Fiche de Résumé : Design Pattern Composite

1. Intention du Pattern

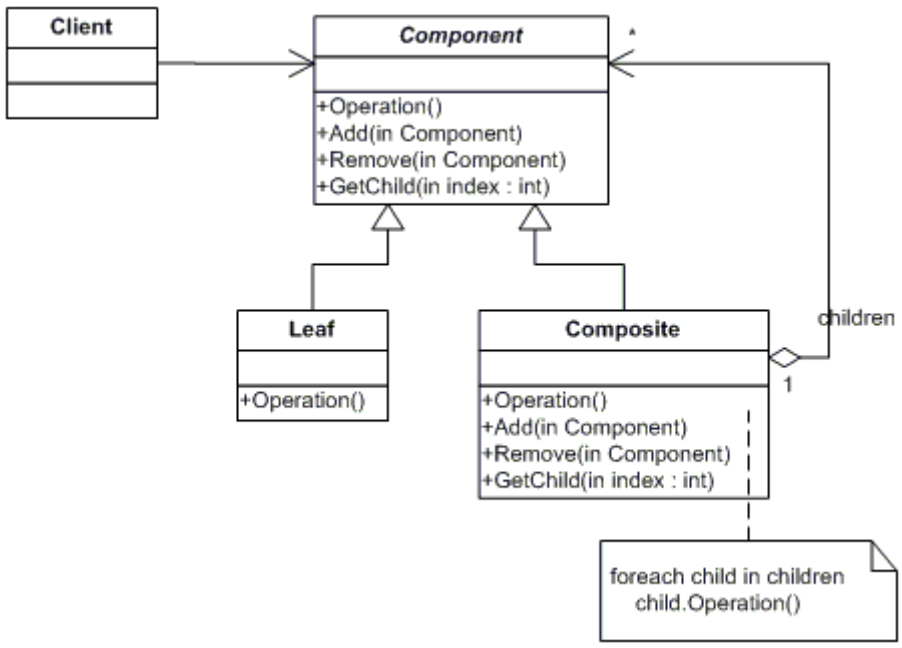
Nom du Pattern	Catégorie GoF	Alias
Composite	**Structurel**	Arbre d'objets, Composition

Le pattern **Composite** a pour intention de composer des objets en structures arborescentes/hiérarchiques de manière à ce que les clients puissent traiter uniformément les objets individuels (*Feuilles*) et les compositions d'objets (*Conteneurs*). Son objectif est de simplifier le code client en éliminant la nécessité de distinguer les objets simples des groupes d'objets complexes.

2. Diagramme de Classes Générique

Le pattern est défini par trois rôles principaux qui partagent une interface commune.

Diagramme UML :



3. Explication des Classes Participantes

Component (Composant) :
C'est l'interface (ou la classe abstraite) commune. Elle déclare les opérations que tous les objets de la composition doivent implémenter. Elle peut également définir des méthodes par défaut pour gérer les enfants.

Leaf (Feuille) :
Représente les objets simples de la composition. Il s'agit des éléments de base qui ne peuvent pas avoir d'enfants. Ils implémentent directement les opérations définies par `Component`.

Composite (Conteneur) :
Représente les objets complexes qui contiennent des sous-éléments. Il implémente l'interface `Component` et stocke une collection d'objets `Component`. Il délègue les requêtes à ses enfants de manière récursive.

Client :
Manipule les objets `Leaf` et `Composite` via l'interface `Component` de manière uniforme.

4. Exemple de Contexte Métier

Contexte	Rôle de chaque classe
Gestion de Personnel d'Entreprise	Le problème est de calculer le salaire total d'un département, sachant qu'un département peut contenir des employés et d'autres sous-départements.
Component	Interface `IPersonnel` avec la méthode `getSalary()`.
Leaf	Classe `Employee` (l'employé simple).
Composite	Classe `Department` (qui agrège plusieurs objets `IPersonnel`).

Mise en œuvre : Lorsqu'on appelle `getSalary()` sur un `Department`, il parcourt récursivement tous les objets qu'il contient (employés et sous-départements) et additionne leurs salaires pour retourner le total.

5. Solidité du Pattern (Liens avec les Principes SOLID)

Le pattern Composite est un modèle d'application des principes SOLID, il sert à améliorer la maintenabilité et la flexibilité du code.

Principe Ouvert/Fermé (Open/Closed Principle - OCP) : Le pattern est ouvert à l'extension (on peut ajouter de nouveaux types de `Leaf` ou de `Composite`) mais fermé à la modification (le code client qui utilise l'interface `Component` n'a pas besoin d'être modifié).

Principe de Substitution de Liskov (Liskov Substitution Principle - LSP) : Les objets `Leaf` et `Composite` peuvent être substitués l'un à l'autre sans casser le code client, car ils implémentent tous deux l'interface `Component`. Le code client ne fait aucune distinction entre eux.

Principe d'Inversion des Dépendances (Dependency Inversion Principle - DIP) : Le pattern garantit que la classe `Composite` (le module de haut niveau) dépend de l'abstraction `Component` et non des classes concrètes `Leaf`. Cela rend la structure plus flexible.

6. Limites du Pattern

Malgré ses avantages, le pattern Composite présente quelques inconvénients et compromis :

Complexité de l'interface (Compromis ISP) : Il peut être difficile de concevoir une interface `Component` qui soit à la fois simple et pertinente pour toutes les `Leaf` et `Composite`, surtout si les feuilles ont des fonctions très différentes.

Sécurité et Transparence : Pour permettre la transparence (uniformité maximale), l'interface `Component` doit parfois inclure des méthodes de gestion des enfants (`add`, `remove`). Si elle le fait, elle enfreint le Principe de Ségrégation des Interfaces (ISP), car les objets `Leaf` devront implémenter ces méthodes inutilement.

7. Conclusion

Le pattern Composite est le modèle structurel idéal pour la gestion des structures arborescentes. Il résout efficacement le problème de la complexité des structures conditionnelles (`if/else`) dans le code client en introduisant une abstraction qui permet de traiter les parties comme des touts. Son application mène directement à des architectures plus flexibles, maintenables et conformes aux principes SOLID.