INFORMATIQUE
TECHNIQUE

Expert

Design Patterns en PHP

Les 23 modèles de conception : descriptions et solutions illustrées en UML2 et PHP

Téléchargement
www.editions-eni.fr

Laurent DEBRAUWER
Yannick EVAIN

eni Editions

Les éléments à télécharger sont disponibles à l'adresse suivante :
http://www.editions-eni.fr
Saisissez la référence ENI de l'ouvrage **EIPHDES** dans la zone de recherche et validez. Cliquez sur le titre du livre puis sur le bouton de téléchargement.

Partie 2 : Patterns de construction

Chapitre 3
Introduction aux patterns de construction

Chapitre 4
Le pattern Abstract Factory

Chapitre 5
Le pattern Builder

Partie 3 : Patterns de structuration

Chapitre 9
Introduction aux patterns de structuration

Chapitre 10
Le pattern Adapter

Chapitre 11
Le pattern Bridge

Chapitre 12
Le pattern Composite

Chapitre 13
Le pattern Decorator

Chapitre 14
Le pattern Facade

Chapitre 15
Le pattern Flyweight

Chapitre 19
Le pattern Command

Chapitre 20
Le pattern Interpreter

Chapitre 21
Le pattern Iterator

Chapitre 22
Le pattern Mediator

Chapitre 23
Le pattern Memento

Chapitre 24
Le pattern Observer

Chapitre 25
Le pattern State

Chapitre 26
Le pattern Strategy

Chapitre 27
Le pattern Template Method

Chapitre 28
Le pattern Visitor

Partie 5 : Application des patterns

Chapitre 29
Compositions et variations de patterns

Chapitre 30
Le pattern composite MVC

Chapitre 31
Les patterns dans la conception de logiciels

Annexe
Exercices

Chapitre 4
Le pattern Abstract Factory

1. Description

Le but du pattern `Abstract Factory` est la création d'objets regroupés en familles sans devoir connaître les classes concrètes destinées à la création de ces objets.

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` qui crée 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 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 `ScooterÉlectricité` 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 par la suite (diesel ou mixte essence-électricité), les modifications à apporter à l'objet `Catalogue` peuvent être assez lourdes.

Le pattern `Abstract Factory` résout ce problème en introduisant une interface `FabriqueVéhicule` qui contient la signature des méthodes pour définir 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 indépendant des familles de produit.

Une sous-classe d'implantation de `FabriqueVéhicule` est introduite pour chaque famille de produit, à savoir les sous-classes `FabriqueVéhicule-Électricité` et `FabriqueVéhiculeEssence`. Une telle sous-classe implante les opérations de création du véhicule appropriée pour la famille à laquelle elle est associée.

L'objet `Catalogue` prend alors pour paramètre une instance répondant à l'interface `FabriqueVéhicule`, c'est-à-dire soit une instance de `FabriqueVéhiculeÉlectricité`, soit une instance de `Fabrique-VéhiculeEssence`. Avec une telle instance, le catalogue peut créer et manipuler des véhicules sans devoir connaître les familles de véhicules et les classes concrètes d'instanciation correspondantes.

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

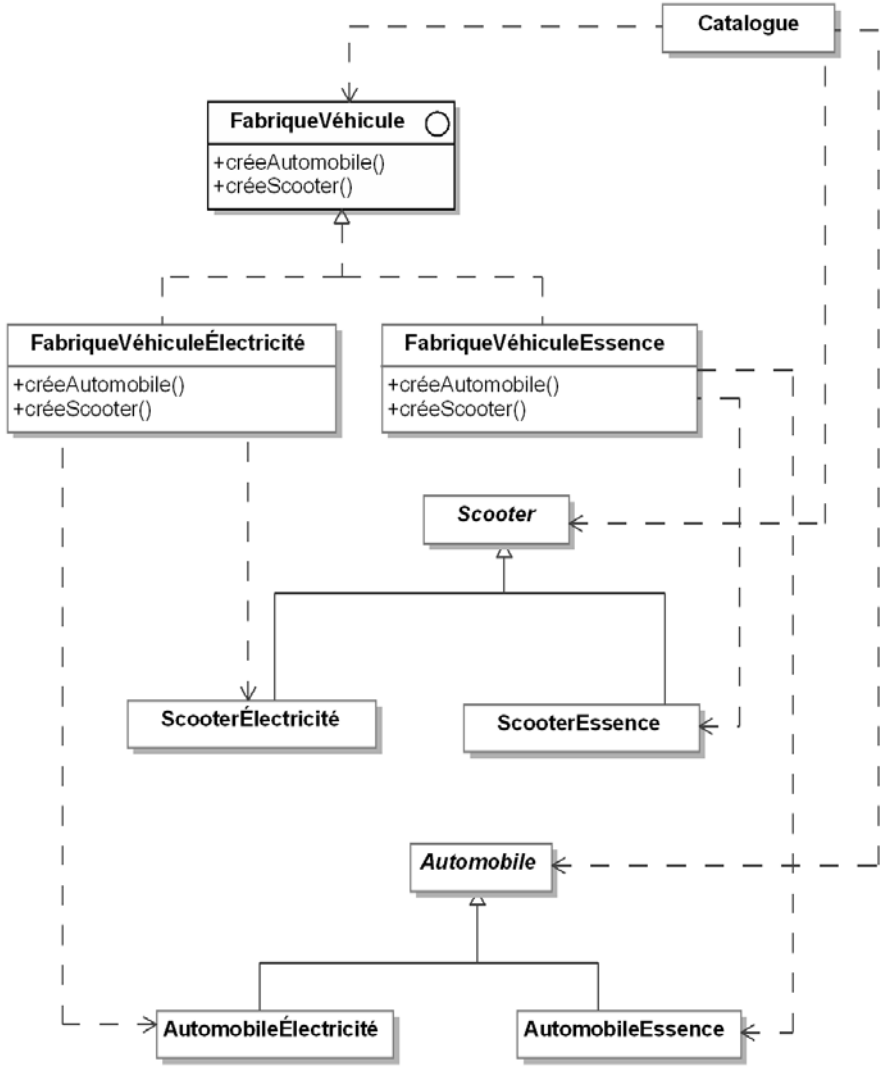

Figure 4.1 - Le pattern `Abstract Factory` appliqué à des familles de véhicules

3. Structure

3.1 Diagramme de classes

La figure 4.2 détaille la structure générique du pattern.

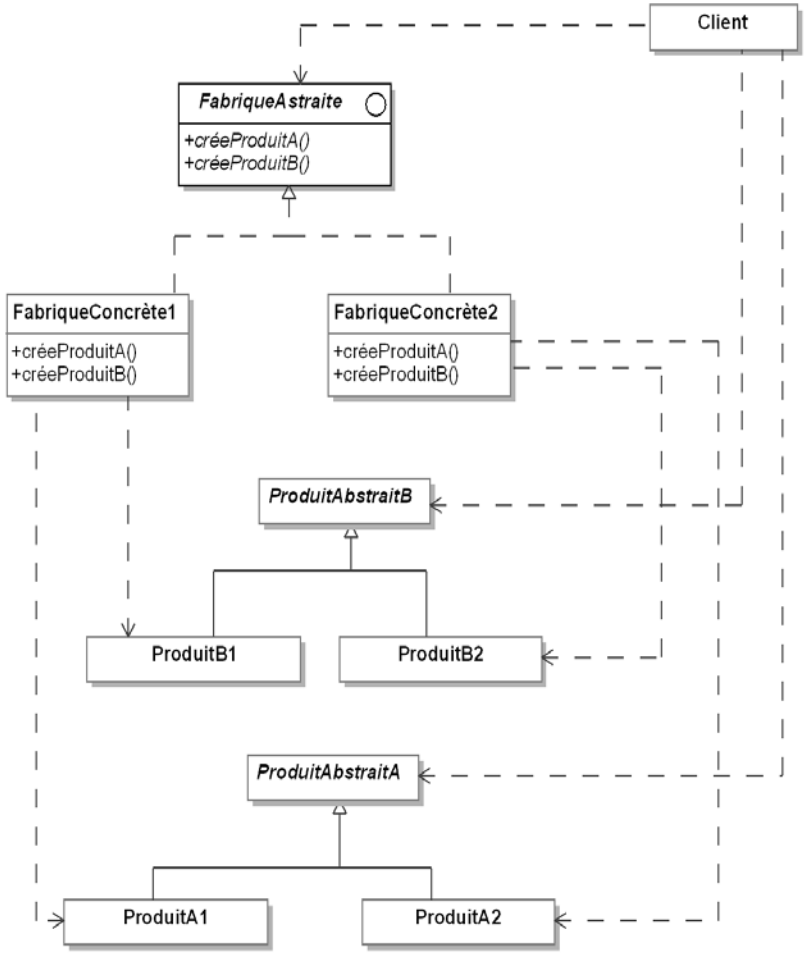

Figure 4.2 - Structure du pattern `Abstract Factory`

3.2 Participants

Les participants au pattern sont les suivants :

– `FabriqueAbstraite` (`FabriqueVéhicule`) est une interface spécifiant les signatures des méthodes créant les différents produits.

– `FabriqueConcrète1`, `FabriqueConcrète2` (`FabriqueVéhicule-Électricité`, `FabriqueVéhiculeEssence`) sont les classes concrètes implantant 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` (`Scooter` et `Automobile`) sont les classes abstraites des produits indépendamment de leur famille. Les familles sont introduites dans leurs sous-classes concrètes.

– `Client` est la classe qui utilise l'interface de `FabriqueAbstraite`.

3.3 Collaborations

La classe `Client` utilise une instance de l'une des fabriques concrètes pour créer ses produits au travers de l'interface de `FabriqueAbstraite`.

Remarque

Normalement, il ne faut créer qu'une seule instance des fabriques concrètes, celle-ci pouvant être partagée par plusieurs clients.

4. Domaines d'utilisation

Le pattern 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

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

```
<?php
namespace AbstractFactory;

abstract class Automobile
{
    /**
     *
     * @var string
     */
    protected $modele;
    /**
     *
     * @var string
     */
    protected $couleur;
    /**
     *
     * @var int
     */
    protected $puissance;
    /**
     *
     * @var double
     */
    protected $espace;

    /**
     *
     * @param string $modele
     * @param string $couleur
     * @param int $puissance
     * @param double $espace
     */
    public function __construct($modele, $couleur, $puissance,
            $espace)
    {
```