UNIVERSIDAD CATOLICA BOLIVIANA SAN PABLO

MAESTRÍA EN INGENIERÍA SOFTWARE AVANZADA 1V.

PROYECTO FINAL

MÓDULO V: DESIGN PRINCIPLES AND PATTERNS

Autores: **ARIEL BENJAMIN TORRICOS PADILLA**

Sucre – Bolivia

2025

1. Descripción de la aplicación

En esta aplicación, desarrollada en NestJS, se optó por una arquitectura hexagonal, la cual facilita una mejor segregación de responsabilidades entre sus diversas capas. La aplicación cuenta con distintos dominios, que corresponden a cada patrón los cuales son Builder para Solicitudes de compra, patrón composite para las categorías de artículos, patrón decorator para la extensión de productos, el patrón factory para la creación de productos distintos, el observer para las notificaciones del inventario, el patrón strategy para el cálculo de precios así también incluyendo autenticación, permitiendo una ágil transición entre diferentes módulos y una coneccion a base de datos en PostgreSQL con el orm Prisma. El modelo del negocio se encuentra centralizado en la capa del Dominio, donde se separó cada funcionalidad de los patrones incorporando una lógica de validación y objetos de valor. Los controladores y los DTOs (Data Transfer Objects) están claramente separados en la capa de Presentación. La adopción de una arquitectura hexagonal promueve la independencia del Dominio la cual nos facilita la aplicación de cada uno de estos patrones en la aplicación de otras dependencias externas, como frameworks, bases de datos o interfaces de usuario. Esto se logra a través de la definición de puertos e interfaces que el core implementa y los adaptadores en las capas externas. Los puertos definen las interacciones del dominio con el exterior, mientras que los adaptadores traducen las solicitudes y respuestas entre el core y las tecnologías externas. Esta separación facilita la mantenibilidad, la flexibilidad y la extensibilidad.

El flujo de trabajo típico que realiza la aplicación es el siguiente.

1. Factory crea productos → **2.** Composite organiza en categorías → **3.** Builder construye solicitudes → **4.** Decorator aplica modificaciones → **5.** Strategy calcula precios → **6.** Observer notifica cambios

2. Patrones Creacionales

2.1. Factory Method - Creación de Productos

El patrón Factory Method define una interfaz para crear objetos, pero permite que las subclases decidan qué clase instanciar. En lugar de llamar directamente al constructor, se utiliza un método factory.

2.1.1. **Problema.-** En una tienda online los productos pueden ser de distintos tipos en este caso tenemos productos físicos,

digitales, y perecederos pues cada uno de estos tiene características y comportamientos específicos; **los productos básicos** requieren almacenamiento y cuentan con características como peso dimensiones (alto ancho y profundidad), los **productos digitales** son licencias y no tienen almacenamiento físico pero sí cuentan con url o pesos de descarga, y por último los **productos perecederos** cuentan con fecha de vencimiento y requieren condiciones especiales de almacenamiento como refrigeración y otros.

2.1.2. **Solucion.-** El patrón Factory nos permite solucionar este problema la cual nos facilita la adición de nuevos tipos de productos ya que al aplicar el encapsulamiento en la creación de estos productos nos proporciona flexibilidad y mantenibilidad para cada tipo de producto.

Link diagrama **Factory** para **Productos**

2.2. Builder - Construcción de Solicitudes de Compra

El patrón Builder permite construir objetos complejos paso a paso. Es especialmente útil cuando el objeto tiene muchos parámetros opcionales o cuando el proceso de construcción debe seguir una secuencia específica.

2.2.1. **Problema.-** las solicitudes de compra son objetos complejos los cuales incluyen múltiples productos con diferentes cantidades y a cada producto se le puede aplicar descuentos, fecha de compra como de vencimiento y otras configuraciones especiales, así como un facturación de toda la compra realizada y para solventar este caso se decidió usar un patrón builder.

2.2.2. **Solución.-** El patrón Builder nos ayuda en la construcción de una solicitud flexible con un control de validación durante la construcción de la solicitud también nos permite reutilizar los diferentes tipos de solicitudes que se puedan crear, y esto hace que el código sea más fácil de entender.

Link diagrama **Builder** para las **solicitudes de Compra**

3. Patrones Estructurales

3.1. Composite - Estructura de Categorías

El patrón composite permite tratar objetos individuales (artículos) que serian las hojas y composiciones de objetos (categorías) de manera uniforme. Organiza objetos en estructuras de árbol para representar jerarquías parte-todo.

3.1.1. Problema.- En un tienda online la organización jerárquica de los productos los cuales son categorías principales como electrónicos, ropa, utensilios del hogar, cada una de estas con subcategorías como para electrónicos serán los celulares o smartphones laptops y otros electronicos y cada uno con productos individuales como iphones o samsung y otras marcas de celulares y laptops.

3.1.2. Solucion.- El patrón composite nos ayuda en el manejo jerárquico de las categorías subcategorías de productos así como en el cálculo de inventario y valores totales para categorías completas, con un búsqueda recursiva en la estructura de productos y este patrón simplifica el manejo jerárquico.

Link diagrama **Composite** para las **categorias y sub categorias**

3.2. Decorator - Extensión de Productos

El patrón Decorator permite agregar nuevas funcionalidades a objetos de forma dinámica sin alterar su estructura. Es una alternativa flexible a la herencia.

3.2.1. Problema.- La extensión de productos cuentan con descuentos, impuestos y promociones para los distintos productos así como los cálculos finales la aplicación de impuestos con estos modificadores y los detalles que implica para cada usuario.

3.2.2. Solucion.- El patrón decorator nos ayuda a agregar múltiples modificadores de precios (descuentos, impuestos y

promociones) y combinarlos de una forma más flexible y evitar la exposición de subclases para cada combinación posible.

Link diagrama **Decorator** para la **extensión de precion de Productos**

4. Patrones de Comportamiento

4.1. Observer - Notificaciones de Inventario

El patrón Observer define una dependencia uno-a-muchos entre objetos, de manera que cuando un objeto cambia de estado, todos sus dependientes son notificados automáticamente.

4.1.1. Problema.- El monitoreo de inventario de una tienda online debe ser constante y hacer una verificación del Stock bajo y mandar alertas para el reabastecimiento también proveer con notificación de stock agotado y notificar a los proveedores así como proveer de reportes de inventarios para el análisis.

4.1.2. Solucion.- El patrón Observer con la dependencia del inventario de las otras clases no da una reacción a los cambios en el inventario de forma desacoplada múltiples clases serán notificadas sobre los cambios para realizar notificaciones reabastecimientos y reportes de inventarios así como permitir la agregación de nuevos observadores sin modificar los parámetros.

Link diagrama **Observer** para las **Notificación de Inventarios**

4.2. Strategy - Estrategias de Precios

El patrón Strategy define una familia de algoritmos, los encapsula y los hace intercambiables. Permite que el algoritmo varíe independientemente de los clientes que lo usan.

4.2.1. Problema.- Se necesita calcular el precio con diferentes estrategias (regular, mayorista, promocional), así como la aplicación de descuentos e impuestos según el contexto,y tener una flexibilidad para cambiar la estrategia de precios en tiempo

de ejecución.

4.2.2. **Solucion.-** El patrón Strategy no ayuda en los diferentes escenarios con diferentes algoritmos de cálculos de precios pues nos dan la facilidad de cambiar entre las distintas estrategia de precios y mantienen una lógica de cálculo separada del contexto del que la usa.

Link diagrama **Strategy** para establecer **distintas estrategias de precios**

LINK REPOSITORIO TIENDA ONLINE