Mediator Pattern

Introducción y nombre

Mediator. De Comportamiento. Define un objeto que hace de procesador central.

Intención

Un Mediator es un patrón de diseño que coordina las relaciones entre sus asociados o participantes. Permite la interacción de varios objetos, sin generar acoples fuertes en esas relaciones. Todos los objetos se comunican con un mediador y es éste quién realiza la comunicación con el resto.

También conocido como

Mediador, Intermediario.

Motivación

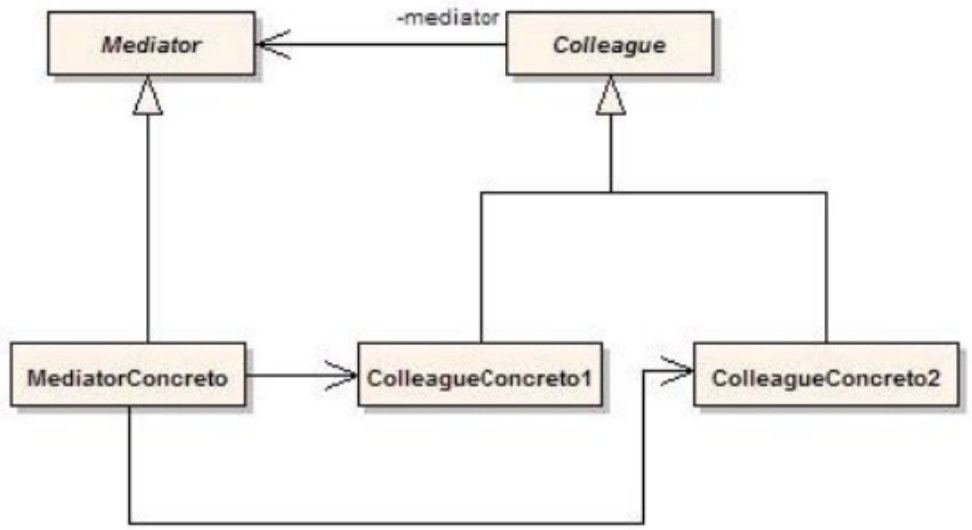
Cuando muchos objetos interactúan con otros objetos, se puede formar una estructura muy compleja, con muchas conexiones entre distintos objetos. En un caso extremo cada objeto puede conocer a todos los demás objetos. Para evitar esto, el patrón Mediator, encapsula el comportamiento de todo un conjunto de objetos en un solo objeto.

Solución

Usar el patrón Mediator cuando:
Un conjunto grande de objetos se comunica de una forma bien definida, pero compleja.
Reutilizar un objeto se hace difícil porque se relaciona con muchos objetos.
Las clases son difíciles de reutilizar porque su función básica está entrelazada con relaciones de dependencia.

Diagrama UML

Estructura en un diagramas de clases.



Participantes

Responsabilidad de cada clase participante.

Mediator: define una interface para comunicarse con los objetos colegas.

MediatorConcreto: implementa la interface y define cómo los colegas se comunican entre ellos. Además los conoce y mantiene, con lo cual hace de procesador central de todos ellos.

Colleague: define el comportamiento que debe implementar cada colega para poder comunicarse el mediador de una manera estandarizada para todos.

ColleagueConcreto: cada colega conoce su mediador, y lo usa para comunicarse con otros colegas.

Colaboraciones

Los colegas envían y reciben requerimientos de un objeto mediador. El mediador gestiona cada mensaje y se lo comunica a otro colega si fuese necesario.

Consecuencias

Desacopla a los colegas: el patrón Mediator promueve bajar el acoplamiento entre colegas. Se puede variar y reusar colegas y mediadores independientemente.

Simplifica la comunicación entre objetos: los objetos que se comunican de la forma "muchos a muchos" puede ser reemplazada por una forma "uno a muchos" que es menos compleja y más elegante. Además esta forma de comunicación es más fácil de entender. Es decir, un objeto no necesita conocer a todos los objetos, tan sólo a un mediador.

Clarifica cómo los objetos se relacionan en un sistema.

Centraliza el control: el mediador es el que se encarga de comunicar a los colegas, este puede ser muy complejo, difícil de entender y modificar. Para que quién conoce el framework Struts, es muy similar al concepto del archivo struts-config.xml: centraliza el funcionamiento de la aplicación, aunque si llega a ser una aplicación muy compleja el archivo se vuelve un tanto complicado de entender y seguir.

Implementación

Sabemos que el patrón Mediator introduce un objeto para mediar la comunicación entre "colegas". Algunas veces el objeto Mediador implementa operaciones simplemente para enviarlas o otros objetos; otras veces pasa una referencia a él mismo y por consiguiente utiliza la verdadera delegación.
Entre los colegas puede existir dos tipos de dependencias:

1. Un tipo de dependencia requiere un objeto para conseguir la aprobación de otros objetos antes de hacer tipos específicos de cambios de estado.
2. El otro tipo de dependencia requiere un objeto para notificar a otros objetos después de que este ha hecho un tipo específico de cambios de estado.

Ambos tipos de dependencias son manejadas de un modo similar. Las instancias de Colega1, Colega2, ... están asociadas con un objeto mediator. Cuando ellos quieren conseguir la aprobación anterior para un cambio de estado, llaman a un método del objeto Mediator. El método del objeto Mediator realiza cuidadoso el resto. Pero hay que tener en cuenta lo siguiente con respecto al mediador: Poner toda la dependencia de la lógica para un conjunto de objetos relacionados en un lugar puede hacer incomprensible la dependencia lógica fácilmente. Si la clase Mediator llega a ser demasiado grande, entonces dividirlo en piezas más pequeñas puede hacerlo más comprensible.

Código de muestra

Nuestro ejemplo será un chat: donde habrá usuarios que se comunicaran entre sí en un salón de chat. Para ellos se define una interface llamada Chateable que todos los objetos que quieran participar de un chat deberán implementar.

[code]
```
public interface Chateable {
        public void recibe(String de, String msg);
        public void envia(String a, String msg);
}
```

La clase Usuario representa un usuario que quiera chatear.

```
public class Usuario implements Chateable {
	private String nombre;
	private SalonDeChat salon;

	public Usuario(SalonDeChat salonDeChat) {
		salon = salonDeChat;
	}

	public void recibe(String de, String msg) {
		String s = "el usuario " + de + " te dice: " + msg;
		System.out.println(nombre + ": " + s);
	}

	public void envia(String a, String msg) {
		salon.envia(nombre, a, msg);
	}

	public String getNombre() {
		return nombre;
	}

	public void setNombre(String nombre) {
		this.nombre = nombre;
	}

	public SalonDeChat getSalon() {
		return salon;
	}

	public void setSalon(SalonDeChat salon) {
		this.salon = salon;
	}
}

public interface IChat {
	public abstract void registra(Usuario participante);
	public abstract void envia(String from, String to, String message);
}

public class SalonDeChat implements IChat {
	private Map<String, Usuario> participantes = new HashMap<String, Usuario>();

	@Override
	public void registra(Usuario user) {
		participantes.put(user.getNombre(), user);
	}
```

```
	@Override
	public void envia(String de, String a, String msg) {
		if (participantes.containsKey(de) && participantes.containsKey(a)) {
			Usuario u = participantes.get(a);
			u.recibe(de, msg);
		} else {
			System.out.println("Usuario inexistente");
		}
	}
}

public class Main {
	public static void main(String[] args) {
		SalonDeChat salonDeChat = new SalonDeChat();
		Usuario usuario1 = new Usuario(salonDeChat);
		usuario1.setNombre("Juan");

		Usuario usuario2 = new Usuario(salonDeChat);
		usuario2.setNombre("Pepe");

		Usuario usuario3 = new Usuario(salonDeChat);
		usuario3.setNombre("Pedro");

		salonDeChat.registra(usuario1);
		salonDeChat.registra(usuario2);
		salonDeChat.registra(usuario3);

		usuario1.envia("Pepe", "Hola como andas?");
		usuario2.envia("Juan", "Todo ok, vos?");
		usuario3.envia("Martin", "Martin estas?");
	}
}
```

El resultado por consola es:
Pepe: el usuario Juan te dice: Hola como andas?
Juan: el usuario Pepe te dice: Todo ok, vos?
Usuario inexistente
[/code]

Cuándo utilizarlo

Un ejemplo real que se puede comparar con este patrón es un framework que se llama Struts. Struts posee un clase que hace de Mediadora que se llama ActionServlet. Esta clase lee un archivo de configuración (el struts-config.xml) para ayudarse con la mediación, pero lo importante es que se encarga de comunicar el flujo de información de todos los

componentes web de una aplicación. Si no utilizamos Struts, entonces cada página debe saber hacia dónde debe dirigir el flujo y el mantenimiento de dicha aplicación puede resultar complicado, especialmente si la aplicación es muy grande.
En cambio, si todas las páginas se comunican con un mediador, entonces la aplicación es mucho más robusta: con cambiar un atributo del mediador todo sigue funcionando igual.
Como conclusión podemos afirmar que este patrón debe ser utilizado en casos donde convenga utilizar un procesador central, en vez de que cada objeto tenga que conocer la implementación de otro. Imaginemos un aeropuerto: que pasaría si no tuviese una torre de control y todos los aviones que deban aterrizar/despegar se tienen que poner todos de acuerdo para hacerlo. Además cada avión debe conocer detalles de otros aviones (velocidad de despegue, nafta que le queda a cada uno que quiera aterrizar, etc).
Para evitar esto se utiliza un torre de control que sincroniza el funcionamiento de un aeropuerto. Esta torre de control se puede ver como un mediador entre aviones.

Patrones relacionados

Patrones con los que potencialmente puede interactuar:
Observer: los colegas pueden comunicarse entre ellos mediante un patrón observador.
Facade: es un concepto similar al mediador, pero este último es un poco más completo y la comunicación es bidireccional.
Singleton: el mediador puede ser Singleton.