

design patterns

par eliot demots



à propos

Eliot

ESGI IW 2021

**Développeur web
fullstack**

Wizards Technologies

PHP / React

et vous ?



test de positionnement

sommaire

1 - Principes de conception

2 - Design patterns

principes de conception

| Méthodologie pour concevoir un programme

S.O.L.I.D

D.R.Y

S.O.L.I.D

Acronyme des cinqs fondamentaux de la POO

Single responsibility

Open-Closed

Liskov Substitution

Interface segregation

Dependency inversion

single responsibility principle (srp)

| Une classe n'a **qu'une seule responsabilité**

Les +

- *Tests* - une classe n'ayant qu'une responsabilité est plus facile à tester
- *Organisation* - des classes plus petites sont plus faciles à rechercher et maintenir

open-closed principle (ocp)

Une classe est **ouverte à l'extension** mais **fermée à la modification**

Les +

- *Evolutivité* - on ne modifie pas le code existant donc on évite l'ajout de bugs potentiels

liskov substitution principle (lsp)

Si une classe B est un sous-type de la classe A, **alors on doit pouvoir remplacer A par B sans perturber le comportement du programme**

interface segregation principle (isp)

Les grandes interfaces doivent être **divisées** en plus petites interfaces.

Les +

- *Organisation* - Les classes qui implémentent les interfaces ne se préoccupent que des méthodes qui les intéressent

dependency inversion principle (dip)

| Découplage de modules grâce à des abstractions

Les +

- *Fléxibilité* - Permettre une meilleure évolutivité au programme

d.r.y

| Don't Repeat Yourself

"Chaque élément [...] doit avoir une représentation unique, non ambiguë et faisant autorité au sein d'un système."

Andy Hunt et Dave Thomas – The Pragmatic Programmer

d.r.y

| Don't Repeat Yourself

- Vise à réduire la répétition de modèle dans le code et la remplaçant par des **abstractions** ou en la **normalisant** pour *éviter la redondance*

design patterns

Solutions typiques pour résoudre des problèmes récurrents

- Ressemblent à des plans préétablis pour résoudre un problème de conception récurrent dans votre code
- C'est un **concept général pour résoudre un problème particulier**

design patterns

| Ce ne sont pas des algorithmes !

- Les 2 concepts décrivent des solutions typiques à des problèmes connus
- *L'algorithme* définit un ensemble d'actions claires pour atteindre un même résultat
- *Un patron de conception* appliqué à deux programmes différents **peut être différent**

catégories de design patterns

**Patrons de
création**

**Patrons
structurels**

**Patrons
comportementaux**

patrons de création

Fournissent un mécanisme de création d'objets augmentant la flexibilité et la réutilisation du code existant

patrons structurels

Expliquent comment assembler des objets et des classes dans des structures plus grandes, tout en gardant les structures flexibles et efficaces

patrons comportementaux

Assurent une communication efficace et l'attribution des responsabilités entre les objets

patrons de création

factory method

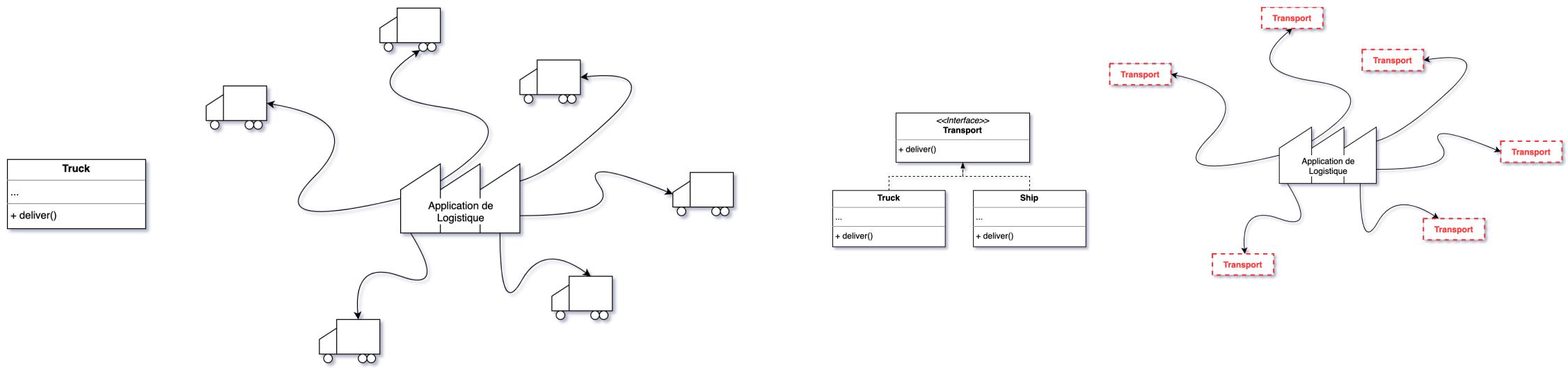
- **Interface** pour la création d'objets
- **Délègue** aux classes enfant le choix du **type d'objet** à créer

--

Use case

Quand on ne connaît pas à l'avance les types et dépendances précis des objets.

factory method



abstract factory

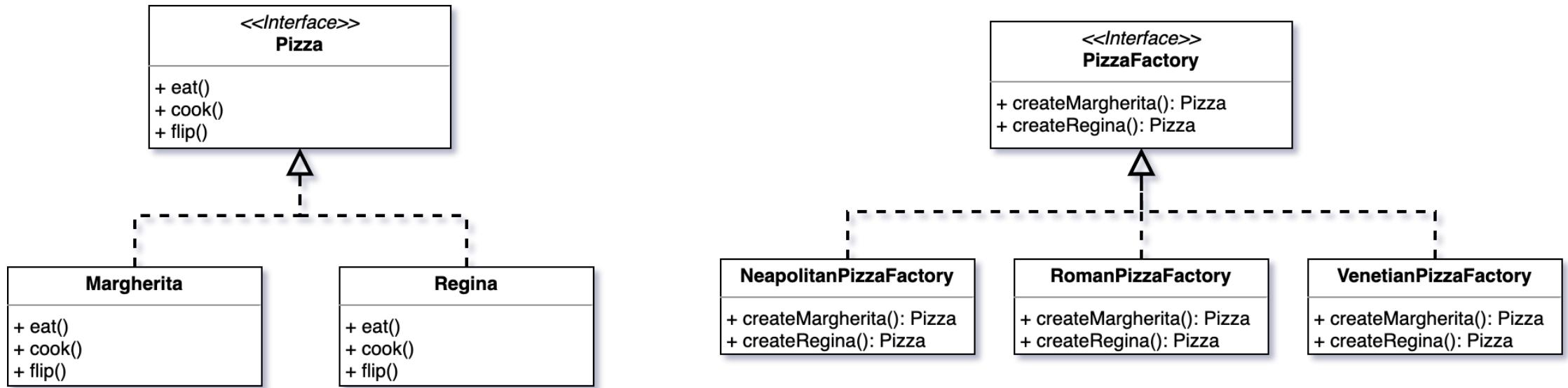
- Permet de créer des familles d'objets sans préciser leur classe concrète

--

Use case

Quand on souhaite manipuler les objets d'un même thème si on ne les connaît pas encore ou si on souhaite rendre le code évolutif

abstract factory



builder

- Permet de construire des objets complexes étape par étape

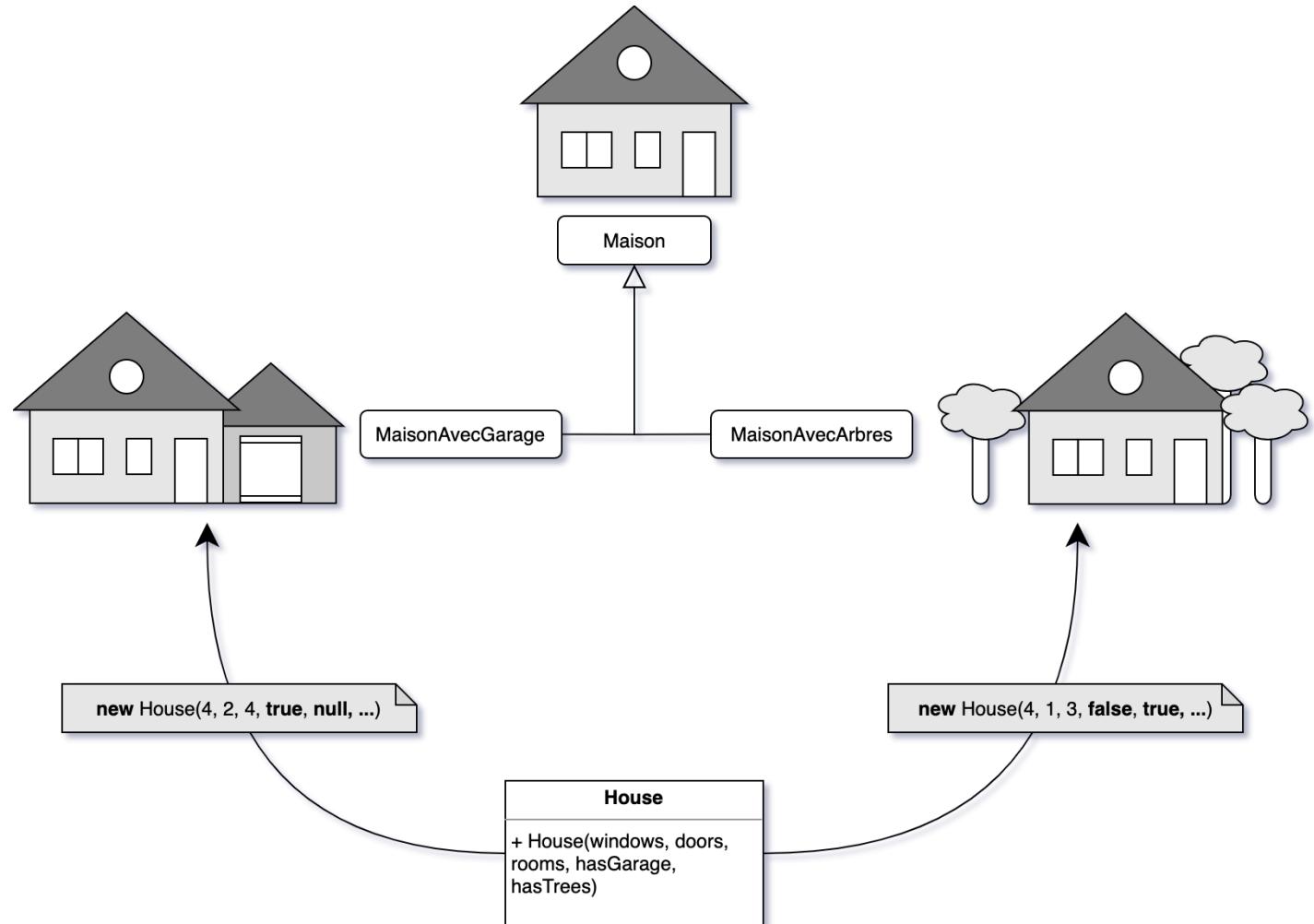
--

Use case

Si l'on souhaite **produire différentes variantes** d'un objet en utilisant le **même code**

builder

```
House  
+ buildWalls()  
+ buildDoors()  
+ buildWindows()  
+ buildRoof()  
+ buildGarage()  
+ buildTrees()  
+ getResult(): House
```



singleton

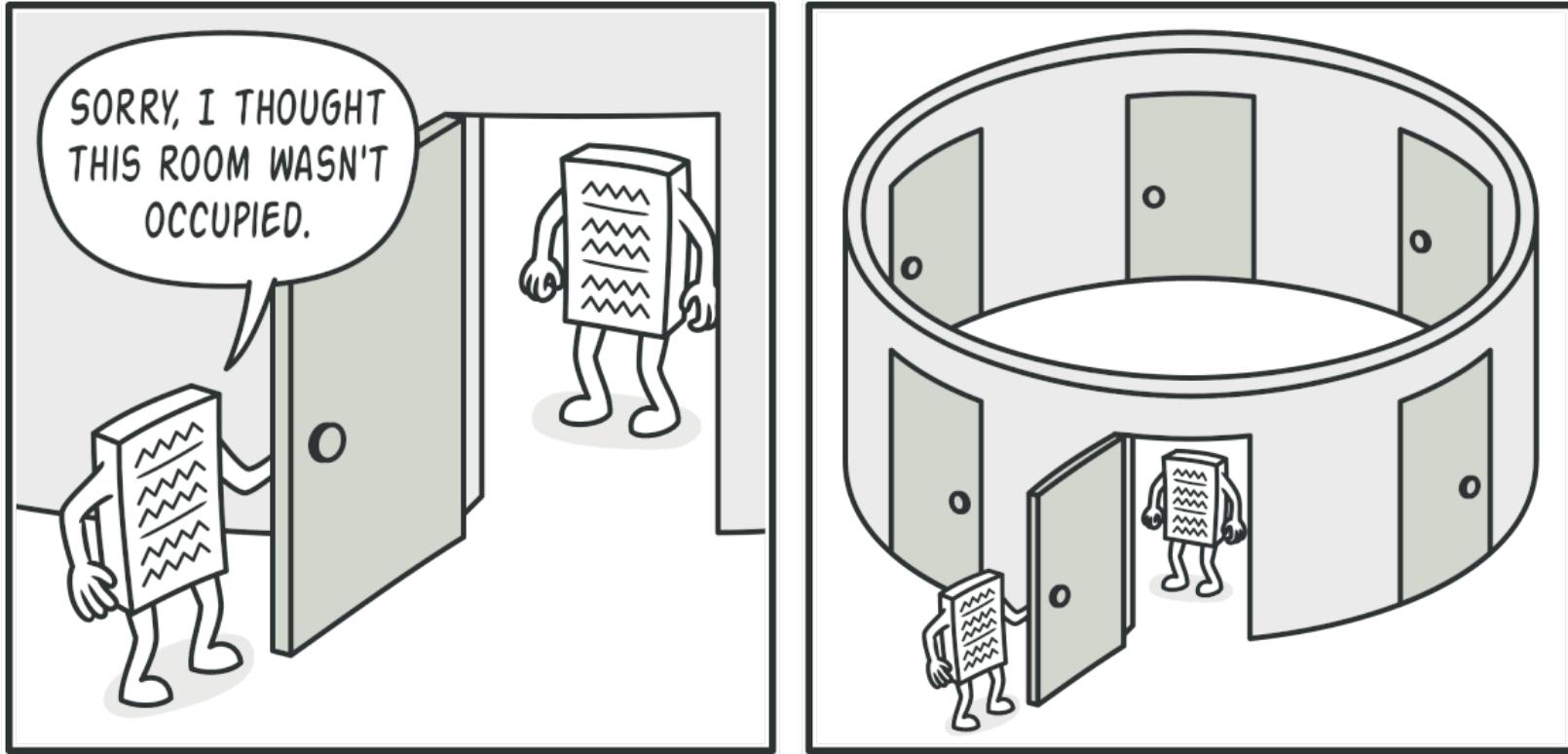
- Créer et fournir une instance qui ne pourra exister qu'en un seul exemplaire

--

Use case

Si l'on souhaite s'assurer qu'une classe n'aura toujours **qu'une seule instance** pour accéder à des ressources partagées - ex: pour se connecter à une base de données, lire un fichier

singleton



patrons structurels

adapter

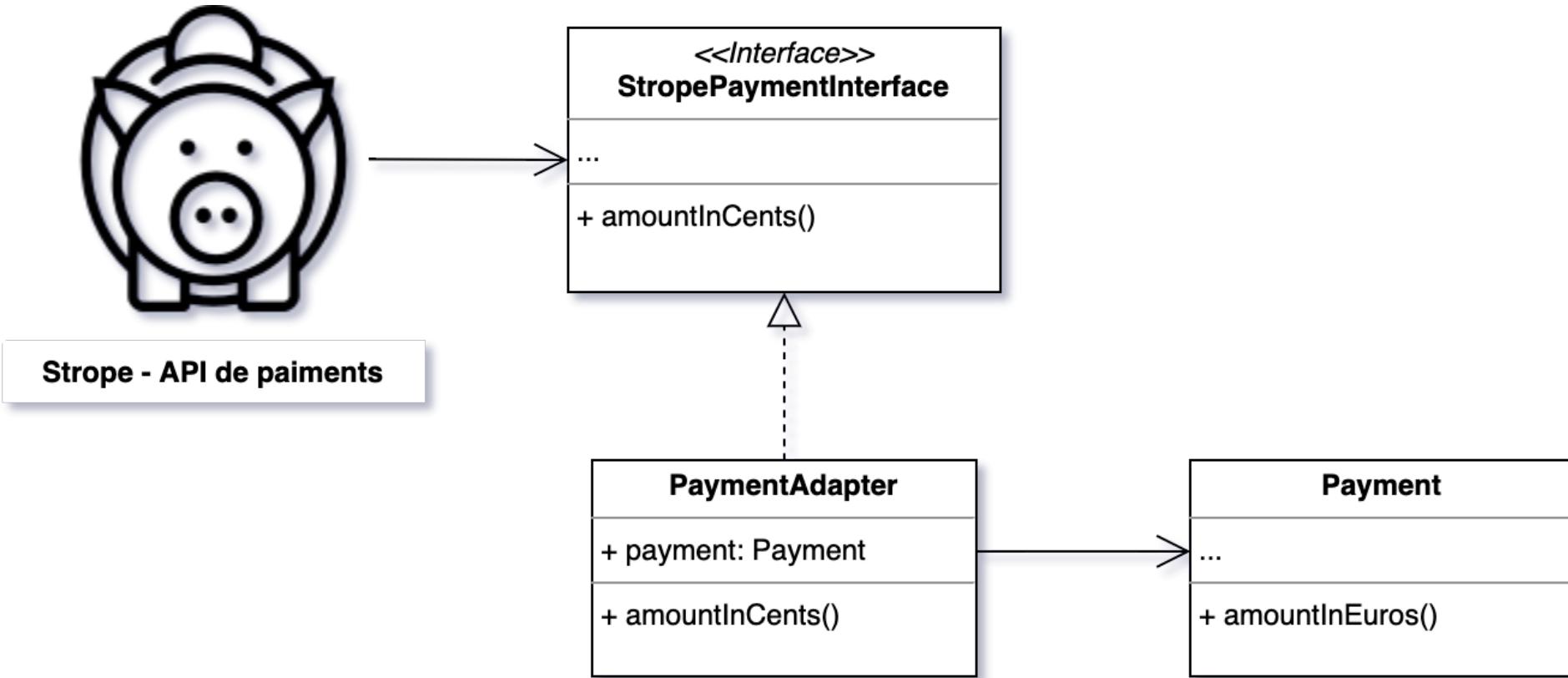
- Permet de faire collaborer des classes normalement incompatibles qui n'implémente pas les mêmes interfaces.

--

Use case

Lorsqu'on souhaite notamment utiliser des librairies externes qui ne sont pas forcément compatibles avec notre code existant

adapter



composite

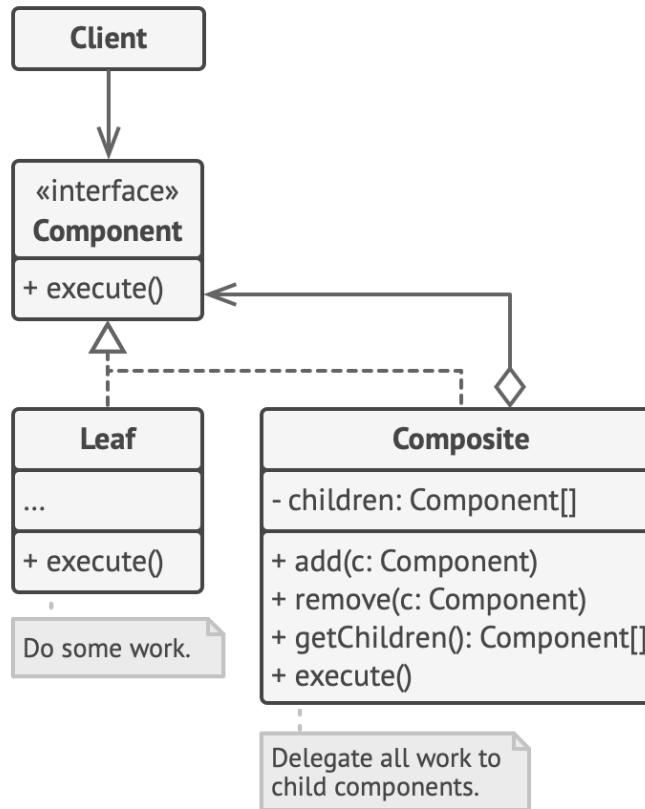
- Agence les objets en **arborescence** pour traiter ces arborescences comme des **objets individuels**.

--

Use case

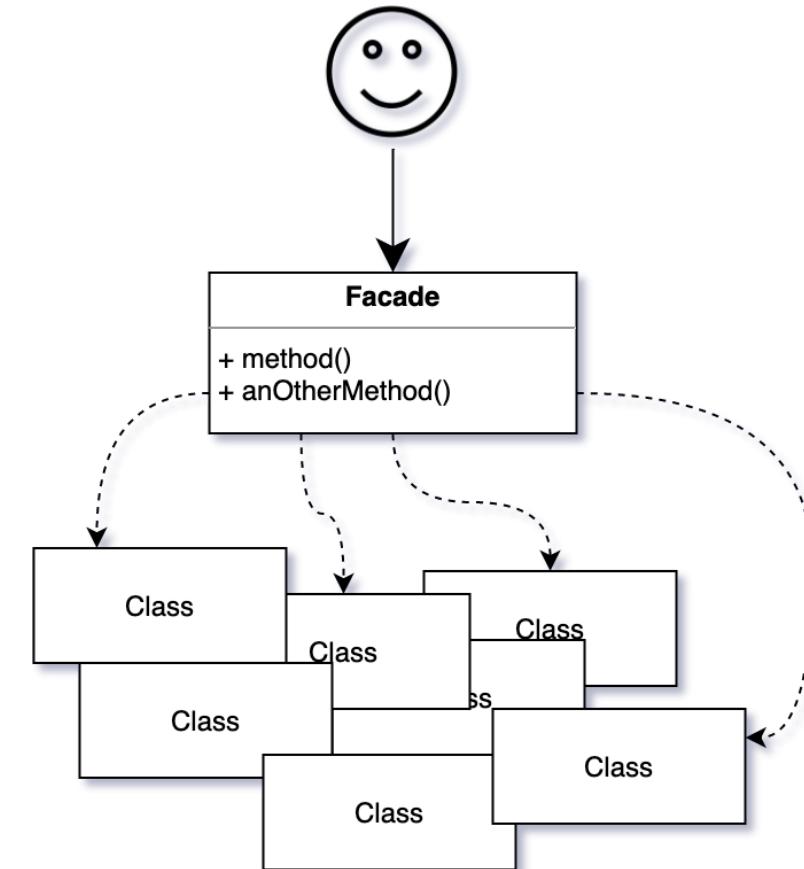
Quand on doit implémenter une arborescence objets

composite

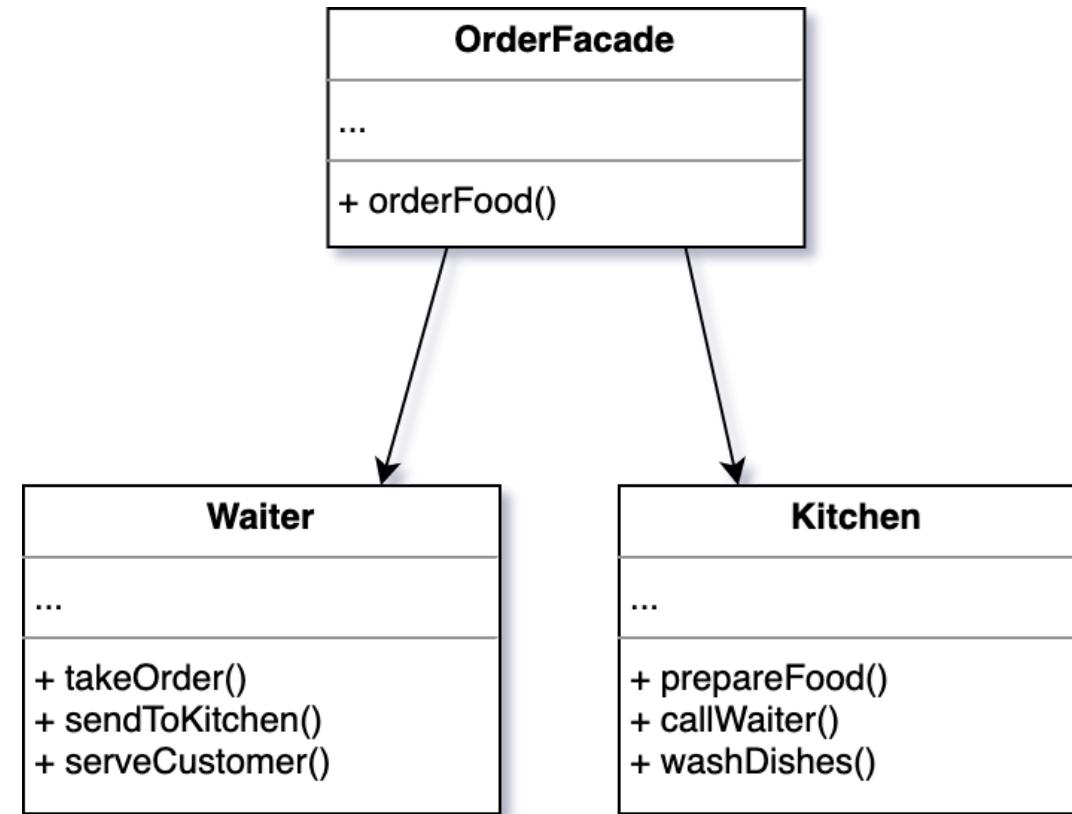


facade

- Procure une classe donnant un accès simplifié à n'importe quel ensemble complexe de classes.



facade



proxy

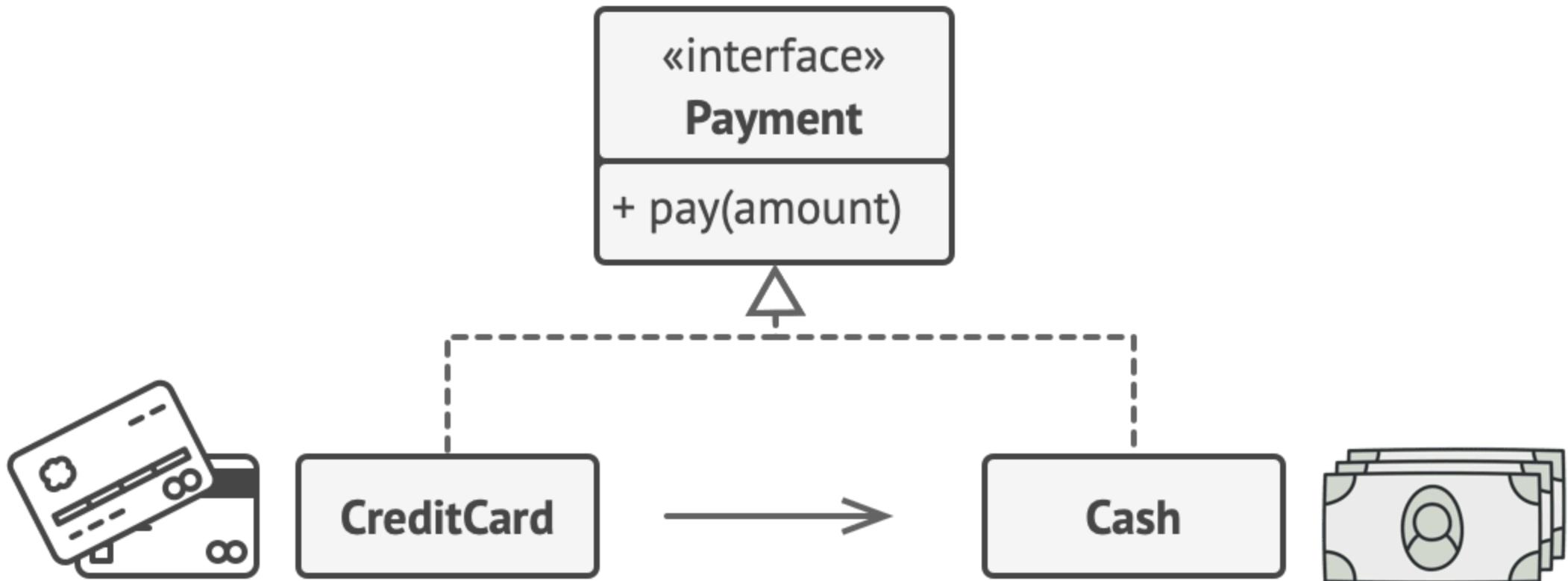
- Permet de maîtriser l'accès à un objet
- Permet d'effectuer des actions et/ou manipulations avant ou après que la demande ne lui parvienne.

--

Use case

Lorsque qu'on souhaite accéder à des données volumineuses, mais seulement de temps en temps

proxy



decorator

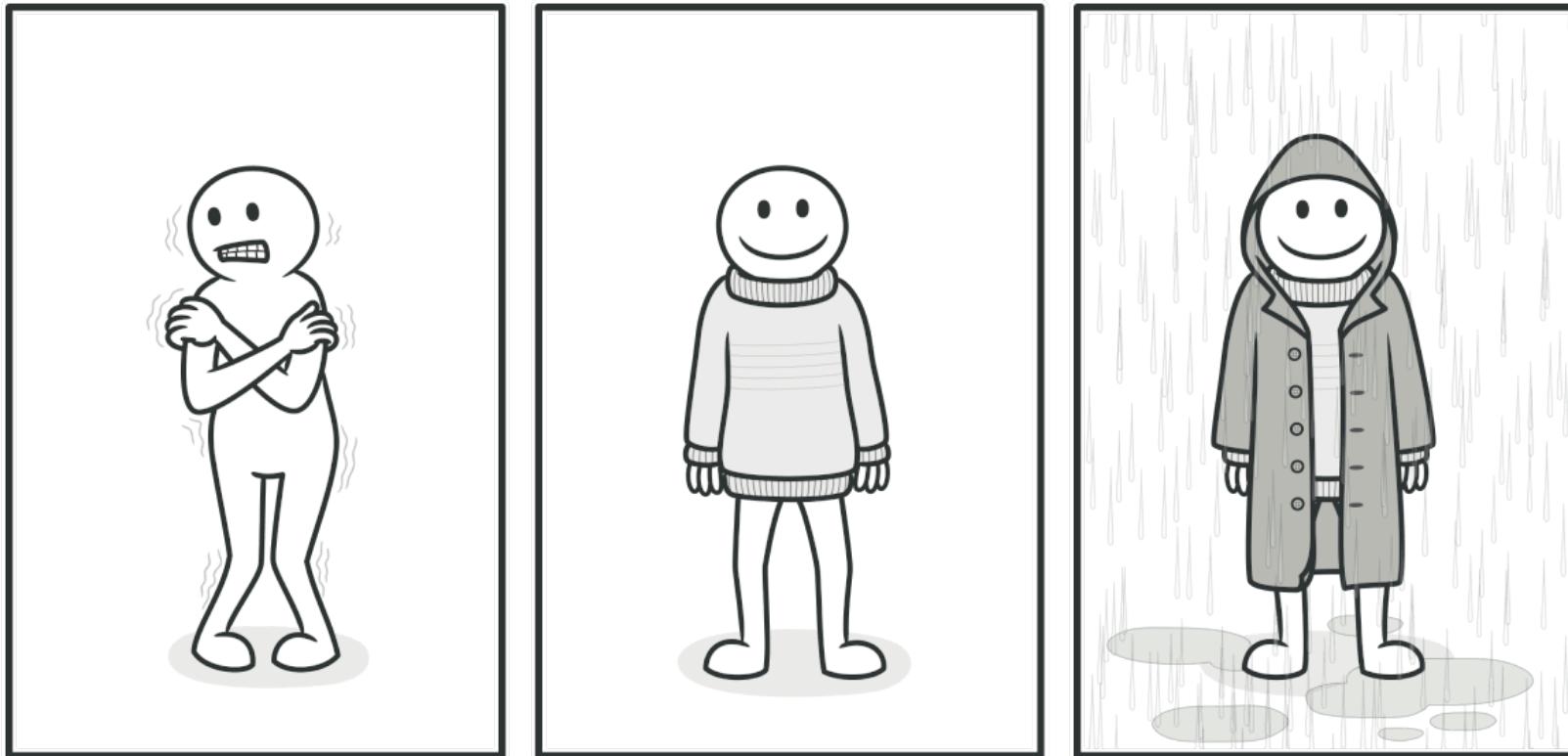
- Ajoute dynamiquement de nouveaux comportements à des objets.
- "Emballe" notre objets dans des boîtes qui ajoutent ces comportements.

--

Use case

Si l'on souhaite ajouter des fonctionnalités cumulables à notre objet.

decorator



bridge

- Sépare l'abstraction de son implémentation pour qu'elles puissent évoluer indépendamment.
- Facilite le découplage en plaçant l'abstraction et l'implémentation dans des hiérarchies distinctes.

--

Use case

Quand on veut éviter une explosion de classes en combinant plusieurs abstractions et implementations.

bridge

EmailWelcomeNotification

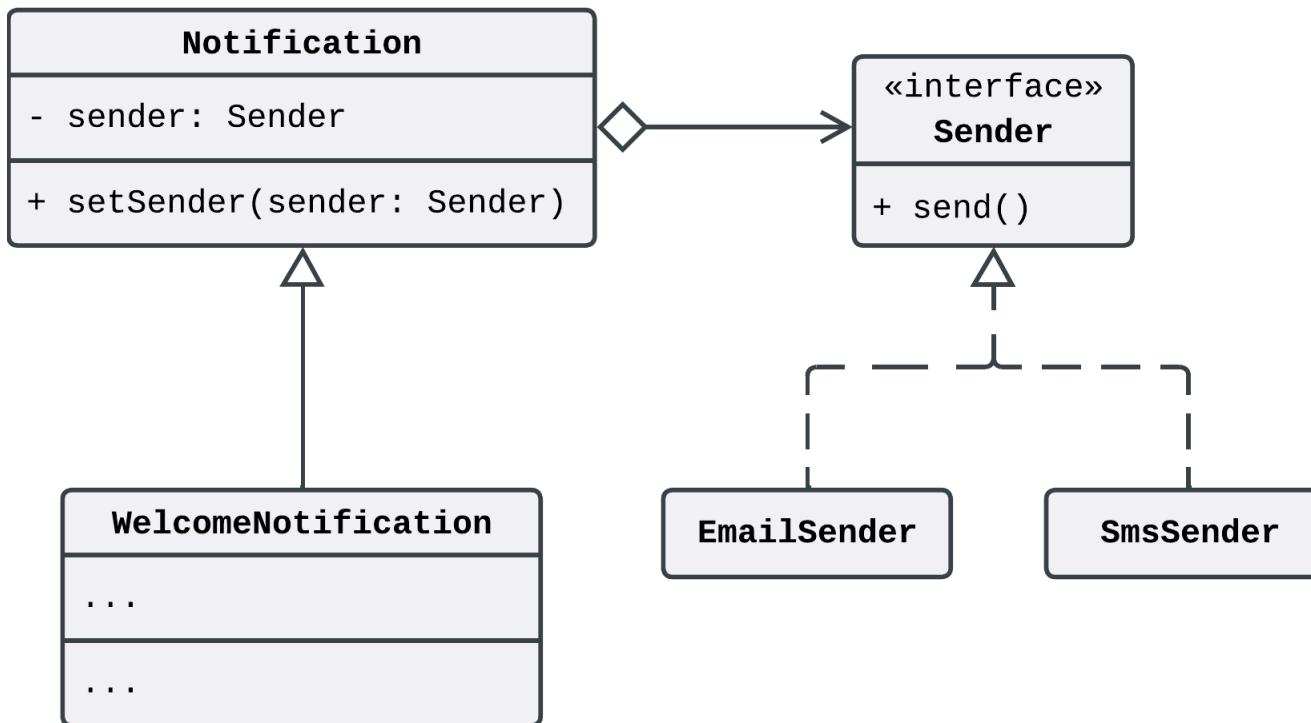
SmsWelcomeNotification

EmailPaymentSuccessNotification

SmsPaymentSuccessNotification

...

bridge



patrons comportementaux



observer

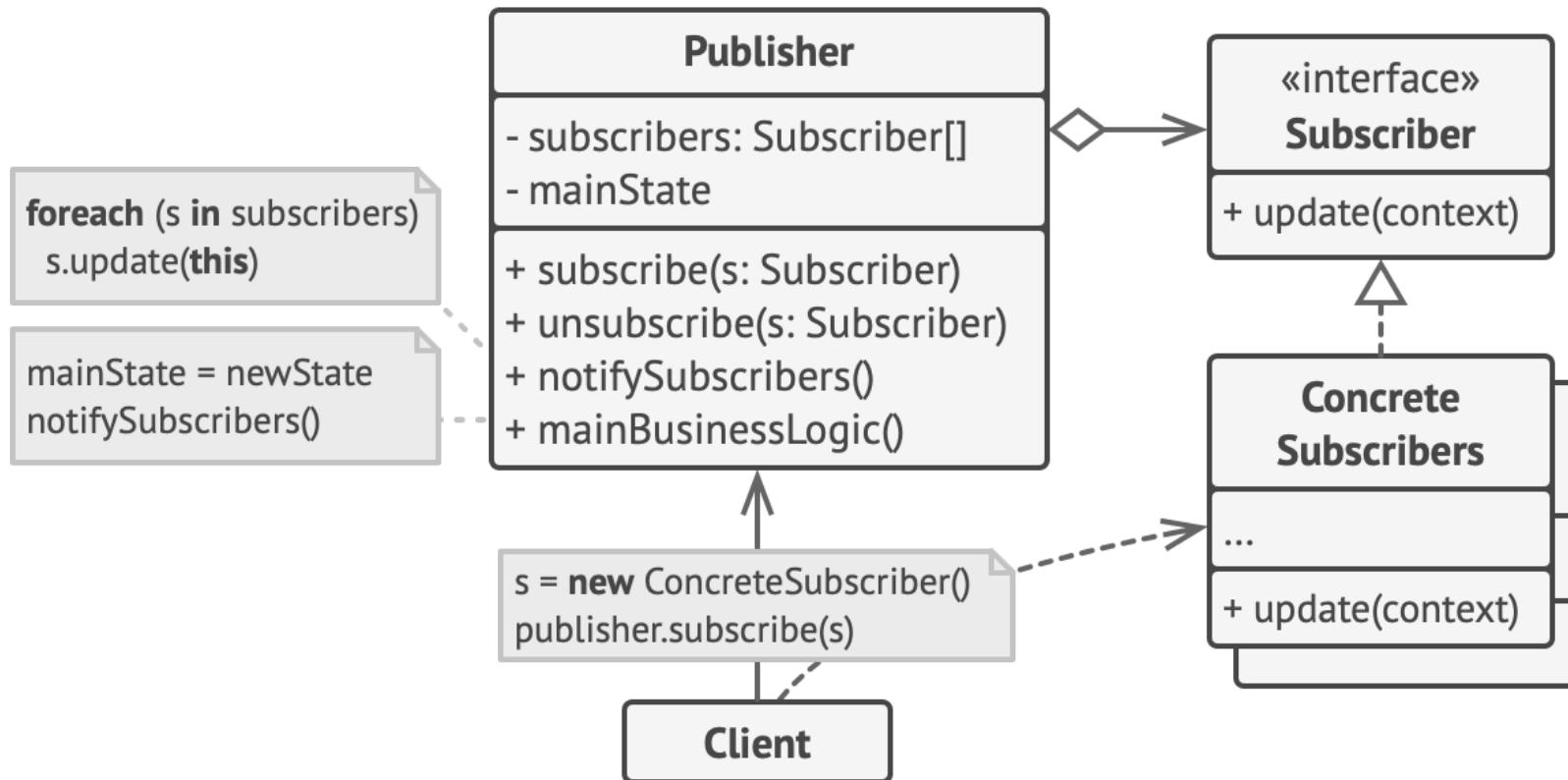
- Permet de notifier des objets qu'un évènement qu'ils observent a eu lieu

--

Use case

Lorsque que vous voulez que la modification de l'état d'un objet en impact d'autres, sans que vous les connaissiez à l'avance et qu'ils puissent changer dynamiquement.

observer



strategy

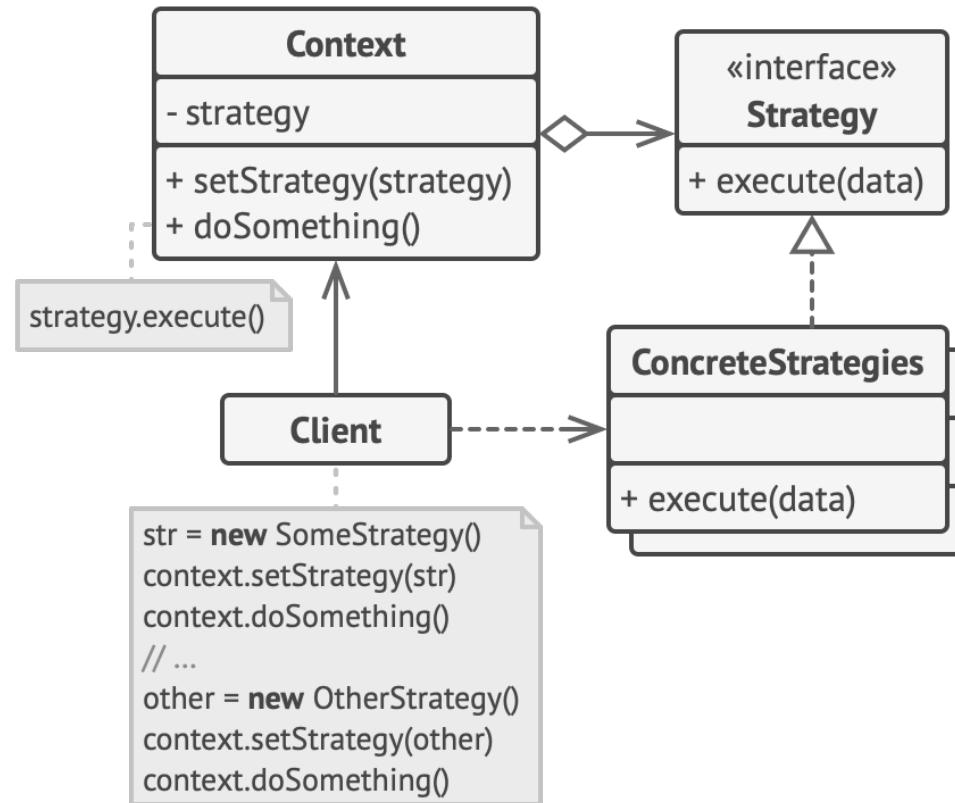
- Permet de définir des familles d'algorithmes dans des classes différentes et de rendre leurs objets interchangeables.

--

Use case

Lorsque vous voulez avoir différentes variantes d'un algorithme dans un objet et que vous voulez passer d'un algo à un autre à l'exécution.

strategy



command

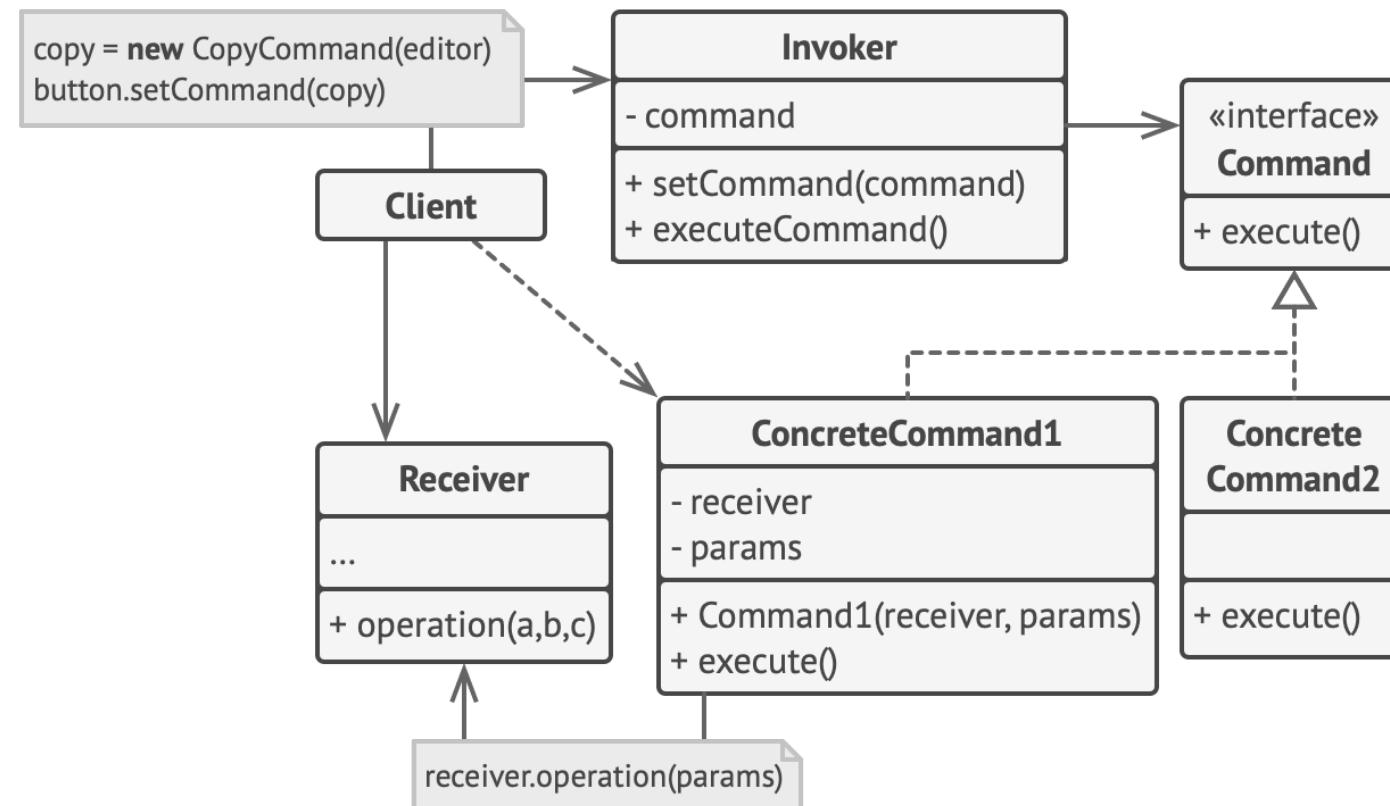
- Permet de transformer une action en objet contenant les détails de cette action
- Permet de planifier, mettre en file d'attente ou annuler des opérations.

--

Use case

- Lorsque qu'on souhaite planifier, mettre en file d'attente, exécuter à distance ou rendre réversible une action.

command



chain of responsibility

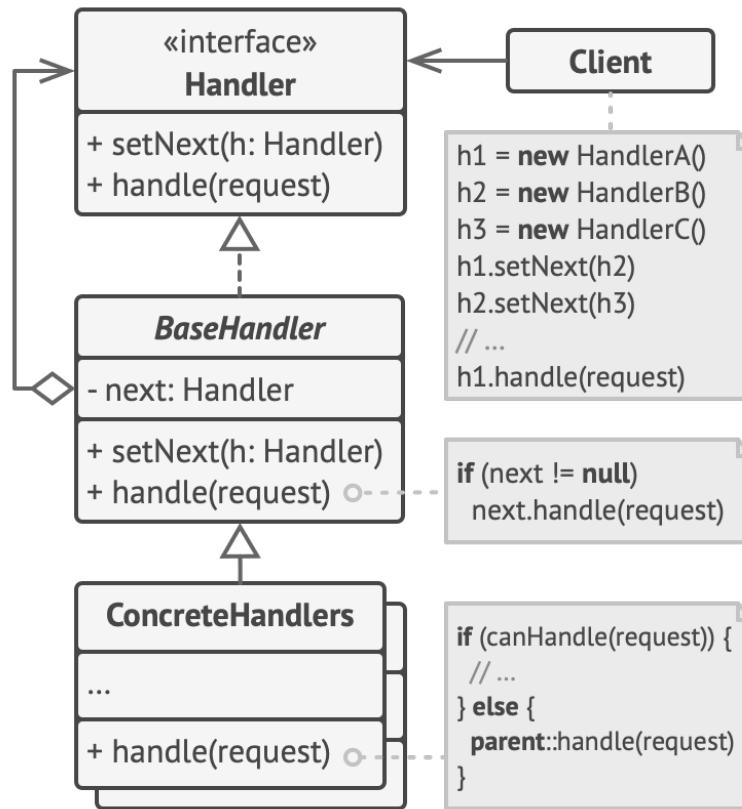
- Permet de faire circuler une demande parmi une chaîne de "manipulateurs".
- Ces manipulateurs peuvent traiter la demande, la passer au suivant ou couper court à la chaîne.

--

Use case

Lorsque que vous voulez effectuer certaines opérations et/ou vérifications sur un objet dans un ordre donné.

chain of responsibility



merci !

une question ?

| ddemots@myges.fr

Sources

- refactoring.guru