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

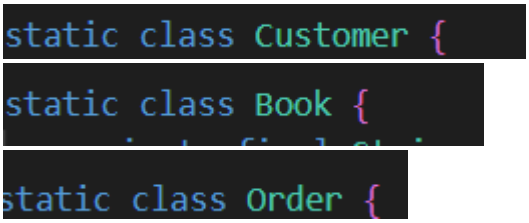
Introduccion

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:



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

Funciones Limpias

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

Comentarios Efectivos

✅ Uso correcto de los comentarios.

✅ Auto-explicativo

Formato y Estructura del Código

Indentación Consistente
Longitud de Línea

Manejo de Errores

✅ Excepciones Personalizadas

Principios SOLID

Principio de Responsabilidad Única

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.