

TEMA 3: Optimización y documentación.

Refactorización

1. Refactorización

La refactorización 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 interna del código. Es una tarea que pretende limpiar el código minimizando la posibilidad de introducir errores.

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 nos ayuda a encontrar errores y a que nuestro programa sea más rápido.

Cuando se refactoriza se está mejorando el diseño del código después de haberlo escrito. Podemos partir de un mal diseño y, aplicando la refactorización, llegaremos a un código bien diseñado. Cada paso es simple, por ejemplo, mover una propiedad desde una clase a otra, convertir determinado código en un nuevo método, etc. La acumulación de todos estos pequeños cambios puede mejorar de forma ostensible el diseño.

1.1. Concepto

<p>FACTORIZACIÓN DE POLINOMIOS</p> $X^2 - 1 = (X + 1)(X - 1)$

El concepto de refactorización de código, se basa en el concepto matemático de factorización de polinomios.

Refactorización: Cambio hecho en la estructura interna del software para hacerlo más fácil de entender y de modificar sin cambiar su comportamiento.

Ejemplos de refactorización es “Extraer Método” y “Encapsular Campos”. La refactorización es normalmente un cambio pequeño en el software que mejora su mantenimiento.

Campos encapsulados: Se aconseja crear métodos getter y setter, (de consulta y de asignación) para cada campo que se defina en una clase. Cuando sea necesario acceder o modificar el valor de un campo, basta con invocar al método getter o setter según convenga.

Refactorizar: Reestructurar el software aplicando una serie de refactorizaciones sin cambiar su comportamiento.

El propósito de la refactorización es hacer el software más fácil de entender y de modificar. Se pueden hacer muchos cambios en el software que pueden hacer algún pequeño cambio en el comportamiento observable.

Solo los cambios hechos para hacer el software más fácil de entender son refactorizaciones.

Hay que diferenciar la **refactorización** de la **optimización**. En ambos procesos, se pretende mejorar la estructura interna de una aplicación o componente, sin modificar su comportamiento. Sin embargo, cuando se optimiza, se persigue una mejora del rendimiento, por ejemplo, mejorar la velocidad de ejecución, pero esto puede hacer un código más difícil de entender.

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.

Con la refactorización de código, estamos modificando un código que funciona correctamente, pero merece la pena el esfuerzo de refactorizar un código ya implementado, para que se comprenda mejor.

Se conoce como Bad Smell o Code Smell (mal olor) a algunos indicadores o síntomas del código que posiblemente ocultan un problema más profundo. Los bad smells no son errores de código, bugs, ya que no impiden que el programa funcione correctamente, pero son indicadores de fallos en el diseño del código que dificultan el posterior mantenimiento del mismo y aumentan el riesgo de errores futuros. Algunos de estos síntomas son:

- **Código duplicado** (Duplicated code). Si se detectan bloques de código iguales o muy parecidos en distintas partes del programa, se debe extraer creando un método para unificarlo.
- **Métodos muy largos** (Long Method). Los métodos de muchas líneas dificultan su comprensión. Un método largo probablemente está realizando distintas tareas, que se podrían dividir en otros métodos. Las funciones deben ser las más pequeñas posibles (3 líneas mejor que 15). Cuanto más corto es un método, más fácil es reutilizarlo.
- **Clases muy grandes** (Large class). Problema anterior aplicado a una clase. Una clase debe tener solo una finalidad. Si una clase se usa para distintos problemas tendremos clases con demasiados métodos, atributos e incluso instancias. Las clases deben el menor número de responsabilidades y que esté bien delimitado.
- **Lista de parámetros extensa** (Long parameter list). Las funciones deben tener el mínimo número de parámetros posible, siendo cero lo perfecto. Si un método requiere muchos parámetros puede que sea necesario crear una clase con esa cantidad de datos y pasarle un objeto de la clase. Un método debe hacer solo una cosa, hacerla bien, y que sea la única que haga.
- **Cambio divergente** (Divergent change). Si una clase necesita ser modificada a menudo y por razones muy distintas, puede que la clase esté realizando demasiadas tareas. Podría ser eliminada y/o dividida.
- **Envidia de funcionalidad** (Feature Envy). Ocurre cuando una clase usa más métodos de otra clase, o un método usa más datos de otra clase, que de la propia.

- **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 ya que no tienen comportamiento alguno.
- **Legado rechazado** (Refused bequest). Cuando una subclase extiende (hereda) de otra clase, y utiliza pocas características de la superclase, puede que haya un error en la jerarquía de clases.

1.2. Limitaciones.

En algunos casos la refactorización ha presentado problemas en algunos aspectos del desarrollo.

Un área problemática de la refactorización son las bases de datos. Una base de datos presenta muchas dificultades para poder ser modificada, dado la gran cantidad de interdependencias que soporta. Cualquier modificación que se requiera de las bases de datos, incluyendo modificación de esquema y migración de datos, puede ser una tarea muy costosa. Es por ello que la refactorización de una aplicación asociada a una base de datos, siempre será limitada, ya que la aplicación dependerá del diseño de la base de datos.

Otra limitación, es cuando cambiamos interfaces. Cuando refactorizamos, estamos modificando la estructura interna de un programa o de un método. El cambio interno no afecta al comportamiento ni a la interfaz. Sin embargo, si renombramos un método, hay que cambiar todas las referencias que se hacen a él. Siempre que se hace esto se genera un problema si es una interfaz pública. Una solución es mantener las dos interfaces, la nueva y la vieja, ya que, si es utilizada por otra clase o parte del proyecto, no podrá referenciarla.

Hay determinados cambios en el diseño que son difíciles de refactorizar. Es muy difícil refactorizar cuando hay un error de diseño o no es recomendable refactorizar, cuando la estructura a modificar es de vital importancia en el diseño de la aplicación.

Hay ocasiones en las que no debería refactorizar en absoluto. Nos podemos encontrar con un código que, aunque se puede refactorizar, sería más fácil reescribirlo desde el principio. Si un código no funciona, no se refactoriza, se reescribe.

1.3. Patrones de refactorización más habituales

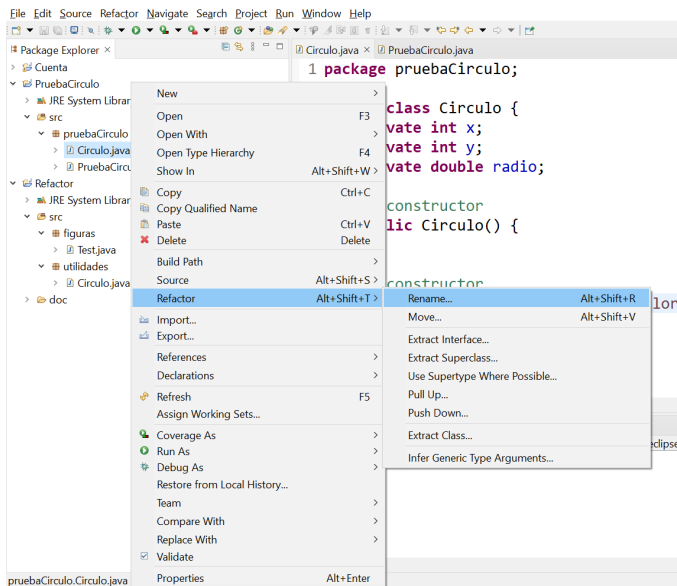
En el proceso de refactorización, se siguen una serie de patrones preestablecidos, los más comunes son los siguientes:

1. **Renombrado** (rename): Este patrón nos indica que debemos cambiar el nombre de un paquete, clase, método o campo, por un nombre más significativo.

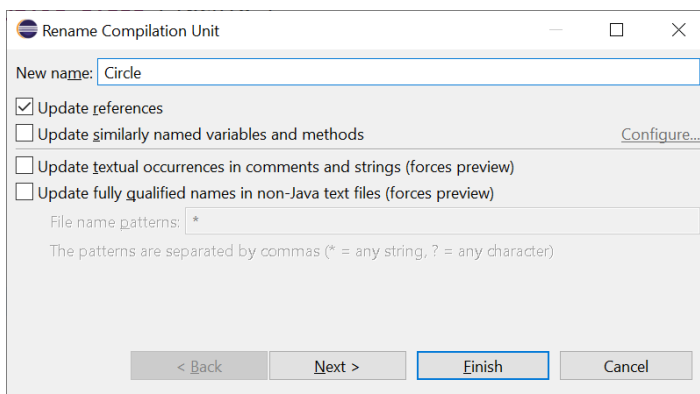
Veamos un ejemplo con las clases Circulo.java y PruebaCirculo.java

Para renombrar una clase, con el botón derecho sobre la misma pulsamos sobre refactor Remane...

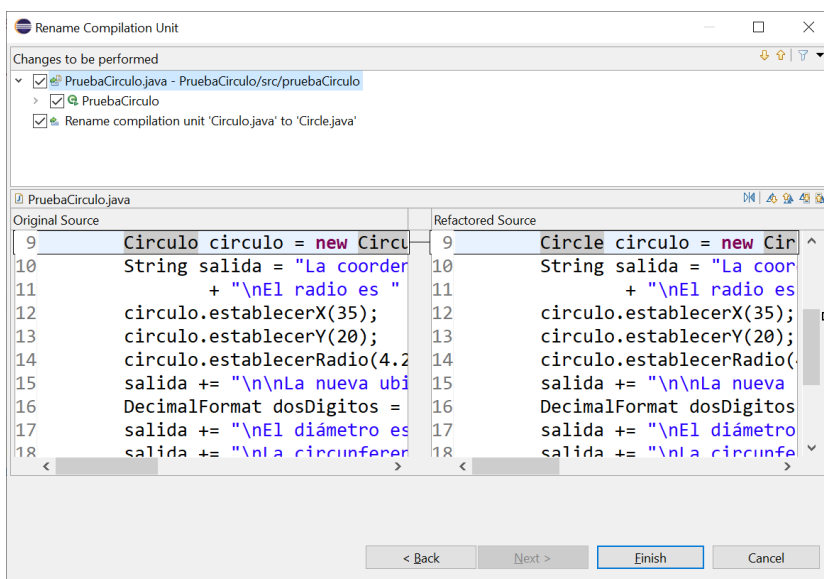
REFACTORIZACIÓN



Introducimos el nuevo nombre:




Se mostrarán todos los cambios comparando la versión anterior (a la izquierda) con la nueva versión (derecha)



REFACTORIZACIÓN

Tras renombrar Circulo por Circle, podemos observar que se ha modificado tanto la clase Circulo como todas sus referencias en la clase PruebaCirculo.



```
Circle.java
1 package pruebaCirculo;
2
3 public class Circle {
4     private int x;
5     private int y;
6     private double radio;
7
8     // constructor
9     public Circle() {
10    }
11
12    // constructor
13    public Circle(int valorX, int valorY, double
14        x = valorX;
15        y = valorY;
16        establecerRadio(valorRadio);
17    }

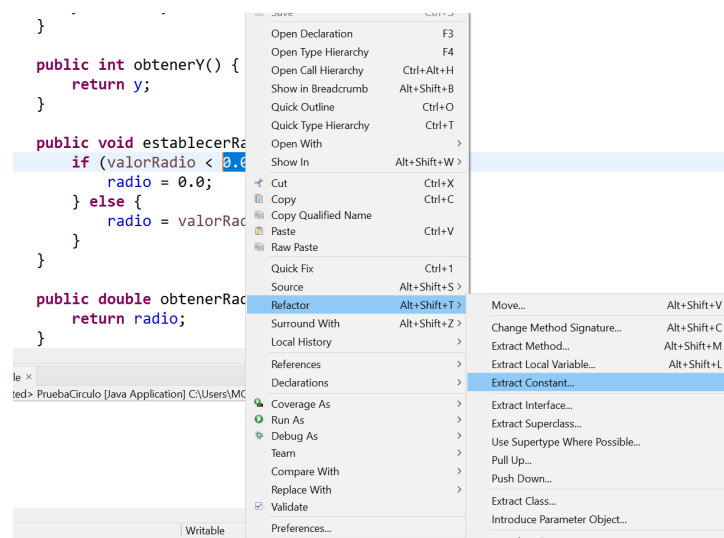
```

```
PruebaCirculo.java
1 package pruebaCirculo;
2
3
4 import java.text.DecimalFormat;
5 import javax.swing.JOptionPane;
6
7 public class PruebaCirculo {
8     public static void main(String[] args) {
9         Circle circulo = new Circle(37, 43, 2.5);
10        String salida = "La coordenada X es " + circulo.obt
11            + "\nEl radio es " + circulo.obtenerRadio()
12        circulo.establecerX(35);
13        circulo.establecerY(20);
14        circulo.establecerRadio(4.25);
15        salida += "\n\nLa nueva ubicación y el radio del ci
16        DecimalFormat dosDigitos = new DecimalFormat("0.00"
17        salida += "\nEl radio es " + dosDigitos.format(circulo.obtenerRadio());
18    }

```

2. **Introducir CONSTANTE:** nos permite modificar el valor asignado a una variable de un método por una constante.

Botón derecho en el valor=> Refactor=> Extract constant...



```

}

public int obtenerY() {
    return y;
}

public void establecerRadio(int valorRadio) {
    if (valorRadio < 0.0) {
        radio = 0.0;
    } else {
        radio = valorRadio;
    }
}

public double obtenerRadio() {
    return radio;
}

```

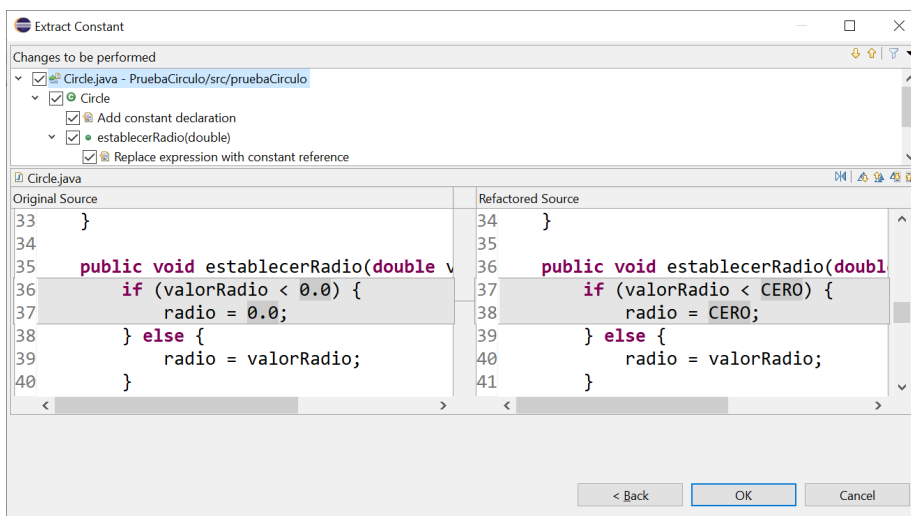
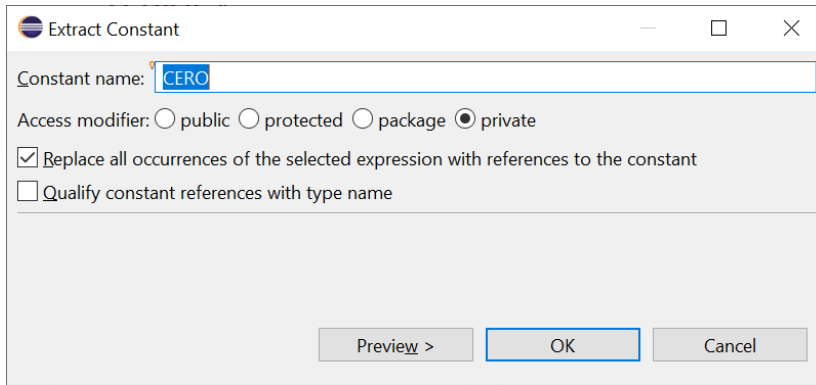
Right-click context menu options:

- Open Declaration (F3)
- Open Type Hierarchy (F4)
- Open Call Hierarchy (Ctrl+Alt+H)
- Show in Breadcrumb (Alt+Shift+B)
- Quick Outline (Ctrl+O)
- Quick Type Hierarchy (Ctrl+T)
- Open With
- Show In (Alt+Shift+W)
- Cut (Ctrl+X)
- Copy (Ctrl+C)
- Copy Qualified Name
- Paste (Ctrl+V)
- Raw Paste
- Quick Fix (Ctrl+I)
- Source (Alt+Shift+S)
- Refactor (Alt+Shift+T)
- Surround With (Alt+Shift+Z)
- Local History
- References
- Declarations
- Coverage As
- Run As
- Debug As
- Team
- Compare With
- Replace With
- Validate
- Preferences...

Refactor submenu options:

- Move... (Alt+Shift+V)
- Change Method Signature... (Alt+Shift+C)
- Extract Method... (Alt+Shift+M)
- Extract Local Variable... (Alt+Shift+L)
- Extract Constant... (selected)
- Extract Interface...
- Extract Superclass...
- Use Supertype Where Possible...
- Pull Up...
- Push Down...
- Extract Class...
- Introduce Parameter Object...
- Introduce Parameter

REFACTORIZACIÓN



```
2  
3 public class Circle {  
4     private static final double CERO = 0.0;  
5     private int x;  
  
36 public void establecerRadio(double valorRadio) {  
37     if (valorRadio < CERO) {  
38         radio = CERO;  
39     } else {  
40         radio = valorRadio;  
41     }  
42 }
```

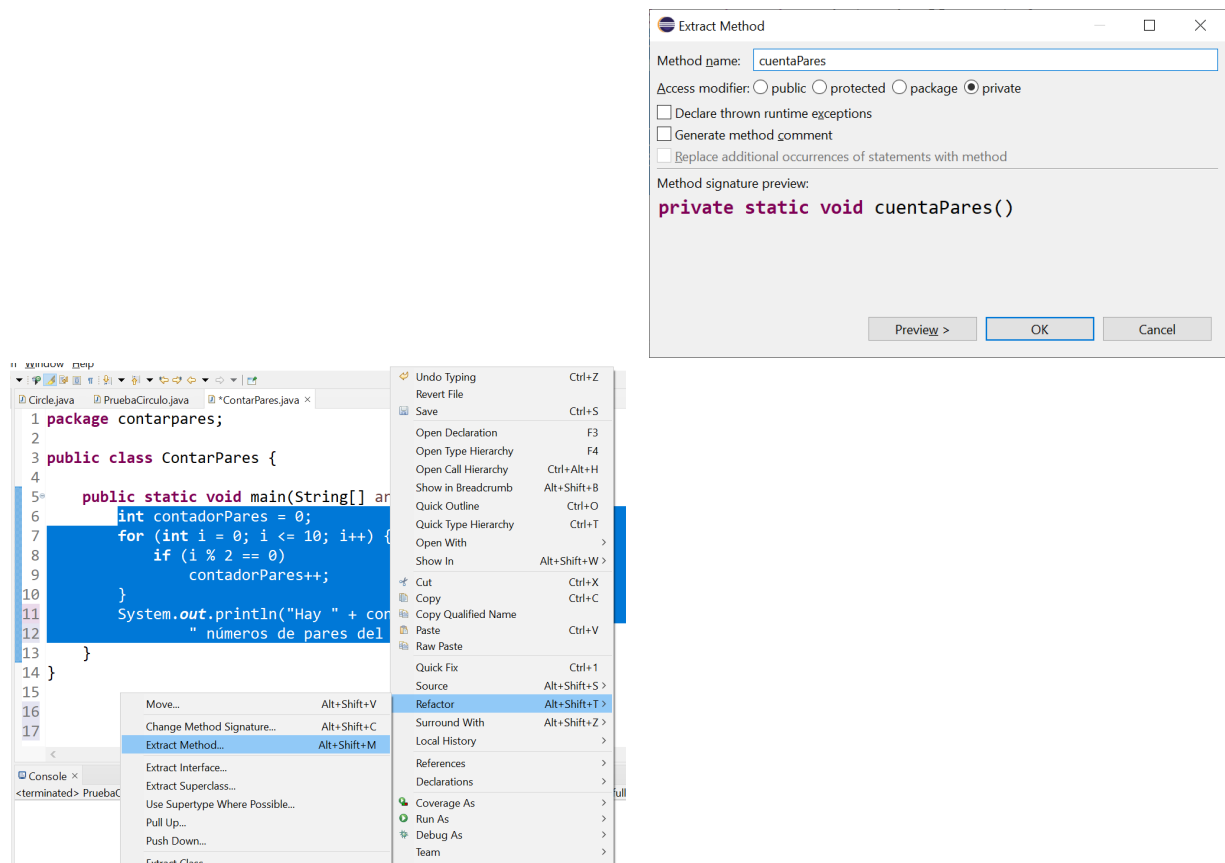
3. **Sustituir bloques de código por un método:** Este patrón nos aconseja sustituir un bloque de código, por un método. De esta forma, cada vez que queramos acceder a ese bloque de código, bastaría con invocar al método.

REFACTORIZACIÓN

```
1 package contarpares;
2
3 public class ContarPares {
4
5     public static void main(String[] args) {
6         int contadorPares = 0;
7         for (int i = 0; i <= 10; i++) {
8             if (i % 2 == 0)
9                 contadorPares++;
10        }
11        System.out.println("Hay " + contadorPares +
12            " números de pares del 0 al 10");
13    }
14 }
```

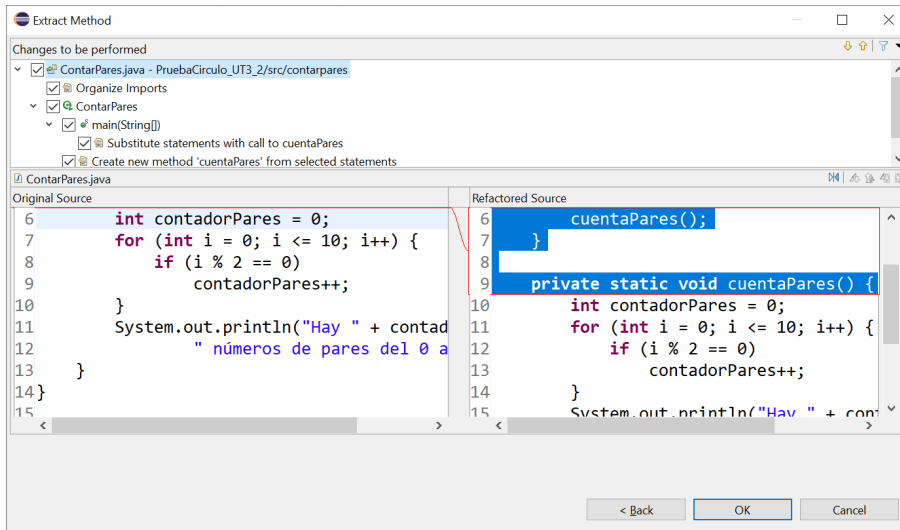
Para introducir método debemos hacer lo siguiente:

Si queremos que contarpares=0 y el bucle for aparezcan en un método los seleccionamos el bloque de código y pulsamos con el botón derecho refactor extract method, pide un nombre para el método y se crea automáticamente tanto el método como la llamada a la función.



Podemos previsualizar como quedará el cambio:

REFACTORIZACIÓN

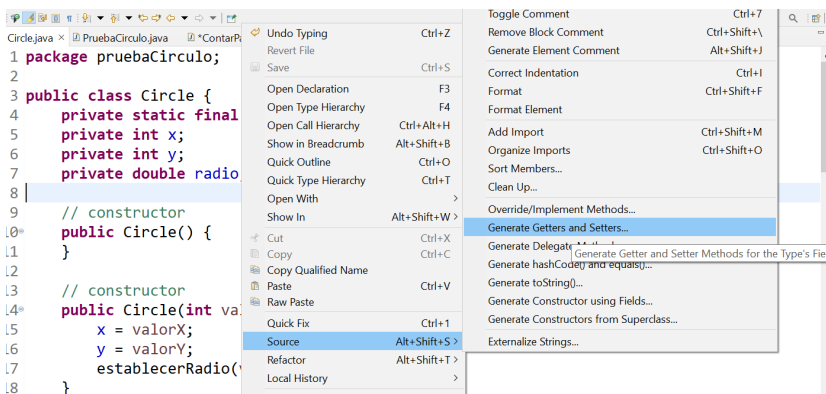


El resultado es el siguiente:

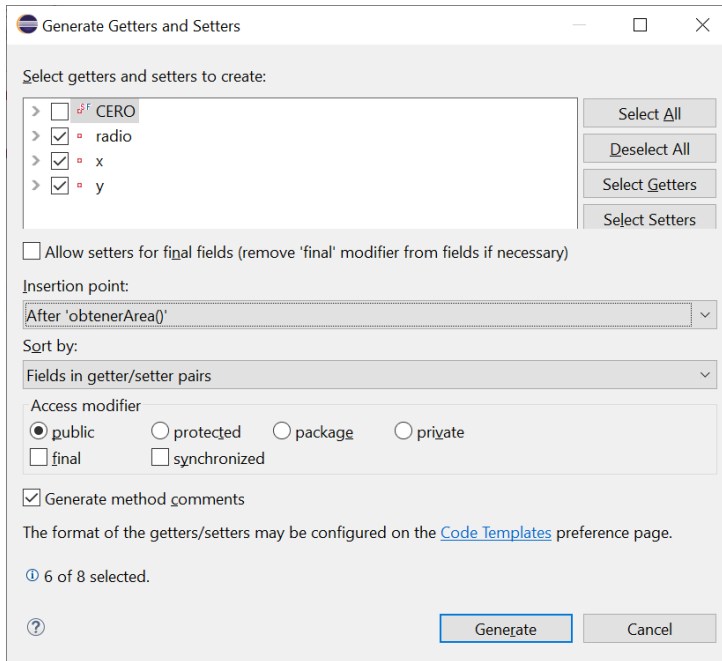
```
1 package contarpares;
2
3 public class ContarPares {
4
5     public static void main(String[] args) {
6         cuentaPares();
7     }
8
9     private static void cuentaPares() {
10         int contadorPares = 0;
11         for (int i = 0; i <= 10; i++) {
12             if (i % 2 == 0)
13                 contadorPares++;
14         }
15         System.out.println("Hay " + contadorPares + " números de pares del 0 al 10");
16     }
17 }
18 }
```

4. **Campos encapsulados:** Se aconseja crear métodos getter y setter, (de asignación y de consulta) para cada campo que se defina en una clase. Cuando sea necesario acceder o modificar el valor de un campo, basta con invocar al método getter o setter según convenga.

Para ello se pulsa botón derecho sobre la clase `source` `Generate Setters and Getters`.



REFACTORIZACIÓN



Y pasamos a obtener:

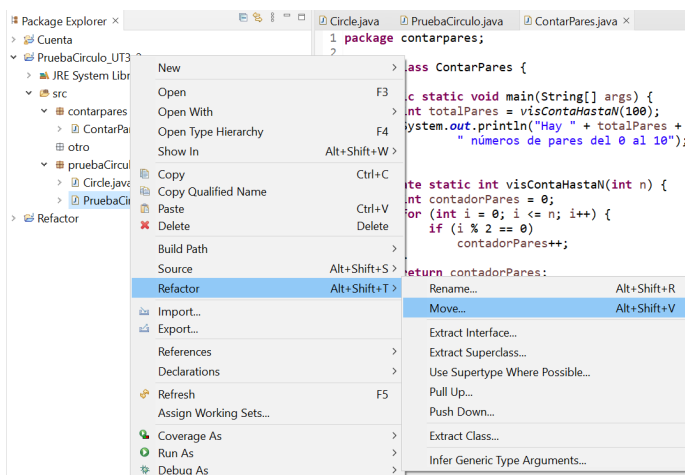
```
/**
 * @return the x
 */
public int getX() {
    return x;
}
/**
 * @param x the x to set
 */
public void setX(int x) {
    this.x = x;
}
/**
 * @return the y
 */
public int getY() {
    return y;
}
/**
 * @param y the y to set
 */
public void setY(int y) {
    this.y = y;
}
/**
 * @return the radio
 */
public double getRadio() {
    return radio;
}
/**
 * @param radio the radio to set
 */
public void setRadio(double radio) {
    this.radio = radio;
}
```

REFACTORIZACIÓN

Ahora podemos eliminar los métodos establecer/obtener que hacen lo mismo que los nuevos set/get, si hay alguna llamada a algún establecer/obtener se cambiará el nombre mediante refactorización y en la clase de prueba, modificar los establecer por set y los obtener por get, mediante botón derecho en la llamada refactor rename

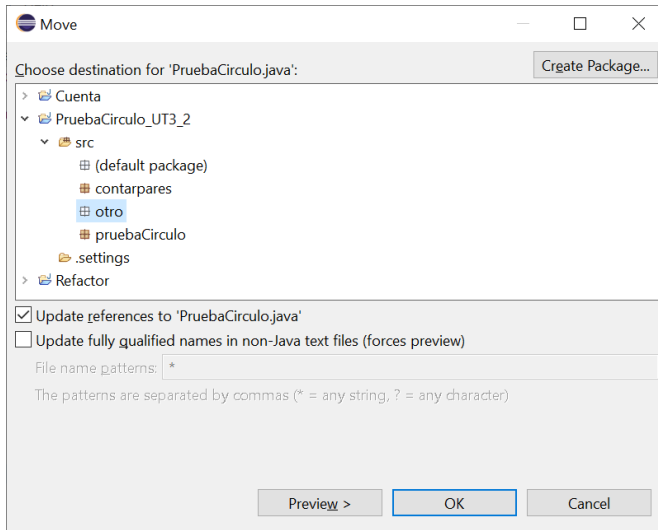
```
1 package pruebaCirculo;
2
3 import java.text.DecimalFormat;
4
5
6 public class PruebaCirculo {
7     public static void main(String[] args) {
8         Circle circulo = new Circle(37, 43, 2.5);
9         String salida = "La coordenada X es " + circulo.getX() + "\nLa coordenada Y es " + circulo.getY()
10             + "\nEl radio es " + circulo.getRadio();
11         circulo.setX(35);
12         circulo.setY(20);
13         circulo.setRadio(4.25);
14         salida += "\n\nLa nueva ubicación y el radio del círculo son\n" + circulo.toString();
15         DecimalFormat dosDigitos = new DecimalFormat("0.00");
16         salida += "\nEl diámetro es " + dosDigitos.format(circulo.obtenerDiametro());
17         salida += "\nLa circunferencia es " + dosDigitos.format(circulo.obtenerCircunferencia());
18         salida += "\nEl área es " + dosDigitos.format(circulo.obtenerArea());
19         JOptionPane.showMessageDialog(null, salida);
20         System.exit(0);
21     } // fin de main
22 } // fin de la clase PruebaCirculo
```

5. **Mover la clase:** Si es necesario, se puede mover una clase de un paquete a otro, o de un proyecto a otro. La idea es no duplicar código que ya se haya generado. Esto impone la actualización en todo el código fuente de las referencias a la clase en su nueva localización

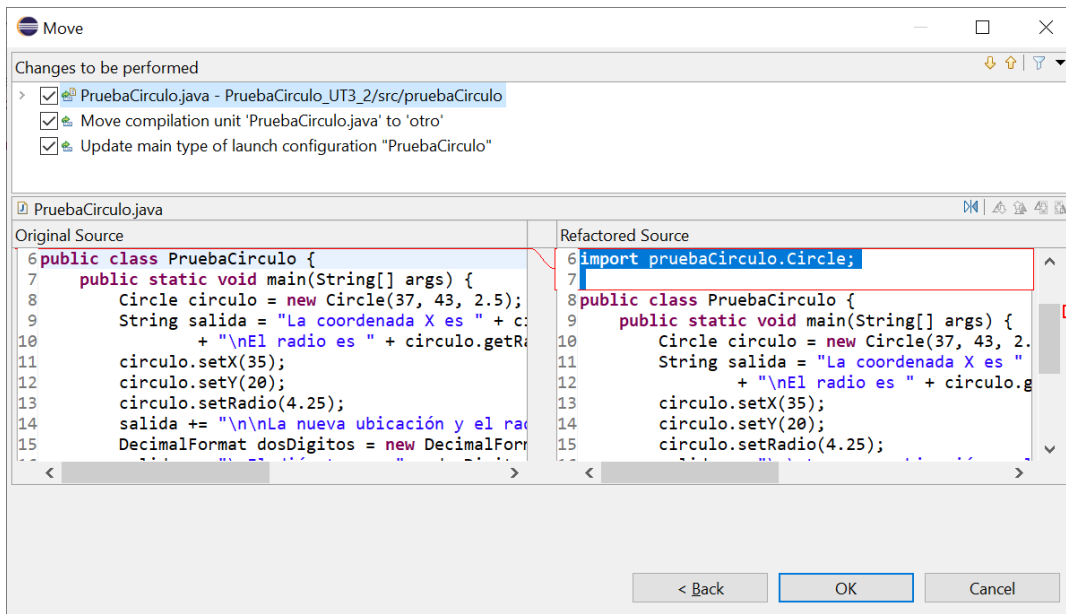


Deberemos elegir a donde queremos mover la clase dentro de nuestro workspace:

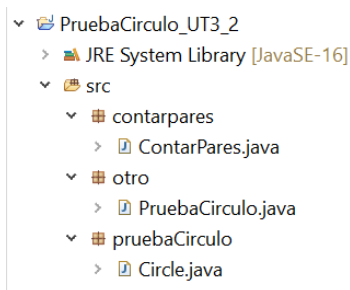
REFACTORIZACIÓN



Podemos previsualizar los cambios que se realizarán:



Si cambio la clase PruebaCirculo.java al paquete otro, quedará de la siguiente forma:

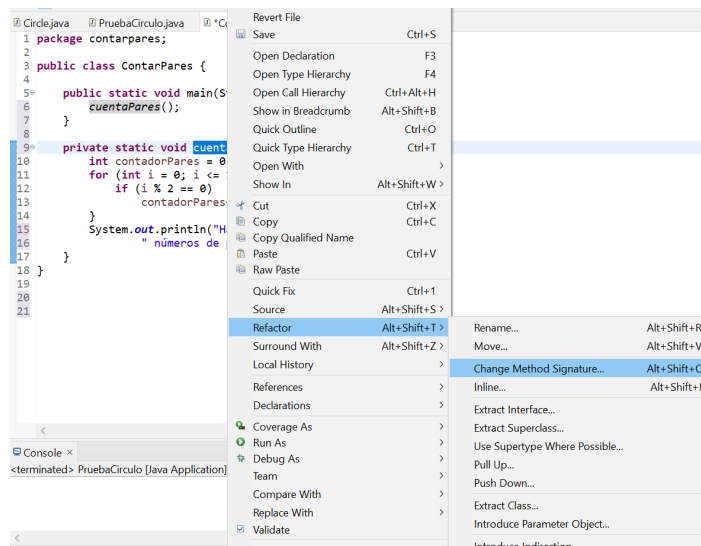


REFACTORIZACIÓN

6. **Change Method signature.** Este método permite cambiar la firma de un método. Es decir, el nombre del método y los parámetros que tiene. De forma automática se actualizarán todas las dependencias y llamadas al método dentro del proyecto.

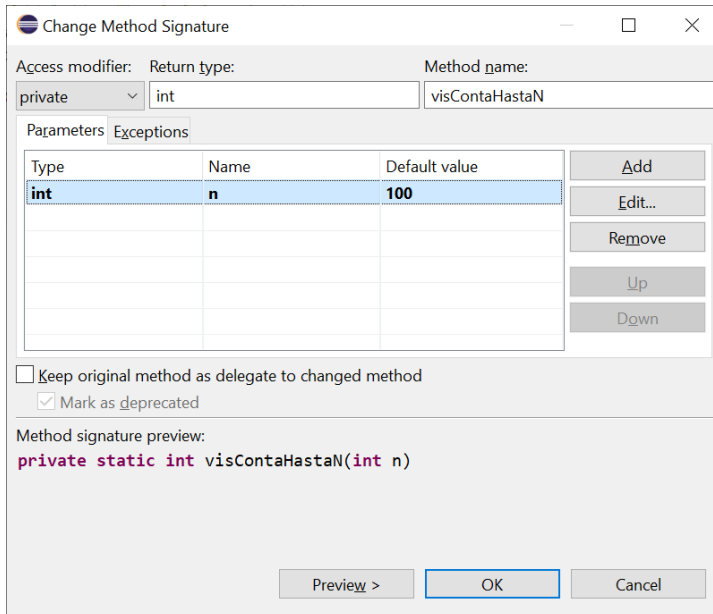
Cambiamos el método `cuentaPares` por `visContaHastaN(n)`

```
3 public class ContarPares {
4
5     public static void main(String[] args) {
6         cuentaPares();
7     }
8
9     private static void cuentaPares() {
10        int contadorPares = 0;
11        for (int i = 0; i <= 10; i++) {
12            if (i % 2 == 0)
13                contadorPares++;
14        }
15        System.out.println("Hay " + contadorPares +
16                           " números de pares del 0 al 10");
17    }
18 }
```



En la siguiente ventana indicamos los cambios que queremos realizar a nuestro método:

REFACTORIZACIÓN



El resultado es el siguiente, deberemos modificar el código para que devuelva la cantidad de pares, y modificar el bucle manualmente, pero podemos observar que se ha modificado su llamada.

```
3 public class ContarPares {
4
5     public static void main(String[] args) {
6         visContaHastaN(100);
7     }
8
9     private static int visContaHastaN(int n) {
10        int contadorPares = 0;
11        for (int i = 0; i <= 10; i++) {
12            if (i % 2 == 0)
13                contadorPares++;
14        }
15        System.out.println("Hay " + contadorPares +
16                           " números de pares del 0 al 10");
17    }
18 }
```

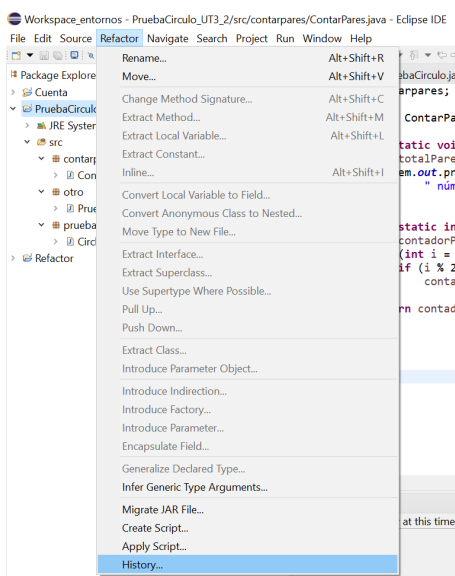
Podrá quedar similar al siguiente código:

REFACTORIZACIÓN

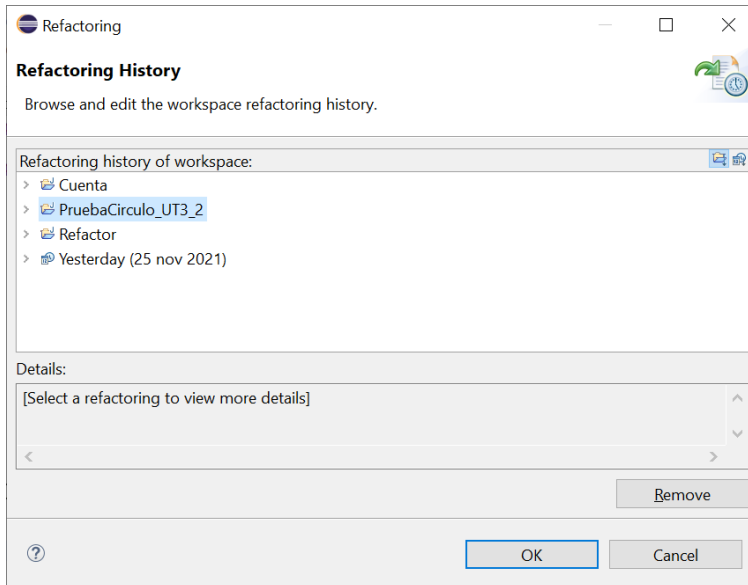
```
3 public class ContarPares {
4
5     public static void main(String[] args) {
6         int totalPares = visContaHastaN(100);
7         System.out.println("Hay " + totalPares +
8             " números de pares del 0 al 10");
9     }
10
11     private static int visContaHastaN(int n) {
12         int contadorPares = 0;
13         for (int i = 0; i <= n; i++) {
14             if (i % 2 == 0)
15                 contadorPares++;
16         }
17         return contadorPares;
18     }
19 }
```

7. **Extract Interface.** Este método permite escoger los métodos de una clase para crear una interface (plantilla que define los métodos acerca de lo que puede hacer o no una clase, pero no los desarrolla. Las clases que implementan la Interface son las que desarrollan los métodos).
8. **Extract Superclass.** Permite extraer una superclase.

Eclipse permite visualizar el histórico de refactorizaciones realizadas sobre un proyecto. Para ello se abre el menú refactor History



REFACTORIZACIÓN



También permite crear un script con todos los cambios realizados y guardarlo en un fichero .XML (menú refactor create script).