Introduction aux objets

OURS BLANC DES CARPATHES™

ISIMA

ECOLE DOCTORALE SCIENCES FONDAMENTALES

1999

Table des matières

Table des illustrations

1. Présentation de la notion d'objet

Un *objet* est une entité cohérente rassemblant des données et du code travaillant sur ses données. Une *classe* peut être considérée comme un moule à partir duquel on peut créer des objets. La notion de *classe* peut alors être considérée comme une expression de la notion de classe d'équivalence chère aux mathématiciens. En fait, on considère plus souvent que les classes sont les *descriptions* des objets (on dit que les classes sont la *méta donnée* des objets), lesquels sont des *instances* de leur classe.

Pourquoi ce vocabulaire ? une classe décrit la structure interne d'un objet : les données qu'il regroupe, les actions qu'il est capable d'assurer sur ses données. Un objet est un état de sa classe. Considérons par exemple la modélisation d'un véhicule telle que présentée par la figure suivante :

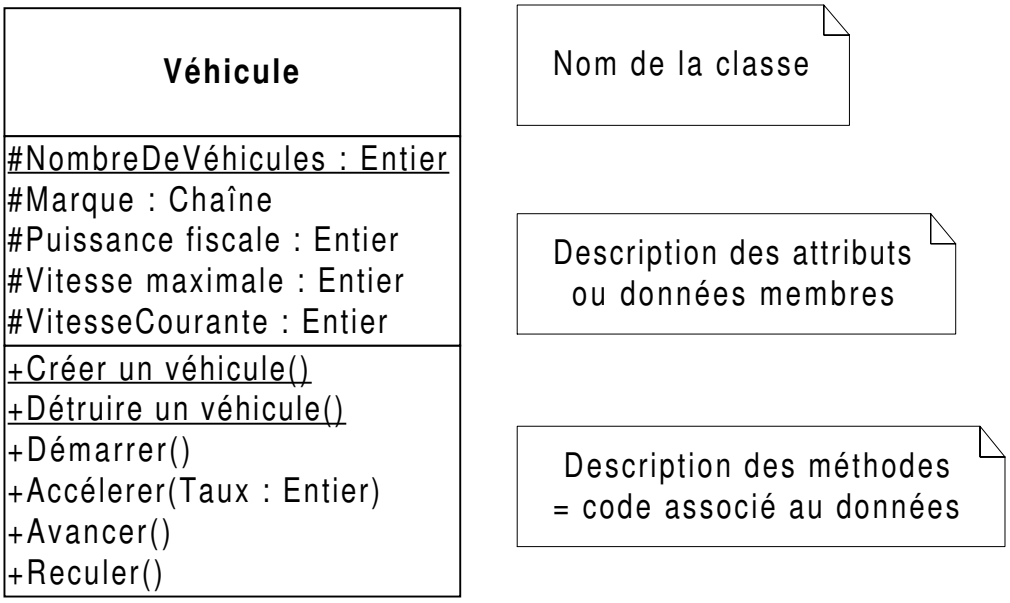

Figure 1.1 Représentation de la classe Véhicule

Dans ce modèle, un véhicule est représenté par une chaîne de caractères (sa *marque*) et trois entiers : la *puissance fiscale*, la *vitesse maximale* et la *vitesse courante*. Toutes ces données sont représentatives d'un véhicule particulier, autrement dit, chaque objet véhicule aura sa propre copie de ses données : on parle alors d'attribut *d'instance*. L'opération *d'instanciation* qui permet de créer un objet à partir d'une classe consiste précisément à fournir des valeurs particulières pour chacun des attributs d'instance.

Le schéma précédent nous permet de présenter UML (Unified Modelling language), langage de représentation des systèmes objet quasi universellement adopté de nos jours. La présente introduction ne permettra de voir qu'une infime partie d'UML au travers de son schéma de représentation statique dont le but est de présenter les différentes classes intervenant dans un modèle,

accompagnées de leurs principales relations. Nous voyons donc qu'une classe est représentée sous forme d'un rectangle lui même divisé en trois volets dans le sens de la hauteur.

❍ Le volet supérieur indique le nom de la classe

❍ Le volet intermédiaire présente les attributs et leur type sous la forme :

```
identificateurAttribut : identificateurType.
```

❍ Le volet inférieur présente les méthodes accompagnées de leur type de retour et de leurs paramètres

Le soulignement indique un membre de classe, que ce soit un attribut ou une méthode.

Finalement, les rectangles avec un coin replié sont associés aux notes ou commentaires explicatifs.

En revanche, considérons l'attribut Nombre de véhicules chargé de compter le nombre de véhicules présents à un moment donné dans la classe. Il est incrémenté par l'opération Créer un véhicule et décrémenté par l'opération Détruire un véhicule. C'est un exemple typique d'attribut partagé par l'ensemble des objets d'une même classe. Il est donc inutile et même dangereux (penser aux opérations de mise à jour) que chaque objet possède sa copie propre de cet attribut, il vaut mieux qu'ils partagent une copie unique située au niveau de la classe. On parle donc d'attribut de classe.

La figure suivante montre l'opération d'instanciation de la classe en 2 objets différents :

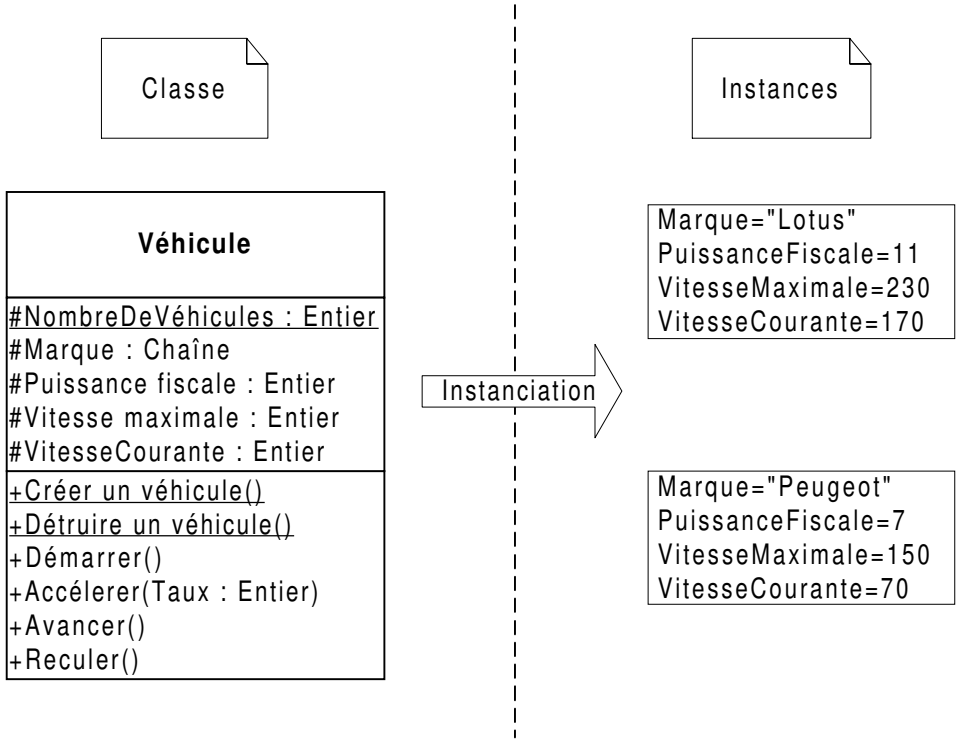

Figure 1.2 Instanciation d'une classe en deux objets

Le même raisonnement s'applique directement aux méthodes. En effet, de la même manière que nous avions établi une distinction entre attributs d'instance et attributs de classe, nous allons différencier méthodes d'instances et méthodes de classe.

Prenons par exemple la méthode Démarrer. Il est clair qu'elle peut s'appliquer individuellement à chaque véhicule pris comme entité séparée. En outre, cette méthode va clairement utiliser les attributs d'instance de l'objet auquel elle va s'appliquer c'est donc une méthode d'instance.

Considérons désormais le cas de la méthode Créer un véhicule. Son but est de créer un nouveau véhicule, lequel aura, dans un second temps, le loisir de positionner des valeurs initiales dans chacun de ces attributs d'instance. Si nous considérons en détail le processus permettant de créer un objet, nous nous apercevons que la première étape consiste à allouer de la mémoire pour le nouvel objet. Hors cette étape n'est clairement pas du ressort d'un objet : seule la classe possède suffisamment d'informations pour la mener à bien : la création d'un objet est donc une méthode de classe. Notons également qu'au cours de cette étape, l'objet recevra des indications additionnelles, telles que, par exemple, une information lui indiquant à quelle classe il appartient. En revanche, considérons la phase d'initialisation des attributs. Celle-ci s'applique à un objet bien précis : celui en cours de création. L'initialisation des attributs est donc une méthode d'instance.

Nous aboutissons finalement au constat suivant : la création d'un nouvel objet est constituée de deux phases :

- Une phase du ressort de la classe : allouer de la mémoire pour le nouvel objet et lui fournir un contexte d'exécution minimaliste
- Une phase du ressort de l'objet : initialiser ses attributs d'instance

Si ces deux phases sont clairement séparées dans un langage tel que l'objective C, elles ne le sont plus en C++ ou en Java qui agglomèrent toute l'opération dans une méthode spéciale, ni vraiment méthode d'instance, ni méthode de classe : le constructeur.

2. Le principe d'encapsulation

Sans le savoir, vous avez déjà mis le doigt sur l'un des trois grands principes du paradigme objet : l'encapsulation. Ce principe, digne héritier des principes d'abstraction de données et d'abstraction procédurale prône les idées suivantes :

- Un objet rassemble en lui même ses données (les attributs) et le code capable d'agir dessus (les méthodes)
- Abstraction de données : la structure d'un objet n'est pas visible de l'extérieur, son interface est constituée de messages invocables par un utilisateur. La réception d'un message déclenche l'exécution de méthodes.
- Abstraction procédurale : Du point de vue de l'extérieur (c'est-à-dire en fait du client de l'objet), l'invocation d'un message est une opération atomique. L'utilisateur n'a aucun élément d'information sur la mécanique interne mise

en œuvre. Par exemple, il ne sait pas si le traitement requis a demandé l'intervention de plusieurs méthodes ou même la création d'objets temporaires etc.

Dans les versions canoniques du paradigme objet, les services d'un objet ne sont invocables qu'au travers de messages, lesquels sont individuellement composés de :

- Un nom
- Une liste de paramètres en entrée
- Une liste de paramètres en sortie

La liste des messages auxquels est capable de répondre un objet constitue son interface : c'est la partie publique d'un objet. Tout ce qui concerne son implémentation doit rester caché à l'utilisateur final : c'est la partie privée de l'objet. Bien entendu, tous les objets d'une même classe présentent la même interface, en revanche deux objets appartenant à des classes différentes peuvent présenter la même interface. D'un certain point de vue, l'interface peut être vue comme un attribut de classe particulier.

En pratique, dans la plupart des langages orientés objet modernes, l'interface représente la liste des méthodes accessibles à l'utilisateur.

Il est très important de cacher les détails les détails d'implémentation des objets à l'utilisateur. En effet, cela permet de modifier, par exemple la structure de données interne d'une classe (remplacer un tableau par une liste chaînée) sans pour autant entraîner de modifications dans le code de l'utilisateur, l'interface n'étant pas atteinte. Autre exemple : considérons une classe modélisant un point en 2 dimensions pour laquelle on fournit deux méthodes renvoyant respectivement l'abscisse et l'ordonnée du point. Peu importe à l'utilisateur de savoir que le point est stocké réellement sous forme cartésienne ou si le concepteur a opté pour la représentation polaire !

Tous les langages orientés objet n'imposent pas de respecter le principe d'encapsulation. C'est donc au programmeur de veiller personnellement au grain.

3. L'héritage

L'héritage est le second des trois principes fondamentaux du paradigme orienté objet. Il est chargé de traduire le principe naturel de Généralisation / Spécialisation.

En effet, la plupart des systèmes réels se prêtent à merveille à une classification hiérarchique des éléments qui les composent. La première idée à ce sujet est liée à l'entomologie et aux techniques de classification des insectes en fonction de divers critères. Expliquons nous d'abord sur le terme d'héritage. Il est basé sur l'idée qu'un objet spécialisé bénéficie ou hérite des caractéristiques de l'objet le plus général auquel il rajoute ses éléments propres.

En terme de concepts objets cela se traduit de la manière suivante :

- On associe une classe au concept le plus général, nous l'appellerons *classe de base* ou *classe mère* ou *super - classe*.

Ceci est un exemple, cliquez sur le lien de téléchargement pour obtenir le cours complet.