Formation UML– Design Patterns

El Hadji Gaye

Auteur	El Hadji Gaye
Pour	Formations
Date	15/01/2024
Objet	UML Design Patterns.

I) Vocabulaire

- UML : c'est l'acronyme anglais pour « Unified Modeling Language ». On le traduit par « Langage de modélisation unifié ».

- OMG : Object Management Group.

II) Introduction

UML est basé sur l'approche par objets. Celle-ci vit le jour bien avant UML dans le domaine des langages de programmation. Simula, le tout premier langage à objets, est né dans les années 1960. Ce langage connut beaucoup de successeurs comme Smalltalk, C++, Java ou encore récemment C#.

Avec un langage de programmation, la description des objets est réalisée de façon formelle selon une syntaxe rigoureuse. Cette syntaxe présente l'inconvénient de ne pas être lisible par des non-programmeurs et d'être assez lente à déchiffrer même pour des programmeurs. Les humains, à la différence des machines, préfèrent les langages graphiques pour représenter des abstractions car ils peuvent ainsi les maîtriser plus facilement et plus rapidement, et obtenir une vue d'ensemble d'un système en un temps beaucoup plus court.

III) Historique de l'UML

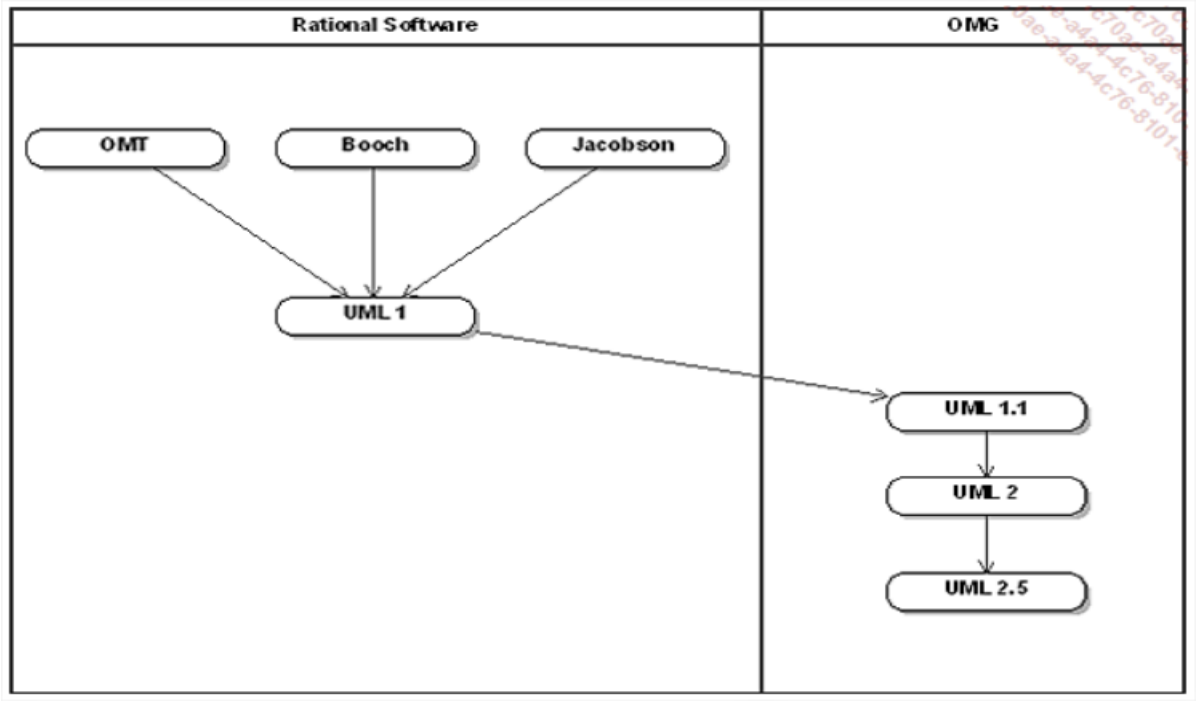
Dans les années 1980 et au début des années 1990, les notations graphiques se multiplièrent, chacune utilisant bien souvent sa propre notation. En 1994, Jim Rumbaugh et Grady Booch décidèrent de se regrouper pour unifier leurs notations. Celles-ci provenaient de leurs méthodes : OMT pour Jim Rumbaugh et la méthode Booch pour Grady Booch. En 1995, Yvar Jacobson décida de rejoindre l'équipe dite des Trois Amigos. Cette équipe travailla alors au sein de Rational Software.

La **version 1.0** d'UML fut publiée en 1997. Le travail d'évolution de la notation était alors devenu trop important pour trois personnes. Les Trois Amigos demandèrent ainsi l'aide de l'Object Management Group (OMG), un consortium de 800 sociétés et universités travaillant dans le domaine des technologies de l'objet. La notation UML fut adoptée par l'OMG en novembre 1997 dans sa version 1.1. L'OMG créa en son sein une Task Force chargée de l'évolution d'UML.

Cette Task Force a mis à jour UML plusieurs fois. En mars 2003, la version 1.5 ajouta la possibilité de décrire des actions grâce à une extension d'UML appelée Action Semantics, ou sémantique des actions.

La **version 2.0** fut publiée en juillet 2005. Elle constitue la première évolution majeure depuis la sortie d'UML en 1997. De nouveaux diagrammes ont été ajoutés et les diagrammes existants ont été enrichis de nouvelles constructions. Depuis juillet 2005, cette version 2.0 a été améliorée. En février 2009, la **version 2.2** a été publiée incluant la taxinomie officielle des profils UML. La **version 2.5** est publiée en juin 2015 et apporte principalement la possibilité de présenter les attributs et méthodes hérités d'une classe. La dernière version est la **version 2.5.1** publiée en 2017.

À l'aide d'un diagramme d'activités ci-dessous illustre l'évolution d'UML et en retrace la genèse ainsi que les principales versions.



IV) Les designs patterns

Un Design Pattern est une solution à un problème récurrent dans la conception d'applications orientées objet. Un patron de conception décrit alors la solution éprouvée pour résoudre ce problème d'architecture de logiciel. L'utilisation des Design Patterns offre de nombreux avantages. Tout d'abord, cela permet de répondre à un problème de conception grâce à une solution éprouvée et validée par des experts. Ainsi, on gagne en rapidité et en qualité de conception ce qui diminue également les coûts.
De plus, les Design Patterns sont réutilisables et permettent de mettre en avant les bonnes pratiques de conception.

1. Formalisme

Les Design Patterns sont représentés par :

- **Nom** : qui permet de l'identifier clairement
- **Problématique** : description du problème auquel il répond
- **Solution** : description de la solution souvent accompagnée d'un schéma UML
- **Conséquences** : les avantages et les inconvénients de cette solution

2. *Familles de patterns (GoF, Grasp)*

a. Les patterns GoF

Le présent ouvrage a été conçu comme une présentation simple et efficace

L'acronyme GoF veut dire « Gang of Four ». Ces patterns très célèbres ont été conçus par 4 informaticiens : Erich Gamma, Richard Helm, Ralph Johnson et John Vlissides, Ils proposent eux aussi des solutions élégantes, et toujours différentes pour résoudre différents problèmes récurrents rencontrés par les architectes logiciels.

Ces vingt-trois modèles de conception ci-dessous furent introduits en 1995 dans le livre « **Design Patterns - Elements of Reusable Object Oriented Software** » du Gang of Four (la bande des quatre auteurs).

Il existe trois familles de patrons de conception selon leur utilisation :

- **Création** : ils permettent d'instancier et de configurer des classes et des objets.
- **Structure** : ils permettent d'organiser les classes d'une application.
- **Comportement** : ils permettent d'organiser les objets pour qu'ils collaborent entre eux.

Les vingt-trois patrons GoF sont :

Patterns de construction d'objets

Singleton

Ce patron vise à assurer qu'il n'y a toujours qu'une seule instance d'une classe en fournissant une interface pour la manipuler. C'est un des patrons les plus simples. L'objet qui ne doit exister qu'en une seule instance comporte une méthode pour obtenir cette unique instance et un mécanisme pour empêcher la création d'autres instances.

Fabrique et Fabrique abstraite

Ce patron fournit une interface pour créer des familles d'objets sans spécifier la classe concrète. Le patron fabrique, ou méthode fabrique (en anglais factory ou factory method) est un patron récurrent. Une fabrique simple retourne une instance d'une classe parmi plusieurs possibles, en fonction des paramètres qui ont été fournis. Toutes les classes ont un lien de parenté, et des méthodes communes, et chacune est optimisée en fonction d'une certaine donnée. Le patron fabrique abstraite (en anglais abstract factory) va un pas plus loin que la fabrique simple. Une fabrique abstraite est utilisée pour obtenir un jeu d'objets connexes. Par exemple pour implémenter une charte graphique : il existe une fabrique qui retourne des objets (boutons, menus) dans le style de Windows, une qui retourne des objets dans le style de Motif, et une dans le style de Macintosh. Une fabrique abstraite est obtenue en utilisant une fabrique simple.

Builder (Monteur)

Ce patron sépare le processus de construction d'un objet du résultat obtenu. Permet d'utiliser le même processus pour obtenir différents résultats. C'est une alternative au pattern fabrique. Au lieu d'une méthode pour créer un objet, à laquelle est passée un ensemble de paramètres, la classe fabrique comporte une méthode pour créer un objet - le monteur (en anglais builder). Cet objet comporte des propriétés qui peuvent être modifiées et une méthode pour créer l'objet final en tenant compte de toutes les propriétés. Ce pattern est particulièrement utile quand il y a de nombreux paramètres de création, presque tous optionnels.

Prototype

Ce patron permet de définir le genre d'objet à créer en dupliquant une instance qui sert d'exemple. L'objectif de ce patron est d'économiser le temps nécessaire pour instancier des objets. Selon ce patron, une application comporte une instance d'un objet, qui sert de prototype. Cet objet comporte une méthode clone pour créer des duplicata. Des langages de programmation comme PHP ont une méthode clone incorporée dans tous les objets.

Patterns de structuration

Adaptateur

Ce patron convertit l'interface d'une classe en une autre interface exploitée par une application. Permet d'interconnecter des classes qui sans cela seraient incompatibles. Il est utilisé dans le cas où un programme se sert d'une bibliothèque de classe qui ne correspond plus à l'utilisation qui en est faite, à la suite d'une mise à jour de la bibliothèque dont l'interface a changé. Un objet adaptateur (en anglais adapter) expose alors l'ancienne interface en utilisant les fonctionnalités de la nouvelle.

Bridge (Pont)

Ce patron permet de découpler une abstraction de son implémentation, de telle manière qu'ils peuvent évoluer indépendamment. Il consiste à diviser une implémentation en deux parties : une classe d'abstraction qui définit le problème à résoudre, et une seconde classe qui fournit une implémentation. Il peut exister plusieurs implémentations pour le même problème et la classe d'abstraction comporte une référence à l'implémentation choisie, qui peut être changée selon les besoins. Le patron pont (en anglais bridge) est fréquemment utilisé pour réaliser des récepteurs d'événements.

Composite

Le patron composite (même nom en anglais) permet de composer une hiérarchie d'objets, et de manipuler de la même manière un élément unique, une branche, ou l'ensemble de l'arbre. Il permet en particulier de créer des objets complexes en reliant différents objets selon une structure en arbre. Ce patron impose que les différents objets aient une même interface, ce qui rend uniformes les manipulations de la structure. Par exemple dans un traitement de texte, les mots sont placés dans des paragraphes disposés dans des colonnes dans des pages ; pour manipuler l'ensemble, une classe composite implémente une interface. Cette interface est héritée par les objets qui représentent les textes, les paragraphes, les colonnes et les pages.

Décorateur

Le patron décorateur (en anglais decorator) permet d'attacher dynamiquement des responsabilités à un objet. Une alternative à l'héritage. Ce patron est inspiré des poupées russes. Un objet peut être caché à l'intérieur d'un autre objet décorateur qui lui rajoutera des fonctionnalités, l'ensemble peut être décoré avec un autre objet qui lui ajoute des fonctionnalités et ainsi de suite. Cette technique nécessite que l'objet décoré et ses décorateurs implémentent la même interface, qui est typiquement définie par une classe abstraite.

Façade

Le patron façade (en anglais facade) fournit une interface unifiée sur un ensemble d'interfaces d'un système. Il est utilisé pour réaliser des interfaces de programmation. Si un sous-système comporte plusieurs composants qui doivent être utilisée dans un ordre précis, une classe façade sera mise à disposition, et permettra de contrôler l'ordre des opérations et de cacher les détails techniques des sous-systèmes.

Flyweight

Dans le patron flyweight (en français poids-mouche), un type d'objet est utilisé pour représenter une gamme de petits objets tous différents. Ce patron permet de créer un ensemble d'objets et de les réutiliser. Il peut être utilisé par exemple pour représenter un jeu de caractères : Un objet factory va retourner un objet correspondant au caractère recherché. La même instance peut être retournée à chaque fois que le caractère est utilisé dans un texte.

Proxy

Ce patron est un substitut d'un objet, qui permet de contrôler l'utilisation de ce dernier. Un proxy est un objet destiné à protéger un autre objet. Le proxy a la même interface que l'objet à protéger. Un proxy peut être créé par exemple pour permettre d'accéder à distance à un objet (via un middleware). Le proxy peut également être créé dans le but de retarder la création de l'objet protégé - qui sera créé immédiatement avant d'être utilisé. Dans sa forme la plus simple, un proxy ne protège rien du tout et transmet tous les appels de méthode à l'objet cible.

Patterns comportementaux

Chaîne de responsabilité

Le patron Chaîne de responsabilité (en anglais chain of responsibility) vise à découpler l'émission d'une requête de la réception et le traitement de cette dernière en permettant à plusieurs objets de la traiter successivement. Dans ce patron chaque objet comporte un lien vers l'objet suivant, qui est du même type. Plusieurs objets sont ainsi attachés et forment une chaîne. Lorsqu'une demande est faite au premier objet de la chaîne, celui-ci tente de la traiter, et s'il ne peut pas il fait appel à l'objet suivant, et ainsi de suite16.

Commande

Ce patron emboîte une demande dans un objet, permettant de paramétrer, mettre en file d'attente, journaliser et annuler des demandes. Dans ce patron un objet commande (en anglais command) correspond à une opération à effectuer. L'interface de cet objet comporte une méthode execute. Pour chaque opération, l'application va créer un objet différent qui implémente cette interface - qui comporte une méthode execute. L'opération est lancée lorsque la méthode execute est utilisée. Ce patron est notamment utilisé pour les barres d'outils.

Interpreter

Le patron comporte deux composants centraux : le contexte et l'expression ainsi que des objets qui sont des représentations d'éléments de grammaire d'un langage de programmation11. Le patron est utilisé pour transformer une expression écrite dans un certain langage de programmation - un texte source - en quelque chose de manipulable par programmation : Le code source est écrit conformément à une ou plusieurs règles de grammaire, et un objet est créé pour chaque utilisation d'une règle de grammaire. L'objet interpreter est responsable de transformer le texte source en objets.

Iterator

Ce patron permet d'accéder séquentiellement aux éléments d'un ensemble sans connaitre les détails techniques du fonctionnement de l'ensemble. C'est un des patrons les plus simples et les plus fréquents. Selon la spécification originale, il consiste en une interface qui fournit les méthodes Next et Current. L'interface en Java comporte généralement une méthode nextElement et une méthode hasMoreElements.

Mediator

Dans ce patron il y a un objet qui définit comment plusieurs objets communiquent entre eux en évitant à chacun de faire référence à ses interlocuteurs. Ce patron est utilisé quand il y a un nombre non négligeable de composants et de relations entre les composants. Par exemple dans un réseau de 5 composants il peut y avoir jusqu'à 20 relations (chaque composant vers 4 autres). Un composant médiateur est placé au milieu du réseau et le nombre de relations est diminué : chaque composant est relié uniquement au médiateur. Le mediator joue un rôle similaire à un sujet dans le patron observer et sert d'intermédiaire pour assurer les communications entre les objets.

Memento

Ce patron vise à externaliser l'état interne d'un objet sans perte d'encapsulation. Permet de remettre l'objet dans l'état où il était auparavant. Ce patron permet de stocker l'état interne d'un objet sans que cet état ne soit rendu public par une interface. Il est composé de trois classes : l´origine - d'où l'état provient, le memento - l'état de l'objet d'origine, et le gardien qui est l'objet qui manipulera le memento. L´origine comporte une méthode pour manipuler les memento. Le gardien est responsable de stocker les memento et de les renvoyer à leur origine. Ce patron ne définit pas d'interface précise pour les différents objets, qui sont cependant toujours au nombre de trois.

Observer

Ce patron établit une relation un à plusieurs entre des objets, où lorsqu'un objet change, plusieurs autres objets sont avisés du changement. Dans ce patron, un objet le sujet tient une liste des objets dépendants des observateurs qui seront avertis des modifications apportées au sujet. Quand une modification est apportée, le sujet émet un message aux différents observateurs. Le message peut contenir une description détaillée du changement. Dans ce patron, un objet observer comporte une méthode pour inscrire des observateurs. Chaque observateur comporte une méthode Notify. Lorsqu'un message est émis, l'objet appelle la méthode Notify de chaque observateur inscrit.

State

Ce patron permet à un objet de modifier son comportement lorsque son état interne change. Ce patron est souvent utilisé pour implémenter une machine à états. Un exemple d'appareil à états est le lecteur audio - dont les états sont lecture, enregistrement, pause et arrêt. Selon ce patron il existe une classe machine à états, et une classe pour chaque état. Lorsqu'un événement provoque le changement d'état, la classe machine à états se relie à un autre état et modifie ainsi son comportement.

Strategy

Dans ce patron, une famille d'algorithmes est encapsulée de manière qu'ils soient interchangeables. Les algorithmes peuvent changer indépendamment de l'application qui s'en sert. Il comporte trois rôles : le contexte, la stratégie et les implémentations. La stratégie est l'interface commune aux différentes implémentations - typiquement une classe abstraite. Le contexte est l'objet qui va associer un algorithme avec un processus.

Template method

Ce patron définit la structure générale d'un algorithme en déléguant certains passages. Permettant à des sous-classes de modifier l'algorithme en conservant sa structure générale. C'est un des patrons les plus simples et les plus couramment utilisés en programmation orientée objet. Il est utilisé lorsqu'il y a plusieurs implémentations possibles d'un calcul. Une classe d'exemple (anglais template) comporte des méthodes d'exemple, qui, utilisées ensemble, implémentent un algorithme par défaut. Certaines méthodes peuvent être vides ou abstraites. Les sous-classes de la classe template peuvent remplacer certaines méthodes et ainsi créer un algorithme dérivé.

Visitor

Ce patron représente une opération à effectuer sur un ensemble d'objets. Permet de modifier l'opération sans changer l'objet concerné ni la structure. Selon ce patron, les objets à modifier sont passés en paramètre à une classe tierce qui effectuera des modifications. Une classe abstraite Visitor définit l'interface de la classe tierce. Ce patron est utilisé notamment pour manipuler un jeu d'objets, où les objets peuvent avoir différentes interfaces, qui ne peuvent pas être modifiés.

b. Les patterns Grasp

L'acronyme GRASP signifie « General responsibility assignment software patterns », en français, « patrons généraux d'affectation des responsabilités », sont un ensemble de principes généraux qui peuvent être utilisés dans toute conception d'objet. On parle de conception pilotée par les responsabilités. Il existe neuf patterns GRASP qui sont décrit ci-dessous :

- **Expert en information**

Problème : Quel est les principes généraux d'affection des responsabilités aux objets ?
Solution : Il faut affecter une responsabilité à l'expert en information, c'est-à-dire la classe qui possède les informations nécessaires pour s'acquitter de la responsabilité.

- **Créateur**

Problème : Déterminer quel objet doit créer une instance de classe donnée.
Solution : Il faut affecter à la classe B la responsabilité de créer la classe A, si au moins une des conditions suivantes est vraie (plus il y en a mieux c'est)

B contient des objets A
B agrège des objets A
B utilise étroitement des objets A
B possède des données nécessaires à l'instanciation des objets A

- **Faible couplage**

Problème : Comment réduire l'impact des modifications ?
Solution : Il faut affecter les responsabilités de sorte à éviter les couplages inutiles

- **Forte cohésion**

Problème : Comment s'assurer que les objets restent compréhensibles et cohérents, tout en contribuant au faible couplage ?
Solution : Il faut affecter les responsabilités de manière à ce que la cohésion reste élevée.

- **Contrôleur**

Problème : Quel est le premier objet qui reçoit, envoie, les données à la couche présentation et coordonne les opérations systèmes ?

Solution : Le pattern Contrôleur indique qu'il faut affecter la responsabilité à un objet représentant un des choix suivant :

Un contrôleur de cas d'utilisation qui gère tous les scénarios d'un cas d'utilisation.

Un contrôleur de façade (couramment appelé Front Controller) commun à tous les cas d'utilisation.

Un contrôleur d'acteur qui gère tous les scénarios d'un acteur donné.

- **Polymorphisme**

Problème : Comment gérer des opérations dépendantes des types d'objets ?
Solution : Il faut simplement créer une nouvelle classe étendant la classe de base par un nouveau comportement.

- **Indirection**

Problème : A quel objet affecter une responsabilité pour éviter le couplage entre plusieurs objets ?
Solution : Il faut affecter la responsabilité à un objet qui sert d'intermédiaire entre plusieurs objets (cet objet peut être une fabrique pure).

- **Fabrication pure**

Problème : Quel est l'objet responsable lorsqu'on veut respecter les principes de faible couplage et de forte cohésion mais que les solutions offertes par le pattern expert (par exemple) ne sont pas appropriées.

Solution : Il faut affecter des responsabilités fortement cohésives à une classe artificielle qui ne représente pas un concept du domaine.

- **Protection des variations**

Problème : Comment faire une bonne conception pour que les modifications n'aient pas d'impact indésirable ?
Solution : Il faut identifier les points de modifications (les évolutions hypothétiques) et affecter les responsabilités sur les objets autours d'eux. Pour cela, nous avons vu précédemment qu'il est possible d'utiliser le polymorphisme et les indirections. Mais il est également possible d'utiliser le principe d'encapsulation de données et des interfaces.

c. Autres patrons de conception

- **Object Pool**

Ce patron permet d'économiser les temps d'instanciation et de suppression lorsque de nombreux objets ont une courte durée d'utilisation. Il consiste à administrer une collection d'objets qui peuvent être recyclés. Une méthode du Pool délivre un objet soit par une nouvelle instanciation, soit par recyclage d'un objet périmé. Lorsque les objets arrivent à la fin de leur cycle de vie, ils sont remis à la disposition du Pool pour un futur recyclage. Dans la phase d'instanciation, le Pool peut instancier plus d'un objet à la fois si l'algorithme d'instanciation a une complexité meilleure que O(n). Le patron Object Pool est particulièrement utile lorsque le nombre total de cycles de vie est très grand devant le nombre d'instances à un moment précis et que les opérations d'instanciation et/ou suppression sont coûteuses en temps d'exécution par rapport à leur recyclage.

- **Modèle-vue-contrôleur**

Ce patron d'architecture logicielle signifie Modèle Vue Contrôleur. Ce patron de conception est combinaison des patrons observateur, stratégie et composite.

Ce patron peut être découpé en trois parties :

Le modèle (modèle de données) : le modèle représente le cœur (algorithmique) de l'application : traitements des données, interactions avec la base de données, etc. Il décrit les données manipulées par l'application. Il regroupe la gestion de ces données et est responsable de leur intégrité. La base de données sera l'un de ses composants. Le modèle comporte des méthodes standards pour mettre à jour ces données (insertion, suppression, changement de valeur).

La vue (présentation, interface utilisateur) : ce avec quoi l'utilisateur interagit se nomme précisément la vue. Sa première tâche est de présenter les résultats renvoyés par le modèle. Sa seconde tâche est de recevoir toute action de l'utilisateur (hover, clic de souris, sélection d'un bouton radio, cochage d'une case, entrée de texte, de mouvements, de voix, etc.). Ces différents événements sont envoyés au contrôleur. La vue n'effectue pas de traitement, elle se contente d'afficher les résultats des traitements effectués par le modèle et d'interagir avec l'utilisateur.

Le contrôleur (logique de contrôle, gestion des événements, synchronisation) : le contrôleur prend en charge la gestion des événements de synchronisation pour mettre à jour la vue ou le modèle et les synchroniser. Il reçoit tous les événements de la vue et enclenche les actions à effectuer. Si une action nécessite un changement des données, le contrôleur demande la modification des données au modèle afin que les données affichées se mettent à jour. D'après le patron de conception observateur/observable, la vue est un « observateur » du modèle qui est lui « observable ».

- **Inversion de contrôle**

L’inversion de contrôle (inversion of control, IoC) est un patron d'architecture commun à tous les frameworks (ou cadre de développement et d'exécution). Il fonctionne selon le principe que le flot d'exécution d'un logiciel n'est plus sous le contrôle direct de l'application elle-même mais du framework ou de la couche logicielle sous-jacente.

L’inversion de contrôle est un terme générique. Selon le problème, il existe différentes formes, ou représentation d'IoC, le plus connu étant l'injection de dépendances (dependency injection) qui est un patron de conception permettant, en programmation orientée objet, de découpler les dépendances entre objets.

- **Injection de dépendances**

L'injection de dépendances (dependency injection en anglais) est un mécanisme qui permet d'implémenter le principe de l'inversion de contrôle.

Il consiste à créer dynamiquement (injecter) les dépendances entre les différents objets en s'appuyant sur une description (fichier de configuration ou métadonnées) ou de manière programmatique. Ainsi les dépendances entre composants logiciels ne sont plus exprimées dans le code de manière statique mais déterminées dynamiquement à l'exécution.

3. *Implémentation des différentes Design Pattern GoF*

a. Singleton

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.

Plus de détails :

Ce patron vise à assurer qu'il n'y a toujours qu'une seule instance d'une classe en fournissant une interface pour la manipuler. C'est un des patrons les plus simples. L'objet qui ne doit exister qu'en une seule instance comporte une méthode pour obtenir cette unique instance et un mécanisme pour empêcher la création d'autres instances.

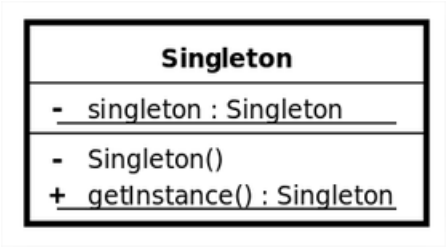
Exemple d'implémentation :

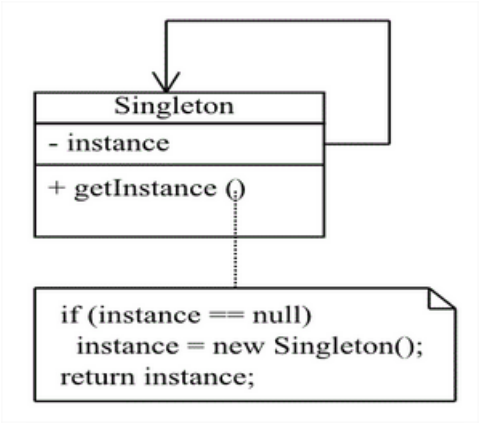
Considérons un site de commerce avec un certain nombre de vendeurs.

- Créer la classe **Vendeur** avec les attributs nom, email, adresse et nbVentes.
- Créer la classe **MySingleton** qui sera un Singleton qui aura pour but de stocker le meilleur de tous les vendeurs d'un site de commerce. Il aura comme attribut **bestVender** et **instance** (static).
- Faites évoluer la classe **MySingleton** en stockant en plus un tableau de tous les vendeurs du site dans un attribut **arrayVendeurs**.

Structure

Diagramme de classes





Domaines 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.

b. Factory Method

Description :

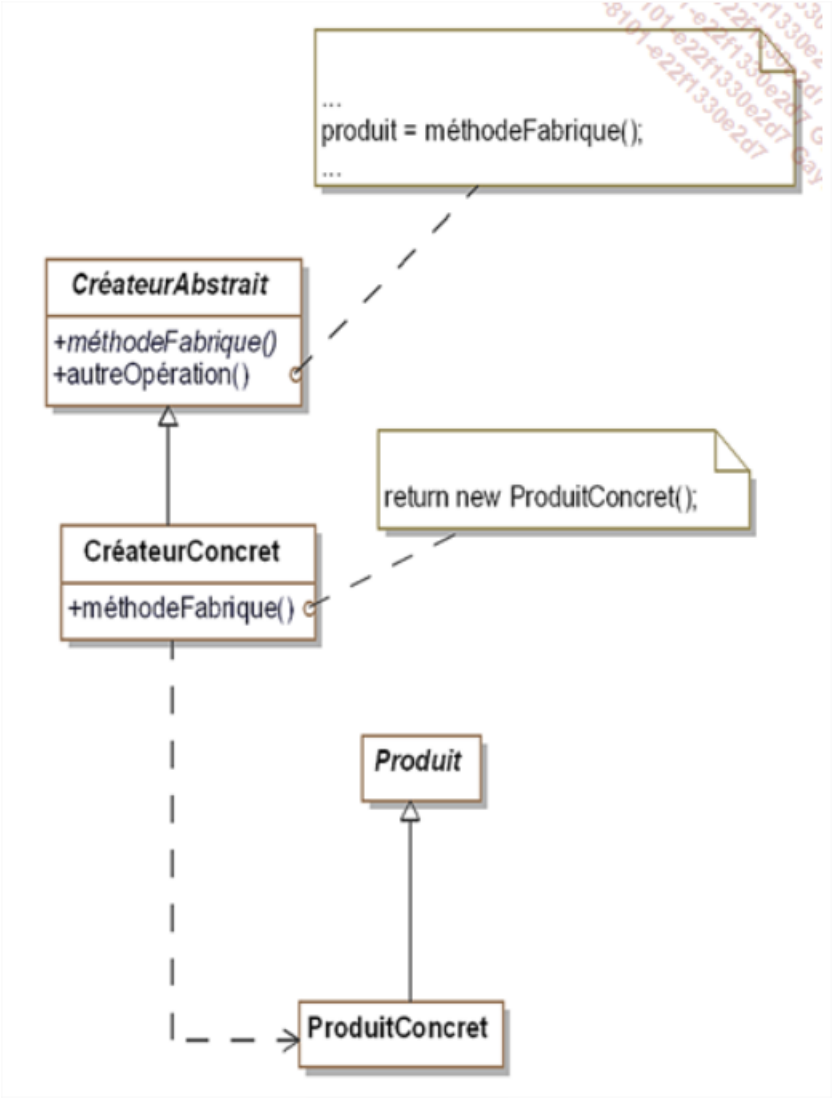
Le but du pattern **Factory Method** est d'introduire une méthode abstraite de création d'un objet en reportant aux sous-classes concrètes la création effective.

Plus de détails :

Le patron fabrique, ou méthode fabrique (en anglais factory ou factory method) fournit une interface pour créer des familles d'objets sans spécifier la classe concrète. Une fabrique simple retourne une instance d'une classe parmi plusieurs possibles, en fonction des paramètres qui ont été fournis. Toutes les classes ont un lien de parenté, et des méthodes communes, et chacune est optimisée en fonction d'une certaine donnée.

Structure

Diagramme de classes

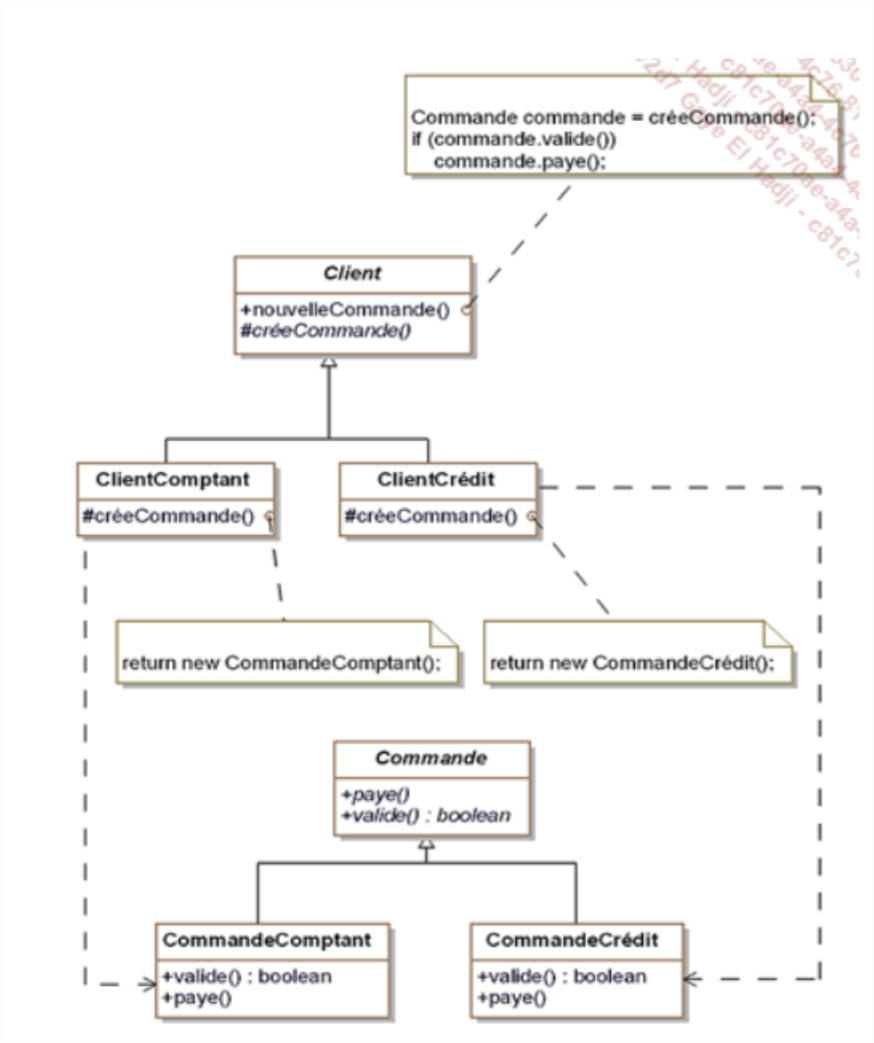


Exemple d'implémentation :

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 ci-dessous.



Veuillez réaliser cet exemple concrètement en utilisant les fichiers en pièces jointes.

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.

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.

c. Abstract Factory

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.

Plus de détails :

Le patron fabrique, ou méthode fabrique (en anglais factory ou factory method) fournit une interface pour créer des familles d'objets sans spécifier la classe concrète. Ce dernier est un patron récurrent. Une fabrique simple retourne une instance d'une classe parmi plusieurs possibles, en fonction des paramètres qui ont été fournis. Toutes les classes ont un lien de parenté, et des méthodes communes, et chacune est optimisée en fonction d'une certaine donnée. Le patron fabrique abstraite (en anglais abstract factory) va un pas plus loin que la fabrique simple. Une fabrique abstraite est utilisée pour obtenir un jeu d'objets connexes. Par exemple pour implémenter une charte graphique : il existe une fabrique qui retourne des objets (boutons, menus) dans le style de Windows, une qui retourne des objets dans le style de Motif, et une dans le style de Macintosh. Une fabrique abstraite est obtenue en utilisant une fabrique simple.

Exemple d'implémentation :

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 ci-dessous, pour l'objet scooter, il existe une classe abstraite Scooter et deux sous-classes concrètes ScooterÉlectricité et ScooterEssence.

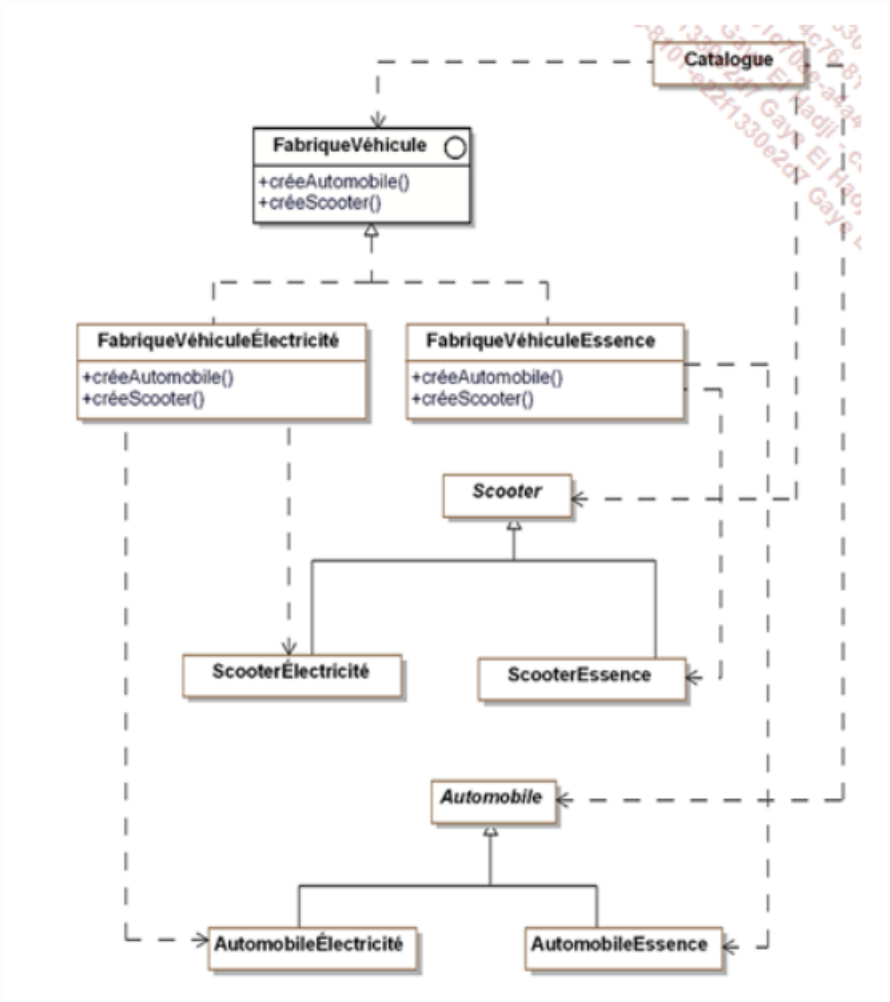
L'objet Catalogue peut utiliser ces sous-classes concrètes pour instancier les produits. Cependant si, par la suite, de nouvelles familles de véhicules doivent être prises en compte par la suite (diesel ou mixte essence-électricité), les modifications à apporter à l'objet Catalogue peuvent être assez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

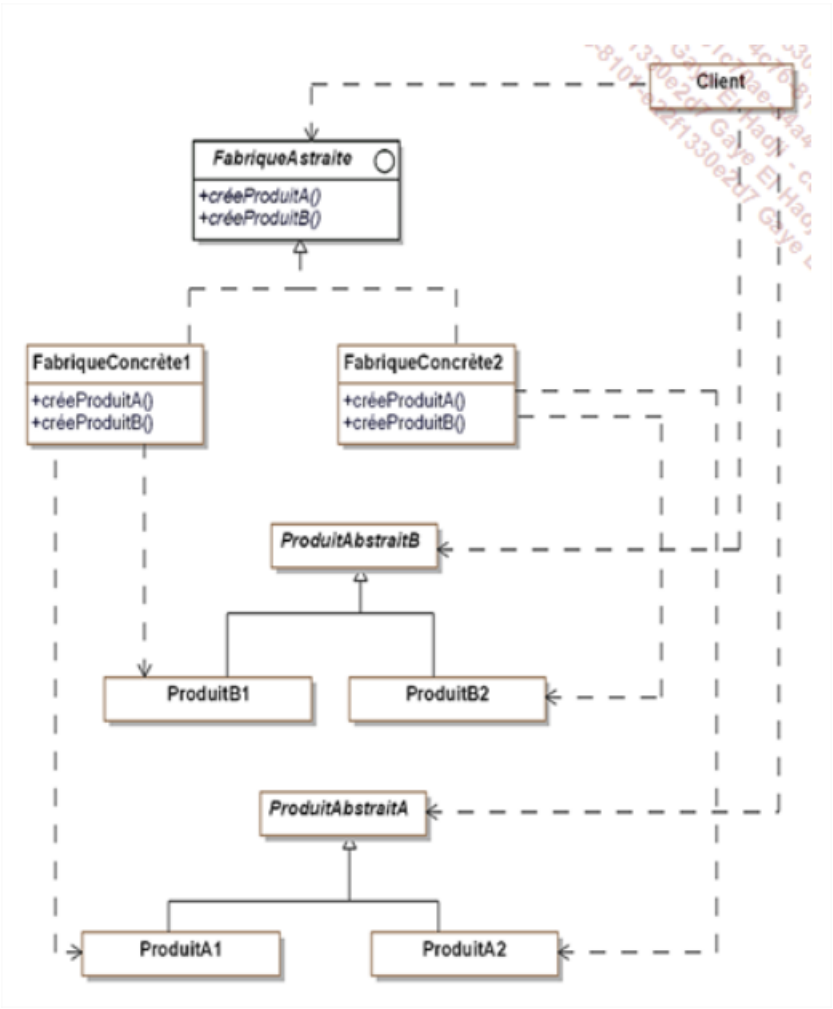
L'ensemble des classes du pattern Abstract Factory pour cet exemple est détaillé à la figure ci-dessous.



Veuillez réaliser cet exemple concrètement en utilisant les fichiers en annexe.

Structure

Diagramme de classes



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 produits. Connaissant la famille et le produit, elles sont capables de créer une instance du produit pour cette famille.
- ProduitAbstraitA et ProduitAbstraitB (Scooter et Automobile) sont les classes abstraites des produits indépendamment de leur famille. Les familles sont introduites dans leurs sous-classes concrètes.
- Client est la classe qui utilise l'interface de FabriqueAbstraite.

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.

d. Builder

Description :

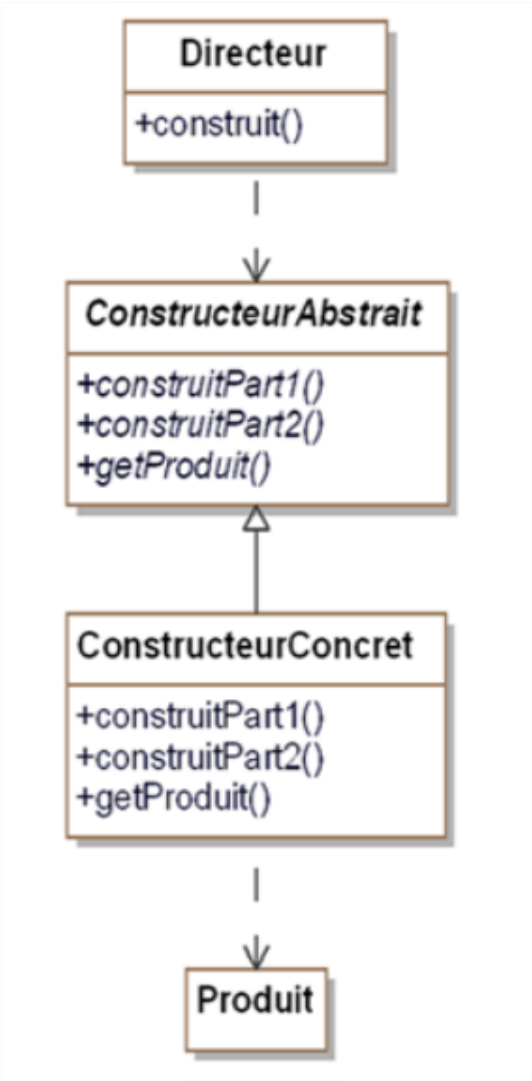
Le but du pattern **Builder** est d'abstraire la construction d'objets complexes de leur implantation de sorte qu'un client puisse créer ces objets complexes sans devoir se préoccuper des différences d'implantation.

Plus de détails :

Ce patron sépare le processus de construction d'un objet du résultat obtenu. Permet d'utiliser le même processus pour obtenir différents résultats. C'est une alternative au pattern fabrique. Au lieu d'une méthode pour créer un objet, à laquelle est passée un ensemble de paramètres, la classe fabrique comporte une méthode pour créer un objet - le monteur (en anglais builder). Cet objet comporte des propriétés qui peuvent être modifiées et une méthode pour créer l'objet final en tenant compte de toutes les propriétés. Ce pattern est particulièrement utile quand il y a de nombreux paramètres de création, presque tous optionnels.

Structure

Diagramme de classes



Exemple d'implémentation :

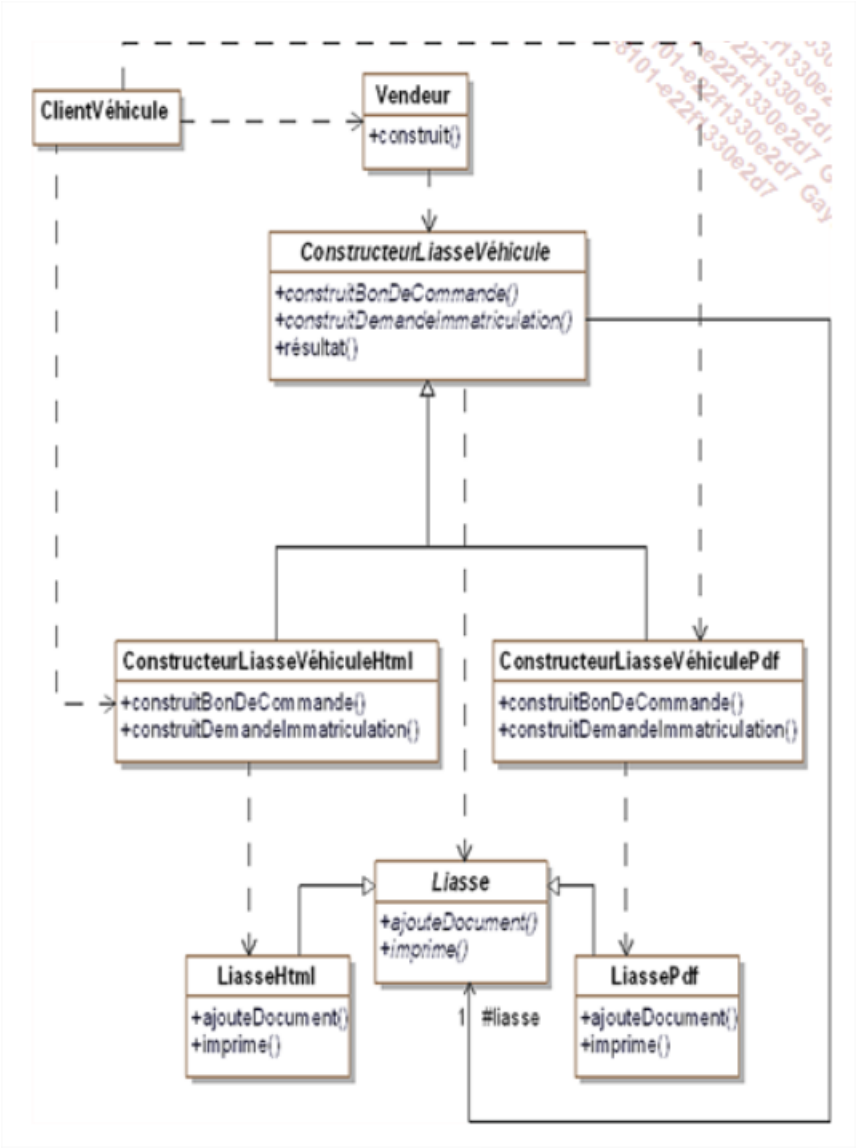
Lors de l'achat d'un véhicule, un vendeur crée une liasse de documents comprenant notamment le bon de commande et la demande d'immatriculation du client. Il peut construire ces documents au format HTML ou au format PDF selon le choix du client. Dans le premier cas, le client lui fournit une instance 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 ci-dessous. 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).



Veuillez réaliser cet exemple concrètement en utilisant les fichiers en annexe.

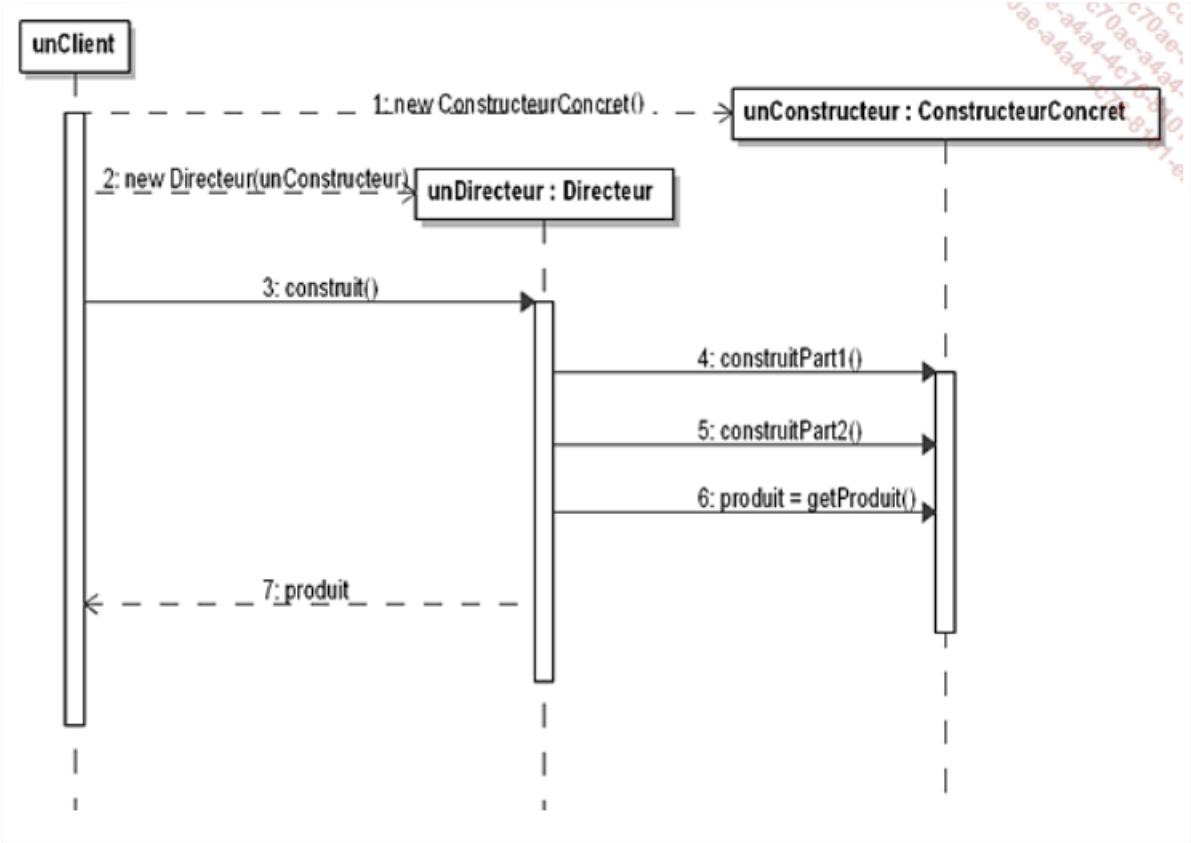
Participants

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 (ConstructeurLiasseVéhiculeHtml et 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.

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 ci-dessous illustre ce fonctionnement avec un diagramme de séquence UML.



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.

e. Prototype

Description :

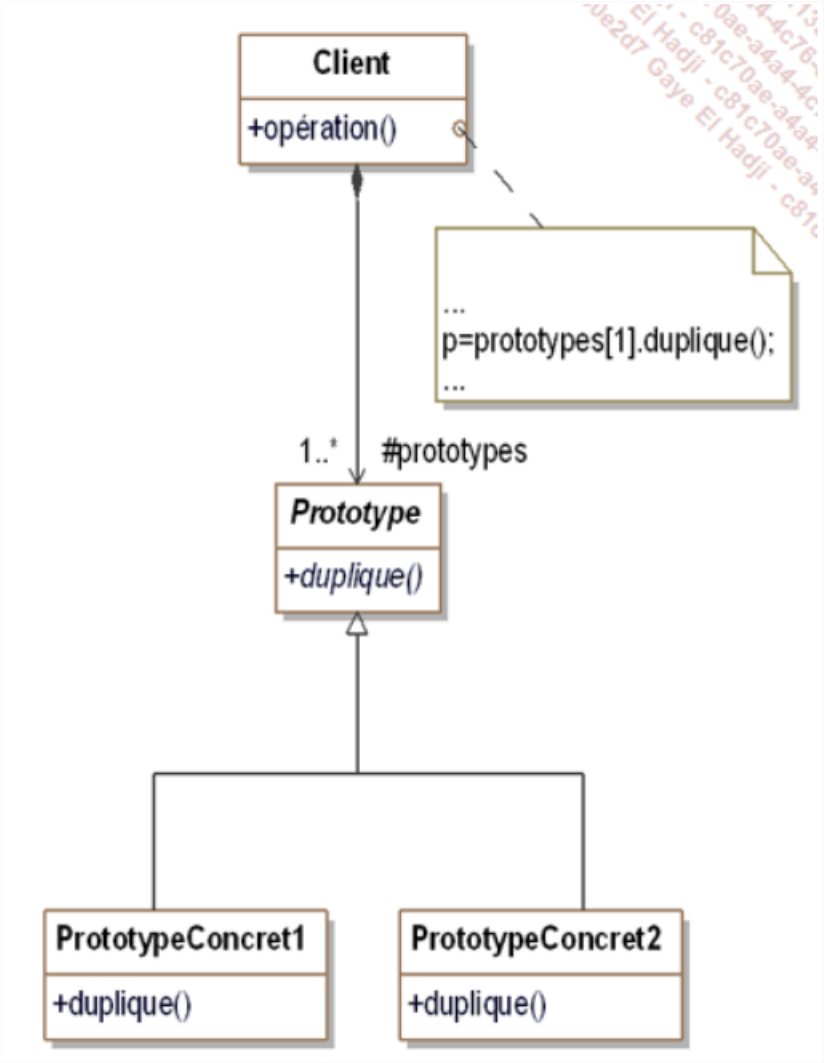
Le but du pattern est la création de nouveaux objets par duplication d'objets existants appelés prototypes qui disposent de la capacité de clonage.

Plus de détails :

Ce patron permet de définir le genre d'objet à créer en dupliquant une instance qui sert d'exemple. L'objectif de ce patron est d'économiser le temps nécessaire pour instancier des objets. Selon ce patron, une application comporte une instance d'un objet, qui sert de prototype. Cet objet comporte une méthode clone pour créer des duplicata. Des langages de programmation comme PHP ont une méthode clone incorporée dans tous les objets.

Structure

Diagramme de classes



Exemple d'implémentation :

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.

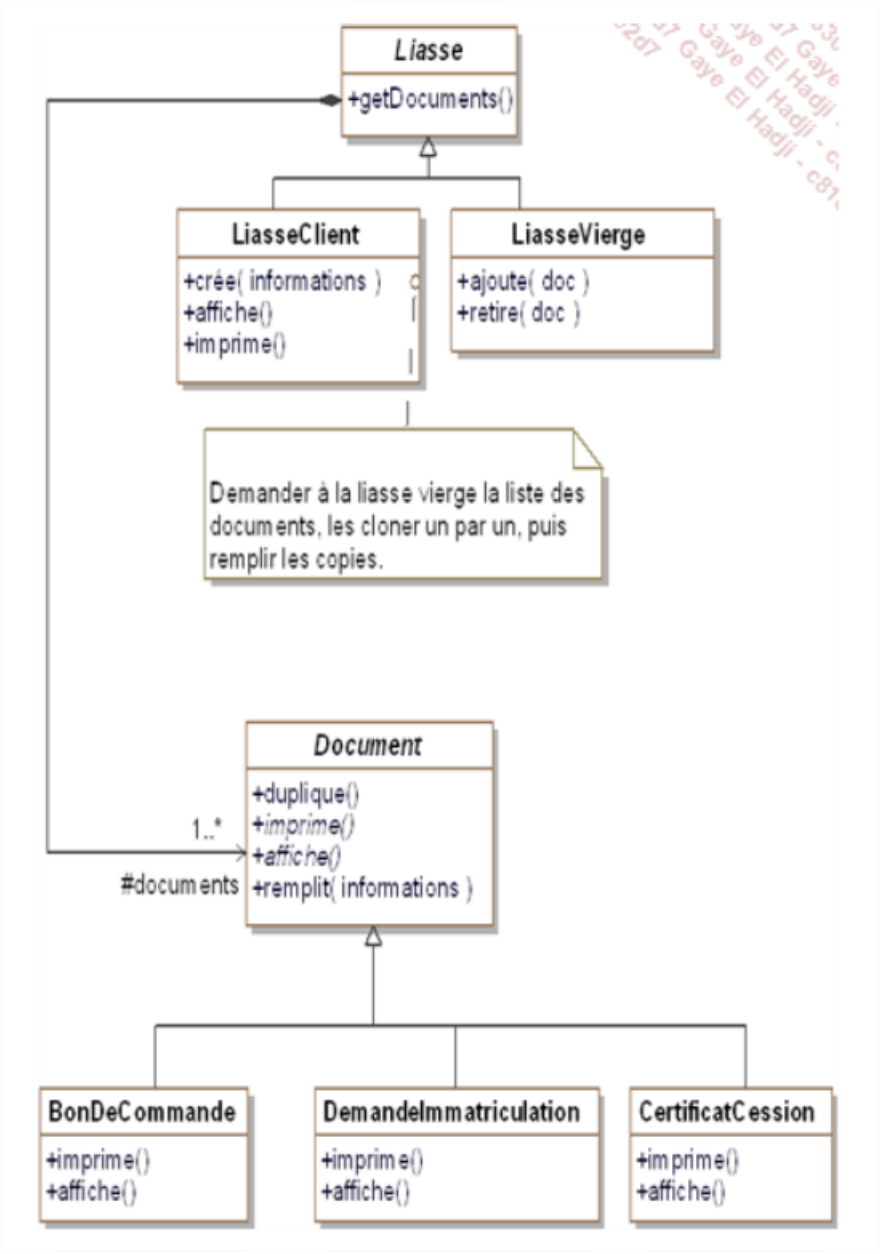
Une fois cette liasse vierge introduite, nous procédons par clonage pour créer les nouvelles liasses. Chaque nouvelle liasse est créée en dupliquant tous les documents de la liasse vierge.

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

La figure ci-dessous 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.



Veuillez réaliser cet exemple concrètement en utilisant les fichiers en annexe.

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.

Collaborations

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.

f. Adapter

Description :

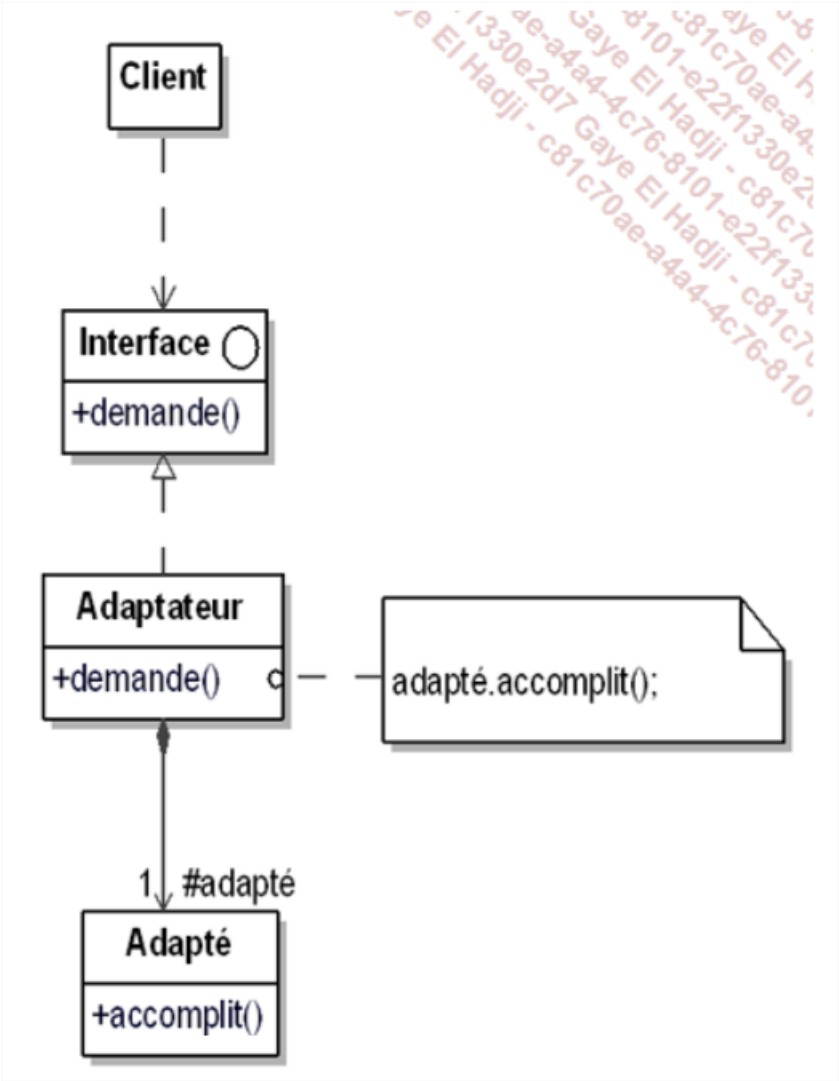
Le but du pattern **Adapter** est de convertir l'interface d'une classe existante en l'interface attendue par des clients également existants afin qu'ils puissent travailler ensemble. Il s'agit de conférer à une classe existante une nouvelle interface pour répondre aux besoins de clients.

Plus de détails :

Ce patron convertit l'interface d'une classe en une autre interface exploitée par une application. Permet d'interconnecter des classes qui sans cela seraient incompatibles. Il est utilisé dans le cas où un programme se sert d'une bibliothèque de classe qui ne correspond plus à l'utilisation qui en est faite, à la suite d'une mise à jour de la bibliothèque dont l'interface a changé. Un objet adaptateur (en anglais adapter) expose alors l'ancienne interface en utilisant les fonctionnalités de la nouvelle.

Structure

Diagramme de classes

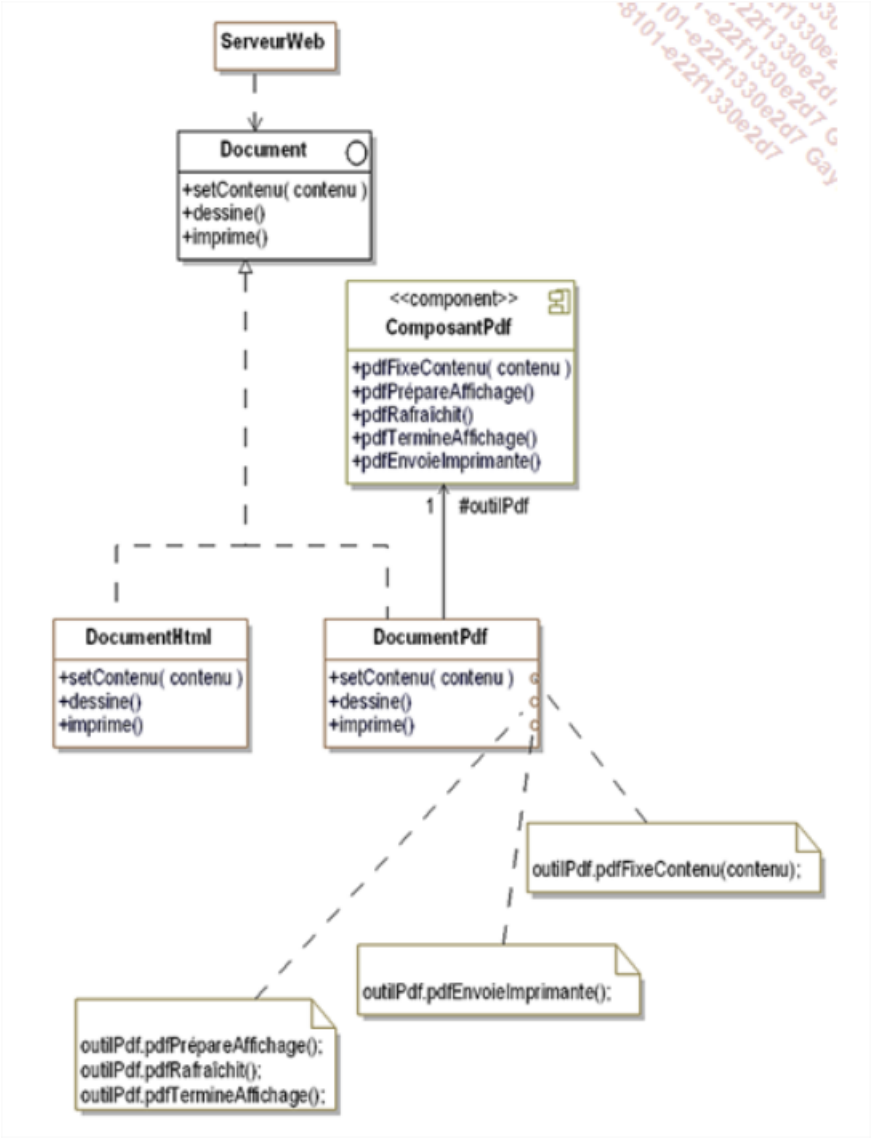


Exemple d'implémentation :

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 ci-dessous 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 ci-dessous 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 ci-dessous, le code des méthodes étant détaillé à l'aide de notes.



Veuillez réaliser cet exemple concrètement en utilisant les fichiers en annexe.

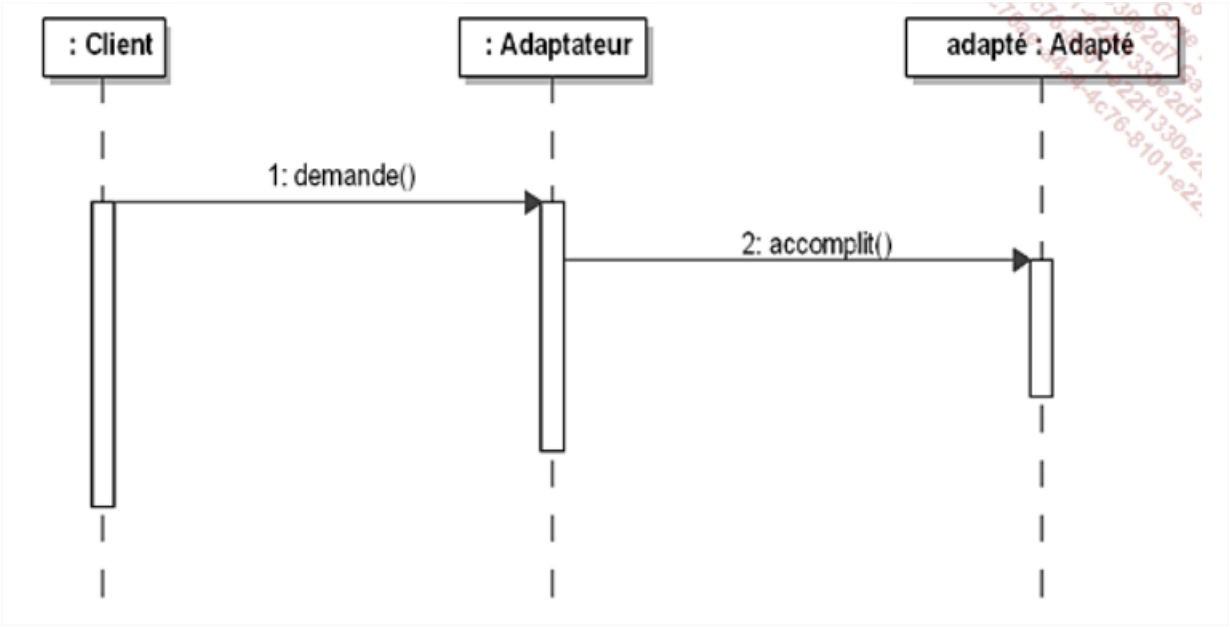
Participants

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 d'interface en invoquant les méthodes de l'objet adapté.
- Adapté (ComposantPdf) introduit l'objet dont l'interface doit être adaptée pour correspondre à Interface.

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 ci-dessous.



Domaines d'utilisation

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.

g. Bridge

Description :

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

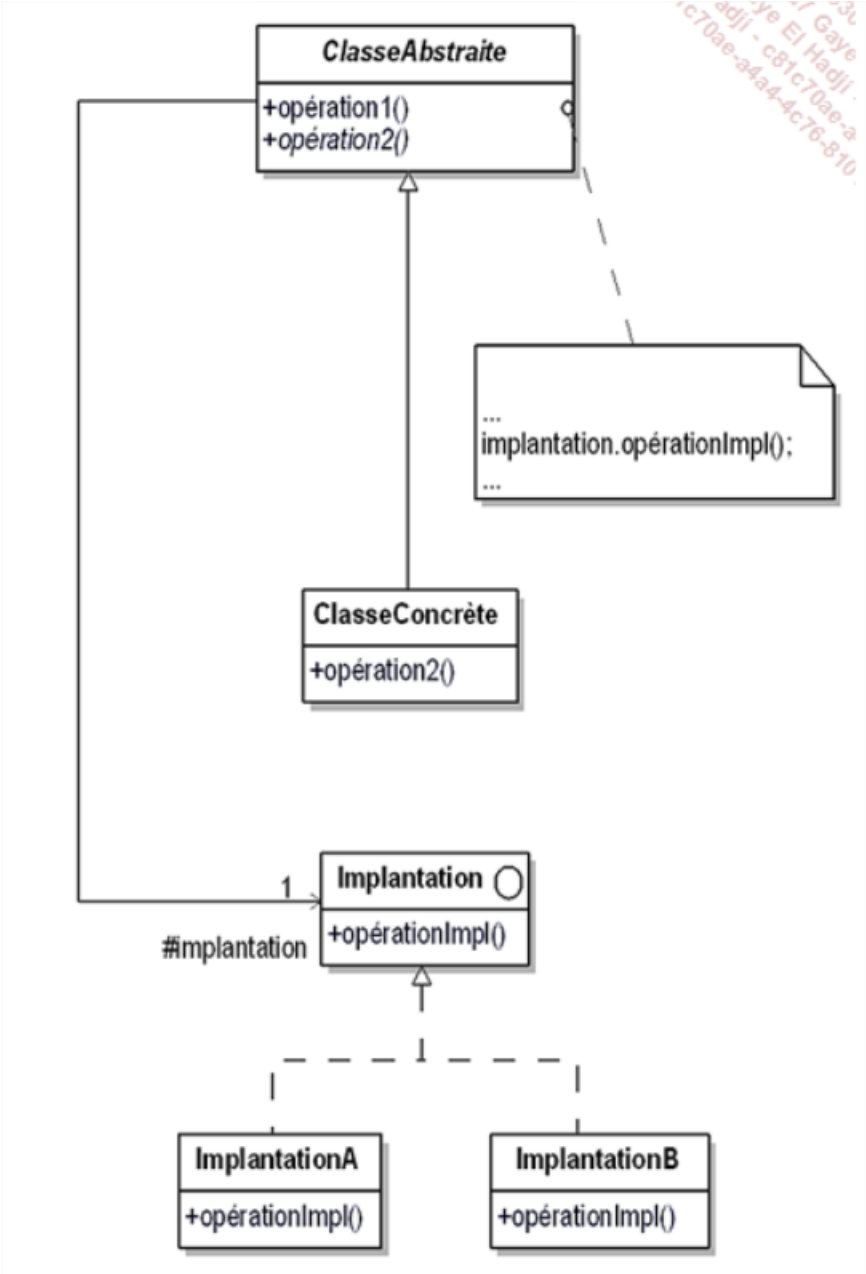
Ainsi, d'une part l'implantation peut être totalement encapsulée et d'autre part l'implantation la représentation peuvent évoluer indépendamment et sans que l'une exerce une contrainte sur l'autre.

Plus de détails :

Ce patron permet de découpler une abstraction de son implémentation, de telle manière qu'ils peuvent évoluer indépendamment. Il consiste à diviser une implémentation en deux parties : une classe d'abstraction qui définit le problème à résoudre, et une seconde classe qui fournit une implémentation. Il peut exister plusieurs implémentations pour le même problème et la classe d'abstraction comporte une référence à l'implémentation choisie, qui peut être changée selon les besoins. Le patron pont (en anglais bridge) est fréquemment utilisé pour réaliser des récepteurs d'événements.

Structure

Diagramme de classes



Exemple d'implémentation :

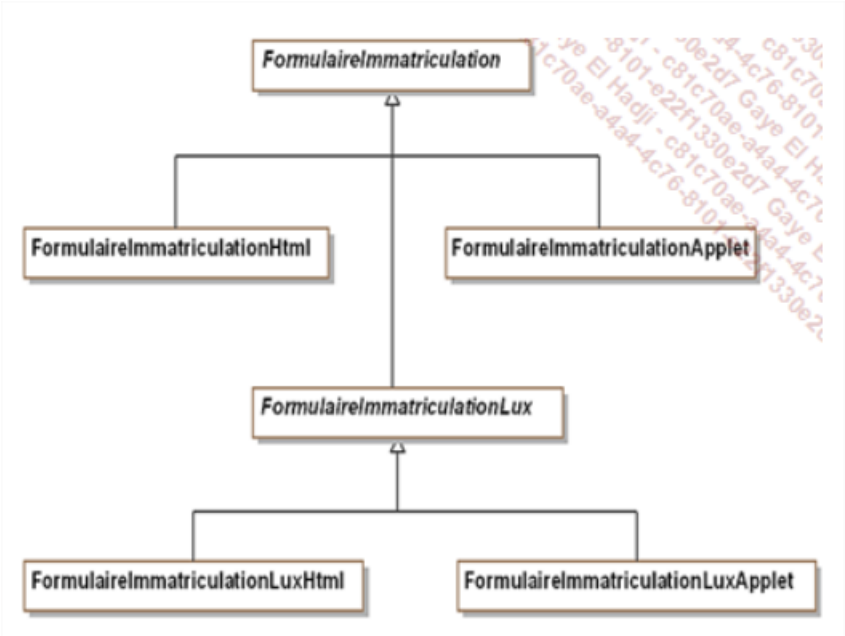
Pour effectuer une demande d'immatriculation d'un véhicule d'occasion, il convient de préciser sur cette demande certaines informations importantes comme le numéro de la plaque existante. Le système affiche un formulaire pour demander ces informations.

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 première figure ci-dessous montre le diagramme de classes correspondant.



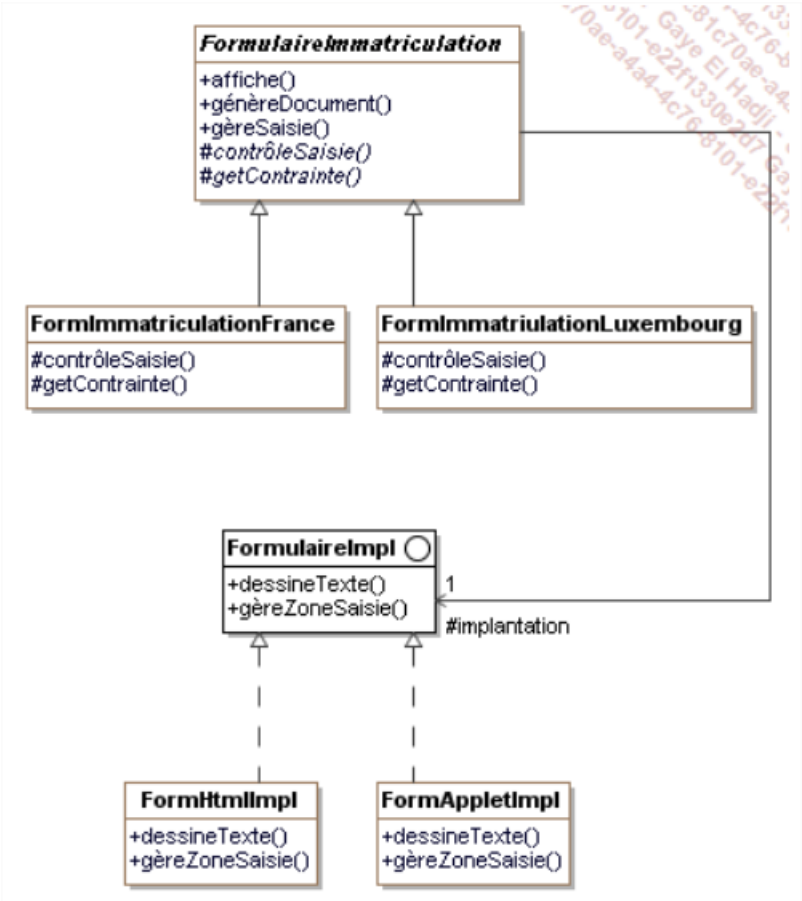
Ce diagramme met en avant deux problèmes :

- 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 ci-dessous. 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).



Veuillez réaliser cet exemple concrètement en utilisant les fichiers en annexe.

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 (FormImmatriculationFrance et 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.

Collaborations

Les opérations de ClasseAbstraite et de ses sous-classes invoquent les méthodes introduites dans l'interface Implantation.

Domaines d'utilisation

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 de l'exemple d'implémentation ci-dessous.

h. Composite

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.

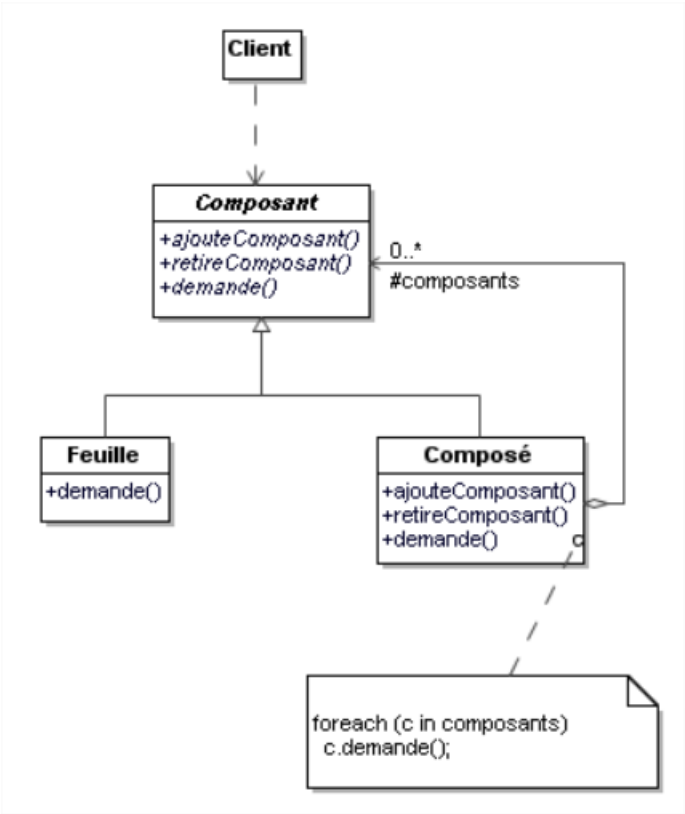
Par ailleurs, cette composition est encapsulée vis-à-vis des clients des objets qui peuvent interagir sans devoir connaître la profondeur de la composition.

Plus de détails :

Le patron composite (même nom en anglais) permet de composer une hiérarchie d'objets, et de manipuler de la même manière un élément unique, une branche, ou l'ensemble de l'arbre. Il permet en particulier de créer des objets complexes en reliant différents objets selon une structure en arbre. Ce patron impose que les différents objets aient une même interface, ce qui rend uniformes les manipulations de la structure. Par exemple dans un traitement de texte, les mots sont placés dans des paragraphes disposés dans des colonnes dans des pages ; pour manipuler l'ensemble, une classe composite implémente une interface. Cette interface est héritée par les objets qui représentent les textes, les paragraphes, les colonnes et les pages.

Structure

Diagramme de classes



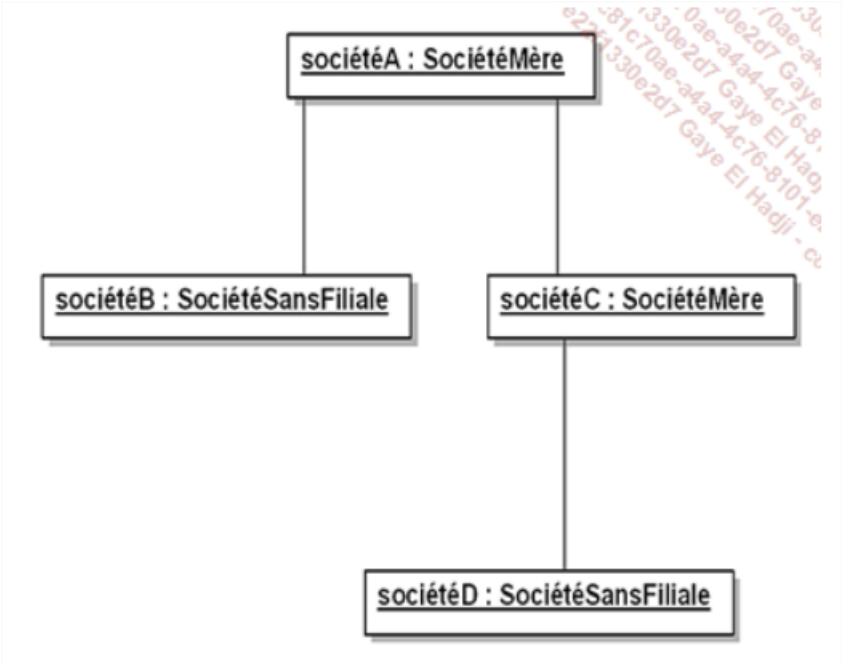
Exemple d'implémentation :

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.

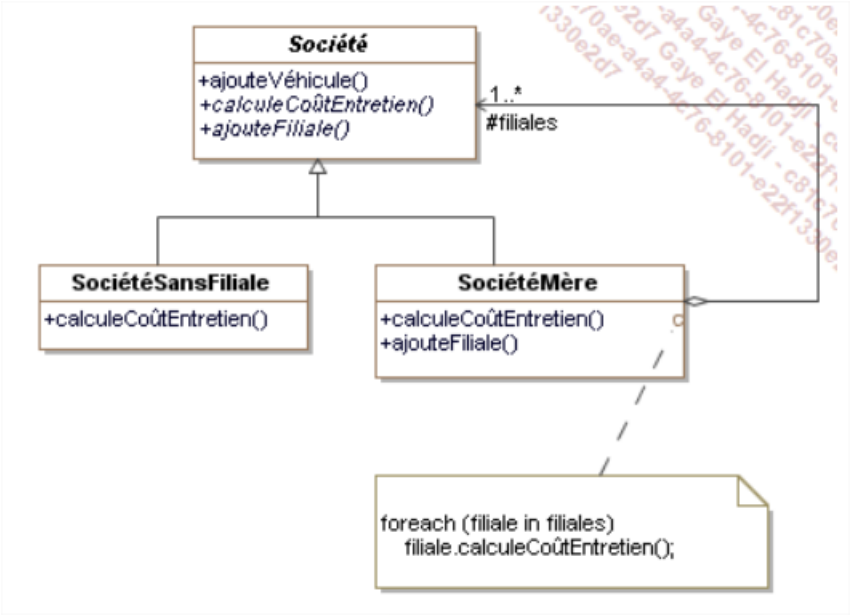
Les sociétés qui possèdent des filiales demandent des offres de maintenance qui prennent en compte le parc de véhicules de leurs filiales.

Une solution immédiate consiste à traiter différemment les sociétés sans filiale et celles 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 société 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 première figure ci-dessous où les sociétés mères sont placées au-dessus de leurs filiales.



La deuxième figure ci-dessous 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.



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).

Veuillez réaliser cet exemple concrètement en utilisant les fichiers en pièces jointes.

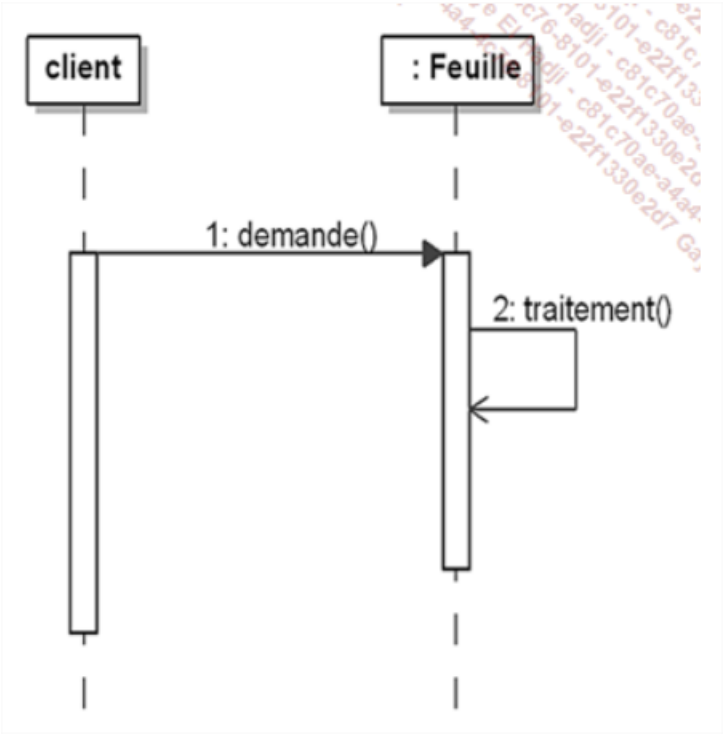
Participants

Les participants au pattern sont les suivants :

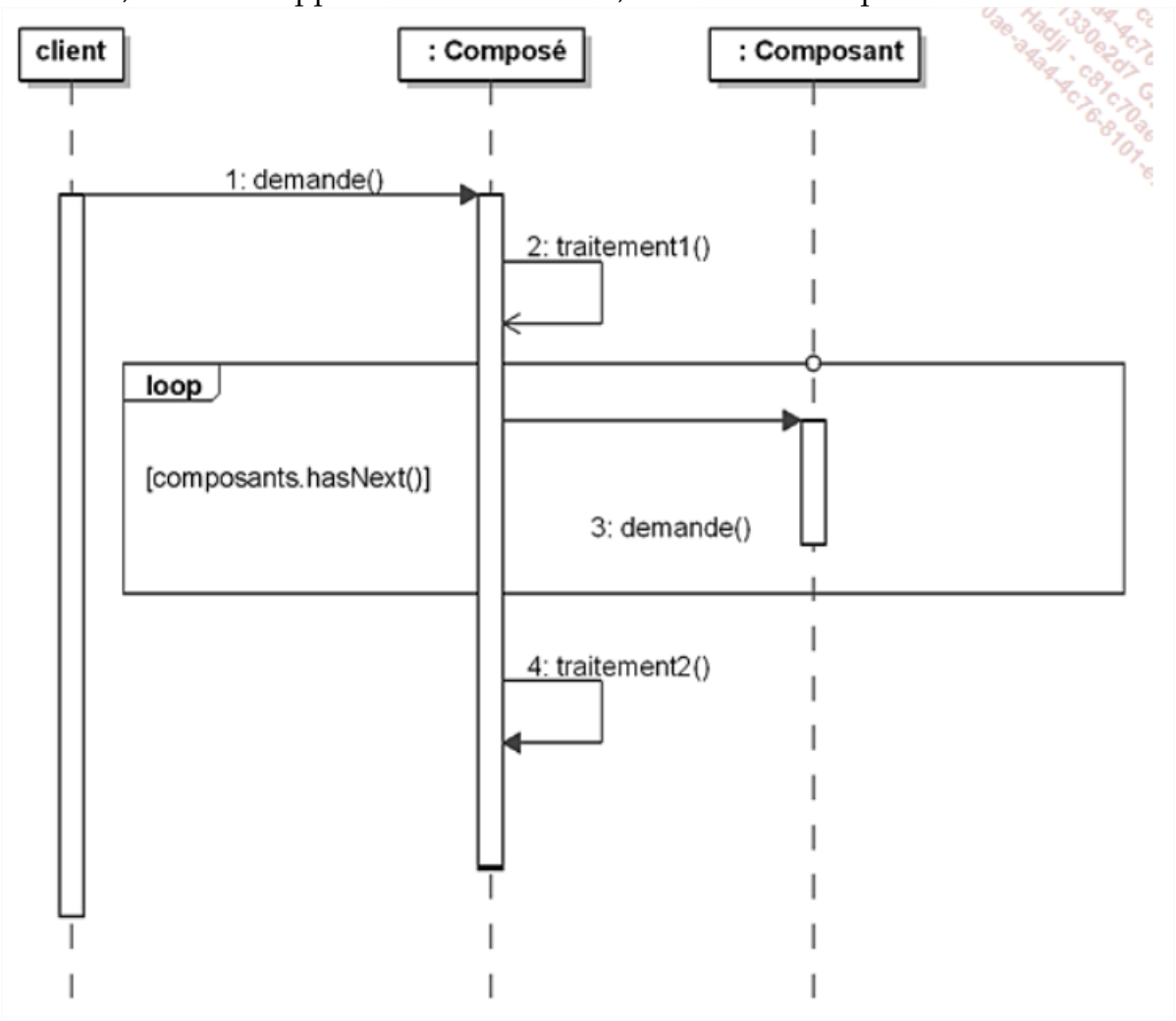
- Composant (Société) est la classe abstraite qui introduit l'interface des objets de la composition, implante les méthodes communes et introduit la signature des méthodes qui gèrent la composition en ajoutant ou en supprimant des composants.
- Feuille (SociétéSansFiliale) est la classe concrète qui 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.

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 première figure ci-dessous.



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 deuxième figure 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.



Domaines d'utilisation

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.

i. Decorator

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.

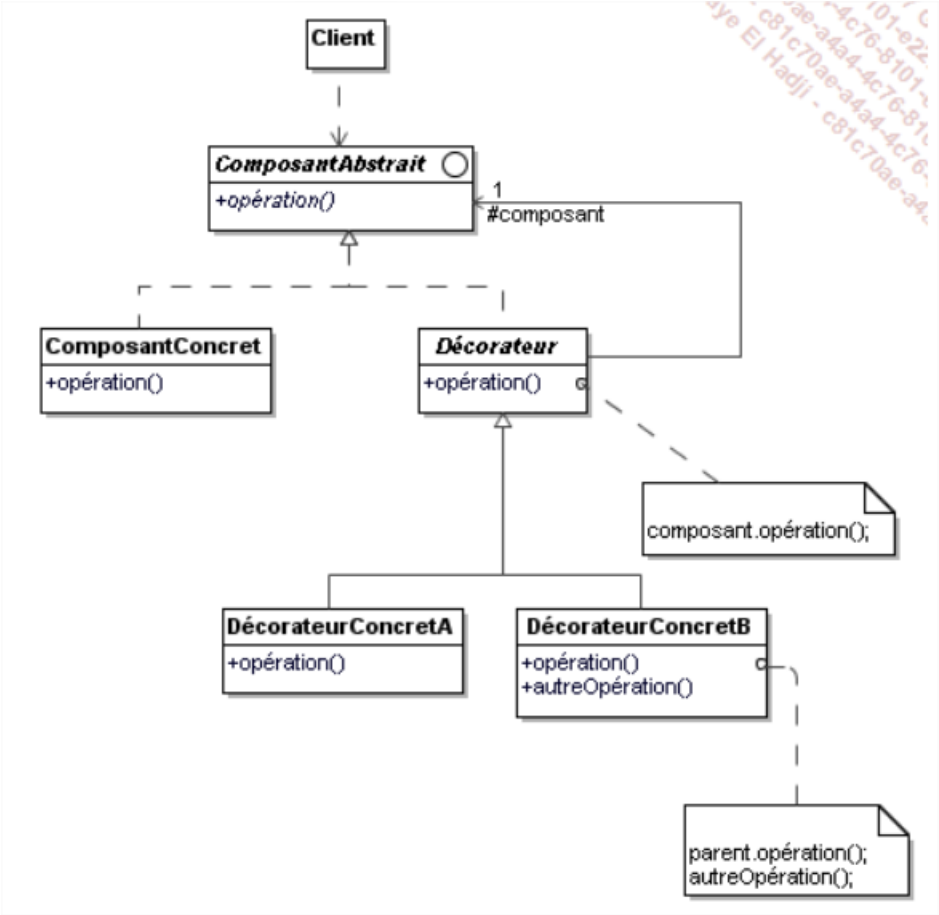
Le pattern **Decorator** constitue une alternative par rapport à la création d'une sous-classe pour enrichir un objet.

Plus de détails :

Le patron décorateur (en anglais decorator) permet d'attacher dynamiquement des responsabilités à un objet. Une alternative à l'héritage. Ce patron est inspiré des poupées russes. Un objet peut être caché à l'intérieur d'un autre objet décorateur qui lui rajoutera des fonctionnalités, l'ensemble peut être décoré avec un autre objet qui lui ajoute des fonctionnalités et ainsi de suite. Cette technique nécessite que l'objet décoré et ses décorateurs implémentent la même interface, qui est typiquement définie par une classe abstraite.

Structure

Diagramme de classes



Exemple d'implémentation :

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.

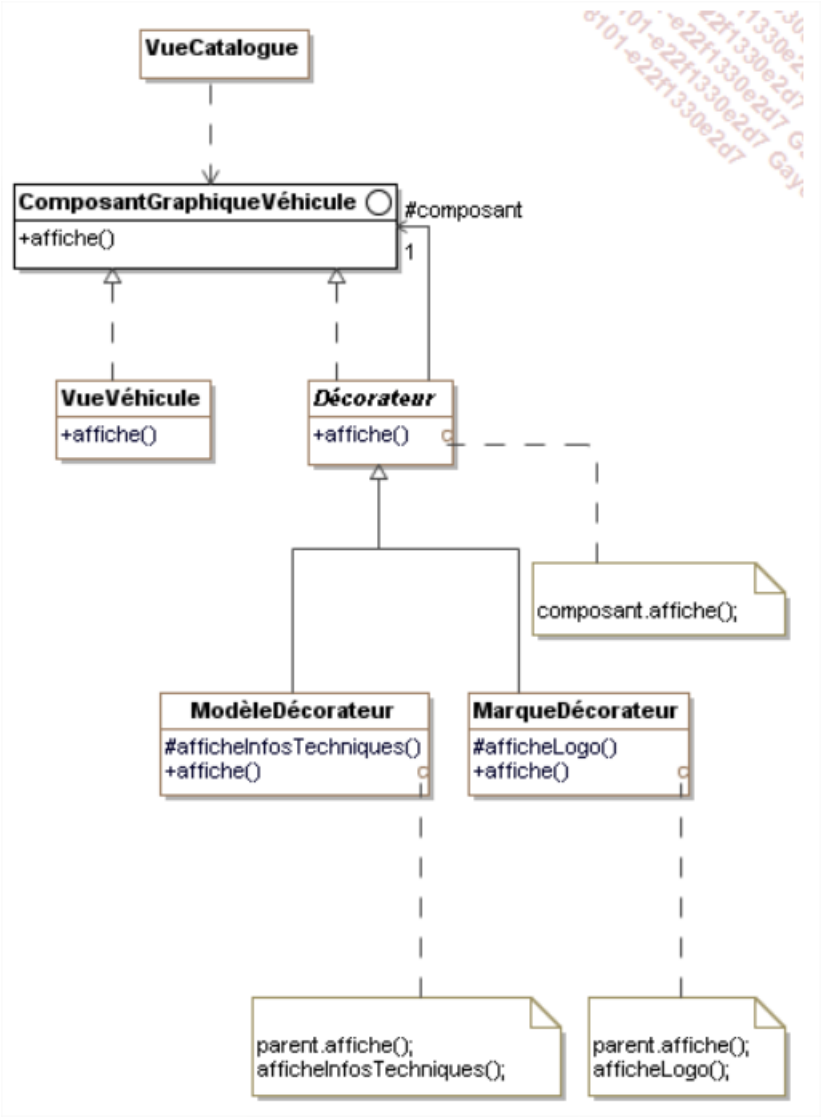
Nous voulons maintenant afficher des données supplémentaires pour les véhicules "haut de gamme", à savoir les informations techniques liées au modèle. Pour réaliser l'ajout de cette fonctionnalité, nous pouvons réaliser une sous-classe d'affichage spécifique pour les véhicules "haut de gamme". Maintenant, 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.

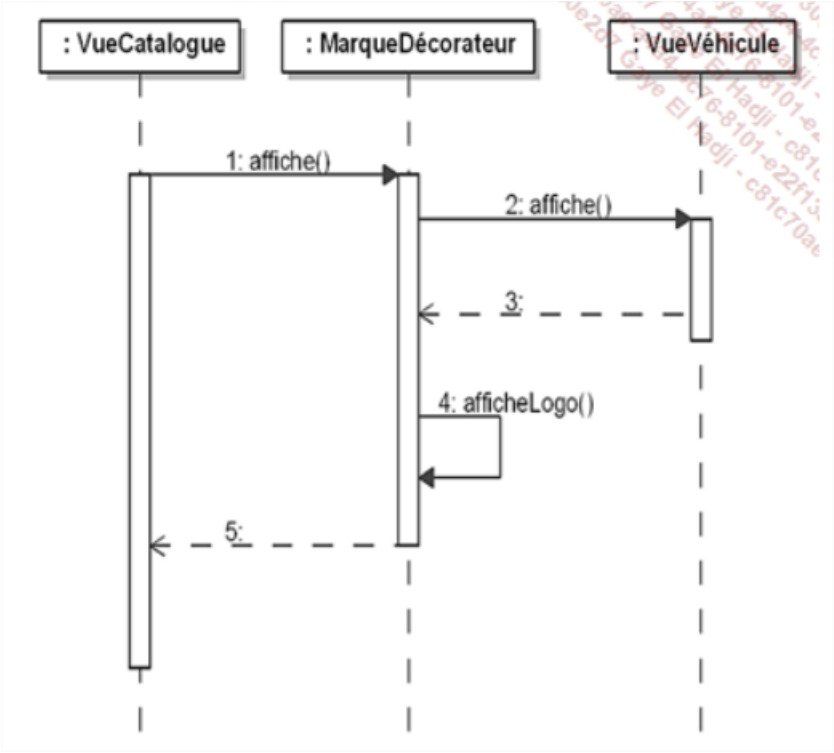
La première figure ci-dessous illustre l'utilisation du pattern Decorator pour enrichir l'affichage de véhicules. L'interface ComposantGraphiqueVéhicule constitue l'interface commune à la classe VueVéhicule, que nous voulons enrichir, et à la classe abstraite Décorateur, interface uniquement constituée 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.



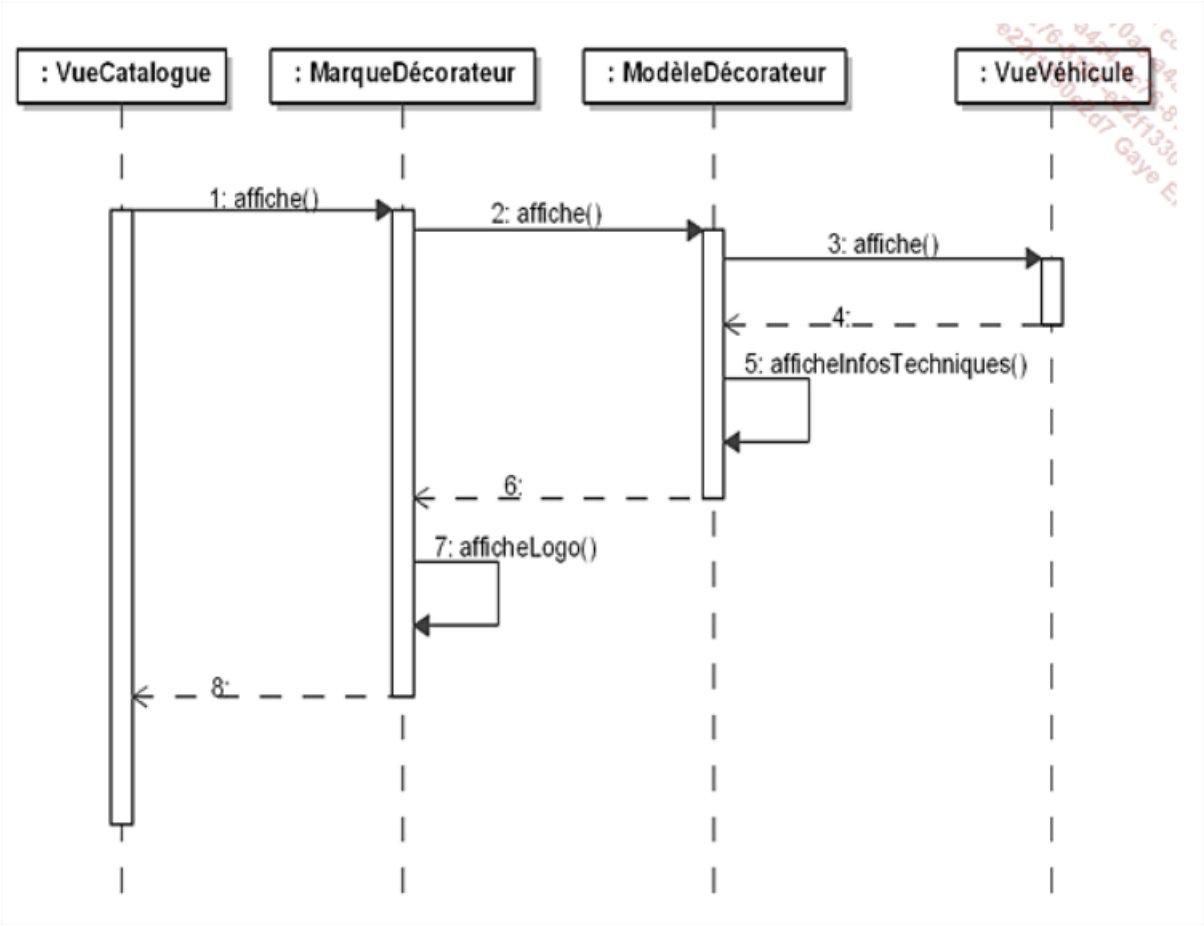
Il existe deux classes concrètes de décorateur, sous-classes de Décorateur. Leur méthode affiche commence par appeler la méthode affiche de Décorateur puis affiche les données complémentaires comme les informations techniques du modèle ou le logo de la marque.

La figure ci-dessous montre la séquence des appels de message destinés à l'affichage d'un véhicule pour lequel le logo de la marque est également affiché.



La figure ci-dessous 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.



Veuillez réaliser cet exemple concrètement en utilisant les fichiers en annexe.

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.

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'utilisation

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.

j. Facade

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.

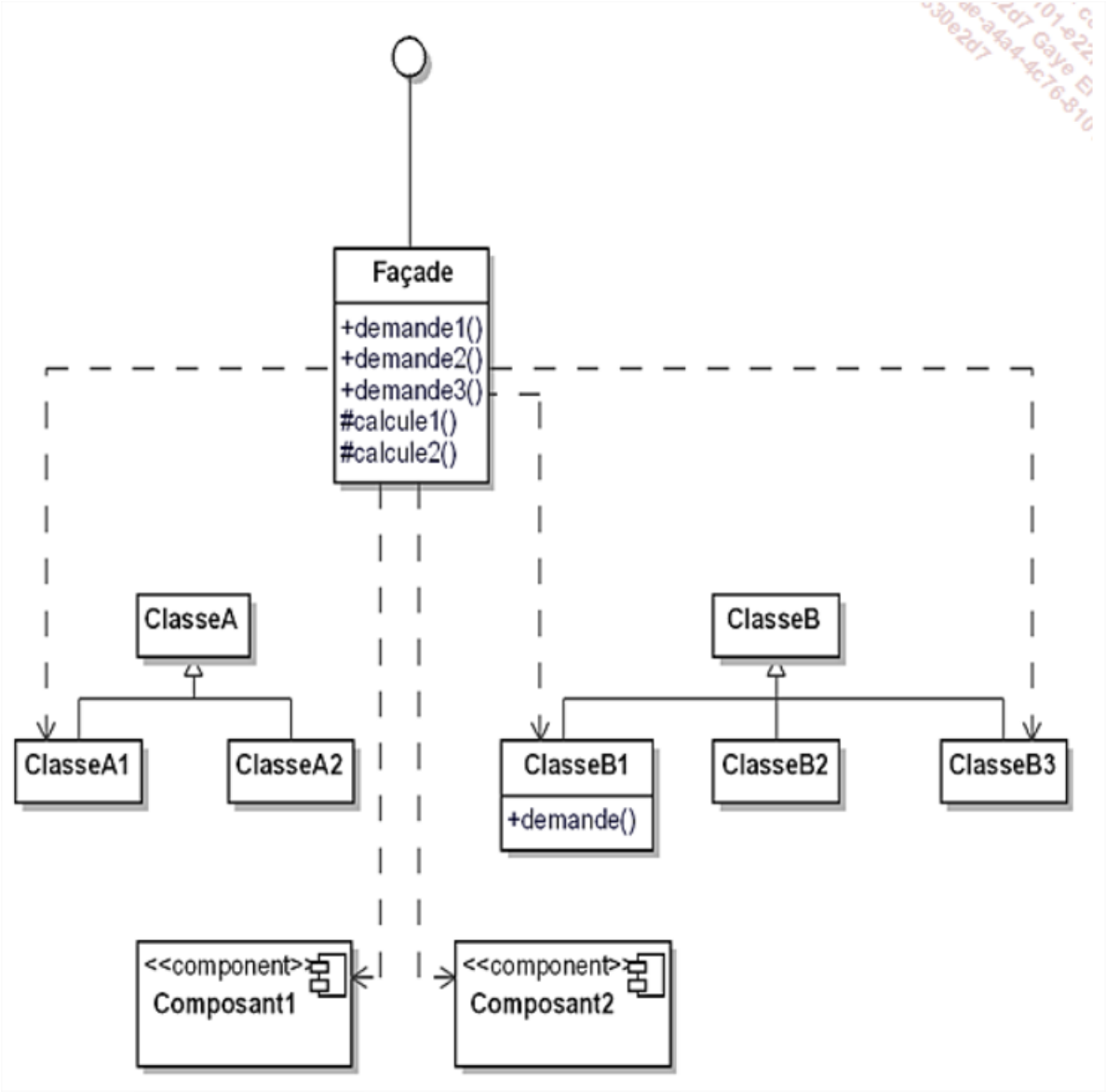
Le pattern **Facade** encapsule l'interface de chaque objet considérée comme interface de bas niveau dans une interface unique de niveau plus élevé. La construction de l'interface unifiée peut nécessiter d'implanter des méthodes destinées à composer les interfaces de bas niveau.

Encore plus de détails :

Le patron façade (en anglais facade) fournit une interface unifiée sur un ensemble d'interfaces d'un système. Il est utilisé pour réaliser des interfaces de programmation. Si un sous-système comporte plusieurs composants qui doivent être utilisée dans un ordre précis, une classe façade sera mise à disposition, et permettra de contrôler l'ordre des opérations et de cacher les détails techniques des sous-systèmes.

Structure

Diagramme de classes



Exemple d'implémentation :

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 :

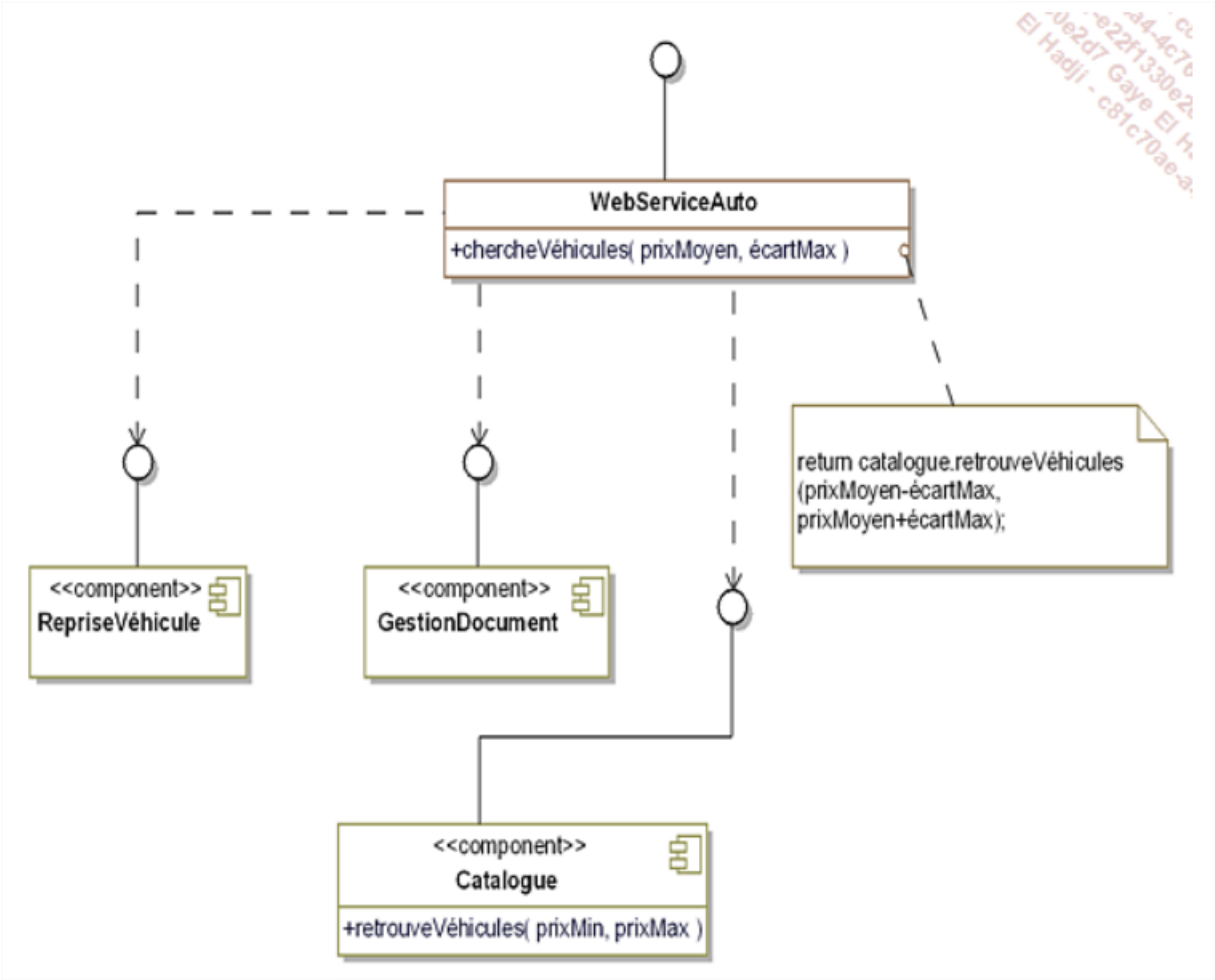
- Certaines fonctionnalités ne sont pas utiles aux clients du service web comme les fonctionnalités d'affichage du catalogue.
- L'architecture interne du système répond à des exigences de modularité et d'évolution qui ne font pas partie des besoins des clients du service web pour lesquels ces exigences 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 ci-dessous. 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.



Veuillez réaliser cet exemple concrètement en utilisant les fichiers en annexe.

Participants

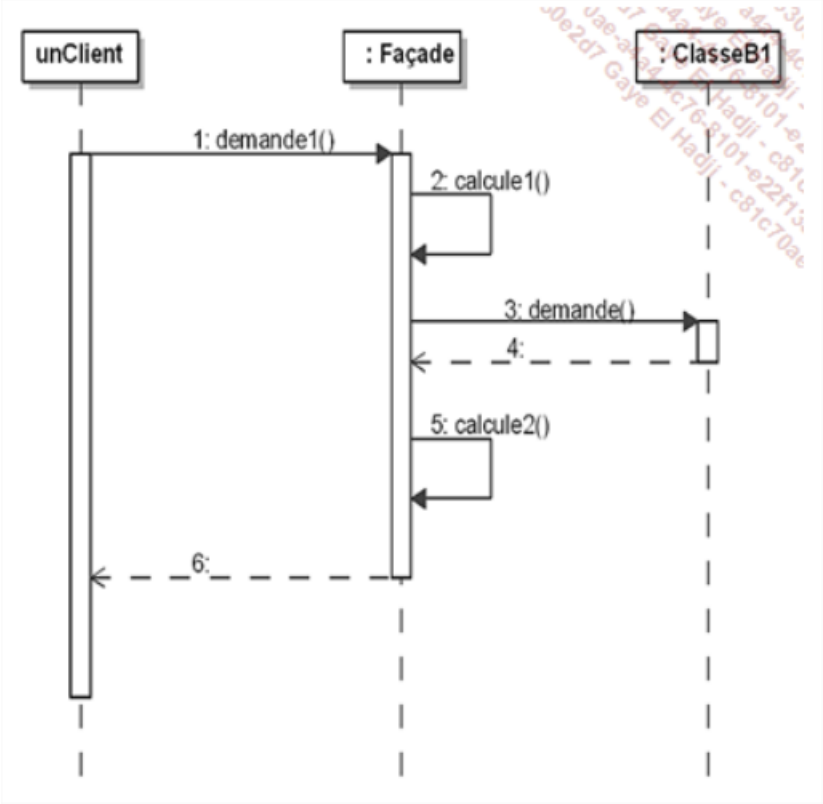
Les participants au pattern sont les suivants :

- 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.

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 ci-dessous 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.



Domaines d'utilisation

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.

k. Flyweight

Description :

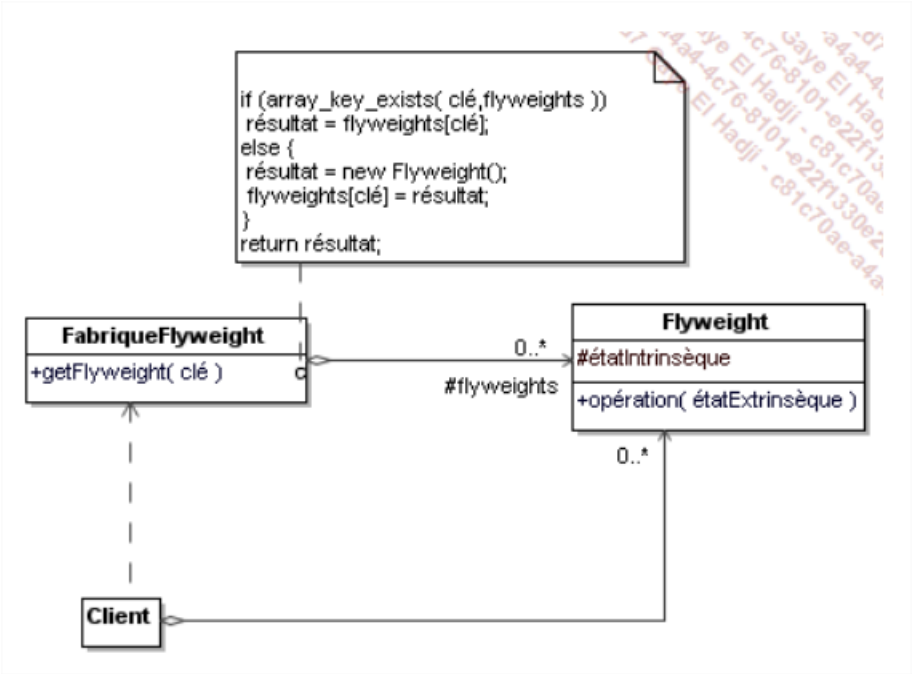
Le but du pattern **Flyweight** est de partager de façon efficace un ensemble important d'objets de grain fin.

Plus de détails :

Dans le patron flyweight (en français poids-mouche), un type d'objet est utilisé pour représenter une gamme de petits objets tous différents. Ce patron permet de créer un ensemble d'objets et de les réutiliser. Il peut être utilisé par exemple pour représenter un jeu de caractères : Un objet factory va retourner un objet correspondant au caractère recherché. La même instance peut être retournée à chaque fois que le caractère est utilisé dans un texte.

Structure

Diagramme de classes



Domaines d'utilisation

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.

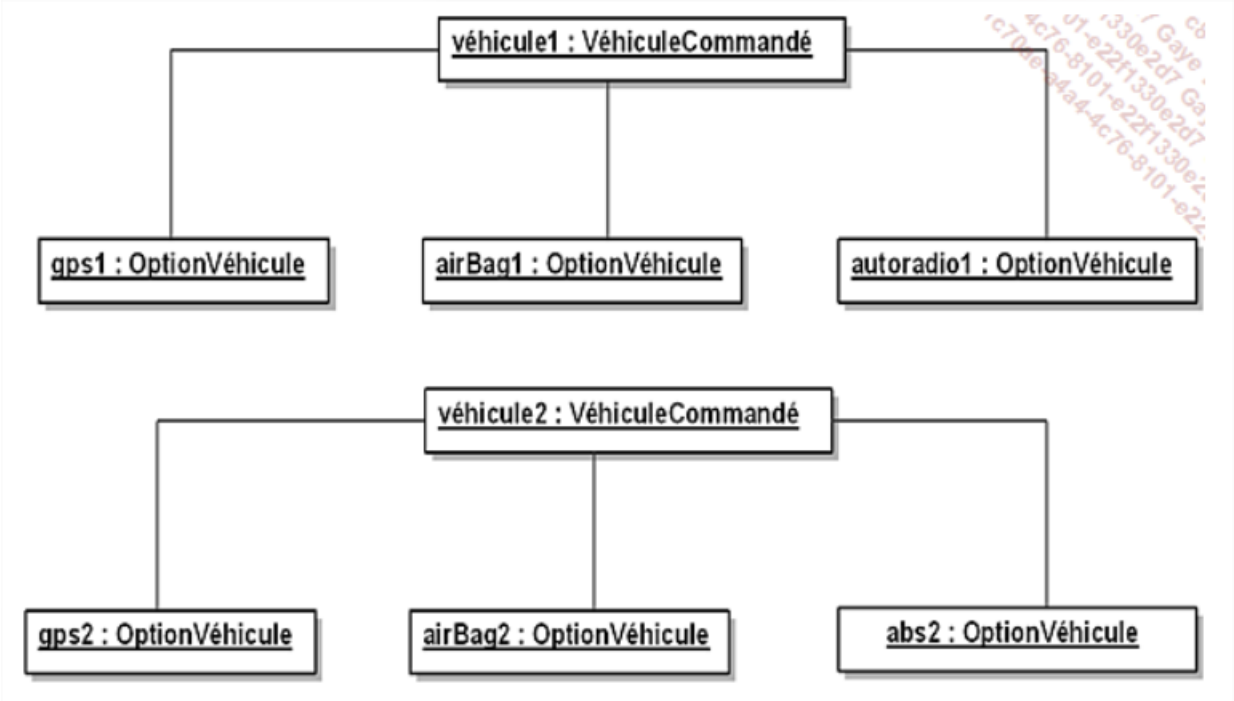
Exemple d'implémentation :

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 première figure ci-dessous 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.



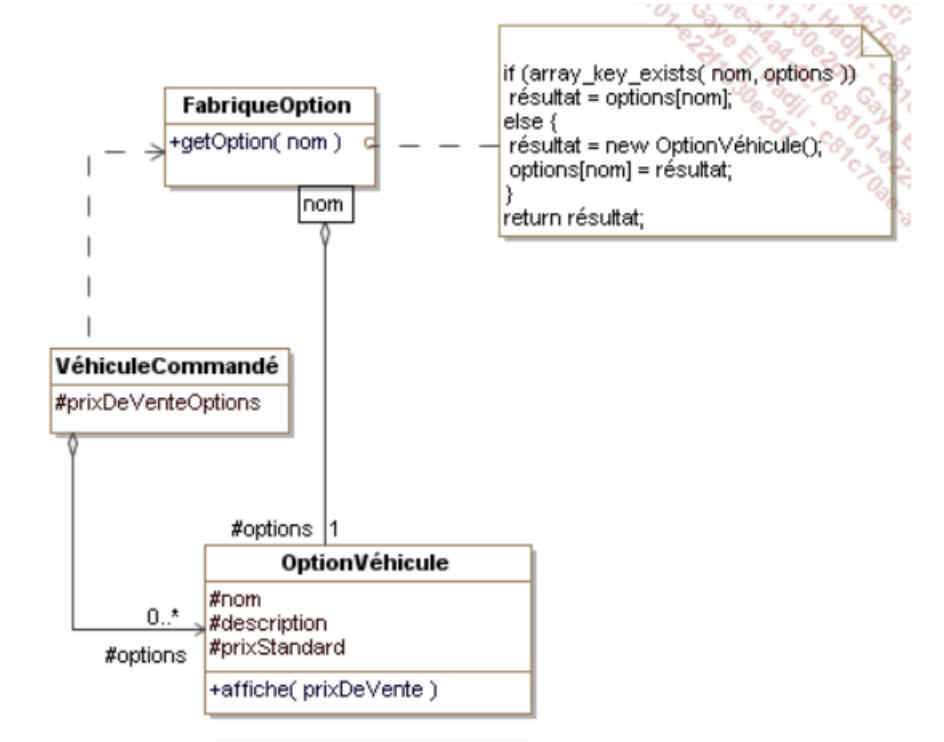
Le pattern Flyweight propose une solution à ce problème en partageant les options :

- 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 ci-dessous illustre le diagramme des classes de cette solution.



Ce diagramme introduit les classes suivantes :

- OptionVéhicule dont les attributs détiennent l'état intrinsèque d'une option. La méthode affiche a 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.

Veuillez réaliser cet exemple concrètement en utilisant les fichiers en annexe.

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.

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.

l. Proxy

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.

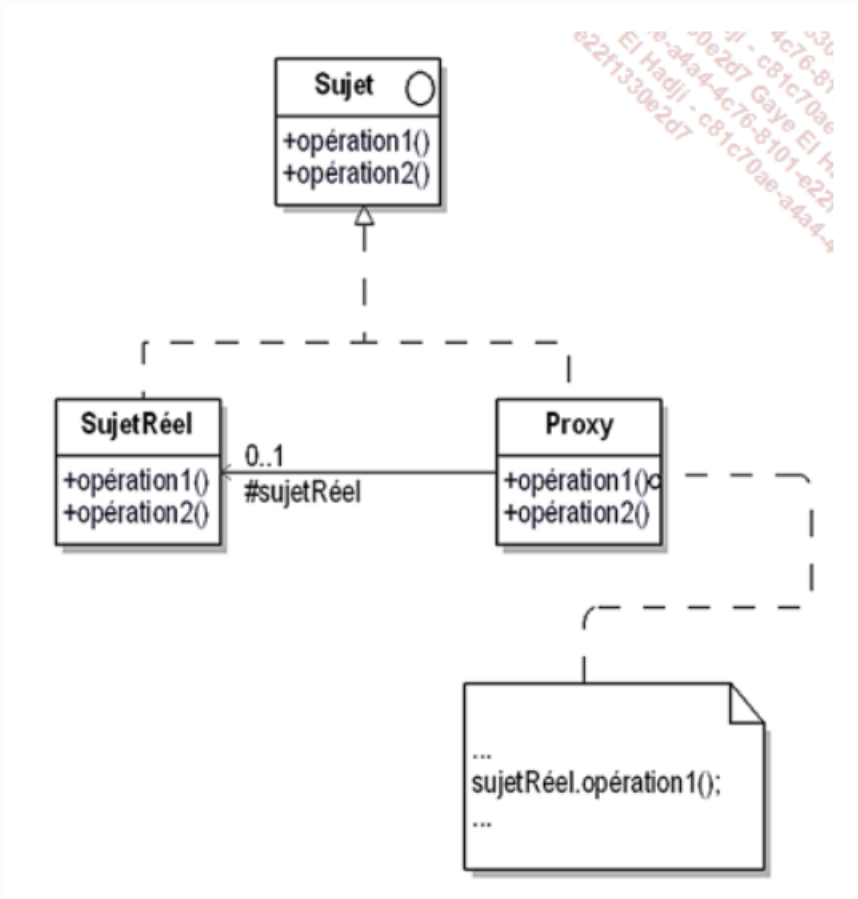
L'objet qui effectue la substitution possède la même interface que le sujet, ce qui rend cette substitution transparente vis-à-vis des clients.

Encore plus de détails :

Ce patron est un substitut d'un objet, qui permet de contrôler l'utilisation de ce dernier. Un proxy est un objet destiné à protéger un autre objet. Le proxy a la même interface que l'objet à protéger. Un proxy peut être créé par exemple pour permettre d'accéder à distance à un objet (via un middleware). Le proxy peut également être créé dans le but de retarder la création de l'objet protégé - qui sera créé immédiatement avant d'être utilisé. Dans sa forme la plus simple, un proxy ne protège rien du tout et transmet tous les appels de méthode à l'objet cible.

Structure

Diagramme de classes



Domaines d'utilisation

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 d'implémentation :

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.

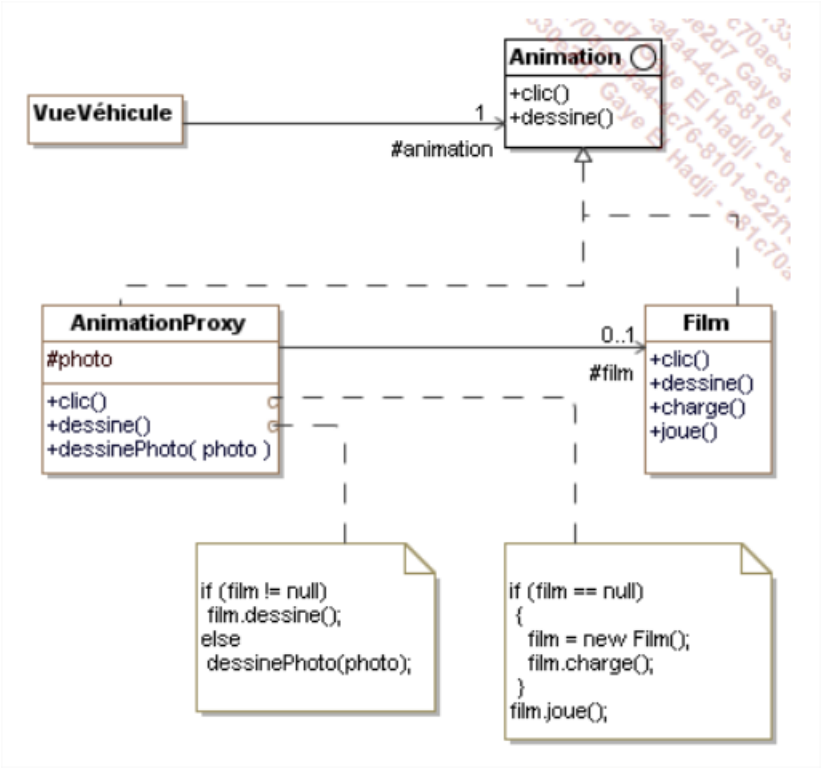
Une page du catalogue contient de nombreux véhicules et il est très lourd de créer en mémoire tous les objets d'animation car les films nécessitent une grande quantité de mémoire et leur transfert au travers d'un réseau prend beaucoup de temps.

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.

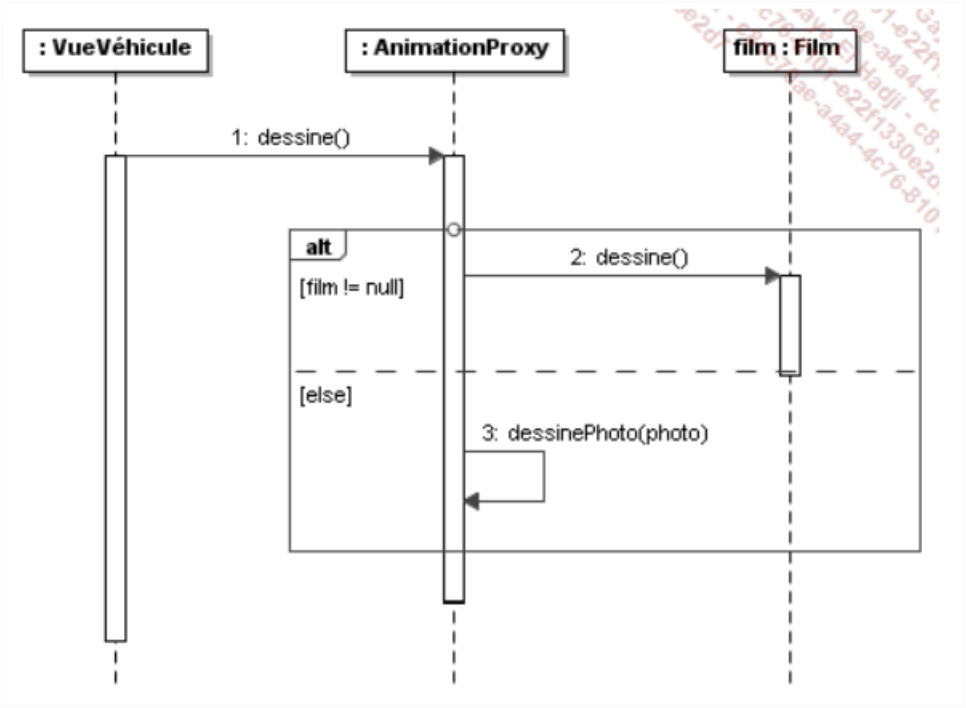
Cette solution apporte deux avantages :

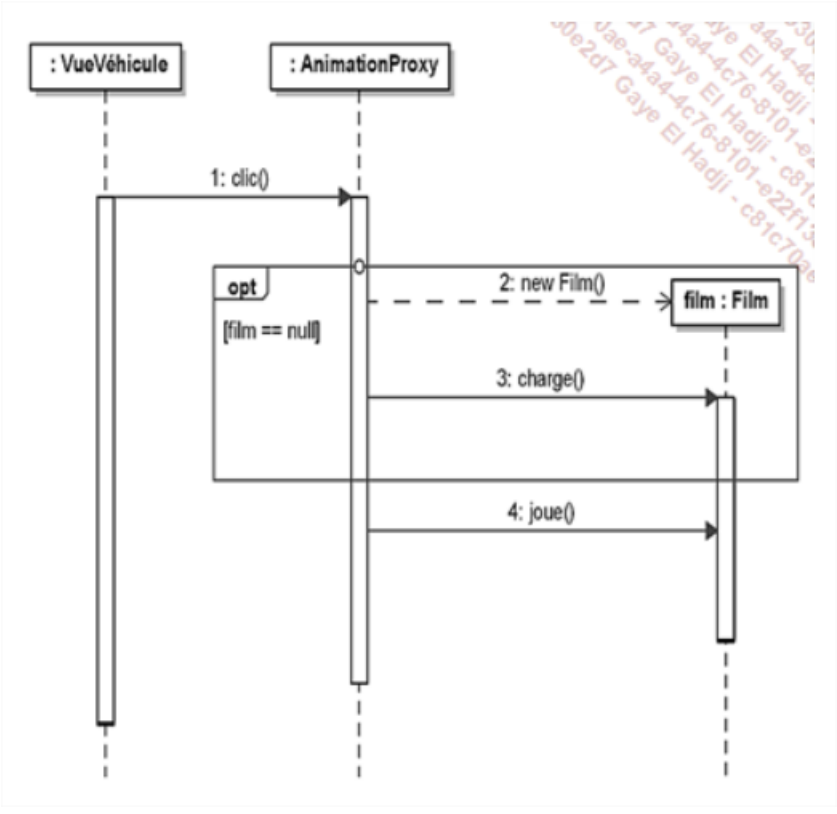
- La page du catalogue est chargée beaucoup plus rapidement surtout si elle doit être chargée au travers d'un réseau comme Internet.
- Seuls les films devant être visualisés sont créés, chargés et joués.

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 implante la même interface que l'objet film. La figure ci-dessous 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.



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 première figure ci-dessous et à la première figure ci-dessous pour le message clic.





Veuillez réaliser cet exemple concrètement en utilisant les fichiers en annexe.

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.

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.

m. Chain of Responsibility

Description :

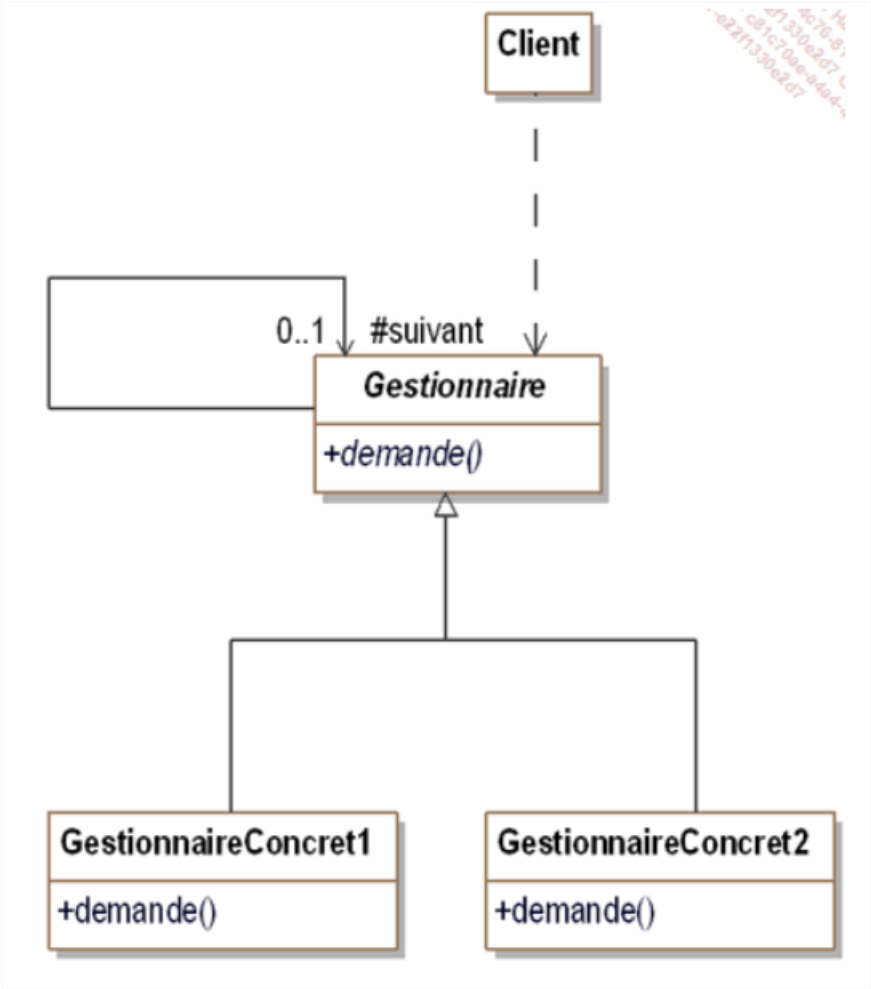
Le pattern **Chain of Responsibility** construit une chaîne d'objets telle que si un objet de la chaîne ne peut pas répondre à une requête, il puisse la transmettre à son successeur et ainsi de suite jusqu'à ce que l'un des objets de la chaîne y réponde.

Plus de détails :

Le patron Chaîne de responsabilité (en anglais chain of responsibility) vise à découpler l'émission d'une requête de la réception et le traitement de cette dernière en permettant à plusieurs objets de la traiter successivement. Dans ce patron chaque objet comporte un lien vers l'objet suivant, qui est du même type. Plusieurs objets sont ainsi attachés et forment une chaîne. Lorsqu'une demande est faite au premier objet de la chaîne, celui-ci tente de la traiter, et s'il ne peut pas il fait appel à l'objet suivant, et ainsi de suite16.

Structure

Diagramme de classes



Domaines d'utilisation

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.

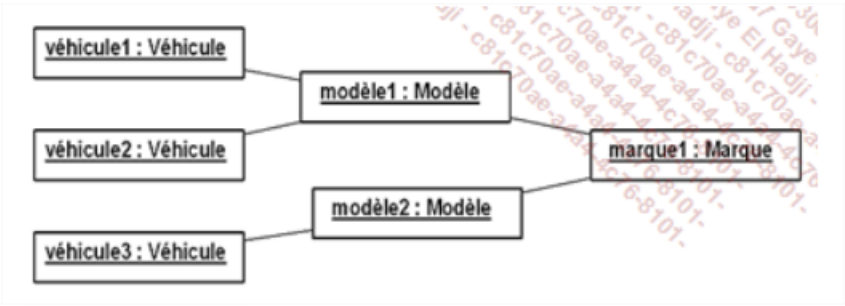
Exemple d'implémentation :

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.

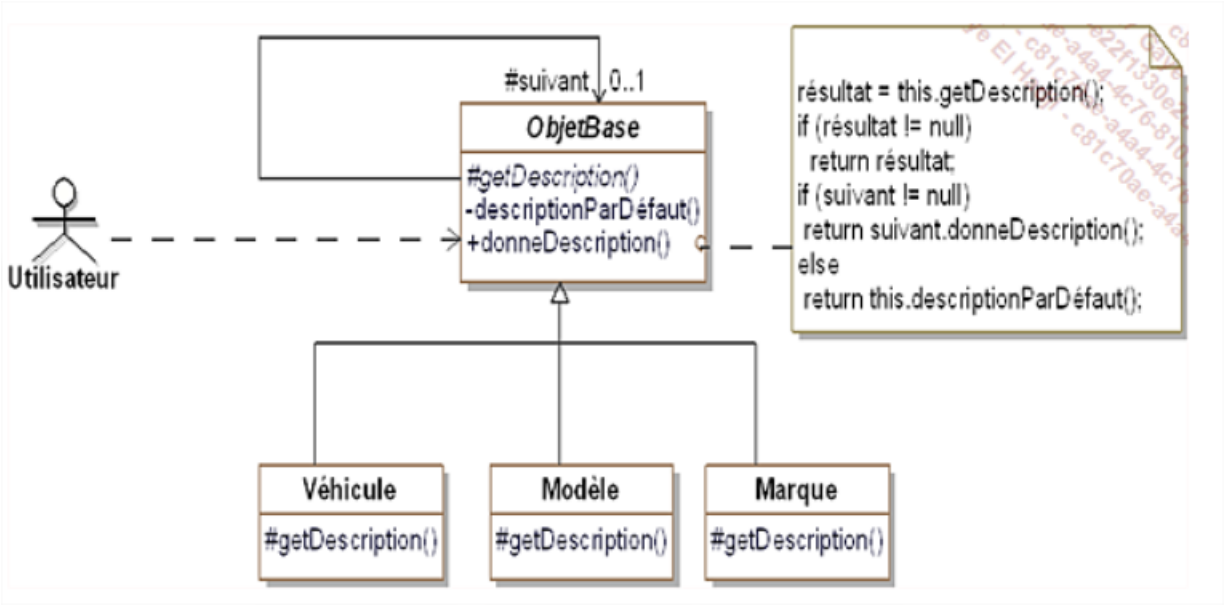
Le pattern Chain of Responsibility fournit une solution pour mettre en œuvre ce mécanisme. Celle-ci consiste à lier les objets entre eux du plus spécifique (le véhicule) au plus général (la marque) pour former la chaîne de responsabilité. La requête de description est transmise le long de cette chaîne jusqu'à ce qu'un objet puisse la traiter et renvoyer la description.

Le diagramme d'objets UML de la première figure ci-dessous illustre cette situation et montre les différentes chaînes de responsabilité (de la gauche vers la droite).



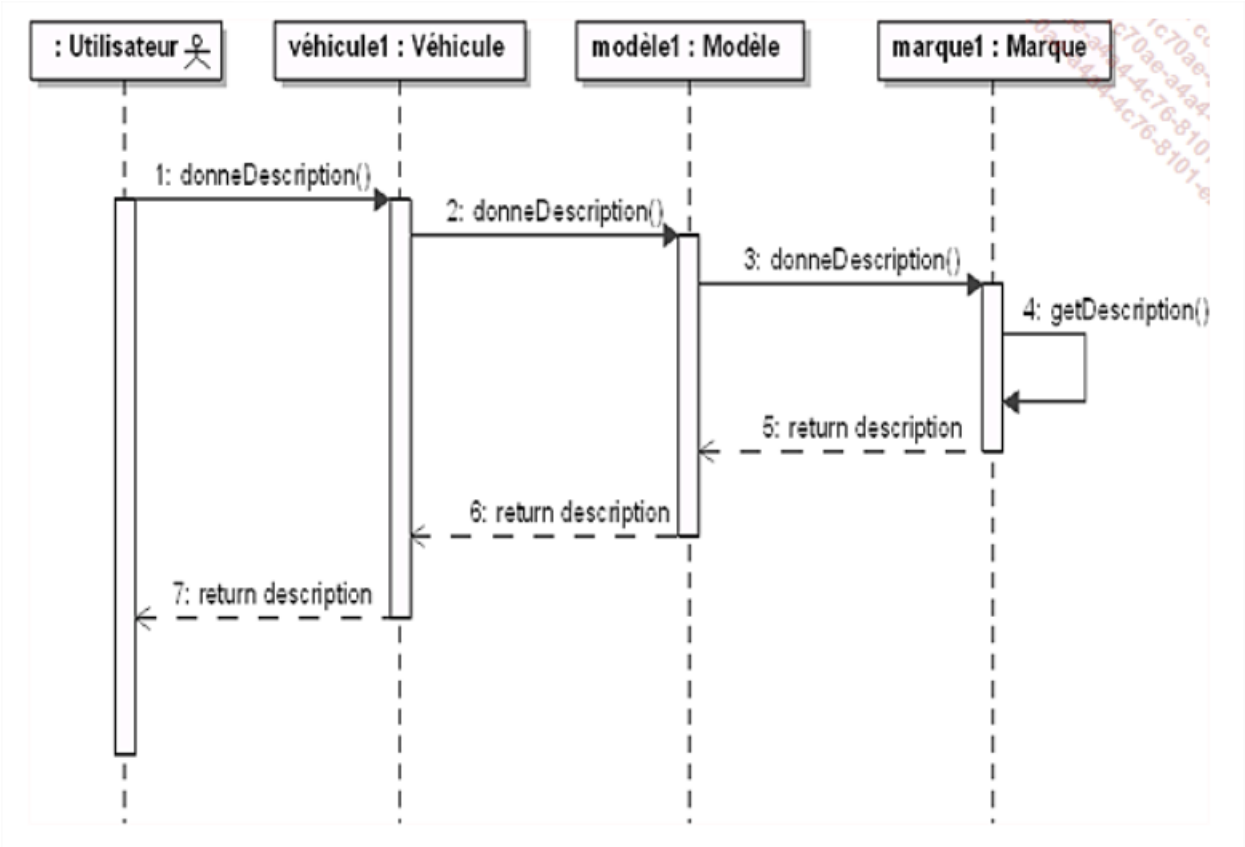
La première figure ci-dessous représente le diagramme des classes du pattern Chain of Responsibility appliqué à l'exemple. Les véhicules, modèles et marques sont décrits par des sous-classes concrètes de la classe ObjetBase. Cette classe abstraite introduit l'association suivant qui implante la chaîne de responsabilité. Elle introduit également trois méthodes :

- getDescription 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.
- donneDescription est la méthode publique destinée à l'utilisateur. Elle invoque la méthode getDescription. Si le résultat est null, alors s'il y a un objet suivant, sa méthode donneDescription est invoquée à son tour sinon c'est la méthode descriptionParDéfaut qui est utilisée.



La figure ci-dessous montre un diagramme de séquence qui est un exemple de requête d'une description basée sur le diagramme d'objets de la première figure ci-dessus.

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.



Veuillez réaliser cet exemple concrètement en utilisant les fichiers en annexe.

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.

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.

n. Command

Description :

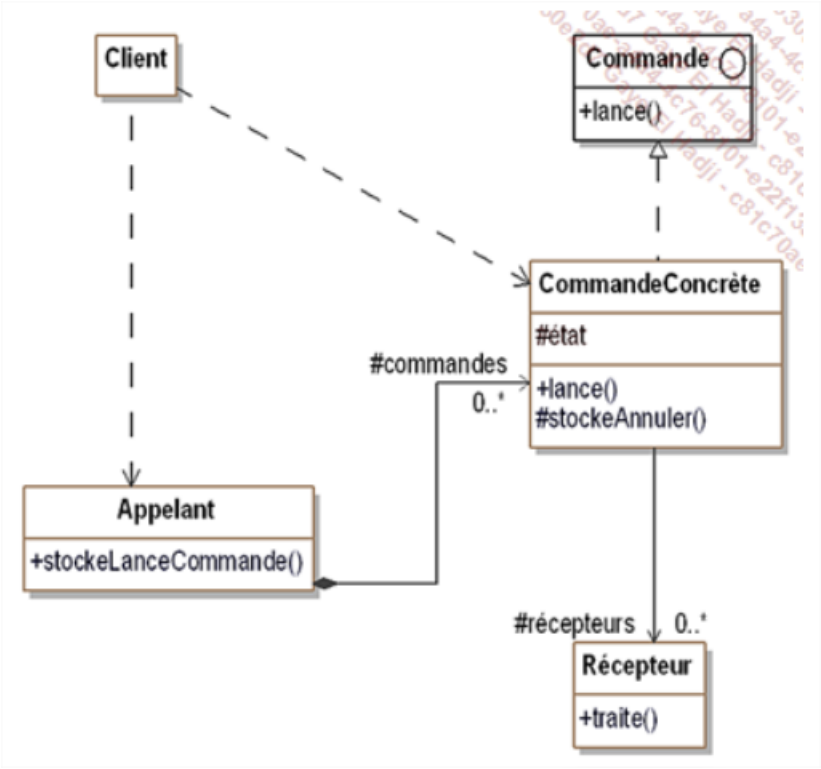
Le pattern **Command** a pour objectif de transformer une requête en un objet, facilitant des opérations comme l'annulation, la mise en file des requêtes et leur suivi.

Plus de détails :

Ce patron emboîte une demande dans un objet, permettant de paramétrer, mettre en file d'attente, journaliser et annuler des demandes. Dans ce patron un objet commande (en anglais command) correspond à une opération à effectuer. L'interface de cet objet comporte une méthode execute. Pour chaque opération, l'application va créer un objet différent qui implémente cette interface - qui comporte une méthode execute. L'opération est lancée lorsque la méthode execute est utilisée. Ce patron est notamment utilisé pour les barres d'outils.

Structure

Diagramme de classes



Domaines d'utilisation

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 d'implémentation :

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.

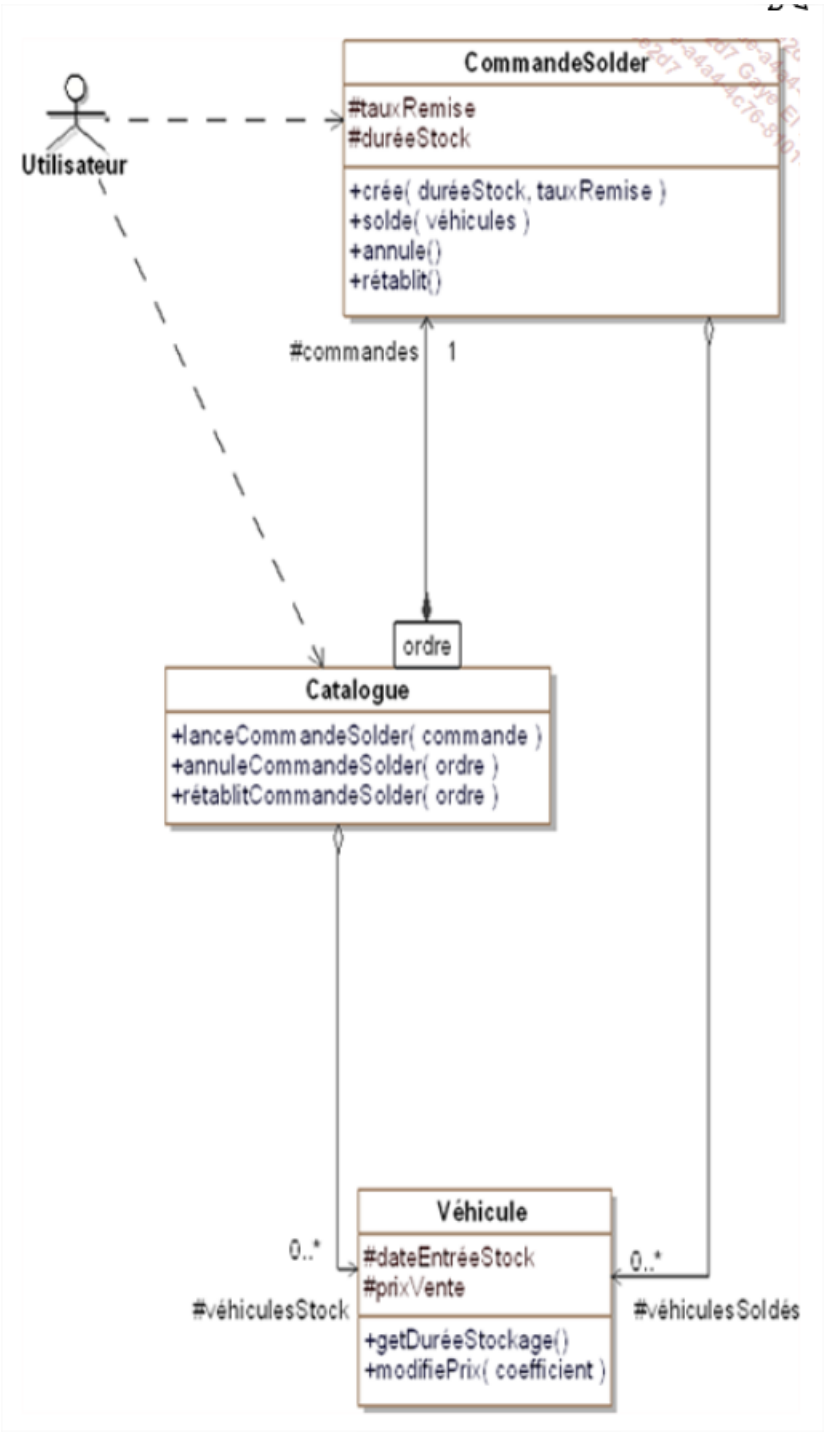
Pour gérer cette annulation, une première solution consiste à indiquer au niveau de chaque véhicule s'il est ou non soldé. Cette solution n'est pas suffisante car un même véhicule peut être soldé plusieurs fois avec des taux différents. Une autre solution serait alors de conserver son prix avant la dernière 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 ci-dessous illustre cette application du pattern Command à notre exemple. La classe CommandeSolder stocke ses deux paramètres (tauxRemise et duréeStock) ainsi que la liste des véhicules pour lesquels la remise a été appliquée (association véhiculesSoldés).

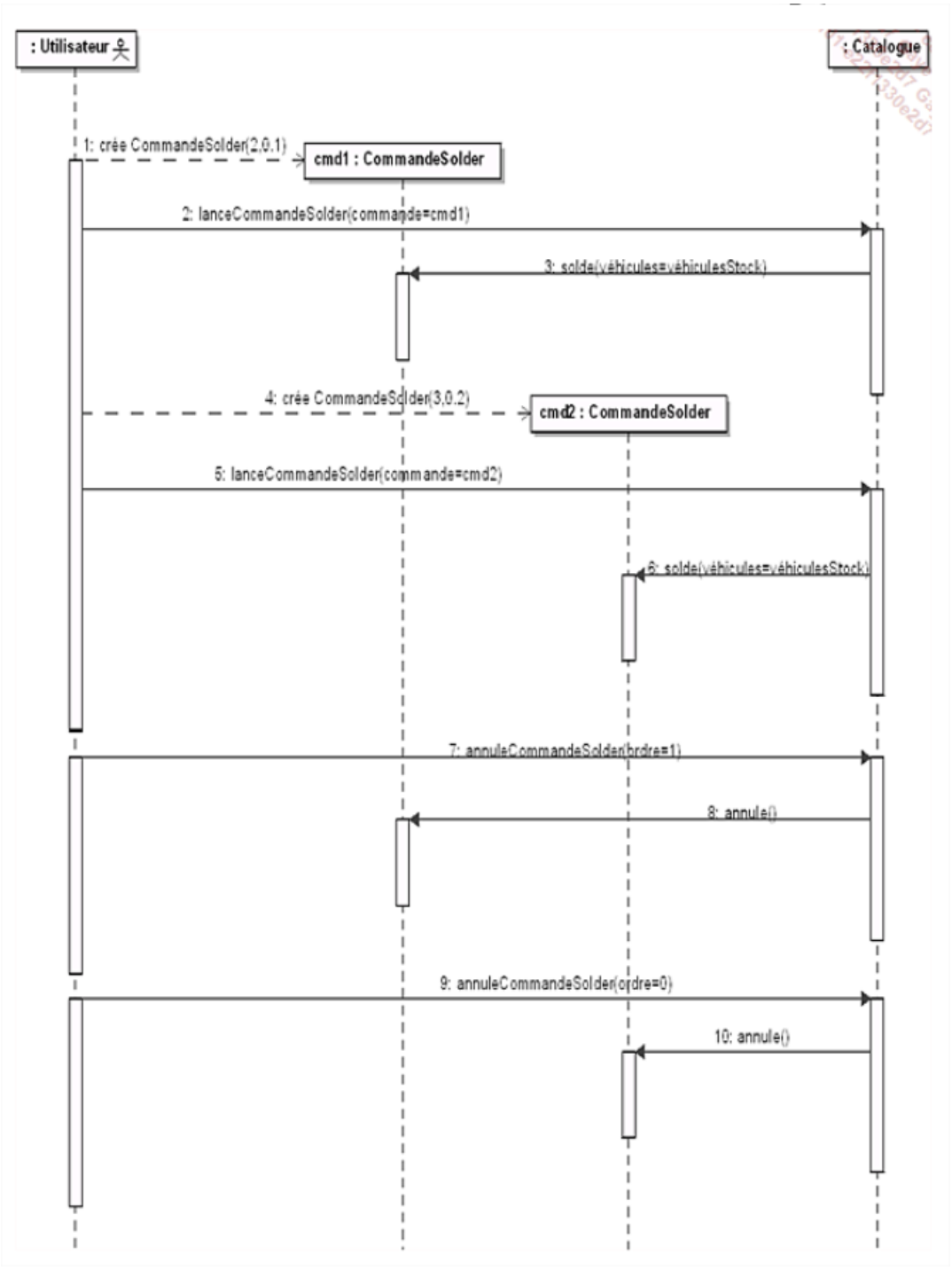
Il faut noter que l'ensemble des véhicules référencés par CommandeSolder est un sous-ensemble de l'ensemble des véhicules référencés par Catalogue.

Lors de l'appel de la méthode lanceCommandeSolder, 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.



Le diagramme de la figure ci-dessous 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 PHP présenté plus loin.



Veuillez réaliser cet exemple concrètement en utilisant les fichiers en annexe.

Participants

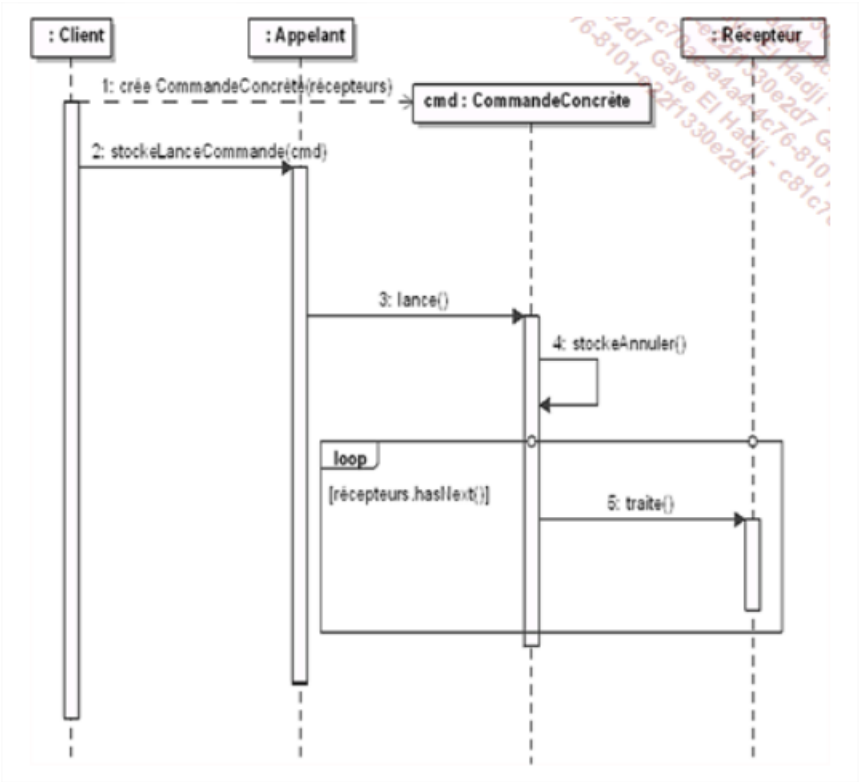
Les participants au pattern sont les suivants :

- 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.

Collaborations

La figure ci-dessus 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.



o. Interpreter

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.

Plus de détails :

Le patron comporte deux composants centraux : le contexte et l'expression ainsi que des objets qui sont des représentations d'éléments de grammaire d'un langage de programmation. Le patron est utilisé pour transformer une expression écrite dans un certain langage de programmation - un texte source - en quelque chose de manipulable par programmation : Le code source est écrit conformément à une ou plusieurs règles de grammaire, et un objet est créé pour chaque utilisation d'une règle de grammaire. L'objet interpreter est responsable de transformer le texte source en objets.

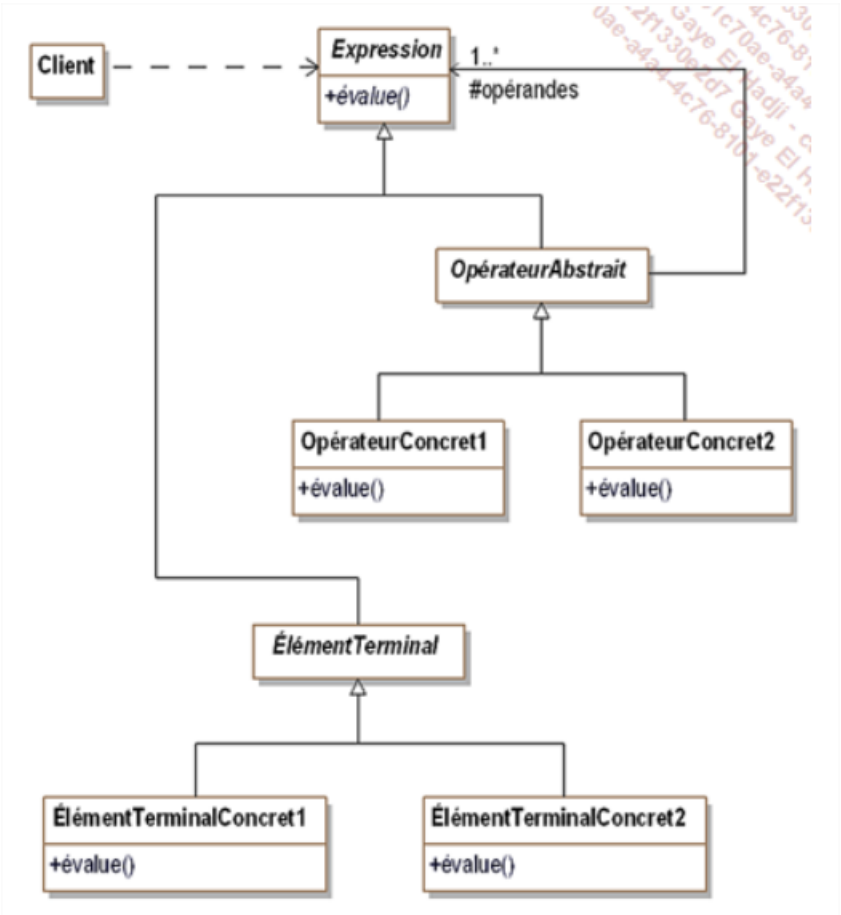
Structure

Diagramme de classes

La figure ci-dessous 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'arguments comme des fonctions.



Domaines d'utilisation

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 d'implémentation :

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 :

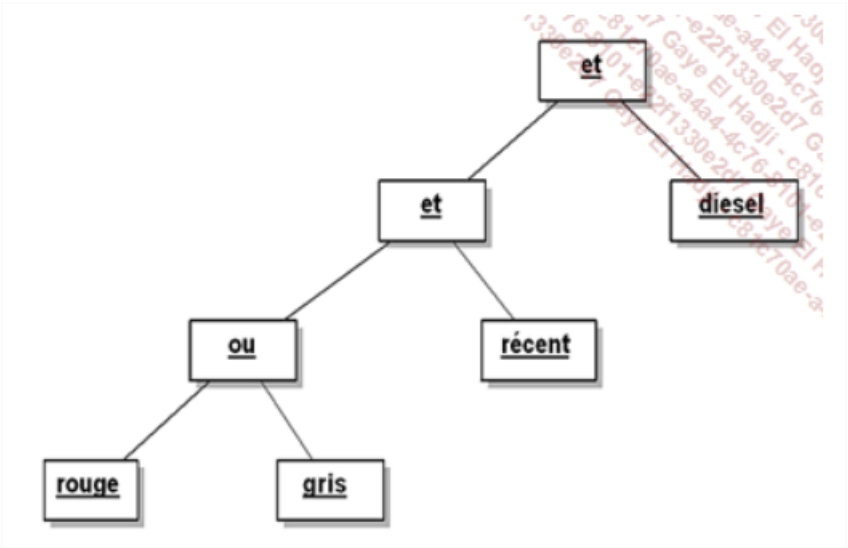
expression ::= terme || mot-clé || (expression)
terme ::= facteur 'ou' facteur
facteur ::= expression 'et' expression
mot-clé ::= 'a'..'z','A'..'Z' {'a'..'z','A'..'Z'}*

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 traduite par l'arbre syntaxique de la figure ci-dessous.

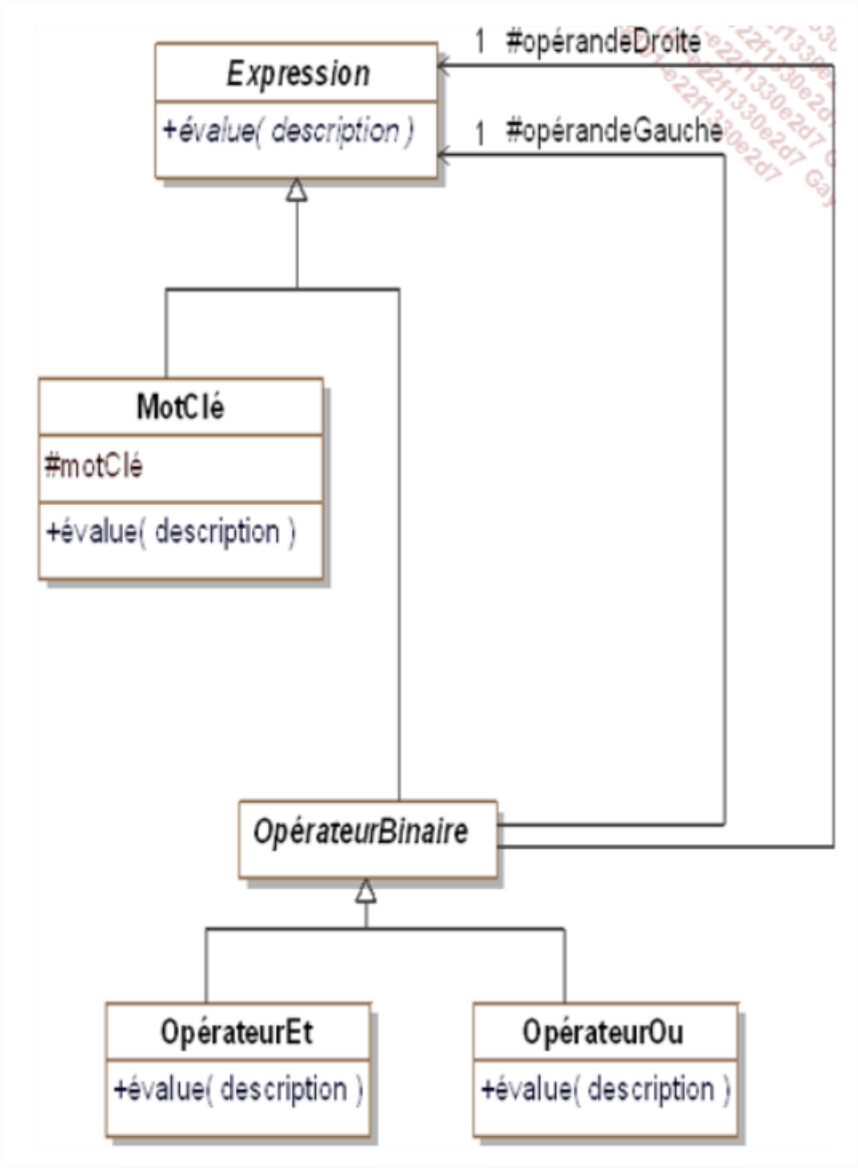


L'évaluation d'un tel arbre pour la description d'un véhicule se fait en commençant par le sommet. Quand un nœud est un opérateur, l'évaluation se fait en calculant récursivement la valeur de chaque sous-arbre (celui de gauche puis celui de droite) et en appliquant l'opérateur. Quand un nœud est un mot-clé, 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 ci-dessus est représenté à la figure ci-dessous. La méthode évalue permet d'évaluer l'expression pour une description d'un véhicule fournie en paramètre.



Veuillez réaliser cet exemple concrètement en utilisant les fichiers en annexe.

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.

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.

p. Iterator

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.

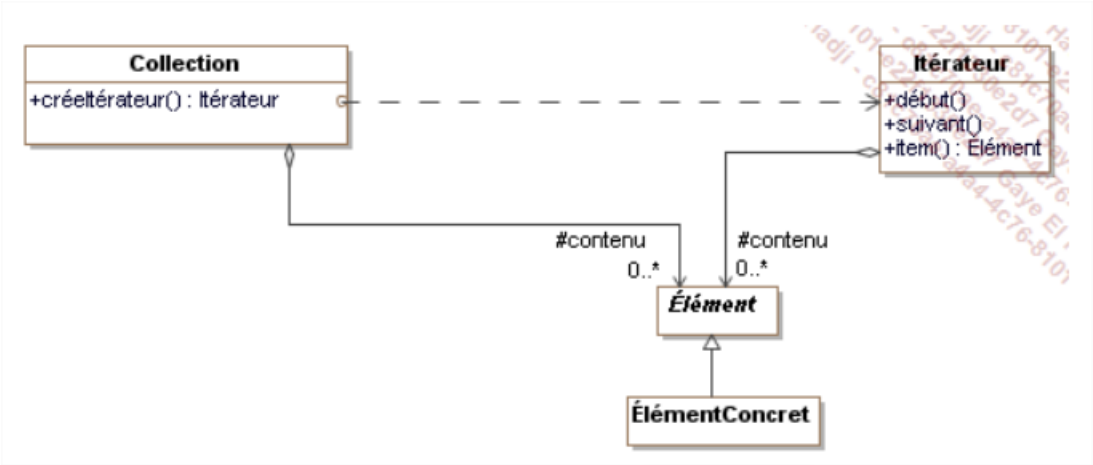
Plus de détails :

Ce patron permet d'accéder séquentiellement aux éléments d'un ensemble sans connaitre les détails techniques du fonctionnement de l'ensemble. C'est un des patrons les plus simples et les plus fréquents. Selon la spécification originale, il consiste en une interface qui fournit les méthodes Next et Current. L'interface en Java comporte généralement une méthode nextElement et une méthode hasMoreElements.

Structure

Diagramme de classes

La figure ci-dessous détaille la structure générique du pattern.



Domaines d'utilisation

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.
- 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 d'implémentation :

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 :

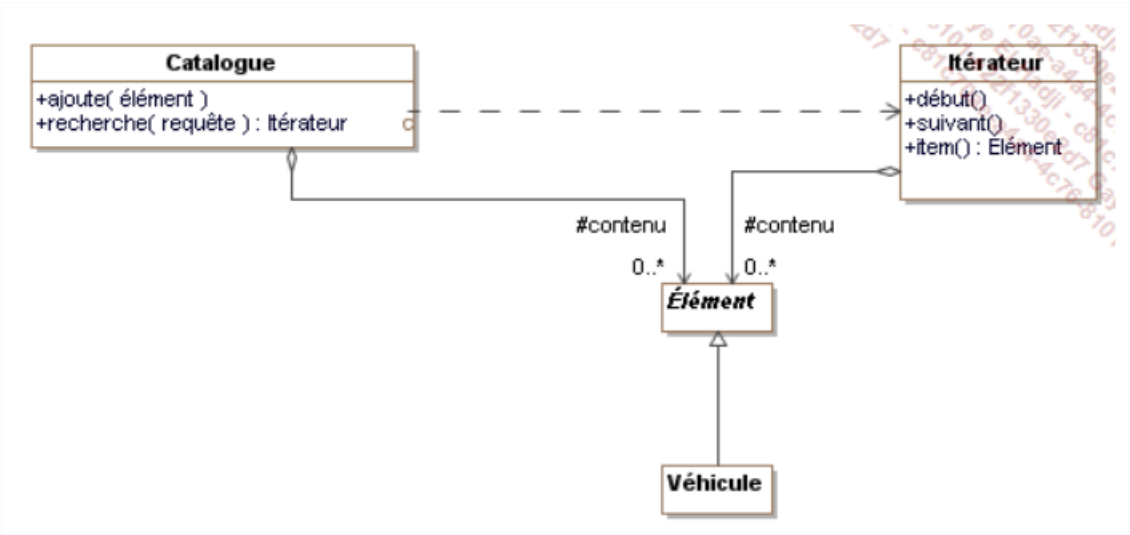
- Elle fait grossir inutilement la classe du catalogue.
- Elle ne permet qu'un seul parcours à la fois, ce qui peut être insuffisant (notamment dans le cas d'applications multitâches).

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.

La classe Itérateur introduit les méthodes début, item et suivant.

La classe Catalogue introduit la méthode recherche qui crée, initialise et retourne une instance de Itérateur.

La figure ci-dessous montre l'utilisation du pattern Iterator pour parcourir les véhicules du catalogue qui répondent à une requête.



Veuillez réaliser cet exemple concrètement en utilisant les fichiers en annexe.

Participants

Les participants au pattern sont les suivants :

- Itérateur est la classe qui implante l'association de l'itérateur avec les éléments de la collection ainsi que les méthodes.
- Collection (Catalogue) est la classe qui implante l'association de la collection avec les éléments et la méthode créeItérateur.
- Élémentest la classe abstraite des éléments de la collection.
- ÉlémentConcret (Véhicule) est une sous-classe concrète de Élément utilisée par Itérateur et Collection.

Collaborations

L'itérateur garde en mémoire l'objet courant dans la collection. Il est capable de calculer l'objet suivant du parcours.

q. Mediator

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.

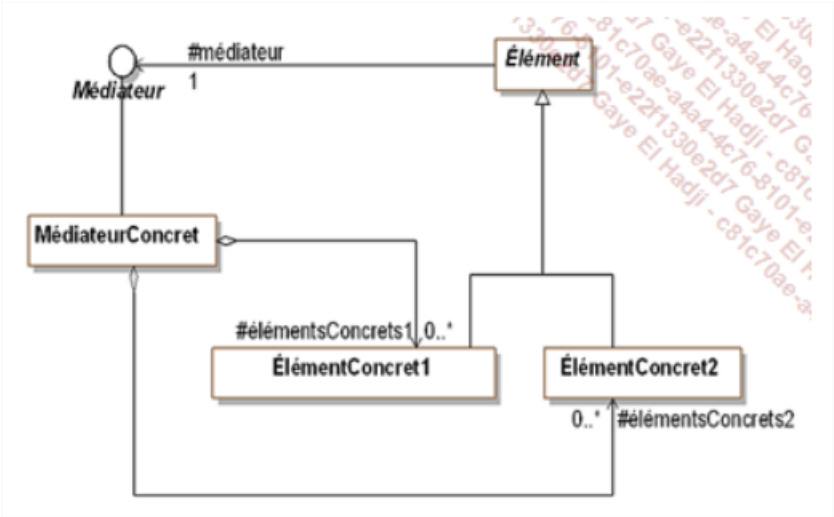
Plus de détails :

Dans ce patron il y a un objet qui définit comment plusieurs objets communiquent entre eux en évitant à chacun de faire référence à ses interlocuteurs. Ce patron est utilisé quand il y a un nombre non négligeable de composants et de relations entre les composants. Par exemple dans un réseau de 5 composants il peut y avoir jusqu'à 20 relations (chaque composant vers 4 autres). Un composant médiateur est placé au milieu du réseau et le nombre de relations est diminué : chaque composant est relié uniquement au médiateur. Le mediator joue un rôle similaire à un sujet dans le patron observer et sert d'intermédiaire pour assurer les communications entre les objets.

Structure

Diagramme de classes

La figure ci-dessous détaille la structure générique du pattern.



Domaines d'utilisation

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.

Exemple d'implémentation :

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.

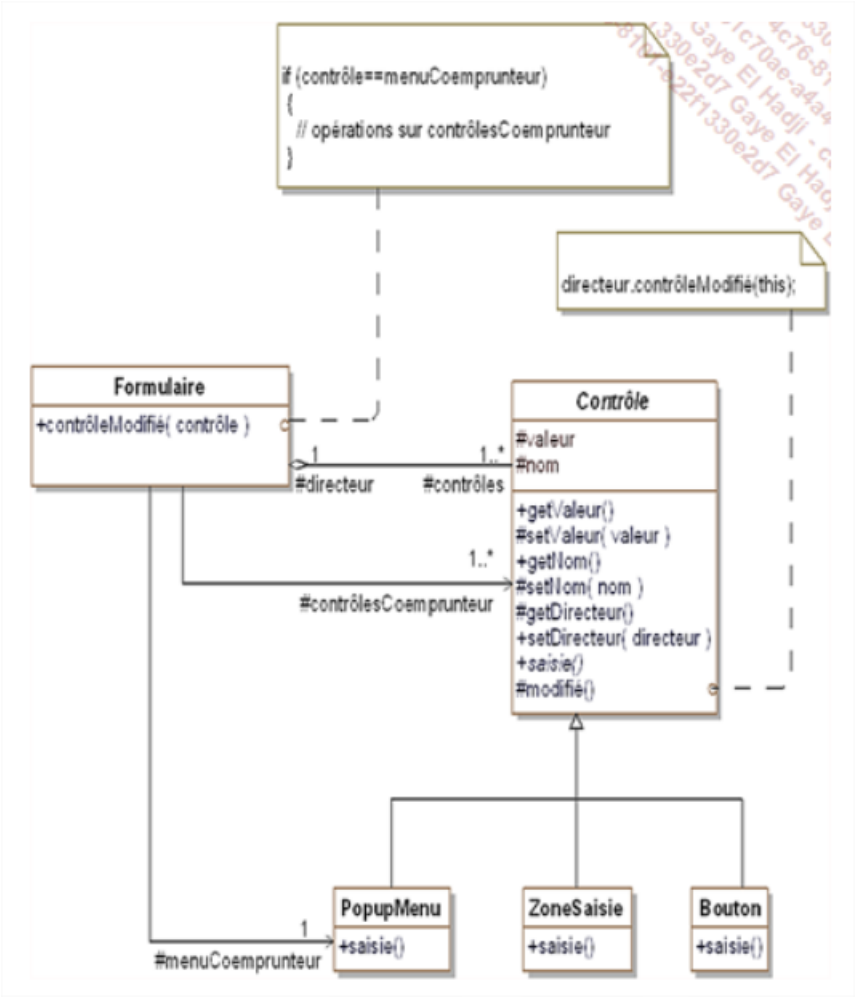
La première possibilité est donc de lier chaque contrôle aux contrôles dont l'aspect change en fonction de sa valeur. Cette possibilité présente les inconvénients cités ci-dessus.

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 coemprunteur. Ce choix se fait à l'aide d'un menu. Si le choix est d'emprunter avec un coemprunteur, tout un ensemble de contrôles graphiques relatifs aux données du coemprunteur doivent apparaître et être saisis.

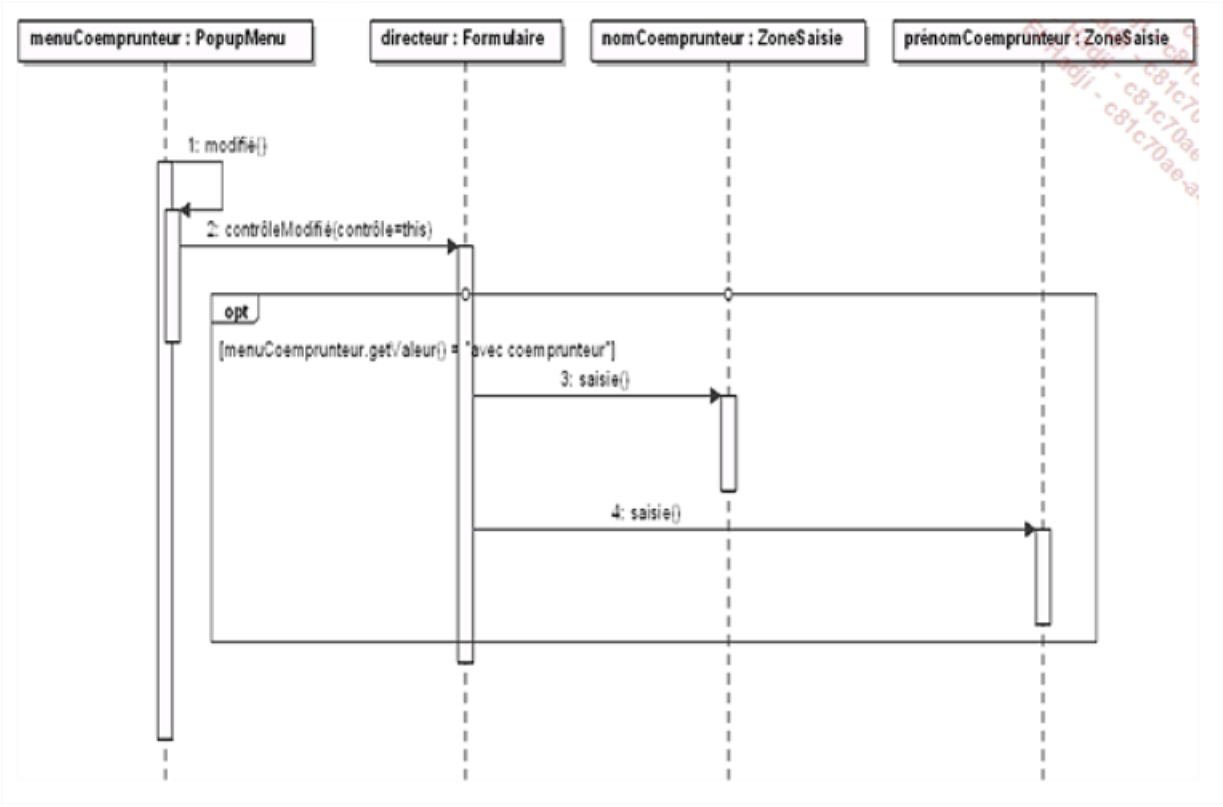
La figure ci-dessous illustre le diagramme des classes correspondant. Ce diagramme introduit les classes suivantes.

- 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 la méthode saisie.
- Formulaire est la classe qui fait office de médiateur. Elle reçoit les notifications de changement des contrôles par invocation de la méthode contrôleModifié.



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 ci-dessous illustre ce fonctionnement de façon partielle sur l'exemple. Quand la valeur du contrôle menuCoemprunteur change, la saisie du nom et du prénom du coemprunteur est effectuée.



Veuillez réaliser cet exemple concrètement en utilisant les fichiers en annexe.

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.

Collaborations

Les éléments envoient des messages au médiateur et en reçoivent. Le médiateur implante la collaboration et la coordination entre les éléments.

r. Memento

Description :

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

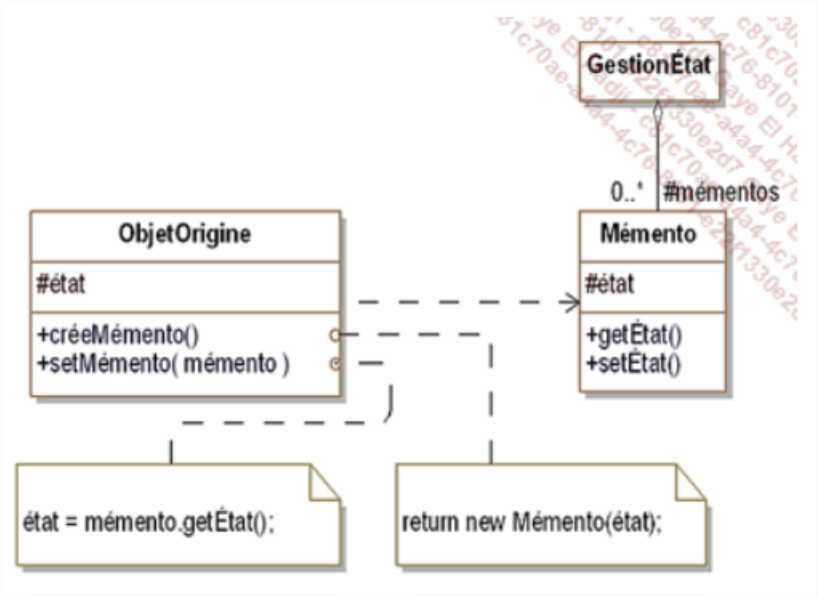
Plus de détails :

Ce patron vise à externaliser l'état interne d'un objet sans perte d'encapsulation. Permet de remettre l'objet dans l'état où il était auparavant. Ce patron permet de stocker l'état interne d'un objet sans que cet état ne soit rendu public par une interface. Il est composé de trois classes : l´origine - d'où l'état provient, le memento - l'état de l'objet d'origine, et le gardien qui est l'objet qui manipulera le memento. L´origine comporte une méthode pour manipuler les memento. Le gardien est responsable de stocker les memento et de les renvoyer à leur origine. Ce patron ne définit pas d'interface précise pour les différents objets, qui sont cependant toujours au nombre de trois.

Structure

Diagramme de classes

La figure ci-dessous détaille la structure générique du pattern.



Domaines d'utilisation

- 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 d'implémentation :

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.

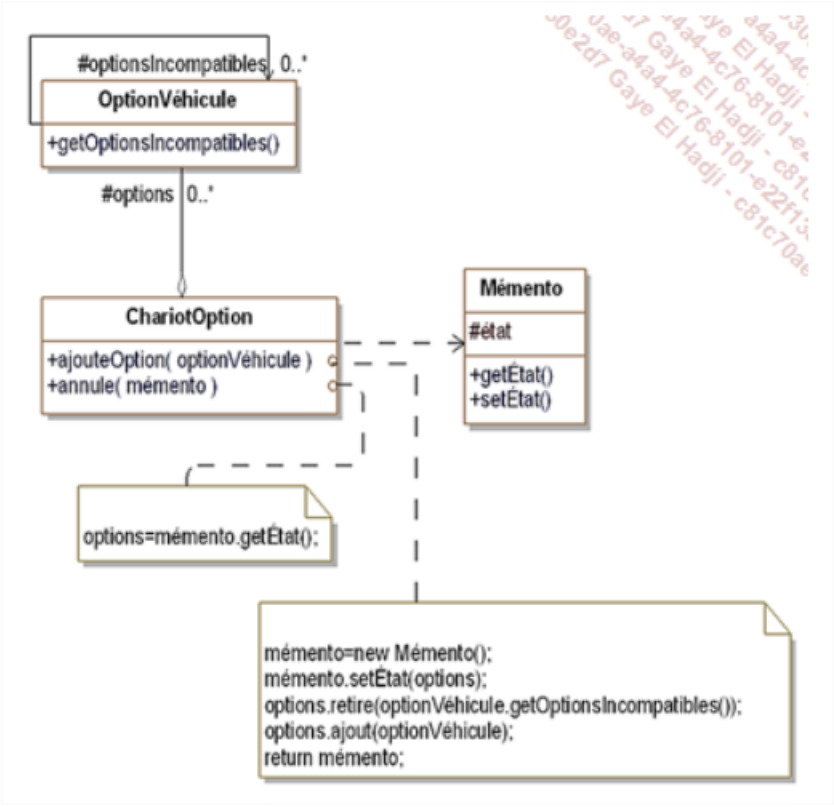
Pour préserver l'encapsulation de l'objet représentant le chariot, une solution consisterait à mémoriser ces états intermédiaires dans le chariot. Cependant cette solution aurait pour effet de complexifier inutilement cet objet.

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 ci-dessous. 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.



Veuillez réaliser cet exemple concrètement en utilisant les fichiers en annexe.

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.

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.

s. Observer

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.

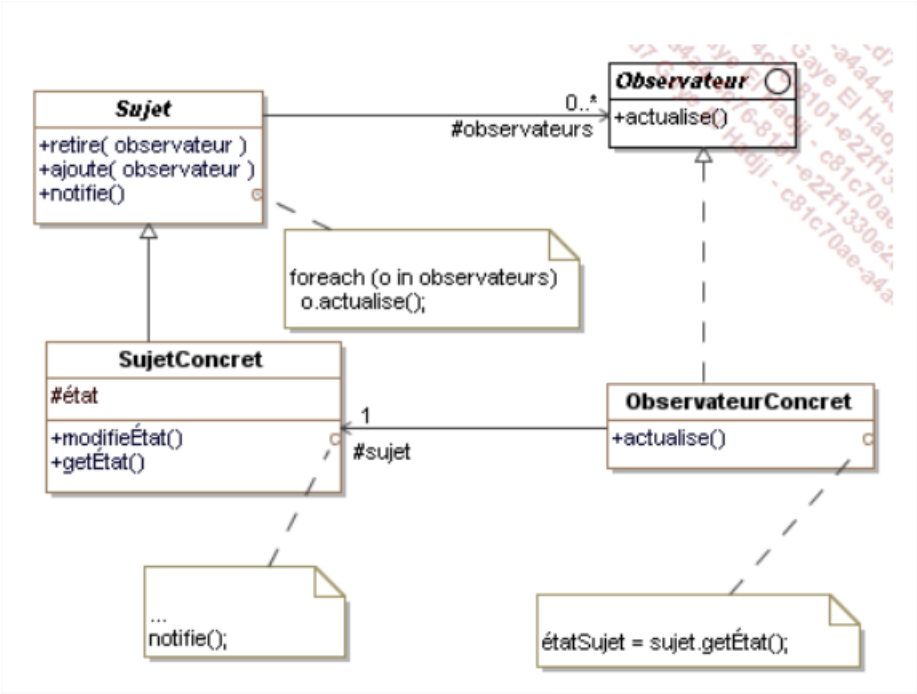
Plus de détails :

Ce patron établit une relation un à plusieurs entre des objets, où lorsqu'un objet change, plusieurs autres objets sont avisés du changement. Dans ce patron, un objet le sujet tient une liste des objets dépendants des observateurs qui seront avertis des modifications apportées au sujet. Quand une modification est apportée, le sujet émet un message aux différents observateurs. Le message peut contenir une description détaillée du changement. Dans ce patron, un objet observer comporte une méthode pour inscrire des observateurs. Chaque observateur comporte une méthode Notify. Lorsqu'un message est émis, l'objet appelle la méthode Notify de chaque observateur inscrit.

Structure

Diagramme de classes

La figure ci-dessous détaille la structure générique du pattern.



Domaines d'utilisation

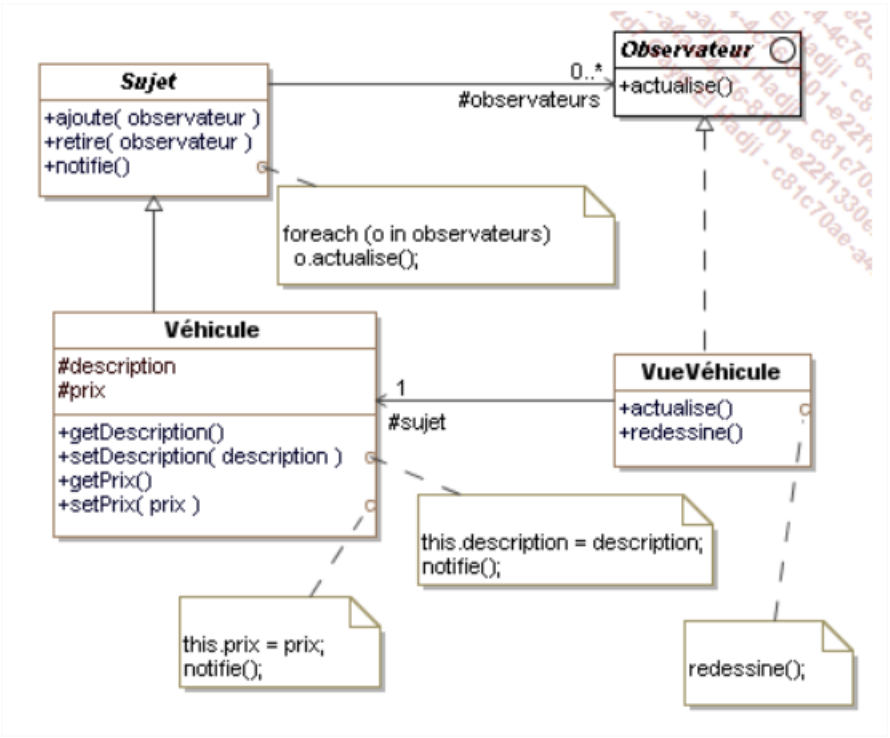
Le pattern est utilisé dans les cas suivants :

- 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.

Exemple d'implémentation :

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.

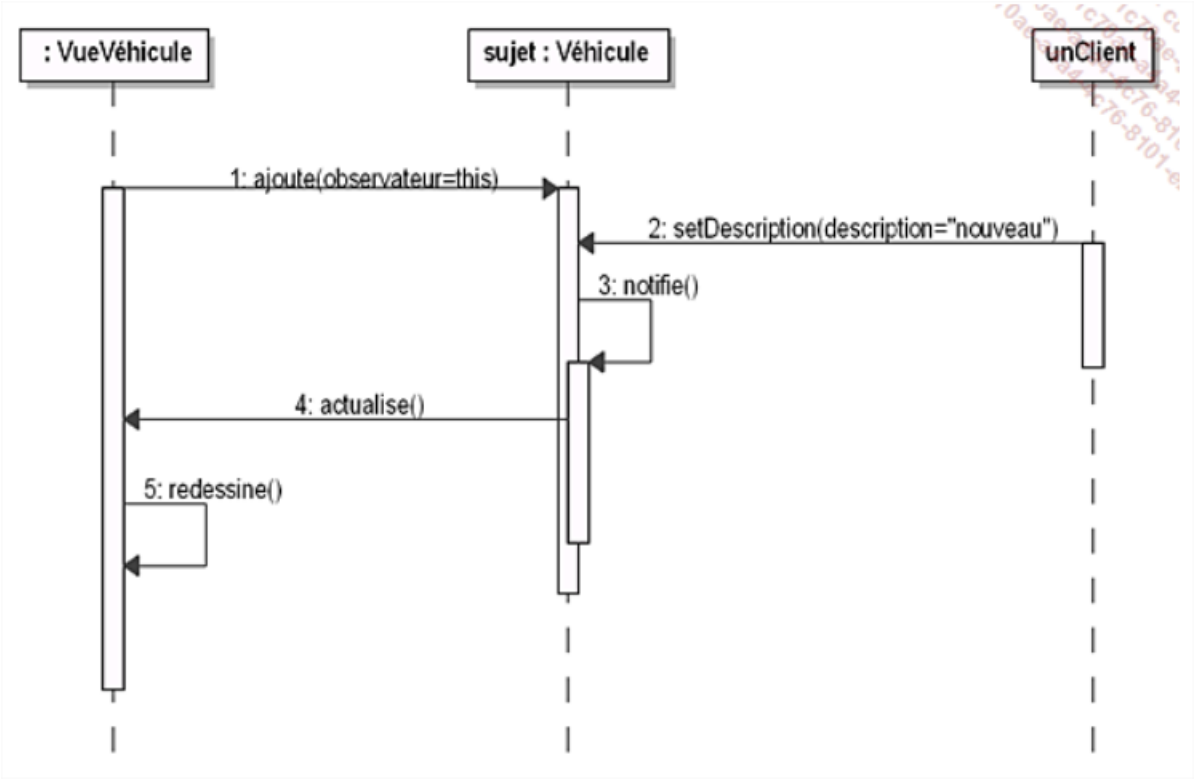
La solution préconisée par le pattern Observer consiste à établir un lien entre chaque véhicule et ses vues pour que le véhicule puisse leur indiquer de se mettre à jour quand son état interne a été modifié. Cette solution est illustrée à la figure ci-dessous.



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 ci-dessous par un diagramme de séquence.



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.

Veuillez réaliser cet exemple concrètement en utilisant les fichiers en annexe.

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.

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.

t. State

Description :

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

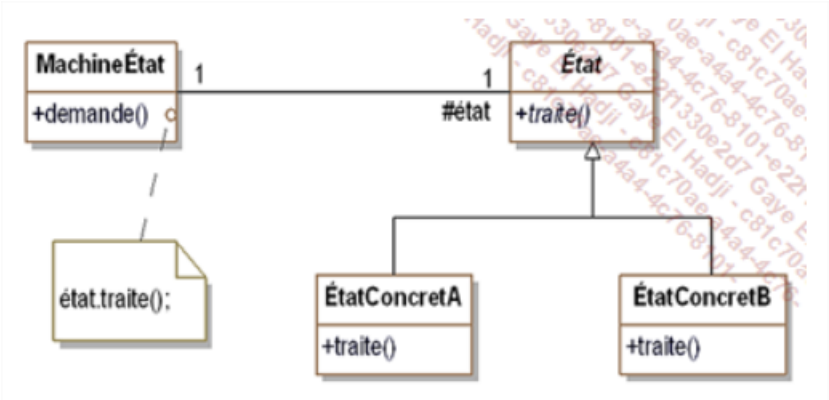
Plus de détails :

Ce patron permet à un objet de modifier son comportement lorsque son état interne change. Ce patron est souvent utilisé pour implémenter une machine à états. Un exemple d'appareil à états est le lecteur audio - dont les états sont lecture, enregistrement, pause et arrêt. Selon ce patron il existe une classe machine à états, et une classe pour chaque état. Lorsqu'un événement provoque le changement d'état, la classe machine à états se relie à un autre état et modifie ainsi son comportement.

Structure

Diagramme de classes

La figure ci-dessous détaille la structure générique du pattern.



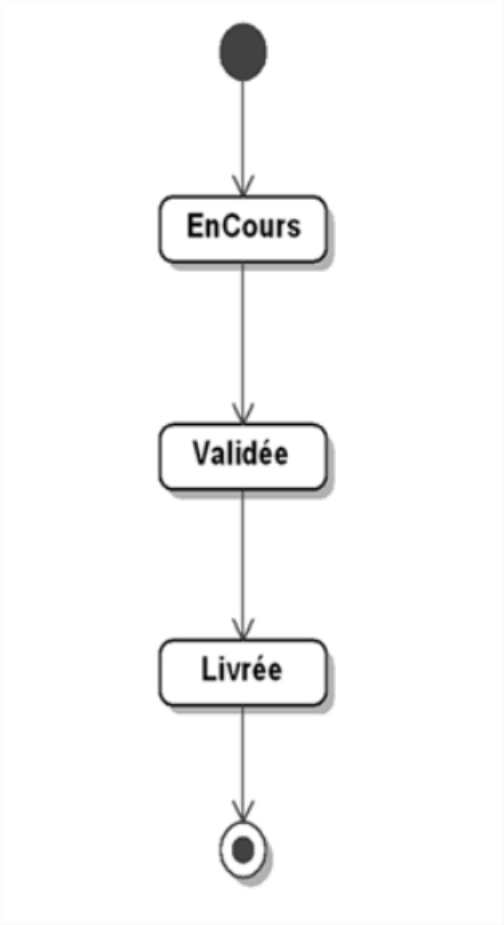
Domaines d'utilisation

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 d'implémentation :

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 ci-dessous. 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.



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.

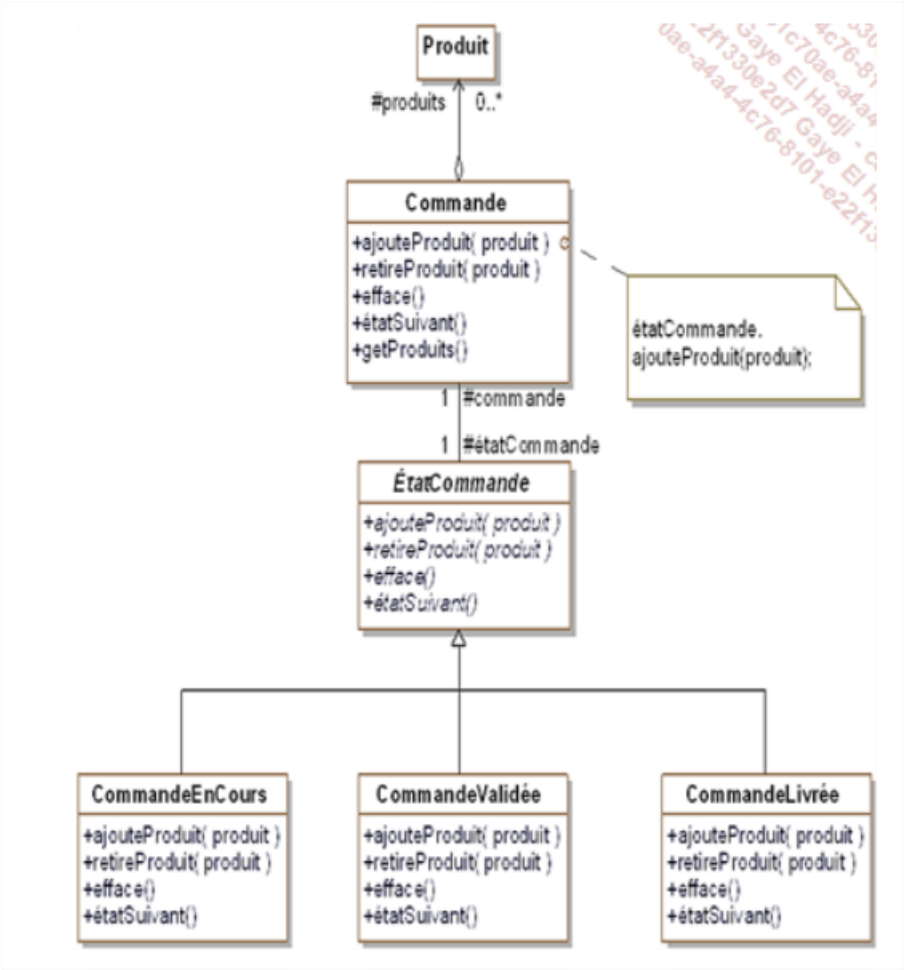
L'approche traditionnelle pour résoudre ces différences de comportement consiste à utiliser des conditions dans le corps des méthodes. Cette approche conduit souvent à des méthodes complexes à écrire et à appréhender.

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 ci-dessous 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 ci-dessous illustre cette délégation.



Veuillez réaliser cet exemple concrètement en utilisant les fichiers en annexe.

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.

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.

u. Stategy

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.

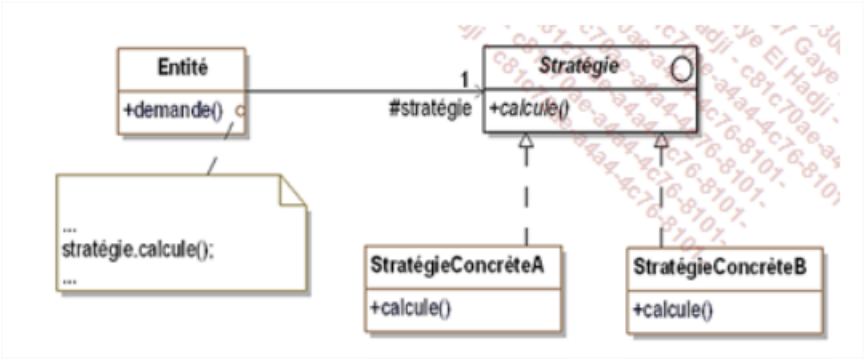
Plus de détails :

Dans ce patron, une famille d'algorithmes est encapsulée de manière qu'ils soient interchangeables. Les algorithmes peuvent changer indépendamment de l'application qui s'en sert. Il comporte trois rôles : le contexte, la stratégie et les implémentations. La stratégie est l'interface commune aux différentes implémentations - typiquement une classe abstraite. Le contexte est l'objet qui va associer un algorithme avec un processus.

Structure

Diagramme de classes

La figure ci-dessous détaille la structure générique du pattern.



Domaines d'utilisation

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.

Exemple d'implémentation :

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.

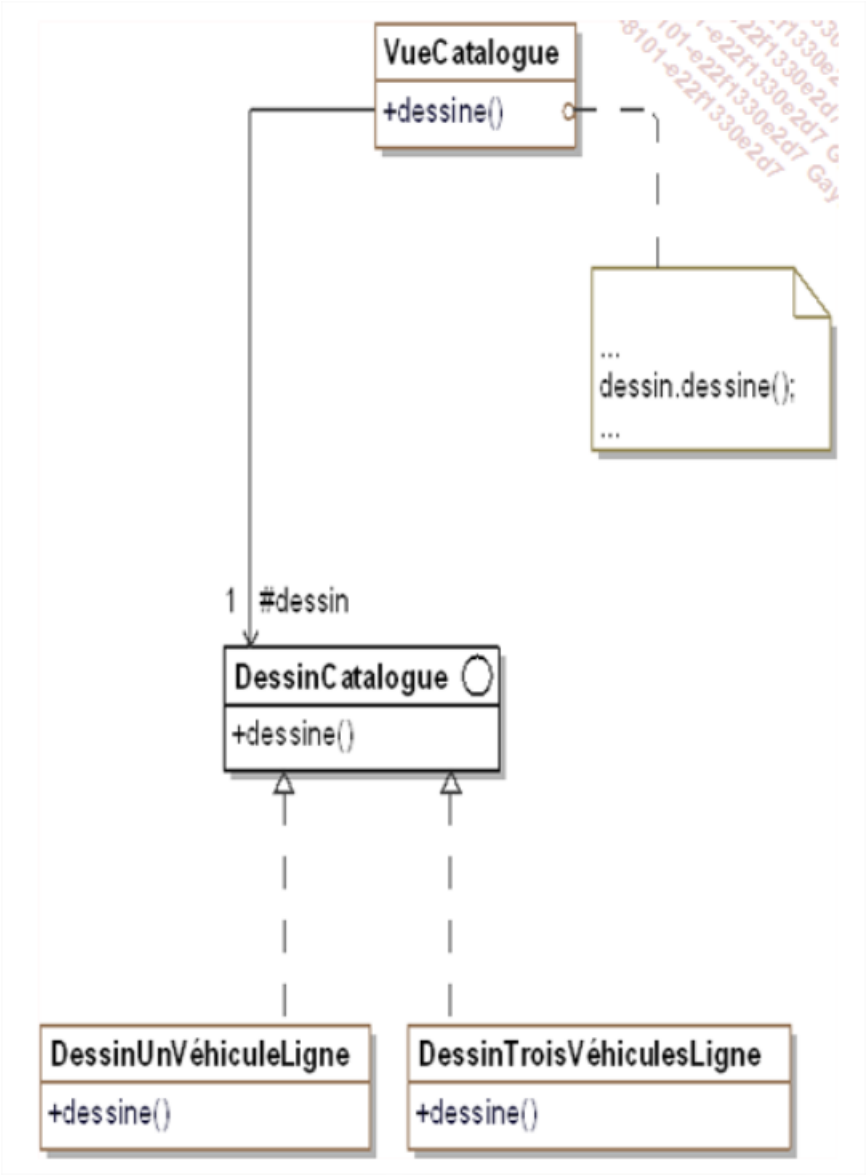
L'interface de la classe VueCatalogue ne dépend pas du choix de l'algorithme de mise en page. Ce choix n'a pas d'impact sur la relation d'une vue de catalogue avec ses clients. Il n'y a que la présentation qui est modifiée.

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 ci-dessous 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. La note détaillant la méthode dessine de la classe VueCatalogue montre comment ces deux classes sont utilisées.



Veuillez réaliser cet exemple concrètement en utilisant les fichiers en annexe.

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.
- Entitéest 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.

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.

v. Visitor

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.

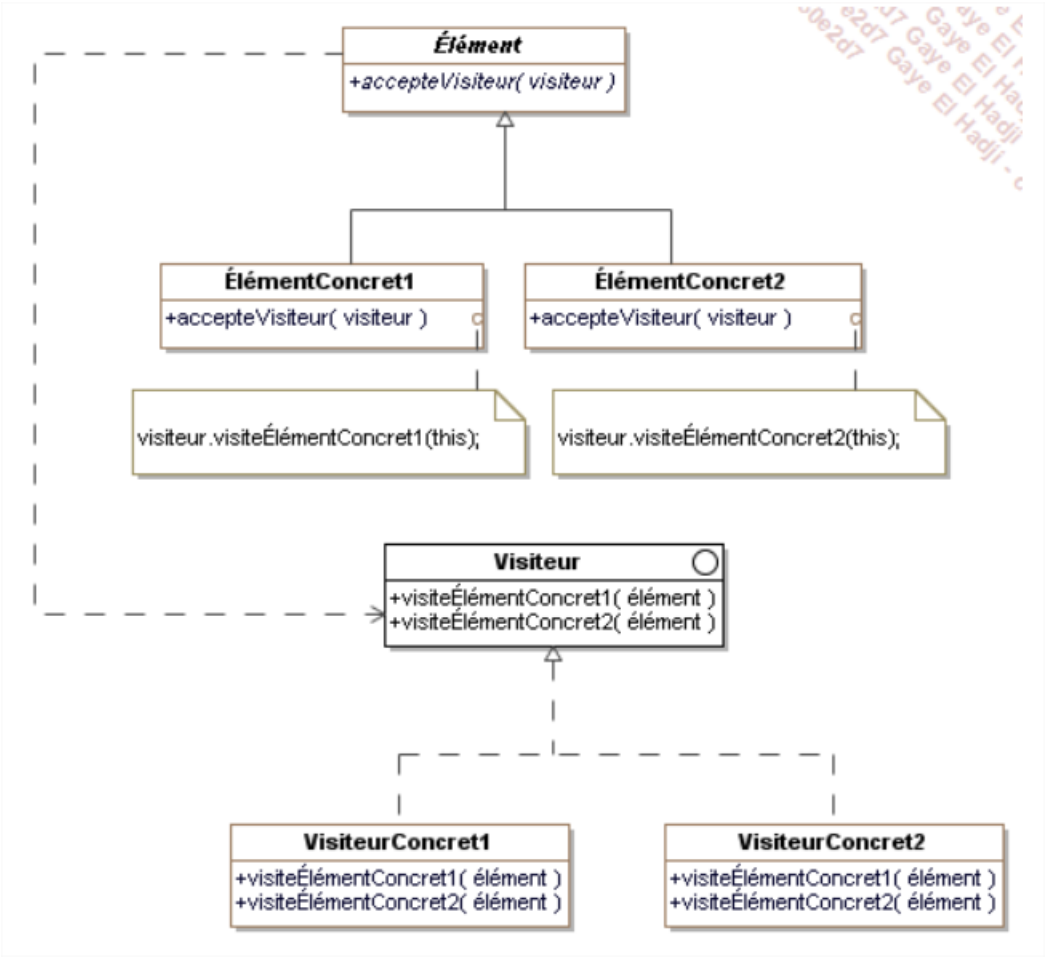
Plus de détails :

Ce patron représente une opération à effectuer sur un ensemble d'objets. Permet de modifier l'opération sans changer l'objet concerné ni la structure. Selon ce patron, les objets à modifier sont passés en paramètre à une classe tierce qui effectuera des modifications. Une classe abstraite Visitor définit l'interface de la classe tierce. Ce patron est utilisé notamment pour manipuler un jeu d'objets, où les objets peuvent avoir différentes interfaces, qui ne peuvent pas être modifiés.

Structure

Diagramme de classes

La figure ci-dessous détaille la structure générique du pattern.



Domaines d'utilisation

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.

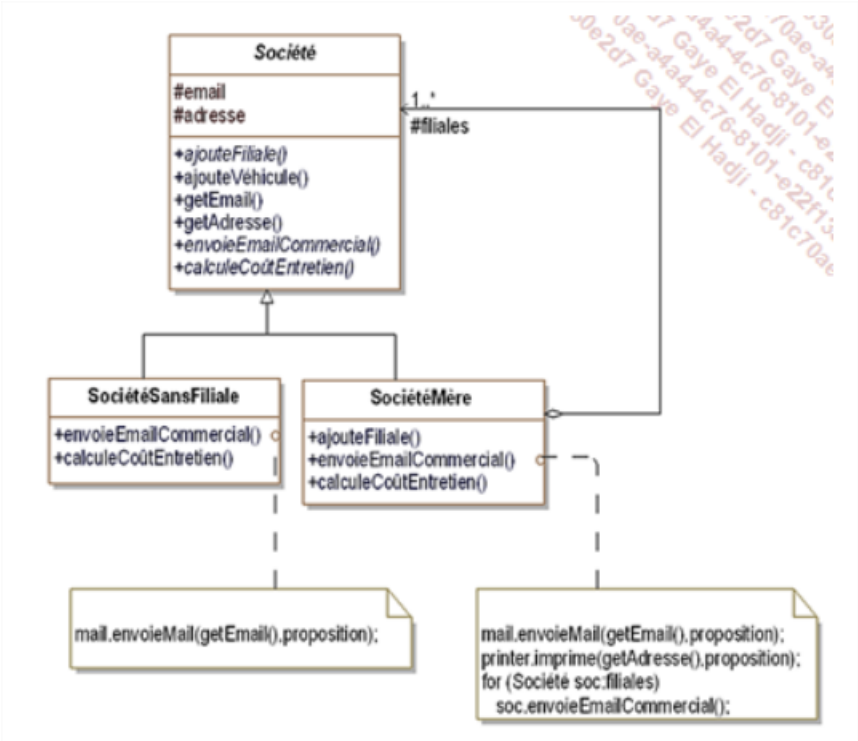
Si la structure de l'ensemble des classes auxquelles il est nécessaire d'adjoindre des fonctionnalités change souvent, le pattern Visitor n'est pas adapté. En effet, toute modification de la structure implique une modification de chaque visiteur, ce qui peut coûter cher.

Exemple d'implémentation :

Considérons la figure ci-dessous qui décrit les clients de notre système organisés sous la forme d'objets composés selon le pattern Composite. À l'exception de la méthode ajouteFiliale 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 ajouteFiliale qui contribue à composer les objets.



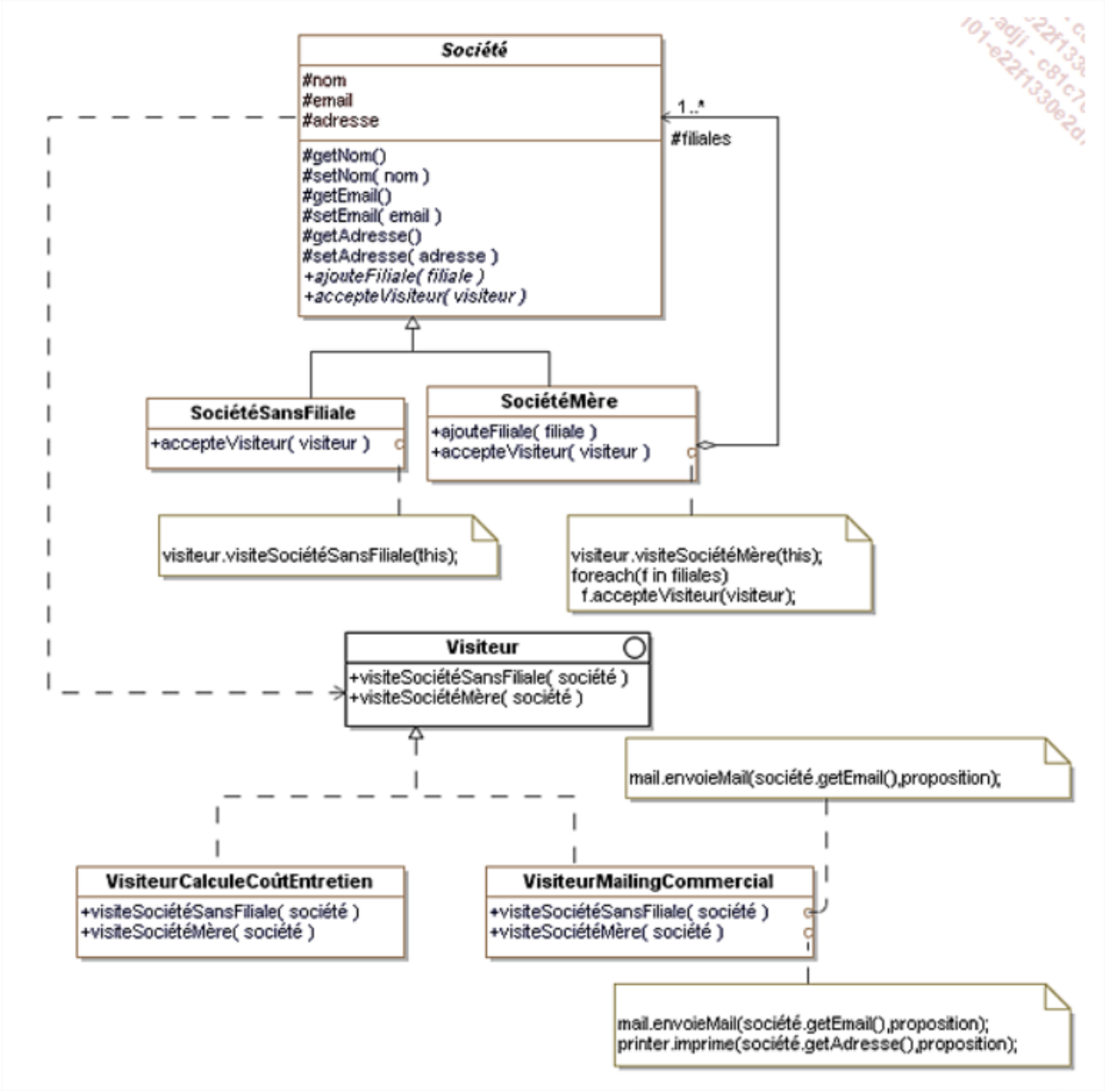
Le pattern Visitor propose d'implanter les nouvelles fonctionnalités dans un objet séparé appelé visiteur. Chaque visiteur établit une fonctionnalité pour plusieurs classes en introduisant pour chacune de ces classes une méthode d'implantation de nom visite suivi du nom de la classe à visiter. Le paramètre de cette méthode est une instance de cette 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 getNom, 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 ci-dessous 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.



Veuillez réaliser cet exemple concrètement en utilisant les fichiers en annexe.

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.
- 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.

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.

4. Patterns d'architecture

a. DAO

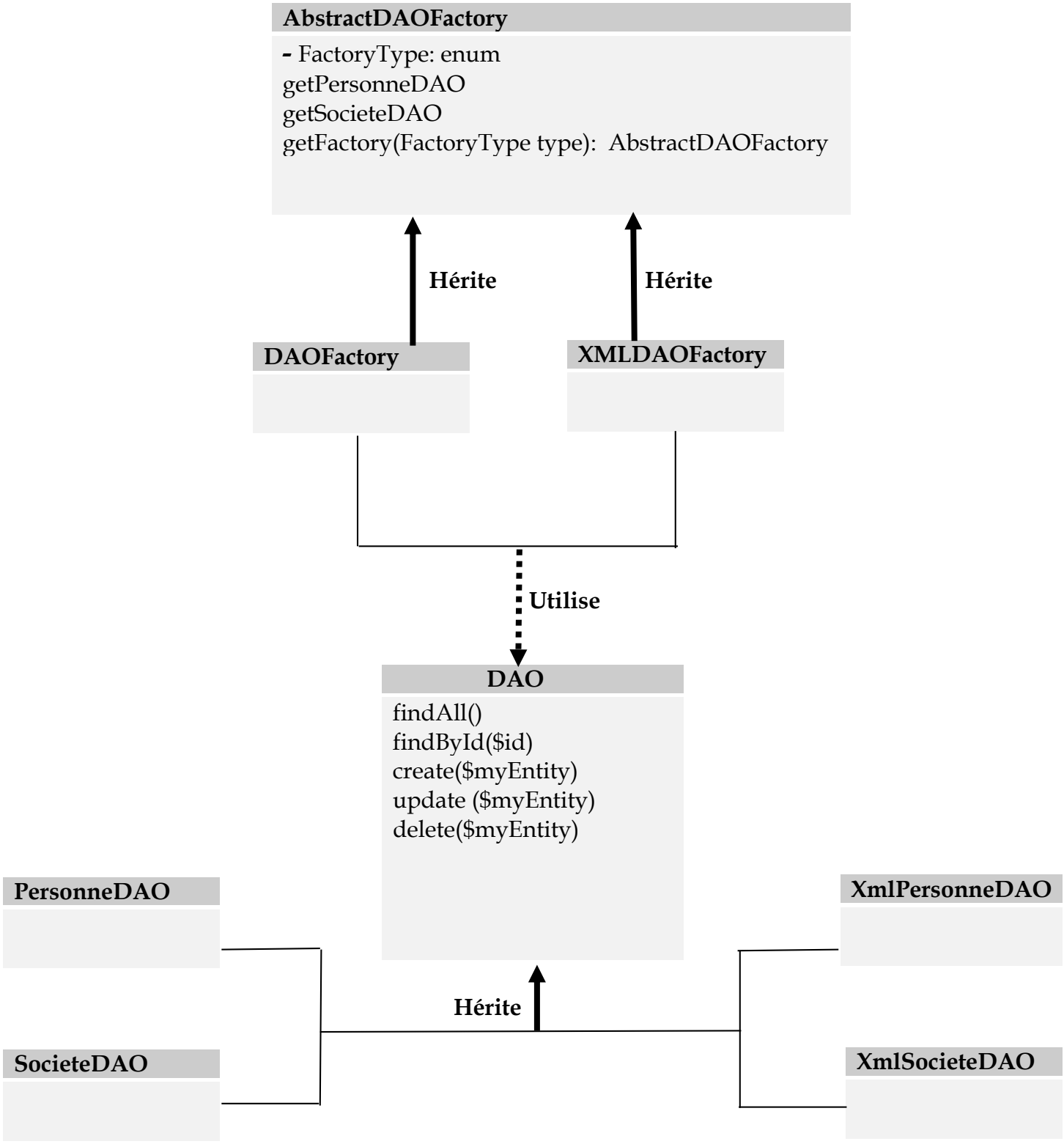
Description :

Le pattern **DAO** (Data Access Object) est un patron de conception qui propose de lier les objets en mémoire vive aux données persistantes (stockées en base de données, dans des fichiers, dans des annuaires, etc.). Ainsi, on centralise les mécanismes de mapping entre le système de stockage (SGBD) et les objets images de Java. Ce pattern a pour intérêt de prévenir un changement éventuel de système de stockage de données (de fichiers vers une base de données, de la SGBD MySQL vers la SGBD Oracle, etc.).

.

Structure

Diagramme de classes



Domaines d'utilisation

Ce pattern a pour intérêt de prévenir un changement éventuel de système de stockage de données (de fichiers vers une base de données, de la SGBD MySQL vers la SGBD Oracle, etc.).

Exemple d'implémentation :

Vous avez un client qui change souvent d'architecture de sources de donnés. Il passe des sources de données fichier CSV, fichier XML, fichier Json, base de données (MySQL, Oracle). Le but de cet exercice est de lui proposer une architecture pour que le code dans la couche de présentation (UI : User Interface ou IHM : Interface Homme/Machine) reste invariable.

Il s'agira d'implémenter les méthodes ci-dessous dans chaque section (Manual, CSV, Xml, Json et SQL) :

findAll(),
findById($id),
create($person),
update($person),
delete($person).

b. DTO

Un objet de transfert de données (data transfer object ou DTO en anglais) est un patron de conception utilisé dans les architectures logicielles objet.

Son but est de simplifier les transferts de données entre les sous-systèmes d'une application logicielle. Les objets de transfert de données sont souvent utilisés en conjonction des objets d'accès aux données.

Le DTO se distingue du DAO (objet d'accès aux données) car il ne permet que de modifier ou d'accéder à ses données (avec des mutateurs et accesseurs).

.

c. MVC, MVP, MVVM

MVC

Le motif MVC a été créé par **Trygve Reenskaug** lors de sa visite du Palo Alto Research Center (abr. PARC) en 1978. Le nom original est thing model view editor pattern, puis il a été rapidement renommé Model-View-Controller pattern. Le patron MVC a été utilisé la première fois pour créer des interfaces graphiques avec le langage de programmation Smalltalk en 1980.

Une application conforme au motif MVC comporte trois types de modules : les modèles, les vues et les contrôleurs.

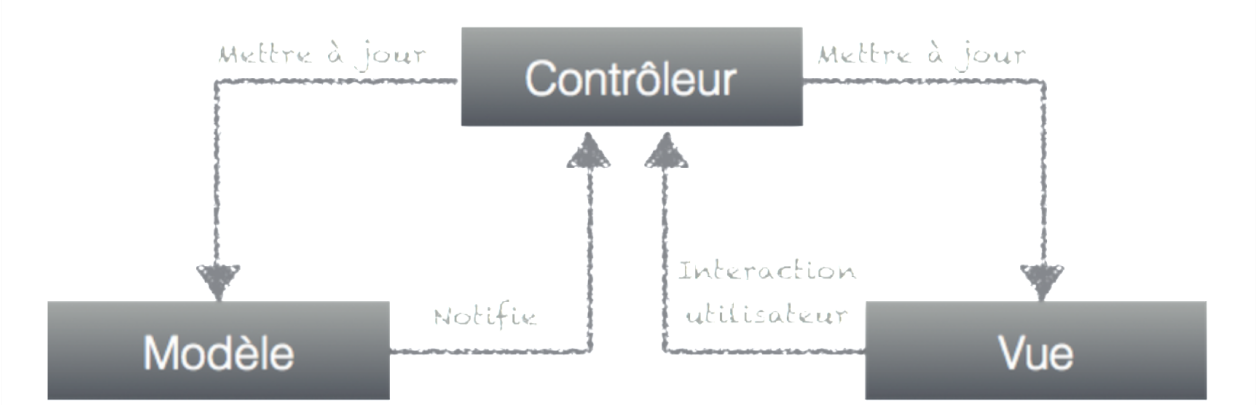
- Modèle

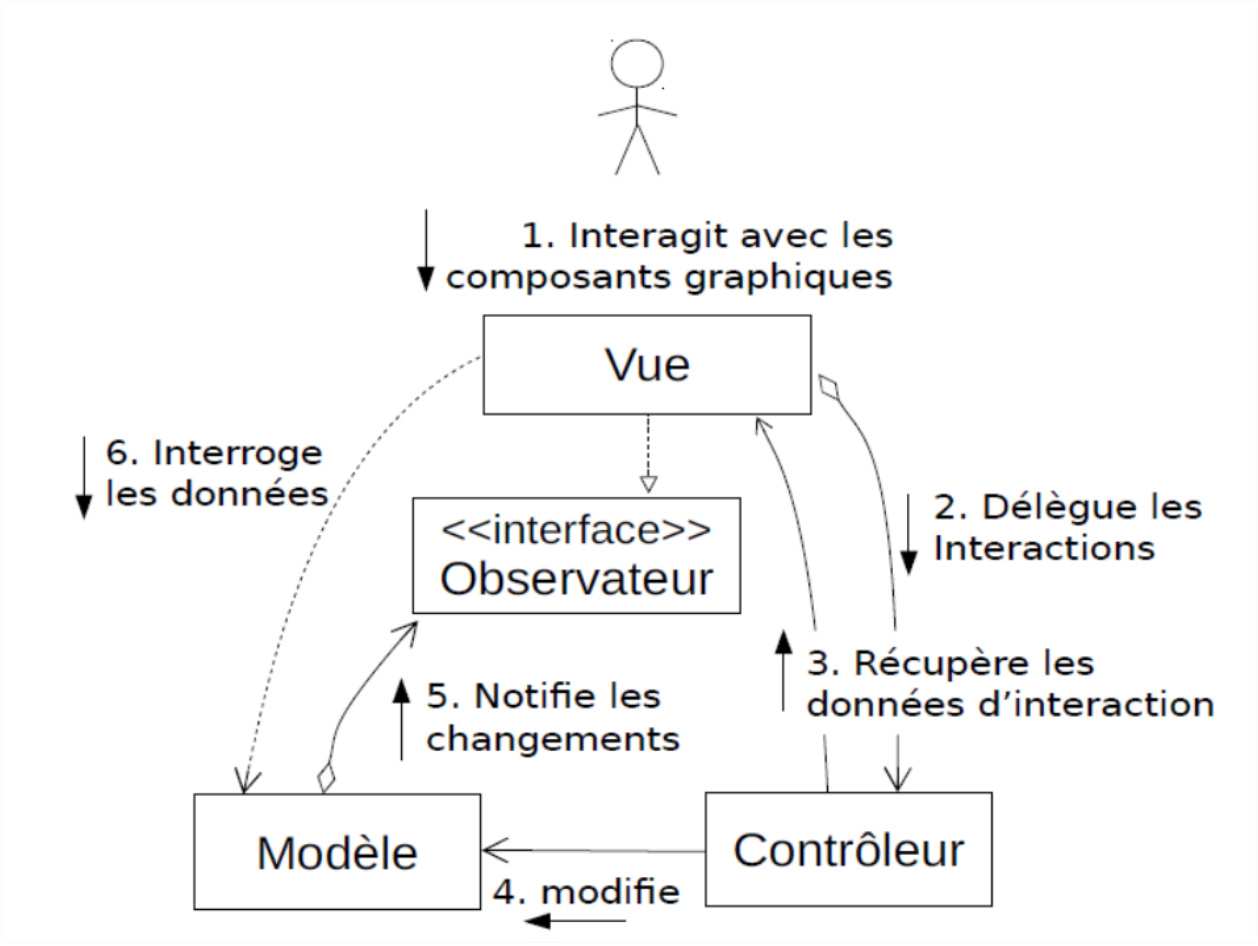
  Élément qui contient les données ainsi que de la logique en rapport avec les données: validation, lecture et enregistrement. Il peut, dans sa forme la plus simple, contenir uniquement une simple valeur, ou une structure de données plus complexe. Le modèle représente l'univers dans lequel s'inscrit l'application. Par exemple pour une application de banque, le modèle représente des comptes, des clients, ainsi que les opérations telles que dépôt et retraits, et vérifie que les retraits ne dépassent pas la limite de crédit.
- Vue

  Partie visible d'une interface graphique. La vue se sert du modèle, et peut être un diagramme, un formulaire, des boutons, etc. Une vue contient des éléments visuels ainsi que la logique nécessaire pour afficher les données provenant du modèle. Dans une application de bureau classique, la vue obtient les données nécessaires à la présentation du modèle en posant des questions. Elle peut également mettre à jour le modèle en envoyant des messages appropriés. Dans une application web une vue contient des balises HTML.

- Contrôleur

  Module qui traite les actions de l'utilisateur, modifie les données du modèle et de la vue.

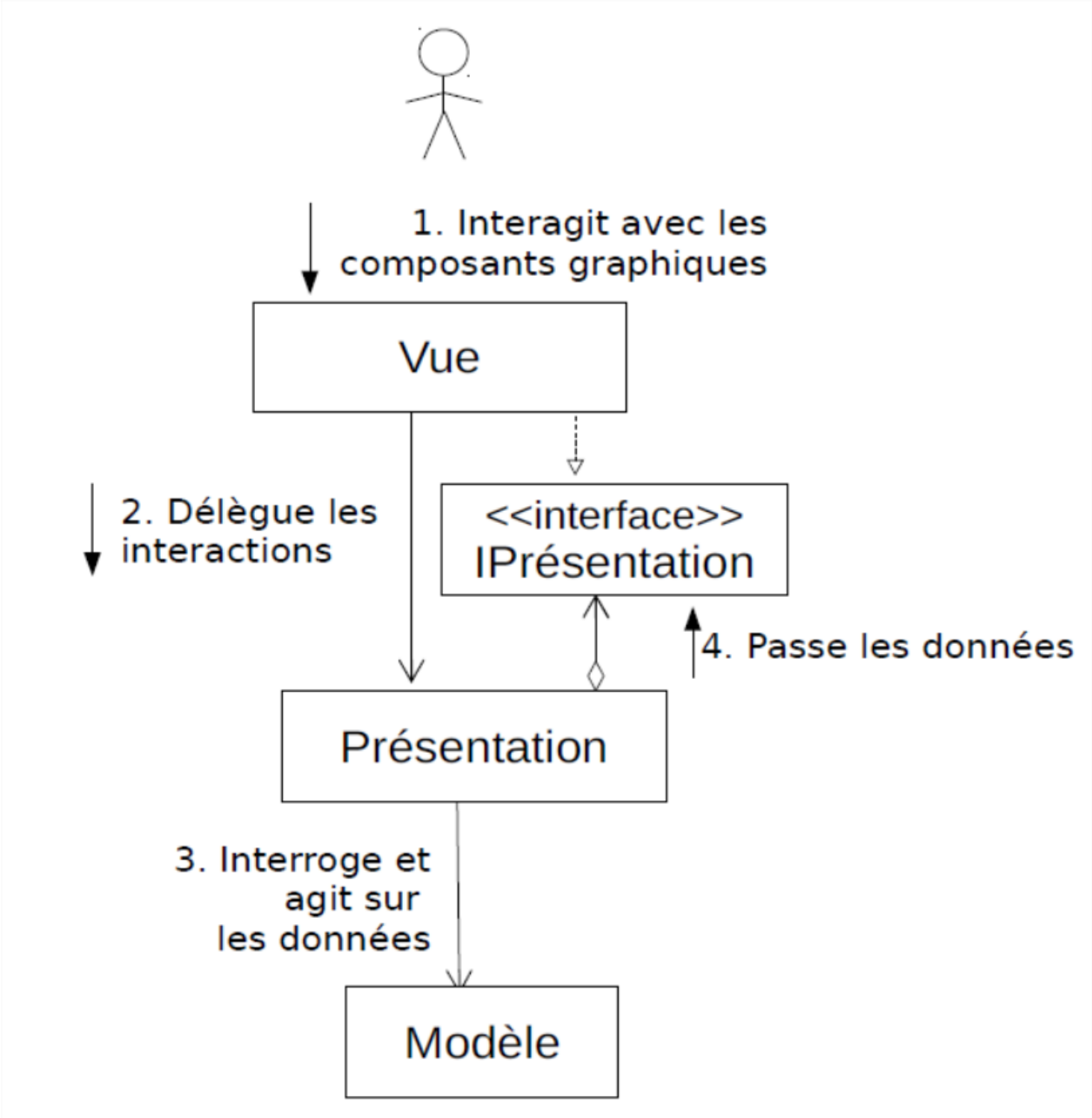



1. Interagit avec les composants graphiques
Vue
6. Interroge les données
<<interface>>
Observateur
2. Délègue les Interactions
3. Récupère les données d'interaction
5. Notifie les changements
Modèle
Contrôleur
4. modifie

MVP

Le **modèle-vue-présentation** (en abrégé **MVP**, de l'anglais *model-view-presenter*) est un patron d'architecture, considéré comme un dérivé du patron d'architecture modèle-vue-contrôleur.

Il garde les mêmes principes que MVC sauf qu'il élimine l'interaction entre la vue et le modèle parce qu'elle sera effectuée par le biais de la *présentation*, qui organise les données à afficher dans la vue.



- Le modèle est uniquement centré sur la logique métier. Il est totalement déconnecté du dialogue avec l'utilisateur. Cette fois, le modèle est totalement indépendant des autres parties. Concrètement, il n'y a aucune importation de classes des autres parties dans le modèle.

- La vue ne concerne que la gestion graphique. Elle est totalement dépendante de la bibliothèque graphique utilisée. A contrario, elle est totalement indépendante du reste de l'application et débarrassée de toute logique.

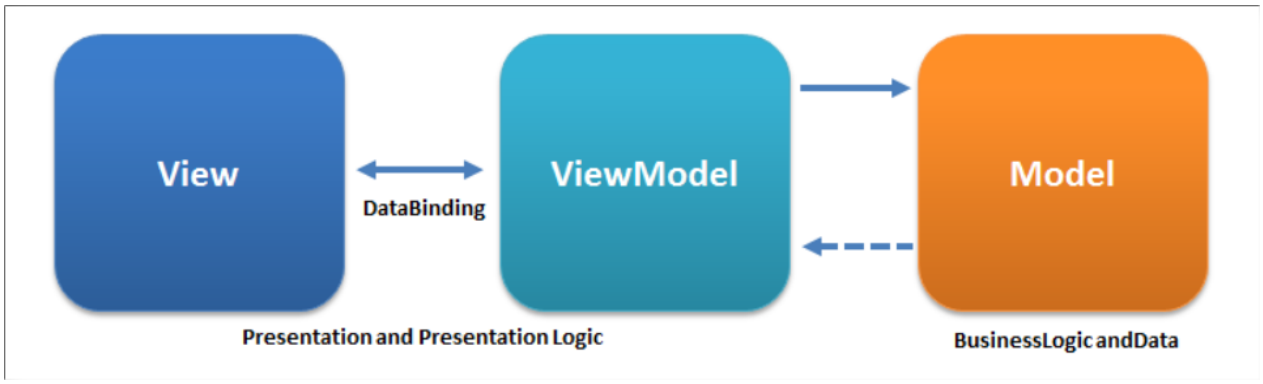
- La présentation contient toute la logique de présentation ainsi que l'état courant du dialogue avec l'utilisateur. Elle sait comment réagir aux événements de l'utilisateur en modifant les données du domaine si nécessaire puis en répercutant les effets vers la vue. Toutefois elle est indépendante de la bibliothèque graphique utilisée. Concrètement, le code n'importe aucune classe de la bibliothèque graphique. Les classes de la présentation font donc le lien entre des classes du modèle et celles de la vue. La présentation est une implémentation du patron de conception Médiateur.

Contrairement à MVC, il n'y a pas de patron Observateur entre le modèle et la vue. La raison est que la vue ne contient plus de logique de présentation. C'est maintenant le travail de la présentation de savoir quand mettre à jour la vue. Les méthodes de mise à jour appartiennent à l'interface IPresentation entièrement défnie par la présentation, ce qui est un excellent exemple du principe d'inversion de dépendance. Toutefois, la présentation ne dépend pas des technologies graphiques utilisées par la vue.Le cycle de dialogue est donc centré sur la présentation. L'utilisateur interagit avec la vue qui fait appel à des méthodes de la présentation. En fonction de la logique de présentation, la présentation met à jour le modèle du domaine et répercute les effets sur les vues.

MVVM

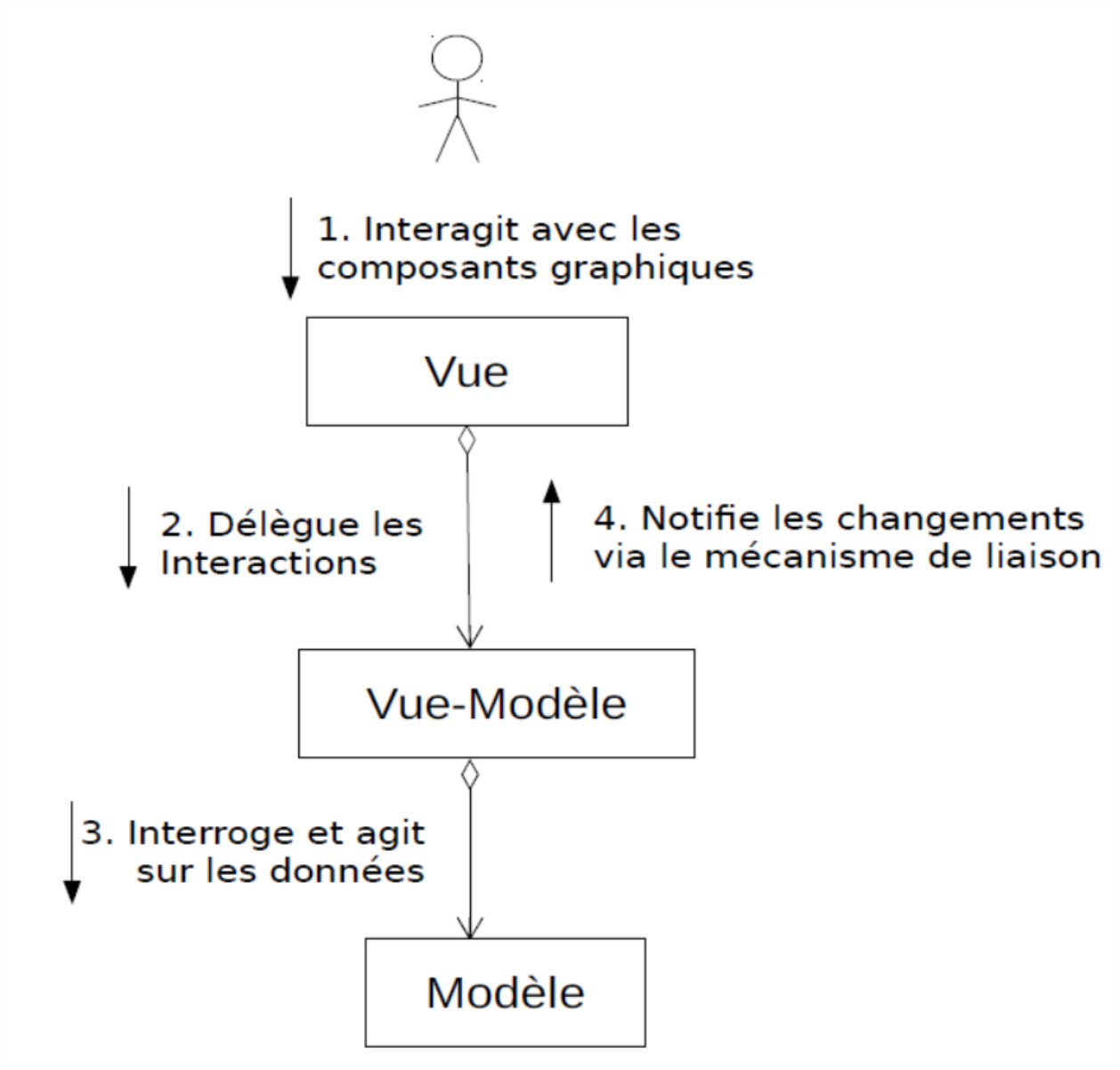
Le **modèle-vue-vue modèle** (en abrégé **MVVM**, de l'anglais *Model View ViewModel*) est une architecture et une méthode de conception utilisée dans le génie logiciel.

Il est apparu en 2004, originaire de Microsoft et adapté pour le développement des applications basées sur les technologies Windows Presentation Foundation et Silverlight via l'outil MVVM Light par exemple. Cette méthode permet, tel le modèle MVC (modèle-vue-contrôleur), de séparer la vue de la logique et de l'accès aux données en accentuant les principes de binding et d'événement.



- Le modèle reste le même que pour le patron **MVP**. Il ne contient donc que le logique métier.
- La vue ne contient aucune logique. La différence avec **MVP** réside dans l'utilisation de liaisons entre des composants de la vue et des attributs de la vue-modèle.
- La vue-modèle contient la logique de présentation et l'état de l'interface comme la présentation de MVP.

Ce patron ne requière pas d'interface entre la vue et la vue-modèle puisque que la vue-modèle est déjà complètement découplée de la vue.


1. Interagit avec les composants graphiques
Vue
2. Délègue les Interactions
4. Notifie les changements via le mécanisme de liaison
Vue-Modèle
3. Interroge et agit sur les données
Modèle

d. Anemic Model

Un modèle de domaine anémique est un modèle dépourvu de logique. Les classes de domaine ressemblent davantage à un groupe de setters et de getters publics sans logique de domaine, dans lequel le client de la classe a le contrôle sur la manière d'instancier et de modifier la classe. Dans ces modèles, le client doit interpréter l'objectif et l'utilisation de la classe. Habituellement, la logique a été transférée vers d'autres classes appelées services, assistance ou gestionnaire.

Ces classes modèles anémiques peuvent aussi contenir des données et des connexions vers d'autres entités.