En los siguientes tres ejercicios deberá:

- Identificar el tipo de patrón (Estructural, comportamiento, creacional)
- Seleccionar el patrón que considera que es.
- Diseñar el diagrama de clases de la solución.
- Desarrollar el código de los tres ejercicios.

Recuerde que deberá entregar la solución de código en un único lenguaje. Los lenguajes permitidos son Java, Python, C#, JS / TS .

1.

Escenario

Imagina que estás desarrollando una aplicación para una compañía automotriz que permite a los clientes personalizar y ordenar un automóvil. Un objeto Automóvil puede tener muchas configuraciones opcionales: tipo de motor, color, llantas, sistema de sonido, interiores, techo solar, navegación GPS, etc.

Problema

Crear un objeto Automóvil con múltiples configuraciones puede llevar a constructores con muchos parámetros (el infame "constructor telescópico") o a múltiples constructores sobrecargados, lo que dificulta el mantenimiento y legibilidad del código.

Beneficios esperados de la solución:

- Legibilidad y claridad: Facilitar la creación de objetos complejos con muchos parámetros sin necesidad de múltiples constructores o valores por defecto.
- Inmutabilidad: Una vez creado el objeto, sus propiedades no se pueden modificar si el constructor lo define como inmutable.
- Flexibilidad: Poder omitir atributos opcionales sin necesidad de crear subclases o múltiples constructores.
- Separación de construcción y representación: Separar la lógica de construcción del objeto en sí, facilitando modificaciones futuras.

2.

Escenario

Estás desarrollando una aplicación que gestiona la visualización de notificaciones en diferentes plataformas (por ejemplo: escritorio, móvil, web). Las notificaciones pueden ser de distintos tipos (mensaje, alerta, advertencia, confirmación) y cada tipo puede mostrarse de distintas formas según la plataforma.

Problema

Si usas herencia tradicional, tendrías que crear clases como:

- NotificacionMensajeWeb, NotificacionAlertaWeb, NotificacionMensajeMovil, NotificacionAlertaMovil, etc.

Esto lleva rápidamente a una explosión combinatoria de subclases difíciles de mantener.

Beneficios esperados de la solución:

- Separación de responsabilidades: Separar la lógica de la notificación del medio por el que se presenta.
- Escalabilidad: Poder agregar nuevas plataformas o tipos de notificación sin modificar el resto del sistema.
- Reducción de clases: Evitar la multiplicación de clases para cada combinación.
- Flexibilidad en tiempo de ejecución: Poder cambiar la plataforma dinámicamente si es necesario.

3.

Escenario

Estás desarrollando una aplicación de chat grupal. Los usuarios pueden enviarse mensajes entre sí dentro de una sala de chat. Sin embargo, gestionar las interacciones directas entre cada usuario haría que cada uno deba conocer y comunicarse con todos los demás, lo que resulta en una alta dependencia entre objetos.

Problema

Sin un mediador, cada usuario tendría que mantener referencias directas a todos los demás, lo que genera un sistema difícil de escalar y mantener. Si agregas o eliminas usuarios, debes actualizar muchas relaciones.

Beneficios esperados de la solución:

1. Facilita el mantenimiento**:** Agregar o eliminar usuarios no debe requerir modificar los demás.
2. Mejor organización**:** La lógica de comunicación debe estar centralizada, no dispersa en muchos objetos.
3. Reduce la complejidad**:** Evitar una red enmarañada de interacciones punto a punto.