TALLER: LIBRERÍA “CLEAN CODE”

Ingeniería de Sistemas

Curso: Calidad de Software
Docente: Jose Llanos
Institución: CORHUILA
Año: 2026

Integrantes:

- Juan Esteban Oliveros Duran
- Daniel Stiven Poveda Cante
- Charith Nikool Chavarro Meneses
- Javier Alfonso Correa Bolivar

20 febrero 2026:

Tabla de Contenido

---

1. Introducción

El presente documento tiene como propósito analizar la implementación de buenas prácticas de Código Limpio en el desarrollo de un módulo backend para una librería.

Se evaluarán aspectos como el Principio de Responsabilidad Única, separación de responsabilidades, uso adecuado de excepciones, nombres significativos, inyección de dependencias y encapsulamiento.

El análisis permitirá evidenciar cómo estas prácticas mejoran la mantenibilidad, claridad y calidad del software.

---

2. Enunciado del Taller

Objetivo

Desarrollar un módulo backend para procesar la compra de libros aplicando rigurosamente las buenas prácticas estudiadas en clase.

Requerimientos Funcionales

- Validación: Verificar que el cliente y la orden contengan datos válidos.
- Inventario: Confirmar disponibilidad de stock antes de procesar.
- Cálculo: Calcular el precio total incluyendo impuestos (IVA 19%).
- Notificación: Simular el envío de un correo de confirmación al cliente.
- Manejo de Errores: Uso estricto de Excepciones, prohibido el uso de códigos de error numéricos.

---

3. Código Fuente en Java

```
import java.util.Optional;

/**
 * SISTEMA DE LIBRERÍA - EJEMPLO DE CÓDIGO LIMPIO
 */
public class BookStoreCleanSystem {

    public static void main(String[] args) {
        // 1. Configuración de dependencias (Inyección pura)
        InventoryService inventory = new InventoryService();
        NotificationService notifier = new NotificationService();
        OrderProcessor processor = new OrderProcessor(inventory, notifier);

        System.out.println("=== INICIO DE PRUEBAS DEL SISTEMA ===\n");

        // CASO DE PRUEBA 1: Flujo Exitoso
        try {
            Customer customer = new Customer("Juan Perez",
"juan@corhuila.edu.co");
            Book book = new Book("Clean Code", 50.00, "REF-9988");
            Order order = new Order(customer, book, 2);

            processor.processOrder(order);
            System.out.println("ÉXITO: Pedido procesado correctamente.");

        } catch (OrderException e) {
            System.err.println("ERROR: " + e.getMessage());
        }

        // CASO DE PRUEBA 2: Fallo por Stock
        try {
            Customer c2 = new Customer("Maria G.", "maria@mail.com");
            Book noStockBook = new Book("Libro Agotado", 20.00, "REF-0000");
            Order order2 = new Order(c2, noStockBook, 1);

            processor.processOrder(order2);

        } catch (OrderException e) {
            System.out.println("CONTROLADO: El sistema detectó falta de stock.");
```

```
            System.out.println("Mensaje: " + e.getMessage());
        }
    }

    // --- CAPA DE DOMINIO ---
    static class Customer {
        private final String name;
        private final String email;

        public Customer(String n, String e) {
            name = n;
            email = e;
        }

        public String getEmail() {
            return email;
        }
    }

    static class Book {
        private final String title;
        private final double basePrice;
        private final String sku;

        public Book(String t, double p, String s) {
            title = t;
            basePrice = p;
            sku = s;
        }

        public double getBasePrice() {
            return basePrice;
        }

        public String getSku() {
            return sku;
        }
    }

    static class Order {
        private final Customer customer;
        private final Book book;
        private final int quantity;

        public Order(Customer c, Book b, int q) {
            customer = c;
            book = b;
            quantity = q;
        }

        public Customer getCustomer() {
            return customer;
        }
```

```
        public Book getBook() {
            return book;
        }

        public int getQuantity() {
            return quantity;
        }
    }

    // --- EXCEPCIONES PERSONALIZADAS ---
    static class OrderException extends RuntimeException {
        public OrderException(String msg) {
            super(msg);
        }
    }

    static class OutOfStockException extends OrderException {
        public OutOfStockException(String sku) {
            super("Sin stock: " + sku);
        }
    }

    // --- SERVICIOS ---
    static class OrderProcessor {

        private static final double TAX_RATE = 1.19;

        private final InventoryService inventory;
        private final NotificationService notifier;

        public OrderProcessor(InventoryService inv, NotificationService notif) {
            this.inventory = inv;
            this.notifier = notif;
        }

        public void processOrder(Order order) {
            validateOrder(order);
            checkStock(order);
            double total = calculateTotal(order);
            completeTransaction(order, total);
        }

        private void validateOrder(Order order) {
            if (order == null)
                throw new OrderException("Orden nula");
        }

        private void checkStock(Order order) {
            if (!inventory.hasStock(order.getBook().getSku()))
                throw new OutOfStockException(order.getBook().getSku());
        }

        private double calculateTotal(Order order) {
            return order.getBook().getBasePrice() *
```

```
                    order.getQuantity() *
                    TAX_RATE;
        }

        private void completeTransaction(Order order, double total) {
            notifier.sendConfirmation(order.getCustomer(), total);
        }
    }

    static class InventoryService {
        public boolean hasStock(String sku) {
            return !"REF-0000".equals(sku);
        }
    }

    static class NotificationService {
        public void sendConfirmation(Customer c, double amount) {
            System.out.println("Enviando correo a: " + c.getEmail());
            System.out.println("Total confirmado: " + amount);
        }
    }
}
```

4. Resultados – Buenas Prácticas Implementadas

4.1 Principio de Responsabilidad Única (SRP)

Cada clase cumple una única responsabilidad:

- `Customer` representa al cliente.
- `Book` representa el libro.
- `Order` representa la orden.
- `InventoryService` gestiona inventario.
- `NotificationService` gestiona notificaciones.
- `OrderProcessor` orquesta el proceso de compra.

Esto mejora la cohesión y facilita el mantenimiento.

4.2 Inyección de Dependencias

```
InventoryService inventory = new InventoryService();
NotificationService notifier = new NotificationService();
OrderProcessor processor = new OrderProcessor(inventory, notifier);
```

El `OrderProcessor` recibe sus dependencias por constructor, reduciendo el acoplamiento y facilitando pruebas.

4.3 Uso de Excepciones en Lugar de Códigos de Error

```
static class OutOfStockException extends OrderException {
    public OutOfStockException(String sku) {
        super("Sin stock: " + sku);
    }
}
```

Se implementan excepciones personalizadas, cumpliendo el requerimiento de no usar códigos numéricos.

4.4 Métodos Pequeños y Enfocados

```
private void validateOrder(Order order)
private void checkStock(Order order)
private void calculateTotal(Order order)
private void completeTransaction(Order order, double total)
```

Cada método realiza una única tarea, mejorando claridad y mantenibilidad.

4.5 Uso de Constantes

```
private static final double TAX_RATE = 1.19;
```

Se evita el uso de números mágicos.

4.6 Encapsulamiento e Inmutabilidad

```
private final String name;
private final String email;
```

Los atributos son `private` y `final`, protegiendo el estado interno.

---

5. Conclusiones

El sistema implementa correctamente principios de Código Limpio como responsabilidad única, separación de funciones, uso adecuado de excepciones, encapsulamiento e inyección de dependencias.

Estas prácticas mejoran la claridad del código, reducen el acoplamiento y facilitan el mantenimiento del sistema.

El ejercicio demuestra que aplicar estándares de calidad desde el diseño inicial permite construir software más robusto, escalable y profesional.