(/)

El patrón mediador en Java

Última modificación: 2 de abril de 2019.

por Krzysztof Woyke (https://www.baeldung.com/author/krzysztof-woyke/) (https://www.baeldung.com/author/krzysztof-woyke/)

Java (https://www.baeldung.com/category/java/) +

Modelo (https://www.baeldung.com/tag/pattern/)

1. Información general

En este artículo, veremos **el patrón de mediador, uno de los patrones de comportamiento de GoF (https://en.wikipedia.org/wiki/Design_Patterns)** . Describiremos su propósito y explicaremos cuándo debemos usarlo.

Como de costumbre, también proporcionaremos un ejemplo de código simple.

2. Patrón de mediador

En la programación orientada a objetos, siempre debemos tratar de **diseñar el sistema de tal manera que los componentes estén acoplados y sean reutilizables** . Este enfoque hace que nuestro código sea más fácil de mantener y probar.

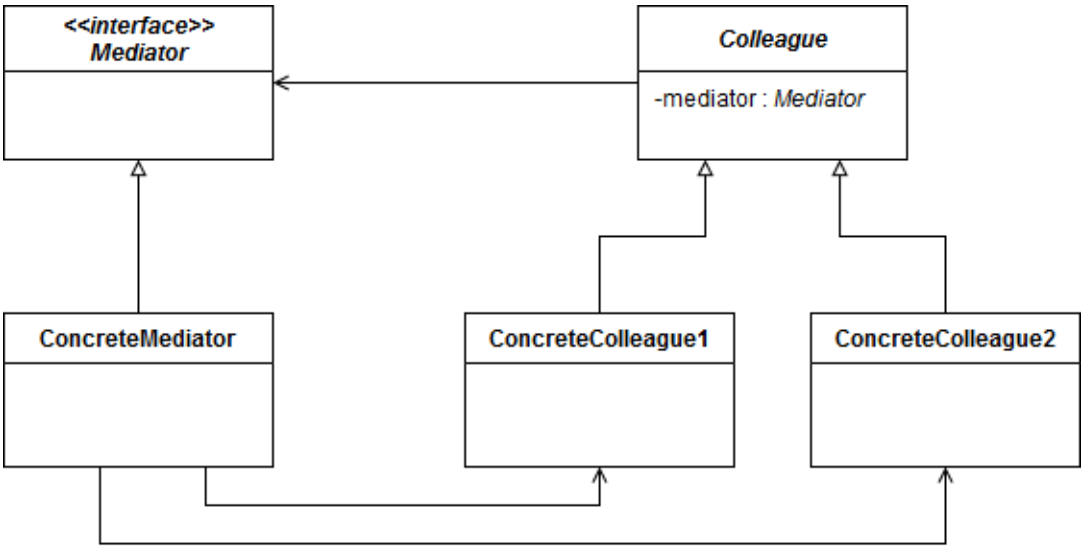
En la vida real, sin embargo, a menudo necesitamos tratar con un conjunto complejo de objetos dependientes. Esto es cuando el patrón de mediador puede ser útil.

La intención del patrón de mediador es reducir la complejidad y las dependencias entre objetos estrechamente acoplados que se comunican directamente entre sí . Esto se logra creando un objeto mediador que cuida la interacción entre los objetos dependientes. En consecuencia, toda la comunicación pasa por el mediador.

Esto promueve el acoplamiento suelto, ya que un conjunto de componentes que trabajan juntos ya no tienen que interactuar directamente. En su lugar, solo se refieren al objeto de mediador único. De esta manera, también es más fácil reutilizar estos objetos en otras partes del sistema.

3. Diagrama UML del patrón mediador

Veamos ahora el patrón visualmente:



(https://www.baeldung.com/wp-content/uploads/2019/03/mediator.png)
En el diagrama UML anterior, podemos identificar a los siguientes participantes:

- *El mediador* define la interfaz que los objetos del *Colega* utilizan para comunicarse
- *Colega* define la clase abstracta que contiene una sola referencia al *mediador.*
- *ConcreteMediator* encapsula la lógica de interacción entre los objetos de *colega*
- *ConcreteColleague1* y *ConcreteColleague2 se* comunican solo a través del *Mediador*

Como podemos ver, los **objetos de *colega* no se refieren directamente entre sí. En su lugar, toda la comunicación es llevada a cabo por el *mediador*** .

En consecuencia, *ConcreteColleague1* y *ConcreteColleague2* se pueden reutilizar más fácilmente.

Además, en caso de que necesitemos cambiar la forma en que los objetos de *Colega* trabajan juntos, solo tenemos que modificar la lógica de *ConcreteMediator* . O podemos crear una nueva implementación del *mediador.*

4. Implementación de Java

Ahora que tenemos una idea clara de la teoría, veamos un ejemplo para comprender mejor el concepto en la práctica.

4.1. Escenario de ejemplo

Imagina que estamos construyendo un sistema de refrigeración simple que consta de un ventilador, una fuente de alimentación y un botón. Al presionar el botón, se encenderá o apagará el ventilador. Antes de encender el ventilador, necesitamos encender la alimentación. Del mismo modo, tenemos que apagar la alimentación inmediatamente después de apagar el ventilador.

Veamos ahora la implementación del ejemplo:

```
public class Button {
    private Fan fan;

    // constructor, getters and setters

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

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

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

A continuación, probemos la funcionalidad:

```
@Test
public void givenTurnedOffFan_whenPressingButtonTwice_fanShouldTurnOr
    assertFalse(fan.isOn());

    button.press();
    assertTrue(fan.isOn());

    button.press();
    assertFalse(fan.isOn());
}
```

Todo parece funcionar bien. Pero note cómo las **clases *Button, Fan* y *PowerSupplier* están estrechamente acopladas** . El *botón* funciona directamente en el *ventilador* y el *ventilador* interactúa tanto con el *botón* como con el *PowerSupplier.*

Sería difícil reutilizar la clase *Button* en otros módulos. Además, si necesitamos agregar una segunda fuente de alimentación a nuestro sistema, tendríamos que modificar la lógica de la clase *Fan* .

4.2. Añadiendo el patrón de mediador

Ahora, implementemos el patrón de mediador para reducir las dependencias entre nuestras clases y hacer que el código sea más reutilizable.

Primero, introduzcamos la clase *Mediator* :

```
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();
    }
}
```

A continuación, modifiquemos las clases restantes:

```
public class Button {
    private Mediator mediator;

    // constructor, getters and setters

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

```
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();
    }
}
```

De nuevo, probemos la funcionalidad:

```
@Test
public void givenTurnedOffFan_whenPressingButtonTwice_fanShouldTurnOr
    assertFalse(fan.isOn());

    button.press();
    assertTrue(fan.isOn());

    button.press();
    assertFalse(fan.isOn());
}
```

Nuestro sistema de refrigeración funciona como se espera.

Ahora que hemos implementado el Patrón del mediador, ninguna de las clases de *Button* , *Fan* o *PowerSupplier* se comunica directamente . Solo tienen una única referencia al *mediador.*

Si necesitamos agregar una segunda fuente de alimentación en el futuro, todo lo que tenemos que hacer es actualizar *la* lógica *del Mediador* ; *Las* clases de *botones* y *abanicos* permanecen intactas.

Este ejemplo muestra con qué facilidad podemos separar los objetos dependientes y hacer que nuestro sistema sea más fácil de mantener.

5. Cuándo usar el patrón de mediador

El patrón de mediador es una buena opción si tenemos que tratar con un conjunto de objetos que están estrechamente acoplados y son difíciles de mantener. De esta manera podemos reducir las dependencias entre objetos y disminuir la complejidad general.

Además, al utilizar el objeto mediador, extraemos la lógica de comunicación al componente único, por lo tanto, seguimos el Principio de Responsabilidad Única (https://www.baeldung.com/solid-principles#s) . Además, podemos introducir nuevos mediadores sin necesidad de cambiar las partes restantes del sistema. Por lo tanto, seguimos el principio abierto-cerrado.

A veces, sin embargo, es posible que tengamos demasiados objetos estrechamente acoplados debido al diseño defectuoso del sistema. Si este es un caso, no debemos aplicar el patrón de mediador . En su lugar, deberíamos dar un paso atrás y repensar la forma en que hemos modelado nuestras clases.

Al igual que con todos los demás patrones, **debemos considerar nuestro caso de uso específico antes de implementar ciegamente el Patrón del mediador** .

6. Conclusión

En este artículo, aprendimos sobre el patrón de mediador. Explicamos qué problema resuelve este patrón y cuándo deberíamos considerar su uso. También implementamos un ejemplo simple del patrón de diseño.

Como siempre, los ejemplos de código completos están disponibles en GitHub (https://github.com/eugenp/tutorials/tree/master/patterns/design-patterns-2) .

Deja una respuesta

Start the discussion...

Suscribir

CATEGORÍAS

PRIMAVERA (HTTPS://WWW.BAELDUNG.COM/CATEGORY/SPRING/)
DESCANSO (HTTPS://WWW.BAELDUNG.COM/CATEGORY/REST/)
JAVA (HTTPS://WWW.BAELDUNG.COM/CATEGORY/JAVA/)
SEGURIDAD (HTTPS://WWW.BAELDUNG.COM/CATEGORY/SECURITY-2/)
PERSISTENCIA (HTTPS://WWW.BAELDUNG.COM/CATEGORY/PERSISTENCE/)
JACKSON (HTTPS://WWW.BAELDUNG.COM/CATEGORY/JSON/JACKSON/)
CLIENTE HTTP (HTTPS://WWW.BAELDUNG.COM/CATEGORY/HTTP/)
KOTLIN (HTTPS://WWW.BAELDUNG.COM/CATEGORY/KOTLIN/)

SERIE

TUTORIAL "VOLVER A LO BÁSICO" DE JAVA (/JAVA-TUTORIAL)
JACKSON JSON TUTORIAL (/JACKSON)
TUTORIAL HTTPCLIENT 4 (/HTTPCLIENT-GUIDE)
DESCANSO CON TUTORIAL DE PRIMAVERA (/REST-WITH-SPRING-SERIES)
TUTORIAL DE PERSISTENCIA DE PRIMAVERA (/PERSISTENCE-WITH-SPRING-SERIES)
SEGURIDAD CON SPRING (/SECURITY-SPRING)

ACERCA DE

ACERCA DE BAELDUNG (/ABOUT)
LOS CURSOS (HTTPS://COURSES.BAELDUNG.COM)
TRABAJO DE CONSULTORÍA (/CONSULTING)
META BAELDUNG (HTTP://META.BAELDUNG.COM/)
EL ARCHIVO COMPLETO (/FULL_ARCHIVE)
ESCRIBIR PARA BAELDUNG (/CONTRIBUTION-GUIDELINES)
EDITORES (/EDITORS)
NUESTROS COMPAÑEROS (/PARTNERS)
PUBLICIDAD EN BAELDUNG (/ADVERTISE)

TÉRMINOS DE SERVICIO (/TERMS-OF-SERVICE)
POLÍTICA DE PRIVACIDAD (/PRIVACY-POLICY)
INFORMACIÓN DE LA COMPAÑÍA (/BAELDUNG-COMPANY-INFO)

CONTACTO (/CONTACT)