DWWM
Cours complémentaire
UML2
Diagramme de classe

Module 2

Table des matières

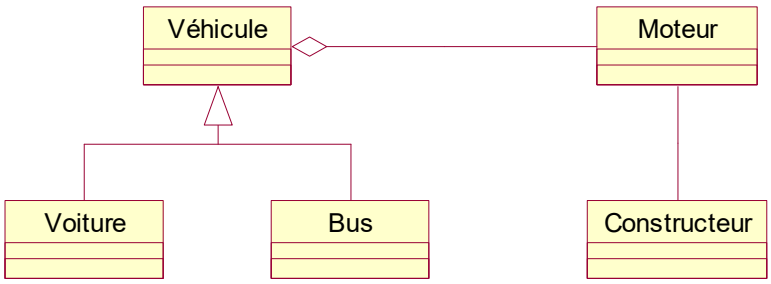
Les classes....3
Attributs....3
Exemple : classe Fenêtre....4
Méthodes (ou Opérations en analyse)....5
Diagrammes d'objets....6
Liens et associations....7
Association ternaire (3 pattes)....8
Multiplicité....10
Rôle....11
Contrainte....12
Contrainte d'exclusion....13
Contrainte d'ordre....14
Agrégation....15
Composition....16
Agrégat récursif....17
Généralisation et héritage....18
Contraintes sur la généralisation....19
Note....20
Package ou « paquetage »en French !....21
Stéréotype....22
Pattern (modèle)....22
Attribut d'association....23
Qualification....24
Modélisation et code java....26

Diagrammes de classes et d'objets

Le diagramme de classe permet de représenter les éléments de modélisation **STATIQUE** :

- Classes
- Attributs et méthodes des classes
- Relations entre les classes

Un diagramme de classe est constitué essentiellement de classes en relation les unes avec les autres. L'ASSOCIATION constitue un type de relation entre deux classes.

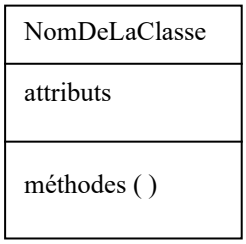


Les classes

Une *classe* décrit un groupe d'objets ayant des propriétés similaires (attributs), un comportement commun (méthodes ou opérations), des relations communes avec les autres objets ainsi qu'une même sémantique. On peut dire d'une classe qu'elle définit ce par quoi les objets sont caractérisés et comment ils se comportent.

Représentation graphique d'une classe :

Une classe est représentée par un carré divisé en trois parties horizontales que l'on peut appeler Cartouche ou Compartiment.



Le compartiment du haut contient le **NOM** de la classe

Le compartiment du milieu contient la liste des **ATTRIBUTS**

Le compartiment du bas contient la liste des **METHODES**

*Dans le **nom de la classe**, chaque mot commence par une **majuscule**.*

Attributs

Un *attribut* est une valeur de donnée détenue par les objets d'une classe. Chaque attribut est caractérisé par un NOM et un TYPE de valeur ou de donnée que l'attribut peut stocker. L'attribut est unique à l'intérieur d'une classe (par opposition au fait d'être unique parmi toutes les classes). Ainsi la classe *Personne* et la classe *Société* peuvent avoir chacune un attribut *adresse*.

Syntaxe complète :

visibilité **nomAttribut** : type = valeur-initiale

nomAttribut : chaque mot commence par une majuscule, sauf le premier

nomDeMonAttribut

Visibilité : marqueur optionnel utilisé en conception

+ ***public*** accessible à tous

protégé accessible à la classe et aux classes qui en héritent

- ***privé*** accessible à la classe seule (valeur par défaut).

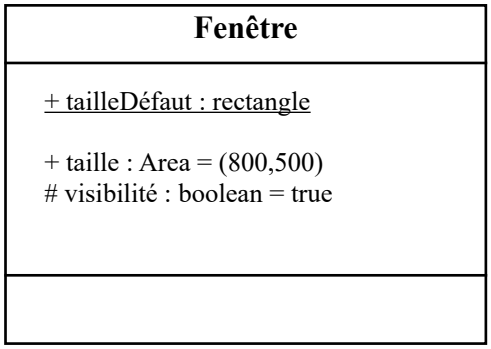
~ ***package*** accessible au niveau du paquetage

Type : type d'un langage de programmation en conception détaillée.

Pour Java, on peut spécifier String pour du texte, int pour un entier, …

Valeur-initiale : servira à donner la valeur initiale d'un attribut lors de la création d'un nouvel objet.

Exemple : classe Fenêtre



L'attribut de classe est une sorte de variable globale pour la classe. C'est en fait une ***valeur unique partagée par tous les objets de la classe et accessible à tous***, ce qui évite les recopies et les risques d'incohérence.

Pour le distinguer, il est souvent précédé du symbole dollar **$** ou bien il est souligné comme dans la figure d'à côté.

Méthodes (ou Opérations en analyse)
Une méthode est un service que l'on peut demander à un objet pour réaliser un comportement. Tous les objets d'une classe partagent les mêmes méthodes. Si, en analyse, on parle plutôt *d'opération*, en conception, on parle de *méthode*. En fait, une méthode est l'algorithme qui produit le résultat de l'opération.
Syntaxe complète :

visibilité **nom** (liste-paramètres) : type-retour

nom : chaque mot commence par une majuscule, sauf le premier.

nomDeOpération ( )

Visibilité : marqueur optionnel utilisé en conception

+	public	accessible à tous (valeur par défaut)
#	protégé	accessible à la classe et aux classes qui en héritent
-	privé	accessible à la classe seule

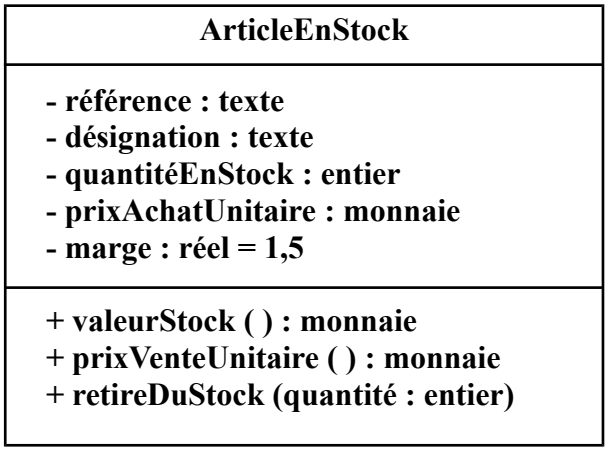
Liste-paramètres : paramètre formel spécifié comme ci-dessous :

Nom : type = valeur-défaut

Quand une opération est décrite avec la liste de ses arguments accompagnés de leurs types, on parle de ***signature*** de la méthode. Le premier argument désigne le type du receveur.

nom-opération (type-receveur, type-arg1, …, type-argn) : type-résultat

Type-retour : optionnel, si la méthode en retourne un, c'est une fonction. Si la méthode ne retourne pas de résultat, c'est une procédure (*void* du C++, C# ou Java).



Diagrammes d'objets

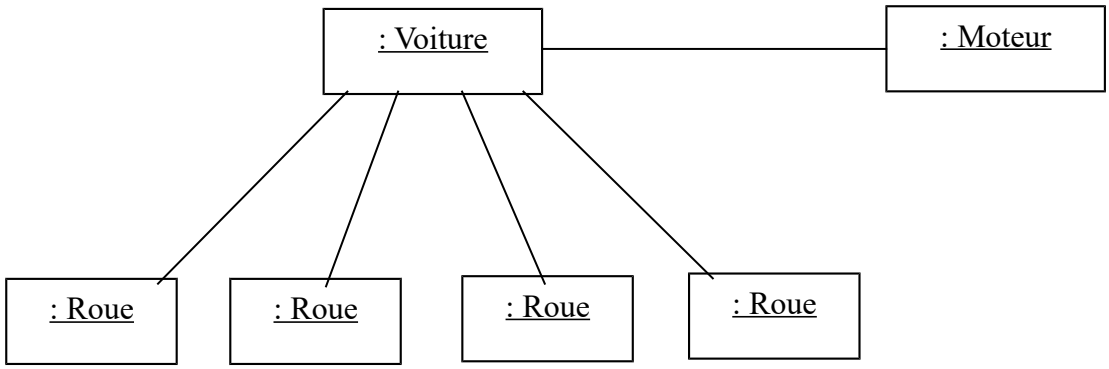
Les diagrammes d'objets quelquefois appelés diagramme d'instances montrent des objets et des liens. Ils s'utilisent principalement pour montrer un CONTEXTE ou pour FACILITER la COMPRÉHENSION des structures de données complexes.

Un diagramme d'objets se distingue graphiquement d'un diagramme de classes car les ***noms d'objets sont soulignés***.

Il existe 3 façons de représenter un objet :

Nom de l'objet	Le nom de l'objet seul correspond à une modélisation incomplète dans laquelle la classe de l'objet n'a pas encore été précisée.
Nom de l'objet : Classe	Le nom de l'objet et le nom de sa classe séparés par un ***double point***.
: Classe	La classe seule (l'objet est alors dit **ANONYME**).

Exemple de diagramme d'objets montrant une partie de la structure générale des voitures.



Dans ce diagramme :

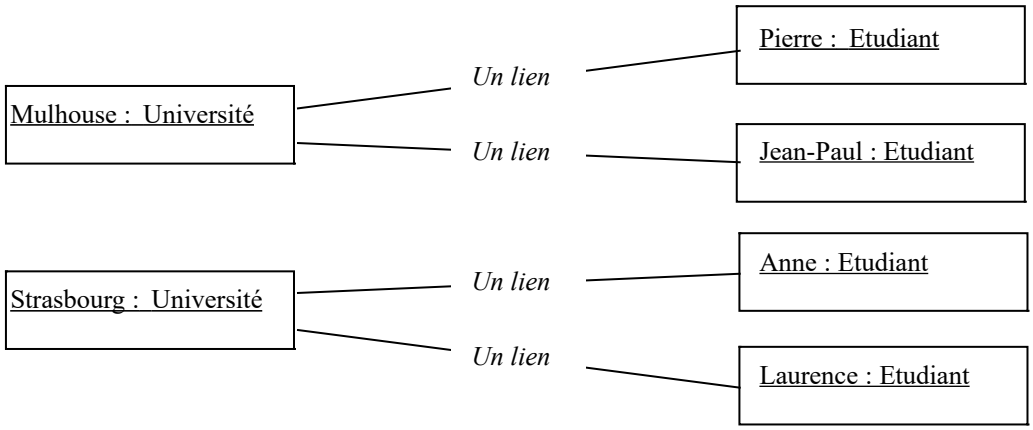
- 4 objets Anonymes de la classe Roue sont liés à l'objet Anonyme de la classe Voiture.
- 1 objet Anonyme de la classe Moteur est lié à l'objet Anonyme de la classe Voiture.

Les liens entre objets ou classes ***sont exprimés par des droites*** en contact avec chaque rectangle concerné par l'association entre les deux classes.

Liens et associations

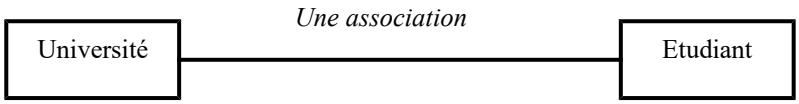
Dans un diagramme d'objets :

Un ***LIEN*** est une connexion **SÉMANTIQUE** entre des **OBJETS.**



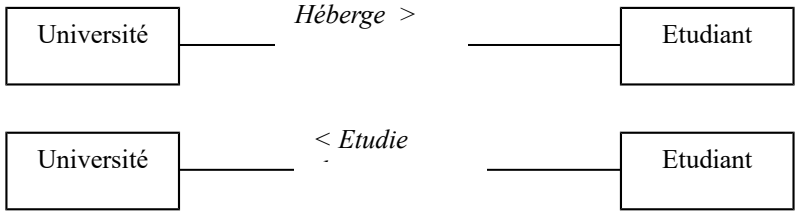
Dans un diagramme de classes :

Un ***LIEN*** est une **INSTANCE d'ASSOCIATION**. Une *association* décrit un groupe de liens ayant une structure et une **SÉMANTIQUE COMMUNE**.



Améliorer la lisibilité :

L'association peut porter un **NOM** sous forme **VERBALE** active ou passive. Le sens de la lecture peut aussi être précisé en faisant précéder le nom par le signe inférieur ou en le faisant suivre par le signe supérieur.



On peut tout simplement exprimer littéralement une association en précisant si elle concerne des classes ou des objets :

Classe Université Héberge Classe Etudiant
Classe Etudiant Etudie dans Classe Université

Objet mulhouse : Classe Université Héberge Objet pierre : Classe Etudiant
Objet anne : Classe Etudiant Etudie dans Objet strasbourg : Classe Université

Remarque:

Nous verrons que les associations sont souvent implémentées dans les langages de programmation sous forme de ***pointeurs*** d'un objet vers un autre. Un pointeur est un attribut qui contient une référence explicite vers un autre objet ou vers un ensemble d'objets.

Association ternaire (3 pattes)

Une association peut être binaire, ternaire, d'ordre quatre ou plus. Dans la pratique, la grande majorité des associations sont binaires. On rencontre peu de ternaires, et encore moins d'ordre quatre ou plus. **Il semble que certains outils de modélisation comme StarUML ne permettent pas ce type de relation**.

Une association ternaire est nécessaire quand elle ne peut pas être subdivisée en associations binaires sans perte d'information.

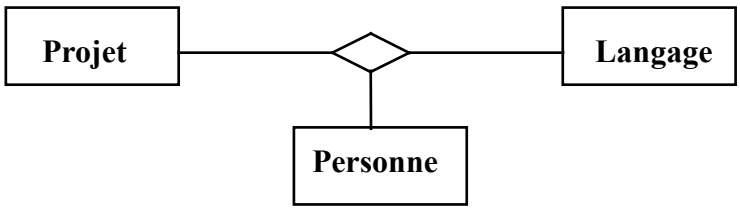


Diagramme de classes

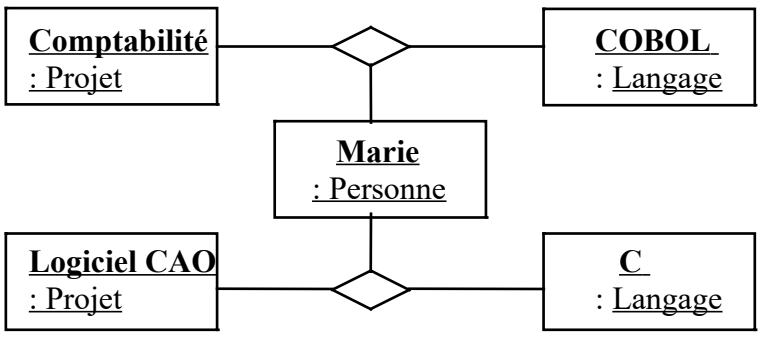
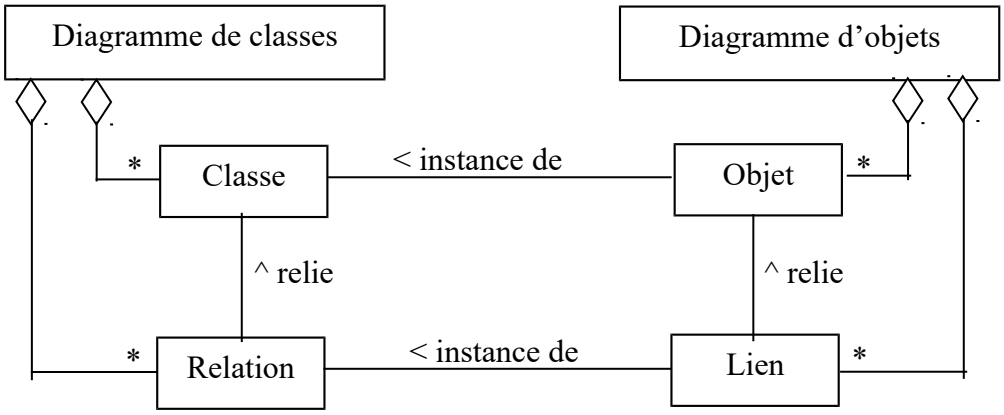


Diagramme d'objets

Extrait du méta modèle UML :



Explication orale pour ce diagramme de modélisation du diagramme de classes et d'objets.

Multiplicité

La ***multiplicité*** précise combien d'instances d'une classe peuvent se rattacher à une seule instance d'une classe donnée. La multiplicité est l'équivalent des ***cardinalités*** spécifiées sur un MCD ou modèle Entité-Association de la méthode Merise.

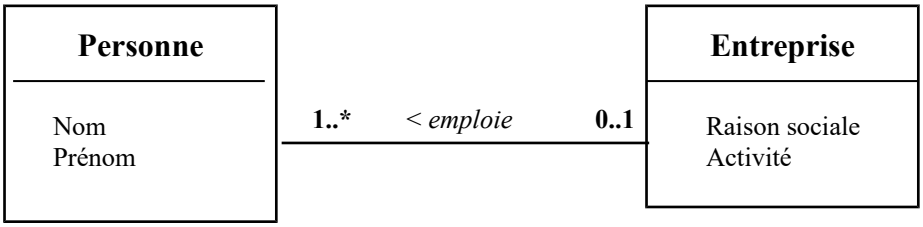
La notation générale adoptée est : **min . . max**

1	obligatoire
0..1	optionnel (aucun ou au plus 1)
0..* ou *****	quelconque (aucun ou plusieurs)
1..*	au moins 1 voire plusieurs
1..5, 10	entre 1 et 5, ou 10

Remarque : Attention !

La multiplicité est écrite au bout du trait symbolisant l'association, ce qui est l'inverse de MERISE. Cependant, avec un peu d'habitude on s'aperçoit rapidement que la lecture est plus logique et même facilité. Contrairement au MCD, ***l'association est représentée par un seul trait qui relie 2 classes***.

Une agence gère ***aucun ou plusieurs*** véhicules (* ou 0..*).
Un véhicule est attaché à ***une et une seule*** agence (1)



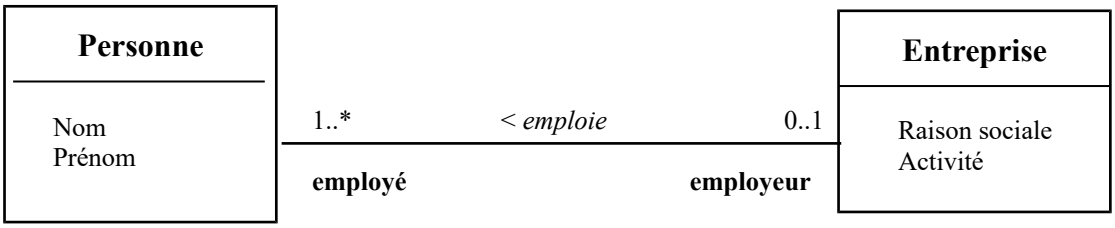
Une personne a aucun ou un (0..1) employeur (entreprise).

Une entreprise emploie de un à plusieurs (1..*) personnes.

Rôle

Un rôle est une extrémité de l'association.

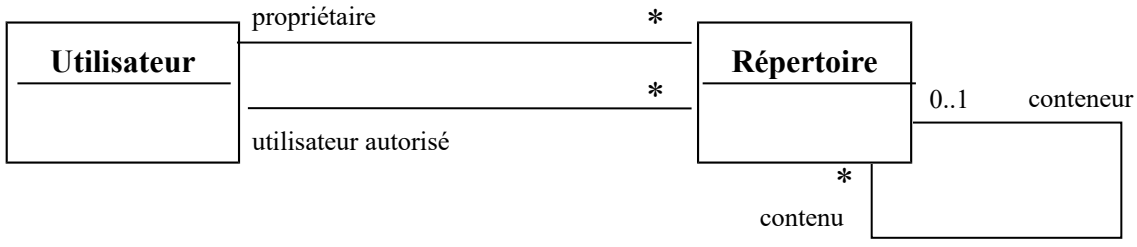
Le *nom de rôle* est un nom qui identifie de façon unique une extrémité de l'association.



L'utilisation des noms de rôle est facultative mais elle prête moins à confusion et est plus commode. Ces noms de rôle peuvent se substituer au nom de l'association ou lui être ajoutés. Le diagramme ci-dessus est le même que le précédent, seuls les noms ***employé*** et ***employeur*** sont spécifiés à chaque extrémité.

Les noms de rôles sont nécessaires pour des associations entre deux objets d'une même classe.
Les noms de rôles sont aussi utiles pour distinguer deux associations d'un même couple de classes.

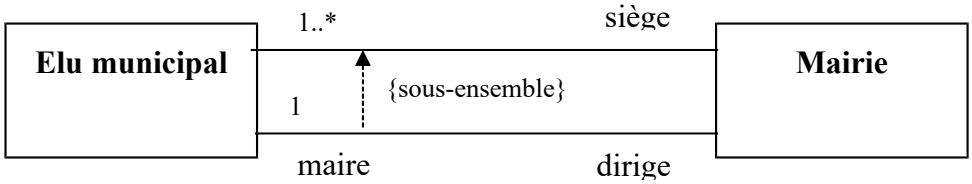
Exemple :



Contrainte

C'est une ***relation sémantique*** entre des éléments du modèle qui spécifie des conditions ou propositions devant rester vraies pour que le modèle soit valide. Une contrainte est généralement spécifiée entre accolades : { contrainte }

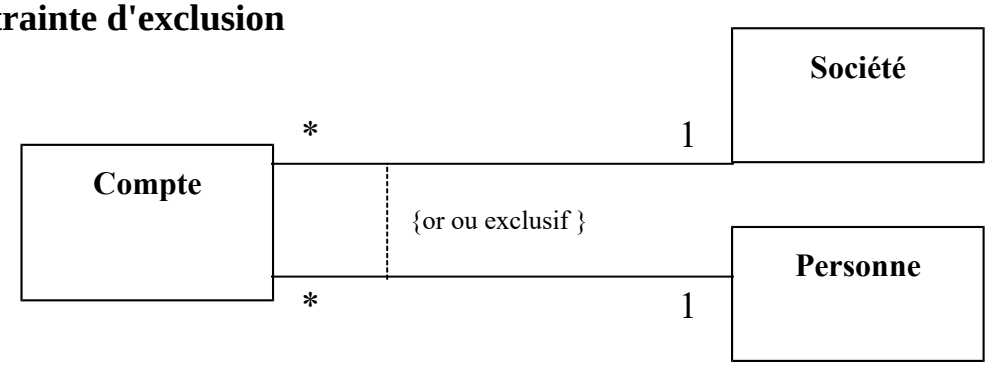
Contrainte d'inclusion :



Dans ce diagramme, le maire doit faire partie des élus qui ***siègent*** à la mairie.
L'association ***dirige*** est un sous-ensemble de l'association *siège*.

Cette contrainte signifie qu'un élu ne peut être Maire que s'il siège à la Mairie.

Contrainte d'exclusion

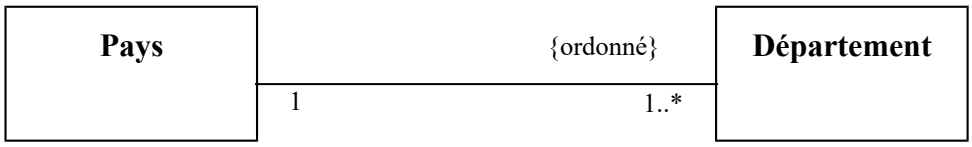


La contrainte {or} ou {ou-exclusif} précise qu'une seule association est valide.

Avec ce type de contrainte d'exclusion on exprime le fait qu'une instance de la classe Compte est ouvert par une instance de la classe Société OU une instance de la classe Personne. Nous sommes en présence d'un OU EXCLUSIF.

La contrainte {or} est exprimée graphiquement par un trait en pointillés qui relie les deux liens.

Contrainte d'ordre



Les éléments du coté "***plusieurs***" ont un ordre explicite qui doit être préservé.

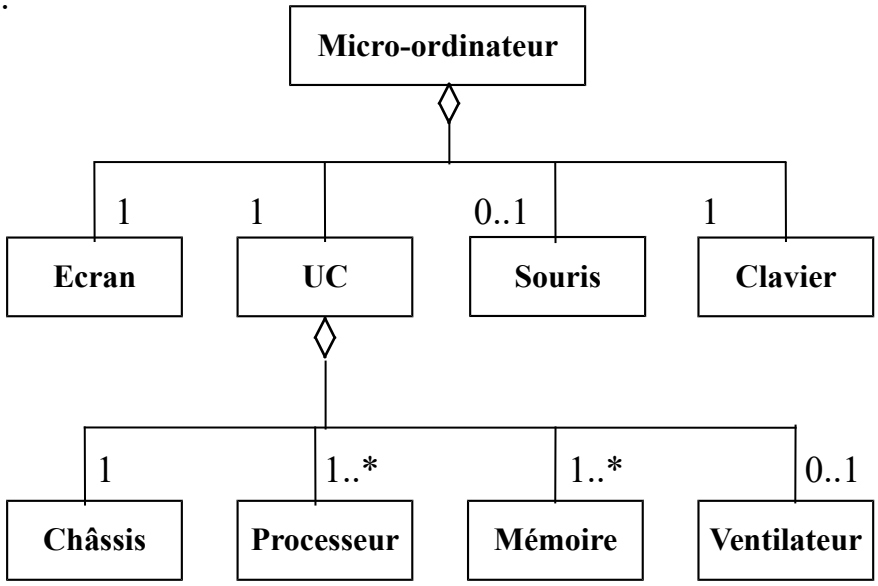
Agrégation

L'agrégation est une forme d'association forte dans laquelle un objet agrégat est *fait* de composants. Les composants font *partie de* l'agrégat. ***L'agrégation est symbolisée par un losange vide en contact avec la classe agrégat***.

L'agrégation met en relation des instances d'objets. Deux objets distincts sont englobés, l'un des deux est une partie de l'autre.

La décision d'utiliser l'agrégation est affaire de jugement et fréquemment arbitraire. Il n'est pas toujours évident de choisir entre association et agrégation.

Une agrégation peut être **fixe** si le nombre et les types des sous-parties sont prédéfinis.



Une agrégation **variable** a un nombre fini de niveaux mais le nombre de parties peut varier.

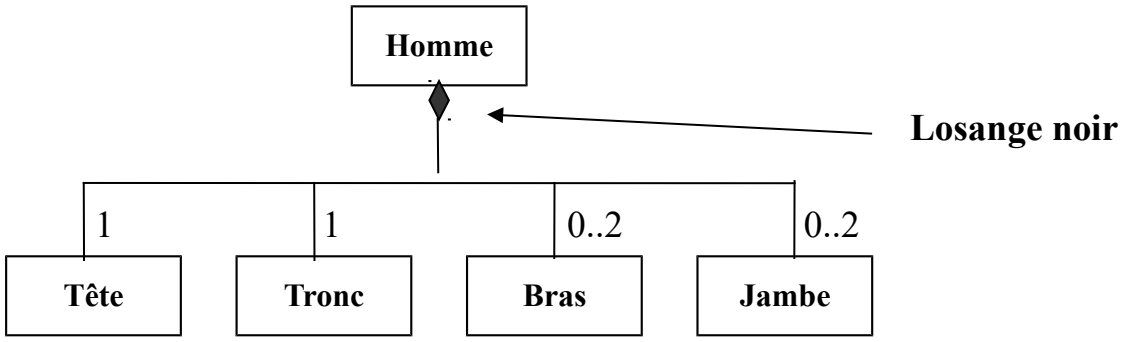
Une *Société* est une agrégation de *Divisions*, qui sont à leur tour des agrégations de *Services*. Une *Société* est donc indirectement une agrégation de *Services*.

Contient = losange vide d'agrégation

Composition

Une composition est une ***forme forte d'agrégation*** dans laquelle le cycle de vie des parties composantes est lié à celui du composé. Les composants n'existent pas seuls. Ils peuvent être créés après le composé, mais ensuite ils vivent et meurent avec lui.

Si on détruit le composé, les composants n'existent plus. Une fois établis, les liens ne peuvent pas être changés.

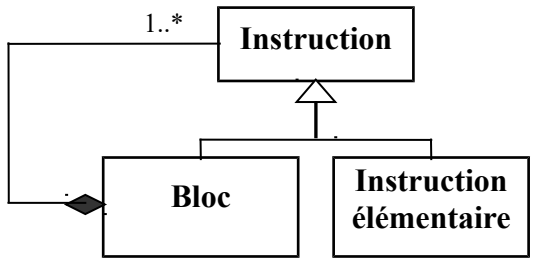


Composer = losange plein (noir) d'agrégation

Agrégat récursif

Un ***agrégat récursif*** contient directement ou indirectement une instance de la même sorte d'agrégat. Le nombre de niveaux potentiels est illimité.

Un programme informatique est un agrégat de blocs, incluant éventuellement des expressions composées récursives. La récursivité s'achève sur des expressions simples. Les blocs peuvent être imbriqués sur un nombre arbitraire de niveaux.



Ce dessin illustre la forme habituelle d'un agrégat récursif.

une ***super-classe*** et deux ***sous-classes***, l'une étant une classe terminale de l'agrégat et l'autre un assemblage d'instances de la super-classe abstraite.

La représentation graphique de la notion d'héritage est étudiée dans les prochains chapitres. Sachez que dans cette illustration, les classes **Bloc** et **Instruction élémentaire** hérite de la super-classe **Instruction**. La classe Bloc est elle-même rattachée à la super-classe Instruction par un lien Composition.

Généralisation et héritage

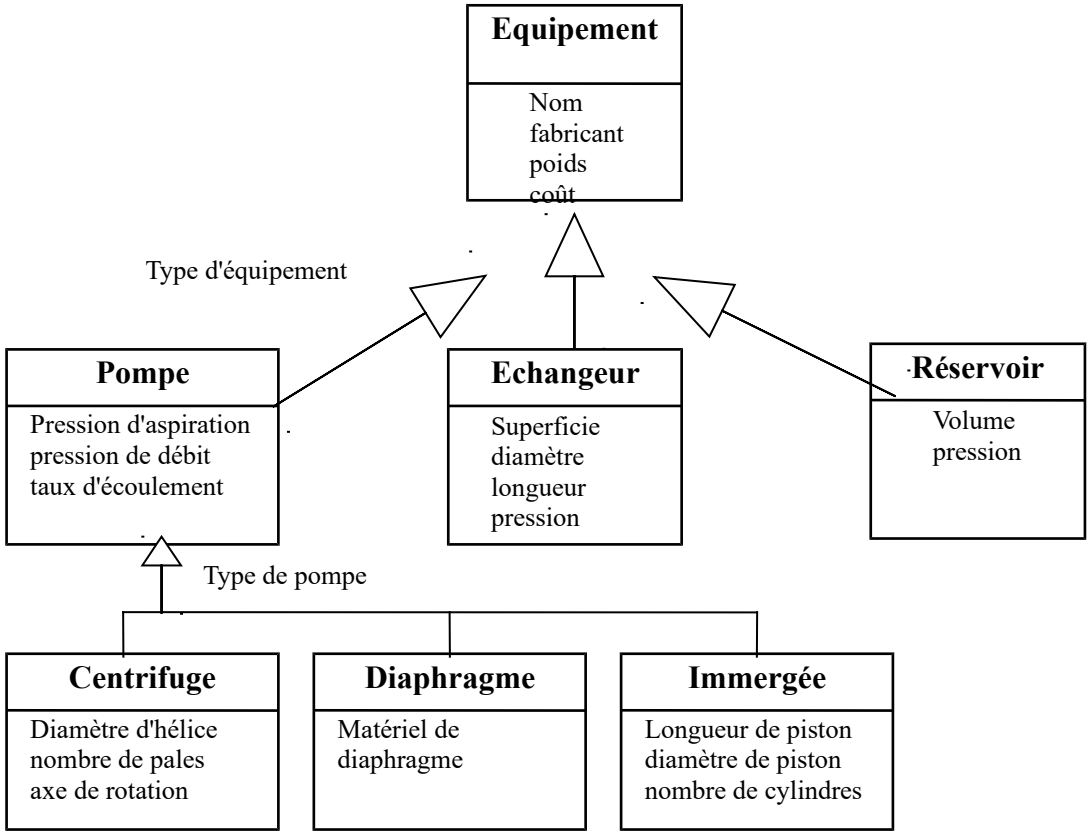
La généralisation met en relation les classes et constitue une façon de structurer la description d'un objet unique.

- La relation de classification correspond à des phrases du type :

EST UNE SORTE DE

- Une instance d'une sous-classe est simultanément une instance de toutes ses classes ancêtres.

Principe de substitution de Liskov : une instance de l'élément plus spécifique peut être utilisé là où l'élément plus général est autorisé.



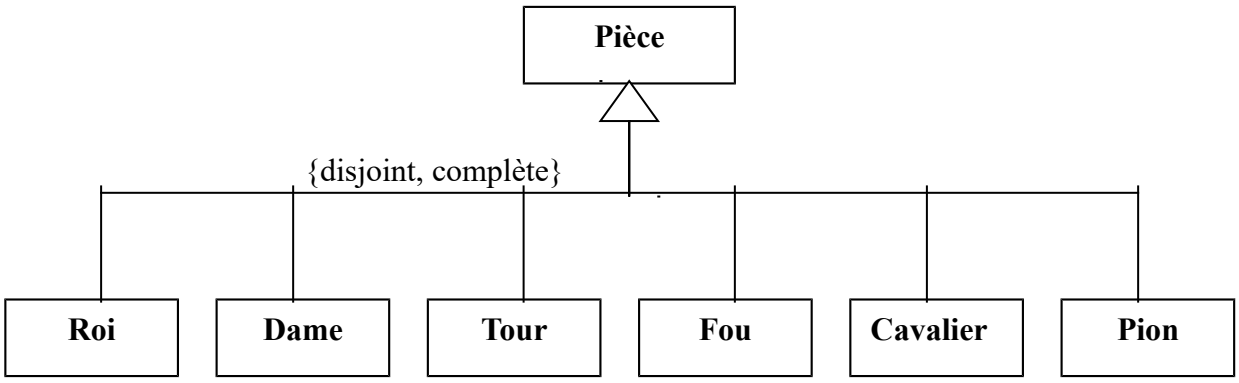
Les mots écrits près des triangles de généralisation sont des discriminants.
Un *discriminant* est un attribut de type énumération qui indique que la propriété d'un objet a été rendue abstraite par une relation de généralisation particulière.
Une seule propriété doit être discriminée à la fois.

L'**héritage** est le mécanisme par lequel des éléments plus *spécifiques* incorporent la structure et le comportement d'éléments plus *généraux*.

Contraintes sur la généralisation

Les deux seules contraintes prédéfinies en UML pour la généralisation sont :

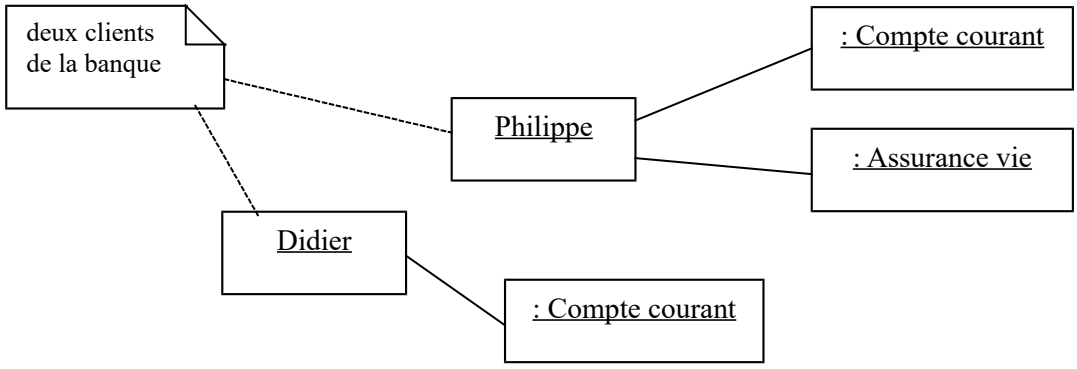
- Disjoint (disjonction)
- Complète (ou incomplète)



Ces contraintes s'expriment entre accolades {disjoint, complète}.

Note

Une note est un commentaire exprimé dans un format libre, afin de faciliter la compréhension du diagramme. Les traits discontinus permettent de relier n'importe quel élément à une note. La note se représente sous la forme d'un rectangle avec un côté haut-droit replié.



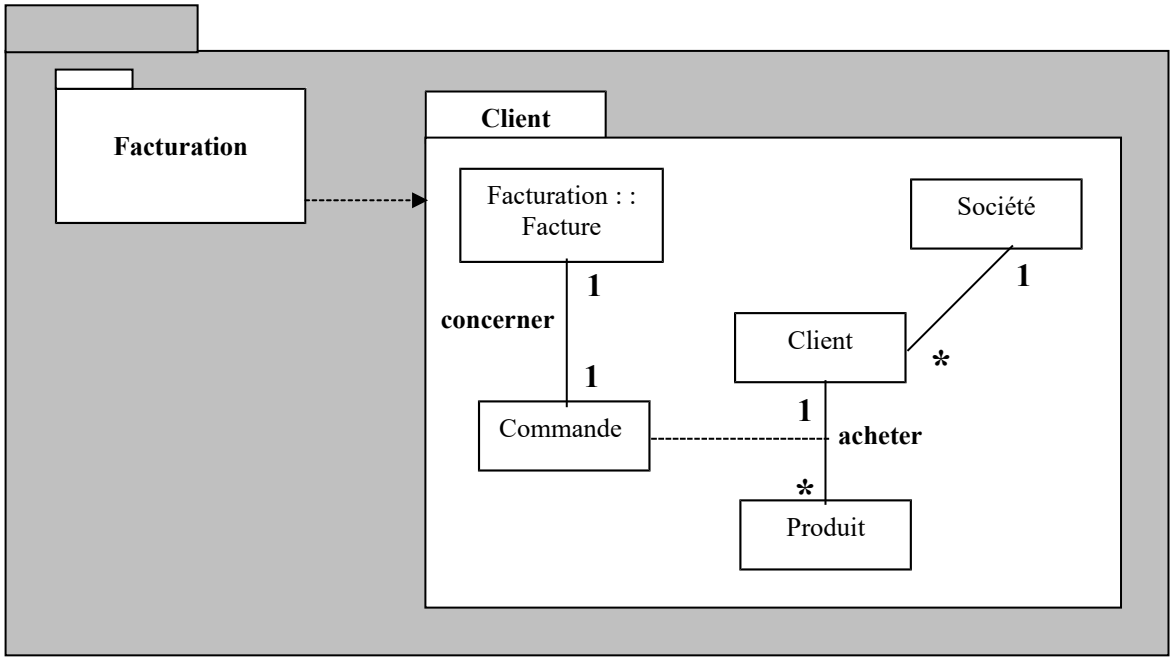
Package ou « paquetage »en French !

Un package est un ***regroupement d'éléments du modèle***. Il peut comporter plusieurs types d'entités (classes, associations, autres paquetages). Un élément du modèle, et plus particulièrement une classe peut être représentée plusieurs fois dans un modèle. Il est alors bon d'indiquer le paquetage d'appartenance d'une classe comme suit :

NomPackage : : NomClasse

Le nom d'une classe est unique dans un paquetage. Par contre il est possible d'avoir deux classes de même nom dans des paquetages différents.

Le système entier peut être vu comme un paquetage unique de haut niveau contenant tout le reste. On peut aussi représenter les dépendances entre paquetages.



Stéréotype

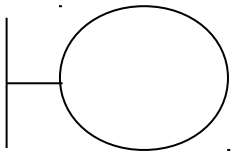
Nouveau type d'élément de modélisation qui ***étend la sémantique du méta-modèle*** !

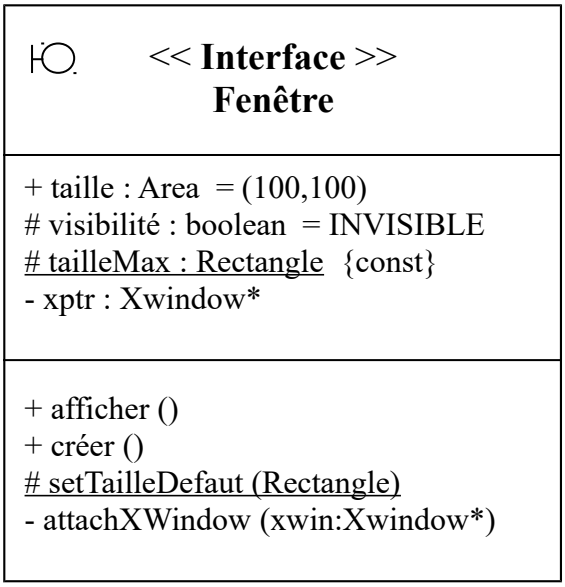
Un stéréotype permet de classer les éléments du modèle en grandes familles. Il peut être associé à tout élément du modèle (classe, association, opération, attribut, package …).

On peut définir ses propres stéréotypes, par exemple les classes **d'interface**, les classes de **contrôle**, les classes **entité**.

On peut même y associer une icône.

<< Interface>>





Pattern (modèle)

Modèle générique résolvant un problème classique et récurrent :

- Solution qui a fait ses preuves, fondée sur l'expérience.
- Solution réutilisable dans des contextes différents.

De plus en plus de Patterns sont disponibles au travers de livres et publications :

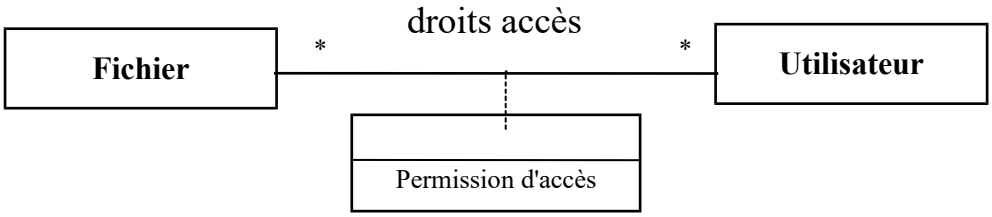
- Design Patterns (conception)
- Analysis Patterns (analyse)

L'agrégat récursif qui modélise les structures arborescentes, est un exemple d'Analysis Pattern.

Attribut d'association

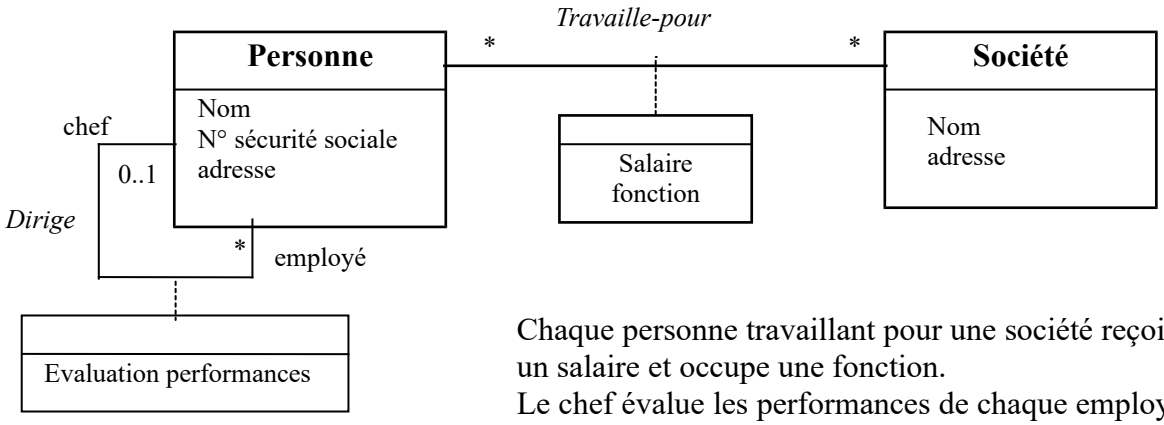
Un attribut d'association est une propriété des liens d'une association.

Les associations plusieurs à plusieurs sont la justification la plus contraignante de l'existence des attributs d'association. Un tel attribut est sans conteste une propriété du lien et ne peut être rattaché à l'un ou l'autre des objets.



/user/cdrgest	(lire)	Pierre Arditi
/user/cdrgest	(lire-écrire)	Christine Angot
/com/grmod/.login	(lire-écrire)	Vincent Delerme

Autre exemple :



Chaque personne travaillant pour une société reçoit un salaire et occupe une fonction.
Le chef évalue les performances de chaque employé.

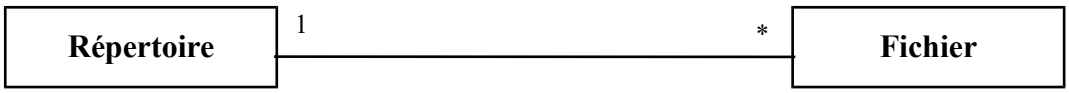
Il est possible de ranger les attributs des associations un à un et un à plusieurs dans la classe opposée au coté "un". Cela n'est pas conseillé, car la souplesse d'évolution serait réduite si les multiplicités de l'association devaient changer.

Qualification

Le qualificatif est un **attribut spécial qui permet de filtrer les éléments de la cible**. Cela réduit souvent la multiplicité effective d'une association.

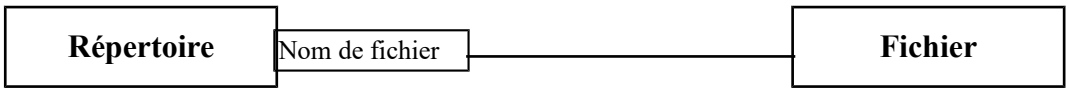
Les associations un à plusieurs et plusieurs à plusieurs peuvent être qualifiées.

Exemple :

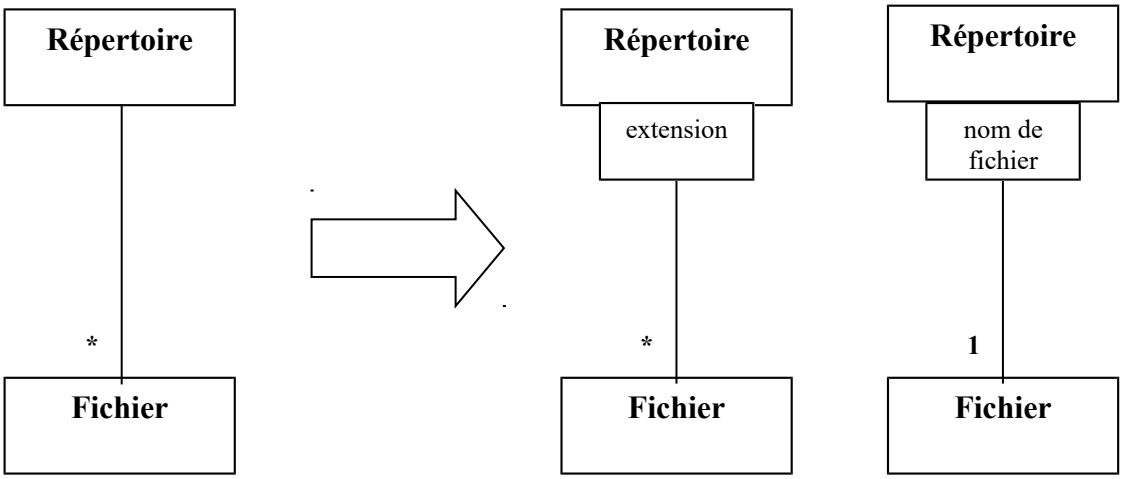


Un répertoire contient plusieurs fichiers.
Un fichier ne peut appartenir qu'à un seul répertoire.

Dans le contexte d'un répertoire, le nom de fichier (attribut de fichier) spécifie un fichier unique dans la classe fichier.
Qualification = Nom de fichier

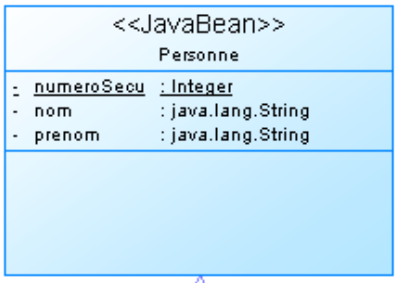


L'utilisation d'un qualificatif ne réduit pas toujours la multiplicité à 1.



Modélisation et code java

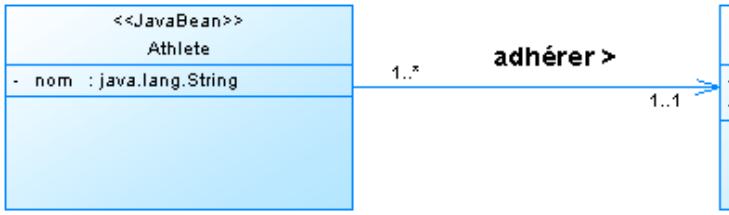
Héritage / Généralisation




```
public class Personne
{
  private Integer numeroSecu;
  private String nom;
  private String prenom;
}
```

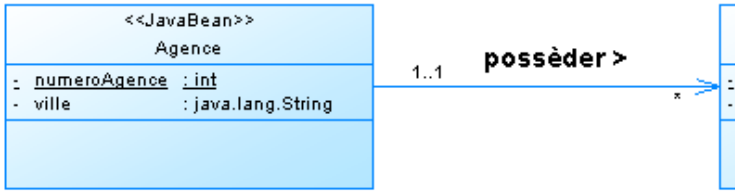

```
public class Stagiaire extends Personne
{
  private Integer anneePromotion;
}
```

Associations



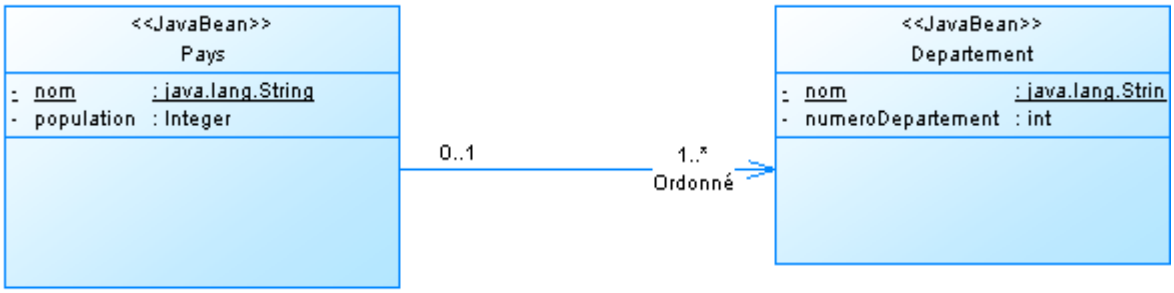

```
public class Athlete
{
   private java.lang.String nom;
   public Club club;
}
```

```
public class Club
{
    private java.lang.String nom;
    private java.lang.String ville;
}
```

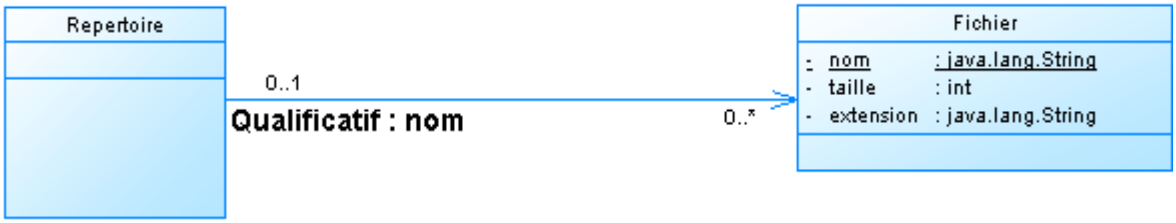



```
public class Agence {
    private int numeroAgence;
    private java.lang.String ville;
    public java.util.Collection<Vehicule> vehicule;
}
```

Associations (suite)




```
public class Pays
 {
   private java.lang.String nom;
   private Integer population;
   public java.util.Collection<Departement> departement;

  /**
   * Exemple de méthode getter
   * Les objets Departements sont mis dans une HashSet
   */
  public java.util.Collection<Departement>
getDepartement() {
     if (departement == null)
       departement = new java.util.HashSet<Departement>();
     return departement;
   }
}
```

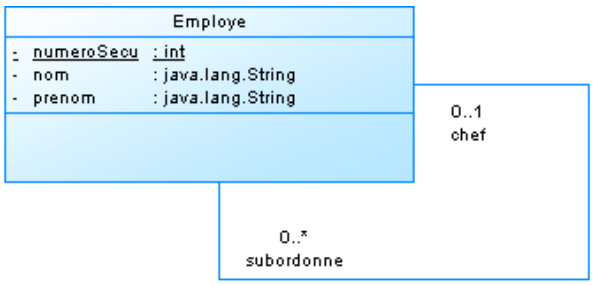



```
On suppose ici, l'utilisation d'un qualificatif "nom de fichier" pour
l'enregistrement d'un fichier dans le tableau (associatif) :
public class Repertoire
{
  private Map<String,Fichier> fichiers = new HashMap<String, Fichier>();
}
```

Association bidirectionnelle -0,1 - 0,1

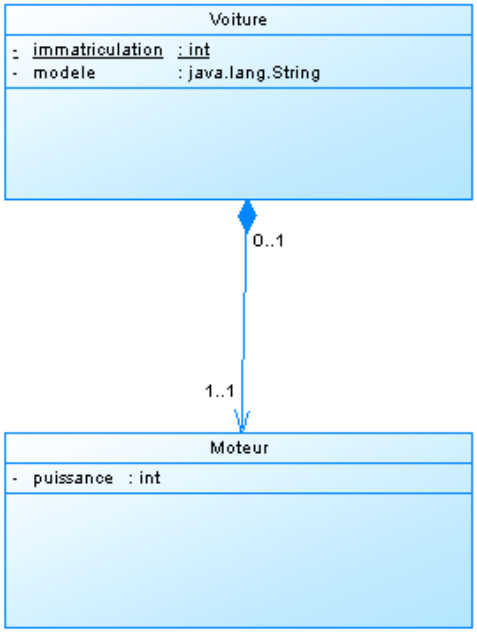

```
public class Coq {
  private java.lang.String nom;
  private Poule femelle;

 public Poule getFemelle()
 {
     return femelle;
 }

 public void setFemelle(Poule newPoule)
 {
     if (this.femelle == null ||
!this.femelle.equals(newPoule))
     {
       Poule oldPoule = this.femelle;
       this.femelle = newPoule;
       if (oldPoule != null)
         oldPoule.setMâle(null);
       // Re-assign the value
       this.femelle = newPoule;
       if (this.femelle != null)
         this.femelle.setMâle(this);
     }
  }

public boolean equals(Object other)
{

     if (other == null)
       return false;

     if (other == this)
       return true;

     if (!(other instanceof Coq))
       return false;

     return true;
   }

public int hashCode()
{
     return 0;
}

public String toString() {
     return "Coq: ";
   }
}
```

```
public class Poule {
  private java.lang.String nom;
   private Coq mâle;

  public Coq getMâle() {
     return mâle;
   }

 public void setMâle(Coq newCoq) {
     if (this.mâle == null ||
!this.mâle.equals(newCoq))
     {
       Coq oldCoq = this.mâle;
       this.mâle = newCoq;
       if (oldCoq != null)
         oldCoq.setFemelle(null);
       // Re-assign the value
       this.mâle = newCoq;
       if (this.mâle != null)
         this.mâle.setFemelle(this);
     }
   }

public boolean equals(Object other)
{

     if (other == null)
       return false;

     if (other == this)
       return true;

     if (!(other instanceof Poule))
       return false;

     return true;
   }

public int hashCode()
{
     return 0;
}

public String toString() {
     return "Poule: ";
   }
}
```




```
public class Employe
{
    private int numeroSecu;
    private String nom;
    private String prenom;
    private Collection<Employe> subordonne;
    private Employe chef;
...
}
```

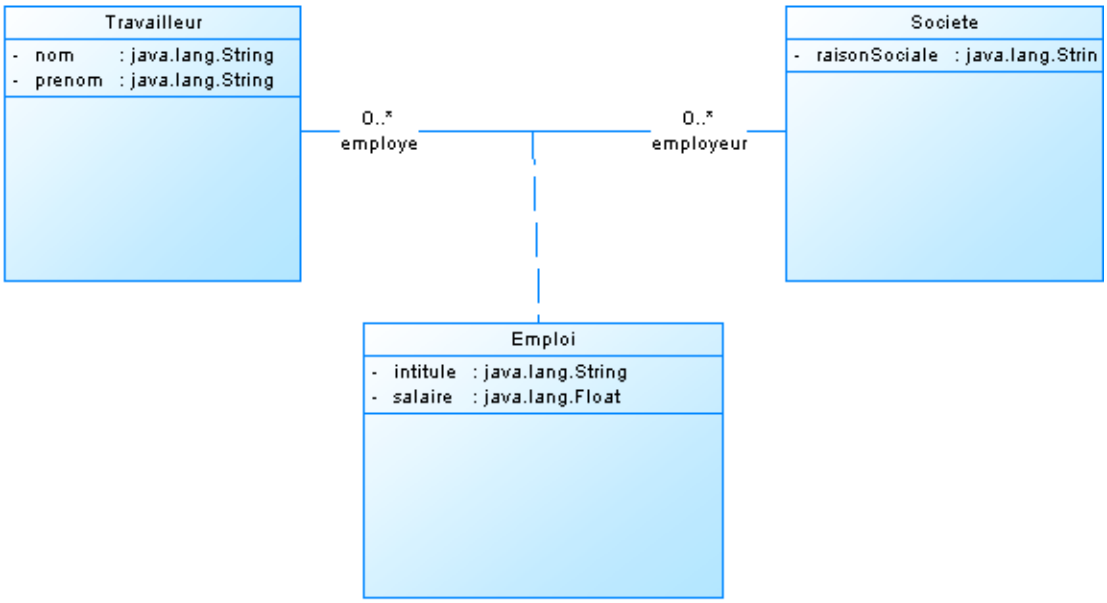
Association de type Composition




```
public class Voiture
{
   private int immatriculation;
   private java.lang.String modele;
   private Moteur moteur;
...
}
```

```
public class Voiture
{
   private int immatriculation;
   private java.lang.String modele;
   private Moteur moteur;

// ici on peut avoir une classe Moteur
imbriquée pour
// renforcer la notion de composition
   private static class Moteur {
      private int puissance;
   ...
 }
...
}
```

Association : Classe d'association




```
public class Emploi {
   private java.lang.String intitule;
   private java.lang.Float salaire;
   private Travailleur employe;
   private Societe employeur;
   ...
}
```