

## UNIDAD 4

### Optimización y documentación

## Contenidos

- 4.1. Refactorización
- 4.2. Analizadores de código
- 4.3. Control de versiones
- 4.4. Documentación

## 4.1. Refactorización

- ❑ Definición: realizar modificaciones en el código con el objetivo de mejorar su estructura interna, sin alterar su comportamiento externo.
- ❑ Razones para refactorizar:



•Calidad

Eficiencia

Evitar la reescritura  
de código

- ❑ Objetivo: conseguir un diseño lo más sencillo posible y de calidad.

### 4.1.1. Los malos olores (*bad smells*)

- ❑ Son los síntomas en el código que aconsejan realizar una refactorización.
- ❑ Clasificación de los malos olores:
  - A nivel de aplicación:
    - Código duplicado (*duplicated code*).
    - Cirugía a tiros (*shotgun surgery*).
    - Complejidad artificial (*contrived complexity*).
  - A nivel de método:
    - Método largo (*long method*).
    - Cadenas de mensajes (*message chains*).
    - Demasiados parámetros (*too many parameters*).
    - Línea de código excesivamente larga (*God line*).
    - Excesiva devolución (*excessive returner*).
    - Tamaño del identificador (*identifier size*).

➤ A nivel de clase:

- Clase larga (*large class*).
- Clase demasiado simple (*freeloader*).
- Envidia de funcionalidad (*feature envy*).
- Código divergente (*divergent code*).
- Grupo de datos (*data clump*).
- Intimidad inapropiada (*inappropriate intimacy*).
- Legado rechazado (*refused bequest*).
- Complejidad ciclomática (*cyclomatic complexity*).

### 4.1.2. Implantación de la refactorización

#### ❑ Dos posibilidades:

- Aplicar refactorización *a posteriori*.
- Refactorización continua.

#### ❑ Buenas prácticas:

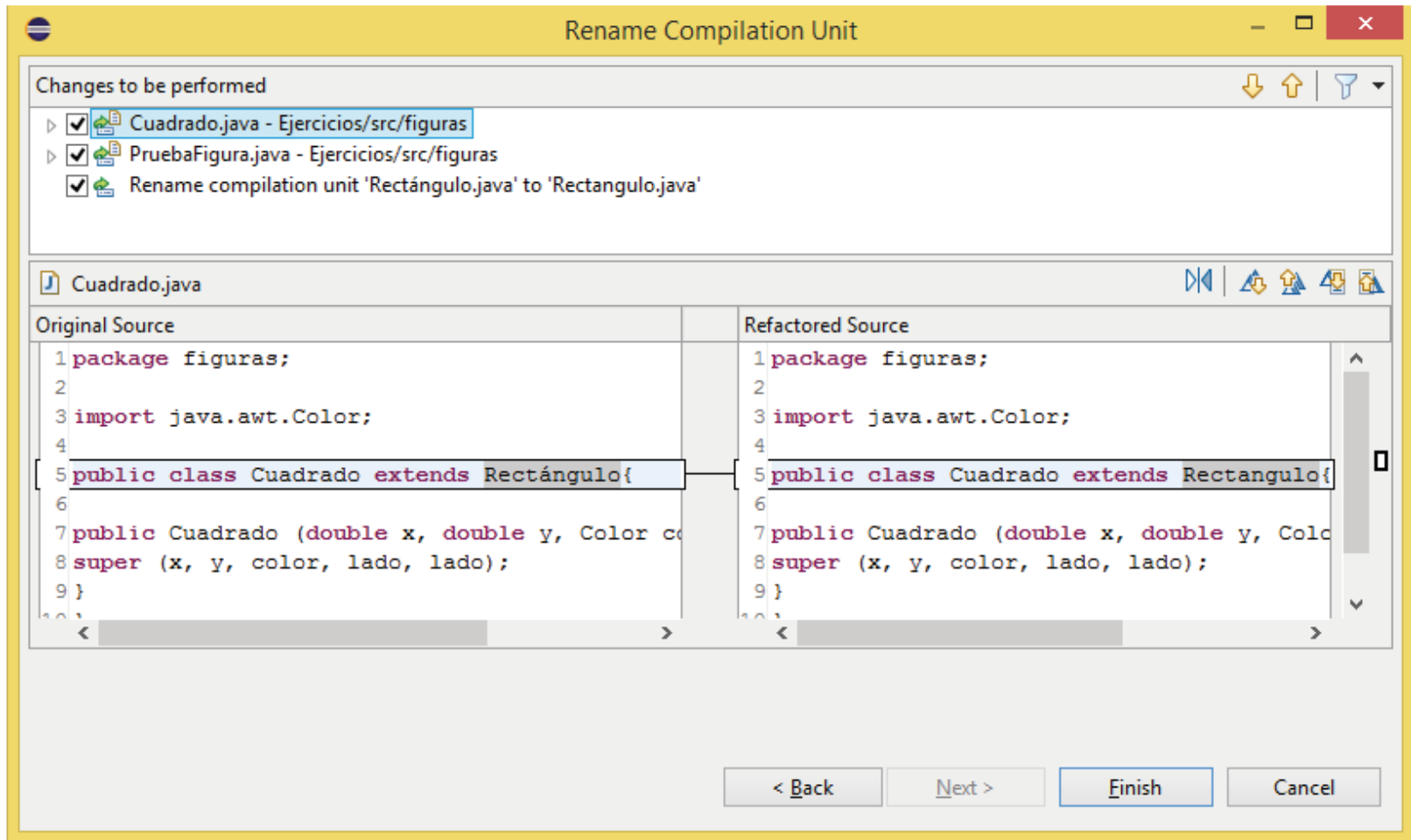
- Hacer pruebas unitarias y funcionales antes de refactorizar.
- Promover y recibir formación sobre patrones de refactorización.
- Usar herramientas especializadas.
- Comenzar refactorizando los principales fallos de diseño.
- Refactorizar tras añadir cada nueva funcionalidad.
- Implantar la refactorización continua.

### 4.1.2. Patrones de refactorización en Eclipse

- La refactorización se puede aplicar sobre una clase, un atributo, una variable, una expresión, un bloque de instrucciones, etcétera.
- Se elige la opción *Refactor* del menú principal o la opción *Refactor* del menú contextual del elemento que se desea refactorizar.

#### *Rename*

- Permite cambiar el nombre de una clase, de un atributo, de un método, de una variable, etcétera.



**Figura 4.1.** Al seleccionar el patrón de refactorización *Rename* de un elemento, se muestran todos los lugares de la aplicación donde ese cambio de nombre tendrá efecto antes de llevarlo a cabo.

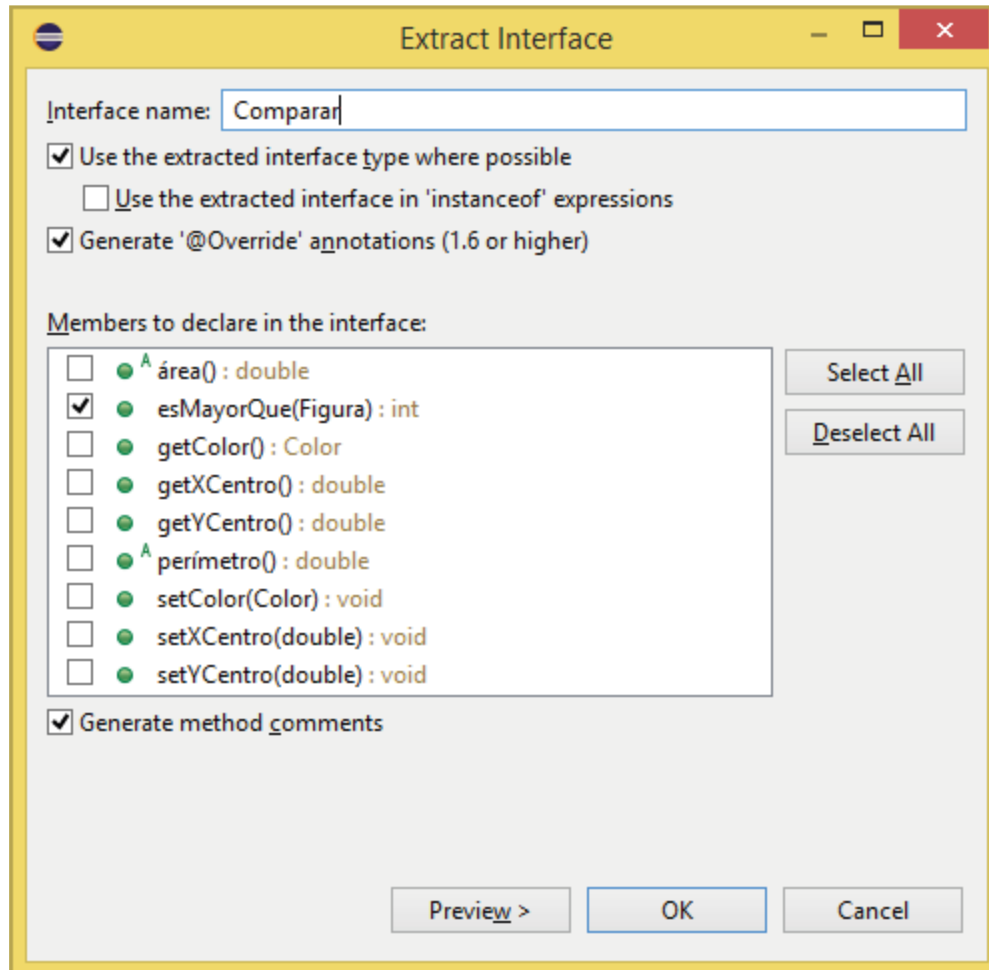


### ***Move***

- Permite mover una clase de un paquete a otro.

### ***Extract Interface***

- Permite extraer ciertos métodos de una clase para crear una interfaz con ellos.



**Figura 4.2.** Al seleccionar el patrón de refactorización *Extract Interface* para una clase, se pide el nombre que se quiere dar a la interfaz y los métodos que se quieren colocar en ella.

### *Extract Superclass*

- Permite crear una superclase con atributos y métodos de una clase.

### *Use Supertype Where Possible*

- Sustituye todas las referencias a una clase por las de su superclase.

### *Pull Up*

- Mueve un atributo o un método de una clase a su superclase.

### *Pull Down*

- Mueve un conjunto de métodos y atributos de una clase a sus subclases.

## Change Method Signature

Change Method Signature

Access modifier: public Return type: double Method name: distancia

Parameters Exceptions

Type	Name	Default value
Punto	p	-

Add Edit... Remove Up Down

☐ Keep original method as delegate to changed method  
☒ Mark as deprecated

Method signature preview:  
public double distancia(Punto p)

✗ Method signature is unchanged.

Preview > OK Cancel

**Figura 4.3.** Al seleccionar el patrón de refactorización *Change Method Signature* para un método, se pueden cambiar todos los elementos definidos en su cabecera: modificador de acceso, nombre del método, tipo de dato que devuelve; nombre, tipo y valor por defecto de sus parámetros; se pueden añadir parámetros o eliminarlos y añadir, eliminar o modificar excepciones que lanza el método.

### *Inline*

Permite escribir en una sola línea la referencia a una variable o a un método y el uso de dicha variable o método.

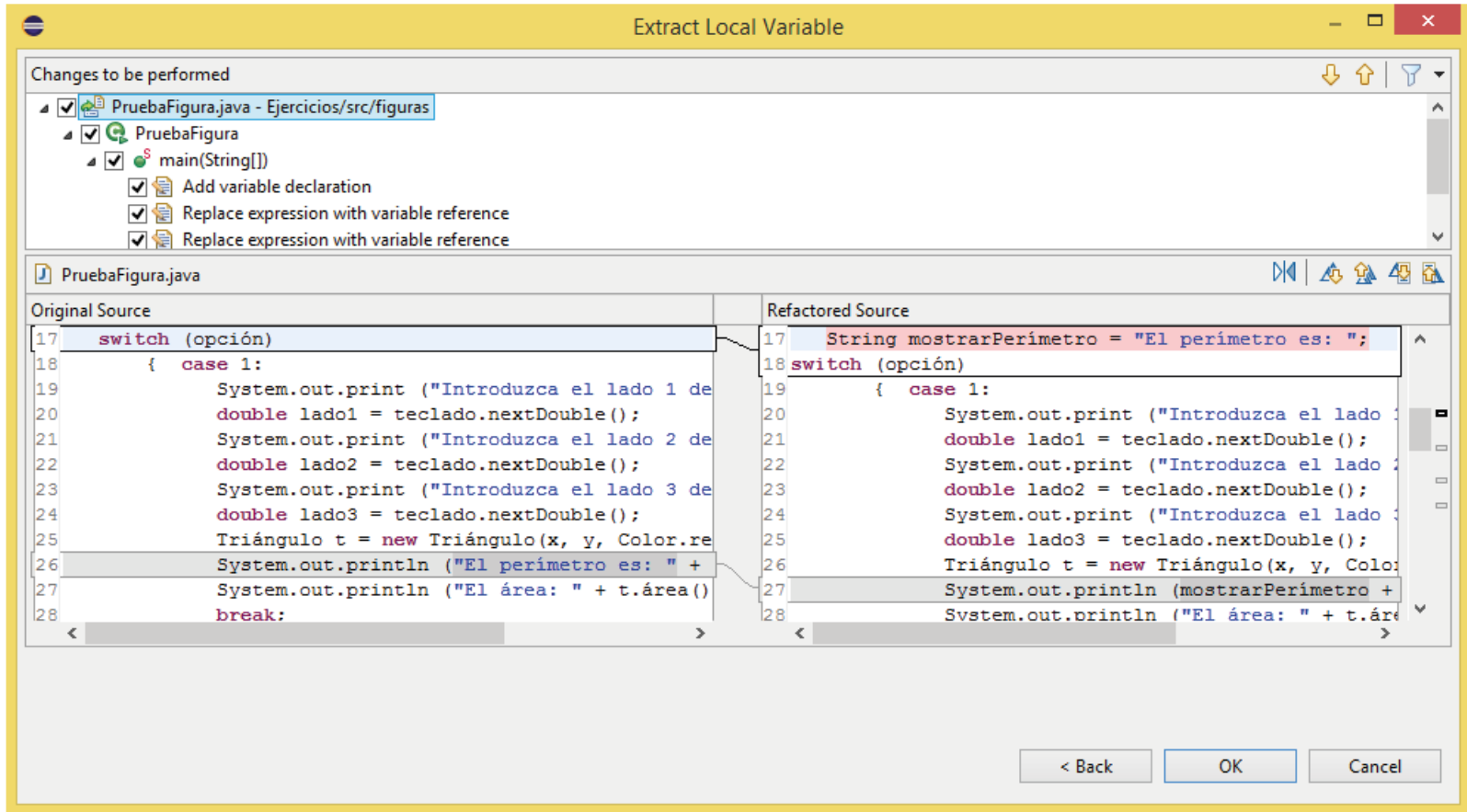
```
1 public Punto simétrico(){
2     Punto nuevoP = new Punto (this.x * -1, this.y);
3     return nuevoP;
4 }
```

```
1 public Punto simétrico(){
2     return new Punto (this.x * -1, this.y);
3 }
```

### *Introduce Parameter Object*

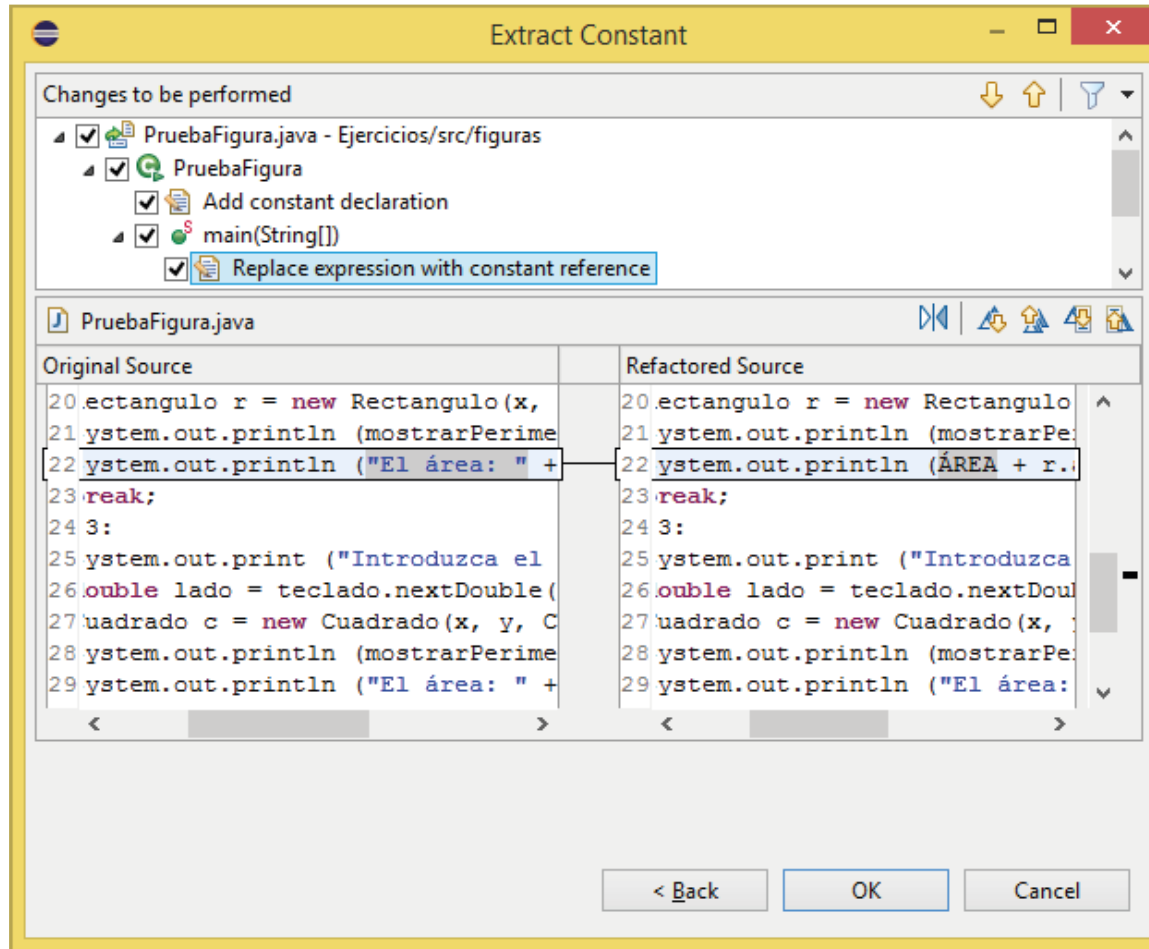
Crea una clase a partir de un conjunto de parámetros de un método.

## Extract Local Variable



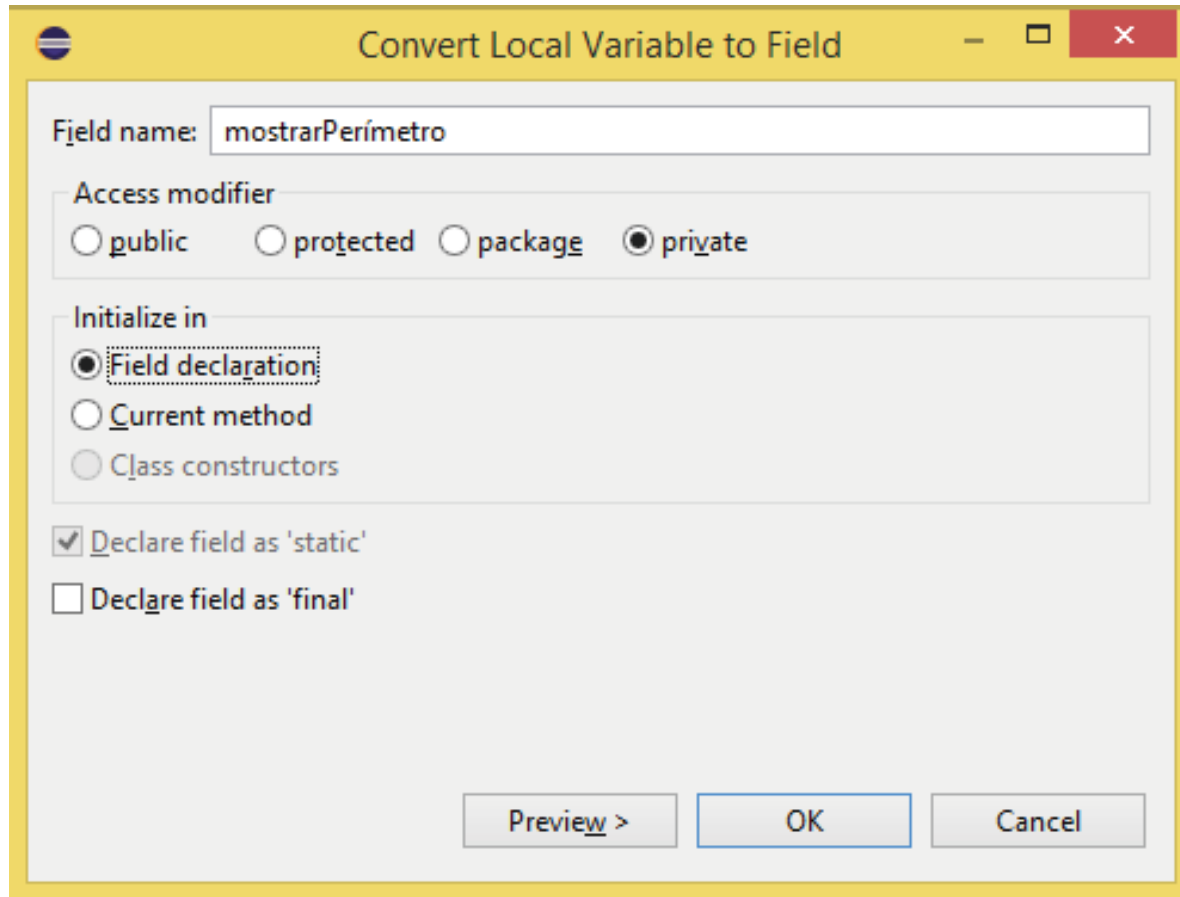
**Figura 4.4.** Al seleccionar el patrón de refactorización *Extract Local Variable* para una expresión, se pueden cambiar todas las ocurrencias de esa expresión por una variable cuyo nombre se debe indicar.

## Extract Constant



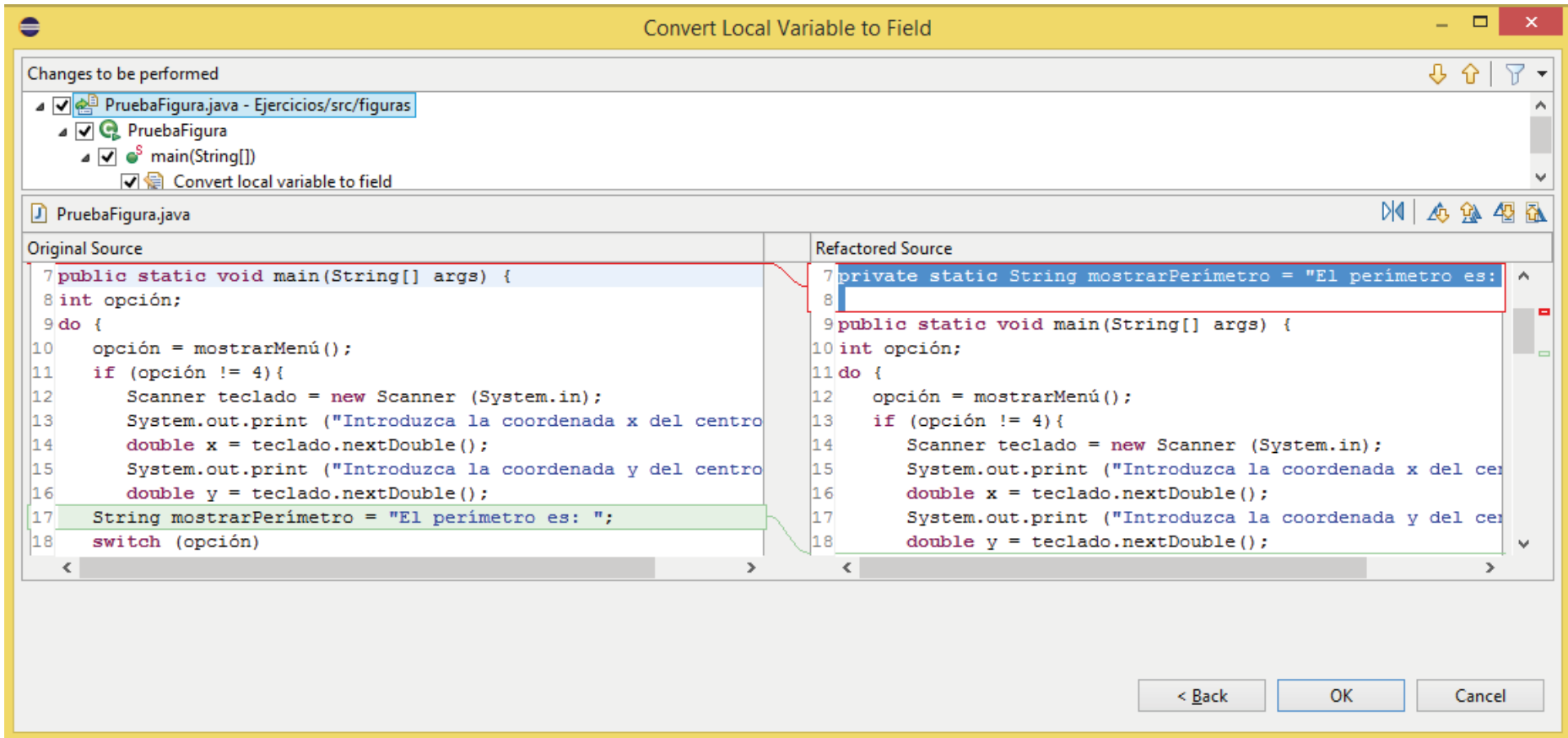
**Figura 4.5.** Al seleccionar el patrón de refactorización *Extract Constant* para una expresión, se pueden cambiar todas las ocurrencias de esa expresión por una constante cuyo nombre se debe indicar.

### *Convert Local Variable to Field*



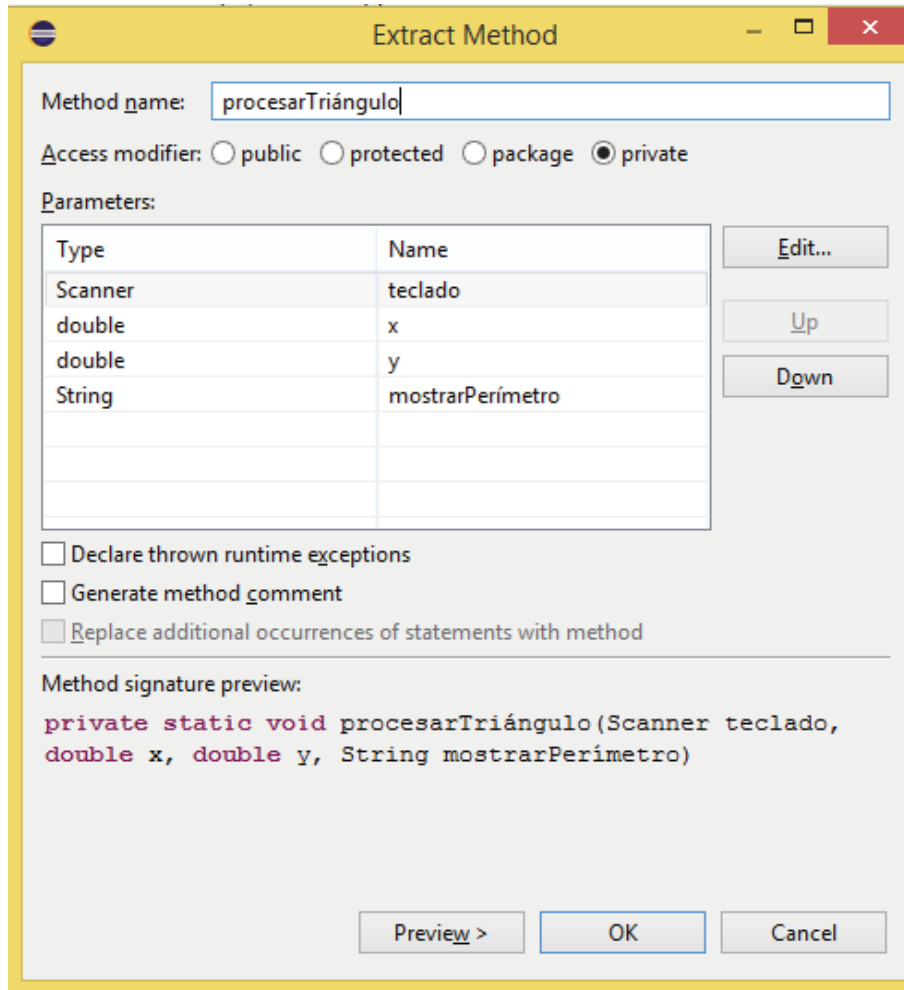
**Figura 4.6.** Al seleccionar el patrón de refactorización *Convert Local Variable to Field* para una variable local, se pueden cambiar todas las ocurrencias de esa variable por un atributo de la clase.





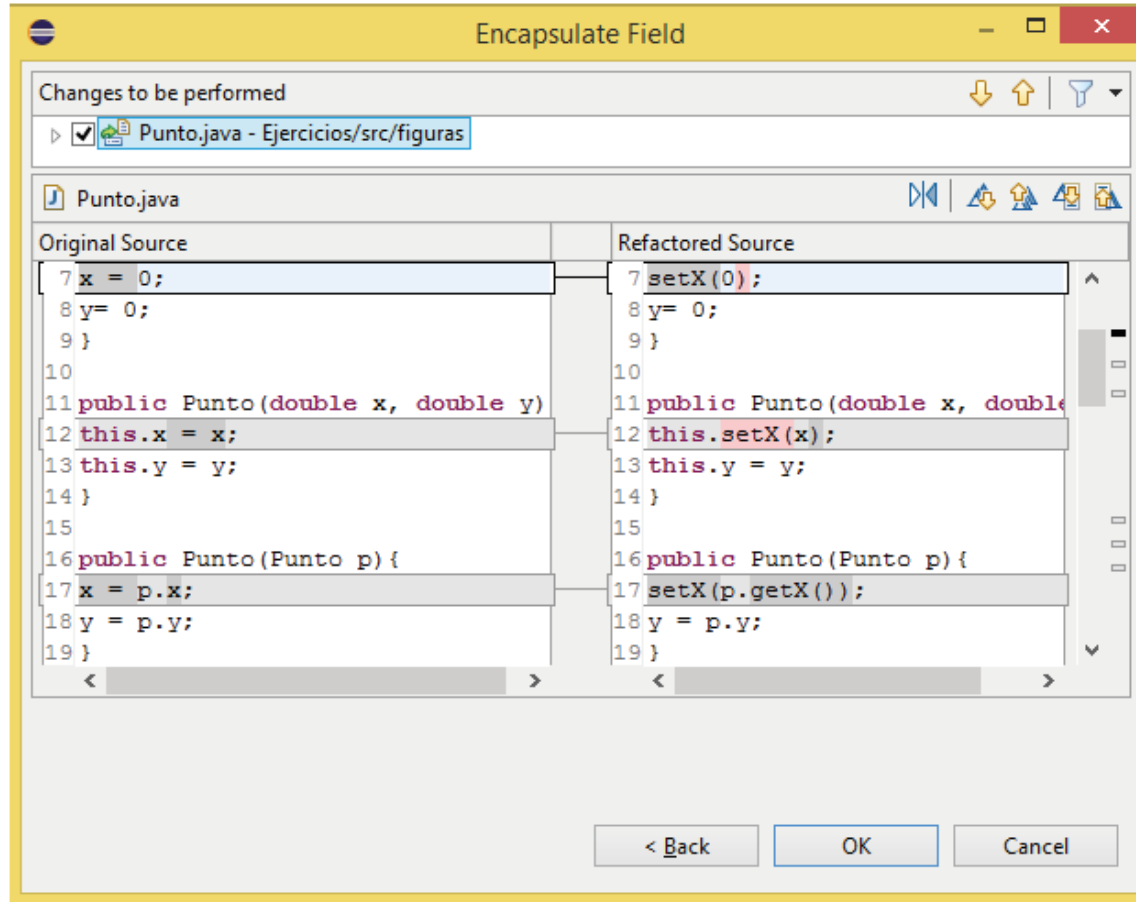
**Figura 4.7.** Ventana que aparece al hacer clic en el botón *Preview* y en la que se visualizan los cambios que supone la conversión de una variable local a un atributo de la clase.

## Extract Method



**Figura 4.8.** Al seleccionar el patrón de refactorización *Extract Method* para un bloque de código, se le da un nombre y un modificador de acceso al método y Eclipse ajusta automáticamente sus parámetros y el valor de retorno.

## Encapsulate Field



**Figura 4.9.** Al seleccionar el patrón de refactorización *Encapsulate Field* para un atributo, se sustituyen todos los usos de este atributo por una llamada al método get correspondiente y todas las asignaciones de valor a dicho atributo por una llamada al método set que corresponda.

### 4.1.3. Otras operaciones de refactorización en Eclipse

- Almacenar en un *script* las refactorizaciones realizadas:

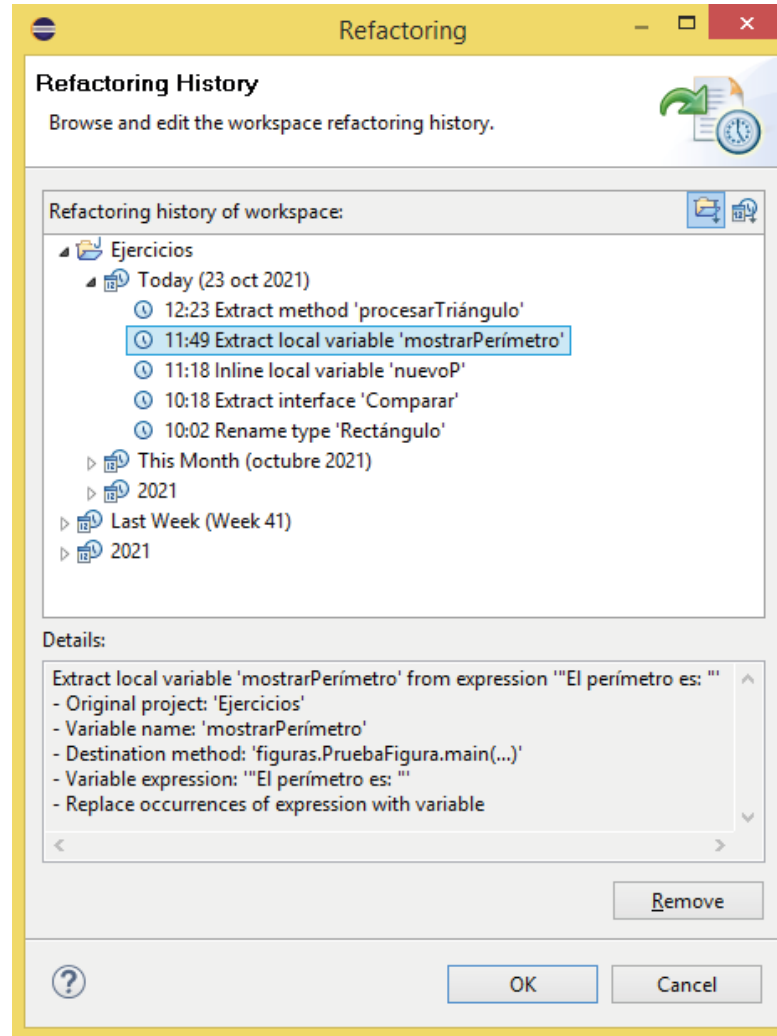
opción de menú Refactor → Create Script

- Cargar un script de refactorizaciones para su aplicación:

opción de menú Refactor → Apply Script

- Consultar las refactorizaciones realizadas:

opción de menú Refactor → History



**Figura 4.10.** Al seleccionar la opción de menú *Refactor > History*, se muestra un histórico de refactorizaciones realizadas. Al colocarse sobre cada una de ellas, se obtiene información detallada.

## 4.2. Analizadores de código

- ❑ Analizan el código fuente y si detectan que es mejorable, lo indican, y proponen la manera de mejorarlo.
- ❑ PMD es un analizador de código gratuito y de código abierto. Puede detectar los siguientes errores:

•Variables,  
métodos y  
parámetros no  
utilizados

Bloques vacíos de  
sentencias

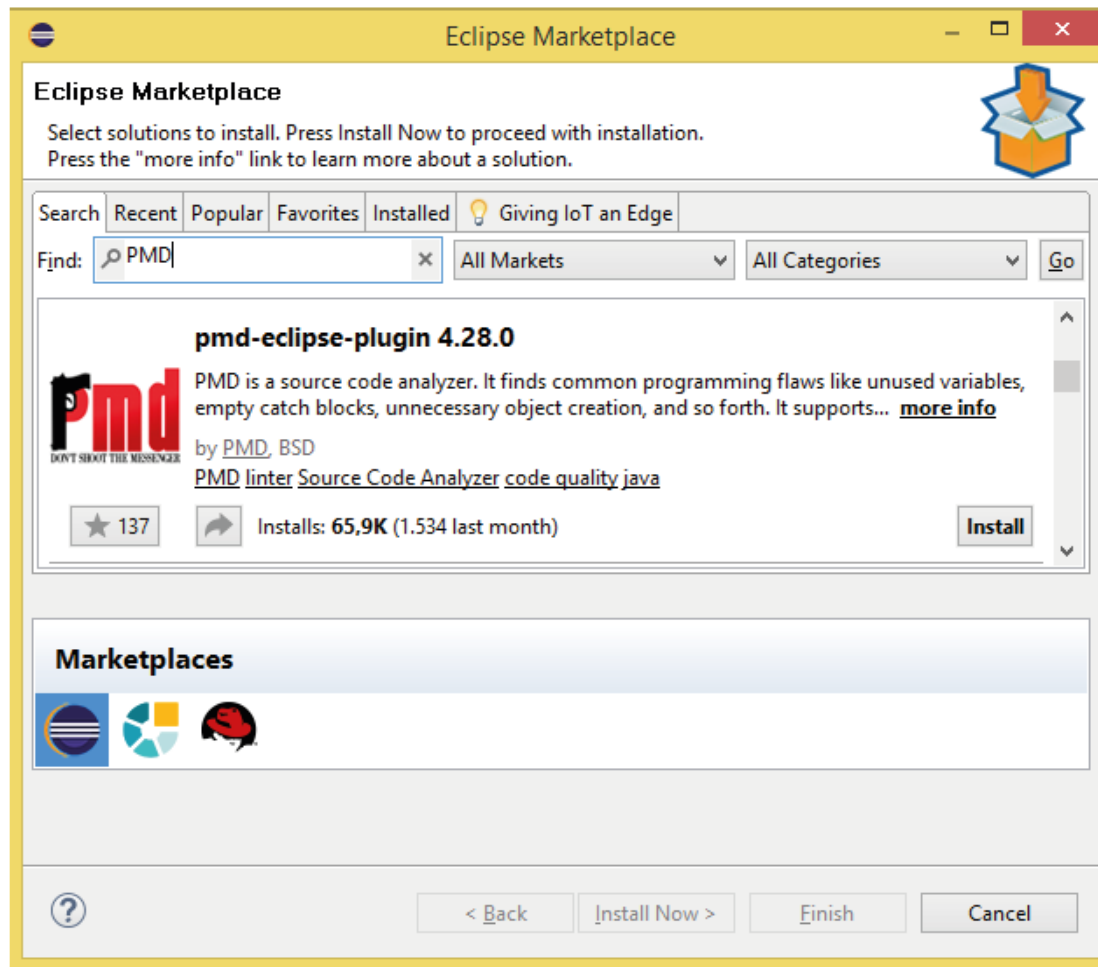
Expresiones  
lógicas que se  
pueden simplificar

Código  
inalcanzable

Código duplicado

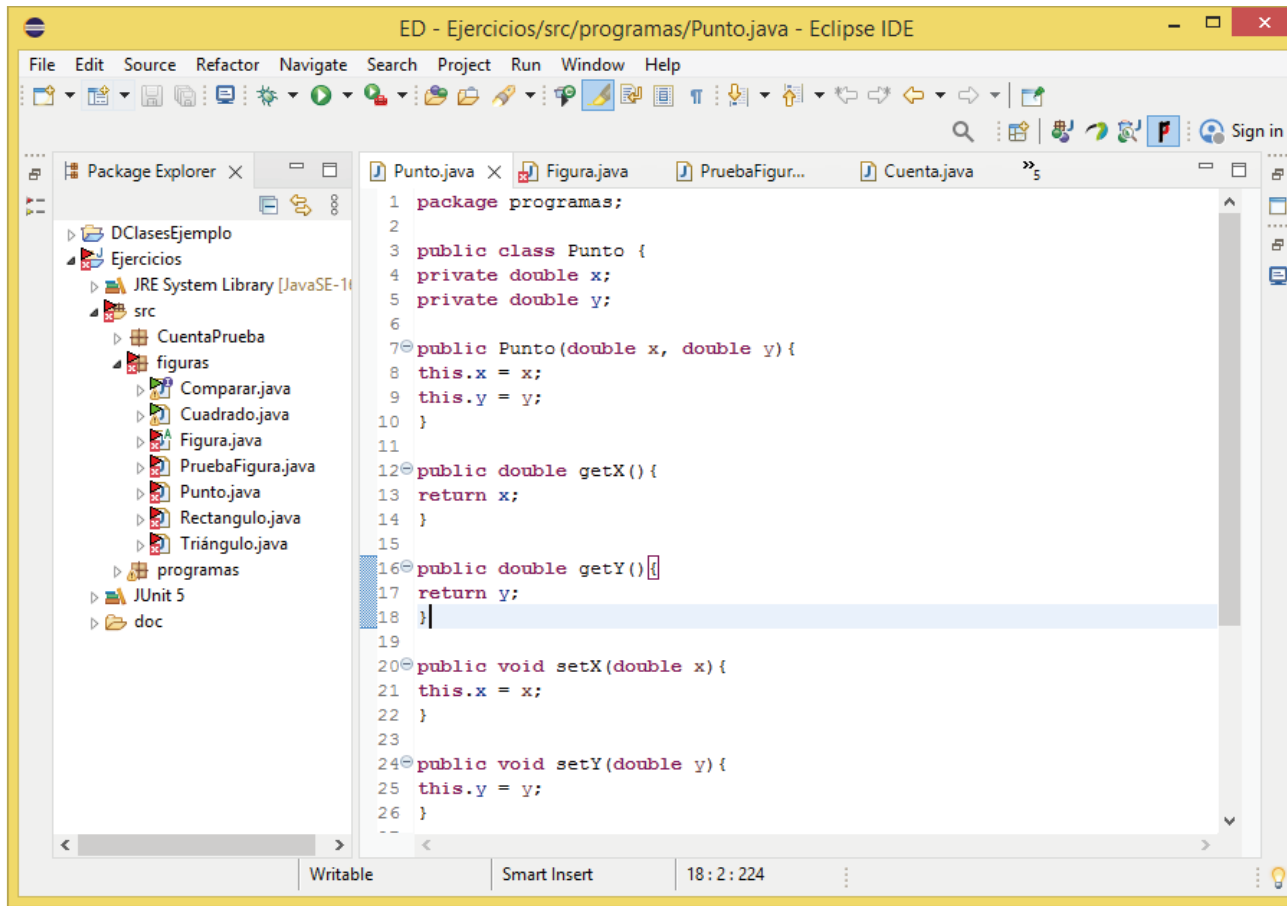
Clases con elevada  
complejidad  
ciclomática

- ❑ Se puede instalar PMD como un módulo en Eclipse.



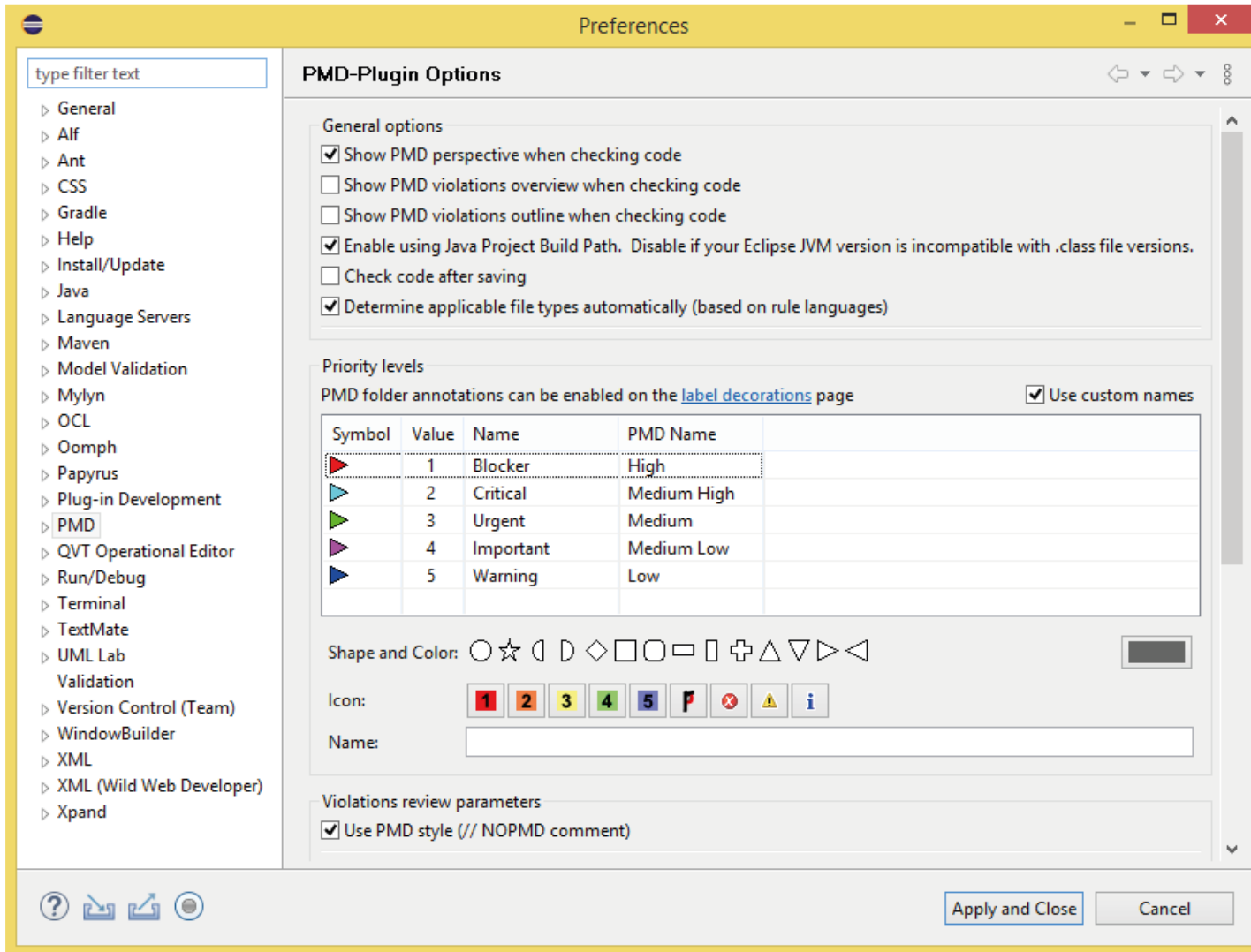
**Figura 4.11.** Ventana que muestra el módulo PMD en Eclipse. Para instalar este analizador de código, se clicla sobre el botón *Install*.

Para analizar el código, se elige del menú contextual la opción PMD → Check code.

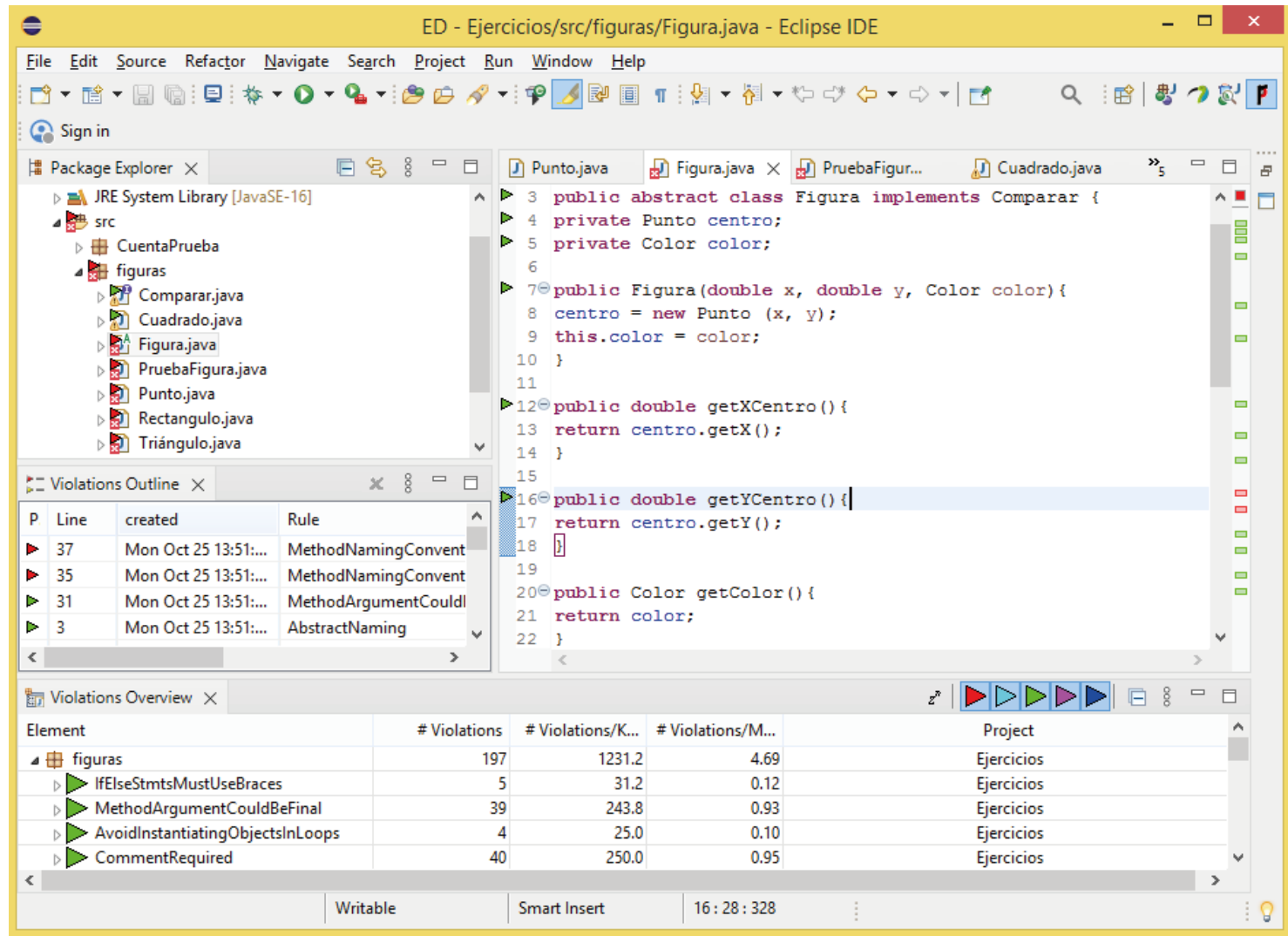


**Figura 4.12.** Ventana que muestra, para el proyecto figuras, el resultado del análisis de código realizado por PMD. Se indica por cada clase y mediante un color distinto si se ha detectado algún defecto.

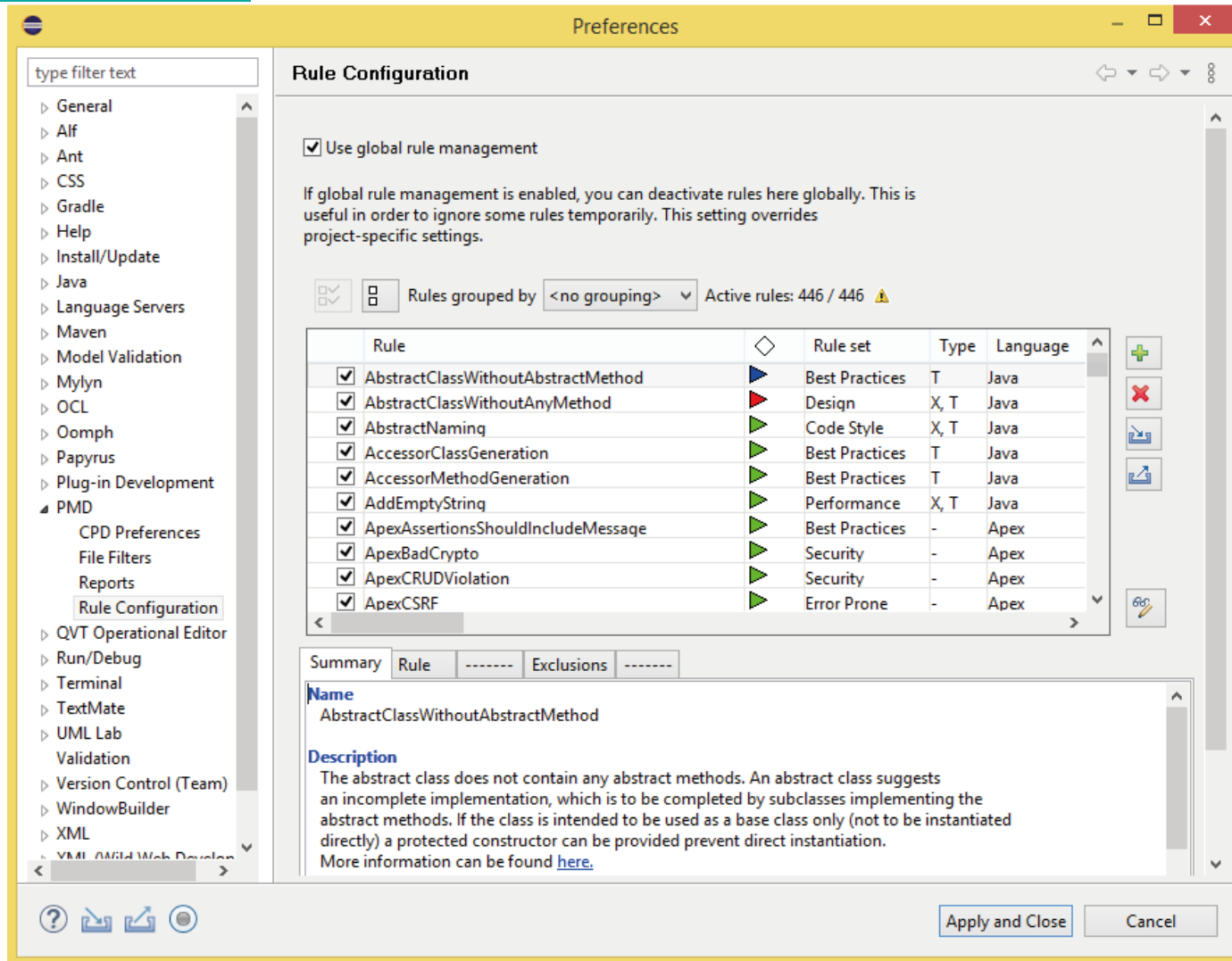




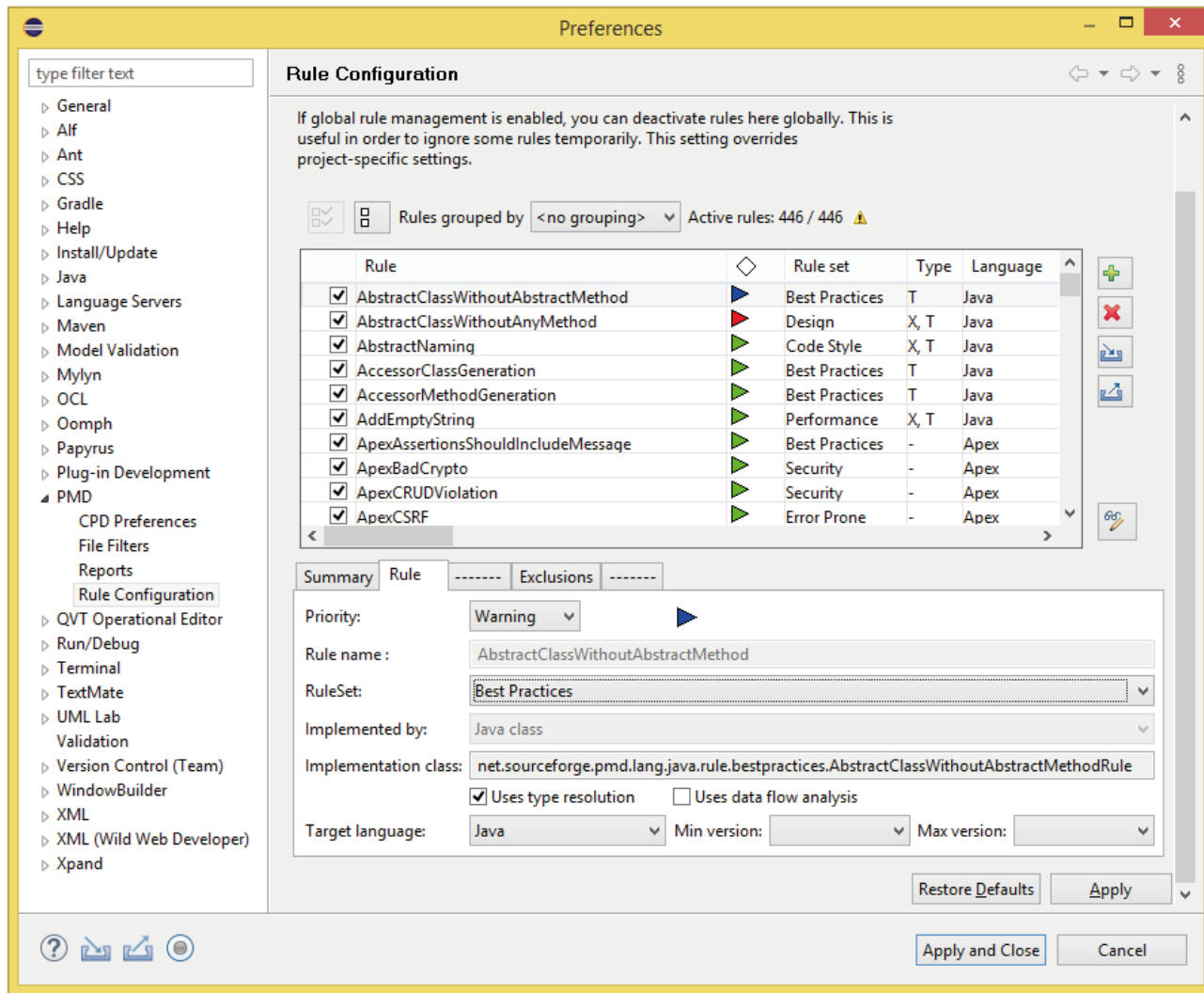
**Figura 4.13.** Ventana que muestra la configuración de PMD en Eclipse, que permite ver los distintos colores que se usan para identificar los defectos detectados por PMD.



**Figura 4.14.** Resultado del análisis de código de un proyecto en la vista PMD, incluyendo dos nuevas áreas en la pantalla, que muestran información detallada sobre los defectos hallados.

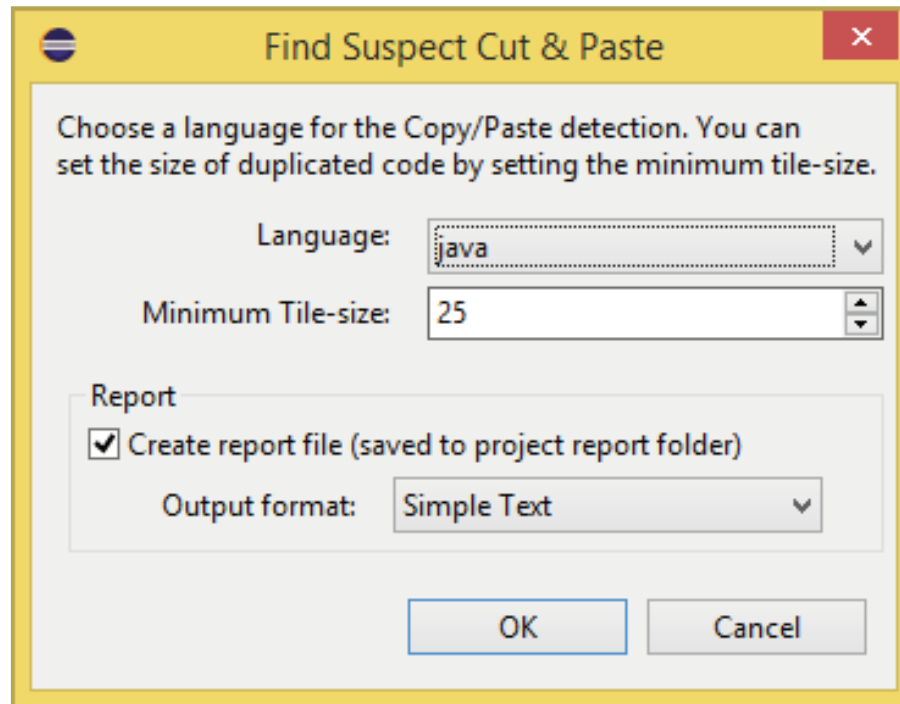


**Figura 4.15.** Ventana que muestra las reglas que emplea PMD al analizar el código. Si se activa la casilla de verificación de la parte superior, se pueden añadir reglas, borrar las existentes o modificar las propiedades de las reglas.



**Figura 4.16.** Ventana en la que se pueden visualizar y modificar las propiedades de una regla concreta en la parte inferior (pestaña *Rule*), haciendo clic en el botón *Apply*.

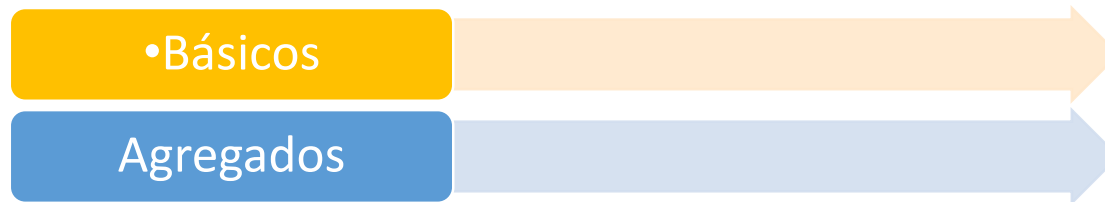
PMD incorpora un detector de código duplicado llamado CPD. Para usarlo, se debe seleccionar del menú contextual de un proyecto la opción PMD → *Find Suspect Cut and Paste*



**Figura 4.17.** Ventana que se muestra al lanzar el detector de código duplicado CPD que lleva incorporado el analizador de código PMD.

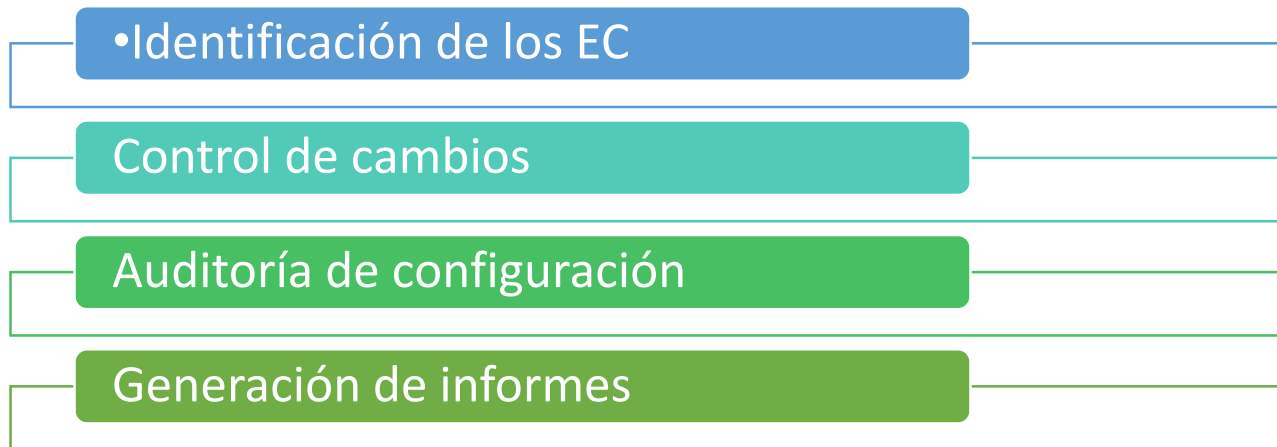
## 4.3. Control de versiones

- ❑ Objetivos de la gestión de la configuración del software (GCS) para cada cambio:
  - Identificar el cambio.
  - Controlar el cambio.
  - Garantizar que el cambio se implemente bien.
  - Informar del cambio a todos los afectados.
- ❑ El software está formado por varios elementos de configuración (EC). Estos pueden ser:



- ❑ Un EC puede adoptar distintas formas (versiones).

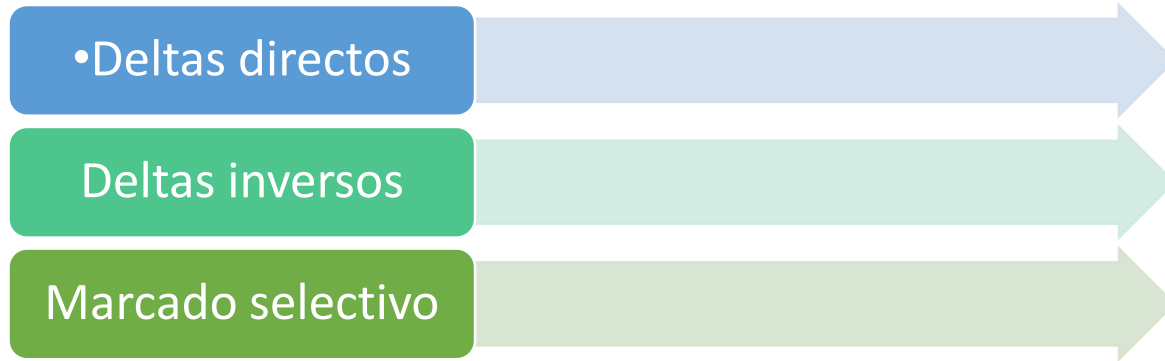
- ❑ Para la GCS se usan herramientas de control de versiones, que hacen uso de un repositorio.
- ❑ Actividades de la GCS:



### 4.3.1. Gestión de versiones

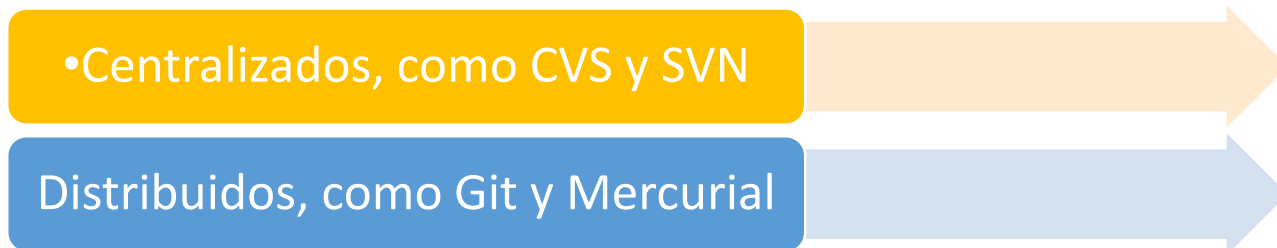
- ❑ Representación de las versiones:
  - Grafo de evolución simple.
  - Árbol.

- Formas de registro de los cambios que suponen las nuevas versiones:

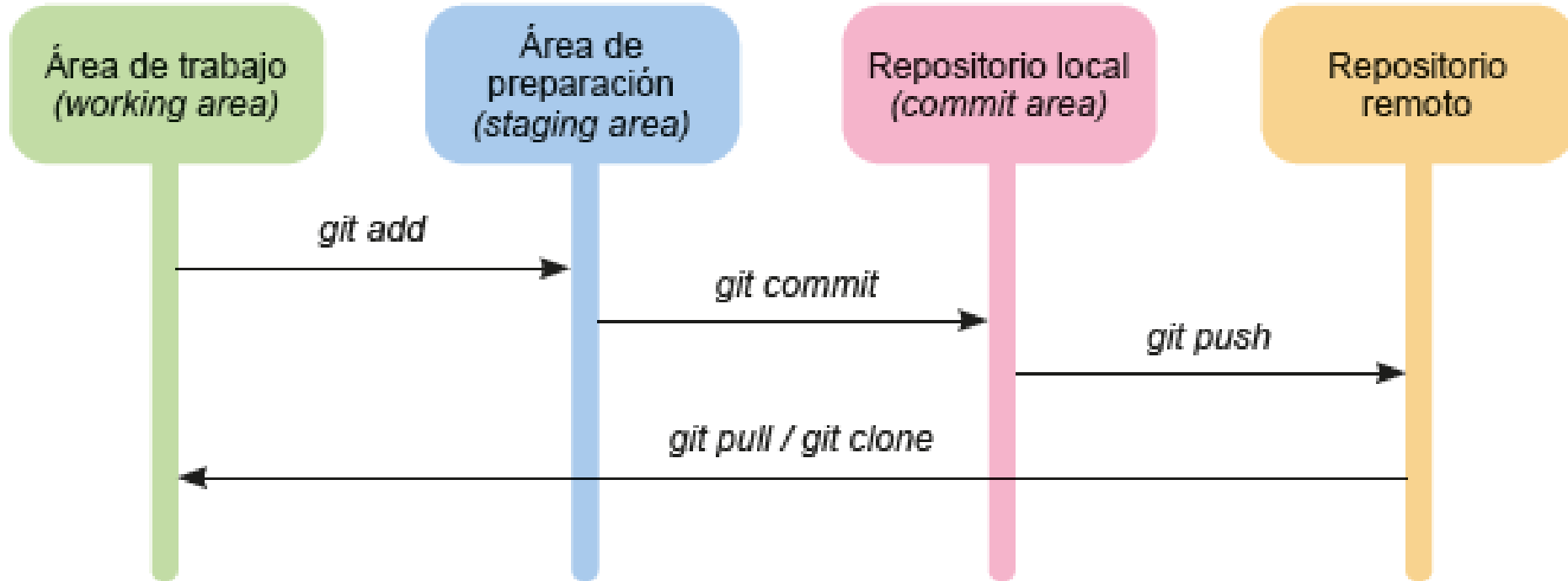


### 4.3.2. Herramientas de control de versiones

- Sistemas de control de versiones:



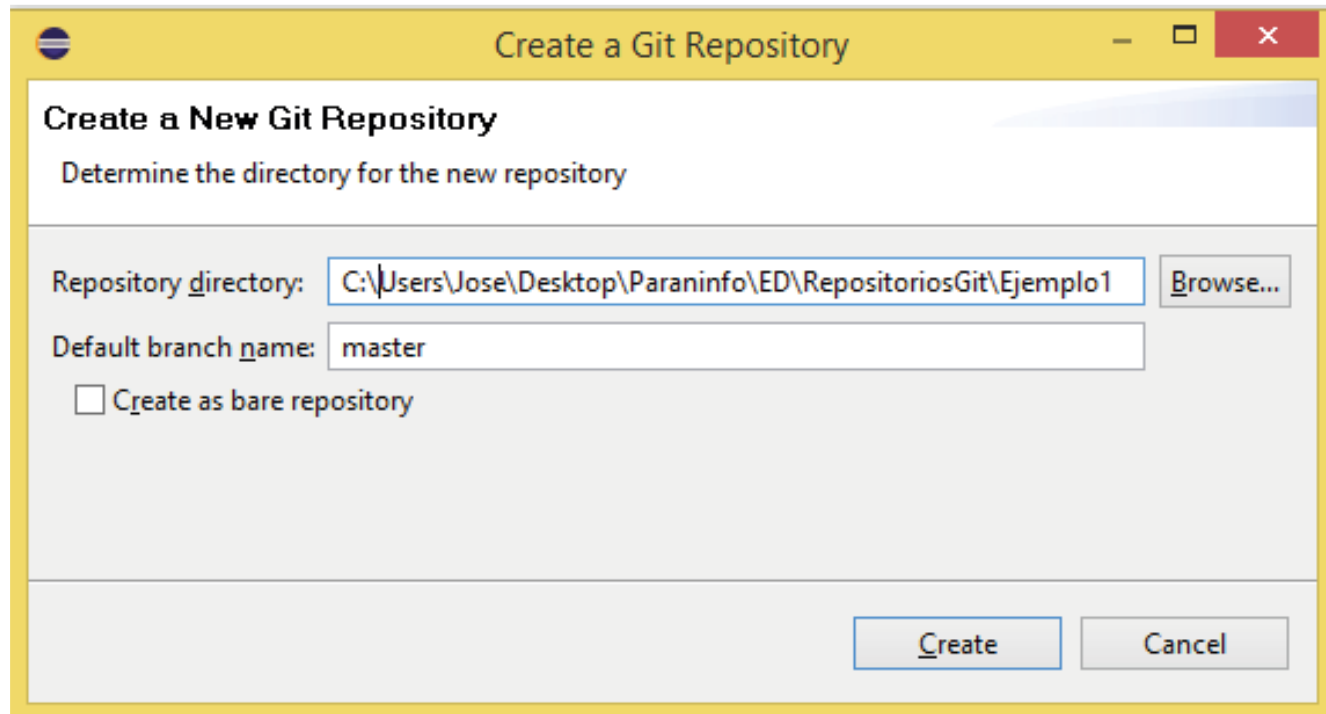


*Git*

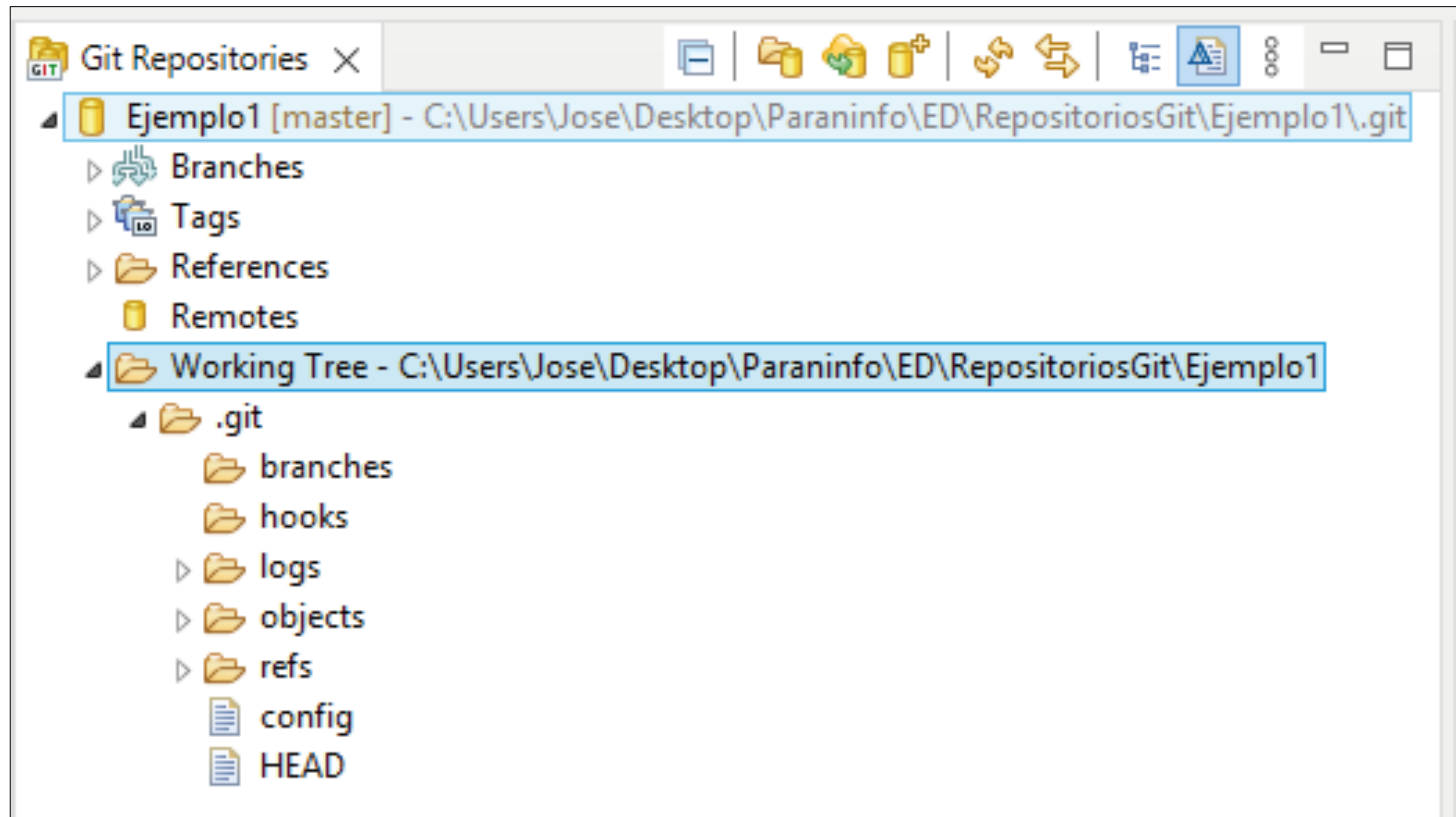
**Figura 4.18.** Diferentes ubicaciones por las que puede pasar un archivo gestionado con Git hasta que llega al repositorio remoto y los comandos que permiten el paso de una ubicación a otra.

## Creación de un repositorio local

En Eclipse se abre la vista Git mediante la opción de menú Window → Perspective → Open Perspective → Other → Git



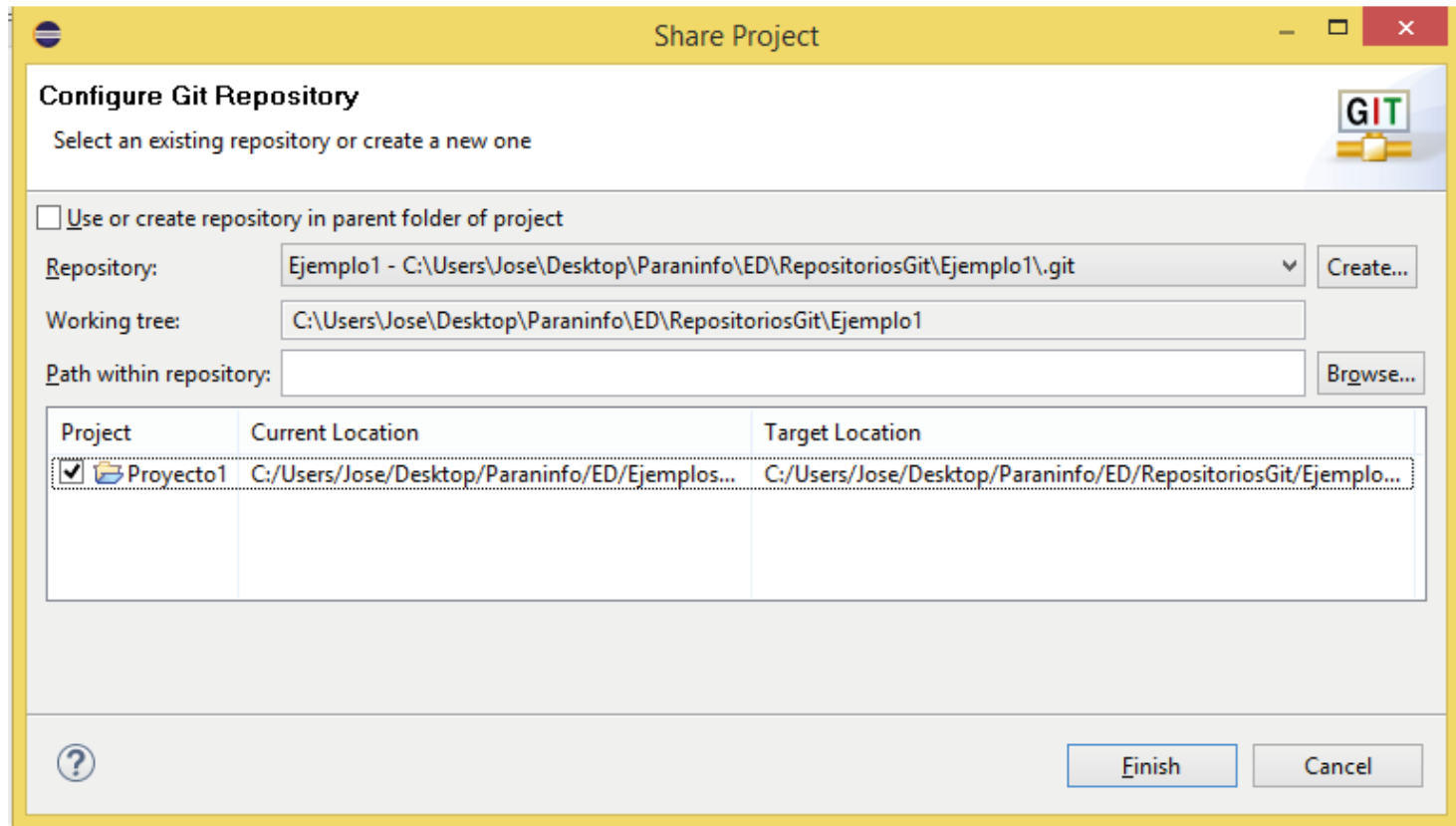
**Figura 4.19.** Ventana en la que se selecciona la ruta del repositorio y se le asigna un nombre. Asimismo, se crea la rama principal o tronco del repositorio, a la que Git llama por defecto con el nombre de *master*.



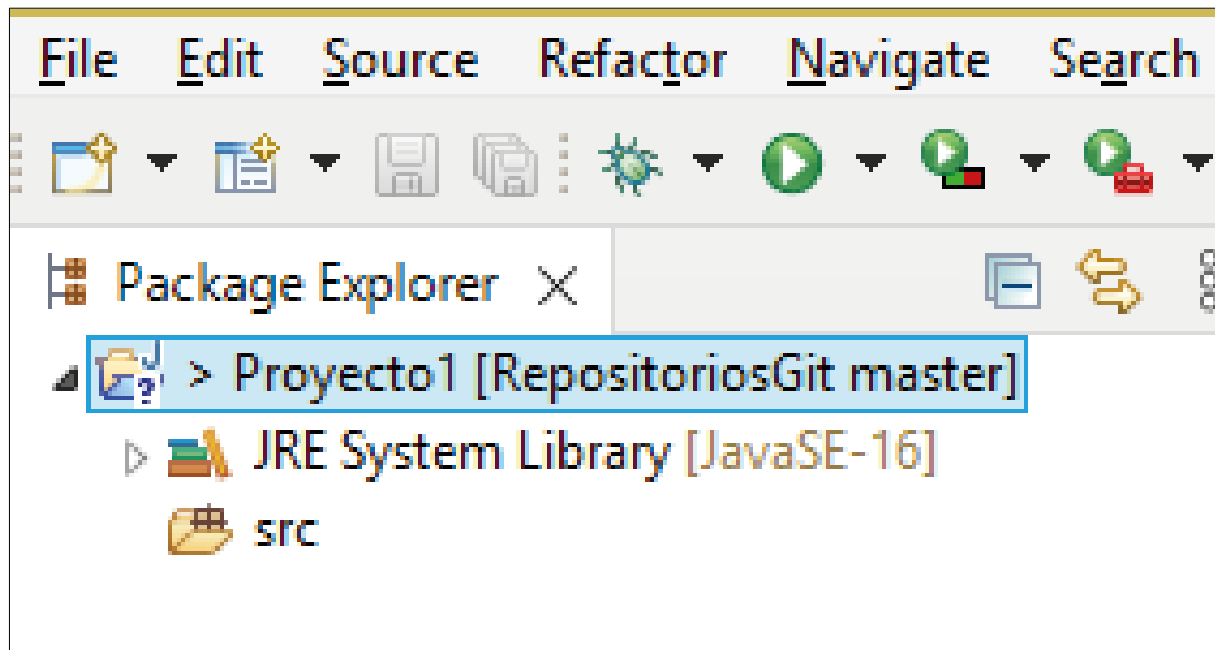
**Figura 4.20.** Estructura del repositorio que se acaba de crear y que se muestra en la parte izquierda de la pantalla.

## Creación de un proyecto en Java

Tras crear el proyecto en Eclipse, se asocia al repositorio local eligiendo en el menú contextual la opción *Team/Share Project...*

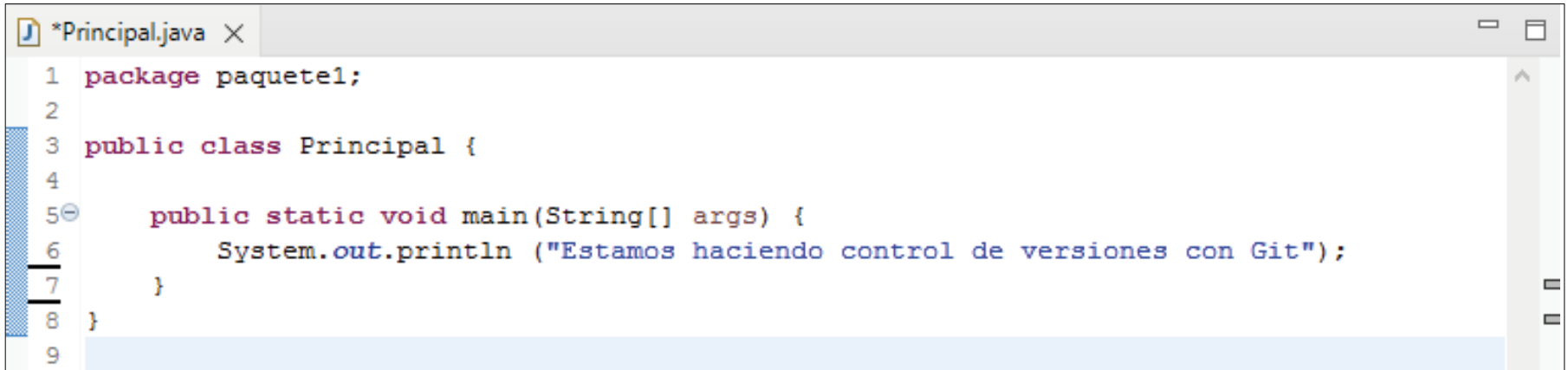


**Figura 4.21.** Al seleccionar el repositorio creado anteriormente, el proyecto queda asociado a él.



**Figura 4.22.** Se muestra el proyecto que se acaba de crear en el explorador de paquetes de la vista Java. El símbolo > antes del nombre del proyecto indica que está bajo el control de versiones de Git.

Se crea dentro del proyecto un paquete y una clase.

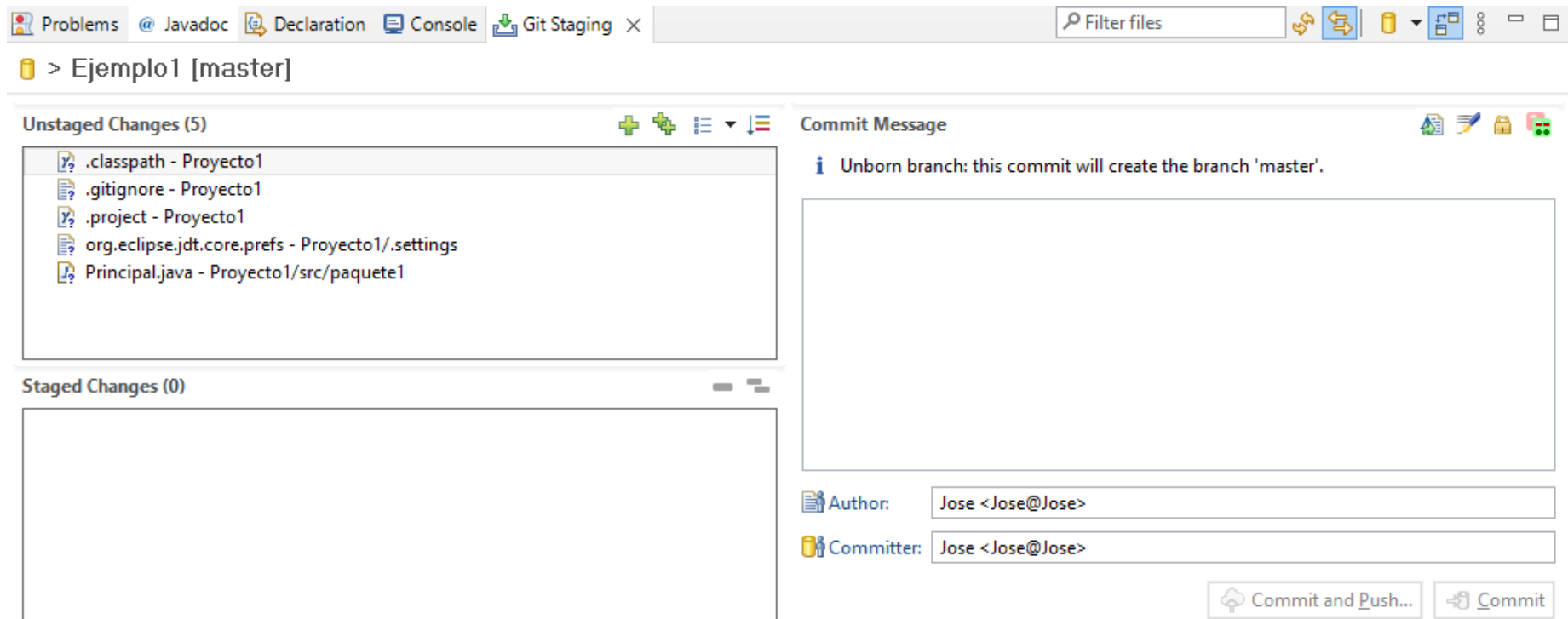


```
*Principal.java X
1 package paquete1;
2
3 public class Principal {
4
5     public static void main(String[] args) {
6         System.out.println ("Estamos haciendo control de versiones con Git");
7     }
8 }
9
```

**Figura 4.23.** Contenido de la clase creada para el proyecto de prueba *Proyecto1* con el que se van a controlar las versiones a través de la herramienta Git.

## Validación de cambios en el repositorio local

Se selecciona del menú contextual de cualquier elemento del proyecto la opción de menú Team → Commit...



**Figura 4.24.** Pantalla desde la que se pueden seleccionar los cambios que se desean confirmar (*commit*) para su paso al área de preparación. Para ello, Eclipse obliga a escribir un mensaje en el cuadro de texto *Commit Message*.

Problems @ Javadoc Declaration Console Git Staging History X

File: Proyecto1/src/paquete1/Principal.java [Ejemplo1]

Id	Message	Author	Authored Date	Committer	Committed Date
360e0e5	<b>master</b> HEAD Segundo commit	Jose	1 seconds ago	Jose	1 seconds ago
ece9397	Primera confirmación	Jose	12 hours ago	Jose	12 hours ago

commit 360e0e570b8f985130ef5e5ed693c66b0b78bc42  
 Author: Jose <Jose@Jose> 2021-12-05 09:34:54  
 Committer: Jose <Jose@Jose> 2021-12-05 09:34:54  
 Parent: [ece9397faalc4741b0078937627ea49544627635](#) (Primera confirmación)  
 Branches: [master](#)

Segundo commit

```
diff --git a/Proyecto1/src/paquete1/Principal.java b/Proyecto1/src/paquete1/Principal.java
index 72b6f83..572a4cb 100644
--- a/Proyecto1/src/paquete1/Principal.java
+++ b/Proyecto1/src/paquete1/Principal.java
@@ -4,5 +4,6 @@

 public static void main(String[] args) {
     System.out.println ("Estamos haciendo control de versiones con Git");
+    System.out.println ("Escribo otra línea para hacer un 2º commit ");
 }
```

Proyecto1/src/paquete1/Principal.java

**Figura 4.25.** Seleccionando la opción del menú contextual *Team > Show in History*, se puede visualizar el historial de versiones de un fichero. Se puede navegar por las distintas versiones y observar las modificaciones que incorpora cada versión en relación con la anterior.



The screenshot displays an IDE window titled 'Java Source Compare'. It compares a local version of 'Principal.java' with a version from commit 'ece9397' by 'Jose'. The code is as follows:

```
1 package paquete1;
2
3 public class Principal {
4
5     public static void main(String[] args) {
6         System.out.println ("Estamos haciendo control de versiones co
7         System.out.println ("Escribo otra línea para hacer un 2º com
8     }
9 }
```

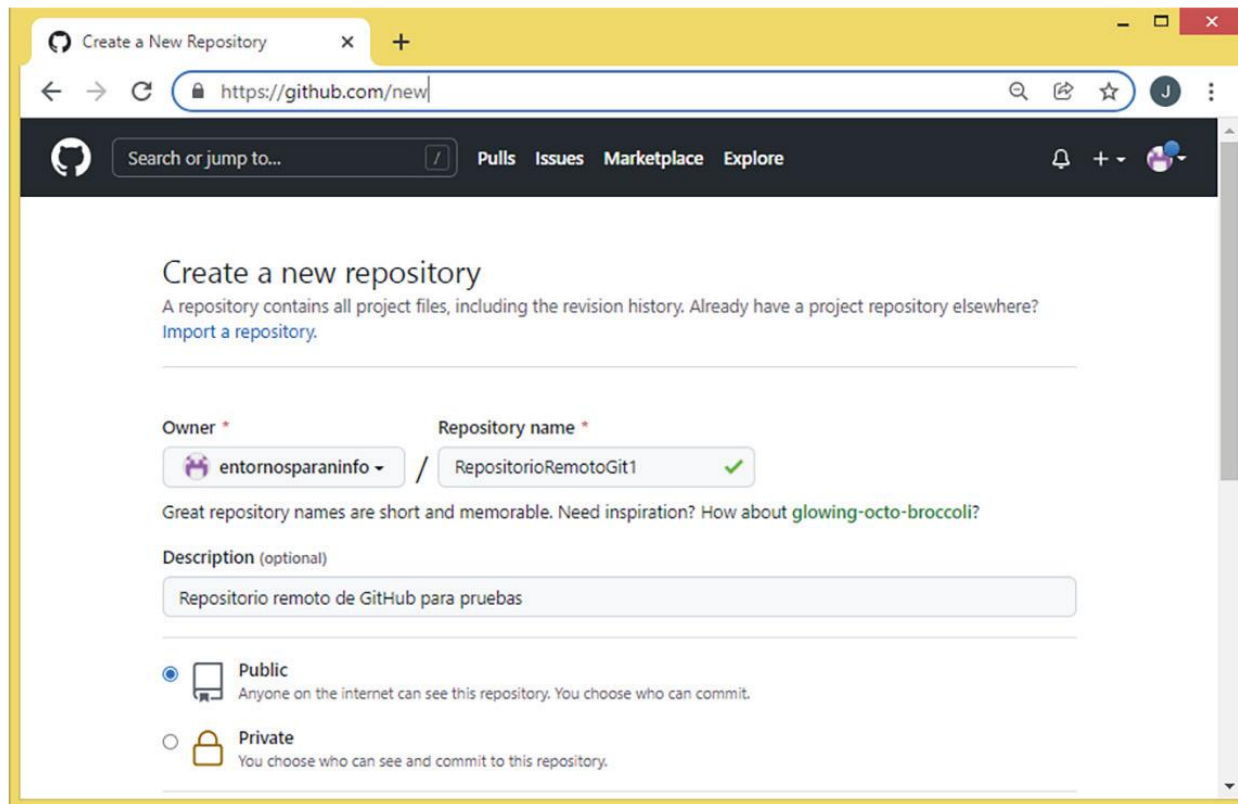
Below the code editor, the 'History' tab is active, showing a table of commit history:

Id	Message	Author	Authored Date	Committer	Committed Date
360e0e5	<b>master</b> (HEAD) Segundo commit	Jose	1 seconds ago	Jose	1 seconds ago
ece9397	Primera confirmación	Jose	12 hours ago	Jose	12 hours ago

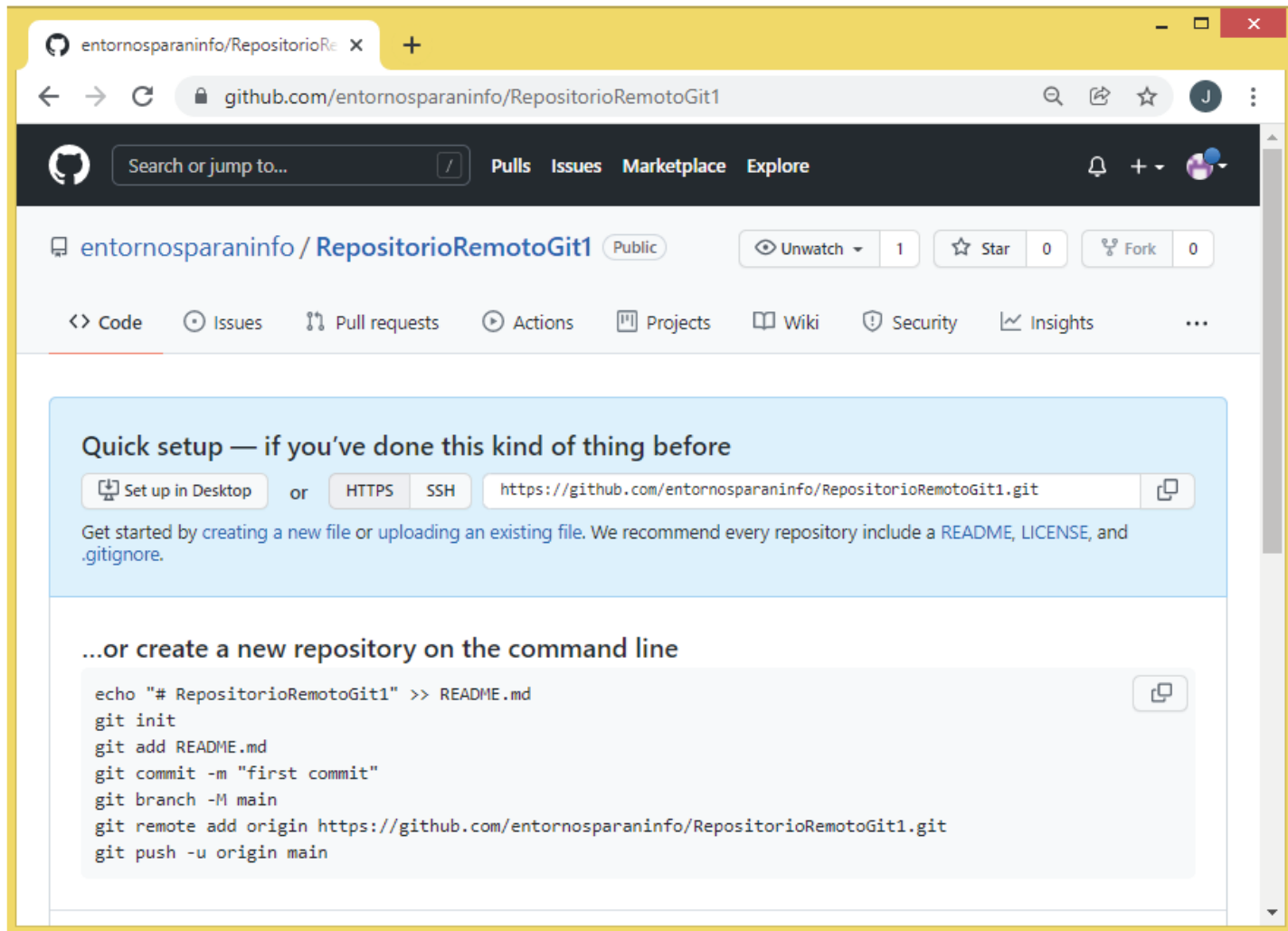
**Figura 4.26.** Historial de versiones de un fichero: si se hace doble clic sobre uno de los *commits*, se puede ver en detalle los cambios realizados en esa versión.

## Creación de un repositorio remoto en GitHub

<https://github.com/>

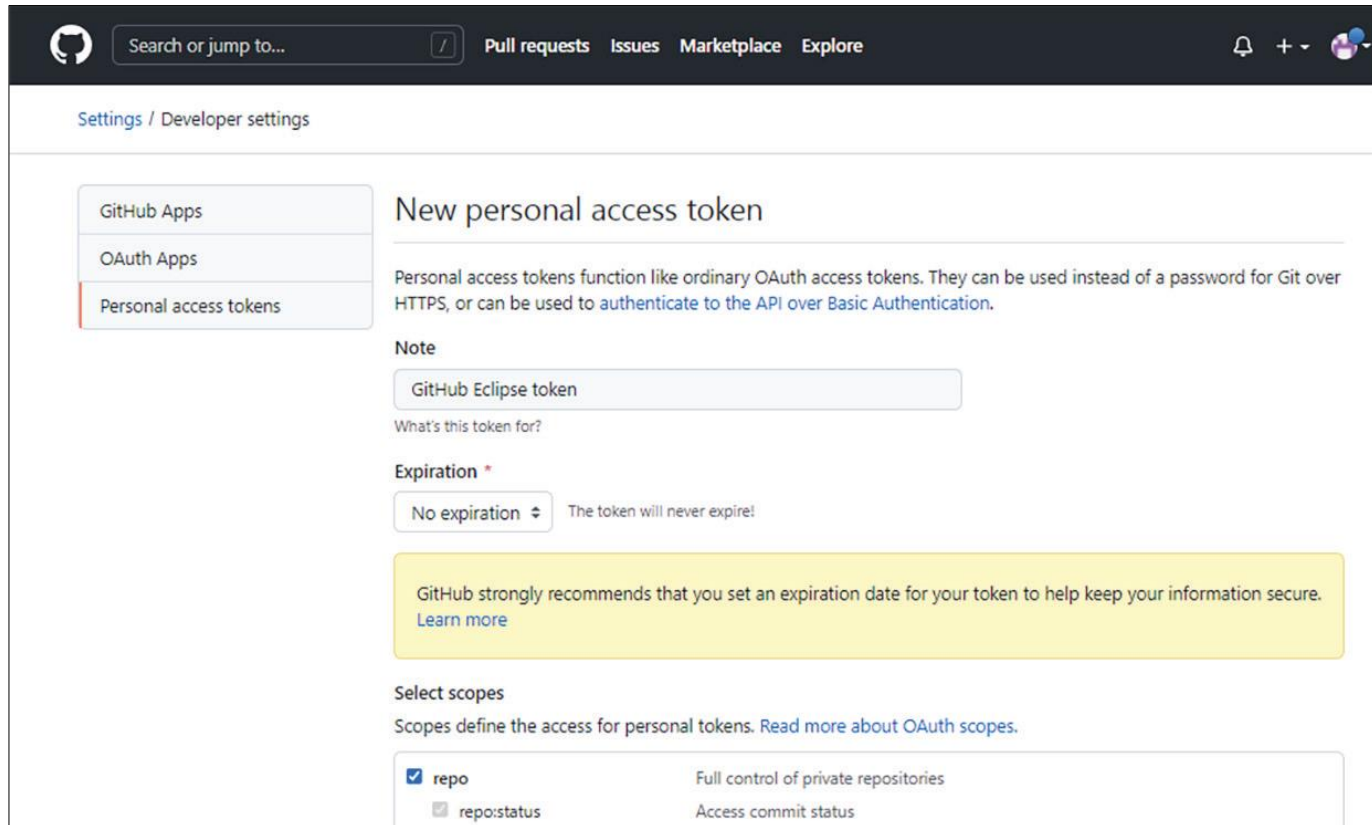


**Figura 4.27.** Vista de la plataforma GitHub en la que, para crear un repositorio remoto, se debe indicar el nombre, que el repositorio sea público y, opcionalmente, una descripción.



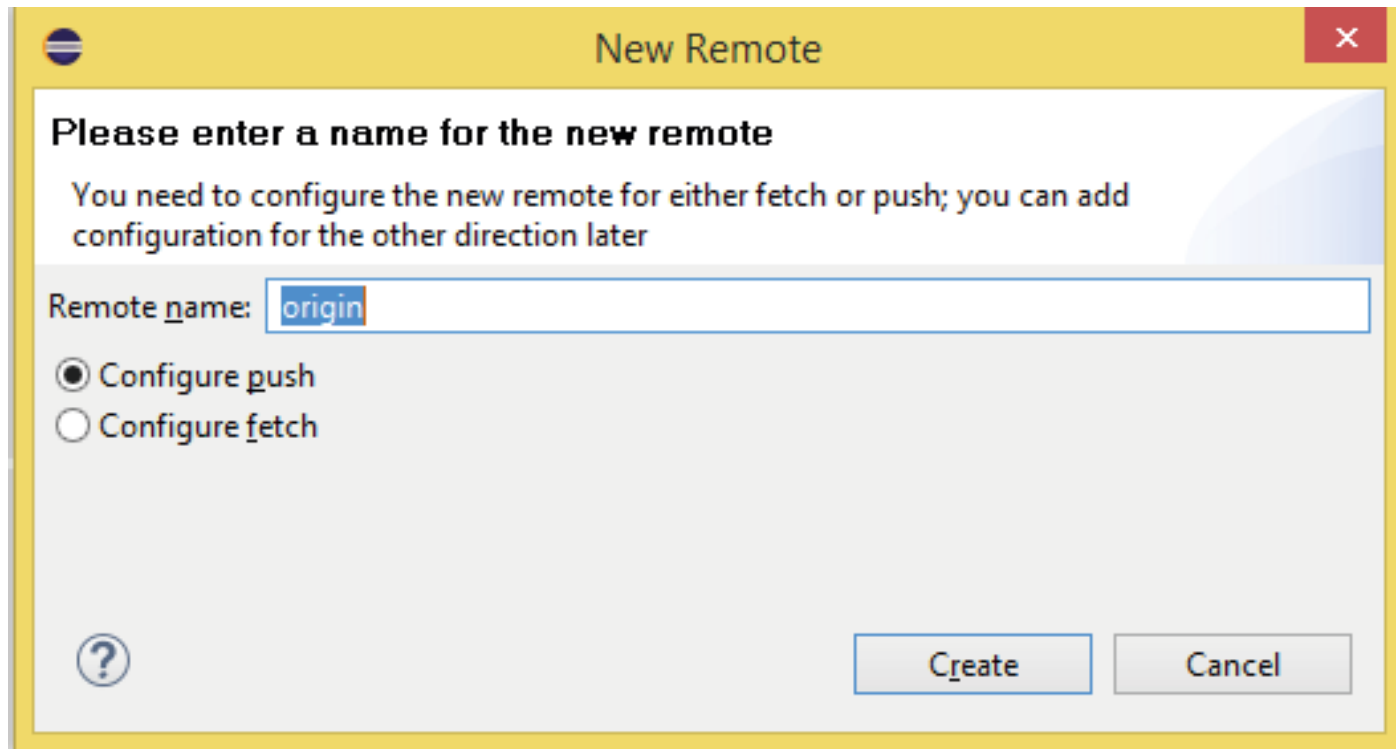
**Figura 4.28.** Pantalla que aparece tras haber creado un repositorio remoto en GitHub. Un dato muy relevante es la URL del repositorio creado, que se puede leer en la parte superior de la pantalla.

Para crear un identificador o *token* de acceso personal se elige la opción de menú *Settings* y luego se clicla sobre el enlace *Developer settings*.

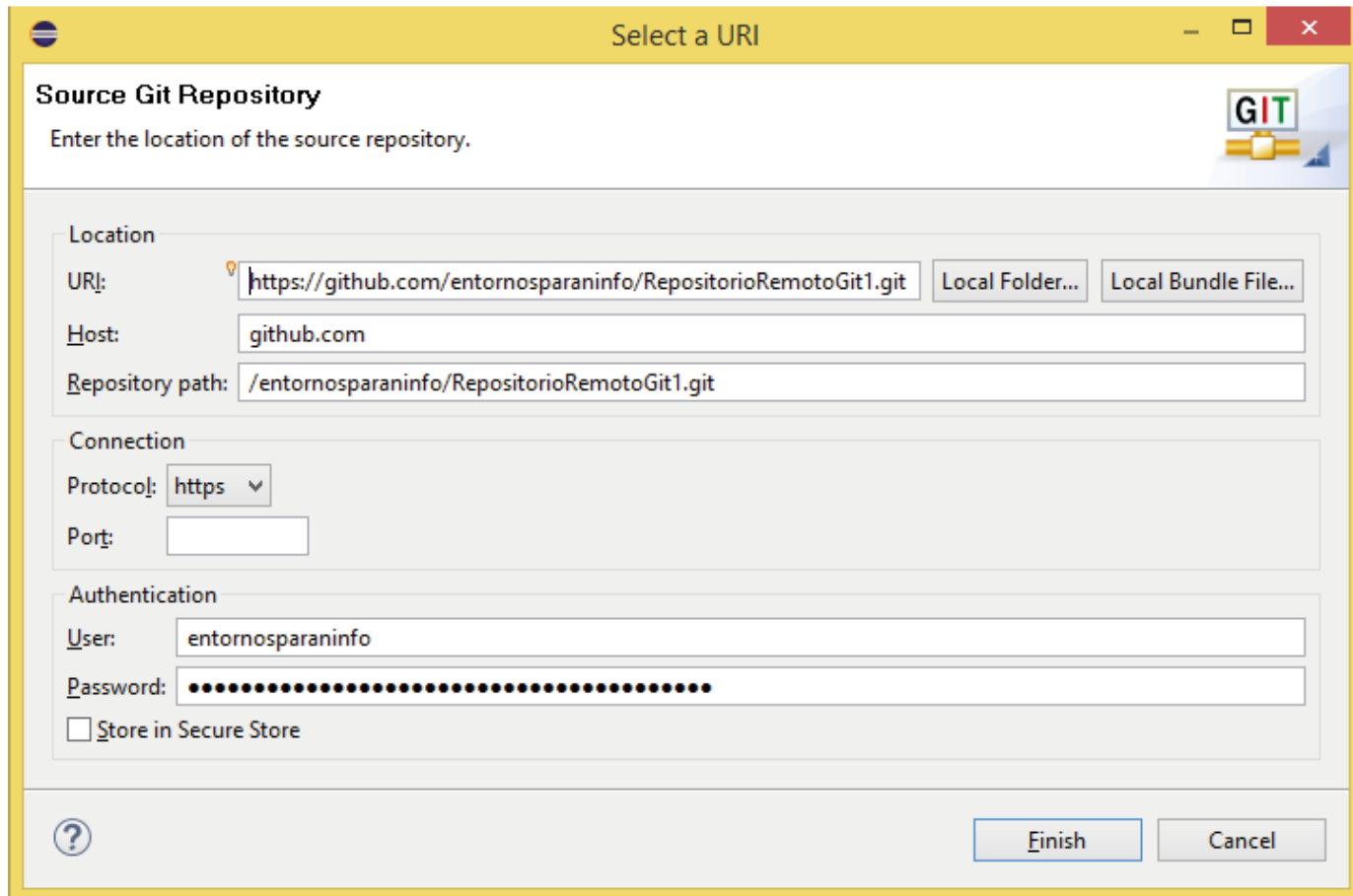


**Figura 4.29.** Creación de un token de acceso personal en GitHub para poder subir ficheros al repositorio remoto. Se indica para qué se va a usar y que no expire nunca. El *token* se creará al hacer clic en el botón *Generate token*.

En Eclipse se debe ir a la vista *Git Repositories* y en el menú contextual del elemento *Remotes* se debe elegir la opción *Create Remote...*



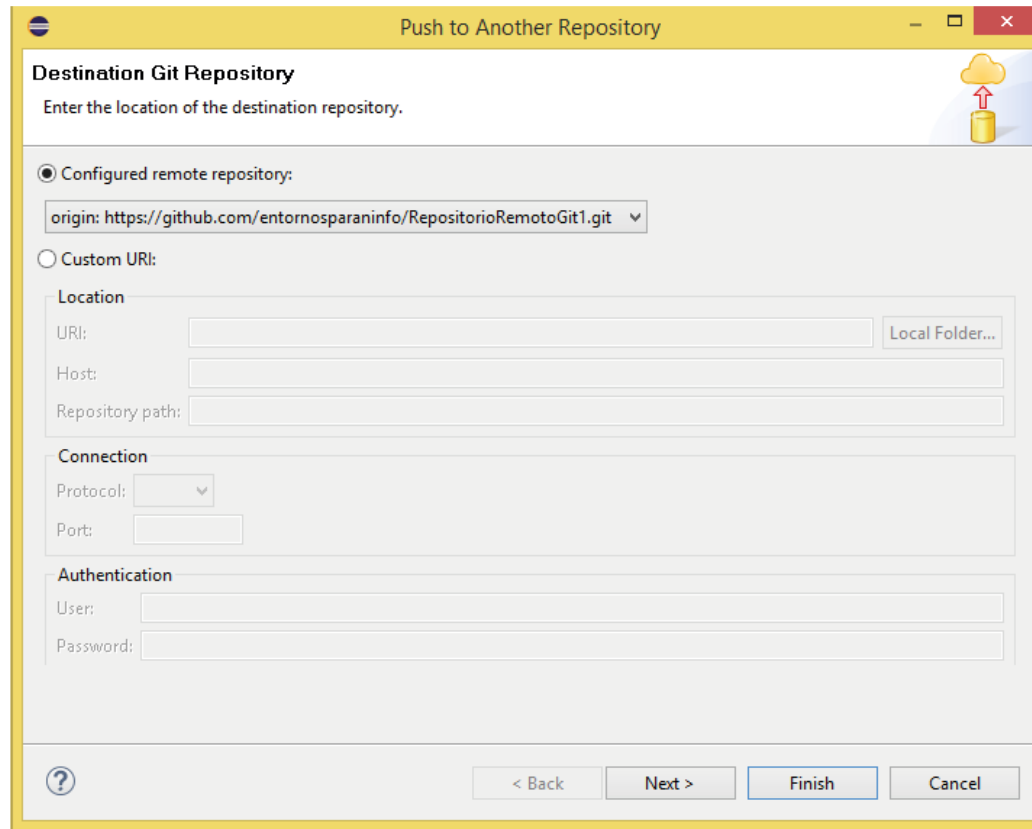
**Figura 4.30.** Desde la vista *Git Repositories* en Eclipse se crea un repositorio remoto para configurar posteriormente las subidas a dicho repositorio (*push*).



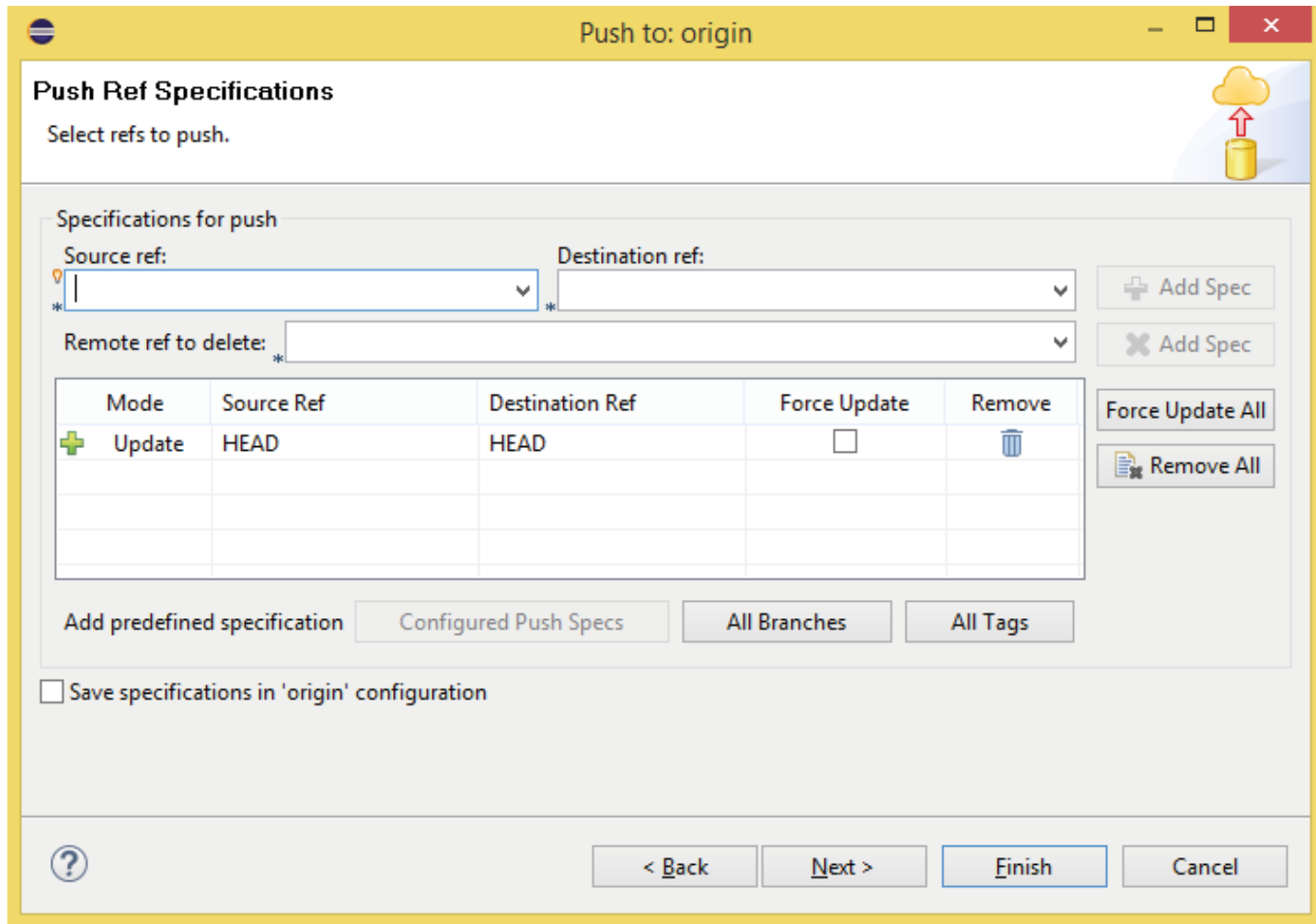
**Figura 4.31.** Ventana en la que se configura la subida de archivos al repositorio remoto de GitHub, proporcionando la URL de dicho repositorio, el nombre de usuario de GitHub y el token personal de acceso.

## Validación de cambios en el repositorio remoto

Para subir los cambios al repositorio remoto se selecciona para el menú contextual del proyecto la opción Team → Remote → Push.



**Figura 4.32.** Ventana en la que ya aparecen configuradas las subidas al repositorio remoto.

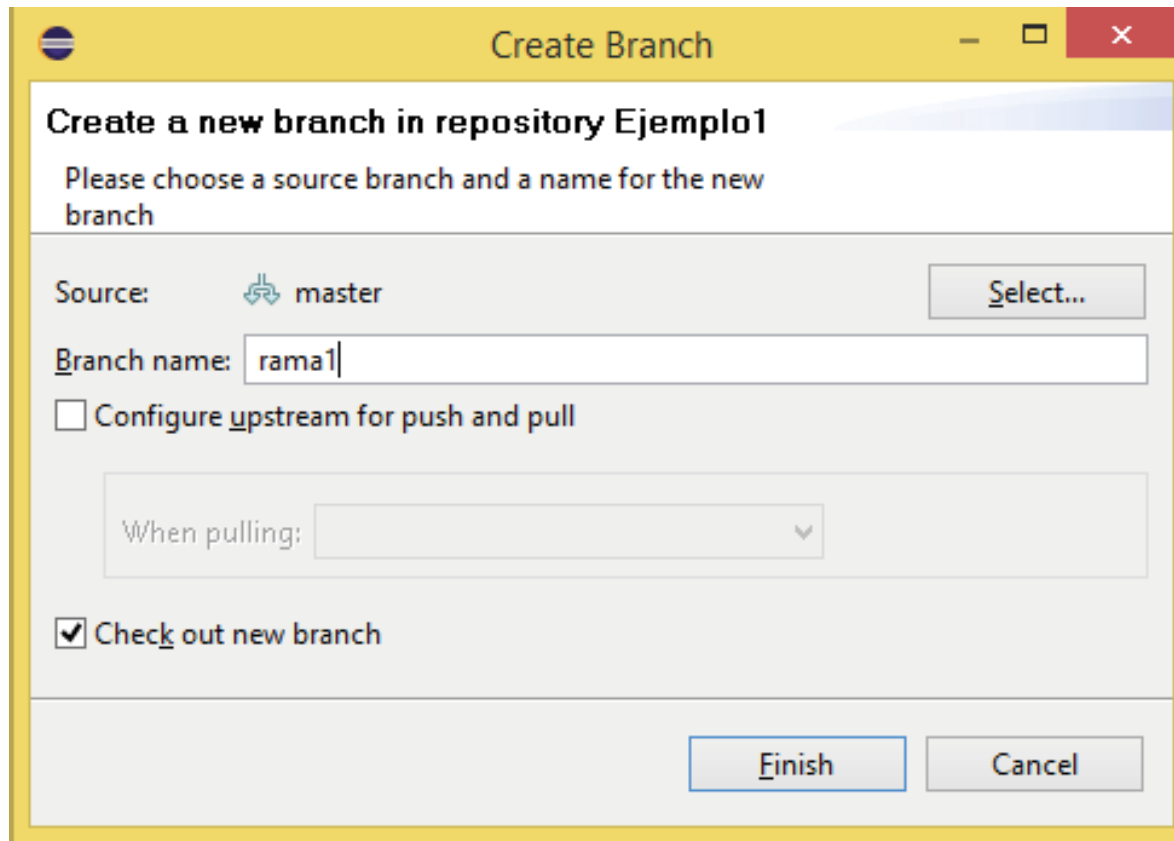


**Figura 4.33.** Ventana en la que se indica qué parte del proyecto se desea subir al repositorio remoto, seleccionándolo en *Source ref* y haciendo clic en el botón *Add Spec*.

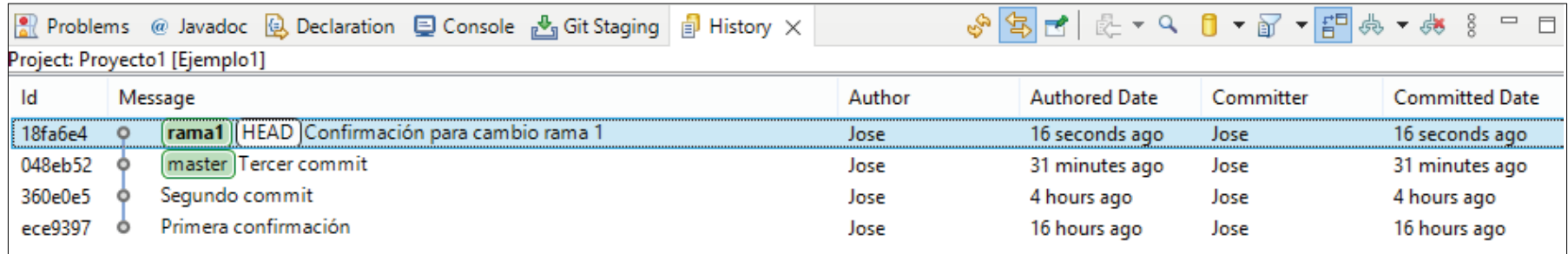


## Creación de una rama y validación de esta en los repositorios local y remoto

En el área de repositorios Git para el elemento *Branches* se elige la opción del menú contextual *Switch To → New Branch...*



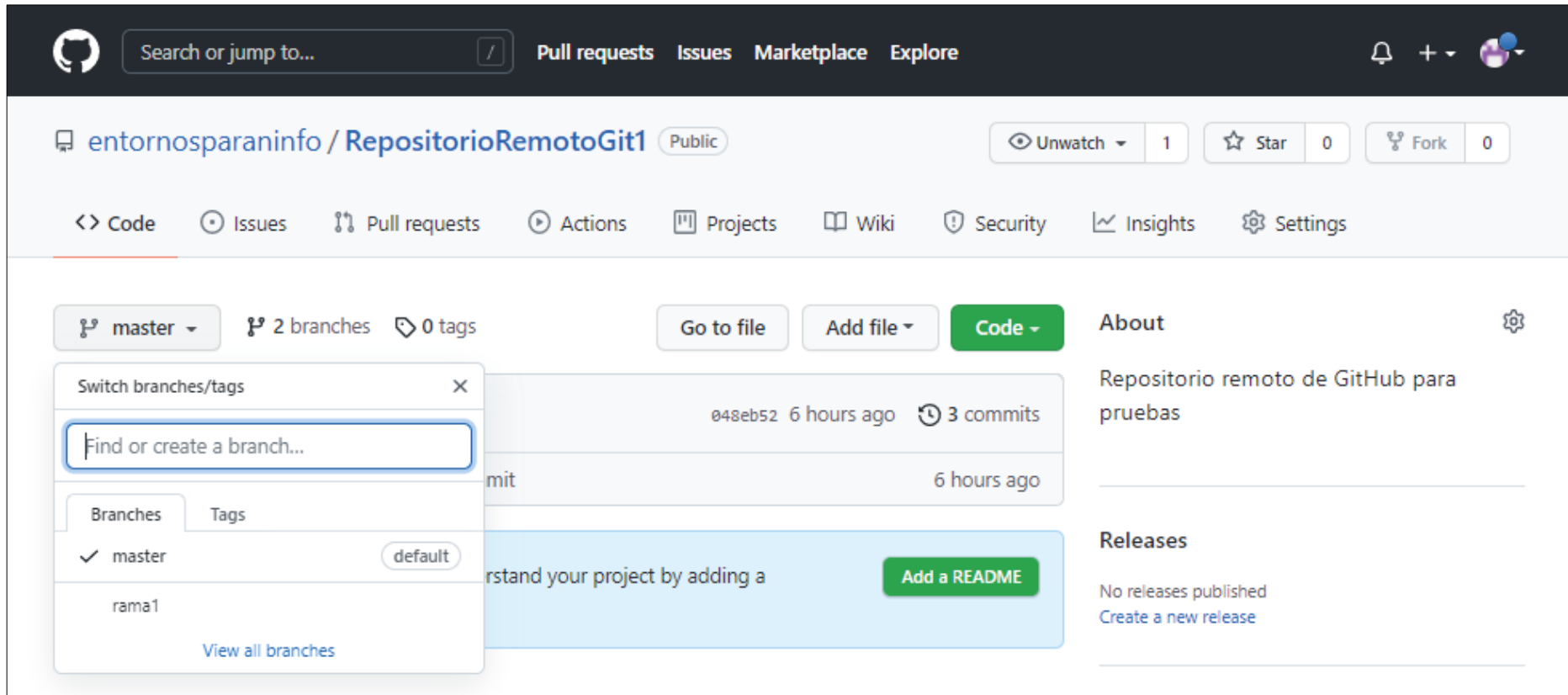
**Figura 4.34.** Ventana en la que se crea una nueva rama, llamada rama1, y nos movemos a ella.



Id	Message	Author	Authored Date	Committer	Committed Date
18fa6e4	rama1 HEAD Confirmación para cambio rama 1	Jose	16 seconds ago	Jose	16 seconds ago
048eb52	master Tercer commit	Jose	31 minutes ago	Jose	31 minutes ago
360e0e5	Segundo commit	Jose	4 hours ago	Jose	4 hours ago
ece9397	Primera confirmación	Jose	16 hours ago	Jose	16 hours ago

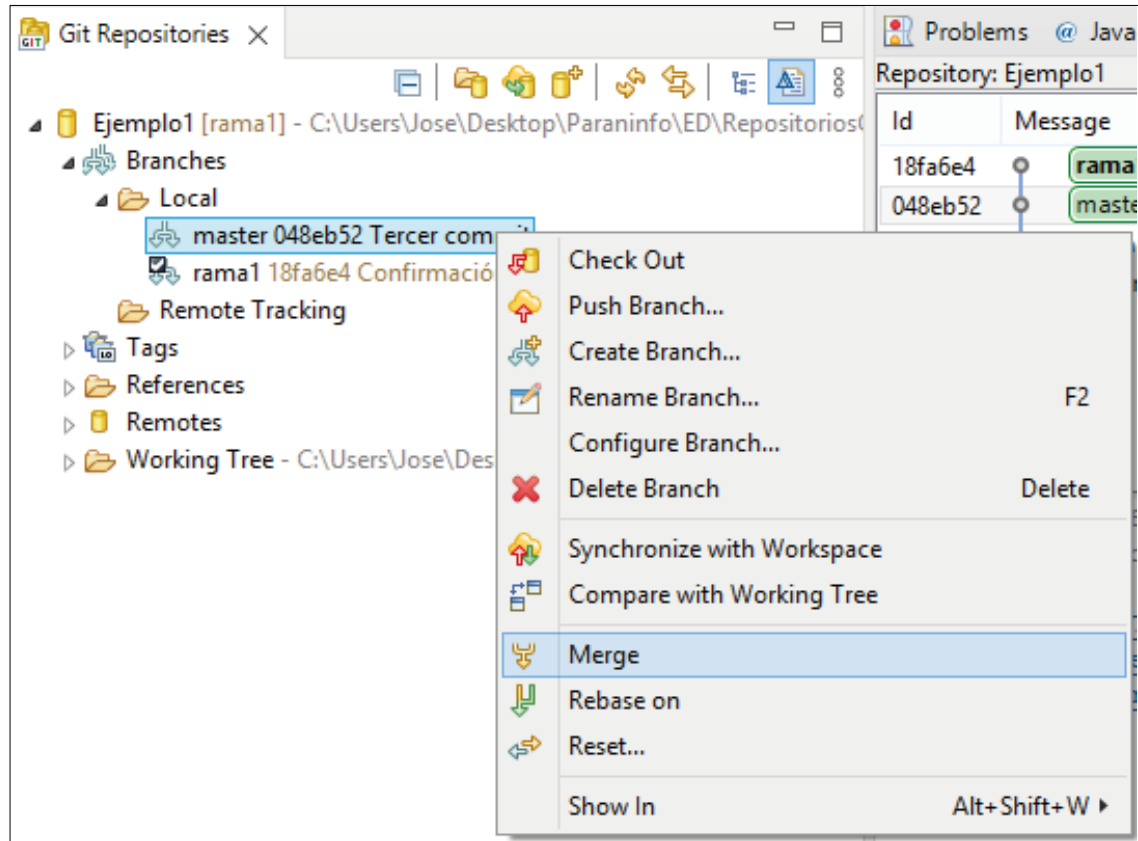
**Figura 4.35.** Vista de las diferentes versiones que se han creado para el proyecto, incluyendo la que se acaba de crear a partir de la rama *rama1*.

Para subir la rama al repositorio remoto, se selecciona para la rama la opción del menú contextual Push Branch...

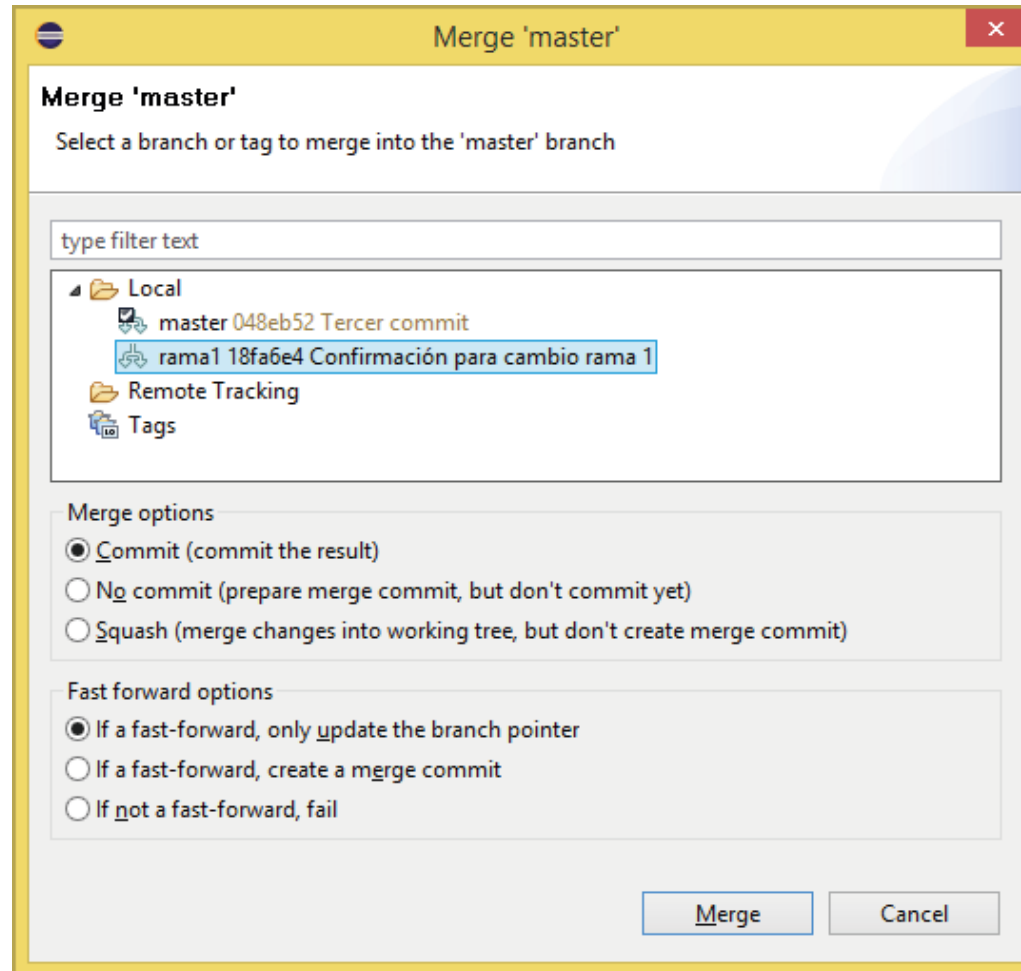


**Figura 4.36.** Pantalla de inicio de GitHub para el repositorio remoto *RepositorioRemotoGit1*, donde se puede observar que hay dos ramas (*branches*), pudiéndose seleccionar la rama *master* o la rama *rama1*.

## Fusión de ramas

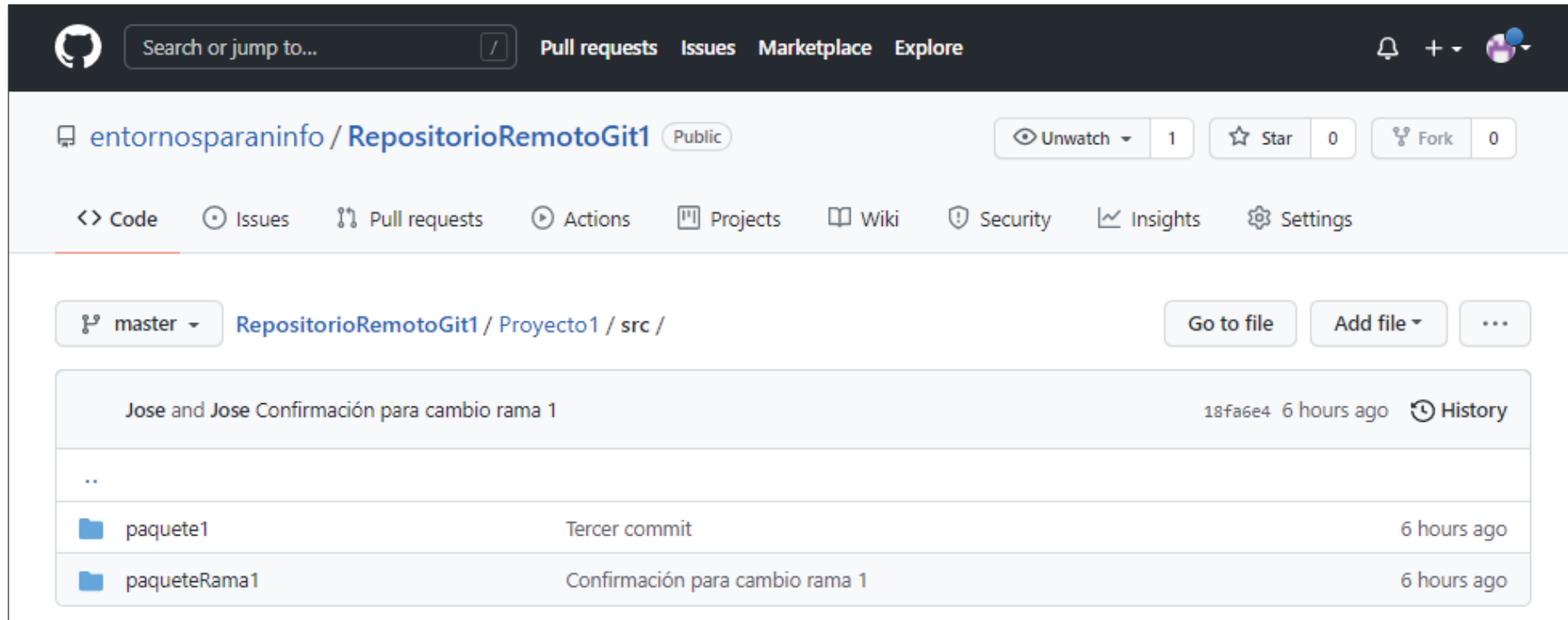


**Figura 4.37.** Para fusionar la rama *rama1* con el tronco (rama *master*), se selecciona en el área de repositorios Git de Eclipse, para la rama *master*, la opción del menú contextual *Merge*.



**Figura 4.38.** Ventana en la que selecciona la rama que se quiere fusionar con la rama master y se confirma haciendo clic en el botón *Merge*.

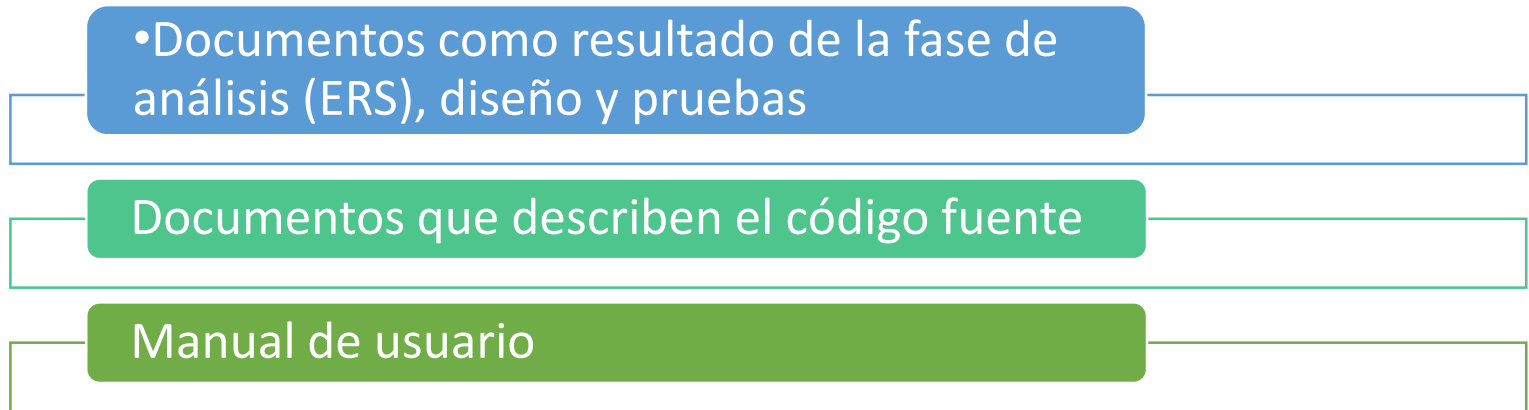
Para subir la fusión al repositorio remoto, se debe elegir la opción *Push Branch* del menú contextual para la rama master.



**Figura 4.39.** En GitHub, para la rama *master*, aparece tanto el contenido de la rama master antes de la fusión (paquete *paquete1*) como el contenido de la rama *rama1* (paquete *paqueteRama1*), puesto que ambas ramas se han fusionado correctamente.

## 4.4. Documentación

- ❑ Proporciona información relevante al cliente y favorece la reutilización.
- ❑ Tipos de documentos:



### 4.4.1. Estructura de los documentos

- ❑ El IEEE propone índices para la ERS, documentación de diseño y documentación de pruebas.
- ❑ No es obligatorio seguir estos índices, pero sí incluir toda la información a la que se refiere cada punto de los índices propuestos.

### 4.4.2. Generación de documentación

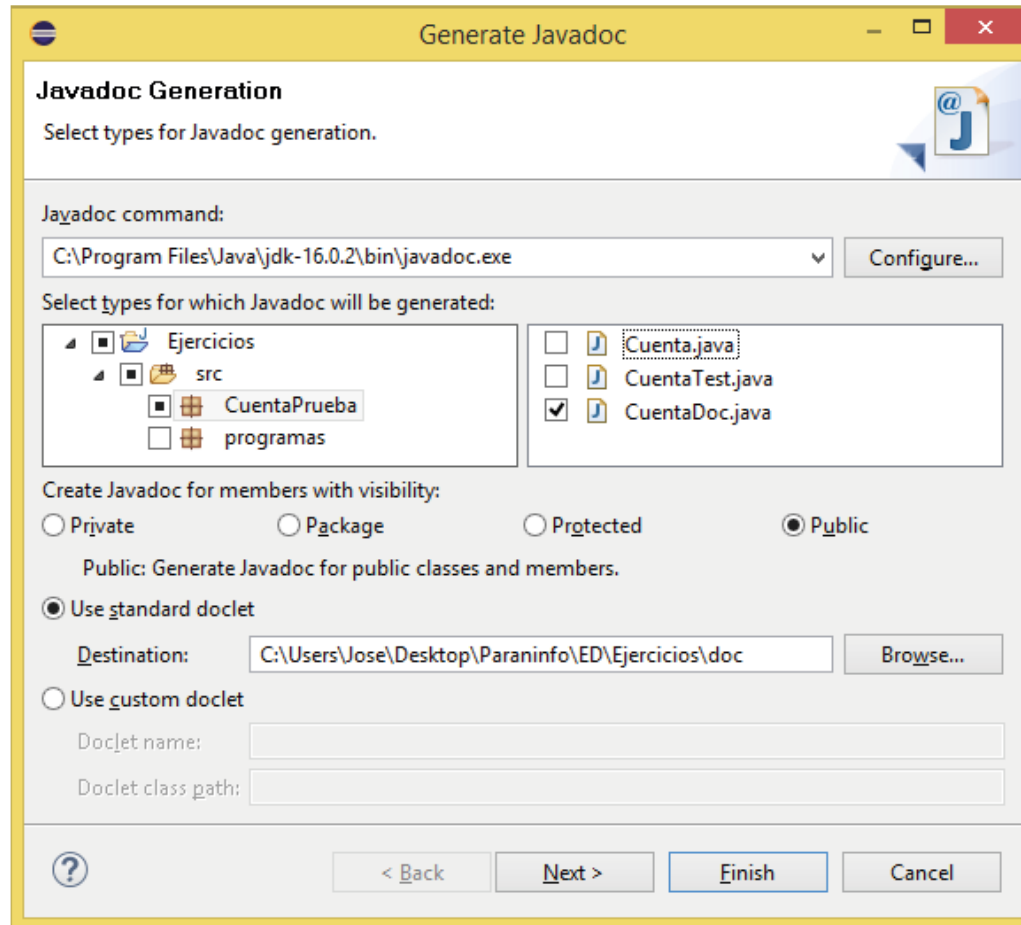
- ❑ Se usa para la documentación del código fuente, que es muy importante para facilitar el mantenimiento.
- ❑ Se deben documentar:
  - La utilidad de cada clase, método, variable.
  - Los algoritmos.
  - El autor de cada elemento.
  - La fecha de última modificación...
- ❑ JavaDoc permite generar documentación en formato HTML.
- ❑ Tipos de comentarios:
  - De línea: `//`
  - De varias líneas: `/* ... */`
  - De documentación JavaDoc: `/** ... */`. Cada línea debe comenzar con `*`



**Tabla 4.1.** Etiquetas que se incluyen en los comentarios de documentación JavaDoc

Etiqueta	Uso
@author nombre_autor	Autor del elemento que se documenta, generalmente una clase.
@param nombre_param descripción	Descripción del parámetro. Se usa una etiqueta de este tipo por cada parámetro.
@return descrip	Descripción de lo que devuelve el método.
@see referencia	Referencia a otro elemento: a una clase del mismo proyecto (paquete.clase), a otro método de la misma clase (#método()), a un método de otra clase (clase#método()), a un método de una clase de otro paquete (paquete.clase#método()).
@version versión	Versión de la clase. Solo es aplicable a clases.
@since descrip	Indica la versión del software a partir de la cual ha estado disponible la entidad declarada.

Tras incluir los comentarios de JavaDoc, para generar la documentación desde Eclipse, se elige la opción de menú *Project* → *Generate JavaDoc*.



**Figura 4.40.** Generación de documentación automática con Javadoc para la clase *CuentaDoc*.

Package CuentaPrueba

### Class CuentaDoc

java.lang.Object<sup>1</sup>  
CuentaPrueba.CuentaDoc

```
public class CuentaDoc  
extends Object1
```

**Clase CuentaDoc, para contener la información de cada una de las cuentas bancarias**

Version:  
01-2021

Author:  
Jose Piñeiro since 14-02-2021

#### Constructor Summary

Constructors

Constructor	Description
CuentaDoc(String <sup>1</sup> numCta, float saldoCta)	Constructor de la clase CuentaDoc con todos los parámetros

#### Method Summary

All Methods | Instance Methods | Concrete Methods

Modifier and Type	Method	Description
void	extraerDinero(float importe)	Método que saca dinero de la cuenta, decrementando su saldo
String <sup>1</sup>	getNúmero()	Método de selección del número de la cuenta
float	getSaldo()	Método de selección del saldo de la cuenta

**Figura 4.41.** Documentación generada con JavaDoc para la clase *CuentaDoc*.