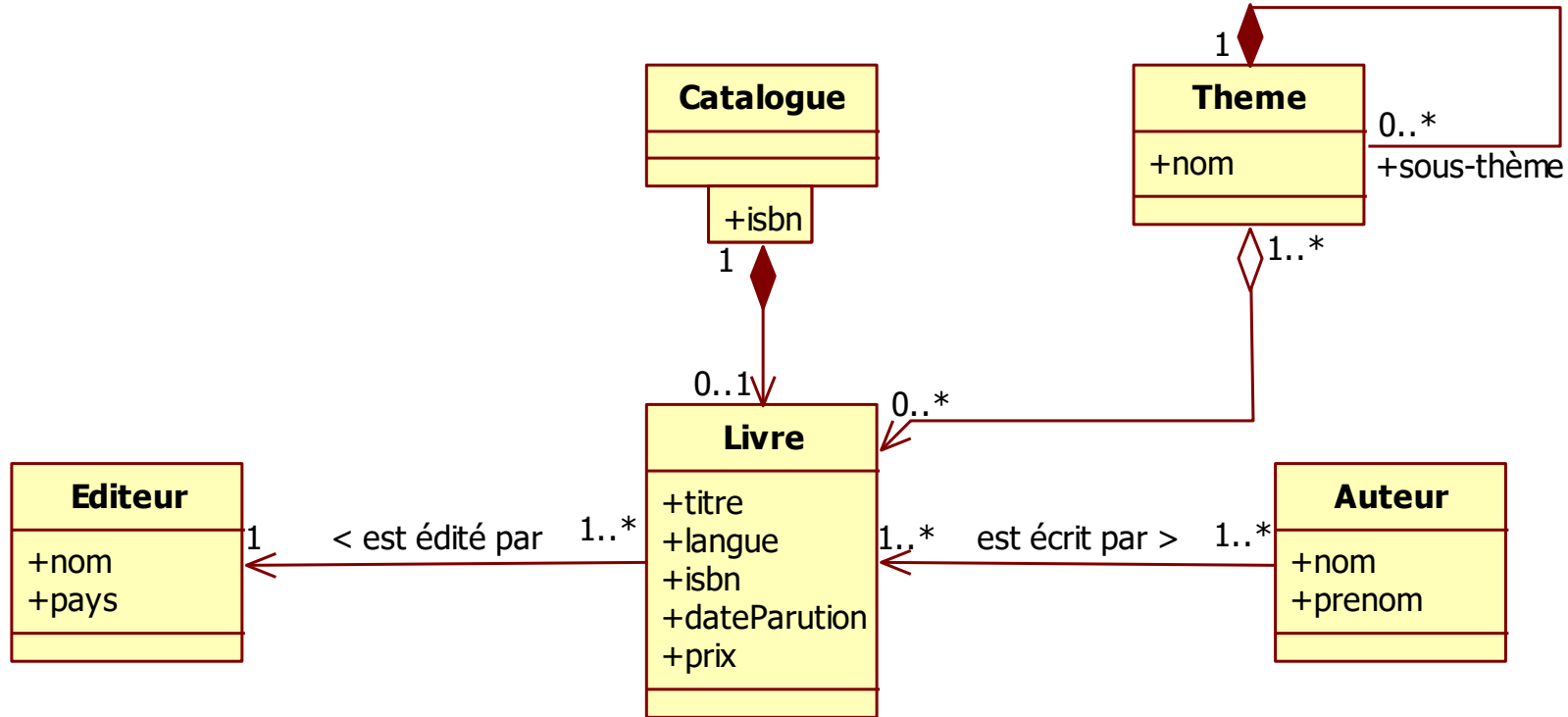
L'association entre les classes A et B  
est navigable de A vers B  
et indéterminée de B vers A



L'association entre les classes A et B  
est uniquement navigable de A vers B



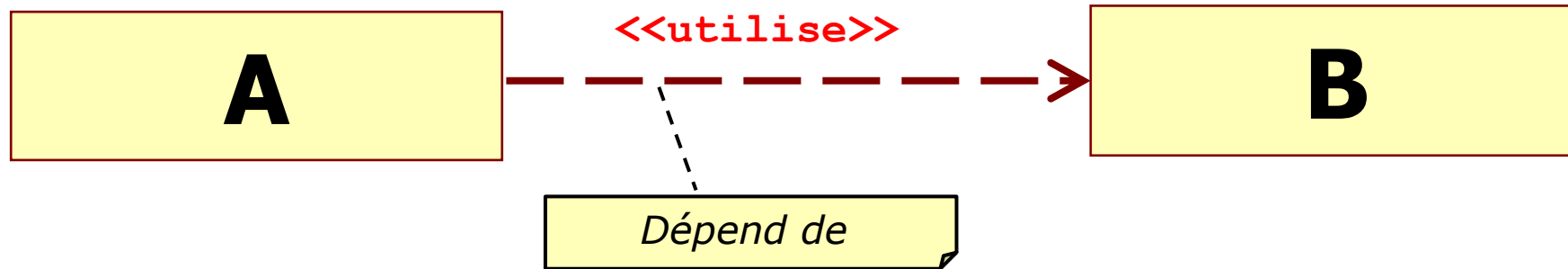
# Exemple de diagramme de classes illustrant les notions précédentes ...



C'est plutôt durant la **phase de conception** (à l'aide notamment du diagramme de séquences) que la **navigabilité** sera ajoutée car  
*limiter la navigabilité à une seule direction permet ainsi de réduire les **couplages** et les **dépendances** entre les classes*

# Dépendance Fonctionnelle — — — — — ➤

Une **dépendance** est une **relation unidirectionnelle** exprimant une **dépendance sémantique** entre des éléments du modèle.



La classe A **dépend de** la classe B si une méthode de A:

- possède un paramètre de type B
- ou*
- retourne une valeur de type B
- ou*
- utilise une variable locale de type B

La relation de dépendance est souvent stéréotypée pour mieux expliciter le lien sémantique entre les éléments du modèle. Il existe des stéréotypes prédéfinis comme :  
<<utilise>>, <<instancie>>, <<instance>>, ...



# Restriction d'une association via un qualificateur

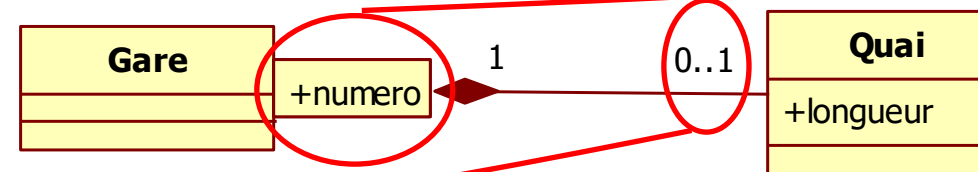
**Modélisons simplement la phrase suivante :**

Une **gare** ferroviaire contient des **quais** de *longueur* variable.  
Chaque *quai* est repéré par son *numéro*.



**Affinons notre raisonnement :**

Un `numero` permet **d'identifier de façon unique** un `Quai` dans une `Gare`.  
UML propose dans ce cas de modéliser `numero` comme un **qualificateur**.



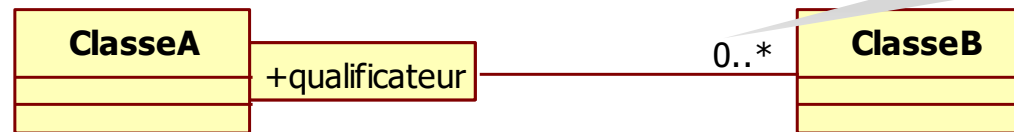
**Multiplicité forcément réduite du côté opposé au qualificateur !**

Etant donnés une instance de `Gare` et un `numéro`, il existe au plus 1 instance de `Quai` ayant ce `numéro` dans la gare donnée (au pire aucun quai n'a le numéro donné)  
⇒ dans une gare, deux quais ne peuvent pas porter le même numéro !

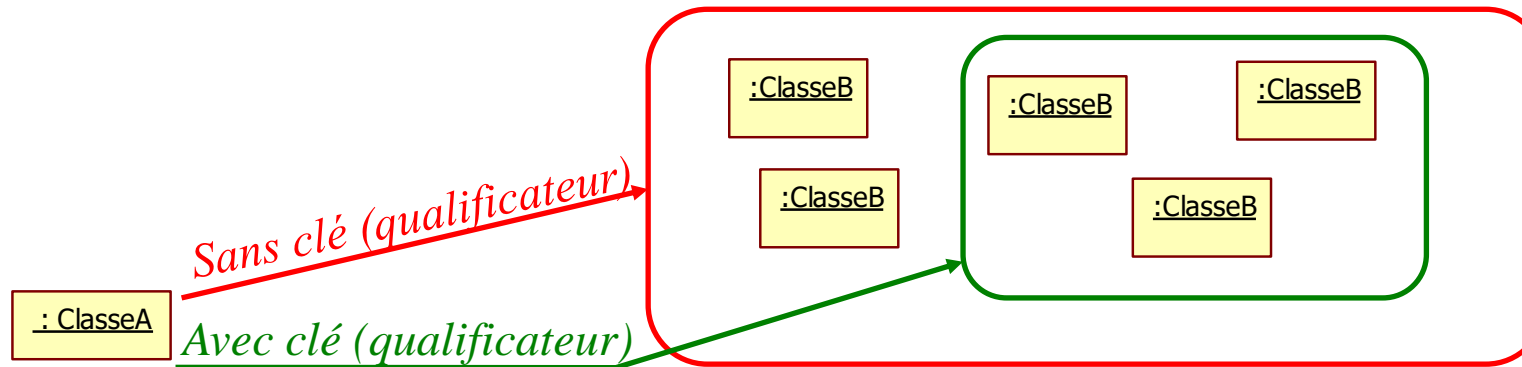
# Qualification, qualificateur : Définition

**La qualification d'une association** (ou **restriction** d'une association) permet de réduire la multiplicité \* ou 1..\* d'un rôle en utilisant un identifiant propre à l'association.

*multiplicité maximale pas forcément limitée à 1 mais réduite par rapport à celle de l'association initiale*



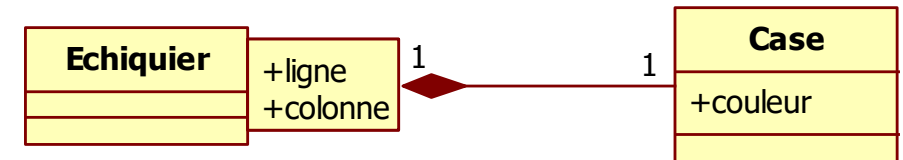
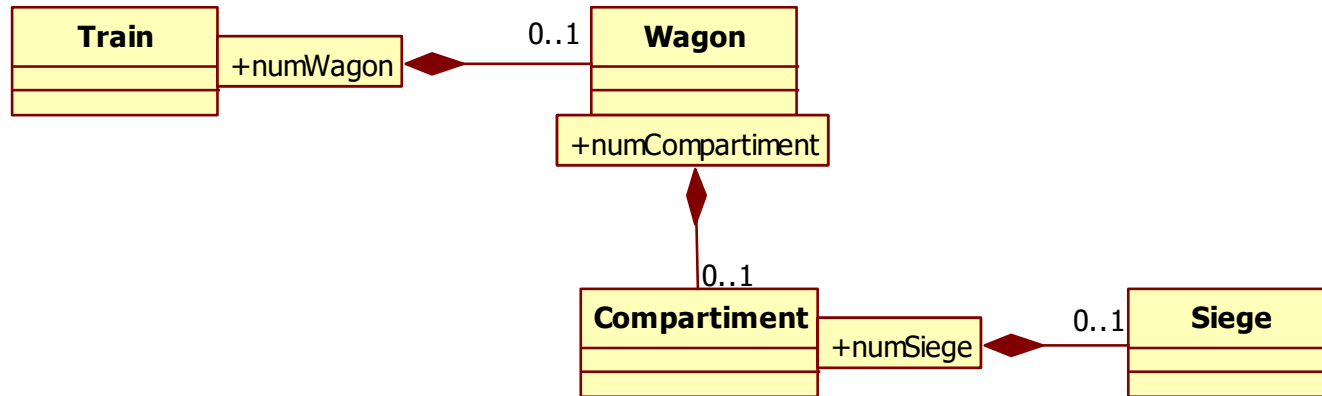
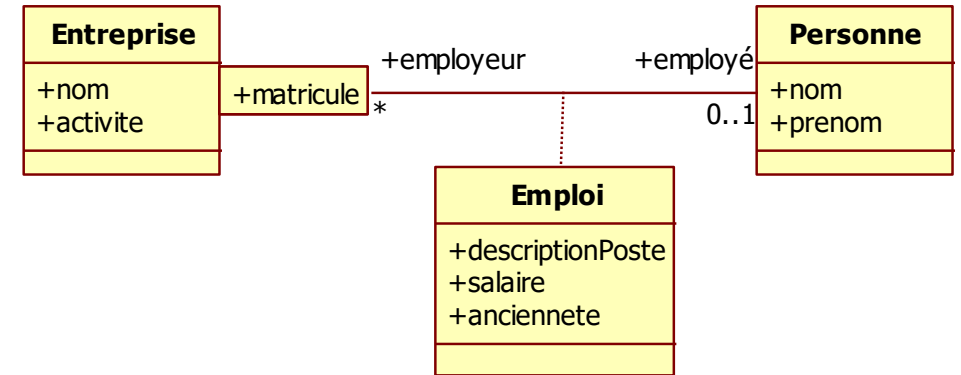
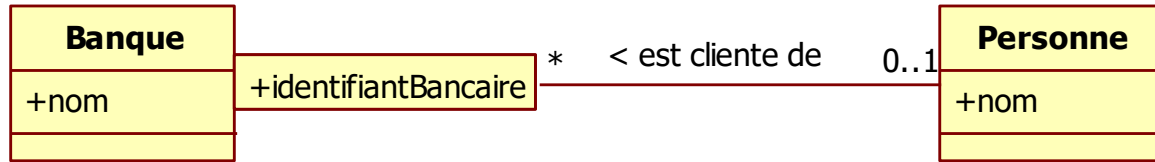
Cet identifiant (clé), appelé **qualificateur**, est placé sur l'extrémité de l'association au niveau de la classe source (*dans le contexte de ...*)



La paire (instance de *ClasseA*, qualificateur) identifie **un sous-ensemble des instances** de *ClasseB*

*La qualification permet de partitionner l'ensemble d'arrivée (un sous-ensemble d'objets) réduisant ainsi la multiplicité.*

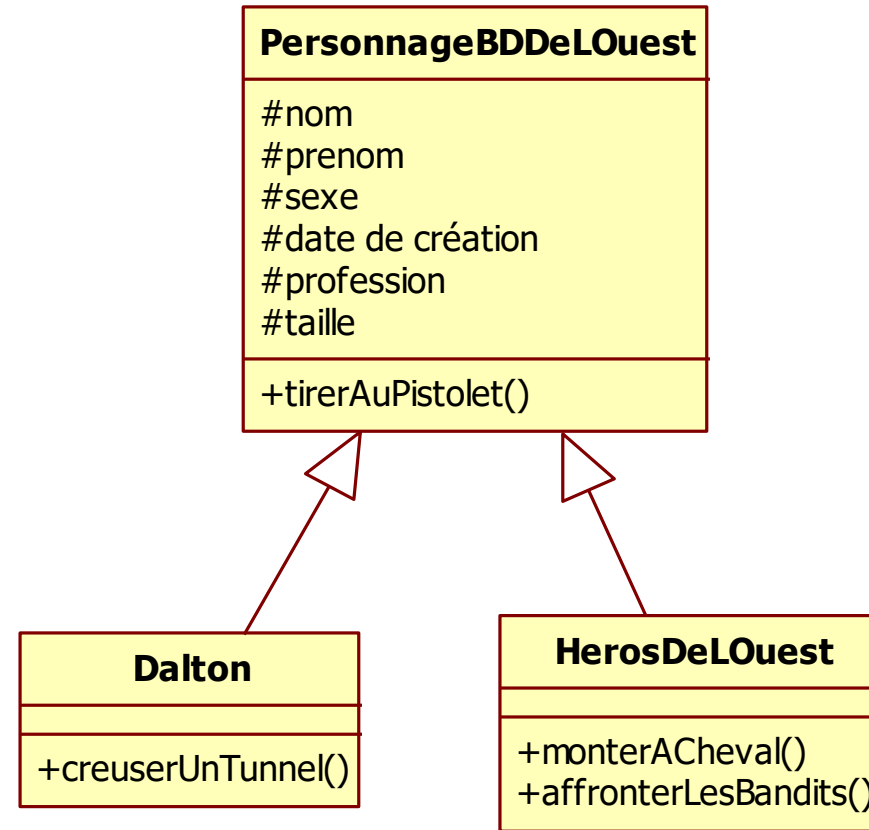
# Qualificateurs : exemples



# Relation de Généralisation

Remarque : En UML, la relation de généralisation n'est pas propre qu'aux classes.  
Elle s'applique à d'autres éléments du langage comme les paquetages,  
les acteurs ou les cas d'utilisation.

# Héritage : Notation UML



La notation UML pour **l'héritage de classe** est :

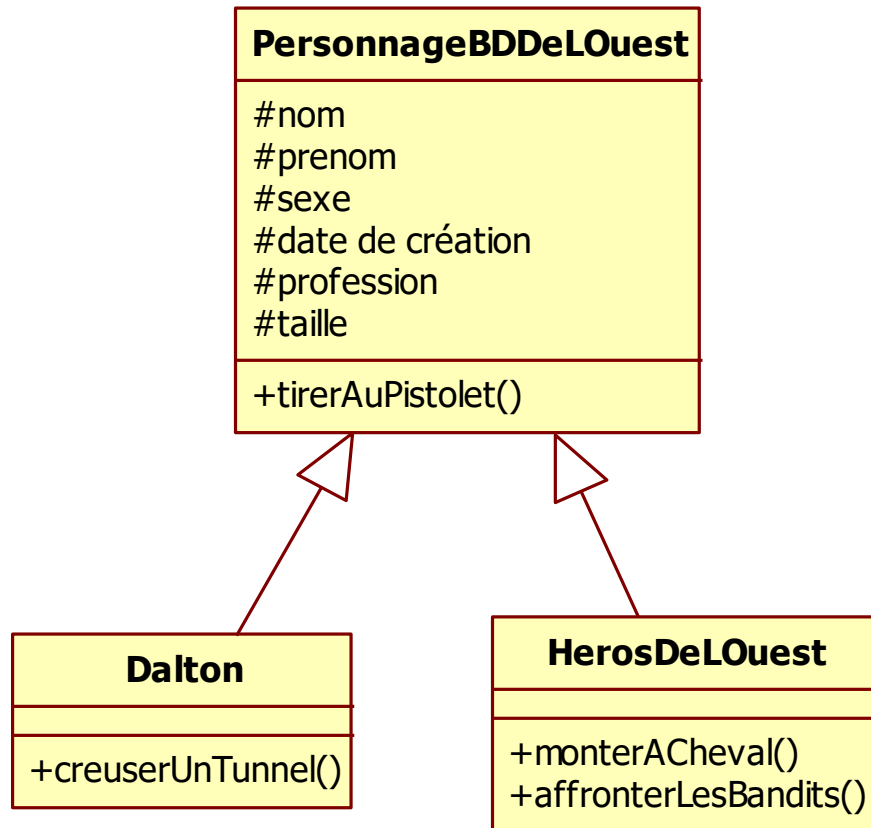
⇒ C'est une relation de **généralisation** qui exprime une relation **EST-UN** (extends)

Un **Dalton** **EST-UN** **PersonnageBDDeLOuest**

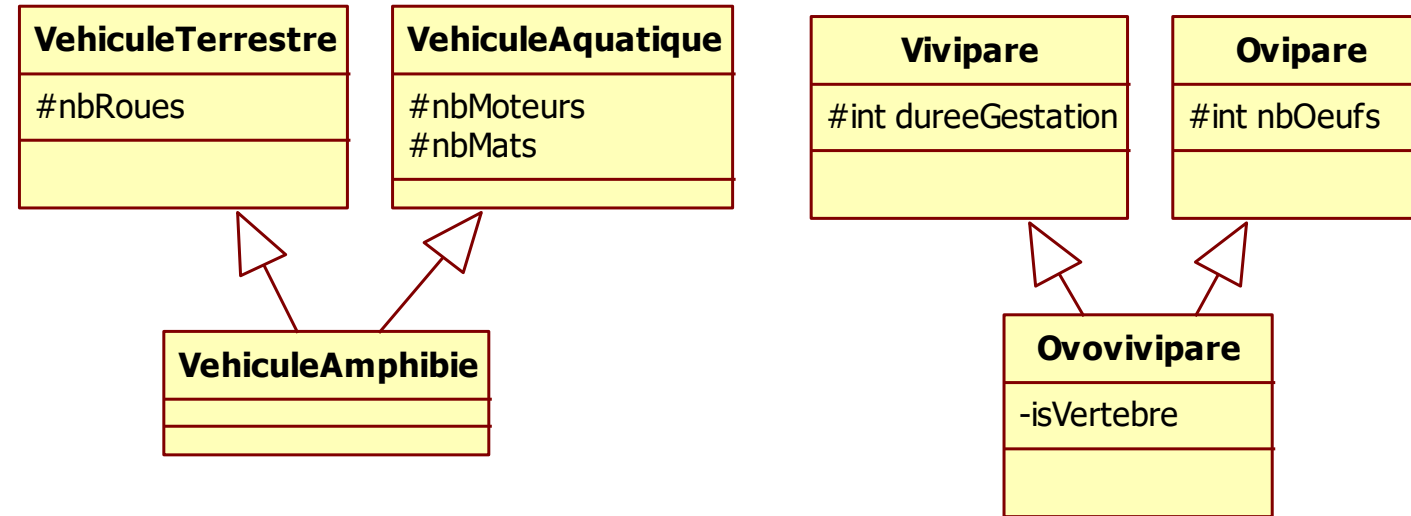
Un **HerosDeLOuest** **EST-UN** **PersonnageBDDeLOuest**

# Héritage Simple ou Multiple

## Héritage Simple



## Héritage Multiple

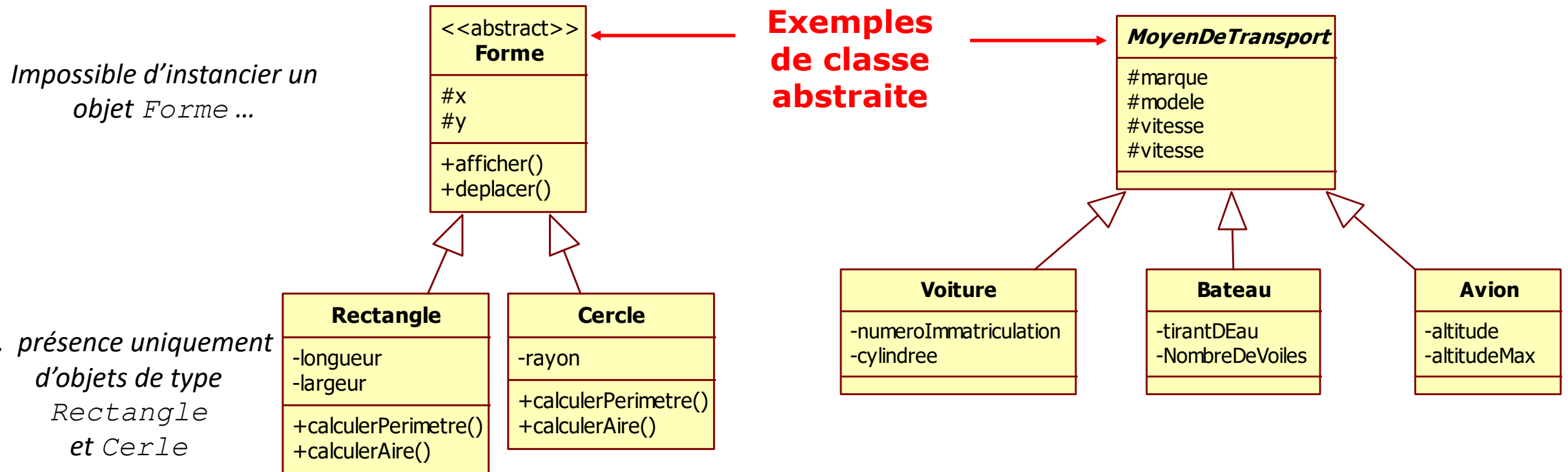


# Classe Abstraite

# Notion de classe abstraite

Une **classe abstraite** est une **classe non instanciable**

En UML, une **classe abstraite** se note ***en italique*** ou à l'aide du stéréotype **<<abstract>>**



Pour être utile, une classe abstraite doit admettre des classes descendantes concrètes (c-a-d instanciables)

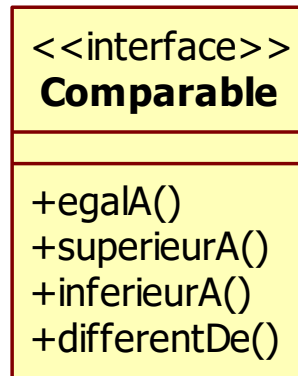


# Interface

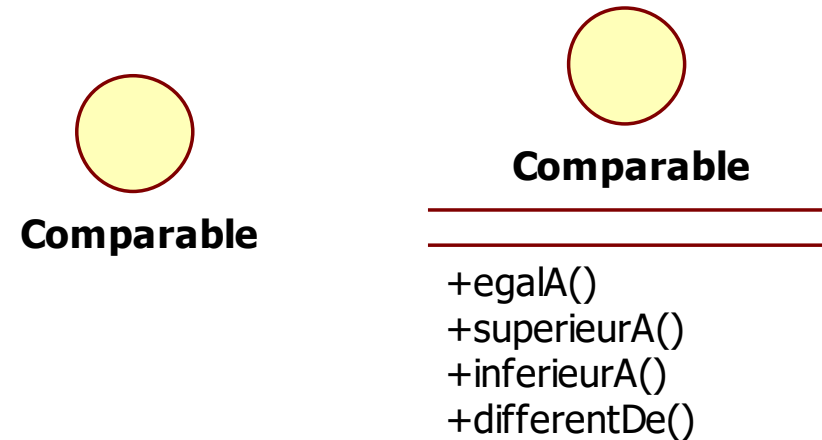
# Interface : Définition

Au sens UML, une **interface** est un **ensemble d'opérations utilisées pour spécifier les services (un contrat)** d'une classe ou d'un composant.

Notation avec le mot clé  
**<<interface>>**



Notation sous forme de **lollipop**  
(sucette)



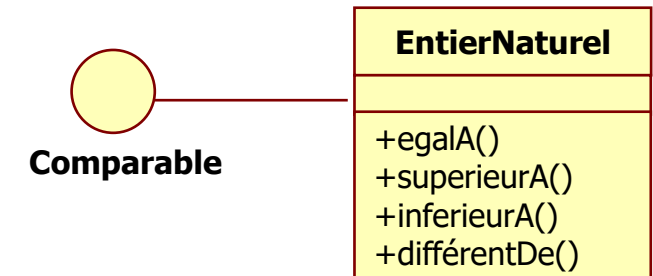
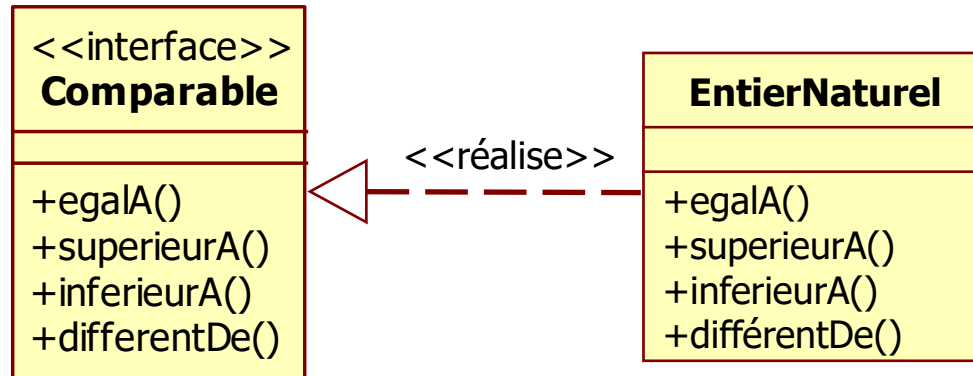
Structurellement, une interface ressemble à une classe avec le mot-clé **<<interface>>**  
Mais, à la différence d'une classe ...

- Une interface ne peut définir **ni attribut, ni association navigable vers d'autres classes.**
- De plus, toutes les opérations d'une interface sont abstraites

# Relation de réalisation (implémentation)



Une interface est destinée à **être "réalisée"** (ou implémentée) par au moins une classe qui va hériter de toutes les descriptions et **concrétiser les opérations abstraites** (*remplir le contrat*)



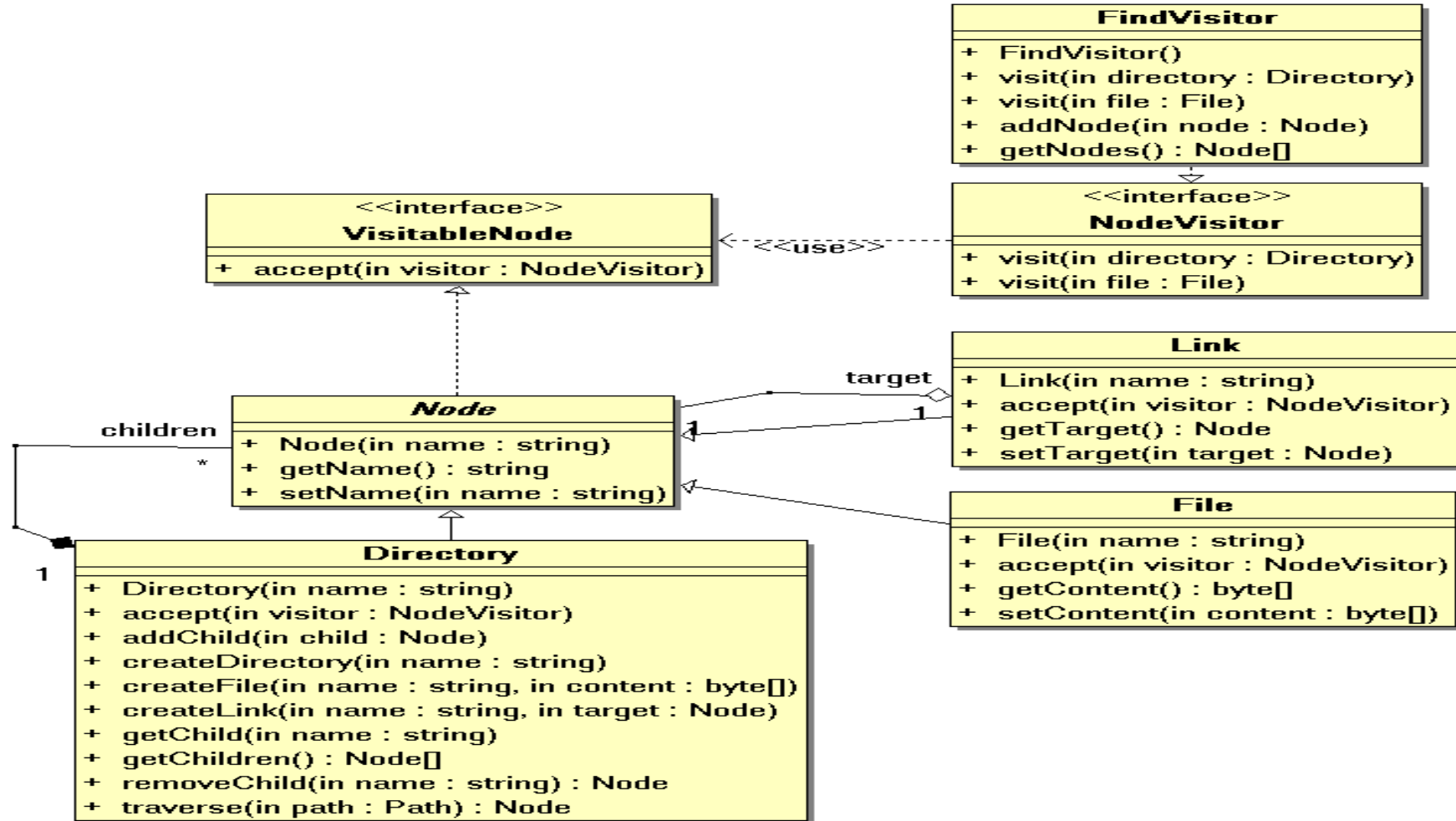
Notation classique : **flèche en pointillés**  
avec la présence ou non du stéréotype `<<réalise>>`

Notation avec la lollipop : **trait continu**

→ Une classe **réalise** (implémente) une interface si elle est capable d'exécuter toutes les opérations de l'interface

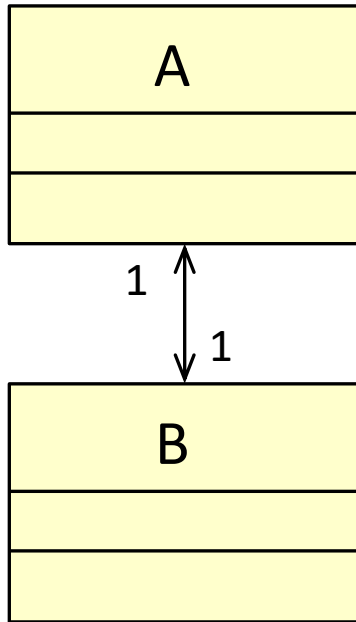
→ Une classe peut très bien réaliser plusieurs interfaces.

# Exemple de diagramme de classes illustrant les notions précédentes ...



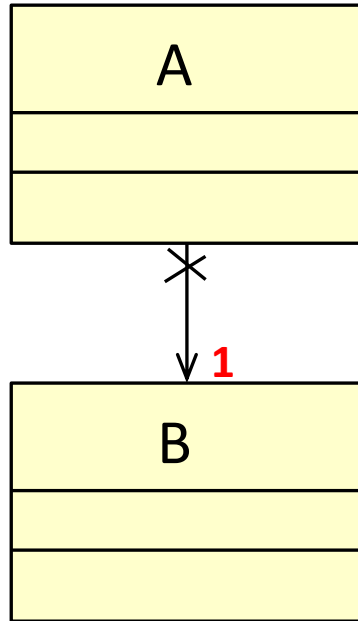
# **De la modélisation UML au code Java**

# Association bidirectionnelle



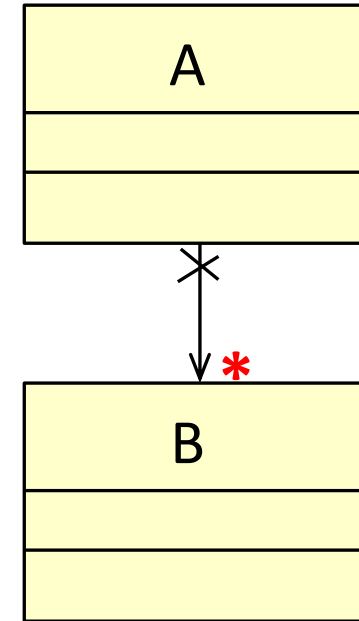
```
class A {  
    private B b ;  
    ...  
}  
  
class B {  
    private A a ;  
    ...  
}
```

# Navigabilité & Multiplicité



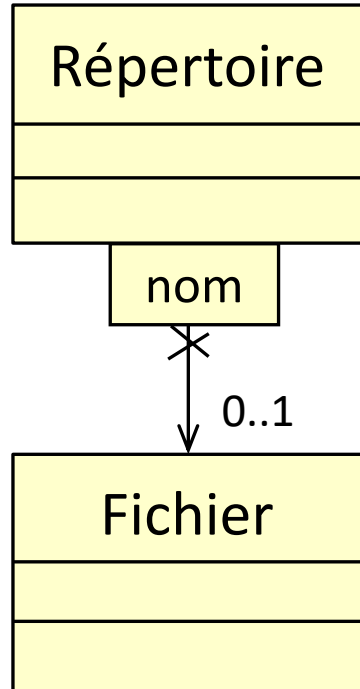
```
class A {  
    private B b ;  
    ...  
}  
  
class B {  
    ...  
}
```

*Avec la navigabilité,  
B ne voit plus A ...*



```
class A {  
    private Collection<B> b ;  
    ...  
}  
  
class B {  
    ...  
}
```

# Restriction d'association

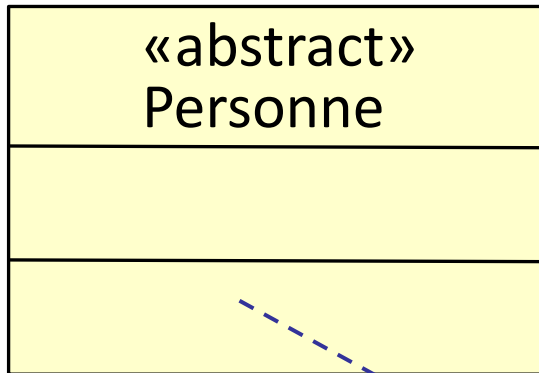


```
class Répertoire {
    private Map<String,Fichier> fichiers ;
    public void add (Fichier f) {...}
    public Fichier get (String nom) {...}
}

class Fichier {
    private String nom ;
}
```

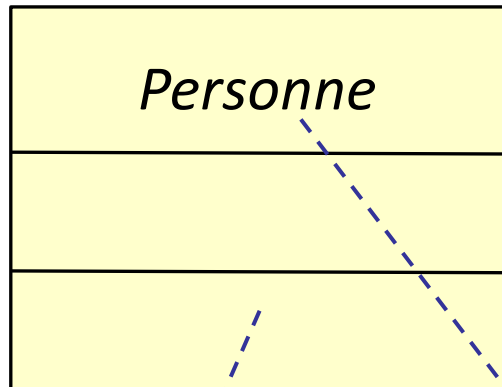


# Classe Abstraite (abstract)



Les méthodes *retardées* sont préfixées  
par le stéréotype «abstract»

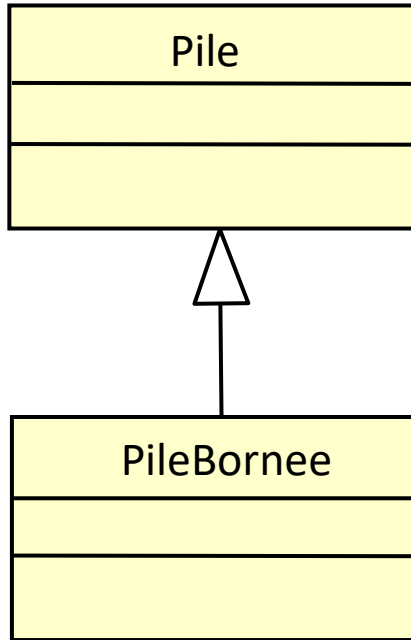
```
abstract class Personne {  
    ...  
    ...  
}
```



Méthodes retardées  
en italique

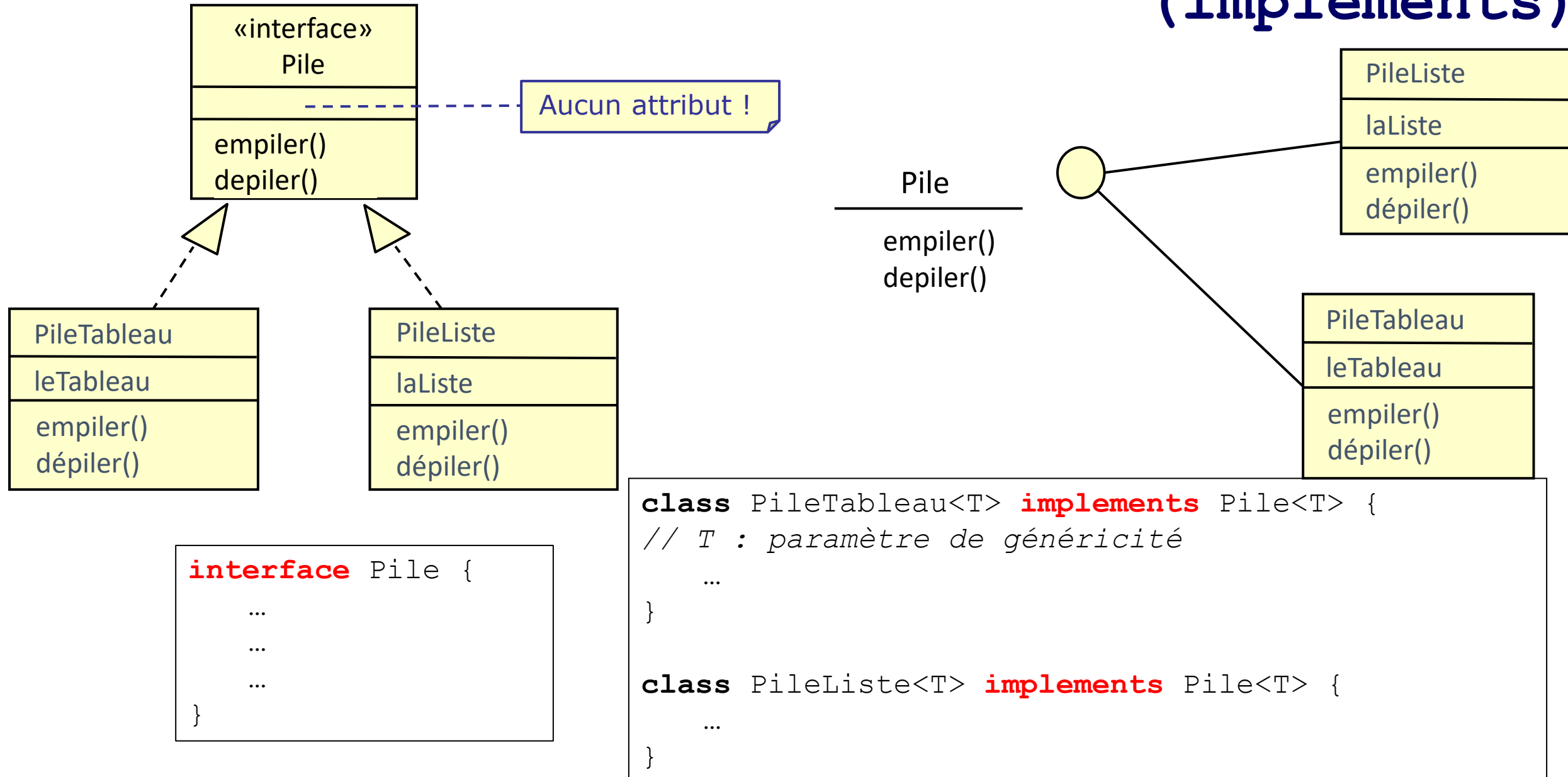
En italique

# Héritage (extends)

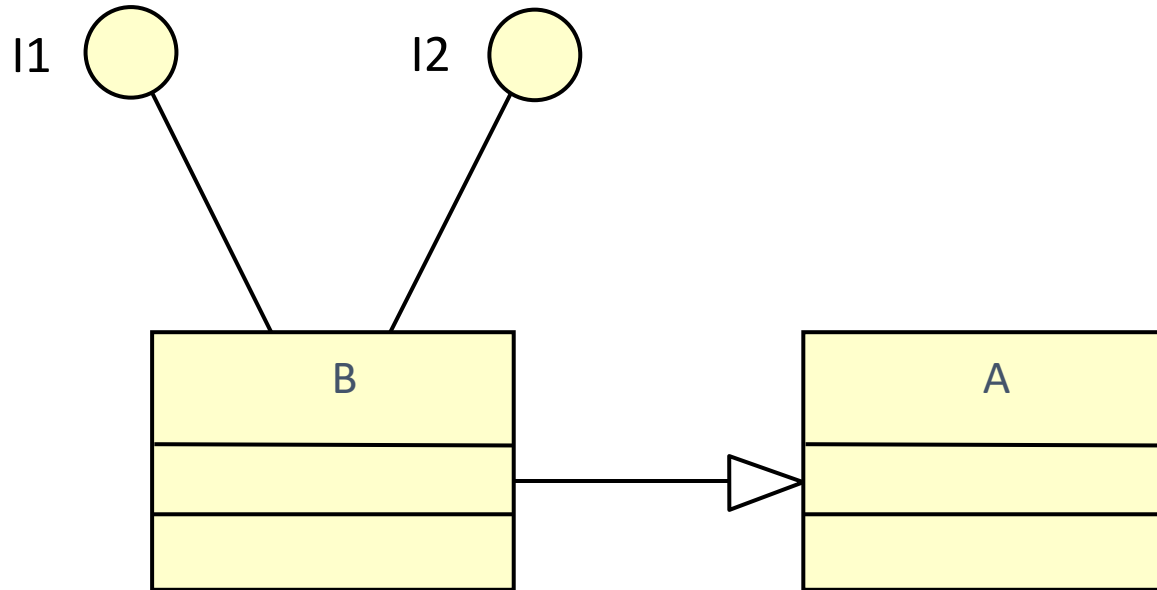


```
class Pile {  
    ...  
}  
  
class PileBornee extends Pile {  
    ...  
}
```

# Interface (interface) et ses réalisations (implements)



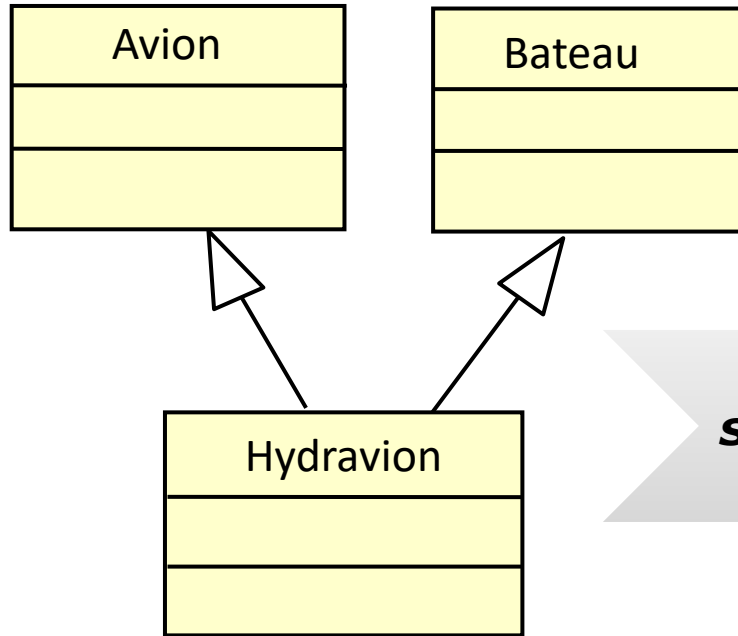
# Interface multiple et Héritage



```
class B extends A
    implements I1, I2 {
    ...
}
```

# Pas d'Héritage multiple en Java

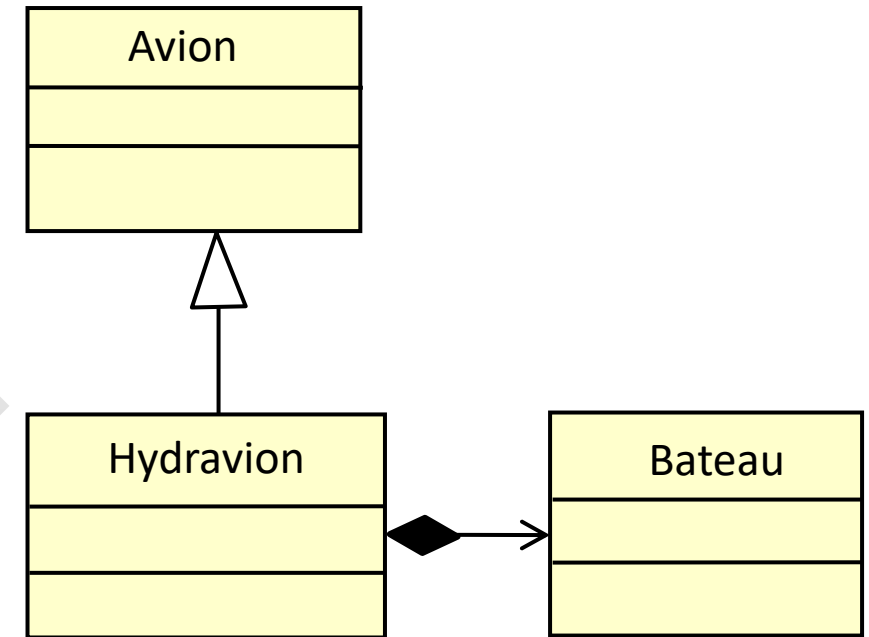
Un Hydravion **EST plutôt** considéré  
comme **UN Avion** qui  
**A** aussi les caractéristiques d'**UN Bateau**



## Héritage Multiple

Un Hydravion EST-UN Avion  
et EST-UN un Bateau

*Transformation de la  
sémantique pour permettre  
l'implémentation en Java*



**Héritage simple  
avec délégation** (une alternative possible à l'héritage multiple)

```
class Hydravion extends Avion {
    private Bateau flotteurs ;
    // pour se poser sur l'eau :
    // délégation de la responsabilité à la classe Bateau
    ...
}
```

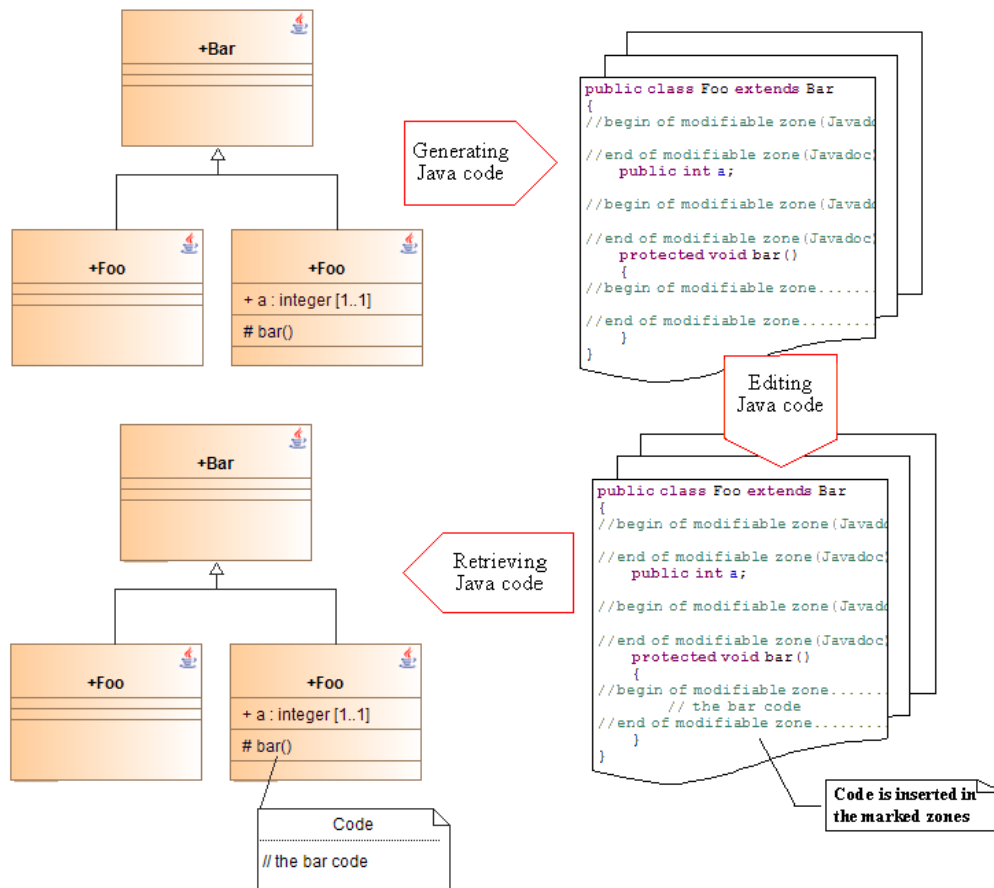
**Modéliser, pour concevoir ...**

# Le diagramme de **classes** : au cœur de la conception orienté **objet**

Donne un **point de vue** sur  
la manière dont est « agencé »  
le **code** (architecture)



Aide à la mise en place et/ou compréhension  
de la Conception du système



**Model-Driven**  
*(top-down)*

*Des modèles au code ...*

Un outil pour  
mettre en œuvre  
l'implémentation  
d'un système

**Reverse-Engineering**  
sur du legacy

*Du code aux « modèles »*

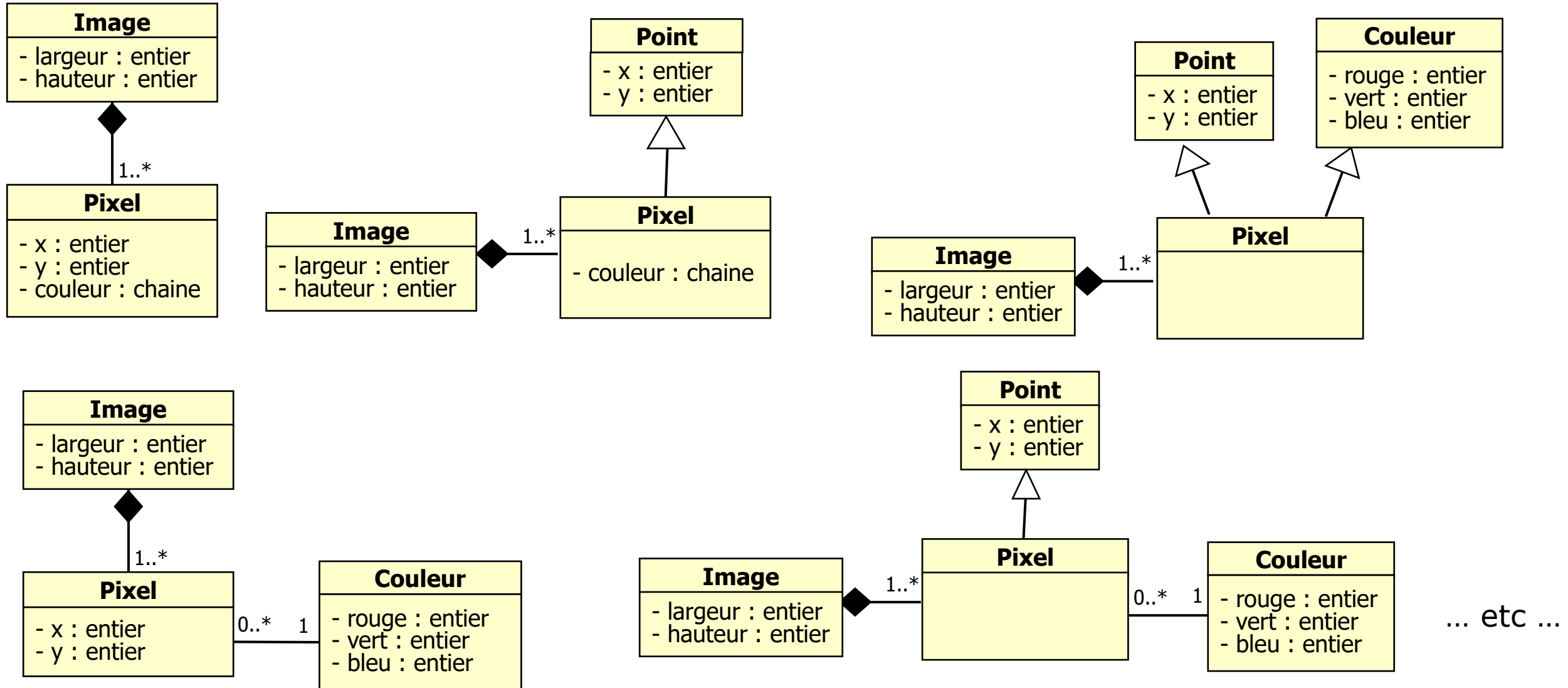
Un outil pour mieux  
comprendre et  
appréhender  
le design d'un  
système existant

**Concevoir, c'est faire des choix ...**



# Faire des choix ...

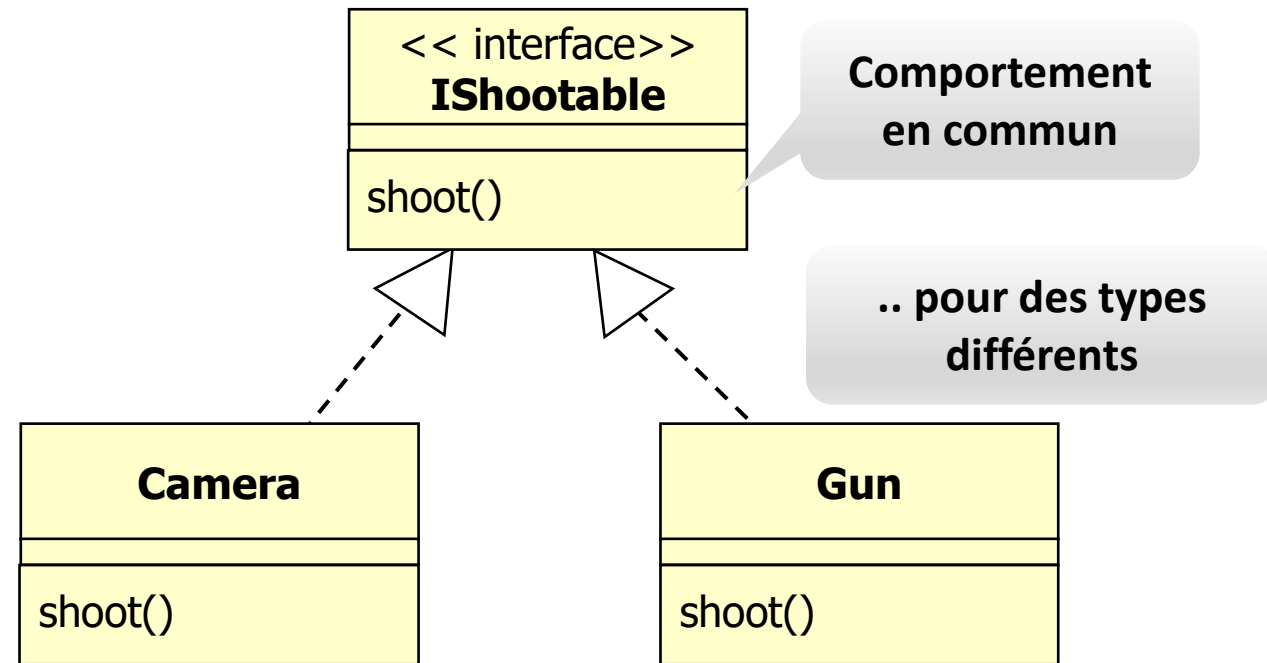
Un pixel (*picture element*) est un point coloré d'une image



# Interface vs Classe Abstraite

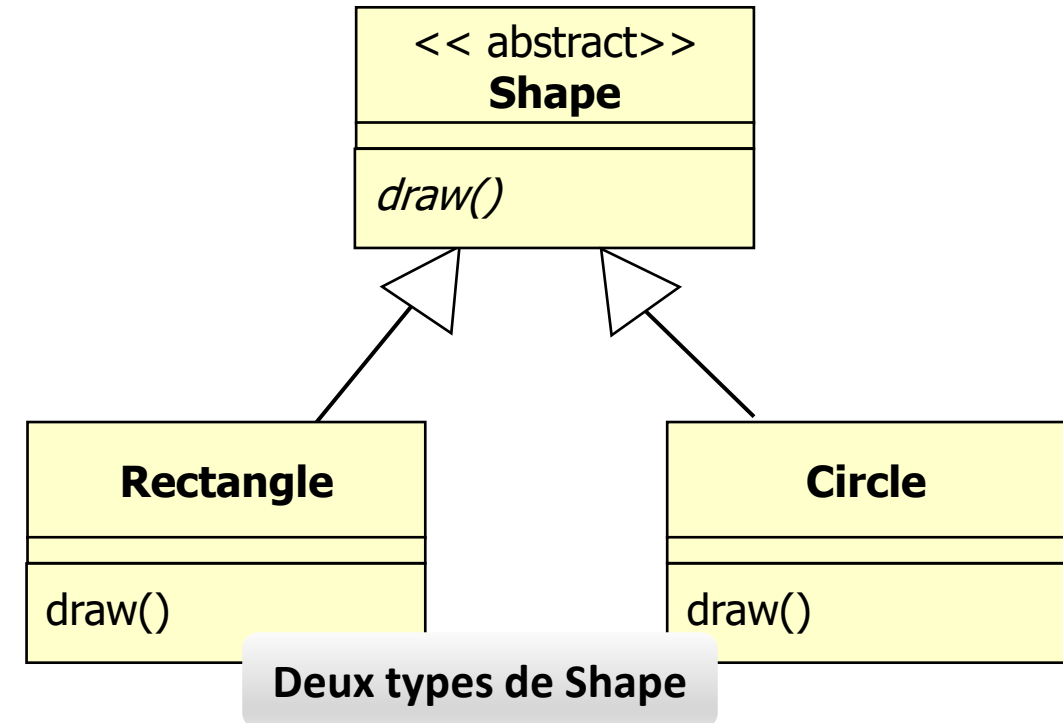
**Interfaces** represent  
*capabilities*

Implementing interfaces  
represents **can-do** relationship



**Abstract classes**  
represent *type*

Inheriting from (abstract) class  
represents an **is-a** relationship



# Concevoir, c'est faire des choix ...

- Choix sur l'abstraction des concepts identifiés
- Choix sur le nommage
- Choix sur la multiplicité
- Choix sur la mise en place de nouvelles classes/interfaces pour réduire le couplage et les dépendances entre classes
- Choix sur l'architecture  
(pour une forte cohésion et un faible couplage par exemple)
- ...

# Évolution des architectures logicielles ....

Ahaa : “The evolution of  
[#SoftwareArchitecture](https://twitter.com/HerveLourdin/status/610418775964418048/photo/1)” [bit.ly/1JPD1k1](https://bit.ly/1JPD1k1)

 Voir la traduction

## THE EVOLUTION OF SOFTWARE ARCHITECTURE

### 1990's

SPAGHETTI-ORIENTED  
ARCHITECTURE  
(aka Copy & Paste)



### 2000's

LASAGNA-ORIENTED  
ARCHITECTURE  
(aka Layered Monolith)



### 2010's

RAVIOLI-ORIENTED  
ARCHITECTURE  
(aka Microservices)



### WHAT'S NEXT?

PROBABLY PIZZA-ORIENTED ARCHITECTURE

# Subjectivité de la modélisation

L'activité de modélisation a un caractère **hautement subjectif**, le choix est souvent difficile à faire entre **simplicité** et **évolutivité**...

→ **Un modèle très compact, simple** à implémenter sera **peu évolutif** lorsque de nouvelles demandes émaneront des utilisateurs ...

→ **Un modèle très souple, mais plus complexe** à implémenter résistera mieux à **l'évolution des besoins des utilisateurs** ...

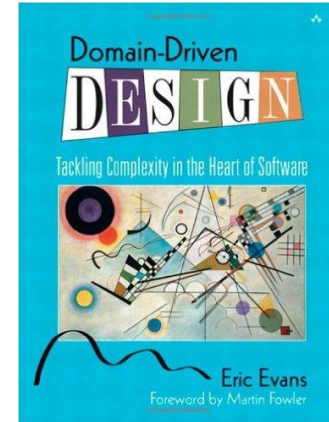
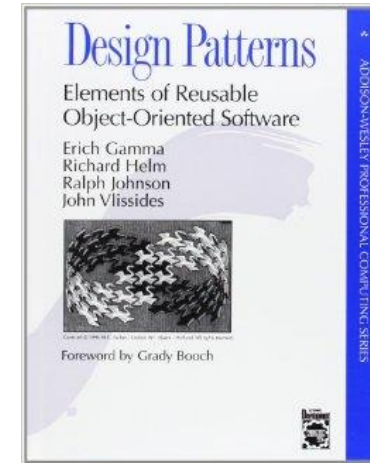
... Le choix entre les 2 solutions doit se faire **en fonction du contexte** :  
faut-il privilégier la simplicité, les détails de réalisation  
ou au contraire la pérennité et évolutivité ? ...

# Et pour vous aider dans vos choix : Tout un ensemble de bonnes pratiques de conception à (re)découvrir...

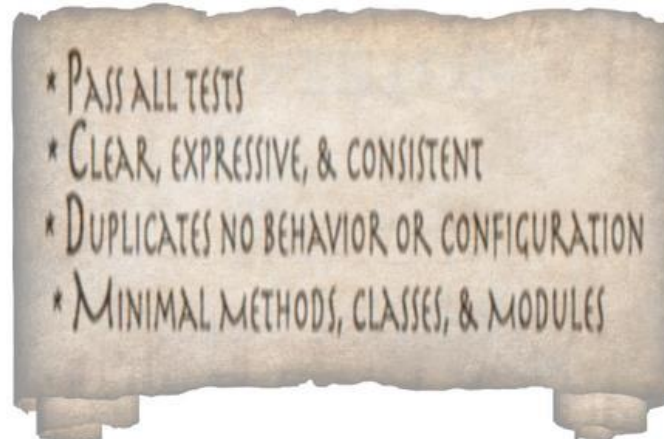
★ Teasing ★ @



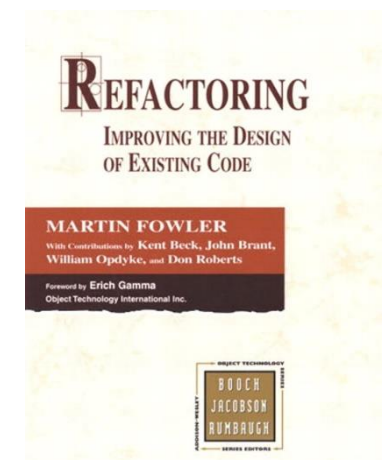
**GRASP**  
General Responsibility Assignment  
Software Patterns



**DRY (Don't Repeat Yourself)**



... et d'autres ...

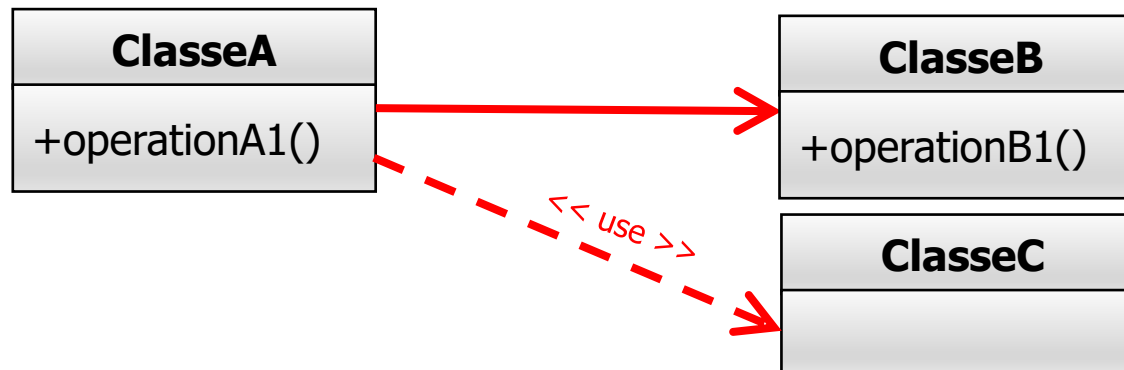


# **Annexes**

# Association vs Dépendance

**Dans une association navigable entre les ClasseA et ClasseB,**  
les objets de ces classes peuvent communiquer.

*Une instance de la ClasseA peut envoyer un message à une instance de la ClasseB  
c-a-d une instruction du type `monObjetB.operation1()` ;*



**Une dépendance entre les ClasseA et ClasseC.**

les objets de ces deux classes ne communiquent pas directement.







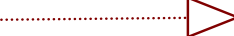
*Un objet de la ClasseC peut par exemple être passé en paramètre dans une opération de la ClasseA  
c-a-d une instruction du type `monObjetA.operationA2(monObjetC)` ;*

La dépendance permet d'affiner une relation entre classes.  
C'est une relation **non structurelle** existant entre plusieurs éléments.

- Un **lien durable** entre éléments va donner lieu à une **association** entre classes
- Un **lien temporaire** va donner lieu à une **dépendance**



# Affiner les relations UML : Récapitulatif

<b>Relation d'association</b>		relation entre deux classes ou plus, qui décrit les <b>connexions structurelles</b> entre leurs instances. Les classes ont le même niveau conceptuel
<b>Relation d'agrégation</b>		forme spéciale d'association exprimant une relation <b>collection/éléments</b>
<b>Relation de composition</b>		cas particulier d'agrégation ( <b>Composé/composant</b> ). La classe ayant le rôle prédominant est la classe <b>composé</b> .
<b>Relation d'association navigable</b>		<b>la navigabilité</b> indique s'il est <i>possible de traverser une association</i>
<b>Relation de dépendance</b>		relation unidirectionnelle indiquant que la cible dépend de la source. Relation non structurelle
<b>Relation de généralisation</b>		relation d'héritage <b>EST-UN</b> entre une classe générale et une classe spécialisée.
<b>Relation de réalisation</b>		relation d'héritage entre une interface et la classe qui réalise cette interface.

# Stéréotype

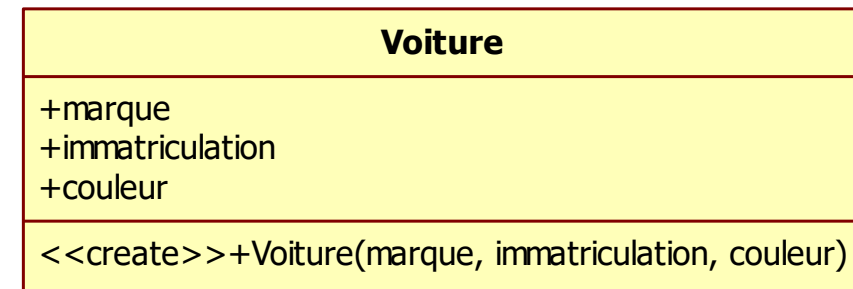
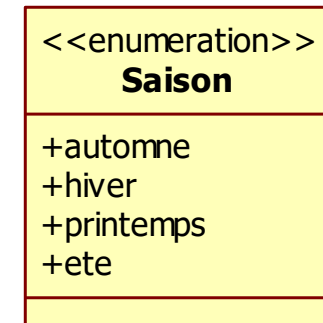
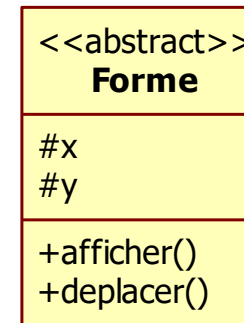
Un stéréotype est une chaînes de caractères entre guillemets  
**<<unStéréotype >>**

Le **stéréotype** est un **mécanisme d'extensibilité** prévu par UML.  
Il permet **d'étendre la sémantique** des éléments de modélisation en créant des types de base, semblables à ceux existant déjà (classe, relation, ...), mais spécifiques au traitement d'un problème donné

## Exemples de stéréotypes prédéfinis en UML :

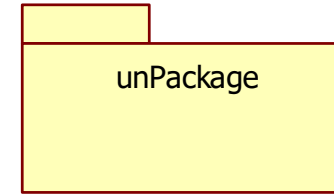
→ Dans le diagramme de classes,  
un stéréotype permet d'étendre des classes déjà existantes  
en leur donnant une **signification sémantique différente**.

→ Pour préciser qu'une opération est  
un constructeur ou un destructeur, on utilise  
les stéréotypes **<<create>>** ou **<<destroy>>**



# Notion de package

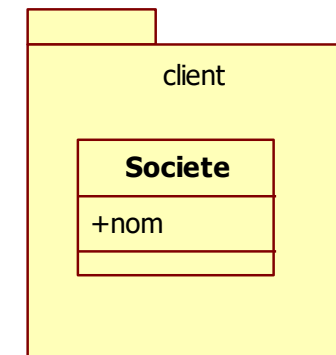
Le **paquetage** (**package**) est un mécanisme général de **regroupement d'éléments UML**



→ Un package peut contenir : des éléments d'un modèle, d'autres packages, des diagrammes qui représentent les éléments du modèle

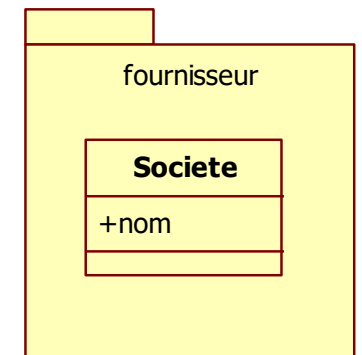
→ Un paquetage définit un **espace de nom** :

- ⇒ 2 éléments ne peuvent pas porter le même nom au sein du même package
- mais*
- ⇒ 2 éléments appartenant à des packages différents peuvent porter le même nom



client.Societe.nom

≠

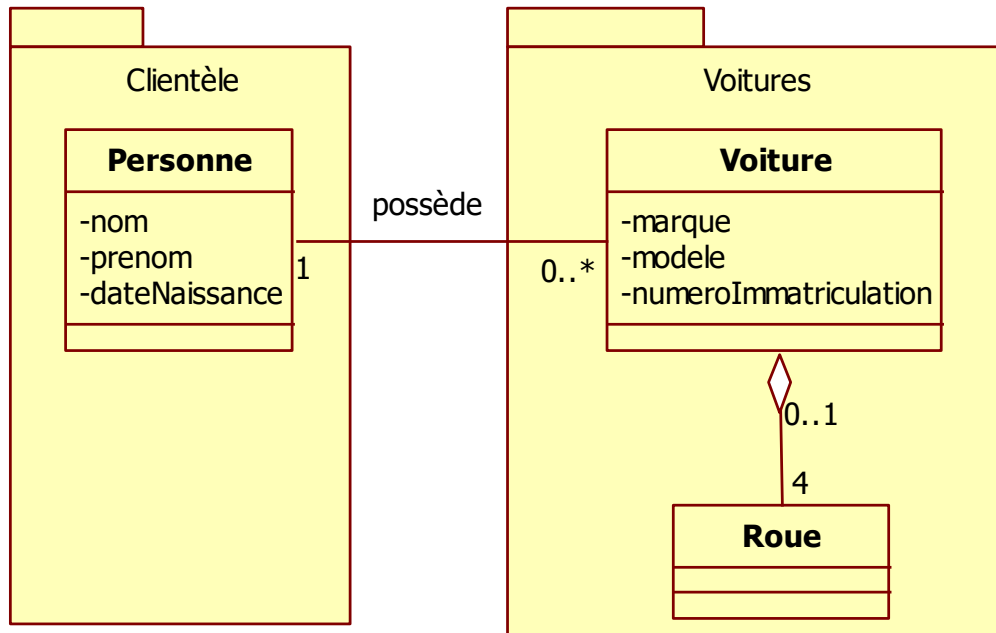


fournisseur.Societe.nom

→ La notion de paquetage est purement **organisationnelle**. Elle ne possède aucune sémantique dans la définition du modèle.

# Organisation des classes en package

Les **paquetages** servent à organiser un modèle objet en domaines et sous-domaines de la même manière que les répertoires organisent les systèmes de fichier.

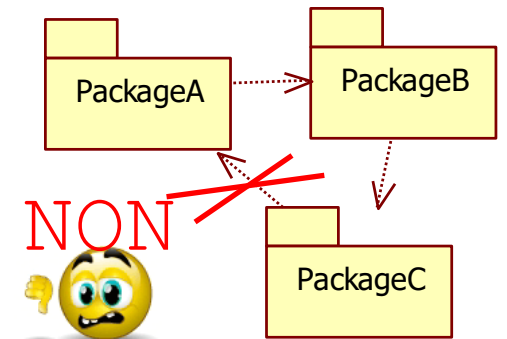
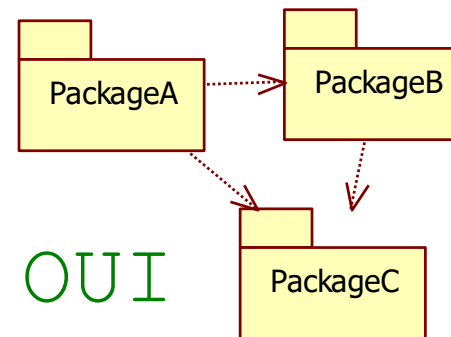
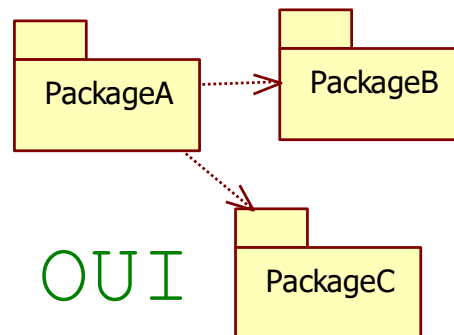


La structuration en **package** s'appuie sur deux principes fondamentaux :

→ **principe de cohérence** : qui consiste à regrouper des classes qui sont proches d'un point de vue sémantique (**forte cohésion**)

→ **principe d'indépendance** : qui s'efforce à minimiser les relations entre classes de packages différents (**couplage faible**)

Pour réduire le couplage, mieux vaut éviter les dépendances circulaires et transitives entre paquetages



# A propos des contraintes ...

Les **contraintes** sont des expressions qui précisent **le rôle** ou **la portée d'un élément de modélisation** (elles permettent d'**étendre** ou **préciser** sa **sémantique**).

→ UML propose, via ses éléments de modélisation, des **contraintes de base**. Par exemple, la **multiplicité** est une contrainte sur le nombre de liens qui existent entre 2 objets, le **type d'un attribut**, la **navigabilité**,... sont d'autres contraintes.

→ ... Mais, pour affiner la modélisation, UML permet aussi d'exprimer **ses propres contraintes** :

- soit en **langage naturel** (le plus souvent)
- soit à partir du **langage OCL** (**Object Constraint Language**), langage formel pour l'expression des contraintes standardisé par l'OMG et faisant partie de la norme UML qui vise à supprimer toute ambiguïté en se basant sur une grammaire précise

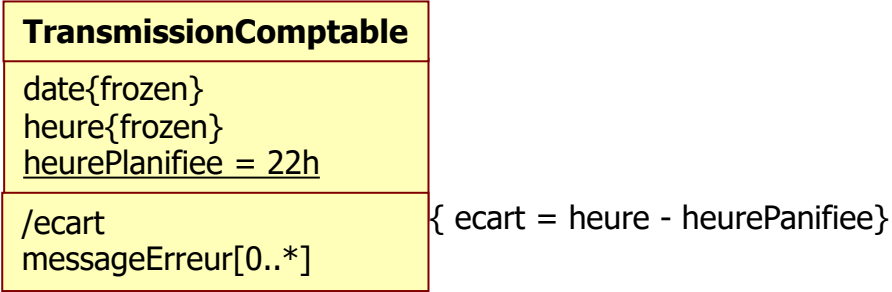
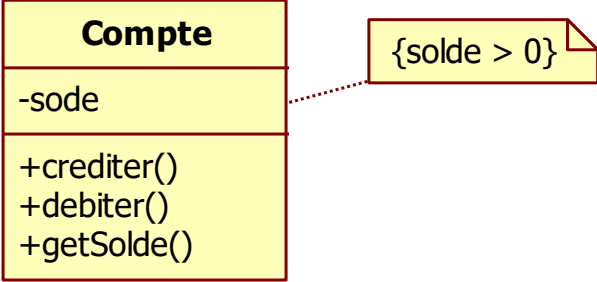
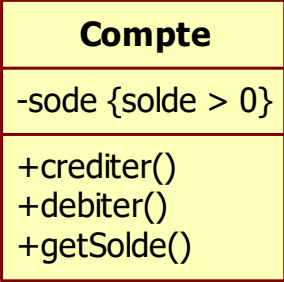
Les contraintes sont représentées par des expressions placées entre accolades

{une contrainte dans une note}

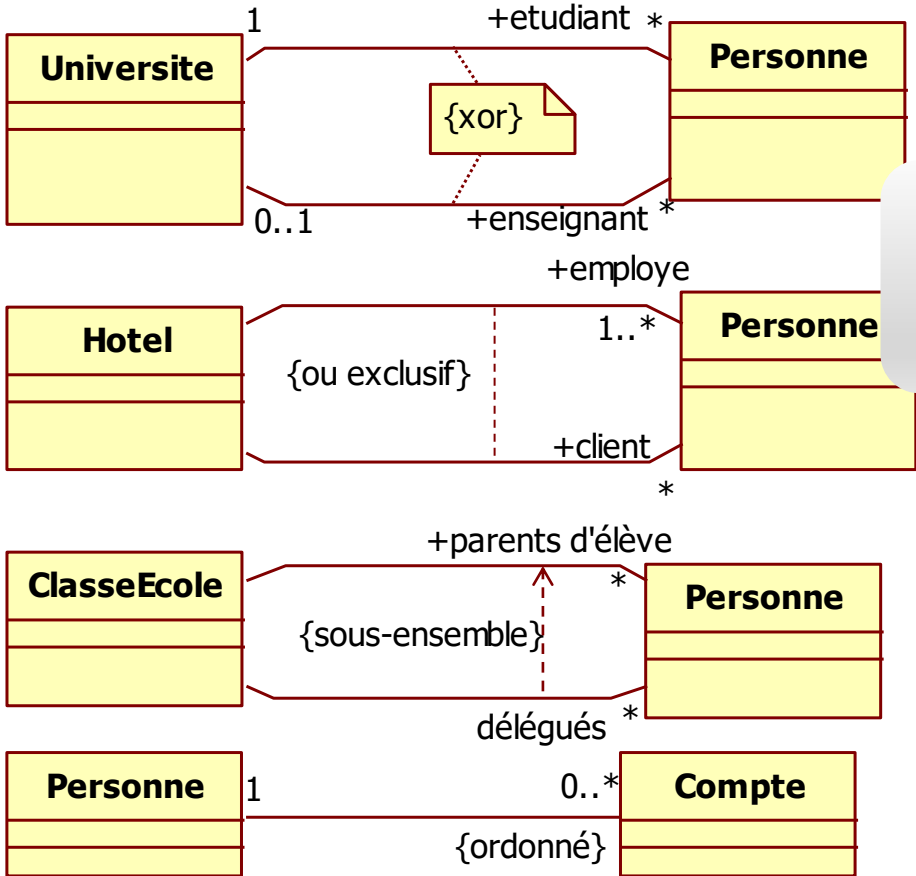
**{une contrainte}**

# Exemples de contraintes (1/2)

## Sur des attributs



## Entre 2 associations



**{ou exclusif}** ou **{xor}** pour un objet donné, une seule association parmi un groupe d'association est valide.  
Les instances sont mutuellement exclusives

**{subset}** ou **{sous-ensemble}** une collection est incluse dans une autre collection.  
La dépendance indique le sens de la contrainte.

# Exemples de contraintes (2/2)

## Sur des relations de généralisation

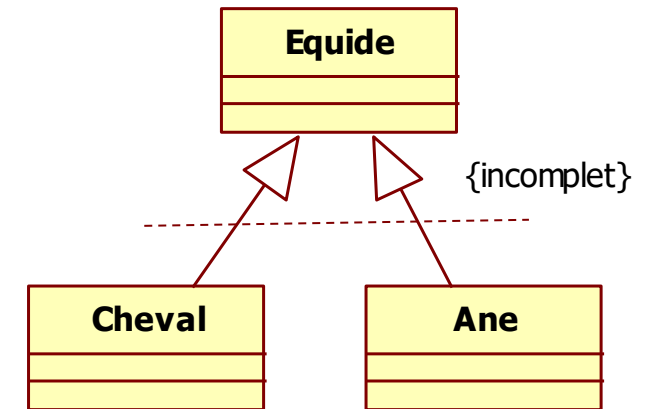
4 contraintes prédéfinies par UML entre une sur-classe et une sous-classe:

→ **{incomplete}** ou **{incomplet}**  
**ensemble des sous-classes incomplet**  
(ensemble des instances des sous-classes est un sous-ensemble des instances des surclasses)

→ **{complete}** ou **{complet}**  
**ensemble des sous-classes est complet**

→ **{disjoint}** (par défaut)  
**les sous-classes n'ont aucune instance en commun**

→ **{overlapping}** ou **{chevauchement}**  
**les sous-classes peuvent avoir une ou plusieurs instances en commun**

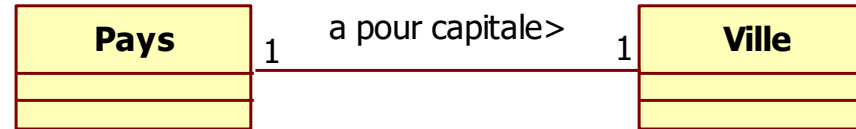


Ajouter des contraintes,  
mais ne pas en abuser !!

Les contraintes ne doivent pas combler des lacunes de modélisation!

# Diagramme de classes vs diagramme d'objets (1/2)

Les **diagrammes de classes** permettent de modéliser les **classes** et les **associations entre classes**.



Les **diagrammes d'objets** (ou diagrammes d'instances) permettent de modéliser les **instances** et les **liens entre instances**.

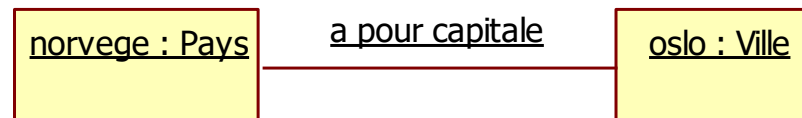


→ **un lien** représente une relation entre deux instances :  
c'est une connexion physique ou conceptuelle

→ un **lien** est une **instance d'association**

⇒ Graphiquement, s'il a un nom, il peut être souligné.

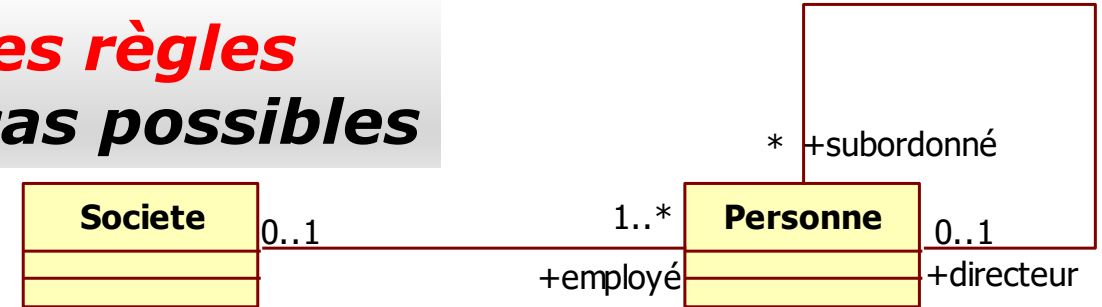
⇒ Les multiplicités ne sont pas représentées sur les diagrammes d'objets  
(*plusieurs objets ⇒ plusieurs liens...*)





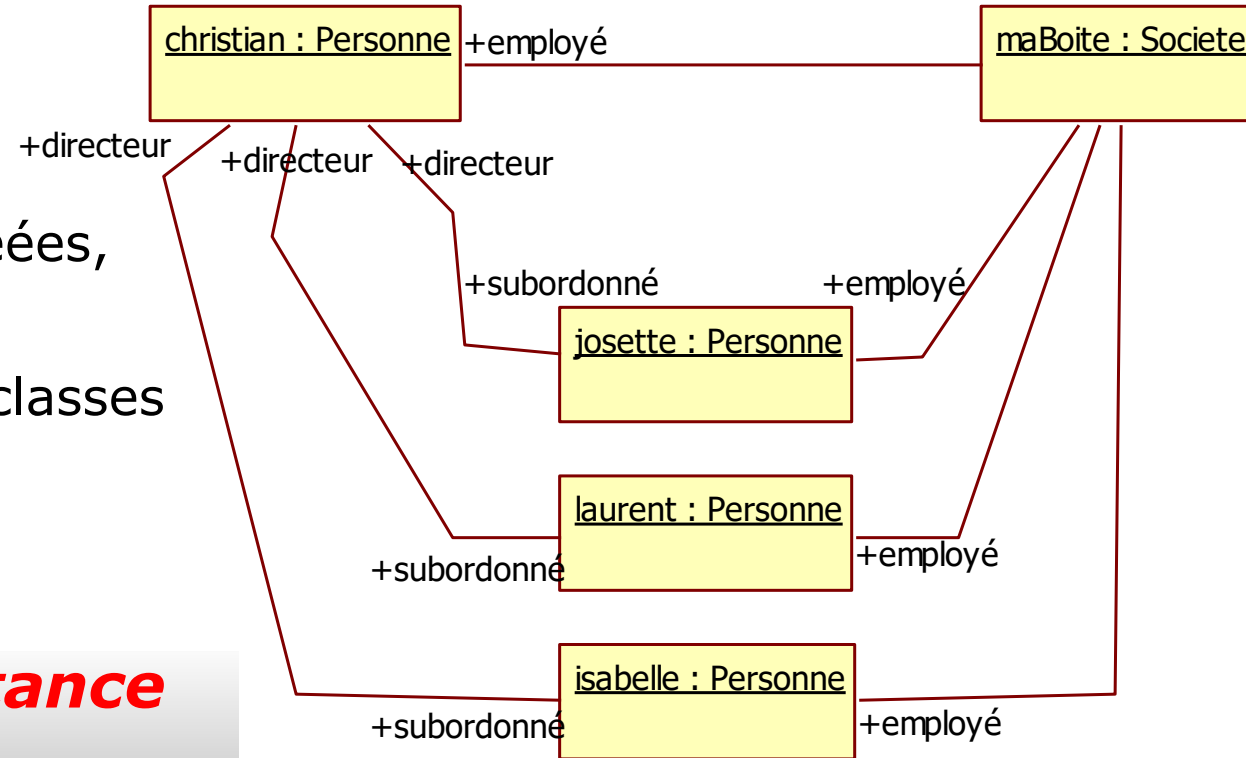
# Diagramme de classes vs diagramme d'objets (2/2)

**Le diagramme de classes** modélise **les règles** en représentant **l'ensemble de tous les cas possibles**



**Le diagramme d'objets** modélise **des faits**

- Il décrit, **à un instant t**, un état du système en montrant les instances créées, leur état et les liens entre elles.
- Il doit toujours rester conforme au modèle de classes



**Un diagramme d'objet est une instance d'un diagramme de classes.**

# Utilisation des diagrammes d'objets

Un **diagramme d'objets** s'utilise pour **montrer un contexte** (avant ou après une interaction entre objets par exemple).

Il peut être vu comme un **instantané**, une **photo** d'un sous-ensemble d'objets d'un système à un certain moment.

Il aide à **raisonner sur la base d'exemples** et est utilisé pour :

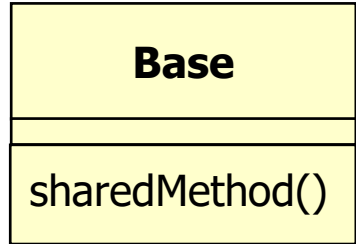
- expliquer un diagramme de classes (exemple simple ou cas particulier)
- valider un diagramme de classes (le "tester")

Un **diagramme d'objets** représente une **vue statique des instances** qui apparaissent dans les diagrammes de classes

Pour représenter une interaction, d'autres diagrammes à base d'objets seront utilisés lors de la **modélisation dynamique**

- **diagramme de séquence**
- **diagramme de communication (2.0) (collaboration en 1.0)**
- **diagramme d'états**

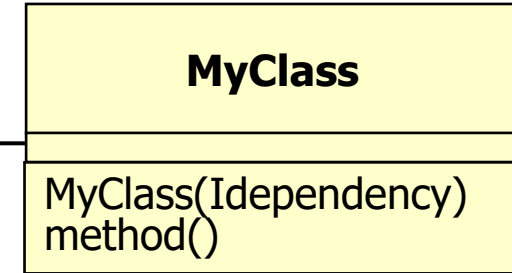
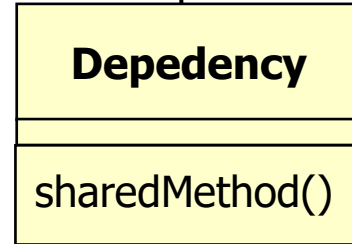
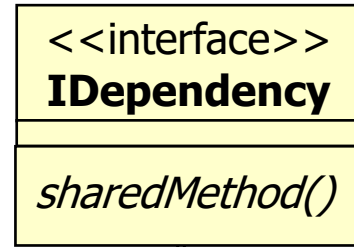
# Héritage vs Composition/Interface



```
public void method() {
    //sharedMethod
    // is available here
}
```

→ Le code que l'on souhaite réutiliser (`sharedMethod`) sera uniquement disponible dans une classe dérivée.

→ pas facile à tester : si `SharedMethod` appelle une BD et qu'un test unitaire doit être écrit sur `MyClass` (pas évident de *simuler* le résultat de l'appel à `SharedMethod` dans `MyClass`)



```
public class MyClass {
    private IDependency dependency;

    public MyClass(IDependency dependency) {
        this.dependency = dependency;
    }

    public void method() {
        //...
        this.dependency.sharedMethod();
    }
}
```

→ N'importe quelle classe peut utiliser le code de `sharedMethod` en référençant la dépendance adéquate

→ `MyClass` contient des informations sur le contexte d'exécution (Exemple : `IDependency` → `IMailService` et `sharedMethod` → `sendEmail`)

→ Test de `MyClass` plus simple car possible de mocker `IDependency` et `sharedMethod`.

D'après : <https://dev.to/ruidfigueiredo/why-composition-is-superior-to-inheritance-as-a-way-of-sharing-code>