Centre de Réadaptation de Mulhouse

Initiation au concept objet

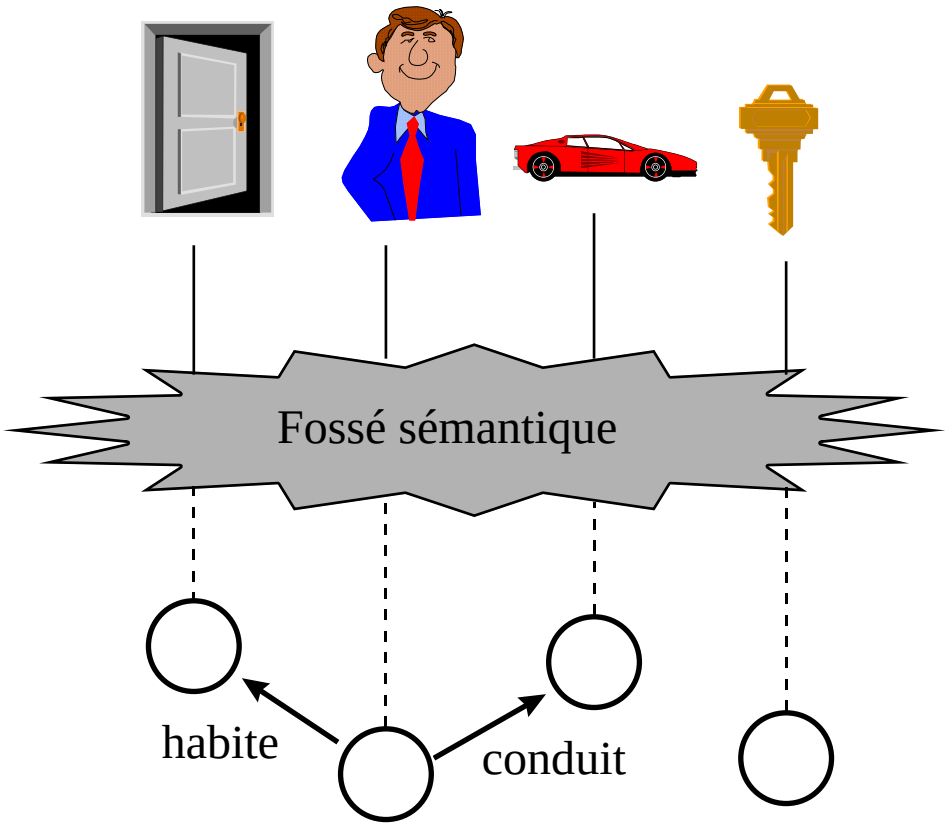
! Attention ce document n’utilise pas encore une notation UML rigoureuse !

Pourquoi les objets ?

Une première raison est d'apporter une solution pour répondre à la complexité croissante des techniques informatiques (bureautique, CAO, robotique, imagerie médicale, cartographie ...).

La deuxième idée est de diminuer le fossé sémantique entre le monde réel et sa représentation informatique.

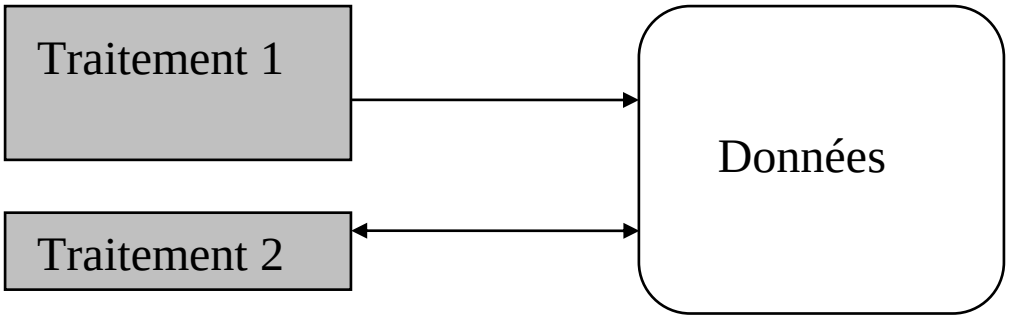
Les gens évoquent leur environnement en termes d'objets. Si l'on parle de quelque chose en lui attribuant des propriétés, ou si cette chose doit être manipulée, alors il faut la représenter sous forme d'objet.



Avec les objets, on se focalise sur la chose, sur ce qui doit être manipulé. De ce fait, concevoir par objets, c'est d'abord répondre à la question : « De quoi parle-t-on ? » avant de répondre à la question : « Que veut-on faire ? ».

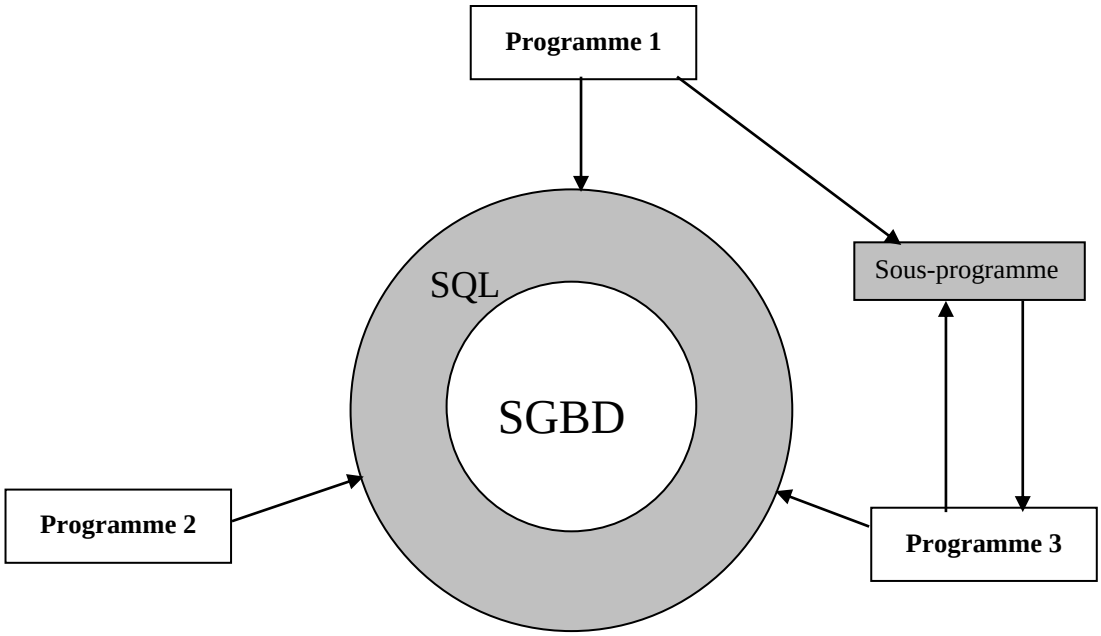
Paradigme classique

Découpage net entre données et traitement



Pour faire face à la complexité, les améliorations portent sur :

- le développement de la programmation structurée avec utilisation de sous-programmes
- l'utilisation de bases de données avec utilisation de requêtes



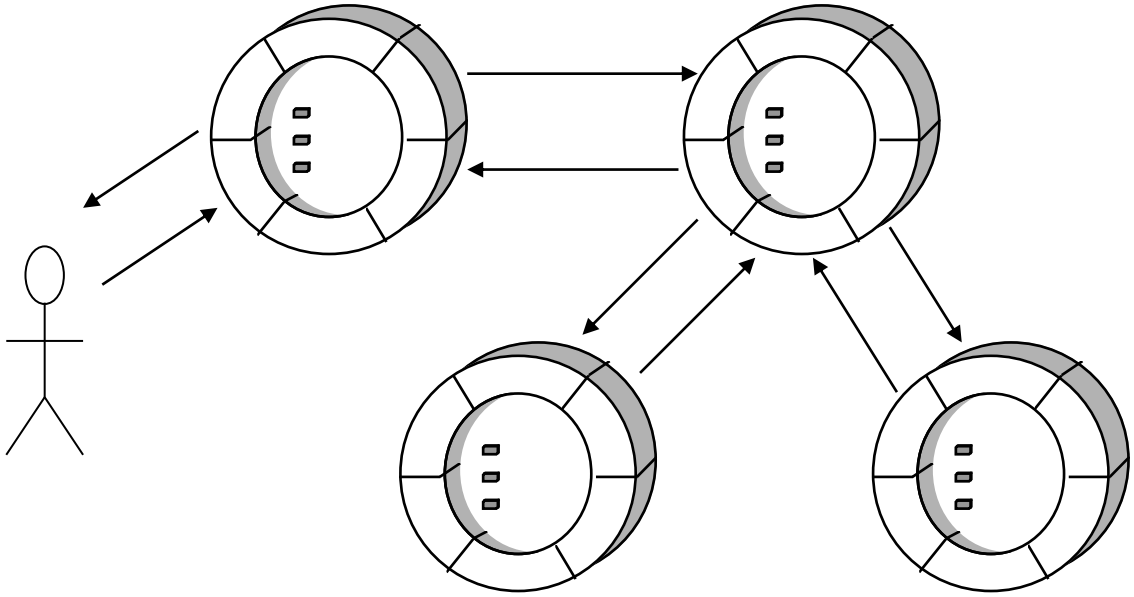
Il y a rupture entre le monde réel et sa représentation informatique.

Paradigme[1] objet

Un programme devient un ensemble de petites entités informatiques qui interagissent et communiquent par messages.

Chacune de ces entités informatiques est autonome et comprend une partie données et une partie traitement.

C'est ce qu'on appelle un **objet.**



L'utilisateur apparaît lui-même comme un premier objet qui dialogue naturellement avec les autres objets par le biais d'icônes, d'une souris, de menus déroulants et de boîtes de dialogue.

[1] Paradigme : du gr. *paradeigma* « exemple ». Modèle théorique de pensée qui oriente la recherche et la réflexion scientifiques.

Les origines

1967 : SIMULA

C’est le premier véritable langage objet. Ses auteurs cherchaient avant tout à disposer d’un outil permettant de simuler des systèmes physiques ou des organisations humaines. Né à Oslo en Norvège, il fut conçu à partir d'ALGOL. Il en reprit la syntaxe et le fonctionnement général. Les adjonctions portèrent sur la définition de classes et l’apparition d'une entité informatique regroupant une structure de données et l'ensemble des procédures pour la manipuler : l'**objet**.

1972-1976 : SMALLTALK

En 1972, au centre de recherche de XEROX à Palo Alto, Adèle GOLDBERG et un prodige de l’informatique : Alan KAY se donnent pour objectif de réaliser un ordinateur réellement convivial avec une nouvelle interface homme-machine.
Ils définissent l’interface graphique que nous connaissons aujourd’hui : souris, écran bitmap haute résolution.
Tout cet environnement se devait d'être facilement programmable. Ils décident donc de développer un langage qui reprenne l'essentiel des caractéristiques de SIMULA en ajoutant le concept d’envoi de message : SMALLTALK-72.
Le langage s’enrichit avec le concept d’héritage : SMALLTALK-76.

1980 : Alan KAY rejoint le groupe APPLE.

1984 : Sortie de LISA, puis du MACINTOSH, premier succès de l’interface graphique.

En parallèle : « travaux de l’école scandinave ». Langage C (1972).

1983 : C++

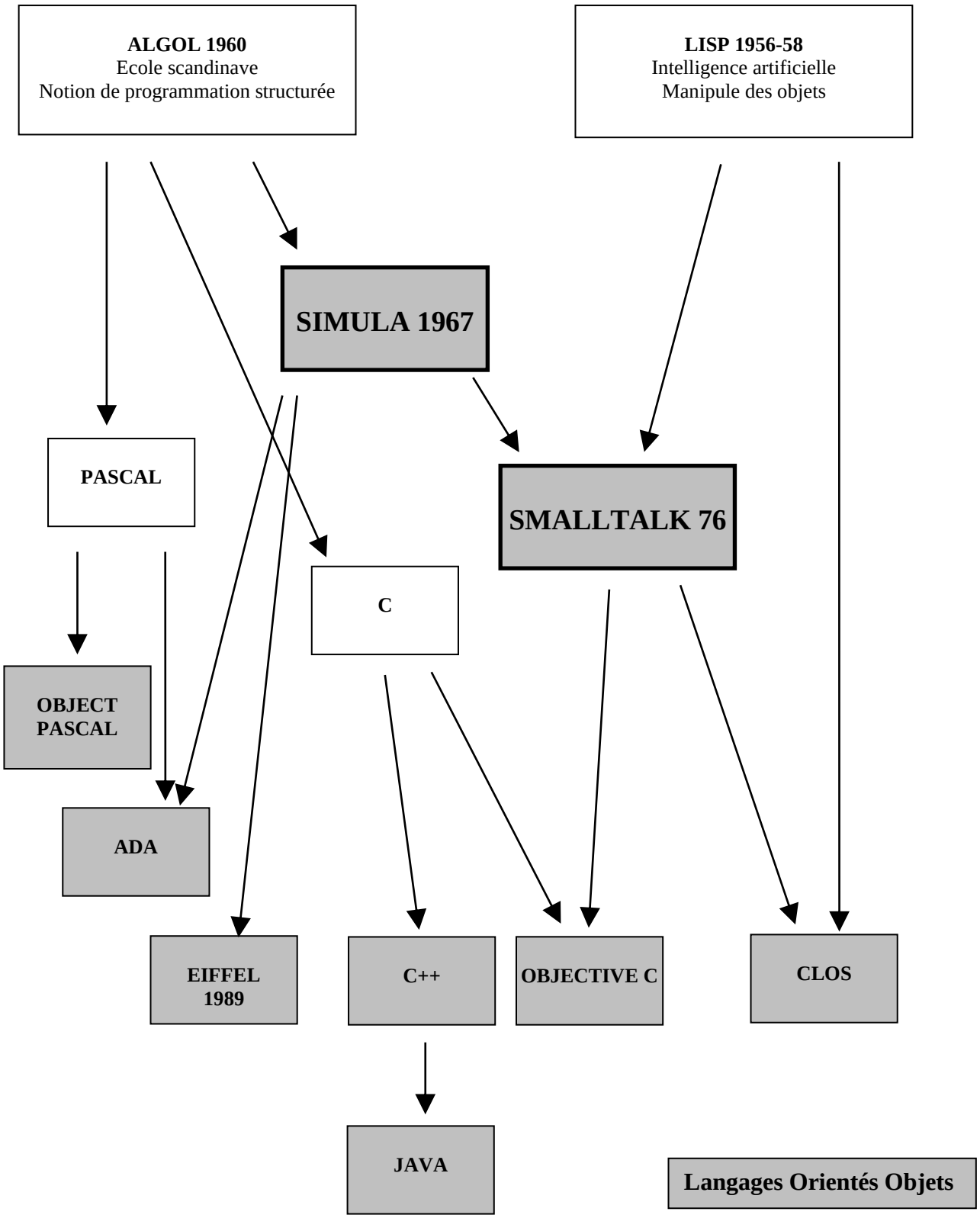
Bjarne STROUSTRUP avait appris à programmer en SIMULA.
Lorsqu'il rejoint les laboratoires Bell, berceau du langage C, il se mit en devoir d'ajouter une couche objet au-dessus du C.

1989 : EIFFEL

C'est un langage français conçu par Bertrand MEYER, mais il fut d'abord commercialisé en Californie.

… et bien d’autres, comme **JAVA** : langage interprété qui facilite la portabilité.

Historique des langages objets



Concepts de base

La programmation par objets repose sur trois concepts de base.

Un concept de *modélisation* avec les notions

d'objets, de classes et d'instances

Un concept *d'activation* avec les notions

d'envoi de messages et de méthodes

Un concept de *classification* avec les notions

de généralisation et d'héritage

Ces principes sont simples, mais combinés ils permettent une très grande puissance d'expression.

L'objet

Le monde dans lequel nous vivons est constitué d'objets matériels de toutes sortes dont la taille est très variable (du grain de sable aux étoiles). Notre perception intuitive de ce qui constitue un objet est donc fondée sur le concept de masse.

Par extension, il est tout à fait possible de définir d'autres objets sans masse mais visibles, comme des fenêtres sur un écran, des figures géométriques, mais aussi des concepts comme les comptes en banque, les équations mathématiques.

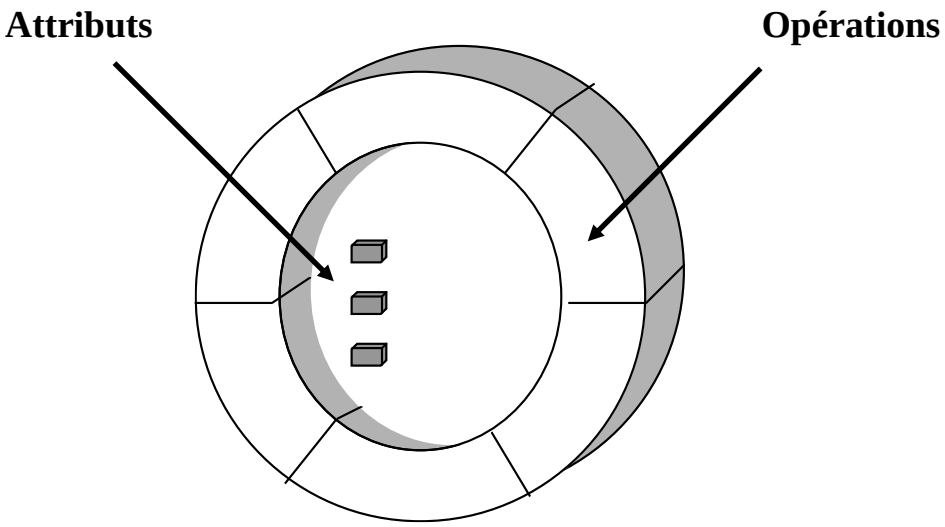
On appelle <u>réification</u> l'opération essentielle de l'approche objet par laquelle quelque chose est représentée informatiquement sous la forme d'un objet.

Réifier : (Petit Robert) transformer en chose ; syn. de « chosifier ».

Cet objet informatique peut se caractériser ainsi :

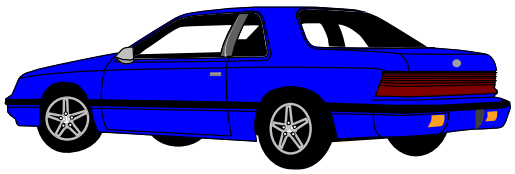
Objet = identité + état + comportement

L'identité permet de différencier chaque objet parmi les autres.
L'état est représenté par des attributs *(ou champs, variables d'instances)*.
Le **comportement** est assuré par des opérations.



Exemples d'objets

Un objet voiture



uneVoiture

Attributs :
couleur = bleue
poids = 979 kg
puissance = 12 CV
capacité carburant = 50 l
conducteur = Dupont
vitesse instantanée = 50 km/h

Opérations :
démarrer ()
déplacer ()
mettreEssence ()

Un objet fenêtre

uneFenêtre

Attributs :
point-sup-gauche
point-inf-droit
texture
en-tête

Opérations :
activer ()
fermer ()
réduire (coeff)
agrandir (coeff)
déplacer (lieu)

Un objet compte

unCompte

Attributs :
débit
crédit
titulaire

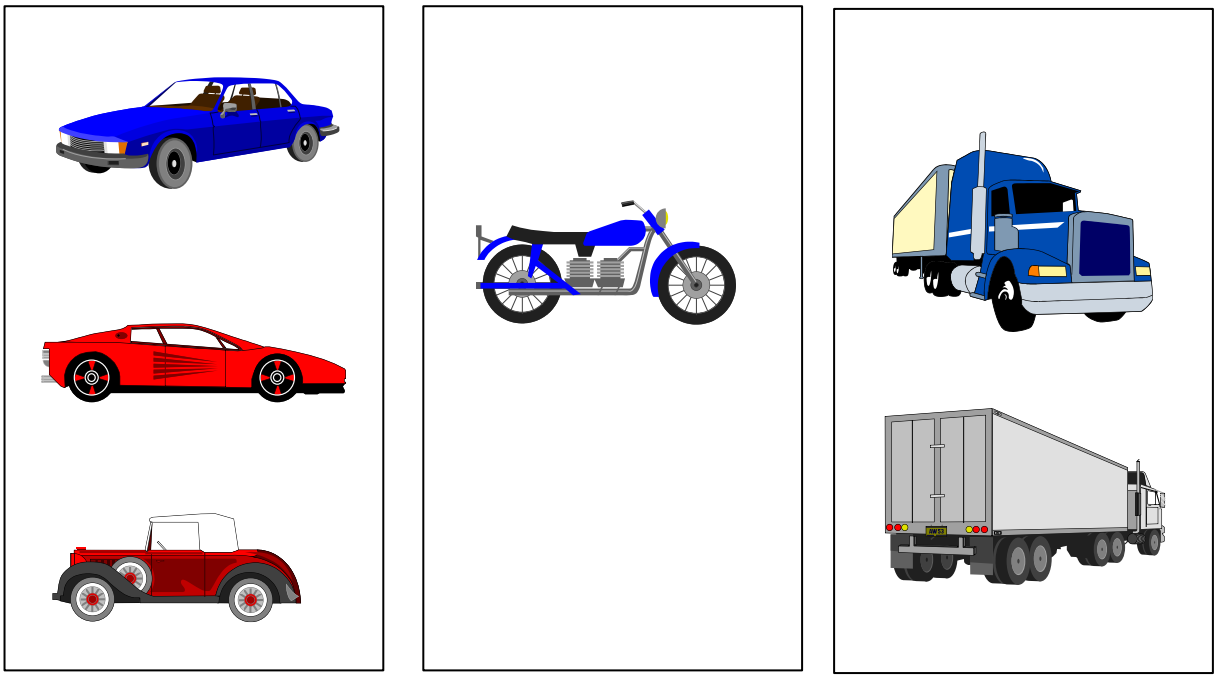
Opérations :
déposer (somme)
retirer (somme)
avoirSolde ()

Classe

Plusieurs objets peuvent posséder une structure et des comportements semblables.

La **classe** factorise les caractéristiques communes de ces objets et, comme son nom le suggère, permet d'en bâtir une *classification*.

Exemple de classes :



Les concepts d'objet et de classe sont interdépendants. En effet, un objet appartient à une classe et une classe décrit la structure et le comportement communs d'objets.

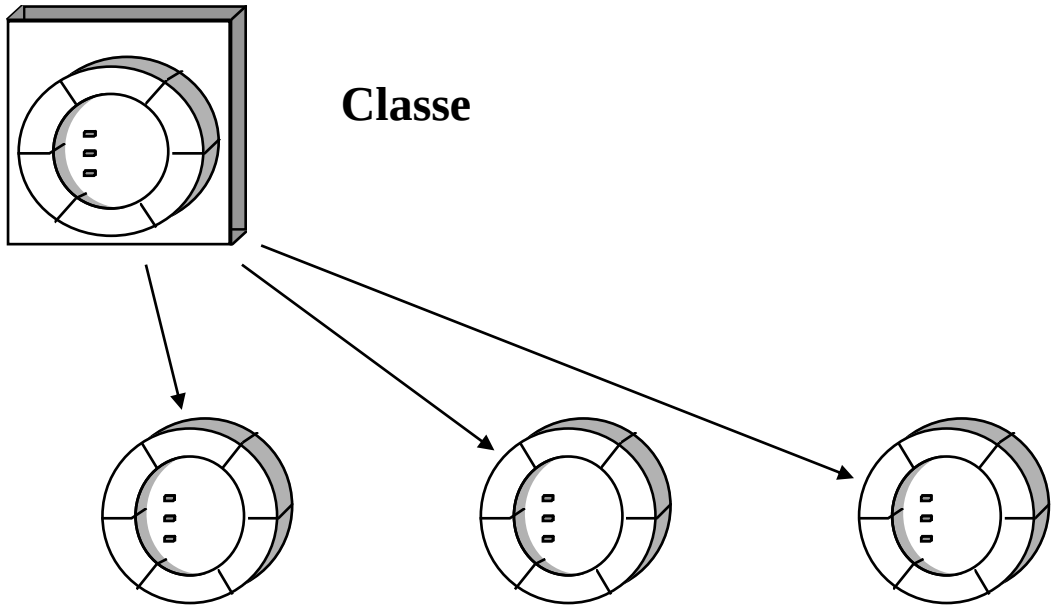
Les classes ont un double rôle :

- décrire et **classer** de façon abstraite des objets
- servir de « moule » à objets (mécanisme **d'instanciation**).

Instance

Une **instance** est un représentant particulier d'une classe.

Elle possède les mêmes attributs et les mêmes opérations que les autres instances de la classe. Les attributs prennent des valeurs distinctes dans chacune des instances.



I n s t a n c e s ...

L'**instanciation** est le processus par lequel on crée de nouveaux objets, à partir du modèle défini par une classe.

Exemple :
En Smalltalk, la création d'un objet se fait en utilisant la méthode *new :*

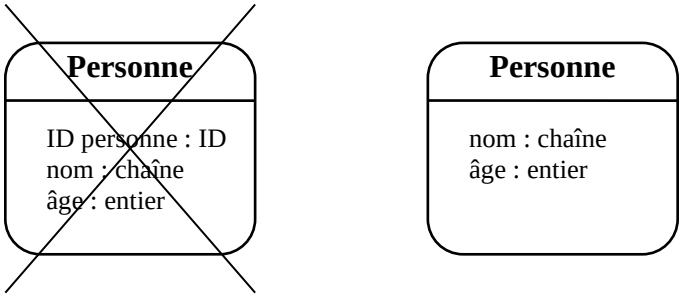
```
Compte new ('Dupont', 5000)
```

crée une instance de la classe compte avec initialisation des attributs nom et solde.

Identification de l'objet

Un objet est unique dans le système.
Son identité est représentée par un identifiant (*object identifier* ou *OID*).
Cet identifiant n’est ni modifiable, ni réutilisable car il est généré par le système lors de la création de l’objet.

Non seulement il n'est pas nécessaire, mais il est préférable de ne pas énumérer de façon explicite les identifiants.



Incorrect ***Correct***

Ne pas confondre l'identifiant interne de l’objet avec un attribut légitime, comme le numéro de sécurité sociale, qui lui a un sens dans le monde réel.

Etat de l'objet

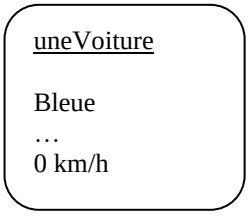
Représente *l'aspect statique* de l'objet.

> **Etat d'un objet** : situation stable pendant la vie d'un objet
> durant laquelle il peut exécuter une *activité* (qui dure),
> et/ou attendre un *événement*.

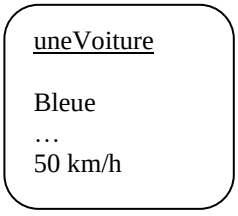
C'est la valeur des **attributs** considérés dans leur ensemble, qui reflète l'**état** dans lequel se trouve l'objet à un instant donné.

Exemples

La voiture :

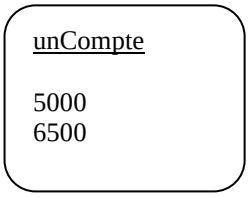


à l'arrêt

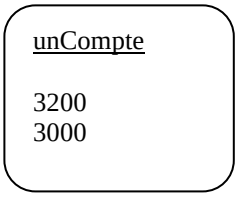


en mouvement

Le compte :



créditeur



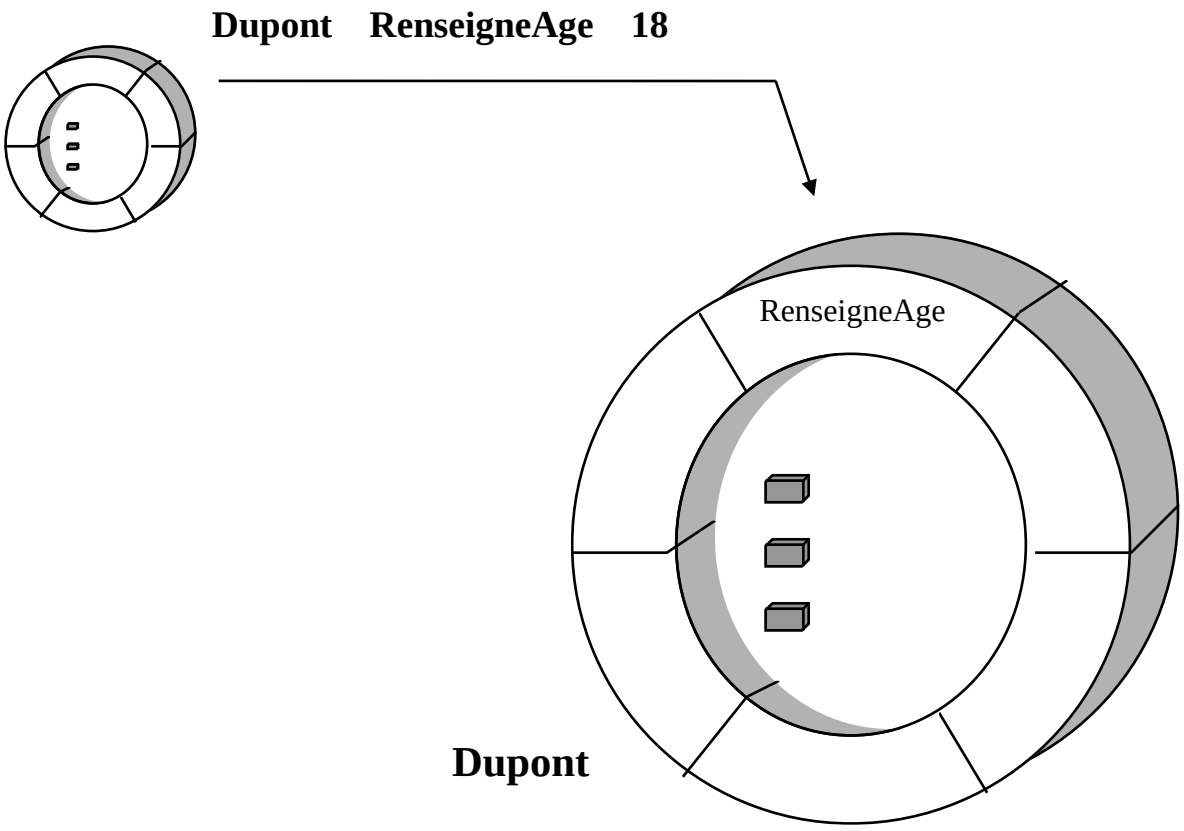
débiteur

L'état d'un objet spécifie également le contexte dans lequel les événements sont reçus. Les réactions de l'objet s'en trouveront parfois affectées.

Envoi de messages

La partie *dynamique* des objets est assurée par la notion **d'envoi de messages**.

Envoyer un message à un objet, c'est lui dire ce qu'il doit faire.



C'est **l'objet récepteur** qui traite le message en activant la **méthode** qui correspond au nom du **sélecteur** indiquée par l'émetteur.

Pour être capable de l'exécuter, l'objet récepteur peut avoir besoin d'informations supplémentaires. Le message contient alors des **paramètres**.

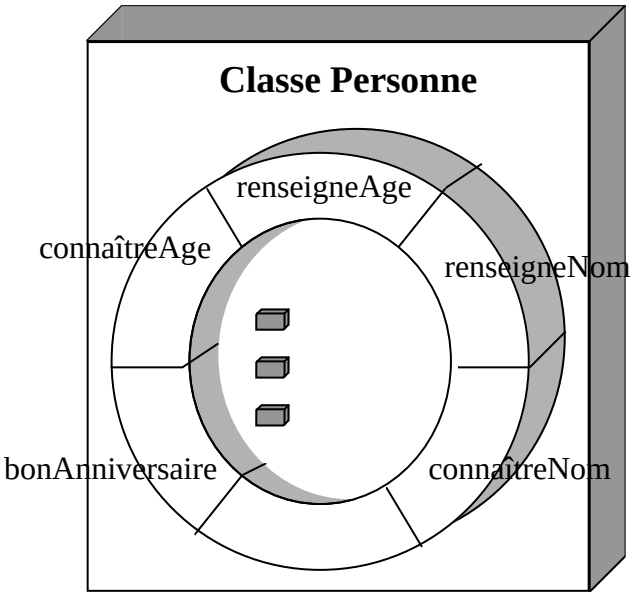
La structure d'un message est donc la suivante :

Récepteur	**Sélecteur**	**Paramètres ...**

Méthodes

Une méthode est un « petit programme » qui exécute une opération.
Les méthodes sont décrites au niveau des classes.
Chacune à un nom (sélecteur) qui permet de l'activer.

Exemple en Smalltalk :



Sélecteurs et méthodes :

renseigneAge : nbAnnée
age := nbAnnées

connaîtreAge
^ age

renseigneNom : unNom
nom := unNom

connaîtreNom
^ nom

bonAnniversaire
age := age + 1

Remarque (SMALLTALK) :

Par défaut, l'objet retourné par une méthode est l'objet receveur du message.

Si on veut que la méthode retourne le résultat de l'évaluation d'une expression donnée, il faut précéder cette expression du symbole ^.

Comportement de l'objet

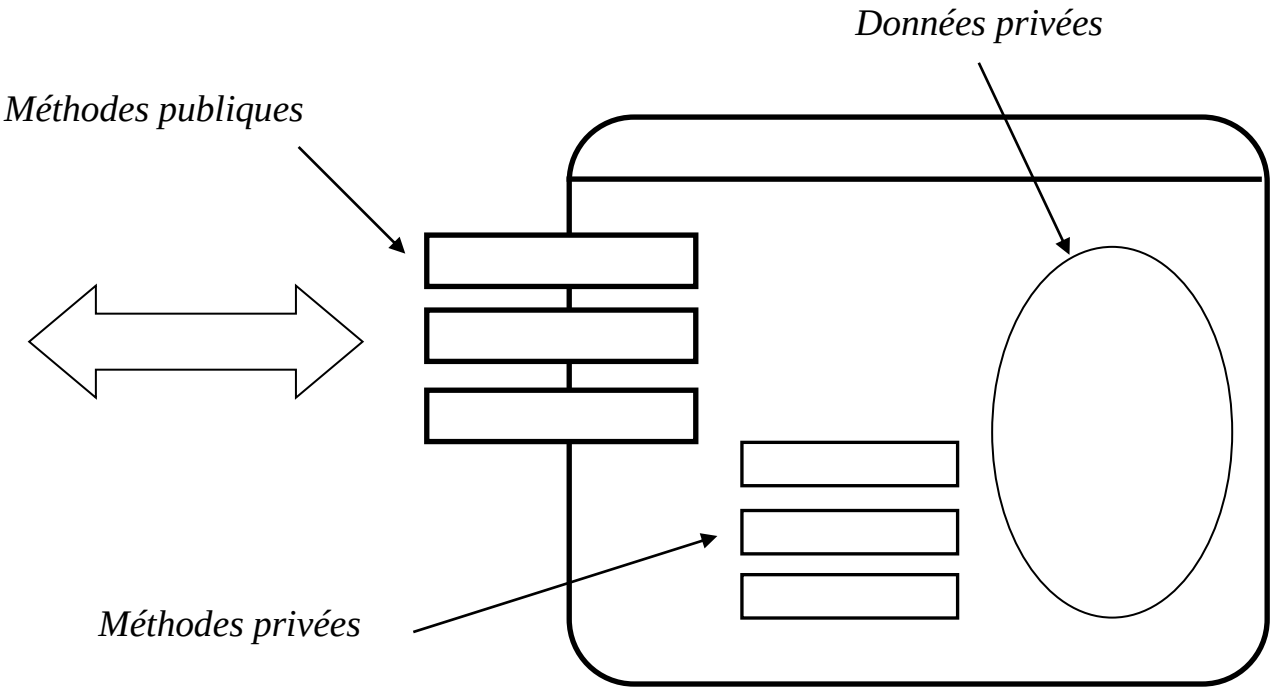
L’ensemble des méthodes permet d’assurer le comportement des objets.

Exemples de méthodes définissant le comportement d'une voiture :

Démarrer ()
Se déplacer ()
Mettre de l'essence ()

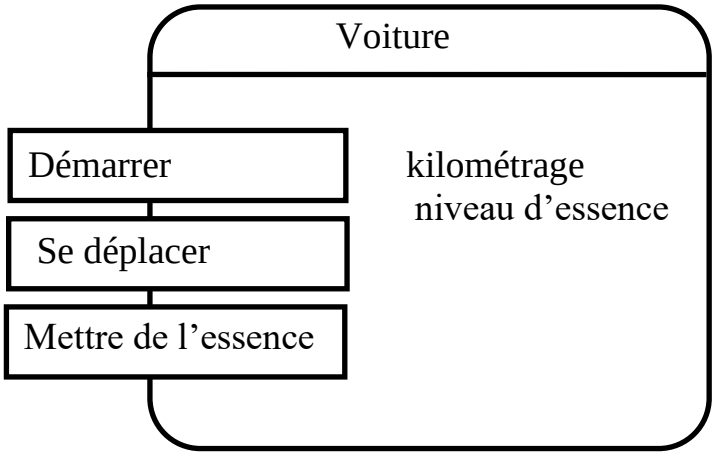
Les méthodes peuvent être publiques ou privées.

- Les **méthodes publiques** constituent l'interface de l'objet avec l'extérieur.
- Les méthodes privées sont invisibles de l’extérieur.



Encapsulation

L'**encapsulation** consiste à masquer aux utilisateurs d'un objet les détails relevant de son implémentation (vue interne ou réalisation), et de ne laisser accessible que la vue externe ou interface.



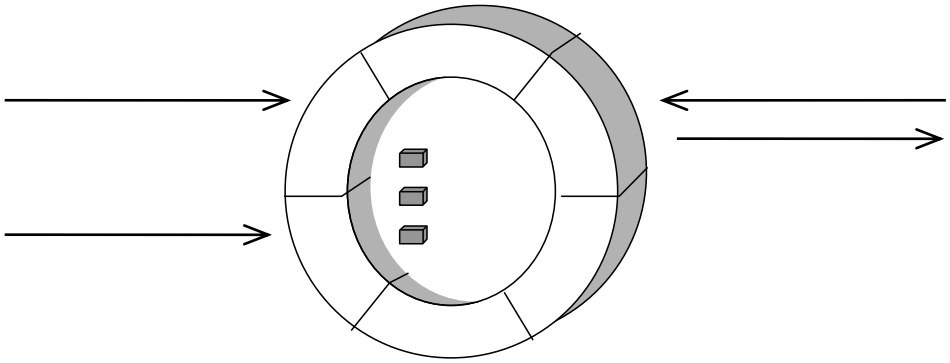
Principe :
Les attributs ne doivent pas être accessibles de l'extérieur.
Seules les opérations sont habilitées à modifier les valeurs des attributs.
En conséquence, un changement de format de ces attributs ne se répercute pas à l'extérieur.

D'autre part, les opérations peuvent apporter une valeur ajoutée et non pas se contenter d'une simple restitution d'accès aux attributs encapsulés.

Exemple : On ne peut pas faire baisser le niveau d'essence sans se déplacer, ce qui modifie aussi le kilométrage.

Il est important de noter :
- qu'un objet n'est connu que par son interface : les services qu'il peut rendre.
- qu'il est autonome : il est responsable des actions qu'il sait effectuer.

Encapsulation - avantages



- **L'intérieur de l'objet est protégé.** L'accès direct aux données est impossible. Il faut passer par une méthode (publique) constituant l'interface de l'objet pour avoir accès aux données. Ceci garantit la sécurité et l'intégrité des données.

- **La complexité est dissimulée.** Le programmeur n'a plus à se préoccuper ni du détail de la structure des données, ni de la manière dont sont effectués les traitements.

- **La maintenance est simplifiée.** La portée des modifications est limitée à l'intérieur de l'objet concerné. L'implémentation d'une opération peut évoluer sans modifier l'interface de l'objet.

- **Les échanges avec l'extérieur sont codifiés.** Seules les caractéristiques que l'on souhaite offrir aux utilisateurs potentiels de l'objet figurent dans la partie externe. Cette interface constitue un contrat entre l'objet et ses clients, c'est-à-dire entre le concepteur et ses utilisateurs.

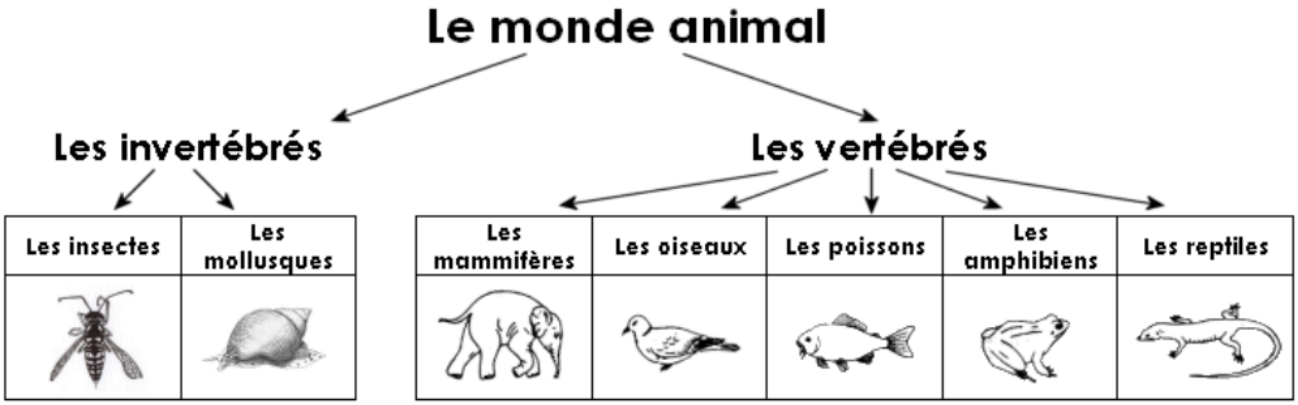
Classification

C'est dans l'invention de la hiérarchie de classes que réside le véritable génie de la technologie orientée objet. Il se trouve que la connaissance humaine est justement structurée de cette manière.

Les hiérarchies de classes ou **classifications** permettent de gérer la complexité en ordonnant les objets au sein d'arborescences de classes **d'abstraction** croissante.

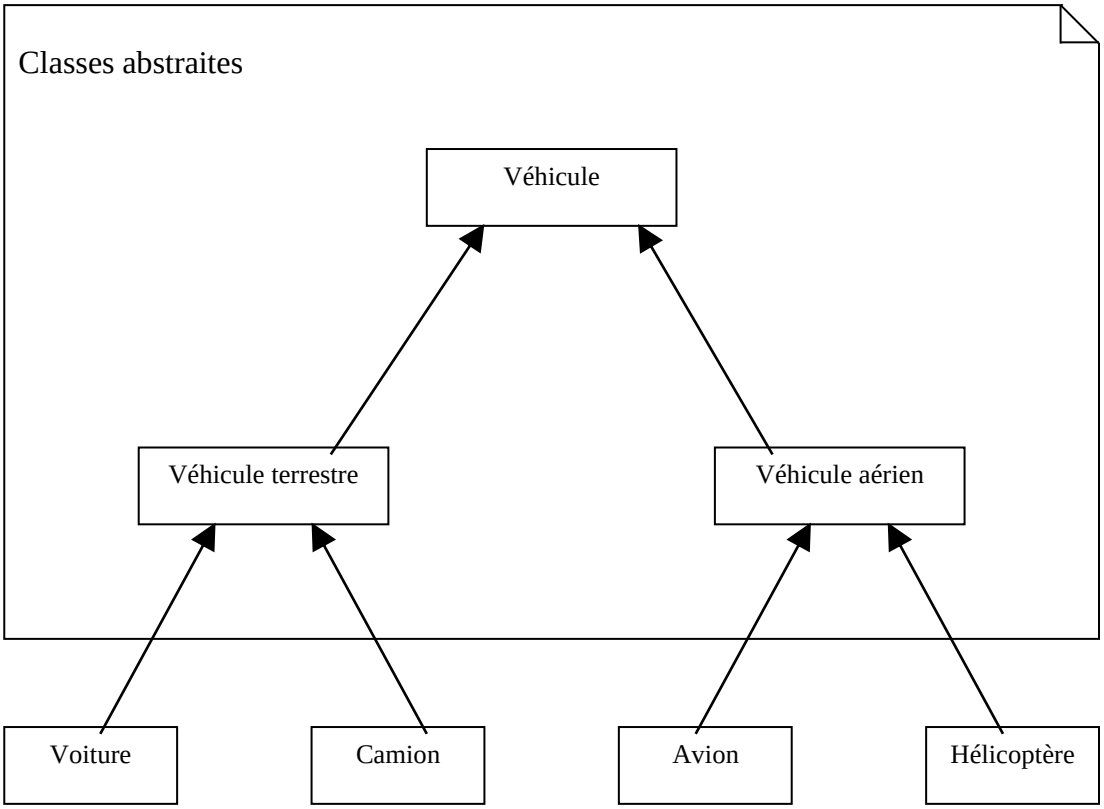
> Les classes abstraites constituent l'essence d'une application.
> *C'est là que l'essentiel se passe.*

La hiérarchisation des classes correspond à une classification des « essences ».



Généralisation

La **généralisation** consiste à factoriser les éléments communs d'un ensemble de classes dans une classe plus générale : la **super-classe** ou **sur-classe**.



La généralisation est une démarche assez difficile car elle demande une bonne capacité d'abstraction.

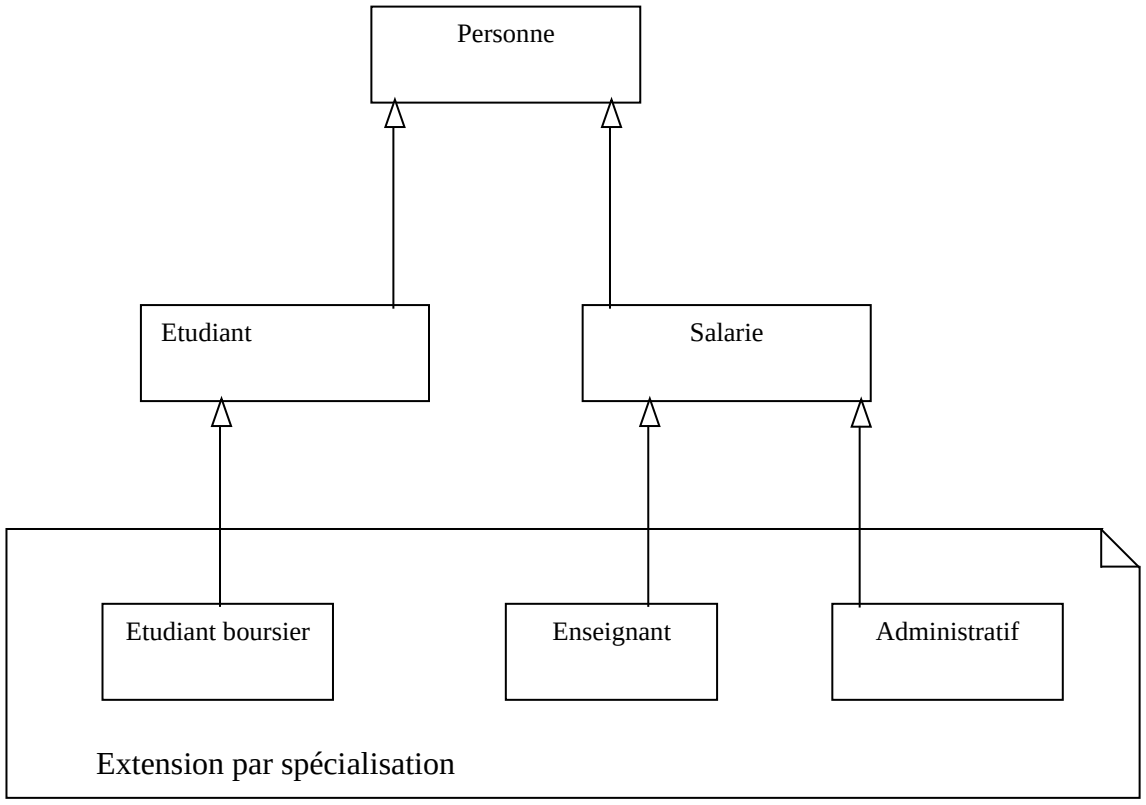
La mise au point d'une hiérarchie optimale est délicate et itérative.

Les arbres de classes sont construits en partant des feuilles qui appartiennent au monde réel alors que les niveaux supérieurs sont des abstractions construites pour ordonner et comprendre.

Spécialisation

La **spécialisation** permet de capturer les particularités d'un ensemble d'objets non discriminées par les classes déjà identifiées. Les nouvelles caractéristiques sont représentées par une nouvelle classe, **sous-classe** d'une des classes existantes.

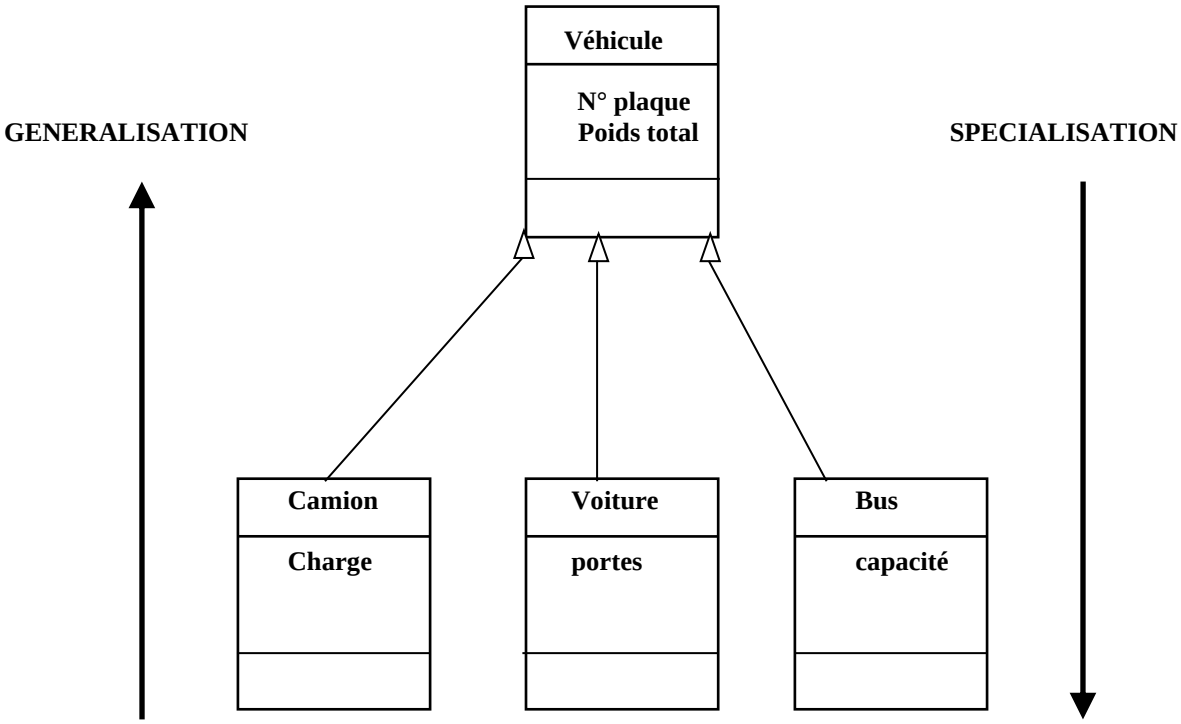
L'exemple suivant montre une classification partielle des personnes rencontrées dans une université, selon deux grandes familles. Les dispositifs concrets sont ajoutés dans la hiérarchie par spécialisation du parent le plus proche.



Généralisation et spécialisation

La généralisation et la spécialisation sont deux points de vue antagonistes du concept de classification.

Elles expriment dans quel sens une hiérarchie de classes est exploitée. Dans toute application réelle, les deux points de vue sont mis en œuvre simultanément.



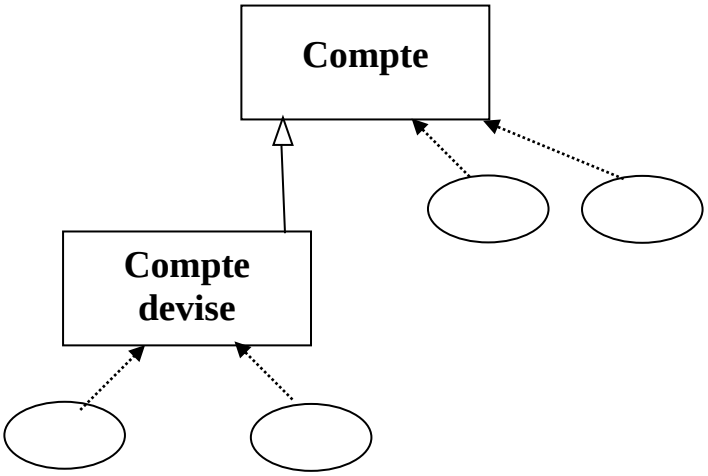
Classes abstraites

Abstraction
- **Classe abstraite :** qui ne peut pas avoir d'instance
- **Classe concrète :** qui peut avoir des instances

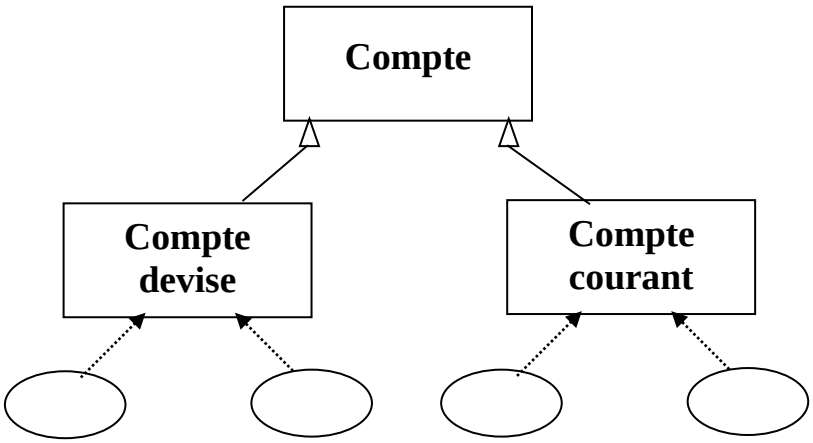
Positionnement
- **Classe terminale :** qui ne possède pas de sous-classe
- **Classe non-terminale :** qui possède des sous-classes

> Une bonne règle pour avoir des classifications équilibrées et extensibles :
> ***Faire en sorte que seules les classes terminales soient concrètes, et que les classes non-terminales soient abstraites.***

mauvais



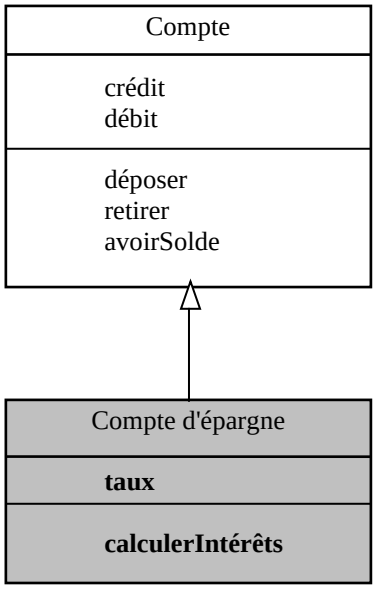
bon



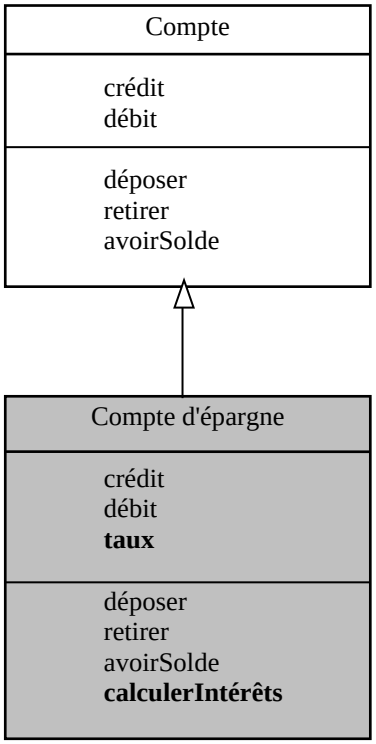
Héritage

L'**héritage** est le mécanisme de transmission des propriétés d'une sur-classe vers ses sous-classes.

Dans cet exemple :



Tout se passe comme si la sous-classe *Compte d'épargne* était décrite ainsi :



On appelle aussi les **sur-classes** des ancêtres et les **sous-classes** des descendants.

La **flèche** est orientée en direction de l'ancêtre et ne porte aucun nom particulier. Elle signifie toujours : **"est un"** (en anglais "is_a" ou ISA).
Dans notre exemple, un compte d'épargne "est un" compte.

Un autre procédé mnémonique est : "tous les trucs sont des machins".
Ici, *tous les comptes d'épargne sont des comptes*.

Héritage remarques

D’après J. Ferber :
La notion d'héritage peut être comprise comme un mécanisme de *copie virtuelle non monotone*.

En effet, tout se passe comme si toute la sur-classe était recopiée dans la sous-classe (mécanisme de copie), même si cela n'est pas effectivement implémenté de cette manière (copie virtuelle), et la recopie ne s'effectue que pour les informations qui ne sont pas définies au niveau de la sous-classe (copie non monotone).

De ce fait, la définition du mécanisme d'héritage peut s'exprimer ainsi :

1. une sous-classe **dispose** implicitement **de tous** les attributs et de toutes les méthodes définies dans la sur-classe.

2. les attributs et les méthodes définies dans la sous-classe sont **prioritaires** par rapport aux attributs et méthodes de même nom définis dans la sur-classe.

- **L’héritage évite la duplication et facilite la réutilisation.**

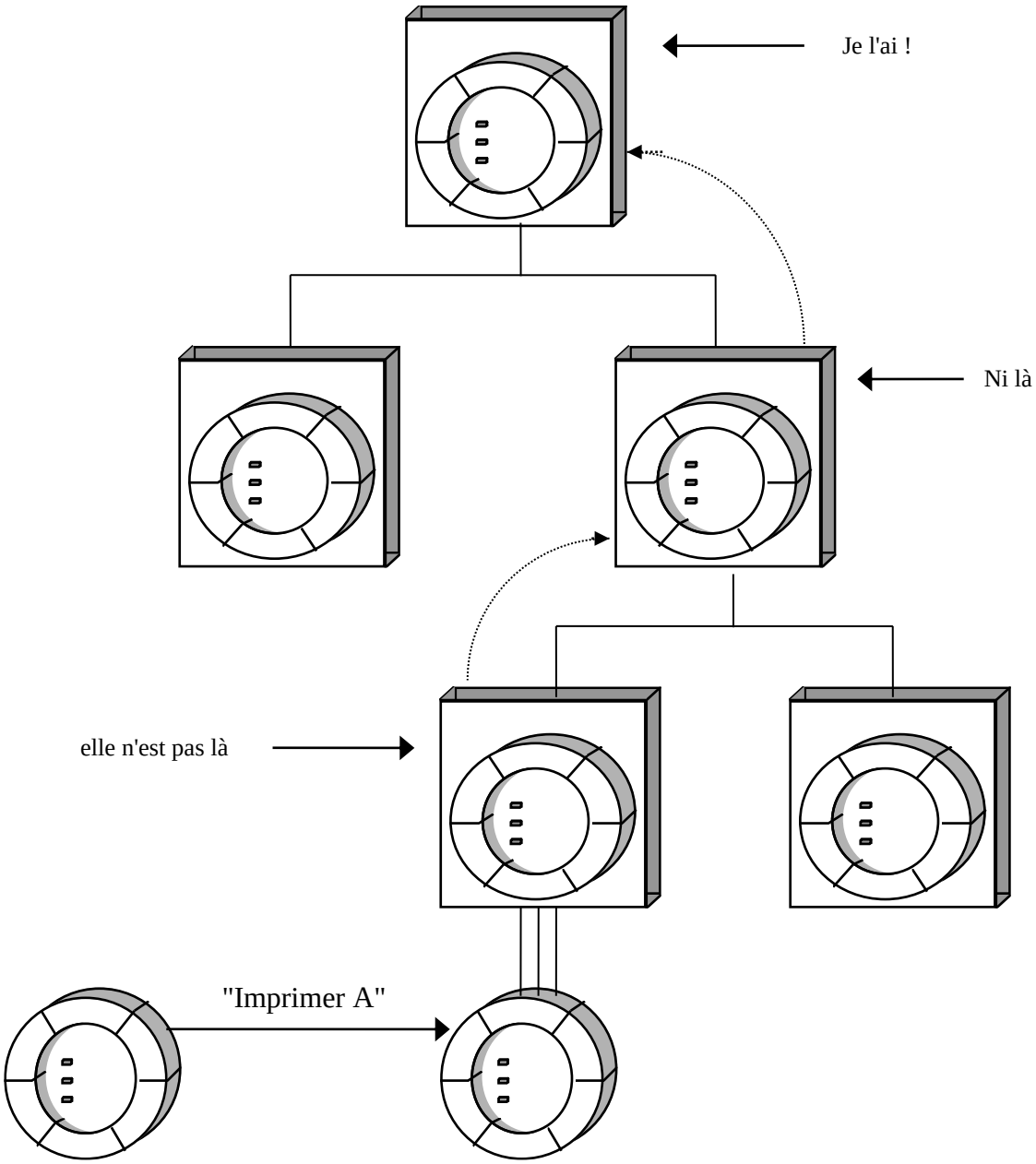
Un petit mot sur le vocabulaire :
Un problème dans ce concept de *sous*-classe et *super*-classe vient du fait que les mots « *super »* et « *sous »* ont des significations qui ne sont pas vraiment en rapport avec la manière dont on les comprend.

Le concept *super* fait souvent penser à quelque chose ayant plus de capacité que le reste. Dans ce contexte, il peut vouloir dire quasiment le contraire.
La classe qui hérite est souvent enrichie avec de nouvelles caractéristiques et dispose donc de plus de capacités que sa super-classe.

Mais un descendant peut aussi être une restriction ou une spécialisation de son ancêtre, c'est-à-dire un sous-ensemble.
En fait, c'est ce cas qui est à l'origine du concept *sous* : les instances de sous-classes représentent un sous-ensemble des instances de la super-classe.

Recherche d’une méthode

Lorsqu'un objet reçoit un message lui demandant d'exécuter une méthode qui n'est pas définie dans sa propre classe, il remonte automatiquement dans la hiérarchie des classes pour retrouver cette méthode.
Si l'objet finit par trouver la méthode, il l'exécute. Si, au contraire, il atteint bredouille le sommet de la hiérarchie, il répond par un message d'erreur : *"j'aimerais bien vous aider, mais je ne comprends pas ce que vous voulez"*.



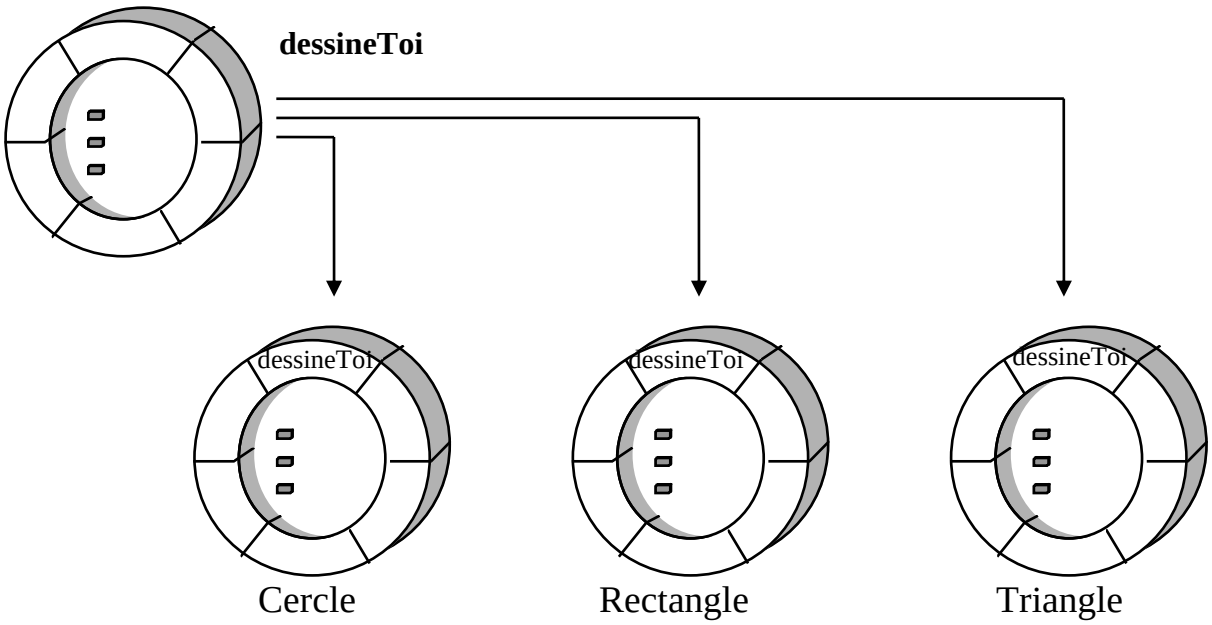
Surcharge

Considérons un programme qui permet de dessiner un grand nombre de formes différentes.
Un langage de programmation classique exigerait de donner un nom de méthode par forme à dessiner.
Le fait de demander à une forme de s'auto-dessiner deviendrait un exercice complexe :

```
Si x  type "cercle"
     Alors dessinerCercle (x)
Sinon
Si x  type "rectangle"
     Alors dessinerRectangle (x)
Sinon
Si x  type "triangle"
     Alors dessinerTriangle (x)
…
Finsi
```

En langage objet, chaque type de forme est représenté par une classe différente.
Grâce à une technique que l'on appelle la **surcharge**, on utilise le même nom pour les méthodes permettant de dessiner, à l'intérieur de chacune des classes.
Le fait de demander à une forme de s'auto-dessiner devient étonnamment simple :

```
message : [x, dessineToi]
```



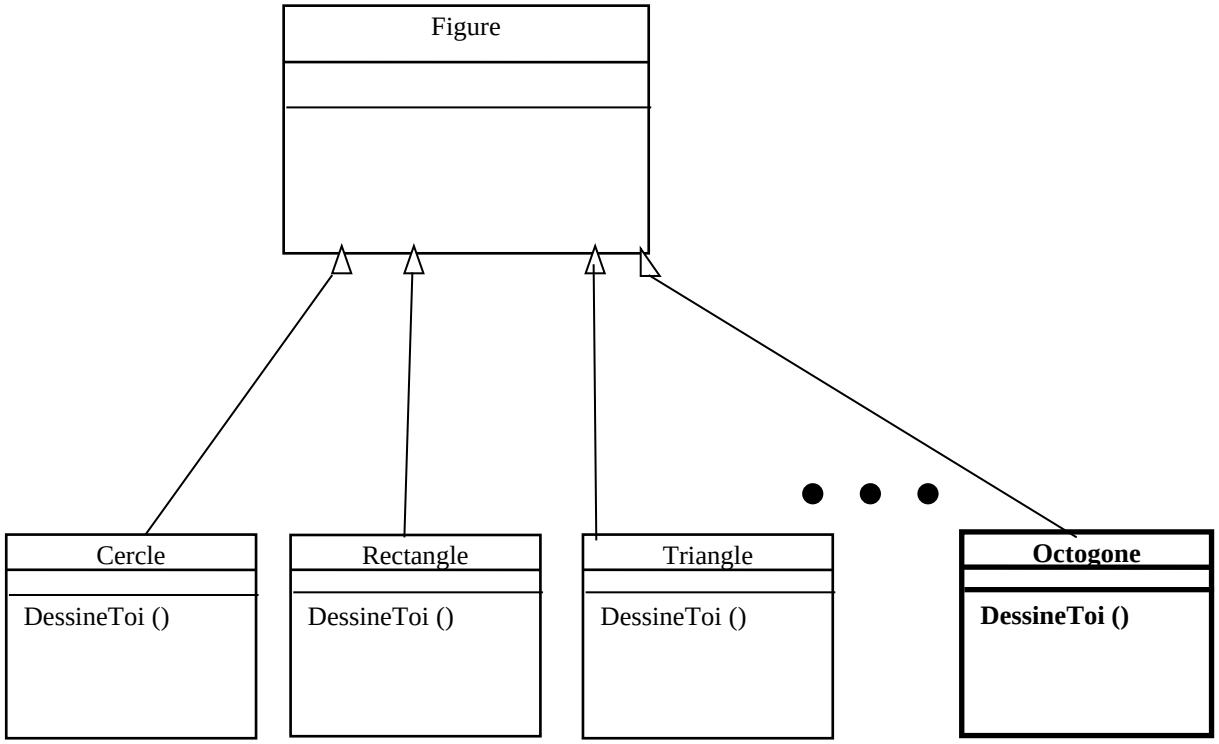
C'est au récepteur d'appliquer cette méthode à sa manière propre.

Polymorphisme

On appelle **polymorphisme** (du grec signifiant « plusieurs formes ») le fait de dissimuler des traitements différents derrière la même interface.

Le véritable intérêt du polymorphisme est de permettre l'évolution du logiciel avec un minimum de modifications.

Ajoutons une nouvelle classe (octogone) à l'exemple précédent :



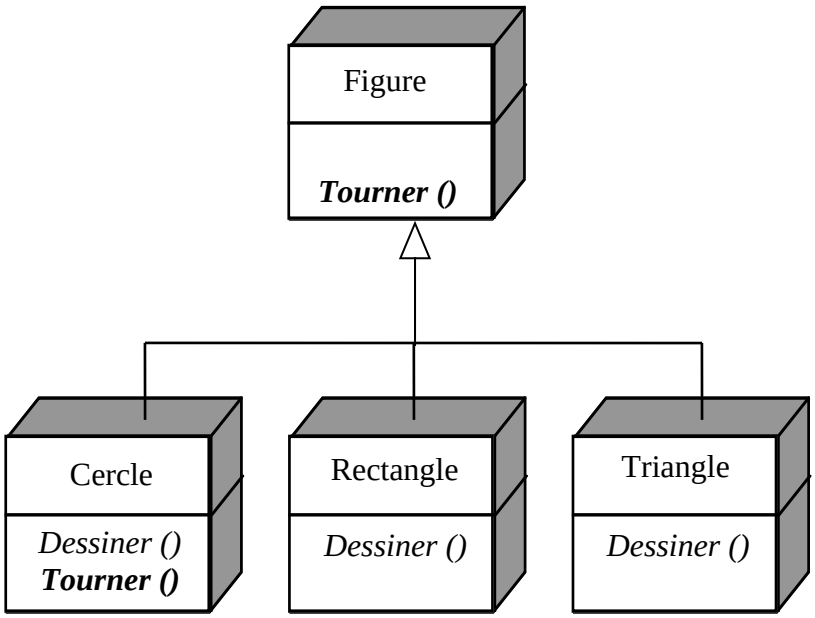
Il suffira de coder la méthode *dessineToi* dans la classe octogone.

Les objets existants n'ont même pas besoin de savoir que l'on a ajouté une nouvelle forme. Ils ne sont en rien affectés par la modification.

Cet exemple montre l'importance de la dissimulation de l'information.
Chaque objet ne doit pas trop en savoir sur les autres.

Redéfinition de méthode

Redéfinir une opération héritée avec un code différent.

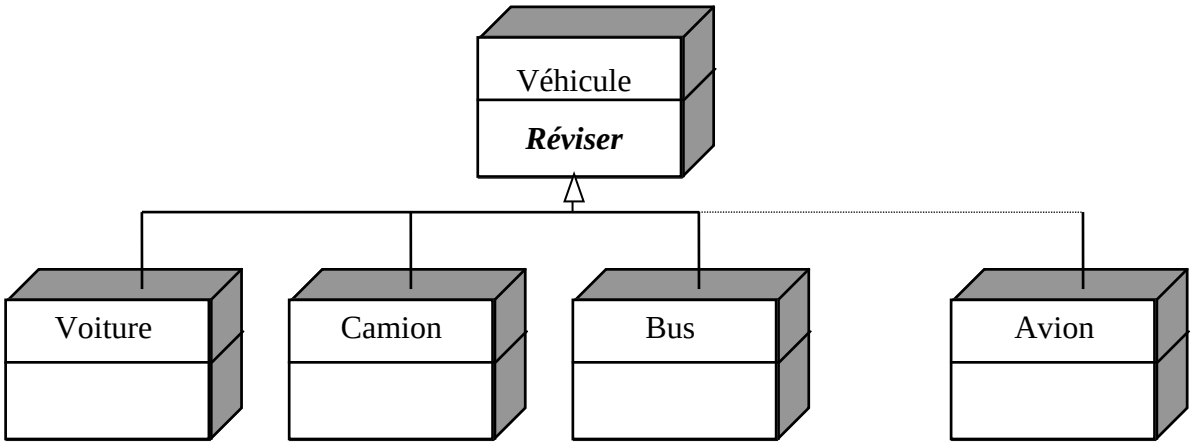


Considérons l'opération *Tourner*.
Elle est définie dans la classe *Figure* et est applicable à toutes les formes.

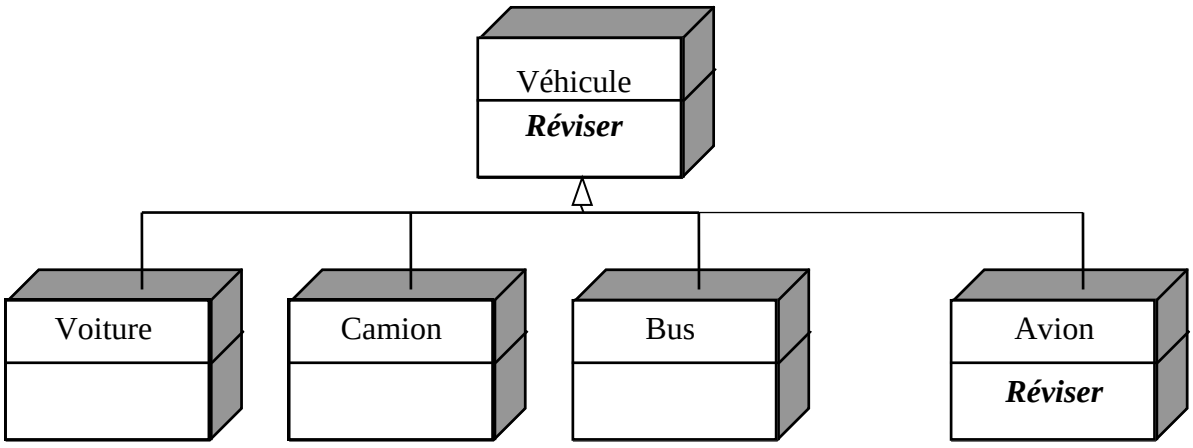
Pour des raisons de performance, on peut être amené à "surcharger" cette opération dans la sous-classe *Cercle* pour qu'elle soit une opération nulle.

Masquage de méthode

Supposons une hiérarchie de la classe des véhicules comprenant dans la super-classe véhicule une méthode *réviser.* Ajoutons maintenant la sous-classe *avion*.



La révision d'un avion n'ayant rien de semblable à celle des autres véhicules, la manière de résoudre ce problème consiste à redéfinir *Réviser* pour la sous-classe avion afin de tenir compte de ce cas particulier.



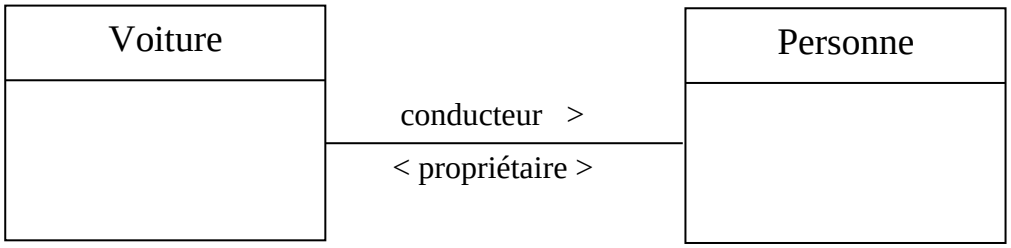
La méthode *Réviser* de la classe V*éhicule* traite le cas général.
La méthode *Réviser* de la classe A*vion* traite l'exception.

Cette technique de redéfinition d'une méthode dans une sous-classe est un cas particulier de surcharge, où le même nom de méthode est utilisé pour décrire des opérations différentes. On parle de **supplantation**, car la méthode de la sous-classe supplante la version plus générale.
L'utilisation de la surcharge est controversée au sein de la communauté objet.

Association

Au niveau objet,
certains attributs représentent des **liens** (ou relations) avec d'autres objets.
Ces liens durent au cours du temps et impliquent que deux objets connaissent chacun l'existence de l'autre.

Au niveau de la classe, **l'association** exprime une connexion sémantique bidirectionnelle entre classes.



L'association peut être exprimée dans les deux sens :

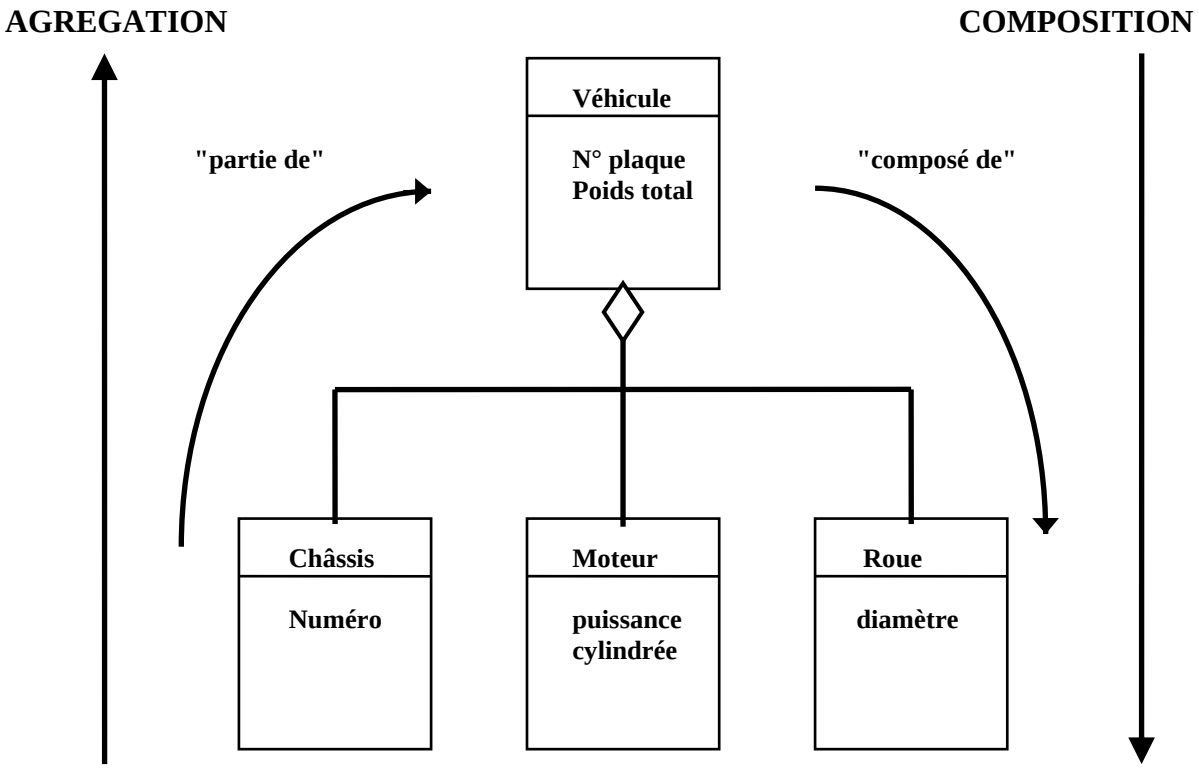
- conducteur : de *voiture* vers *personne* ;
- propriétaire : de *personne* vers *voiture*.

Méta-modèle : Un lien entre objets est une instance d'association entre classes.

Agrégation

L'**agrégation** est une relation "composé-composant" ou "partie de" dans laquelle les objets représentant les *composants* d'une chose sont associés à un objet représentant l'*assemblage* entier.
(agrégation vient du latin *aggregare* qui signifie assembler).

C'est est une forme particulière d'association qui exprime un couplage fort entre des instances d'objets.



Remarque :
On dit agrégation quand on exprime cette relation du composant (la partie) vers le composé (l'ensemble).

On dit **composition** quand on exprime cette relation du composé vers le composant. (ce qui traduit le fait qu'un objet *est composé de* plusieurs autres).