Mediator Pattern

Presenté par :

Papa Ibra NDIAYE

Mame Yande TRAORE

Daba MBAYE

Prof: Mr DIOP

Licence GLSI A

2020/2021

Introduction

Dans la programmation orientée objet, nous devrions toujours essayer de concevoir le système de telle sorte que les composants soient faiblement couplés et réutilisables . Cette approche rend notre code plus facile à maintenir et à tester.

Dans la vraie vie, cependant, nous devons souvent faire face à un ensemble complexe d'objets dépendants. C'est à ce moment-là que le modèle de médiateur peut s'avérer utile.

L'intention du modèle de médiateur est de réduire la complexité et les dépendances entre les objets étroitement couplés communiquant directement les uns avec les autres . Ceci est réalisé en créant un objet médiateur qui prend en charge l'interaction entre les objets dépendants. Par conséquent, toute la communication passe par le médiateur.

Contexte d’utilisation

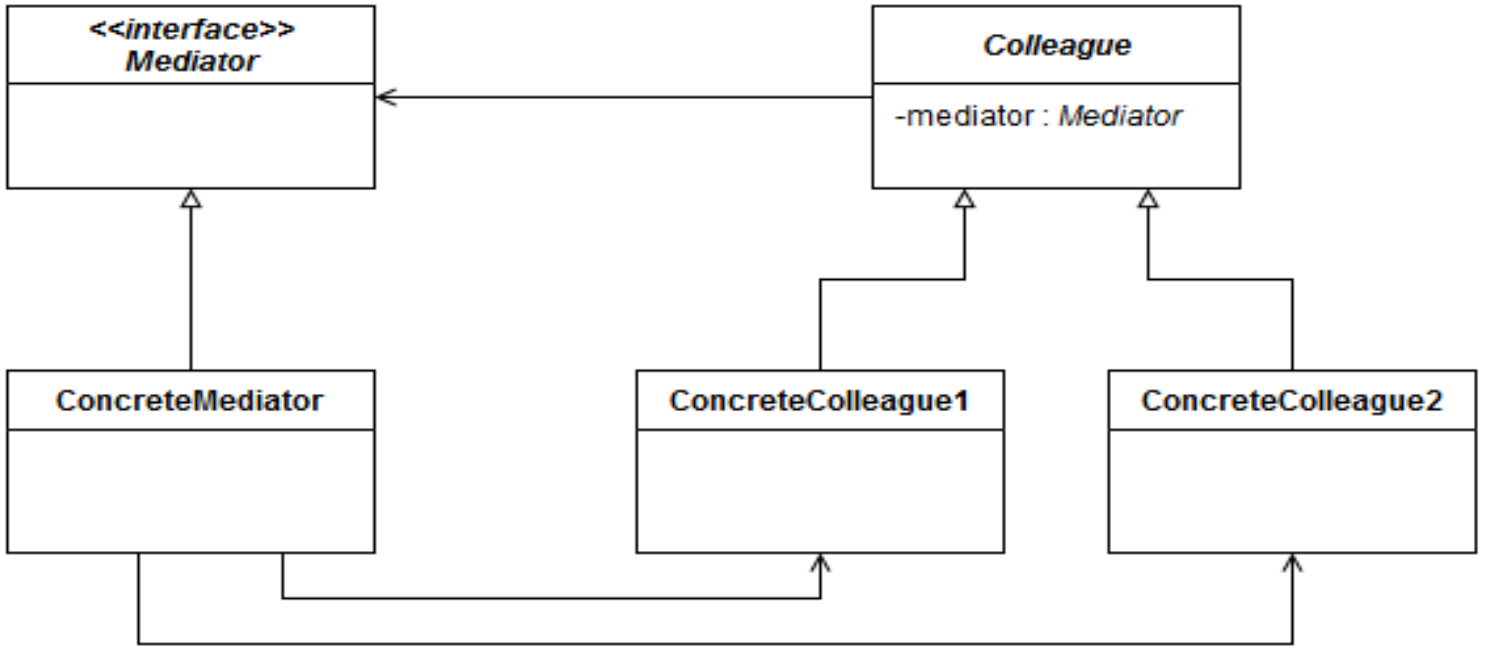
Le modèle Mediator est un bon choix si nous devons traiter un ensemble d'objets étroitement couplés et difficiles à maintenir. De cette façon, nous pouvons réduire les dépendances entre les objets et diminuer la complexité globale.

De plus, en utilisant l'objet médiateur, nous extrayons la logique de communication vers le composant unique, nous suivons donc le principe de responsabilité unique . De plus, nous pouvons introduire de nouveaux médiateurs sans qu'il soit nécessaire de modifier les parties restantes du système. Par conséquent, nous suivons le principe ouvert-fermé.

Objectifs

- Gerer la transmission d’informations entre des objets interagissant entre eux.
- Apres un couplage faible entre les objets puisqu’ils n’ont pas de lien direct entre eux.
- Pouvoir varier leur interaction independamment.

Diagramme de classe



Dans le diagramme UML ci-dessus, nous pouvons identifier les participants suivants :

- *Le **médiateur*** définit l'interface que les objets ***Colleague*** utilisent pour communiquer.
- *Un **collègue*** définit la classe abstraite contenant une seule référence au *médiateur.*
- ***ConcreteMediator*** encapsule la logique d'interaction entre les objets ***Colleague.***
- ***ConcreteColleague1*** et ***ConcreteColleague2*** communiquent uniquement via le ***médiateur.***

Comme nous pouvons le voir, les objets ***Colleague*** ne se réfèrent pas directement les uns aux autres. Au lieu de cela, toute la communication est effectuée par le ***Médiateur*** .

Par conséquent, ***ConcreteColleague1*** et ***ConcreteColleague2*** peuvent être plus facilement réutilisés.

Implémentation

construisons un système de refroidissement simple composé d'un ventilateur, d'une alimentation et d'un bouton. Appuyez sur le bouton pour allumer ou éteindre le ventilateur. Avant d'allumer le ventilateur, nous devons allumer le courant. De même, nous devons couper l'alimentation juste après l'arrêt du ventilateur.

Voyons maintenant l'exemple d'implémentation :

```java
public class Button {
    private Fan fan;

    // constructor, getters and setters

    public void press(){
        if(fan.isOn()){
            fan.turnOff();
        } else {
            fan.turnOn();
        }
    }
}
```

```java
public class Fan {
    private Button button;
    private PowerSupplier powerSupplier;
    private boolean isOn = false;

    // constructor, getters and setters

    public void turnOn() {
        powerSupplier.turnOn();
        isOn = true;
    }

    public void turnOff() {
        isOn = false;
        powerSupplier.turnOff();
    }

    public boolean isOn() {
        return false;
    }
}
```

```
public class PowerSupplier {
    public void turnOn() {
        // implementation
    }

    public void turnOff() {
        // implementation
    }
}
```

Implémentons maintenant le modèle Mediator pour réduire les dépendances entre nos classes et rendre le code plus réutilisable.

Tout d'abord, introduisons la classe *Mediator* :

```java
public class Mediator {
    private Button button;
    private Fan fan;
    private PowerSupplier powerSupplier;

    // constructor, getters and setters

    public void press() {
        if (fan.isOn()) {
            fan.turnOff();
        } else {
            fan.turnOn();
        }
    }

    public void start() {
        powerSupplier.turnOn();
    }

    public void stop() {
        powerSupplier.turnOff();
    }
}
```

Ensuite, modifions les classes restantes :

```
public class Button {
    private Mediator mediator;

    // constructor, getters and setters

    public void press() {
        mediator.press();
    }
}
```

```java
public class Fan {
    private Mediator mediator;
    private boolean isOn = false;

    // constructor, getters and setters

    public void turnOn() {
        mediator.start();
        isOn = true;
    }

    public void turnOff() {
        isOn = false;
        mediator.stop();
    }
}
```

Maintenant que nous avons implémenté le modèle Mediator, aucune des classes *Button* , *Fan* ou *PowerSupplier* ne communique directement . Ils n'ont qu'une seule référence au *Médiateur.*

Si nous devons ajouter une deuxième alimentation à l'avenir, tout ce que nous avons à faire est de mettre à jour *la* logique *de Mediator* ; *Les* classes *Button* et *Fan* restent inchangées.

Cet exemple montre avec quelle facilité nous pouvons séparer les objets dépendants et rendre notre système plus facile à entretenir.