

*Optimización y Documentación*

***1º DAM***

***Sergio Ramírez Cruz***

***Autor: Sergio***

*2018*

Índice

[Refactorización 3](#_Toc508747445)

[Patrones de refactorización 3](#_Toc508747446)

[Analizadores de código 7](#_Toc508747447)

[Herramientas para analizar código 8](#_Toc508747448)

[Control de Versiones 10](#_Toc508747449)

[Iniciar un repositorio 10](#_Toc508747450)

[Cambios en el editor de código 11](#_Toc508747451)

[Visor de control de versiones 12](#_Toc508747452)

[Documentación 13](#_Toc508747453)

[Uso de comentarios 13](#_Toc508747454)

[Documentación de clases 14](#_Toc508747455)

[Herramientas 15](#_Toc508747456)

*Optimización y Documentación*

# Refactorización

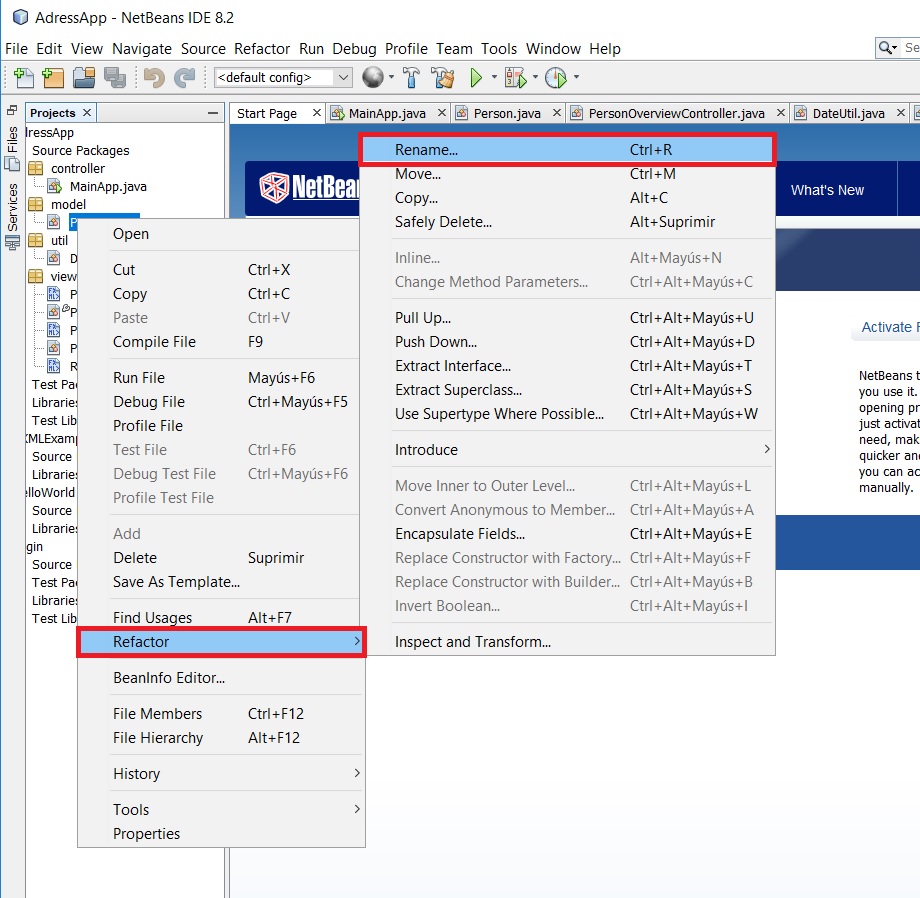
La refactorización es una disciplina técnica, que consiste en realizar pequeñas transformaciones en el código de un programa, para mejorar la estructura sin que cambie el comportamiento ni funcionalidad del mismo. Su objetivo es mejorar la estructura interna del código. Es una tarea que pretender limpiar 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.

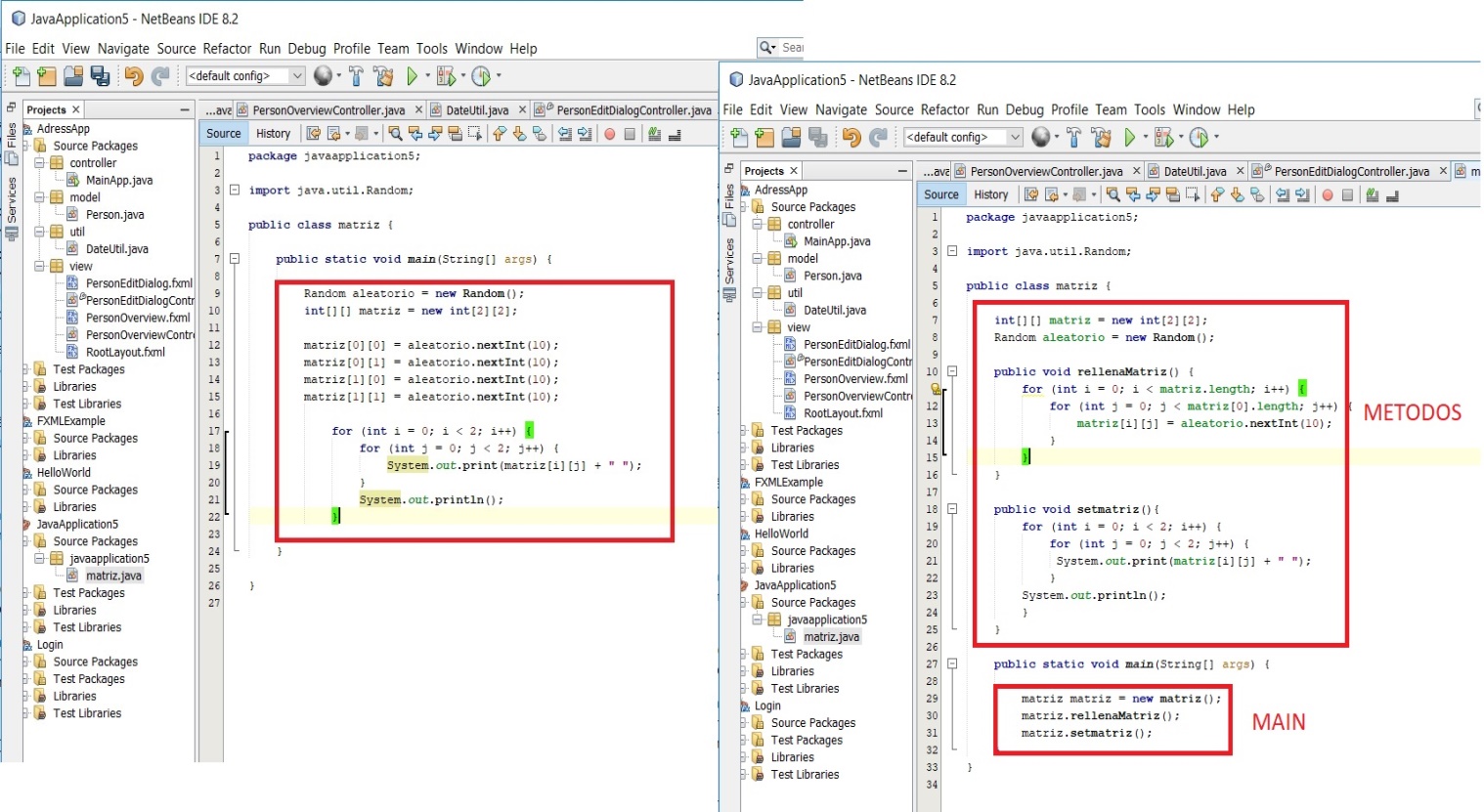
Cuando se refactoriza se está mejorando el diseño del código después de haberlo escrito. Podemos partir de un mal diseño y, aplicando la refactorización, llegaremos a un código bien diseñado. Cada paso es simple, por ejemplo mover una propiedad desde una clase a otra, convertir determinado código en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

## Patrones de refactorización

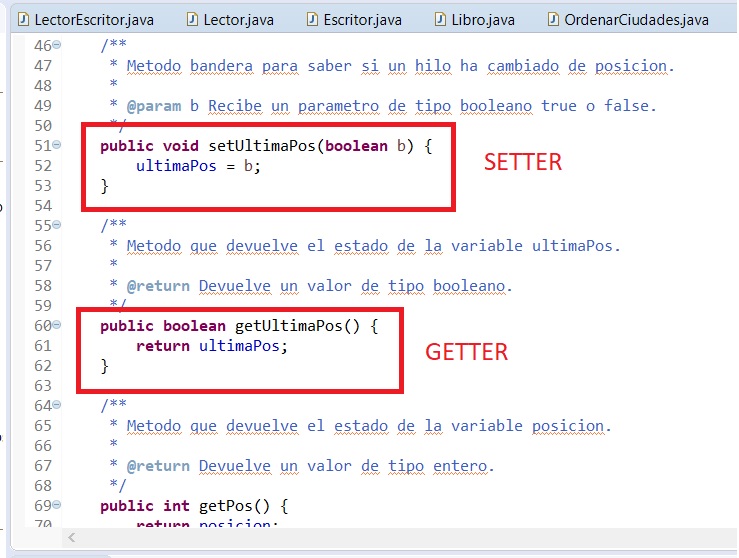
**Renombrado** (rename): Este patrón nos indica que debemos cambiar el nombre de un paquete, clase, método o campo, por un nombre más significativo.



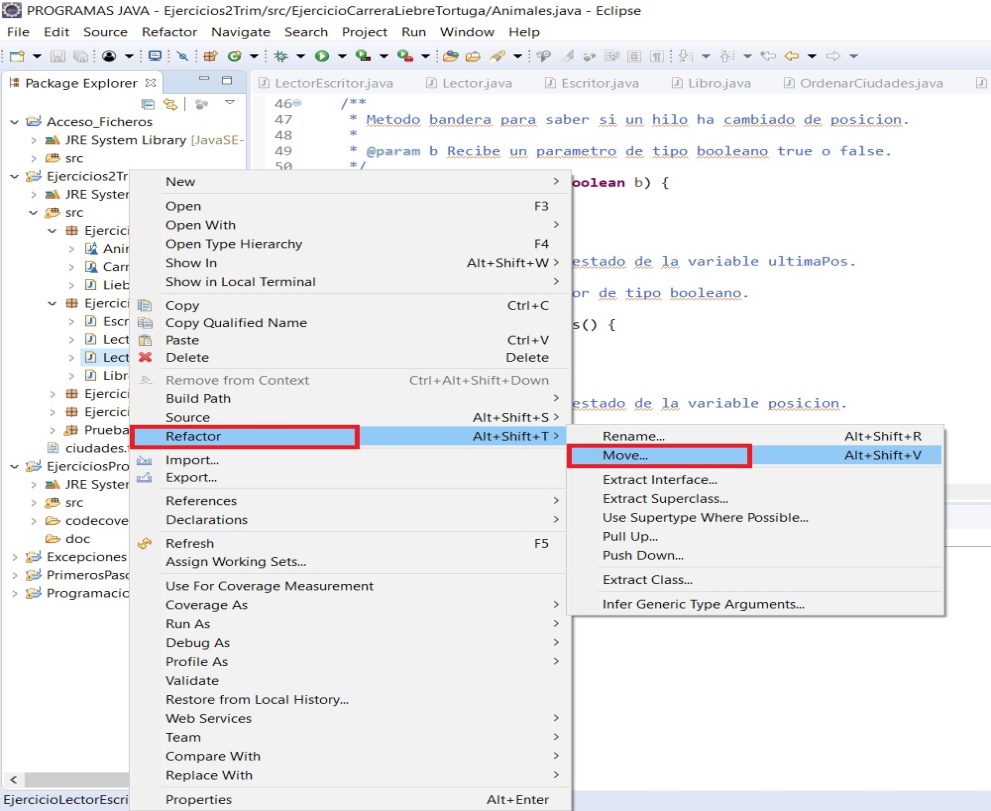
**Sustituir bloques de código por un método**: Este patrón nos aconseja sustituir un bloque de código, por un método. De esta forma, cada vez que queramos acceder a ese bloque de código, bastaría con invocar al método.



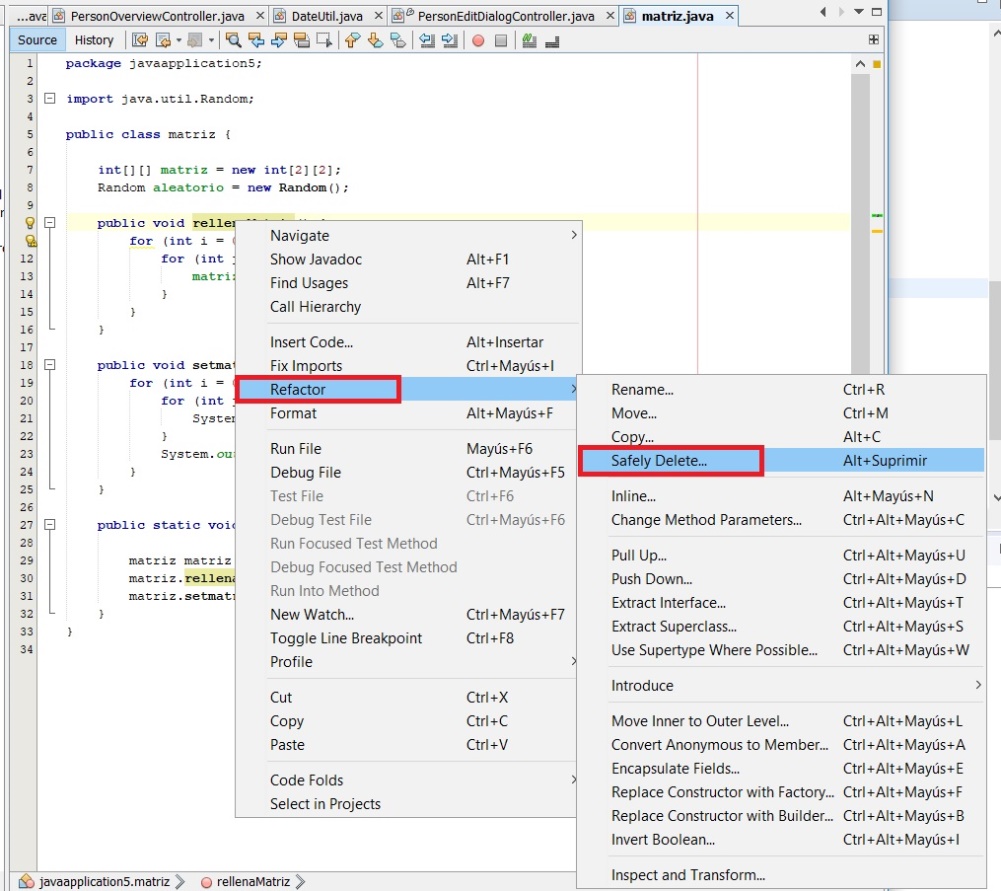
**Campos encapsulados**: Se aconseja crear métodos getter y setter, (de asignación y de consulta) para cada campo que se defina en una clase. Cuando sea necesario acceder o modificar el valor de un campo, basta con invocar al método getter o setter según convenga.



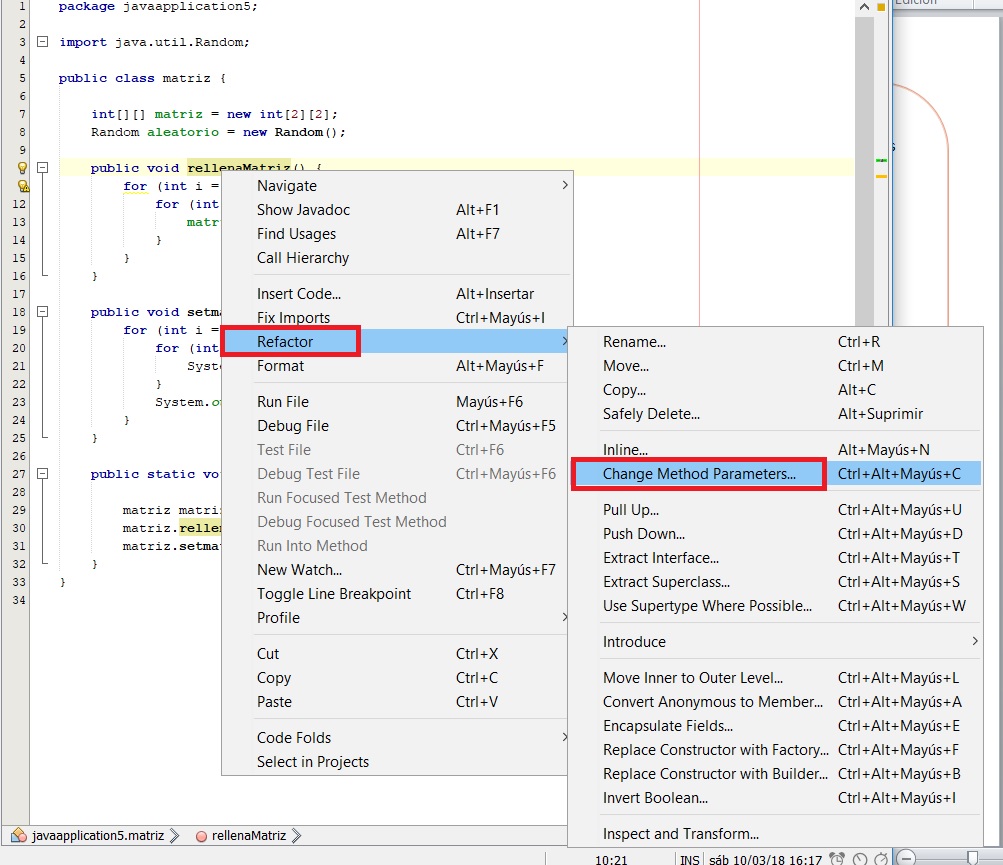
**Mover la clase**: Si es necesario, se puede mover una clase de un paquete a otro, o de un proyecto a otro. La idea es no duplicar código que ya se haya generado. Esto impone la actualización en todo el código fuente de las referencias a la clase en su nueva localización.



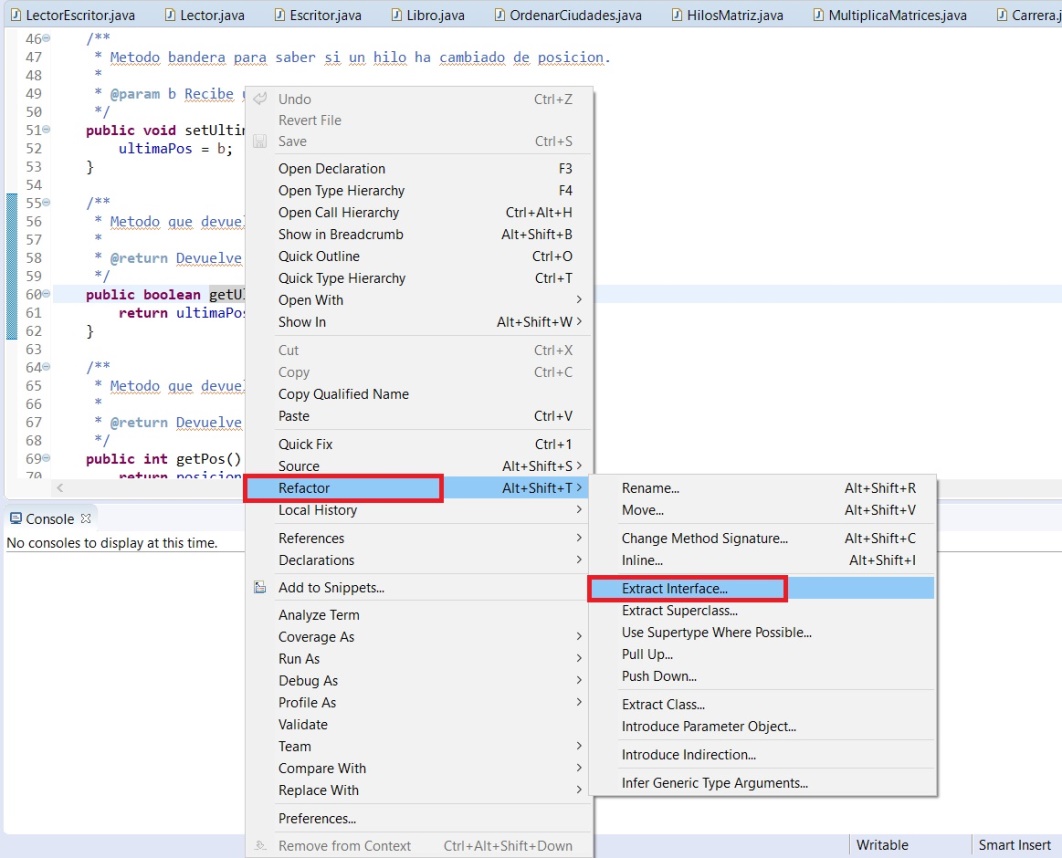
**Borrado seguro**: Se debe comprobar, que cuándo un elemento del código ya no es necesario, se han borrado todas las referencias a él que había en cualquier parte del proyecto.



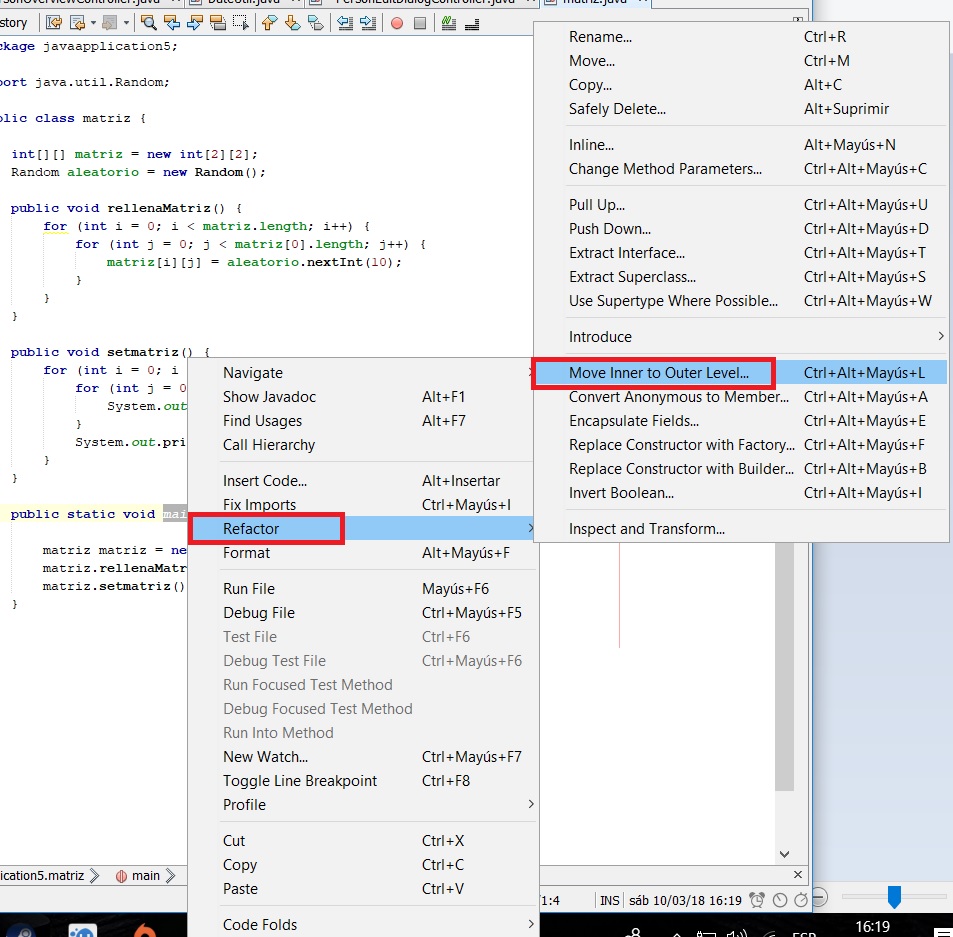
**Cambiar los parámetros del proyecto**: Nos permite añadir nuevos parámetros a un método y cambiar los modificadores de acceso.



**Extraer la interfaz**: Crea un nueva interfaz de los métodos public non‐static seleccionados en una clase o interfaz.



**Mover del interior a otro nivel**: Consiste en mover una clase interna a un nivel superior en la jerarquía.



## 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, pero sin que se modifique la semántica.

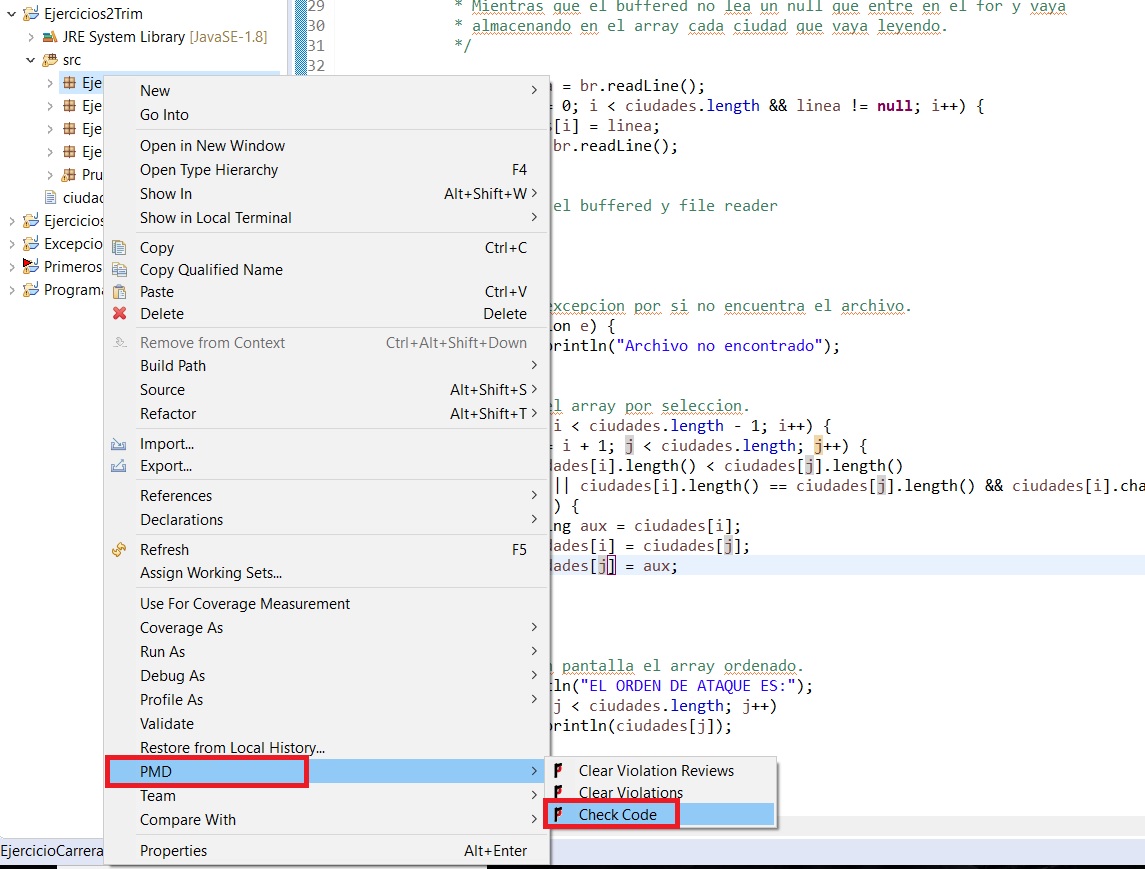
Los analizadores de código, son las herramientas encargadas de realizar esta labor. El analizador estático de código recibirá el código fuente de nuestro programa, lo procesará intentando averiguar la funcionalidad del mismo, y nos dará sugerencias, o nos mostrará posibles mejoras.

Cada IDE incluye herramientas de refactorización y analizadores de código. En el caso de software libre, existen analizadores de código que se pueden añadir como complementos a los entornos de desarrollo. Respecto a la refactorización, los IDE ofrecen asistentes que de forma automática y sencilla, ayudan a refactorizar el código.

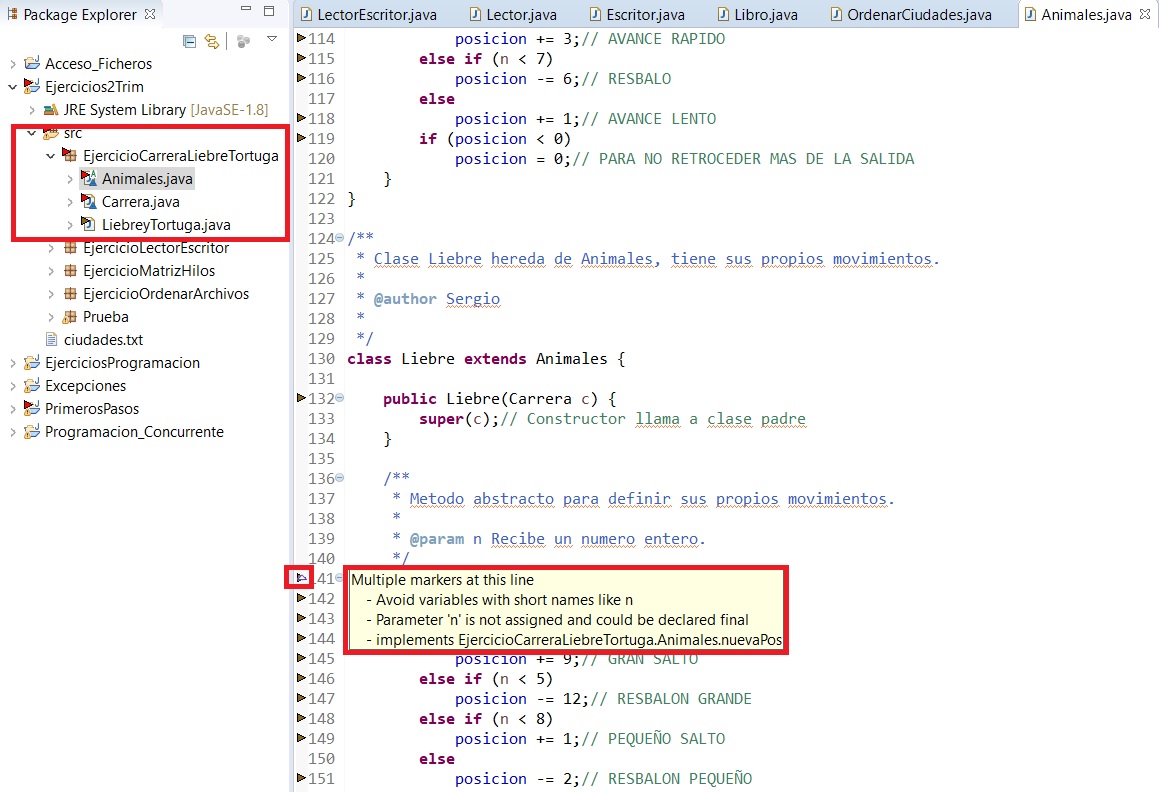
### Herramientas para analizar código

1º Instalaremos el plugin PMD en Eclipse en este caso.

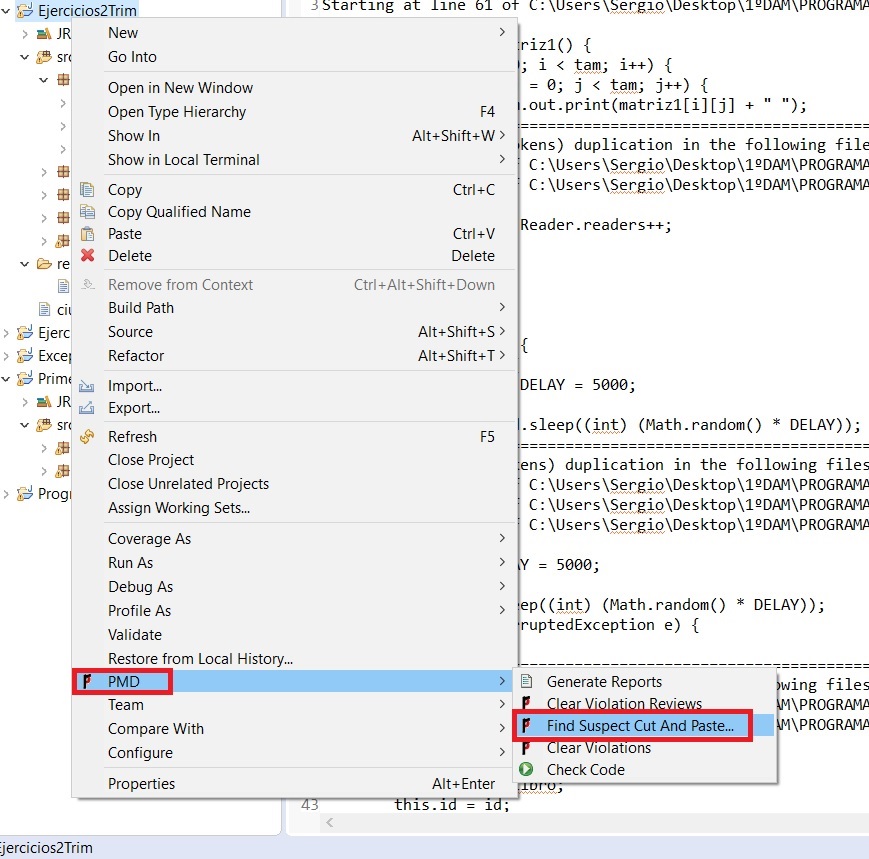
2º Hacemos click derecho en el paquete que queramos analizar después en PMD > Check Code.



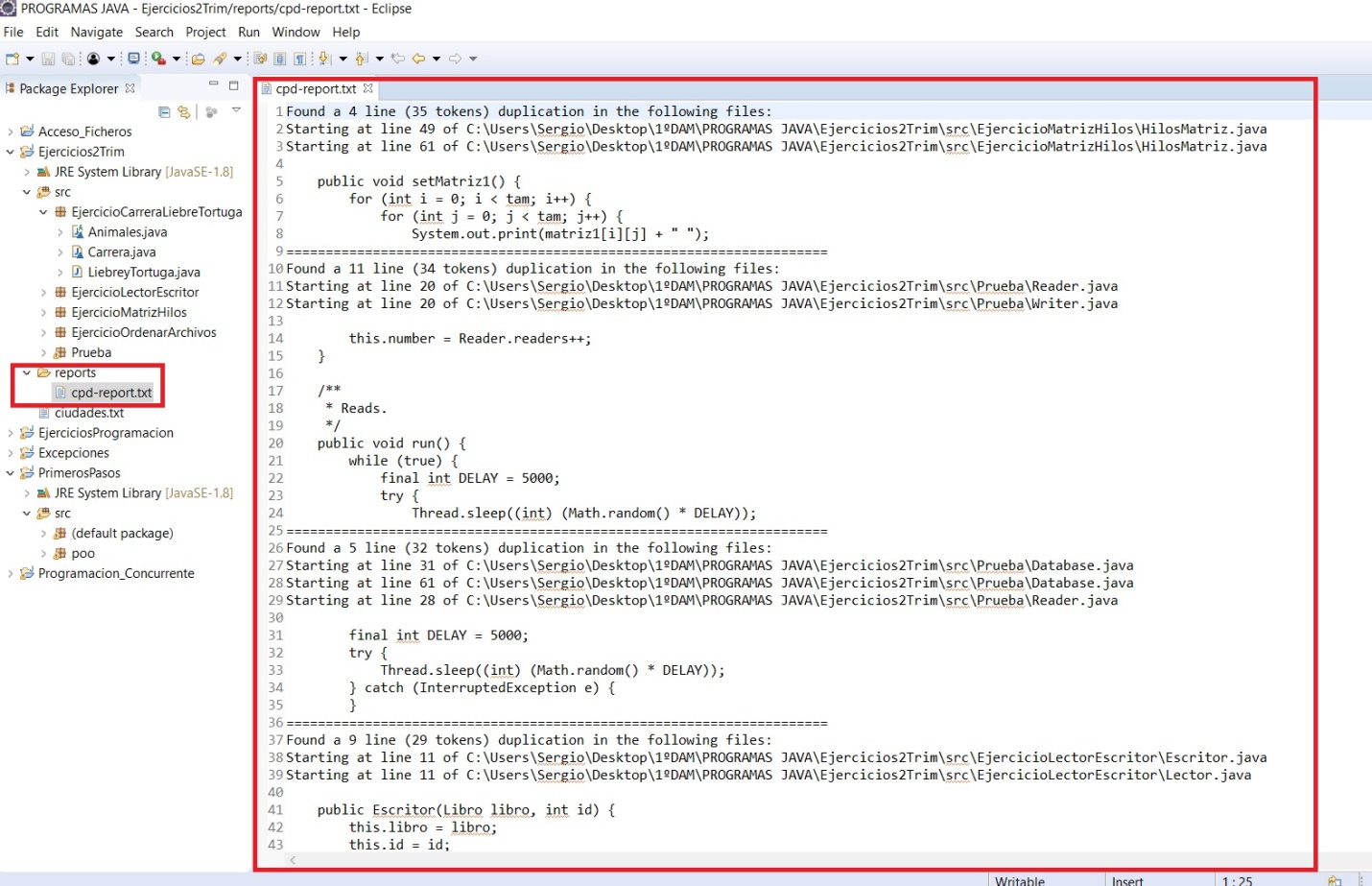
3º Observamos los consejos que nos da el PMD para mejorar nuestro código.



4º También podemos buscar código repetido en nuestro programa para ello vamos al proyecto click derecho y en PMD > Find Suspect Cut And Paste.



5º Nos genera una carpeta en nuestro proyecto llamada reports y un archivo con las posibles líneas de código repetidas.



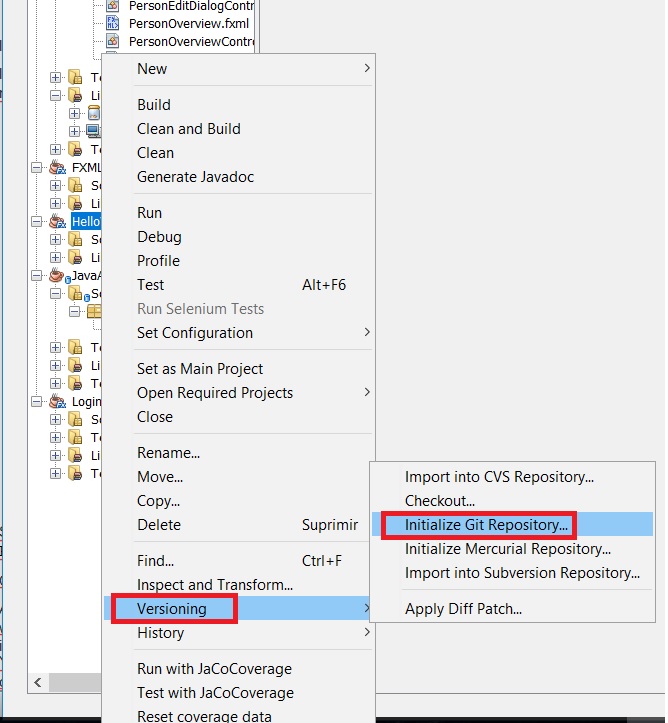
# Control de Versiones

Para gestionar las distintas versiones que se van generando durante el desarrollo de una aplicación, los IDE, proporcionan herramientas de Control de Versiones y facilitan el desarrollo en equipo de aplicaciones. En este trabajo instalaremos la herramienta CVS.

## Iniciar un repositorio

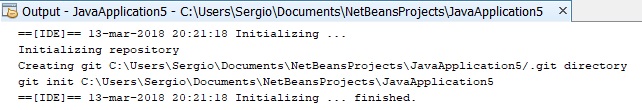
1º Hacemos click derecho en el proyecto.

2º Nos vamos a  Versioning > Initialize Git Repository.

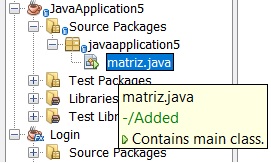


3º Especificamos la ruta en la que vamos a almacenar los archivos versionados.

4º Comprobamos que se ha creado correctamente mirando la salida.



Después de hacer esto vemos que nuestro árbol del proyecto tiene una marca esto quiere decir que ya está inicializado si dejamos el cursor sobre una clase nos saldrá que esta añadida en verde.

  
Después de inicializar el repositorio de Git, podemos agregar o enviar archivos directamente.

## Cambios en el editor de código

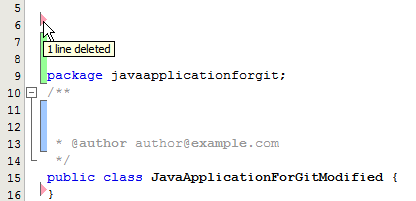
Cuando abre un archivo versionado en el Editor de código fuente del IDE, puede ver los cambios en tiempo real que ocurren en su archivo cuando lo modifica contra la versión base del repositorio de Git. A medida que trabaja, el IDE utiliza códigos de colores en los márgenes del Editor de origen para transmitir la siguiente información:

Azul 🡪 Indica las líneas que se han cambiado desde la versión anterior.

Verde 🡪 Indica las líneas que se han agregado desde la versión anterior.

Rojo 🡪 Indica las líneas que se han eliminado desde la versión anterior.

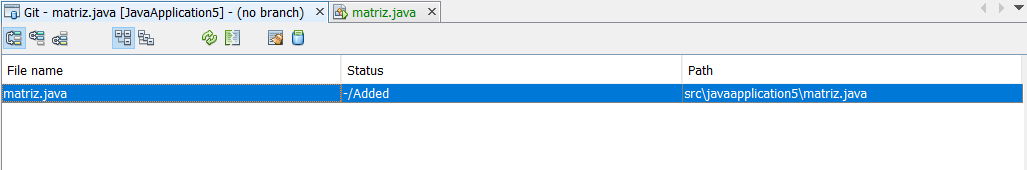
El margen izquierdo del Editor de origen muestra los cambios que se producen línea por línea. Cuando modifica una línea determinada, los cambios se muestran inmediatamente en el margen izquierdo.



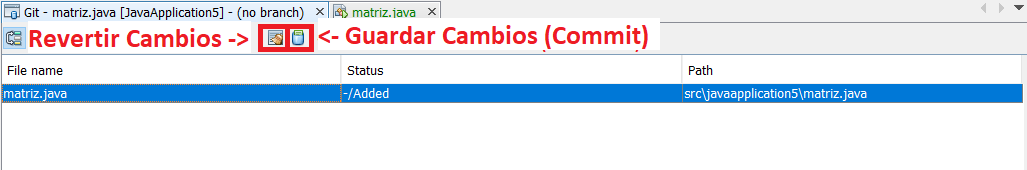
Visor de control de versiones

La vista de control de versiones de Git nos proporciona una lista en tiempo real de todos los cambios realizados en los archivos dentro de una carpeta seleccionada de su árbol de proyecto.

Para abrir la vista de versiones, seleccionamos nuestro proyecto y vamos a Git > Show Changes. La siguiente ventana aparece en la parte inferior del IDE:



En esta ventana también tenemos los botones para revertir cambios y para guardarlos.



# Documentación

El proceso de documentación de código, es uno de los aspectos más importantes de la labor de un programador. Documentar el código nos sirve para explicar su funcionamiento, punto por punto, de forma que cualquier persona que lea el comentario, puede entender la finalidad del código.

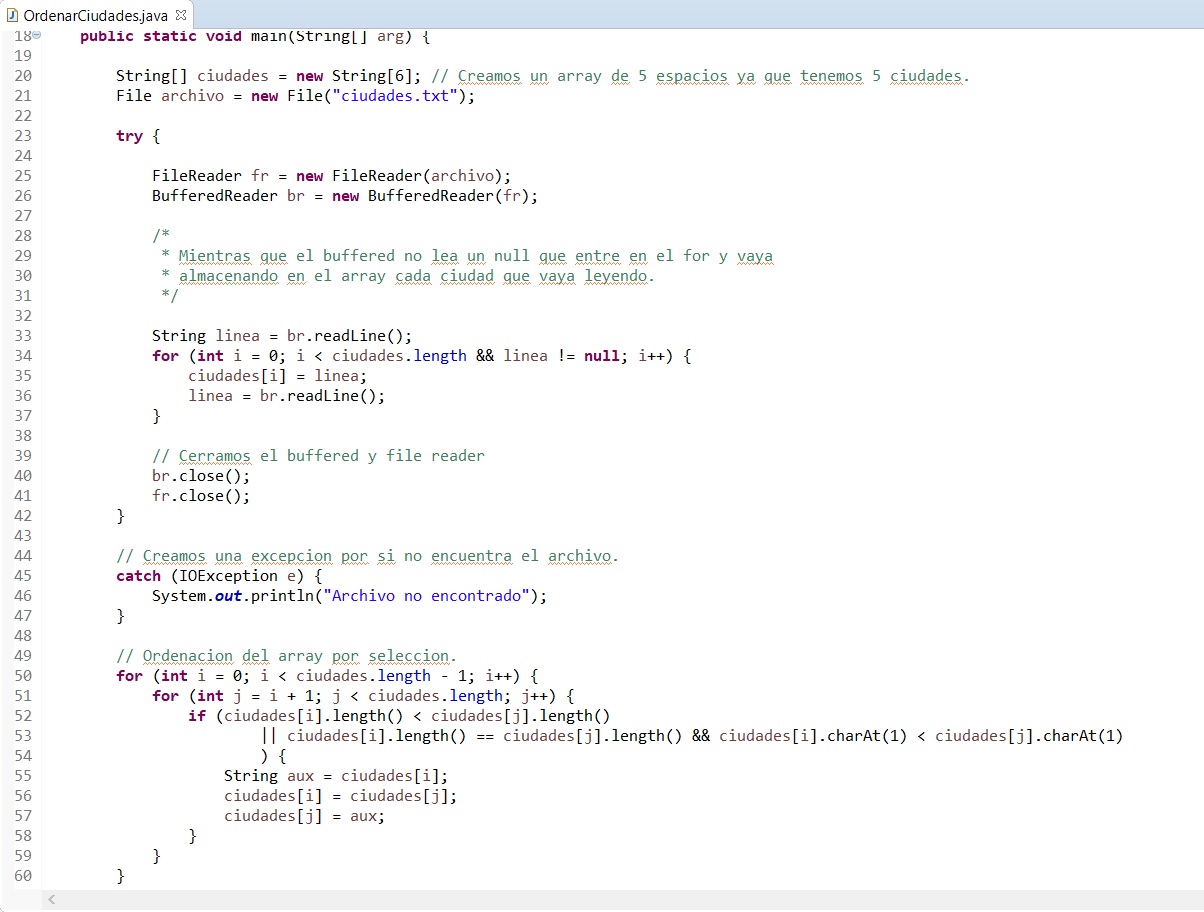
La documentación añade explicaciones de la función del código, de las características de un método, etc. Debe tratar de explicar todo lo que no resulta evidente. Su objetivo no es repetir lo que hace el código, sino explicar por qué se hace.

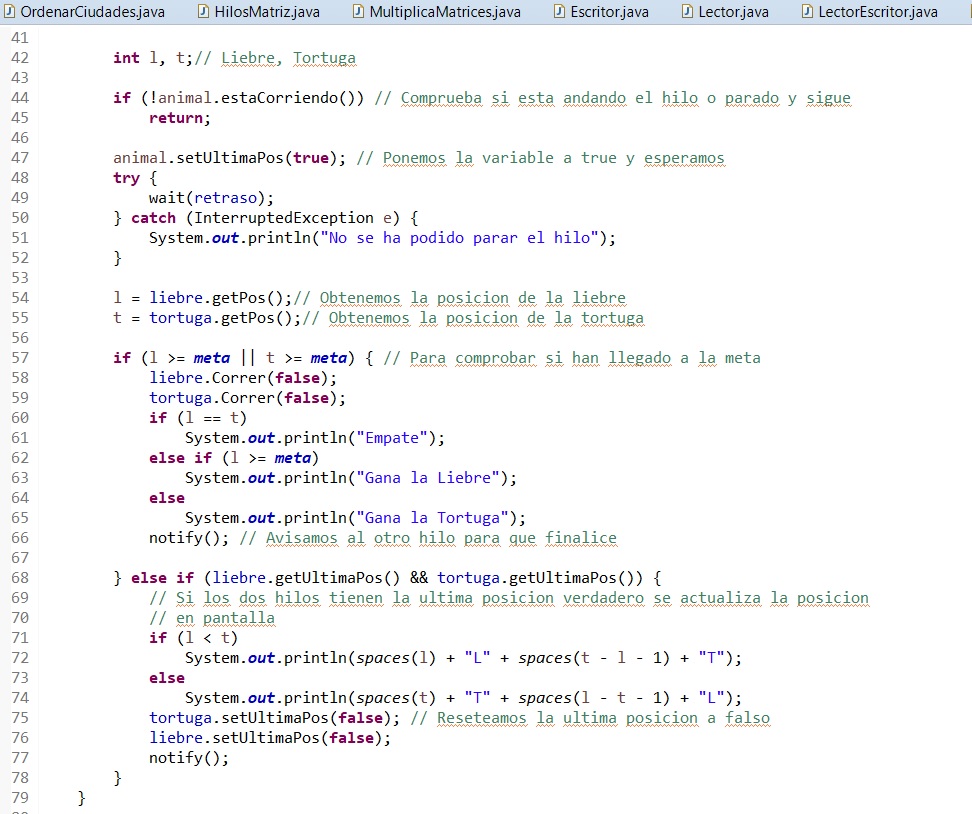
## Uso de comentarios

Uno de los elementos básicos para documentar código, es el uso de comentarios. Un comentario es una anotación que se realiza en el código, pero que el compilador va a ignorar, sirve para indicar a los desarrolladores de código diferentes aspectos del código que pueden ser útiles. En principio, los comentarios tienen dos propósitos diferentes:

* Explicar el objetivo de las sentencias.
* Explicar qué realiza un método, o clase, no cómo lo realiza.

En el caso del lenguaje Java, C# y C, los comentarios, se implementan de forma similar. Cuando se trata de explicar la función de una sentencia, se usan los caracteres **//** seguidos del comentario, o con los caracteres **/\*** y **\*/**, situando el comentario entre ellos: **/\* *comentario* \*/** a continuación unos ejemplos.





## Documentación de clases

Los comentarios de una clase deben comenzar con **/\*\*** y terminar con **\*/**.

Con el uso de los entornos de desarrollo, las etiquetas se añaden de forma automática, estableciendo el **@autor** y la **@version** de la clase de forma transparente al programador‐programadora. También se suele añadir la etiqueta **@see**, que se utiliza para referenciar a otras clases y métodos.

Dentro de la la clase, también se documentan los constructores y los métodos. Al menos se indican las etiquetas:

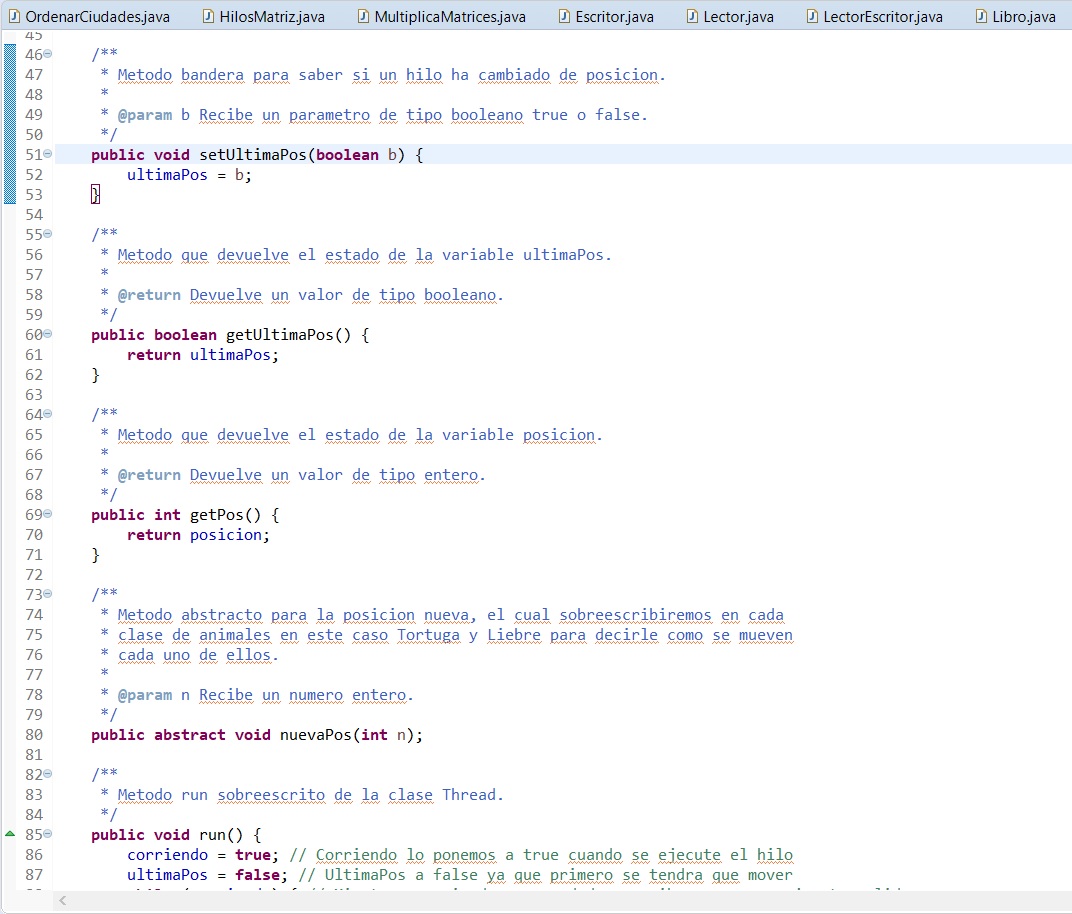
**@param**: seguido del nombre, se usa para indicar cada uno de los parámetros que tienen el constructor o método.

**@return**: si el método no es void, se indica lo que devuelve.

**@exception**: se indica el nombre de la excepción, especificando cuales pueden lanzarse.

**@throws**: se indica el nombre de la excepción, especificando las excepciones que pueden lanzarse.

A continuación unos ejemplos.



## Herramientas

La herramienta que utilizaremos es JavaDoc. Para que JavaDoc pueda generar las páginas HTML es necesario seguir una serie de normas de documentación en el código fuente, estas son:

* Los comentarios JavaDoc deben empezar por /\*\* y terminar por \*/.
* Los comentarios pueden ser a nivel de clase, a nivel de variable y a nivel de método.
* La documentación se genera para métodos public y protected.
* Se puede usar tag para documentar diferentes aspectos determinados del código, como parámetros

Adjunto un .rar con un JavaDoc con el trabajo.