PATTERN COMMAND

Présenté par Léo Elian Hippolyte et Romain

Sommaire

Qu’est-ce qu’un design pattern ?

Un **design pattern** est une solution éprouvée à des problèmes récurrents rencontrés dans la conception logicielle. Il rend les logiciels plus **flexibles**, **évolutifs** et **faciles à maintenir**.

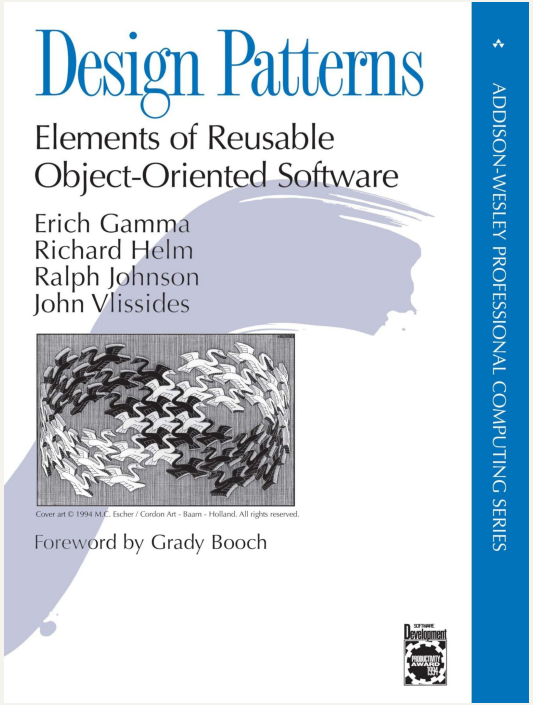
Il existe trois types principaux de design patterns :

Créationnels : pour gérer la création des objets.

Structuraux : pour organiser les relations entre les objets.

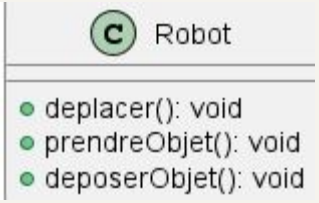
Comportementaux : pour définir comment les objets interagissent.

Le design pattern command est un pattern comportemental

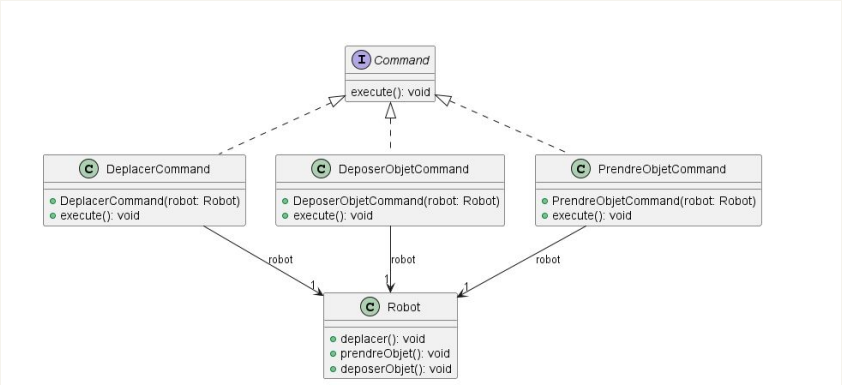


Énoncé d’un besoin :

On souhaite développer une application pour gérer les commandes d'un robot domestique. Le robot peut effectuer différentes actions, comme **se déplacer**, **prendre un objet**, ou **le déposer**. Ces actions peuvent être lancées depuis une interface utilisateur.



Modélisation résolvant le problème



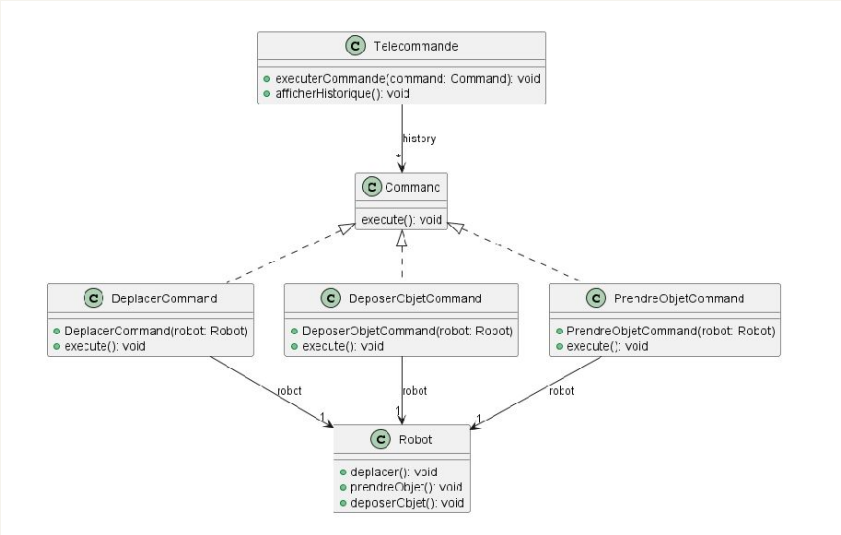
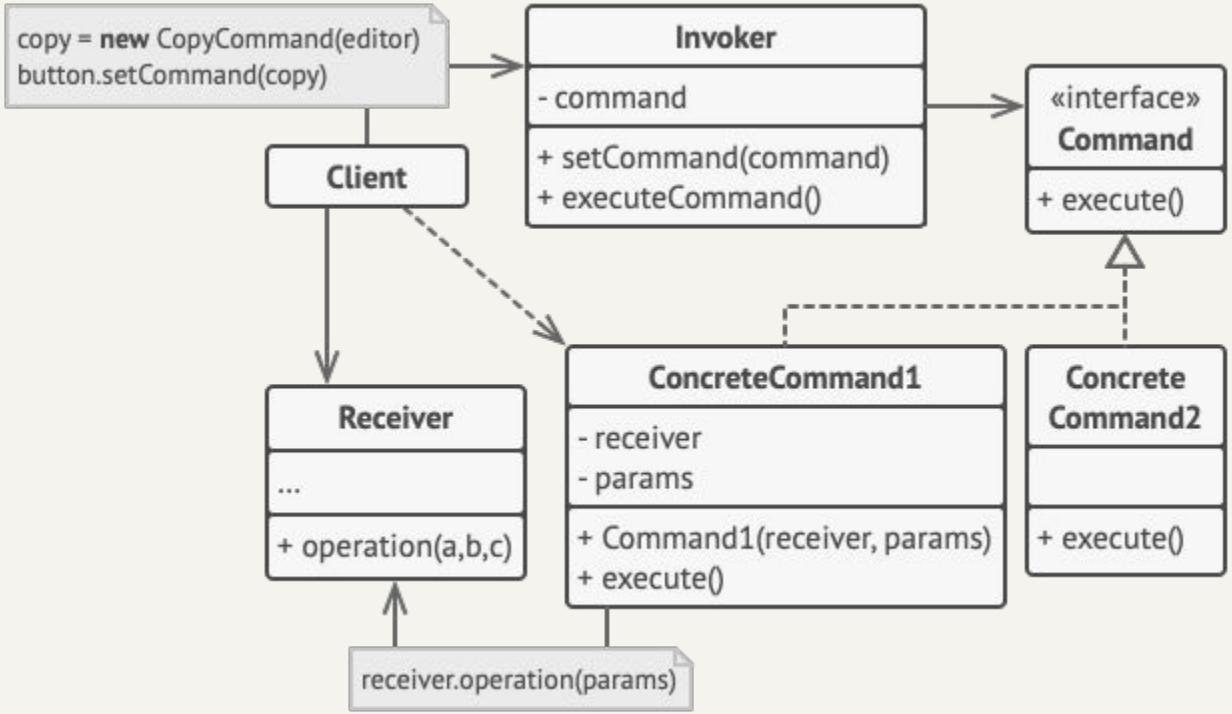


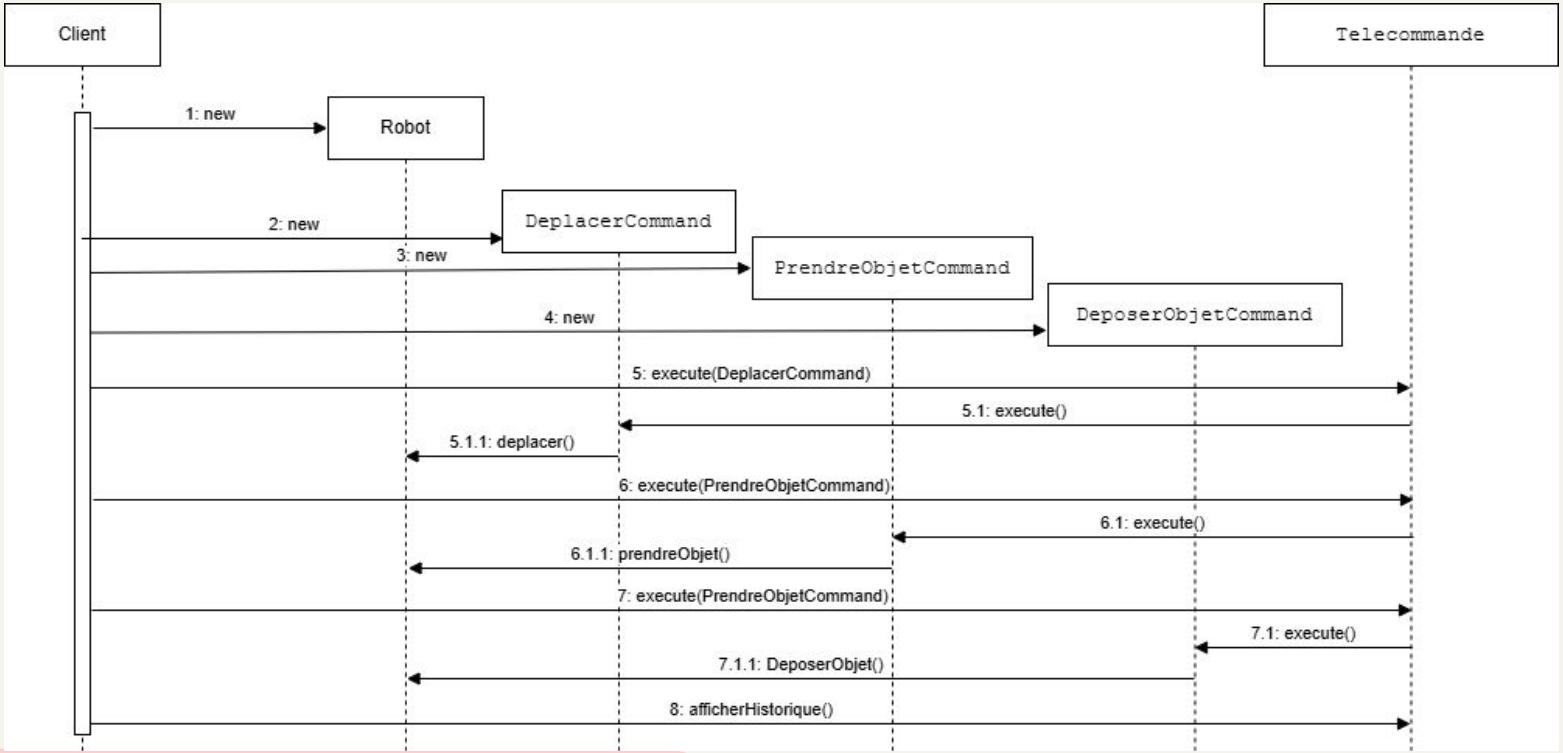
Diagramme de classe général



Pattern Command

Le pattern Command est un pattern comportemental qui transforme une requête en objet. Cela permet de paramétrer des actions à exécuter, de les enchaîner, et de les annuler facilement si besoin.

Diagramme de séquence



Le rôle des classes

- **Receiver** = Robot
- **Invoker** = Telecommande
- **Command** = Command
- **ConcreteCommand**: DeplacerCommand, DeposerObjetCommand, PrendreObjetCommand

Principes **SOLID**

SRP
(Single Responsibility Principle)

Une classe doit avoir une seule responsabilité.

Chaque commande a sa propre classe, donc elle se concentre uniquement sur une action spécifique.

OCP
(Open/Closed Principle)

Une classe doit être ouverte aux extensions sans modifier le code existant.

On peut ajouter de nouvelles commandes sans toucher aux autres.

LSP
(Lyskov Substitution Principle)

Les sous-types doivent pouvoir être substitués à leurs types de base.

Les commandes concrètes implémentent la même interface Commande

ISP
(Interface Segregation Principle)

Un client ne devrait jamais être forcé de dépendre d'une interface qu'il n'utilise pas.

La seul interface utilisé est l'interface command qui ne possède qu'une méthode

DIP
(Dependency Inversion Principle)

Les modules de haut niveau ne doivent pas dépendre des modules de bas niveau.

L'Invoker dépend de l'interface Command, pas des commandes concrètes.

Limites

Complexité accrue : Implémenter une commande pour chaque action peut ajouter de la complexité.

Surcharge mémoire : Conserver l'historique des commandes pour annuler/répéter peut être gourmand en mémoire

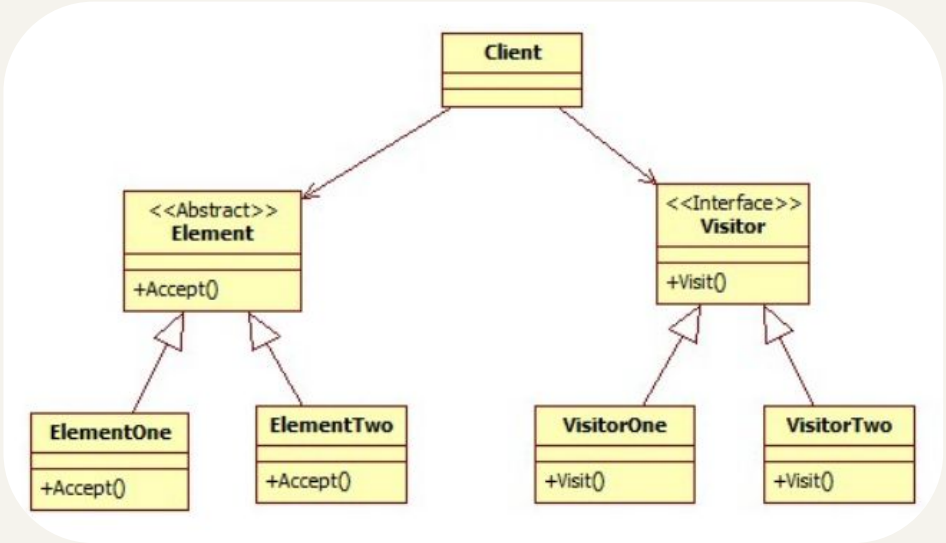
Relation avec le pattern Visiteur

Similarité :

- **sépare le code de traitement de l'objet** dans lequel il est utilisé.
- Ils peuvent être utilisés ensemble

Différences :

- **Intention**
- **L'objectif**



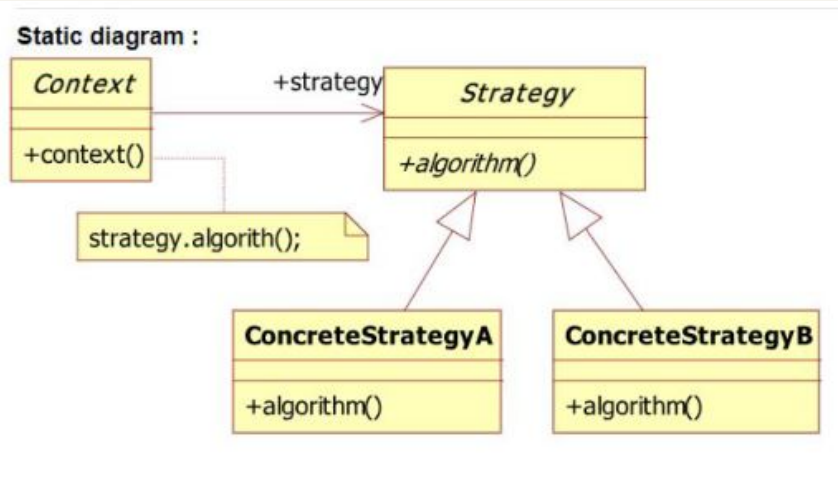
Relation avec le pattern Stratégie

Similarité :

- Les deux patterns encapsulent une action ou une logique de traitement dans un objet distinct
- Les deux patterns apportent de la flexibilité au programme.

Différences :

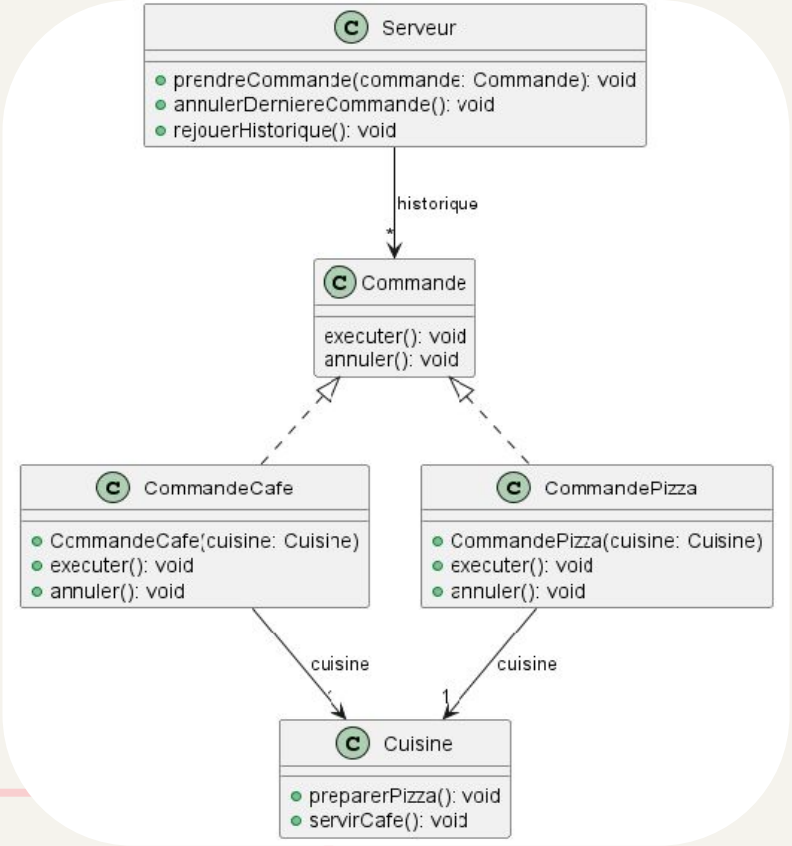
- **Intention**



Deuxième exemple

Restaurant où l'on peut faire une commande, avoir la liste des commandes passées et une possibilité d'annuler la dernière commande

Diagramme de classes



Code

```java
public interface Commande {
    void executer();
    void annuler();
}
```

```java
public class Cuisine {
    public void preparerPizza() {
        System.out.println("Préparation d'une pizza.");
    }

    public void servirCafe() {
        System.out.println("Service d'un café.");
    }
    public void annulerPizza() {
        System.out.println("Annulation de la préparation de la pizza.");

    }
    public void annulerCafe() {
        System.out.println("Annulation du service du café");
    }
}
```

```java
public class CommandePizza implements Commande {

    private Cuisine cuisine;

    public CommandePizza(Cuisine cuisine) {
        this.cuisine=cuisine;
    }

    @Override
    public void executer() {
        cuisine.preparerPizza();
    }

    @Override
    public void annuler() {
        cuisine.annulerPizza();
    }
}
```

```java
public class CommandeCafe implements Commande{
    private Cuisine cuisine;

    public CommandeCafe(Cuisine cuisine) {
        this.cuisine = cuisine;
    }

    @Override
    public void executer() {
        cuisine.servirCafe();

    }

    @Override
    public void annuler() {
        cuisine.annulerCafe();
    }

}
```

```java
import java.util.ArrayList;
import java.util.List;

public class Serveur {
    private List<Commande> historique = new ArrayList<>();

    public void prendreCommande (Commande commande) {
        historique.add(commande);
        commande.executer();

    }

    public void annulerDerniereCommande() {
        if (!historique.isEmpty()) {
            Commande dernieresCommande = historique.remove(historique.size() - 1);
            dernieresCommande.annuler();
        } else {
            System.out.println("Aucune commande à annuler.");
        }
    }

    public void rejouerHistorique() {
        for (Commande commande : historique) {
            commande.executer();
        }
    }
}
```

```java
public static void main(String[] args) {

    Cuisine cuisine = new Cuisine();

    Commande pizza = new CommandePizza(cuisine);
    Commande cafe = new CommandeCafe(cuisine);

    Serveur serveur = new Serveur();

    serveur.prendreCommande(pizza);
    serveur.prendreCommande(cafe);
    System.out.println("");

    System.out.println("L'historique des commandes est le suivant:");
    serveur.rejouerHistorique();
    System.out.println("");



    serveur.annulerDerniereCommande();
    System.out.println("");
    System.out.println("L'historique des commandes est le suivant:");
    serveur.rejouerHistorique();
```

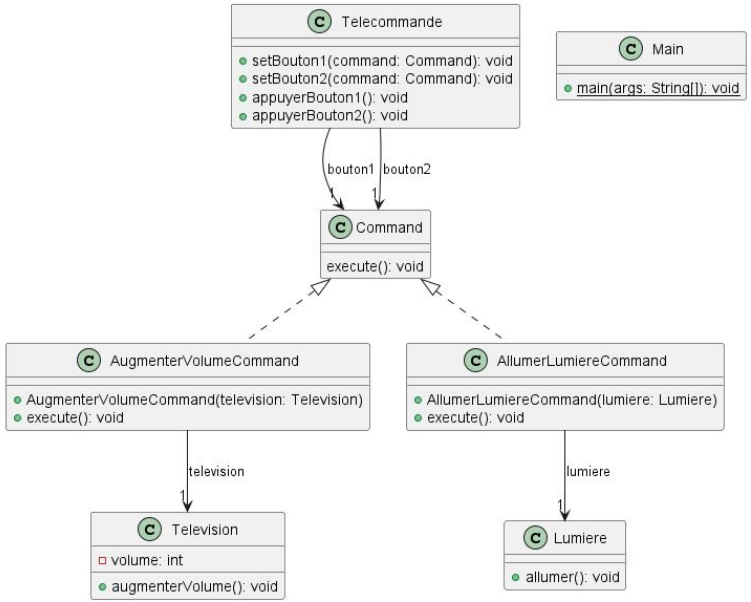
```
Préparation d'une pizza.
Service d'un café.

L'historique des commandes est le suivant:
Préparation d'une pizza.
Service d'un café.

Annulation du service du café

L'historique des commandes est le suivant:
Préparation d'une pizza.
```

Live **Coding**



Lien : https://youtu.be/qAG1jGon5Z8


Kahoot!

Bibliographie

- https://www.youtube.com/watch?v=UfGD60BYzPM
- https://chatgpt.com/
- https://refactoring.guru/fr/design-patterns/command
- https://medium.com/@kapildas_37013/command-pattern-and-solid-principles-a81928eb0207
- https://en.wikipedia.org/wiki/Command_pattern