Module 0-M02

INTRODUCTION À LA PROGRAMMATION
Introduction à la conception orientée objet

Camille Coti
camille.coti@iutv.univ-paris13.fr
IUT de Villetaneuse - Département R&T - 2011-2012

Table des matières

1 Introduction

La programmation orientée object (POO) est un paradigme de conception de logiciels. Il étend la programmation impérative, où la succesion d'opérations à effectuer par le programme sont définies par une séquence d'instructions.

La POO définit des *objets*, qui sont les briques de base d'un programme. Ces objets interagissent entre eux via des actions qu'ils peuvent réaliser les uns sur les autres, en modifiant éventuellement des données qui leurs sont internes.

La POO fournit donc un niveau d'*abstraction* supérieur à la programmation impérative telle que nous l'avons vu dans le cours d'initiation à l'algorithmique. On agit sur des structures de données bien définies (les objets) qui *encapsulent* les spécifications des éléments qu'elles représentent.

2 Classes, objets

Une classe est la définition d'un de ces objets. On y définit les caractéristiques de ces objets (les attributs) et les actions pouvant être réalisées (les méthodes).

2.1 Attributs

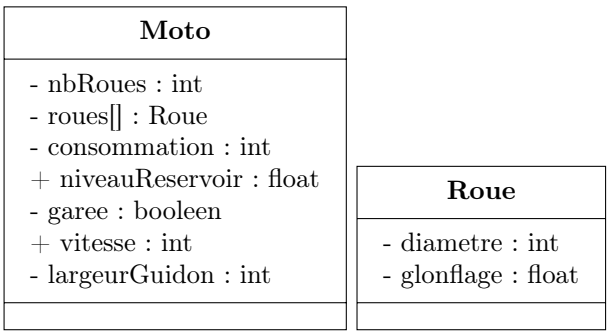
Les attributs sont les données relatives à la classe. Ce sont des variables qui concernent l'état de l'objet manipulé, et elles sont conservées dans la mémoire de l'objet.

Prenons par exemple le cas d'une moto. Une moto a des roues, un réservoir d'essence plus ou moins rempli, elle roule à une vitesse donnée (qui peut être nulle) et elle peut être garée ou non.

La classe Moto a donc les attributs suivants :
- `nbRoues`, qui est un entier
- `roues[]`, qui est un tableau de Roue (la classe Roue est donc à définir par ailleurs)
- `consommation`, qui est un entier
- `niveauReservoir`, qui est un flottant compris entre 0 et 1, 0 signifiant que le réservoir est vide et 1 qu'il est plein
- `garee`, qui est un booléen
- `vitesse`, qui est un entier
- `largeurGuidon`, qui est un entier

On doit donc définir une classe Roue, qui a les attributs suivants :
- `diametre`, qui est un entier
- `gonflage`, qui est un flottant compris entre 0 et 1, 0 signifiant que le pneu est complètement dégonflé et 1 qu'il est complètement gonflé

On peut représenter ces deux classes dans un diagramme de classe comme suit :



2.2 Méthodes

Les méthodes sont les actions que l'on peut réaliser sur les objets. Ce sont des fonctions ou des procédures qui définissent le comportement des objets d'une classe.

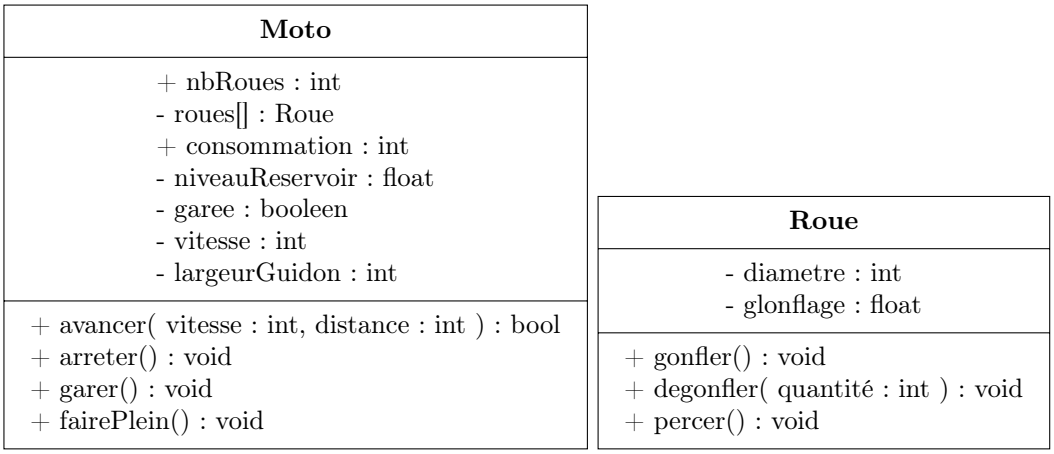
Si on reprend la classe Roue vue précédemment, on peut effectuer les actions suivantes sur ses objets :

- `void gonfler()`, c'est-à-dire mettre son attribut gonflage à 1. Cette méthode est une procédure : elle ne retourne rien
- `void degonfler( int quantité )`, c'est-à-dire retirer une certaine quantité d'air (passé en paramètre). Cette méthode est une procédure : elle ne retourne rien
- `void percer()`, c'est-à-dire mettre son attribut gonflage à 0 (retirer tout l'air de la roue). Cette méthode est une procédure : elle ne retourne rien

La classe Moto peut posséder les méthodes suivantes :

- `bool avancer( int vitesse, int distance )` : la moto se déplace, donc sa vitesse est modifiée et de l'essence est consommée. Elle retourne un booléen : `true` si tout s'est bien passé, `false` sinon (par exemple, si il n'y a pas assez d'essence). Si elle était garée, son attribut `garee` est mis à `false`.
- `void arreter()` : la moto s'arrête, c'est-à-dire que sa vitesse est mise à 0
- `void garer()` : la moto s'arrête si elle n'est pas déjà arrêtée, et son attribut `garee` est mis à `true`
- `void fairePlein()` : on remplit le réservoir d'essence

Le diagramme de classes est alors complété avec les méthodes :



3 Relations entre les classes

Considérons maintenant deux autres classes : une classe Voiture et une classe Camion. Ces classes disposent de quelques attributs et méthodes communes avec la classe Moto, et quelques attributs et méthodes qui leur sont propres.

Voiture
+ nbRoues : int - roues[] : Roue + consommation : int - niveauReservoir : float - garee : booleen - vitesse : int - diametreVolant : int - capaCoffre : int - remplissageCoffre : float
+ avancer(vitesse : int, distance : int) : bool + arreter() : void + garer() : void + fairePlein() : void + remplirCoffre(volume : float) : bool + viderCoffre( ) : void

Camion
+ nbRoues : int - roues[] : Roue + consommation : int - niveauReservoir : float - garee : booleen - vitesse : int - diametreVolant : int - nbRemorques : int - remorques[] : Remorque
+ avancer(vitesse : int, distance : int) : bool + arreter() : void + garer() : void + fairePlein() : void + ajouterRemorque() : void

3.1 Héritage

On remarque que nos trois classes Moto, Voiture et Camion ont un certain nombre d'attributs et de méthodes en commun. On peut alors les regrouper dans une classe que nous appellerons Vehicule :

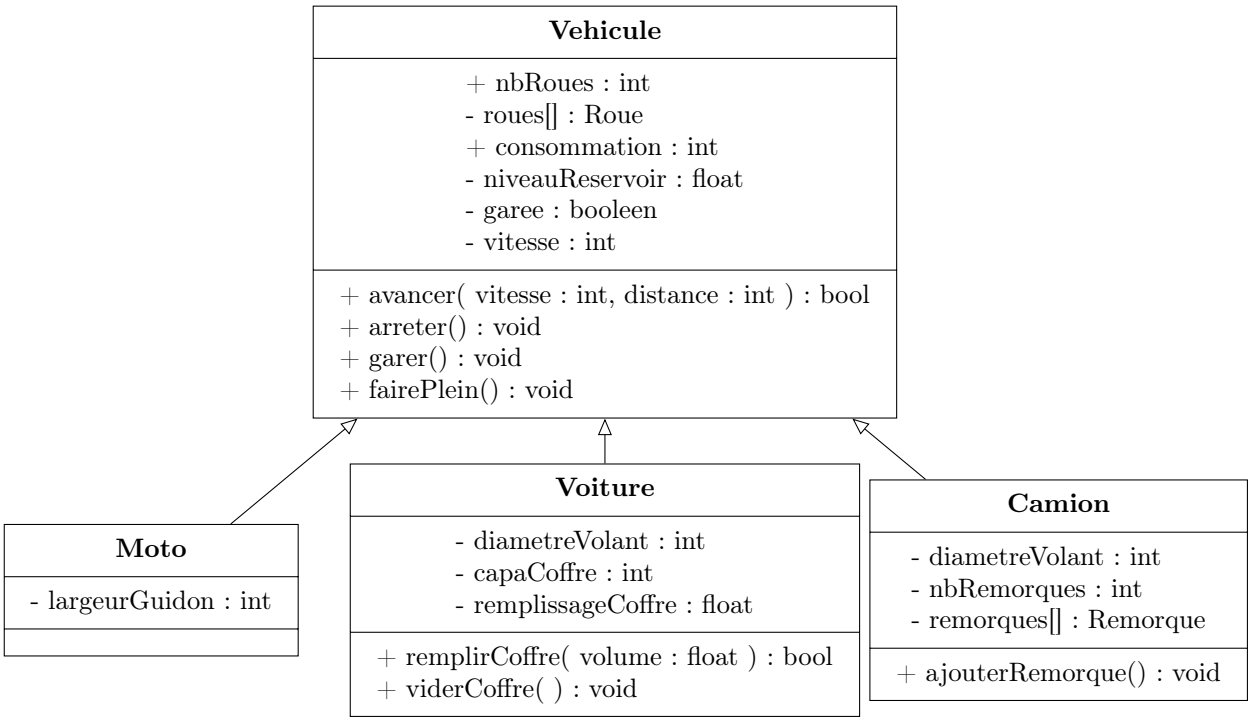
Vehicule
+ nbRoues : int - roues[] : Roue + consommation : int - niveauReservoir : float - garee : booleen - vitesse : int
+ avancer(vitesse : int, distance : int) : bool + arreter() : void + garer() : void + fairePlein() : void

Ce regroupement permet de factoriser le code de ces trois classes, au lieu de dupliquer du code identique dans trois classes différentes.

Les trois classes Moto, Camion et Voiture n'ont alors que les attributs et les méthodes qui leur sont propres : ce sont des spécialisations de la classe Vehicule.

Ces trois classes utilisent les méthodes et les attributs de la classe Vehicule : on dit qu'elles *héritent* de la classe Vehicule. Ainsi, on pourra utiliser la méthode `garer()` d'un objet de la classe Moto : celle-ci est définie dans la classe Vehicule, dont hérite la classe Moto.

On représente l'héritage sur le diagramme de classes par une flèche allant de la classe fille à la classe mère.



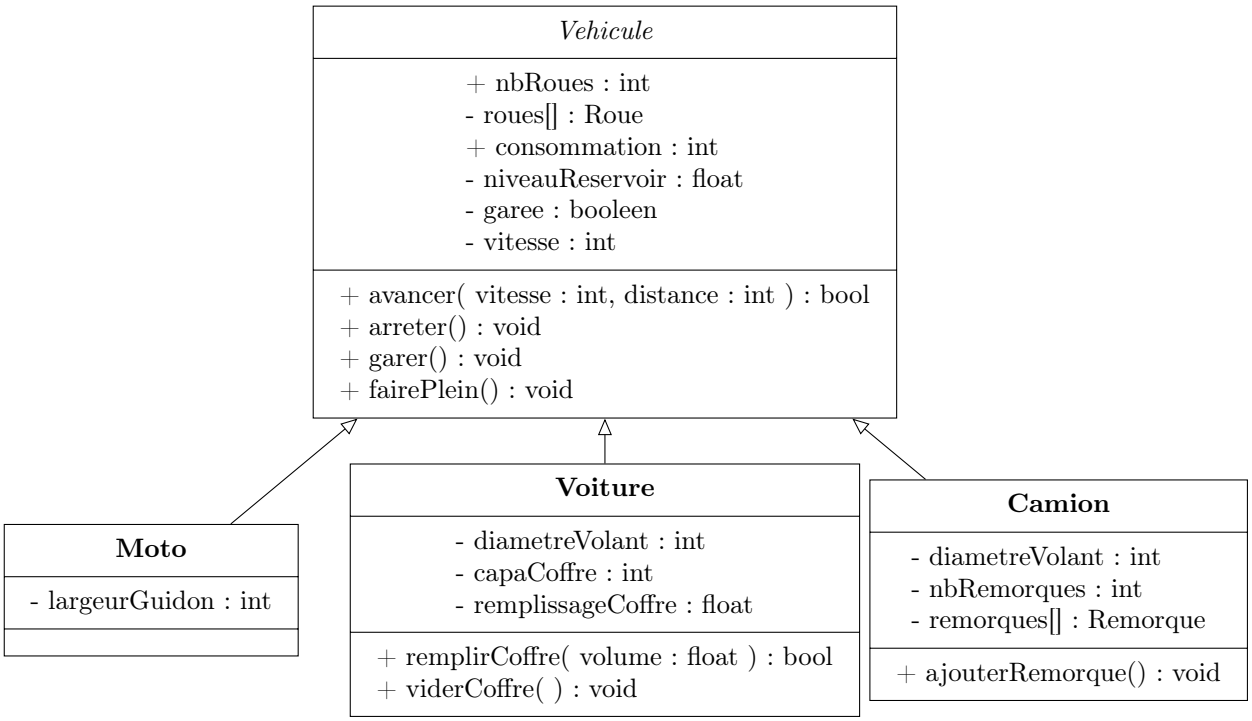
Attention : on ne peut hériter que d'une seule classe à la fois. Certains langages orientés objet prennent des libertés avec les règles de la conception orientée objet, comme C++. Il s'agit d'une particularité propre à ces langages et, de manière générale, l'héritage multiple n'est pas autorisé.

3.2 Classes abstraites

Une classe abstraite est une classe dont il n'est pas possible d'instancier des objets. Elle ne peut être utilisée que dans un arbre d'héritage.

Par exemple, la classe Vehicule définie plus haut n'a pas de sens en tant que telle : elle est utilisée par héritage par les classes Moto, Voiture et Camion, mais on ne souhaite pas créer d'objet de la classe Vehicule. Pour s'assurer que cela ne sera pas fait par erreur, on définit la classe Vehicule comme *une classe abstraite*.

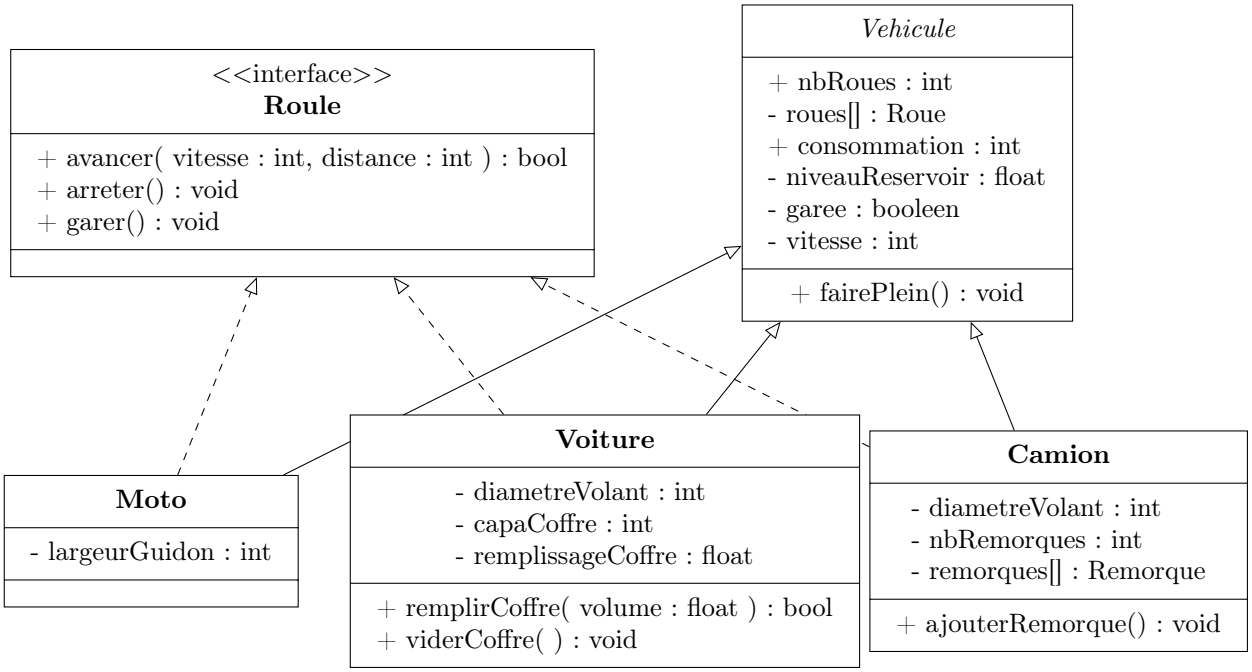
Dans le diagramme de classes, son nom apparaît en italique.



3.3 Interfaces

Une interface est un type particulier de classe qui ne définit *que des méthodes*, ne donne *que leurs prototypes* et n'est pas instanciable.

Ainsi, une interface est un contrat de comportement : les classes qui *implémentent* une interface doivent fournir des méthodes qui respectent le prototype défini dans l'interface.



4 Portée des attributs et des méthodes

Les attributs peuvent être accessibles depuis n'importe quel objet, ou seulement depuis l'objet concerné. On parle alors de *portée* des attributs. La même notion existe pour les méthodes : elles peuvent être appelées soit depuis n'importe quel objet, soit uniquement depuis l'objet concerné.

Un attribut ou une méthode qui n'est accessible que depuis l'objet dont il fait partie est *privé*. Si il est accessible depuis n'importe quel objet, il est *public*.

Une troisième possibilité existe : des attributs ou des méthodes peuvent n'être accessibles que depuis l'arbre d'héritage. Ils sont alors *protégés*.

Dans un diagramme de classes, on représente

- les attributs privés en précédant leur nom d'un signe `-`
- les attributs publics en précédant leur nom d'un signe `+`
- les attributs protégés en précédant leur nom d'un signe `#`

On utilise souvent des attributs privés afin d'éviter des modifications "involontaires", sources d'erreurs. On essaye alors le plus souvent de définir les attributs comme étant privés, et d'utiliser des méthodes publiques pour les modifier : on parle d'*accesseurs*. Par exemple, si on a un attribut `diametre` de type entier, on le définit comme étant privé et on définit deux méthodes publiques pour y accéder :

- `void setDiametre( int diametre )`
- `int getDiametre( )`