# UT4 – OPTIMIZACIÓN Y DOCUMENTACIÓN – PARTE III- REFACTORIZACIÓN

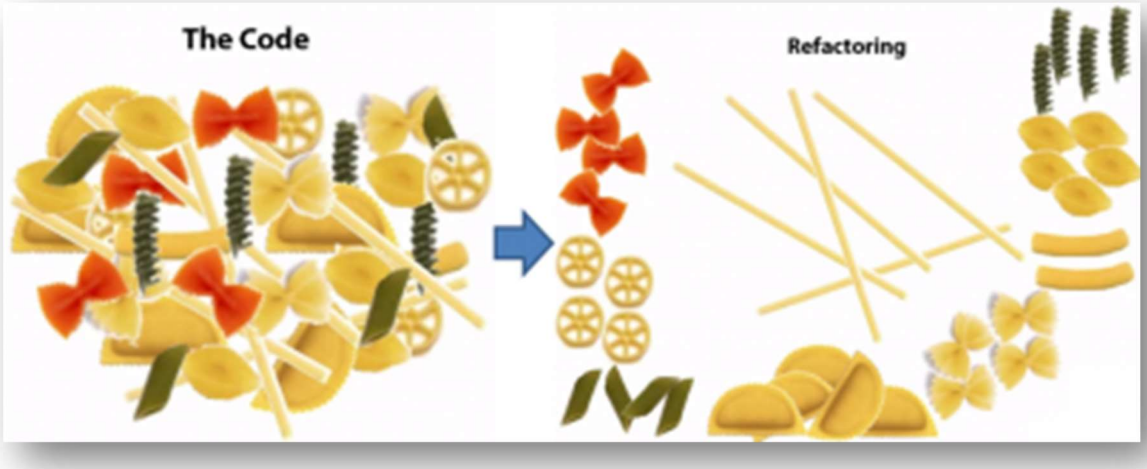
| **RESULTADOS DE APRENDIZAJE ASOCIADOS** |
|---|
| 4.- Optimiza código empleando las herramientas disponibles en el entorno de desarrollo. |
| **CRITERIOS DE EVALUACIÓN** |
| a) Se han identificado los patrones de refactorización más usuales. |
| b) Se han elaborado las pruebas asociadas a la refactorización. |
| c) Se ha revisado el código fuente usando un analizador de código. |
| d) Se han identificado las posibilidades de configuración de un analizador de código. |
| e) Se han aplicado patrones de refactorización con las herramientas que proporciona el entorno de desarrollo. |
| f) Se ha realizado el control de versiones integrado en el entorno de desarrollo. |
| g) Se han utilizado herramientas del entorno de desarrollo para documentar las clases. |

# UT4 – OPTIMIZACIÓN Y DOCUMENTACIÓN – PARTE III - REFACTORIZACIÓN

## Índice de contenido

# 1.- Refactorización

La refactorización constituye una técnica intermedia en la ingeniería del software, facilitando la mejora del código existente sin alterar su funcionalidad externa. En otras palabras, **el proceso de refactorización implica reescribir el comportamiento actual del código de una manera que resulte más comprensible y manejable**, favoreciendo la legibilidad para el desarrollador a cargo, garantizando que, tras la refactorización, el proyecto funcione igualmente bien y ofrezca idénticos resultados.




¿Qué contribuciones aporta la refactorización?

- Depura el código, optimizando su coherencia y transparencia.
- Preserva el código, sin rectificar fallos ni incorporar funciones adicionales.
- Facilita la implementación de modificaciones en el código.
- Se consigue un código nítido y con una alta modularidad. Refactorización previa y posterior.

## 1.1.- Cuándo refactorizar. Malos olores (bad smells)

El proceso de refactorización debería implementarse progresivamente durante el avance del desarrollo de la aplicación. Piattini y García (2003) estudian las señales que sugieren la necesidad de refactorizar, a las que Martin Fowler y otros expertos (1999) denominaron inconvenientes palpables (bad smells). Dichas señales incluyen:

- **Código replicado** (*Duplicated code*). Representa el motivo principal para proceder con la refactorización. Cuando se identifica el mismo código en distintos puntos, es esencial buscar una manera de consolidarlo y estandarizarlo.

Ejemplo:

En programación, se conoce como código redundante a cualquier parte del código fuente que tenga algún tipo de redundancia tales como recalcular un valor que ha sido calculado previamente y todavía está disponible.

```
int suma (int numero1, int numero2) {
   int resultado = numero1 + numero2;    // Esta línea puede eliminarse
   return numero1 + numero2;
}
```

- **Métodos excesivamente extensos** (*Long method*). Cuanto mayor es la longitud de un método, más compleja es su comprensión. Un método demasiado prolongado suele encargarse de tareas que deberían delegarse a otros. Es necesario discernir estas funciones y fragmentar el método en unidades más reducidas. En la programación orientada a objetos, la brevedad de un método facilita su reutilización.

## Código sin refactorizar

```
public class CalculadoraSalario {

    public void calcularSalario(int horasTrabajadas, double tarifaPorHora, boolean
esFestivo) {
        double salarioBase = horasTrabajadas * tarifaPorHora;
        double bonificacion = 0.0;

        if (horasTrabajadas > 40) {
            // Bonificación por horas extras
            int horasExtras = horasTrabajadas - 40;
            bonificacion += horasExtras * tarifaPorHora * 0.5;
        }

        if (esFestivo) {
            // Bonificación del 50% del salario base por trabajar en un día festivo
            bonificacion += salarioBase * 0.5;
        }

        double salarioTotal = salarioBase + bonificacion;

        System.out.println("El salario total es: " + salarioTotal);
    }
}
```

## Código refactorizado

```
public class CalculadoraSalario {

    public void calcularSalario(int horasTrabajadas, double tarifaPorHora, boolean
esFestivo) {
        double salarioBase = calcularSalarioBase(horasTrabajadas, tarifaPorHora);
        double bonificacion = calcularBonificaciones(horasTrabajadas, tarifaPorHora,
esFestivo);
        double salarioTotal = salarioBase + bonificacion;

        imprimirSalario(salarioTotal);
    }

    private double calcularSalarioBase(int horasTrabajadas, double tarifaPorHora) {
        return horasTrabajadas * tarifaPorHora;
```

```
    }

    private double calcularBonificaciones(int horasTrabajadas, double tarifaPorHora,
boolean esFestivo) {
        double bonificacion = 0.0;

        bonificación+=calcularBonificacionPorHorasExtras (horasTrabajadas,
tarifaPorHora);
        bonificacion += calcularBonificacionPorFestivo(horasTrabajadas, tarifaPorHora,
esFestivo);

        return bonificacion;
    }

    private double calcularBonificacionPorHorasExtras(int horasTrabajadas, double
tarifaPorHora) {
        if (horasTrabajadas > 40) {
            int horasExtras = horasTrabajadas - 40;
            return horasExtras * tarifaPorHora * 0.5;
        }
        return 0.0;
    }

    private double calcularBonificacionPorFestivo(int horasTrabajadas, double
tarifaPorHora, boolean esFestivo) {
        if (esFestivo) {
            double salarioBase = calcularSalarioBase(horasTrabajadas, tarifaPorHora);
            return salarioBase * 0.5;
        }
        return 0.0;
    }

    private void imprimirSalario(double salarioTotal) {
        System.out.println("El salario total es: " + salarioTotal);
    }
}
```

**Tienes el código fuente (en JAVA) disponible en el aula para que puedas verlo mejor.**

Para refactorizar este método, podemos **dividirlo en varios métodos más pequeños**, **cada uno con una única responsabilidad**. **Esto mejora la legibilidad y facilita el mantenimiento y la prueba del código**.

- **Clases muy grandes** (*Large classes*): Si una clase intenta resolver muchos problemas, tendremos una clase con demasiados métodos y atributos, y hacer instancias. La clase está asumiendo demasiadas responsabilidades, hay que intentar fraccionarlas y delegarlas, de forma que cada una trate con su conjunto de responsabilidades más pequeñas.

**Código sin refactorizar**

```
// Código original

public class Estudiante {

   private String nombre;
   private int edad;
   private String direccion;

   // métodos para acceder y modificar atributos
}
```

**Código refactorizado**

```
// Refactorizado
public class Estudiante {
   private DatosPersonales datosPersonales;

   // métodos para acceder y modificar los datos personales
}

public class DatosPersonales {
```

```
    private String nombre;
    private int edad;
    private String direccion;

    // métodos para acceder y modificar los atributos
}
```

**Tienes el código fuente (en JAVA) disponible en el aula para que puedas verlo mejor.**

- **Lista de parámetros larga** (*Long Parameter List*): En la programación orientada a objetos no se suelen pasar muchos parámetros a los métodos, sino solo aquellos mínimamente necesarios para que el objeto invoque los métodos correspondientes. Tener demasiados parámetros puede estar indicando un problema de diseño. Es necesario, en vez de tener argumentos con datos, tener objetos de encapsulación de datos o la necesidad de crear una clase de objetos a partir de varios parámetros y pasar ese objeto como argumento en vez de todos los parámetros. Especialmente si esos parámetros suelen tener que ver unos con otros y suelen ir juntos siempre.

**Tienes el código fuente (en JAVA) disponible en el aula para que puedas verlo mejor.**

- **Cambio divergente** (*Divergent change*): una clase es frecuentemente modificada por diversos motivos, los cuales no suelen estar relacionados entre sí, a lo mejor conviene eliminar la clase. Este síntoma es el opuesto del siguiente.

- **Cirugía a tiro pistola** (*Shotgun surgery*): este síntoma se presenta cuando después de un cambio en una determinada clase, se deben realizar varias modificaciones adicionales en diversos lugares para compatibilizar dicho cambio.

- **Clase de solo datos** *(Data class*): Clases que solo tienen atributos y métodos de acceso a ellos ("get" y "set"). Este tipo de clases deberían cuestionarse dado que no suelen tener comportamiento alguno.

- **Legado rechazado** (*Refused bequest*): este síntoma lo encontramos en subclases que utilizan solo unas pocas características de sus superclases. Si las subclases no necesitan o no requieren todo lo que sus superclases les proveen por herencia, esto puede indicar como que pensar la jerarquía de clases no es correcta. La delegación suele ser la solución a este tipo de inconvenientes.

- **Renombrar variables y hacerlas descriptivas.**

Ejemplo:

**Código sin refactorizar:**

```
// Código original
public double calcular(double[] nums) {
   double sum = 0;
   for (int i = 0; i < nums.length; i++) {
      sum += nums[i];
   }
   return sum / nums.length;
}
```

**Código refactorizado**

```
// Refactorizado
public double calcularPromedio(double[] numeros) {
   double sumaTotal = 0;
   for (int indice = 0; indice < numeros.length; indice++) {
      sumaTotal += numeros[indice];
   }
   return sumaTotal / numeros.length;
}
```

1. **Renombramiento de Métodos y Variables:**
   - El método calcular ha sido renombrado a ***calcularPromedio***, lo que proporciona una descripción más precisa de lo que hace el método. Esto es fundamental para la claridad y el mantenimiento del código, ya que el nombre calcular es muy genérico y podría referirse a cualquier tipo de cálculo.
   - Las variables también se han renombrado para ser más descriptivas. ***nums*** se cambió a ***numeros***, y ***sum*** a ***sumaTotal***, haciendo que el código sea más fácil de leer para alguien que habla español o para cualquiera tratando de comprender el propósito de estas variables sin necesidad de deducirlo por el contexto.

2. **Claridad en el Propósito:**
   - Al renombrar el método a **calcularPromedio**, se elimina cualquier ambigüedad sobre qué tipo de cálculo se está realizando. Esto es especialmente importante en bases de código más grandes donde los métodos pueden realizar una amplia variedad de operaciones.

3. **Mantenibilidad:**

   - Con nombres de variables y métodos más descriptivos, el código es más fácil de entender y, por lo tanto, de mantener. Otros desarrolladores (o el mismo desarrollador en el futuro) pueden entender rápidamente qué hace este método y cómo lo hace.

4. **Estilo de Código:**

   - Aunque la lógica central del método no cambió (ambos fragmentos de código hacen esencialmente lo mismo), el código refactorizado sigue un estilo de codificación que favorece la claridad. Usar nombres de variables y métodos descriptivos es una práctica recomendada en programación porque reduce la necesidad de comentarios adicionales para explicar qué hace el código.

- **Eliminar código muerto:** En programación, se conoce como código muerto a una parte del código fuente que se ejecuta, pero sus resultados nunca se usan. La ejecución de este tipo de código consume tiempo de cómputo en algo que jamás se utiliza.

Ejemplo:

```
int suma (int numero1, int numero2) {
   int resultado = numero1 * numero2;      // Esta línea puede eliminarse
   return numero1 + numero2;
}
```

- **Eliminar almacenamiento muerto:** En programación se conoce como almacenamiento muerto o dead store a la acción de asignarle un valor cualquiera a una variable local y no utilizarlo en ninguna instrucción subsecuente. Este tipo de error de software es indeseable debido a que requiere tiempo de computación y accesos a memorias de forma innecesaria, lo que impacta en el rendimiento.

```
int suma (int numero1, int numero2) {
   int resultado = numero1 + numero2;      // Esta línea puede eliminarse
   return numero1 + numero2;
}
```

- **Eliminar código inalcanzable:** En programación, el código inalcanzable es una parte del código fuente que nunca podrá ser ejecutado porque no existe ningún camino dentro de las estructuras de control en el resto del programa para llegar a este código.

```
int multiplica(int numero1, int numero2) {
   return numero1 * numero2;
   int resultado = numero1 / numero2;     // Esta línea puede eliminarse
}
```

- **Reemplazar número mágico con constante simbólica.**

**Ejemplo1:**

```
// Código original
public double calcularDescuento(double precio) {
   return precio * 0.1;  // 0.1 es un número mágico que representa el 10%
de descuento
}
```

```
// Refactorizado
private static final double DESCUENTO_PORCENTUAL = 0.1;

public double calcularDescuento(double precio) {
```

```
    return precio * DESCUENTO_PORCENTUAL;
}
```

**Ejemplo 2:**

```
// Código original
public double CalcularAreaCirculo(double radio) {
    return 3.14 * radio * radio;
}
```

```
// Refactorizado
public double calcularAreaCirculo(double radio) {
    return Math.PI * radio * radio;
}
```

## 1.2.- Resumen de refactorización

La refactorización es una técnica clave en el desarrollo de software para mejorar la legibilidad, mantenibilidad y calidad del código sin alterar su comportamiento externo. Aquí tienes un resumen de varias técnicas de refactorización discutidas, basadas en los principios de Piattini y García (2003), y popularizadas por Martin Fowler y otros expertos:

1. **Código Replicado:** Identifica y elimina duplicaciones en el código, consolidándolo para mejorar la coherencia y reducir errores.
2. **Métodos Excesivamente Largos**: Descompone métodos largos en unidades más pequeñas y manejables, facilitando su comprensión, mantenimiento y reutilización.
3. **Clases Muy Grandes:** Divide clases grandes en clases más pequeñas, cada una con su conjunto específico de responsabilidades, mejorando la cohesión.

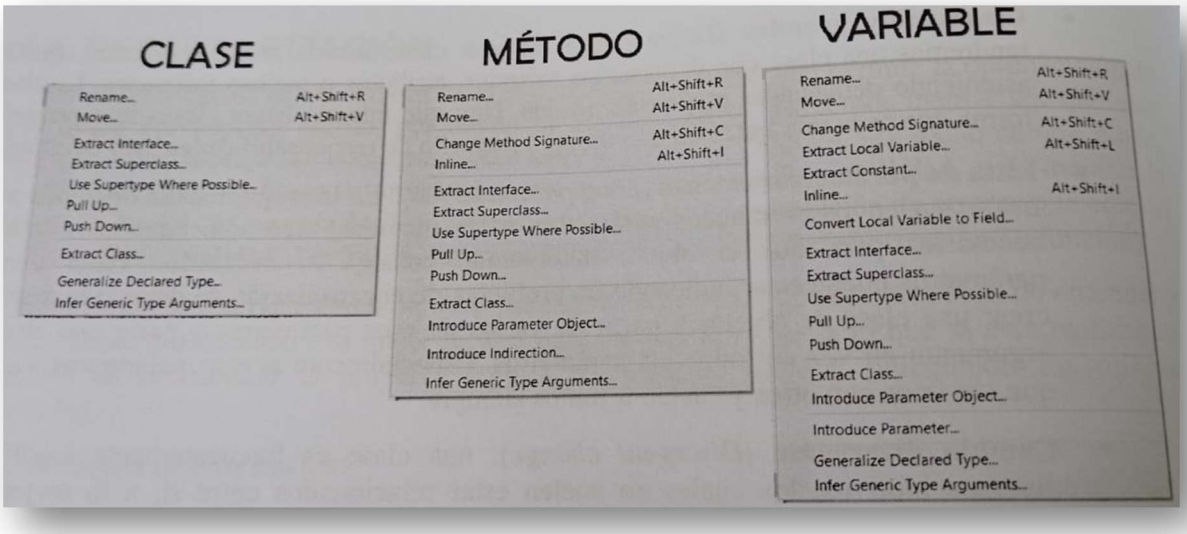
4. **Lista de Parámetros Larga:** Reduce el número de parámetros de un método agrupándolos en objetos si están estrechamente relacionados, o utilizando patrones de diseño que simplifiquen la interacción.
5. **Cambio Divergente y Cirugía a Tiro Pistola:** Identifica y soluciona problemas de diseño donde cambios en una clase requieren múltiples modificaciones en otros lugares (cirugía a tiro pistola), o una clase es modificada por muchas razones diferentes (cambio divergente).
6. **Clase de Solo Datos:** Añade comportamiento a clases que solo contienen datos para que cumplan con principios de diseño orientado a objetos, promoviendo el encapsulamiento de datos.
7. **Legado Rechazado:** Reorganiza la herencia o utiliza composición sobre herencia cuando subclases no utilizan o necesitan todas las propiedades o métodos heredados.
8. **Renombrar Variables y Métodos Descriptivos:** Utiliza nombres significativos y descriptivos para variables y métodos, mejorando la legibilidad y comprensión del código.
9. **Eliminar Código Muerto y Almacenamiento Muerto:** Elimina partes del código que no se utilizan o variables a las cuales se les asignan valores que nunca se usan, optimizando la ejecución y claridad.
10. **Eliminar Código Inalcanzable:** Remueve secciones del código que nunca podrán ser ejecutadas debido a la lógica de control de flujo del programa.
11. **Reemplazar Números Mágicos con Constantes Simbólicas:** Sustituye valores numéricos directos en el código por constantes nombradas, facilitando la comprensión y mantenimiento del código.

Cada una de estas técnicas aborda diferentes "malos olores" en el código, apuntando a mejorar la calidad general del software y hacerlo más sencillo de entender, extender y mantener.

## 2.- Refactorización en Eclipse

Eclipse tiene diversos métodos de refactorizar o ***refactorin***g. Dependiendo de dónde invoquemos a la refactorización tendremos un menú contextual u otro con sus distintas opciones de refactorización. También hay una

opción de menú que es ***Refactor***. Para refactorizar elegiremos la opción ***Refactor*** del menú contextual.




Los ***patrones de refactorización*** son las prácticas para refactorizar el código, utilizando las herramientas podremos plantear cambios para refactorizar y se mostrarán las posibles soluciones de las que podremos ver el antes y el después de la refactorización. También se llama ***catálogos de refactorización***.

Para refactorizar se selecciona el elemento (puede ser una clase, una variable, una expresión, un paquete, un método, un módulo, etc.…), se pulsa el botón derecho del ratón, se selecciona ***Refactor***, y seguidamente se selecciona la opción de refactorización. A continuación, se muestran algunos de los patrones más comunes:

- **Rename**. Es una de las opciones más utilizadas. Cambia el nombre de variables, clases, métodos, paquetes, directorios y casi cualquier identificador Java. Tras la refactorización, se modifican las referencias a ese identificador.

- **Move**. Mueve una clase de un paquete a otro, se mueve el archivo .java a la carpeta y se cambian todas las referencias. También se puede arrastrar y soltar una clase a un nuevo paquete, se realiza una refactorización automática.

- **Extract Constant**. Convertir un número o cadena literal en una constante. Al hacer la refactorización se mostrará donde se van a producir los cambios, y se pide visualizar el antes de la refactorización y después de refactorizar. Tras la refactorización, todos los usos del literal se sustituyen por esa constante. El objetivo es modificar el valor del literal en un único lugar.

- **Extract Local Variable**. Asignar una expresión a variable local. Tras la refactorización, cualquier referencia a la expresión en el ámbito local se sustituye por la variable. La misma expresión en otro método no se modifica.

## 2.1.- Otras operaciones de refactorización

Eclipse permite visualizar el ***histórico de refactorizaciones*** realizado sobre un proyecto. Para ver el histórico, se abre el menú ***Refactor/History.***

En la ventana que se muestra se pueden observar todos los cambios realizados y el detalle de dichos cambios.

Eclipse también permite crear un *Script* con todos los cambios realizados en la refactorización y guardarlo en un fichero XML (menú ***Refactor/Create Script***)

# 3.- Análisis de código estático

Refactorizar código estático con ***PMD*** implica analizar y mejorar el código fuente para hacerlo más eficiente y mantenible, sin cambiar su comportamiento externo.

***PMD*** es una herramienta de análisis de código estático que ayuda a identificar patrones de código problemáticos que podrían ser mejorados.

Aquí tienes algunos apuntes sobre cómo realizar esta tarea:

## 3.1.- ¿Qué es PMD?

PMD es una herramienta de análisis de código fuente que detecta patrones de código ineficientes, susceptibles a errores o que no siguen ciertas convenciones de codificación.

Es ampliamente usado en varios lenguajes de programación, como *Java*, *JavaScript*, *PLSQL*, *XML* y más.

Ofrece un conjunto de reglas predefinidas que se pueden personalizar según las necesidades del proyecto.

## 3.2.- Beneficios de usar PMD para refactorizar código

- **Mejora de la Calidad del Código:** Ayuda a mantener el código limpio y de alta calidad al identificar problemas comunes.
- **Prevención de Bugs:** Detecta patrones de código que son propensos a errores.
- **Optimización:** Encuentra métodos o bloques de código que podrían ser optimizados para mejorar el rendimiento.
- **Consistencia:** Fomenta la consistencia en las prácticas de codificación dentro de un equipo.

## 3.3.- Pasos para refactorizar con PMD

1. **Instalación y Configuración:**
   - Instalar PMD desde su sitio oficial o gestionar su dependencia a través de herramientas de construcción de proyectos como Maven o Gradle.
   - Configurar las reglas de PMD según las necesidades del proyecto, habilitando o deshabilitando ciertas reglas o creando reglas personalizadas.

2. **Ejecución del Análisis:**
   - Ejecutar PMD a través de la línea de comandos, plugins de IDEs (como Eclipse, IntelliJ IDEA), o integraciones de CI/CD.
   - Analizar los resultados para identificar los problemas detectados por PMD.

3. **Refactorización del Código:**
    - Priorizar las advertencias basadas en su importancia y el impacto en el proyecto.
    - Refactorizar el código según las recomendaciones de PMD, mejorando la legibilidad, eficiencia, y mantenibilidad sin alterar su funcionalidad.

4. **Verificación y Pruebas:**
    - Verificar que los cambios no introduzcan nuevos errores o problemas de rendimiento.
    - Ejecutar pruebas unitarias y de integración para asegurar que la refactorización no haya alterado el comportamiento esperado del software.

5. **Iteración:**
    - Repetir el proceso regularmente como parte del ciclo de desarrollo para mantener la calidad del código a lo largo del tiempo.

## 3.4.- Consejos para una Refactorización Efectiva con PMD

- **Comprender las Reglas:** Antes de empezar, es crucial entender qué busca cada regla de PMD y cómo se aplica al código.
- **Refactorización Incremental:** Es más manejable hacer cambios pequeños y progresivos que grandes modificaciones de una sola vez.
- **Pruebas Continuas:** Mantener un enfoque riguroso en las pruebas para asegurar que la refactorización no rompa ninguna funcionalidad existente.
- **Integración Continua:** Integrar PMD en el proceso de integración continua para detectar y corregir problemas de forma temprana en el ciclo de desarrollo.

# 4.- Referencias bibliográficas


- Moreno Pérez, J.C. *Entornos de desarrollo.* Editorial Síntesis.
- Ramos Martín, A. & Ramos Martín, M.J. *Entornos de desarrollo*. Grupo editorial Garceta.