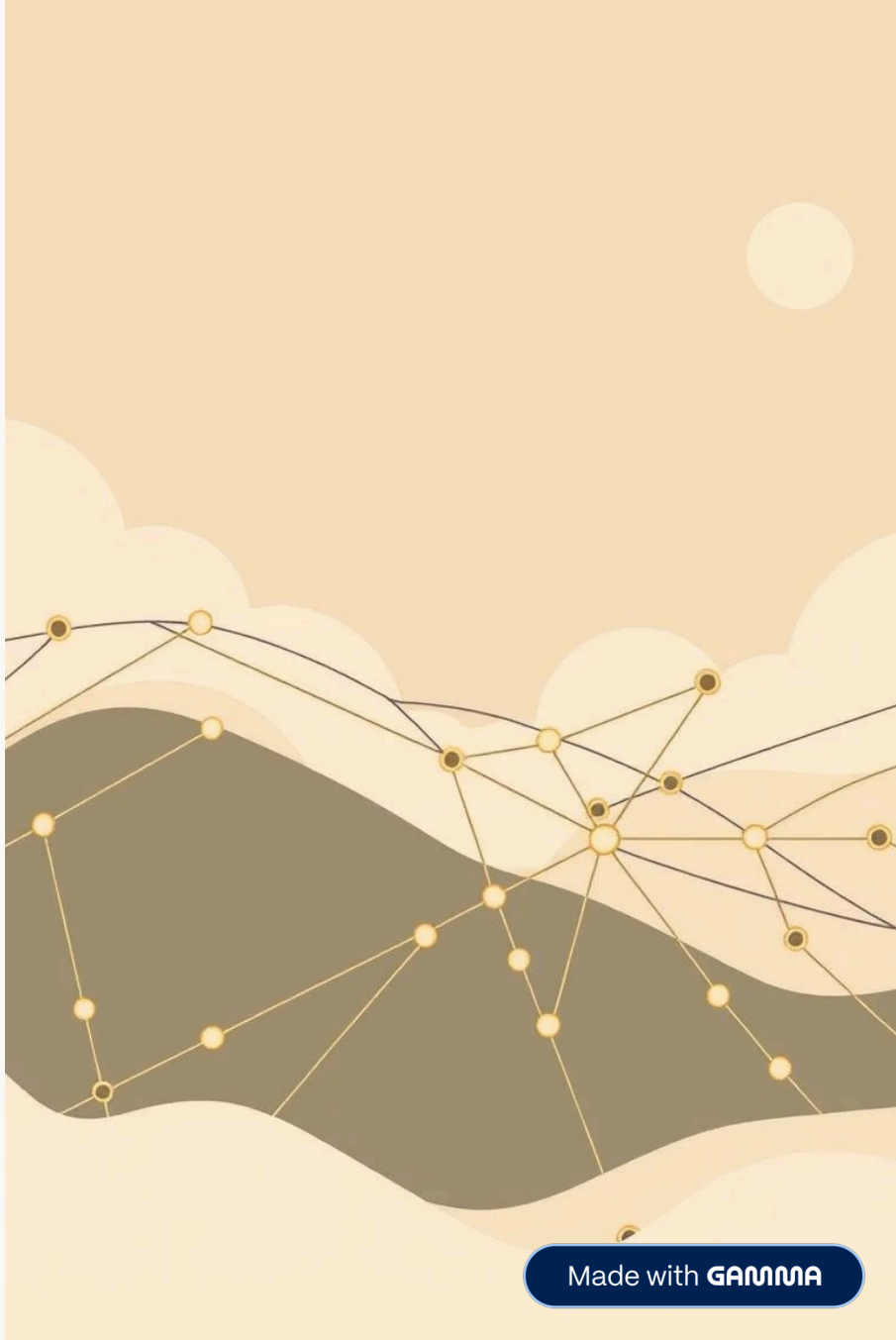
Patrón de Diseño Mediator: Comunicación Centralizada y Desacoplamiento

En el desarrollo de software, la forma en que los objetos se comunican entre sí es fundamental para la calidad del código. El patrón de diseño Mediator surge como una solución elegante para gestionar interacciones complejas, promoviendo un diseño más limpio y robusto.

- Mediator es un patrón de comportamiento que reduce las dependencias directas entre objetos.
- Actúa como un intermediario que gestiona la comunicación entre componentes, evitando que se llamen directamente.
- Beneficios clave: menor acoplamiento, mayor mantenibilidad y facilidad para extender sistemas complejos.



¿Por qué usar Mediator? Problema y Solución

El Problema: Dependencias Entrelazadas

En sistemas con numerosos objetos que necesitan interactuar, la creación de dependencias directas entre ellos puede llevar a un "espagueti code". Cada cambio en un componente puede requerir modificaciones en muchos otros, haciendo el código difícil de mantener, depurar y escalar.

Imagina una interfaz de usuario donde un botón afecta un campo de texto y ese campo de texto, a su vez, actualiza una lista. Sin un mediador, cada elemento necesita conocer y comunicarse con los demás directamente.

La Solución: Comunicación Centralizada

Centralización

Mediator centraliza la lógica de comunicación en un único objeto mediador.

Desacoplamiento

Los componentes ("colegas") se comunican solo con el mediador, no entre sí.

Coordinación

El mediador gestiona y coordina todas las interacciones entre los colegas.

Esto simplifica el diseño de los componentes individuales, ya que solo necesitan una referencia al mediador. El mediador sabe qué colegas necesitan ser notificados o actuar en respuesta a un evento.

Ejemplo: Mediator Simplificado

A continuación, se presenta un ejemplo sencillo del patrón Mediator en Python, aplicando la lógica de comunicación centralizada a un sistema básico de notificaciones entre usuarios en una aplicación de chat.

```
class ChatRoomMediator:
    def __init__(self):
        self._users = []

    def add_user(self, user):
        self._users.append(user)

    def send_message(self, message, sender):
        for user in self._users:
            if user != sender:
                user.receive_message(message)

class User:
    def __init__(self, name, mediator):
        self.name = name
        self._mediator = mediator

    def send(self, message):
        print(f"{self.name} envía: '{message}'")
        self._mediator.send_message(message, self)

    def receive_message(self, message):
        print(f"{self.name} recibe: '{message}'")

# Uso del patrón Mediator
mediator = ChatRoomMediator()

john = User("John", mediator)
jane = User("Jane", mediator)
mike = User("Mike", mediator)

mediator.add_user(john)
mediator.add_user(jane)
mediator.add_user(mike)

john.send("¡Hola a todos!")
jane.send("Saludos, John y Mike!")
```

En este ejemplo, la clase **ChatRoomMediator** actúa como el mediador, gestionando el envío de mensajes entre los objetos **User**. Ningún usuario necesita conocer directamente a otro; solo se comunican a través del mediador. Esto mantiene a los componentes **User** simples y reutilizables.