

# MODÉLISATION ET CONCEPTION ORIENTÉES OBJET AVEC UML

Ilhem Boussaïd  
[ilhem\\_boussaid@yahoo.fr](mailto:ilhem_boussaid@yahoo.fr)

Université des Sciences et de la Technologie Houari Boumediene  
Licence 3 Académique  
<http://sites.google.com/site/ilhemboussaid>

1<sup>er</sup> décembre 2010

# PLAN

## 1 MODÉLISATION

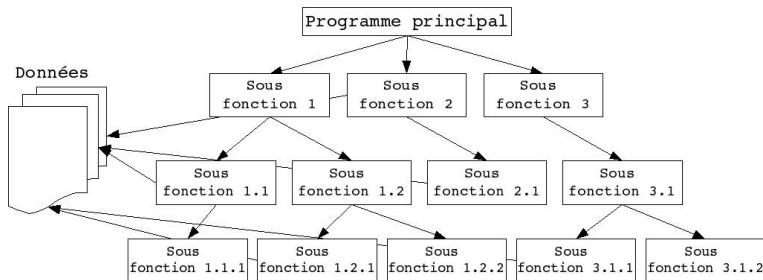
- Démarche fonctionnelle
- Démarche orientée objet

## 2 DIAGRAMME DE CLASSES

- Classes et instances
- Relations entre classes
- Compléments sur la modélisation des classes

# MODÉLISATION PAR DÉCOMPOSITION FONCTIONNELLE DESCENDANTE

- La forme la plus immédiate pour décrire un travail à effectuer est de lister les **actions à réaliser**.
- On découpe une tâche complexe à effectuer en une **hiérarchie d'actions** à réaliser de plus en plus simples, petites et précises (Pour décrire, on utilise le **verbe**) :

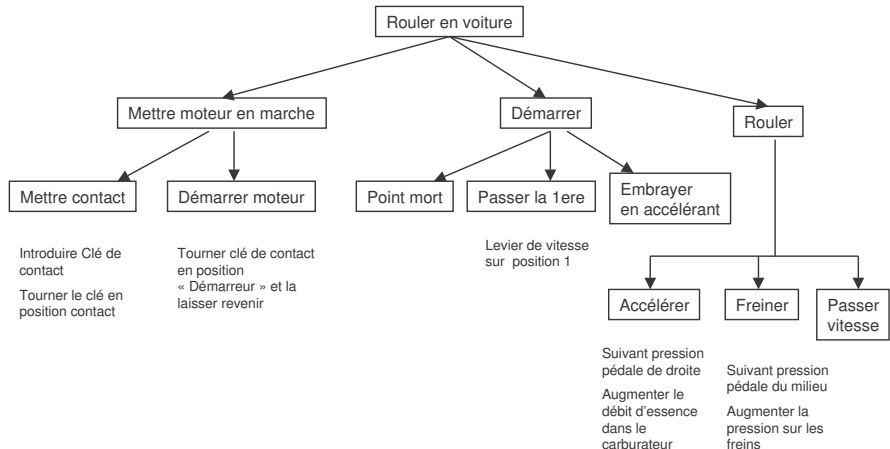


# MODÉLISATION PAR DÉCOMPOSITION FONCTIONNELLE DESCENDANTE

Exemple :

- ① **Rouler En Voiture :**
  - ① **Mettre** moteur en marche :
    - ① **Mettre** Contact
    - ② **Démarrer** Moteur
  - ② **Démarrer** Voiture :
    - ① **Mettre** Point Mort
    - ② **Passer** La Première
    - ③ **Embrayer** En Accélérant
  - ③ **Rouler :**
    - ① **Accélérer**
    - ② **Freiner**
    - ③ **Passer** Vitesse

# MODÉLISATION PAR DÉCOMPOSITION FONCTIONNELLE DESCENDANTE



# MODÉLISATION PAR DÉCOMPOSITION FONCTIONNELLE DESCENDANTE

L'implémentation en code source d'une solution décrite en terme d'actions est un **code organisé en fonctions (programmation procédurale)** :

```
roulerEnVoiture()  
    mettreMoteurEnMarche()  
        mettreContact()  
        demarrerMoteur()  
    demarrer()  
        mettrePointMort()  
        passerLaPremiere()  
        embrayerEnAccelerant()  
rouler()  
    accelerer()  
    freiner()  
    passerVitesse()
```

# MODÉLISATION PAR DÉCOMPOSITION FONCTIONNELLE DESCENDANTE

Dans ce cadre de travail :

- L'**analyse** est une découpe fonctionnelle descendante des fonctionnalités à pourvoir.
- La **conception** est une découpe du logiciel en une hiérarchie descendante d'actions permettant de satisfaire les fonctionnalités à pourvoir.
- L'**implémentation** est une programmation procédurale.

# MODÉLISATION PAR DÉCOMPOSITION FONCTIONNELLE DESCENDANTE

Dans une programmation **procédurale** issue d'une découpe fonctionnelle descendante, **les données et les fonctions travaillant sur ces données sont dispersées dans des modules différents.**

```
type typeVitesse est {pointMort, premiere, seconde, troisieme, quatrieme  
    cinquieme, marcheArriere};
```

```
vitesseCourante : typeVitesse := pointMort;
```

```
procedure passerVitesse(  
    vitesseCourante : in out typeVitesse;  
    vitesseAPasser : in typeVitesse);
```

```
procedure mettreAuPointMort(  
    vitesseCourante : in out typeVitesse);
```



# MODÉLISATION PAR DÉCOMPOSITION FONCTIONNELLE DESCENDANTE

## Dispersion données/fonctions :

- Lorsque le logiciel évolue, il faut faire évoluer les structures de données et les fonctions en parallèle (probablement dans des modules différents).
- Maintenir cette cohérence est laborieuse parce que données et fonctions sont dispersées.

**La dispersion données/fonctions nuit à l'extensibilité !**

# MODÉLISATION PAR DÉCOMPOSITION FONCTIONNELLE DESCENDANTE

## RAPPELEZ VOUS !

Pour qu'un logiciel soit extensible et réutilisable, il faut qu'il soit découpé en modules

- **faiblement couplés**, ainsi chaque entité peut être modifiée en limitant l'impact du changement au reste de l'application. et
- **à forte cohésion**. Il faut réunir ce qui est impacté par une même modification (« qui se ressemble s'assemble »)

# MODÉLISATION PAR DÉCOMPOSITION FONCTIONNELLE DESCENDANTE

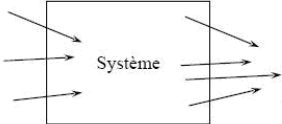
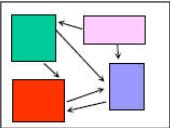
Comment évaluer la découpe fonctionnelle descendante et la programmation procédurale en termes de couplage et cohésion ?

Séparer données et fonctions sur ces données entraîne :

- Un **fort couplage** par les données,
- **Perte de cohésion** (par dispersion).

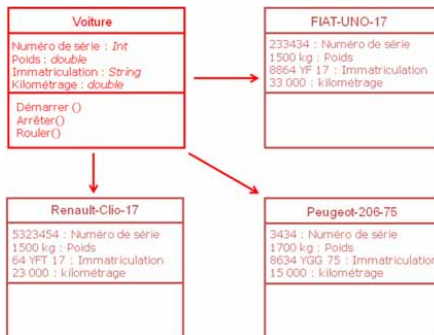
**Le code est donc peu extensible et peu réutilisable !**

# FONCTIONNEL VERSUS OBJET

LA DEMARCHE FONCTIONNELLE	LA DEMARCHE OBJET
<p data-bbox="65 384 591 439">Accent mis sur ce que fait le système (ses fonctions)</p>  <p data-bbox="203 650 454 677">«Que fait le système ?»</p>	<p data-bbox="696 384 1300 439">Accent mis sur ce qu'est le système (ses composants) : <b>les objets</b></p>  <p data-bbox="801 640 1195 667">«De quoi se compose le système ?»</p>

# DÉMARCHE ORIENTÉE OBJET

- On ne raisonne plus en termes d'actions mais plutôt en **concepts du monde physique**.
  - Puisqu'ils appartiennent au monde physique, ces concepts peuvent être stables et réutilisables.
- On ne raisonne uniquement en verbes, mais davantage en **noms**.



# PLAN

## 1 MODÉLISATION

- Démarche fonctionnelle
- Démarche orientée objet

## 2 DIAGRAMME DE CLASSES

- Classes et instances
- Relations entre classes
- Compléments sur la modélisation des classes

# OBJETS

## OBJET

### Identité + Etat + Comportement

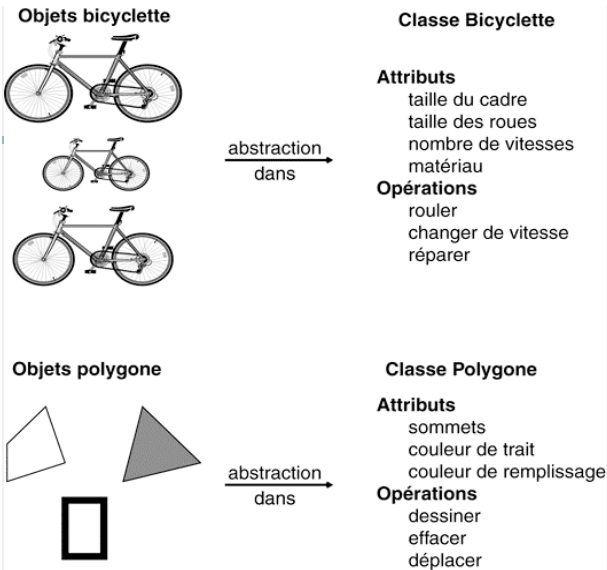
- Une **identité**
  - deux objets différents ont des identités différentes
  - on peut désigner l'objet (y faire référence)
- Un **état** (attributs)
  - ensemble de propriétés/caractéristiques définies par des valeurs
  - permet de le personnaliser/distinguer des autres objets
  - peut évoluer dans le temps
- Un **comportement** (méthodes)
  - ensemble des traitements que peut accomplir un objet (ou que l'on peut lui faire accomplir)

# OBJETS - EXEMPLE

Objet	États	Comportements
Chien	Nom, race, âge, couleur	Aboier, chercher le baton, mordre, faire le beau
Compte	N°, type, solde, ligne de crédit	Retrait, virement, dépôt, consultation du solde
Téléphone	N°, marque, sonnerie, répertoire, opérateur	Appeler, Prendre un appel, Envoyer SMS, Charger
Voiture	Plaque, marque, couleur, vitesse	Tourner, accélérer, s'arrêter, faire le plein, klaxonner



# CLASSES ET INSTANCES



# CLASSES ET INSTANCES

- Les **objets** possédant la **même structure de données** (attributs) et le **même comportement** (opérations) sont les représentants d'une même **classe**.
- Une classe est une **abstraction** qui décrit les propriétés pertinentes pour une application.
- Données et opérations traitant les données ne sont pas séparées, mais réunies au sein d'un même module. **Cohésion !**
- Chaque **objet** est une **instance** d'une classe.

# CLASSES ET INSTANCES

- La classe « chien » définit :
  - Les attributs d'un chien (nom, race, couleur, âge, ...)
  - Les comportements d'un chien (Aboier, chercher le bâton, mordre...)
- Il peut exister dans le monde plusieurs objets (ou instances) de chien

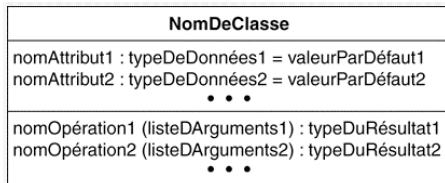
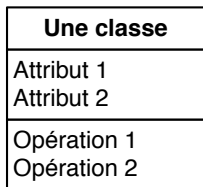
Classe	Instances (Objets)
Chien	Mon chien: Bill, Teckel, Brun, 1 an Le chien de mon voisin: Hector, Labrador, Noir, 3 ans
Compte	Mon compte à vue: N° 210-1234567-89, Courant, 1.734 DZ, 12.500DZ Mon compte épargne: N° 083-9876543-21, Epargne, 27.000 DZ, 0DZ
Voiture	Ma voiture: ABC-123, VW Polo, grise, 0 km/h La voiture que je viens de croiser: ZYX-987, Porsche, noire, 170 km/h

# OBJECTIF

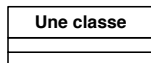
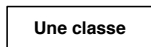
- Les diagrammes de cas d'utilisation modélisent à **QUOI** sert le système.
- Le système est composé d'objets qui interagissent entre eux et avec les acteurs pour réaliser ces cas d'utilisation.
- Les **diagrammes de classes** permettent de spécifier la **structure** statique d'un système, en termes de classes et de relations entre ces classes.

# REPRÉSENTATION D'UNE CLASSE

- rectangle à 3 compartiments
  - nom (singulier, majuscule)
  - attributs
  - opérations



- plus ou moins de détails en fonction des besoins :



# ATTRIBUTS ET OPÉRATIONS

Représentation d'un **attribut** :

*visibilité nom : type [multiplicité] = valeur\_initiale*

↑  
public +  
privé -  
protégé #

↑  
facultatif  
mais impératif  
pour l'implémentation

↑  
facultatif  
par défaut: 1

[0..1] élément pouvant être nul  
[4] tableau de 4 éléments  
[2..\*] tableau dynamique  
d'au moins 2 éléments

↑  
facultatif

# ATTRIBUTS ET OPÉRATIONS

Représentation d'une **opération** :

- Une *opération* représente un élément de **comportement** (un **service**) contenu dans une classe.
- Nous ajouterons surtout les opérations en conception objet, car cela fait partie des choix d'**attribution des responsabilités aux objets**.

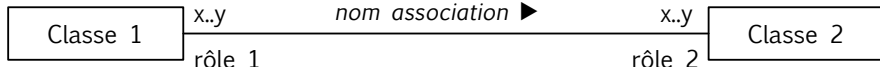
Personne
nom dateDeNaissance
changerDEmploi changerDAdresse

Fichier
nomDeFichier tailleEnOctets dernièreMaj
imprimer

ObjetGéométrique
couleur position
déplacer (delta : Vector) sélectionner (p : Point): Boolean pivoter (in angle : float = 0.0)

# LES ASSOCIATIONS

- Connexion sémantique bidirectionnelle entre classes
- **Représentation des associations :**
  - **Nom** : forme verbale, avec un sens de lecture
  - **Rôles** : forme nominale, décrit une extrémité de l'association
  - **Multiplicité** : 1, 0..1, 0..\*, 1..\*,  $n..m$

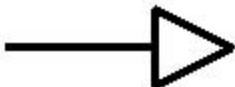




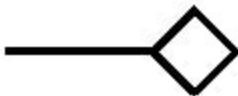
# LES TYPES D'ASSOCIATIONS



**Simple**



**Généralisation**



**Agrégation**

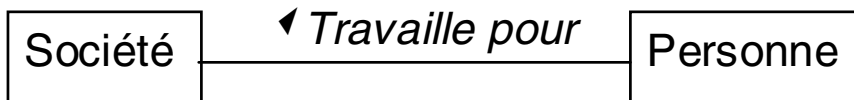


**Composition**

# NOMMAGE DES ASSOCIATIONS

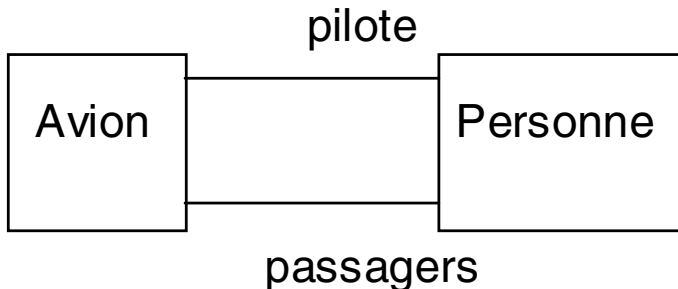
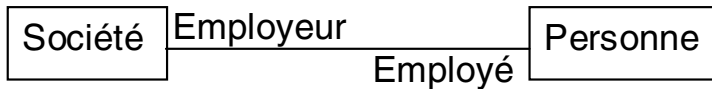


On peut ajouter un sens de lecture :



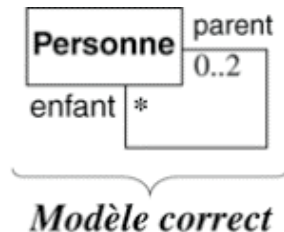
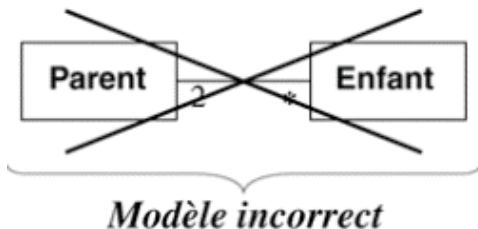
# NOMS D'EXTRÉMITÉ D'ASSOCIATION - RÔLE

- Chaque extrémité d'association peut être nommée.

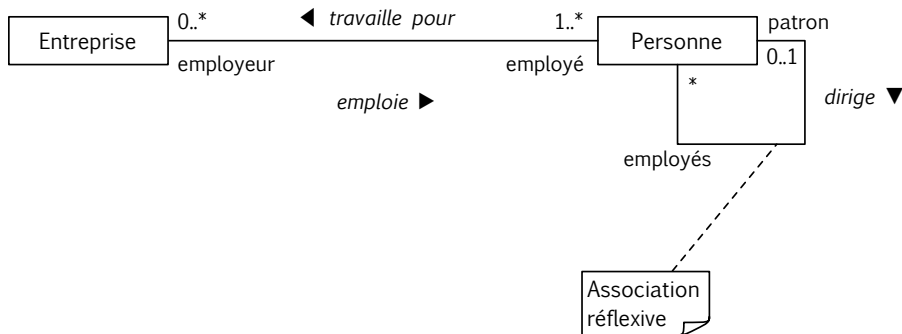


# NOMS D'EXTRÉMITÉ D'ASSOCIATION - RÔLE

- Nommez les extrémités pour modéliser plusieurs références à la même classe.

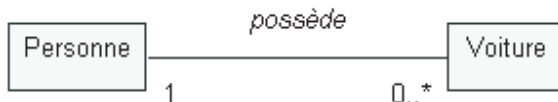


# EXEMPLE RÉCAPITULATIF



# MULTIPlicité DES ASSOCIATIONS

- La multiplicité spécifie le nombre d'instances d'une classe pouvant être liées à une seule instance d'une classe associée. **Elle contraint le nombre d'objets liés.**
- **Exemple** : une personne peut posséder plusieurs voitures (entre zéro et un nombre quelconque) ; une voiture est possédée par une seule personne.

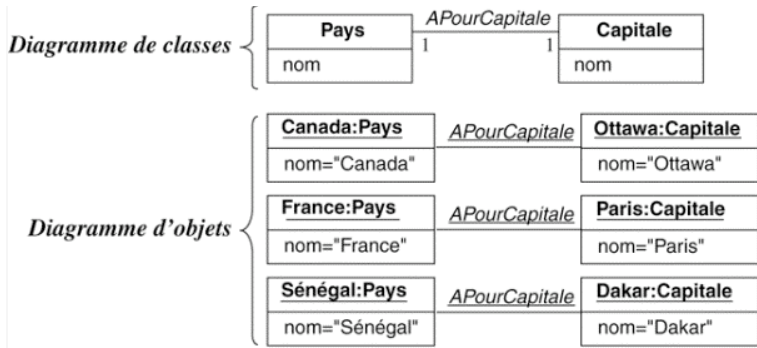


# MULTIPLICITÉ DES ASSOCIATIONS

<b>1</b>	Un et un seul
<b>0..1</b>	Zéro ou un
<b>N</b>	N (entier naturel)
<b>M .. N</b>	De M à N (entiers naturels)
<b>*</b>	De zéro à plusieurs
<b>0 .. *</b>	De zéro à plusieurs
<b>1 .. *</b>	D'un à plusieurs

# MULTIPLICITÉ DES ASSOCIATIONS

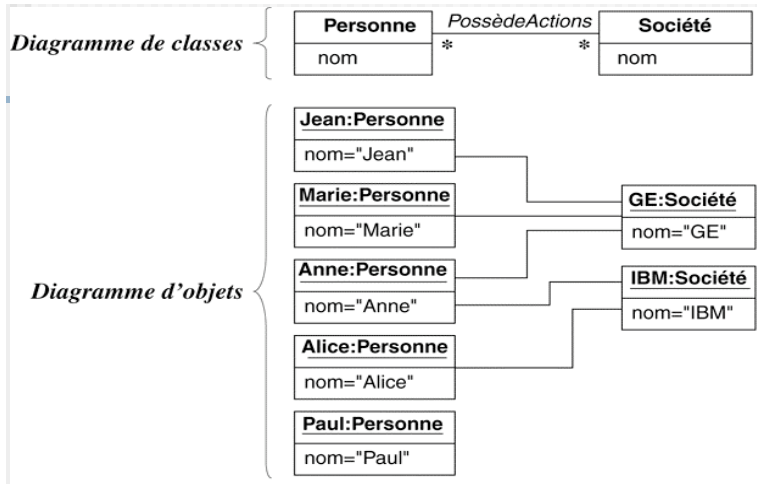
## Exemple :





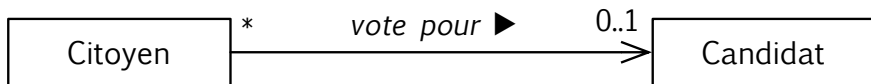
# MULTIPLICITÉ DES ASSOCIATIONS

- Exemple :

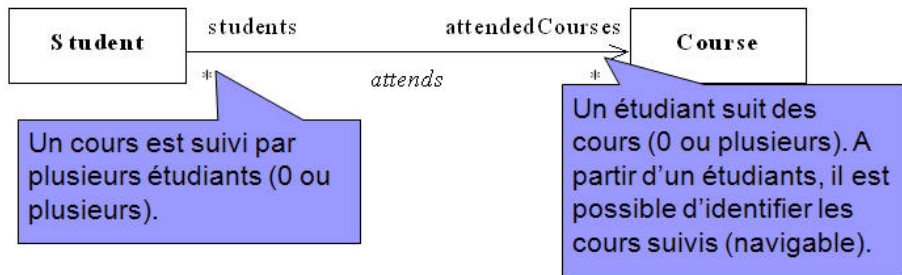


# NAVIGABILITÉ D'UNE ASSOCIATION

- Par défaut, les associations sont navigables dans les deux directions.
- la navigation peut être restreinte à une seule direction : **les instances d'une classe ne "connaissent" pas les instances d'une autre.**
- On restreint la navigabilité d'une association à un seul sens à l'aide d'une flèche.



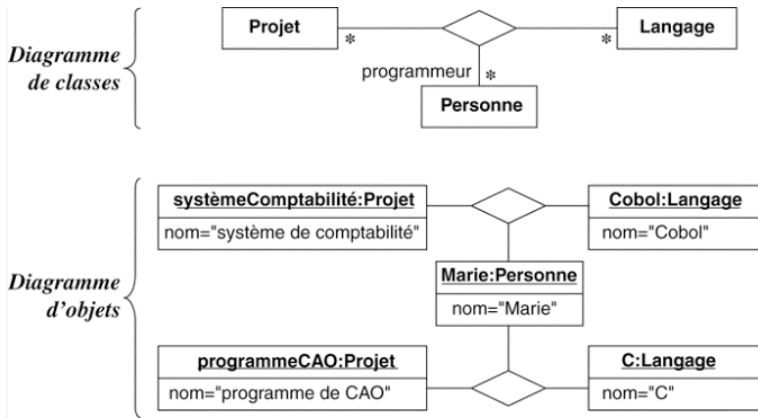
# NAVIGABILITÉ D'UNE ASSOCIATION



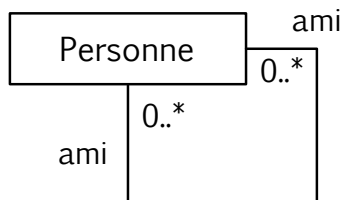
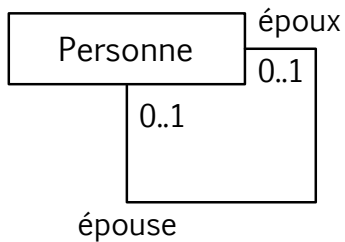
# ASSOCIATION N-AIRE

- En général, les associations sont binaires
- **N-aires** : au moins trois instances impliquées

**A n'utiliser que lorsqu'aucune autre solution n'est possible !**

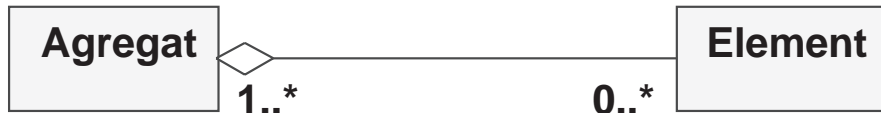


# LES ASSOCIATIONS REFLEXIVES

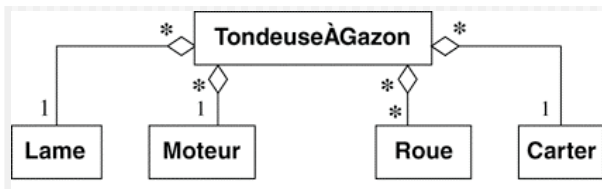


# AGRÉGATION

- Une **agrégation** est un cas particulier d'association **non symétrique** exprimant une relation de **contenance** d'un élément dans un ensemble.
- Les agrégations n'ont pas besoin d'être nommées : implicitement elles signifient « contient », « est composé de ».
- On représente l'agrégation par l'ajout d'un losange vide du côté de l'agregat (l'ensemble).



# AGRÉGATION - EXEMPLES



# COMPOSITION

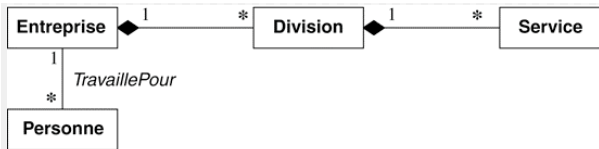
- Une **composition** est une agrégation plus forte impliquant que :
  - un élément ne peut appartenir qu'à un seul agrégat composite (agrégation non partagée);
  - la destruction de l'agrégat composite (l'ensemble) entraîne la destruction de tous ses éléments (les parties)
  - le composite est responsable du cycle de vie des parties.





# COMPOSITION - EXEMPLES

- Une partie constituante ne peut appartenir à plus d'un assemblage ;
- une fois une partie constituante affectée à un assemblage, sa durée de vie coïncide avec ce dernier.



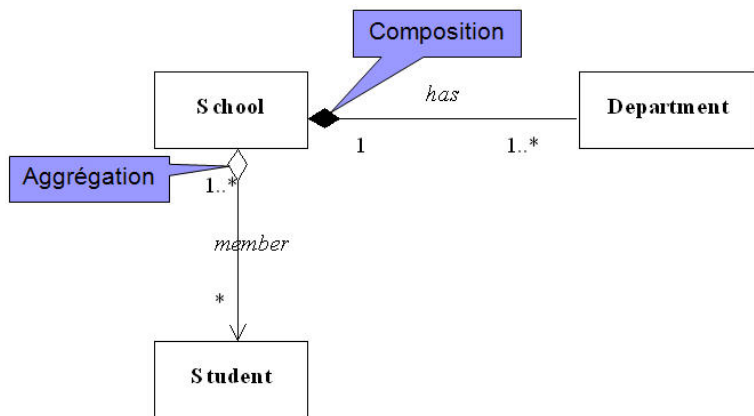
# AGRÉGATION VS. COMPOSITION

QUAND METTRE UNE **COMPOSITION** PLUTÔT QU'UNE AGRÉGATION ?

- Pour décider de mettre une composition plutôt qu'une agrégation, on doit se poser les questions suivantes :
  - Est-ce que la destruction de l'objet **composite (du tout)** implique nécessairement la destruction des objets composants (les parties) ? C'est le cas si les composants n'ont pas d'autonomie vis-à-vis des composites.
  - Lorsque l'on copie le composite, doit-on aussi copier les composants, ou est-ce qu'on peut les «réutiliser», auquel cas un composant peut faire partie de plusieurs composites ?

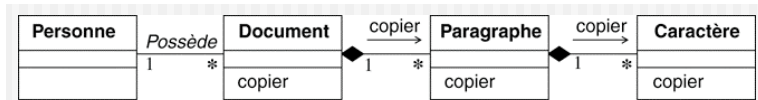
Si on répond par l'affirmative à ces deux questions, on doit utiliser une **composition**.

# AGRÉGATION VS. COMPOSITION



# PROPAGATION (OU DÉCLENCHEMENT)

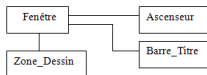
- Application automatique d'une opération à un réseau d'objets à partir d'un objet initial quelconque.
- l'agrégation permet un mécanisme de **délégation d'opérations** : l'opération *Document.copier()* peut être déléguée à l'opération *Paragraphe.copier()* en l'appliquant à toutes les instances de paragraphe qui composent le document. Celle-ci peut à son tour être déléguée dans les mêmes conditions à *Caractère.copier()*.



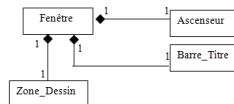
# AGRÉGATION VS. COMPOSITION



1) Composition :



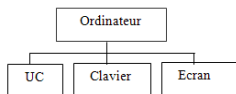
2) Composition :



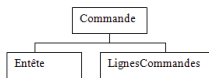
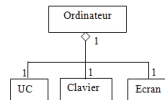
3) Agrégation



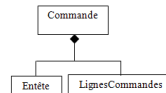
4) Agrégation :



5) Agrégation :

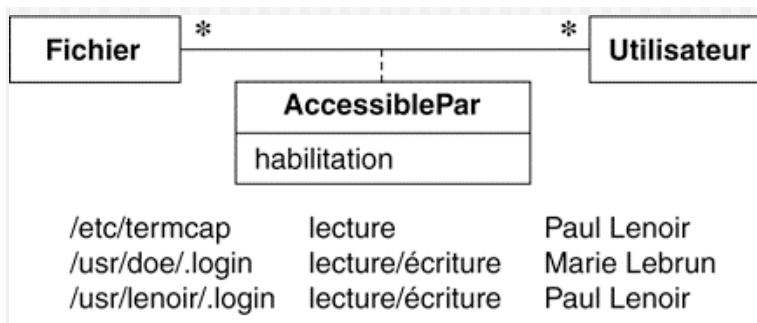


6) composition :



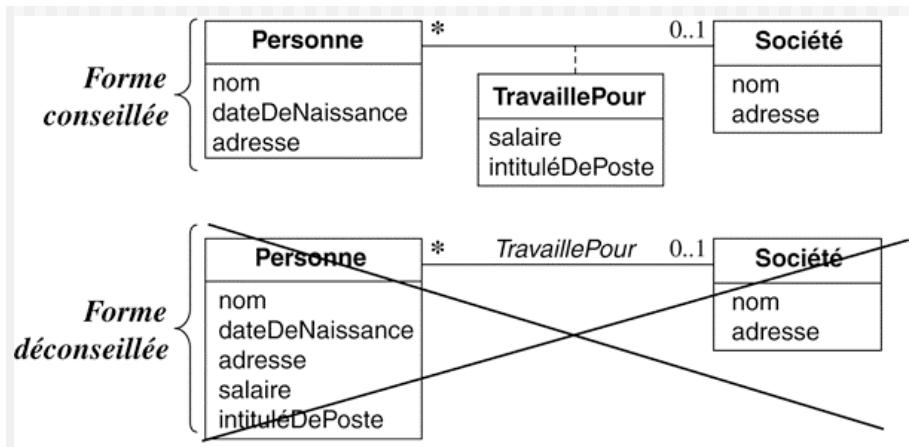
# CLASSE D'ASSOCIATION

- Une **classe-association** est une association qui est aussi une classe.



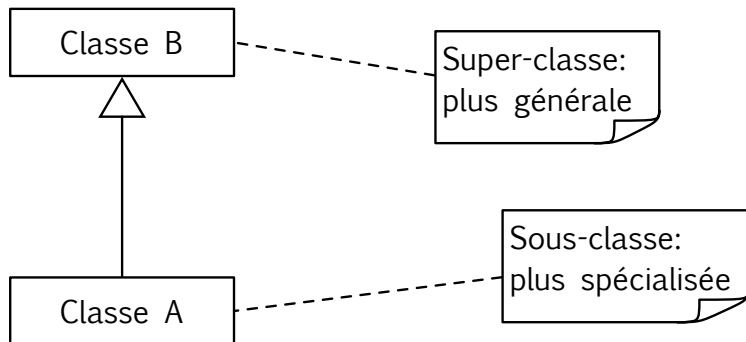
# CLASSE D'ASSOCIATION

- Ne placez pas les attributs d'une association dans une classe.



# HÉRITAGE (GÉNÉRALISATION/SPÉCIALISATION)

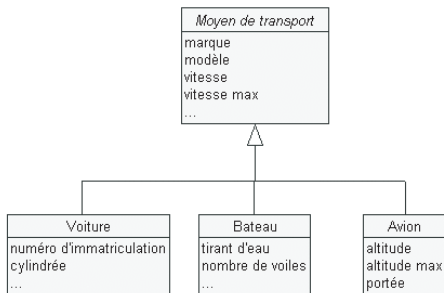
- L'héritage une relation de **spécialisation/généralisation**.
- Les éléments spécialisés héritent de la structure et du comportement des éléments plus généraux (attributs et opérations)





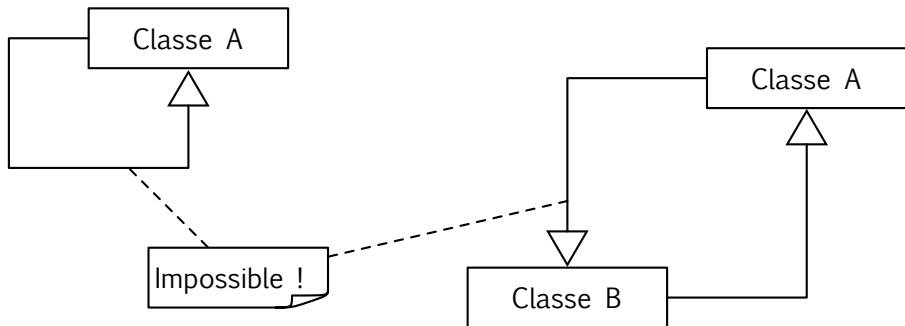
# HÉRITAGE

- Pour que ça fonctionne :
  - **Principe de substitution** : toutes les propriétés de la classe parent doivent être valables pour les classes enfant
  - principe du « **A est un B** » ou « **A est une sorte de B** » : toutes les instances de la sous-classe sont aussi instances de la super-classe. Par exemple, toute opération acceptant un objet d'une classe Animal doit accepter tout objet de la classe Chat (l'inverse n'est pas toujours vrai).

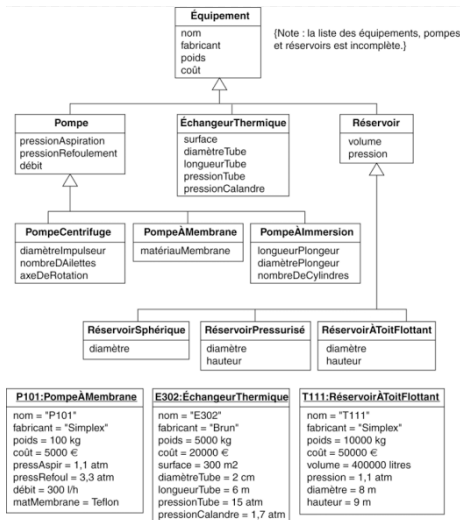


# HÉRITAGE

- Relation non-réflexive, non-symétrique !

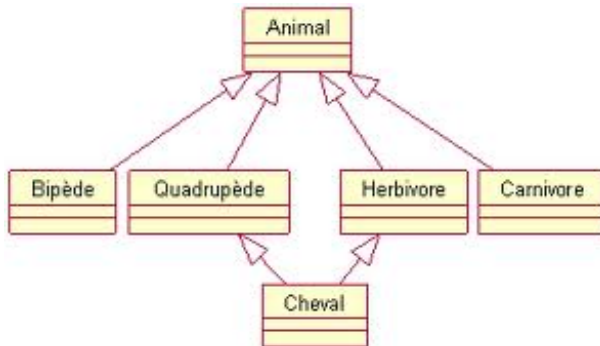


# HÉRITAGE - EXEMPLE



# HÉRITAGE MULTIPLE

- Une classe peut avoir plusieurs classes parents. On parle alors d'héritage multiple.
- Exemple :

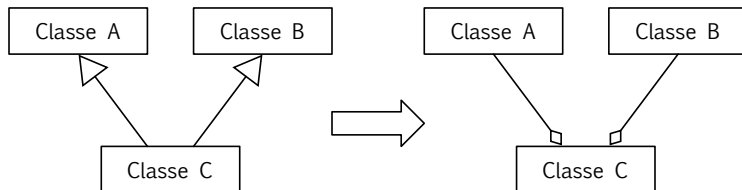


- Le langage C++ est un des langages objet permettant son

# HÉRITAGE MULTIPLE

COMMENT ÉVITER L'HÉRITAGE MULTIPLE ?

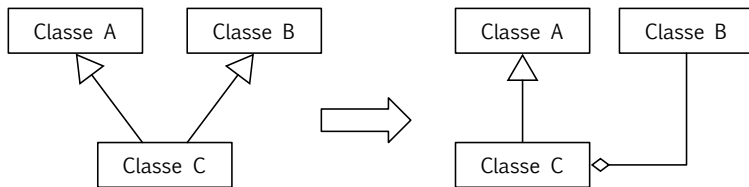
Première solution : **déléguer**



# HÉRITAGE MULTIPLE

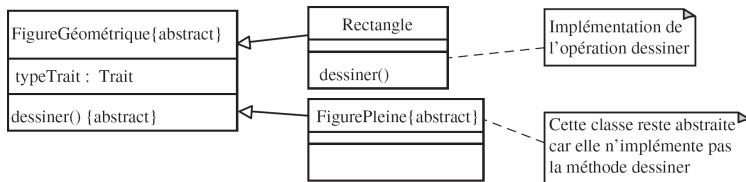
## COMMENT ÉVITER L'HÉRITAGE MULTIPLE ?

Deuxième solution : hériter de la classe la plus importante et déléguer les autres

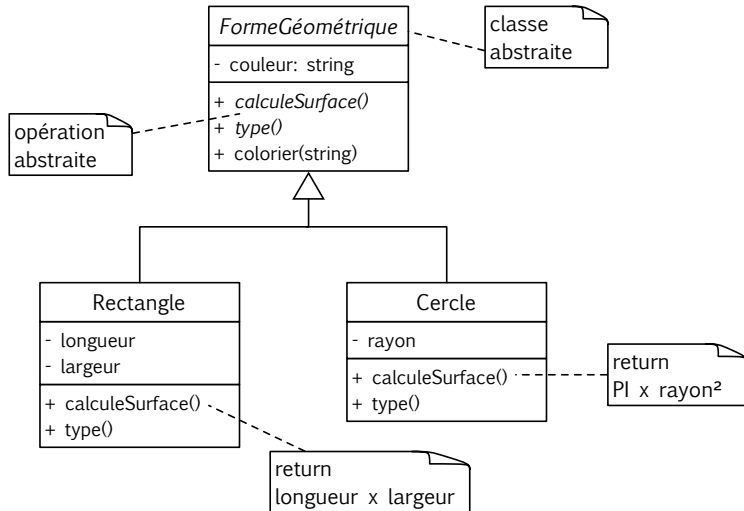


# CLASSES ABSTRAITES

- Une **méthode** est dite **abstraite** lorsqu'on connaît son entête (signature) mais pas la manière dont elle peut être réalisée.
  - Il appartient aux classes enfant de définir les méthodes abstraites.
- Une **classe** est dite **abstraite** lorsqu'elle définit **au moins** une méthode abstraite ou lorsqu'une classe parent contient une méthode abstraite non encore réalisée.



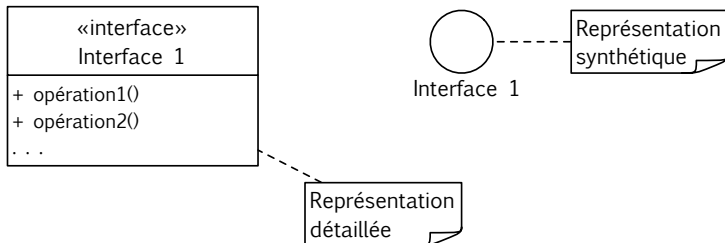
# CLASSES ABSTRAITES - EXEMPLE





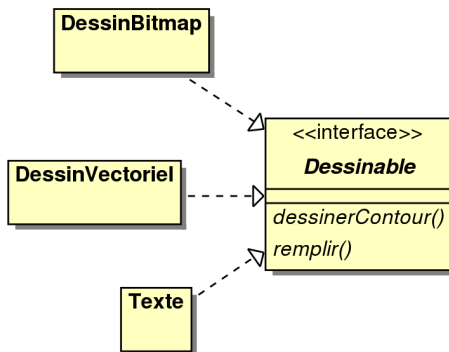
# LES INTERFACES

- Une interface spécifie un ensemble d'opérations (comportement)
- C'est un contrat
  - Les classes liées s'engagent à respecter le contrat
  - elles doivent mettre en oeuvre les opérations de l'interface

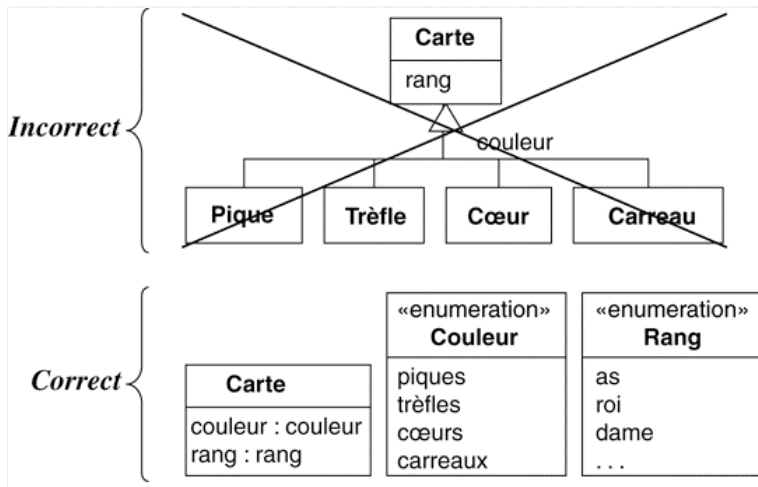


# LES INTERFACES

- On utilise une relation de type **réalisation** entre une interface et une classe qui l'implémente.
- Les classes implémentant une interface doivent implémenter toutes les opérations décrites dans l'interface



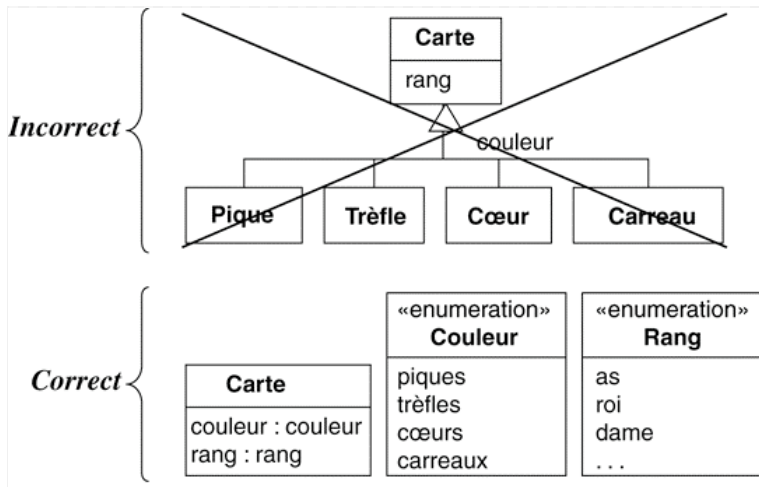
# ENUMÉRATION



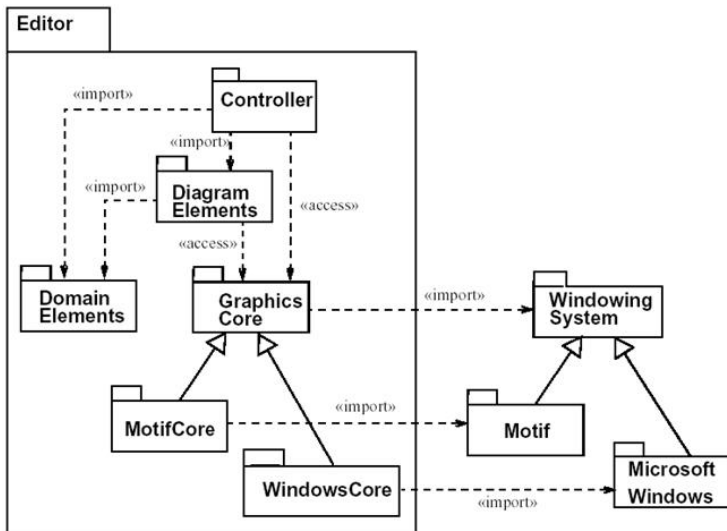
# PACKAGES

- Un package permet de grouper des éléments
- Un package sert d'espace de désignation
- Un package peut inclure d'autres package
- Un package peut importer d'autres package
- L'héritage entre package est possible

# PACKAGES



# PACKAGES





Michael Blaha et James Rumbaugh *Modélisation et conception orientées objet avec UML2*,



Gérard, Pierre *Génie Logiciel - Principes et Techniques*,



A. Lewandowski *Méthode de Conception Orientée Objet*