UML 2

De l'apprentissage à la pratique

Table des matières

Précédent Sommaire Suivant

3. Chapitre 3 Diagramme de classes (Class Diagram)

3-1. Introduction

Le diagramme de classes est considéré comme le plus important de la modélisation orientée objet, il est le seul obligatoire lors d'une telle modélisation.

Alors que le diagramme de cas d'utilisation montre un système du point de vue des acteurs, le diagramme de classes en montre la structure interne. Il permet de fournir une représentation abstraite des objets du système qui vont interagir pour réaliser les cas d'utilisation. Il est important de noter qu'un même objet peut très bien intervenir dans la réalisation de plusieurs cas d'utilisation. Les cas d'utilisation ne réalisent donc pas une partition(7) des classes du diagramme de classes. Un diagramme de classes n'est donc pas adapté (sauf cas particulier) pour détailler, décomposer, ou illustrer la réalisation d'un cas d'utilisation particulier.

Il s'agit d'une vue statique, car on ne tient pas compte du facteur temporel dans le comportement du système. Le diagramme de classes modélise les concepts du domaine d'application ainsi que les concepts internes créés de toutes pièces dans le cadre de l'implémentation d'une application. Chaque langage de Programmation orienté objet donne un moyen spécifique d'implémenter le paradigme objet (pointeurs ou pas, héritage multiple ou pas, etc.), mais le diagramme de classes permet 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.

3-2. Les classes

3-2-1. Notions de classe et d'instance de classe

Une *instance* est une concrétisation d'un concept abstrait. Par exemple :

- la Ferrari *Enzo* qui se trouve dans votre garage est une instance du concept abstrait *Automobile ;*
- l'amitié qui lie Jean et Marie est une instance du concept abstrait *Amitié ;*

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 ou services),
- des composants d'une application (ex. : les boutons des boîtes de dialogue),
- des structures informatiques (ex. : des tables de hachage),
- des éléments comportementaux (ex. : des tâches), etc.

Tout système orienté objet est organisé autour des classes.

Une classe est la description formelle d'un ensemble d'objets ayant une sémantique et des caractéristiques communes.

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.

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

3-2-2. Caractéristiques d'une classe

Une classe définit un jeu d'objets dotés de caractéristiques communes. Les caractéristiques d'un objet permettent de spécifier son *état* et son *comportement*. Dans les sections 1.3.2 et 1.3.4, nous avons dit que les caractéristiques d'un objet étaient soit des attributs, soit des opérations. Ce n'est pas exact dans un diagramme de classe, car *les terminaisons d'associations sont des propriétés qui peuvent faire partie des caractéristiques d'un objet au même titre que les attributs et les opérations* (cf. section 3.3.2).

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*(8), qui décrivent l'état d'un objet. Les attributs sont utilisés pour des valeurs de données pures, dépourvues d'identité, telles que les nombres et les chaînes de caractères. Les associations sont utilisées pour connecter les classes du diagramme de classe ; dans ce cas, la terminaison de l'association (du côté de la classe cible) est généralement une propriété de la classe de base (cf. section 3.3.1 et 3.3.2).

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 :

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.

Une opération est la spécification (*i.e.* déclaration) d'une méthode. L'implémentation (*i.e.* définition) d'une méthode est également appelée méthode. Il y a donc une ambiguïté sur le terme *méthode*.

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

3-2-3. Représentation graphique

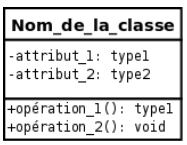


Figure 3.1 : Représentation UML d'une classe.

Une classe est un classeur(9). Elle est représentée par un rectangle divisé en trois à cinq compartiments (figure 3.1).

Le premier indique le nom de la classe (cf. section 3.2.5), le deuxième ses attributs (cf. section 3.2.6) et le troisième ses opérations (cf. section 3.2.7). Un compartiment des responsabilités peut être ajouté pour énumérer l'ensemble de tâches devant être assurées par la classe, mais pour lesquelles on ne dispose pas encore assez d'informations. Un compartiment des exceptions peut également être ajouté pour énumérer les situations exceptionnelles devant être gérées par la classe.

3-2-4. Encapsulation, visibilité, interface

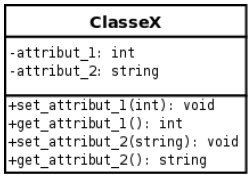


Figure 3.2 : Bonnes pratiques concernant la manipulation des attributs.

Nous avons déjà abordé cette problématique section 1.3.4. L'encapsulation est un mécanisme consistant à rassembler les données et les méthodes au sein d'une structure en cachant l'implémentation de l'objet, c'est-à-dire en empêchant l'accès aux données par un autre moyen que les services proposés. Ces services accessibles (offerts) aux utilisateurs de l'objet définissent ce que l'on appelle l'interface de l'objet (sa vue externe). L'encapsulation permet donc de garantir l'intégrité des données contenues dans l'objet.

L'encapsulation permet de définir des niveaux de visibilité des éléments d'un conteneur. La visibilité déclare la possibilité pour un élément de modélisation de référencer un élément qui se trouve dans un espace de noms différent de celui de l'élément qui établit la référence. Elle fait partie de la relation entre un élément et le conteneur qui l'héberge, ce dernier pouvant être un paquetage, une classe ou un autre espace de noms. Il existe quatre visibilités prédéfinies.

***Public* ou + :**

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.

Par ailleurs, UML 2.0 donne la possibilité d'utiliser n'importe quel langage de programmation pour la spécification de la visibilité.

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.

Dans un paquetage, le marqueur de visibilité se situe sur des éléments contenus directement dans le paquetage, comme les classes, les paquetages imbriqués, etc. Il indique si un autre paquetage susceptible d'accéder au premier paquetage peut voir les éléments.

Dans la pratique, lorsque des attributs doivent être accessibles de l'extérieur, il est préférable que cet accès ne soit pas direct, mais se fasse par l'intermédiaire d'opérations (figure 3.2).

3-2-5. Nom d'une 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 :

Sélectionnez

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

3-2-5-a. Métalangage des syntaxes

Nous aurons régulièrement recours à ce métalangage pour décrire des syntaxes de déclaration. Ce métalangage contient certains métacaractères :

`[ ]` **:**

les crochets indiquent que ce qui est à l'intérieur est optionnel ;

`< >` **:**

les signes inférieur et supérieur indiquent que ce qui est à l'intérieur est plus ou moins libre ; par exemple, la syntaxe de déclaration d'une variable comme `compteur : int` est `<nom_variable> : <type> ;`

`' '` **:**

les cotes sont utiles quand on veut utiliser un métacaractère comme un caractère ; par exemple, pour désigner un crochet (`[`) il faut écrire `'[»`, car `[` est un métacaractère ayant une signification spéciale ;

`…` **:**

permet de désigner une suite de séquence de longueur non définie, le contexte permettant de comprendre de quelle suite il s'agit.

3-2-6. Les attributs

3-2-6-a. 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. Chacune de ces informations est définie par un nom, un type de données, une visibilité et peut être initialisée. Le nom de l'attribut doit être unique dans la classe. La syntaxe de la déclaration d'un attribut est la suivante :

Sélectionnez

```
<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 donnée 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*}).

3-2-6-b. 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.

3-2-6-c. Attributs dérivés

Les attributs dérivés peuvent être calculés à partir d'autres attributs et de formules de calcul. Lors de la conception, un attribut dérivé peut être utilisé comme marqueur jusqu'à ce que vous puissiez déterminer les règles à lui appliquer.

Les attributs dérivés sont symbolisés par l'ajout d'un « / » devant leur nom.

3-2-7. Les méthodes

3-2-7-a. Méthode de la classe

Dans une classe, une opération (même nom et mêmes types de paramètres) doit être unique. Quand le nom d'une opération apparaît plusieurs fois avec des paramètres différents, on dit que l'opération est 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 :

Sélectionnez

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

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

Sélectionnez

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

La direction peut prendre l'une des valeurs suivantes :

in :

paramètre d'entrée passé par valeur. Les modifications du paramètre ne sont pas disponibles pour l'appelant. C'est le comportement par défaut.

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 un type de donnée prédéfini.

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

3-2-7-b. Méthode de classe

Comme pour les attributs de classe, il est possible de déclarer des méthodes de classe. Une méthode de classe 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). L'accès à une méthode de classe ne nécessite pas l'existence d'une instance de cette classe.

Graphiquement, une méthode de classe est soulignée.

3-2-7-c. 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 parent (cf. section **3.3.9**) contient une méthode abstraite non encore réalisée.

On ne peut instancier une classe abstraite : elle est vouée à se spécialiser (cf. section **3.3.9**). 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.

Pour indiquer qu'une classe est abstraite, il faut ajouter le mot-clef *abstract* derrière son nom.

3-2-8. Classe active

Une classe est passive par défaut, elle sauvegarde les données et offre des services aux autres. Une classe active initie et contrôle le flux d'activités.

Graphiquement, une classe active est représentée comme une classe standard dont les lignes verticales du cadre, sur les côtés droit et gauche, sont doublées.

3-3. Relations entre classes

3-3-1. Notion d'association

Une association est une relation entre deux classes (association binaire) ou plus (association n-aire), qui décrit les connexions structurelles entre leurs instances. Une association indique donc qu'il peut y avoir des liens entre des instances des classes associées.

Comment une association doit-elle être modélisée ? Plus précisément, quelle différence existe-t-il entre les deux diagrammes de la figure 3.3 ?

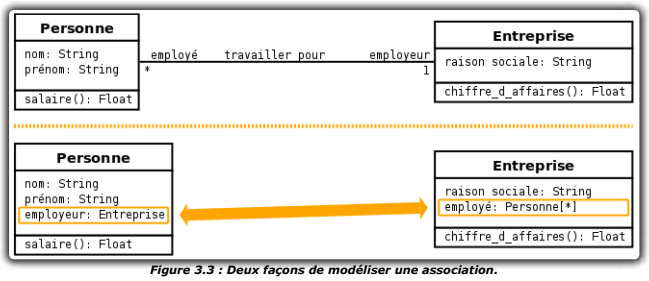


Figure 3.3 : Deux façons de modéliser une association.

Dans la première version, l'association apparaît clairement et constitue une entité distincte. Dans la seconde, l'association se manifeste par la présence de deux attributs dans chacune des classes en relation. C'est en fait la manière dont une association est généralement implémentée dans un langage objet quelconque (cf. section **3.6.2**), mais pas dans tout langage de

représentation (cf. section 3.6.3).

La question de savoir s'il faut modéliser les associations en tant que telles a longtemps fait débat. UML a tranché pour la première version, car elle se situe plus à un niveau conceptuel (par opposition au niveau d'implémentation) et simplifie grandement la modélisation d'associations complexes (comme les associations plusieurs à plusieurs par exemple).

Un attribut peut alors être considéré comme une association dégénérée dans laquelle une terminaison d'association(10) est détenue par un classeur (généralement une classe). Le classeur détenant cette terminaison d'association devrait théoriquement se trouver à l'autre extrémité, non modélisée, de l'association. Un attribut n'est donc rien d'autre qu'une terminaison d'un cas particulier d'association (cf. figure 3.9 section 3.3.5).

Ainsi, les terminaisons d'associations et les attributs sont deux éléments conceptuellement très proches que l'on regroupe sous le terme de *propriété*.

3-3-2. Terminaison d'association

3-3-2-a. Propriétaire d'une terminaison d'association

La possession d'une terminaison d'association par le classeur situé à l'autre extrémité de l'association peut être spécifiée graphiquement par l'adjonction d'un petit cercle plein (point) entre la terminaison d'association et la classe (cf. figure 3.4). Il n'est pas indispensable de préciser la possession des terminaisons d'associations. Dans ce cas, la possession est ambiguë. Par contre, si l'on indique des possessions de terminaisons d'associations, toutes les terminaisons d'associations sont non ambiguës : la présence d'un point implique que la terminaison d'association appartient à la classe située à l'autre extrémité, l'absence du point implique que la terminaison d'association appartient à l'association.

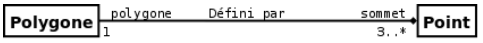


Figure 3.4 : Utilisation d'un petit cercle plein pour préciser le propriétaire d'une terminaison d'association.

Par exemple, le diagramme de la figure 3.4 précise que la terminaison d'association *sommet* (*i.e.* la propriété *sommet*) appartient à la classe *Polygone* tandis que la terminaison d'association *polygone* (*i.e.* la propriété *polygone*) appartient à l'association *Défini par*.

3-3-2-b. Une terminaison d'association est une propriété

Une propriété est une caractéristique structurelle. Dans le cas d'une classe, les propriétés sont constituées par les attributs et les éventuelles terminaisons d'association que possède la classe. Dans le cas d'une association, les propriétés sont constituées par les terminaisons d'association que possède l'association.

> Attention, une association ne possède pas forcément toutes ses terminaisons d'association !

Une propriété peut être paramétrée par les éléments suivants (on s'intéresse ici essentiellement aux terminaisons d'associations puisque les attributs ont été largement traités section 3.2.1) :

nom :

comme un attribut, une terminaison d'association peut être nommée. Le nom est situé à proximité de la terminaison, mais contrairement à un attribut, ce nom est facultatif. Le nom d'une terminaison d'association est appelé *nom du rôle*. Une association peut donc posséder autant de noms de rôle que de terminaisons (deux pour une association binaire et *n* pour une association n-aire) ;

visibilité :

comme un attribut, une terminaison d'association possède une visibilité (cf. section 3.2.4). La visibilité est mentionnée à proximité de la terminaison, et plus précisément, le cas échéant, devant le nom de la terminaison ;

multiplicité :

comme un attribut, une terminaison d'association peut posséder une multiplicité. Elle est mentionnée à proximité de la terminaison. Il n'est pas impératif de la préciser, mais, contrairement à un attribut dont la multiplicité par défaut est 1, la multiplicité par défaut d'une terminaison d'association est *non spécifiée*. L'interprétation de la multiplicité pour une terminaison d'association est moins évidente que pour un attribut (cf. section 3.3.4) ;

navigabilité :

pour un attribut, la navigabilité est implicite, navigable, et toujours depuis la classe vers l'attribut. Pour une terminaison d'association, la navigabilité peut être précisée (cf. section 3.3.5).

3-3-3. Association binaire et n-aire

3-3-3-a. Association binaire

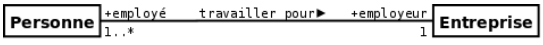


Figure 3.5 : Exemple d'association binaire.

Une association binaire est matérialisée par un trait plein entre les classes associées (cf. figure 3.5). Elle peut être ornée d'un nom, avec éventuellement une précision du sens de lecture (► ou ◄).

Quand les deux extrémités de l'association pointent vers la même classe, l'association est dite *réflexive*.

3-3-3-b. Association n-aire

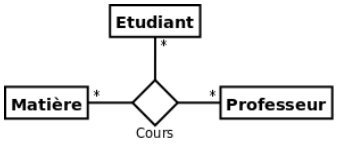


Figure 3.6 : Exemple d'association n-aire.

Une association n-aire lie plus de deux classes. La section 3.3.4 détaille comment interpréter les multiplicités d'une association n-aire. La ligne pointillée d'une classe-association (cf. section 3.3.7) peut être reliée au losange par une ligne discontinue pour représenter une association n-aire dotée d'attributs, d'opérations ou d'associations.

On représente une association n-aire par un grand losange avec un chemin partant vers chaque classe participante (cf. figure 3.6). Le nom de l'association, le cas échéant, apparaît à proximité du losange.

3-3-4. Multiplicité ou cardinalité

La multiplicité associée à une terminaison d'association, d'agrégation ou de composition 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 ;
- plusieurs : * ou 0..* ;
- au moins un : 1..* ;
- de un à six : 1..6.

Dans une association binaire (cf. figure 3.5), la multiplicité sur la terminaison cible contraint le nombre d'objets de la classe cible pouvant être associés à un seul objet donné de la classe source (la classe de l'autre terminaison de l'association).

Dans une association n-aire, la multiplicité apparaissant sur le lien de chaque classe s'applique sur une instance de chacune des classes, à l'exclusion de la classe-association et de la classe considérée. Par exemple, si on prend une association ternaire entre les classes (A, B, C), la multiplicité de la terminaison C indique le nombre d'objets C qui peuvent apparaître dans l'association (définie section **3.3.7**) avec une paire particulière d'objets A et B.

> Pour une association n-aire, la multiplicité minimale doit en principe, mais pas nécessairement, 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 !

> Il faut noter que, pour les habitués du modèle entité/relation, les multiplicités sont en UML « à l'envers » (par référence à Merise) pour les associations binaires et « à l'endroit » pour les n-aires avec *n*>2.

3-3-5. Navigabilité

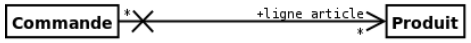


Figure 3.7 : Navigabilité.

La navigabilité indique s'il est possible de traverser une association. On représente graphiquement la navigabilité 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 3.7). Par défaut, une association est navigable dans les deux sens.

Par exemple, sur la figure 3.7, 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.

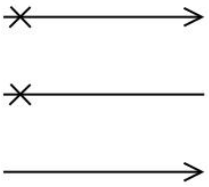


Figure 3.8 : Implicitement, ces trois notations ont la même sémantique.

Lorsque l'on représente la navigabilité uniquement sur l'une des extrémités d'une association, il faut remarquer que, implicitement, les trois associations représentées sur la figure 3.8 ont la même signification : l'association ne peut être traversée que dans un sens.

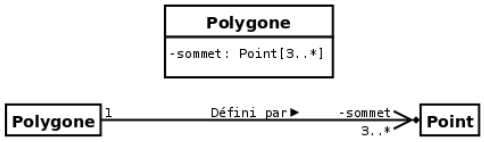


Figure 3.9 : Deux modélisations équivalentes.

Dans la section **3.3.1**, nous avons dit que :

« Un attribut est une association dégénérée dans laquelle une terminaison d'association est détenue par un classeur (généralement une classe). Le classeur détenant cette terminaison d'association devrait théoriquement se trouver à l'autre terminaison, non modélisée, de l'association. Un attribut n'est donc rien d'autre qu'une terminaison d'un cas particulier d'association. »

La figure 3.9 illustre parfaitement cette situation.

> Attention toutefois, si vous avez une classe *Point* dans votre diagramme de classe, il est extrêmement maladroit de représenter des classes (comme la classe *Polygone*) avec un ou plusieurs attributs de type *Point*. Il faut, dans ce cas, matérialiser cette propriété de la classe en question par une ou plusieurs associations avec la classe *Point*.

3-3-6. Qualification

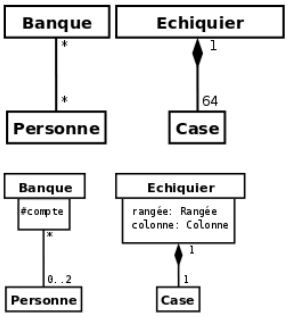


Figure 3.10 : En haut, un diagramme représentant l'association entre une banque et ses clients (à gauche), et un diagramme représentant l'association entre un échiquier et les cases qui le composent (à droite). En bas, les diagrammes équivalents utilisant des associations qualifiées.

Généralement, une classe peut être décomposée en sous-classes ou posséder plusieurs propriétés. Une telle classe rassemble un ensemble d'éléments (d'objets). 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*. Un qualificatif permet donc de sélectionner un ou des objets dans le jeu des objets d'un objet (appelé *objet qualifié*) relié par une association à un autre objet. L'objet sélectionné par la valeur du qualificatif est appelé *objet cible*. L'association est appelée *association qualifiée*. Un qualificatif agit toujours sur une association dont la multiplicité est *plusieurs* (avant que l'association ne soit qualifiée) du côté *cible*.

Un objet qualifié et une valeur de qualificatif génèrent un objet cible lié unique. En considérant un objet qualifié, chaque valeur de qualificatif désigne un objet cible unique.

Par exemple, le diagramme de gauche de la figure 3.10 nous dit que :

- un compte dans une banque appartient à au plus deux personnes. 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, une instance du couple {*Personne , compte*} est en association avec une instance unique de la classe *Banque*.

Le diagramme de droite de cette même figure nous dit que :

- une instance du triplet {Échiquier*, rangée, colonne*} est en association avec une instance unique de la classe *Case* ;
- inversement, une instance de la classe *Case* est en association avec une instance unique du triplet {Échiquier, *rangée, colonne*}.

3-3-7. Classe-association

3-3-7-a. Définition et représentation

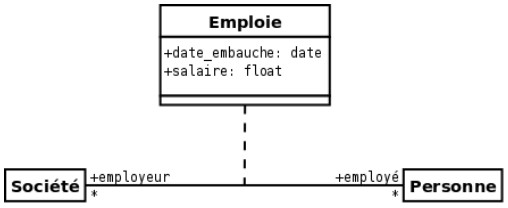


Figure 3.11 : Exemple de classe-association.

Parfois, une association doit posséder des propriétés. Par exemple, l'association *Emploie* entre une société et une personne possède comme propriétés le salaire et la date d'embauche. En effet, ces deux propriétés n'appartiennent ni à la société, qui peut employer plusieurs personnes, ni aux personnes, qui peuvent avoir plusieurs emplois. Il s'agit donc bien de propriétés de l'association *Emploie*. Les associations ne pouvant posséder de propriété, il faut introduire un nouveau concept pour modéliser cette situation : celui de *classe-association*.

Une classe-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.

Une classe-association est caractérisée par un trait discontinu entre la classe et l'association qu'elle représente (figure 3.11).

3-3-7-b. Classe-association pour plusieurs associations

Il n'est pas possible de rattacher une classe-association à plus d'une association puisque la classe-association constitue elle-même l'association. Dans le cas où plusieurs classe-associations doivent disposer des mêmes caractéristiques, elles doivent hériter d'une même classe possédant ces caractéristiques, ou l'utiliser en tant qu'attribut.

De même, il n'est pas possible de rattacher une instance de la classe d'une classe-association à plusieurs instances de l'association de la classe-association. En effet, la représentation graphique (association reliée par une ligne pointillée à une classe déportée) est trompeuse : une classe-association est une entité sémantique atomique et non composite qui s'instancie donc en bloc. Ce problème et à nouveau abordé et illustré section **3.5.2**.

3-3-7-c. Autoassociation sur classe-association

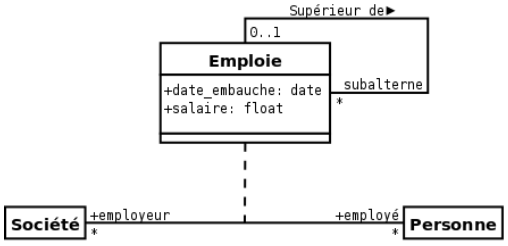


Figure 3.12 : Exemple d'autoassociation sur classe-association.

Imaginons que nous voulions ajouter une association *Supérieur de* dans le diagramme 3.11 pour préciser qu'une personne est le supérieur d'une autre personne. On ne peut simplement ajouter 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 donnée, 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-association *Emploie* (cf. figure 3.12).

3-3-7-d. Liens multiples

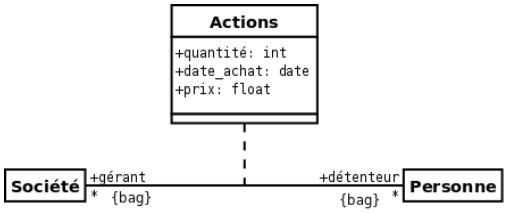


Figure 3.13 : Exemple de classe-association avec liens multiples.

Plusieurs instances d'une même association ne peuvent lier les mêmes objets. Cependant, si l'on s'intéresse à l'exemple de la figure 3.13, on voit bien qu'il doit pouvoir y avoir plusieurs instances de la classe-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.

C'est justement la raison de la contrainte *{bag}* qui, placée sur les terminaisons d'association de la classe-association *Actions*, indique qu'il peut y avoir des liens multiples impliquant les mêmes paires d'objets.

3-3-7-e. Équivalences

Parfois, l'information véhiculée par une classe-association peut être véhiculée sans perte d'une autre manière (cf. figure 3.14 et 3.15).

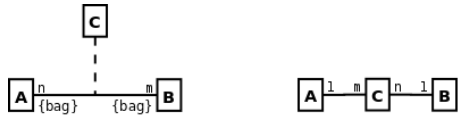


Figure 3.14 : Deux modélisations modélisant la même information.

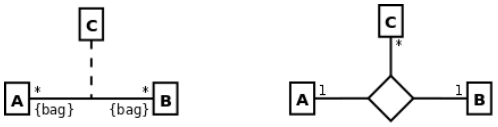


Figure 3.15 : Deux modélisations modélisant la même information.

3-3-7-f. Classe-association, association n-aire ou association qualifiée ?

Il n'est souvent pas simple trancher entre l'utilisation d'une classe-association, d'une association n-aire ou encore d'une association qualifiée. Lorsque l'on utilise l'un de ces trois types d'association, il peut être utile ou instructif de se demander si l'un des deux autres types ne serait pas plus pertinent. Dans tous les cas, il faut garder à l'esprit qu'une classe-association est d'abord et avant tout une association et que, dans une classe-association, la classe est indissociable de l'association.

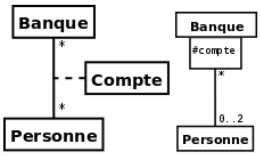


Figure 3.16 : Pour couvrir le cas des comptes joints, il faut utiliser le modèle de droite.

Ainsi, le cas d'un compte joint ne peut être représenté correctement par le diagramme de gauche dans figure 3.16 : mieux vaut utiliser une association qualifiée (diagramme de droite dans la figure). Ce problème et à nouveau abordé et illustré section **3.5.2**.

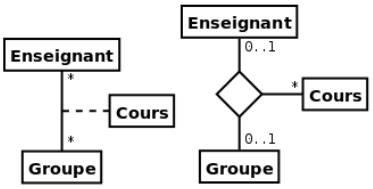


Figure 3.17 : Si un cours doit pouvoir exister indépendamment d'un lien entre un enseignant et un groupe, il faut utiliser le modèle de droite.

Dans le diagramme de gauche de la figure 3.17, un cours ne peut exister que s'il existe un lien entre un objet *Enseignant* et un objet *Groupe*. Quand le lien est rompu (effacé), le cours l'est également. Si un cours doit pouvoir exister indépendamment de l'existence d'un lien (on n'a pas encore trouvé d'enseignant pour ce cours, le cours n'est pas enseigné cette année, mais le sera probablement l'année prochaine…) il faut opter pour une association ternaire (modèle de droite dans figure 3.17).

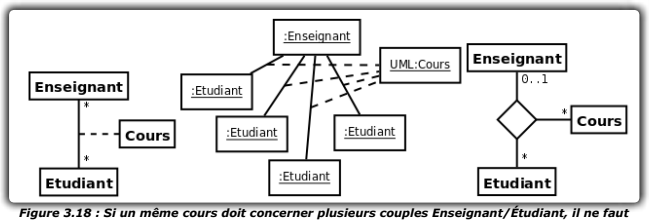


Figure 3.18 : Si un même cours doit concerner plusieurs couples Enseignant/Étudiant, il ne faut pas utiliser une classe-association, mais une association ternaire comme sur le modèle de droite.

Le cas de figure est encore pire si l'on remplace *Groupe* par Étudiant (cf. modèle à gauche sur la figure 3.18). En effet, le cas général décrit par ce modèle ne correspond pas du tout au diagramme d'objet (cf. section **3.5**) situé au centre. Nous reviendrons sur ce problème dans la section **3.5.2** avec l'illustration 3.24. Dans cette situation, il faut opter pour une association ternaire comme sur le modèle de droite.

3-3-8. Agrégation et composition

3-3-8-a. Agrégation

Figure 3.19 : Exemple de relation d'agrégation et de composition.

Une association simple entre deux classes représente une relation structurelle entre pairs, c'est-à-dire entre deux classes de même niveau conceptuel : aucune des deux n'est plus importante que l'autre. Lorsque l'on souhaite modéliser une relation *tout/partie* où une classe constitue un élément plus grand (*tout*) composé d'éléments plus petits (*partie*), il faut utiliser une agrégation.

Une agrégation est une association qui représente une relation d'inclusion structurelle ou comportementale d'un élément dans un ensemble. Graphiquement, on ajoute un losange vide (♦) du côté de l'agrégat (cf. figure 3.19). Contrairement à une association simple, l'agrégation est transitive.

La signification de cette forme simple d'agrégation est uniquement conceptuelle. Elle ne contraint pas la navigabilité ou les multiplicités de l'association. Elle n'entraîne pas non plus de contrainte sur la durée de vie des parties par rapport au tout.

3-3-8-b. Composition

La composition, également appelée agrégation composite, décrit une contenance structurelle entre 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 de l'agrégat (cf. figure 3.19).

> Les notions d'agrégation et surtout de composition posent de nombreux problèmes en modélisation et sont souvent le sujet de querelles d'experts et sources de confusions. Ce que dit la norme *UML Superstructure version 2.1.1*reflète d'ailleurs très bien le flou qui entoure ces notions :
> *Precise semantics of shared aggregation varies by application area and modeler. The order and way in which part instances are created is not defined.*

3-3-9. Généralisation et Héritage

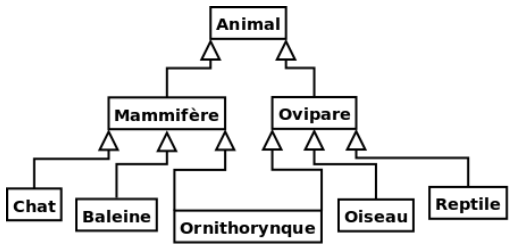


Figure 3.20 : Partie du règne animal décrit avec l'héritage multiple.

La généralisation décrit une relation entre une classe générale (classe de base ou classe parent) et une classe spécialisée (sous-classe). 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.

Dans le langage UML, ainsi que dans la plupart des langages objet, cette relation de généralisation se traduit par le concept d'héritage. On parle également de relation d'héritage. Ainsi, l'héritage permet la classification des objets (cf. figure 3.20).

Le symbole utilisé pour la relation d'héritage ou de généralisation est une flèche avec un trait plein dont la pointe est un triangle fermé désignant le cas le plus général (cf. figure 3.20).

Les propriétés principales de l'héritage sont :

- la classe enfant possède toutes les caractéristiques de ses classes parents, mais elle ne peut accéder aux caractéristiques privées de cette dernière ;
- une classe enfant peut redéfinir (même signature) une ou plusieurs méthodes de la classe parent. Sauf indication contraire, un objet utilise les opérations les plus spécialisées dans la hiérarchie des classes ;
- toutes les associations de la classe parent s'appliquent aux classes dérivées ;
- une instance d'une classe peut être utilisée partout où une instance de sa classe parent est attendue. Par exemple, en se basant sur le diagramme de la figure 3.20, toute opération acceptant un objet d'une classe *Animal* doit accepter un objet de la classe *Chat* ;
- une classe peut avoir plusieurs parents, on parle alors d'héritage multiple (cf. la classe *Ornithorynque* de la figure 3.20). Le langage C++ est un des langages objet permettant son implémentation effective, le langage Java ne le permet pas.

En UML, la relation d'héritage n'est pas propre aux classes. Elle s'applique à d'autres éléments du langage comme les paquetages, les acteurs (cf. section **2.3.3**) ou les cas d'utilisation (cf. section **2.3.2**).

3-3-10. Dépendance

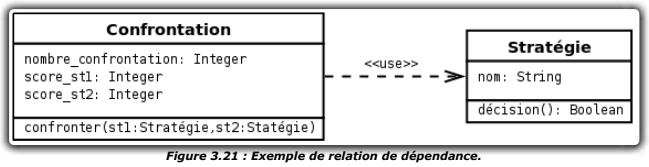


Figure 3.21 : Exemple de relation de dépendance.

Une dépendance est une relation unidirectionnelle exprimant une dépendance sémantique entre des éléments du modèle. Elle est représentée par un trait discontinu orienté (cf. figure 3.21). Elle indique que la modification de la cible peut impliquer une modification de la source. La dépendance est souvent stéréotypée pour mieux expliciter le lien sémantique entre les éléments du modèle (cf. figure 3.21 ou 3.25).

On utilise souvent une dépendance quand une classe en utilise une autre comme argument dans la signature d'une opération. Par exemple, le diagramme de la figure 3.21 montre que la classe *Confrontation* utilise la classe *Stratégie,* car la classe *Confrontation* possède une méthode *confronter* dont deux paramètres sont du type *Stratégie*. Si la classe *Stratégie*, notamment son interface, change, alors des modifications devront également être apportées à la classe *Confrontation*.

3-4. 3.4 Interfaces

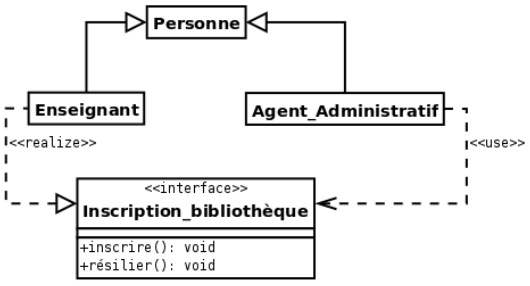


Figure 3.22 : Exemple de diagramme mettant en œuvre une interface.

Nous avons déjà abordé la notion d'interface dans les sections **1.3.4** et **3.2.4**. En effet, les classes permettent de définir en même temps un objet et son interface. Le classeur, que nous décrivons dans cette section, ne permet de définir que des éléments d'interface. Il peut s'agir de l'interface complète d'un objet, ou simplement d'une partie d'interface qui sera commune à plusieurs objets.

Le rôle de ce classeur, stéréotypé << *interface* >>, est de regrouper un ensemble de propriétés et d'opérations assurant un service cohérent. L'objectif est de diminuer le couplage entre deux classeurs. La notion d'interface en UML est très proche de la notion d'interface en Java.

Une interface est représentée comme une classe excepté l'absence du mot-clef *abstract* (car l'interface et toutes ses méthodes sont, par définition, abstraites) et l'ajout du stéréotype << *interface* >> (cf. figure 3.22).

Une interface doit être réalisée par au moins une classe et peut l'être par plusieurs. Graphiquement, cela est représenté par un trait discontinu terminé par une flèche triangulaire et le stéréotype *« realize »*. Une classe peut très bien réaliser plusieurs interfaces. Une classe (classe cliente de l'interface) peut dépendre d'une interface (interface requise). On représente cela par une relation de dépendance et le stéréotype *« use »*.

> Attention aux problèmes de conflits si une classe dépend d'une interface réalisée par plusieurs autres classes.

La notion d'interface est assez proche de la notion de classe abstraite avec une capacité de découplage plus grand. En C++ (le C++ ne connaît pas la notion d'interface), la notion d'interface est généralement implémentée par une classe abstraite.

3-5. 3.5 Diagramme d'objets (object diagram)

3-5-1. 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 ;
- prendre une image (*snapshot*) d'un système à un moment donné.

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

Par exemple, le diagramme de classes de la figure 3.23 montre qu'une entreprise emploie au moins deux personnes et qu'une personne travaille dans au plus deux entreprises. Le diagramme d'objets modélise lui une entreprise particulière (*PERTNE*) qui emploie trois personnes.

Un diagramme d'objets ne montre pas l'évolution du système dans le temps. Pour représenter une interaction, il faut utiliser un diagramme de communication (cf. section 7.2) ou de séquence (cf. section 7.3).

3-5-2. Représentation

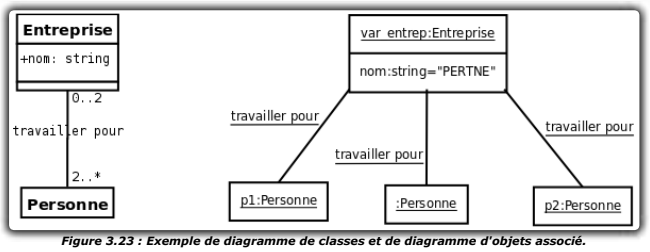


Figure 3.23 : Exemple de diagramme de classes et de diagramme d'objets associé.

Graphiquement, un objet se représente comme une classe. Cependant, le compartiment des opérations n'est pas utile. De plus, le nom de la classe dont l'objet est une instance est précédé d'un << : >> et est souligné. Pour différencier les objets d'une même classe, leur identifiant peut être ajouté devant le nom de la classe. Enfin les attributs reçoivent 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. Naturellement, on ne représente pas les multiplicités.

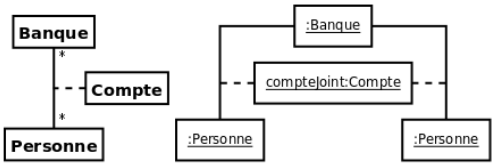


Figure 3.24 : Le diagramme d'objets de droite, illustrant le cas de figure d'un compte joint, n'est pas une instance normale du diagramme de classe de gauche, mais peut préciser une situation exceptionnelle.

La norme UML 2.1.1 précise qu'une instance de classe-association ne peut être associée qu'à une instance de chacune des classes associées[11] ce qui interdit d'instancier le diagramme de classe à gauche dans la figure 3.24 par le diagramme d'objet à droite dans cette même figure. Cependant, un diagramme d'objet peut être utilisé pour exprimer une exception. Sur la figure , le diagramme d'objets à droite peut être légitime s'il vient préciser une situation exceptionnelle non prise en compte par le diagramme de classe représenté à gauche. Néanmoins, le cas des comptes joints n'étant pas si exceptionnel, mieux vaut revoir la modélisation comme préconisé par la figure 3.16.

3-5-3. Relation de dépendance d'instanciation

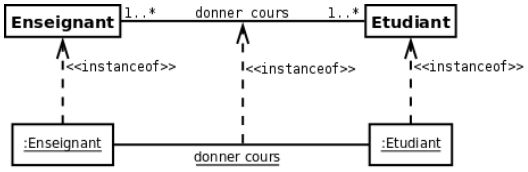


Figure 3.25 : Dépendance d'instanciation entre les classeurs et leurs instances.

La relation de dépendance d'instanciation (stéréotypée << *instanceof* >>) décrit la relation entre un classeur et ses instances. Elle relie, en particulier, les liens aux associations et les objets aux classes.

3-6. Élaboration et implémentation d'un diagramme de classes

3-6-1. Élaboration d'un diagramme de classes

Une démarche couramment utilisée pour bâtir un diagramme de classes consiste à :

Trouver les classes du domaine étudié.

Cette étape empirique se fait généralement en collaboration avec un expert du domaine. Les classes correspondent généralement à des concepts ou des substantifs du domaine ;

Trouver les associations entre classes.

Les associations correspondent souvent à des verbes, ou des constructions verbales, mettant en relation plusieurs classes, comme << est composé de >>, << pilote >>, << travaille pour >>.

Attention, méfiez-vous de certains attributs qui sont en réalité des relations entre classes.

Trouver les attributs des classes.

Les attributs correspondent souvent à des substantifs, ou des groupes nominaux, tels que << la masse d'une voiture >> ou << le montant d'une transaction >>. Les adjectifs et les valeurs correspondent souvent à des valeurs d'attributs. Vous pouvez ajouter des attributs à toutes les étapes du cycle de vie d'un projet (implémentation comprise). N'espérez pas trouver tous les attributs dès la construction du diagramme de classes ;

Organiser et simplifier le modèle.

En éliminant les classes redondantes et en utilisant l'héritage ;

Itérer et raffiner le modèle.

Un modèle est rarement correct dès sa première construction. La modélisation objet est un processus non pas linéaire, mais itératif.

3-6-2. Implémentation en Java

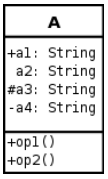
3-6-2-a. Classe

Parfois, la génération automatique de code produit, pour chaque classe, un constructeur et une méthode finalize comme ci-dessous. Rappelons que cette méthode est invoquée par le *ramasse-miettes* lorsque celui-ci constate que l'objet n'est plus référencé. Pour des raisons de simplification, nous ne ferons plus figurer ces opérations dans les sections suivantes.

Sélectionnez

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

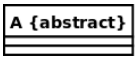
3-6-2-b. Classe avec attributs et opérations



Sélectionnez

```
public class A {
  public    String a1;
  package   String a2;
  protected String a3;
  private   String a4;
  public void op1() {
    ...
  }
  public void op2() {
    ...
  }
}
```

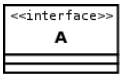
3-6-2-c. Classe abstraite



Sélectionnez

```
public abstract class A {
  ...
}
```

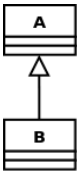
3-6-2-d. Interface



Sélectionnez

```
public interface A {
  ...
}
```

3-6-2-e. Héritage simple

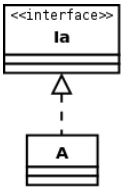


Sélectionnez

```
public class A {
  ...
}

public class B extends A {
  ...
}
```

3-6-2-f. Réalisation d'une interface par une classe

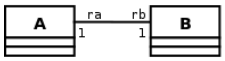


Sélectionnez

```
public interface Ia {
  ...
}

public class A implements Ia {
  ...
}
```

3-6-2-g. Association bidirectionnelle 1 vers 1

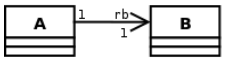


Sélectionnez

```
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 {
  private A ra;
  public void addA( A a ) {
    if( a != null ) {
      if (a.getB() != null) {   // si a est déjà connecté à un autre B
        a.getB().setA( null );  // cet autre B doit se déconnecter
      }
      this.setA( a );
      a.setB( this );
    }
  }
  public void setA(A a){ this.ra=a; }
  public A getA(){ return(ra); }
}
```

3-6-2-h. Association unidirectionnelle 1 vers 1

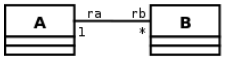


Sélectionnez

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

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

3-6-2-i. Association bidirectionnelle 1 vers N



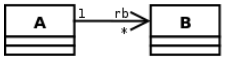
Sélectionnez

```
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);
      }
    }
  }
}
```

3-6-2-j. Association unidirectionnelle 1 vers plusieurs

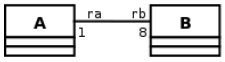


Sélectionnez

```
public class A {
  private ArrayList <B> rb;
  public A() { rb = new  ArrayList<B>(); }
  public void addB(B b){
    if( !rb.contains( b ) ) {
      rb.add(b);
    }
  }
}

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

3-6-2-k. Association 1 vers N

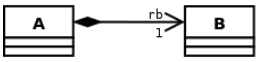


Dans ce cas, il faut utiliser un tableau plutôt qu'un vecteur. La dimension du tableau étant donnée par la cardinalité de la terminaison d'association.

3-6-2-l. Agrégations

Les agrégations s'implémentent comme les associations.

3-6-2-m. Composition



Une composition peut s'implémenter comme une association unidirectionnelle.

3-6-3. Implémentation en SQL

3-6-3-a. Introduction

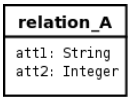
Il est possible de traduire un diagramme de classe en modèle relationnel. Bien entendu, les méthodes des classes ne sont pas traduites. Aujourd'hui, lors de la conception de base de données, il devient de plus en plus courant d'utiliser la modélisation UML plutôt que le traditionnel modèle entités-associations.

Cependant, à moins d'avoir respecté une méthodologie adaptée, la correspondance entre le modèle objet et le modèle relationnel n'est pas une tâche facile. En effet, elle ne peut que rarement être complète puisque 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 ? Toutefois, de nombreux AGL (Atelier de Génie Logiciel) comportent maintenant des fonctionnalités de traduction en SQL qui peuvent aider le développeur dans cette tâche.

Dans la section 9.3.1, nous présentons un type de diagramme de classes, appelé *modèle du domaine*, tout à fait adapté à une implémentation sous forme de base de données.

3-6-3-b. Classe avec attributs

Chaque classe devient une relation. Les attributs de la classe deviennent des attributs de la relation. Si la classe possède un identifiant, il devient la clé primaire de la relation, sinon, il faut ajouter une clé primaire arbitraire.

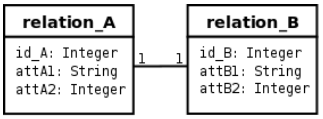


Sélectionnez

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

3-6-3-c. 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.



Sélectionnez

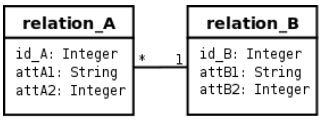
```
create table relation_A (
    id_A integer primary key,
    attA1 text,
```

```
    attA2 integer);

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

3-6-3-d. Association 1 vers plusieurs

Pour représenter une association 1 vers plusieurs, on procède comme pour une association 1 vers 1, excepté que c'est forcément la relation du côté plusieurs qui reçoit comme clé étrangère la clé primaire de la relation du côté 1.



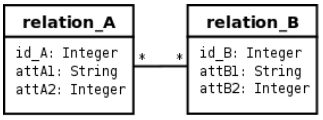
Sélectionnez

```
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);
```

3-6-3-e. 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.



Sélectionnez

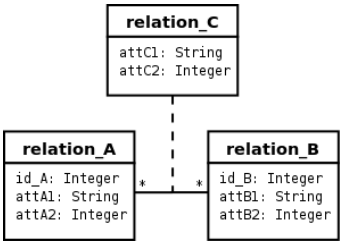
```
create table relation_A (
    id_A integer primary key,
    attA1 text,
    attA2 integer);

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));
```

3-6-3-f. Classe-association plusieurs vers plusieurs

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



Sélectionnez

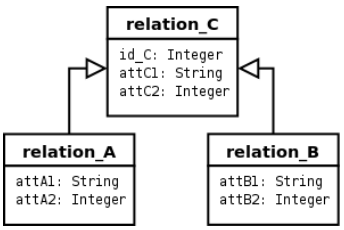
```
create table relation_A (
    id_A integer primary key,
    attA1 text,
    attA2 integer);

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

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

3-6-3-g. Héritage

Les relations correspondant aux sous-classes ont comme clés 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.



Sélectionnez

```
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 option 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.

Sélectionnez

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

Précédent Sommaire Suivant

(7) Une partition d'un ensemble est un ensemble de parties non vides de cet ensemble, deux à deux disjointes et dont la réunion est égale à l'ensemble.

(8) Il faut ici aborder un petit problème de terminologie autour du mot *propriété*. En effet, dans la littérature, le mot *propriété* est parfois utilisé pour désigner toutes les caractéristiques d'une classe (*i.e.* les attributs comme les méthodes). Dans ce cas, les attributs et les terminaisons d'association sont rassemblés sous le terme de *propriétés structurelles*, le qualificatif *structurelle* prenant ici toute son importance. D'un autre côté, le mot *propriété* est souvent utilisé dans l'acception du terme anglais *property*(dans la norme UML Superstructure version 2.1.1), qui, lui, ne désigne que les attributs et les terminaisons d'association, c'est-à-dire les *propriétés structurelles*. Pour englober les méthodes, il faut alors utiliser le terme plus générique de *caractéristiques*, qui prend ainsi le rôle de traduction du terme anglais *feature* dans la norme. Dans le présent cours, je m'efforce de me conformer à cette deuxième solution où *propriété* et *propriété structurelle* désignent finalement la même chose.

(9) De manière générale, toute boîte non stéréotypée dans un diagramme de classes est implicitement une classe. Ainsi, le stéréotype ***class*** est le stéréotype par défaut.

(10) Une terminaison d'associations est une extrémité de l'association. Une association binaire en possède *deux*, une association n-aire en possède *n*.

(11) UML Superstructure Specification, v2.1.1 ; p.49 : << *It should be noted that in an instance of an association class, there is only one instance of the associated classifiers at each end, i.e., from the instance point of view, the multiplicity of the associations ends are '1'* >>