



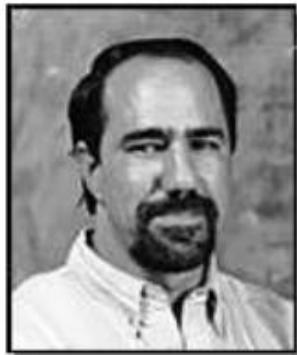
Modélisation UML

2

Ouassila Labbani-Narsis
ouassila.narsis@u-bourgogne.fr

Des méthodes de modélisation

- L'apparition du paradigme objet à permis la naissance de plusieurs méthodes de modélisation



Grady Booch
OOAD

Object Oriented
Analysis & Design



Ivar Jacobson
OOSE

Object Oriented
Software Engineering



James Rumbaugh
OMT

Object Modeling
Technique

- Chacune de ces méthodes fournie une notation graphique et des règles pour élaborer les modèles
- Certaines méthodes sont outillées

Trop de modèles...

- Entre 1989 et 1994 : le nombre de méthodes orientées objet est passé de 10 à plus de 50
- Toutes les méthodes avaient pourtant d'énormes points communs (objets, méthode, paramètres, ...)
- Au milieu des années 90, G. Booch, I. Jacobson et J. Rumbaugh ont chacun commencé à adopter les idées des autres. Les 3 auteurs ont souhaité créer un **langage de modélisation unifié**



Naissance d'UML...

Mai 2015

Standardisation OMG

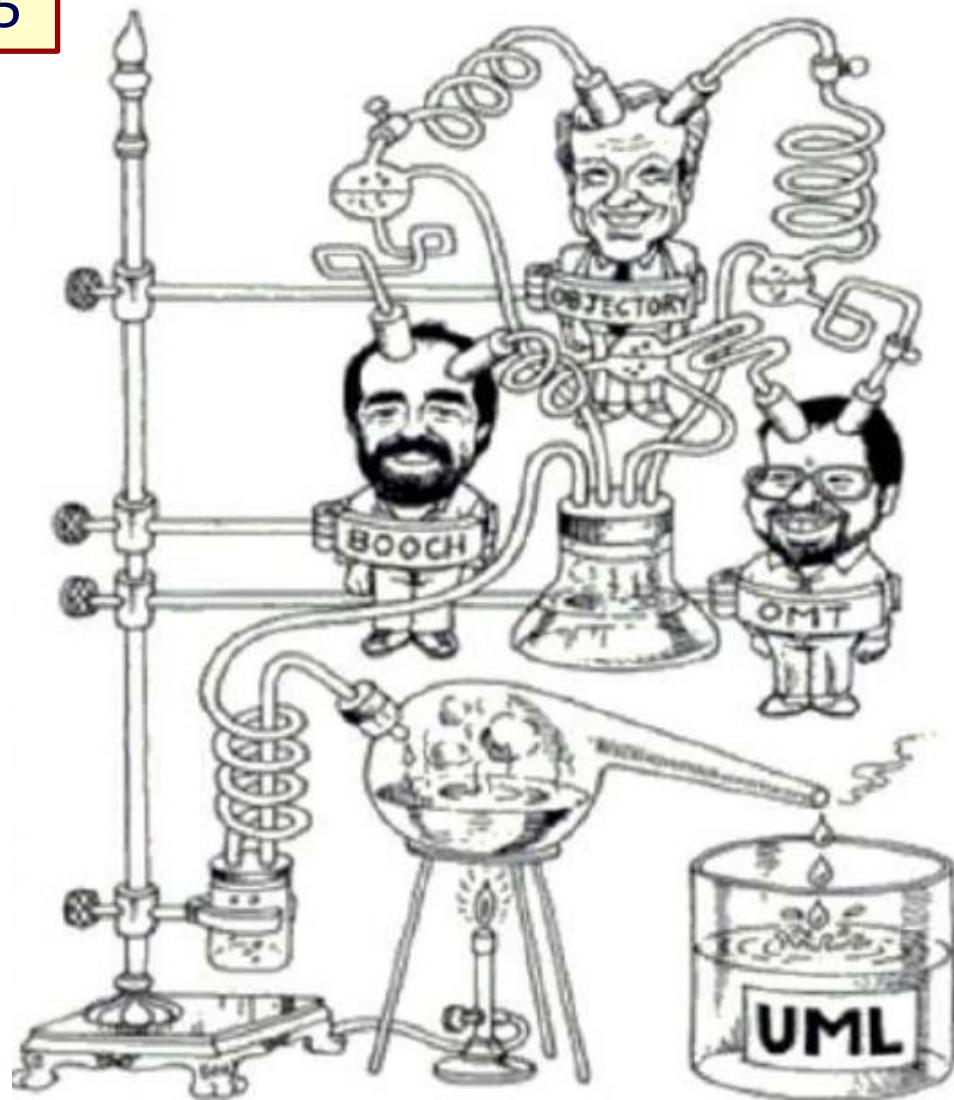
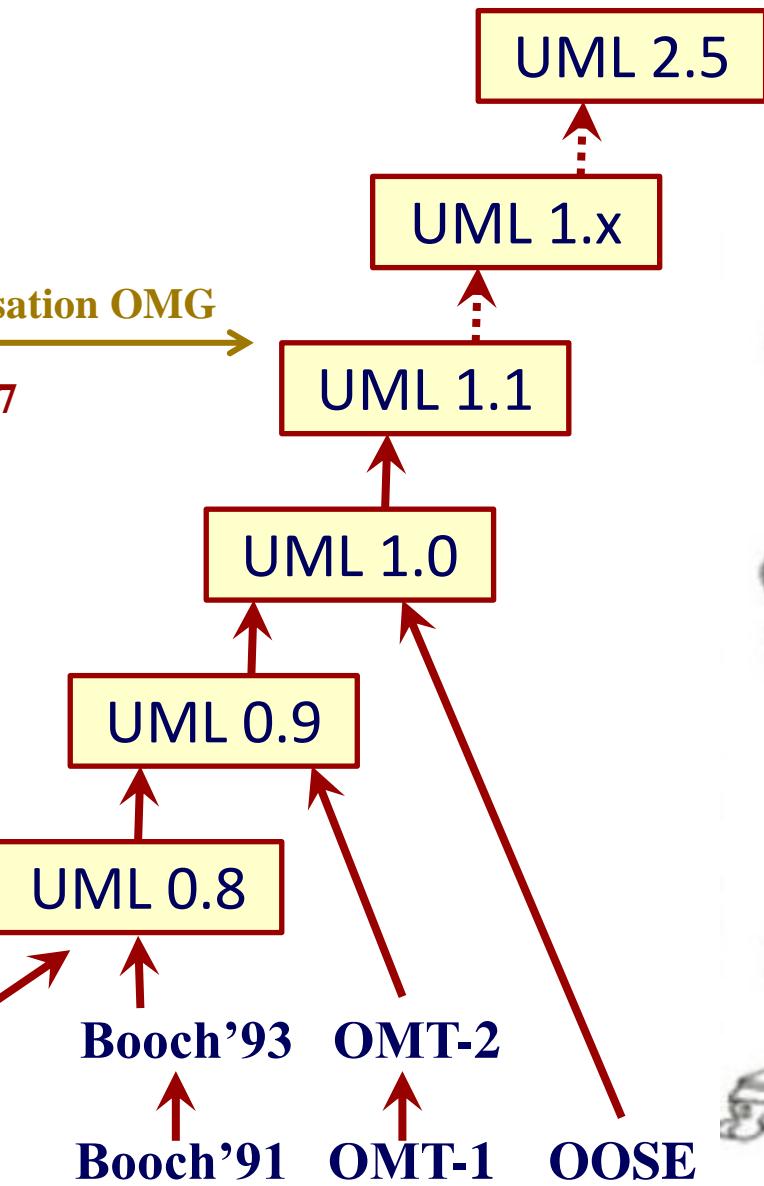
Novembre 97

Janvier 97

Juillet 96

Octobre 95

Autres méthodes



UML : Unified Modeling Language

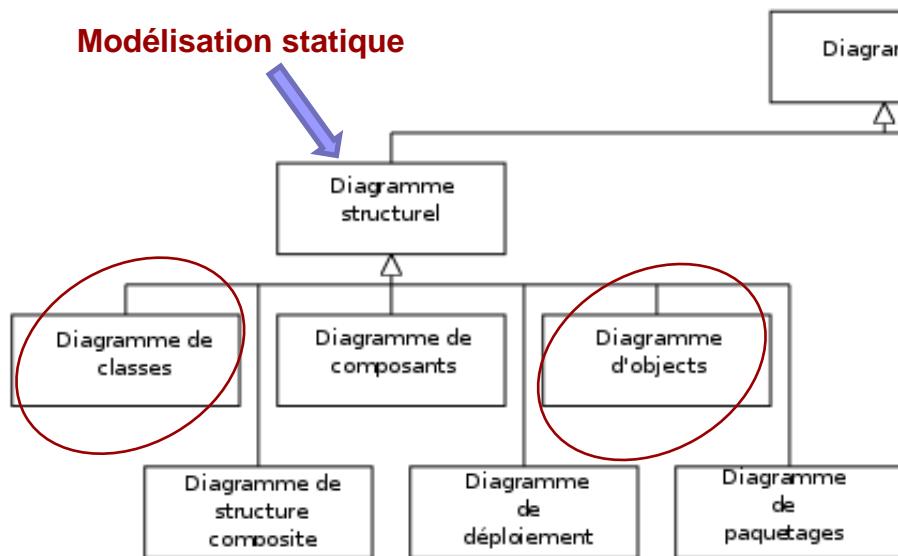
- UML est le langage visuel de modélisation orienté objet
- C'est un langage **semi-formel**
- Une norme OMG
- UML n'est pas une méthode
- Peu d'utilisateurs connaissent le standard, ils ont une vision outillée d'UML (Vision Utilisateur)
- Le marché UML est important et s'accroît
 - MDA, MDE, UML2.0, IBM a racheté Rational Rose, ...



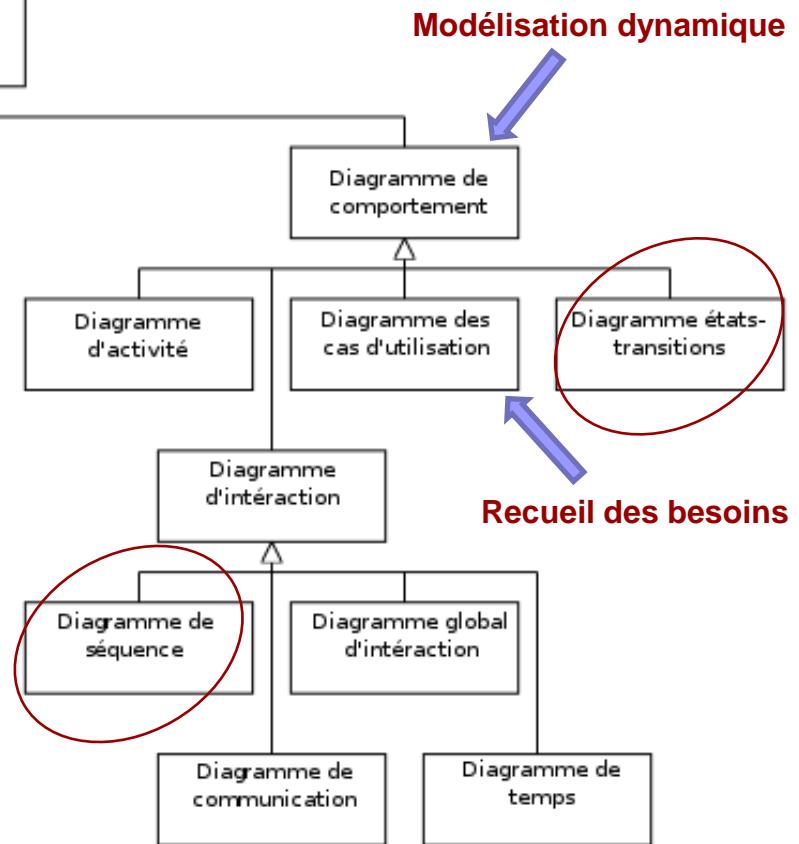
UML est au cœur de l'approche IDM

Un problème - Un diagramme

Modélisation statique



Modélisation dynamique



Recueil des besoins

1. Diagramme de Cas d'Utilisation

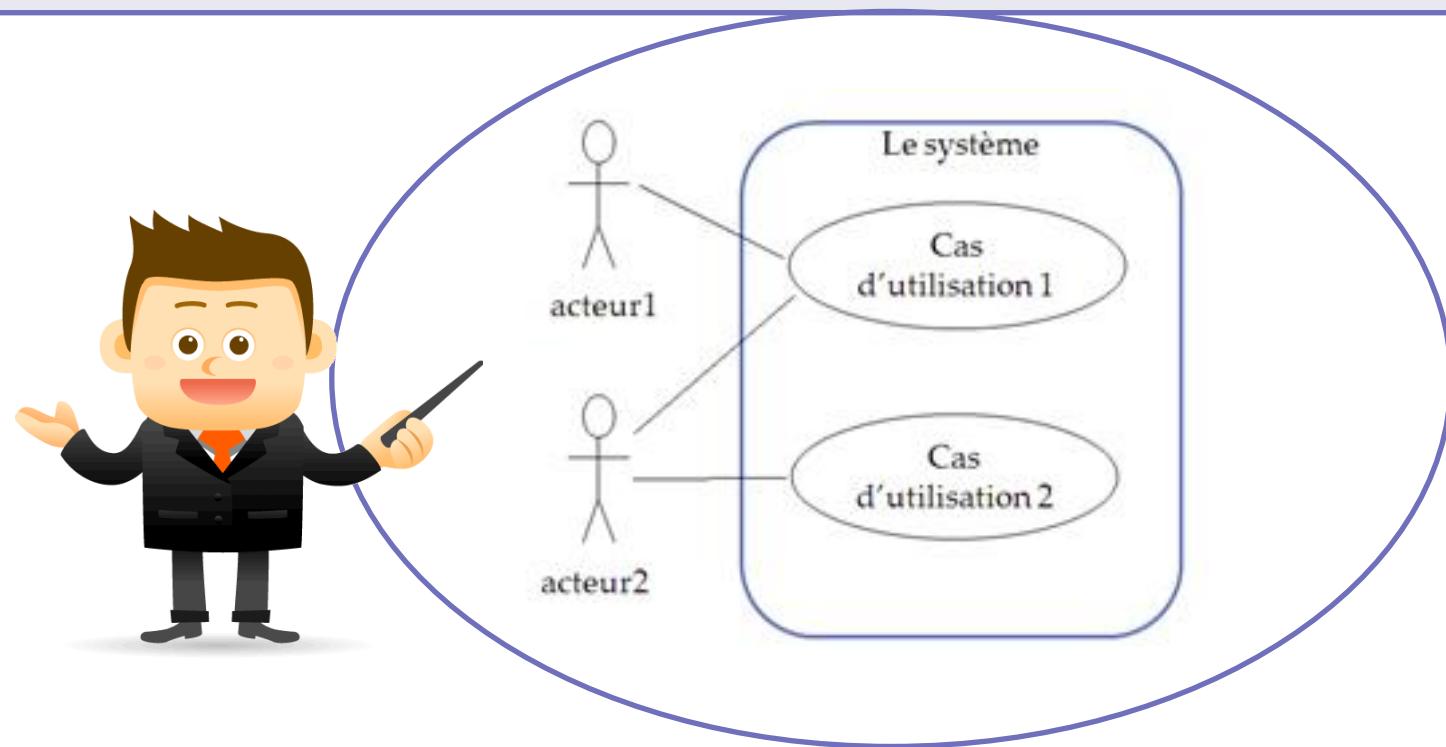


Diagramme de Cas d'Utilisation

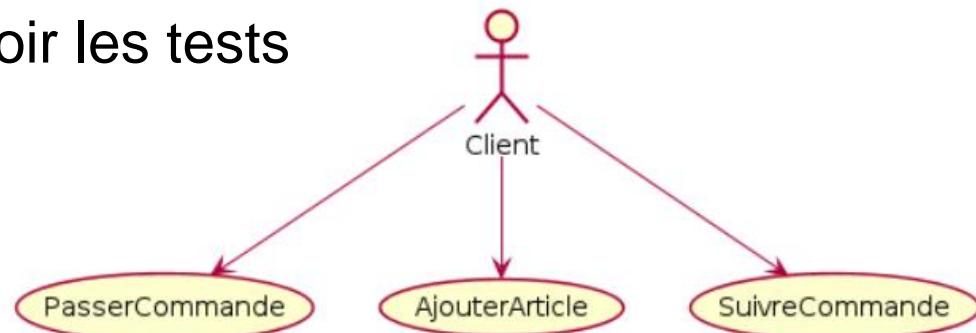
Pourquoi les cas d'utilisation ?

- Répondre à la question : “*A quoi va servir le logiciel ?*”
- Un système est conçu pour les utilisateurs :
 - ils savent ce que le système doit faire mais pas comment le faire
 - ils connaissent l'aspect fonctionnelle du système
- Les utilisateurs du système sont les plus aptes à décrire comment ils s'en servent

Le système doit donc être bâti à partir des descriptions des utilisateurs

Diagramme de Cas d'Utilisation

- Le diagramme des cas d'utilisation est modélisé par :
 - Des acteurs qui représentent les entités externes du système
 - Des cas d'utilisation qui représentent les fonctionnalités attendues du système
- Ses objectifs sont :
 - Capturer les besoins fonctionnels du système
 - Délimiter le système
 - Visualiser le cahier des charges graphiquement
 - Peut servir à concevoir les tests



Identification des acteurs

Comment ?

- Par un dialogue avec le client et les utilisateurs
- En délimitant les frontières du système



Qui sont-ils ?

- Les principaux acteurs sont les utilisateurs du système
- En plus des utilisateurs, les acteurs peuvent être :
 - des logiciels déjà disponibles à intégrer dans le projet;
 - des systèmes informatiques externes au système mais qui interagissent avec lui;
 - tout élément *extérieur* au système et avec lequel il interagit

Identification des acteurs

Rôles et personnes physiques

Un acteur correspond à un **rôle**, pas à une personne physique :

- Une même personne physique peut être représentée par plusieurs acteurs si elle a plusieurs rôles
- Si plusieurs personnes jouent le même rôle vis-à-vis du système, elles seront représentées par un seul acteur

Exemples :

- Pierre, Paul ou Jacques sont tous les Clients du point de vue d'un distributeur automatique
- En tant que Webmestre, Paul a certains droits sur son site, mais il peut également s'identifier en tant qu'Utilisateur



Identification des Cas d'Utilisation

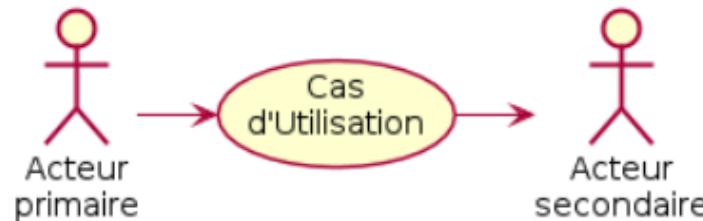
- Un cas d'utilisation est un service rendu à un acteur : c'est une fonctionnalité de son point de vue
- Pour découvrir les uses case d'un système, il faut répondre aux questions suivantes :
 - Quels sont les services rendus par le système ?
 - Quelles sont les interactions Acteurs/ système ?
 - Pour chaque acteur identifié
 - Rechercher les différentes intentions métier avec lesquelles il utilise le système
 - Déterminer les services fonctionnels attendus du système par cet acteur
 - Quels sont les évènements perçus par le système (externes, temporels, changement d'état) ?

Cas
d'Utilisation

Relations liant les acteurs

Associations entre cas et acteurs

- Les acteurs demandant des services aux systèmes, ils sont le plus souvent à l'initiative des échanges avec le système :
 - ils sont dits **acteurs primaires**
- Lorsqu'ils sont sollicités par le système (dans le cas de serveurs externes par exemple), ils sont dits **acteurs secondaires**

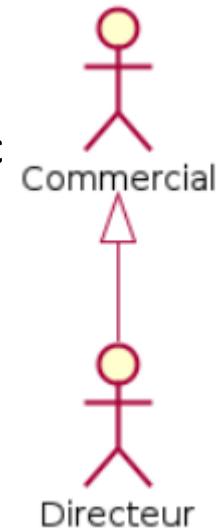


- On représente une association entre un acteur et un cas d'utilisation par une **ligne pleine**
- Un acteur est souvent associé à plusieurs cas d'utilisation

Relations liant les acteurs

Relations entre acteurs

- Il n'y a qu'un seul type de relation possible entre acteurs : la relation de ***généralisation***
- Il y a généralisation entre un cas A et un cas B lorsqu'on peut dire : *A est une sorte de B*
- Exemple :
 - Un directeur *est une sorte de* commercial : il peut faire avec le système tout ce que peut faire un commercial, plus d'autres activités liées à la fonction de directeur



Relations entre cas d'utilisation

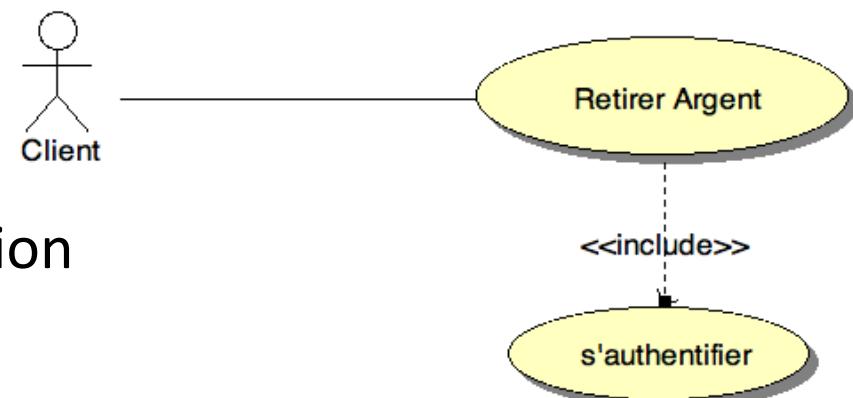
Types de relations possibles

- Inclusion (<<include>>) : le cas d'utilisation source (départ de la flèche) incorpore directement et nécessairement (contient TOUJOURS) le cas d'utilisation cible à un endroit précis dans son enchaînement. Le cas inclus n'est jamais exécuté tout seul



B est une partie obligatoire de A et on lit A *inclus B* (dans le sens de la flèche).

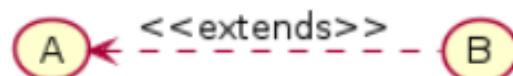
Dans une relation «include», le cas d'utilisation source est donc celui qui a besoin d'un autre cas d'utilisation dit interne, indiqué par une flèche



Relations entre cas d'utilisation

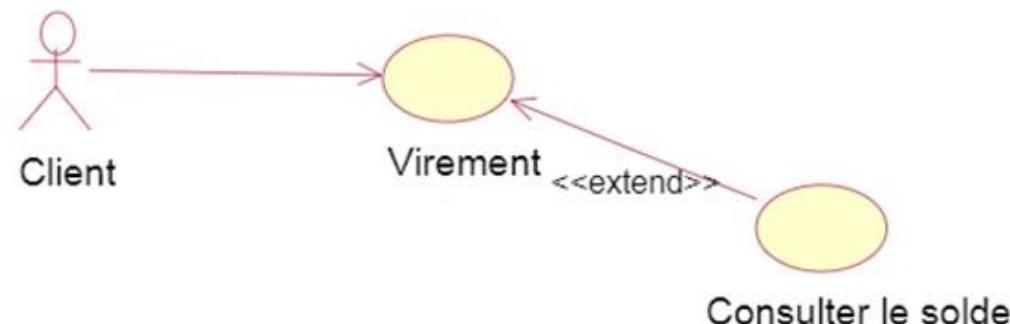
Types de relations possibles

- Extension (<<extend>>) : le cas d'utilisation de base (arrivé de la flèche) en incorpore indirectement et pas nécessairement un autre cas d'utilisation à un endroit précis dans son enchaînement. Le cas de base peut être exécuté tout seul



B est une partie optionnelle de A et on lit B étend A (dans le sens de la flèche)

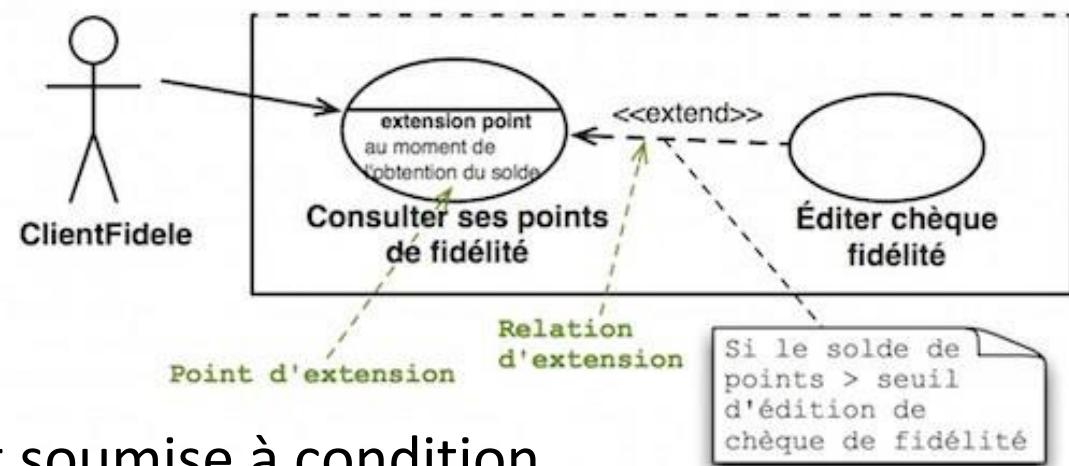
Cette relation est utilisée pour indiquer que le cas d'utilisation source (à l'origine de la flèche) **n'est pas toujours nécessaire** au cas d'utilisation principal



Relations entre cas d'utilisation

Types de relations possibles

- La relation **<<extend>>** peut devenir nécessaire au cas d'utilisation principal dans certaines situations. On doit alors préciser la condition qui fera que le cas d'utilisation en «extend» est nécessaire
- La relation **<<extend>>** peut intervenir à un point précis du cas étendu. Ce point s'appelle le **point d'extension**, et est éventuellement associé à une contrainte indiquant le moment où l'extension intervient.



Une extension est souvent soumise à condition.

Relations entre cas d'utilisation

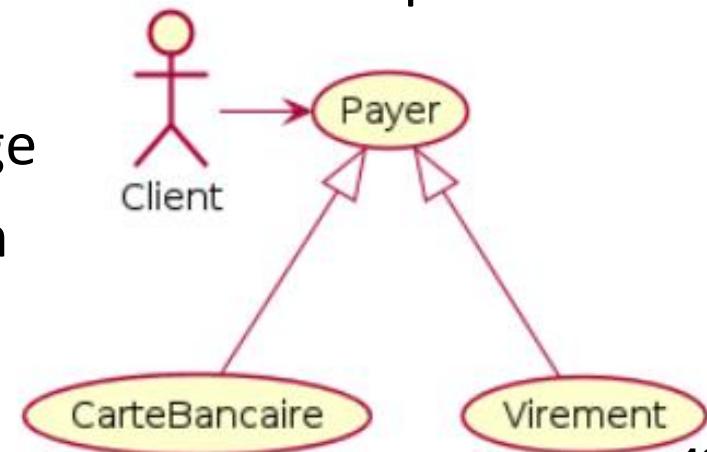
Types de relations possibles

- Généralisation (héritage) : le cas d'utilisation fils spécialise le cas d'utilisation père



le cas A est une généralisation du cas du cas B et on lit *B est une sorte de A*

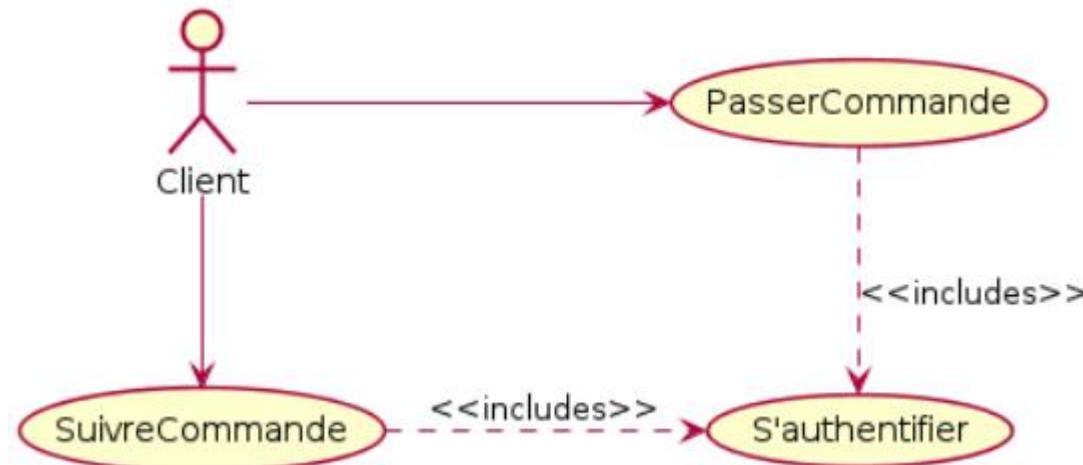
- Cette relation de généralisation/spécialisation est présente dans la plupart des diagrammes UML et se traduit par le concept d'héritage dans les langages de programmation orientés objet



Relations entre cas d'utilisation

Réutilisation de cas d'utilisation

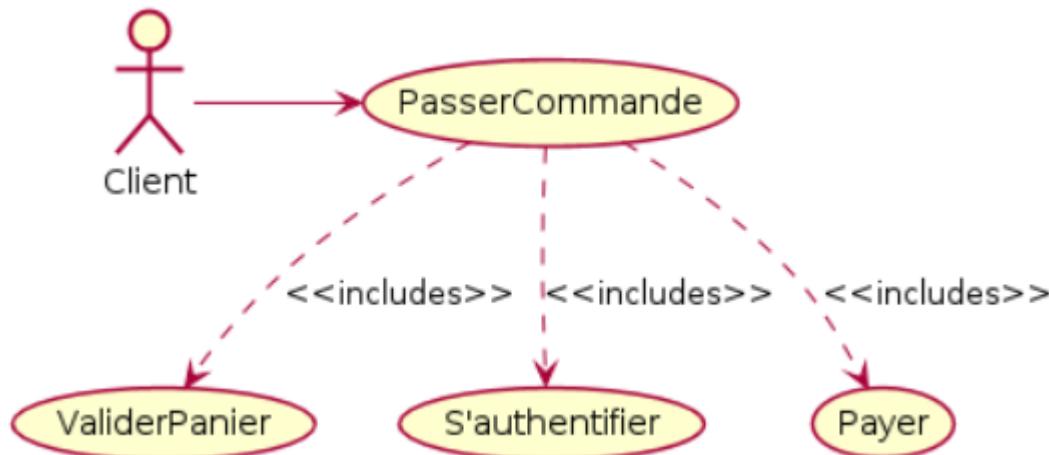
- Les relations d'inclusion et d'extension permettent d'isoler un service réutilisable comme partie de plusieurs autres cas d'utilisation. On parle alors de **réutilisation**
- Le code développé pour implémenter le cas d'utilisation réutilisé est d'emblée identifié comme ne devant être développé qu'une seule fois, puis réutilisé



Relations entre cas d'utilisation

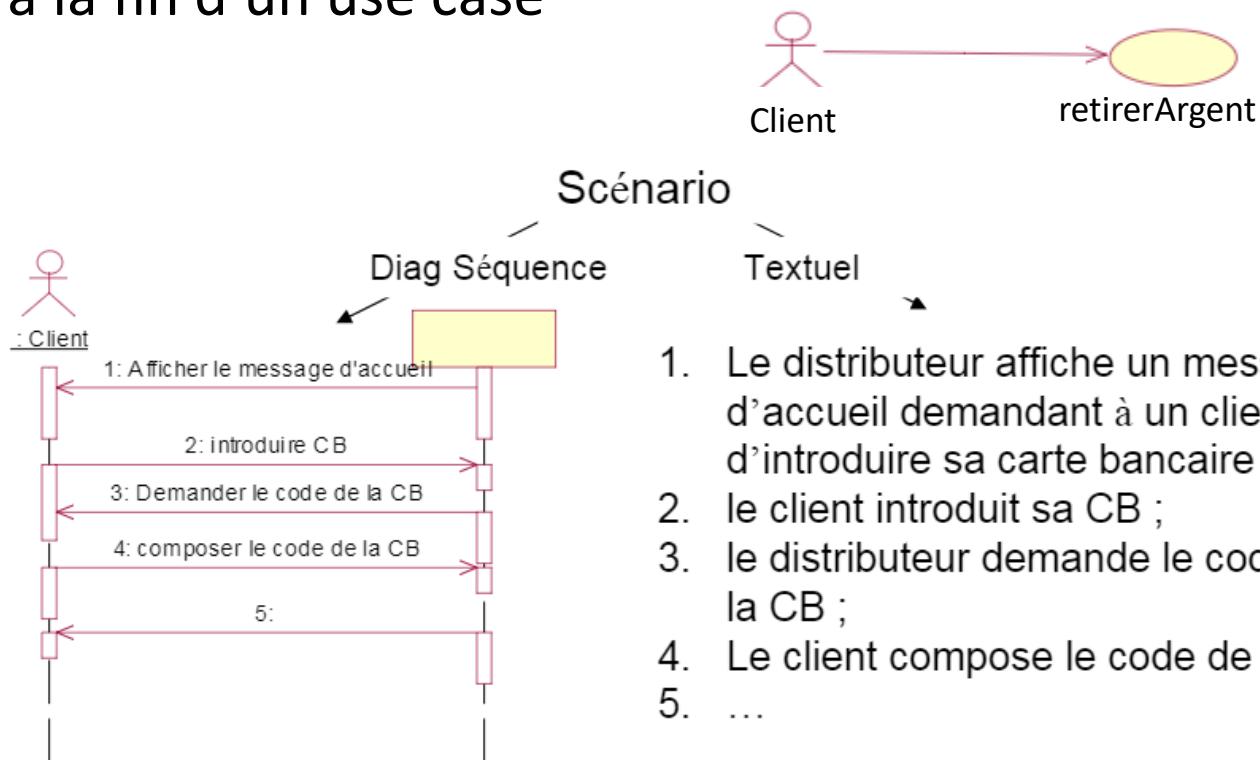
Décomposition de cas d'utilisation

- Un cas d'utilisation ne doit jamais se réduire à une seule action : il doit occasionner des traitements d'une complexité minimale
- Toutefois, il peut arriver qu'un cas d'utilisation recouvre un ensemble très important d'échanges et de traitements. Dans ce cas, on peut utiliser les relations d'inclusion et d'extension



Description des cas d'utilisation

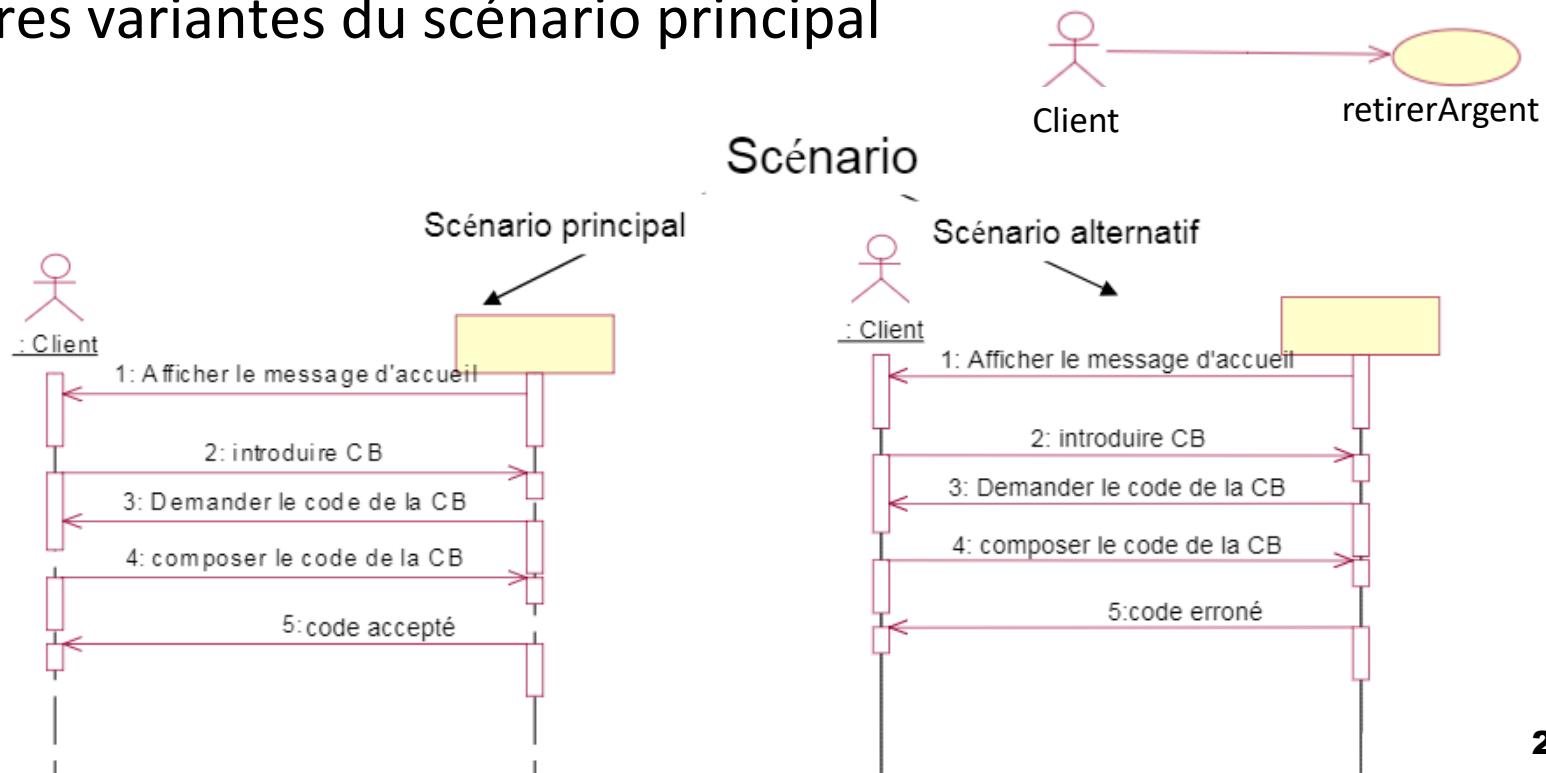
- Un nom ne suffit pas à comprendre le détail de ce que recouvre un cas d'utilisation
- Les cas d'utilisation peuvent être décrits sous la forme de scénarios
- Un scénario représente une séquence d'actions qui s'exécute du début à la fin d'un use case



Description des cas d'utilisation

Un même use case peut être décrit par plusieurs scénarii:

- Un scénario principal (*nominal*): le service est accompli sans exceptions ou erreurs
- Plusieurs scénario alternatifs: cas des exceptions, des erreurs, autres variantes du scénario principal

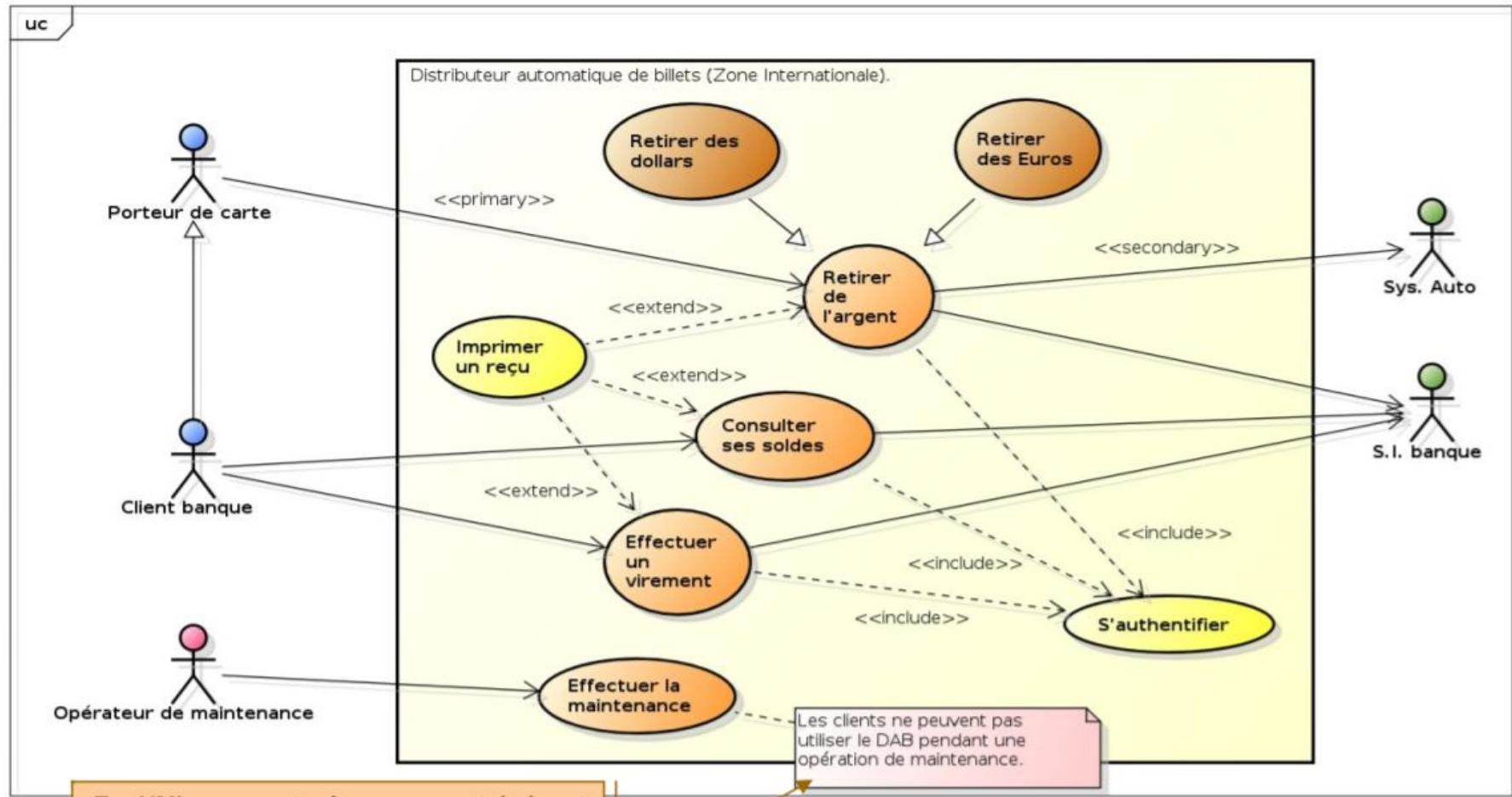


Synthèse

- Les diagrammes de cas d'utilisation sont souvent employés
 - Ils permettent de décrire le système de façon très abstraite
 - Ils offrent une vue fonctionnelle (par opposition à une vue Orienté Objet)
 - Ils sont très simples
- Les cas d'utilisation se limitent à décrire le « **quoi** » d'un système mais pas le « **comment** »
- Les cas d'utilisation sont une description fonctionnelle d'un système après quoi il faut passer à une description objet (les scénarii) en utilisant :
 - les diagrammes de séquences ou les diagrammes d'activités comme alternative
 - les diagrammes de collaboration

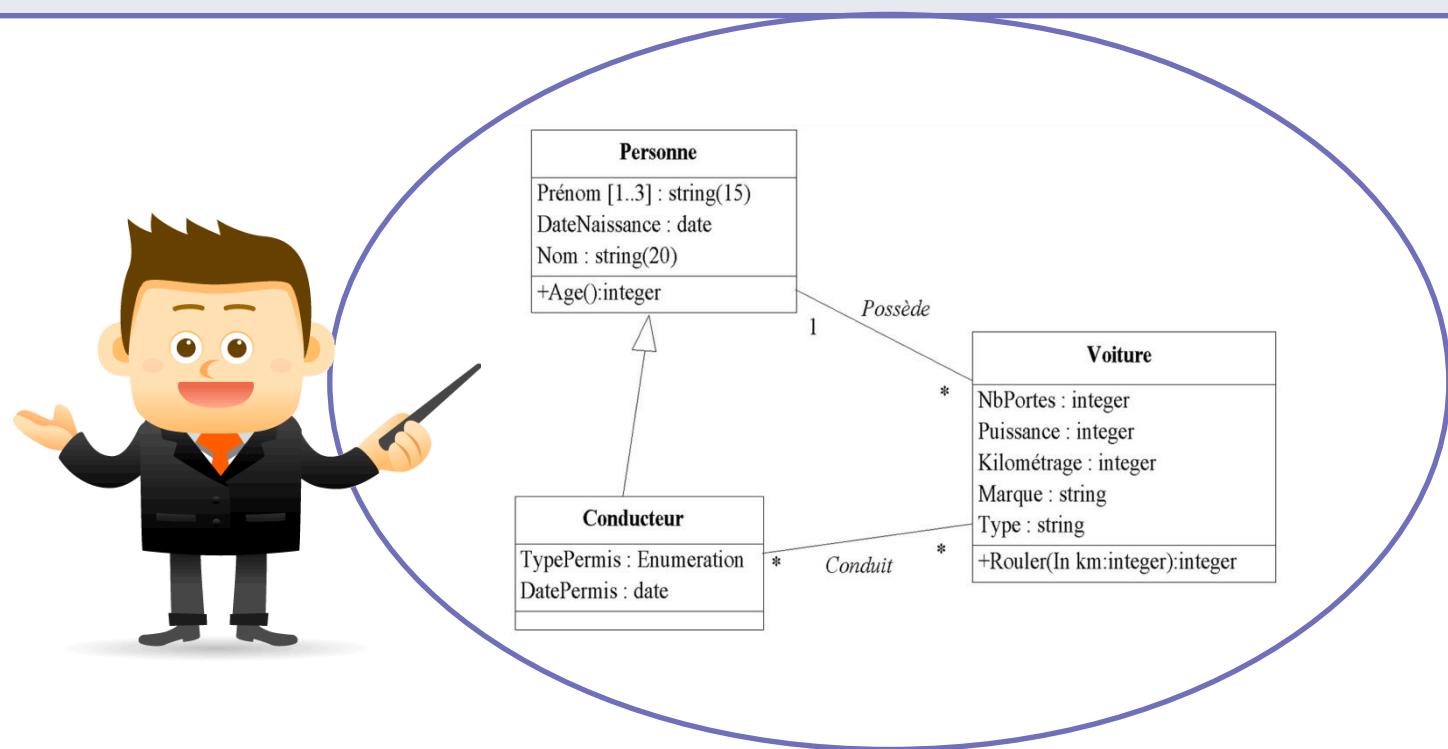
Synthèse

Exemple d'un diagramme de cas d'utilisation



En UML une note (un commentaire) est représentée par un rectangle dont l'un des coins est retourné. La note est reliée à l'élément ou aux éléments qu'elle décrit par une ou plusieurs lignes pointillées.

2. Diagramme de classes



Modélisation Orientée-Objets

- La conception d'un système est réalisée autour des **données**
- Une réflexion autour des données conduit à :
 - Déterminer les données à manipuler
 - Réaliser, pour chaque type de données, les fonctions qui permettent de les manipuler
- On parle alors d'**OBJETS**
 - un bâtiment, un mur, une pièce, ...
 - un panneau de circulation, une voiture, un bus, ...
 - un patient, un médecin, un médicament, une image médicale, ...

Principes de base de la POO

Un **objet** est une association de **données** et des **fonctions** (méthodes) opérant sur ces données



La POO propose un ensemble de concepts indépendants de tout langage de programmation facilitant la manipulation et la représentation des objets

Principes de base de la POO

Un objet est caractérisé par :

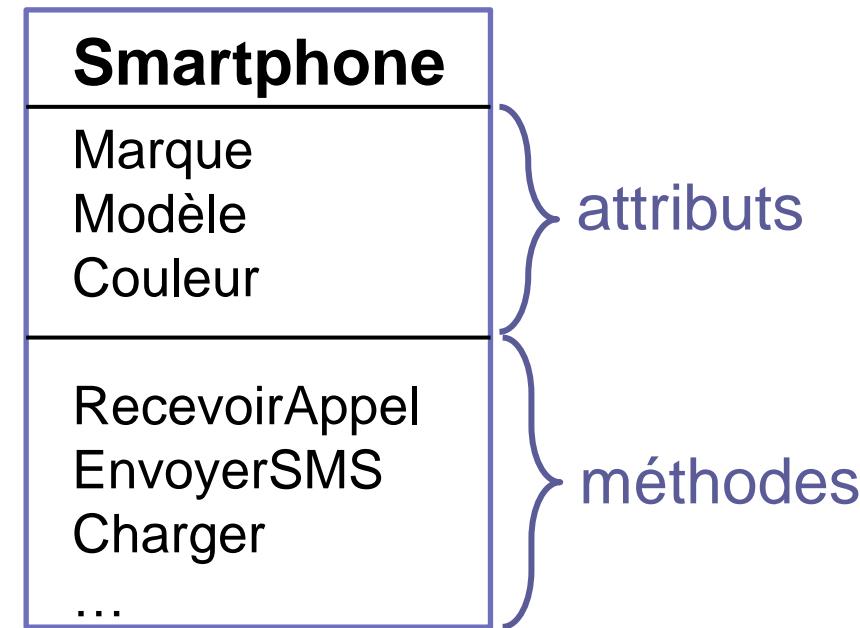
- Un **état** : ensemble de valeurs des données (valeur de ses attributs)
- Un **comportement** : ensemble d'opérations réalisables

Chaque objet :

- **Connait** ses caractéristiques
- Sait comment **effectuer les opérations** qui le concernent
- **Réagit à des messages** lui demandant d'effectuer un certain traitement

Principes de base de la POO

- Une **donnée** stockée dans un objet est un **attribut**
 - Les valeurs des attributs définissent l'état de l'objet
- Une **méthode** permet d'effectuer une **action**
 - Obtenir une information sur l'objet ou lui donner un ordre



Principes de base de la POO

Concept de classe

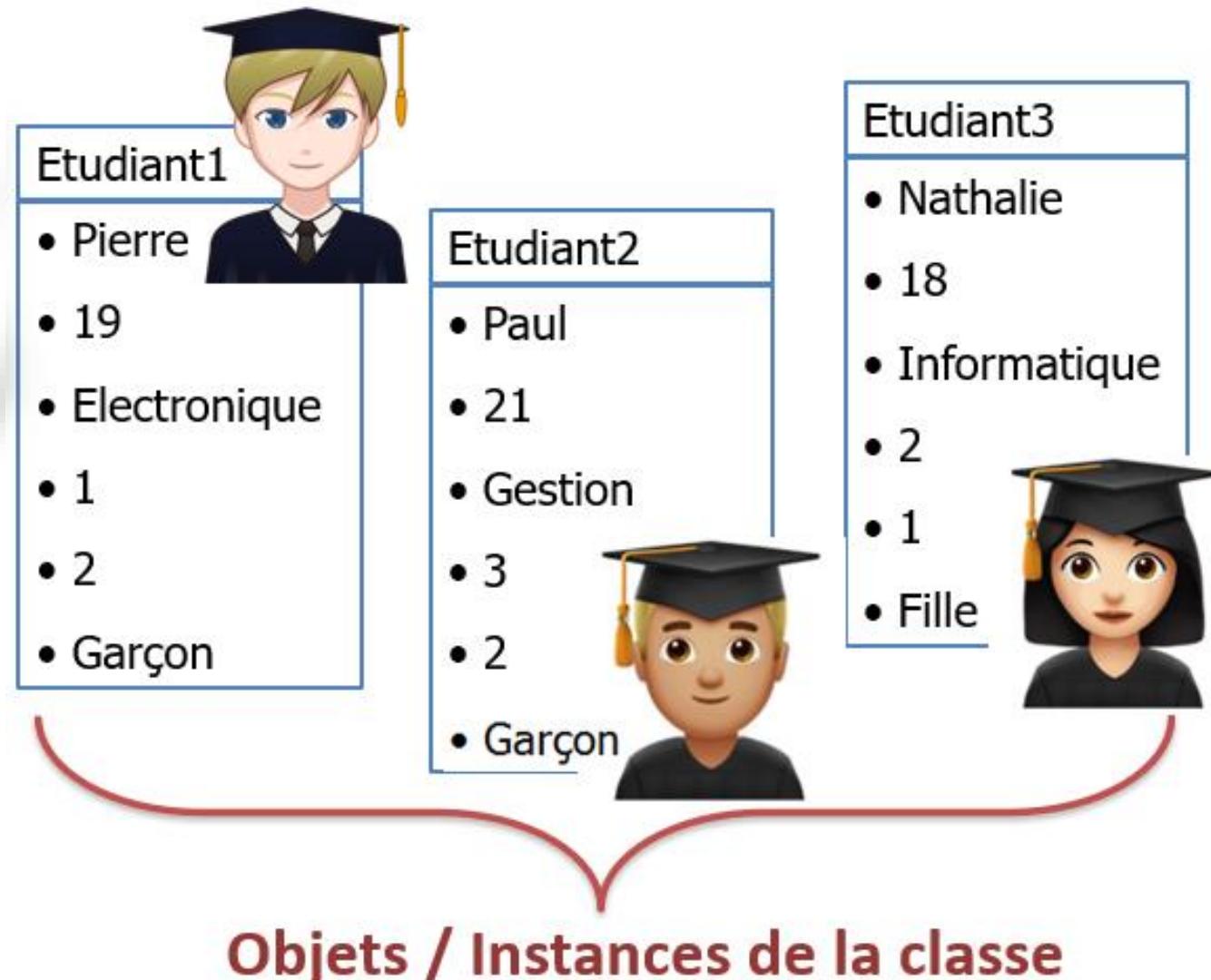
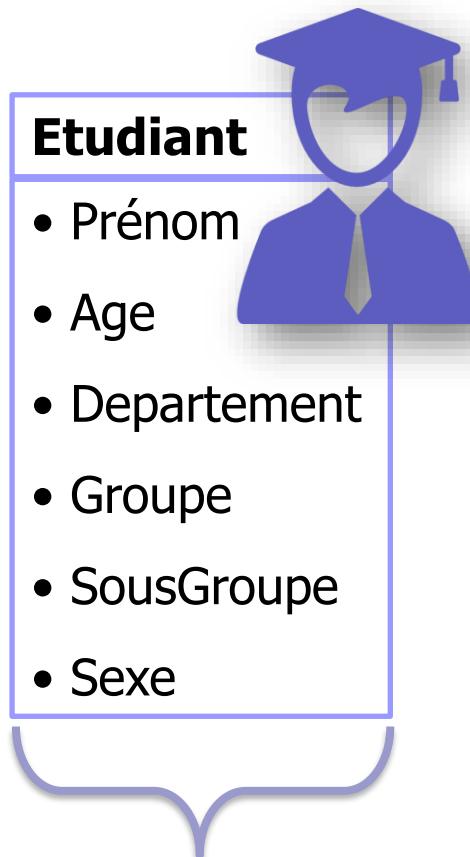
Les objets ayant le même comportement forment un même ensemble, appelé **CLASSE**. Cette notion n'est autre que la généralisation de la notion de type

Un objet d'une classe s'appelle **INSTANCE** de cette classe

Une classe est décrite par :

- **Le type des données représentées** (déclaration des attributs)
- **Le prototype (ou interface) des méthodes** accédant aux attributs

Principes de base de la POO



Principes de base de la POO

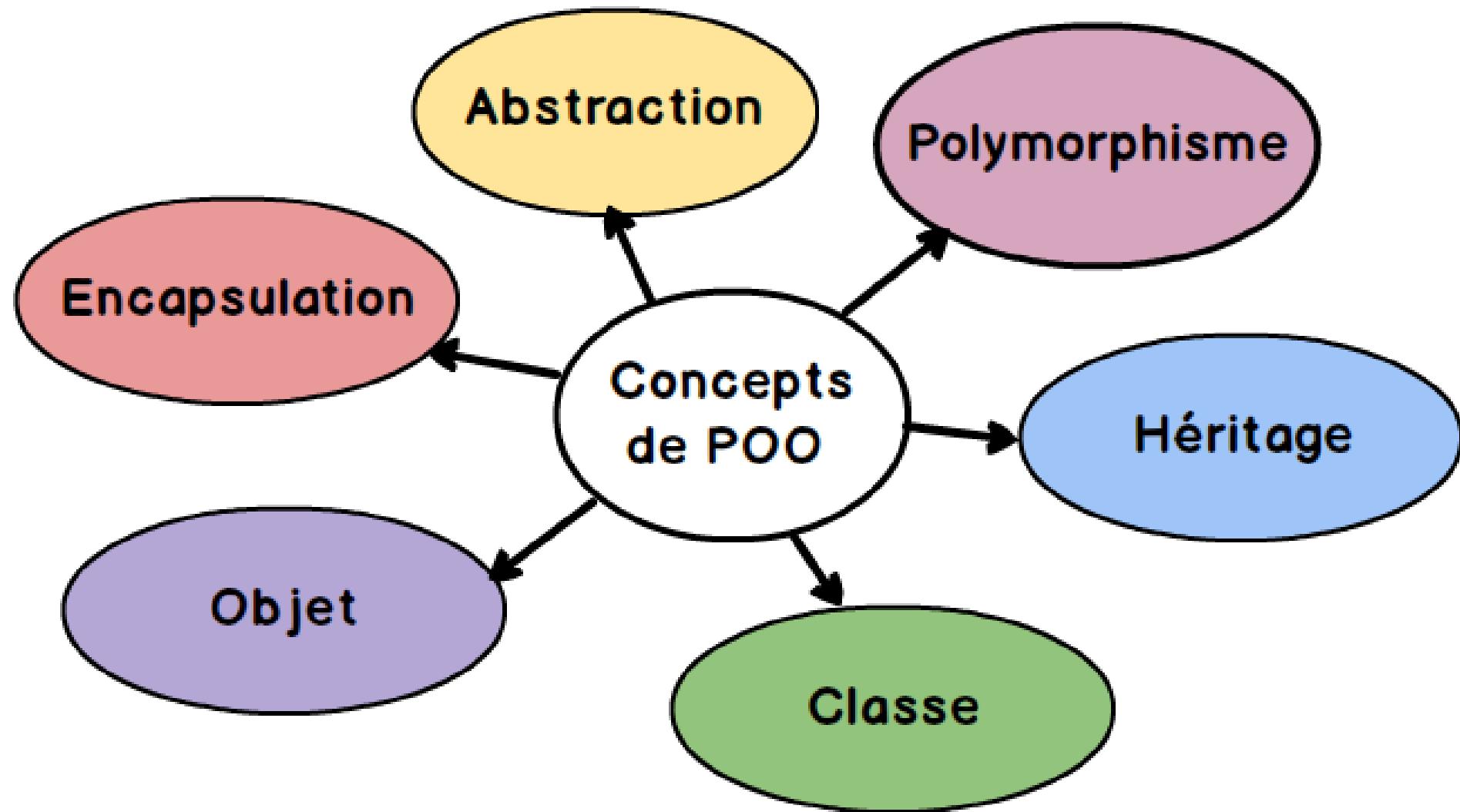
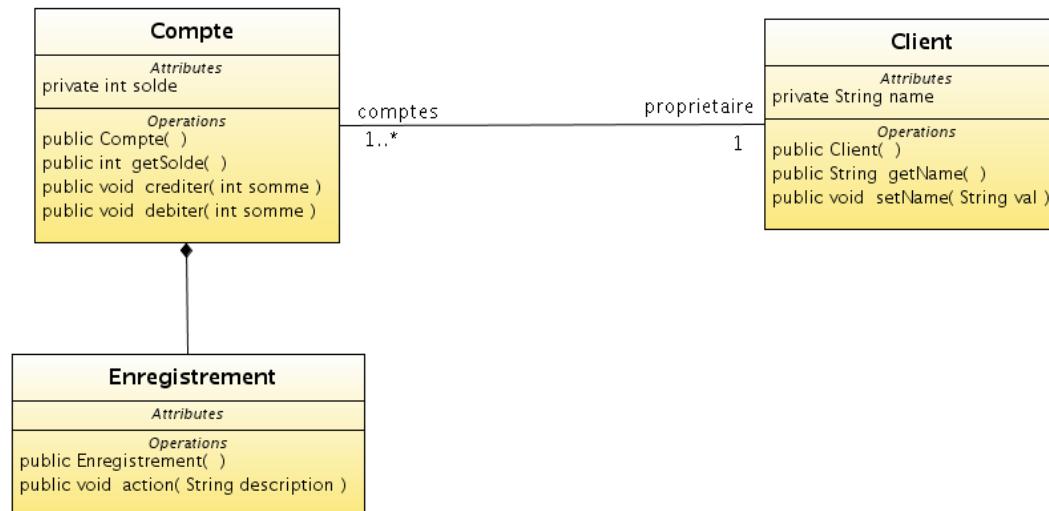


Diagramme de Classes

- Un diagramme de classe est un graphe d'éléments connectés par des relations
- Il représente une vue graphique de la **structure statique** d'un système (*diagramme structurel*)
- Le diagramme de classes permet de modéliser les classes du système et leurs relations indépendamment d'un langage de programmation particulier



Classes

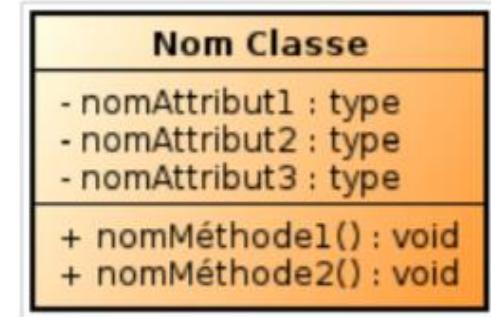
Rappel sur les classes

- Une classe représente la **structure commune** et la représentation **abstraite** d'un ensemble d'objets
- Elle contient les informations nécessaires à la construction de l'objet (*attributs* et *méthodes*)
- La classe peut donc être considérée comme le **modèle**, le **moule** ou la **notice** qui va permettre la construction d'un objet. Nous pouvons encore parler de type (comme pour une donnée). On dit également qu'un objet est *l'instance d'une classe* (la concrétisation d'une classe)

Classes

Représentation des classes

- Une classe est représentée par un rectangle qui contient une chaîne de caractères correspondant au nom de la classe
- Ce rectangle peut être séparé en **trois parties** :
 - La première partie contient le **nom** de la classe qui :
 - représente le type d'objet instancié
 - débute par une lettre majuscule
 - ne pas contenir le caractère ‘::’
 - il est en italique si la classe est abstraite (IMPOSSIBLE d'instancier un objet)
 - La deuxième partie contient les **attributs**
 - La troisième partie contient les **méthodes/opérations**



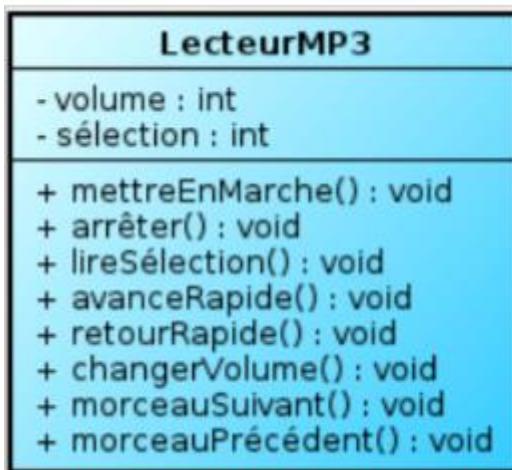
Classes

Représentation des classes

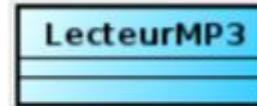
- Si la modélisation ne s'intéresse qu'aux relations entre les différentes classe du système (et pas au contenu des classes), nous pouvons ne pas représenter les attributs et les méthodes de chaque classe



Exemple :



ou



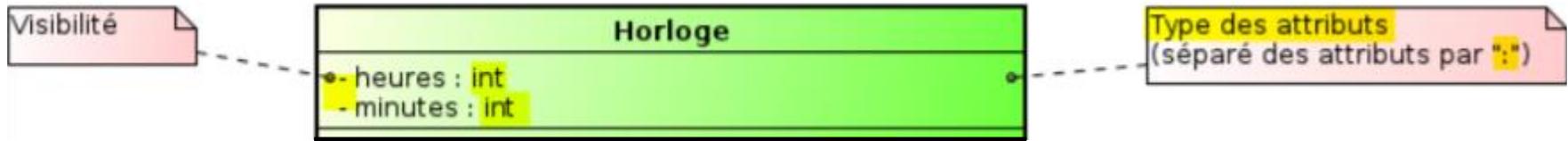
Exemple d'instance de la classe **LecteurMP3** :



En fonction des objectifs de modélisation, la représentation d'une classe peut être plus ou moins exhaustive

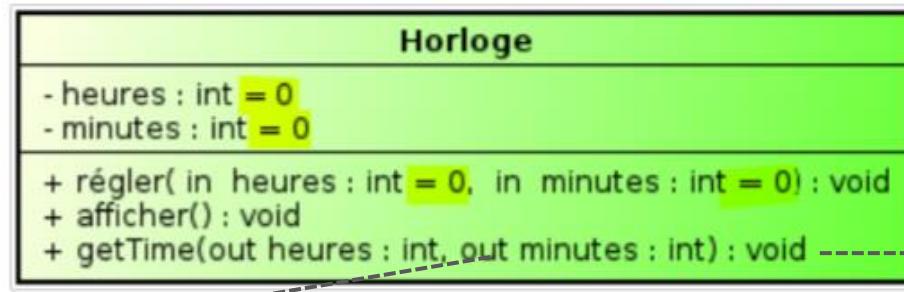
Attributs

- Une classe peut contenir des attributs
- La syntaxe d'un attribut est : *visibilité nom : type*
- La visibilité est :
 - '+' pour **public** : l'élément est visible par tous
 - '-' pour **private** : l'élément est visible seulement dans la classe
 - '#' pour **protected** : l'élément est visible dans la classe et dans les sous-classes
 - '~' pour **package** : l'élément est visible dans les classes du même package
- UML définit son propre ensemble de types
 - *Integer, real, string, ...*



Opérations

- Une opération est un service qu'une instance de la classe peut exécuter
- La syntaxe d'une opération est : *visibilité nom(parameters) : return*
- La syntaxe des paramètres est : *kind nom : type*
- Le kind peut être:
 - **In** : la valeur du paramètre est transmise à l'appel de la méthode (par l'appelant de la méthode) et ne peut pas être modifiée
 - **Out** : la valeur finale du paramètre est transmise au retour de l'appel de la méthode (à l'appelant de la méthode)
 - **Inout** : la valeur du paramètre est transmise à l'appel et au retour



Type de retour des méthodes
(séparé des attributs par ";")

Signature des méthodes
(Nom et Type des arguments séparés par ";")

Attributs et méthodes de classe

- Attributs et méthodes statiques
- Une classe peut contenir des attributs et des méthodes qui lui sont propres et auxquels nous pouvons accéder sans nécessairement instancier des objets
- Un attribut de classe n'appartient pas à un objet en particulier mais à toute la classe (il n'est pas instancié avec l'objet)
- Un attribut ou une méthode de classe est représenté par un nom souligné
- Cela permet également d'avoir une **information commune** à tous les objets instanciés

Gasoil	
- quantité : double {quantité<100}	
- prixParLitre : double = 1.23	
+ setQuantité(quantité : double) : void	
+ getQuantité() : double {query}	
+ fixePrix(parLitre : double) : void	

L'attribut **prixParLitre** appartient à la classe **Gasoil**, il ne fait pas parti des objets instanciés, mais chaque objet individuel peut y accéder.

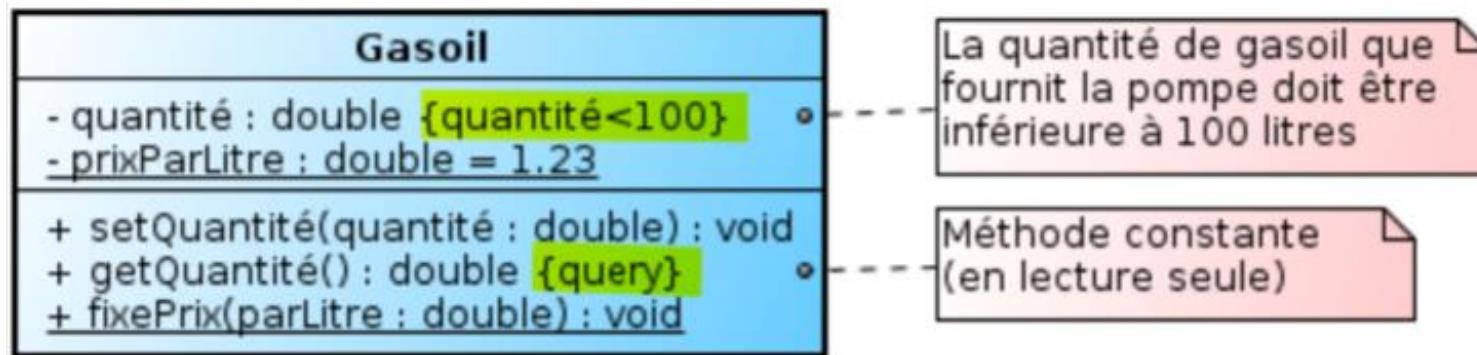
La méthode **fixePrix()** permet de modifier la valeur de l'attribut de classe **prixParLitre** (c'est le seul autorisé).

Les contraintes

Une contrainte est une condition écrite entre 2 accolades elle peut être exprimée dans :

- Un langage naturel (description textuelle)
- Un langage formel (C++, java, OCL...)

Remarques : OCL (Object Constraint Language) est un langage spécialement conçu pour exprimer des contraintes (*voir cours 4*)



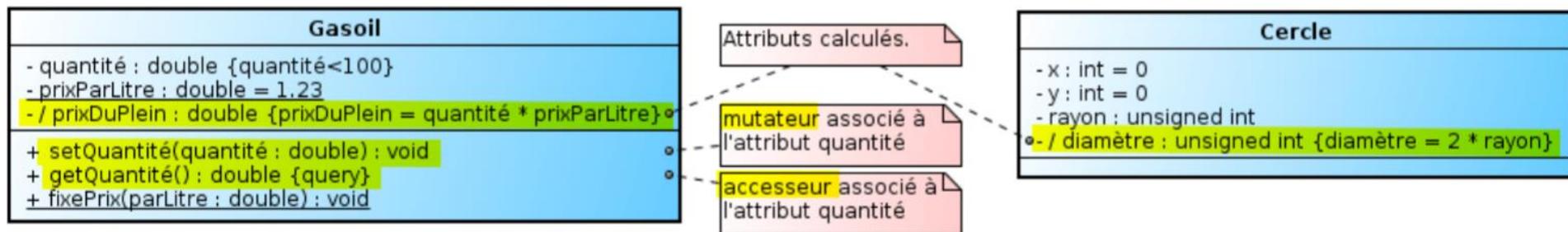
Les contraintes

Quelques contraintes utiles sur les attributs et les opérations :

- ✗ **{readOnly}** : si une telle contrainte est appliquée à un attribut, alors la valeur de celui-ci ne peut plus être modifiée une fois la valeur initiale fixée (équivalent à un attribut constant)
- ✗ **{query}** : une méthode peut être déclarée comme étant de type requête (*query*) si le code implémentant celle-ci ne modifie nullement l'état de l'objet, donc aucun de ses attributs
- ✗ **{ordered} {list}** : lorsqu'une multiplicité supérieure à 1 est précisée, il est possible d'ajouter une contrainte pour préciser si les valeurs sont ordonnées *{ordered}* ou pas *{list}* . Dans ce dernier cas, nous ne précisons même pas cette deuxième contrainte, c'est le mode par défaut
- ✗ **{unique}** : on demande cette fois-ci qu'il n'y ait aucun doublon dans les valeurs de la collection
- ✗ **{not null}** : L'attribut doit à tout prix être initialisé (utile dans le cas des pointeurs)

Les attributs dérivés (calculés)

- Une classe peut avoir des attributs calculés, c'est-à-dire que leurs valeurs sont proposées au travers d'une fonction utilisant les autres attributs précédemment exprimés
- Un tel attribut possède un nom précédé du signe « / »
- Il peut être suivi d'une contrainte permettant de le calculer



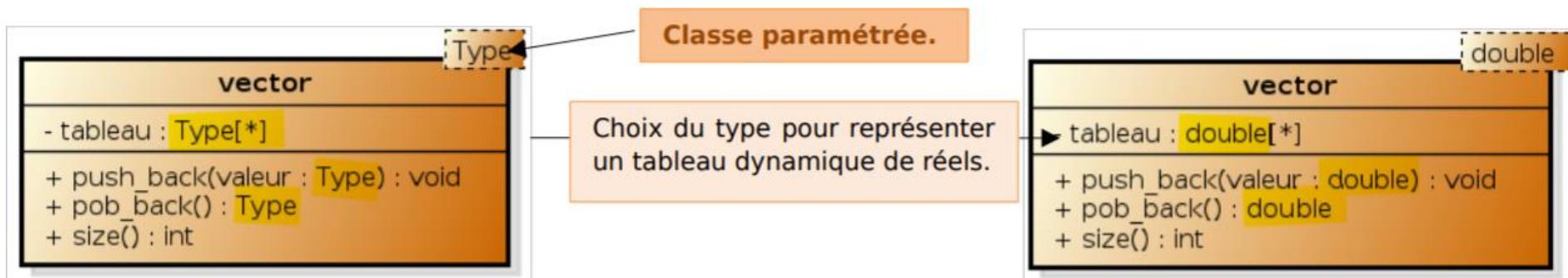
Les énumérations

- Une énumération est un type possédant un nombre fini et arbitraires de valeurs possibles, construite sur mesure par le développeur, pour typer des variables bien particulières, comme la représentation des jours de la semaine ou des mois de l'année
- **Représentation** : En UML, une énumération ne se définit pas par une classe, mais par un classeur stéréotypé « *enumeration* ». Il s'agit d'un type de données, possédant un nom, et utilisé pour énumérer un ensemble de littéraux correspondant à toutes les valeurs possibles que peut prendre une expression de ce type

« enumeration »
JourSemaine
Lundi
Mardi
Mercredi
Jeudi
Vendredi
Samedi
Dimanche

Les modèles de classe

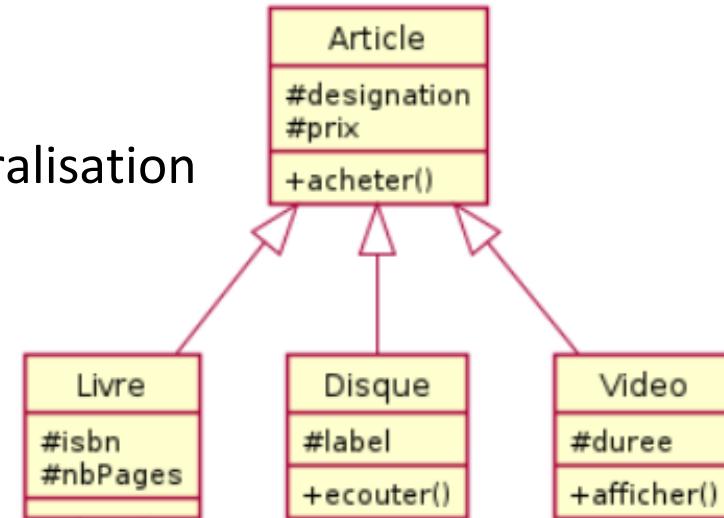
- Les modèles (*templates*) sont une fonctionnalité avancée de l'orientée objet. Un modèle est une classe paramétrée qui permet ainsi de choisir le type des attributs au besoin suivant le paramètre précisé, dans le coin supérieur droit dans un rectangle dont les côtés sont en pointillés



Les relations entre les classes

1. La relation d'héritage

- L'héritage est une relation de spécialisation/généralisation
- Les éléments spécialisés héritent de la structure et du comportement des éléments plus généraux (propriétés, associations et autres héritages)
- L'héritage indique qu'une classe B est une spécialisation d'une classe A. La classe B (appelé classe fille, classe dérivée ou sous classe) hérite des attributs et des méthodes de la classe A (appelée classe mère, classe de base ou super classe)
- Il se représente par un triangle vide afin d'indiquer le sens de la généralisation

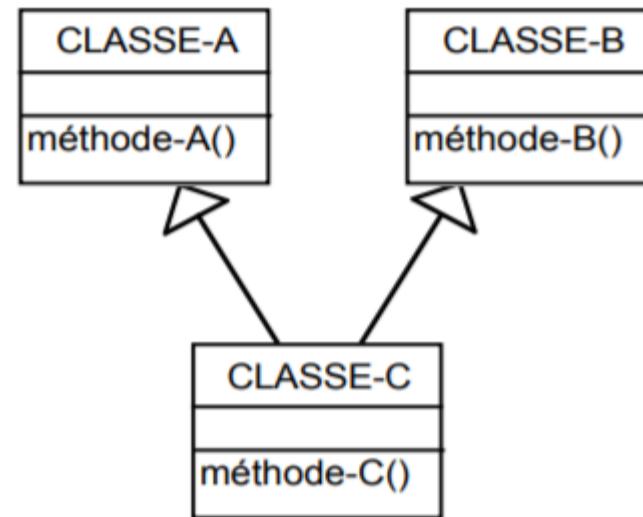


Les relations entre les classes

1. La relation d'héritage

Remarque : l'**héritage multiple** est possible en UML

C'est la capacité pour une classe dérivée, d'hériter de plusieurs classes de base



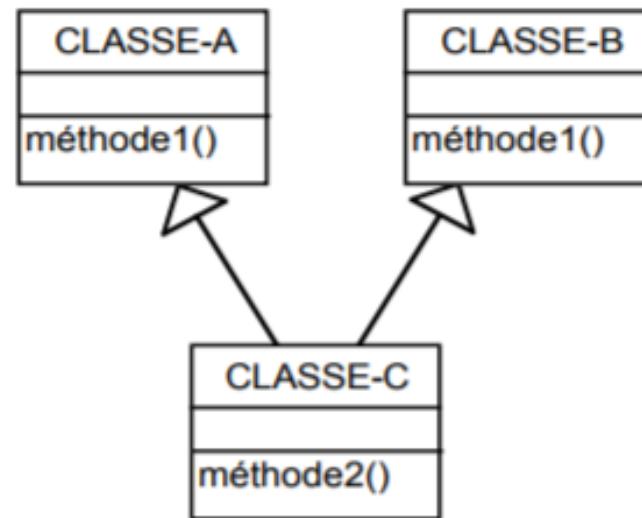
La classe **CLASSE-C** hérite des méthodes **méthode-A()** et **méthode-B()** des **CLASSE-A** et **CLASSE-B**

Les relations entre les classes

1. La relation d'héritage

Attention : Il y a **conflit** de nom quand une classe dérivée hérite plusieurs fois d'un même nom

Problème : Désigner l'élément logiciel hérité dans la classe dérivée



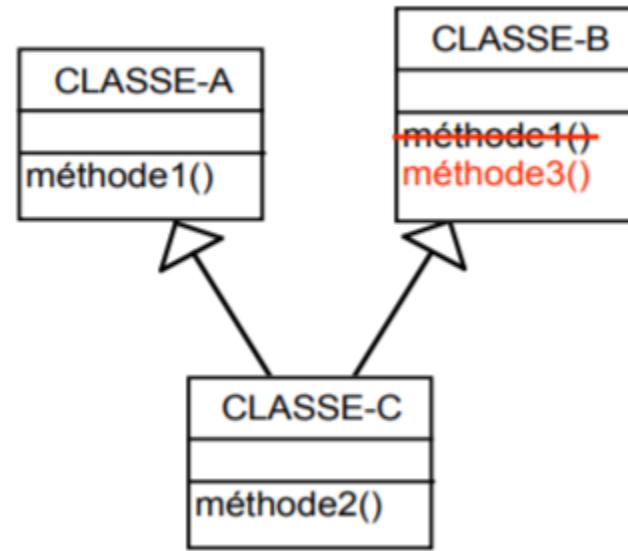
Conflit sur la **méthode1()** héritée par **CLASSE-C**

Les relations entre les classes

1. La relation d'héritage

Attention : Les conflits doivent être résolus par le programmeur (le compilateur doit savoir de quel classe on parle)

Résolution par renommage : multiplicité des noms, donc perte des avantages de la surcharge



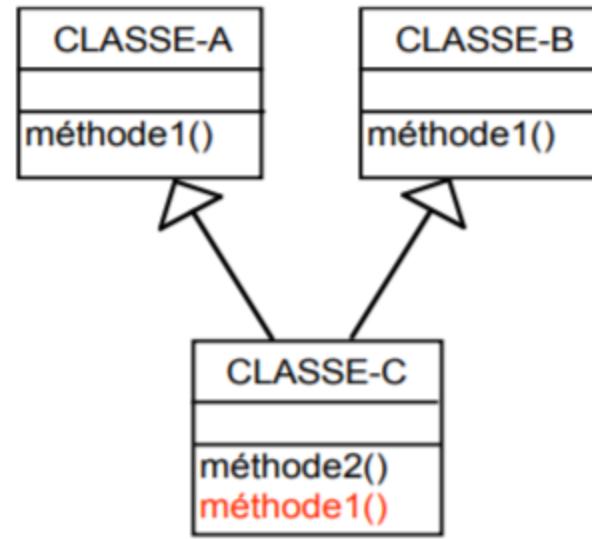
Renommage de **méthode1()** en **méthode3()**

Les relations entre les classes

1. La relation d'héritage

Attention : Les conflits doivent être résolus par le programmeur (le compilateur doit savoir de quel classe on parle)

Résolution par surcharge : pas de multiplicité des noms, l'ambiguïté est levée



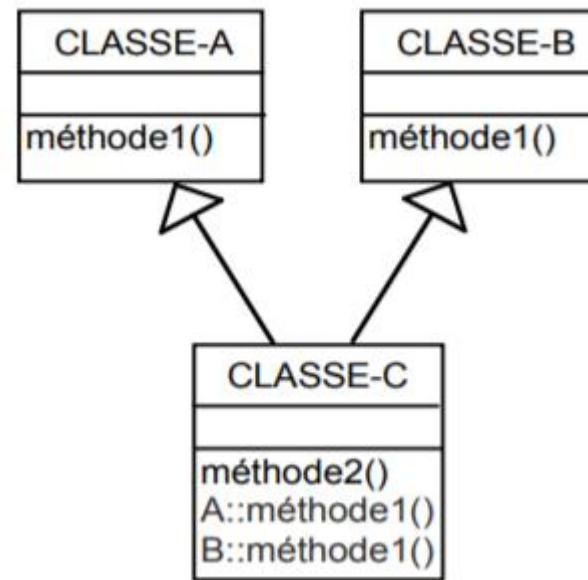
La **méthode1()** est redéfinie dans la classe **CLASSE-C**

Les relations entre les classes

1. La relation d'héritage

Attention : Les conflits doivent être résolus par le programmeur (le compilateur doit savoir de quel classe on parle)

Résolution par qualification : l'ambiguïté est levée, mais revient à une multiplicité des noms

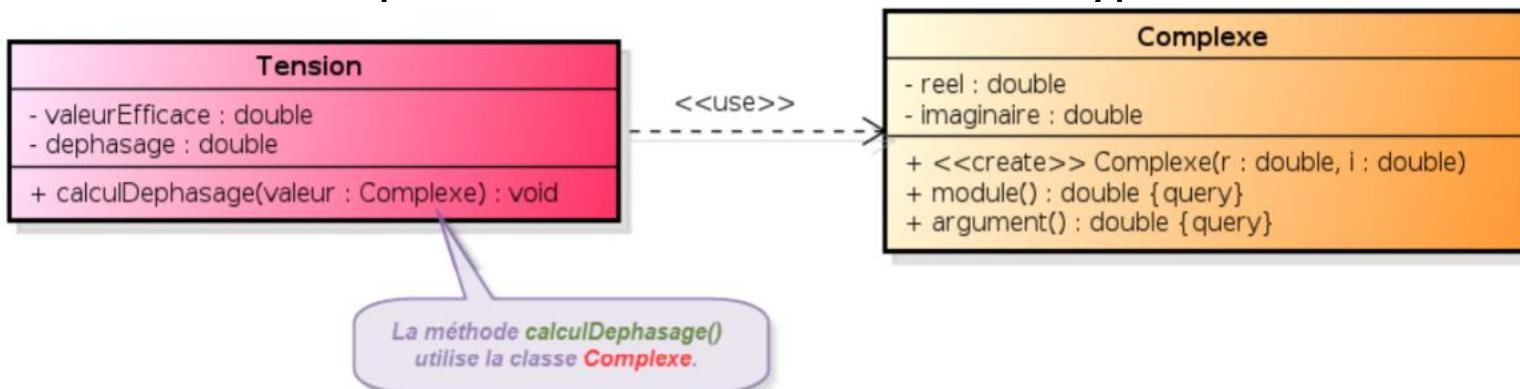


L'emploi de l'héritage multiple est souvent peu justifié et d'autres solutions de conception sont souvent préférables

Les relations entre les classes

2. La relation de dépendance

- La dépendance est la forme la plus faible de relation entre classes. Une dépendance entre deux classes signifie que l'une des deux utilise l'autre
- Une dépendance peut s'interpréter comme une relation de type «*utilise un* ». Elle est habituellement utilisée lorsqu'une classe utilise un objet d'une autre classe comme argument dans la signature d'une méthode ou lorsque l'objet de l'autre classe est créé à l'intérieur de la méthode. Dans les deux cas la durée de vie de l'objet est très courte, elle correspond à la durée d'exécution de la méthode
- **Notation** : Elle est représentée par un trait discontinu orienté, reliant les deux classes. La dépendance est souvent stéréotypée «*use*»



Les relations entre les classes

3. Les associations

- L'association signifie qu'une classe contiendra une référence (ou un pointeur) de l'objet de la classe associée sous la forme d'un attribut
- C'est une relation plus forte. Elle indique qu'une classe est en relation avec une autre pendant un certain laps de temps
- La ligne de vie des deux objets concernés ne sont pas associés étroitement (un objet peut être détruit sans que l'autre le soit nécessairement)
- L'association est représentée par un simple trait continu, reliant les deux classes : **association binaire**



Les relations entre les classes

3. Les associations

- L'association peut avoir un nom (texte), avec un éventuel sens de lecture. Ce texte n'est absolument pas exploité dans le code. Le nom d'une association doit respecter les conventions de nommage des classeurs : commencer par une lettre majuscule



- Une association binaire est composée de deux **associations ends**



Les relations entre les classes

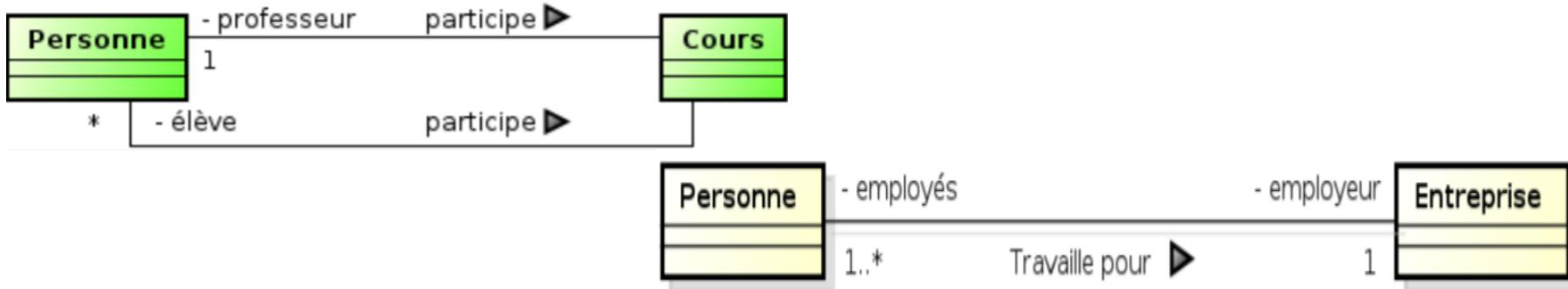
3. Les associations

- Une association end est paramétrée par :

1. **Un nom** : le rôle joué par l'entité connectée. Ce nom indique la manière dont l'objet est vu de l'autre côté de l'association



2. **Une multiplicité** (ou la cardinalité) : indique le nombre d'instances de classe étant en relation avec la classe situé à l'autre extrémité de l'association. En l'absence de spécification, la cardinalité vaut 1



Les relations entre les classes

3. Les associations

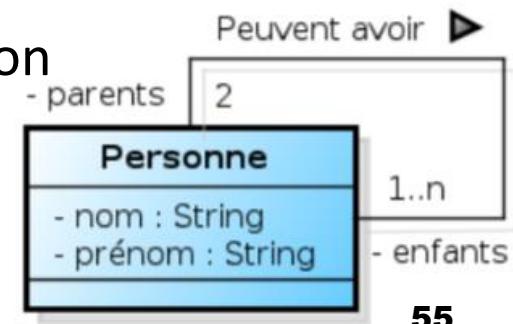
- Une association end est paramétrée par :

3. La navigabilité : les associations possèdent une navigation bidirectionnelle par défaut. Cela suppose que chaque classe possède un attribut qui fait référence à l'autre classe en association.

Il est beaucoup plus fréquent d'avoir besoin d'une navigabilité unidirectionnelle. Dans ce cas, une seule classe possède un attribut qui fait référence à l'autre classe



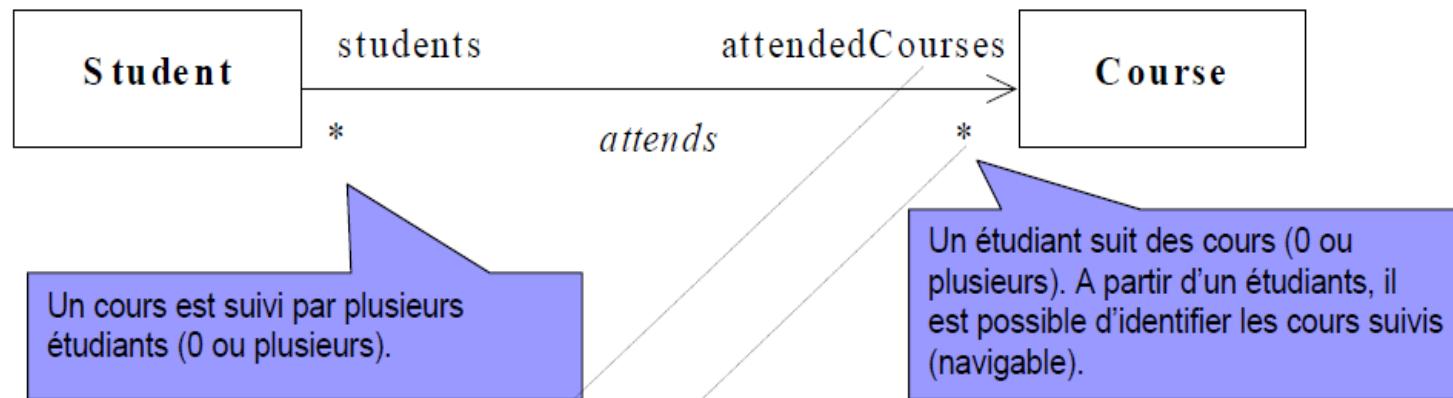
- **Association réflexives** (ou récursive) : Une association qui lie une classe avec elle-même est une *association réflexive*



Les relations entre les classes

3. Les associations

- Impact de la navigabilité sur la génération de code



```
public class Student
{
    public Course attendedCourses[];
    public Student()
    {
    }
}
```

```
public class Course
{
    public Course()
    {
    }
}
```

Les relations entre les classes

4. Associations particulières : Composition – Agrégation

Composition

- La composition indique qu'un objet A (appelé conteneur) est constitué d'un autre objet B. Cet objet A n'appartient qu'à l'objet B et ne peut pas être partagé avec un autre objet
- C'est une relation très forte, si l'objet A disparaît, alors l'objet B disparaît aussi

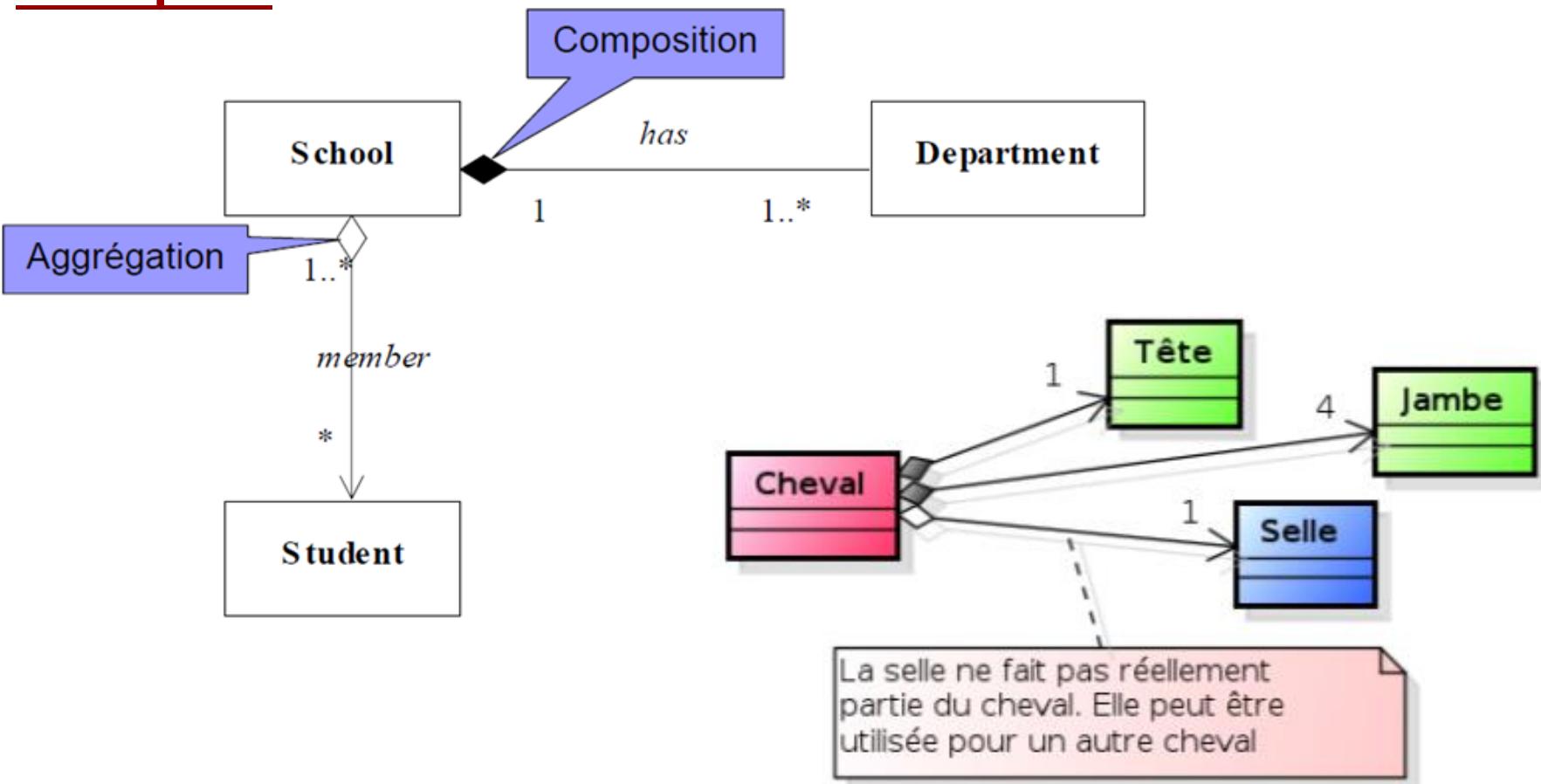
Agrégation

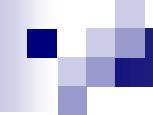
- L'agrégation indique qu'un objet A possède un autre objet B, mais contrairement à la composition, l'objet B peut exister indépendamment de l'objet A
- La suppression de l'objet A n'entraîne pas la suppression de l'objet B. L'objet A est plutôt à la fois possesseur et utilisateur de l'objet B

Les relations entre les classes

4. Associations particulières : Composition – Agrégation

Exemples:

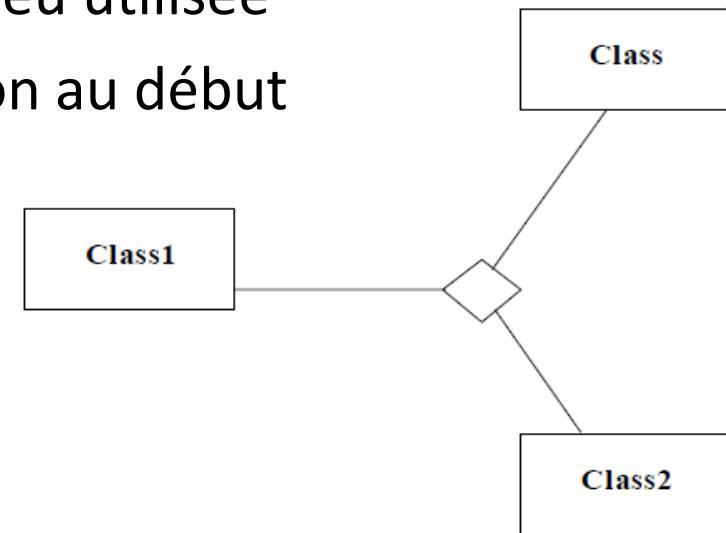




Associations reliant plusieurs classes

1. Association n-aire

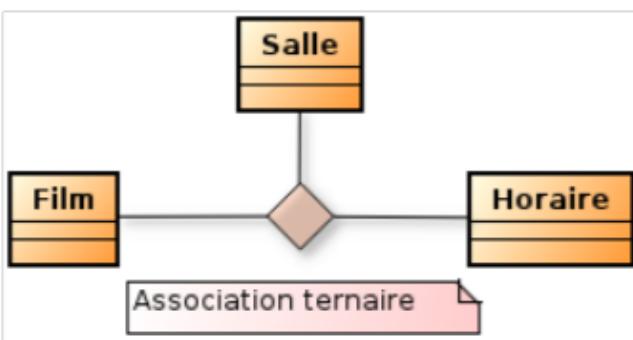
- Une association qui lie plus de 2 classes entre elles
- L'association n-aire se représente par un losange d'où part un trait allant à chaque classe
- L'association n-aire est imprécise, difficile à interpréter et souvent source d'erreur, elle est donc très peu utilisée
- Peut être utilisé pour la modélisation au début du projet, puis remplacée par un ensemble d'associations binaires afin de lever toute ambiguïté



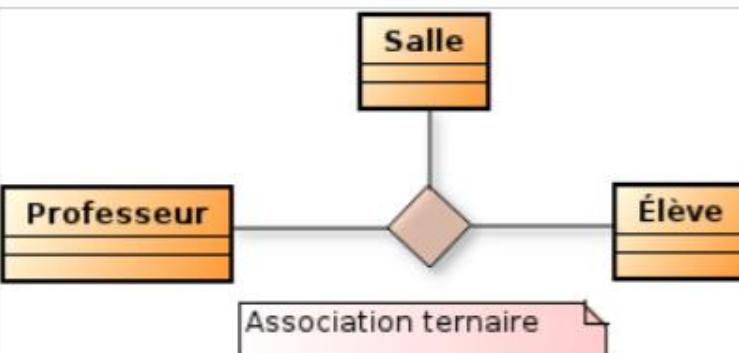
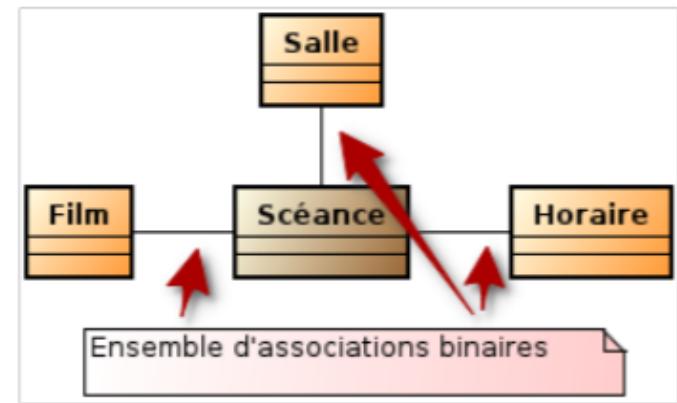
Associations reliant plusieurs classes

1. Association n-aire

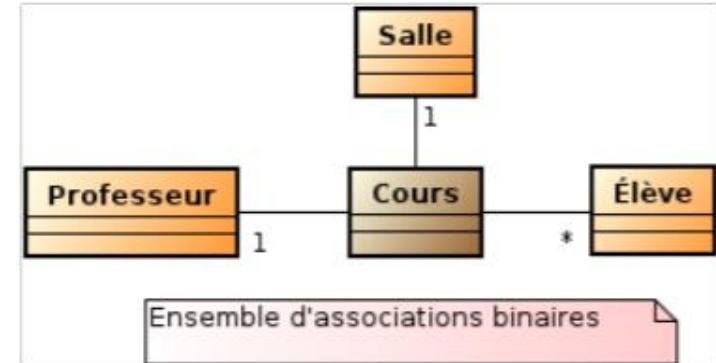
Exemples:

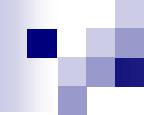


À remplacer par



À remplacer par

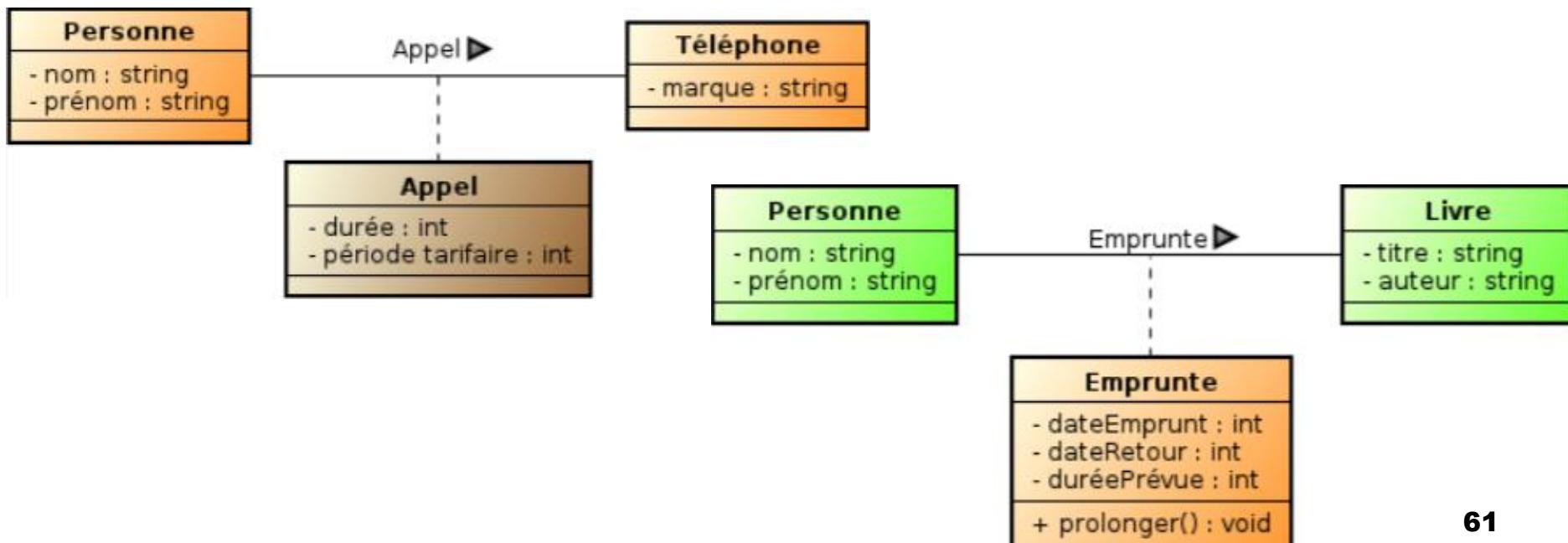




Associations reliant plusieurs classes

2. Classe association

- Une association peut apporter de nouvelles informations (attributs et méthodes) qui n'appartiennent à aucune des deux classes qu'elle relie et qui sont spécifiques à l'association
- Ces nouvelles informations peuvent être représentées par une nouvelle classe attachée à l'association via un trait en pointillés

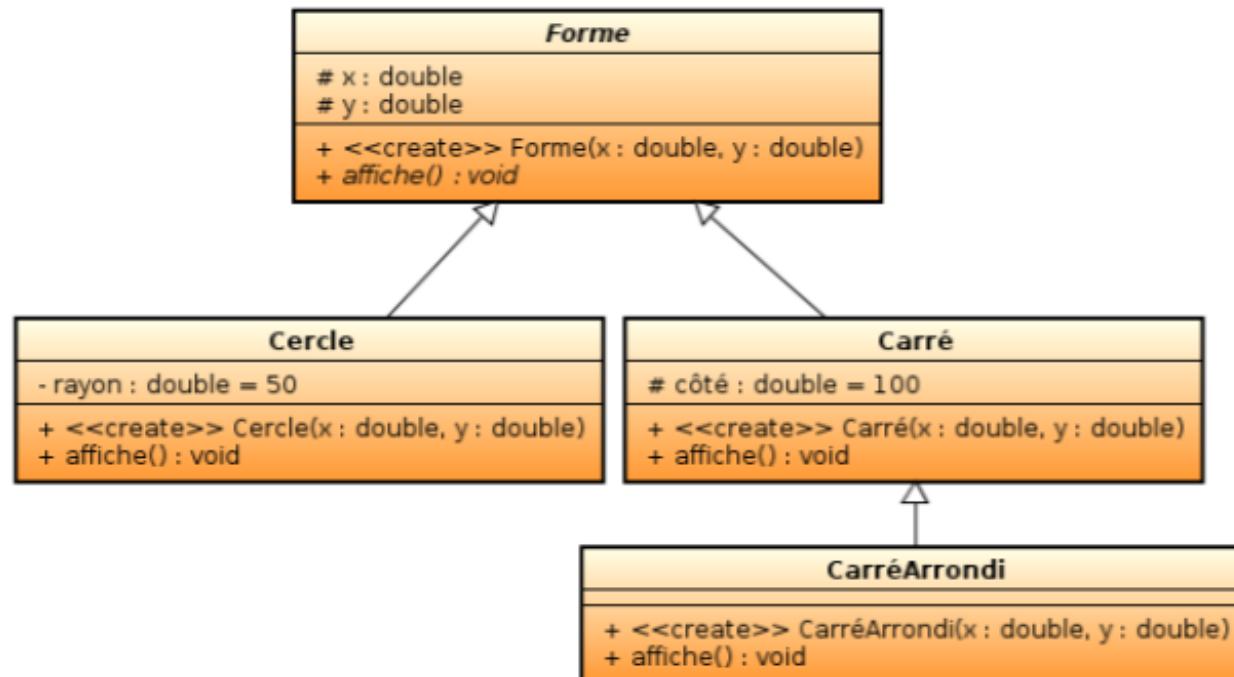


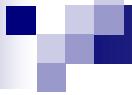
Classes concrètes et abstraites

- Une classe **concrète** possède des instances. Elle constitue un modèle complet d'objet (tous les attributs et méthodes sont complètement décrits)
- Une classe **abstraite** ne peut pas posséder d'instance directe car elle ne fournit pas une description complète. Elle a pour vocation de posséder des sous-classes concrètes et sert à factoriser des attributs et des méthodes à ses sous-classes
- Une classe abstraite possède généralement des méthodes communes aux sous-classes qui sont uniquement déclarées (sans codage interne). Une méthode introduite dans une classe avec sa seule signature et sans code est appelée une méthode abstraite

Classes concrètes et abstraites

- En UML, une classe ou une méthode abstraite sont représentées avec une mise en italique du nom de la classe ou de la méthode
- Tout classe possédant au moins une méthode abstraite est une classe abstraite. En effet, la seule présence d'une méthode incomplète (le code est absent) implique que la classe ne soit pas une description complète.





Interfaces et relation de réalisation

Interface

Définition : description d'un ensemble d'opérations utilisées pour spécifier un service offert par une classe

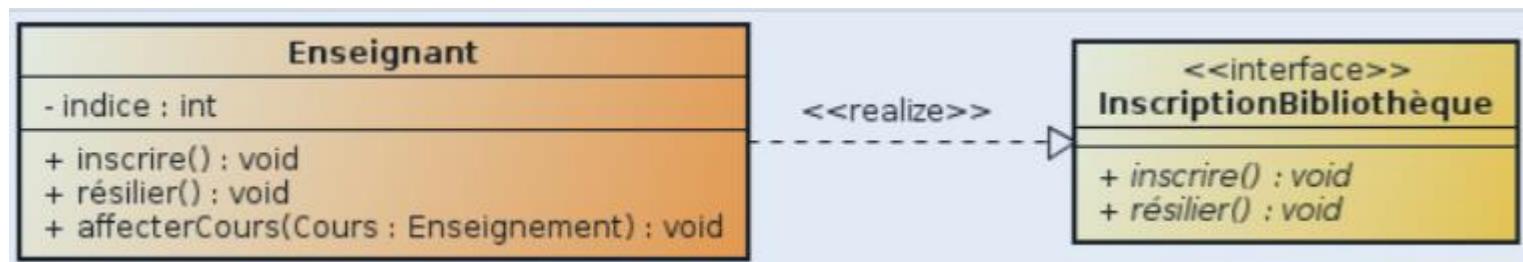
- ❑ Elle ne contient ni attribut, ni association, ni implémentation des opérations (les opérations sont abstraites)
- ❑ Une classe réalisant une interface doit :
 - ❑ soit implémenter les opérations de l'interface
 - ❑ soit définir les opérations de l'interface comme des opérations abstraites (implémentables par les classes filles)

Représentation UML : classe ayant le stéréotype **<<interface>>**, ou par un cercle pour faire référence à l'interface utilisée dans la classe



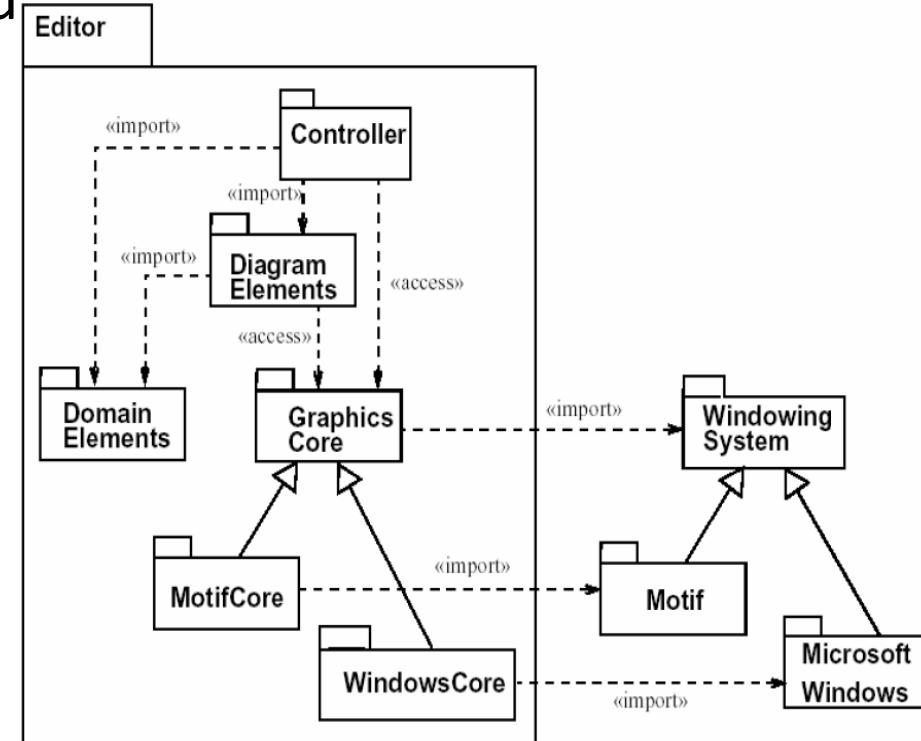
Interfaces et relation de réalisation

- Le rôle d'une **interface** est de regrouper un ensemble d'opérations assurant un service cohérent
- Une interface ne dispose que de méthodes publiques abstraites
- La généralisation peut être utilisée entre interfaces, par contre, une interface ne peut être instanciée (comme une classe abstraite)
- Comme l'interface n'a qu'un objectif de spécification, au moins un élément d'implémentation (une classe) doit lui être associée
- La relation de réalisation (**«realize»**) permet justement de mettre en relation un élément de spécification avec son implémentation



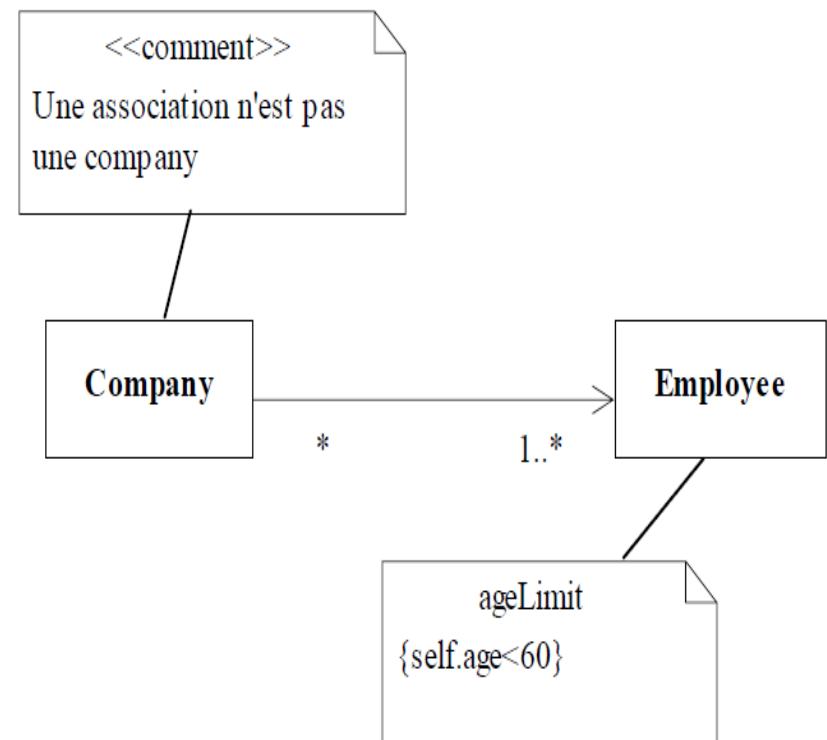
Package

- Un **package** permet de grouper des éléments
- L'idée est de rapprocher les éléments sémantiquement proches, offrant un ensemble de services homogènes et cohérents
- Ils séparent le modèle en éléments logiques, et montrent leurs interactions à un plus haut niveau
- Un package peut servir d'espace de désignation
- Un package peut inclure d'autres
- Un package peut importer d'autres package
- L'héritage entre package est possible



Contraintes et notes

- Il est possible de contraindre ou d'annoter n'importe quel élément du modèle
- Les contraintes et les notes sont bien souvent écrites en langage naturel
- Le **langage OCL** est cependant préconisé pour décrire des contraintes (*voir cours 4*)
 - Exemple : `self.age<60`



Établir un diagramme de classes

A partir de(s)

- mots clés de l'énoncé du problème,
- l'analyse du diagramme de Use cases
- l'analyse des diagrammes de séquence correspondants aux scénarios des use cases

1. Établir une liste des objets candidats pour le modèle statique de l'application
2. Evaluer chaque objet de cette liste en vu de le retenir ou de l'éliminer dans le modèle statique de l'application

Classe/ Objet candidat	Eliminé pour les raisons suivantes	Nom de classe pour la modélisation
<i>Nom trouvé dans les spécifications</i>	<i>L'élément peut être un simple attribut, une classe utilitaire ou en dehors du sujet de modélisation</i>	<i>Nom retenu pour la modélisation</i>
...

Établir un diagramme de classes

1. Réaliser un premier diagramme de classes qui contiendra l'ensemble des classes retenues et qui mettra en évidence des associations entre les classe. Les associations seront de simples traits banalisés : pas de nom, pas de multiplicité.

2. Raffiner le diagramme précédent
 1. Renseigner les association
 - nom, multiplicité, agrégation,navigabilité, rôles des classes associées
 2. Renseigner les classes
 - attributs, classes associations, classes abstraites , héritage

3. Vérifier que le diagramme permet de remplir les différents cas d'utilisation et leurs scénarii associés

3. Diagramme d'objets

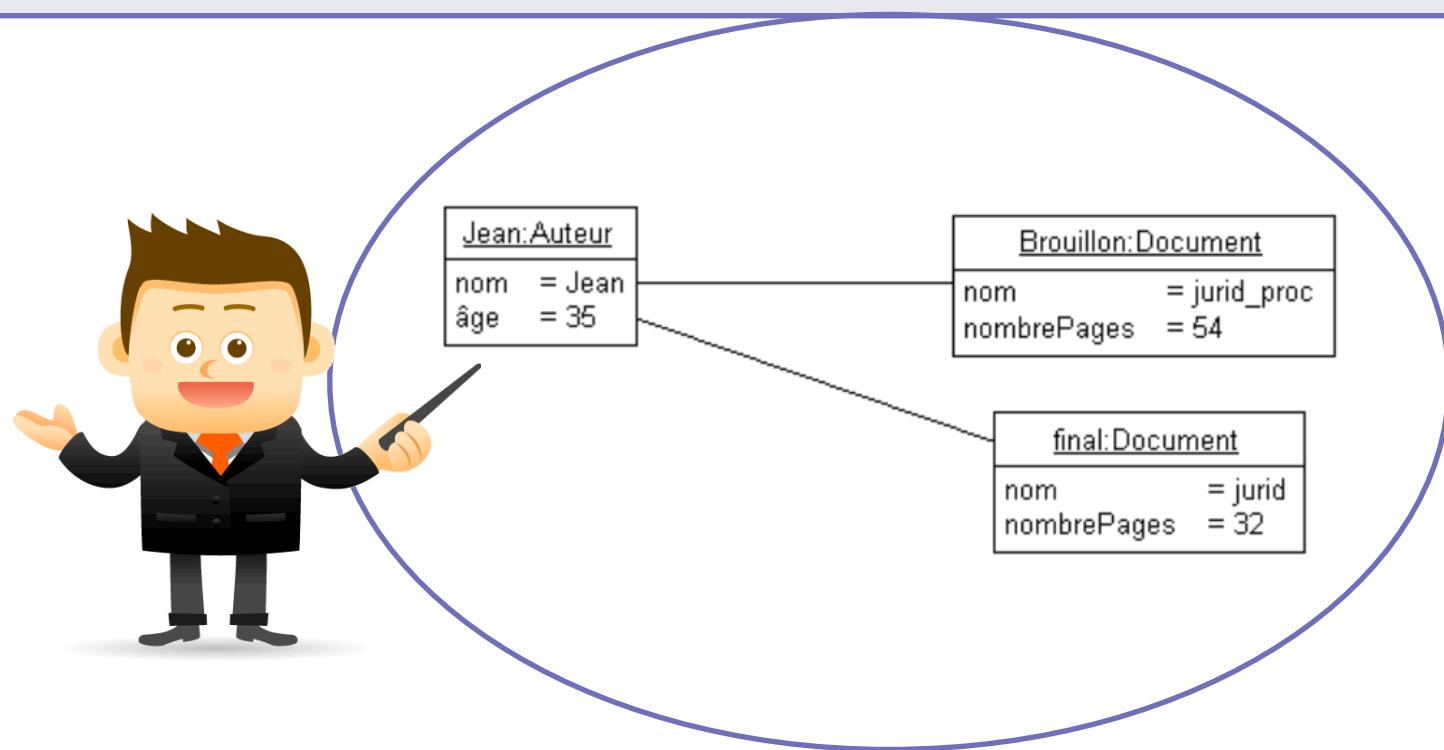


Diagramme d'Objets

Rôle du diagramme d'objets :

- ❑ Le diagramme d'objets fait parties des diagrammes structurels (*statique*)
- ❑ Il représente les objets d'un système (c.a.d. les instances des classes) et leurs liens (c.a.d. les instances des associations) à un instant donné
- ❑ Il donne une vue figée du système à un moment précis
- ❑ A un diagramme de classe correspond une infinité de diagrammes d'objets
- ❑ Nous nous servirons du diagramme d'objet pour donner des exemples, des cas de figure, qui permettront d'affiner le diagramme de classe et de mieux le comprendre

Diagramme d'Objets

Représentation graphique :

Chaque objet est représenté dans un rectangle dans lequel figure le nom de l'objet (souligné) et éventuellement la valeur de un ou plusieurs de ses attributs.

Comme pour la représentation d'une classe, la représentation d'un objet pourra être plus ou moins détaillée

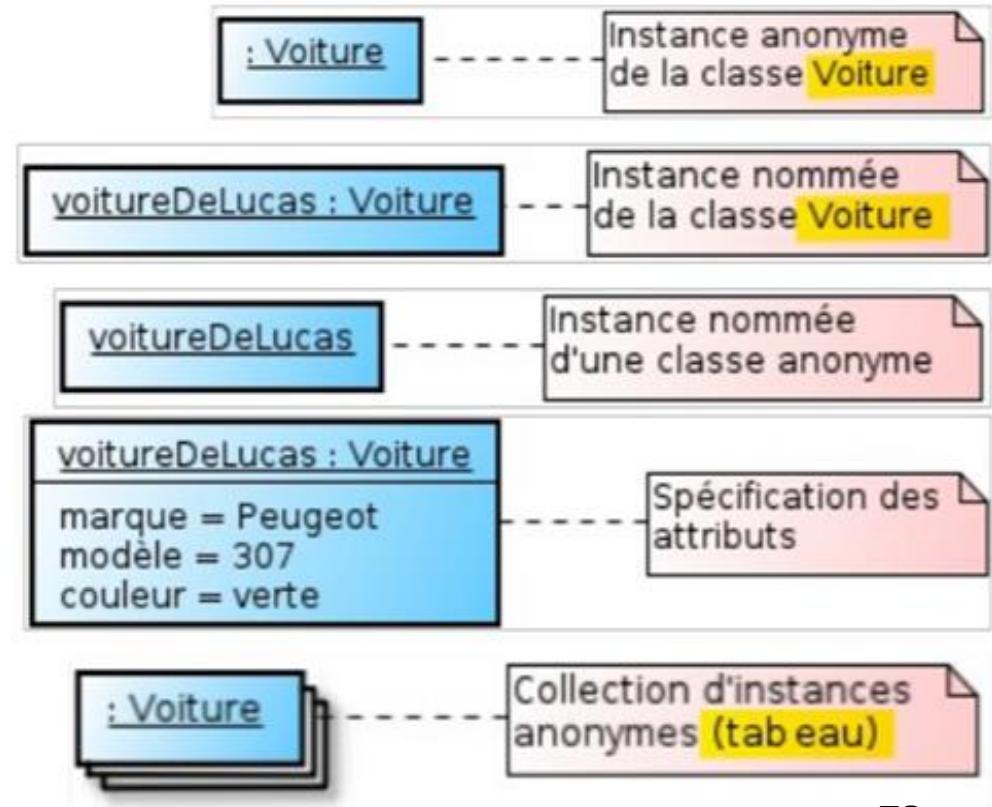
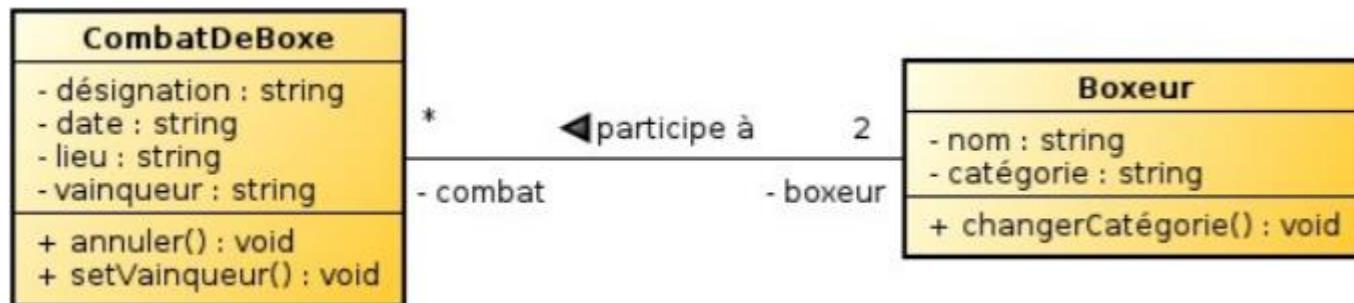


Diagramme d'Objets

Exemple : Soit le diagramme de classe suivant :



Avec le diagramme objet ci-dessous on définit une instance particulière du diagramme de classe (qui est le combat entre Mohamed Ali et George Foreman qui a eu lieu le 24 septembre 1974 à Kinshasa)

