

Présenté par Godwin

Godwin AVODAGBE

A blurred background photograph shows several people in an office setting, smiling and clapping their hands, suggesting a positive event or presentation.

DESIGN
PATTERN



Sommaire

- I. Introduction
- II. Présentation du projet du cours
- III. Pattern de construction
- IV. Pattern de structuration
- V. Pattern de comportement
- VI. Conclusion

INTRODUCTION

Nous ferons dans ce cours, une présentation des vingt-trois modèles de conception qui furent introduits en 1995 dans le livre « *Design Patterns - Elements of Reusable Object Oriented Software* » du **Gang of Four** (la bande des quatre auteurs).

Un design pattern constitue une solution à un problème récurrent de conception en programmation par objets. Chaque pattern est présenté en **décrivant le problème correspondant, la solution apportée par le pattern et sa structure générique** à l'aide d'un ou de plusieurs diagrammes UML et les **domaines d'application**. La solution est approfondie sous la forme d'un petit programme écrit en C# qui montre une mise en œuvre du pattern.

« *Design Patterns pour C#* » s'adresse aux concepteurs et aux développeurs pratiquant régulièrement la programmation par objets pour réaliser des applications complexes et ayant la volonté de réutiliser des solutions connues et robustes pour améliorer la qualité des logiciels qu'ils élaborent.

INTRODUCTION

Les objectifs que nous allons viser dans ce cours sont les suivants :

- Acquérir une connaissance des éléments essentiels des vingt-trois patterns, notamment leur structure générique sous forme d'un diagramme de classes UML.
- Affiner nos connaissances en examinant des exemples C# de mise en œuvre, en étudiant les compositions et variantes de patterns décrites dans le chapitre correspondant et en réalisant les exercices.
- Etudier les design patterns pour améliorer notre maîtrise des principes de l'approche à objets comme le polymorphisme, la surcharge des méthodes, les interfaces, les classes et les méthodes abstraites, la délégation, la généricité...

INTRODUCTION

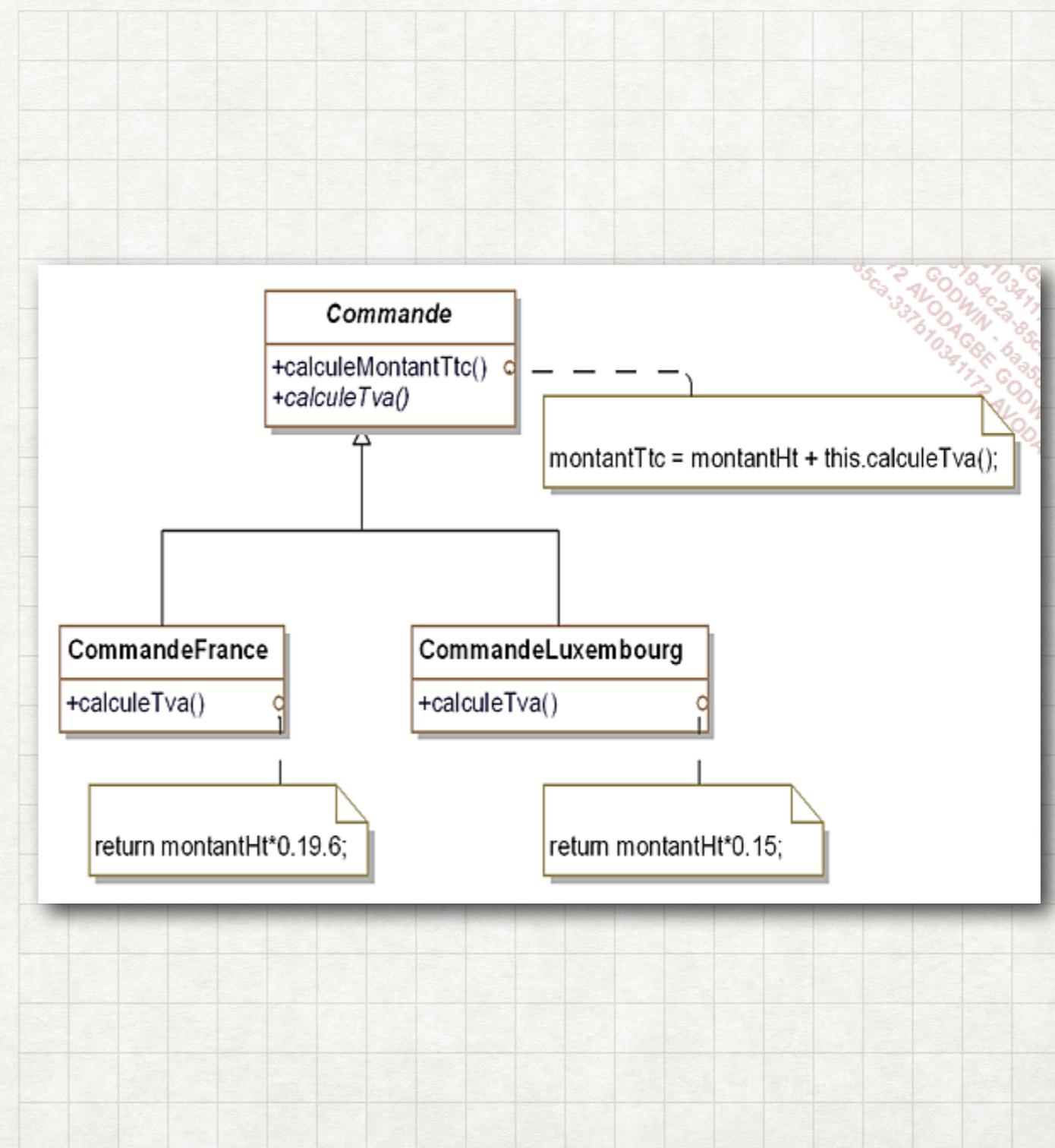
Un design pattern ou pattern de conception consiste en un schéma à objets qui forme une solution à un problème connu et fréquent. Le schéma à objets est constitué par un ensemble d'objets décrits par des classes et des relations liant les objets.

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

INTRODUCTION

LA DESCRIPTION

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



INTRODUCTION

LE CATALOGUE DES PATTERNS DE CONCEPTION

La liste qui suit n'est pas exhaustive, elle provient de l'ouvrage de référence "GoF":

- Abstract Factory** (Fabrique abstraite): a pour objectif la création d'objets regroupés en familles sans devoir connaître les classes concrètes destinées à la création de ces objets
- Builder** (Monteur): permet de séparer la construction d'objets complexes de leur implantation de sorte qu'un client puisse créer ces objets complexes avec des implantations différentes
- Factory Method** (Fabrique): a pour but d'introduire une méthode abstraite de création d'un objet en reportant aux sous-classes concrètes la création effective
- Prototype** (Prototype): permet la création de nouveaux objets par duplication d'objets existants appelés prototypes qui disposent de la capacité de clonage
- Singleton** (Singleton): permet de s'assurer qu'une classe ne possède qu'une seule instance et de fournir une méthode unique retournant cette instance
- Adapter** : a pour but de convertir l'interface d'une classe existante en l'interface attendue par des clients également existants afin qu'ils puissent travailler ensemble

INTRODUCTION

LE CATALOGUE DES PATTERNS DE CONCEPTION

- Bridge** : a pour but de séparer les aspects conceptuels d'une hiérarchie de classes de leur implantation
- Composite** : offre un cadre de conception d'une composition d'objets dont la profondeur de composition est variable, la conception étant basée sur un arbre
- Decorator** : permet d'ajouter dynamiquement des fonctionnalités supplémentaires à un objet
- Facade** : a pour but de regrouper les interfaces d'un ensemble d'objets en une interface unifiée rendant cet ensemble plus simple à utiliser
- Flyweight** : facilite le partage d'un ensemble important d'objets dont le grain est fin
- Proxy** : construit un objet qui se substitue à un autre objet et qui contrôle son accès
- Chain of responsibility** : crée une chaîne d'objets telle que si un objet de la chaîne ne peut pas répondre à une requête, il puisse la transmettre à ses successeurs jusqu'à ce que l'un d'entre eux y réponde
- Command** : a pour objectif de transformer une requête en un objet, facilitant des opérations comme l'annulation, la mise en file des requêtes et leur suivi
- Interpreter** : fournit un cadre pour donner une représentation par objets de la grammaire d'un langage afin d'évaluer, en les interprétant, des expressions écrites dans ce langage.

INTRODUCTION

LE CATALOGUE DES PATTERNS DE CONCEPTION

- Iterator** : fournit un accès séquentiel à une collection d'objets sans que les clients se préoccupent de l'implantation de cette collection
- Mediator** : construit un objet dont la vocation est la gestion et le contrôle des interactions au sein d'un ensemble d'objets sans que ses éléments se connaissent mutuellement
- Memento** : sauvegarde et restaure l'état d'un objet
- Observer** : construit une dépendance entre un sujet et des observateurs de façon à ce que chaque modification du sujet soit notifiée aux observateurs afin qu'ils puissent mettre à jour leur état
- State** : permet à un objet d'adapter son comportement en fonction de son état interne
- Strategy** : adapte le comportement et les algorithmes d'un objet en fonction d'un besoin sans changer les interactions avec les clients de cet objet
- Template Method** : permet de reporter dans des sous-classes certaines étapes de l'une des opérations d'un objet, ces étapes étant alors décrites dans les sous-classes
- Visitor** : construit une opération à réaliser sur les éléments d'un ensemble d'objets. De nouvelles opérations peuvent ainsi être ajoutées sans modifier les classes de ces objets.

INTRODUCTION

LE CATALOGUE DES PATTERNS DE CONCEPTION

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

Puis, il convient d'étudier en détail le ou les patterns découverts à l'aide de leur description complète.

En particulier, il convient d'étudier au travers de l'exemple fourni et de la structure générique si le pattern répond de façon pertinente au problème. Cette étude doit surtout inclure la possibilité pour la structure générique d'être adaptée et il faut donc, en fait, chercher si le pattern après adaptation répond au problème. Cette étape d'adaptation est une étape importante de l'utilisation du pattern pour résoudre un problème. Nous la décrivons par la suite.

INTRODUCTION

COMMENT CHOISIR ET UTILISER UN PATTERN DE CONCEPTION POUR RÉSOUDRE UN PROBLÈME ?

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

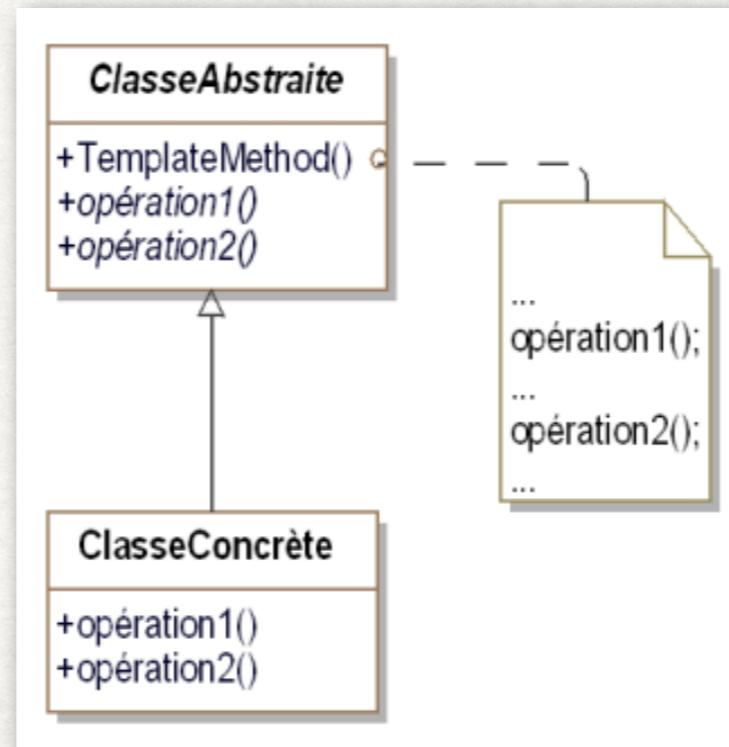
- étudier de façon approfondie sa structure générique qui sert de base pour utiliser un pattern
- renommer les classes et les méthodes introduites dans la structure générique. En effet, dans la structure générique d'un pattern, le nom des classes et des méthodes est abstrait. À l'inverse, une fois intégrées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

INTRODUCTION

COMMENT CHOISIR ET UTILISER UN PATTERN DE CONCEPTION POUR RÉSOUDRE UN PROBLÈME ?

- adapter la structure générique pour répondre aux contraintes de l'application, ce qui peut impliquer des changements dans le schéma d'objets

Nous donnons un exemple d'adaptation du pattern Template method , la structure générique de ce pattern est donnée à la figure ci-dessous :



INTRODUCTION

COMMENT CHOISIR ET UTILISER UN PATTERN DE CONCEPTION POUR RÉSOUDRE UN PROBLÈME

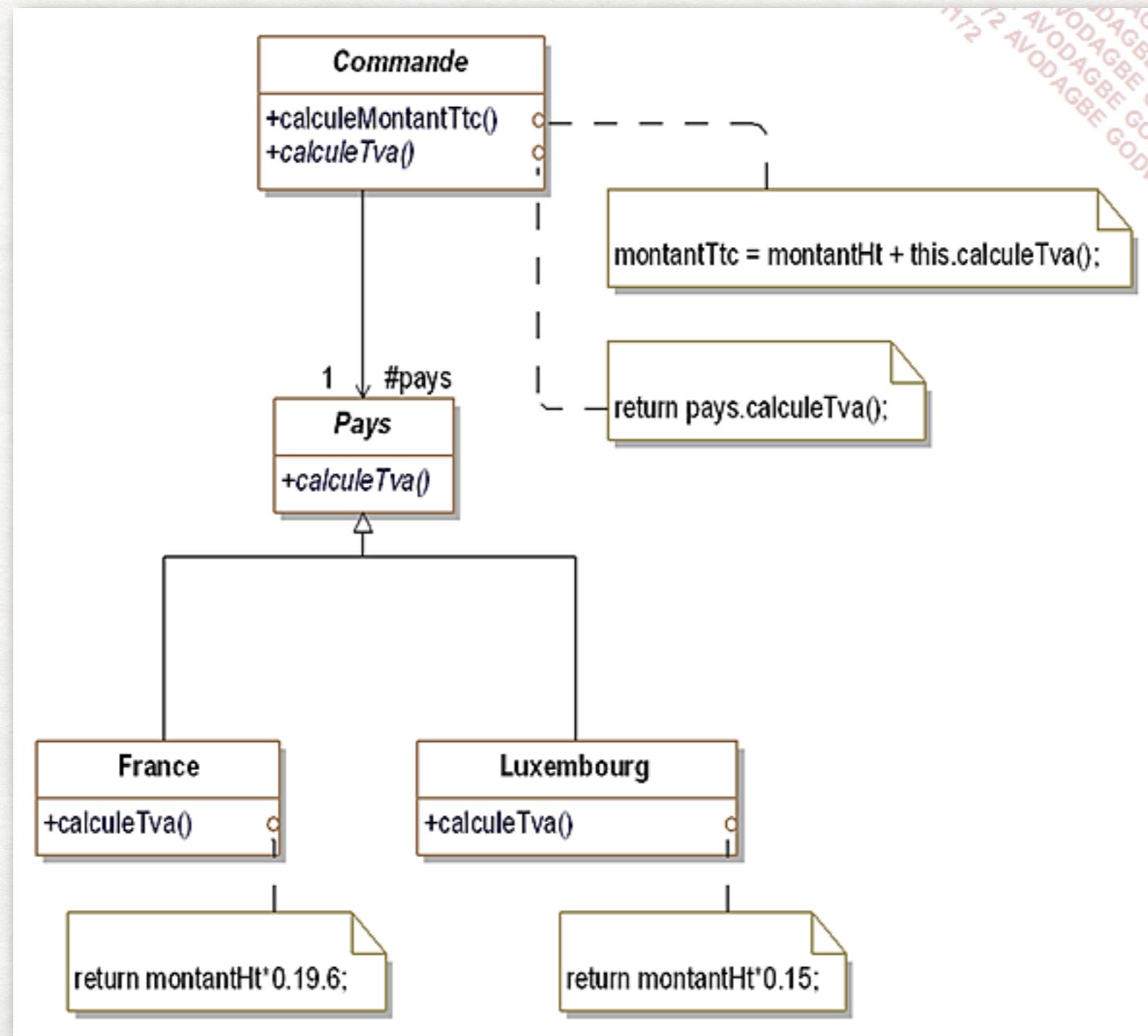
Nous voulons adapter cette structure dans le cadre d'une application de commerce où la méthode du calcul de la TVA n'est pas incluse dans la classe `Commande` mais dans les sous-classes concrètes de la classe abstraite `Pays`. Ces sous-classes contiennent toutes les méthodes de calcul des taxes spécifiques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

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

Le pattern adapté prêt à l'utilisation dans l'application est illustré à la figure suivante :

INTRODUCTION

COMMENT CHOISIR ET UTILISER UN PATTERN DE CONCEPTION POUR RÉSOUDRE UN PROBLÈME ?



INTRODUCTION

ORGANISATION DU CATALOGUE DES PATTERNS DE CONCEPTION

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

- Les patterns de construction ont pour objectif l'abstraction des mécanismes de création d'objets. Les mécanismes d'instanciation des classes concrètes sont encapsulés au sein des patterns. En cas de modification de l'ensemble des classes concrètes à instancier, les modifications à apporter dans le système seront faibles voire inexistantes. Ils sont au nombre de cinq : Abstract Factory, Builder, Factory Method, Prototype et Singleton.
- Les patterns de structuration ont pour but d'abstraire l'interface d'un objet ou d'un ensemble d'objets de son implantation. Dans le cas d'un ensemble d'objets, l'objectif est aussi d'abstraire l'interface des relations d'héritage ou de composition présentes dans l'ensemble. Ils sont au nombre de sept : Adapter, Bridge, Composite, Decorator, Facade, Flyweight et Proxy.
- Les patterns de comportement fournissent des solutions pour structurer les données et les objets ainsi qu'organiser les interactions en distribuant les traitements et les algorithmes entre les objets. Ils sont au nombre de onze : Chain of responsibility, Command, Interpreter, Iterator, Mediator, Memento, Observer, State, Strategy, Template Method et Visitor.

PRESENTATION DU PROJET DU COURS

PRESENTATION DU PROJET

Pour les besoins du cours, nous prendrons l'exemple de la conception d'un système pour illustrer la mise en œuvre des vingt-trois patterns de conception.

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

PRESENTATION DU PROJET

CAHIER DES CHARGES

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

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

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

PRESENTATION DU PROJET

CAHIER DES CHARGES

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

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

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

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

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

Description de la partie	Pattern de conception
Construire les objets du domaine (Automobile à essence, automobile à électricité, etc.).	Abstract Factory
Construire les liasses de documents nécessaires en cas d'acquisition d'un véhicule.	Builder, Prototype
Créer les commandes.	Factory Method
Créer la liasse vierge de documents.	Singleton
Gérer des documents PDF.	Adapter
Implémenter des formulaires en HTML ou à l'aide d'une applet.	Bridge
Représenter les sociétés clientes.	Composite
Afficher les véhicules du catalogue.	Decorator, Observer, Strategy
Fournir l'interface en service Web du site.	Facade
Gérer les options d'un véhicule commandé.	Flyweight, Memento
Gérer l'affichage d'animations pour chaque véhicule du catalogue.	Proxy
Gérer la description d'un véhicule.	Chain of responsibility

Description de la partie	Pattern de conception
Solder les véhicules restés en stock pendant une longue durée.	Command
Rechercher dans la base de véhicules à l'aide d'une requête écrite sous la forme d'une expression logique.	Interpreter
Retrouver séquentiellement les véhicules du catalogue.	Iterator
Gérer le formulaire d'une demande de crédit.	Mediator
Gérer les états d'une commande.	State
Calculer le montant d'une commande.	Template Method
Envoyer des propositions commerciales par e-mail à certaines sociétés clientes.	Visitor

PATTERN DE CONSTRUCTION

PATTERN DE CONSTRUCTION

- 📍 Introduction
- 📍 Abstract Factory
- 📍 Builder
- 📍 Factory Method
- 📍 Prototype
- 📍 Singleton



INTRODUCTION

PRESENTATION

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

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

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

INTRODUCTION

PROBLÈMES LIÉS À LA CRÉATION D'OBJETS

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

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

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

INTRODUCTION

PROBLÈMES LIÉS À LA CRÉATION D'OBJETS

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

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

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

INTRODUCTION

PROBLÈMES LIÉS À LA CRÉATION D'OBJETS

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

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

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

PATTERN DE CONSTRUCTION

ABSTRACT FACTORY



ABSTRACT FACTORY

PRESENTATION

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.

ABSTRACT FACTORY

EXEMPLE

Le système de vente de véhicules gère des véhicules fonctionnant à l'essence et des véhicules fonctionnant à l'électricité. Cette gestion est confiée à l'objet Catalogue qui crée de tels objets.

Pour chaque produit, nous disposons d'une classe abstraite, d'une sous-classe concrète décrivant la version du produit fonctionnant à l'essence et d'une sous-classe décrivant la version du produit fonctionnant à l'électricité. Par exemple, à la figure suivante, 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 (diesel ou mixte essence-électricité), les modifications à apporter à l'objet Catalogue peuvent être assez lourdes.

ABSTRACT FACTORY

EXEMPLE

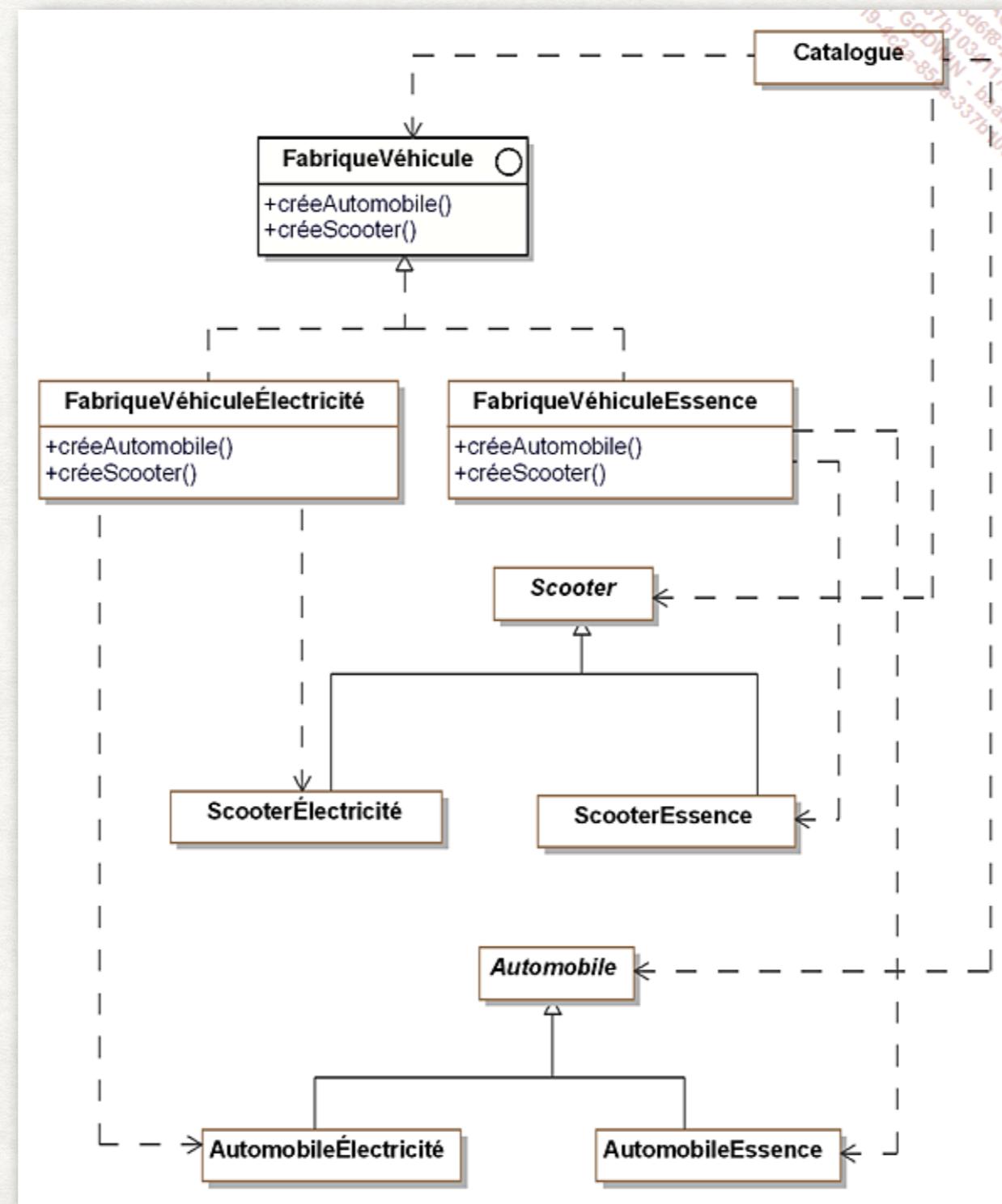
Le pattern Abstract Factory résout ce problème en introduisant une interface `FabriqueVéhicule` qui contient la signature des méthodes pour définir chaque produit. Le type de retour de ces méthodes est constitué par l'une des classes abstraites de produit. Ainsi l'objet Catalogue n'a pas besoin de connaître les sous-classes concrètes et reste indépendant des familles de produit.

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

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

ABSTRACT FACTORY

EXEMPLE



ABSTRACT FACTORY

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.

DEMO

PATTERN DE CONSTRUCTION **BUILDER**



BUILDER

PRESENTATION

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.

BUILDER

EXEMPLE

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

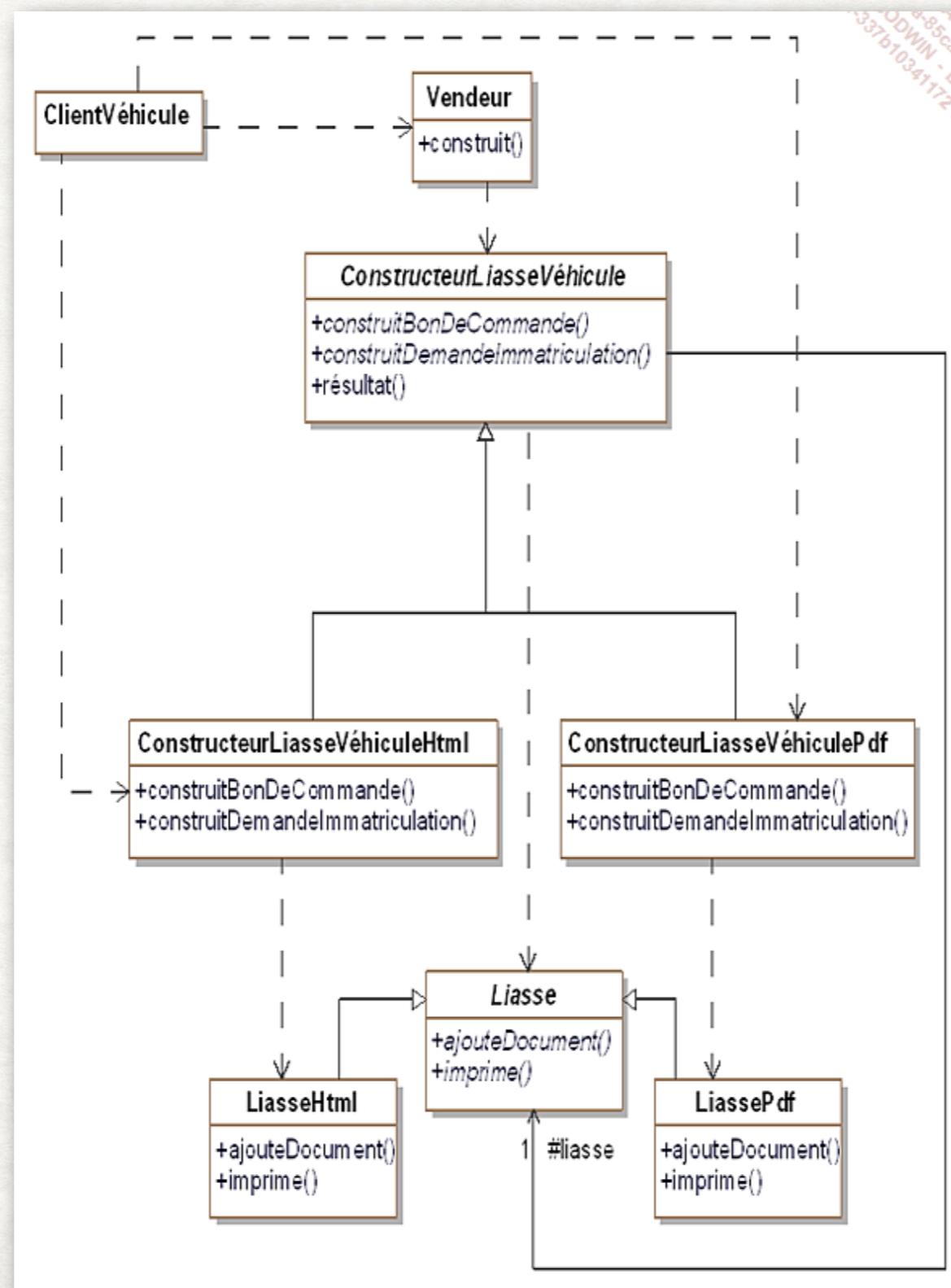
Ainsi le vendeur crée les documents de la liasse à l'aide des méthodes `construitBonDeCommande` et `construitDemandeImmatriculation`.

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.

BUILDER

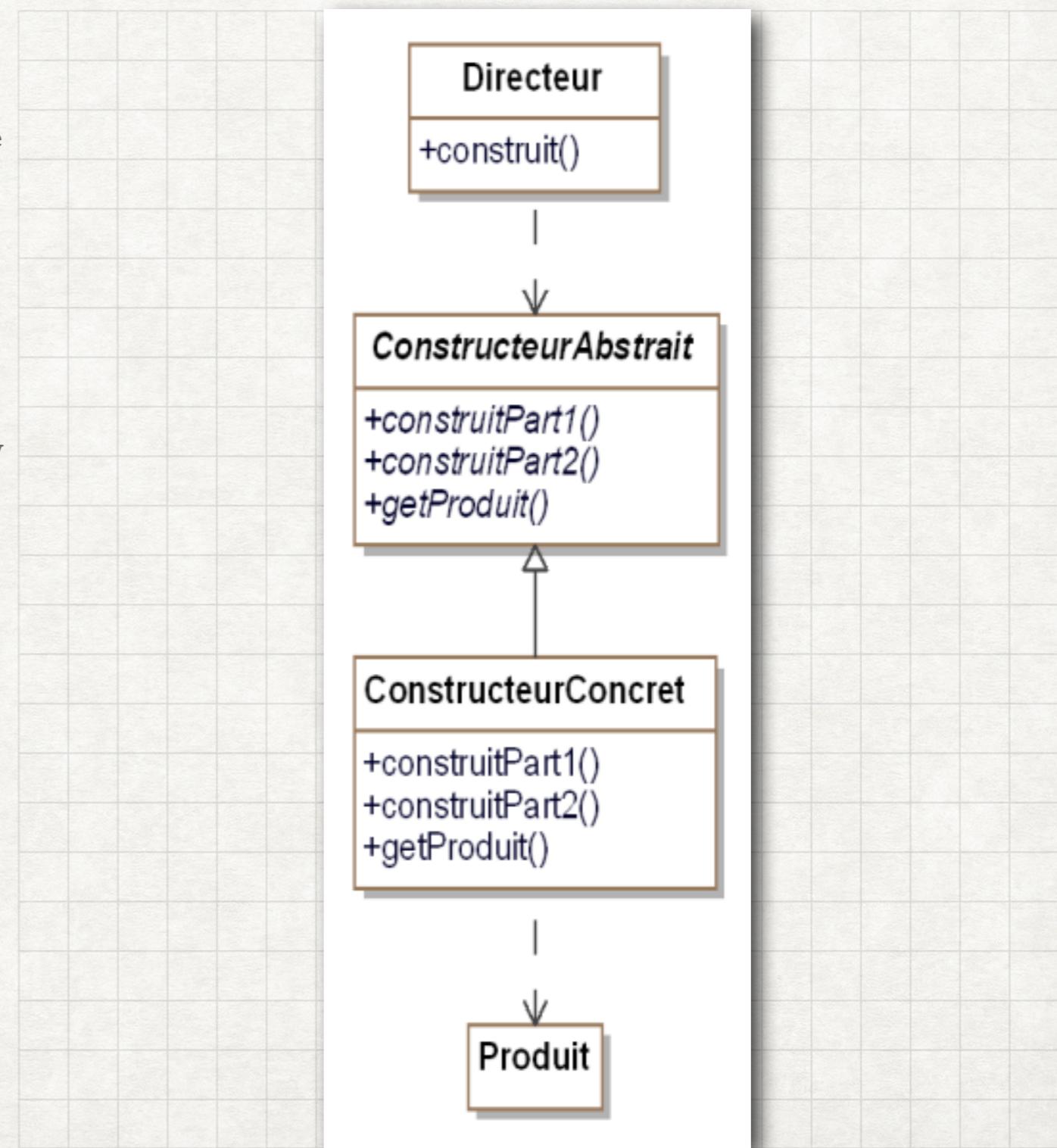
EXEMPLE



BUILDER STRUCTURE

Les participants au pattern sont les suivants :

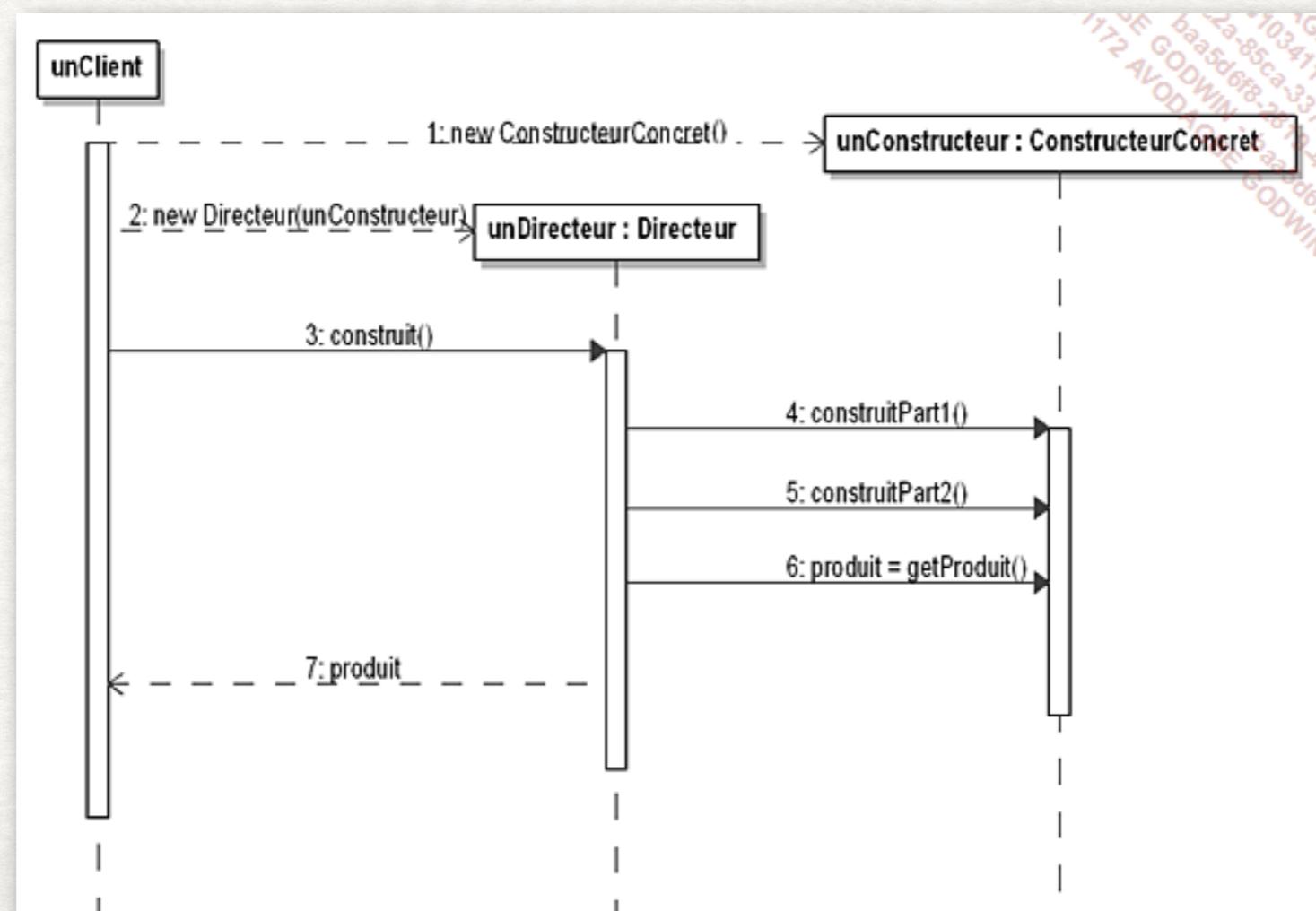
- ConstructeurAbstrait (ConstructeurLiisseVéhicule) **est la classe introduisant les signatures des méthodes construisant les différentes parties du produit ainsi que la signature de la méthode permettant d'obtenir le produit, une fois celui-ci construit**
- ConstructeurConcret (ConstructeurLiisseVéhiculeHtml et ConstructeurLiisseVéhiculePdf) **est la classe concrète implantant les méthodes du constructeur abstrait**
- Produit (Liisse) est la classe définissant le produit. Elle peut être abstraite et posséder plusieurs sous-classes concrètes (LiisseHtml et LiissePdf) en cas d'implantations différentes
- Directeur **est la classe chargée de construire le produit au travers de l'interface du constructeur abstrait.**



BUILDER

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.



BUILDER

DOMAINES D'UTILISATION

Le pattern est utilisé dans les domaines suivants :

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

DEMO

PATTERN DE CONSTRUCTION

FACTORY METHOD



FACTORY METHOD

PRESENTATION

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.

FACTORY METHOD

EXEMPLE

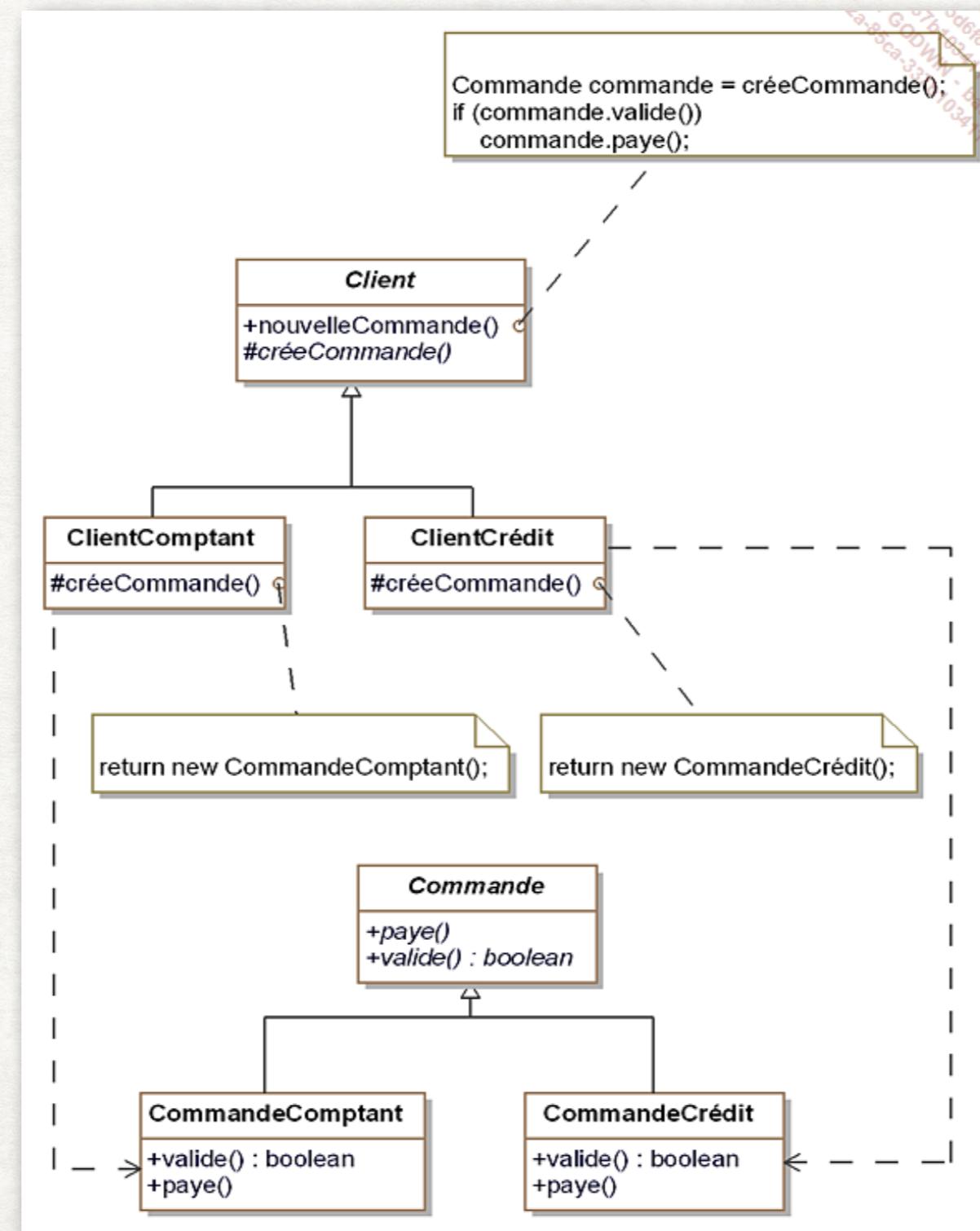
Nous nous intéressons aux clients et aux commandes. La classe Client introduit la méthode créeCommande qui doit créer la commande. Certains clients commandent un véhicule en payant au comptant et d'autres clients utilisent un crédit. En fonction de la nature du client, la méthode créeCommande doit créer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

- la classe concrète ClientComptant dont la méthode créeCommande crée une instance de la classe CommandeComptant
- la classe concrète ClientCrédit dont la méthode créeCommande crée une instance de la classe CommandeCrédit.

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 :

FACTORY METHOD

EXEMPLE

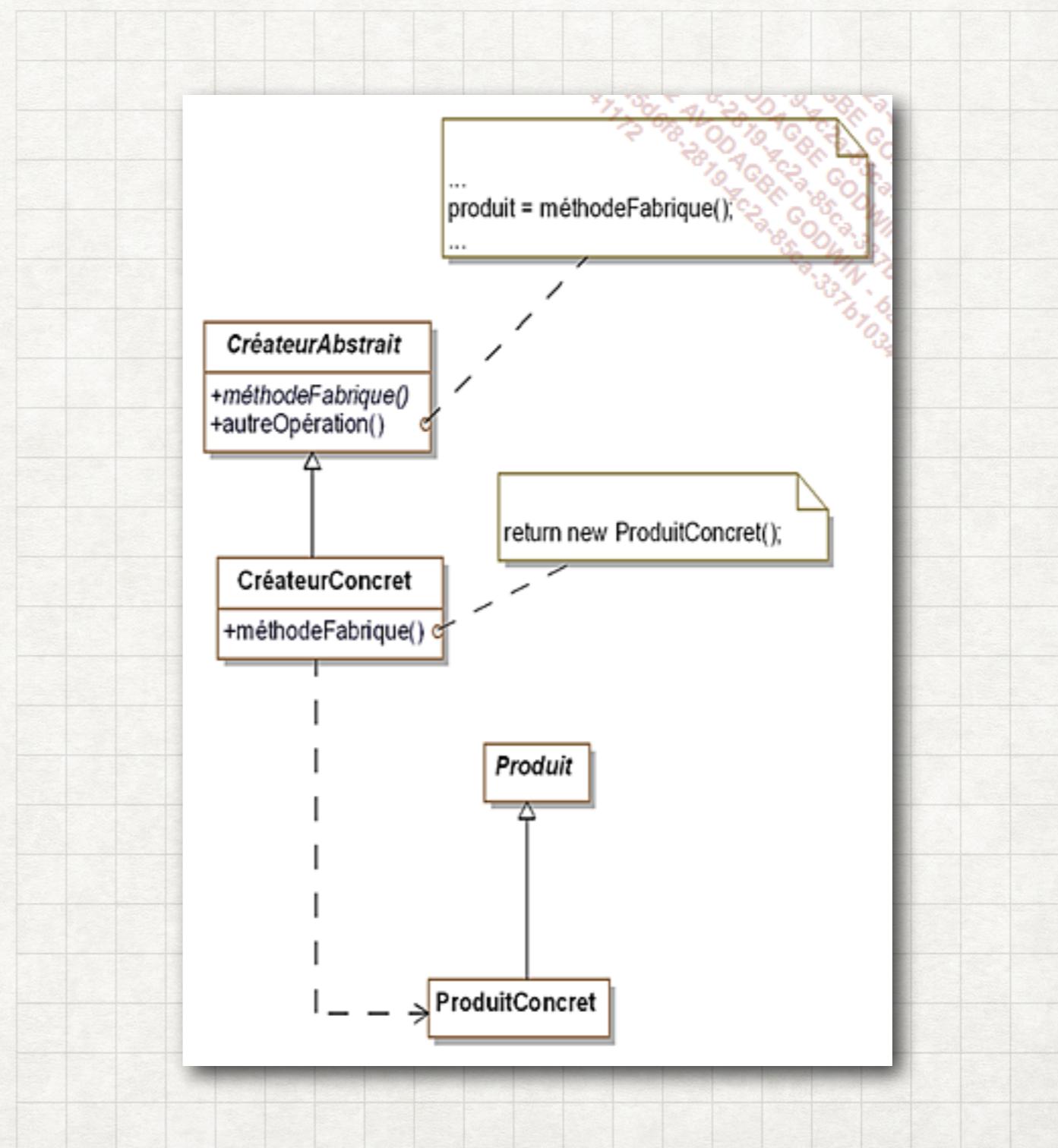


FACTORY METHOD

STRUCTURE

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.



FACTORY METHOD

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`.

FACTORY METHOD

DOMAINES D'UTILISATION

Le pattern est utilisé dans les domaines 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.

DEMO

PATTERN DE CONSTRUCTION PROTOTYPE



PROTOTYPE

PRESENTATION

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

PROTOTYPE

EXEMPLE

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

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

PROTOTYPE

EXEMPLE

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.

PROTOTYPE

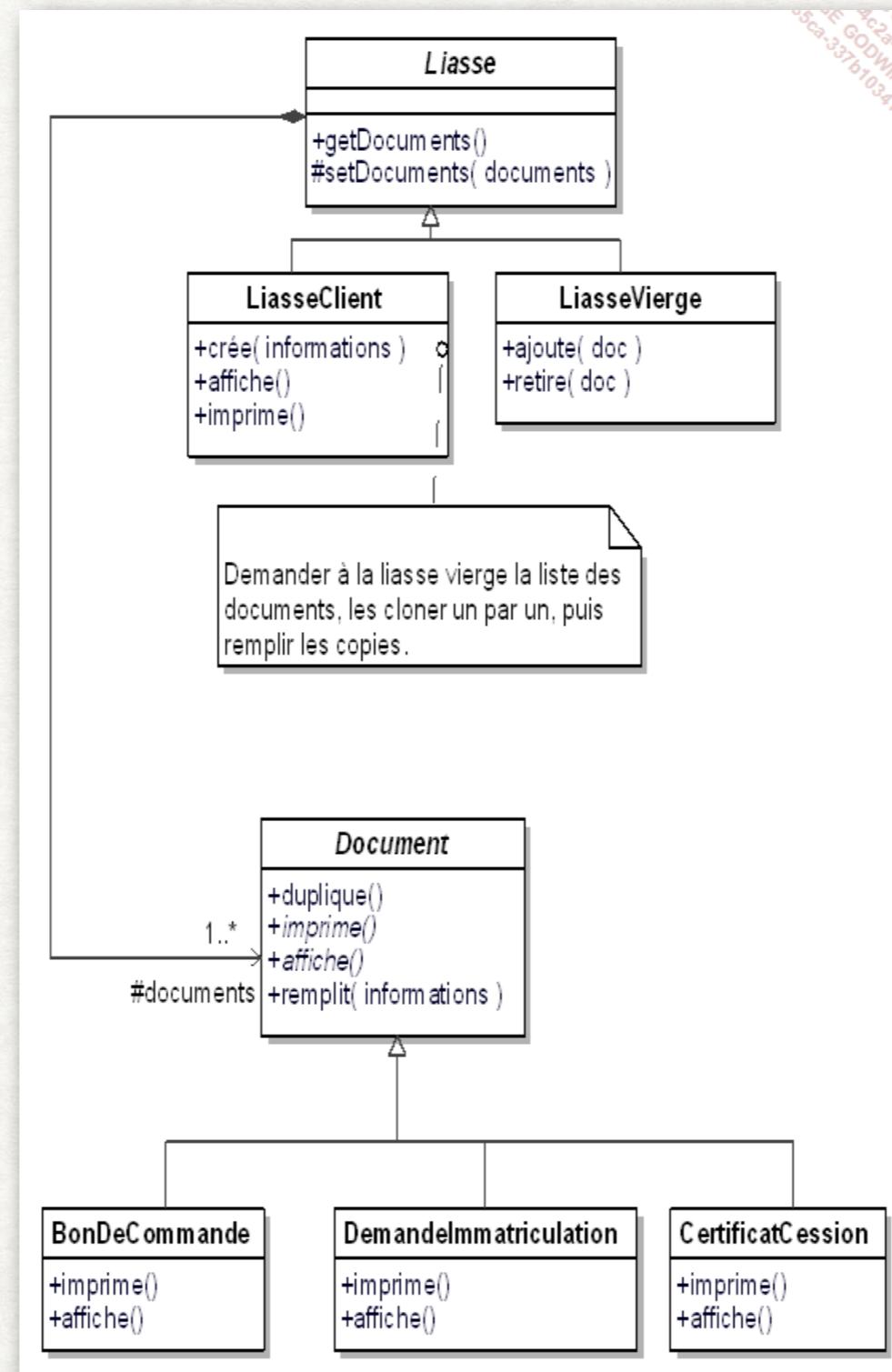
EXEMPLE

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.

PROTOTYPE

EXAMPLE

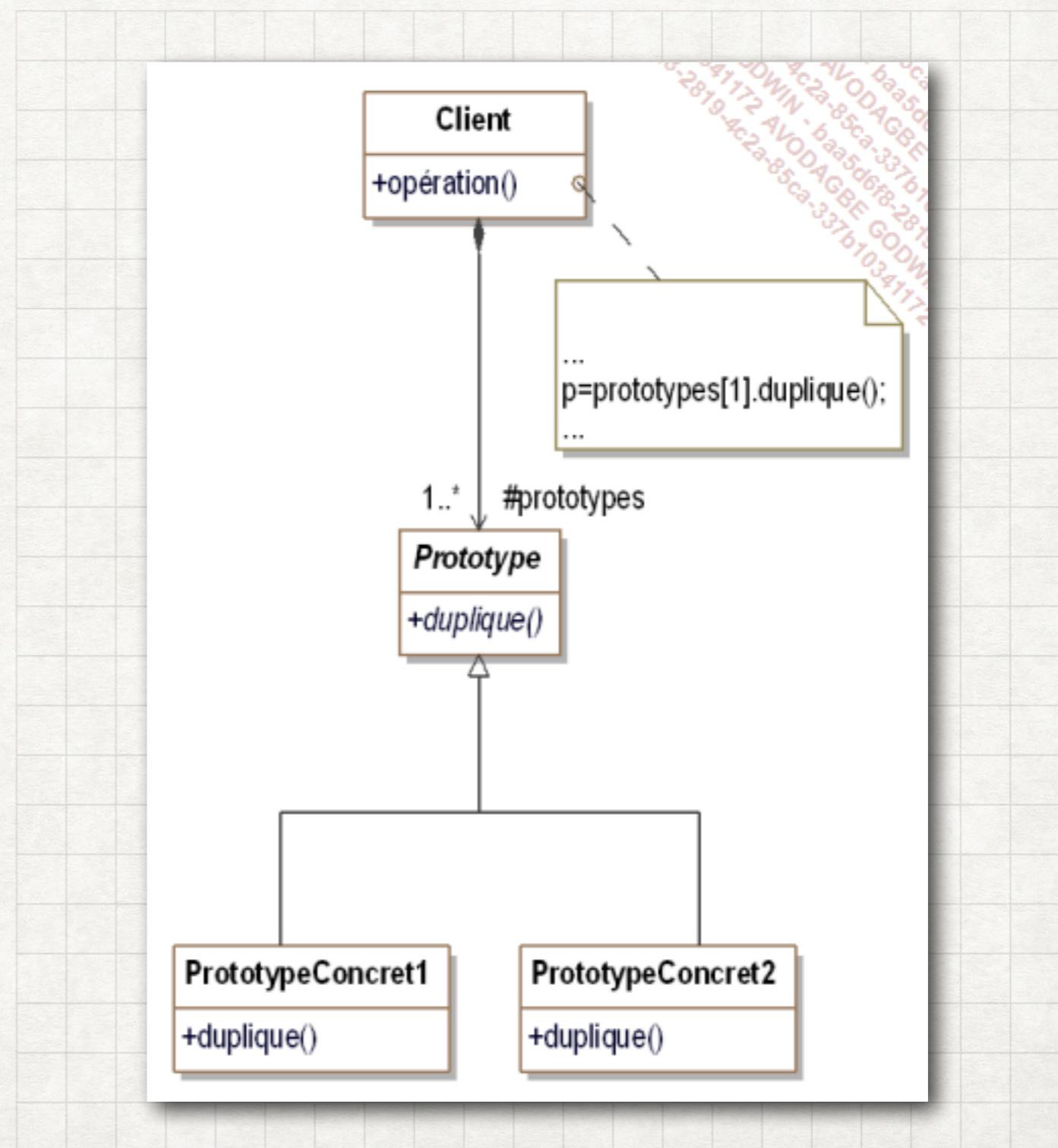


PROTOTYPE

STRUCTURE

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 implantent la méthode duplique.



PROTOTYPE

DOMAINES D'UTILISATION

Le pattern 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.

DEMO

PATTERN DE CONSTRUCTION SINGLETON



SINGLETON

PRESENTATION

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

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

SINGLETON

EXEMPLE

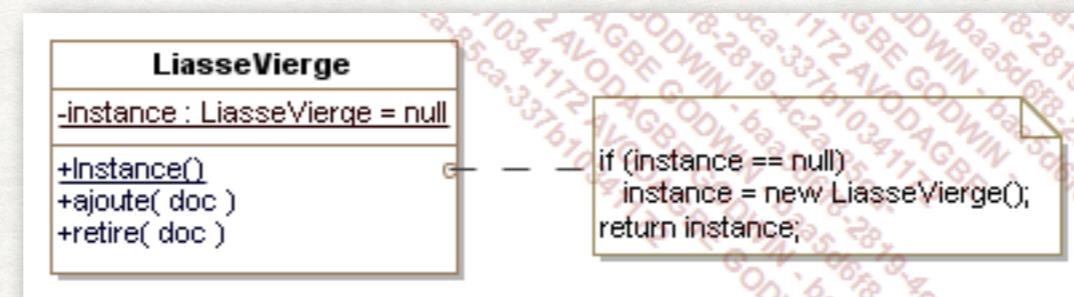
Dans le système de vente en ligne de véhicules, nous devons gérer des classes possédant une seule instance.

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

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

SINGLETON

EXEMPLE

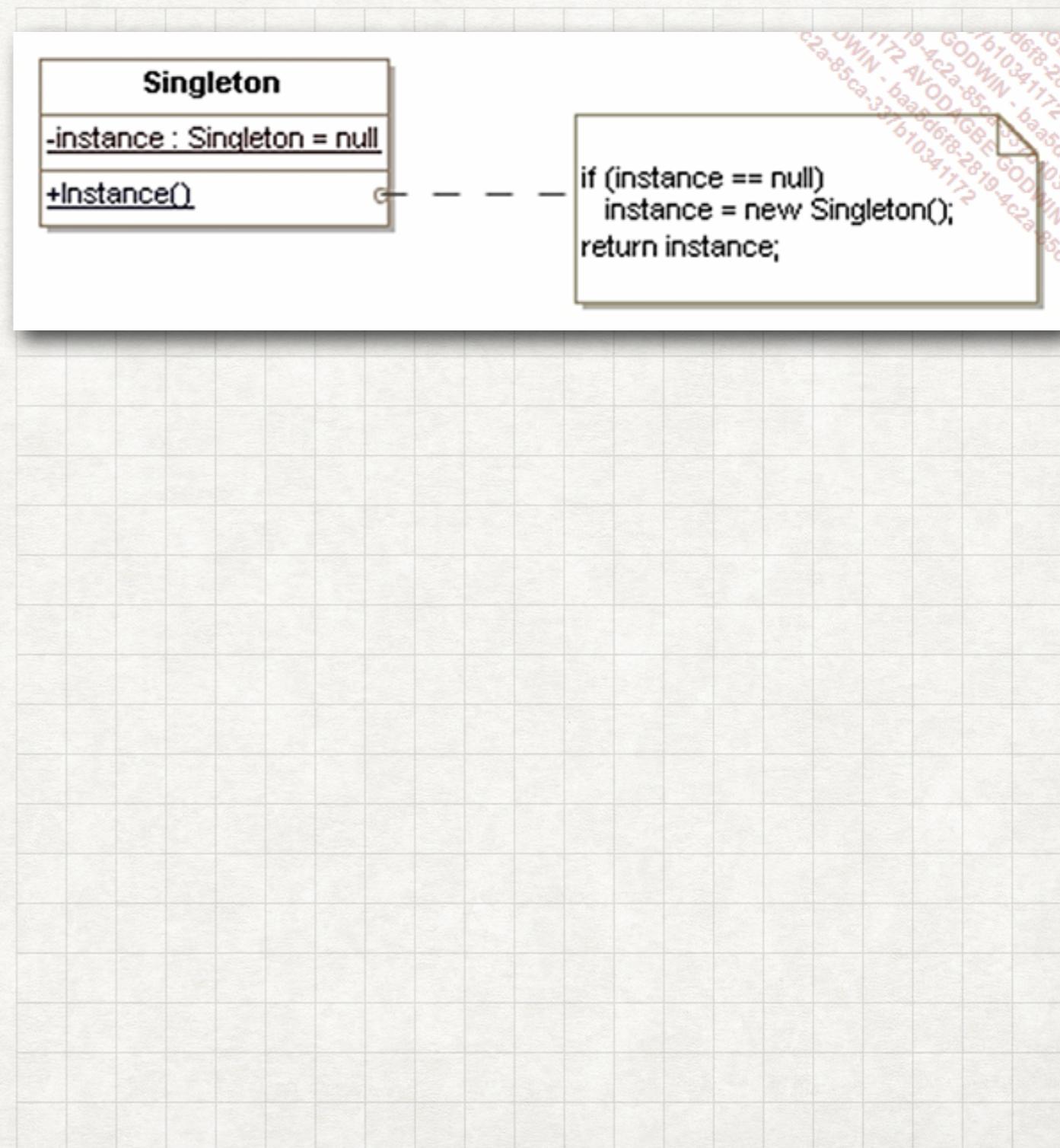


SINGLETON

STRUCTURE

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

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



SINGLETON

DOMAINES D'UTILISATION

Le pattern est utilisé dans les cas suivants :

- 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.

DEMO

PATTERN DE STRUCTURATION

PATTERN DE STRUCTURATION

- Introduction
- Adapter
- Bridge
- Composite
- Facade
- Flyweight
- Proxy



INTRODUCTION

PRESENTATION

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

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

INTRODUCTION

PRESENTATION

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

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

INTRODUCTION

COMPOSITION STATIQUE ET DYNAMIQUE

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

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

PATTERN DE STRUCTURATION ADAPTER



ADAPTER

PRESÉNTATION

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.

ADAPTER

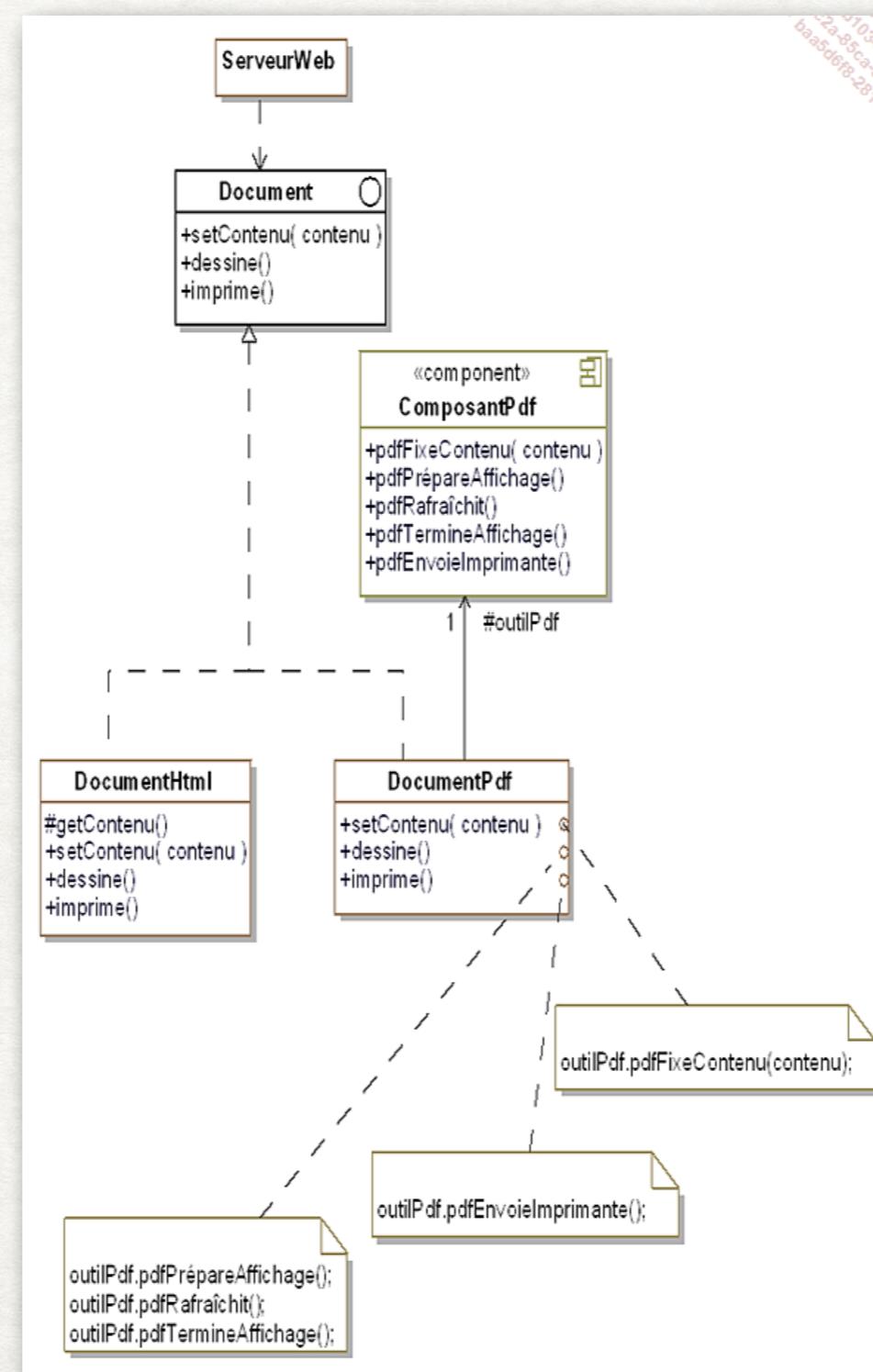
EXEMPLE

Le serveur Web du système de vente de véhicules crée et gère des documents destinés aux clients. L'interface Document a été définie pour cette gestion. Sa représentation UML est montrée à la figure 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.

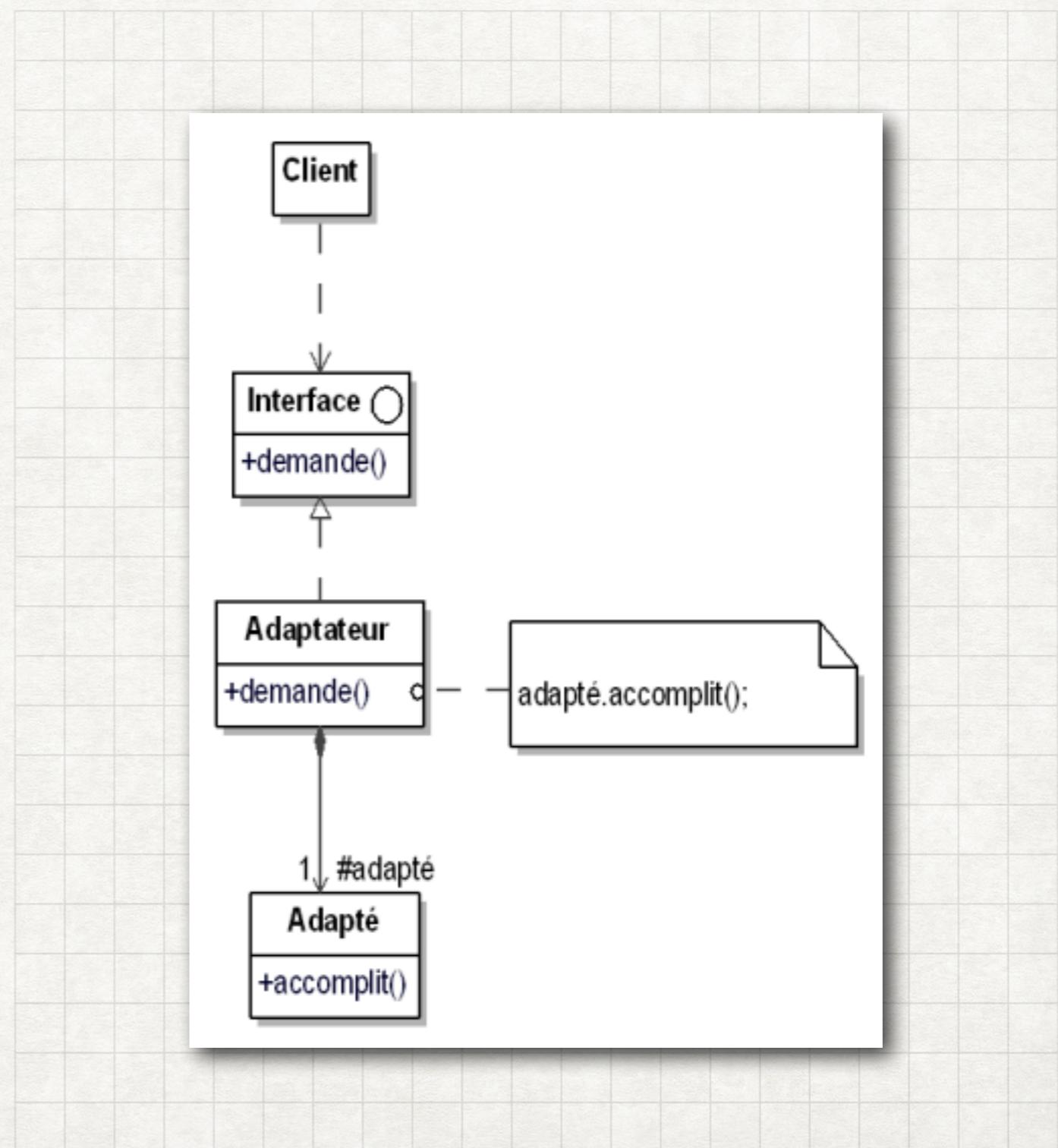
ADAPTER EXAMPLE



ADAPTER STRUCTURE

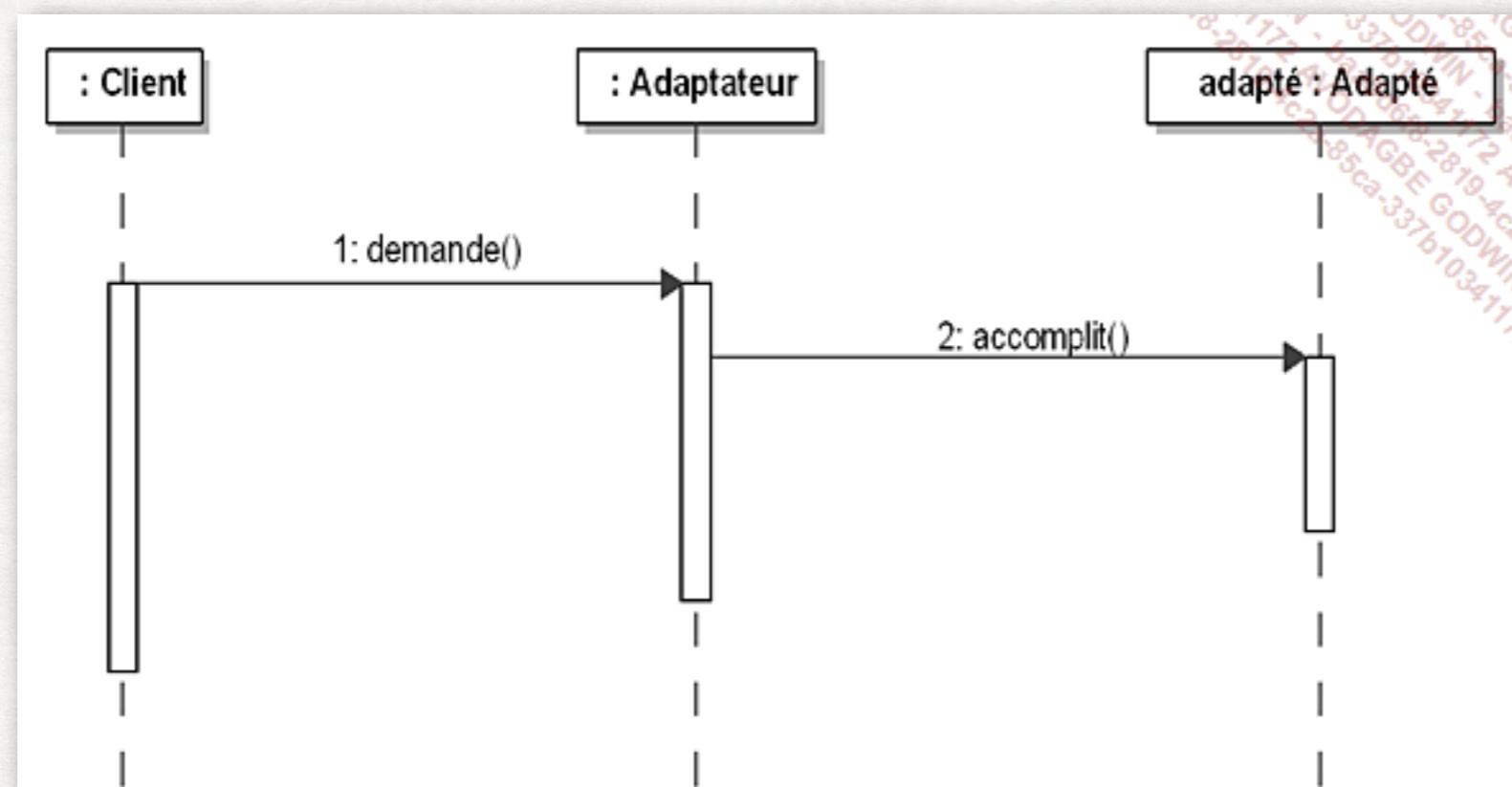
Les participants au pattern sont les suivants :

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



ADAPTER COLLABORATIONS

Le client invoque la méthode demande de l'adaptateur qui, en conséquence, interagit avec l'objet adapté en appelant la méthode accompli.



ADAPTER 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.

DEMO

PATTERN DE STRUCTURATION BRIDGE



BRIDGE

PRESENTATION

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

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

BRIDGE

EXEMPLE

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

Il existe deux implantations des formulaires :

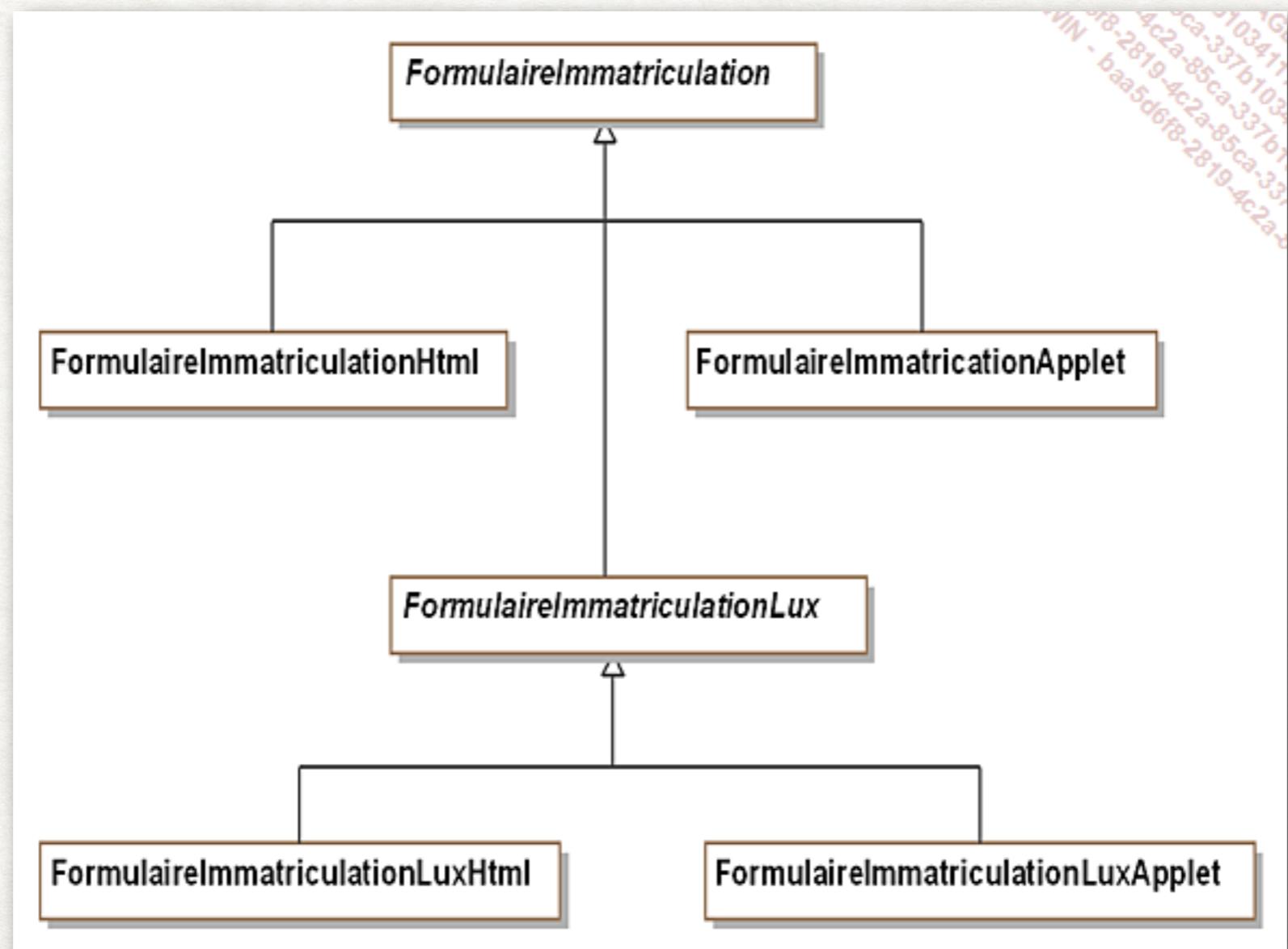
- les formulaires HTML
- les formulaires basés sur une applet.

Il est donc possible d'introduire une classe abstraite `FormulaireImmatriculation` et deux sous-classes concrètes `FormulaireImmatriculationHtml` et `FormulaireImmatriculationApplet`.

Dans un premier temps, les demandes d'immatriculation ne concernaient que la France. Par la suite, il est devenu nécessaire d'introduire une nouvelle sous-classe de `FormulaireImmatriculation` correspondant aux demandes d'immatriculation au Luxembourg, sous-classe appelée `FormulaireImmatriculationLux`. Cette sous-classe doit également être abstraite et avoir également deux sous-classes concrètes pour chaque implantation. La figure ci-dessous montre le diagramme de classes correspondant.

BRIDGE

EXAMPLE



BRIDGE

EXEMPLE

Ce diagramme met en avant deux problèmes : Il existe deux implantations des formulaires :

- 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.

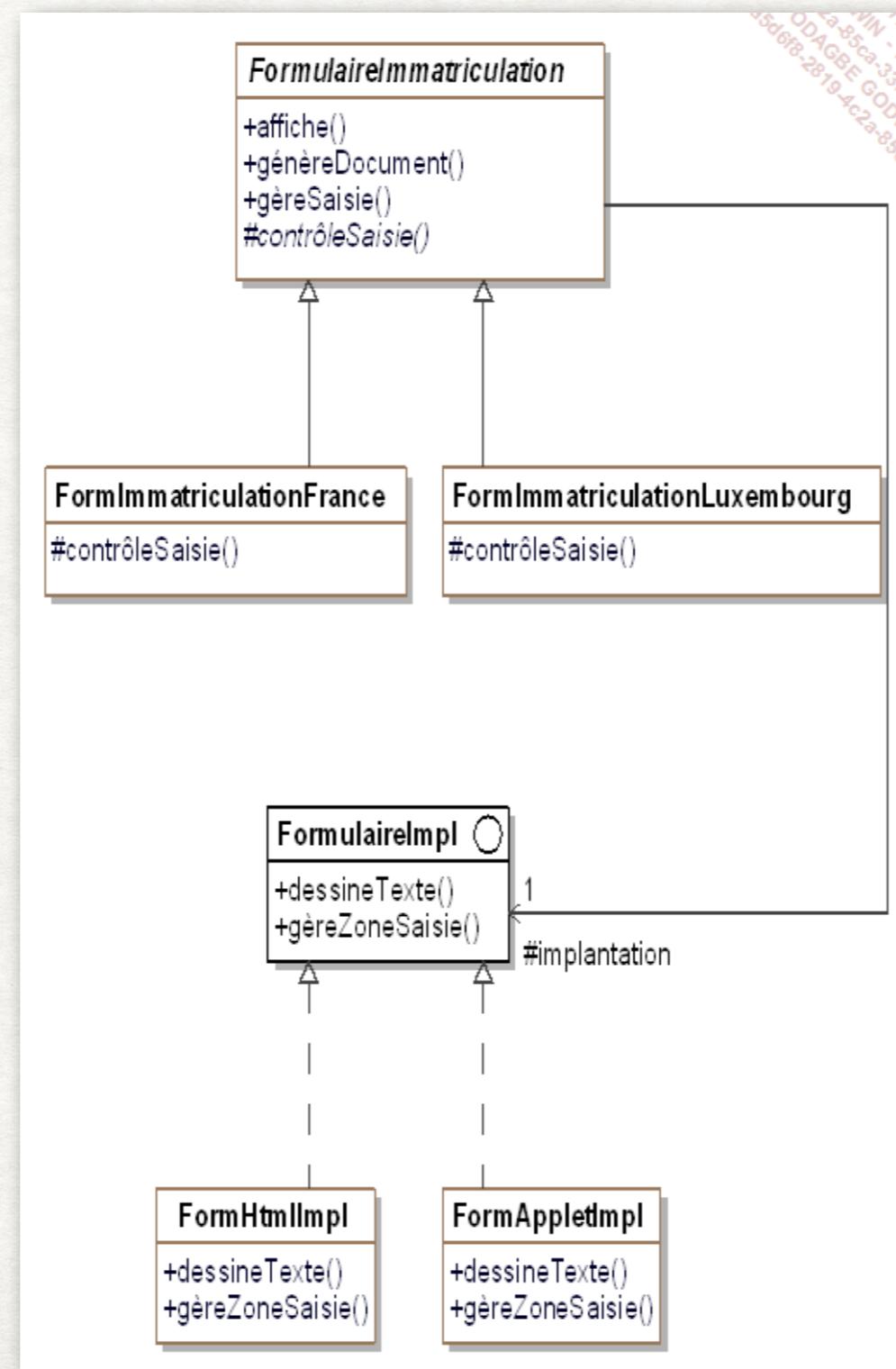
La solution du pattern Bridge consiste donc à séparer les aspects de représentation de ceux d'implantation et à créer deux hiérarchies de classes comme illustré à la figure ci-dessus. Les instances de la classe `FormulaireImmatriculation` détiennent le lien `implantation` vers une instance répondant 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`).

BRIDGE

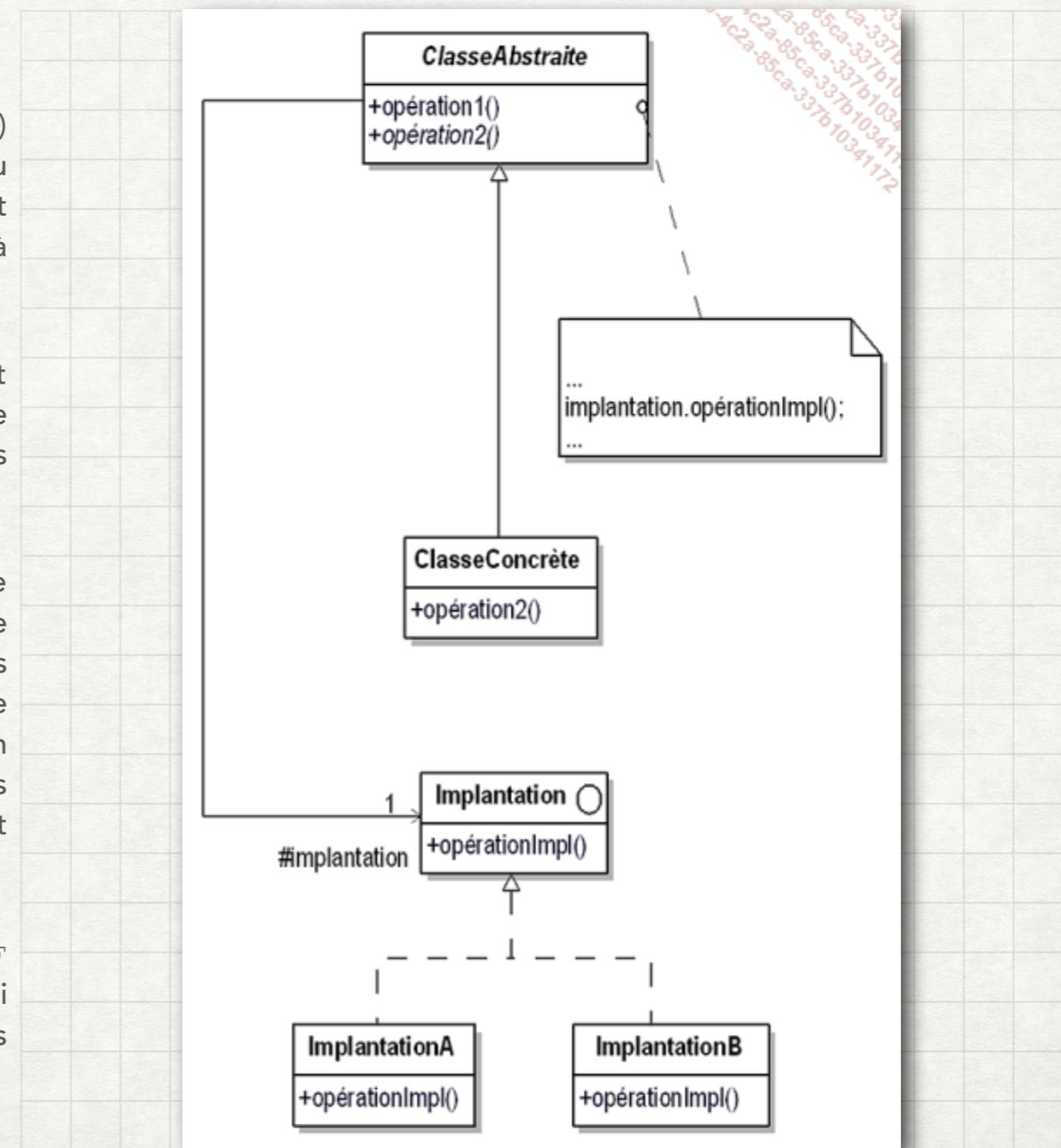
EXAMPLE



BRIDGE STRUCTURE

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`.



BRIDGE

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

DEMO

PATTERN DE STRUCTURATION COMPOSITE



COMPOSITE

PRESENTATION

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.

COMPOSITE

EXEMPLE

Au sein de notre système de vente de véhicules, nous voulons représenter les sociétés clientes, notamment pour connaître le nombre de véhicules dont elles disposent et leur proposer des offres de maintenance de leur parc.

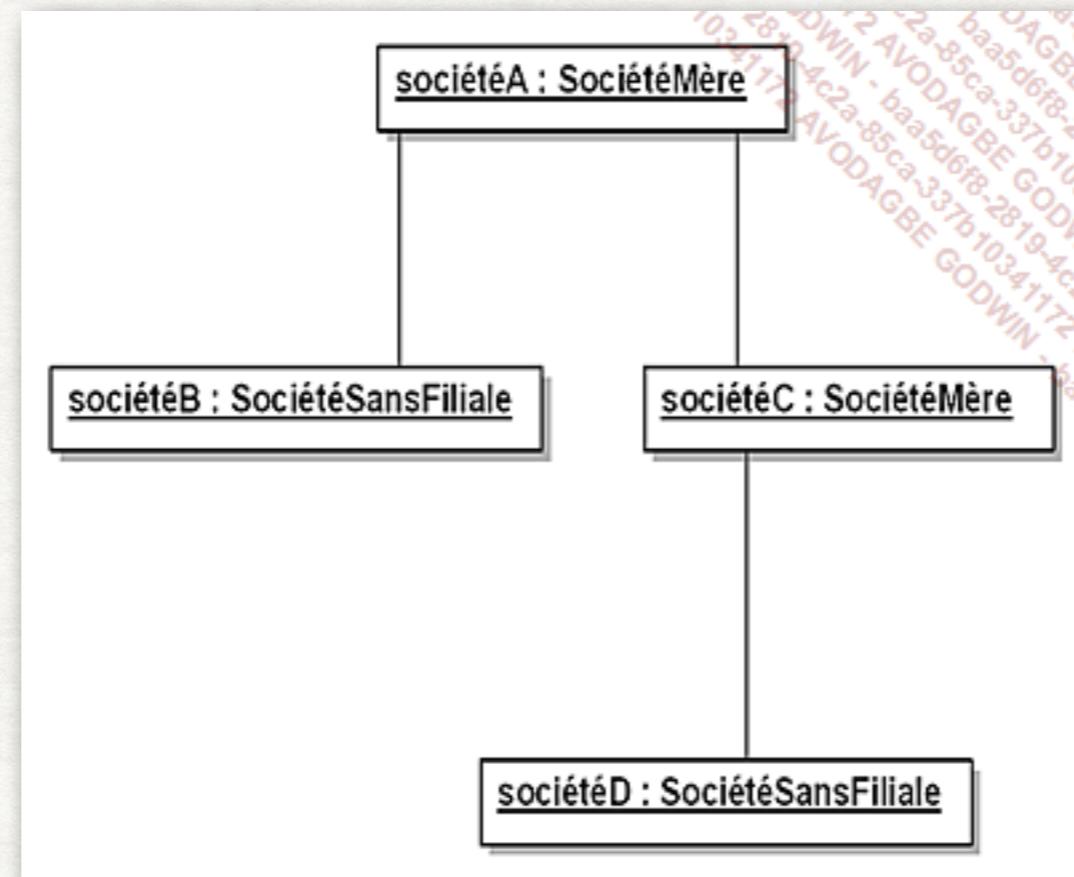
Les sociétés qui possèdent des filiales demandent des offres de maintenance qui prennent en compte le parc de véhicules de leurs filiales.

Une solution immédiate consiste à traiter différemment les sociétés sans filiale et celle possédant des filiales. Cependant cette différence de traitement entre les deux types de société rend l'application plus complexe et dépendante de la composition interne des sociétés clientes.

Le pattern Composite résout ce problème en unifiant l'interface des deux types de sociétés et en utilisant la composition récursive. Cette composition récursive est nécessaire car une société peut posséder des filiales qui possèdent elles-mêmes d'autres filiales. Il s'agit d'une composition en arbre (nous faisons l'hypothèse de l'absence de filiale commune entre deux sociétés) comme illustrée à la figure ci-dessous où les sociétés mères sont placées au-dessus de leurs filiales.

COMPOSITE

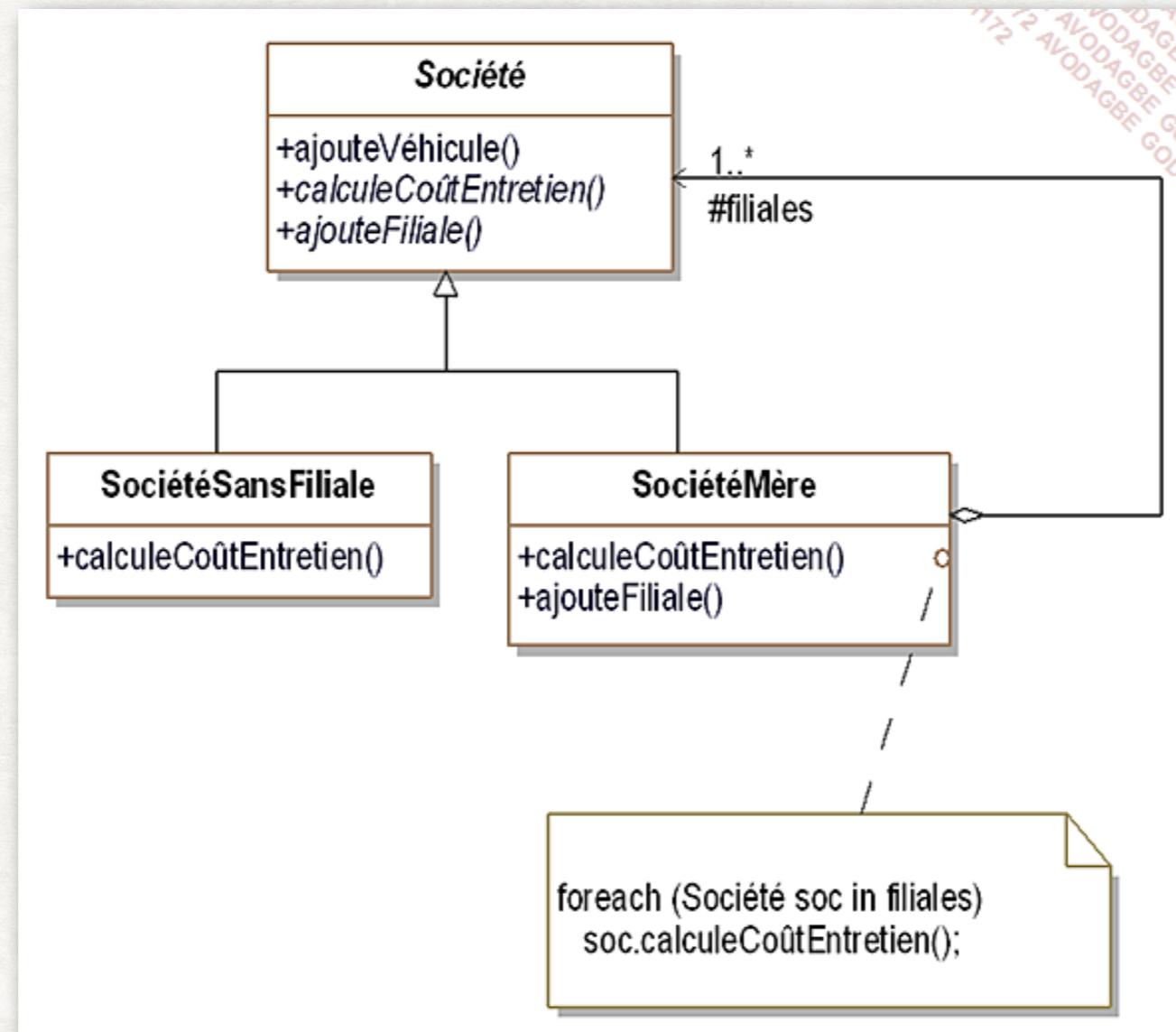
EXAMPLE



La figure suivante 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.

COMPOSITE

EXEMPLE



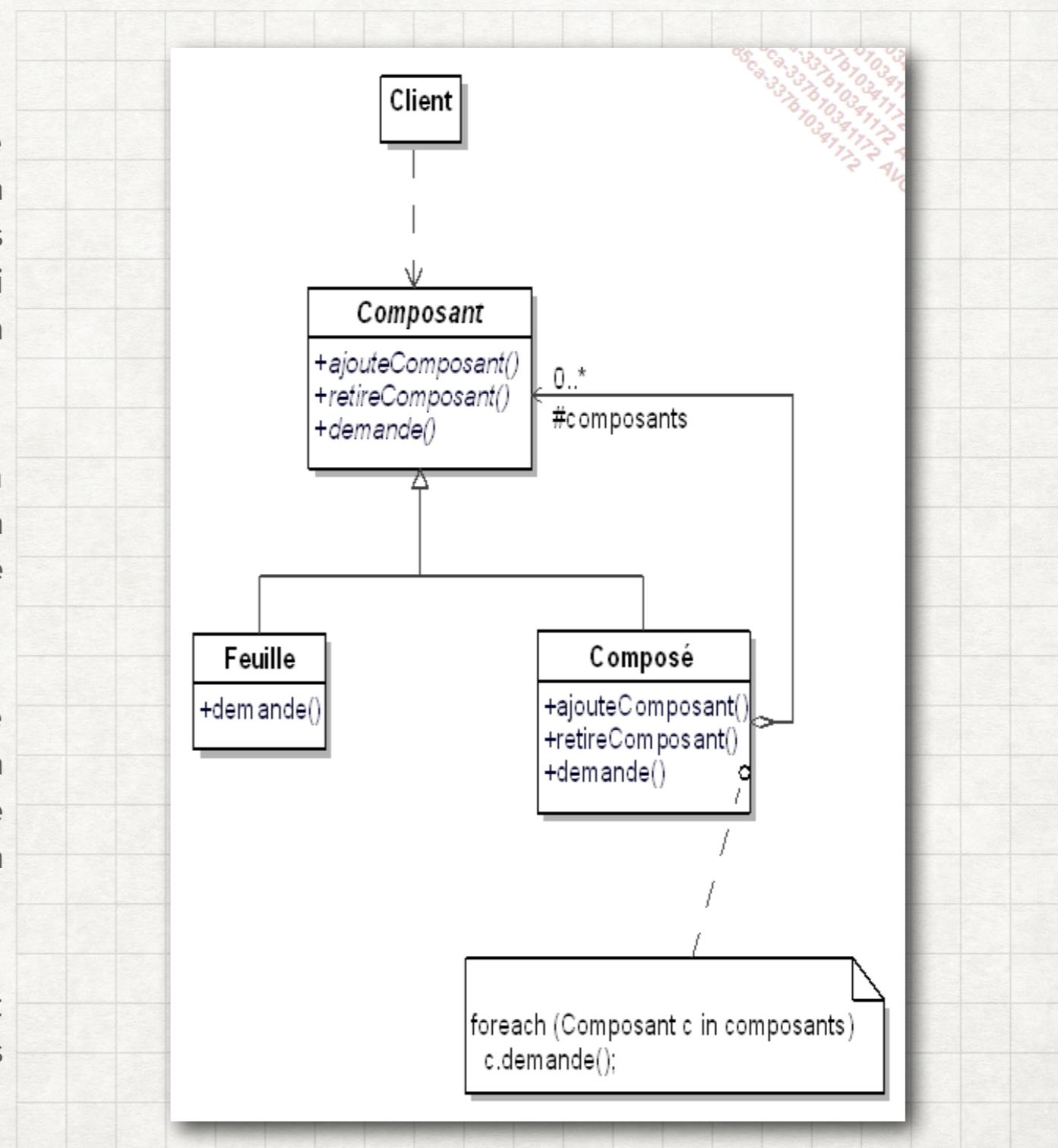
La classe **Société** possède trois méthodes publiques dont une seule est concrète et les deux autres sont abstraites. La méthode concrète est la méthode `ajouteVéhicule` qui ne dépend pas de la composition en filiales de la société. Quant aux deux autres méthodes, elles sont implantées dans les sous-classes concrètes (`ajouteFiliale` ne possède qu'une implantation vide dans **SociétéSansFiliale** donc elle n'est pas représentée dans le diagramme de classes).

COMPOSITE

STRUCTURE

Les participants au pattern sont les suivants :

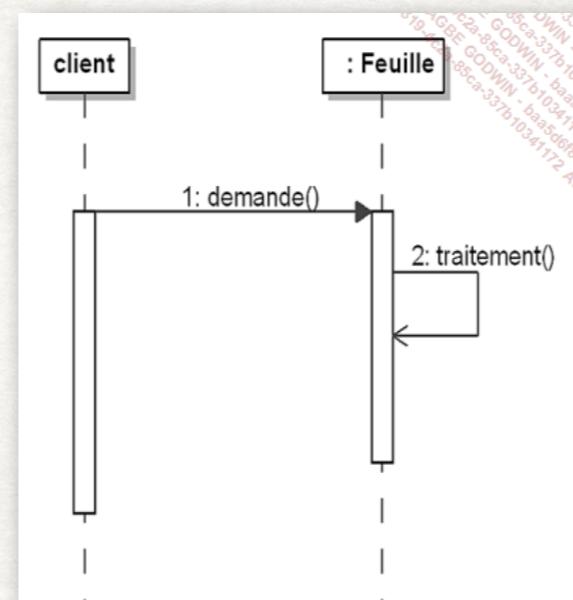
- Composant (Société) est la classe abstraite qui introduit l'interface des objets de la composition, implante les méthodes communes et introduit la signature des méthodes qui gèrent la composition en ajoutant ou en supprimant des composants.
- Feuille (SociétéSansFiliale) est la classe concrète qui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



COMPOSITE COLLABORATIONS

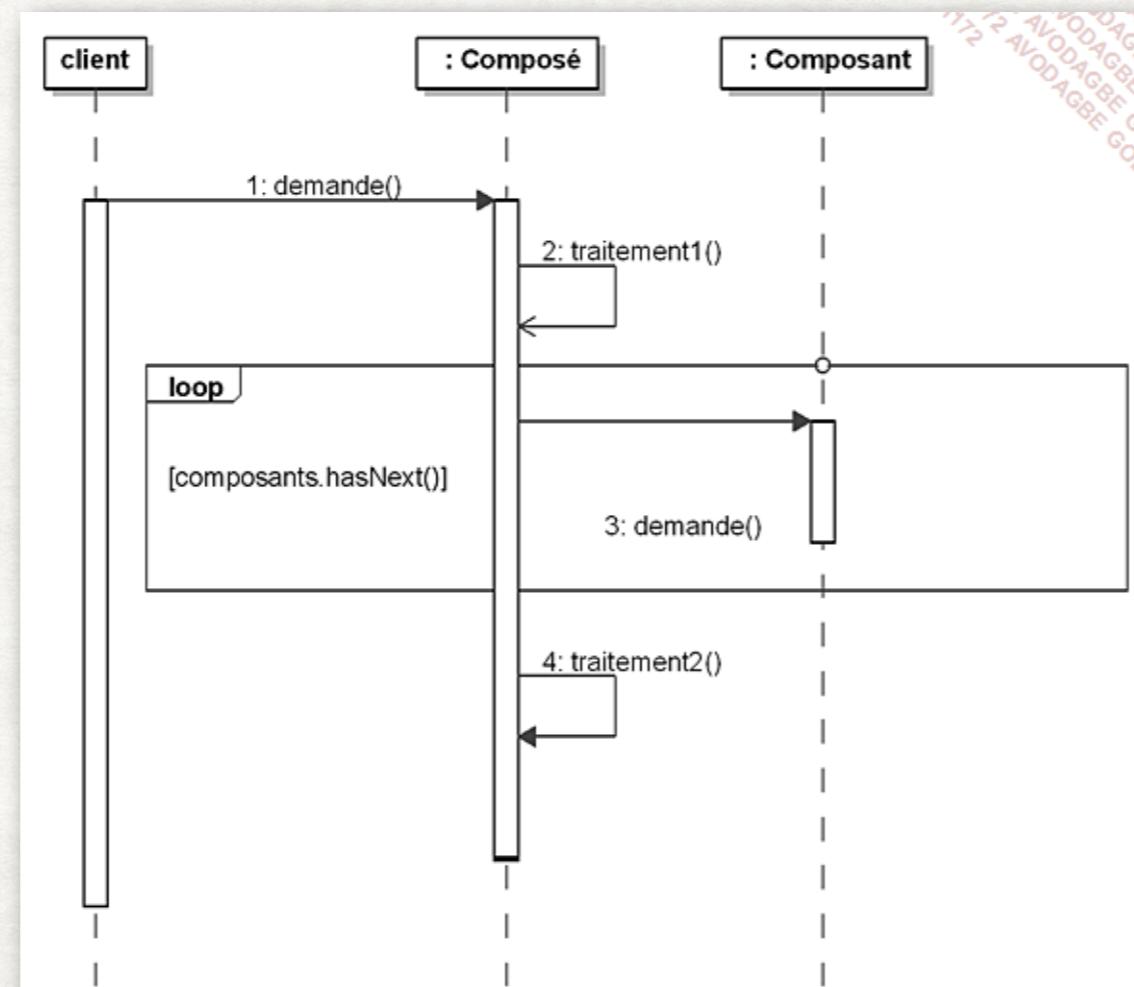
Les clients envoient leurs requêtes aux composants au travers de l'interface de la classe Composant.

Lorsqu'un composant reçoit une requête, il réagit en fonction de sa classe. Si le composant est une feuille, il traite la requête comme illustré à la figure ci-dessous.



COMPOSITE COLLABORATIONS

Si le composant est une instance de la classe Composé, il effectue un traitement préalable puis généralement envoie un message à chacun de ses composants puis effectue un traitement postérieur. La figure 12.5 illustre ce cas avec l'appel récursif à d'autres composants qui vont, à leur tour, traiter cet appel soit comme feuille, soit comme composé.



COMPOSITE

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.

DEMO

PATTERN DE STRUCTURATION DECORATOR



DECORATOR

PRESENTATION

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.

DECORATOR

EXEMPLE

Le système de vente de véhicules dispose d'une classe `VueCatalogue` qui affiche, sous la forme d'un catalogue électronique, les véhicules disponibles sur une page Web.

Nous voulons maintenant afficher des données supplémentaires pour les véhicules "haut de gamme", à savoir les informations techniques liées au modèle. Pour réaliser l'ajout de cette fonctionnalité, nous pouvons réaliser une sous-classe d'affichage spécifique pour les véhicules "haut de gamme". Maintenant, nous voulons afficher le logo de la marque des véhicules "moyen et haut de gamme". Il convient alors d'ajouter une nouvelle sous-classe pour ces véhicules, surclasse de la classe des véhicules "haut de gamme", ce qui devient vite complexe. Il est aisément 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.

DECORATOR

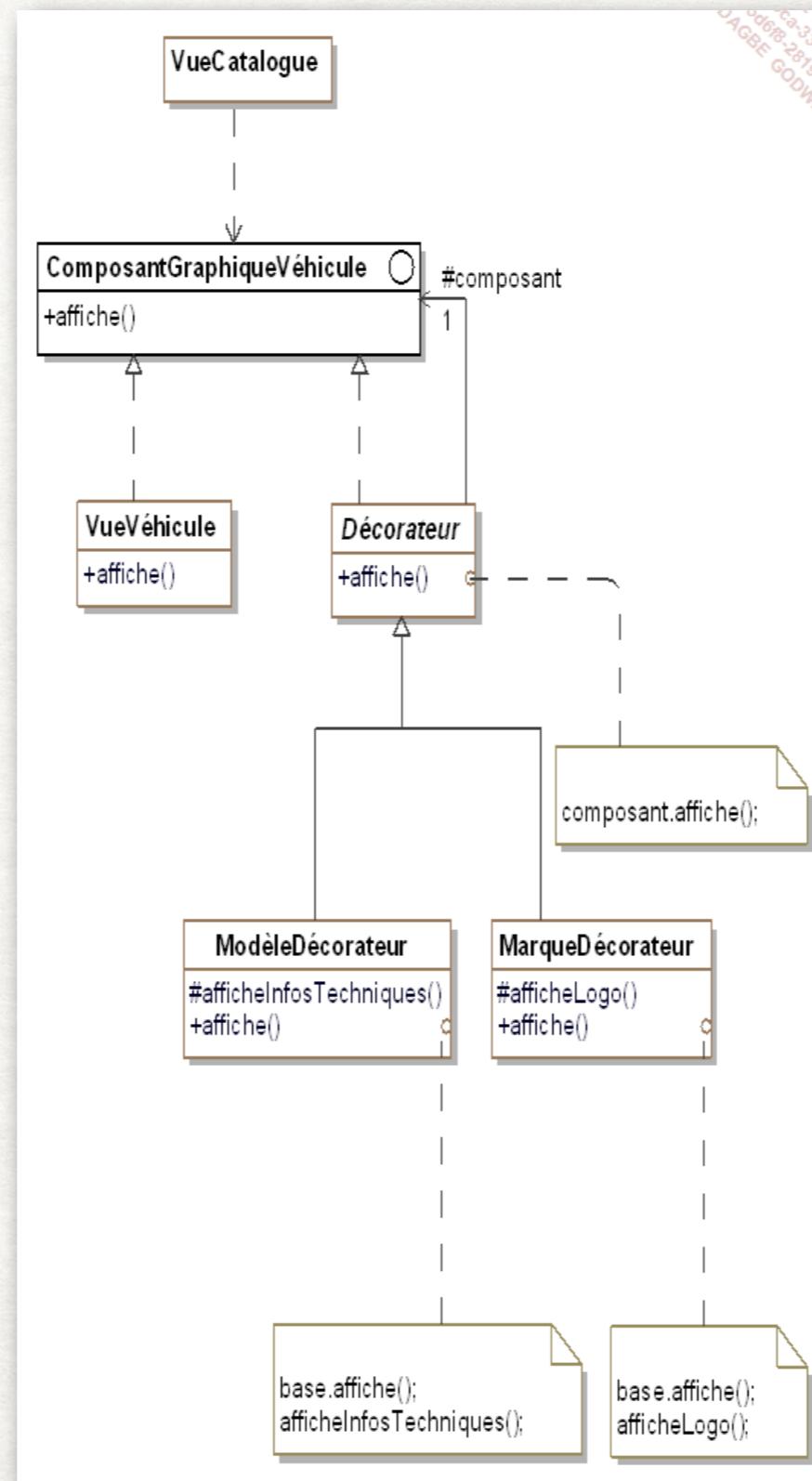
EXEMPLE

La figure suivante 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 délègue l'affichage vers ce composant.

DECORATOR

EXEMPLE

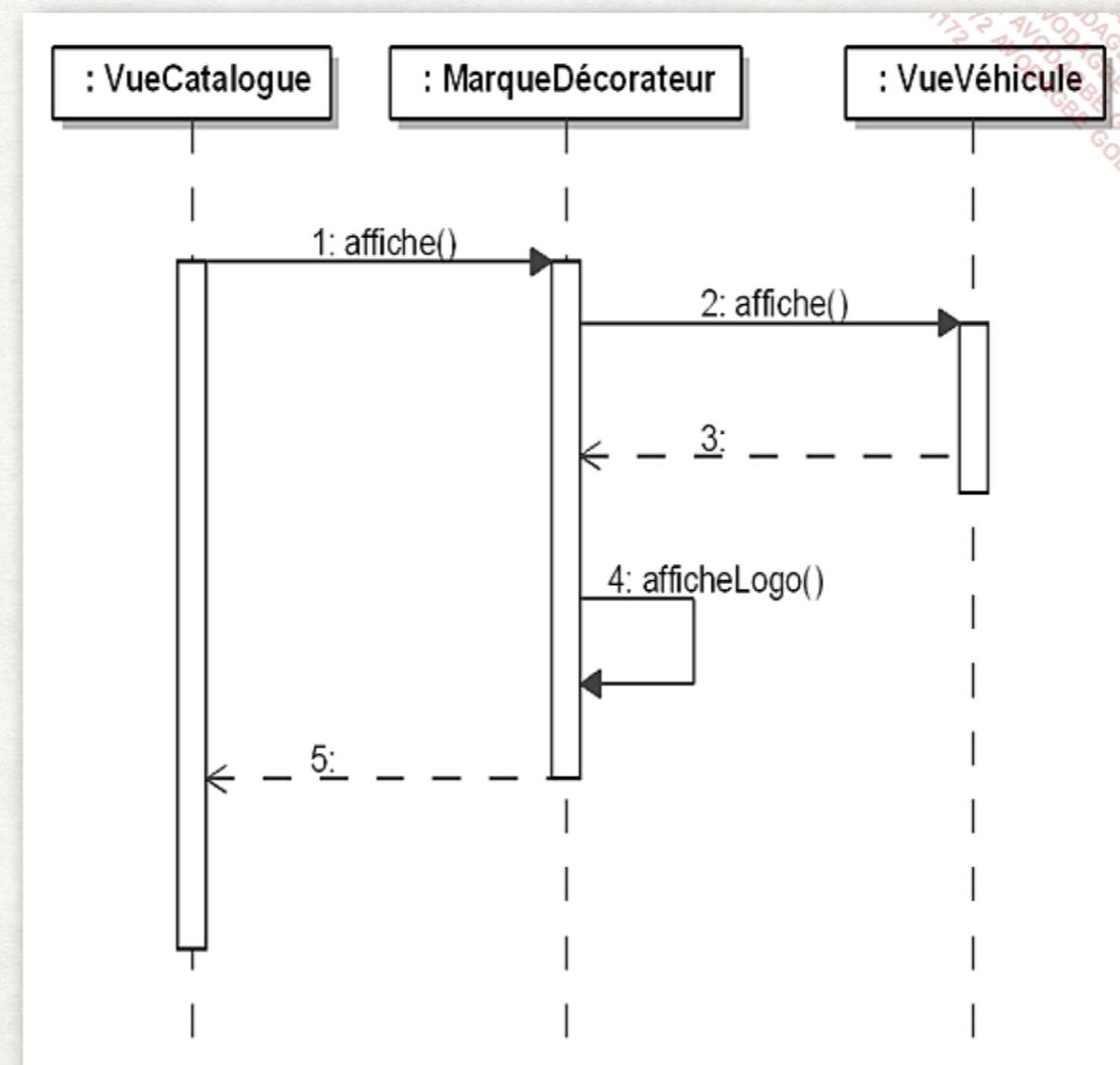


DECORATOR

EXEMPLE

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.

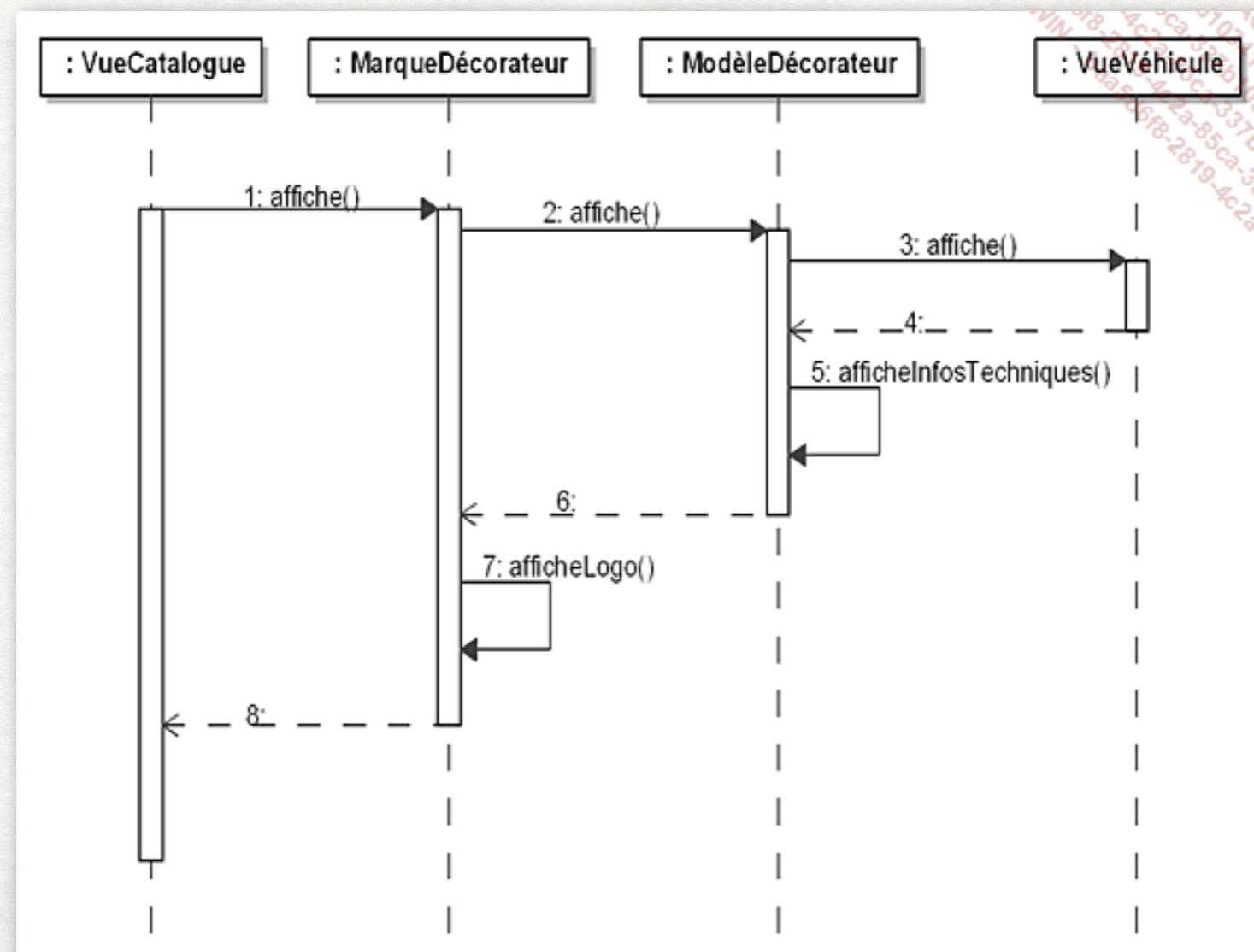


DECORATOR

EXEMPLE

La figure 13.3 montre la séquence des appels de message destinés à l'affichage d'un véhicule pour lequel les informations techniques du modèle et le logo de la marque sont également affichés.

Cette figure illustre bien le fait que les décorateurs sont des composants puisqu'ils peuvent devenir le composant d'un nouveau décorateur, ce qui donne lieu à une chaîne de décorateurs. Cette possibilité de chaîne dans laquelle il est possible d'ajouter ou de retirer dynamiquement un décorateur procure une grande souplesse.

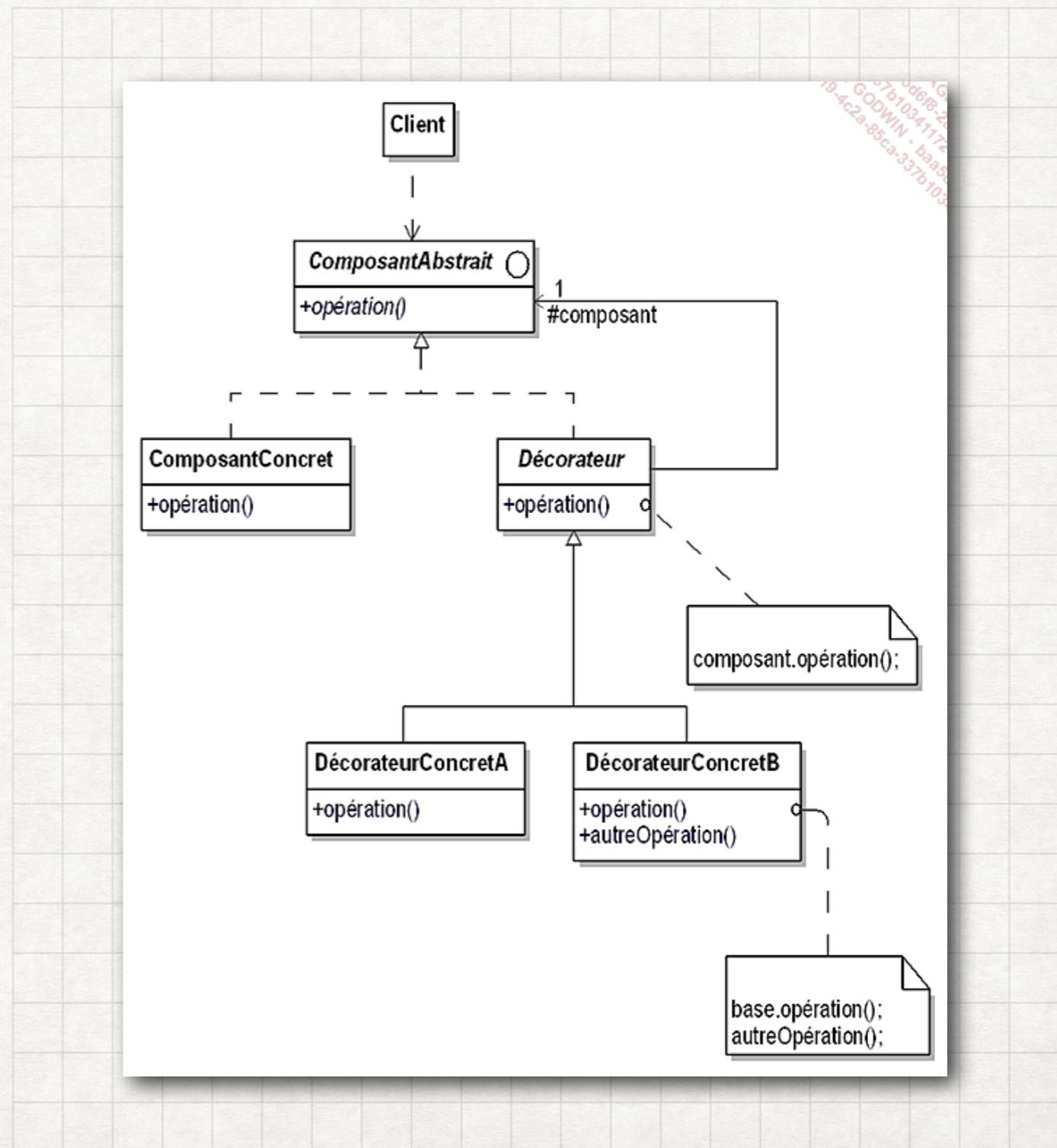


DECORATOR

STRUCTURE

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.



DECORATOR

DOMAINES D'UTILISATION

Le pattern est utilisé dans les cas 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.

DEMO

PATTERN DE STRUCTURATION FACADE



FACADE

PRESÉNTATION

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.

FACADE

EXAMPLE

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.

FACADE

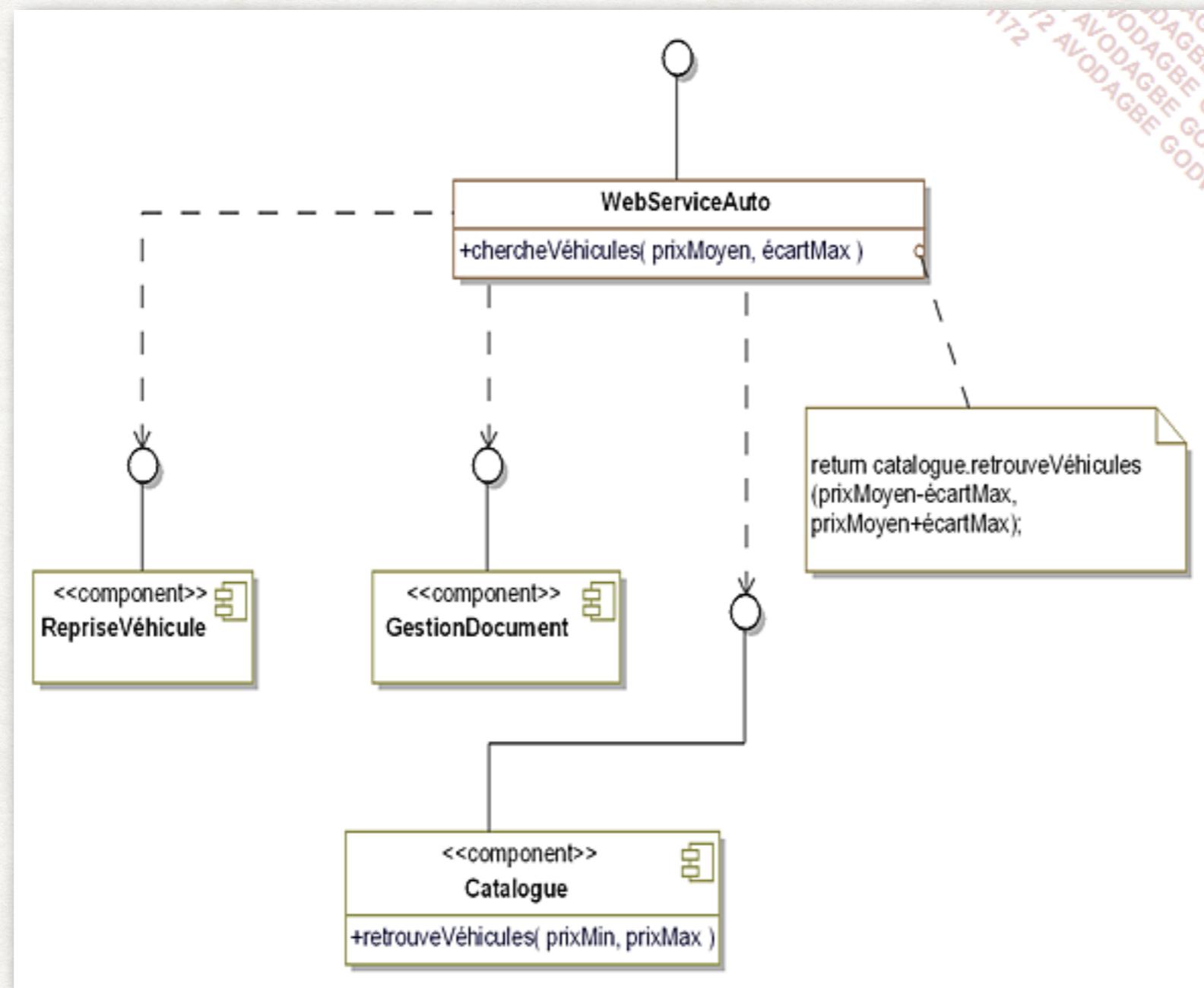
EXAMPLE

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.

FACADE

EXAMPLE

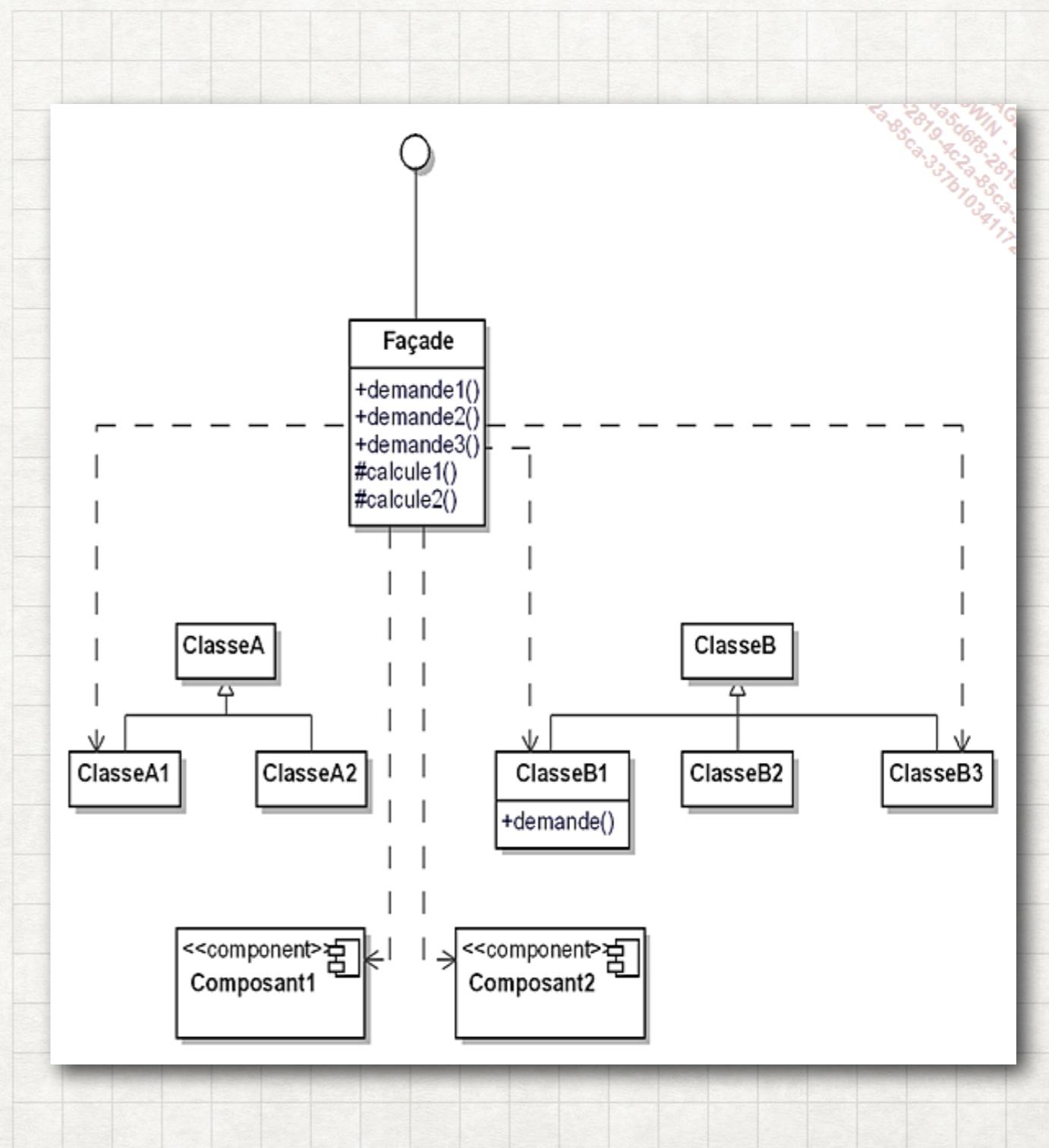


FACADE

STRUCTURE

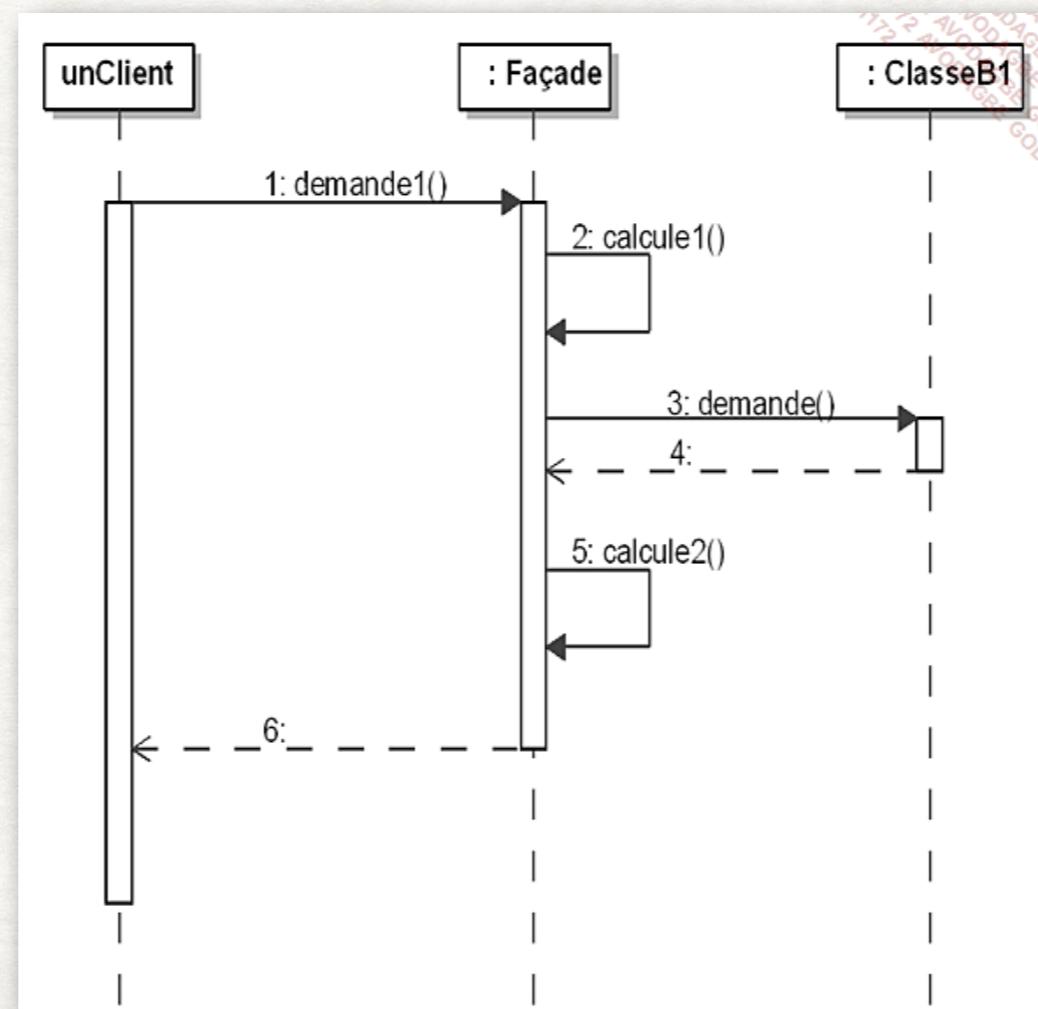
Les participants au pattern sont les suivants :

- 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.



COMPOSITE 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`).



FACADE

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.

DEMO

PATTERN DE STRUCTURATION FLYWEIGHT



FLYWEIGHT PRESENTATION

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

FLYWEIGHT

EXEMPLE

Dans le système de vente de véhicules, il est nécessaire de gérer les options que l'acheteur peut choisir lorsqu'il commande un nouveau véhicule.

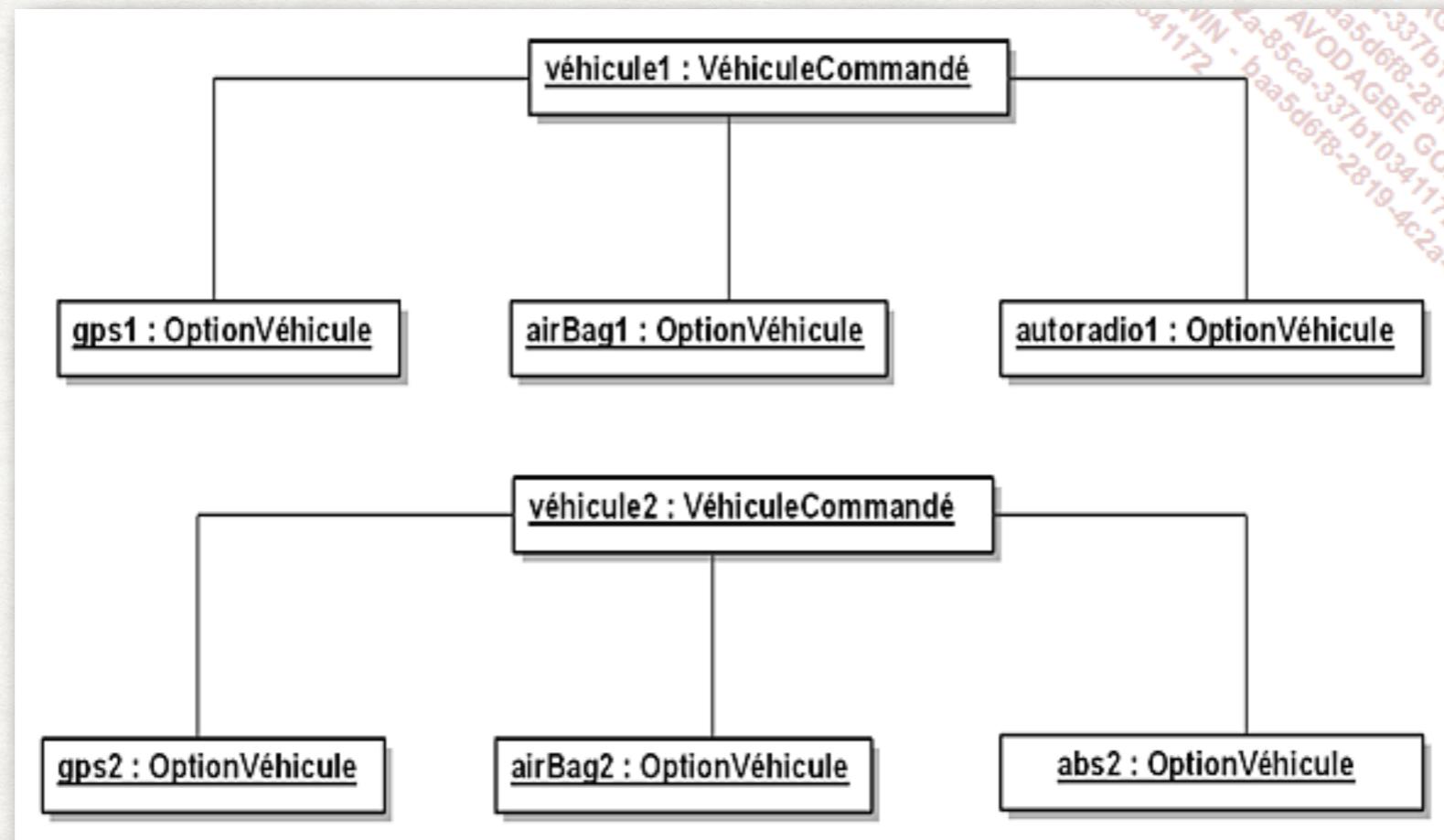
Ces options sont décrites par la classe `OptionVéhicule` qui contient plusieurs attributs comme le nom, l'explication, un logo, le prix standard, les incompatibilités avec d'autres options, avec certains modèles, etc.

Pour chaque véhicule commandé, il est possible d'associer une nouvelle instance de cette classe. Cependant un grand nombre d'options sont souvent présentes pour chaque véhicule commandé, ce qui oblige le système à gérer un grand ensemble d'objets de petite taille (de grain fin). Cette approche présente toutefois l'avantage de pouvoir stocker au niveau de l'option des informations spécifiques à celle-ci et au véhicule comme le prix de vente de l'option qui peut différer d'un véhicule commandé à un autre.

Cette solution est présentée sur un petit exemple à la figure ci-dessous et il est aisément 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.

FLYWEIGHT

EXEMPLE



FLYWEIGHT

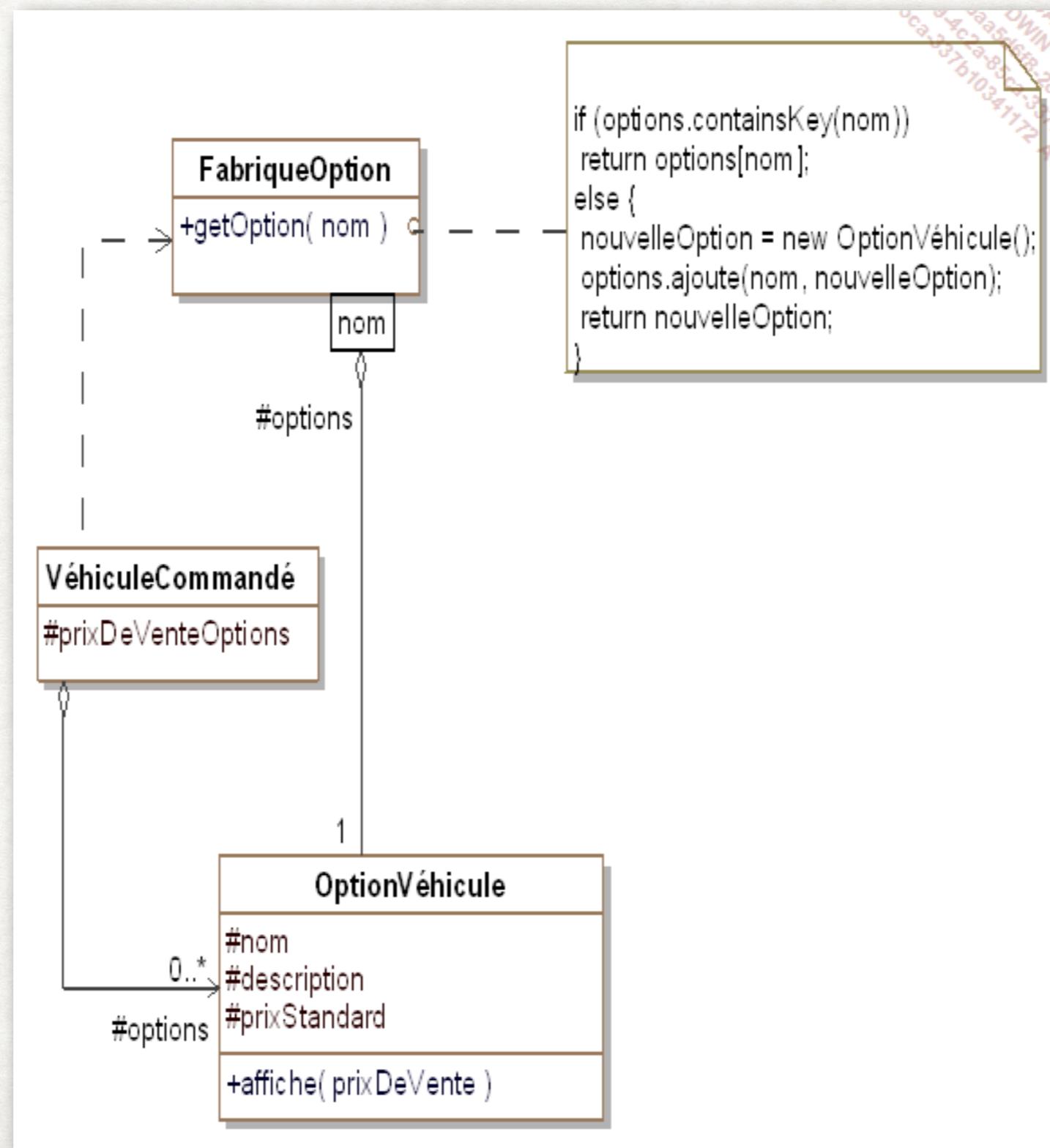
EXEMPLE

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.

FLYWEIGHT

EXEMPLE



FLYWEIGHT

EXEMPLE

Ce diagramme introduit les classes suivantes :

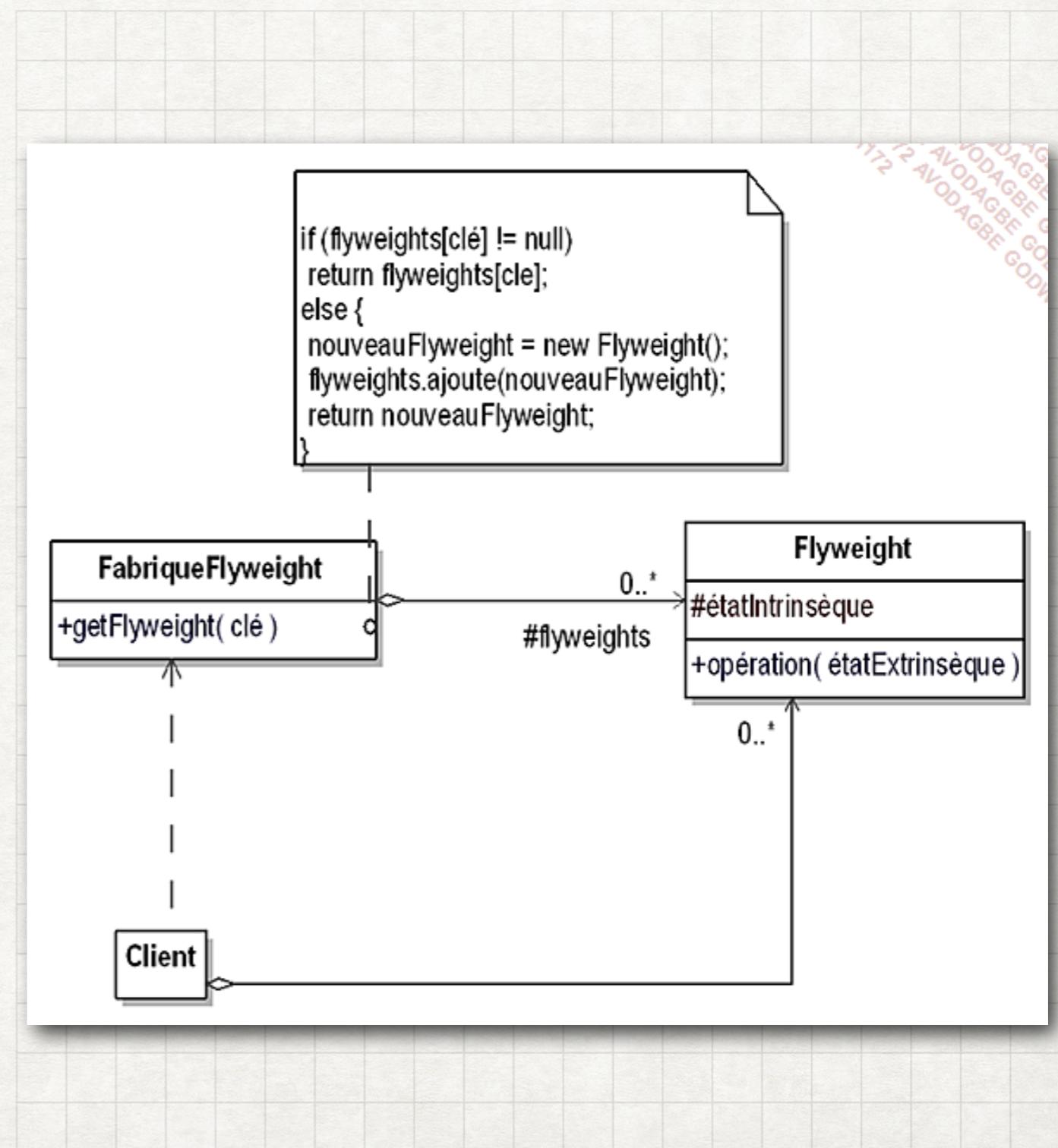
- OptionVéhicule dont les attributs détiennent l'état intrinsèque d'une option. La méthode `affichea` pour paramètre `prixDeVente` qui représente l'état extrinsèque d'une option ;
- FabriqueOption dont la méthode `getOption` renvoie une option à partir de son nom. Son fonctionnement consiste à rechercher 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.

FLYWEIGHT

STRUCTURE

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.



FLYWEIGHT 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.

FLYWEIGHT

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.

DEMO

PATTERN DE STRUCTURATION PROXY



PROXY PRESENTATION

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.

PROXY

EXAMPLE

Nous voulons offrir pour chaque véhicule du catalogue la possibilité de visualiser un film qui présente ce véhicule. Un clic sur la photo de la présentation du véhicule permet de jouer ce film.

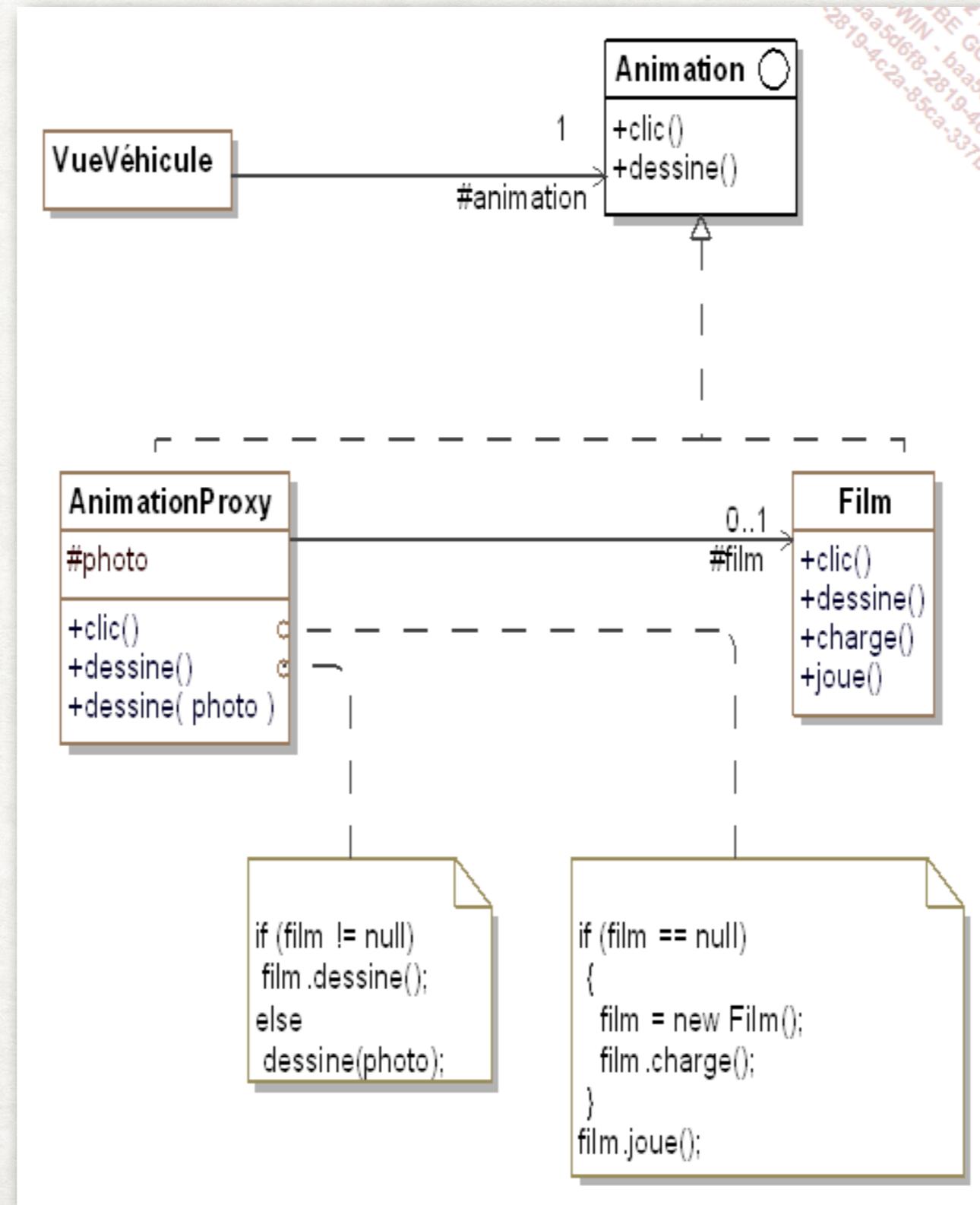
Une page du catalogue contient de nombreux véhicules et il est très lourd de créer en mémoire tous les objets d'animation car les films nécessitent une grande quantité de mémoire et leur transfert au travers d'un réseau prend beaucoup de temps.

Le pattern Proxy offre une solution à ce problème en différant la création des sujets jusqu'au moment où le système a besoin d'eux, ici lors du clic sur la photo du véhicule. Cette solution apporte deux avantages :

- la page du catalogue est chargée beaucoup plus rapidement surtout si elle doit être chargée au travers d'un réseau comme Internet ;
- seuls les films devant être visualisés sont créés, chargés et joués.
- L'objet photo est appelé le proxy du film. Il se substitue au film pour l'affichage. Il procède à la création du sujet uniquement lors du clic. Il possède la même interface que l'objet film. La figure 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`.

PROXY

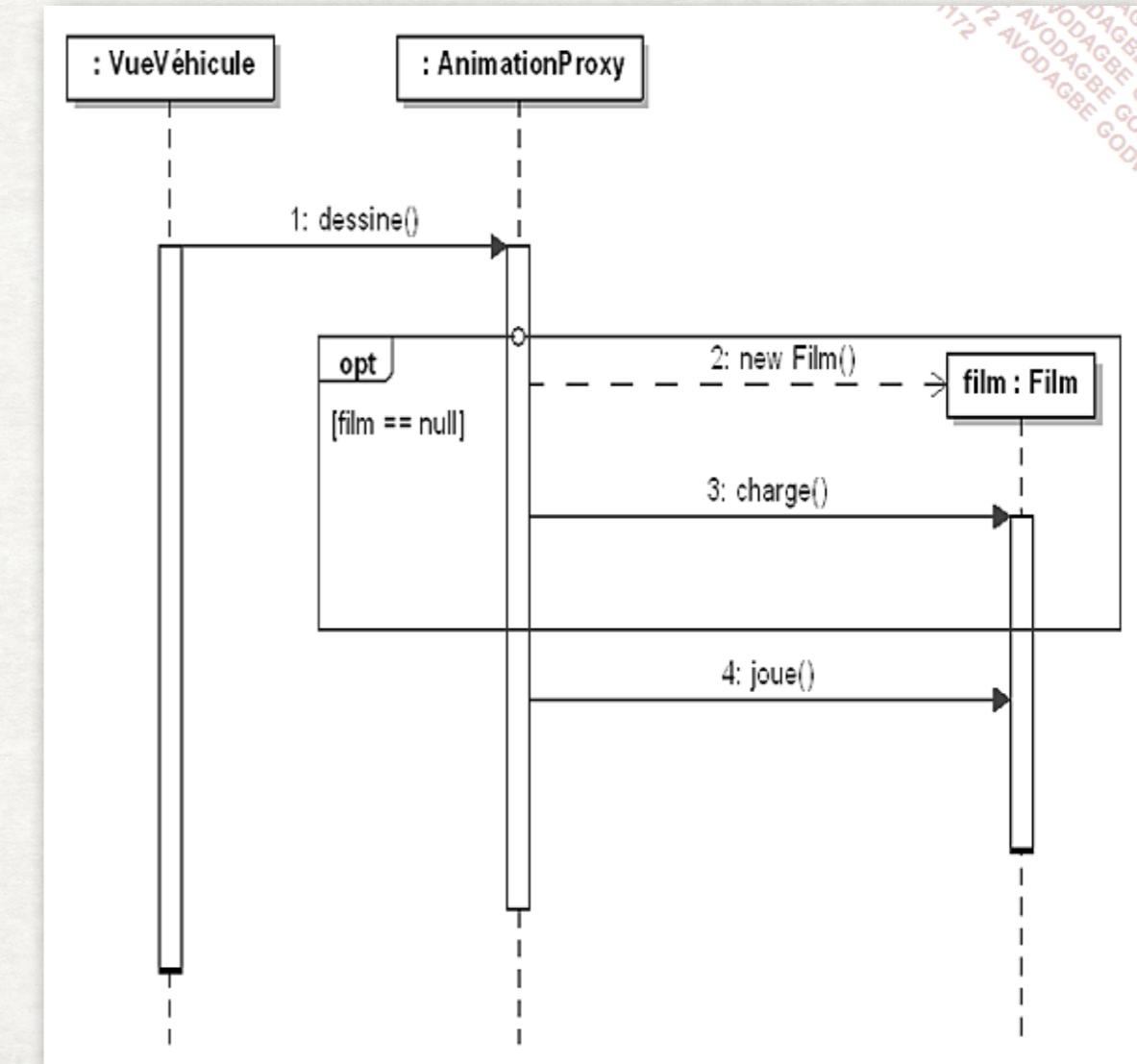
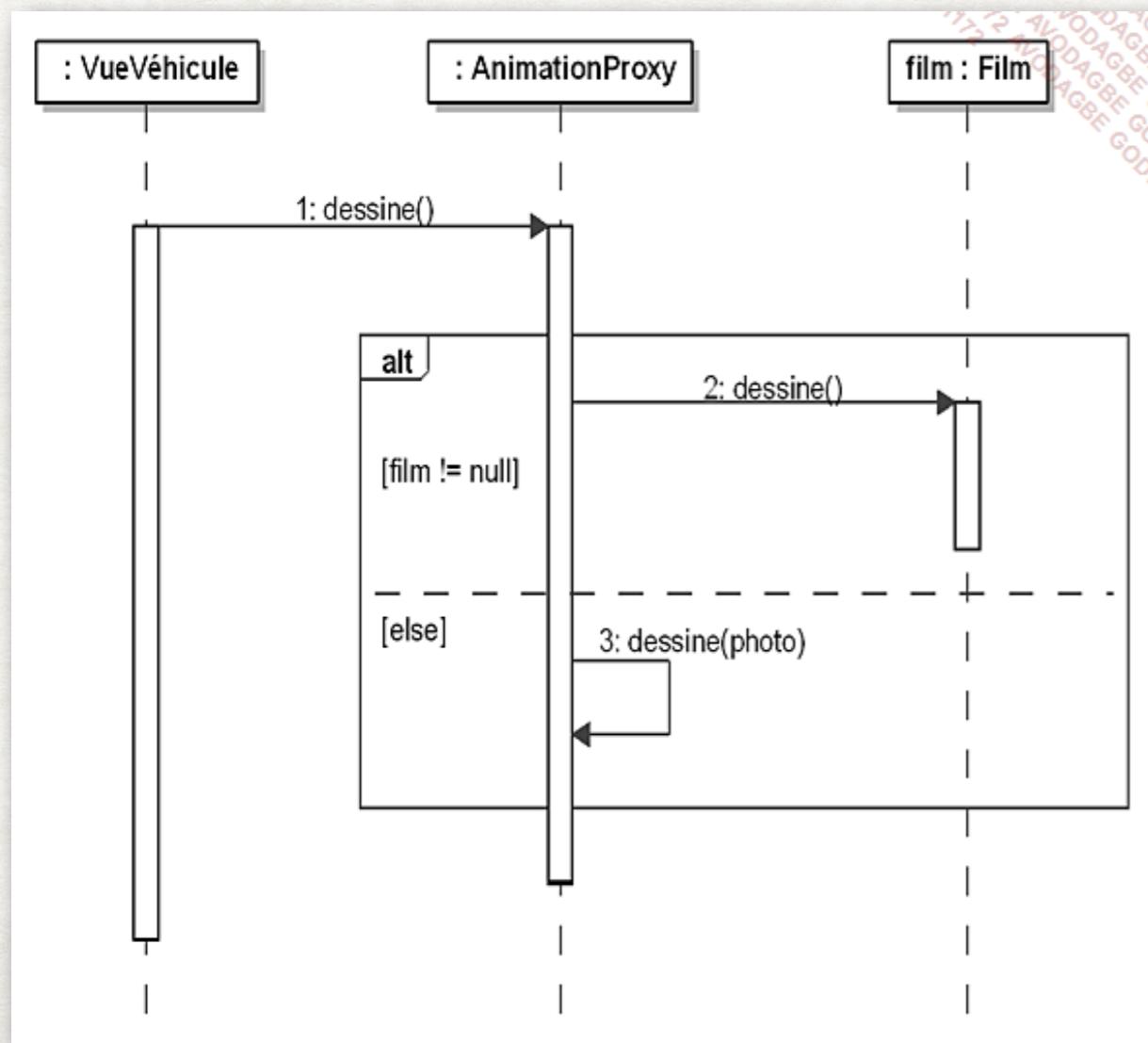
EXAMPLE



PROXY

EXAMPLE

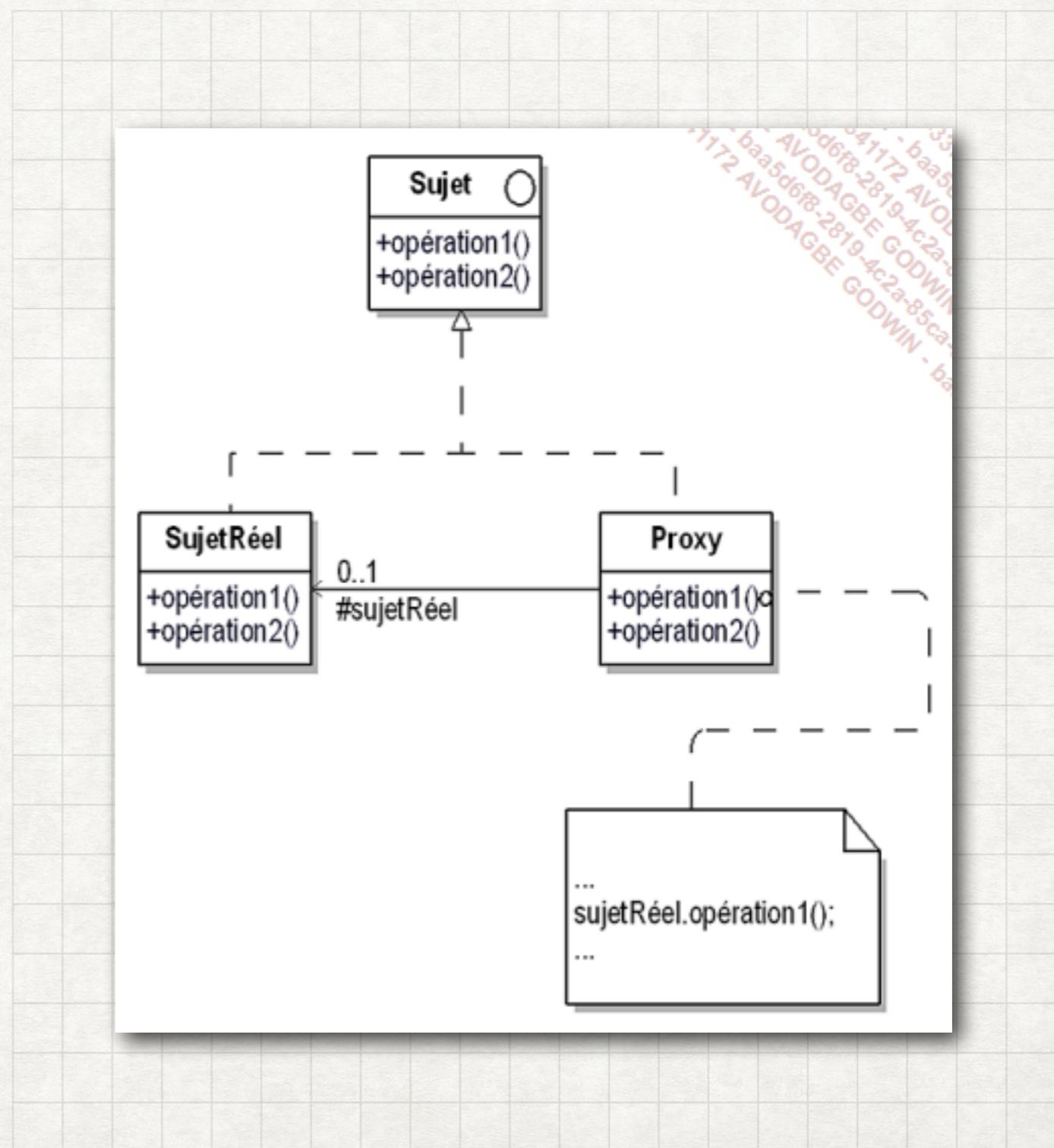
Quand le proxy reçoit le message dessine, il affiche le film si celui-ci a déjà été créé et chargé. Quand le proxy reçoit le message clic, il joue le film après l'avoir préalablement créé et chargé. Le diagramme de séquence pour le message clic est détaillé aux figures ci-dessous pour le message dessine.



PROXY STRUCTURE

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.



PROXY COLLABORATIONS

Le proxy reçoit les appels du client à la place du sujet réel. Quand il le juge approprié, il délègue ces messages au sujet réel. Il doit, dans ce cas, créer préalablement le sujet réel si ce n'est déjà fait.

PROXY

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.

DEMO

PATTERN DE COMPORTEMENT

PATTERN DE COMPORTEMENT

- ⌚ Introduction
- ⌚ Chain of Responsibility
- ⌚ Command
- ⌚ Interpreter
- ⌚ Iterator
- ⌚ Mediator
- ⌚ Memento
- ⌚ Observer
- ⌚ State
- ⌚ Strategy
- ⌚ Template Method
- ⌚ Visitor



INTRODUCTION

PRESENTATION

Le concepteur d'un système d'objets est souvent confronté au problème de la découverte des objets. Celle-ci peut être réalisée à partir des deux aspects suivants :

- la structuration des données ;
- la distribution des traitements et des algorithmes.

Les patterns de structuration apportent des solutions aux problèmes de structuration des données et des objets.

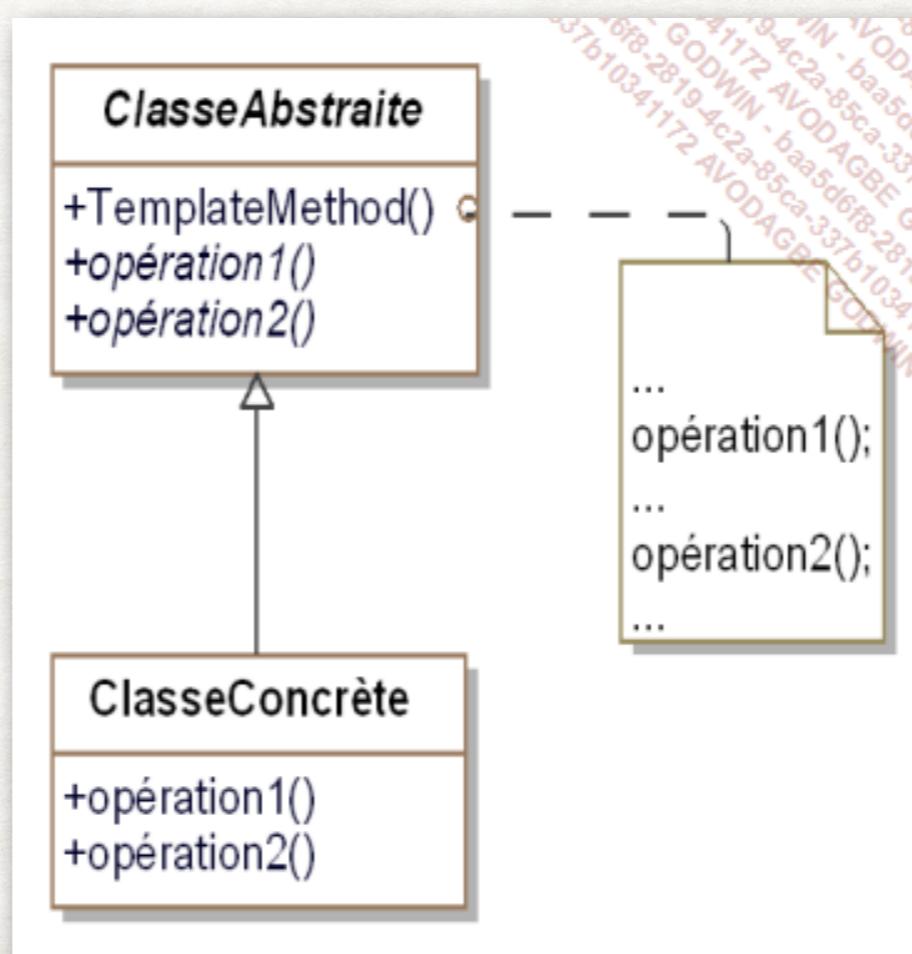
L'objectif des patterns de comportement est de fournir des solutions pour distribuer les traitements et les algorithmes entre les objets.

Ces patterns organisent les objets ainsi que leurs interactions en spécifiant les flux de contrôle et de traitement au sein d'un système d'objets.

INTRODUCTION

DISTRIBUTION PAR HÉRITAGE OU PAR DÉLÉGATION

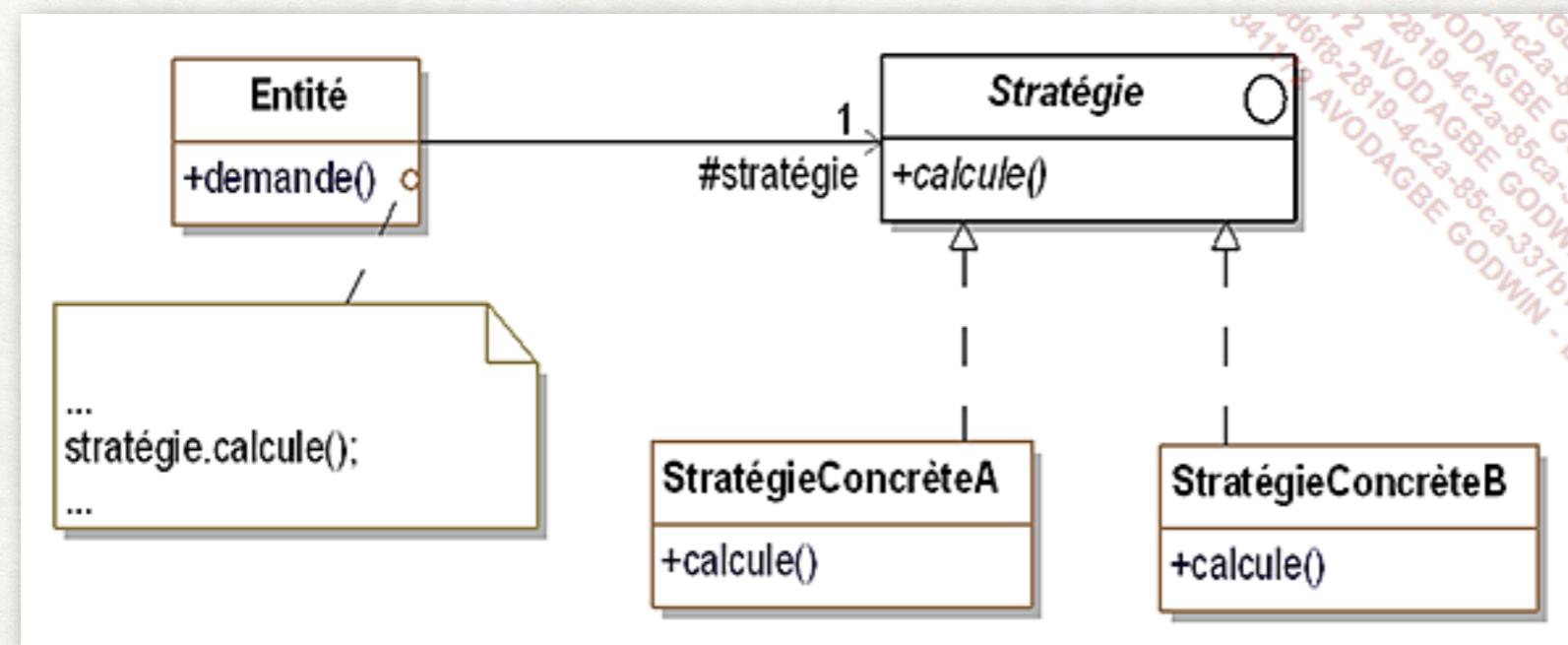
Une première approche pour distribuer un traitement est de le répartir dans les sous-classes. Cette répartition se fait par l'utilisation dans la classe de méthodes abstraites qui sont implantées dans les sous-classes. Comme une classe peut posséder plusieurs sous-classes, cette approche autorise la possibilité d'obtenir des variantes des parties décrites dans les sous-classes. Cette possibilité est mise en œuvre par le pattern Template Method comme l'illustre la figure ci-dessous.



INTRODUCTION

DISTRIBUTION PAR HÉRITAGE OU PAR DÉLÉGATION

Une seconde possibilité de répartition est mise en œuvre par la distribution des traitements dans des objets dont les classes sont indépendantes. Dans cette approche, un ensemble d'objets coopérant entre eux concourent à la réalisation d'un traitement ou d'un algorithme. Le pattern Strategy illustre ce mécanisme à la figure ci-dessous. La méthode demande de la classe Entité invoque pour la réalisation de son traitement la méthode calcule spécifiée par l'interface Stratégie. Il convient de noter que cette dernière peut avoir plusieurs implantations..



INTRODUCTION

DISTRIBUTION PAR HÉRITAGE OU PAR DÉLÉGATION

Le tableau suivant indique pour chaque pattern de comportement le type de répartition utilisé.

Pattern	Répartition
Chain of Responsibility	Délégation
Command	Délégation
Interpreter	Héritage
Iterator	Délégation
Mediator	Délégation
Memento	Délégation
Observer	Délégation
State	Délégation
Strategy	Délégation
Template Method	Héritage
Visitor	Délégation

PATTERN DE COMPORTEMENT CHAIN OF RESPONSIBILITY



CHAIN OF RESPONSIBILITY

PRESENTATION

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.

CHAIN OF RESPONSIBILITY

EXEMPLE

Nous nous plaçons dans le cadre de la vente de véhicules d'occasion. Lorsque le catalogue de ces véhicules est affiché, l'utilisateur peut demander une description de l'un des véhicules mis en vente. Si une telle description n'a pas été fournie, le système doit alors renvoyer la description associée au modèle de ce véhicule. Si à nouveau, cette description n'a pas été fournie, il convient de renvoyer la description associée à la marque du véhicule. Une description par défaut est renvoyée s'il n'y a pas non plus de description associée à la marque.

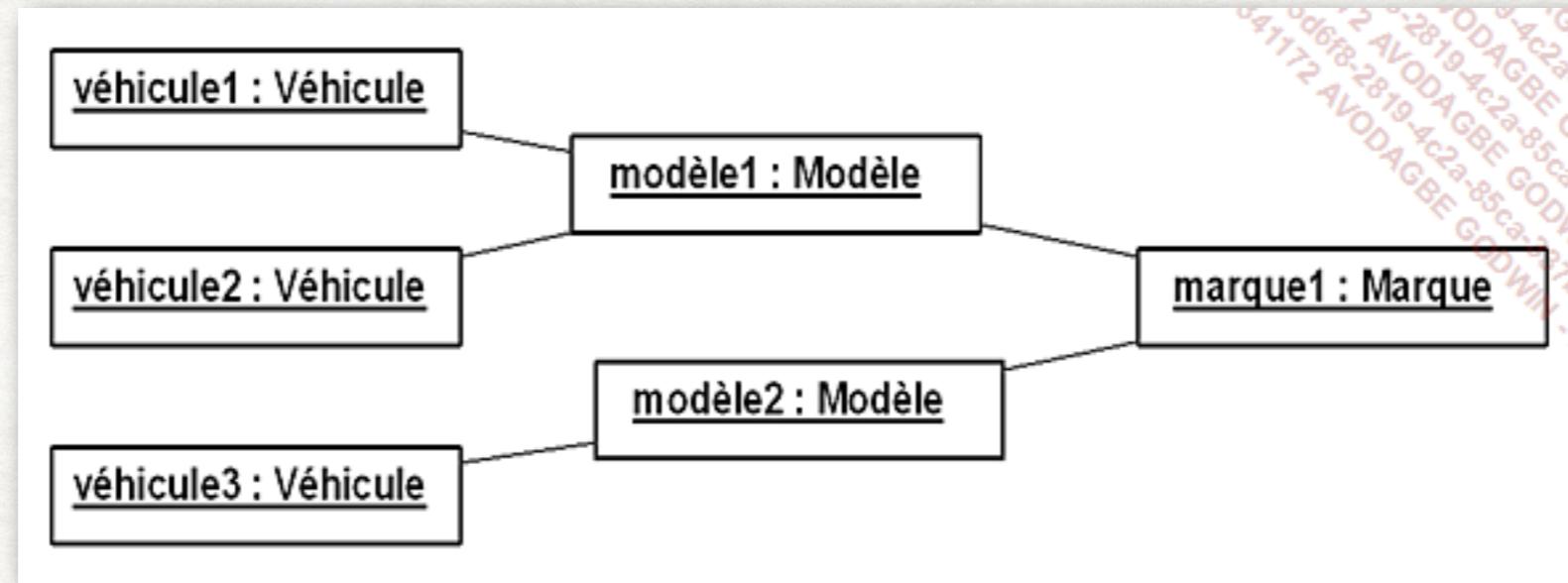
Ainsi, l'utilisateur reçoit la description la plus précise qui est disponible dans le système.

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

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

CHAIN OF RESPONSIBILITY

EXEMPLE

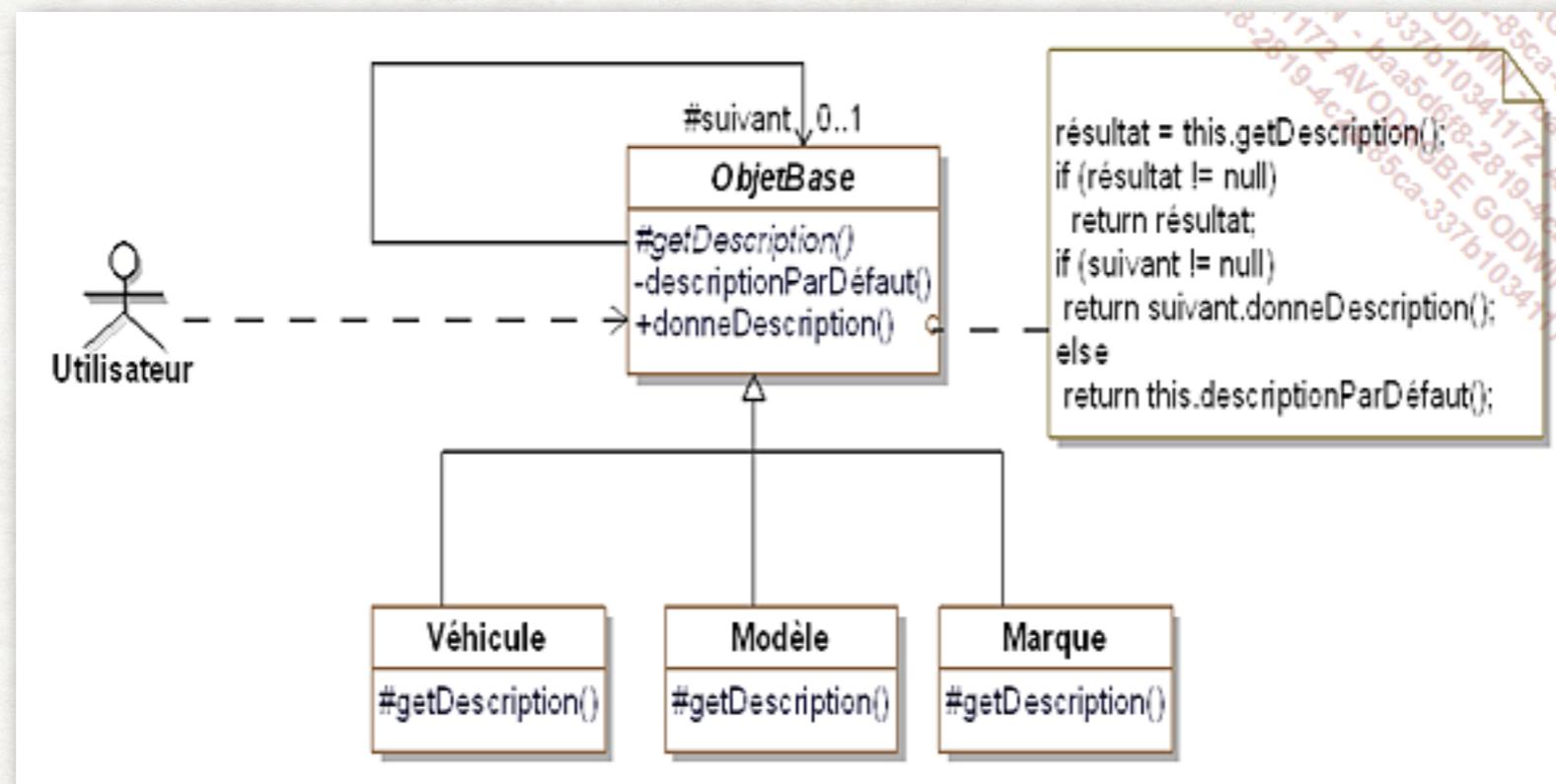


La figure ci-dessous représente le diagramme des classes du pattern Chain of Responsibility appliqué à l'exemple. Les véhicules, modèles et marques sont décrits par des sous-classes concrètes de la classe `ObjetBase`. Cette classe abstraite introduit l'association suivant qui implante la chaîne de responsabilité.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

CHAIN OF RESPONSIBILITY

EXEMPLE

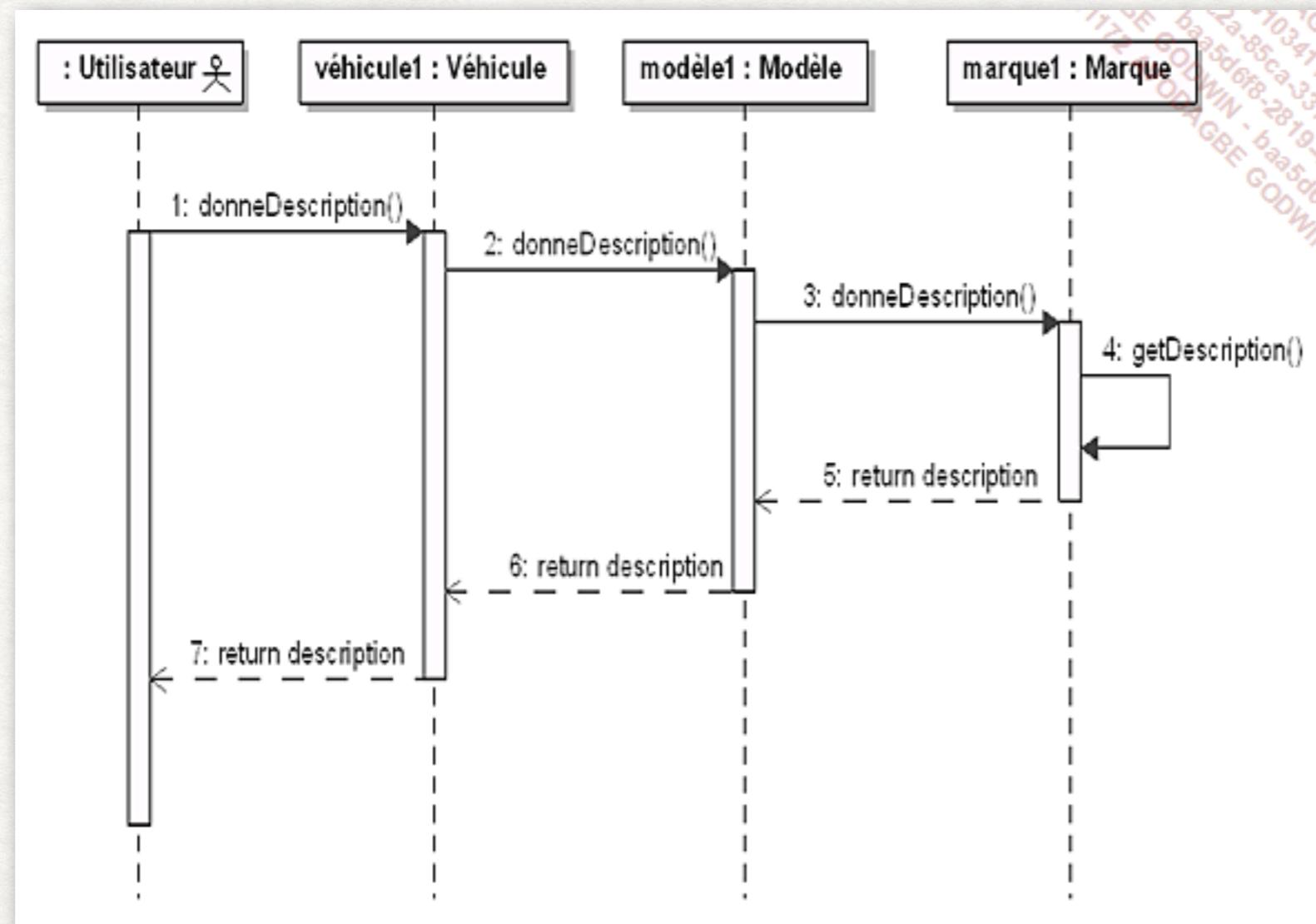


La figure ci-dessous montre un diagramme de séquence qui est un exemple de requête d'une description basée sur le diagramme d'objets de la première figure de cet exemple.

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.

CHAIN OF RESPONSIBILITY

EXEMPLE

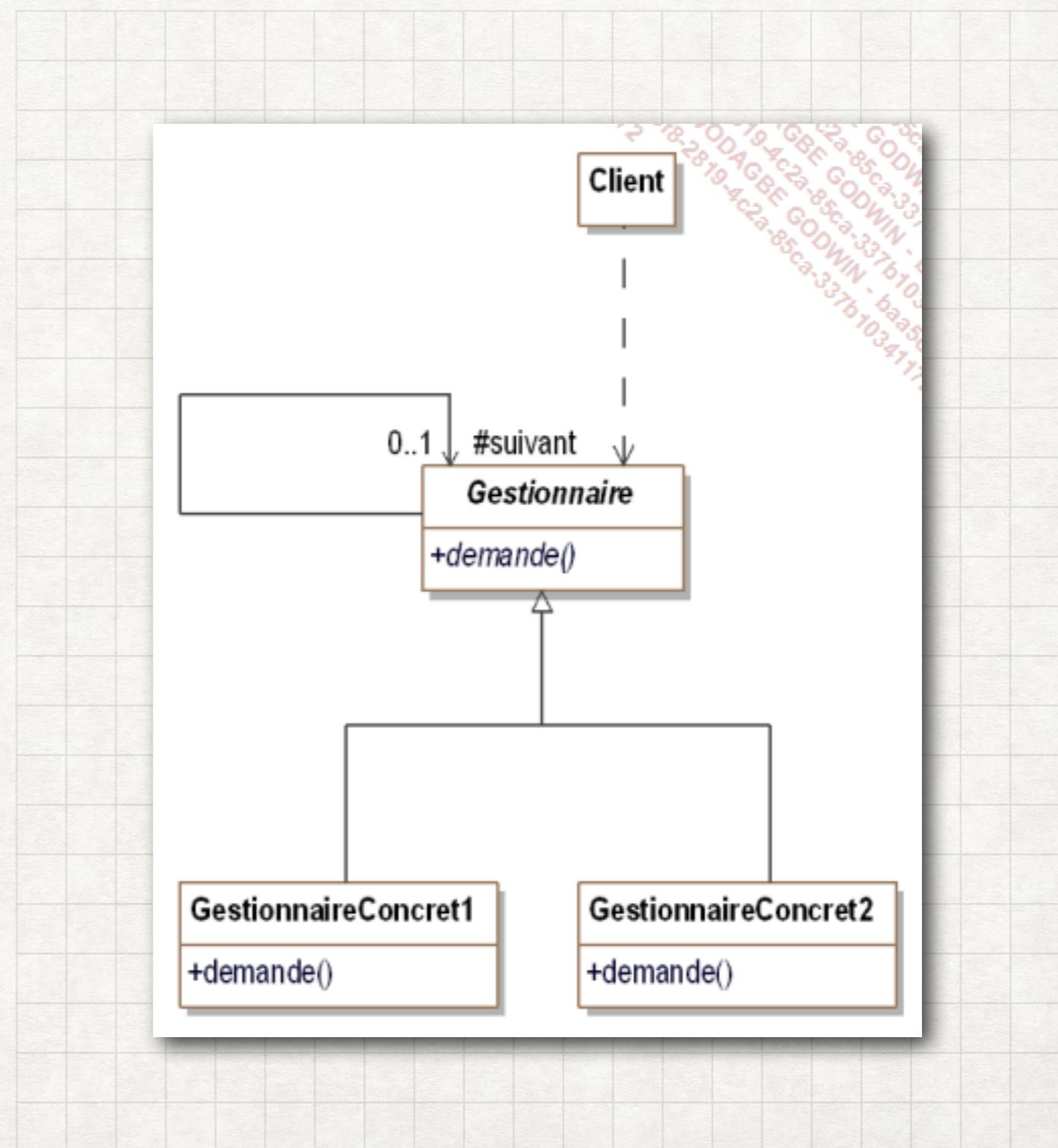


CHAIN OF RESPONSIBILITY

STRUCTURE

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.



CHAIN OF RESPONSIBILITY

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.

CHAIN OF RESPONSIBILITY

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.

DEMO

PATTERN DE COMPORTEMENT COMMAND



COMMAND PRESENTATION

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.

COMMAND

EXAMPLE

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.

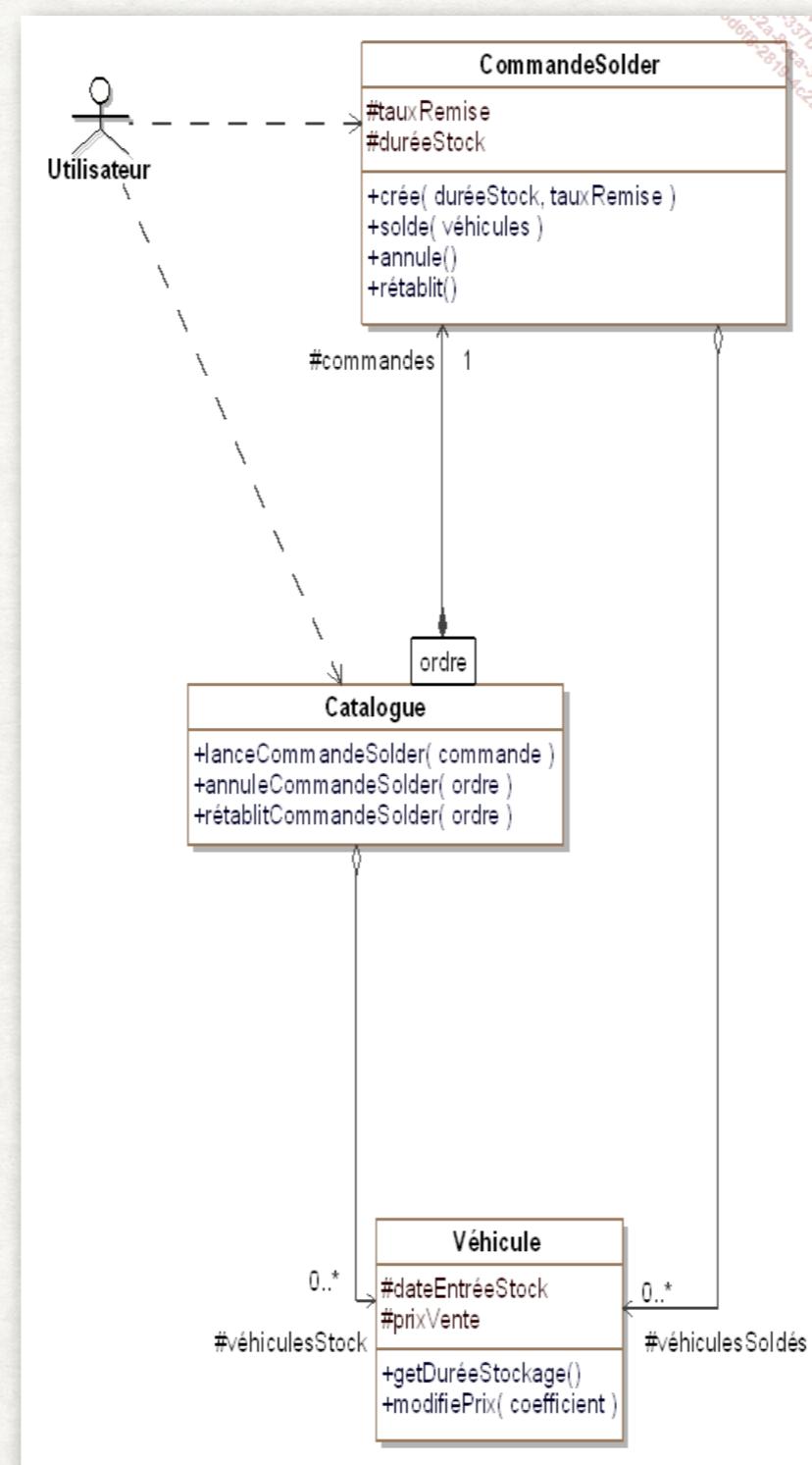
La figure ci-dessous illustre cette application du pattern Command à notre exemple. La classe CommandeSolder stocke ses deux paramètres (tauxRemise et duréeStock) ainsi que la liste des véhicules pour lesquels la remise a été appliquée (association véhiculesSoldés).

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 lance CommandeSolder, la commande passée en paramètre est exécutée puis elle est stockée dans un ordre tel que la dernière commande stockée se retrouve en première position.

COMMAND

EXEMPLE



COMMAND

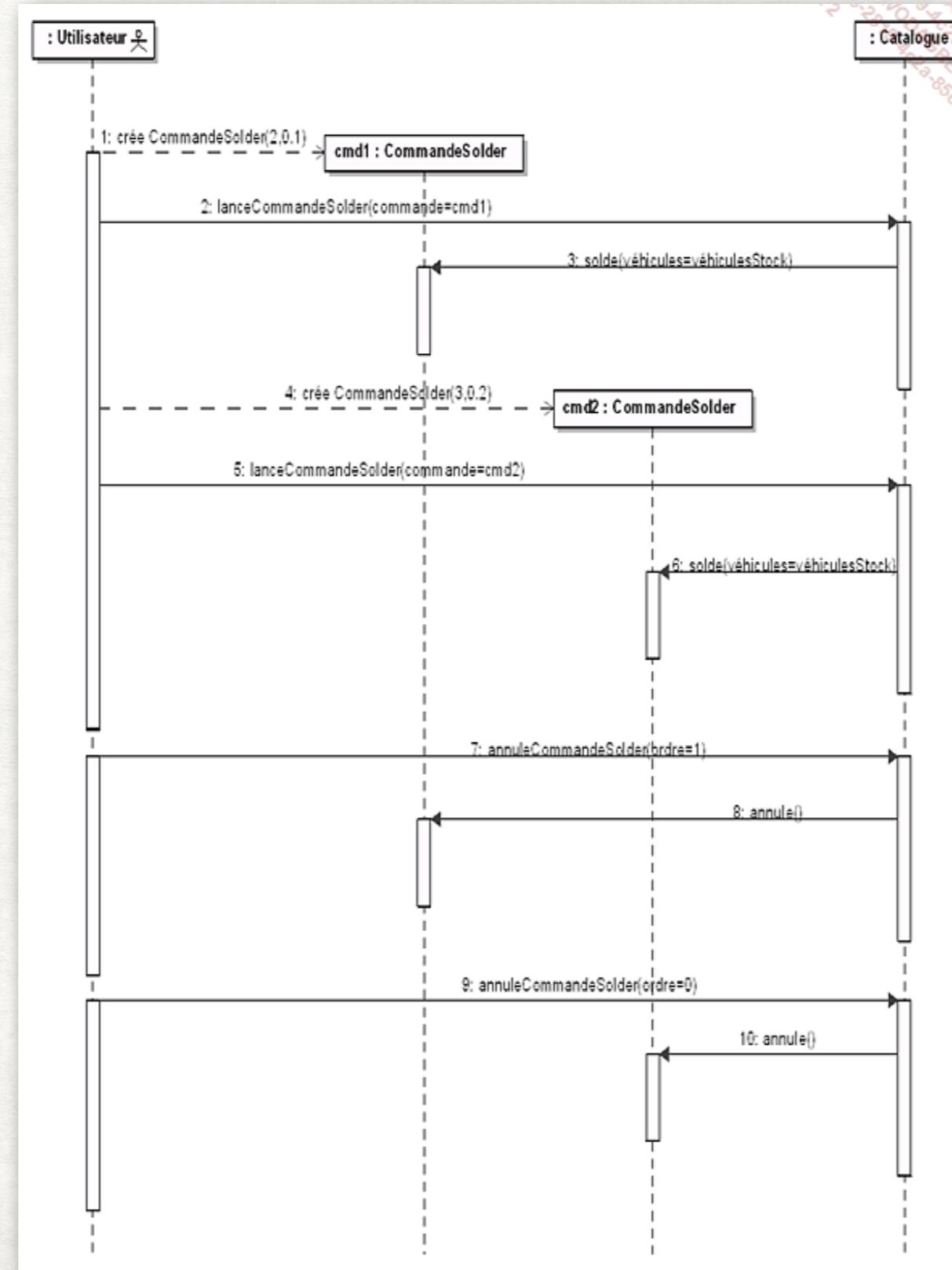
EXAMPLE

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 C# présenté plus loin.

COMMAND

EXAMPLE

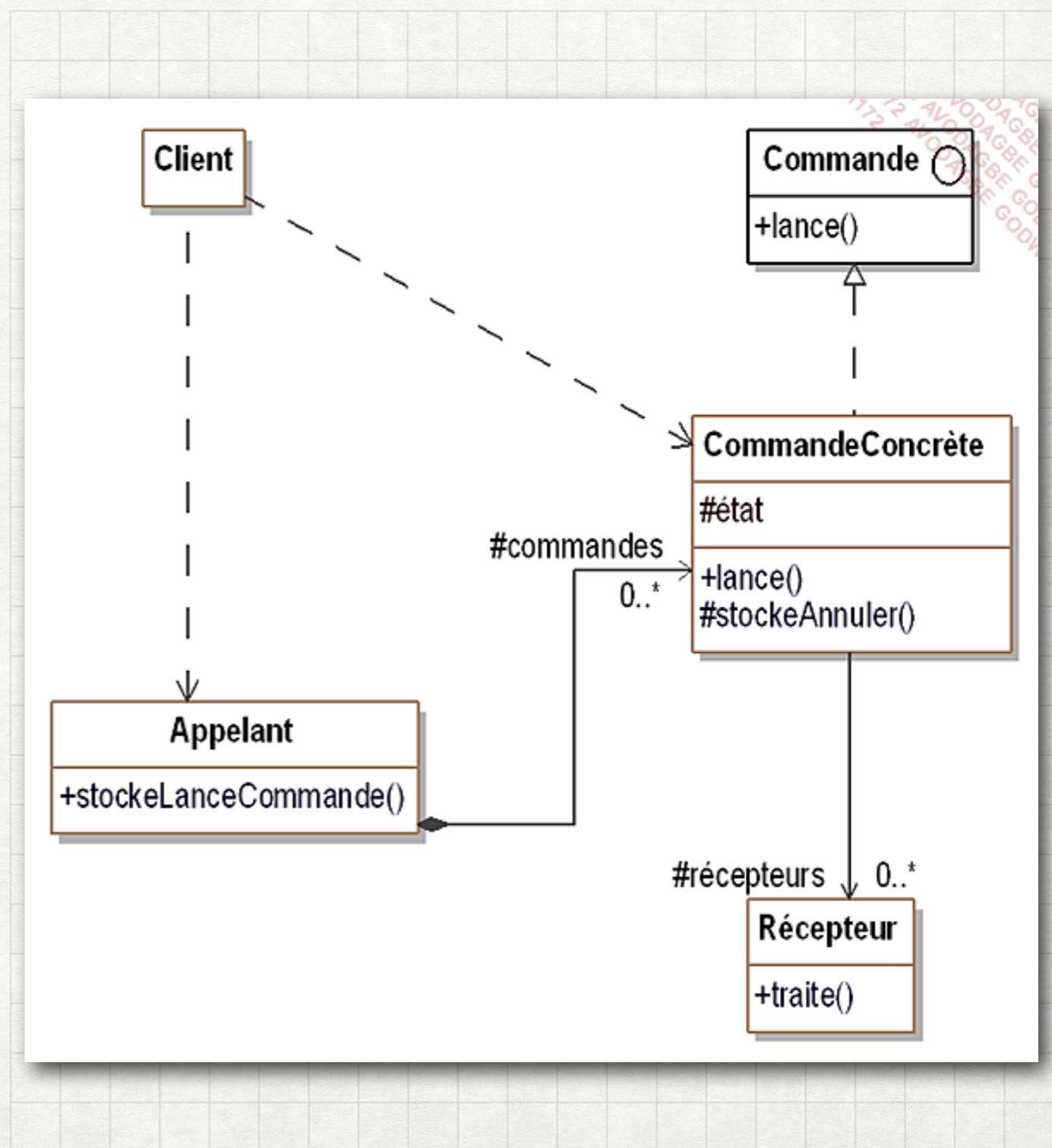


COMMAND

STRUCTURE

Les participants au pattern sont les suivants :

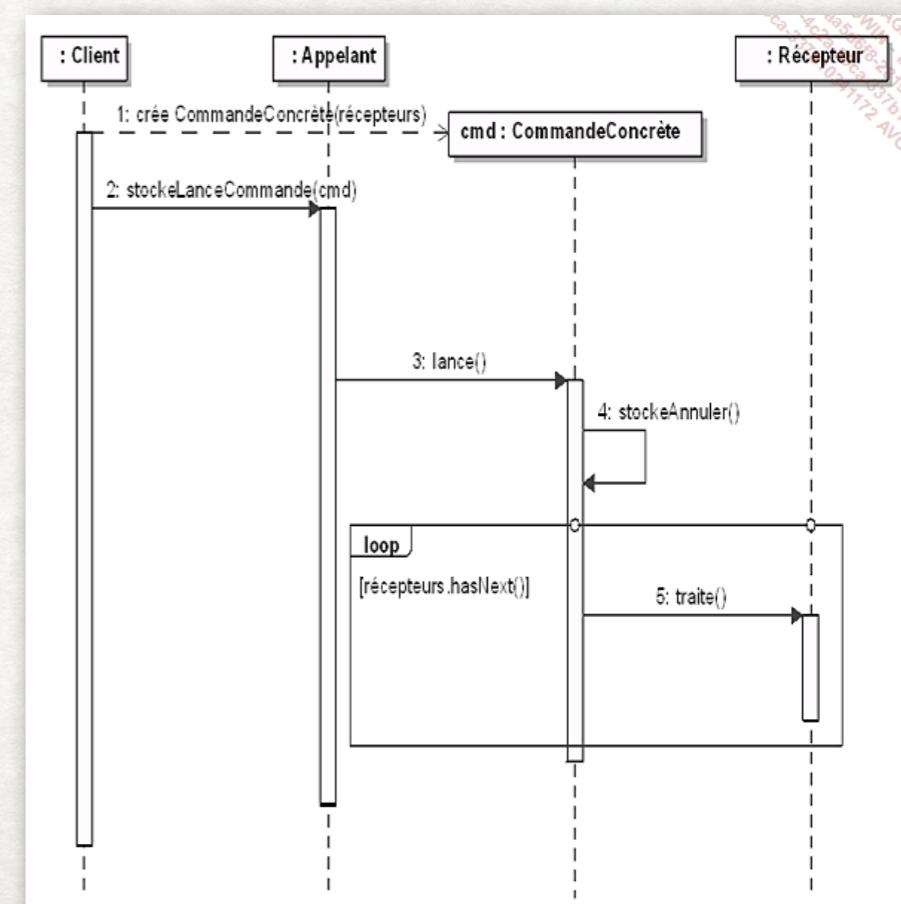
- Commande est l'interface qui introduit la signature de la méthode lance qui exécute la commande ;
- CommandeConcrète (CommandeSolder) implante la méthode lance, gère l'association avec le ou les récepteurs et implante la méthode stockeAnnuler qui stocke l'état (ou les valeurs nécessaires) pour pouvoir 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.



COMMAND COLLABORATIONS

La figure ci-dessous 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.



COMMAND

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.

DEMO

PATTERN DE COMPORTEMENT INTERPRETER



INTERPRETER

PRESENTATION

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.

INTERPRETER

EXEMPLE

Nous voulons créer un petit moteur de recherche des véhicules basé sur la recherche par mot-clé dans la description des véhicules à l'aide d'expressions booléennes selon la grammaire très simple suivante :

expression ::= terme || mot-clé || (expression)

terme ::= facteur 'ou' facteur

facteur ::= expression 'et' expression

mot-clé ::= 'a'..'z', 'A'..'Z' {'a'..'z', 'A'..'Z'}*

Les symboles entre apostrophes sont des symboles terminaux. Les symboles non terminaux sont expression, terme, facteur et mot-clé. Le symbole de départ est expression.

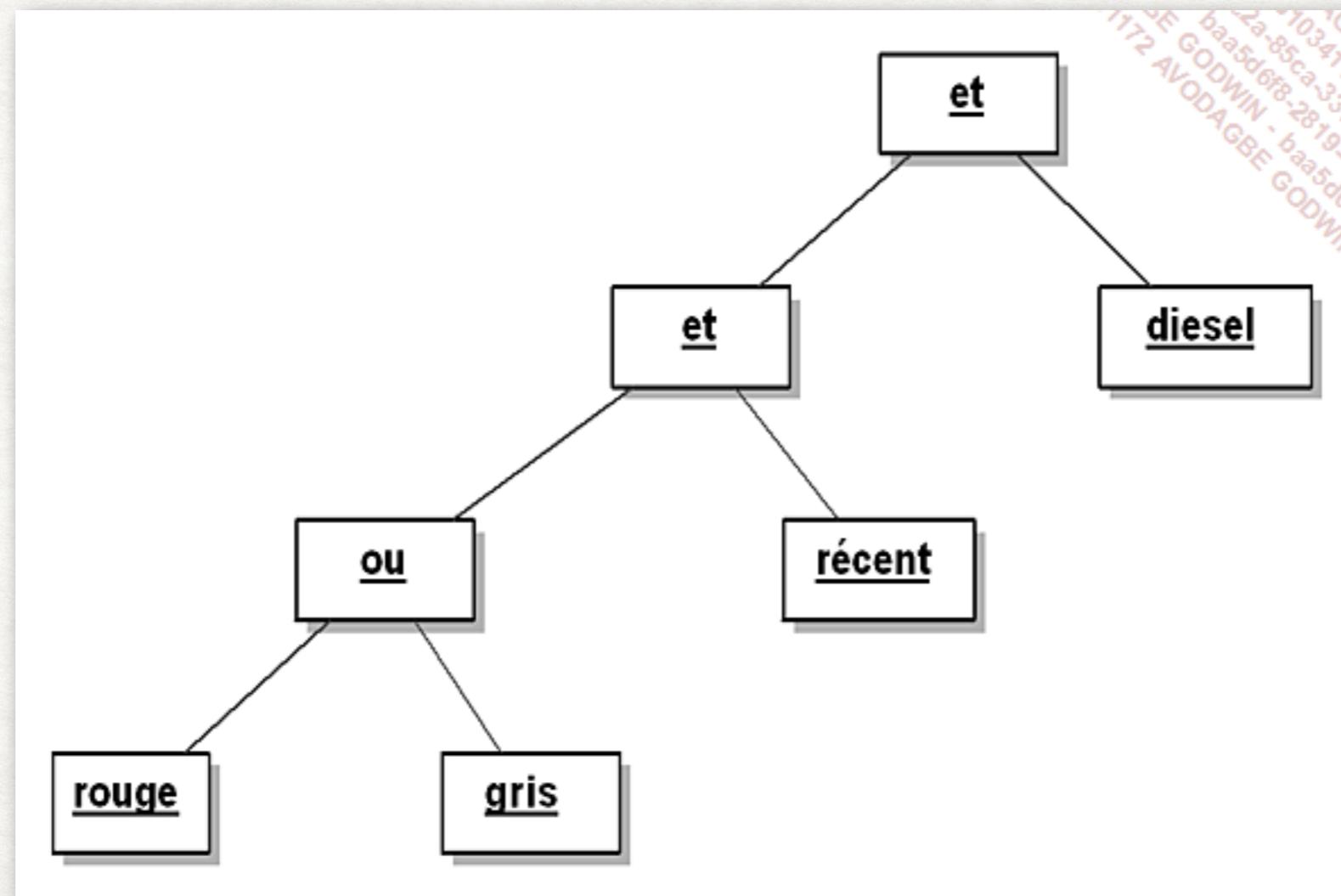
Nous mettons en œuvre le pattern Interpreter afin de pouvoir exprimer toute expression répondant à cette grammaire selon un arbre syntaxique constitué d'objets afin de pouvoir l'évaluer en l'interprétant.

Un tel arbre n'est constitué que de symboles terminaux. Pour simplifier, nous considérons qu'un mot-clé constitue un symbole terminal en tant que chaîne de caractères.

L'expression (rouge ou gris) et récent et diesel va être traduire par l'arbre syntaxique de la figure ci-dessous.

INTERPRETER

EXEMPLE



INTERPRETER

EXEMPLE

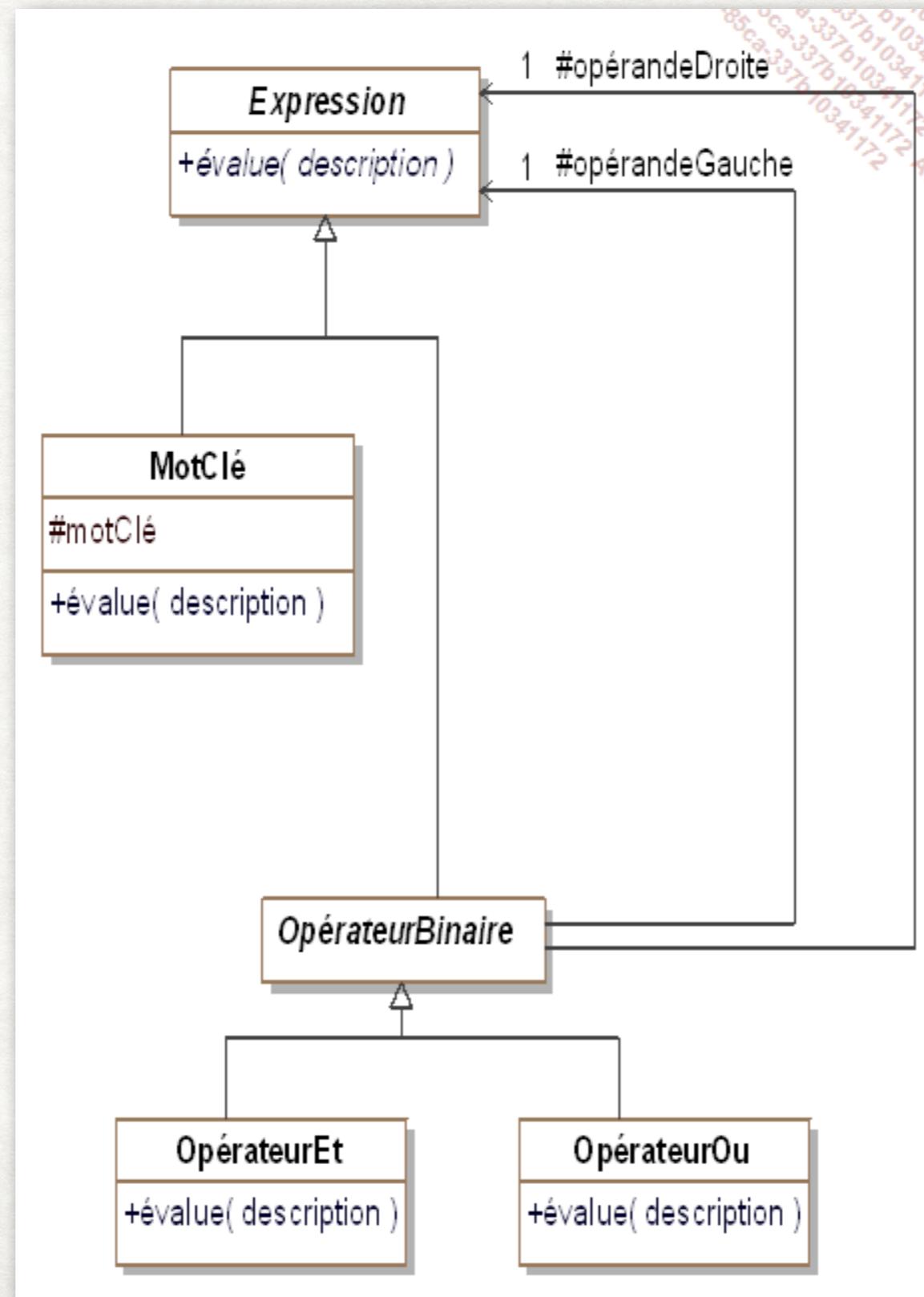
L'évaluation d'un tel arbre pour la description d'un véhicule se fait en commençant par le sommet. Quand un nœud est un opérateur, l'évaluation se fait en calculant récursivement la valeur de chaque sous-arbre (celui de gauche puis celui de droite) et en appliquant l'opérateur. Quand un nœud est un mot-clé, 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.

Le diagramme des classes permettant de décrire des arbres syntaxiques comme celui de la figure 20.1 est représenté à la figure ci-dessous. La méthode `évalue` permet d'évaluer l'expression pour une description d'un véhicule fournie en paramètre.

INTERPRETER

EXEMPLE



INTERPRETER

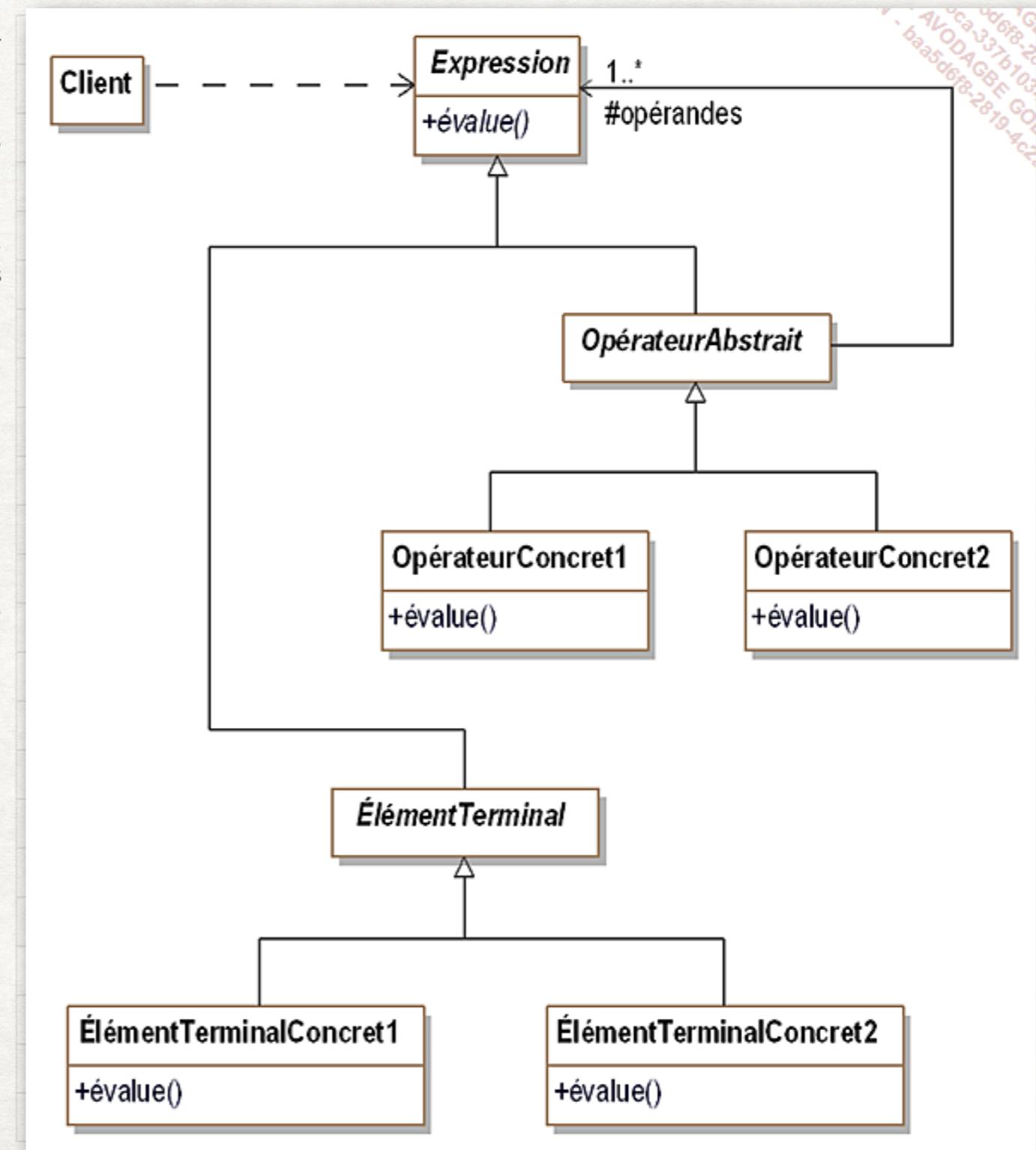
STRUCTURE

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, unitaires (opérateur « - ») ou prenant plus d'argument comme des fonctions.

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 (Mot Clé) sont des classes concrètes correspondant à un élément terminal, capables d'évaluer cet élément.



INTERPRETER

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.

INTERPRETER

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.

DEMO

PATTERN DE COMPORTEMENT ITERATOR



ITERATOR

PRESENTATION

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.

ITERATOR

EXEMPLE

Nous voulons donner un accès séquentiel aux véhicules composant le catalogue. Pour cela, nous pouvons implanter dans la classe du catalogue les méthodes suivantes :

- début : initialise le parcours du catalogue ;
- item : renvoie le véhicule courant ;
- suivant : passe au véhicule suivant.

Cette technique présente deux inconvénients :

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

Le pattern **Iterator** propose une solution à ce problème. L'idée est de créer une classe **Itérateur** dont chaque instance peut gérer un parcours dans une collection. Les instances de cette classe **Itérateur** sont créées par la classe de collection qui se charge de les initialiser.

Le but du pattern **Iterator** est de fournir une solution qui puisse être paramétrée par le type des éléments des collections. Nous introduisons donc deux classes abstraites génériques :

- Itérateur** est une classe abstraite générique qui introduit les méthodes **début**, **item** et **suivant** ;
- Catalogue** est également une classe abstraite générique qui introduit la méthode qui crée, initialise et retourne une instance de **Itérateur**.

ITERATOR

EXEMPLE

Il est ensuite possible de créer les sous-classes concrètes de ces deux classes abstraites génériques, sous-classes qui lient notamment les paramètres de générnicité aux types utilisés dans l'application.

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

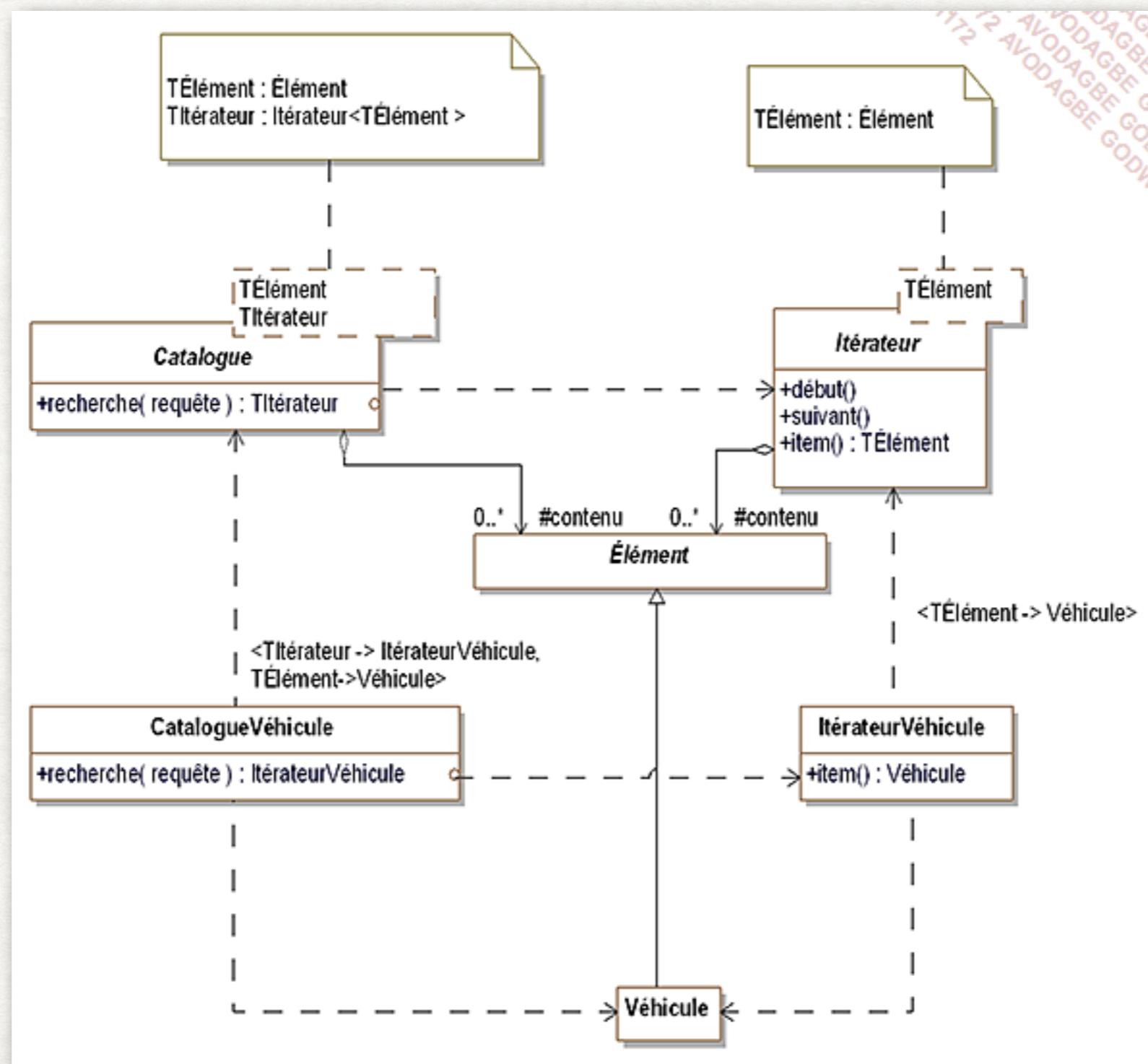
Ce diagramme de classes utilise des paramètres génériques qui sont contraints (`Télément` est un sous-type de `Élément` et `ITérateur` est un sous-type de `Itérateur<Télément>`). Les deux classes `Catalogue` et `Itérateur` possèdent une association avec un ensemble d'éléments, l'ensemble des éléments référencés par `Itérateur` étant un sous-ensemble de ceux référencés par `Catalogue`.

Les sous-classes `CatalogueVéhicule` et `ItérateurVéhicule` héritent par une relation qui fixe les types des paramètres de générnicité de leurs surclasses respectives.



ITERATOR

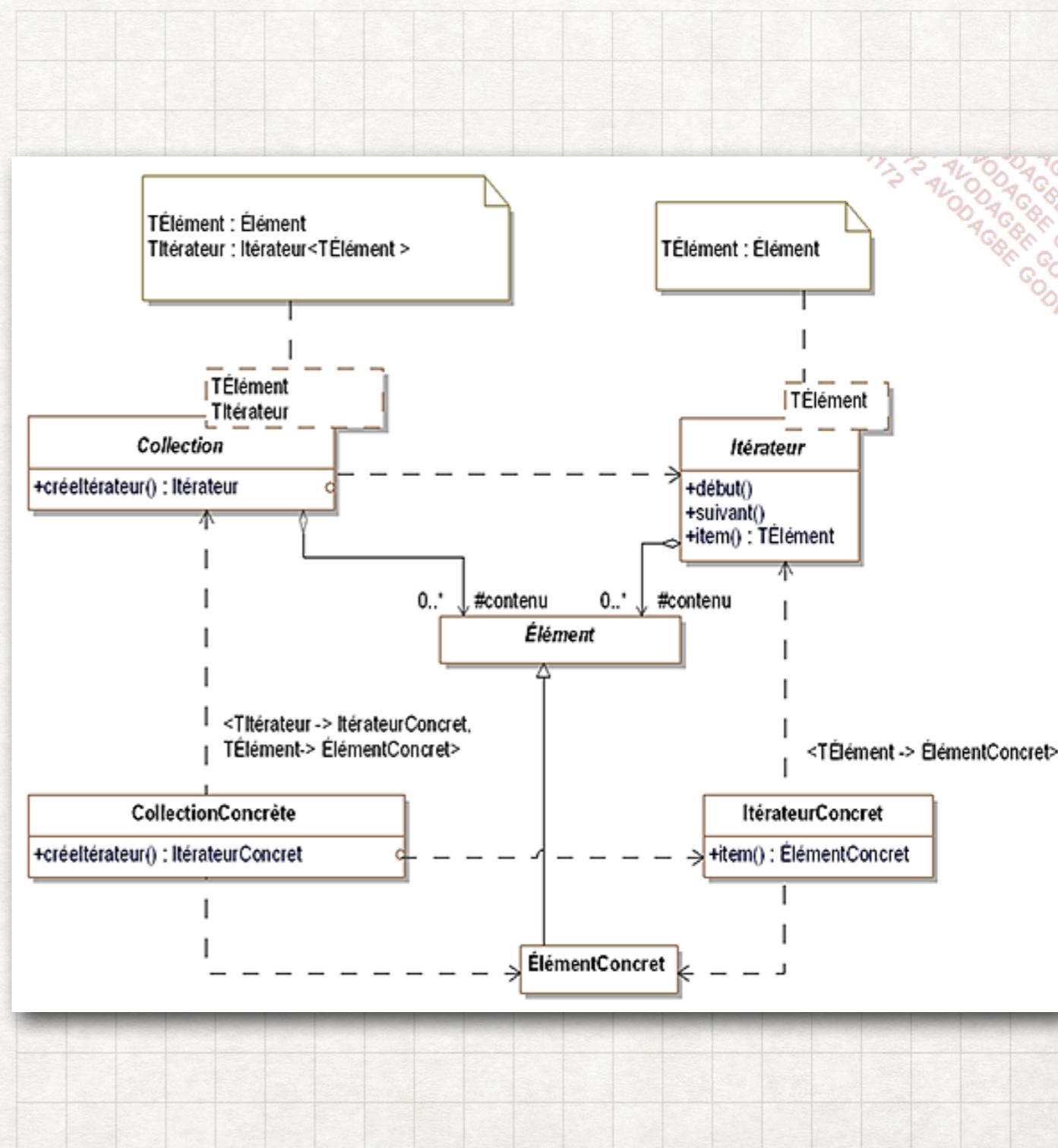
EXAMPLE



ITERATOR STRUCTURE

Les participants au pattern sont les suivants :

- Itérateur** est la classe abstraite qui implante l'association de l'itérateur avec les éléments de la collection ainsi que les méthodes. Elle est générique et paramétrée par le type **TÉlément** ;
- ItérateurConcret** (**ItérateurVéhicule**) est une sous-classe concrète de **Itérateur** qui lie **TÉlément** à **ÉlémentConcret** ;
- Collection** (**Catalogue**) est la classe abstraite qui implante l'association de la collection avec les éléments et la méthode `créeItérateur` ;
- CollectionConcrète** (**CatalogueVéhicule**) est une sous-classe concrète de **Collection** qui lie **TÉlément** à **ÉlémentConcret** et **Itérateur** à **ItérateurConcret** ;
- Élément** est la classe abstraite des éléments de la collection ;
- ÉlémentConcret** (**Véhicule**) est une sous-classe concrète de **Élément** utilisée par **ItérateurConcret** et **CollectionConcrète**.



ITERATOR COLLABORATIONS

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

ITERATOR

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.

DEMO

PATTERN DE COMPORTEMENT MEDIATOR



MEDIATOR

PRESENTATION

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.

MEDIATOR

EXEMPLE

La conception par objets favorise la distribution du comportement entre les objets du système. Cependant, à l'extrême, cette distribution peut conduire à un très grand nombre de liaisons obligeant quasiment chaque objet à connaître tous les autres objets du système. Une conception avec une telle quantité de liaisons peut s'avérer être de mauvaise qualité. En effet, la modularité et les possibilités de réutilisation des objets sont alors réduites. Chaque objet ne peut pas travailler sans les autres et le système devient monolithique, perdant toute modularité. De surcroît pour adapter et modifier le comportement d'une petite partie du système, il devient nécessaire de définir de nombreuses sous-classes.

Les interfaces utilisateur dynamiques sont un bon exemple d'un tel système. Une modification de la valeur d'un contrôle graphique peut conduire à modifier l'aspect d'autres contrôles graphiques comme, par exemple :

- devenir visible ou masqué ;
- modifier le nombre de valeurs possibles (pour un menu) ;
- changer le format des valeurs à saisir.

MEDIATOR

EXEMPLE

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

L'autre possibilité est de mettre en œuvre le pattern Mediator. Celle-ci consiste à construire un objet central chargé de la coordination des contrôles graphiques. Lorsque la valeur d'un contrôle est modifiée, il prévient l'objet médiateur qui se charge d'invoquer les méthodes adéquates des autres contrôles graphiques afin qu'ils puissent réaliser les modifications nécessaires.

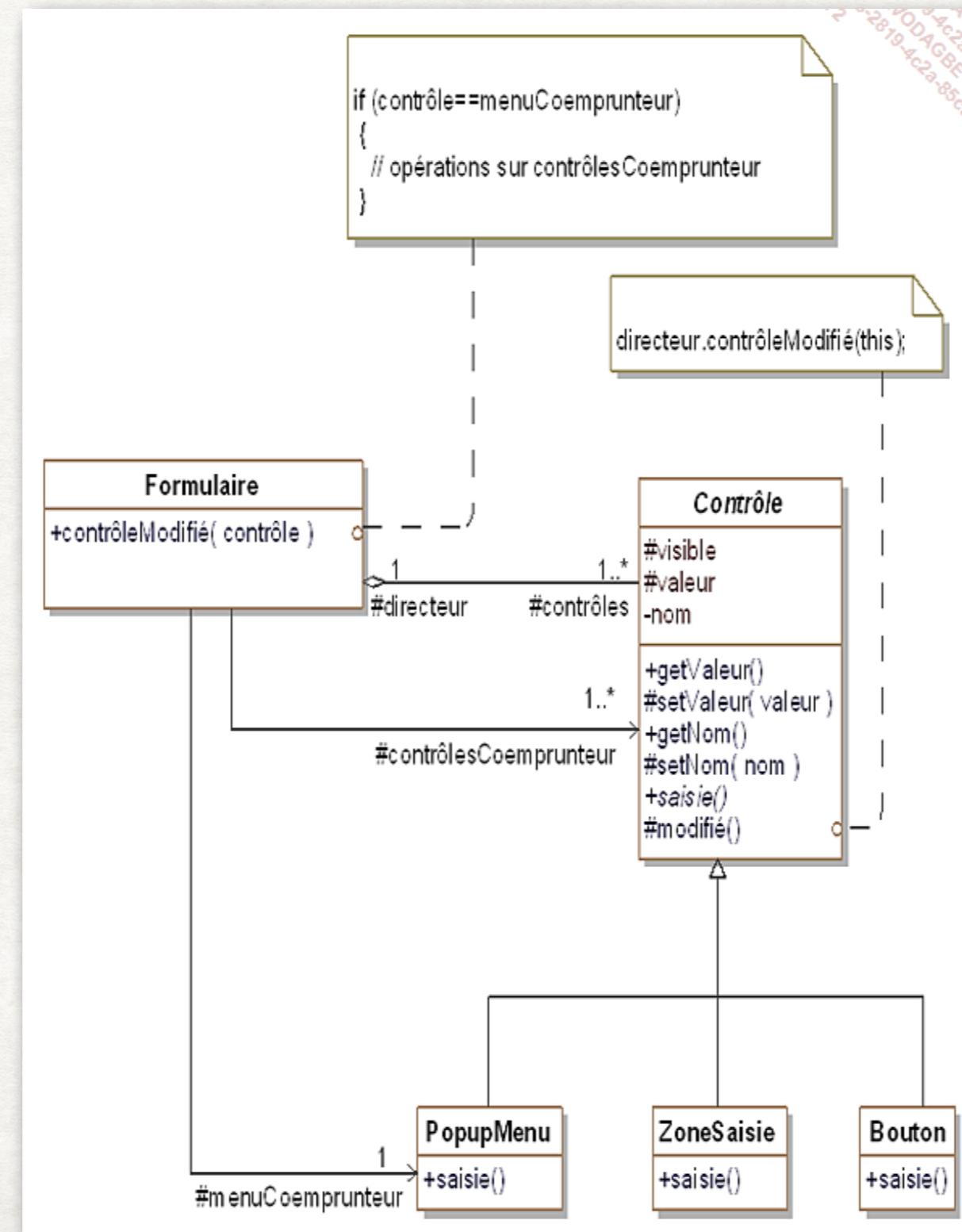
Dans notre système de vente en ligne de véhicules, un emprunt peut être demandé pour acquérir un véhicule en remplissant un formulaire en ligne. Il est possible d'emprunter seul ou avec un co-emprunteur. Ce choix se fait à l'aide d'un menu. Si le choix est d'emprunter avec un co-emprunteur, tout un ensemble de contrôles graphiques relatifs aux données du co-emprunteur doivent apparaître et être 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**.

MEDIATOR

EXEMPLE

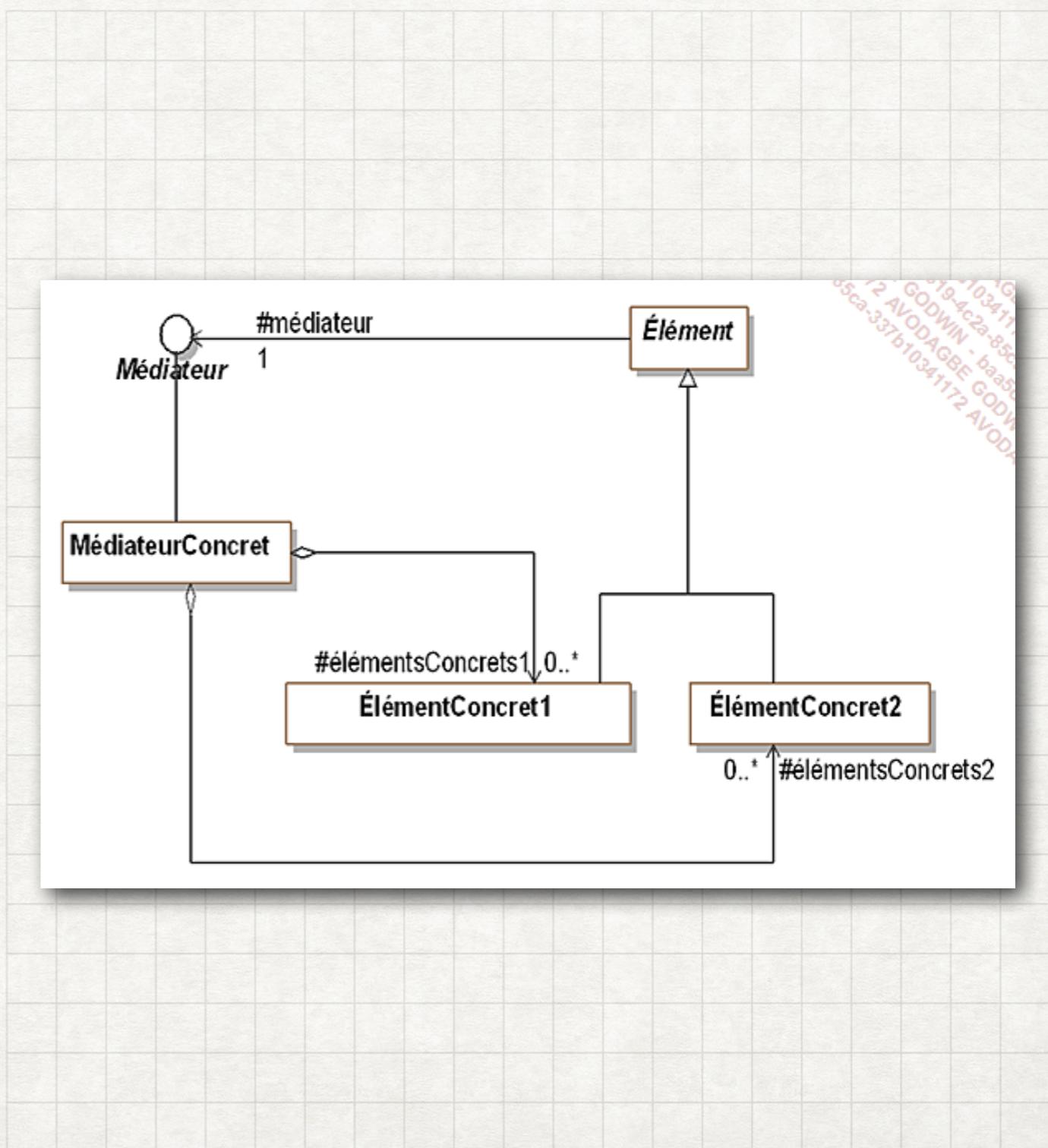


MEDIATOR

STRUCTURE

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 (PopMenu, ZoneSaisie et Bouton) sont les classes concrètes des éléments qui communiquent avec le médiateur au lieu de communiquer avec les autres éléments.



MEDIATOR COLLABORATIONS

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

MEDIATOR

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.

DEMO

PATTERN DE COMPORTEMENT MEMENTO



MEMENTO

PRESÉNTATION

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

MEMENTO

EXEMPLE

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

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

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

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

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

MEMENTO

EXEMPLE

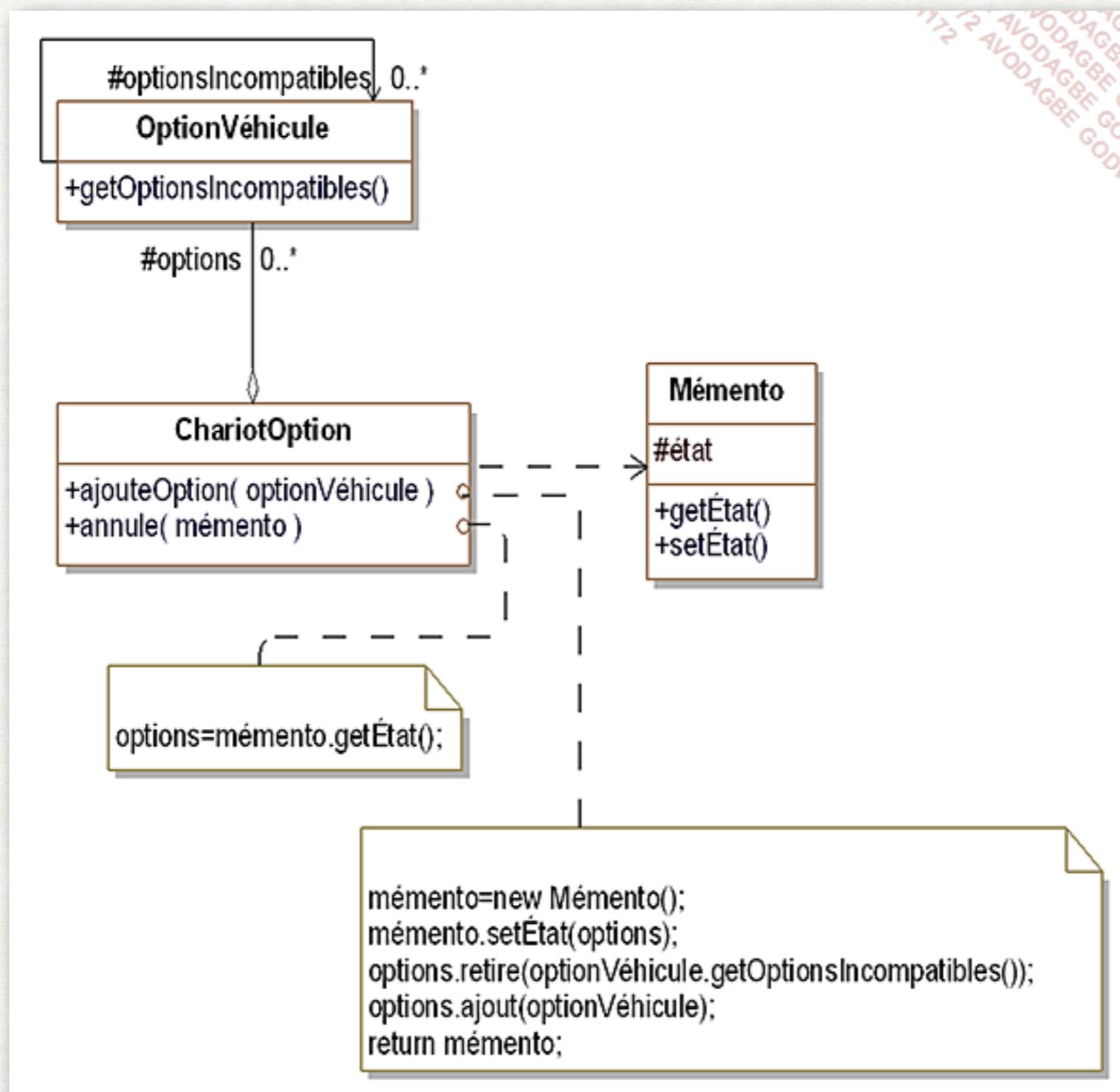
Le pattern Memento propose une réponse à ce problème. Elle consiste à mémoriser les états du chariot dans un objet appelé memento. Lors de l'ajout d'une nouvelle option, le chariot crée un memento, l'initialise avec son état, retire les options incompatibles avec cette nouvelle option, procède à l'ajout de cette nouvelle option et renvoie le memento 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 memento et y restaurer un état précédent : le memento est opaque vis-à-vis des autres objets.

Le diagramme de classes correspondant est illustré à la figure 23.1. Le chariot y est représenté par la classe ChariotOption et le memento par la classe Memento. 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.

MEMENTO

EXEMPLE

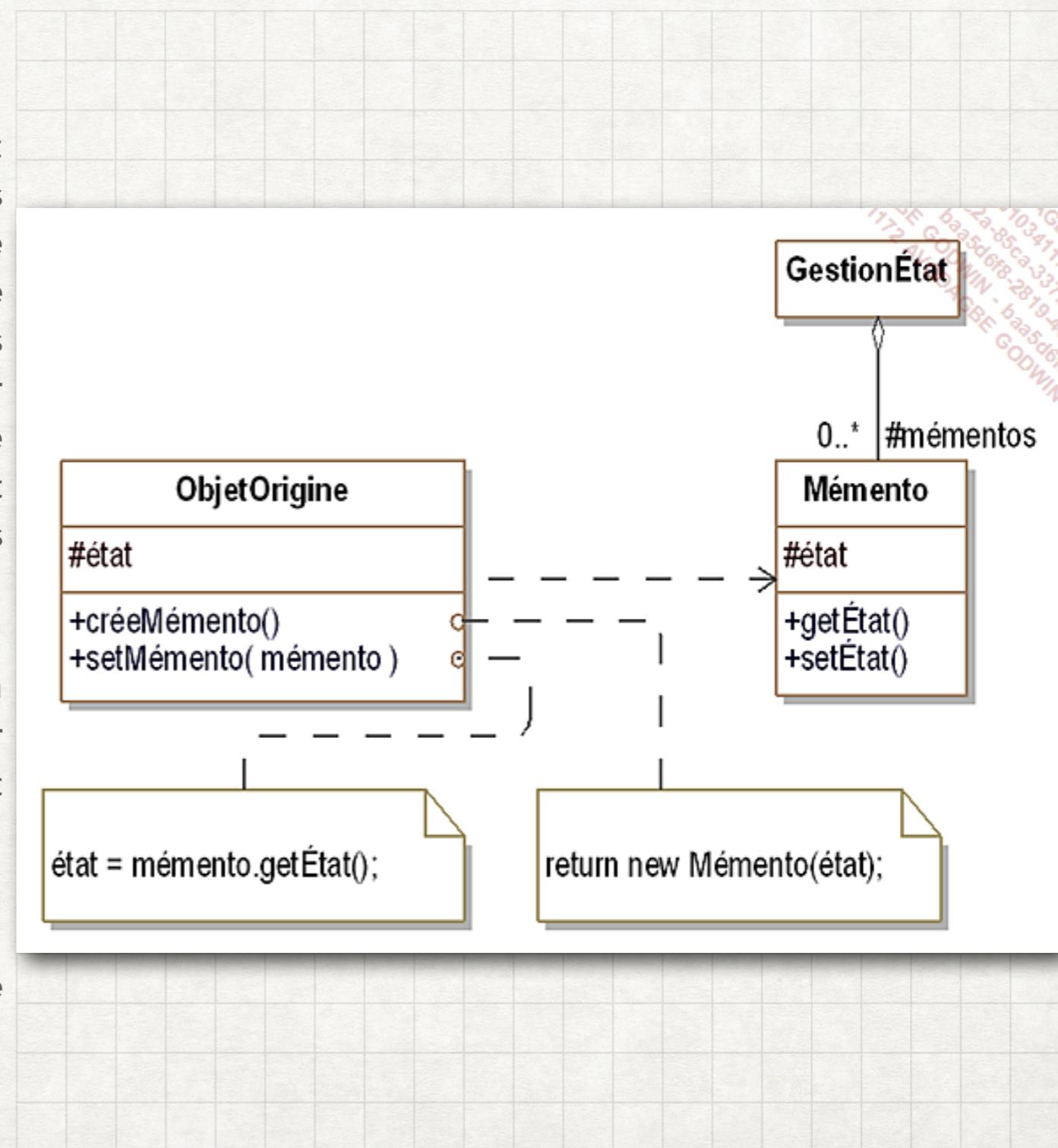


MEMENTO

STRUCTURE

Les participants au pattern sont les suivants :

- Memento 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 memento 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 memento pour mémoriser leur état interne qu'ils peuvent également restaurer à partir d'un memento ;
- GestionEtat est responsable de la gestion des mémentos et n'accède pas à l'état interne des objets d'origine.



MEMENTO

COLLABORATIONS

Une instance de `GestionEtat` demande un memento à l'objet d'origine par appel de la méthode `creerMemento`, le sauvegarde et en cas de besoin d'annulation et de retour à l'état mémorisé dans le memento, le transmet à nouveau à l'objet d'origine par la méthode `setMemento`.

MEMENTO

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.

DEMO

PATTERN DE COMPORTEMENT OBSERVER



OBSERVER

PRESNTATION

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.

OBSERVER

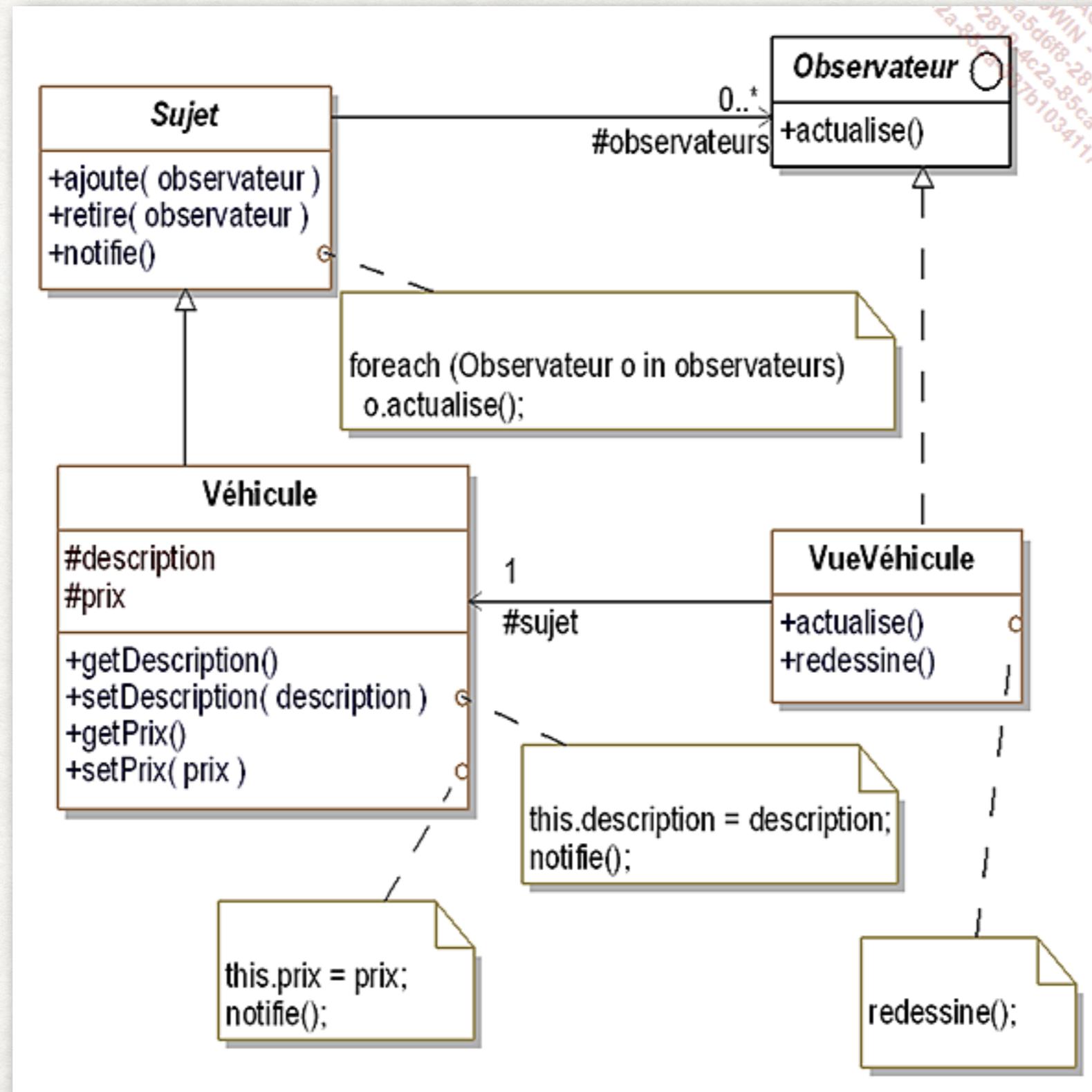
EXEMPLE

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

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

OBSERVER

EXEMPLE



OBSERVER

EXEMPLE

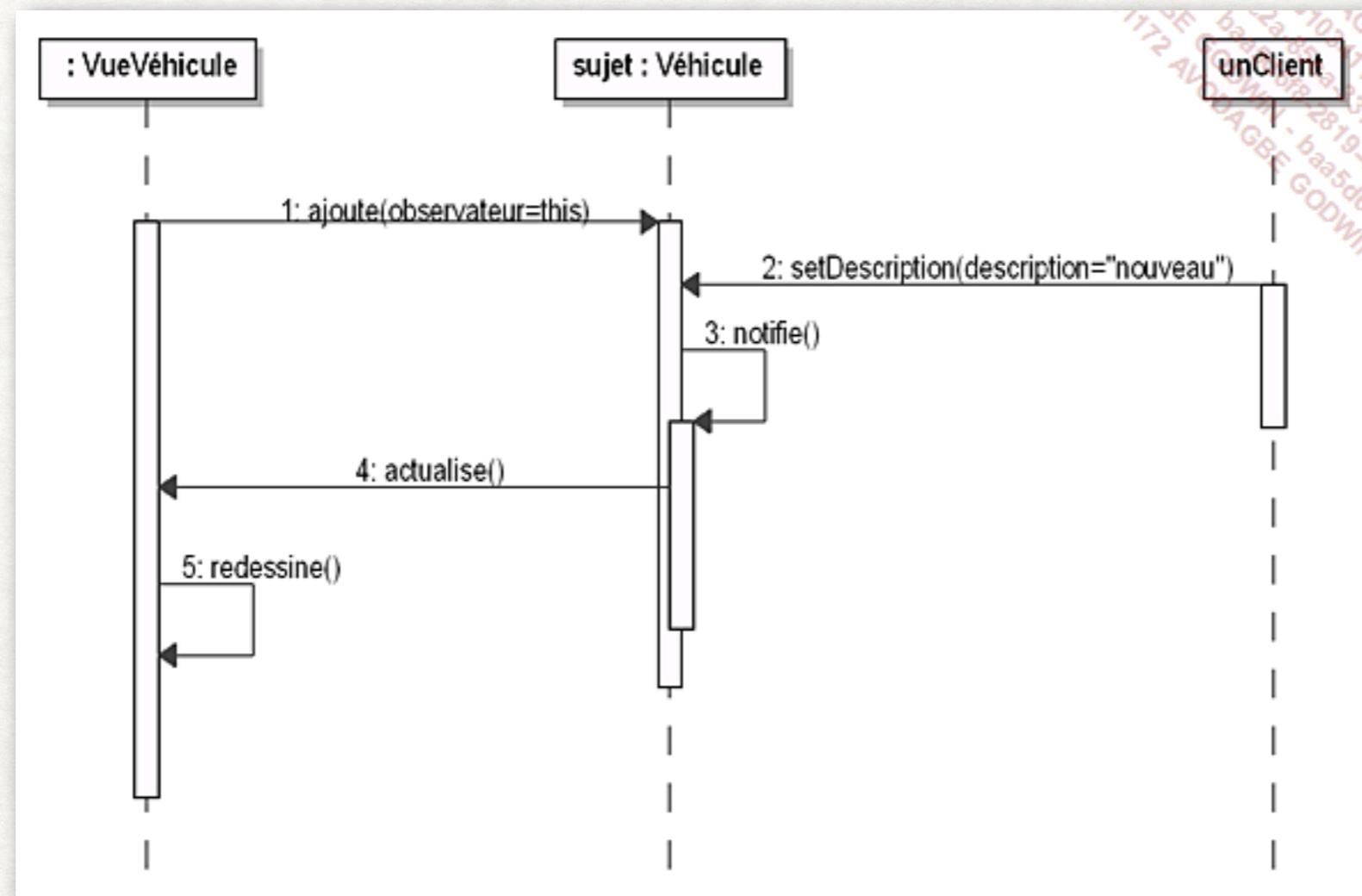
Le diagramme contient les quatre classes suivantes :

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

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

OBSERVER

EXEMPLE

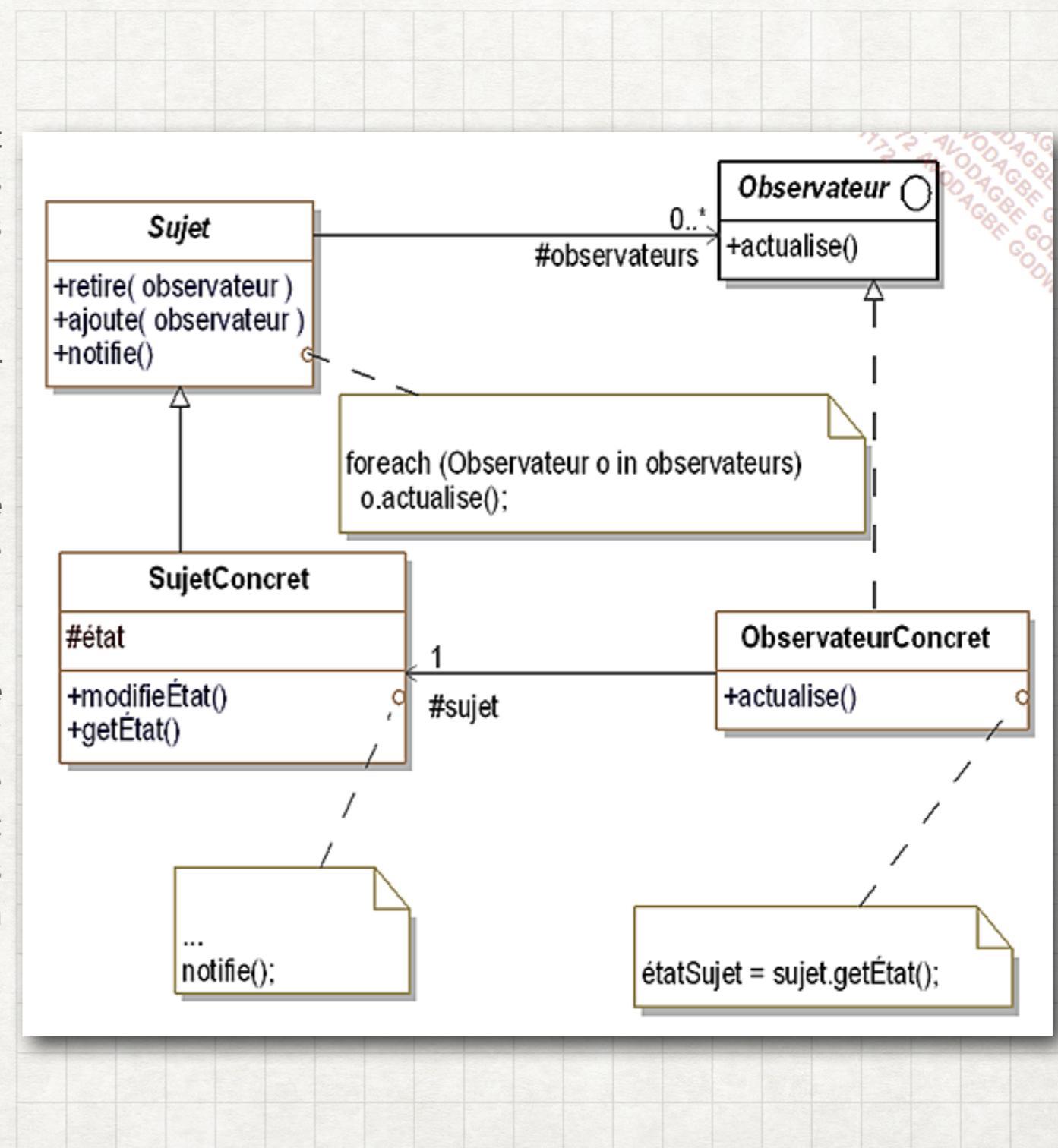


OBSERVER

STRUCTURE

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()`.



OBSERVER

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.

DEMO

PATTERN DE COMPORTEMENT STATE

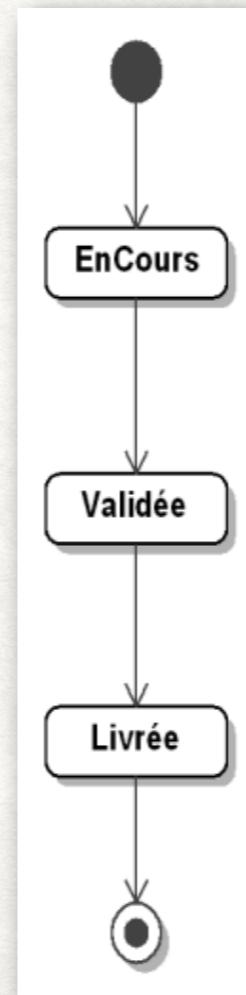


STATE PRESENTATION

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

STATE EXEMPLE

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



STATE EXEMPLE

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`.

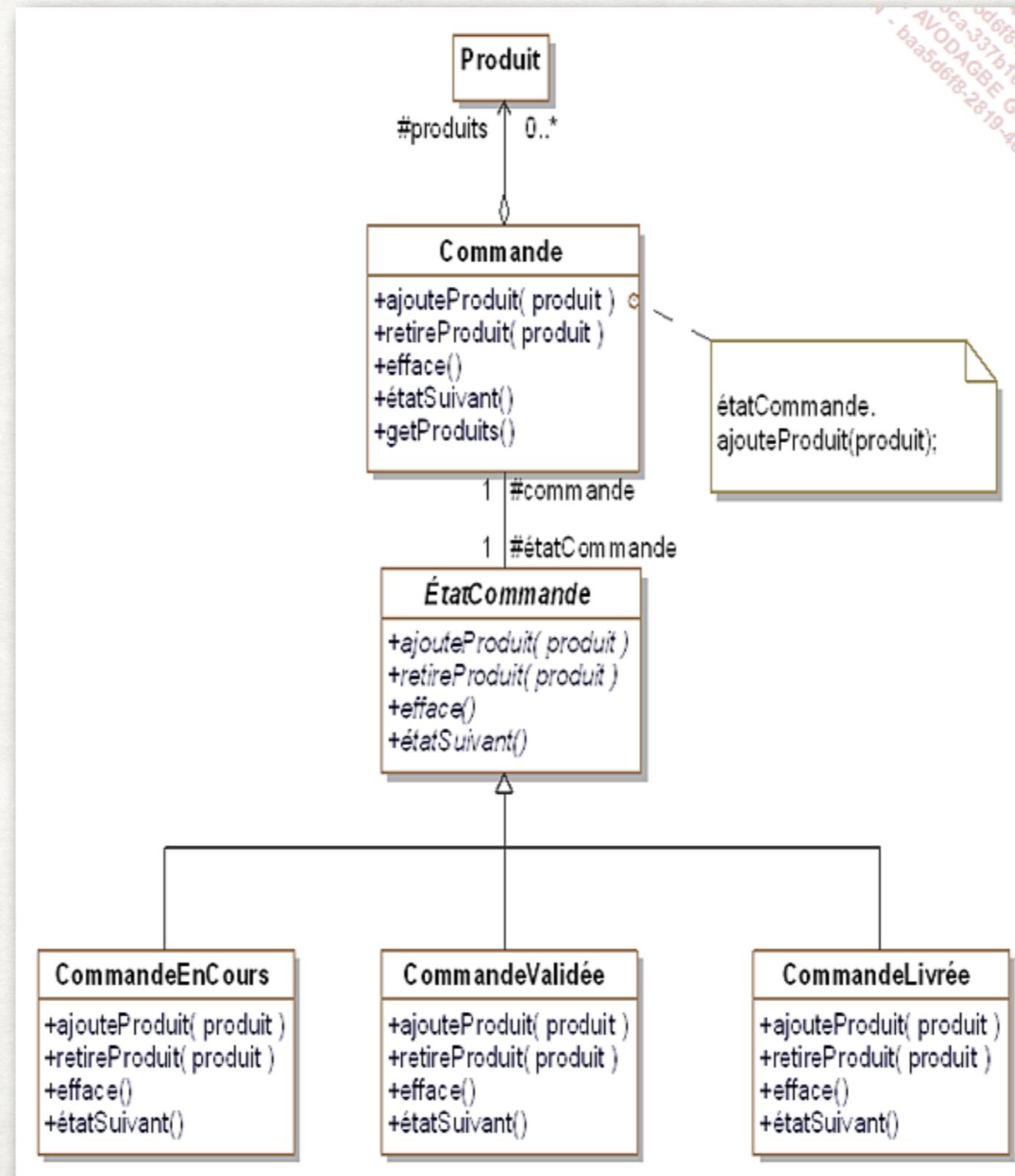
L'approche traditionnelle pour résoudre ces différences de comportement consiste à utiliser des conditions dans le corps des méthodes. Cette approche conduit souvent à des méthodes complexes à écrire et à appréhender.

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 délèguent leur invocation à cette instance. La note relative à la méthode `ajouteProduit` dans la figure ci-dessous illustre cette délégation.

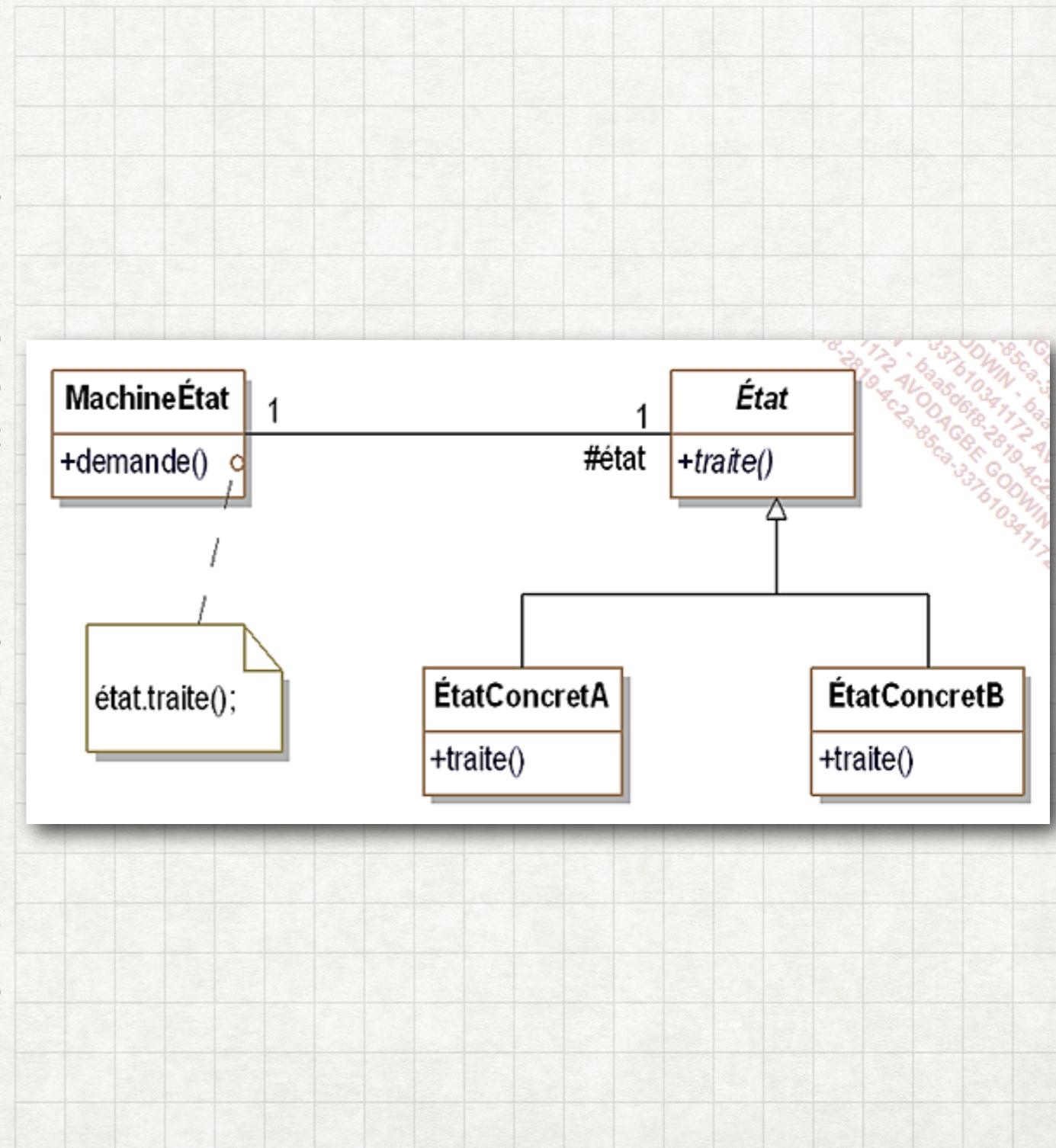
STATE EXAMPLE



STATE STRUCTURE

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.



STATE 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.

STATE 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.

DEMO

PATTERN DE COMPORTEMENT STRATEGY



STRATEGY

PRESENTATION

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.

STRATEGY

EXEMPLE

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

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

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

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

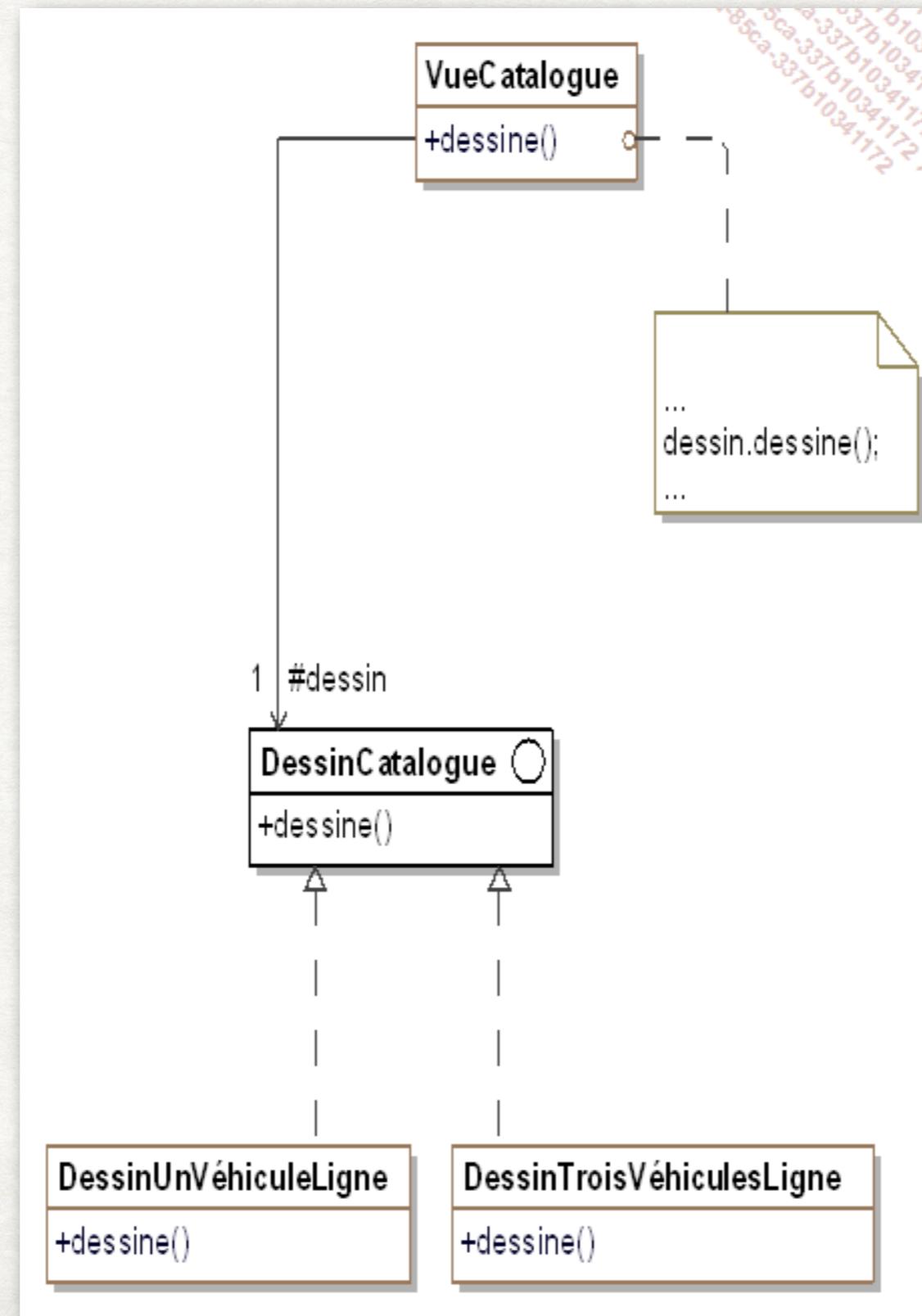
Une autre possibilité est d'implanter les deux algorithmes dans la classe `VueCatalogue` et à l'aide d'instructions conditionnelles d'effectuer les choix. Mais cela consiste à développer une classe relativement lourde et dont le code des méthodes est difficile à appréhender.

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

Ce diagramme montre les deux classes d'algorithmes : `DessinUnVéhiculeLigne` et `DessinTroisVéhiculesLigne` implantant l'interface `DessinCatalogue`. La note détaillant la méthode `dessiner` de la classe `VueCatalogue` montre comment ces deux classes sont utilisées.

STRATEGY

EXAMPLE

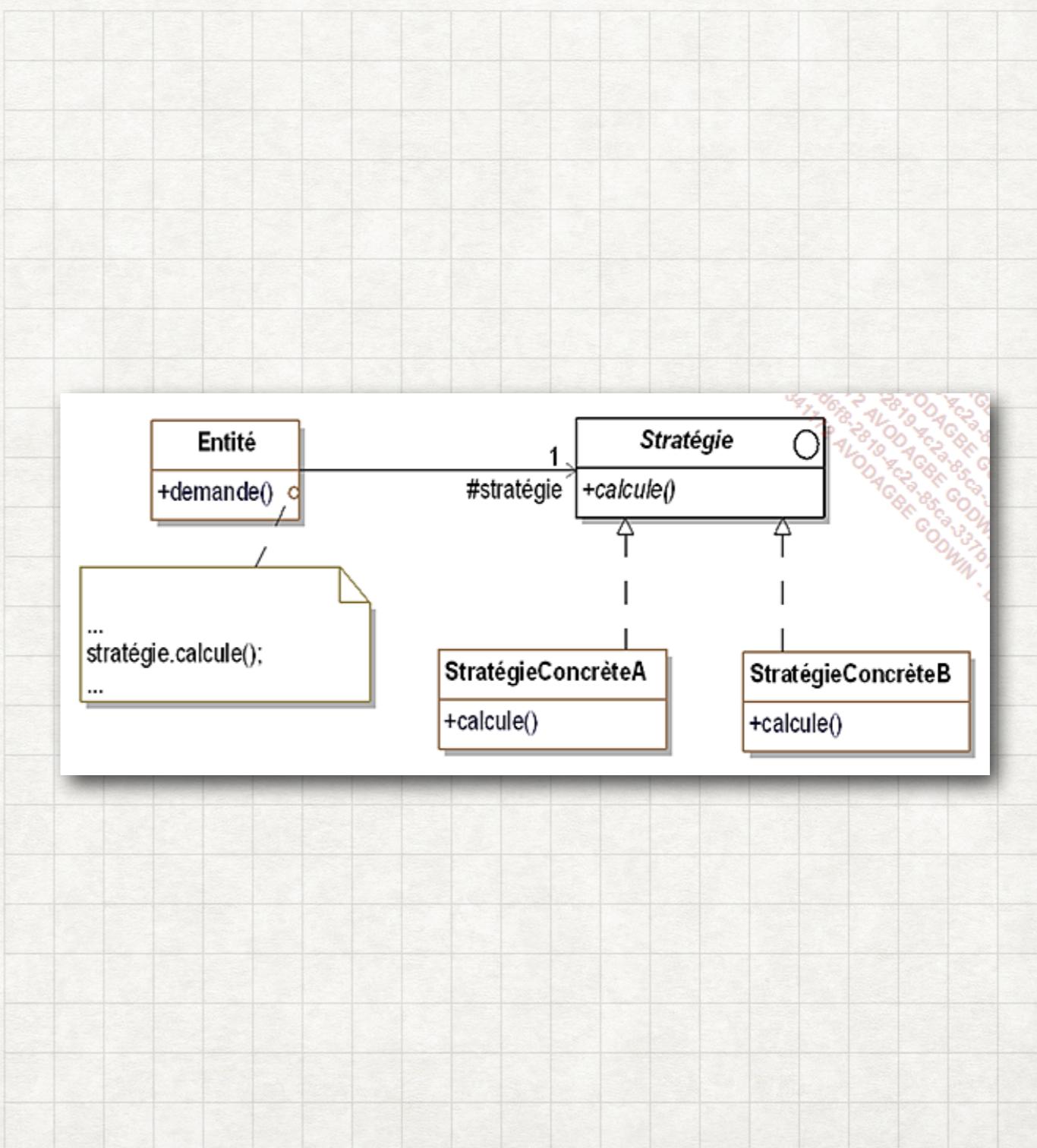


STRATEGY

STRUCTURE

Les participants au pattern sont les suivants :

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



STRATEGY COLLABORATIONS

L'entité et les instances des classes d'implantation de Stratégie interagissent pour planter 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**.

STRATEGY

DOMAINES D'UTILISATION

Le pattern est utilisé dans le cas suivant :

- 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.

DEMO

PATTERN DE COMPORTEMENT TEMPLATE METHOD



TEMPLATE METHOD

PRESENTATION

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

TEMPLATE METHOD

EXEMPLE

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

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

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

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

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

TEMPLATE METHOD

EXEMPLE

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

calculeMontantTtc :

*montantTva = montantHt * 0,196;*

montantTtc = montantHt + montantTva;

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

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

L'`calculeMontantTtc` : `calculeTva();`

`montantTtc = montantHt + montantTva;`

TEMPLATE METHOD

EXEMPLE

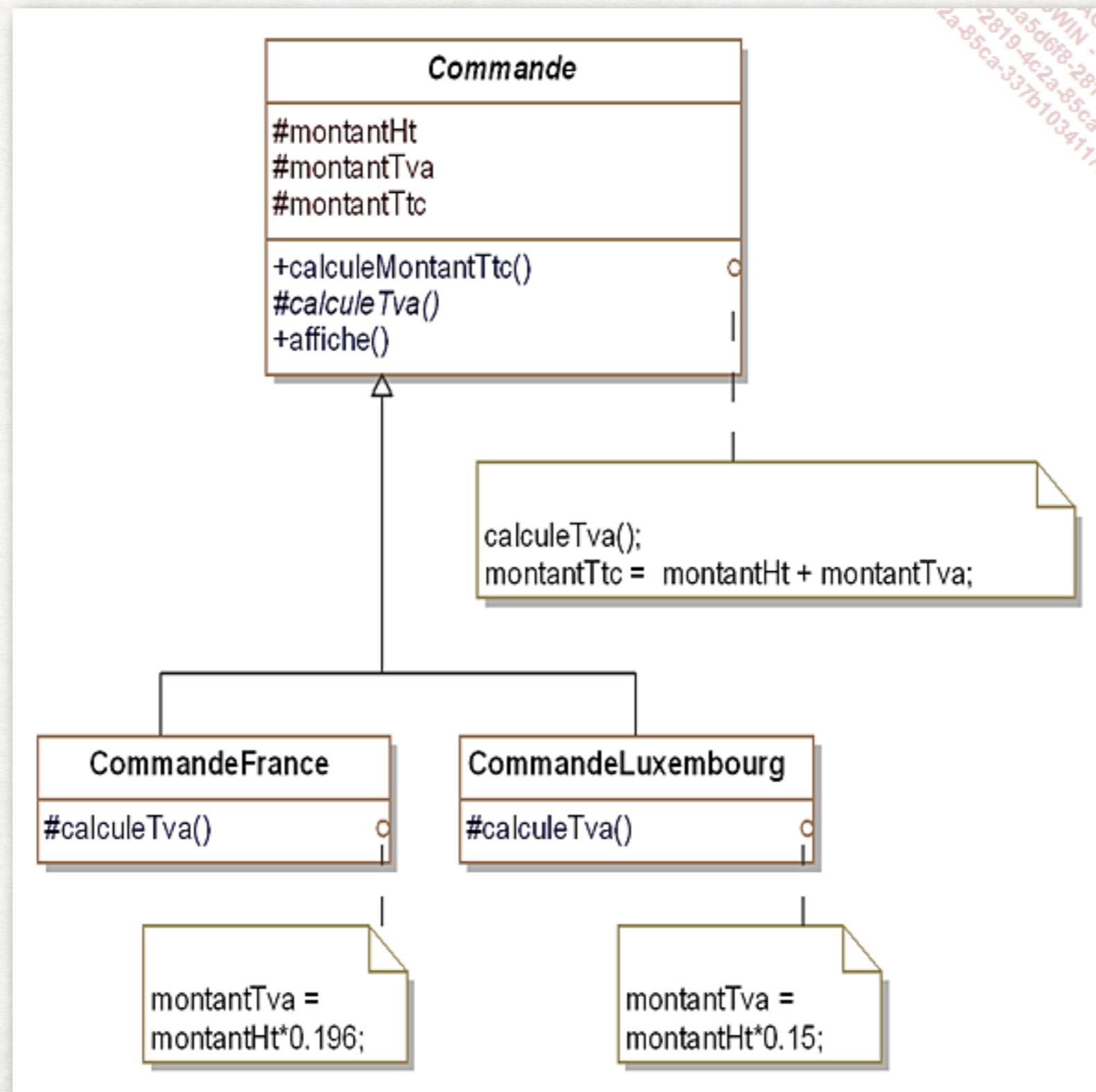
La méthode `calculeMontantTtc` peut maintenant être factorisée. Le code spécifique a été déplacé dans la méthode `calculeTva` dont l'implantation reste spécifique à chaque pays. La méthode `calculeTva` est introduite dans la classe `Commande` en tant que méthode abstraite.

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

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

TEMPLATE METHOD

EXEMPLE



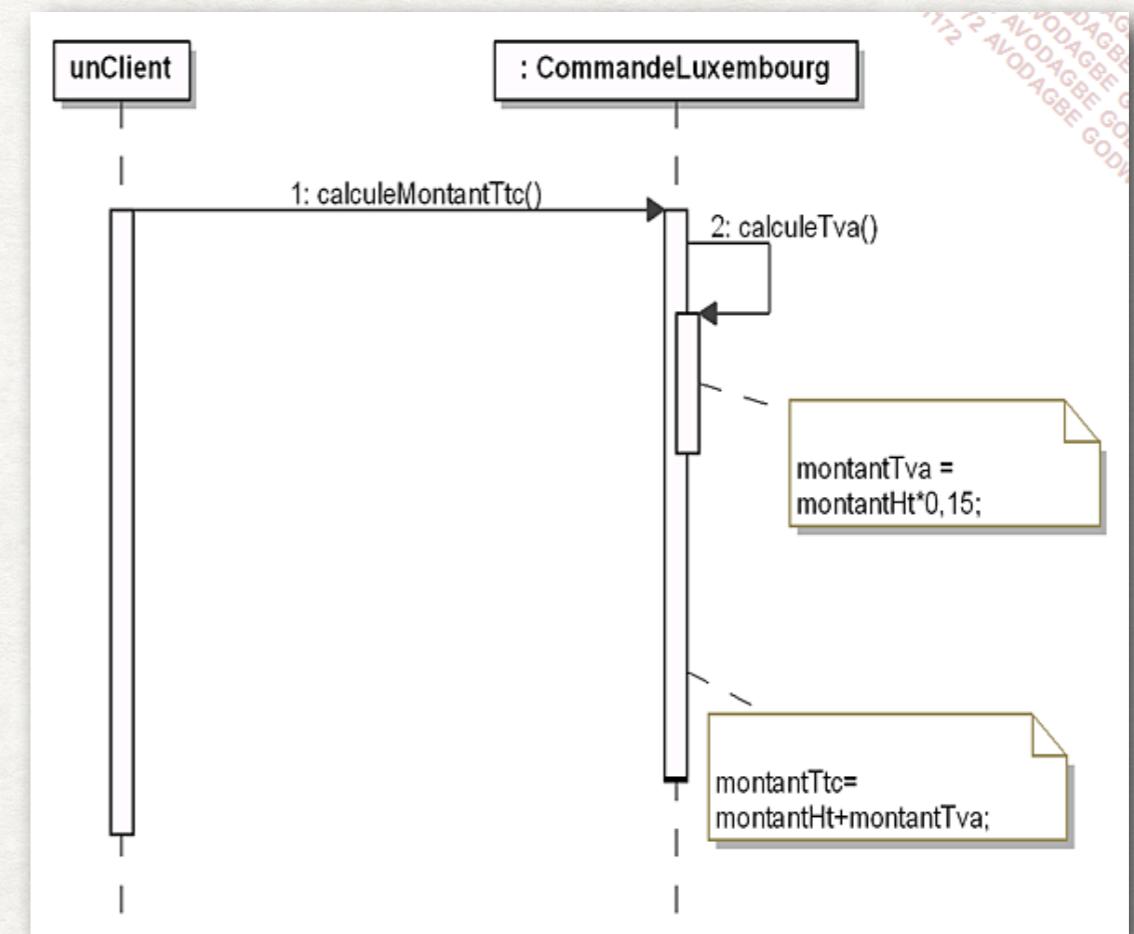
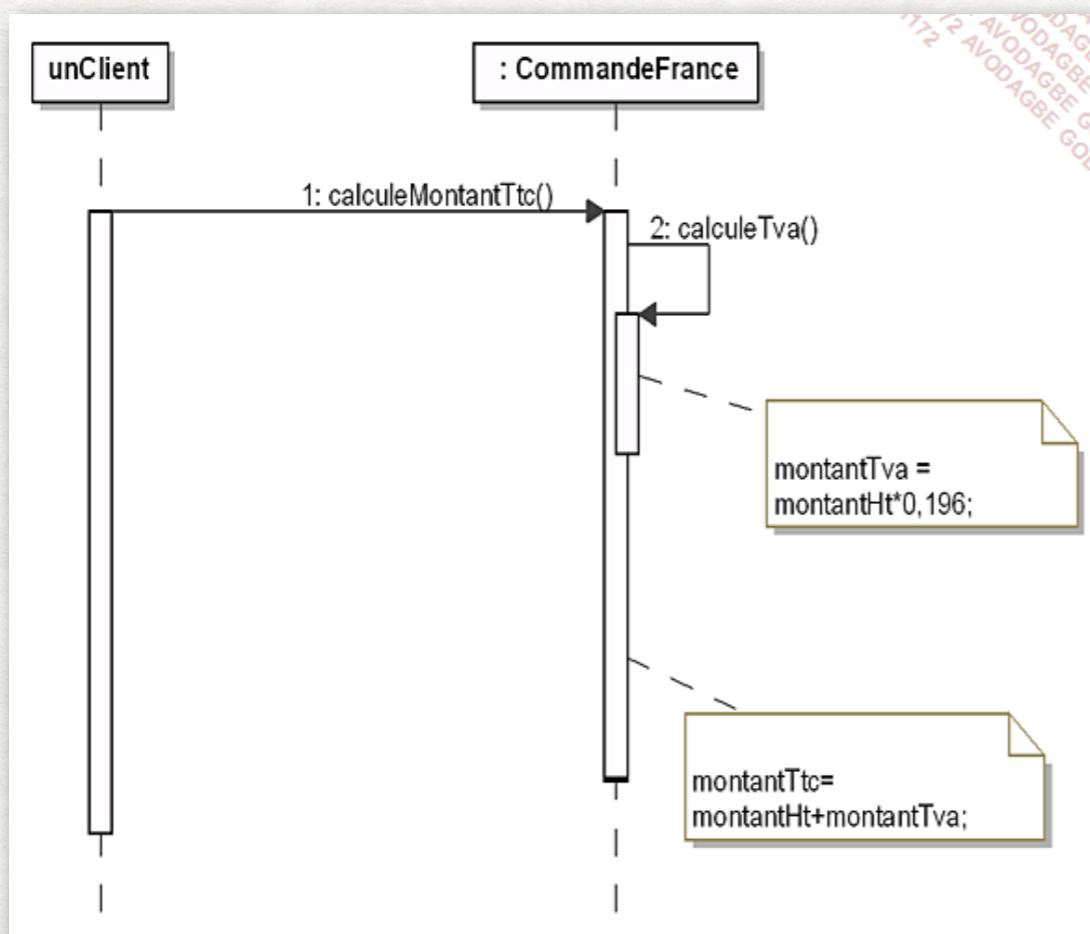
TEMPLATE METHOD

EXEMPLE

Lorsqu'un client appelle la méthode `calculeMontantTtc` d'une commande, celle-ci invoque la méthode `calculeTva`.

L'implantation de cette méthode dépend de la classe concrète de la commande :

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

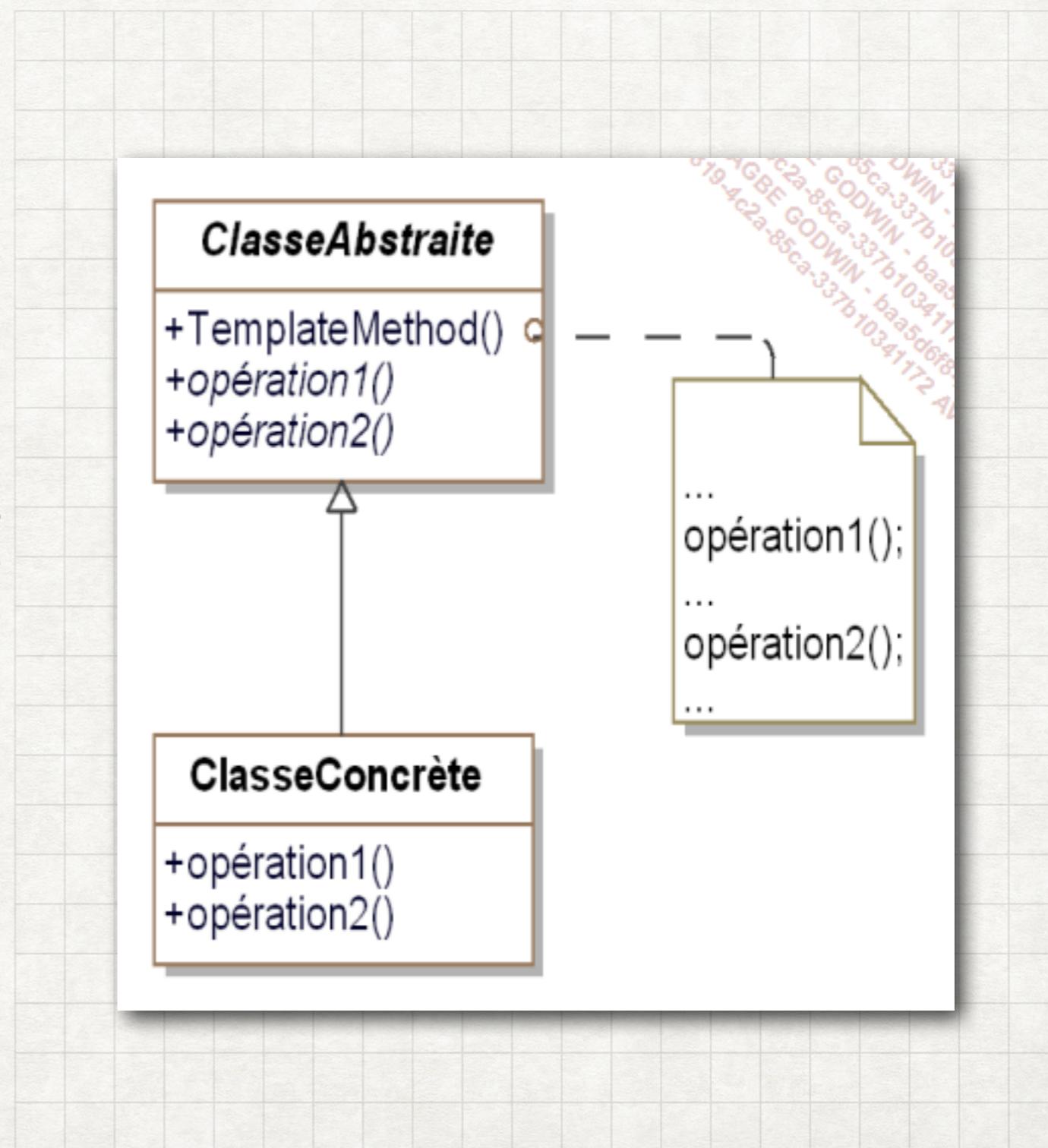


TEMPLATE METHOD

STRUCTURE

Les participants au pattern sont les suivants :

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



TEMPLATE METHOD

COLLABORATIONS

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

TEMPLATE METHOD

DOMAINES D'UTILISATION

Le pattern est utilisé dans le cas suivant :

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

DEMO

PATTERN DE COMPORTEMENT VISITOR



VISITOR

PRESENTATION

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.

VISITOR

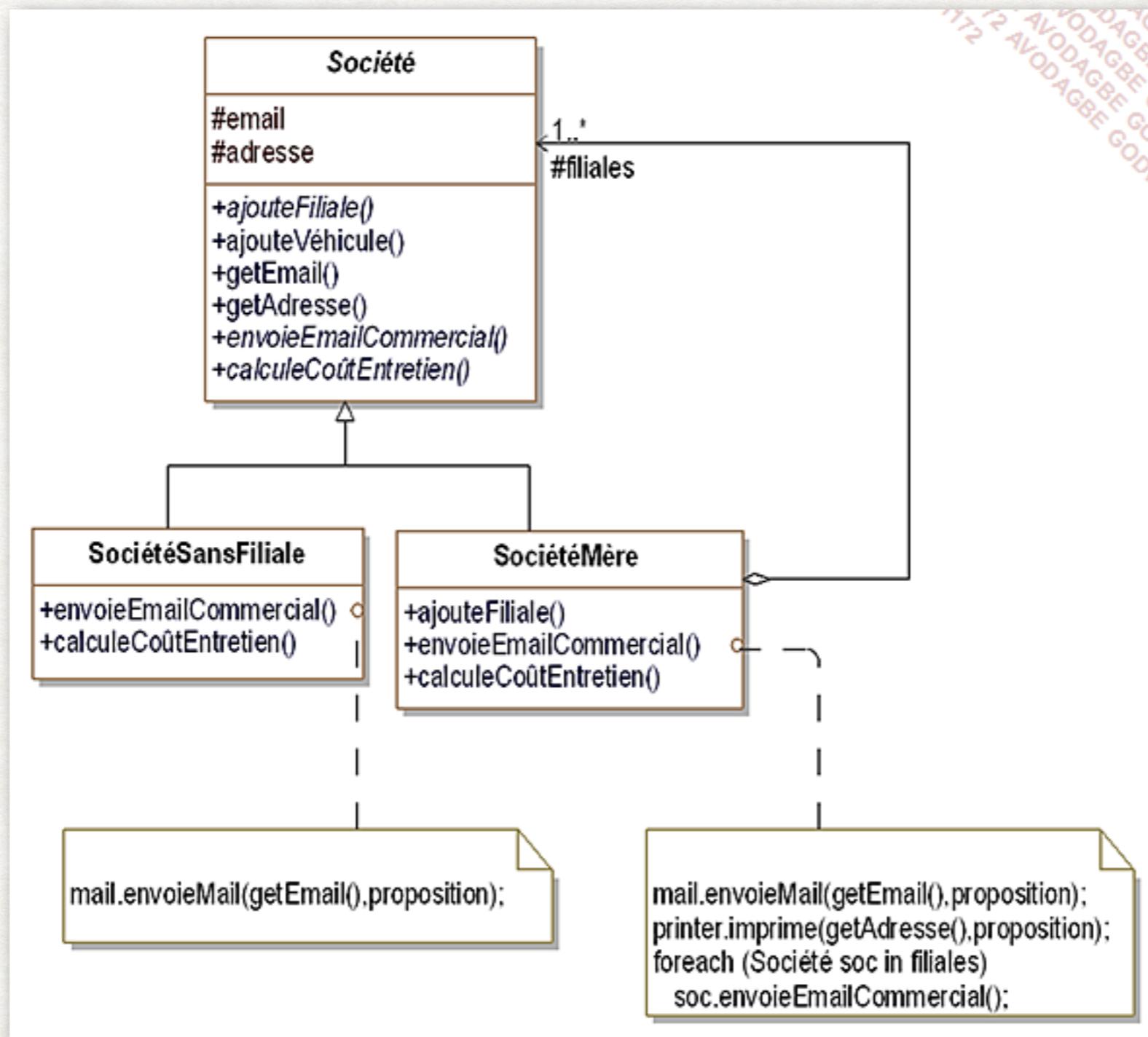
EXAMPLE

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.

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.

VISITOR

EXAMPLE



VISITOR

EXAMPLE

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` et dont le paramètre est typé par la classe à visiter.

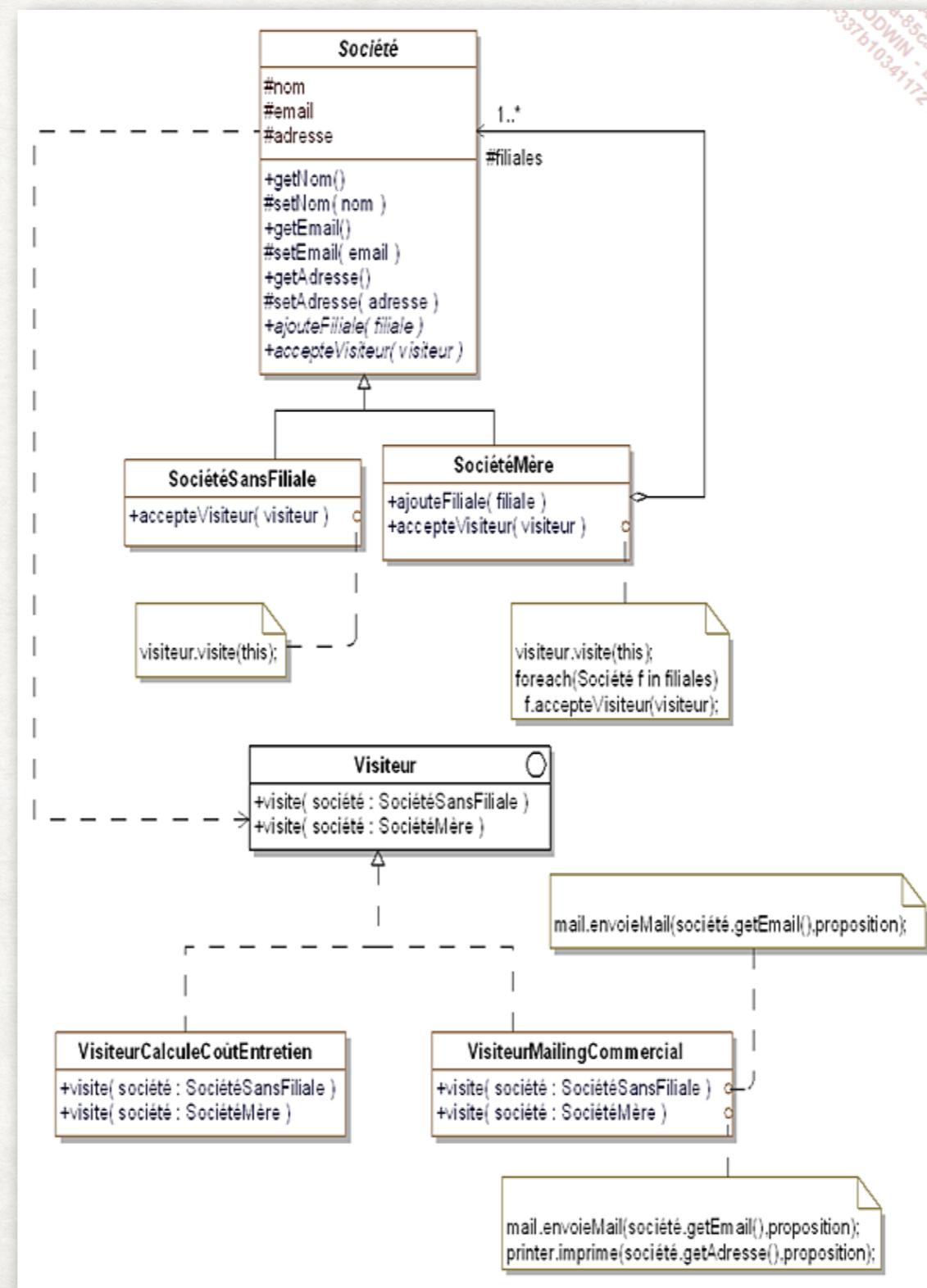
Ensuite, le visiteur est transmis à la méthode `accepteVisiteur` de ces classes. Cette méthode appelle la méthode du visiteur correspondant à sa classe. Ainsi quel que soit le nombre de fonctionnalités à planter dans un ensemble de classes, seule la méthode `accepteVisiteur` doit être écrite. Il peut 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 l'accesseur `get` des propriétés `nom`, `email` et `adresse` représenté dans le diagramme des classes par les méthodes `getNow`, `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.

VISITOR

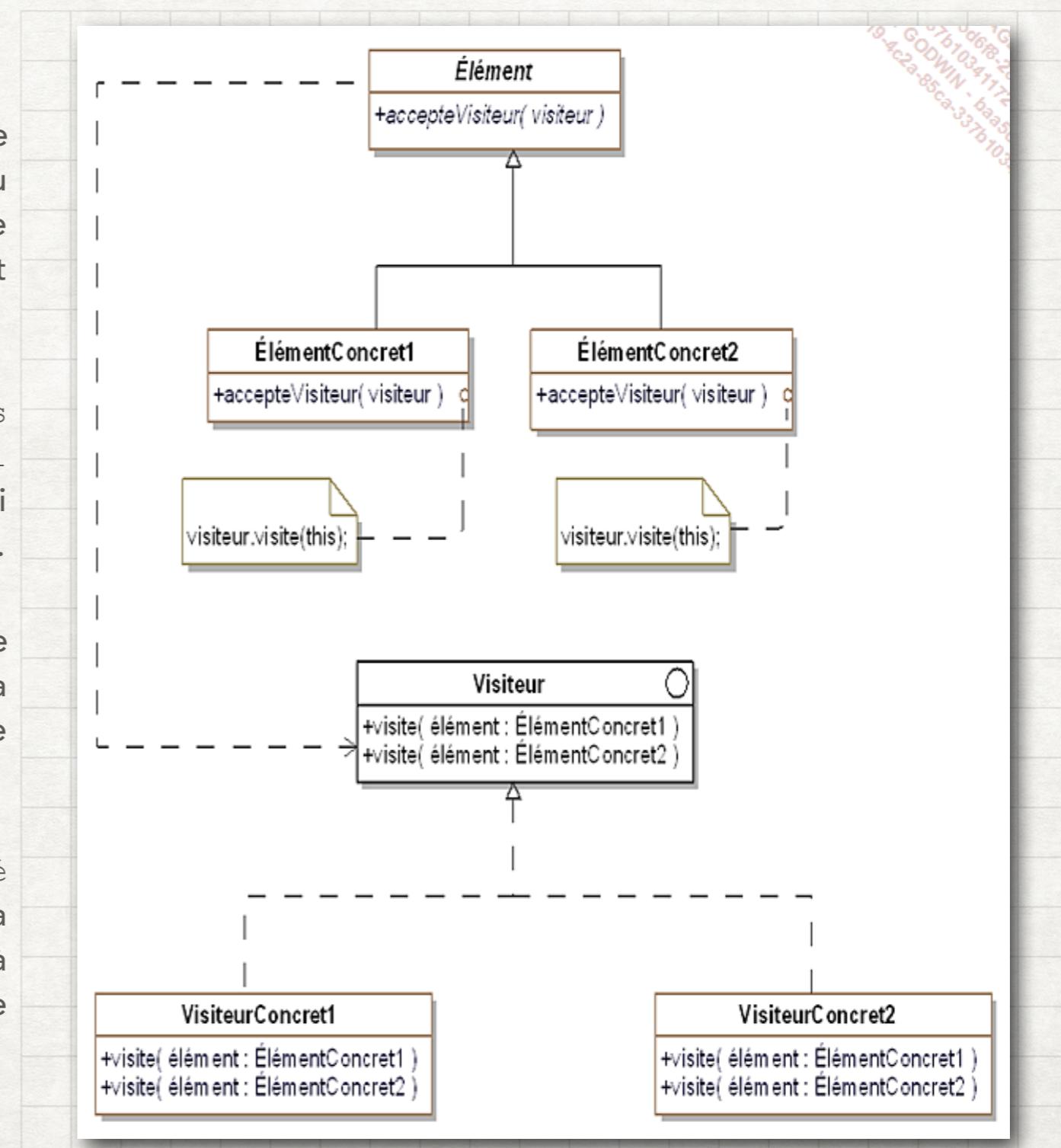
EXEMPLE



VISITOR STRUCTURE

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 VisiteurMail ingCommercial) 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 (Socié téSansFiliale et SociétéMère) implantent la méthode accepteVisiteur qui consiste à rappeler le visiteur au travers de la méthode correspondant à la classe.



VISITOR 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.

VISITOR

DOMAINES D'UTILISATION

Le pattern est utilisé dans le cas suivant :

- 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.

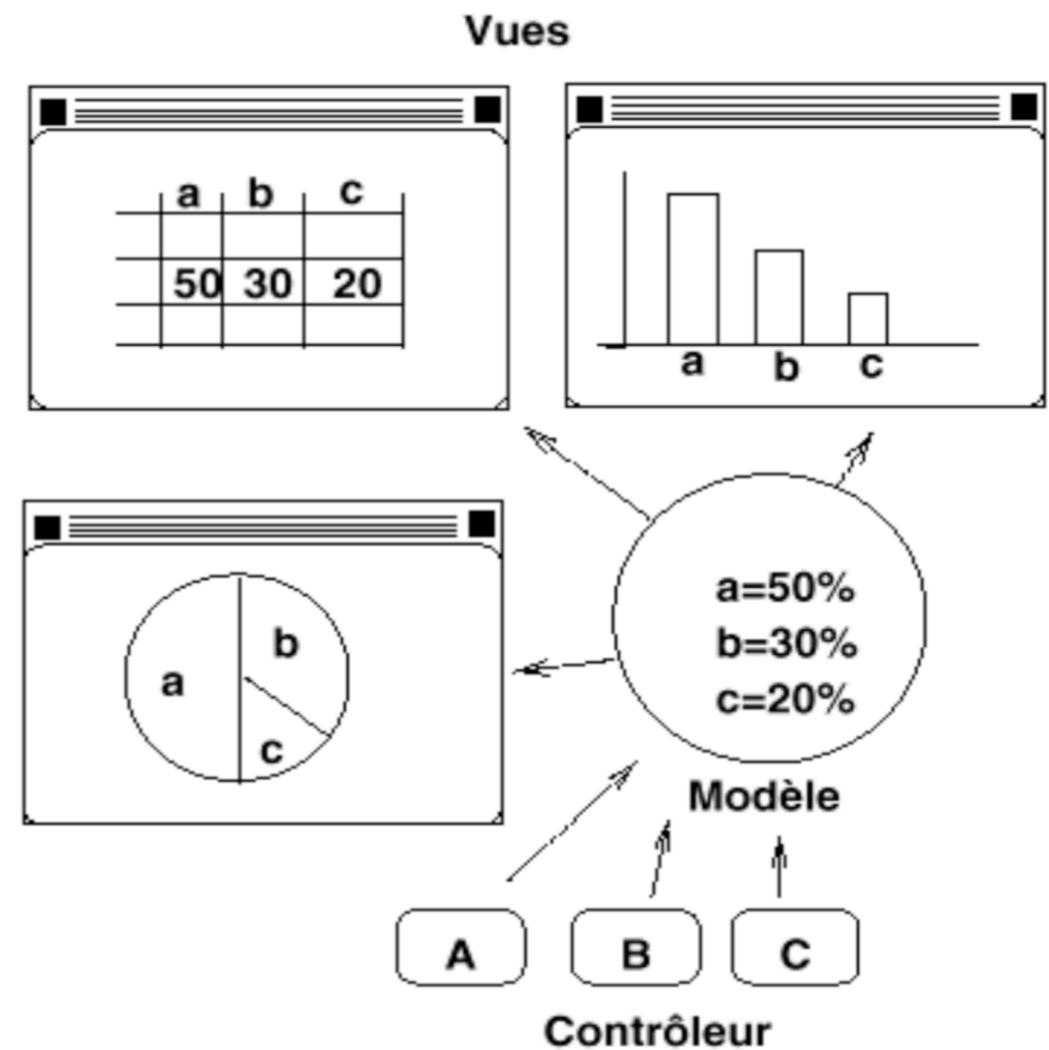
DEMO

CONCLUSION

CONCLUSION

CAS D'UTILISATION

- View-View
 - Composite
 - Decorator
- View-Model
 - Observer
- View-Controller
 - Strategy
 - Factory Method
- Controller-Model
 - Command



CONCLUSION

TROUVER LES BONS OBJETS

- Les patterns proposent des abstractions qui n'apparaissent pas "naturellement" en observant le monde réel
 - Composite : permet de traiter uniformément une structure d'objets hétérogènes
 - Strategy : permet d'implanter une famille d'algorithmes interchangeables
 - State
- Ils améliorent la flexibilité et la réutilisabilité

CONCLUSION

BIEN CHOISIR LA GRANULARITÉ

- La taille des objets peut varier considérablement : comment choisir ce qui doit être décomposé ou au contraire regroupé ?

- Facade
- Flyweight
- Abstract Factory
- Builder

CONCLUSION

PENSER « INTERFACE »

- Qu'est-ce qui fait partie d'un objet ou non ?

- Memento : mémorise les états, retour arrière
- Decorator : augmente l'interface
- Proxy : interface déléguée
- Visitor : regroupe des interfaces
- Facade : cache une structure complexe d'objet

CONCLUSION

SPÉCIFIER L'IMPLÉMENTATION

- Différence type-classe...
 - Chain of Responsibility ; même interface, mais implantations différentes
 - Composite : les Components ont une même interface dont l'implantation est en partie partagée dans le Composite
 - Command, Observer, State, Strategy ne sont souvent que des interfaces abstraites
 - Prototype, Singleton, Factory, Builder sont des abstractions pour créer des objets qui permettent de penser en termes d'interfaces et de leur associer différentes implantations

CONCLUSION

EN BREF, LES DESIGN PATTERNS...

- C'est ...
 - une description d'une solution classique à un problème récurrent
 - une description d'une partie de la solution... avec des relations avec le système et les autres parties...
 - une technique d'architecture logicielle
- Ce n'est pas ...
 - une brique : Un pattern dépend de son environnement
 - une règle : Un pattern ne peut pas s'appliquer mécaniquement
 - une méthode : Ne guide pas une prise de décision : un pattern est la décision prise

FIN