COURS

Créez des applications full-stack robustes et structurées à l'aide de design patterns

P1C3 - Appliquez votre premier design pattern

Vérifiez votre travail

Dans les trois stacks, le résultat doit faire apparaître les mêmes rôles :

Rôle de Strategy	Java	Python	C#
Contrat	ShippingCostStrategy	Protocol, classe abstraite ou fonction injectée	IShippingCostStrategy
Contexte	ShippingCostService	ShippingCostService	ShippingCostService
Stratégies concrètes	StandardShipping, ExpressShipping, StorePickup	StandardShipping, ExpressShipping, StorePickup	StandardShipping, ExpressShipping, StorePickup

Avant

Avant le refactoring, le contexte reçoit une chaîne de caractères, choisit une branche et contient lui-même toutes les formules. Ajouter un mode de livraison oblige donc à modifier une méthode qui connaît déjà les autres calculs.

Pendant le refactoring

1. Créez le contrat de calcul.
2. Déplacez la formule standard dans une première stratégie et vérifiez le résultat.
3. Déplacez la formule express et vérifiez à nouveau le résultat.
4. Injectez la stratégie dans le contexte.
5. Supprimez le paramètre `mode` et la condition.
6. Ajoutez le retrait en magasin, qui retourne `0`, sans modifier le contexte.

Après

Voici une proposition de correction en Java :

```
public interface ShippingCostStrategy {
    double calculate(Order order);
}

public final class StandardShipping implements ShippingCostStrategy {
    public double calculate(Order order) {
        return 4.90 + order.getWeight() * 0.50;
    }
}

public final class ExpressShipping implements ShippingCostStrategy {
    public double calculate(Order order) {
        return 9.90 + order.getWeight() * 0.90;
    }
}

public final class StorePickup implements ShippingCostStrategy {
    public double calculate(Order order) {
        return 0;
    }
}

public final class ShippingCostService {
    private final ShippingCostStrategy strategy;

    public ShippingCostService(ShippingCostStrategy strategy) {
        this.strategy = strategy;
    }

    public double calculate(Order order) {
        return strategy.calculate(order);
    }
}
```

Voici une proposition de correction en Python :

```
from typing import Protocol

class ShippingCostStrategy(Protocol):
    def calculate(self, order) -> float:
        ...

class StandardShipping:
    def calculate(self, order) -> float:
        return 4.90 + order.weight * 0.50

class ExpressShipping:
    def calculate(self, order) -> float:
        return 9.90 + order.weight * 0.90

class StorePickup:
    def calculate(self, order) -> float:
        return 0.0

class ShippingCostService:
    def __init__(self, strategy: ShippingCostStrategy):
        self.strategy = strategy

    def calculate(self, order) -> float:
        return self.strategy.calculate(order)
```

En Python, une fonction de calcul injectée pourrait également porter le contrat. Le `Protocol` rend simplement la capacité attendue explicite pour les outils de typage.

Voici une proposition de correction en C# :

```
public interface IShippingCostStrategy
{
    decimal Calculate(Order order);
}

public sealed class StandardShipping : IShippingCostStrategy
{
    public decimal Calculate(Order order) =>
        4.90m + order.Weight * 0.50m;
}

public sealed class ExpressShipping : IShippingCostStrategy
{
    public decimal Calculate(Order order) =>
        9.90m + order.Weight * 0.90m;
}

public sealed class StorePickup : IShippingCostStrategy
{
    public decimal Calculate(Order order) => 0m;
}

public sealed class ShippingCostService
{
    private readonly IShippingCostStrategy _strategy;

    public ShippingCostService(IShippingCostStrategy strategy) =>
        _strategy = strategy;

    public decimal Calculate(Order order) =>
        _strategy.Calculate(order);
}
```

Bénéfice vérifié

Le contexte ne reçoit plus de valeur `mode` et ne connaît plus les formules concrètes. L'ajout de `StorePickup` n'a nécessité aucune modification de `ShippingCostService`, de `StandardShipping` ou de `ExpressShipping`.

Votre justification doit faire apparaître trois idées :

1. le service initial rassemble plusieurs calculs qui évoluent indépendamment ;

2. ces calculs partagent la même intention : déterminer des frais de livraison ;

3. Strategy permet de les tester, de les remplacer et d'en ajouter un sans modifier le contexte.

Si votre solution conserve le paramètre `mode` et la condition dans `ShippingCostService`, la responsabilité n'a pas réellement été déplacée. Si les formules étaient deux valeurs courtes, stables et sans perspective d'évolution, la condition pourrait rester plus lisible. Ici, la nouvelle variante annoncée et l'évolution indépendante des règles justifient le pattern.