EJERCICIOS PATRONES DE DISEÑO #1

**Ejercicio 1:

En una aplicación web, necesitas acceder a una configuración global (por ejemplo, la URL de la base de datos o las claves API) desde cualquier lugar de la aplicación. La configuración debe ser única para toda la aplicación y debe garantizarse que no se creen múltiples instancias.

1. **¿Qué patrón usarías?**
2. Explica brevemente tu razonamiento.

**Ejercicio 2:

Un sistema de comercio electrónico gestiona el ciclo de vida de los pedidos. Un pedido puede estar en varios estados como: "Pendiente", "Pagado", "Enviado", "Entregado". Según su estado, el pedido tiene diferentes comportamientos: por ejemplo, solo se puede cancelar si está en estado "Pendiente".

1. **¿Qué patrón usarías?**
2. Justifica tu respuesta.

**Ejercicio 3:

Una aplicación móvil tiene un sistema de notificaciones que alerta a los usuarios sobre cambios en sus datos, como nuevas publicaciones o mensajes. Varios componentes del sistema, como los servicios de mensajes y la interfaz de usuario, necesitan ser notificados automáticamente cuando se producen cambios.

1. **¿Qué patrón usarías?**
2. Explica cómo implementarías este patrón en el sistema.

**Ejercicio 4:

En una aplicación bancaria, necesitas interactuar con varios subsistemas como el de cuentas, préstamos y tarjetas de crédito. En lugar de permitir que los desarrolladores llamen directamente a estos subsistemas, quieres proporcionar una interfaz única y simple para realizar estas operaciones.

1. **¿Qué patrón usarías?**

2. ¿Cómo estructurarías las clases para lograr esto?

**Ejercicio 5:

Un sistema de gestión de inventarios muestra imágenes de los productos en una interfaz gráfica. Estas imágenes se almacenan en un servidor remoto, y la carga inicial de todas las imágenes puede ser costosa en términos de tiempo y recursos. Solo quieres cargar las imágenes de los productos que el usuario esté visualizando.

1. **¿Qué patrón usarías?**
2. Justifica tu elección.

**Ejercicio 6:

En un videojuego, los jugadores pueden elegir entre diferentes tipos de vehículos como "Coche", "Moto" o "Camión". Necesitas garantizar que la creación de estos objetos sea flexible y extensible, de manera que en el futuro puedan agregarse nuevos tipos de vehículos sin afectar al código existente.

1. **¿Qué patrón usarías?**
2. ¿Cómo diseñarías la clase principal de creación?

RESPUESTAS

¡Perfecto! Aquí tienes las respuestas a los ejercicios junto con el razonamiento para identificar cada patrón.

Ejercicio 1: Configuración de Aplicación

1. **Patrón Aplicado**: Singleton
2. **Razonamiento**:
   - **Problema clave**: Garantizar que solo haya una única instancia de la configuración global.
   - **Indicadores**: Se necesita un punto de acceso único y global para un recurso que debe ser compartido en toda la aplicación.
   - **Implementación**: El patrón Singleton asegura una única instancia y proporciona un método de acceso controlado.

Ejercicio 2: Gestión de Estados de un Pedido

1. **Patrón Aplicado**: State
2. **Razonamiento**:
   - **Problema clave**: Los comportamientos del pedido cambian dinámicamente dependiendo de su estado (Pendiente, Pagado, etc.).
   - **Indicadores**: La lógica específica de cada estado debe mantenerse separada y facilitar cambios futuros.
   - **Implementación**: Se crea una clase para cada estado (como `EstadoPendiente`, `EstadoPagado`) que encapsula su comportamiento específico. El pedido utiliza un objeto de estado para delegar su comportamiento.

Ejercicio 3: Sistema de Notificaciones

1. **Patrón Aplicado**: Observer
2. **Razonamiento**:
   - **Problema clave**: Notificar automáticamente a varios componentes cuando se producen cambios en los datos.
   - **Indicadores**: Se necesita una relación de uno-a-muchos entre el objeto observado (datos) y los observadores (componentes como UI y servicios).
   - **Implementación**: El sujeto (por ejemplo, `SistemaNotificaciones`) mantiene una lista de observadores (`ObservadorUI`, `ObservadorMensajes`) y los notifica cuando cambia.

Ejercicio 4: Acceso a Subsistemas Bancarios

1. **Patrón Aplicado**: Facade
2. **Razonamiento**:
   - **Problema clave**: Ocultar la complejidad de interactuar con múltiples subsistemas (cuentas, préstamos, tarjetas) y proporcionar una interfaz simplificada.
   - **Indicadores**: Varios subsistemas deben ser accesibles desde un único punto.
   - **Implementación**: La clase `BancoFacade` proporciona métodos como `ConsultarSaldo()`, que internamente interactúan con las clases de los subsistemas (`Cuentas`, `Préstamos`, `Tarjetas`).

Ejercicio 5: Carga de Imágenes de Productos

1. **Patrón Aplicado**: Proxy
2. **Razonamiento**:
   - **Problema clave**: Cargar las imágenes de manera diferida solo cuando se necesitan.
   - **Indicadores**: Controlar el acceso y reducir el costo de la carga inicial.
   - **Implementación**: La clase `ProxyImagen` actúa como intermediaria entre el cliente y la imagen real, cargándola solo cuando se solicita.

Ejercicio 6: Creación de Vehículos

1. **Patrón Aplicado**: Factory Method
2. **Razonamiento**:
   - **Problema clave**: Permitir la creación flexible de vehículos (Coche, Moto, Camión) y facilitar la adición de nuevos tipos sin modificar el código existente.
   - **Indicadores**: La lógica de creación varía según el tipo de vehículo.
   - **Implementación**: La clase `FabricaVehiculos` tiene un método factory que devuelve objetos específicos como `Coche`, `Moto` o `Camión`.

Resumen de las Respuestas

Ejercicio	Patrón Aplicado	Indicadores Clave
1	Singleton	Única instancia compartida y acceso global.
2	State	Cambios dinámicos de comportamiento según el estado interno.
3	Observer	Notificaciones automáticas a múltiples observadores.
4	Facade	Simplificación del acceso a subsistemas complejos.
5	Proxy	Control de acceso y carga diferida de recursos costosos.
6	Factory Method	Creación flexible y extensible de objetos, soportando nuevos tipos en el futuro.