Université Claude Bernard  Lyon 1

Introduction à UML – Unified Modeling Language

Diagrammes de Classe et d'Objet

IUT de LYON
2017-2018

Sommaire

Table des figures

Diagramme de classes d'UML

1. Introduction

Le diagramme de classes constitue le diagramme incontournable de la modélisation orientée objet. Il représente la structure interne du système à mettre en place. Il traduit une représentation abstraite de l'ensemble des objets du système qui vont interagir ensemble pour l'achèvement des cas d'utilisation.

Le diagramme de classes modélise les concepts du domaine d'application ainsi que les concepts internes conçus dans le cadre de l'implémentation d'une application. Il s'agit d'une vue statique ne tenant pas compte du facteur temporel dans le comportement du système, permettant de modéliser les classes du système et leurs relations indépendamment d'un langage de programmation particulier.

Les principaux éléments de cette vue statique sont les classes et leurs relations : association, généralisation et plusieurs types de dépendances, telles que la réalisation et l'utilisation.

2. Les classes

2.1 Notions de classe et d'instance de classe

Le monde est composé d'entités qui « collaborent ». L'approche objet consiste à résoudre un problème en termes d'objets qui collaborent. Ces objets sont des abstractions des objets réels.

Une classe est la description formelle d'un ensemble d'objets ayant une sémantique et des caractéristiques communes. Ainsi, une classe est un concept abstrait représentant des éléments variés comme :

- des éléments concrets (ex : des avions),
- des éléments abstraits (ex : des commandes de marchandises),
- des composants d'une application (ex : les boîtes de dialogue),
- des structures informatiques (ex : des listes chaînées),
- des éléments comportementaux (ex : des tâches), etc.

Tout système orienté objet est organisé autour des classes. Une *instance* est une concrétisation d'un concept abstrait. Par exemple : ***Batman*** est une instance du concept abstrait *Héros* ;

Un objet est une instance d'une classe. C'est une entité discrète dotée d'une identité, d'un état et d'un comportement que l'on peut invoquer. Les objets sont des éléments individuels d'un système en cours d'exécution. En d'autres termes, un objet définit une représentation d'une entité atomique réelle ou virtuelle, dans le but de le piloter ou de le simuler. Ils encapsulent une partie des connaissances du monde dans lequel ils évoluent.

Par exemple, si l'on considère que *Homme* (au sens être humain) est un concept abstrait, on peut dire que la personne Marie est une instance de Homme. Si *Homme* était une classe, Marie en serait une instance : un objet.

2.2 Caractéristiques d'une classe

Une classe définit un jeu d'objets dotés de caractéristiques communes. Les caractéristiques d'un objet permettent de spécifier son *identité*, son *état* et son *comportement*. La Classe est par conséquent une description d'un ensemble d'objets qui partagent les mêmes attributs, opérations, méthodes, relations et contraintes.

Identité d'un objet :

En plus de son état un objet possède une identité qui permet de le distinguer de manière non ambiguë indépendamment de son Etat.

État d'un objet :

Ce sont les attributs et généralement les *terminaisons d'associations*, tous deux réunis sous le terme de *propriétés structurelles*, ou tout simplement *propriétés*, qui décrivent l'état d'un objet. Les attributs sont utilisés pour des valeurs de données. Les associations sont utilisées pour connecter les classes du diagramme de classe.

Les propriétés décrites par les attributs prennent des valeurs lorsque la classe est instanciée. L'instance d'une association est appelée un lien.

Comportement d'un objet :

Il regroupe toutes les compétences d'un objet et décrit les actions et les réactions de cet objet. Les opérations décrivent les éléments individuels d'un comportement que l'on peut invoquer. Ce sont des fonctions qui peuvent prendre des valeurs en entrée et modifier les attributs ou produire des résultats.

Les attributs, les terminaisons d'association et les méthodes constituent donc les caractéristiques d'une classe (et de ses instances).

2.3 Représentation graphique

Une classe est représentée par un rectangle divisé en trois compartiments (figure 1).

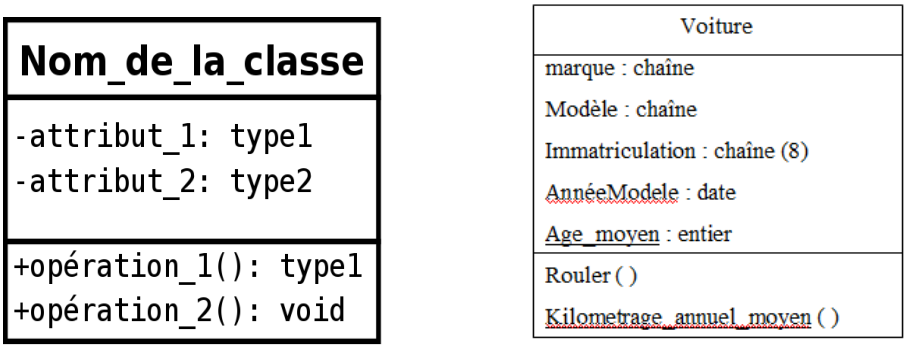


Figure 1: Représentation UML d'une classe et illustration avec la classe Voiture

Le premier compartiment indique le nom de la classe, le deuxième ses attributs et le troisième ses opérations.

1er compartiment : Nom de la classe

Le nom de la classe doit évoquer le concept décrit par la classe. Il commence par une majuscule. On peut ajouter des informations subsidiaires comme le nom de l'auteur de la modélisation, la date, etc. Pour indiquer qu'une classe est abstraite, il faut ajouter le mot-clef *abstract*.

La syntaxe de base de la déclaration d'un nom d'une classe est la suivante :

```
[ <Nom_du_paquetage_1>::...::<Nom_du_paquetage_N> ]
<Nom_de_la_classe> [ { [abstract], [<auteur>], [<date>], ... } ]
```

2ème compartiment : Attributs de la classe

Les attributs définissent des informations qu'une classe ou un objet doivent connaître. Ils représentent les données encapsulées dans les objets de cette classe. Chaque attribut est défini par un nom, un type de données, une visibilité et peut être initialisé. Le nom de l'attribut est obligatoirement unique dans la classe. La syntaxe de la déclaration d'un attribut est la suivante :

```
<visibilité> [/] <nom_attribut> :
<type> [ '['<multiplicité>']' [{<contrainte>}] ] [ = <valeur_par_défaut> ]
```

Le type de l'attribut (`<type>`) peut être un nom de classe, un nom d'interface ou un type de donné prédéfini. La multiplicité (`<multiplicité>`) d'un attribut précise le nombre de valeurs que l'attribut peut contenir. Lorsqu'une multiplicité supérieure à 1 est précisée, il est possible d'ajouter une contrainte (`{<contrainte>}`) pour préciser si les valeurs sont ordonnées (*{ordered}*) ou pas (*{list}*).

Attributs de classe

Par défaut, chaque instance d'une classe possède sa propre copie des attributs de la classe. Les valeurs des attributs peuvent donc différer d'un objet à un autre. Cependant, il est parfois nécessaire de définir un *attribut de classe* (*static* en Java ou en C++) qui garde une valeur unique et partagée par toutes les instances de la classe. Les instances ont accès à cet attribut mais n'en possèdent pas une copie. Un attribut de classe n'est donc pas une propriété d'une instance mais une propriété de la classe et l'accès à cet attribut ne nécessite pas l'existence d'une instance.

Graphiquement, un attribut de classe est souligné.

Attributs dérivés

Les attributs dérivés peuvent être calculés à partir d'autres attributs et de formules de calcul. Les attributs dérivés sont symbolisés par l'ajout d'un « / » devant leur nom. (cf. figure 2)

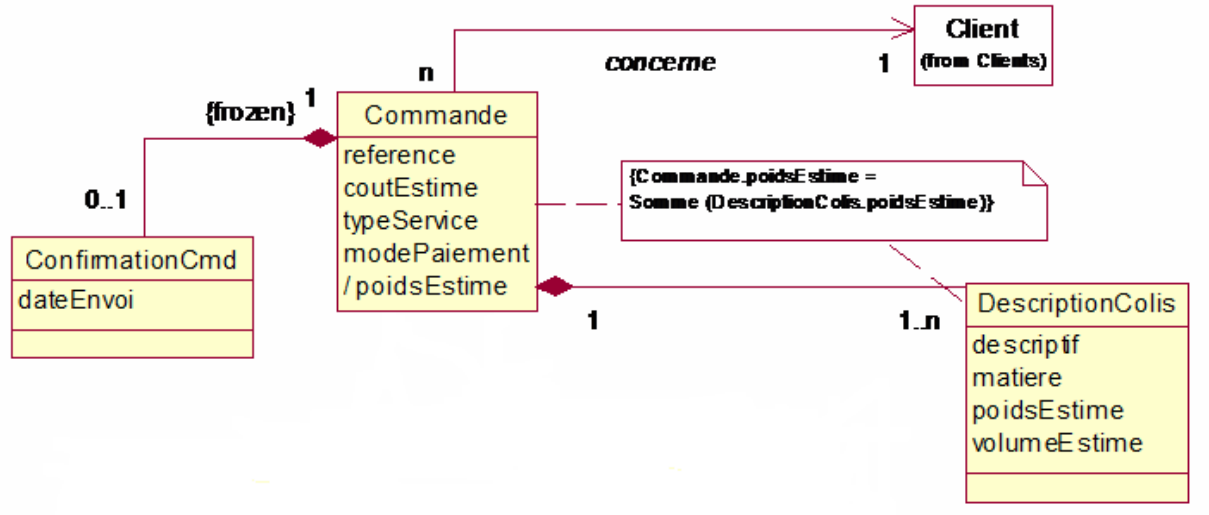


Figure 2 : Illustration d'attribut dérivé (/poidsEstime dans la classe Commande)1

Lors de la conception, un attribut dérivé peut être utilisé comme marqueur afin de déterminer les règles à lui appliquer.

3ème compartiment : méthodes de la classe

Dans une classe, une opération (même nom et même types de paramètres) doit être unique. Il est possible que le nom d'une opération apparaisse plusieurs fois mais avec des paramètres différents, il s'agit alors d'une opération dite surchargée. En revanche, il est impossible que deux opérations ne se distinguent que par leur valeur retournée.

La déclaration d'une opération contient les types des paramètres et le type de la valeur de retour, sa syntaxe est la suivante :

```
<visibilité> <nom_méthode> ([<paramètre_1>, ... , <paramètre_N>]) :
[<type_renvoyé>] [{<propriétés>}]
```

La syntaxe de définition d'un paramètre (`<paramètre>`) est la suivante :

```
[<direction>] <nom_paramètre>:<type> ['['<multiplicité>']']
[=<valeur_par_défaut>]
```

La direction peut prendre l'une des valeurs suivante :

in : Paramètre d'entrée passé par valeur.

out : Paramètre de sortie uniquement. Il n'y a pas de valeur d'entrée et la valeur finale est disponible pour l'appelant.

inout : Paramètre d'entrée/sortie. La valeur finale est disponible pour l'appelant.

Le type du paramètre (`<type>`) peut être un nom de classe, un nom d'interface ou encore un type de donné prédéfini.

[1] Pour des raisons de lisibilité, les méthodes ont été occultées.

Les propriétés (`<propriétés>`) sont des contraintes ou des informations complémentaires comme les exceptions, les préconditions, les postconditions ou encore l'indication qu'une méthode est abstraite (mot-clef *abstract*).

Méthode de classe

Comme pour les attributs de classe, il est possible de déclarer des méthodes de classe. Une méthode de classe s'utilise au niveau de la classe, sans tenir compte des instances. C'est le cas par ex. de méthodes de tri des instances, ou d'un accesseur au compteur du nombre d'instances de la classe. Elle ne peut manipuler que des attributs *de classe* et ses propres paramètres. Cette méthode n'a pas accès aux attributs *de la classe* (*i.e.* des instances de la classe) puisqu'elle est définie *en dehors* de toute instance. L'accès à une méthode de classe ne nécessite pas l'existence d'une instance de cette classe.

Rappel : l'appel en langage de programmation se fait de la façon suivante : **`maClasse.méthode-de-classe()`** ;

Ex. en java : **`Collections.sort(), Personnel.getDernierNuméro() ;`**

Graphiquement en UML, une méthode de classe est soulignée.

Méthodes et classes abstraites

Une méthode est dite abstraite lorsqu'on connaît son entête mais pas la manière dont elle peut être réalisée (*i.e.* on connaît sa déclaration mais pas sa définition).

Une classe est dite abstraite lorsqu'elle définit au moins une méthode abstraite ou lorsqu'une classe dite parent possède une méthode abstraite non encore réalisée. On ne peut instancier une classe abstraite : elle est vouée à se spécialiser (notion d'héritage & de Généralisation). Il est à noter qu'une classe abstraite peut très bien contenir des méthodes concrètes.

Une classe abstraite pure, ne comporte que des méthodes abstraites. En programmation orientée objet, une telle classe est appelée *une interface*. L'indication d'une classe abstraite se fait par l'ajout du mot-clef *abstract* derrière son nom.

Représentation graphique d'un objet : Un objet se représente en UML sous la forme d'un rectangle; le nom de l'objet est **toujours souligné** indiquant le nom de la classe précédé de "**:**". Un objet peut être nommé ou pas. (cf. figure 3)

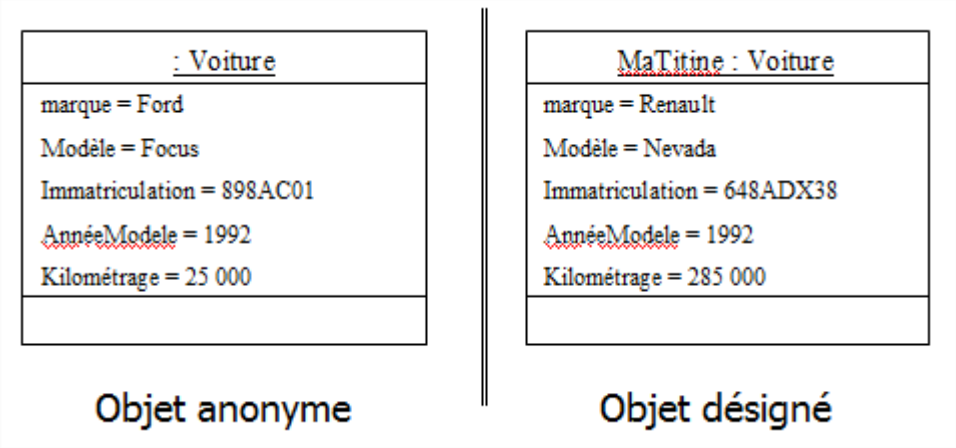


Figure 3 : Représentation graphique d'un objet

2.4 Encapsulation, visibilité, interface

L'encapsulation est un mécanisme consistant à incorporer les données et les méthodes dans 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 proposés. Ces services accessibles (offerts) aux utilisateurs de l'objet définissent ce que l'on appel l'interface de l'objet (sa vue externe). En d'autres termes, l'encapsulation assure l'intégrité des données contenues dans l'objet : c'est l'occultation des détails de réalisation.

Par défaut les valeurs des attributs d'un objet sont encapsulées dans l'objet et ne peuvent pas être manipulées directement par un autre objet. Des règles de visibilité précisent la notion d'encapsulation qui a pour but entre autre de réduire le temps d'accès aux attributs. Il existe quatre visibilités prédéfinies.

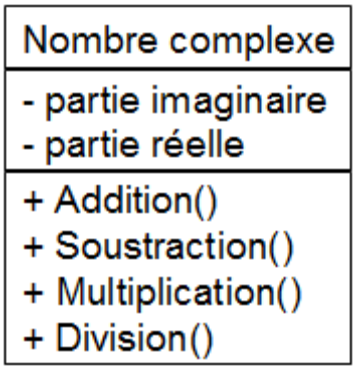


Figure 4 : Exemple d'encapsulation.

***Public* ou + :** qui indique que l'attribut est visible pour toutes les classes. En d'autres termes, tout élément qui peut voir le conteneur peut également voir l'élément indiqué.

***Protected* ou # :** seul un élément situé dans le conteneur ou un de ses descendants peut voir l'élément indiqué.

***Private* ou - :** seul un élément situé dans le conteneur peut voir l'élément.

***Package* ou ~ ou rien :** seul un élément déclaré dans le même paquetage peut voir l'élément.

Dans une classe, le marqueur de visibilité se situe au niveau de chacune de ses caractéristiques (attributs, terminaisons d'association et opération). Il permet d'indiquer si une autre classe peut y accéder.

Lorsque des attributs doivent être accessibles de l'extérieur, l'accès doit se faire par l'intermédiaire d'opérations (figure 4) et non directement.

3. Relations entre classes

3.1 Notion d'association

Une association décrit un groupe de liens ayant une structure et une sémantique commune entre deux classes (association binaire) ou plus (association n-aire). Un lien est une connexion physique ou conceptuelle entre des instances d'objets. Une association (fig 5.b) indique par conséquent l'existence de liens entre des instances des classes associées (fig 5.a).

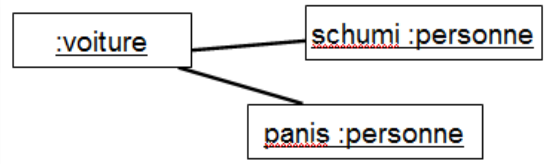


fig 5.a – Instance d'association : liens entre objets

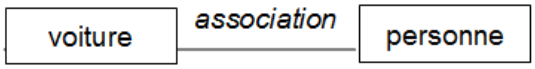


fig 5.b – Une association

Une association se représente par une ligne continue tracée entre les classes. L'association correspond à une abstraction des liens qui existent entre les objets instances. Les associations sont **fondamentalement bidirectionnelles**. Le nommage des associations facilite par conséquent la compréhension des modèles (fig 6).

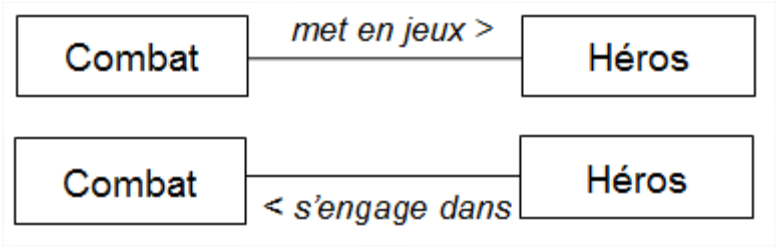


Figure 6. Illustration de nommage d'association

3.2 Terminaison d'association : notion de Rôle

Il est possible que selon l'association, une classe joue un rôle différent. La terminaison d'une association indique par conséquent l'implication de la classe (cf figure 7). Dans cette figure, selon l'association, un Personnage, qu'il soit spectateur (rôle=Public) ou acteur (rôle=Héros) peut s'engager dans le Combat (spectacle).

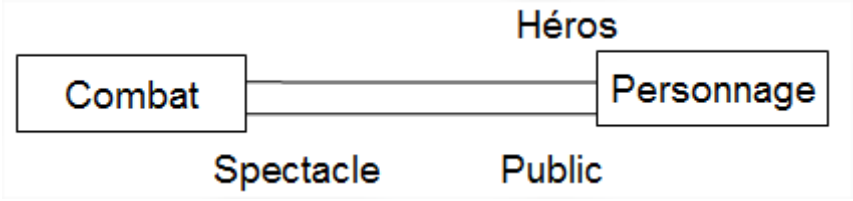


Figure 7. Illustration de notion de rôle

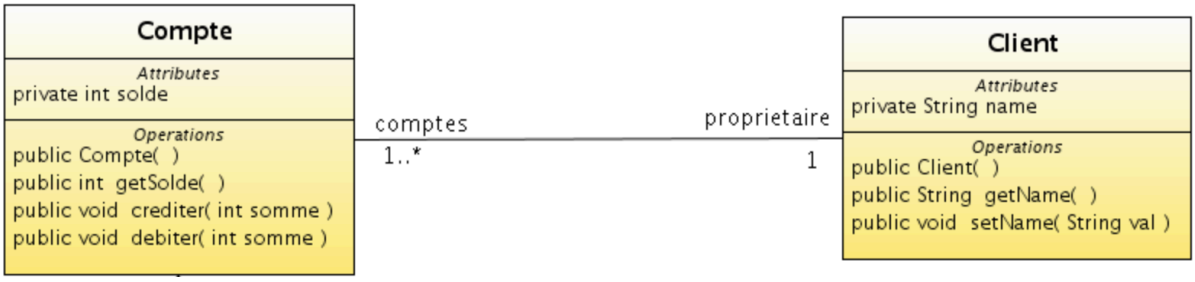
Par définition, le rôle décrit comment une classe voit une autre classe au travers d'une association. **Il prend tout son intérêt lorsque plusieurs associations existent entre 2 classes**.

3.3 Multiplicité (cardinalité)

La multiplicité précise le *nombre possible d'instances* reliées à l'autre instance dans la relation. En d'autres termes, elle déclare le nombre d'objets susceptibles d'occuper la position définie par la terminaison d'association. Voici quelques exemples de multiplicité :

- exactement un : 1 ou 1..1 ou rien (valeur par défaut)
- plusieurs : * ou 0..*
- un à plusieurs : 1..*
- de un à dix : 1..10

Illustration :



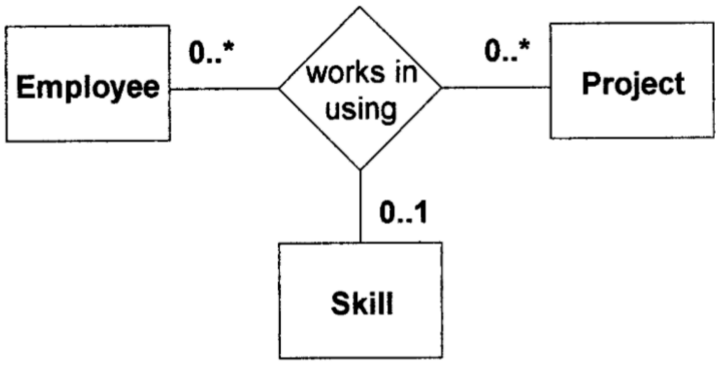
Lecture de la multiplicité coté Client : « *Un* Compte est la propriété d'un seul Client (rôle 'propriétaire' dans la relation) ». On part toujours de la classe opposée, considérée au singulier (« Un Compte »), pour trouver le nb possible d'objets reliés à cette unique instance.

Lecture de la multiplicité coté Compte : « Un Client est propriétaire d'un ou plusieurs Compte(s) ».

Les multiplicités sont fondamentales dans la modélisation du diagramme de classes. Ce n'est pas parce que ce sont de *petits* chiffres du diagramme qu'il faut les considérer comme secondaire ! Selon leur valeur, la génération de code sera complètement différente.

Nota : pour une association n-aire, la multiplicité minimale doit en principe être 0. En effet, une multiplicité minimale de 1 (ou plus) sur une extrémité implique qu'il doit exister un lien (ou plus) pour TOUTES les combinaisons possibles des instances des classes situées aux autres extrémités de l'association n-aire, ce qui représente une forte contrainte de modélisation.

Illustration des multiplicités pour une ternaire, traduisant le fait qu'un Salarié travaille sur un Projet avec une Compétence donnée (ex. : connaissance d'un langage de programmation) :



Lecture : Pour un couple (Salarié, Projet) donné, on trouvera au maximum une Compétence (peut-être aucune).

Pour un couple (Projet, Compétence), on peut trouver plusieurs Salariés (ou aucun). Pour un couple, (Salarié, Compétences), on peut trouver plusieurs projets (ou aucun).

3.4 Navigabilité

La navigabilité indique qu'un objet de la classe cible peut être atteint par un objet de la classe source au travers de l'association. En d'autres termes, elle montre s'il est possible de traverser une association. Graphiquement la navigabilité est représentée par une flèche du côté de la terminaison navigable et on empêche la navigabilité par une croix du côté de la terminaison non navigable (cf. figure 8). Par défaut, une association est navigable dans les deux sens.

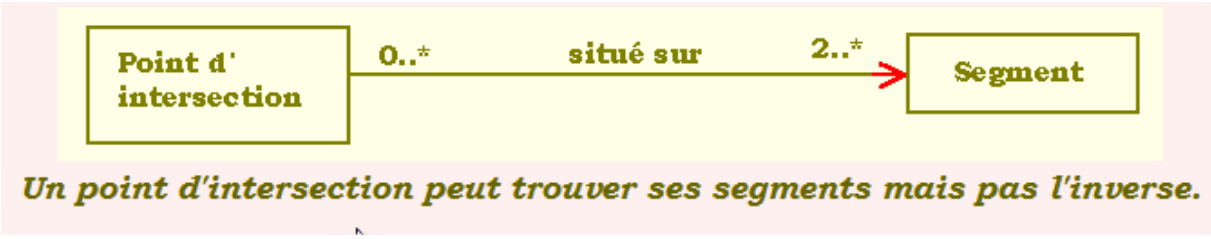


Figure 8.a : Navigabilité dans UML1 [2]

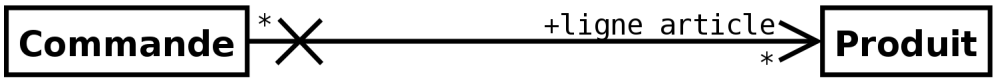


Figure 8.b : Navigabilité dans UML2

[2] Extrait de : http://www.iutc3.unicaen.fr/~moranb/cours/acsi/static2/stat9.htm

Par exemple, sur la figure 8.b, la terminaison du côté de la classe *Commande* n'est pas navigable : cela signifie que les instances de la classe *Produit* ne stockent pas de liste d'objets du type *Commande*. Inversement, la terminaison du côté de la classe *Produit* est navigable : chaque objet commande contient une liste de produits.

3.5 Contraintes d'association

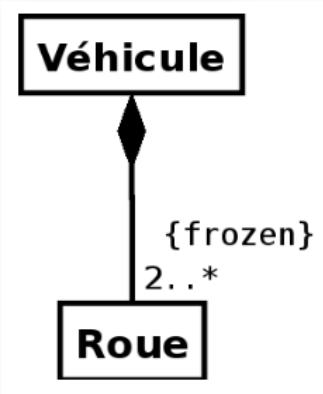
Toutes sortes de contraintes peuvent être définies sur une relation ou un groupe de relations. En effet, UML permet d'associer une contrainte à un, ou plusieurs, élément(s) de modèle de différentes façons :

- en plaçant directement la contrainte à côté d'une propriété ou d'une opération dans un classeur ;
- en ajoutant une note associée à l'élément à contraindre ;
- en plaçant la contrainte à proximité de l'élément à contraindre, comme une extrémité d'association par exemple ;
- en plaçant la contrainte sur une flèche en pointillés joignant les deux éléments de modèle à contraindre ensemble, la direction de la flèche constituant une information pertinente au sein de la contrainte ;
- en plaçant la contrainte sur un trait en pointillés joignant les deux éléments de modèle à contraindre ensemble dans le cas où la contrainte est bijective ;
- en utilisant une note reliée, par des traits en pointillés, à chacun des éléments de modèle, subissant la contrainte commune, quand cette contrainte s'applique sur plus de deux éléments de modèle.

Rappel : toutes les contraintes en UML s'expriment **entre accolades**.

Les contraintes portant sur les instances des classes, à une extrémité d'une association sont par exemple (cf. figure 9):

- {ordered} ou {ordonné} : Les objets doivent être ordonnés. Rq : on ne sait pas *comment* cela sera ordonné (c'est un choix de conception) ;
- {frozen} ou {gelé} : Un lien ne peut plus être modifié ni détruit après sa création ;
- {addOnly} : On ne peut qu'ajouter un objet, pas le détruire ;
- {subset} : indiquant qu'une occurrence fait partie obligatoirement partie des occurrences d'une autre association ;
- {xor} ou {ou-exclusif} : indique qu'une instance ne peut appartenir en même temps à la réalisation de deux associations.



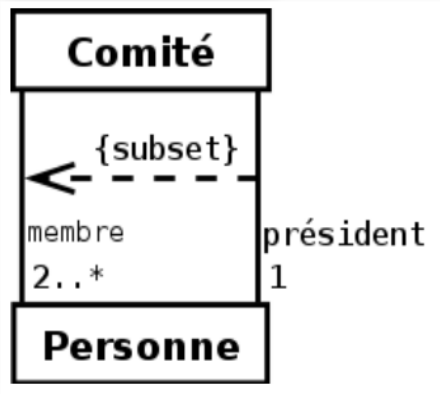


Figure 9a. Contrainte frozen **Figure 9b. Contrainte subset**

Lecture de la Figure 9b : un Comité possède plusieurs personnes (membres), mais au moins 2; le Comité possède un président qui fait nécessairement partie du Comité.

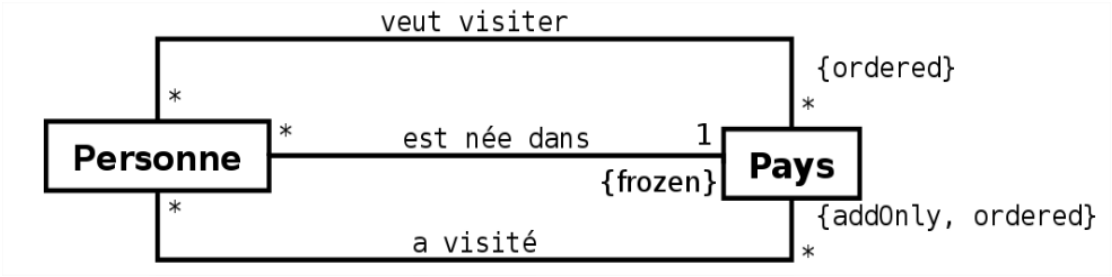


Figure 9c – Contraintes frozen, ordered et addOnly

Lecture de la Figure 9c : une Personne veut visiter ou pas des pays (ordre traduisant sa préférence) ; il est né dans un pays (qui ne changera jamais) ; il a visité un certain nombre de pays (peut-etre aucun), on ne peut qu'en ajouter, et ces pays visités sont ordonnés (selon sans doute leur date de visite).

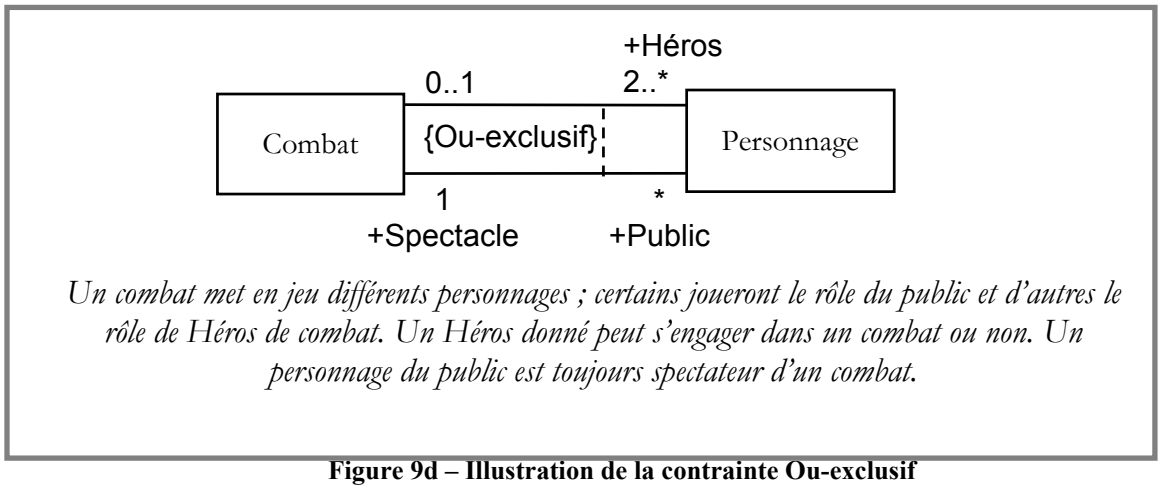


Figure 9d – Illustration de la contrainte Ou-exclusif

3.6 Qualification

Une qualification consiste à sélectionner un sous-ensemble d'objets parmi l'ensemble des objets qui participent à une relation. Quand une classe est liée à une autre classe par une association, il est parfois préférable de restreindre la portée de l'association à quelques éléments ciblés (comme un ou plusieurs attributs) de la classe. Ces éléments ciblés sont appelés un *qualificatif*. En d'autres termes, le qualificatif est un attribut spécial qui réduit la multiplicité effective d'une association. Les associations " un à plusieurs " et " plusieurs à plusieurs " peuvent être qualifiées. Le qualificatif caractérise l'ensemble d'objets à l'extrémité " plusieurs " (cf. figure 10).

Un objet qualifié et une valeur de qualificatif génèrent un objet cible lié unique (ou de multiplicité donnée, ici 2). En considérant un objet qualifié (dans l'exemple, la banque qualifiée d'un numéro de compte donné), chaque valeur de qualificatif désigne un objet cible quasi unique (un client).

Figure 10 : Illustration d'association qualifiée

Dans cette figure, le diagramme de gauche représente la relation existant entre une banque et ses clients ; à droite, le diagramme traduit que :

- Un compte dans une banque appartient à au plus deux personnes (compte joint). Autrement dit, une instance du couple {*Banque , compte*} est en association avec zéro à deux instances de la classe *Personne*.
- Mais une personne peut posséder plusieurs comptes dans plusieurs banques. C'est-à-dire qu'une instance de la classe *Personne* peut être associée à plusieurs (zéro compris) instances du couple {*Banque , compte*}.
- Bien entendu, et dans tous les cas, un instance du couple {*Personne , compte*} est en association avec une instance unique de la classe *Banque*.

3.7 Classe d'association

Il est parfois nécessaire de compléter une association par des attributs qui caractérisent la nature de la relation existant entre 2 classes. Cela peut être représenté par une classe d'association, à laquelle on ajoute des attributs. Par exemple, l'association *S'engage dans* entre un Héros et un Combat possède comme propriétés une prime et un niveau de challenge. En effet, ces deux propriétés n'appartiennent ni à Héros, qui peut avoir plusieurs primes, ni au

Combat, qui peut avoir plusieurs niveaux de challenge. Il s'agit donc bien de propriétés de l'association. Les associations ne pouvant posséder de propriété, il faut introduire un nouveau concept pour modéliser cette situation : celui de *classe d'association*.

Une *classe d'association* possède les caractéristiques des associations et des classes : elle se connecte à deux ou plusieurs classes et possède également des attributs et des opérations.

La notation UML utilise une ligne pointillée pour attacher une classe à une association (cf. figure 11).

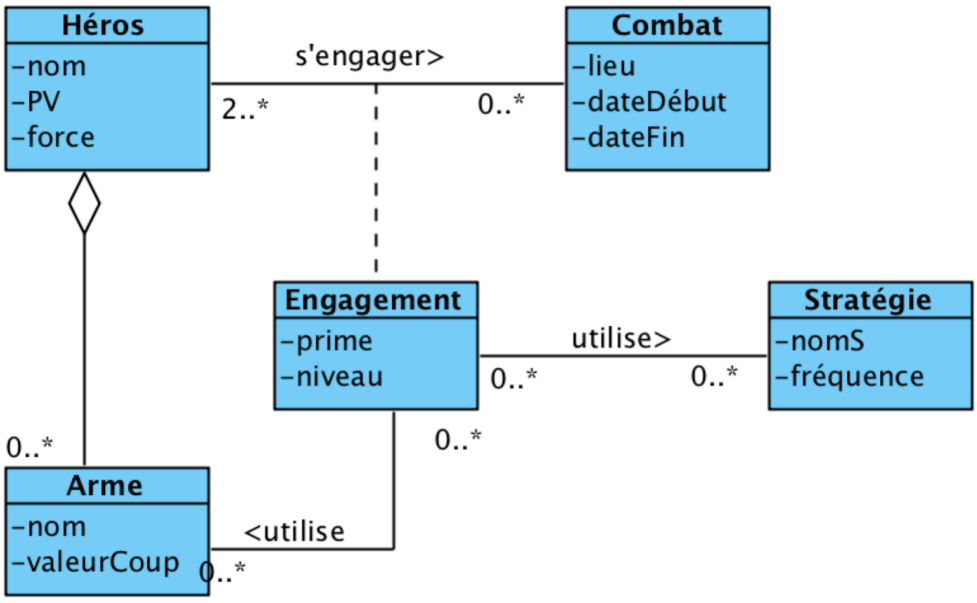


Figure 11 : Exemple de classe d'ssociation

Lecture de la Figure11 : un combat concerne au moins 2 héros ou plus. Pour chaque couple (unHéros, unCombat) on connaît la prime et le niveau de l'engagement du héros sur ce combat. Un engagement donné peut utiliser des stratégies et des armes. Ici une instance d'engagement a des valeurs pour les attributs *prime* et *niveau*, ainsi que les identifiants des instances qu'il relie (ideHéros et idCombat).

Rappel : on ne mentionne pas les identifiants dans les diagrammes de classes en phase d'Analyse.

Classe d'ssociation pour plusieurs associations

Une classe d'association NE PEUT ETRE rattachée à plus d'une association puisque la classe d'association forme elle-même l'association. Dans le cas où plusieurs classes d'association doivent disposer des mêmes caractéristiques, elles doivent hériter d'une même classe possédant ces caractéristiques, ou l'utiliser en tant qu'attribut.

De même, nous ne pouvons rattacher une instance de la classe d'une classe d'association à plusieurs instances de l'association. En effet, une classe d'association est une entité sémantique atomique et non composite, qui s'instancie en conséquence en bloc.

Auto-association sur classe d'association

Il est possible d'ajouter une **association réflexive** sur une classe d'association dans un diagramme. Par exemple, nous ajoutons une association « *Supérieur de »* sur le figure 12 pour préciser qu'une personne est le supérieur d'une autre personne. Nous ne pouvons dans ce cas ajouter simplement une association réflexive sur la classe *Personne*. En effet, une personne n'est pas le supérieur d'une autre dans l'absolu. Une personne est, en tant qu'employé d'une entreprise donné, le supérieur d'une autre personne dans le cadre de son emploi pour une entreprise donnée (généralement, mais pas nécessairement, la même). Il s'agit donc d'une association réflexive, non pas sur la classe *Personne* mais sur la classe d'association *Emploie*.

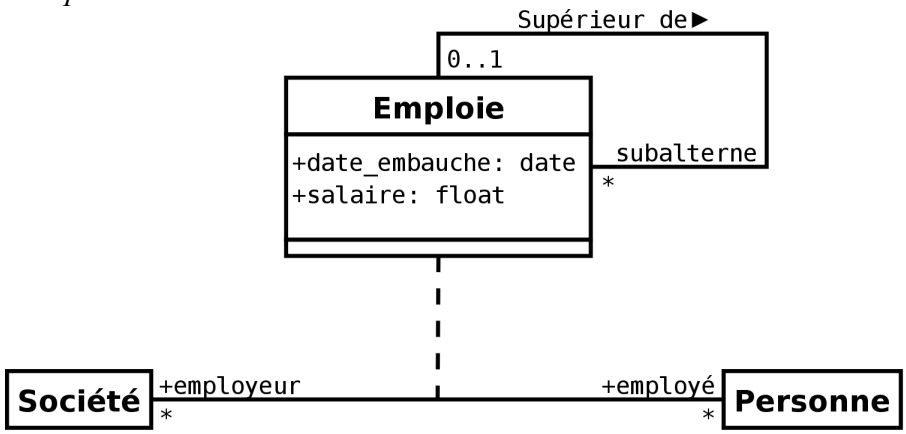


Figure 12 : Exemple d'auto-association sur classe-association.

Liens multiples

Plusieurs instances d'une même association ne peuvent lier les mêmes objets. Cependant, si l'on s'intéresse à l'exemple de la figure 13, on voit bien qu'il doit pouvoir y avoir plusieurs instances de la classe d'association *Actions* liant une même personne à une même société : une même personne peut acheter à des moments différents des actions d'une même société.

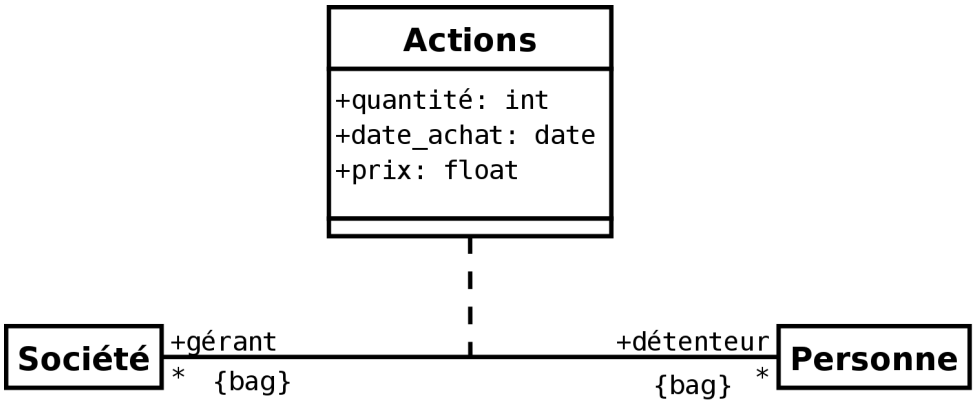


Figure 13 : Exemple de classe d'association avec liens multiples.

A cette fin, il est indispensable d'ajouter la contrainte *{bag}* placée sur les terminaisons d'association de la classe d'association *Actions*, afin d'indiquer qu'il peut y avoir des liens multiples impliquant les mêmes paires d'objets.

3.8 Généralisation et Héritage

Dans le langage UML, ainsi que dans la plupart des langages objet, il est possible d'effectuer des abstractions puissantes qui permettent de partager les points communs entre les classes tout en préservant leurs différences. Cette relation permet en d'autres termes de généraliser une *classe spécialisée* (sous-classe) en une classe de base dite aussi *classe parent*. La classe spécialisée est intégralement cohérente avec la classe de base, mais comporte des informations supplémentaires (attributs, opérations, associations). Un objet de la classe spécialisée peut être utilisé partout où un objet de la classe de base est autorisé.

Définition / Terminologie : la *généralisation* est une factorisation des éléments communs d'un ensemble de classes dits *sous-classes* dans une classe plus générale dite *super-classe.* Elle signifie que la sous-classe *est un* ou *est une sorte de* la super-classe. Le lien inverse est appelé *spécialisation*.

Cette abstraction en hiérarchie de classes est aussi connue sous le nom d'**Héritage**.

Le symbole utilisé pour la relation d'héritage ou de généralisation est une flèche blanche avec un trait plein, dont la pointe est un triangle du côté de la classe parent (cf. figure 14a).

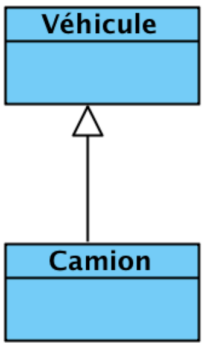


Figure 14a : Relation d'héritage ou de généralisation

Lectures possibles : un Camion 'est une sorte de' Véhicule ; un Camion est un Véhicule spécial.

Les propriétés principales de l'héritage sont :

- La classe enfant possède toutes les caractéristiques des ses classes parents, mais elle peut en avoir en d'autres qui lui sont propres ;
- Une classe enfant ne peut accéder aux caractéristiques *privées* des classes parent.
- Une classe enfant peut redéfinir (même signature) une ou plusieurs méthodes de la classe parent. Un objet utilise les opérations de la classe dont il dépend, pas celles des classes parent.

- Une classe enfant hérite des **attributs**, des **opérations** et des **associations** des classes parent.
- L'instance d'une classe peut être utilisée partout où une instance de sa classe parent est attendue (principe de substitution de Liskov) ;
- Une classe peut avoir plusieurs parents, on parle alors d'héritage multiple[3].

L'héritage permet notamment la classification des objets (cf. figure 14b).

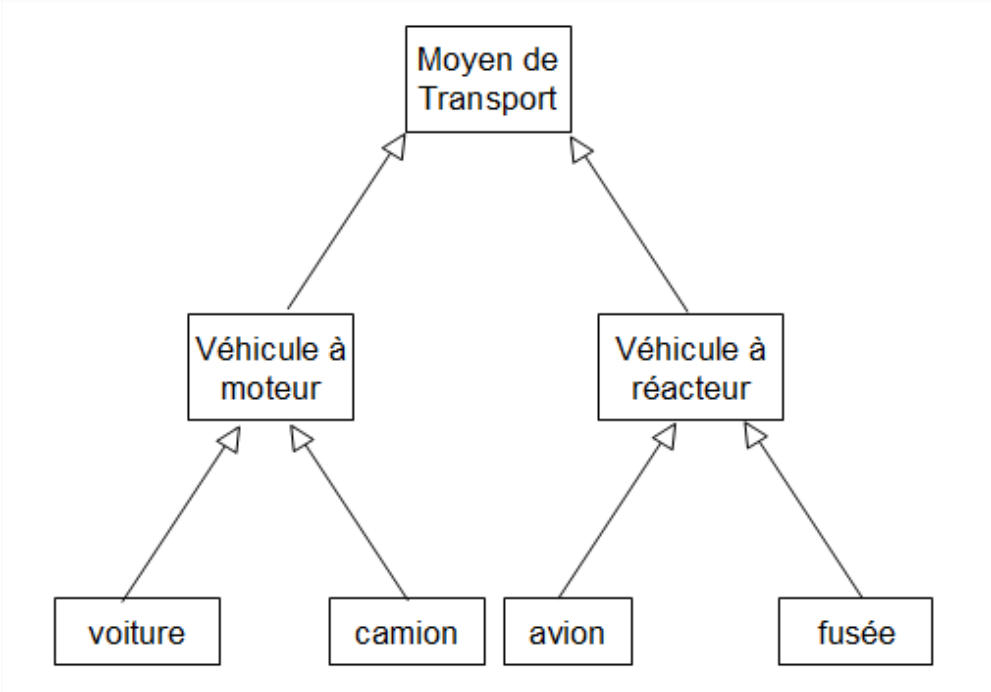


Figure 14b : Exemples d'héritage

3.9 Contraintes de classes

Comme pour les contraintes d'associations, il est possible d'enrichir les classes par des contraintes particulièrement en cas d'héritage. Nous citons par exemple les contraintes de :

- Discriminant + contrainte **{inclusif}** : indiquant des sous-classes non disjointes

  Exemple : véhicules à voile / mécanique ; Véhicules nautiques ou terrestres
- Contrainte **{exclusif}** : indiquant des sous-classes disjointes

  Exemple : véhicules à essence diesel, avec plomb, ou GPL
- Contrainte **{complète}** vs. **{incomplète}** : indiquant si l'arborescence est définitive ou extensible (cf. figure 15). Dans cet exemple, on signifie que d'autres sous-classes pourront être ajoutées ultérieurement (ex. vélo, moto).

[3] Le langage C++ est un des langages objet permettant son implémentation effective, le langage Java ne le permet pas car l'héritage multiple est source d'erreurs en programmation (conflits entre classes parent).

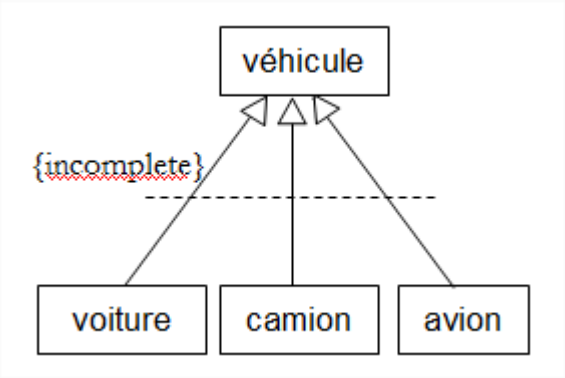


Figure 15 : exemple de contrainte d'arborescence extensible

3.10 Le polymorphisme

Par définition, le polymorphisme est un mécanisme qui permet à des objets partageant une interface commune d'en réaliser les opérations de façon propre. Chaque sous-classe peut modifier localement le comportement de ses opérations pour considérer le particularisme de son niveau d'abstraction. En d'autres termes, c'est un mécanisme qui permet à un objet client d'utiliser les services d'une interface « générique », sans savoir quel est l'objet qui va véritablement répondre à la requête.

C'est la caractéristique d'un élément « qui peut prendre plusieurs formes » (cf. figure 16).

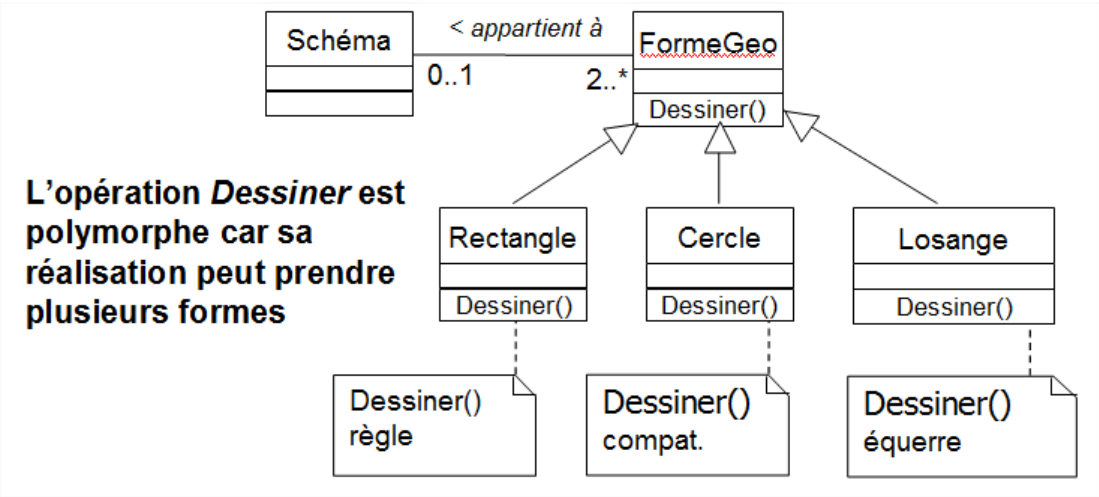


Figure 16 : concept de polymorphisme

3.11 Composition et Agrégation

Il est des cas particuliers d'associations qui amènent une sémantique plus forte dans un diagramme de classes, mais qui peuvent poser des problèmes. En effet, une association simple entre deux classes représente une relation structurelle symétrique où les deux classes sont du même niveau conceptuel : aucune des deux n'est plus importante que l'autre.

Si l'on souhaite représenter une association portant les sémantiques asymétriques suivantes : *composé et composants*, *maître et esclaves*, ou encore *tout et parties*, il existe 2 solutions avec UML.

Composition et **agrégation** expriment toute deux une relation transitive anti-symétrique. Globalement, on utilisera ces relations dans les cas suivants :

- Quand une classe fait partie d'une autre classe (*composé et composants*) ;
- Quand les valeurs d'attributs d'une classe se propagent dans les valeurs d'attributs d'une autre classe (*dépendance par les propriétés*) ;
- Quand une action sur une classe entraîne une action sur une autre classe (*délégation d'opération*) ;
- Quand les objets d'une classe sont subordonnés à ceux d'une autre classe (*dépendance de classes*).

La composition

La composition, représentée par un losange noir, représente une relation Composé/composants avec une dépendance des cycles de vie des objets, et qui stipule que l'objet « composé » ne peut appartenir qu'à un seul tout. Ainsi les parties d'une composition naissent et meurent avec l'objet propriétaire. Par exemple, les pièces entretiennent une relation de composition avec un bâtiment. Si on rase le bâtiment, on détruit les pièces (figure 17).

La composition décrit donc une contenance *structurelle* entre les instances. Ainsi, la destruction de l'objet composite implique la destruction de ses composants. Une instance de la partie appartient toujours à *au plus une* instance de l'élément composite : la multiplicité du côté composite ne doit pas être supérieure à 1 (*i.e.* 1 ou 0..1). Graphiquement, on ajoute un losange plein (♦) du côté du tout (l'agrégat).

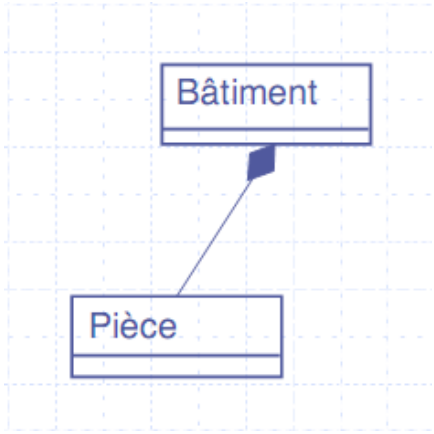


Figure 17 : Exemple de relation de composition

L'Agrégation

Analogiquement, on parle d'agrégation, **relation moins forte que la composition**, quand les objets « partie de » sont juste référencés par l'objet Agrégat, qui peut y accéder, mais n'en est pas propriétaire. Il en a juste la connaissance, et sans doute il en a la responsabilité (par ex.

une équipe est un agrégat de joueurs, elle sait qu'elle est complète quand cet agrégat possède 11 joueurs). Dans une agrégation il y indépendance des cycles de vie des objets : ainsi on peut dire qu'un train est constitué d'une série de wagons, mais ces wagons peuvent être employés pour former d'autres trains. Si le train est démantelé, les wagons existent toujours. Ainsi, lorsque l'on souhaite modéliser une relation *tout/partie* où une classe constitue un élément plus grand *(tout)* composé d'éléments plus petit *(partie),* il faut utiliser une agrégation.

Graphiquement, on ajoute un losange vide (◇) du côté de l'agrégat. Contrairement à une association simple, l'agrégation est transitive (figure 18).

Le sens de cette forme simple d'agrégation est simplement conceptuel. Elle ne contraint pas la navigabilité ou les multiplicités de l'association. **Elle n'entraîne pas de contrainte sur la durée de vie des parties par rapport au tout**.

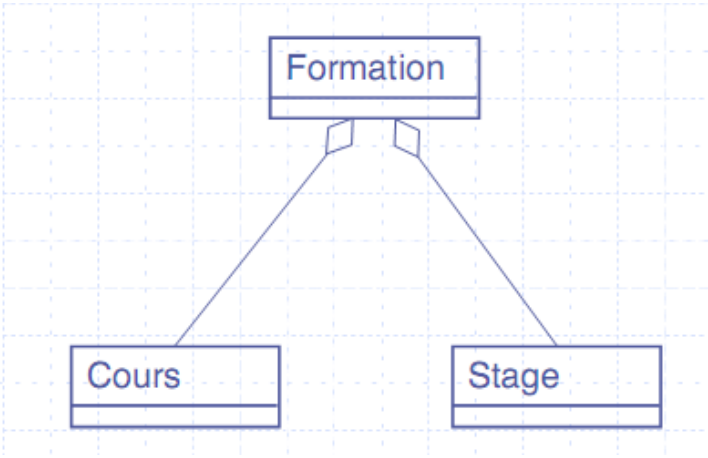


Figure 18 : Exemple de relation d'agrégation

La confusion entre composition et agrégation illustre parfaitement la relative perméabilité entre l'analyse et la conception. Devant la difficulté à décider de la nature d'une relation, décision qui relève pourtant de l'analyse, on s'appuie généralement sur la conception pour fixer son choix. En pratique, on se demande si l'objet « partie de » peut ou doit être détruit lorsqu'on détruit l'objet qui le contient et, si la réponse est affirmative, on choisit une relation de composition.

Remarque

Les notions d'agrégation et surtout de composition posent de nombreux problèmes en modélisation et sont souvent le sujet de querelles d'experts et de confusions. Ce que dit la norme *UML Superstructure version 2.1.1* reflète d'ailleurs très bien ce flou : « *Precise semantics of shared aggregation varies by application area and modeler. The order and way in which part instances are created is not defined".* On ne perdra donc pas de temps en analyse pour distinguer ces 2 notions, la relation d'agrégation (losange blanc), qui est plus souple que la composition, suffit généralement pour traduire ces notions de composé/composant ou maître/esclave.

Premier résumé des Concepts du DCL

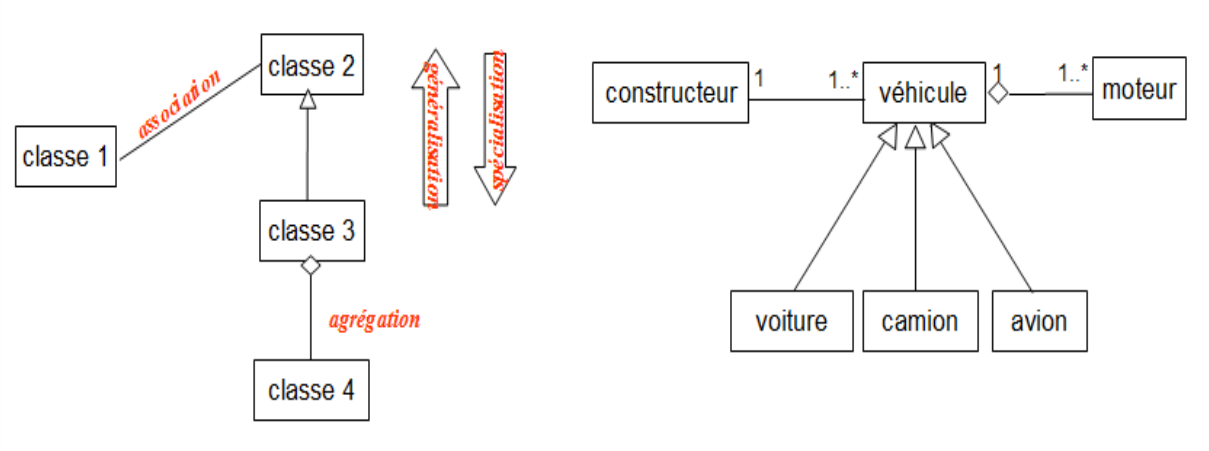


Figure 19 : Principaux concepts du diagramme de classes

3.12 Interface

Selon les principes de l'encapsulation, l'interface est la vue externe d'un objet : elle définit les services accessibles (offerts) aux utilisateurs de l'objet, qui ne savent pas comment ces services sont effectués.
Quand on mentionne une interface dans un diagramme de classe, il peut s'agir de l'interface complète d'un objet, ou simplement d'une partie d'interface qui sera commune à plusieurs objets (figure 20).

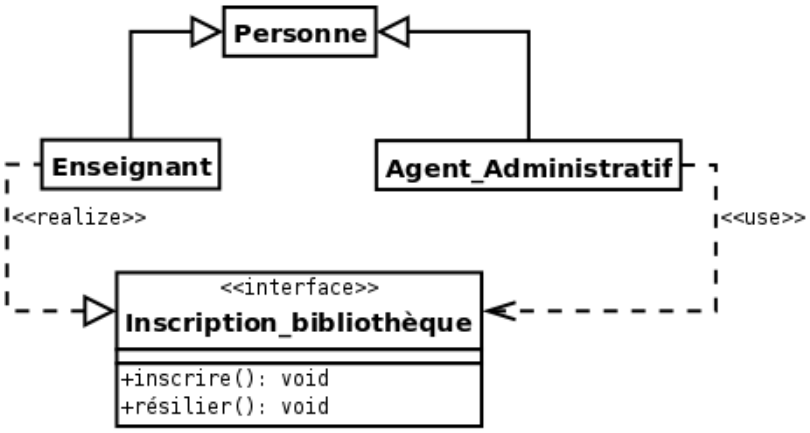


Figure 20 – Un exemple de classe « interface »

Le rôle de cette classe, stéréotypée « interface », est de regrouper un ensemble de propriétés et d'opérations assurant un service cohérent. L'objectif est de diminuer le couplage entre deux classes. La notion d'interface en UML est très proche de la notion d'interface en Java.

Une interface doit être réalisée par *au moins* une classe, et peut l'être par plusieurs. Graphiquement, on représente cela par une relation « de Réalisation », *i.e.* un trait discontinu terminé par une flèche triangulaire et le stéréotype « realize ». Une classe peut implémenter (réaliser) plusieurs interfaces. Une classe cliente peut d'autre part utiliser une interface (elle utilise l'un des services offerts par l'interface). On représente cela par une relation de

dépendance et le stéréotype « use ». Dans la figure 20, c'est la classe Enseignant qui **implémentera** les méthodes *inscrire()* et *résilier()* de l'interface Bibliothèque, alors que la classe Agent_Administratif **utilisera** ces méthodes (fera appel à elles).

La notion d'interface est proche de la notion de classe abstraite, avec une capacité de découplage plus grand ; en effet, le comportement d'une interface est purement abstrait, par contre une classe abstraite peut implémenter tout ou partie du comportement. En C++ (le C++ ne connaît pas la notion d'interface), la notion d'interface est généralement implémentée par une classe abstraite. En JAVA, où l'héritage multiple n'est pas autorisé, les interfaces permettent de spécifier qu'une classe « dépend » (hérite) de plusieurs autres classes.

Pour conclure, on retiendra que les classes abstraites servent à **factoriser** du code, tandis que les interfaces servent à définir des **contrats de service,** définis (implémentés) dans différentes classes concrètes.

3.13 Notion de Dépendance

Une dépendance est une relation unidirectionnelle formulant une dépendance sémantique entre des éléments du modèle (packages ou classes). Elle est représentée par un trait discontinu orienté (cf. figure 21). Elle indique que la modification de la cible peut impliquer une modification de la source.

La dépendance peut être stéréotypée (<< use >>, << import >>, etc.)) pour préciser le lien sémantique entre les éléments du modèle.

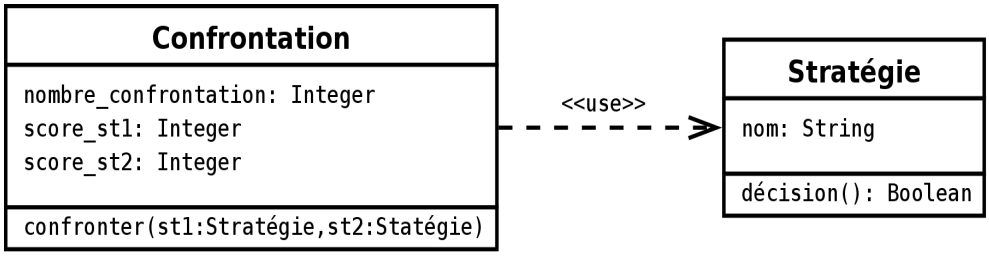


Figure 21 - Exemple de relation de dépendance

Le plus souvent, l'emploi d'une dépendance permet d'indiquer qu'une classe **utilise** une autre comme argument dans la signature d'une opération. La figure 20 montre que la classe *Confrontation* utilise la classe *Stratégie* car la classe *Confrontation* possède une méthode *confronter* dont deux paramètre sont du type *Stratégie*. Si la classe *Stratégie*, notamment son interface, change, alors des modifications devront également être apportées à la classe *Confrontation*.

4. Diagramme d'objets

4.1 Présentation

Un diagramme d'objets représente des objets (*i.e.* instances de classes) et leurs liens (*i.e.* instances de relations) pour donner une vue figée de l'état d'un système à un instant donné. Un diagramme d'objets peut être utilisé pour :

- illustrer le modèle de classes en montrant un exemple qui explique le modèle ;
- préciser certains aspects du système en mettant en évidence des détails imperceptibles dans le diagramme de classes ;
- exprimer une exception en modélisant des cas particuliers ou des connaissances non généralisables qui ne sont pas modélisés dans un diagramme de classe ;

Le diagramme de classes modélise les règles et le diagramme d'objets modélise des faits.

Par exemple, le diagramme de classes de la figure 22 montre qu'une *personne* jouant le rôle de *patron* emploie plusieurs (à cause de la multiplicité) *personne* comme étant des collaborateurs et qu'une personne travaille dans au plus deux entreprises. Le diagramme d'objets modélise lui une personne particulière (*ETIENNE*) qui emploie une personne collaboratrice (*JEAN-LUC*).

Un diagramme d'objets ne montre pas l'évolution du système dans le temps. Pour représenter une interaction, il faut utiliser un diagramme de séquence (cf. section diagrammes dynamiques).

4.2 Représentation

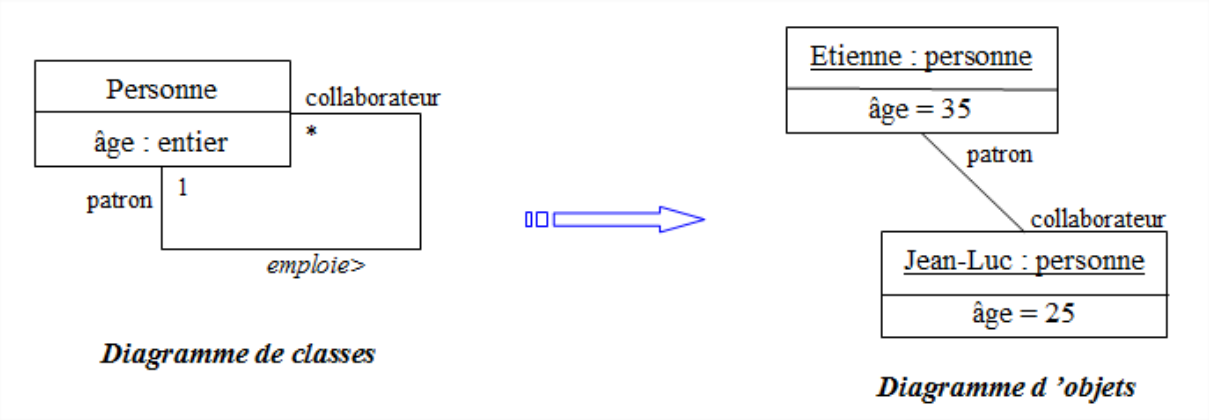


Figure 22 - Exemple de diagramme de classes et de diagramme d'objets associé.

Il est à noter que graphiquement, un objet se représente comme une classe. Toutefois, le compartiment des opérations n'est pas utile. Par ailleurs, pour distinguer les objets d'une même classe, leur identifiant peut être ajouté devant le nom de la classe. Enfin les attributs ont des valeurs. Quand certaines valeurs d'attribut d'un objet ne sont pas renseignées, on dit que l'objet est **partiellement défini**.

Dans un diagramme d'objets, les relations du diagramme de classes deviennent des liens. La relation de généralisation ne possède pas d'instance, elle n'est donc jamais représentée dans un diagramme d'objets. Graphiquement, un lien se représente comme une relation, mais, s'il y a un nom, il est souligné. Nous notons aussi que les multiplicités ne sont pas représentées.

5. Élaboration et implémentation d'un diagramme de classes

5.1 Élaboration d'un diagramme de classes

Une démarche couramment utilisée pour bâtir un diagramme de classes est la suivante :

1. Pour identifier les classes, souligner tous les *termes* décrivant le domaine applicatif (cours, salles, professeurs… pour une application gérant un Emploi du Temps) ;
2. Réduire l'ensemble obtenu avec les critères suivants :
   - supprimer des *synonymes*,
   - supprimer des classes *trop vagues* (elles sont surement ailleurs),
   - supprimer des classes non *pertinentes* pour l'application,
   - découvrir les *associations* exprimant l'interdépendance des classes, et les nommer ;
   - trouver les *attributs* des classes, vérifier que chaque attribut caractérise bien l'a classe dont on parle (pas d'identifiant Client dans une classe Commande !).
   - identifier les *opérations* des classes : association ou attribut ?
3. Inclure les relations de *généralisation/spécialisation*
4. Placer des *agrégations/compositions* si nécessaire
5. Factoriser avec des *interfaces*
6. Utiliser les *contraintes*
7. Réorganiser le diagramme de classes en construisant des *packages*
8. Vérifier *l'atteignabilité* de tous les éléments (se poser des questions sur les fonctionnalités utilisateurs : par ex. pour une application marchande, est-ce que je peux connaître les éléments de mon panier, en transitant par les différentes associations de mon DCL ?)
9. Valider le modèle obtenu avec les *utilisateurs*
10. Incrémenter le modèle (en itérant) : on n'obtient jamais le bon DCL dès le 1er coup ; un indicateur de bon diagramme peut être le suivant : compter les versions obtenues. A partir de la 3ème, on peut penser être proche d'un diagramme correct.

5.2 Illustration d'une implémentation du DCL en Java

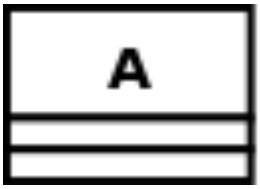
Dans la majorité des cas, le diagramme de classes sert à préparer l'implémentation dans un langage cible : vous construisez votre DCL à l'aide d'un AGL (Atelier de Génie Logiciel) et vous spécifiez le langage cible (Java, PHP5, C++, C#...). L'AGL utilisé à l'IUT est PowerAMC de SYBASE, il est payant –il existe toutefois une version de démonstration-, mais vous avez aussi Visual Paradigm, BOULM, ArgoUML et RationalModeler d'IBM), qui sont gratuits.

Ensuite vous générez le code obtenu : vous obtenez ainsi le squelette de vos classes, avec les constructeurs et les accesseurs (get/set) des attributs automatiquement implémentés, ce qui simplifie grandement le travail.

Dans la section suivante, on donne une illustration du type de code obtenu pour certains éléments du DCL.

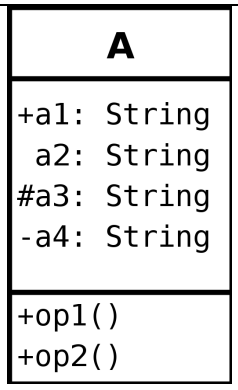
Classe

Parfois, la génération automatique de code produit, pour chaque classe, un constructeur et une méthode finalize comme ci-dessous.




```
public class A {
  public A() {
    ...
  }
  protected void finalize() throws
Throwable {
    super.finalize();
    ...
  }

}
```

Classe avec attributs et opérations

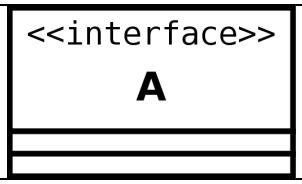



```
public class A {
  public    String a1;
            String a2; // portée Package
  protected String a3;
  private   String a4;

  public void op1() {
    ...
  }
  public void op2() {
    ...
  }
}
```

Classe abstraite

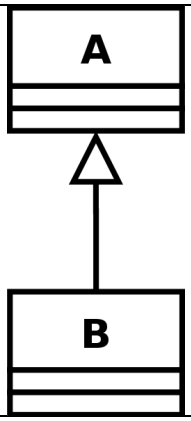

```
public abstract class A {
  ...

}
```

Interface



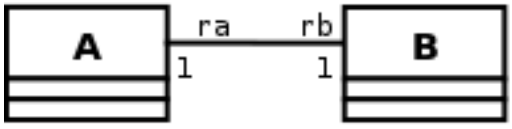

```
public interface A {
  ...

}
```

Héritage simple



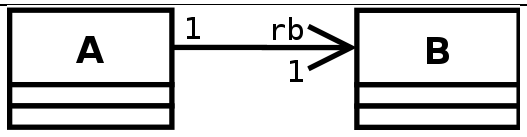

```
public class A {
  ...
}

public class B extends A {
  ...

}
```

Association bidirectionnelle 1 vers 1




```
public class A {
  private B rb;
  public void addB( B b ) {
    if( b != null ){
      if ( b.getA() != null ) {
// si b est déjà connecté à un autre A
        b.getA().setB(null) ;
// cet autre A doit se déconnecter
      }
      this.setB( b );
      b.setA( this );  }
  }
  public B getB() { return( rb ); }
  public void setB( B b ) {
     this.rb=b;    }
}
public class B {
  //   meme   code,   A   et   B   sont
symétriques
```

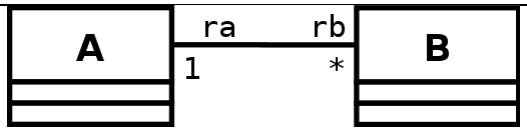
Association unidirectionnelle 1 vers 1




```
public class A {
  private B rb;
  public void addB( B b ) {
    if( b != null ) {
      this.rb=b;
```

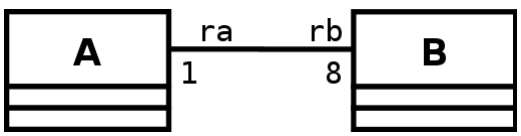
```
    }
  }
}

public class B {
 ... // rien : la classe B ne connaît
// pas l'existence de la classe A
}
```

Association bidirectionnelle 1 vers N



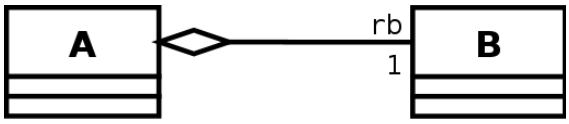

```
public class A {
  private ArrayList<B> rb;
  public A() {
    rb = new  ArrayList<B>();
  }
  public ArrayList<B> getArray() {
     return(rb);}
  public void remove(B b){
     rb.remove(b);}
  public void addB(B b){
    if( !rb.contains(b) ){
      if (b.getA()!=null)
        b.getA().remove(b);
        b.setA(this);
        rb.add(b);
    }
  }
}

public class B {
  private  A ra;
  public B() {}
  public A getA() { return (ra); }
  public void setA(A a){ this.ra=a; }
  public void addA(A a){
    if( a != null ) {
      if(  !a.getArray().contains(this))
{
       if (ra != null)
        ra.remove(this);
        this.setA(a);
        ra.getArray().add(this);
      }
    }
  }
}
```

Association 1 vers N



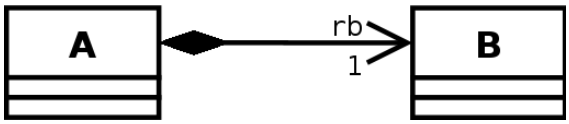
Dans ce cas, il faut utiliser un tableau plutôt qu'un vecteur. La dimension du tableau étant donnée par la cardinalité de la terminaison d'association.

Agrégations



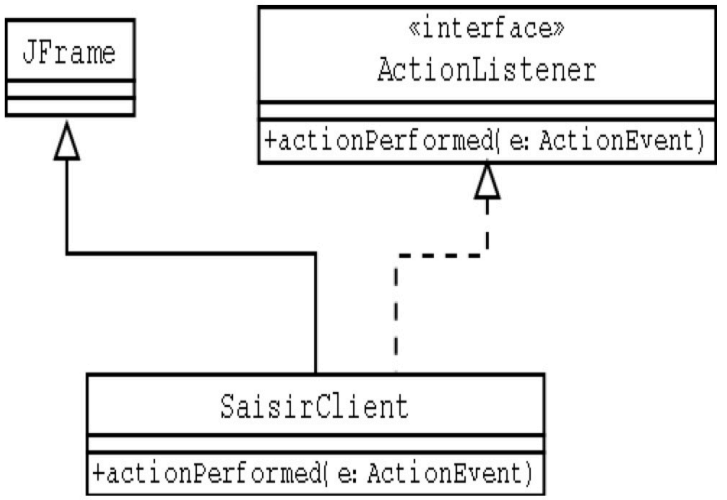
Les agrégations s'implémentent comme les associations.

Composition



Une composition peut s'implémenter comme une association unidirectionnelle. On aura simplement à implémenter des règles de création/ suppression dans les constructeurs, de façon à *lier les cycles de vie* des parties avec le tout : en cas de suppression d'une instance de A, supprimer l'instance de B associée ; lors de la création d'un objet A, créer aussi l'objet B associé.

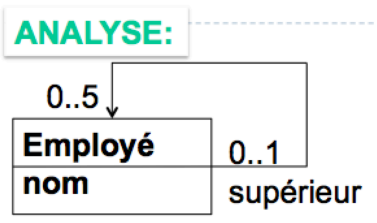
Relation d'implémentation

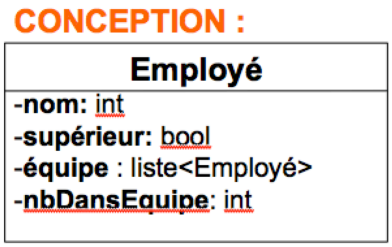



```
public SaisirClient extends
JFrame implements
ActionListener {

  private  JButton boutonOK;
   // etc…

  public void
actionPerformed(ActionEvent
e){
 // …


  }
}
```

Implémentation d'une association réflexive

Soit le cas où on modélise le fait qu'un employé d'entreprise peut être le supérieur d'autres employés (au max 5), ou pas :





Implémentation Java :

```
public class Employé {
  private String nom;
  private boolean supérieur;
  private ArrayList<Employé>
équipe;
  private int nbDansEquipe;
      // entre 0 et 5
}
```

Explications :

Si supérieur = `true`, alors `équipe` contient entre 0 et 5 employés

Si supérieur = `false`, alors `équipe` est vide et `nbDansEquipe` vaut 0

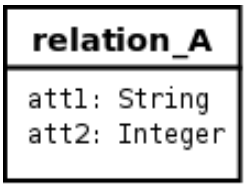
5.3 Illustration d'une implémentation du DCL en SQL

Certaines classes du diagramme de classes vont être persistantes, i.e. leurs données seront stockées dans des systèmes de fichiers ou des SGBD pour être réutilisées d'une exécution à l'autre. Dans la grande majorité des applications, on utilise des SGBD relationnels. Pour cela il est ainsi nécessaire de faire la correspondance Objet / Relationnel, et cela n'est pas une tâche facile. En effet, l'expressivité d'un diagramme de classes est bien plus grande que celle d'un schéma relationnel. Par exemple, comment représenter dans un schéma relationnel des notions comme la navigabilité ou la composition ?

Là aussi, l'AGL peut aider le concepteur, car il permet de générer automatiquement des scripts SQL permettant de construire les tables relationnelles à partir des classes persistantes du DCL. Bien entendu, les méthodes des classes ne sont pas traduites. On observera que pour la conception de base de données, il devient de plus en plus courant d'utiliser la modélisation UML plutôt que le traditionnel modèle entités-associations.

Les scripts ci-dessous datent de 2008.

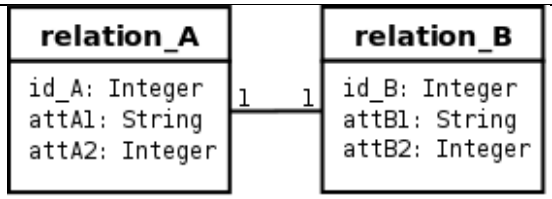
Classe avec attributs



Chaque classe devient une relation; s'il n'y a pas d'attribut, un numéro est crée en tant qu'identifiant primaire :

```
create table relation_A (
   num_relation_A integer primary key,
   att1 text,
   att2 integer);
```

Association 1 vers 1

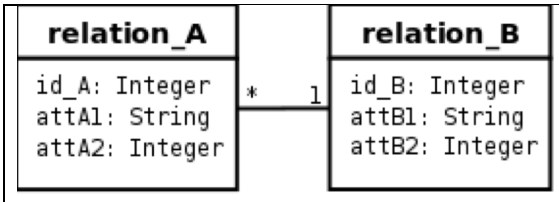


Pour représenter une association 1 vers 1 entre deux relations, la clé primaire de l'une des relations doit figurer comme clé étrangère dans l'autre relation :

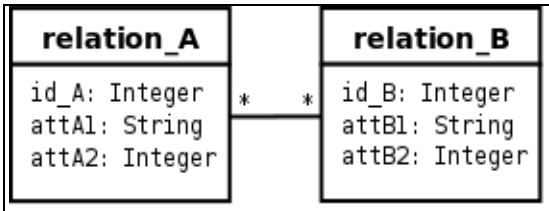
```
create table relation_A (
  id_A integer primary key,
  attA1 text,
  attA2 integer);

create table relation_B (
```

	id_B integer primary key, num_A integer references relation_A, attB1 text, attB2 integer);

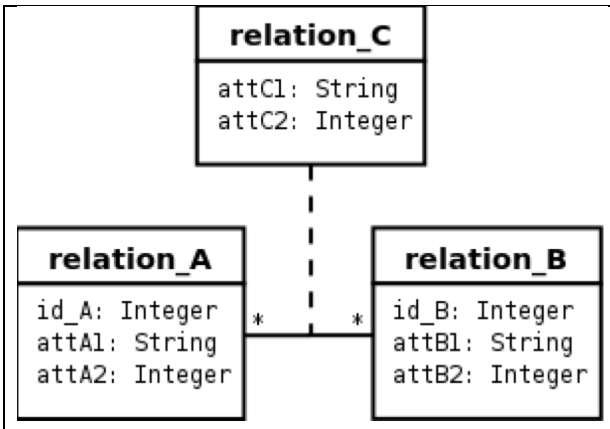
Association 1 vers plusieurs

	Pour représenter une association 1 vers plusieurs, on procède comme pour une association 1 vers 1, excepté que c'est la relation du côté *plusieurs* qui reçoit la clé primaire de la relation du côté 1, en clé étrangère : create table relation_A (id_A integer primary key, num_B integer references relation_B, attA1 text, attA2 integer); create table relation_B (id_B integer primary key, attB1 text, attB2 integer);

Association plusieurs vers plusieurs

	Pour représenter une association du type plusieurs vers plusieurs, il faut introduire une nouvelle relation dont les attributs sont les clés primaires des relations en association et dont la clé primaire est la concaténation de ces deux attributs : create table relation_A (id_A integer primary key, attA1 text, attA2 integer); create table relation_B (id_B integer primary key, attB1 text, attB2 integer); create table relation_A_B (num_A integer references relation_A, num_B integer references relation_B, primary key (num_A, num_B));

Classe d'association plusieurs vers plusieurs

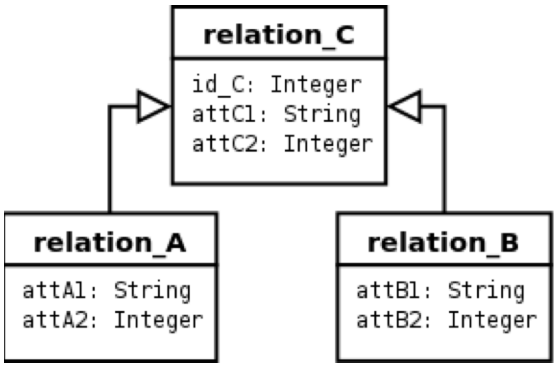


Le cas est proche de celui d'une association plusieurs vers plusieurs, les attributs de la classe d'association étant ajoutés à la 3ème relation qui représente, cette fois ci, la classe d'association elle-même :

```
create table relation_A (
  id_A integer primary key,
  attA1 text,
  attA2 integer);

create table relation_B (
  id_B integer primary key,
  attB1 text,
  attB2 integer);

create table relation_A_B (
  num_A integer references
relation_A,
  num_B integer references
relation_B,
  attC1 text,
  attC2 integer,
  primary key (num_A, num_B));
```

Héritage

Les relations correspondant aux sous-classes ont comme clé étrangère et primaire la clé de la relation correspondant à la classe parente. Un attribut type est ajouté dans la relation correspondant à la classe parente. Cet attribut permet de savoir si les informations d'un tuple de la relation correspondant à la classe parente peuvent être complétées par un tuple de l'une des relations correspondant à une sous-classe, et, le cas échéant, de quelle relation il s'agit. Ainsi, dans cette solution, un objet peut avoir ses attributs répartis dans plusieurs relations. Il faut donc opérer des jointures pour reconstituer un objet. L'attribut type de la relation correspondant à la classe parente doit indiquer quelles jointures faire.




```
create table relation_C (
   id_C integer primary key,
   attC1 text,
   attC2 integer,
   type text);

create table relation_A (
   id_A references relation_C,
   attA1 text,
   attA2 integer,
   primary key (id_A));

create table relation_B (
   id_B references relation_C,
   attB1 text,
   attB2 integer,
   primary key (id_B));
```

Une alternative à cette représentation est de ne créer qu'une seule table par arborescence d'héritage. Cette table doit contenir tous les attributs de toutes les classes de l'arborescence plus l'attribut type dont nous avons parlé ci-dessus. L'inconvénient de cette solution est qu'elle implique que les tuples contiennent de nombreuses valeurs nulles.

```
create table relation_ABC (
   id_C integer primary key,
   attC1 text, attC2 integer,
   attA1 text, attA2 integer,
   attB1 text, attB2 integer,
   type text);
```

Bibliographie

LIVRES

- P.-A. Muller, *Modélisation objet avec UML*, Eyrolles, 1998.
- P. Roques et F. Vallée, *UML en action – De l'analyse des besoins à la conception en Java*, Eyrolles, 2000.
- Laurent Audibert, *UML 2 - de l'apprentissage à la pratique (cours et exercices),* Collection Info+, Editions Ellipses, 2009.
- Grady Booch, James Rumbaugh and Ivar Jacobson, *An introduction to Object-Oriented Analysis and Design and the Unified Process*, Prentice Hall, 1997.

SITES INTERNET

Quelques sites de référence :

- www.omg.org : site de l'**OMG**
- www.uml.org/ : site officiel d'**UML**
- http://www-01.ibm.com/software/rational/ : site de **Rational** Software Corporation, maintenant racheté par IBM
- http://uml.free.fr/ : **cours en français** (de L. Piechocki)

OUTILS DE MODELISATION GRATUITS / AGL

- VisualParadigm for UML :
  http://www.visual-paradigm.com/product/vpuml/
- ArgoUML open-source (MacOS, Windows 64bits):
  http://www.zdnet.fr/telecharger/logiciel/argouml-mac-os-x-39829535s.htm
- Rational Modeler d'IBM, free UML modelling tool :
  http://www-01.ibm.com/software/awdtools/modeler/
- BOUML (v6 Sept. 2016) de Bruno Pagès : http://www.bouml.fr/
  Tourne sur Windows et Linux, requiert peu de mémoire
- Un AGL free écrit en Java : http://www.umlet.com/