

Optimization

Vestibulum molestie nisi nec nunc viverra efficitur. efficitur quam tristique aliquam. Proin ut est lectus. risus quis sagittis porta.

- Fusce luctus blandit nisi.
- Ut imperdiet dui at tincidunt mattis.

GET STARTED



Optimización y documentación

Objetivos

- Comprender el concepto de refactorización y los patrones de refactorización más usuales.
- Aplicar patrones de refactorización en el ámbito del entorno de desarrollo.
- Revisar el código fuente mediante un analizador de código.
- Conocer el concepto de control de versiones.
- Utilizar herramientas de control de versiones integradas en el entorno de desarrollo.
- Comprender la necesidad de una correcta documentación de una aplicación.
- Emplear herramientas del entorno de desarrollo para documentar clases.

Contenidos

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

Introducción

La calidad del código creado, como producto final de un proyecto de desarrollo, es fundamental para facilitar la tarea de mantenimiento, la cual supone un porcentaje muy elevado del esfuerzo realizado para el desarrollo de software. Por eso, actualmente, la refactorización del código fuente o mejora de su estructura interna, sin alterar su comportamiento externo, es una tarea muy relevante. En esta unidad, se explicará con detalle dicho proceso y se estudiarán los patrones de refactorización más usuales y su aplicación en Eclipse.

Por otro lado, los analizadores de código sirven para la mejora de la calidad del código fuente, pues detectan defectos en este, que podrían tener efectos adversos. Aquí, se describe el funcionamiento de un analizador de código integrado en el IDE Eclipse.

Un correcto control sobre las diferentes versiones que se van generando de una aplicación informática también es de gran importancia para controlar los cambios que se van produciendo y para facilitar el mantenimiento del software, motivo por el cual se dedica una parte de esta unidad al control de versiones.

La realización de una documentación clara y explicativa de una aplicación informática, incluyendo el código fuente, también es esencial para facilitar la tarea de mantenimiento. En este tema, se contempla la posibilidad de usar las herramientas de documentación que vienen incorporadas en los entornos de desarrollo.

4.1. Refactorización

Uno de los productos finales del proceso de desarrollo de software es el código fuente y, ante cualquier cambio que sea necesario llevar a cabo en una aplicación, habrá que modificar dicho código. Es muy frecuente que, a lo largo del ciclo de vida de una aplicación, haya que realizar cambios sobre esta por varios motivos y, para cada uno de ellos, existen distintos tipos de mantenimiento:

- **Mantenimiento perfectivo:** es el que se lleva a cabo porque el cliente propone nuevos requisitos funcionales o de cualquier otro tipo.
- **Mantenimiento correctivo:** es el que se realiza como consecuencia de la detección por parte del cliente de algún error tras la entrega de la aplicación.
- **Mantenimiento adaptativo:** es el que se lleva a cabo para poder utilizar la aplicación en nuevos entornos de hardware o software, como por ejemplo, el que se requiere cuando se necesita usar un programa en un nuevo sistema operativo.

La probabilidad de que haya que realizar alguno de estos tipos de mantenimiento sobre una aplicación es casi del 100 % y, además, es muy probable que la persona o personas que tengan que ocuparse de este no sean las mismas que se encargaron de crear la aplicación. Por todo ello, es del todo conveniente que el código fuente sea sencillo y esté bien estructurado.

Martin Fowler (2018) define así la **refactorización**: «realizar modificaciones en el código con el objetivo de mejorar su estructura interna, sin alterar su comportamiento externo». No se trata, por tanto, de encontrar y corregir errores en una aplicación, lo que constituiría una tarea de mantenimiento correctivo de esta, sino de realizar pequeños cambios en el código fuente sin alterar su comportamiento, de manera que el efecto acumulativo de todas estas modificaciones dé lugar a una simplificación del código y a una mejora de su estructura. El resultado de esta tarea es que el código será más fácil de entender y modificar, facilitando así la tarea de mantenimiento.

Argot técnico



El vocablo *refactorizar* no aparece en el diccionario de la RAE. Proviene del inglés *refactoring*, pero esta palabra inglesa tampoco aparece de momento en los diccionarios de inglés.

Para evitar usar una palabra extranjera, se podrían emplear vocablos como «perfeccionar», «reestructurar», «reorganizar», pero estos términos resultan más genéricos de lo deseable, por lo que a lo largo de esta unidad se usará el término *refactorizar*, cuyo significado se ha señalado en el párrafo anterior.

Existen varias razones por las que puede ser conveniente refactorizar:

- **Calidad:** un código de calidad es un código sencillo y bien estructurado, de manera que sea fácilmente legible y comprensible por cualquier persona sin que esta haya formado parte del equipo de desarrollo.
- **Eficiencia:** el esfuerzo que se invierta en refactorizar se verá recompensado cuando se tenga que realizar cualquier tarea de mantenimiento del software.
- **Evitar la reescritura de código:** refactorizar suele ser mejor que reescribir el código, ya que suele ser más costoso escribir código desde cero.

Mediante la refactorización se pretende conseguir un diseño lo más sencillo posible y de calidad, lo que implica:

- Que el código funcione, es decir, que haya superado la etapa de pruebas de acuerdo con el nivel de cobertura deseado.
- Que no existe código duplicado.
- Que el código permite entender el diseño.
- Que se ha minimizado el número de clases y de métodos.

4.1.1. Los malos olores (*bad smells*)

Es muy importante saber cuándo es conveniente refactorizar. Para abordar esta necesidad, Martin Fowler, quien popularizó la técnica de la refactorización, acuñó por primer vez el término *bad smells* (*malos olores*), que son aquellos síntomas en el código que aconsejan la realización de una refactorización. Que el código presente estos síntomas no significa que el software no funcione, pero puede llevar a una ejecución más lenta del programa o a un código difícil de mantener, y debido a la baja calidad del código, este será más propenso a tener fallos en el futuro.



Argot técnico

El término *bad smell*, que traducido literalmente del inglés quiere decir ‘mal olor’, se debe entender en sentido metafórico. Es frecuente escuchar la expresión española de que «algo no huele bien», la cual se emplea no en el sentido estricto del verbo *oler* (‘percibir un olor’), sino en sentido figurado. De hecho, en el diccionario de la RAE, la expresión *no oler bien* significa «dar sospecha de que encubre un daño o fraude». Esto es precisamente lo que ocurre cuando se detecta un *mal olor* en el código: se trata de un síntoma o una sospecha de que hay algo mal en el código que puede tener efectos secundarios negativos.

En la página web <https://www.codegrip.tech/productivity/everything-you-need-to-know-about-code-smells/> se realiza una clasificación de los *malos olores* que se pueden encontrar en el código, en función del nivel al que afectan:

■ A nivel de aplicación:

- *Código duplicado (duplicated code)*: consiste en la existencia de código similar en varios lugares. Es uno de los síntomas más frecuentes, ante el que es necesario extraer ese código y colocarlo en una única ubicación.
- *Cirugía a tiros (shotgun surgery)*: un cambio en una clase conlleva la realización de cambios en otras muchas clases.
- *Complejidad artificial (contrived complexity)*: consiste en usar patrones de diseño complejos cuando se podría usar un diseño más simple o de menor complejidad.

■ A nivel de clase:

- *Clase larga (large class)*: consiste en tener una clase con demasiados atributos, métodos o instancias. Esto es debido a que la clase tiene encomendadas más tareas de las que le deberían corresponder.
- *Clase demasiado simple (freeloader)*: consiste en tener una clase con muy pocas responsabilidades. Muchas veces se trata de clases que solo tienen atributos y métodos de acceso (*set*) y de consulta (*get*).
- *Envidia de funcionalidad (feature envy)*: existe un método en una clase que parece más interesado en otra clase que en la clase en la que se encuentra, es decir, un método que usa en exceso métodos de otra clase. Se suele resolver moviendo el método a la clase cuyos elementos más utiliza.
- *Código divergente (divergent code)*: al realizar un cambio en el sistema, una clase sufre demasiados tipos de cambios.
- *Grupo de datos (data clump)*: ocurre cuando conjuntos de datos se agrupan en varias partes del programa. Lo más probable es que sea más adecuado agrupar las variables en un único objeto.
- *Intimidad inapropiada (inappropriate intimacy)*: se da cuando una clase depende de los detalles de implementación de otra clase.
- *Legado rechazado (refused bequest)*: una clase no usa métodos y atributos de la superclase, lo que suele indicar que la jerarquía de clases se ha creado incorrectamente.

- *Complejidad ciclomática (cyclomatic complexity)*: se trata de una clase con demasiadas ramas y bucles. Esto quiere decir que el método o los métodos a los que afecta este problema se deben dividir en varios métodos o se deben simplificar.

■ A nivel de método:

- *Método largo (long method)*: es un método difícil de entender. Además, un método debe realizar una función única y bien definida, por lo que es muy probable que un método largo realice más tareas de las que debería. Por este motivo, se debería descomponer en varios métodos más pequeños.
- *Cadenas de mensajes (message chains)*: un método llama a otro método, el cual llama a otro método y este a otro, y así sucesivamente.
- *Demasiados parámetros (too many parameters)*: un método con una larga lista de parámetros hace que sea difícil de invocar y de probar. Además, puede indicar que el método está realizando más tareas de las que debería.
- *Línea de código excesivamente larga (God line)*: una línea de código demasiado larga hace el código difícil de leer, entender, corregir y dificulta la reutilización.
- *Excesiva devolución (excessive returner)*: consiste en un método que devuelve más datos de los que son necesarios.
- *Tamaño del identificador (identifier size)*: el nombre del identificador es demasiado largo o demasiado corto.

■ ■ ■ 4.1.2. Implantación de la refactorización

Existen dos posibilidades a la hora de realizar la refactorización: comenzar con un nuevo desarrollo o aplicar la refactorización *a posteriori* o, dicho de otro modo, ejecutarla sobre una aplicación ya desarrollada.

En el primero de los casos, la refactorización continua es lo más adecuado. Esta consiste en llevar a cabo pequeñas refactorizaciones y, a menudo, en vez de realizar pocas pero grandes refactorizaciones. Se debe recordar que una refactorización debe suponer un cambio en la estructura del código sin cambiar la funcionalidad de la aplicación y, por lo tanto, cuanto mayor sea el cambio de código producido por la refactorización, mayor es la posibilidad de que, como consecuencia de esos cambios, la aplicación deje de funcionar o lo haga de manera anómala.

En el caso de que la aplicación ya esté desarrollada, no hay más remedio que aplicar la refactorización sobre toda la aplicación en lugar de incorporar la refactorización como tarea que se va aplicando de forma continua a medida que se va escribiendo el código. Aun así, lo recomendable es ir llevando a cabo pequeñas refactorizaciones secuencialmente y, tras cada una, realizar pruebas para comprobar que la aplicación sigue funcionando correctamente, de manera que no se hayan introducido errores como consecuencia de las modificaciones incorporadas al código.

Para la implantación de la refactorización, se recomiendan las siguientes buenas prácticas:

1. Antes de comenzar la refactorización, es necesario llevar a cabo pruebas unitarias y funcionales.
2. Promover y recibir formación sobre patrones de refactorización, pues el conocimiento de las refactorizaciones más comunes permite a quienes realizan la programación detectar los *malos olores* de los que no serían conscientes si no tuvieran esa formación.
3. Usar herramientas especializadas, ya que las refactorizaciones muchas veces suponen pequeñas modificaciones muy simples y en muchas clases, que se pueden realizar de manera automática y sin riesgo mediante el empleo de dichas herramientas.
4. Comenzar refactorizando los principales fallos de diseño (código duplicado, clases largas, etc.). Se trata de refactorizaciones que son fáciles de aplicar y muy beneficiosas.
5. Refactorizar el código tras añadir cada nueva funcionalidad. Resulta muy productivo realizar discusiones en grupo sobre la conveniencia de realizar una refactorización tras la adición de cada nueva función al software.
6. Implantar la refactorización continua, por lo que las personas responsables del desarrollo del software deben incorporar la tarea de refactorización como una más dentro del proceso de desarrollo.

Sin embargo, aunque la refactorización continua puede ser una práctica muy adecuada, es cierto que hay ocasiones en las que no es tan fácil aplicarla por uno de los siguientes motivos:

- Porque un equipo de desarrollo debe comenzar a trabajar con código desarrollado por otro equipo y el código no tiene buena calidad o no está preparado para la adición de nuevas funcionalidades.
- Porque se ha aplicado refactorización continua, pero la calidad del código se ha degradado debido a que no se detectó a tiempo la necesidad de una refactorización o bien se aplicó una refactorización de manera incorrecta.

La mejor estrategia en estos casos es, como se indicó antes, dividir la refactorización completa en el mayor número posible de pequeñas refactorizaciones. También es aconsejable, antes de llevar a cabo cada pequeña refactorización, comunicar los cambios que se van a realizar a las personas afectadas, esto es, a las personas que están trabajando con el código que va a ser objeto de la refactorización. Asimismo, puede resultar conveniente explicar cada refactorización llevada a cabo una vez que esta se haya completado.

4.1.3. Patrones de refactorización en Eclipse

A la hora de refactorizar, se deben seguir distintos métodos o patrones. En el momento de utilizar cada patrón de refactorización, es posible previsualizar la solución que se ofrece, ante la cual, se puede optar por aplicarla o no.

Para aplicar los patrones de refactorización en el IDE Eclipse, hay que seleccionar el elemento sobre el que se desea aplicar la refactorización y elegir del menú contextual la opción *Refactor*, o bien seleccionar la opción *Refactor* del menú principal. La refactorización se puede aplicar sobre una clase, un atributo, una variable, una expresión, un bloque de instrucciones, etcétera.

En los siguientes subapartados, se describen los patrones de refactorización más habituales. A modo de ejemplo, se aplicarán algunos de los patrones de refactorización a una aplicación llamada *figuras*, que se encuentra disponible como material disponible previo registro en www.paraninfo.es. Esta aplicación dispone de una clase abstracta, *Figura*, con sus subclases *Rectángulo* y *Triángulo*. Además, *Rectángulo* dispone de la subclase *Cuadrado*. La clase *Figura* es abstracta porque no se puede crear ninguna figura sin saber si se trata de un rectángulo, un triángulo o un cuadrado. Cada figura se caracteriza por tener un centro y por su color. Para definir el centro de una figura, se utiliza un punto, motivo por el que se incluye también la clase *Punto*. El programa principal muestra un menú en el que se solicita elegir una figura (triángulo, rectángulo o cuadrado), se piden sus datos (por ejemplo, en el caso de un rectángulo, información sobre el centro y dos lados) y se muestran en pantalla su perímetro y su área.

Rename

Este patrón de Eclipse permite cambiar el nombre de una clase, de un atributo, de un método, de una variable, etc., de forma que se renombran todas las referencias a ese elemento a lo largo del programa.

A modo de ejemplo, aquí se va a cambiar el nombre de la clase *Rectángulo* en el proyecto *figuras* de manera que se elimine la tilde del nombre, esto es, su nuevo nombre será *Rectangulo*. Seleccionada la clase *Rectángulo*, se elige la opción del menú contextual *Refactor → Rename*. A continuación, se escribe su nuevo nombre en el campo *New name*. Si se pulsa en el botón *Next*, en la ventana emergente (Figura 4.1), aparecen en la parte superior las partes de la aplicación en las que se propone realizar la modificación, pudiéndose seleccionar las que nos interesan (lo más probable es que sean todas). Además, se puede visualizar el estado antes y después del renombrado. Si se hace clic en el botón *Finish*, se procede al renombrado en las partes seleccionadas.

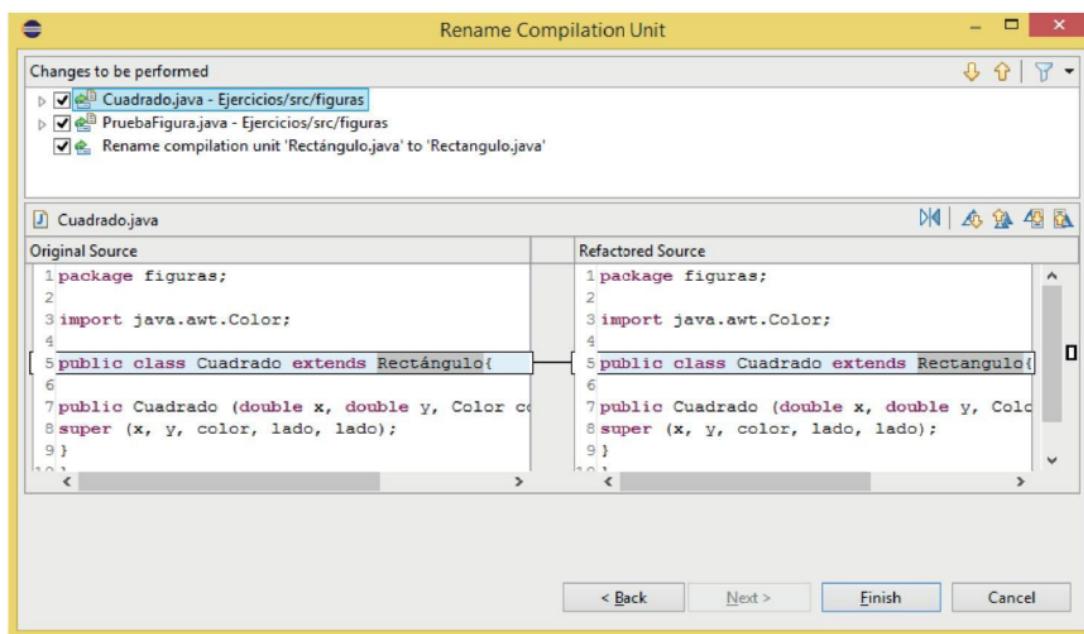


Figura 4.1. Al seleccionar el patrón de refactorización *Rename* de un elemento, se muestran todos los lugares de la aplicación donde ese cambio de nombre tendrá efecto antes de llevarlo a cabo.

Move

Esta opción permite mover una clase de un paquete a otro, procediéndose a cambiar todas las referencias a esa clase.

Extract Interface

Si se selecciona una clase en Eclipse, el programa permite extraer los métodos deseados de esa clase para crear una interfaz (véase en qué consiste una interfaz en el Apartado 5.4.5). En la interfaz se definen los métodos especificando solo su cabecera, es decir, el nombre del método, el tipo de valor de retorno (si devuelve algo) y sus parámetros. El código de los métodos se especifica en las clases que implementan la interfaz.

Por ejemplo, podría resultar interesante llevar a una interfaz el método *esMayorQue* de la clase *Figura* porque se van a incluir en la aplicación otras clases para las que también interesa saber si un objeto es mayor que otro. Así, se va a añadir la clase *Fracción*, para la que nos interesa saber si una fracción es mayor que otra. En este caso, es conveniente crear una interfaz porque aunque tanto las figuras como las fracciones se pueden comparar, no se comparan de la misma manera: en el caso de las figuras, una es mayor que otra en función de su área, mientras que en el caso de las fracciones, una es mayor que otra en función de su valor. Entonces, se va a crear una interfaz llamada *Comparar* que incluya el método *esMayorQue*.

Una vez seleccionada la clase *Figura*, se activa la opción del menú contextual *Refactor → → Extract Interface* y aparece una ventana como la que se muestra la Figura 4.2, en la que se pide el nombre de la interfaz y se deben seleccionar los métodos que se incorporarán a esta.

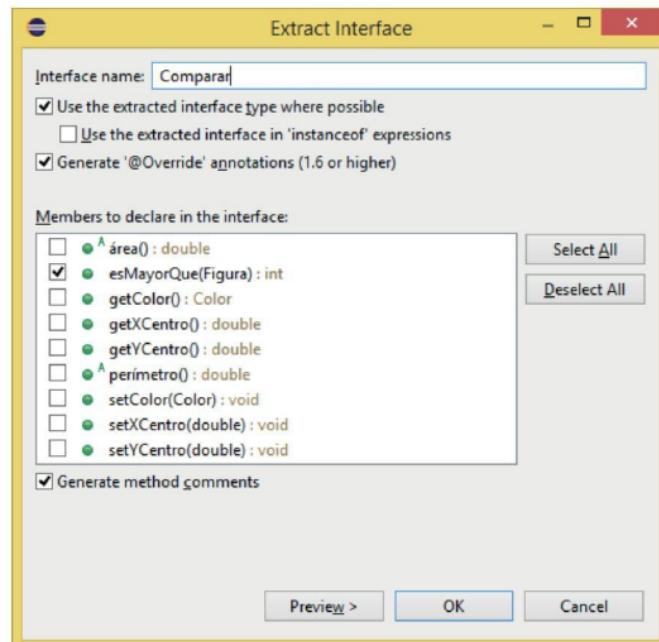


Figura 4.2. Al seleccionar el patrón de refactorización Extract Interface para una clase, se pide el nombre que se quiere dar a la interfaz y los métodos que se quieren colocar en ella.

Extract Superclass

Permite crear a partir de una clase una superclase para ella, la cual contendrá los métodos y atributos seleccionados de la clase en cuestión. Conviene tener en cuenta que puede haber errores en los métodos de la nueva superclase si estos hacen referencia a atributos de la clase original, esto es, a atributos de la subclase tras la refactorización.

Use Supertype Where Possible

Sustituye todas las referencias a una clase por la de su superclase después de identificar todos los lugares donde esta sustitución es posible. Resulta especialmente útil emplear este patrón después del patrón *Extract superclass*.

Pull Up

Mueve un atributo o un método de una clase a su superclase o, en el caso de los métodos, declara el método como abstracto en la superclase.

Pull Down

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

Extract Class

Sustituye un conjunto de atributos por una nueva clase. Todas las referencias a los atributos son reemplazadas para acceder a la nueva clase que contiene los atributos. Se proporciona la opción de crear métodos de acceso (*set*) y de consulta (*get*) para la nueva clase.

Change Method Signature

Permite cambiar la firma o cabecera de un método, esto es, su nombre, el nombre de sus parámetros, los tipos de sus parámetros o el tipo de dato que devuelve. Se actualizarán de manera automática todas las llamadas al método dentro del proyecto. Al seleccionar la opción del menú contextual *Refactor → Change Method Signature* para un método, como, por ejemplo, el método *distancia* de la clase *Punto*, aparecerá una ventana, como la de la Figura 4.3, en la que se puede cambiar el modificador de acceso del método, el tipo de dato que devuelve y su nombre. Asimismo, por cada uno de sus parámetros, haciendo clic en el botón *Edit...*, se puede modificar su tipo de dato, su nombre y se le puede modificar o asignar un valor por defecto. También es posible añadir nuevos parámetros pulsando el botón *Add*, o eliminar un parámetro, estando este seleccionado y clicando sobre el botón *Remove*. Los botones *Up* y *Down* sirven para cambiar el orden de los parámetros. También se pueden cambiar, añadir o eliminar las excepciones que lanza el método en la pestaña *Exceptions*.

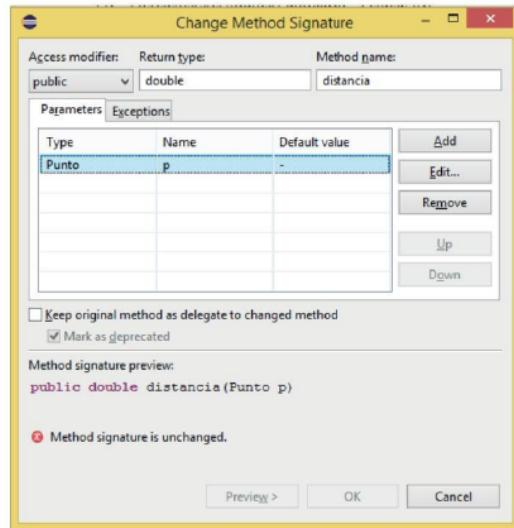


Figura 4.3. Al seleccionar el patrón de refactorización Change Method Signature para un método, se pueden cambiar todos los elementos definidos en su cabecera: modificador de acceso, nombre del método, tipo de dato que devuelve; nombre, tipo y valor por defecto de sus parámetros; se pueden añadir parámetros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

Por ejemplo, en el siguiente código del método *simétrico* de la clase *Punto*

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

se puede seleccionar la variable *nuevoP* y, en el menú contextual, elegir la opción *Refactor → → Inline*. El resultado será el que se muestra a continuación, donde se ha sustituido, en la instrucción *return*, la referencia a la variable *nuevoP* por el valor que se le asignó en la instrucción anterior:

```
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 y sustituye dichos parámetros por un objeto de la clase. Además, actualiza todas las llamadas al método para sustituir esos parámetros por un objeto de la nueva clase.

Extract Local Variable

Sustituye la expresión seleccionada por una nueva variable, de manera que cualquier referencia a esa expresión es sustituida por dicha variable en ese ámbito. Esta modificación solo afecta al método en el que se realiza la refactorización.

Por ejemplo, en el método *main* de la clase *PruebaFigura*, se selecciona la cadena “El perímetro es:” para extraerla a una variable local llamada *mostrarPerímetro*. Al seleccionar esta cadena, se activa la opción de menú Refactor → Extract Local Variable y se hace clic en el botón Preview, para poder ver los diferentes lugares del método donde se va a realizar la sustitución, la cual se llevará a cabo tras clicar sobre el botón OK (Figura 4.4).

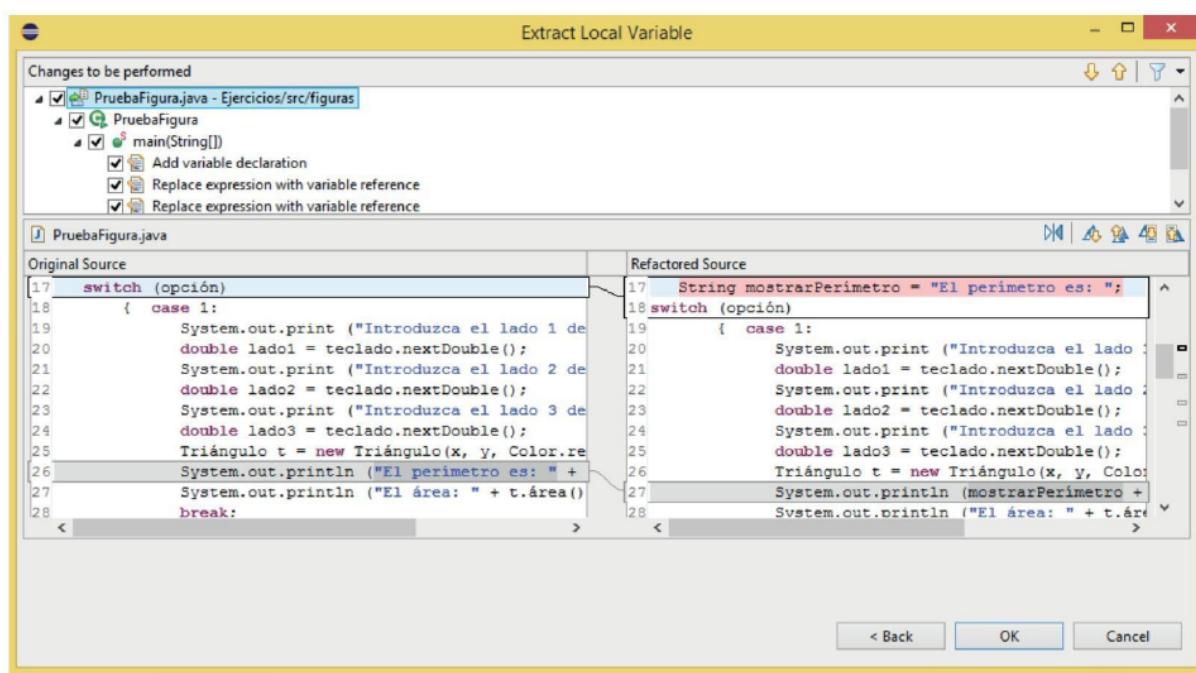


Figura 4.4. Al seleccionar el patrón de refactorización Extract Local Variable para una expresión, se pueden cambiar todas las ocurrencias de esa expresión por una variable cuyo nombre se debe indicar.

Extract Constant

Sustituye un número o una cadena de caracteres por una constante. Todas las apariciones del número o cadena son sustituidas por la constante indicada.

No obstante, se pueden visualizar los lugares donde se va a realizar el cambio y seleccionar aquellos donde se desea que se refactorice, aunque lo habitual es que se deseé que el cambio se lleve a cabo para todas las apariciones del número o cadena de caracteres.

El objetivo es que el valor literal se ubique en un solo lugar para facilitar su modificación, si fuera el caso.

Actividad resuelta 4.1

Uso del patrón de refactorización Extract Constant en Eclipse

Sustituye la cadena “El área es: ” del método *main* de la clase *PruebaFigura* por una constante llamada ÁREA.

Solución

Se selecciona la cadena en el método indicado, se activa la opción de menú contextual *Refactor → Extract Local Variable* y se hace clic en el botón *Preview*. Se podrán ver entonces los diferentes lugares del método donde se va a realizar la sustitución, tal y como se muestra en la Figura 4.5.

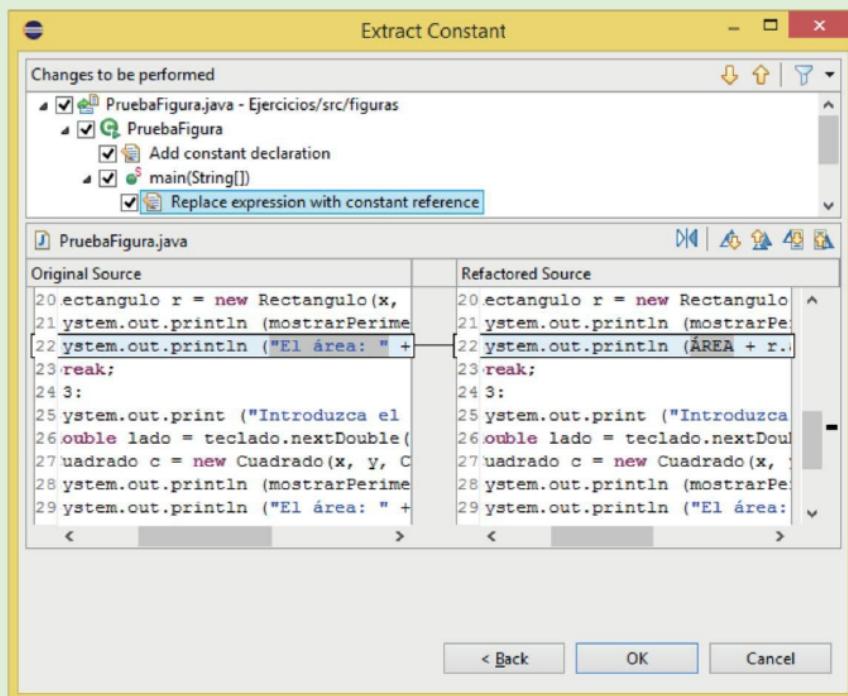


Figura 4.5. Al seleccionar el patrón de refactorización Extract Constant para una expresión, se pueden cambiar todas las ocurrencias de esa expresión por una constante cuyo nombre se debe indicar.

La sustitución se llevará a cabo tras clicar sobre el botón *OK*.

Convert Local Variable to Field

Convierte una variable local de un método en un atributo privado de la clase correspondiente. Tras la refactorización, todas las referencias a la variable local son sustituidas por el atributo. Si la variable es inicializada en su declaración, esta refactorización mueve dicha inicialización a la declaración del atributo, al método donde se usa o al constructor de la clase.

A modo de ejemplo, se puede convertir la variable local *mostrarPerímetro*, que se creó antes dentro del método *main* de la clase *PruebaFigura*, en un atributo de la clase. Para ello, se selecciona esta variable y se activa la opción de menú *Refactor → Convert Local Variable to Field*. Aparecerá una ventana (Figura 4.6), donde se debe asignar un nombre

al atributo, indicar su modificador de acceso y dónde se desea que se lleve a cabo su inicialización (en la declaración del atributo o en el método en el que se encuentra).

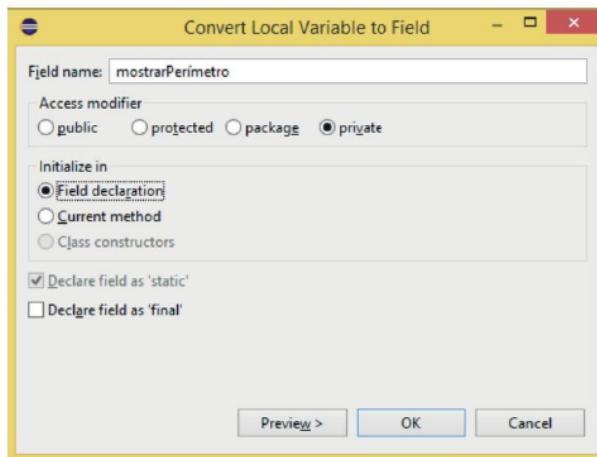


Figura 4.6. Al seleccionar el patrón de refactorización Convert Local Variable to Field para una variable local, se pueden cambiar todas las ocurrencias de esa variable por un atributo de la clase.

Al pinchar sobre el botón *Preview*, Eclipse también permite ver los efectos que tendrá el cambio realizado (véase Figura 4.7).

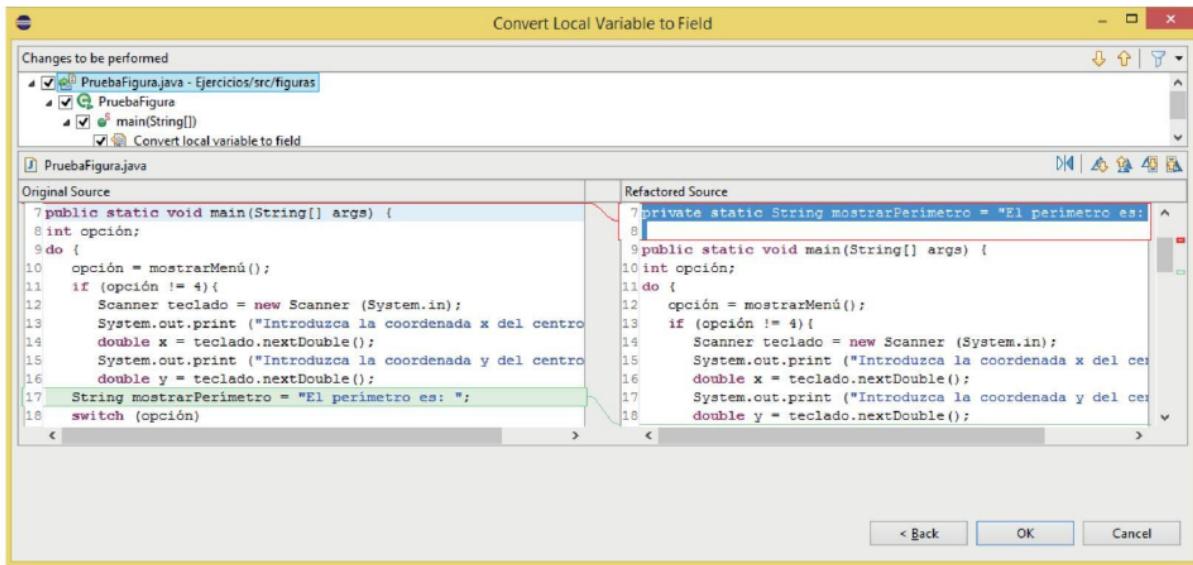


Figura 4.7. Ventana que aparece al hacer clic en el botón Preview y en la que se visualizan los cambios que supone la conversión de una variable local a un atributo de la clase.

Los cambios se llevarán a cabo al pulsar sobre el botón *OK*.

Extract Method

Convierte un conjunto de instrucciones en un método. Eclipse determinará de manera automática los parámetros y el valor de retorno del método. Es útil para acortar métodos excesivamente largos o para extraer porciones de código repetidas en varias partes de una aplicación.

Para ilustrar esto, en el método *main* de la clase *PruebaFigura* se podría asignar un método a cada bloque de código que se ejecuta al seleccionar cada opción de menú. En ese caso, se seleccionaría la porción de código entre *case 1* y la correspondiente instrucción *break* y se activaría la opción de menú *Refactor → Extract Method*. Aparece entonces una ventana en la que se debe asignar un nombre al método (*procesarTriángulo*) y un modificador de acceso (*private*). En dicha ventana se visualizan sus parámetros y, en la parte inferior, la cabecera del método resultante (Figura 4.8).

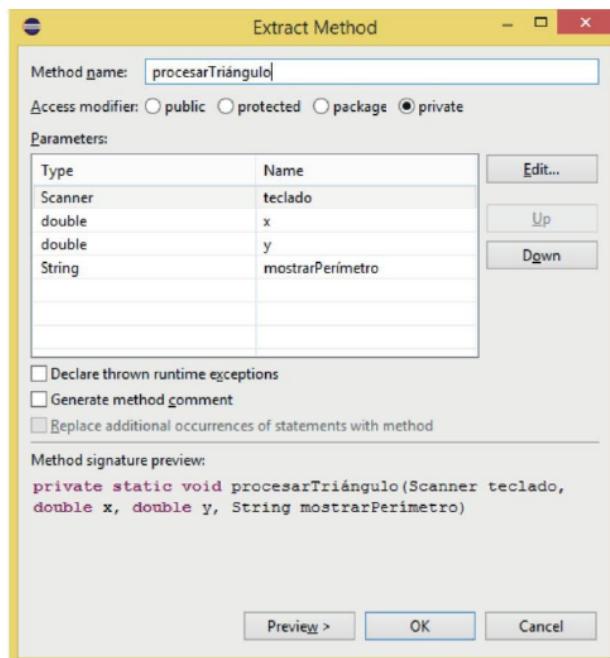


Figura 4.8. Al seleccionar el patrón de refactorización Extract Method para un bloque de código, se le da un nombre y un modificador de acceso al método y Eclipse ajusta automáticamente sus parámetros y el valor de retorno.

Además, se puede visualizar cómo quedará el código haciendo clic en el botón *Preview*. La refactorización se llevará a cabo pinchando sobre el botón *OK*.

Actividad propuesta 4.1

Uso del patrón de refactorización Extract Method en Eclipse

Sustituye cada una de las porciones de código en las que se procesa un rectángulo y un cuadrado del método *main* por un método, de manera similar a como se ha hecho para las instrucciones mediante las que se procesaba un triángulo.

Encapsulate Field

Este patrón sustituye todas las referencias a un atributo de una clase por los correspondientes métodos de consulta (*get*) y de acceso (*set*).

Por ejemplo, se podrían encapsular los atributos *x* e *y* de la clase *Punto*. Para ello, se selecciona el atributo *x* y se activa la opción *Refactor → Encapsulate Field* del menú

contextual. Eclipse indicará los nombres de los métodos de consulta y acceso que se usarán para el encapsulamiento, que serán los presentes en la clase, en caso de que ya estén declarados. Como siempre, al hacer clic en el botón *Preview*, es posible ver los cambios que supondría llevar a cabo esta refactorización, la cual se activará al pulsar el botón *OK* (Figura 4.9). Se debe hacer lo mismo para el atributo *y*.

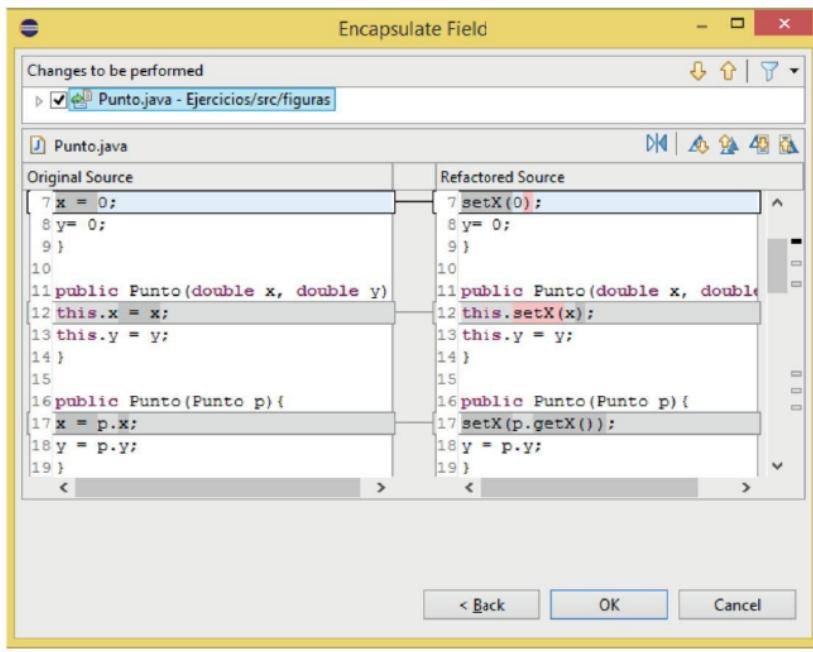


Figura 4.9. Al seleccionar el patrón de refactorización *Encapsulate Field* para un atributo, se sustituyen todos los usos de este atributo por una llamada al método get correspondiente y todas las asignaciones de valor a dicho atributo por una llamada al método set que corresponda.

Actividad propuesta 4.2

Uso del patrón de refactorización *Encapsulate Field* en Eclipse

Encapsula todos los atributos de la clase *Triángulo* del proyecto *figuras*.

4.1.4. Otras operaciones de refactorización en Eclipse

Eclipse permite almacenar en un *script* los cambios que han supuesto las refactorizaciones realizadas. Para ello, es necesario seleccionar la opción de menú *Refactor* → *Create Script*. En la pantalla que se muestra, se pueden seleccionar las refactorizaciones que se quieren almacenar en el *script*. También es posible cargar un *script* de refactorizaciones para aplicarlo por medio de la opción de menú *Refactor* → *Apply Script*.

Además, es posible consultar las refactorizaciones que se han llevado a cabo visualizando un histórico de refactorizaciones. A tal fin, se elige la opción de menú *Refactor* → → *History*. Entonces, se pueden seleccionar las refactorizaciones que se desea consultar y, al colocarse sobre cada una de ellas, se muestra información detallada sobre cada refactorización (véase la Figura 4.10).

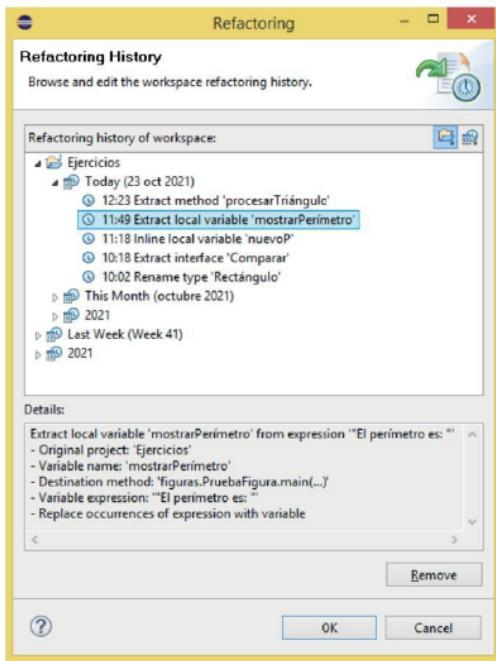


Figura 4.10. Al seleccionar la opción de menú Refactor → History, se muestra un histórico de refactorizaciones realizadas. Al colocarse sobre cada una de ellas, se obtiene información detallada.

Por último, se pueden eliminar las refactorizaciones que se deseen del historial, seleccionándolas y haciendo clic en el botón Remove.

4.2. Analizadores de código

Los analizadores de código son herramientas que realizan un análisis estático del código fuente. Estos analizadores evalúan el código fuente sin llegar a ejecutarlo. El objetivo es una mejora del código fuente sin modificar su comportamiento, que es exactamente el mismo fin que el de la refactorización. Sin embargo, en el caso de la refactorización, es el programador o la programadora quien debe detectar los síntomas de código mejorable (*bad smells*) y aplicar, entonces, el patrón de refactorización que permite eliminar esos síntomas. Los analizadores de código automáticamente detectan los síntomas y aportan también de forma automática una solución que la persona encargada de la programación puede decidir si aplicar o no.

Los analizadores de código realizan un análisis léxico y sintáctico del código fuente y si detectan que este es mejorable, lo indicarán y propondrán la manera de realizar la mejora.

La función principal de los analizadores de código es encontrar porciones de código que puedan generar efectos adversos como:

- Reducir el rendimiento.
- Provocar errores.
- Crear problemas de seguridad.

- Tener una excesiva complejidad.
- Complicar el flujo de datos.

Los analizadores de código son herramientas automáticas que realizan un análisis estático del código fuente con el fin de detectar deficiencias en este y proponer mejoras, para lo que se basan en una serie de reglas predefinidas. Si el análisis del código es realizado de manera manual por parte de una persona, recibe más bien el nombre de *comprensión de programas* o *revisión de código*.

Por un lado, existen analizadores de código gratuitos y de pago y, por otro lado, de código abierto y de código cerrado. En este sentido, PMD (*Programming Mistake Detector*) es un analizador de código gratuito y de código abierto. PMD es capaz de detectar errores comunes de programación como:

- Variables, métodos y parámetros no utilizados.
- Bloques vacíos de sentencias *catch*, *try*, *finally*, *switch*, etc.
- Expresiones lógicas que se pueden simplificar.
- Código que no se ejecuta nunca porque es inalcanzable.
- Código duplicado.
- Clases con complejidad ciclomática elevada.

Este analizador de código está orientado a los lenguajes de programación Java y Apex, pero también se puede emplear con JavaScript, Visualforce, PL/SQL, ApacheVelocity, XML y XLS.

Se puede instalar PMD como un módulo en Eclipse. Para ello, en Eclipse, se activa la opción de menú *Help → Eclipse Marketplace*. Tras escribir PMD en la casilla *Find* y hacer clic en el botón *Go*, uno de los módulos que aparece se llama *pmd-Eclipse-plugin 4.28.0* (Figura 4.11). Para proceder a su instalación, se pulsa en el botón *Install*.

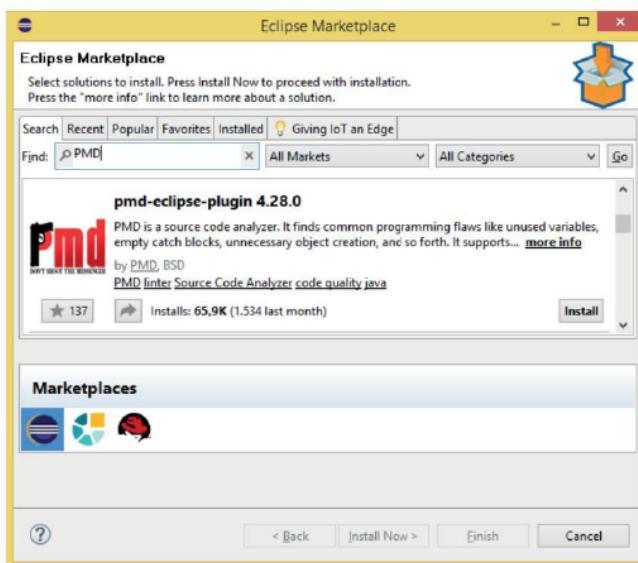


Figura 4.11. Ventana que muestra el módulo PMD en Eclipse. Para instalar este analizador de código, se clica sobre el botón *Install*.

Después de aceptar el acuerdo de licencia, comienza la instalación del analizador de código, el cual estará operativo tras reiniciar Eclipse.

Realizados los pasos anteriores, en Eclipse, tras seleccionar un proyecto, un paquete o una clase, se podrá elegir del menú contextual la opción de menú PMD → Check code. Aquí se hará para el proyecto *figuras*. Aparecerá una pantalla como la de la Figura 4.12.

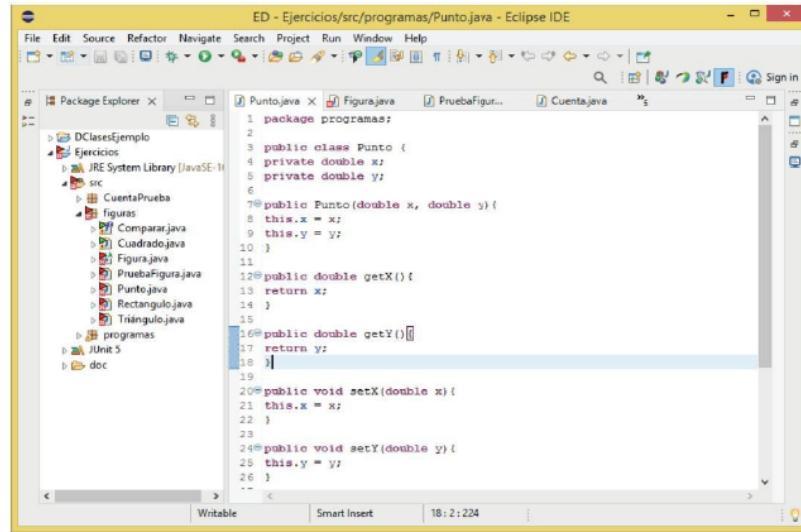


Figura 4.12. Ventana que muestra, para el proyecto *figuras*, el resultado del análisis de código realizado por PMD. Se indica por cada clase y mediante un color distinto si se ha detectado algún defecto.

En esta pantalla, se muestra la vista PMD, que aparece por defecto cuando se solicita el análisis de código. Por cada clase, se indica si se ha detectado algún defecto mediante símbolos de determinados colores. Se puede ver el significado de estos colores eligiendo la opción de menú Window → Preferences y luego seleccionando en la parte izquierda PMD (Figura 4.13).

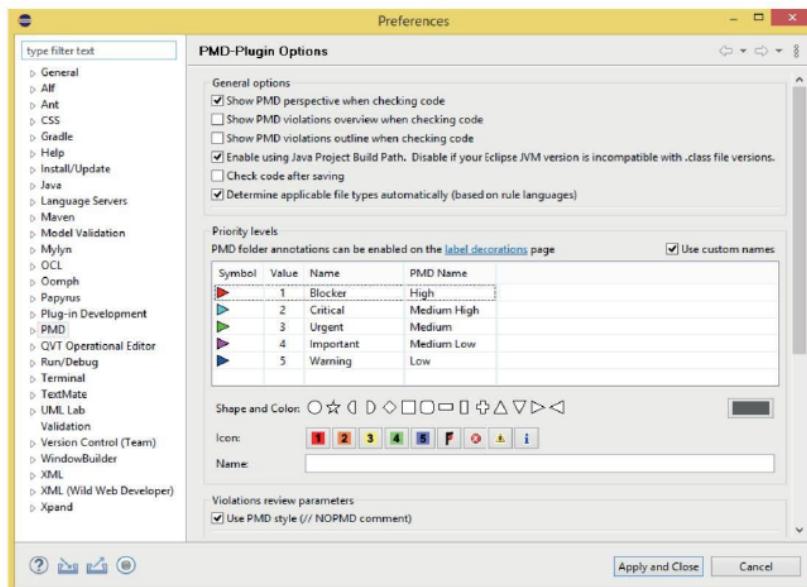


Figura 4.13. Ventana que muestra la configuración de PMD en Eclipse, que permite ver los distintos colores que se usan para identificar los defectos detectados por PMD.

Como se puede observar en la Figura 4.13, está activada la casilla de verificación que indica que se muestre la vista PMD cuando se solicita la realización del análisis de código. Las dos casillas de verificación que se muestran a continuación sirven para indicar si se desea que se muestren dos áreas de la pantalla al hacer la revisión de código: *Violations Overview* y *Violations Outline*, respectivamente. Más abajo se indican los símbolos que se emplean para cada tipo de defecto descubierto al lanzar PMD: la gama va desde el color rojo, que se usa para los defectos más graves hasta el color morado para los avisos (*warnings*). Se pueden modificar el símbolo y el color que se desea para la señalización de cada tipo de defecto. En este caso, se marcarán las casillas de verificación para solicitar que se muestren en la vista PMD las áreas *Violations Overview* y *Violations Outline*. Si se vuelve a analizar el código correspondiente al proyecto *figuras*, aparecerá la pantalla que se representa en la Figura 4.14.

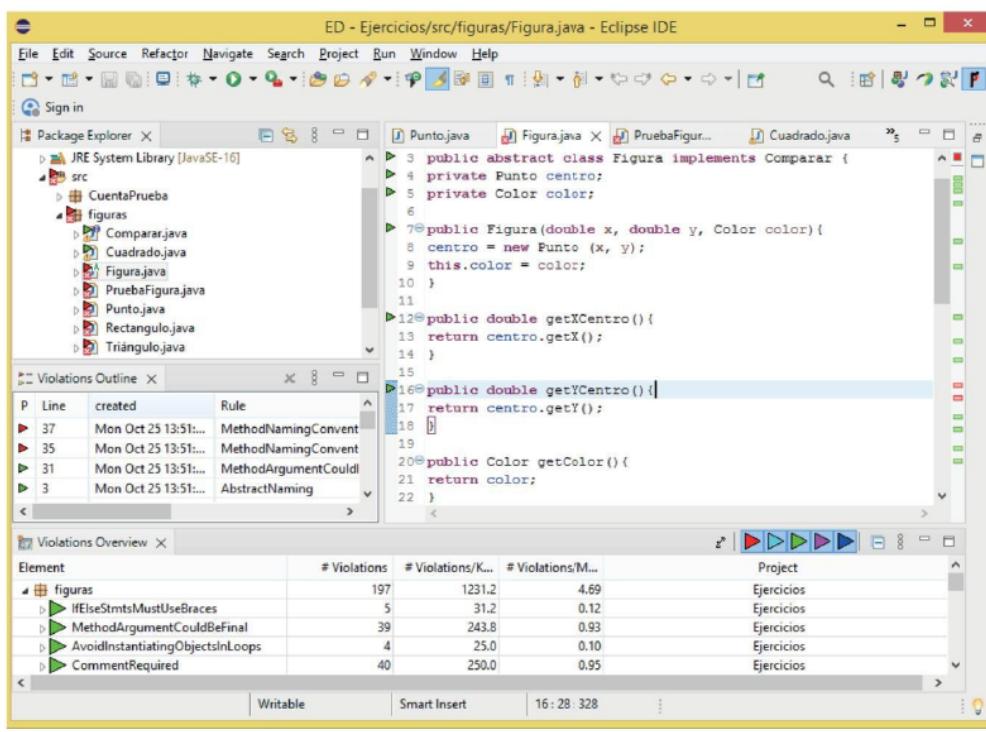


Figura 4.14. Resultado del análisis de código de un proyecto en la vista PMD, incluyendo dos nuevas áreas en la pantalla, que muestran información detallada sobre los defectos hallados.

En el área *Violations Outline*, se muestran en detalle los defectos encontrados en la clase cuyo código se está mostrando (en este caso, para la clase *Figura*). Los defectos aparecen ordenados por gravedad, de mayor a menor, y por cada uno de ellos, se indica la línea de código afectada, cuándo se creó, la regla que incumple y el mensaje de error generado. Por ejemplo, se señala como error grave que los nombres de los métodos *área()* y *perímetro()* no cumplen con el patrón *[a-z][a-zA-Z0-9]** por contener una letra con tilde. Por otra parte, se consideran errores de gravedad media que no haya comentarios para los atributos y métodos y que no se hayan usado llaves en las sentencias *if...else*.

En el área *Violations Overview*, se muestra un resumen de los defectos encontrados en todo el elemento analizado (proyecto, paquete o clase). Por defecto, se muestra por cada error detectado, su número de ocurrencias y se puede desplegar para ver en qué clases

se han cometido. Haciendo clic en el botón con los tres puntos de la parte derecha, se puede seleccionar *Show violations to files*, en cuyo caso los errores no aparecerán agrupados por tipo de error, sino por cada uno de los ficheros (clases) en que se han producido. Por cada fichero, se podrá desplegar la información para que se muestre cada uno de los tipos de errores.

En el área *Violations Overview*, es posible seleccionar cada error detectado y, en su menú contextual, elegir cada una de las siguientes opciones:

- *Show details*: se mostrará información detallada sobre el defecto encontrado.
- *Mark as reviewed*: si se marca esta opción, se da a entender que ya se ha revisado el error y que se ha tratado de la forma más conveniente, por lo que en posteriores análisis de código, no se volverá a mostrar.
- *Remove violation*: al seleccionar esta opción, se le indica al IDE que no se considera esta situación como un error.
- *Disable rule*: por medio de esta opción, se inhabilita la regla correspondiente a dicho error, por lo que todos los errores relacionados con esta regla no se volverán a mostrar hasta que no se vuelva a habilitar la regla.

PMD se basa en una serie de reglas predefinidas para la detección de defectos, de manera que, al lanzar este analizador de código, se considerará errónea toda aquella porción de código que incumpla alguna de las reglas establecidas. Se pueden consultar estas reglas eligiendo la opción de menú *Windows → Preferences* y seleccionando en la parte izquierda *PMD → Rule Configuration* (véase Figura 4.15).

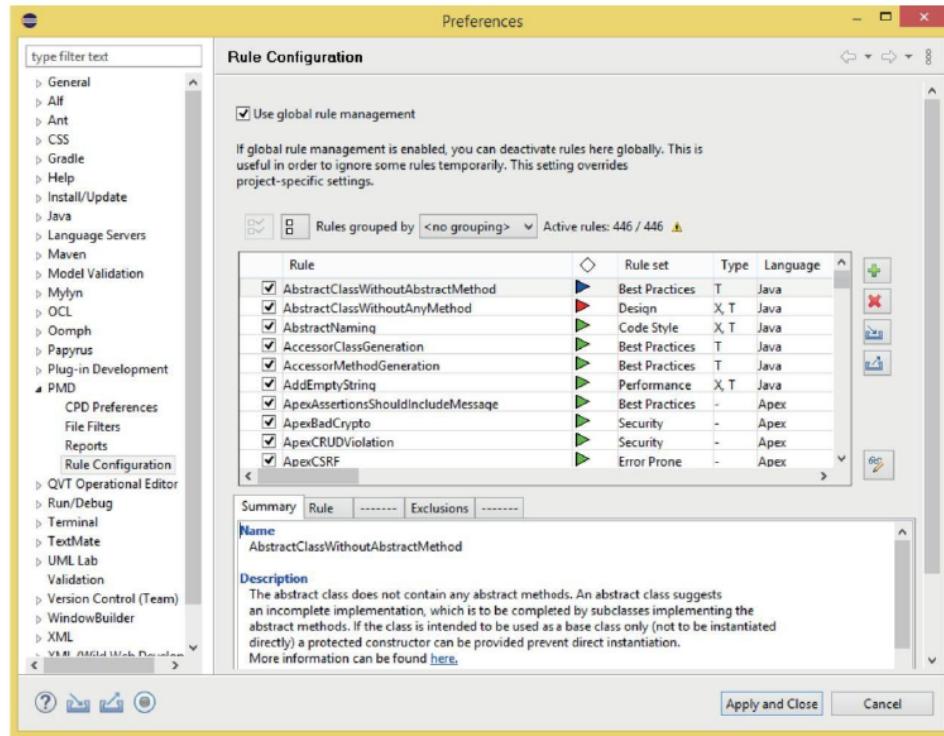


Figura 4.15. Ventana que muestra las reglas que emplea PMD al analizar el código. Si se activa la casilla de verificación de la parte superior, se pueden añadir reglas, borrar las existentes o modificar las propiedades de las reglas.

Si se marca la casilla de verificación de la parte superior *Use global rule management*, se pueden realizar modificaciones sobre las reglas. En la parte inferior, en la pestaña *Summary*, se puede leer el nombre, la descripción de la regla y un ejemplo. Como se puede observar en la Figura 4.16, en la pestaña *Rule*, se muestran las propiedades de la regla (su prioridad, el conjunto de reglas al que pertenece, el lenguaje para el que se aplica, etc.). Estas propiedades se pueden modificar, lo que tendrá efecto cuando se clique sobre el botón *Apply*.

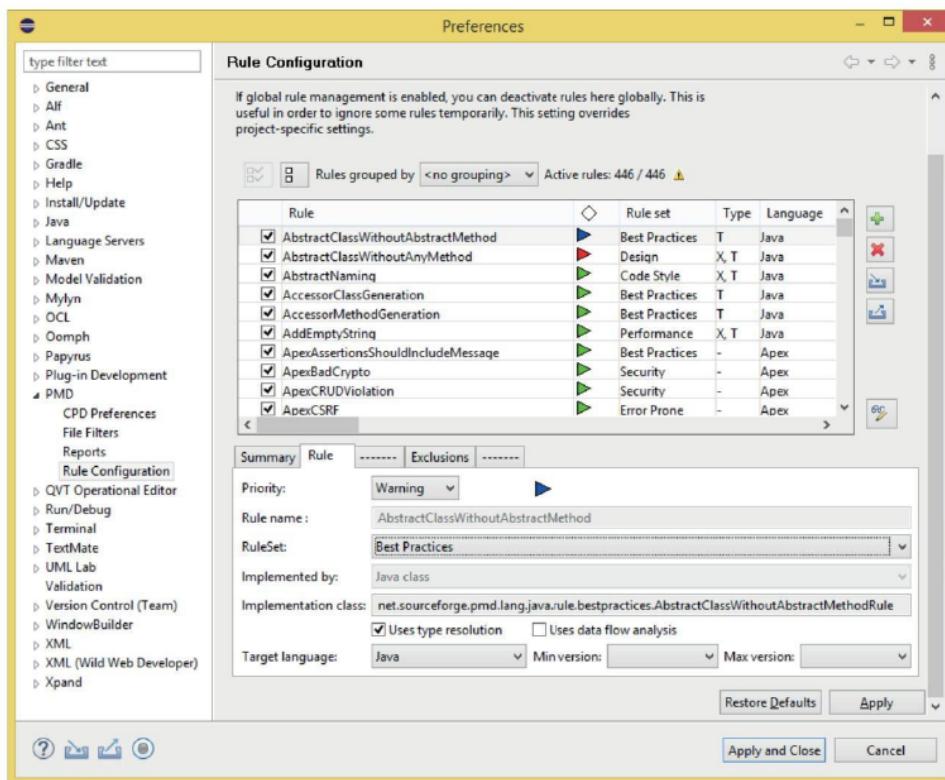


Figura 4.16. Ventana en la que se pueden visualizar y modificar las propiedades de una regla concreta en la parte inferior (pestaña Rule), haciendo clic en el botón Apply.

También es posible eliminar reglas o crear nuevas reglas haciendo clic en los botones y , respectivamente.

Si tras seleccionar en el explorador de proyectos un proyecto, un paquete o una clase, se elige en el menú contextual la opción de menú *PMD* → *Clear Violations Review*, se eliminan las revisiones de error realizadas, por lo que al volver a analizar el código, esos errores volverán a aparecer si siguen presentes en el código o si se han cometido de nuevo.

En caso de que se active la opción del menú contextual *PMD* → *Clear Violations*, se eliminarán todas las violaciones detectadas. No obstante, si se vuelve a analizar el código, estas volverán a aparecer.

Por otra parte, se puede utilizar PMD para encontrar código duplicado. Para ello, PMD incluye un detector de código duplicado conocido como CPD (*Cut and Paste Detector*). Para usarlo, se selecciona un proyecto en el explorador de proyectos y, en el menú contextual, se elige la opción *PMD* → *Find Suspect Cut and Paste*. Emergerá una ventana como la de la Figura 4.17.

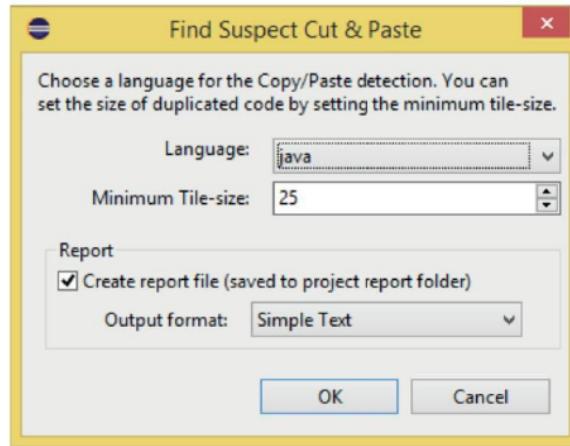


Figura 4.17. Ventana que se muestra al lanzar el detector de código duplicado CPD que lleva incorporado el analizador de código PMD.

Será necesario elegir el lenguaje de programación (en nuestro caso, Java) y, en el campo *Minimum Tile-size*, se debe indicar el número mínimo de *tokens* (identificadores, palabras reservadas, operadores, etc.), los cuales, si se repiten en varios lugares de la aplicación, se consideran código duplicado. El resultado de este detector o bien se puede mostrar en el área inferior de la pantalla, o bien, si se mantiene activada la casilla de verificación bajo la palabra *Report*, se generará un informe con el nombre *pmd-report* que se almacenará en la carpeta del proyecto y para el que se pueden elegir diversos formatos: texto plano (archivo con extensión .txt), XML o CSV.

Argot técnico



El vocablo *token* tiene diferentes acepciones. En el párrafo anterior, se ha usado este término con el siguiente significado: «cadena de caracteres que tiene un significado coherente en cierto lenguaje de programación».

En el Apartado 4.3.2, se usa otra acepción de *token*, que se traduce por ‘identificador’. En ese caso, se usa este identificador a modo de contraseña para la protección de cierta información sensible.

4.3. Control de versiones

El software está sometido a continuos cambios, no solo durante las tareas de desarrollo que se llevan a cabo antes de que el producto sea entregado al cliente (análisis, diseño, programación y pruebas), sino también después. Esto ha llevado a que se incluya la tarea de mantenimiento dentro de las fases del desarrollo de una aplicación informática.

Aparte de dicha tarea, ha surgido también la necesidad de otra nueva labor, más de gestión que técnica, que se debe llevar a cabo de manera paralela a las tareas técnicas de desarrollo (análisis, diseño, etc.) y que se conoce como **gestión de la configuración del software** (GCS). Esta tarea tiene fundamentalmente cuatro objetivos ante cualquier cambio:

1. Identificar el cambio.
2. Controlar el cambio.
3. Garantizar que el cambio se implemente de manera adecuada.
4. Informar de los cambios aplicados a todos aquellos a los que pueda afectar.

Recuerda



A la tarea de gestión de la configuración del software (GCS) ya se hizo referencia en el Apartado 1.6, en el que se señaló que, además de las actividades técnicas típicas del proceso de desarrollo de software (análisis, diseño, programación, etc.), hay otra serie de tareas de apoyo a todo el proceso, más de gestión que técnicas, una de las cuales Pressman (2010) llamó **administración de la configuración del software**, que tiene por objeto administrar los efectos del cambio a lo largo del proceso de desarrollo de software. Se trata de las mismas tareas pero con distinto nombre.

Como es sabido, el desarrollo de un producto de software, a no ser que este sea trivial, se debe descomponer en una serie de tareas como consecuencia de las cuales se obtienen una serie de productos intermedios hasta llegar al producto final (programa ejecutable) deseado por el cliente. Se puede afirmar, por tanto, que la configuración del software está formada por una serie de **elementos de configuración** (EC) aprobados en un momento dado del ciclo de vida. A medida que evoluciona el software, como consecuencia de la realización de uno o varios EC, o como consecuencia de una nueva versión de uno o varios EC, surge una nueva configuración del software. Normalmente, se trabaja con aquella configuración del software formada por las últimas versiones de los EC correspondientes.

Los EC pueden ser de dos tipos: **básicos** o **agregados**, donde un EC agregado se puede dividir en varios EC básicos. Así, por ejemplo, un EC agregado lo constituye el código fuente de la aplicación completa, y un EC básico, el código fuente de una clase.

La GCS debe gestionar los EC, identificándolos, controlándolos, informando acerca de su estado y validando su contenido. El objetivo de la GCS es facilitar el trabajo con los EC y los métodos para construir una configuración software que satisfaga las necesidades del cliente.

Conforme avanza el proyecto, se crearán nuevas versiones de los EC. El término **versión** tiene diferentes acepciones en el diccionario de la RAE, pero la que más directa aplicación tiene en nuestro ámbito es la siguiente: «cada una de las formas que adopta la relación de un suceso, el texto de una obra o la interpretación de un tema». Por lo tanto, este concepto hace referencia a las diferentes formas que puede adoptar un EC a lo largo su existencia. Y por consiguiente, una tarea correspondiente a la GCS es la identificación de las versiones de cada EC.

En la actualidad, la GCS se puede llevar a cabo mediante herramientas de control de versiones. Toda herramienta de este tipo hace uso de un **repositorio**, que se puede definir como un almacén que contiene toda la información sobre un proyecto, lo que incluye, por tanto, todas las versiones de todos los EC que forman parte de la configuración del

software. La existencia de este depósito integrado posibilita el almacenamiento de las relaciones existentes entre los diferentes EC, lo cual es fundamental para conocer los impactos de los cambios realizados sobre el software. Gracias al repositorio, es posible saber qué elementos del software se verán afectados, por ejemplo, por la realización de una modificación en una parte de un diagrama de clases.

La GCS, según está regulada en el estándar IEEE 828-2005, consta de las siguientes actividades:

- **Identificación de los EC:** se establece un método claro y consistente para identificar cada EC y cada una de sus versiones.
- **Control de cambios:** el objetivo de esta actividad es asegurarse de que los cambios aceptados sobre un proyecto se incorporan a este sin causar problemas. Para ello, ante una petición de cambio, se analiza y se evalúa su impacto, y tomando esto como base, se aprueba o no el cambio solicitado. Si se acepta el cambio, se genera una orden de cambio que describe este y cómo se debe llevar a cabo, indica las restricciones que se deben respetar y los criterios para su revisión. Una vez implementado el cambio, se actualiza la versión del EC o de los EC afectados por el cambio.
- **Auditoría de configuración:** tiene por objetivo comprobar si el cambio se implementó correctamente. Para esto, se dispone de dos medios: **revisiones técnicas formales** y auditorías de configuración. Las primeras se centran en determinar si el o los EC cambiados son correctos desde el punto de vista técnico; las auditorías de configuración pretenden averiguar otros aspectos: si se realizó el cambio especificado en la orden de cambio, si se llevó a cabo una revisión técnica, si se aplicaron adecuadamente los estándares de ingeniería del software, si se siguieron los procedimientos de la GCS para reflejar el cambio en el o los EC afectados, si los EC se actualizaron correctamente, si se informó del cambio al personal afectado, etcétera.
- **Generación de informes:** se debe documentar de manera adecuada el cambio que se ha realizado sobre la configuración del software para que los miembros del equipo de desarrollo puedan determinar qué EC deben utilizar, cuál está sujeto a peticiones de cambio y cuáles conforman la versión construida.

4.3.1. Gestión de versiones

Las versiones de un EC hacen referencia a las diferentes formas que va tomando este a lo largo del proceso de desarrollo de software. Se considera que existen versiones a nivel de cada EC y a nivel de la configuración del software completo.

Es posible representar las diferentes versiones de una configuración de software en forma de grafo, donde los nodos se corresponden con las diferentes versiones numeradas y los arcos muestran la transición de una versión a otra. A este respecto, hay dos tipos de grafos:

1. **Grafo de evolución simple:** que se representa mediante una simple secuencia lineal de versiones de forma que cada revisión de una versión da lugar a una nueva versión. No coexisten varias versiones en el tiempo.

2. **Árbol:** en el que existe un tronco, donde está la versión principal, desde el que se pueden crear ramas con el fin de realizar modificaciones sobre ellas para incorporar cambios al tronco, pero no modificándolo directamente, sino haciéndolo sobre la rama en cuestión. Se llama **cabeza** a la última versión del tronco. Dado que el producto que se entrega al cliente se encuentra en el tronco, siempre es necesario fusionar las ramas con el tronco, lo que consiste en unir con el tronco los cambios realizados en una rama. Puede ocurrir que incluso sea preciso fusionar varias ramas y luego fusionar el resultado con el tronco.

Cada vez que se crea una versión, a esta se le debe asignar una etiqueta. Los cambios que incorpora una versión con respecto a la anterior llevan el nombre de **delta** y esta información se puede registrar en el repositorio de diferentes maneras:

- **Deltas directos:** se almacena la primera versión completa y los cambios que presenta una versión con respecto a la anterior.
- **Deltas inversos:** se almacena la última versión completa y los cambios necesarios para reconstruir cada versión anterior a partir de la siguiente.
- **Marcado selectivo:** se almacena el texto refundido de todas las versiones como una secuencia en la que se indica por cada sección la versión a la que corresponde.

4.3.2. Herramientas de control de versiones

Hoy en día es común emplear herramientas o sistemas de control de versiones que son de gran ayuda en las tareas correspondientes a la GCS. Estos sistemas son especialmente útiles para posibilitar el trabajo colaborativo que caracteriza a todos los proyectos de desarrollo de software. El modo de trabajo con estas herramientas es como se indica a continuación:

1. El proyecto se encontrará cargado en un repositorio en un servidor.
2. Se realiza una copia del proyecto desde el repositorio al equipo local del programador.
3. Se realizan los cambios necesarios sobre el código.
4. Se envían los cambios al servidor.

Las herramientas o sistemas de control de versiones han ido evolucionando a lo largo del tiempo y, básicamente, se puede hablar de dos tipos de herramientas:

1. **Sistemas centralizados:** el repositorio del proyecto se encuentra en un servidor y quienes desarrollan el software disponen de herramientas para acceder desde su propio ordenador a este servidor, recuperar el proyecto, realizar cambios en su equipo local y enviarlos al servidor. El principal inconveniente de estos sistemas es que si se produce un fallo en el servidor, se pierde toda la información del proyecto si no se han realizado copias de seguridad y los usuarios no disponen de copias en sus equipos locales. Son ejemplos de sistemas centralizados CVS (*Concurrent Versions System*) y SVN (*Apache Subversion*).
2. **Sistemas distribuidos:** en este caso, el repositorio no se almacena en un único servidor, sino que está replicado para cada desarrollador, lo que tiene la ventaja

de que si se produce un fallo en el servidor, el contenido del repositorio se puede copiar a partir de la copia existente en los equipos de cada uno de los clientes. Son ejemplos de sistemas distribuidos Git y Mercurial. Dado que Git es la herramienta de control de versiones más empleada en la actualidad, se analizará en detalle a continuación.

Git

Git es una herramienta libre de control de versiones de software que fue creada por Linus Torvalds durante el proceso de desarrollo del núcleo de Linux.

Trabajando con Git, los ficheros deben pasar por varias ubicaciones antes de quedar almacenados en el repositorio. Así, es posible distinguir las siguientes áreas:

- **Área de trabajo (*working area*):** se trata del directorio local de trabajo en el que se crean los archivos primero y luego se modifican.
- **Área de preparación (*staging area*):** es una zona intermedia en la que se colocan los archivos del área de trabajo que se encuentran en esta área pendientes de confirmación.
- **Repositorio local (*commit area*):** este repositorio debe ser creado en el equipo local y es donde se almacenan los archivos con sus modificaciones y las diferentes versiones antes de enviarlos al repositorio remoto. Normalmente, los archivos pasan del área de preparación al repositorio local.
- **Repositorio remoto:** este repositorio se encuentra fuera del equipo local y es donde se almacenan los ficheros totalmente confirmados. Es posible establecer una sincronización entre el repositorio local y el remoto.

Por otro lado, en Git un archivo puede pasar por los siguientes estados:

- **Fuera de seguimiento (*untracked*):** se encuentran en este estado los archivos nuevos creados en el repositorio local o añadidos a él. El resto de los estados se pueden considerar estados bajo seguimiento.
- **Modificado (*modified*):** cuando los archivos han sido modificados en el repositorio local, pero los cambios no se han confirmado en este repositorio, los archivos se encuentran en estado modificado y en el área de trabajo.
- **Preparado para confirmación (*staged*):** cuando los archivos modificados en el ordenador local se han preparado para formar parte de la siguiente confirmación, se dice que están en estado preparado y se encontrarán en el área de preparación.
- **Confirmado (*committed*):** después de hacer una confirmación (*commit*), los ficheros preparados para su confirmación se almacenan en el repositorio local (*commit area*).

En Git se emplean una serie de comandos que es conveniente aprender. El objetivo de estos comandos es el cambio de estado de los diferentes archivos de un proyecto hasta que se crea la versión definitiva que se entrega al cliente. Hay que tener en cuenta que los archivos se crean en modo local y, una vez confirmados los cambios realizados sobre ellos, deben subirse al repositorio remoto. Los comandos básicos son los siguientes:

- **`git add`**: permite pasar un fichero o un directorio del área de trabajo al área de preparación.
- **`git commit`**: confirma cambios realizados en el área de preparación, de manera que los ficheros correspondientes pasan al repositorio local (*commit area*). Al confirmarlos (*commit*), Git exige la escritura de un mensaje de confirmación.
- **`git push`**: permite pasar los cambios realizados en el repositorio local al repositorio remoto.
- **`git pull`**: permite actualizar el repositorio local con los cambios más recientes del repositorio remoto, de forma que los archivos que reflejan esos cambios llegan al área de preparación.
- **`git clone`**: hace una copia del repositorio remoto en el repositorio local.
- **`git branch`**: permite crear una nueva rama.
- **`git checkout`**: permite cambiar a la rama indicada.
- **`git merge`**: permite fusionar las dos ramas indicadas.

En la Figura 4.18, se muestra cuáles son los comandos que permiten el paso de los ficheros de un área a otra.

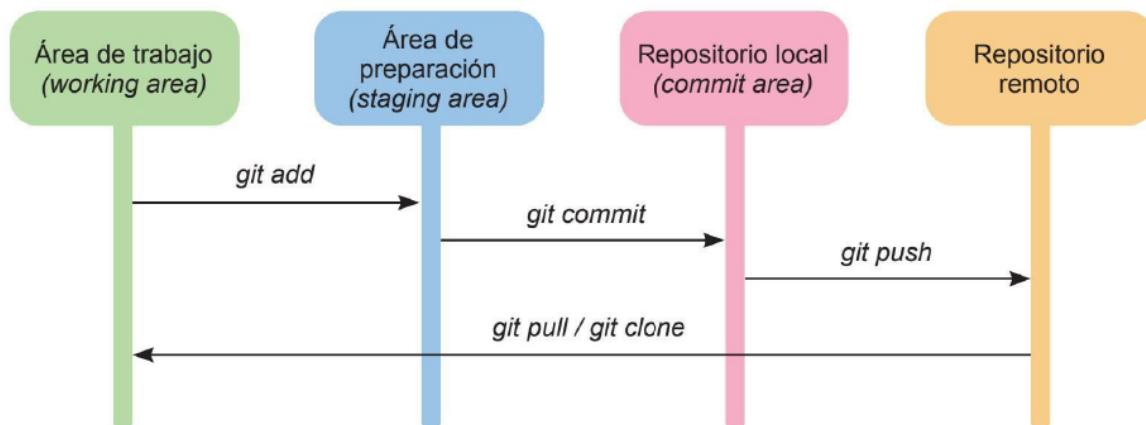


Figura 4.18. Diferentes ubicaciones por las que puede pasar un archivo gestionado con Git hasta que llega al repositorio remoto y los comandos que permiten el paso de una ubicación a otra.

Aquí se estudiará cómo se trabaja con Git integrado en el IDE Eclipse. Para ello, primero, se crea una carpeta, que se puede llamar *EjemplosControlVersionesGit* y será el nuevo espacio de trabajo (workspace) para Eclipse, y otra carpeta que se puede llamar *Repositorios-Git* para almacenar los repositorios locales de los proyectos.

Creación de un repositorio local

En primer lugar, se abre en Eclipse la vista Git seleccionando la opción de menú **Window → → Perspective → Open Perspective → Other → Git**. Aparecerán entonces en la parte izquierda de la pantalla tres opciones: 1) añadir un repositorio de Git local ya existente, 2) clonar un repositorio de Git o 3) crear un nuevo repositorio local de Git. Se elige esta

última opción, tras lo cual, se pide indicar la ruta del repositorio (carpeta *RepositoriosGit*) y el nombre del nuevo repositorio: *Ejemplo1*. En la ventana representada en la Figura 4.19, se puede observar que el nombre del repositorio se debe indicar después de la ruta y que la rama que se crea por defecto se llama *master*. Se trata del nombre que Git asigna a la rama principal, también llamada *tronco*.

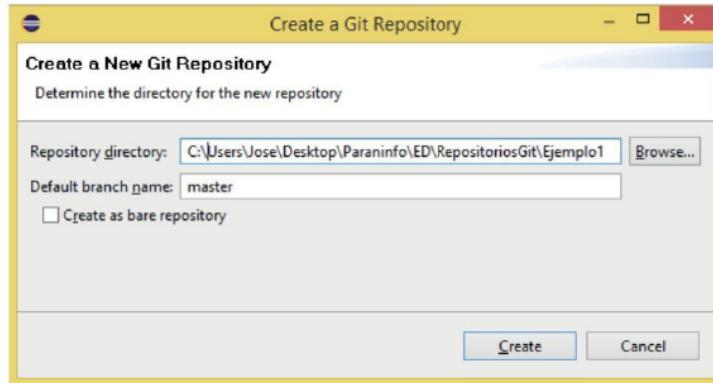


Figura 4.19. Ventana en la que se selecciona la ruta del repositorio y se le asigna un nombre. Asimismo, se crea la rama principal o tronco del repositorio, a la que Git llama por defecto con el nombre de master.

Al hacer clic en el botón *Create*, se crea el repositorio, cuya estructura se puede observar en el área de repositorios Git, que aparece en la parte izquierda de la pantalla debajo del área de proyectos (Figura 4.20). El *Working Tree* que se puede observar en esta área se corresponde con el área de trabajo y se ubica en la carpeta asignada al repositorio local.

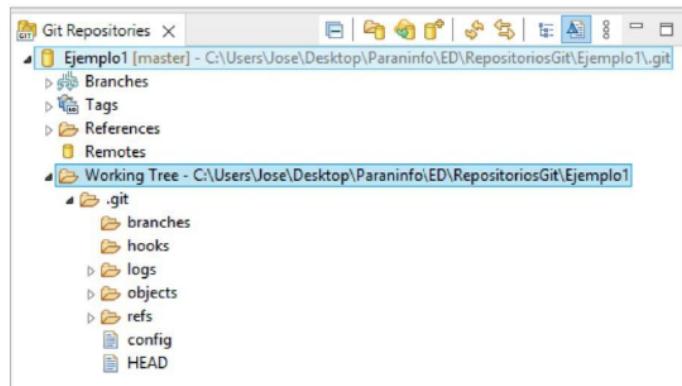


Figura 4.20. Estructura del repositorio que se acaba de crear y que se muestra en la parte izquierda de la pantalla.

Creación de un proyecto Java

A continuación, se crea un nuevo proyecto en Eclipse, para lo que se abre la vista Java, a la que se accede activando la opción de menú *Window → Perspective → Open Perspective → Other → Java (default)*. Se crea el proyecto en la carpeta *EjemplosCtrlVersionesGit* y se le asigna el nombre *Proyecto1*. Después, se asocia este proyecto al repositorio local *Ejemplo1* que antes se ha creado. Para ello, en el menú contextual del proyecto, se pulsa sobre la opción *Team/Share Project...* En la ventana que aparece (Figura 4.21), se selecciona, en la casilla *Repository*, el repositorio *Ejemplo1* y se clica sobre el botón *Finish*.

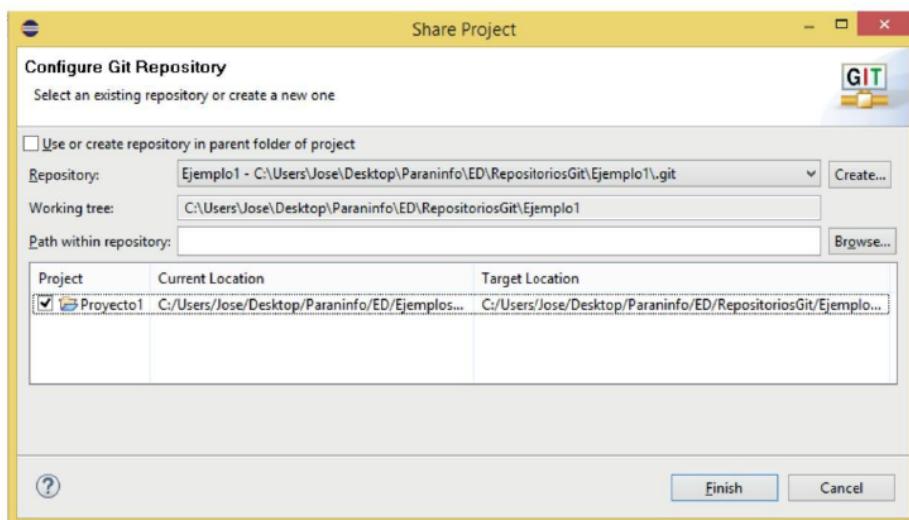


Figura 4.21. Al seleccionar el repositorio creado anteriormente, el proyecto queda asociado a él.

Como se puede observar en la Figura 4.22, en el área de proyectos, delante del nombre del proyecto, aparece el símbolo > y además se indica que es la rama *master*. Con esto, Eclipse informa a la persona que está programando que el proyecto está bajo el control de versiones de Git.

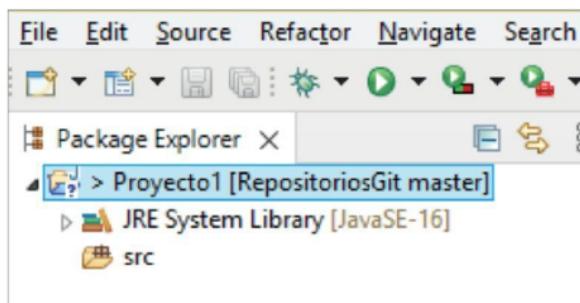


Figura 4.22. Se muestra el proyecto que se acaba de crear en el explorador de paquetes de la vista Java. El símbolo > antes del nombre del proyecto indica que está bajo el control de versiones de Git.

El siguiente paso es crear dentro del proyecto Proyecto1 un paquete llamado *paquete1* y, dentro de este, una clase llamada *Principal* con un método *main*, como se muestra en la Figura 4.23.

```
*Principal.java ×
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

Después, se validan estos cambios, esto es, se confirman (*commit*) para pasar este fichero al área de preparación. Para ello, desde el menú contextual de cualquier elemento del proyecto en el explorador de proyectos se selecciona la opción *Team → Commit...*. Seguidamente, aparece una pantalla (Figura 4.24), en la que en el área *Unstaged Changes*, se muestran los cambios que se han detectado en el área de trabajo que no han sido validados, es decir, que no han pasado al área de preparación. Por otro lado, en el área *Staged Changes*, se ven los cambios confirmados, es decir, los que están en el área de preparación. En la parte derecha de la pantalla, hay un cuadro de texto para escribir el mensaje correspondiente a la validación, información que es obligatorio llenar.

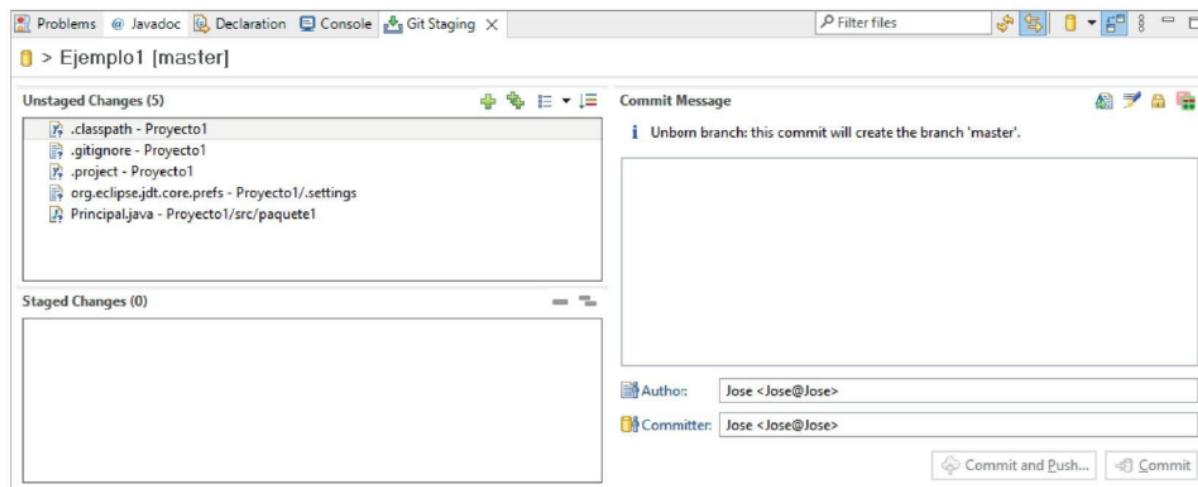


Figura 4.24. Pantalla desde la que se pueden seleccionar los cambios que se desean confirmar (*commit*) para su paso al área de preparación. Para ello, Eclipse obliga a escribir un mensaje en el cuadro de texto *Commit Message*.

Para confirmar los cambios, se arrastran los elementos correspondientes del área *Unstaged Changes* al área *Staged Changes*, lo cual supone pasar los ficheros modificados del área de trabajo al área de preparación. Si se desea que estos cambios sean confirmados en el repositorio local, se debe escribir un mensaje en el cuadro *Commit Message* y clicar en el botón *Commit*. Por otro lado, si se desea que los cambios lleguen al repositorio remoto, habrá que hacer clic en el botón *Commit and Push*.

Para continuar con esta práctica, después de escribir un mensaje en el cuadro *Commit Message* y pulsar sobre el botón *Commit*, se puede realizar algún cambio adicional en el método *main* de la clase *Principal*, por ejemplo, se puede escribir otro mensaje por pantalla. De nuevo, se activa la opción de menú contextual *Team → Commit...*. De esta forma, ya se han llevado a cabo dos confirmaciones o *commits*. Es posible visualizar el historial de versiones de la clase *Principal* seleccionando en su menú contextual la opción *Team → Show in History* (véase la Figura 4.25).

Como se puede ver en la Figura 4.25, para el segundo *commit*, aparecen las etiquetas *master* y *HEAD*. Como ya se indicó, Git llama *master* a la rama principal o tronco. Por otro lado, *HEAD* hace referencia a la versión más reciente de una determinada rama o del tronco.

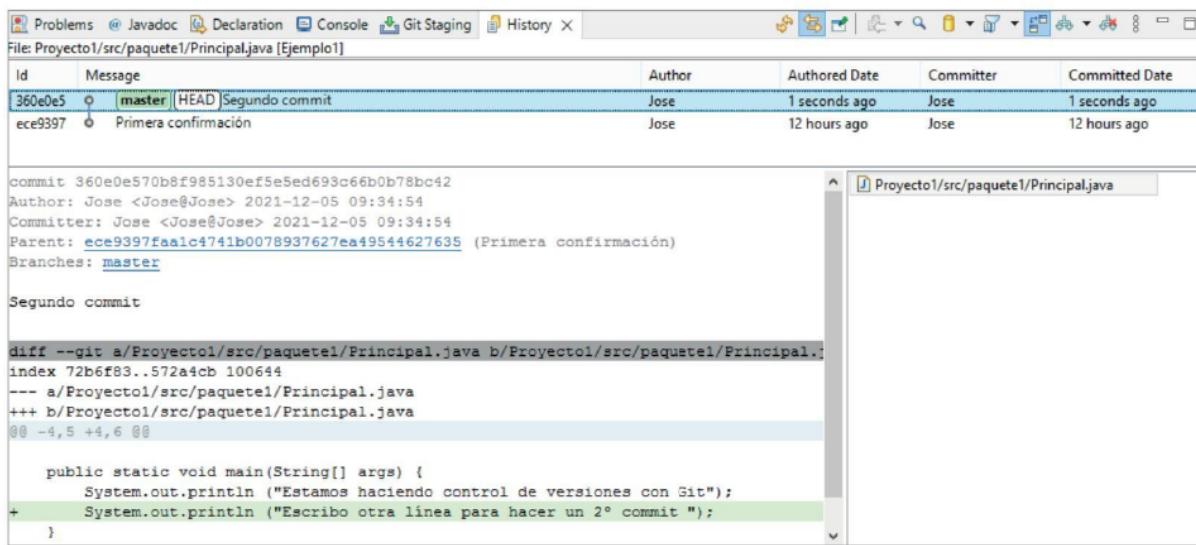


Figura 4.25. Seleccionando la opción del menú contextual Team → Show in History, se puede visualizar el historial de versiones de un fichero. Se puede navegar por las distintas versiones y observar las modificaciones que incorpora cada versión en relación con la anterior.

Al hacer doble clic sobre una versión, se pueden ver más en detalle los cambios realizados (Figura 4.26). Así, si se hace doble clic sobre el primer commit, se muestra en la parte derecha de la pantalla la versión actual y, a la izquierda, el contenido de la versión correspondiente al commit seleccionado, marcando en azul los cambios que ha supuesto una versión en relación con la otra.

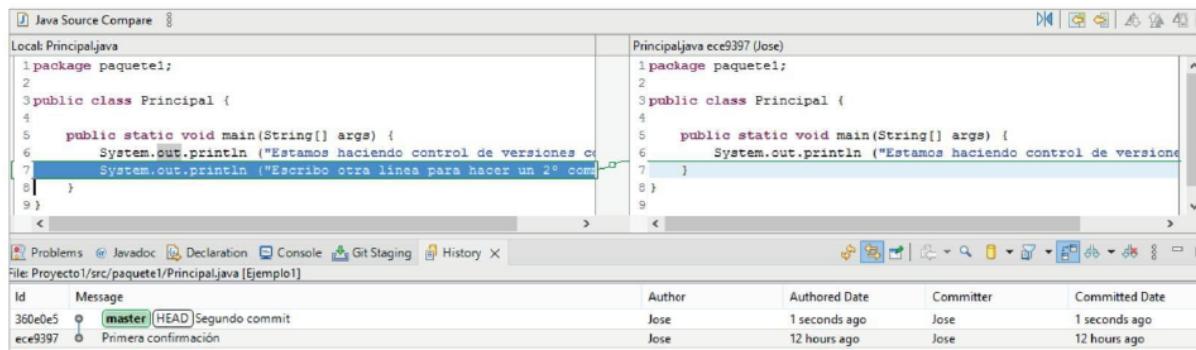


Figura 4.26. Historial de versiones de un fichero: si se hace doble clic sobre uno de los commits, se puede ver en detalle los cambios realizados en esa versión.

Para seguir practicando la validación de cambios, se podría realizar un cambio adicional consistente en mostrar otro mensaje en el método *main* de la clase *Principal* y confirmar este cambio en el repositorio local. De esta manera, se habrían llevado a cabo tres confirmaciones o *commits*.

Creación de un repositorio remoto con GitHub

Para poder utilizar un repositorio remoto, se puede hacer uso de GitHub, que ofrece la posibilidad de crear un repositorio en la nube. Con este fin, se accede al sitio web <https://github.com/> y se pincha sobre el botón *Sign up*, para darse de alta y crear una cuenta gratuita de GitHub.

A continuación, se crea un repositorio haciendo clic en el botón *Create repository*, tras lo cual aparece una pantalla (véase Figura 4.27), en la que se debe dar un nombre al repositorio, se puede proporcionar una descripción para este y se indica que sea público para que pueda ser visto por cualquiera en internet, si bien aquí se elegirá quién o quiénes pueden confirmar los cambios (*commit*).

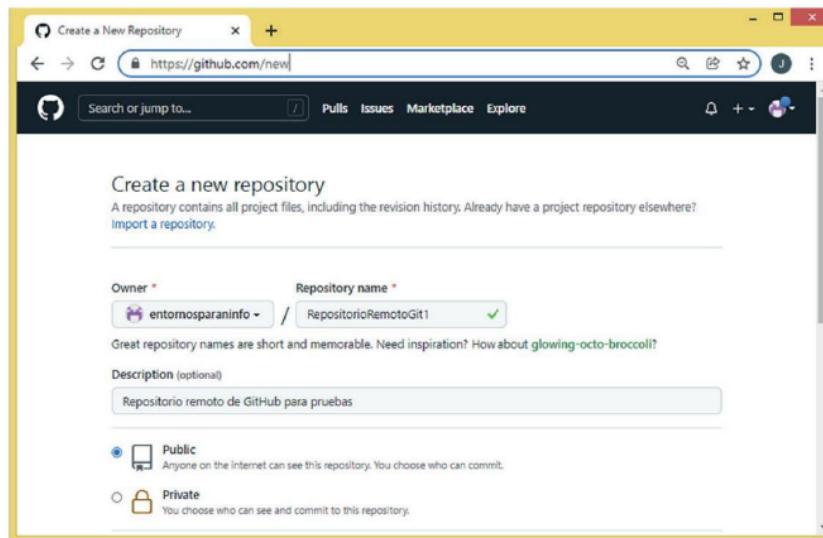


Figura 4.27. Vista de la plataforma GitHub en la que, para crear un repositorio remoto, se debe indicar el nombre, que el repositorio sea público y, opcionalmente, una descripción.

Al hacer clic en el botón *Create repository*, que aparece en la parte inferior de la pantalla, se creará el repositorio. Después aparecerá otra pantalla como la que se representa en la Figura 4.28, en la que se indica la URL del repositorio que se acaba de crear y se proporcionan comandos para trabajar con este repositorio.

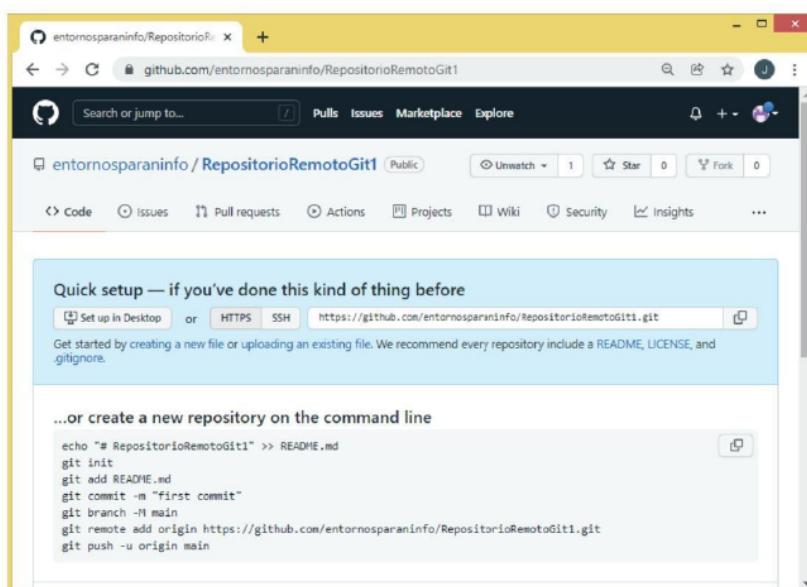


Figura 4.28. Pantalla que aparece tras haber creado un repositorio remoto en GitHub. Un dato muy relevante es la URL del repositorio creado, que se puede leer en la parte superior de la pantalla.

GitHub no hace uso de la autenticación vía HTTPS (*Hypertext Transfer Protocol Secure*) con la contraseña de la cuenta de GitHub por motivos de seguridad. Debido a esto, para solucionar este problema, el cual impide subir ficheros al repositorio remoto, se debe crear un identificador o *token* de acceso personal. Para ello, en nuestra cuenta de GitHub (ícono de arriba a la derecha), se selecciona la opción de menú *Settings* y se clica en el enlace *Developer settings* de entre las opciones que se muestran a la izquierda en la pantalla que aparece después. Seguidamente, se hace clic en el enlace *Personal Access tokens* en el menú de la izquierda (Figura 4.29). En *Note*, se señala el uso que se va a dar al *token*; en *Expiration*, se elige la opción *No expiration*; se marca la casilla de verificación *repo* y, finalmente, se pulsa el botón de la parte inferior de la pantalla *Generate token*.

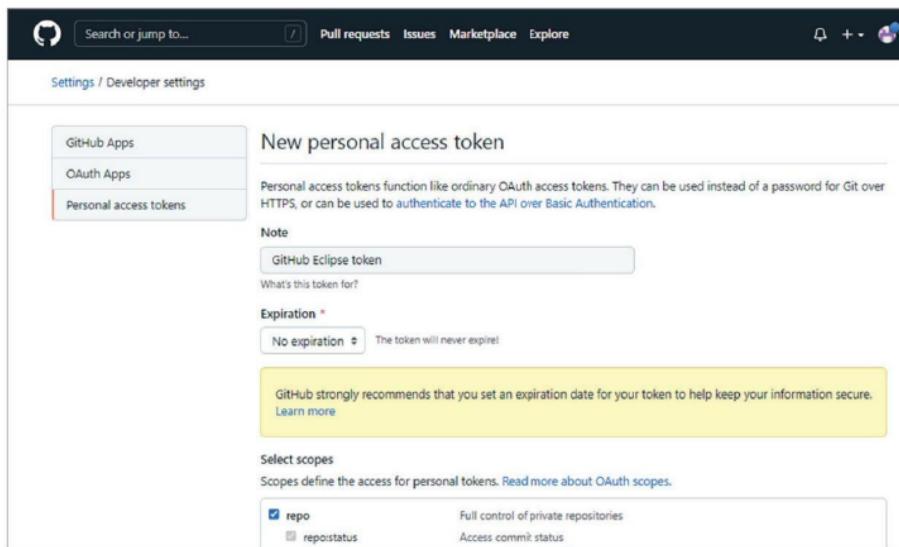


Figura 4.29. Creación de un token de acceso personal en GitHub para poder subir ficheros al repositorio remoto. Se indica para qué se va a usar y que no expire nunca. El token se creará al hacer clic en el botón Generate token.

Tras haber creado el *token*, se puede copiar al portapapeles, pues se necesitará para configurar, desde Eclipse, la subida de archivos al repositorio remoto. Para realizar dicha configuración, en Eclipse se debe abrir la vista de repositorios de Git, para lo que se activa la opción de menú *Window* → *Show view* → *Other* → *Git* → *Git Repositories*.

Desde la vista *Git Repositories*, que se puede ver debajo del explorador de paquetes, para el elemento *Remotes*, se elegirá del menú contextual la opción *Create Remote...* Aparecerá una ventana (Figura 4.30) en la que se pueden dejar las opciones que aparecen por defecto y se hace clic en el botón *Create*.



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

Aparece entonces una ventana para configurar las subidas a este repositorio remoto (Figura 4.31). Para ello, se indica la URL del repositorio remoto que aparecía indicada en GitHub (Figura 4.28). Al introducirla, se rellenan automáticamente los cuadros de texto *Host* y *Repository path*. También, se debe indicar el nombre de usuario y el token de acceso personal de GitHub (no la contraseña del usuario de GitHub). Por último, se pincha sobre el botón *Finish*.

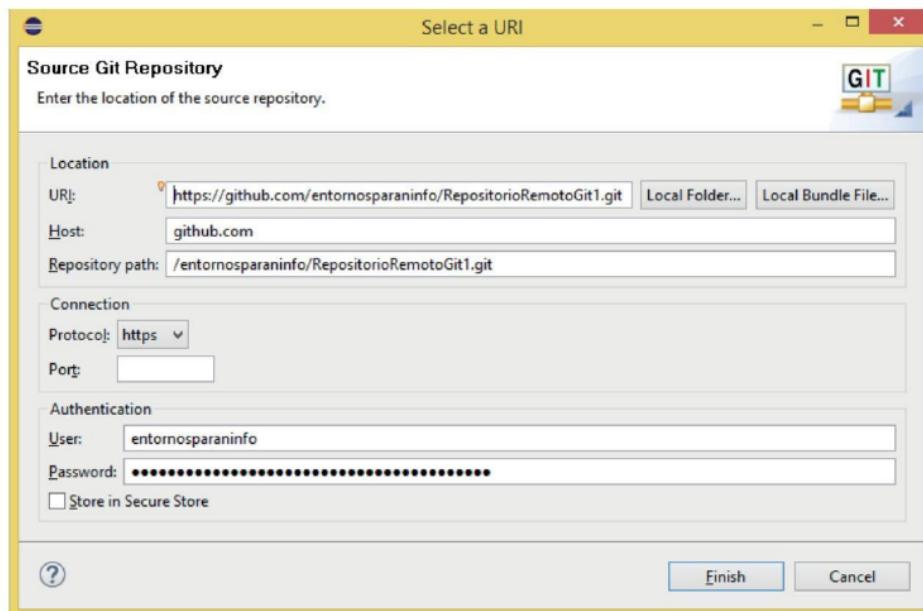


Figura 4.31. Ventana en la que se configura la subida de archivos al repositorio remoto de GitHub, proporcionando la URL de dicho repositorio, el nombre de usuario de GitHub y el token personal de acceso.

Validación de cambios en el repositorio remoto

Una vez se dispone de un repositorio remoto y se han configurado las subidas a dicho repositorio, se pueden subir los ficheros del repositorio local al remoto. Para ello, se disponen de varias opciones:

- Seleccionar para el fichero que se quiere subir la opción del menú contextual *Team* → → *Commit...* y hacer clic en el botón *Commit and Push* (como el que se muestra en la Figura 4.24), o bien en el botón *Push HEAD* si el fichero ya está confirmado en el repositorio local.
- Seleccionar para el fichero que se desea subir la opción del menú contextual *Team* → → *Repository* → *Push Branch 'master'* ...
- Seleccionar para el proyecto que se va a subir la opción del menú contextual *Team* → → *Remote* → *Push...*
- Seleccionar para el proyecto que se desea subir la opción del menú contextual *Team* → *Push Branch 'master'* ...

Al seleccionar el proyecto y la opción de menú *Team* → *Remote* → *Push...*, aparece una ventana (Figura 4.32) con la opción de usar los datos del repositorio ya configurado, que es lo que se seleccionará, y después se hará clic en el botón *Next*.

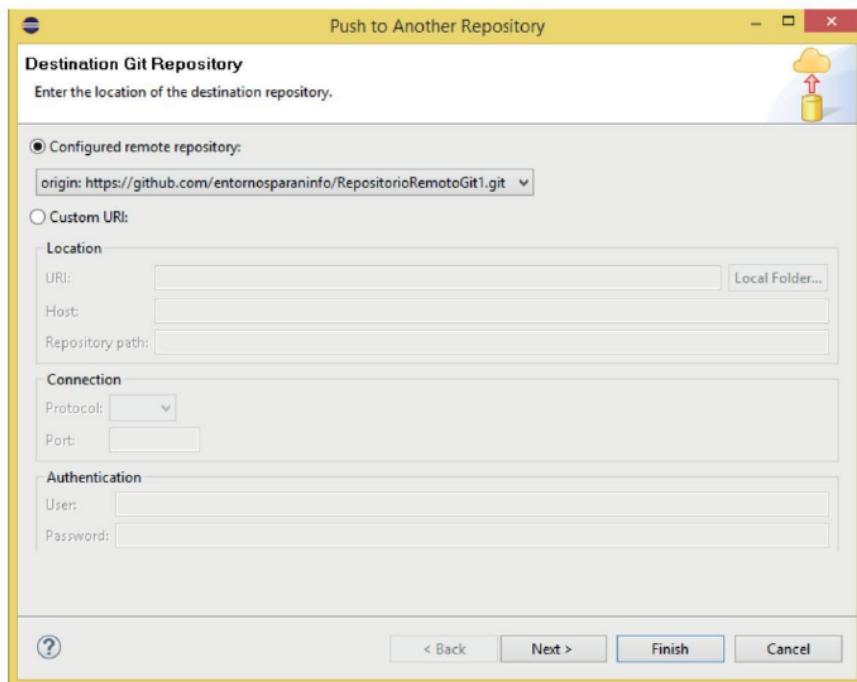


Figura 4.32. Ventana en la que ya aparecen configuradas las subidas al repositorio remoto.

En la siguiente ventana que aparece (Figura 4.33), en la casilla situada bajo *Source ref*, se elige lo que se quiere subir al repositorio remoto (en este caso, *HEAD*) y se clica en el botón *Add Spec*.

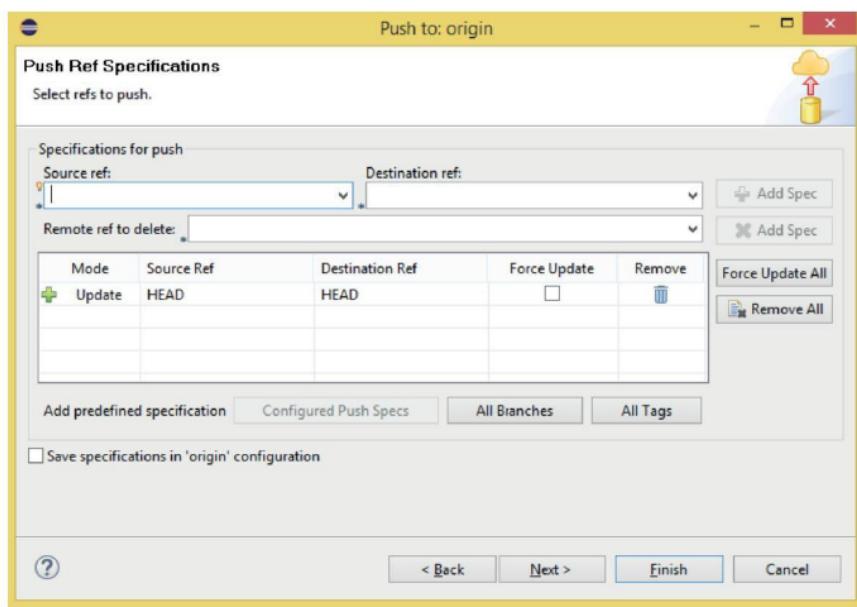


Figura 4.33. Ventana en la que se indica qué parte del proyecto se desea subir al repositorio remoto, seleccionándolo en *Source ref* y haciendo clic en el botón *Add Spec*.

Luego, se pulsa sobre *Finish* y, si se solicitan de nuevo los datos de autenticación (nombre de usuario de GitHub y token personal de acceso), se introducen estos. El proyecto se subirá al repositorio remoto de GitHub. Si se va ahora a GitHub, se verá que, en el repositorio creado (*RepositorioRemotoGit1*), aparece el proyecto que se acaba de subir.

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

Se verá a continuación cómo es posible crear una rama en el repositorio local, fusionarla luego con el tronco y subirla al repositorio remoto.

Para crear una rama, en el área de repositorios Git, se elige para el elemento *Branches* la opción del menú contextual *Switch To → New Branch...* (Figura 4.34). Se asignará un nombre a la rama (por ejemplo, *rama1*) y se dejará marcada la casilla de verificación *Check out new branch*, mediante la cual nos cambiaremos a la rama *rama1*, que se está creando.

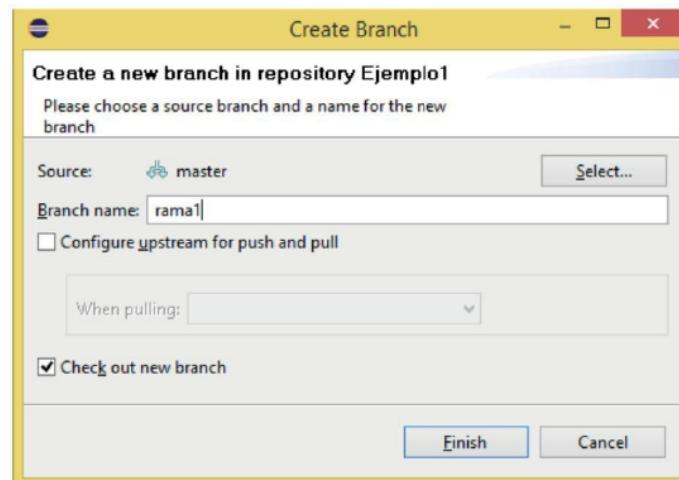


Figura 4.34. Ventana en la que se crea una nueva rama, llamada *rama1*, y nos movemos a ella.

Dentro del proyecto, se crea un nuevo paquete (*paqueteRama1*) con una clase llamada *Rama1*, en cuyo método *main*, se mostrará un mensaje por pantalla.

A continuación, se confirma este cambio en el repositorio local mediante la opción *Commit*. Si se selecciona para el proyecto la opción de menú *Team → Show in History*, se pueden ver todas las confirmaciones (*commits*) realizadas y cómo, para la *rama1*, aparece el indicador *HEAD* (Figura 4.35), pues se trata de la versión más reciente.

Project: Proyecto1 [Ejemplo1]					
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 esta rama al repositorio remoto, se selecciona la rama *rama1* en el área de repositorios Git y la opción del menú contextual *Push Branch...* Se hace clic en el botón *Preview* y luego en *Finish* en la siguiente pantalla, y entonces, la subida ya se habrá realizado, como se podrá observar en GitHub, donde se indica que hay dos ramas (*master* y *rama1*) (véase Figura 4.36).

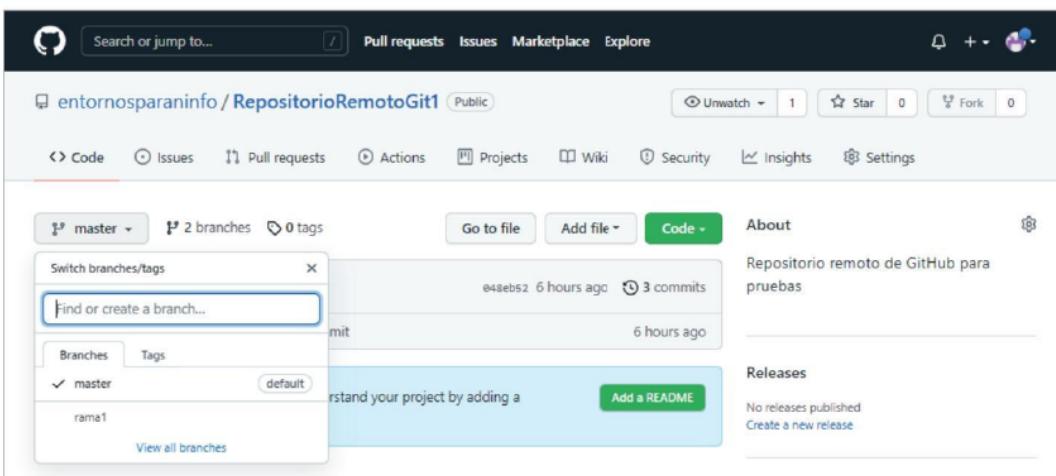


Figura 4.36. Pantalla de inicio de GitHub para el repositorio remoto RepositorioRemotoGit1, donde se puede observar que hay dos ramas (branches), pudiéndose seleccionar la rama master o la rama rama1.

Al seleccionar una rama, se puede navegar por todos sus ficheros.

Fusión de ramas

Para terminar, se explica a continuación cómo se pueden fusionar dos ramas en el repositorio local. A modo de ejemplo, aquí se fusionarán las ramas *master* y *rama1*. Para ello, en Eclipse, en el área de repositorios Git, se hace doble clic sobre la rama *master* con el fin de cambiar a esa rama. De hecho, el programa pregunta si se quiere hacer un *checkout* a la rama *master*, a lo que se debe responder que sí, haciendo clic en el botón *Check Out*. A continuación, en el menú contextual de la rama *master*, se activa la opción *Merge* (Figura 4.37).

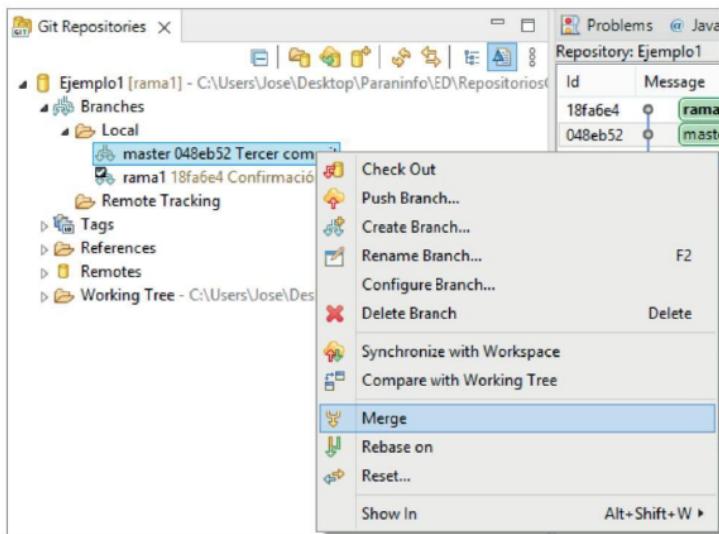


Figura 4.37. Para fusionar la rama rama1 con el tronco (rama master), se selecciona en el área de repositorios Git de Eclipse, para la rama master, la opción del menú contextual Merge.

En la ventana que aparece después (Figura 4.38), se selecciona la rama que se desea fusionar, en este caso, *rama1*, y se clica en el botón *Merge*.

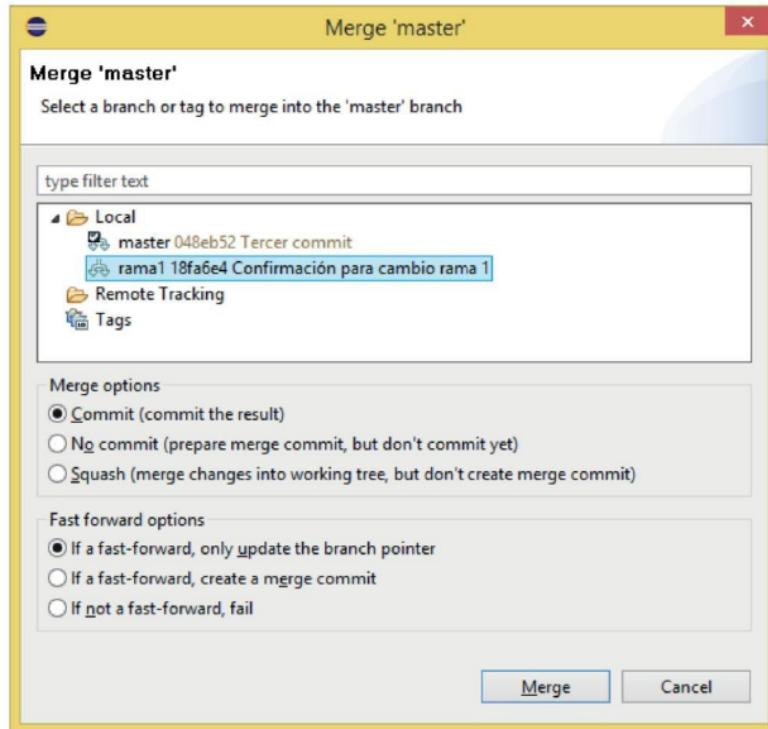


Figura 4.38. Ventana en la que selecciona la rama que se quiere fusionar con la rama master y se confirma haciendo clic en el botón Merge.

Para que esta fusión se plasme en el repositorio remoto, se debe activar la opción *Push Branch* del menú contextual para la rama *master*, y, en la siguiente ventana, hacer clic en el botón *Preview* y, en la siguiente, pulsar el botón *Push*. Se indicará si se ha realizado correctamente la subida al repositorio remoto.

Se puede corroborar si la subida ha sido exitosa accediendo a GitHub y entrando en la rama *master*, en cuya carpeta, *src*, deben aparecer tanto el paquete *paquete1* como el paquete *paqueteRama1*, como se puede ver en la Figura 4.39.

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

Uno de los componentes del software que se incluyó en la definición de software en el Apartado 1.1 fue «información descriptiva tanto en papel como en formas virtuales que describen la operación y uso de los programas», lo cual se refiere a la documentación que debe acompañar a todo software cuando se entrega al cliente. Así, se deben documentar las diferentes fases de cada proyecto, lo que proporcionará información relevante al cliente y favorecerá la realización de las tareas de mantenimiento que afectan a la mayoría del software que se crea.

Básicamente, se pueden distinguir tres tipos de documentos relacionados con el software:

1. Documentos que describen el resultado de las diferentes tareas del desarrollo de software, excepto la de programación: **documentos resultado de la fase de análisis** (especificación de requisitos del software), **de la fase de diseño** (documentación de diseño) y **de la fase de pruebas**.
2. Documentos que describen el **código fuente**.
3. Documentos que describen el **uso del software** (manual de usuario).

No existen normas estandarizadas que describan cómo producir estos documentos. Lo más adecuado es seguir un índice establecido para cada documento en la empresa de desarrollo de software para la que se esté trabajando. Puede haber índices estandarizados para cada uno de los tipos de documentos indicados anteriormente, pero estos siempre se pueden adaptar a la aplicación en concreto que se esté desarrollando. A continuación, se describen los formatos que se pueden seguir para la elaboración de diferentes documentos.

■ ■ 4.4.1. Estructura de los documentos

El documento resultado de la fase de análisis recibe comúnmente el nombre de **Especificación de requisitos del software** (ERS) y según Piattini et al. (2007) su objetivo es «describir los requisitos esenciales (funciones, rendimiento, diseño, restricciones y atributos) del software y de sus interfaces externas».

Seguidamente, se ofrece el índice de ERS propuesto por el IEEE en la norma IEEE 830. No es obligatorio seguir este índice, pero toda ERS debería incluir toda la información que aquí se indica:

1. Introducción.
 - 1.1. Objetivo.
 - 1.2. Ámbito.
 - 1.3. Definiciones, acrónimos y abreviaturas.
 - 1.4. Referencias.
 - 1.5. Visión general del documento.

2. Descripción general.
 - 2.1. Perspectiva del producto.
 - 2.2. Funciones del producto.
 - 2.3. Características del usuario.
 - 2.4. Limitaciones generales.
 - 2.5. Supuestos y dependencias.
 - 2.6. Requisitos futuros.
3. Requisitos específicos.
 - 3.1. Interfaces externas.
 - 3.1.1. Interfaces de usuario.
 - 3.1.2. Interfaces de *hardware*.
 - 3.1.3. Interfaces de *software*.
 - 3.1.4. Interfaces de comunicaciones.
 - 3.2. Requisitos funcionales.
 - 3.3. Requisitos de rendimiento.
 - 3.4. Restricciones de diseño.
 - 3.5. Atributos de calidad del sistema.
 - 3.6. Otros requisitos.
4. Apéndices.

Con relación a la documentación del diseño, esta es definida por el IEEE como una descripción del software creada para facilitar el análisis, la planificación, la implementación y la toma de decisiones. Este documento debe explicar cómo un producto de software será construido para satisfacer un conjunto de requisitos técnicos. Una ERS describe el qué del proyecto, mientras que la documentación del diseño se enfoca en el cómo. Este documento debe servir de base para el equipo de desarrollo y otros interesados en el proyecto. Debería contener toda la información necesaria para que un programador pueda escribir el código. El índice de documentación de diseño propuesto por el IEEE en la norma IEEE 1016 es el siguiente:

1. Introducción.
 - 1.1. Objetivo.
 - 1.2. Ámbito.
 - 1.3. Visión general del documento.
 - 1.4. Material de referencia.
 - 1.5. Definiciones y acrónimos.
2. Visión del sistema.
3. Arquitectura del sistema.
 - 3.1. Diseño de la arquitectura.
 - 3.2. Descripción de la arquitectura.
 - 3.3. Justificación del diseño.

4. Diseño de datos.
 - 4.1. Descripción de los datos.
 - 4.2. Diccionario de datos.
5. Diseño de componentes.
6. Diseño de la interfaz hombre-máquina.
 - 6.1. Visión de la interfaz de usuario.
 - 6.2. Imágenes de las pantallas.
 - 6.3. Objetos de las pantallas y acciones.
7. Matriz de requisitos.
8. Apéndices.

En cuanto a la documentación de las pruebas, es conveniente que, para una buena organización de estas y para asegurar su reutilización, se documenten tanto el diseño de las pruebas como el resultado o ejecución de estas. El estándar IEEE 829 propone una serie de documentos cuya utilidad se describe en el Apartado 3.4. A continuación se muestran los índices propuestos en dicho estándar para cada uno de los documentos.

PLAN DE PRUEBAS DEL SOFTWARE

1. Identificador del documento.
2. Introducción y resumen de elementos y características a probar.
3. Elementos software que se van a probar.
4. Características que se van a probar.
5. Características que no se prueban.
6. Enfoque general de la prueba (actividades, técnicas, herramientas, etc.).
7. Criterios de paso/fallo para cada elemento.
8. Criterios de suspensión y requisitos de reanudación.
9. Documentos que se van a entregar.
10. Actividades de preparación y ejecución de pruebas.
11. Necesidades de entorno.
12. Responsabilidades en la organización y realización de las pruebas.
13. Necesidades de personal y de formación.
14. Esquema de tiempos (con tiempos estimados, hitos, etc.).
15. Riesgos asumidos por el plan y planes de contingencia para cada riesgo.
16. Aprobación y firmas con nombre y puesto desempeñado.

ESPECIFICACIÓN DEL DISEÑO DE LAS PRUEBAS

1. Identificador para la especificación, con una referencia al plan asociado.
2. Características que se van a probar de los elementos software.

3. Detalles sobre el plan de pruebas, incluyendo las técnicas de prueba específicas y los métodos de análisis de resultados.
4. Identificación de cada prueba:
 - 4.1. Identificador.
 - 4.2. Casos que se van a utilizar.
 - 4.3. Procedimientos que se van a seguir.
5. Criterios de paso/fallo de la prueba.

ESPECIFICACIÓN DE CASO DE PRUEBA

1. Identificador único de la especificación.
2. Elementos *software* que se van a probar con indicación de las características que ejercitará cada caso.
3. Especificaciones de cada entrada requerida para ejecutar el caso de prueba, incluyendo las relaciones entre las diversas entradas, por ejemplo, la sincronización de estas.
4. Especificaciones de todas las salidas y las características requeridas (por ejemplo, el tiempo de respuesta) para los elementos que se van a probar.
5. Necesidades de entorno (*hardware*, *software*, personal, etc.).
6. Requisitos especiales de procedimiento.
7. Dependencias entre casos de prueba, identificando los casos que se han de ejecutar antes de este caso de prueba.

ESPECIFICACIÓN DE PROCEDIMIENTO DE PRUEBA

1. Identificador único de la especificación y referencia a la correspondiente especificación de diseño de prueba.
2. Objetivo del procedimiento y lista de casos que se ejecutan con él.
3. Requisitos especiales para la ejecución (por ejemplo, entorno especial o personal especial).
4. Pasos en el procedimiento. Se debe indicar la manera de registrar los resultados y los incidentes de la ejecución, y además:
 - La secuencia necesaria de acciones para preparar la ejecución.
 - Acciones necesarias para empezar la ejecución.
 - Acciones necesarias durante la ejecución.
 - Cómo se realizarán las medidas, por ejemplo, el tiempo de respuesta.
 - Acciones necesarias para suspender la prueba.
 - Puntos para reinicio de la ejecución y acciones necesarias para el reinicio en estos puntos.
 - Acciones necesarias para detener ordenadamente la ejecución.
 - Acciones necesarias para restaurar el entorno y dejarlo en la situación existente antes de las pruebas.
 - Acciones necesarias para tratar los acontecimientos anómalos.

HISTÓRICO DE PRUEBAS

1. Identificador.
2. Descripción de la prueba: elementos probados y entorno de la prueba.
3. Anotación de datos sobre cada hecho ocurrido:
 - Fecha y hora.
 - Identificador del informe de incidente.
4. Otras informaciones.

INFORME DE INCIDENTE

1. Identificador.
2. Resumen del incidente.
3. Descripción de datos objetivos (fecha y hora, entradas, resultados esperados, etc.).
4. Impacto que tendrá sobre las pruebas.

INFORME RESUMEN DE LAS PRUEBAS

1. Identificador.
2. Resumen de la evaluación de los elementos probados.
3. Variaciones del *software* respecto a su especificación de diseño, así como las variaciones en las pruebas.
4. Valoración de la extensión de las pruebas (cobertura lógica, funcional, de requisitos, etc.).
5. Resumen de los resultados obtenidos en las pruebas.
6. Evaluación de cada elemento *software* sometido a pruebas.
7. Resumen de las actividades de prueba, incluyendo el consumo de todo tipo de recursos.
8. Firmas y aprobaciones de quienes deban supervisar el informe.

Actividad resuelta 4.2

Generación de documentación para las fases de análisis, diseño y pruebas

En este apartado, se han descrito una serie de documentos propuestos por el IEEE que describen el resultado de las etapas de análisis, diseño y pruebas de una aplicación. ¿Es necesario seguir el índice propuesto en estas normas? Justifica tu respuesta.

Solución

No es necesario. Se trata de unos documentos propuestos por el IEEE en los que se incluyen todos los apartados que debería incluir cada documento, como por ejemplo, la ERS. No obstante, aunque no es preciso seguir los índices de estos documentos, en cualquier documento que se elabore, sí se debería incluir toda la información a la que se refiere cada uno de los puntos de los citados índices.

■ ■ ■ 4.4.2. Generación de documentación

La generación automática de documentación se usa fundamentalmente para la documentación del código fuente.

La documentación del código fuente, aunque parezca una tarea innecesaria o poco interesante, en la vida real, es una tarea de gran importancia porque un porcentaje muy importante del esfuerzo de desarrollo de software de una organización se dedica a la tarea de mantenimiento, en cualquiera de sus modalidades (mantenimiento correctivo, preventivo o adaptativo). Todos los estudios realizados en relación con el esfuerzo de la tarea de mantenimiento, cifran su coste en más del 50 % del esfuerzo dedicado al desarrollo de software.

Hay que tener en cuenta que todos los programas que tengan éxito tendrán que sufrir cambios a lo largo de su uso por errores no detectados, porque se requieran nuevas funcionalidades o cambios en funcionalidades existentes, o porque sea necesario adaptar el software a nuevos entornos.

Además, es muy probable que las personas que tengan que realizar modificaciones sobre un determinado programa no sean las mismas personas que lo elaboraron. Y aunque lo sean, lo cual no es muy frecuente, no es muy probable que recuerden por qué escribieron cierto código. La utilidad de la documentación debe ser precisamente facilitar la modificación de programas, es decir, la tarea de mantenimiento del software. Se debe posibilitar la rápida detección de errores y su depuración y la detección de aquellas partes del software que es necesario modificar, por ejemplo, por la inclusión de nuevas funcionalidades.

Es importante documentar o comentar la utilidad de cada clase, de cada método, de cada variable, el algoritmo que se usa para resolver un problema, indicar el autor de cada uno de estos elementos y la fecha de la última modificación, etcétera.

Se pueden documentar programas escritos en casi cualquier lenguaje de programación mediante el empleo de herramientas de generación automática, como por ejemplo, el sistema JavaDoc que sirve para documentar programas escritos en Java.

JavaDoc permite generar documentación de código escrito en Java en formato HTML. Para que esto resulte útil, es necesario seguir una serie de recomendaciones de JavaDoc, herramienta que generará un fichero que incluirá el código y la documentación. JavaDoc recomienda lo siguiente en relación con los comentarios.

Los tipos de comentarios que se pueden utilizar son los siguientes:

- **Comentarios de línea:** comienzan con los caracteres “//” y finalizan con la línea.
- **Comentarios de varias líneas:** pueden abarcar varias líneas, empiezan con “/*” y terminan con “*/”.
- **Comentarios de documentación JavaDoc:** estos comentarios deben comenzar con “/**” y finalizar con “*/”, y, también, pueden abarcar varias líneas. Sin embargo, cada línea dentro de estos comentarios debe comenzarse con el símbolo * y se pueden incluir etiquetas HTML dentro de ellos. Estos comentarios no se pueden

colocar en cualquier lugar de los programas, sino que deben situarse antes de la declaración de una clase, un atributo o un método. Incluyen dos partes: una descripción y un bloque de etiquetas o *tags*. Las etiquetas más utilizadas se muestran en la Tabla 4.1.

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.

Se muestra a continuación la descripción de la clase *CuentaDoc*, que es como la clase *Cuenta* que se creó en el Apartado 3.6, pero con una serie de comentarios de JavaDoc añadidos.

```

1 package CuentaPrueba;
2
3 /**
4 * <h3> Clase CuentaDoc, para contener la información de cada una
5 * de las
6 * cuentas bancarias </h2>
7 * @version 01-2021
8 * @author Jose Piñeiro
9 * since 14-02-2021
10 */
11 public class CuentaDoc {
12     private String número;      //Número de la cuenta bancaria
13     private float saldo;        //Saldo de la cuenta bancaria en euros
14
15     /**
16      * Constructor de la clase CuentaDoc con todos los parámetros
17      * @param numCta Número de la cuenta bancaria
18      * @param saldoCta Saldo inicial de la cuenta
19      */
20     public CuentaDoc(String numCta, float saldoCta){
21         número= numCta;

```

```
21         saldo = saldoCta;
22     }
23
24     /**
25      * Método de selección del número de la cuenta
26      * @return Número de la cuenta bancaria
27      */
28     public String getNúmero(){
29         return número;
30     }
31
32     /**
33      * Método de selección del saldo de la cuenta
34      * @return Saldo de la cuenta bancaria
35      */
36     public float getSaldo(){
37         return saldo;
38     }
39
40     /**
41      * Método de acceso al número de la cuenta
42      * @param numCta Número de cta. que se desea asignar a la cuenta
43      * bancaria
44      */
45     public void setNúmero(String numCta){
46         número = numCta;
47     }
48
49     /**
50      * Método de acceso al saldo de la cuenta
51      * @param saldoCta Valor que se desea asignar al saldo de la
52      * cuenta
53      */
54     public void setSaldo(float saldoCta){
55         saldo = saldoCta;
56     }
57
58     /**
59      * Método que ingresa dinero en la cuenta, incrementando su
60      * saldo
61      * @param importe Valor con que se desea incrementar el saldo
62      */
63     public void ingresarDinero(float importe){
64         saldo = saldo + importe;
65     }
66
67     /**
68      * Método que saca dinero de la cuenta, decrementando su saldo
69      * @param importe Valor con que se desea decrementar el saldo
70      */
```

```

67     public void extraerDinero(float importe) {
68         saldo = saldo - importe;
69     }
70     /**
71      * Método que muestra el nº de la cuenta y su saldo
72      */
73     public void mostrarCuenta() {
74         System.out.println ("Nº cuenta: " + getNúmero());
75         System.out.println ("Saldo: " + getSaldo() + " €");
76     }
77 }
```

Para ejecutar JavaDoc desde Eclipse, se activa la opción de menú *Project → Generate Javadoc*. En la pantalla que se muestra en la Figura 4.40, se debe:

1. Indicar debajo de *Javadoc command* la ruta en la que se encuentra el archivo ejecutable de JavaDoc, *javadoc.exe*. Se puede seleccionar la ubicación en el equipo haciendo clic en el botón *Configure*. Se debe buscar el archivo *javadoc.exe* en la carpeta *bin*, que estará dentro de la carpeta en la que se encuentra el JDK.
2. Seleccionar el proyecto y las clases que se desean documentar.
3. Seleccionar la visibilidad máxima de los elementos que se desean documentar. Si se selecciona *Private*, se está indicando que se desean documentar todos los miembros (atributos y métodos), incluso los que tienen asignada la visibilidad *private*.
4. Indicar, a la derecha de *Destination*, la carpeta donde se almacenará el código HTML.

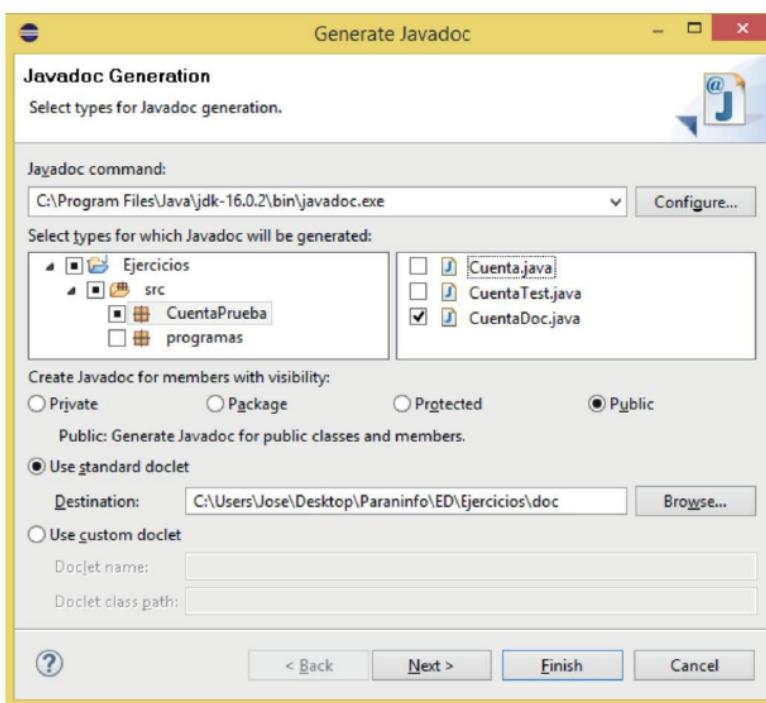


Figura 4.40. Generación de documentación automática con JavaDoc para la clase CuentaDoc.

Seguidamente, se clica en el botón *Next* y, en la siguiente ventana, se indica, en la parte superior, el nombre del documento que se va a generar y se seleccionan las opciones deseadas para la generación del documento HTML. Finalmente, se pulsa el botón *Finish*.

Si nos dirigimos a la carpeta especificada en *Destination* por medio del explorador de archivos, se puede hacer doble clic en el documento *index.html*, cuyo contenido es el que se muestra en la Figura 4.41.

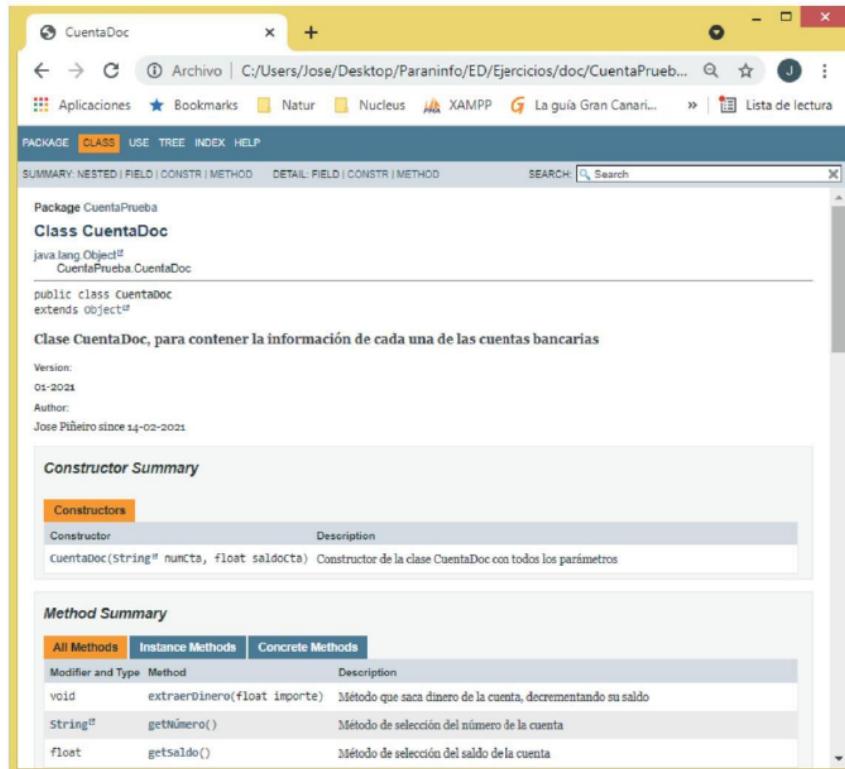
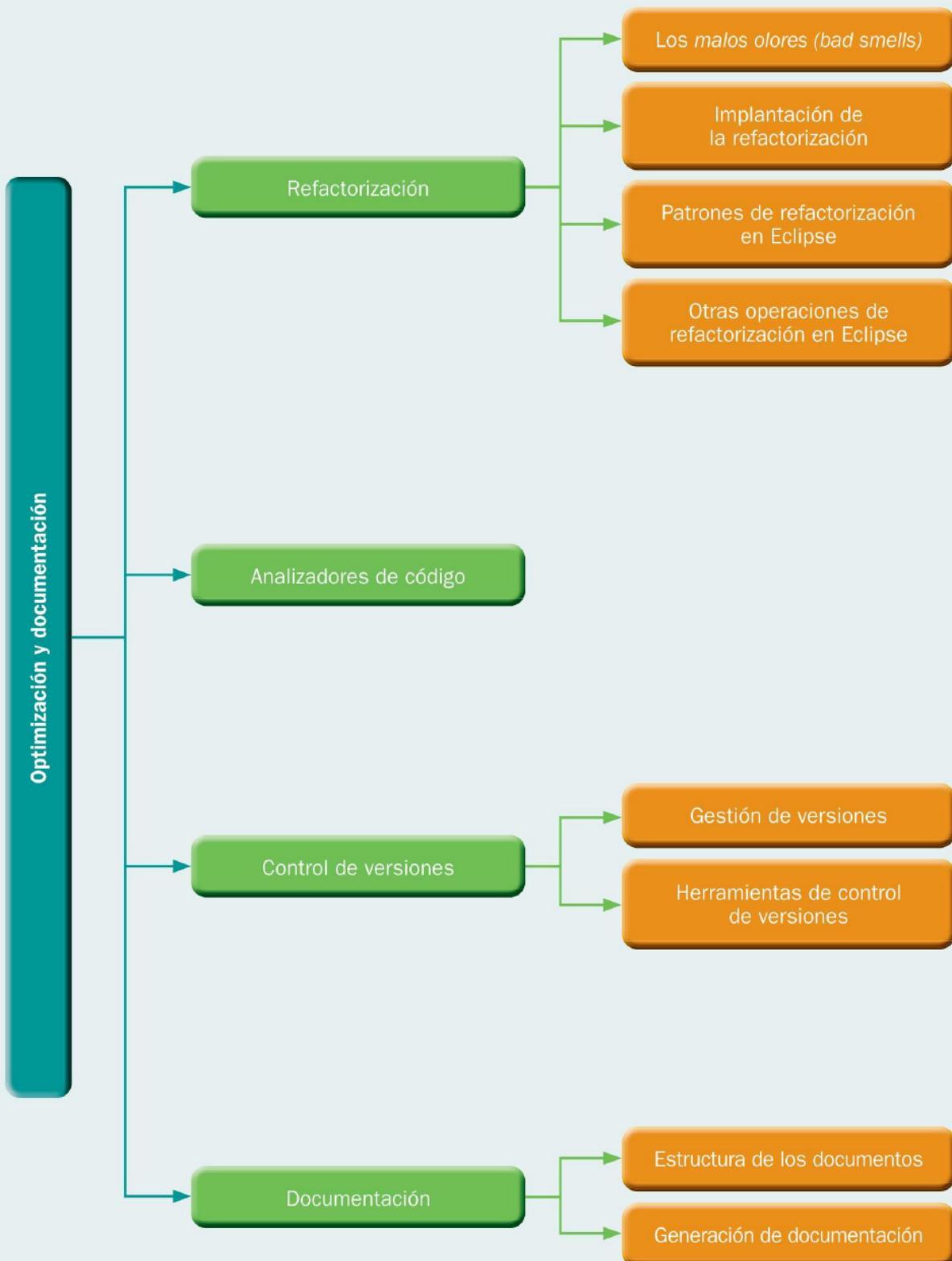


Figura 4.41. Documentación generada con JavaDoc para la clase CuentaDoc.

Actividad propuesta 4.3

Documentación de una clase con JavaDoc

Documenta la clase *Punto* del proyecto *figuras* de manera similar a como se ha hecho para la clase *CuentaDoc* y obtén, como resultado, un documento HTML.



Actividades de comprobación

- 4.1.** En relación con el momento de aplicación de la refactorización, indica cuál es la afirmación verdadera:
- a) Es más adecuado aplicar la refactorización sobre una aplicación ya finalizada.
 - b) Es mejor realizar pequeñas refactorizaciones a lo largo del proceso de desarrollo de software.
 - c) Es conveniente aplicar refactorizaciones tras la adición de cada nueva funcionalidad al software.
 - d) Las respuestas b y c son correctas.
- 4.2.** ¿Qué nombre recibe el *mal olor (bad smell)* consistente en que existen demasiadas ramas y bucles en un método?
- a) Método largo (*long method*).
 - b) Código divergente (*divergent code*).
 - c) Código duplicado (*duplicated code*).
 - d) Complejidad ciclomática (*cyclomatic complexity*).
- 4.3.** ¿Cuál de los siguientes patrones de refactorización en Eclipse es aplicable a un bloque de código?
- a) *Encapsulate Field*.
 - b) *Extract Method*.
 - c) *Change Method Signature*.
 - d) *Pull up*.
- 4.4.** ¿Cuál de los siguientes patrones de refactorización se debe aplicar en Eclipse para mover un atributo de una clase a su superclase?
- a) *Move*.
 - b) *Pull up*.
 - c) *Pull down*.
 - d) *Inline*.
- 4.5.** ¿Cuál de los siguientes patrones de refactorización en Eclipse es aplicable sobre un atributo de una clase?
- a) *Move*.
 - b) *Extract Class*.
 - c) *Extract Local Variable*.
 - d) *Encapsulate Field*.
- 4.6.** En cuanto a las diferencias entre las herramientas automáticas para la refactorización y las que se usan para el análisis estático de código (analizadores de código):
- a) Los dos tipos de herramientas son capaces de detectar automáticamente defectos en el código.
 - b) Los dos tipos de herramientas pueden ser empleadas por las personas que llevan a cabo la programación como medio para corregir defectos cometidos en esta.
 - c) Los dos tipos de herramientas son capaces de detectar automáticamente *malos olores (bad smells)*.
 - d) Ninguna de las respuestas anteriores es correcta.

- 4.7. Indica cuál de las siguientes afirmaciones es verdadera en relación con el código duplicado:**
- a) Las herramientas de refactorización de Eclipse pueden emplearse para la detección automática de código duplicado.
 - b) Existe algún patrón de refactorización en Eclipse que se puede emplear para solucionar el problema del código duplicado.
 - c) El analizador de código PMD se puede emplear para la detección automática de código duplicado.
 - d) Las respuestas b y c son correctas.
- 4.8. En Git un archivo que está pendiente de confirmación, ¿en qué área se encuentra?**
- a) En el repositorio local (*commit area*).
 - b) En el área de trabajo (*working area*).
 - c) En el área de preparación (*staging area*).
 - d) En el repositorio remoto.
- 4.9. ¿Qué orden se emplea en Git para pasar cambios realizados en el repositorio local al repositorio remoto?**
- a) Pull.
 - b) Push.
 - c) Check out.
 - d) Commit.
- 4.10. ¿Qué orden se emplea en Git para cambiar de rama?**
- a) Pull.
 - b) Push.
 - c) Check out.
 - d) Commit.
- 4.11. En Git, ¿a qué elemento se le atribuye el nombre de *master*?**
- a) A la última versión de una rama.
 - b) A la última rama que se ha creado.
 - c) A la rama principal o tronco.
 - d) Al repositorio remoto.
- 4.12. ¿JavaDoc se emplea para generar qué tipo de documentación?**
- a) Documentos que describen el uso del software (manual de usuario).
 - b) Documentos que describen el código fuente.
 - c) Documentos que describen el resultado de las fases de análisis y diseño del software.
 - d) Todas las respuestas son correctas.

Actividades de aplicación

- 4.13. Describe en qué consiste la técnica de la refactorización.
- 4.14. ¿Es mejor aplicar la refactorización continua o *a posteriori*? , ¿por qué?
- 4.15. ¿En qué consiste el patrón de refactorización *Extract Local Variable* de Eclipse?
- 4.16. ¿En qué consiste el patrón de refactorización *Encapsulate Field* de Eclipse?
- 4.17. En la aplicación *figuras*, cambia el nombre de la clase *Triángulo* para eliminar su tilde, de forma que su nuevo nombre sea *Triangulo*. Elimina también las tildes de los nombres de los métodos abstractos *área()* y *perímetro()* de la clase *Figura*.
- 4.18. En la clase *PruebaFigura* de la aplicación *figuras*, elimina las tildes de las variables *opción* y *mostrarPerímetro* del método *main*, del nombre de los métodos *mostrarMenú()* y *procesarTriángulo()* y de la variable *opción* del método *mostrarMenú()*.
- 4.19. En la clase *PruebaFigura*, sustituye la variable local *opcion* del método *main* por un atributo privado de la clase.
- 4.20. ¿Qué diferencias hay entre las herramientas para la refactorización y los analizadores de código?
- 4.21. ¿Es posible en Eclipse consultar las reglas que emplea el analizador de código PMD al realizar sus análisis? Si es posible, ¿cómo se pueden consultar?
- 4.22. ¿Es posible en Eclipse modificar las reglas que emplea el analizador de código PMD al realizar sus análisis? Si es posible, ¿cómo se pueden modificar?
- 4.23. Crea en Eclipse un repositorio local llamado *Ejemplo2* dentro de la carpeta *RepositoriosGit* que se creó anteriormente. Crea un proyecto Java en Eclipse llamado *Proyecto2* dentro de la carpeta *EjemplosControlVersionesGit*. Crea dentro del proyecto una clase mediante la que se muestre un mensaje por pantalla. Confirma los cambios realizados en el repositorio local.

Crea un repositorio remoto en GitHub llamado *RepositorioRemotoGit2* y pasa el contenido del repositorio local al repositorio remoto.

Crea una rama llamada *rama1*, en la cual debes añadir una nueva clase que muestre dos mensajes por pantalla, y confírmala en el repositorio local. Añade para la clase creada en la rama *rama1* dos nuevos mensajes por pantalla, valida los cambios en el repositorio local y súbela al repositorio remoto. Fusiona esta rama con el tronco y súbela al repositorio remoto.

Crea otra rama llamada *rama2* con una nueva clase que muestre un mensaje por pantalla y confirma estos cambios en el repositorio local. Visualiza el histórico de versiones del proyecto. Confirma esta rama en el repositorio local y súbela al repositorio remoto. Luego fusiona esta rama con el tronco y sube los cambios al repositorio remoto.

- 4.24.** ¿A qué hacen referencia las siglas ERS? Este documento se obtiene como resultado de una fase del desarrollo de software, ¿cuál es esta fase?

Actividades de ampliación

- 4.25.** En el Apartado 4.1, se indicó que existen tres tipos de mantenimiento (perfectivo, correctivo y adaptativo), sin embargo, se suele hablar a veces de mantenimiento preventivo. ¿En qué consiste este tipo de mantenimiento?, ¿tiene alguna relación con la refactorización?
- 4.26.** Uno de los *malos olores* que se señaló en el Apartado 4.1.1 es el llamado *cirugía a tiros* (*shotgun surgery*), consistente en que un cambio en una clase conlleva la realización de cambios en otras muchas clases. Investiga cuál puede ser el motivo de este *mal olor* y cómo se puede solucionar.
- 4.27.** Otro de los *malos olores* que se describió en el Apartado 4.1.1 es el llamado código divergente (*divergent code*), que consiste en que un cambio en el sistema conlleva muchos cambios en una misma clase. Se trata del *mal olor* contrario a *cirugía a tiros*. Investiga cuál puede ser el motivo de este *mal olor* y cómo se puede solucionar.
- 4.28.** En el Apartado 4.3.3, se ha empleado la herramienta de control de versiones Git integrada en el IDE Eclipse, pero también es posible emplearla de manera independiente en modo línea de comandos. ¿Desde qué página web es posible realizar la descarga? ¿En qué sistemas operativos se puede instalar Git?
- 4.29.** ¿Es posible instalar Git con interfaz gráfica sin estar integrado en un IDE? En caso de que sea posible, ¿desde qué página web se puede realizar la descarga correspondiente y en qué sistemas operativos se puede instalar?
- 4.30.** Busca la utilidad de los siguientes comandos que se pueden emplear en Git en modo línea de comandos: *git init* y *git diff*.
- 4.31.** Investiga si es posible emplear Git integrado en el IDE Apache NetBeans. En caso de que sea posible, ¿es necesario realizar alguna instalación?

Enlaces web de interés

-  **Baeldung** - <https://www.baeldung.com/Eclipse-refactoring>
(Tutorial sobre temas relacionados con el desarrollo en Java)
-  **Linuxtopia** - https://www.linuxtopia.org/online_books/eclipse_guides.html
(Página web con guías sobre Linux y el desarrollo de software. Incluye una sección sobre Eclipse)
-  **Refactoring.Guru** - <https://refactoring.guru/es>
(Sitio web sobre refactorización, patrones de diseño, principios SOLID y otros temas relacionados con la programación)
-  **Git-scm** - <https://git-scm.com/>
(Sitio web dedicado a la gestión de la configuración del software empleando la herramienta Git)
-  **PMD** - <https://pmd.github.io/>
(Sitio web sobre el analizador de código PMD)