**PROBLEMAS DE PATRONES DE DISEÑO**

**Código Fuente:** https://github.com/mstealth/Patrones

1. **Patron Prototype**

* Facilita la clonación de objetos complejos (en este caso, héroes con habilidades, nivel, experiencia, etc·)
* Permite la creación de copias personalizables de un objeto base
* Mejora la eficiencia al clonar objetos sin reconfigurarlos desde cero
* **Ventajas del Patrón Prototype:**
  + **Eficiencia en la creación de objetos:** Evita la recreación desde cero al clonar objetos existentes, lo que es especialmente útil cuando la inicialización de un objeto es costosa
  + **Personalización sencilla:** Los clones pueden modificarse después de ser creados sin afectar el objeto original
  + **Independencia de la clase concreta:** Permite trabajar con objetos sin necesidad de conocer la clase exacta, ya que se basa en la interfaz clone()
  + **Flexibilidad:** Facilita la extensión del sistema al agregar nuevas clases de héroes sin cambiar el código existente

Diagrama

Descripción generada automáticamente

1. **Implementación del Patrón Observer para Mensajería en Tiempo Real**

* **Notificación en tiempo real:** El patrón Observer permite que todos los dispositivos suscritos (observadores) reciban una notificación en tiempo real cuando el usuario recibe un nuevo mensaje
* **Desacoplamiento**: La aplicación de mensajería (sujeto o publicador) no necesita conocer detalles específicos de cada dispositivo, sino solo notificar a todos los dispositivos suscritos· Esto ayuda a mantener la lógica de notificación independiente de los detalles de cada dispositivo
* **Flexibilidad**: Los dispositivos pueden suscribirse o cancelar su suscripción en tiempo de ejecución, lo cual es posible gracias a la estructura del patrón Observer·
* **Explicación del Código**
* **Interfaz IObservador:** Define el método Update que cada dispositivo implementará para recibir notificaciones
* **Dispositivo**: Representa un dispositivo que implementa la interfaz IObservador· Cuando recibe una notificación, simplemente muestra el mensaje
* **Clase Usuario**: Implementa Maneja una lista de dispositivos suscritos· Cuando un nuevo mensaje llega, notificarDispositivos llama a Update en cada dispositivo·
* **Ventajas**
* Desacoplamiento: La aplicación no tiene que saber cuántos o cuáles dispositivos están conectados
* Flexibilidad: Puedes añadir o eliminar dispositivos en tiempo de ejecución sin modificar la lógica principal
* Escalabilidad: Puedes agregar tantos dispositivos como necesites sin cambios en la estructura principal·

Diagrama

Descripción generada automáticamente

1. **Patrón de Diseño Template Method:** Este patrón permite definir la estructura general de un algoritmo en una clase base y delegar la implementación de pasos específicos a subclases.
   * **Encapsulación de Estructura Común:** El Template Method define una secuencia común para el procesamiento de pedidos (toma de pedido, verificación, preparación, etc.) en la clase base, eliminando duplicación de código.
   * **Flexibilidad para Pasos Específicos:** Permite que subclases (como OrderFood, OrderBeverage, OrderDessert) redefinan solo aquellos pasos que requieren un comportamiento específico.
   * **Reutilización del Código:** Los pasos comunes están centralizados en la clase base, reduciendo la duplicación de lógica en las subclases.
2. **Patrón Command:** es ideal para encapsular solicitudes como objetos, permitiendo registrar, deshacer (revertir) y ejecutar acciones de manera desacoplada.

* **Desacoplamiento:** Desacopla el invocador (cliente) del receptor (quien ejecuta la acción), ya que encapsula las acciones en comandos.
* **Extensibilidad:** Es fácil agregar nuevas operaciones simplemente creando nuevos comandos sin modificar el código existente.
* **Registro de acciones:** Cada comando se puede almacenar en una lista o pila, lo que facilita realizar operaciones de deshacer o rehacer.
* **Flexibilidad:** Permite ejecutar comandos de forma diferida o remota, y agrupar comandos para ejecutar varias acciones como una sola operación.

1. **Patrón Decorator:** permite agregar dinámicamente nuevas funcionalidades a un objeto existente sin modificar su estructura básica, proporcionando una alternativa flexible a la herencia.
   * **Extensibilidad Dinámica:** Las habitaciones pueden decorarse de forma personalizada, añadiendo servicios adicionales sin modificar la clase principal de la habitación.
   * **Flexibilidad sin Subclases:** Podemos crear decoraciones y mejoras adicionales sin generar múltiples subclases para cada combinación posible de decoraciones.
   * **Separación de Responsabilidades:** Cada mejora (por ejemplo, flores frescas, chocolate, vino) se maneja en clases independientes, facilitando la lectura y el mantenimiento del código.

**SOLUCION SOLID VIOLATION 2**

# **Falencias encontradas**

## **Single Responsibility Principle – SRP:**

La clase Order tiene múltiples responsabilidades:

* Almacenamiento en la base de datos
* Generación de reportes

La función saveOrder() en la clase Order viola SRP

* Delega la persistencia a la clase DatabaseMysql

La función generateOrderReport() también viola el SRP

* La generación de reportes debería estar separada de la lógica de la orden-

## **Open-Closed Principle - OCP**

La clase Order no está abierta para extensiones sin modificaciones, especialmente en la forma en que se maneja la persistencia. Si se quisiera cambiar el tipo de base de datos, habría que modificar la clase Order, lo cual es contrario al OCP.

## **Liskov Substitution Principle - LSP**

No aplica debido a que no se está utilizando herencia.

## **Interface Segregation Principle - ISP**

No aplica , debido a que actualmente, no hay interfaces implementadas

## **Dependency Inversion Principle - DIP**

La clase Order depende directamente de DatabaseMysql, lo que viola el DIP. En su lugar, debería depender de una abstracción (como una interfaz), para que los detalles de implementación sean inyectados y no estén acoplados a una clase específica de la base de datos.

## **Reestructuración para Cumplir con SOLID**

## Para resolver las violaciones de los principios SOLID, se han aplicado los siguientes cambios:

## **Aplicación del SRP**

Se separan las responsabilidades de almacenamiento y generación de reportes en clases distintas. La clase Order se centrará únicamente en almacenar los detalles de la orden.

## **Aplicación del OCP**

Se crea una interfaz OrderRepository para definir el contrato de almacenamiento de órdenes, permitiendo implementar distintas estrategias de persistencia (como DatabaseMysqlRepository).

## **Aplicación del DIP**

La clase Order recibe una instancia de OrderRepository para desacoplar la dependencia de la clase DatabaseMysql.

## **Adición de ISP (Interface Segregation)**

Se introducen interfaces específicas (OrderRepository y ReportGenerator) para segregar las funcionalidades de almacenamiento y generación de reportes.