2020

PATRÓN DE DISEÑO MEDIATOR

MATERIA: DISEÑO DE SISTEMAS
ENRICO, MAURO. LEGAJO: 14375

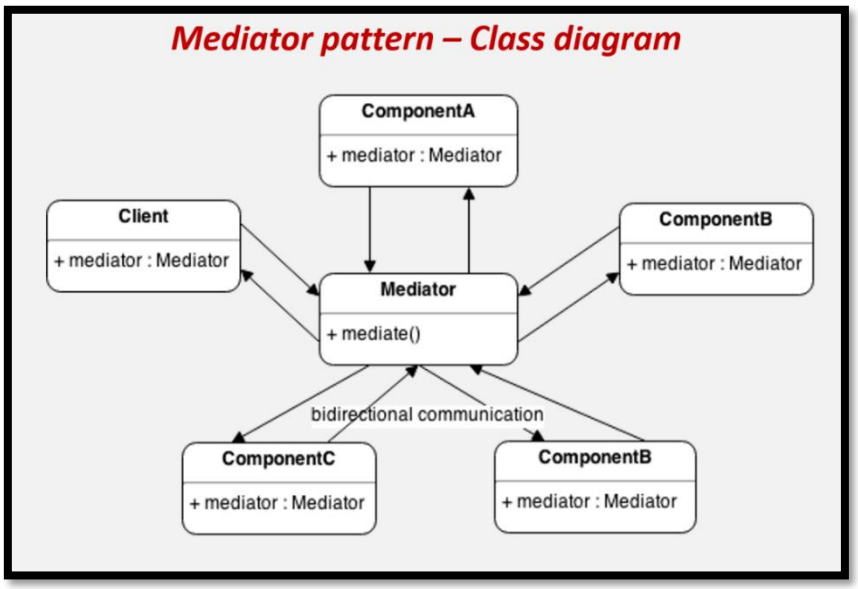
UNIVERSIDAD TECNOLÓGICA NACIONAL - SAN FCO

Patrones de diseño: “Mediator”

El patrón de diseño Mediator se encarga de gestionar la forma en que un conjunto de clases se comunica entre sí. Este patrón es especialmente útil cuando tenemos una gran cantidad de clases que se comunican de forma directa, ya que mediante la implementación de este esquema podemos crear una capa de comunicación bidireccional, en la cual las clases se pueden comunicar con el resto de ellas por medio de un objeto en común que funge como un mediador o intermediario.

En proyectos grandes podemos llegar a tener un problema, y es que el número de clases aumenta y con esto también aumentan las relaciones que tienen las clases con el resto de clases del proyecto. Esto puede suponer un grave problema de acoplamiento con el resto de clases de nuestro proyecto, sobre todo por que creamos canales de comunicación directos y difíciles de rastrear o depurar.

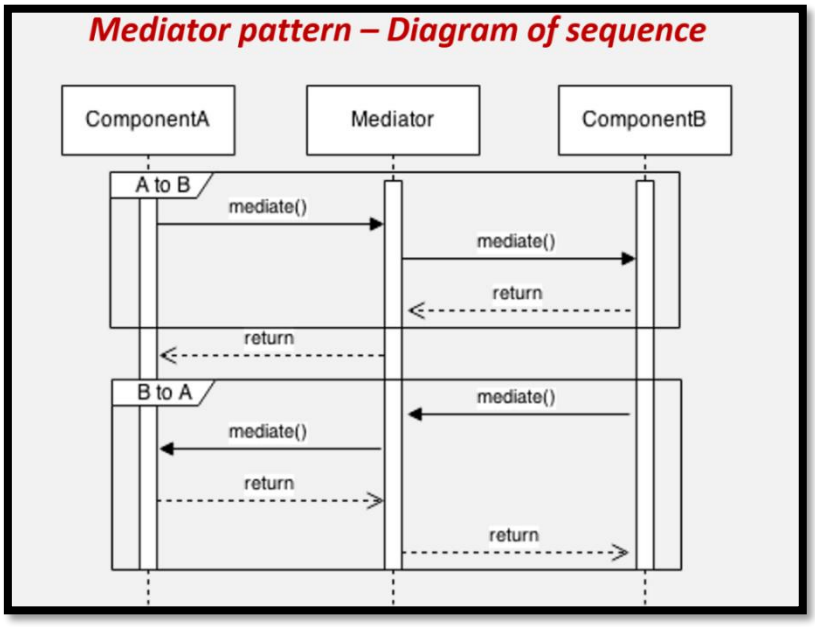
Para comprender mejor como funciona el concepto del patrón mediator, procederemos con un ejemplo:



Los componentes que conforman el patrón:

- **Client:** componente que inicia la comunicación con el resto de los componentes por medio del mediador.
- **Components:** componentes que son parte de la red de comunicación por medio del mediador, éstos pueden ser diversos objetos que comparten el mismo mediador para comunicarse.
- **Mediador:** componente que sirve de mediador entre el resto de componentes, tiene como principal rol canalizar los mensajes entrantes al destinatario correspondiente

Diagrama de secuencia del patrón Mediator:



1. El ComponenteA desea comunicarse con el ComponenteB y le envía un mensaje por medio del mediador.

2. El mediador puede analizar el mensaje con fines de depuración, seguimiento o para canalizar el mensaje al destinatario.

3. El mensaje es entregado al destinatario y regresa una respuesta al mediador.

4. El mediador recibe la respuesta y la redirecciona al ComponenteA.

5. De igual forma, el proceso se puede repetir del ComponentB al ComponentA repitiendo los pasos anteriores logrando una comunicación bidireccional.

La idea de este sencillo ejemplo es comprender la esencia del patrón, la función que cumple y de qué manera lo hace.

Un enfoque más orientado a la programación

En el mundo de la programación orientada a objetos una de las máximas que debemos cumplir, si queremos desarrollar un código de calidad, es buscar una elevada cohesión con bajo acoplamiento. Con el fin de ayudarnos en esta tarea, aparece el patrón Mediator.

La **cohesión** es una de las características más importantes de la OOP (Object Oriented Programming). Se refiere a que hay que dotar a las clases de un solo ámbito de desarrollo, y a su vez, todo lo referido a ese ámbito, quede encapsulado dentro de una sola clase. Este tipo de programación ofrece la ventaja de reutilizar el código fuente, pero a medida que creamos objetos que se interrelacionan entre sí es menos probable que un objeto pueda funcionar sin la ayuda de otros.

Para evitar esto podemos utilizar el patrón Mediator, en el que se define una clase que hará de mediadora encapsulando la comunicación entre los objetos, evitándose con ello la necesidad de que lo hagan directamente entre sí.

Debemos buscar un bajo acoplamiento y una alta cohesión

Para lograr comprender este apartado, primero debemos tener en claro qué es un acoplamiento.

Decimos que existe **acoplamiento** cuando clases relacionadas necesitan conocer detalles sobre comportamiento interno, unas de otras, para poder desempeñar correctamente su función. De esta forma, los cambios se irán propagando de unas clases a otras y como resultado tendremos un código difícil de seguir, leer y por lo tanto mantener.

Existen varias formas de evitar el acoplamiento entre clases. Pero cuando en nuestro escenario nos encontramos una gran cantidad de clases que necesitan interactuar entre sí, para funcionar correctamente; podríamos crear un mecanismo para facilitar la comunicación y así evitar que unas clases tengan que conocer la existencia de otras.

Un patrón de comportamiento y muy útil en UI

El Mediator es considerado como un patrón de comportamiento debido al hecho de que puede alterar el comportamiento del programa en ejecución, y además define una forma de comunicación entre clases.

Remarcamos esta característica porque es sencillo caer en el error de pensar que este patrón está destinado en exclusiva a acompañar a la interfaz gráfica de usuario.

Un ejemplo muy común que podemos encontrar por la red es la gestión de una aplicación de chat. Aquí se implementa el patrón Mediator al convertir a los “Speakers” en “Colleagues” y la “ChatRoom” en el “Mediator” del sistema, que se dedicará a realizar broadcast de los mensajes a todos los usuarios conectados.

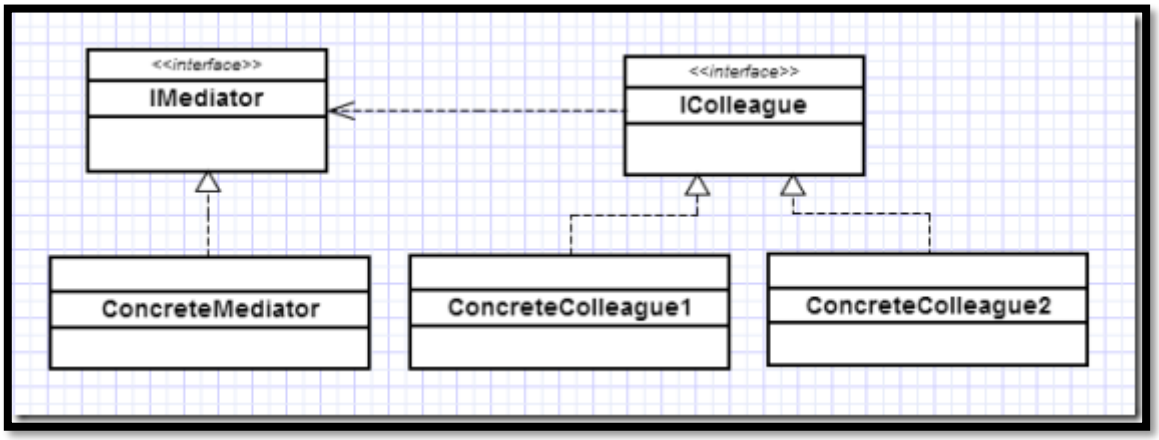
Cuando un “speaker” quiere comunicarse el resto de los integrantes de la “ChatRoom”, envía un mensaje al mediador, que tiene conocimiento acerca de aquellos que se encuentran activos o conectados.

```
public class Speaker : ISpeaker
{
    public IChatRoom ChatRoom { get; set; }

    public void OnReceive(string message)
    {
        // write message on UI
    }

    public void OnSend(string message)
    {
        this.ChatRoom.Send(message);
    }
}
```

Estamos ante un patrón que resulta muy útil a la hora de conectar artefactos en UI, y más usando patrones como MVVM (model–view–viewmodel), MVC (model, view, control) o cualquier derivación del Model Presenter.



Ahora bien, veamos cómo se traduce este patrón a código.

La clase "***mediator***" y "***colleague****"* serán la interfaz:

```
public interface Mediator
{
	public void send (String  message, Colleague colleage);
}


Public abstract class Colleague
{
	protected Mediator mediador;

	public Colleague(Mediator mediador) {
		this.mediador = mediador;
	}

	Public abstract void send(String message);
	Public abstract void messageReceived(String message);
}
```

En el programa principal, cuando comienza la ejecución, instanciamos una clase objeto mediador como así también los dos usuarios del chat (colega1 y colega2).
Llamamos a los métodos "*seters"* del mediador para registrar usuarios.

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

	ContreteMediator mediador = new ConcreteMediator();

	ConcreteColleague1 colleague1 = new ConcreteColleague1 (mediator);
	ConcreteColleague1 colleague2 = new ConcreteColleague2 (mediator);

	mediator.setColleage1(coleague1);
	mediator.setColleage2(coleague2);

	coleage1.send("Hola, como estas?");
	coleage1.send("Bien, y vos?");
  }
}
```

En este caso son 2 colegas y el problema está reducido a su mínima expresión.
"ConcreteColleague1" *y* ***"ConcreteColleague2"*** implementaran los 2 métodos, tanto el de enviar como el de recibir un mensaje. Lo único que hace, cuando un usuario envía un mensaje, es invocar a su vez al método *"send"* del mediador.

```
public class ConcreteColleague1 extends Colleague {

  public ConcreteColleague1 (Mediator mediador) {

	super(mediador);
   }

  public void send(String message) {

	mediador.send(message, this);
   }

  public void messageReceived(String message) {
```

```
        System.out.println("colleague1 recibe el mensaje : " + message);
    }

}

public class ConcreteColleague2 extends Colleague {

  public ConcreteColleague2 (Mediator mediador) {

        super(mediador);
    }

  public void send(String message) {

        mediador.send(message, this);
    }

  public void messageReceived(String message) {

        System.out.println("colleague2 recibe el mensaje : " + message);
    }

}
```

Contamos con un **"*concreteMediator"*** que será la clase instanciable, al igual que en el caso de ***"ConcreteColleague1"*** y ***"ConcreteColleague2"***.
"concreteMediator" únicamente aplica el método *"send"*, que recibiría 3 parámetros, mensaje, origen y destino (en este caso, al haber 2 usuarios, el destino es uno u el otro).

```
public class concreteMediator implements Mediador {

  private ConcreteColleague1 usuario1;
  private ConcreteColleague2 usuario2;

  public void setColleague1(ConcretColleague1 colleage1) {
        usuario1 = colleage1;
  }

  public void setColleague1(ConcretColleague2 colleage2) {
        usuario2 = colleage2;
  }

  public void send(String message, Colleague colleage) {

        if(colleage == usuario1)
        {
               Usuario2.messageReceived(message);
        }
        else if(collage == usuario2)
        {
               Usuario1.messageReceived(message);
        }
  }

}
```

Comprobamos que usuario1 envía el mensaje, y el propio mediador es el que invoca el método de recibir mensaje del colega (usuario2). De igual forma ocurre en el caso del primer usuario.

Si observamos, en ***"ConcreteColleague1"***, lo único que hace es visualizar el mensaje.

A modo de conclusión

El uso del patrón Mediator va hacer nuestro código mucho más legible, ya que favorece la cohesión. Su uso ayudará a crear clases desacopladas, y por lo tanto nuestro código será más fácil de probar usando tests unitarios. Además, va a simplificar los protocolos de comunicación, al estar centralizados en un solo artefacto.

Debemos tener en cuenta que Mediator es un patrón difícil de implementar. Lo más recomendable es usarlo como pasarela de comunicación sin lógica interna, ya que podemos caer en el antipatrón de acoplar todas nuestras clases con un mediador de lógica bastante compleja.