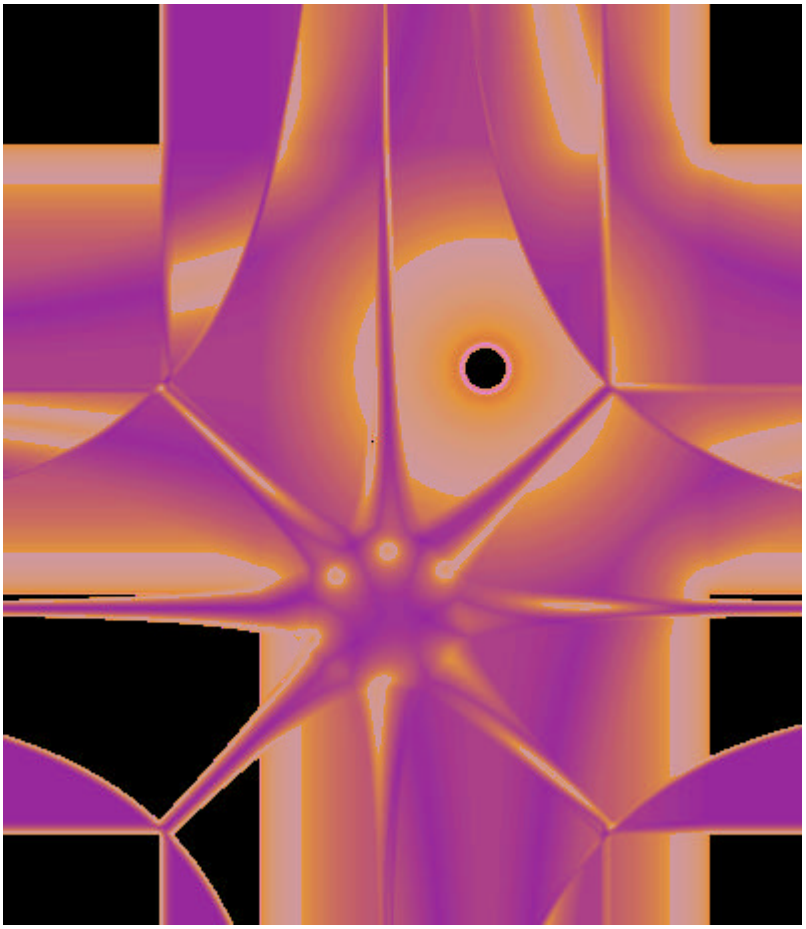
CHAPITRE 4

Constructeurs et fabriques d'objets



4.1 *Pourquoi un constructeur ou une fabrique d'objets ?*

Une classe sert de modèle à des instances, comme un type à des données. Pour qu'un objet (donc, une instance d'une classe donnée) soit réellement indépendant, il faut que l'on puisse garantir que chaque instance sera confectionnée de manière cohérente, prête à servir.

Un objet qui, après sa création nécessiterait encore des mesures d'initialisation particulières serait, au mieux, une mauvaise application de techniques procédurales. Les techniques de programmation orientées objets incluent un ou plusieurs segments de code qui permettent l'initialisation, de manière implicite, des instances. Cette initialisation sera transparente - dans la mesure du possible- à l'utilisateur de l'objet.

Cette opération garantit la fabrication d'un objet à partir d'une classe donnée de telle façon que l'objet ne nécessite aucune opération supplémentaire à son fonctionnement cohérent. L'initialisation s'effectue généralement de deux manières, selon le langage ou l'implémentation du système à objets :

- Le **constructeur** est une méthode particulière, qui ne peut pas être appelée explicitement par l'utilisateur de l'objet, mais qui s'exécute automatiquement lors de l'instanciation. Java et C++, entre autres, utilisent cette technique.
- La **fabrique d'objets** (Object Factory) crée une instance à partir d'une description d'objet (qui peut être une définition de classe). COM de Microsoft utilise cette technique. Certains cas particuliers d'application imposent le recours à cette technique particulière, même si le langage utilisé supporte d'autres constructions syntaxiques éventuellement plus simples.

Les deux techniques ne sont pas exclusives !

4.2 *Le constructeur d'objets*

Le constructeur est la manière la plus populaire d'implémenter un mécanisme d'initialisation. Il implique, à chaque création d'un nouvel objet ("d'une nouvelle instance d'une classe") l'appel implicite du code implémentant le constructeur. Selon les langages, cet appel peut ne pas être totalement implicite : Java demande un appel explicite, alors que C++, selon les cas, peut se contenter d'un appel implicite ou requérir un appel explicite.

Le constructeur ressemble, par certains côtés, à la clause d'initialisation que l'on trouve dans certains langages dits "modulaires" (p. ex. MODULA-2). Mais un module est généralement toujours initialisé de la même façon, alors qu'une instance est initialisée individuellement : c'est l'objet que l'on initialise, non la classe.

De fait, en programmation orientée objets, on a souvent deux opérations d'initialisation distinctes : l'une met en place des caractéristiques communes à toutes les instances (initialisation de la classe), l'autre spécifie les caractéristiques individuelles de l'objet.

L'exemple de code donné ci-après illustre l'utilisation d'un constructeur en C++.

```
class CompteEnBanque {
        public :
                // initialisation au niveau de la classe
        static float tauxInteret = 0.035;
        float montantInitial;
                // initialisation au niveau de l'instance
        CompteEnBanque(float depotInitial) {
                montantInitial = depotInitial;
                }
        void deposer(float montant) { ... }
        float retirer(float montant) { ... }
        float consulter(float montant) { ... }
        }

...
CompteEnBanque monCb(1000.0); // nouveau Compte en Banque
CompteEnBanque autreCb(2000.0); // autre compte
```

En Java, on aurait :

```
public class CompteEnBanque {
                // initialisation au niveau de la classe
        public static float tauxInteret = 0.035;
        public float montantInitial;
                // initialisation au niveau de l'instance
        public CompteEnBanque(float depotInitial) {
                montantInitial = depotInitial;
                }
        public void deposer(float montant) { ... }
        public float retirer(float montant) { ... }
        public float consulter(float montant) { ... }
```

```
        }

...
CompteEnBanque monCb = new CompteEnBanque(1000.0);
         // nouveau Compte en Banque
CompteEnBanque autreCb = new CompteEnBanque(2000.0);
        // autre compte
```

4.2.1 *Constructeur et héritage*

Lors d'une relation d'héritage, les choses se compliquent !

La classe dérivée est un cas particulier (une spécialisation) de la classe de base; elle doit donc, d'une certaine manière, contenir la classe de base. En toute logique, il faut donc, pour qu'une instance de la classe dérivée puisse exister, que l'instance correspondante de la classe de base ait été correctement construite.

En conséquence, il est donc nécessaire qu'une classe dérivée soit en mesure de construire une instance de sa classe de base.

En réalité, si on a correctement utilisé la notion d'héritage, ceci ne devrait poser aucun problème.

FIGURE 4.1 Classe de base et classe dérivée

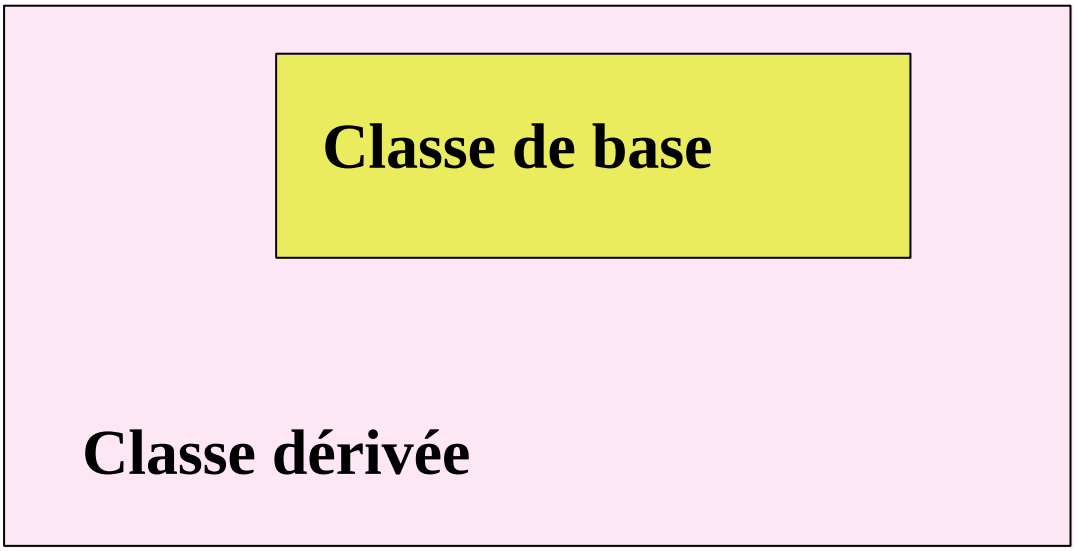
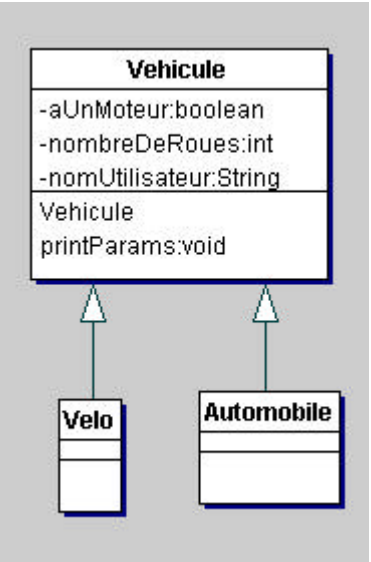


FIGURE 4.2 Exemple d'un véhicule




```
class Vehicule {
          bool aUnMoteur; // boolean aUnMoteur
          int nombreDeRoues;
          String possesseur;
          Vehicule(bool moteur, int roues, String poss) {
                    aUnMoteur = moteur;
                    nombreDeRoues = roues;
                    possesseur = poss;
                    }
          void printParams() {
          cout<<aUnMoteur<<'  ' <<nombreDeRoues<<'  ' <<possesseur<<endl;
                    }
          }

class Automobile : public Vehicule {
          Automobile(String possesseur):
                    Vehicule(true, 4, possesseur);
                    { … }
          }


class Velo : public Vehicule {
          Velo(String possesseur) :
                    Vehicule(false, 2, possesseur);
                    { … }
          }
```

Soit la séquence d'instructions :

...

```
Velo monVelo("Tartempion");
Automobile monAuto("Toto");
monAuto.printParams();
monVelo.printParams();
...
```

Qui a pour résultat :

```
true 4 Toto
false 2 Tartempion
```

Le constructeur joue un rôle extrêmement important en programmation orientée objets. Du bon choix et de la bonne implémentation des constructeurs dépend la bonne implementation de l'objet entier.

C'est l'implémentation correcte du constructeur qui permet de garantir l'autonomie de l'objet relativement à son environnement. A l'inverse, l'implémentation des constructeurs permet souvent de vérifier que la conception a été correctement faite, et que l'héritage n'a pas été abusivement utilisé.

4.2.2 *Destructeur*

Si un constructeur est nécessaire, il faut, corollairement, également définir un destructeur (implicitement ou explicitement invoqué, comme le constructeur).

Le destructeur a pour rôle de défaire tout ce qui a été fait au cours de la "vie" de l'objet.

Cette destruction consiste essentiellement en libération de mémoire reservée, mais aussi en la finalisation de ressources, comme des fichiers, des connexions TCP-IP, des séances MIDI, etc... De fait, il s'agit de restituer l'état de l'environnement d'exécution comme il l'était avant l'existence de l'objet.

La majorité des langages «OO» définit un constructeur - et également un destructeur - par défaut. Il en va ainsi de C++ et de Java. Il faut se méfier de ces mécanismes, car le constructeur par défaut ne fait que rarement le travail que l 'on aurait effectivement souhaité de la part d'un constructeur.

Pour un langage purement procédural, le constructeur n'a pas de raison d'être (sinon pour effectuer la réservation de mémoire) en raison de l'absence de tout lien entre les données et le code censé manipuler ces données.

4.2.3 *La notion de classe non instantiable (ou abstraite)*

Une classe peut ne pas être instanciable. Ainsi, dans la hiérarchie de la figure4.2, page45, il ne serait pas raisonnable d'instancier un objet de la classe Vénicule. Il s'agit d 'un objet générique, qui ne peut en aucun cas correspondre à un objet réel.

Cette classe est alors dite abstraite.

```
// Définitions de classe (incomplètes)
// La syntaxe ne correspond pas exactement à du C++ !
// Le mot abstract n 'existe pas dans ce langage, bien
// que la notion de classe abstraite existe, elle !



abstract class Animal {}
abstract class Mammifere : public Animal {}
abstract class Carnivore : public Mammifere {}
class Chat : public Carnivore {}


// Utilisation de ces classes :
Animal dahu; // Erreur à la compilation !
               // Animal est une classe abstraite, qui ne
               // peut donc être instanciée
Chat tom;    // Ok, Chat est une classe non abstraite
```

Une classe abstraite possède également un constructeur et un destructeur. Bien que l 'on ne puisse pas créer une instance d'une classe abstraite, cette classe possède néanmoins des propriétés et des méthodes qu 'une classe dérivée va utiliser. Ces propriétés doivent bien évidemment être correctement initialisées.

Bien que l'on ne puisse pas créer d'instance d'une classe abstraite, il est néanmoins possible de manipuler une telle instance : on manipulera alors une instance crée par une sous-classe.

```
void  faireMangerUnAnimal(Animal cetAnimal) {
         cetAnimal.manger(); // On manipule une
                                    // instance d 'Animal
         }
…
Chat monChat;
         // Nous savons qu 'un Chat EST UN Animal
         // en raison de l 'héritage.
faireMangerUnAnimal(monChat);
```

Une classe abstraite peut (mais ne doit pas) comporter des méthodes abstraites. Une telle méthode ne comporte pas d'implémentation. Ainsi, la méthode manger() de la classe Animal est certainement abstraite, puisque l'on ne sait pas de quel animal il s'agit, et qu'il serait donc difficile de décrire sa manière de s 'alimenter. Une classe ayant une méthode abstraite est forcément abstraite elle aussi.

Une méthode abstraite peut en revanche être invoquée : l'implémentation sera réalisée par la sous-classe ayant permis l'instanciation. Dans l'exemple précédent, c'est le chat qui va «manger».

Attention à ne pas confondre « classe abstraite» et type abstrait, tel que défini par certains langages (Ada, en particulier). Le type abstrait permet la définition générique de types

concrets, alors que la classe abstraite regroupe des propriétés et des méthodes que les classes dérivées (sous-classes) peuvent soit redéfinir, soit affiner.

Les classes abstraites sont très utilisées en conjonction avec la notion de polymorphisme, surtout dans les langages réellement orientés objets, comme Java.

4.3 *Fabriques d'objets*

Un constructeur est une manière assez confortable et rapide permettant de mettre sur pied une instance d'une classe déterminée. Dans la majorité des cas courants, ce mécanisme suffit et est tout à fait approprié. Pourtant, il existe des cas où cette technique n'est que difficilement, ou pas du tout, applicable; citons quelques exemples :

- Nécessité de créer une instance d'une classe que l'on ne connaît pas au moment de la compilation, mais uniquement au moment de l'exécution. Dans ce cas, un constructeur n'existe pas, ou s'il existe, il est hors de portée du développeur. Ceci est en particulier le cas lors de l'utilisation de librairies normalisées, comme JDBC ou JTAPI, pour lesquelles il existe plusieurs implémentations (une pour SQL Server, une pour Oracle, une pour MySQL, etc...), mais sans que l'on sache laquelle de ces implémentations sera finalement utilisée.
- Faire de l'orienté objets avec un langage non prévu pour, comme C.
- Interfacer un concept orienté objets avec des mécanismes non orientés objets (par exemple, interfacer Java à une base de données relationnelle).
- Construction d'objets nécessitant des précautions spéciales, comme la préparation de ressources particulières lors de la création de la première instance, ou le contrôle du nombre d'instances crées.