




UD04 – OPTIMIZACIÓN Y DOCUMENTACIÓN

ENTORNOS DE DESARROLLO

JORGE RODRÍGUEZ CHACÓN
IES JACARANDÁ



1 Refactorización

Definición de refactorización: “La refactorización consiste en realizar una transformación al software preservando su comportamiento, modificando su estructura interna para mejorarlo” (William F. Opdyke, 1992).

En múltiples ocasiones durante el transcurso de un proyecto de grandes proporciones o de larga duración es frecuente encontrarse con que necesitamos reevaluar y modificar código creado anteriormente, o ponerse a trabajar con un proyecto de otra persona, para lo cual antes se deberá estudiar y comprender. A pesar de los comentarios o la documentación del proyecto en cuestión, la refactorización ayuda a tener un código más sencillo de comprender, más compacto, más limpio, y por supuesto, más fácil de modificar.

Las refactorizaciones suelen efectuarse en la fase de mantenimiento del desarrollo de software, por lo que podría verse como un tipo de mantenimiento preventivo con el propósito de simplificar el código en vista de futuras necesidades y funcionalidades que añadir al software del programa.

La piedra angular de la refactorización se encuentra en una única y sencilla frase: “No cambia la funcionalidad del código ni el comportamiento del programa, el programa deberá comportarse de la misma forma antes y después de efectuarse las refactorizaciones”.

1.1 Tabulación

La tabulación puede que no sea una técnica o una práctica propia de la refactorización en sí misma, ya que no se modifica el código. No obstante, con la tabulación, el código queda más claro, con lo que también resulta más fácil de ver y entender, que es parte del objetivo de la refactorización.

Definición de tabulación: “La tabulación (también llamada sangrado o indentación) nos permite visualizar el código organizado jerárquicamente sangrando las líneas de código dentro de los bloques de código”.

Hoy en día cualquier entorno de desarrollo con el que trabajemos nos maquetará el código del programa con un sangrado y nos coloreará las palabras reservadas, tipos de variables, nombres de clases...

En el siguiente ejemplo se puede observar el sangrado de un programa sencillo, y a continuación el mismo programa sin sangrado. Se puede observar que hay una clase con dos métodos al mismo nivel, mientras que las instrucciones y estructuras de control que se encuentran dentro de los métodos están en un nivel inferior, y así consecutivamente dentro de cada instrucción y estructura de control.

Sin embargo, en el ejemplo sin sangrado el código no parece tan sencillo, hay que estar más pendientes de dónde acaban las instrucciones y bloques, por lo que resulta más complicado y engorroso trabajar sin código tabulado. Aun así, se puede pensar que no es para tanto, que todavía se podría modificar este código sin necesidad de sangrados, pero imaginaos ahora que tengáis que trabajar con un código de mil líneas sin tabular.

En Eclipse se podrá corregir la indentación del código con un simple atajo de teclado “Ctrl+I”.

<pre> package refactoring; import java.util.ArrayList; import comun.Persona; public class Sangrado { private Persona[] participantes; Boolean esParticipante (Persona x) { for (int i=0; i<participantes.length; i++) { if (participantes[i].getDNI() == x.getDNI()) { return true; } } return false; } ArrayList<Persona> getHijosFromParticipantes (Persona p) { ArrayList<Persona> hijos = new ArrayList<>(); if (participantes.length>0) { for (Persona aux:participantes) { if (aux.getPadre() == p) { hijos.add(aux); } } } return hijos; } } </pre>	<pre> package refactoring; import java.util.ArrayList; import comun.Persona; public class Sangrado { private Persona[] participantes; Boolean esParticipante (Persona x) { for (int i=0; i<participantes.length; i++) { if (participantes[i].getDNI().equals(x.getDNI())) { return true; } } return false; } ArrayList<Persona> getHijosFromParticipantes (Persona p) { ArrayList<Persona> hijos = new ArrayList<>(); if (participantes.length>0) { for (Persona aux:participantes) { if (aux.getPadre() == p) { hijos.add(aux); } } } return hijos; } } </pre>
---	--

1.2 Patrones de refactorización

Los patrones de refactorización, comúnmente llamados catálogos de refactorización o métodos de refactorización son diversas prácticas concretas para refactorizar nuestro código. Plantean casos o problemas concretos y su resolución refactorizadora, pudiendo ver así un antes y después y sobre todo comprendiendo el porqué. Sin lugar a dudas el catálogo de patrones de refactorización más extendido y aceptado es el de Martin Fowler, "Refactoring: Improving the Design of Existing Code".

1.2.1 Extraer Método

- Se tiene un fragmento de código que puede agruparse.
- Conviertes el fragmento en un método cuyo nombre explique el propósito del método.

```

public void imprimirTodoNoRefactor() {
    imprimirBanner();

    //detalles de impresión
    System.out.println("nombre: " + getNombre());
    System.out.println("cantidad: " + getCargoPendiente());
}

```



```

public void imprimirTodoRefactor() {
    imprimirBanner();
    imprimirDetalle(getNombre(), getCargoPendiente());
}

private void imprimirDetalle(String n, int cp) {
    System.out.println("nombre: " + n);
    System.out.println("cantidad: " + cp);
}

```

1.2.2 Encapsular atributo

- Tenemos un atributo público.
- Lo convertimos a privado y le creamos métodos de acceso.

```
public String atributoNoEncapsulado;
```



```
private String atributoEncapsulado;

public String getAtributoEncapsulado() {
    return atributoEncapsulado;
}

public void setAtributoEncapsulado(String atributoEncapsulado) {
    this.atributoEncapsulado = atributoEncapsulado;
}
```

1.2.3 Reemplazar número mágico con constante

- Tenemos un literal u operación con un significado particular.
- Creamos una constante, la nombramos significativamente y la sustituimos por el literal.

```
double energiaPotencial(double masa, double altura) {
    return masa * altura * 9.81;
}
```

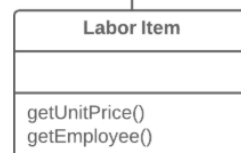
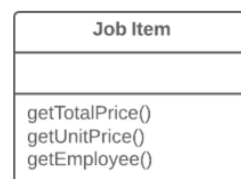
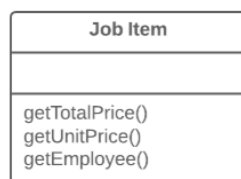


```
private static final double _CONSTANTE_GRAVITACIONAL = 9.81;

double energiaPotencialRefactor(double masa, double altura) {
    return masa * altura * _CONSTANTE_GRAVITACIONAL;
}
```

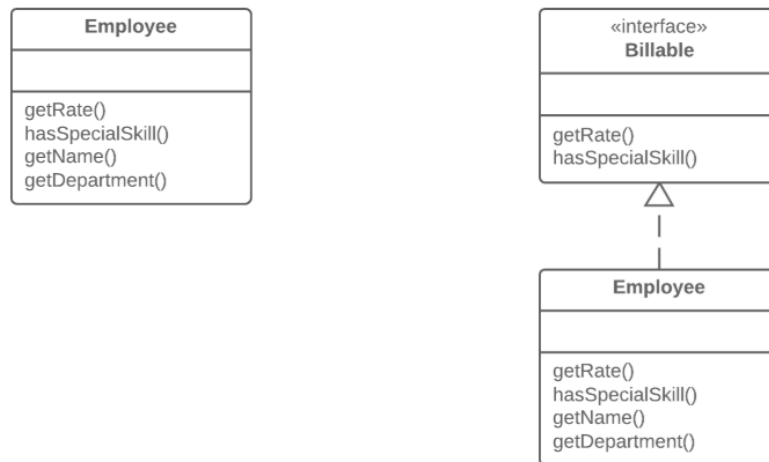
1.2.4 Extraer subclase

- Una clase tiene propiedades que solo son usadas en determinadas instancias.
- Creamos una subclase para dicho conjunto de propiedades.



1.2.5 Extraer interfaz

- Múltiples clases usan métodos similares.
- Creamos una interfaz que contiene los métodos comunes. Y el resto de las clases la implementan.



1.3 Malos olores

Los “malos olores” son una relación de malas prácticas de desarrollo, indicadores de que nuestro código podría necesitar ser refactorizado. No siempre que detectemos un posible “mal olor” es un fallo de diseño en nuestro código y deberemos refactorizarlo, pero nos ayudará saber reconocer los indicadores y valorar si ese indicador es válido y tendremos que refactorizar.

Los “malos olores” no son necesariamente un problema en sí mismos, pero nos indican que hay un problema cerca.

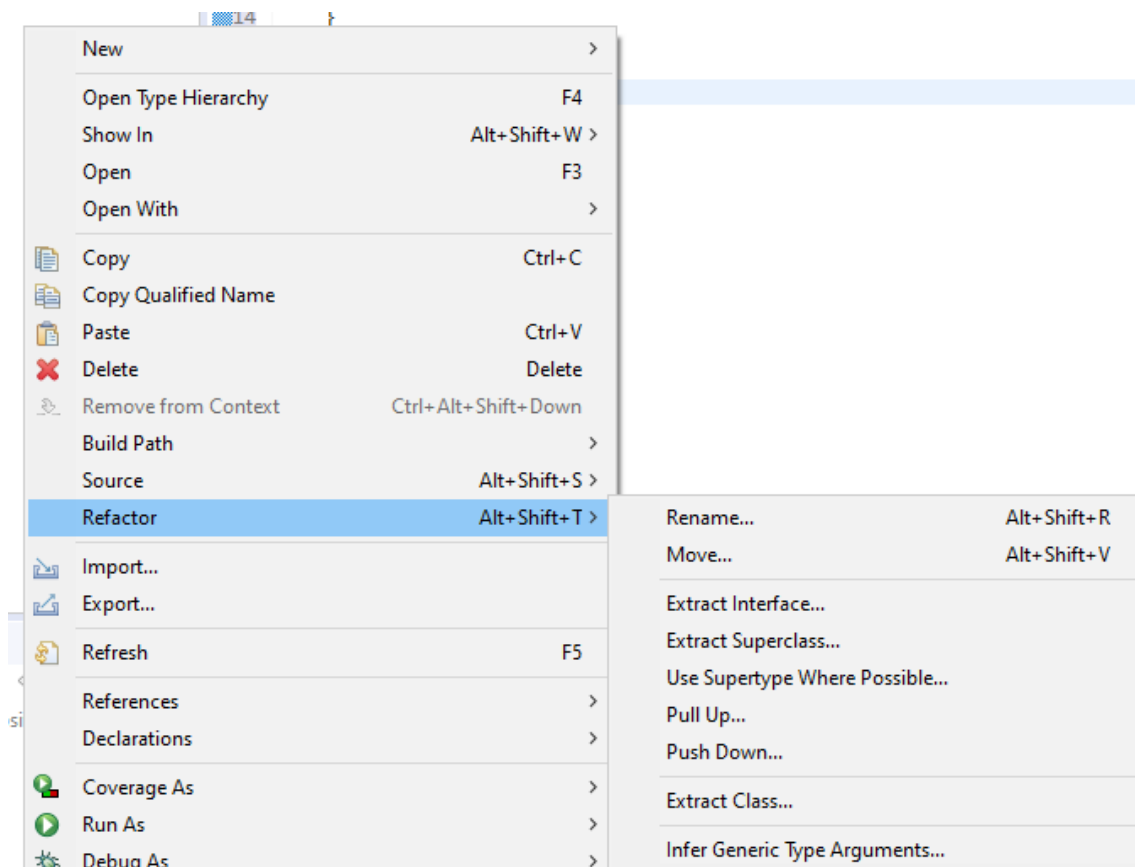
- **Método largo.** Los programas que viven más y mejor son aquellos con métodos cortos, que son más reutilizables y aportan mayor semántica.
- **Clase grande.** Clases que hacen demasiado y por lo general con una baja cohesión, siendo muy vulnerables al cambio.
- **Lista de parámetros larga.** Los métodos con muchos parámetros elevan el acoplamiento, son difíciles de comprender y cambian con frecuencia.
- **Obsesión primitiva.** Uso excesivo de tipos primitivos. Existen grupos de tipos primitivos que deberían modelarse como objetos, siempre que vayan a usarse para tareas con algo de lógica.
- **Clases de datos.** Clases que solo tienen atributos y métodos get y set. Las clases siempre deben disponer de algún comportamiento no trivial.
- **Estructuras de agrupación condicional.** Lo que comentamos en un case o switch con muchas clausulas, o muchos ifs anidados, tampoco es una buena idea.
- **Comentarios.** No son estrictamente “malos olores”, más bien “desodorantes”. Al encontrar un gran comentario, se debería reflexionar sobre por qué necesita ser tan explicado.
- **Atributo temporal.** Algunos objetos tienen atributos que se usan solo en ciertas circunstancias. Tal código es difícil de comprender, ya que lo esperado es que un objeto use todas sus variables.
- **Generalidad especulativa.** Jerarquías con clases sin utilidad actual, pero que se introducen por si en un futuro fuesen necesarias. El resultado son jerarquías difíciles de mantener y comprender, con clases que pudieran no ser nunca de utilidad.
- **Jerarquías paralelas.** Cada vez que se añade una subclase a una jerarquía hay que añadir otra nueva clase en otra jerarquía distinta.
- **Intermediario.** Clases cuyo único trabajo es la delegación y ser intermediarias.

- **Legado rechazado.** Subclases que usan solo un poco de lo que sus padres les dan. Si las clases hijas no necesitan lo que heredan, generalmente la herencia está mal aplicada.
- **Intimidad inadecuada.** Clases que tratan con la parte privada de otras. Se debe restringir el acceso al conocimiento interno de una clase.
- **Cadena de mensajes.** Un cliente pide algo a un objeto que a su vez lo pide a otro y éste a otro, etc.
- **Clase perezosa.** Una clase que no está haciendo nada o casi nada debería eliminarse.
- **Cambios en cadena.** Un cambio en una clase implica cambiar otras muchas. En estas circunstancias es muy difícil afrontar un proceso de cambio.
- **Envidia de características.** Un método que utiliza más cantidad de cosas de otro objeto que de sí mismo.
- **Duplicación de código.** Duplicar, o copiar y pegar, código no es una buena idea.
- **Grupos de datos.** Manojos de datos que se arrastran juntos deberían situarse en una clase.

1.4 Refactorización en Eclipse

Eclipse como IDE que es, posee un amplio abanico de posibilidades de refactorización donde se encuentran las vistas en esta unidad, además de otras que también pueden ser útiles en proyectos de complejidad avanzada.

Para acceder a las opciones de refactorización tendremos que usar el menú contextual pulsando el botón derecho en cualquier área de nuestro entorno de trabajo o mediante al acceso directo Alt+Shift+T. Ofreciendo en cada caso las opciones posibles.



En el siguiente [enlace](#) se puede ver una descripción completa de las opciones de refactorización que posee Eclipse.

2 Control de versiones

En la segunda unidad ya se hizo una primera toma de contacto con el desarrollo colaborativo, su funcionalidad y su relación directa con el control de versiones.

Como ya se comentó, no es el único uso que nos ofrece el control de versiones, ya que, aunque desde luego su funcionalidad parece destinada a un desarrollo colaborativo completo, en donde muchos programadores trabajan de manera simultánea en un proyecto, también se suele utilizar de manera muy habitual para llevar un control de versiones o revisiones de un determinado programa y poder tener un repositorio accesible con las diferentes versiones creadas, siendo utilizado tanto como copia de respaldo como para poder volver a una versión anterior o incluso porque por necesidades específicas necesitemos utilizar el código existente en una versión determinada. Por ejemplo, podríamos necesitar crear un parche que solucione un problema con una versión en concreto sin que el usuario final tuviese que actualizar todo el producto a la versión final.

2.1 Repositorios

Un repositorio es básicamente un servidor de archivos típico, con una gran diferencia: lo que hace a los repositorios en comparación con esos servidores de archivos es que recuerdan todos los cambios que alguna vez se hayan escrito en ellos, de este modo, cada vez que actualizamos el repositorio, éste recuerda cada cambio realizado en el archivo o estructura de directorios. Además, permite establecer información adicional por cada actualización, pudiendo tener por ejemplo un *changelog* de las versiones en el propio repositorio.

Cada herramienta de control de versiones tiene su propio repositorio, y, por desgracia, no son compatibles, es decir, no puedes obtener los datos del repositorio o actualizarlo si el repositorio y el control de versiones no coinciden.

2.1.1 GIT

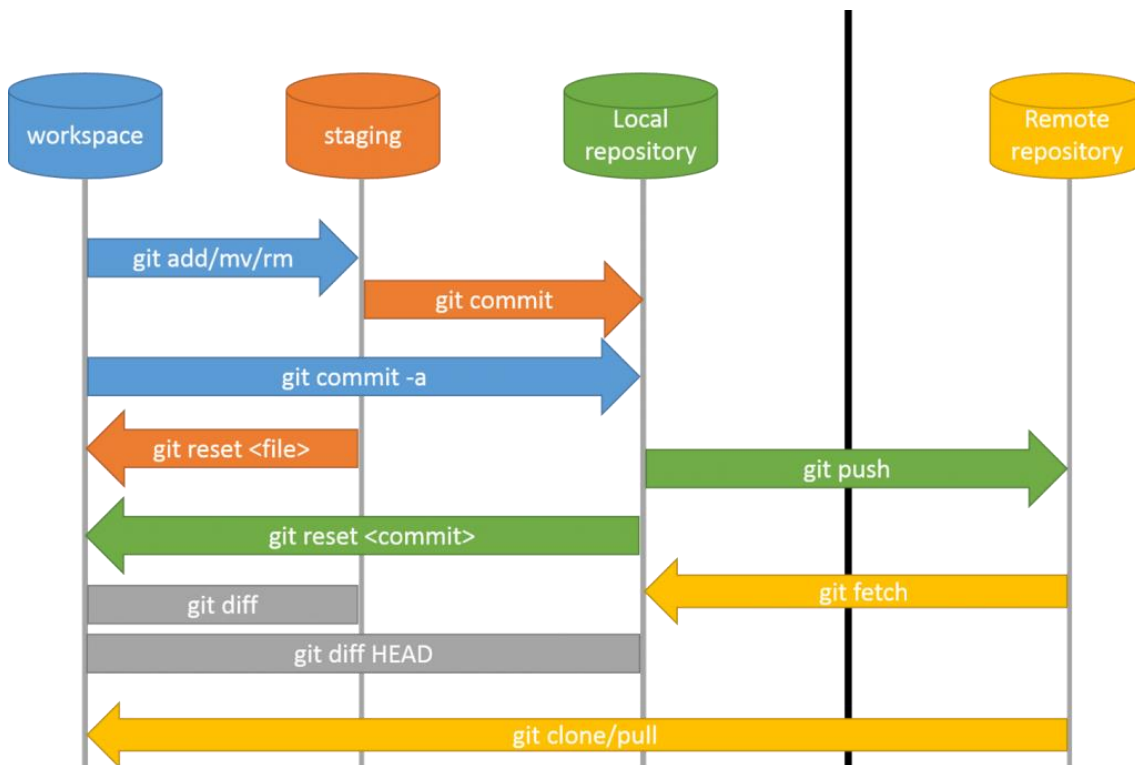
GIT es un software libre para la gestión de repositorios de código y control de versiones diseñado por Linus Torvalds en 2005. Parte de la idea de ofrecer un mantenimiento eficiente y confiable de versiones en aplicaciones con gran cantidad de ficheros de código. Y la principal diferencia con respecto a otros sistemas de la época, como SVN o CVS, es que se trata de un sistema distribuido.

La principal ventaja de ser un sistema distribuido radica en la independencia que proporciona a los equipos de desarrollo, ya que al poseer cada integrante el proyecto un servidor local puede seguir trabajando en caso de que el servidor centralizado falle.

Los principales elementos con los que trabaja GIT tanto a nivel local como remoto son los siguientes:

- **Directorio de trabajo (Working dir):** Contiene los archivos de trabajo de nuestro proyecto. En este directorio se encuentran las copias de trabajo y editables.
- **Index – Staging:** Zona intermedia donde se almacenan los ficheros que pasarán al repositorio local.
- **Repositorio local (Local repository):** Almacén que contendrá todos los ficheros del proyecto, con sus distintas versiones y estructura de ramas.
- **Repositorio remoto (Remote repository):** Almacén compartido que permitirá trabajar en un entorno distribuido.

Tomando como principales elementos los anteriores veremos el flujo de trabajo con GIT, haciendo una revisión de sus operaciones más importantes:



Si comenzamos trabajando en local, comenzaremos revisando la imagen desde la izquierda. Mediante el comando **git add** se enviarán los ficheros a la zona intermedia *staging*, a la espera de realizar un **git commit** que los envíe al repositorio local. Para colaborar con el resto de las personas de nuestro equipo tendremos que subir los ficheros al repositorio remoto mediante **git push**.

El camino inverso cuando necesitamos obtener ficheros de un repositorio remoto puede comenzar de dos maneras. Si es la primera vez usaremos **git clone**, para obtener una copia completa del repositorio remoto. Mientras que si ya hemos comenzado a trabajar usaremos **git fetch** para obtener los cambios con respecto a nuestro repositorio local, y posteriormente **git checkout** para actualizar nuestro directorio de trabajo. O hacer uso de **git pull**, para actualizar el repositorio local y el directorio de trabajo al mismo tiempo.

Además, existen otro tipo de operaciones que complementan a las anteriores:

- **Git reset:** La operación inversa a **git add**. Extrae ficheros de *staging* devolviéndolos al directorio de trabajo.
- **Git status:** Indica la situación actual de los ficheros que componen el directorio de trabajo.
- **Git log:** Muestra la historia de commits del proyecto de manera visual.
- **Git diff:** Muestra las diferencias introducidas entre dos objetos.

2.1.2 GitHub

Para poder hacer uso del control de versiones, antes es necesario un repositorio con el que sincronizar nuestra copia de trabajo. En nuestro caso se hará uso de la plataforma GitHub (<https://github.com/>)



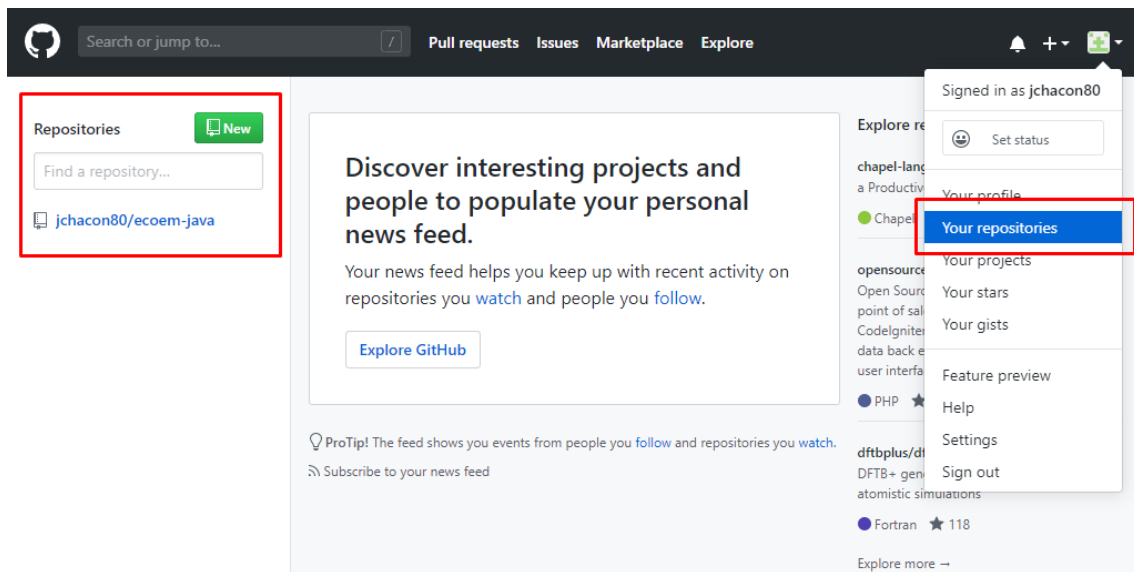
GitHub es una plataforma de desarrollo colaborativo donde se pueden crear repositorios de código para alojar proyectos utilizando el sistema de control de versiones GIT.

El primer paso será crear un usuario de GitHub para poder hacer uso de su plataforma. Habrá que acceder a su web (<https://github.com>) y completar el formulario de registro. Dentro de los planes que tienen usaremos la opción **Free**.

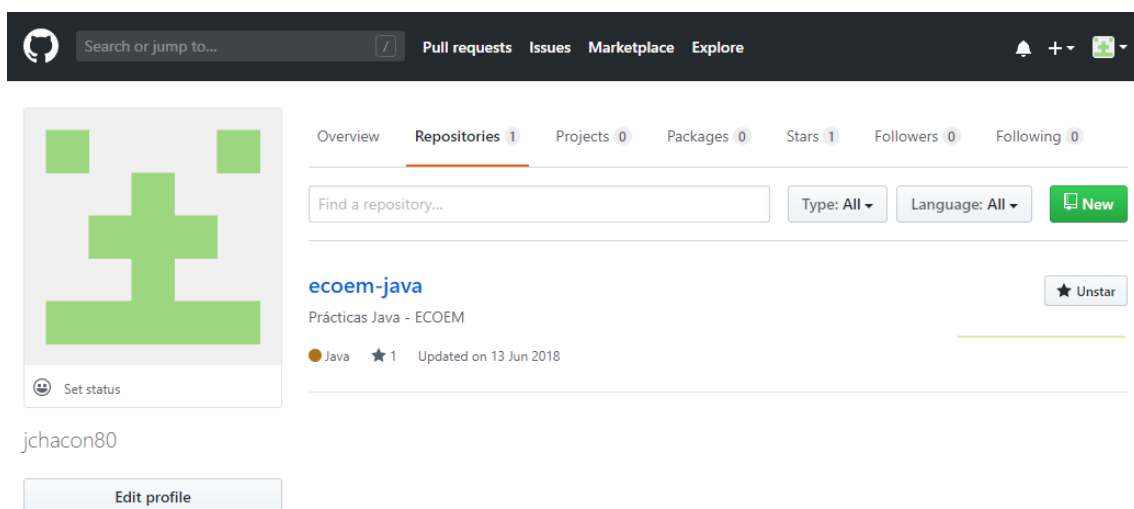
A screenshot of the GitHub website's sign-up page. The page has a dark background with the text "Built for developers" in large white font. Below this, it says "GitHub is a development platform inspired by the way you work. From open source to business, you can host and review code, manage projects, and build software alongside 40 million developers." On the right side, there is a white sign-up form with fields for "Username" (containing "jchacon80@gmail.com"), "Email", and "Password" (masked with dots). Below the password field, there is a note: "Make sure it's at least 15 characters OR at least 8 characters including a number and a lowercase letter. Learn more." At the bottom of the form is a green button that says "Sign up for GitHub". Below the button, there is a small disclaimer: "By clicking 'Sign up for GitHub', you agree to our Terms of Service and Privacy Statement. We'll occasionally send you account related emails." The top of the page features a navigation bar with links like "Why GitHub?", "Enterprise", "Explore", "Marketplace", and "Pricing", along with a search bar and "Sign in" / "Sign up" buttons.

Una vez accedamos veremos la página de bienvenida en la que podremos acceder a nuestro perfil (esquina superior derecha), tener una vista de resumen de nuestros repositorios (zona izquierda), y muchas más opciones.

Ahora mismo nos centraremos en la creación de nuestro primer repositorio, por lo que pulsaremos en la opción **New** en la parte izquierda, o accederemos a la opción **Your repositories** desde nuestro perfil.



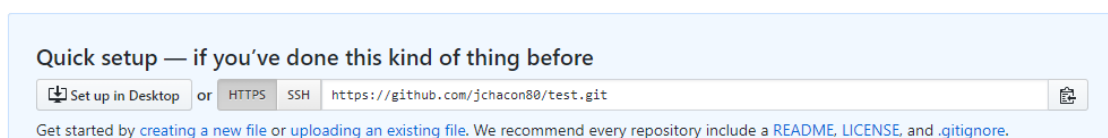
Si accedemos a la página de nuestro repositorios podremos hacer una gestión completa de todos los repositorios que hayamos creado o tengamos acceso. Además de poder crear nuevos repositorios.



A través del botón New podremos introducir la información nuestro nuevo repositorio.

- Owner (propietario): por defecto será el usuario con el que estamos logados.
- Repository name: será el nombre que le demos a nuestro repositorio.
- Description: breve descripción de nuestro repositorio.
- Visibilidad: podrá ser público o privado, en el primer caso puede ser consultado por todo el mundo, y en el segundo no. En ambos casos se debe definir quién puede hacer commit.

Una vez creado es importante guardar la ruta HTTPS de nuestro repositorio:



Si ahora volvemos a nuestra página principal podremos ver como tenemos un nuevo repositorio y pulsando en él podremos gestionarlo sin problemas.

2.2 Herramientas de control de versiones

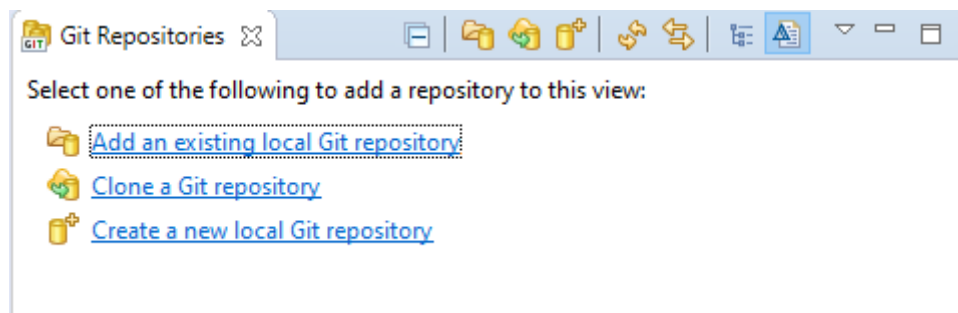
Para poder asociar nuestro código a un repositorio del control de versiones, es necesaria una herramienta que se encargue de ir observando los cambios realizados para luego actualizarlo de manera que el repositorio los entienda y pueda llevar el control de versiones de manera apropiada.

2.2.1 EGIT – Git integrations for Eclipse

En la instalación por defecto de Eclipse se incluye un plugin para trabajar con el control de versiones GIT, su nombre es EGit (<https://www.eclipse.org/egit/>).



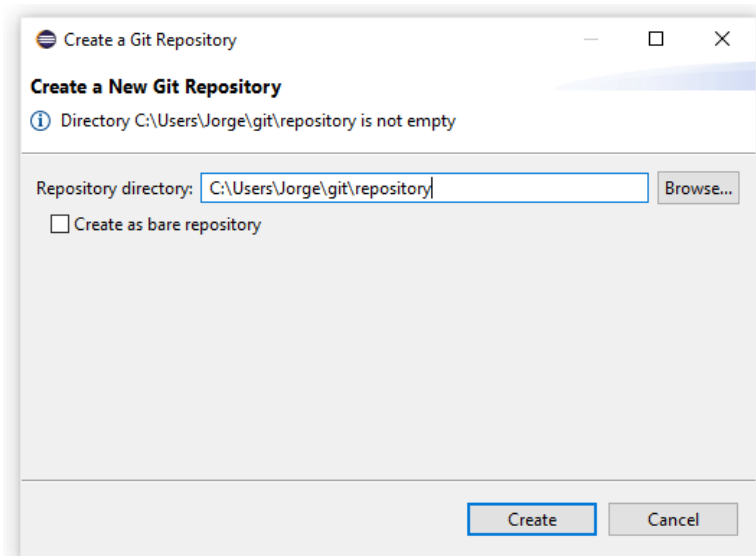
El trabajo inicial con EGit dependerá de lo comentado en el punto anterior sobre el flujo de GIT, si estamos creando un repositorio local que sincronizaremos con un repositorio remoto, o si vamos a usar un repositorio remoto para añadir código. En cualquier caso, se tendrá que activar la vista **Git repositories**.



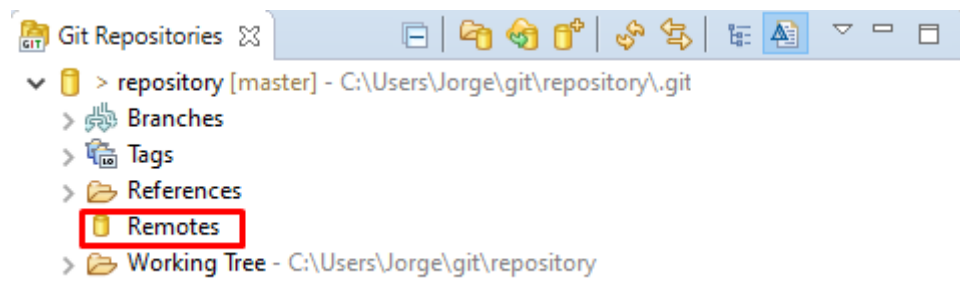
2.2.1.1 Creación de un repositorio local

A continuación, se verá el primer escenario posible, vamos a crear un repositorio local para conectar posteriormente con un repositorio remoto y poder compartir nuestro código con otras personas o hacer una copia de seguridad en un servidor remoto.

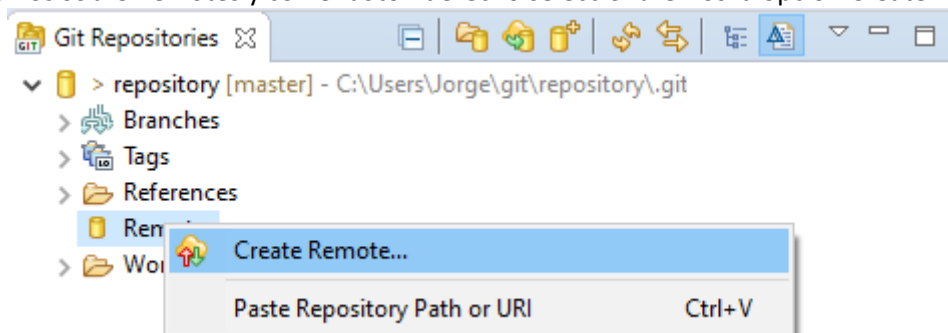
Se pulsará la opción **“Create a new local Git repository”** y aparecerá una ventana donde tendremos que indicar la carpeta donde se creará (IMPORTANTE: esta carpeta deberá ser o será un workspace de Eclipse);:



A partir de ahora en la vista **Git repositories** veremos la conexión que acabamos de crear, pero por ahora con una particularidad y es que en el árbol de objetos no tendremos nada en **Remotes**.

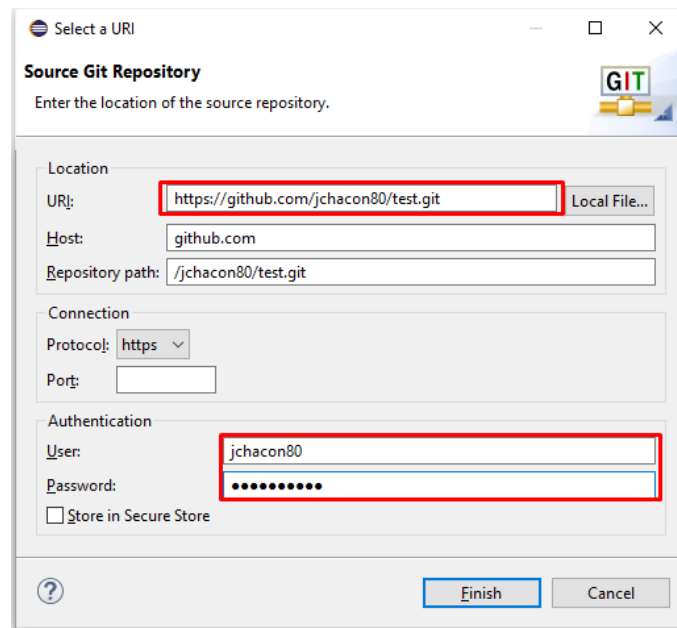


Por lo que el siguiente paso consistirá en conectar con nuestro repositorio remoto en GitHub. Pulsaremos sobre Remotes y con el botón derecho seleccionaremos la opción Create Remote...



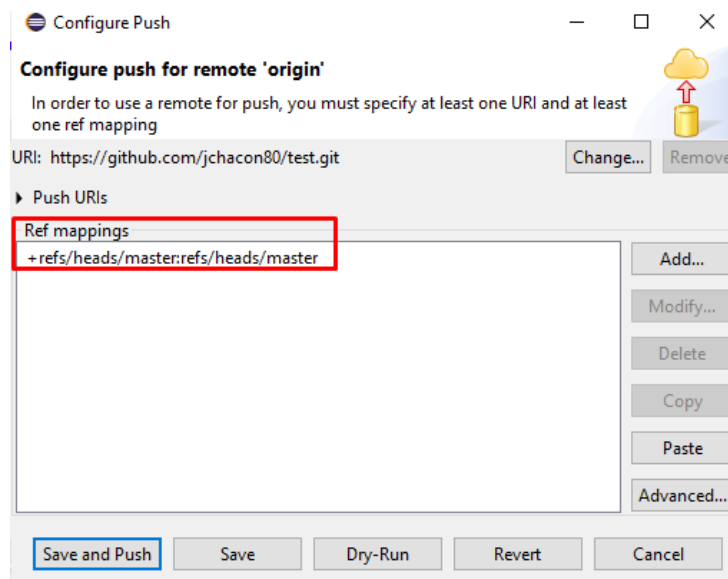
En la siguiente ventana dejaremos las opciones por defecto, y posteriormente tendremos que añadir la URI de nuestro repositorio. Donde tendremos que indicar la URI copiada de la web de GitHub, el usuario y clave de acceso.

Una vez hecho esta configuración tendremos nuestro repositorio local conectado a un repositorio remoto y por lo tanto podremos realizar acciones de sincronización.

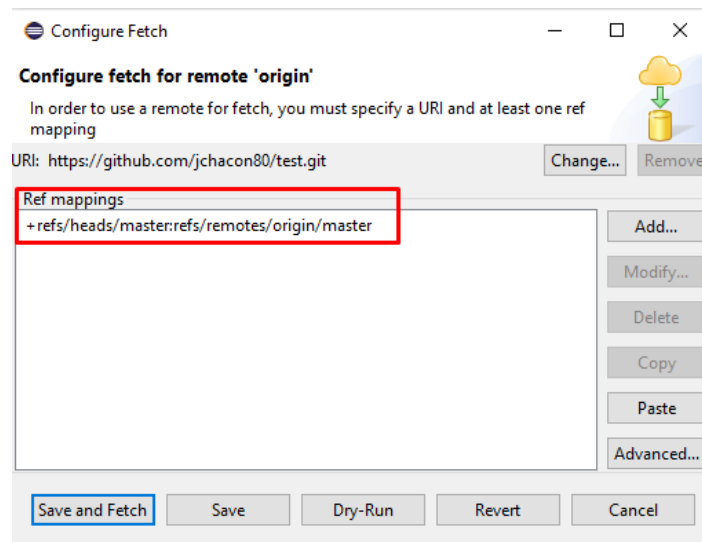


Un punto muy importante será la configuración de las operaciones **Push** y **Fetch**, donde se tendrán que indicar los puntos de sincronización. En ambos casos se realizará directamente desde la vista **Git Repositories** pulsando con el botón derecho sobre el repositorio remoto.

A continuación, la configuración de la operación **Push**, es decir, la operación que permitirá subir cambios al repositorio remoto.



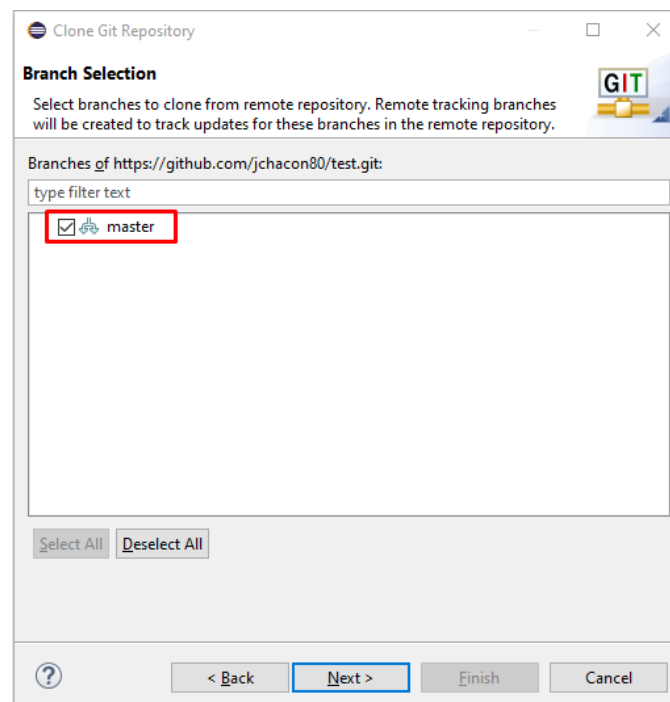
En la siguiente imagen se puede ver como quedará la operación **Fetch**, es decir, la que permite obtener información del repositorio remoto.



2.2.1.2 Clonar un repositorio remoto

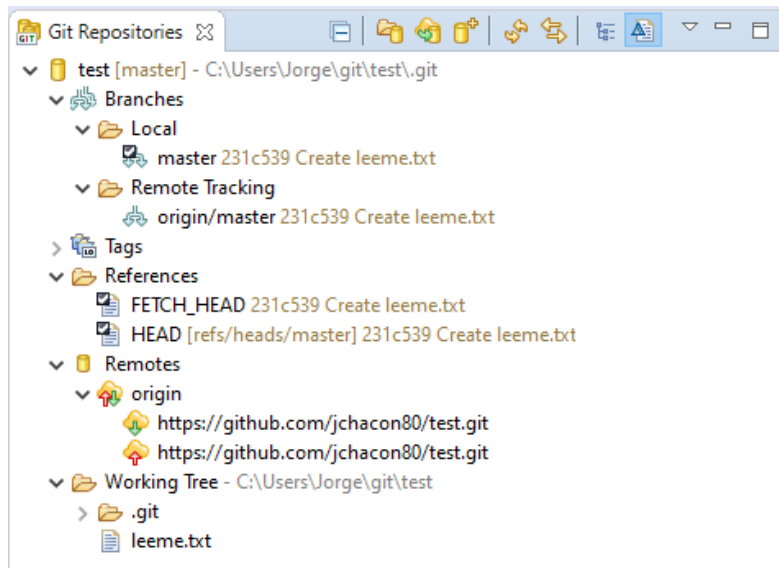
En esta ocasión seleccionaremos la opción “Clone a Git repository”, para mostrar una ventana similar a la vista en el paso final del punto anterior, es decir, la configuración del repositorio remoto.

Una vez introducidos los datos, pulsaremos Next y aparecerá una ventana para seleccionar la rama (Branch) que deseamos usar. Por defecto Master.



En el paso final se podrá elegir la carpeta de destino, el nombre del repositorio remoto e importar los proyectos directamente a Eclipse.

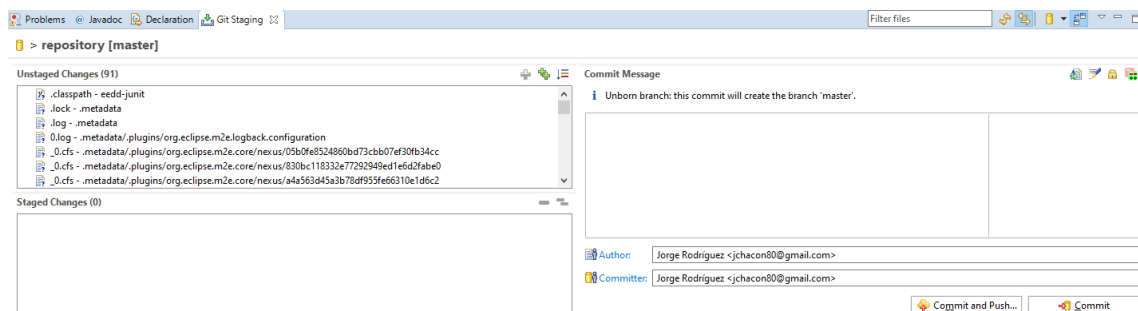
Con esta opción se crea automáticamente el repositorio local como copia del repositorio remoto clonado.



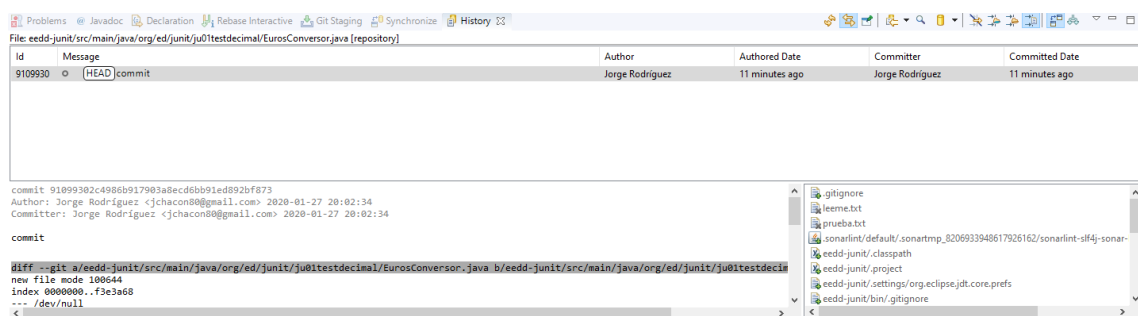
2.2.1.3 Operaciones de sincronización

El plugin EGit posee una perspectiva propia y varias vistas que son de utilidad para el trabajo diario. Por ejemplo, la vista **Git Staging** permite ver en todo momento ficheros han sido modificados y no han sido sincronizados en nuestro repositorio local.

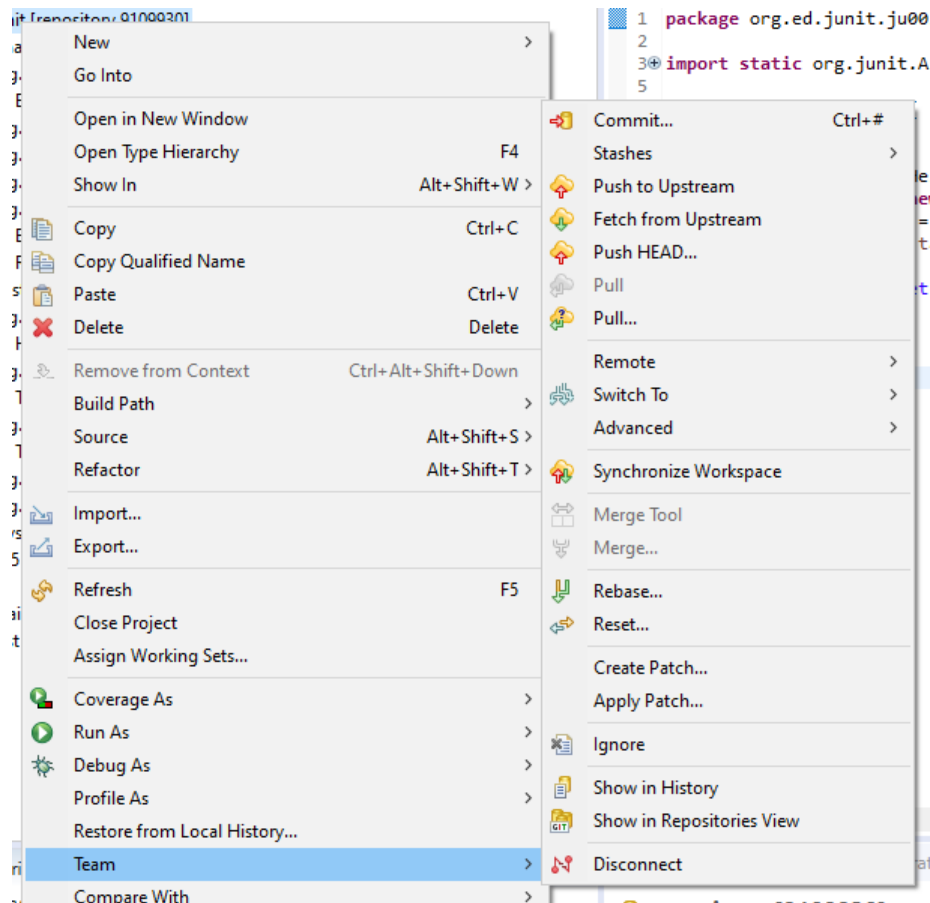
Además, podremos pasar masivamente ficheros a **Staging** y hacer **commit** o **commit y push** de manera simultánea.



O la vista History, que nos permitirá ver el histórico de versiones de un fichero y seleccionar una versión u otra, e incluso realizar comparaciones para ver los cambios realizados.

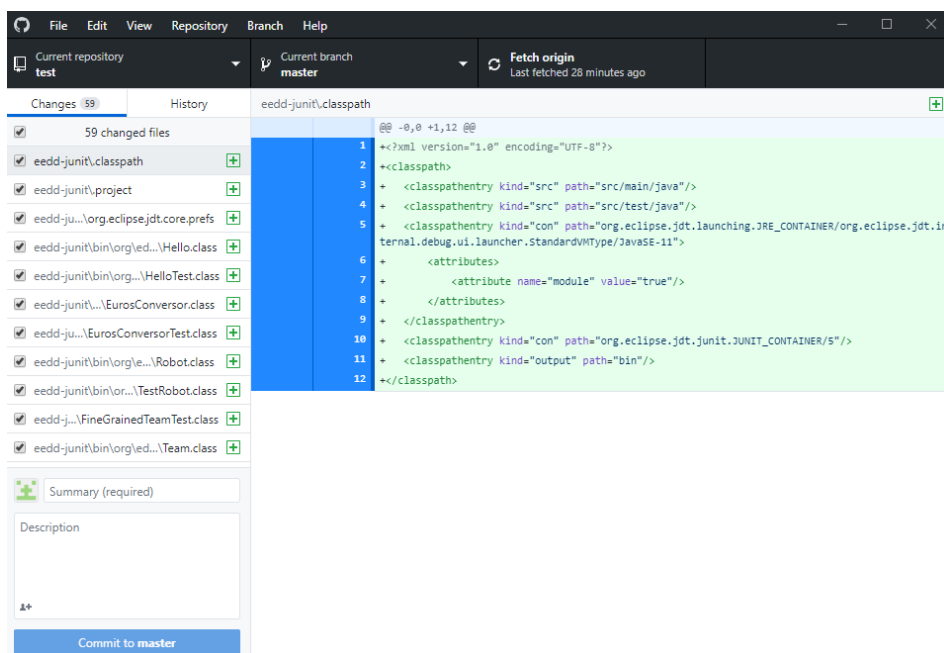


Otra opción será trabajar directamente desde el árbol de ficheros a través de la opción **Team** del menú, que mostrará todas las opciones disponibles de trabajo con GIT.



2.2.2 GitHub Desktop

Una alternativa a EGit si no se trabajara con Eclipse, o si se quisiera trabajar de manera nativa desde el sistema operativo será GitHubDesktop (<https://desktop.github.com/>). Se trata de una aplicación de escritorio muy potente y fácil de manejar, donde la configuración consiste solamente en introducir nuestras credenciales de GitHub para comenzar a trabajar con los repositorios GIT.



3 Documentación

En todo proyecto de software, es muy importante la documentación, no solo para el usuario final, sino para los propios desarrolladores, tanto para uno mismo como para otros desarrolladores que tengan que trabajar en el presente o futuro con el proyecto.

Ya hemos comentado cómo ayuda la refactorización en el entendimiento del código, pero siempre nos será más comprensible una buena documentación, no solo porque está escrito en nuestro propio lenguaje, sino porque puede contener notas aclaratorias que el código no nos podría decir directamente, como por ejemplo el patrón utilizado o documentación y referencia sobre las librerías ajenas utilizadas en el proyecto.

3.1 Uso de comentarios

El primer paso en una buena documentación es el uso apropiado de comentarios. En este mismo capítulo hemos hablado sobre los comentarios, señalando que si un código necesita ser comentado es porque no es suficientemente descriptivo por sí solo y necesita ser refactorizado. Aunque esto sea efectivamente así, sigue siendo una buena ayuda al entendimiento del código y sobre todo muy utilizado a la hora de crear notas dentro del código que no necesariamente tienen que ser una explicación del funcionamiento del código, además se utiliza de manera recurrente con el fin de omitir algunas instrucciones para que no se ejecuten, muy importante en los procesos de depuración.

Cada lenguaje suele tener sus propios modos de establecer comentarios. En el caso de Java tenemos dos modos de comentarios: comentarios de línea o comentarios de bloque.

```
//Esto es un comentario de línea
int x=2;

/*
 * Esto es un comentario de bloque
 * Solo es necesario establecer
 * el inicio y el fin de bloque
 */
```

Como se puede ver en el ejemplo, los comentarios de bloque nos ahorran tener que comentar las líneas una a una cuando queremos comentar toda una porción de código.

Los comentarios se utilizan para clarificar algún algoritmo complicado, además de las opciones que hemos comentado previamente. Además, también se suelen utilizar en las pruebas de caja blanca, explicando entre otra cosas el porqué de los datos utilizados y de los resultados esperados.

3.2 Herramientas – JavaDoc

JavaDoc es una utilidad de Oracle, e incluida en las distribuciones de Java, para la generación de documentación. La herramienta analiza las declaraciones y los comentarios incluidos en los ficheros fuente .java y produce un conjunto de páginas HTML que describen clases, interfaces, constructores, métodos y campos.

3.2.1 Comentando el código con JavaDoc

Para hacer uso de la API de JavaDoc debemos usar etiquetas HTML o ciertas palabras reservadas precedidas del carácter “@”.

Estas etiquetas deben escribirse al principio de cada clase, miembro o método, dependiendo de qué objeto se desee describir, mediante un comentario iniciado con “/**” y acabado con “*/”.

Las principales palabras reservadas se pueden ver en la siguiente tabla, y para más información se puede consultar la documentación oficial de Oracle (<https://docs.oracle.com/javase/8/docs/technotes/tools/windows/javadoc.html>).

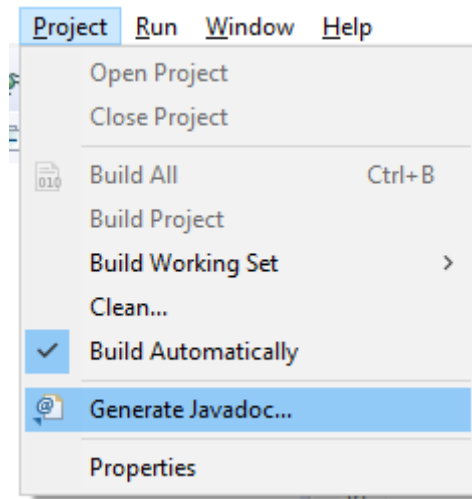
Tag	Descripción	Uso	Versión
@author	Nombre del desarrollador.	nombre_autor	1.0
@version	Versión del método o clase.	versión	1.0
@param	Definición de un parámetro de un método, es requerido para todos los parámetros del método.	nombre_parametro descripción	1.0
@return	Informa de lo que devuelve el método, no se puede usar en constructores o métodos "void".	descripción	1.0
@throws	Excepción lanzada por el método, posee un sinónimo de nombre @exception	nombre_clase descripción	1.2
@see	Asocia con otro método o clase.	referencia (#método(); clase#método(); paquete.clase; paquete.clase#método()).	1.0
@since	Especifica la versión del producto	indicativo numerico	1.2
@serial	Describe el significado del campo y sus valores aceptables. Otras formas válidas son @serialField y @serialData	campo_descripcion	1.2
@deprecated	Indica que el método o clase es antigua y que no se recomienda su uso porque posiblemente desaparecerá en versiones posteriores.	descripción	1.0

Y a continuación se puede observar su uso en una clase java.

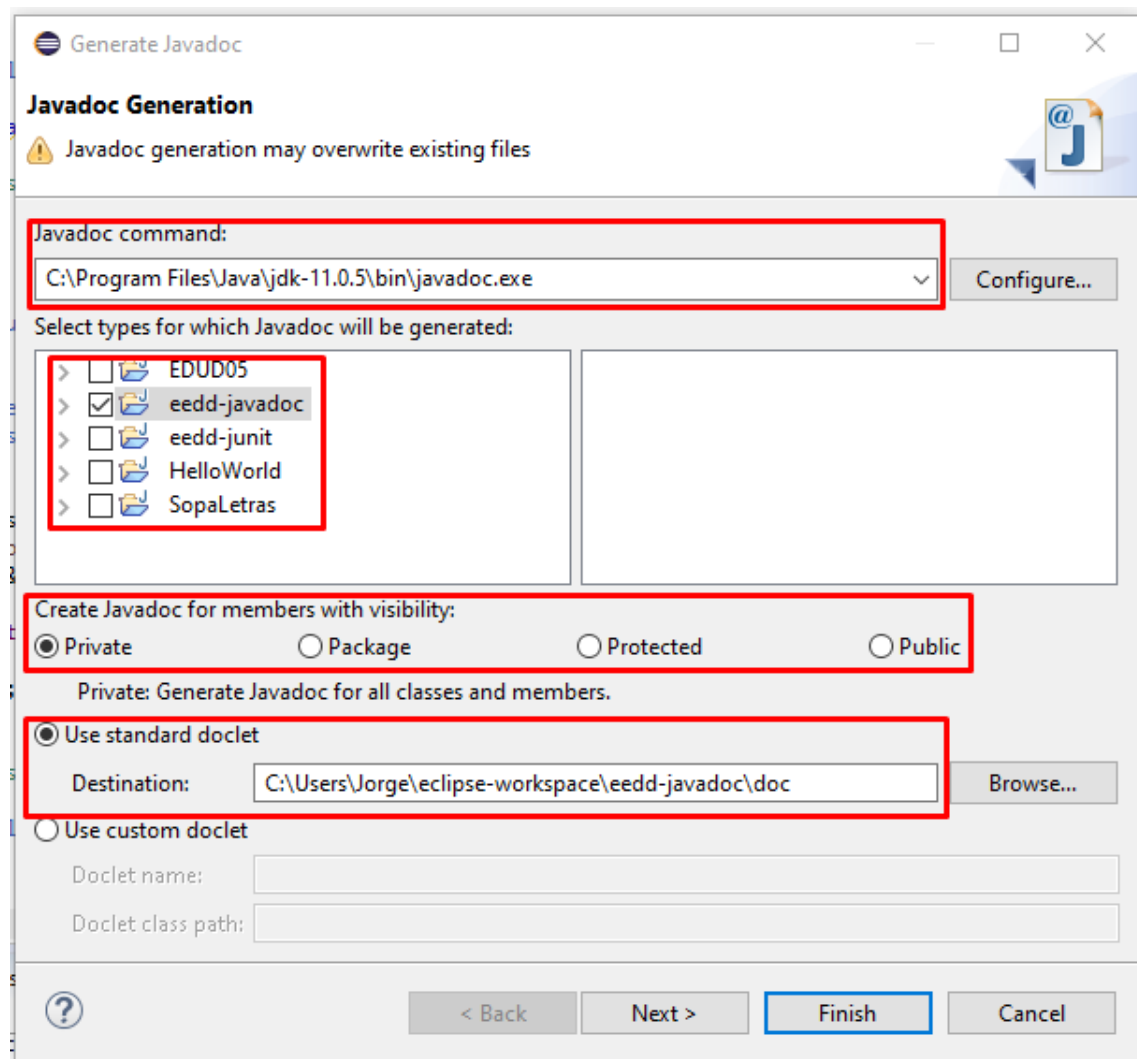
```
/**
 * Suma un plus al salario del empleado si el empleado tiene mas de 40 años
 *
 * @param sueldoPlus valor del plus que se suma al salario
 * @return
 * <ul>
 * <li>true: se suma el plus al sueldo</li>
 * <li>false: no se suma el plus al sueldo</li>
 * </ul>
 */
public boolean plus(double sueldoPlus) {
    boolean aumento = false;
    if (edad > 40 && compruebaNombre()) {
        salario += sueldoPlus;
        aumento = true;
    }
    return aumento;
}
```

3.2.2 Generando documentación con Javadoc

Una vez se ha documentado el código, la generación de la documentación HTML asociada es un proceso sencillo haciendo uso de Eclipse. Se accederá a la opción Project → Generate Javadoc.



En la siguiente ventana tendremos que seleccionar la ruta donde se encuentra javadoc (por defecto, la ruta de nuestro JDK), el proyecto, la visibilidad de los elementos que queremos incluir en la documentación y la ruta donde almacenaremos la documentación.



En la dos siguientes ventanas dejaremos las opciones que aparecen por defecto, y pulsaremos “Finish”. Viendo en la consola de Eclipse el proceso de generación.

Una vez finalizado podremos acceder a la ruta donde hemos generado la documentación y abriremos el fichero “Index.html” mostrándose la página inicial de nuestra documentación donde veremos la información organizada en paquetes.

PACKAGE

CLASS

USE

TREE

DEPRECATED

INDEX

HELP

ALL CLASSES

SEARCH:

Package main

Class Summary

Class	Description
Empleado	Clase Empleado Contiene informacion de cada empleado

PACKAGE

CLASS

USE

TREE

DEPRECATED

INDEX

HELP

ALL CLASSES