Université de Monastir
Institut Supérieur d'Informatique et de Mathématiques
Département Informatique

Cours
Design Patterns et conception par contrats

ING 2 - Génie Logiciel

03

Patterns de Structure

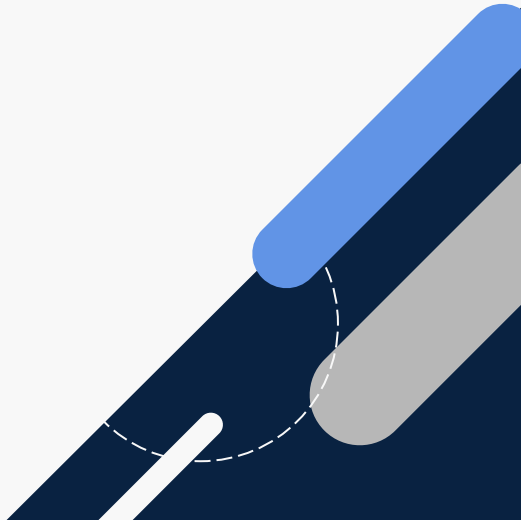
Objectifs des patterns de structure

- Les patterns de structure ont pour objectif de **composer** des classes et des objets pour former des structures plus grandes, tout en gardant ces structures **flexibles** et **extensibles**.
- Ils se concentrent sur la manière dont les objets sont reliés entre eux (composition, héritage, agrégation, délégation, etc.) plutôt que sur leur comportement interne.

Objectifs des patterns de structure

- Réutiliser le code existant sans le modifier (principe Open/Closed).
- Simplifier les relations entre objets (réduire la complexité des dépendances).
- Permettre des extensions futures sans casser le code existant.
- Encapsuler les variations structurelles (par exemple, remplacer une bibliothèque ou un composant sans tout réécrire).
- Faciliter la maintenance grâce à une architecture claire et découplée.

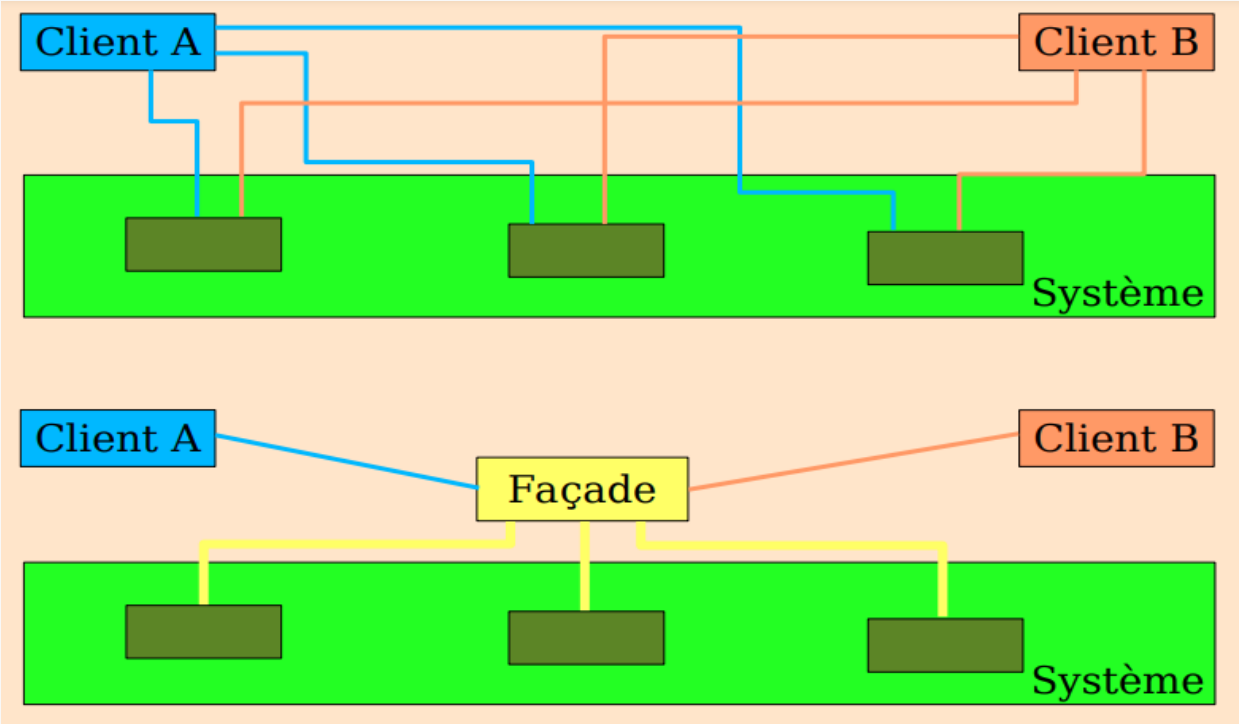
Pattern
Façade



Définition

- Façade est un patron de conception structurel qui procure une interface offrant un accès simplifié à une librairie, un framework ou à n'importe quel ensemble complexe de classes.
- Le patron façade **masque la complexité** du système et fournit une interface au client à l'aide de laquelle il peut accéder au système.
- Une façade ne se limite pas à simplifier une interface : elle **découple le client** d'un sous-système de composants (principe de faible couplage).

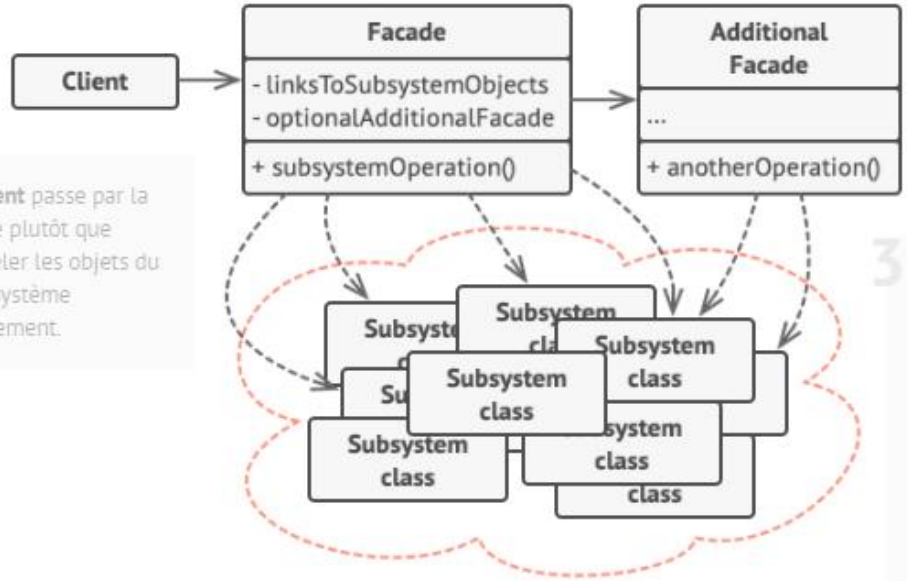
Principe



Structure UML

1 La **Façade** procure un accès pratique aux différentes parties des fonctionnalités du sous-système. Elle sait où diriger les requêtes du client et comment utiliser les différentes parties mobiles.

2 Une classe **Façade Supplémentaire** peut être créée pour éviter de polluer une autre façade avec des fonctionnalités qui pourraient la rendre trop complexe. Les façades supplémentaires peuvent être utilisées à la fois par le client et par les autres façades.



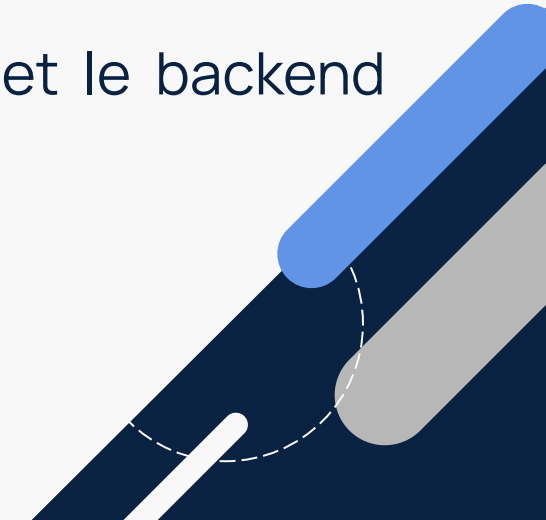
4 Le **Client** passe par la façade plutôt que d'appeler les objets du sous-système directement.

3 Le **Sous-système Complexe** est composé de dizaines d'objets variés. Pour leur donner une réelle utilité, vous devez plonger au cœur des détails de l'implémentation du sous-système, comme initialiser les objets dans le bon ordre et leur fournir les données dans le bon format.

Les classes du sous-système ne sont pas conscientes de l'existence de la façade. Elles opèrent et interagissent directement à l'intérieur de leur propre système.

Cas d’utilisation typiques

- Interface simplifiée pour une bibliothèque complexe (ex. JavaFX, JDBC).
- Point d’entrée unique pour un module applicatif.
- API d’un framework ou SDK.
- Couche intermédiaire entre le frontend et le backend (par ex., dans une architecture MVC).



Exemple d’implémentation: Façade dans un système bancaire

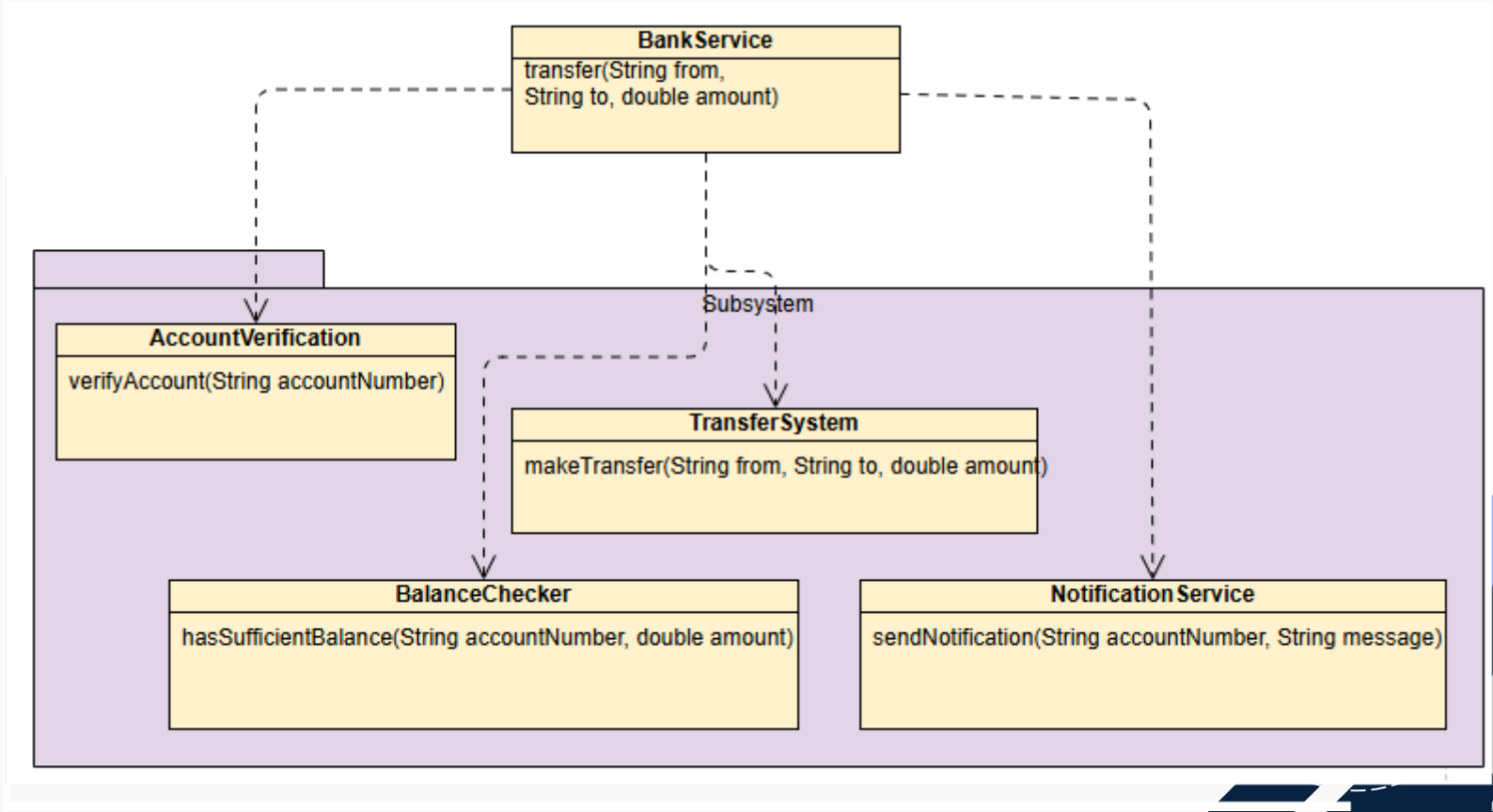
Un client souhaite effectuer un virement bancaire. Mais en réalité, plusieurs sous-systèmes doivent coopérer :

- Vérification du compte
- Vérification du solde
- Système de transfert
- Notification au client

Sans façade, le client devrait interagir avec toutes ces classes.

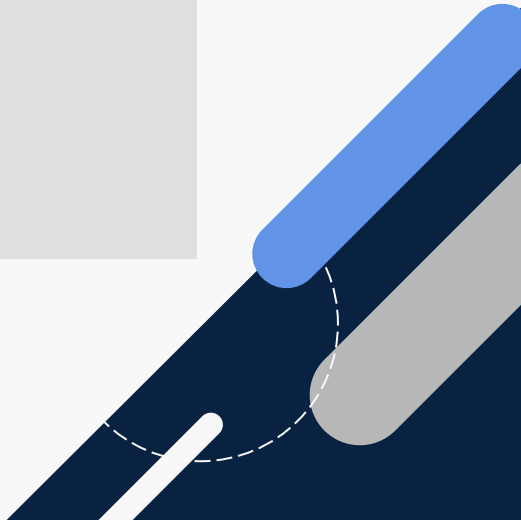
Solution: Créer la Façade BankService avec une seule méthode transfer() qui invoque tous les sous-systèmes.

Architecture du système



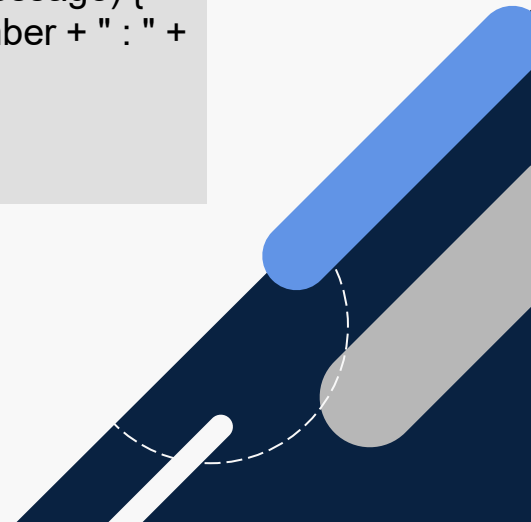
Implémentation des sous-systèmes

```
class AccountVerification {
    public boolean verifyAccount(String accountNumber) {
        System.out.println("Vérification du compte : " + accountNumber);
        return true;
    }
}

class BalanceChecker {
    public boolean hasSufficientBalance(String accountNumber, double
amount) {
        System.out.println("Vérification du solde du compte : " +
accountNumber);
        return amount <= 5000; // limite fictive pour l'exemple
    }
}
```



Implémentation des sous-systèmes

```
class TransferSystem {
    public void makeTransfer(String from, String to, double amount) {
        System.out.println("Transfert de " + amount + "€ de " + from + " vers "
+ to);
    }
}

class NotificationService {
    public void sendNotification(String accountNumber, String message) {
        System.out.println("Notification envoyée à " + accountNumber + " : " +
message);
    }
}
```



Implémentation de la façade

```
class BankService {
// Références aux sous-systèmes
    private AccountVerification verifier = new AccountVerification();
    private BalanceChecker balanceChecker = new BalanceChecker();
    private TransferSystem transferSystem = new TransferSystem();
    private NotificationService notifier = new NotificationService();
// Implémentation de la méthode transfer() qui encapsule l'appel aux sous-systèmes
    public void transfer(String from, String to, double amount) {
        // Appel aux sous-systèmes
        if (verifier.verifyAccount(from) && verifier.verifyAccount(to)) {
            if (balanceChecker.hasSufficientBalance(from, amount)) {
                transferSystem.makeTransfer(from, to, amount);
                notifier.sendNotification(from, "Vous avez transféré " + amount + "€ à " + to);
                notifier.sendNotification(to, "Vous avez reçu " + amount + "€ de " + from);
            } else {
                System.out.println("Solde insuffisant !");
            }
        } else {
            System.out.println("Compte invalide !");
        } } }
```

Implémentation de la façade

```
class BankService {
// Références aux sous-systèmes
    private AccountVerification verifier = new AccountVerification();
    private BalanceChecker balanceChecker = new BalanceChecker();
    private TransferSystem transferSystem = new TransferSystem();
    private NotificationService notifier = new NotificationService();
// Implémentation de la méthode transfer() qui encapsule l'appel aux sous-systèmes
    public void transfer(String from, String to, double amount) {
        // Appel aux sous-systèmes
        if (verifier.verifyAccount(from) && verifier.verifyAccount(to)) {
            if (balanceChecker.hasSufficientBalance(from, amount)) {
                transferSystem.makeTransfer(from, to, amount);
                notifier.sendNotification(from, "Vous avez transféré " + amount + "€ à " + to);
                notifier.sendNotification(to, "Vous avez reçu " + amount + "€ de " + from);
            } else {
                System.out.println("Solde insuffisant !");
            }
        } else {
            System.out.println("Compte invalide !");
        } } }
```

Avantages et inconvénients

Avantages

- Simplifie l'utilisation du système.
- Cache la complexité technique.
- Réduit le couplage entre client et sous-systèmes.
- Facilite la maintenance et l'évolution.

Inconvénients

- Peut devenir une façade "trop grosse" si elle regroupe trop de responsabilités.
- Masque parfois les fonctionnalités avancées disponibles dans les sous-systèmes.

Exercice d’application

Problème :

Une entreprise développe une application de maison intelligente (SmartHome).Elle possède déjà plusieurs sous-systèmes indépendants :

- LightSystem : contrôle des lumières,
- HeatingSystem : contrôle du chauffage,
- SecuritySystem : alarme et caméras,
- MusicSystem : lecture musicale.

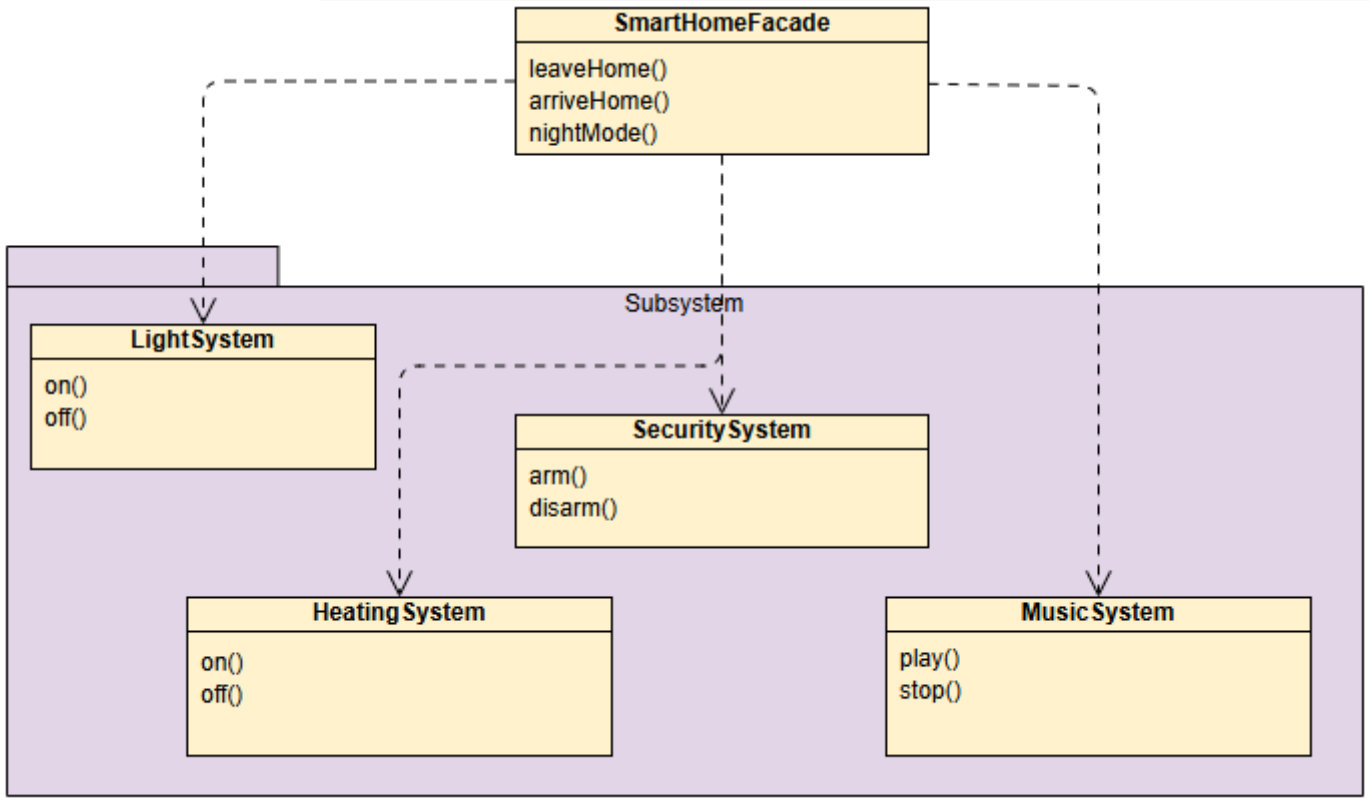
Les utilisateurs trouvent le système trop compliqué, car ils doivent appeler plusieurs classes pour exécuter une seule action.

Exercice d’application

Questions

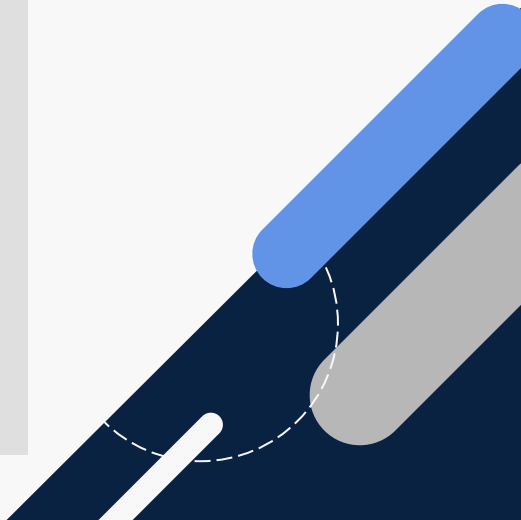
Implémenter la façade SmartHomeFacade avec les méthodes suivantes :

- **leaveHome()** → prépare la maison quand on sort.
- **arriveHome()** → prépare la maison quand on rentre.
- **nightMode()** → active le mode nuit.


SmartHomeFacade
leaveHome()
arriveHome()
nightMode()
Subsystem
LightSystem
on()
off()
SecuritySystem
arm()
disarm()
HeatingSystem
on()
off()
MusicSystem
play()
stop()

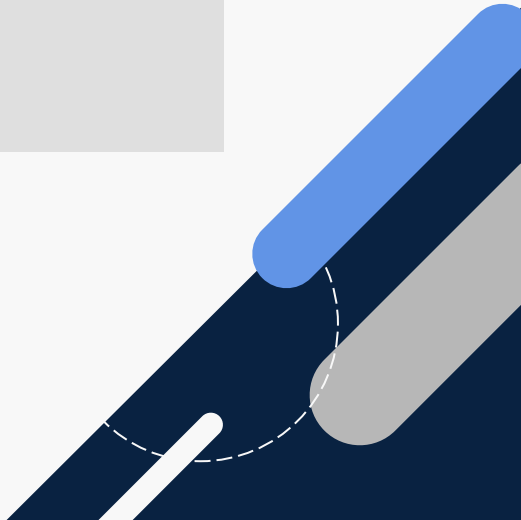
Implémentation des sous-systèmes:

```
class LightSystem {
    public void on() { System.out.println("Lumières allumées"); }
    public void off() { System.out.println("Lumières éteintes"); }
}

class HeatingSystem {
    public void on() { System.out.println("Chauffage activé"); }
    public void off() { System.out.println("Chauffage éteint"); }
}


class SecuritySystem {
    public void arm() { System.out.println("Système de sécurité activé"); }
    public void disarm() { System.out.println("Système de sécurité désactivé"); }
}

class MusicSystem {
    public void play() { System.out.println("Musique lancée"); }
    public void stop() { System.out.println("Musique arrêtée"); }
}
```

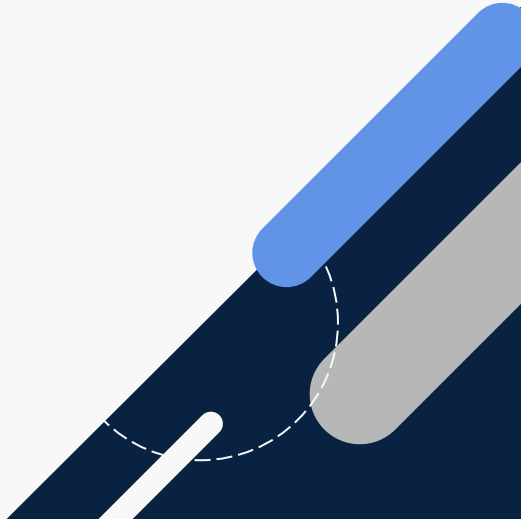


Programme Client :

```
public class Main {
    public static void main(String[] args) {
        SmartHomeFacade home = new SmartHomeFacade();

        home.arriveHome();
        home.nightMode();
        home.leaveHome();
    }
}
```



Pattern Composite



Objectif

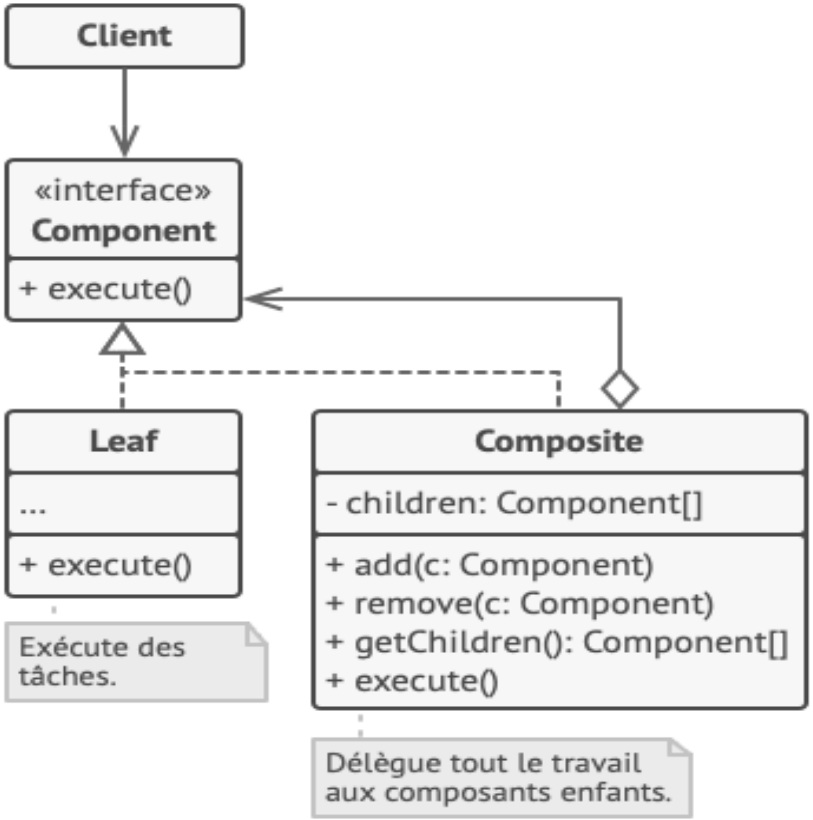
- Permettre de composer des objets en structures hiérarchiques (arborescentes)et de traiter uniformément les objets simples et les objets composés.

Exemple :

- Un dossier contient des fichiers ou d'autres dossiers. On veut pouvoir **afficher**, **supprimer** ou **déplacer** un dossier exactement comme un fichier, sans se soucier s'il contient d'autres éléments.

➡ Le pattern Composite rend cela possible en traitant objets simples et composites de façon uniforme.

Structure UML



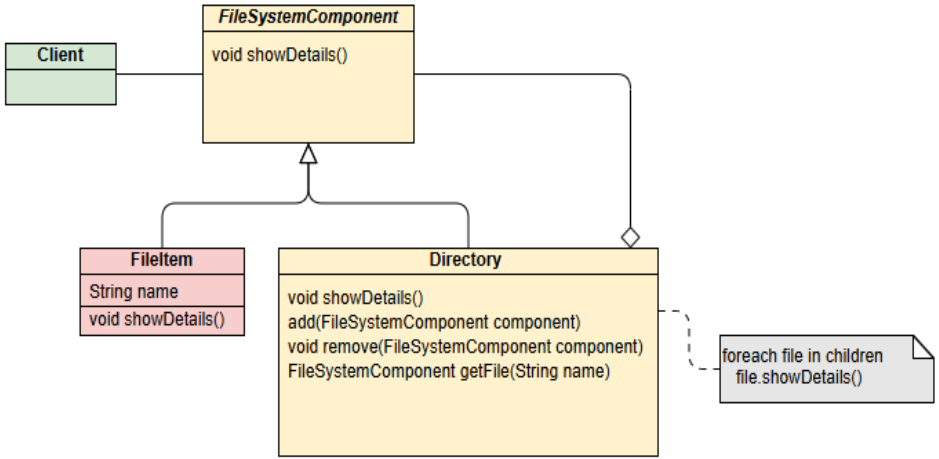
Élément	Rôle
Component	Interface commune (déclare les opérations communes)
Leaf	Élément de base (objet simple, sans enfants)
Composite	Objet composé (contient d'autres Components)
Client	Utilise les objets via l'interface Component, sans savoir s'il s'agit d'un Leaf ou d'un Composite

Exemple d’implémentation: Système de fichiers

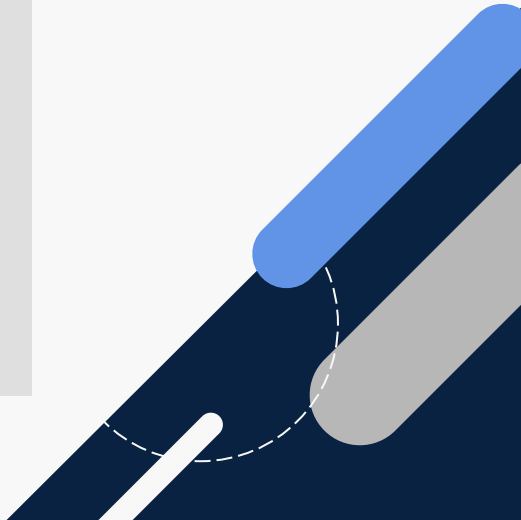
On modélise des dossiers et des fichiers :

- Un fichier est une Leaf.
- Un dossier est un Composite (il contient d’autres fichiers ou dossiers).

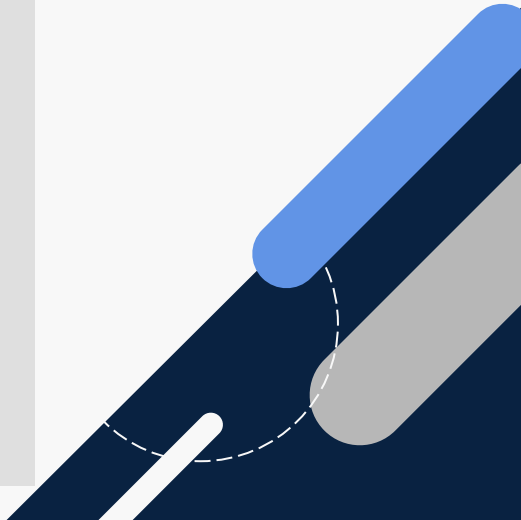
Le client manipule les deux avec la même méthode showDetails().



```
import java.util.ArrayList;
import java.util.List;

interface FileSystemComponent {
    void showDetails();
}

class FileItem implements FileSystemComponent {
    private String name;

    public FileItem(String name) {
        this.name = name;
    }

    @Override
    public void showDetails() {
        System.out.println("  Fichier : " + name);
    }
}
```



```
class Directory implements FileSystemComponent {
    private String name;
    private List<FileSystemComponent> children = new ArrayList<>();

    public Directory(String name) {
        this.name = name;
    }

    public void add(FileSystemComponent component) {
        children.add(component);
    }

    public void remove(FileSystemComponent component) {
        children.remove(component);
    }

    @Override
    public void showDetails() {
        System.out.println(«  Dossier : " + name);
        for (FileSystemComponent child : children) {
            child.showDetails();
        }     } }
```



Avantages et inconvénients

Avantages

- Simplifie le code client (traitement uniforme).
- Permet de créer des structures hiérarchiques dynamiques.
- Facilite l'ajout de nouveaux types d'éléments (principe Open/Closed).

Inconvénients

- Peut rendre le système trop général (le client peut manipuler un fichier comme un dossier sans distinction).

Exercice d’application

On veut modéliser un menu hiérarchique pour un restaurant. Chaque élément de menu doit pouvoir être :

soit un Plat (élément simple : nom + prix),

soit un Menu (élément composé contenant d’autres plats ou sous-menus).

Travail demandé :

1. Créez une interface MenuComponent avec une méthode showDetails().

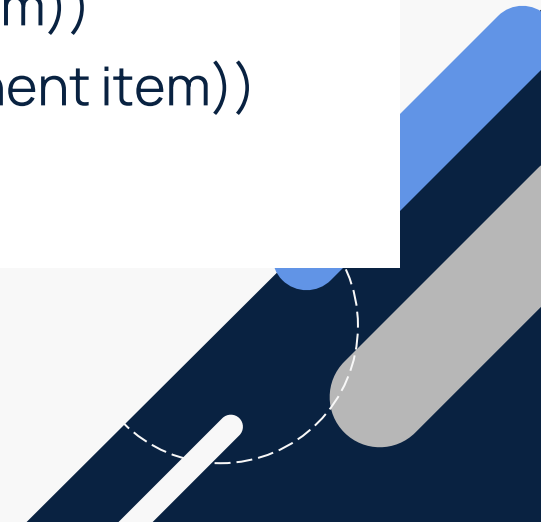
Exercice d’application

2. Créez deux classes :

- Dish (Feuille)
- Menu (Composite)

Un objet Menu doit pouvoir :

- ajouter un élément (add(MenuComponent item))
- supprimer un élément (remove(MenuComponent item))
- afficher tout son contenu récursivement.



Exercice d’application

3. Dans la classe principale, créez la hiérarchie suivante :

Menu principal : "Menu Restaurant"

- Menu "Entrées"
  - Plat : "Soupe du jour" - 5.5€
  - Plat : "Salade verte" - 4.0€
- Menu "Plats principaux"
  - Plat : "Steak frites" - 12.0€
  - Plat : "Pâtes carbonara" - 10.0€
- Menu "Desserts"
  - Plat : "Assiette fruits de saison" - 6.0€