Design Pattern

Objectifs :

1. Introduire la notion des designs Pattern
2. Comprendre comment utiliser les design Patterns
3. Comprendre le fonctionnement d'un design Patterns depuis son diagramme de classe
4. Cas pratiques (en Java)

Définition des design patterns

- En informatique, et plus particulièrement en développement logiciel, un patron de conception (en anglais : « design pattern ») est un arrangement caractéristique de modules, reconnu comme bonne pratique en réponse à un problème de conception d'un logiciel. Il décrit une solution standard, utilisable dans la conception de différents logiciels
- Un patron de conception est issu de l'expérience des concepteurs de logiciels. Il décrit sous forme de diagrammes un arrangement récurrent de rôles et d'actions joués par des modules d'un logiciel, et le nom du patron sert de vocabulaire commun entre le concepteur et le programmeur

Définition des design patterns

- Des architectures de classes permettant d'apporter une solution à des problèmes fréquemment rencontrés lors des phases d'analyse et de conception d'applications.

- Un design pattern est un élément de logiciel orienté objet réutilisable présentant une solution générique à des problèmes récurrents.

Résumé

Un Design pattern n'est pas :

- la description d'un algorithme
- un fragment de code
- une méthode d'analyse

Mais plutôt :

- Une recette de conception
- Un guide, catalogue de solutions classiques

Objectifs

- La réutilisation de l'expérience.
- Présenter des solutions sans s'attacher aux détails du problème à résoudre.

Avantages et inconvénients

Avantages:

- Une architecture facilement compréhensible et identifiables pour un programmeur.
- Réduire le temps de développement d'une application.
- Faciliter la maintenance d'une application.
- Solutions éprouvés, adaptables …

Avantages et inconvénients

Inconvénients:

- C'est comme une formule mathématique, c'est la solution mais encore faut-il l'appliquer au bon moment avec les bonnes variables
- Nécessite un apprentissage et de l'expérience.
- Les patterns sont nombreux, abstraits, peu clairs
- …

Classification des design Patterns

- On retrouve une classification en 3 catégories :
- Creational Patterns (Les Patterns de création)
- Structural Patterns (Les Patterns de structure)
- Behavioral Patterns (Les patterns de comportement)

Classification des design Patterns

- **Creational Patterns (Les Patterns de création) :**
  - ils définissent comment faire l'instanciation et la configuration des classes et des objets :
    - Abstract Factory (Fabrique abstraite),
    - Builder (Monteur),
    - Factory Method (Fabrique),
    - Prototype (Prototype),
    - Singleton (Singleton)

Classification des design Patterns

- **Structural Patterns (Les Patterns de structure) :**
  - Ils permettent la résolution des problèmes liés à la structuration des classes et leur interface en particulier. :
    - Adaptateur (Adapter)
    - Pont (Bridge)
    - Objet composite (Composite)
    - Décorateur (Decorator)
    - Façade (Facade)
    - Poids-mouche ou poids-plume (Flyweight)
    - Proxy (Proxy)

Classification des design Patterns

- **Behavioral Patterns (Les patterns de comportement)**
  - ils définissent comment organiser les objets pour que ceux-ci collaborent (distribution des responsabilités) et expliquent le fonctionnement des algorithmes impliqués.
    - Chaîne de responsabilité (Chain of responsibility)
    - Commande (Command)
    - Interpréteur (Interpreter)
    - Itérateur (Iterator)
    - Médiateur (Mediator)
    - Mémento (Memento)

Classification des design Patterns

- **Behavioral Patterns (Les patterns de comportement)**
  - ils définissent comment organiser les objets pour que ceux-ci collaborent (distribution des responsabilités) et expliquent le fonctionnement des algorithmes impliqués :
    - Observateur (Observer)
    - État (State)
    - Stratégie (Strategy)
    - Patron de méthode (Template Method)
    - Visiteur (Visitor)
    - Fonction de rappel (Callback)

Patrons de conception du GoF

- Les patrons de conception les plus connus sont au nombre de 23. Ils sont couramment appelés « patrons GoF » (de « Gang of Four », d'après les quatre créateurs du concept). Le patron Modèle-Vue-Contrôleur (MVC) est une combinaison des patrons *Observateur*, *Stratégie* et *Composite*, ce qui forme ainsi un patron d'architecture.

Pattern Adaptator

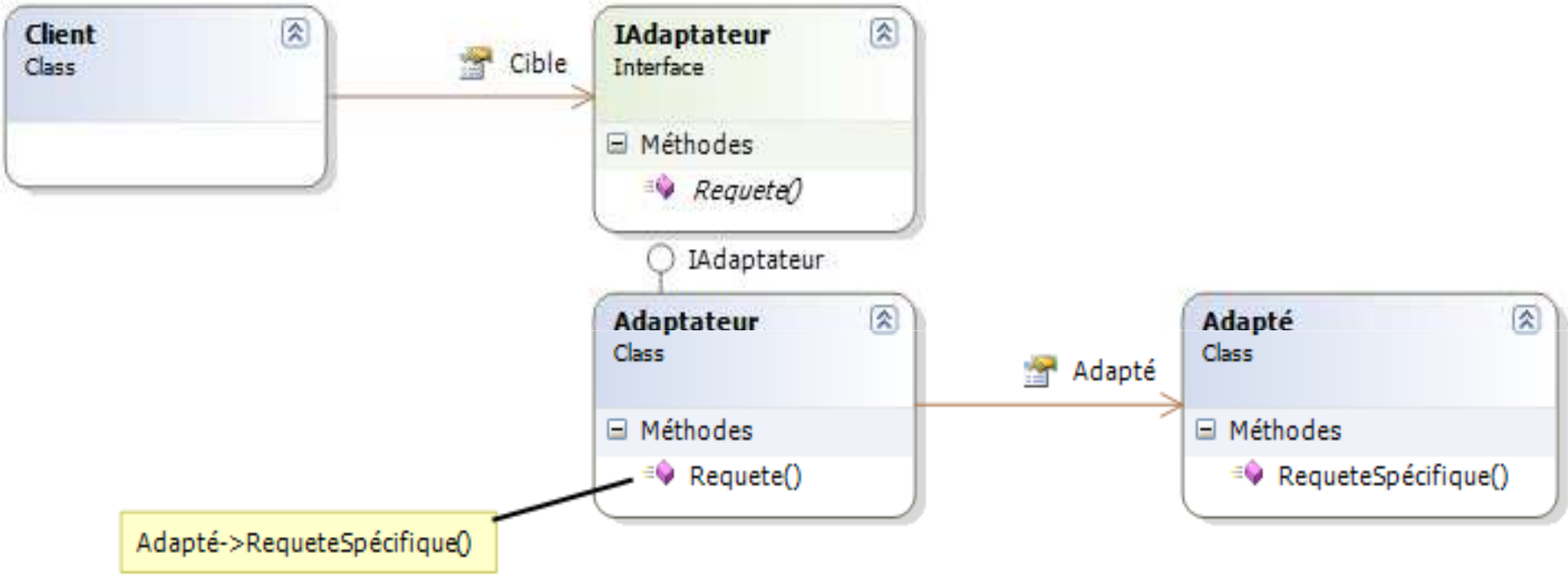
Pourquoi un Adaptateur

- Pour des raisons de conformité à une norme
- Moyen commode de faire fonctionner un objet avec une interface qu'il ne possède pas.

⇨ Conférer à une classe existante une nouvelle interface pour répondre aux besoins de clients.

⇨ Concevoir une classe supplémentaire qui se charge d'implémenter la bonne interface (L'Adaptateur) et d'appeler les méthodes correspondantes dans l'objet à utiliser (L'adapté).

Structure générique Adaptateur



Participants

I' adaptateur (interface): Définit l'interface métier utilisée par le **Client**.
Client : Travaille avec des objets implémentant l'interface **IAdaptateur**.
Adapté : Définit une interface existante devant être adaptée.
Adaptateur : implante les méthodes l'interface **Iadaptateur** en invoquant les méthodes de **Adapté** .

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.
- Une première classe d'implantation de cette méthode a été réalisée : la classe *DocumenHtml* qui implante des méthodes de cette interface .

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 pas à l'interface *Document*.

Le pattern Adapter propose une solution qui consiste à créer la classe *DocumentPdf* implantant L'interface *Document* et possédant une association avec *ComposantPdf*

Exemple en java

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

```
public class DocumentHtml implements Document {

   String contenu;

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

   }

   public void dessine() {
        // redefinition de dessine
   }

        public void imprime() {
        // redefinition de imprime
   }


}
```

```java
public class ComposantPdf
{
    String contenu;

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

    public void pdfPrépareAffichage() {    }

    public void pdfRafraîchit() {   }

    public void pdfTermineAffichage() {   }

    public void pdfEnvoieImprimante() {   }

}
```

Classe adapté :Le composant existant qui est intégré dans l'application, indépendant de l'application, en particulier de l'interface *Document*

L'adaptateur:

-Associée à la classe adapté par attribut *outilPdf* .

- Implante *Document*

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

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

    }

    public void dessine(){
        outilPdf.pdfRafraîchit();
        outilPdf.pdfRafraîchit();
        outilPdf.pdfTermineAffichage();
    }

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

Domaines d’applications

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 Composite

Le pattern Composite

Description

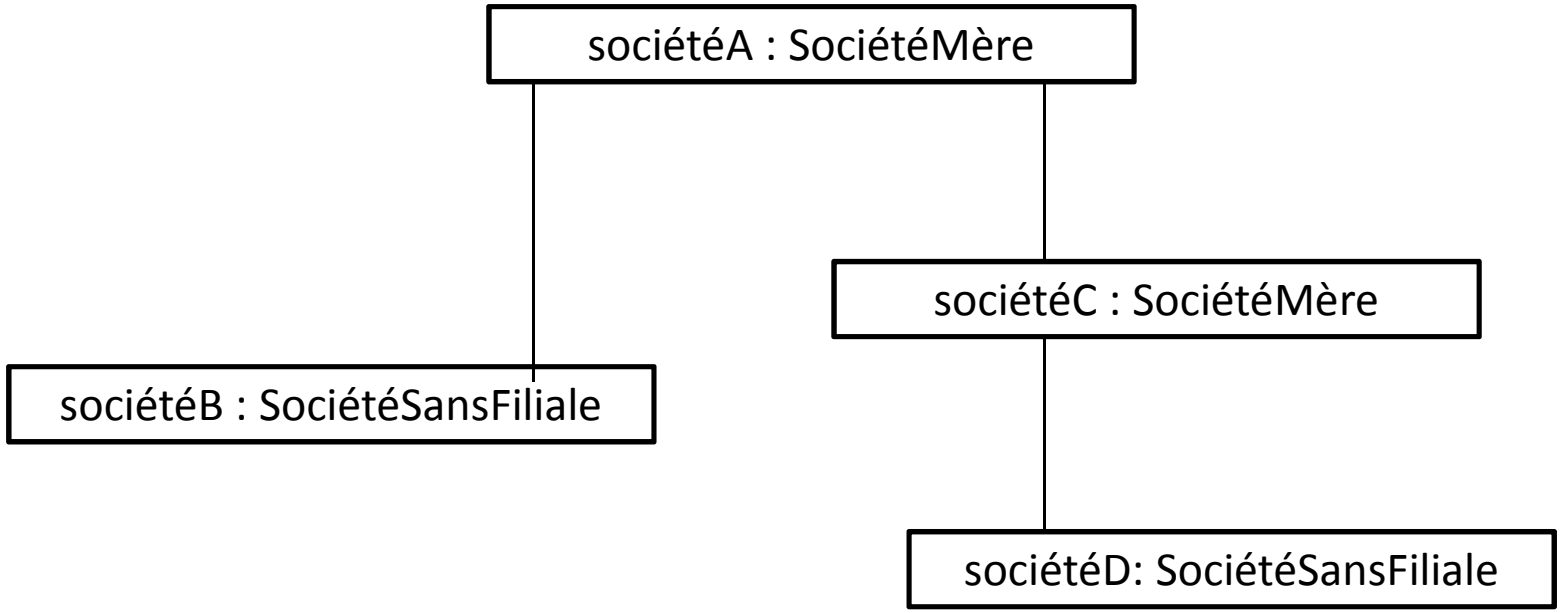
Le but du pattern Composite est d'offrir un cadre de conception d'une composition d'objets dont la profondeur est variable, cette conception étant basée sur un arbre

Exemple

Nous voulons représenter les sociétés clientes , notamment pour connaitre le nombre de véhicules dont elles disposent et leur proposer des offres de maintenance
De leur parc.

La solution consiste à traiter différemment les sociétés sans filiale et celle possédant des filiales

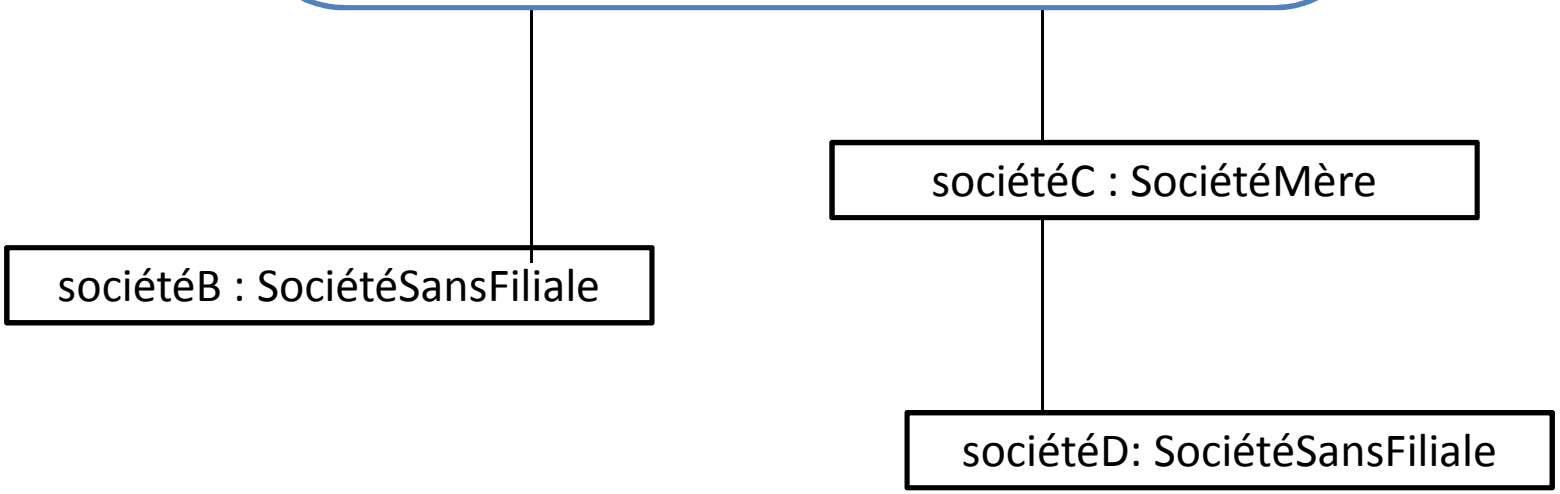
Cette différence de traitement entre les deux types de société rend l'application plus complexe et dépendante de la composition interne des sociétés clientes.



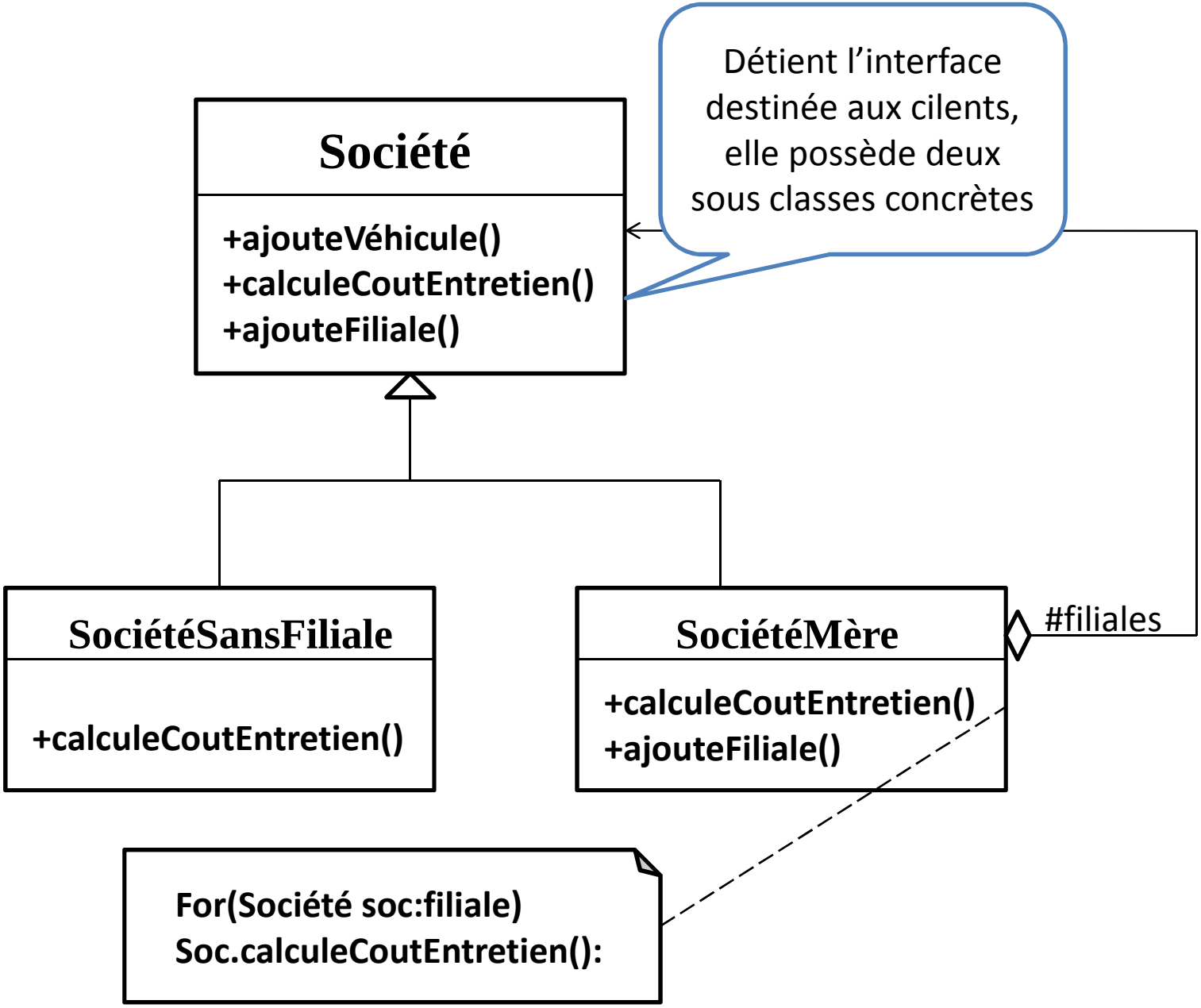
La solution consiste à traiter différemment les sociétés sans filiale et celle possédant des filiales

Cette di de
société nte
de la co

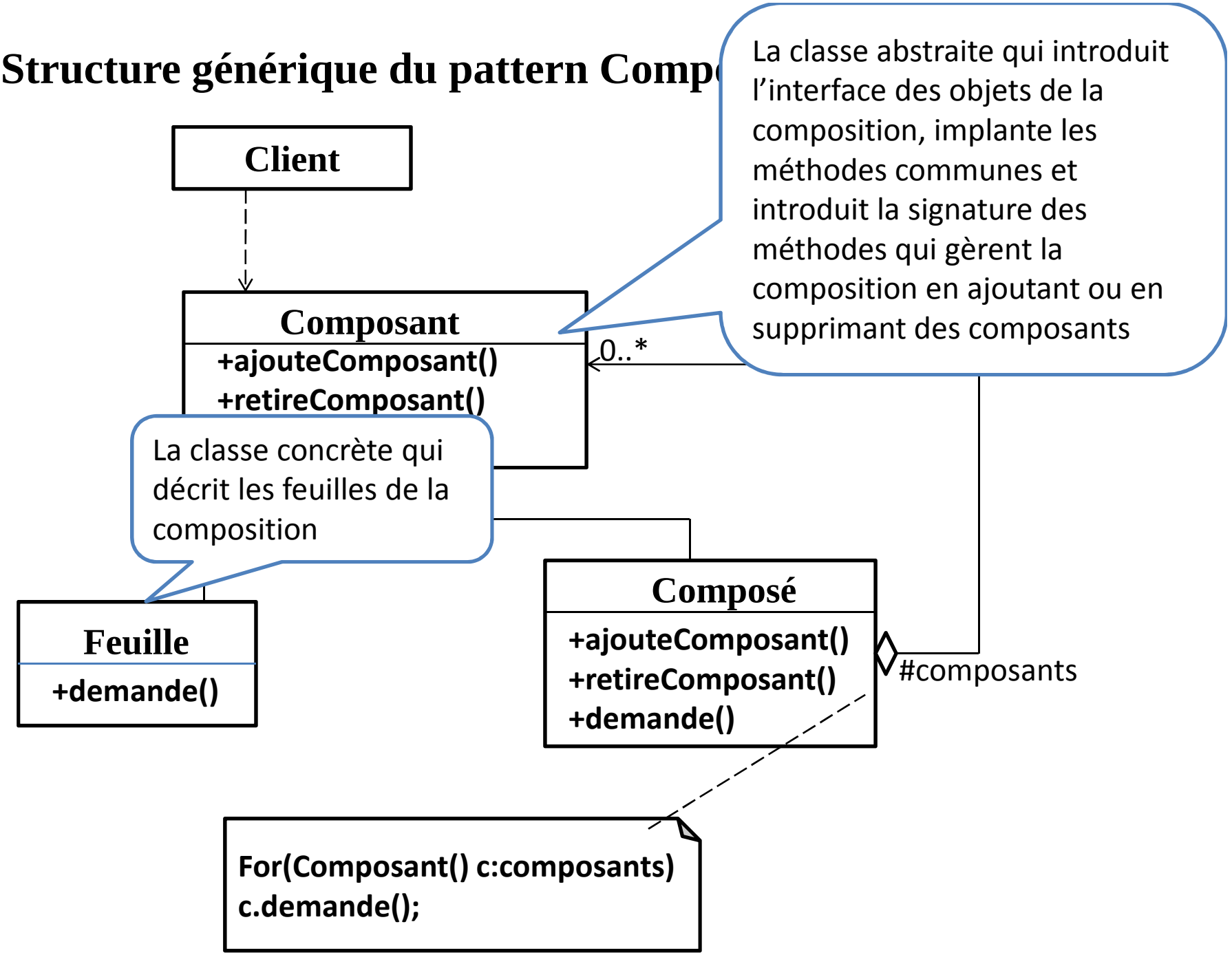
Le pattern composite résout ce problème en unifiant l'interface des deux types de sociétés et en utilisant la composition récursive



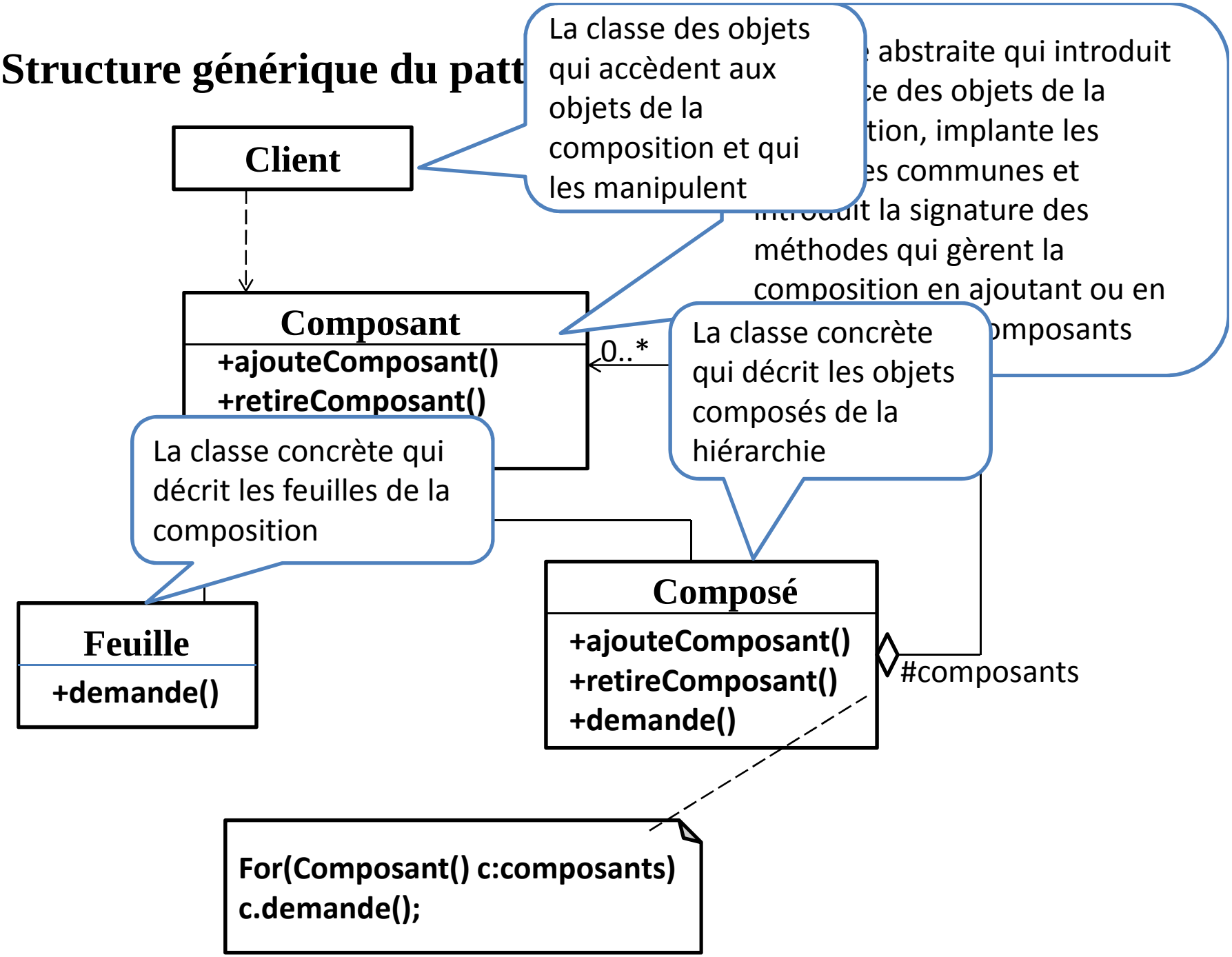
Le pattern Composite appliqué à la représentation de sociétés et de leurs filiales



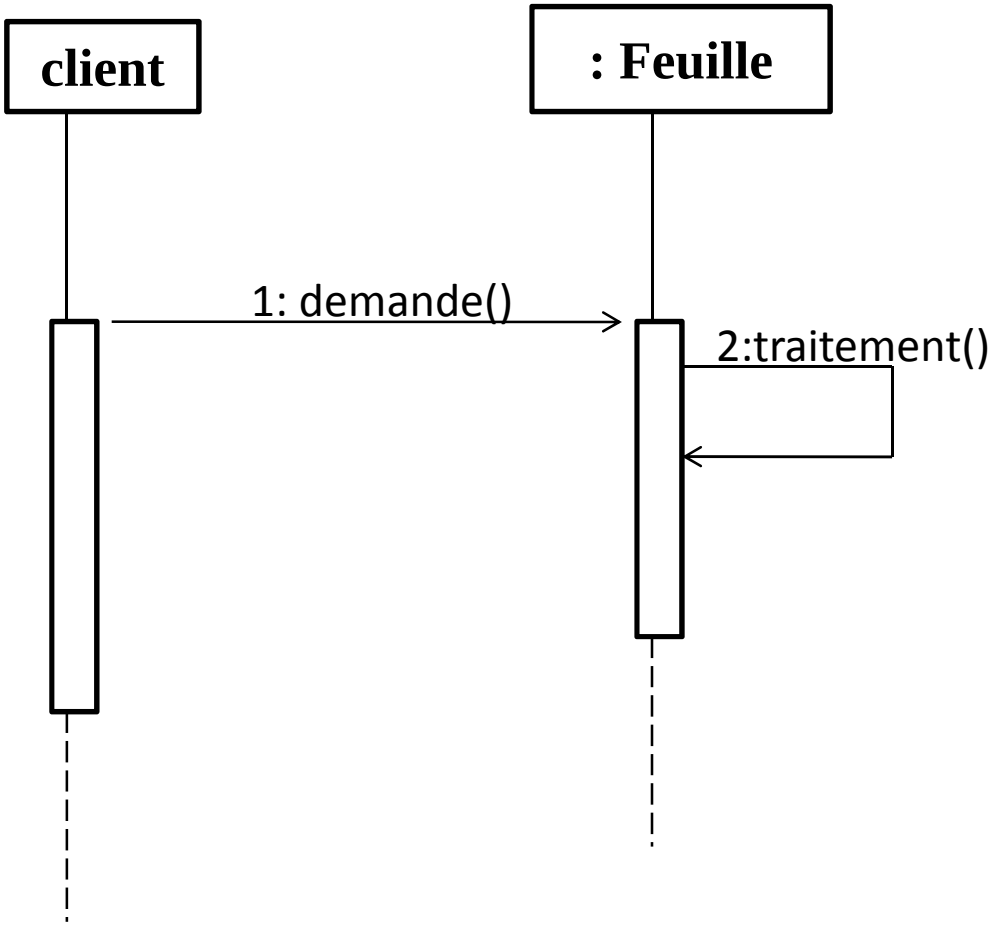
Structure générique du pattern Compo



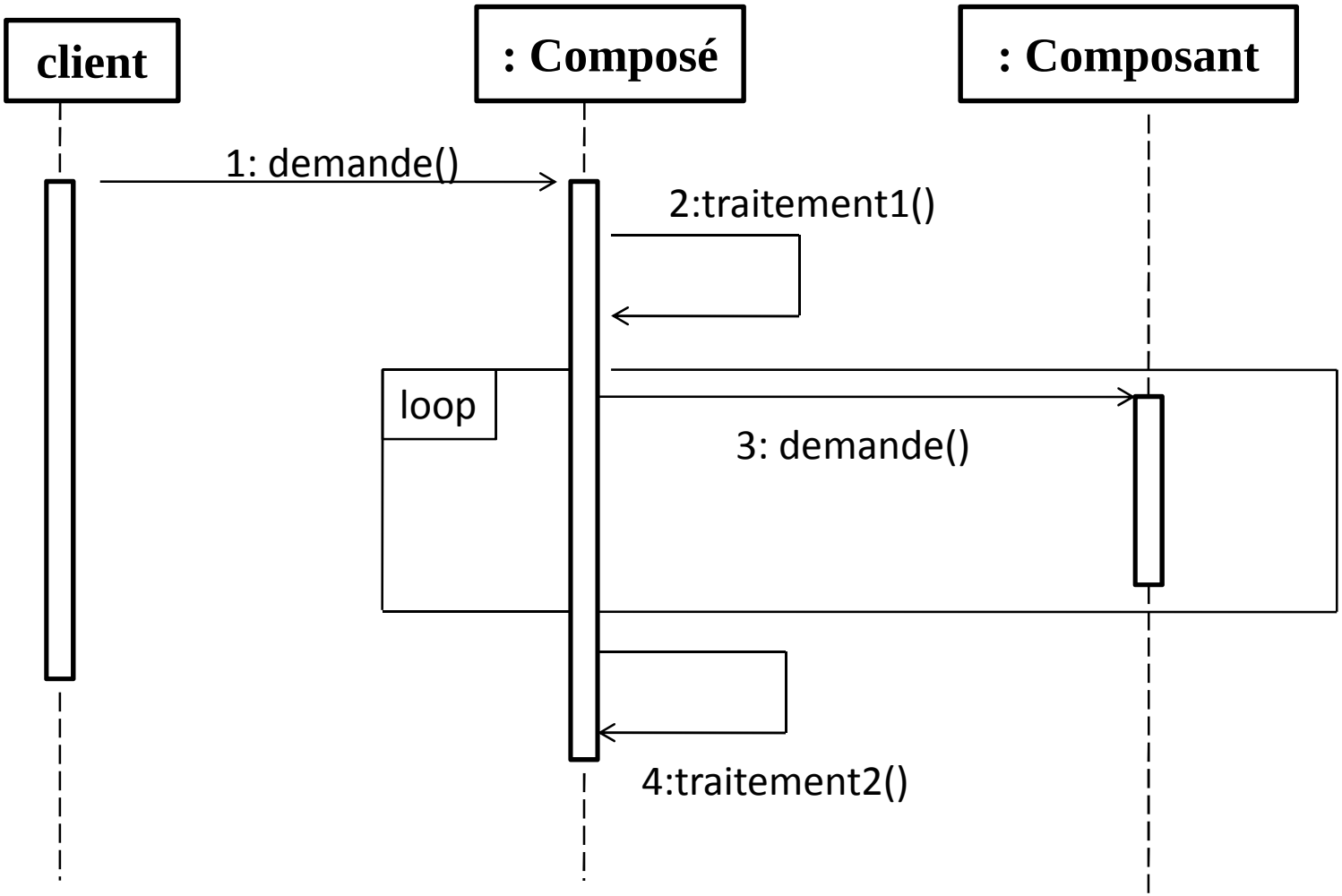
Structure générique du patt



Traitement d’un message par une feuille



Traitement d’un message par une feuille



Domaines d’application

- **Il est nécessaire de représenter au sein d’un système des hiérarchies de composition**

- **Les clients d’une composition doivent ignorer s’ils communiquent avec des objets composés ou non.**

Exemple en Java

```
Public abstract class Société{
        protected static double coutUnitVéhicule = 5.0;
        protected  int  nbrVéhicules;

Public void ajouteVéhicule(){
        nbrVéhicules = nbrVéhicules + 1;
}

Public abstract double calculeCoutEntretien();
Public abstract boolean ajouteFiliale(Société filiale);
}
```

```
Public class SociétéSansFiliale extends Société{
        public boolean ajouteFiliale(Société filiale){
        return false;
}
Public double calculeCoutEntretien(){
        return nbrVéhicules*coutUnitVéhicule;
}
}
```

Exemple en Java

```
Import java.util.*
Public class SociétéMère extends Société{
        List<Société> filiales = new ArrayList <Société>();
Public boolean ajoutefiliale(Société filiale){
        return filiales.add(filiale);
}

Public double calculeCoutEntretien(){
Double cout = 0.0;
For(Société filiale : filiales)
Cout = cout + filiale.calculeCoutEntretien();
Return cout + nbrVéhicules*coutUnitVéhicule;
}
}
```

Dont le résultat est la somme du cout des filiales et du cout de la société mère

Pattern Façade

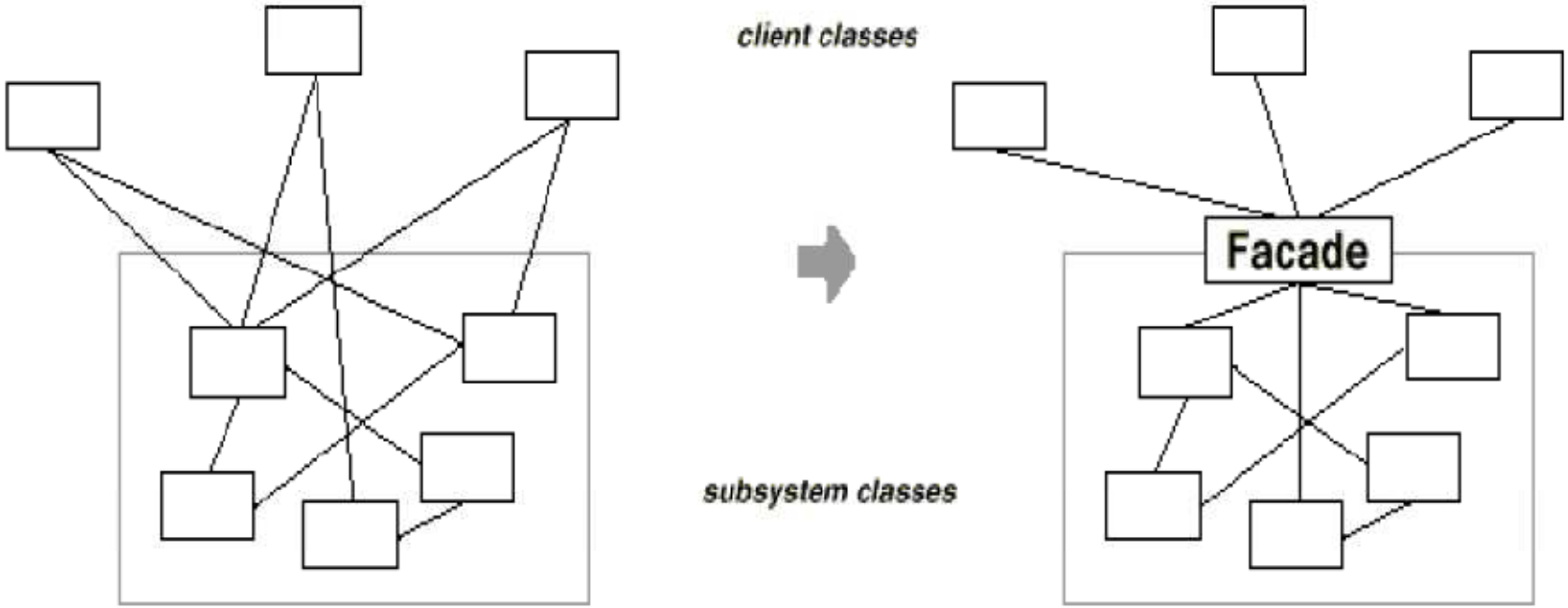
Le pattern Façade

Objectif

Regrouper les interfaces d'un ensemble d'objets en une interface unifiée

➔ Plus simple et plus ergonomique à utiliser pour un client

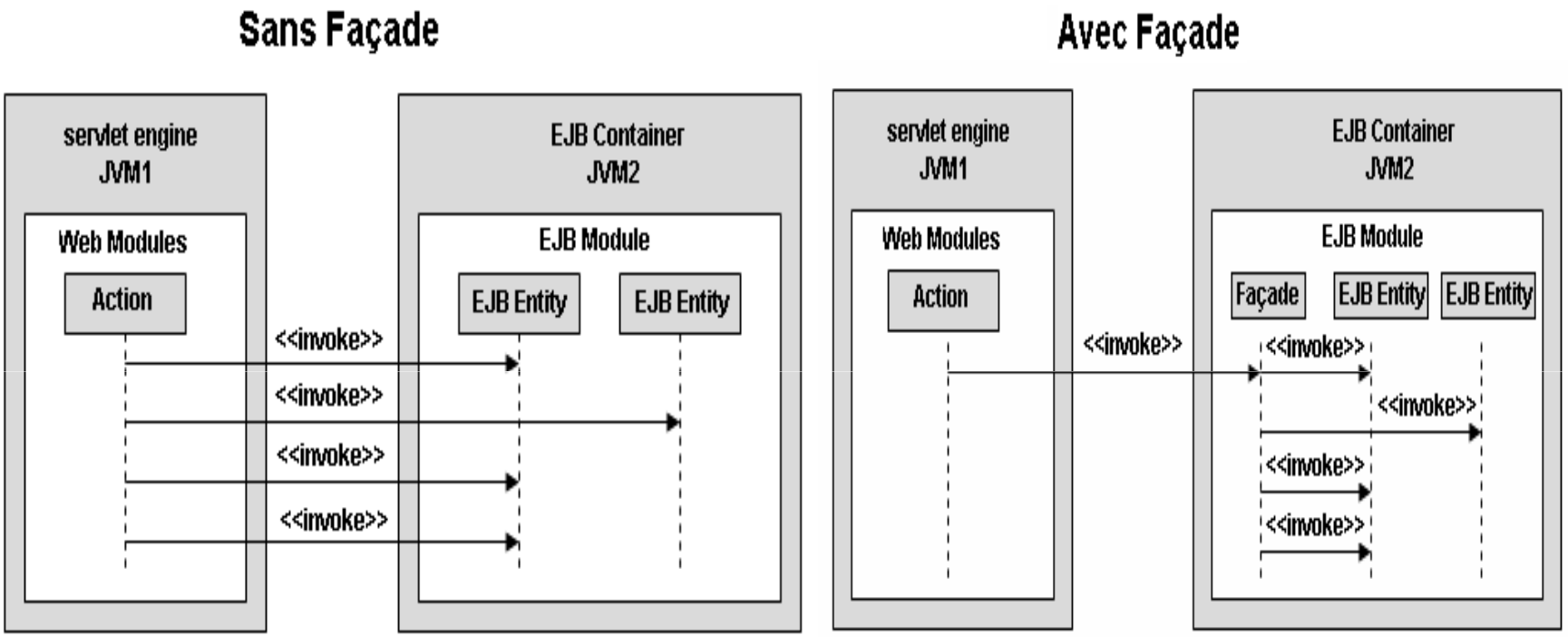
Structure de Façade



La Façade est une **classe sans attribut**. Ses instances sont donc sans état, indistincts

Inutile de créer plusieurs puisque toutes leurs instances seront identiques

Exemple


Sans Façade
servlet engine
JVM1
Web Modules
Action
EJB Container
JVM2
EJB Module
EJB Entity
EJB Entity
<<invoke>>
<<invoke>>
<<invoke>>
<<invoke>>
Avec Façade
servlet engine
JVM1
Web Modules
Action
<<invoke>>
EJB Container
JVM2
EJB Module
Façade
EJB Entity
EJB Entity
<<invoke>>
<<invoke>>
<<invoke>>
<<invoke>>
<<invoke>>

Le pattern Proxy

1. Généralités

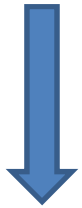
Le pattern proxy est utilisé lorsque l'on veut contrôler l'accès à un objet ou pour le remplacer.

Quand l'utiliser?

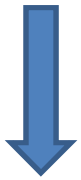
- Pour créer un objet de taille importante au moment approprié (proxy virtuel).
- Pour accéder à un objet s'exécutant dans un autre environnement. Ce type de proxy est mis en œuvre dans les systèmes d'objets distants (proxy remote).
- Pour sécuriser l'accès à un objet, par exemple par des techniques d'authentification (proxy de protection).

2. *Exemple*: Gérer l'affichage d'animations pour chaque véhicule du catalogue.

- Pour chaque véhicule visualiser un film via un click sur la photo.
- Une page du catalogue contient plusieurs véhicule.



- Il faut créer en mémoire tous les objets d'animation



- Lourd, lent, coûteux, …

2. *Exemple*: Gérer l'affichage d'animations pour chaque véhicule du catalogue.

- R… sur la photo.

Le pattern Proxy

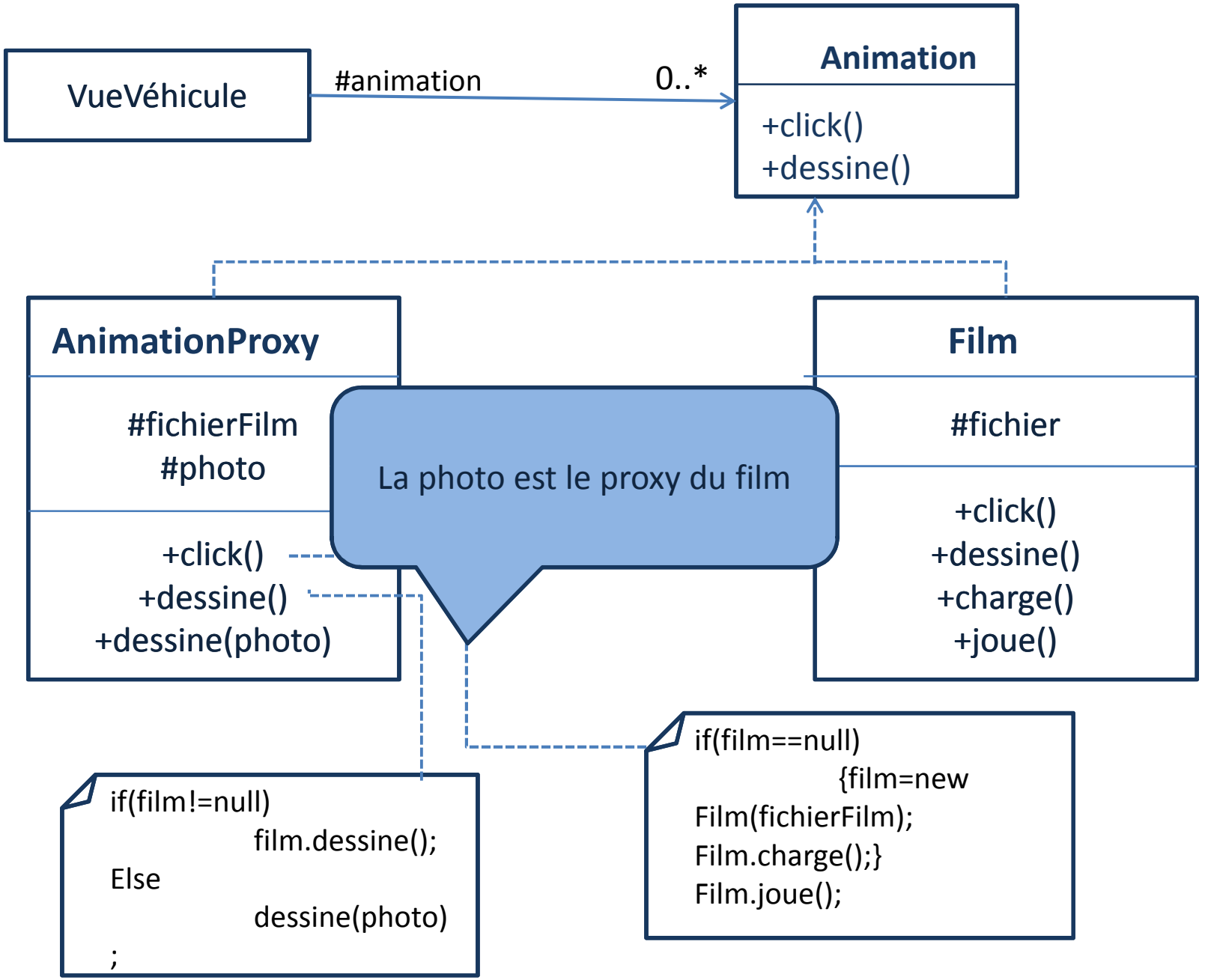
Différer la création des sujets jusqu'au moment où le système a besoin d'eux

- Il faut … animation

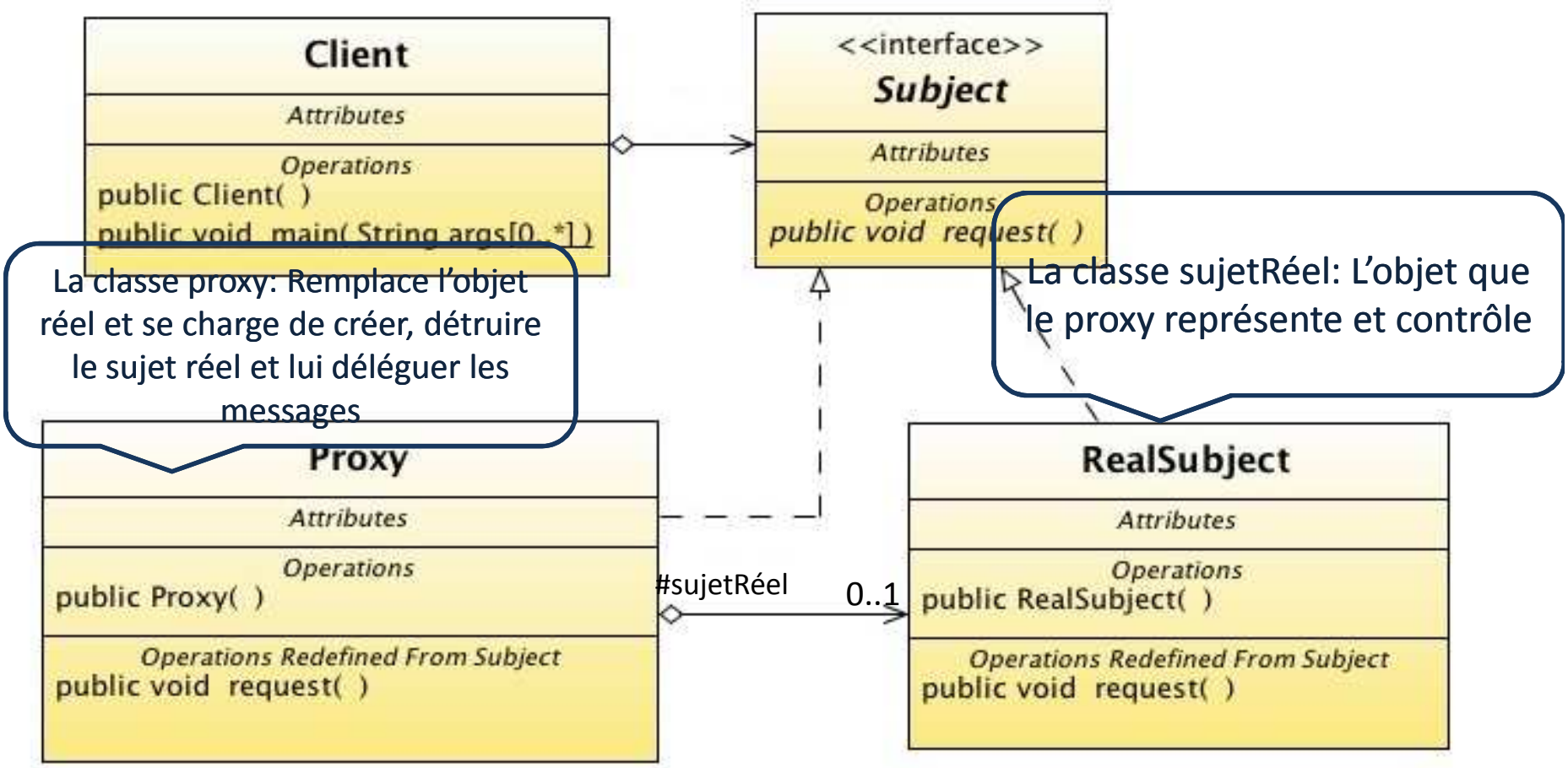
- Lourd, lent, coûteux, …

- **Diagramme de classe du pattern Proxy appliqué à l'affichage d'animation**

Le pattern Proxy



3. Structure du pattern



➢ 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.