Buenas Prácticas de Programación

Curso IF0004 – Desarrollo de Software II

M.Sc. José Pablo Noguera Espinoza

Universidad de Costa Rica
Sede del Sur

2 de diciembre de 2025

Agenda

Motivación

- Un programa puede **funcionar** y aun así estar mal escrito.
- Las malas prácticas generan:
  - Código difícil de leer y mantener.
  - Mayor cantidad de errores y fallos en producción.
  - Costos elevados de corrección y evolución.
- Las **buenas prácticas de programación** permiten:
  - Mejorar la calidad interna del software.
  - Facilitar el trabajo en equipo.
  - Aumentar la confiabilidad y la robustez de las aplicaciones.

Qué entendemos por "buenas prácticas"

Definición

Conjunto de principios, patrones, estándares y convenciones que guían la forma en que se escribe, organiza, documenta, prueba y mantiene el código fuente.

- No son reglas absolutas, pero sí **recomendaciones sólidas**.
- Surgen de:
  - Experiencia acumulada de la comunidad de desarrollo.
  - Estudios sobre mantenibilidad, calidad y productividad.
  - Normas y lineamientos institucionales o de la industria.

Principio de simplicidad

- **KISS (Keep It Simple, Stupid)**:
  - Preferir soluciones simples frente a diseños innecesariamente complejos.
  - Evitar agregar funcionalidades "por si acaso".
- Un diseño simple:
  - Es más fácil de entender para otras personas.
  - Reduce la probabilidad de errores ocultos.
  - Disminuye el esfuerzo de pruebas y mantenimiento.

Cohesión y responsabilidad única

Responsabilidad única (SRP)

Cada módulo, clase o método debe tener **una única responsabilidad clara**.

- Un método que:
  - Lee datos de consola,
  - Procesa la información,
  - Imprime reportes en pantalla y archivos,

  tiene demasiadas responsabilidades.
- Mejor:
  - Un método para leer datos.
  - Otro para procesar.
  - Otro para formatear/imprimir resultados.

Ejemplo: responsabilidad única

```java
// Ejemplo menos recomendable
public void procesarYReportarVentas() {
    // 1. Leer datos
    // 2. Calcular totales
    // 3. Imprimir reporte en consola
    // 4. Guardar reporte en archivo
}

// Ejemplo mejor estructurado
public void procesarVentas() { /* ... */ }

public void imprimirReporteConsola() { /* ... */ }

public void guardarReporteArchivo(String ruta) { /* ... */ }
```

Acoplamiento bajo

- **Acoplamiento**: grado de dependencia entre módulos o clases.
- Buen diseño:
  - Procura que las clases dependan lo menos posible de detalles concretos.
  - Prefiere depender de **interfaces** o abstracciones.
- Ventajas:
  - Permite cambiar una parte del sistema sin afectar el resto.
  - Facilita las pruebas unitarias y la reutilización.

Otros principios útiles

- **DRY (Don't Repeat Yourself)**:
  - Evitar duplicar lógica.
  - Extraer código repetido en métodos reutilizables.
- **YAGNI (You Aren't Gonna Need It)**:
  - No implementar funcionalidades adelantadas que todavía no se requieren.
- **Principios SOLID** (a nivel de POO):
  - Responsabilidad Única, Abierto/Cerrado, Sustitución de Liskov, Segregación de Interfaces, Inversión de Dependencias.

Documentación interna (comentarios)

- Los comentarios deben:
  - Explicar el **porqué** de las decisiones, no repetir el **qué**.
  - Ser breves, claros y actualizados.
- Evitar:
  - Comentarios obvios (// *suma a + b* sobre `a + b`).
  - Comentarios desactualizados que confunden más de lo que ayudan.

Ejemplo de comentarios internos

```
1 // MAL: comentario redundante
2 // suma a y b
3 int total = a + b;
4
5 // MEJOR: explica el contexto
6 // total representa el monto a pagar antes de impuestos
7 int total = a + b;
```

Documentación externa

- Manuales de usuario.
- Guías de instalación y despliegue.
- Diagramas y modelos del sistema (UML, casos de uso, etc.).
- Documentación de API (por ejemplo, usando JavaDoc).

Objetivo

Facilitar la comprensión del sistema a personas que no necesariamente conocen el código fuente (usuarios finales, equipo de soporte, nuevos integrantes del proyecto).

Nombres significativos

- Utilizar nombres que comuniquen intención:
  - `totalVentasMensuales` es mejor que `x1`.
- Reglas generales:
  - Variables y métodos: `camelCase`.
  - Clases e interfaces: `PascalCase`.
  - Constantes: `MAYUSCULAS_CON_GUIONES_BAJOS`.

Ejemplo de nombres

```
1 // Nombres poco claros
2 int a, b;
3 double c;
4
5 // Nombres descriptivos
6 int cantidadProductos;
7 int unidadesPorCaja;
8 double precioUnitario;
```

Estilo consistente

- Definir y seguir una **guía de estilo**:
  - Indentación uniforme (2 o 4 espacios).
  - Reglas para llaves, espacios y saltos de línea.
  - Organización de imports y paquetes.
- Consistencia:
  - Más importante que el estilo concreto.
  - Permite que el código parezca escrito por una sola persona.

Rol de las pruebas de software

- Verificar que el código realiza lo que se espera.
- Aumentar la confianza ante cambios y refactorizaciones.
- Detectar errores temprano, cuando son menos costosos de corregir.
- Servir como documentación ejecutable del comportamiento del sistema.

Tipos de pruebas

- **Pruebas unitarias**:
  - Verifican el comportamiento de métodos o clases aisladas.
- **Pruebas de integración**:
  - Evalúan cómo interactúan varios componentes entre sí.
- **Pruebas de sistema y aceptación**:
  - Evalúan el sistema completo desde la perspectiva del usuario.

Ejemplo sencillo de prueba unitaria (conceptual)

```java
// Ejemplo conceptual (usando estilo de frameworks comunes)
public void testCalcularTotalFactura() {
    Factura factura = new Factura();
    factura.agregarLinea("Cuaderno", 2, 500.0);
    factura.agregarLinea("L piz", 3, 200.0);

    double total = factura.calcularTotal();

    // Se espera: 2*500 + 3*200 = 1600
    assert total == 1600.0 : "Total incorrecto";
}
```

Resumen de buenas prácticas

- Aplicar principios de diseño:
  - Simplicidad, responsabilidad única, alta cohesión, bajo acoplamiento.
- Documentar adecuadamente:
  - Comentarios útiles y actualizados.
  - Documentación externa para usuarios y equipo técnico.
- Respetar convenciones del lenguaje:
  - Nombres significativos y estilo consistente.
- Incorporar pruebas de software:
  - Pruebas unitarias, de integración y de sistema como parte del proceso.

Referencias sugeridas

- Robert C. Martin (2008). *Clean Code: A Handbook of Agile Software Craftsmanship*. Prentice Hall.
- Steve McConnell (2004). *Code Complete (2nd Edition)*. Microsoft Press.
- Joshua Bloch (2018). *Effective Java (3rd Edition)*. Addison-Wesley.
- IEEE Std 829 y otros estándares de pruebas de software.
- Material de apoyo de la cátedra IF0004 – Desarrollo de Software II.

Gracias por su atención