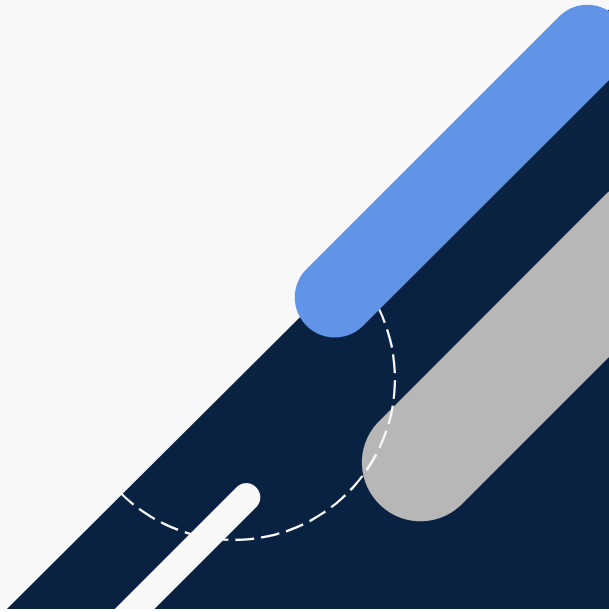
Pattern Decorator

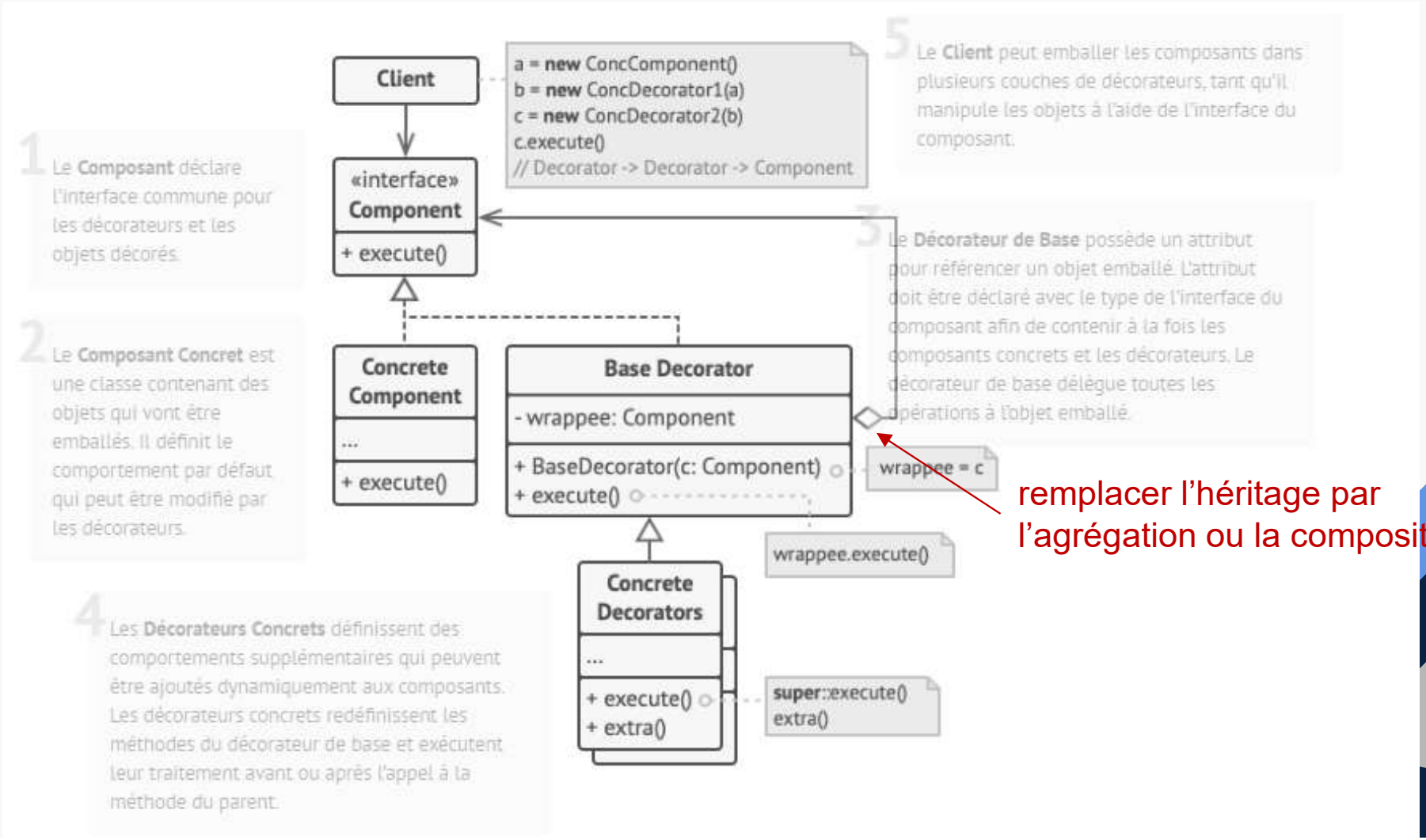


Définition

Le décorateur:

- est un patron structurel qui permet d'ajouter dynamiquement des fonctionnalités à un objet sans modifier sa classe.
- Il consiste à "envelopper" un objet dans une autre classe (appelée décorateur) qui implémente la même interface (même comportement).

Structure UML



remplacer l'héritage par l'agrégation ou la composition.

Exemple d’implémentation du pattern decorator: gestion d’un Coffee Shop

Imaginons un système de gestion d’un Coffee Shop. Chaque boisson (café simple, cappuccino, etc.) possède un coût et une description.

Au départ, nous avons une seule boisson :

- Un café simple à 2€.

Mais, réellement, un client peut vouloir ajouter :

- du lait,
- du sucre,
- du chocolat, ou
- même plusieurs combinaisons (ex. café + lait + sucre).

Exemple d’implémentation du pattern decorator: gestion d’un Coffee Shop

Avec l’héritage classique, nous aurions besoin de définir plusieurs classes comme :

- CaféAvecLait
- CaféAvecSucre
- CaféAvecLaitEtSucre
- CaféAvecChocolatEtLait
- Etc

= › Explosion de classes, code rigide et difficile à maintenir.

Exemple d’implémentation du pattern decorator: gestion d’un Coffee Shop

Objectifs :

Le patron Décorateur permet de :

- Ajouter dynamiquement des options (lait, sucre, chocolat…) à une boisson existante.
- Éviter la multiplication des sous-classes.
- Combiner librement les options à l’exécution.
- Respecter le principe Ouvert/Fermé (OCP) :on étend le comportement sans modifier les classes existantes.

Exemple d’implémentation du pattern decorator: gestion d’un Coffee Shop

Principe du decorator:

Chaque option de coffe est un décorateur (lait, sucre, chocolat…) qui :

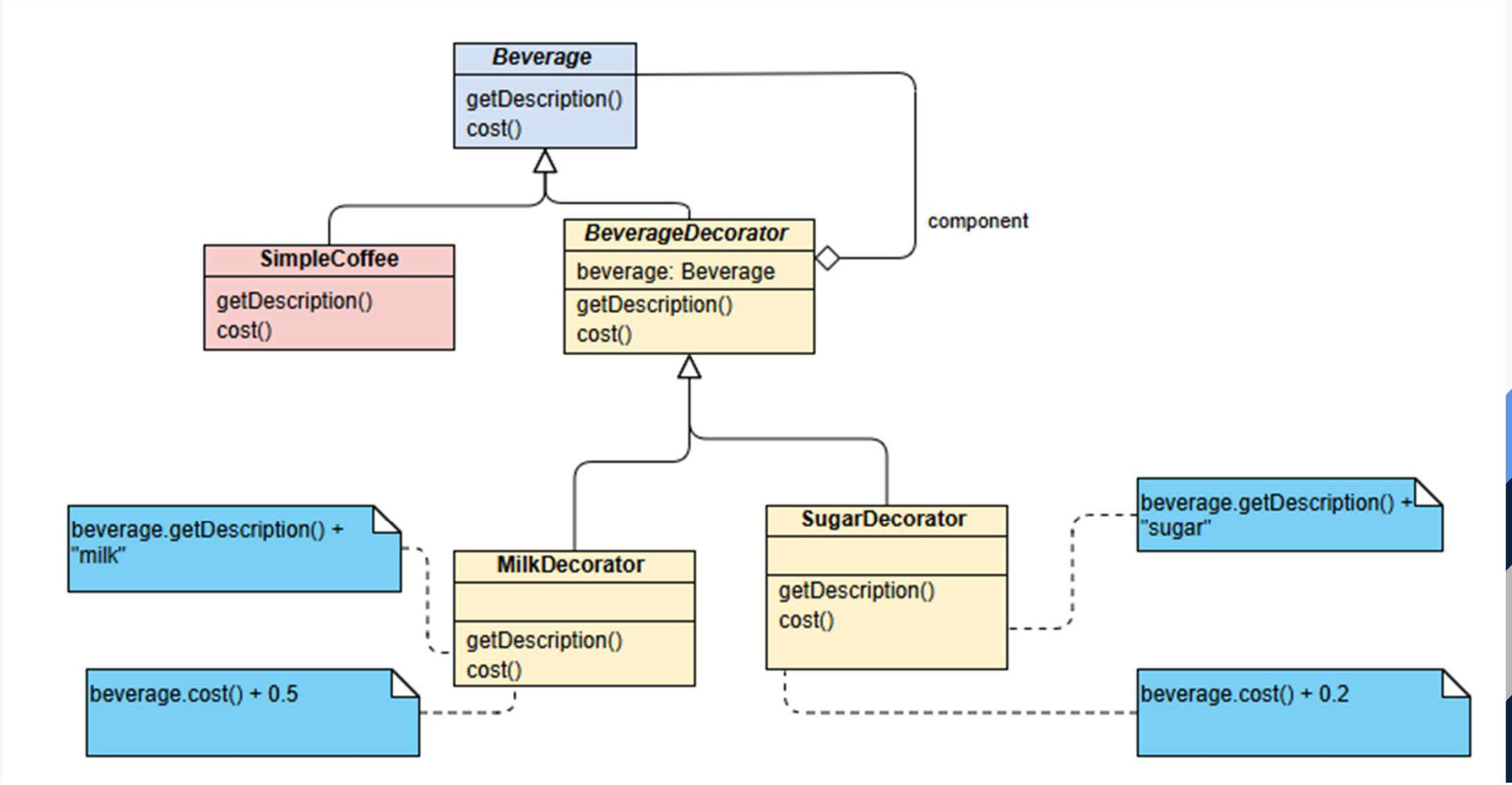
- implémente la même interface **Beverage** que la boisson de base.
- enveloppe un objet de type Beverage

Exemple: si on désire créer un :“Café simple, avec lait, avec sucre”

On aura:

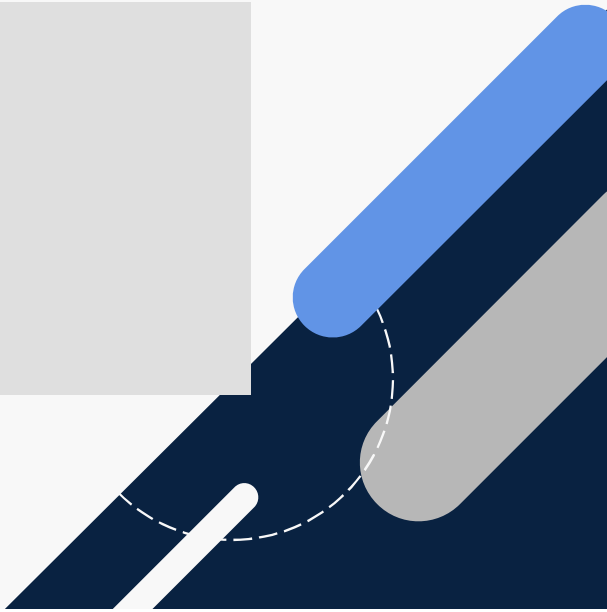
```
Beverage deluxe = new SugarDecorator(new
MilkDecorator(new SimpleCoffee()));
```

Structure UML

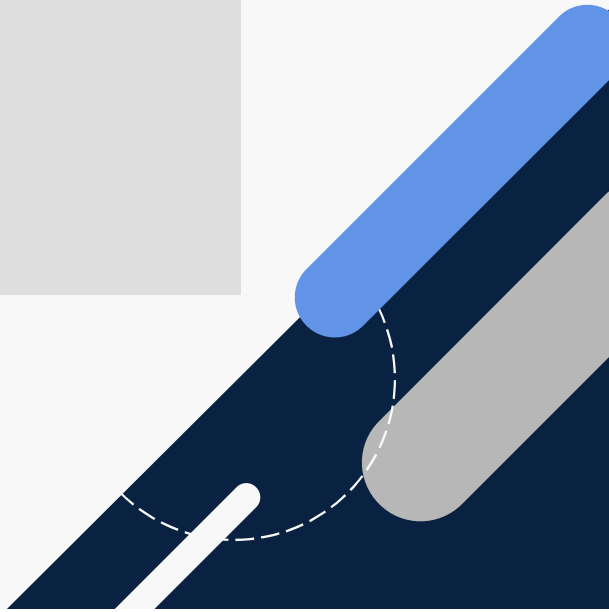


```
public class SimpleCoffee implements Beverage {
    @Override
    public String getDescription() {
        return "Café simple";
    }
    @Override
    public double cost() {
        return 2.0;
    }
}
```

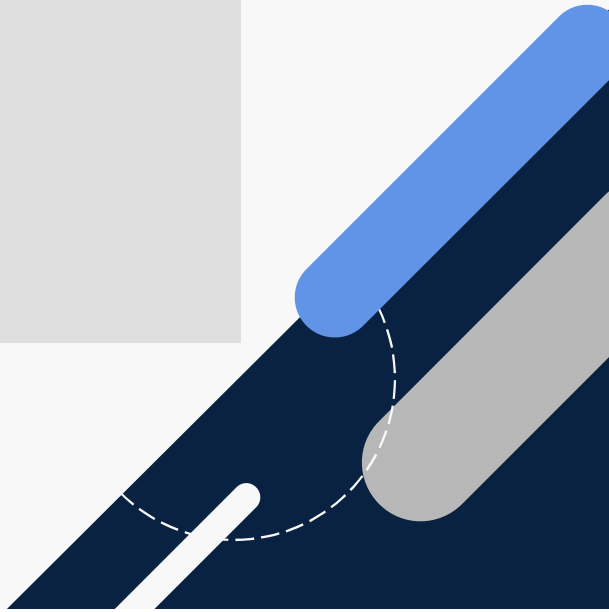
```
public abstract class BeverageDecorator implements Beverage {
    protected Beverage beverage;

    public BeverageDecorator(Beverage beverage) {
        this.beverage = beverage;
    }
}
```



```
public class MilkDecorator extends BeverageDecorator {
    public MilkDecorator(Beverage beverage) {
        super(beverage);
    }

    @Override
    public String getDescription() {
        return beverage.getDescription() + ", lait";
    }

    @Override
    public double cost() {
        return beverage.cost() + 0.5;
    }
}
```



```
public class SugarDecorator extends BeverageDecorator {
    public SugarDecorator(Beverage beverage) {
        super(beverage);
    }

    @Override
    public String getDescription() {
        return beverage.getDescription() + ", sucre";
    }

    @Override
    public double cost() {
        return beverage.cost() + 0.2;
    }
}
```



```
public class Main {
    public static void main(String[] args) {
        Beverage coffee = new SimpleCoffee();
        System.out.println(coffee.getDescription() + " = " + coffee.cost() + "€");

        Beverage milkCoffee = new MilkDecorator(coffee);
        System.out.println(milkCoffee.getDescription() + " = " + milkCoffee.cost() + "€");

        Beverage deluxe = new SugarDecorator(new MilkDecorator(new
SimpleCoffee()));
        System.out.println(deluxe.getDescription() + " = " + deluxe.cost() + "€");
    }
}
```