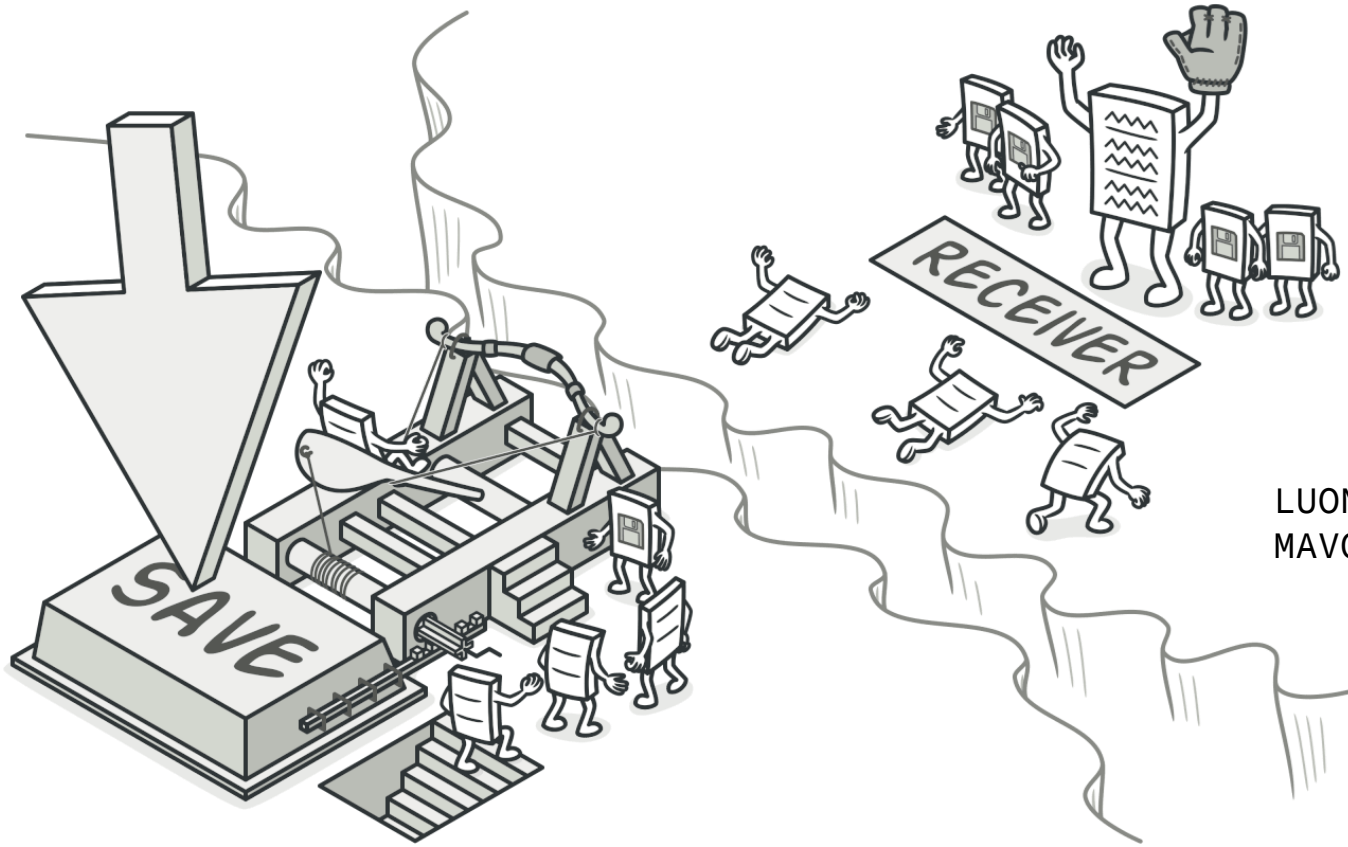


DESIGN PATTERN : COMMAND



LUONG Tristan
MAVOUNGOU Ken

SOMMAIRE

- I. Présentation du design pattern
- II. Cas d'utilisation
- III. Exemple concret
- IV. Implémentation
- V. Conclusion

I. PRÉSENTATION DU DESIGN PATTERN

A stylized illustration of a human head in profile, facing right. The head is composed of various geometric shapes and is filled with a light gray color. Inside the head, there are several colorful gears (blue, red, yellow, and white) and a stack of white rectangular blocks, representing the internal workings of a design pattern.

Caractéristiques : Encapsuler des commandes ou des requêtes dans d'autres commandes ou requêtes.

But : Émettre des requêtes sur un objet dont les caractéristiques et les fonctions sont inconnus.

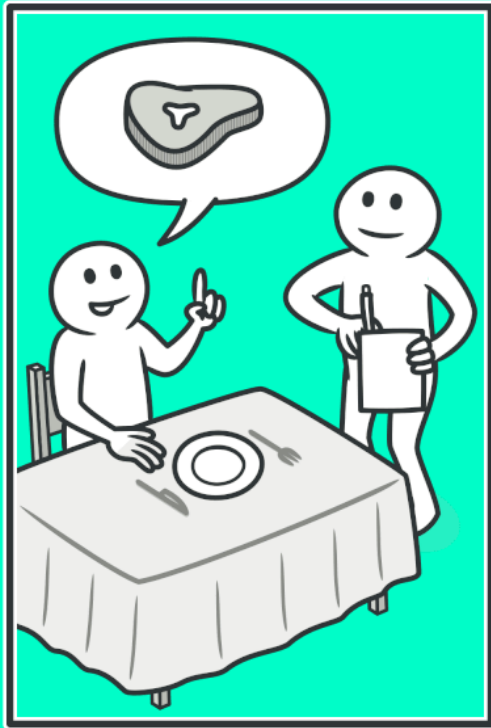
C'est une couche d'abstraction qui permet de garantir l'intégrité des données contenues dans un objet.

Résultat -> Moins d'abstraction : Qui permet une meilleure maintenabilité du code et permet d'étendre facilement un projet.

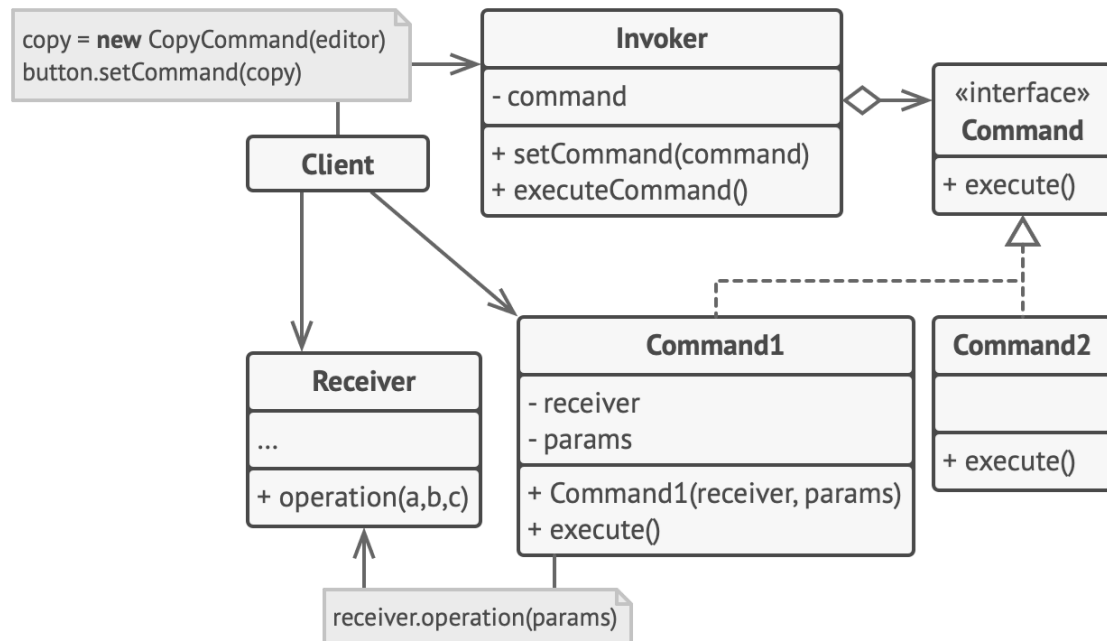
II. CAS D'UTILISATIONS

- Créer une pile d'exécution de fonctions -> **Effectuer des appels asynchrones.**
- Effectuer une des opérations "Undo Redo" car ce pattern permet d'enregistrer dans une pile l'ensemble des commandes précédentes. -> **Pattern le plus utilisé pour ce genre de traitement**
- Transformer un appel de méthode spécifique en un objet autonome. -> **Permet de passer des commandes en tant qu'argument de méthode, les stocker dans d'autres objets, etc...**

III. EXEMPLE CONCRET



IMPLÉMENTATION



L'invoker est le déclencheur de la commande.

La Command regroupe les informations envoyés par l'invoker

Le Receiver contient la logique métier et reçoit la commande

EXEMPLE DE CODE : RECEIVER

```
public class Console {  
  
    public void on(){  
        System.out.println("La console est allumée");  
    }  
  
    public void off(){  
        System.out.println("La console est éteinte");  
    }  
  
    public void startGame(){  
        System.out.println("Jeu lancé");  
    }  
  
    public void stopGame(){  
        System.out.println("Jeu arrêté");  
    }  
  
}
```

```
public class Radio {  
  
    private int volume = 0;  
  
    public void on(){  
        System.out.println("La radio est allumée");  
    }  
  
    public void off(){  
        System.out.println("La radio est éteinte");  
    }  
  
    public void volumeUp(){  
        this.volume++;  
        System.out.println("Le volume est de : " + this.volume);  
    }  
  
    public void volumeDown(){  
        this.volume--;  
        System.out.println("Le volume est de : " + this.volume);  
    }  
  
}
```

EXEMPLE DE CODE: TÉLÉCOMMANDE

```
public class Telecommande {
    private ICommande bouton1;
    private ICommande bouton2;
    private ICommande bouton3;
    private ICommande bouton4;

    public Telecommande(ICommande btn1, ICommande btn2, ICommande btn3, ICommande btn4){
        this.bouton1 = btn1;
        this.bouton2 = btn2;
        this.bouton3 = btn3;
        this.bouton4 = btn4;
    }

    public Telecommande(ICommande btn1, ICommande btn2){
        this.bouton1 = btn1;
        this.bouton2 = btn2;
    }

    public void pressBtn1(){
        this.bouton1.execute();
    }

    public void pressBtn2(){
        this.bouton2.execute();
    }


    public void pressBtn3(){
        this.bouton3.execute();
    }

    public void pressBtn4(){
        this.bouton4.execute();
    }
}
```

```
public interface ICommande {

    public void execute();
    public void undo();

}
```



EXEMPLE DE CODE : COMMANDE

```
public class Commande_ConsoleOn implements ICommande{

    private Console console;

    public Commande_ConsoleOn(Console theConsole) {
        this.console = theConsole;
    }

    @Override
    public void execute() { ←
        this.console.on();
    }

    @Override
    public void undo() { ←
        this.console.off();
    }
}
```

```
public class Commande_RadioVolumeUp implements ICommande{

    private Radio radio;

    public Commande_RadioVolumeUp(Radio theRadio) {
        this.radio = theRadio;
    }

    @Override
    public void execute() {
        this.radio.volumeUp();
    }

    @Override
    public void undo() {
        this.radio.volumeDown();
    }
}
```

CONCLUSION

- Pattern idéal lorsque l'on agit sur un objet dont les paramètres sont inconnues
- Facilement extensible selon le besoin -> **Dans notre exemple, au lieu d'avoir 2 télécommandes, on aurait pu créer une troisième commande qui englobe les 2 précédentes afin d'avoir une télécommande globale.**



?

?

?

?



ANNEXES

- https://sourcemaking.com/design_patterns/command/ **SourceMaking**
- <https://www.codingame.com/playgrounds/36502/design-pattern-command/le-pattern-command>
CodinGame
- https://en.wikipedia.org/wiki/Command_pattern#:~:text=In%20object%2Doriented%20programming%2C%20the,event%20at%20a%20later%20time.&text=Four%20terms%20always%20associated%20with,%2C%20receiver%2C%20invoker%20and%20client. **Wikipédia**
- <https://refactoring.guru/design-patterns/command> **Refactoring Guru**
- <https://medium.com/elp-2018/command-design-pattern-quest-ce-78bc93fb0942> **Medium**