ÍNDICE

1. Introducción

¿Podemos mejorar la estructura del código y que sea de mayor calidad, sin que cambie su comportamiento? ¿Cómo hacerlo? ¿Qué patrones hay que seguir?

La **refactorización** es una técnica, que consiste en realizar **pequeñas transformaciones** en el código de un programa, para mejorar la estructura sin que cambie el comportamiento ni funcionalidad del mismo. Su objetivo es mejorar la estructura, la legibilidad o la eficiencia del código.

Con la refactorización se mejora el diseño del software, hace que el software sea más fácil de entender, hace que el mantenimiento del software sea más sencillo, la refactorización ayuda a que el programa sea más rápido.

La idea de refactorización de código, se basa en el concepto matemático de factorización de polinomios. Así́, resulta que (x + 1)(x − 1) se puede expresar como x2 − 1 sin que se altere su sentido.

Algunas pistas que nos pueden indicar la necesidad de refactorizar un programa son:

- Código duplicado.
- Métodos demasiado largos.
- Clases muy grandes o con demasiados métodos.
- Métodos más interesados en los datos de otra clase que en los de la propia.
- Grupos de datos que suelen aparecer juntos y parecen más una clase que datos sueltos.
- Clases con pocas llamadas o que se usan muy poco.
- Exceso de comentarios explicando el código.

Hay que resaltar que la refactorización no cambia el comportamiento observable del software. El software sigue cumpliendo la misma función que hacía antes. Ningún usuario, ya sea usuario final u otro programador, podrá determinar qué cosas han cambiado.

2. Patrones de refactorización más habituales.

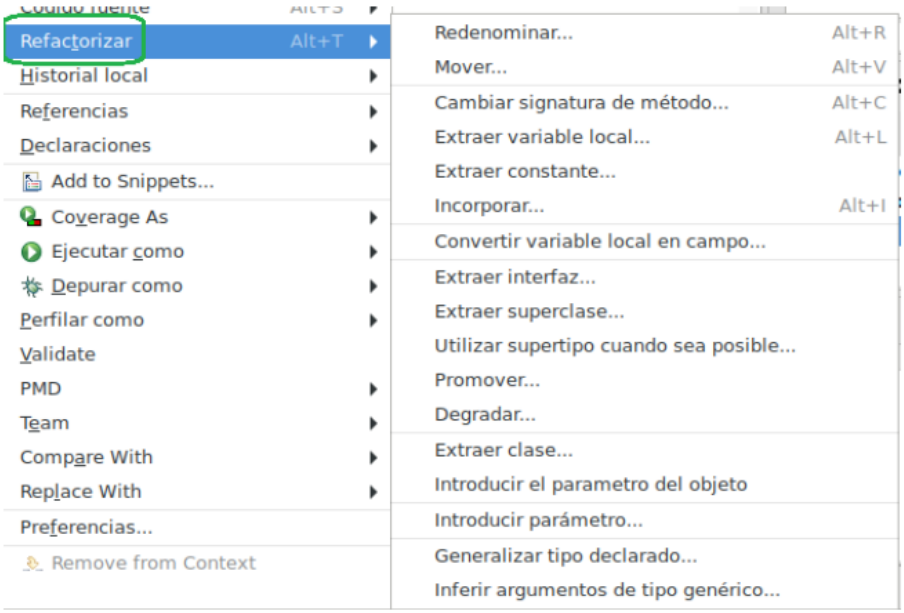
Algunos de los patrones más habituales de refactorización, que vienen ya integrados en la mayoría de los entornos de desarrollos, son los siguientes:

- **Renombrar**. Cambiar el nombre de un paquete, clase, método o campo, por un nombre más significativo.
- **Encapsular campos**. Crear métodos de asignación y de consulta (getters y setters) para los campos de la clase, que permitan un control sobre el acceso de estos campos, debiendo hacerse siempre mediante el uso de estos métodos.
- **Sustituir bloques de código por un método**. En ocasiones se observa que un bloque de código puede constituir el cuerpo de un método, dado que implementa una función por sí mismo o aparece repetido en múltiples sitios. De esta forma, cada vez que queramos acceder a ese bloque de código, bastaría con invocar al método.
- **Modificar la extensión del código**. Hacer un código más extenso si se gana en claridad o menos extenso sólo si con eso se gana eficiencia. Reorganizar código condicional complejo. Patrón aplicable cuando existen varios if o condiciones anidadas o complejas.
- **Crear código común** (en una clase o método) para evitar el código repetido.
- **Mover la clase**. Mover una clase de un paquete a otro, o de un proyecto a otro. Esto implica la actualización en todo el código fuente de las referencias a la clase en su nueva localización.

- **Borrado seguro**. Garantizar que cuando un elemento del código ya no es necesario, se borran todas las referencias a él que había en cualquier parte del proyecto.
- **Cambiar los parámetros del método**. Permite añadir/modificar/eliminar los parámetros en un método y cambiar los modificadores de acceso. Extraer la interfaz. Crea una nueva interfaz de los métodos public non-static seleccionados en una clase o interfaz.

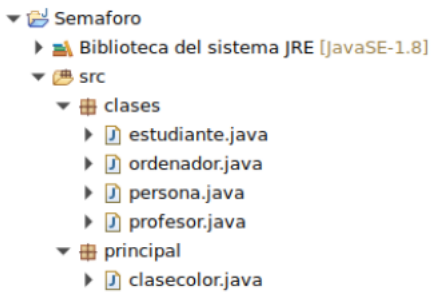
3. Refactorización en Eclipse.

A continuación, se muestra el menú contextual disponible al hacer clic con el botón secundario sobre un fragmento de código en algunas versiones de Eclipse. Tras escoger la opción **Refactorizar** aparece un menú con muchas opciones, de las que estudiaremos algunas.



Nota: el menú mostrado es contextual, por lo que la opción **Refactorizar**, podrá mostrar algunas opciones diferentes en función de la porción de código sobre la que sea llamado.

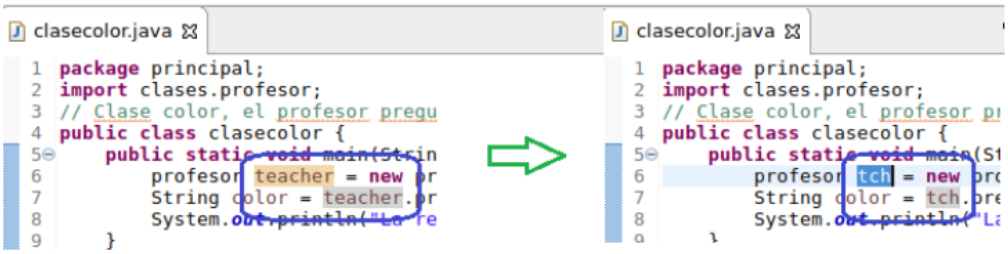
La mayor parte de estas funciones permanecen disponibles en las versiones más actuales de Eclipse. A continuación, se van a explorar las funciones de refactorización utilizando el proyecto semaforo que ya conocemos de ejercicios anteriores.



3.1. Redenominar

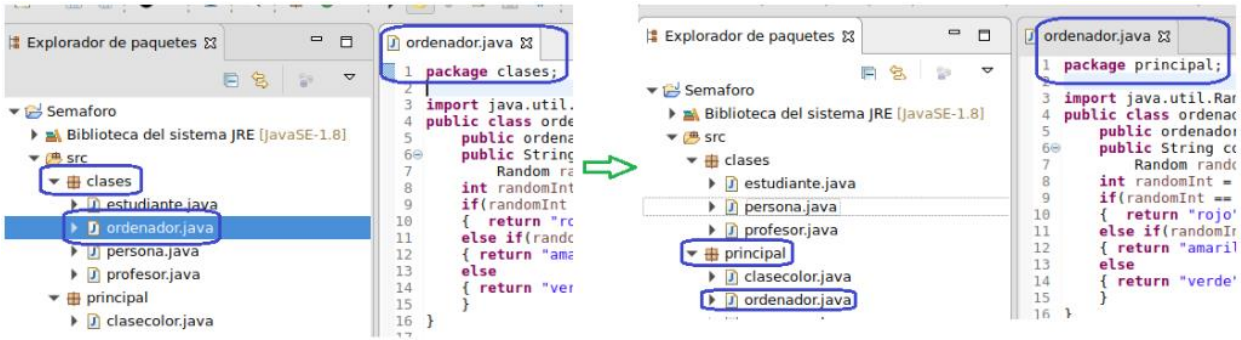
Es la opción más común. Modifica el nombre a cualquier elemento (variable, atributo, método, clase...) y hace los cambios necesarios en las referencias que haya a dicho elemento en todo el proyecto.

Ejemplo: sustituir en el método main el nombre de la variable local teacher por tch. El cambio se realiza en todas sus apariciones.



3.2. Mover.

Cambia una clase de un paquete a otro, afectando a su declaración “package” y a su localización en el disco. Ejemplo: mover la clase ordenador del paquete clases al paquete principal.



3.3. Cambiar signatura del método.

Modifica la “firma” o cabecera del método. Si el método ha sido ya usado, cambiar el número o tipo de parámetros (así como el tipo de valor devuelto) provocará fallos de compilación.

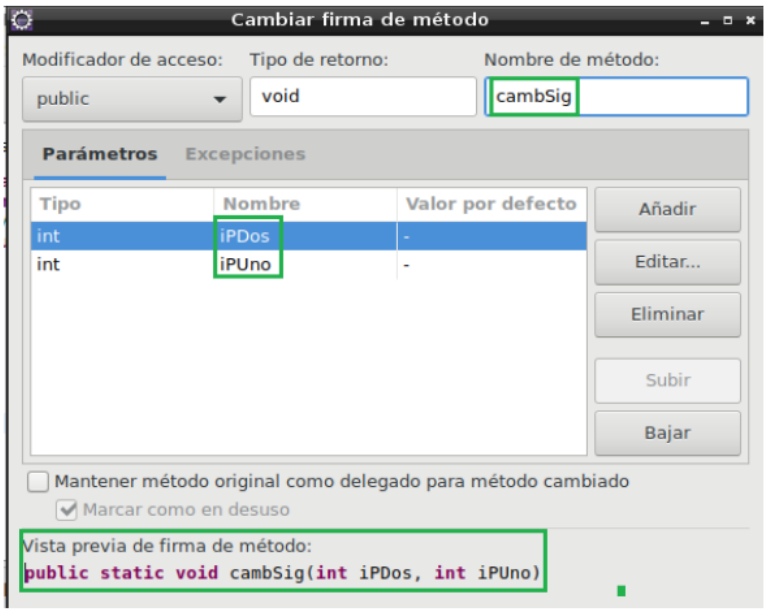
Es útil para cambiar el nombre de los parámetros, o su orden (Eclipse modificará también el orden de entrada de los parámetros en todas las llamadas al método).

Ejemplo: cambiar el nombre del método y de los parámetros al método public void CambSignatura(int iParmUno, int iParmDos):

```java
package principal;
import clases.profesor;
// Clase color, el profesor pregunta a un alumno por un color ent
public class clasecolor {
    public static void main(String[] args) {
        profesor tch = new profesor();
        String color = tch.preguntacolor();
        System.out.println("La respuesta recibida es:" + color);

        CambSignatura(5,10);
    }

    public static void CambSignatura(int iParmUno, int iParmDos)
    {
        System.out.println("Primer Parametro" + iParmUno);
        System.out.println("Segundo Parametro" + iParmDos);
    }
}
```



clasecolor.java

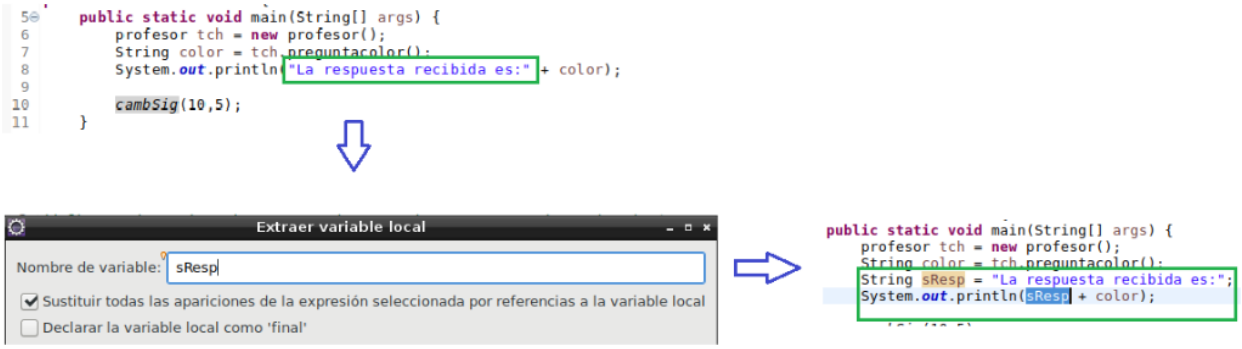
```java
package principal;
import clases.profesor;
// Clase color, el profesor pregunta a un alumno por un color en
public class clasecolor {
    public static void main(String[] args) {
        profesor tch = new profesor();
        String color = tch.preguntacolor();
        System.out.println("La respuesta recibida es:" + color);

        cambSig(10,5);
    }

    public static void cambSig(int iPDos, int iPUno)
    {
        System.out.println("Primer Parametro" + iPUno);
        System.out.println("Segundo Parametro" + iPDos);
    }
}
```

3.4. Extraer variable local.

Crear una variable local inicializada con el valor de un literal (número, String...). Las referencias a esa expresión se modifican por una referencia a la variable.



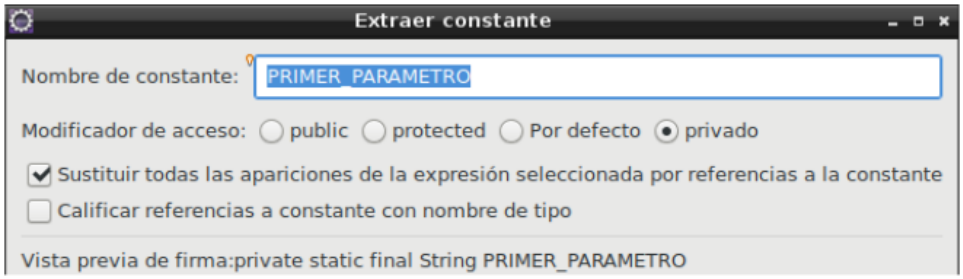
Nota: Sólo afecta al ámbito actual, es decir, si la expresión existe por ejemplo en otro método, no se hará ningún cambio en ese otro método.

3.5. Extraer constante.

Exactamente igual que el anterior, pero genera una constante con la expresión seleccionada.

Ejemplo: sobre el método cambSig anteriormente creado, convertir el literal "Primer parametro" en una constante en el ámbito de la clase.

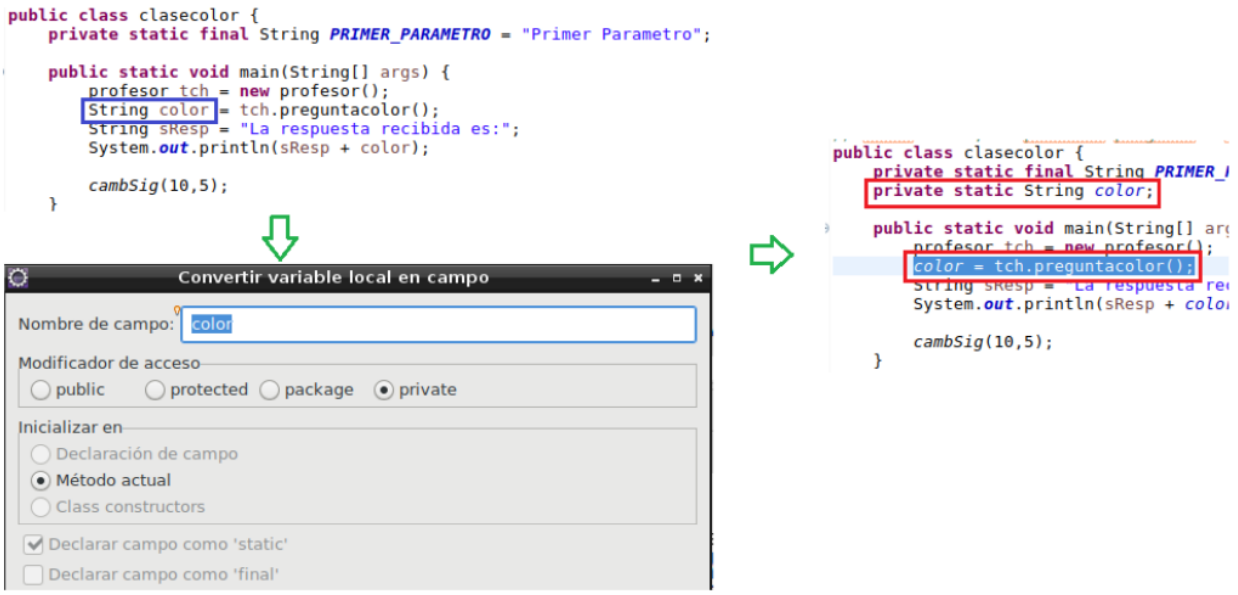
```
public static void cambSig(int iPDos, int iPUno)
{
    System.out.println("Primer Parametro" + iPUno);
    System.out.println("Segundo Parametro" + iPDos);
}
```




```
4  public class clasecolor {
5      private static final String PRIMER_PARAMETRO = "Primer Parametro";
6
7      public static void cambSig(int iPDos, int iPUno)
8      {
9          System.out.println(PRIMER_PARAMETRO + iPUno);
10         System.out.println("Segundo Parametro" + iPDos);
11     }
```

3.6. Convertir variable local en atributo.

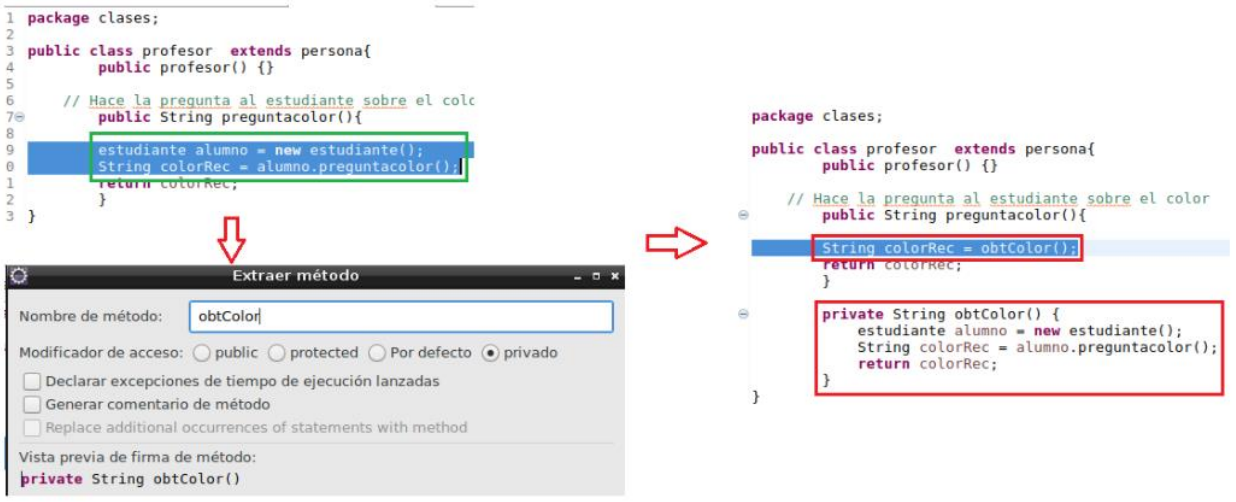
A veces se definen variables locales dentro de un método que luego resultan ser relevantes en el ámbito de la clase, por tanto, debe ser considerada un atributo de la clase.



3.7. Extaer método.

Convierte el código seleccionado en un método, útil en código que es reutilizado en varios sitios del programa. También puede servir para aligerar un método que es demasiado largo.

Eclipse sólo solicita el nombre del método, pero descubre automáticamente los parámetros y tipo de retorno necesarios.



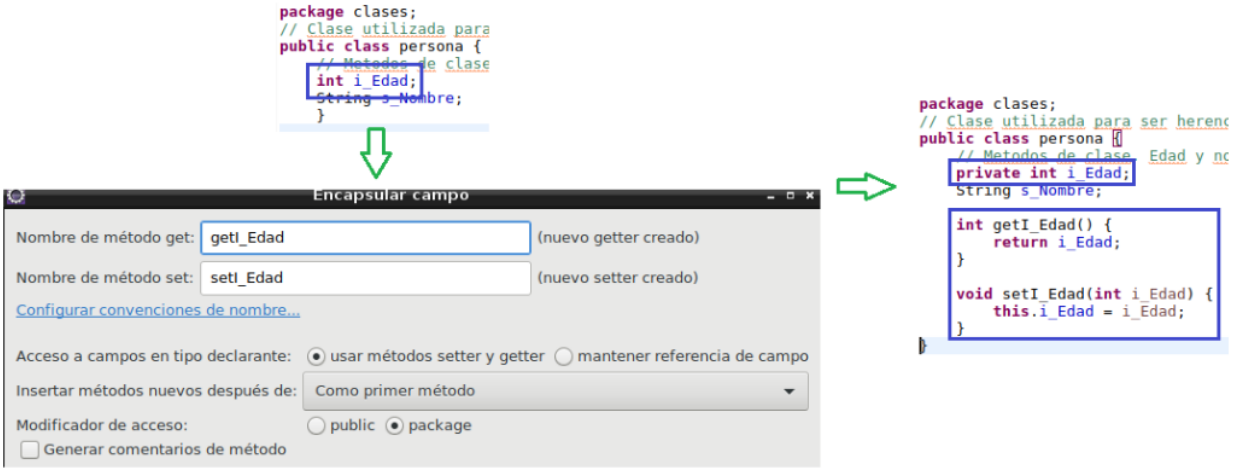
3.8. Incorporar.

Hace lo contrario que los “Extract”: elimina una declaración de variable, método o constante y coloca su valor (en el caso de variables y constantes) o su código (en el caso de métodos) en aquellos lugares en que se referenciaba a esa variable, método o constante que ya no existirán.

Es muy útil cuando se comprueba que el contenido de una variable o constante se va a usar una sola vez y por tanto no merece la pena almacenarlo, sino que queda más limpio el código en una línea. También cuando se observa que un método sólo se llama una o dos veces, por lo que no merece la pena aislar ese código en un método.

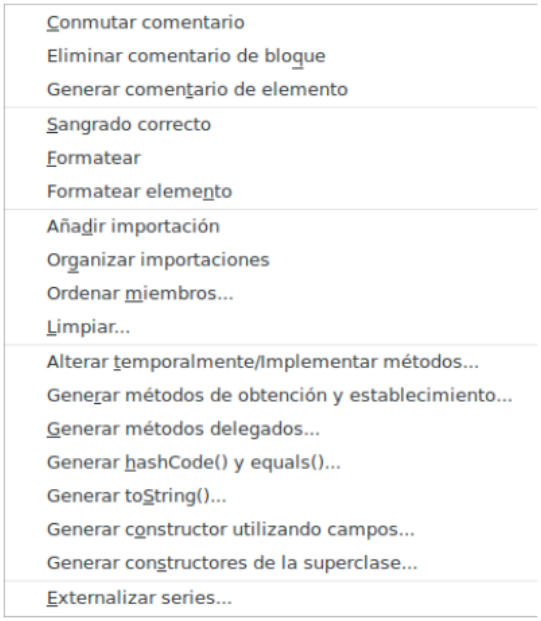
3.9. Autoencapsular atributo.

Convierte una variable de clase en privada y genera los métodos Get y Set públicos para acceder a la misma. Opción también disponible desde el menú código fuente que se verá a continuación.



3.10. Menú código fuente.

Otras muchas opciones para el refactorizado del código aparecen a partir del menú contextual código fuente.



Prueba algunas de las opciones ofrecidas.

4. Analizadores de código.

El análisis estático de código, es un proceso que tiene como objetivo, evaluar el software, sin llegar a ejecutarlo. Esta técnica se va a aplicar directamente sobre el código fuente, para poder obtener información que nos permita mejorar la base de código. Si el analizador considera que nuestro código fuente tiene una estructura mejorable, nos lo indicará y también nos comunicará la mejora a realizar. Por lo tanto, el uso de analizadores de código proporciona información sobre algunos aspectos a considerar en la refactorización de los programas.

Las principales funciones de los analizadores es **encontrar partes del código que puedan reducir el rendimiento**, provocar errores en el software, tener una excesiva complejidad, complicar el flujo de datos, crear problemas de seguridad.

El análisis se realiza siguiendo una serie de reglas predefinidas.

Un ejemplo es **PMD**, una herramienta para Java que basa su funcionamiento en detectar patrones, que son posibles errores en tiempo de ejecución, código que no se puede ejecutar nunca porque no se puede llegar a él, código que puede ser optimizado, expresiones lógicas que pueden ser simplificadas, malos usos del lenguaje, etc.

Otro analizador de código disponible en el mercado es Sonarcube, herramienta open source de análisis de calidad del código disponible para gran cantidad de lenguajes de programación.

4.1. Instalación de PMD

La instalación de PMD como un plugin Eclipse se puede realizar como se indica a continuación:

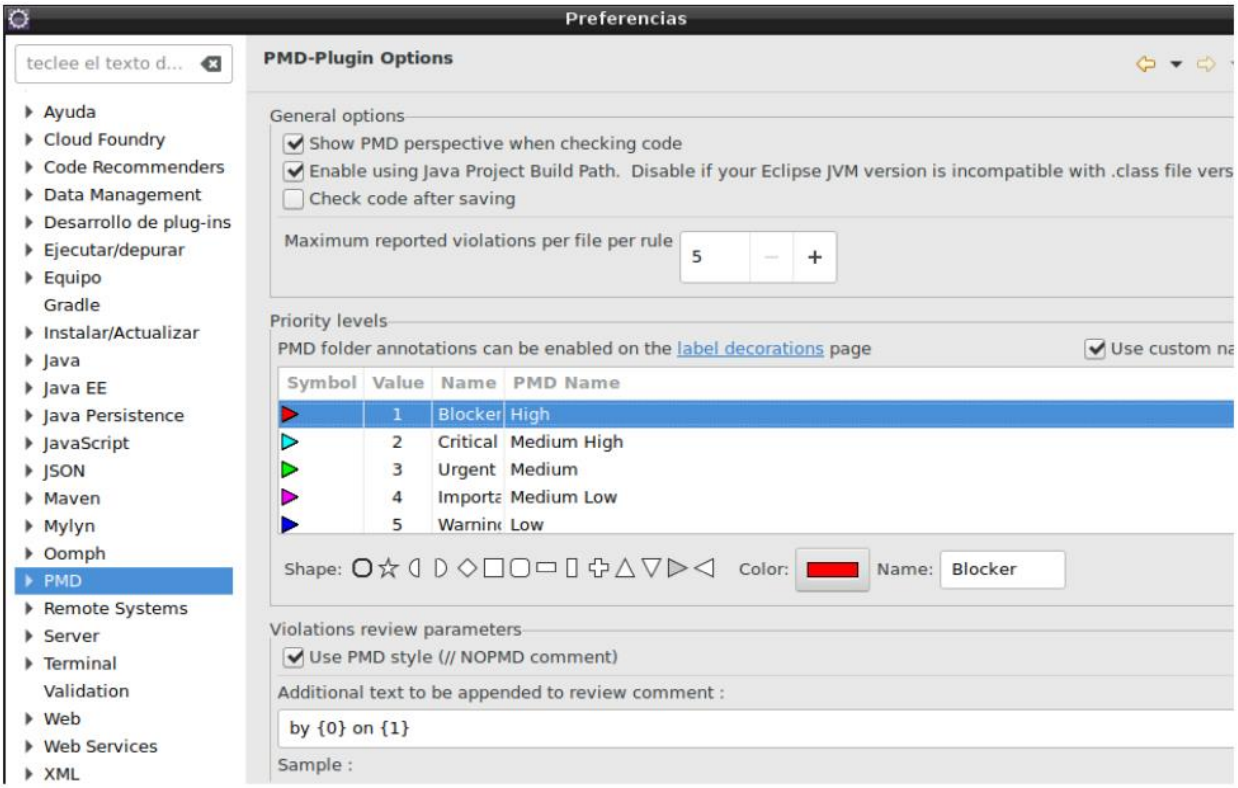
1. En Eclipse, seleccionar la opción de menú Ayuda-> Install New Software.
2. Pulsar el botón Añadir e introducir la siguiente información:
   - Nombre: PMD for Eclipse Update Site.
   - Ubicación: https://dl.bintray.com/pmd/pmd-eclipse-plugin/updates/

   y pulsar Add.

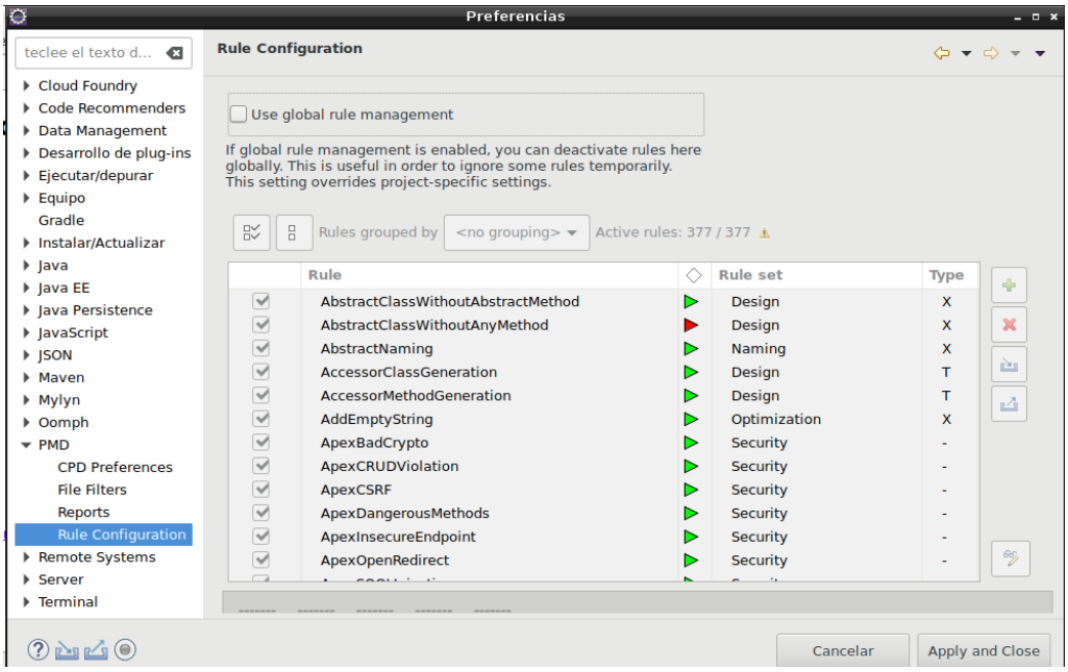
3. Seleccionar PMD for Eclipse 4 y pulsar Siguiente.
4. Aceptar la licencia y continuar con la instalación sin considerar la advertencia de la falta de firma digital del plugin.
5. Reiniciar Eclipse.

4.2. Configuración de PMD.

Una vez instalado el plugin se pueden ver algunos de los parámetros de su configuración en la opción de menú Ventana -> Preferencias -> PMD.



Es especialmente interesante la sección Rule Configuration, donde es posible seleccionar aquellas reglas a las que quieres someter al código. PDM trae gran cantidad de reglas predefinidas, aunque también dispone de herramientas para crear otras personalizadas.



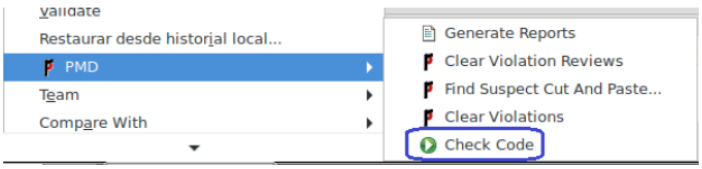
4.3. Ejemplo PMD.

El siguiente ejemplo servirá para entender mejor el plugin PDM.

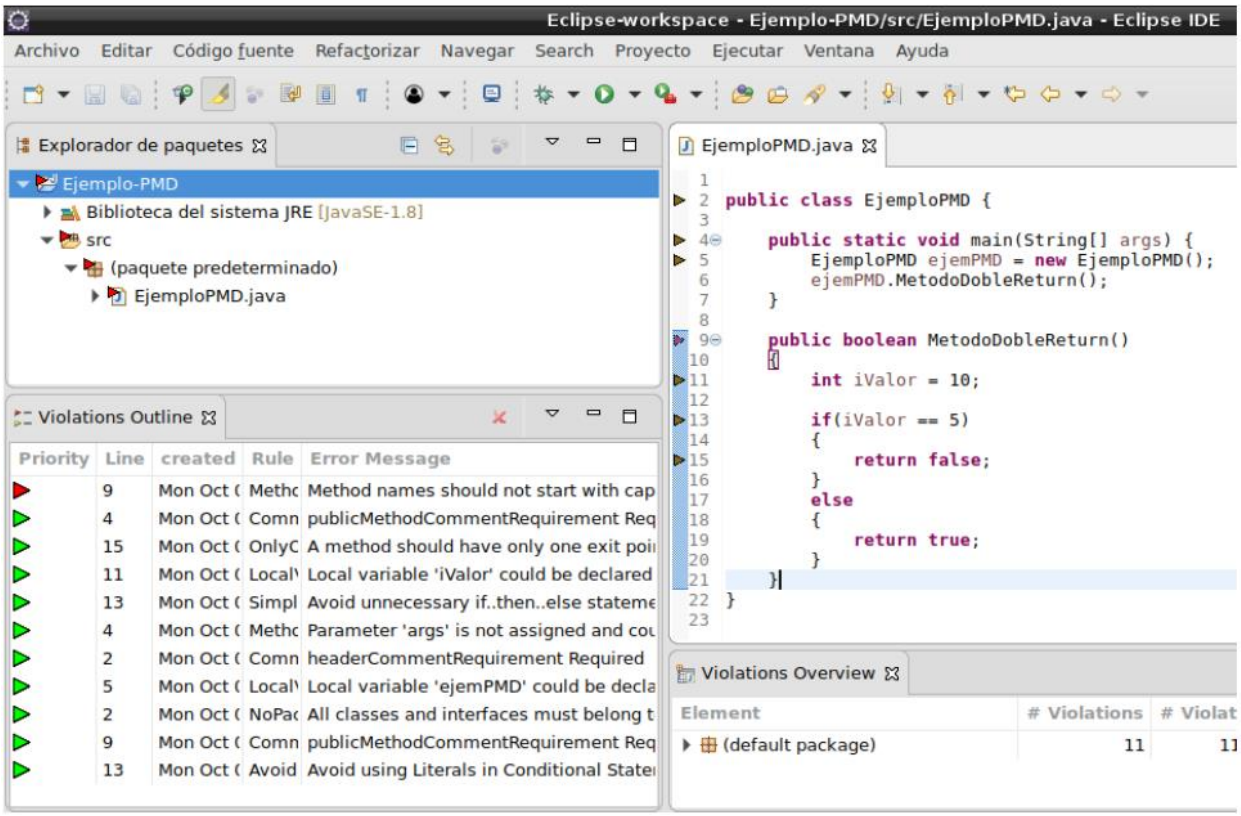
- Crear el proyecto Java Prueba-PMD y la clase EjmeploPMD.
- Incluir el siguiente código.

Clase EjemploPMD (EjemploPMD.java)	
1	public class EjemploPMD {
2	public static void main(String[] args) {
3	EjemploPMD ejemPMD = new EjemploPMD();
4	ejemPMD.MetodoDobleReturn();
5	}
6	public boolean MetodoDobleReturn()
7	{
8	int iValor = 10;
9	if(iValor == 5)
10	{
11	return false;
12	}
13	else
14	{
15	return true;
16	}
17	}
18	}

Sin haber modificado la selección de reglas que ofrece el plugin PMD por defecto, solicitar una validación del código (Check code).



Observar la cantidad de sugerencias que nos ofrece PMD para mejorar el código. **Once** violaciones de las reglas definidas en un programa de unas 20 líneas de código.


Eclipse-workspace - Ejemplo-PMD/src/EjemploPMD.java - Eclipse IDE
Archivo Editar Código fuente Refactorizar Navegar Search Proyecto Ejecutar Ventana Ayuda
Explorador de paquetes
Ejemplo-PMD
Biblioteca del sistema JRE [JavaSE-1.8]
src
(paquete predeterminado)
EjemploPMD.java
Violations Outline
Priority Line created Rule Error Message
9 Mon Oct (Methc Method names should not start with cap
4 Mon Oct (Comn publicMethodCommentRequirement Req
15 Mon Oct (OnlyC A method should have only one exit poi
11 Mon Oct (LocalV Local variable 'iValor' could be declared
13 Mon Oct (Simpl Avoid unnecessary if..then..else stateme
4 Mon Oct (Methc Parameter 'args' is not assigned and cou
2 Mon Oct (Comn headerCommentRequirement Required
5 Mon Oct (LocalV Local variable 'ejemPMD' could be decla
2 Mon Oct (NoPac All classes and interfaces must belong t
9 Mon Oct (Comn publicMethodCommentRequirement Req
13 Mon Oct (Avoid Avoid using Literals in Conditional State
EjemploPMD.java
1
2 public class EjemploPMD {
3
4 public static void main(String[] args) {
5 EjemploPMD ejemPMD = new EjemploPMD();
6 ejemPMD.MetodoDobleReturn();
7 }
8
9 public boolean MetodoDobleReturn()
10 {
11 int iValor = 10;
12
13 if(iValor == 5)
14 {
15 return false;
16 }
17 else
18 {
19 return true;
20 }
21 }
22 }
23
Violations Overview
Element # Violations # Violat
(default package) 11 11