

## PARTE 1: REFACTORIZACIÓN

1. INTRODUCCIÓN.....	1
2. ANALIZADORES DE CÓDIGO.....	1
3. REFACTORIZACIÓN.....	5

### 1. INTRODUCCIÓN

#### REFACTORIZACIÓN

Permite **optimizar** el código , realizando cambios en la estructura interna, sin afectar a su comportamiento.

Previamente se requiere **analizar el código**.

El **objetivo**: código sea más **fácil de entender y de mantener**.

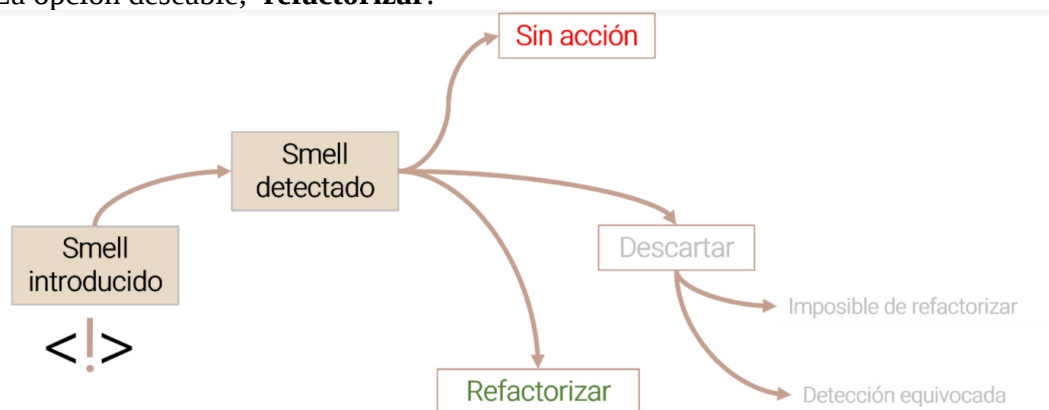
### 2. ANALIZADORES DE CÓDIGO

Es el **paso previo a la refactorización**.

Consiste en aplicar técnicas que **facilitan la escritura y lectura** de un código, facilitando su **entendimiento y su mantenimiento**.

#### CODE SMELL(‘código que huele’)

- Es cualquier característica en el código que **posiblemente** indique un problema.
- Usualmente **no son errores de programación** ya que no impiden que nuestra aplicación funcione correctamente.
- **Indican deficiencias en el diseño** software que pueden ralentizar el desarrollo o aumentar el riesgo de errores o fallos en el futuro.
- Cuando detectamos **code smell** podemos:
  - No hacer nada
  - Descartar el cambio porque es inviable o porque nos hemos equivocado al detectarlo
  - La opción deseable; **refactorizar**.



## **CODE SMELLS A NIVEL DE APLICACIÓN**

- **Nombre misterioso:** Clases, variables o funciones que se nombran de una manera que no comunica lo que hacen o cómo deben usarse.
- **Código duplicado:** Código idéntico o muy similar que aparece en más de una función en nuestro programa. Seguramente se pueda parametrizar y codificar en una única función.
- **Complejidad artificial:** Uso forzado de patrones de diseño demasiado complicados donde bastarían patrones de diseño más simples.
- **Cirugía de escopeta:** Un único cambio que debe aplicarse a varias clases al mismo tiempo.
- **Efectos secundarios incontrolados:** Código que lanza excepciones en tiempo de ejecución, y las pruebas unitarias no pueden capturar la causa exacta del problema.

## **CODE SMELLS A NIVEL DE CLASE:**

- **Clase grande:** Una clase que ha crecido demasiado.
- **Clase perezosa:** Una clase que hace muy poco.
- **Envidia de clase:** Una clase que usa en exceso métodos de otra clase.
- **Uso excesivo de literales:** Los literales deben codificarse como constantes con nombre para mejorar la legibilidad y evitar errores de programación. Además, estos pueden y deben externalizarse en archivos de recursos u otros almacenes de datos.
- **Complejidad ciclomática alta:** Incluye lógica condicional compleja o muchos bucles. Esto puede indicar que una función debe dividirse en funciones más pequeñas o que tiene potencial para simplificarse.
- **Agrupación de datos:** Ocurre cuando un grupo de variables se pasan juntas en varias partes del programa. En general, esto sugiere que sería más apropiado agrupar formalmente las diferentes variables en un solo objeto y, en su lugar, pasar solo el nuevo objeto.

## **CODE SMELLS A NIVEL DE MÉTODO:**

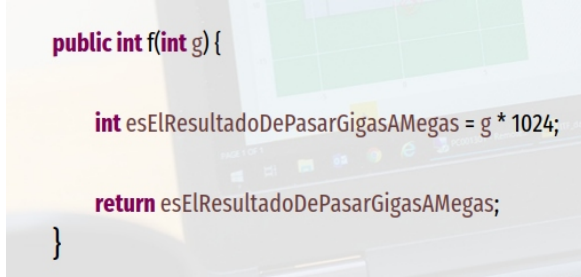
- **Demasiados parámetros:** Una lista larga de parámetros es difícil de leer y hace que las llamadas a la función y la prueba de estas sean complicadas.
- **Método largo:** Un método que ha crecido hasta hacerse demasiado grande e incluye fragmentos de código secuencial muy extensos.
- **Identificadores excesivamente largos:** Los nombres de las variables y los métodos deben reflejar la función dentro del programa; ahora bien, no pueden definir en el mismo nombre su funcionalidad, para ello deben utilizar un comentario.
- **Identificadores excesivamente cortos:** Se desaconseja totalmente usar nombres de variables tales como i (para indicar el tipo de IVA), t (para indicar el total de un cálculo), aux, etcétera.
- **Envidia de objeto:** Un método accede continuamente a los datos de un objeto de una clase diferente a la clase en la que está definida. Posiblemente, el método debería pertenecer a la

otra clase.

– **Comentarios excesivos:** Una clase, una función o un método tiene comentarios irrelevantes o triviales.

– **Línea de código excesivamente larga:** Una línea de código demasiado larga hace que el código sea difícil de leer, comprender, depurar y mantener.

**EJEMPLO:** Qué Code smells puedes identificar. Qué cambios propones?



```
public int f(int g) {  
  
    int esElResultadoDePasarGigasAMegas = g * 1024;  
  
    return esElResultadoDePasarGigasAMegas;  
}
```

Se identifican los code smells siguientes:

- **Nombre misterioso:** no sabemos qué significa el nombre «**f**» de la función.
- **Identificadores excesivamente cortos:** el parámetro de la función.
- **Identificadores excesivamente largos:** la variable para calcular el resultado.

Tras las modificaciones, el resultado podría ser el siguiente:

```
public int deGigasAMegas(int gigas) {  
  
    int megas = gigas * 1024;  
  
    return megas;  
}
```

## **FORMAS DE ANALIZAR EL CÓDIGO FUENTE**

### **– Revisiones de código sobre el hombro**

- Consiste en un programador experimentado que revisa el código fuente de otro programador, en el mismo puesto de trabajo, mediante una conversación informal.
- En este tipo de conversaciones se pueden sugerir cambios de mejora o cambios por un error detectado.

### **– Programación por pares**

- Dos programadores están trabajando con el mismo ordenador,
  - uno está programando
  - mientras el otro revisa y sugiere los cambios o las modificaciones en tiempo real.
- Es poco utilizada por el coste económico que supone para la empresa tener dos programadores en una única tarea.

### **– Sistema de Control de Versiones**

- Son repositorios donde se almacenan los cambios del código fuente del proyecto (a menudo en un servidor).
- Una versión es el estado en el que se encuentra el proyecto en un momento dado de su desarrollo o modificación.
- Hoy en día son muy utilizados ya que existen muchos equipos de trabajo que se encuentran distribuidos geográficamente y no trabajan en la misma oficina.

### **– Con ayuda de herramientas**

- Estos programas analizan desde amenazas a la seguridad hasta problemas de cumplimiento de propiedad intelectual.

### **3. REFACTORIZACIÓN**

Es **reestructurar el código** sin cambiar su comportamiento.

Es **muy importante** porque facilita la adaptación a la nuevas necesidades.

Se aplicará después de analizar el código.

#### **BENEFICIOS DE REFACTORIZAR**

- **Mejora su diseño.** Si modificamos y ampliamos nuestro programa, este pierde la estructura y suele aparecer código duplicado que debemos eliminar para simplificar su mantenimiento.
- **Legibilidad del código.** Será más fácil de entender y facilitará el mantenimiento.
- **Encontrar errores.** Al refactorizar un programa, se pueden apreciar con mayor facilidad los flujos de los algoritmos y sus funcionalidades.
- **Programar más rápido.** Al mejorar el diseño y la legibilidad del código y reducir los errores que se cometen al programar, se mejora la productividad de los programadores.

#### **CLEAN CODE**

Nace de los beneficios de refactorizar.

Es **código limpio** que es fácil de entender y modificar.

#### **REFACTORIZACIÓN EN ECLIPSE: Menú Refactorización**

Redenominar...	
Mover...	Alt+Mayús+V
Cambiar signatura de método...	Alt+Mayús+C
Extraer método...	Alt+Mayús+M
Extraer variable local...	Alt+Mayús+L
Extraer constante...	
Incorporar...	Alt+Mayús+I
Convertir variable local en campo...	
Convertir clase anónima en anidada...	
Move Type to New File...	
Extraer interfaz...	
Extraer superclase...	
Utilizar supertipo cuando sea posible...	
Promover...	
Degradar...	
Extraer clase...	
Introducir el parametro del objeto	
Introducir direccionamiento indirecto...	
Introducir fábrica...	
Introducir parámetro...	
Autoencapsular campo...	
Generalizar tipo declarado...	
Inferir argumentos de tipo genérico...	

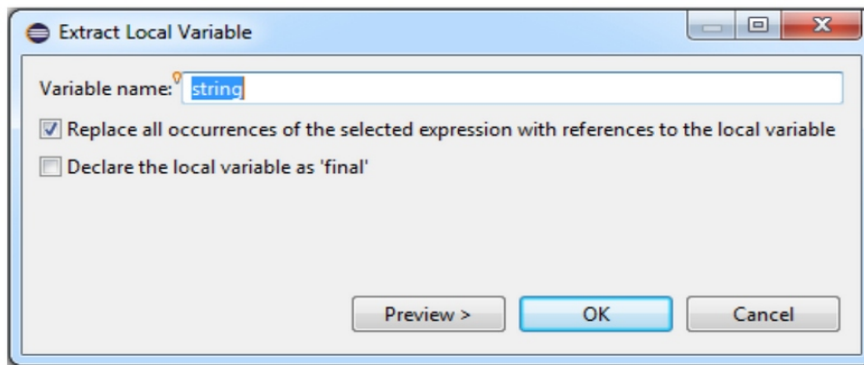
**Video explicativo:** <https://player.vimeo.com/video/613846833?byline=0&badge=0&portrait=0&title=0>

– **Rename (Redenominar):** Es una de las opciones más utilizadas. Cambia, el nombre de las variables, las clases, los métodos, los paquetes, etc en toda la aplicación.

– **Move (Mover):** Mueve una clase de un paquete a otro. Se mueve el fichero .java a la carpeta del nuevo paquete y **se cambian todas las referencias (imports)** en el proyecto.

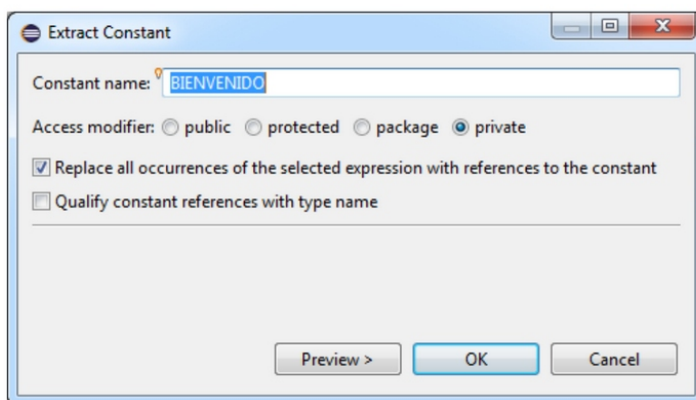
– **Extract Local Variable (Extraer variable local):** Asigna una expresión a una variable local.

```
public static void main(String[] args) {
    System.out.println("Bienvenido!");
}
```



– **Extract Constant(Extraer constante):** Convierte un número o una cadena de texto en una constante.

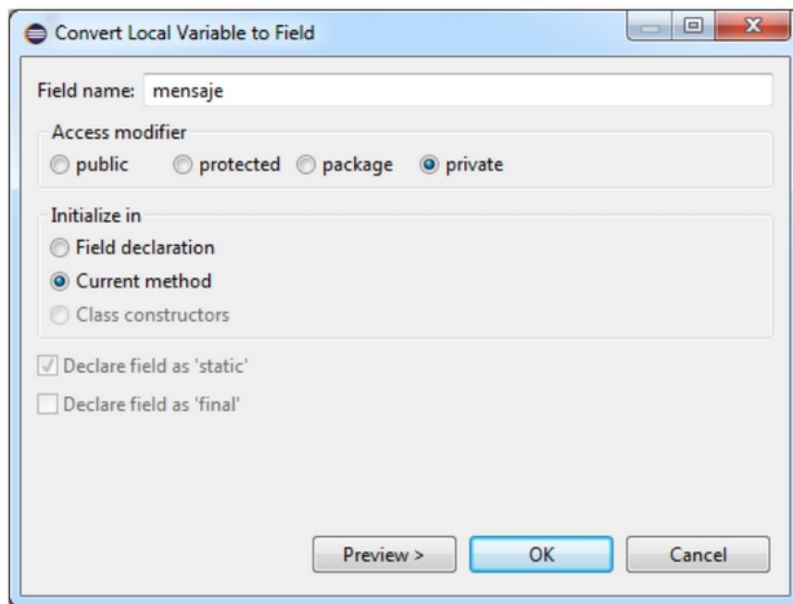
Partiendo del mismo código de ejemplo que en el caso anterior



```
public class Refactorizacion {
    private static final String BIENVENIDO = "Bienvenido!";
    public static void main(String[] args) {
        System.out.println(BIENVENIDO);
    }
}
```

– **Convert Local Variable to Field:** Convierte una variable de un método local en un atributo privado de la clase. Se cambian todas las ocurrencias de la variable.

```
public static void main(String[] args) {
    String mensaje = "Bienvenido!";
    System.out.println(mensaje);
}
```



```
public class Refactorizacion {
    private static String mensaje;
    public static void main(String[] args) {
        mensaje = "Bienvenido!";
        System.out.println(mensaje);
    }
}
```

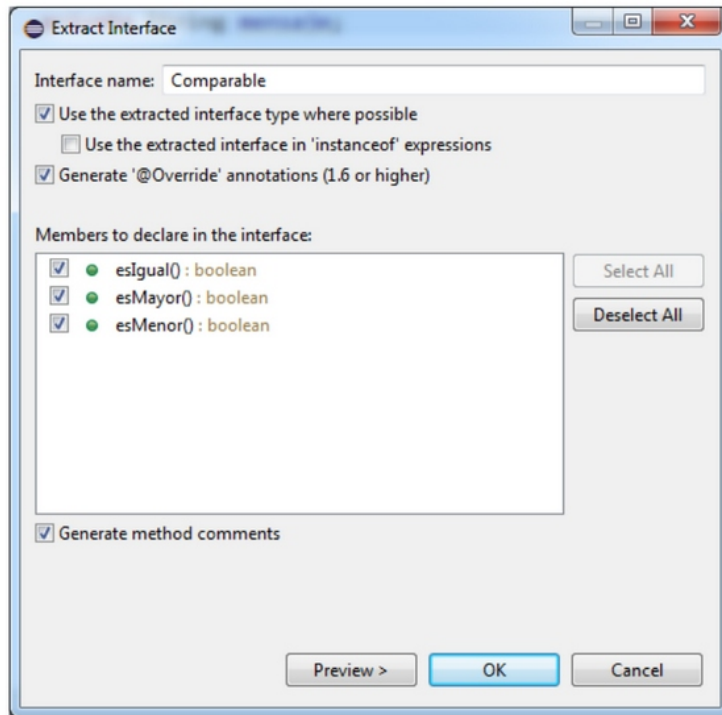
– **Extract Interface:** Se crea automáticamente una interfaz con los métodos de la clase.

Permite escoger qué métodos de la clase formarán parte de la interfaz.

Automáticamente, Eclipse hará que nuestra clase implemente la nueva interfaz y dará al programador la opción de marcar los métodos como sobrescritos (`@Override`).

```
public class Refactorizacion {
    public boolean esMayor() {
        return true;
    }
    public boolean esMenor() {
        return true;
    }
    public boolean esIgual() {
        return true;
    }
}
```





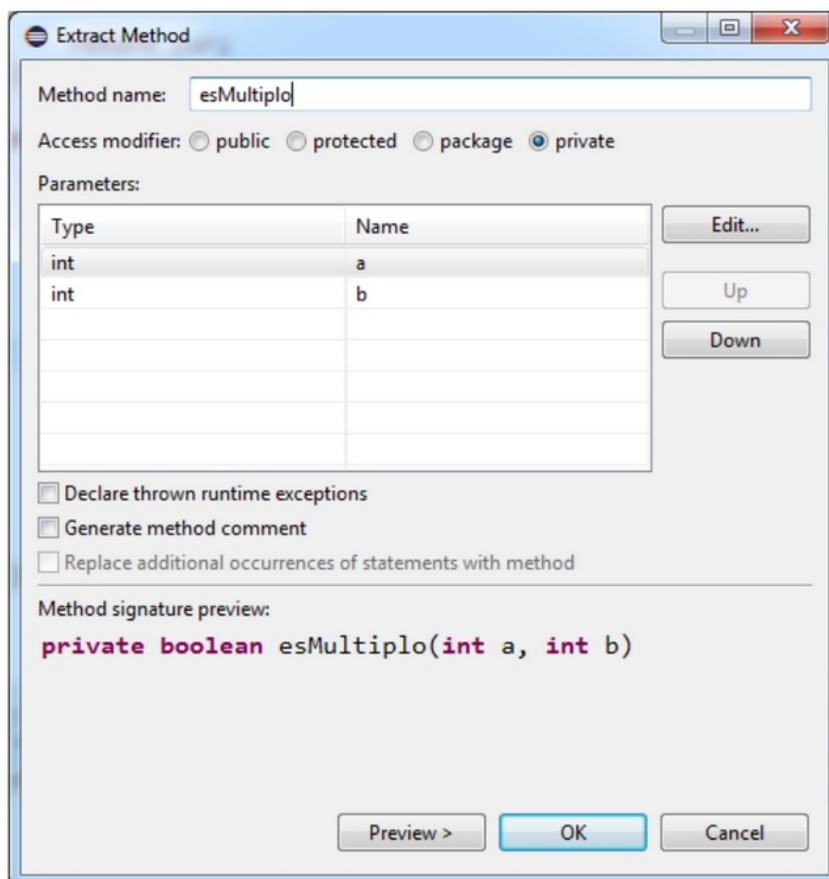
```
public class Refactorizacion implements Comparable {  
    @Override  
    public boolean esMayor() {  
        return true;  
    }  
    @Override  
    public boolean esMenor() {  
        return true;  
    }  
    @Override  
    public boolean esIgual() {  
        return true;  
    }  
}  
  
public interface Comparable {  
    boolean esMayor();  
    boolean esMenor();  
    boolean esIgual();  
}
```

- **Extract Superclass:** Extrae una superclase en lugar de una interfaz. También se pueden seleccionar los métodos que formarán parte de la superclase. A diferencia de la creación de una interfaz, los métodos se pasan a la nueva clase, y esto, si hay referencias en alguna otra parte del proyecto, puede llevar a fallos de compilación.
- **Extract Method:** Extraemos un fragmento de código de un método para crear un nuevo método. En el nuevo método se debe configurar el nombre, la visibilidad y el tipo devuelto.

Si partimos de la función siguiente, que devuelve verdadero si el número pasado como parámetro es primo y falso en caso contrario, y seleccionamos el código de la condición (como puedes observar en el código), podemos crear una nueva función que determine si un número es múltiplo de otro o no.

Eclipse creará de forma automática el método y sustituirá en el método actual el código reemplazado por la función.

```
public boolean esPrimo(int numero) {  
    int contador = 2;  
    boolean primo = true;  
    while (primo && contador != numero) {  
        if (numero % contador == 0)  
            primo = false;  
        contador++;  
    }  
    return primo;  
}
```



```

public boolean esPrimo(int numero) {
    ...
    if(esMultiplo(numero, contador))
        primo= false;
    ...
}
private boolean esMultiplo(int a, int b) {
    return a% b == 0;
}

```

- **Change Method Signature:** Este proceso nos permite cambiar la cabecera o signatura de un método. La cabecera de un método está compuesta por el nombre y sus parámetros. De forma automática, se actualizarán todas las llamadas a este método en nuestro proyecto. Este proceso puede provocar errores de compilación que deberemos cambiar manualmente.

### **CASO PRÁCTICO 1:**

Realiza dos veces el proceso de **extraer una variable local** para la clase:

```

public class Refactorizacion {

    public static void main(String[] args){

        System.out.println( 5 + 8 );
    }
}

```

Después de refactorizar, la clase quedaría de la manera siguiente:

```

public class Refactorizacion {

    public static void main(String[] args){

        int i = 5;
        int j = 8;
        System.out.println(i+ j);
    }
}

```

**CASO PRÁCTICO 2:** Dada la clase, deduce qué refactorización es adecuada realizar, teniendo en cuenta que existe un valor que **no va a cambiar nunca**.

```

public class Refactorizacion {
    public double perimetro(int radio) {

        return 2 * 3.1416 * radio;
    }
}

```

Resultado:

```

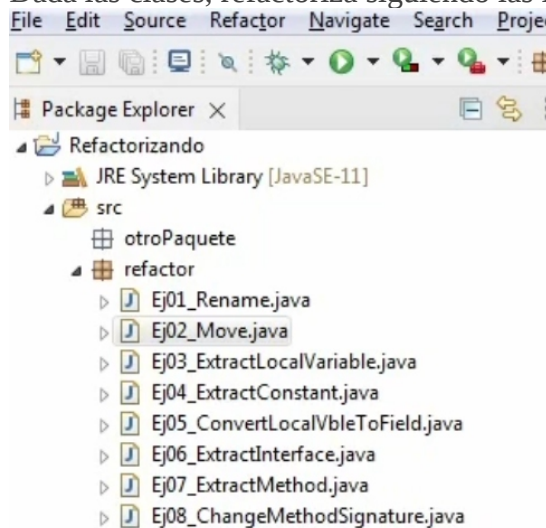
private static final double PI = 3.1416;

public double perimetro(int radio) {
    return 2*PI*radio;
}

```

### CASO PRÁCTICO 3:

Dada las clases, refactoriza siguiendo las instrucciones del vídeo:



<https://player.vimeo.com/video/613846833?byline=0&badge=0&portrait=0&title=0>

Crear el paquete con todas las clases o importar el proyecto que se proporciona.