Institut Universitaire de Technologie
Aix*Marseille Université

Module M6501

Techniques de programmation avancée

Chapitre1: Design Patterns

Hana ALOUAOUI

Hana.ALOUAOUI@univ-amu.fr

Introduction

En architecture: En étudiant les bâtiments, les villes, les rues et pratiquement chaque espace construit par les êtres humains, on peut déterminer que les bonnes constructions qui résolvent le même problème ont des **points communs** (qualifiés de patterns).

Définition : Pattern

Un patron décrit à la fois un **problème** qui se produit très fréquemment dans l'environnement et l'architecture de la **solution** à ce problème de telle façon que l'on puisse **utiliser** cette solution des milliers de fois sans jamais l'**adapter** deux fois de la même manière.

C. Alexander

Définition : Pattern

Un pattern doit contenir:

- Un nom de pattern
- Son but, le problème qu'il résout
- La solution
- Les limites à prendre en compte pour le mettre en place

Design patterns informatiques

- Les problèmes **récurrents** rencontrés lors du développement des logiciels peuvent-ils être résolus d'une manière similaire?
- Est il possible de concevoir un logiciel à l'aide de patterns qui permettront de créer des solutions par la suite?

Design Patterns (patterns de conception)

- Un design pattern ou pattern de conception consiste en un schéma à objets qui forme une solution à un problème connu et fréquent.

- Le schéma à objets est constitué par un ensemble **d'objets** décrits par des **classes** et des **relations** liant les objets.

- Ce sont des solutions connues et éprouvées dont la conception provient de **l'expérience de programmeurs**.

Intérêt des design patterns

Réutiliser les solutions

→ En utilisant des solutions prédéfinies, vous bénificiez de l'experience d'autres développeurs.

→Vous n'avez pas à réinventer des solutions à des problèmes **récurrents**.

Intérêt des design patterns

Améliorer la communication et l'apprentissage

Les patterns impliquent l'adoption d'une terminologie et d'une **approche commune** au sein d'une équipe de développeurs.

Intérêt des design patterns

Souplesse du logiciel

- Faciliter la modification ultérieure du logiciel

→ des solutions plus facilement modifiables que celles auxquelles vous pouvez penser en début de projet.

Organisation du catalogue des patterns de conception

- **Les patterns de construction** ont pour but d'organiser **la création d'objets**. Ils sont au nombre de cinq : Abstract Factory, Builder, Factory Method, Prototype et Singleton.
- **Les patterns de structuration** facilitent l'organisation de **la hiérarchie** des classes et de leurs relations. Ils sont au nombre de sept : Adapter, Bridge, Composite, Decorator, Facade, Flyweight et Proxy.
- **Les patterns de comportement** proposent des solutions pour **organiser les interactions** et pour répartir les traitements entre les objets. Ils sont au nombre de onze : Chain of responsibility, Command, Interpreter, Iterator, Mediator, Memento, Observer, State, Strategy, Template Method et Visitor.

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

- **Regarder les descriptions et déterminer s'il existe un ou plusieurs patterns dont la description s'approche de celle du problème (voir annexe)**
- Etudier au travers de la structure générique si le pattern répond de façon pertinente au problème.
- Chercher si **le pattern après adaptation répond au problème**.
- Renommer les classes et les méthodes introduites dans la structure générique.

Description des patterns de conception

Pour chaque pattern, les éléments suivants sont présentés :

- Le nom du pattern.
- La description du pattern.
- Un exemple décrivant le problème
- La solution basée sur le pattern (diagramme de classes UML)

→ Au sein de ce schéma, le corps des méthodes est décrit à l'aide de notes.

Patterns de construction

Présentation

- Abstraire les mécanismes de création d'objets.
- système **indépendant de la façon dont les objets sont créés**
- **favorisent l'utilisation des interfaces** dans les relations entre objets

Les problèmes liés à la création d'objets

Problématique

- Dans la plupart des langages à objets, la création d'objets se fait grâce au mécanisme **d'instanciation**
- En Java, une instruction de création d'un objet s'écrit ainsi :

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

Les problèmes liés à la création d'objets

Dans certains cas, il est nécessaire de paramétrer la création d'objets.

Exemple: méthode construitDoc qui crée des documents. Elle peut construire des documents PDF, RTF ou HTML. Généralement le type du document à créer est transmis en paramètre à la méthode sous forme d'une chaîne de caractères, ce qui donne le code suivant :

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

Dans notre exemple, il faut changer le code de la méthode construitDoc en cas d'ajout de nouvelles classes de document.

Les solutions proposées par les patterns de construction

- 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 cet objet.

Les solutions proposées par les patterns de construction

- 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.

Etude de cas : vente de véhicules en ligne

Etude de cas : vente de véhicules en ligne

Description du système

- 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**.

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 une commande** d’un véhicule

Cahier des charges

- Le système doit **gérer les commandes**.
- Il doit être capable de calculer les taxes en fonction du pays de livraison du véhicule.
- 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.

Prise en compte des patterns de conception

Description de la partie	Pattern de conception
Construire les objets du domaine (automobile à essence, automobile à électricité, etc.).	Abstract Factory
Représenter un Vendeur	Singleton
Représenter les sociétés clientes.	Composite
Calculer le montant d'une commande.	Template Method

Patterns de construction

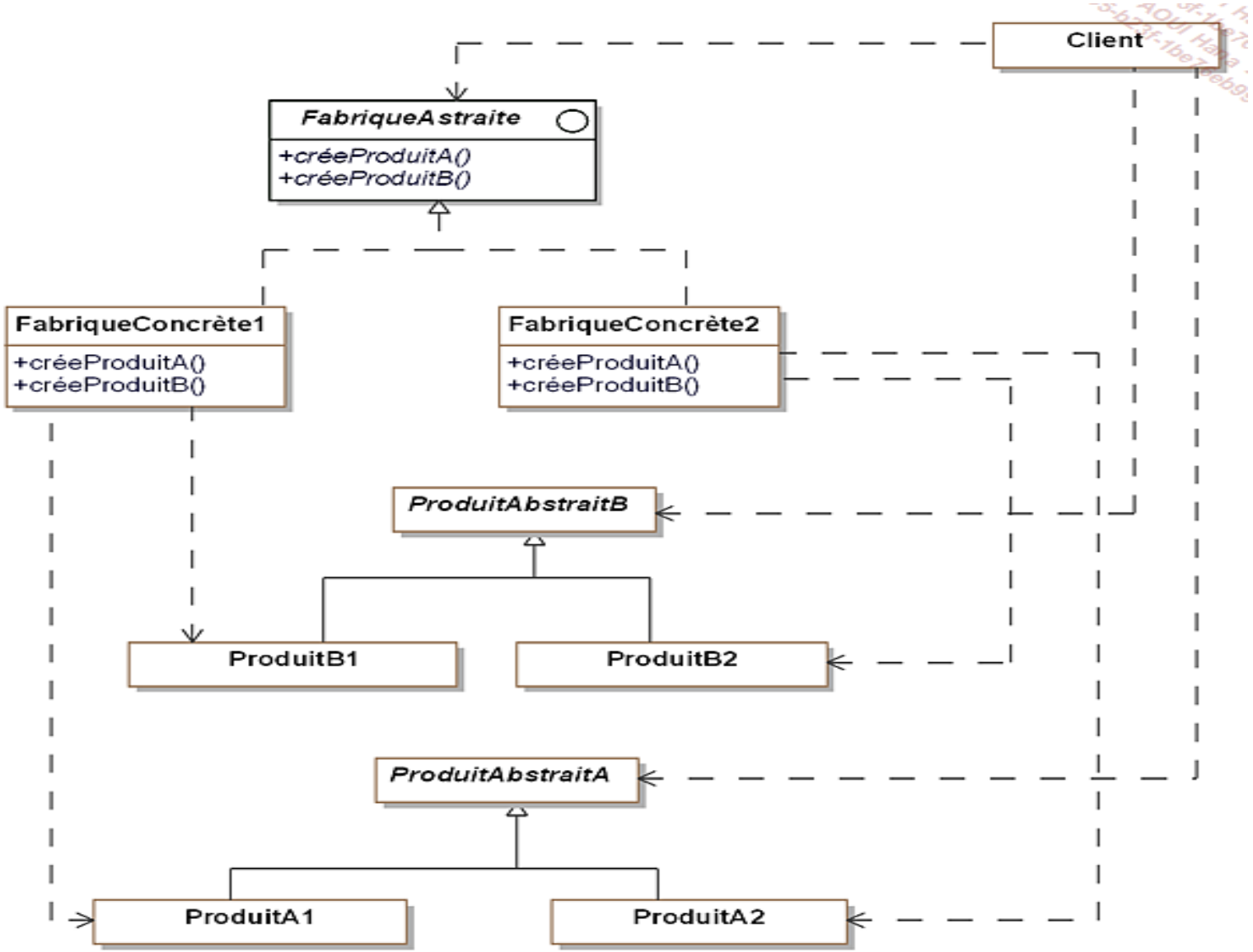
Pattern Abstract Factory

- 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.

Exemple

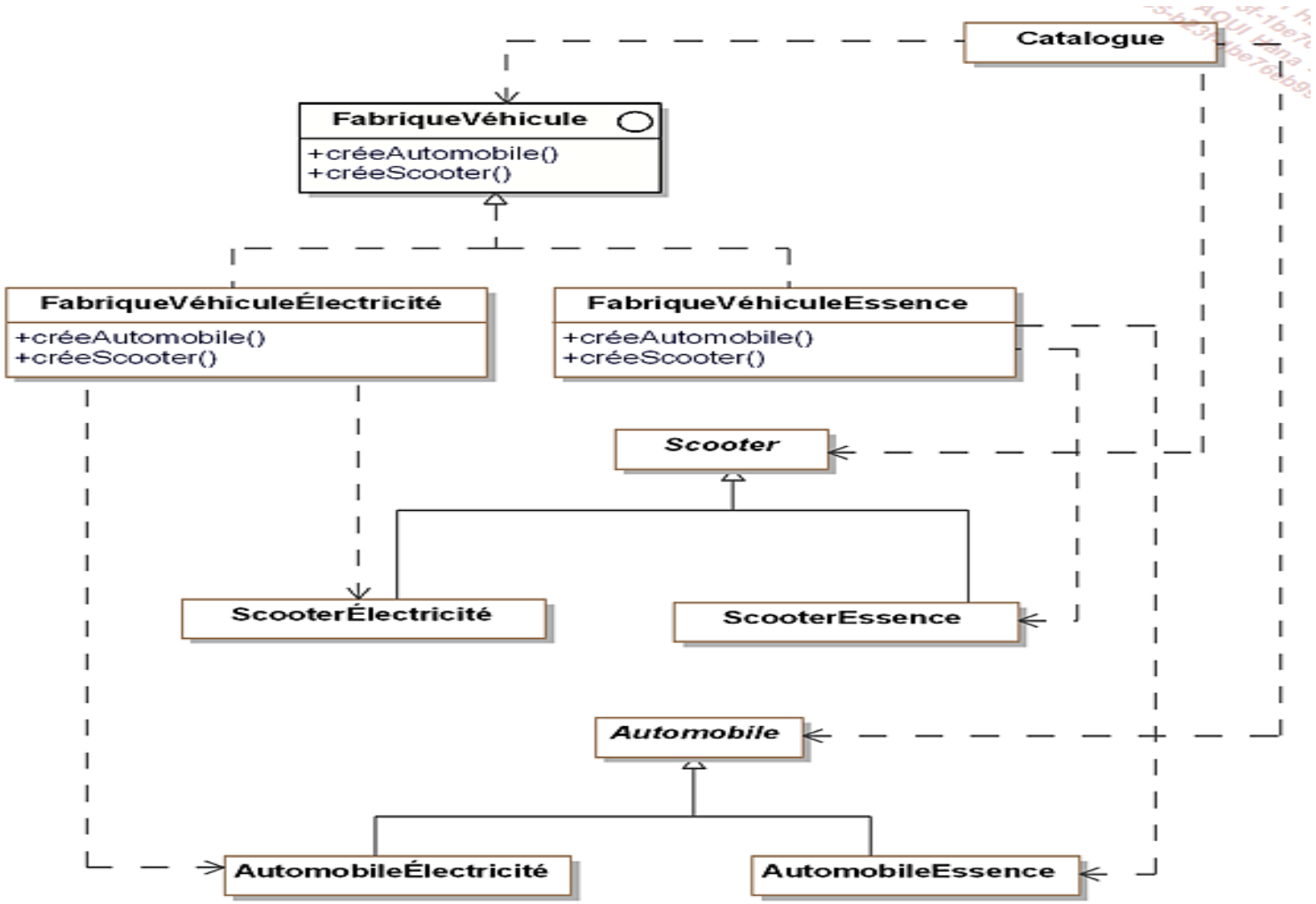
- Le système de vente de véhicules gère des véhicules fonctionnant à **l'essence** et des véhicules fonctionnant à **l'électricité**. Cette gestion est confiée à **l'objet Catalogue** qui crée de tels objets.

Pattern Abstract Factory



Structure générique du pattern `Abstract Factory`

Pattern Abstract Factory



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

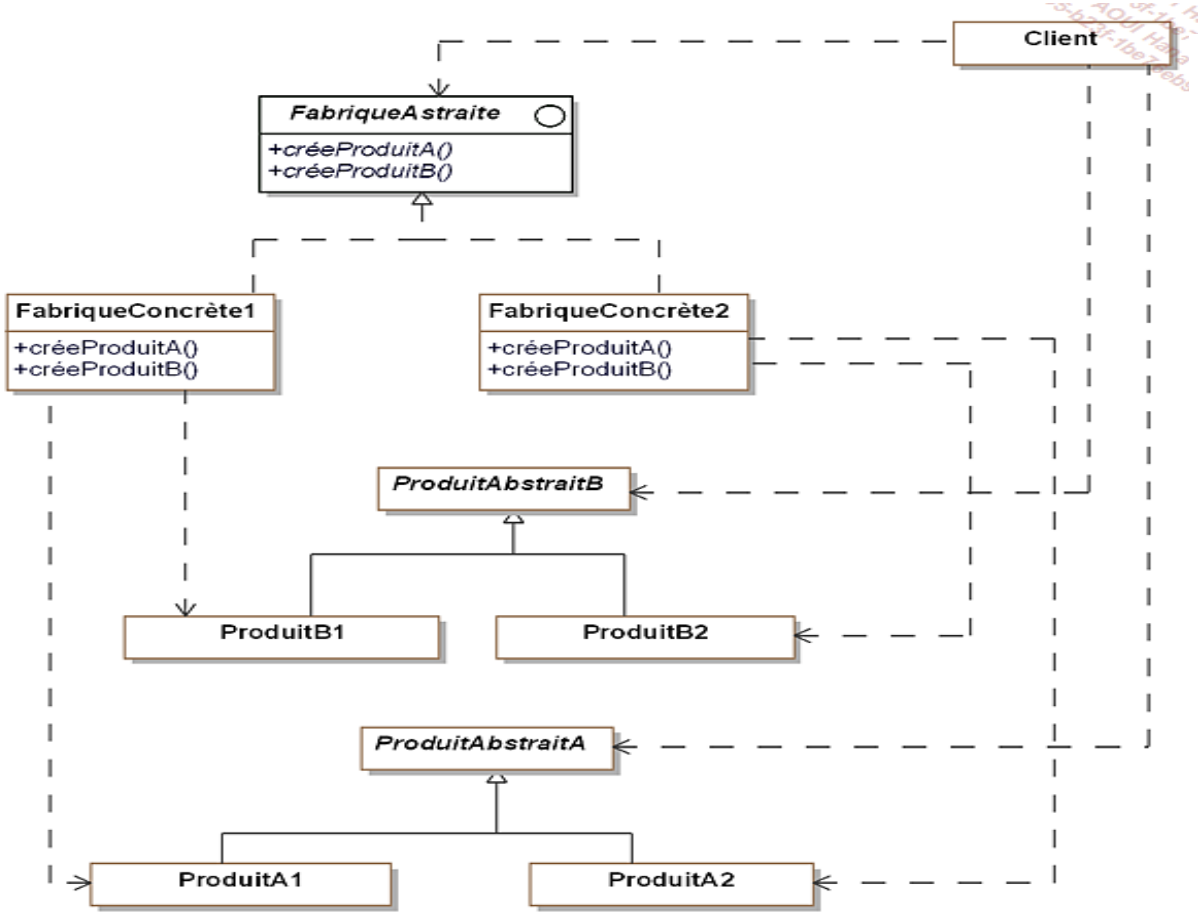
Pattern Abstract Factory

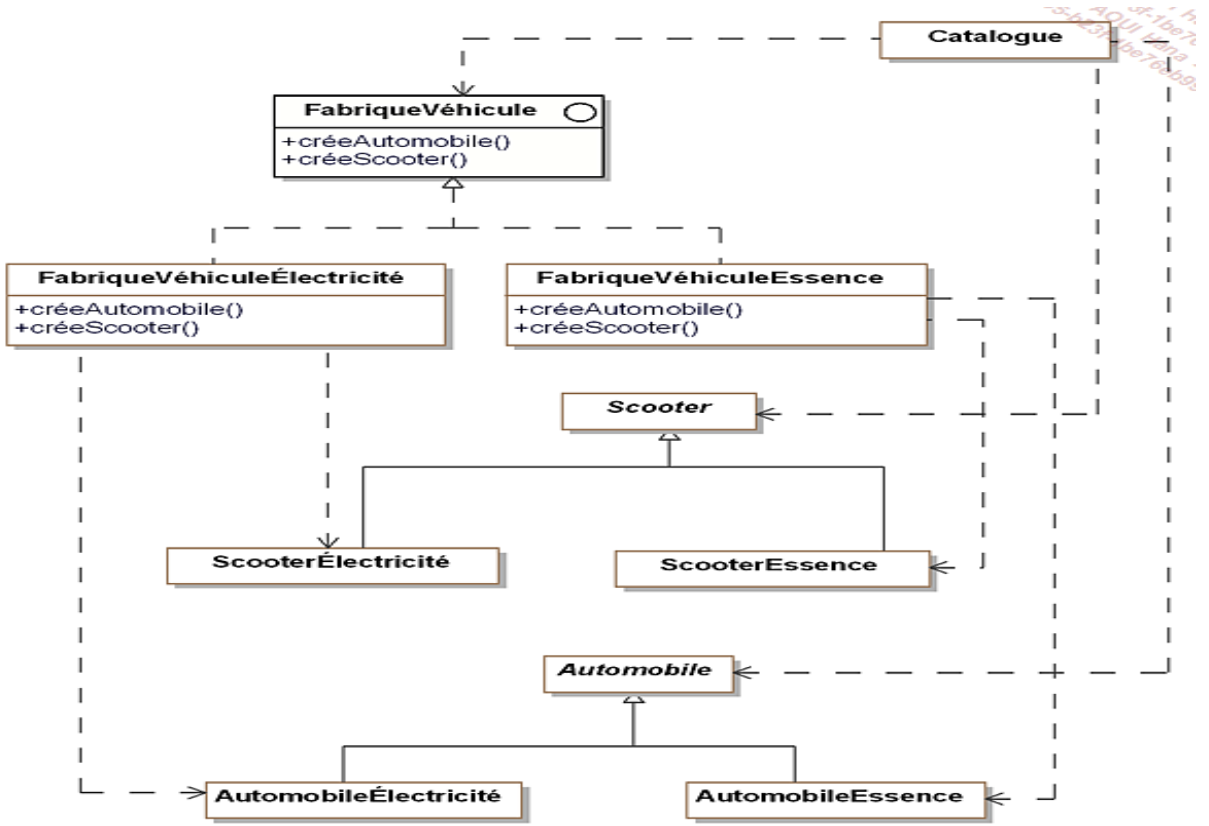
Le pattern Abstract Factory introduit une interface FabriqueVéhicule qui contient la signature des méthodes pour définir chaque produit.

Le type de retour de ces méthodes est constitué par l'une des classes abstraites de produit.

→ l'objet Catalogue n'a pas besoin de connaître les sous-classes concrètes et reste indépendant des familles de produit.

Pattern Abstract Factory





Structure générique du pattern `Abstract Factory`

Pattern Abstract Factory

Participants

Les participants au pattern sont les suivants :

- **FabriqueAbstraite** (FabriqueVéhicule) est une interface spécifiant les signatures des méthodes créant les différents produits.
- **FabriqueConcrète1, FabriqueConcrète2** (FabriqueVéhiculeÉlectricité, FabriqueVéhiculeEssence) sont les classes concrètes implantant les méthodes créant les produits pour chaque famille de produits. Connaissant la famille et le produit, elles sont capables de créer une instance du produit pour cette famille.

Pattern Abstract Factory

Participants

- **ProduitAbstraitA et ProduitAbstraitB** (Scooter et Automobile) sont les classes abstraites des produits indépendamment de leur famille. Les familles sont introduites dans leurs sous-classes concrètes.
- **Client** (catalogue) est la classe qui utilise l'interface de FabriqueAbstraite.

Pattern 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.

Pattern Abstract Factory

Code en Java du pattern (A faire en TP)

- Donner le code Java correspondant à la classe abstraite Automobile et ses sous-classes.
- Donner le code Java correspondant à la classe abstraite Scooter et ses sous-classes

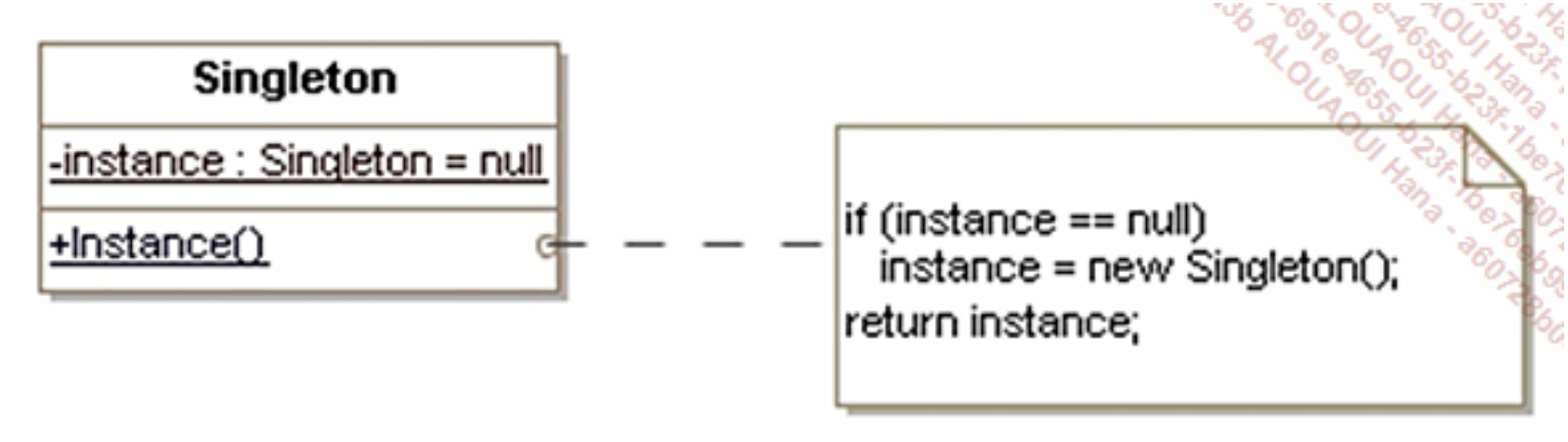
Patterns de construction

Pattern 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.
- 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.

Pattern Singleton



Structure du pattern Singleton

Pattern Singleton

Participants

- 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.

Pattern Singleton

Code en Java du pattern (A faire en TP)

La classe Vendeur

- Dans le système de vente de véhicules, un vendeur est représenté par une classe afin d' y mémoriser ses informations plutôt que d'utiliser des variables globales qui vont respectivement contenir son nom, son adresse, etc.

- Donner le code en java de la classe vendeur.
- Donner le programme principal utilisant la classe Vendeur afin d'afficher le nom, l'adresse et l'email d'un vendeur.

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.**

Patterns de structuration

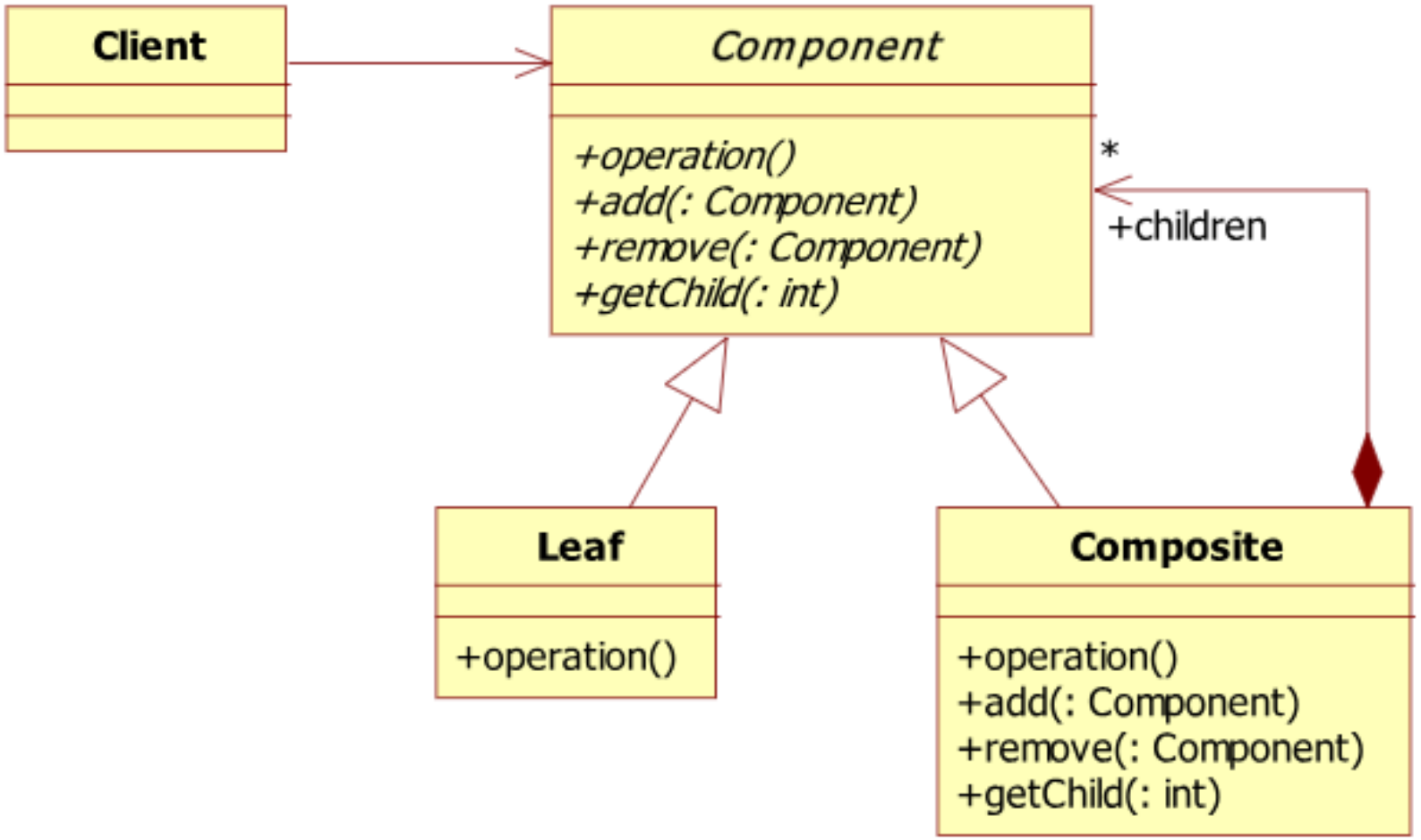
Pattern Composite

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.
- Le pattern est utilisé quand il est nécessaire de représenter au sein d'un système des hiérarchies de composition.

Patterns de structuration

Pattern Composite



Prise en compte des patterns de conception
Pattern Composite

Description de la partie	Pattern de conception
Construire les objets du domaine (automobile à essence, automobile à électricité, etc.).	Abstract Factory
Représenter un Vendeur	Singleton
Représenter les sociétés clientes.	**Composite**
Calculer le montant d'une commande.	Template Method

Patterns de structuration

Pattern Composite

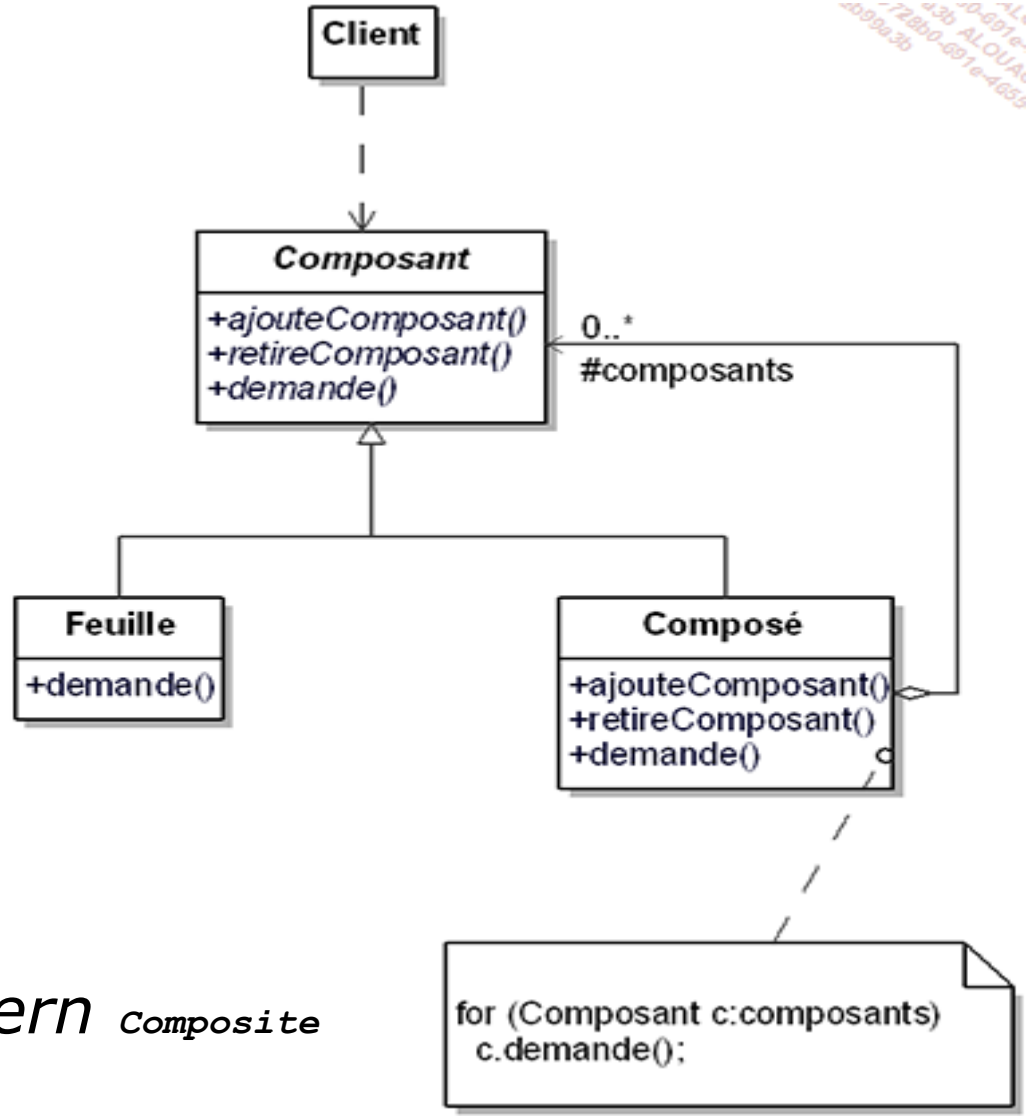
Exemple

- 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.

Patterns de structuration

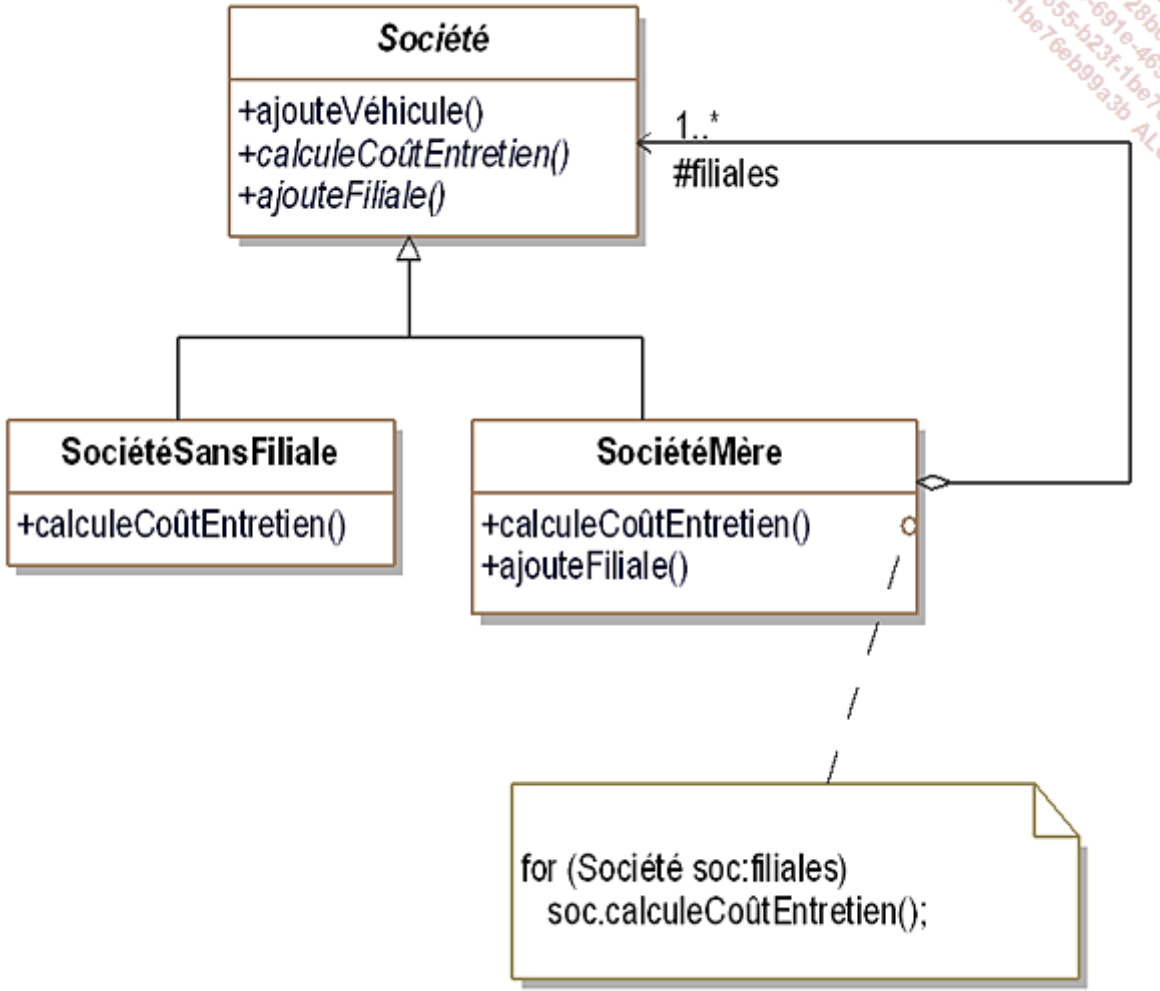
Pattern Composite



Structure du pattern `Composite`

Patterns de structuration

Pattern Composite



Le pattern `composite` *appliqué à la représentation de sociétés et de leurs filiales*

Patterns de structuration

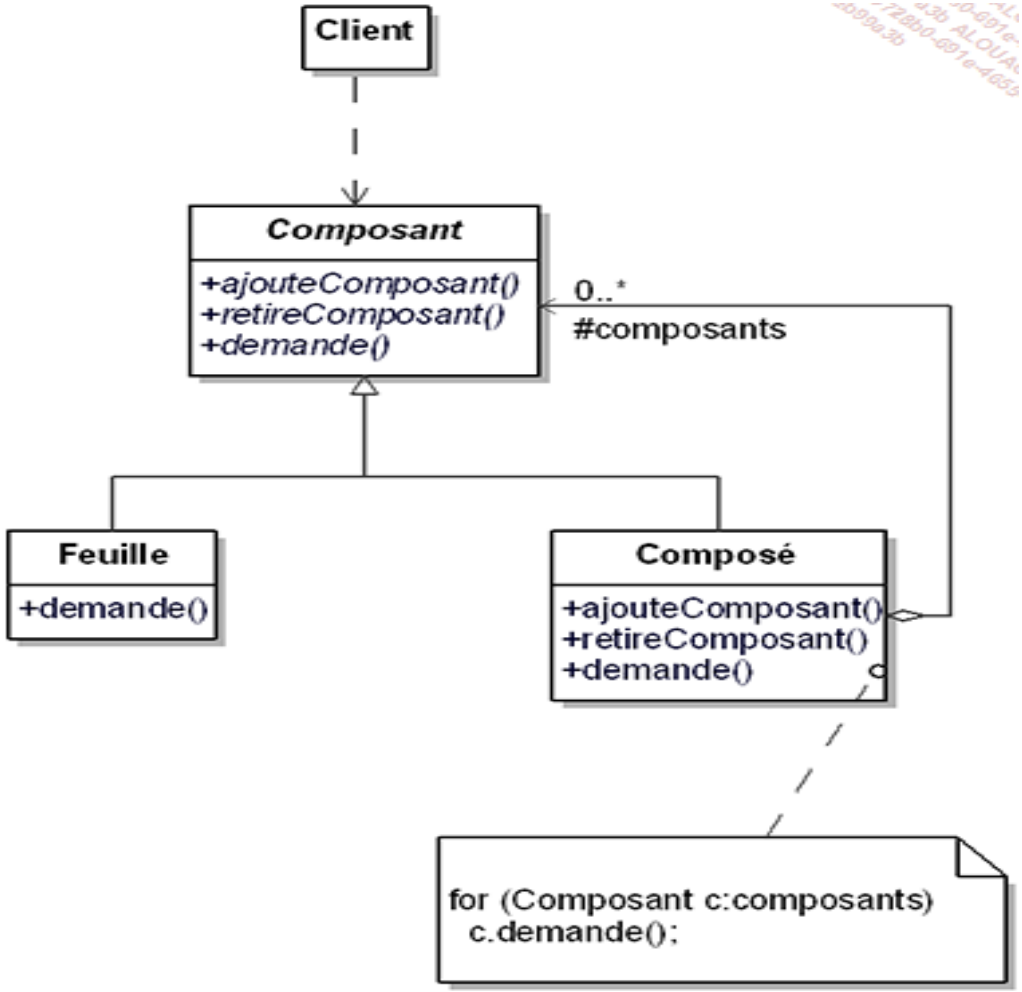
Pattern Composite

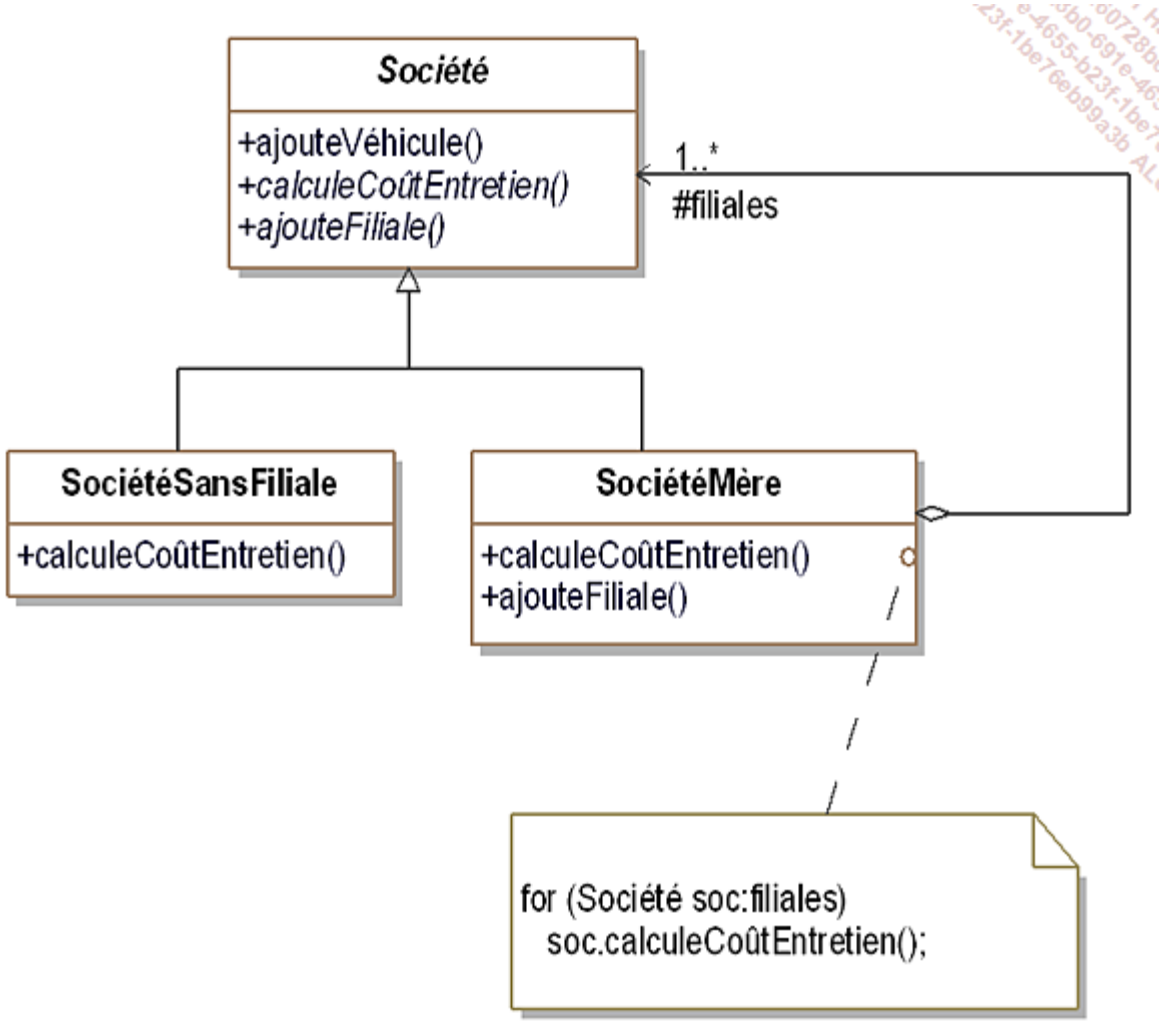
- La classe abstraite Société détient l'interface destinée aux clients.
- Elle possède deux sous-classes concrètes à savoir SociétéSansFiliale et SociétéMère.
- SociétéMère a une relation d'agrégation avec la classe Société représentant les liens avec ses filiales.

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.

Patterns de structuration
Pattern Composite





Structure du pattern **`Composite`**

Pattern Composite
Participants

Les participants au pattern sont les suivants :

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

Pattern Composite
Participants

- 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 manipule.

Pattern Composite

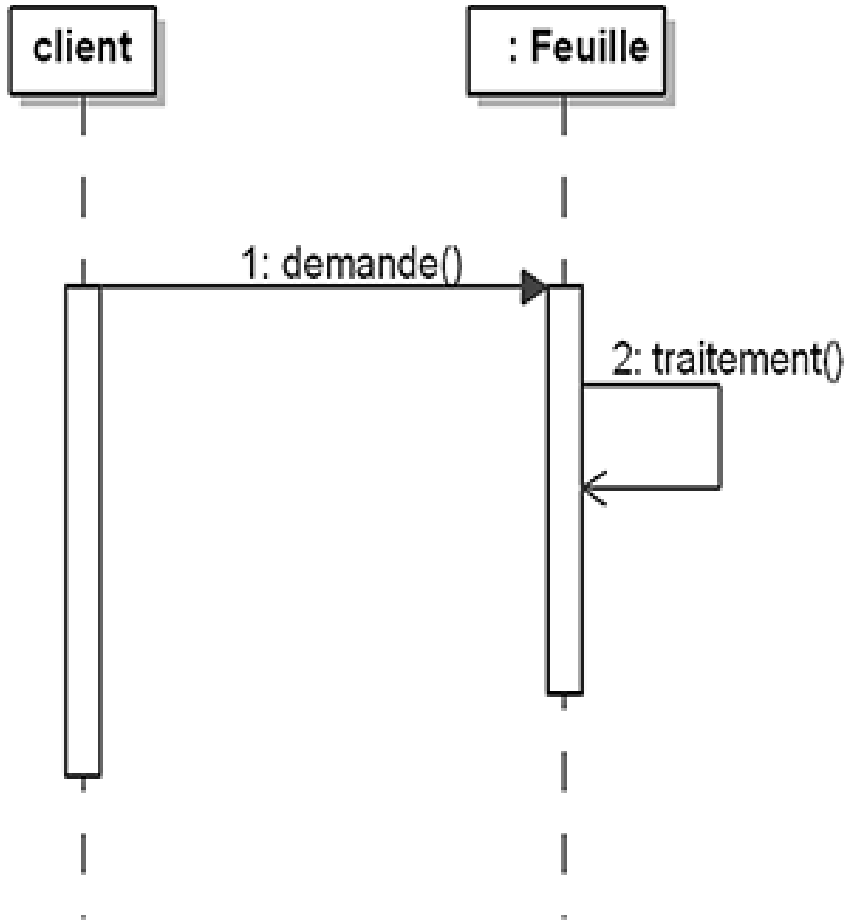
Collaborations

- 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 suivante.

Pattern Composite

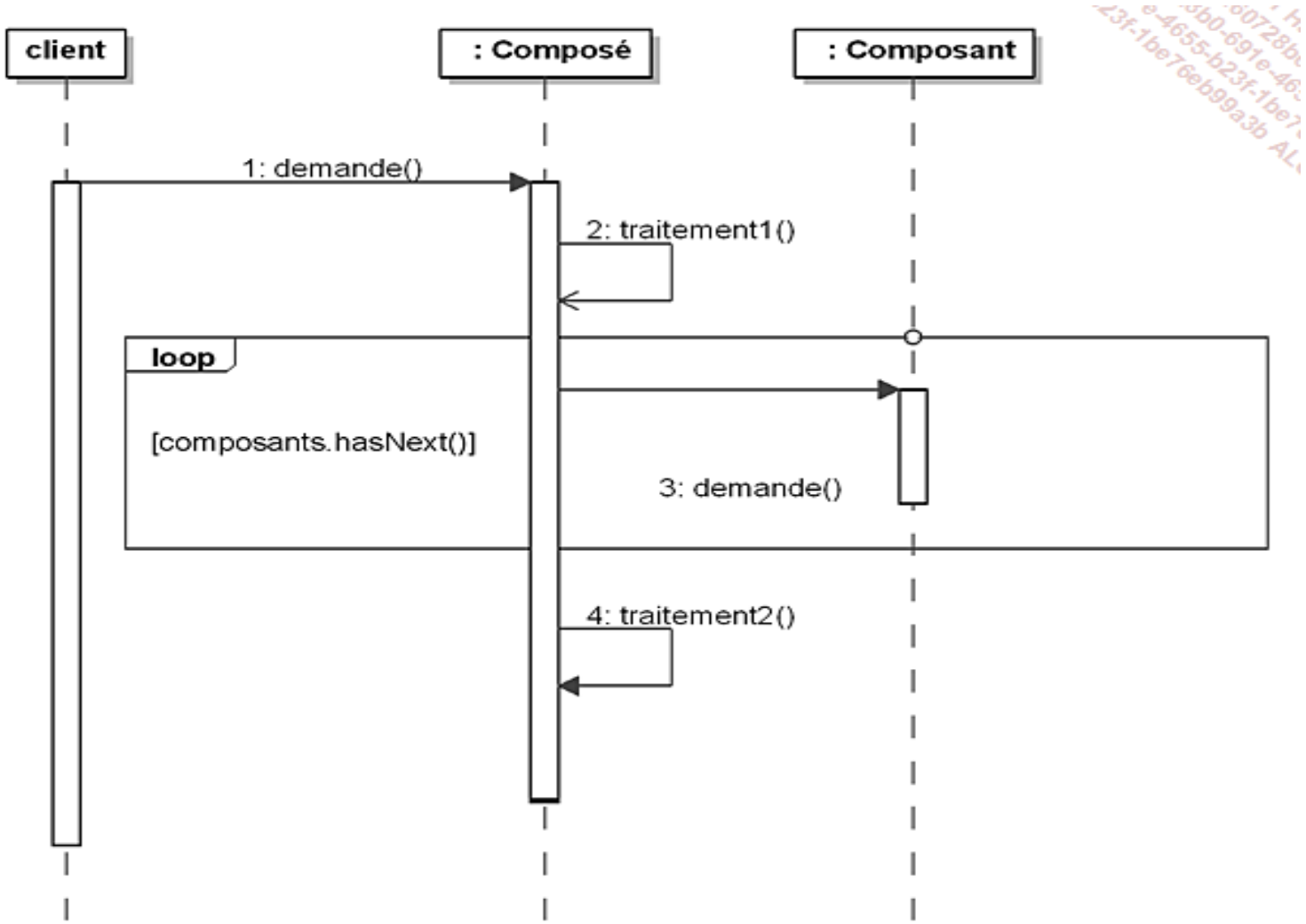


Traitement d'un message par une feuille

Pattern Composite

- 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.

Pattern Composite



Traitement d'un message par un composé

Patterns de comportements

Présentation

- 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 entre les objets.

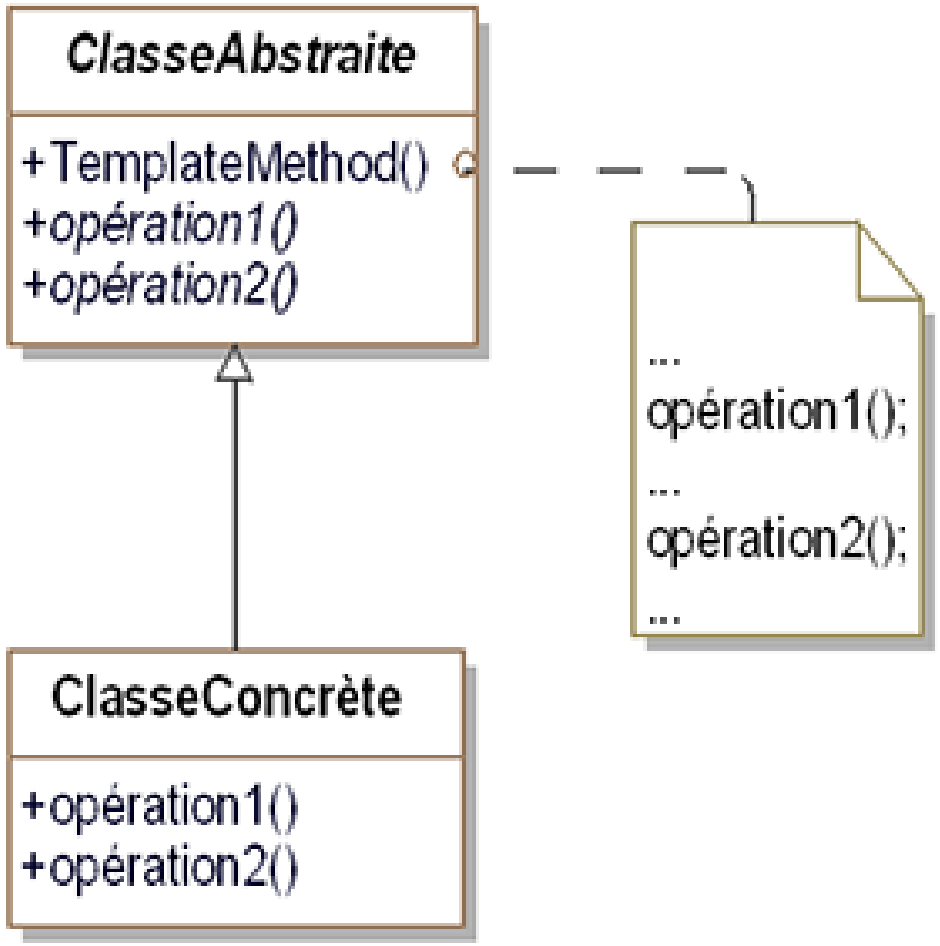
→ Ces patterns organisent les objets ainsi que leurs interactions en spécifiant les flux de contrôle et de traitement au sein d'un système d'objets.

Patterns de comportements

Distribution par héritage

- Distribuer un traitement en le répartissant dans les sous-classes.
- Cette répartition se fait par l'utilisation dans la classe de méthodes abstraites qui sont implantées dans les sous-classes.

Patterns de comportements



Structure générique du pattern Template Method

Patterns de comportements
pattern Template Method

Description

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

Prise en compte des patterns de conception

Description de la partie	Pattern de conception
Construire les objets du domaine (automobile à essence, automobile à électricité, etc.).	Abstract Factory
Représenter un Vendeur	Singleton
Représenter les sociétés clientes.	Composite
Calculer le montant d'une commande.	**Template Method**

pattern 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 le calcul de la TVA.

- En France, le taux de TVA est de 19,6 %

- Au Luxembourg, il est de 15 %

Le calcul de la TVA demande deux opérations de calcul distinctes en fonction du pays.

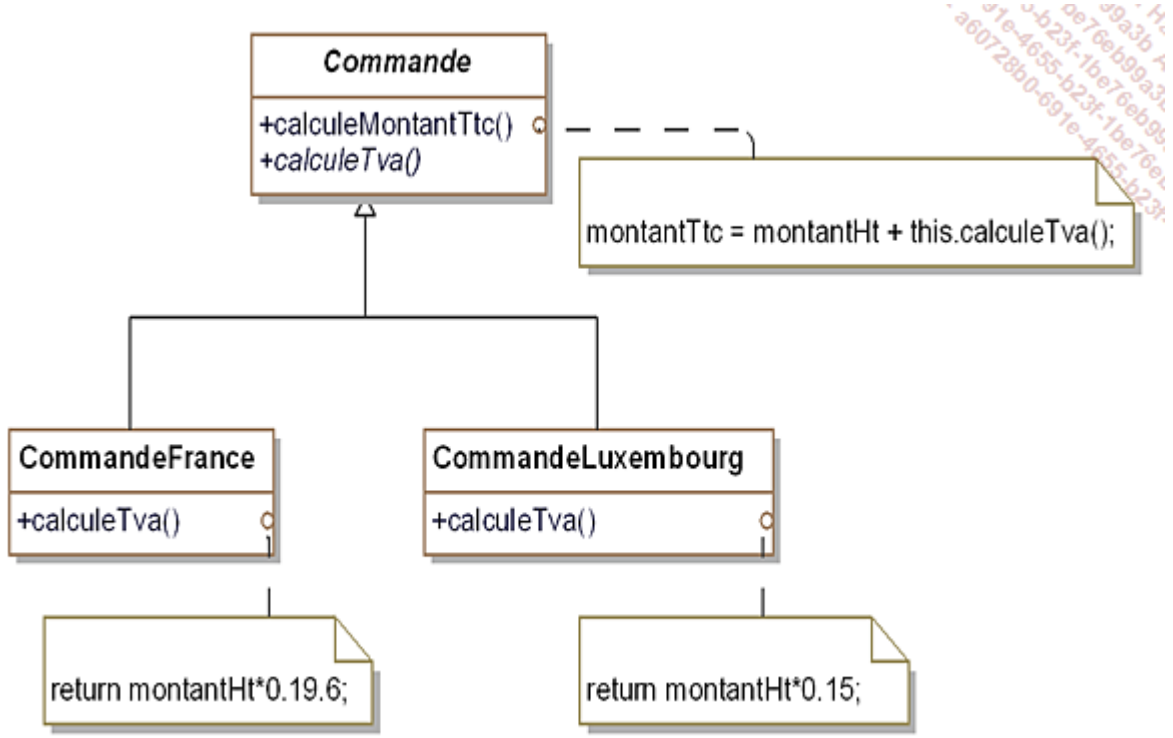
pattern Template Method

Une première solution consiste à implanter deux classes distinctes sans surclasse commune : CommandeFrance et CommandeLuxembourg.

inconvénient majeur: 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

pattern Template Method



Exemple d'utilisation du pattern Template Method

pattern Template Method

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.

pattern Template Method

- 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 suivante. La TVA est calculée avec le taux de 19,6 %.

pattern Template Method

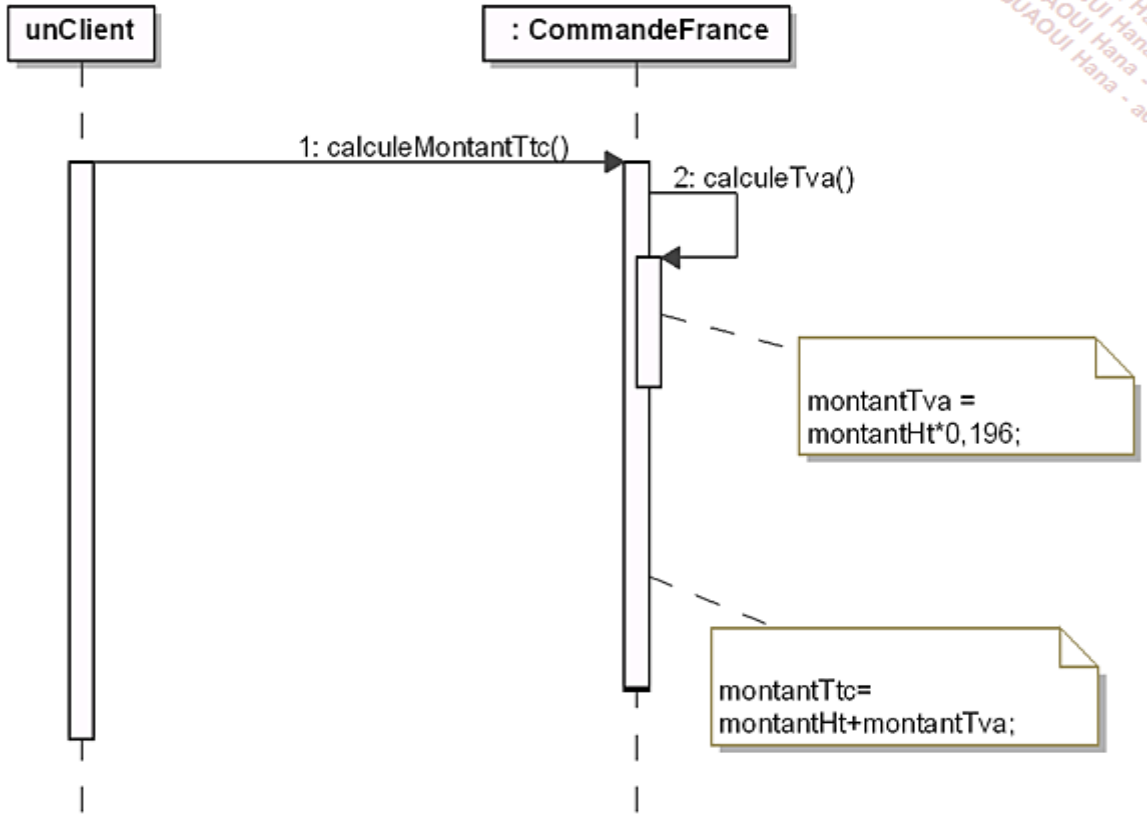


Diagramme de séquence du calcul du montant TTC d'une commande française

pattern Template Method

Si cette classe est CommandeLuxembourg, le diagramme de séquence est décrit à la figure suivante. La TVA est calculée avec le taux de 15 %.

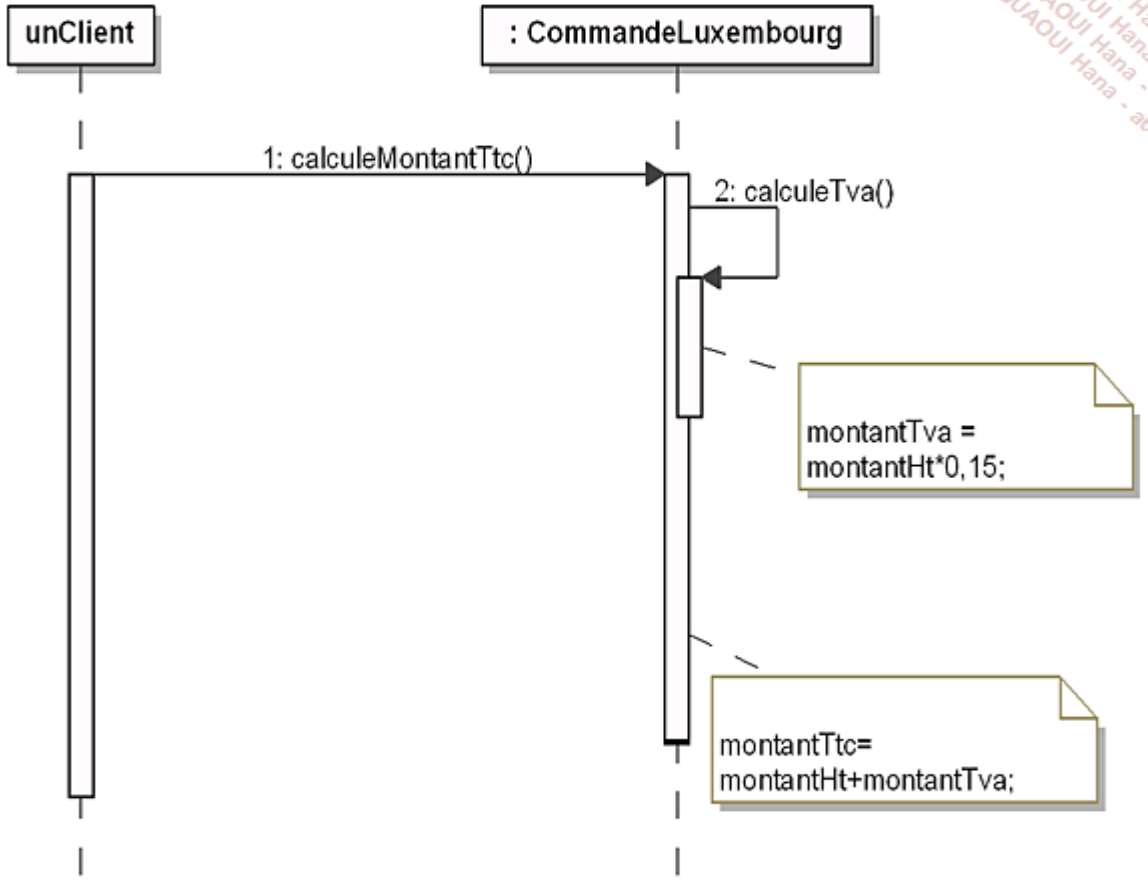


Diagramme de séquence du calcul du montant TTC d'une commande luxembourge

Annexes

Le catalogue des patterns de conception

vingt-trois patterns de conception :

- **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.

Le catalogue des patterns de conception

- **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.

Le catalogue des patterns de conception

- **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.

Le catalogue des patterns de conception

- **Proxy** : construit un objet qui se substitue à un autre objet et qui contrôle son accès.
- **Chain of responsibility** : crée une chaîne d'objets telle que si un objet de la chaîne ne peut pas répondre à une requête, il puisse la transmettre à ses successeurs jusqu'à ce que l'un d'entre eux y réponde.
- **Command** : a pour objectif de transformer une requête en un objet, facilitant des opérations comme l'annulation, la mise en file des requêtes et leur suivi.
- **Interpreter** : fournit un cadre pour donner une représentation par objets de la grammaire d'un langage afin d'évaluer, en les interprétant, des expressions écrites dans ce langage.
- **Iterator** : fournit un accès séquentiel à une collection d'objets sans que les clients se préoccupent de l'implantation de cette collection.

Le catalogue des patterns de conception

- **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.

Le catalogue des patterns de conception

- **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.
-