Collection EXPERT IT

Design Patterns pour Java

Les 23 modèles de conception : descriptions et solutions illustrées en UML 2 et Java

Seconde Edition

Laurent DEBRAUWER

Fichiers à télécharger
www.editions-eni.fr

eni Editions

Design Patterns pour Java

Les 23 modèles de conception [2ième édition]

Laurent DEBRAUWER

Résumé

Ce livre sur UML 2 présente **de façon concise et pratique** les 23 modèles de conception (design patterns) fondamentaux en les illustrant par des **exemples pertinents et rapides à appréhender**. Chaque exemple est décrit en UML et en Java sous la forme d´un petit programme complet et exécutable. Pour chaque pattern, l´auteur détaille son nom, **le problème à résoudre**, la solution qu´il apporte, ses **domaines d´application** et sa **structure** générique.
Le livre s´adresse aux **concepteurs et développeurs en Programmation Orientée Objet**. Pour bien l´appréhender, il est préférable de disposer de connaissances sur les principaux éléments des diagrammes de classes UML et sur la dernière version du langage Java.
Le livre est organisé en trois parties qui correspondent aux trois familles des patterns de conception : les **patterns de construction**, les **patterns de structuration** et les **patterns de comportement**.
Les exemples utilisés dans ces parties sont issus d´une application de vente en ligne de véhicules et sont en téléchargement sur cette page.

L'auteur

Laurent Debrauwer est docteur en informatique de l'Université de Lille 1. Auteur de logiciels dans le domaine de la linguistique et de la sémantique, il exerce le métier de consultant indépendant en tant que spécialiste de l'approche par objets. Il enseigne l'ingénierie logicielle et la programmation en Java à l'université du Luxembourg ainsi que la modélisation en UML à l'université de Lille 1.

Design Patterns ou patterns de conception

Un design pattern ou pattern de conception consiste en un schéma à objets qui forme une solution à un problème connu et fréquent. Le schéma à objets est constitué par un ensemble d'objets décrits par des classes et des relations liant les objets.

Les patterns répondent à des problèmes de conception de logiciels dans le cadre de la programmation par objets. Ce sont des solutions connues et éprouvées dont la conception provient de l'expérience de programmeurs. Il n'y a pas d'aspect théorique dans les patterns, notamment pas de formalisation (à la différence des algorithmes).

Les patterns sont basés sur les bonnes pratiques de la programmation par objets. Par exemple, la figure 1.1 donne l'exemple du pattern `Template method` qui sera décrit au chapitre Le pattern Template Method. Dans ce pattern, la méthode `calculeMontantTtc` invoque la méthode `calculeTva` qui est abstraite dans la classe `Commande`. Elle est définie dans les sous-classes de `Commande` à savoir les classes `CommandeFrance` et `CommandeLuxembourg`. En effet, le taux de TVA varie en fonction du pays. La méthode `calculeMontantTtc` est appelée méthode "patron" (`Template method`). Elle introduit un algorithme basé sur une méthode abstraite.

Ce pattern est basé sur le polymorphisme, une propriété importante de la programmation par objets. Le montant d'une commande en France ou au Luxembourg est soumis à la TVA. Mais le taux n'est pas le même, le calcul de TVA est donc différent en France et au Luxembourg. Par conséquent, le pattern `Template method` constitue une bonne illustration du polymorphisme.

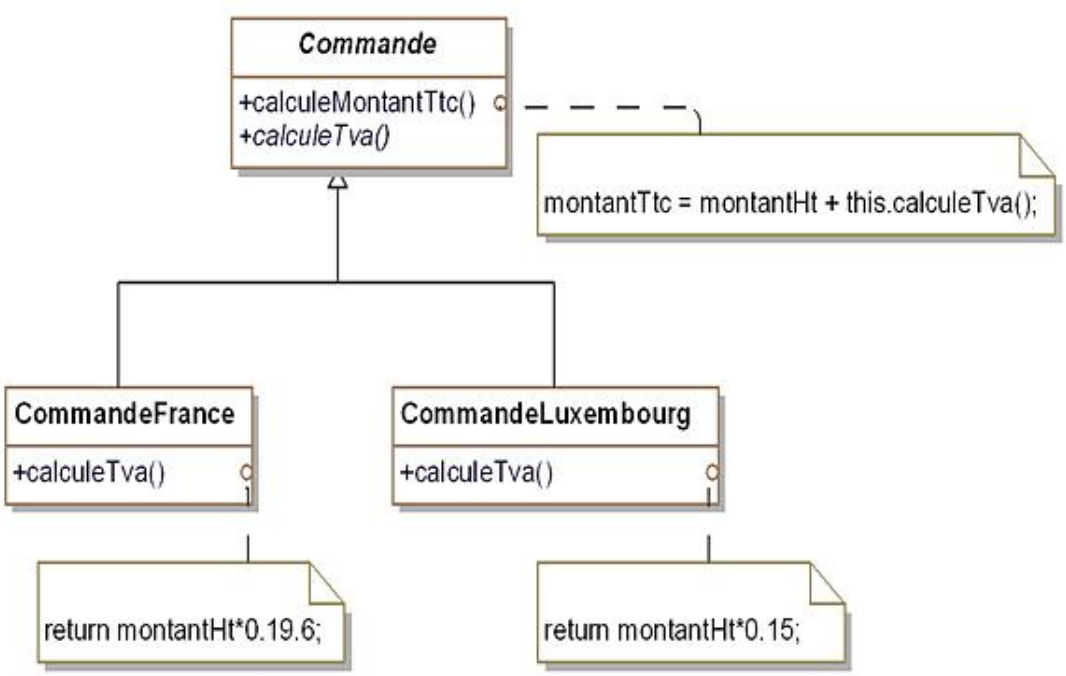


Figure 1.1 - Exemple d'utilisation du pattern Template Method

Les patterns ont été introduits en 1995 dans le livre dit "GoF" pour Gang of Four (qui sont les quatre auteurs) intitulé "Design Patterns - Elements of Reusable Object-Oriented Software" et écrit par Erich Gamma, Richard Helm, Ralph Johnson et John Vlissides. Ce livre constitue l'ouvrage de référence des patterns de conception.

La description des patterns de conception

Nous avons fait le choix de décrire les patterns de conception à l'aide des deux langages suivants :

- le langage de modélisation UML introduit par l'OMG (http://www.omg.org) ;
- le langage de programmation Java créé par la société "Sun Microsystems" (http://www.java.com).

Les patterns de conception sont introduits dans les parties 2 à 4. Pour chaque pattern, les éléments suivants sont présentés :

- le nom du pattern ;
- la description du pattern ;
- un exemple décrivant le problème et la solution basée sur le pattern décrit sous la forme d'un diagramme de classes UML. Au sein de ce schéma, le corps des méthodes est décrit à l'aide de notes ;
- la structure générique du pattern, à savoir :
  - son schéma en dehors de tout contexte particulier sous la forme d'un diagramme de classes UML ;
  - la liste des participants au pattern ;
  - les collaborations au sein du pattern ;
- les domaines d'application du pattern ;
- l'exemple présenté cette fois sous la forme d'un programme Java complet et documenté. Ce programme n'utilise pas d'interface graphique mais exclusivement les entrées/sorties de l'écran et du clavier.

Le catalogue des patterns de conception

Dans ce livre nous présentons les vingt-trois patterns de conception introduits dans l'ouvrage de référence "GoF". Ces patterns sont autant de réponses différentes à des problèmes connus de la programmation par objets. La liste qui suit n'est pas exhaustive, elle provient comme nous l'avons déjà expliqué de l'expérience.

- **Abstract Factory** : a pour objectif la création d'objets regroupés en familles sans devoir connaître les classes concrètes destinées à la création de ces objets ;
- **Builder** : permet de séparer la construction d'objets complexes de leur implantation de sorte qu'un client puisse créer ces objets complexes avec des implantations différentes ;
- **Factory Method** : a pour but d'introduire une méthode abstraite de création d'un objet en reportant aux sous-classes concrètes la création effective ;
- **Prototype** : permet la création de nouveaux objets par duplication d'objets existants appelés prototypes qui disposent de la capacité de clonage ;
- **Singleton** : permet de s'assurer qu'une classe ne possède qu'une seule instance et de fournir une méthode unique retournant cette instance ;
- **Adapter** : a pour but de convertir l'interface d'une classe existante en l'interface attendue par des clients également existants afin qu'ils puissent travailler ensemble ;
- **Bridge** : a pour but de séparer les aspects conceptuels d'une hiérarchie de classes de leur implantation ;
- **Composite** : offre un cadre de conception d'une composition d'objets dont la profondeur de composition est variable, la conception étant basée sur un arbre ;
- **Decorator** : permet d'ajouter dynamiquement des fonctionnalités supplémentaires à un objet ;
- **Facade** : a pour but de regrouper les interfaces d'un ensemble d'objets en une interface unifiée rendant cet ensemble plus simple à utiliser ;
- **Flyweight** : facilite le partage d'un ensemble important d'objets dont le grain est fin ;
- **Proxy** : construit un objet qui se substitue à un autre objet et qui contrôle son accès ;
- **Chain of responsibility** : crée une chaîne d'objets telle que si un objet de la chaîne ne peut pas répondre à une requête, il puisse la transmettre à ses successeurs jusqu'à ce que l'un d'entre eux y réponde ;
- **Command** : a pour objectif de transformer une requête en un objet, facilitant des opérations comme l'annulation, la mise en file des requêtes et leur suivi ;
- **Interpreter** : fournit un cadre pour donner une représentation par objets de la grammaire d'un langage afin d'évaluer, en les interprétant, des expressions écrites dans ce langage ;
- **Iterator** : fournit un accès séquentiel à une collection d'objets sans que les clients se préoccupent de l'implantation de cette collection ;
- **Mediator** : construit un objet dont la vocation est la gestion et le contrôle des interactions au sein d'un ensemble d'objets sans que ses éléments se connaissent mutuellement ;
- **Memento** : sauvegarde et restaure l'état d'un objet ;
- **Observer** : construit une dépendance entre un sujet et des observateurs de façon à ce que chaque modification du sujet soit notifiée aux observateurs afin qu'ils puissent mettre à jour leur état ;

- **State** : permet à un objet d'adapter son comportement en fonction de son état interne ;
- **Strategy** : adapte le comportement et les algorithmes d'un objet en fonction d'un besoin sans changer les interactions avec les clients de cet objet ;
- **Template Method** : permet de reporter dans des sous-classes certaines étapes de l'une des opérations d'un objet, ces étapes étant alors décrites dans les sous-classes ;
- **Visitor** : construit une opération à réaliser sur les éléments d'un ensemble d'objets. De nouvelles opérations peuvent ainsi être ajoutées sans modifier les classes de ces objets.

Comment choisir et utiliser un pattern de conception pour résoudre un problème

Pour savoir s'il existe un pattern de conception qui réponde à un problème donné, une première étape consiste à regarder les descriptions de la section précédente et à déterminer s'il existe un ou plusieurs patterns dont la description s'approche de celle du problème.

Puis, il convient d'étudier en détail le ou les patterns découverts à l'aide de leur description complète se trouvant dans les parties 2 à 4. En particulier, il convient d'étudier au travers de l'exemple fourni et de la structure générique si le pattern répond de façon pertinente au problème. Cette étude doit surtout inclure la possibilité pour la structure générique d'être adaptée et il faut donc, en fait, chercher si le pattern après adaptation répond au problème. Cette étape d'adaptation est une étape importante de l'utilisation du pattern pour résoudre un problème. Nous la décrivons par la suite.

Une fois qu'un pattern est choisi, son utilisation dans une application comprend plusieurs étapes :

- étudier de façon approfondie sa structure générique qui sert de base pour utiliser un pattern ;
- renommer les classes et les méthodes introduites dans la structure générique. En effet, dans la structure générique d'un pattern, le nom des classes et des méthodes est abstrait. À l'inverse, une fois intégrées dans une application, ces classes et méthodes doivent être respectivement nommées relativement aux objets qu'elles décrivent et aux opérations qu'elles réalisent. Cette étape est le minimum obligatoire à réaliser pour utiliser un pattern ;
- adapter la structure générique pour répondre aux contraintes de l'application, ce qui peut impliquer des changements dans le schéma d'objets.

Nous donnons un exemple d'adaptation du pattern `Template method` dont un exemple a été introduit dans la première section de ce chapitre. La structure générique de ce pattern est donnée à la figure 1.2.

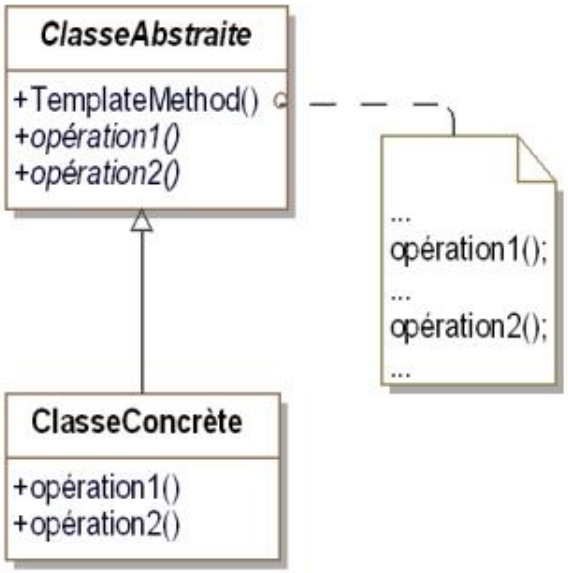


Figure 1.2 - Structure générique du pattern Template Method

Nous voulons adapter cette structure dans le cadre d'une application de commerce où la méthode du calcul de la TVA n'est pas incluse dans la classe `Commande` mais dans les sous-classes concrètes de la classe abstraite `Pays`. Ces sous-classes contiennent toutes les méthodes de calcul des taxes spécifiques à chaque pays.

La méthode `calculeTVA` de la classe `Commande` va alors invoquer la méthode `calculeTVA` de la sous-classe du pays concerné au travers d'une instance de cette sous-classe. Cette instance peut être transmise en paramètre lors de la création de la commande.

Le pattern adapté prêt à l'utilisation dans l'application est illustré à la figure 1.3.

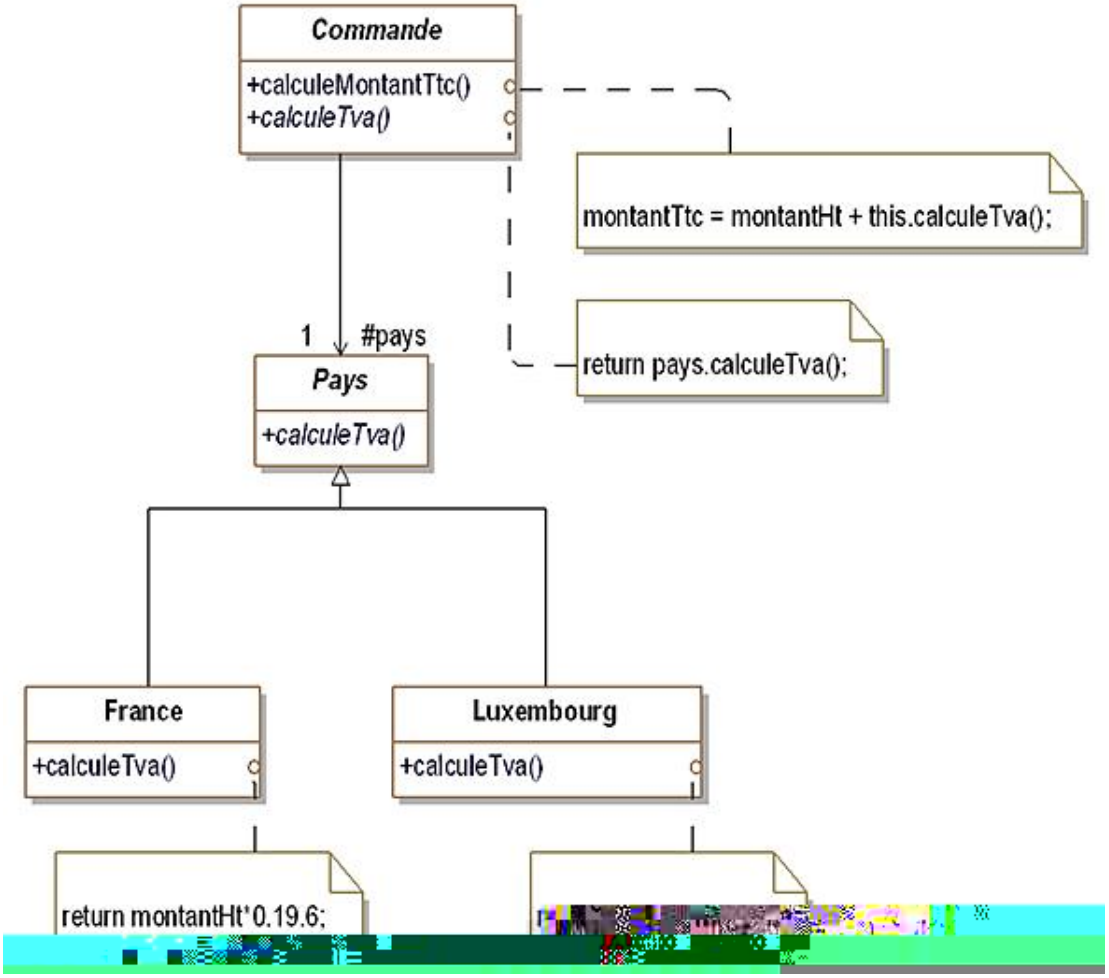


Figure 1.3 - Exemple d'utilisation avec adaptation du pattern Template Method

Organisation du catalogue des patterns de conception

Pour organiser le catalogue des patterns de conception, nous reprenons la classification du "GoF" qui organise les patterns selon leur vocation : construction, structuration et comportement.

Les patterns de construction ont pour but d'organiser la création d'objets. Ils sont décrits dans la partie 2. Ils sont au nombre de cinq : Abstract Factory, Builder, Factory Method, Prototype et Singleton.

Les patterns de structuration facilitent l'organisation de la hiérarchie des classes et de leurs relations. Ils sont décrits dans la partie 3. Ils sont au nombre de sept : Adapter, Bridge, Composite, Decorator, Facade, Flyweight et Proxy.

Enfin, les patterns de comportement proposent des solutions pour organiser les interactions et pour répartir les traitements entre les objets. Ils sont décrits dans la partie 4. Ils sont au nombre de onze : Chain of responsibility, Command, Interpreter, Iterator, Mediator, Memento, Observer, State, Strategy, Template Method et Visitor.

Description du système

Dans cet ouvrage, nous prenons l'exemple de la conception d'un système pour illustrer la mise en œuvre des vingt-trois patterns de conception.

Le système à concevoir est un site web de vente en ligne de véhicules comme, par exemple, des automobiles ou des scooters. Ce système autorise différentes opérations comme l'affichage d'un catalogue, la prise de commande, la gestion et le suivi de clientèle. De plus il est également accessible sous la forme d'un web service.

Cahier des charges

Le site permet d'afficher un catalogue de véhicules proposés à la vente, d'effectuer des recherches au sein de ce catalogue, de passer commande d'un véhicule, de choisir des options pour celui-ci avec un système de chariot virtuel. Les options incompatibles entre elles doivent également être gérées (par exemple "sièges sportifs" et "sièges en cuir" sont des options incompatibles). Il est également possible de revenir à un état précédent du chariot.

Le système doit gérer les commandes. Il doit être capable de calculer les taxes en fonction du pays de livraison du véhicule. Il doit également gérer les commandes payées au comptant et celles assorties d'une demande de crédit. Il prend en compte les demandes de crédit. Le système gère les états de la commande : en cours, validée et livrée.

Lors de la commande d'un véhicule, le système construit la liasse des documents nécessaires comme la demande d'immatriculation, le certificat de cession et le bon de commande. Ces documents sont disponibles au format PDF ou au format HTML.

Le système permet également de solder les véhicules difficiles à vendre, à savoir ceux qui sont dans le stock depuis longtemps.

Il permet également une gestion des clients, en particulier des sociétés possédant des filiales afin de leur proposer, par exemple, l'achat d'une flotte de véhicules.

Lors de la visualisation du catalogue, il est possible de visualiser des animations associées à un véhicule. Le catalogue peut être présenté avec un ou trois véhicules par ligne.

La recherche dans le catalogue peut s'effectuer à l'aide de mots clés et d'opérateurs logiques (et, ou).

Il est possible d'accéder au système via une interface web classique ou au travers d'un système de web services.

Prise en compte des patterns de conception

Afin de réaliser les différentes contraintes exprimées dans le cahier des charges, nous utilisons dans les chapitres suivants les patterns de conception. Ceux-ci sont pris en compte dans les parties suivantes de la conception du site web :

Description de la partie	Pattern de conception
Construire les objets du domaine (Automobile à essence, automobile à électricité, etc.).	Abstract Factory
Construire les liasses de documents nécessaires en cas d'acquisition d'un véhicule.	Builder, Prototype
Créer les commandes.	Factory Method
Créer la liasse vierge de documents.	Singleton
Gérer des documents PDF.	Adapter
Implanter des formulaires en HTML ou à l'aide d'une applet.	Bridge
Représenter les sociétés clientes.	Composite
Afficher les véhicules du catalogue.	Decorator, Observer, Strategy
Fournir l'interface en service web du site.	Facade
Gérer les options d'un véhicule commandé.	Flyweight, Memento
Gérer l'affichage d'animations pour chaque véhicule du catalogue.	Proxy
Gérer la description d'un véhicule.	Chain of responsibility
Solder les véhicules restés en stock pendant une longue durée.	Command
Rechercher dans la base de véhicules à l'aide d'une requête écrite sous la forme d'une expression logique.	Interpreter
Retrouver séquentiellement les véhicules du catalogue.	Iterator
Gérer le formulaire d'une demande de crédit.	Mediator
Gérer les états d'une commande.	State
Calculer le montant d'une commande.	Template Method
Envoyer des propositions commerciales par e-mail à certaines sociétés clientes.	Visitor

Exemple en Java

Nous proposons de simuler la saisie d'un formulaire à l'aide d'entrées/sorties classiques en bouclant sur une saisie séquentielle de chaque contrôle jusqu'à ce que le bouton « OK » soit validé (au clavier). Un menu permet de choisir si l'emprunt se fait avec ou sans co-emprunteur.

Le code Java de la classe `Controle` s'écrit à la suite.

```
public abstract class Controle
{
  protected String valeur = "";
  protected Formulaire directeur;
  protected String nom;

  public Controle(String nom)
  {
    this.nom = nom;
  }

  public abstract void saisie();

  public String getNom()
  {
    return nom;
  }

  public void setDirecteur(Formulaire directeur)
  {
    this.directeur = directeur;
  }

  public String getValeur()
  {
    return valeur;
  }

  protected void modifie()
  {
    directeur.controleModifie(this);
  }
}
```

Le code source de la sous-classe `ZoneSaisie` est le suivant. La méthode saisie est très simple, elle lit la valeur de la zone au clavier.

```
import java.util.*;
public class ZoneSaisie extends Controle
{
  Scanner reader = new Scanner(System.in);

  public ZoneSaisie(String nom)
  {
    super(nom);
  }

  public void saisie()
  {
    System.out.println("Saisie de : " + nom);
    valeur = reader.nextLine();
    this.modifie();
  }
}
```

La sous-classe `Bouton` a le code décrit à la suite. La méthode `saisie` demande à l'utilisateur s'il désire activer le bouton et en cas de réponse favorable, invoque la méthode `modifie (modifié)` pour signaler cette réponse au médiateur.

```
import java.util.*;
public class Bouton extends Controle
```

```
{
  Scanner reader = new Scanner(System.in);

  public Bouton(String nom)
  {
    super(nom);
  }

  public void saisie()
  {
    System.out.println("Désirez-vous activer le bouton "
      + nom + " ?");
    String reponse = reader.nextLine();
    if (reponse.equals("oui"))
      this.modifie();
  }
}
```

La méthode `saisie` de la sous-classe `PopupMenu` affiche toutes les options possibles puis demande à l'utilisateur son choix et en cas de changement de valeur, le signale au médiateur en invoquant la méthode `modifie`.

```
import java.util.*;
public class PopupMenu extends Controle
{
  protected List<String> options =
    new ArrayList<String>();
  protected Scanner reader = new Scanner(System.in);

  public PopupMenu(String nom)
  {
    super(nom);
  }

  public void saisie()
  {
    System.out.println("Saisie de : " + nom);
    System.out.println("Valeur actuelle : " + valeur);
    for (int index = 0; index < options.size(); index++)
      System.out.println("- " + index + " )" +
        options.get(index));
    int choix = reader.nextInt();
    if ((choix >= 0) && (choix < options.size()))
    {
      boolean change = !(valeur.equals(options.get(choix))
        );
      if (change)
      {
        valeur = options.get(choix);
        this.modifie();
      }
    }
  }

  public void ajouteOption(String option)
  {
    options.add(option);
  }
}
```

La classe `Formulaire` est présentée à la suite et introduit deux méthodes importantes :

- la méthode `saisie` fonctionne en bouclant sur la saisie de chaque contrôle jusqu'au moment où l'attribut `enCours` devient faux ;
- la méthode `controleModifie` demande la saisie des informations du co-emprunteur si le contrôle `menuCoemprunteur` change de valeur et prend la valeur "avec coemprunteur". Elle met également à faux la valeur de l'attribut `enCours` si le bouton « OK » est activé.

```
import java.util.*;
public class Formulaire
{
  protected List<Controle> controles =
    new ArrayList<Controle>();
  protected List<Controle> controlesCoemprunteur =
    new ArrayList<Controle>();
  protected PopupMenu menuCoemprunteur;
  protected Bouton boutonOK;
  protected boolean enCours = true;

  public void ajouteControle(Controle controle)
  {
    controles.add(controle);
    controle.setDirecteur(this);
  }

  public void ajouteControleCoemprunteur(Controle
    controle)
  {
    controlesCoemprunteur.add(controle);
    controle.setDirecteur(this);
  }

  public void setMenuCoemprunteur(PopupMenu
    menuCoemprunteur)
  {
    this.menuCoemprunteur = menuCoemprunteur;
  }

  public void setBoutonOK(Bouton boutonOK)
  {
    this.boutonOK = boutonOK;
  }

  public void controleModifie(Controle controle)
  {
    if ((controle == menuCoemprunteur) &&
      (controle.getValeur().equals("avec coemprunteur")))
    {
      for (Controle elementCoemprunteur:
        controlesCoemprunteur)
        elementCoemprunteur.saisie();
    }
    if (controle == boutonOK)
    {
      enCours = false;
    }
  }

  public void saisie()
  {
    while (true)
    {
      for (Controle controle: controles)
      {
        controle.saisie();
        if (!enCours)
          return ;
      }
    }
  }

}
```

Enfin, la classe `Utilisateur` contient le programme principal qui effectue les actions suivantes :

- construction du formulaire ;

- ajout de deux zones de saisie ;
- ajout du menu Coemprunteur ;
- ajout du bouton « OK » ;
- ajout des zones de saisie pour le co-emprunteur ;
- lancement de la saisie du formulaire.

```
public class Utilisateur
{

  public static void main(String[] args)
  {
    Formulaire formulaire = new Formulaire();
    formulaire.ajouteControle(new ZoneSaisie("Nom"));
    formulaire.ajouteControle(new ZoneSaisie("Prénom"));
    PopupMenu menu = new PopupMenu("Coemprunteur");
    menu.ajouteOption("sans coemprunteur");
    menu.ajouteOption("avec coemprunteur");
    formulaire.ajouteControle(menu);
    formulaire.setMenuCoemprunteur(menu);
    Bouton bouton = new Bouton("OK");
    formulaire.ajouteControle(bouton);
    formulaire.setBoutonOK(bouton);
    formulaire.ajouteControleCoemprunteur(new ZoneSaisie(
      "Nom du coemprunteur"));
    formulaire.ajouteControleCoemprunteur(new ZoneSaisie(
      "Prénom du coemprunteur"));
    formulaire.saisie();
  }
}
```

Un exemple d'exécution est donné ci-dessous. Les mots en gras sont les saisies de l'utilisateur.

```
Saisie de : Nom
Martin
Saisie de : Prénom
Jean
Saisie de : Coemprunteur
Valeur actuelle :
- 0 )sans coemprunteur
- 1 )avec coemprunteur
1
Saisie de : Nom du coemprunteur
Dupont
Saisie de : Prénom du coemprunteur
Henri
Désirez-vous activer le bouton OK ?
oui
```

Description

Le pattern `Memento` a pour but de sauvegarder et de restaurer l'état d'un objet sans en violer l'encapsulation.

Exemple

Lors de l'achat en ligne d'un véhicule neuf, le client peut choisir des options supplémentaires qui vont être ajoutées à son chariot. Cependant, il existe des options incompatibles comme, par exemple, les sièges sportifs qui sont incompatibles avec les sièges en cuir ou les accoudoirs.

La conséquence de cette incompatibilité est que si les accoudoirs ont été choisis et qu'ensuite les sièges sportifs sont choisis, l'option des accoudoirs est retirée du chariot.

Nous désirons ensuite ajouter une option d'annulation de la dernière opération effectuée dans le chariot. Retirer la dernière option ajoutée n'est pas suffisant car il faut aussi remettre les options présentes et qui ont été retirées pour cause d'incompatibilité. Une solution consiste à mémoriser l'état du chariot avant l'ajout d'une nouvelle option.

Par la suite, nous souhaitons étendre ce mécanisme pour gérer un historique des états du chariot et pouvoir revenir à n'importe quel état. Il faut alors, dans ce cas, mémoriser tous les états successifs du chariot.

Pour préserver l'encapsulation de l'objet représentant le chariot, une solution consisterait à mémoriser ces états intermédiaires dans le chariot. Cependant cette solution aurait pour effet de complexifier inutilement cet objet.

Le pattern `Memento` propose une réponse à ce problème. Elle consiste à mémoriser les états du chariot dans un objet appelé mémento. Lors de l'ajout d'une nouvelle option, le chariot crée un mémento, l'initialise avec son état, retire les options incompatibles avec cette nouvelle option, procède à l'ajout de cette nouvelle option et renvoie le mémento ainsi créé. Celui-ci sera utilisé par la suite en cas d'annulation de cet ajout et de retour à l'état précédent.

Seul le chariot peut mémoriser son état dans le mémento et y restaurer un état précédent : le mémento est opaque vis-à-vis des autres objets.

Le diagramme de classes correspondant est illustré à la figure 23.1. Le chariot y est représenté par la classe `ChariotOption` et le mémento par la classe `Mémento`. L'état du chariot consiste en l'ensemble de ses liens avec les options choisies. Les options sont représentées par la classe `OptionVéhicule` qui introduit une association réflexive pour décrire les options incompatibles.

Il convient de remarquer que les options forment un ensemble d'instances de la classe `OptionVéhicule`. Ces instances sont partagées par tous les chariots.

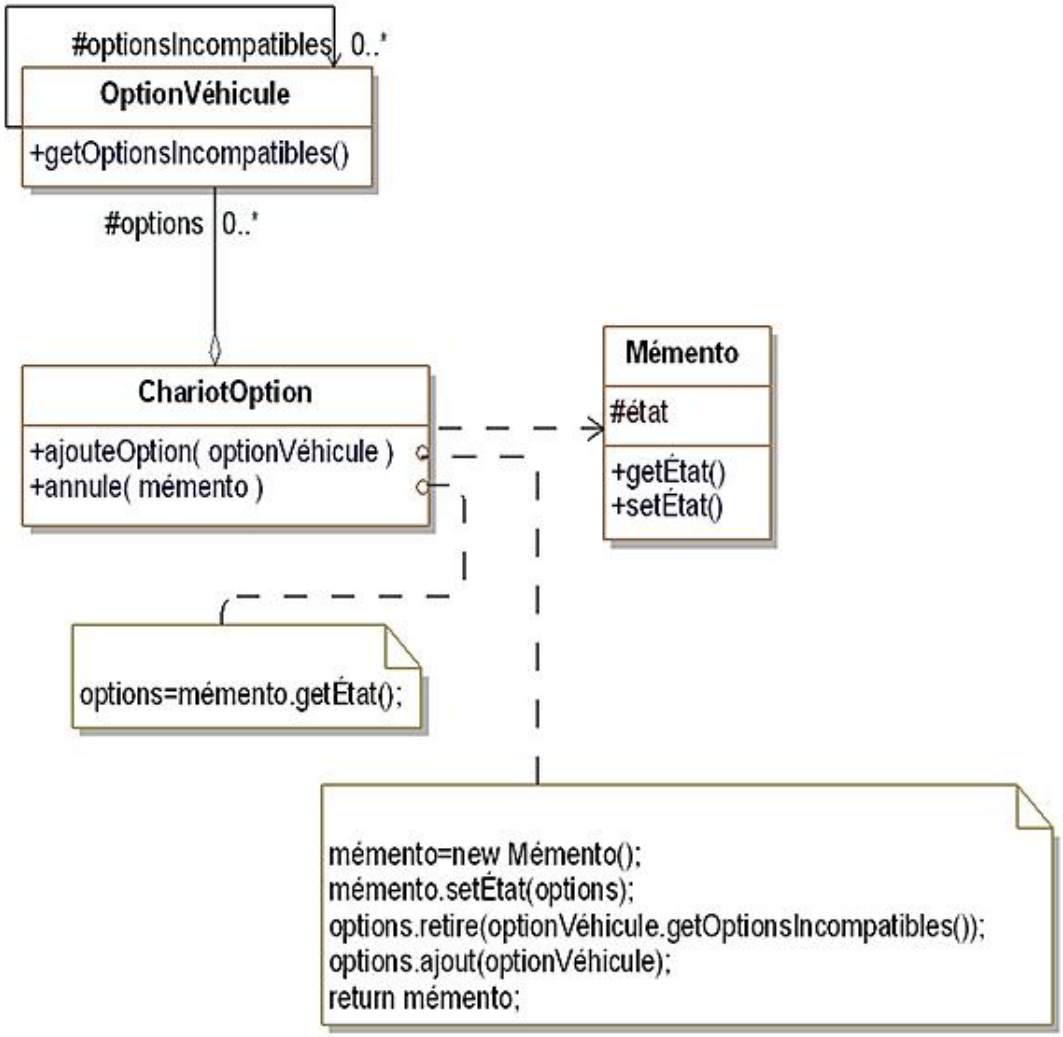


Figure 23.1 - Le pattern `Memento` pour gérer les états d'un chariot d'options

Structure

1. Diagramme de classes

La figure 23.2 détaille la structure générique du pattern.

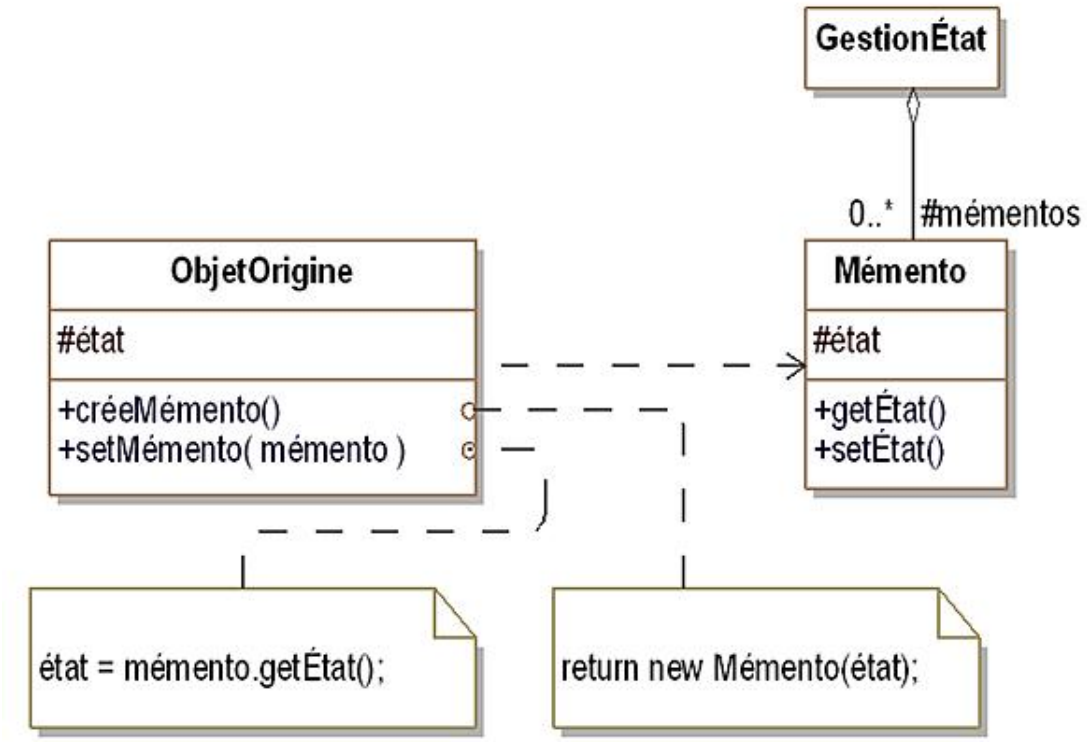


Figure 23.2 - Structure du pattern `Memento`

2. Participants

Les participants au pattern sont les suivants :

- `Mémento` est la classe des mémentos qui sont les objets qui mémorisent l'état interne des objets d'origine (ou une partie de cet état). Le mémento possède deux interfaces : une interface complète destinée aux objets d'origine qui offre la possibilité de mémoriser et restaurer leur état et une interface réduite pour les objets de gestion de l'état qui n'ont pas le droit d'accéder à l'état interne des objets d'origine ;
- `ObjetOrigine (ChariotOption)` est la classe des objets qui créent un mémento pour mémoriser leur état interne qu'ils peuvent également restaurer à partir d'un mémento ;
- `GestionÉtat` est responsable de la gestion des mémentos et n'accède pas à l'état interne des objets d'origine.

3. Collaborations

Une instance de `GestionÉtat` demande un mémento à l'objet d'origine par appel de la méthode `créeMémento`, le sauvegarde et en cas de besoin d'annulation et de retour à l'état mémorisé dans le mémento, le transmet à nouveau à l'objet d'origine par la méthode `setMémento`.

Domaines d’application

Le pattern est utilisé dans le cas où l’état interne d’un objet (totalement ou en partie) doit être mémorisé afin de pouvoir être restauré ultérieurement sans que l’encapsulation de cet objet ne doive être brisée.

Exemple en Java

Nous commençons la présentation de l'exemple Java par le mémento. Celui-ci est décrit par l'interface `Memento` et la classe `MementoImpl`. La classe introduit les méthodes `getEtat` et `setEtat` dont l'invocation est réservée au seul chariot. L'interface est vide, elle ne sert qu'à déterminer un type pour les autres objets qui doivent référencer le mémento sans pouvoir accéder aux méthodes `getEtat` et `setEtat`.

Le mémento stocke l'état du chariot d'options à savoir une liste qui est construite par duplicata de la liste des options du chariot.

```
public interface Memento
{
}

import java.util.ArrayList;
import java.util.List;
public class MementoImpl implements Memento
{
  protected List<OptionVehicule> options =
    new ArrayList<OptionVehicule>();

  public void setEtat(List<OptionVehicule> options)
  {
    this.options.clear();
    this.options.addAll(options);
  }

  public List<OptionVehicule> getEtat()
  {
    return options;
  }
}
```

La classe `ChariotOption` décrit les chariots. La méthode `ajouteOption` procède bien à la suppression des options incompatibles de la nouvelle option avant d'ajouter celle-ci. Cette méthode crée un nouveau mémento qui reçoit l'état initial, mémento qui est renvoyé à l'appelant de la méthode. La méthode `annule` revient à l'état sauvegardé dans le mémento. Il convient de noter qu'un downcast est nécessaire pour retrouver un accès à la méthode `getEtat`.

```
import java.util.*;
public class ChariotOption
{
  protected List<OptionVehicule> options =
    new ArrayList<OptionVehicule>();

  public Memento ajouteOption(OptionVehicule
    optionVehicule)
  {
    MementoImpl resultat = new MementoImpl();
    resultat.setEtat(options);
    options.removeAll
      (optionVehicule.getOptionsIncompatibles());
    options.add(optionVehicule);
    return resultat;
  }

  public void annule(Memento memento)
  {
    MementoImpl mementoImplInstance;
    try
    {
      mementoImplInstance = (MementoImpl)memento;
    }
    catch (ClassCastException e)
    {
      return ;
    }
    options = mementoImplInstance.getEtat();
  }
```

```
  public void affiche()
  {
    System.out.println("Contenu du chariot des options");
    for (OptionVehicule option: options)
      option.affiche();
    System.out.println();
  }
}
```

La classe `OptionVehicule` décrit une option de véhicule neuf.

```
import java.util.*;
public class OptionVehicule
{
  protected String nom;
  protected List<OptionVehicule> optionsIncompatibles =
    new ArrayList<OptionVehicule>();

  public OptionVehicule(String nom)
  {
    this.nom = nom;
  }

  public void ajouteOptionIncompatible(OptionVehicule
    optionIncompatible)
  {
    if (!optionsIncompatibles.contains(optionIncompatible))
    {
      optionsIncompatibles.add(optionIncompatible);
      optionIncompatible.ajouteOptionIncompatible(this);
    }
  }

  public List<OptionVehicule> getOptionsIncompatibles()
  {
    return optionsIncompatibles;
  }

  public void affiche()
  {
    System.out.println("option : " + nom);
  }
}
```

Enfin le programme principal est introduit par la classe `Utilisateur`. Il commence par créer la liste des options et de spécifier les options incompatibles. Il ajoute ensuite les deux premières options puis la troisième qui est incompatible avec les deux premières. Ensuite, il annule ce dernier ajout.

```
public class Utilisateur
{
  public static void main(String[] args)
  {
    Memento memento;
    OptionVehicule option1 = new OptionVehicule(
      "Sièges en cuir");
    OptionVehicule option2 = new OptionVehicule(
      "Accoudoirs");
    OptionVehicule option3 = new OptionVehicule(
      "Sièges sportifs");
    option1.ajouteOptionIncompatible(option3);
    option2.ajouteOptionIncompatible(option3);
    ChariotOption chariotOptions = new ChariotOption();
    chariotOptions.ajouteOption(option1);
    chariotOptions.ajouteOption(option2);
    chariotOptions.affiche();
    memento = chariotOptions.ajouteOption(option3);
    chariotOptions.affiche();
    chariotOptions.annule(memento);
```

```
    chariotOptions.affiche();
  }
}
```

Le résultat de l'exécution du programme est le suivant.

```
Contenu du chariot des options
option : Sièges en cuir
option : Accoudoirs

Contenu du chariot des options
option : Sièges sportifs

Contenu du chariot des options
option : Sièges en cuir
option : Accoudoirs
```

Description

Le pattern `Observer` a pour objectif de construire une dépendance entre un sujet et des observateurs de sorte que chaque modification du sujet soit notifiée aux observateurs afin qu'ils puissent mettre à jour leur état.

Exemple

Nous voulons mettre à jour l'affichage d'un catalogue de véhicules en temps réel. Chaque fois que les informations relatives à un véhicule sont modifiées, nous voulons mettre à jour l'affichage de celles-ci. Il peut y avoir plusieurs affichages simultanés.

La solution préconisée par le pattern `Observer` consiste à établir un lien entre chaque véhicule et ses vues pour que le véhicule puisse leur indiquer de se mettre à jour quand son état interne a été modifié. Cette solution est illustrée à la figure 24.1.

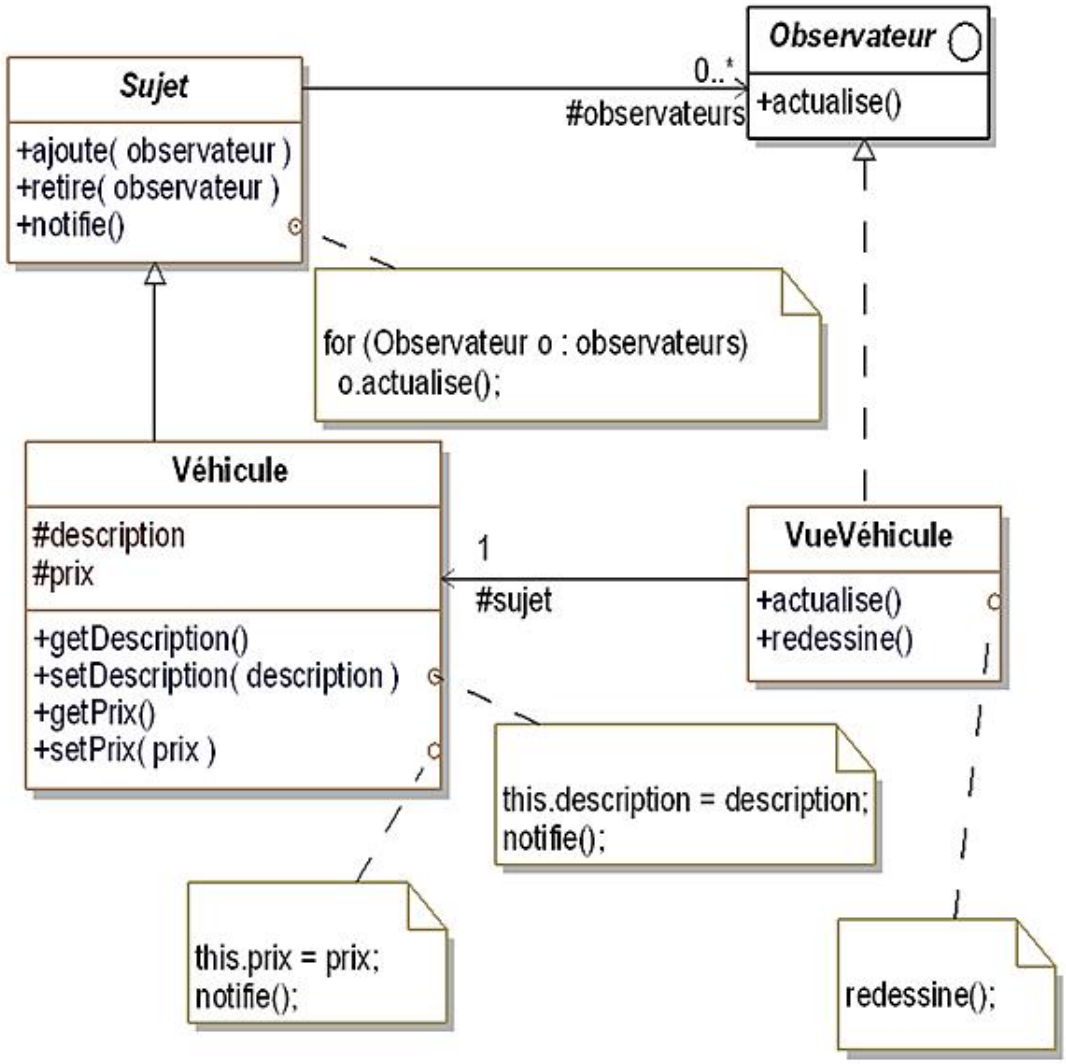


Figure 24.1 - Le pattern `Observer` appliqué à l'affichage de véhicules

Le diagramme contient les quatre classes suivantes :

- `Sujet` est la classe abstraite qui introduit tout objet qui notifie d'autres objets des modifications de son état interne ;
- `Véhicule` est la sous-classe concrète de `Sujet` qui décrit les véhicules. Elle gère deux attributs : `description` et `prix` ;
- `Observateur` est l'interface de tout objet qui a besoin de recevoir des notifications de changement d'état provenant des objets auprès desquels il s'est préalablement inscrit ;
- `VueVéhicule` est la sous-classe concrète d'implantation de `Observateur` dont les instances affichent les informations d'un véhicule.

Le fonctionnement est le suivant : chaque nouvelle vue s'inscrit en tant qu'observateur auprès de son véhicule à l'aide de la méthode `ajoute`. Chaque fois que la description ou le prix sont mis à jour, la méthode `notifie` est appelée. Celle-ci demande à tous les observateurs de se mettre à jour en invoquant leur méthode `actualise`. Dans la classe `VueVéhicule`, cette dernière méthode appelle `redessine`. Ce fonctionnement est illustré à la figure 24.2 par un diagramme de séquence.

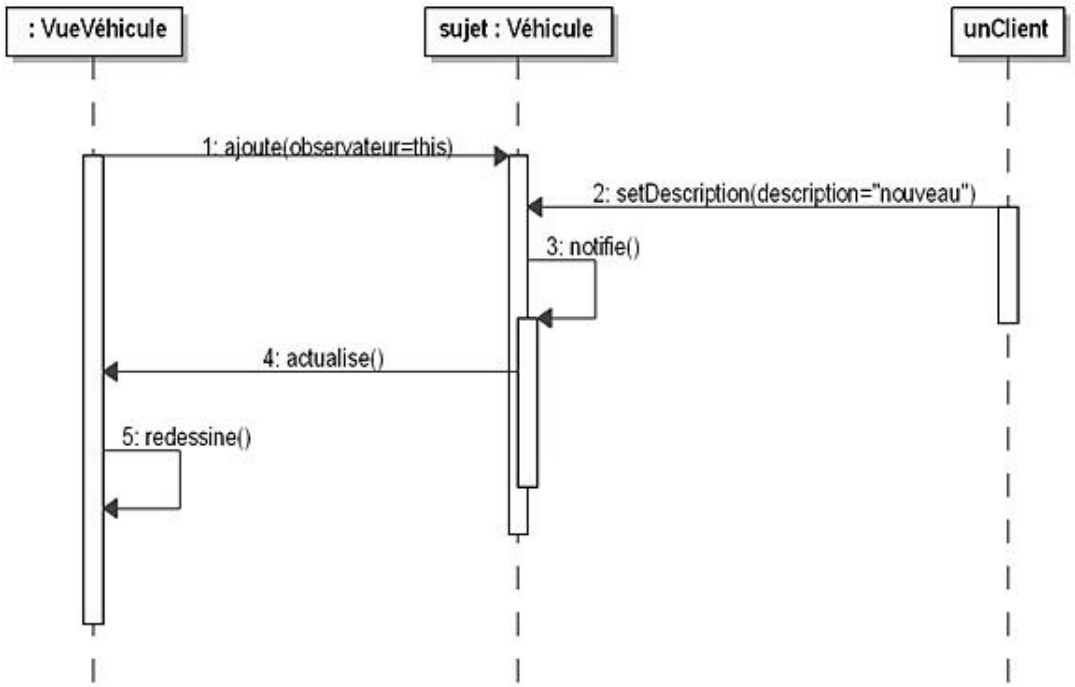


Figure 24.2 - Diagramme de séquence détaillant l'utilisation du pattern `Observer`

La solution mise en œuvre par le pattern `Observer` est générique. En effet, tout le mécanisme d'observation est implanté dans la classe `Sujet` et l'interface `Observateur` qui peuvent avoir d'autres sous-classes que `Véhicule` et `VueVéhicule`.

Structure

1. Diagramme de classes

La figure 24.3 détaille la structure générique du pattern.

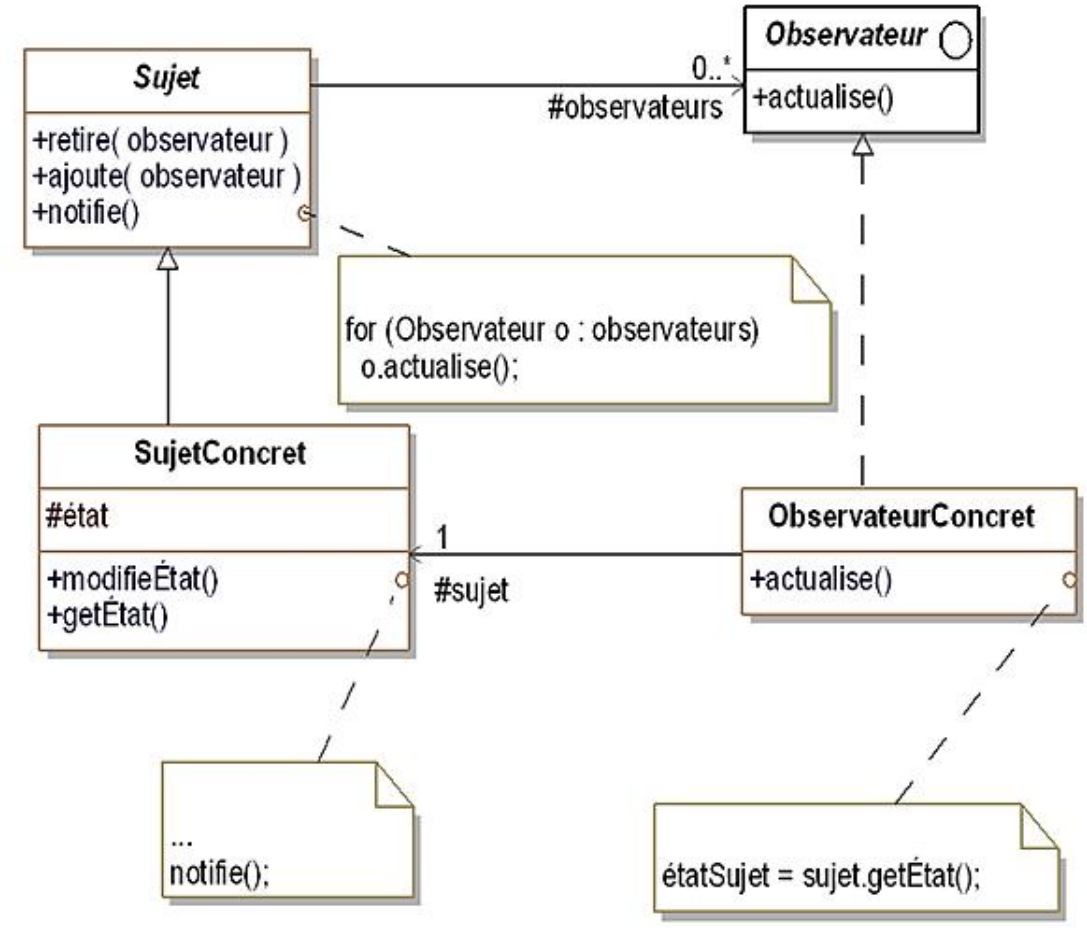


Figure 24.3 - Structure du pattern `Observer`

2. Participants

Les participants au pattern sont les suivants :

- `Sujet` est la classe abstraite qui introduit l'association avec les observateurs ainsi que les méthodes pour ajouter ou retirer des observateurs ;
- `Observateur` est l'interface à implanter pour recevoir des notifications (méthode `actualise`) ;
- `SujetConcret` (`Véhicule`) est une classe d'implantation d'un sujet. Un sujet envoie une notification quand son état est modifié ;
- `ObservateurConcret` (`VueVéhicule`) est une classe d'implantation d'un observateur. Celui-ci maintient une référence vers son sujet et implante la méthode `actualise`. Elle demande à son sujet des informations faisant partie de son état lors des mises à jour par invocation de la méthode `getÉtat`.

3. Collaborations

Le sujet concret notifie ses observateurs lorsque son état interne est modifié. Lorsqu'un observateur reçoit cette notification, il se met à jour en conséquence. Pour réaliser cette mise à jour, il peut invoquer des méthodes du sujet donnant accès à son état.

Domaines d’application

Le pattern est utilisé dans les cas suivants :

- une modification dans l’état d’un objet engendre des modifications dans d’autres objets qui sont déterminés dynamiquement ;
- un objet veut prévenir d’autres objets sans devoir connaître leur type, c’est-à-dire sans être fortement couplé à ceux-ci ;
- on ne veut pas fusionner deux objets en un seul.

Présentation

Les patterns de construction ont pour vocation d'abstraire les mécanismes de création d'objets. Un système utilisant ces patterns devient indépendant de la façon dont les objets sont créés et, en particulier, des mécanismes d'instanciation des classes concrètes.

Ces patterns encapsulent l'utilisation des classes concrètes et favorisent ainsi l'utilisation des interfaces dans les relations entre objets, augmentant les capacités d'abstraction dans la conception globale du système.

Ainsi, le pattern `Singleton` permet de construire une classe possédant au maximum une instance. Le mécanisme gérant l'accès unique à cette seule instance est entièrement encapsulé dans la classe. Il est transparent pour les clients de cette classe.

Exemple en Java

Nous reprenons l'exemple de la figure 24.1. Le code source de la classe `Sujet` est donné à la suite. Les observateurs sont gérés à l'aide d'une liste.

```
import java.util.*;
public abstract class Sujet
{
  protected List<Observateur> observateurs =
    new ArrayList<Observateur>();

  public void ajoute(Observateur observateur)
  {
    observateurs.add(observateur);
  }

  public void retire(Observateur observateur)
  {
    observateurs.remove(observateur);
  }

  public void notifie()
  {
    for (Observateur observateur: observateurs)
      observateur.actualise();
  }
}
```

Le code source de l'interface `Observateur` est très simple car il ne contient que la signature de la méthode `actualise`.

```
public interface Observateur
{
  void actualise();
}
```

Le code source de la classe `Vehicule` se trouve à la suite. Elle contient deux attributs et les accesseurs en lecture et écriture pour ces deux attributs. Les deux accesseurs en écriture invoquent la méthode `notifie`.

```
public class Vehicule extends Sujet
{
  protected String description;
  protected Double prix;

  public String getDescription()
  {
    return description;
  }

  public void setDescription(String description)
  {
    this.description = description;
    this.notifie();
  }

  public Double getPrix()
  {
    return prix;
  }

  public void setPrix(Double prix)
  {
    this.prix = prix;
    this.notifie();
  }
}
```

La classe `VueVehicule` gère un texte qui contient la description et le prix du véhicule associé (le sujet). Ce texte est mis

à jour lors de chaque notification dans le corps la méthode `actualise`. La méthode `affiche` imprime ce texte à l'écran.

```
public class VueVehicule implements Observateur
{
  protected Vehicule vehicule;
  protected String texte = "";

  public VueVehicule(Vehicule vehicule)
  {
    this.vehicule = vehicule;
    vehicule.ajoute(this);
    this.actualise();
  }

  public void actualise()
  {
    texte = "Description " + vehicule.getDescription()
      + " Prix : " + vehicule.getPrix();
  }

  public void affiche()
  {
    System.out.println(texte);
  }
}
```

Enfin, la classe `Utilisateur` introduit le programme principal. Celui-ci crée un véhicule puis une vue dont il demande l'affichage. Le prix est ensuite modifié et la vue est réaffichée. Puis une seconde vue est créée associée au même véhicule. Le prix est à nouveau modifié et les deux vues sont réaffichées.

```
public class Utilisateur
{

  public static void main(String[] args)
  {
    Vehicule vehicule = new Vehicule();
    vehicule.setDescription("Véhicule bon marché");
    vehicule.setPrix(5000.0);
    VueVehicule vueVehicule = new VueVehicule(vehicule);
    vueVehicule.affiche();
    vehicule.setPrix(4500.0);
    vueVehicule.affiche();
    VueVehicule vueVehicule2 = new VueVehicule(vehicule);
    vehicule.setPrix(5500.0);
    vueVehicule.affiche();
    vueVehicule2.affiche();
  }
}
```

Le résultat de l'exécution de ce programme est à la suite.

```
Description Véhicule bon marché Prix : 5000.0
Description Véhicule bon marché Prix : 4500.0
Description Véhicule bon marché Prix : 5500.0
Description Véhicule bon marché Prix : 5500.0
```

Description

Le pattern `State` permet à un objet d’adapter son comportement en fonction de son état interne.

Exemple

Nous nous intéressons aux commandes de produits sur notre site de vente en ligne. Elles sont décrites par la classe `Commande`. Les instances de cette classe possèdent un cycle de vie qui est illustré par le diagramme d'états-transitions de la figure 25.1. L'état `EnCours` est l'état où la commande est en cours de constitution : le client ajoute des produits. L'état `Validée` est l'état où la commande a été validée et réglée par le client. Enfin l'état `Livrée` est l'état où les produits ont été livrés.

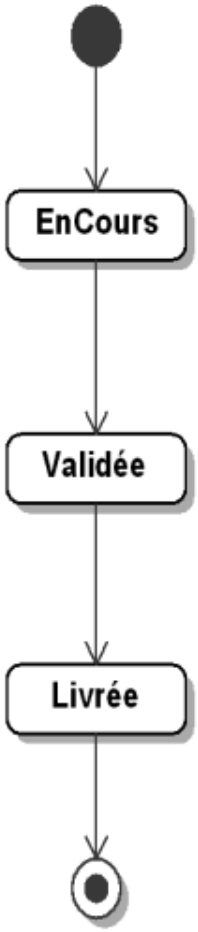


Figure 25.1 - Diagramme d'états-transitions d'une commande

La classe `Commande` possède des méthodes dont le comportement diffère en fonction de cet état. Par exemple, la méthode `ajouteProduit` n'ajoute des produits que si la commande se trouve dans l'état `EnCours`. La méthode `efface` n'a pas de comportement dans l'état `Livrée`.

L'approche traditionnelle pour résoudre ces différences de comportement consiste à utiliser des conditions dans le corps des méthodes. Cette approche conduit souvent à des méthodes complexes à écrire et à appréhender.

Le pattern `State` propose une autre solution qui consiste à transformer chaque état en une classe. Cette classe introduit les méthodes de la classe `Commande` dépendant des états en leur conférant le comportement propre à cet état.

La figure 25.2 illustre le diagramme de classes correspondant à cette approche. Les trois sous-classes correspondant aux états sont `CommandeEnCours`, `CommandeValidée` et `CommandeLivrée`. Elles sont sous-classes de la classe abstraite `ÉtatCommande` qui détient l'association avec la classe `Commande`. La classe `ÉtatCommande` introduit également les signatures des méthodes dont le comportement dépend de l'état courant, méthodes implantées dans ses sous-classes.

Une instance de la classe `Commande` possède une référence vers une instance de la sous-classe d'`ÉtatCommande` qui correspond à son état courant. Dans cette classe `Commande`, les méthodes qui dépendent de l'état courant déléguent leur invocation à cette instance. La note relative à la méthode `ajouteProduit` dans la figure 25.2 illustre cette délégation.

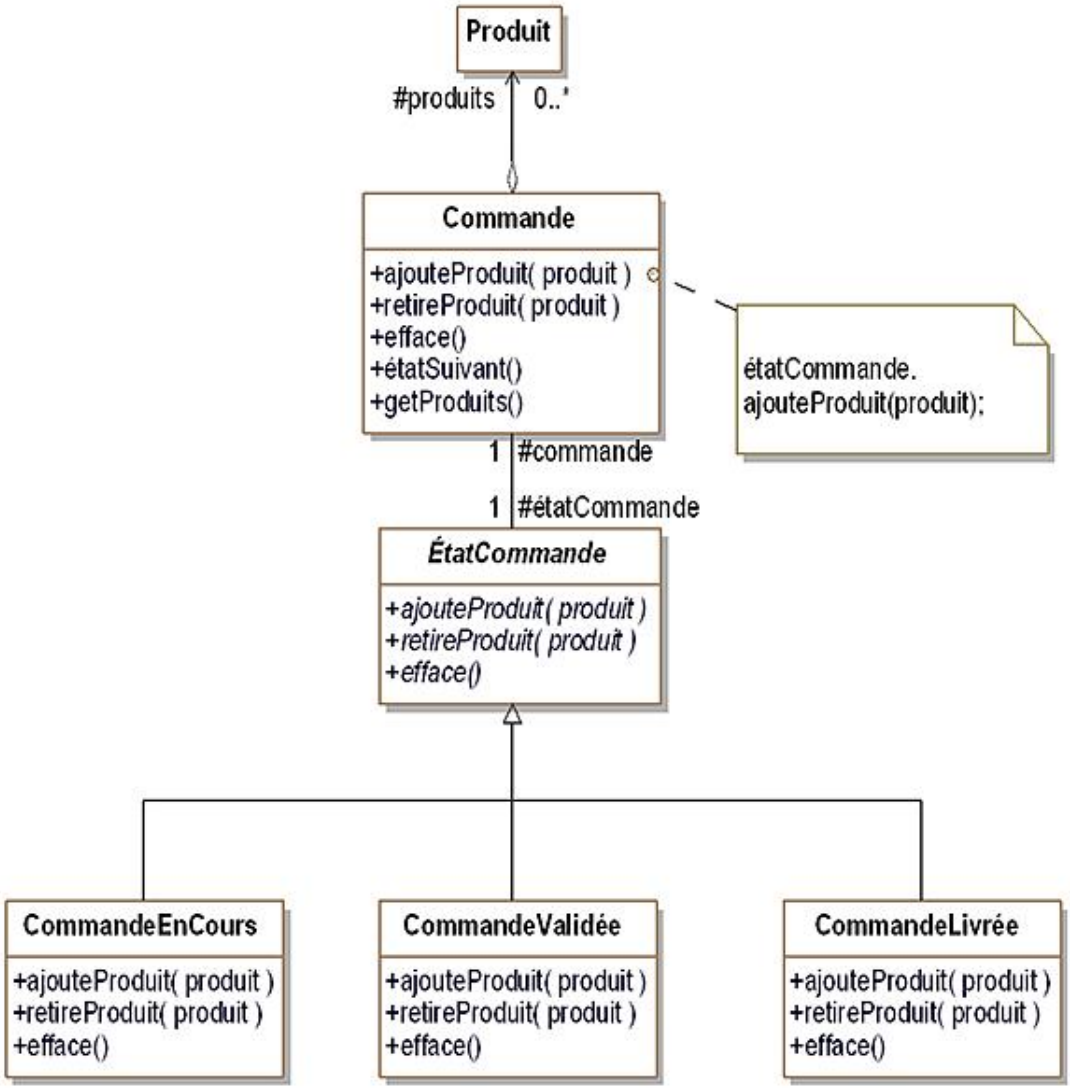


Figure 25.2 - Le pattern `State` appliqué aux états d'une commande

Structure

1. Diagramme de classes

La figure 25.3 illustre la structure générique du pattern.

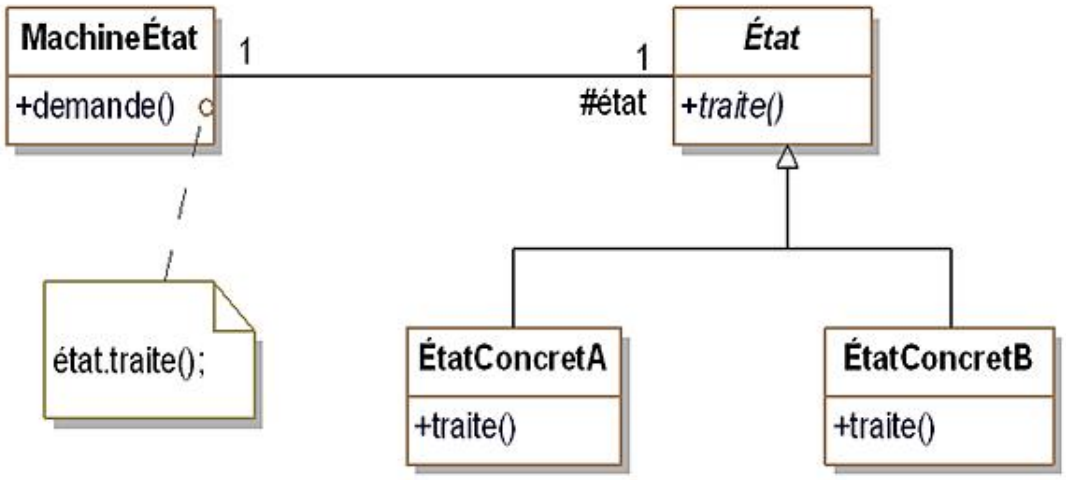


Figure 25.3 - Structure du pattern `State`

2. Participants

Les participants au pattern sont les suivants :

- `MachineÉtat (Commande)`est une classe concrète décrivant des objets qui sont des machines à états, c'est-à-dire qui possèdent un ensemble d'états pouvant être décrit par un diagramme d'états-transitions. Cette classe maintient une référence vers une instance d'une sous-classe d'État qui définit l'état courant ;
- `État (ÉtatCommande)`est une classe abstraite qui introduit la signature des méthodes liées à l'état et qui gère l'association avec la machine à états ;
- `ÉtatConcretA`et `ÉtatConcretB` (`CommandeEnCours`, `CommandeValidée` et `CommandeLivrée`) sont des sous-classes concrètes qui implantent le comportement des méthodes relativement à chaque état.

3. Collaborations

La machine à états délègue les appels des méthodes dépendant de l'état courant vers un objet d'état.

La machine à états peut transmettre à l'objet d'état une référence vers elle-même si c'est nécessaire. Cette référence peut être passée lors de chaque délégation ou à l'initialisation de l'objet d'état.

Domaines d’application

Le pattern est utilisé dans le cas suivant :

- le comportement d’un objet dépend de son état ;
- l’implantation de cette dépendance à l’état par des instructions conditionnelles est trop complexe.

Exemple en Java

Nous présentons l'exemple de la figure 25.2 en Java. La classe `Commande` est décrite à la suite. Les méthodes `ajouteProduit`, `retireProduit` et `efface` dépendent de l'état. Par conséquent leur implantation consiste à appeler la méthode correspondante de l'instance référencée par `etatCommande`.

Le constructeur de la classe initialise l'attribut `etatCommande` avec une instance de la classe `CommandeEnCours`. La méthode `etatSuivant` passe à l'état suivant en associant une nouvelle instance à l'attribut `etatCommande`.

```
import java.util.*;
public class Commande
{
  protected List<Produit> produits = new ArrayList<Produit>();
  protected EtatCommande etatCommande;

  public Commande()
  {
    etatCommande = new CommandeEnCours(this);
  }

  public void ajouteProduit(Produit produit)
  {
    etatCommande.ajouteProduit(produit);
  }

  public void retireProduit(Produit produit)
  {
    etatCommande.retireProduit(produit);
  }

  public void efface()
  {
    etatCommande.efface();
  }

  public void etatSuivant()
  {
    etatCommande = etatCommande.etatSuivant();
  }

  public List<Produit> getProduits()
  {
    return produits;
  }

  public void affiche()
  {
    System.out.println("Contenu de la commande");
    for (Produit produit: produits)
      produit.affiche();
    System.out.println();
  }
}
```

La classe abstraite `EtatCommande` gère le lien avec une instance de `Commande` ainsi que la signature des méthodes de `Commande` qui dépendent de l'état.

```
public abstract class EtatCommande
{
  protected Commande commande;

  public EtatCommande(Commande commande)
  {
    this.commande = commande;
  }

  public abstract void ajouteProduit(Produit produit);
```

```
  public abstract void efface();
  public abstract void retireProduit(Produit produit);
  public abstract EtatCommande etatSuivant();
}
```

La sous-classe `CommandeEnCours` implante les méthodes d'`EtatCommande` pour l'état `EnCours`.

```
public class CommandeEnCours extends EtatCommande
{
  public CommandeEnCours(Commande commande)
  {
    super(commande);
  }

  public void ajouteProduit(Produit produit)
  {
    commande.getProduits().add(produit);
  }

  public void efface()
  {
    commande.getProduits().clear();
  }

  public void retireProduit(Produit produit)
  {
    commande.getProduits().remove(produit);
  }

  public EtatCommande etatSuivant()
  {
    return new CommandeValidee(commande);
  }
}
```

La sous-classe `CommandeValidee` implante les méthodes d'`EtatCommande` pour l'état `Validée`.

```
public class CommandeValidee extends EtatCommande
{
  public CommandeValidee(Commande commande)
  {
    super(commande);
  }

  public void ajouteProduit(Produit produit){}

  public void efface()
  {
    commande.getProduits().clear();
  }

  public void retireProduit(Produit produit){}

  public EtatCommande etatSuivant()
  {
    return new CommandeLivree(commande);
  }
}
```

La sous-classe `CommandeLivree` implante les méthodes d'`EtatCommande` pour l'état `Livrée`. Dans cet état, le corps des méthodes est vide.

```
public class CommandeLivree extends EtatCommande
{
  public CommandeLivree(Commande commande)
  {
    super(commande);
  }
```

```
  public void ajouteProduit(Produit produit){}

  public void efface(){}

  public void retireProduit(Produit produit){}

  public EtatCommande etatSuivant()
  {
    return this;
  }
}
```

La classe `Produit` référencée par la classe `Commande` possède le code source Java suivant.

```
public class Produit
{
  protected String nom;

  public Produit(String nom)
  {
    this.nom = nom;
  }

  public void affiche()
  {
    System.out.println("Produit : " + nom);
  }
}
```

Le programme principal est introduit par la classe `Utilisateur`. Le programme crée deux commandes. Il efface la première dans l'état `Validée`, ce qui conduit bien à une remise à zéro. Quant à la seconde, elle est effacée par le programme une fois qu'elle se trouve dans l'état `Livrée`, ce qui ne provoque rien.

```
public class Utilisateur
{

  public static void main(String[] args)
  {
    Commande commande = new Commande();
    commande.ajouteProduit(new Produit("véhicule 1"));
    commande.ajouteProduit(new Produit("Accessoire 2"));
    commande.affiche();
    commande.etatSuivant();
    commande.ajouteProduit(new Produit("Accessoire 3"));
    commande.efface();
    commande.affiche();

    Commande commande2 = new Commande();
    commande2.ajouteProduit(new Produit("véhicule 11"));
    commande2.ajouteProduit(new Produit("Accessoire 21"));
    commande2.affiche();
    commande2.etatSuivant();
    commande2.affiche();
    commande2.etatSuivant();
    commande2.efface();
    commande2.affiche();
  }
}
```

L'exécution de ce programme fournit donc le résultat suivant.

```
Contenu de la commande
Produit : véhicule 1
Produit : Accessoire 2

Contenu de la commande

Contenu de la commande
Produit : véhicule 11
```

```
Produit : Accessoire 21

Contenu de la commande
Produit : véhicule 11
Produit : Accessoire 21

Contenu de la commande
Produit : véhicule 11
Produit : Accessoire 21
```

Description

Le pattern `Strategy` a pour objectif d'adapter le comportement et les algorithmes d'un objet en fonction d'un besoin sans changer les interactions de cet objet avec les clients.

Ce besoin peut relever de plusieurs aspects comme des aspects de présentation, d'efficacité en temps ou en mémoire, de choix d'algorithmes, de représentation interne, etc. Mais évidemment, il ne s'agit pas d'un besoin fonctionnel vis-à-vis des clients de l'objet car les interactions entre l'objet et ses clients doivent rester inchangées.

Exemple

Dans le système de vente en ligne de véhicules, la classe `VueCatalogue` dessine la liste des véhicules destinés à la vente. Un algorithme de dessin est utilisé pour calculer la mise en page en fonction du navigateur. Il existe deux versions de cet algorithme :

- une première version qui n'affiche qu'un seul véhicule par ligne (un véhicule prend toute la largeur disponible) et qui affiche le maximum d'informations ainsi que quatre photos ;
- une seconde version qui affiche trois véhicules par ligne mais qui affiche moins d'informations et une seule photo.

L'interface de la classe `VueCatalogue` ne dépend pas du choix de l'algorithme de mise en page. Ce choix n'a pas d'impact sur la relation d'une vue de catalogue avec ses clients. Il n'y a que la présentation qui est modifiée.

Une première solution consiste à transformer la classe `VueCatalogue` en une interface ou en une classe abstraite et à introduire deux sous-classes d'implantation différant par le choix de l'algorithme. Ceci présente l'inconvénient de complexifier inutilement la hiérarchie des vues de catalogue.

Une autre possibilité est d'implanter les deux algorithmes dans la classe `VueCatalogue` et à l'aide d'instructions conditionnelles d'effectuer les choix. Mais cela consiste à développer une classe relativement lourde et dont le code des méthodes est difficile à appréhender.

Le pattern `Strategy` propose une autre solution en introduisant une classe par algorithme. L'ensemble des classes ainsi créées possède une interface commune qui est utilisée pour dialoguer avec la classe `VueCatalogue`. La figure 26.1 montre le diagramme des classes de l'application du pattern `Strategy`.

Ce diagramme montre les deux classes d'algorithmes : `DessinUnVéhiculeLigne` et `DessinTroisVéhiculesLigne` implantant l'interface `DessinCatalogue`. Les notes détaillant les deux méthodes de la classe `VueCatalogue` montrent comment ces deux classes sont utilisées.

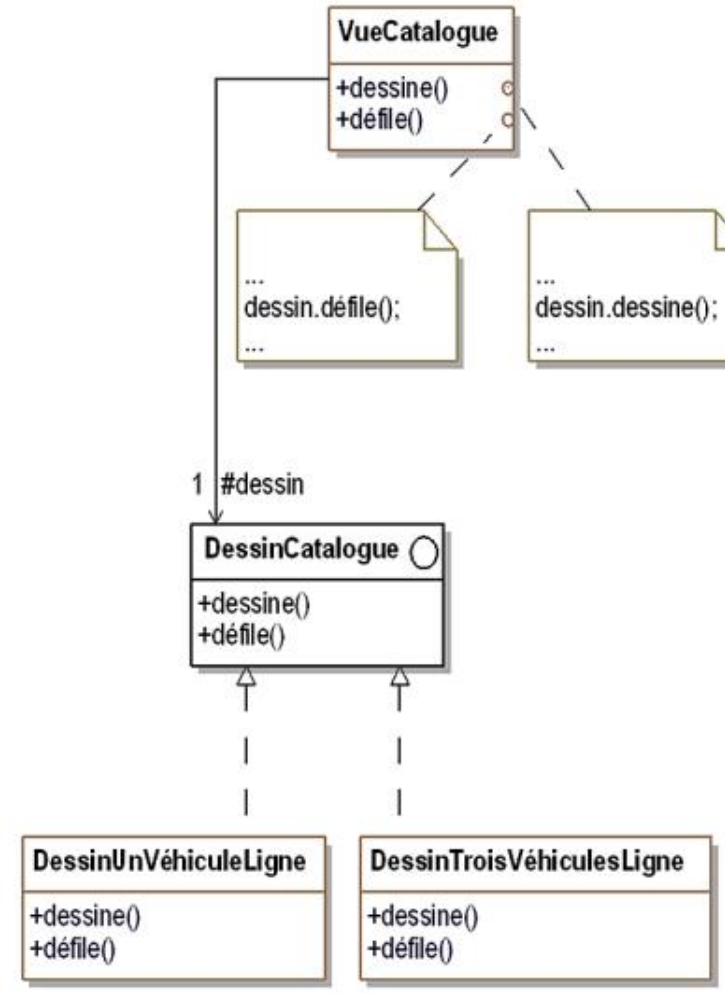


Figure 26.1 - Application du pattern `Strategy` pour le dessin de catalogues de véhicules

Structure

1. Diagramme de classes

La figure 26.2 montre la structure générique du pattern.

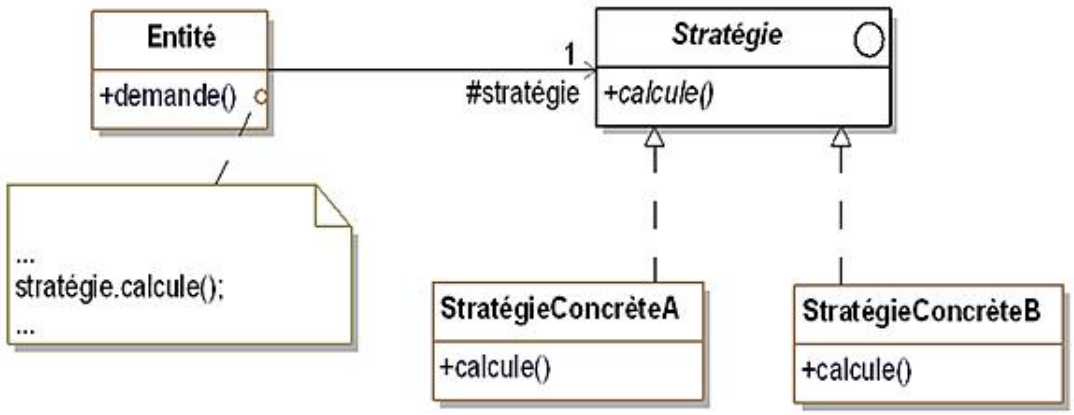


Figure 26.2 - Structure du pattern `Strategy`

2. Participants

Les participants au pattern sont les suivants :

- `Stratégie (DessinCatalogue)`est l'interface commune à tous les algorithmes. Cette interface est utilisée par `Entité` pour invoquer l'algorithme ;
- `StratégieConcrèteA`et `StratégieConcrèteB` (`DessinUnVéhiculeLigne` et `DessinTroisVéhiculesLigne`) sont les sous-classes concrètes qui implantent les différents algorithmes ;
- `Entité`est la classe utilisant un des algorithmes des classes d'implantation de `Stratégie`. En conséquence, elle possède une référence vers une instance de l'une de ces classes. Enfin, si nécessaire, elle peut exposer ses données internes aux classes d'implantation.

3. Collaborations

L'entité et les instances des classes d'implantation de `Stratégie` interagissent pour implanter les algorithmes. Dans le cas le plus simple, les données nécessaires à l'algorithme sont transmises en paramètre. Si nécessaire, la classe `Entité` implante des méthodes pour donner accès à ses données internes.

Le client initialise l'entité avec une instance de la classe d'implantation de `Stratégie`. Il choisit lui-même cette classe et, en général, ne la modifie plus par la suite. L'entité peut modifier ensuite ce choix.

L'entité redirige les requêtes provenant de ses clients vers l'instance référencée par son attribut `stratégie`.

Domaines d’application

Le pattern est utilisé dans les cas suivants :

- le comportement d’une classe peut être implanté par différents algorithmes dont certains sont plus efficaces en temps d’exécution ou en consommation mémoire ou encore contiennent des mécanismes de mise au point ;
- l’implantation du choix de l’algorithme par des instructions conditionnelles est trop complexe ;
- un système possède de nombreuses classes identiques à l’exception d’une partie de leur comportement.

Dans le dernier cas, le pattern `Strategy` permet de regrouper ces classes en une seule, ce qui simplifie l’interface pour les clients.

Les problèmes liés à la création d'objets

1. Problématique

Dans la plupart des langages à objets, la création d'objets se fait grâce au mécanisme d'instanciation qui consiste à créer un nouvel objet par appel de l'opérateur `new` paramétré par une classe (et éventuellement des arguments du constructeur de la classe dont le but est de donner aux attributs leur valeur initiale). Un tel objet est donc une instance de cette classe.

Les langages les plus utilisés aujourd'hui comme Java, C++ ou C# utilisent le mécanisme de l'opérateur `new`.

En Java, une instruction de création d'un objet peut s'écrire ainsi :

```
objet = new Classe();
```

Dans certains cas, il est nécessaire de paramétrer la création d'objets. Prenons l'exemple d'une méthode `construitDoc` qui crée des documents. Elle peut construire des documents PDF, RTF ou HTML. Généralement le type du document à créer est transmis en paramètre à la méthode sous forme d'une chaîne de caractères, ce qui donne le code suivant :

```
public Document construitDoc(String typeDoc)
{
  Document resultat;

  if (typeDoc.equals("PDF"))
    resultat = new DocumentPDF();
  else if (typeDoc.equals("RTF"))
    resultat = new DocumentRTF();
  else if (typeDoc.equals("HTML"))
    resultat = new DocumentHTML();
  // suite de la methode
}
```

Cet exemple nous montre qu'il est difficile de paramétrer le mécanisme de création d'objets, la classe transmise en paramètre à l'opérateur `new` ne pouvant être substituée par une variable. L'utilisation d'instructions conditionnelles dans le code du client est souvent pratiquée avec l'inconvénient que chaque changement dans la hiérarchie des classes à instancier demande des modifications dans le code des clients. Dans notre exemple, il faut changer le code de la méthode `construitDoc` en cas d'ajout de nouvelles classes de document.

Néanmoins, certains langages offrent des mécanismes plus ou moins souples et souvent assez complexes pour créer des instances à partir du nom d'une classe contenu dans une variable de type `String`.

La difficulté est encore plus grande quand il faut construire des objets composés dont les composants peuvent être instanciés à partir de classes différentes. Par exemple, une liasse de documents peut être formée de documents PDF, RTF ou HTML. Le client doit alors connaître toutes les classes possibles des composants et des composés. Chaque modification dans ces ensembles de classes devient alors très lourde à gérer.

2. Les solutions proposées par les patterns de construction

Les patterns `Abstract Factory`, `Builder`, `Factory Method` et `Prototype` proposent une solution pour paramétrer la création d'objets. Dans le cas des patterns `Abstract Factory`, `Builder` et `Prototype`, un objet est utilisé comme paramètre du système. Cet objet est chargé de l'instanciation des classes. Ainsi toute modification dans la hiérarchie des classes n'entraîne que des changements dans la modification de cet objet.

Le pattern `Factory Method` propose un paramétrage basé sur les sous-classes de la classe cliente. Ses sous-classes implantent la création des objets. Tout changement dans la hiérarchie des classes entraîne par conséquent une modification de la hiérarchie des sous-classes de la classe cliente.

Exemple en Java

Notre exemple en Java est basé sur l'affichage du catalogue de véhicules, simulé ici simplement avec des sorties à l'écran.

L'interface `DessinCatalogue` introduit la méthode dessine qui prend en paramètre une liste d'instances de `VueVehicule`.

```
import java.util.*;
public interface DessinCatalogue
{
  void dessine(List<VueVehicule> contenu);
}
```

La classe `DessinUnVehiculeLigne` implante la méthode `dessine` en affichant chaque véhicule sur une ligne (impression d'un saut de ligne après l'affichage d'un véhicule).

```
import java.util.*;
public class DessinUnVehiculeLigne implements
  DessinCatalogue
{

  public void dessine(List<VueVehicule> contenu)
  {
    System.out.println(
      "Dessine les véhicules avec un véhicule par ligne");
    for (VueVehicule vueVehicule: contenu)
    {
      vueVehicule.dessine();
      System.out.println();
    }
    System.out.println();
  }
}
```

La classe `DessinTroisVehiculesLigne` implante la méthode `dessine` en affichant trois véhicule par ligne (impression d'un saut de ligne après l'affichage de trois véhicules).

```
import java.util.*;
public class DessinTroisVehiculesLigne implements
  DessinCatalogue
{

  public void dessine(List<VueVehicule> contenu)
  {
    int compteur;
    System.out.println(
      "Dessine les véhicules avec trois véhicules par ligne");
    compteur = 0;
    for (VueVehicule vueVehicule: contenu)
    {
      vueVehicule.dessine();
      compteur++;
      if (compteur == 3)
      {
        System.out.println();
        compteur = 0;
      }
      else
        System.out.print(" ");
    }
    if (compteur != 0)
      System.out.println();
    System.out.println();
  }
}
```

La classe `VueVehicule` qui dessine un véhicule possède le code suivant.

Il convient de noter que la méthode `dessine` de `VueVehicule`, dans le cas de cette simulation, est identique pour un affichage sur une ligne et sur trois lignes, ce qui n'est pas le cas dans la réalité d'une interface graphique.

```
public class VueVehicule
{
  protected String description;

  public VueVehicule(String description)
  {
    this.description = description;
  }

  public void dessine()
  {
    System.out.print(description);
  }
}
```

La classe `VueCatalogue` possède un constructeur qui prend comme paramètre une instance de l'une des classes d'implantation de `DessinCatalogue`, instance qui est mémorisée dans l'attribut `dessin`. Ce constructeur initialise également le contenu qui devrait être normalement lu depuis une base de données.

La méthode `dessine` redirige l'appel vers l'instance mémorisée dans `dessin`. Elle transmet en paramètre une référence vers le contenu du catalogue.

```
import java.util.*;
public class VueCatalogue
{
  protected List<VueVehicule> contenu =
    new ArrayList<VueVehicule>();
  protected DessinCatalogue dessin;

  public VueCatalogue(DessinCatalogue dessin)
  {
    contenu.add(new VueVehicule("véhicule bon marché"));
    contenu.add(new VueVehicule("véhicule spacieux"));
    contenu.add(new VueVehicule("véhicule rapide"));
    contenu.add(new VueVehicule("véhicule confortable"));
    contenu.add(new VueVehicule("véhicule sportif"));
    this.dessin = dessin;
  }

  public void dessine()
  {
    dessin.dessine(contenu);
  }
}
```

Enfin, le programme principal est implanté par la classe `Utilisateur`. Celui-ci crée deux instances de `VueCatalogue`, la première est paramétrée par le dessin sur trois lignes. La seconde instance est paramétrée par le dessin sur une ligne. Après les avoir créés, le programme principal invoque la méthode `dessine` de ses instances.

```
public class Utilisateur
{

  public static void main(String[] args)
  {
    VueCatalogue vueCatalogue1 = new VueCatalogue(new
      DessinTroisVehiculesLigne());
    vueCatalogue1.dessine();
    VueCatalogue vueCatalogue2 = new VueCatalogue(new
      DessinUnVehiculeLigne());
    vueCatalogue2.dessine();
  }
}
```

L'exécution de ce programme produit le résultat suivant, ce qui montre bien que le comportement de la méthode

`dessine` est paramétré par l'instance transmise en paramètre au constructeur.

```
Dessine les véhicules avec trois véhicules par ligne
véhicule bon marché véhicule spacieux véhicule rapide
véhicule confortable véhicule sportif

Dessine les véhicules avec un véhicule par ligne
véhicule bon marché
véhicule spacieux
véhicule rapide
véhicule confortable
véhicule sportif
```

Description

Le pattern `Template Method` permet de reporter dans des sous-classes certaines étapes de l'une des opérations d'un objet, ces étapes étant alors décrites dans les sous-classes.

Exemple

Au sein du système de vente en ligne de véhicules, nous gérons des commandes issues de clients en France et au Luxembourg. La différence entre ces deux commandes concerne notamment le calcul de la TVA. Si en France, le taux de TVA est toujours de 19,6%, il est variable au Luxembourg (12% pour la partie des prestations, 15% pour le matériel). Le calcul de la TVA demande deux opérations de calcul distinctes en fonction du pays.

Une première solution consiste à implanter deux classes distinctes sans surclasse commune : `CommandeFrance` et `CommandeLuxembourg`. Cette solution présente l'inconvénient majeur d'avoir du code identique mais qui n'a pas été factorisé comme l'affichage des informations de la commande (méthode `affiche`).

Une classe abstraite `Commande` peut être introduite pour factoriser les méthodes communes comme la méthode `affiche`.

Le pattern `Template Method` propose d'aller plus loin en proposant de factoriser du code commun au sein des méthodes. Nous prenons l'exemple de la méthode `calculeMontantTtc` dont l'algorithme est le suivant pour la France (en pseudo-code).

```
calculeMontantTtc :
montantTva = montantHt * 0,196 ;
montantTtc = montantHt + montantTva ;
```

L'algorithme pour le Luxembourg est donné par le pseudo-code suivant.

```
calculeMontantTtc :
montantTva = (montantPrestationHt * 0,12) +
(montantMatérielHt * 0,15) ;
montantTtc = montantHt + montantTva ;
```

Nous voyons sur cet exemple que la dernière ligne de la méthode est commune aux deux pays (dans cet exemple, il n'y a qu'une ligne mais dans un cas réel, la partie commune est plus importante).

Nous remplaçons la première ligne par un appel d'une nouvelle méthode appelée `calculeTva`. Ainsi la méthode `calculeMontantTtc` est décrite dorénavant ainsi :

```
calculeMontantTtc :
calculeTva() ;
montantTtc = montantHt + montantTva ;
```

La méthode `calculeMontantTtc` peut maintenant être factorisée. Le code spécifique a été déplacé dans la méthode `calculeTva` dont l'implantation reste spécifique à chaque pays. La méthode `calculeTva` est introduite dans la classe `Commande` en tant que méthode abstraite.

La méthode `calculeMontantTtc` est appelée une méthode "patron" (template method). Une méthode "patron" introduit la partie commune d'un algorithme qui est ensuite complétée par des parties spécifiques.

Cette solution est illustrée par le diagramme de classes de la figure 27.1 où, pour des raisons de simplification, le calcul de la TVA luxembourgeoise a été ramené à un taux unique de 15%.

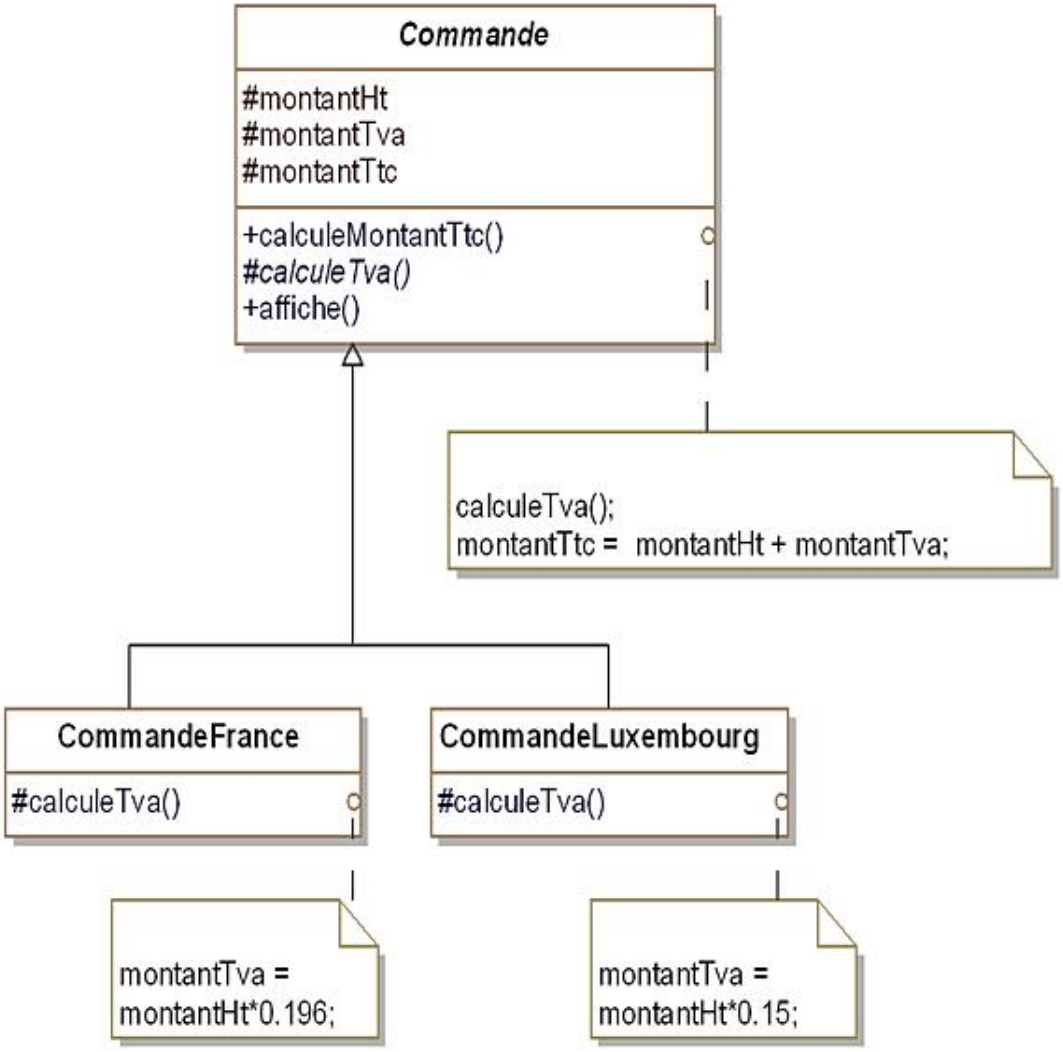


Figure 27.1 - Application du pattern `Template Method` pour le calcul de la TVA d'une commande en fonction du pays

Lorsqu'un client appelle la méthode `calculeMontantTtc` d'une commande, celle-ci invoque la méthode `calculeTva`. L'implantation de cette méthode dépend de la classe concrète de la commande :

- si cette classe est `CommandeFrance`, le diagramme de séquence est décrit à la figure 27.2. La TVA est calculée avec le taux de 19,6% ;
- si cette classe est `CommandeLuxembourg`, le diagramme de séquence est décrit à la figure 27.3. La TVA est calculée avec le taux de 15%.

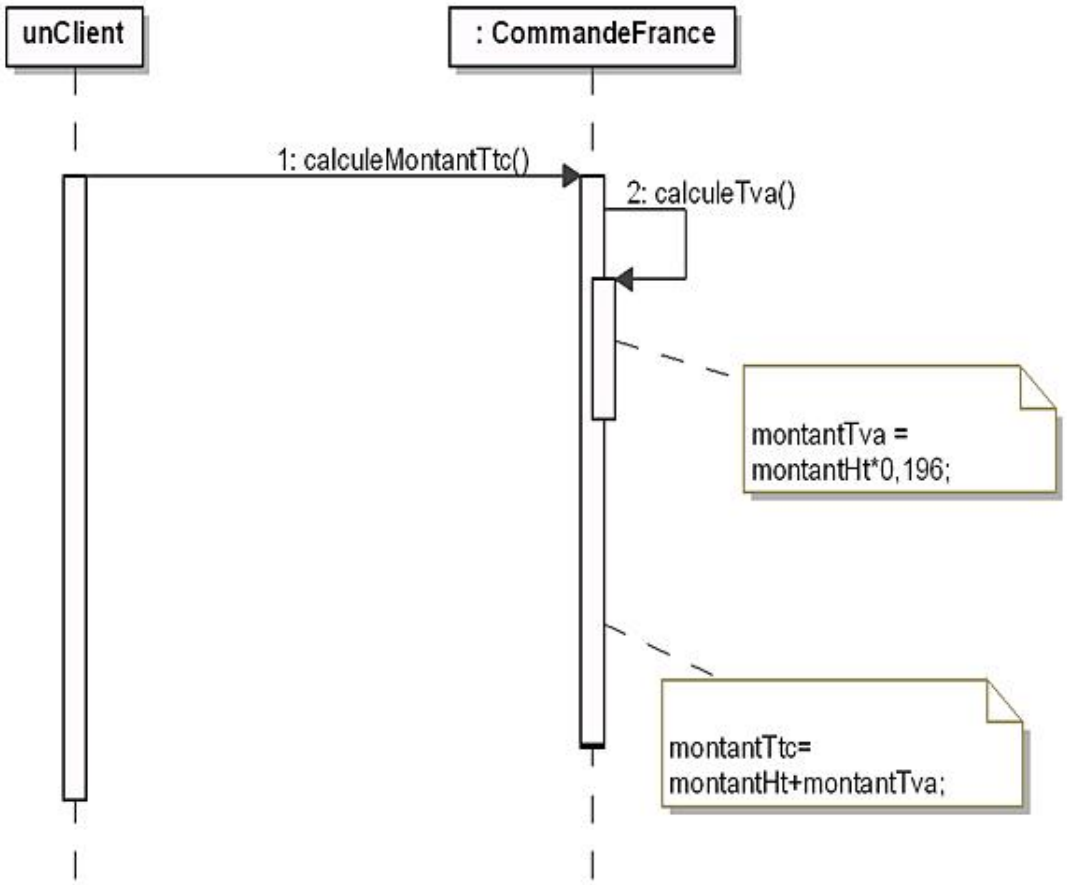


Figure 27.2 - Diagramme de séquence du calcul du montant TTC d'une commande française

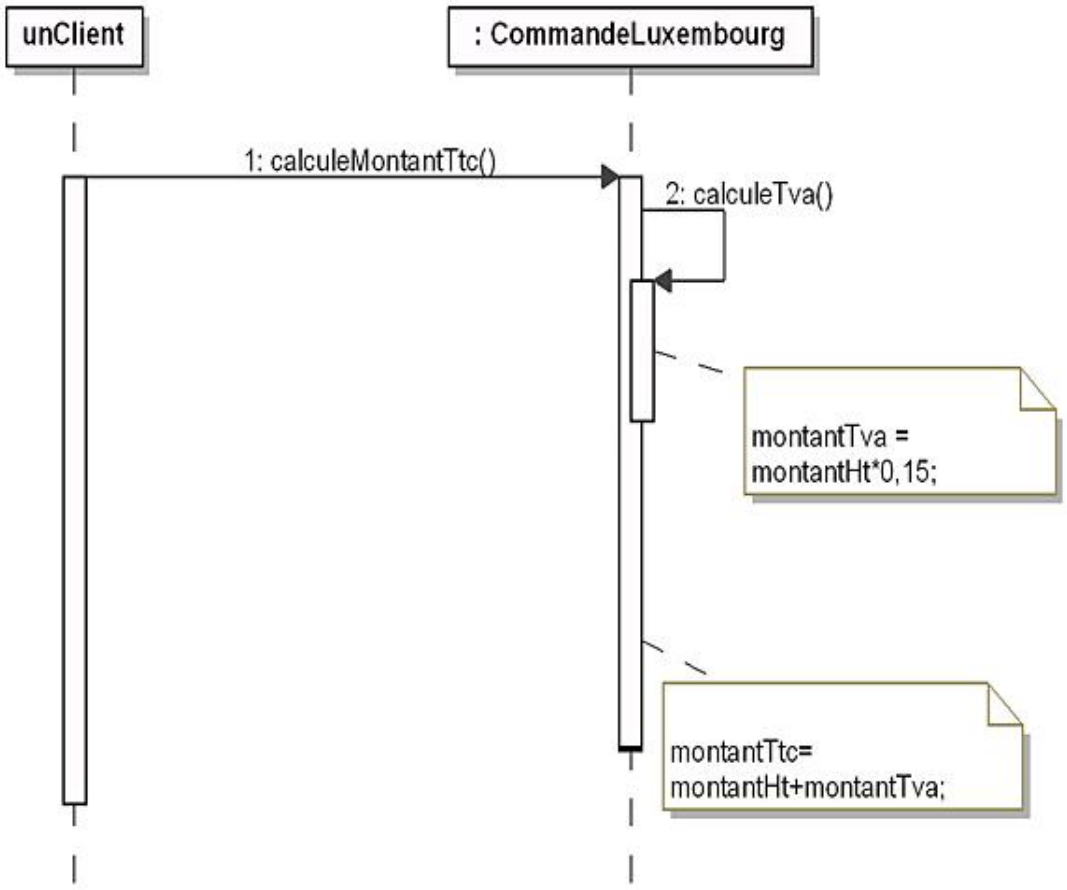


Figure 27.3 - Diagramme de séquence du calcul du montant TTC d'une commande luxembourgeoise

Structure

1. Diagramme de classes

La figure 27.4 montre la structure générique du pattern.

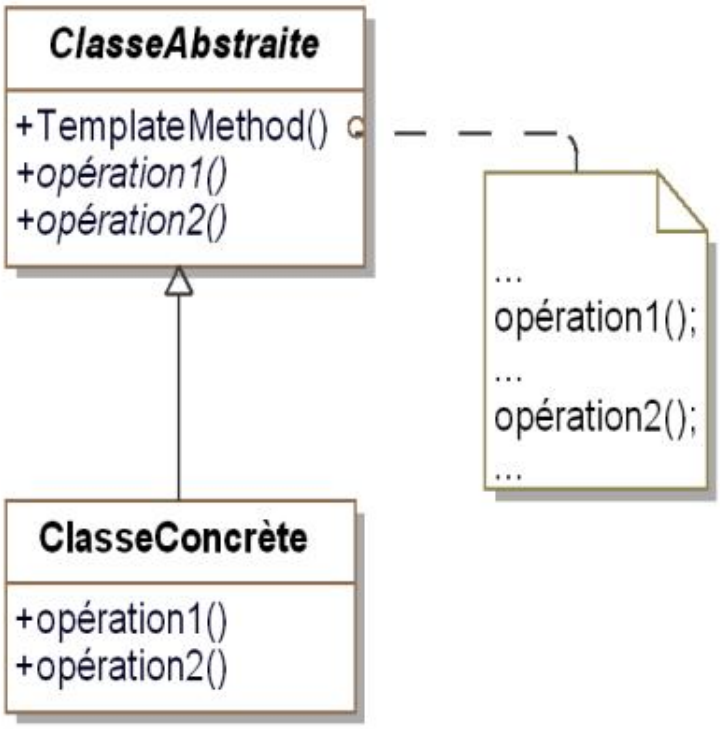


Figure 27.4 - Structure du pattern `Template Method`

2. Participants

Les participants au pattern sont les suivants :

- La classe abstraite `ClasseAbstraite (Commande)`introduit la méthode "patron" ainsi que la signature des méthodes abstraites que cette méthode invoque.
- La sous-classe concrète `ClasseConcrète (CommandeFrance` et `CommandeLuxembourg)` implante les méthodes abstraites utilisées par la méthode "patron" de la classe abstraite. Il peut y avoir plusieurs classes concrètes.

3. Collaborations

L'implantation de l'algorithme est réalisée par la collaboration entre la méthode "patron" de la classe abstraite et les méthodes d'une sous-classe concrète qui complètent l'algorithme.

Domaines d’application

Le pattern est utilisé dans les cas suivants :

- une classe partage avec une autre ou plusieurs autres classes du code identique qui peut être factorisé après que le ou les parties spécifiques à chaque classe aient été déplacées dans de nouvelles méthodes ;
- un algorithme possède une partie invariable et des parties spécifiques à différents types d’objets.

Exemple en Java

La classe abstraite `Commande` introduit la méthode "patron" `calculeMontantTtc` qui invoque la méthode abstraite `calculeTva`.

```
public abstract class Commande
{
  protected double montantHt;
  protected double montantTva;
  protected double montantTtc;

  protected abstract void calculeTva();

  public void calculeMontantTtc()
  {
    this.calculeTva();
    montantTtc = montantHt + montantTva;
  }

  public void setMontantHt(double montantHt)
  {
    this.montantHt = montantHt;
  }

  public void affiche()
  {
    System.out.println("Commande");
    System.out.println("Montant HT " + montantHt);
    System.out.println("Montant TTC " + montantTtc);
  }
}
```

La sous-classe concrète `CommandeFrance` implante la méthode `calculeTva` avec le taux de TVA français.

```
public class CommandeFrance extends Commande
{
  protected void calculeTva()
  {
    montantTva = montantHt * 0.196;
  }
}
```

La sous-classe concrète `CommandeLuxembourg` implante la méthode `calculeTva` avec le taux de TVA luxembourgeois.

```
public class CommandeLuxembourg extends Commande
{
  protected void calculeTva()
  {
    montantTva = montantHt * 0.15;
  }
}
```

Enfin la classe `Utilisateur` contient le programme principal. Celui-ci crée une commande française, en fixe le montant HT, calcule le montant TTC puis affiche la commande. Ensuite, le programme principal fait la même chose avec une commande luxembourgeoise.

```
public class Utilisateur
{

  public static void main(String[] args)
  {
    Commande commandeFrance = new CommandeFrance();
    commandeFrance.setMontantHt(10000);
    commandeFrance.calculeMontantTtc();
    commandeFrance.affiche();
```

```
    Commande commandeLuxembourg = new CommandeLuxembourg();
    commandeLuxembourg.setMontantHt(10000);
    commandeLuxembourg.calculeMontantTtc();
    commandeLuxembourg.affiche();
  }

}
```

L’exécution du programme fournit le résultat suivant.

```
Commande
Montant HT 10000.0
Montant TTC 11960.0
Commande
Montant HT 10000.0
Montant TTC 11500.0
```

Description

Le pattern `Visitor` construit une opération à réaliser sur les éléments d'un ensemble d'objets. De nouvelles opérations peuvent ainsi être ajoutées sans modifier les classes de ces objets.

Exemple

Considérons la figure 28.1 qui décrit les clients de notre système organisés sous la forme d'objets composés selon le pattern `Composite`. À l'exception de la méthode `ajouteFililale` spécifique à la gestion de la composition, les deux sous-classes possèdent deux méthodes de même nom : `calculeCoûtEntretien` et `envoieEmailCommercial`. Chacune de ces méthodes correspond à une même fonctionnalité mais dont l'implantation est bien sûr adaptée en fonction de la classe. De nombreuses autres fonctionnalités pourraient également être implantées comme par exemple, le calcul du chiffre d'affaires d'un client (filiales incluses ou non), etc.

Sur le diagramme, le calcul du coût de l'entretien n'est pas détaillé. Le détail se trouve dans le chapitre consacré au pattern `Composite`.

Cette approche est utilisable tant que le nombre de fonctionnalités reste faible. En revanche, si celui-ci devient important, nous obtenons alors des classes contenant beaucoup de méthodes, difficiles à appréhender et à maintenir. De surcroît, ces fonctionnalités donnent lieu à des méthodes (`calculeCoûtEntretien` et `envoieEmailCommercial`) sans lien entre elles et sans lien avec le cœur des objets à la différence, par exemple, de la méthode `ajouteFililale` qui contribue à composer les objets.

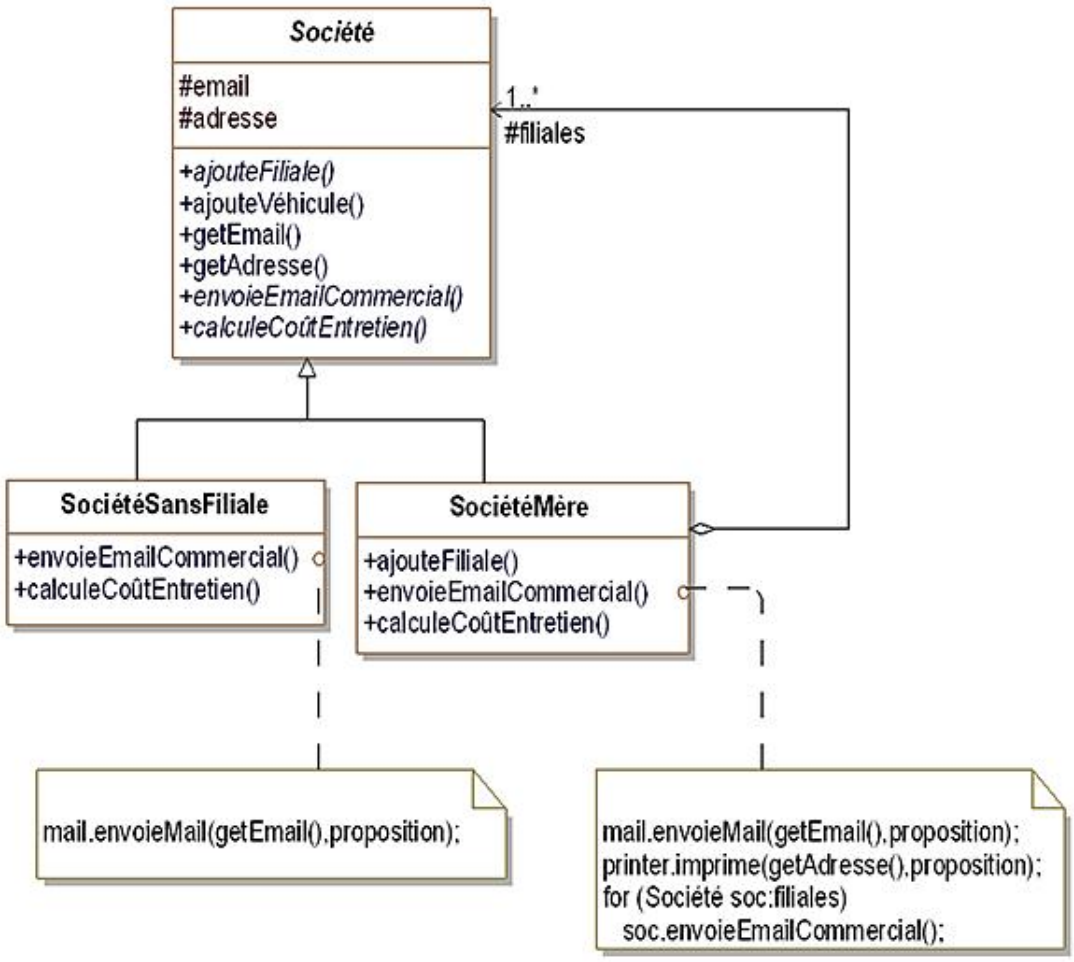


Figure 28.1 - Fonctionnalités multiples au sein d'objets composés

Le pattern `Visitor` propose d'implanter les nouvelles fonctionnalités dans un objet séparé appelé visiteur. Chaque visiteur établit une fonctionnalité pour plusieurs classes en introduisant pour chacune de ces classes une méthode d'implantation dont le nom respecte une convention de nommage unique, à savoir `visite` suivi du nom de la classe.

Ensuite, le visiteur est transmis à la méthode `accepteVisiteur` de ces classes. Cette méthode appelle la méthode du visiteur correspondant à sa classe. Ainsi quel que soit le nombre de fonctionnalités à implanter dans un ensemble de classes, seule la méthode `accepteVisiteur` doit être écrite. Il peut être nécessaire de donner la possibilité au visiteur d'accéder à la structure interne de l'objet visité (de préférence par des accesseurs en lecture comme ici les méthodes `getEmail` et `getAdresse`).

Si les objets sont composés alors leur méthode `accepteVisiteur` appelle la méthode `accepteVisiteur` de leurs

composants. C'est le cas ici pour chaque instance de la classe `SociétéMère` qui appelle la méthode `accepteVisiteur` de ses filiales.

Le diagramme de classes de la figure 28.2 illustre la mise en œuvre du pattern `Visitor`. L'interface `Visiteur` introduit la signature des méthodes implantant les fonctionnalités pour chaque classe à visiter. Cette interface possède deux sous-classes d'implantation, une par fonctionnalité.

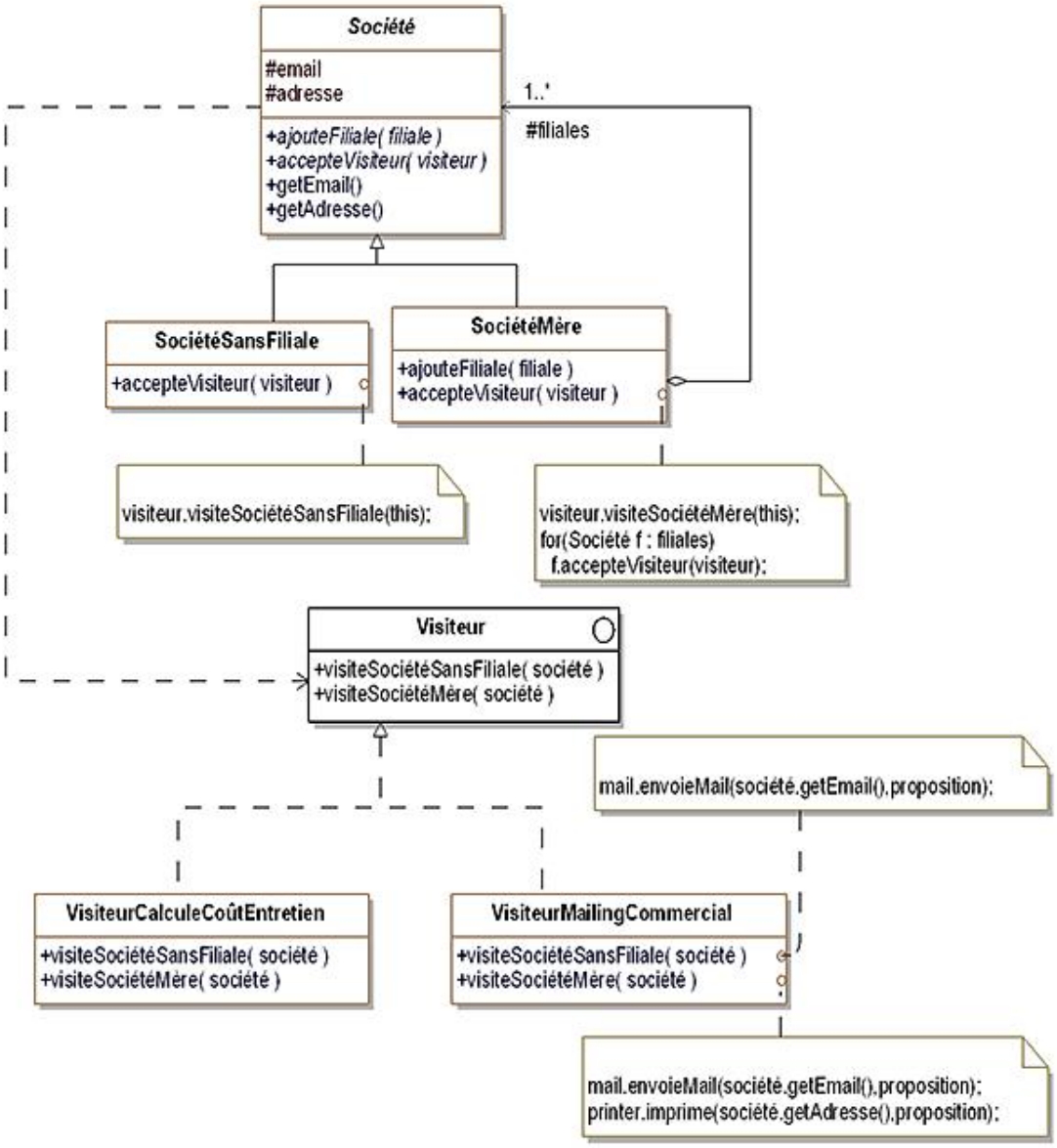


Figure 28.2 - Application du pattern `Visitor` pour l'ajout d'une fonctionnalité de mailing

Structure

1. Diagramme de classes

La figure 28.3 détaille la structure générique du pattern.

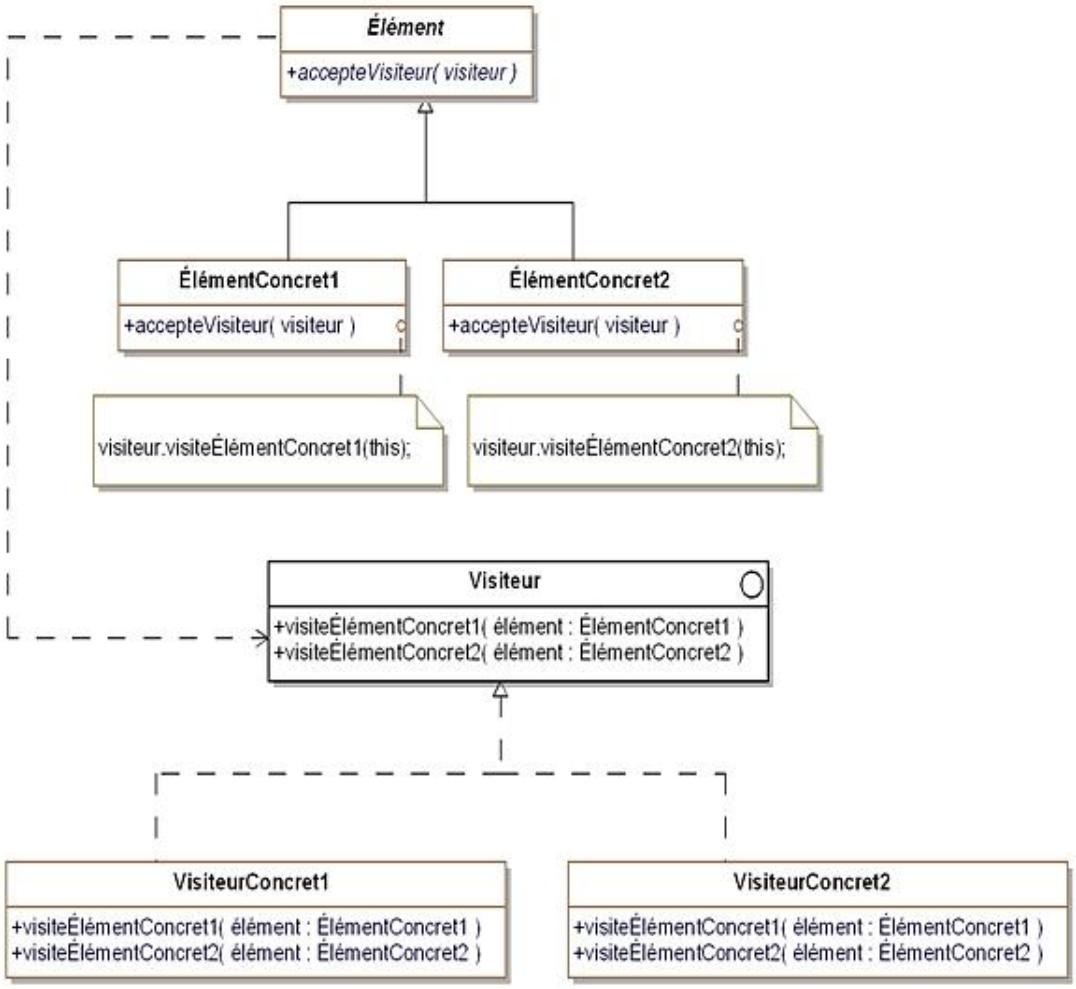


Figure 28.3 - Structure du pattern `Visitor`

2. Participants

Les participants au pattern sont les suivants :

- `Visiteur` est l'interface qui introduit la signature des méthodes qui réalisent une fonctionnalité au sein d'un ensemble de classes. Il existe une méthode par classe qui reçoit comme argument une instance de cette classe.

- `VisiteurConcret1` et `VisiteurConcret2` (`VisiteurCalculeCoûtEntretien` et `VisiteurMailingCommercial`) implantent les méthodes qui réalisent la fonctionnalité correspondant à la classe. Cette fonctionnalité est distribuée dans les différents éléments.

- `Élément` (`Société`) est une classe abstraite surclasse des classes d'éléments. Elle introduit la méthode abstraite `accepteVisiteur` qui accepte un visiteur comme argument.

- `ÉlémentConcret1` et `ÉlémentConcret2` (`SociétéSansFiliale` et `SociétéMère`) implantent la méthode

`accepteVisiteur` qui consiste à rappeler le visiteur au travers de la méthode correspondant à la classe.

3. Collaborations

Un client qui utilise un visiteur doit d'abord le créer comme instance de la classe de son choix puis le transmettre comme argument de la méthode `accepteVisiteur` d'un ensemble d'éléments.

L'élément rappelle la méthode du visiteur qui correspond à sa classe. Il transmet une référence vers lui-même comme argument afin que le visiteur puisse accéder à sa structure interne.

Domaines d’application

Le pattern est utilisé dans les cas suivants :

- de nombreuses fonctionnalités doivent être ajoutées à un ensemble de classes sans que ces ajouts viennent alourdir ces classes ;
- un ensemble de classes possèdent une structure fixe et il est nécessaire de leur adjoindre des fonctionnalités sans modifier leur interface.

Si la structure de l’ensemble des classes auxquelles il est nécessaire d’adjoindre des fonctionnalités change souvent, le pattern `Visitor` n’est pas adapté. En effet, toute modification de la structure implique une modification de chaque visiteur, ce qui peut coûter cher.

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.

Exemple en Java

Nous reprenons l'exemple de la figure 28.2. La classe `Societe` est décrite en Java comme suit. La méthode `accepteVisiteur` est abstraite car son code dépend de la sous-classe.

```
public abstract class Societe
{
  protected String nom, email, adresse;

  public Societe(String nom, String email, String adresse)
  {
    this.nom = nom;
    this.email = email;
    this.adresse = adresse;
  }

  public String getNom()
  {
    return nom;
  }

  public String getEmail()
  {
    return email;
  }

  public String getAdresse()
  {
    return adresse;
  }

  public abstract boolean ajouteFiliale(Societe filiale);

  public abstract void accepteVisiteur(Visiteur visiteur);

}
```

Le code source de la sous-classe `SocieteSansFiliale` est le suivant. La méthode `accepteVisiteur` rappelle la méthode `VisiteSocieteSansFililale` du visiteur.

```
public class SocieteSansFiliale extends Societe
{

  public SocieteSansFiliale(String nom, String email,
    String adresse)
  {
    super(nom, email, adresse);
  }

  public void accepteVisiteur(Visiteur visiteur)
  {
    visiteur.VisiteSocieteSansFiliale(this);
  }

  public boolean ajouteFiliale(Societe filiale)
  {
    return false;
  }

}
```

Le code source de la sous-classe `SocieteMere` est le suivant. La méthode `accepteVisiteur` rappelle la méthode `VisiteSocieteMere` du visiteur puis elle invoque la méthode `accepteVisiteur` de ses filiales.

```
import java.util.ArrayList;
import java.util.List;
public class SocieteMere extends Societe
```

```
{
  protected List<Societe> filiales =
    new ArrayList<Societe>();

  public SocieteMere(String nom, String email, String
    adresse)
  {
    super(nom, email, adresse);
  }

  public void accepteVisiteur(Visiteur visiteur)
  {
    visiteur.VisiteSocieteMere(this);
    for (Societe filiale: filiales)
      filiale.accepteVisiteur(visiteur);
  }

  public boolean ajouteFiliale(Societe filiale)
  {
    return filiales.add(filiale);
  }
}
```

L'interface `Visiteur` introduit la signature des deux méthodes, une par classe devant être visitée.

```
public interface Visiteur
{
  void VisiteSocieteSansFiliale(SocieteSansFiliale
    societe);
  void VisiteSocieteMere(SocieteMere societe);
}
```

La classe `VisiteurMailingCommercial` envoie les mailings aux sociétés en implantant l'interface `Visiteur`. Les sociétés possédant des filiales reçoivent une proposition particulière et de surcroît par courrier. Ici le tout est simulé par des impressions à l'écran.

```
public class VisiteurMailingCommercial implements Visiteur
{
  public void VisiteSocieteSansFiliale(SocieteSansFiliale
    societe)
  {
    System.out.println("Envoi d'un email à " +
      societe.getNom() + " adresse : " + societe.getEmail
      () + " Proposition commerciale pour votre société");
  }

  public void VisiteSocieteMere(SocieteMere societe)
  {
    System.out.println("Envoi d'un email à " +
      societe.getNom() + " adresse : " + societe.getEmail
      () + " Proposition commerciale pour votre groupe");
    System.out.println("Envoi d'un courrier à " +
      societe.getNom() + " adresse : " +
      societe.getAdresse() +
      " Proposition commerciale pour votre groupe");
  }
}
```

Enfin la classe `Utilisateur` crée un groupe (groupe 2) constitué de la société 3 et du groupe 1, lui-même constitué de la société 1 et de la société 2.

Elle procède ensuite à l'envoi du mailing à toutes les sociétés du groupe 2 en invoquant sa méthode `accepteVisiteur` avec un visiteur, instance de la classe `VisiteurMailingCommercial`.

```
public class Utilisateur
{

  public static void main(String[] args)
  {
```

```
    Societe societe1 = new SocieteSansFiliale("société1",
      "info@societe1.com", "rue de la société 1");
    Societe societe2 = new SocieteSansFiliale("société2",
      "info@societe2.com", "rue de la société 2");
    Societe groupe1 = new SocieteMere("groupe1",
      "info@groupe1.com", "rue du groupe 1");
    groupe1.ajouteFiliale(societe1);
    groupe1.ajouteFiliale(societe2);
    Societe societe3 = new SocieteSansFiliale("société3",
      "info@societe3.com", "rue de la société 3");
    Societe groupe2 = new SocieteMere("groupe2",
      "info@groupe2.com", "rue du groupe 2");
    groupe2.ajouteFiliale(groupe1);
    groupe2.ajouteFiliale(societe3);
    groupe2.accepteVisiteur(new VisiteurMailingCommercial
      ());
  }
}
```

Le résultat de l'exécution est le suivant.

```
Envoi d'un email à groupe2 adresse : info@groupe2.com
Proposition commerciale pour votre groupe
Envoi d'un courrier à groupe2 adresse :
rue du groupe 2 Proposition commerciale pour votre groupe
Envoi d'un email à groupe1 adresse : info@groupe1.com
Proposition commerciale pour votre groupe
Envoi d'un courrier à groupe1
adresse : rue du groupe 1 Proposition commerciale
pour votre groupe
Envoi d'un email à société1 adresse : info@societe1.com
Proposition commerciale pour votre société
Envoi d'un email à société2 adresse : info@societe2.com
Proposition commerciale pour votre société
Envoi d'un email à société3 adresse : info@societe3.com
Proposition commerciale pour votre société
```

Préliminaire

Les vingt-trois patterns de conception introduits dans cet ouvrage ne constituent bien évidemment pas une liste exhaustive. Il est possible de créer de nouveaux patterns soit ex nihilo, soit en composant ou adaptant des patterns existants. Ces nouveaux patterns peuvent avoir une portée générale, à l'image de ceux introduits dans les chapitres précédents ou être spécifiques à un environnement de développement particulier. Nous pouvons citer ainsi le pattern de conception des JavaBeans ou les design patterns des EJB.

Dans ce chapitre, nous allons vous montrer trois nouveaux patterns obtenus par composition et variation de patterns existants. Ces trois patterns ont été décrits par John Vlissides, l'un des membres du GoF.

Le pattern Pluggable Factory

1. Introduction

Nous avons introduit dans un précédent chapitre le pattern `Abstract Factory` pour abstraire la création (instanciation) de produits de leurs différentes familles. Une fabrique est alors associée à chaque famille de produits. Sur le diagramme de la figure 29.1, deux produits sont exposés : les automobiles et les scooters, décrits chacun par une classe abstraite. Ces produits sont organisés en deux familles : traction essence et traction à l'électricité. Chacune de ces deux familles engendre une sous-classe concrète de chaque classe de produit.

Il existe donc deux fabriques pour les familles `FabriqueVéhiculeEssence` et `FabriqueVéhiculeÉlectricité`. Chaque fabrique permet de créer l'un des deux produits à l'aide des méthodes appropriées.

Ce pattern organise de façon très structurée la création d'objets. Chaque nouvelle famille de produits oblige à ajouter une nouvelle fabrique et donc une nouvelle classe.

À l'opposé, le pattern `Prototype` introduit dans le chapitre du même nom offre la possibilité de créer des nouveaux objets de façon très souple.

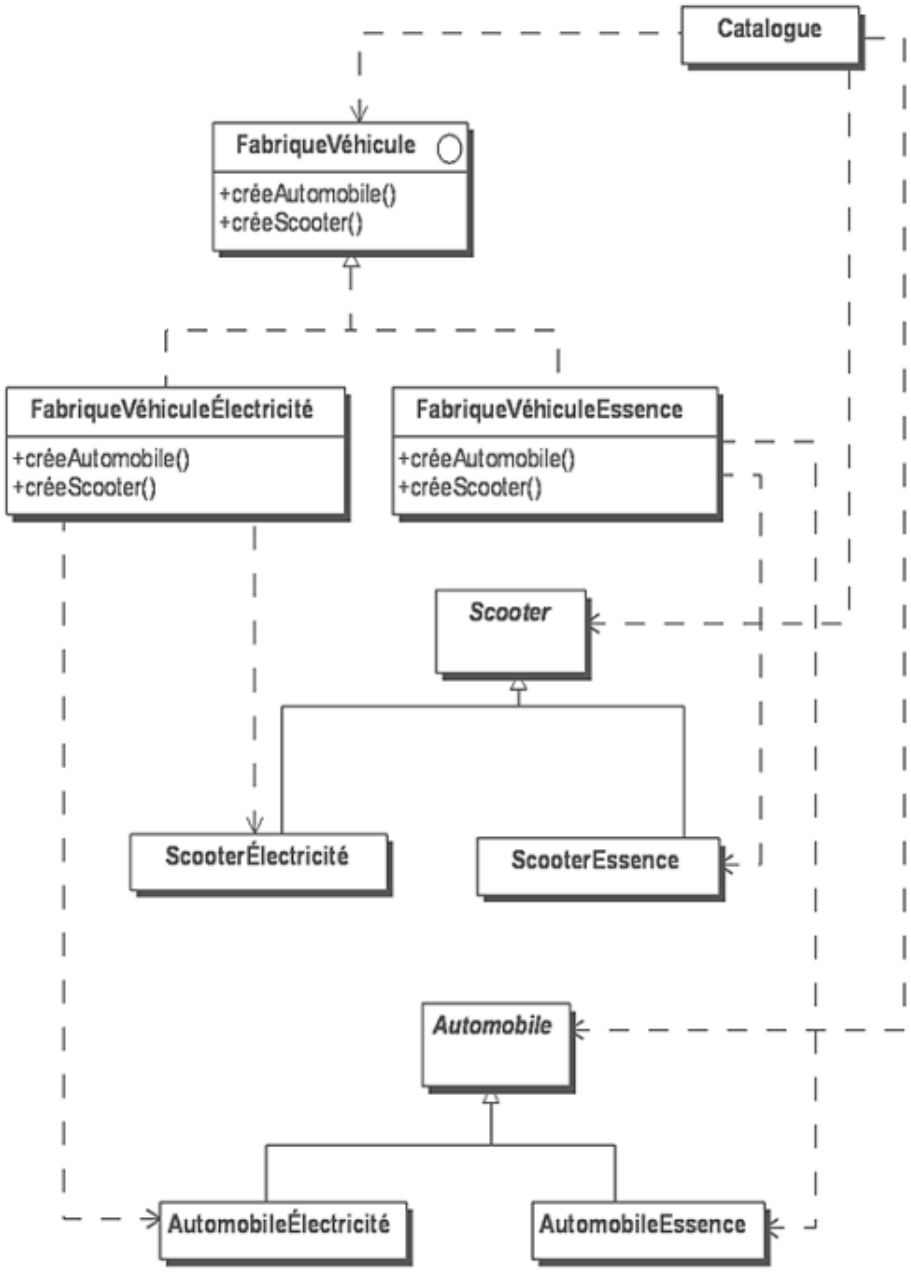


Figure 29.1 - Exemple d'utilisation du pattern `Abstract Factory`

La structure du pattern Prototype est décrite à la figure 29.2. Un objet initialisé afin d'être prêt à l'emploi et détenant la capacité de se dupliquer est appelé un prototype.

Le client dispose d'une liste de prototypes qu'il peut dupliquer lorsqu'il le désire. Cette liste est construite dynamiquement et peut être modifiée à tout moment lors de l'exécution. Le client peut ainsi construire de nouveaux objets sans connaître la hiérarchie des classes dont ils proviennent.

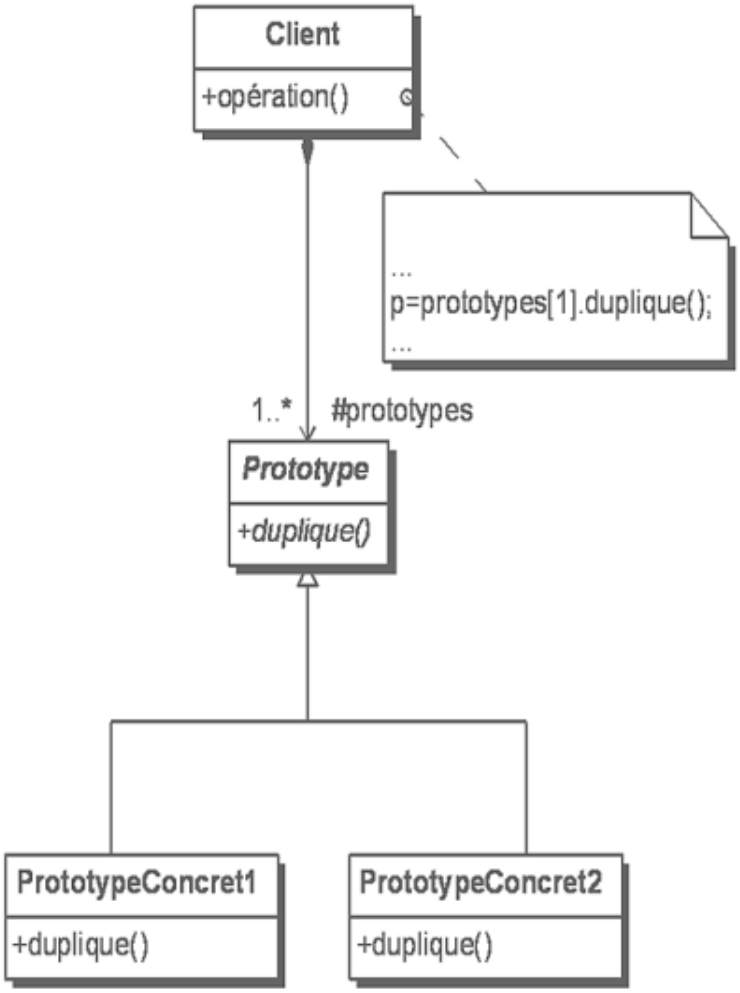


Figure 29.2 - Exemple d'utilisation du pattern Prototype

L'idée du pattern Pluggable Factory est de composer ces deux patterns pour conserver l'idée de création d'un produit par l'invocation d'une méthode de la fabrique et d'autre part, de pouvoir changer dynamiquement la famille à créer. Ainsi, la fabrique ne doit plus connaître les familles d'objets, le nombre de familles peut être différent pour chaque produit et enfin, il est possible de faire varier le produit à créer non plus uniquement par sa sous-classe (sa famille) mais aussi par des valeurs différentes de certains attributs. Nous reviendrons sur ce dernier point dans l'exemple Java.

La figure 29.3 reprend l'exemple du chapitre Le pattern Abstract Factory structuré cette fois à l'aide du pattern Pluggable Factory. La classe de fabriques d'objets FabriqueVéhicule n'est plus une interface comme à la figure 29.1 mais une classe concrète qui permet la création des objets et qui n'a donc plus besoin de sous-classes. Chaque fabrique possède un lien vers un prototype de chaque produit. De façon plus précise, il s'agit d'un lien vers une instance de l'une des sous-classes de la classe Automobile et d'un lien vers une instance de l'une des sous-classes de la classe Scooter.

C'est ici qu'intervient le pattern Prototype. Chaque produit devient un prototype. La classe abstraite qui introduit et décrit chaque famille de produits leur confère la capacité de clonage. Elle joue ainsi le rôle de la classe abstraite Prototype de la figure 29.2.

Les deux liens présents dans FabriqueVéhicule vers chaque prototype peuvent être modifiés dynamiquement à l'aide des méthodes setPrototypeAutomobile et setPrototypeScooter. La fabrique nécessite d'ailleurs que ces liens soient initialisés soit à l'aide de ces deux méthodes, soit par un autre moyen (comme le constructeur de la classe) pour pouvoir fonctionner. Le fonctionnement de la fabrique est réalisé par les deux méthodes créeAutomobile et créeScooter qui s'appuient sur la capacité de clonage des deux prototypes.

La classe `Test` représente le client de la fabrique et des classes de produits. Nous verrons son rôle dans l'exemple Java.

Le nom du pattern provient d'une part du fonctionnement de la fabrique de produits qui est dépendant des prototypes fournis et d'autre part de la possibilité de changer (« plugger ») dynamiquement ces prototypes.

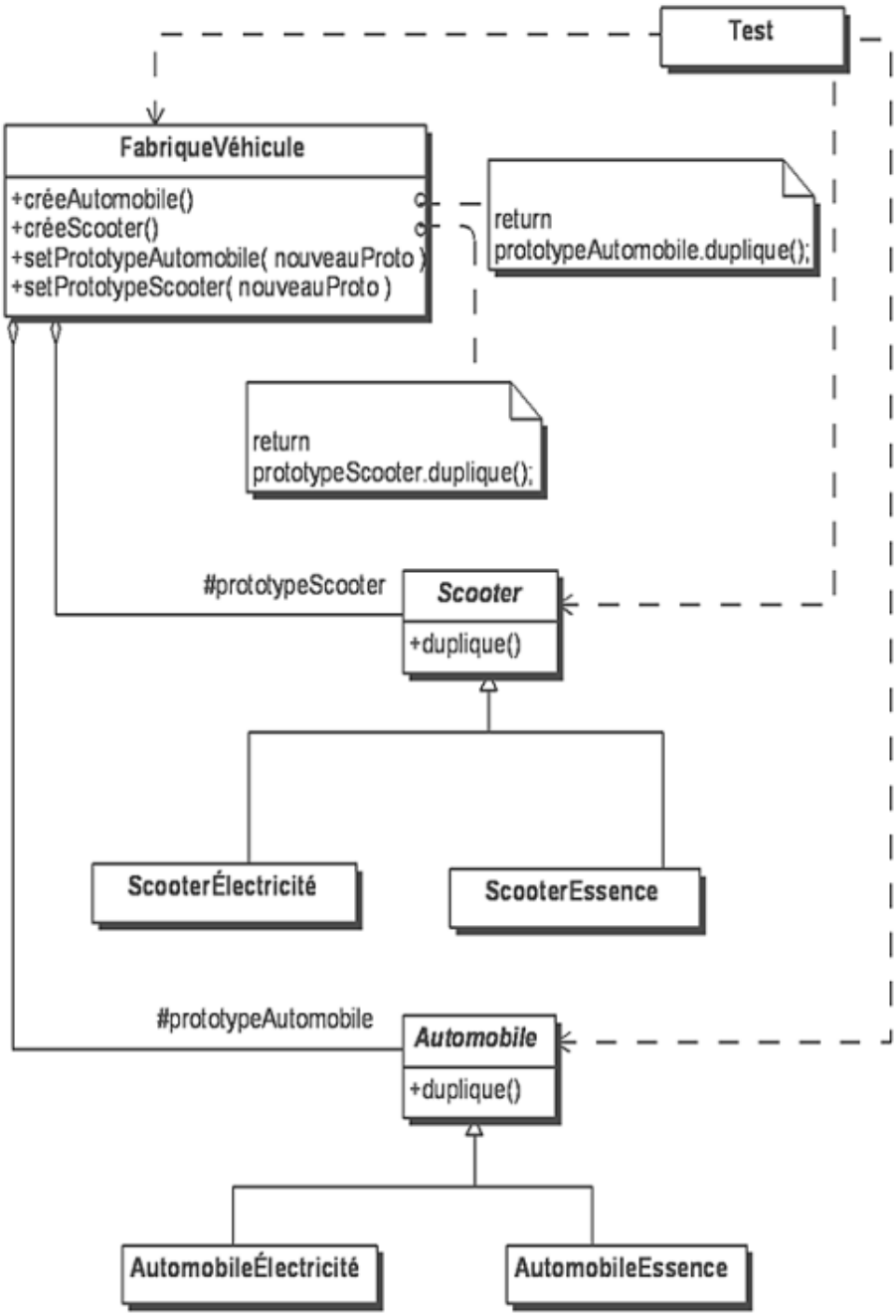


Figure 29.3 - Exemple d'utilisation du pattern `Pluggable Factory`

2. Structure

La figure 29.4 illustre la structure générique du pattern `Pluggable Factory`.

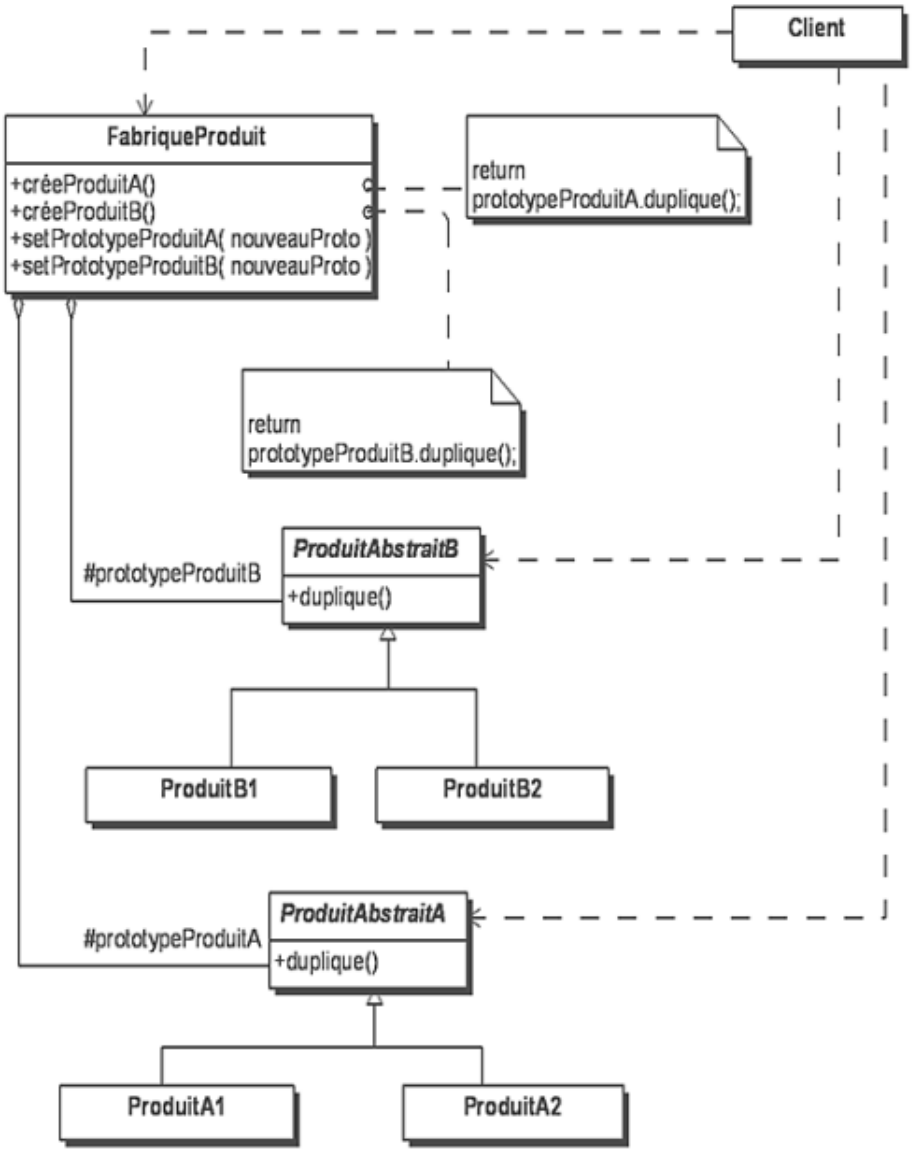


Figure 29.4 - Structure du pattern `Pluggable Factory`

Les participants du pattern sont les suivants :

- `FabriqueProduit` (`FabriqueVéhicule`) est la classe concrète qui détient les liens vers les prototypes de produit, offre les méthodes créant les différents produits ainsi que les méthodes permettant de fixer les prototypes.
- `ProduitAbstraitA` et `ProduitAbstraitB` (`Scooter` et `Automobile`) sont les classes abstraites des produits indépendamment de leur famille. Elles fournissent aux produits la capacité de clonage pour leur conférer le statut de prototype. Les familles sont introduites dans leurs sous-classes concrètes.
- `Client` est la classe qui utilise la classe `FabriqueProduit`.

La collaboration entre les objets est décrite à la suite :

- Le client crée ou obtient les prototypes de produits dont il a besoin.
- Le client crée une instance de la classe `FabriqueProduit` et lui fournit les prototypes nécessaires pour son fonctionnement.
- Il utilise ensuite cette instance pour créer ses produits au travers des méthodes de création. Il peut fournir de nouveaux prototypes à la fabrique.

À la différence du pattern Abstract Factory où il est conseillé de ne créer qu'une seule instance des fabriques concrètes (celles-ci ne possédant pas d'état), il est concevable ici de créer plusieurs instances de la fabrique de produits, chaque instance étant liée à des prototypes différents de produits.

3. Exemple en Java

Nous donnons maintenant le code Java de l'exemple correspondant au diagramme de classes de la figure 29.3. Nous introduisons d'abord les classes de produits (`Automobile` et `Scooter`) ainsi que leurs sous-classes. Il faut noter que ces classes introduisent maintenant des prototypes. Leur capacité de clonage est fournie à chaque fois par la méthode `duplique`. Des accesseurs permettent de fixer et d'obtenir la valeur des attributs pertinents de ces objets.

```
public abstract class Automobile implements Cloneable
{
  protected String modele;
  protected String couleur;
  protected int puissance;
  protected double espace;

  public Automobile duplique()
  {
    Automobile resultat;
    try
    {
      resultat = (Automobile)this.clone();
    }
    catch (CloneNotSupportedException exception)
    {
      return null;
    }
    return resultat;
  }

  public String getModele()
  {
    return modele;
  }

  public void setModele(String modele)
  {
    this.modele = modele;
  }

  public String getCouleur()
  {
    return couleur;
  }

  public void setCouleur(String couleur)
  {
    this.couleur = couleur;
  }

  public int getPuissance()
  {
    return puissance;
  }

  public void setPuissance(int puissance)
  {
    this.puissance = puissance;
  }

  public double getEspace()
  {
```

```
    return espace;
  }

  public void setEspace(double espace)
  {
    this.espace = espace;
  }

  public abstract void afficheCaracteristiques();
}

public class AutomobileElectricite extends Automobile
{

  public void afficheCaracteristiques()
  {
    System.out.println(
      "Automobile électrique de modèle : " + modele +
      " de couleur : " + couleur + " de puissance : " +
      puissance + " d'espace : " + espace);
  }
}

public class AutomobileEssence extends Automobile
{

  public void afficheCaracteristiques()
  {
    System.out.println(
      "Automobile à essence de modèle : " + modele +
      " de couleur : " + couleur + " de puissance : " +
      puissance + " d'espace : " + espace);
  }
}

public abstract class Scooter implements Cloneable
{
  protected String modele;
  protected String couleur;
  protected int puissance;

  public Scooter duplique()
  {
    Scooter resultat;
    try
    {
      resultat = (Scooter)this.clone();
    }
    catch (CloneNotSupportedException exception)
    {
      return null;
    }
    return resultat;
  }


  public String getModele()
  {
    return modele;
  }


  public void setModele(String modele)
  {
    this.modele = modele;
  }


  public String getCouleur()
```

```
  {
    return couleur;
  }


  public void setCouleur(String couleur)
  {
    this.couleur = couleur;
  }


  public int getPuissance()
  {
    return puissance;
  }


  public void setPuissance(int puissance)
  {
    this.puissance = puissance;
  }


  public abstract void afficheCaracteristiques();
}

public class ScooterElectricite extends Scooter
{

  public void afficheCaracteristiques()
  {
    System.out.println("Scooter électrique de modèle : "
      + modele + " de couleur : " + couleur +
      " de puissance : " + puissance);
  }

}

public class ScooterEssence extends Scooter
{

  public void afficheCaracteristiques()
  {
    System.out.println("Scooter à essence de modèle : " +
      modele + " de couleur : " + couleur +
      " de puissance : " + puissance);
  }

}
```

Nous donnons maintenant le code de la classe `FabriqueVehicule` basée sur l'utilisation des prototypes. Il convient de noter que les méthodes de création prêtent attention au cas où la référence vers un prototype a pour valeur `null`. Le client d'une fabrique peut spécifier les prototypes lors de l'instanciation en fournissant leur référence au constructeur.

```
public class FabriqueVehicule
{
  protected Automobile prototypeAutomobile;
  protected Scooter prototypeScooter;

  public FabriqueVehicule()
  {
    prototypeAutomobile = null;
    prototypeScooter = null;
  }

  public FabriqueVehicule(Automobile prototypeAutomobile,
    Scooter prototypeScooter)
  {
    this.prototypeAutomobile = prototypeAutomobile;
```

```
    this.prototypeScooter = prototypeScooter;
  }

  public Automobile getPrototypeAutomobile()
  {
    return prototypeAutomobile;
  }

  public void setPrototypeAutomobile(Automobile
    prototypeAutomobile)
  {
    this.prototypeAutomobile = prototypeAutomobile;
  }

  public Scooter getPrototypeScooter()
  {
    return prototypeScooter;
  }

  public void setPrototypeScooter(Scooter
    prototypeScooter)
  {
    this.prototypeScooter = prototypeScooter;
  }

  Automobile creeAutomobile()
  {
    if (prototypeAutomobile == null)
      return null;
    return prototypeAutomobile.duplique();
  }

  Scooter creeScooter()
  {
    if (prototypeScooter == null)
      return null;
    return prototypeScooter.duplique();
  }

}
```

Enfin, nous donnons un exemple de programme de test. Il est intéressant de voir comment y sont construits les produits. En effet, nous ne nous limitons pas à spécifier la classe des prototypes mais nous fournissons aussi des valeurs par défaut. Par exemple, le prototype `protoScooterClassicRouge` est un scooter à essence de modèle « classic » et de couleur « rouge ». Lorsqu'un scooter est créé sur la base de ce prototype, c'est un modèle « classic » de couleur « rouge ».

```
public class Test
{

  public static void main(String[] args)
  {
    Automobile protoAutomobileStandardBleu = new
      AutomobileElectricite();
    protoAutomobileStandardBleu.setModele("standard");
    protoAutomobileStandardBleu.setCouleur("bleu");

    Scooter protoScooterClassicRouge = new ScooterEssence();
    protoScooterClassicRouge.setModele("classic");
    protoScooterClassicRouge.setCouleur("rouge");

    FabriqueVehicule fabrique = new FabriqueVehicule();
    fabrique.setPrototypeAutomobile
      (protoAutomobileStandardBleu);
    fabrique.setPrototypeScooter(protoScooterClassicRouge);

    Automobile auto = fabrique.creeAutomobile();
    auto.afficheCaracteristiques();
    Scooter scooter = fabrique.creeScooter();
```

```
    scooter.afficheCaracteristiques();

    Automobile protoAutomobile2NewNoir = new
      AutomobileEssence();
    protoAutomobile2NewNoir.setModele("new");
    protoAutomobile2NewNoir.setCouleur("noir");
    fabrique.setPrototypeAutomobile
      (protoAutomobile2NewNoir);

    Automobile automobile = fabrique.creeAutomobile();
    automobile.afficheCaracteristiques();
  }
}
```

Enfin, l'exécution de ce programme fournit le résultat suivant :

```
Automobile électrique de modèle : standard de couleur : bleu
de puissance : 0 d'espace : 0.0
Scooter à essence de modèle : classic de couleur : rouge
de puissance : 0
Automobile à essence de modèle : new de couleur : noir
de puissance : 0 d'espace : 0.0
```

Le pattern Generation Gap

1. Introduction

Les outils de création d'interface utilisateur génèrent un code qui inclut :

- la création et la gestion des widgets de l'interface ;
- la création des méthodes invoquées lors du déclenchement des événements provenant de l'interface utilisateur.

Le corps de ces dernières méthodes doit être rempli par le développeur afin d'implanter la partie fonctionnelle de l'interface utilisateur de son application. L'utilisation de ces outils permet ainsi de construire rapidement une application possédant une interface utilisateur.

Nous donnons à la suite un exemple de code qu'un générateur aurait pu créer (il a été écrit manuellement en reprenant les principes d'un code généré automatiquement). Il s'agit d'une classe gérant le dialogue dont la figure 29.5 montre une copie d'écran. Cette classe est basée sur la bibliothèque `Swing` qui permet d'élaborer une interface utilisateur dans une application Java. Le dialogue permet la saisie du modèle d'une automobile.

Figure 29.5 - Dialogue de saisie

Les deux événements possibles sont l'appui sur les boutons **Annuler** et **OK**. Le code Java généré est le suivant :

```
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;

import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JTextField;

public class SaisieModele extends Dialogue
{
  public SaisieModele()
  {
    frame = new JFrame("Saisie du modèle d'automobile");
    frame.setDefaultCloseOperation
      (JFrame.DISPOSE_ON_CLOSE);
    frame.setBounds(20, 30, 300, 140);
    frame.getContentPane().setLayout(null);

    etiquetteModele = new JLabel("Modèle : ");
    etiquetteModele.setBounds(20, 30, 70, 20);
    frame.getContentPane().add(etiquetteModele);

    texteModele = new JTextField("");
    texteModele.setBounds(80, 30, 200, 20);
    frame.getContentPane().add(texteModele);

    boutonAnnuler = new JButton("Annuler");
    frame.getContentPane().add(boutonAnnuler);
    boutonAnnuler.setBounds(130, 70, 80, 20);
```

```
    boutonAnnuler.addActionListener(new ActionListener()
    {
      public void actionPerformed(ActionEvent e)
      {
        SaisieModele.this.boutonAnnulerEnfonce();
      }
    }
    );

    boutonOk = new JButton("OK");
    frame.getContentPane().add(boutonOk);
    boutonOk.setBounds(220, 70, 60, 20);

    boutonOk.addActionListener(new ActionListener()
    {
      public void actionPerformed(ActionEvent e)
      {
        SaisieModele.this.boutonOkEnfonce();
      }
    }
    );

    frame.setVisible(true);
  }

  protected JTextField texteModele;
  protected JLabel etiquetteModele;
  protected JButton boutonAnnuler;
  protected JButton boutonOk;

  protected void boutonAnnulerEnfonce()
  {
  }

  protected void boutonOkEnfonce()
  {
  }

}
```

Nous voyons que le code généré consiste à créer la classe `SaisieModèle` en introduisant respectivement :

- le code d'élaboration du dialogue et de ses widgets : c'est le constructeur de la classe qui en est chargé ;
- la déclaration des attributs référençant les widgets du dialogue ;
- les méthodes `boutonAnnulerEnfoncé` et `boutonOkEnfoncé` invoquées lors du déclenchement des événements correspondants.

Le corps des méthodes `boutonAnnulerEnfoncé` et `boutonOkEnfoncé` doit ensuite être rempli par le développeur de l'application.

La représentation par un diagramme de classes UML est donnée à la figure 29.6 qui montre également la surclasse abstraite `Dialogue` qui contient un ensemble de méthodes utilitaires destinées à faciliter le travail du développeur de l'application.

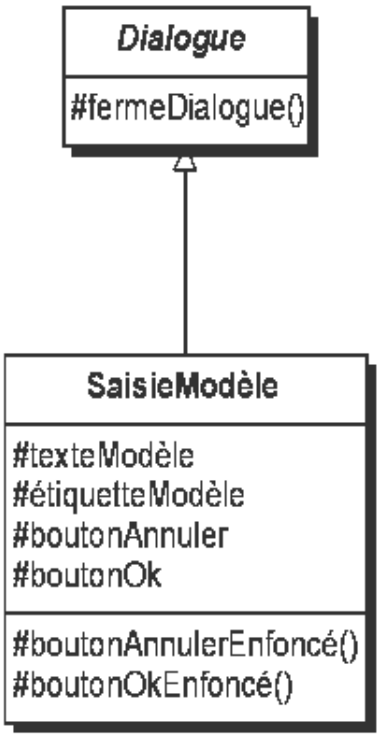


Figure 29.6 - Diagramme des classes générées

Nous remplissons les méthodes `boutonAnnulerEnfoncé` et `boutonOkEnfoncé` de la façon suivante :

```
  protected void boutonAnnulerEnfonce()
  {
    System.out.println("bouton Annuler enfoncé");
    fermeDialogue();
  }

  protected void boutonOkEnfonce()
  {
    System.out.println("bouton OK enfoncé");
    fermeDialogue();
  }
```

Ces deux méthodes invoquent la méthode `fermeDialogue` de la classe `Dialogue` :

```
import javax.swing.JFrame;

public abstract class Dialogue
{
  JFrame frame;

  protected void fermeDialogue()
  {
    frame.dispose();
  }
}
```

Une fois le corps de ces deux méthodes rempli, cette technique de génération va poser plusieurs problèmes lors des phases futures de maintenance :

- Il va devenir de plus en plus difficile de distinguer dans la classe `SaisieModèle` la partie qui a été générée automatiquement de la partie qui a été écrite par le développeur. Il est bien sûr possible d'utiliser des commentaires mais les commentaires ne sont pas des outils précis ;

- Toute modification dans le dialogue de saisie va impliquer de générer à nouveau la classe `SaisieModèle`. Cette nouvelle génération ne doit pas effacer le corps des méthodes `boutonAnnulerEnfoncé` et

`boutonOkEnfoncé`. Ce travail peut être d'autant plus difficile que le développeur a pu introduire de nouvelles méthodes (destinées à gérer des aspects communs aux deux méthodes) et de nouveaux attributs ;

- Il faut s'assurer que le développeur ne modifie pas le code généré. C'est assez difficile car le développeur doit insérer du code dans certaines méthodes de la classe générée et ne pas modifier d'autres méthodes de cette même classe.

2. Présentation

Le pattern `Generation Gap` offre une solution à l'ensemble de ces problèmes. Il consiste simplement à séparer la partie générée de la partie que le développeur doit écrire en les isolant dans deux classes différentes. Le diagramme des classes de la figure 29.7 illustre cette solution.

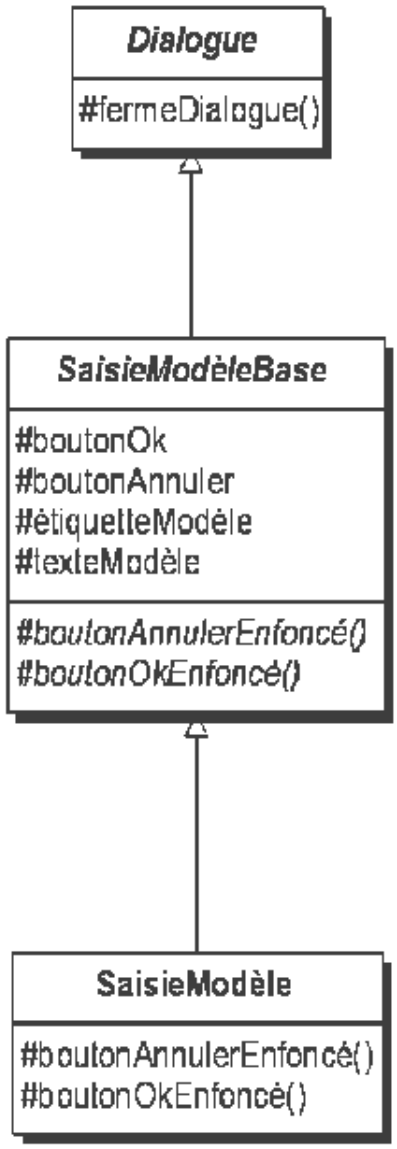


Figure 29.7 - Exemple d'utilisation du pattern `Generation Gap`

La classe `SaisieModèle` est transformée en une classe abstraite appelée `SaisieModèleBase`. Les méthodes `boutonAnnulerEnfoncé` et `boutonOkEnfoncé` deviennent des méthodes abstraites.

La sous-classe `SaisieModèle` est la seule classe à pouvoir être modifiée et à posséder du code écrit par le développeur de l'application. Cette sous-classe implante les méthodes `boutonAnnulerEnfoncé` et `boutonOkEnfoncé`.

Il est même possible d'avoir plusieurs sous-classes de `SaisieModèleBase`. Ceci offre la possibilité d'avoir deux codes distincts pour la partie applicative de l'interface, par exemple, un code de test et un code définitif. Le

pattern Generation Gap présente donc des similarités avec le pattern Template Method décrit dans le chapitre Le pattern Template Method dans l'utilisation des méthodes abstraites.

Le code Java des classes SaisieModèleBase, SaisieModèle ainsi que la classe TestSaisieModèle servant à faire fonctionner le dialogue est fourni à la suite. La classe TestSaisieModèle est le client du dialogue, il n'utilise que la classe SaisieModèle.

Enfin, il faut noter que le code du constructeur de SaisieModèleBase n'est pas fourni, il est identique au code décrit précédemment.

```
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;

import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JTextField;

public abstract class SaisieModeleBase extends Dialogue
{
  public SaisieModeleBase()
  {
  // code inchangé
  }

  protected JTextField texteModele;
  protected JLabel etiquetteModele;
  protected JButton boutonAnnuler;
  protected JButton boutonOk;

  protected abstract void boutonAnnulerEnfonce();

  protected abstract void boutonOkEnfonce();

}

public class SaisieModele extends SaisieModeleBase
{

  protected void boutonAnnulerEnfonce()
  {
    System.out.println("bouton Annuler enfoncé");
    fermeDialogue();
  }

  protected void boutonOkEnfonce()
  {
    System.out.println("bouton OK enfoncé");
    fermeDialogue();
  }
}

public class TestSaisieModele
{

  public static void main(String[] args)
  {
    new SaisieModele();
  }

}
```

3. Structure

La figure 29.8 illustre la structure générique du pattern Generation Gap.

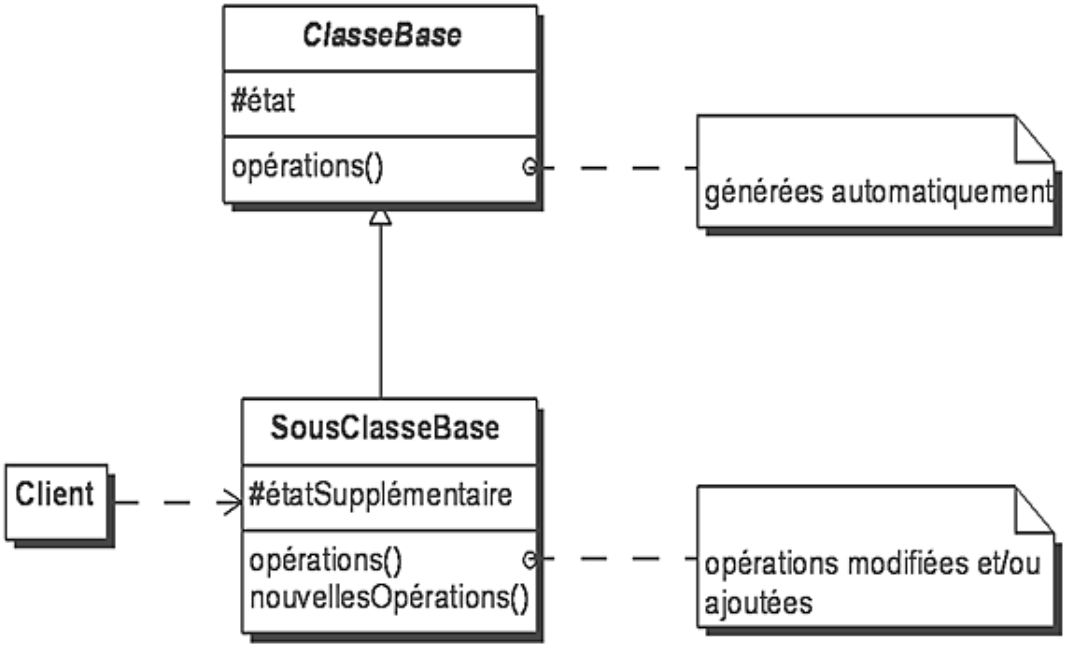


Figure 29.8 - Structure du pattern `Generation Gap`

Les participants du pattern sont les suivants :

- `ClasseBase` (`SaiseModèleBase`) est la classe abstraite créée par un outil de construction d'interface utilisateur. Elle n'est jamais modifiée à la main. Elle est régénérée par l'outil à chaque modification ;
- `SousClasseBase` (`SaiseModèle`) est une sous-classe de `ClasseBase`. Elle implante la partie fonctionnelle de l'interface utilisateur de l'application. Elle est préservée lors des régénérations produites par l'outil de construction ;
- `Client` (`TestSaisieModèle`) : le client utilise uniquement la classe `SousClasseBase`.

La collaboration entre les méthodes de la classe abstraite `ClasseBase` et les méthodes de la classe `SousClasseBase` réalise l'implantation de l'interface utilisateur de l'application.

Le pattern Multicast

1. Description et exemple

Le but du pattern `Multicast` est de gérer les événements produits dans un programme afin de les transmettre à un ensemble de récepteurs concernés. Le pattern est basé sur un mécanisme d'inscription des récepteurs auprès des expéditeurs.

Nous voulons mettre en œuvre un programme d'envoi de messages entre les directions (générale, commerciale, financière, etc.) d'un concessionnaire et ses employés.

Chaque employé peut s'inscrire auprès de la direction à laquelle il appartient et recevoir ainsi tous les messages émis par cette dernière. Un employé ne peut pas s'inscrire auprès d'une direction à laquelle il n'appartient pas. Tous les employés peuvent bien sûr s'inscrire auprès de la direction générale afin d'en recevoir les messages.

La structure des messages peut varier d'une direction à l'autre : simple ligne de texte pour les messages commerciaux, liste de lignes pour les messages généraux provenant de la direction générale.

Le diagramme de classes de la figure 29.9 expose la solution proposée par le pattern `Multicast`. La généricité de types est utilisée pour créer un message, un expéditeur et un récepteur abstraits et génériques, à savoir les classes `MessageAbstrait` et `ExpéditeurAbstrait` ainsi que l'interface `RécepteurAbstrait`.

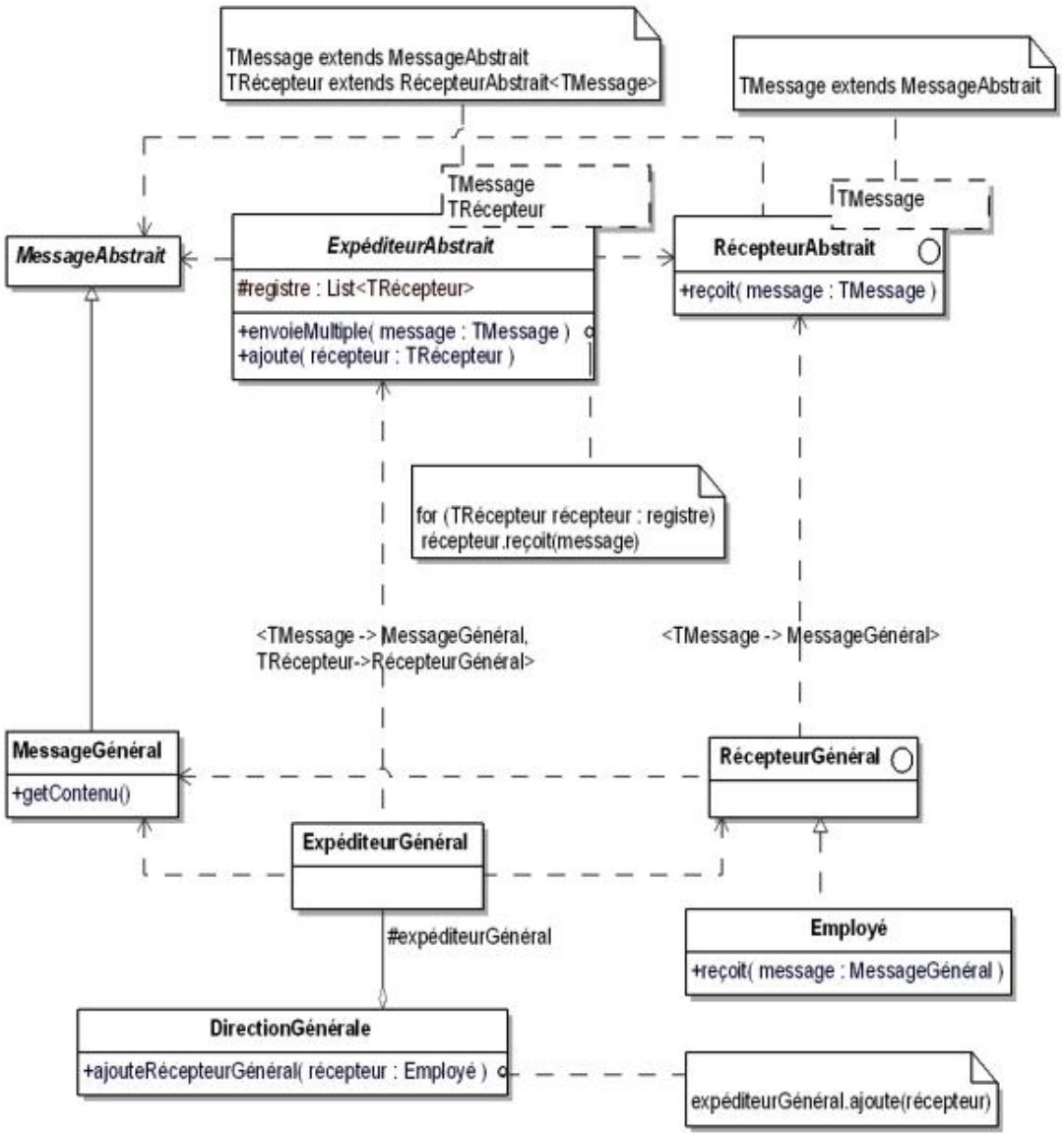


Figure 29.9 - Le pattern `Multicast` appliqué à un système de messages

La classe ExpéditeurAbstrait présente deux fonctionnalités :

- Elle gère un registre (une liste) de récepteurs avec la possibilité de s'y inscrire grâce à la méthode ajoute.
- Elle permet d'envoyer un message à l'ensemble des récepteurs présents dans le registre grâce à la méthode envoieMultiple.

Elle est basée sur deux types génériques à savoir TMessage qui est contraint par MessageAbstrait et TRécepteur contraint par RécepteurAbstrait<TMessage>. Ainsi, tout message est forcément une sous-classe de MessageAbstrait et tout récepteur une sous-classe de RécepteurAbstrait<TMessage>.

La classe MessageAbstrait est une classe abstraite totalement vide. Elle n'existe qu'à des fins de typage.

L'interface RécepteurAbstrait est une interface qui introduit la signature de la méthode reçoit. Cette interface est basée sur le type générique TMessage contraint par MessageAbstrait.

Nous nous intéressons maintenant au cas particulier des messages provenant de la direction générale. Pour ces messages, nous créons une sous-classe pour chaque classe abstraite :

- La sous-classe concrète MessageGénéral qui décrit la structure d'un message de la direction générale. La méthode getContenu permet d'obtenir le contenu d'un tel message.
- La sous-classe concrète ExpéditeurGénéral obtenue en liant les deux paramètres génériques à MessageGénéral pour TMessage et à RécepteurGénéral pour TRécepteur.
- L'interface RécepteurGénéral qui hérite de l'interface RécepteurAbstrait en liant le paramètre générique TMessage à MessageGénéral.

La classe Employé introduit les objets qui représentent les employés de la concession. Elle implante l'interface RécepteurGénéral. Ainsi ses instances sont dotées de la capacité de recevoir les messages provenant de la direction générale et peuvent donc s'inscrire pour les recevoir.

La classe DirectionGénérale représente la direction générale. Elle possède un lien vers expéditeurGénéral, instance de la classe ExpéditeurGénéral qui lui permet d'envoyer des messages. La méthode ajouteRécepteurGénéral permet de reporter l'inscription des employés au niveau de la classe DirectionGénérale.

2. Structure

La structure générique du pattern Multicast est illustrée par le diagramme de classes de la figure 29.10.

Ce diagramme de classes est très similaire au diagramme de l'exemple de la figure 29.9, les classes abstraites ayant été conservées dans l'exemple.

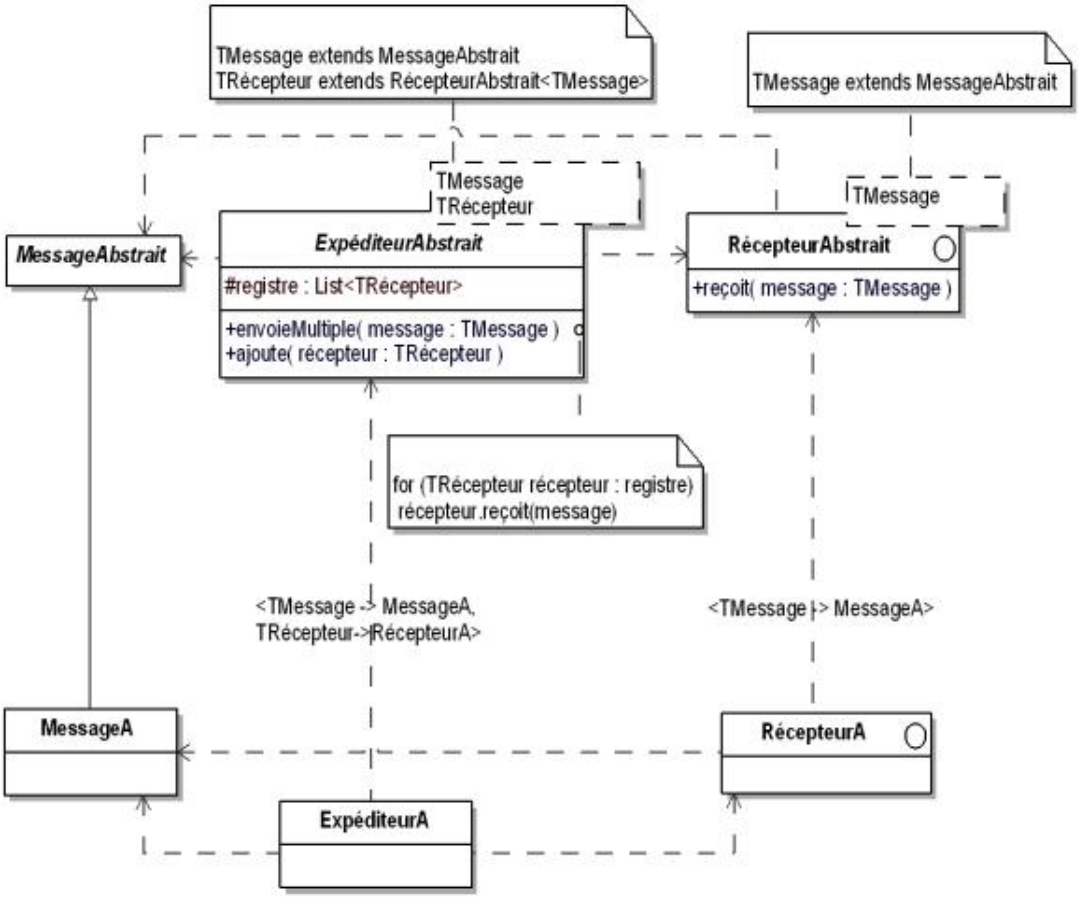


Figure 29.10 - La structure du pattern `Multicast`

Les participants au pattern sont les suivants :

- `MessageAbstrait` est la classe abstraite qui introduit le type des messages ;
- `ExpéditeurAbstrait` est la classe abstraite qui implante le registre des récepteurs et la méthode `envoieMultiple` qui envoie un message à tous les récepteurs du registre ;
- `RécepteurAbstrait` est l'interface qui définit la signature de la méthode `reçoit` ;
- `MessageA` (`MessageGénéral`) est une sous-classe concrète de `MessageAbstrait` qui décrit la structure des messages ;
- `ExpéditeurA` (`ExpéditeurGénéral`) est une sous-classe concrète représentant les expéditeurs de messages. Elle lie le paramètre `TMessage` à `MessageA` et le paramètre `TRécepteur` à `RécepteurA` ;
- `RécepteurA` (`RécepteurGénéral`) est une interface héritant de `RécepteurAbstrait` en liant le paramètre `TMessage` à `MessageA`. Elle doit être implantée par toute classe d'objets qui veulent avoir la capacité de recevoir des messages typés par la classe `MessageA`.

La collaboration entre les objets est décrite à la suite :

- Les récepteurs s'inscrivent auprès de l'expéditeur de message.
- Les expéditeurs envoient les messages aux récepteurs inscrits.

3. Exemple en Java

Nous donnons maintenant le code Java de l'exemple correspondant au diagramme de classes de la figure 29.9. Nous introduisons d'abord les classes abstraites et interfaces `MessageAbstrait`, `RecepteurAbstrait` et `ExpediteurAbstrait`.

```
public abstract class MessageAbstrait{
}

public interface RecepteurAbstrait
  <TMessage extends MessageAbstrait>
{
  public void recoit(TMessage message);
}

import java.util.*;
public abstract class ExpediteurAbstrait
  <TMessage extends MessageAbstrait,
   TRecepteur extends RecepteurAbstrait<TMessage>>
{
  protected List<TRecepteur> registre =
    new ArrayList<TRecepteur>();

  public void ajoute(TRecepteur recepteur)
  {
    registre.add(recepteur);
  }

  public void envoieMultiple(TMessage message)
  {
    for (TRecepteur recepteur: registre)
      recepteur.recoit(message);
  }

}
```

Nous présentons ensuite les classes et interfaces relatives aux messages généraux : `MessageGeneral`, `RecepteurGeneral` et `ExpediteurGeneral`.

```
import java.util.*;
public class MessageGeneral extends MessageAbstrait
{
  protected List<String> contenu;

  public List<String> getContenu()
  {
    return contenu;
  }

  public MessageGeneral(List<String> contenu)
  {
    super();
    this.contenu = contenu;
  }

}

public interface RecepteurGeneral extends
  RecepteurAbstrait<MessageGeneral> {
}

public class ExpediteurGeneral extends ExpediteurAbstrait
  <MessageGeneral, RecepteurGeneral> {
}
```

Le code de la classe `DirectionGenerale` est donné à la suite. Cette classe possède un lien vers une instance de la classe `ExpediteurGeneral`. Celle-ci est utilisée pour ajouter des employés au registre et pour envoyer des messages.

```
import java.util.*;
public class DirectionGenerale
{
```

```
  protected ExpediteurGeneral expediteurGeneral = new
    ExpediteurGeneral();

  public void envoieMessages()
  {
    List<String> contenu = new ArrayList<String>();
    contenu.add("Informations générales");
    contenu.add("Informations spécifiques");
    MessageGeneral message = new MessageGeneral(contenu);
    expediteurGeneral.envoieMultiple(message);
  }

  public void ajouteRecepteurGeneral(Employe recepteur)
  {
    expediteurGeneral.ajoute(recepteur);
  }

}
```

La classe `Employe` est décrite à la suite. Elle implante l'interface `RecepteurGeneral` afin de pouvoir recevoir les messages généraux. Cette classe est abstraite : nous allons par la suite la doter des deux sous-classes concrètes `Administratif` et `Commercial`.

```
public abstract class Employe implements RecepteurGeneral
{
  protected String nom;

  public Employe(String nom)
  {
    super();
    this.nom = nom;
  }

  public void recoit(MessageGeneral message)
  {
    System.out.println("Nom : " + nom);
    System.out.println("Message : ");
    for (String ligne: message.getContenu())
      System.out.println(ligne);
  }

}
```

La sous-classe concrète `Administratif` est décrite ci-dessous. Elle est très simple.

```
public class Administratif extends Employe
{

  public Administratif(String nom)
  {
    super(nom);
  }

}
```

Nous introduisons également un second message : les messages commerciaux liés à la direction commerciale. Le code Java des classes et interfaces correspondantes, à savoir `MessageCommercial`, `RecepteurCommercial`, `ExpediteurCommercial`, `DirectionCommerciale` et `Commercial` se trouve ci-dessous.

```
public class MessageCommercial extends MessageAbstrait
{
  protected String contenu;

  public String getContenu()
  {
    return contenu;
  }

  public MessageCommercial(String contenu)
```

```
  {
    super();
    this.contenu = contenu;
  }

}

public interface RecepteurCommercial extends
  RecepteurAbstrait<MessageCommercial> {
}

public class ExpediteurCommercial extends
  ExpediteurAbstrait<MessageCommercial,
    RecepteurCommercial> {
}

public class DirectionCommerciale
{
  protected ExpediteurCommercial expediteurCommercial =
    new ExpediteurCommercial();

  public void envoieMessages()
  {
    MessageCommercial message = new MessageCommercial(
      "Annonce nouvelle gamme");
    expediteurCommercial.envoieMultiple(message);
    message = new MessageCommercial(
      "Annonce suppression modèle");
    expediteurCommercial.envoieMultiple(message);
  }

  public void ajouteRecepteurCommercial
    (RecepteurCommercial recepteur)
  {
    expediteurCommercial.ajoute(recepteur);
  }

}

public class Commercial extends Employe
{

  protected RecepteurCommercial recepteurCommercial =
    new RecepteurCommercial()
  {
    public void recoit(MessageCommercial message)
    {
      System.out.println("Nom : " + nom);
      System.out.println("Message : " +
        message.getContenu());
    }
  };

  public Commercial(String nom)
  {
    super(nom);
  }

  public RecepteurCommercial getRecepteurCommercial()
  {
    return recepteurCommercial;
  }

}
```

La classe `Commercial` n'implante pas `RecepteurCommercial` mais utilise à la place une instance d'une classe anonyme interne implantant cette interface ! C'est cette instance et non l'instance de la classe `Commercial` qui reçoit les messages. La raison est que Java refuse qu'une classe implante deux fois une même interface. Si la

classe commercial implantait RecepteurCommercial, alors elle implanterait deux fois l'interface RecepteurAbstrait : une fois avec RecepteurCommercial qui hérite de RecepteurAbstrait<MessageCommercial> et une deuxième fois en héritant de la classe Employe qui implante RecepteurGeneral qui hérite de RecepteurAbstrait<MessageGeneral>.

Enfin, nous fournissons le code Java d'un programme de test de cet ensemble de classes.

```
public class Concession
{

  public static void main(String[] args)
  {
    DirectionGenerale directionGenerale = new
      DirectionGenerale();
    DirectionCommerciale directionCommerciale = new
      DirectionCommerciale();
    Commercial commercial1 = new Commercial("Paul");
    Commercial commercial2 = new Commercial("Henri");
    Administratif administratif = new Administratif(
      "Jacques");
    directionGenerale.ajouteRecepteurGeneral(commercial1);
    directionGenerale.ajouteRecepteurGeneral(commercial2);
    directionGenerale.ajouteRecepteurGeneral
      (administratif);
    directionGenerale.envoieMessages();
    directionCommerciale.ajouteRecepteurCommercial
      (commercial1.getRecepteurCommercial());
    directionCommerciale.ajouteRecepteurCommercial
      (commercial2.getRecepteurCommercial());
    directionCommerciale.envoieMessages();
  }
}
```

L'exécution de ce programme fournit le résultat suivant.

```
Nom : Paul
Message :
Informations générales
Informations spécifiques
Nom : Henri
Message :
Informations générales
Informations spécifiques
Nom : Jacques
Message :
Informations générales
Informations spécifiques
Nom : Paul
Message : Annonce nouvelle gamme
Nom : Henri
Message : Annonce nouvelle gamme
Nom : Paul
Message : Annonce suppression modèle
Nom : Henri
Message : Annonce suppression modèle
```

4. Discussion : comparaison avec le pattern Observer

Le pattern Observer (chapitre Le pattern Observer) présente de fortes similitudes avec le pattern Multicast. D'une part, il permet d'inscrire des observateurs, l'équivalent des récepteurs. D'autre part, il peut envoyer une notification d'actualisation aux observateurs, c'est-à-dire un équivalent des messages.

Dans le chapitre Le pattern Observer, la notification d'actualisation ne transmet pas d'information, à la différence des messages du pattern Multicast. Cependant, il n'est pas très compliqué d'étendre le pattern Observer pour ajouter une transmission d'information lors de la notification d'actualisation.

Il est alors légitime de se demander si le pattern Multicast n'est donc pas qu'une simple extension du pattern Observer. La réponse est négative ! Le but du pattern Observer est de construire une dépendance entre un sujet et

des observateurs de sorte que chaque modification du sujet soit notifiée aux observateurs. L'ensemble formé du sujet et des observateurs constitue, d'une certaine façon, un seul objet composé. Par ailleurs, une utilisation quasi immédiate du pattern `Multicast` est la possibilité de créer plusieurs expéditeurs envoyant des messages à un seul ou plusieurs récepteurs. Un récepteur peut être connecté à plusieurs expéditeurs, comme c'est le cas dans notre exemple où un commercial peut recevoir des messages de la direction générale et de la direction commerciale. Cette utilisation est aux antipodes des objectifs du pattern `Observer`, prouvant bien que `Observer` et `Multicast` sont bien deux patterns distincts.

Modélisation et conception avec les patterns de conception

Dans cet ouvrage, nous avons étudié les patterns de conception au travers de leur mise en œuvre dans des exemples. Ces patterns facilitent la conception en offrant des solutions solides à des problèmes connus. Ces solutions sont basées sur une architecture qui respecte les bonnes pratiques de la programmation par objets.

Dans l'analyse d'un nouveau projet, l'étape de découverte des objets et de leur modélisation ne nécessite pas l'utilisation des patterns de conception. Celle-ci débouche sur plusieurs diagrammes de classes contenant les classes représentant les objets du domaine. Ces objets sont issus de la modélisation, ils ne sont pas destinés à résoudre directement les aspects fonctionnels d'une application. Dans le cadre de notre système de ventes en ligne, ces objets sont les véhicules, les constructeurs, les clients, le vendeur, les fournisseurs, les commandes, les factures, etc. La figure 30.1 montre une partie de cette modélisation, à savoir le diagramme de classes des véhicules.

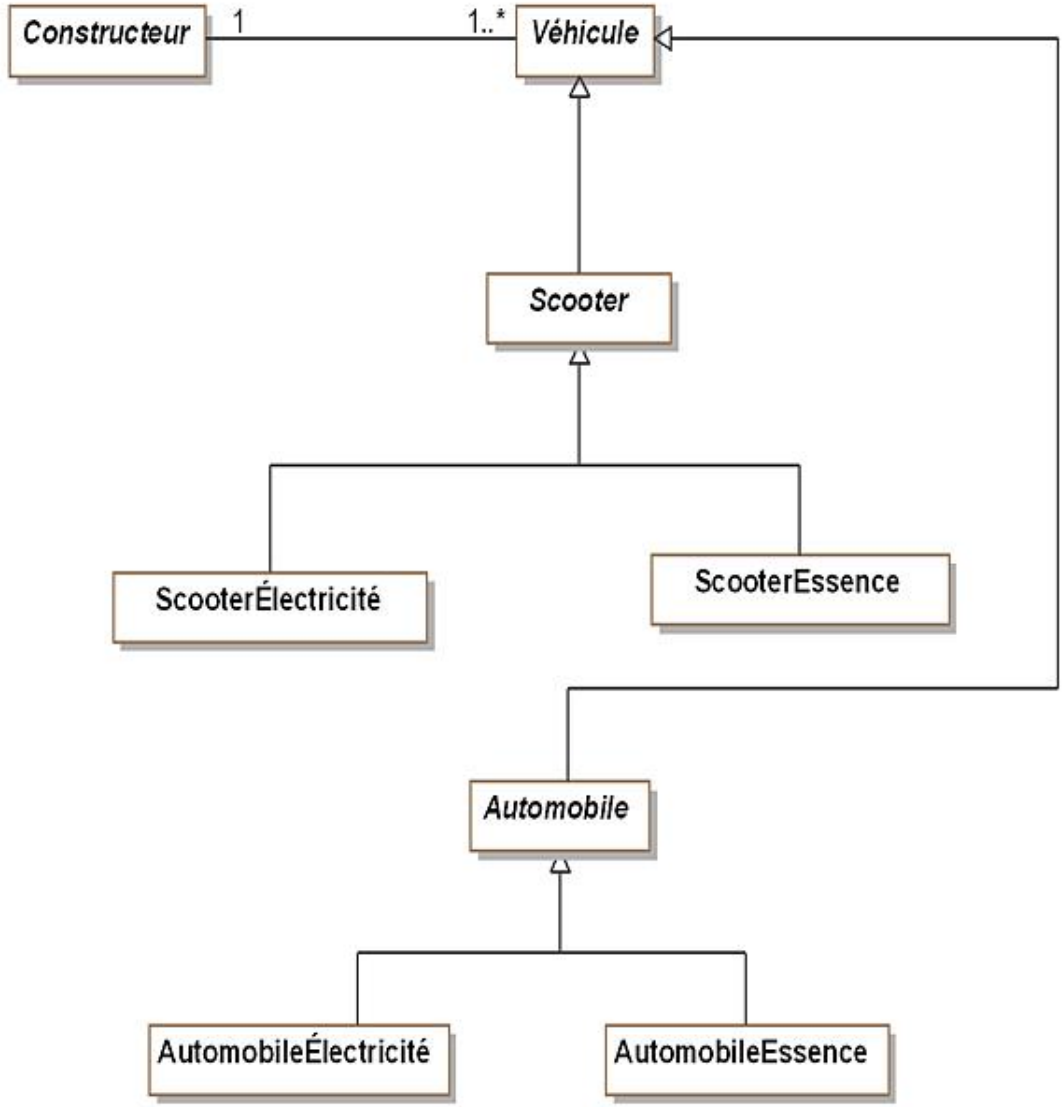


Figure 30.1 - Diagramme de classes des véhicules

Cette hiérarchie décrit la structure en objets du domaine. Ensuite, il faut implanter les fonctionnalités du système, ce qui nécessite de nouveaux objets. À la différence des objets du domaine, ces objets sont purement techniques et spécialisés dans la réalisation des fonctionnalités.

Prenons d'abord l'exemple du catalogue en ligne des véhicules disponibles à la vente. L'implantation de cette fonctionnalité nécessite l'introduction d'un objet technique `VueVéhicule` chargé de dessiner un véhicule avec ses caractéristiques et ses photos. Pour implanter son lien avec l'objet du domaine `Véhicule` et ses exigences en terme de rafraîchissement à l'écran, nous adoptons le pattern `Observer` comme l'illustre l'exemple du chapitre Le pattern Observer.

Un deuxième exemple concerne la création des objets du domaine. Il s'agit d'un aspect important du niveau fonctionnel d'un système. Si nous voulons rendre les aspects fonctionnels indépendants des familles de véhicules (véhicules

électriques et véhicules à essence dans le cas de la figure 30.1), le pattern `Abstract Factory` peut être mis en œuvre comme l'illustre l'exemple du chapitre Le pattern Abstract Factory.

Un autre exemple d'objets du domaine est la commande. Au niveau de la modélisation, cet objet est notamment décrit par le diagramme d'états-transitions de la figure 30.2.

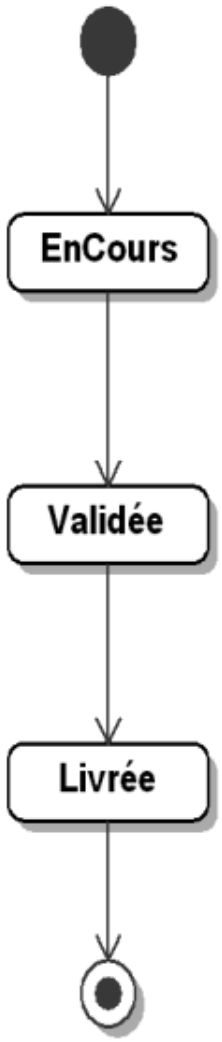


Figure 30.2 - Diagramme d'états-transitions d'une commande

Ce diagramme d'états-transitions peut être réalisé à l'aide du pattern `State` décrit au chapitre Le pattern State. Celui-ci montre comment un objet complémentaire qui représente l'état peut être associé à la commande. Cet objet sert à adapter le comportement de la méthode en fonction de son état interne.

Autres apports des patterns de conception

1. Un référentiel commun

Comme les classes, les patterns de conception constituent des abstractions. Mais à la différence des classes, les patterns portent sur plusieurs objets interagissant entre eux. Au long des chapitres, nous les avons représentés par une structure constituée d'un diagramme de classes complété par des explications sur les participations et sur les collaborations. Il s'agit donc d'une abstraction plus riche qu'une classe. Le fait de pouvoir nommer, décrire et classifier les patterns confère à leur catalogue un statut de référentiel. Ainsi, lors de la conception d'un système, il est possible d'évoquer l'utilisation d'un pattern par son nom, ce qui renvoie à une structure connue.

2. Un ensemble récurrent de techniques de conception

Il est tout à fait possible de concevoir des systèmes sans utiliser les patterns de conception. Mais après quelques temps, tout concepteur aura découvert de lui-même la plupart des patterns. L'avantage de les découvrir en lisant un ouvrage sur le sujet constitue un gain de temps et permet d'éviter les éventuels écueils d'utilisation.

3. Un outil pédagogique de l'approche à objets

Les patterns de conception possèdent également un aspect pédagogique : ils fournissent à un débutant un apprentissage des bonnes pratiques de la programmation par objets. Le débutant peut apprendre comment sont mis en œuvre les principes de l'approche à objets comme les associations entre objets, le polymorphisme, les interfaces, les classes et les méthodes abstraites, la délégation, la paramétrisation.

Énoncés des exercices

1. Création de cartes de paiement

a. Création en fonction du client

Les clients d'une banque sont classés en deux catégories :

- ceux qui ont le droit au crédit ;
- ceux qui n'ont pas ce droit.

Lors de la demande d'une carte de paiement, les premiers reçoivent une carte de crédit (à débit différé sur leur compte) alors que les seconds peuvent seulement avoir une carte de débit (à débit immédiat sur leur compte).

1. Quel pattern de conception permet-il de modéliser la création de la carte de paiement en fonction du client ?

2. Modélisez son utilisation par un diagramme de classes.

b. Création à l'aide d'une fabrique

Il existe deux modèles de cartes de débit et de crédit, à savoir les cartes Visa et les cartes MasterCard.

Modélisez, à l'aide d'un diagramme de classes, la création d'une carte de paiement en fonction de sa famille (de crédit ou de débit) en utilisant le pattern `Abstract Factory`.

2. Autorisation des cartes de paiement

Lors d'un achat avec une carte de paiement, une autorisation doit être accordée. Si la carte est une carte de débit (débit immédiat), l'autorisation est accordée si le solde du compte sur lequel la carte est débitée est suffisant. Si la carte est une carte de crédit (débit différé), l'autorisation est accordée si le montant mensuel des dépenses n'a pas dépassé le plafond.

1. Quel pattern de conception permet-il de modéliser l'autorisation lors d'un achat avec une carte de paiement en fonction du modèle de la carte ?

2. Modélisez son utilisation par un diagramme de classes.

3. Système de fichiers

Un répertoire contient des sous-répertoires et des fichiers. Un système de fichiers consiste en un ensemble de sous-répertoires et de fichiers contenus dans un répertoire "racine". Les fichiers et les répertoires appartiennent à un utilisateur.

1. Quel pattern permet-il de modéliser un système de fichiers ?

2. Montrez cette modélisation à l'aide d'un diagramme de classes.

Chaque fichier possède un attribut qui contient sa taille. Nous voulons maintenant connaître le nombre de fichiers et de répertoires ainsi que la taille globale du système de fichiers (les deux calculs doivent être séparés).

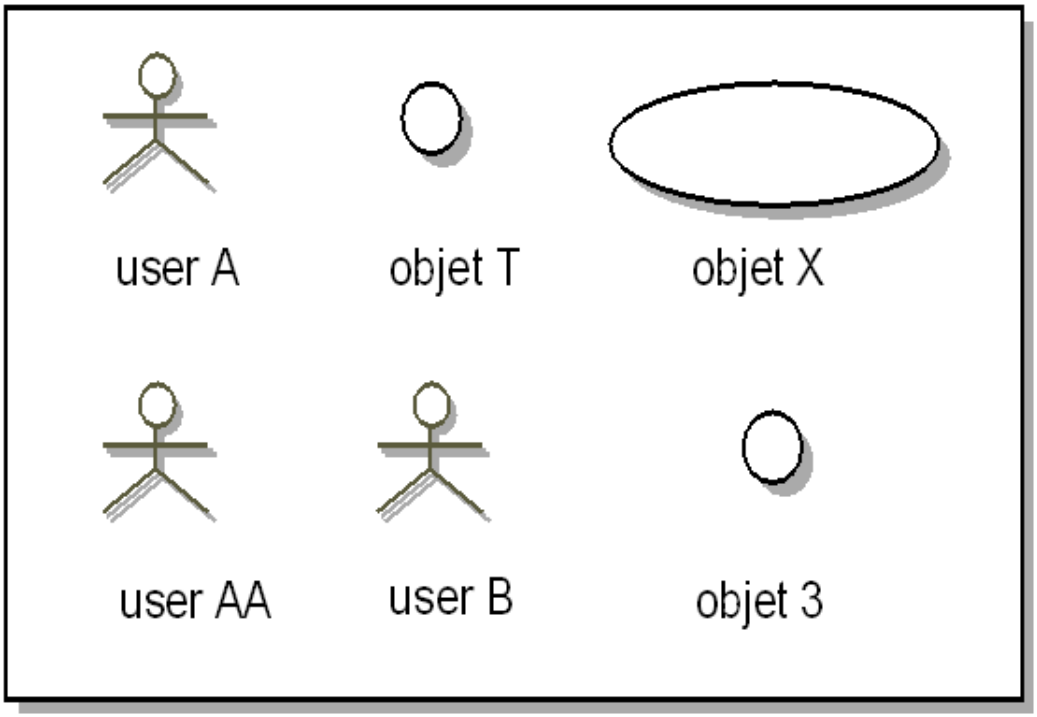
3. Quel pattern permet-il de calculer ces informations en modifiant au minimum le diagramme des classes de la question 2 ?

4. Intégrez ce pattern dans le diagramme de classes de la question 2.

5. Programmez ce diagramme de classes en Java en simulant à l'aide d'objets les fichiers et les répertoires. Le programme principal devra construire un exemple de système de fichiers et calculer le nombre de fichiers et de répertoires de cet exemple ainsi que sa taille globale.

4. Browser graphique d'objets

Un browser graphique permet d'afficher différents objets dans une fenêtre comme illustré ci-dessous.



Sur cette figure, trois types d'objets sont présents :

- les utilisateurs représentés par un personnage ;
- les ronds et les ovales représentés par un icône rond ou ovale.

Le browser affiche un tableau d'icônes qui représentent l'ensemble des objets présents dans un tableau. Il existe de nombreux autres objets représentés par des icônes spécifiques.

1. Quel pattern est le mieux adapté pour concevoir le browser ?

2. Concevez le diagramme de classes décrivant le browser utilisant ce pattern.

5. États de la vie professionnelle d'une personne

Une personne possède un cycle au long de sa vie professionnelle. Elle est d'abord étudiante, puis intègre la vie active puis prend sa retraite.

Son comportement varie en fonction de l'état où elle se trouve. Notamment, elle ne cotise pour sa retraite que lors de sa vie active. Les cotisations pour la retraite donnent lieu à l'ajout de points de retraite.

1. Quel pattern est le mieux adapté pour décrire une personne au long de sa vie professionnelle ?

2. Concevez le diagramme de classes décrivant la personne en utilisant ce pattern. Introduisez les méthodes `getNom` et `ajoutePoints` dont le comportement dépend de l'état. La méthode `getNom` renvoie le nom de l'état de la personne. La méthode `ajoutePoints` ajoute des points de retraite.

Il s'agit maintenant de créer une interface graphique qui affiche la personne ainsi que son état professionnel. Chaque fois que des données de la personne ou de son état sont changées, les données affichées dans l'interface graphique sont mises à jour automatiquement. Il s'agit ici du nom de l'état et des points de retraite.

3. Quel pattern est le mieux adapté pour concevoir cette interface graphique ?

4. Concevez le diagramme de classes correspondant en intégrant le diagramme de la question 2.

6. Cache d'un dictionnaire persistant d'objets

La classe `DictPersistant` permet de stocker de façon persistante des objets puis de les retrouver. Elle contient deux

méthodes :

- `ajoute` qui prend comme arguments la clef et l'objet à stocker et qui ajoute ce dernier au dictionnaire sauf si la clef est déjà présente ;

- `get` qui prend comme argument la clef de l'objet et renvoie l'objet.

La méthode `ajoute` renvoie `true` si l'ajout a pu être réalisé, `false` dans le cas contraire. La méthode `get` renvoie `null` si l'objet n'a pas été retrouvé dans le dictionnaire.

La classe `DictPersistant` est générique et prend comme argument le type des objets à stocker. Elle implante l'interface `DictPersistantIntf`, elle-même générique :

```
public interface DictPersistantIntf<T>
{
  boolean ajoute(String cle, T objet);
  T get(String cle);
}
```

Le code source de `DictPersistant` est le suivant :

```
import java.io.*;
public class DictPersistant<T> implements
  DictPersistantIntf<T>
{

  public boolean ajoute(String cle, T objet)
  {
    try
    {
      File fichierSortie = new File(cle);
      if (fichierSortie.createNewFile())
      {
        FileOutputStream streamSortie = new
          FileOutputStream(fichierSortie);
        ObjectOutputStream streamObjet = new
          ObjectOutputStream(streamSortie);
        streamObjet.writeObject(objet);
        streamObjet.flush();
        streamSortie.close();
        return true;
      }
    }
    catch (IOException e)
    {
      return false;
    }
    return false;
  }

  @SuppressWarnings({"unchecked"})
  public T get(String cle)
  {
    try
    {
      File fichierLecture = new File(cle);
      if (fichierLecture.exists())
      {
        T resultat;
        FileInputStream streamLecture = new
          FileInputStream(fichierLecture);
        ObjectInputStream streamObjet = new
          ObjectInputStream(streamLecture);
        try
        {
          resultat = (T)streamObjet.readObject();
        }
        catch (ClassNotFoundException e)
```

```
        {
          resultat = null;
        }
        streamLecture.close();
        return resultat;
      }
    }
    catch (IOException e)
    {
      return null;
    }
    return null;
  }

}
```

Chaque fois qu'un client accède à un objet au travers de la méthode `get`, une lecture depuis le disque se produit. Si les clients font des accès fréquents, il devient alors nécessaire de programmer un cache sous la forme d'un proxy. Ce cache conserve en mémoire les objets déjà chargés ou ceux qui ont été ajoutés lors d'une session.

Implantez ce proxy en Java en respectant le pattern `Proxy`.

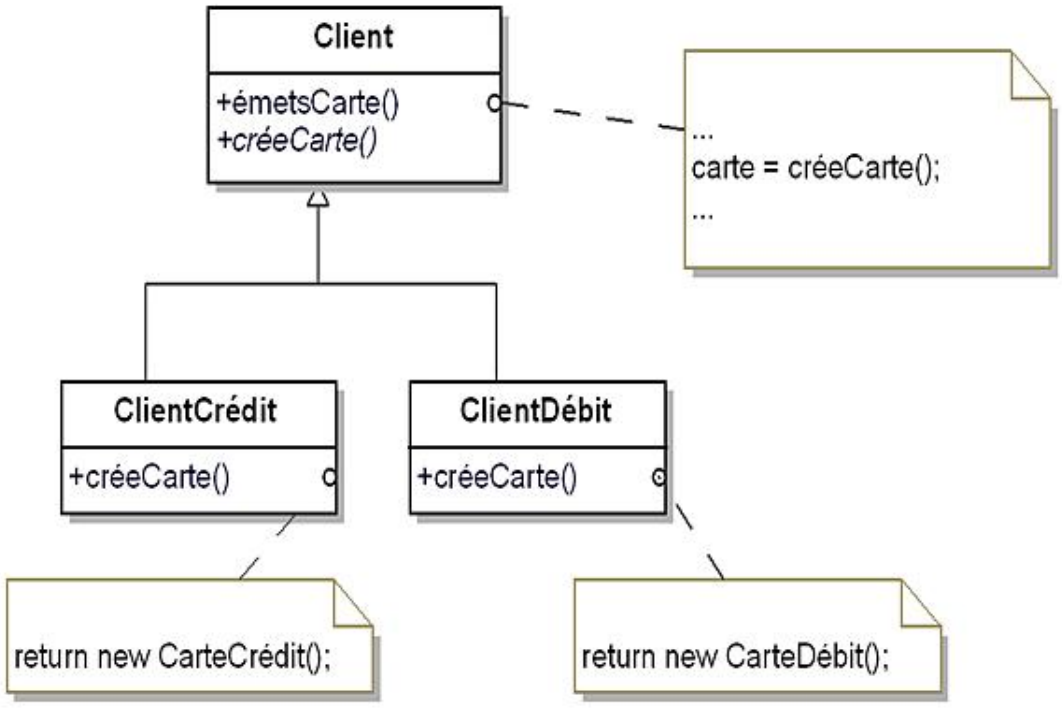
Correction des exercices

1. Création de cartes de paiement

a. Création en fonction du client

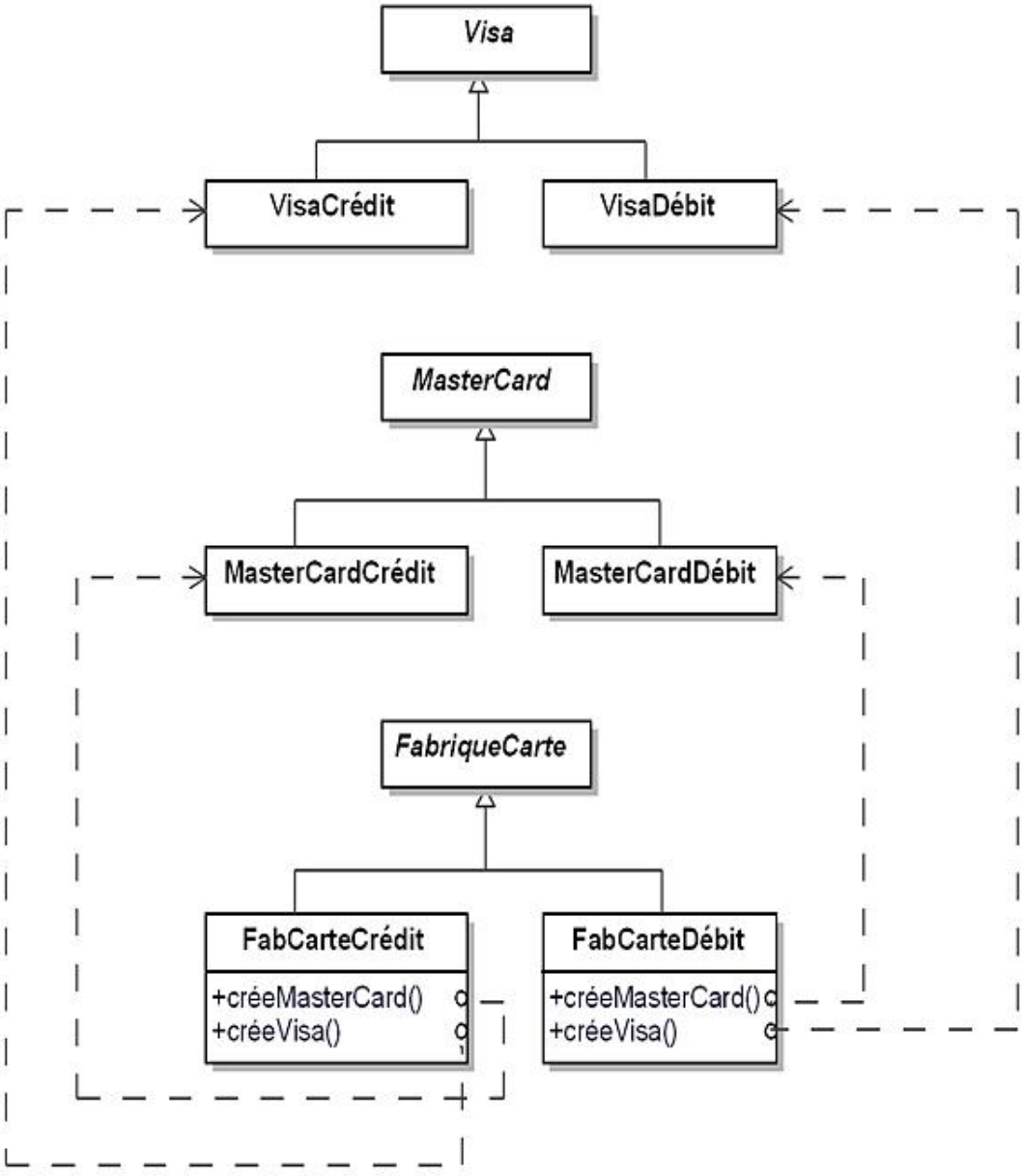
1. Le pattern adapté pour créer une carte en fonction du client est le pattern `Factory Method`. La création de la carte est réalisée dans la sous-classe correspondant à la nature du client.

2. Le diagramme des classes correspondant est donné à la suite.



b. Création à l'aide d'une fabrique

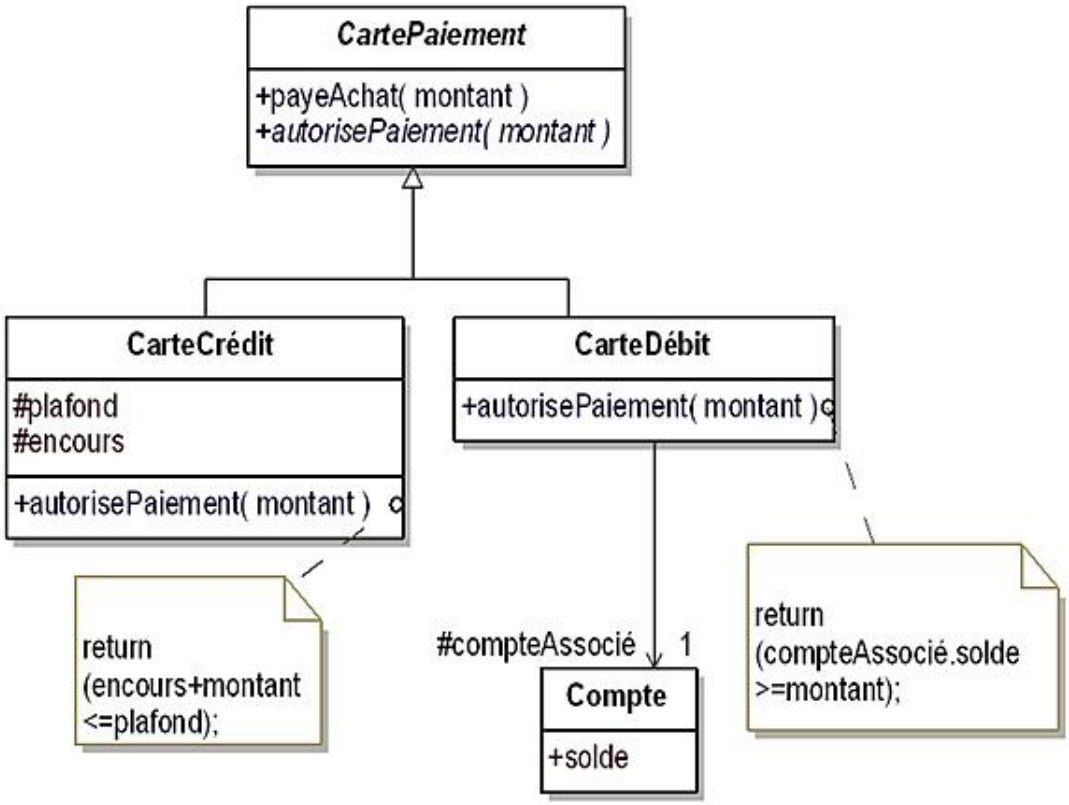
Le but est ici d'obtenir une fabrique de cartes MasterCard et Visa pour les cartes de crédit et une fabrique similaire pour les cartes de débit. Le diagramme des classes correspondant est donné ci-dessous.



2. Autorisation des cartes de paiement

1. Le pattern adapté est `Template Method`. Il permet de distinguer l'autorisation de paiement en fonction du modèle de carte. La méthode `autorisePaiement` est abstraite dans la classe `CartePaiement`. Elle est implantée différemment dans les deux sous-classes `CarteCrédit` et `CarteDébit` relativement à l'énoncé.

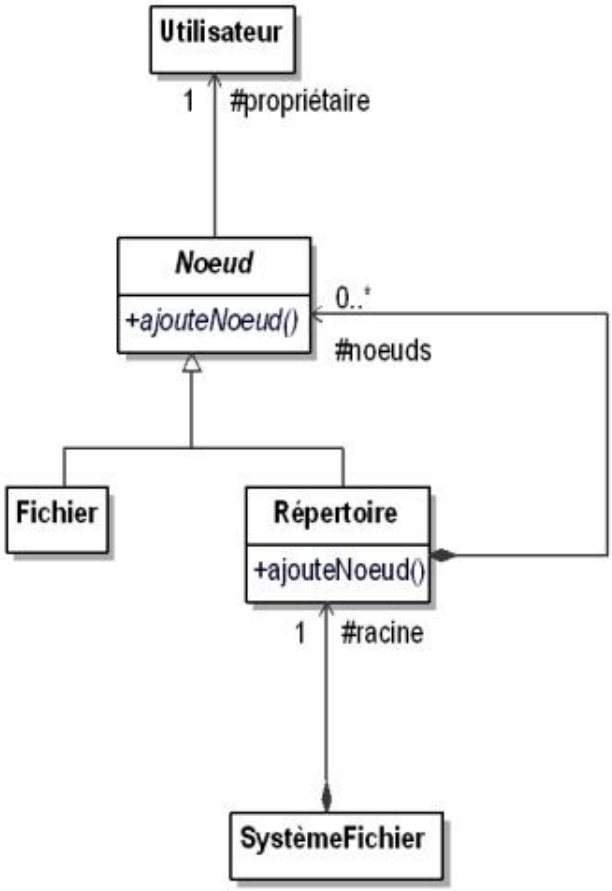
2. Le diagramme des classes est le suivant :



3. Système de fichiers

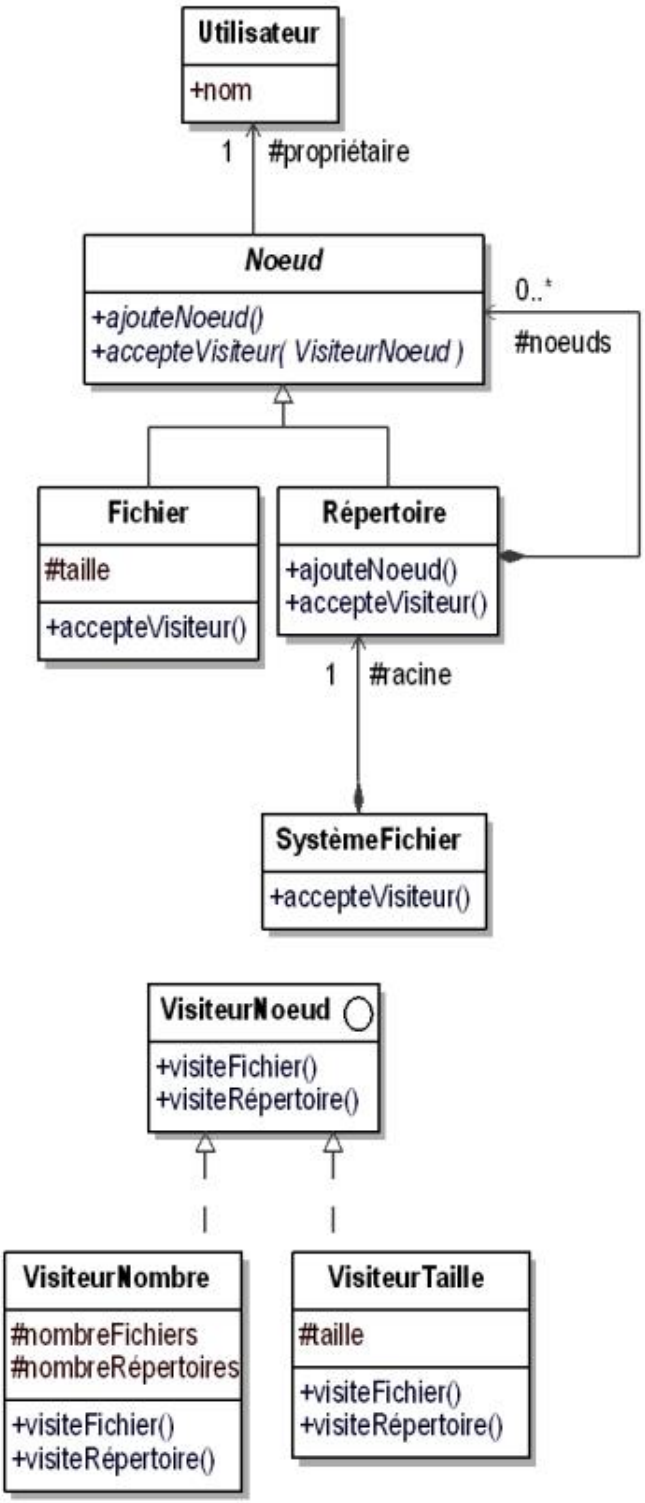
1. Un système de fichiers est basé sur une composition récursive. L'utilisation du pattern `Composite` est totalement adaptée pour modéliser un tel système.

2. Le diagramme des classes donnant la modélisation du système de fichiers se trouve à la suite. Un répertoire est composé de nœuds qui peuvent être soit des fichiers soit des répertoires. Le diagramme montre également la relation entre le système des fichiers et le répertoire "racine".



3. Le pattern qui ajoute des fonctionnalités à des objets existants en nécessitant le minimum de modifications est le pattern `Visitor`. C'est donc lui qui est le plus adapté pour réaliser des opérations sur le système de fichiers.

4. Le diagramme de classes après intégration du pattern `Visitor` est donné ci-dessous. Pour accepter les visiteurs, la méthode `accepteVisiteur` a été introduite dans les classes `Noeud`, `Fichier`, `Répertoire` et `SystèmeFichier`. La méthode `accepteVisiteur` de `SystèmeFichier` invoque la méthode `accepteVisiteur` de la racine. Cette dernière invoque la méthode `visiteRépertoire` du visiteur puis la méthode `accepteVisiteur` de chaque nœud. Ce fonctionnement est illustré dans le programme Java dont le détail est fourni à la suite. Les deux classes de visiteur ont un comportement différent : la classe `VisiteurNombre` compte le nombre de fichiers et de répertoires tandis que la classe `VisiteurTaille` compte la taille globale occupée par le système de fichiers.



5. Le code source des différentes classes Java est fourni à la suite. La classe `Test` introduit un programme de test. Celui-ci crée un sous-répertoire de la racine qui contient deux fichiers de taille 100 et 200 puis crée au niveau de la racine trois fichiers de taille 1000, 2000 et 3000. Le nombre de répertoires est donc de deux, le nombre de fichiers de trois et la taille totale de 6300 comme le montre l'exécution du programme principal.

```
public abstract class Noeud
{
  protected Utilisateur utilisateur;

  public Noeud(Utilisateur utilisateur)
  {
    this.utilisateur = utilisateur;
  }
```

```
  public abstract boolean ajouteNoeud(Noeud noeud);
  public abstract void accepteVisiteur(VisiteurNoeud
    visiteur);
}


public class Fichier extends Noeud
{
  protected int taille;

  public Fichier(Utilisateur utilisateur, int taille)
  {
    super(utilisateur);
    this.taille = taille;
  }

  public int getTaille()
  {
    return taille;
  }

  public void accepteVisiteur(VisiteurNoeud visiteur)
  {
    visiteur.visiteFichier(this);
  }

  public boolean ajouteNoeud(Noeud noeud)
  {
    return false;
  }

}

import java.util.*;
public class Repertoire extends Noeud
{
  protected List<Noeud> noeuds =
    new ArrayList<Noeud>();

  public Repertoire(Utilisateur utilisateur)
  {
    super(utilisateur);
  }

  public void accepteVisiteur(VisiteurNoeud visiteur)
  {
    visiteur.visiteRepertoire(this);
    for (Noeud noeud: noeuds)
    {
      noeud.accepteVisiteur(visiteur);
    }
  }

  public boolean ajouteNoeud(Noeud noeud)
  {
    return noeuds.add(noeud);
  }

}

public class Utilisateur
{
  protected String nom;

  public Utilisateur(String nom)
  {
    this.nom = nom;
  }
```

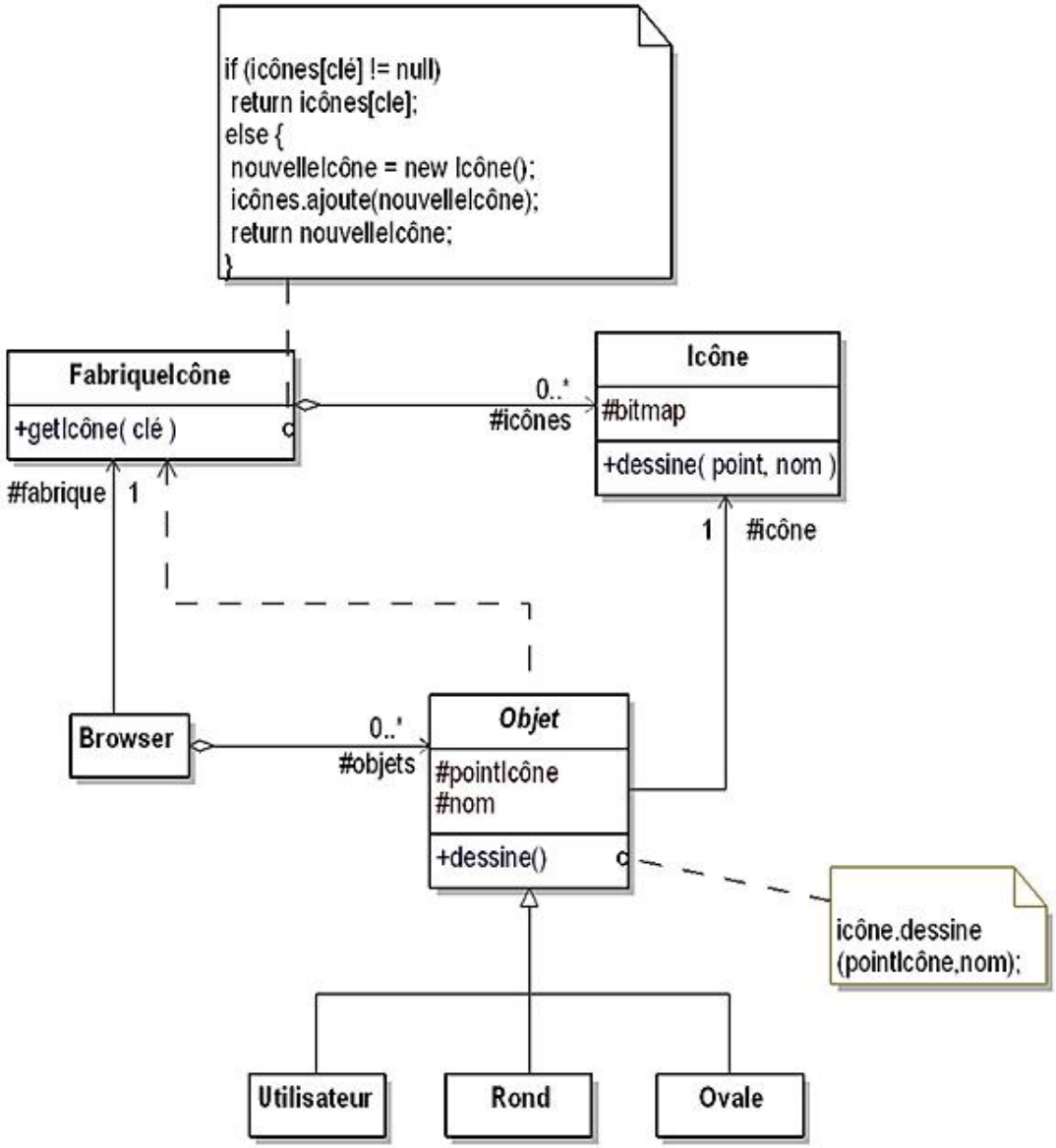
```
  public String getNom()
  {
    return nom;
  }
}

public class SystemeFichier
{
  protected Repertoire racine;

  public SystemeFichier(Repertoire racine)
  {
    this.racine = racine;
  }

  public void accepteVisiteur(VisiteurNoeud visiteur)
  {
    racine.accepteVisiteur(visiteur);
  }
}

public abstract class VisiteurNoeud
{
  public abstract void visiteFichier(Fichier fichier);
  public abstract void visiteRepertoire(Repertoire
    repertoire);
}

public class VisiteurNombre extends VisiteurNoeud
{
  protected int nombreFichiers = 0;
  protected int nombreRepertoires = 0;

  public void visiteFichier(Fichier fichier)
  {
    nombreFichiers = nombreFichiers + 1;
  }

  public void visiteRepertoire(Repertoire repertoire)
  {
    nombreRepertoires = nombreRepertoires + 1;
  }

  public int getNombreFichiers()
  {
    return nombreFichiers;
  }

  public int getNombreRepertoires()
  {
    return nombreRepertoires;
  }

}

public class VisiteurTaille extends VisiteurNoeud
{
  protected int tailleTotale = 0;

  public void visiteFichier(Fichier fichier)
  {
    tailleTotale = tailleTotale + fichier.getTaille();
  }

  public void visiteRepertoire(Repertoire repertoire){}

  public int getTailleTotale()
  {
    return tailleTotale;
```

```
  }

}

public class Test
{

  public static void main(String[] args)
  {
    Utilisateur moiMeme = new Utilisateur("moi même");
    Repertoire racine = new Repertoire(moiMeme);
    Repertoire rep = new Repertoire(moiMeme);
    racine.ajouteNoeud(rep);
    racine.ajouteNoeud(new Fichier(moiMeme, 100));
    racine.ajouteNoeud(new Fichier(moiMeme, 200));
    rep.ajouteNoeud(new Fichier(moiMeme, 1000));
    rep.ajouteNoeud(new Fichier(moiMeme, 2000));
    rep.ajouteNoeud(new Fichier(moiMeme, 3000));
    SystemeFichier systemeFichier = new SystemeFichier
      (racine);
    VisiteurNombre visiteurNombre = new VisiteurNombre();
    systemeFichier.accepteVisiteur(visiteurNombre);
    System.out.println(
      "Nombre de fichiers du système de fichiers : " +
      visiteurNombre.getNombreFichiers());
    System.out.println(
      "Nombre de répertoires du système de fichiers : " +
      visiteurNombre.getNombreRepertoires());
    VisiteurTaille visiteurTaille = new VisiteurTaille();
    systemeFichier.accepteVisiteur(visiteurTaille);
    System.out.println("Taille du système de fichiers : "
      + visiteurTaille.getTailleTotale());
  }
}

Nombre de fichiers du système de fichiers : 5
Nombre de répertoires du système de fichiers : 2
Taille du système de fichiers : 6300
```

4. Browser graphique d'objets

1. Il existe de nombreux objets et donc de nombreux icônes à afficher. Ces icônes doivent être partagés car ils sont de grain fin et il n'y a aucun intérêt à dupliquer les bitmaps associées aux icônes. Le pattern adapté pour partager de nombreux objets de grain fin est le pattern `Flyweight`.

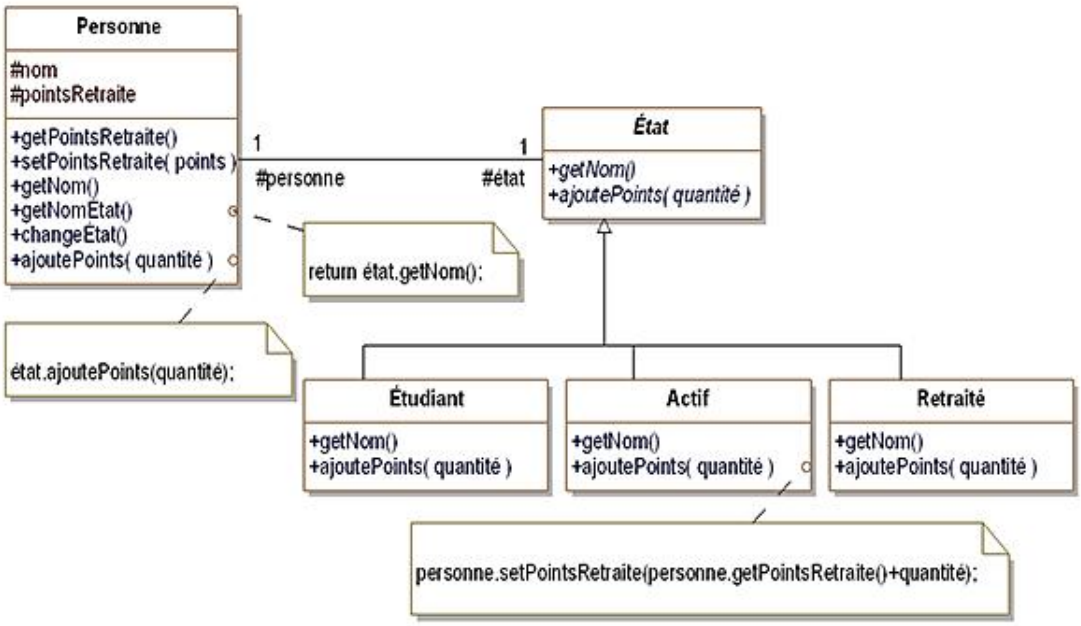
2. Le diagramme de classes du browser est illustré à la suite. Le browser est constitué d'un ensemble d'objets, chacun étant lié, lors de sa création, à un icône partagé. L'état intrinsèque des icônes est constitué de la bitmap. L'état extrinsèque est constitué par les deux paramètres de la méthode dessine, à savoir `point` et `nom`. Le premier paramètre donne l'endroit où il faut dessiner l'icône et le second le titre à imprimer sous l'icône.



5. États de la vie professionnelle d'une personne

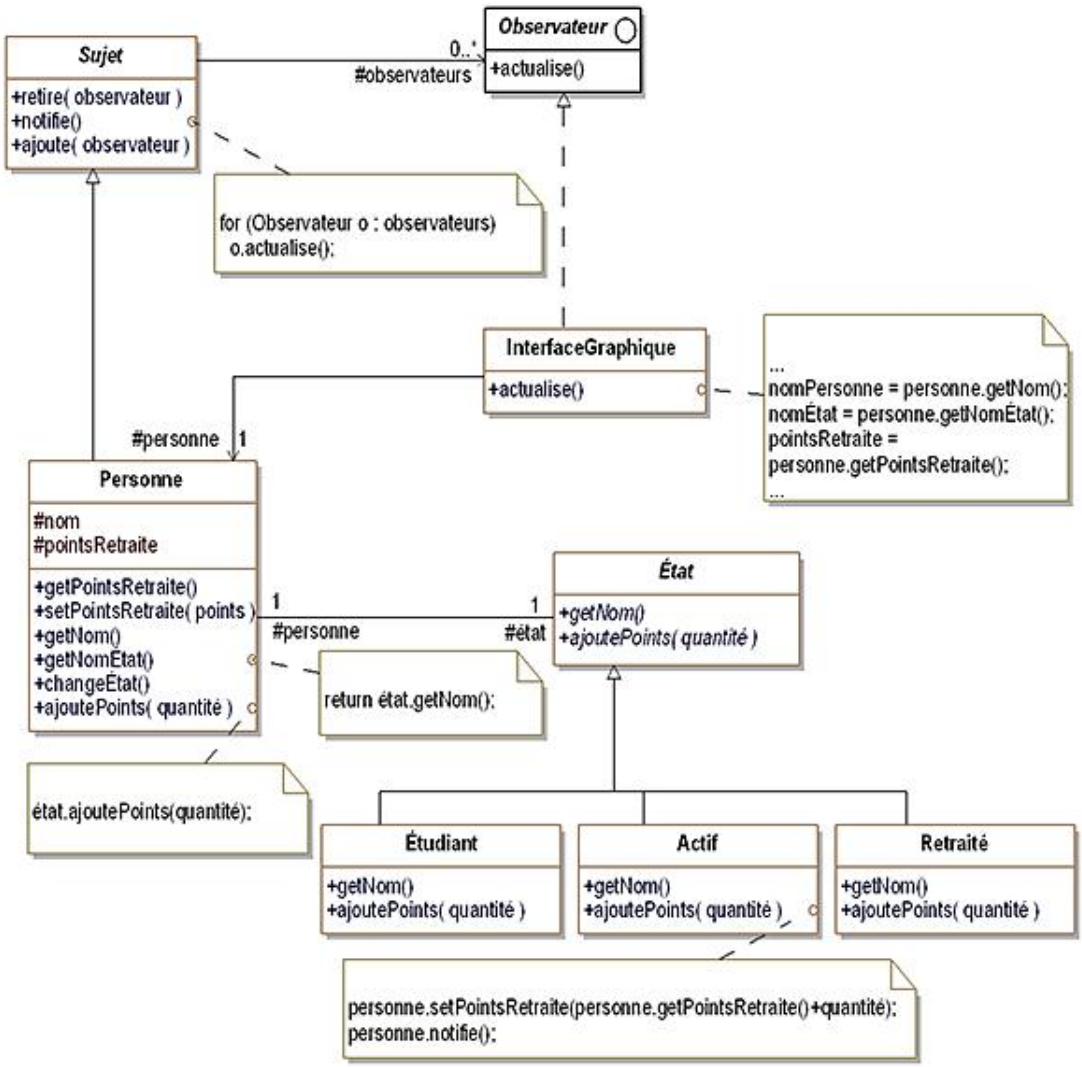
1. Le pattern le mieux adapté pour représenter les différents états de la vie professionnelle d'une personne est le pattern `State`. Il permet d'adapter le comportement des méthodes en fonction de l'état de l'objet.

2. Le diagramme des classes décrivant les différents états est illustré à la figure suivante. La méthode `ajoutePoints` qui ajoute des points de retraite ne possède un comportement que dans la vie active. Elle utilise la méthode `setPointsRetraite` de la classe `Personne` dont l'usage est réservé à la classe `État` et à ses sous-classes.



3. Le pattern `Observer` permet de réaliser des mises à jour automatiques entre plusieurs objets. Il est donc possible de l'utiliser dans le cadre de cet exercice.

4. Le diagramme des classes basé sur le pattern `Observer` et intégrant le diagramme précédent se trouve à la suite. La méthode `ajoutePoints` de la classe `Actif` invoque la méthode `notifie` de la personne pour informer les observateurs du changement de valeur du nombre de points.



6. Cache d'un dictionnaire persistant d'objets

Le code source du proxy qui implante le cache mémoire d'accès au dictionnaire persistant est donné à la suite. Son implantation est très simple :

- la méthode `ajoute` introduit dans l'attribut `contenu` la nouvelle valeur ajoutée ;
- la méthode `get` recherche dans l'attribut `contenu`. Si la valeur n'est pas trouvée, elle est recherchée dans le dictionnaire persistant et si elle est trouvée, elle est ajoutée à l'attribut `contenu`.

```
import java.util.*;
public class DictPersistantProxy<T> implements
  DictPersistantIntf<T>
{
  protected DictPersistant<T> dictPersistant =
    new DictPersistant<T>();
  protected Map<String, T> contenu =
    new TreeMap<String, T>();

  public boolean ajoute(String cle, T objet)
  {
    boolean resultat = dictPersistant.ajoute(cle, objet);
    if (resultat)
      contenu.put(cle, objet);
    return resultat;
```

```
  }

  public T get(String cle)
  {
    T resultat;
    resultat = contenu.get(cle);
    if (resultat == null)
    {
      resultat = dictPersistant.get(cle);
      if (resultat != null)
      {
        contenu.put(cle, resultat);
      }
    }
    return resultat;
  }
}
```

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 `FabriqueVéhiculeÉlectricité` 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 `FabriqueVéhiculeEssence`. Avec une telle instance, le catalogue peut créer et manipuler des véhicules sans devoir connaître les familles de véhicule 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.

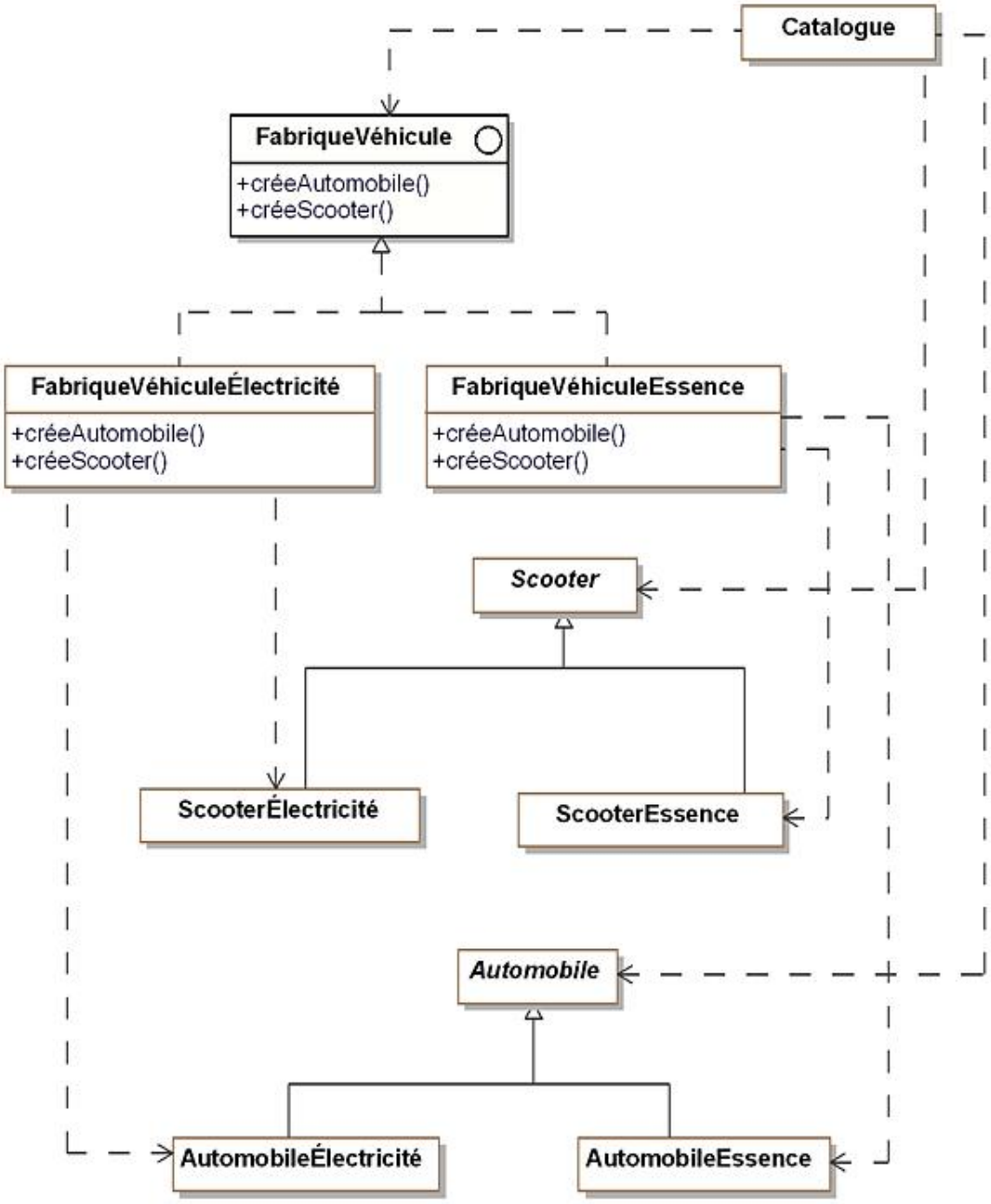


Figure 4.1 - Le pattern `Abstract Factory` appliqué à des familles de véhicules

Structure

1. Diagramme de classes

La figure 4.2 détaille la structure générique du pattern.

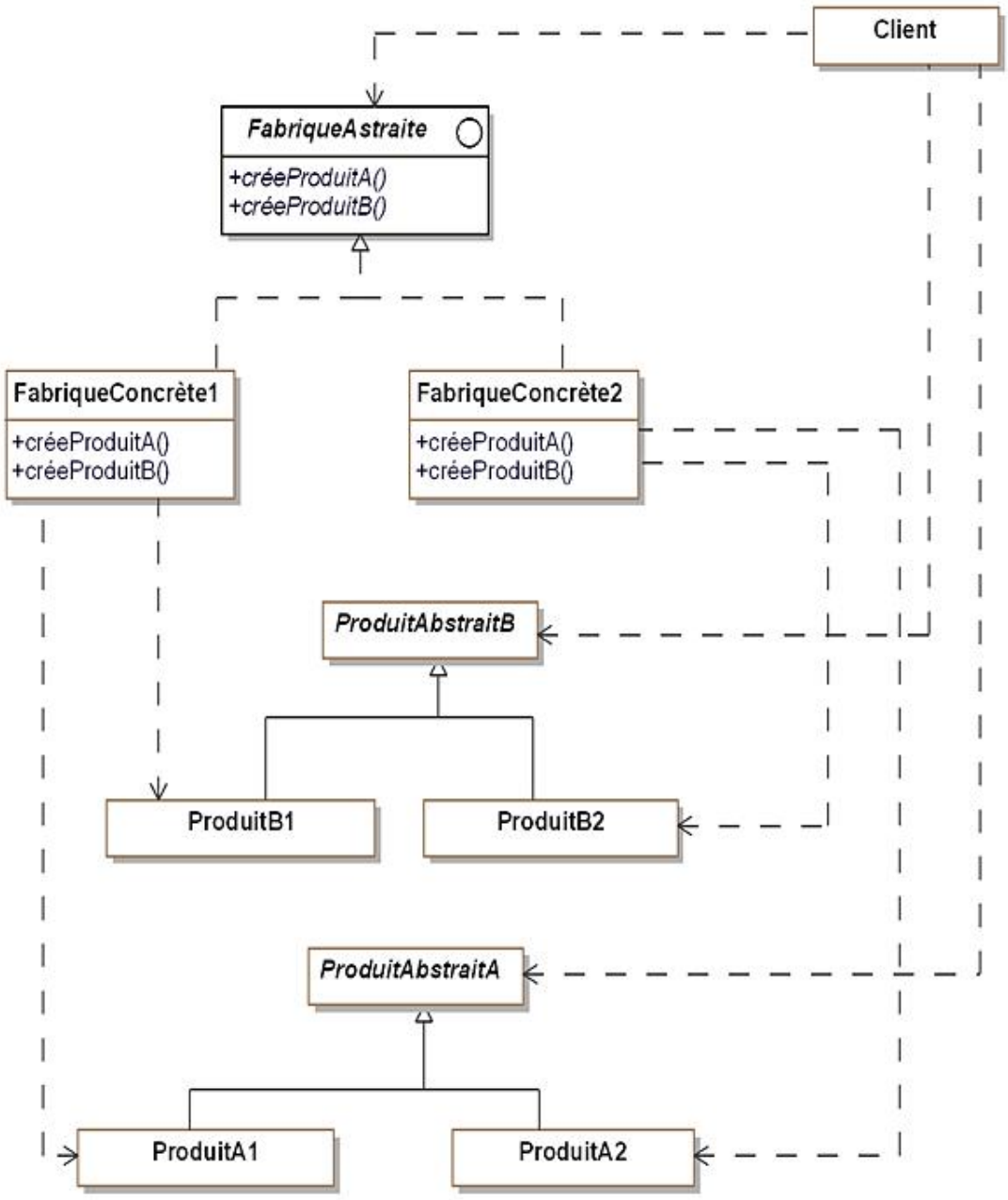


Figure 4.2 - Structure du pattern `Abstract Factory`

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` (`FabriqueVéhiculeÉlectricité`, `FabriqueVéhiculeEssence`) sont les classes concrètes implantant les méthodes créant les produits pour chaque famille de produit. 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. Collaborations

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

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

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.

Exemple en Java

Nous introduisons maintenant un petit exemple d'utilisation du pattern écrit en Java. Le code Java 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 `afficheCaracteristique` qui permet de les afficher.

```
public abstract class Automobile
{
  protected String modele;
  protected String couleur;
  protected int puissance;
  protected double espace;

  public Automobile(String modele, String couleur, int
    puissance, double espace)
  {
    this.modele = modele;
    this.couleur = couleur;
    this.puissance = puissance;
    this.espace = espace;
  }

  public abstract void afficheCaracteristiques();
}

public class AutomobileElectricite extends Automobile
{
  public AutomobileElectricite(String modele, String
    couleur, int puissance, double espace)
  {
    super(modele, couleur, puissance, espace);
  }

  public void afficheCaracteristiques()
  {
    System.out.println(
      "Automobile électrique de modèle : " + modele +
      " de couleur : " + couleur + " de puissance : " +
      puissance + " d'espace : " + espace);
  }
}

public class AutomobileEssence extends Automobile
{
  public AutomobileEssence(String modele, String couleur,
    int puissance, double espace)
  {
    super(modele, couleur, puissance, espace);
  }

  public void afficheCaracteristiques()
  {
    System.out.println(
      "Automobile à essence de modèle : " + modele +
      " de couleur : " + couleur + " de puissance : " +
      puissance + " d'espace : " + espace);
  }
}
```

Le code Java correspondant à la classe abstraite `Scooter` et ses sous-classes est donné à la suite. Il est similaire à celui des automobiles, à l'exception de l'attribut `espace` qui n'existe pas pour les scooters.

```
public abstract class Scooter
{
  protected String modele;
  protected String couleur;
  protected int puissance;
```

```
  public Scooter(String modele, String couleur, int
    puissance)
  {
    this.modele = modele;
    this.couleur = couleur;
    this.puissance = puissance;
  }
  public abstract void afficheCaracteristiques();
}

public class ScooterElectricite extends Scooter
{
  public ScooterElectricite(String modele, String couleur,
    int puissance)
  {
    super(modele, couleur, puissance);
  }

  public void afficheCaracteristiques()
  {
    System.out.println("Scooter électrique de modèle : "
      + modele + " de couleur : " + couleur +
      " de puissance : " + puissance);
  }

}

public class ScooterEssence extends Scooter
{
  public ScooterEssence(String modele, String couleur,
    int puissance)
  {
    super(modele, couleur, puissance);
  }

  public void afficheCaracteristiques()
  {
    System.out.println("Scooter à essence de modèle : " +
      modele + " de couleur : " + couleur +
      " de puissance : " + puissance);
  }

}
```

Nous pouvons maintenant introduire l'interface `FabriqueVehicule` et ses deux classes d'implantation, une pour chaque famille (électricité/essence). Il est aisé de voir que seules les classes d'implantation utilisent les classes concrètes des véhicules.

```
public interface FabriqueVehicule
{
  Automobile creeAutomobile(String modele, String couleur,
    int puissance, double espace);

  Scooter creeScooter(String modele, String couleur, int
    puissance);
}

public class FabriqueVehiculeElectricite implements
  FabriqueVehicule
{
  public Automobile creeAutomobile(String modele, String
    couleur, int puissance, double espace)
  {
    return new AutomobileElectricite(modele, couleur,
      puissance, espace);
  }

  public Scooter creeScooter(String modele, String
    couleur, int puissance)
```

```
  {
    return new ScooterElectricite(modele, couleur,
      puissance);
  }
}

public class FabriqueVehiculeEssence implements
  FabriqueVehicule
{
  public Automobile creeAutomobile(String modele, String
    couleur, int puissance, double espace)
  {
    return new AutomobileEssence(modele, couleur,
      puissance, espace);
  }

  public Scooter creeScooter(String modele, String
    couleur, int puissance)
  {
    return new ScooterEssence(modele, couleur, puissance);
  }
}
```

Enfin, nous donnons le code source Java du client de la fabrique, à savoir le catalogue qui est, dans notre exemple, le programme principal. Pour des raisons de simplification, le début de celui-ci consiste à demander la fabrique à utiliser (électricité ou essence). Cette fabrique devrait normalement être fournie en paramètre au catalogue.

Le reste du programme est totalement indépendant de la famille d'objets, respectant le but du pattern `Abstract Factory`.

```
import java.util.*;
public class Catalogue
{
  public static int nbAutos = 3;
  public static int nbScooters = 2;

  public static void main(String[] args)
  {
    Scanner reader = new Scanner(System.in);
    FabriqueVehicule fabrique;
    Automobile[] autos = new Automobile[nbAutos];
    Scooter[] scooters = new Scooter[nbScooters];
    System.out.print("Voulez-vous utiliser des " +
      "Véhicules électriques (1) ou à essence (2) :");
    String choix = reader.next();
    if (choix.equals("1"))
    {
      fabrique = new FabriqueVehiculeElectricite();
    }
    else
    {
      fabrique = new FabriqueVehiculeEssence();
    }
    for (int index = 0; index < nbAutos; index++)
      autos[index] = fabrique.creeAutomobile("standard",
        "jaune", 6+index, 3.2);
    for (int index = 0; index < nbScooters; index++)
      scooters[index] = fabrique.creeScooter("classic",
        "rouge", 2+index);
    for (Automobile auto: autos)
      auto.afficheCaracteristiques();
    for (Scooter scooter: scooters)
      scooter.afficheCaracteristiques();
  }
}
```

Un exemple d'exécution pour des véhicules à électricité est le suivant :

```
Voulez-vous utiliser des véhicules électriques (1) ou
à essence (2) :1
```

```
Automobile électrique de modèle : standard de couleur :
jaune de puissance : 6 d’espace : 3.2
Automobile électrique de modèle : standard de couleur :
jaune de puissance : 7 d’espace : 3.2
Automobile électrique de modèle : standard de couleur :
jaune de puissance : 8 d’espace : 3.2
Scooter électrique de modèle : classic de couleur :
rouge de puissance : 2
Scooter électrique de modèle :
classic de couleur : rouge de puissance : 3
```

Dans cet exemple d’exécution, c’est une fabrique de véhicules électriques qui a été créée, ainsi le catalogue construit des automobiles et des scooters électriques.

Description

Le but du pattern `Builder` est d'abstraire la construction d'objets complexes de leur implantation de sorte qu'un client puisse créer ces objets complexes sans devoir se préoccuper des différences d'implantation.

Exemple

Lors de l'achat d'un véhicule, un vendeur crée une liasse de documents comprenant notamment le bon de commande et la demande d'immatriculation du client. Il peut construire ces documents au format HTML ou au format PDF selon le choix du client. Dans le premier cas, le client lui fournit une instance de la classe `CréateurLiasseVéhiculeHtml` et, dans le second cas, une instance de la classe `CréateurLiasseVéhiculePdf`. Le vendeur effectue ensuite la demande de création de chaque document de la liasse à cette instance.

Ainsi le vendeur crée les documents de la liasse à l'aide des méthodes `construitBonDeCommande` et `construitDemandeImmatriculation`.

L'ensemble des classes du pattern `Builder` pour cet exemple est détaillé à la figure 5.1. Cette figure montre la hiérarchie des classes `ConstructeurLiasseVéhicule` et `Liasse`. Le vendeur peut créer les bons de commande et les demandes d'immatriculation sans connaître les sous-classes de `ConstructeurLiasseVéhicule` ni celles de `Liasse`.

Les relations de dépendance entre le client et les sous-classes de `ConstructeurLiasseVéhicule` s'expliquent par le fait que le client crée une instance de ces sous-classes.

La structure interne des sous-classes concrètes de `Liasse` n'est pas montrée (dont, par exemple, la relation de composition avec la classe `Document`).

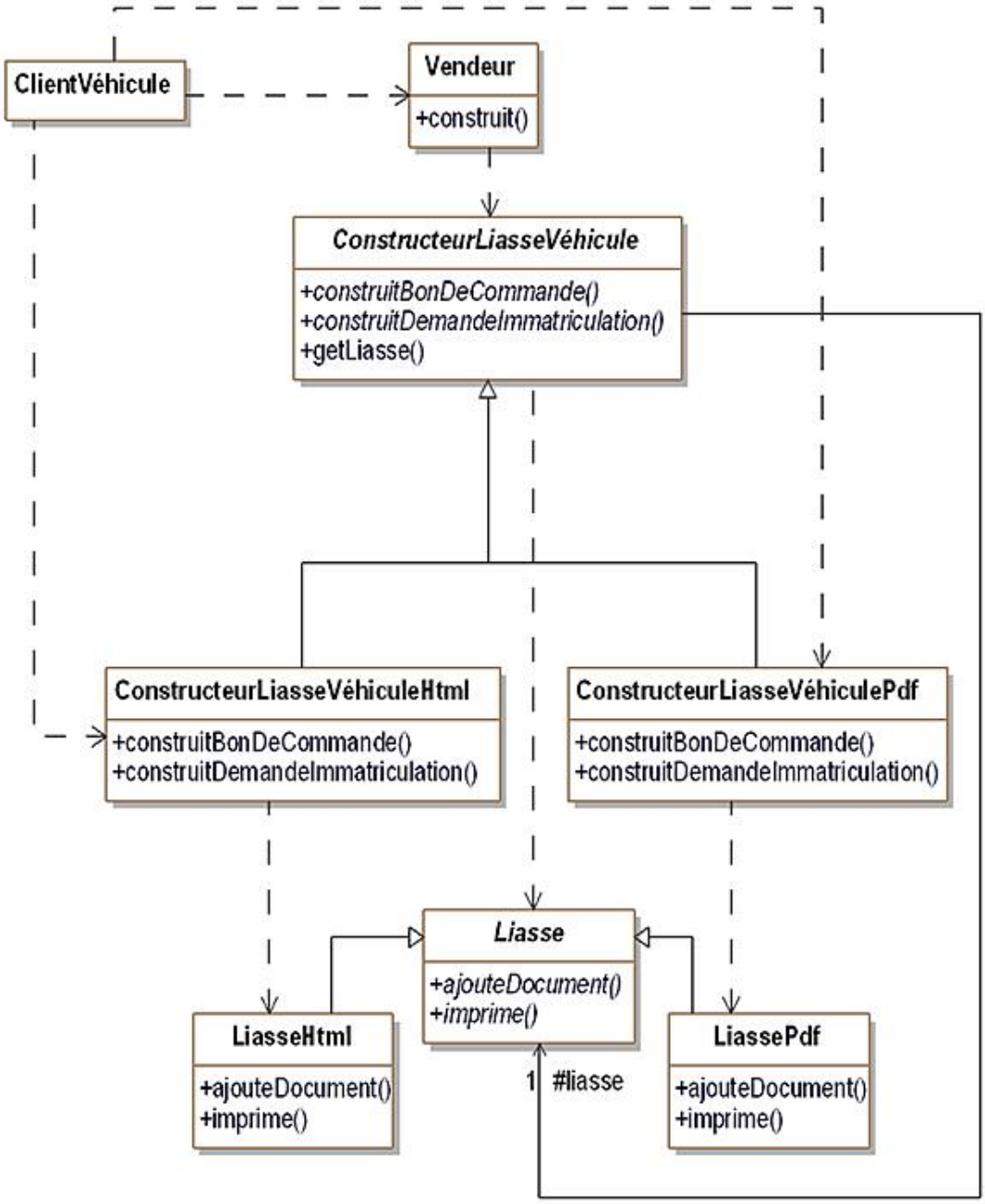


Figure 5.1 - Le pattern `Builder` appliqué à des liasses de documents

Structure

1. Diagramme de classes

La figure 5.2 détaille la structure générique du pattern.

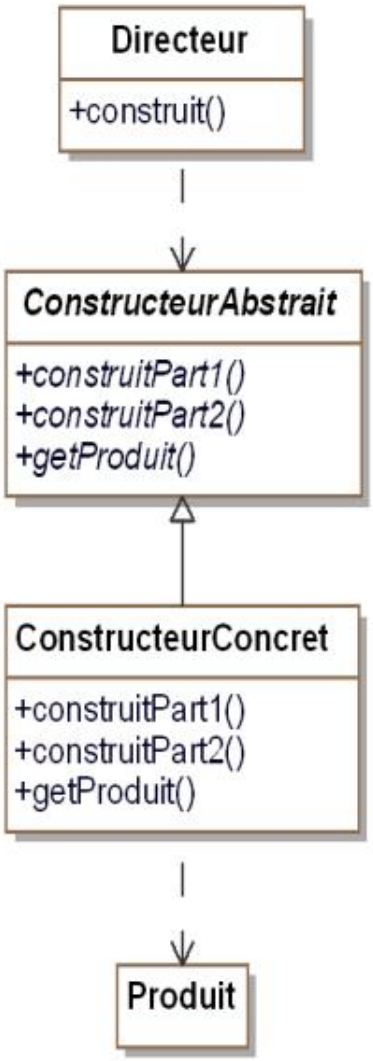


Figure 5.2 - Structure du pattern `Builder`

2. Participants

Les participants au pattern sont les suivants :

- `ConstructeurAbstrait` (`ConstructeurLiasseVéhicule`) est la classe introduisant les signatures des méthodes construisant les différentes parties du produit ainsi que la signature de la méthode permettant d'obtenir le produit, une fois celui-ci construit ;
- `ConstructeurConcret` (`ConstructeurLiasseVéhiculeHtmlet ConstructeurLiasseVéhiculePdf`) est la classe concrète implantant les méthodes du constructeur abstrait ;
- `Produit` (`Liasse`) est la classe définissant le produit. Elle peut être abstraite et posséder plusieurs sous-classes concrètes (`LiasseHtml` et `LiassePdf`) en cas d'implantations différentes ;

- `Directeur`est la classe chargée de construire le produit au travers de l'interface du constructeur abstrait.

3. Collaborations

Le client crée un constructeur concret et un directeur. Le directeur construit sur demande du client en invoquant le constructeur et renvoie le résultat au client. La figure 5.3 illustre ce fonctionnement avec un diagramme de séquence UML.

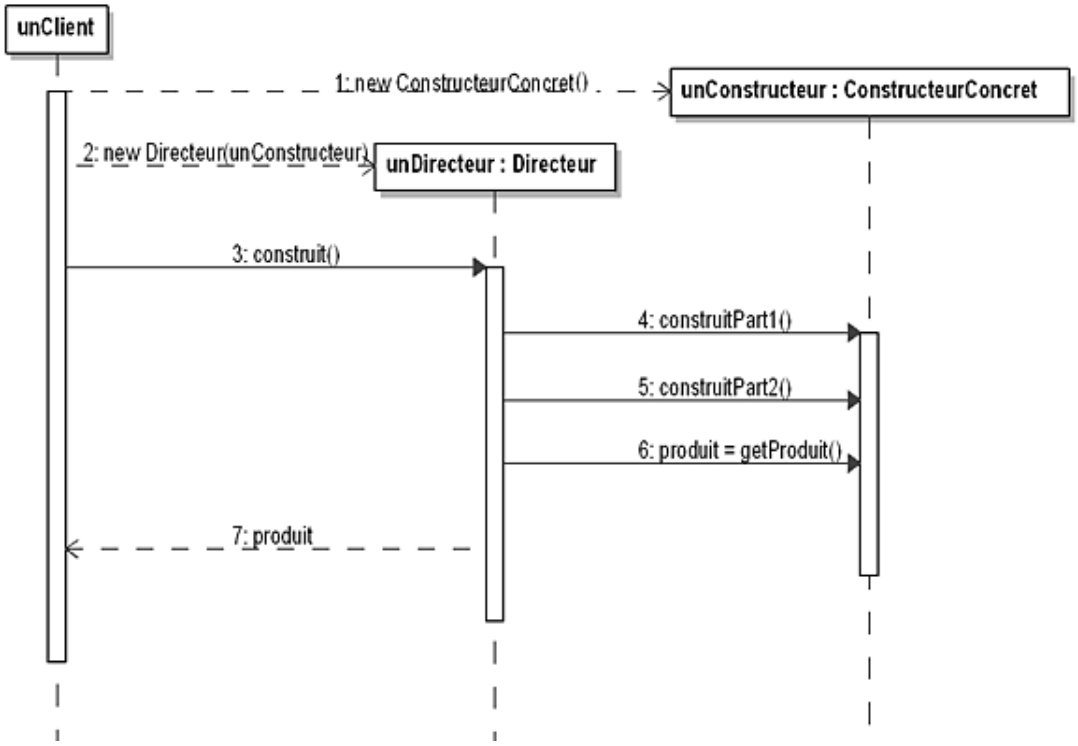


Figure 5.3 - Diagramme de séquence du pattern `Builder`

Domaines d’utilisation

Le pattern est utilisé dans les domaines suivants :

- un client a besoin de construire des objets complexes sans connaître leur implantation ;
- un client a besoin de construire des objets complexes ayant plusieurs représentations ou implantations.

Exemple en Java

Nous introduisons le petit exemple d'utilisation du pattern écrit en Java. Le code Java correspondant à la classe abstraite `Liasse` et ses sous-classes est donné à la suite. Par souci de simplification, les documents sont des chaînes de caractères pour les liasses au format HTML et PDF. La méthode `imprime` affiche les différentes chaînes de caractères qui représentent les documents.

```
import java.util.*;

public abstract class Liasse
{
  protected List<String> contenu =
          new ArrayList<String>();

  public abstract void ajouteDocument(String document);
  public abstract void imprime();
}

public class LiasseHtml extends Liasse
{
  public void ajouteDocument(String document)
  {
    if (document.startsWith("<HTML>"))
      contenu.add(document);
  }

  public void imprime()
  {
    System.out.println("Liasse HTML");
    for (String s: contenu)
      System.out.println(s);
  }
}

public class LiassePdf extends Liasse
{
  public void ajouteDocument(String document)
  {
    if (document.startsWith("<PDF>"))
      contenu.add(document);
  }

  public void imprime()
  {
    System.out.println("Liasse PDF");
    for (String s: contenu)
      System.out.println(s);
  }
}
```

Le code source des classes qui construisent les liasses est fourni à la suite.

```
public abstract class ConstructeurLiasseVehicule
{
  protected Liasse liasse;

  public abstract void construitBonDeCommande(String
    nomClient);

  public abstract void construitDemandeImmatriculation
    (String nomDemandeur);

  public Liasse resultat()
  {
    return liasse;
  }
}
```

```
public class ConstructeurLiasseVehiculeHtml extends
  ConstructeurLiasseVehicule
{
  public ConstructeurLiasseVehiculeHtml()
  {
    liasse = new LiasseHtml();
  }

  public void construitBonDeCommande(String nomClient)
  {
    String document;
    document = "<HTML>Bon de commande Client : " +
      nomClient + "</HTML>";
    liasse.ajouteDocument(document);
  }

  public void construitDemandeImmatriculation(String
    nomDemandeur)
  {
    String document;
    document =
      "<HTML>Demande d'immatriculation Demandeur : " +
      nomDemandeur + "</HTML>";
    liasse.ajouteDocument(document);
  }
}

public class ConstructeurLiasseVehiculePdf extends
  ConstructeurLiasseVehicule
{
  public ConstructeurLiasseVehiculePdf()
  {
    liasse = new LiassePdf();
  }

  public void construitBonDeCommande(String nomClient)
  {
    String document;
    document = "<PDF>Bon de commande Client : " +
      nomClient + "</PDF>";
    liasse.ajouteDocument(document);
  }

  public void construitDemandeImmatriculation(String
    nomDemandeur)
  {
    String document;
    document =
      "<PDF>Demande d'immatriculation Demandeur : " +
      nomDemandeur + "</PDF>";
    liasse.ajouteDocument(document);
  }
}
```

La classe `Vendeur` est décrite à la suite. Son constructeur prend comme paramètre une instance de `ConstructeurLiasseVehicule`. Il convient de noter que la méthode `construit` prend comme paramètre les informations du client, ici limitées au nom du client.

```
public class Vendeur
{
  protected ConstructeurLiasseVehicule constructeur;

  public Vendeur(ConstructeurLiasseVehicule constructeur)
  {
    this.constructeur = constructeur;
  }

  public Liasse construit(String nomClient)
  {
```

```
    constructeur.construitBonDeCommande(nomClient);
    constructeur.construitDemandeImmatriculation
      (nomClient);
    Liasse liasse = constructeur.resultat();
    return liasse;
  }
}
```

Enfin, nous donnons le code source Java du client du constructeur, à savoir la classe `ClientVehicule` qui constitue le programme principal. Le début de ce programme demande à l'utilisateur le constructeur à utiliser qui est fourni au vendeur.

```
public class ClientVehicule
{
  public static void main(String[] args)
  {
    Scanner reader = new Scanner(System.in);
    ConstructeurLiasseVehicule constructeur;
    System.out.print("Voulez-vous construire des" +
      "liasses HTML (1) ou PDF (2) :");
    String choix = reader.next();
    if (choix.equals("1"))
    {
      constructeur = new ConstructeurLiasseVehiculeHtml();
    }
    else
    {
      constructeur = new ConstructeurLiasseVehiculePdf();
    }
    Vendeur vendeur = new Vendeur(constructeur);
    Liasse liasse = vendeur.construit("Martin");
    liasse.imprime();
  }
}
```

Un exemple d'exécution pour une liasse en PDF est le suivant :

```
Voulez-vous construire des liasses HTML (1) ou PDF (2) :2
Liasse PDF
<PDF>Bon de commande Client : Martin</PDF>
<PDF>Demande d'immatriculation Demandeur : Martin</PDF>
```

Conformément à la demande du client, la liasse et ses documents ont été créés au format PDF. Si le client demande une liasse HTML, la sortie est la suivante :

```
Voulez-vous construire des liasses HTML (1) ou PDF (2) :1
Liasse HTML
<HTML>Bon de commande Client : Martin</HTML>
<HTML>Demande d'immatriculation Demandeur : Martin</HTML>
```

Description

Le but du pattern `Factory Method` est d'introduire une méthode abstraite de création d'un objet en reportant aux sous-classes concrètes la création effective.

Exemple

Nous nous intéressons aux clients et aux commandes. La classe `Client` introduit la méthode `créeCommande` qui doit créer la commande. Certains clients commandent un véhicule en payant au comptant et d'autres clients utilisent un crédit. En fonction de la nature du client, la méthode `créeCommande` doit créer une instance de la classe `CommandeComptant` ou une instance de la classe `CommandeCrédit`. Pour réaliser cette alternative, la méthode `créeCommande` est abstraite. Les deux types de clients sont distingués en introduisant deux sous-classes concrètes de la classe abstraite `Client` :

- la classe concrète `ClientComptant` dont la méthode `créeCommande` crée une instance de la classe `CommmandeComptant` ;
- la classe concrète `ClientCrédit` dont la méthode `créeCommande` crée une instance de la classe `CommmandeCrédit`.

Une telle conception est basée sur le pattern `Factory Method`, la méthode `créeCommande` étant la méthode de fabrique. L'exemple est détaillé à la figure 6.1.

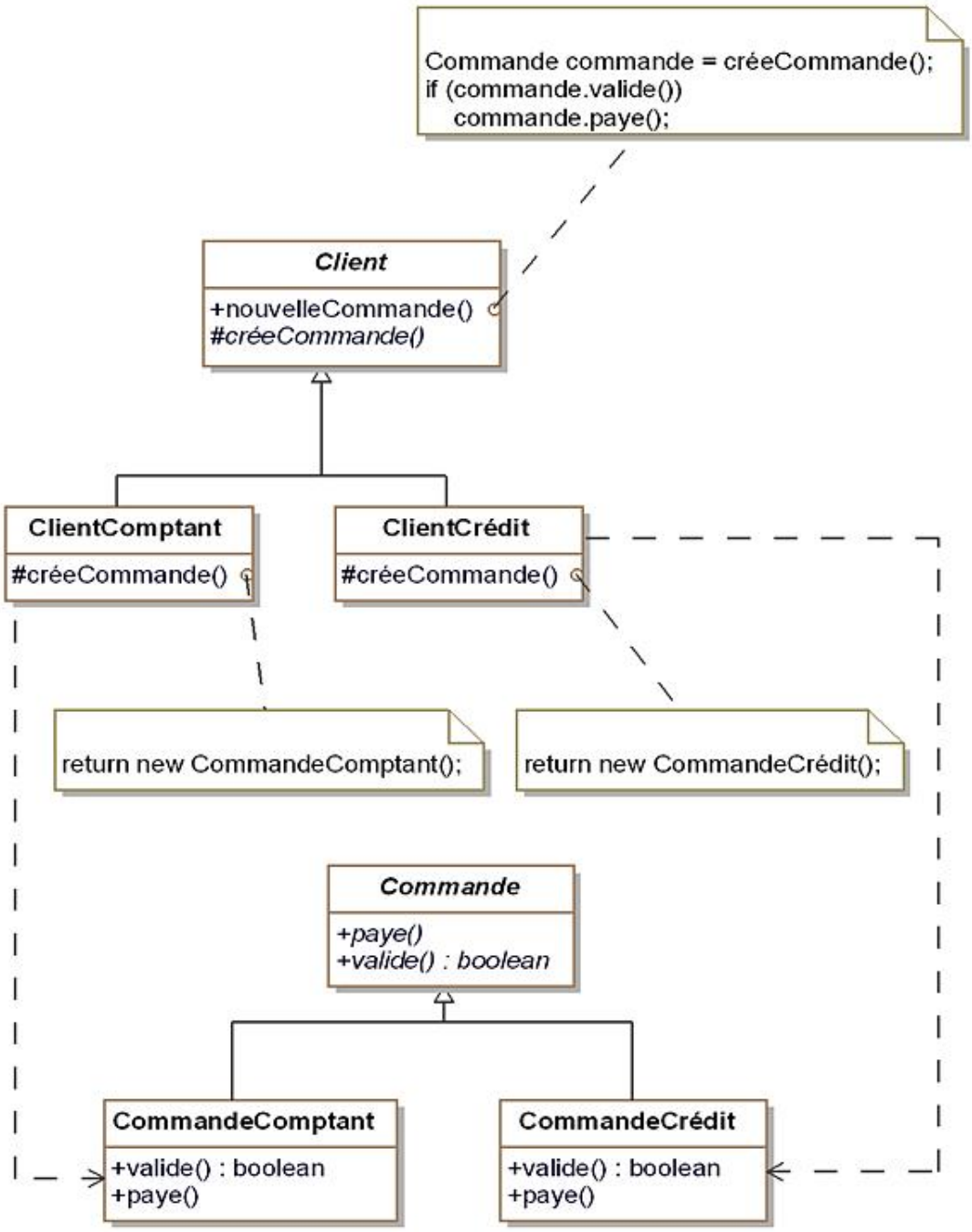


Figure 6.1 - Le pattern `Factory Method` appliqué à des clients et à leurs commandes

Structure

1. Diagramme de classes

La figure 6.2 détaille la structure générique du pattern.

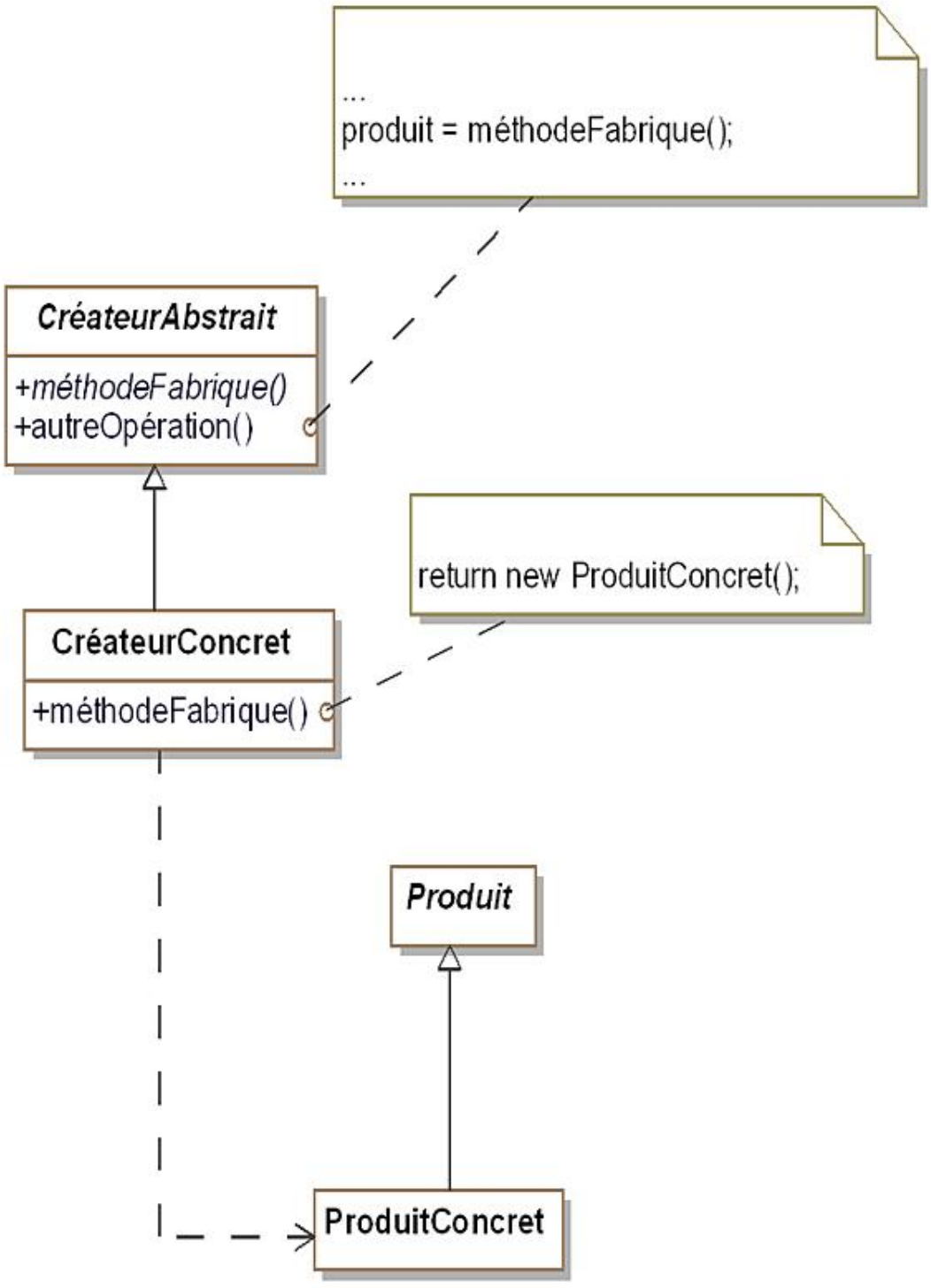


Figure 6.2 - Structure du pattern `Factory Method`

2. Participants

Les participants au pattern sont les suivants :

- `CréateurAbstrait` (`Client`) est une classe abstraite qui introduit la signature de la méthode de fabrique et l'implantation de méthodes qui invoquent la méthode de fabrique ;
- `CréateurConcret` (`ClientComptant`, `ClientCrédit`) est une classe concrète qui implante la méthode de fabrique. Il peut exister plusieurs créateurs concrets ;
- `Produit` (`Commande`) est une classe abstraite décrivant les propriétés communes des produits ;
- `ProduitConcret` (`CommandeComptant`, `CommandeCrédit`) est une classe concrète décrivant complètement un produit.

3. Collaborations

Les méthodes concrètes de la classe `CréateurAbstrait` se basent sur l'implantation de la méthode de fabrique dans les sous-classes. Cette implantation crée une instance de la sous-classe adéquate de `Produit`.

Domaines d’utilisation

Le pattern est utilisé dans les cas suivants :

- une classe ne connaît que les classes abstraites des objets avec lesquels elle possède des relations ;
- une classe veut transmettre à ses sous-classes les choix d’instanciation en profitant du mécanisme de polymorphisme.

Exemple en Java

Le code source de la classe abstraite `Commande` et de ses deux sous-classes concrètes est à la suite. Le montant de la commande est passé en paramètre du constructeur de la classe. Si la validation d'une commande au comptant est systématique, nous avons le choix pour notre exemple de n'accepter que les commandes assorties d'un crédit dont la valeur se situe entre 1000 et 5000.

```
public abstract class Commande
{
  protected double montant;

  public Commande(double montant)
  {
    this.montant = montant;
  }
  public abstract boolean valide();
  public abstract void paye();
}

public class CommandeComptant extends Commande
{
  public CommandeComptant(double montant)
  {
    super(montant);
  }

  public void paye()
  {
    System.out.println(
      "Le paiement de la commande au comptant de : " +
      montant + " est effectué.");
  }

  public boolean valide()
  {
    return true;
  }
}

public class CommandeCredit extends Commande
{
  public CommandeCredit(double montant)
  {
    super(montant);
  }

  public void paye()
  {
    System.out.println(
      "Le paiement de la commande au crédit de : " +
      montant + " est effectué.");
  }

  public boolean valide()
  {
    return (montant >= 1000.0) && (montant <= 5000.0);
  }
}
```

Le code source de la classe abstraite `Client` et de ses deux sous-classes concrètes est fourni à la suite. Un client peut passer plusieurs commandes, seules celles qui sont validées sont ajoutées à sa liste.

```
import java.util.*;
public abstract class Client
{
  protected List<Commande> commandes =
          new ArrayList<Commande>();
```

```
  protected abstract Commande creeCommande(double montant)
    ;

  public void nouvelleCommande(double montant)
  {
    Commande commande = this.creeCommande(montant);
    if (commande.valide())
    {
      commande.paye();
      commandes.add(commande);
    }
  }
}

public class ClientComptant extends Client
{
  protected Commande creeCommande(double montant)
  {
    return new CommandeComptant(montant);
  }
}

public class ClientCredit extends Client
{
  protected Commande creeCommande(double montant)
  {
    return new CommandeCredit(montant);
  }
}
```

Enfin, la classe `Utilisateur` montre un exemple de mise en œuvre du pattern `Factory Method`.

Le nom Utilisateur dénote ici un objet utilisateur d'un pattern.

```
public class Utilisateur
{

  public static void main(String[] args)
  {
    Client client;
    client = new ClientComptant();
    client.nouvelleCommande(2000.0);
    client.nouvelleCommande(10000.0);
    client = new ClientCredit();
    client.nouvelleCommande(2000.0);
    client.nouvelleCommande(10000.0);
  }
}
```

Un exemple d'exécution de l'utilisateur donne la sortie suivante :

```
Le paiement de la commande au comptant de : 2000.0 est effectué.
Le paiement de la commande au comptant de : 10000.0 est effectué.
Le paiement de la commande au crédit de : 2000.0 est effectué.
```

On peut remarquer que la demande d'une commande assortie d'un crédit de 10000 a été refusée.

Description

Le but du pattern est la création de nouveaux objets par duplication d'objets existants appelés prototypes qui disposent de la capacité de clonage.

Exemple

Lors de l'achat d'un véhicule, un client doit recevoir une liasse définie par un nombre précis de documents tels que le certificat de cession, la demande d'immatriculation ou encore le bon de commande. D'autres types de documents peuvent être ajoutés ou retirés à cette liasse en fonction des besoins de gestion ou des changements de réglementation. Nous introduisons une classe `Liasse` dont les instances sont des liasses composées des différents documents nécessaires. Pour chaque type de document, nous introduisons une classe correspondante.

Puis nous créons un modèle de liasse qui est une instance particulière de la classe `Liasse` et qui contient les différents documents nécessaires, documents qui restent vierges. Nous appelons cette liasse, la liasse vierge. Ainsi nous définissons au niveau des instances le contenu précis de la liasse que doit recevoir un client et non au niveau des classes. L'ajout ou la suppression d'un document dans la liasse vierge n'impose pas de modification dans sa classe.

Une fois cette liasse vierge introduite, nous procédons par clonage pour créer les nouvelles liasses. Chaque nouvelle liasse est créée en dupliquant tous les documents de la liasse vierge.

Cette technique basée sur des objets disposant de la capacité de clonage utilise le pattern `Prototype`, les documents constituant les différents prototypes.

La figure 7.1 illustre cette utilisation. La classe `Document` est une classe abstraite connue de la classe `Liasse`. Ses sous-classes correspondent aux différents types de documents. Elles possèdent la méthode `duplique` qui permet de cloner une instance existante pour en obtenir une nouvelle.

La classe `Liasse` est également abstraite. Elle possède deux sous-classes concrètes :

- la classe `LiasseVierge` qui ne possède qu'une seule instance, une liasse contenant tous les documents nécessaires (documents vierges). Cette instance est manipulée au travers des méthodes `ajoute` et `retire`.

- La classe `LiasseClient` dont l'ensemble des documents est créé en demandant à l'unique instance de la classe `LiasseVierge` la liste des documents vierges puis en les ajoutant un à un après les avoir clonés.

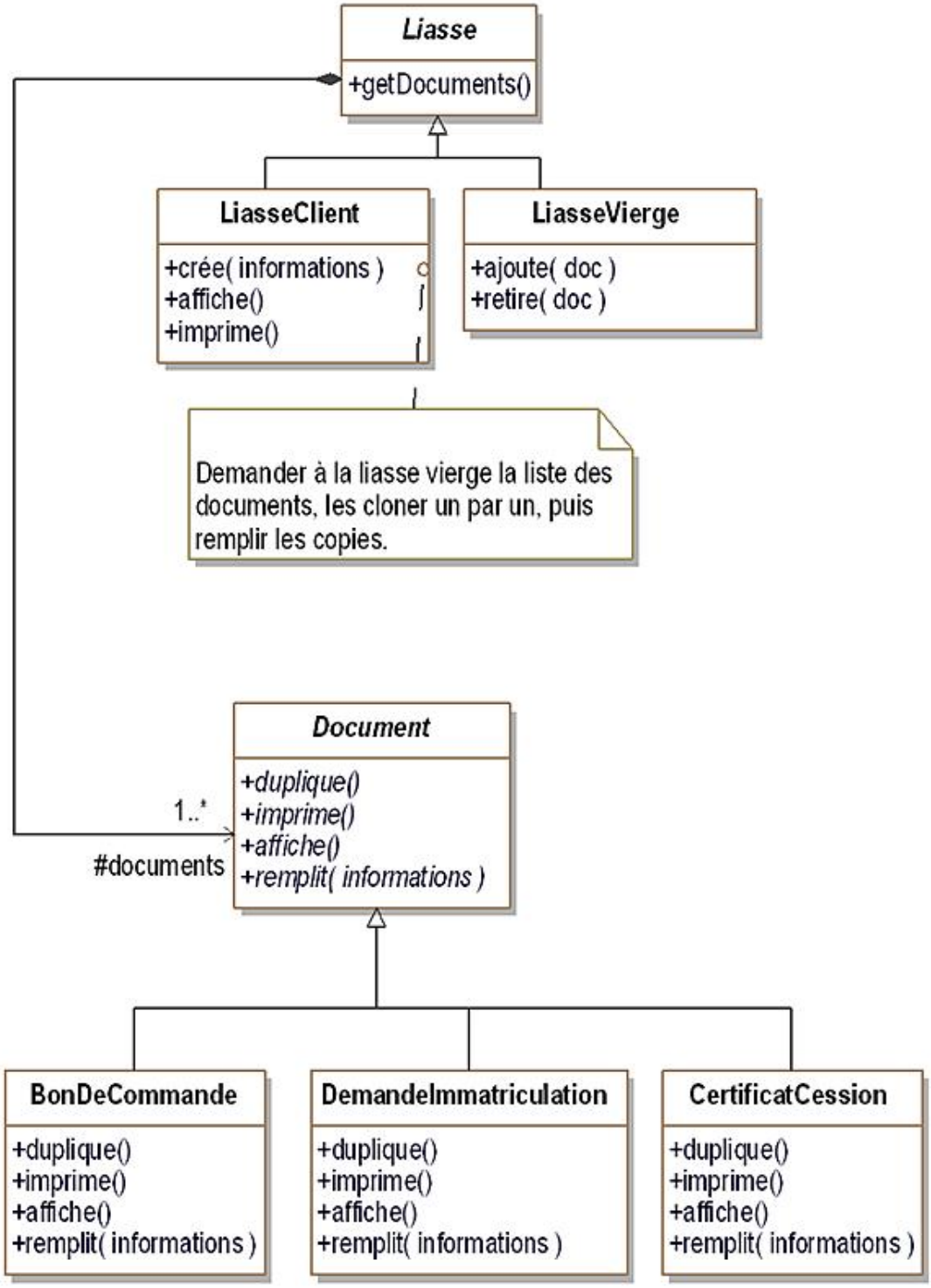


Figure 7.1 - Le pattern `Prototype` appliqué à la création de liasse de contenu variable

Structure

1. Diagramme de classes

La figure 7.2 détaille la structure générique du pattern.

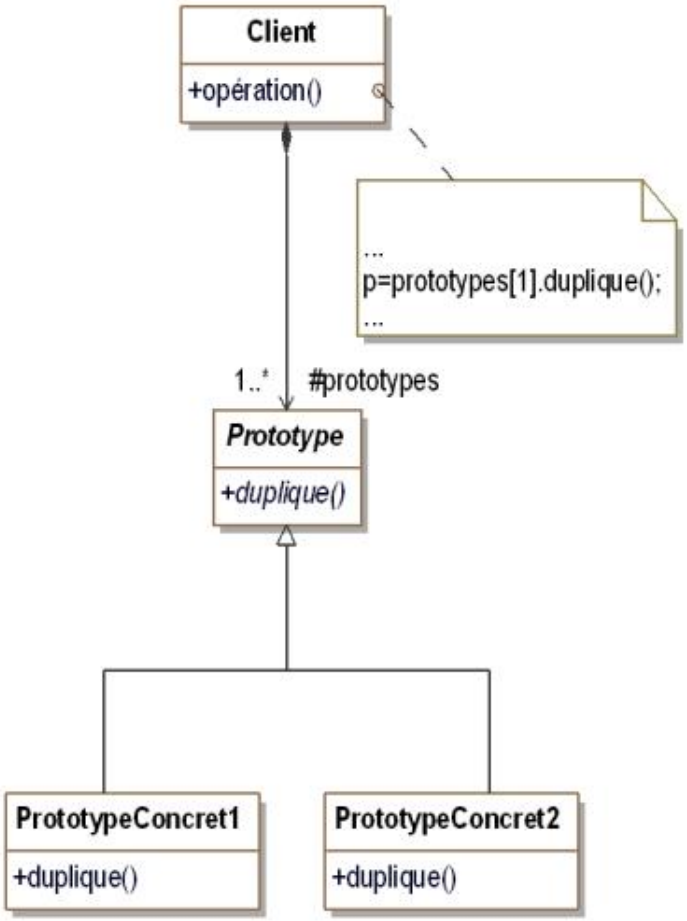


Figure 7.2 - Structure du pattern `Prototype`

2. Participants

Les participants au pattern sont les suivants :

- `Client` (`Liasse`, `LiasseClient`, `LiasseVierge`) est une classe composée d'un ensemble d'objets appelés prototypes, instances de la classe abstraite `Prototype`. La classe Client a besoin de dupliquer ces prototypes sans avoir à connaître ni la structure interne de `Prototype` ni sa hiérarchie de sous-classes.

- `Prototype` (`Document`) est une classe abstraite d'objets capables de se dupliquer eux-mêmes. Elle introduit la signature de la méthode duplique.

- `PrototypeConcret1`et `PrototypeConcret2` (`BonDeCommande`, `DemandeImmatriculation`, `CertificatCession`) sont les sous-classes concrètes de `Prototype` qui définissent complètement un prototype et en implante la méthode `duplique`.

3. Collaboration

Le client demande à un ou plusieurs prototypes de se dupliquer eux-mêmes.

Domaines d’utilisation

Le pattern `Prototype` est utilisé dans les domaines suivants :

- un système d’objets doit créer des instances sans connaître la hiérarchie des classes les décrivant ;
- un système d’objets doit créer des instances de classes chargées dynamiquement ;
- le système d’objets doit rester simple et ne pas inclure une hiérarchie parallèle de classes de fabrique.

Exemple en Java

Le code source de la classe abstraite `Document` et de ses sous-classes concrètes est donné à la suite. Pour simplifier, à la différence du diagramme de classes, les méthodes `duplique` et `remplit` sont concrètes dans la classe `Document`. La méthode `duplique` utilise la méthode `clone` fournie par Java.

```
public abstract class Document implements Cloneable
{
  protected String contenu = new String();

  public Document duplique()
  {
    Document resultat;
    try
    {
      resultat = (Document)this.clone();
    }
    catch (CloneNotSupportedException exception)
    {
      return null;
    }
    return resultat;
  }

  public void remplit(String informations)
  {
    contenu = informations;
  }

  public abstract void imprime();
  public abstract void affiche();
}

public class BonDeCommande extends Document
{
  public void affiche()
  {
    System.out.println("Affiche le bon de commande : " +
      contenu);
  }

  public void imprime()
  {
    System.out.println("Imprime le bon de commande : " +
      contenu);
  }
}

public class DemandeImmatriculation extends Document
{
  public void affiche()
  {
    System.out.println(
      "Affiche la demande d'immatriculation : " + contenu);
  }

  public void imprime()
  {
    System.out.println(
      "Imprime la demande d'immatriculation : " + contenu);
  }
}

public class CertificatCession extends Document
{
  public void affiche()
  {
```

```
    System.out.println(
      "Affiche le certificat de cession : " + contenu);
  }

  public void imprime()
  {
    System.out.println(
      "Imprime le certificat de cession : " + contenu);
  }
}
```

Le code source de la classe abstraite `Liasse` est le suivant :

```
import java.util.*;
public abstract class Liasse
{
  protected List<Document> documents;

  public List<Document> getDocuments()
  {
    return documents;
  }
}
```

Le code source de la sous-classe `LiasseVierge` de `Liasse` est fourni à la suite. Ce code utilise le pattern `Singleton` qui est présenté dans le chapitre suivant et qui a pour objectif d'assurer qu'une classe ne possède qu'une seule instance.

```
import java.util.*;
public class LiasseVierge extends Liasse
{
  private static LiasseVierge _instance = null;

  private LiasseVierge()
  {
    documents = new ArrayList<Document>();
  }

  public static LiasseVierge Instance()
  {
    if (_instance == null)
      _instance = new LiasseVierge();
    return _instance;
  }

  public void ajoute(Document doc)
  {
    documents.add(doc);
  }

  public void retire(Document doc)
  {
    int indexDoc = documents.indexOf(doc);
    if (indexDoc > 0)
      documents.remove(indexDoc);
  }
}
```

La sous-classe `LiasseClient` est écrite en Java à la suite. Son constructeur commence par obtenir la liste des documents de la liasse vierge puis les duplique, les remplis et les ajoute un à un au contenu de la liasse.

```
import java.util.*;
public class LiasseClient extends Liasse
{
  public LiasseClient(String informations)
  {
    documents = new ArrayList<Document>();
    LiasseVierge laLiasseVierge = LiasseVierge.Instance();
    List<Document> documentsVierges =
      laLiasseVierge.getDocuments();
```

```
    for (Document document: documentsVierges)
    {
      Document copieDocument = document.duplique();
      copieDocument.remplit(informations);
      documents.add(copieDocument);
    }
  }

  public void affiche()
  {
    for (Document document: documents)
      document.affiche();
  }

  public void imprime()
  {
    for (Document document: documents)
      document.imprime();
  }
}
```

Enfin, nous donnons le code source de la classe `Utilisateur` dont la méthode main commence par construire la liasse vierge et, en particulier, son contenu. Puis cette méthode crée et affiche les liasses de deux clients.

```
public class Utilisateur
{

  public static void main(String[] args)
  {
    // initialisation de la liasse vierge
    LiasseVierge liasseVierge = LiasseVierge.Instance();
    liasseVierge.ajoute(new BonDeCommande());
    liasseVierge.ajoute(new CertificatCession());
    liasseVierge.ajoute(new DemandeImmatriculation());
    // création d'une nouvelle liasse pour deux clients
    LiasseClient liasseClient1 = new LiasseClient(
      "Martin");
    LiasseClient liasseClient2 = new LiasseClient(
      "Durant");
    liasseClient1.affiche();
    liasseClient2.affiche();
  }
}
```

Le résultat de l'exécution est le suivant :

```
Affiche le bon de commande : Martin
Affiche le certificat de cession : Martin
Affiche la demande d'immatriculation : Martin
Affiche le bon de commande : Durant
Affiche le certificat de cession : Durant
Affiche la demande d'immatriculation : Durant
```

Description

Le pattern `Singleton` a pour but d'assurer qu'une classe ne possède qu'une seule instance et de fournir une méthode de classe unique retournant cette instance.

Dans certains cas, il est utile de gérer des classes ne possédant qu'une seule instance. Dans le cadre des patterns de construction, nous pouvons citer le cas d'une fabrique de produits (pattern `Abstract Factory`) dont il n'est pas nécessaire de créer plus d'une instance.

Exemple

Dans le système de vente en ligne de véhicules, nous devons gérer des classes possédant une seule instance.

Le système de liasse de documents destinés au client lors de l'achat d'un véhicule (comme le certificat de cession, la demande d'immatriculation et le bon de commande) utilise la classe `LiasseVierge` qui ne possède qu'une seule instance. Cette instance référence tous les documents nécessaires pour le client. Cette instance unique est appelée la liasse vierge car les documents qu'elle référence sont tous vierges. L'utilisation complète de la classe `LiasseVierge` est expliquée dans le chapitre consacré au pattern `Prototype`.

La figure 8.1 illustre l'utilisation du pattern `Singleton` pour la classe `LiasseVierge`. L'attribut de classe `instance` contient soit `null` soit l'unique instance de la classe `LiasseVierge`. La méthode de classe `Instance` renvoie cette unique instance en retournant la valeur de l'attribut `instance`. Si cet attribut a pour valeur `null`, il est préalablement initialisé lors de la création de l'unique instance.

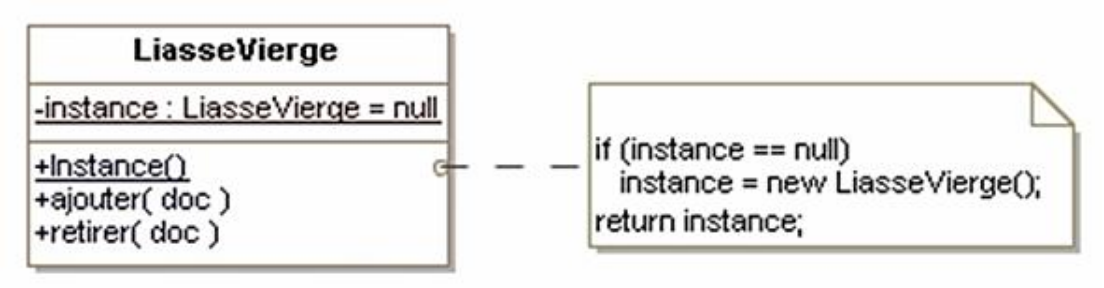


Figure 8.1 - Le pattern `Singleton` appliqué à la classe `LiasseVierge`

Structure

1. Diagramme de classe

La figure 8.2 détaille la structure générique du pattern.

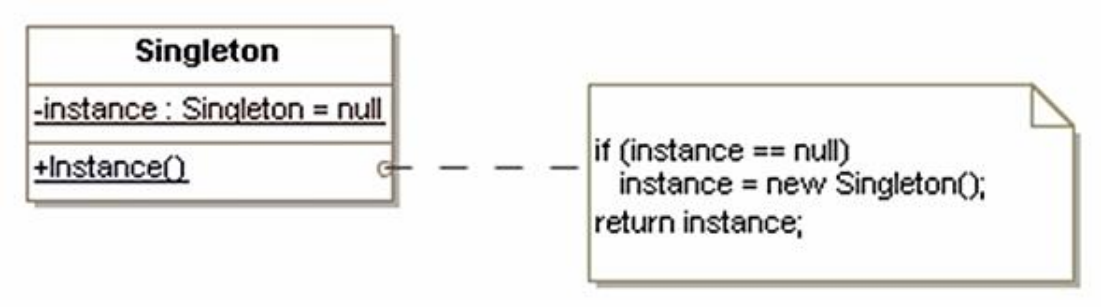


Figure 8.2 - Structure du pattern `Singleton`

2. Participant

Le seul participant est la classe `Singleton` qui offre l'accès à l'unique instance par sa méthode de classe `Instance`.

Par ailleurs, la classe `Singleton` possède un mécanisme qui assure qu'elle ne peut posséder au plus qu'une seule instance. Ce mécanisme bloque la création d'autres instances.

3. Collaboration

Chaque client de la classe `Singleton` accède à l'unique instance par la méthode de classe `Instance`. Il ne peut pas créer de nouvelles instances en utilisant l'opérateur habituel d'instanciation (opérateur `new`) qui est bloqué.

Domaine d’utilisation

Le pattern est utilisé dans le cas suivant :

- il ne doit y avoir qu’une seule instance d’une classe ;
- cette instance ne doit être accessible qu’au travers d’une méthode de classe.

L’utilisation du pattern Singleton offre également la possibilité de ne plus utiliser de variables globales.

Exemples en Java

1. La liasse vierge

Le code Java complet de la classe `LiasseVierge` est donné dans le chapitre du pattern `Prototype`. La partie de cette classe relative à l'utilisation du pattern `Singleton` est fournie à la suite.

Le constructeur de cette classe a une visibilité privée afin que seule la méthode `Instance` puisse l'utiliser. Ainsi, aucun objet externe à la classe `LiasseVierge` ne peut en créer d'instance en utilisant l'opérateur `new`.

De la même façon, l'attribut `_instance` détient également une visibilité privée pour que l'accès ne soit possible que depuis la méthode de classe `Instance`.

```
import java.util.*;
public class LiasseVierge extends Liasse
{
  private static LiasseVierge _instance = null;

  private LiasseVierge()
  {
    documents = new ArrayList<Document>();
  }

  public static LiasseVierge Instance()
  {
    if (_instance == null)
      _instance = new LiasseVierge();
    return _instance;
  }

  ...

}
```

Le seul client de la classe `LiasseVierge` est la classe `LiasseClient` qui, dans son constructeur, obtient une référence à la liasse vierge en invoquant la méthode `Instance`. Ensuite, le constructeur accède à la liste des documents vierges.

```
    LiasseVierge laLiasseVierge = LiasseVierge.Instance();
    List<Document> documentsVierges =
      laLiasseVierge.getDocuments();
```

2. La classe Vendeur

Dans le système de vente de véhicules, nous voulons représenter le vendeur par une classe pour y mémoriser ses informations plutôt que d'utiliser des variables globales qui vont respectivement contenir son nom, son adresse, etc.

La classe `Vendeur` est décrite ci-dessous.

```
public class Vendeur
{
  protected String nom;
  protected String adresse;
  protected String email;

  private static Vendeur _instance = null;

  private Vendeur(){}

  public static Vendeur Instance()
  {
    if (_instance == null)
      _instance = new Vendeur();
    return _instance;
  }
```

```
  public void affiche()
  {
    System.out.println("Nom : " + nom);
    System.out.println("Adresse : " + adresse);
    System.out.println("Email : " + email);
  }

  public String getNom()
  {
    return nom;
  }

  public void setNom(String nom)
  {
    this.nom = nom;
  }

  public String getAdresse()
  {
    return adresse;
  }

  public void setAdresse(String adresse)
  {
    this.adresse = adresse;
  }

  public String getEmail()
  {
    return email;
  }

  public void setEmail(String email)
  {
    this.email = email;
  }

}
```

Le programme principal suivant utilise la classe `Vendeur`.

```
public class TestVendeur
{

  public static void main(String[] args)
  {
    // initialisation du vendeur du système
    Vendeur leVendeur = Vendeur.Instance();
    leVendeur.setNom("Vendeur Auto");
    leVendeur.setAdresse("Paris");
    leVendeur.setEmail("vendeur@vendeur.com");
    // affichage du vendeur du système
    affiche();
  }

  public static void affiche()
  {
    Vendeur leVendeur = Vendeur.Instance();
    leVendeur.affiche();
  }

}
```

Son exécution montre bien qu'il n'y a qu'une seule instance car la méthode `affiche` de `testVendeur` ne reçoit pas de paramètre.

```
Nom : Vendeur Auto
Adresse : Paris
```

```
Email : vendeur@vendeur.com
```

Présentation

L'objectif des patterns de structuration est de faciliter l'indépendance de l'interface d'un objet ou d'un ensemble d'objets vis-à-vis de son implantation. Dans le cas d'un ensemble d'objets, il s'agit aussi de rendre cette interface indépendante de la hiérarchie des classes et de la composition des objets.

En fournissant les interfaces, les patterns de structuration encapsulent la composition des objets, augmentant le niveau d'abstraction du système à l'image des patterns de création qui encapsulent la création des objets. Les patterns de structuration mettent en avant les interfaces.

L'encapsulation de la composition est réalisée non pas en structurant l'objet lui-même mais en transférant cette structuration à un second objet. Celui-ci est intimement lié au premier objet. Ce transfert de structuration signifie que le premier objet détient l'interface vis-à-vis des clients et gère la relation avec le second objet qui lui gère la composition et n'a aucune interface avec les clients externes.

Cette réalisation offre une autre propriété qui est la souplesse de la composition qui peut être modifiée dynamiquement. En effet, il est aisé de substituer un objet par un autre pourvu qu'il soit issu de la même classe ou qu'il respecte la même interface. Les patterns `Composite`, `Decorator` et `Bridge` illustrent pleinement ce mécanisme.

Composition statique et dynamique

Nous prenons l'exemple des aspects d'implantation d'une classe. Nous nous plaçons dans un cadre où il est possible d'avoir plusieurs implantations possibles. La solution classique consiste à les différencier au niveau des sous-classes. C'est le cas de l'utilisation de l'héritage d'une interface dans plusieurs classes d'implantation comme l'illustre le diagramme des classes de la figure 9.1.

Cette solution consiste à réaliser une composition statique. En effet, une fois que le choix de la classe d'implantation d'un objet est effectué, il n'est plus possible d'en changer.

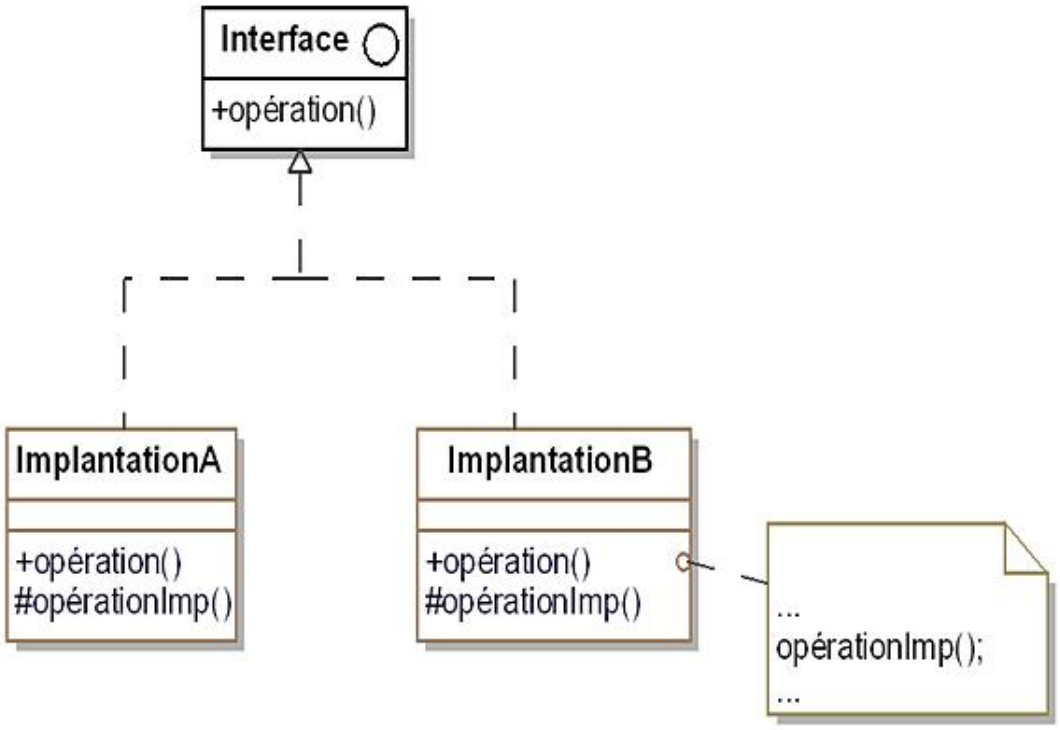


Figure 9.1 - Implantation d'un objet par héritage

Comme expliqué dans la précédente section, une autre solution est de séparer l'aspect d'implantation dans un autre objet comme l'illustre la figure 9.2. Les parties d'implantation sont gérées par une instance de la classe `ImplantationConcrèteA` ou par une instance de la classe `ImplantationConcrèteB`. Cette instance est référencée par l'attribut `implantation`. Elle peut être substituée facilement par une autre instance lors de l'exécution. Par conséquent, la composition est dynamique.

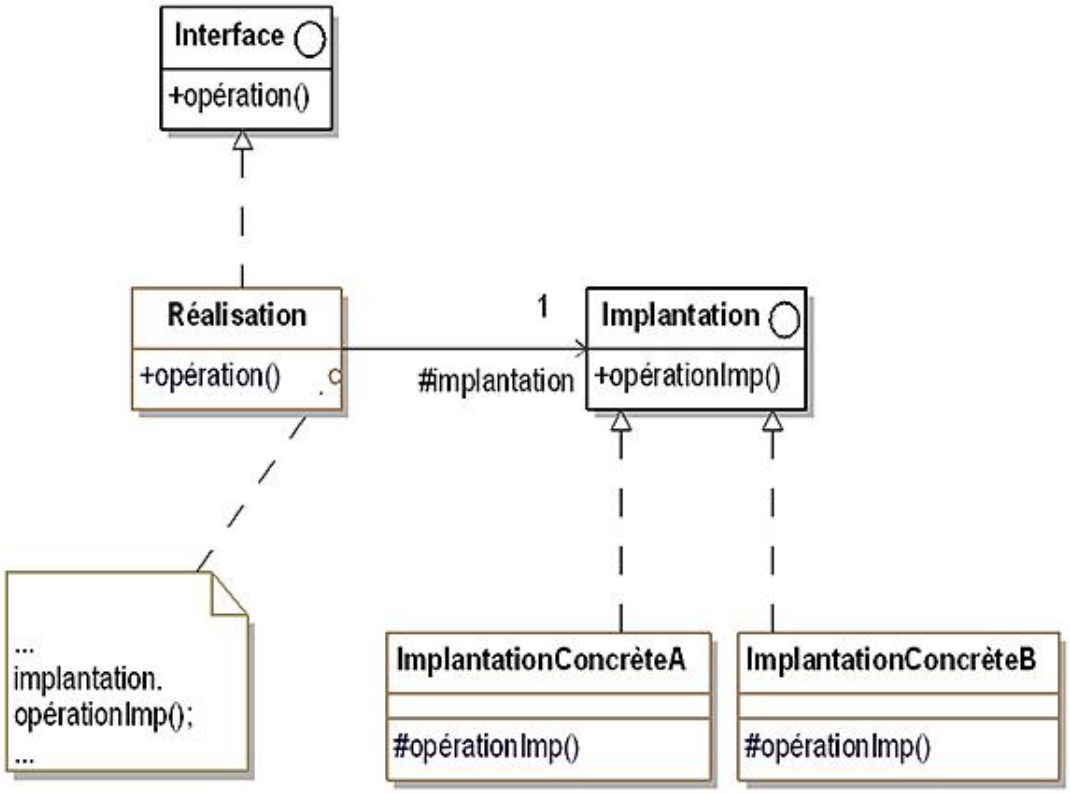


Figure 9.2 - Implantation d'un objet par association

La solution de la figure 9.2 est détaillée dans le chapitre consacré au pattern `Bridge`.

Cette solution présente aussi l'avantage d'encapsuler la partie d'implantation qui devient ainsi totalement transparente pour les clients.

Tous les patterns de structuration sont basés sur l'utilisation d'un ou de plusieurs objets déterminant la structuration. La liste suivante décrit la fonction que remplit cet objet pour chaque pattern.

- **Adapter** : adapte un objet existant.
- **Bridge** : implante un objet.
- **Composite** : organise la composition hiérarchique d'un objet.
- **Decorator** : se substitue à l'objet existant en lui ajoutant de nouvelles fonctionnalités.
- **Facade** : se substitue à un ensemble d'objets existants en leur conférant une interface unifiée.
- **Flyweight** : est destiné au partage et détient un état indépendant des objets qui le référencent.
- **Proxy** : se substitue à l'objet existant en fournissant un comportement adapté à des besoins d'optimisation ou de protection.

Description

Le but du pattern `Adapter` est de convertir l'interface d'une classe existante en l'interface attendue par des clients également existants afin qu'ils puissent travailler ensemble. Il s'agit de conférer à une classe existante une nouvelle interface pour répondre aux besoins de clients.

Exemple

Le serveur web du système de vente de véhicules crée et gère des documents destinés aux clients. L'interface `Document` a été définie pour cette gestion. Sa représentation UML est montrée à la figure 10.1 ainsi que ses trois méthodes `setContenu`, `dessine` et `imprime`. Une première classe d'implantation de cette interface a été réalisée : la classe `DocumentHtml` qui implante ces trois méthodes. Des objets clients de cette interface et de cette classe ont été conçus.

Par la suite, l'ajout des documents PDF a posé un problème car ceux-ci sont plus complexes à construire et à gérer que des documents HTML. Un composant du marché a été choisi mais dont l'interface ne correspond à l'interface `Document`. La figure 10.1 montre le composant `ComposantPdf` dont l'interface introduit plus de méthodes et dont la convention de nommage est de surcroît différente (préfixe pdf).

Le pattern `Adapter` propose une solution qui consiste à créer la classe `DocumentPdf` implantant l'interface `Document` et possédant une association avec `ComposantPdf`. L'implantation des trois méthodes de l'interface `Document` consiste à déléguer correctement les appels au composant PDF. Cette solution est visible sur la figure 10.1, le code des méthodes étant détaillé à l'aide de notes.

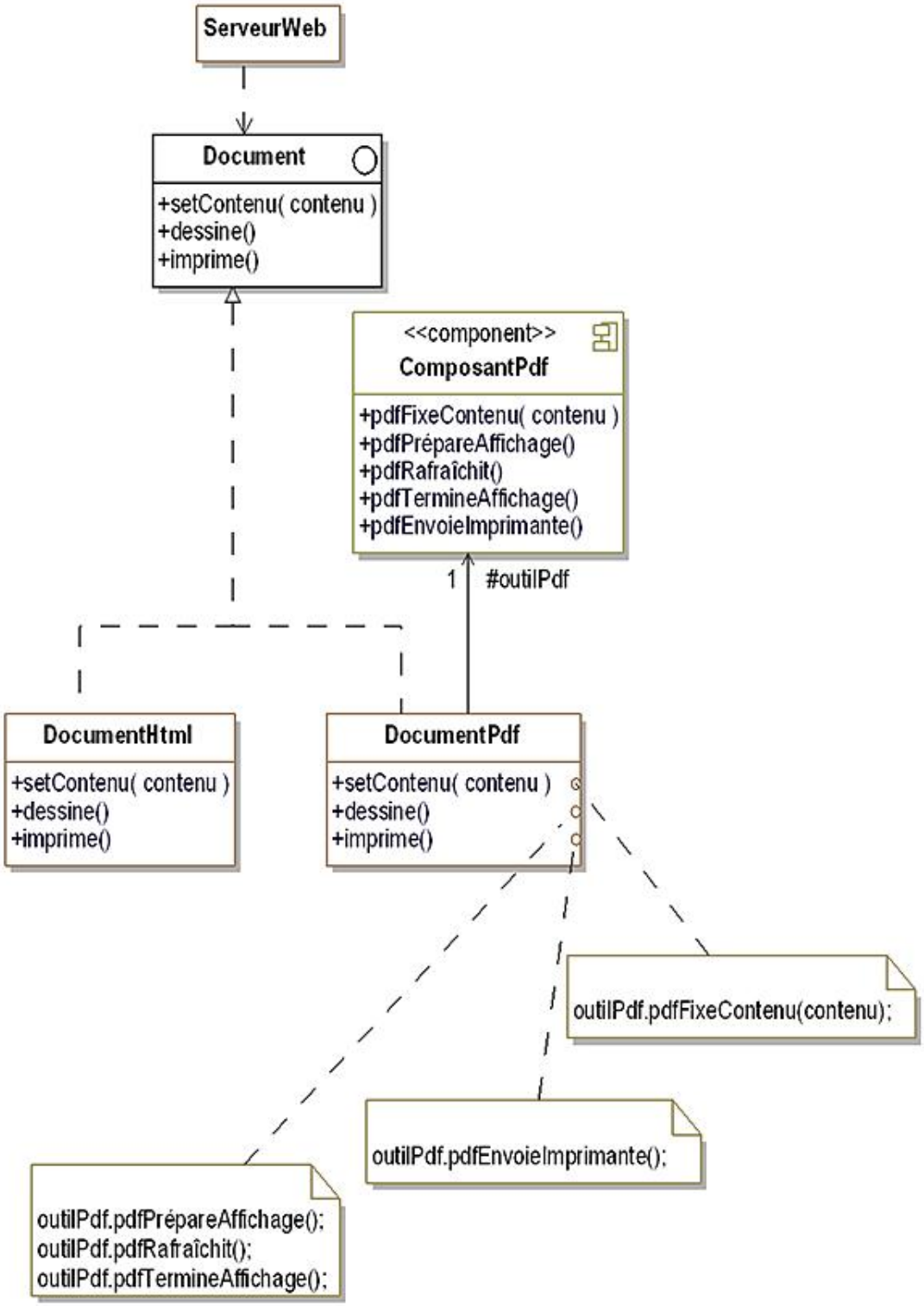


Figure 10.1 - Le pattern `Adapter` appliqué à un composant de documents PDF

Structure

1. Diagramme de classes

La figure 10.2 détaille la structure générique du pattern.

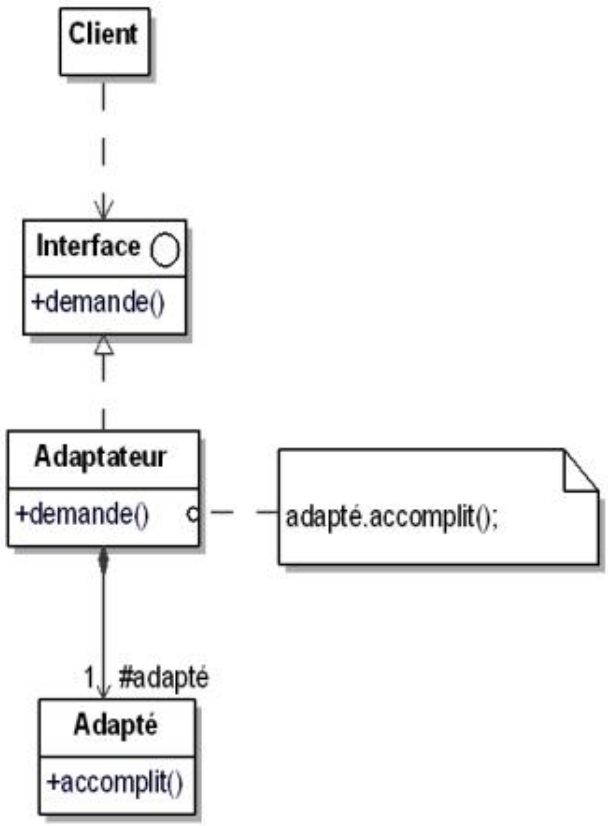


Figure 10.2 - Structure du pattern `Adapter`

2. Participants

Les participants au pattern sont les suivants :

- `Interface (Document)`introduit la signature des méthodes de l'objet ;
- `Client (ServeurWeb)`interagit avec les objets répondant à `Interface` ;
- `Adaptateur (DocumentPdf)`implante les méthodes de `Interface` en invoquant les méthodes de l'objet adapté ;
- `Adapté (ComposantPdf)`introduit l'objet dont l'interface doit être adaptée pour correspondre à `Interface`.

3. Collaborations

Le client invoque la méthode `demande` de l'adaptateur qui, en conséquence, interagit avec l'objet adapté en appelant la méthode `accomplit`. Ces collaborations sont illustrées à la figure 10.3.

Figure 10.3 - Diagramme de séquence du pattern `Adapter`

Domaines d’application

Le pattern est utilisé dans les cas suivants :

- pour intégrer dans un système un objet dont l’interface ne correspond pas à l’interface requise au sein de ce système ;
- pour fournir des interfaces multiples à un objet lors de sa conception.

Exemple en Java

Nous présentons le code source de l'exemple en Java.

Nous commençons par l'interface `Document` :

```
public interface Document
{
  void setContenu(String contenu);
  void dessine();
  void imprime();
}
```

La classe `DocumentHtml` est l'exemple de classe implantant l'interface `Document`.

```
public class DocumentHtml implements Document
{
  protected String contenu;

  public void setContenu(String contenu)
  {
    this.contenu = contenu;
  }

  public void dessine()
  {
    System.out.println("Dessine document HTML : " +
      contenu);
  }

  public void imprime()
  {
    System.out.println("Imprime document HTML : " +
      contenu);
  }

}
```

La classe `ComposantPdf` représente le composant existant qui est intégré dans l'application. Sa conception est indépendante de l'application et, en particulier, de l'interface `Document`. Cette classe devra être adaptée par la suite.

```
public class ComposantPdf
{
  protected String contenu;

  public void pdfFixeContenu(String contenu)
  {
    this.contenu = contenu;
  }

  public void pdfPrepareAffichage()
  {
    System.out.println("Affichage PDF : Début");
  }

  public void pdfRafraichit()
  {
    System.out.println("Affichage contenu PDF : " +
      contenu);
  }

  public void pdfTermineAffichage()
  {
    System.out.println("Affichage PDF : Fin");
  }

  public void pdfEnvoieImprimante()
```

```
  {
    System.out.println("Impression PDF : " + contenu);
  }
}
```

La classe `DocumentPdf` représente l'adaptateur. Elle est associée à la classe `ComposantPdf` au travers de l'attribut `outilPdf` qui associe l'objet adapté.

Elle implante l'interface `Document` et chacune de ses méthodes invoque les méthodes nécessaires de l'objet adapté afin de réaliser l'adaptation entre les deux interfaces.

```
public class DocumentPdf implements Document
{
  protected ComposantPdf outilPdf = new ComposantPdf();

  public void setContenu(String contenu)
  {
    outilPdf.pdfFixeContenu(contenu);
  }

  public void dessine()
  {
    outilPdf.pdfPrepareAffichage();
    outilPdf.pdfRafraichit();
    outilPdf.pdfTermineAffichage();
  }

  public void imprime()
  {
    outilPdf.pdfEnvoieImprimante();
  }
}
```

Le programme principal correspond à la classe `ServeurWeb` qui crée un document HTML, en fixe le contenu puis le dessine.

Ensuite, le programme fait la même chose avec un document PDF.

```
public class ServeurWeb
{

  public static void main(String[] args)
  {
    Document document1, document2;
    document1 = new DocumentHtml();
    document1.setContenu("Hello");
    document1.dessine();
    System.out.println();
    document2 = new DocumentPdf();
    document2.setContenu("Bonjour");
    document2.dessine();
  }
}
```

L'exécution de ce programme principal donne le résultat suivant.

```
Dessine document HTML : Hello

Affichage PDF : Début
Affichage contenu PDF : Bonjour
Affichage PDF : Fin
```

Description

Le but du pattern `Bridge` est de séparer l'aspect d'implantation d'un objet de son aspect de représentation et d'interface.

Ainsi, d'une part l'implantation peut être totalement encapsulée et d'autre part l'implantation et la représentation peuvent évoluer indépendamment et sans que l'une exerce une contrainte sur l'autre.

Exemple

Pour effectuer une demande d'immatriculation d'un véhicule d'occasion, il convient de préciser sur cette demande certaines informations importantes comme le numéro de la plaque existante. Le système affiche un formulaire pour demander ces informations.

Il existe deux implantations des formulaires :

- les formulaires HTML ;
- les formulaires basés sur une applet Java.

Il est donc possible d'introduire une classe abstraite `FormulaireImmatriculation` et deux sous-classes concrètes `FormulaireImmatriculationHtml` et `FormulaireImmatriculationApplet`.

Dans un premier temps, les demandes d'immatriculation ne concernaient que la France. Par la suite, il est devenu nécessaire d'introduire une nouvelle sous-classe de `FormulaireImmatriculation` correspondant aux demandes d'immatriculation au Luxembourg, sous-classe appelée `FormulaireImmatriculationLux`. Cette sous-classe doit également être abstraite et avoir également deux sous-classes concrètes pour chaque implantation. La figure 11.1 montre le diagramme de classes correspondant.

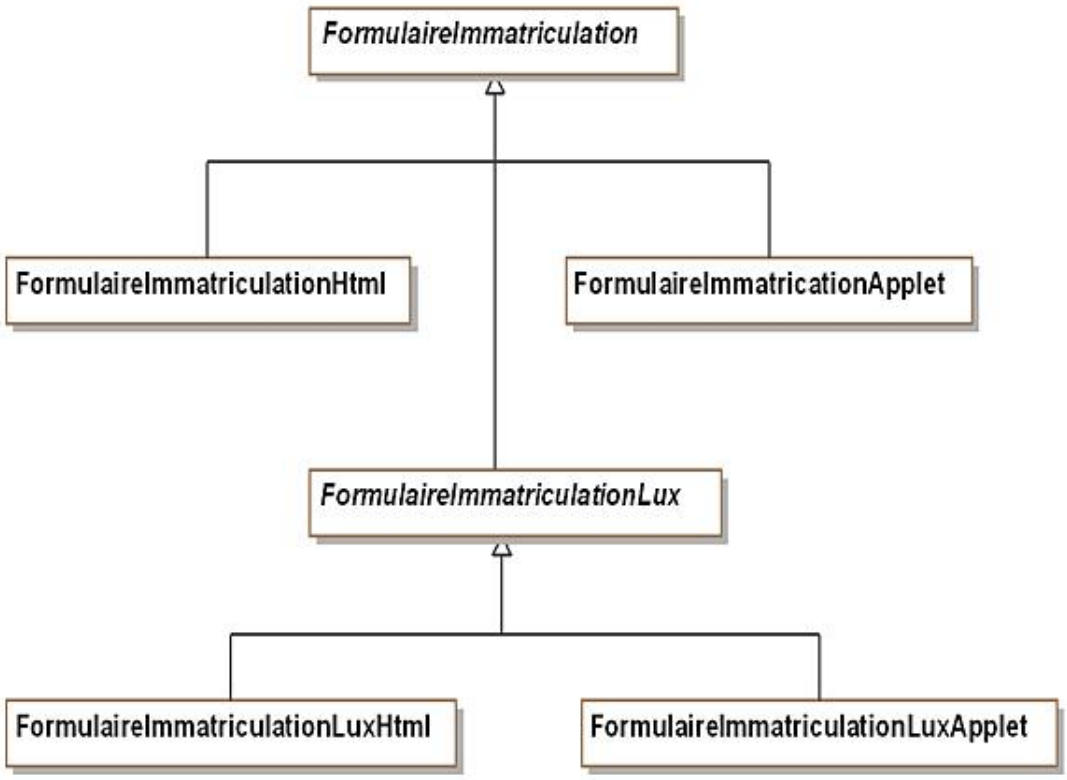


Figure 11.1 - Hiérarchie des formulaires intégrant les sous-classes d'implantation

Ce diagramme met en avant deux problèmes :

- La hiérarchie mélange au même niveau des sous-classes d'implantation et une sous-classe de représentation : `FormulaireImmatriculationLux`. De plus pour chaque classe de représentation, il faut introduire deux sous-classes d'implantation, ce qui conduit rapidement à une hiérarchie très complexe.
- Les clients sont dépendants de l'implantation. En effet, ils doivent interagir avec les classes concrètes d'implantation.

La solution du pattern `Bridge` consiste donc à séparer les aspects de représentation de ceux d'implantation et à créer deux hiérarchies de classes comme illustré à la figure 11.2. Les instances de la classe `FormulaireImmatriculation` détiennent le lien `implantation` vers une instance répondant à l'interface `FormulaireImpl`.

L'implantation des méthodes de `FormulaireImmatriculation` est basée sur l'utilisation des méthodes décrites dans `FormulaireImpl`.

Quant à la classe `FormulaireImmatriculation`, elle est maintenant abstraite et il existe une sous-classe concrète pour chaque pays (`FormImmatriculationFrance` et `FormImmatriculationLuxembourg`).

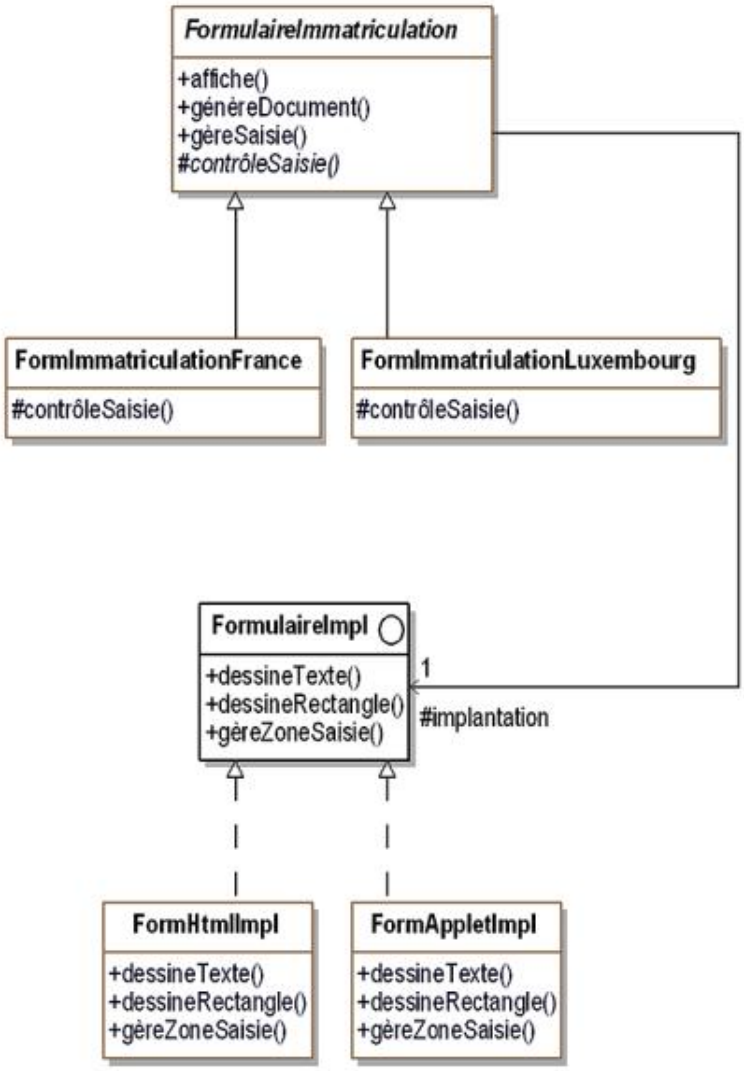


Figure 11.2 - Le pattern `Bridge` appliqué à l'implantation de formulaires

Structure

1. Diagramme de classes

La figure 11.3 détaille la structure générique du pattern.

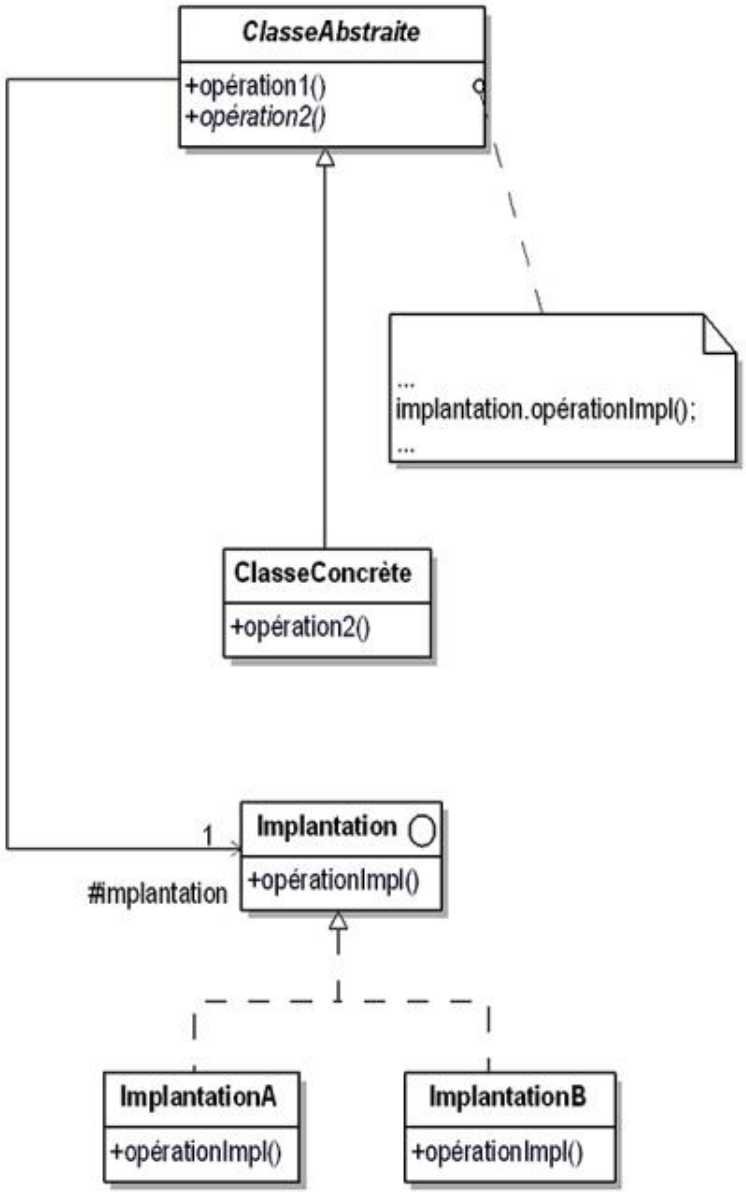


Figure 11.3 - Structure du pattern `Bridge`

2. Participants

Les participants au pattern sont les suivants :

- `ClasseAbstraite` (`FormulaireImmatriculation`) est la classe abstraite qui représente les objets du domaine. Elle détient l'interface pour les clients et contient une référence vers un objet répondant à l'interface `Implantation`.

- `ClasseConcrète` (`FormImmatriculationFranceet FormImmatriculationLuxembourg`) est la classe concrète qui

implante les méthodes de `ClasseAbstraite`.

- `Implantation` (`FormulaireImpl`) définit l'interface des classes d'implantation. Les méthodes de cette interface ne doivent pas correspondre aux méthodes de `ClasseAbstraite`. Les deux ensembles de méthodes sont différents. L'implantation introduit en général des méthodes de bas niveau et les méthodes de `ClasseAbstraite` sont des méthodes de haut niveau.

- `ImplantationA`, `ImplantationB` (`FormHtmlImpl`, `FormAppletImpl`) sont des classes concrètes qui réalisent les méthodes introduites dans l'interface `Implantation`.

3. Collaborations

Les opérations de `ClasseAbstraite` et de ses sous-classes invoquent les méthodes introduites dans l'interface `Implantation`.

Domaines d’application

Le pattern est utilisé dans les cas suivants :

- pour éviter une liaison forte entre la représentation des objets et leur implantation, notamment quand l’implantation est sélectionnée au cours de l’exécution de l’application ;
- pour que les changements dans l’implantation des objets n’aient pas d’impact dans les interactions entre les objets et leurs clients ;
- pour permettre à la représentation des objets et à leur implantation de conserver leur capacité d’extension par la création de nouvelles sous-classes ;
- pour éviter d’obtenir des hiérarchies de classes extrêmement complexes comme illustré à la figure 11.1.

Exemple en Java

Nous introduisons maintenant un exemple en Java basé sur le diagramme de classes de la figure 11.2.

Nous commençons par l'interface décrivant l'implantation des formulaires qui contient deux méthodes, l'une pour afficher un texte et l'autre pour gérer une zone de saisie.

```
public interface FormulaireImpl
{
  void dessineTexte(String texte);
  String gereZoneSaisie();
}
```

Nous montrons la classe d'implantation `FormHtmlImpl` qui simule l'affichage et la saisie d'un formulaire HTML.

```
import java.util.*;
public class FormHtmlImpl implements FormulaireImpl
{
  Scanner reader = new Scanner(System.in);

  public void dessineTexte(String texte)
  {
    System.out.println("HTML : " + texte);
  }

  public String gereZoneSaisie()
  {
    return reader.next();
  }
}
```

Nous montrons la classe d'implantation `FormAppletImpl` qui simule l'affichage et la saisie d'un formulaire à l'aide d'une applet.

```
import java.util.Scanner;
public class FormAppletImpl implements FormulaireImpl
{
  Scanner reader = new Scanner(System.in);

  public void dessineTexte(String texte)
  {
    System.out.println("Applet : " + texte);
  }

  public String gereZoneSaisie()
  {
    return reader.next();
  }
}
```

Nous passons à la classe abstraite `FormulaireImmatriculation`.

Son constructeur prend en paramètre une instance gérant l'implantation et qui est utilisée dans les autres méthodes pour dessiner du texte ou gérer la saisie d'entrées au clavier.

Il convient de noter la méthode `controleSaisie` qui vérifie que le numéro d'une plaque d'immatriculation est correct, ce qui dépend du pays. Cette méthode est donc abstraite et implantée dans les sous-classes.

```
public abstract class FormulaireImmatriculation
{
  protected String contenu;
  protected FormulaireImpl implantation;

  public FormulaireImmatriculation(FormulaireImpl
    implantation)
  {
    this.implantation = implantation;
  }
```

```
  public void affiche()
  {
    implantation.dessineTexte(
      "numéro de la plaque existante : ");
  }

  public void genereDocument()
  {
    implantation.dessineTexte("Demande d'immatriculation");
    implantation.dessineTexte("numéro de plaque : " +
      contenu);
  }

  public boolean gereSaisie()
  {
    contenu = implantation.gereZoneSaisie();
    return this.controleSaisie(contenu);
  }

  protected abstract boolean controleSaisie(String plaque);

}
```

La sous-classe concrète de formulaire d'immatriculation en France implante la méthode `controleSaisie` qui vérifie que le numéro de la plaque a une longueur comprise entre 7 et 8.

```
public class FormImmatriculationFrance extends
  FormulaireImmatriculation
{
  public FormImmatriculationFrance(FormulaireImpl
    implantation)
  {
    super(implantation);
  }

  protected boolean controleSaisie(String plaque)
  {
    return (plaque.length() >= 7) &&
     (plaque.length() <= 8);
  }
}
```

La sous-classe concrète de formulaire d'immatriculation au Luxembourg implante la méthode `controleSaisie` qui vérifie que le numéro de la plaque a une longueur de 5.

```
public class FormImmatriculationLuxembourg extends
  FormulaireImmatriculation
{
  public FormImmatriculationLuxembourg(FormulaireImpl
    implantation)
  {
    super(implantation);
  }

  protected boolean controleSaisie(String plaque)
  {
    return plaque.length() == 5;
  }
}
```

Enfin, nous présentons le programme principal de la classe `Utilisateur` qui crée un formulaire de génération d'un document de demande d'immatriculation pour le Luxembourg et si sa saisie est correcte affiche le document à l'écran.

Ensuite, le programme fait la même chose avec un document de demande d'immatriculation pour la France.

```
public class Utilisateur
{
  public static void main(String[] args)
  {
```

```
    FormImmatriculationLuxembourg formulaire1 = new
      FormImmatriculationLuxembourg(new FormHtmlImpl());
    formulaire1.affiche();
    if (formulaire1.gereSaisie())
      formulaire1.genereDocument();
    System.out.println();
    FormImmatriculationFrance formulaire2 = new
      FormImmatriculationFrance(new FormAppletImpl());
    formulaire2.affiche();
    if (formulaire2.gereSaisie())
      formulaire2.genereDocument();
  }
}
```

Un exemple d'exécution est le suivant (les numéros de plaques introduits sont 2345X et 97AAA59).

```
HTML : numéro de la plaque existante :
2345X
HTML : Demande d'immatriculation
HTML : numéro de plaque : 2345X

Applet : numéro de la plaque existante :
97AAA59
Applet : Demande d'immatriculation
Applet : numéro de plaque : 97AAA59
```

Description

Le but du pattern `Composite` est d'offrir un cadre de conception d'une composition d'objets dont la profondeur est variable, cette conception étant basée sur un arbre.

Par ailleurs, cette composition est encapsulée vis-à-vis des clients des objets qui peuvent interagir sans devoir connaître la profondeur de la composition.

Exemple

Au sein de notre système de vente de véhicules, nous voulons représenter les sociétés clientes, notamment pour connaître le nombre de véhicules dont elles disposent et leur proposer des offres de maintenance de leur parc.

Les sociétés qui possèdent des filiales demandent des offres de maintenance qui prennent en compte le parc de véhicules de leurs filiales.

Une solution immédiate consiste à traiter différemment les sociétés sans filiale et celle possédant des filiales. Cependant cette différence de traitement entre les deux types de société rend l'application plus complexe et dépendante de la composition interne des sociétés clientes.

Le pattern `Composite` résout ce problème en unifiant l'interface des deux types de sociétés et en utilisant la composition récursive. Cette composition récursive est nécessaire car une société peut posséder des filiales qui possèdent elles-mêmes d'autres filiales. Il s'agit d'une composition en arbre (nous faisons l'hypothèse de l'absence de filiale commune entre deux sociétés) comme illustrée à la figure 12.1 où les sociétés mères sont placées au-dessus de leurs filiales.

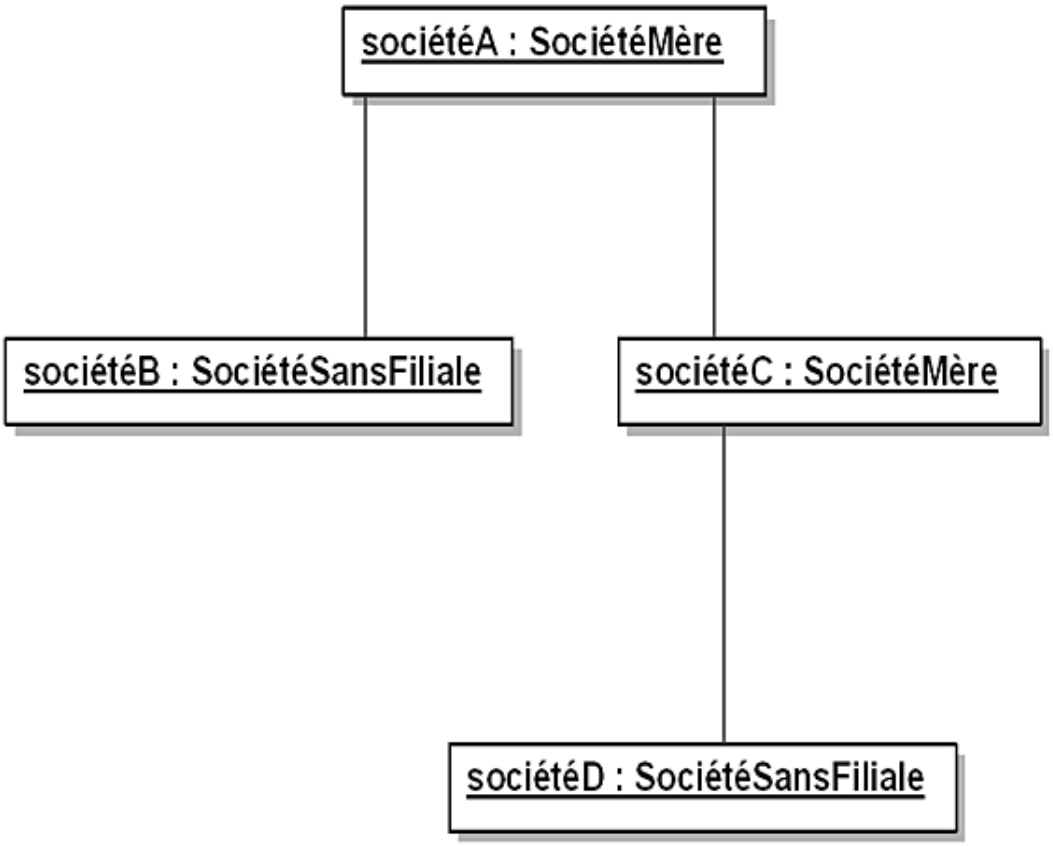


Figure 12.1 - Arbre de sociétés mères et de leurs filiales

La figure 12.2 introduit le diagramme des classes correspondant. La classe abstraite `Société` détient l'interface destinée aux clients. Elle possède deux sous-classes concrètes à savoir `SociétéSansFiliale` et `SociétéMère`, cette dernière détenant une association d'agrégation avec la classe `Société` représentant les liens avec ses filiales.

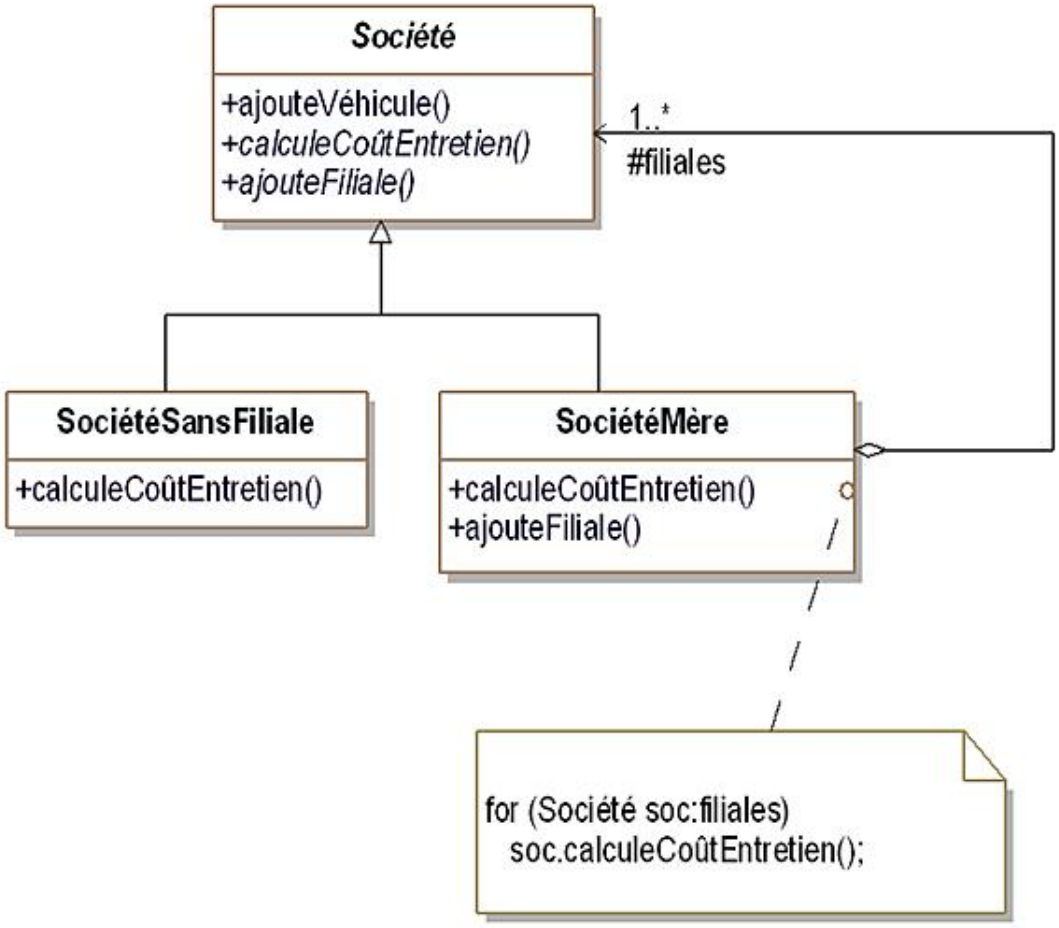


Figure 12.2 - Le pattern `Composite` appliqué à la représentation de sociétés et de leurs filiales

La classe `Société` possède trois méthodes publiques dont une seule est concrète et les deux autres sont abstraites. La méthode concrète est la méthode `ajouteVéhicule` qui ne dépend pas de la composition en filiales de la société. Quant aux deux autres méthodes, elles sont implantées dans les sous-classes concrètes (`ajouteFiliale` ne possède qu'une implantation vide dans `SociétéSansFiliale` donc elle n'est pas représentée dans le diagramme de classes).

Structure

1. Diagramme de classes

La figure 12.3 détaille la structure générique du pattern.

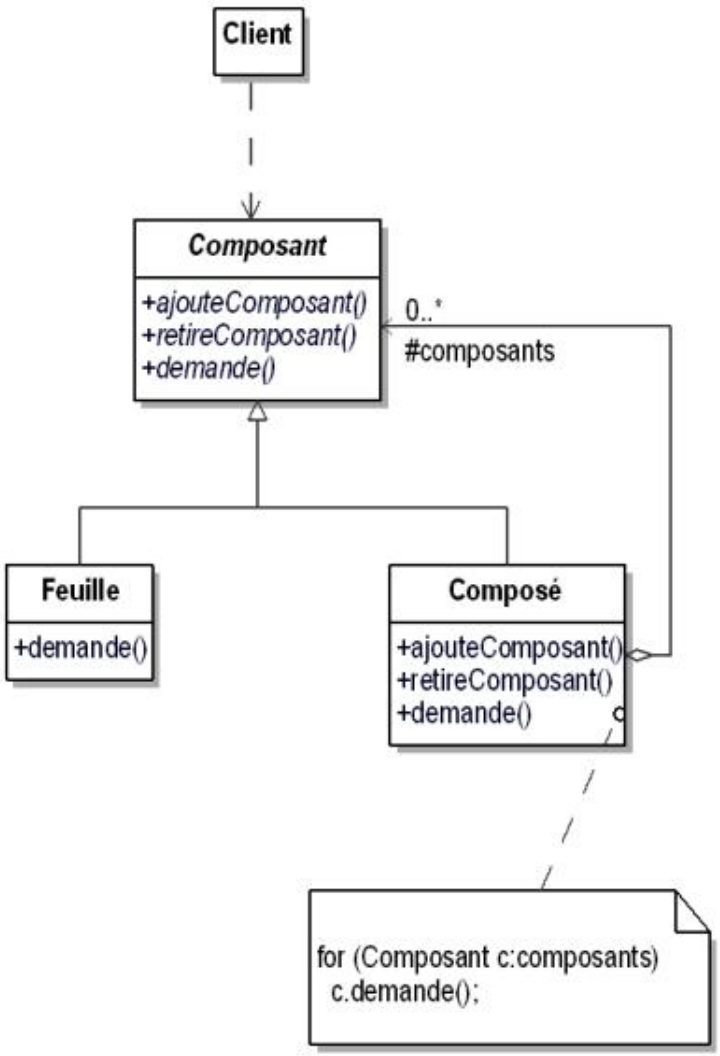


Figure 12.3 - Structure du pattern `Composite`

2. Participants

Les participants au pattern sont les suivants :

- `Composant (Société)`est la classe abstraite qui introduit l'interface des objets de la composition, implante les méthodes communes et introduit la signature des méthodes qui gèrent la composition en ajoutant ou en supprimant des composants ;
- `Feuille (SociétéSansFiliale)`est la classe concrète qui décrit les feuilles de la composition (une feuille ne possède pas de composants) ;
- `Composé (SociétéMère)`est la classe concrète qui décrit les objets composés de la hiérarchie. Cette classe possède une association d'agrégation avec la classe `Composant` ;

- `Client` est la classe des objets qui accèdent aux objets de la composition et qui les manipulent.

3. Collaborations

Les clients envoient leurs requêtes aux composants au travers de l'interface de la classe `Composant`.

Lorsqu'un composant reçoit une requête, il réagit en fonction de sa classe. Si le composant est une feuille, il traite la requête comme illustré à la figure 12.4.

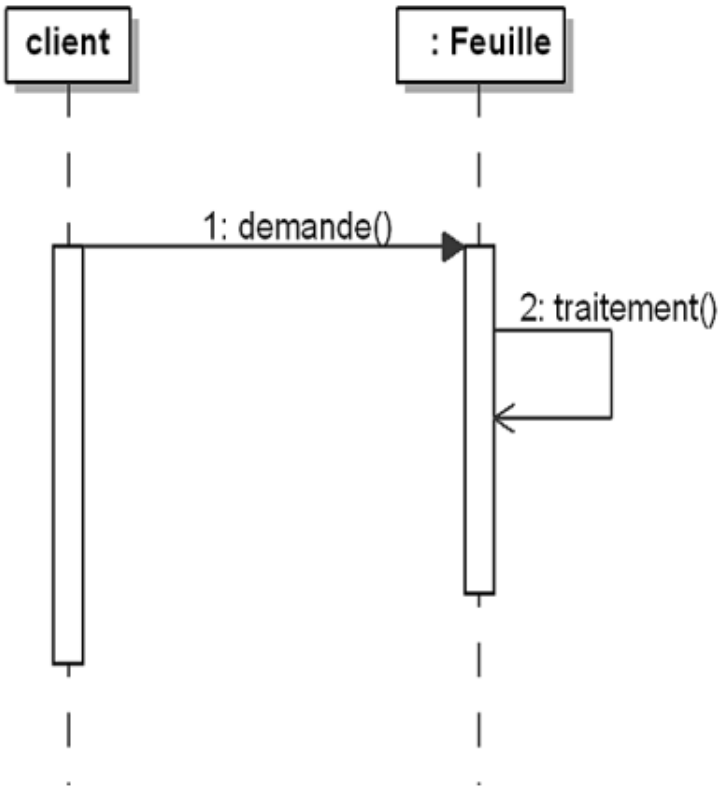


Figure 12.4 - Traitement d'un message par une feuille

Si le composant est une instance de la classe `Composé`, il effectue un traitement préalable puis généralement envoie un message à chacun de ses composants puis effectue un traitement postérieur. La figure 12.5 illustre ce cas avec l'appel récursif à d'autres composants qui vont, à leur tour, traiter cet appel soit comme feuille, soit comme composé.

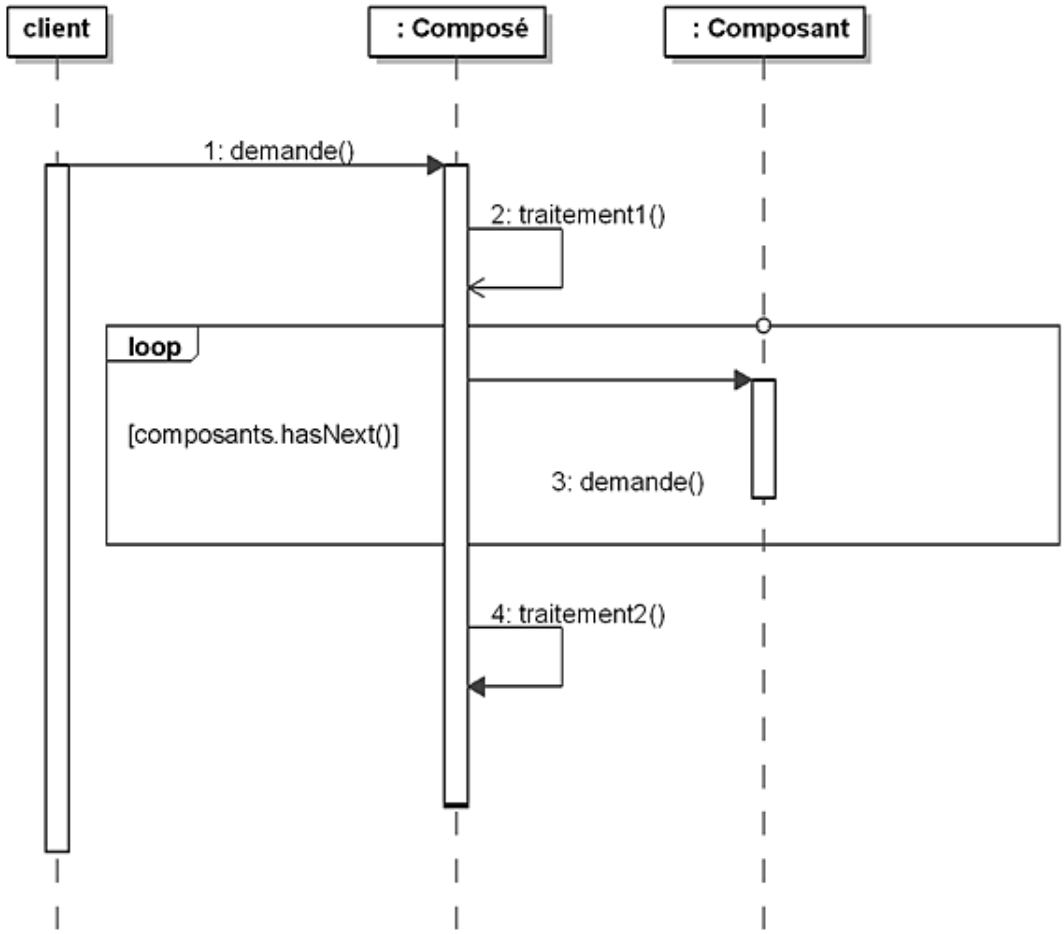


Figure 12.5 -Traitement d'un message par un composé

Domaines d’application

Le pattern est utilisé dans les cas suivants :

- il est nécessaire de représenter au sein d’un système des hiérarchies de composition ;
- les clients d’une composition doivent ignorer s’ils communiquent avec des objets composés ou non.

Exemple en Java

Nous reprenons l'exemple des sociétés et de la gestion de leur parc de véhicules.

La classe abstraite `Societe` est écrite en Java comme suit. Il convient de noter que la méthode `ajouteFiliale` renvoie un résultat booléen qui indique si l'ajout a pu ou non être réalisé.

```
public abstract class Societe
{
  protected static double coutUnitVehicule = 5.0;
  protected int nbrVehicules;

  public void ajouteVehicule()
  {
    nbrVehicules = nbrVehicules + 1;
  }

  public abstract double calculeCoutEntretien();

  public abstract boolean ajouteFiliale(Societe filiale);
}
```

Le code source de la classe `SocieteSansFiliale` est fourni à la suite. Les instances de cette classe ne peuvent pas ajouter de filiales.

```
public class SocieteSansFiliale extends Societe
{
  public boolean ajouteFiliale(Societe filiale)
  {
    return false;
  }

  public double calculeCoutEntretien()
  {
    return nbrVehicules * coutUnitVehicule;
  }
}
```

Ensuite, la classe `SocieteMere` s'écrit en Java comme suit. La méthode intéressante est `calculeCoutEntretien` dont le résultat est la somme du coût des filiales et du coût de la société mère.

```
import java.util.*;
public class SocieteMere extends Societe
{
  protected List<Societe> filiales =
    new ArrayList<Societe>();

  public boolean ajouteFiliale(Societe filiale)
  {
    return filiales.add(filiale);
  }

  public double calculeCoutEntretien()
  {
    double cout = 0.0;
    for (Societe filiale: filiales)
      cout = cout + filiale.calculeCoutEntretien();
    return cout + nbrVehicules * coutUnitVehicule;
  }
}
```

Enfin, nous donnons le code source d'un client. Celui-ci crée une société mère qui possède un véhicule et deux filiales. La première filiale possède un véhicule tandis que la seconde filiale en possède deux. La société mère possède donc quatre véhicules, pour un coût total d'entretien de 20 (le coût pour un véhicule est de 5).

```
public class Utilisateur
{
```

```
  public static void main(String[] args)
  {
    Societe societe1 = new SocieteSansFiliale();
    societe1.ajouteVehicule();
    Societe societe2 = new SocieteSansFiliale();
    societe2.ajouteVehicule();
    societe2.ajouteVehicule();
    Societe groupe = new SocieteMere();
    groupe.ajouteFiliale(societe1);
    groupe.ajouteFiliale(societe2);
    groupe.ajouteVehicule();
    System.out.println(
      " Coût d'entretien total du groupe : " +
      groupe.calculeCoutEntretien());
  }
}
```

L'exécution du programme fournit bien le résultat de 20 :

```
Coût d'entretien total du groupe : 20.0
```

Description

Le but du pattern `Decorator` est d'ajouter dynamiquement des fonctionnalités supplémentaires à un objet. Cet ajout de fonctionnalités ne modifie pas l'interface de l'objet et reste donc transparent vis-à-vis des clients.

Le pattern `Decorator` constitue une alternative par rapport à la création d'une sous-classe pour enrichir un objet.

Exemple

Le système de vente de véhicules dispose d'une classe `VueCatalogue` qui affiche, sous la forme d'un catalogue électronique, les véhicules disponibles sur une page web.

Nous voulons maintenant afficher des données supplémentaires pour les véhicules "haut de gamme", à savoir les informations techniques liées au modèle. Pour réaliser l'ajout de cette fonctionnalité, nous pouvons réaliser une sous-classe d'affichage spécifique pour les véhicules "haut de gamme". Maintenant, nous voulons afficher le logo de la marque des véhicules "moyen et haut de gamme". Il convient alors d'ajouter une nouvelle sous-classe pour ces véhicules, surclasse de la classe des véhicules "haut de gamme", ce qui devient vite complexe. Il est aisé de comprendre ici que l'utilisation de l'héritage n'est pas adaptée à ce qui est demandé pour deux raisons :

- l'héritage est un outil trop puissant pour réaliser un tel ajout de fonctionnalité,
- l'héritage est un mécanisme statique.

Le pattern `Decorator` propose une autre approche qui consiste à ajouter un nouvel objet appelé décorateur qui se substitue à l'objet initial et qui le référence. Ce décorateur possède la même interface ce qui rend la substitution transparente vis-à-vis des clients. Dans notre cas, la méthode `affiche` est alors interceptée par le décorateur qui demande à l'objet initial de s'afficher puis affiche ensuite des informations complémentaires.

La figure 13.1 illustre l'utilisation du pattern `Decorator` pour enrichir l'affichage de véhicules. L'interface `ComposantGraphiqueVéhicule` constitue l'interface commune à la classe `VueVéhicule`, que nous voulons enrichir, et à la classe abstraite `Décorateur`, interface uniquement constituée de la méthode `affiche`.

La classe `Décorateur` possède une référence vers un composant graphique. Cette référence est utilisée par la méthode `affiche` qui délégue l'affichage vers ce composant.

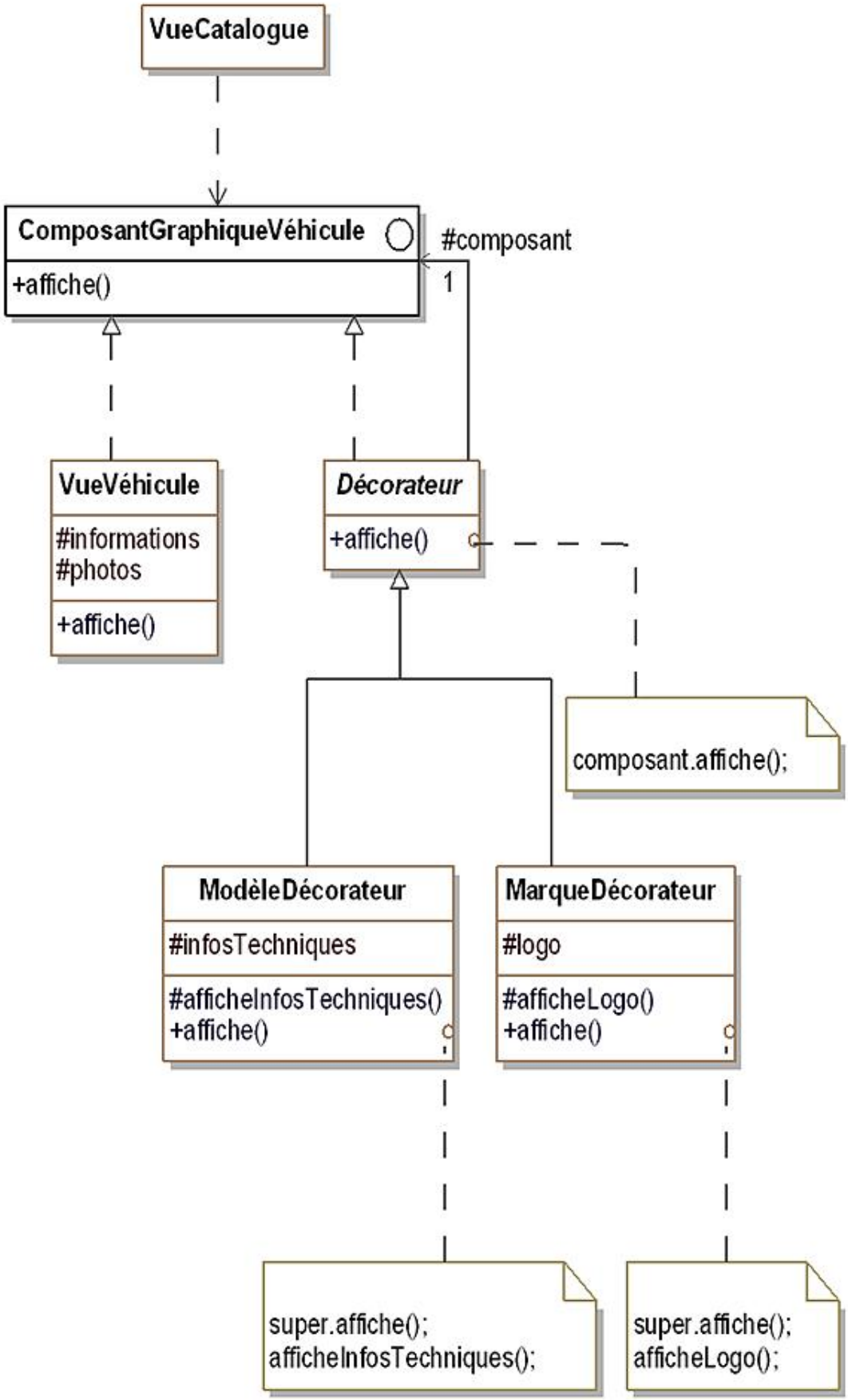


Figure 13.1 - Le pattern `Decorator` pour l'affichage de véhicules dans un catalogue électronique

Il existe deux classes concrètes de décorateur, sous-classes de `Décorateur`. Leur méthode `affiche` commence par

appeler la méthode `affiche` de `Décorateur` puis affiche les données complémentaires comme les informations techniques du modèle ou le logo de la marque.

La figure 13.2 montre la séquence des appels de message destinés à l'affichage d'un véhicule pour lequel le logo de la marque est également affiché.

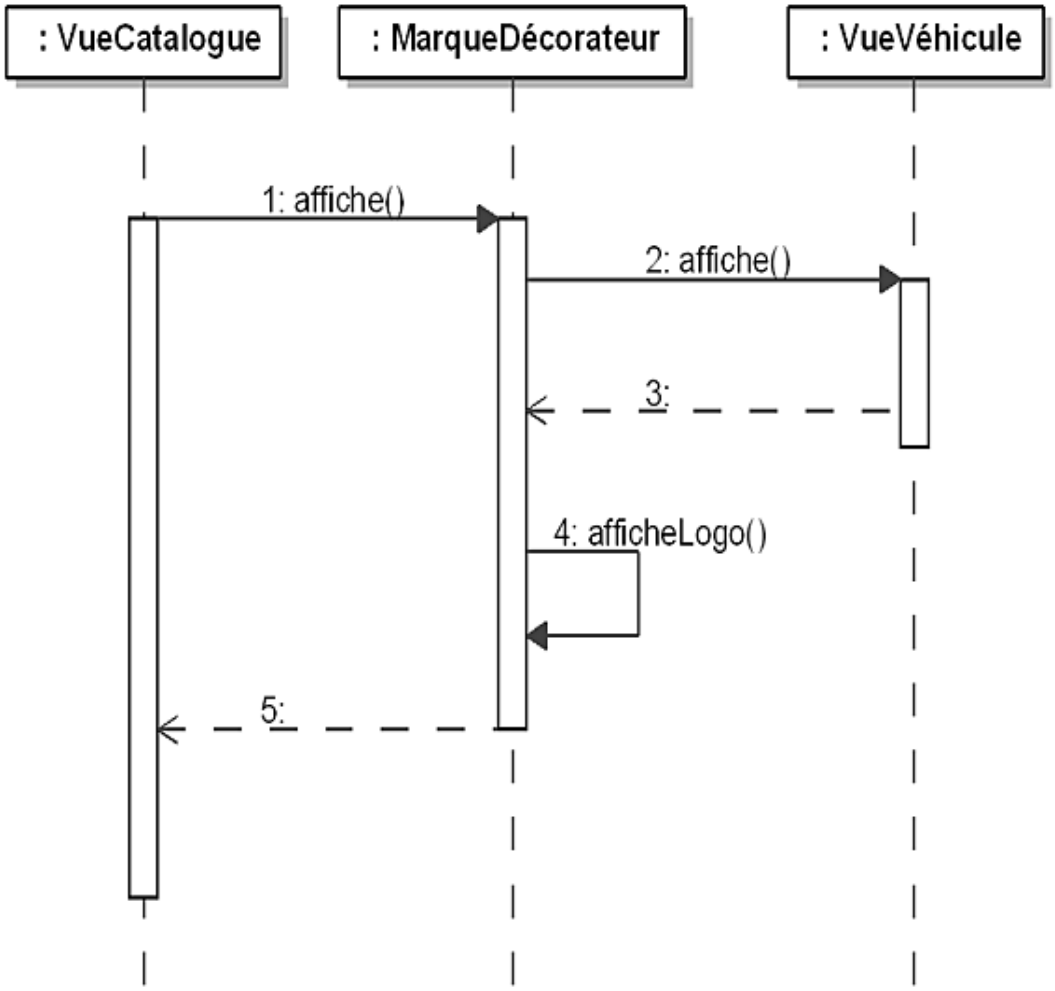


Figure 13.2 - Diagramme de séquence de l'affichage d'un véhicule avec le logo de sa marque

La figure 13.3 montre la séquence des appels de message destinés à l'affichage d'un véhicule pour lequel les informations techniques du modèle et le logo de la marque sont également affichés.

Cette figure illustre bien le fait que les décorateurs sont des composants puisqu'ils peuvent devenir le composant d'un nouveau décorateur, ce qui donne lieu à une chaîne de décorateurs. Cette possibilité de chaîne dans laquelle il est possible d'ajouter ou de retirer dynamiquement un décorateur procure une grande souplesse.

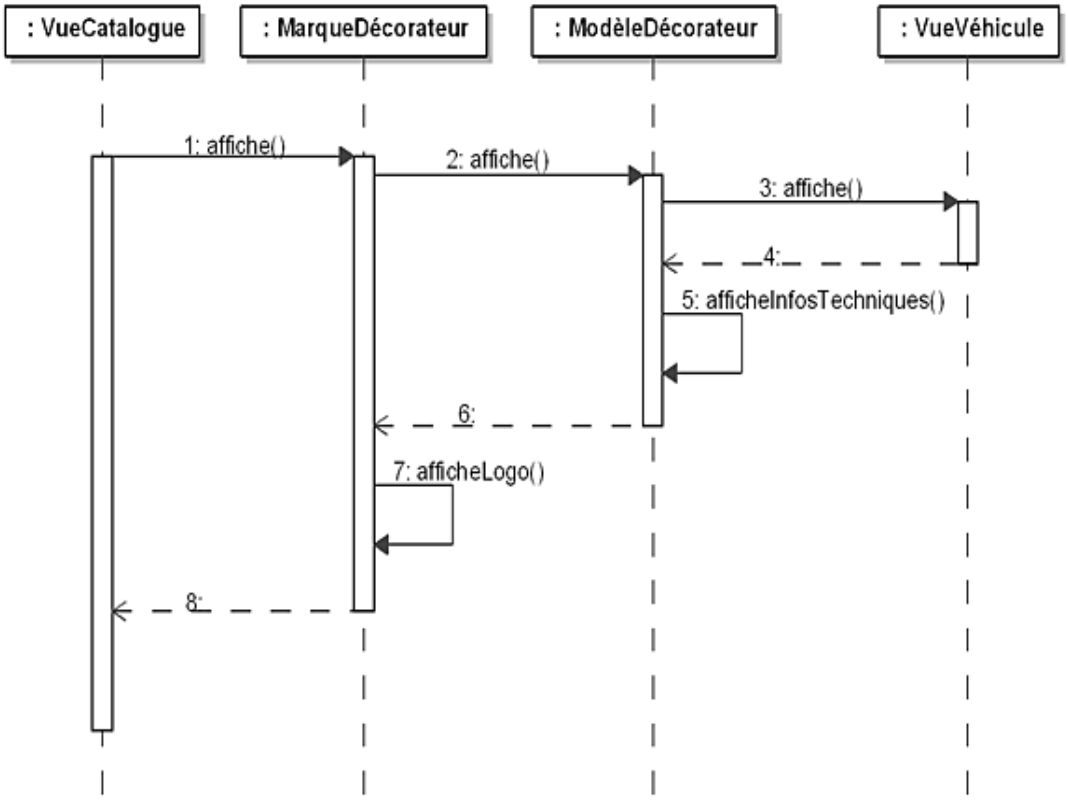


Figure 13.3 - Diagramme de séquence de l'affichage d'un véhicule avec les informations techniques de son modèle et le logo de sa marque

Structure

1. Diagramme de classes

La figure 13.4 détaille la structure générique du pattern.

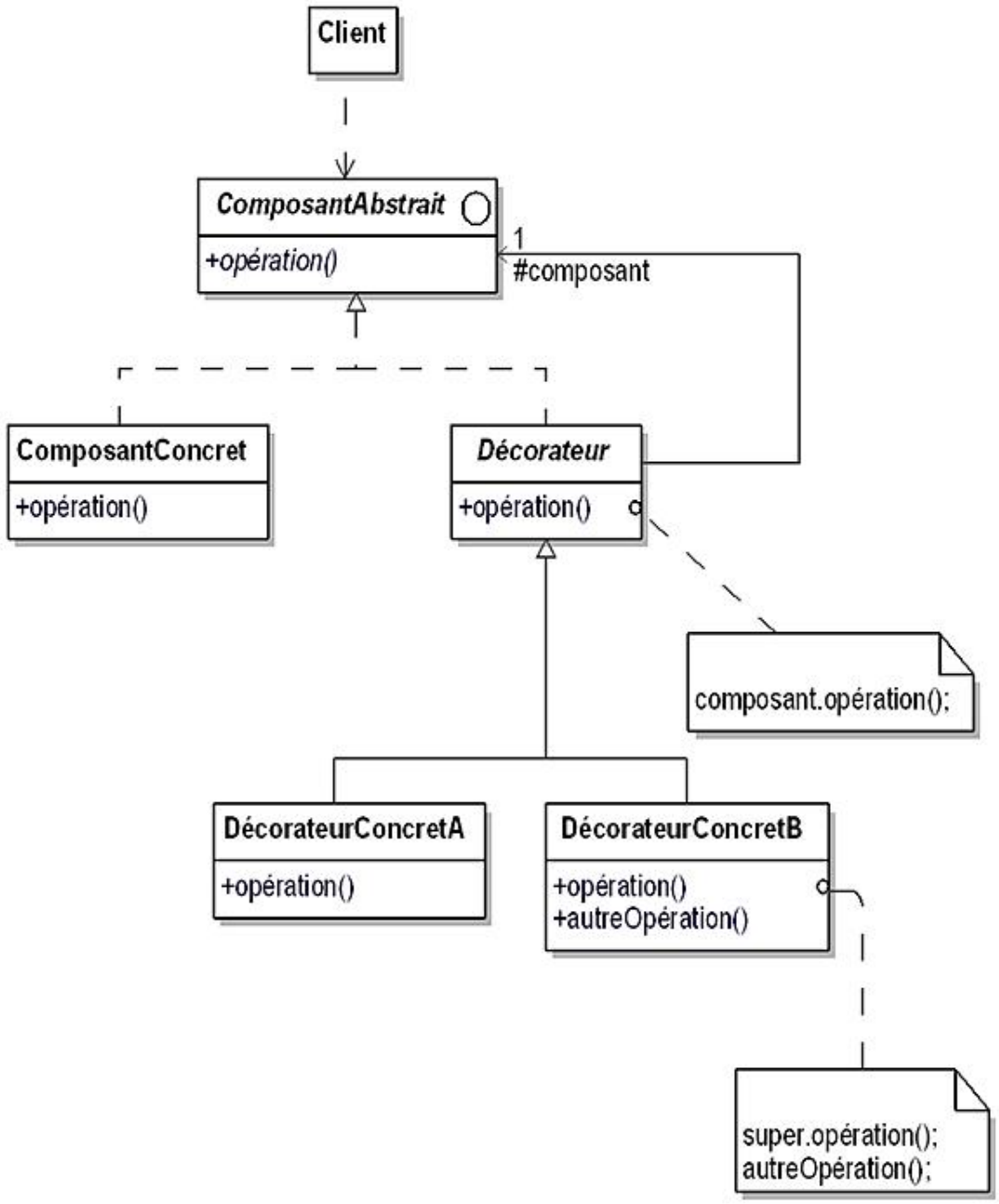


Figure 13.4 - Structure du pattern `Decorator`

2. Participants

Les participants au pattern sont les suivants :

- `ComposantAbstrait` (`ComposantGraphiqueVéhicule`)est l'interface commune au composant et aux décorateurs ;

- `ComposantConcret (VueVéhicule)`est l'objet initial auquel de nouvelles fonctionnalités doivent être ajoutées ;
- `Décorateur` est une classe abstraite qui détient une référence vers un composant ;
- `DécorateurConcretA et DécorateurConcretB (ModèleDécorateur et MarqueDécorateur)`sont des sous-classes concrètes de `Décorateur` qui ont pour but l'implantation des fonctionnalités ajoutées au composant.

3. Collaborations

Le décorateur se substitue au composant. Lorsqu'il reçoit un message destiné à ce dernier, il le redirige au composant en effectuant des opérations préalables ou postérieures à cette redirection.

Domaines d’application

Le pattern `Decorator` peut être utilisé dans les domaines suivants :

- Un système ajoute dynamiquement des fonctionnalités à un objet, sans modifier son interface, c’est-à-dire sans que les clients de cet objet doivent être modifiés.
- Un système gère des fonctionnalités qui peuvent être retirées dynamiquement.
- L’utilisation de l’héritage pour étendre des objets n’est pas pratique, ce qui peut arriver quand leur hiérarchie est déjà complexe.

Exemple en Java

Nous présentons le code source Java de l'exemple, en commençant par l'interface `ComposantGraphiqueVehicule`.

```
public interface ComposantGraphiqueVehicule
{
  void affiche();
}
```

La classe `VueVehicule` implante la méthode `affiche` de l'interface `ComposantGraphiqueVehicule`.

```
public class VueVehicule implements
  ComposantGraphiqueVehicule
{
  public void affiche()
  {
    System.out.println("Affichage du véhicule");
  }
}
```

La classe `Decorateur` implante également la méthode `affiche` en déléguant l'appel. Elle détient un attribut qui contient une référence vers un composant. Ce dernier est passé en paramètre au constructeur de `Decorateur`.

```
public abstract class Decorateur implements
  ComposantGraphiqueVehicule
{
  protected ComposantGraphiqueVehicule composant;

  public Decorateur(ComposantGraphiqueVehicule composant)
  {
    this.composant = composant;
  }

  public void affiche()
  {
    composant.affiche();
  }
}
```

La méthode `affiche` du décorateur concret `ModeleDecorateur` appelle l'affichage du composant (au travers de la méthode `affiche` de `Decorateur`) puis affiche les informations techniques du modèle.

```
public class ModeleDecorateur extends Decorateur
{
  public ModeleDecorateur(ComposantGraphiqueVehicule
    composant)
  {
    super(composant);
  }
  protected void afficheInfosTechniques()
  {
    System.out.println(
      "Informations techniques du modèle");
  }

  public void affiche()
  {
    super.affiche();
    this.afficheInfosTechniques();
  }
}
```

La méthode `affiche` du décorateur concret `MarqueDecorateur` appelle l'affichage du composant puis affiche le logo de la marque.

```
public class MarqueDecorateur extends Decorateur
{
```

```
  public MarqueDecorateur(ComposantGraphiqueVehicule
    composant)
  {
    super(composant);
  }

  protected void afficheLogo()
  {
    System.out.println("Logo de la marque");
  }

  public void affiche()
  {
    super.affiche();
    this.afficheLogo();
  }
}
```

Enfin, la classe `VueCatalogue` est le programme principal. Ce programme crée une vue de véhicule, un décorateur de modèle qui prend la vue comme composant, puis un décorateur de marque qui prend le décorateur de modèle comme composant. Ensuite, ce programme demande l'affichage au décorateur de marque.

```
public class VueCatalogue
{
  public static void main(String[] args)
  {
    VueVehicule vueVehicule = new VueVehicule();
    ModeleDecorateur modeleDecorateur = new
      ModeleDecorateur(vueVehicule);
    MarqueDecorateur marqueDecorateur = new
      MarqueDecorateur(modeleDecorateur);
    marqueDecorateur.affiche();
  }
}
```

Le résultat est le suivant :

```
Affichage du véhicule
Informations techniques du modèle
Logo de la marque
```

Description

L’objectif du pattern Facade est de regrouper les interfaces d’un ensemble d’objets en une interface unifiée rendant cet ensemble plus simple à utiliser pour un client.

Le pattern Facade encapsule l’interface de chaque objet considérée comme interface de bas niveau dans une interface unique de niveau plus élevé. La construction de l’interface unifiée peut nécessiter d’implanter des méthodes destinées à composer les interfaces de bas niveau.

Exemple

Nous voulons offrir la possibilité d'accéder au système de vente de véhicule en tant que service web. Le système est architecturé sous la forme d'un ensemble de composants possédant leur propre interface comme :

- le composant `Catalogue` ;
- le composant `GestionDocument` ;
- le composant `RepriseVéhicule`.

Il est possible de donner l'accès à l'ensemble de l'interface de ces composants aux clients du service web mais cette démarche présente deux inconvénients majeurs :

- certaines fonctionnalités ne sont pas utiles aux clients du service web comme les fonctionnalités d'affichage du catalogue ;
- l'architecture interne du système répond à des exigences de modularité et d'évolution qui ne font pas partie des besoins des clients du service web pour lesquels ces exigences engendrent une complexité inutile.

Le pattern `Facade` résout ce problème en proposant l'écriture d'une interface unifiée plus simple et d'un plus haut niveau d'abstraction. Une classe est chargée d'implanter cette interface unifiée en utilisant les composants du système.

Cette solution est illustrée à la figure 14.1. La classe `WebServiceAuto` offre une interface aux clients du service web. Cette classe et son interface constituent une façade vis-à-vis de ces clients.

L'interface de la classe `WebServiceAuto` est ici constituée de la méthode `chercheVéhicules(prixMoyen, écartMax)` dont le code consiste à appeler la méthode `retrouveVéhicules(prixMin, prixMax)` du catalogue en adaptant la valeur des arguments de cette méthode en fonction du prix moyen et de l'écart maximum.


Il convient de noter que si l'idée du pattern est de constituer une interface de plus haut niveau d'abstraction, rien n'empêche de fournir également dans la façade des accès directs aux méthodes des composants du système.

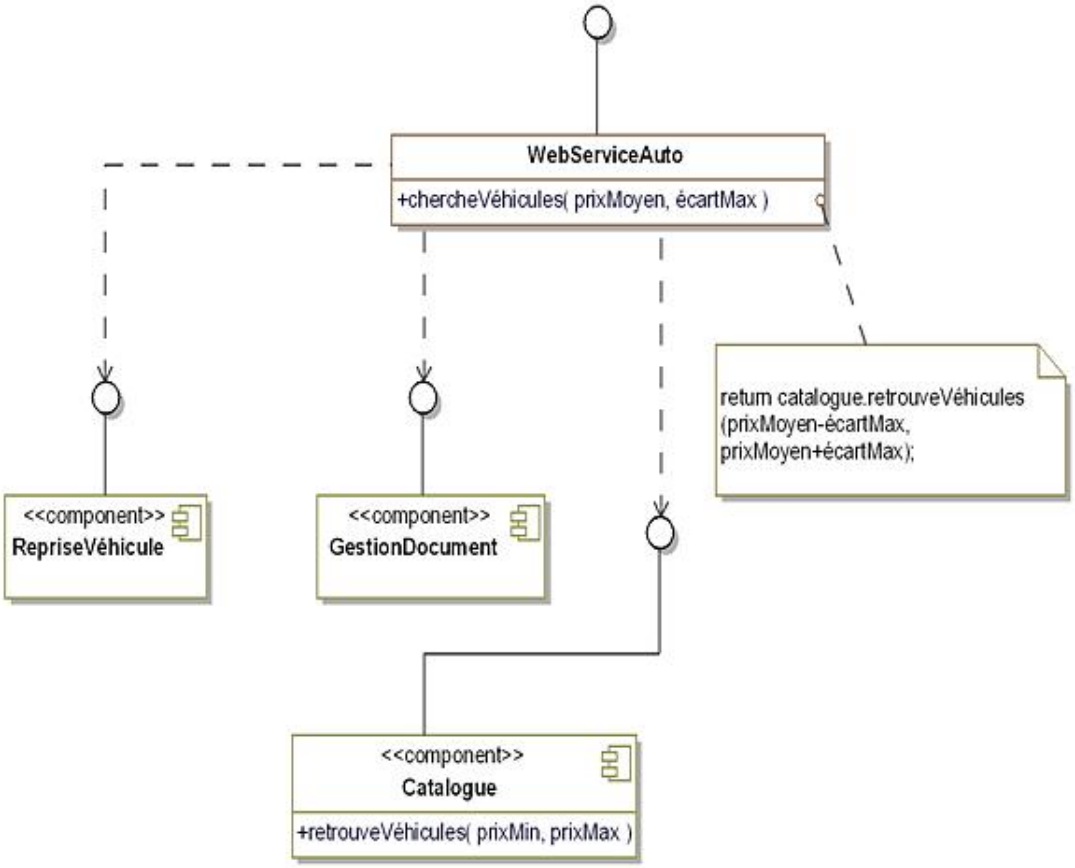


Figure 14.1 - Application du pattern Facade à la mise en œuvre d'un service web du système de vente de véhicules

Structure

1. Diagramme de classes

La figure 14.2 détaille la structure générique du pattern.

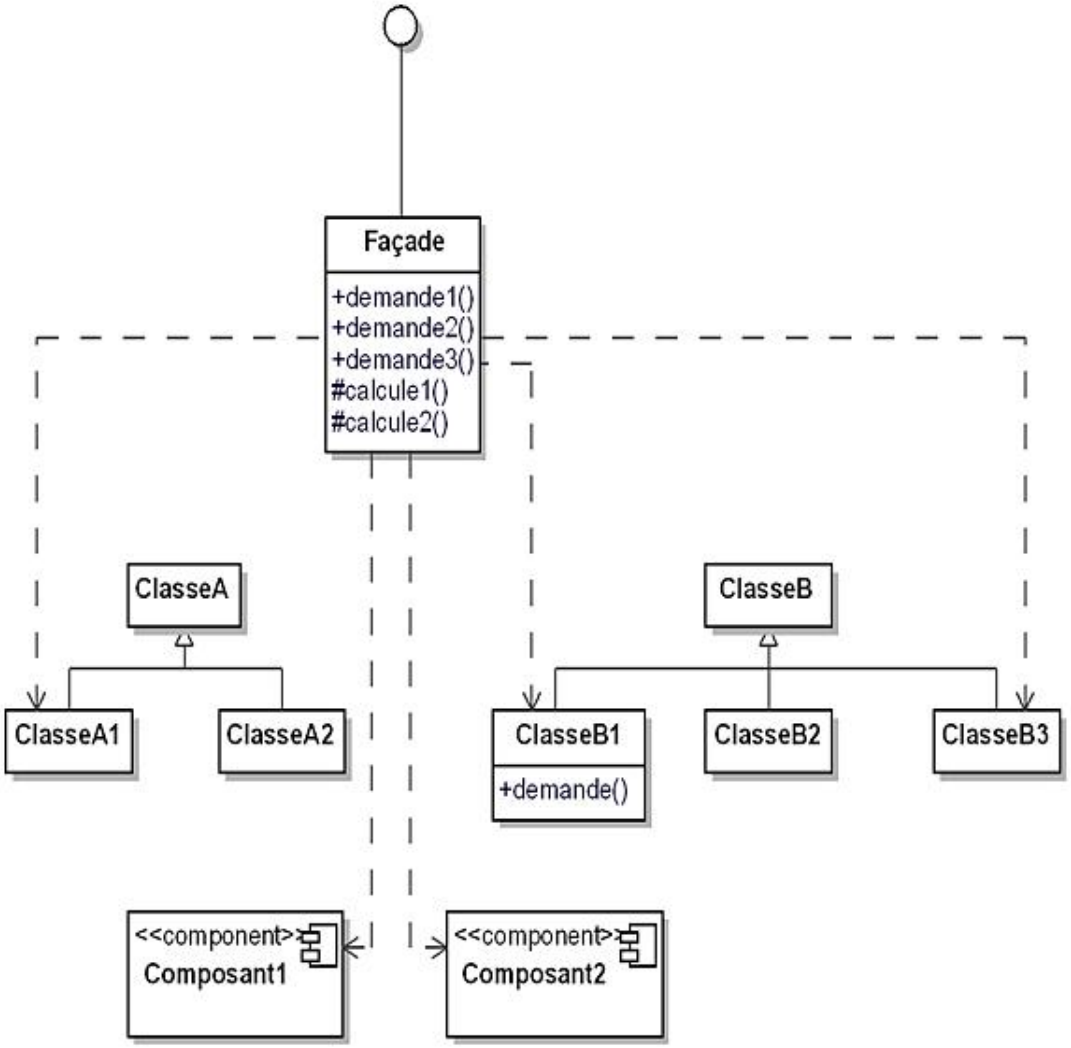


Figure 14.2 - Structure du pattern Facade

2. Participants

Les participants au pattern sont les suivants :

- `Façade` (`WebServiceAuto`) et son interface constituent la partie abstraite exposée aux clients du système. Cette classe possède des références vers les classes et composants constituant le système et dont les méthodes sont utilisées par la façade pour implanter l'interface unifiée ;
- Les classes et composants du système (`RepriseVéhicule`, `GestionDocument` et `Catalogue`) implantent les fonctionnalités du système et répondent aux requêtes de la façade. Elles n'ont pas besoin de la façade pour travailler.

3. Collaborations

Les clients communiquent avec le système au travers de la façade qui se charge, à son tour, d'invoquer les classes et composants du système. La façade ne peut pas se limiter à transmettre des invocations. Elle doit aussi réaliser l'adaptation entre son interface et l'interface des objets du système au moyen de code spécifique. Le diagramme de séquence de la figure 14.3 illustre cette adaptation sur un exemple où du code spécifique à la façade doit être invoqué (méthodes `calcule1` et `calcule2`).

Les clients qui utilisent la façade ne doivent pas accéder directement aux objets du système.

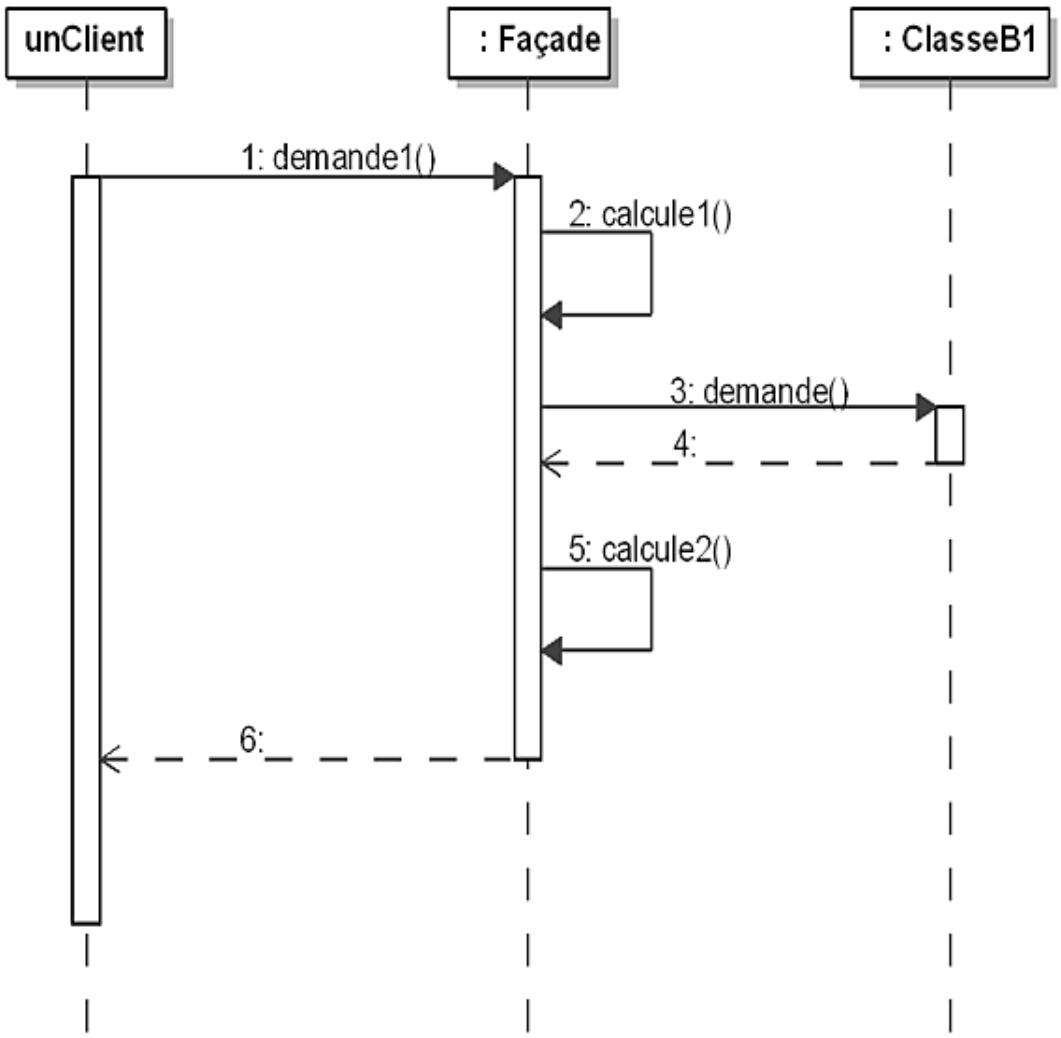


Figure 14.3 - Appel de code spécifique nécessaire pour l'adaptation des méthodes de la façade

Domaines d’application

Le pattern est utilisé dans les cas suivants :

- pour fournir une interface simple d’un système complexe. L’architecture d’un système peut être basée sur de nombreuses petites classes, lui offrant une bonne modularité et des capacités d’évolution. Cependant ces bonnes propriétés du système n’intéressent pas ses clients qui ont besoin d’un accès simple qui répond à leurs exigences ;

- pour diviser un système en sous-systèmes, la communication entre sous-systèmes étant mise en œuvre de façon abstraite de leur implantation grâce aux façades ;

- pour systématiser l’encapsulation de l’implantation d’un système vis-à-vis de l’extérieur.

Exemple en Java

Nous reprenons l'exemple du service web que nous allons simuler à l'aide d'un petit programme Java. Nous donnons d'abord le code source des composants du système et pour commencer celui de la classe `ComposantCatalogue` et de son interface `Catalogue`.

La base de données constituant le catalogue est remplacée par un simple tableau d'objets. La méthode `retrouveVehicules` effectue la recherche d'un ou de plusieurs véhicules en fonction de leur prix à l'aide d'une simple boucle.

```
import java.util.*;
public interface Catalogue
{
  List<String> retrouveVehicules(int prixMin, int
    prixMax);
}

import java.util.*;
public class ComposantCatalogue implements Catalogue
{
  protected Object[] descriptionsVehicule =
  {
    "Berline 5 portes", 6000, "Compact 3 portes", 4000,
    "Espace 5 portes", 8000, "Break 5 portes", 7000,
    "Coupé 2 portes", 9000, "Utilitaire 3 portes", 5000
  };

  public List<String> retrouveVehicules(int prixMin,
    int prixMax)
  {
    int index, taille;
    List<String> resultat = new ArrayList<String>();
    taille = descriptionsVehicule.length / 2;
    for (index = 0; index < taille; index++)
    {
      int prix = ((Integer)descriptionsVehicule[2 * index
        + 1]).intValue();
      if ((prix >= prixMin) && (prix <= prixMax))
        resultat.add((String)descriptionsVehicule[2 *
          index]);
    }
    return resultat;
  }
}
```

Nous continuons avec le composant de gestion de documents constitué de l'interface `GestionDocument` et de la classe `ComposantGestionDocument`. Ce composant constitue une simulation d'une base de documents.

```
public interface GestionDocument
{
  String document(int index);
}

public class ComposantGestionDocument implements
  GestionDocument
{

  public String document(int index)
  {
    return "Document numéro " + index;
  }
}
```

L'interface de la façade appelée `WebServiceAuto` introduit la signature de deux méthodes destinées aux clients du service web.

```
import java.util.List;
public interface WebServiceAuto
```

```
{
  String document(int index);
  List<String> chercheVehicules(int prixMoyen, int ecartMax);
}
```

La classe `WebServiceAutoImpl` implante ces deux méthodes. Remarquons le calcul du prix minimum et maximum pour pouvoir invoquer la méthode `retrouveVehicules` de la classe `ComposantCatalogue`.

```
import java.util.List;
public class WebServiceAutoImpl implements WebServiceAuto
{
  protected Catalogue catalogue = new ComposantCatalogue();
  protected GestionDocument gestionDocument = new
    ComposantGestionDocument();

  public String document(int index)
  {
    return gestionDocument.document(index);
  }

  public List<String> chercheVehicules(int prixMoyen,
    int ecartMax)
  {
    return catalogue.retrouveVehicules(prixMoyen -
      ecartMax, prixMoyen + ecartMax);
  }
}
```

Enfin, un client du service web peut s'écrire en Java comme suit :

```
import java.util.*;
public class UtilisateurWebService
{
  public static void main(String[] args)
  {
    WebServiceAuto webServiceAuto = new
      WebServiceAutoImpl();
    System.out.println(webServiceAuto.document(0));
    System.out.println(webServiceAuto.document(1));
    List<String> resultats =
      webServiceAuto.chercheVehicules(6000, 1000);
    if (resultats.size() > 0)
    {
      System.out.println(
        "Véhicule(s) dont le prix est compris" +
        "entre 5000 et 7000");
      for (String resultat: resultats)
        System.out.println("   " + resultat);
    }
  }
}
```

Ce client affiche deux documents ainsi que les véhicules dont le prix est compris entre 5000 et 7000. Le résultat de l'exécution de ce programme principal est le suivant :

```
Document numéro 0
Document numéro 1
Véhicule(s) dont le prix est compris entre 5000 et 7000
   Berline 5 portes
   Break 5 portes
   Utilitaire 3 portes
```

Description

Le but du pattern `Flyweight` est de partager de façon efficace un ensemble important d'objets de grain fin.

Exemple

Dans le système de vente de véhicules, il est nécessaire de gérer les options que l'acheteur peut choisir lorsqu'il commande un nouveau véhicule.

Ces options sont décrites par la classe `OptionVéhicule` qui contient plusieurs attributs comme le nom, l'explication, un logo, le prix standard, les incompatibilités avec d'autres options, avec certains modèles, etc.

Pour chaque véhicule commandé, il est possible d'associer une nouvelle instance de cette classe. Cependant un grand nombre d'options sont souvent présentes pour chaque véhicule commandé, ce qui oblige le système à gérer un grand ensemble d'objets de petite taille (de grain fin). Cette approche présente toutefois l'avantage de pouvoir stocker au niveau de l'option des informations spécifiques à celle-ci et au véhicule comme le prix de vente de l'option qui peut différer d'un véhicule commandé à un autre.

Cette solution est présentée sur un petit exemple à la figure 15.1 et il est aisé de se rendre compte qu'un grand nombre d'instances de `OptionVéhicule` doit être géré alors que nombre d'entre elles contiennent des données identiques.

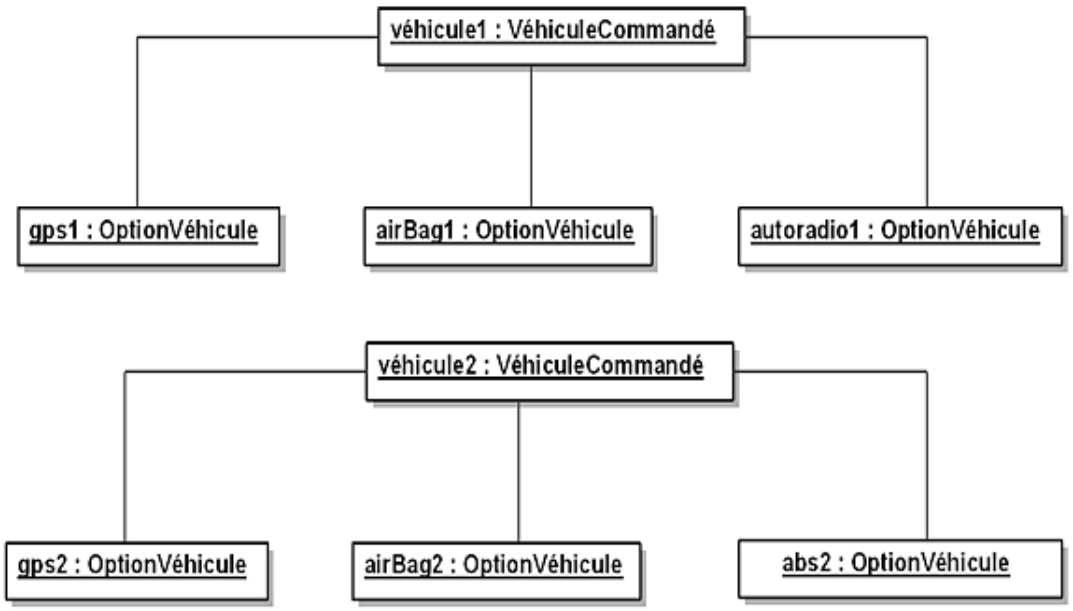


Figure 15.1 - Exemple d'absence de partage d'objets de grain fin

Le pattern `Flyweight` propose une solution à ce problème en partageant les options :

- le partage est réalisé par une fabrique à laquelle le système s'adresse pour obtenir une référence vers une option. Si cette option n'a pas été créée jusqu'à présent, la fabrique procède à sa création avant d'en renvoyer la référence ;
- les attributs d'une option ne contiennent que ses informations spécifiques indépendamment des véhicules commandés : ces informations constituent l'**état intrinsèque**des options ;
- les informations particulières à une option et à un véhicule sont stockées au niveau du véhicule : ces informations constituent l'**état extrinsèque** des options. Elles sont transmises comme paramètres lors des appels des méthodes des options.

Dans le cadre de ce pattern, les options sont les objets appelés flyweights (poids mouche en français).

La figure 15.2 illustre le diagramme des classes de cette solution.

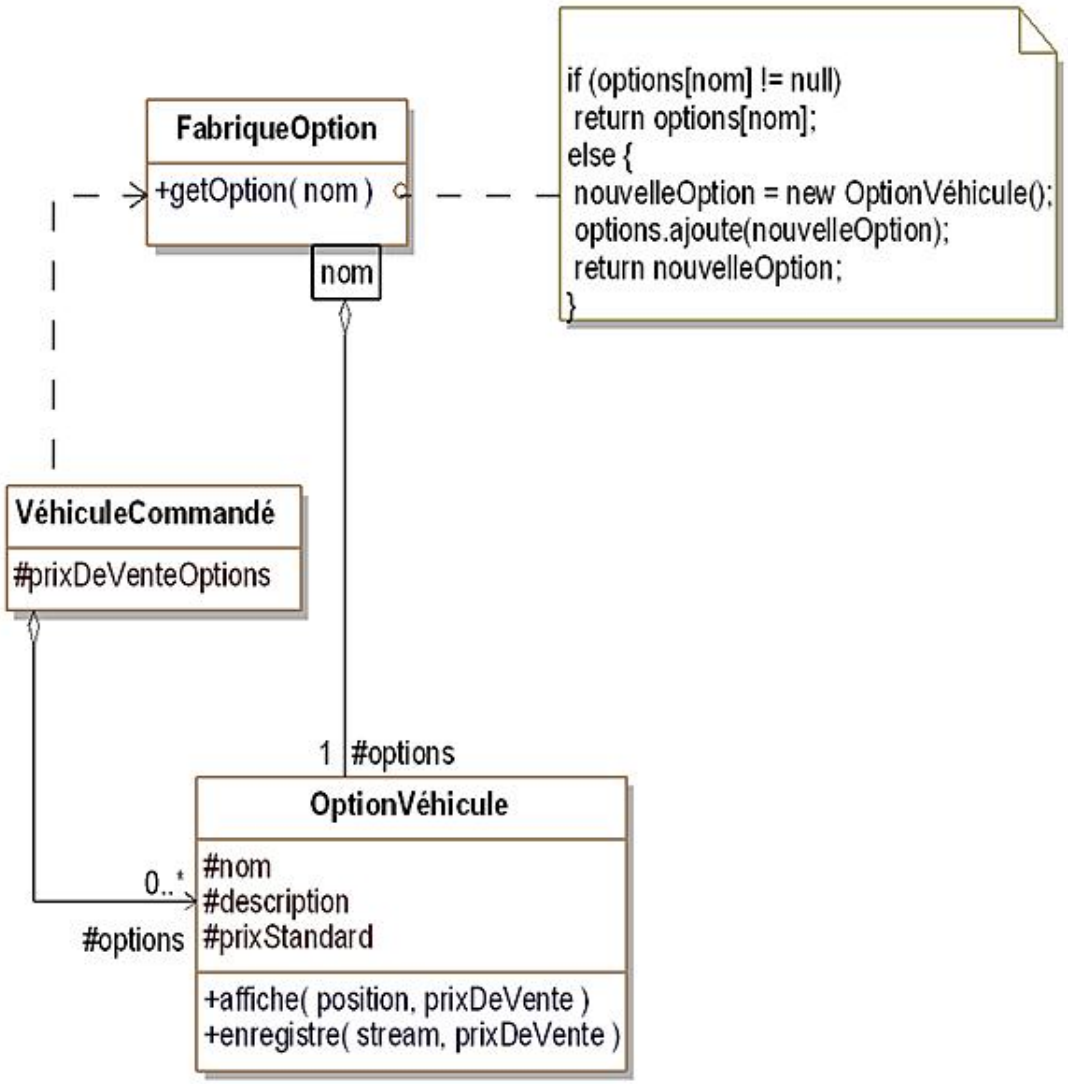


Figure 15.2 - Le pattern `Flyweight` appliqué à des options de véhicules

Ce diagramme introduit les classes suivantes :

- `OptionVéhicule` dont les attributs détiennent l'état intrinsèque d'une option. Les méthodes `affiche` et `enregistre` ont pour paramètre `prixDeVente` qui représente l'état extrinsèque d'une option ;
- `FabriqueOption` dont la méthode `getOption` renvoie une option à partir de son nom. Son fonctionnement consiste à rechercher l'option dans l'association qualifiée et à la créer dans le cas contraire ;
- `VéhiculeCommandé` qui possède une liste des options commandées ainsi que leur prix de vente.

Structure

1. Diagramme de classes

La figure 15.3 détaille la structure générique du pattern.

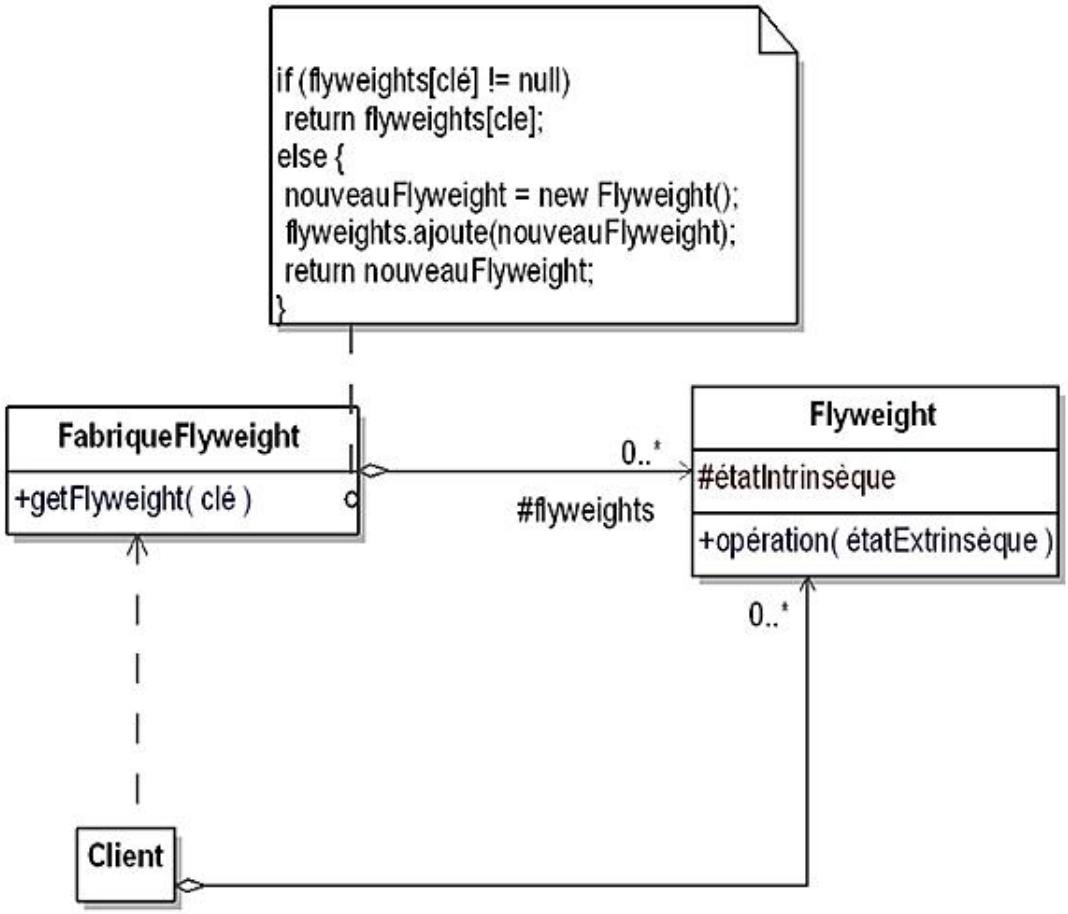


Figure 15.3 - Structure du pattern Flyweight

2. Participants

Les participants au pattern sont les suivants :

- `FabriqueFlyweight (FabriqueOption)`crée et gère les flyweights. La fabrique s'assure que les flyweights sont partagés grâce à la méthode `getFlyweight` qui renvoie les références vers les flyweights ;
- `Flyweight (OptionVéhicule)`détient l'état intrinsèque et implante les méthodes. Ces méthodes reçoivent et déterminent également l'état extrinsèque des flyweights ;
- `Client (VéhiculeCommandé)`contient un ensemble de références vers les flyweights qu'il utilise. Le client doit également détenir l'état extrinsèque de ces flyweights.

3. Collaborations

Les clients ne doivent pas créer eux-mêmes les flyweights mais utiliser la méthode `getFlyweight` de la classe

`FabriqueFlyweight` qui garantit que les flyweights sont partagés.

Lorsqu'un client invoque une méthode d'un flyweight, il doit lui transmettre son état extrinsèque.

Domaine d’application

Le domaine d’application du pattern `Flyweight` est le partage de petits objets (poids mouche). Les critères d’utilisation sont les suivants :

- le système utilise un grand nombre d’objets ;
- le stockage des objets est coûteux à cause d’une grande quantité d’objets ;
- il existe de nombreux ensembles d’objets qui peuvent être remplacés par quelques objets partagés une fois qu’une partie de leur état est rendue extrinsèque.

Exemple en Java

La classe `OptionVehicule` possède un constructeur qui permet de définir l'état intrinsèque de l'option. Dans cet exemple, à part le nom, les autres attributs prennent des valeurs constantes ou basées directement sur le nom. Normalement, ces valeurs devraient provenir d'une base de données.

La méthode `affiche` prend le prix de vente comme paramètre qui constitue l'état extrinsèque.

```
public class OptionVehicule
{
  protected String nom;
  protected String description;
  protected int prixStandard;

  public OptionVehicule(String nom)
  {
    this.nom = nom;
    this.description = "Description de " + nom;
    this.prixStandard = 100;
  }

  public void affiche(int prixDeVente)
  {
    System.out.println("Option");
    System.out.println("Nom : " + nom);
    System.out.println(description);
    System.out.println("Prix standard : " + prixStandard);
    System.out.println("Prix de vente : " + prixDeVente);
  }
}
```

La classe `FabriqueOption` gère le partage des options à l'aide d'un dictionnaire (`TreeMap`) dont la clé d'accès est le nom de l'option. La méthode `getOption` recherche dans ce dictionnaire et si l'option n'est pas trouvée, elle est créée, ajoutée au dictionnaire et retournée.

```
import java.util.*;
public class FabriqueOption
{
  protected Map<String, OptionVehicule> options = new
    TreeMap<String, OptionVehicule>();
  public OptionVehicule getOption(String nom)
  {
    OptionVehicule resultat;
    resultat = options.get(nom);
    if (resultat == null)
    {
      resultat = new OptionVehicule(nom);
      options.put(nom, resultat);
    }
    return resultat;
  }
}
```

La classe `VehiculeCommande` gère la liste des options ainsi que la liste des prix de vente. Ces deux listes sont gérées en parallèle. Le prix de vente d'une option se trouve au même indice dans la liste `prixDeVenteOptions` que l'option dans la liste `option`.

La méthode `ajouteOptions` prend comme paramètres le nom de l'option (état intrinsèque), le prix de vente (état extrinsèque) et la fabrique d'options à utiliser.

Lors de l'appel de la méthode `affiche` d'une option, son prix de vente est transmis en paramètre comme il est possible de s'en apercevoir dans la méthode `afficheOptions`.

```
import java.util.*;
public class VehiculeCommande
{
  protected List<OptionVehicule> options = new
    ArrayList<OptionVehicule>();
```

```
  protected List<Integer> prixDeVenteOptions = new
    ArrayList<Integer>();

  public void ajouteOptions(String nom, int prixDeVente,
    FabriqueOption fabrique)
  {
    options.add(fabrique.getOption(nom));
    prixDeVenteOptions.add(prixDeVente);
  }

  public void afficheOptions()
  {
    int index, taille;
    taille = options.size();
    for (index = 0; index < taille; index++)
    {
      options.get(index).affiche(prixDeVenteOptions.get
        (index).intValue());
      System.out.println();
    }
  }
}
```

Enfin, le code source d'un client est montré à la suite. Il s'agit d'un programme principal qui crée une fabrique d'options, un véhicule commandé, lui ajoute trois options puis les affiche.

```
public class Client
{
  public static void main(String[] args)
  {
    FabriqueOption fabrique = new FabriqueOption();
    VehiculeCommande vehicule = new VehiculeCommande();
    vehicule.ajouteOptions("air bag", 80, fabrique);
    vehicule.ajouteOptions("direction assistée", 90,
      fabrique);
    vehicule.ajouteOptions("vitres électriques", 85,
      fabrique);
    vehicule.afficheOptions();
  }
}
```

Le résultat de l'exécution de ce programme est le suivant (affichage des trois options, avec leurs états intrinsèque et extrinsèque).

```
Option
Nom : air bag
Description de air bag
Prix standard : 100
Prix de vente : 80

Option
Nom : direction assistée
Description de direction assistée
Prix standard : 100
Prix de vente : 90

Option
Nom : vitres électriques
Description de vitres électriques
Prix standard : 100
Prix de vente : 85
```

Description

Le pattern `Proxy` a pour objectif la conception d'un objet qui se substitue à un autre objet (le sujet) et qui en contrôle l'accès.

L'objet qui effectue la substitution possède la même interface que le sujet, ce qui rend cette substitution transparente vis-à-vis des clients.

Exemple

Nous voulons offrir pour chaque véhicule du catalogue la possibilité de visualiser un film qui présente ce véhicule. Un clic sur la photo de la présentation du véhicule permet de jouer ce film.

Une page du catalogue contient de nombreux véhicules et il est très lourd de créer en mémoire tous les objets d'animation car les films nécessitent une grande quantité de mémoire et leur transfert au travers d'un réseau prend beaucoup de temps.

Le pattern `Proxy` offre une solution à ce problème en différant la création des sujets jusqu'au moment où le système a besoin d'eux, ici lors du clic sur la photo du véhicule. Cette solution apporte deux avantages :

- la page du catalogue est chargée beaucoup plus rapidement surtout si elle doit être chargée au travers d'un réseau comme Internet ;
- seuls les films devant être visualisés sont créés, chargés et joués.

L'objet photo est appelé le proxy du film. Il se substitue au film pour l'affichage. Il procède à la création du sujet uniquement lors du clic. Il possède la même interface que l'objet film. La figure 16.1 montre le diagramme des classes correspondant. La classe du proxy, `AnimationProxy`, et la classe du film, `Film`, implantent toutes les deux la même interface, à savoir `Animation`.

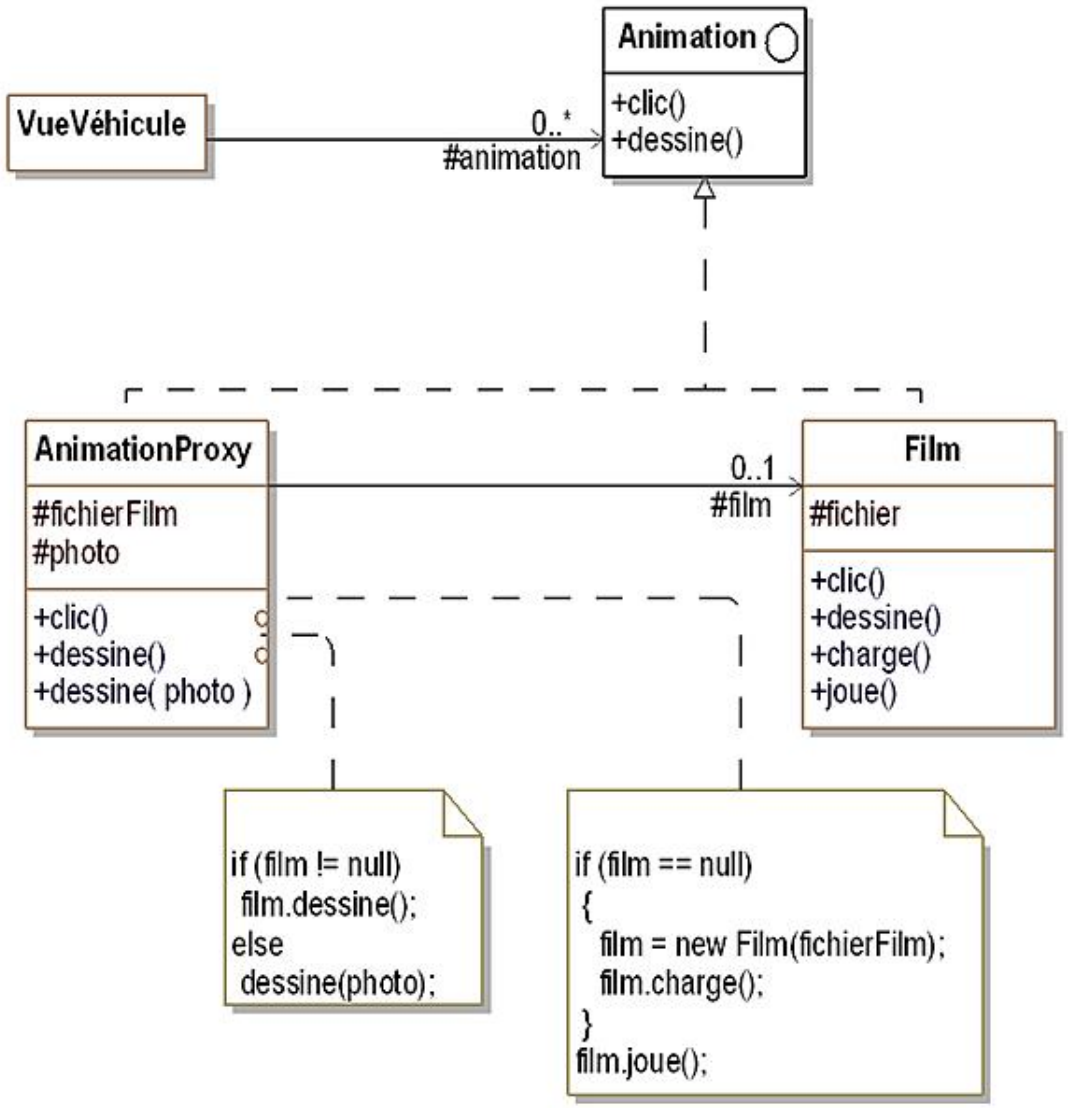


Figure 16.1 - le pattern `Proxy` appliqué à l'affichage d'animations

Quand le proxy reçoit le message `dessine`, il affiche le film si celui-ci a déjà été créé et chargé. Quand le proxy reçoit le

message `clic`, il joue le film après avoir préalablement créé et chargé le film. Le diagramme de séquence pour le message `dessine` est détaillé à la figure 16.2 et à la figure 16.3 pour le message `clic`.

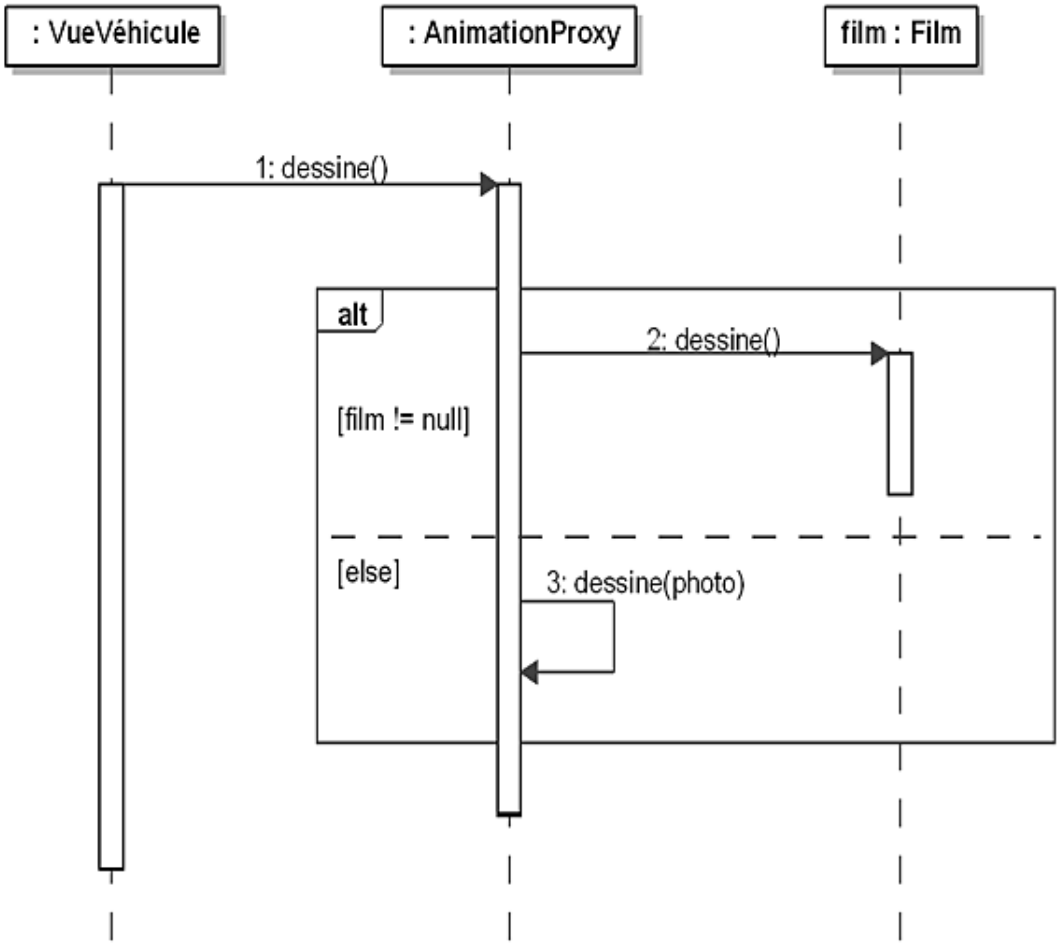


Figure 16.2 - Diagramme de séquence du message `clic`

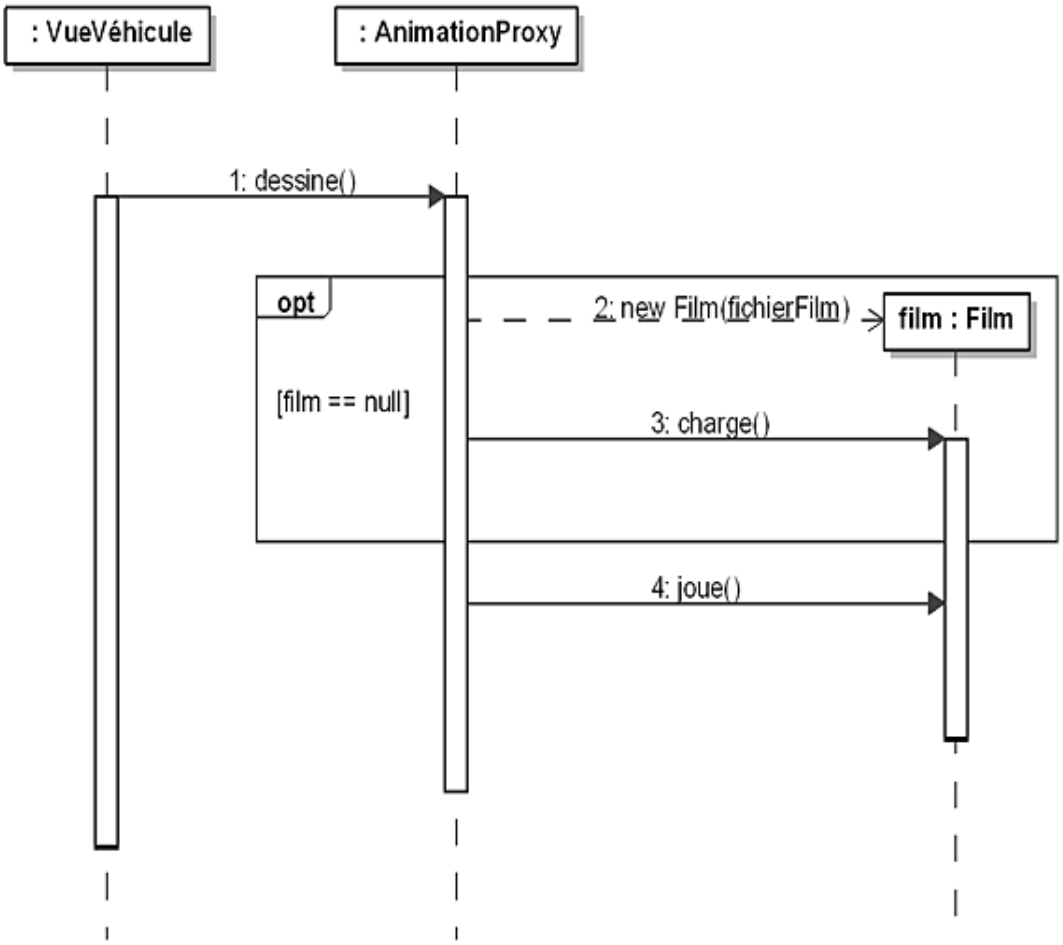


Figure 16.3 - Diagramme de séquence du message `dessine`

Structure

1. Diagramme de classes

La figure 16.4 illustre la structure générique du pattern.

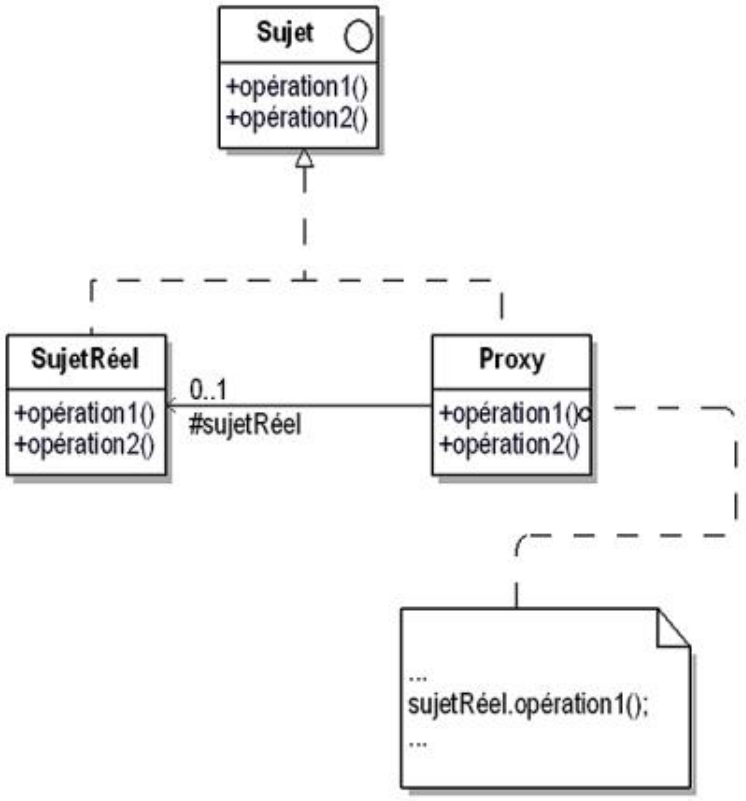


Figure 16.4 - Structure du pattern `Proxy`

Il convient de noter que les méthodes du proxy ont deux comportements possibles quand le sujet réel n'a pas été créé : soit elles créent le sujet réel puis lui déléguent le message (cas de la méthode `clic` de l'exemple), soit elles exécutent un code de substitution (cas de la méthode `dessine` de l'exemple).

2. Participants

Les participants au pattern sont les suivants :

- `Sujet (Animation)`est l'interface commune au proxy et au sujet réel ;
- `SujetRéel (Film)`est l'objet que le proxy contrôle et représente ;
- `Proxy (AnimationProxy)`est l'objet qui se substitue au sujet réel. Il possède une interface identique à ce dernier (interface `Sujet`). Il est chargé de créer et de détruire le sujet réel et de lui déléguer les messages.

3. Collaborations

Le proxy reçoit les appels du client à la place du sujet réel. Quand il le juge approprié, il délégue ces messages au sujet réel. Il doit, dans ce cas, créer préalablement le sujet réel si ce n'est déjà fait.

Domaines d’application

Les proxys sont très utilisés en programmation par objets. Il existe différents types de proxy. Nous en illustrons trois :

- proxy virtuel : permet de créer un objet de taille importante au moment approprié ;
- proxy remote : permet d’accéder à un objet s’exécutant dans un autre environnement. Ce type de proxy est mis en œuvre dans les systèmes d’objets distants (CORBA, Java RMI) ;
- proxy de protection : permet de sécuriser l’accès à un objet, par exemple par des techniques d’authentification.

Exemple en Java

Nous reprenons notre exemple en Java. Le code source de l'interface `Animation` est donné ci-dessous.

```
public interface Animation
{
  void dessine();
  void clic();
}
```

Le code Java de la classe `Film` qui implante cette interface se trouve à la suite. Dans le cadre de la simulation, chaque méthode affiche simplement un message à l'exception de la méthode `clic` qui n'a pas d'action.

```
public class Film implements Animation
{
  public void clic(){}

  public void dessine()
  {
    System.out.println("Affichage du film");
  }

  public void charge()
  {
    System.out.println("Chargement du film");
  }

  public void joue()
  {
    System.out.println("Lecture du film");
  }
}
```

Le code source du proxy, donc de la classe `AnimationProxy` suit. Le code des méthodes correspond à celui spécifié dans le diagramme des classes de la figure 16.1.

```
public class AnimationProxy implements Animation
{
  protected Film film = null;

  public void clic()
  {
    if (film == null)
    {
      film = new Film();
      film.charge();
    }
    film.joue();
  }

  public void dessine()
  {
    if (film != null)
      film.dessine();
    else
      System.out.println("affichage de la photo");
  }
}
```

Enfin, la classe `VueVehicule` qui représente le programme principal s'écrit comme suit.

```
public class VueVehicule
{

  public static void main(String[] args)
  {
    Animation animation = new AnimationProxy();
```

```
    animation.dessine();
    animation.clic();
    animation.dessine();
  }
}
```

L'exécution de ce programme montre la différence de comportement de la méthode `dessine` du proxy selon que la méthode `clic` a été ou non préalablement invoquée.

```
affichage de la photo
Chargement du film
Lecture du film
Affichage du film
```

Présentation

Le concepteur d'un système d'objets est souvent confronté au problème de la découverte des objets. Celle-ci peut être réalisée à partir des deux aspects suivants :

- la structuration des données ;
- la distribution des traitements et des algorithmes.

Les patterns de structuration apportent des solutions aux problèmes de structuration des données et des objets.

L'objectif des patterns de comportement est de fournir des solutions pour distribuer les traitements et les algorithmes entre les objets.

Ces patterns organisent les objets ainsi que leurs interactions en spécifiant les flux de contrôle et de traitement au sein d'un système d'objets.

Distribution par héritage ou par délégation

Une première approche pour distribuer un traitement est de le répartir dans les sous-classes. Cette répartition se fait par l'utilisation dans la classe de méthodes abstraites qui sont implantées dans les sous-classes. Comme une classe peut posséder plusieurs sous-classes, cette approche autorise la possibilité d'obtenir des variantes des parties décrites dans les sous-classes. Cette possibilité est mise en œuvre par le pattern `Template Method` comme l'illustre la figure 17.1.

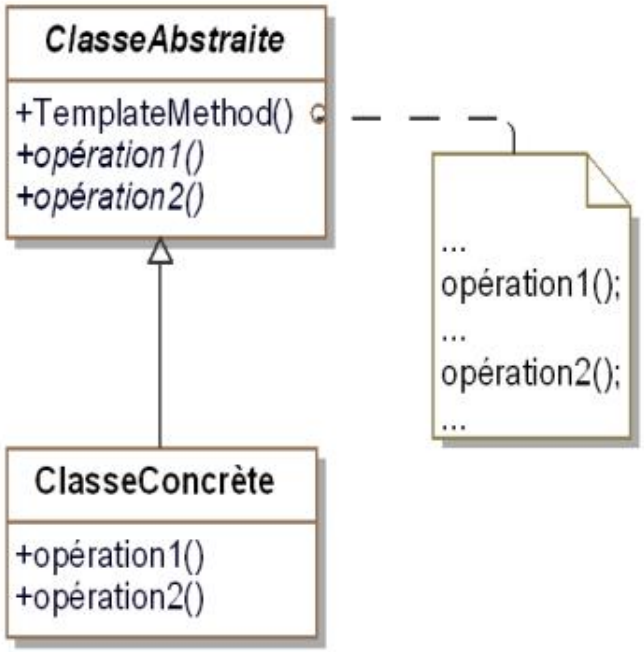


Figure 17.1 - La répartition des traitements par héritage illustrée par le pattern `Template Method`

Une seconde possibilité de répartition est mise en œuvre par la distribution des traitements dans des objets dont les classes sont indépendantes. Dans cette approche, un ensemble d'objets coopérant entre eux concourent à la réalisation d'un traitement ou d'un algorithme. Le pattern `Strategy` illustre ce mécanisme à la figure 17.2. La méthode `demande` de la classe `Entité` invoque pour la réalisation de son traitement la méthode `calcule` spécifiée par l'interface `Stratégie`. Il convient de noter que cette dernière peut avoir plusieurs implantations.

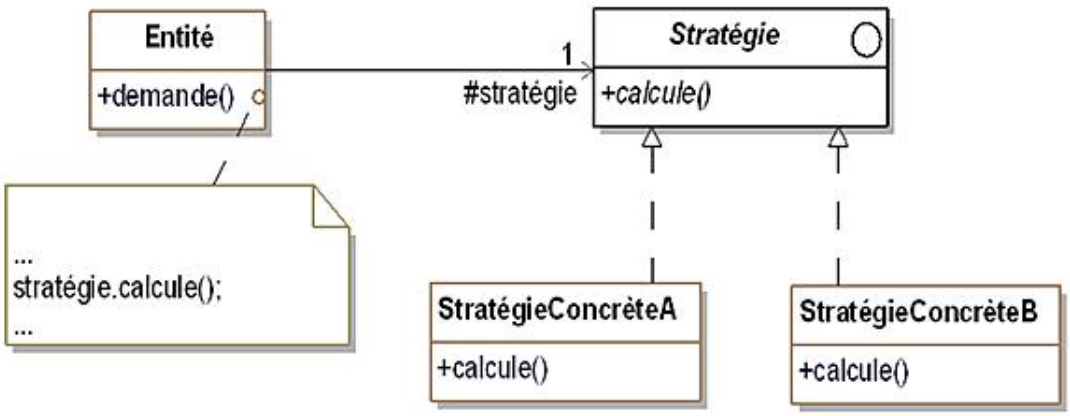


Figure 17.2 - La distribution des traitements entre plusieurs objets illustrés par le pattern `Strategy`

Le tableau suivant indique pour chaque pattern de comportement le type de répartition utilisé.

Chain of Responsibility	Délégation
Command	Délégation

Interpreter	Héritage
Iterator	Délégation
Mediator	Délégation
Memento	Délégation
Observer	Délégation
State	Délégation
Strategy	Délégation
Template Method	Héritage
Visitor	Délégation

Description

Le pattern `Chain of Responsibility` construit une chaîne d'objets telle que si un objet de la chaîne ne peut pas répondre à une requête, il puisse la transmettre à son successeur et ainsi de suite jusqu'à ce que l'un des objets de la chaîne y réponde.

Exemple

Nous nous plaçons dans le cadre de la vente de véhicules d'occasion. Lorsque le catalogue de ces véhicules est affiché, l'utilisateur peut demander une description de l'un des véhicules mis en vente. Si une telle description n'a pas été fournie, le système doit alors renvoyer la description associée au modèle de ce véhicule. Si à nouveau, cette description n'a pas été fournie, il convient de renvoyer la description associée à la marque du véhicule. Une description par défaut est renvoyée s'il n'y a pas non plus de description associée à la marque.

Ainsi, l'utilisateur reçoit la description la plus précise qui est disponible dans le système.

Le pattern `Chain of Responsibility` fournit une solution pour mettre en œuvre ce mécanisme. Celle-ci consiste à lier les objets entre eux du plus spécifique (le véhicule) au plus général (la marque) pour former la chaîne de responsabilité. La requête de description est transmise le long de cette chaîne jusqu'à ce qu'un objet puisse la traiter et renvoyer la description.

Le diagramme d'objets UML de la figure 18.1 illustre cette situation et montre les différentes chaînes de responsabilité (de la gauche vers la droite).

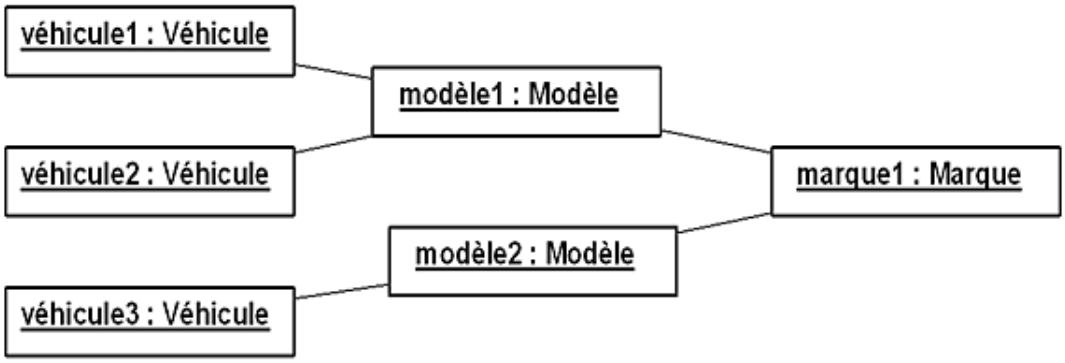


Figure 18.1 - Diagramme d'objets de véhicules, modèles et marques avec les liens de la chaîne de responsabilité

La figure 18.2 représente le diagramme des classes du pattern `Chain of Responsibility` appliqué à l'exemple. Les véhicules, modèles et marques sont décrits par des sous-classes concrètes de la classe `ObjetBase`. Cette classe abstraite introduit l'association `suivant` qui implante la chaîne de responsabilité. Elle introduit également trois méthodes :

- `description` est une méthode abstraite. Elle est implantée dans les sous-classes concrètes. Cette implantation doit retourner soit la description si elle existe soit la valeur `null` dans le cas contraire ;
- `descriptionParDéfaut` retourne une valeur de description par défaut, valable pour tous les véhicules du catalogue ;
- `getDescription` est la méthode publique destinée à l'utilisateur. Elle invoque la méthode `description`. Si le résultat est `null`, alors s'il y a un objet `suivant`, sa méthode `getDescription` est invoquée à son tour sinon c'est la méthode `descriptionParDéfaut` qui est utilisée.

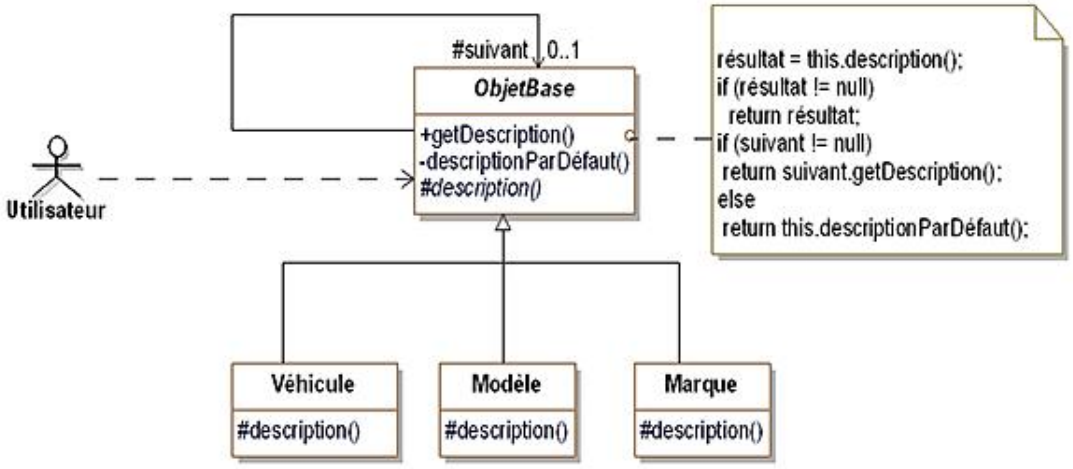


Figure 18.2 - Le pattern `Chain of Responsibility` pour organiser la description de véhicules d'occasion

La figure 18.3 montre un diagramme de séquence qui est un exemple de requête d'une description basée sur le diagramme d'objets de la figure 18.1.

Dans cet exemple, ni le `véhicule1`, ni le `modèle1` ne possèdent de description. Seule `marque1` possède une description qui est donc utilisée pour le `véhicule1`.

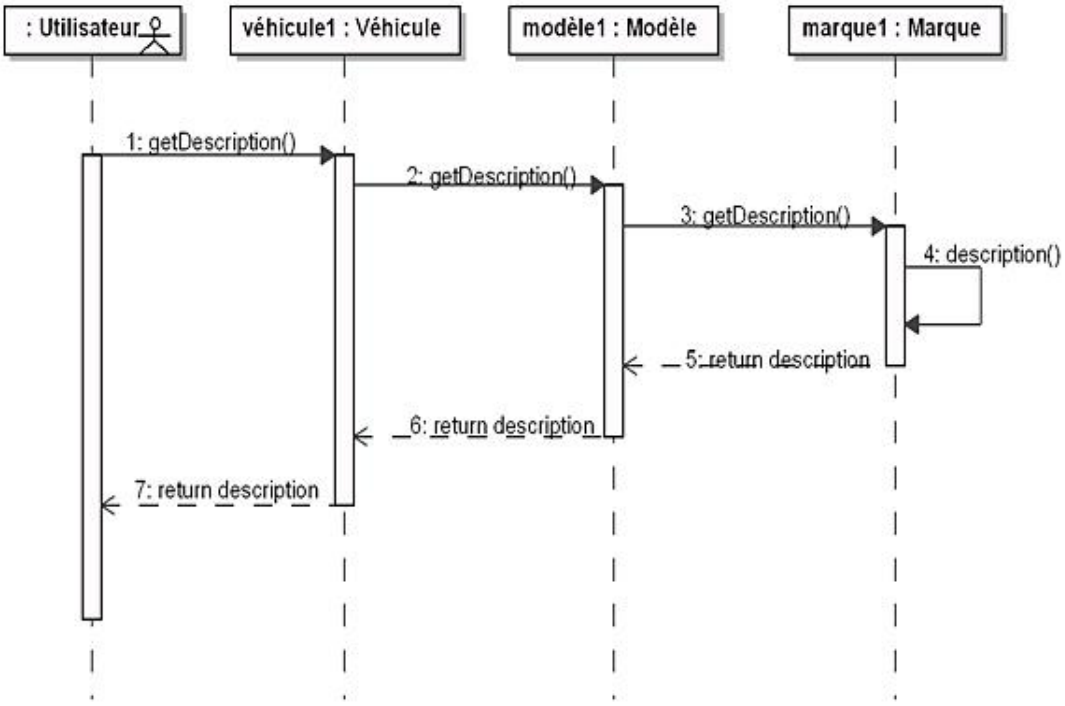


Figure 18.3 - Diagramme de séquence illustrant sur un exemple le pattern `Chain of Responsibility`

Structure

1. Diagramme de classes

La figure 18.4 détaille la structure générique du pattern.

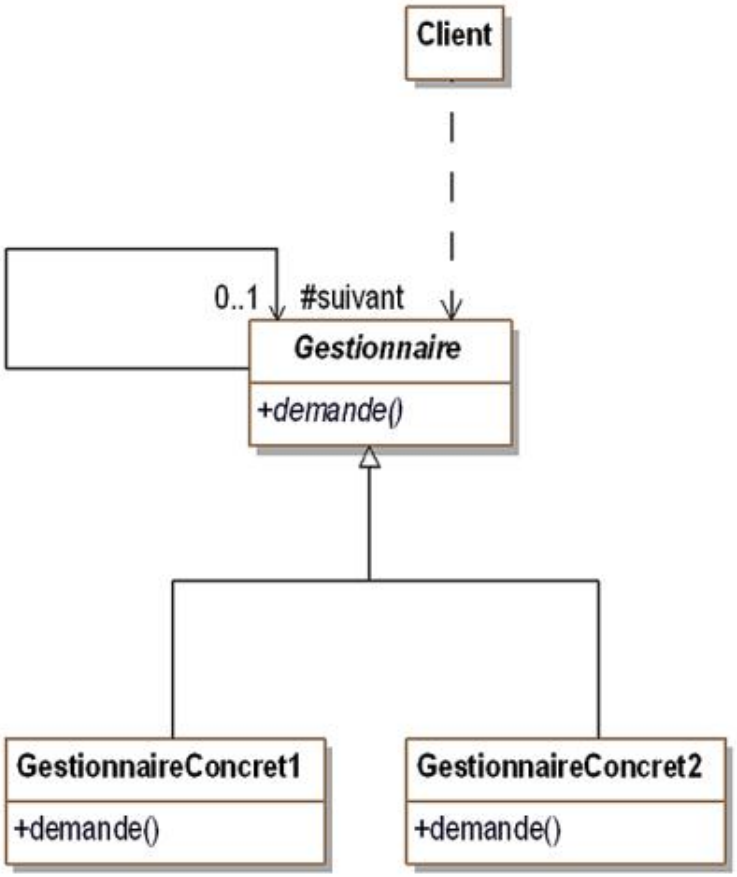


Figure 18.4 - Structure du pattern `Chain of Responsibility`

2. Participants

Les participants au pattern sont les suivants :

- `Gestionnaire (ObjetBase)`est une classe abstraite qui implante sous forme d'une association la chaîne de responsabilité ainsi que l'interface des requêtes ;
- `GestionnaireConcret1`et `GestionnaireConcret2` (`Véhicule`, `Modèle` et `Marque`) sont les classes concrètes qui implantent le traitement des requêtes en utilisant la chaîne de responsabilité si elles ne peuvent pas les traiter ;
- `Client (Utilisateur)`initie la requête initiale auprès d'un objet de l'une des classes `GestionnaireConcret1` ou `GestionnaireConcret2`.

3. Collaborations

Le client effectue la requête initiale auprès d'un gestionnaire. Cette requête est propagée le long de la chaîne de responsabilité jusqu'au moment où l'un des gestionnaires la traite.

Domaines d’application

Le pattern est utilisé dans les cas suivants :

- une chaîne d’objets gère une requête selon un ordre qui est défini dynamiquement ;
- la façon dont une chaîne d’objets gère une requête ne doit pas être connue de ses clients.

Exemple en Java

Nous introduisons maintenant l'exemple en langage Java. La classe `ObjetBase` est décrite à la suite. Elle implante la chaîne de responsabilité par l'attribut `suivant` dont la valeur peut être fixée par la méthode `setSuivant`. Les autres méthodes correspondent aux spécifications introduites dans la figure 18.2.

```
public abstract class ObjetBase
{
  protected ObjetBase suivant;

  private String descriptionParDefaut()
  {
    return "description par défaut";
  }

  protected abstract String description();

  public String getDescription()
  {
    String resultat;
    resultat = this.description();
    if (resultat != null)
      return resultat;
    if (suivant != null)
      return suivant.getDescription();
    else
      return this.descriptionParDefaut();
  }

  public void setSuivant(ObjetBase suivant)
  {
    this.suivant = suivant;
  }
}
```

Les trois sous-classes concrètes de la classe `ObjetBase` sont `Vehicule`, `Modele` et `Marque`, leur code source Java est présenté à la suite. La classe `Vehicule` gère une description simple fournie au moment de sa construction (le paramètre `null` est utilisé en cas d'absence de description).

```
public class Vehicule extends ObjetBase
{
  protected String description;

  public Vehicule(String description)
  {
    this.description = description;
  }

  protected String description()
  {
    return description;
  }
}
```

La classe `Modele` gère une description et un nom.

```
public class Modele extends ObjetBase
{
  protected String description;
  protected String nom;

  public Modele(String nom, String description)
  {
    this.description = description;
    this.nom = nom;
  }
```

```
  protected String description()
  {
    if (description != null)
      return "Modèle " + nom + " : " + description;
    else
      return null;
  }
}
```

La classe `Marque` gère deux descriptions et un nom.

```
public class Marque extends ObjetBase
{
  protected String description1, description2;
  protected String nom;

  public Marque(String nom, String description1, String
    description2)
  {
    this.description1 = description1;
    this.description2 = description2;
    this.nom = nom;
  }

  protected String description()
  {
    if ((description1 != null) && (description2 != null))
      return "Marque " + nom + " : " + description1 + " "
        + description2;
    else if (description1 != null)
      return "Marque " + nom + " : " + description1;
    else
      return null;
  }
}
```

Enfin, la classe `Utilisateur` représente le programme principal.

```
public class Utilisateur
{

  public static void main(String[] args)
  {
    ObjetBase vehicule1 = new Vehicule(
      "Auto++ KT500 Véhicule d'occasion en bon état ");
    System.out.println(vehicule1.getDescription());
    ObjetBase modele1 = new Modele("KT400",
      "Le véhicule spacieux et confortable");
    ObjetBase vehicule2 = new Vehicule(null);
    vehicule2.setSuivant(modele1);
    System.out.println(vehicule2.getDescription());
    ObjetBase marque1 = new Marque("Auto++",
      "La marque des autos", "de grande qualité");
    ObjetBase modele2 = new Modele("KT700", null);
    modele2.setSuivant(marque1);
    ObjetBase vehicule3 = new Vehicule(null);
    vehicule3.setSuivant(modele2);
    System.out.println(vehicule3.getDescription());
    ObjetBase vehicule4 = new Vehicule(null);
    System.out.println(vehicule4.getDescription());
  }
}
```

Le résultat de l'exécution de programme produit le résultat suivant.

```
Auto++ KT500 Véhicule d'occasion en bon état
Modèle KT400 : Le véhicule spacieux et confortable
Marque Auto++ : La marque des autos de grande qualité
description par défaut
```

Description

Le pattern `Command` a pour objectif de transformer une requête en un objet, facilitant des opérations comme l'annulation, la mise en file des requêtes et leur suivi.

Exemple

Dans certains cas, la gestion d'une commande peut être assez complexe : elle peut être annulable, mise dans une file d'attente ou être tracée. Dans le cadre du système de vente de véhicules, le gestionnaire peut demander au catalogue de solder les véhicules d'occasion présents dans le stock depuis une certaine durée. Pour des raisons de facilité, cette demande doit pouvoir être annulée puis, éventuellement, rétablie.

Pour gérer cette annulation, une première solution consiste à indiquer au niveau de chaque véhicule s'il est ou non soldé. Cette solution n'est pas suffisante car un même véhicule peut être soldé plusieurs fois avec des taux différents. Une autre solution serait alors de conserver son prix avant la dernière remise, mais cette solution n'est pas non plus satisfaisante car l'annulation peut porter sur une autre requête de remise que la dernière.

Le pattern `Command` résout ce problème en transformant la requête en un objet dont les attributs vont contenir les paramètres ainsi que l'ensemble des objets sur lesquels la requête a été appliquée. Dans notre exemple, il devient ainsi possible d'annuler ou de rétablir une requête de remise.

La figure 19.1 illustre cette application du pattern `Command` à notre exemple. La classe `CommandeSolder` stocke ses deux paramètres (`pourcentage` et `duréeStock`) ainsi que la liste des véhicules pour lesquels la remise a été appliquée (association `véhiculesSoldés`).

Il faut noter que l'ensemble des véhicules référencés par `CommandeSolder` est un sous-ensemble de l'ensemble des véhicules référencés par `Catalogue`.

Lors de l'appel de la méthode lance `CommandeSolder`, la commande passée en paramètre est exécutée puis elle est stockée dans un ordre tel que la dernière commande stockée se retrouve en première position.

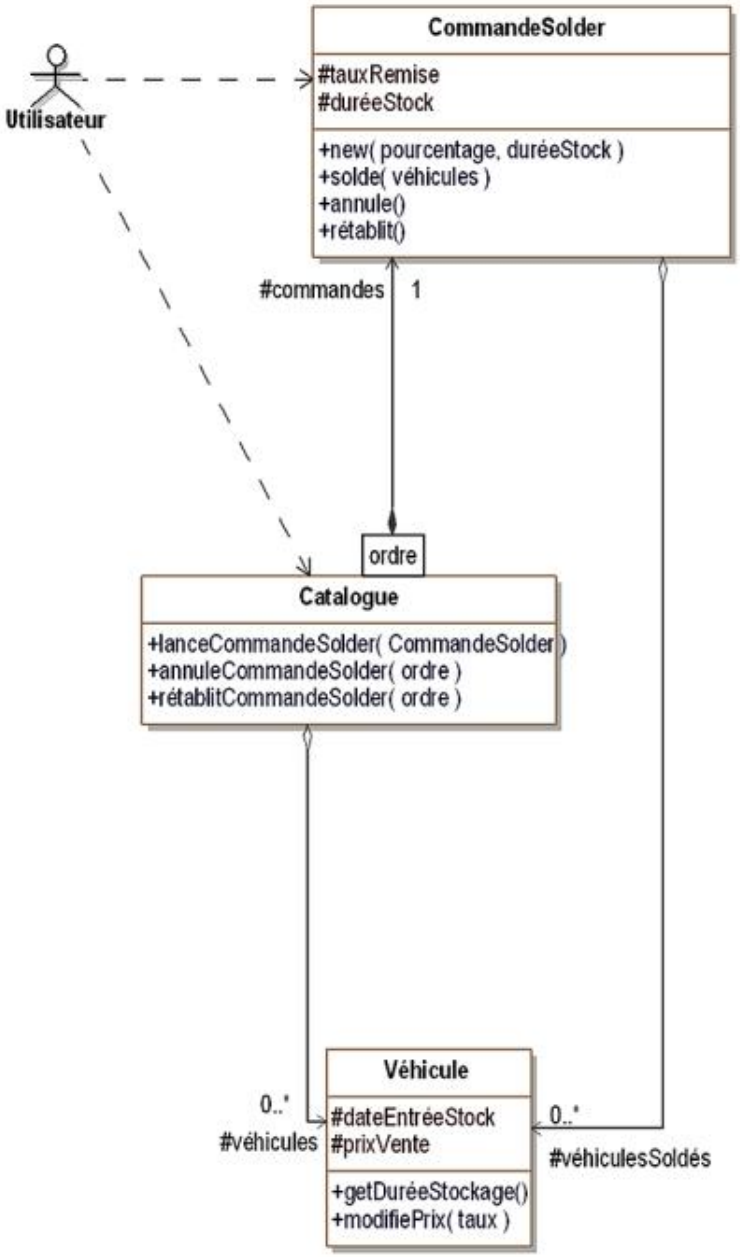


Figure 19.1 - Le pattern `Command` appliqué à la gestion des remises appliquées à des véhicules d'occasion

Le diagramme de la figure 19.2 montre un exemple de séquence d'appels. Les deux paramètres fournis au constructeur de la classe `CommandeSolder` sont le taux de remise et la durée minimale de stockage exprimée en mois. Par ailleurs, le paramètre `ordre` de la méthode `annuleCommandeSolder` vaut zéro pour la dernière commande exécutée, un pour l'avant-dernière, etc.

Les interactions entre les instances de `CommandeSolder` et de `Véhicule` ne sont pas représentées dans un but de simplification. Pour bien comprendre leur fonctionnement, il convient de se reporter au code Java présenté plus loin.

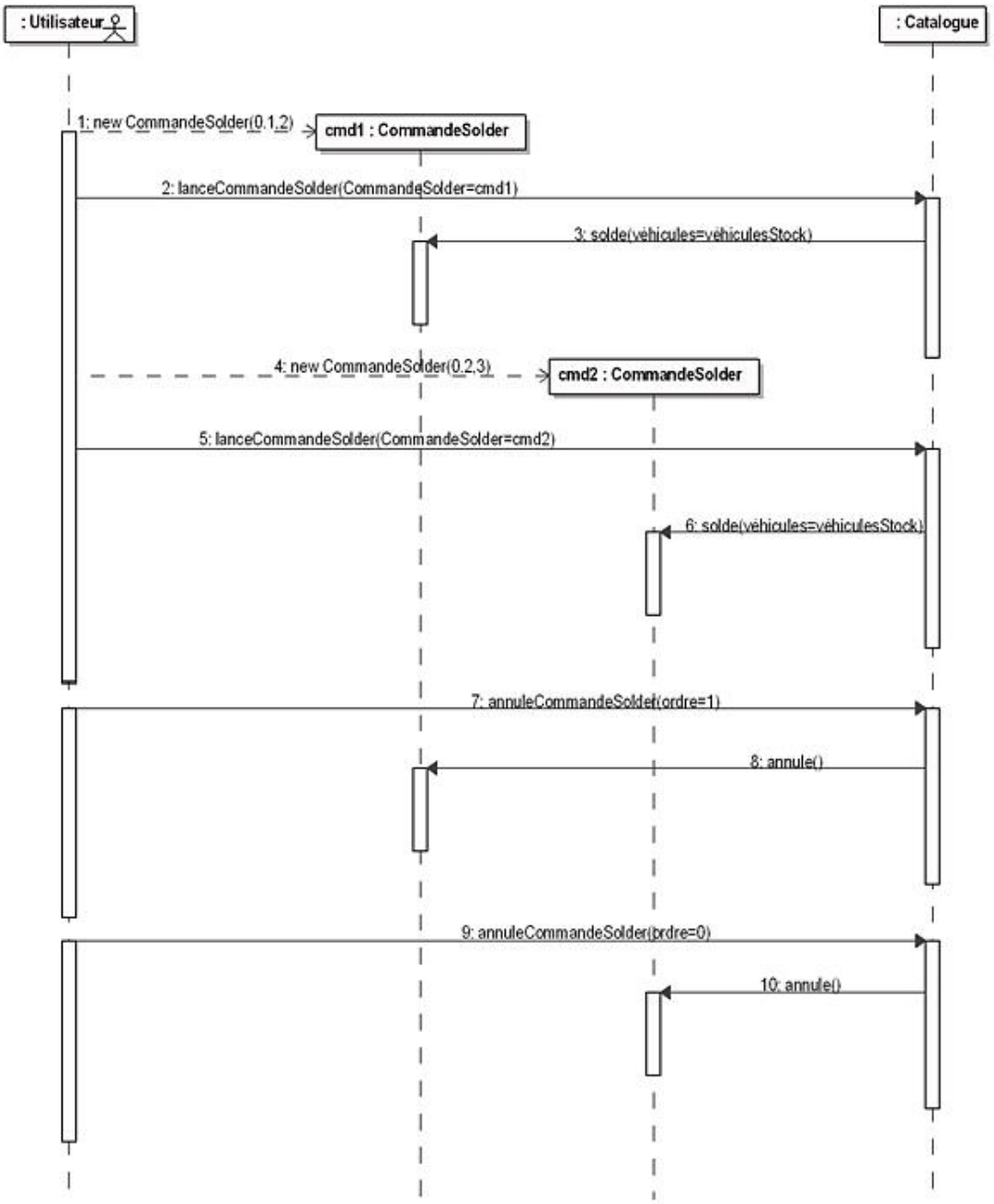


Figure 19.2 - Exemple de séquence des appels de méthodes du diagramme 19.1

Structure

1. Diagramme de classes

La figure 19.3 détaille la structure générique du pattern.

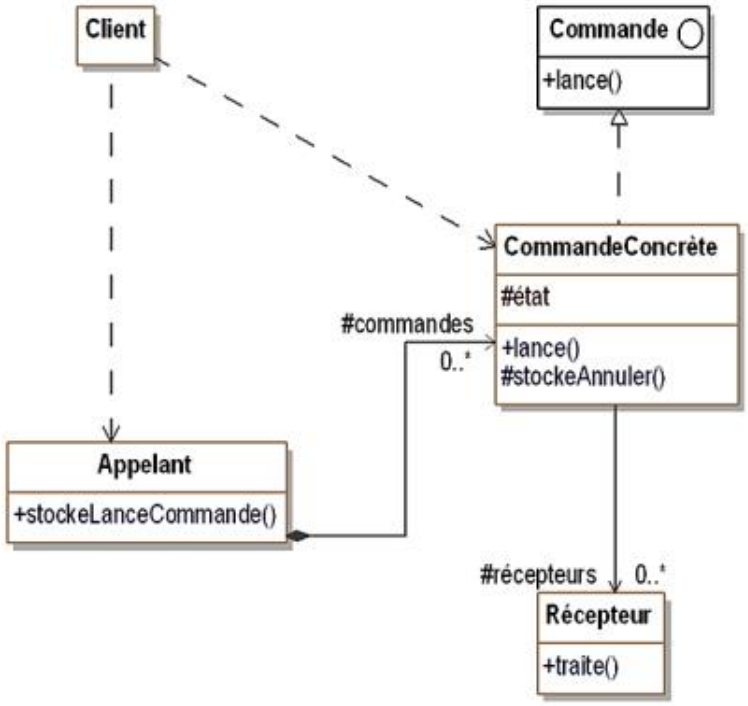


Figure 19.3 - Structure du pattern `Command`

2. Participants

Les participants au pattern sont les suivants :

- `Commande` est l'interface qui introduit la signature de la méthode `lance` qui exécute la commande ;
- `CommandeConcrète (CommandeSolder)` implante la méthode `lance`, gère l'association avec le ou les récepteurs et implante la méthode `stockeAnnuler` qui stocke l'état (ou les valeurs nécessaires) pour pouvoir annuler par la suite ;
- `Client (Utilisateur)`crée et initialise la commande et la transmet à l'appelant ;
- `Appelant (Catalogue)`stocke et lance la commande (méthode `stockeLanceCommande`) ainsi qu'éventuellement les requêtes d'annulation ;
- `Récepteur (Véhicule)`traite les actions nécessaires pour effectuer la commande ou pour l'annuler.

3. Collaborations

La figure 19.4 illustre les collaborations du pattern `Command` :

- le client crée une commande concrète en spécifiant le ou les récepteurs ;

- le client transmet cette commande à la méthode `stockeLanceCommande` de l'appelant qui commence par stocker la commande ;
- l'appelant lance ensuite la commande en invoquant la méthode `lance` ;
- l'état ou les données nécessaires à l'annulation sont stockés (méthode `stockeAnnuler`) ;
- la commande demande au ou aux récepteurs de réaliser les traitements.

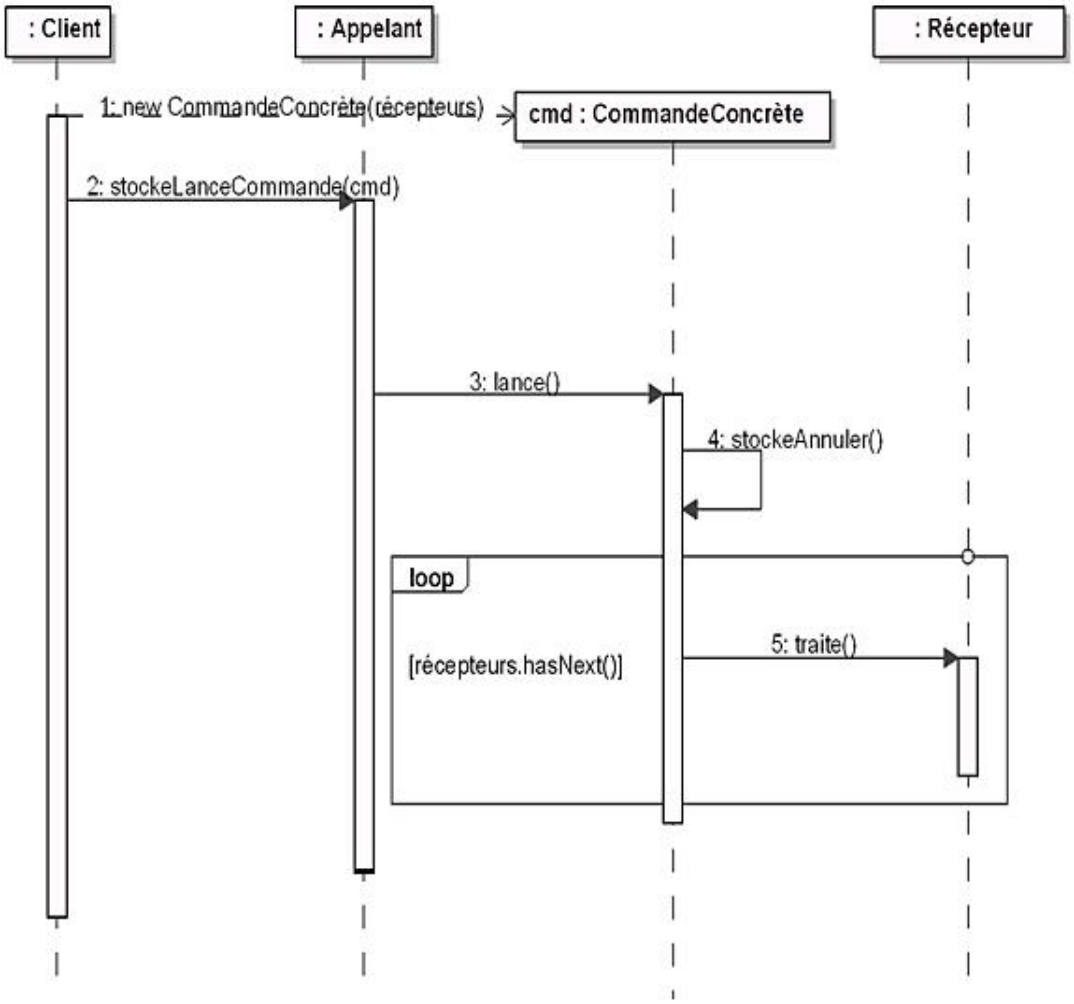


Figure 19.4 - Collaborations au sein du pattern `Command`

Domaines d’application

Le pattern est utilisé dans les cas suivants :

- un objet doit être paramétré par un traitement à réaliser. Dans le cas du pattern `Command`, c’est l’appelant qui est paramétré par une commande qui contient la description d’un traitement à réaliser sur un ou plusieurs récepteurs ;
- les commandes doivent être stockées dans une file et pouvoir être exécutées à un moment quelconque, éventuellement plusieurs fois ;
- les commandes sont annulables ;
- les commandes doivent être tracées dans un fichier de log ;
- les commandes doivent être regroupées sous la forme d’une transaction. Une transaction est un ensemble ordonné de commandes qui agissent sur l’état d’un système et qui peuvent être annulées.

Exemple en Java

Nous introduisons maintenant l'exemple en Java. La classe `Vehicule` s'écrit en Java comme suit. Chaque véhicule possède un nom, une date d'entrée dans le stock et un prix de vente. La méthode `modifiePrix` permet d'ajuster le prix avec un coefficient.

```
public class Vehicule
{
  protected String nom;
  protected long dateEntreeStock;
  protected double prixVente;

  public Vehicule(String nom, long dateEntreeStock,
    double prixVente)
  {
    this.nom = nom;
    this.dateEntreeStock = dateEntreeStock;
    this.prixVente = prixVente;
  }

  public long getDureeStockage(long aujourdhui)
  {
    return aujourdhui - dateEntreeStock;
  }

  public void modifiePrix(double coefficient)
  {
    this.prixVente = 0.01 * Math.round(coefficient *
      this.prixVente * 100);
  }

  public void affiche()
  {
    System.out.println(nom + " prix : " + prixVente +
      " date entrée Stock " + dateEntreeStock);
  }
}
```

La classe `CommandeSolder` possède les attributs suivants :

- `vehiculesSoldes` : la liste des véhicules soldés ;
- `aujourdhui` : la valeur d'aujourd'hui ;
- `dureeStock` : la durée de stockage que doit dépasser un véhicule pour être soldé ;
- `tauxRemise` : le pourcentage de remise à appliquer sur les véhicules soldés.

La méthode `solde` calcule d'abord les véhicules à solder, puis ensuite modifie leur prix. Quant à la méthode `annule`, elle rétablit le prix des véhicules soldés en utilisant l'inverse du taux de remise initial.

```
import java.util.*;
public class CommandeSolder
{
  protected List<Vehicule> vehiculesSoldes = new
    ArrayList<Vehicule>();
  protected long aujourdhui;
  protected long dureeStock;
  protected double tauxRemise;

  public CommandeSolder(long aujourdhui, long dureeStock,
    double tauxRemise)
  {
    this.aujourdhui = aujourdhui;
    this.dureeStock = dureeStock;
```

```
    this.tauxRemise = tauxRemise;
  }

  public void solde(List<Vehicule> vehicules)
  {
    vehiculesSoldes.clear();
    for (Vehicule vehicule: vehicules)
      if (vehicule.getDureeStockage(aujourdhui) >=
        dureeStock)
        vehiculesSoldes.add(vehicule);
    for (Vehicule vehicule: vehiculesSoldes)
      vehicule.modifiePrix(1.0 - tauxRemise);
  }

  public void annule()
  {
    for (Vehicule vehicule: vehiculesSoldes)
      vehicule.modifiePrix(1.0 / (1.0 - tauxRemise));
  }

  public void retablit()
  {
    for (Vehicule vehicule: vehiculesSoldes)
      vehicule.modifiePrix(1.0 - tauxRemise);
  }
}
```

La classe `Catalogue` a le code source Java qui suit.

Elle gère la liste de tous les véhicules (attribut `vehicules`) ainsi que la liste des commandes (attribut `commandes`). Chaque nouvelle commande est ajoutée au début de la liste des commandes comme l'indique la première ligne de la méthode `lanceCommandeSolder`.

```
import java.util.*;
public class Catalogue
{
  protected List<Vehicule> vehicules =
    new ArrayList<Vehicule>();
  protected List<CommandeSolder> commandes =
    new ArrayList<CommandeSolder>();

  public void lanceCommandeSolder(CommandeSolder commande)
  {
    commandes.add(0, commande);
    commande.solde(vehicules);
  }

  public void annuleCommandeSolder(int ordre)
  {
    commandes.get(ordre).annule();
  }

  public void retablitCommandeSolder(int ordre)
  {
    commandes.get(ordre).retablit();
  }

  public void ajoute(Vehicule vehicule)
  {
    vehicules.add(vehicule);
  }

  public void affiche()
  {
    for (Vehicule vehicule: vehicules)
      vehicule.affiche();
  }
}
```

Enfin, la classe `Utilisateur` décrit le programme principal. Il crée trois véhicules, deux commandes qui s'appliquent au premier et au troisième véhicule, la première appliquant une remise de 10%, la seconde une remise de 50%. La remise totale est de 55%, puis de 50% après l'annulation de la première remise, puis à nouveau 55% après que cette première remise soit rétablie.

```
public class Utilisateur
{
  public static void main(String[] args)
  {
    Vehicule vehicule1 = new Vehicule("A01", 1, 1000.0);
    Vehicule vehicule2 = new Vehicule("A11", 6, 2000.0);
    Vehicule vehicule3 = new Vehicule("Z03", 2, 3000.0);
    Catalogue catalogue = new Catalogue();
    catalogue.ajoute(vehicule1);
    catalogue.ajoute(vehicule2);
    catalogue.ajoute(vehicule3);
    System.out.println("Affichage du catalogue initial");
    catalogue.affiche();
    System.out.println();
    CommandeSolder commmandeSolder = new CommandeSolder
      (10, 5, 0.1);
    catalogue.lanceCommandeSolder(commmandeSolder);
    System.out.println("Affichage du catalogue après " +
      "exécution de la première commande");
    catalogue.affiche();
    System.out.println();
    CommandeSolder commmandeSolder2 = new CommandeSolder
      (10, 5, 0.5);
    catalogue.lanceCommandeSolder(commmandeSolder2);
    System.out.println("Affichage du catalogue après " +
      "exécution de la seconde commande");
    catalogue.affiche();
    System.out.println();
    catalogue.annuleCommandeSolder(1);
    System.out.println("Affichage du catalogue après " +
      "annulation de la première commande");
    catalogue.affiche();
    System.out.println();
    catalogue.retablitCommandeSolder(1);
    System.out.println("Affichage du catalogue après " +
      "rétablissement de la première commande");
    catalogue.affiche();
    System.out.println();
  }
}
```

L'exécution de ce programme donne le résultat suivant.

```
Affichage du catalogue initial
A01 prix : 1000.0 date entrée Stock 1
A11 prix : 2000.0 date entrée Stock 6
Z03 prix : 3000.0 date entrée Stock 2

Affichage du catalogue après exécution de la première
commande
A01 prix : 900.0 date entrée Stock 1
A11 prix : 2000.0 date entrée Stock 6
Z03 prix : 2700.0 date entrée Stock 2

Affichage du catalogue après exécution de la seconde
commande
A01 prix : 450.0 date entrée Stock 1
A11 prix : 2000.0 date entrée Stock 6
Z03 prix : 1350.0 date entrée Stock 2

Affichage du catalogue après annulation de la première
commande
A01 prix : 500.0 date entrée Stock 1
A11 prix : 2000.0 date entrée Stock 6
```

```
Z03 prix : 1500.0 date entrée Stock 2

Affichage du catalogue après rétablissement
de la première commande
A01 prix : 450.0 date entrée Stock 1
A11 prix : 2000.0 date entrée Stock 6
Z03 prix : 1350.0 date entrée Stock 2
```

Description

Le pattern `Interpreter` fournit un cadre pour donner une représentation par objets de la grammaire d'un langage afin d'évaluer, en les interprétant, des expressions écrites dans ce langage.

Exemple

Nous voulons créer un petit moteur de recherche des véhicules basé sur la recherche par mot-clé dans la description des véhicules à l'aide d'expressions booléennes selon la grammaire très simple suivante :

```
expression ::= terme || mot-clé || (expression)

terme ::= facteur 'ou' facteur

facteur ::= expression 'et' expression

mot-clé ::= 'a'..'z','A'..'Z' {'a'..'z','A'..'Z'}*
```

Les symboles entre apostrophes sont des symboles terminaux. Les symboles non terminaux sont `expression`, `terme`, `facteur` et `mot-clé`. Le symbole de départ est `expression`.

Nous mettons en œuvre le pattern `Interpreter` afin de pouvoir exprimer toute expression répondant à cette grammaire selon un arbre syntaxique constitué d'objets afin de pouvoir l'évaluer en l'interprétant.

Un tel arbre n'est constitué que de symboles terminaux. Pour simplifier, nous considérons qu'un mot-clé constitue un symbole terminal en tant que chaîne de caractères.

L'expression `(rouge ou gris) et récent et diesel` va être traduire par l'arbre syntaxique de la figure 20.1.

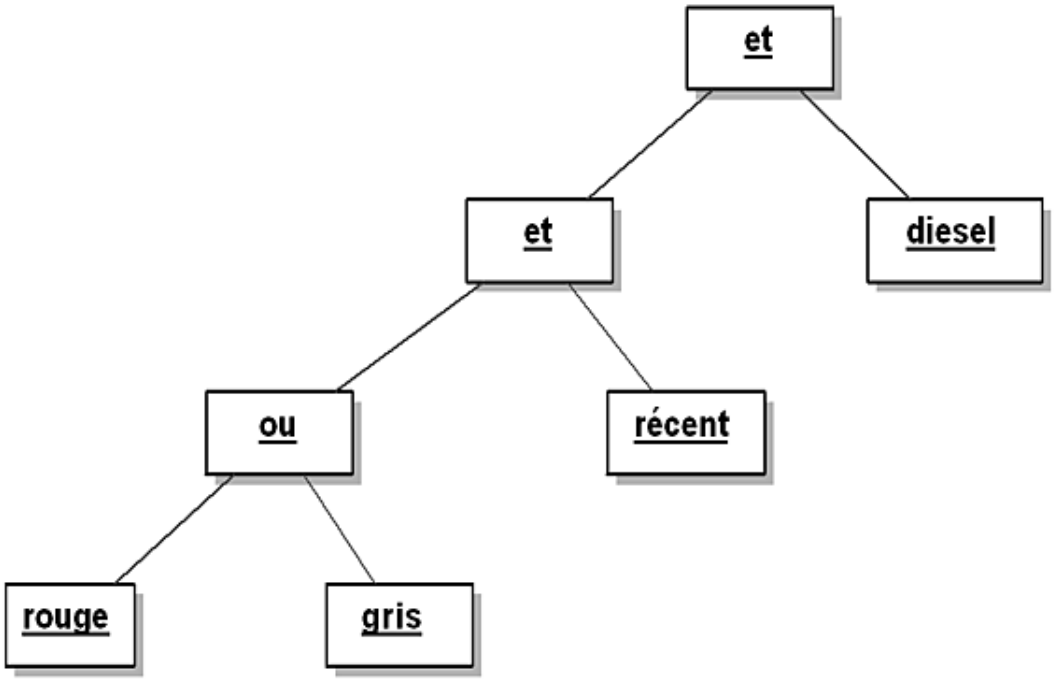


Figure 20.1 - Arbre syntaxique correspondant à l'expression `(rouge ou gris) et récent et diesel`

L'évaluation d'un tel arbre pour la description d'un véhicule se fait en commençant par le sommet. Quand un nœud est un opérateur, l'évaluation se fait en calculant récursivement la valeur de chaque sous-arbre (celui de gauche puis celui de droite) et en appliquant l'opérateur. Quand un nœud est un mot-clé, l'évaluation se fait en recherchant la chaîne correspondante dans la description du véhicule.

Le moteur de recherche consiste donc à évaluer l'expression pour chaque description et à renvoyer la liste des véhicules pour lesquels l'évaluation est vraie.

Cette technique de recherche n'est pas optimisée, elle n'est donc valable que pour une petite quantité de véhicules.

Le diagramme des classes permettant de décrire des arbres syntaxiques comme celui de la figure 20.1 est représenté à la figure 20.2. La méthode `évalue` permet d'évaluer l'expression pour une description d'un véhicule fournie en paramètre.

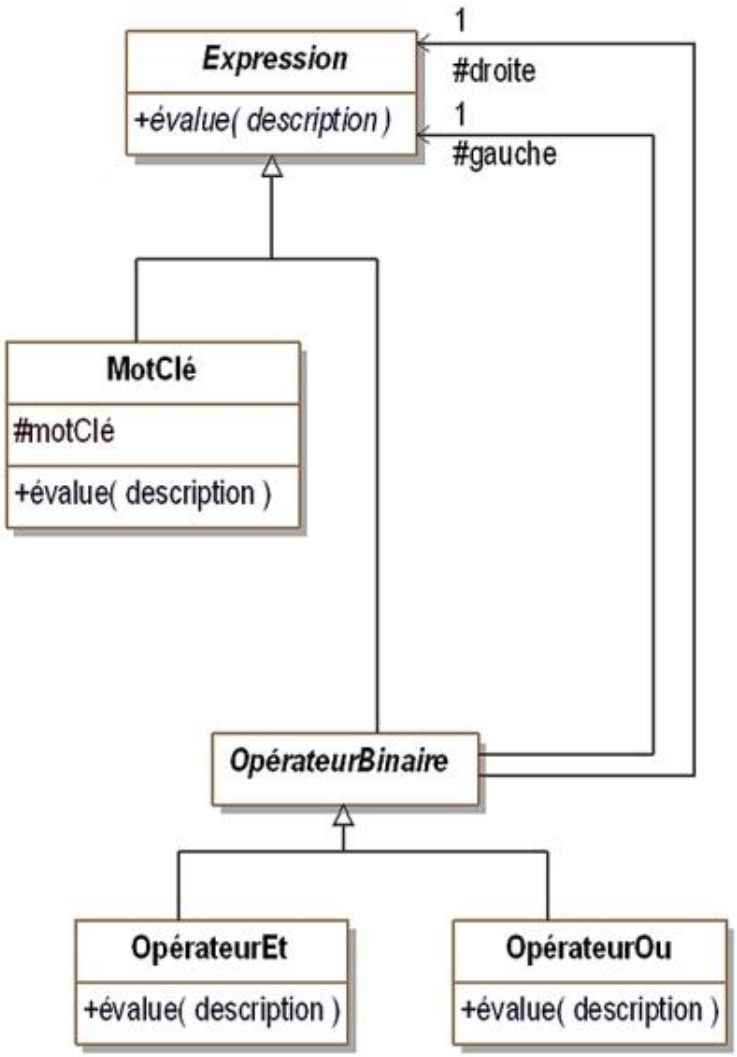


Figure 20.2 - Le pattern `Interpreter` pour représenter des arbres syntaxiques et les évaluer

Structure

1. Diagramme de classes

La figure 20.3 détaille la structure générique du pattern.

Ce diagramme de classes montre qu'il existe deux types de sous-expression, à savoir :

- les éléments terminaux qui peuvent être des noms de variable, des entiers, des nombres réels ;
- les opérateurs qui peuvent être binaires comme dans l'exemple, unaires (opérateur « - ») ou prenant plus d'argument comme des fonctions.

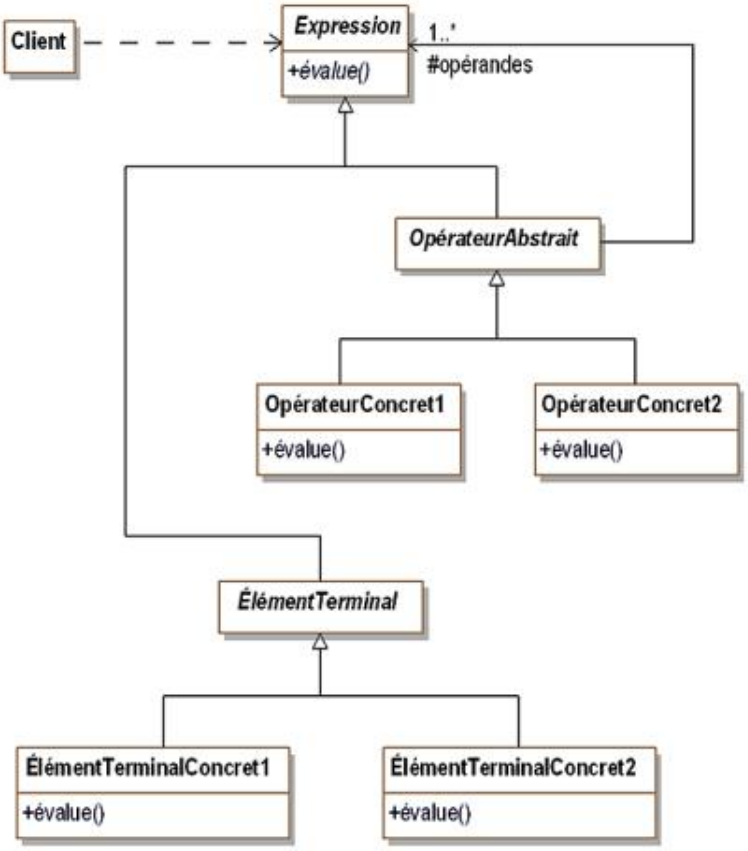


Figure 20.3 - Structure du pattern `Interpreter`

2. Participants

Les participants au pattern sont les suivants :

- `Expression` est une classe abstraite représentant tout type d'expression, c'est-à-dire tout nœud de l'arbre syntaxique ;
- `OpérateurAbstrait (OpérateurBinaire)` est également une classe abstraite. Elle décrit tout nœud de type opérateur, c'est-à-dire possédant des opérandes qui sont des sous-arbres de l'arbre syntaxique ;
- `OpérateurConcret1` et `OpérateurConcret2 (OpérateurEt, OpérateurOu)`sont des implantations d'`OpérateurAbstrait` décrivant totalement la sémantique de l'opérateur et donc capables de l'évaluer ;

- `ÉlémentTerminal` est une classe abstraite décrivant tout nœud correspondant à un élément terminal ;
- `ÉlémentTerminalConcret1` et `ÉlémentTerminalConcret2` (`MotClé`) sont des classes concrètes correspondant à un élément terminal, capables d'évaluer cet élément.

3. Collaborations

Le client construit une expression sous la forme d'un arbre syntaxique dont les nœuds sont des instances des sous-classes d'`Expression`. Il demande ensuite à l'instance qui est au sommet de l'arbre de procéder à l'évaluation :

- si cette instance est un élément terminal, l'évaluation est directe ;
- si cette instance est un opérateur, il y a lieu de procéder d'abord à l'évaluation des opérandes. Cette évaluation est faite récursivement, chaque opérande étant le sommet d'une expression.

Domaines d’application

Le pattern est utilisé pour interpréter des expressions représentées sous la forme d’arbres syntaxiques. Il s’applique principalement dans les cas suivants :

- la grammaire des expressions est simple ;
- l’évaluation n’a pas besoin d’être rapide.

Si la grammaire est complexe, il vaut mieux se tourner vers des analyseurs syntaxiques spécialisés. Si l’évaluation doit être effectuée rapidement, l’utilisation d’un compilateur peut s’avérer nécessaire.

Exemple en Java

Nous fournissons le code complet d'un exemple en Java qui non seulement permet l'évaluation d'un arbre syntaxique mais le construit également.

La construction de l'arbre syntaxique, appelée analyse syntaxique, est également répartie dans les classes, à savoir celles de la figure 20.2 sous la forme de méthodes de classe (méthodes préfixées du mot-clé `static` en Java).

Le code source de la classe `Expression` est donné à la suite. La partie relative à l'évaluation se limite à la déclaration de la signature de la méthode `évalue`.

Les méthodes `prochainJeton`, `analyse`, et `parse` sont dédiées à l'analyse syntaxique. La méthode `analyse` est utilisée pour parser une expression entière alors que `parse` est dédiée à l'analyse soit d'un mot-clé soit d'une expression mise entre parenthèses.

```
public abstract class Expression
{
  public abstract boolean evalue(String description);

  // partie analyse syntaxique
  static protected String source;
  static protected int index;
  static protected String jeton;

  static protected void prochainJeton()
  {
    while ((index < source.length()) && (source.charAt
      (index) == ' '))
      index++;
    if (index == source.length())
      jeton = null;
    else if ((source.charAt(index) == '(') ||
      (source.charAt(index) == ')'))
    {
      jeton = source.substring(index, index + 1);
      index++;
    }
    else
    {
      int debut = index;
      while ((index < source.length()) && (source.charAt
        (index) != ' ') && (source.charAt(index) != ')'))
        index++;
      jeton = source.substring(debut, index);
    }
  }

  static public Expression analyse(String source)throws
    Exception
  {
    Expression.source = source;
    index = 0;
    prochainJeton();
    return OperateurOu.parse();
  }

  static public Expression parse()throws Exception
  {
    Expression resultat;
    if (jeton.equals("("))
    {
      prochainJeton();
      resultat = OperateurOu.parse();
      if (jeton == null)
        throw new Exception("Erreur de syntaxe");
      if (!jeton.equals(")"))
        throw new Exception("Erreur de syntaxe");
      prochainJeton();
    }
```

```
    else
      resultat = MotCle.parse();
    return resultat;
  }
}
```

Nous donnons maintenant le source code des sous-classes d'`Expression`. Tout d'abord la classe concrète `MotCle` dont la méthode `evalue` recherche le mot-clé dans la description. Cette classe gère également l'analyse syntaxique d'un mot-clé.

```
public class MotCle extends Expression
{
  protected String motCle;

  public MotCle(String motCle)
  {
    this.motCle = motCle;
  }

  public boolean evalue(String description)
  {
    return description.indexOf(motCle) !=  - 1;
  }

  // partie analyse syntaxique
  static public Expression parse()throws Exception
  {
    Expression resultat;
    resultat = new MotCle(jeton);
    prochainJeton();
    return resultat;
  }
}
```

La classe abstraite `OperateurBinaire` gère les liens vers les deux opérandes de l'opérateur.

```
public abstract class OperateurBinaire extends Expression
{
  protected Expression operandeGauche, operandeDroite;

  public OperateurBinaire(Expression operandeGauche,
    Expression operandeDroite)
  {
    this.operandeGauche = operandeGauche;
    this.operandeDroite = operandeDroite;
  }
}
```

La classe concrète `OperateurOu` implante la méthode `evalue` et gère l'analyse syntaxique d'un terme.

```
public class OperateurOu extends OperateurBinaire
{
  public OperateurOu(Expression operandeGauche,
    Expression operandeDroite)
  {
    super(operandeGauche, operandeDroite);
  }

  public boolean evalue(String description)
  {
    return operandeGauche.evalue(description) ||
      operandeDroite.evalue(description);
  }

  // partie analyse syntaxique
  static public Expression parse()throws Exception
  {
    Expression resultatGauche, resultatDroit;
    resultatGauche = OperateurEt.parse();
```

```
    while ((jeton != null) && (jeton.equals("ou")))
    {
      prochainJeton();
      resultatDroit = OperateurEt.parse();
      resultatGauche = new OperateurOu(resultatGauche,
        resultatDroit);
    }
    return resultatGauche;
  }
}
```

La classe concrète `OperateurEt` implante la méthode `evalue` et gère l'analyse syntaxique d'un facteur.

```
public class OperateurEt extends OperateurBinaire
{
  public OperateurEt(Expression operandeGauche,
    Expression operandeDroite)
  {
    super(operandeGauche, operandeDroite);
  }

  public boolean evalue(String description)
  {
    return operandeGauche.evalue(description) &&
      operandeDroite.evalue(description);
  }

  // partie analyse syntaxique
  static public Expression parse()throws Exception
  {
    Expression resultatGauche, resultatDroit;
    resultatGauche = Expression.parse();
    while ((jeton != null) && (jeton.equals("et")))
    {
      prochainJeton();
      resultatDroit = Expression.parse();
      resultatGauche = new OperateurEt(resultatGauche,
        resultatDroit);
    }
    return resultatGauche;
  }
}
```

Enfin, la classe `Utilisateur` implante le programme principal.

```
import java.util.*;
public class Utilisateur
{
  public static void main(String[] args)
  {
    Expression expressionRequete = null;
    Scanner reader = new Scanner(System.in);
    System.out.print("Entrez votre requête : ");
    String requete = reader.nextLine();
    try
    {
      expressionRequete = Expression.analyse(requete);
    }
    catch (Exception e)
    {
      System.out.println(e.getMessage());
      expressionRequete = null;
    }
    if (expressionRequete != null)
    {
      System.out.println(
        "Entrez le texte de description d'un véhicule : ");
      String description = reader.nextLine();
      if (expressionRequete.evalue(description))
```

```
        System.out.println(
          "La description répond à la requête");
      else
        System.out.println(
          "La description ne répond pas à la requête");
    }
  }
}
```

Un exemple d’exécution du programme se trouve ci-dessous.

```
Entrez votre requête : (rouge ou gris) et récent et diesel
Entrez le texte de description d’un véhicule :
Ce véhicule rouge fonctionnant au diesel est récent
La description répond à la requête
```

Description

Le pattern `Iterator` fournit un accès séquentiel à une collection d'objets à des clients sans que ceux-ci doivent se préoccuper de l'implantation de cette collection.

Exemple

Nous voulons donner un accès séquentiel aux véhicules composant le catalogue. Pour cela, nous pouvons implanter dans la classe du catalogue les méthodes suivantes :

- `début` : initialise le parcours du catalogue ;
- `item` : renvoie le véhicule courant ;
- `suivant` : passe au véhicule suivant.

Cette technique présente deux inconvénients :

- elle fait grossir inutilement la classe du catalogue ;
- elle ne permet qu'un seul parcours à la fois, ce qui peut être insuffisant (notamment dans le cas d'applications multitâches).

Le pattern `Iterator` propose une solution à ce problème. L'idée est de créer une classe `Itérateur` dont chaque instance peut gérer un parcours dans une collection. Les instances de cette classe `Itérateur` sont créées par la classe de collection qui se charge de les initialiser.

Le but du pattern `Iterator` est de fournir une solution qui puisse être paramétrée par le type des éléments des collections. Nous introduisons donc deux classes abstraites génériques :

- `Itérateur` est une classe abstraite générique qui introduit les méthodes `début`, `item` et `suivant` ;
- `Catalogue` est également une classe abstraite générique qui introduit la méthode qui crée, initialise et retourne une instance de `Itérateur`.

Il est ensuite possible de créer les sous-classes concrètes de ces deux classes abstraites génériques, sous-classes qui lient notamment les paramètres de généricité aux types utilisés dans l'application.

La figure 21.1 montre l'utilisation du pattern `Iterator` pour parcourir les véhicules du catalogue qui répondent à une requête.

Ce diagramme de classes utilise des paramètres génériques qui sont contraints (`TÉlément` est un sous-type de `Élément` et `TItérateur` est un sous-type de `Itérateur<TÉlément>`). Les deux classes `Catalogue` et `Itérateur` possèdent une association avec un ensemble d'éléments, l'ensemble des éléments référencés par `Itérateur` étant un sous-ensemble de ceux référencés par `Catalogue`.

Les sous-classes `CatalogueVéhicule` et `ItérateurVéhicule` héritent par une relation qui fixe les types des paramètres de généricité de leurs surclasses respectives.

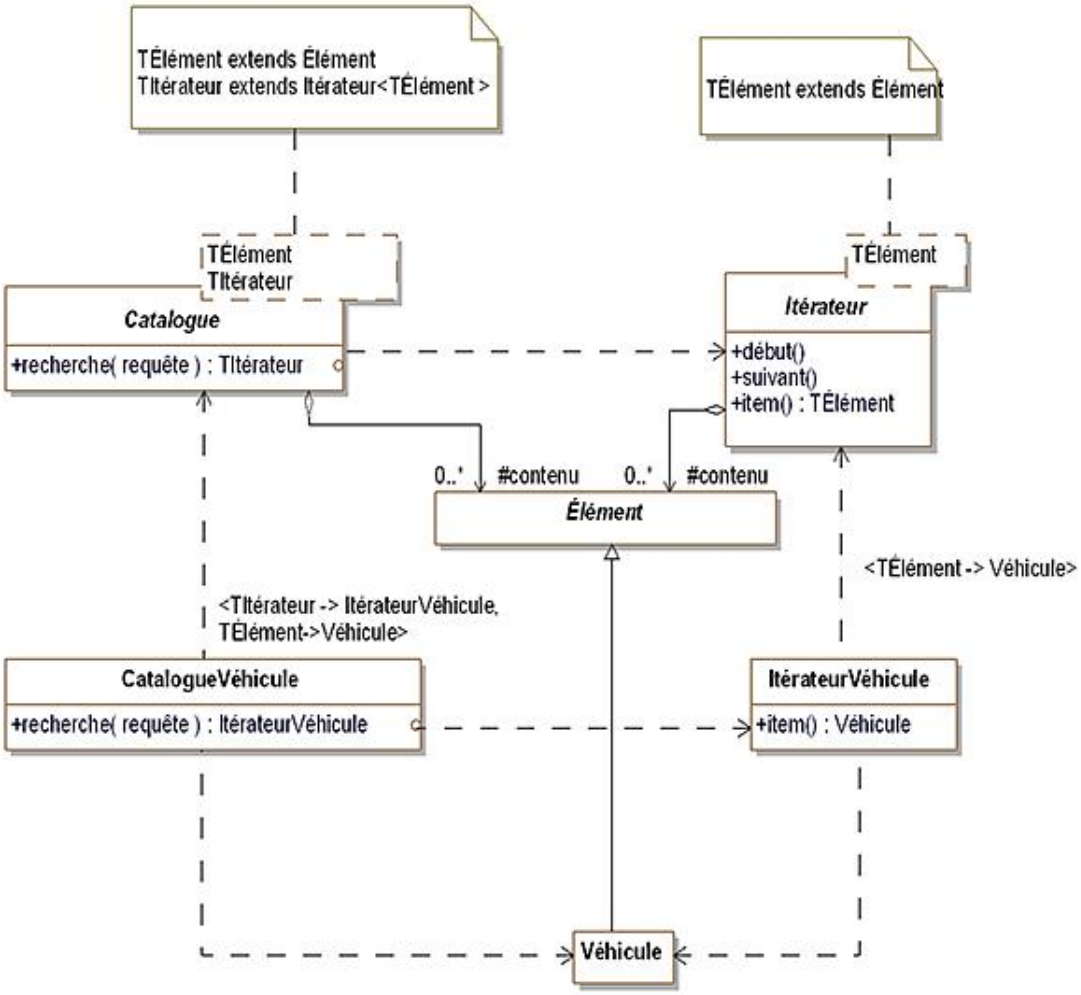


Figure 21.1 - Le pattern `Iterator` pour accéder séquentiellement à des catalogues de véhicules

Structure

1. Diagramme de classes

La figure 21.2 détaille la structure générique du pattern, qui est très proche du diagramme des classes de la figure 21.1.

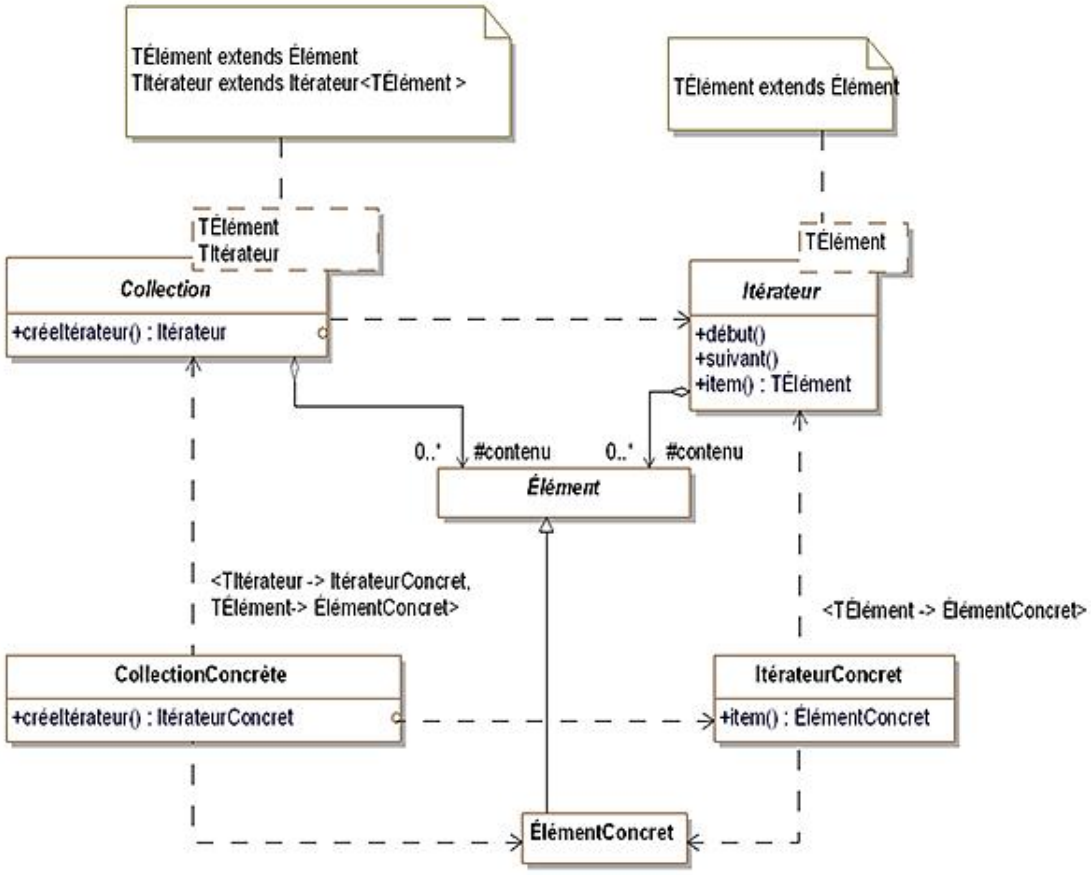


Figure 21.2 - Structure du pattern `Iterator`

2. Participants

Les participants au pattern sont les suivants :

- `Itérateur`est la classe abstraite qui implante l'association de l'itérateur avec les éléments de la collection ainsi que les méthodes. Elle est générique et paramétrée par le type `TÉlément` ;
- `ItérateurConcret` `(ItérateurVéhicule)`est une sous-classe concrète de `Itérateur` qui lie `TÉlément` à `ÉlémentConcret` ;
- `Collection` `(Catalogue)`est la classe abstraite qui implante l'association de la collection avec les éléments et la méthode `créeItérateur` ;
- `CollectionConcrète` `(CatalogueVéhicule)`est une sous-classe concrète de `Collection` qui lie `TÉlément` à `ÉlémentConcret` et `TItérateur` à `ItérateurConcret` ;
- `Élément`est la classe abstraite des éléments de la collection ;

- `ÉlémentConcret (Véhicule)`est une sous-classe concrète de `Élément` utilisée par `ItérateurConcret` et `CollectionConcrète`.

3. Collaborations

L'itérateur garde en mémoire l'objet courant dans la collection. Il est capable de calculer l'objet suivant du parcours.

Domaines d’application

Le pattern est utilisé dans les cas suivants :

- un parcours d’accès au contenu d’une collection doit être réalisé sans accéder à la représentation interne de cette collection ;
- il doit être possible de gérer plusieurs parcours simultanément.

Exemple en Java

Nous présentons l'exemple en Java du parcours du catalogue de véhicules à l'aide d'un itérateur.

Le code source de classe abstraite `Element` se trouve à la suite. Les éléments possèdent une description. La méthode `motCleValide` vérifie l'appartenance d'un mot-clé à la description.

```
public abstract class Element
{
  protected String description;

  public Element(String description)
  {
    this.description = description;
  }

  public boolean motCleValide(String motCle)
  {
    return description.indexOf(motCle) !=  - 1;
  }
}
```

La sous-classe concrète `Vehicule` introduit une méthode `affiche`.

```
public class Vehicule extends Element
{
  public Vehicule(String description)
  {
    super(description);
  }

  public void affiche()
  {
    System.out.println("Description du véhicule : " +
      description);
  }
}
```

La classe `Iterateur` introduit les méthodes `debut`, `suivant`, `item` ainsi que la méthode `setMotCleRequete` qui initialise l'itérateur.

```
import java.util.List;
public abstract class Iterateur
  <TElement extends Element>
{
  protected String motCleRequete;
  protected int index;
  protected List<TElement> contenu;

  public void setMotCleRequete(String motCleRequete,
     List<TElement> contenu)
  {
    this.motCleRequete = motCleRequete;
    this.contenu = contenu;
  }

  public void debut()
  {
    index = 0;
    int taille = contenu.size();
    while ((index < taille) && (!contenu.get(index)
      .motCleValide(motCleRequete)))
      index++;
  }

  public void suivant()
  {
```

```
    int taille = contenu.size();
    index++;
    while ((index < taille) && (!contenu.get(index)
      .motCleValide(motCleRequete)))
      index++;
  }

  public TElement item()
  {
    if (index < contenu.size())
      return contenu.get(index);
    else
      return null;
  }
}
```

La sous-classe `IterateurVehicule` se contente de lier `TElement` à `Vehicule`.

```
public class IterateurVehicule extends
  Iterateur<Vehicule>
{
}
```

La classe `Catalogue` gère l'attribut `contenu` qui est la collection des éléments et introduit la méthode `recherche` qui crée, initialise et retourne l'itérateur.

La méthode `creeIterateur` est abstraite. En effet, il n'est pas possible de créer une instance avec un type qui est un paramètre de généricité. Son implantation doit être réalisée dans une sous-classe qui lie le paramètre à une classe concrète.

```
import java.util.*;
public abstract class Catalogue
  <TElement extends Element,
  TIterateur extends Iterateur<TElement>>
{
  protected List<TElement> contenu =
    new ArrayList<TElement>();

  protected abstract TIterateur creeIterateur();

  public TIterateur recherche(String motCleRequete)
  {
    TIterateur resultat = creeIterateur();
    resultat.setMotCleRequete(motCleRequete, contenu);
    return resultat;
  }
}
```

La sous-classe concrète `CatalogueVehicule` lie `TElement` à `Vehicule` et `TIterateur` à `IterateurVehicule`.

Elle introduit deux éléments :

- un constructeur qui construit la collection des véhicules (dans un véritable applicatif, cette collection viendrait d'une base de données) ;
- l'implantation de la méthode `creeIterateur`.

```
public class CatalogueVehicule extends
  Catalogue<Vehicule, IterateurVehicule>
{
  public CatalogueVehicule()
  {
    contenu.add(new Vehicule("véhicule bon marché"));
    contenu.add(new Vehicule("petit véhicule bon marché"));
    contenu.add(new Vehicule("véhicule grande qualité"));
  }

  protected IterateurVehicule creeIterateur()
```

```
  {
    return new IterateurVehicule();
  }
}
```

Enfin, la classe `Utilisateur` introduit le programme principal qui crée le catalogue des véhicules et un itérateur basé sur la recherche du mot-clé « bon marché ». Ensuite, le programme principal affiche la liste des véhicules retournés par l'itérateur.

```
public class Utilisateur
{
  public static void main(String[] args)
  {
    CatalogueVehicule catalogue = new CatalogueVehicule();
    IterateurVehicule iterateur = catalogue.recherche(
      "bon marché");
    Vehicule vehicule;
    iterateur.debut();
    vehicule = iterateur.item();
    while (vehicule != null)
    {
      vehicule.affiche();
      iterateur.suivant();
      vehicule = iterateur.item();
    }
  }
}
```

L'exécution de ce programme produit le résultat suivant.

```
Description du véhicule : véhicule bon marché
Description du véhicule : petit véhicule bon marché
```

Description

Le pattern `Mediator` a pour but de construire un objet dont la vocation est la gestion et le contrôle des interactions dans un ensemble d'objets sans que ses éléments doivent se connaître mutuellement.

Exemple

La conception par objets favorise la distribution du comportement entre les objets du système. Cependant, à l'extrême, cette distribution peut conduire à un très grand nombre de liaisons obligeant quasiment chaque objet à connaître tous les autres objets du système. Une conception avec une telle quantité de liaisons peut s'avérer être de mauvaise qualité. En effet, la modularité et les possibilités de réutilisation des objets sont alors réduites. Chaque objet ne peut pas travailler sans les autres et le système devient monolithique, perdant toute modularité. De surcroît pour adapter et modifier le comportement d'une petite partie du système, il devient nécessaire de définir de nombreuses sous-classes.

Les interfaces utilisateur dynamiques sont un bon exemple d'un tel système. Une modification de la valeur d'un contrôle graphique peut conduire à modifier l'aspect d'autres contrôles graphiques comme, par exemple :

- devenir visible ou masqué ;
- modifier le nombre de valeurs possibles (pour un menu) ;
- changer le format des valeurs à saisir.

La première possibilité est donc de lier chaque contrôle aux contrôles dont l'aspect change en fonction de sa valeur. Cette possibilité présente les inconvénients cités ci-dessus.

L'autre possibilité est de mettre en œuvre le pattern `Mediator`. Celle-ci consiste à construire un objet central chargé de la coordination des contrôles graphiques. Lorsque la valeur d'un contrôle est modifiée, il prévient l'objet médiateur qui se charge d'invoquer les méthodes adéquates des autres contrôles graphiques afin qu'ils puissent réaliser les modifications nécessaires.

Dans notre système de vente en ligne de véhicules, un emprunt peut être demandé pour acquérir un véhicule en remplissant un formulaire en ligne. Il est possible d'emprunter seul ou avec un co-emprunteur. Ce choix se fait à l'aide d'un menu. Si le choix est d'emprunter avec un co-emprunteur, tout un ensemble de contrôles graphiques relatifs aux données du co-emprunteur doivent apparaître et être gérés. Ces mêmes contrôles doivent à nouveau disparaître si le choix porte à nouveau sur un emprunt sans co-emprunteur.

La figure 22.1 illustre le diagramme des classes correspondant. Ce diagramme introduit les classes suivantes :

- `Contrôle` est une classe abstraite qui introduit les éléments communs à tous les contrôles graphiques ;
- `PopupMenu`, `ZoneSaisie` et `Bouton` sont les sous-classes concrètes de `Contrôle` qui implantent les méthodes `dessine` et `clic` ;
- `Formulaire` est la classe qui fait office de médiateur. Elle reçoit les notifications de changement des contrôles par invocation de la méthode `contrôleModifié`.

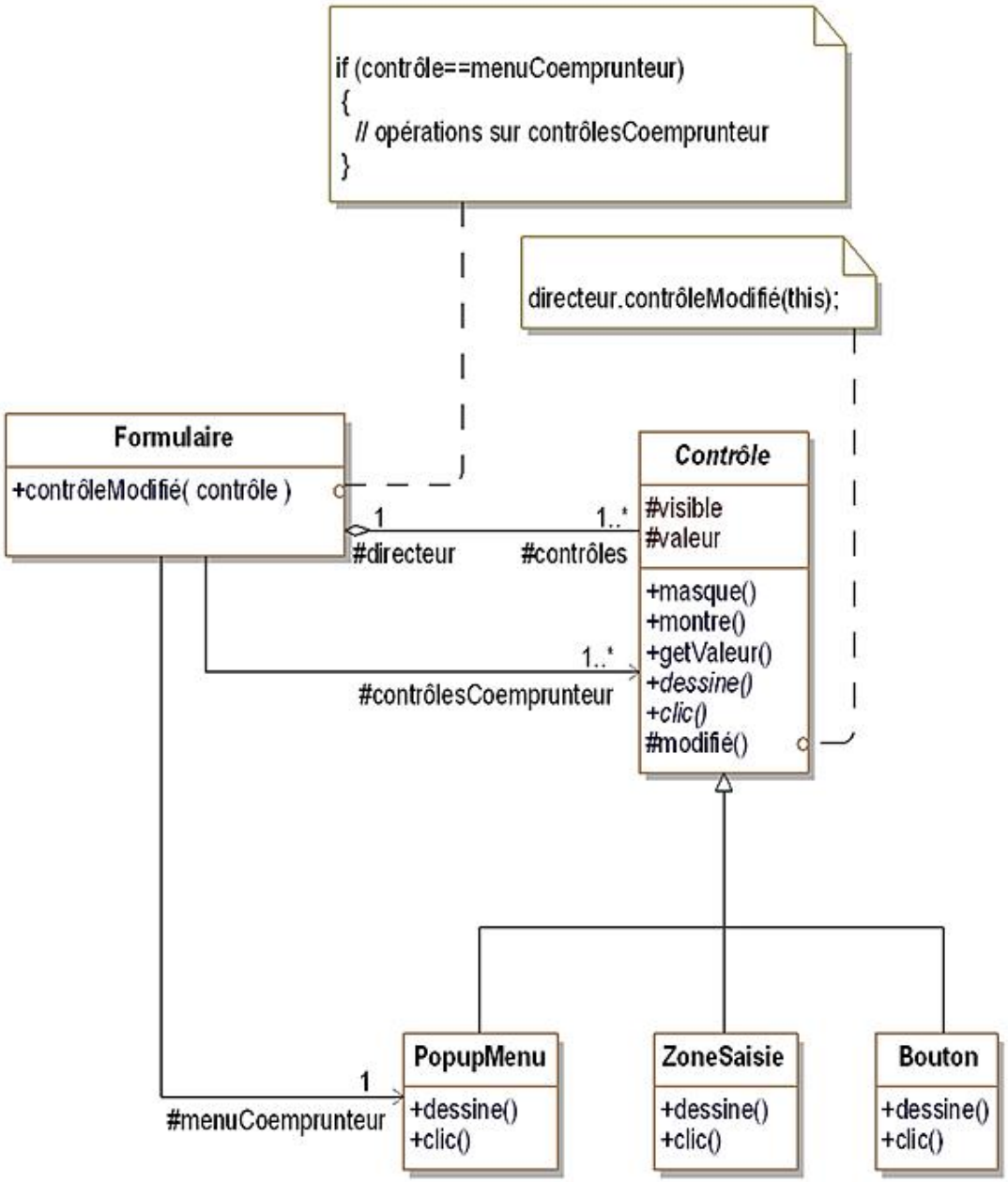


Figure 22.1 - Le pattern `Mediator` pour gérer un formulaire de demande d'emprunt

Chaque fois que la valeur d'un contrôle graphique est modifiée, la méthode `modifié` du contrôle est invoquée. Cette méthode héritée de la classe abstraite `Contrôle` invoque à son tour la méthode `contrôleModifié` de `Formulaire` (le médiateur). Celle-ci invoque, à son tour, les méthodes des contrôles du formulaire pour réaliser les actions nécessaires.

La figure 22.2 illustre ce fonctionnement de façon partielle sur l'exemple. Quand la valeur du contrôle `menuCoemprunteur` change, la zone de saisie du nom du co-emprunteur est respectivement affichée ou masquée selon que la valeur du contrôle `menuCoemprunteur` vaut "avec" ou "sans".

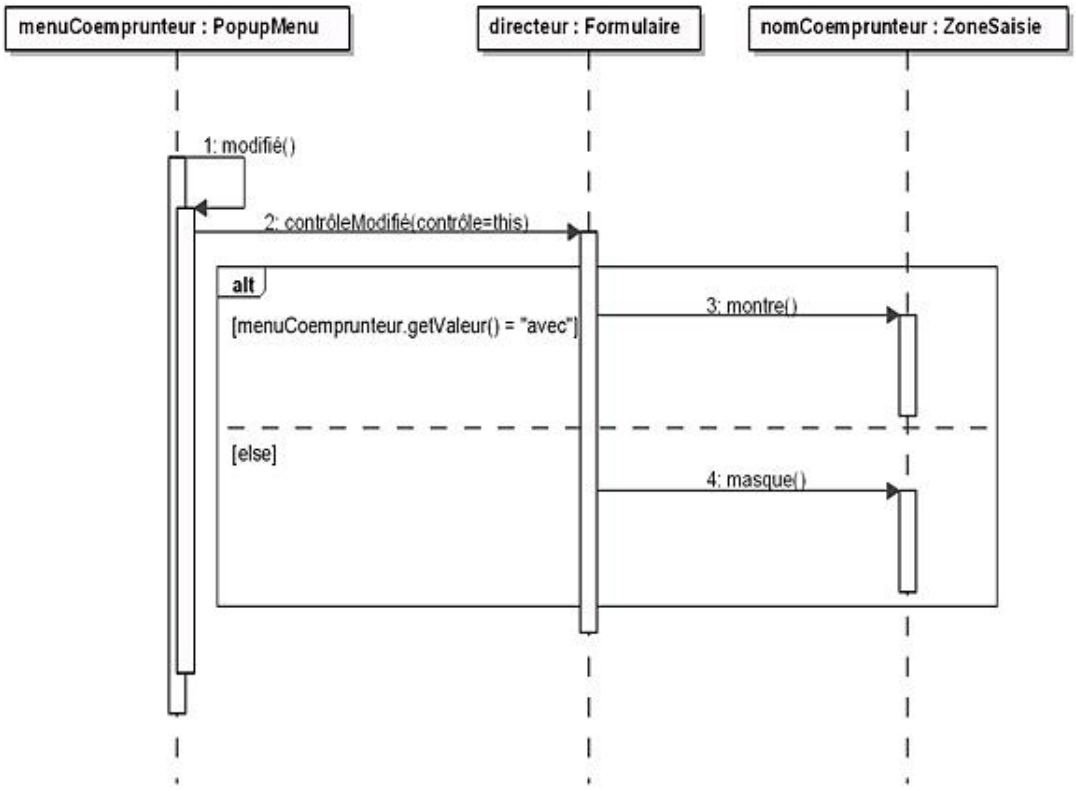


Figure 22.2 - Exemple de séquence d'utilisation du pattern `Mediator`

Structure

1. Diagramme de classes

La figure 22.3 détaille la structure générique du pattern.

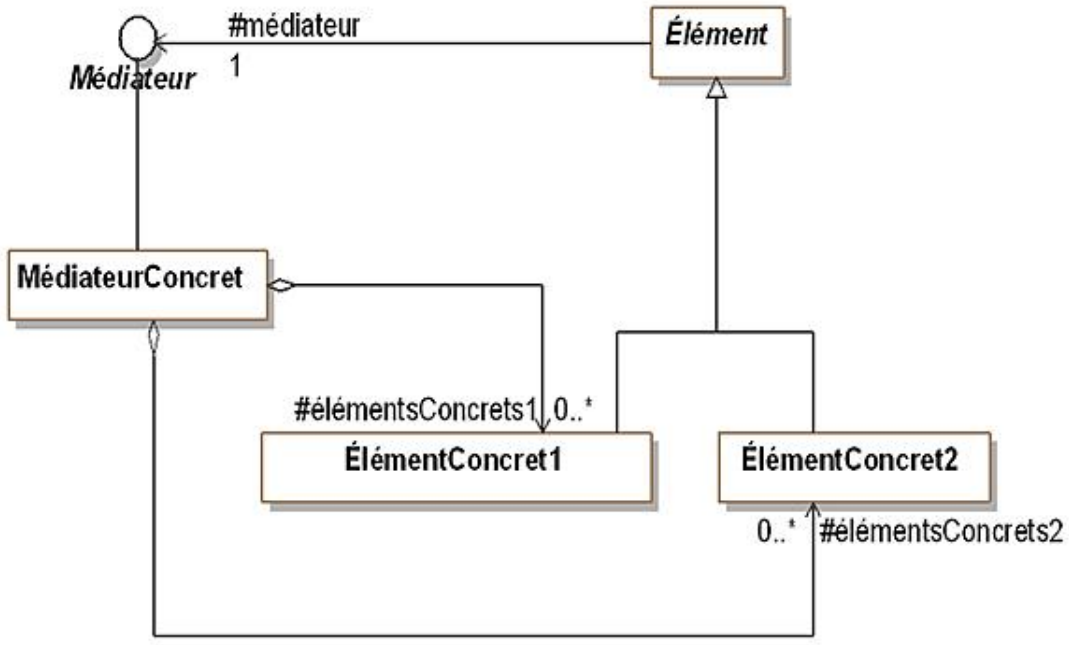


Figure 22.3 - Structure du pattern `Mediator`

2. Participants

Les participants au pattern sont les suivants :

- `Médiateur` définit l'interface du médiateur pour les objets `Élément` ;
- `MédiateurConcret (Formulaire)` implante la coordination entre les éléments et gère les associations avec les éléments ;
- `Élément (Contrôle)` est la classe abstraite des éléments qui introduit leurs attributs, associations et méthodes communes ;
- `ÉlémentConcret1`et `ÉlémentConcret2 (PopupMenu, ZoneSaisie et Bouton)` sont les classes concrètes des éléments qui communiquent avec le médiateur au lieu de communiquer avec les autres éléments.

3. Collaborations

Les éléments envoient des messages au médiateur et en reçoivent. Le médiateur implante la collaboration et la coordination entre les éléments.

Domaines d’application

Le pattern est utilisé dans les cas suivants :

- un système est formé d’un ensemble d’objets basé sur une communication complexe conduisant à associer de nombreux objets entre eux ;
- les objets d’un système sont difficiles à réutiliser car ils possèdent de nombreuses associations avec d’autres objets ;
- la modularité d’un système est médiocre, obligeant dans le cas d’une adaptation d’une partie du système à écrire de nombreuses sous-classes.