UML
Dr Khalifa SYLLA
Séquence 2: LE CONCEPT ORIENTE OBJET

LE CONCEPT ORIENTE OBJET

1.1. Notion d'objet

Rappel *: L'approche objet a été inventée pour faciliter l'évolution des applications. Elle considère une application comme une* ***société d'objets*** *qui coopèrent. L'achèvement d'une tâche par une application repose sur la communication entre tous ou certains de ses objets.*

Un objet est caractérisé par :

- Un état
- Un comportement
- Un identificateur

Exemple : Objet véhicule

Etat correspond à =

Marque : Renault

Modèle : Megane

NombresPortes : 4

Immatriculation : DK1234AX

Comportement = rouler, freiner

1.1.1. Etat d'un objet

L'état regroupe les valeurs instantanées de tous les **attributs** d'un objet. Il évolue au cours du temps : les attributs peuvent prendre de nouvelles valeurs en fonction du comportement du système.

Ainsi, l'état d'un objet à un instant donné est la conséquence de ses comportements passés. Il conditionne son comportement futur.

1.1.2. Comportement d'un objet

Il décrit les actions ou les réactions d'un objet. Il correspond à la mise en oeuvre d'une des compétences de l'objet. Ces compétences correspondent aux : fonctions, méthodes ou opérations. Le comportement est déclenché par un **message** (stimuli) interne ou externe à l'objet.

L'état et le comportement sont liés :

- Le comportement dépend de l'état
- L'état est modifié par le comportement

Exemple : Rouler défini un comportement de l'objet Renault Megane

1.1.3. L'identité

Tout objet possède une identité qui lui est propre et qui le caractérise. L'identité permet de distinguer tout objet de façon non ambiguë, indépendamment de l'état.

1.1.4. Communication entre objets

Rappel : L'approche objet considère une application comme une société d'objets qui coopèrent.

Les objets travaillent en synergie afin de réaliser les fonctions de l'application. Le comportement global d'une application repose sur la communication entre les objets qui la composent.

1.2. Notion de classe

Définition : une classe est une description abstraite d'un ensemble d'objets.

En d'autres termes, c'est une factorisation d'attributs et de comportements d'un ensemble d'objets.

Un objet n'est rien d'autres qu'une instance de classe.

Une classe se représente par un rectangle constitué de 4 compartiments :

Nomdelaclasse	{commence toujours par une majuscule}
attribut1 attribut2 ... attributN	{les attributs}
Operation1 Operation2 ... OperationN	{les methodes}
compartimentOptionnelNommé	{compartiment peu utilisé en pratique}

Exemples :

Figure : Classe non documentée

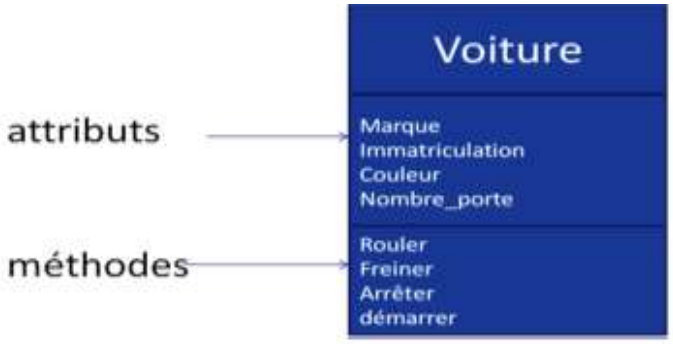

Figure : Classe documentée

Figure : Classe détaillée : attributs typés, méthodes spécifiées, niveau de protection des membres renseignés

Figure : Classe et Objet

1.3. Encapsulation et interface

L'encapsulation consiste à masquer les détails d'implémentation d'un objet, en définissant une interface.

L'interface est la vue externe d'un objet, elle définit les services accessibles (offerts) aux utilisateurs de l'objet.

L'encapsulation facilite l'évolution d'une application car elle stabilise l'utilisation des objets : on peut modifier l'implémentation des attributs d'un objet sans modifier son interface. Elle garantit l'intégrité des données, car elle permet d'interdire l'accès direct aux attributs des objets (utilisation d'accesseurs et de mutateurs).

1.4. Relations entre classes

1.4.1. Association

Une association exprime une connexion **sémantique** entre deux classes. Plus précisément, c'est une relation entre deux classes (association **binaire**) ou plus (association **n-aire),** qui décrit les connexions entre leurs instances. Une association est donc distanciable, sous forme de liens entre objets issus de classes associées.

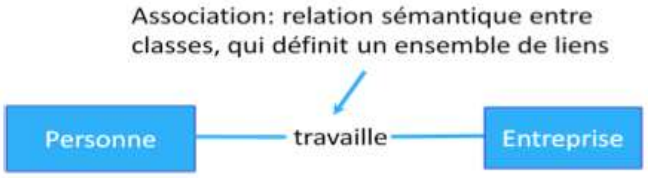

Figure : Exemple d'association

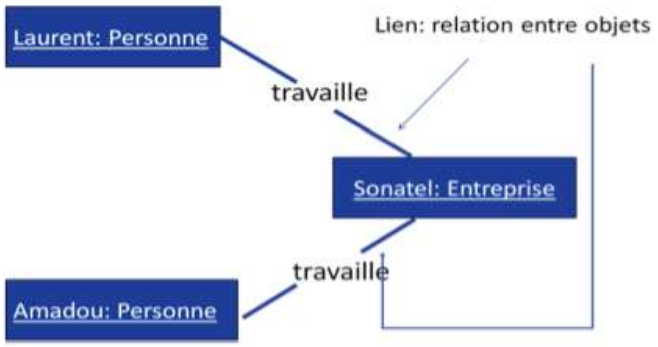

Figure: Exemple de lien entre objets

Figure : Exemple de lien entre objets

Une association peut être :

- **Active** : précise le sens de lecture principal d'une association.

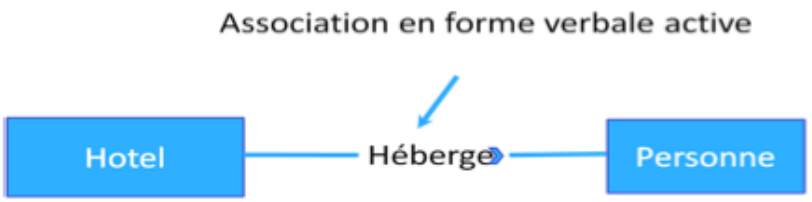

Figure : Association en forme verbale active

- ou **passive**

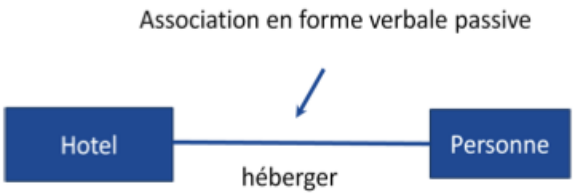

Figure : Association en forme verbale passive

Une association peut posséder un ou plusieurs **rôles**.

Définition : c'est le nom donné à une terminaison d'association. Le rôle est facultatif. Une association binaire peut posséder deux rôles, et une association n-aire N rôles.

Exemples :

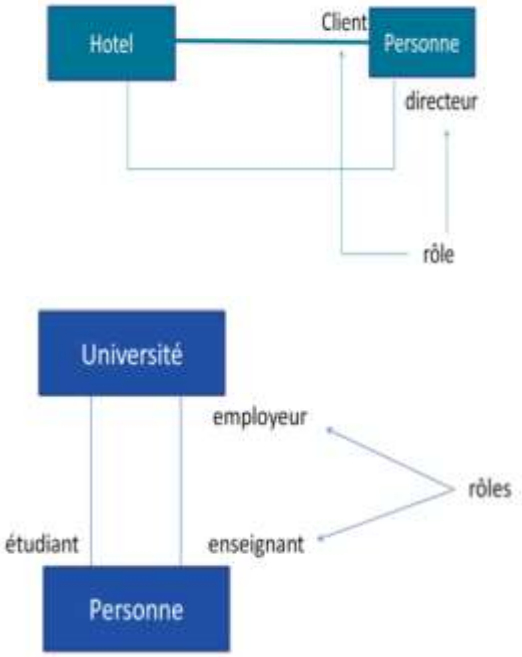

Figure : Exemple de rôle

a. Association n-aire

Il s'agit d'une association qui relie plus de deux classes... NB: De telles associations sont difficiles à déchiffrer et peuvent induire en erreur. Il vaut mieux limiter leur utilisation, en définissant de nouvelles catégories d'associations.

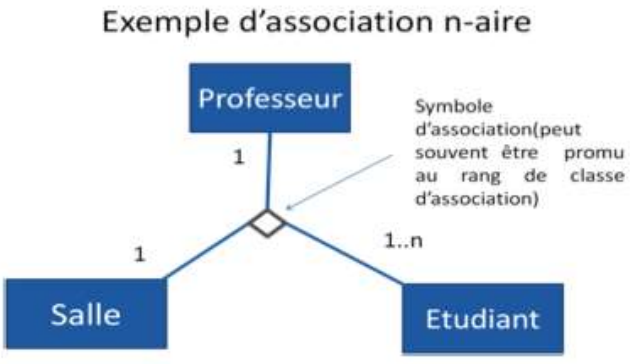

Figure : Exemple d'association n-aire

b. Association à navigabilité restreinte

Par défaut, une association est navigable dans les deux sens. La réduction de la portée de l'association est souvent réalisée en phase d'implémentation, mais peut aussi être exprimée dans un modèle pour indiquer que les instances d'une classe ne "connaissent" pas les instances d'une autre classe.

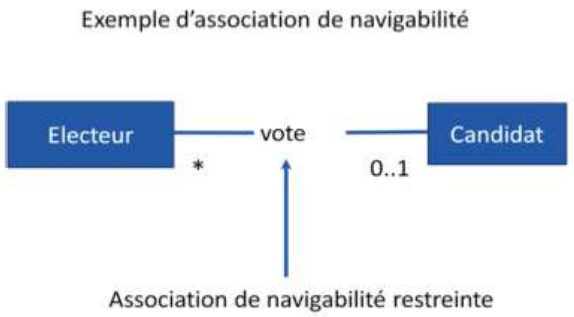

Figure : Exemple d'association de navigabilité

c. Classe d'association

Il s'agit d'une classe qui réalise la navigation (la relation) entre les instances d'autres classes.

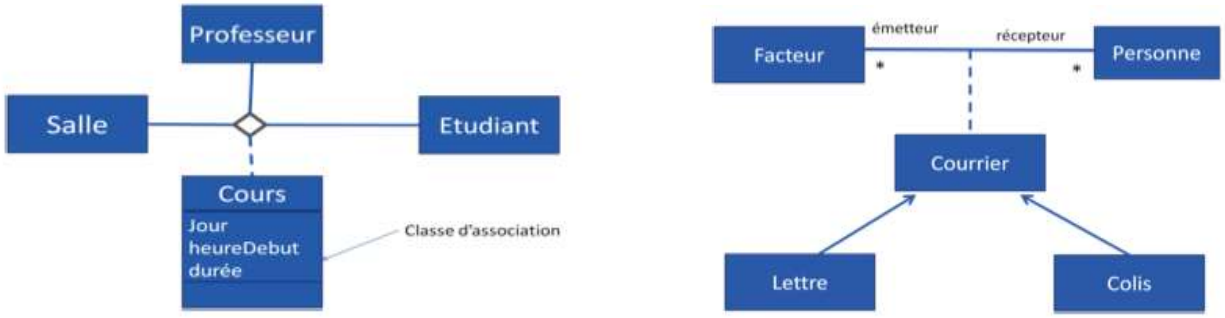

Figure : Exemples de classe d'association

d. Qualification

Elle permet de sélectionner un sous-ensemble d'objets, parmi l'ensemble des objets qui participent à une association. La restriction de l'association est définie par une clé, qui permet de sélectionner les objets ciblés.

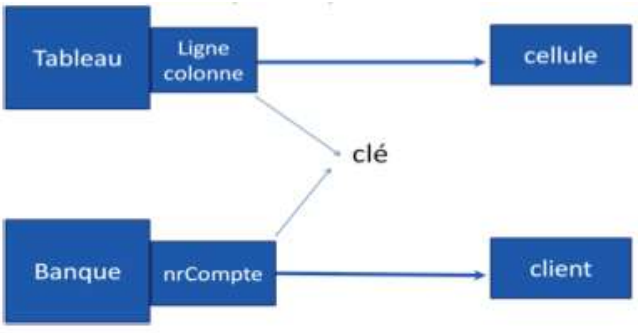

Figure : Qualification

e. Contrainte sur une association

Les contraintes sont des expressions qui précisent le rôle ou la portée d'un élément de modélisation (elles permettent d'étendre ou préciser sa sémantique).

Sur une association, elles peuvent par exemple restreindre le nombre d'instances visées (ce sont alors des "expressions de navigation").

Les contraintes peuvent s'exprimer en langage naturel. Graphiquement, il s'agit d'un texte encadré d'accolades.

NB: La qualification est un cas particulier de contrainte.

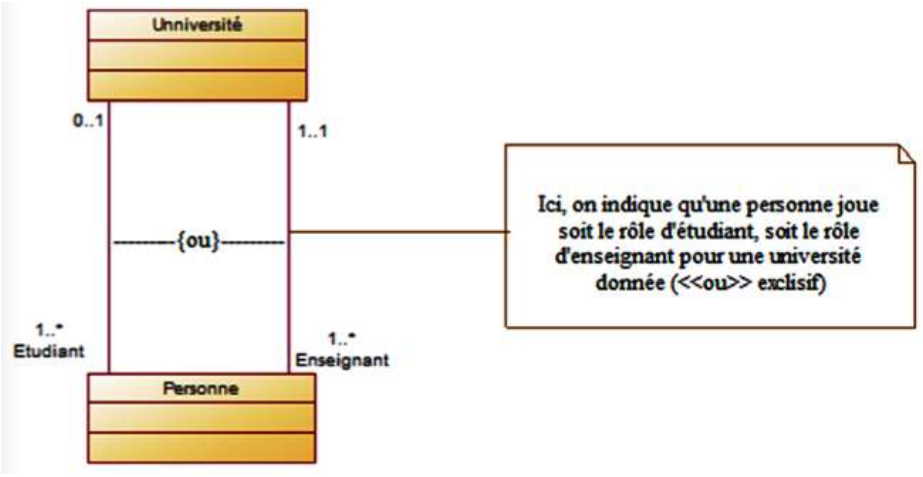

Figure: Contrainte sur une association

1.4.2. Dépendance

Définition : c'est une relation unidirectionnelle exprimant une dépendance sémantique entre des éléments du modèle.

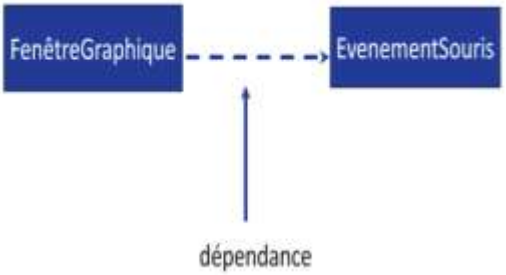

Figure : Exemple de dépendance

1.4.3. Agrégation

C'est une relation entre deux classes, spécifiant que les objets d'une classe sont des composants de l'autre classe. Une relation d'agrégation permet donc de définir des objets composés d'autres objets.

L'agrégation permet d'assembler des objets de base, afin de construire des objets plus complexes.

L'agrégation est une association non symétrique, qui exprime un couplage fort et une relation de subordination. Elle représente une relation de type "ensemble/ élément".

Une agrégation peut notamment (mais pas nécessairement) exprimer :

- qu'une classe (un " élément") fait partie d'une autre ("l'agrégat"),
- qu'un changement d'état d'une classe, entraîne un changement d'état d'une autre,
- qu'une action sur une classe, entraîne une action sur une autre.

A un même moment, une instance d'élément agrégé peut être liée à plusieurs instances d'autres classes (l'élément agrégé peut être partagé).

Une instance d'élément agrégé peut exister sans agrégat (et inversement): les cycles de vies de l'agrégat et de ses éléments agrégés peuvent être indépendants.

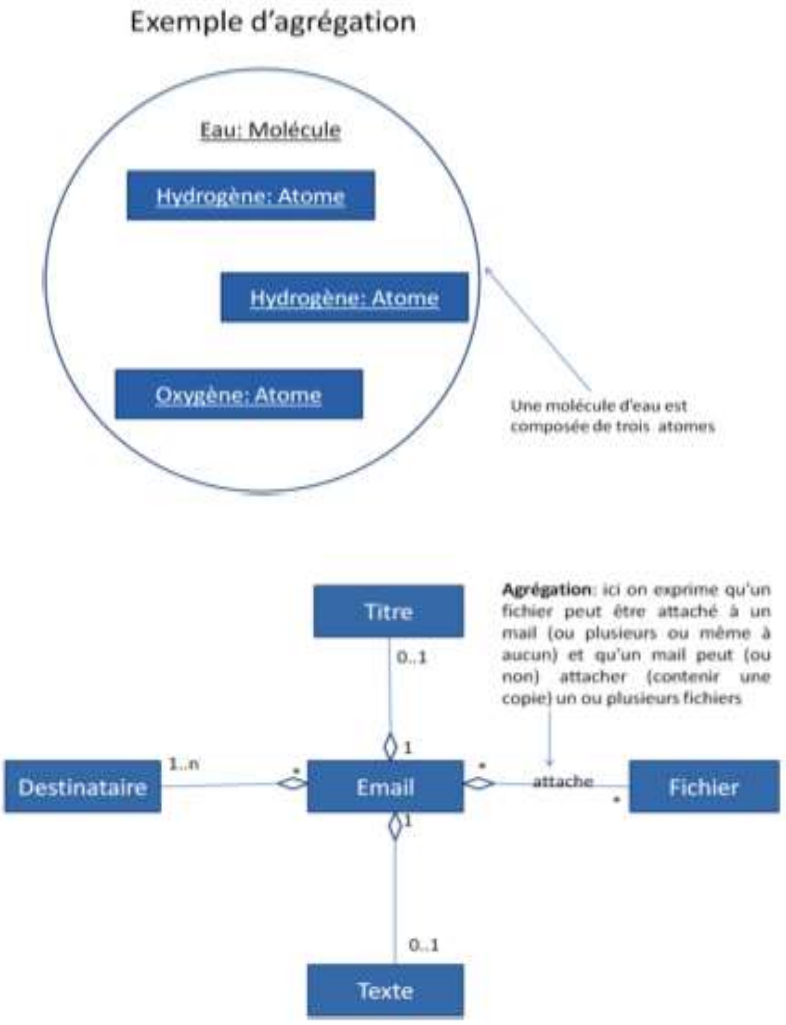

Figure : Exemples d'agrégation

1.4.4. Composition

La composition est une agrégation forte (agrégation par valeur). Les cycles de vies des éléments (les "composants") et de l'agrégat sont liés : si l'agrégat est détruit (ou copié), ses composants le sont aussi. A un même moment, une instance de composante peut être liée qu'à un seul agrégat.

Les "objets composites" sont des instances de classes composées.

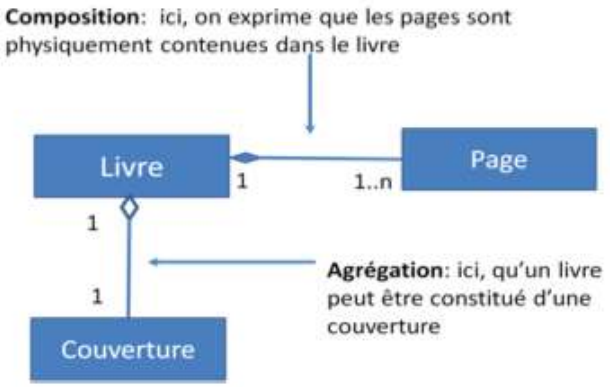

Figure : Exemples de composition

1.4.5. Multiplicité et cardinalité

Définition : précise le nombre d'instances qui participent à une relation.

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.

Les cardinalités d'une relation sont :

- n : exactement "n" (n, entier naturel > 0)
- n..m : de "n" à "m" (entiers naturels m > n)
- ∗ : plusieurs (équivalent `a "0..n" et "0..∗")
- n..∗ : "n" ou plus (n, entier naturel ou variable)

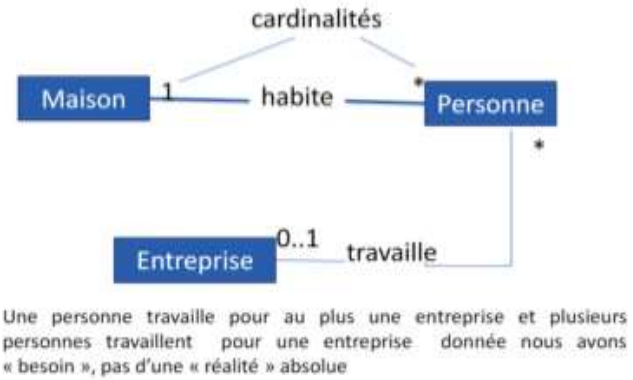

Figure : Exemple de cardinalité

1.4.6. Héritage

L'héritage est un mécanisme de transmission des propriétés d'une classe (ses attributs et méthodes) vers une sous-classe. Il peut être simple ou multiple.

L'héritage évite la duplication et encourage la réutilisation.

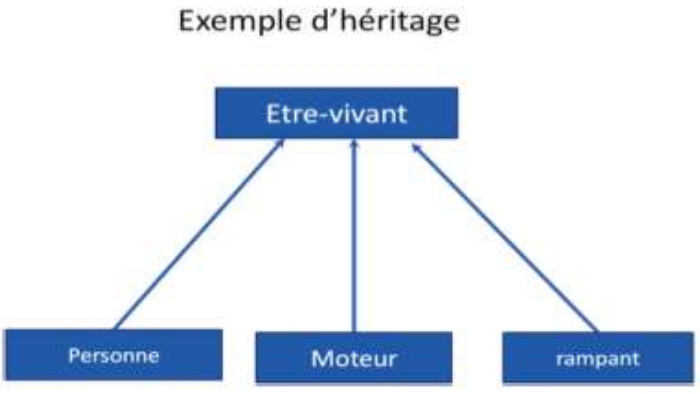

Figure : Exemple d'héritage

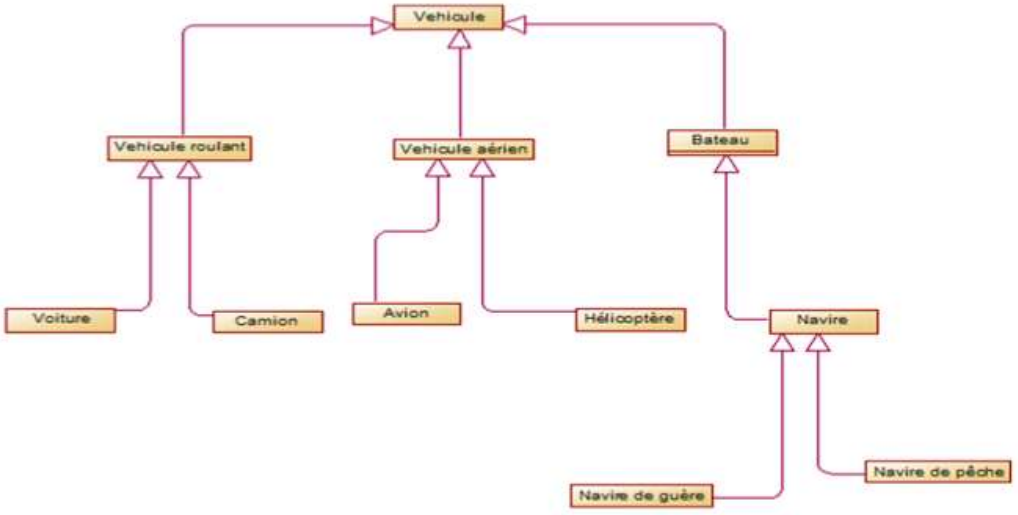

Figure : Exemple d'héritage

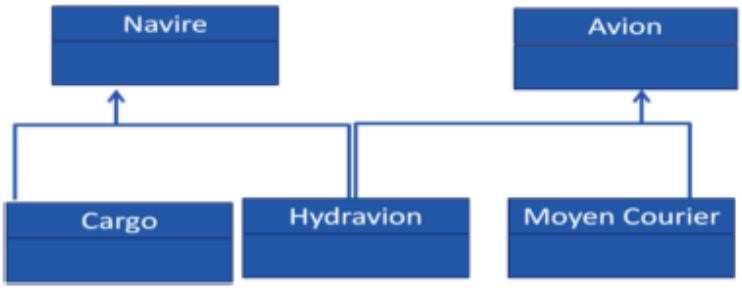

Figure : Exemple d'héritage multiple

La **redéfinition** d'une méthode héritée peut se faire de deux manières différentes:

- par réutilisation : dans ce cas, il faut spécialiser la méthode héritée en mettant une implémentation qui utilise celle héritée.

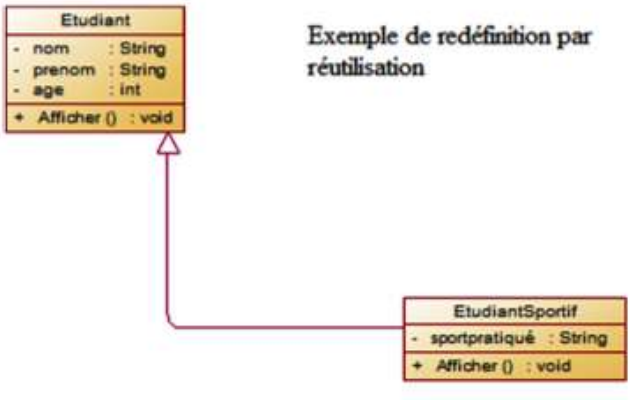

Figure: Redéfinition par réutilisation

- par substitution : On remplace complètement la méthode héritée par une nouvelle implémentation

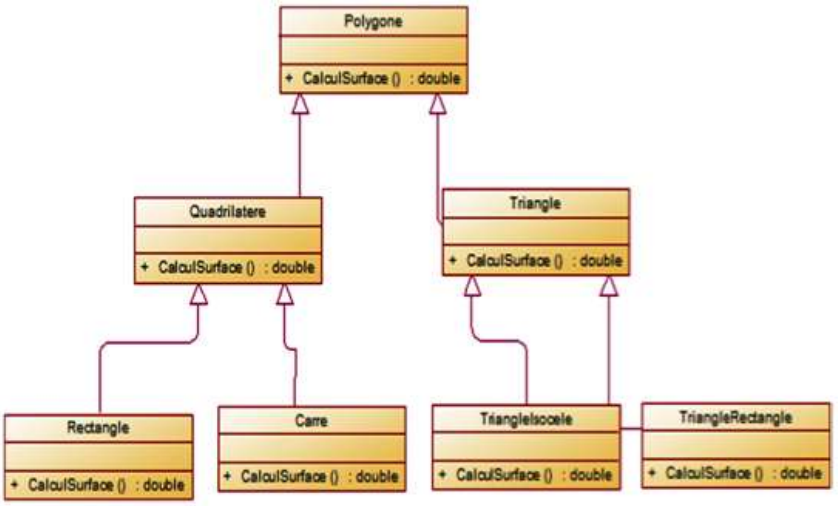

Figure: Redéfinition par réutilisation

1.4.7. Généralisation/ Spécialisation entre classes

La spécialisation et la généralisation permettent de construire des hiérarchies de classes.

- La généralisation est une démarche ascendante, qui consiste à capturer les particularités communes d'un ensemble d'objets, issues de classes différentes. Elle consiste à factoriser les propriétés d'un ensemble de classes, sous forme d'une super-classe, plus abstraite (permet de gagner en généricité).
- La spécialisation est une démarche descendante, qui consiste à capturer les particularités d'un ensemble d'objets, non discriminés par les classes déjà identifiées.

Une classe peut être spécialisée en d'autres classes, appelées sous-classes, afin d'y ajouter des caractéristiques spécifiques ou d'en adapter certaines (permet l'extension du modèle par réutilisation).

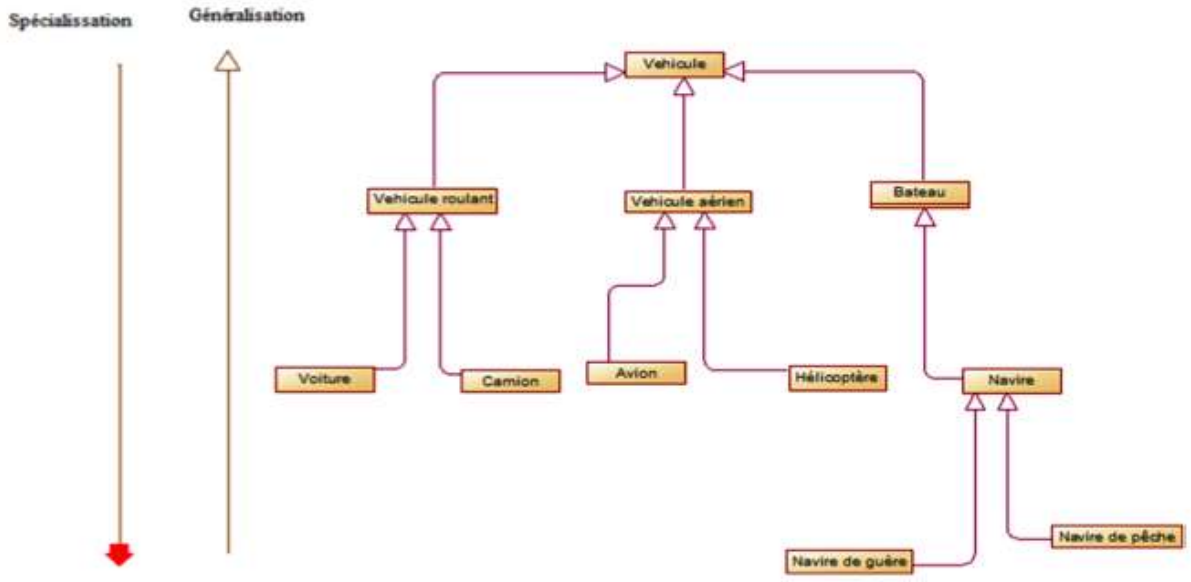

Figure: Spécialisation et généralisation

1.4.8. Polymorphisme

Le polymorphisme représente la faculté d'une méthode à pouvoir s'appliquer à des objets de classes différentes, qui héritent d'une même classe de base.

Le polymorphisme augmente la généricité du code.

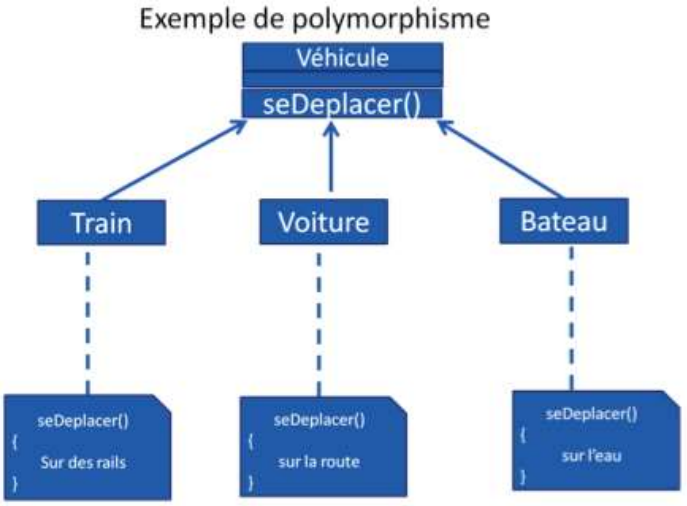

Figure: Polymorphisme