ECE PARIS
ÉCOLE D'INGÉNIEURS

Robin FERCOQ

Conception et Programmation Orientée Objet C++

POO - C++

Sommaire général du semestre

COURS	Semaine suivante →	**TPs**
1. *Intro, concepts, 1 exemple*		1. *Organisation objet des données*
2. *Modélisation objet / UML*		2. *Diagrammes de classe UML*
3. *C++ pratique 1*		3. *C++ pratique, E/S, string, vector*
4. *C++ pratique 2*		4. *C++ pratique, type &, surcharge*
5. *Classes & C++ : bases*		5. *Date : une classe simple en C++*
6. *Classes & C++ : compléments*		6. *UML et C++, associations*
7. *Conteneurs & C++ : la STL*		7. *Gestion de collections complexes*
8. *Héritage / polymorphisme*		8. *Collections polymorphes*
9. *Modèles objets avancés*		9. *Modèle composite et graphismes*
10. *Exceptions, flots, fichiers ...*		10. *Persistance / fichiers / except.*
11. *Templates côté développeur*		11. *Développement de templates*
12. *Gestion mémoire / smarts ptrs*		12. *Soutenance de **projet** ...*

COURS 2

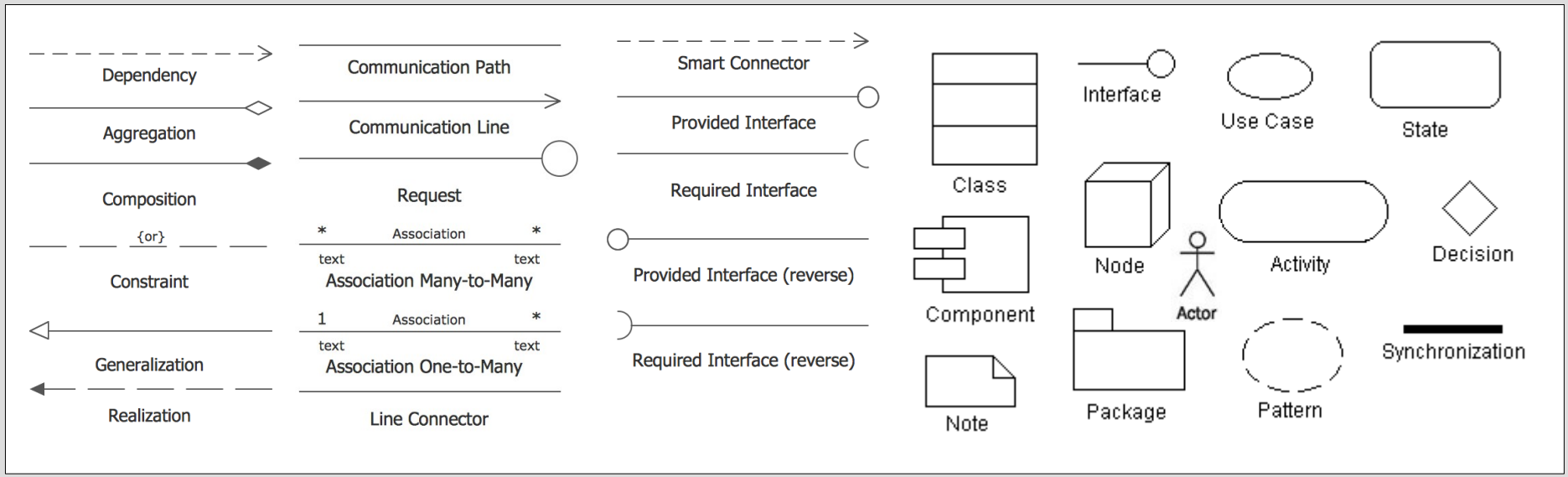
A) UML et méthodes dev. objet
B) Diagramme de classes en UML
C) La classe en UML !
D) Les associations entre classes
E) L’héritage et le polymorphisme
F) Compléments

COURS 2

A) UML et méthodes dev. objet
B) Diagramme de classes en UML
C) La classe en UML !
D) Les associations entre classes
E) L'héritage et le polymorphisme
F) Compléments

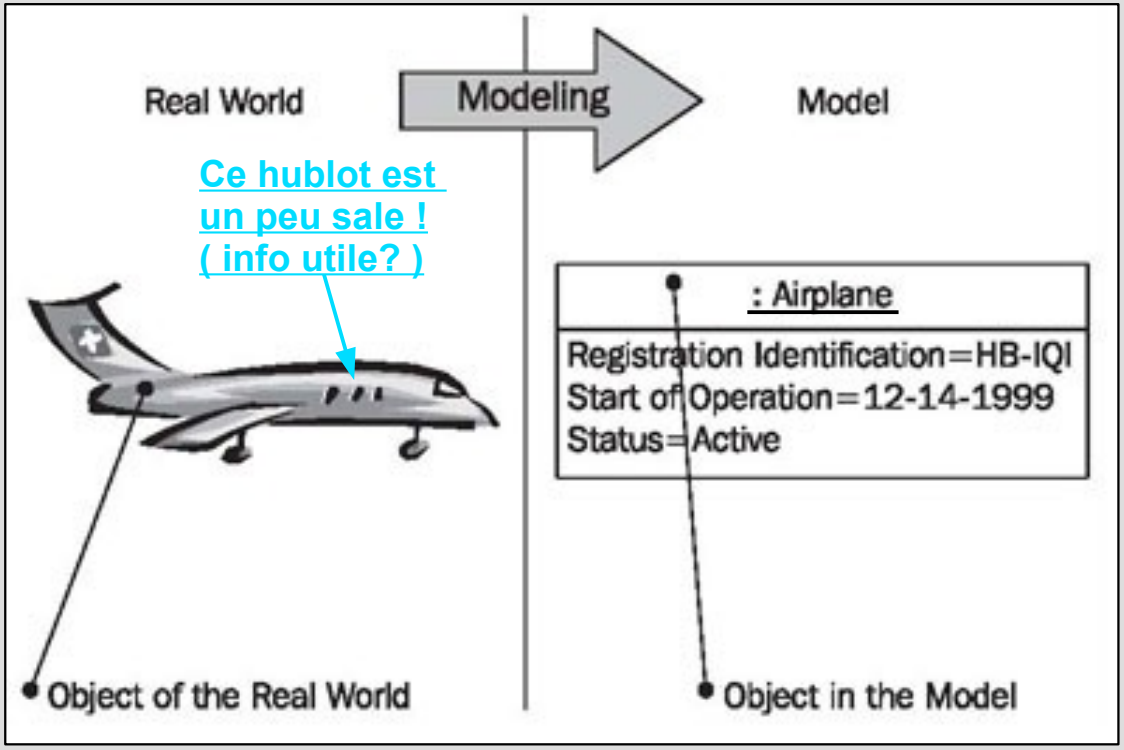
UML et méthodes dev. objet

- *UML = Unified Modeling Language*
- *UML est un « langage » de représentation graphique standardisé de modèles informatiques orientés objet*
- *UML permet de décrire et visualiser la structure et le comportement d'un système logiciel orienté objet en cours de conception ou d'évolution ou déjà conçu*



UML et méthodes dev. objet

- *Un modèle est une « projection » des entités réelles dans l'espace des informations **utiles** à l'application logicielle ciblée : à quoi vont **servir** les informations du modèle ? On va donc simplifier/choisir des données.*



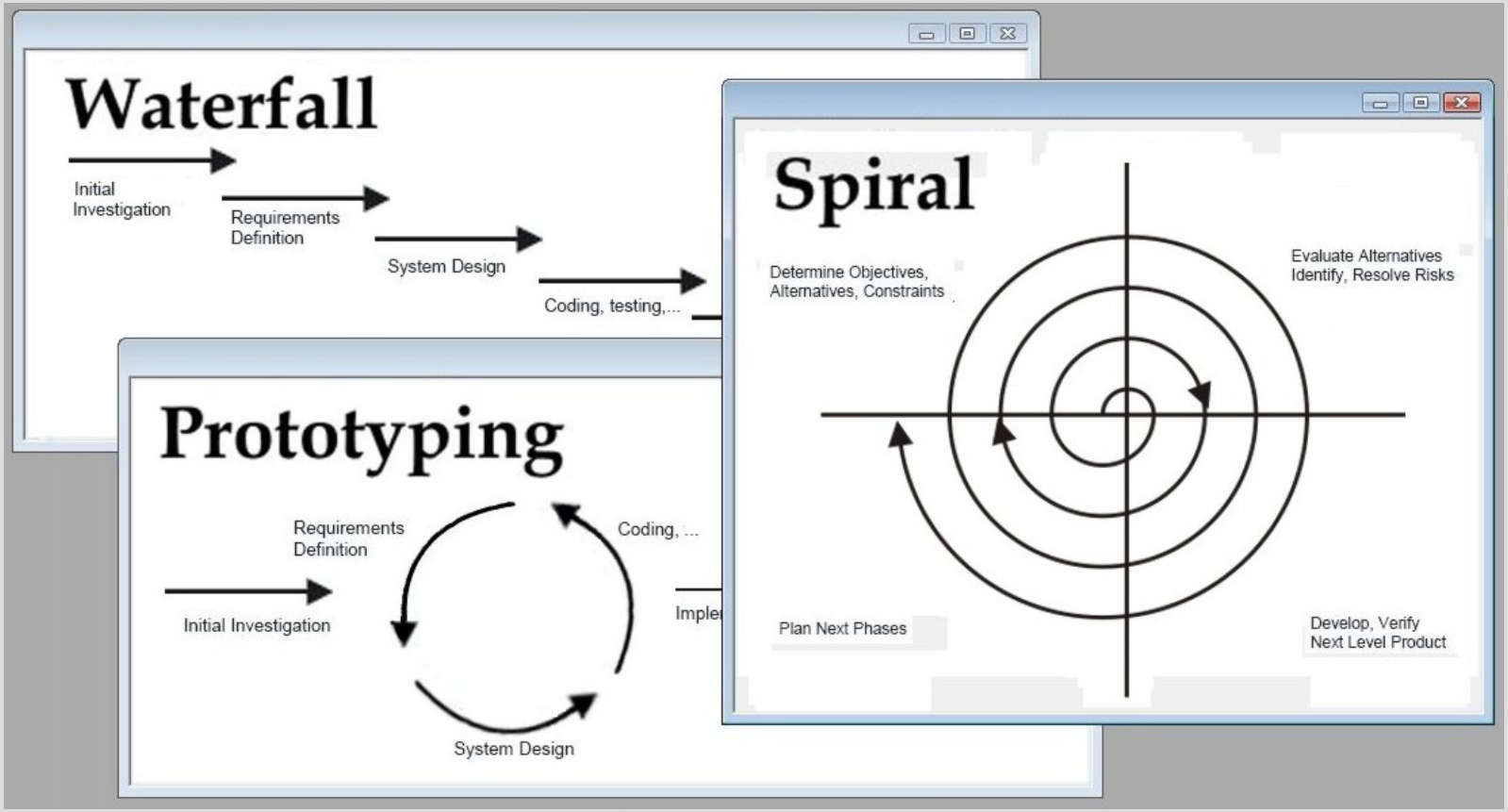
CDC 1
Optimiser le renouvellement de la flotte

CDC 2
Suivre et améliorer le service clients 1ère classe / VIP

On modélise par rapport au périmètre et aux objectifs d'un CDC

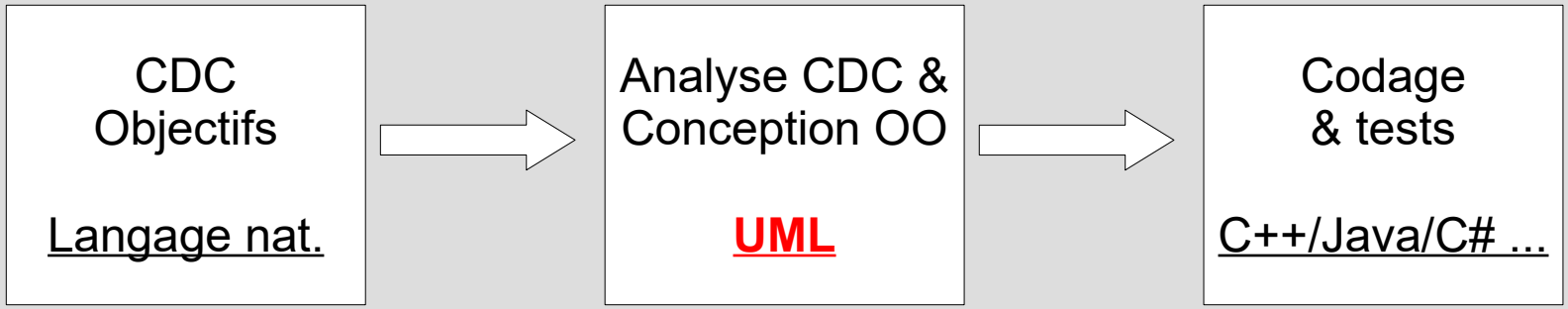
UML et méthodes dev. objet

- *UML n'est **pas** une méthodologie de développement*
- *Il y a plusieurs méthodologies, selon le type de projet*



UML et méthodes dev. objet

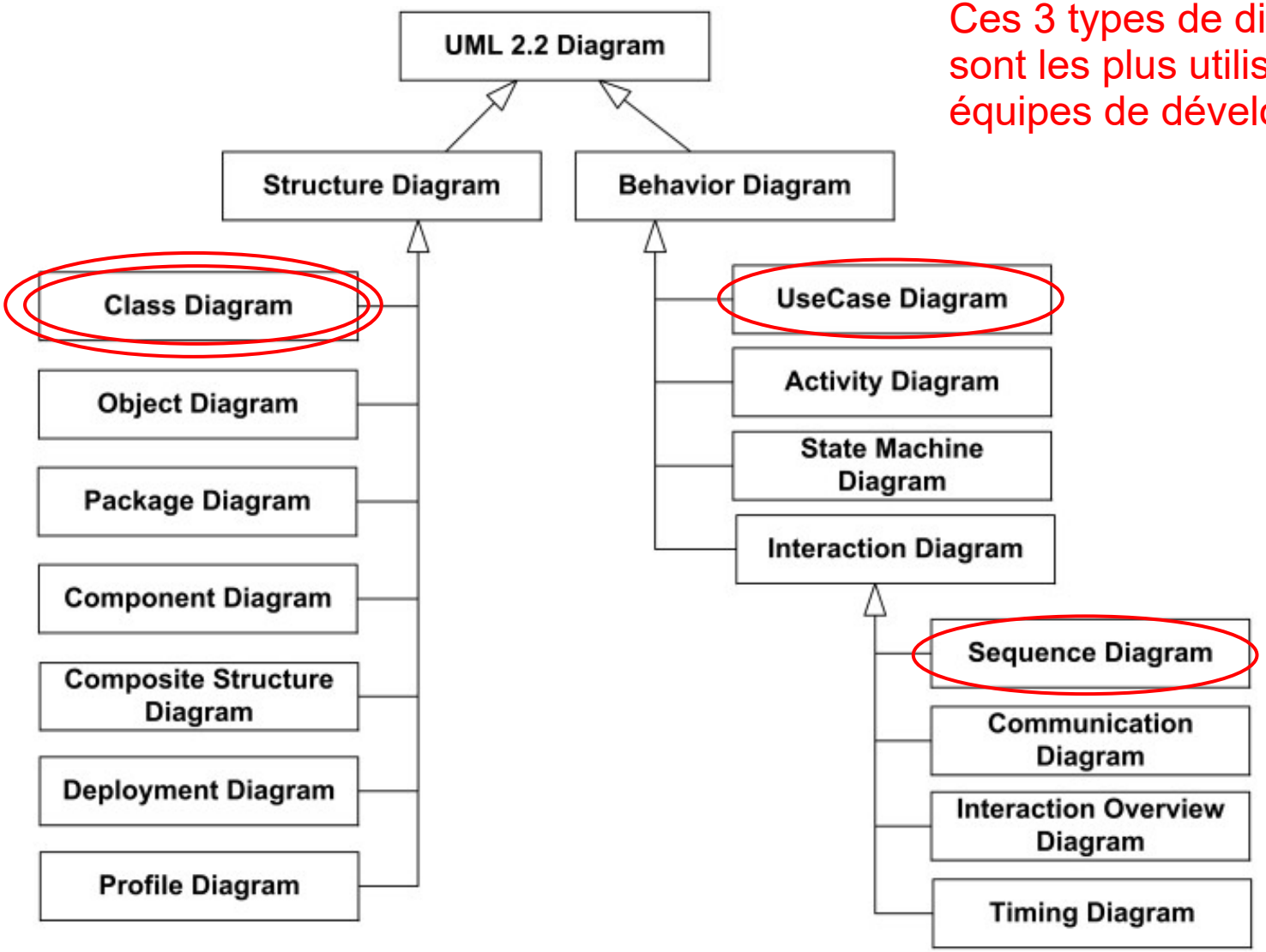
- *UML n'est **pas** une méthodologie de développement*
- *Il y a plusieurs méthodologies, selon le type de projet*
- *Ce sont différentes façons de (re)parcourir la séquence*



- *UML est un langage commun et un pivot dans la plupart des méthodologies quand on développe objet*

UML et méthodes dev. objet

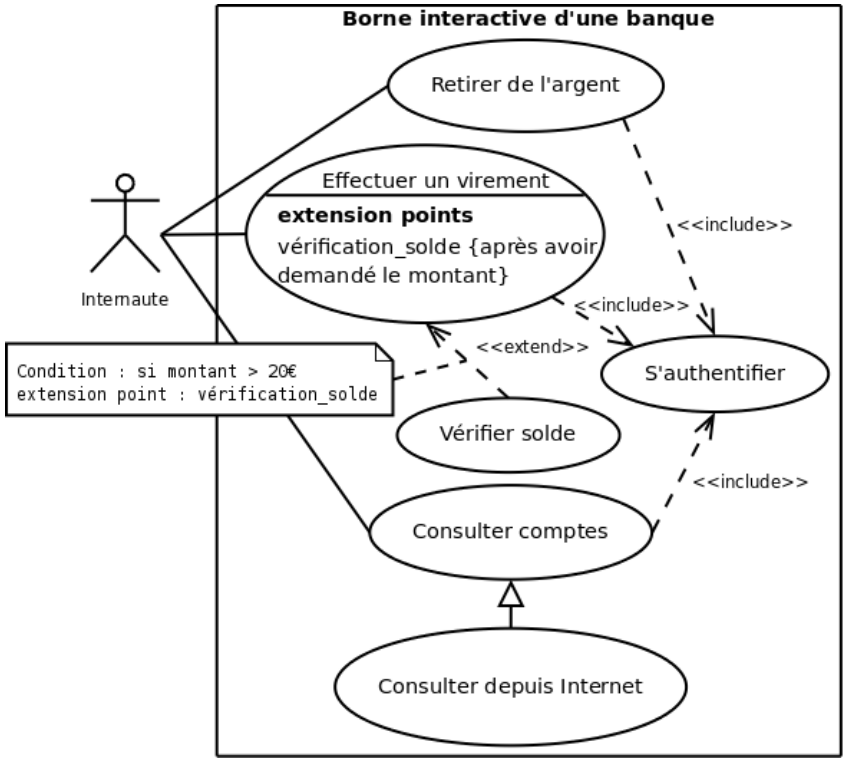
- *La norme UML est riche et complexe*



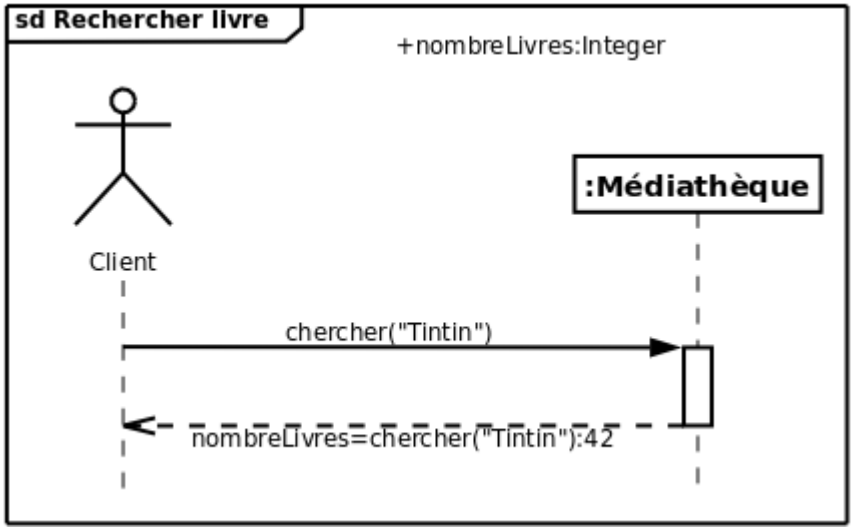
Ces 3 types de diagrammes sont les plus utilisés par les équipes de développement

UML et méthodes dev. objet

- *Les diagrammes de comportement (behavior) décrivent des aspects dynamiques du modèle*



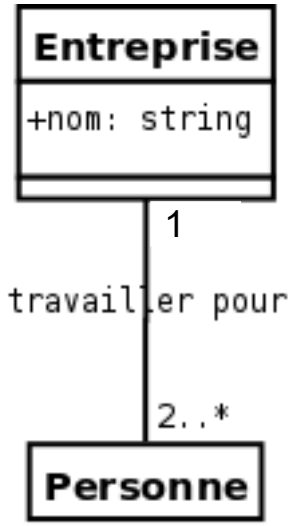
Cas d'usage



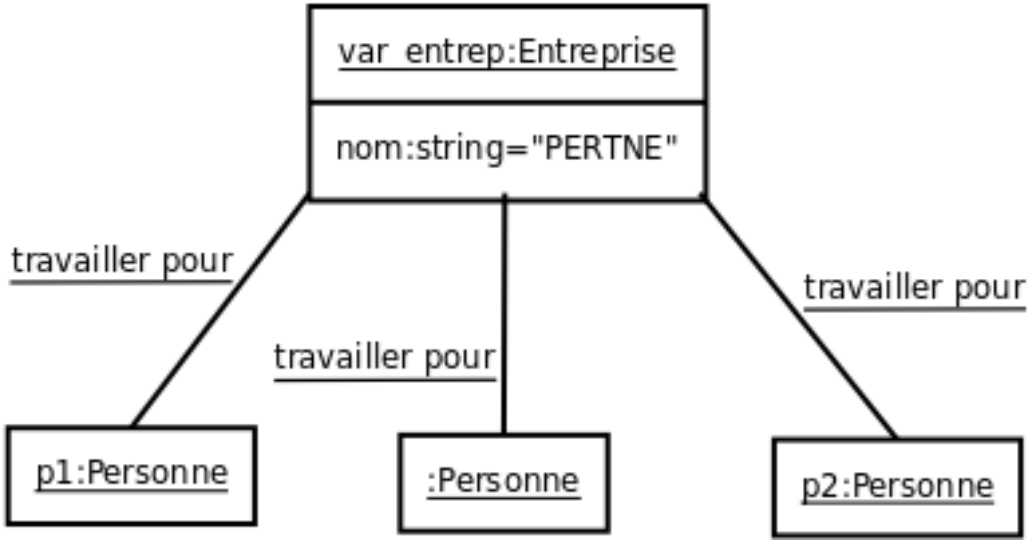
Sequence

UML et méthodes dev. objet

- *Les diagrammes de structure décrivent des aspects statiques du modèle*



Diagrammes de classes



Diagrammes d'objets

COURS 2

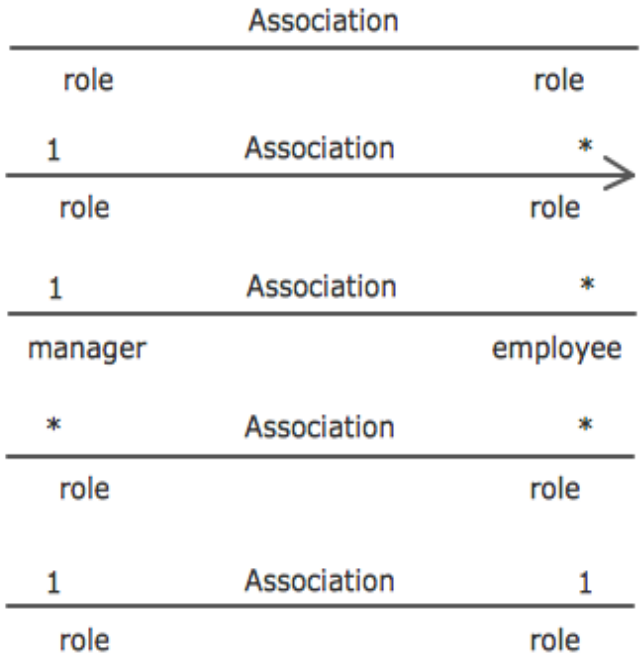
A) UML et méthodes dev. objet
B) Diagramme de classes en UML
C) La classe en UML !
D) Les associations entre classes
E) L'héritage et le polymorphisme
F) Compléments

Diagramme de classes en UML

- *On ne couvrira ici que les **diagrammes de classes** qui sont les diagrammes les plus importants pour coder un projet et assurer sa cohérence, sa lisibilité*
- *Ils représentent les plans : l'architecture du logiciel*
- *Une classe =*
  - ***Une seule boite dans le diagramme de classes***
  - *Une déclaration de classe en C++ ...*
    ```
    class MaClasse {  attributs …  méthodes … };
    ```
  - *Un fichier en-tête maClasse.h*
  - *Un fichier d'implémentation maClasse.cpp*
    ```
    void MaClasse::uneMethode(int param) { ... }
    float MaClasse::autreMethode()       { ... }
    ```

Diagramme de classes en UML

- *Respecter les usages et notations de la norme UML*
- *Universellement (re)connu par les développeurs*
- *Principaux éléments du **diagramme de classes** :*



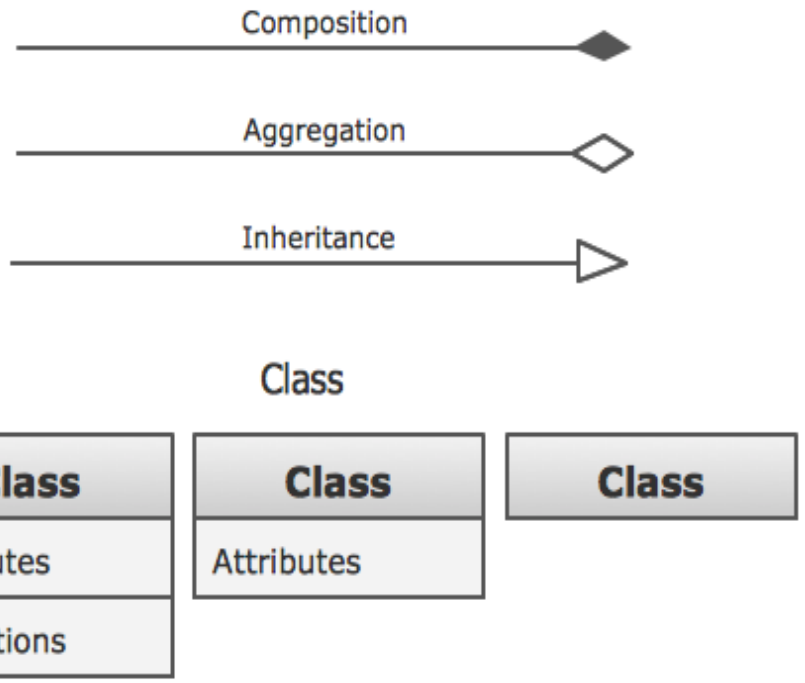


Diagramme de classes en UML

- *Les **diagrammes d'objets** sont illustratifs, particuliers*
- *Les **diagrammes de classe** sont abstraits, généraux*

Objets...
représentants concrets des classes

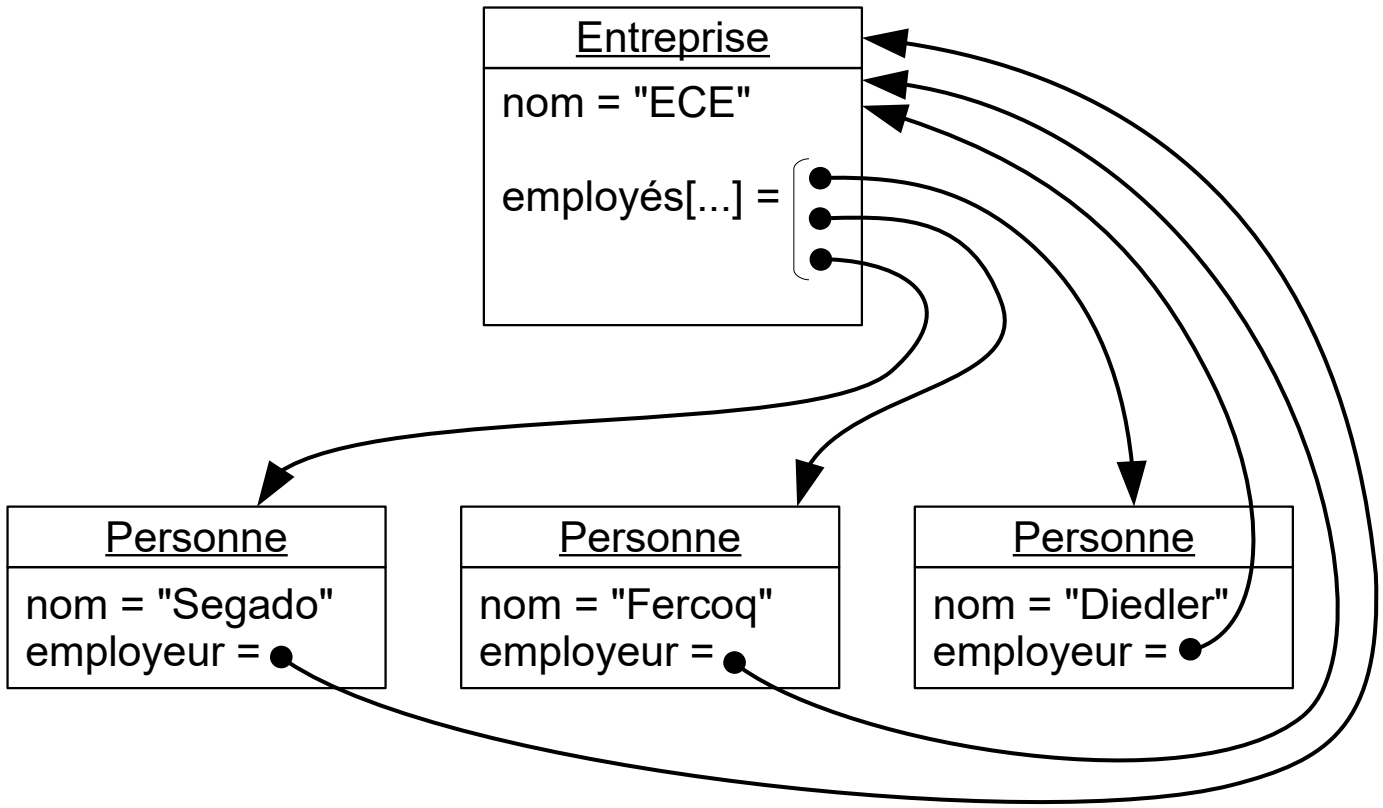


Diagramme d'objets

Ici on s'écarte de la norme UML : on reprend les « schémas mémoire » vus en ING1

Classes...
catégories d'objets
de même nature

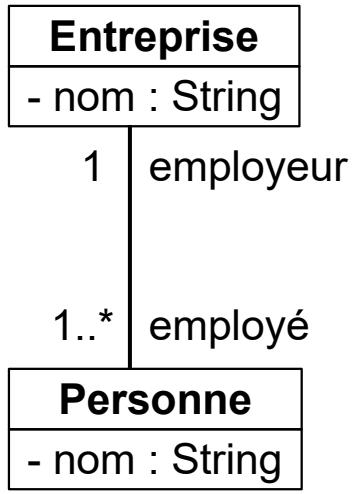


Diagramme de classes

Norme UML stricte

Diagramme de classes en UML

- *Un diagramme d'objets est un cas particulier, c'est un diagramme « jetable » qui sert à illustrer ou comprendre **une certaine situation** représentative d'objets manipulés par le système*
- *Pour couvrir différentes situations il faut différents diagrammes d'objets, on n'est pas sûr de tout voir*
- *Les diagrammes objets deviennent vite illisibles !*

- *Un diagramme de classes est général : il couvre **toutes les situations possibles** entre les (objets des) classes*
- *Les diagrammes de classes sont plus abstraits, ils ne représentent pas directement des objets !*

Diagramme de classes en UML

- *Avec un autre diagramme d'objets on peut découvrir de nouvelles situations qui changent le modèle !*

Objets...
représentants concrets des classes

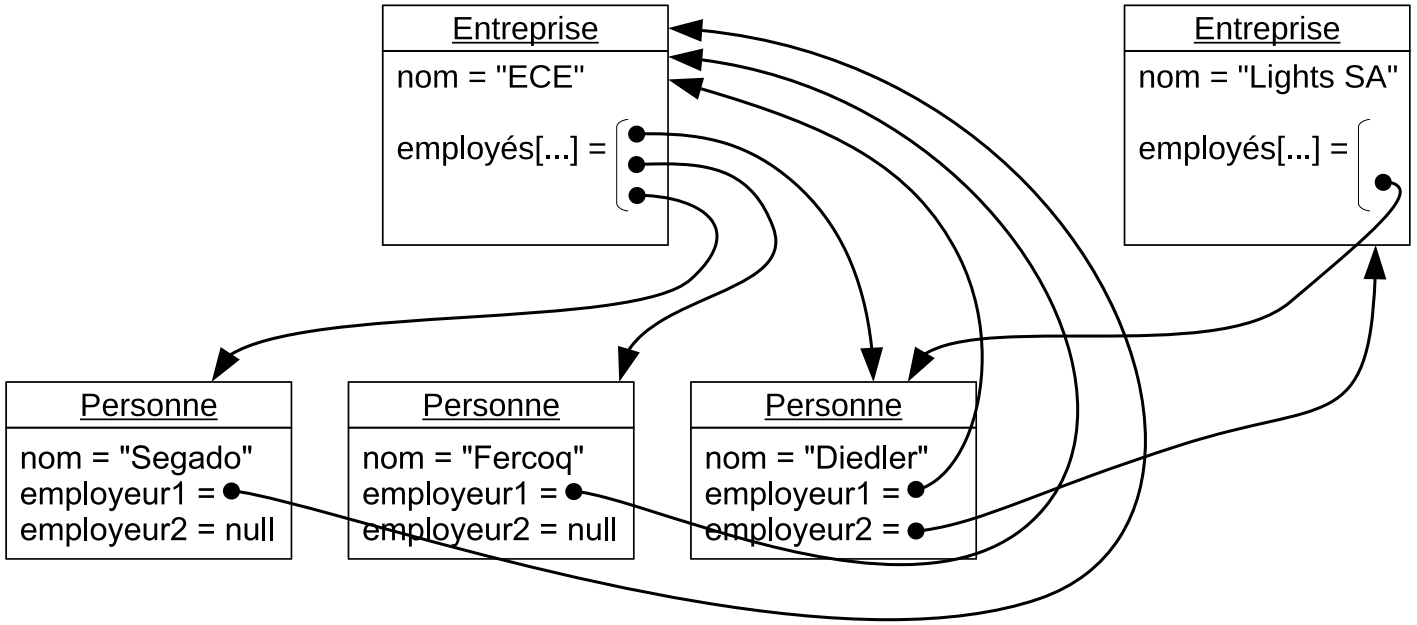


Diagramme d'objets

Classes...
catégories d'objets
de même nature

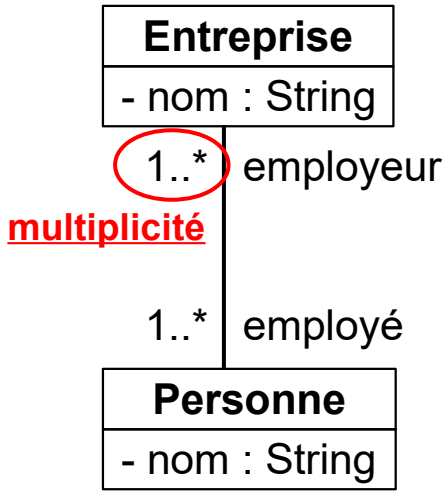


Diagramme de classes

Diagramme de classes en UML

- *Une fois maîtrisée l'abstraction des diagrammes de classes il est possible de zapper la phase concrète des diagrammes d'objets...*

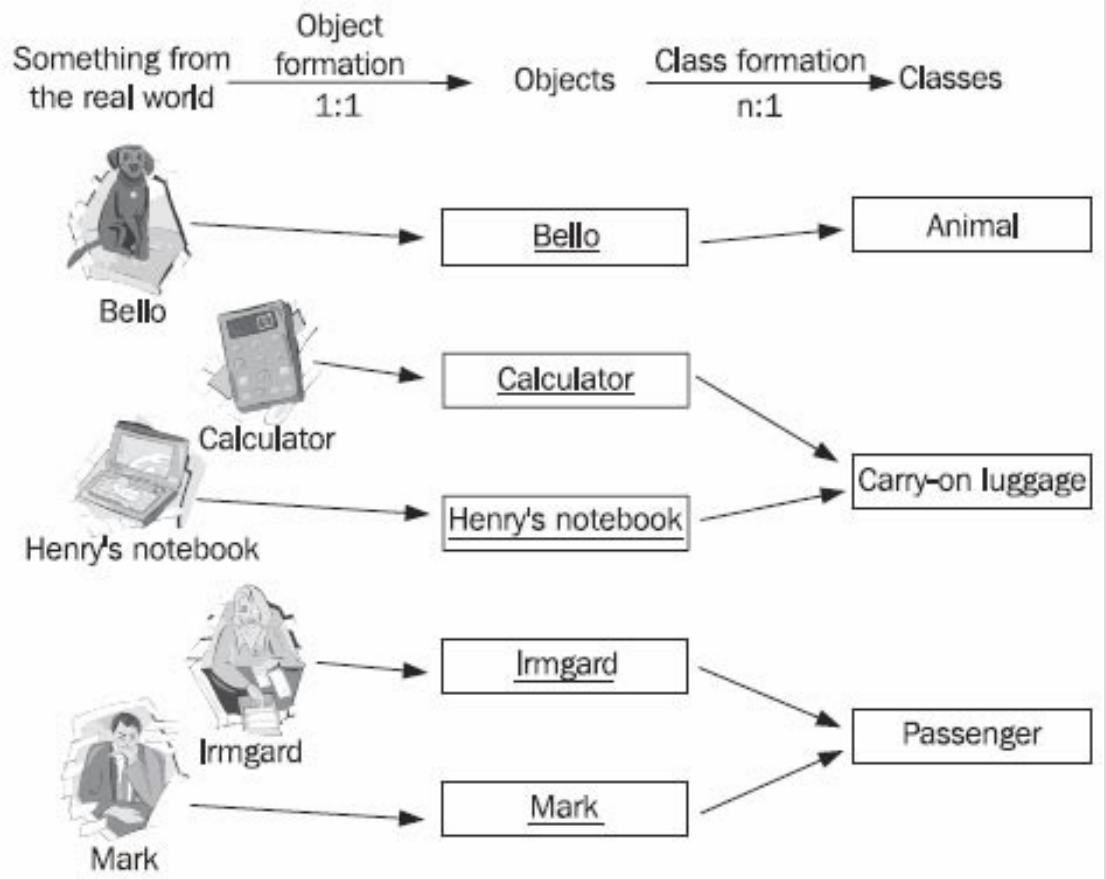
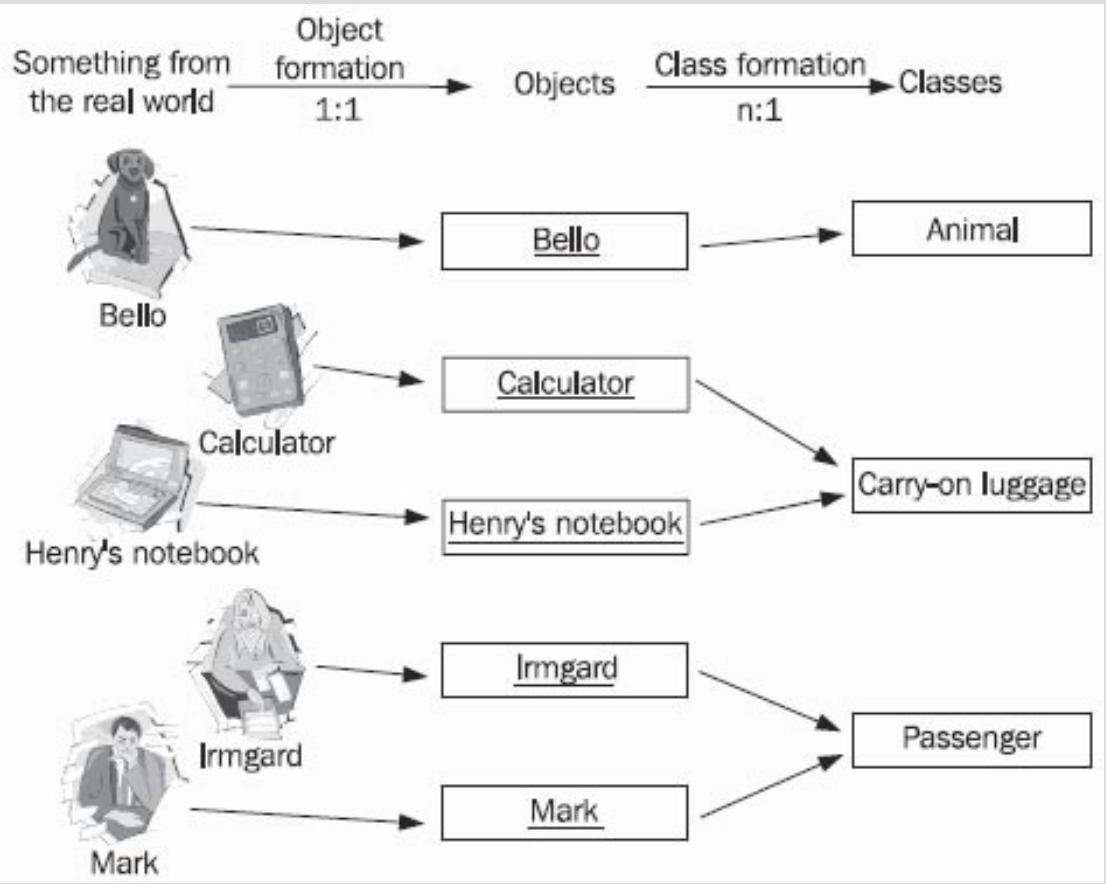


Diagramme de classes en UML

Directement !



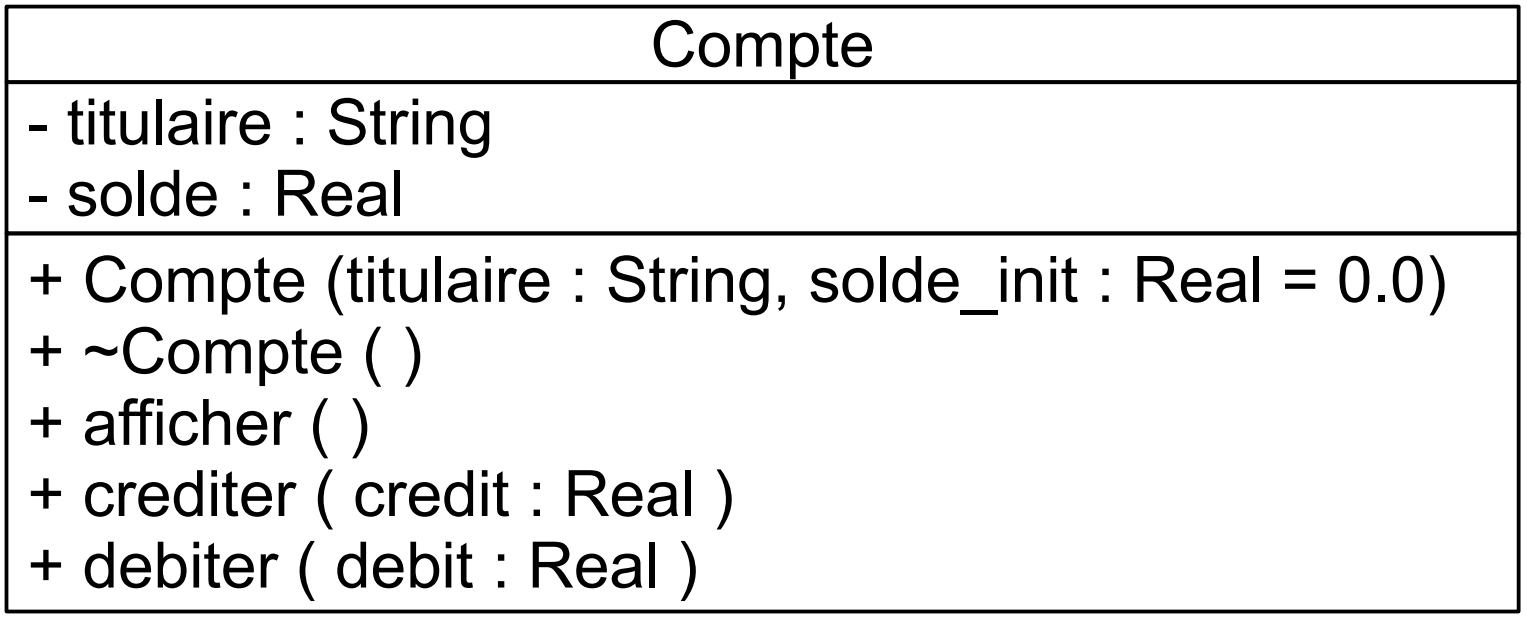
Il reste utile de savoir faire des des diagrammes d'objets « dans sa tête »

COURS 2

A) UML et méthodes dev. objet
B) Diagramme de classes en UML
C) La classe en UML !
D) Les associations entre classes
E) L'héritage et le polymorphisme
F) Compléments

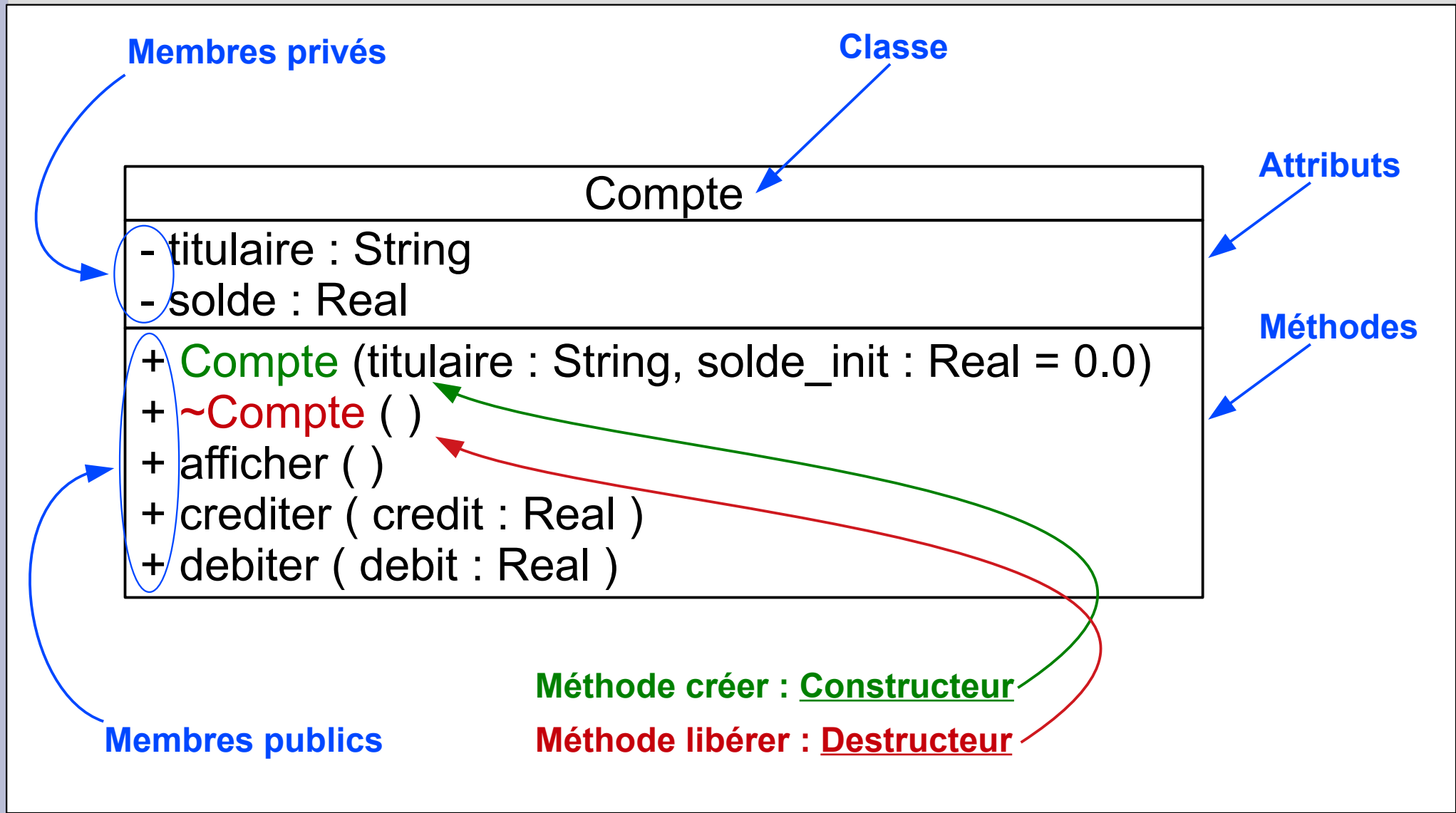
La classe en UML !

- *Une **classe** Compte en notation UML normalisée*



La classe en UML !

- *Une **classe** Compte en notation UML normalisée*



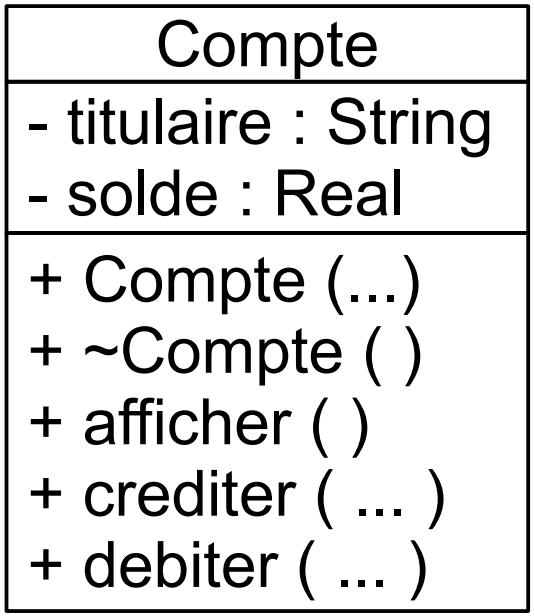
La classe en UML !

- *Différents stades d'élaboration du modèle*

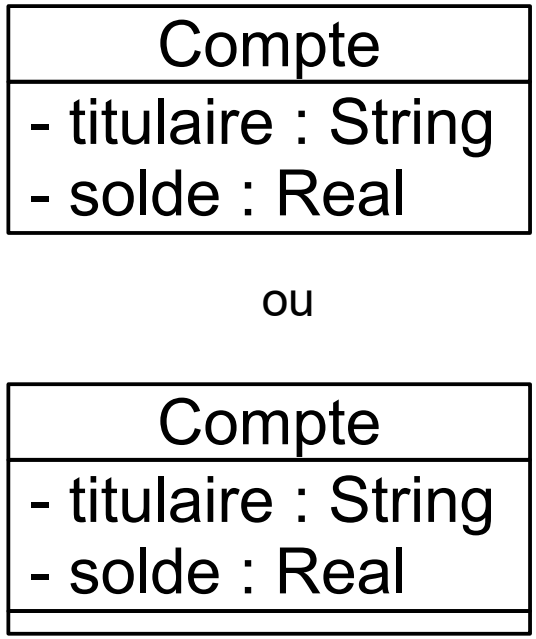
  analyse initiale, faisabilité, cohérence des données, conception détaillée...

- *On peut représenter les classes d'un diagramme complètement ou partiellement selon les besoins*

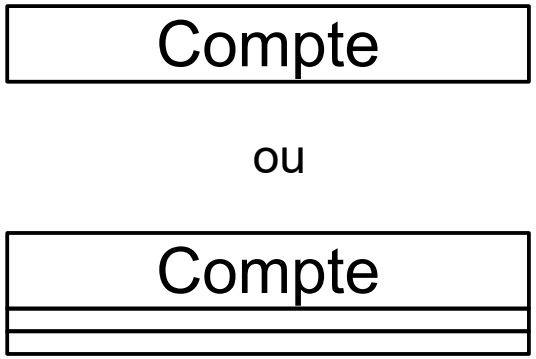
Complet



Sans les méthodes



Seulement le nom !



La classe en UML !

- *Le format général des attributs*

`accès nomAttribut : Type = valeur par défaut`

  - *accès* **-** *privé* **+** *publique* *# protégé (cf chap. E)*
    le principe d'encapsulation ***exclut*** *en général l'accès publique*
  - *nomAttribut commence par une minuscule*
  - *Type voir slides suivants*
  - *valeur par défaut facultatif (fréquemment omis)*

La classe en UML !

- *Le format général des méthodes*

`accès nomMéthode( param1, param2… ) : TypeRetour`

  - *accès* **-** *privé* **+** *publique* *# protégé (cf chap. E)*
    les méthodes qui constituent l'interface (le mode d'emploi)
    des objets de la classe sont évidemment en accès publique
  - *nomMéthode commence par une minuscule*
  - *TypeRetour idem Type, voir slides suivants*
    Si la méthode ne retourne rien on ne met pas : TypeRetour

La classe en UML !

- *Le format général de paramètre de méthode*

`direction nomParametre : Type = valeur par défaut`

  - *direction in / out / inout*
    facultatif mais utile pour savoir si les données de l'appelant doivent être initialisées (in ou inout) et si après l'appel l'appelant voit les données modifiées (out ou inout)
    Exemple : méthode de classe AutomateBancaire
    `+ codeValide(out nbEssais : Integer) : Boolean`
  - *nomParametre commence par une minuscule*
  - *Type voir slides suivants*
  - *valeur par défaut facultatif*

La classe en UML !

- *Les types de base*
  - *Integer (→ int unsigned int …)*
  - *Real (→ float double …)*
  - *Boolean (→ bool valeurs true / false)*
  - *String (chaînes de caractère (→ std::string)*
- *Les types tableaux ou listes : notation* [cardinalité]
  - *Integer[3] 3 entiers*
  - *Real[1..5] entre 1 et 5 flottants*
  - *Real[1..*] au moins 1 réel, peut-être 1000000*
  - *String[*] 0 ou nb. quelconque de chaînes*

La classe en UML !

- *Les types de classes de bas niveau (bibliothèques)*
  - *Date*
  - *AdressePostale*
  - *CoordsGPS*
  - *ComplexNumber*
  - *… Toute classe « utilitaire » du domaine bien connue par l'équipe (classe déjà en place)*

La classe en UML !

- *Les types des autres classes qu'on développe...*
  - *En paramètre/retours des méthodes OUI*
  - *Comme type d'attribut, en général NON*

!?

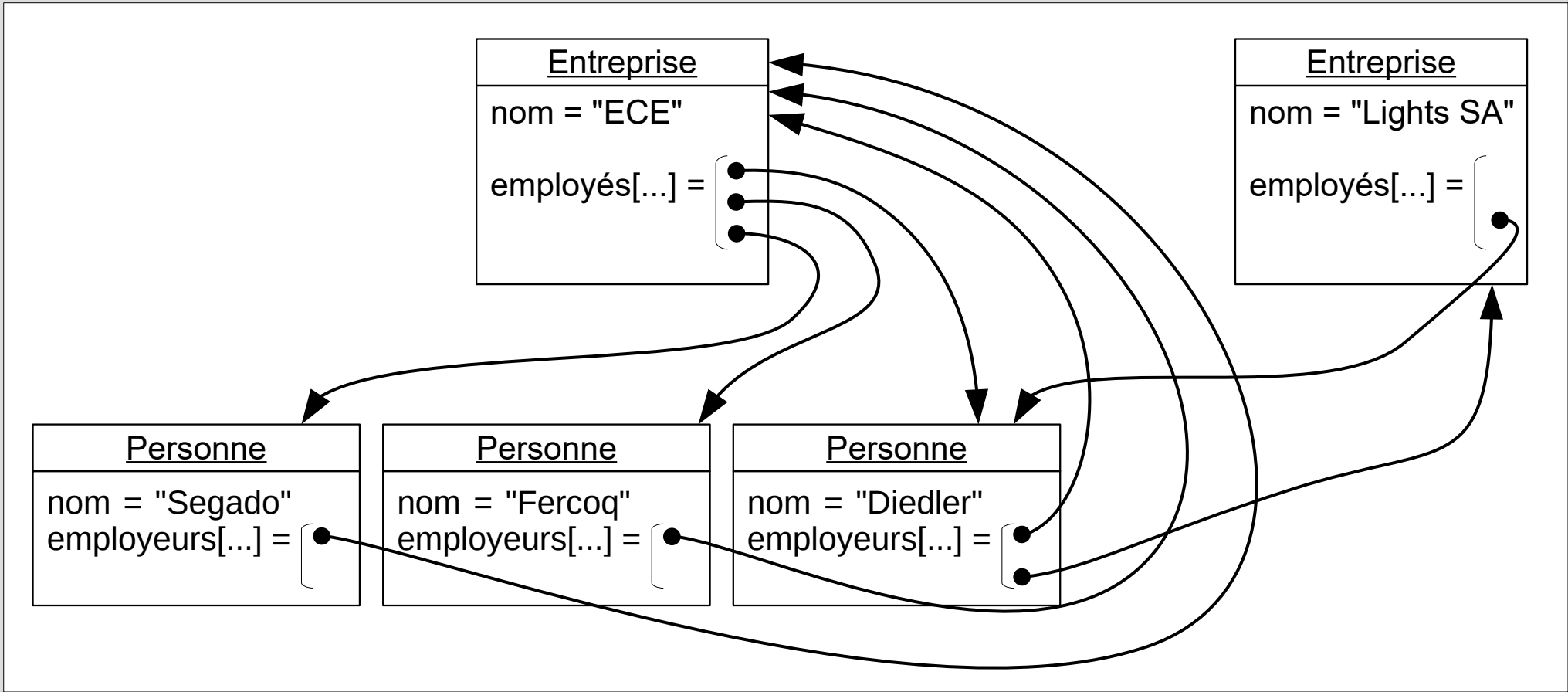
- *En effet les attributs (données membres) d'une classe qu'on développe qui sont des données d'autres classes en développement seront représentés graphiquement sous forme de liaisons. Ceci mérite un gros chapitre !*

COURS 2

A) UML et méthodes dev. objet
B) Diagramme de classes en UML
C) La classe en UML !
D) Les associations entre classes
E) L'héritage et le polymorphisme
F) Compléments

Les associations entre classes

- *Reprenons la relation Entreprise / Personne*
- *Elargissons le CDC :*
  - une entreprise a 1 ou plusieurs employés
  - une personne a 0, 1 ou plusieurs employeurs



Les associations entre classes

- *Concrètement on va coder ça avec des attributs*
  - tableau de pointeurs sur personnes côté Entreprise
  - tableau de pointeurs sur entreprises côté Personne

Diagramme de classes correct mais non visuel

Entreprise
- nom : String - adresse : String - bilan : Real - presse : String[*] - affaires : String[*] - employés : ptr Personne[1..*]

Personne
- nom : String - employeurs : ptr Entreprise[*]

Les associations entre classes

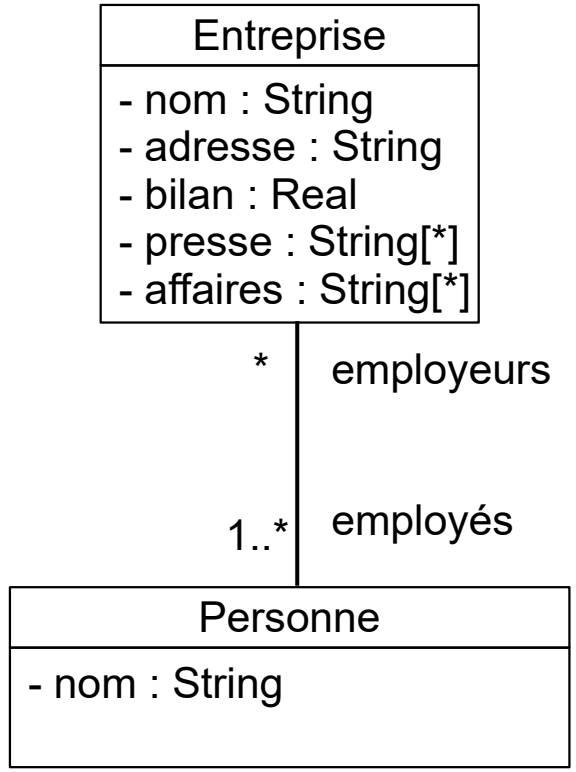
- *Mais ces attributs seront représentés par une liaison, ici une* ***association simple à double sens***

Diagramme non visuel

Entreprise
- nom : String - adresse : String - bilan : Real - presse : String[*] - affaires : String[*] - employés : ptr Personne[1..*]

Personne
- nom : String - employeurs : ptr Entreprise[*]

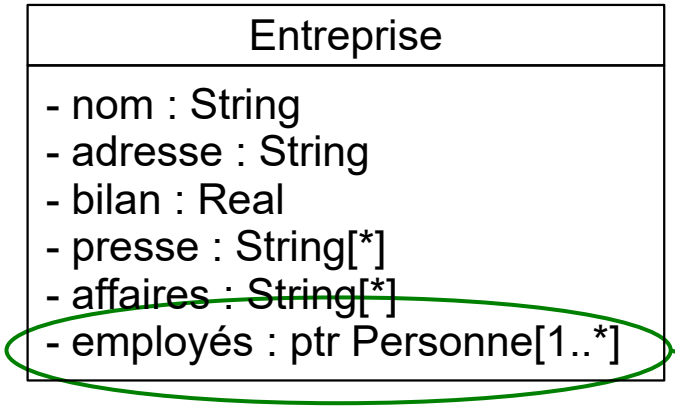
Diagramme visuel normal



Les associations entre classes

- *On retrouve les informations d'un attribut « référence à d'autres objets » du côté des objets référencés*

Diagramme non visuel



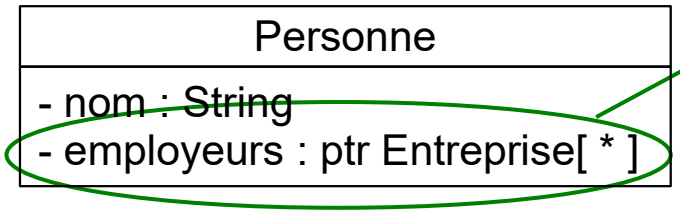
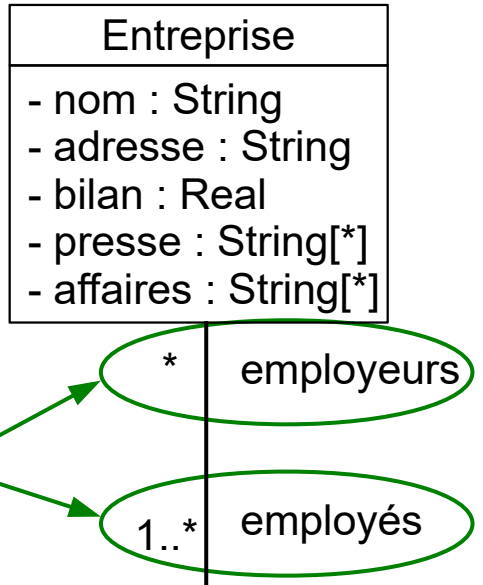
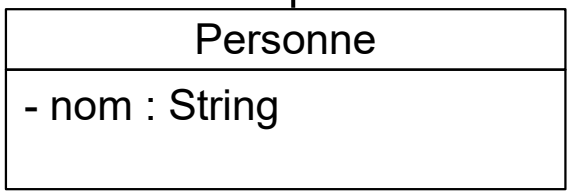


Diagramme visuel normal

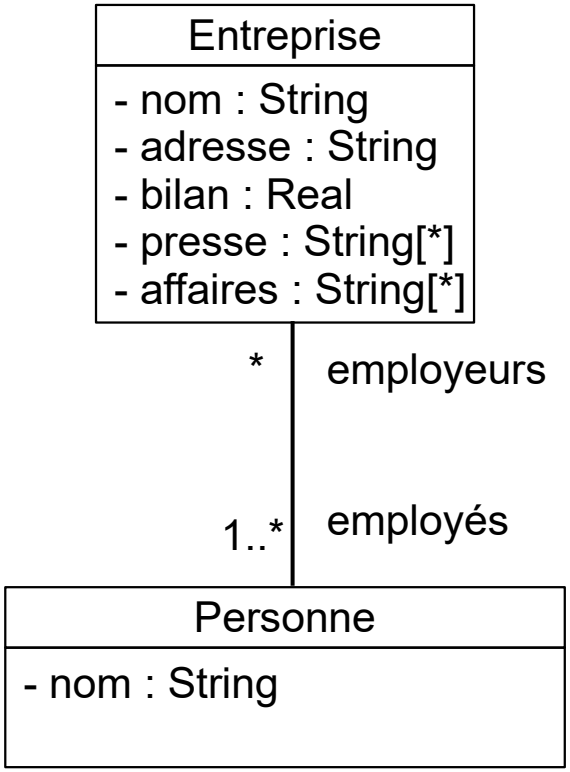




Les associations entre classes

- *Chaque terminaisons d'une association porte*
  - *Une information de **multiplicité** (ou **cardinalité**)*
  - *Une information de **rôle***

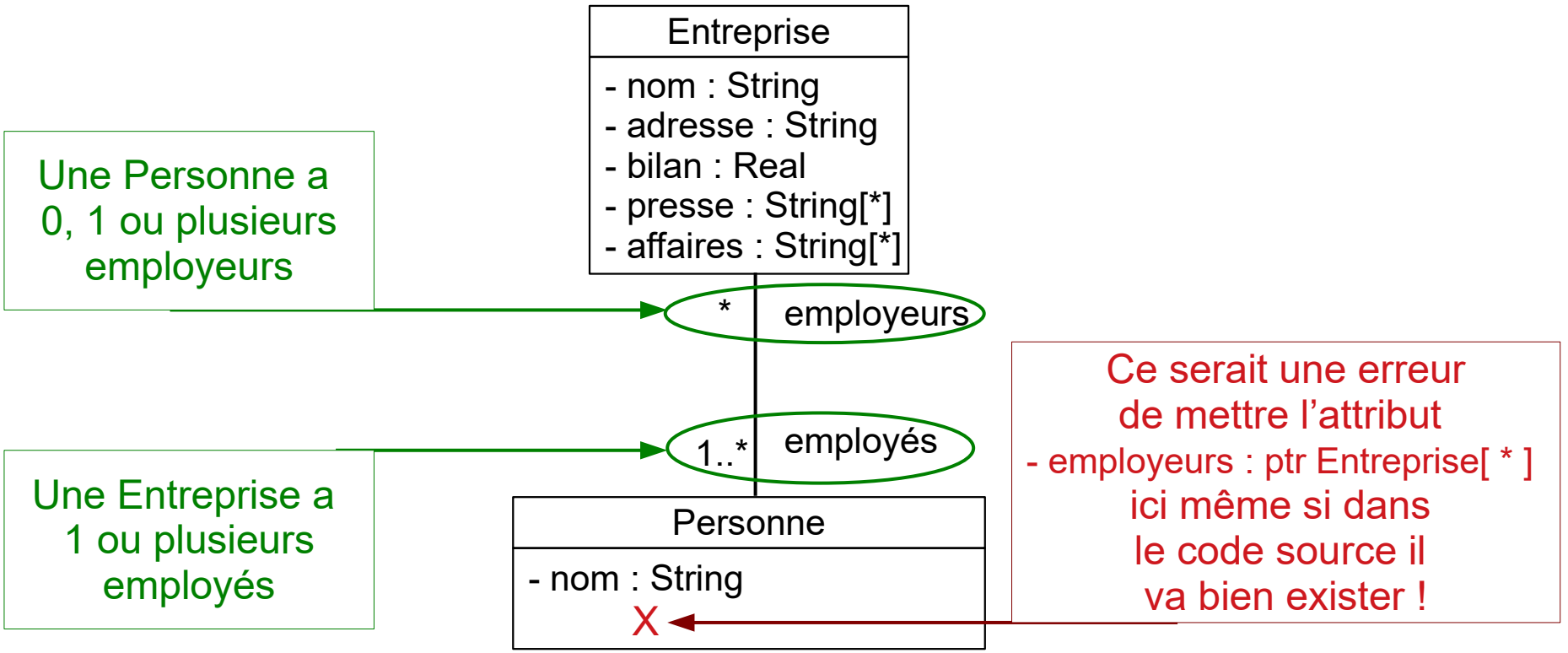
Diagramme de classes normal



Les associations entre classes

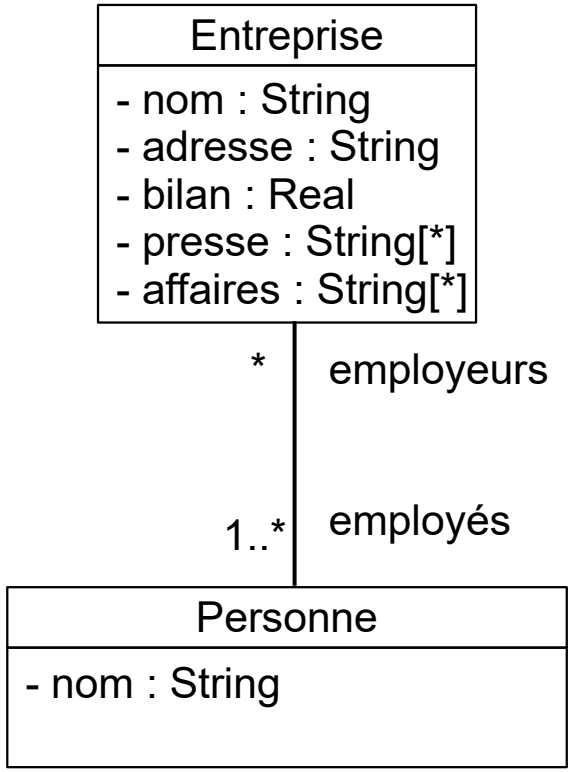
- *Chaque terminaisons d'une association porte*
  - *Une information de **multiplicité** (ou **cardinalité**)*
  - *Une information de **rôle***

Diagramme de classes normal



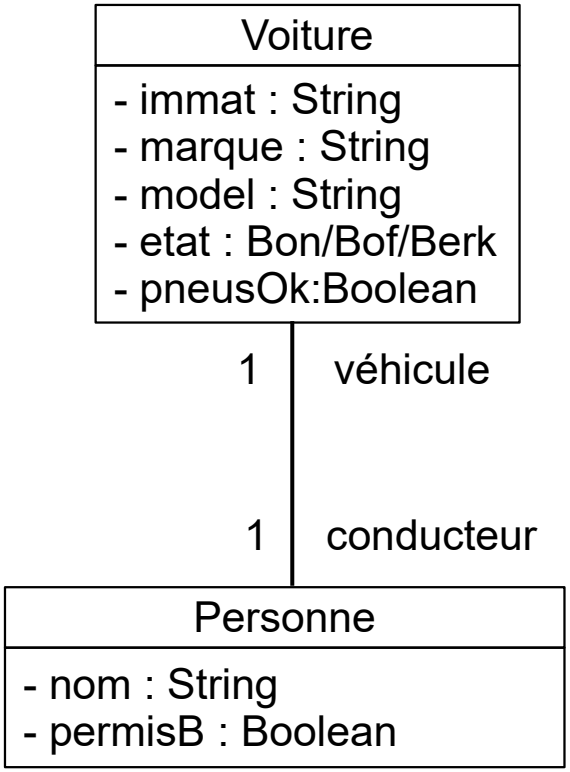
Les associations entre classes

- *Certaines associations sont de type « **plusieurs à plusieurs** »*



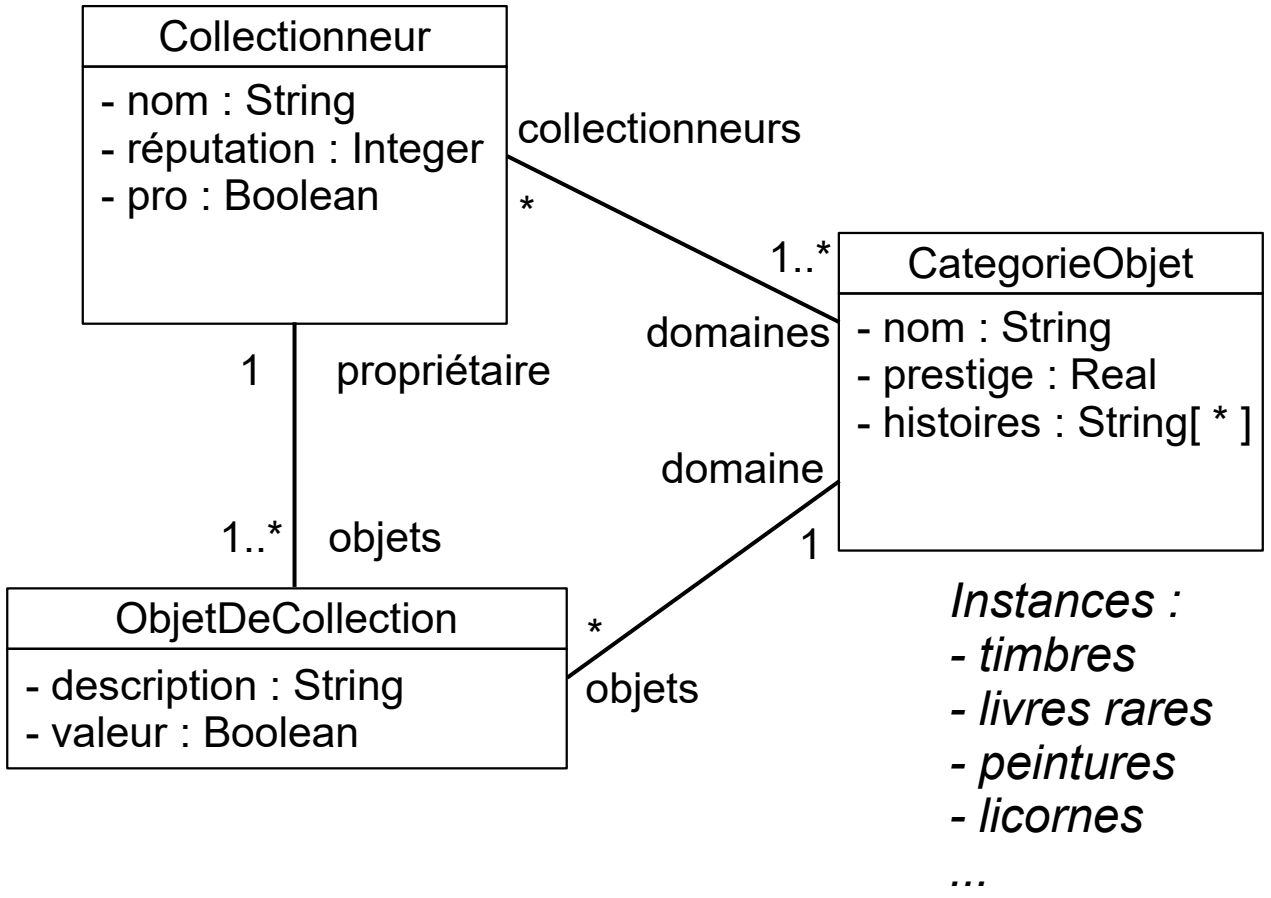
Les associations entre classes

- *Certaines associations sont de type « **un à un** »*



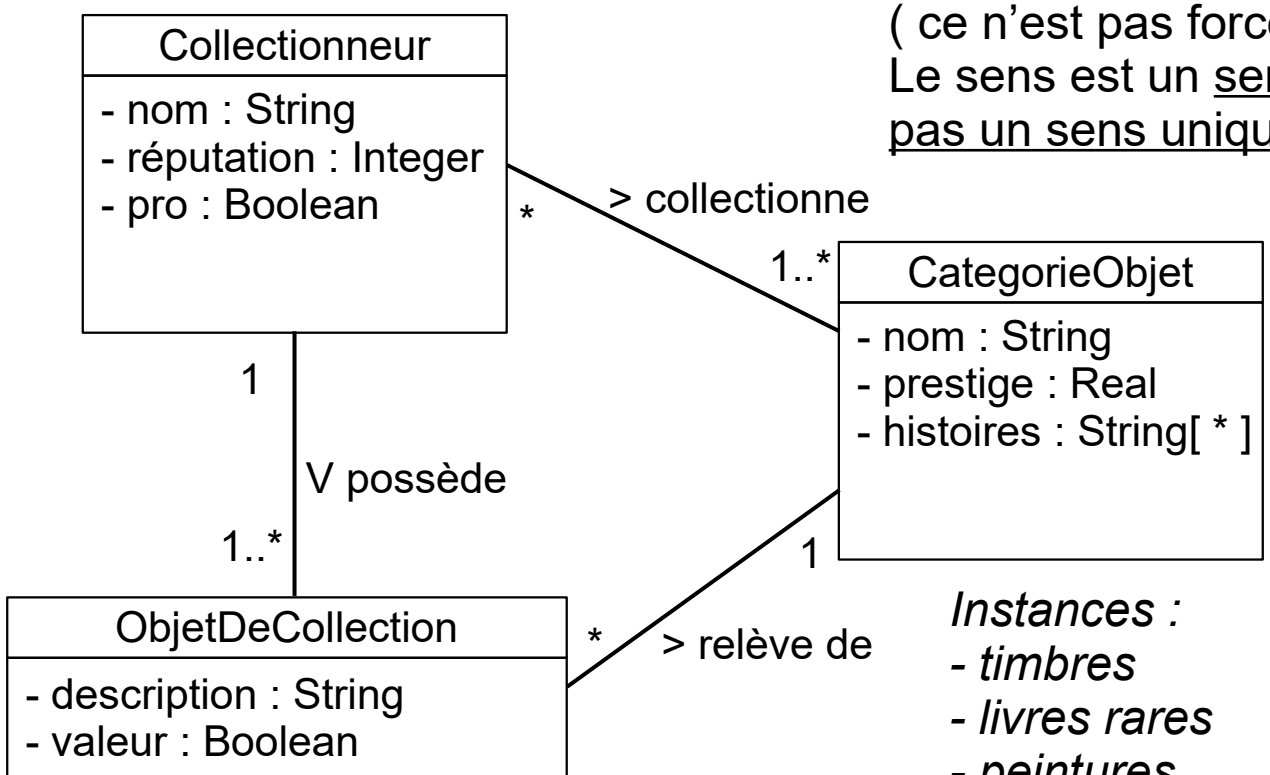
Les associations entre classes

- *Certaines associations sont de type « **un à plusieurs** »*



Les associations entre classes

- *A la place des rôles aux terminaisons on peut indiquer des noms aux associations (en précisant le sens)*



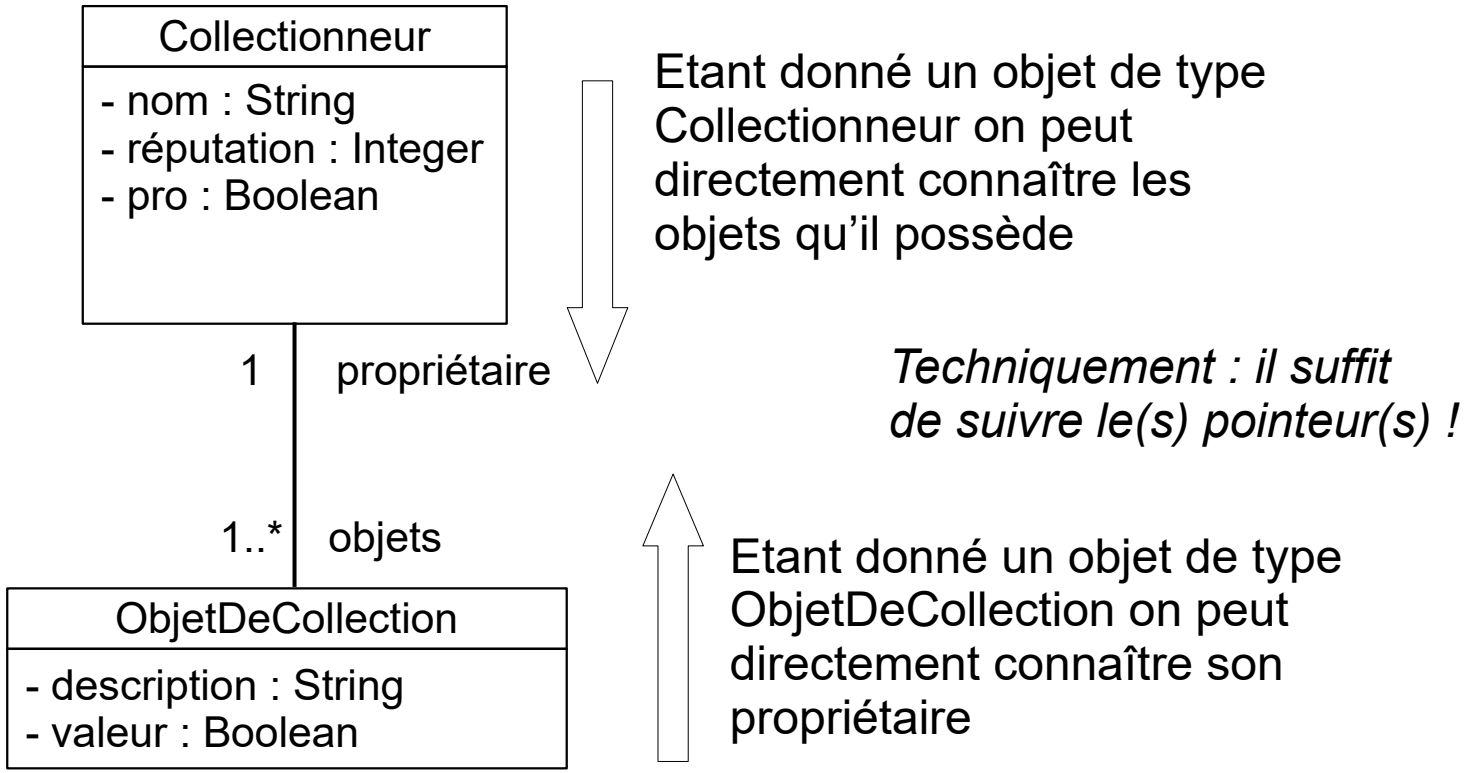
Cette notation est moins proche de l'implémentation (ce n'est pas forcément un défaut) Le sens est un sens d'interprétation pas un sens unique de navigation !

Instances :
- *timbres*
- *livres rares*
- *peintures*
- *licornes*

...

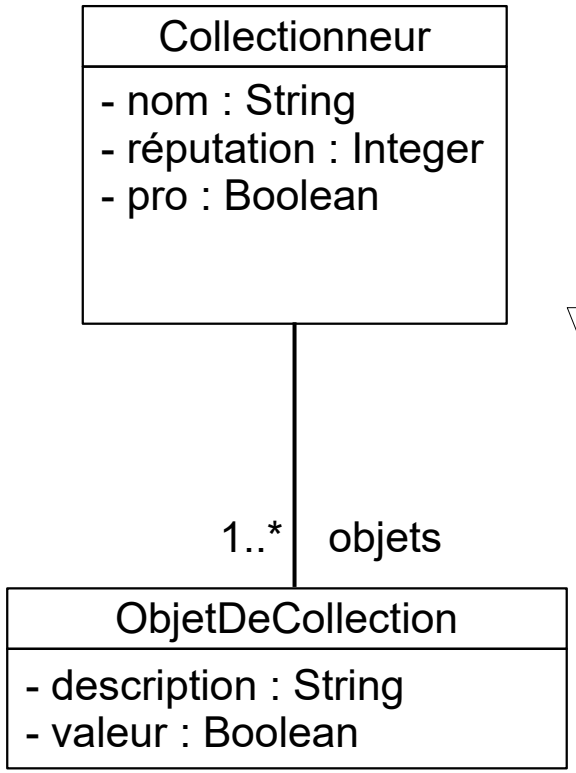
Les associations entre classes

- *Par défaut la navigabilité d'une association est bidirectionnelle (ou « à double sens »)*



Les associations entre classes

- *Si on omet les infos à une terminaison on ne peut pas aller directement aux objets de cette terminaison*



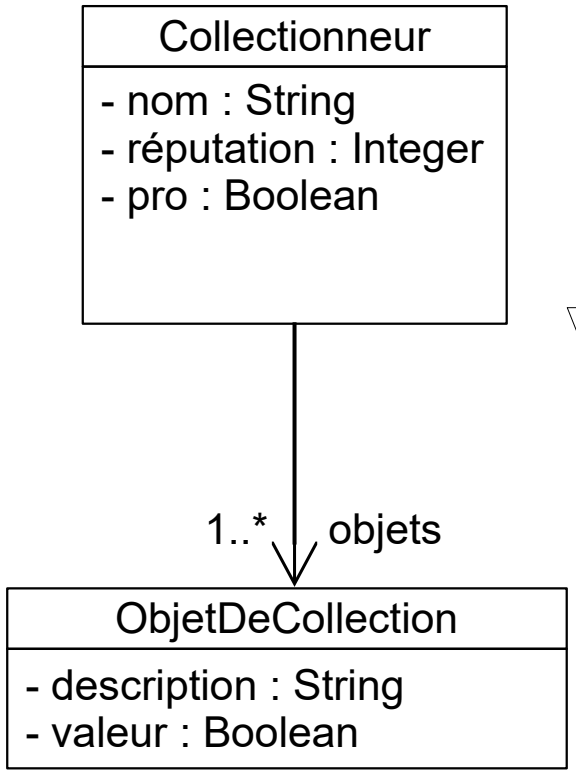
Etant donné un objet de type Collectionneur on peut directement connaître les objets qu'il possède

Techniquement : il n'y a plus de pointeur de ObjetDeCollection vers Collectionneur

Etant donné un objet de type ObjetDeCollection on ne peut pas directement connaître son propriétaire

Les associations entre classes

- *Convention par défaut, pas très clair... Pour indiquer la **navigabilité** mieux vaut préciser **avec une flèche***



Etant donné un objet de type Collectionneur on peut directement connaître les objets qu'il possède

Techniquement : il n'y a plus de pointeur de ObjetDeCollection vers Collectionneur

Etant donné un objet de type ObjetDeCollection on ne peut pas directement connaître son propriétaire

Les associations entre classes

- *Pour les adeptes de l'abstraction dans les phases initiales de conception, la question de la **navigabilité** ne devrait se poser que lors des phases détaillées juste avant l'implémentation (càd. **pas trop tôt**)*

- *D'expérience nous avons pu constater qu'une mauvaise anticipation des problèmes de navigation pouvait plomber un projet, en particulier pour des débutants → **la navigabilité est un aspect important***

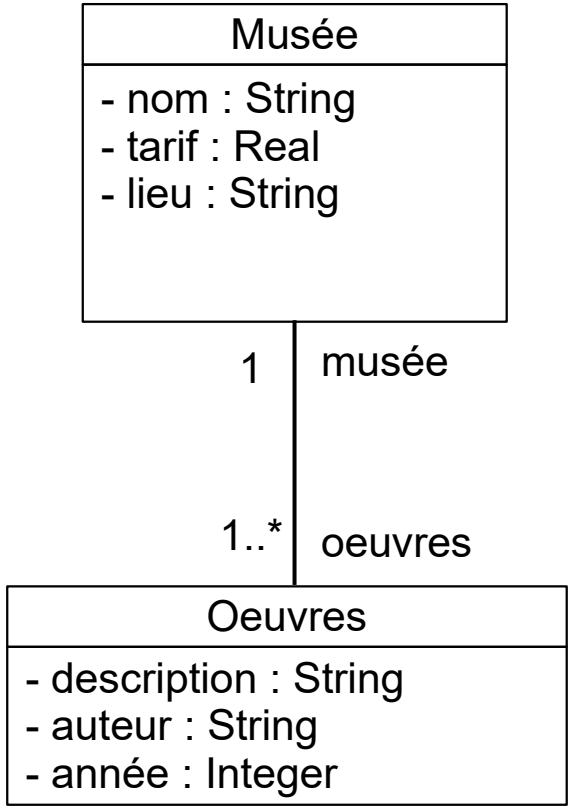
Les associations entre classes

- *Le double sens (par défaut en UML) semble la meilleure solution parce que la plus souple...*
- *Il dispense d'avoir à analyser les sens vraiment nécessaires c'est donc parfois le choix de la paresse (du concepteur)*

- *Mais côté implémentation le choix du double sens complique grandement : il va falloir gérer des liens (pointeurs) réciproques et **garantir leur cohérence** dans des conditions **cycliques**. Faisable, mais dur.*
- *Donc ne laisser un double sens que si c'est vraiment indispensable par rapport aux contraintes*

Les associations entre classes

- *Certaines associations ont une sémantique de type contenant/contenu ou composé/composant et le contenu ou composant est « séparable »*

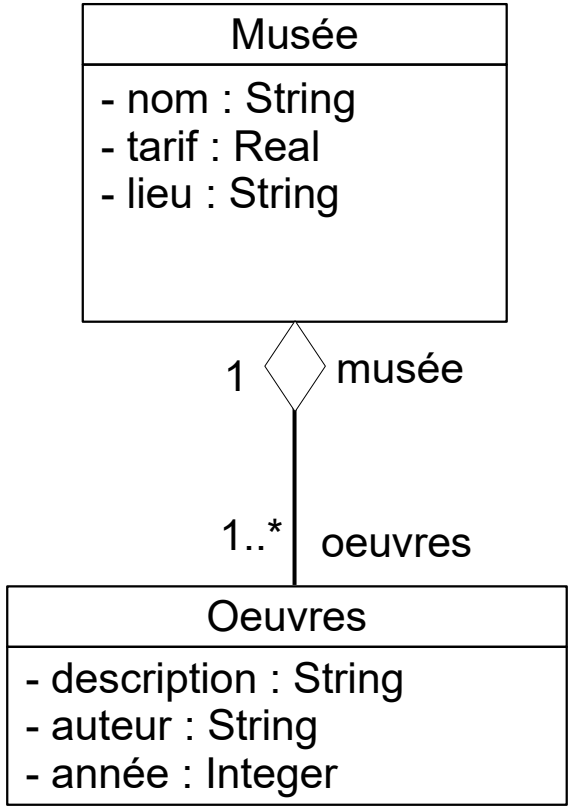


Une œuvre peut quitter un musée pour aller dans un autre (où une collection privée)

Un musée peut fermer définitivement sans que ses œuvres disparaissent

Les associations entre classes

- *Dans ses conditions on peut souligner la sémantique contenant/contenu ou composé/composant en indiquant une **agrégation***



*Lors de l'implémentation du modèle cette indication **ne change rien** (par rapport au schéma précédent)*

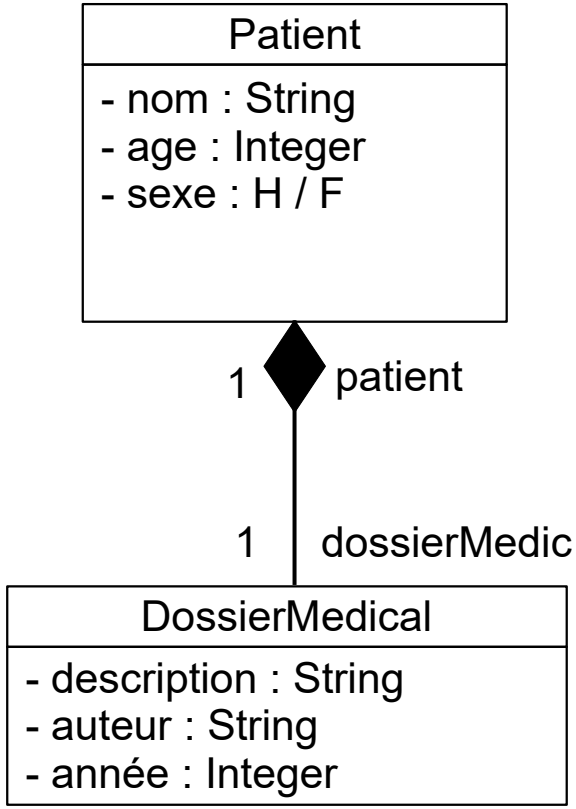
C'est donc une indication qui ne peut faire sens (ou controverse) que par rapport au lecteur humain : elle n'a pas grande importance pratique

Une œuvre peut quitter un musée pour aller dans un autre (où une collection privée)

Un musée peut fermer définitivement sans que ses œuvres disparaissent

Les associations entre classes

- *Certaines associations ont une sémantique de type composé/composant et le composant n'est* ***ni séparable ni partageable*** *en tant que composant*

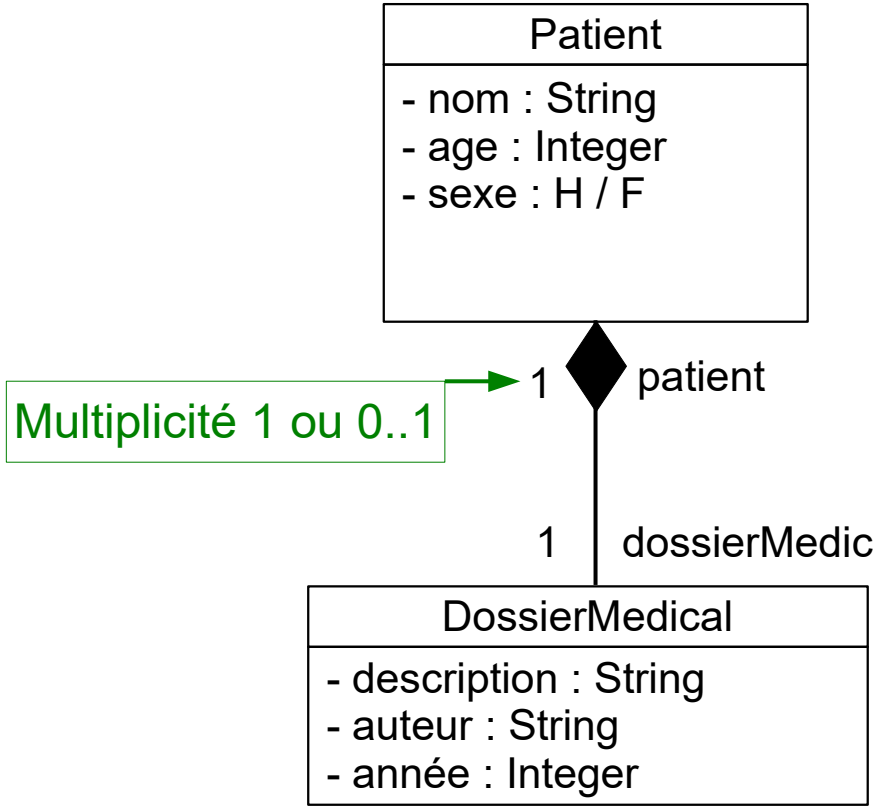


La relation de composition est une relation **a un** :
Un patient **a un** dossier médical

Un dossier médical est intimement lié à la personne qu'il représente, il ne peut pas devenir le dossier médical de quelqu'un d'autre !
Si on détruit (l'objet informatique local) Patient alors on libère (l'objet informatique local) Dossier

Les associations entre classes

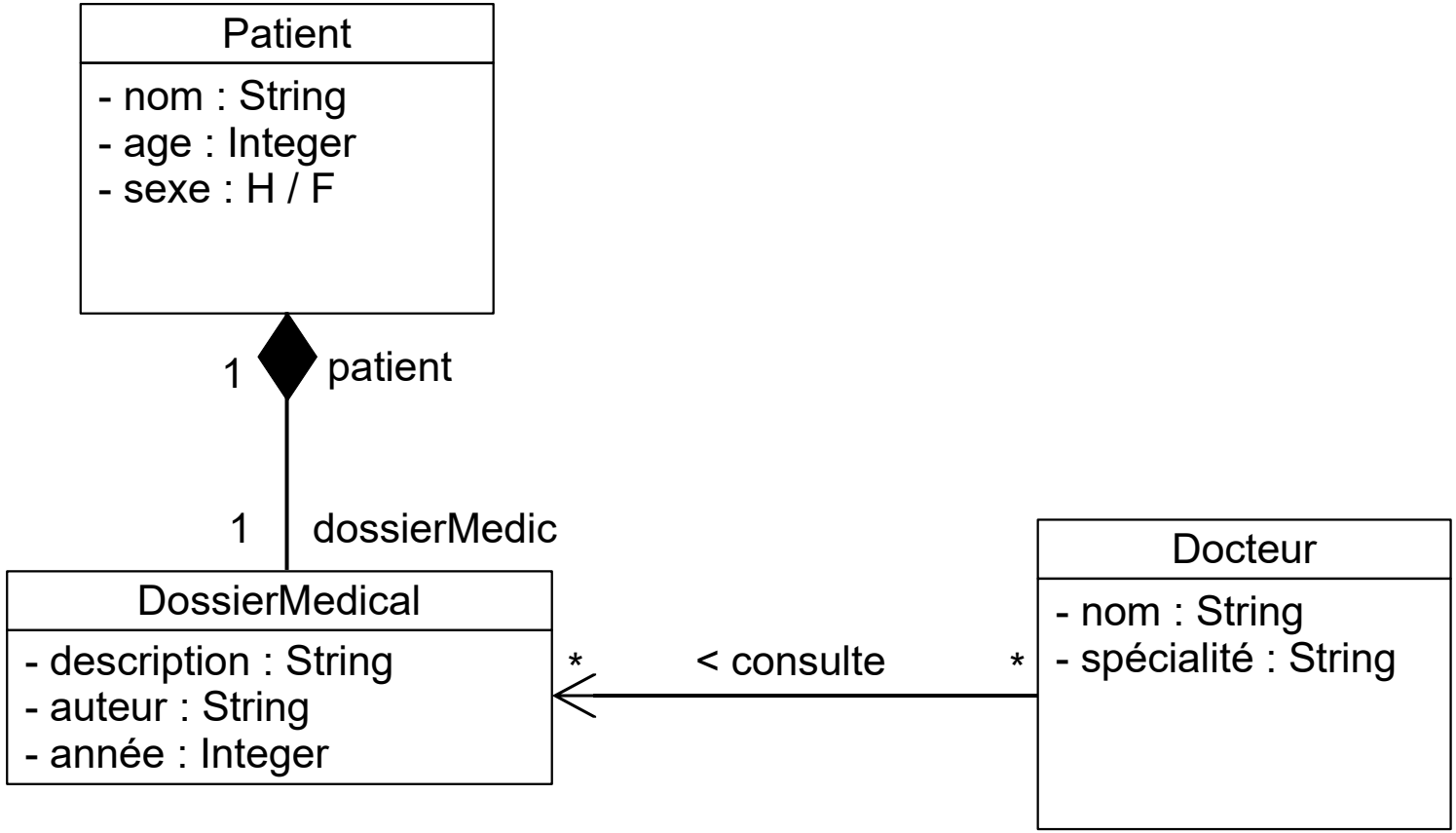
- *Dans ces conditions il est **important** d'indiquer cette sémantique en utilisant le symbole de **composition***



Un dossier médical est intimement lié à la personne qu'il représente, il ne peut pas devenir le dossier médical de quelqu'un d'autre !
Si on détruit (l'objet informatique local) Patient alors on libère (l'objet informatique local) Dossier

Les associations entre classes

- *Non partageable en tant que composant, ça n'empêche pas d'autres objets d'être en **association** avec un composant ...*

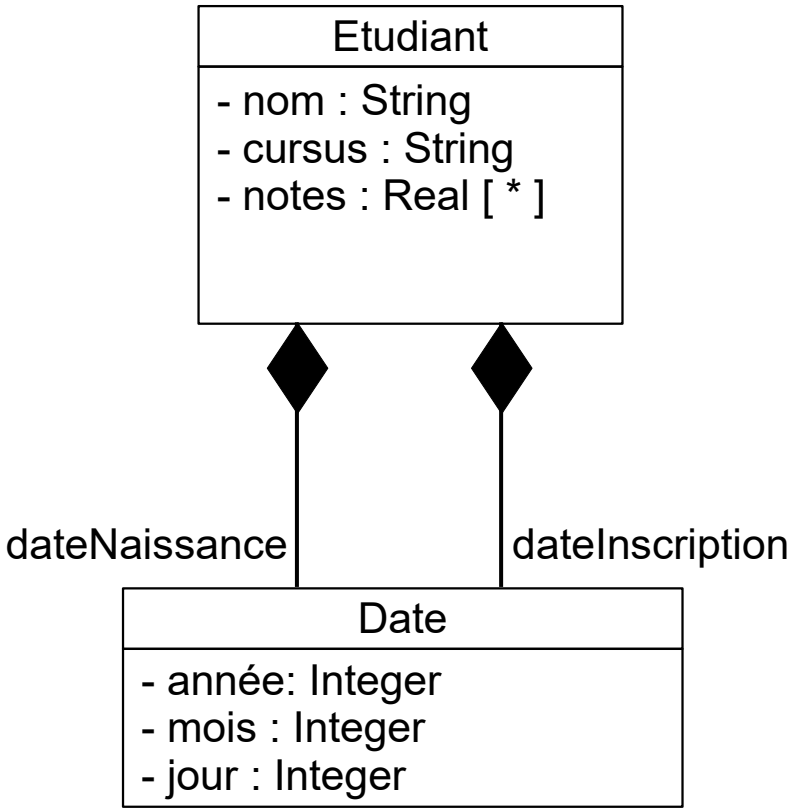


Les associations entre classes

- *Indiquer les situations de **composition** est important*
  - *Parce que c'est une contrainte forte sur le cycle de vie des instances composées par rapport au composite*
  - *Parce que du point de vue de l'implémentation la composition peut souvent (pas toujours...) se réaliser avec un attribut directement du type de l'objet composant (et non pas pointeur sur) on dira qu'on a une sémantique **par valeur***
  - *Quand les conditions sont réunies pour effectivement implémenter les relations sous forme d'attributs **par valeur** cela peut simplifier grandement la gestion du cycle de vie des objets*

Les associations entre classes

- *Exemple de composition avec une classe utilitaire et une sémantique par valeur « naturelle » ...*

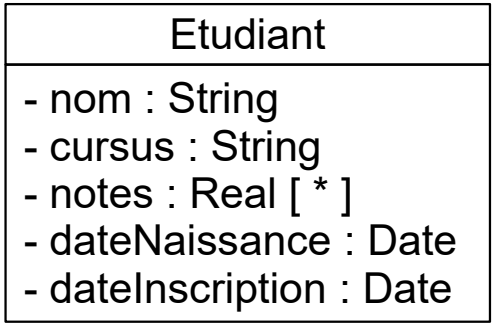


On ne précise ici ni multiplicité ni rôle en haut ni multiplicité en bas. En l'absence de multiplicité la valeur par défaut est 1, et la navigation est implicitement unidirectionnelle du composé vers le composant, ce qui est souvent le cas avec les attributs **par valeur**.

Une date **ne se partage pas**
Si elle se corrige la correction ne concerne qu'une date **pour** quelqu'un. On peut dire « on s'est trompé de date de naissance pour Olivier Martin »
On ne peut pas dire « on s'est trompé le 5 Janvier 2000 était en fait le 6 Janvier 2000 pour tout le Monde ! »

Les associations entre classes

- *Notons au passage que le fait d'utiliser une classe relativement triviale et manipulée par valeur autorise ici probablement de la considérer en attribut direct*



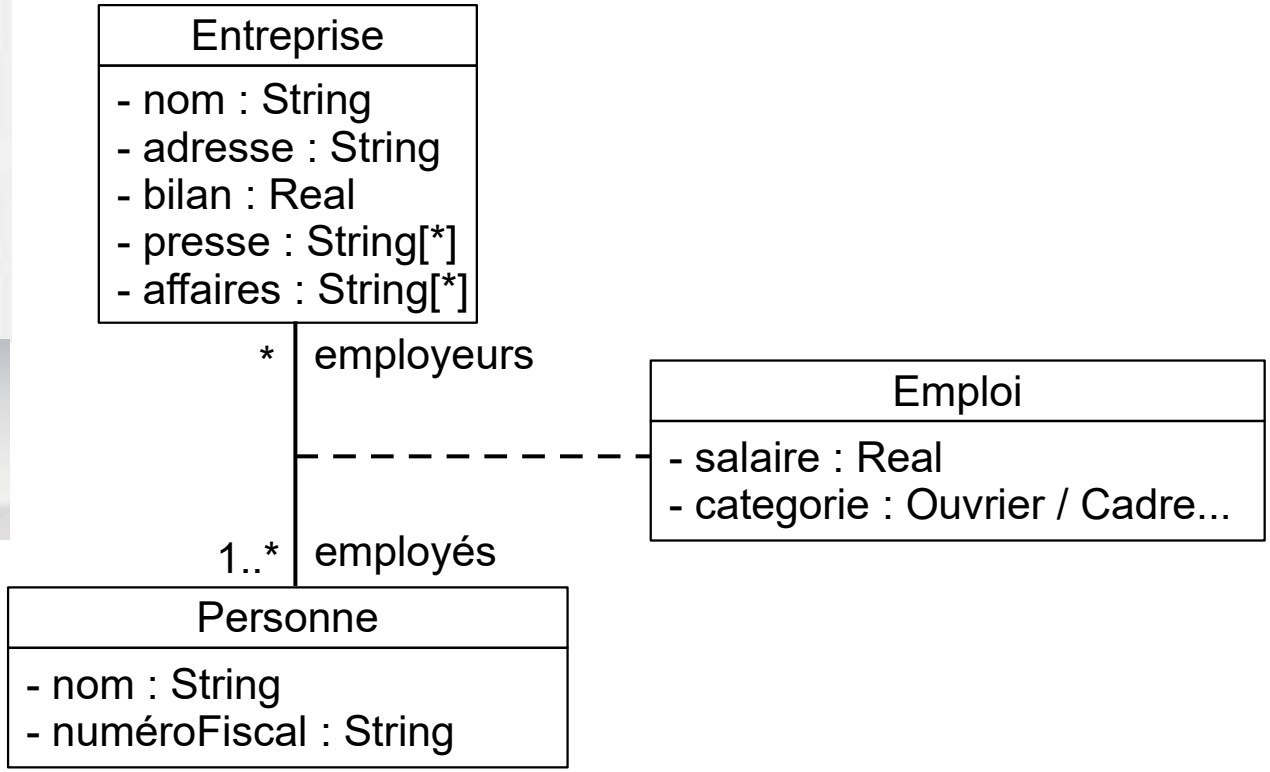
On allège le diagramme !

```
class Date
{
   private :
      int annee, mois, jour ;
   ...
};
```

```
class Etudiant
{
   private :
      ...
      Date dateNaissance ;
      Date dateInscription ;
   ...
};
```

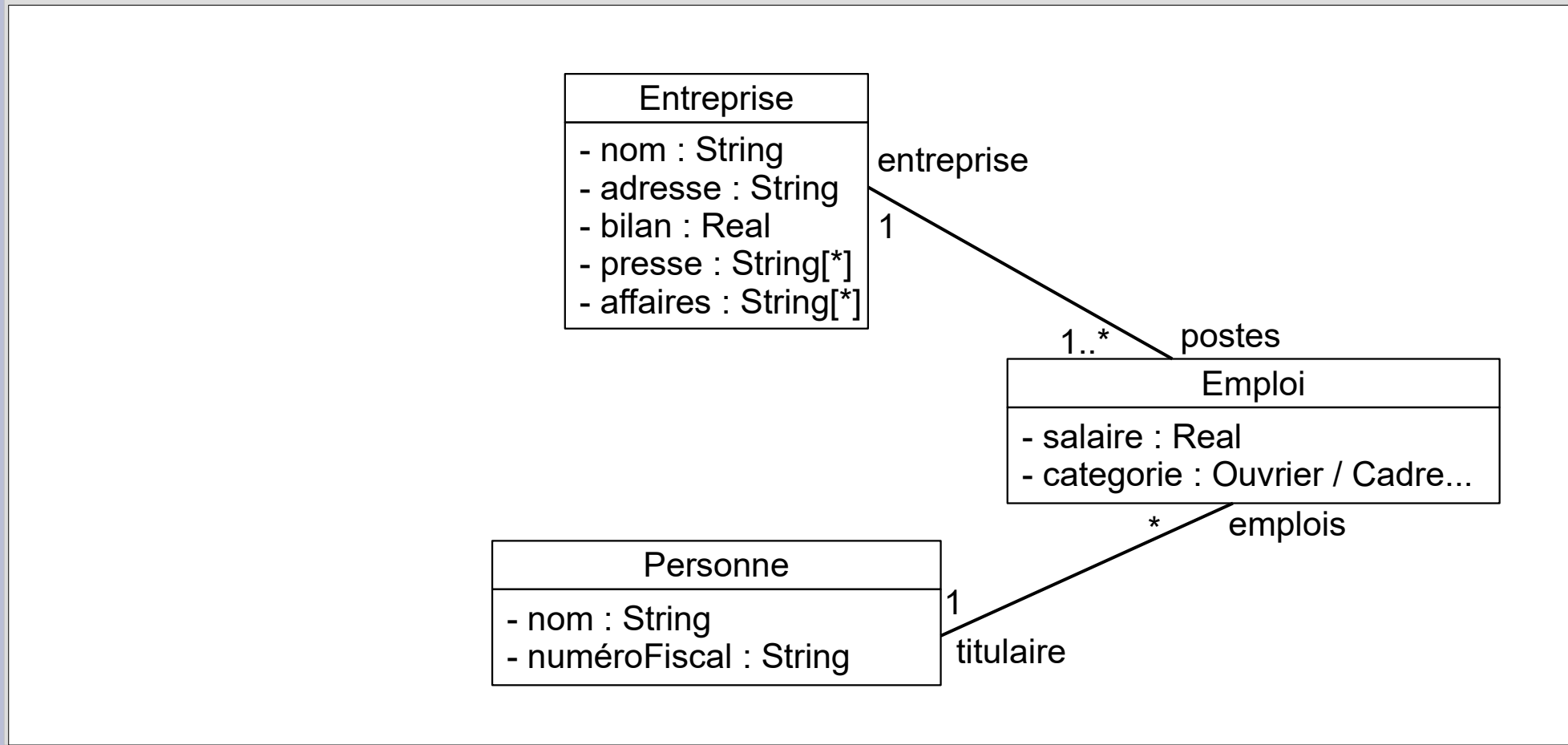
Les associations entre classes

- *Chaque « instance d'association » peut nécessiter de porter des données, on a une* ***classe association***



Les associations entre classes

- *On peut transformer en diagramme **équivalent** et on se retrouve avec des associations usuelles qu'on sait implémenter concrètement*



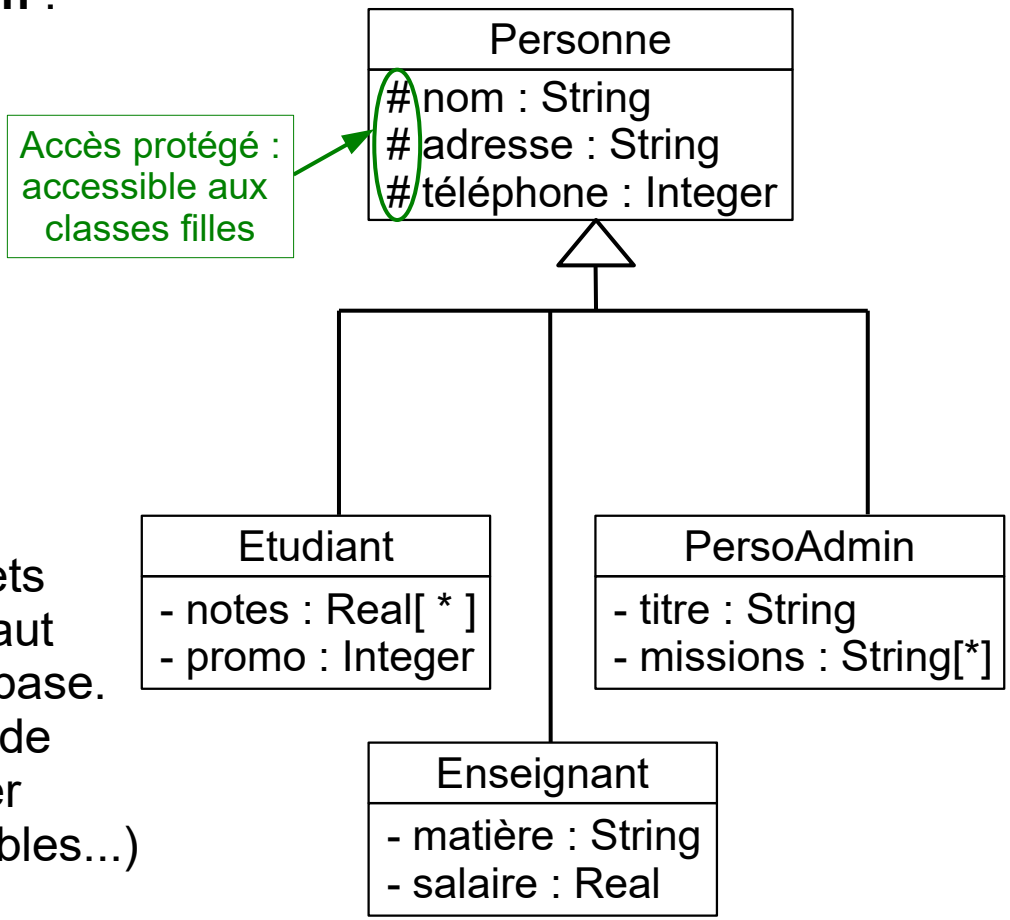
COURS 2

A) UML et méthodes dev. objet
B) Diagramme de classes en UML
C) La classe en UML !
D) Les associations entre classes
E) L’héritage et le polymorphisme
F) Compléments

L'héritage et le polymorphisme

- *L'**héritage** permet de dériver une/des classe(s) fille(s) d'une classe mère ou classe « de base »*

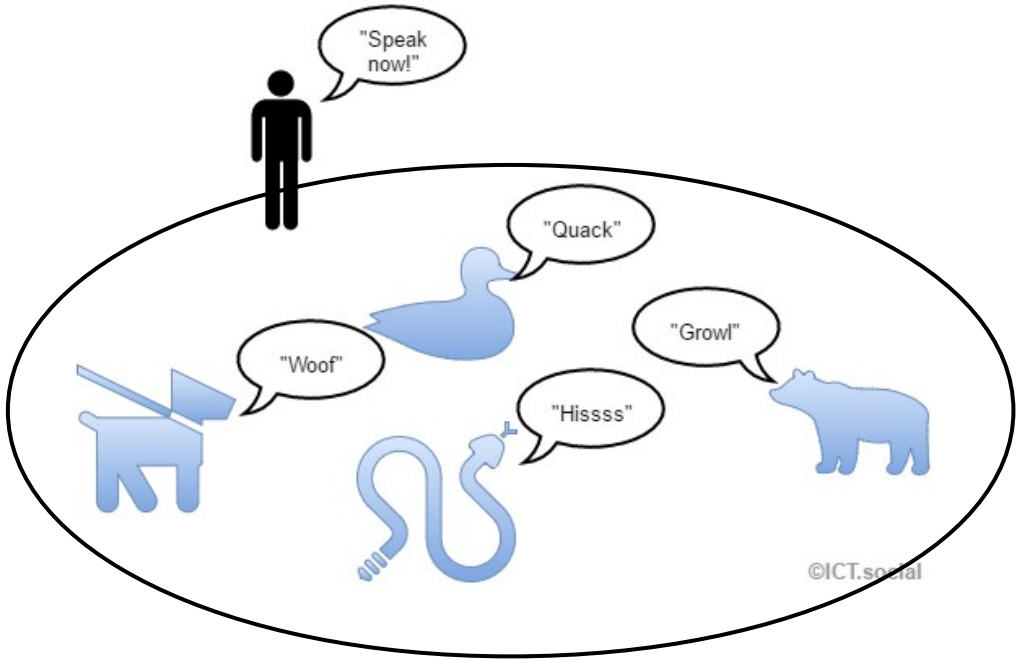
La relation d'héritage est une relation **est un** :
- un Etudiant **est une** Personne
- un Enseignant **est une** Personne
- un PersoAdmin **est une** Personne



Les attributs et méthodes de la classe mère se retrouvent automatiquement dans les objets des classes filles : il n'y a pas besoin (il ne faut pas) redéclarer les attributs de la classe de base. Ceci permet (entre autre) de **factoriser** le code en évitant des répétitions, ainsi que d'articuler des sémantiques ensemblistes (sous-ensembles...)

L'héritage et le polymorphisme

- *Le **polymorphisme** est le fait de pouvoir regrouper des objets des types dérivés et les traiter de façon homogène alors qu'ils correspondent à des types distincts et qu'ils ont des comportements spécifiques*



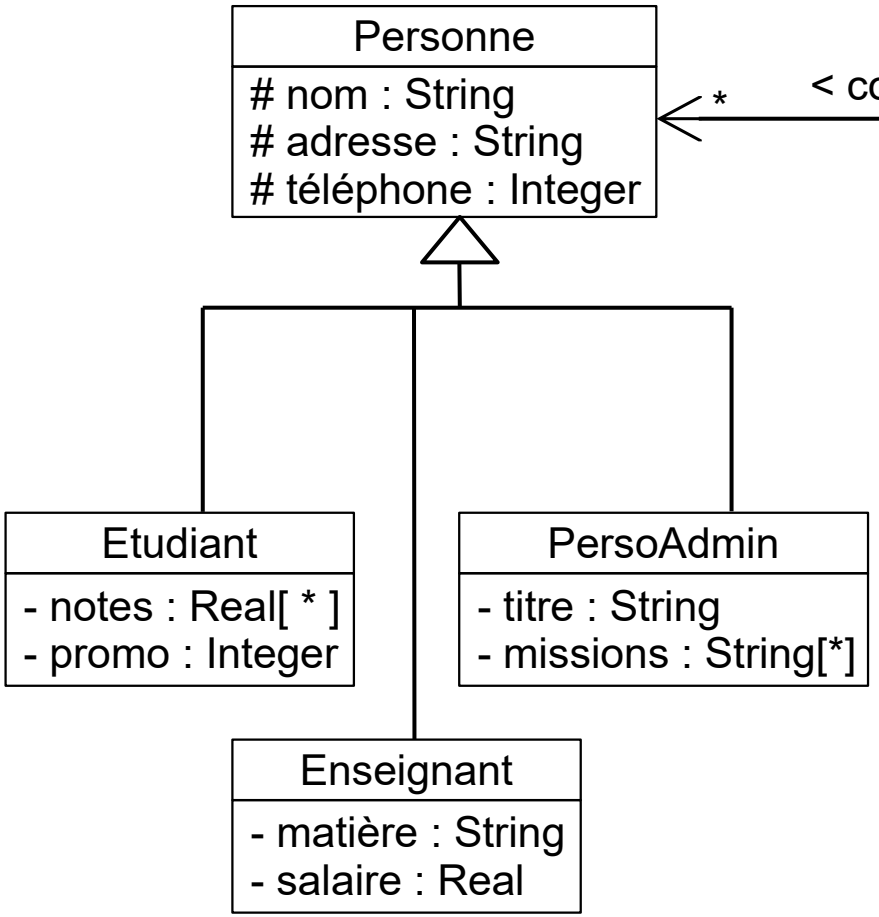
Conteneur (tableau) d'éléments de type « pointeur sur Animal » avec des instances mélangées de classes filles de Animal : Ours, Chien …

La méthode « speak » commune (classe Animal) déclenche des traitement spécialisés à chaque classe fille

- *C'est un sujet délicat sur lequel nous reviendrons !*

L'héritage et le polymorphisme

- *L'**héritage** permet grouper des éléments de types distincts dans un même « paquet »*



Une instance de AnnuaireEcole connaît des Personnes qui peuvent plus spécifiquement être des Etudiants, des Enseignants ou des PersoAdmin

COURS 2

A) UML et méthodes dev. objet
B) Diagramme de classes en UML
C) La classe en UML !
D) Les associations entre classes
E) L'héritage et le polymorphisme
F) Compléments

Compléments

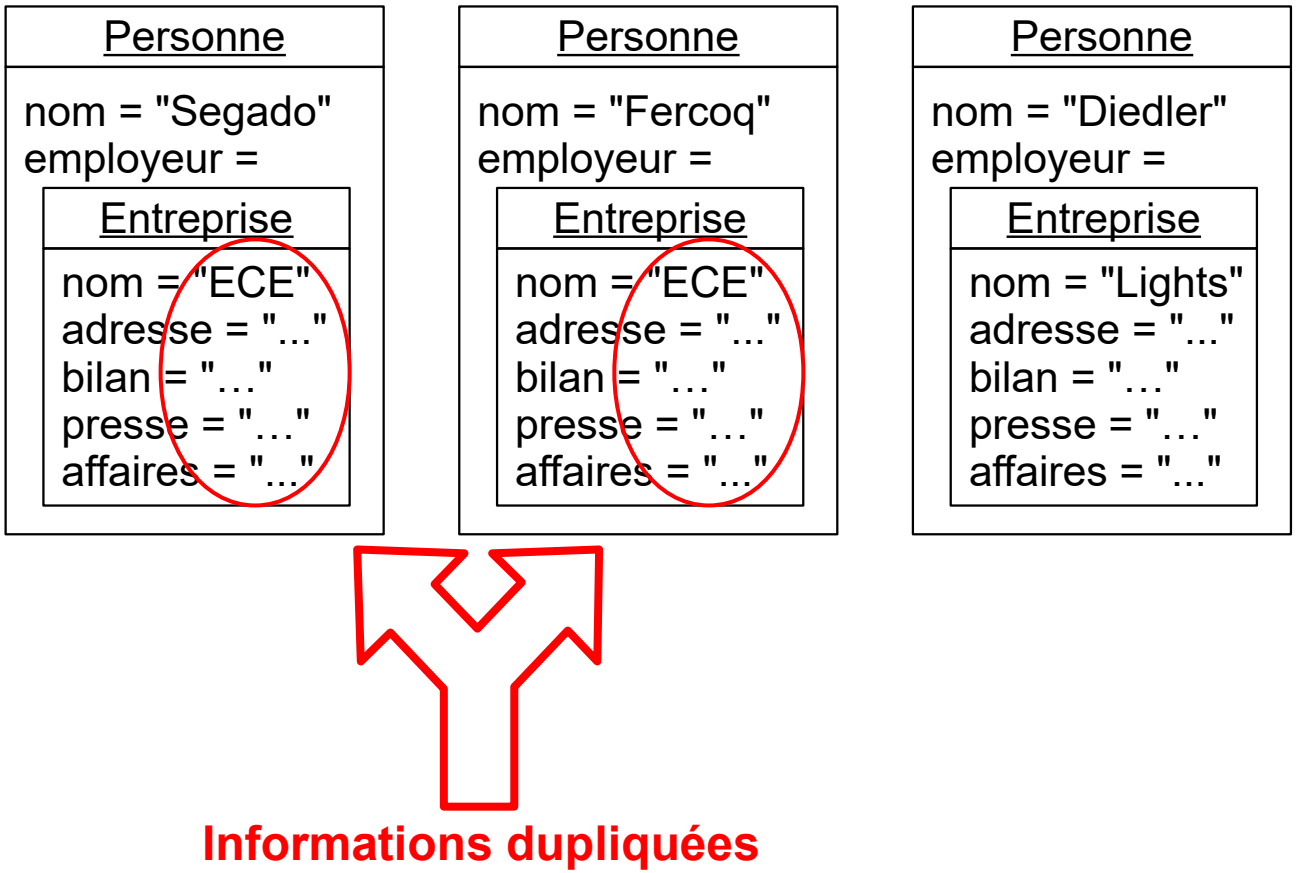
- *Les slides qui suivent sont **à lire en autonomie** selon votre motivation, il sont facultatifs...*
- *Ils précisent certains points et en particulier apportent une justification à l'emploi des pointeurs pour coder informatiquement les associations entre objets*
- *En fin de présentation vous trouverez un exo. corrigé*
- *N'hésitez pas à demander à vos chargés de TD/TP des précisions si certains concepts ne passent pas !*

Compléments

- *Reprenons la relation Entreprise / Personne*
- *Réduisons le CDC en termes de fonctionnalités et voyons comment cela donne un modèle plus simple...*
- CDC : une organisation syndicale souhaite enregistrer un certain nombre de personnes, toutes salariées, et toutes avec un seul employeur. On peut ajouter une entreprise au système (même si aucune personne y travaillant n'est connue). On ne peut ajouter une personne au système que si on précise à la fois le nom de la personne et l'entreprise unique dans laquelle il travaille. On doit ensuite pouvoir indiquer une personne et le système trouve l'entreprise dans laquelle il travaille et affiche plein d'informations (nom, adresse, chiffres comptables, actualités, affaires pénales...)

Compléments

- *Un 1er **diagramme d'objets** avec une structure qui ne convient pas : duplication d'informations...*



Compléments

- *Le schéma précédent correspondrait à l'utilisation directe d'un attribut de type Entreprise dans la classe Personne. C'est simple et « ça compile »* ***mais***

Compléments

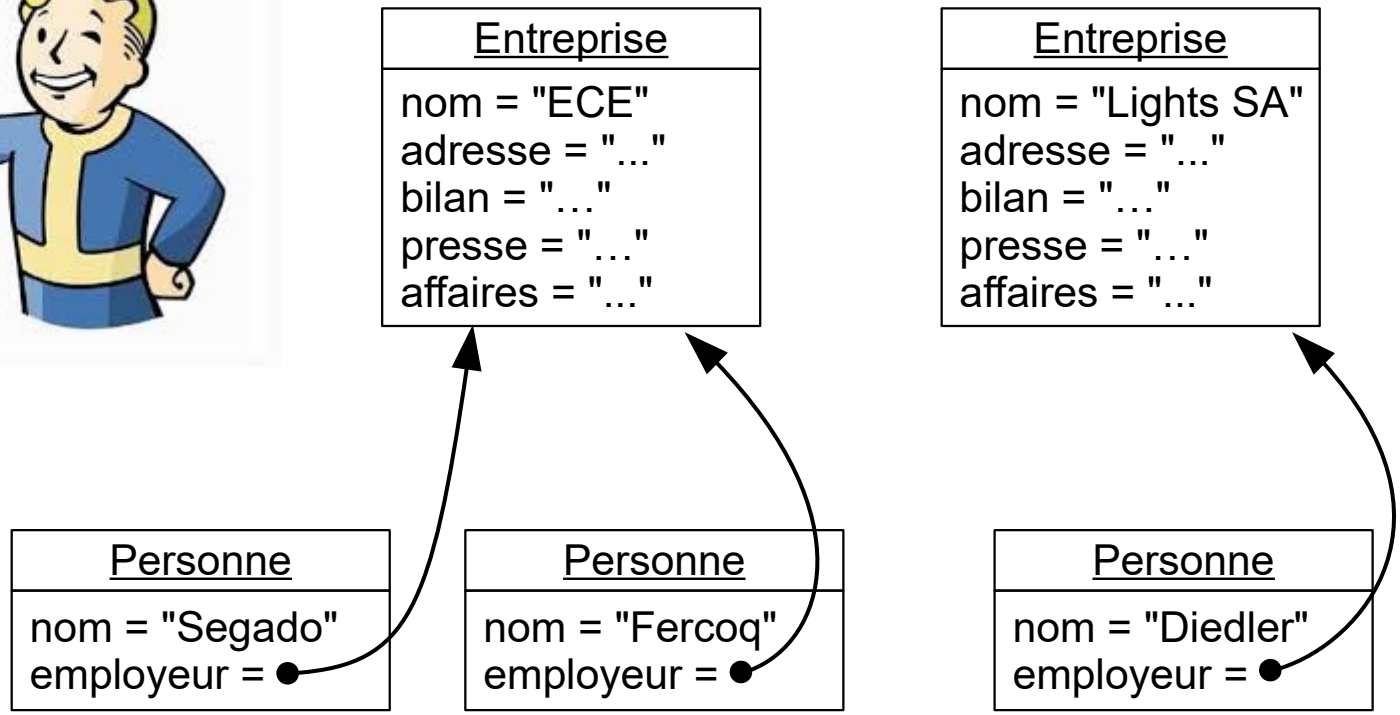
- *Une telle duplication d'information au sein d'un logiciel en cours d'exécution (ou au sein d'une même BDD) est **en général inacceptable** :*
  - *Le volume stocké est multiplié (passe encore...)*
  - *A une seule entité du « monde réel » devrait logiquement correspondre un seul objet logiciel sinon c'est la confusion dans la conception*
  - *La mise à jour d'une donnée sur une entreprise nécessite la synchronisation avec toutes les instances représentant cette entreprise*
    - *c'est une surcharge pour le hardware (bof...)*
    - *c'est la catastrophe assurée → des données similaires vont devenir **incohérentes**...*

Compléments

- *Ce problème est central en informatique*
- *On a utilisé une **sémantique par valeur** pour des données*
  - *non constantes*
    des données d'une Entreprise peuvent et vont changer
  - *qui sont partagées*
    plusieurs Personnes se rapportent à la même Entreprise
- *La solution : ne pas dupliquer l'objet entreprise, faire en sorte que les objets en relation avec lui le désigne*
  *→ passer à une **sémantique par référence***
- *Concrètement : utiliser des **pointeurs sur objets !***
- *Il existe des alternatives : clés, clés uniques...*

Compléments

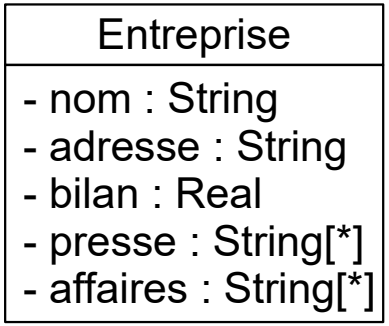
- *La même situation gérée correctement avec des pointeurs : nouveau **diagramme d'objets***



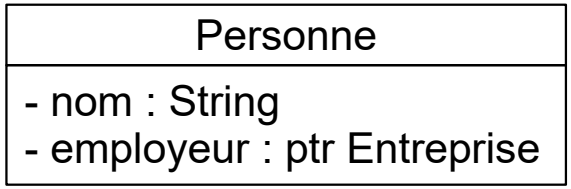
Compléments

- *Le schéma précédent correspond à l'utilisation d'un attribut de type pointeur sur Entreprise dans la classe Personne. C'est simple et « ça compile » aussi !*

Diagramme de classes !



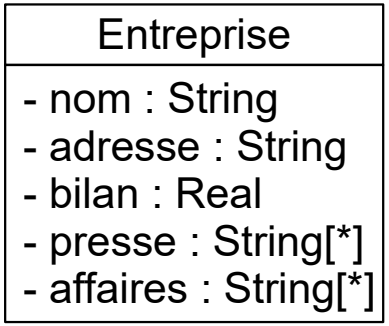
Ce diagramme de classes est correct mais pas très visuel ...



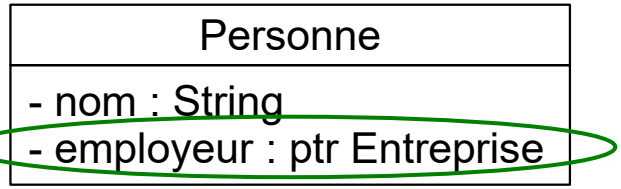
Compléments

- *Techniquement en C++ la référence d'un objet de type Personne sur un objet de type Entreprise est un* ***attribut*** `Entreprise* employeur;`

Diagramme de classes !



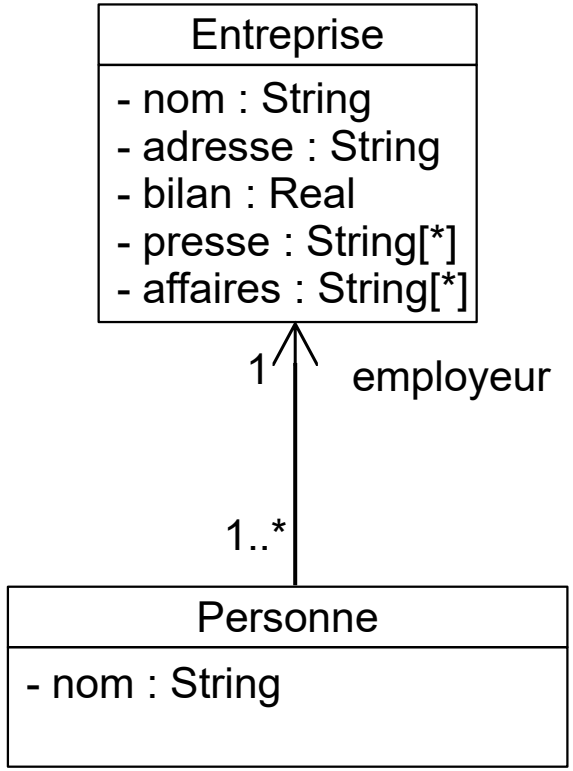
Ce diagramme de classes est correct mais pas très visuel ...



Compléments

- *Visuellement la référence sera représentée par une **liaison** : on dira que les 2 classes sont **associées***

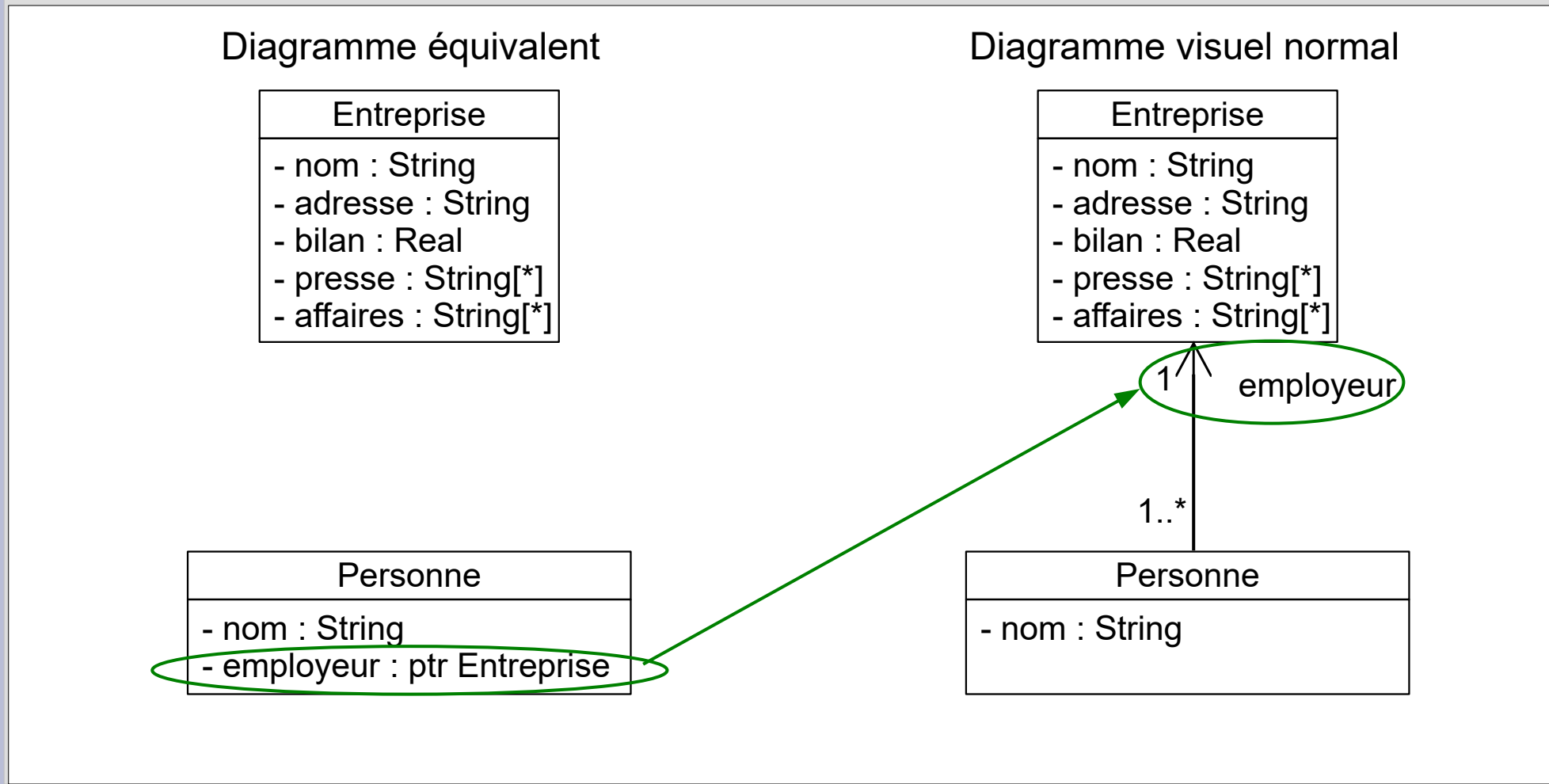
Diagramme de classes !



Ce diagramme de classes est la façon normale de représenter un attribut pointeur sur objet d'une autre classe !

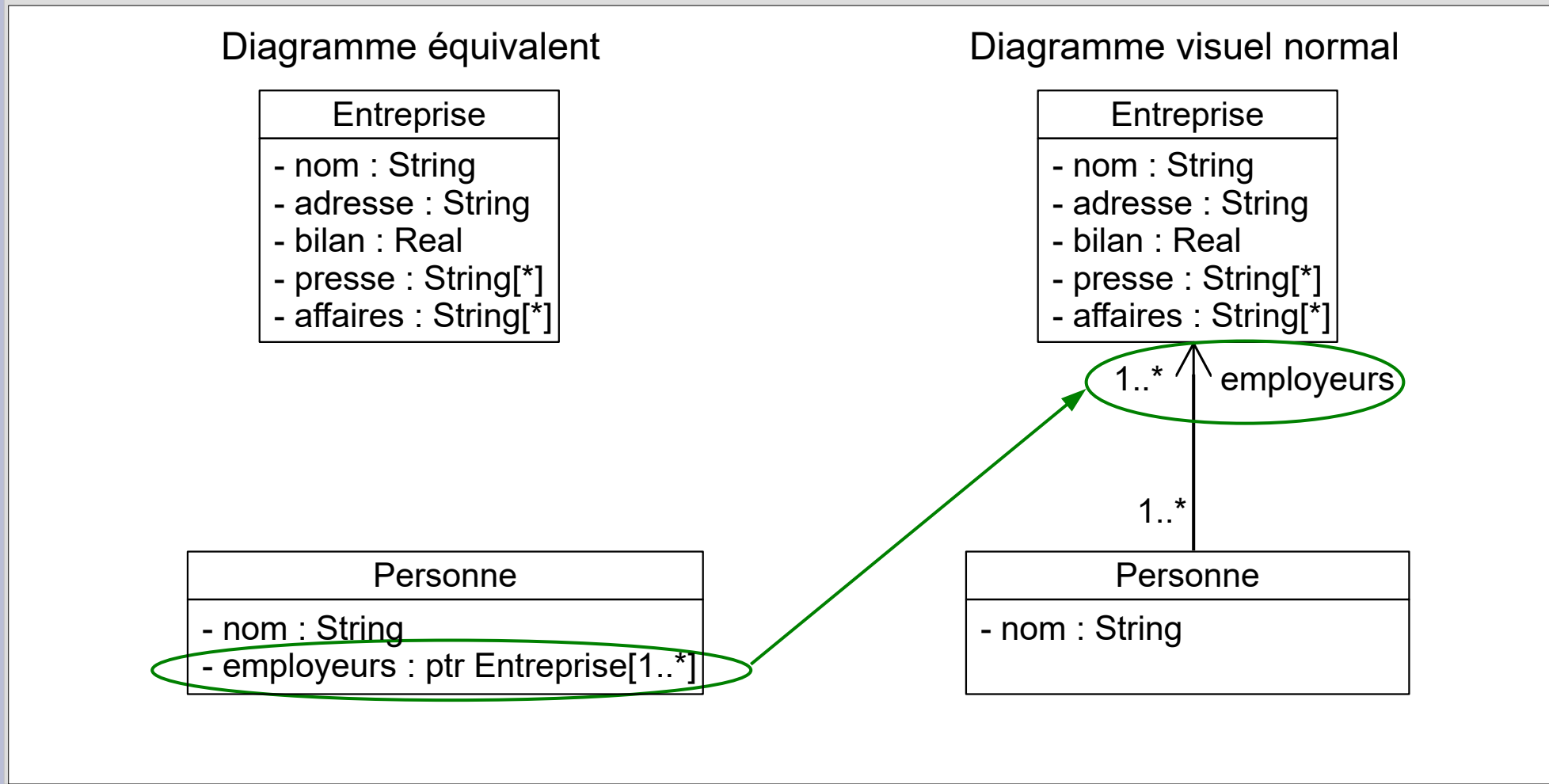
Compléments

- *Ici **l'association** se **navigue** dans un seul sens*
- *Les informations de l'attribut « référence à l'objet » se retrouvent à l'autre extrémité de l'association*



Compléments

- *Si on modifie le CDC : une personne peut avoir plusieurs employeurs, on arrive à un attribut tableau de pointeurs (en C++ nous aurons un* ***`std::vector`*** *)*

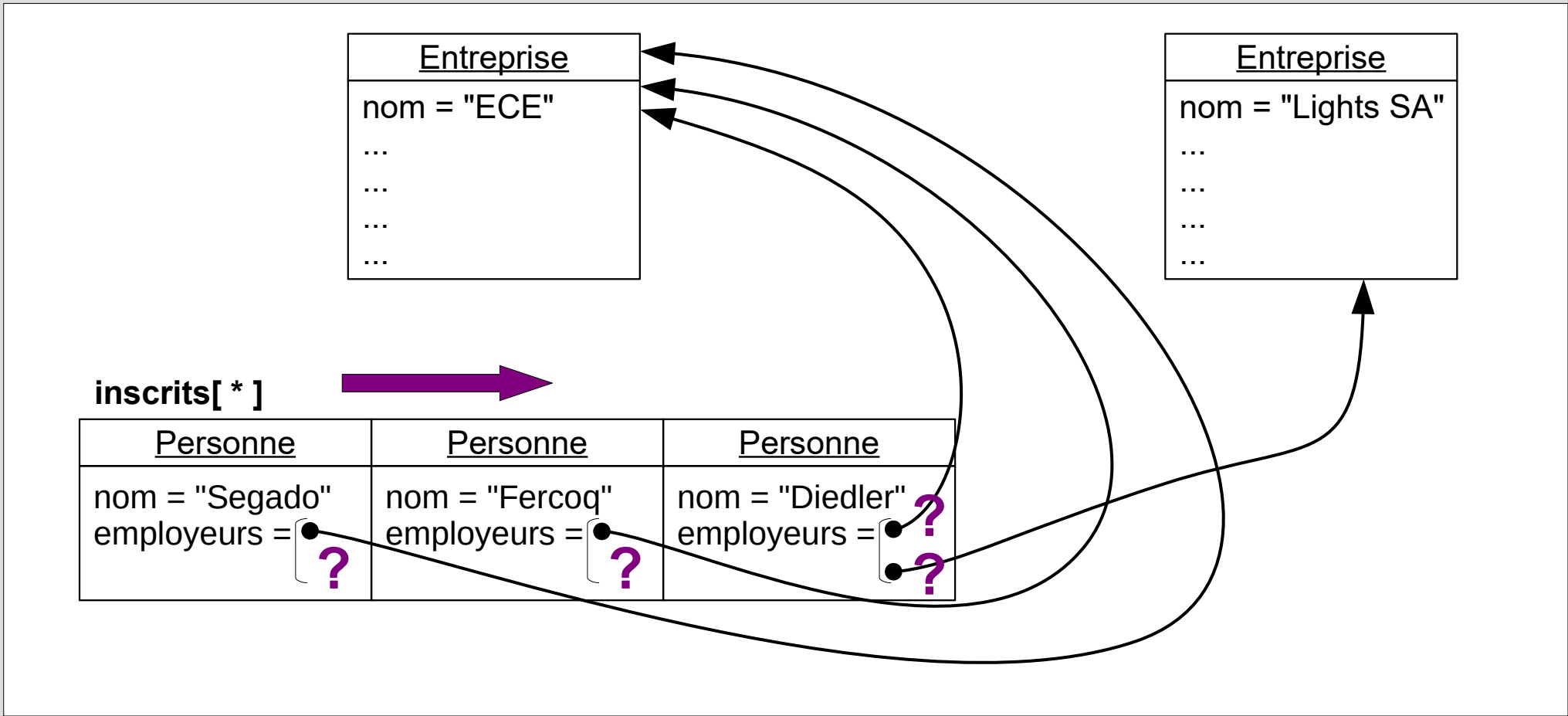


Compléments

- *On modifie encore le CDC... Ça arrive tout le temps !*
- *une personne peut avoir 0 1 ou plusieurs employeurs*
- *connaissant une entreprise on veut pouvoir avoir accès à toutes les personnes y travaillant...*

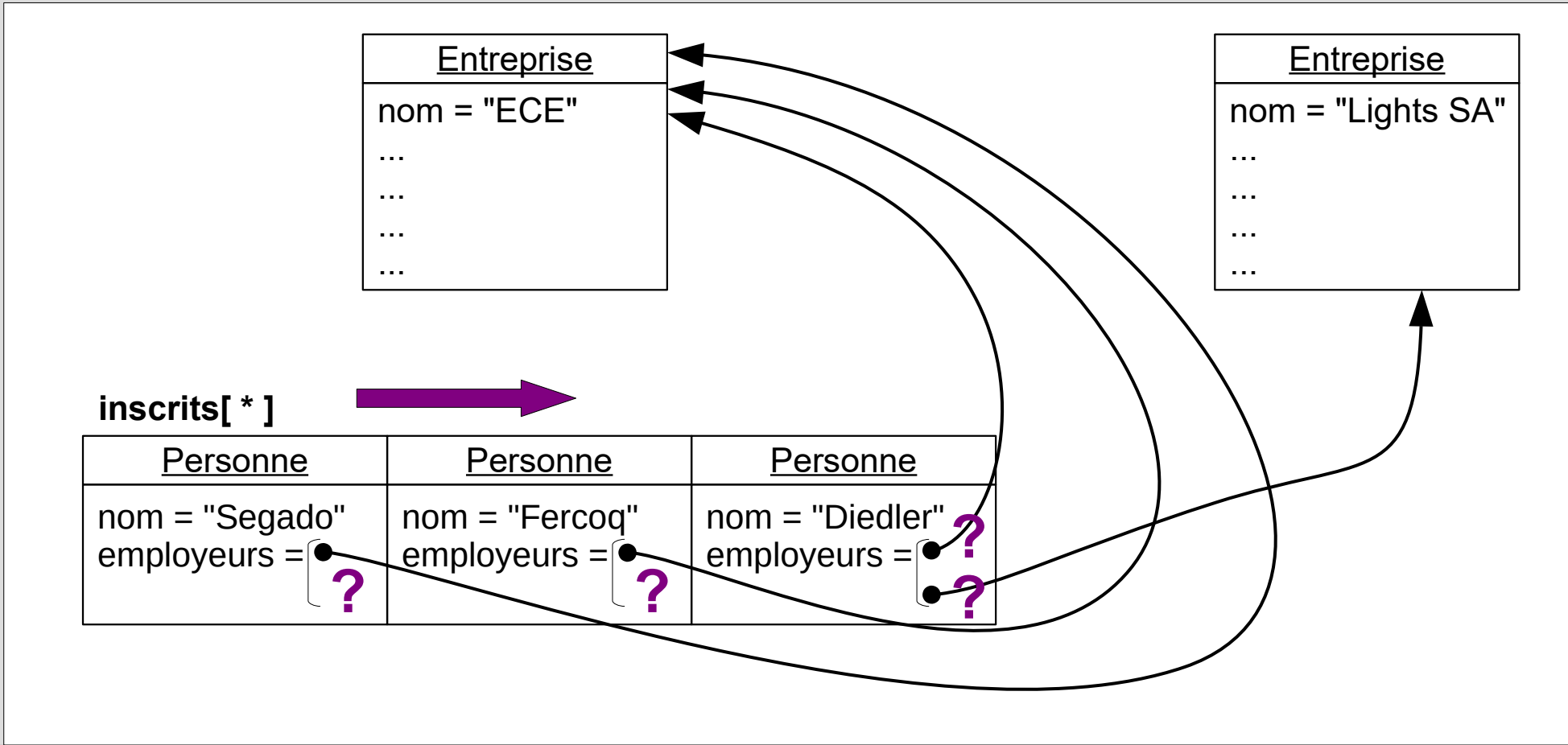
Compléments

- *On peut trouver toutes les personnes travaillant dans une certaine Entreprise si on a une collection (liste, tableau, vecteur peu importe) de tous les inscrits : il suffit de parcourir cette liste et de tester à chaque fois*



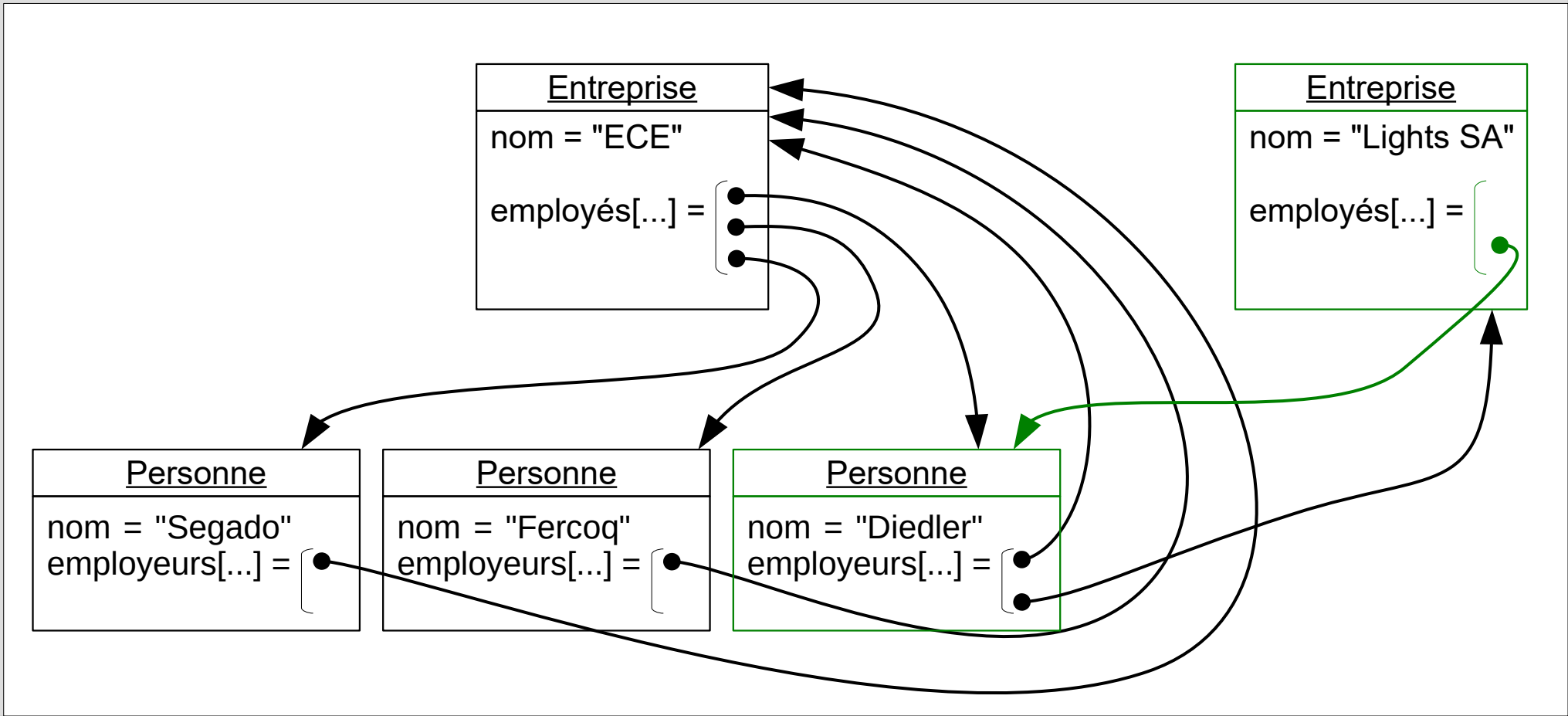
Compléments

- *Mais cette approche n'est pas efficace si le nombre d'inscrits est important et si cette recherche a lieu souvent !*



Compléments

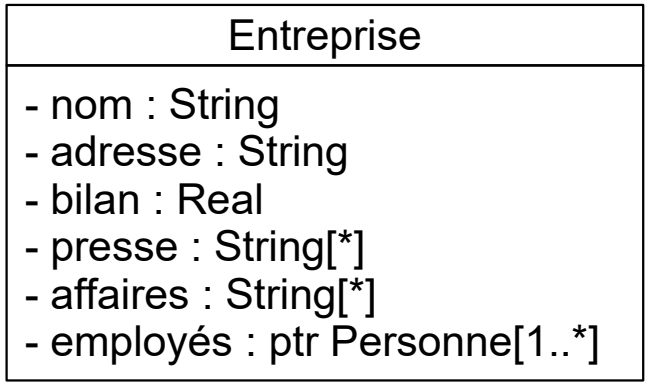
- *Si l'approche précédente est trop pénalisante on peut adopter une **navigation à double sens** beaucoup plus performante sur les recherche mais plus lourde à mettre en place et à maintenir*



Compléments

- *Ce qui conduit à une association simple avec **navigation à double sens** beaucoup plus performante sur les recherche mais plus lourde à mettre en place et à maintenir*

Diagramme équivalent



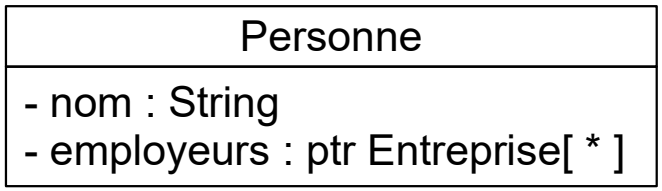
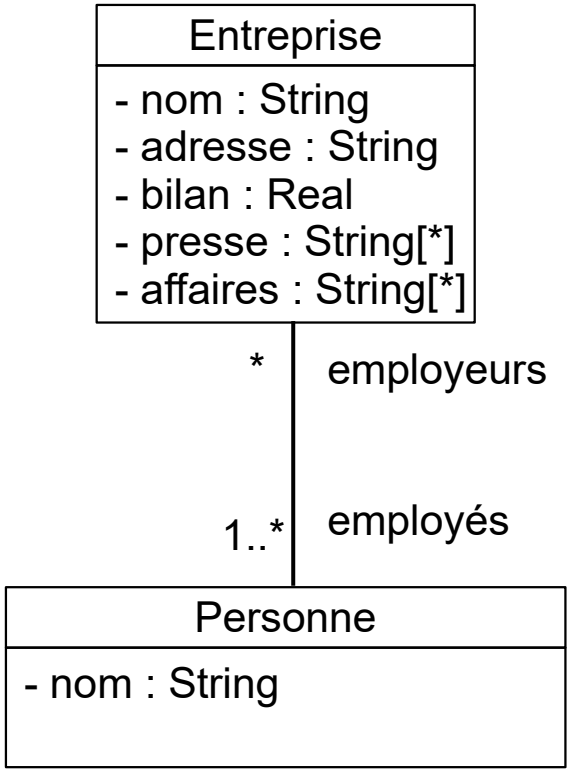


Diagramme visuel normal



Compléments

- *La **navigation à double sens** est la navigation par défaut d'une association quand aucune flèche n'est spécifiée. A n'utiliser qu'en cas de nécessité car nettement plus long à bien implémenter ensuite.*

Diagramme équivalent

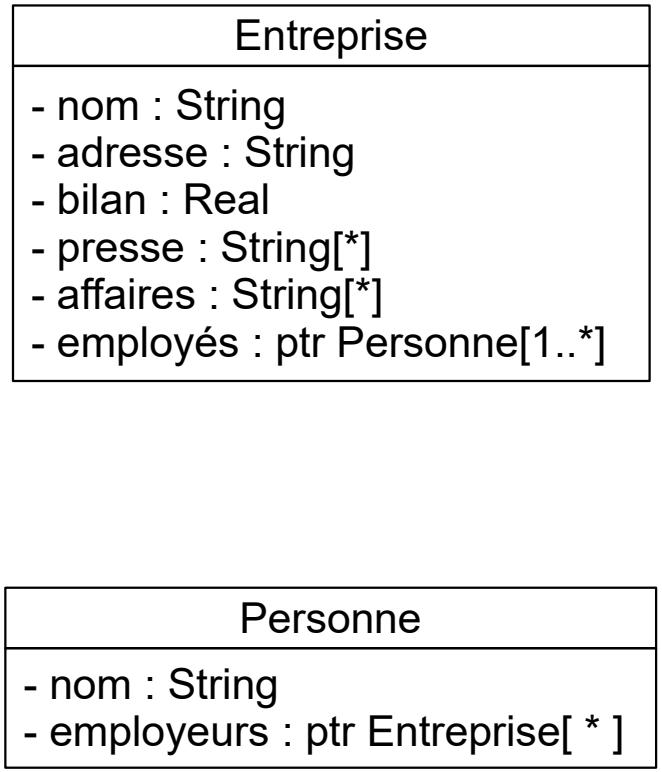
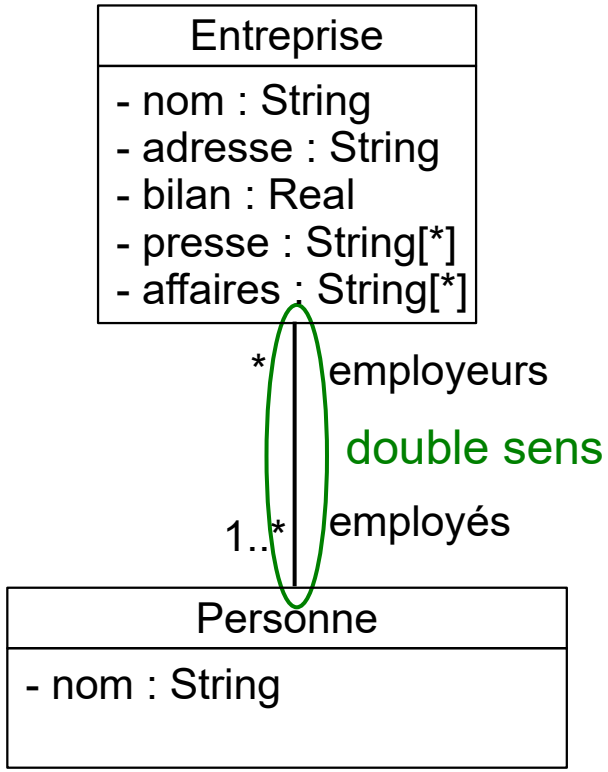


Diagramme visuel normal

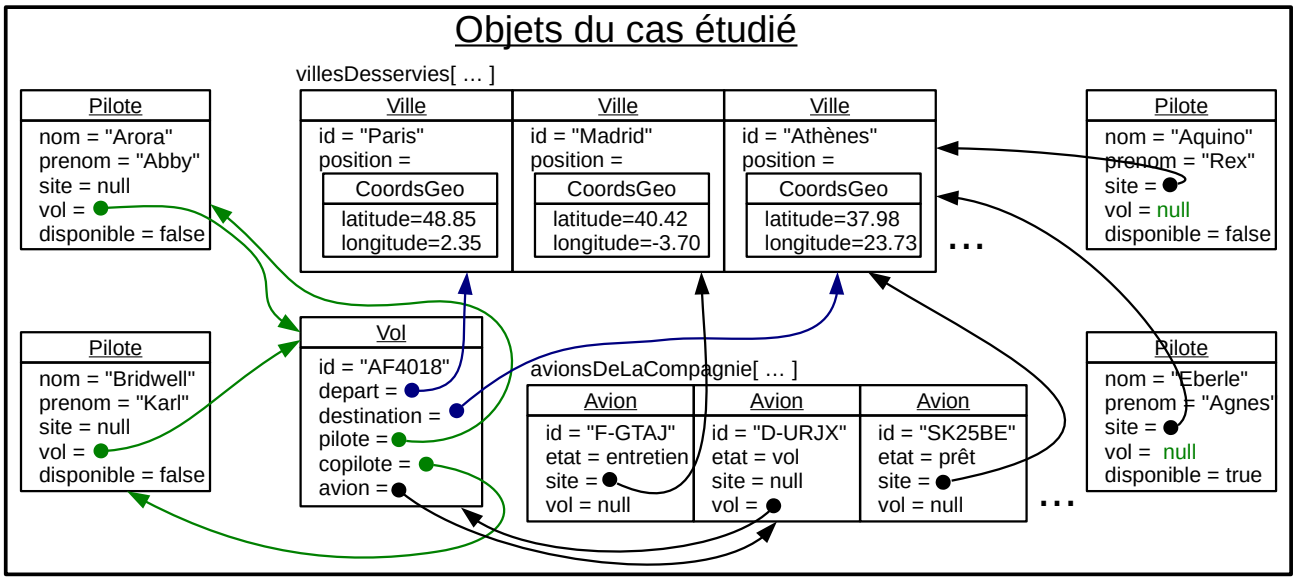


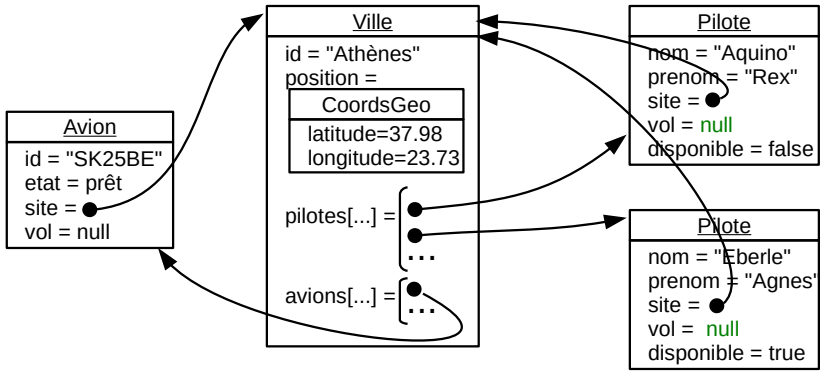
Compléments

Exercice corrigé

- *Faire le diagramme de classe du diagramme d'objets slide suivant.*
- *Vous indiquerez les multiplicité mais pas les rôles*
- *Le CDC est au début du TD/TP 1*
- *Attention spoiler, le corrigé est au slide d'après...*

Compléments

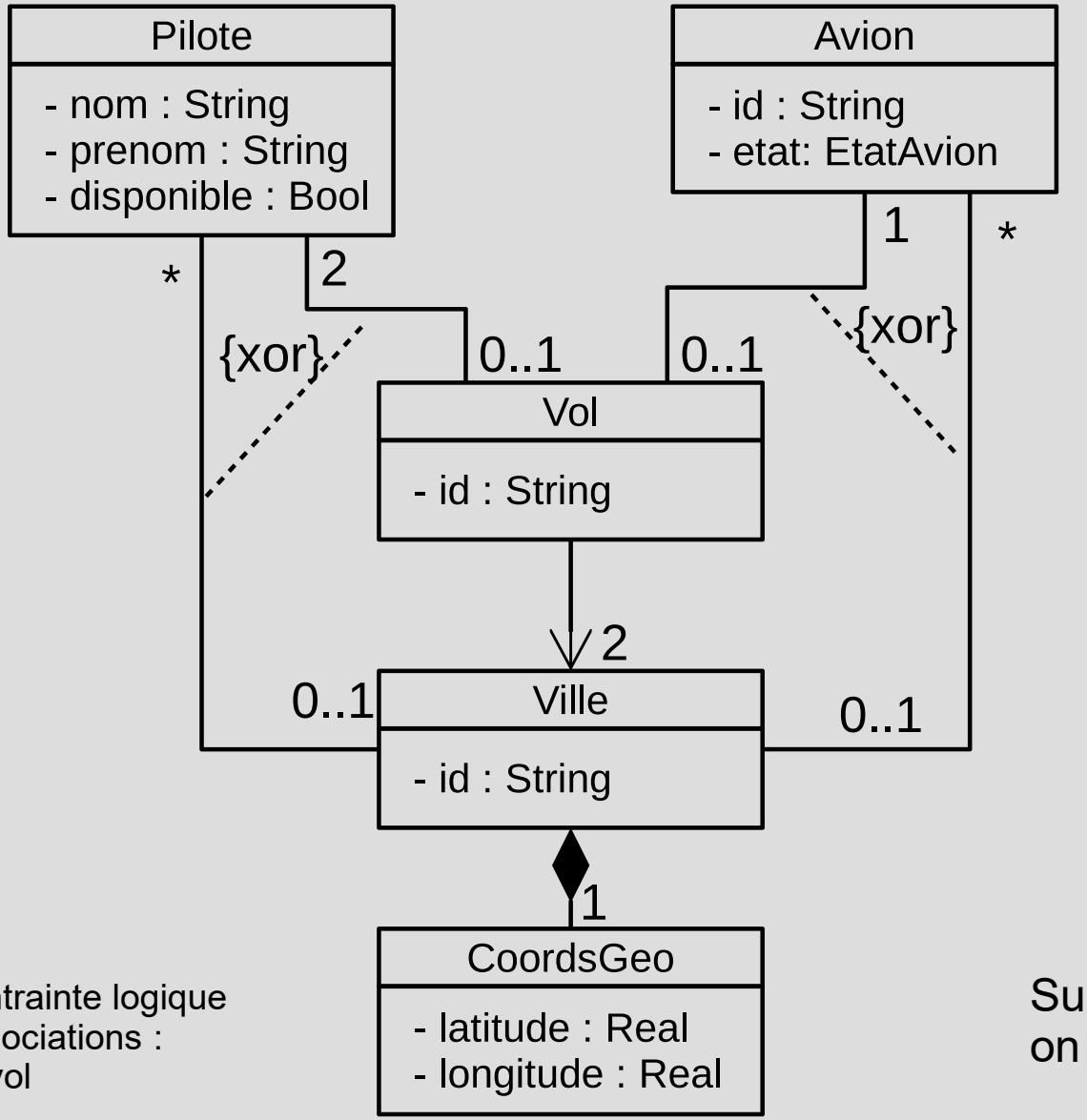




Au modèle ci-dessus
on a ajouté ces navigations
à double sens

Compléments

Corrigé de l'exercice



{xor} indique une contrainte logique d'exclusion entre associations : un avion est soit en vol soit dans une ville

Sur ce diagramme on a omis les rôles...