Escuela Politécnica Nacional

Clase 006

Taller: Aplicando Principios de Código Limpio en Proyectos Reales

Curso: 2025-B SW Constr GR2

Integrantes: Jotcelyn Godoy, Cristian Robles
Canal: 002 – Definición T. Clase
Post: Clase 006

Objetivo del taller

Comprender los principios del código limpio y aplicarlos al análisis de repositorios públicos reales, identificando problemas de diseño, proponiendo mejoras y planteando refactorizaciones justificadas.

1. Introducción

¿Qué es el Código Limpio y por qué es importante?

El código limpio es aquel que se lee con facilidad, se entiende sin esfuerzo y se mantiene en el tiempo. No solo funciona, sino que comunica claramente su intención. Su importancia radica en que reduce errores, facilita el mantenimiento, mejora la colaboración en equipo y permite que los proyectos escalen sin volverse inmanejables.

Impacto del Código Limpio

- **Mantenibilidad:** cambios y correcciones más rápidas.
- **Escalabilidad:** el sistema crece sin perder claridad.
- **Trabajo en equipo:** cualquier desarrollador puede entender el código sin depender del autor original.

Código sucio vs. código limpio (idea general)

- Código sucio: funciones largas, nombres confusos, lógica mezclada, comentarios innecesarios.
- Código limpio: funciones pequeñas, nombres claros, responsabilidades bien definidas y estructura ordenada.

Principios clave trabajados

- Reglas generales de diseño.

- Comprensibilidad y uso de buenos nombres.
- Funciones y comentarios.
- Organización del código, objetos y estructuras.
- Pruebas y detección de olores de código.

2. Actividad Principal: Detectives de Código

Paso 1: Selección del repositorio

Cada grupo (2–3 estudiantes) selecciona un repositorio público en GitHub o GitLab. Puede elegirse cualquiera de los siguientes ejemplos o uno similar:

Lenguajes y repositorios sugeridos:

- **Python:** awesome-python, tensorflow
- **JavaScript:** react, javascript-algorithms, 30-seconds-of-code
- **Java:** developer-roadmap
- **C++:** recursos-programacion
- **Go:** recursos-programacion
- **Rust:** Rust_by_Example (GitLab)
- **PHP:** afip.php
- **Kotlin:** one-day-one-language
- **Swift:** recursos-programacion
- **SQL:** recursos-programacion
- **Multi-lenguaje:** freeCodeCamp, build-your-own-x, practice-and-examples

Paso 2: Selección de archivos

Elegir dos archivos del proyecto que contengan lógica relevante (evitar archivos solo de configuración).

Paso 3: Análisis del código

Analizar los archivos aplicando los principios de código limpio:

- ¿Los nombres de variables, funciones y clases son claros?
- ¿Las funciones son cortas y cumplen una sola responsabilidad?
- ¿Los comentarios aportan valor o son innecesarios?
- ¿Existen olores de código (duplicación, funciones extensas, variables ambiguas, lógica repetida)?
- ¿La estructura del código es ordenada y coherente?

Paso 4: Propuestas de mejora

Documentar al menos cinco (5) mejoras que se aplicarían al código, justificando cada una con base en los principios de código limpio.

3. Hallazgos del Taller (para publicar en el canal)

Ejemplo resuelto – Lenguaje Python

Repositorio analizado: javascript-algorithms (sección Python-equivalente / ejemplos educativos)
Lenguaje: Python
Archivos revisados:

- search.py (búsqueda lineal y binaria)
- sort.py (algoritmos de ordenamiento básicos)

Nota: El análisis se centra en la calidad del código y no en la eficiencia algorítmica.

Principales problemas identificados

1. **Nombres poco descriptivos:** variables como arr, l, r, tmp dificultan entender la intención del código.
2. **Funciones con múltiples responsabilidades:** algunas funciones validan datos, ejecutan el algoritmo y muestran resultados.
3. **Comentarios redundantes:** existen comentarios que repiten exactamente lo que el código ya expresa.
4. **Duplicación de lógica:** validaciones similares repetidas en varios métodos.
5. **Falta de pruebas unitarias:** no hay tests simples que validen los casos normales y de borde.

Propuestas de mejora (con justificación)

1. **Renombrar variables y funciones**
   Justificación: Usar nombres explícitos (numbers, left_index, right_index) mejora la legibilidad y reduce la carga cognitiva.
2. **Separar responsabilidades en funciones pequeñas**
   Justificación: Aplicar el principio de responsabilidad única facilita el mantenimiento y las pruebas.
3. **Eliminar comentarios innecesarios y reemplazarlos por código expresivo**
   Justificación: El código limpio se explica solo; los comentarios deben explicar el *por qué*, no el *qué*.

4. **Extraer validaciones comunes a funciones reutilizables**
   Justificación: Reduce duplicación y posibles inconsistencias futuras.
5. **Agregar pruebas unitarias básicas**
   Justificación: Aumenta la confianza en el código y permite refactorizar sin romper funcionalidad.

Ejemplo de refactorización (Python)

Antes (código poco limpio):

```
def bs(arr, x):
l = 0
r = len(arr) - 1
while l <= r:
m = (l + r) // 2
if arr[m] == x:
return m
elif arr[m] < x:
l = m + 1
else:
r = m - 1
return -1
```

Después (código limpio):

```
def binary_search(numbers: list[int], target: int) -> int:
    left_index = 0
    right_index = len(numbers) - 1

    while left_index <= right_index:
        middle_index = (left_index + right_index) // 2
        current_value = numbers[middle_index]

        if current_value == target:
            return middle_index

        if current_value < target:
            left_index = middle_index + 1
        else:
            right_index = middle_index - 1
```

```
return -1
```

Conclusión del grupo

- El análisis permitió comprobar que un código funcional no siempre es un código limpio. Aplicar principios como nombres claros, funciones pequeñas y eliminación de duplicación mejora significativamente la legibilidad y mantenibilidad del software. Trabajar con repositorios reales ayudó a desarrollar una mirada crítica sobre la calidad del código en proyectos profesionales.
- Este ejercicio demostró que el código limpio en Python no depende del lenguaje en sí, sino de las decisiones del desarrollador. Pequeños cambios generan un impacto grande en la calidad del proyecto.