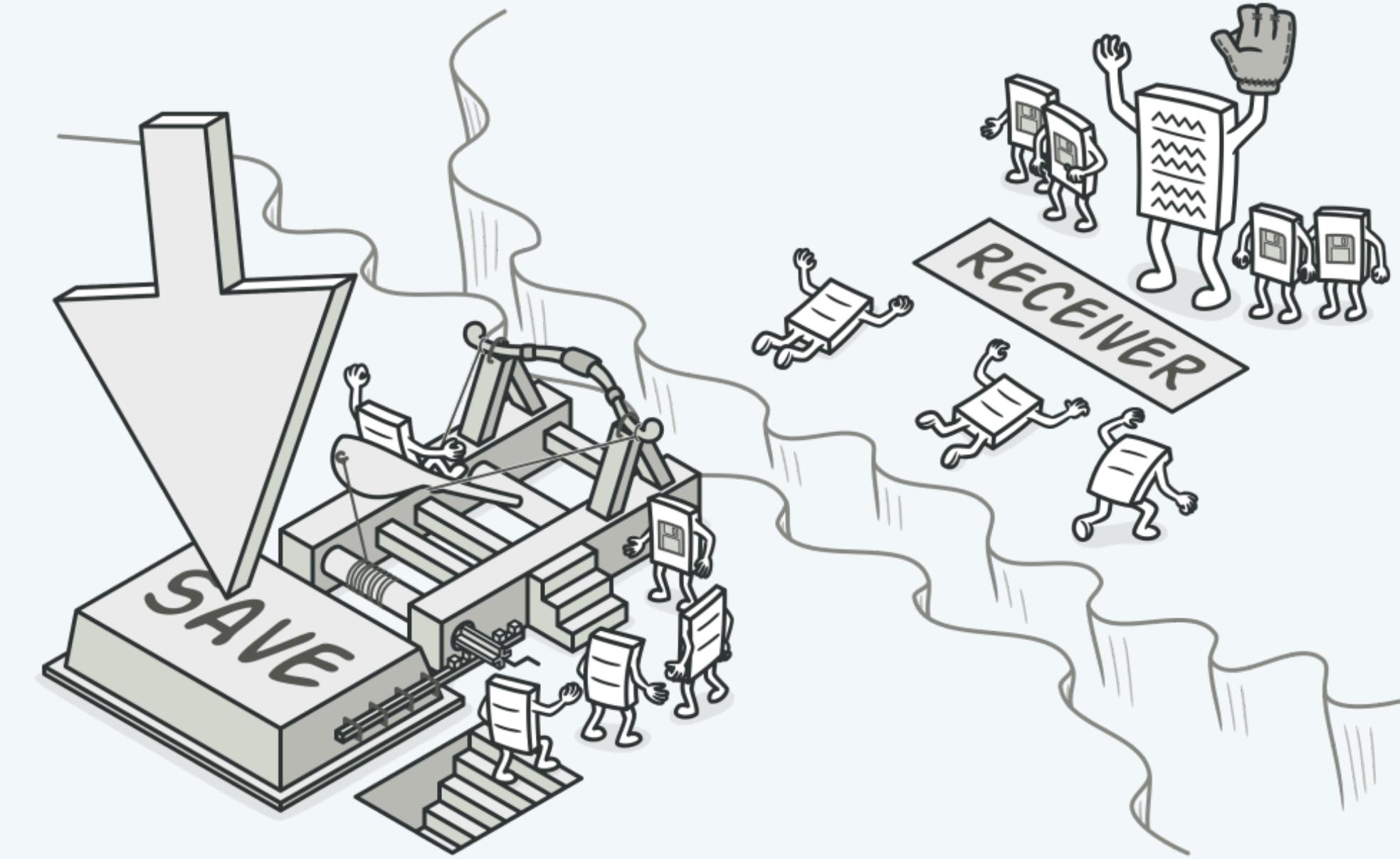


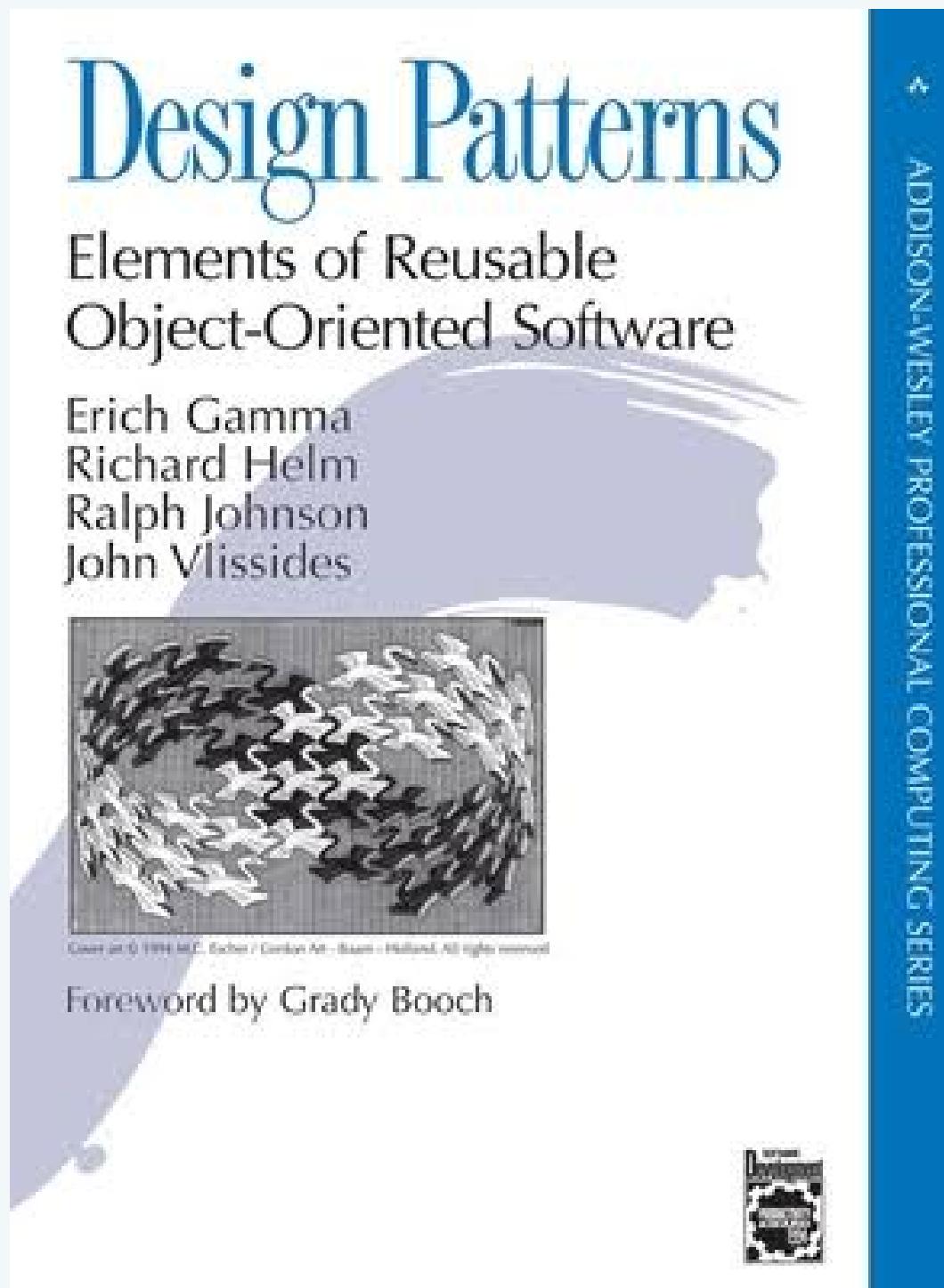
Pattern command



S O M M A I R E

- 1 : Introduction sur GOF
- 2 : Énoncé d'un besoin
- 3 : Modélisation du besoin
- 4 : Modélisation résolvant le problème
- 5 : Généralisation du diagramme
- 6 : Nom / type / intention du pattern
- 7 : Diagramme de classe et de séquence
- 8 : SOLID
- 9 : Limite du pattern
- 10 : Lien avec les autres pattern
- 11 : 2ème exemple
- 12 : Live coding
- 13 : QCM
- 14 : Bibliographie

Les patterns du GoF

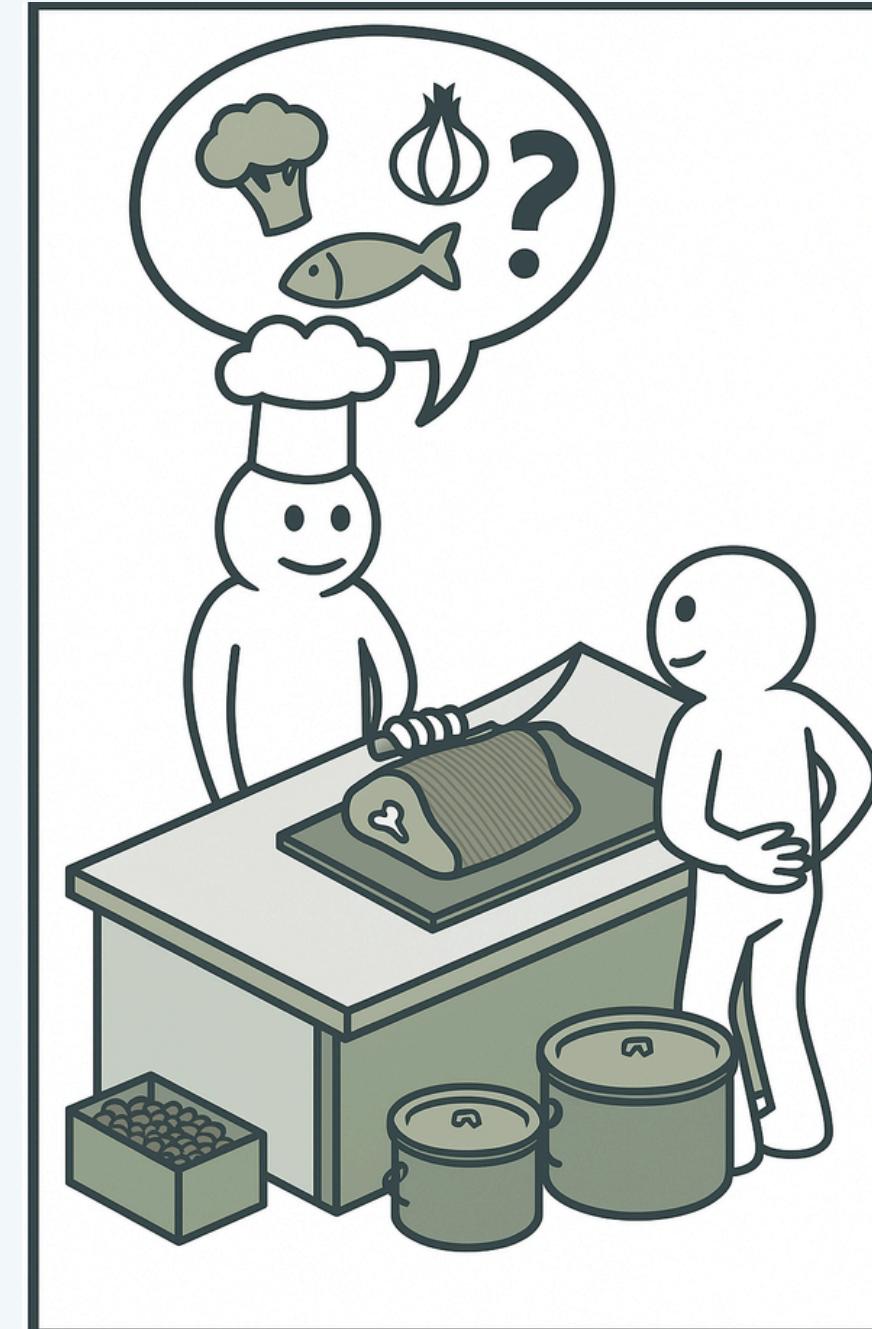
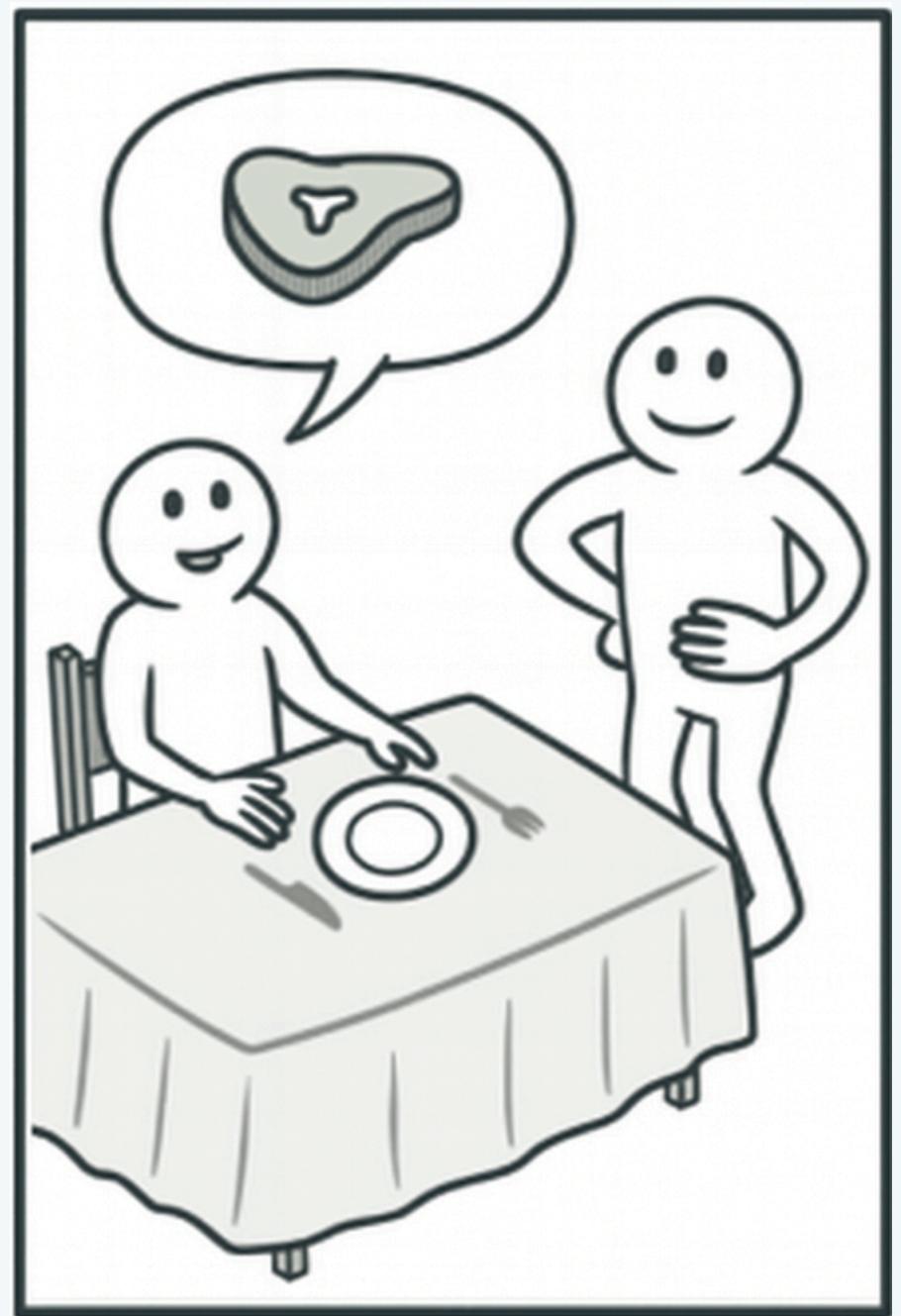


Publié en 1994

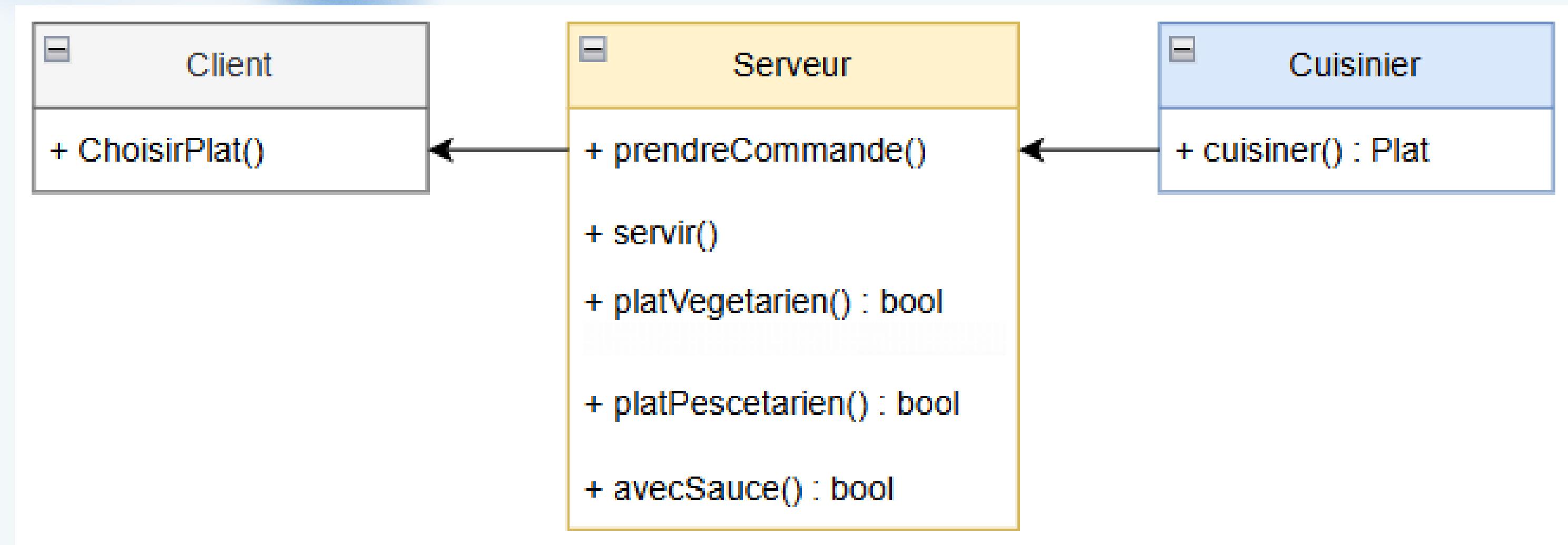
Bonne pratique
du développer

Solutions à des problèmes
récurrents

Énoncé d'un besoin



Modélisation du besoin



Les problèmes de cette modélisation :

- **Pas d'objet représentant la commande**

- Tout passe directement du Client au Serveur, puis du Serveur au Cuisinier.
- Rien ne formalise la demande du client (ex. le plat choisi, les options...).

- **Responsabilités mal réparties**

- Le Serveur fait trop de choses : il prend la commande, mémorise les choix, transmet, sert, etc.
- Le Cuisinier dépend du Serveur pour obtenir les détails au lieu de travailler à partir d'un ordre clair.

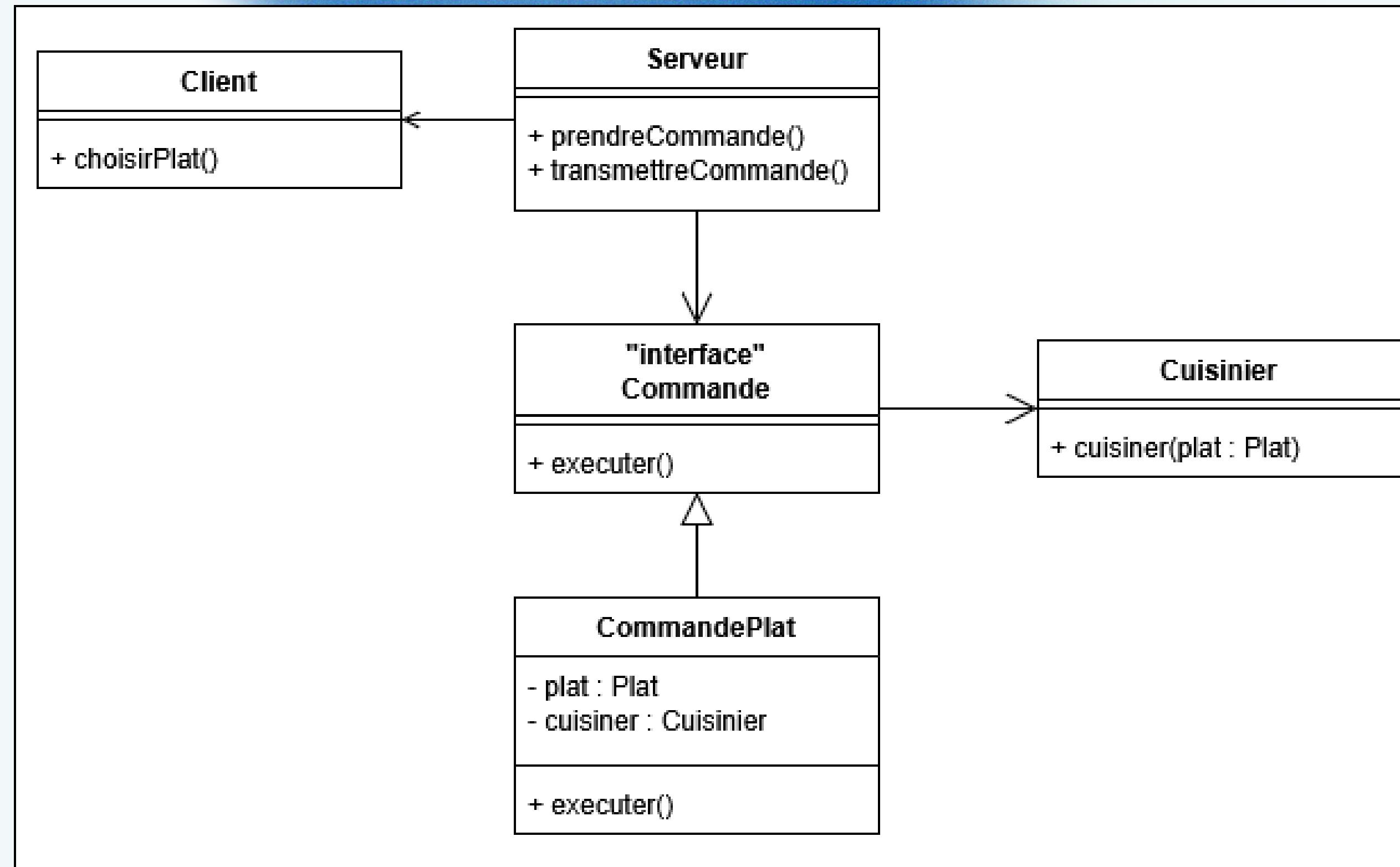
- **Absence de traçabilité ou de persistance**

- Le Plat n'est jamais explicitement associé à une commande.
- Si plusieurs clients commandent, impossible de savoir quel plat correspond à qui.

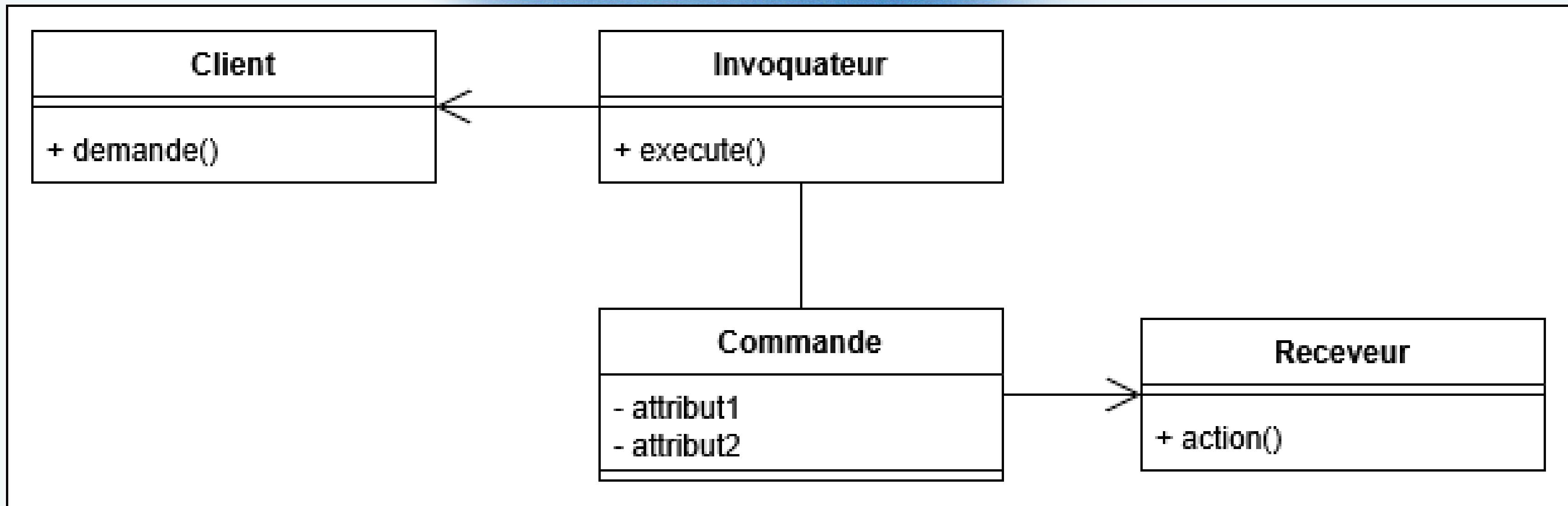
- **Couplage fort entre Serveur et Cuisinier**

- Le Serveur appelle directement cuisiner() du Cuisinier : il n'y a pas d'abstraction.
- Si le mode de préparation change, il faut modifier le code du serveur.

Modélisation qui résout ces problèmes :



Généralisation du diagramme :



Problématique et intentions du pattern

Le design pattern Commande est une approche qui prend une action à réaliser et la convertit en un objet autonome qui encapsule tous les détails de cette action.

Cette conversion permet de paramétriser des méthodes avec différentes actions, de planifier leur exécution, de les mettre en file d'attente ou d'annuler des opérations déjà effectuées.

Diagramme de classe

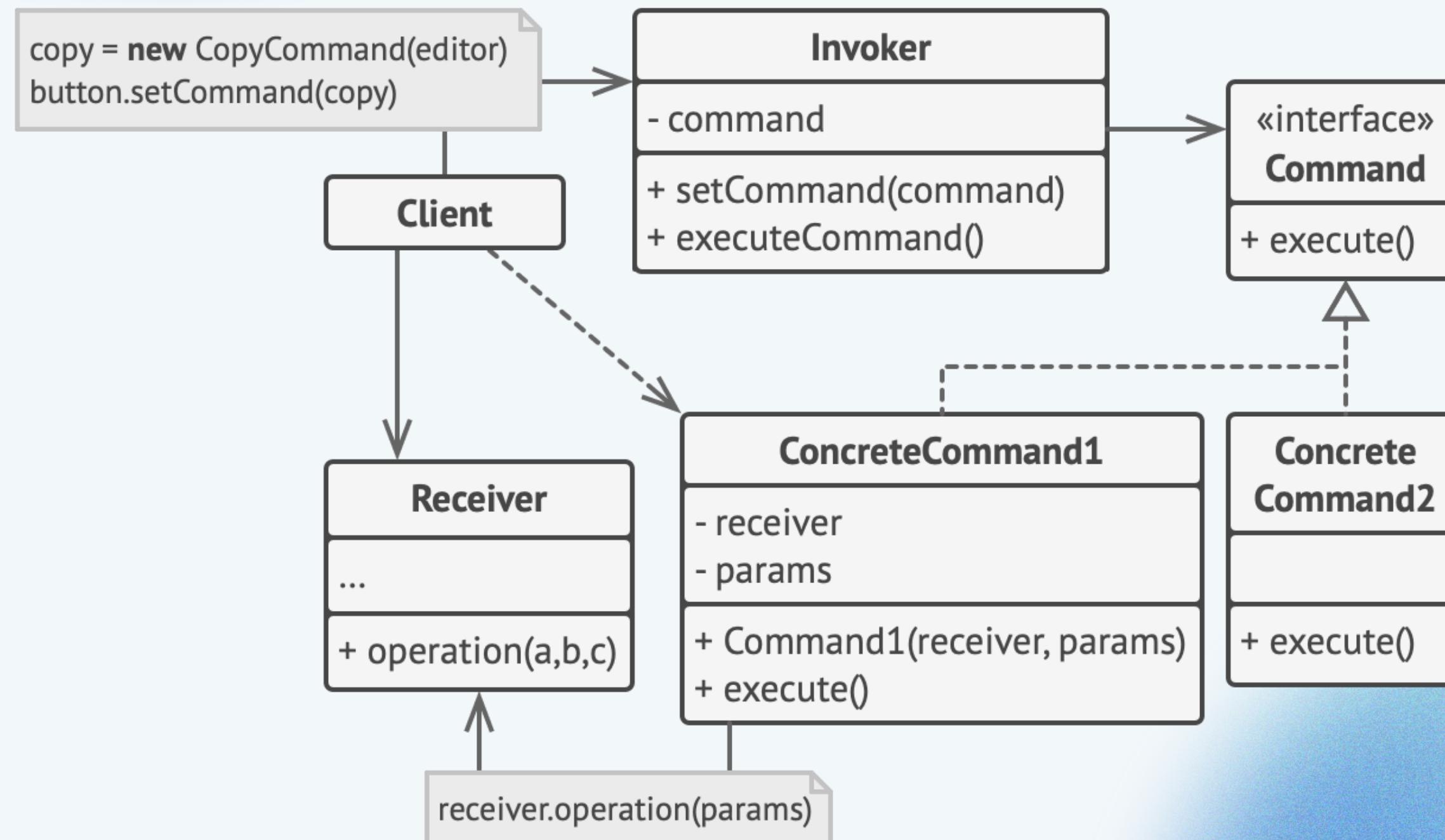
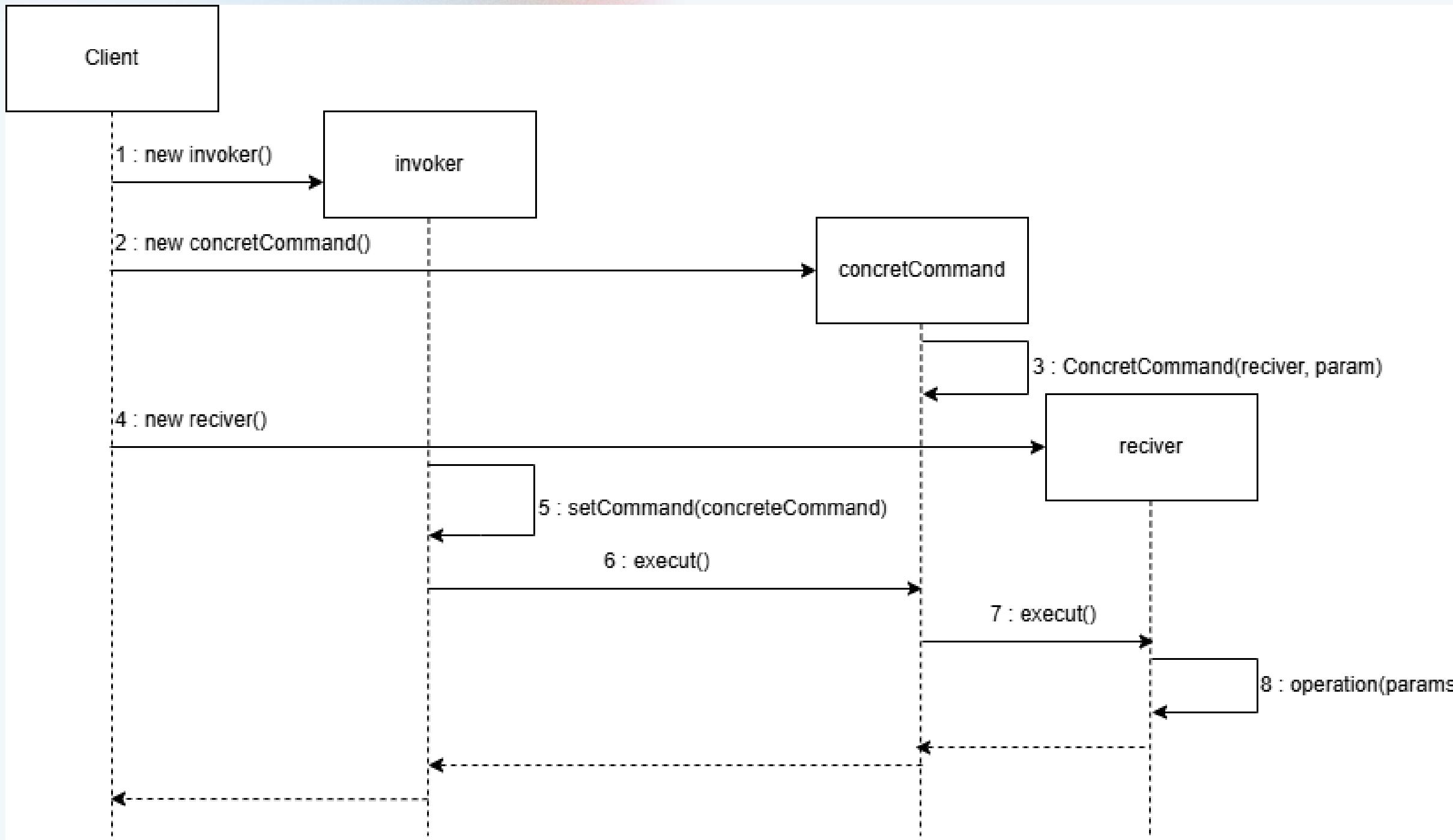


Diagramme de séquence



Classes participantes

client : Le client crée et configure les objets des commandes concrètes. Il les configure en y mettant les paramètres nécessaires. C'est lui qui effectue les commandes avec les objets créés.

invoker : L'invoker est responsable de l'envoi des commandes, càd qu'il va déclencher la commande que le client lui a mis en paramètre indépendamment de ce même paramètre. Il transmet juste le fait d'exécuter une commande.

command : l'interface command sert juste à déclarer la méthode execute.

concrete command : elle permet d'exécuter la commande d'un objet métier. Elle prend en paramètre un receiver qui va faire son opération.

receiver : c'est lui qui exécute la commande avec les paramètres que la commande concrète lui a transmis.

SOLID

SRP : une classe à une responsabilité. Chaque classe a sa responsabilité propre.

OCP : ouvert aux extensions, fermées aux modifications. On peut ajouter une commande sans endommager le code existant.

LSP : Les sous types doivent pouvoir être substitué à leur type de base. Respecté, car il n'y a aucun héritage

ISP : Un client ne doit pas être forcé d'utiliser un service qu'il n'a pas besoin. Respecté, car il n'y a qu'un seul interface avec une méthode qui est implémentée par toutes les classes.

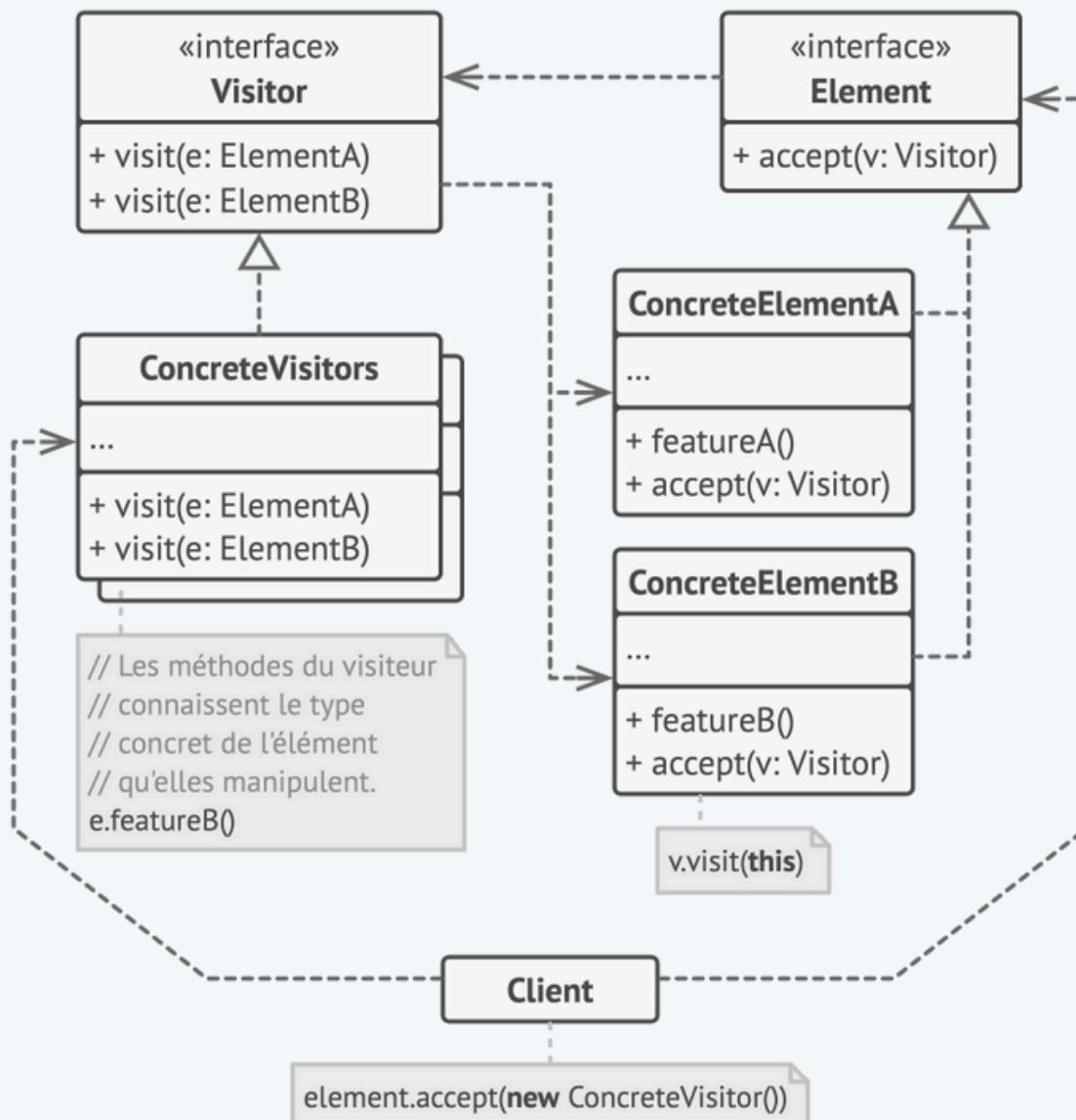
DIP : Respecté, car il n'y a pas de module de bas ou de haut niveau.

Limites du pattern

- Compliquer à lire
- Multiplication des classes
- Complexité : création invoker → concret command → receiver → chaque classe effectue son opération
- Difficile de débugger

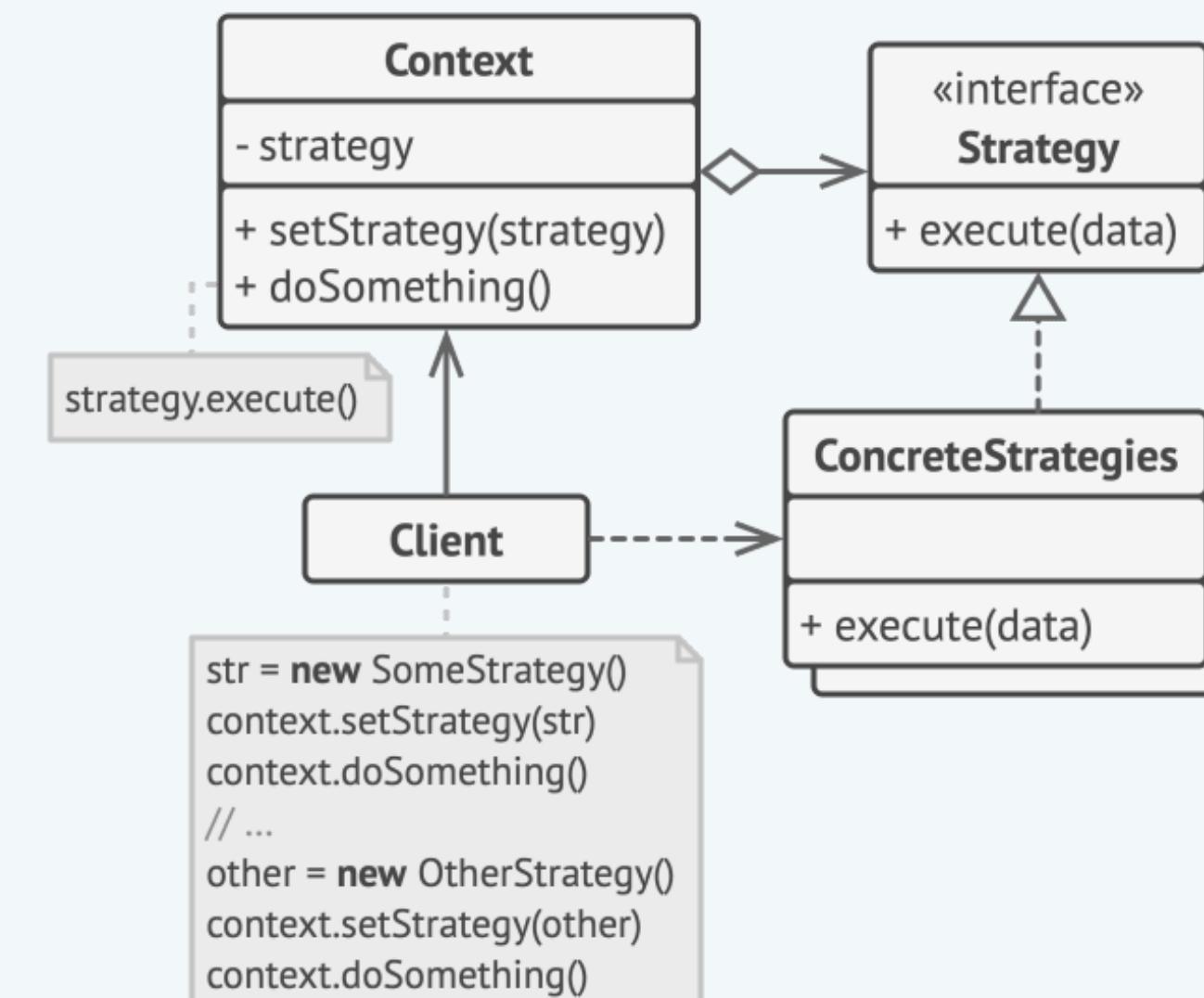
Lien avec d'autres pattern

Visitor



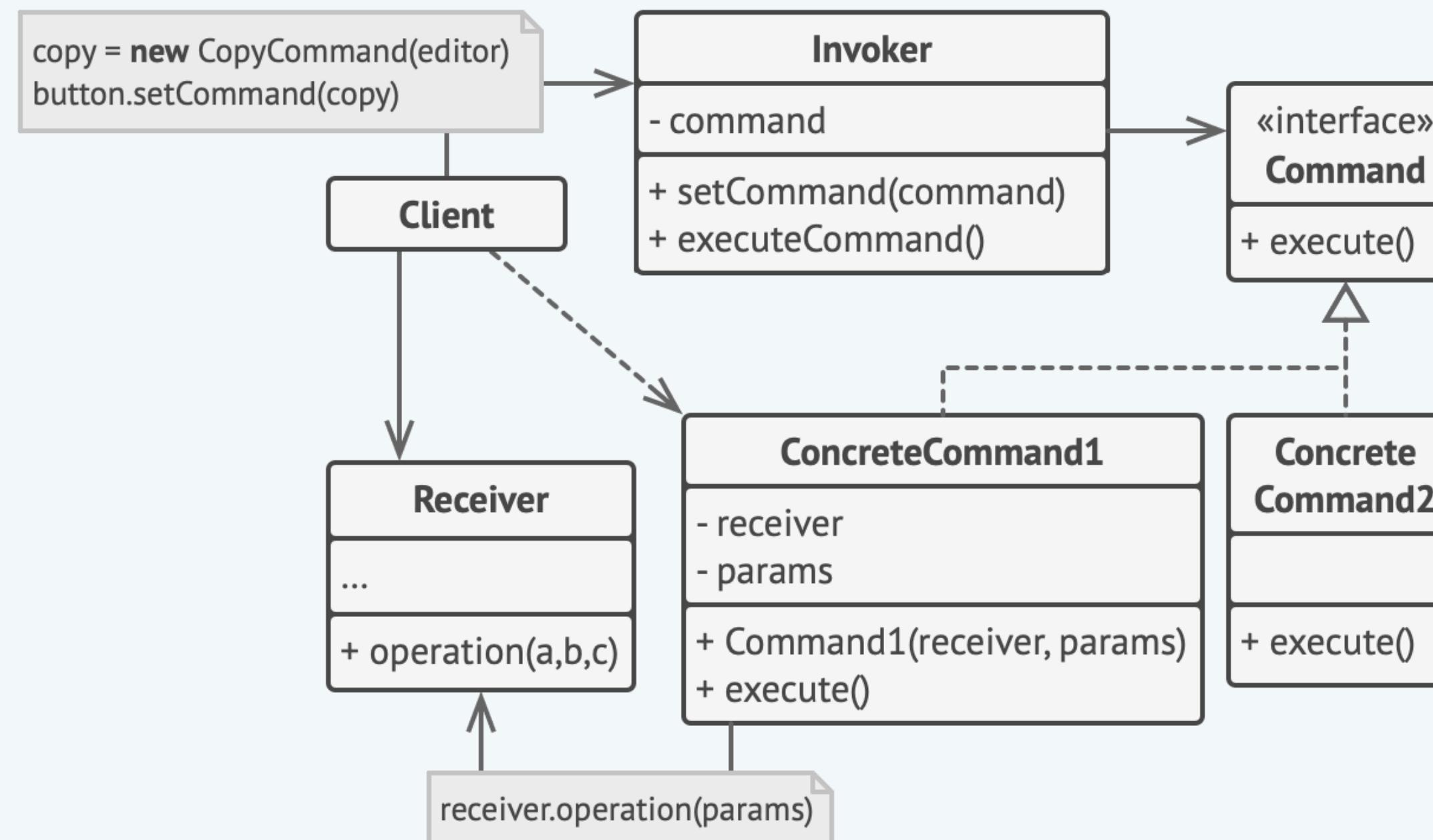
Visiteur = généralisation du command. Le pattern visiteur donne un ensemble de commande que chaque objet implémente à sa manière alors que le pattern command donne une commande par objet.

strategy



Les deux patterns ont une structure similaire. Le pattern command encapsule un objet pour en faire une action alors que le strategy encapsule un algorithme pour en faire une action. La différence est l'intention du pattern.

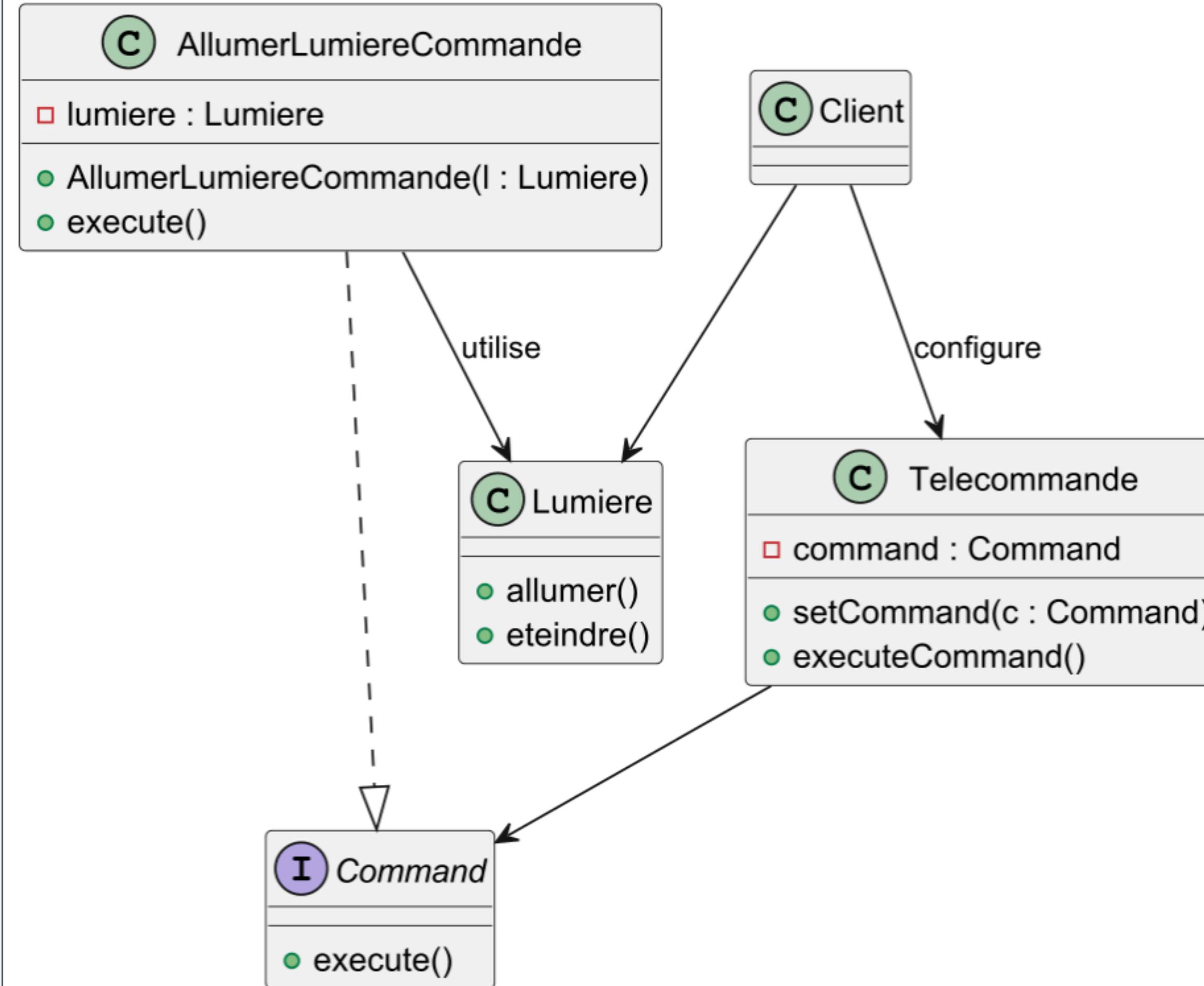
Deuxième exemple : Gestion d'un système de domotique



Qu'est-ce qu'il nous faut pour un système comme celui-ci ?

- Un objet permettant d'enregister les commandes
(Smartphone, télécommande)
- Les objets qui doivent changer d'état (Lumière, volet, aspirateur, LED...)
- Les commandes qui doivent s'appliquer pour réaliser les actions

Pattern Commande - Exemple Domotique



```
class Lumiere { 4 usages
    public void allumer() {
        System.out.println("💡 La lumière est allumée.");
    }
}
```

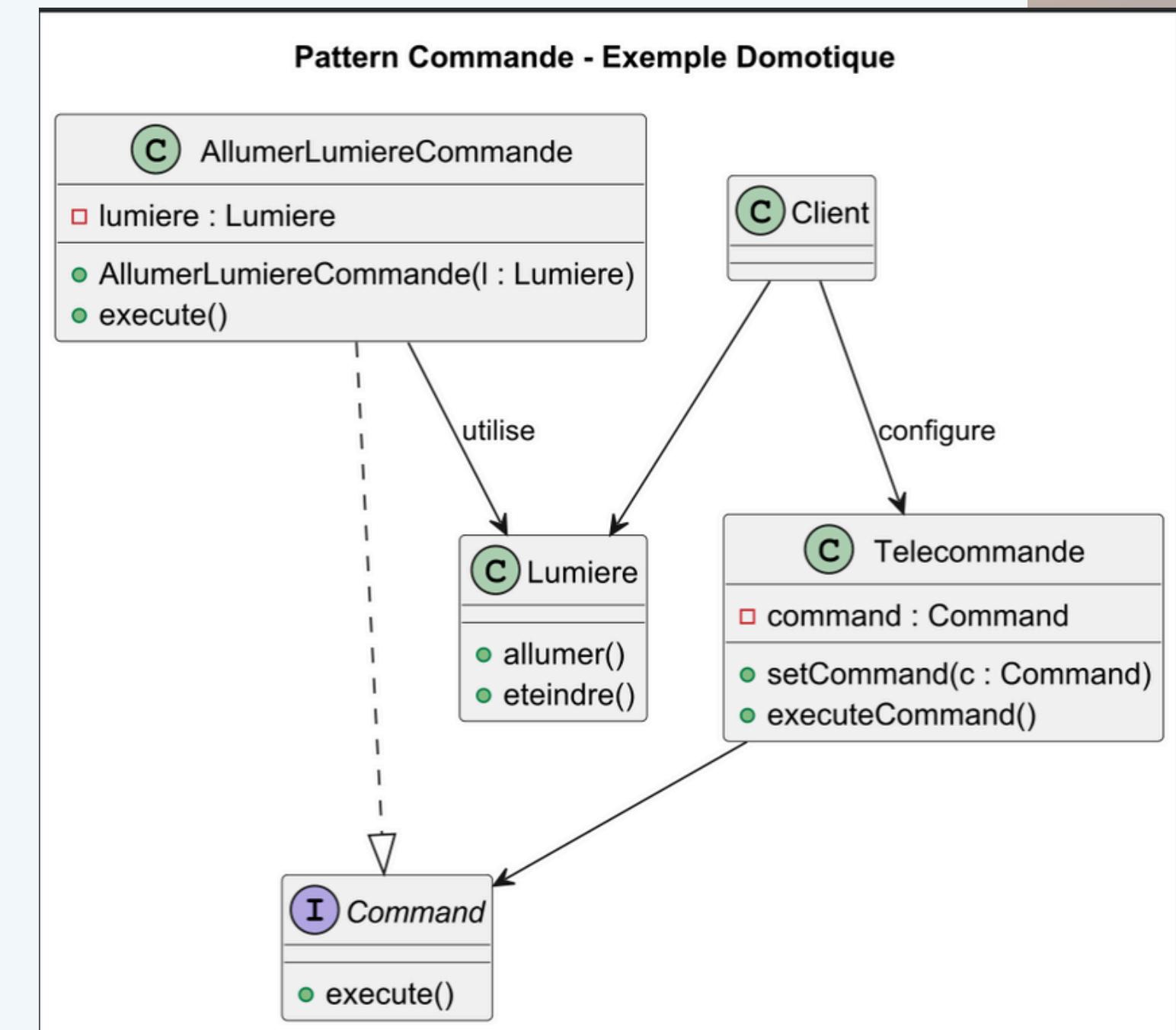
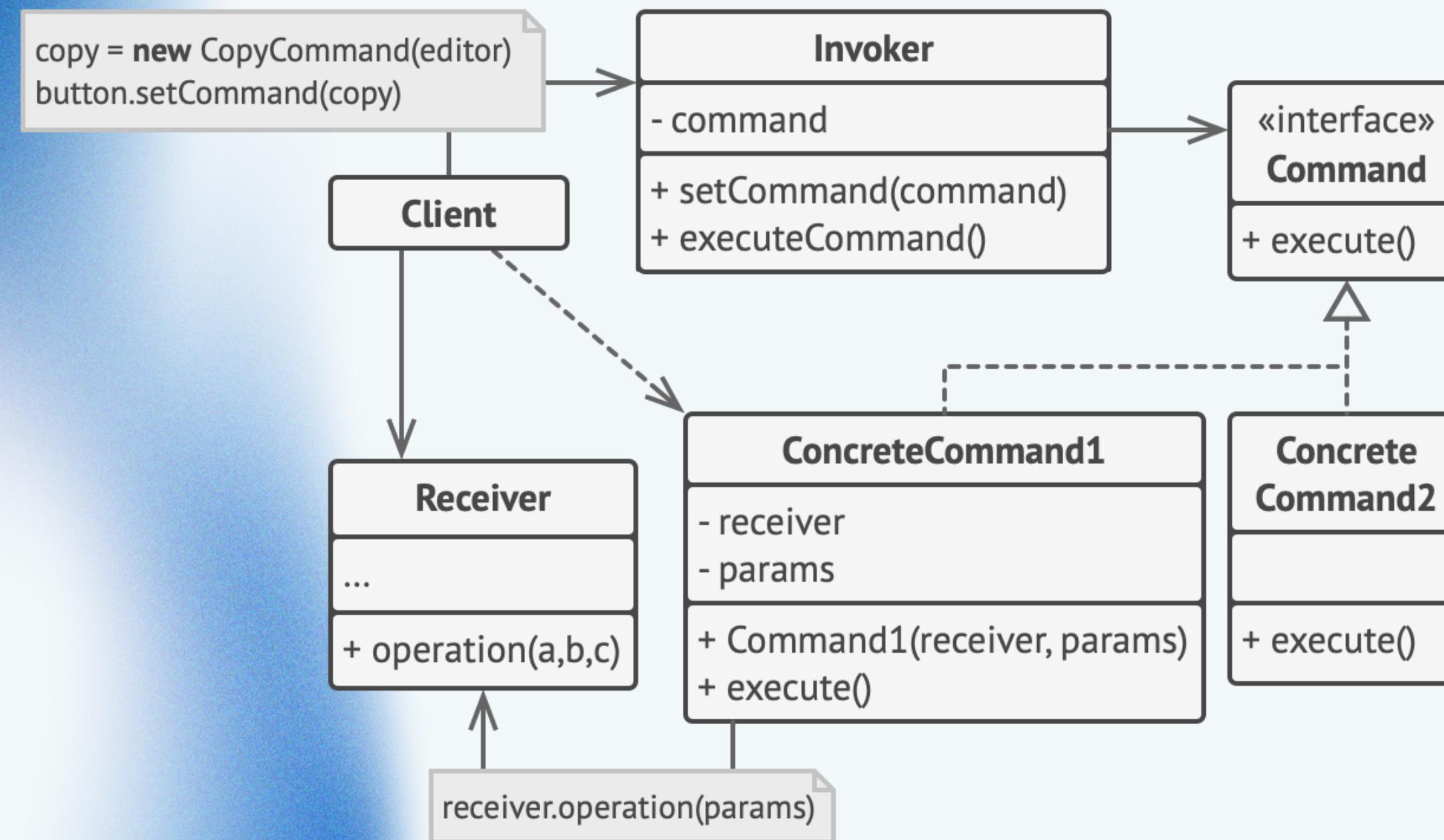
```
class AllumerLumiereCommande implements Command { 1
    private Lumiere lumiere; 2 usages

    public AllumerLumiereCommande(Lumiere lumiere) {
        this.lumiere = lumiere;
    }

    public void execute() { 1 usage
        lumiere.allumer();
    }
}
```

```
public interface Command {  
    void execute();  
}  
public class Telecommande {  
    private Command command;  
  
    public void setCommand(Command command) {  
        this.command = command;  
    }  
  
    public void executeCommand() {  
        command.execute();  
    }  
}
```

```
public class Main {  
    public static void main(String[] args) {  
        Lumiere lumiereSalon = new Lumiere();  
        Command cmdAllumer = new AllumerLumiereCommande(lumiereSalon);  
        Telecommande telecommande = new Telecommande();  
        telecommande.setCommand(cmdAllumer);  
        telecommande.executeCommand();  
    }  
}
```





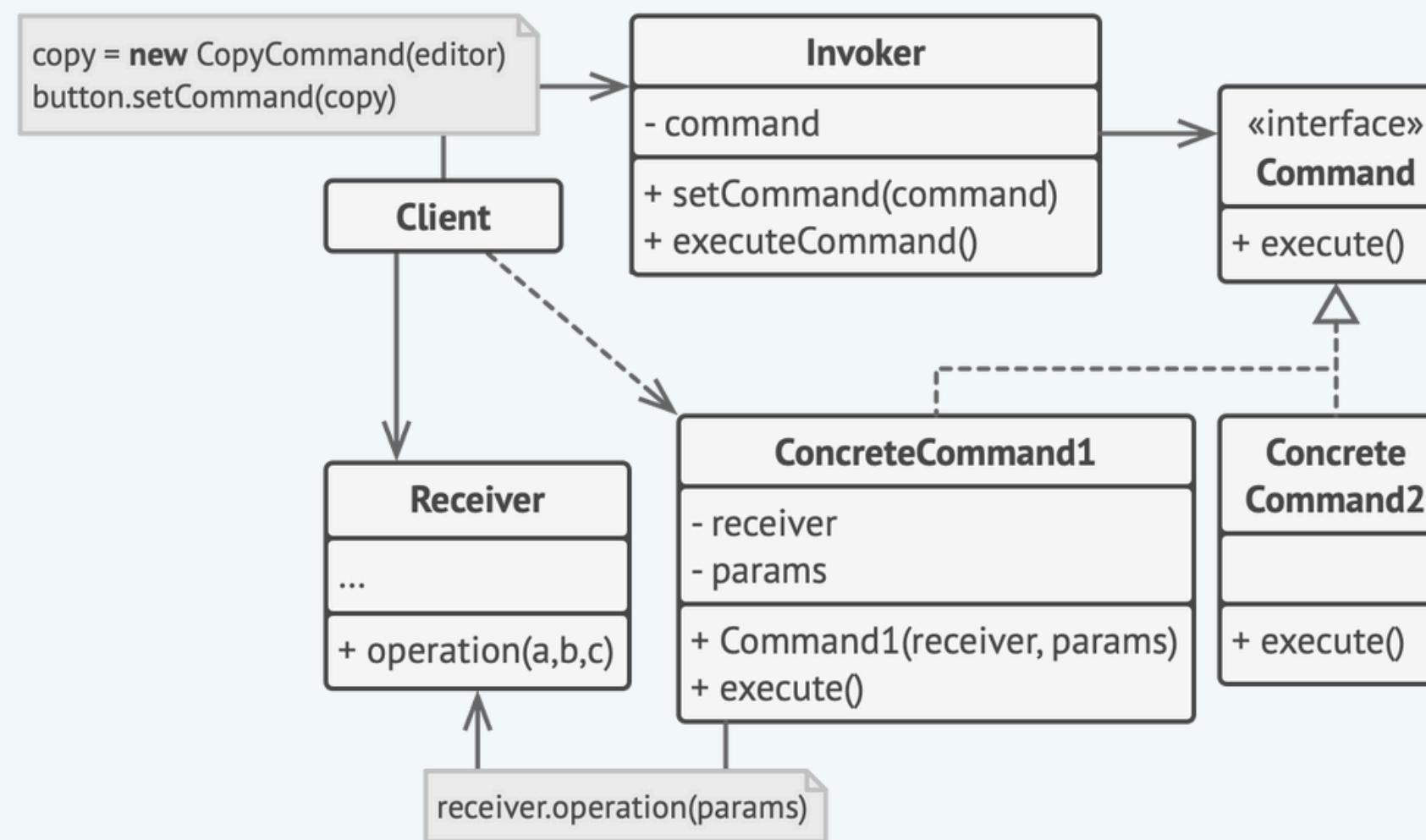
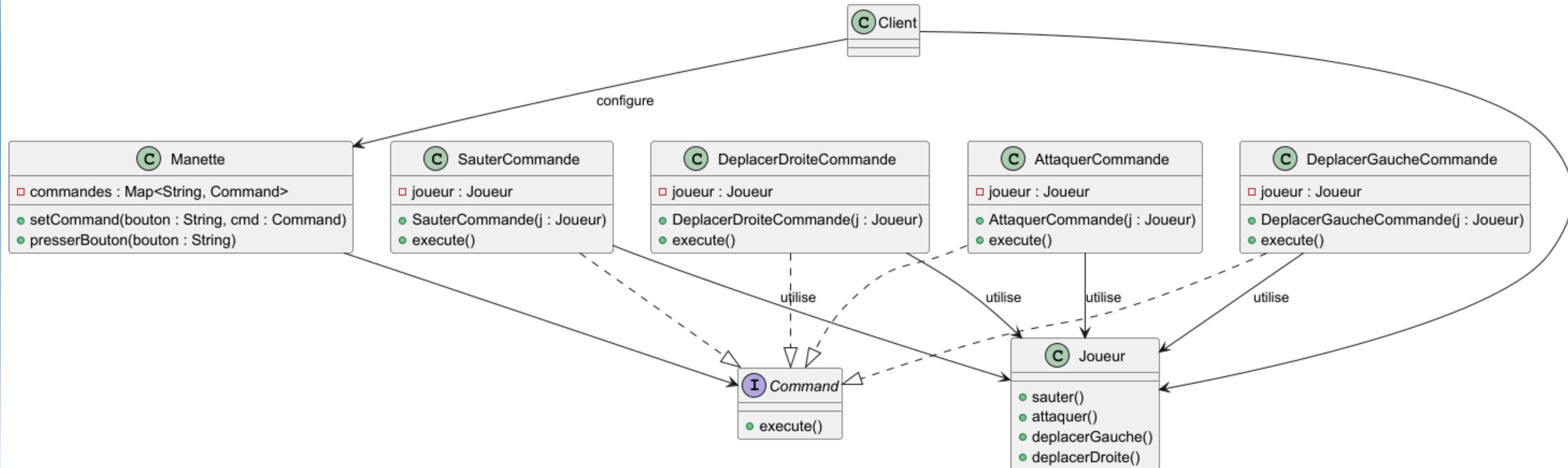
[Watch video on YouTube](#)

Error 153

Video player configuration error



Pattern Commande - Exemple Jeu Vidéo



Bibliographie/Webographie :

- refactoring.guru
- wikipedia.fr
- koor.fr
- sfeir.dev
- goprod.bouhours.net



THANK YOU

