Metodología de Sistemas II

Clase 7
Refactoring y Clean Code

Jeremías Fassi

2025

Refactoring y Clean Code

Buenas prácticas

Jeremías Fassi

2025

Objetivos de la clase

- Comprender qué es refactoring y su importancia en el ciclo de desarrollo de software.
- Identificar code smells y relacionarlos con problemas de mantenibilidad.
- Analizar la deuda técnica y cómo afecta a proyectos reales.
- Aplicar los principios de Clean Code y SOLID en el proyecto integrador.
- Practicar refactorización en pasos pequeños, con control mediante tests.

¿Qué es Refactoring?

Jeremías Fassi

¿Qué es Refactoring?

- El refactoring es el proceso de revisar y mejorar la estructura interna del código fuente sin alterar su **comportamiento observable**.
  Es decir, el programa sigue haciendo lo mismo desde el punto de vista del usuario final, pero su diseño interno se vuelve más claro y mantenible.

¿Qué es Refactoring?

Objetivos principales

- **Legibilidad**: cualquier programador puede comprender el código rápidamente.
- **Mantenibilidad**: facilitar correcciones de bugs y adición de nuevas características.
- **Extensibilidad**: permitir que el código se adapte mejor a cambios futuros.
- **Reducción de complejidad**: eliminar duplicaciones, simplificar estructuras, dividir responsabilidades.
- **Prevención de deuda técnica**: cada refactor reduce “intereses” acumulados en el código.

¿Qué es Refactoring?

→ Diferencia clave

El refactoring **no es añadir nuevas funcionalidades**, sino **mejorar la implementación existente**.

- Desarrollo de nuevas *features* = agregar comportamiento visible para el usuario.
- Refactoring = ordenar lo que ya existe. Esto es “ordenar o limpiar la habitación” sin comprar muebles nuevos.

¿Qué es Refactoring?

¿Cuándo aplicar refactoring?

- Cuando detectamos code smells (el código “huele raro”).
- Antes de agregar una nueva funcionalidad en un módulo que es difícil de leer.
- Después de escribir código rápido para entregar algo urgente.
- Durante revisiones de código en equipo (PRs).

Clean Code

Jeremías Fassi

2025

Clean Code (Código Limpio)

El objetivo principal de la refactorización es “saldar” la deuda técnica. Transforma un código desordenado, en código limpio y un diseño simple. Características:

- El código limpio es obvio para otros programadores: una mala nomenclatura de variables, clases y métodos sobrecargados, números mágicos, etc, hacen que el código sea descuidado y difícil de comprender.

- El código limpio no contiene duplicaciones: cada vez que se necesita hacer un cambio en un código duplicado, hay que recordar hacer el mismo cambio en cada instancia. Esto aumenta la carga cognitiva y ralentiza el progreso.

Clean Code (Código Limpio)

- El código limpio tiene un número mínimo de clases: menos código significa menos cosas que recordar, significa menos mantenimiento, y menos errores. **El código es su responsabilidad; manténgalo breve y simple**.

- Un código limpio supera todas las pruebas: un código está sucio cuando “solo” el 95% de sus pruebas son superadas. Un código está en problemas cuando la cobertura de sus pruebas es del 0 %.

¡Un código limpio es más fácil y económico de mantener!

Clean Code (Código Limpio)

Definición intuitiva:

- Código obvio para otros programadores. Esto es:
  - Sin duplicaciones.
  - Nombres significativos y consistentes.
  - Métodos cortos y con una sola responsabilidad.
  - Estilo uniforme (indentación, nombres, comentarios necesarios).
  - Que pasa todos los tests existentes.

Ejemplos

- Python
  - bad-code.py
  - clean-code.py

Deuda Técnica

Jeremías Fassi

2025

Definición

- La deuda técnica es una **metáfora**: describe el costo futuro de escribir código de forma rápida y desprolija hoy.

- Así como en una deuda financiera, uno recibe un beneficio inmediato (dinero disponible ahora), pero a cambio debe pagar intereses en el futuro. En el marco del desarrollo de software, se obtiene velocidad inicial, pero se acumula un “interés” en forma de dificultades de mantenimiento, bugs y costos crecientes de cambio.

Características

- Interés técnico: cuanto más tiempo se retrasa el refactoring, más caro se vuelve introducir cambios.
- Principal de la deuda: código escrito sin calidad (duplicaciones, acoplamientos fuertes, falta de pruebas).
- Pagos de interés: tiempo adicional invertido en cada modificación futura (entender código sucio, arreglar efectos colaterales).

Causas comunes

- Presión de negocio: priorizar entregas rápidas en lugar de calidad.
- Ausencia de pruebas automatizadas: sin tests, refactorizar es riesgoso y se acumula suciedad.
- Falta de documentación y estándares: cada desarrollador programa a su manera.
- Postergar refactoring: “después lo mejoramos”, pero nunca llega ese momento.
- Trabajo en ramas largas y aisladas: la integración se hace difícil y costosa.

Consecuencias

- Menor velocidad de desarrollo: cada cambio requiere mayor esfuerzo.
- Mayor riesgo de bugs: modificar una parte afecta otras inesperadamente.
- Equipo desmotivado: la complejidad excesiva desgasta y ralentiza.
- En casos extremos, parálisis del proyecto: llega un punto donde modificar cualquier cosa es tan costoso que el sistema colapsa.

Ejemplo

Imaginen un sistema de pedidos donde, por apuro, se decide “meter toda la lógica en un mismo método”:

- Funcional, pero difícil de entender.
- Cada nueva regla de negocio requiere editar el mismo bloque gigante.
- Los programadores tienen miedo de modificarlo porque “puede romper todo”.

→ **Resultado**: cada nueva *feature* tarda más en implementarse, generando los “intereses” de la deuda técnica.

Estrategias para gestionarla

- Pagar la deuda poco a poco: refactoring continuo, no dejar que se acumule.
- Adoptar prácticas de Clean Code y SOLID.
- Aplicar “Clean as You Code”: dejar cada módulo un poco más limpio de lo que lo encontraste.
- Tests automáticos: garantizar seguridad al refactorizar.
- Revisiones de código: detectar y frenar malas prácticas a tiempo.

Code Smells

El código “huele raro”

Jeremías Fassi

2025

Definición

Code smell es una señal de alerta en el código que indica un posible problema en su diseño o estructura. No significa necesariamente que haya un bug, sino que el código “huele mal”: esto es, código difícil de entender, mantener o extender.

Martin Fowler los define como “características superficiales del código que indican una debilidad más profunda en el sistema”.

Cuáles son los objetivos de detectarlos:

- Servir como síntomas tempranos de mala calidad interna.
- Guiar al programador hacia la necesidad de refactorizar.
- Evitar que la deuda técnica crezca sin control.

Clasificación

- *Bloaters* (Excesos): Código que crece demasiado.
  - Métodos largos.
  - Clases enormes.
  - Listas interminables de parámetros.
  - Obsesión por primitivos (usar `int` o `string` en lugar de objetos de dominio).

- Dispensables (Prescindibles): Elementos que podrían eliminarse.
  - Código duplicado.
  - Comentarios innecesarios (el código debería ser autoexplicativo).
  - Clases “perezosas” (hacen casi nada).

Clasificación

- Couplers (Acopladores): Código demasiado dependiente de otros módulos.
  - Dependencia excesiva en detalles de implementación.
  - "*Feature Envy*": métodos que usan más datos de otra clase que de la propia.
  - Clases con relaciones demasiado estrechas.

- *Change Preventers* (Bloqueadores de cambio): Código que hace difícil la evolución del sistema.
  - *Divergent Change*: una clase debe cambiar por demasiadas razones distintas.
  - *Shotgun Surgery*: un cambio pequeño requiere modificaciones en muchos archivos.

Clasificación

- Object-Orientation Abusers (Abusos de la orientación a objetos): Mal uso de herencia o de polimorfismo.
  - Uso abusivo de `switch` o `if-else` en vez de polimorfismo.
  - Jerarquías de clases rígidas.
  - Interfaces enormes que fuerzan a implementar métodos innecesarios.

Por qué son importantes

- Los *code smells* no rompen el programa, pero lo vuelven frágil y propenso a errores cuando evoluciona.
- Detectarlos y refactorizarlos a tiempo evita que la deuda técnica se convierta en un bloqueo.
- Son la alarma temprana que guía hacia mejoras.

Cómo refactorizar correctamente

Jeremías Fassi

2025

Principio fundamental

El refactoring no es un cambio caótico, sino un **proceso disciplinado**. Su meta no es “reescribir el código desde cero”, sino mejorar, paso a paso, el diseño interno con la seguridad de que el sistema sigue funcionando igual.

- Buenas prácticas

1. Pequeños pasos, cambios atómicos.
   a. Aplicar refactorings en unidades mínimas (extraer un método, renombrar una variable, mover una clase).
   b. Cada paso debe ser lo suficientemente pequeño como para revertirse fácilmente si algo falla.

Principio fundamental

2. Ejecutar los tests en cada paso.
    a. Tras cada refactoring, los tests deben seguir pasando.
    b. Si no hay tests, primero se deben escribir pruebas básicas antes de tocar el código.

3. No mezclar con nuevas funcionalidades.
    a. Un commit debe ser solo refactoring o solo feature, nunca ambos.
    b. Así se facilita la revisión de código y se evita confusión en caso de errores.

Principio fundamental

4. Mantener el comportamiento externo.
    a. Si el sistema se comporta distinto, ya no es refactoring: es un cambio funcional.
    b. Regla de oro: “refactor = mismo output, distinto camino interno”.

5. Detener el refactor cuando deje de aportar valor.
    a. El objetivo no es perfección absoluta, sino hacer el código suficientemente bueno para la próxima etapa de desarrollo.

Ciclo clásico de Refactoring

Red–Green–Refactor:

- Rojo (Red): escribir un test que inicialmente falla.
- Verde (Green): escribir el código mínimo para que el test pase.
- Refactor: mejorar la estructura interna del código sin romper los tests.

Este ciclo corto asegura que el refactor esté siempre bajo control.

Ejemplos

- Python
  - refactoring_01.py
  - refactoring_02.py
  - refactoring_03.py

Herramientas de soporte

- **IDE** (PyCharm, IntelliJ, VSCode): refactors automáticos como “rename”, “extract method”, “move class”.
- **Linters**: detectan code smells y sugieren mejoras.
- **SonarQube**: mide deuda técnica y duplicación.
- **Tests unitarios**: red de seguridad indispensable.

Estrategia en proyectos reales

- Refactorizar en paralelo al desarrollo (no esperar a que el código esté totalmente “podrido”).
- Aplicar la regla del *Boy Scout*: “Deja el código un poco más limpio de lo que lo encontraste”.
- Incluir refactorings como parte de las revisiones de PR.
- Documentar cada cambio: “refactor: extract method en OrderService para mejorar SRP”.

Principios SOLID

y su relación con el Refactoring

Jeremías Fassi

2025

Principios SOLID

- Los principios SOLID son guías de diseño orientadas a objetos que ayudan a crear sistemas más mantenibles, legibles y extensibles. **El refactoring suele tener como objetivo alinear el código con SOLID**.

    - S - Single-responsibility Principle
    - O - Open-closed Principle
    - L - Liskov Substitution Principle
    - I - Interface Segregation Principle
    - D - Dependency Inversion Principle

Principios SOLID

- S – Responsabilidad Única: refactorizar métodos grandes en métodos pequeños.
- O – Abierto/Cerrado: reemplazar condicionales extensos por polimorfismo.
- L – Sustitución de Liskov: asegurar que subclases respeten contratos.
- I – Segregación de Interfaces: dividir interfaces demasiado grandes.
- D – Inversión de Dependencias: usar abstracciones en lugar de clases concretas.

S – Single Responsibility Principle (SRP)

Definición: una clase/módulo debe tener una sola razón de cambio.

- Code smell asociado: métodos largos, clases gigantes.
- Refactoring: *Extract Method*, *Extract Class*.

Ejemplos Python:

- Antes
- Después

O – Open/Closed Principle (OCP)

Definición: el código debe estar abierto a extensión pero cerrado a modificación.

- Code smell: `switch` gigante, condicionales por tipo.
- Refactoring: *Replace Conditional with Polymorphism*, *Strategy*.

Ejemplos TS:

- Antes
- Después

L — Liskov Substitution Principle (LSP)

Definición: una subclase debe poder reemplazar a su superclase sin romper el programa.

- Code smell: subclases que lanzan errores en métodos heredados.
- Refactoring: rediseñar jerarquías, introducir interfaces más específicas.

Ejemplos Python:

- Antes
- Después

I – Interface Segregation Principle (ISP)

Definición: es mejor tener varias interfaces pequeñas y específicas que una grande y general.

- Code smell: interfaces gordas que fuerzan métodos innecesarios.
- Refactoring: *Extract Interface*.

Ejemplos Python:

- Antes
- Después

D – Dependency Inversion Principle (DIP)

Definición: los módulos de alto nivel no deben depender de implementaciones concretas, sino de abstracciones.

- Code smell: creación directa de dependencias con new.
- Refactoring: Introduce Interface, Dependency Injection.

Ejemplos Python:

- Antes
- Después

Recursos:

- Refactoring

- DigitalOcean Community – SOLID Design Principles Explained: Building Better Software Architecture

Muchas Gracias

Jeremías Fassi

Javier Kinter