

# 1. Refactorización

## 1.1 Introducción

La ingeniería de software ha ido cambiando y creando nuevas técnicas que mejoran la calidad del software y sobre todo que lo hacen más adaptable y reutilizable. Un ejemplo es la técnica de programación orientada a objetos.

Los expertos vieron que ciertos problemas de diseño y código se repiten y que sería conveniente formalizar y documentar la descripción de los mismos y su solución para ahorrar esfuerzos, de tal forma que esta documentación permita a otros desarrolladores aplicar la solución en base a la experiencia.

La programación orientada a objetos combinada con el uso de patrones de diseño, antipatrones de diseño y refactorización logra crear un código más adaptable y reutilizable.

Un patrón de diseño documenta el conocimiento relacionado con soluciones exitosas.

Refactorizar es "modificar la estructura interna del software con el fin de facilitar su entendimiento y capacidad de ser modificado en el futuro, de tal forma que no afecte al comportamiento observable del software cuando se ejecuta" (Fowler).

## 1.2 Patrón de diseño

Un patrón de diseño es una forma estandarizada de solución de diseño de un problema recurrente encontrado en el diseño de software orientado a objetos, probado con éxito en el pasado y bien documentado.

- Es una descripción de cómo resolver un problema de diseño, que se puede utilizar en diversas situaciones pero debe adaptarse al contexto del problema actual.
- Es una herramienta para el desarrollador pero por sí sola no garantiza nada, ya que no es un esquema rígido a seguir, necesita del desarrollador y su creatividad.
- No es un diseño terminado que se pueda pasar directamente al código.
- No es un principio abstracto, una teoría o una idea no comprobada.
- No es una solución de una sola vez.
- Hay patrones de diseño en muchas áreas entre las que se encuentra OOP.

### Objetivos

- Evita repetir la búsqueda de soluciones a problemas que otros diseñadores ya han hecho y resuelto.
- Crear catálogos de patrones.
- Crear un vocabulario común entre los diseñadores.
- Estandarizar el diseño pero sin eliminar la creatividad del desarrollador.
- Facilitar la transmisión de conocimientos entre generaciones de diseñadores.

## Clasificación

Una posible clasificación de patrones en general puede ser:

- Patrones de Arquitectura: Enfocados en la arquitectura del sistema. Proviene de conjuntos predefinidos de subsistemas y cómo se relacionan.
- Patrones de Diseño: Centrados en el diseño.
- Patrones de codificación o dialectos (lenguajes) que ayudan a implementar aspectos particulares del diseño en un lenguaje de programación específico.
- Patrones de interacción enfocados al diseño de interfaces web.

## 1.3 Catálogo de patrones de diseño.

El catálogo principal de patrones de diseño se encuentra en el libro Design Patterns escrito por Gang of Four (GoF) compuesto por Erich Gamma, Richard Helm, Ralph Johnson y John Vlissides, referenciado como Gamma et al. 1995 en el que se recopilaron 23 patrones de diseño comunes, agrupados en tres grupos:

- patrones de creación para el proceso de creación de instancias de objetos.
- patrones estructurales para combinar clases y objetos para formar estructuras más grandes.
- patrones de comportamiento para organizar el flujo de control dentro del sistema.

El grupo GoF describe un patrón usando una plantilla con:

- Nombre generalmente en inglés.
- Nombres alternativos.
- Clasificación del patrón: creacional, estructural o conductual.
- Descripción del problema que el patrón pretende resolver.
- Razones por las que se debe aplicar el patrón.
- Estructura y descripción de las clases y entidades que participan en el patrón así como las relaciones entre ellas.
- Ventajas y desventajas de aplicar el patrón.
- Implementación. Muestra cómo aplicar el patrón.
- Ejemplo de código fuente.
- Usos conocidos en sistemas reales.
- Relación con otros patrones

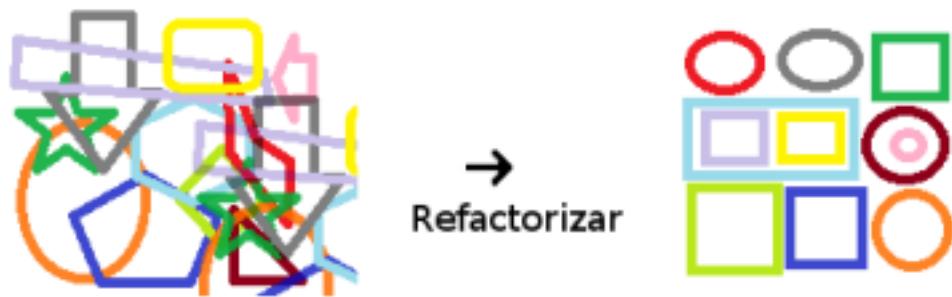
## 1.4 Refactorización

Cuando un programador comienza un nuevo trabajo de programación, seguramente producirá código limpio y bien organizado siguiendo patrones de diseño, pero a medida que ese código se mantiene, incluso si funciona correctamente, el código se vuelve cada vez más ilegible y puede ser necesario reorganizarlo.

El mantenimiento del código puede ser complicado si:

- Participan diferentes programadores.
- Se agregan nuevas características. Se complica aún más si para incorporar nuevas funcionalidades se incorporan piezas de código adaptadas de otros programas.

Cuanto más antiguo y grande sea el código, más obvia será la necesidad de reorganizar.

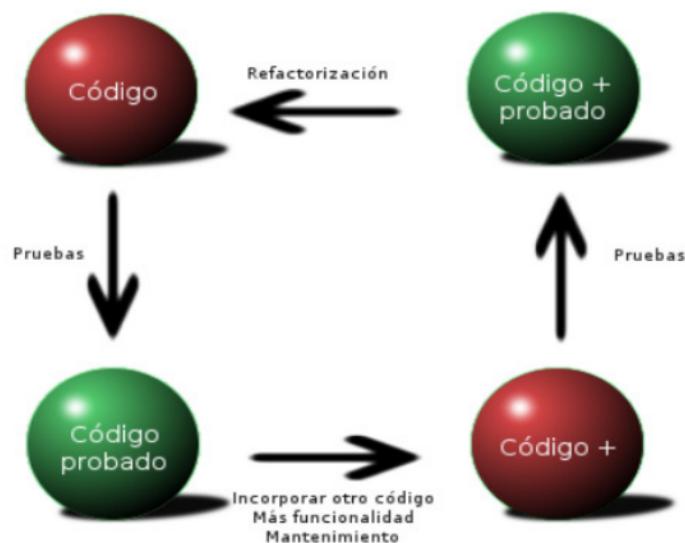


La refactorización es la transformación controlada del código fuente que asegura la reorganización del código para mejorar su mantenibilidad y/o hacerlo más comprensible.

- No altera su comportamiento externo pero mejora su estructura interna.
- Te permite tomar diseños defectuosos, con código que no sigue patrones de diseño, por ejemplo con duplicaciones innecesarias, y adaptarlo para lograr uno bueno y bien organizado.

## Refactorización y pruebas

La referencia clásica para la refactorización es el libro **Refactoring** de Martin Fowler de 1999. Los desarrolladores alternan la inserción de nuevas funciones y pruebas con la refactorización.



El proceso de cambio consiste en:

- Probar automáticamente el programa con la nueva funcionalidad a través de un lote de casos de prueba que validan su correcto funcionamiento.
- Analizar las refactorizaciones a realizar.
- Se recomienda realizar los cambios de forma progresiva, uno a uno y paso a paso.
- Repetir la prueba automática obteniendo los mismos resultados que antes de la refactorización.
- Si es necesario, agregar nuevas pruebas.

Es fundamental que los cambios en la refactorización se realicen uno por uno y paso a paso. Por ejemplo, si un método es muy grande y desea dividirlo en métodos más pequeños, la refactorización manual consistiría en:

- Crear los nuevos métodos vacíos. Ejecutar pruebas. Puede haber problemas con la definición de los nuevos métodos.
- Escribir el código para los nuevos métodos. El método original todavía existe. Nadie llama a los nuevos métodos.
- Posiblemente se tendrían que crear nuevas pruebas unitarias para los nuevos métodos. Ejecutar pruebas con los nuevos métodos.
- Reemplazar en el método original, las piezas de código correspondientes con llamadas a los nuevos métodos uno por uno si es posible. Ejecutar pruebas.

Las pruebas unitarias permiten comprobar que el código funciona correctamente, pero también tienen las siguientes ventajas (Massol y Husted, 2003):

- Facilitan las pruebas de regresión, es decir, pruebas para detectar un error que aparece tras un cambio en el código.
- Limitan las pruebas de depuración(debug) de código.
- Brindan confianza a la hora de modificar el código ya que pueden ejecutarse nuevamente y verificar que la aplicación de un cambio fue exitosa.
- Facilitan la refactorización.
- Es la primera prueba a la que se somete el código fuente. Si el código no es fácil de probar mediante el uso de tests unitarios es una señal de que algo no está bien y que tal vez se necesita reorganizar el código.

Las pruebas unitarias tienen las ventajas anteriores, pero también tienen un costo asociado ya que deben mantenerse a medida que se modifica el código fuente. Esto significa que si se elimina una funcionalidad del sistema, se deben eliminar los casos de prueba para esa funcionalidad, si se agrega una funcionalidad, se deben agregar nuevos casos y, si se modifica alguna funcionalidad existente, las pruebas unitarias se deben actualizar de acuerdo con el cambio hecho.

Las ventajas se incrementan si las pruebas se realizan de forma automática, ya que entonces se dispone de un sistema de pruebas automáticas que se puede utilizar en cualquier momento. Estas pruebas son fundamentales para la refactorización ya que sin ellas sería una **operación de alto riesgo**.

## Ventajas de refactorizar

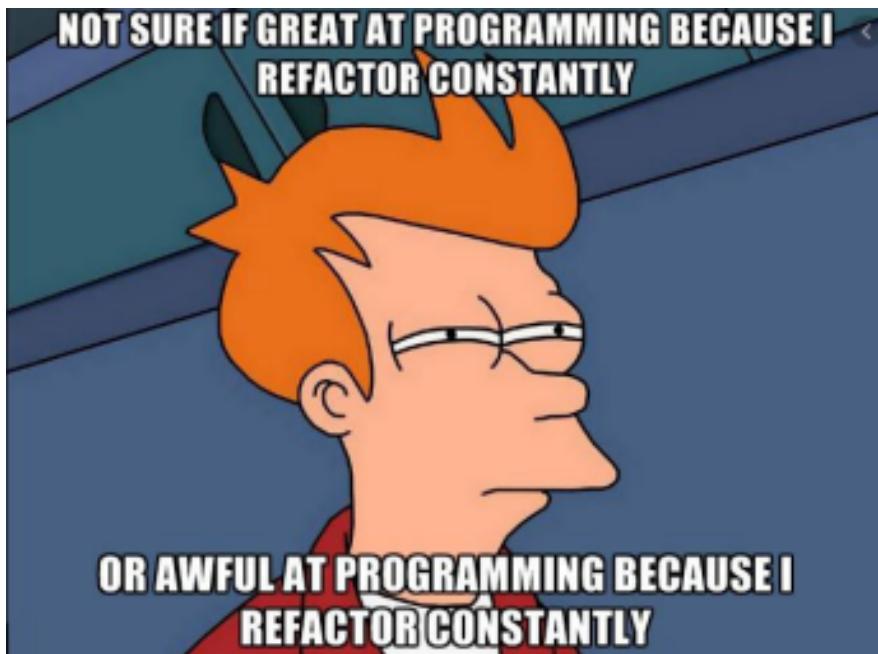
Tanto Fowler como Piattini y García creen que la refactorización aumenta la productividad porque:

- Facilita la comprensión del código fuente, principalmente para desarrolladores que no han estado involucrados desde el comienzo del desarrollo, ya que hace que sea más fácil de leer, expresa más claramente cuáles son sus funciones y es lo más autodocumentable posible.
- Reduce los errores ya que comprender el código también permite detectarlos más fácilmente.
- Permite una programación más rápida, ya que permite mejorar los diseños base.
- Facilita los cambios.

## Limitaciones de la refactorización

Tanto Fowler como Piattini y García consideran que las áreas conflictivas para la refactorización son las bases de datos y los cambios en interfaces.

- Es muy costoso aplicar cambios que están vinculados con bases de datos, ya que si estos implican cambios en el esquema de la base de datos, habría que aplicar los cambios y luego migrar los datos.
- El cambio en una interfaz no tiene demasiada complejidad si se tiene acceso y se puede modificar el código fuente de todos los clientes de la interfaz a refactorizar. En cambio, sí es problemático cuando esto se convierte en lo que (Fowler et al, 1999) llama una interfaz publicada, ya en este caso no es posible modificar el código fuente de sus clientes.



## Malos olores en el código (bad smells)

Tanto Fowler como Piattini y García describen los malos olores como los síntomas que indican que se debe de refactorizar. Algunos de ellos son:

- Código duplicado. Es la principal razón para refactorizar. Si se detecta el mismo código en más de un lugar, se debe buscar la manera de extraerlo y unificarlo.
- Método largo. Los métodos cortos son más fáciles de reutilizar.
- Clase grande. Si una clase tiene demasiados métodos públicos, puede ser conveniente dividirla.
- Lista de parámetros extensa. Los métodos con muchos parámetros son difíciles de entender.
- Cambio divergente (Divergent change). Se presenta cuando al realizar el cambio 'A' en una clase, se deben modificar algunos métodos y al realizar el cambio 'B' se deben modificar otros métodos. Se debería separar la clase original en varias, de modo que un cambio afecte a una sola de las nuevas clases. Este caso es el contrario al siguiente.
- Cirugía de escopeta (Shotgun surgery). Lo contrario de la anterior. Este síntoma ocurre cuando un cambio en un determinado lugar, obliga a hacer otros en varios lugares. La solución consiste en reunir en una única clase los cambios que se propagan a otras clases.

- Envidia de funcionalidad (Feature envy). Ocurre cuando un método usa más elementos de otra clase que de la propia. El problema generalmente se resuelve pasando el método a la clase cuyos componentes son requeridos.
- Clase de datos (Data class). Ocurre en clases que solo tienen atributos y métodos públicos para acceder a ellos ("get" y "set"). Este tipo de clases deben ser cuestionadas ya que no suelen tener ningún comportamiento.
- Legado rechazado (refused bequest). Ocurre cuando hay subclases que usan pocas características de sus superclases. Si las subclases no requieren todo lo que sus superclases les proporcionan a través de la herencia, generalmente indica que la jerarquía de clases no es correcta.

Detectar malos olores en el código es difícil. A veces incluso puede parecer que contradice algunos patrones de diseño. Por ejemplo, el hedor de la "envidia de la funcionalidad" puede parecer contradictorio con el patrón de "visitante" cuya implementación permite crear una clase externa para actuar sobre los datos de otra clase.

## Cuándo no refactorizar

- Cuando el código no funciona, la refactorización no es la solución, ya que no soluciona los errores. Puede ayudar a que el código sea más comprensible y, por lo tanto, puede ayudar a encontrar el problema, pero no soluciona el problema. El código refactorizado que no funciona puede agregar nuevos problemas al código porque no existe una prueba confiable para garantizar que el código aún tenga la misma funcionalidad después de la refactorización.
- Cuando se necesitan nuevas características, ya que la refactorización no agrega funcionalidad.
- Cuando la fecha de entrega del software está cerca.

## Patrones de refactorización

Ejemplos:

- Clases: extraer clase, extraer subclase, extraer superclase, mover, copiar, ....
- Campos: ascender, descender, ....
- Métodos: Agregar parámetros, eliminar parámetros, extraer método, cambiar modificador de acceso, ascender, descender, renombrar, reemplazar condicional con polimorfismo, reemplazar código de error con Excepción, ....
- Interfaces: extraer, ....

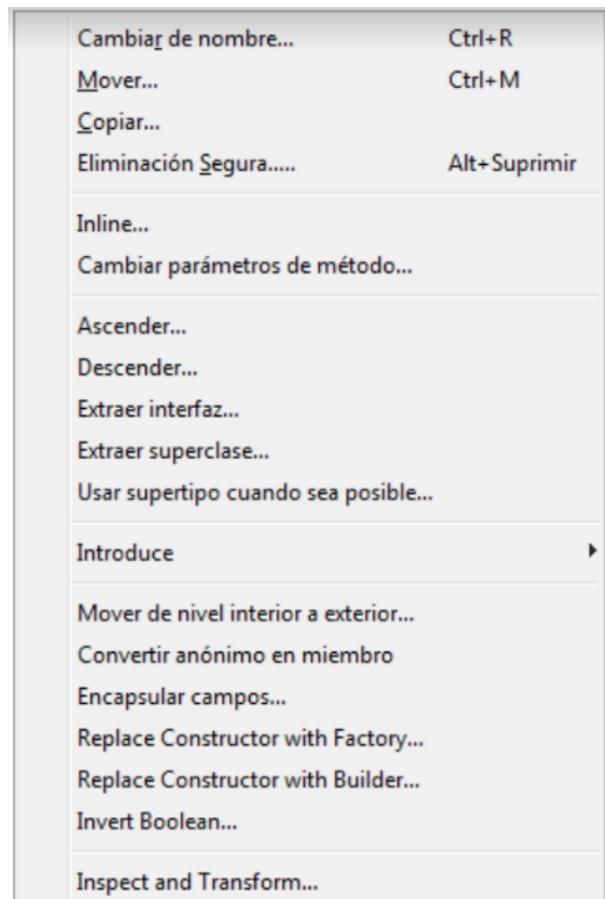
# 1.5 Refactorizar código Java con NetBeans

## Introducción

NetBeans permite **Refactorizar** código Java sin cambiar el comportamiento del programa. La refactorización es una operación que conserva la funcionalidad del código original y permite:

- Hacer que el código sea más fácil de leer y comprender.
- Facilitar la adición de nuevas funciones.
- Eliminar repeticiones de código innecesarias.
- Permitir usar el código para necesidades más generales.
- Mejorar el rendimiento del código.

NetBeans ayuda con la **refactorización** del código Java al mostrar las partes del software que se verán afectadas, lo que le permite incluir o excluir dichos cambios. Por ejemplo, si la Refactorización implica cambiar el nombre de una clase, NetBeans encontrará ocurrencias de ese nombre en el código y ofrecerá la posibilidad de cambiarlo; si consiste en cambiar la estructura del código, actualizará el resto del código para reflejar ese cambio en la estructura.



## Opciones Refactor

Las posibles refactorizaciones que se pueden realizar son accesibles desde la opción Refactor en el menú principal o seleccionando un elemento de código, haciendo clic derecho y eligiendo Refactor. Esas **refactorizaciones** son:

Refactorización	Descripción
Cambiar de nombre	Cambia el nombre de una clase, interfaz, variable o método a algo más significativo y actualiza todo el código fuente del proyecto para reflejar este cambio.
Introducir variable, constante, campo o método	El programador selecciona un fragmento de código, genera una declaración basada en el código seleccionado y reemplaza el bloque de código con una llamada a esa declaración. La declaración puede ser la creación de una variable, constante, campo o método
Cambiar parámetros de un método	Agregar, eliminar, modificar o cambiar el orden de los parámetros de un método, o cambiar el modificador de acceso ( <i>público</i> , <i>privado</i> , <i>protegido</i> )

Encapsular campos	Generar métodos get y set para un campo y, opcionalmente, actualiza todas las referencias a ese campo utilizando los métodos get y set
Ascender	Mover métodos y campos a una superclase de la que hereda la clase actual
Descender	Mover clases, métodos y campos internos a una subclase de la clase actual
Mover clase	Mover una clase a otro paquete o dentro de otra clase. Además, todo el código fuente del proyecto se actualiza para hacer referencia a la clase en el nuevo paquete.
Copiar clase	Copie una clase en el mismo paquete o en uno diferente
Mover de nivel interior a exterior	Mueve una clase miembro hacia arriba un nivel en la jerarquía de clases
Convertir anónimo en miembro	Convierte una clase anónima en una clase miembro que contiene un nombre y un constructor. La clase anónima se reemplaza con una llamada a la clase miembro
Extraer interface	Crea una nueva interfaz formada a partir del método público no estático seleccionado en una clase o interfaz. Si se extrae de una clase, implementará la interfaz recién creada. Si se extrae de una interfaz, ampliará la interfaz creada.
Extraer superclase	Informa al programador de métodos y campos que se pueden mover a una superclase. El programador selecciona los que desea mover y NetBeans: <ul style="list-style-type: none"> <li>- Crea una nueva clase abstracta que contendrá dichos campos y métodos.</li> <li>- Cambia la clase actual para extender la nueva clase y mueve los métodos y campos seleccionados a la nueva clase.</li> </ul>

## Deshacer y rehacer cambios

La opción **deshacer** se activará después de hacer una refactorización y aparecerá como **Undo[nombre de la refactorización]** permitiendo deshacer todos los cambios en todos los archivos afectados por la refactorización. La opción **Rehacer** se activará después de deshacer una refactorización y aparecerá como **Redo[nombre de la refactorización]** sirviendo para rehacer los cambios realizados.

Estas opciones no tendrán efecto si alguno de los archivos afectados ha sido modificado desde que se realizó la Refactorización o cuando la Refactorización se puede deshacer simplemente con los iconos:

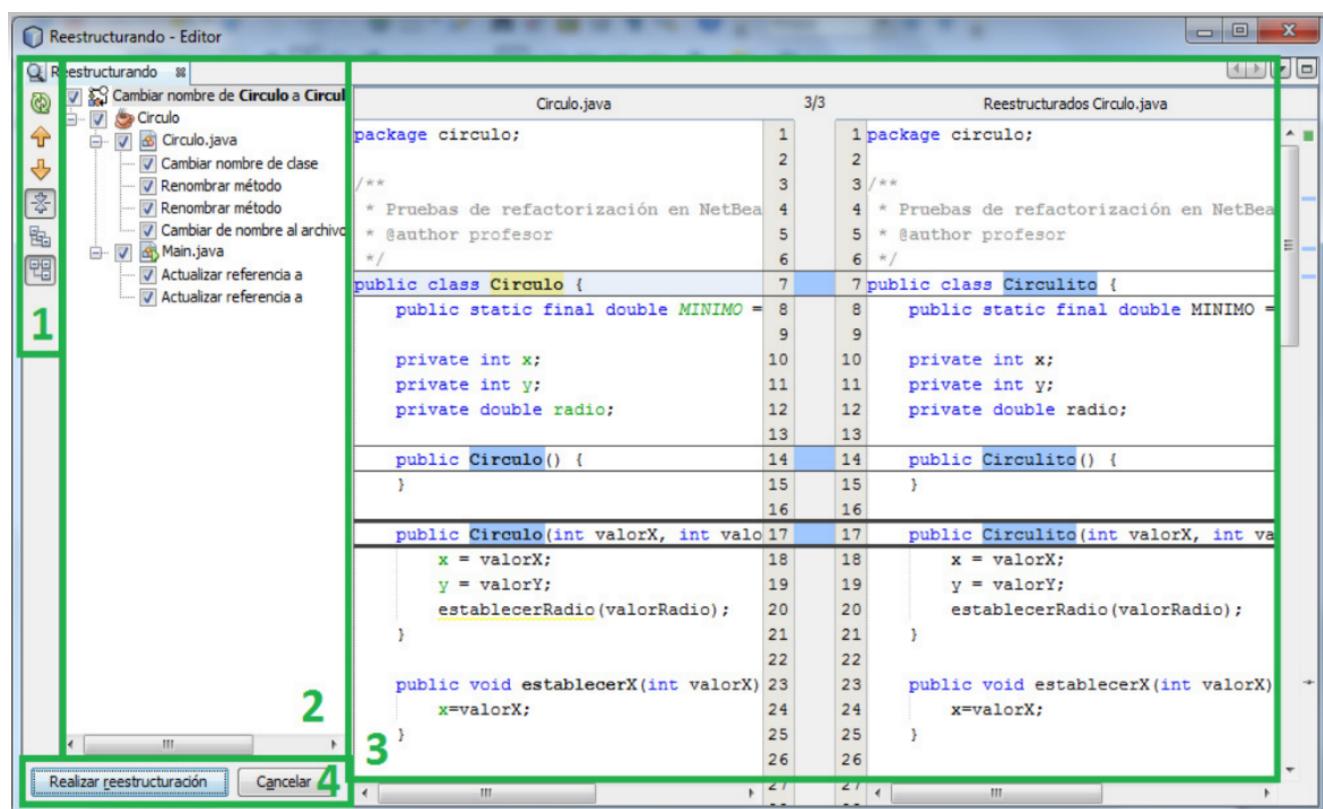


## Ventana de Refactorización (Vista previa)

Esta ventana aparece cuando en el proceso de Refactorización, elige obtener una vista previa de los cambios antes de llevar a cabo realmente la Refactorización.

Por ejemplo, para cambiar el nombre de una clase o interfaz, vaya a la ventana del proyecto o a la ventana de edición, haga clic derecho en la clase o interfaz, elija Refactorizar->Cambiar nombre..., escriba el nuevo nombre de la clase y haga clic en el botón Vista previa.

La ventana de Refactorización se divide en varias áreas:



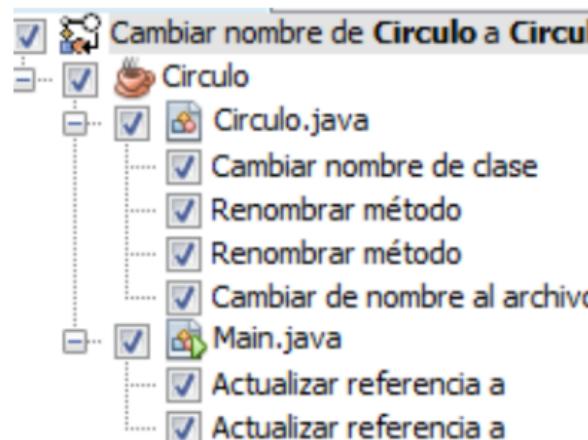
- **Zona 1**, menú de iconos para actuar sobre la vista previa de la Refactorización que permite:

Icono	Descripción
	Actualizar las zonas después de modificar las condiciones de Refactorización.
	Expandir o colapsar completamente los nodos del árbol de la zona 2
	Mostrar la vista lógica
	Mostrar la vista física
	Moverse a través de las ocurrencias.

- **Zona 2:** Muestra el número de ocurrencias de la Refactorización en el proyecto y el árbol de ocurrencias, mostrando la ocurrencia actual que es la que se detalla en la zona 3.

Se puede:

- marcar o desmarcar una ocurrencia para incluirla o excluirla de la Refactorización
- expandir o colapsar cada nodo en el árbol
- haga doble clic en un nodo para convertirlo en la ocurrencia actual y mostrarlo en la zona 3.



- **Zona 3:** muestra los detalles de la ocurrencia resaltada en la zona 2 para permitir la comparación entre el código fuente inicial (en el lado izquierdo) y lo que resultaría después de la Refactorización (en el lado derecho).

La vista de los dos archivos tiene:

- Líneas de código numeradas.
- Los cambios destacados.
- Todas las líneas de código afectadas por los cambios se resaltan entre finas líneas paralelas que conectan los dos archivos.
- La ocurrencia actual resaltada entre gruesas líneas paralelas que conectan los dos archivos.

Circulo.java	3/3	Reestructurados Circulo.java
package circulo;	1	1 package circulo;
/**	2	2
* Pruebas de refactorización en NetBea	3	3 /**
* @author profesor	4	4 * Pruebas de refactorización en NetBea
*/	5	5 * @author profesor
public class Circulo {	6	6 */
public static final double MINIMO =	7	7 public class Circulito {
private int x;	8	public static final double MINIMO =
private int y;	9	private int x;
private double radio;	10	private int y;
}	11	private double radio;
public Circulo() {	12	}
}	13	public Circulito() {
public Circulo(int valorX, int valo	14	public Circulito(int valorX, int va
x = valorX;	15	x = valorX;
y = valorY;	16	y = valorY;
establecerRadio(valorRadio);	17	establecerRadio(valorRadio);
}	18	}
public void establecerX(int valorX)	19	public void establecerX(int valorX)
x=valorX;	20	x=valorX;
}	21	}
	22	
	23	
	24	
	25	
	26	

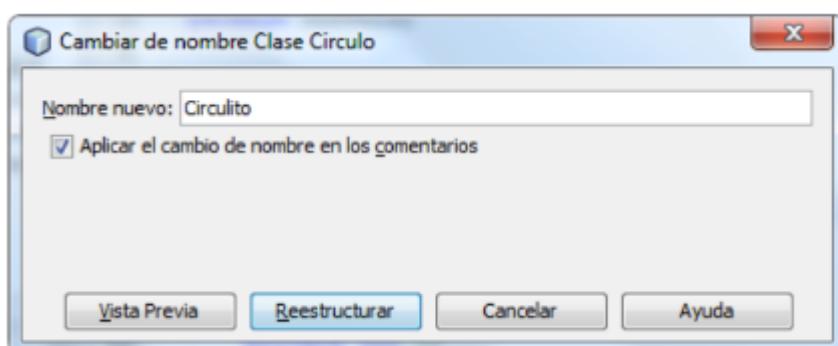
- **Zona 4:** Contiene los botones que permiten realizar la Refactorización o cancelarla.

## Casos de Refactorización

### Cambiar nombre

Para cambiar el nombre de una clase, interfaz, variable o método, debe ir a la ventana del proyecto o a la ventana de edición, hacer clic con el botón derecho en la clase o la interfaz y elegir **Refactorizar->Cambiar nombre...** Aparece un cuadro que le permite:

- Escribir el nuevo nombre
- Opcionalmente cambiar el nombre en los comentarios.
- Tener una vista previa de los cambios
- Llevar a cabo la Refactorización
- Cancelar la operación de Refactorización
- Consulte la ayuda en línea de NetBeans



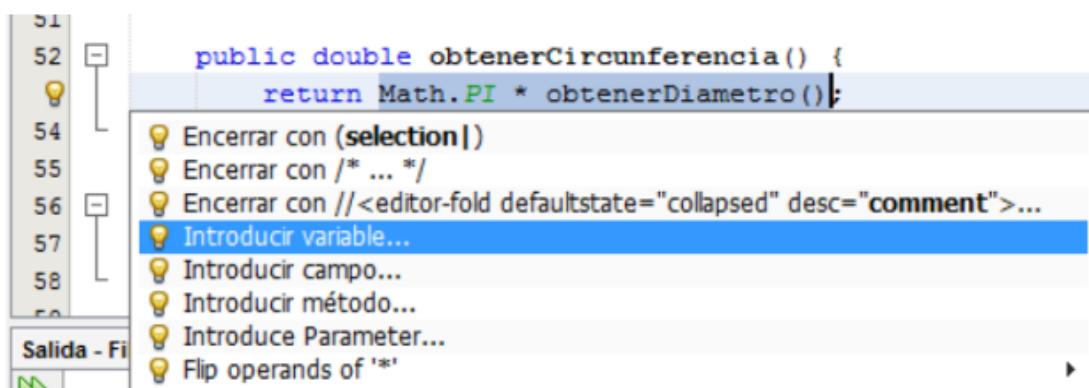
Cambiará todas las apariciones o referencias de la clase Circulo a Circulito en el proyecto y renombrará **Circulo.java** a **Circulito.java**.

### Encerrar con... Introducir...

NetBeans puede hacer sugerencias sobre si encerrar un código seleccionado entre estructuras de control, sobre la posibilidad de introducir una nueva variable, campo o método con el código seleccionado o comentarlo.

Para ello, en la ventana de edición del código fuente, se debe seleccionar la expresión o grupo de sentencias y presionar **Alt-Enter**. Según el código seleccionado, NetBeans sugerirá encerrar el código entre las estructuras de control y los comentarios que sugiere, o introducir una variable, constante, campo o método.

Por ejemplo, para la siguiente expresión seleccionada, NetBeans sugiere comentarla o crear una nueva variable, campo o método con ella y no sugiere crear una constante porque no tiene sentido en este caso.



Por ejemplo, para la siguiente expresión seleccionada, NetBeans sugiere además de las otras opciones, crear una constante con ella.

The screenshot shows a portion of a Java code editor. Line 39 contains the assignment statement `radio=(valorRadio < 0.0 ? 0.0 : valorRadio);`. A context menu is open over the variable `radio`, listing several options: "Encerrar con (selection)", "Encerrar con /\* ... \*/", "Encerrar con //<editor-fold defaultstate="collapsed" desc="comment">...", "Introducir variable...", "Introducir constante...", "Introducir campo...", "Introducir método...", "Introduce Parameter...", and "Flip '<' to '>='". The option "Introducir constante..." is highlighted with a blue background.

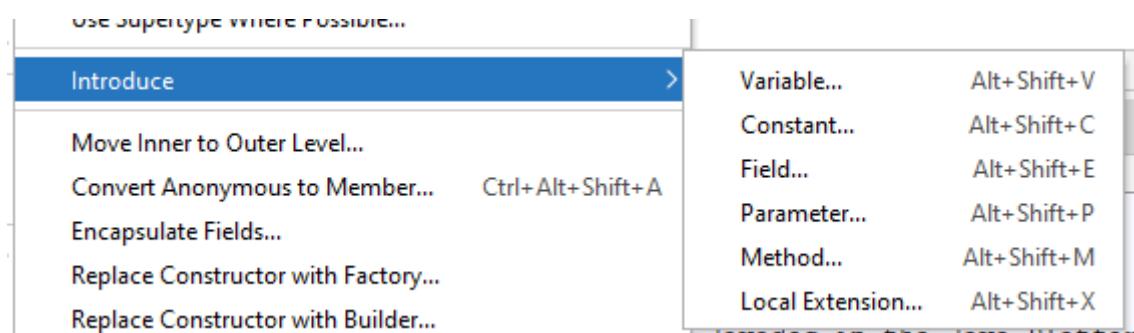
Por ejemplo, para la siguiente selección de código, NetBeans sugiere incluirla entre estructuras de control, try, comentario o introducir método:

The screenshot shows a portion of a Java code editor. Lines 10 through 26 define a `Main` class with a `main` method. The code creates a `Circulo` object, sets its coordinates and radius, and prints its properties to the console. A context menu is open over the entire block of code from line 10 to 26, listing various suggestions: "Encerrar con /\* ... \*/", "Encerrar con //<editor-fold defaultstate="collapsed" desc="comment">...", "Encerrar con Runnable r = new Runnable() { ... }", "Encerrar con do { ... }", "Encerrar con for (Map.Entry<KeyType, ValueType> entry : map.entrySet()) { ... }", "Encerrar con for (Iterator it = col.iterator(); it.hasNext()); { ... }", "Encerrar con for ( StringTokenizer TOKENIZER = new StringTokenizer(STRING); TOKENIZER.hasMoreTokens()); { ... }", "Encerrar con for (Object elem : col) { ... }", "Encerrar con for (int i = 0; i < 10; i++) { ... }", "Encerrar con for (int idx = 0; idx < arr.length; idx++) { ... }", "Encerrar con for (int idx = 0; idx < lst.size(); idx++) { ... }", "Encerrar con for (int idx = 0; idx < vct.size(); idx++) { ... }", "Encerrar con if (exp) { ... }", "Encerrar con if (exp) { ... } else { ... }", "Encerrar con try { ... }", "Encerrar con while (en.hasMoreElements()) { ... }", "Encerrar con while (true) { ... }", "Encerrar con while (it.hasNext()) { ... }", and "Introducir método...". The suggestion "Encerrar con try { ... }" is highlighted with a blue background.

## Introducir variable, constante, campo o método

Un fragmento de código seleccionado se puede refactorizar para crear con él una variable, una constante o un método. Esto se hace cuando se desea dividir el código en partes más pequeñas o más significativas y así facilitar futuras modificaciones, la reusabilidad del código y mejorar la comprensión del código.

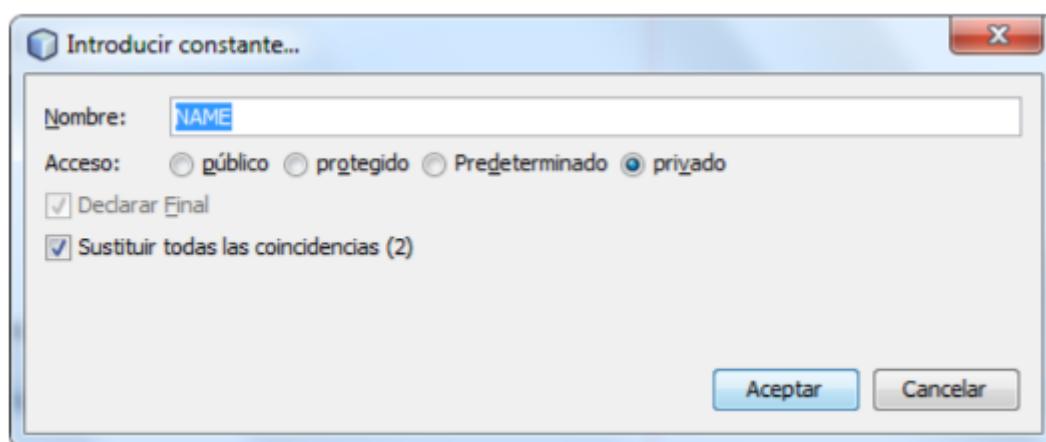
Se puede introducir una variable, una constante, un campo o un método como se vio en la sección anterior. Otra forma consiste en seleccionar una expresión en el código fuente y elegir en el menú principal **Refactorizar->Insertar variable... o constante o campo o método**. De cualquier manera, aparecerá una nueva ventana detallando la Refactorización como se indica inmediatamente.



### Introducir constante

Para introducir una constante se debe teclear el nombre de la constante (NetBeans propone un nombre en mayúsculas) y el tipo de acceso. No permite cambiar la declaración final y además permite reemplazar todas las coincidencias o solo las que estén seleccionadas.

Considerando seleccionado el valor 0.0 del método **establecerRadio()**:



Ambas coincidencias se cambiarán al nombre de la constante.

The screenshot shows a portion of Java code in the NetBeans code editor. Line 41 contains the assignment statement: `radio = (valorRadio < NAME ? NAME : valorRadio);`. The word 'NAME' is highlighted in green, indicating it is a placeholder for a constant. To the left, the line numbers 39, 40, 41, and 42 are visible.

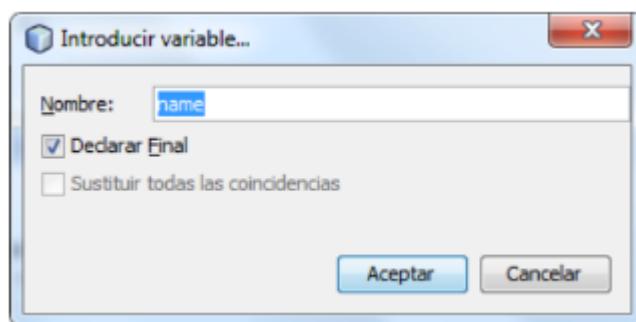
Aparecerá la declaración de la constante en la clase.

```
7  public class Círculo {  
8      public static final double NAME = 0.0;  
9  
10     private int x;  
11     private int y;  
12     private double radio;
```

### Introducir variable

Para introducir una variable hay que teclear el nombre de la nueva variable (NetBeans sugiere un nombre en minúsculas), la declaración **final** si se desea y decidir si la sustitución se realizará en todas coincidencias o sólo en la selección actual.

Considerando seleccionada la expresión **Math.PI \* obtenerDiametro()** del módulo **obtenerCircunferencia()**:



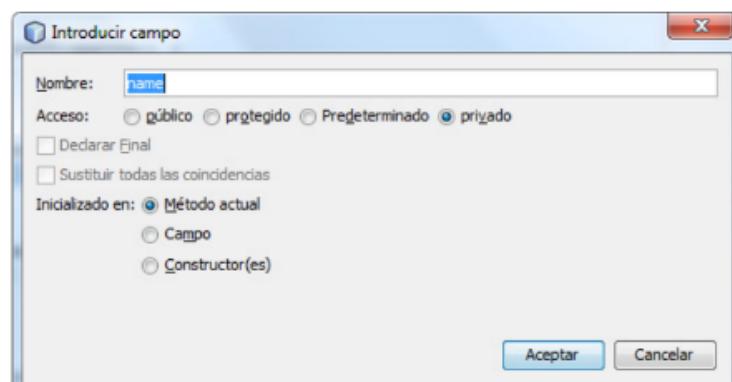
La expresión aparecerá reemplazada dentro del método por esta variable:

```
51  [ ]    public double obtenerCircunferencia() {  
52      final double name = Math.PI * obtenerDiametro();  
53      return name;  
54  }
```

### Introducir campo

Para introducir un campo, hay que indicar las características del campo (nombre, tipo de acceso, si la declaración es **final** y dónde se inicializará).

Considerando seleccionada la expresión **Math.PI \* obtenerDiametro()** del método **obtenerCircunferencia()**:



Aparecerá este nuevo campo declarado en la clase

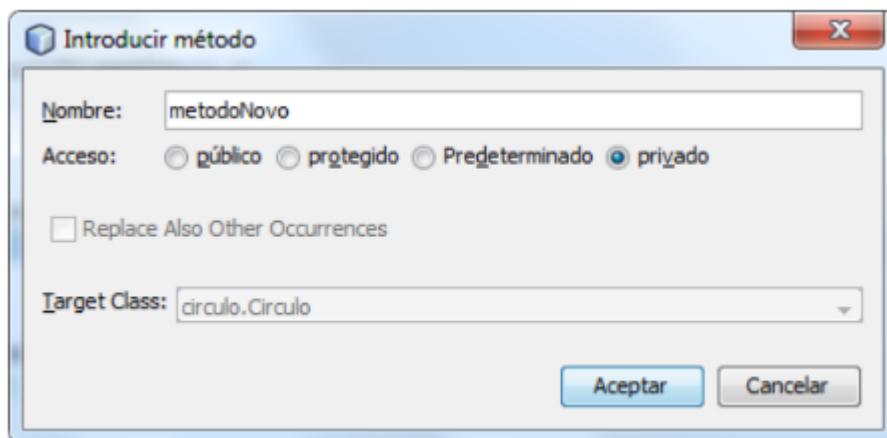
```
7     public class Circulo {  
8         private double name;  
9  
10        private int x;  
11        private int y;  
12        private double radio;
```

Aparecerá el campo inicializado en el método actual

```
52     public double obtenerCircunferencia() {  
53         name = Math.PI * obtenerDiametro();  
54         return name;  
55     }
```

### Introducir método

Si se elige introducir método, se debe indicar el nombre del método y el tipo de acceso. Teniendo en cuenta que está seleccionada la expresión **Math.PI \* obtenerDiametro()** del método **obtenerCircunferencia()**:



Ese método y la referencia a él se crearán dentro del método actual

```
51     public double obtenerCircunferencia() {  
52         return metodoNovo();  
53     }  
54  
55     private double metodoNovo() {  
56         return Math.PI * obtenerDiametro();  
57     }  
58  
59     public double obtenerArea() {  
60         return Math.PI * radio * radio;  
61     }
```

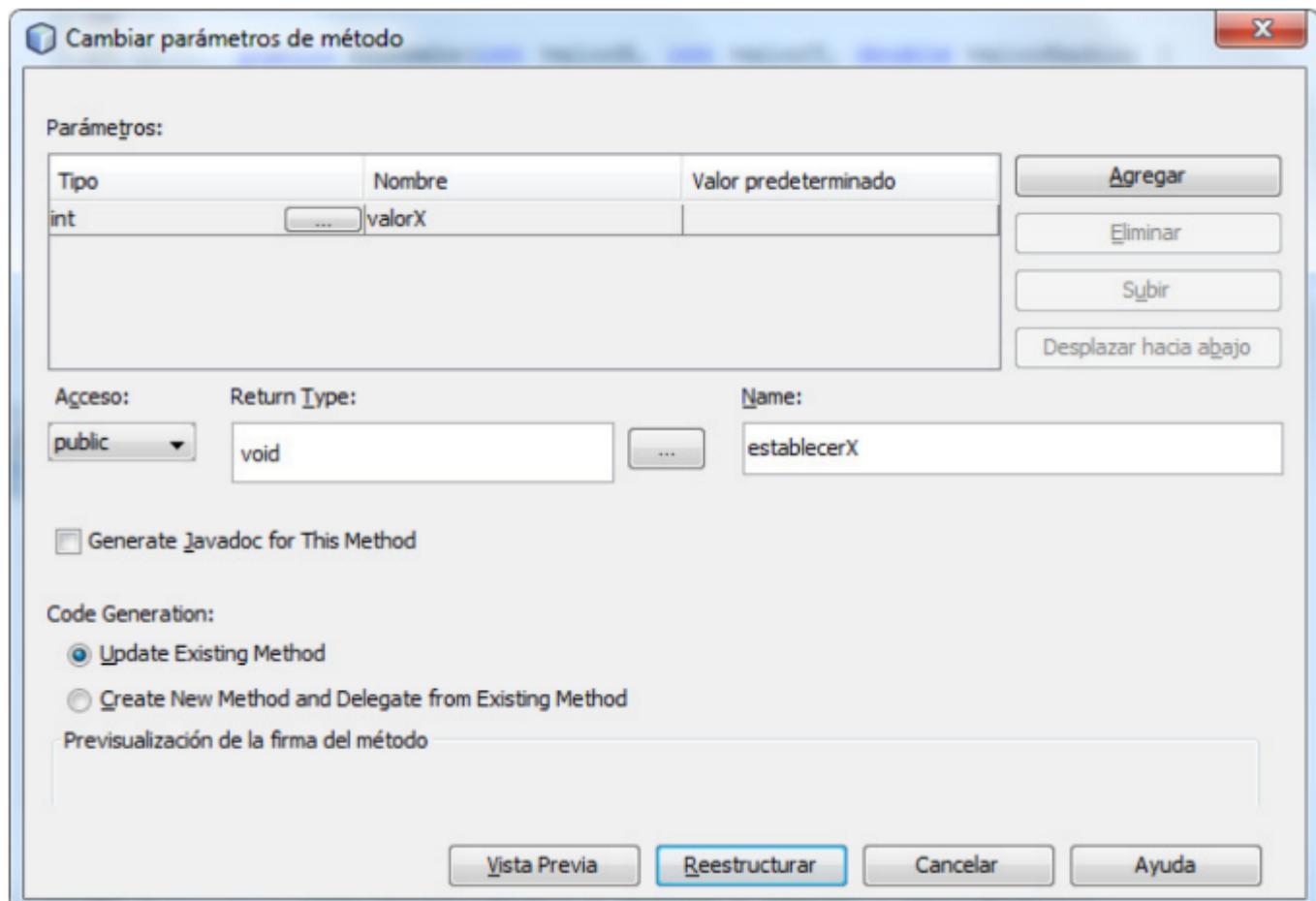
Si aparece un mensaje de error al ejecutar **introducir método**, puede ser porque el código seleccionado no puede convertirse en un método:

- La selección no puede tener más de un parámetro de salida.
- La selección no puede tener instrucciones **break** ni **continue** si la siguiente operación a realizar no está dentro de la selección.
- La selección no puede contener una sentencia **return** que no sea la última sentencia de la selección.
- La selección no puede tener un **return** condicional.

### Cambiar parámetros de un método

Cambiar los parámetros en un método permite alterar la firma del método agregando parámetros, cambiando el orden de los parámetros, cambiando el tipo de acceso del método y permitiendo que esos cambios se propaguen por todo el código.

Para realizar estos cambios, hay que estar sobre el método dentro del código fuente, hacer clic con el botón derecho del ratón y elegir **Refactorizar->Cambiar parámetros de método....**. A continuación se abre la ventana de **Cambiar parámetros de método** en la que se pueden añadir parámetros, eliminarlos, cambiar el orden de parámetros colocándolos antes o después, y cambiar el modificador de acceso del método.



## ☒ Agregar parámetros

En la ventana **Cambiar parámetros de método** hacer click sobre el botón **Agregar** y en la tabla de parámetros escribir el nombre, tipo y valor predeterminado del parámetro para colocar en las llamadas a métodos. Para editar el nombre, tipo o valor predeterminado es necesario hacer doble clic en la celda correspondiente.

## ☒ Cambiar el orden de los parámetros

En la ventana **Cambiar parámetros de método**, se selecciona el parámetro a mover y se presionan los botones **Subir** o **Desplazar hacia abajo**.

## ☒ Cambiar tipo de acceso

En la ventana **Cambiar parámetros de método**, se debe seleccionar el modificador de acceso requerido

Como ejemplo, se van a agregar dos parámetros al método **trasladarCentro()** para que el centro del círculo se mueva según los valores de los parámetros.

```
64     public void trasladarCentro() {
65         x+=5;
66         y+=5;
67     }
```

Que se usa en Main.java

```
24     System.out.println(salida);
25     circulo.trasladarCentro();
26     salida="\n\nLa nueva ubicación y el radio de círculo son\n"+circulo.toString();
27     System.out.println(salida);
```

Y en CirculoTest.java:

```
139     @Test
140     public void testTrasladarCentro() {
141         System.out.println("trasladarCentro");
142         Circulo instance = new Circulo();
143         int resultx=instance.obtenerX();
144         int resulty=instance.obtenerY();
145         instance.trasladarCentro();
146         int resultnx = instance.obtenerX();
147         int resultny = instance.obtenerY();
148         assertEquals(resultx+5, resultnx);
149         assertEquals(resulty+5, resultny);
150     }
```

Se añadirán los parámetros **trasladarY** y **trasladarX** de tipo int y con valor 5 por defecto, quedando los códigos modificados como:

```
64  public void trasladarCentro(int trasladarY, int trasladarX){  
65      x+=5;  
66      y+=5;  
67  }  
  
24      System.out.println(salida);  
25      circulo.trasladarCentro(5, 5);  
26      salida="\n\nLa nueva ubicación y el radio de círculo son\n"+circulo.toString();  
27      System.out.println(salida);  
  
139     @Test  
140     public void testTrasladarCentro() {  
141         System.out.println("trasladarCentro");  
142         Circulo instance = new Circulo();  
143         int resultx=instance.obtenerX();  
144         int resulty=instance.obtenerY();  
145         instance.trasladarCentro(5, 5);  
146         int resultnx = instance.obtenerX();  
147         int resultny = instance.obtenerY();  
148         assertEquals(resultx+5, resultnx);  
149         assertEquals(resulty+5, resultny);  
150     }  
}
```

La refactorización realizada no cambia el código del método, por lo que si después de la refactorización necesitas cambiar las líneas:

```
x+=5;  
y+=5;
```

por:

```
x+=trasladarX;  
y+=trasladarY;
```

debería hacerse manualmente.

## Encapsular campos

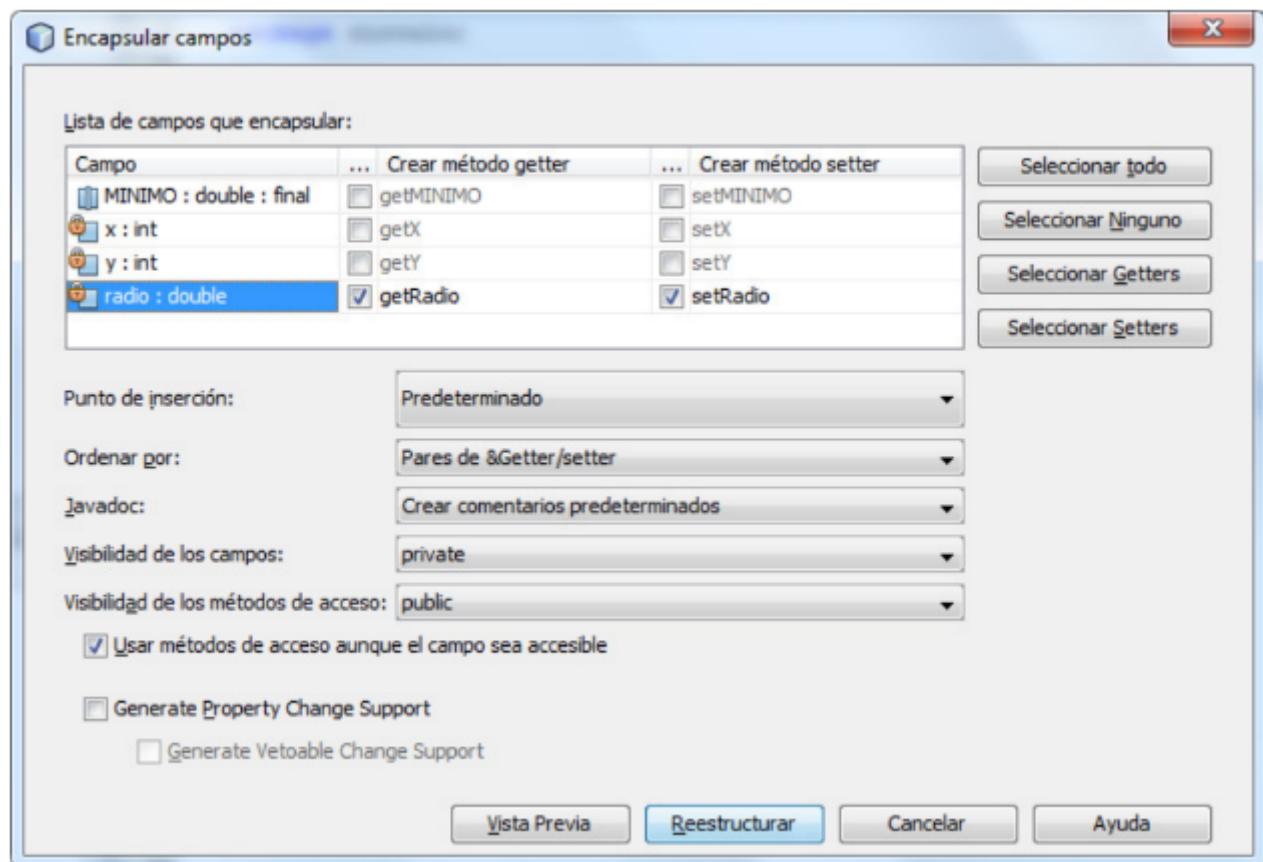
Encapsular un campo consiste en hacer que el campo tenga acceso **privado** y sea accesible utilizando un par de métodos de tipo **get** y **set** serán públicos.

La refactorización para encapsular campos en NetBeans es muy flexible y permite generar métodos de tipo **get** y **set** para acceder a un campo. Este proceso se subdivide en:

- Generar los métodos de acceso.
- Ajustar los modificadores de acceso para los campos.
- Sustituir las referencias a ese campo en el código con llamadas a los métodos de acceso.

Esta refactorización no eliminará los métodos que ya existen para acceder a los campos desde dentro de la clase.

Para encapsular un campo, hacer clic con el botón derecho en el campo o en una referencia al campo y elegir **Refactorizar->Encapsular campos...** y se abre el cuadro de diálogo **Encapsular campos**.

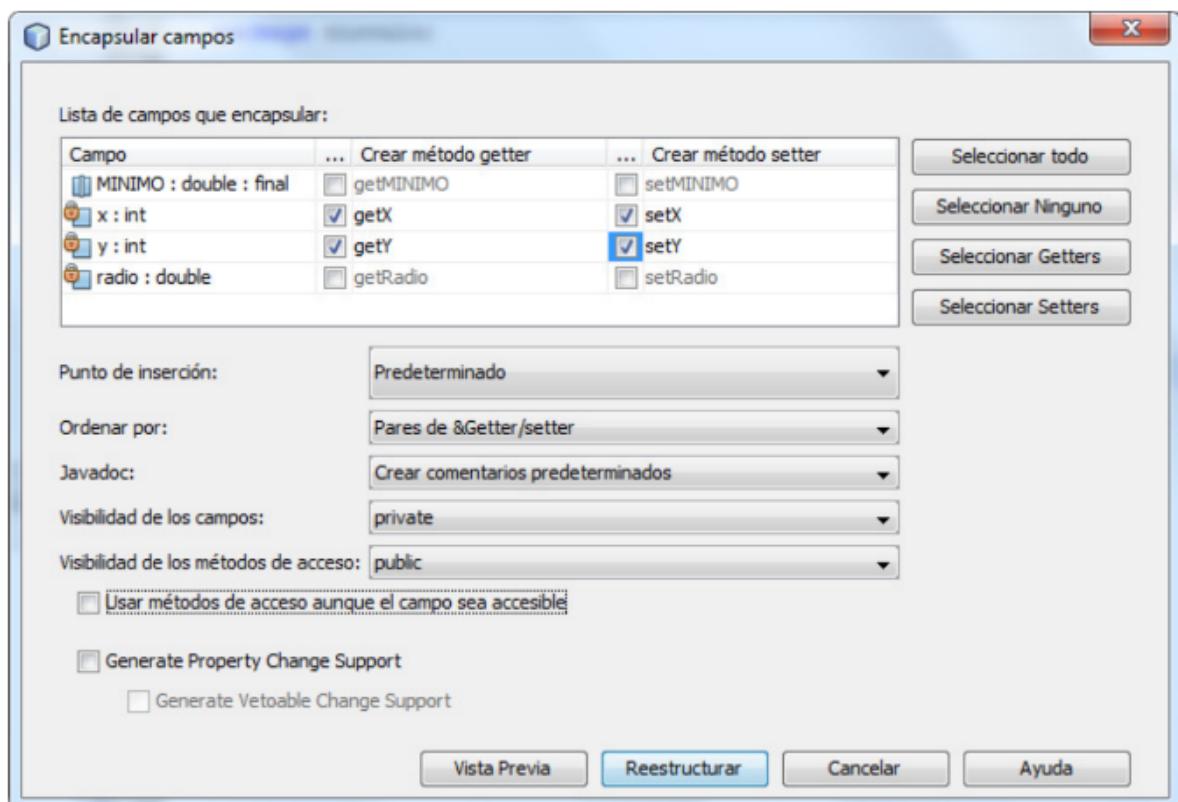


Aparece la lista de campos que se pueden encapsular y aparece marcado el campo seleccionado con el que se accedió a la refactorización.

Se puede:

- Marcar los campos que se desea encapsular de forma individual usando los checkbox que están en la lista de campos, o utilizar los botones que están a la derecha y que permiten marcar todos, desmarcar todos, marcar solo los métodos de tipo **get** o solo el tipo **set**.
- Indicar en qué punto del código se deben insertar los nuevos métodos: como primer método, como último método o después de uno de los métodos existentes en la clase (aparece la lista de todos).
- Indicar cómo se deben colocar los diferentes métodos tipo **get** y **set** dentro de la ubicación anterior: intercalado **get** y **set** de cada campo, por nombre de método o todos los **get** primero y después los **set**.
- Indicar cómo se quiere que sea la documentación de los métodos.
- Indicar la visibilidad de los campos que serán encapsulados.
- Indicar la visibilidad de los métodos tipo **get** y **set**.
- Desmarcar el checkbox **Usar métodos de acceso aunque el campo sea accesible**, si no desea utilizar los nuevos métodos de acceso si el campo ya está accesible.

Como ejemplo, se encapsularán los campos **x**, **y** de la clase **Circulo**, creando los métodos **get** y **set**, poniendo todos los métodos **get** juntos, después todos los **set** juntos y luego colocando los métodos como los primeros en la clase



Que dará como resultado:

```

64     public void trasladarCentro() {
65         setX(getX() + 5);
66         setY(getY() + 5);
67     }
68
69     /**
70      * @return the x
71      */
72     public int getX() {
73         return x;
74     }
75
76     /**
77      * @param x the x to set
78      */
79     public void setX(int x) {
80         this.x = x;
81     }

```

```

82
83     /**
84      * @return the y
85      */
86     public int getY() {
87         return y;
88     }
89
90     /**
91      * @param y the y to set
92      */
93     public void setY(int y) {
94         this.y = y;
95     }
96
97

```

e incluye las llamadas a estos métodos en los métodos que hacen referencia a los campos **x** e **y**:

```
60  public String toString() {
61      return "Centro = [" + getX() + "," + getY() + "]; Radio = " + radio;
62  }
63
64  public void trasladarCentro() {
65      setX(getX() + 5);
66      setY(getY() + 5);
67  }
```

pero que no elimina, sino que modifica aquellos métodos que hacen lo mismo que los métodos de tipo **get** y **set**. Antes de la refactorización eran:

```
22  public void establecerX(int valorX) {
23      x=valorX;
24  }
25
26  public int obtenerX() {
27      return x;
28  }
```

y ahora son:

```
22  public void establecerX(int valorX) {
23      setX(valorX);
24  }
25
26  public int obtenerX() {
27      return getX();
28  }
```

que no tienen ninguna utilidad nueva y deberían eliminarse.

### Borrar de forma segura

Se debe usar esta opción siempre que se necesite eliminar un elemento del código, ya que permite saber si hay referencias a él dentro del proyecto antes de realizar la eliminación real.

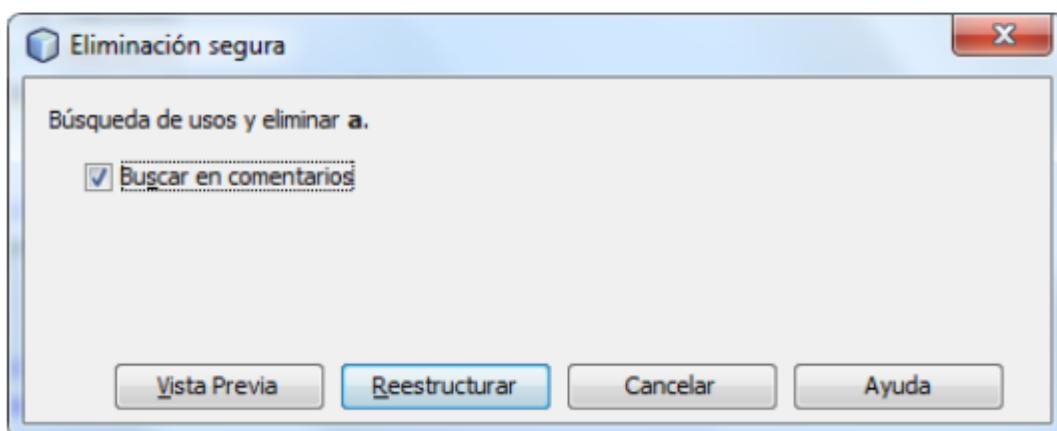
Pasos a seguir para un borrado seguro:

- Colocar el cursor en el elemento del código que desea eliminar y seleccione **Refactor->Safety Delete**. Se abrirá el cuadro de diálogo Safety Delete.
- Asegurarse de que el elemento que aparece en el cuadro sea el que se desea eliminar y realizar una vista previa del borrado.
- Para eliminar un método, ponga el cursor sobre su firma (primera línea) sin seleccionar nada y elija **Refactor -> Safety Delete**.

- ☒ Si no se hace referencia al elemento en ninguna otra parte del código, aparecerá la ventana **Reestructurando** y se puede proceder a eliminarlo.
- ☒ Si se hace referencia al elemento en alguna parte, aparece la ventana **Eliminación segura** con la advertencia de que se hace referencia al elemento en otra ubicación y, por lo tanto, no se puede eliminar directamente. En esta ventana se puede:
  - ✓ cancelar la operación de borrado.
  - ✓ presiona el botón **Mostrar uso...** para obtener información detallada de las referencias. En la ventana **Usos** aparece una vista de árbol con los archivos y elementos de clase que hacen referencia al elemento a borrar y una serie de iconos como los de la ventana **Reestructurando**. Se pueden usar los íconos a la izquierda de esa ventana de manera que haciendo doble clic en una de las ocurrencias se puede editar el fragmento de código en el que se encuentra la referencia y cambiarlo para dejar de hacer referencia a él. A continuación se puede hacer clic en el botón **Ejecutar de nuevo Eliminación segura** para reiniciar el proceso de eliminación hasta que no se haga referencia al elemento.

Por ejemplo, eliminación segura del método **a** que nunca se llama:

```
public int a(){
    return x;
}
```



Al hacer clic en **Vista Previa**, aparece la ventana **Reestructurando** ya que no hay aviso de que existen referencias al método y se marca en morado el método a eliminar. La refactorización se ejecuta al hacer clic en el botón **Reestructurar**.

**Reestructurando**

Borrar los elementos, pero mantener  
Círculo  
Círculo.java  
Update

```

Círculo.java          Reestructurados Círculo.java
1/1
20  establecerRadio(valorRadio); 20  establecerRadio(valorRad ...
21 }                                21 }
22                                         22
23 public void establecerX(int valor 23  public void establecerX(int
24     x=valorX;                      x=valorX;
25 }                                25 }
26                                         26
27 public int obtenerX() {           27  public int obtenerX() {
28     return x;                      return x;
29 }                                29 }
30                                         30
31 public int a() {                31
32     return x;                      32
33 }                                33
34                                         34
35 public void establecerY(int valor 35  public void establecerY(int
36     y=valorY;                      y=valorY;
37 }                                37 }
38                                         38
39 public int obtenerY() {           39  public int obtenerY() {
40     return y;                      return y;
41 }                                41 }
42                                         42

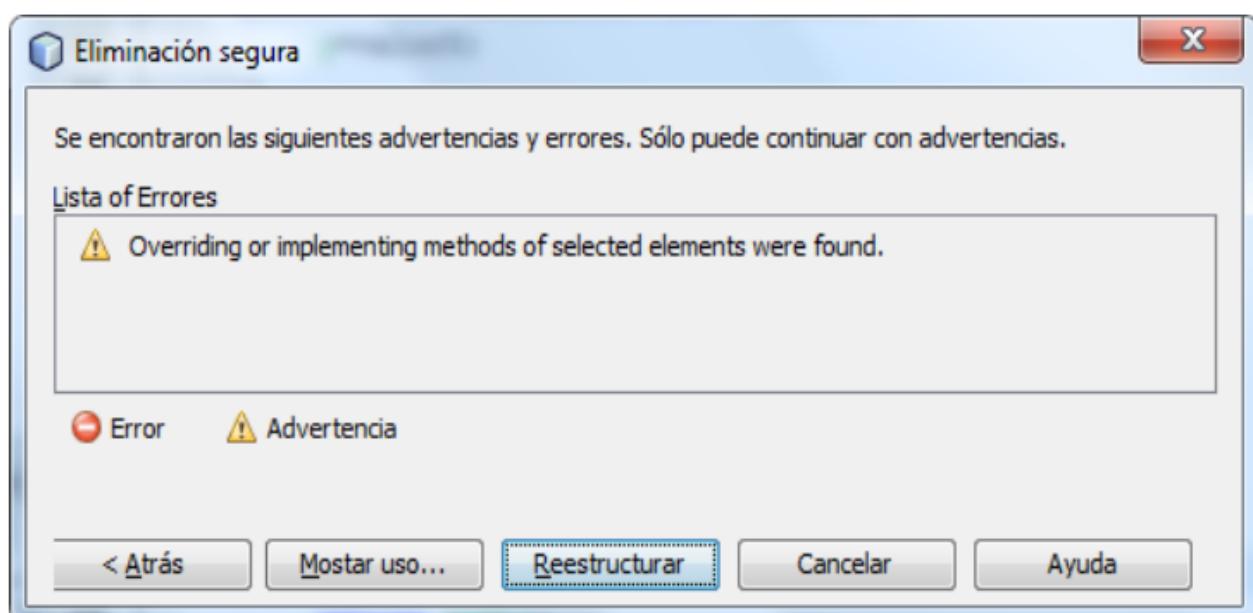
```

Realizar reestructuración Cancelar

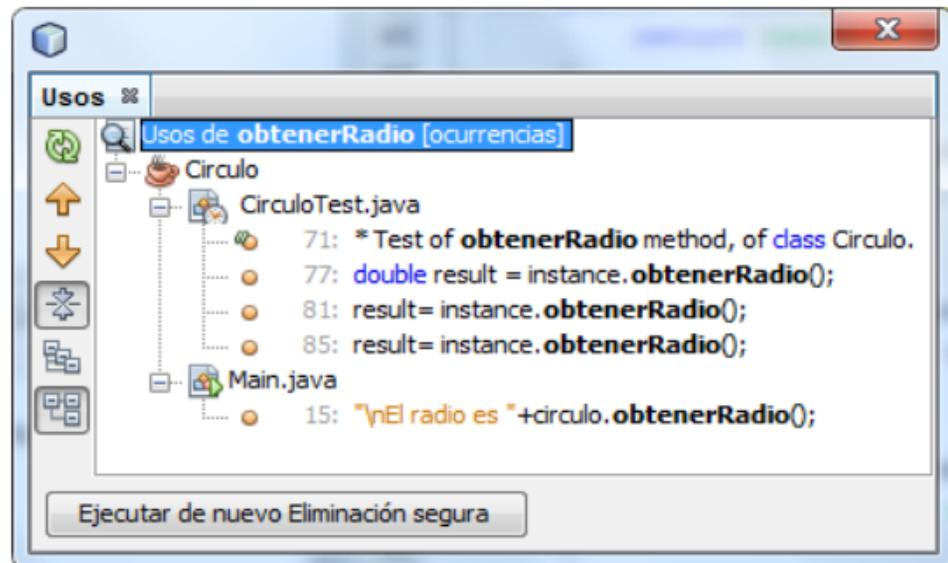
Si el método que se va a eliminar estuviera referenciado en otra parte del código, como en el siguiente ejemplo:



Al hacer clic en **Vista Previa** aparecería el siguiente cuadro:



En el que podrá hacer, entre otras operaciones, la cancelación del borrado o pinchar en **Mostrar uso...** para ver los detalles de las referencias en la ventana de **Usos**:



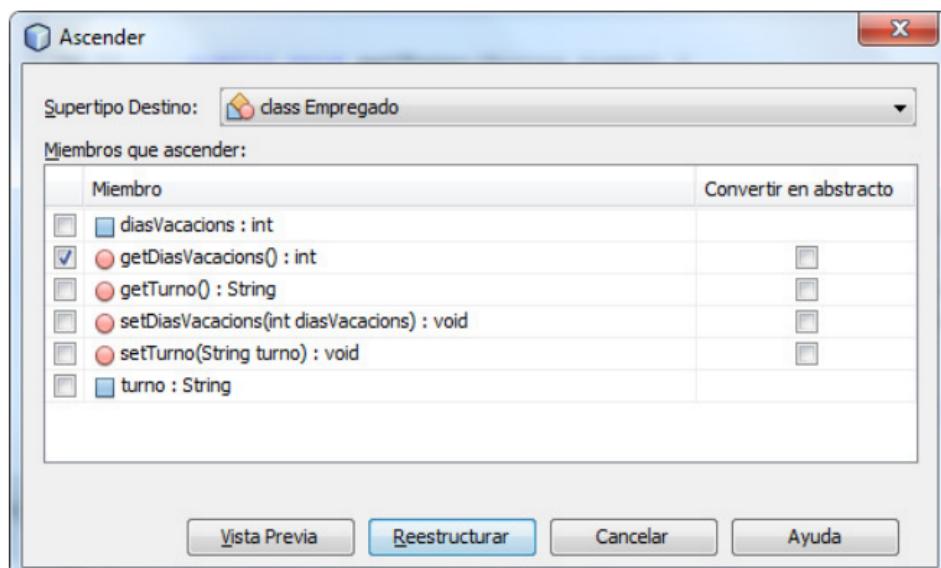
En la ventana de **Usos** se indica que hay 5 referencias: 1 en el archivo **Main.java** y 4 en el archivo **CirculoTest.java**. Moviéndote por esas referencias con los iconos del menú de la izquierda o directamente haciendo doble clic sobre una referencia, puedes ver y editar la línea de código fuente en la que se encuentra la referencia, pudiendo modificarla para que la referencia ya no exista. Después de modificar la referencia, se debe hacer clic en **Ejecutar de nuevo Eliminación segura** para volver a iniciar el proceso de eliminación segura hasta que no haya referencias al método **obtenerRadio** y aparezca la ventana **Reestructurando**, en ese momento podrá eliminarse el método con seguridad.

Hay que recordar que si se comete algún error en el borrado seguro o una vez finalizado el mismo, la operación se puede deshacer en **Edición->Deshacer**.

### Mover miembros de una clase a una superclase

Los métodos y campos se pueden mover a una superclase. Los pasos a seguir son:

- ☒ En la ventana de código fuente del proyecto, debe seleccionar la clase que contiene los miembros que se quieren mover y elegir **Refactorizar->Ascender... (Pull Up...)**.
- ☒ Aparece el cuadro de diálogo **Ascender** con una lista de miembros de clase e interfaces que implementa la clase.



- Seleccionar la superclase a la que se desea mover.
- Seleccionar los miembros que se quieren mover. Si la clase actual implementa interfaces, hay checkboxes para esas interfaces que, si se marcan, se moverán las implementaciones a la superclase.
- Si desea crear un método abstracto, se debe seleccionar el checkbox correspondiente para ese método, entonces será declarado en la superclase como un método abstracto y será sobreescrito en la clase actual.

Si la clase cuyos miembros se están moviendo a la superclase, tiene subclases y no se desea que todos sus elementos sean movidos, deberá accederse a la **vista previa de la Refactorización** y desmarcar los checkboxes correspondientes.

### Mover miembros de una clase a una subclase

Las clases internas, los métodos y los campos se pueden mover a todas las subclases de la clase actual.

Los pasos a seguir son:

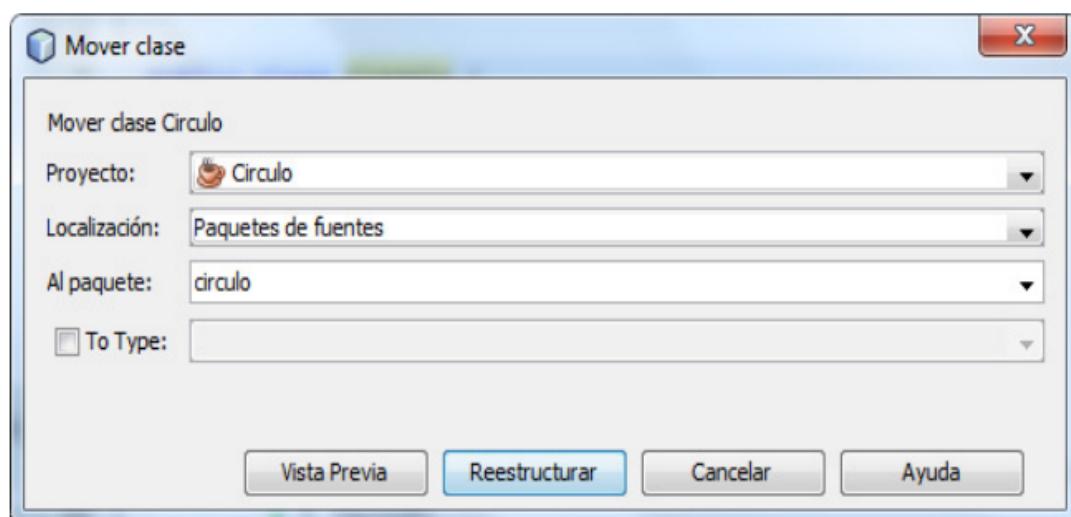
- En el código fuente o en la ventana del proyecto, tienes que seleccionar los elementos que quieres mover y elegir **Refactorizar->Descender... (Push Down...)**.
- El cuadro de diálogo **Descender** aparece con una lista de miembros de la clase que tiene el mismo aspecto que el cuadro de diálogo Ascender. Tienes que marcar la casilla de aquellos que quieras mover.
- Si hay métodos abstractos que desea mantener definidos en la clase actual y tenerlos implementados en la subclase, se debe marcar el checkbox **Mantener abstracto**. El checkbox en la columna izquierda debe estar marcado para que la definición de clase se copie a la subclase.

Si la clase cuyos miembros se están moviendo a la subclase, tiene subclases y no se desea que todos sus elementos sean movidos, deberá accederse a la **vista previa de la Refactorización** y desmarcar los checkboxes correspondientes.

### Mover clase a otro paquete Java

Pasos para mover una clase a otro paquete y cambiar el código que hace referencia a esa clase:

- En el código fuente o en la ventana del proyecto, haga clic en el botón de la clase que desea mover con el botón derecho del ratón y elegir **Refactorizar->Mover**.
- Aparece el cuadro de diálogo **Mover clase**.



Este cuadro también se muestra después de cortar y pegar archivos Java en la ventana de Proyectos, en la ventana Archivos, o después de arrastrar y soltar archivos en las mismas ventanas.

Tiene los campos:

- **Proyecto:** con el nombre del proyecto que contiene las clases a mover.
- **Localización:** la parte del proyecto que contiene las clases que se van a mover. Normalmente Paquetes de fuentes.
- **Al paquete:** nombre del paquete al que desea mover las clases o nombre completo del paquete como por ejemplo: **com.myCom.myPkg**.

Si la clase que se está moviendo tiene subclases y no desea que se muevan todos sus elementos, debe acceder a la **vista previa de la Refactorización** y desmarcar los checkboxes correspondientes.

No se recomienda mover una clase a otro paquete sin utilizar la refactorización, aunque es posible realizando los siguientes pasos:

- ☒ Mover manualmente la clase a otro paquete desde la ventana del proyecto cortando y pegando o arrastrando y soltando.
- ☒ Se muestra el cuadro de diálogo **Mover clase** y seleccionar el checkbox **mover sin Reestructurar**.

### Copiar clase

Pasos para copiar una clase dentro del mismo paquete o en otro paquete y cambiar el código que hace referencia a esa clase:

- ☒ En el código fuente dentro de la ventana de proyectos, sobre la clase, hay que hacer clic con el botón derecho del ratón y seleccionar **Refactorizar->Copiar**.
- ☒ Aparece el cuadro de diálogo **Copiar clase** que es similar a la caja **Mover clase**. En este cuadro, hay que seleccionar el paquete al que desea copiar (se recomienda esta opción) o escribir el nombre completo, por ejemplo, **com.myCom.myPkg**.

Si la clase que se está copiando tiene subclases y no se desea que se copien todos sus elementos, debe acceder a la **vista previa de la Refactorización** y desmarcar los checkboxes correspondientes.

No se recomienda copiar la clase sin utilizar la refactorización, aunque es posible:

- ☒ Copiar manualmente la clase a otro paquete o al mismo paquete desde la ventana del proyecto cortando y pegando o arrastrando y soltando.
- ☒ Aparecerá el cuadro de diálogo **Copiar clase** y seleccionar el checkbox **Copiar sin Reestructurar**.

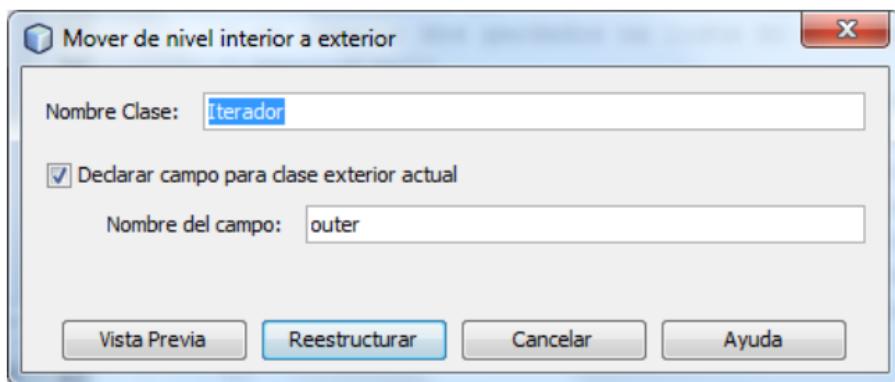
### Mover una clase del nivel interno al externo

Una clase interna se puede mover a un nivel superior en la jerarquía de clases. Por ejemplo, si la clase seleccionada está anidada dentro de una clase superior, la clase seleccionada se crea en ese nivel superior. Si la clase seleccionada está anidada en una clase interna, la clase seleccionada se mueve al nivel de la clase interna. Pasos a seguir:

- ☒ Se debe hacer clic con el botón derecho del ratón sobre la clase interna que se desea mover y elegir **Refactorizar->Mover de nivel interior a exterior**.
- ☒ Aparece el cuadro de diálogo **Mover de nivel interior a exterior**.

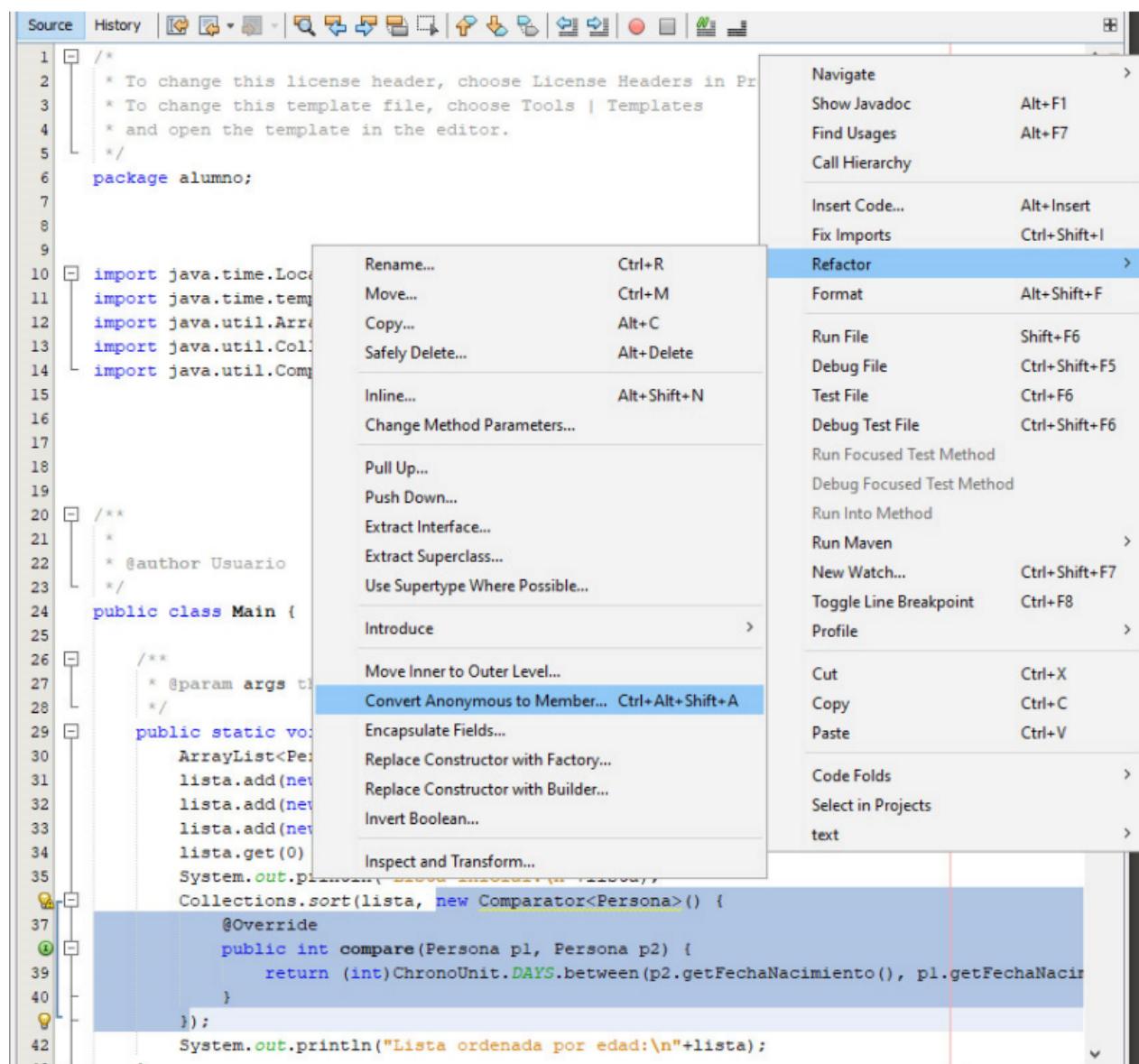
En este cuadro:

- Se puede cambiar el nombre de la nueva clase.
- Opcionalmente, se puede crear un campo en la nueva clase que hará referencia a un objeto de la clase contenadora original y darle nombre a ese campo.



### Convertir clase anónima en miembro

Una clase anónima se puede convertir en una clase interna con un nombre y un constructor. Se crea la nueva clase interna y se reemplaza la clase anónima por una llamada a la clase interna recién creada. Para ello debemos seleccionar exactamente el código de la clase interna (mira la siguiente imagen) y pulsar el **botón derecho >Refactorizar > Convert anonymous to Member...**



Después de refactorizar se obtiene:

```
    Collections.sort(lista, new ComparatorImpl());
    System.out.println("Lista ordenada por edad:\n"+lista);
}

private static class ComparatorImpl implements Comparator<Persona> {

    public ComparatorImpl() {
    }

    @Override
    public int compare(Persona p1, Persona p2) {
        return (int)ChronoUnit.DAYS.between(p2.getFechaNacimiento(), p1.getFechaNacimiento());
    }
}
```

## Extraer interface

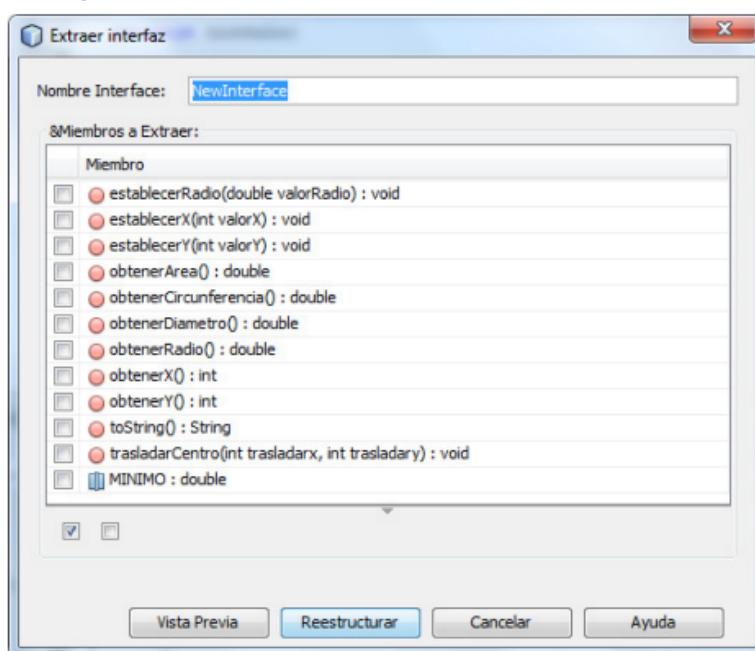
Esta opción permite seleccionar los métodos públicos no estáticos de una clase o interfaz, que terminarán en una nueva interfaz. Una interfaz no restringe cómo se implementan sus métodos, las interfaces se pueden usar en clases que tienen diferentes funciones. La creación de interfaces puede aumentar la reutilización del código.

Cuando se extrae una interfaz, el IDE hace lo siguiente:

- Crea una nueva interfaz con los métodos seleccionados en el mismo paquete que la clase o interfaz actual.
- Actualiza, aplica o extiende la clase actual o interfaz para incluir la nueva interfaz.

Para extraer una interfaz:

- Abrir la clase o interfaz que contiene los métodos que desea mover a una interfaz.
- Hacer clic en la opción de menú **Refactorizar ->Extraer interfaz**.
- Se muestra el cuadro de diálogo **Extraer interfaz**.



- Escribir el nombre de la nueva interfaz en el campo de texto.
- Elegir los miembros que desea extraer a la nueva interfaz.

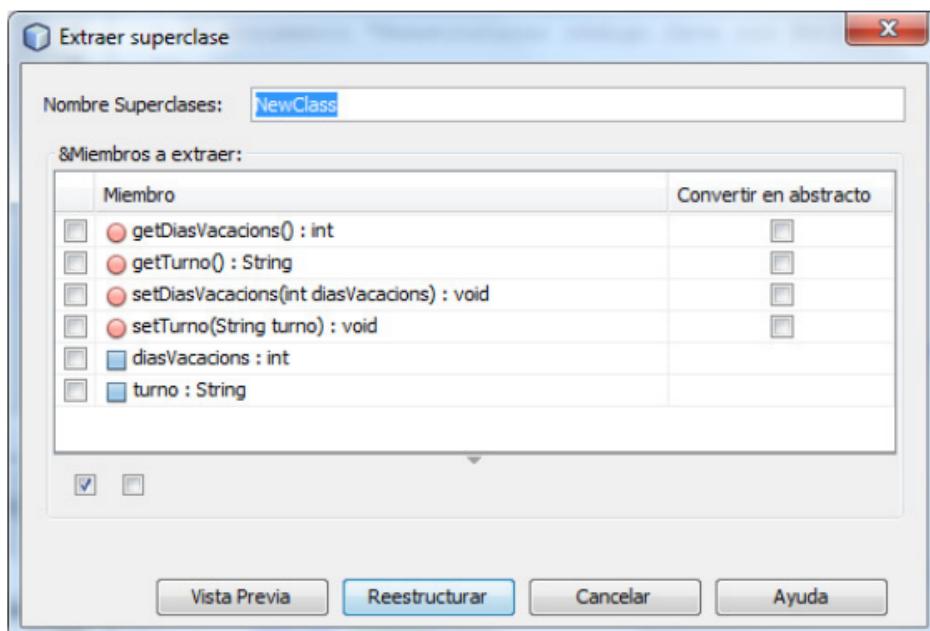
## Extraer superclase

Cuando se extrae una superclase, el IDE hace lo siguiente:

- ☒ Crea una nueva clase con los métodos y campos seleccionados de la clase seleccionada. También puede aplicar interfaces a la nueva clase que se implementan en la clase seleccionada.
- ☒ Si la clase seleccionada extiende una clase, la nueva clase también extiende la misma clase.
- ☒ La clase seleccionada se modifica para que extienda la nueva superclase.
- ☒ Mueve los campos públicos o protegidos seleccionados a la nueva superclase.
- ☒ Obtenga una vista previa de cómo se verán las clases después de extraer la superclase.

Para extraer una superclase:

- ☒ Abrir la clase que contiene los métodos o campos que desea mover a la nueva superclase.
- ☒ Hacer clic en la opción de menú **Refactorizar ->Extraer superclase**.
- ☒ Se abre el cuadro de diálogo **Extraer superclase** con los métodos y campos que se pueden extraer.



- ☒ Escribir el nombre de la nueva superclase en el cuadro de texto.
- ☒ Seleccionar los miembros que desea extraer a la nueva superclase.
- ☒ (Opcional) Se puede crear un método abstracto, seleccione el checkbox **Convertir en abstracto** para el método. Si selecciona este checkbox, el método se declarará en la superclase como un método abstracto y se anulará en la clase actual.

## Después de la Refactorización

Después de cualquier **refactorización**, especialmente si implica varias operaciones complicadas, el proyecto debe limpiarse y reconstruirse. Para ello, con el cursor sobre el nombre del proyecto en la ventana del proyecto, debe hacer clic con el botón derecho del ratón y elegir **Limpiar y generar(Clean and Build)**.