SESIÓN 15
Técnicas de refactorización y uso en IDEs (IntelliJ IDEA / Netbeans/ Eclipse)

¿Qué vamos a ver?:

- Definir correctamente el concepto de **refactorización**
- Distinguir refactorización de otros procesos (optimización, corrección de errores…)
- Identificar **malos olores de código (code smells)** habituales
- Aplicar técnicas básicas de refactorización:
  - Rename
  - Extract Method
  - Extract Variable
  - Extract Class
- Utilizar las herramientas de refactorización de **IntelliJ IDEA, NetBeans o Eclipse**
- Valorar la importancia del código limpio y mantenible

INTRODUCCIÓN

Un programa no termina cuando funciona.
Termina cuando se puede mantener.

- ¿Cuántas veces habéis entendido un código el día que lo escribís… y no una semana después?
- ¿Qué es peor: un código que no funciona o uno que nadie entiende?

Conclusión:

- El código **evoluciona**
- Los proyectos **crecen**
- El código mal escrito **se convierte en un problema**

¿QUÉ ES LA REFACTORIZACIÓN?

La refactorización es el proceso de **mejorar la estructura interna del código fuente**

sin modificar su **comportamiento externo**,
con el objetivo de mejorar su **calidad, legibilidad y mantenibilidad**.

¿Por qué se refactoriza?

- Para que el código:
  - sea más fácil de leer
  - sea más fácil de modificar
  - tenga menos errores en el futuro
  - sea reutilizable
- Para reducir el **coste de mantenimiento**
- Para adaptarse a cambios sin reescribir todo

Qué NO es refactorizar

Acción	¿Refactorización?	Motivo
Cambiar nombres	Si	Mejora comprensión
Dividir métodos largos	Si	Mejora diseño
Eliminar código duplicado	Si	Mejora mantenibilidad
Corregir un bug	No	Cambia comportamiento
Añadir funcionalidad	No	Cambia el sistema

Refactorizar mejora el código, no el resultado.

CODE SMELLS: SEÑALES DE CÓDIGO DE MALA CALIDAD

Un **code smell** es una **señal de alerta** en el código que indica que algo puede estar mal diseñado, aunque el programa funcione correctamente.
No son errores, son **síntomas**.

SMELL 1: NOMBRES POCO DESCRIPTIVOS

```
int x;
int y;
int z = x * y;
```

Problema:

- No sabemos qué representa cada variable
- Dificulta el mantenimiento

Refactorizado (Rename)

```
int ancho;
int alto;
int area = ancho * alto;
```

Los nombres deben explicar la intención del código.

SMELL 2: MÉTODOS DEMASIADO LARGOS

```
void procesarOrden() {
   // validar usuario
   // comprobar stock
   // calcular precio
   // aplicar descuento
   // guardar pedido
   // enviar email
}
```

Problema:

- Demasiadas responsabilidades
- Difícil de modificar
- Difícil de probar

Refactorizado (Extract Method)

```
void procesarOrden() {
   validarUsuario();
   comprobarStock();
   double precio = calcularPrecio();
```

```
    guardarOrden(precio);
    EnviarConfirmacionEmail();
}
```

- Principio de responsabilidad única (SRP)
- Métodos pequeños y claros

SMELL 3: CÓDIGO DUPLICADO

```
double precioFinal = precio * 1.21;
// ...
double totalConIVA = total * 1.21;
```

Problema:

- Si cambia el IVA, hay que cambiarlo en varios sitios

Refactorizado

```
double aplicarIVA(double cantidad) {
    return cantidad * 1.21;
}
```

Teoría asociada:

- DRY (Don't Repeat Yourself)
- Centralizar la lógica

TÉCNICAS DE REFACTORIZACIÓN

1. RENAME

Cambiar el nombre de variables, métodos o clases para que reflejen claramente su función.

Ejemplo:

```
void f() {}
```

Refactorizado:

```
void calcularPrecioFinal () {}
```

2. EXTRACT METHOD

Extraer fragmentos de código en métodos independientes para reducir complejidad.

Ejemplo:

```
if(age > 18 && hasPermission && !isBanned) {
   // acceso permitido
}
```

Refactorizado:

```
if(canAccess()) {
   // acceso permitido
}

boolean canAccess() {
   return age > 18 && hasPermission && !isBanned;
}
```

3. EXTRACT VARIABLE

Guardar expresiones complejas en variables con nombre descriptivo.

Ejemplo:

```
double total = precio * cantidad * 1.21;
```

Refactorizado:

```
double iva = 1.21;
double total = precio * cantidad * iva;
```

4. EXTRACT CLASS

Separar responsabilidades cuando una clase hace demasiado.

Ejemplo:

```
class Usuario {
   // datos
   // validaciones
   // acceso a base de datos
}
```

Refactorizado:

```
class Usuario {}
class UsuarioValidar {}
class UsuarioAcceso {}
```

USO DE IDEs PARA REFACTORIZAR

Refactorizar manualmente es arriesgado.
Los IDEs garantizan seguridad y coherencia.

IntelliJ IDEA
Atajo principal:
Ctrl + Alt + Shift + T

Permite:

- Rename
- Extract Method
- Extract Variable
- Inline
- Change Signature

Ventajas:

- Actualiza todas las referencias
- Evita errores
- Refactorización segura

Eclipse

Atajos más usados:

- Alt + Shift + R → Rename
- Alt + Shift + M → Extract Method
- Alt + Shift + L → Extract Variable

BUENAS PRÁCTICAS AL REFACTORIZAR

- Refactorizar **cuando el código funciona**
- Hacer cambios pequeños y progresivos
- Usar siempre el IDE
- No mezclar refactorización con nuevas funciones
- Refactorizar pensando en el futuro

Un buen programador hace que funcione. Un programador profesional hace que se pueda mantener.