MEDIATOR

Patrón de diseño de sistemas de información

Iván Dietta

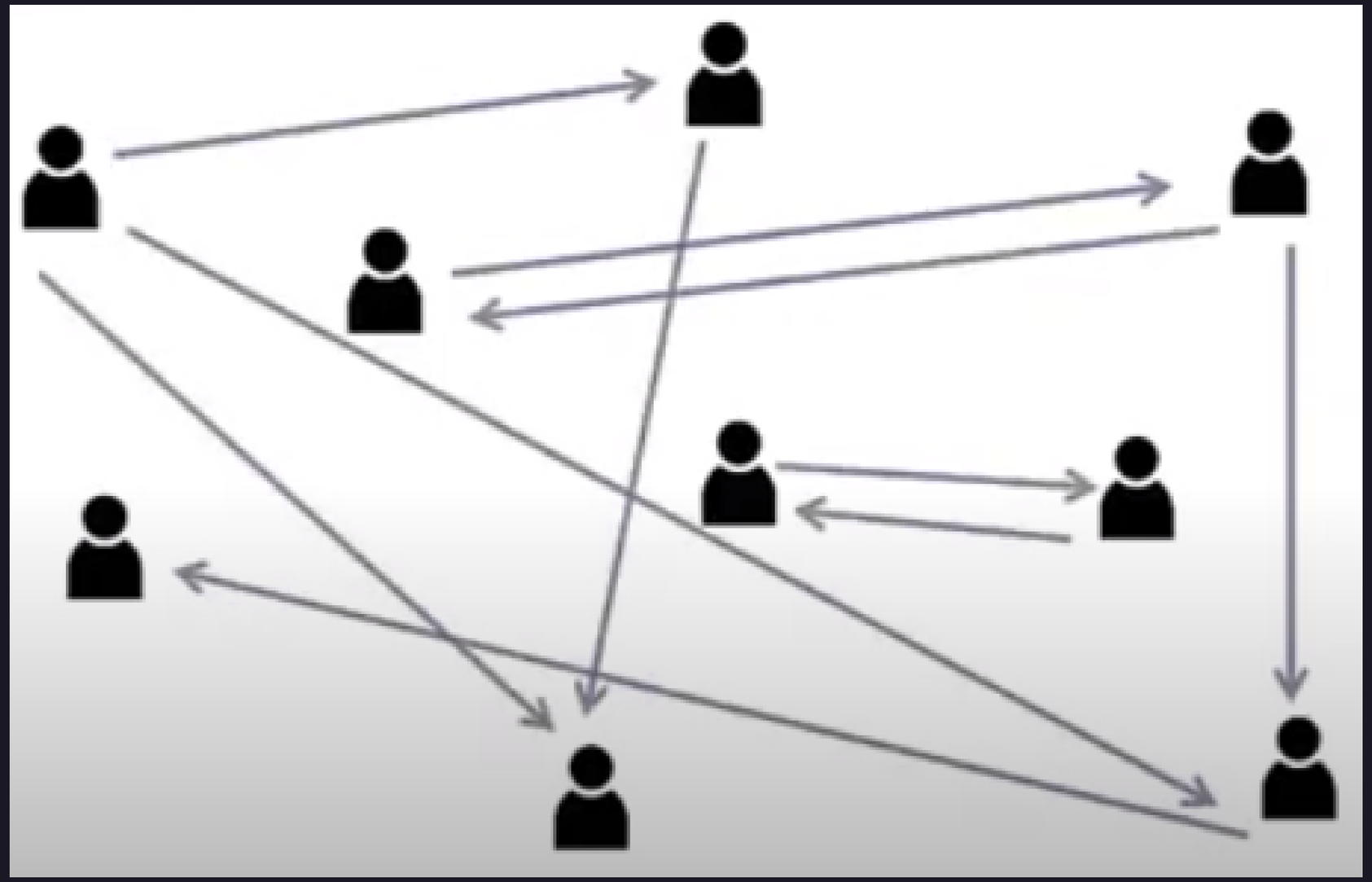
MEDIATOR

Mediator es un patrón de diseño de comportamiento que te permite reducir las dependencias caóticas entre objetos. El patrón restringe las comunicaciones directas entre los objetos, forzándolos a colaborar únicamente a través de un objeto mediador.

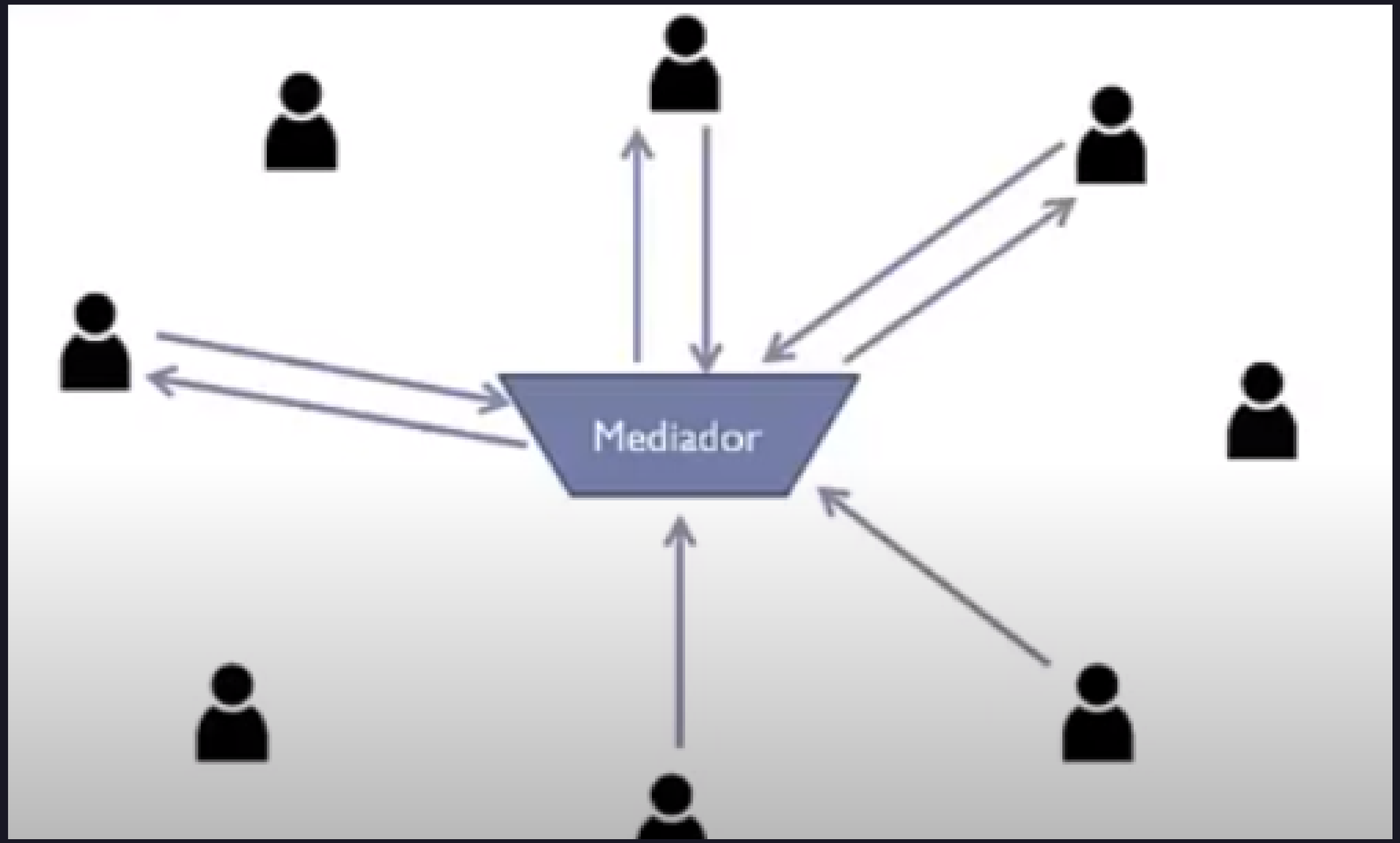
MEDIATOR

El patrón Mediator se utiliza comúnmente en sistemas de interfaz de usuario, sistemas de chat, sistemas de control de tráfico aéreo y en cualquier lugar donde sea necesario gestionar interacciones complejas entre objetos. Ayuda a mejorar la mantenibilidad y la organización del código al centralizar la lógica de comunicación en un solo lugar.

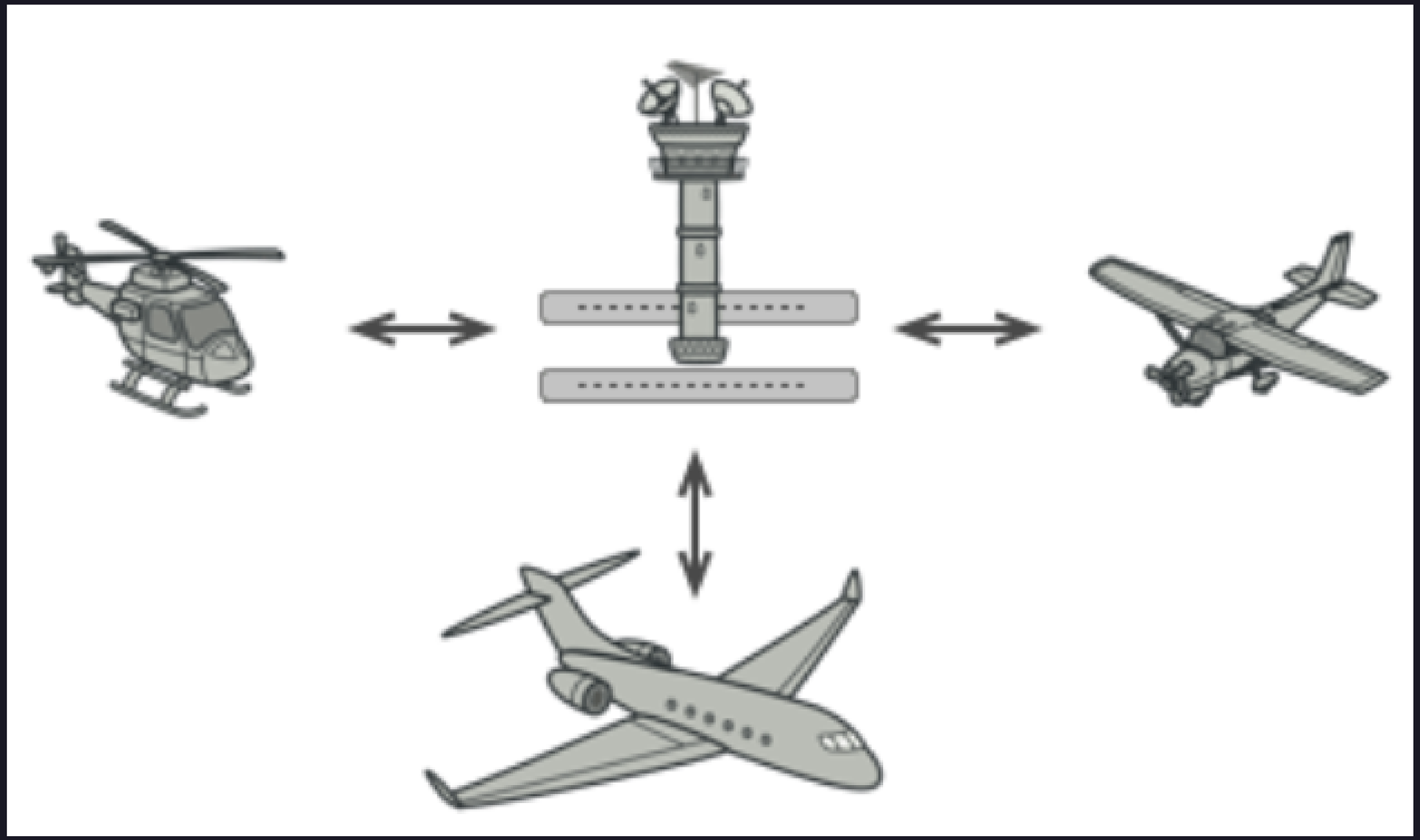
Problema



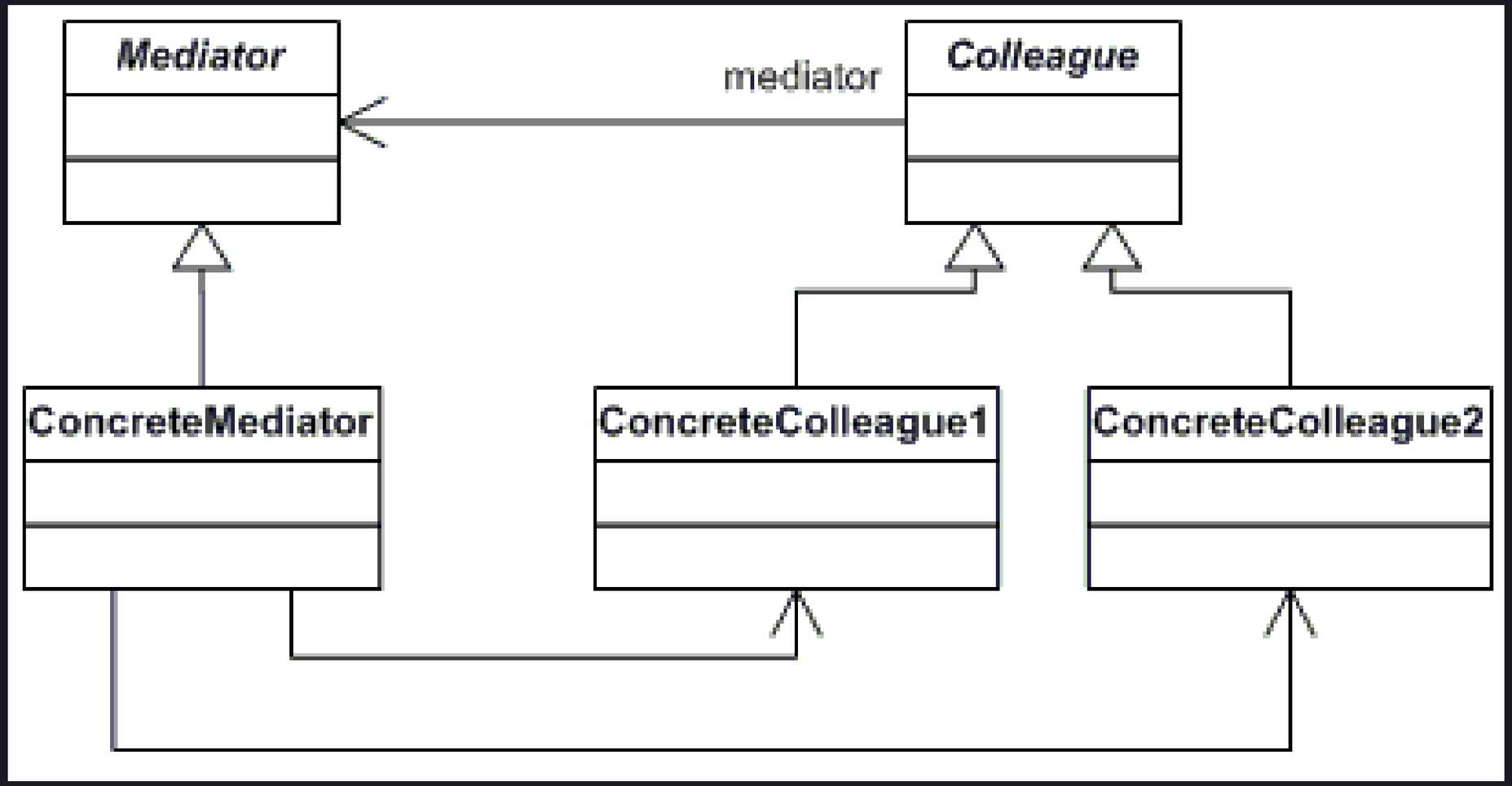
Solución



Ejemplo



Estructura


Mediator
mediator
Colleague
ConcreteMediator
ConcreteColleague1
ConcreteColleague2

Código

```python
# Interfaz Mediator (Mediador)
class ChatMediator(ABC):
    @abstractmethod
    def enviar_mensaje(self, mensaje, destinatario):
        pass
```

Interface que define métodos para los mediadores

```python
# Interfaz Colleague (Colega)
class Usuario(ABC):
    def __init__(self, mediator, nombre):
        self.mediator = mediator
        self.nombre = nombre

    @abstractmethod
    def enviar(self, mensaje, destinatario):
        pass

    @abstractmethod
    def recibir(self, mensaje):
        pass
```

Interface que define métodos para los colegas

```python
# ConcreteMediator (Mediador Concreto)
class SalaChat(ChatMediator):
    def __init__(self):
        self.usuarios = {}

    def agregar_usuario(self, usuario):
        self.usuarios[usuario.nombre] = usuario

    def enviar_mensaje(self, mensaje, destinatario):
        destinatario.recibir(mensaje)
```

```python
# ConcreteColleague (Colega Concreto)
class ChatUsuario(Usuario):
    def enviar(self, mensaje, destinatario):
        print(f"{self.nombre} envía el mensaje: {mensaje}")
        self.mediator.enviar_mensaje(mensaje, destinatario)

    def recibir(self, mensaje):
        print(f"{self.nombre} recibe el mensaje: {mensaje}")
```

```python
# Uso del patrón Mediator
mediador_chat = SalaChat()

usuario1 = ChatUsuario(mediador_chat, "Usuario 1")
usuario2 = ChatUsuario(mediador_chat, "Usuario 2")
usuario3 = ChatUsuario(mediador_chat, "Usuario 3")

mediador_chat.agregar_usuario(usuario1)
mediador_chat.agregar_usuario(usuario2)
mediador_chat.agregar_usuario(usuario3)

usuario1.enviar("Hola a todos!", usuario2)  # Usuario 1 envía un mensaje
```

Ventajas:

- Desacoplamiento
- Centralización de la lógica de comunicación
- Facilita la extensibilidad
- Mejora la legibilidad del código
- Facilita el testing

Desventajas:

- Complejidad adicional
- Pérdida de eficiencia
- Identificación de responsabilidades del mediador
- Costo inicial de implementación
- Puede evolucionar a un objeto todopoderoso

Gracias por su atención!