Es esto:Proyecto de Sistema de Gestión para Peluquería: Explicación y Aplicación de Principios SOLID

1. Principio de Responsabilidad Única (SRP)

El Principio de Responsabilidad Única (Single Responsibility Principle) establece que cada clase debe tener una única razón para cambiar, es decir, debe encargarse de una sola responsabilidad. En nuestro proyecto, este principio se aplicó de la siguiente manera:

Cliente: Esta clase se encarga exclusivamente de gestionar la información de los clientes, incluyendo nombre, correo electrónico y teléfono. No tiene ninguna lógica relacionada con reservas, servicios o inventario, lo que permite que si en el futuro se desea cambiar la forma de almacenar datos de los clientes, solo se modifique esta clase.

Estilista: Gestiona únicamente los datos de los estilistas y sus especialidades. Cada estilista puede ofrecer varios servicios, pero la lógica de reservas y disponibilidad está en otra clase.

Producto: Maneja los datos de los productos, incluyendo precio y stock, y cuenta con un método para descontar unidades (consume). No se mezcla con la lógica de ventas o servicios.

Reserva: Se encarga de almacenar los detalles de una reserva, como cliente, estilista, servicio, fecha, estado y notas. También tiene métodos para confirmar o cancelar reservas.

ReservaService: Gestiona la lógica de creación, cancelación y reprogramación de reservas, incluyendo validaciones y disponibilidad de estilistas.

InventarioService: Gestiona el stock de productos consumidos en los servicios, con métodos para consumir y reponer productos.

ReportService: Genera reportes de reservas en JSON y CSV, sin modificar ninguna otra lógica del sistema.

Repositorios (ClienteRepository, EstilistaRepository, ReservaRepository): Gestionan la persistencia de datos, ya sea en memoria o archivos JSON.

Ventaja: Al tener cada módulo una única responsabilidad, se facilita el mantenimiento y las pruebas unitarias, y se minimiza el riesgo de introducir errores al modificar funcionalidades. Por ejemplo, si se cambia la forma de guardar clientes, no se afecta la lógica de reservas o inventario.

2. Principio de Abierto/Cerrado (OCP)

El Principio Abierto/Cerrado (Open/Closed Principle) indica que las clases deben estar abiertas a extensión pero cerradas a modificación, es decir, se pueden agregar nuevas funcionalidades sin alterar el código existente.

En el proyecto:

La clase Servicio es abstracta y actúa como plantilla para todos los servicios que ofrece la peluquería.

Las clases Corte, Tinte y Barba extienden Servicio y definen la descripción específica de cada servicio, reutilizando la lógica base.

Para agregar un nuevo servicio, como un tratamiento capilar o masaje de barba, solo se crea una nueva clase que extienda Servicio sin tocar la lógica de reservas o inventario existentes.

De forma similar, ReportService puede extenderse para generar reportes PDF o HTML sin modificar la lógica de reservas.

Beneficio: Esto permite escalar el sistema sin romper funcionalidades existentes, haciendo que la aplicación sea más robusta y flexible a cambios futuros.

3. Principio de Sustitución de Liskov (LSP)

El Principio de Sustitución de Liskov establece que los objetos de una clase derivada deben poder sustituir a los de la clase base sin alterar el funcionamiento del sistema.

En nuestro proyecto, Corte, Tinte y Barba extienden Servicio. Todas estas clases implementan el método descripcion() y los getters necesarios (getPrecio(), getDuracion()), cumpliendo la interfaz de la clase base.

Esto permite que la clase Reserva o ReservaService pueda manejar cualquier tipo de servicio sin preocuparse del tipo concreto:

4. Principio de Segregación de Interfaces (ISP)

El Principio de Segregación de Interfaces recomienda que las interfaces sean específicas y no obliguen a las clases a implementar métodos que no utilizan.

IRepository<T> define métodos esenciales: crear, listar, actualizar, eliminar y buscarPorCampo.

Cada repositorio (ClienteRepository, EstilistaRepository, ReservaRepository) implementa solo estos métodos, sin sobrecargar la clase con funcionalidades innecesarias.

ReservaService depende únicamente de los métodos que realmente necesita (listar, crear), evitando acoplarse a métodos que no se usan.

Beneficio: Reduce la complejidad y hace que cada módulo sea más ligero y fácil de mantener.

5. Principio de Inversión de Dependencias (DIP)

El Principio de Inversión de Dependencias establece que las clases de alto nivel no deben depender de clases concretas, sino de abstracciones.

ReservaService depende de la interfaz IRepository<T>, no de repositorios concretos.

Esto permite que ClienteRepository pueda ser reemplazado por otro tipo de almacenamiento (memoria, JSON, base de datos) sin modificar ReservaService.

Uso de exports, herencia y condicionales

Exports

Cada clase o interfaz se exporta desde su módulo (Cliente.ts, Reserva.ts, Servicio.ts) y se importa donde se necesita (main.ts o index.ts).

Esto garantiza modularidad, permitiendo reutilizar las clases en distintos contextos sin duplicación de código.

Herencia

Servicio es la clase base abstracta.

Corte, Tinte y Barba heredan métodos y atributos, como getPrecio, getDuracion y descripcion.

Esto permite polimorfismo, es decir, tratar distintos tipos de servicios de la misma manera, simplificando la lógica de reservas.

Condicionales

Se utilizan condicionales para validar datos y estados, garantizando seguridad y coherencia del sistema