PARCIAL 2 Ingeniería de Software 2 – Juan de Dios Rodriguez Perez 2210004

**Caso cotidiano propuesto**: Aplicación de pedidos para una cafetería (**CoffeeApp**)

**CoffeeApp** es una aplicación para una cafetería moderna que permite a los usuarios consultar el menú, personalizar sus bebidas, realizar pedidos y recibir notificaciones cuando su pedido esté listo. El sistema se implementa bajo una arquitectura basada en microservicios, con componentes independientes para menú, pedidos, personalización, notificaciones y usuarios.

Un sistema en el que los usuarios pueden:

* Ver el menú
* Personalizar bebidas
* Hacer pedidos
* Ver el estado del pedido
* Recibir notificaciones cuando el pedido está listo

**Microservicios clave:**

* Servicio de Menú
* Servicio de Pedidos
* Servicio de Personalización
* Servicio de Notificaciones
* Servicio de Usuarios

**Patrones propuestos:**

1. Strategy Pattern (Servicio de Personalización)

¿Por qué? Porque puedes tener diferentes estrategias para preparar una bebida: con leche, sin leche, leche de almendra, más azúcar, sin azúcar, etc. Cada tipo de personalización puede ser una estrategia que se aplica a una bebida base.

Encapsulas algoritmos intercambiables (cada ingrediente es una estrategia) y los aplicas a un objeto Bebida sin modificar su estructura. Esto permite seleccionar dinámicamente una o varias estrategias en tiempo de ejecución.

Ventajas:

Flexibilidad para agregar nuevas estrategias sin cambiar código existente.

Separación de lógica de negocio y opciones de personalización.

Código más mantenible y escalable

2. Observer Pattern (Servicio de Notificaciones)

¿Por qué? Porque los usuarios pueden suscribirse a eventos del sistema (por ejemplo: “tu pedido está listo”) y recibir notificaciones automáticas. Ideal para un patrón publicador-suscriptor.

cada estrategia decora al objeto bebida, añadiendo funcionalidad como costo adicional o descripción, sin alterar su clase original. Cada ingrediente envuelve al objeto bebida, agregando comportamiento de forma dinámica.

Ventajas:

Añadir múltiples características sin modificar la clase original.

Composición flexible de comportamiento.

Admite combinaciones complejas de personalización.

**Antipatrón propuesto:** God Object

¿Dónde? En el Servicio de Pedidos (mal implementado).

¿Por qué?

Porque concentra demasiadas responsabilidades en un solo módulo o clase, violando el principio de Responsabilidad Única. Esto dificulta el mantenimiento, testing y escalabilidad

**Arquitectura microservicios**

| **Servicio** | **Puerto** | **Función Principal** |
| --- | --- | --- |
| Menu | 3001 | Muestra las bebidas disponibles |
| Pedido | 3002 | (Pendiente revisar, pero se asume gestiona pedidos) |
| Notificación | 3004 | (Probablemente envía alertas o confirmaciones) |
| Usuario | 3005 | (Gestión de usuarios, no visto aún) |
| Personalización | 4002 | Aplica estrategias de personalización a bebidas |

**Patrón Strategy**

Implementado en el microservicio de personalización (aplicarEstrategias).

Cada tipo de estrategia como leche, azúcar, canela, se puede representar como una estrategia que modifica el comportamiento de la bebida.

Este patrón te permite agregar más estrategias fácilmente sin modificar el código base.

**Patrón Decorator**

Si cada estrategia envuelve a la bebida base para agregar funcionalidad (como en un patrón Decorator clásico), este patrón también se estaría aplicando.

Ejemplo típico: una clase LecheDecorator que extiende el comportamiento de Bebida.

**Antipatrones:**

**God Object**: Si una clase como Cafe hace demasiado (por ejemplo, contiene lógica de precios, estrategias y control de estado).

**Hardcoded Values**: Si defines lógica o decisiones clave con strings directamente en múltiples lugares (if (tipo === 'cafe')).

**Spaghetti Code**: Si el flujo de control y responsabilidad está poco claro.

Uml:

Diagrama

El contenido generado por IA puede ser incorrecto.