Modélisation et Conception Objet Avancées

« INTRODUCTION

Le concept d'Orienté Objet »

2A-ICL
L. Gzara, A. Shahwan

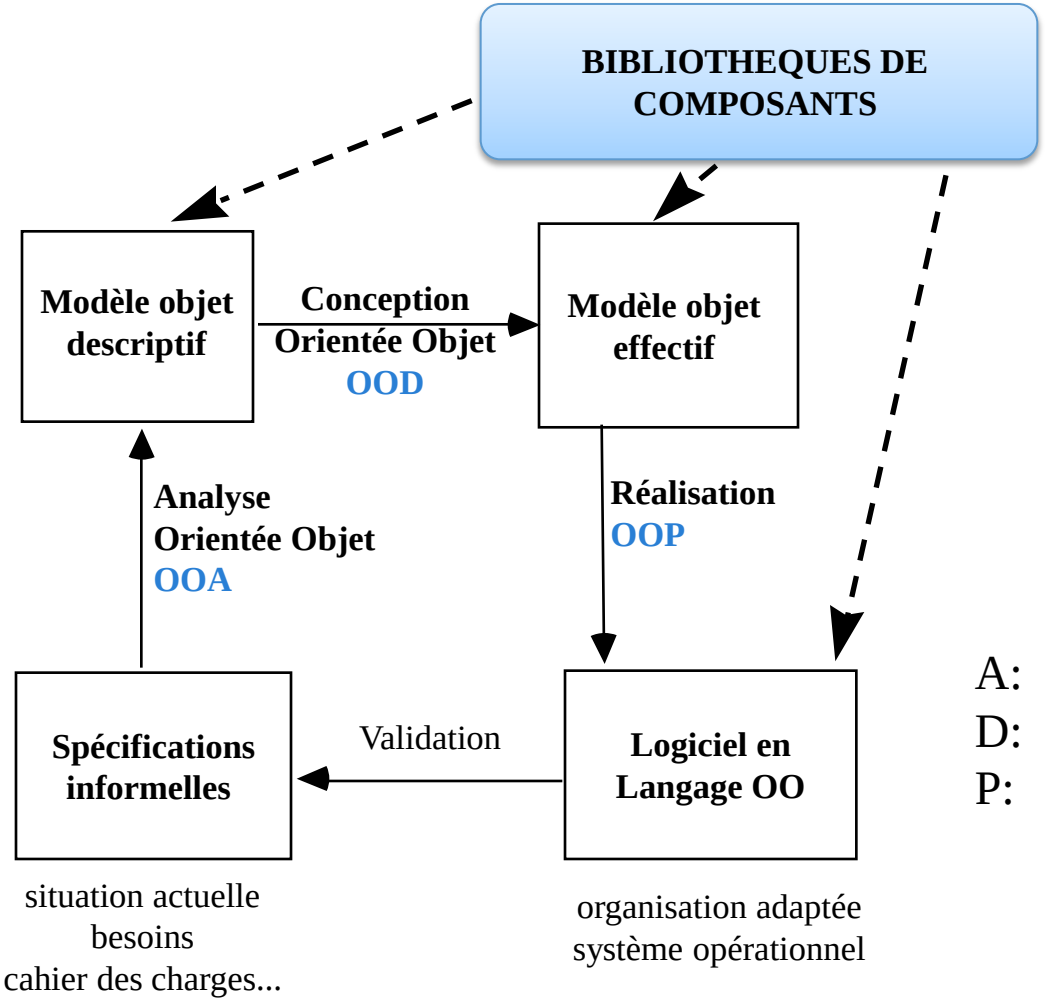
L'OBJET
UNE VAGUE DEFERLANTE

- **1) GENIE LOGICIEL**
  - Une discipline qui touche à tout aspect production de logiciel.
  - Des méthodes et outils orienté objet (Rational Unified Porcess, UML).
  - Des Langages Orientés Objets: C++, Java (J2EE, EJB…), DotNet…
- **2) INTELLIGENCE ARTIFICIELLE**
  - Modélisation du raisonnement humain.
  - Des Systèmes de Représentation de Connaissances à Objets.
- **3) BASES DE DONNEES**
  - Prise en compte d'Objets Complexes et Evolutifs
  - Langages de programmation et BD.
  - Des SGBDOO : O2, Versant, Objectivity, ObjectStore, Perst, db4o…

Qualité d'un bon logiciel

- Modularité
  - REPRESENTER LE SYSTEME SOUS LA FORME D'UN ENSEMBLE STRUCTURE DE MODULES:
    - autonomes
    - communiquant par un protocole unique
    - stables dans le temps
    - réutilisables dans d'autres système
  - Système extensible par ajout de composants (modules) éventuellement existants

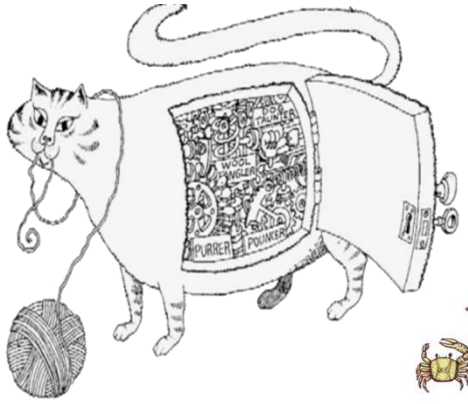
REUTILISATION

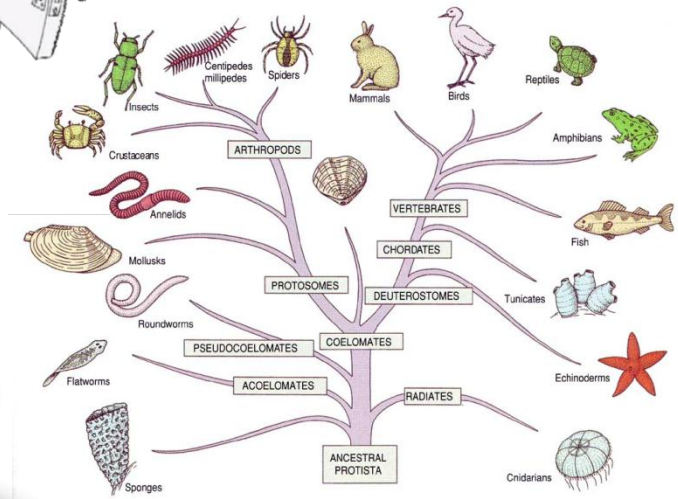


A: Analysis
D: Design
P: Programming

Les 3 Principes Orienté Objet

- **Encapsulation**
- **Héritage**
- **Polymorphisme**





VERS LA PROGRAMMATION OBJET

- **De la programmation structurée (Pascal, C, Basic…)**
  - **procédures + structures de données**

  ➔ diviser pour mieux régner

  ➔ programmation dirigée par les traitements

  ➔ limite : réutilisation difficile, lien structure-traitement non explicite
- **A la programmation objet (Smalltalk, C++, C#, Java…)**
  - **données + procédures = objet : lien structure-traitement explicite**
  - **communication par l'interface**

  ➔Programmation dirigée par les objets

  ➔Objet : une entité informatique complète

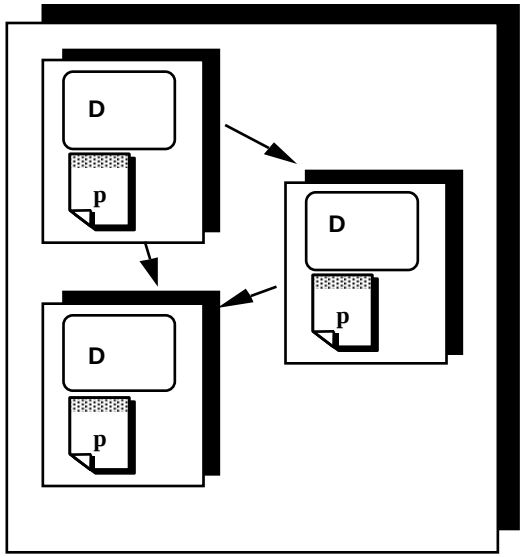
PROGRAMMATION CLASSIQUE / PAR OBJETS

Programmation par objets:

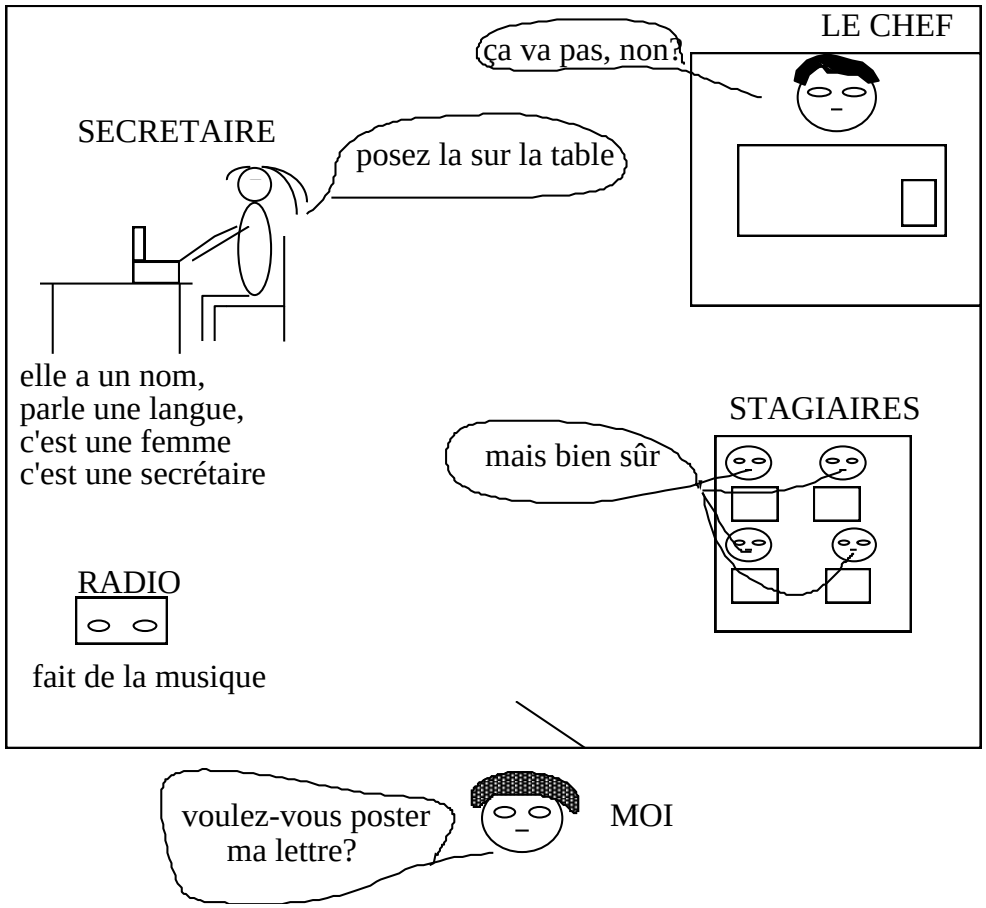
un programme = un ensemble d'objets

un objet = données + procédures

Interaction = par envoi de message



OO : une Approche Intuitive



UNE PREMIERE APPROCHE

- **ENVIRONNEMENT COMPOSE D'OBJETS**
  une radio
- **DETIENNENT LEUR COMPORTEMENT**
  faire de la musique
- **CONNAISSANCE DE CE COMPORTEMENT**
  je sais qu'une radio fait de la musique
- **MAIS PAS DE SA REALISATION**
  aucune connaissance d'électronique
- **L'OBJET REAGIT A DES MESSAGES**
  tourner un bouton → musique

UNE PREMIERE APPROCHE

- **PLUSIEURS OBJETS PEUVENT REAGIR AU MEME MESSAGE**

  qui veut poster ma lettre ?
- **MAIS DE DIFFERENTES MANIERES**

  réponses différentes du chef et de la secrétaire
- **L'OBJET APPARTIENT A UNE CATEGORIE D'OBJETS**

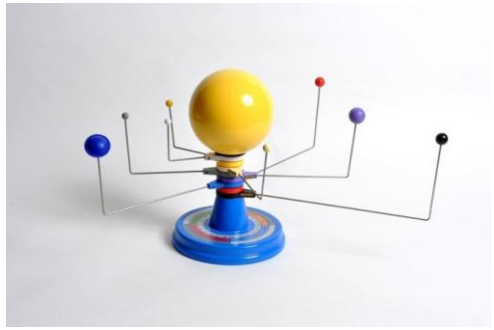
  la secrétaire est une personne, le message n'est pas envoyé à la radio
- **A DES SOUS-CATEGORIES**

  c'est une femme et plus précisément une secrétaire

LA MODÉLISATION ORIENTÉ OBJET

• Un **modèle** est une abstraction (représentation simplifiée) de certains aspects du monde réel.

Monde Réel



Modèle

• La **modélisation**:

Créer une représentation des éléments du monde réel auxquels on s'intéresse, sans se préoccuper de l'implémentation (indépendamment d'un langage de programmation).

• La modélisation **orienté objet**:

Il s'agit de déterminer les objets présents et d'isoler leurs données et les fonctions qui les utilisent. Pour cela des méthodes ont été mises au point.

Outils et Méthodes OO

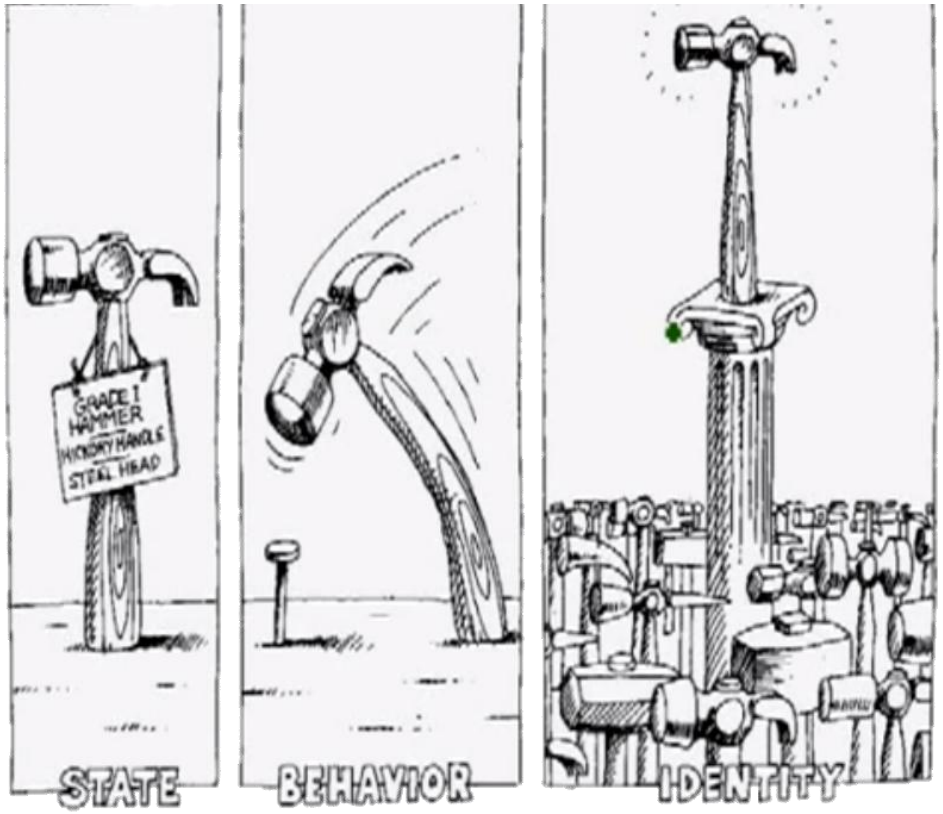
- Entre 1970 et 1990, de nombreux analystes ont mis au point des approches orientées objets, si bien qu'en 1994 il existait plus de 50 méthodes objet.
- A partir de 1994, unification des efforts pour mettre au point la méthode unifiée (*Unified Method 0.8*), incorporant les avantages de chacune des méthodes précédentes. La méthode unifiée à partir de la version 1.0 devient **UML** (*Unified Modeling Language*), une notation universelle et standardisée pour la modélisation objet.
- Dernière Version en date d'UML : 2.4.1
- UML n'est pas une méthode dans la mesure où il ne présente aucune démarche. UML est juste un langage de modélisation objet.
- RUP (*Rational Unified Process*) et OUM (*Oracle Unified Method*) sont des méthodes qui appuie sur les notions d'UML.

OBJET

Objet : une entité ayant une existence matérielle (un arbre, une personne, un radio…) ou bien virtuelle (un compte bancaire…). Un objet est caractérisé par plusieurs notions:

- Les attributs (propriétés): données caractérisant l'objet. Ce sont des variables stockant des informations d'état de l'objet.
- Les méthodes (appelées parfois fonctions membres): caractérisent son comportement, c'est-à-dire l'ensemble des actions (appelées opérations) que l'objet est à même de réaliser. Ces opérations permettent de faire réagir l'objet aux sollicitations extérieures ou d'agir sur les autres objets. Les opérations sont étroitement liées aux attributs, car leurs actions peuvent dépendre des valeurs des attributs, ou bien les modifier.
- L'identité: permet de le distinguer des autres objets, indépendamment de son état. On construit généralement cette identité grâce à un identifiant découlant naturellement du problème (par exemple un produit pourra être repéré par un code, une voiture par un numéro de série…).

OBJET



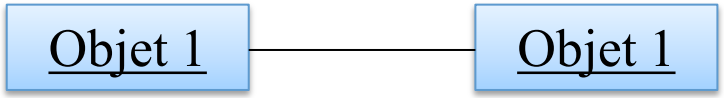
An object has state, exhibits some well-defined behavior, and has a unique identity.

OBJET

- **Les attributs d'un objet**
  - L'ensemble des valeurs des attributs d'un objet constituent son état.
  - UML propose de représenter un objet de la manière suivante.
  - Les méthodes ne sont pas représentées

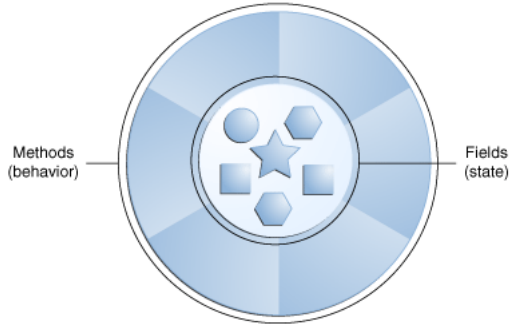
Laure: Etudiant
Taille = 170
Poids = 57
Ville-naissance = « Grenoble »

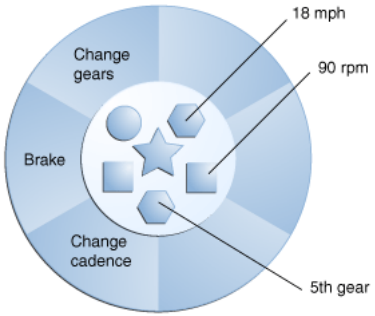
- **Les liens entre objets**
- **Les objets ne sont pas des corps inertes isolés. Même s'ils possèdent leurs caractéristiques propres par l'intermédiaire des valeurs de leurs attributs, ils ont la possibilité d'interagir entre-eux grâce à leurs méthodes.**



Objet
données + comportement

- **ÉTAT:** la structure de données de l'objet
  - ensemble des valeurs d'attributs (variables d'instances)
- **COMPORTEMENT:** ensemble de méthodes (opérations, routines)
  - accès aux données
  - envoi de messages à d'autres objets
- **Seules les méthodes sont autorisées à modifier les données**



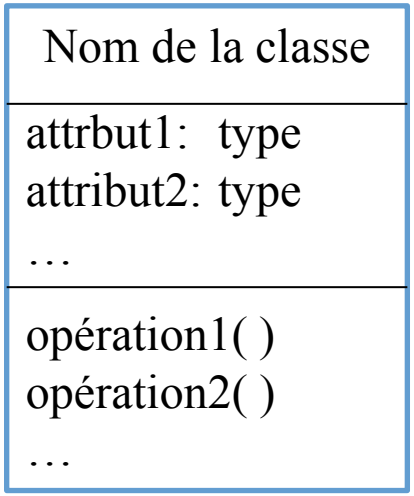


Classe

- Classe: une **entité** décrivant un ensemble d'objets **de même** :
  - structure → ATTRIBUTS
  - comportement → METHODES
  - protocole de communication → INTERFACE (on verra ça plus tard)
- Un **moule** pour générer ses représentants physique (INSTANCES).
- Un objet est donc "issu" d'une classe, on dit que c'est une *instanciation* d'une classe.
- Exemple
  - Si on définit la classe Salle, les objets F007A, F007B, H114, H203 seront des instanciations de cette classe.

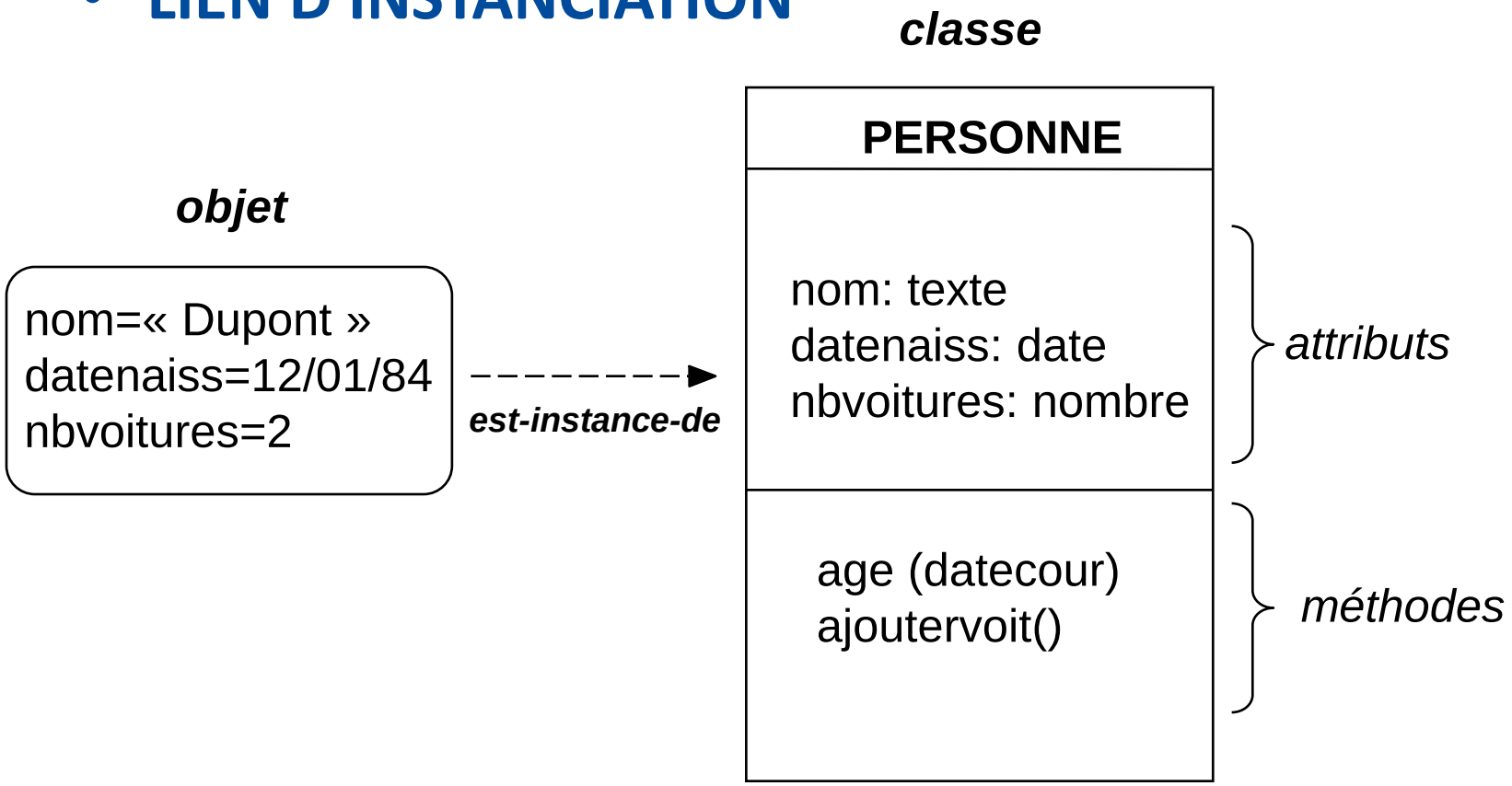
Classe
attributs + méthodes

- Une classe est composée:
  - d'attributs: il s'agit de la structure de données, dont les valeurs représentent l'état de l'objet.
  - de méthodes : il s'agit des opérations applicables aux objets.
- Une classe est représentée de la manière suivante en UML.



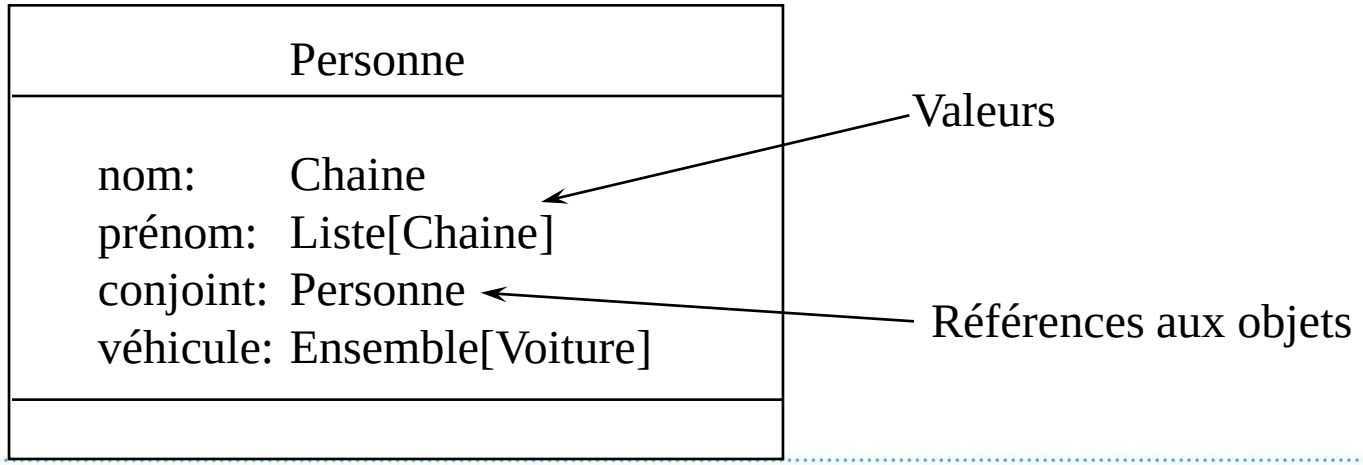
CLASSE, OBJET, INSTANCIATION

- **LIEN D'INSTANCIATION**



PARTIE STATIQUE
ensemble d'attributs

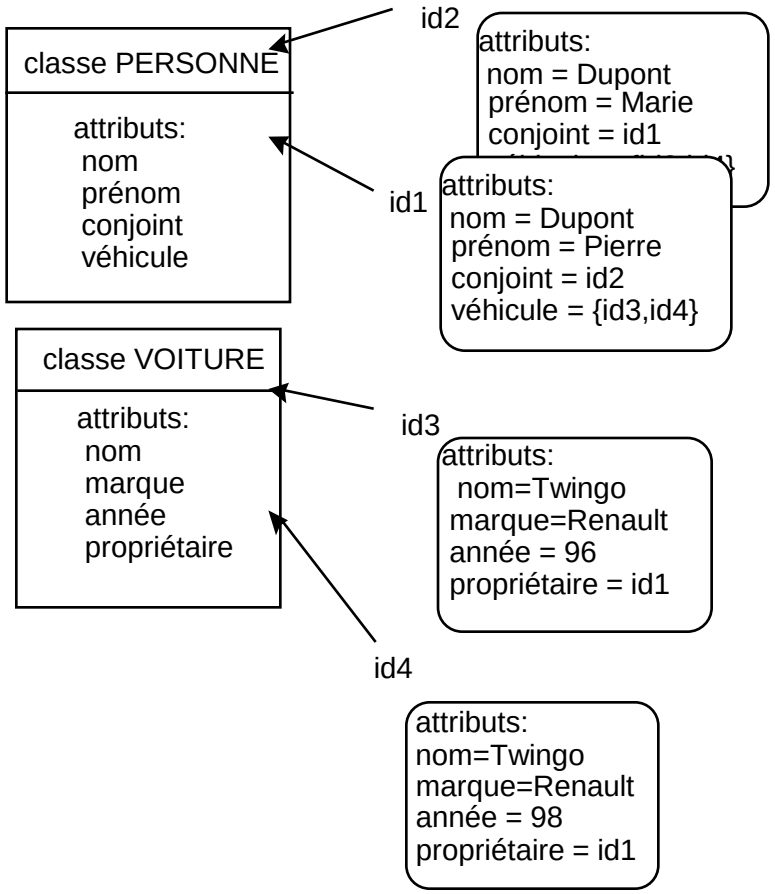
- **Qu'est ce qu'un attribut?**
  - **un nom + un type :** un champ de la structure d'un objet **(**définition dans la classe**)**
  - **une place mémoire** : contient une ou plusieurs **valeurs**, ou bien une ou plusieurs **références** sur d'autres objets (utilisation dans l'instance)
- La partie statique (les attributs) définit **l'ensemble des valeurs possibles** d'un
- objet donc définit ses **états possibles.**



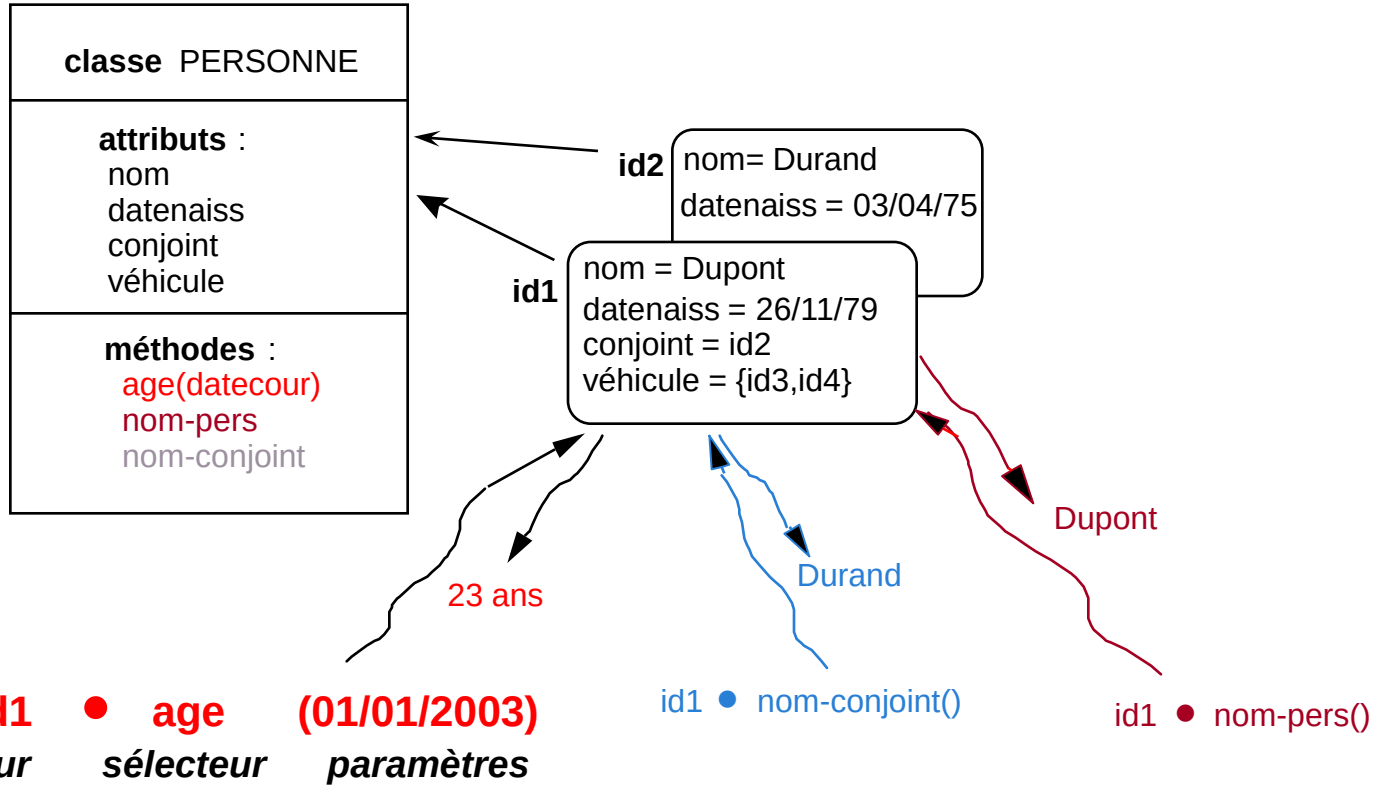
IDENTITE D'OBJET

- **Identificateur Système (Object IDentifier)**
  - indépendant de l'adresse et des valeurs des objets (des attributs)
- **Permet de mieux comparer les objets**
  - Objets identiques
    ⇔ même identificateur
  - Objets égaux
    ⇔ même valeur des attributs

- Donc :
  - Objet identifié par son identificateur
  - Valeur identifiée par elle même

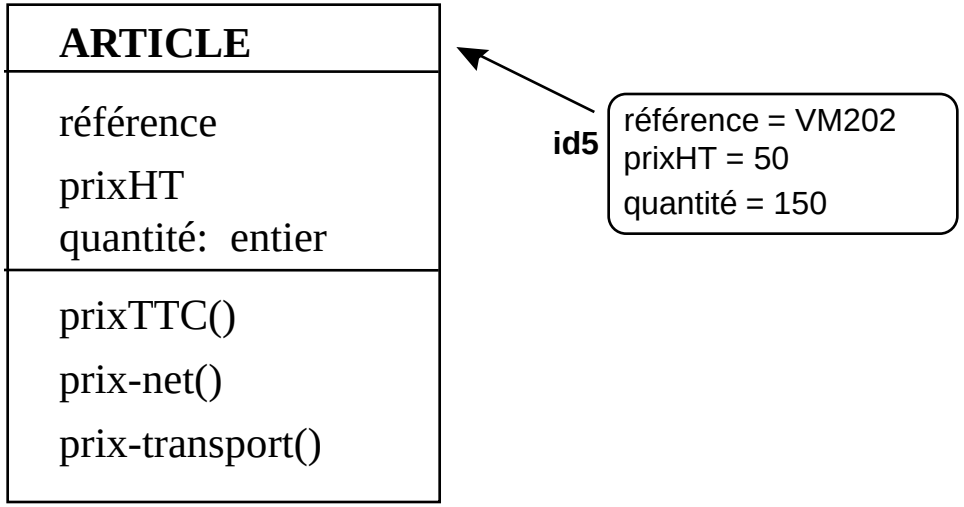


PARTIE DYNAMIQUE
ensemble de méthodes + messages



CODE DES METHODES

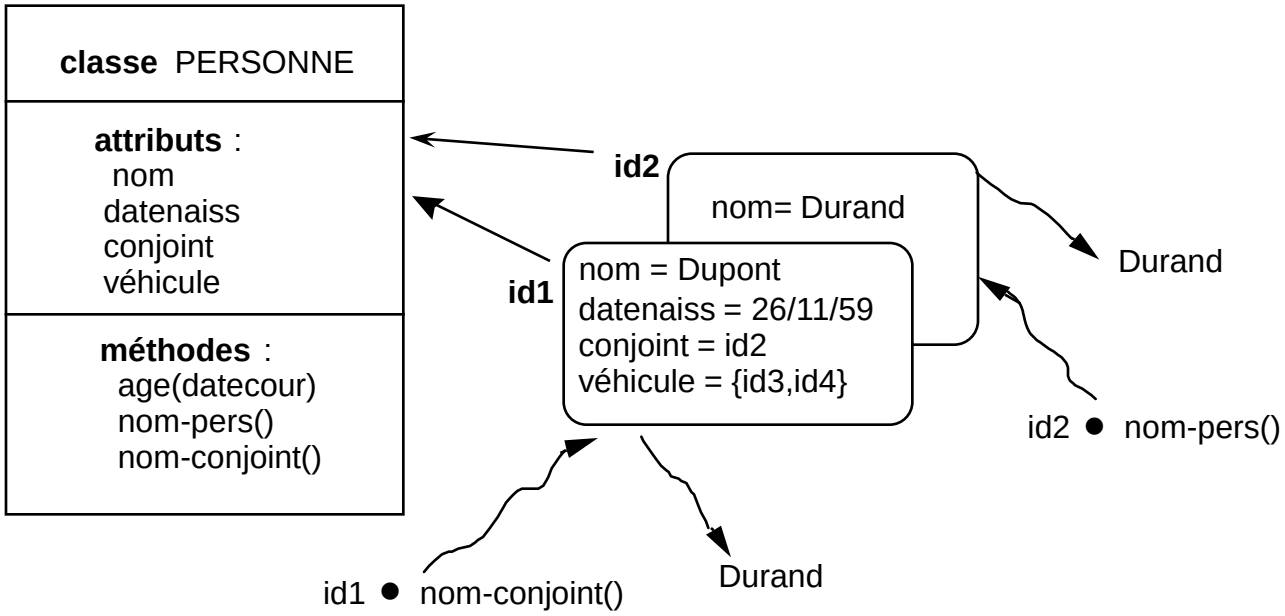
- **Les self-messages**
  - Permet à un objet de s'auto-envoyer des messages
  - Exemple :



- Pour l'objet id5, on veut calculer son prix net = prixTTC + prix-transport
- ***Comment coder la méthode prix-net?***
- **prix-net():** retourner **self .** prixTTC() + **self .** prix-transport()

  self étant une référence désignant l'objet actuel.

CODE DES METHODES

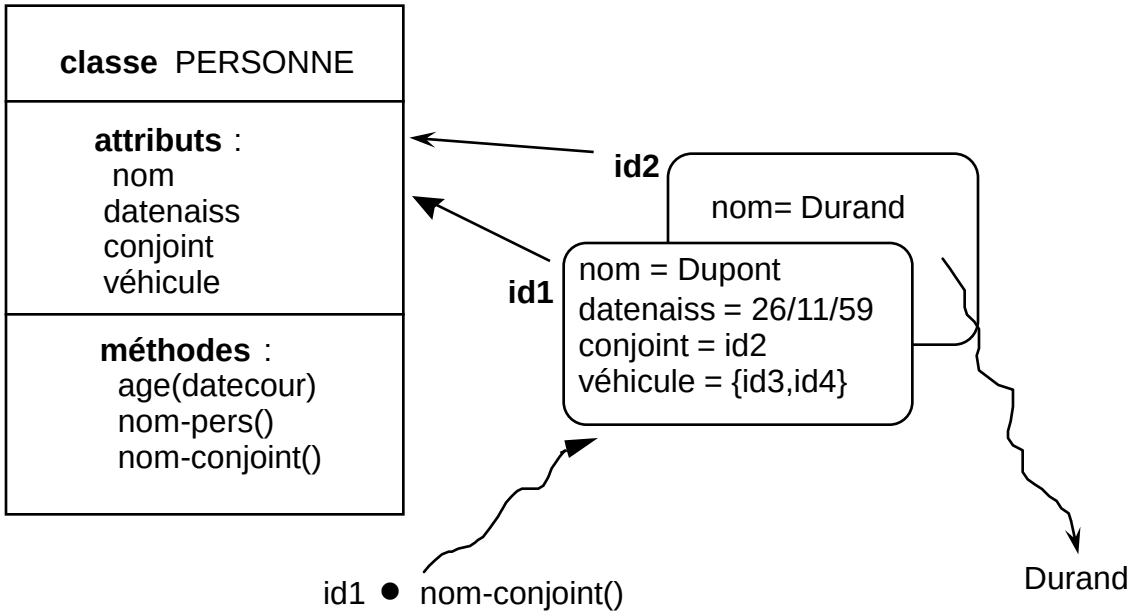


- **Cas où le domaine privé = instance**
  - Seule l'instance peut directement manipuler (lire, écrire) ses propres attributs. Il faut sinon passer par un message et donc une méthode.
  - Exemple :

    nom-pers (): retourner nom

    nom-conjoint (): retourner conjoint · nom-pers()

CODE DES METHODES



- **Cas où le domaine privé = classe**
  - Toute instance de la classe peut directement manipuler (lire, écrire) les attributs de tout autre instance de cette même classe.
  - Exemple :

    nom-pers (): retourner nom

    nom-conjoint (): retourner conjoint · nom

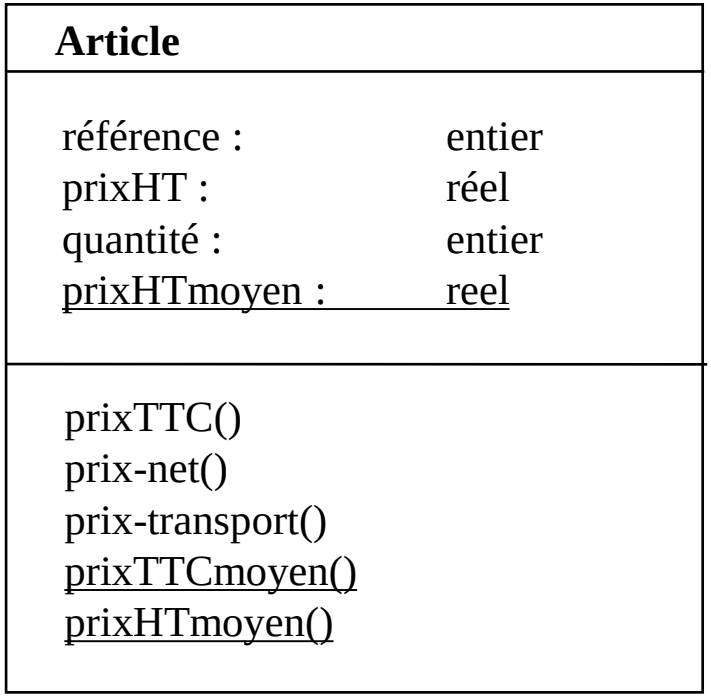
ATTRIBUTS & METHODES DE CLASSES

- **Attribut de classe :**
  - sa valeur est partagée par l'ensemble des objets de la classe
- **Méthode de classe :**
  - les messages invoquant cette méthode ont pour receveur la classe
- **Exemple :**

  Article.prixTTCmoyen()

- **ATTENTION !**

  Une classe n'est receveur d'un message que si la méthode invoquée est une méthode de classe



Encapsulation

- C’est un mécanisme consistant à rassembler les données et les méthodes au sein d'une structure en cachant l'implémentation de l'objet, c'est-à-dire **en empêchant l'accès aux données par un autre moyen que les services proposées** ; des méthodes sont créées à cet effet (on parle d'**interface**). L'encapsulation permet de garantir l'intégrité des données contenues dans l'objet.
- En interdisant l'utilisateur de modifier directement les attributs, et en l'obligeant à utiliser les fonctions définies pour les modifier (**interfaces**), l'encapsulation permet de garantir l'intégrité des données contenues dans l'objet.
- L'encapsulation permet de définir des niveaux de visibilité des éléments de la classe. Ces niveaux de visibilité définissent les droits d'accès aux données selon que l'on y accède par une méthode *de la classe elle-même*, *d'une classe héritière*, ou bien *d'une classe quelconque*.

Encapsulation

- **Pour chaque objet, il y a :**
  - Ceux qui le **programment**, et
  - Ceux qui l'**utilisent** (dans d'autres programmes).

  Ce ne sont pas toujours les mêmes.

- **Pour chaque objet, il y a :**
  - Des attributs et des méthodes **internes**, utilisés pour l'**implémenter**.
  - Des attributs et des méthodes **externes**, utilisés pour l'**interfacer**.

  Ce ne sont pas toujours les mêmes.

Encapsulation

- Un objet a une **interface** par laquelle on le manipule, et que tout le monde connait. Cette interface fait partie de sa **spécification**. Elle est tout ce dont ont besoin les utilisateurs de l'objet.
- On a tout intérêt à **cacher** les sordides détails de l'implémentation d'un objet à ses utilisateurs :
  - Ils en ont ainsi une vue plus simple, plus lisible;
  - Leur code ne risque pas d'interférer avec le code interne à l'objet;
  - On peut changer l'implémentation d'un objet de manière indolore/transparente pour ses utilisateurs.

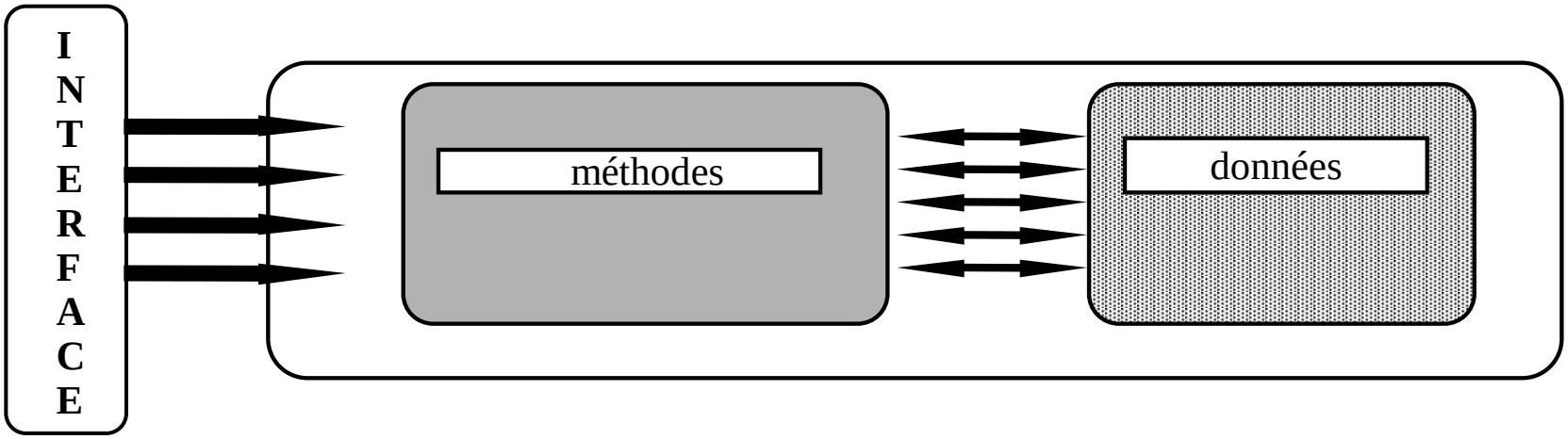
Encapsulation

- Techniquement, l'encapsulation c'est très simple : un attribut ou une méthode sont :
  - Soit **publics**, visibles par tout le monde, et moralement destinés à être accédés par tous les utilisateurs de l'objet ;
  - Soit **privés**, relatif au fonctionnement interne de l'objet, et cela ne regarde pas les utilisateurs.
  - Il y a en général au moins un niveau d'accès intermédiaire, appelé **protégé**. Il permet l'accès par du code "ami" : typiquement, les autres objets d'une même bibliothèque et les objets des sous-classes.
- Remarque : On peut implémenter cette philosophie dans n'importe quel langage, mais les langages OO apportent une *garantie mécanique* qu'elle est respectée.

Encapsulation

- **Pour être plus clair, qui fait quoi dans une équipe projet informatique :**
  - Le chef donne la spécification de chaque objet, en décrivant
    - sa syntaxe **publique** (en UML par exemple) ;
    - sa sémantique (en langue naturelle jusqu'a nouvel ordre).
  - Le programmeur responsable de l'objet bricole comme il veut ce qui est **privé**, tant qu'il respecte la spécification.
  - L'utilisateur de l'objet (qui est aussi un programmeur) ne voit et n'a besoin que de ce qui est **public**. Il ne risque pas d'interférer avec le boulot du responsable de l'objet puisqu'il ne le voit pas.

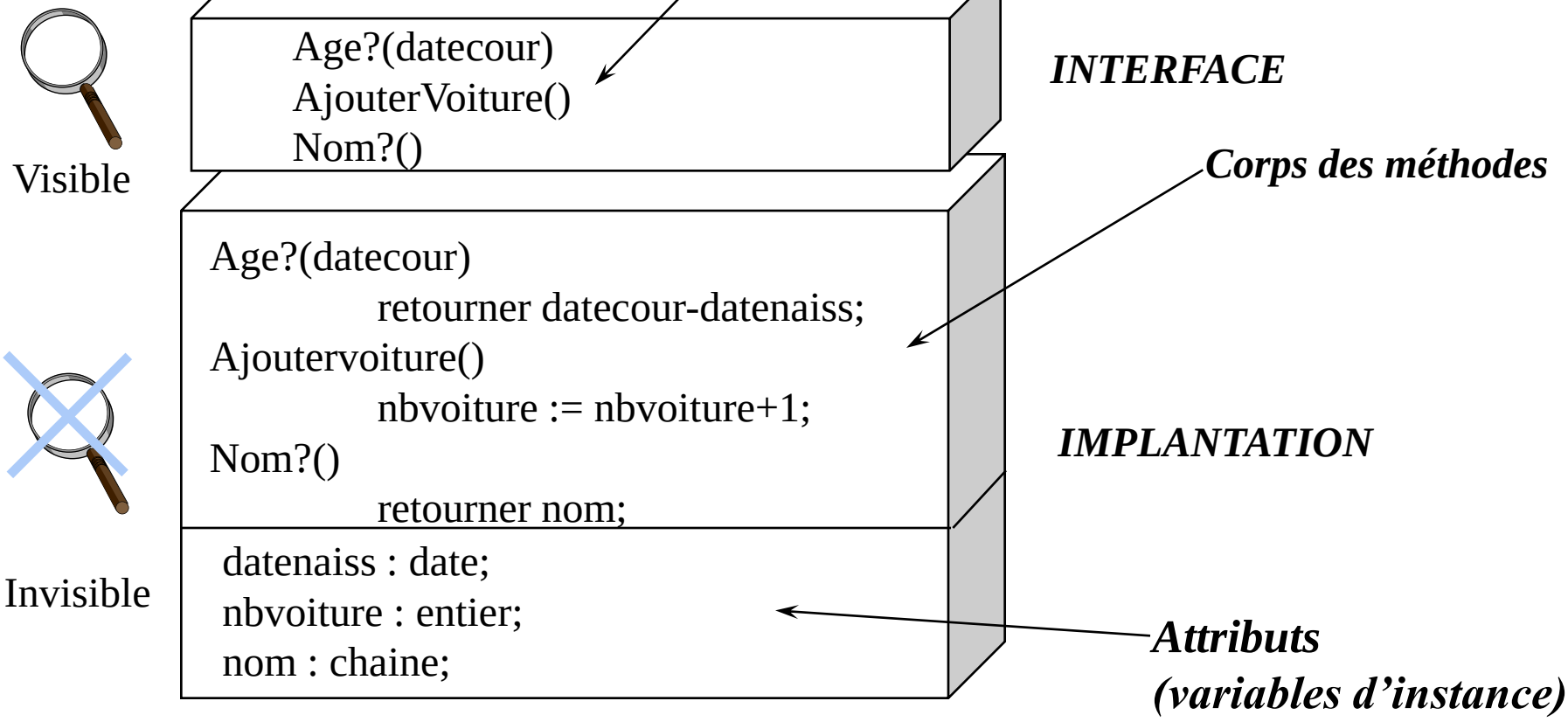
Encapsulation: résumé



- **Activation d'une méthode uniquement via un sélecteur de l'interface**
  - l'implantation peut évoluer sans toucher à l'interface.
  - protection contre les mauvaises utilisations
  - l'utilisateur peut ignorer l'implantation (le comment)
- **Seules les méthodes de l'objet sont autorisées à modifier ses propres données**
  - Sécurité et cohérence de l'objet

Encapsulation: exemple

Classe PERSONNE



Héritage

- **Le mécanisme d'héritage permet de définir une classe à partir d'une ou plusieurs autres classes**
- **Factorisation, spécialisation, redéfinition des connaissances.**
- **Exemple**
  - les cercles et les triangles sont des objets graphiques qui ont des couleurs, des dimensions et qui peuvent être dessinés, colorées, tournés…

Héritage

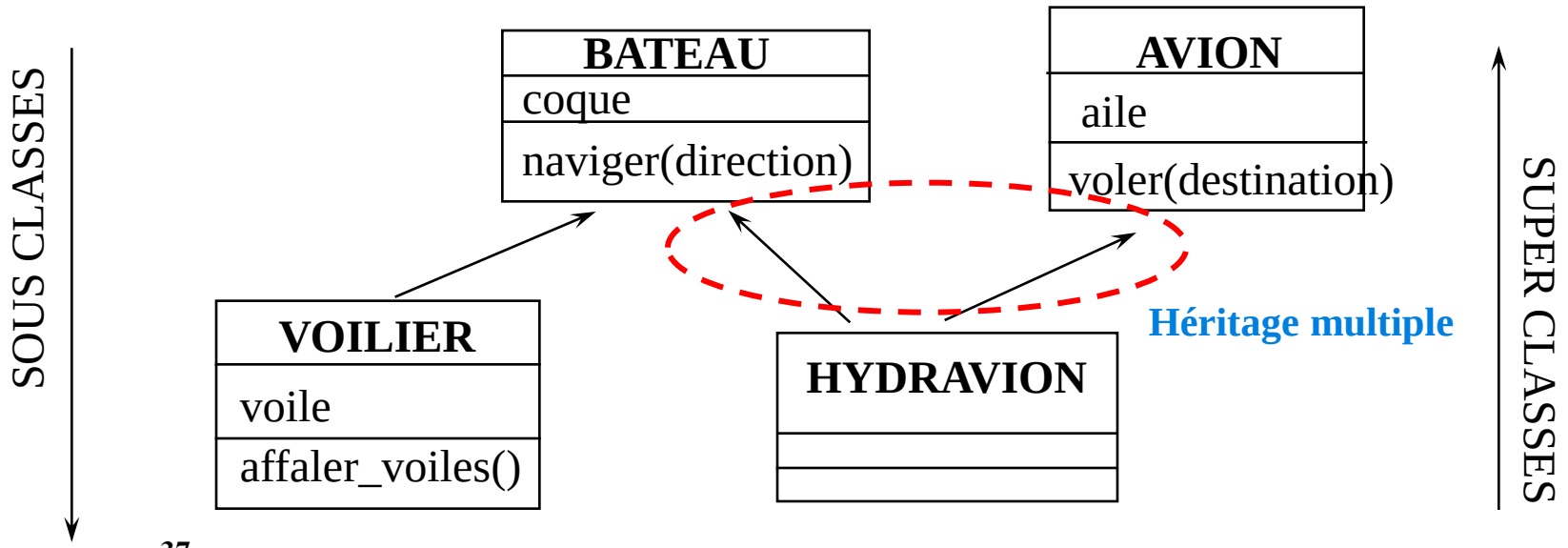
- Définition :
  - La classe **Machin** hérite de la classe **Truc** signifie tout simplement :
  - **Machin** *est un* **Truc** / **Machin** *est une sorte de* **Truc**
- Une **coccinelle** est un **coléoptère** qui est un **insecte** qui est un **animal** qui est un **être vivant**.
- Ce qui montre qu'on peut construire un graphe d'héritage, aussi appelé hiérarchie *is-a* ou hiérarchie de classes.

Héritage

- Principe propre à la programmation OO, permettant de créer une nouvelle classe à partir d'une classe existante.
- Le nom d'*héritage* (appelé aussi dérivation de classe) provient du fait que la classe dérivée (la classe nouvellement créée) contient les attributs et les méthodes de sa superclasse (la classe dont elle dérive).
- L'intérêt majeur de l'héritage est de pouvoir définir de nouveaux attributs et de nouvelles méthodes pour la classe dérivée, qui viennent s'ajouter à ceux et celles héritées.
- Par ce moyen on crée une hiérarchie de classes de plus en plus spécialisées.
- Avantage majeur : ne pas avoir à repartir de zéro lorsque l'on veut spécialiser une classe existante.
- Il est possible d'acheter dans le commerce des librairies de classes, qui constituent une base, pouvant être spécialisées à loisir (d'où l'intérêt pour l'entreprise qui vend les classes de protéger les paramètres grâce à l'encapsulation...).

Héritage

- Relation entre classes pour le partage de savoir (attributs) et de savoir faire (méthodes)
  - **Factorisation** dans une super classe d'attributs et de méthodes de ses sous classes
  - **Distribution** d'une super classe vers les sous classes d'attributs et de méthodes
- Graphe d'héritage
  - Arbre : héritage simple
  - Treillis : héritage multiple

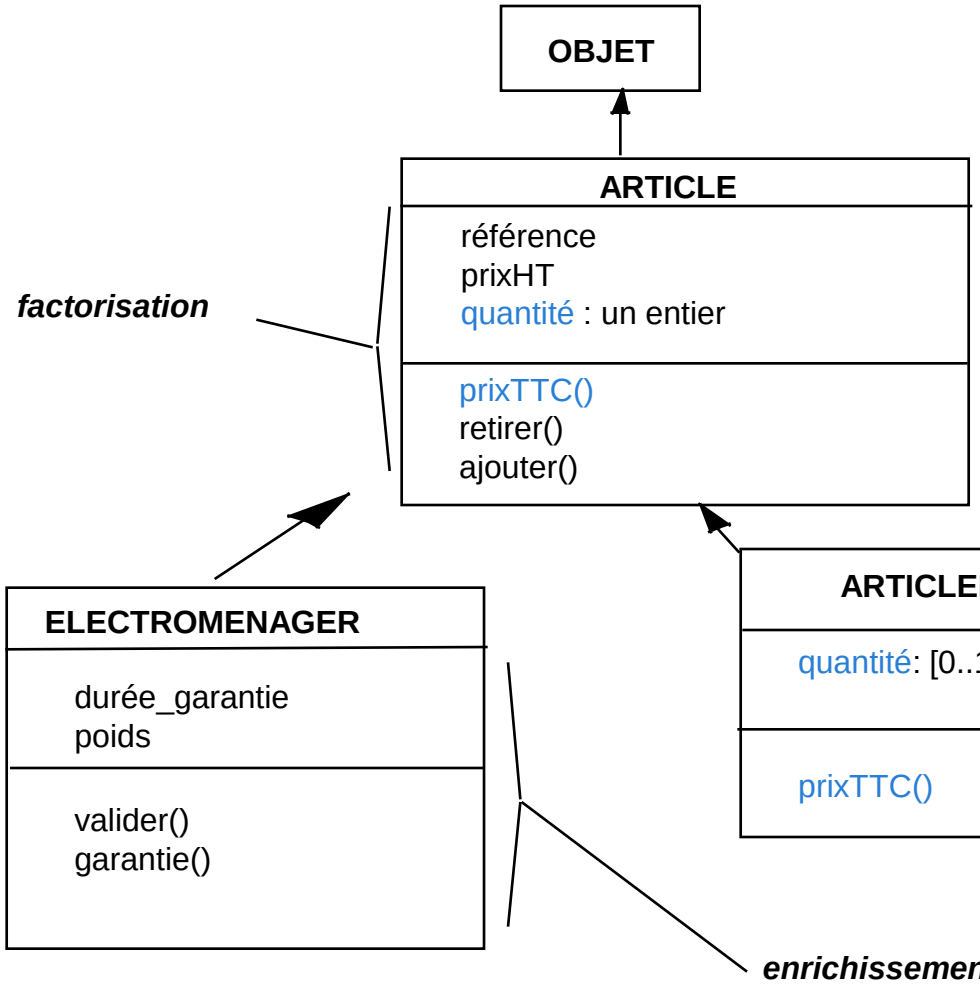


Héritage : caractéristiques

- **Spécialisation : de la super vers les sous classes**
  - Redéfinition des propriétés (attributs ou méthodes) héritées;
  - Rajout de nouvelles propriétés (attributs ou méthodes).

- **Généralisation : de sous classes vers une super classe**
  - Mise en facteur de méthodes dans la super classe, en généralisant éventuellement leur code.
  - Mise en facteur d'attributs dans la super classe, en généralisant éventuellement leur type.

Héritage: exemple



Dans ARTICLE :

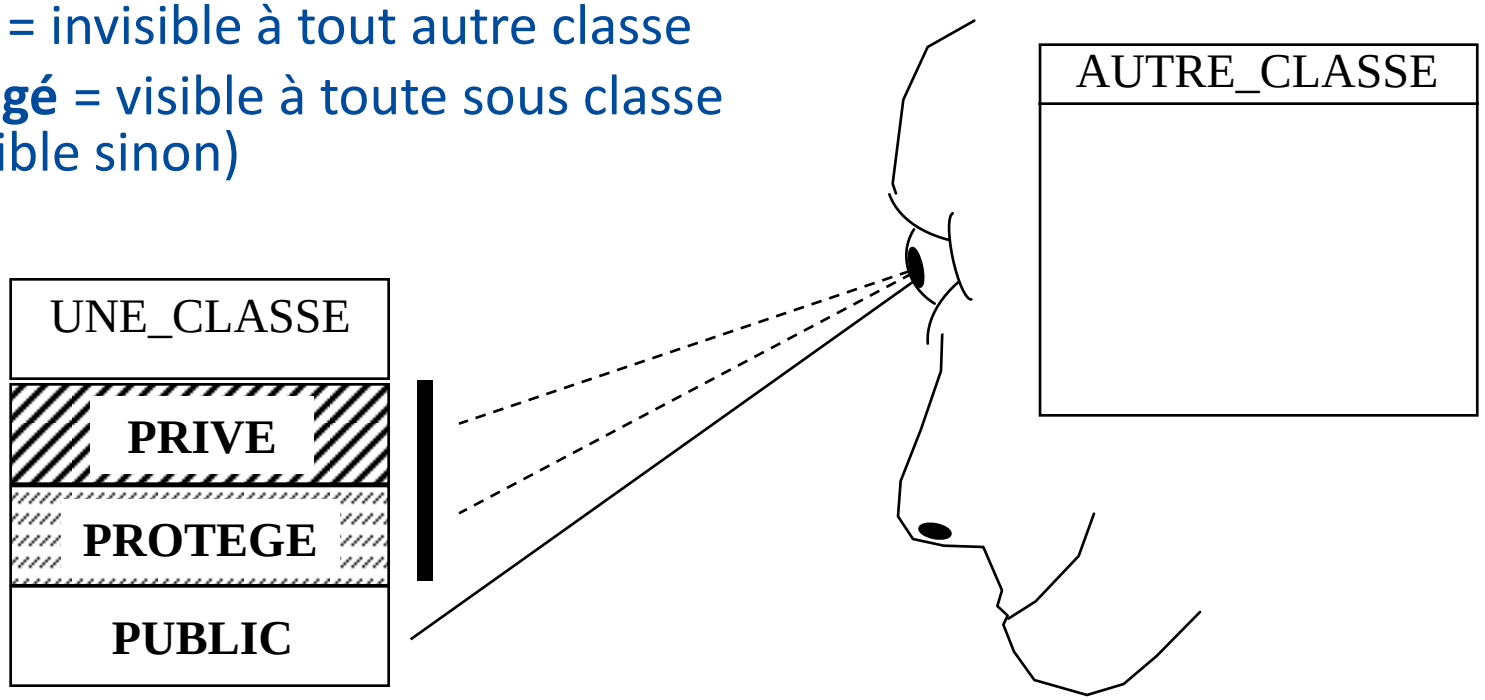
```
prixTTC() : retourner 1,206 * prixHT
```

Dans ARTICLELUXE :

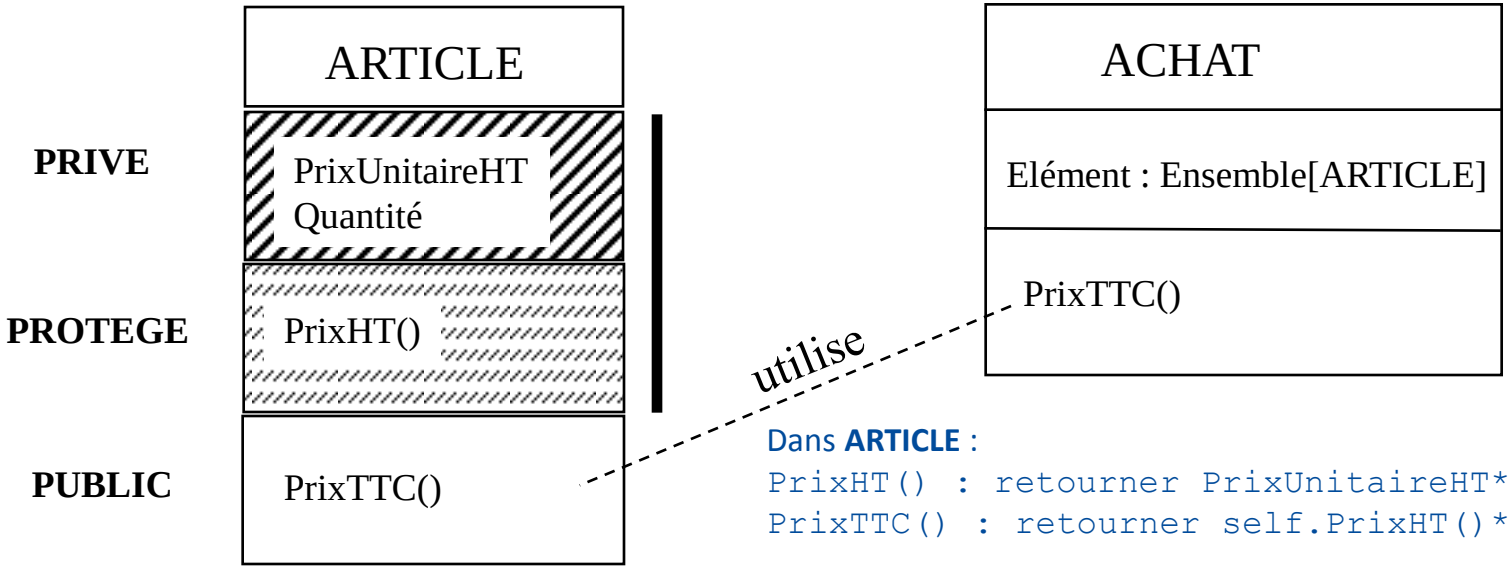
```
prixTTC() : retourner 1,33 * prixHT
```

VISIBILITE ENTRE CLASSES

- Pour une classe donnée :
- **Public** = visible depuis toute autre classe
- **Privé** = invisible à tout autre classe
- **Protégé** = visible à toute sous classe (invisible sinon)



VISIBILITE : exemple



Dans **ARTICLE** :

```
PrixHT() : retourner PrixUnitaireHT*Quantité
PrixTTC() : retourner self.PrixHT()*1,206
```

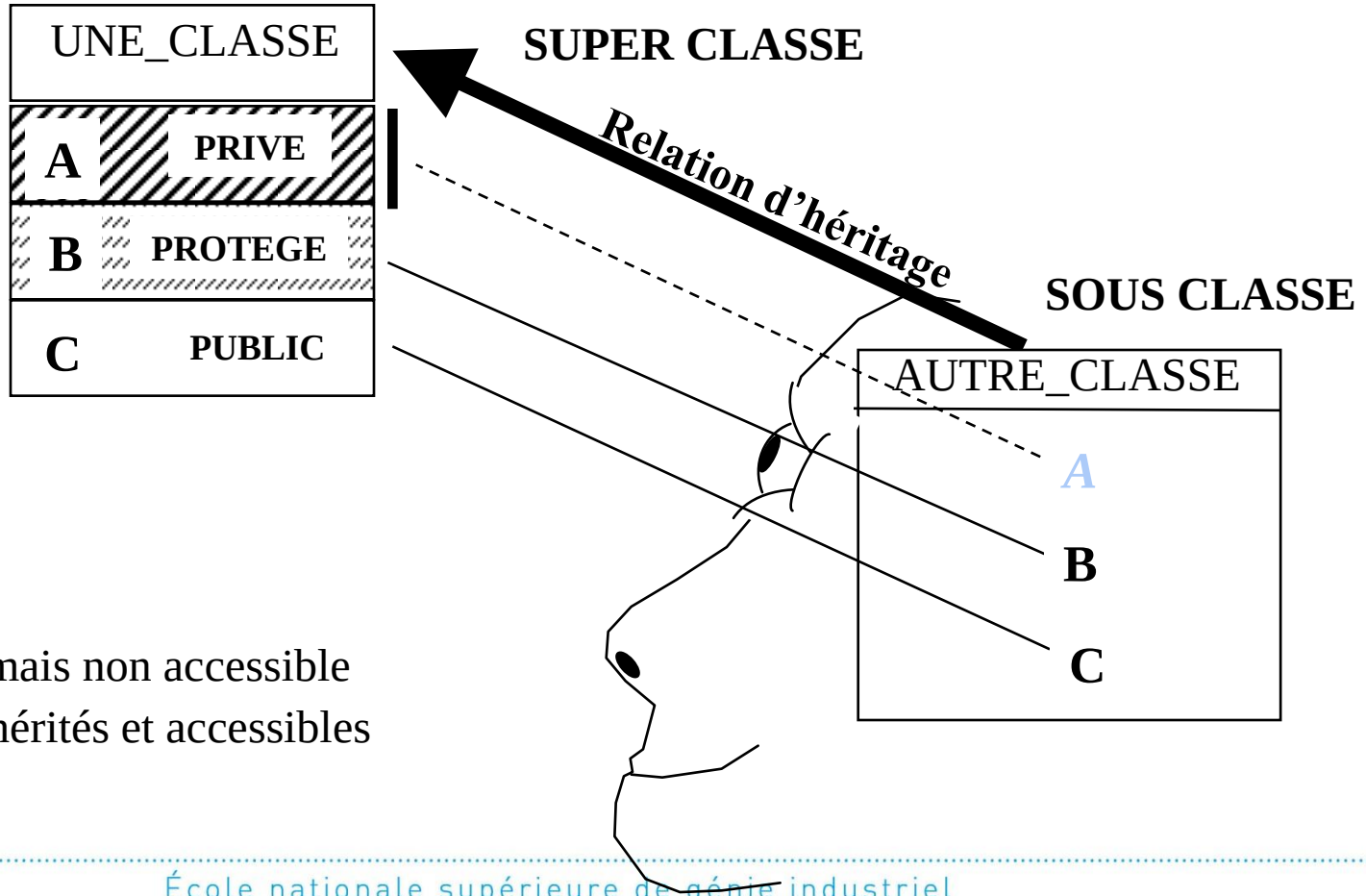
Dans **ACHAT** :

On ne peut pas connaître le prix HT car on ne peut pas utiliser PrixHT() de **ARTICLE**, mais on peut utiliser PrixTTC() qui est visible.

```
PrixTTC() :
        s := 0
        pour tout e dans Element
                s := s + e·PrixTTC()
        retourner s
```

VISIBILITE & HERITAGE

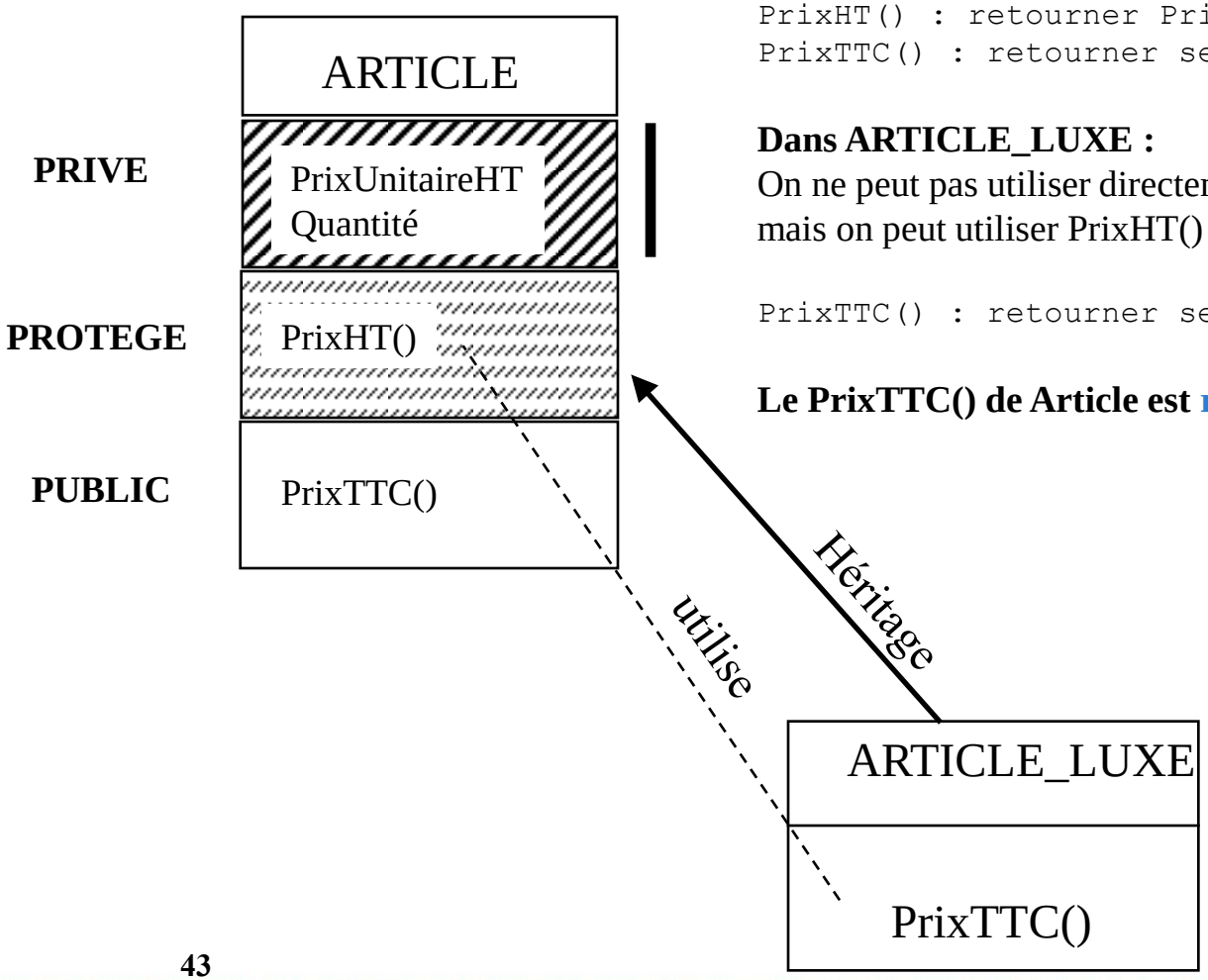
- Pour une super classe donnée : si l'héritage est **total**, tout est hérité, mais pas forcément **visible** et donc pas forcément directement accessible.



A est hérité mais non accessible

B et **C** sont hérités et accessibles

VISIBILITE & HERITAGE
Exemple



Dans ARTICLE :

```
PrixHT() : retourner PrixUnitaireHT*Quantité
PrixTTC() : retourner self.PrixHT()*1,206
```

Dans ARTICLE_LUXE :

On ne peut pas utiliser directement PrixUnitaireHT ni Quantité mais on peut utiliser PrixHT() qui est visible ici.

```
PrixTTC() : retourner self.PrixHT() * 1,330
```

Le PrixTTC() de Article est redéfini dans ARTICLE_LUXE

SEMANTIQUE DE L'HERITAGE

- **Structurel :** spécialisation de concepts

  Permet la définition incrémentale des classes

  La définition (attributs et méthodes) de **Article** est héritée et affinée dans **Article_Luxe**

- **Ensembliste :** inclusion ensembliste

  L'ensemble des instances de **Article_Luxe** est inclus dans l'ensemble des instances de **Article**.

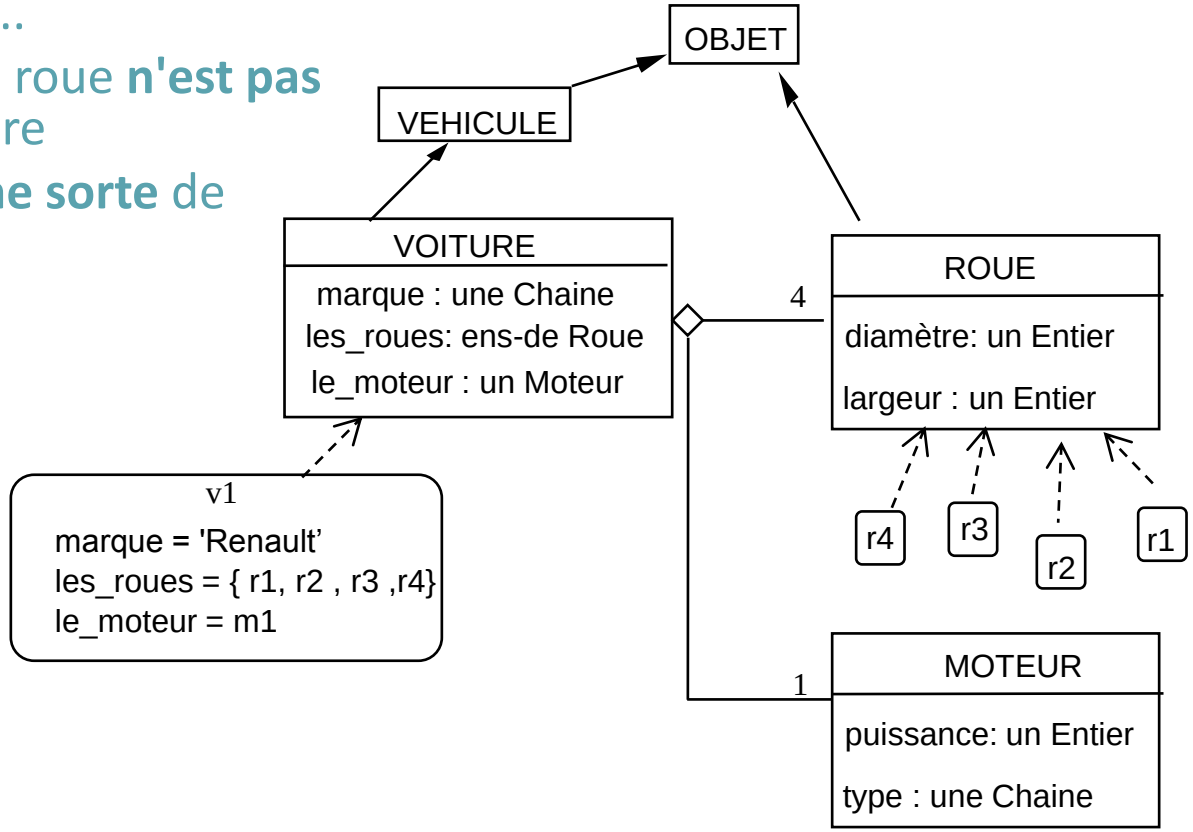
- NE PAS CONFONDRE AVEC LA COMPOSITION

SEMANTIQUE DE L'HERITAGE

- Héritage de propriétés : phénomène descendant (top-down)
  - les propriétés de la super- sont transmises à la sous-classe
  - super → sous
- Inclusion ensembliste : phénomène ascendant (bottom-up)
  - une instance d'une sous-classe est aussi instance de sa super-classe
  - sous → super

COMPOSITION V.S. SPECIALISATION

- **Composition** = agrégat, regroupement d'éléments
  - Une voiture **est composée** d'un moteur, de roues ...
  - Un moteur ou une roue **n'est pas une sorte** de voiture
  - Une voiture **est une sorte** de véhicule



HERITAGE : SUBTILITES

- Notion de méthode abstraite :

  La super-classe déclare une méthode mais ne la définit pas. Elle oblige toutes ses sous-classes à la définir.

- Exemple :

  la méthode cueillir() de la classe Fruits : on cueille tous les fruits, mais pas de la même manière, et il n'y a pas de manière générique.

- Notion de classe abstraite

- Notion de polymorphisme : fonctionnement différent d'une méthode selon l'objet manipulé (ex. la méthode PrixTTC() de la classe Article)

- Héritage multiple avec des conflits possibles : on verra plus tard.

HERITAGE : INTERETS

- L'intérêt c'est de définir les attributs et méthodes là où ils doivent l'être (approche par responsabilité) :
  - PrixTTC() est une méthode de tout article, utilisée pour calculer son prix. Par défaut elle affiche la valeur du prix du pointeur vers l'objet.
- La plupart des classes redéfinissent (surchargent) PrixTTC() pour lui faire calculer quelque chose de plus spécifique en fonction de la nature de l'article.
  - Mais un programme peut appeler toto.PrixTTC() quelque soit la classe de toto.
  - Lorsqu'il y a plusieurs définitions de PrixTTC() sur la branche de l'arbre d'héritage qui mène à la classe de toto, la définition du langage dit laquelle est appelée (typiquement la dernière), et comment appeler les autres si nécessaire.

SELECTION DE METHODES

- **Plusieurs méthodes de même nom**
  - La méthode surchargée est la **super-méthode** de la méthode qui la spécialise.
  - Exemple : prixTTC dans Article et Articleluxe
- **Sélection**
  - Lors d'un envoi de message, la méthode exécutée **dépend de la classe de l'objet receveur**
  - Dans les langages fortement typés (ex: C++), la méthode exécutée **dépend du type de la variable** qui désigne le receveur du message

    Exemple **:** `prix := y.prixTTC()`

    si y désigne une instance de **Article** ou **Electromenager** :

    on exécute "prixTTC" de **ARTICLE**

    si y désigne une instance de **Articleluxe** :

    on exécute "prixTTC" de **ARTICLELUXE**

Polymorphisme

- Utiliser la **même** méthode sur des objets différents
- Exemple :

```
class Animal {
        void deplacer() {
                System.out.println("Je bouge");
        }
}
class Chien extends Animal {
        void deplacer() {
                System.out.println("Je marche");
        }
}
class Oiseau extends Animal {
        void deplacer(){
                System.out.println("Je vole");
        }
}
class Pigeon extends Oiseau {
        void deplacer() {
                System.out.println("Je vole et en plus ... sur les passants");
        }
}
```

Polymorphisme

- Sur toutes ces classes, on peut donc appeler deplacer().
- Le polymorphisme permet alors d'appeler la bonne méthode selon le type d'objet.

```
public static void main(String[] args) {
        Animal a1 = new Animal();
        Animal a2 = new Chien();
        Animal a3 = new Pigeon();
        a1.deplacer();
        a2.deplacer();
        a3.deplacer();
}
```

A l'exécution ça donne:

```
Je bouge
Je marche
Je vole et en plus ... sur les passants
```

Polymorphisme

- Cela est particulièrement intéressant avec les conteneurs (Vector, List, etc.).
- Si l'on veut mettre des instances dedans (des chiens, des pigeons etc.) et que l'on veut qu'ils se déplacent tous.
- créer un conteneur d'animal :

```
List<Animal> lst=new ArrayList<Animal>();
```

- et appeler la méthode deplacer() sur chaque élément :

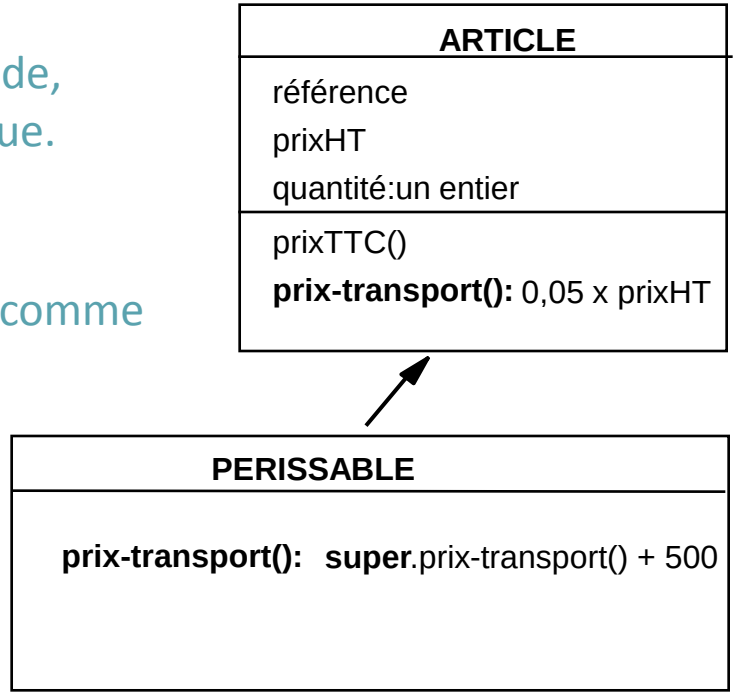
```
for(int i=0; i<lst.size(); i++) {
        // Appelle la méthode propre à chaque animal
        lst.get(i).deplacer();
}
```

Réutilisation du code des super-méthodes

- Une sous-méthode a souvent besoin de :
  - "récupérer" le code de sa super-méthode,
  - rajouter ensuite un traitement spécifique.

- **super**

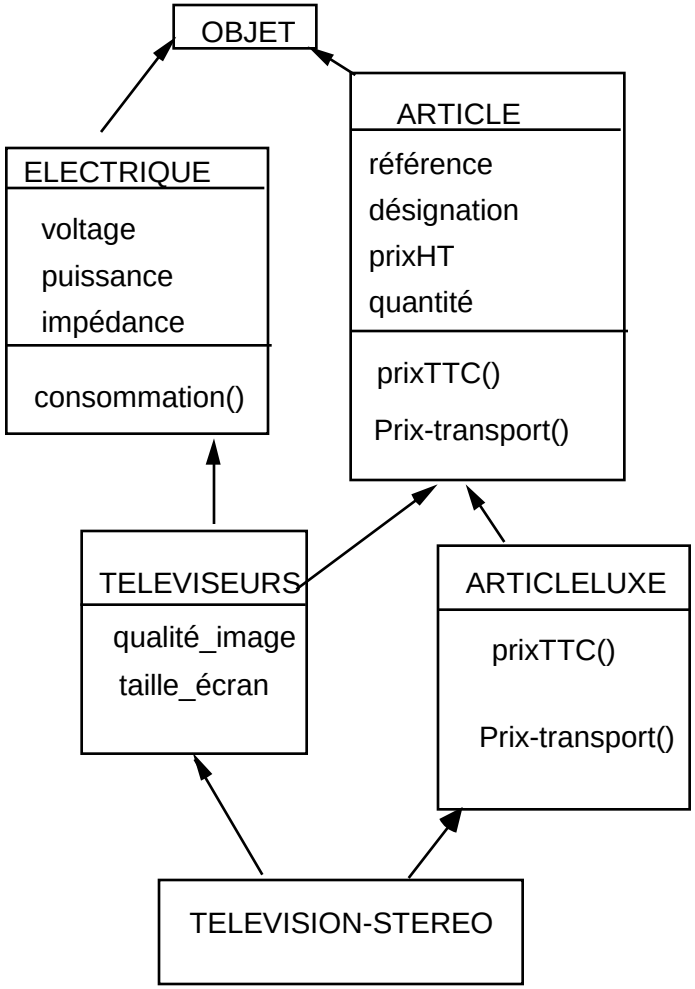
  une variable qui désigne l'objet receveur (comme **self**)



- Activation

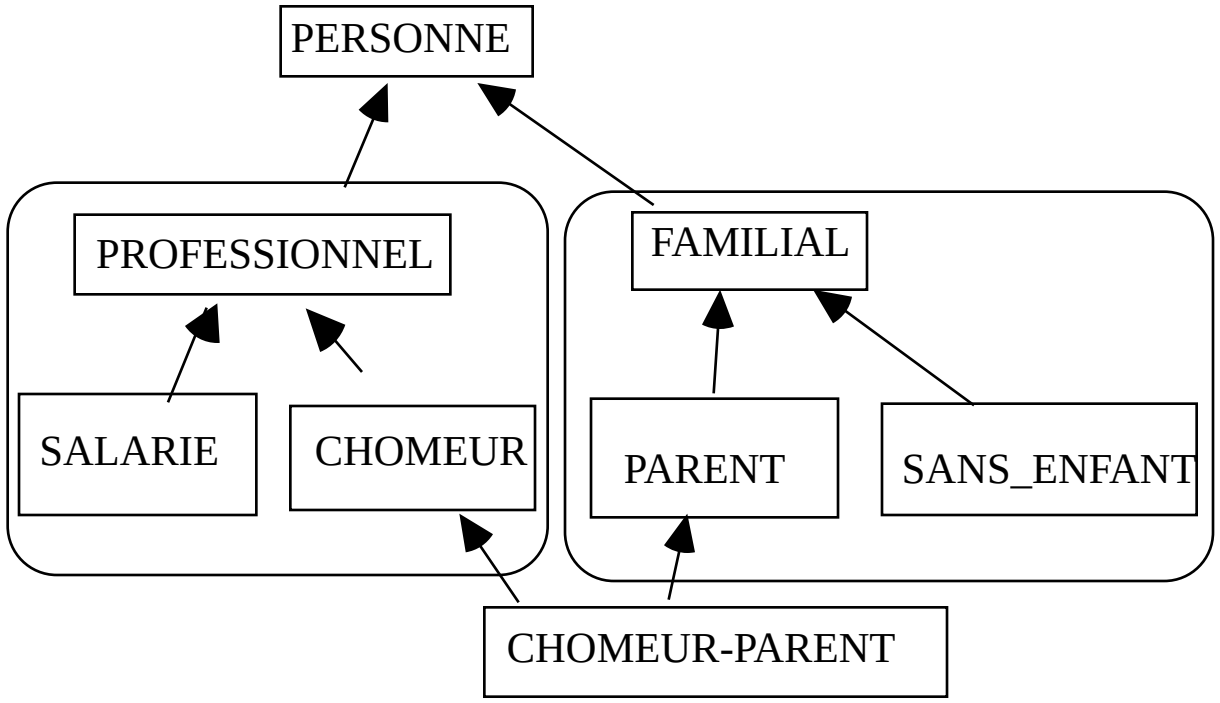
  la méthode "prix-transport" activée sera la première rencontrée dans la hiérarchie d'héritage de la super classe de Périssable.

HERITAGE MULTIPLE



- Graphe Orienté sans cycle :
  - encore plus de partage d'informations
  - un graphe plus complexe
  - des possibilités de conflits de noms

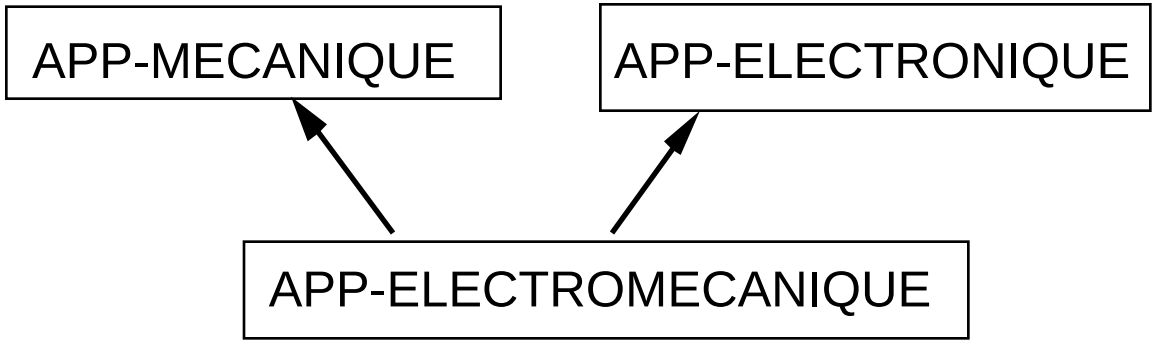
POINTS DE VUE PAR HERITAGE MULTIPLE



- Fusion de liens de spécialisation perçus comme deux points de vue différents d'un même objet.

POINTS DE VUE

Par héritage multiple:



- classification par inclusion
- héritage des propriétés

Par composition :

APP-ELECTROMECANIQUE

vuemécanique:APP-MECANIQUE

vueélectronique: APP-ELECTRONIQUE

- évite les problèmes de l'héritage multiple
- plus de classification par inclusion
- pas de préservation de l'identité d'objet

BIBLIOGRAPHIE

- M.-C. Gaudel, B. Marre, F. Schlienger, G. Bernot. *Précis de génie logiciel.* Masson 1996.
- J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy. *Object-Oriented Modeling and Design.* Prentice Hall, 1991.
- P.-A. Muller, N. Gaertner. *Modélisation objet avec UML. Deuxième édition.* Eyrolles, 2000.
- J. Rumbaugh, I. Jacobson, G. Booch. *Unified Modeling Language Reference Manuel.* Addison Wesley, 1999.
- E. Gamma, R. Helm, R. Johnson, J. Vlissides. *Design Patterns. Elements of Reusable Object-Oriented Software.* Addison Wesley. 1995.
- R´emy Fannader, Herv´e Leroux. *UML, Principes de modélisation.* Dunod, 1999.
- C. Larman. *UML et les Design Patterns.* Campus Press, 2002.
- Robert C. Martin. *Agile Software Development. Principles, Patterns and Practices.* Prentice Hall 2002.
- Robert C. Martin. *UML for Java Programmers.* Prentice Hall 2003.
- Sinan Si Alhir. *Introduction à UML.* O'Reilly, 2004.
- B. Meyer. *Conception et programmation orientées objet*. Eyrolles 2000.
- P. Roques. *UML par la pratique*. Eyrolles 2003.
- F. Buschman, R. Meunier, H. Rohnert, P. Sommerlad, M. Stal. *Pattern-Oriented Software Architecture. A System of Patterns.* Wiley, 1996.
- Unified Modeling Language Specification (1.5). OMG, 2003.
- Site sur UML : http://www.uml.org