R3.04 - Pattern Décorateur

Introduction

Petite intro sur la première diapo de qui on est et sur quoi on a bossé

Qu'est-ce qu'un Design Pattern ?

En programmation orientée objet, une conception pouvant paraître tout à fait fonctionnelle peut rencontrer des problèmes techniques (ex : utilisation de classes). Pour remédier à ces problèmes se répétant régulièrement, des Design Patterns ont été pensés. Un Design Pattern est, comme son nom l'indique, un **patron de conception**, il est donc général (parallèle avec la couture où on suit un patron pour faire un vêtement). Il permet de gagner du temps et de simplifier nos conceptions selon les problèmes rencontrés.

Énoncé de notre besoin 1

Nous tenons une concession de tracteur, notre but est de vendre le plus de tracteur possible mais chaque véhicule est différent. Certains ont des options d'autre non donc nous avons besoin de nos options comme la clim ou encore un autoradio ainsi que de notre véhicule de base. À l'avenir rajouter d'autres options sur notre véhicule mais comment faire pour que ce soit simple et rapide.

Proposition de modélisation du besoin

On a fait une classe abstraite Tracteur où nous avons mis à l'intérieur le nom et la marque ainsi que le prix. Puis on a créé notre tracteur sans option mais si on voulait en faire un autre avec la clim, il aurait fallu faire une autre classe pour notre nouveau tracteur.

Problème de notre modélisation

Notre code fonctionne très bien sur cette modélisation mais malheureusement on se retrouverait vite avec 40 classes pour une option en plus ou d'autre option dessus. Notre diagramme de classe en serait énorme et incompréhensible pour celui qui passerait derrière nous pour modifier le code. Pour nous également après avoir laissé notre code sans y avoir touché pendant plusieurs semaines. Et si on veut modifier le prix d'une option, il faut modifier tous les prix de tous les tracteurs qui bénéficient de cette option. Donc notre code manque cruellement de flexibilité. Pour le rendre moins statique on va utiliser un design pattern nommé le pattern decorator qui va nous permettre de la rendre plus dynamique et donc de pouvoir y apporter des modifications sans avoir à réécrire tout l'ancien code.

Nouvelle modélisation

On a tout en haut une interface où l'on va créer nos méthodes que nous instancierons dans nos autres classes en dessous on a nos classes abstraite ici on en a 2 car on a plusieurs méthodes à instancier donc plutôt que de surcharger

une seule classe on va en créer 2 petite rapport avec le solid que l'on va parler après et enfin on as nos classes concrète en dessous qui ont une relation d'héritage avec nos classe abstraite car la par exemple notre clim est une option du tracteur mais pour qu'il valide le solid on pourrait dire que le tracteur avec option a une clim en option donc mettre une relation d'agrégation plutôt que d'héritage.

Généralisation du diagramme de classes
Voici le diagramme de classe généraliste qui montre à peut prêt la création de notre design pattern nous détaillerons plus en détails chaque case à quoi elles correspondent mais globalement de base on a 4 classes la première en haut est notre interface c est le niveau haut, ensuite en dessous notre classe abstraite et notre classe concrète qui est notre objet décoré et enfin en dernier nous avons toute nos classes concrètes dont celles qui auront une relation d'héritage avec notre classe abstraite. Ici il y en a qu'une mais il peut y en avoir plusieurs autres et c'est le même cas pour l'interface (il peut y en avoir plusieurs)

Le pattern Decorator

Un décorateur à plusieurs significations et oui c'est la langue française 1 mot pour signifier 40 choses c'est magnifique non ?, pour plein de personne pas trop calé en informatique le décorateur est quelqu'un qui te refait toute la maison pour une somme astronomique mais pour nous autres informaticiens le pattern décorateur est le nom d'une structure de conception utilisée dans plusieurs type de langage de programmation qui va nous permettre de refactor ton code et le rendre plus agréable pour les autres qui passeront après toi, il en existe en tout 23 des design pattern tous expliqué dans dans le livre du GOF, le gang of four de 1993 qui sont répartis en 3 catégorie, le patrons de création, celui de structure et celui de comportement en l'occurrence le nôtre fait partie du patrons de structure avec 4 autres design pattern.

Utilité du pattern décorateur
Notre pattern est très utile dans la vie du codeur de tous les jours. Il permet notamment de pouvoir modifier un code dynamiquement, il nous permet d'ajouter des fonctionnalités à l'exécution du code, et il est plus flexible que l'héritage. Il simplifie le code car on y ajoute des fonctionnalités en utilisant des classes simples. Et enfin on peut ajouter des extensions à notre code sans avoir à le réécrire entièrement donc notre ancien code est conservé.

Solution du pattern décorateur

"Attach additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality".

qui veut dire

« Attachez dynamiquement des responsabilités supplémentaires à un objet. Le décorateur offre une alternative flexible à la sous-classification pour étendre les fonctionnalités. »

Dans une conception qui n'utilise pas le pattern Decorator, on peut rapidement se retrouver avec un grand nombre de classes. Devoir modifier ces classes qui sont statiques peut se retrouver très long. C'est pourquoi, avec une classe decorator, qui sera liée à l'objet à décorer (une sorte de base), on pourra "parfaire" notre objet final grâce à des sous-classes. C'est à dire qu'en appelant notre décorateur, on pourra compléter notre objet de base grâce aux sous-classes. C'est pourquoi on dit que l'on dit que c'est dynamique.

Détail du rôle des classes

Tout en haut on a le component qui est l'interface qui sert à implémenter à la fois l'objet décoré et les objets qui le décorent

Classe decorator qui est une classe abstraite de base qui va permettre d'implémenter un décorateur qui va rappeler toute les méthodes de base de la classe abstraite

Ensuite en dessous on a plusieurs classes concrètes qui vont juste ré-implémenter certaines méthodes ce qui permet d'avoir des classes qui vont avoir de l'ajout de fonctionnalités ou d'appliquer des transformations sur certaines méthodes

Enfin concrètement Component est l'objet décoré de notre code dans notre exemple précédent c est le tracteur

Le principe SOLID

SRP qui est single responsability principle donc principe de responsabilité unique veut dire que nous devons plutôt que tout mettre sur une seul classe créer de nouvelle classe pour pas trop surcharger ce que fait notre pattern design qui va préférer avoir de nouvelle classe plutôt que créer des sous-classes surchargé

OCP qui est open-closed principle donc principe d'ouverture et fermeture nous stipule qu'il ne faut pas faire des modification mais plutôt y ajouter des extensions un fois de plus ce que fait notre pattern parfaitement car ce dernier garde l'ancien code et y rajoute juste des nouvelle méthodes pour le rendre plus facile à comprendre et éviter de surcharger une nouvelle fois

LSP, le Liskov substitution principle ou principe de substitution de Liskov nous sert dans le solid a vérifier que notre code n'a pas de méthodes donnée par l'héritage qui sont inutiles on peut le vérifier sur notre pattern que le principe est

validé grâce au fait que si on échange 2 classe concrète entre elle, le code fonctionne toujours

ISP interface segregation principle, le principe de séparation des interface le but de ce principe est de créer de nouvelle interface pour éviter d' en surcharger 1 seule, notre pattern le remplis même si sur notre exemple nous avons qu une seule interface si nous avions besoin de plus de méthodes nous créerons une nouvelle interface qui récupère ces méthodes pour les instancier autre part

et enfin le DIP , dependency inversion principle ou en français principe d'inversion des dépendances, ce principe pour le respecter nous allons créer une interface et une classe abstraite pour découplée nos dépendances ce que nous faisons dans notre pattern car on crée une interface dans l exemple d avant le véhicule et une classe abstraite pour pouvoir découplée nos méthodes

Les limites du Pattern

Notre pattern a beau être simple au premier abord il y a quand même quelques limites à son utilisation, le premier étant que si on est un débutant dans le codage ce n'est pas forcément à notre portée. Car il faut quand même avoir quelques bases sur le sujet. La deuxième est qu'au bout d'un moment, à force d'ajouter de nouvelles choses, on va se retrouver avec un diagramme de classe gigantesque. Comme on vous en a parlé au début et par conséquent pour éviter ça, on va devoir passer par d'autres patterns comme le pattern Builder qui vous sera présenté plus tard. Et enfin le code est plus facile a comprendre, mais plus dur a construire dans sa globalité car il y a un risque de s'y perdre avec trop d'informations dessus.

Rapprochement avec d'autre pattern

Facultatif juste on introduit les autres pattern

Classe javadoc

De ce que nous avons pu trouver notre design pattern se rapproche essentiellement des code qui utilise le concept de flux donc ce qui est en rapport avec la commande stream

Énoncé de notre besoin 2

Nous sommes un restaurant de kebab, c'est pourquoi nous voulons faire des kebabs. Pour commencer, nous voulons commercialiser le très célèbre kebab salade Tomate Oignon. Nous utiliserons donc du pain pita, et trois ingrédients différents. À l'avenir, nous voudrions ajouter des ingrédients ou d'autres types de kebabs. Alors comment ajouter simplement des kebabs à notre carte ?

Proposition de modélisation du besoin
Tout d'abord nous aurons notre classe abstraite Kebab, puis plein de classe qui hériteront de notre classe kebab suivant de quoi il est composé.

Problème de notre modélisation

Comme pour l'autre, notre code est en soit fonctionnel mais il y a un gros souci dans ce dernier. Si on veut rajouter à notre carte de nouveau kebab nous allons être obligés de créer une nouvelle sous-classe alors qu'on voulait juste retirer un ingrédient d'un kebab déjà existant. Le deuxième gros souci **est le prix d'un produit** car il y en a pas qu'un dans chaque kebab et que si l'on décide pour notre magasin de faire une réduction sur un produit se trouvant dans une vingtaine de kebab ou que l'on décide de supprimer tout simplement ce produit il va falloir modifier le prix final de tous les kebabs qui ont ce produit et cela fonctionne aussi dans l'autre sens si on veut rajouter un produit ou un accompagnement. Donc notre programme a beau fonctionner, il manque de flexibilité. Pour le rendre plus flexible on va transformer notre conception qui actuellement est statique en une conception totalement dynamique.

Nouvelle modélisation

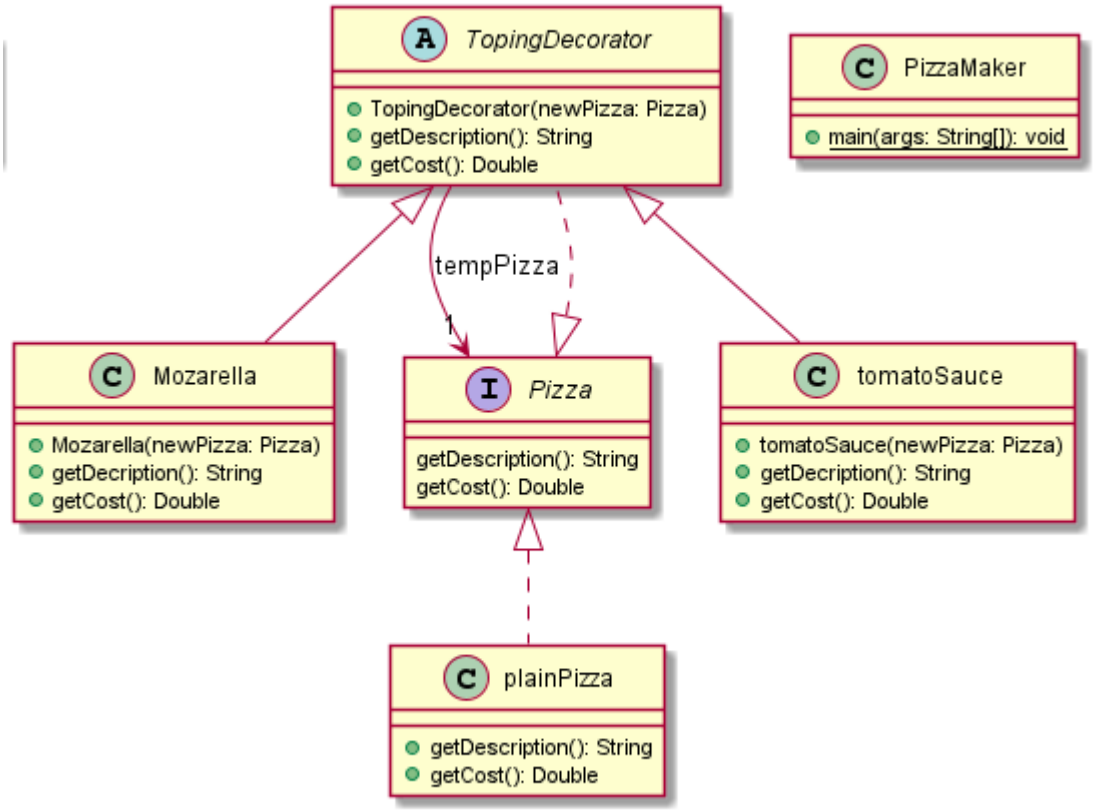
- Decorator (quand plusieurs objets ont une utilité similaire sans devoir s'affecter les uns les autres, et doivent être changés dans leur comportement à la volée, pensez à la garniture de pizza à commander ou à la mise en forme d'un élément dans Word) **Viens du cours de William**

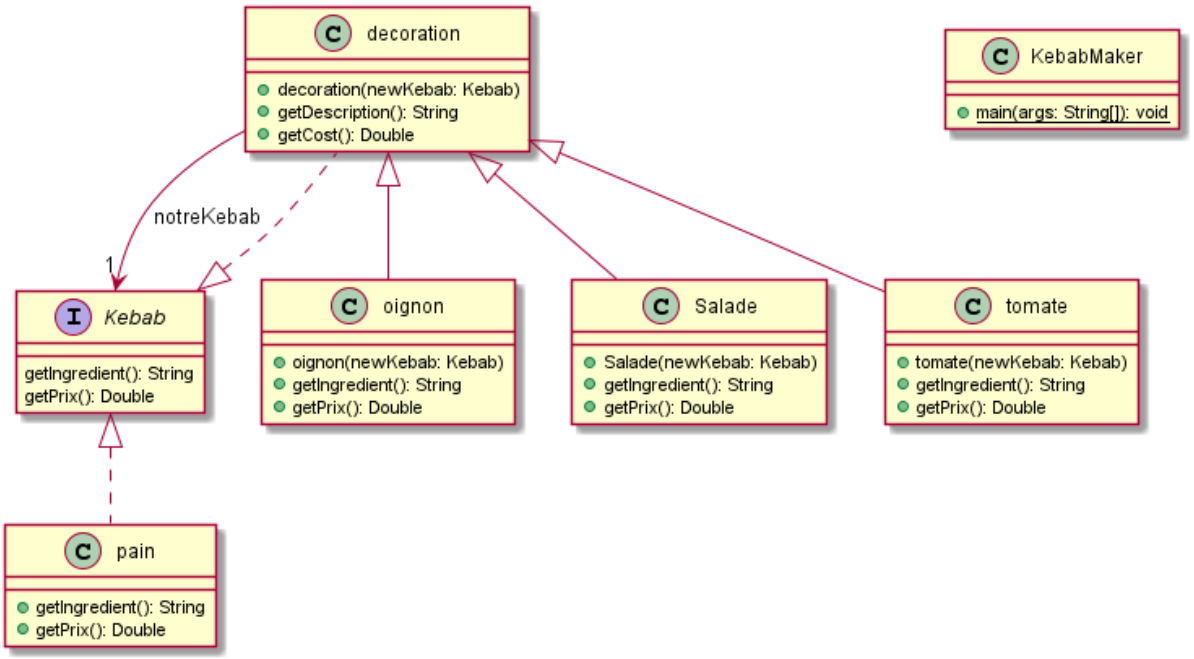
Le pattern design décorateur nous permet de modifier un objet dynamiquement. On va utiliser ce modèle quand on veut des capacités d'héritage avec des sous-classes mais qu'on doit ajouter des fonctionnalités au moment de l'exécution.
Ça simplifie le code car on ajoute beaucoup plus de fonctionnalités en utilisant des classes simples plutôt que d'essayer de passer par l'héritage. Le grand avantage est qu'il nous permet d'éviter de devoir réécrire l'ancien code mais au contraire de pouvoir l'étendre à un nouveau code en le gardant tel qu'il est actuellement.

```java
public abstract class Kebab {

    public abstract void setDescription();

    public abstract String getIngredient();

    public abstract Double getPrix();
}
```

```java
public class TroisFromage extends Pizza{

    @Override
    public void setDescription(String newDescription) {
        // TODO Auto-generated method stub

    }

    @Override
    public String getDescription() {
        return "Mozzarella, Parmesan, Emmental";
    }

    @Override
    public Double getPrix() {
        return 20.00;
    }


}
```

Le code est bon et fonctionnel, mais a un gros souci. Le souci est qu'avec cette méthode, on va, pour chaque type pizza, devoir faire des sous-classes à l'infini, on doit en créer une pour la TroisFromage, mais si demain on ajoute 4 pizzas, il faut créer pour chacune d'elle 1 sous classes de pizza ce qui peut vite devenir contraignant et pénible. Autre problème mis en avant, le prix pour cette pizza la si jamais on fait une réduction ou qu'on change le prix d'un des articles présent dans la pizza, on va devoir modifier le prix dans toutes les sous-classes pour faire fonctionner le code correctement à nouveau. Alors qu'on voulait modifier juste une chose, on se retrouve à devoir modifier chaque prix dans toutes les sous-classes .
Actuellement, notre code est Statique, mais nous voulons une composition dynamique pour le pattern.


ToppingDecorator
ToppingDecorator(newPizza: Pizza)
getDescription(): String
getCost(): Double
PizzaMaker
main(args: String[]): void
tempPizza
1
Mozarella
Mozarella(newPizza: Pizza)
getDecription(): String
getCost(): Double
Pizza
getDescription(): String
getCost(): Double
tomatoSauce
tomatoSauce(newPizza: Pizza)
getDecription(): String
getCost(): Double
plainPizza
getDescription(): String
getCost(): Double


decoration
decoration(newKebab: Kebab)
getDescription(): String
getCost(): Double
KebabMaker
main(args: String[]): void
notreKebab
1
Kebab
getIngredient(): String
getPrix(): Double
oignon
oignon(newKebab: Kebab)
getIngredient(): String
getPrix(): Double
Salade
Salade(newKebab: Kebab)
getIngredient(): String
getPrix(): Double
tomate
tomate(newKebab: Kebab)
getIngredient(): String
getPrix(): Double
pain
getIngredient(): String
getPrix(): Double

(JCP si mon diagramme UML est bon mais ok tier)
globalement on a une relation entre la pizza et le topping donc chaque pizza aura un topping
fleche d'héritage entre topingdecorator et mozzarella remplacer par association MNT avec le pattern décorateur

```java
package Creation;

public interface Kebab {

    public String getIngredient();

    public Double getPrix();

}
```

Notre classe pizza est maintenant passée en interface. Ce sera alors plus facile à gérer et on rend notre description et prix public pour avoir le prix de toutes les garnitures et la description de chaque pizza comme on le souhaite.

```java
package Creation;

public class pain implements Kebab {

    public String getIngredient() {
        return "Pita";
    }

    public Double getPrix() {
        System.out.println("Prix du pain : " + 1.85 + "€");
        return 1.85;
    }


}
```

Ensuite, on va faire une classe qui va implémenter notre interface pizza. On l'appelle plainPizza, et dedans on va aussi implémenter toutes les méthodes de notre interface ainsi que toutes les pizzas que nous allons créer. Elles auront une base sur cette classe donc on complète ensuite les méthodes en mettant ce qu'on a envie de voir apparaître. Pour le prix on met le prix de la pâte et on passe a la classe suivante

```java
package Creation;

abstract class decoration implements Kebab {

    protected Kebab notreKebab;

    public decoration(Kebab newKebab) {
        notreKebab = newKebab;
    }

    public String getDescription() {
        return notreKebab.getIngredient();
    }

    public Double getCost() {
        return notreKebab.getPrix();
    }

}
```

On va passer ensuite sur notre TopingDecorator on va la mettre en classe abstraite pour pouvoir par la suite l'étendre sur d'autres classes puis on va implémenter notre interface avec cette classe pour avoir accès aux mêmes méthodes. Ensuite on va déclarer un protected pour y avoir accès dans cette classe et dans toutes les autres du même package. Puis on va modifier pour obtenir le prix et la description de la pizza.

```java
package Creation;

public class tomate extends decoration{


    public tomate(Kebab newKebab) {
        super(newKebab);

    }

    public String getIngredient() {
        return notreKebab.getIngredient() + ", tomate";
    }

    public Double getPrix() {
        System.out.println("prix de la tomate: " + 0.20);

        return notreKebab.getPrix()+0.20;
    }

}
```

Enfin, on va créer notre ingrédient qui va étendre le toping. Ensuite on va rajouter dans la description que ce toping s'y trouve et combien on ajoute en prix pour la pizza.

```java
package Creation;

public class Salade extends decoration{


    public Salade(Kebab newKebab) {
        super(newKebab);

    }

    public String getIngredient() {
        return notreKebab.getIngredient() + ", salade";
    }

    public Double getPrix() {
        System.out.println("prix de la salade: " + 0.30);

        return notreKebab.getPrix()+0.30;
    }

}
```

```java
package Creation;

public class KebabMaker {

    public static void main(String[] args) {
        Kebab kebabSaladeTomateOignon = new Salade(new tomate(new oignon(new pain())));

        System.out.println("Ingrédients : " + kebabSaladeTomateOignon.getIngredient());

        System.out.println("Prix du kebab: " + kebabSaladeTomateOignon.getPrix());
    }

}
```

Enfin, on va tout rassembler et faire un main pour faire fonctionner notre ode sur la console.

Donc grâce à notre pattern on a pas beaucoup touché à notre ancien code d'un point de vue théorique car on utilise toujours nos même méthodes mais grâce aux autres modification apportée ajouter un autre ingrédient n'est pas un problème il suffit de créer une nouvelle classe, de rajouter à la description l'ingrédient et aux prix combien coute l'ingrédient puis dans notre main de rajouter a notre pizza l'ingrédient ce qui nous fait drastiquement gagner du

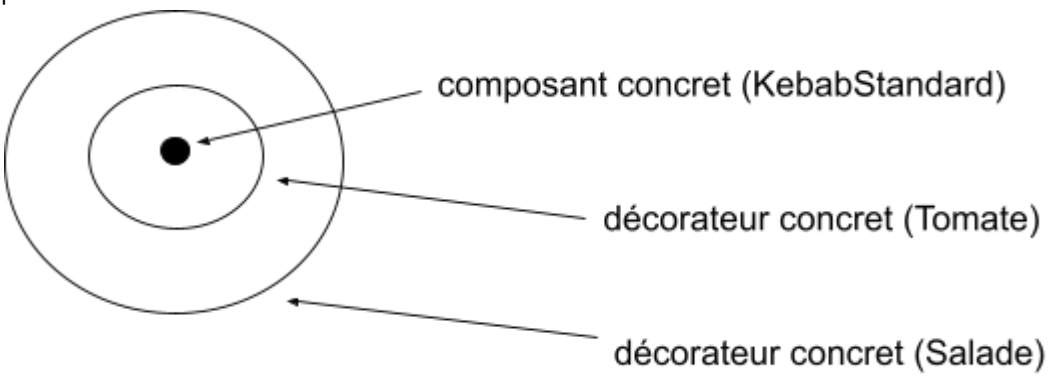
temps car on a pas a tout modifier juste ajouter ou supprimer des infos et/ou une classe si on veut créer un nouvelle ingrédient pour une nouvelle pizza et pas a avoir pour une nouvelle pizza à créer une classe avec la nouvelle pizza et tous ses ingrédient plus son prix

SOLID :

DIT : Le haut niveau ne doit pas dépendre du bas niveau, et le haut niveau doit dépendre d'abstraction.

Nouvelle modélisation (**kebab**) :

Cette nouvelle modélisation est bien plus intelligente car plus flexible. On a ici notre composant qui est ici une interface comprenant des méthodes. Ensuite, on a notre composant concret qui est ici notre Kebab Standard qui hérite de notre interface et de ses méthodes. Maintenant on a notre décorateur qui lui aussi va hériter de notre interface Kebab et de ses méthodes. En utilisant le pattern décorateur notre décorateur n'est pas seulement un kebab (EST-UN) mais également (A-UN) composant. Car quand on instancie le décorateur, il fournit un autre composant. Le décorateur se comporte comme un composant mais fait également référence à un autre composant concret. Et ce composant peut être un autre décorateur. Parce-que un décorateur est un composant. Ensuite nous avons nos décorateurs concrets (Salade,Tomate,Oignon). Et donc avec cette modélisation nous pouvons créer un grand nombre de kebab différent tout en cumulant les prix de manière efficace. (Explication avec schéma plus simple pour prouver la flexibilité.



)

Le décorateur concret Salade, va s'ajouter au (composant concret + décorateur concret Tomate).

Webographie :

https://refactoring.guru/fr/design-patterns/decorator/java/example
https://blog.cellenza.com/developpement-specifique/le-design-pattern-decorator-decorateur/
https://www.adimeo.com/blog-technique/design-patterns-a-quoi-ca-sert-et-comment-les-utiliser
https://ryax.tech/fr/design-pattern-cest-quoi-et-pourquoi-lutiliser/

https://www.youtube.com/watch?v=j40kRwSm4VE (pizzeria)

https://www.youtube.com/watch?v=GCraGHx6gso (autre vidéo)

https://www.youtube.com/watch?v=WghYGmUd9nQ&ab_channel=BracketShow (quebec)

https://fr.wikipedia.org/wiki/D%C3%A9corateur_(patron_de_conception) wikipedia

https://www.elao.com/blog/dev/design-pattern-decorator