# Chapitre 6
# La modélisation des objets

## 1. Introduction

L'objectif de ce chapitre est de vous faire découvrir les techniques UML de modélisation statique des objets.

Cette modélisation est statique car elle ne décrit pas les interactions ou le cycle de vie des objets. Les méthodes sont introduites d'un point de vue statique, sans description de leur enchaînement.

Nous découvrirons le diagramme de classes. Ce diagramme contient les attributs, les méthodes et les associations des objets. Comme nous l'avons vu au chapitre Les concepts de l'approche par objets, cette description est réalisée par les classes.

Ce diagramme est central lors d'une modélisation par objets d'un système. De tous les diagrammes UML, il est le seul obligatoire lors d'une telle modélisation.

Nous étudierons comment le langage OCL (*Object Constraint Language* ou langage de contraintes objet) peut étendre le diagramme de classes pour exprimer de façon plus riche les contraintes. Ensuite, le diagramme d'objets nous montrera comment illustrer la modélisation réalisée dans le diagramme de classes. Enfin, nous découvrirons comment décrire les objets composés au moyen du diagramme de structure composite.

L'emploi d'OCL, du diagramme d'objets ou du diagramme de structure composite est optionnel. Leur utilisation dépend des contraintes du projet de modélisation.

## 2. Découvrir les objets du système par décomposition

Au chapitre La modélisation de la dynamique, nous avons étudié comment découvrir les objets d'un point de vue dynamique. Nous avons présenté les cas d'utilisation sous forme de diagrammes de séquence. Puis nous avons enrichi ces diagrammes au niveau de l'envoi de message pour découvrir les objets du système.

La décomposition des messages fait apparaître les objets du système, car elle conduit à des messages plus fins dont il convient de rechercher le destinataire.

Une autre approche possible est la décomposition de l'information contenue dans un objet. Souvent, cette information est trop complexe pour n'être représentée que par la structure d'un seul objet. Elle doit parfois être également répartie entre plusieurs objets.

Exemple

Dans l'exemple du chapitre La modélisation de la dynamique, le directeur recherche les papiers (dans le sens d'informations) de la jument à vendre dans la base de données de l'élevage. Cette base constitue un objet à grosse granularité composé lui-même d'autres objets, comme les papiers des chevaux, les informations financières et comptables, les documents d'achat et de vente de chevaux. Les papiers d'une jument sont composés, entre autres, de son carnet de vaccination et des papiers de ses descendants. Les papiers des descendants sont partagés par d'autres objets, comme les papiers de leur père étalon. Cette décomposition est guidée par les données et non par des aspects dynamiques. La figure 6.1 illustre la composition de `PapiersJument` dans le diagramme de classes.

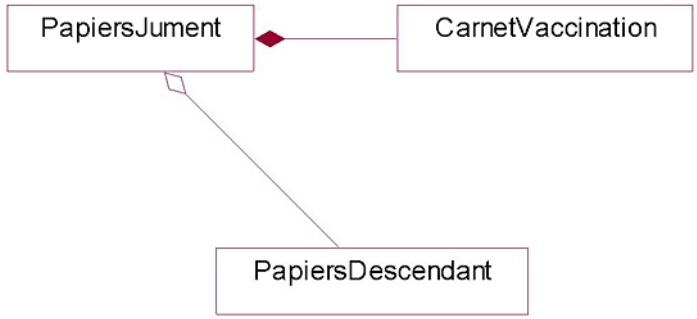


*Figure 6.1 - Composition de* `PapiersJument`

Remarque

*Rappelons que la granularité d'un objet définit sa taille. Le système pris comme un objet est de gros grain ou de granularité importante. À l'opposé, le carnet de vaccination d'un cheval est un objet de grain beaucoup plus fin que le système.*

Exemple

La décomposition d'un cheval pour faire apparaître ses différents organes peut se faire soit par la décomposition d'un diagramme de séquence, soit par la décomposition guidée par les données.

La décomposition par le diagramme de séquence consiste à analyser différents envois de message : faire peur, courir, manger, dormir. Ces derniers feront apparaître progressivement les différents organes du cheval. Elle est illustrée à la figure 6.2 pour le message `fairePeur`. Un cheval dilate ses naseaux pour marquer l'alerte, la surprise ou la peur. Il pince la bouche pour indiquer tension, peur ou colère. Enfin, la ruade est un mouvement défensif.

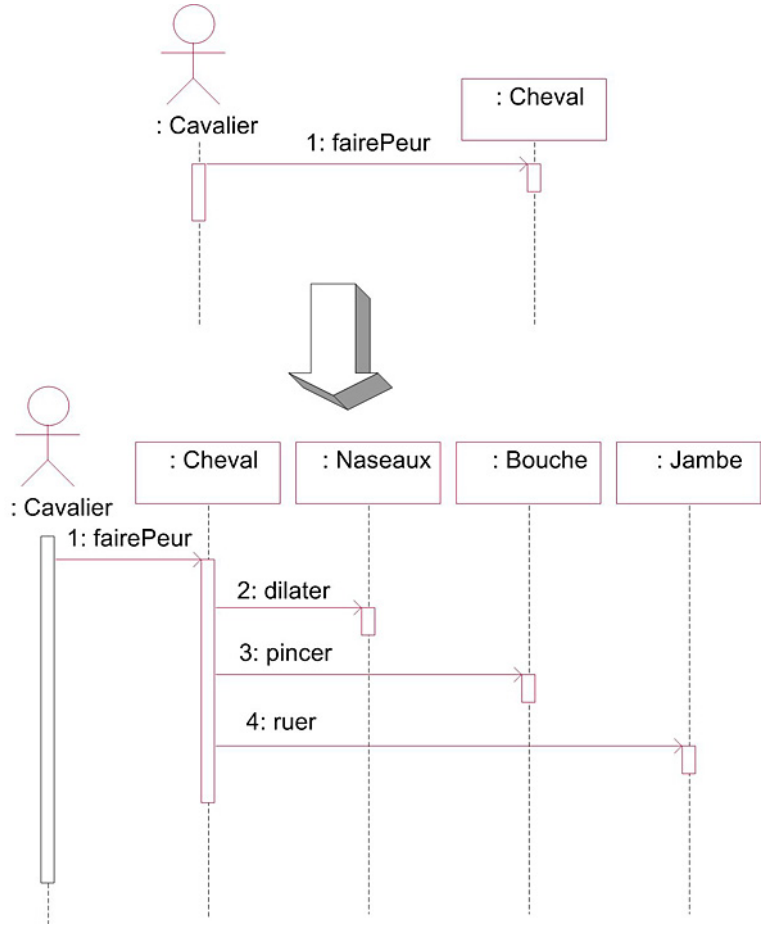


*Figure 6.2 - Découverte des objets par enrichissement du diagramme de séquence*

La décomposition guidée par les données consiste à étudier directement les différents organes d'un cheval et à les prendre en compte dans le diagramme de classes. La figure 6.3 illustre un cheval composé de ses différents organes.

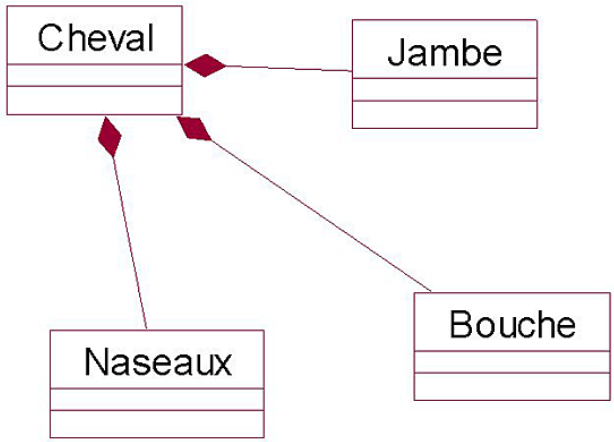


*Figure 6.3 - Composition de* `Cheval`

Remarque

*La décomposition par le diagramme de séquence et la décomposition guidée par les données apparaissent naturelles pour découvrir les objets, ce qui est normal car un objet est l'assemblage d'une structure et d'un comportement. Il convient enfin de remarquer que ces deux approches ne sont pas incompatibles.*

Remarque

*La décomposition guidée par les données est plus efficace quand la personne chargée de la modélisation connaît bien le domaine. La décomposition en objets est alors immédiate.*

# 3. La représentation des classes

## 3.1 La forme simplifiée de représentation des classes

Les objets du système sont décrits par des classes dont une forme simplifiée de la représentation en UML est donnée à la figure 6.4. Cette représentation est constituée de trois parties.

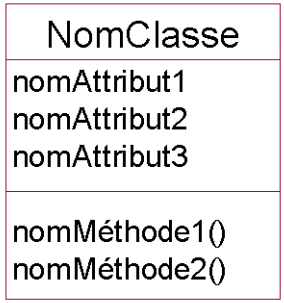


*Figure 6.4 - Représentation simplifiée d'une classe en UML*

La première partie contient le nom de la classe.

Remarque

*Rappelons que le nom d'une classe est au singulier. Il est constitué d'un nom commun précédé ou suivi d'un ou plusieurs adjectifs qualifiant le nom. Ce nom est significatif de l'ensemble des objets constituant la classe. Il représente la nature des instances d'une classe.*

La deuxième partie contient les attributs. Ceux-ci contiennent l'information portée par un objet. L'ensemble des attributs forme la structure de l'objet.

La troisième partie contient les méthodes. Celles-ci correspondent aux services offerts par l'objet. Elles peuvent modifier la valeur des attributs. L'ensemble des méthodes forme le comportement de l'objet.



# Chapitre 4
# Le design pattern Abstract Factory

## 1. Description

Le but du design pattern `Abstract Factory` est la création d'objets regroupés en familles sans avoir à connaître leurs classes concrètes.

## 2. Exemple

Le système de vente de véhicules gère des véhicules fonctionnant à l'essence et des véhicules fonctionnant à l'électricité. Cette gestion est confiée à l'objet `Catalogue`, à qui incombe la responsabilité de créer de tels objets.

Pour chaque produit, nous disposons d'une classe abstraite, d'une sous-classe concrète décrivant la version du produit fonctionnant à l'essence et d'une sous-classe concrète décrivant la version du produit fonctionnant à l'électricité. Par exemple, à la figure 4.1, pour l'objet scooter, il existe une classe abstraite `Scooter` et deux sous-classes concrètes : `ScooterElectricite` et `ScooterEssence`.

L'objet `Catalogue` peut utiliser ces sous-classes concrètes pour instancier les produits. Cependant, si par la suite de nouvelles familles de véhicules doivent être prises en compte (diesel ou hybride essence-électricité), les modifications à apporter à l'objet `Catalogue` peuvent s'avérer assez fastidieuses.

Le design pattern `Abstract Factory` résout ce problème en introduisant une interface `FabriqueVehiculeInterface` qui contient la signature des méthodes à utiliser pour créer chaque produit. Le type de retour de ces méthodes est constitué par l'une des classes abstraites de produit. Ainsi, l'objet `Catalogue` n'a pas besoin de connaître les sous-classes concrètes et reste parfaitement indépendant des familles de produits. En révélant l'interface de nos fabriques et non leur implémentation, nous découplons le code client des produits concrets : notre objet client `Catalogue` demande des produits à la fabrique qu'on lui passe en paramètre lors de sa construction sans avoir la moindre idée de qui elle est ni de ce qui se passe en coulisses pour qu'il obtienne le bon produit.

Une classe implémentant `FabriqueVehiculeInterface` est créée pour chaque famille de produits, à savoir les classes `FabriqueVehiculeElectricite` et `FabriqueVehiculeEssence`. Une telle classe a la responsabilité d'implémenter les opérations de création du véhicule appropriée pour la famille à laquelle elle est associée.

L'objet client `Catalogue` prend alors pour paramètre un objet se conformant à l'interface `FabriqueVehiculeInterface`, c'est-à-dire soit une instance de `FabriqueVehiculeElectricite`, soit une instance de `FabriqueVehiculeEssence`. Avec une telle instance, le catalogue peut créer et manipuler des véhicules sans devoir connaître les familles de véhicules et les classes concrètes correspondantes.

L'ensemble des classes du design pattern `Abstract Factory` pour cet exemple est détaillé à la figure 4.1.

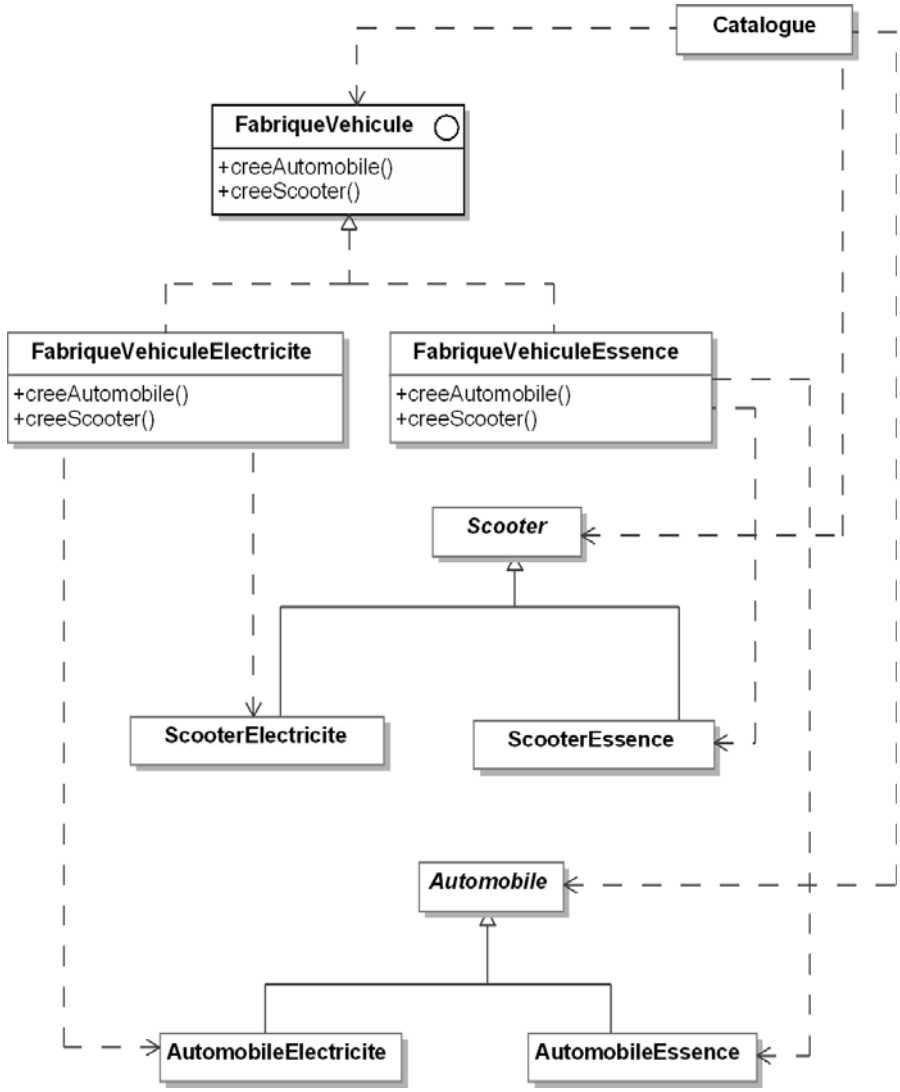


*Figure 4.1 - Le design pattern `Abstract Factory` appliqué à des familles de véhicules*

# 3. Structure

## 3.1 Diagramme de classes

La figure 4.2 détaille la structure générique du design pattern `Abstract Factory`.

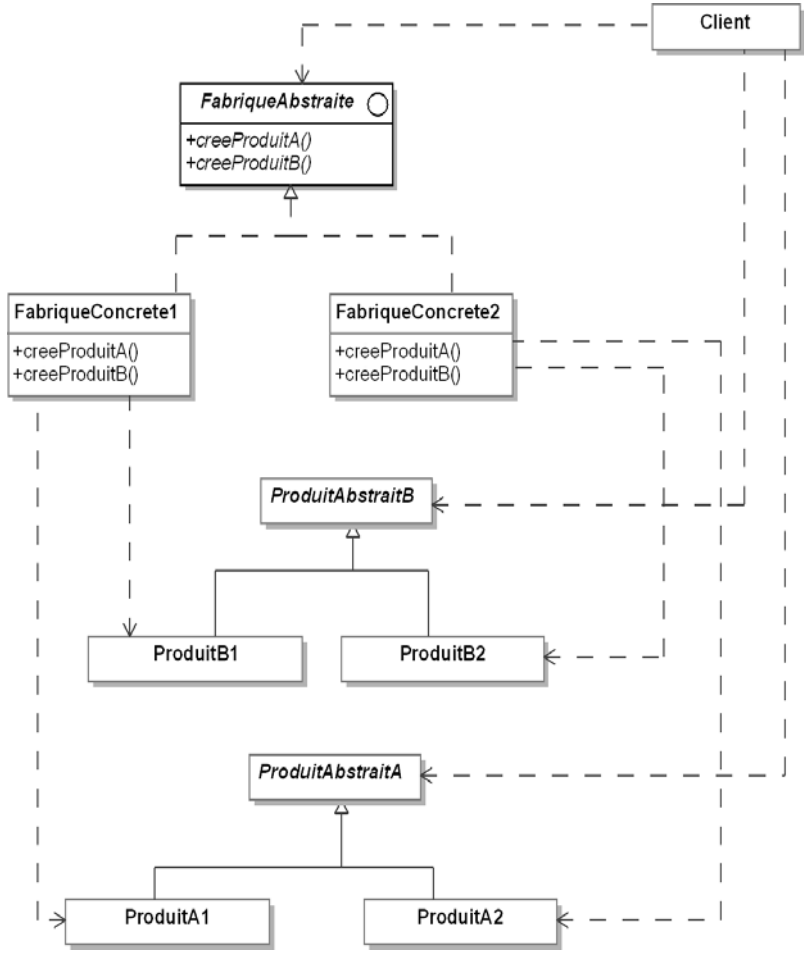


*Figure 4.2 - Structure du design pattern* `Abstract Factory`

## 3.2 Participants

Les participants au design pattern `Abstract Factory` sont les suivants :

- `FabriqueAbstraite` (`FabriqueVehiculeInterface`) est une interface spécifiant les signatures des méthodes créant les différents produits.
- `FabriqueConcrete1`, `FabriqueConcrete2` (`FabriqueVehicule-Electricite`, `FabriqueVehiculeEssence`) sont les classes concrètes implémentant les méthodes créant les produits pour chaque famille de produits. Connaissant la famille et le produit, elles sont capables de créer une instance du produit pour cette famille.
- `ProduitAbstraitA` et `ProduitAbstraitB` (`AbstractScooter` et `AbstractAutomobile`) sont les classes abstraites des produits indépendamment de leur famille. Les familles sont introduites dans leurs sous-classes concrètes.
- `Client` (`Catalogue`) est la classe qui utilise l'interface `FabriqueAbstraite`.

## 3.3 Collaborations

La classe `Catalogue` utilise une instance de l'une des fabriques concrètes pour créer ses produits au travers de l'interface exposée par `FabriqueAbstraiteInterface`.

**Remarque**

*Il est recommandé de ne créer qu'une seule instance des fabriques concrètes, celle-ci pouvant être partagée par plusieurs clients. Nous verrons plus tard un design pattern capable de garantir qu'une seule instance d'une classe est disponible à l'exécution : `Singleton`.*

## 4. Domaines d'utilisation

Le design pattern `Abstract Factory` est utilisé dans les domaines suivants :

– Un système utilisant des produits a besoin d'être indépendant de la façon dont ces produits sont créés et regroupés.

– Un système est paramétré par plusieurs familles de produits qui peuvent évoluer.

## 5. Exemple en PHP

Voici maintenant un exemple d'utilisation du design pattern écrit en PHP. Le code PHP correspondant à la classe abstraite `AbstractAutomobile` et ses sous-classes est donné à la suite. Il est très simple, il décrit les quatre propriétés des automobiles ainsi que la méthode `afficheCaracteristiques` qui permet de les afficher.

```
<?php

declare(strict_types=1);

namespace ENI\DesignPatterns\AbstractFactory;

abstract class AbstractAutomobile
{
   protected string $marque;

   protected string $couleur;

   protected int $puissance;

   protected float $espace;

   public function __construct(string $marque, string $couleur,
int $puissance, float $espace)
   {
       $this->marque = $marque;
       $this->couleur = $couleur;
       $this->puissance = $puissance;
```

```
        $this->espace = $espace;
    }

    abstract public function afficheCaracteristiques(): void;
}


<?php

declare(strict_types=1);

namespace ENI\DesignPatterns\AbstractFactory;

class AutomobileElectricite extends AbstractAutomobile
{
    public function afficheCaracteristiques(): void
    {
        echo "Automobile électrique - marque: $this->marque"
            . ", couleur: $this->couleur"
            . ", puissance: $this->puissance"
            . ", espace: $this->espace" . PHP_EOL;
    }
}


<?php

declare(strict_types=1);

namespace ENI\DesignPatterns\AbstractFactory;

class AutomobileEssence extends AbstractAutomobile
{
    public function afficheCaracteristiques(): void
    {
        echo "Automobile à essence - marque: $this->marque"
            . ", couleur: $this->couleur"
            . ", puissance: $this->puissance"
            . ", espace: $this->espace" . PHP_EOL;
    }
}
```

Le code PHP correspondant à la classe abstraite `AbstractScooter` et ses sous-classes est donné à la suite. Il est similaire à celui des automobiles, à