



CORPORACION UNIVERSITARIA DEL HUILA
CORHUILA

Facultad De Ingeniería - Ingeniería De Sistemas

Asignatura: Calidad De Software

Oscar Mauricio Areiza Paramo (9023125050)
Hermes Pascuas Herrera (9023125057)
Sergio Andrés Ordoñez Diaz (902312)

Febrero de 2026
Neiva – Huila

Contenido

Introducción.....	3
Taller 1 - Código Limpio.....	3
Funciones Limpias	4
Comentarios Efectivos	4
Formato y Estructura del Código.....	5
Manejo de Errores	5
SOLID.....	5
Open/Closed Principle	6
Principio de Sustitución de Liskov.....	6
Principio de Inversión de Dependencias.....	6
Conclusiones	7

Introducción

En la industria del software, el desarrollo de una aplicación no concluye con su compilación exitosa; por el contrario, ese es solo el inicio de su ciclo de vida. El verdadero desafío técnico radica en la mantenibilidad y la escalabilidad: la capacidad de un sistema para ser modificado y extendido sin comprometer su estabilidad original.

Este taller aborda la transición del "código funcional" al "código profesional" mediante la implementación de Buenas Prácticas de Programación y los Principios SOLID. Estos estándares no son reglas arbitrarias, sino soluciones probadas a problemas recurrentes en el diseño de sistemas. A través del análisis de un sistema de librería, exploraremos cómo la separación de responsabilidades, el bajo acoplamiento y la inyección de dependencias permiten construir arquitecturas robustas. El objetivo es internalizar que la calidad del código es una inversión directa en la agilidad del desarrollo y en la reducción de la deuda técnica a largo plazo.

Taller 1 - Código Limpio

Nombres Claros:

```
static class Customer {  
    ...  
}  
  
static class Book {  
    ...  
}  
  
static class Order {  
    ...  
}
```

- Evita Abreviaturas y Nombres Genéricos
- Usa Nombres Pronunciables
- Usa Nombres que Revelen la Intención

Funciones Limpias

```
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);  
}
```

- Las Funciones Deben Ser Pequeñas
- Una Responsabilidad
- Pocos Parámetros

Comentarios Efectivos

```
/*  
 * Sistema de librería - ejemplo de código limpio  
 */  
  
// La implementación de dependencias permanece fuera.  
  
// El código se mantiene en una única ubicación.
```

Uso correcto de los comentarios.

Auto-explicativo

Formato y Estructura del Código

Indentación Consistente

Longitud de Línea

Manejo de Errores

```
// CÓDIGO DE REFERENCIA
try {
    Customer customer = new Customer("Juan Pérez", "juan.perez@abc.com");
    Book book = new Book("Clase 100%", 50.00, "978-0987");
    Order order = new Order(customer, book, 2);

    processor.processOrder(order);
    System.out.println("El sistema procesó correctamente.");
} catch (OrderException e) {
    System.out.println("✗ ERROR: " + e.getMessage());
}

try {
    Customer c2 = new Customer("Marta Gómez", "marta.gomez@abc.com");
    Book noStockBook = new Book("Elma Agustín", 30.00, "978-0988");
    Order order2 = new Order(c2, noStockBook, 1);

    processor.processOrder(order2); // Lanza una excepción
} catch (OrderException e) {
    System.out.println("✗ CONTROLADO: El sistema detectó falta de stock.");
    System.out.println("Mensaje: " + e.getMessage());
}
```

Excepciones Personalizadas

Principios SOLID

Principio de Responsabilidad Única

```
// 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; }
}
```

Cada clase tiene una razón de ser única y clara, no hay una clase compleja que haga todo.

Open/Closed Principle

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

La clase OrderProcessor lanza OrderException.

Puedes crear nuevas excepciones como OutOfStockException que heredan de OrderException.

El bloque try-catch en el main está "cerrado" para modificación (maneja el tipo base OrderException), pero "abierto" para extenderse a errores más específicos sin cambiar la lógica de captura principal.

Principio de Sustitución de Liskov

Se aplica en la jerarquía de las excepciones:

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

En el main, cuando haces catch (OrderException e), el programa funciona perfectamente aunque lo que realmente se haya lanzado sea un OutOfStockException. El hijo puede sustituir al padre sin romper el comportamiento esperado del sistema de errores.

Principio de Segregación de Interfaces

No hay interfaces en el código... pero el principio se aplica de forma conceptual. Las clases son pequeñas y especializadas. OrderProcessor no obliga a InventoryService a tener métodos que no necesita (como métodos de facturación o envío). Cada servicio ofrece solo lo que es relevante para su dominio.

Principio de Inversión de Dependencias

```
// --- SERVICIOS ---
static class OrderProcessor {
    private static final double TAX_RATE = 1.19;
    private final InventoryService inventory;
    private final NotificationService notifier;
```

OrderProcessor **no crea** los servicios internamente (no hace new InventoryService()). En su lugar, los "recibe" desde fuera. Esto desacopla la lógica de negocio de la implementación técnica, facilitando pruebas unitarias (testing) y cambios futuros. Por lo tanto hay alta cohesión y bajo acoplamiento.

Conclusiones

- Modularidad y Cohesión: La aplicación del principio de Responsabilidad Única (SRP) demuestra que al segmentar la lógica de negocio en componentes especializados, se facilita el aislamiento de errores y se simplifican las pruebas unitarias.
- Gestión de Dependencias: Se ha evidenciado que la Inversión de Dependencias (DIP) es la piedra angular para lograr sistemas flexibles. Al delegar la creación de servicios a un nivel superior, el núcleo del negocio se vuelve independiente de las implementaciones técnicas externas.
- Extensibilidad Predictiva: El uso de jerarquías de excepciones y tipos de datos bien definidos permite que el sistema crezca de forma orgánica. Siguiendo el principio de Abierto/Cerrado (OCP), podemos integrar nuevas reglas de negocio con un impacto mínimo en el código preexistente.
- El Código como Activo de Comunicación: Un código limpio actúa como la documentación técnica más fiable. La claridad en la estructura y el manejo semántico de las clases reducen la curva de aprendizaje para nuevos desarrolladores y aseguran la continuidad del proyecto.

