Taller 1 - Librería Clean Code – BookStoreCleanSystem

Programa: Ingeniería De Sistemas

Asignatura: Calidad De Software

Profesor: Jose Miguel llanos Mosquera

Integrantes

María Sofia Aljure Herrera

María Juliana Ferro Bonilla

Jose Miguel Vera Garzón

Corporación Universitaria Del Huila Corhuila

Neiva – Huila

Febrero 2026

Contenido

Introducción

Cuando programamos, muchas veces nos concentramos únicamente en que el código funcione y olvidamos algo igual de importante: que sea claro y fácil de entender. El concepto de Código Limpio (Clean Code) nos enseña que un buen programa no solo debe cumplir su función, sino que también debe estar bien organizado y ser fácil de mantener.

En este taller trabajamos con un sistema backend para una librería llamada “Clean Code”, cuyo objetivo es procesar la compra de libros. El sistema valida la información del cliente, revisa si el libro tiene stock disponible, calcula el precio total con IVA del 19%, envía una notificación al cliente y maneja errores utilizando excepciones.

Más que enfocarnos solo en que el sistema funcione, el propósito fue analizar qué buenas prácticas se aplicaron en el código y cómo los conceptos vistos en clase se reflejan en una implementación real.

Objetivo General

Analizar la implementación de un sistema backend para la gestión de compras en una librería, identificando y evaluando las buenas prácticas de Código Limpio aplicando las buenas prácticas de programación que estudiamos.

Objetivos Específicos

1. Identificar las prácticas de código limpio presentes en la estructura del sistema.
2. Reconocer tanto las buenas como las malas prácticas detectadas en el código.
3. Comprender la importancia del código mantenible y estructurado en proyectos de software.

Requerimientos Funcionales

• **Validación:** El sistema debe verificar que el cliente y la orden tengan datos válidos antes de procesar nada.

• **Inventario:** Tiene que confirmar que hay stock disponible antes de aceptar la compra.

• **Cálculo:** Calcular el precio total incluyendo el IVA del 19%.

• **Notificación:** Simular el envío de un correo de confirmación al cliente.

• **Manejo de Errores:** Usar exclusivamente excepciones. Está prohibido usar códigos de error numéricos.

Código fuente

A continuación, está el código completo del sistema:

```
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 (Manejo de Errores)
        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); // Lanzará excepción

        } 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);
        }
    }
}
```

Resultados:

¿qué prácticas de código limpio se usaron?

Después de revisar el código, se pueden identificar varias prácticas importantes:

Buenas prácticas

1. **Nombres claros**

Las clases, atributos y métodos tienen nombres específicos y claros. Esto facilita comprender el propósito de cada componente sin necesidad de leer toda la implementación.

2. **Las funciones son cortas y especificas**

No se sobrepasan de las 20-30 líneas, tienen una única responsabilidad. Cada parte hace solo una cosa. EJE: Hay una parte que revisa el stock, otra que calcula el total, etc.

3. **El sistema esta divido por secciones**

Esta separado en parte de datos, una parte que procesa y una que maneja errores. Esto hace que sea más organizado y fácil de entender. Cada clase tiene una sola responsabilidad.

4. **Manejo de errores**

No va a devolver números raros como -1 o -2 ya que usa excepciones como: OrderException y OrderException

5. **Pocos parámetros**

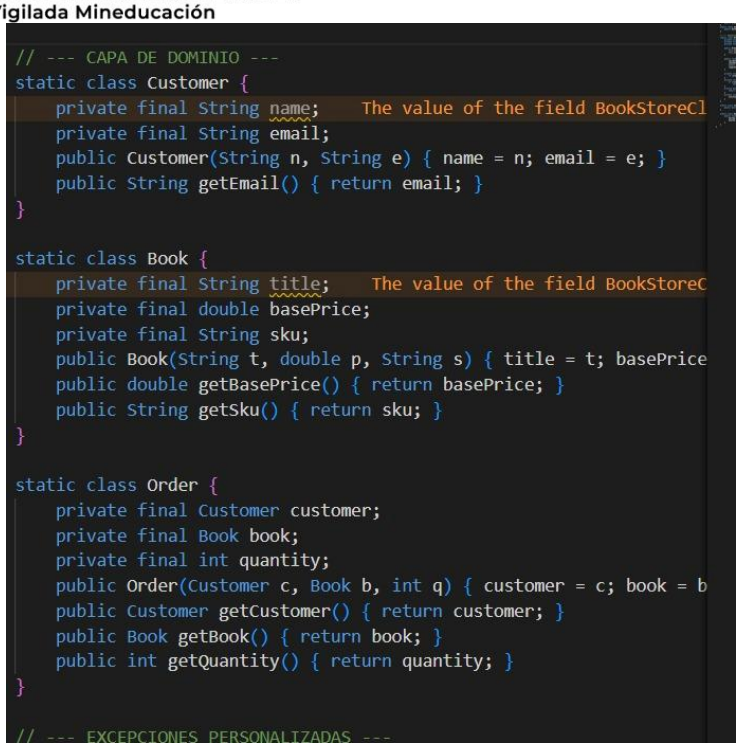
Los métodos no exceden el número de parámetros, como máximo tienen 3.

6. **Indentación consistente**

En el código se refleja la buena indentación en todo el proyecto para las diferentes funciones y métodos.

7. **Encapsulación**

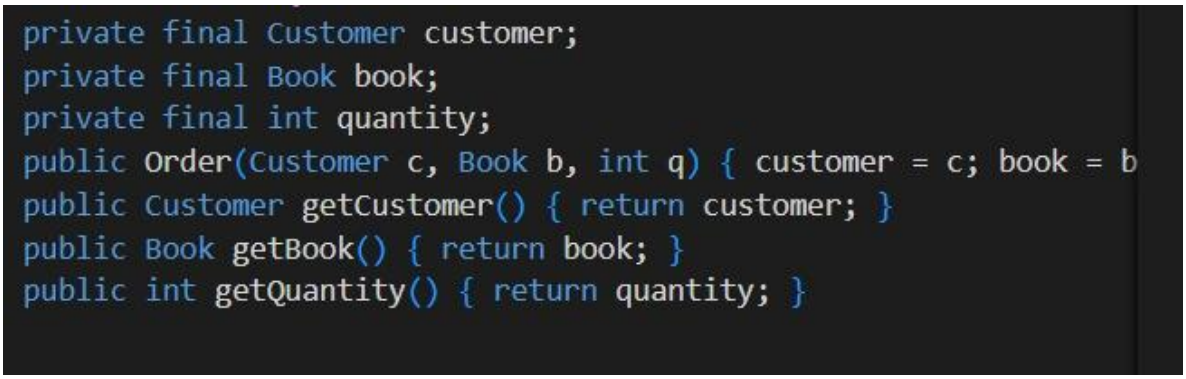
Los datos de las clases están protegidos, se usa PRIVATE para proteger los atributos.



```
// --- CAPA DE DOMINIO ---
static class Customer {
    private final String name;    The value of the field BookStoreCl
    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;    The value of the field BookStoreC
    private final double basePrice;
    private final String sku;
    public Book(String t, double p, String s) { title = t; basePrice
    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
    public Customer getCustomer() { return customer; }
    public Book getBook() { return book; }
    public int getQuantity() { return quantity; }
}

// --- EXCEPCIONES PERSONALIZADAS ---
```

Ilustración 1 Clases y atributos

Las clases, atributos y métodos tienen nombres específicos y claros. Esto facilita comprender el propósito de cada componente sin necesidad de leer toda la implementación.



```
private final Customer customer;
private final Book book;
private final int quantity;
public Order(Customer c, Book b, int q) { customer = c; book = b
public Customer getCustomer() { return customer; }
public Book getBook() { return book; }
public int getQuantity() { return quantity; }
```

Ilustración 2 ()

```
        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(msg: "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);
        }
    }
}
```

Ilustración 3 funciones cortas

La imagen muestra métodos pequeños y específicos que cumplen una sola responsabilidad, como validateOrder(), checkStock() y calculateTotal().

```
43
44  // --- CAPA DE DOMINIO ---
45  static class Customer {
46      private final String name;    The value of the field BookStoreCleanSystem.Customer.name is not used
47      private final String email;
48      public Customer(String n, String e) { name = n; email = e; }
49      public String getEmail() { return email; }
50  }
51
52  static class Book {
53      private final String title;    The value of the field BookStoreCleanSystem.Book.title is not used
54      private final double basePrice;
55      private final String sku;
56      public Book(String t, double p, String s) { title = t; basePrice = p; sku = s; }
57      public double getBasePrice() { return basePrice; }
58      public String getSku() { return sku; }
59  }
60
61  static class Order {
62      private final Customer customer;
63      private final Book book;
64      private final int quantity;
65      public Order(Customer c, Book b, int q) { customer = c; book = b; quantity = q; }
66      public Customer getCustomer() { return customer; }
67      public Book getBook() { return book; }
68      public int getQuantity() { return quantity; }
69  }
70
```

Ilustración 4 Esta divido por partes (dominio)

Se observa la separación del sistema en secciones organizadas (Dominio, Excepciones y Servicios), lo que mejora la estructura y legibilidad del código.

```
// --- 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); }
}
```

Ilustración 5 excepciones

Se muestran las excepciones personalizadas OrderException y OutOfStockException, lo que evidencia un manejo claro y estructurado de errores.

```
// --- 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(msg: "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);
    }
}
```

Ilustración 6 Servicios

Se visualizan las clases InventoryService y NotificationService, cada una con una responsabilidad específica dentro del sistema.

```
// --- 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
}
```

Ilustración 7 Manejo de errores

En esta imagen se observa el uso de bloques try-catch para controlar excepciones, evitando el uso de códigos de error numéricos.

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

Ilustración 8 Pocos parámetros

Los métodos no reciben una cantidad excesiva de parámetros, lo que mejora la claridad y reduce la complejidad del código.

```
// --- 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(msg: "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
    }
    private void completeTransaction(Order order, double total) {
        notifier.sendConfirmation(order.getCustomer(), total);
    }
}
```

Ilustración 9 Indentación

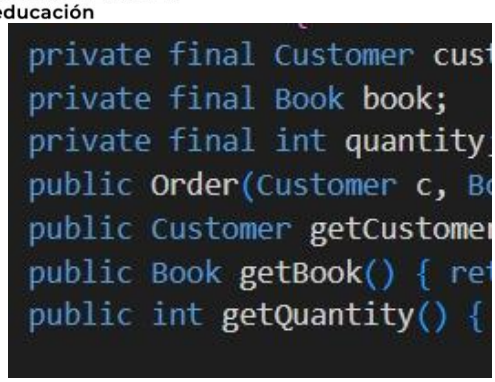


Ilustración 10 Encapsulación (public o private)

Malas practicas

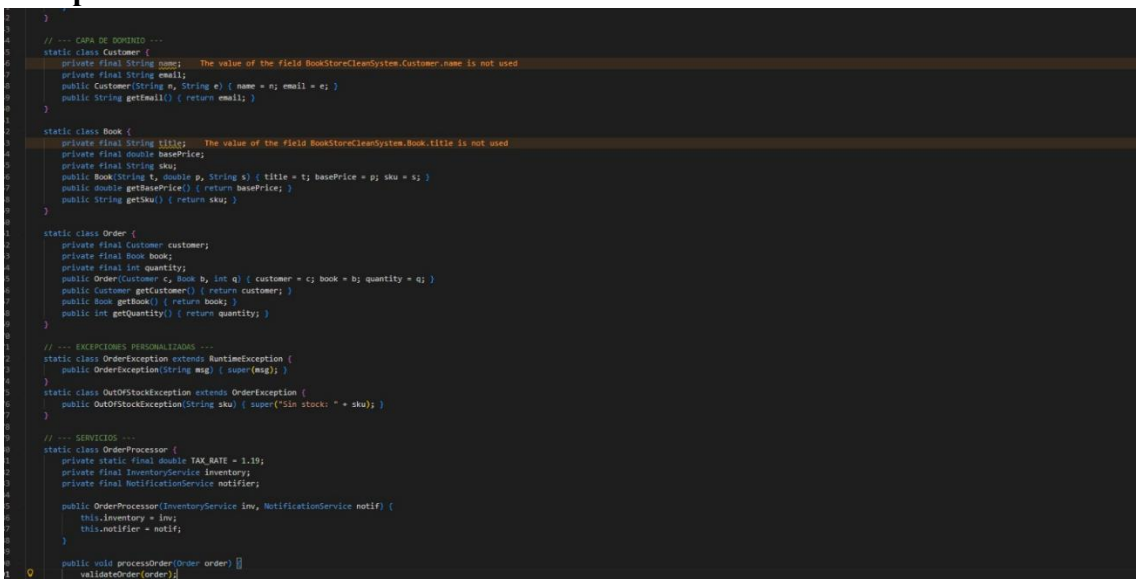


Ilustración 11 Todo en un solo archivo (dominio, servicios, etc.)

Aunque el código está organizado internamente, todas las clases se encuentran dentro de un único archivo. En un proyecto real, lo ideal sería separar cada clase en su propio archivo para mejorar la escalabilidad y organización.

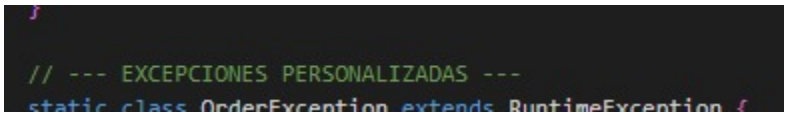


Ilustración 12 comentarios no efectivos

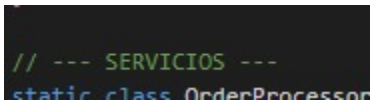


Ilustración 13 comentarios no efectivos

Se observa que algunos comentarios no aportan información relevante sobre la lógica del código. En Código Limpio, los comentarios deben complementar el código, no repetir lo que ya es evidente.

Conclusiones

El desarrollo de este taller permitió aplicar en un contexto práctico los principios del Código Limpio vistos en clase. A través del análisis del sistema "Librería Clean Code", se evidenció cómo la correcta asignación de responsabilidades, el uso de nombres descriptivos y la implementación de métodos pequeños contribuyen significativamente a la claridad del código.

Asimismo, el uso de excepciones personalizadas en lugar de códigos de error numéricos demuestra una forma más profesional y estructurada de manejar fallos dentro del sistema. La inyección de dependencias también mostró cómo se puede reducir el acoplamiento entre componentes, facilitando futuras modificaciones.

Sin embargo, también se identificaron aspectos que podrían mejorar, como la separación de clases en diferentes archivos para proyectos más grandes. Esto refuerza la idea de que el Código Limpio no solo implica escribir bien, sino también estructurar adecuadamente el software.

En conclusión, este ejercicio permitió comprender que la calidad del software no depende únicamente de su funcionamiento, sino también de la forma en que está construido, organizado y preparado para evolucionar con el tiempo.

Sede Quirinal: Calle 21 No. 6 - 01
Sede Prado Alto: Calle 8 No. 32 – 49 PBX: (608) 8754220
Sede Pitalito: Carrera 2 No. 1 – 27 - PBX: (608) 8360699
Email: contacto@corhuila.edu.co - www.corhuila.edu.co
Personería Jurídica Res. Ministerio de Educación No. 21000 de Diciembre 22 de 1989
NIT. 800.107.584-2