DESIGN PATTERNS

EN PROGRAMMATION PAR OBJETS

PLAN

- Design patterns
  - De quoi s'agit-il?
  - Pourquoi faut-il les utiliser?
- Design patterns essentiels
- Quelques design patterns
- Concepts avancés en design patterns
  - Langages de design pattern
  - Niveaux d'architecture et design patterns
- Références

UN PEU D ’HISTOIRE

- 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 ObjectOriented Software*" et écrit par Erich Gamma, Richard Helm, Ralph Johnson et John Vlissides.

[Gamma, et al., 1984] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns*, Reading, MA, Addison-Wesley, 1984.

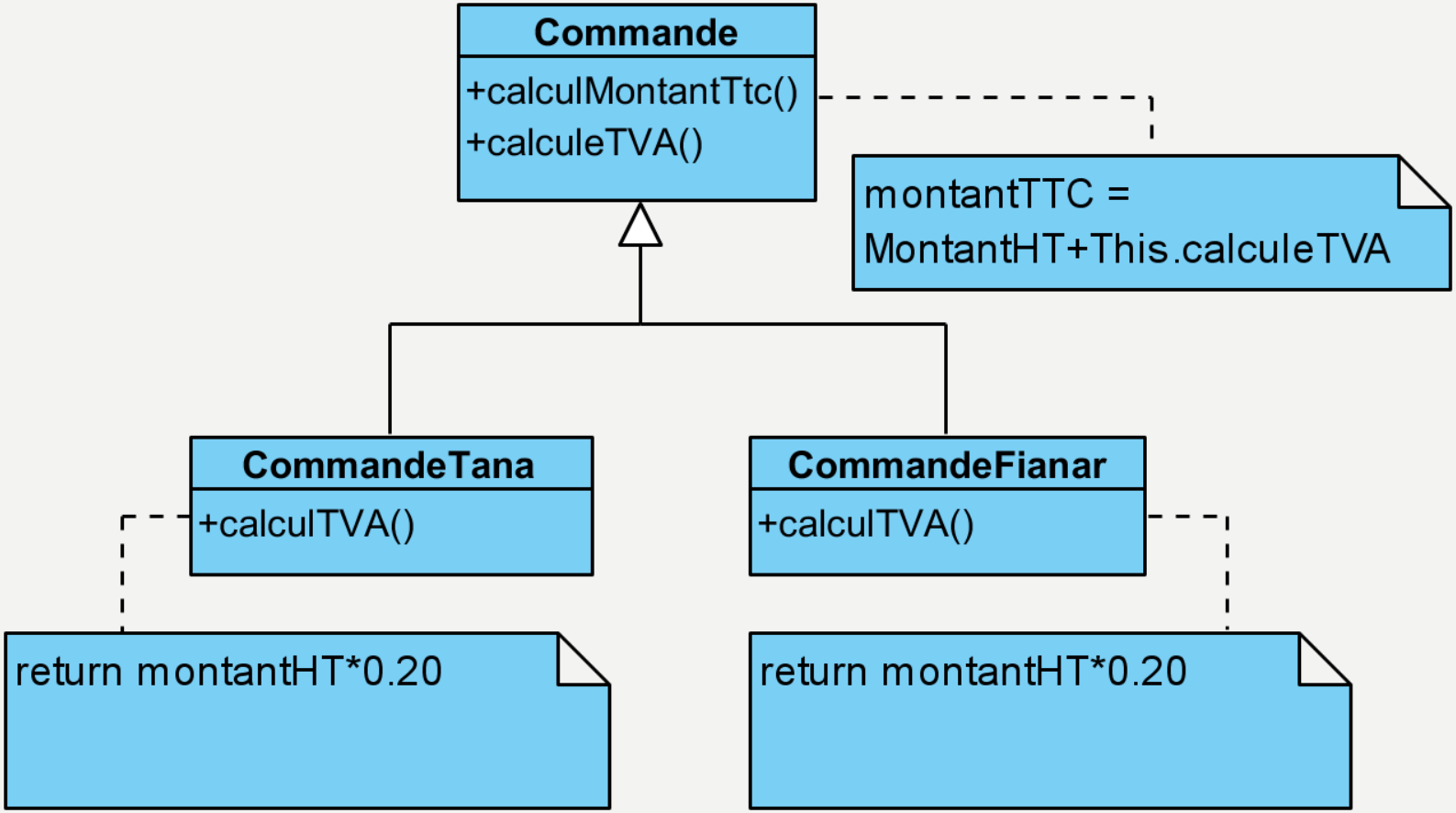
QU'EST-CE QU'UN DESIGN PATTERN?

- Le nom du pattern résume le problème de design, ses solutions et ses conséquences en un mot ou deux.
- Le problème décrit quand appliquer un pattern.
- La solution décrit les éléments qui forment le design, les interrelations, les responsabilités et les collaborations.
- Les conséquences décrivent les résultats et les compromis qui résultent de l'application du pattern.
- ***Les patterns répondent à des problèmes de conception de logiciels dans le cadre de la programmation par objets***
- **Les patterns sont basés sur les bonnes pratiques de la programmation par objets.**

POURQUOI UTILISER LES DESIGN PATTERNS?

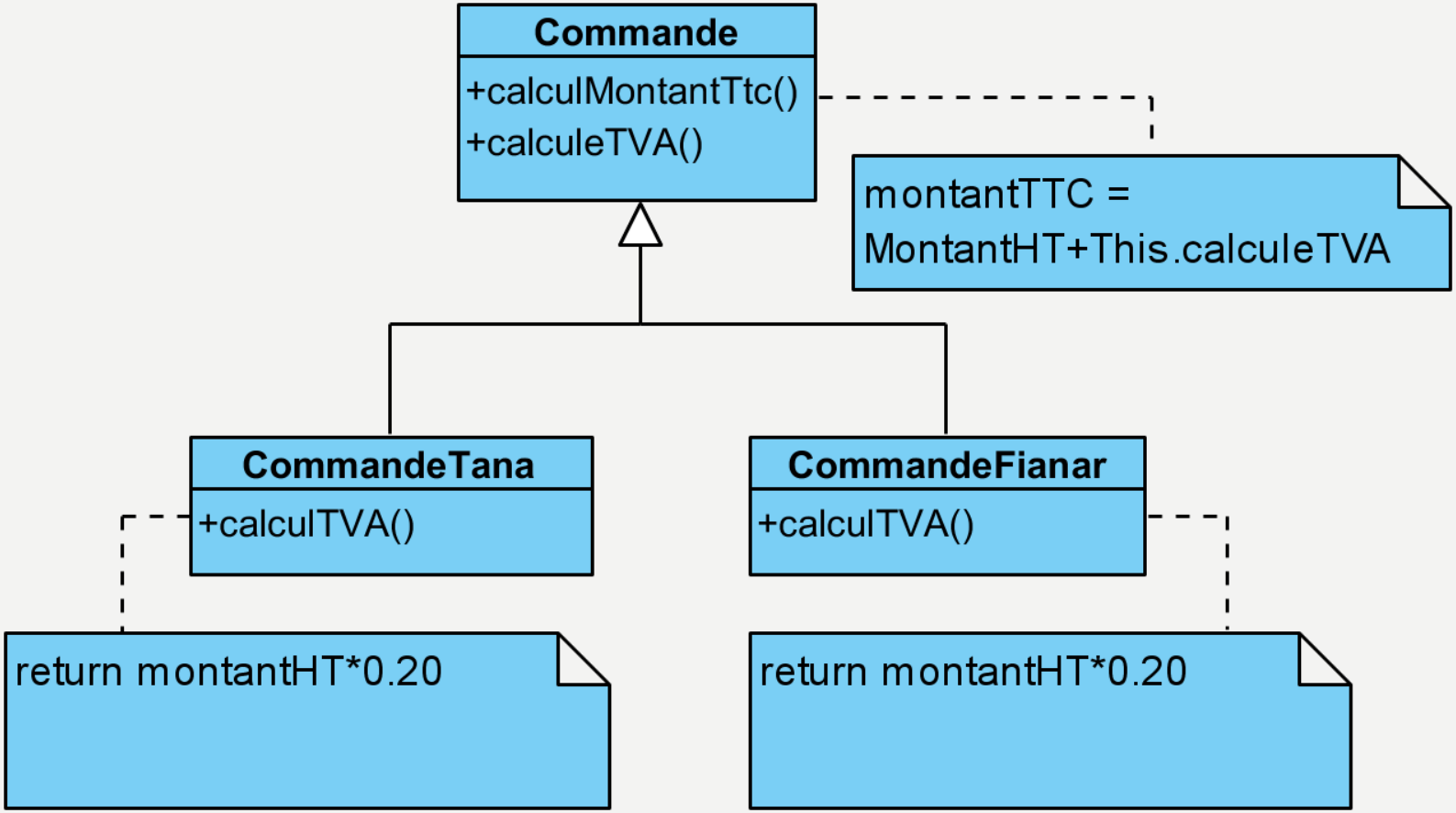
- Pour se concentrer sur de bons designs objets
- Pour apprendre en suivant de bons exemples
- Pour écrire du code facilement compréhensible par les autres programmeurs

EX : PATTERN TEMPLATE METHOD



Dans ce pattern, la méthode **calculeMontantTtc** invoque la méthode **calculeTva** qui est abstraite dans la classe Commande.

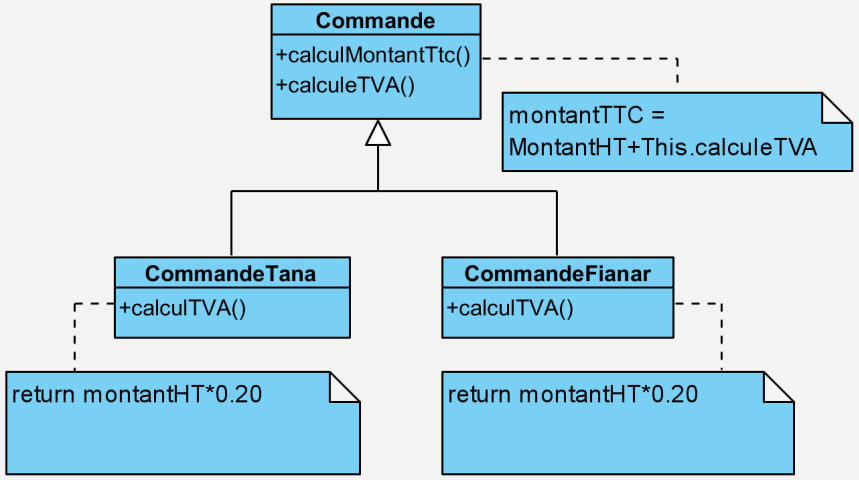
EX : PATTERN TEMPLATE METHOD



Elle est définie ans les sousclasses de ***Commande*** à savoir les classes ***CommandeTana*** et ***CommandeFianar***

EX : PATTERN TEMPLATE METHOD

La méthode **calculeMontantTtc** est appelée méthode "*patron*" (Template method). Elle introduit un algorithme basé sur une méthode abstraite.



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

DESCRIPTION DES PATTERNS DE CONCEPTION

- Langages de description des patterns
  - UML introduit par l'OMG (Object Management Group)
  - Java

DESIGN PATTERNS ESSENTIELS

- ***Créationnel*** : processus de création des objets
- ***Structurel*** : composition des classes ou des objets
- ***Comportemental*** : comment les classes et les objets interagissent et distribuent les responsabilités

		Purpose		
		Creational	Structural	Behavioral
Scope	Class	Factory Method (107)	Adapter (139)	Interpreter (243) Template Method (325)
	Object	Abstract Factory (87) Builder (97) Prototype (117) Singleton (127)	Adapter (139) Bridge (151) Composite (163) Decorator (175) Facade (185) Proxy (207)	Chain of Responsibility (223) Command (233) Iterator (257) Mediator (273) Memento (283) Flyweight (195) Observer (293) State (305) Strategy (315) Visitor (331)

CATALOGUE

- **Abstract Factory** : a pour objectif la création d'objets regroupés en familles sans devoir connaître les classes concrètes destinées à la création de ces objets ;
- **Builder** : permet de séparer la construction d'objets complexes de leur implantation de sorte qu'un client puisse créer ces objets complexes avec des implantations différentes ;
- **Factory Method** : a pour but d'introduire une méthode abstraite de création d'un objet en reportant aux sous-classes concrètes la création effective ;
- **Prototype** : permet la création de nouveaux objets par duplication d'objets existants appelés prototypes qui disposent de la capacité de clonage ;

CATALOGUE

- **Singleton** : permet de s'assurer qu'une classe ne possède qu'une seule instance et de fournir une méthode unique retournant cette instance ;
- **Adapter** : a pour but de convertir l'interface d'une classe existante en l'interface attendue par des clients également existants afin qu'ils puissent travailler ensemble ;
- **Bridge** : a pour but de séparer les aspects conceptuels d'une hiérarchie de classes de leur implantation ;
- **Composite** : offre un cadre de conception d'une composition d'objets dont la profondeur de composition est variable, la conception étant basée sur un arbre ;
- **Decorator** : permet d'ajouter dynamiquement des fonctionnalités supplémentaires à un objet ;

CATALOGUE

- **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 ;

CATALOGUE

- **Interpreter** : fournit un cadre pour donner une représentation par objets de la grammaire d'un langage afin d'évaluer, en les interprétant, des expressions écrites dans ce langage ;
- **Iterator** : fournit un accès séquentiel à une collection d'objets sans que les clients se préoccupent de l'implantation de cette collection ;
- **Mediator** : construit un objet dont la vocation est la gestion et le contrôle des interactions au sein d'un ensemble d'objets sans que ses éléments se connaissent mutuellement ;
- **Memento** : sauvegarde et restaure l'état d'un objet ;
- **Observer** : construit une dépendance entre un sujet et des observateurs de façon à ce que chaque modification du sujet soit notifiée aux observateurs afin qu'ils puissent mettre à jour leur état ;

CATALOGUE

- **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.

CHOIX ET UTILISATION

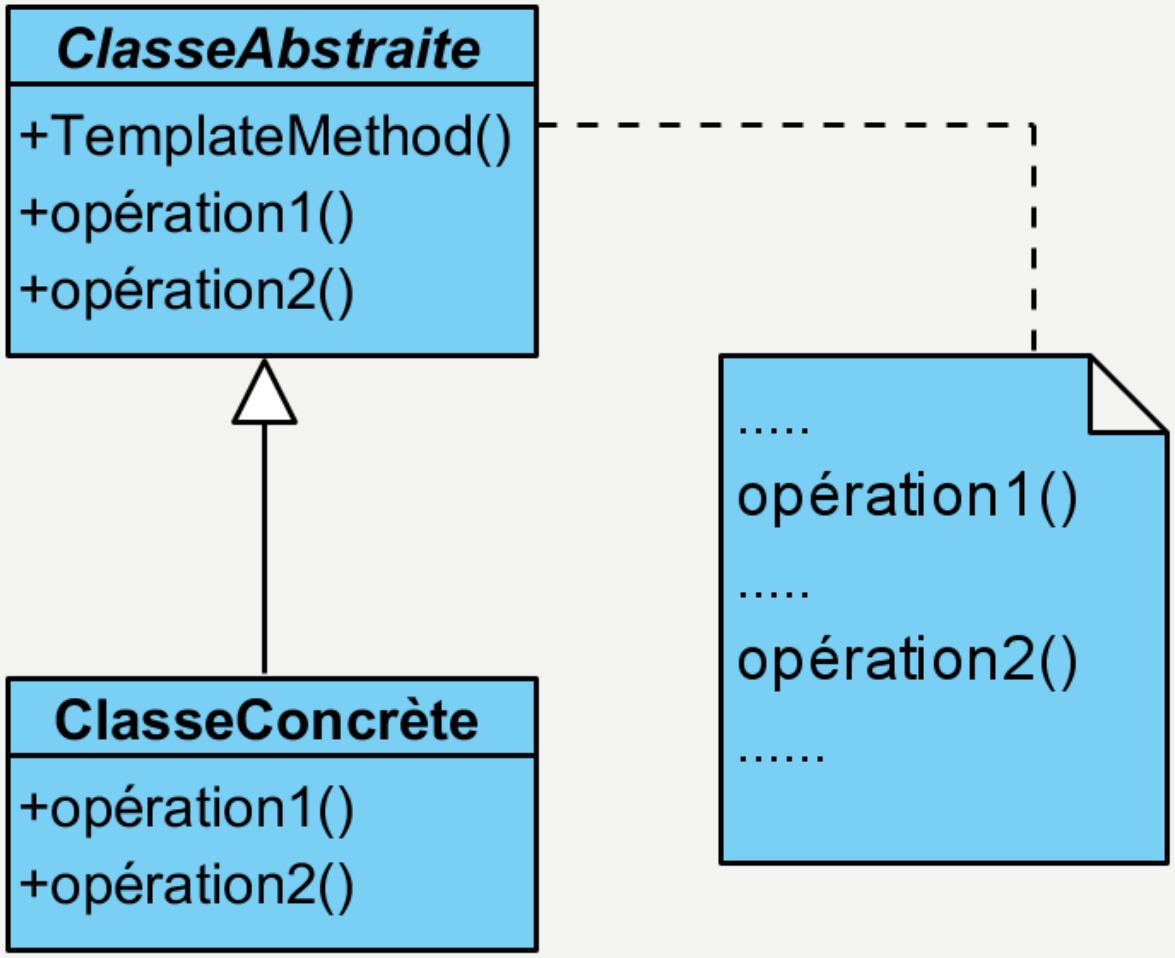
- Pour savoir s'il existe un pattern de conception qui réponde à un problème donné, une première étape consiste à regarder les descriptions des patterns et à déterminer s'il existe un ou plusieurs patterns dont la description s'approche de celle du problème.
- Une fois qu'un pattern est choisi, son utilisation dans une application comprend plusieurs étapes.

CHOIX ET UTILISATION

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

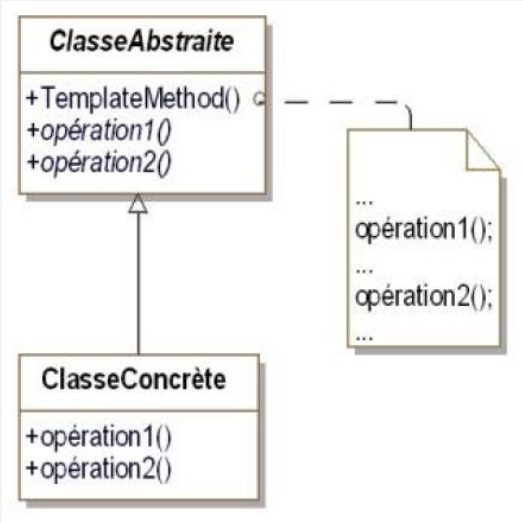
1. étudier de façon approfondie sa structure générique qui sert de base pour utiliser un pattern ;
2. renommer les classes et les méthodes introduites dans la structure générique. En effet, dans la structure générique d'un pattern, le nom des classes et des méthodes est abstrait. À l'inverse, une fois 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 ;
3. adapter la structure générique pour répondre aux contraintes de l'application, ce qui peut impliquer des changements dans le schéma d'objets.

STRUCTURE GÉNÉRIQUE D'UN TEMPLATE METHOD

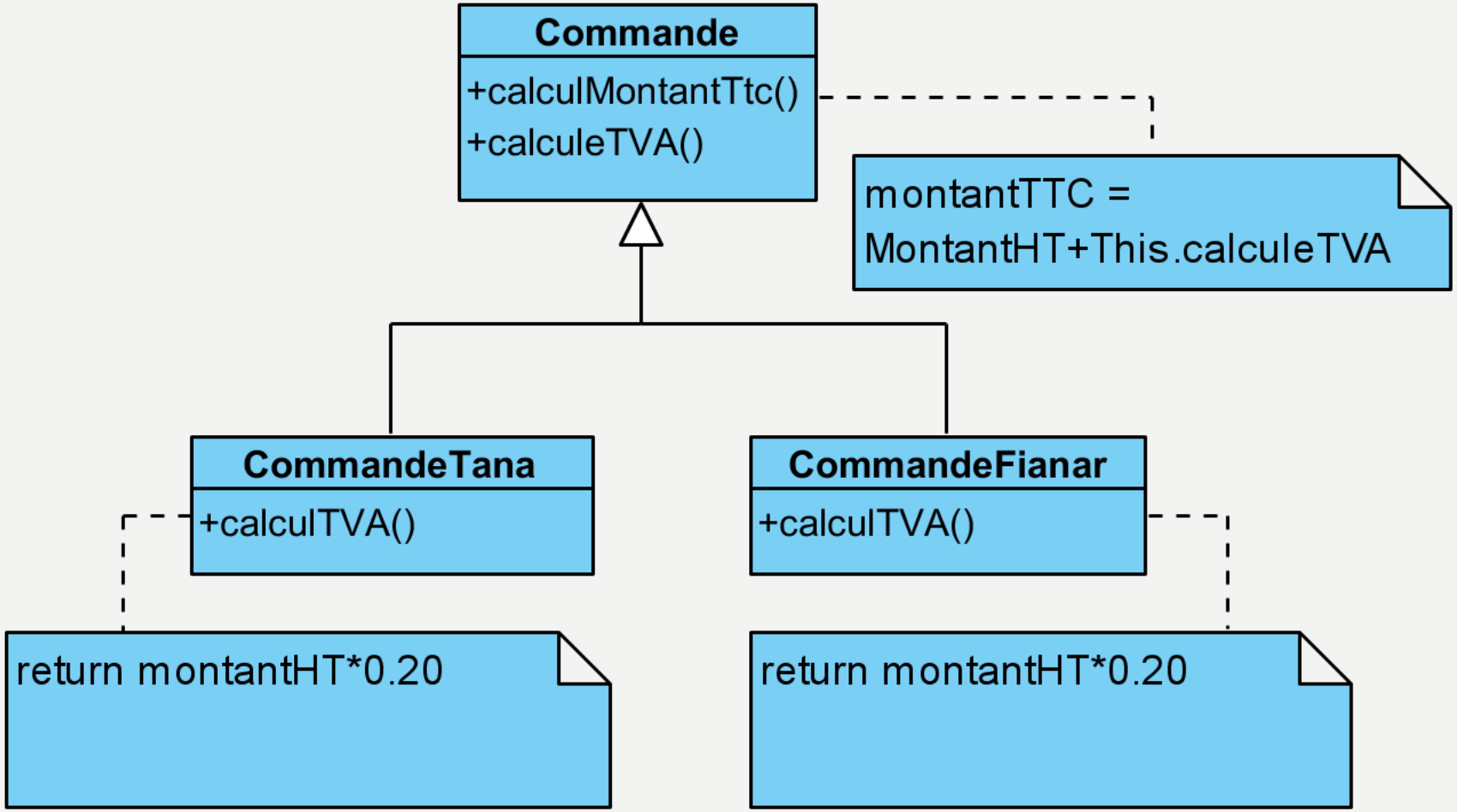


STRUCTURE GÉNÉRIQUE D'UN TEMPLATE METHOD

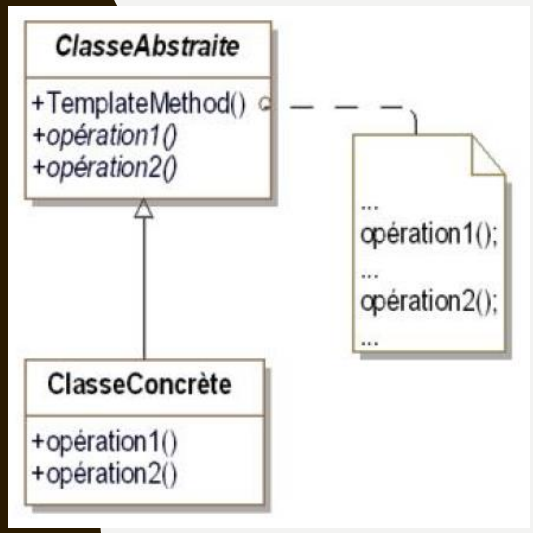
- Nous voulons adapter cette structure dans le cadre d'une application de commerce où la méthode du calcul de la TVA n'est pas incluse dans la classe Commande mais dans les sousclasses concrètes de la classe abstraite Ville. Ces sous-classes contiennent toutes les méthodes de calcul des taxes spécifiques à chaque Ville.
- La méthode **calculeTVA** de la classe Commande va alors invoquer la méthode **calculeTVA** de la sous-classe de la ville concerné au travers d'une instance de cette sous-classe.
- Cette instance peut être transmise en paramètre lors de la création de la commande.



EXEMPLE D'UTILISATION AVEC ADAPTATION DU PATTERN TEMPLATE METHOD



EXEMPLE D'UTILISATION AVEC ADAPTATION DU PATTERN TEMPLATE METHOD



ETUDE DE CAS

- 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.

CAHIERS 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.

CAHIERS DES CHARGES

- Lors de la commande d'un véhicule, le système construit la liasse des documents nécessaires comme la demande d'immatriculation, le certificat de cession et le bon de commande. Ces documents sont disponibles au format PDF ou au format HTML.
- Le système permet également de solder les véhicules difficiles à vendre, à savoir ceux qui sont dans le stock depuis longtemps. Il permet également une gestion des clients, en particulier des sociétés possédant des filiales afin de leur proposer, par exemple, l'achat d'une flotte de véhicules.
- Lors de la visualisation du catalogue, il est possible de visualiser des animations associées à un véhicule. Le catalogue peut être présenté avec un ou trois véhicules par ligne.
- La recherche dans le catalogue peut s'effectuer à l'aide de mots clés et d'opérateurs logiques (et, ou).
- Il est possible d'accéder au système via une interface web classique ou au travers d'un système de web services.

PRISE EN COMPTE DES PATTERNS DE CONCEPTION

Description	Pattern de conception
Construire les objets du domaine (Automobile à essence, automobile électrique, etc.).	Abstract Factory
Construire les liasses de documents nécessaires en cas d'acquisition d'un véhicule.	Builder, Prototype
Créer les commandes.	Factory Method
Créer la liasse vierge de documents.	Singleton
Gérer des documents PDF.	Adapter
Implanter des formulaires en HTML ou à l'aide d'une applet.	Bridge

ABSTRACT FACTORY
PROBLÉMATIQUE

- Dans la plupart des langages à objets, la création d'objets se fait grâce au mécanisme d'instanciation qui consiste à créer un nouvel objet par appel de l'opérateur new paramétré par une classe (et éventuellement des arguments du constructeur de la classe dont le but est de donner aux attributs leur valeur initiale). Un tel objet est donc une ***instance*** de cette classe.
- Les langages les plus utilisés aujourd'hui comme Java, C++ ou C# utilisent le mécanisme de l'opérateur new.

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

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

```
public Document construitDoc(String typeDoc)
{
        Document résultat;
        if (typeDoc.equals("PDF"))
                résultat = new DocumentPDF();
        else if (typeDoc.equals("RTF"))
                résultat = new DocumentRTF();
        else if (typeDoc.equals("HTML"))
                résultat = new DocumentHTML();
        // suite de la méthode
}
```

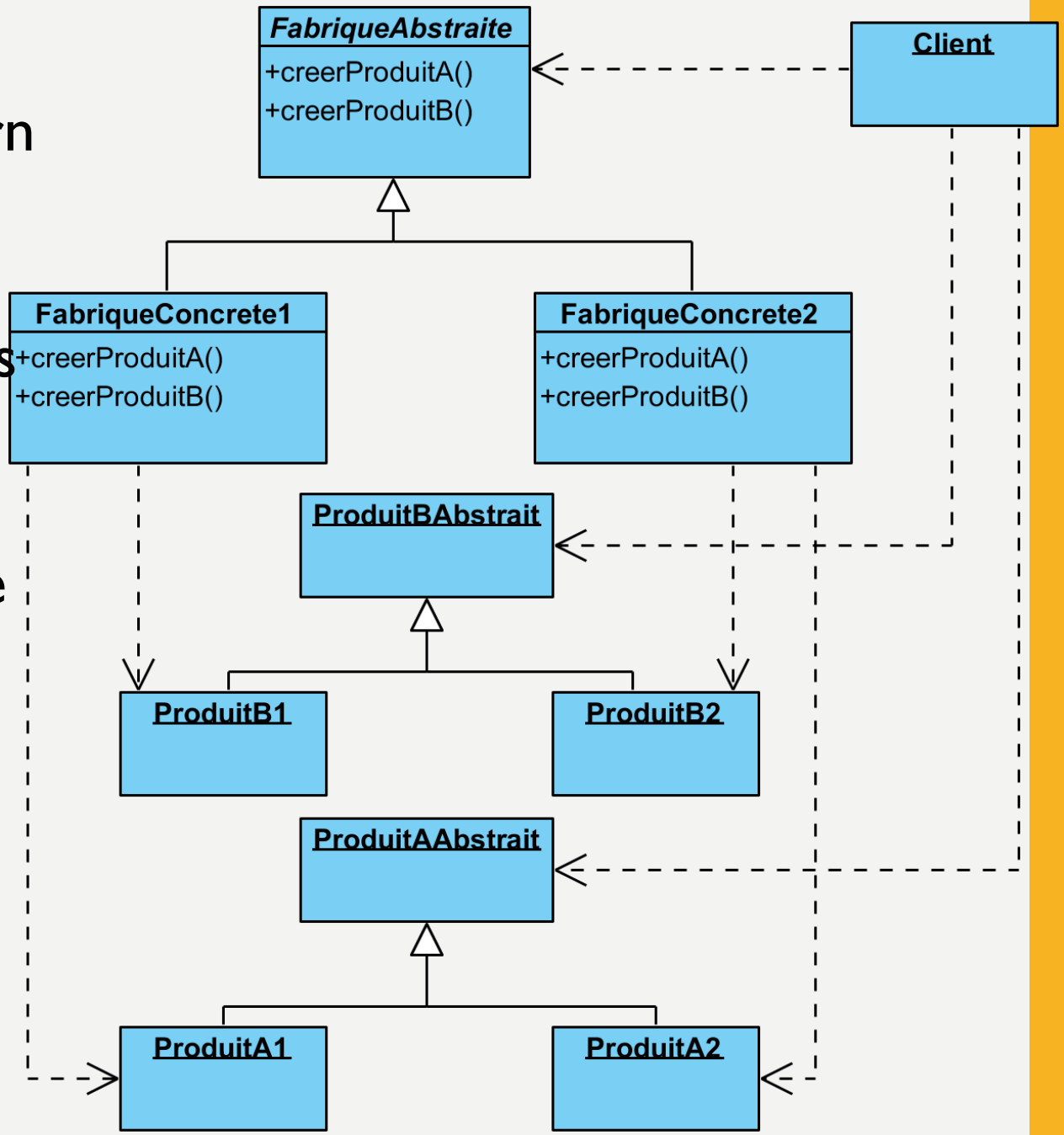
Cet exemple nous montre qu'il est difficile de paramétrer le mécanisme de création d'objets, la classe transmise en paramètre à l'opérateur ***new*** ne pouvant être substituée par une variable. L'utilisation d'instructions conditionnelles dans le code du client est souvent pratiquée avec l'inconvénient 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.

SOLUTIONS

- Les patterns ***Abstract Factory, Builder, Factory Method et Prototype*** proposent une solution pour paramétrer la création d'objets.
- Dans le cas des patterns ***Abstract Factory, Builder*** et ***Prototype,*** un objet est utilisé comme paramètre du système. Cet objet est chargé de l'instanciation des classes. Ainsi toute modification dans la hiérarchie des classes n'entraîne que des changements dans la modification de cet objet.
- Le pattern ***Factory Method*** propose un paramétrage basé sur les sous-classes de la classe cliente. Ses sous-classes implantent la création des objets. Tout changement dans la hiérarchie des classes entraîne par conséquent une modification de la hiérarchie des sous-classes de la classe cliente.

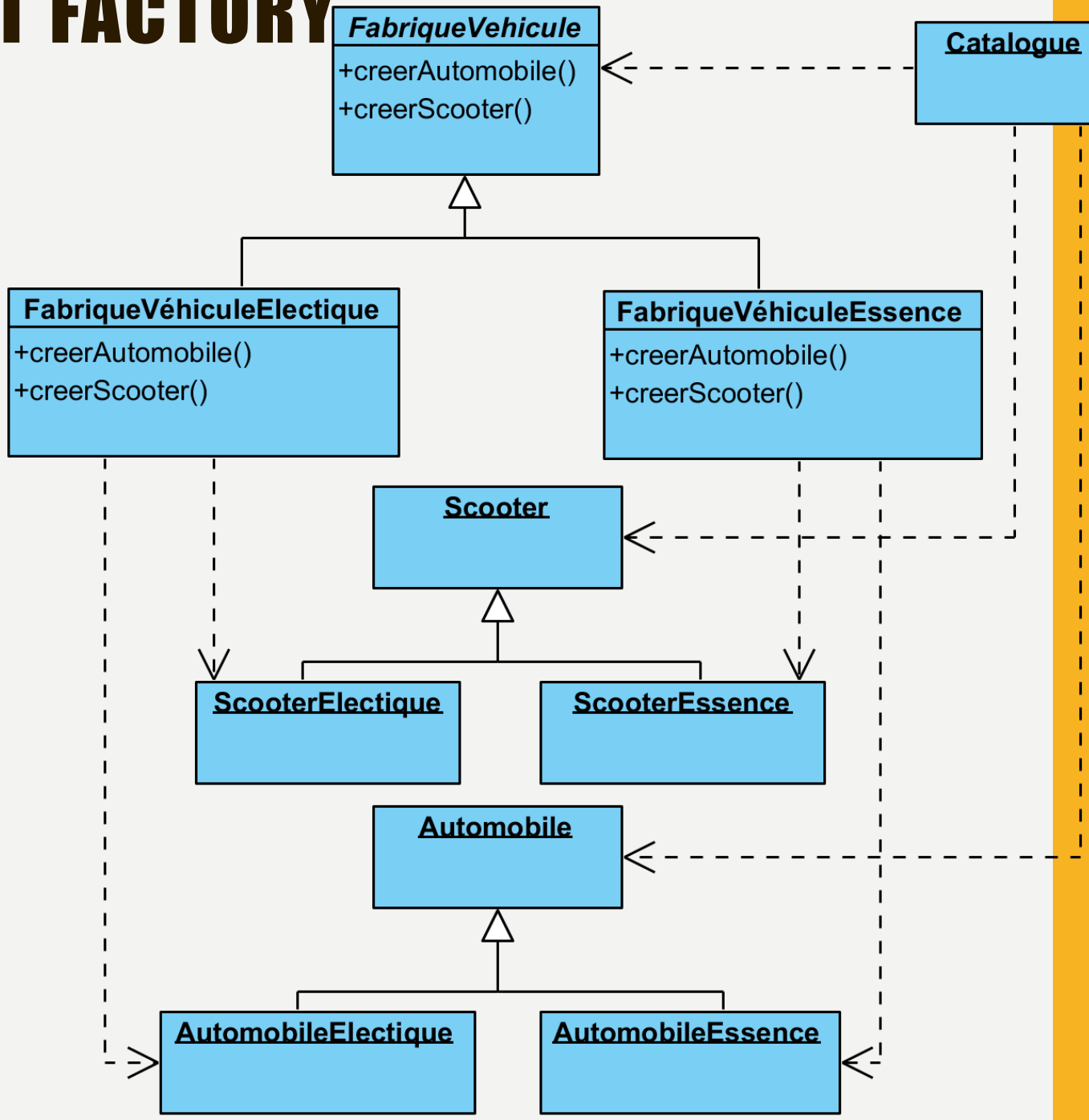
STRUCTURE GÉNÉRIQUE ABSTRACT FACTORY

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

Le pattern ***Abstract Factory*** *appliqué à des familles de véhicules*



PATTERN BUILDER

PATTERN BUIDLER DESCRIPTION

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

PATTERN BUIDLER EXEMPLE

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

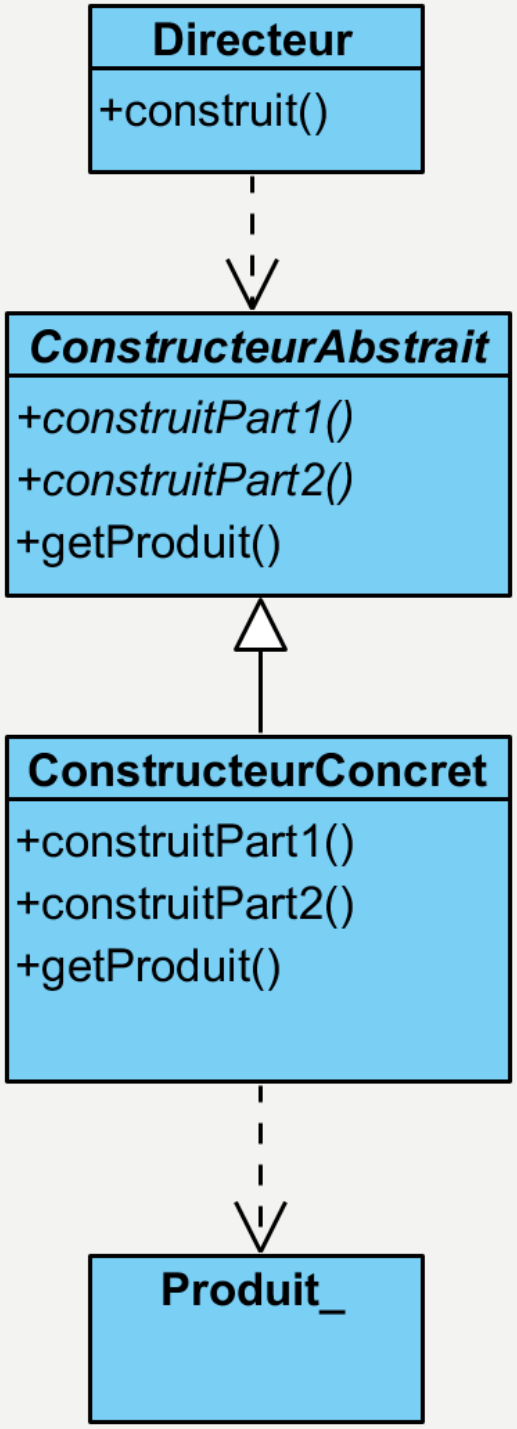
PATTERN BUIDLER EXEMPLE

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

PATTERN BUIDLER
NB

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

STRUCTURE GÉNÉRIQUE DU PATTERN BUIDLER



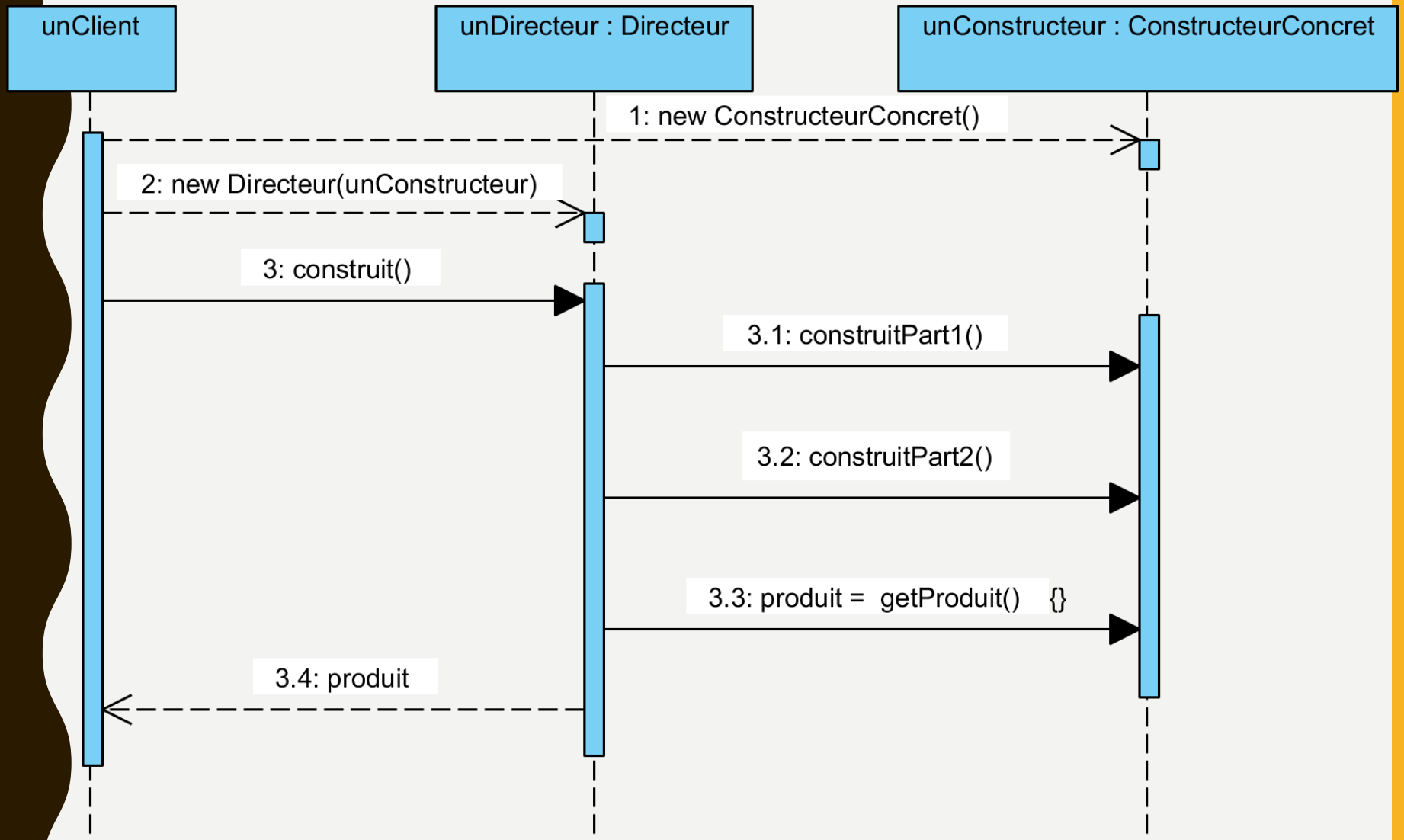
PATTERN BUILDER
PARTICIPANTS

- Les participants au pattern sont les suivants :
- **ConstructeurAbstrait** (**ConstructeurLiasseVéhicule**) est la classe introduisant les signatures des méthodes construisant les différentes parties du produit ainsi que la signature de la méthode permettant d'obtenir le produit, une fois celui-ci construit ;
- **ConstructeurConcret** (**ConstructeurLiasseVéhiculeHtml** et **ConstructeurLiasseVéhiculePdf**) est la classe concrète implantant les méthodes du constructeur abstrait ;
- **Produit** (**Liasse**) est la classe définissant le produit. Elle peut être abstraite et posséder plusieurs sous-classes concrètes (**LiasseHtml** et **LiassePdf**) en cas d'implantations différentes ;
- **Directeur** est la classe chargée de construire le produit au travers de l'interface du constructeur abstrait.

PATTERN 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 suivante illustre ce fonctionnement avec un diagramme de séquence UML.

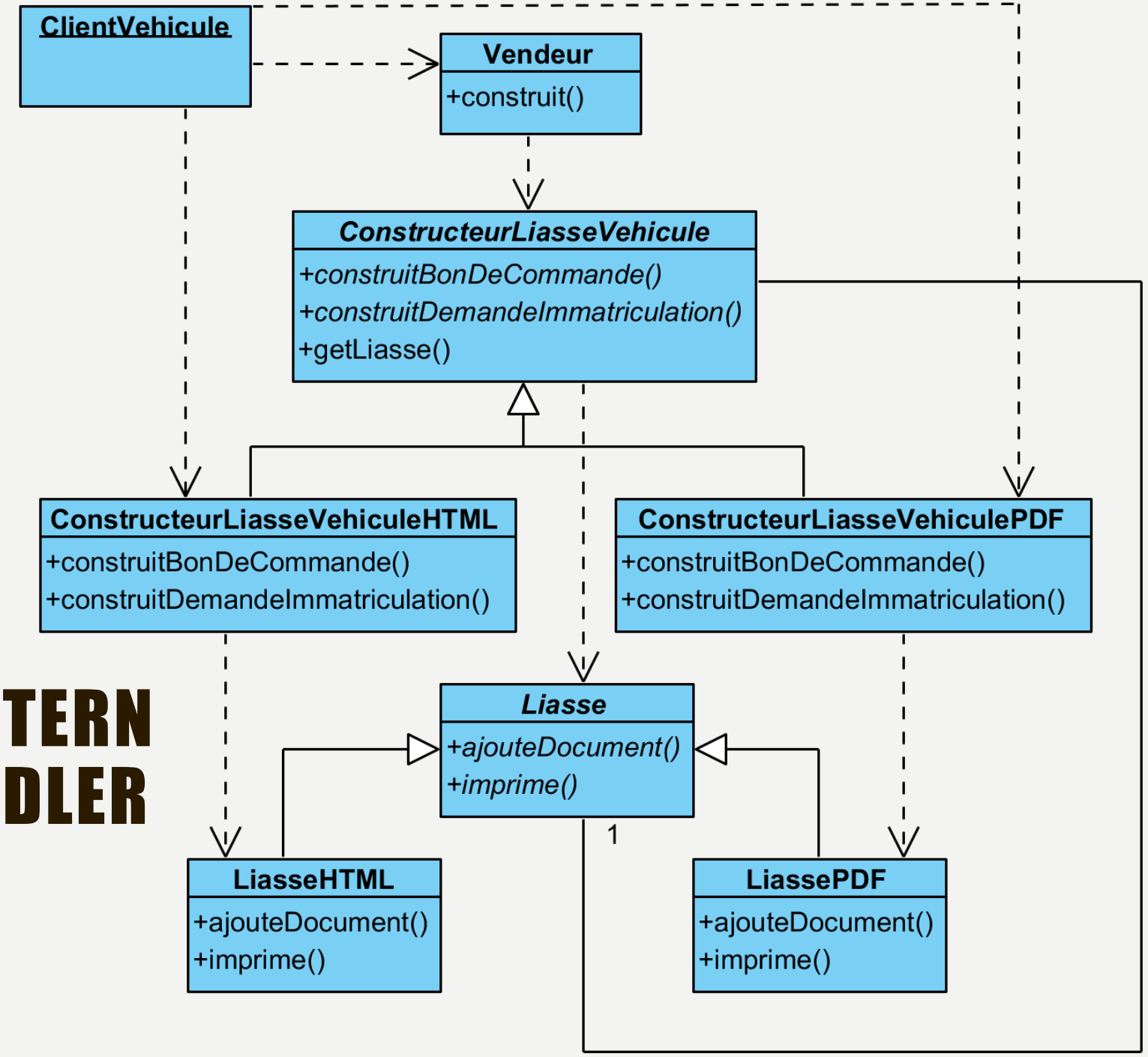
PATTERN BUILDER
COLLABORATIONS



PATTERN BUILDER
DOMAINES D'UTILISATION

- Le pattern est utilisé dans les domaines suivants :
- un client a besoin de construire des objets complexes sans connaître leur implantation ;
- un client a besoin de construire des objets complexes ayant plusieurs représentations ou implantations.

PATTERN BUIDLER



FACTORY METHOD

FACTORY METHOD
DESCRIPTION

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

FACTORY METHOD
EXEMPLE

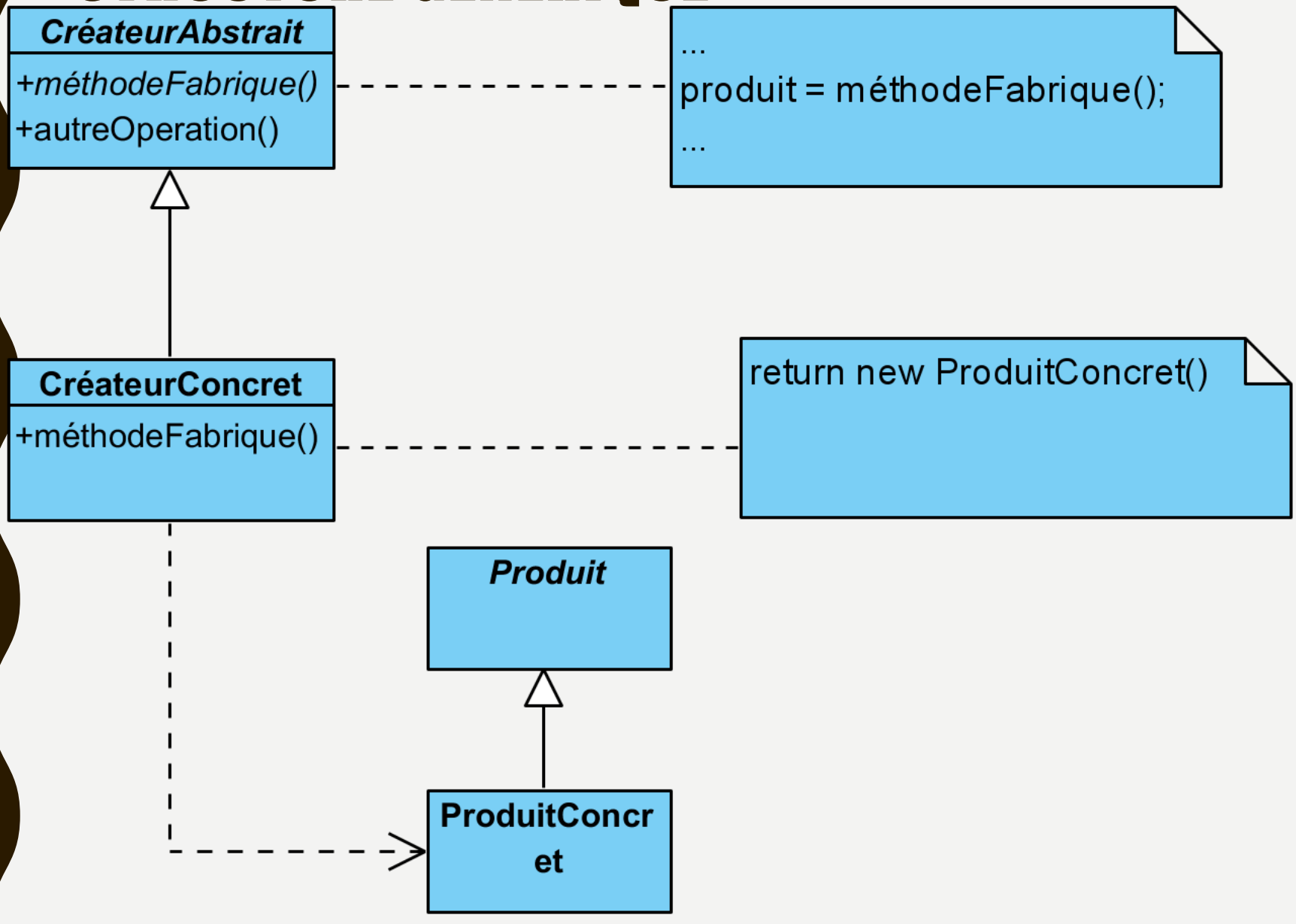
Nous nous intéressons aux clients et aux commandes. La classe Client introduit la méthode **créeCommande** qui doit créer la commande. Certains clients commandent un véhicule en payant au comptant et d'autres clients utilisent un crédit. En fonction de la nature du client, la méthode **créeCommande** 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 :

FACTORY METHOD
EXEMPLE

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

- ***Une telle conception est basée sur le pattern Factory Method, la méthode créeCommande étant la méthode de fabrique.***

FACTORY METHOD
STRUCTURE GÉNÉRIQUE

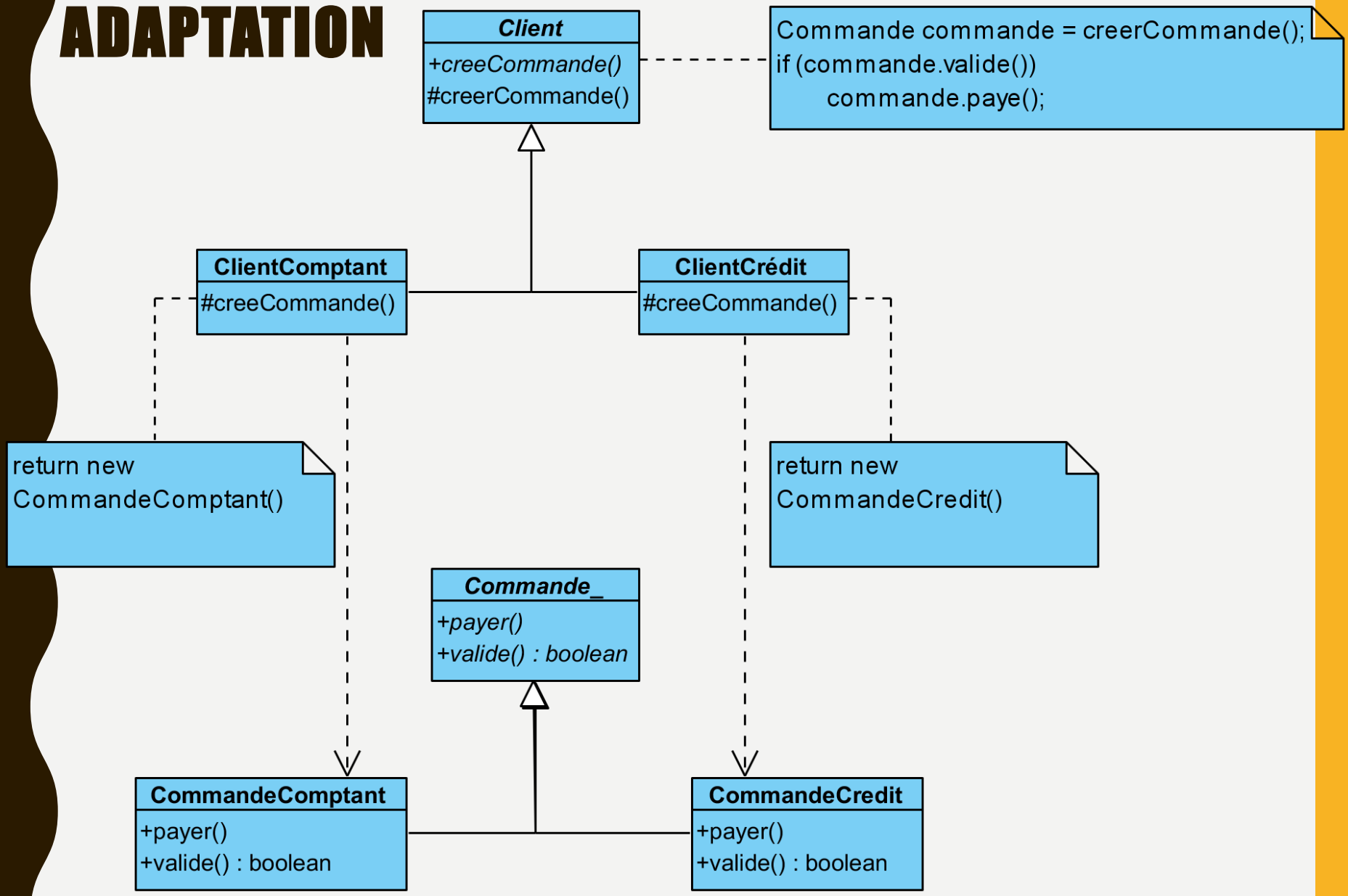


FACTORY METHOD
PARTICIPANTS

- **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

ADAPTATION



PATTERN PROTOTYPE

PROTOTYPE
DESCRIPTION

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

PROTOTYPE
EXEMPLE

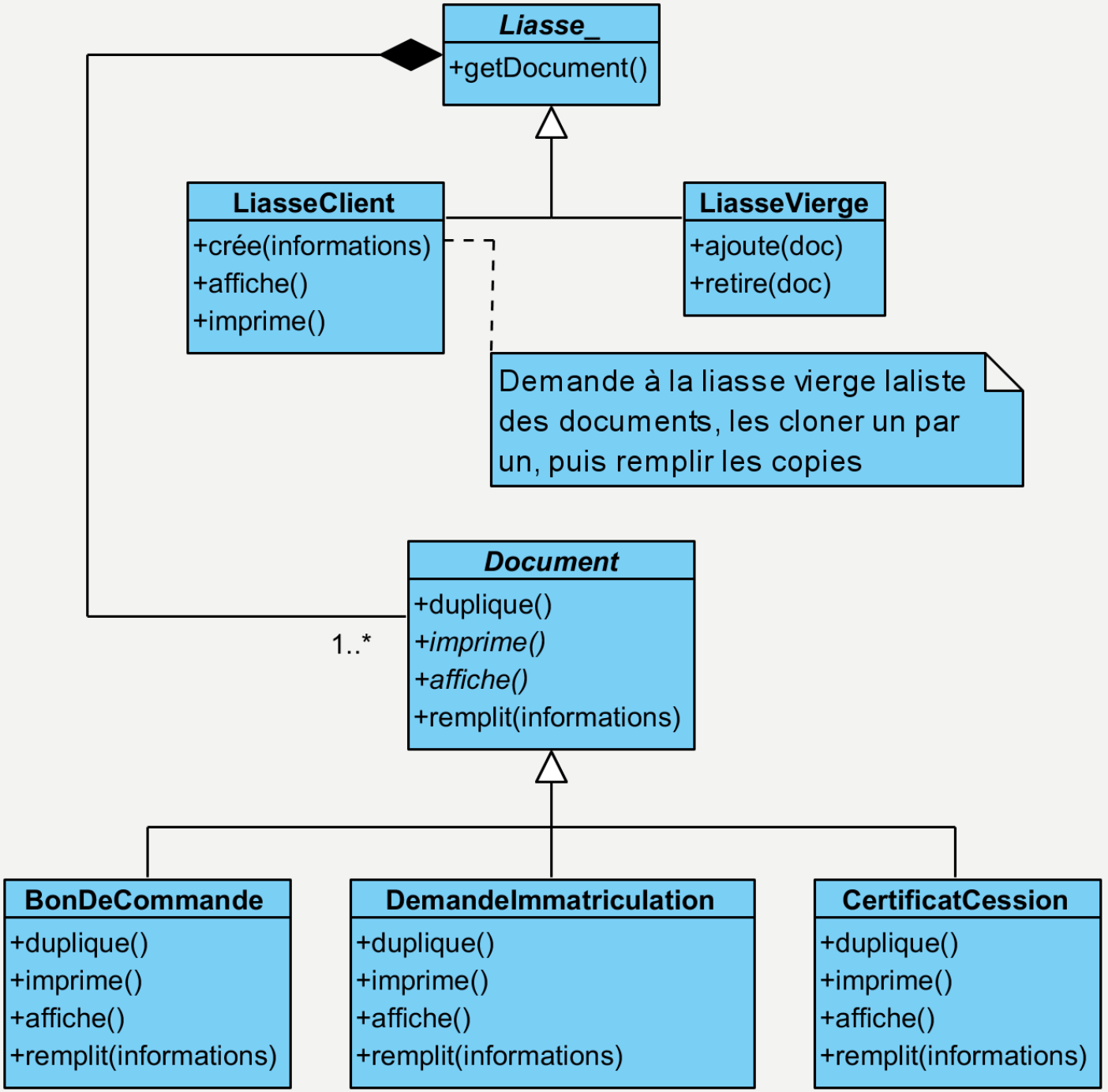
- Lors de l'achat d'un véhicule, un client doit recevoir une liasse définie par un nombre précis de documents tels que le certificat de cession, la demande d'immatriculation ou encore le bon de commande. D'autres types de documents peuvent être ajoutés ou retirés à cette liasse en fonction des besoins 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 classe **Document** est une classe abstraite connue de la classe Liasse. Ses sous-classes correspondent aux différents types de documents. Elles possèdent la méthode **duplique** qui permet de cloner une instance existante pour en obtenir une nouvelle.
- La classe **Liasse** est également abstraite. Elle possède deux sous-classes concrètes :

PROTOTYPE
EXEMPLE

- 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.


Liasse_
+getDocument()
LiasseClient
+crée(informations)
+affiche()
+imprime()
LiasseVierge
+ajoute(doc)
+retire(doc)
Demande à la liasse vierge laliste des documents, les cloner un par un, puis remplir les copies
Document
+duplique()
+imprime()
+affiche()
+remplit(informations)
1..*
BonDeCommande
+duplique()
+imprime()
+affiche()
+remplit(informations)
DemandeImmatriculation
+duplique()
+imprime()
+affiche()
+remplit(informations)
CertificatCession
+duplique()
+imprime()
+affiche()
+remplit(informations)

PROTOTYPE
PARTICIPANTS

- **Client (Liasse, LiasseClient, LiasseVierge)** est une classe composée d'un ensemble d'objets appelés prototypes, instances de la classe abstraite Prototype. La classe Client a besoin de dupliquer ces prototypes sans avoir à connaître ni la structure interne de **Prototype** ni sa hiérarchie de sous-classes.
- **Prototype (Document)** est une classe abstraite d'objets capables de se dupliquer eux mêmes. Elle introduit la signature de la méthode duplique.
- **PrototypeConcret1** et **PrototypeConcret2** (**BonDeCommande, DemandeImmatriculation, CertificatCession**) sont les sous-classes concrètes de Prototype qui définissent complètement un prototype et en implante la méthode duplique.

PROTOTYPE

Collaboration

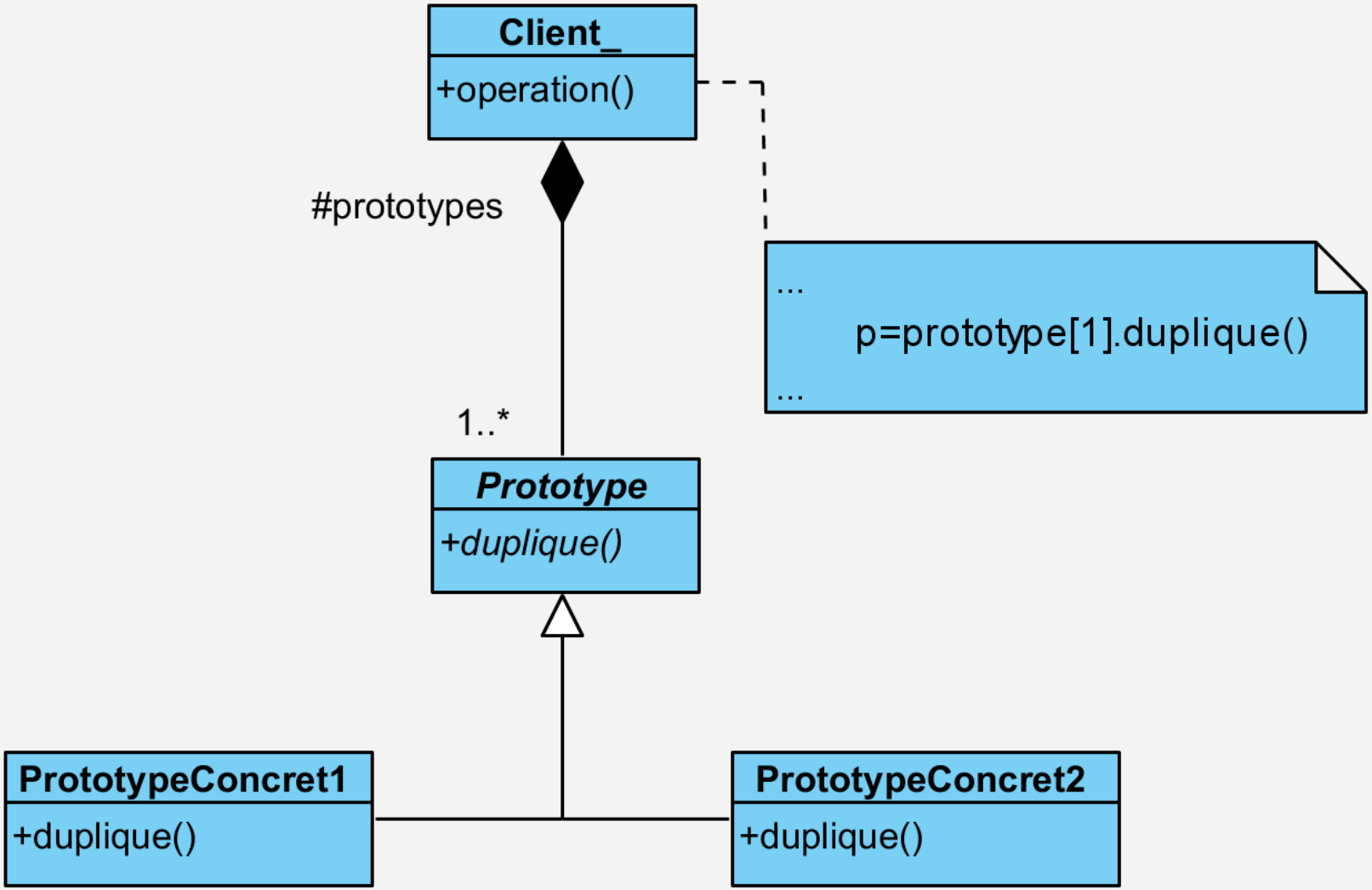
- Le client demande à un ou plusieurs prototypes de se dupliquer eux-mêmes.

Domaines d'utilisation

Le pattern Prototype est utilisé dans les domaines suivants :

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

PATTERN PROTOTYPE
STRUCTURE GÉNÉRIQUE



EXERCICES

- Enumérer trois différences entres classes abstraites et les interfaces Java
- Donner un exemple d'interface avec des méthodes n'impliquant aucune responsabilité pour la classe d'implémentation de retourner une valeur ou d'accomplir une quelconque action pour le compte de l'appelant

PATTERN SINGLETON

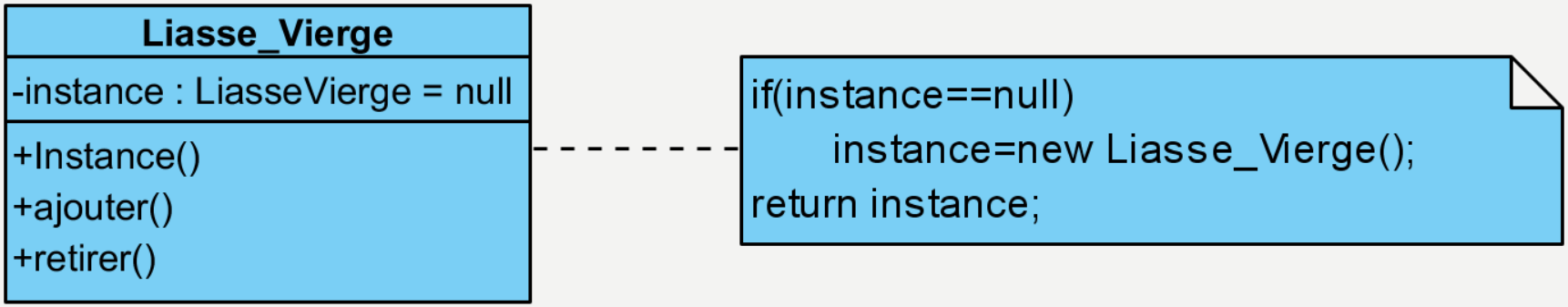
SINGLETON
DESCRIPTION

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

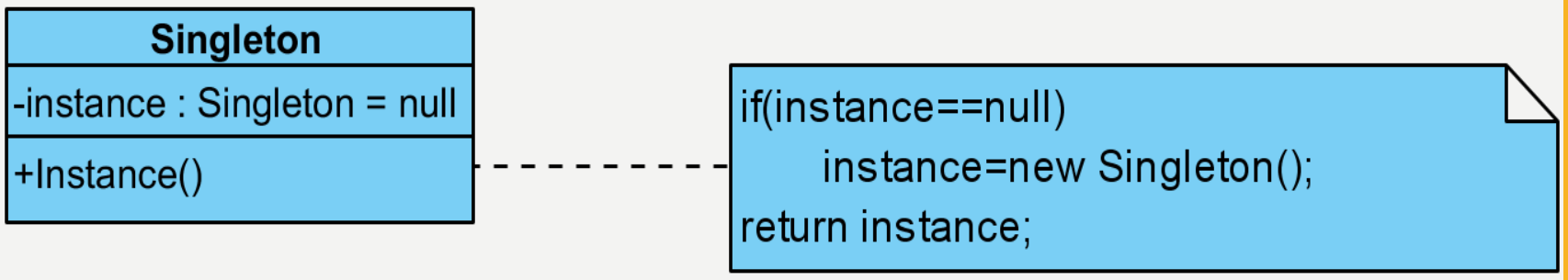
SINGLETON
EXEMPLE

- 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**.
- 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
MODÈLE ADAPTÉ



SINGLETON
MODÈLE GÉNÉRIQUE



SINGLETON

Participant

- Le seul participant est la classe **Singleton** qui offre l'accès à l'unique instance par sa méthode de classe Instance. Par ailleurs, la classe Singleton possède un mécanisme qui assure qu'elle ne peut posséder au plus qu'une seule instance. Ce mécanisme bloque la création d'autres instances.

Collaboration

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

Domaine d'utilisation

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

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

PATTERNS DE STRUCTURATION

PRÉSENTATION

- L'objectif des patterns de structuration est de faciliter l'indépendance de l'interface d'un objet ou d'un ensemble d'objets vis-à-vis de son implantation. Dans le cas d'un ensemble d'objets, il s'agit aussi de rendre cette interface indépendante de la hiérarchie des classes et de la composition des objets.
- En fournissant les interfaces, les patterns de structuration encapsulent la composition des objets, augmentant le niveau d'abstraction du système à l'image des patterns de création qui encapsulent la création des objets. Les patterns de structuration mettent en avant les interfaces. L'encapsulation de la composition est réalisée non pas en structurant l'objet lui-même mais en transférant cette structuration à un second objet. Celui-ci est intimement lié au premier objet. Ce transfert de structuration signifie que le premier objet détient l'interface vis-à-vis des clients et gère la relation avec le second objet qui lui gère la composition et n'a aucune interface avec les clients externes.

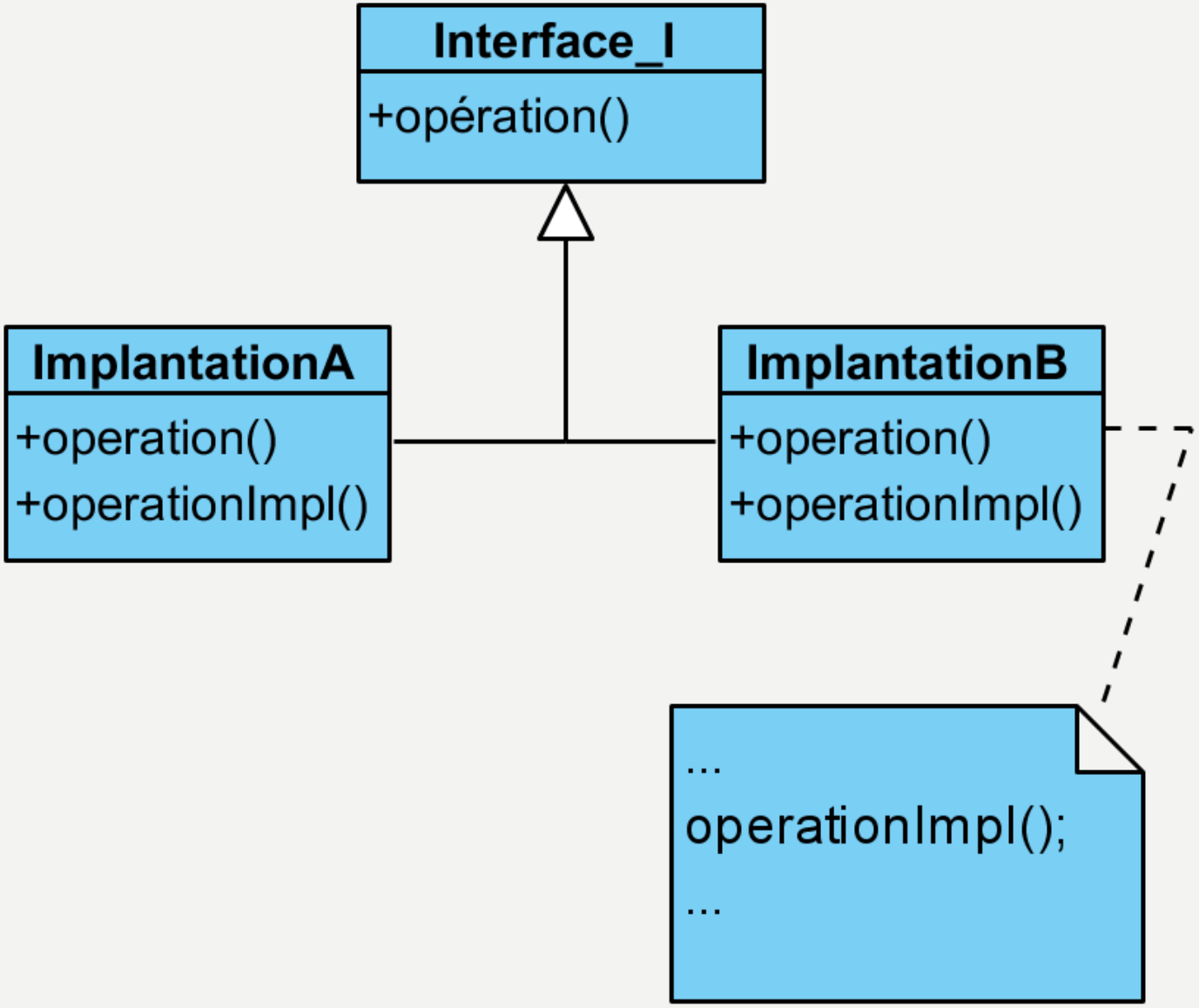
PRÉSENTATION

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

COMPOSITION STATIQUE ET DYNAMIQUE

- Nous prenons l'exemple des aspects d'implantation d'une classe. Nous nous plaçons dans un cadre où il est possible d'avoir plusieurs implantations possibles. La solution classique consiste à les différencier au niveau des sous-classes.
- C'est le cas de l'utilisation de l'héritage d'une interface dans plusieurs classes d'implantation.
- Cette solution consiste à réaliser une composition statique. En effet, une fois que le choix de la classe d'implantation d'un objet est effectué, il n'est plus possible d'en changer.

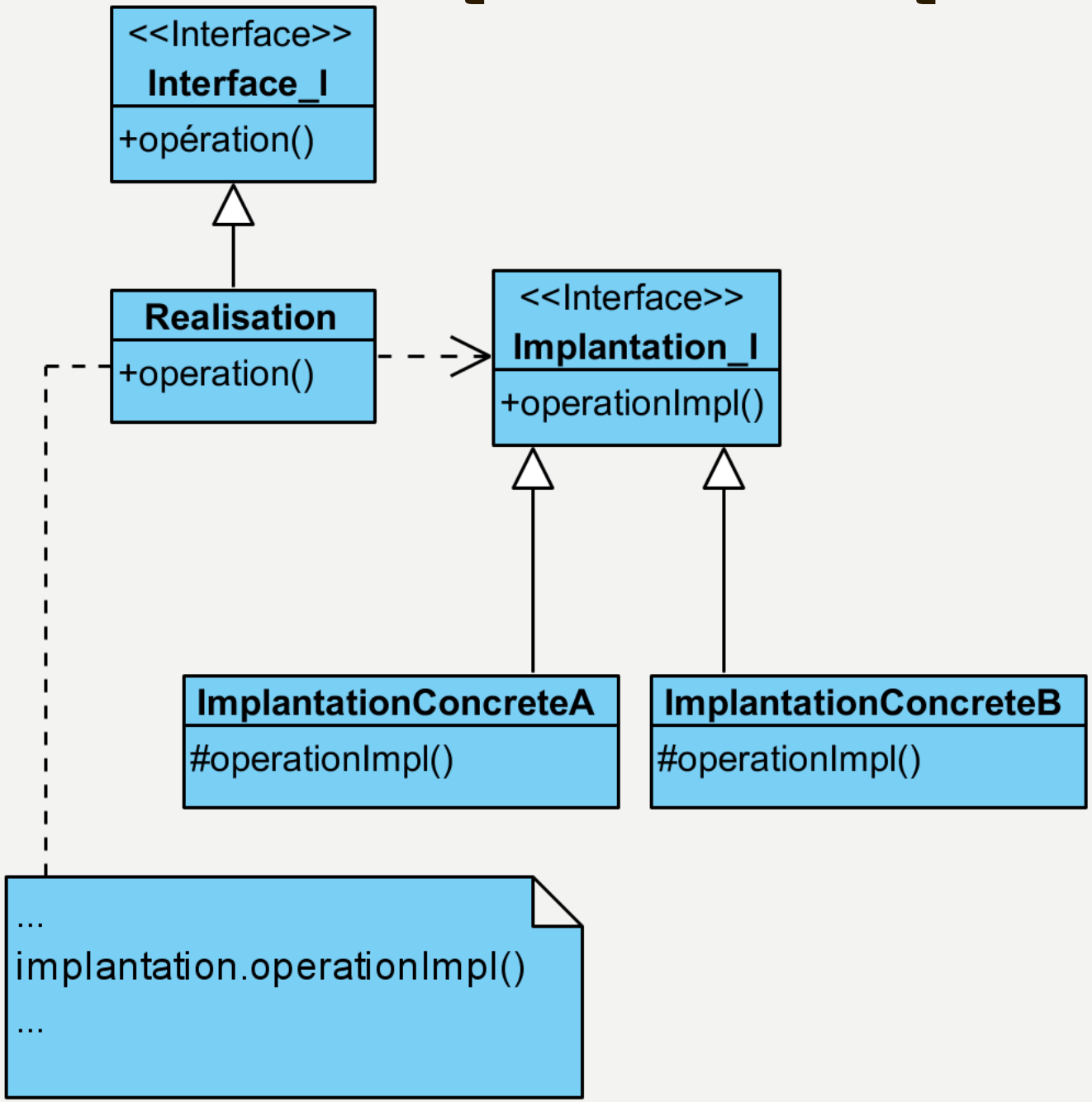
COMPOSITION STATIQUE ET DYNAMIQUE



COMPOSITION STATIQUE ET DYNAMIQUE

- Les parties d'implantation sont gérées par une instance de la classe **ImplantationConcrèteA** ou par une instance de la **classeImplantationConcrèteB**. Cette instance est référencée par l'attribut implantation. Elle peut être substituée facilement par une autre instance lors de l'exécution. Par conséquent, la composition est dynamique.

COMPOSITION STATIQUE ET DYNAMIQUE



PATTERN DE STRUCTURATION

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 ADAPTER

ADAPTER
DESCRIPTION

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

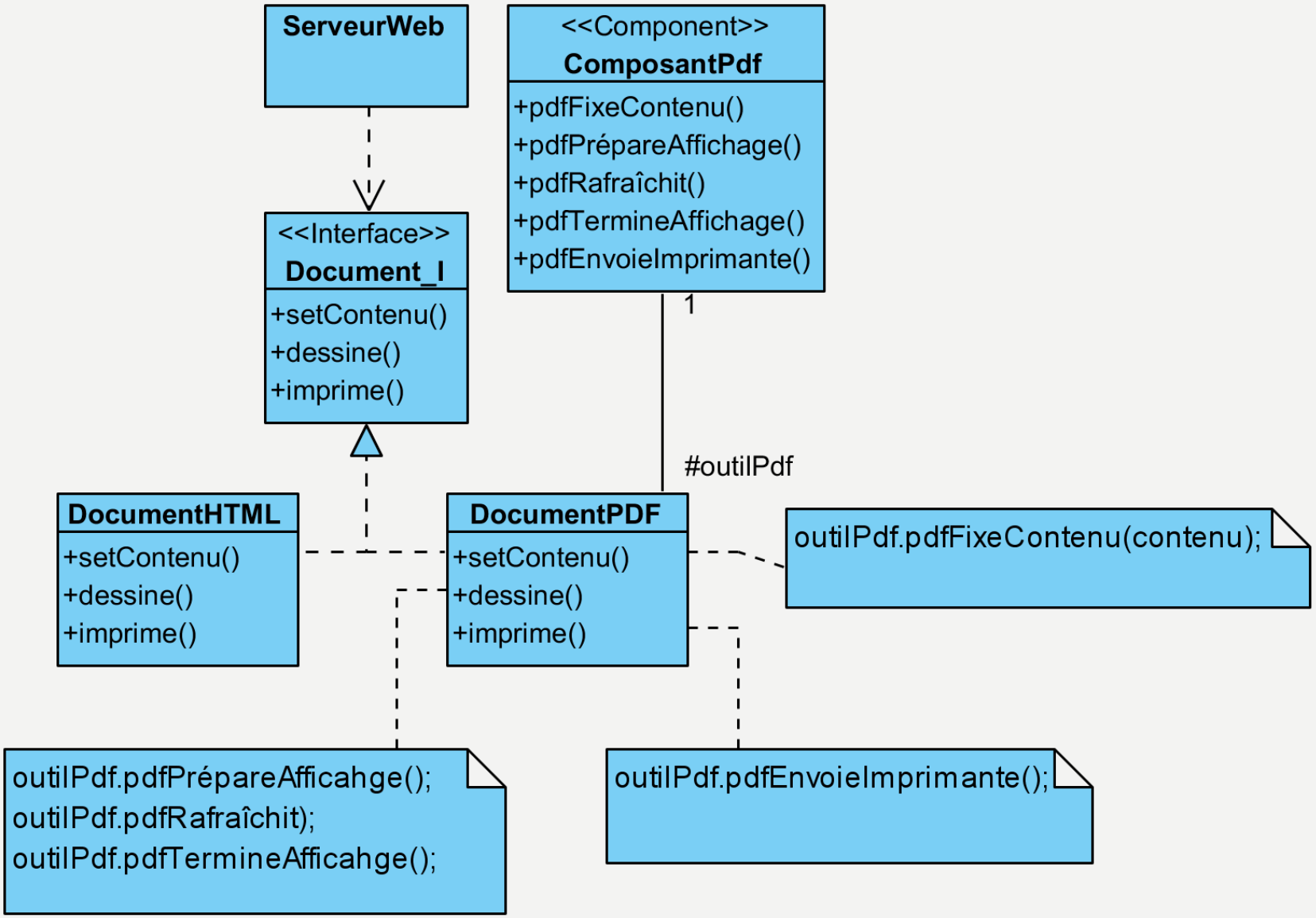
ADAPTER
EXEMPLE

- 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 suivante 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**.

ADAPTER
EXEMPLE

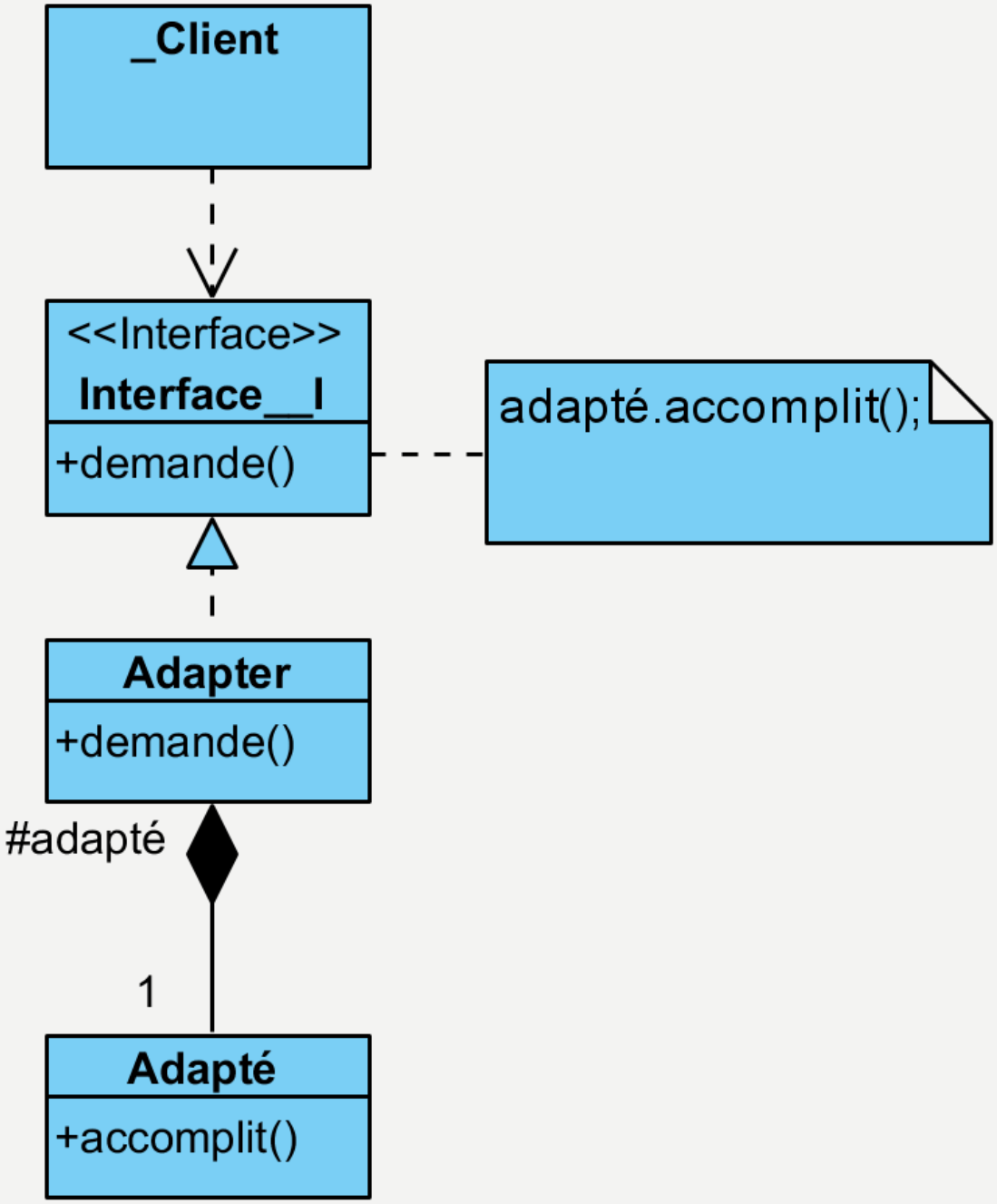
- La figure montre le composant **ComposantPdf** dont l'interface introduit plus de méthodes et dont la convention de nommage est de surcroît différente (préfixe **pdf**).
- Le pattern Adapter propose une solution qui consiste à créer la classe **DocumentPdf** implantant l'interface **Document** et possédant une association avec **ComposantPdf**. L'implantation des trois méthodes de l'interface **Document** consiste à déléguer correctement les appels au composant PDF.

STRUCTURE ADAPTÉE



ADAPTER

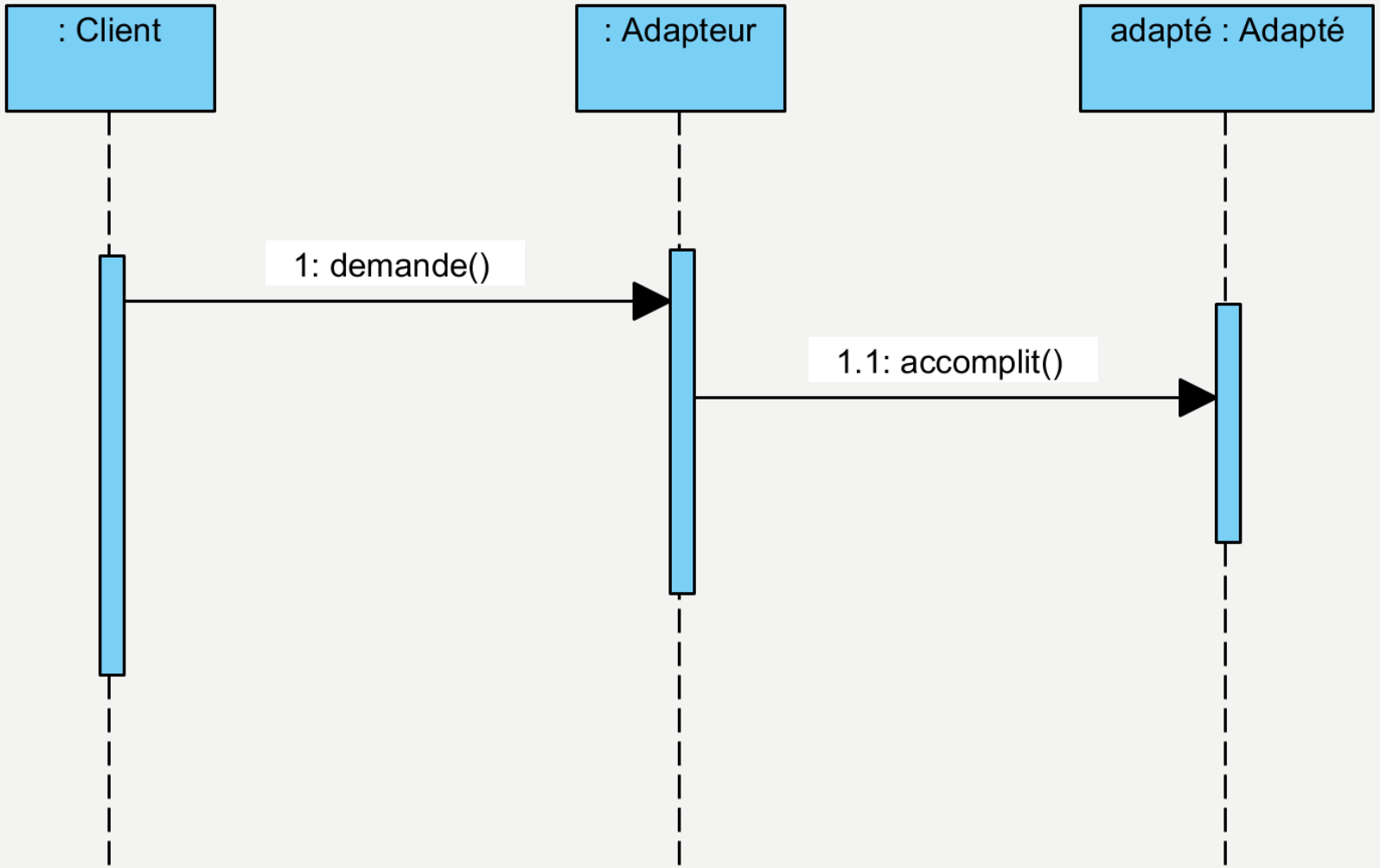
STRUCTURE GÉNÉRIQUE



ADAPTER PARTICIPANTS

- **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 (DIAGRAMME DE SÉQUENCES)



ADAPTER

Domaines d'application

Le pattern est utilisé dans les cas suivants :

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

PATTERN BRIDGE

PATTERN BRIDGE
DESCRIPTION

- Le but du pattern **Bridge** est de séparer l'aspect d'implantation d'un objet de son aspect de représentation et d'interface.
- Ainsi, d'une part l'implantation peut être totalement encapsulée et d'autre part l'implantation et la représentation peuvent évoluer indépendamment et sans que l'une exerce une contrainte sur l'autre.

PATTERN 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 Java.

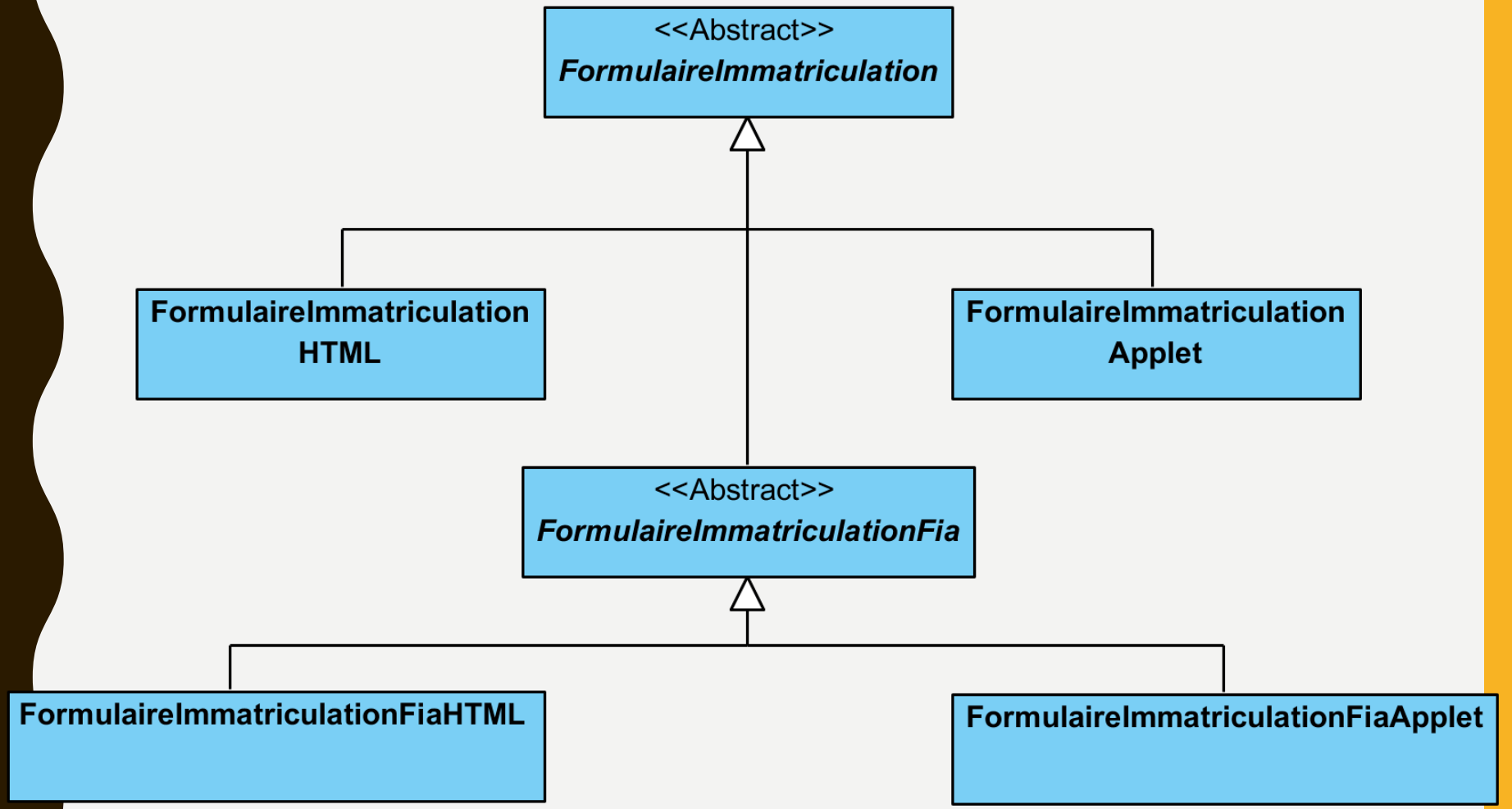
Il est donc possible d'introduire une classe abstraite **FormulaireImmatriculation** et deux sous-classes concrètes **FormulaireImmatriculationHtml** et **FormulaireImmatriculationApplet**.

PATTERN BRIDGE
EXEMPLE

- Dans un premier temps, les demandes d'immatriculation ne concernaient que la ville d'Antananarivo. Par la suite, il est devenu nécessaire d'introduire une nouvelle sous-classe de **FormulaireImmatriculation** correspondant aux demandes d'immatriculation à Fianarantsoa, sous-classe appelée **FormulaireImmatriculationFia**.
- Cette sous-classe doit également être abstraite et avoir également deux sous-classes concrètes pour chaque implantation.

PATTERN BRIDGE

HIÉRARCHIE DES FORMULAIRES INTÉGRANT LES SOUS-CLASSES D'IMPLANTATION



BRIDGE

Ce diagramme met en avant deux problèmes :

- *La hiérarchie mélange au même niveau des sous-classes d'implantation et une sous-classe de représentation :* ***FormulaireImmatriculationFia.*** *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.*

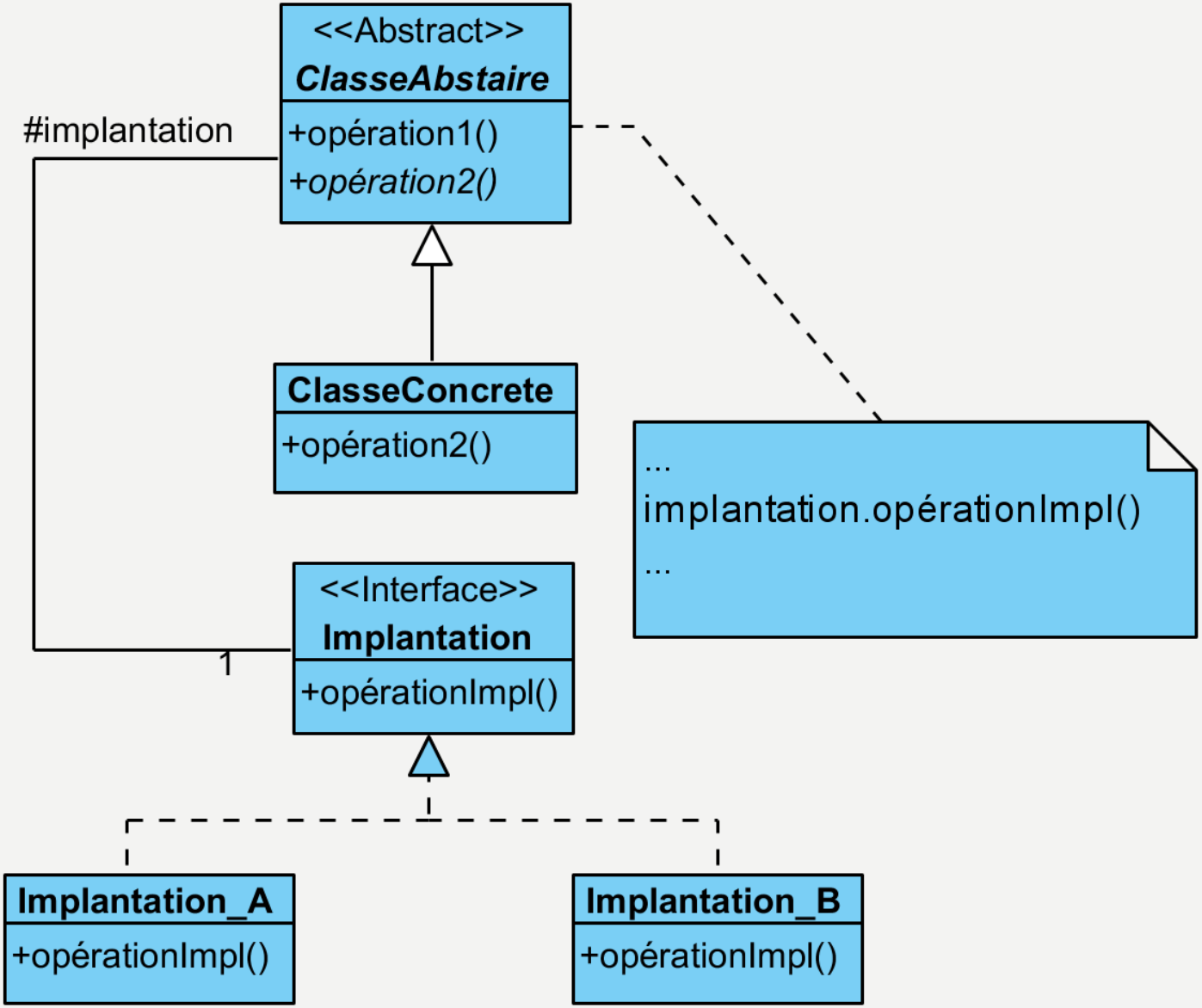
La solution du pattern **Bridge** consiste donc à **séparer les aspects de représentation de ceux d'implantation et à créer deux hiérarchies de classes**

Les instances de la classe **FormulaireImmatriculation** détiennent le lien implantation vers une instance répondant à l'interface **FormulaireImpl**.

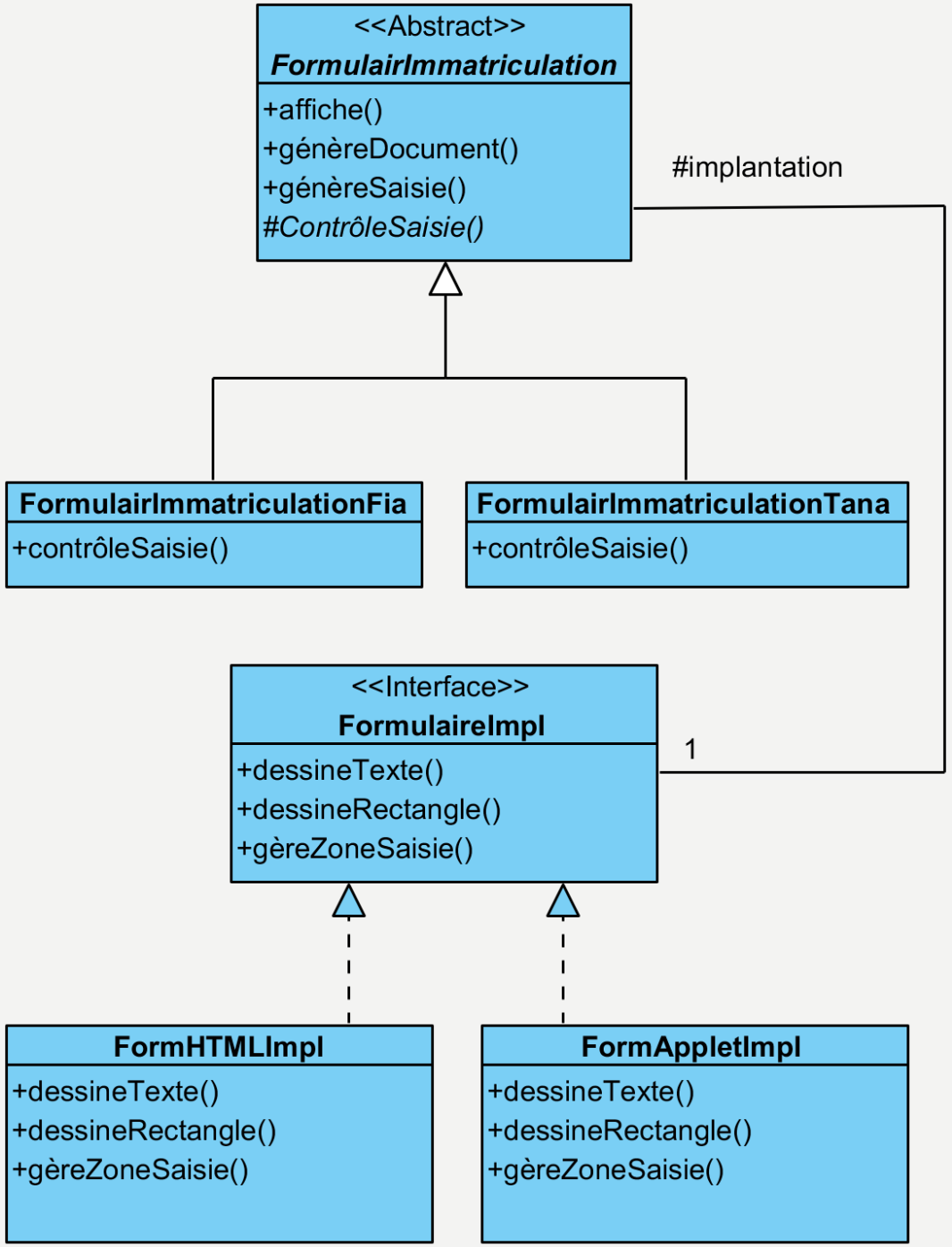
BRIDGE

- 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 ville (**FormImmatriculationFia** et **FormImmatriculationTana**).

BRIDGE
STRUCTURE GÉNÉRIQUE



BRIDGE ADAPTÉ



PATTERN BRIDGE
PARTICIPANTS

- **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 (FormImmatriculationFia et FormImmatriculationTana)** est la classe concrète qui implante les méthodes de **Classe Abstraite**.
- **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.

PATTERN BRIDGE
COLLABORATION

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

PATTERN BRIDGE
DOMAINE 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

PATTERN COMPOSITE

COMPOSITE

Description

- Le but du pattern **Composite** est d'offrir un cadre de conception d'une composition d'objets dont la profondeur est variable, cette conception étant basée sur un arbre.
- Par ailleurs, cette composition est encapsulée vis-à-vis des clients des objets qui peuvent interagir sans devoir connaître la profondeur de la composition.

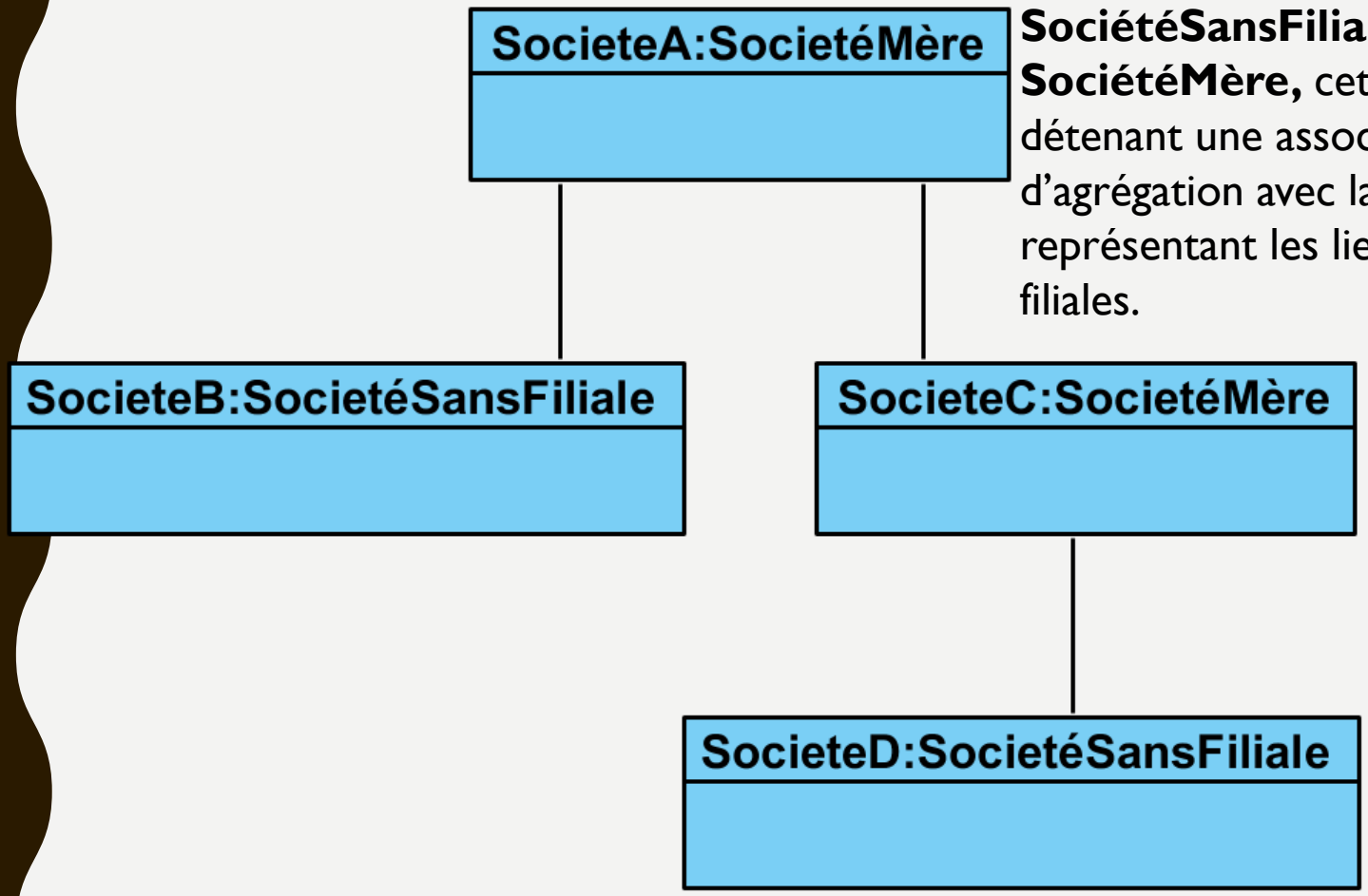
COMPOSITE

Exemple

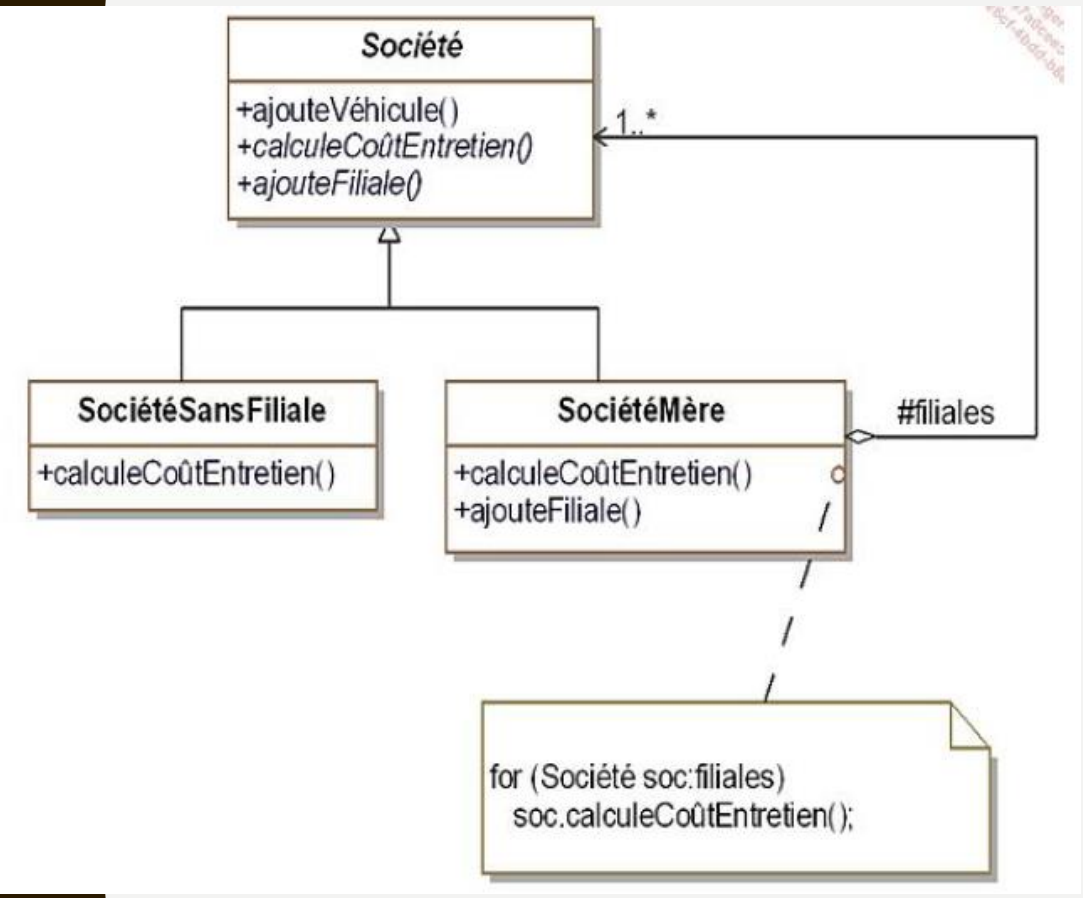
- Au sein du système de vente de véhicules, nous voulons représenter les sociétés clientes, notamment pour connaître le nombre de véhicules dont elles disposent et leur proposer des offres de maintenance de leur parc. Les sociétés qui possèdent des filiales demandent des offres de maintenance qui prennent 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) où les sociétés mères sont placées au-dessus de leurs filiales.

COMPOSITE

La figure 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.

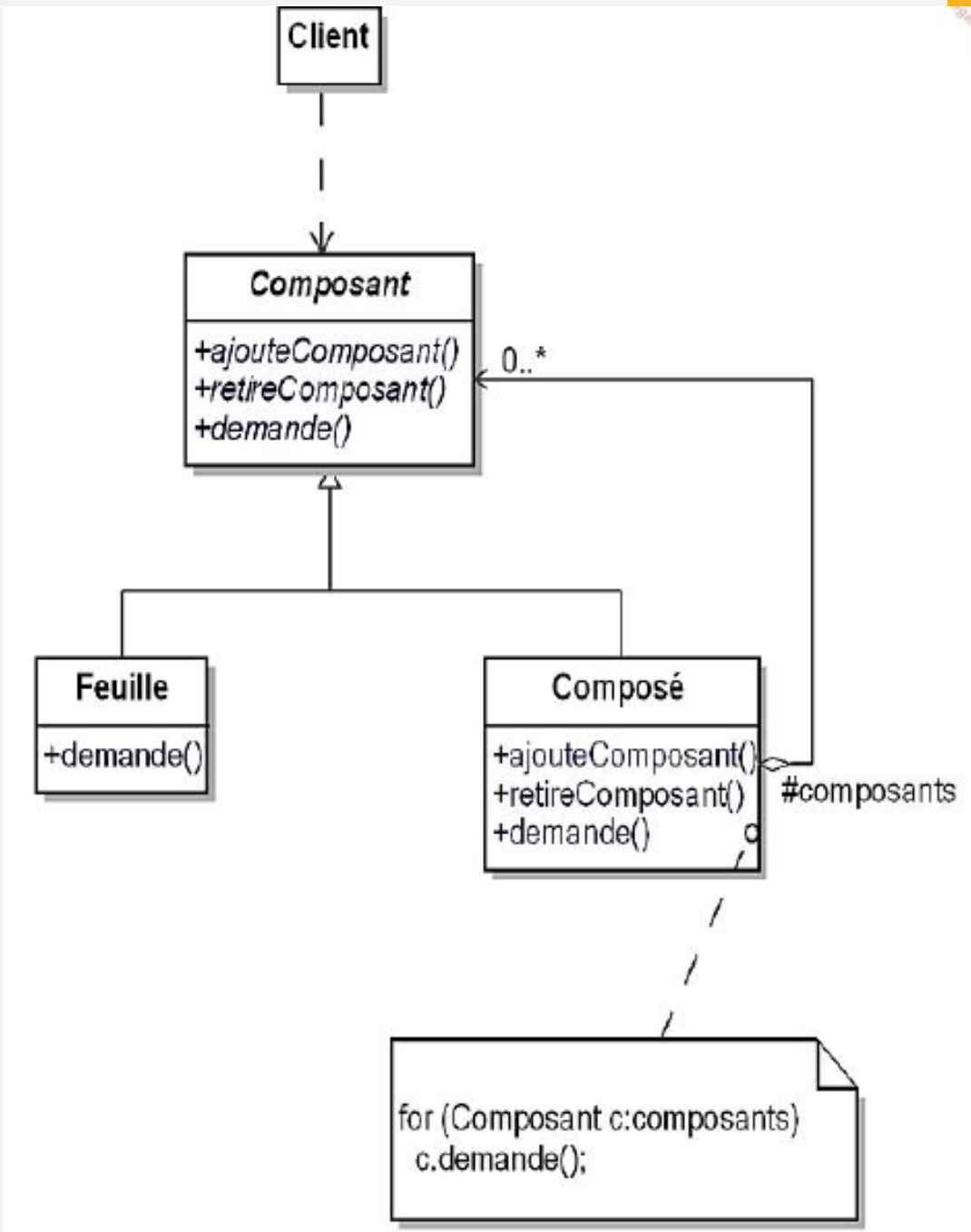


COMPOSITE



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


Client
Composant
+ajouteComposant()
+retireComposant()
+demande()
0..*
Feuille
+demande()
Composé
+ajouteComposant()
+retireComposant()
+demande()
#composants
for (Composant c:composants)
c.demande();

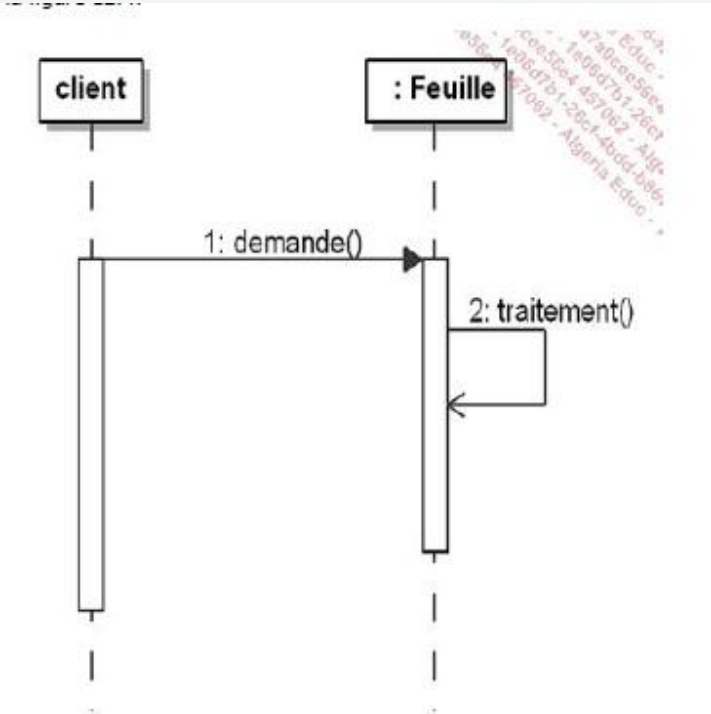
COMPOSITE

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



QUELQUES EXEMPLES DE DESIGN PATTERN

- Créationnel au niveau des classes
  - Délègue une partie du processus de création aux sous-classes
  - Ex.: Factory Method
- Créationnel au niveau des objets
  - Délègue une partie du processus de création à un autre objet
  - Ex.: Abstract Factory, Singleton
- Structurel au niveau des classes et des objets
  - Adaptateurs
- Comportemental au niveau des objets
  - Observer
  - Iterator

PATTERN DECORATOR

DECORATOR

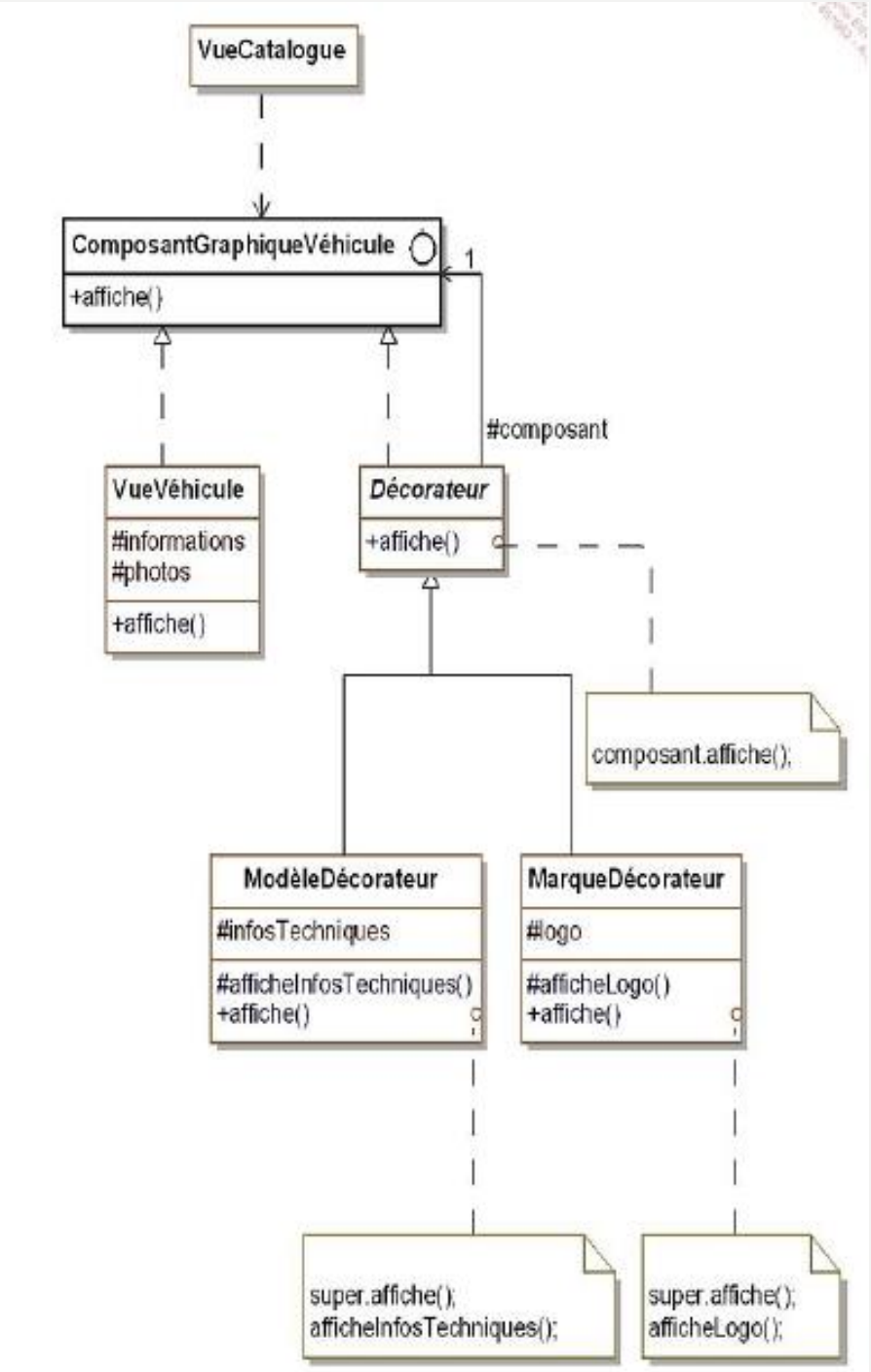
- **Description**
- Le but du pattern Decorator est d'ajouter dynamiquement des fonctionnalités supplémentaires à un objet. Cet ajout de fonctionnalités ne modifie pas l'interface de l'objet et reste donc transparent vis-à-vis des clients.
- Le pattern Decorator constitue une alternative par rapport à la création d'une sous-classe pour enrichir un objet.

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 aisé de comprendre ici que l'utilisation de l'héritage n'est pas adaptée à ce qui est demandé pour deux raisons :
  - l'héritage est un outil trop puissant pour réaliser un tel ajout de fonctionnalité,
  - l'héritage est un mécanisme statique.

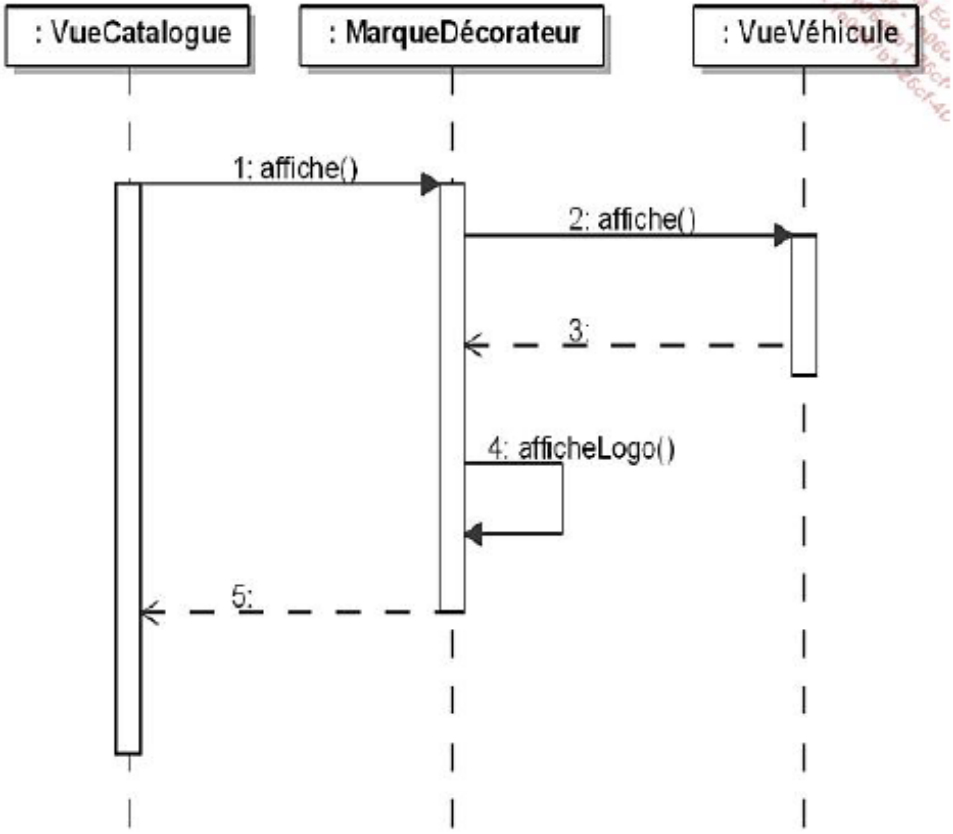
DECORATOR

- **Exemple**
- 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.
- 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.



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

COLLABORATIONS

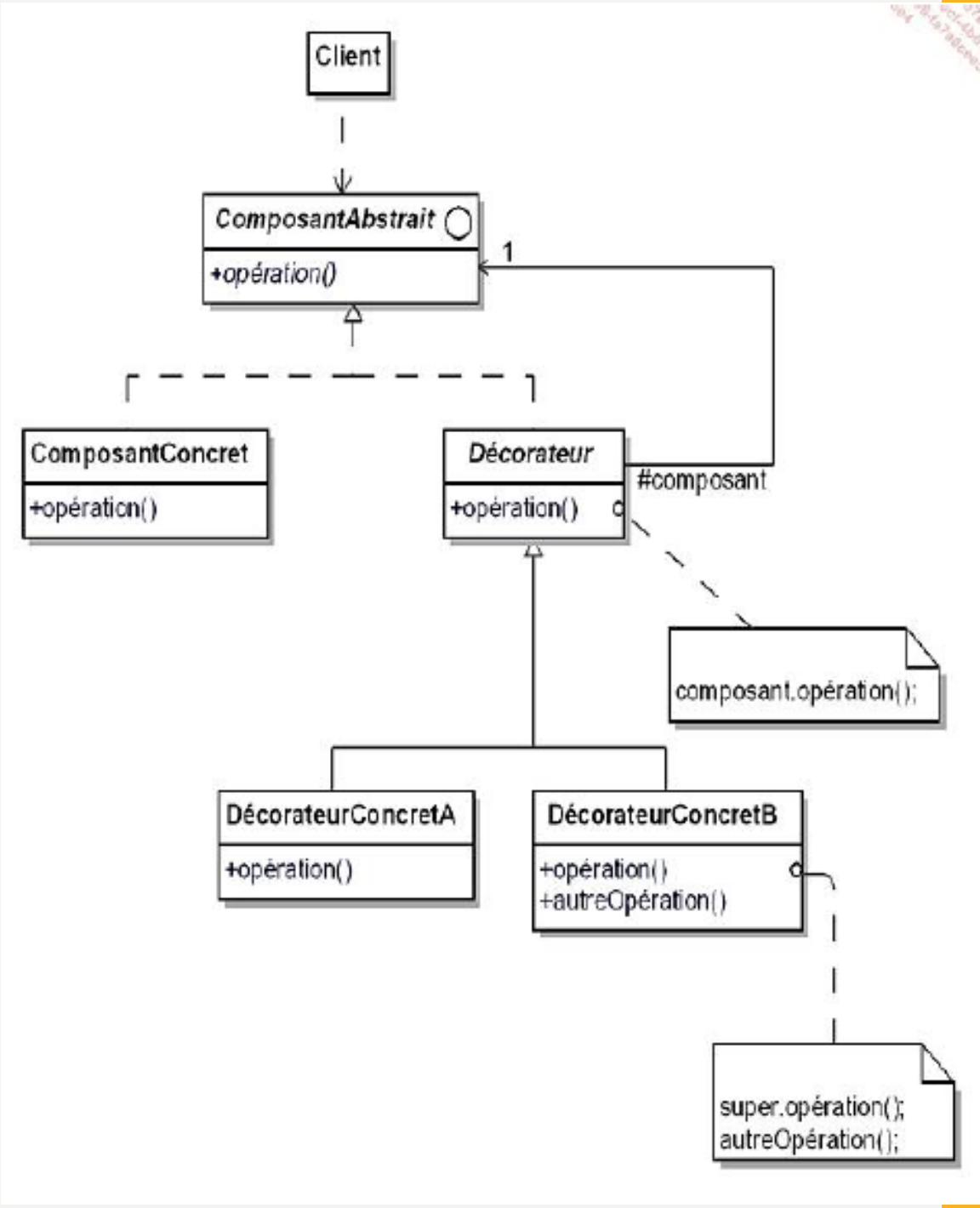


La figure montre la séquence des appels de message destinés à l'affichage d'un véhicule pour lequel les informations techniques du modèle et le logo de la marque sont également affichés.

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

MODÈLE GÉNÉRIQUE



DECORATOR

- 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 ;
- **Décorateurest** une classe abstraite qui détient une référence vers un composant ;
- **DécorateurConcretA** et **DécorateurConcretB** (**ModèleDécorateur** et **MarqueDécorateur**)sont des sousclasses concrètes de Décorateur qui ont pour but l'implantation des fonctionnalités ajoutées au composant.

DOMAINES D'UTILISATION

- **Domaines d'application**
- Le pattern **Decorator** peut être utilisé dans les domaines suivants :
- Un système ajoute dynamiquement des fonctionnalités à un objet, sans modifier son interface, c'est-à-dire sans que les clients de cet objet doivent être modifiés.
- Un système gère des fonctionnalités qui peuvent être retirées dynamiquement.
- L'utilisation de l'héritage pour étendre des objets n'est pas pratique, ce qui peut arriver quand leur hiérarchie est déjà complexe.

PATTERN FACADE

FACADE

- **Description**
- L'objectif du pattern **Facade** est de regrouper les interfaces d'un ensemble d'objets en une interface unifiée rendant cet ensemble plus simple à utiliser pour un client.
- Le pattern **Facade** encapsule l'interface de chaque objet considérée comme interface de bas niveau dans une interface unique de niveau plus élevé. La construction de l'interface unifiée peut nécessiter d'implanter des méthodes destinées à composer les interfaces de bas niveau.

FACADE

- **Exemple**
- Nous voulons offrir la possibilité d'accéder au système de vente de véhicule en tant que service web. Le système est architecturé sous la forme d'un ensemble de composants possédant leur propre interface comme :
- le composant **Catalogue** ;
- le composant **GestionDocument** ;
- le composant **RepriseVéhicule**.

FACADE

- **Exemple**
- 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.

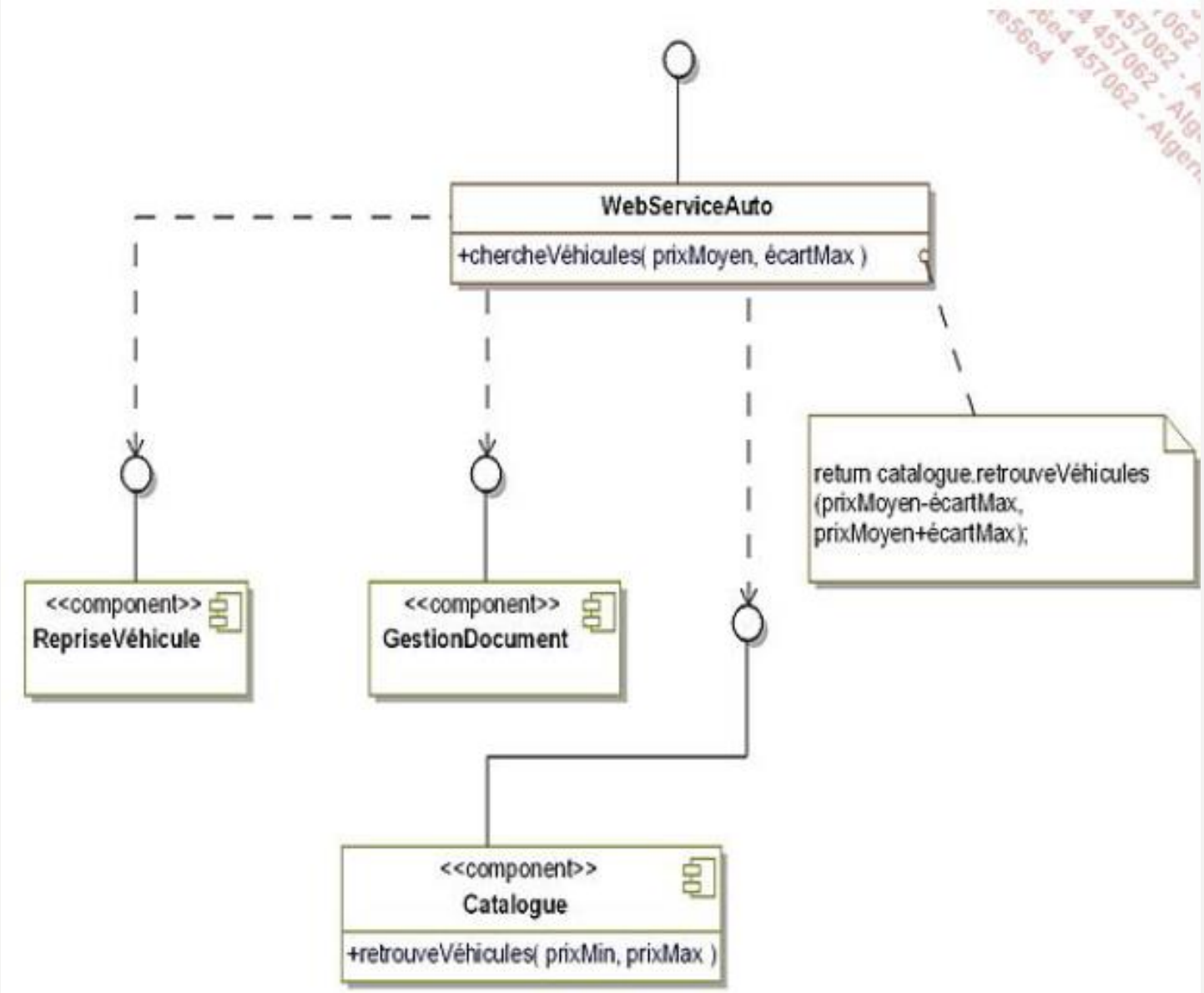
FACADE

- **Exemple**
- 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.
- 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.

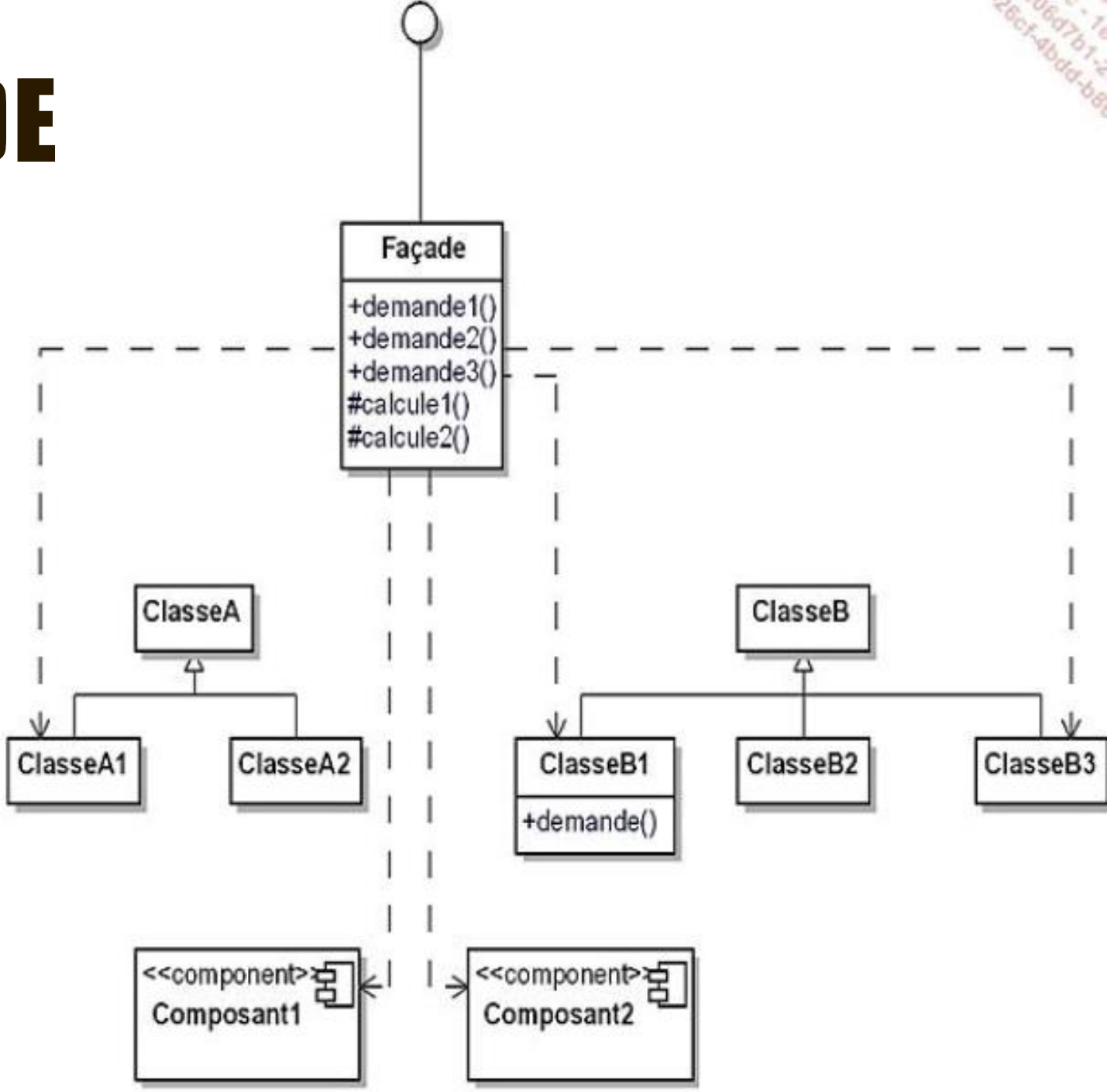
FACADE

- Il convient de noter que si l'idée du pattern est de constituer une interface de plus haut niveau d'abstraction, rien n'empêche de fournir également dans la façade des accès directs aux méthodes des composants du système.

FACADE


WebServiceAuto
+chercheVéhicules(prixMoyen, écartMax)
return catalogue.retrouveVéhicules
(prixMoyen-écartMax,
prixMoyen+écartMax);
<<component>>
RepriseVéhicule
<<component>>
GestionDocument
<<component>>
Catalogue
+retrouveVéhicules(prixMin, prixMax)

FACADE


Façade
+demande1()
+demande2()
+demande3()
#calcule1()
#calcule2()
ClasseA
ClasseB
ClasseA1
ClasseA2
ClasseB1
+demande()
ClasseB2
ClasseB3
<<component>>
Composant1
<<component>>
Composant2

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.

PATTERN FLYWEIGHT

FLYWEIGHT

Description

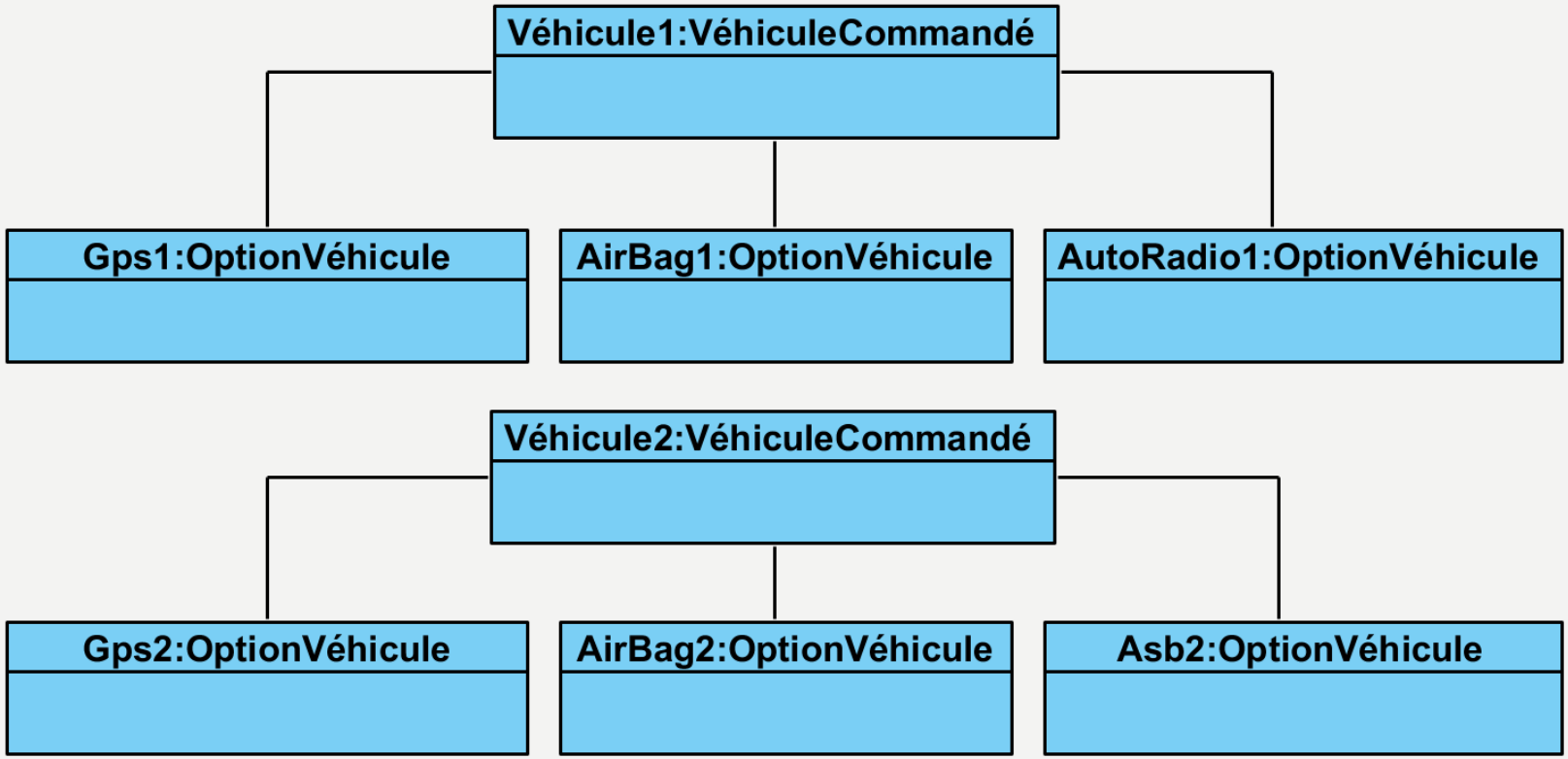
- Le but du pattern **Flyweight** est de partager de façon efficace un ensemble important d'objets de grain fin.

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 suivante et il est aisé de se rendre compte qu'un grand nombre d'instances de **OptionVéhicule** doit être géré alors que nombre d'entre elles contiennent des données identiques.

FLYWEIGHT

Exemple d'absence de partage d'objets de grain fin



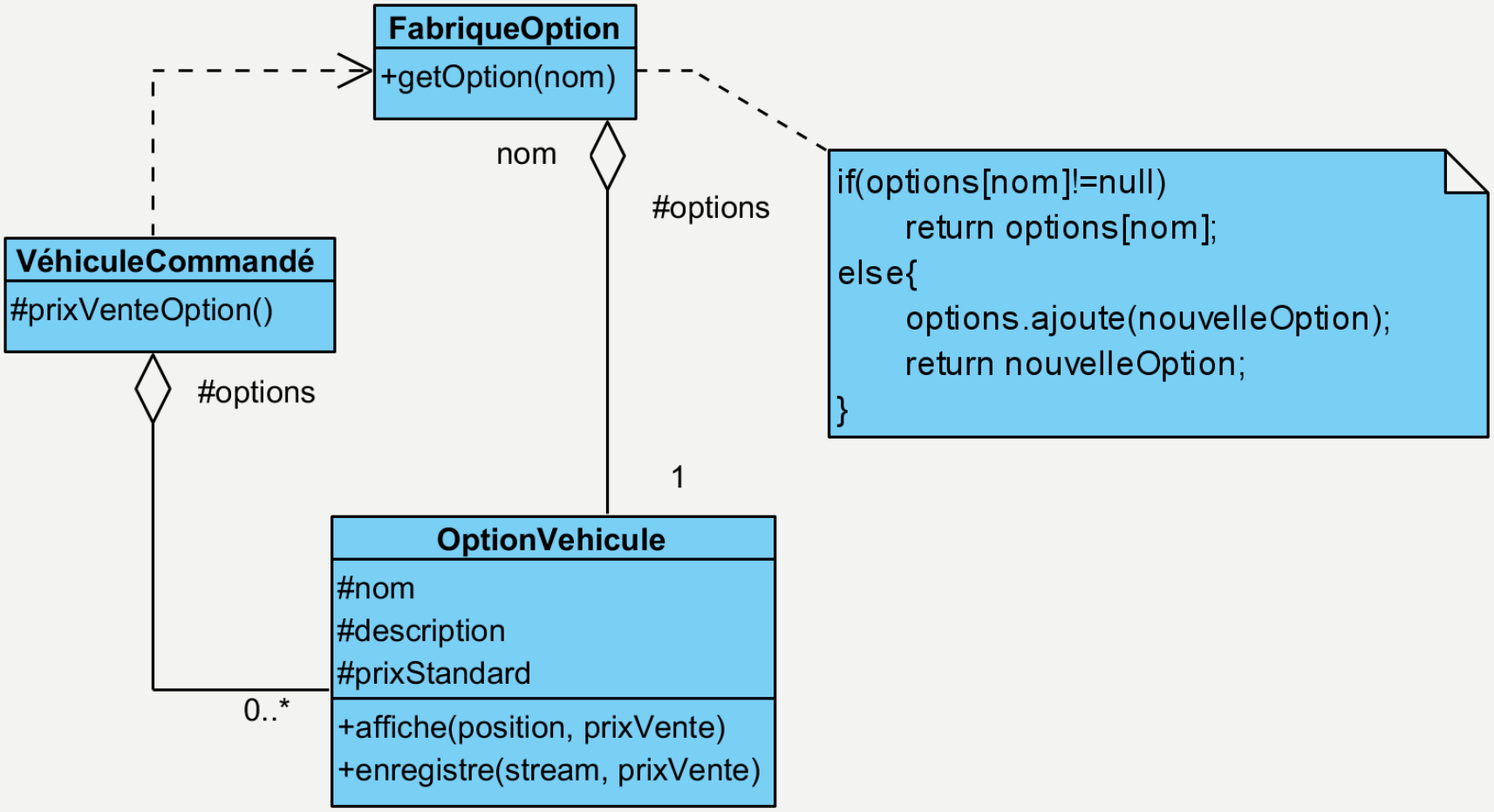
FLYWEIGHT

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

- le partage est réalisé par une fabrique à laquelle le système s'adresse pour obtenir une référence vers une option. Si cette option n'a pas été créée jusqu'à présent, la fabrique procède à sa création avant d'en renvoyer la référence ;
- les attributs d'une option ne contiennent que ses informations spécifiques indépendamment des véhicules commandés : ces informations constituent l'**état intrinsèque** des options ;
- les informations particulières à une option et à un véhicule sont stockées au niveau du véhicule : ces informations constituent l'**état extrinsèque** des options. Elles sont transmises comme paramètres lors des appels des méthodes des options.

Dans le cadre de ce pattern, les options sont les objets appelés **flyweights** (poids mouche en français)

FLYWEIGHT
ADAPTÉ À DES OPTIONS DE VÉHICULES

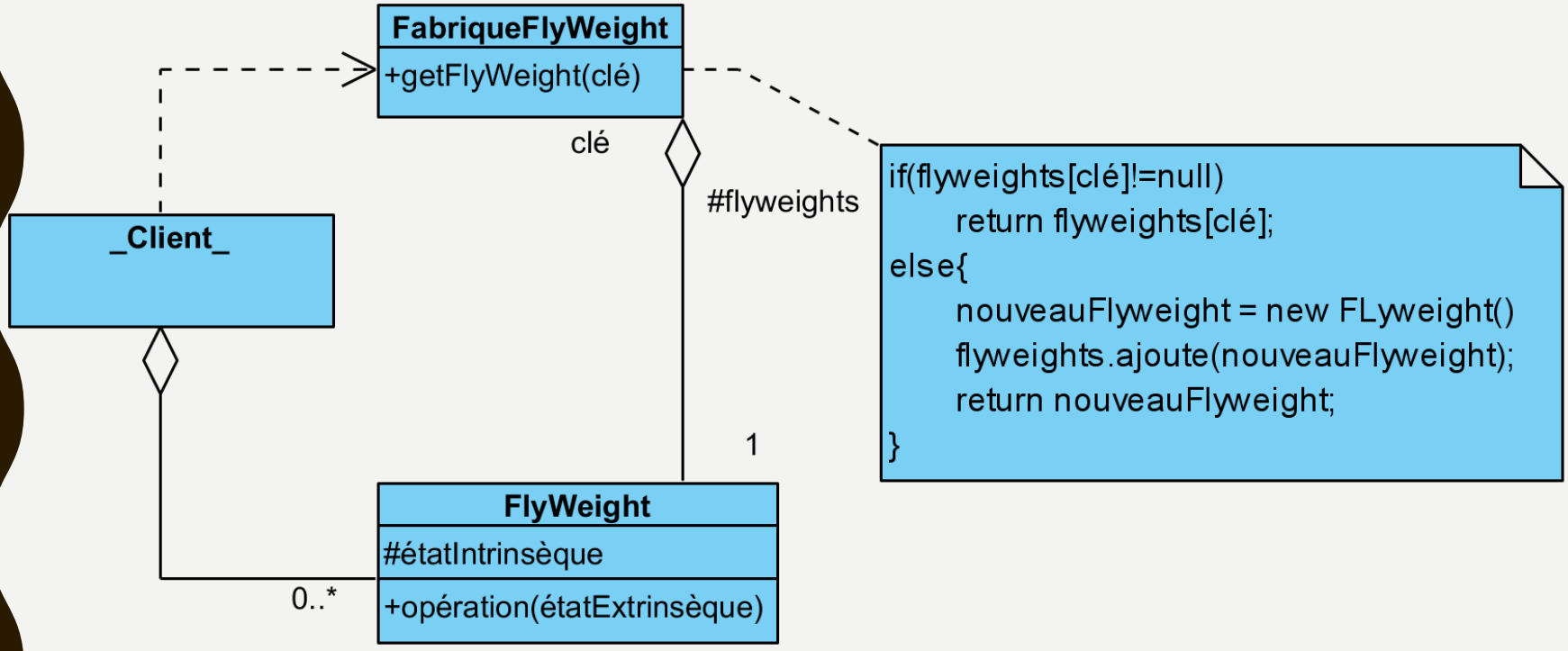


FLYWEIGHT
ADAPTÉ À DES OPTIONS DE VÉHICULES

Ce diagramme introduit les classes suivantes :

- **OptionVéhicule** dont les attributs détiennent l'état intrinsèque d'une option. Les méthodes affiche et enregistre ont pour paramètre **prixDeVente** qui représente l'état extrinsèque d'une option ;
- **FabriqueOption** dont la méthode **getOption** renvoie une option à partir de son nom. Son fonctionnement consiste à rechercher l'option dans l'association qualifiée et à la créer dans le cas contraire ;
- **VéhiculeCommandé** qui possède une liste des options commandées ainsi que leur prix de vente.

FLYWEIGHT
STRUCTURE GÉNÉRIQUE



FLYWEIGHT

Participants

Les participants au pattern sont les suivants :

- **FabriqueFlyweight** (**FabriqueOption**)crée et gère les **flyweights**. La fabrique s'assure que les **flyweights** sont partagés grâce à la méthode **getFlyweight** qui renvoie les références vers les **flyweights** ;
- **Flyweight** (**OptionVéhicule**)détient l'état intrinsèque et implante les méthodes. Ces méthodes reçoivent et déterminent également l'état extrinsèque des **flyweights** ;
- **Client** (**VéhiculeCommandé**)contient un ensemble de références vers les **flyweights** qu'il utilise. Le client doit également détenir l'état extrinsèque de ces **flyweights**.

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.

Domaine d'application

Le domaine d'application du pattern Flyweight est le partage de petits objets (poids mouche). Les critères d'utilisation sont les suivants :

- le système utilise un grand nombre d'objets ;
- le stockage des objets est coûteux à cause d'une grande quantité d'objets ;
- il existe de nombreux ensembles d'objets qui peuvent être remplacés par quelques objets partagés une fois qu'une partie de leur état est rendue extrinsèque.

EXERCICES

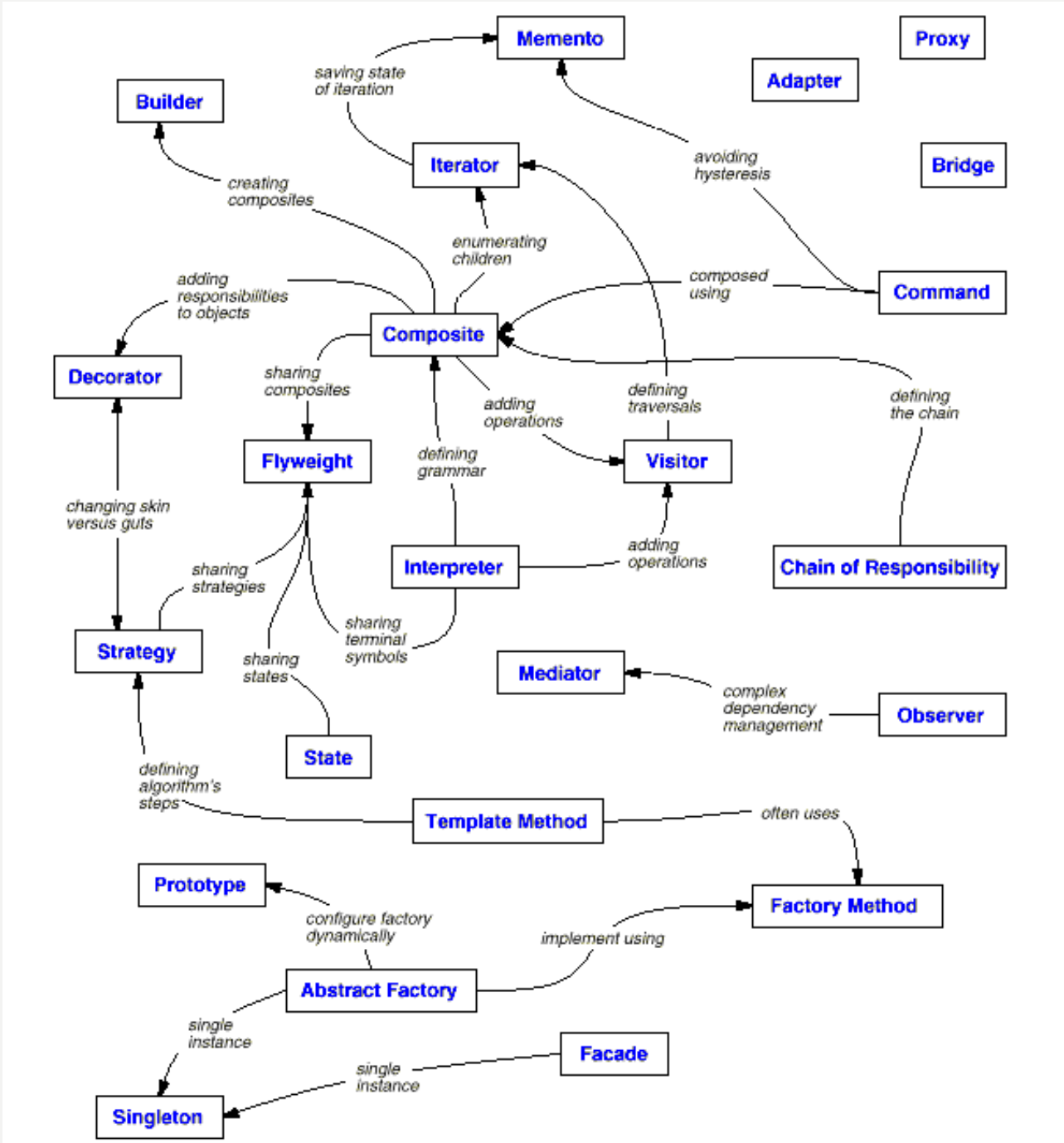
- Modifier le programme en ne permettant de créer qu'un seul objet à partir de FabriqueOption (Considérer le pattern singleton)

PATTERN PROXY

PROXY

Description

- Le pattern **Proxy** a pour objectif la conception d'un objet qui se substitue à un autre objet (le **sujet**) et qui en contrôle l'accès.
- L'objet qui effectue la substitution possède la même interface que le sujet, ce qui rend cette substitution transparente vis-à-vis des clients.

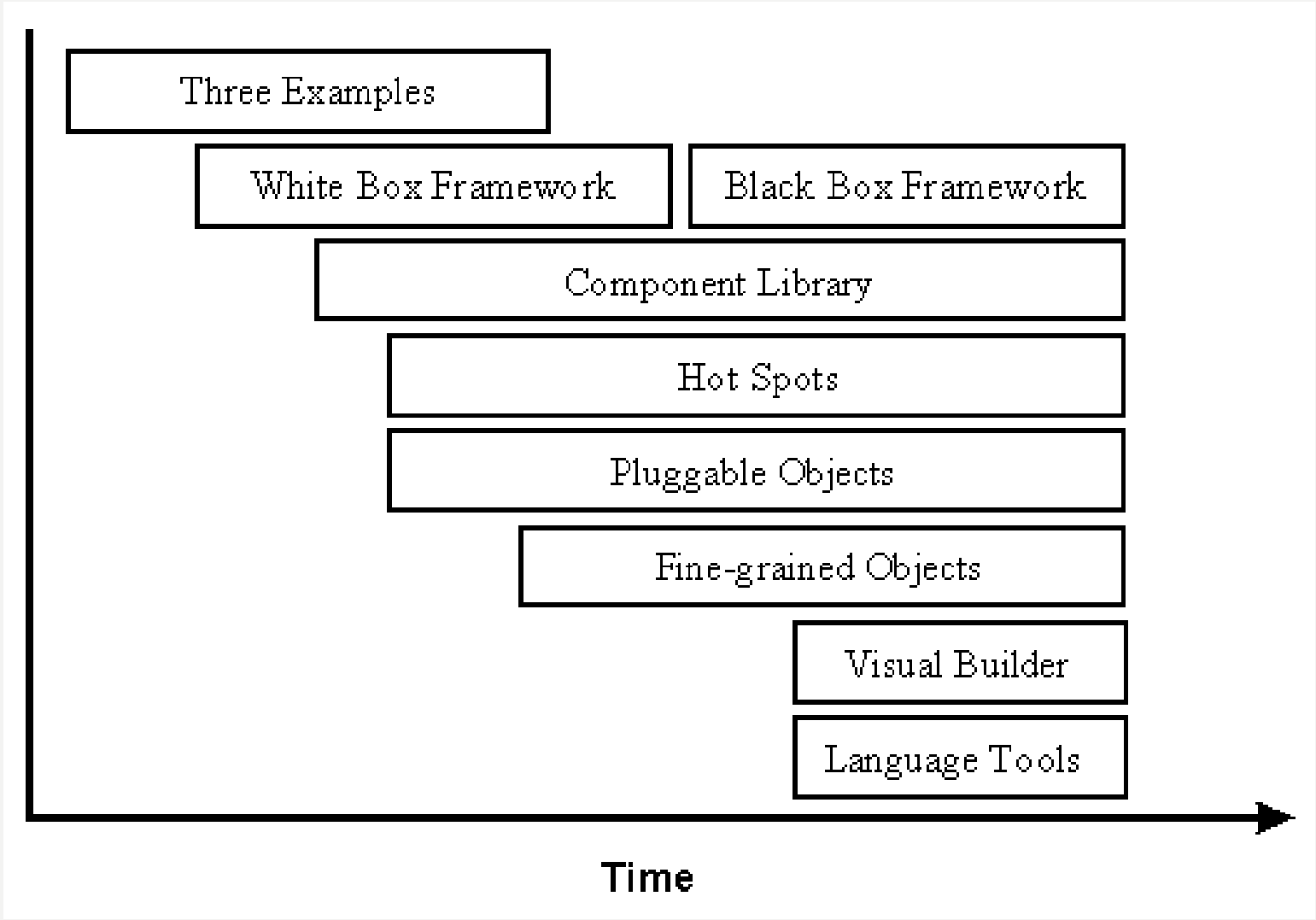


Design pattern relationships

CONCEPTS AVANCÉS EN DESIGN PATTERNS

- Langage de design patterns
- Design patterns et niveaux architecturaux

DESIGN PATTERNS LANGUAGES EXEMPLE: EVOLVING F



NIVEAUX D'ARCHITECTURE I

- Niveau PROGRAMME
  - DESIGN PATTERNS
    - Créationnels
    - Structurels
    - Comportementaux
- Niveau APPLICATION
  - DESIGN PATTERNS.
    - Maximiser le parallélisme.
    - Améliorer l'implémentation des objets.
    - Modifier les « stubs » client et autres trucs.

NIVEAUX D'ARCHITECTURE II

- Niveau SYSTEME
  - DESIGN PATTERNS.
    - Principes des architectures à objets.
    - Structurels.
    - Utilisation des services CORBA.
- Niveau ENTREPRISE
  - DESIGN PATTERNS.
    - Construire l 'infrastructure organisationnelle.
- Niveau GLOBAL
  - DESIGN PATTERNS.
    - Rôle des systèmes ouverts.
    - Internet.

RÉFÉRENCES

- [Gamma, et al., 1984] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns*, Reading, MA, Addison-Wesley, 1984.

- [Mowbray and Malveau, 1997] Thomas J. Mowbray and Raphael C. Malveau, *CORBA Design Patterns*, New York, Wiley Computer Publishing, 1997, 334 p.

- [Roberts and Johnson] Don Roberts, Ralph Johnson, *Evolving Frameworks A Pattern Language for Developing Object-Oriented Frameworks*, http://st-www.cs.uiuc.edu/users/droberts/evolve.html.