

Pattern Décorateur

Betton Yaël/Enon Loïs/
Français Nina

R3.04



Sommaire



INTRODUCTION GÉNÉRALE	03
ÉNONCÉ DU BESOIN	04
PREMIÈRE MODÉLISATION	05
IDENTIFICATION DES PROBLÈMES	05
MODÉLISATION POSSIBLE	06
PROBLÉMATIQUE DU PATTERN	07
SOLUTION DU PATTERN	08
LIEN AVEC SOLID	10
LIMITE DU PATTERN	11
PATTERNS APPARENTÉS	12
LIVE CODING	15
QCM	16
BIBLIOGRAPHIE / WEBOGRAPHIE	17

Introduction

AUX DESIGN PATTERNS (GOF)

- Les design patterns sont des solutions éprouvées à des problèmes récurrents rencontrés lors du développement logiciel.
- Les patterns du GoF (Gang of Four) sont classés en trois catégories :
 - Création
 - Structure
 - Comportement
- L'utilisation des patterns permet un code plus lisible, maintenable et extensible.





Énoncé du Besoin

1 DANS UNE APPLICATION DE GESTION DE RESTAURATION RAPIDE, IL EST NÉCESSAIRE DE MODÉLISER UN SANDWICH ET SES INGRÉDIENTS POUR POUVOIR CONSTRUIRE, PERSONNALISER ET FACTURER DES COMMANDES.

2 **LA SOLUTION DOIT PERMETTRE DE :**

- DÉFINIR UN SANDWICH AVEC UN NOM ET UN PRIX DE BASE.
- AJOUTER DYNAMIQUEMENT DIFFÉRENTS INGRÉDIENTS (SALADE, TOMATE, FROMAGE, BEURRE, JAMBON...) À UN SANDWICH EXISTANT.
- CONSULTER À TOUT MOMENT LE NOM DU SANDWICH ET SON PRIX TOTAL.
- MODIFIER FACILEMENT LES CARACTÉRISTIQUES D'UN SANDWICH SANS AVOIR À CRÉER UNE SOUS-CLASSE PAR COMBINAISON D'INGRÉDIENTS.



Sandwich

```
- hasLettuce : boolean  
- hasTomato : boolean  
- hasCheese : boolean  
- hasButter : boolean  
- hasHam : boolean  
- name: String  
- price: Integer
```

```
+ addLettuce(): void  
+ addTomato(): void  
+ addCheese(): void  
+ addButter(): void  
+ addHam(): void  
+ getName(): String  
+ getPrice(): Integer  
+ setName(): void  
+ setPrice(): void
```

MODÉLISATION POSSIBLE

Classe trop rigide

- Tous les ingrédients (lettuce, tomato, cheese) sont directement dans la classe Sandwich.
- Pour ajouter un nouvel ingrédient, il faut modifier la classe → pas extensible.

Pas de vraie extensibilité

- Chaque ingrédient a sa propre méthode (addLettuce, addTomato, etc.).
- Ça devient vite ingérable si on veut ajouter plein d'ingrédients.

Responsabilités mal séparées

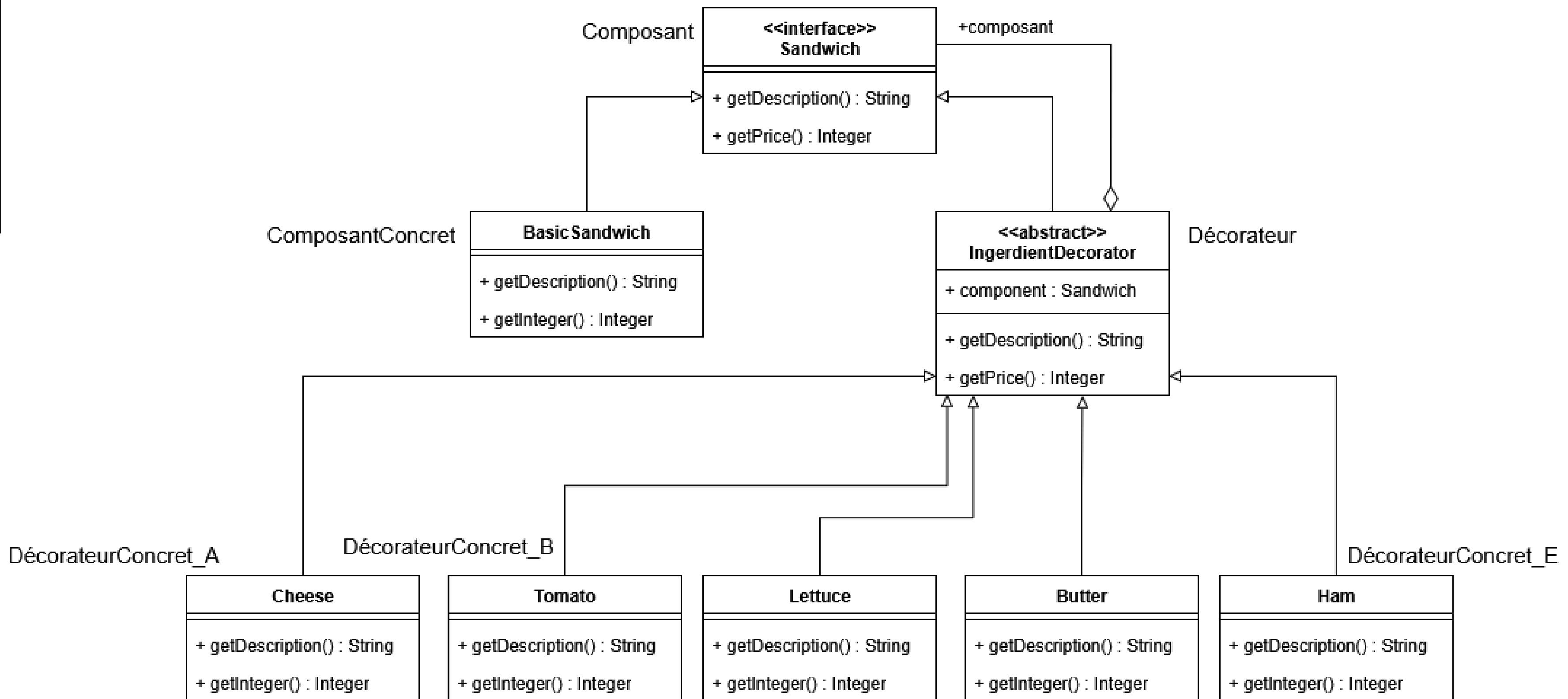
- Sandwich gère à la fois le sandwich de base et tous les ingrédients.
- Mélange des rôles → difficile à maintenir.

Pas de décorateur

- Le pattern Décorateur consiste à envelopper un objet pour ajouter dynamiquement des fonctionnalités.
- Ici, on modifie directement la classe au lieu de créer des décorateurs.

MODÉLISATION POSSIBLE

GÉNÉRALISATION DE CAS



Pattern Décorateur

- Nom du pattern : Décorateur (Decorator)
- Type de pattern : Pattern Structurel (structural pattern)

Problématique

- Comment ajouter dynamiquement des responsabilités (fonctionnalités, comportements, attributs) à un objet, sans modifier sa classe ?
- Comment éviter d'avoir une classe avec des dizaines de variantes (via héritage ou attributs booléens), ce qui entraîne une explosion de sous-classes ?

Intention (définition)

- Le Décorateur attache dynamiquement des responsabilités supplémentaires à un objet.

Il fournit une alternative flexible à l'héritage pour étendre les fonctionnalités.

✓ **Flexible**

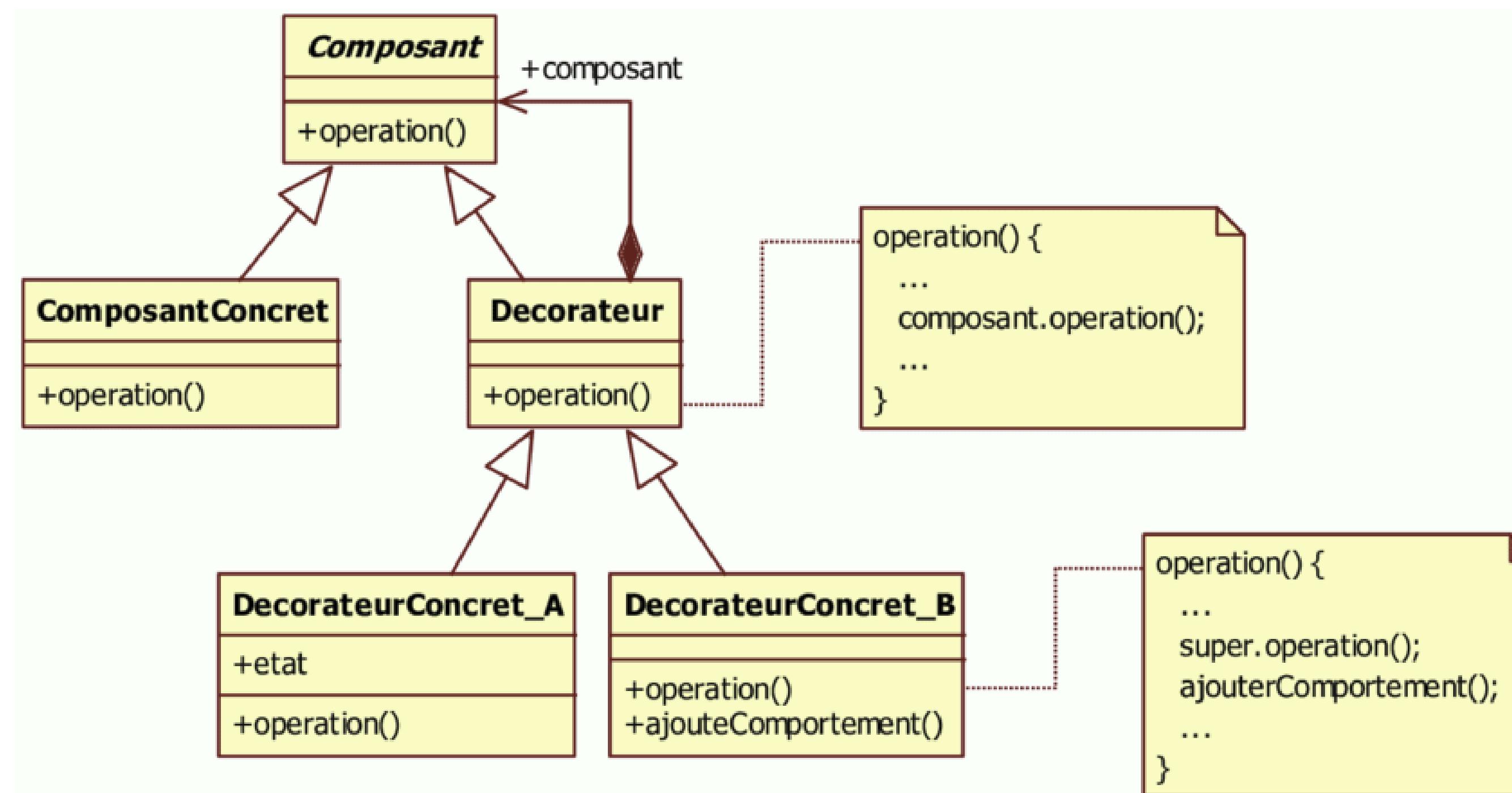
✓ **Fonctionnalités combinables**

✗ **Chaîne difficile à lire**

Principe : un objet est "enveloppé" par un décorateur qui implémente la même interface.

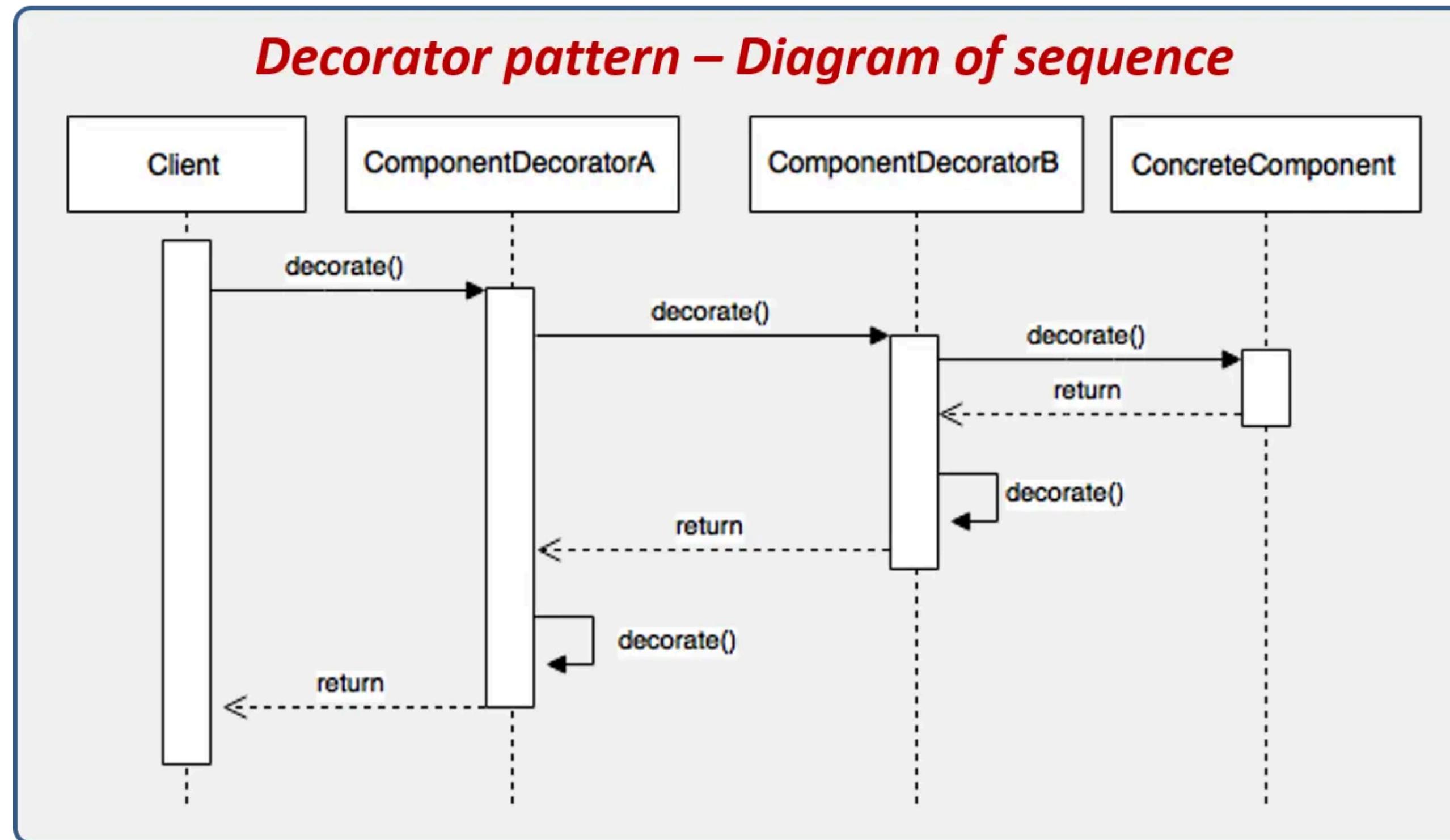
Solution du Pattern Décorateur

Diagramme de classes



Solution du Pattern Décorateur

Diagramme de séquence



Lien avec SOLID

S

SINGLE RESPONSIBILITY PRINCIPLE (SRP)

- Chaque classe a une responsabilité unique :
- BasicSandwich = sandwich de base.
- Cheese, Tomato, etc. = ajout d'un ingrédient.
- ➔ Pas de classe avec 50 booléens.

O

OPEN/CLOSED PRINCIPLE (OCP)

- On peut ajouter de nouveaux comportements (nouveaux décorateurs) sans modifier le code existant.
- Exemple : ajouter Bacon → on écrit une nouvelle classe, pas besoin de toucher à BasicSandwich.

L

LISKOV SUBSTITUTION PRINCIPLE (LSP)

- Tout décorateur reste un Sandwich (même interface).
- Le client peut utiliser un Cheese ou un Tomato comme un Sandwich normal.

I

INTERFACE SEGREGATION PRINCIPLE (ISP)

- Le contrat Sandwich est simple et cohérent (getDescription(), getPrice()), pas d'interface massive imposant des méthodes inutiles.

D

DEPENDENCY INVERSION PRINCIPLE (DIP)

- Les décorateurs dépendent de l'abstraction (Sandwich), pas d'une classe concrète.
- Le client manipule uniquement des Sandwich, sans connaître leur "décoration".

Limites du pattern Décorateur



- 1 MULTIPLICATION DES CLASSES**
 - Chaque fonctionnalité additionnelle = un nouveau décorateur → peut vite créer beaucoup de petites classes.
- 2 COMPLEXITÉ DE LA LECTURE**
 - Empilement de décorateurs peut rendre difficile la compréhension de l'objet final et de son comportement.
- 3 DIFFICULTÉS DE DÉBOGAGE**
 - La chaîne de délégation peut compliquer le suivi du flux d'exécution.
- 4 PAS IDÉAL POUR CERTAINES EXTENSIONS**
 - Si les ajouts sont simples et peu nombreux, l'héritage ou des attributs peuvent parfois suffire.

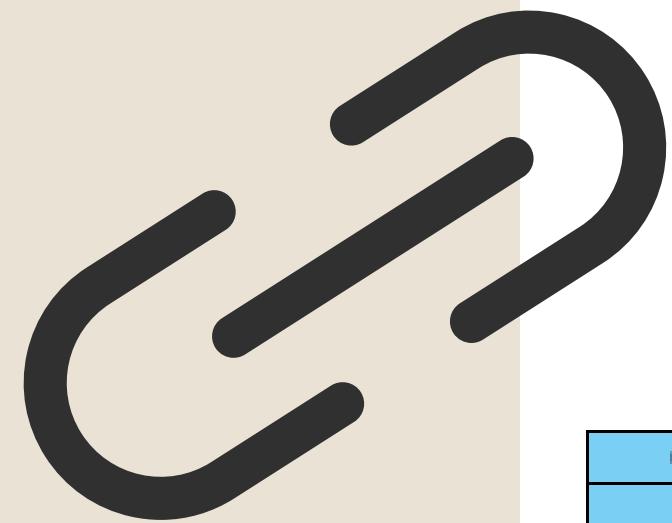




Décorateur : patterns apparentés

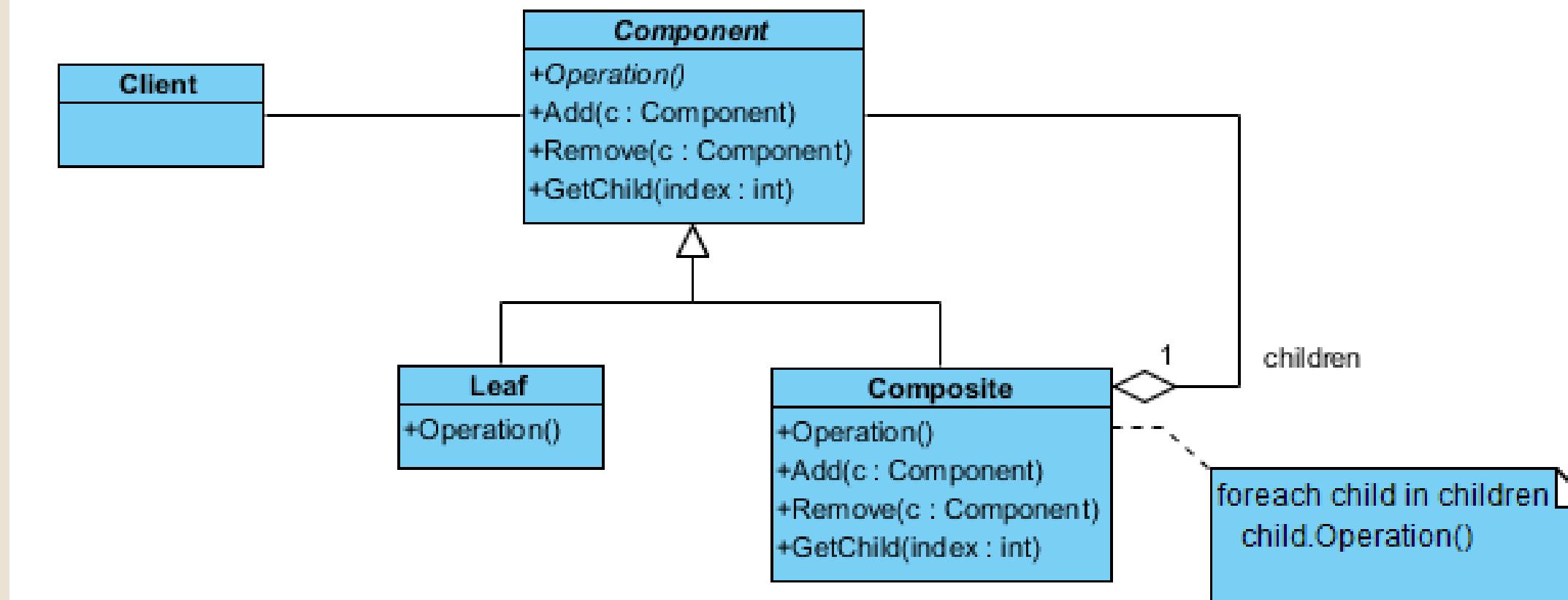
Le pattern Décorateur est souvent rapproché de :

- Composite :
 - Les deux manipulent des objets via une interface commune.
 - Dans Composite, on organise des objets en hiérarchie (arbre) pour traiter de façon uniforme des objets simples et composés.
 - Dans Décorateur, on enchaîne des objets pour ajouter dynamiquement des responsabilités.
 - Similitude : on a une structure récursive (chaînage ou arbre) et on délègue à l'objet contenu.
- Adapter :
 - Décorateur et Adapter enveloppent tous deux un autre objet.
 - Différence : Adapter change l'interface pour qu'elle corresponde à une autre, alors que Décorateur conserve l'interface et ajoute des comportements.

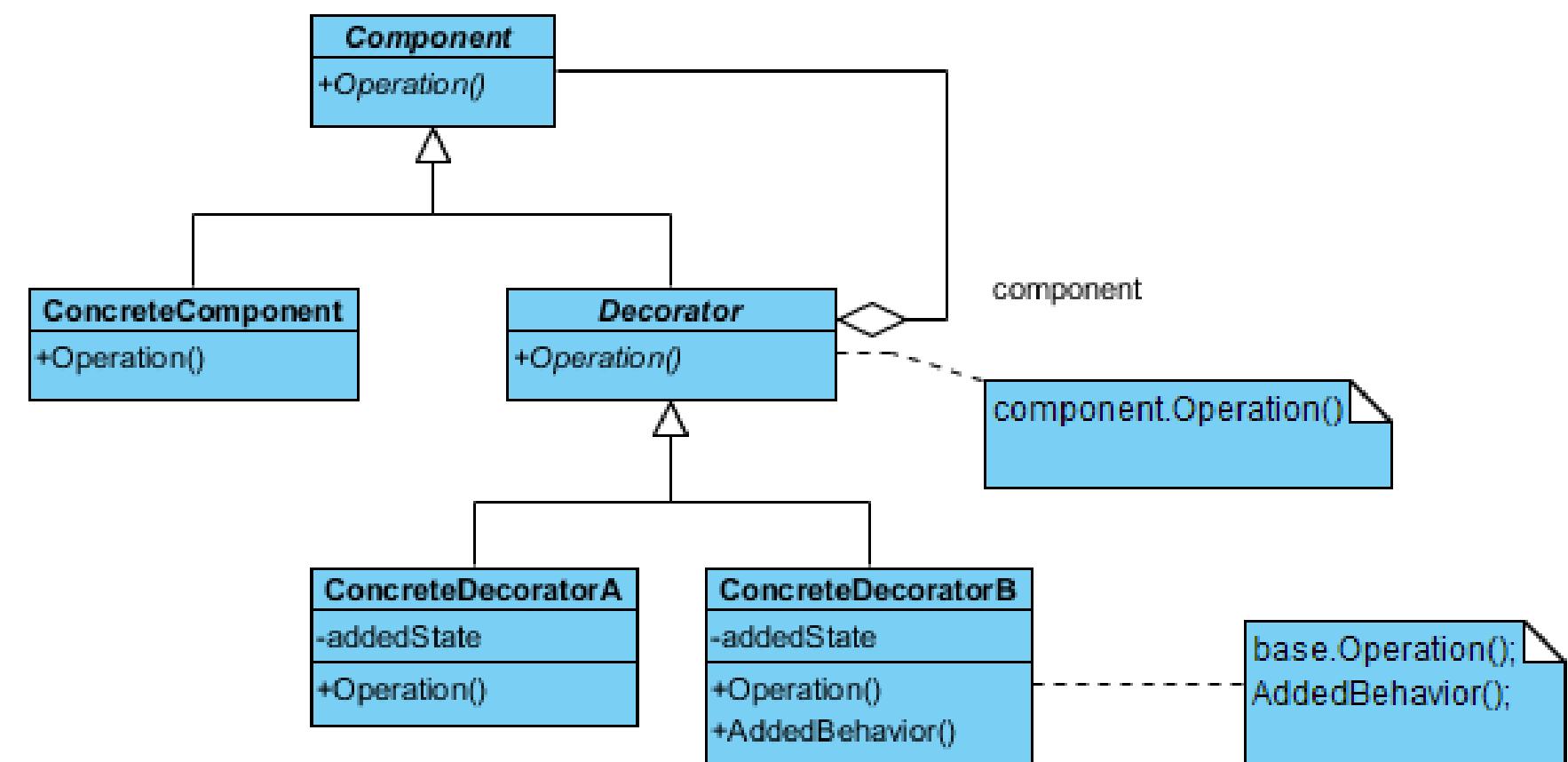


Composite vs Décorateur

Composite



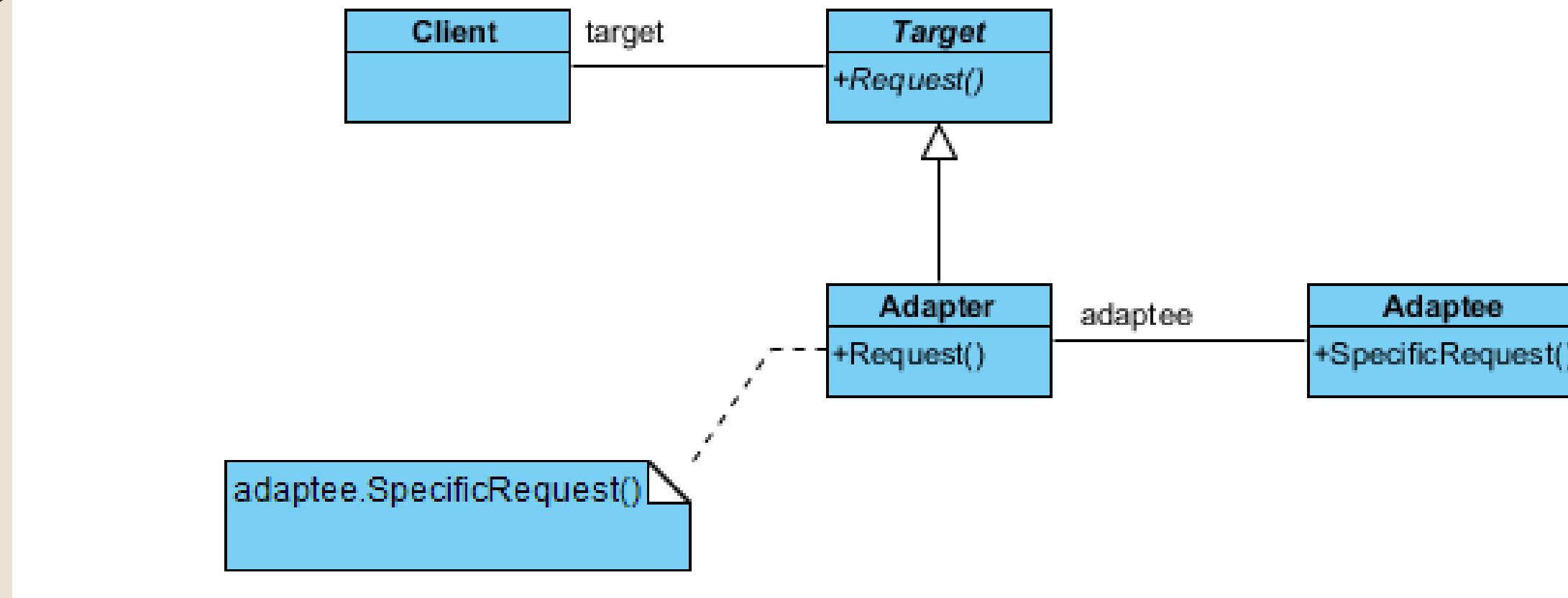
Decorateur



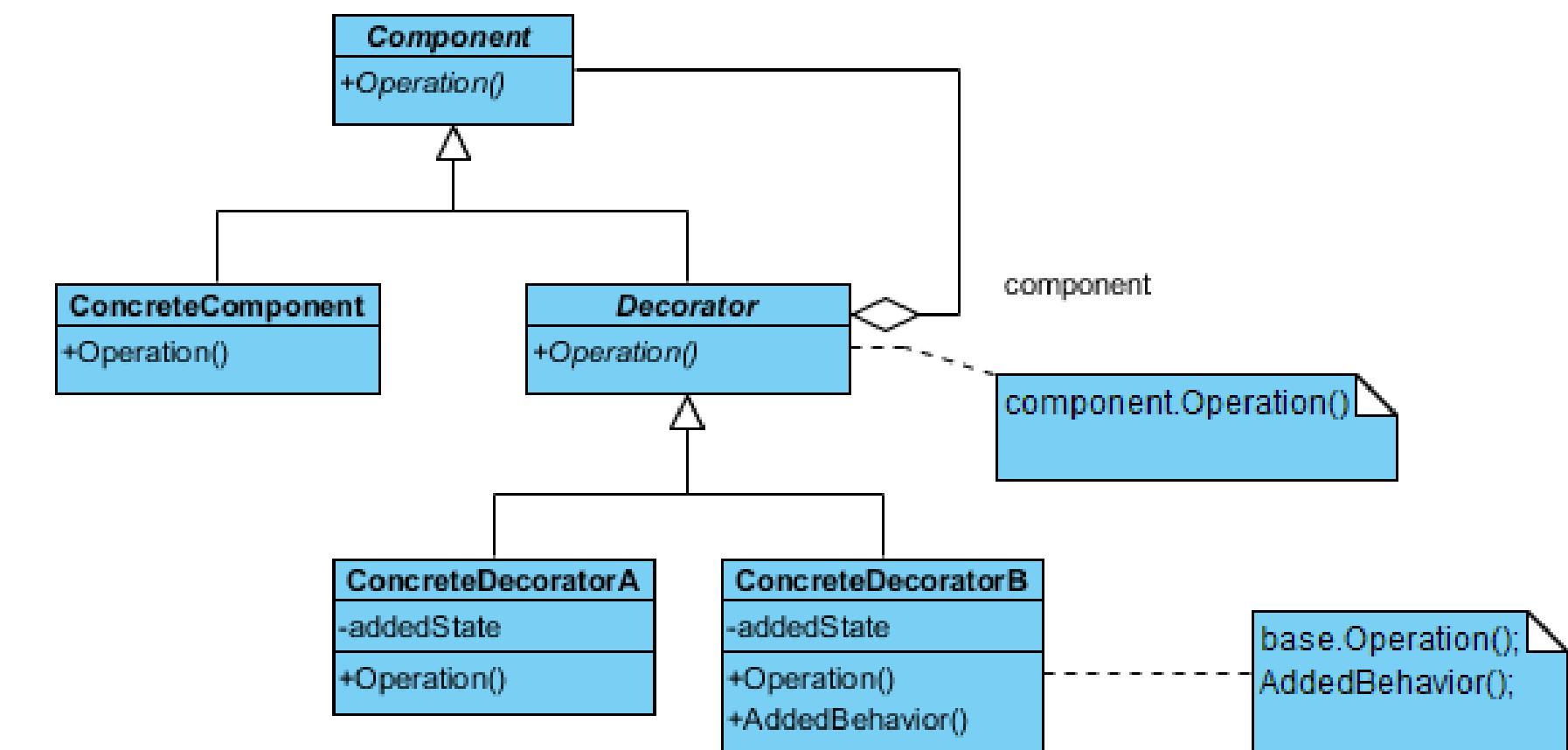


Adapter vs Décorateur

Composite



Decorateur

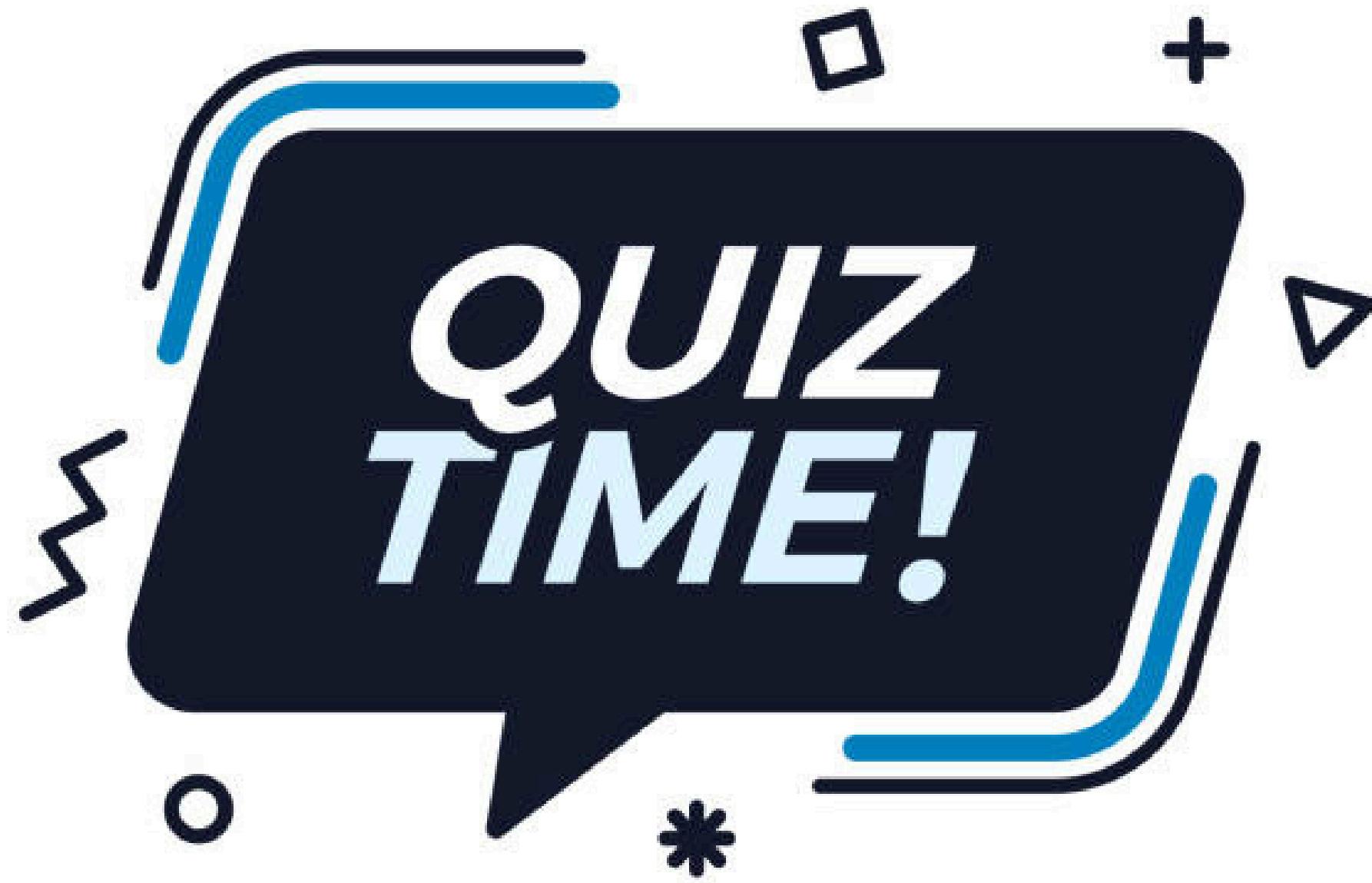


Live Coding

[https://www.youtube.com/watch?
v=BDF1Z5FMNA0](https://www.youtube.com/watch?v=BDF1Z5FMNA0)



QCM



Bibliographie

- https://sourcemaking.com/design_patterns
- <https://dzone.com/articles/gof-design-patterns-using-java-part-1>
- https://fr.wikipedia.org/wiki/Patron_de_conception
- https://www.youtube.com/watch?v=WghYGmUd9nQ&list=PLAFXqCQG3IKX2_GBt1xVu5qz1dXrd_wtN
- <https://www.elao.com/blog/tag/design-pattern/>
- https://www.lirmm.fr/~pvalicov//Cours/M3105/DP_x4.pdf