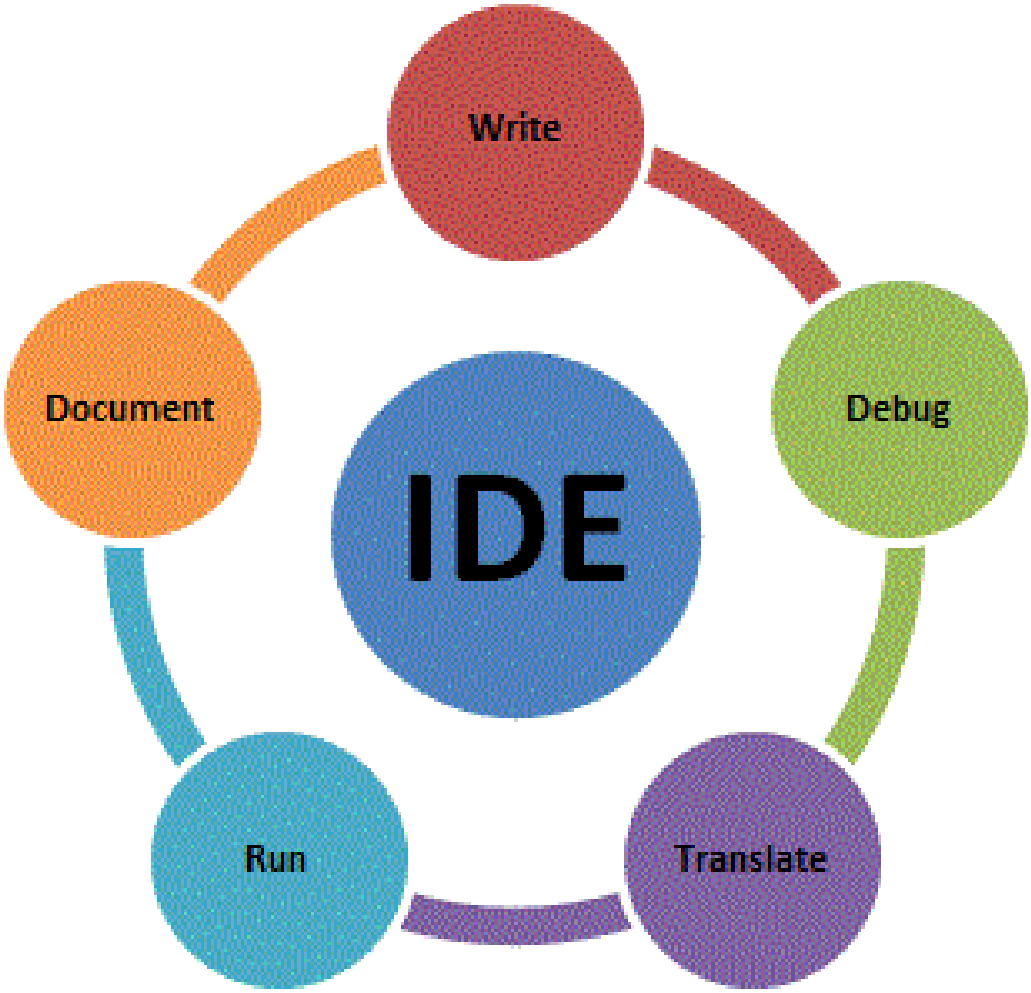
GOBIERNO DE ESPAÑA
MINISTERIO DE EDUCACIÓN Y FORMACIÓN PROFESIONAL
Unión Europea
Fondo Social Europeo
"El FSE invierte en tu futuro"
Junta de Andalucía
Consejería de Educación y Deporte



UNIDAD 4. OPTIMIZACIÓN Y CONTROL DE VERSIONES

1 INTRODUCCIÓN

Una vez que tenemos nuestro software generado y comprobado mediante pruebas, puede ser buena idea optimizarlo para que haga lo mismo, pero más rápido o consumiendo menos recursos. A este tipo de operaciones se le denomina **Optimización** del código y veremos varias técnicas para conseguirlo en este tema.

Oro aspecto igual de importante que el anterior es la Documentación de todo el proyecto, desde sus primeras fases de análisis hasta las últimas pruebas y también estas nuevas de optimización que vamos a ver a continuación.

2 REFACTORIZACIÓN

La refactorización (del inglés refactoring) se puede definir como la "técnica que consiste en realizar una transformación al software preservando su comportamiento, modificando su estructura interna para mejorarlo" (William F. Opdyke, 1992).

La refactorización se realiza a menudo como parte del proceso de desarrollo del software: los desarrolladores alternan la inserción de nuevas funcionalidades y casos de prueba con la refactorización del código para mejorar su consistencia interna y su claridad. Los test aseguran que la refactorización no cambia el comportamiento del código.

La refactorización es la parte del mantenimiento del código que no arregla errores ni añade funcionalidad. El objetivo, por el contrario, es mejorar la facilidad de comprensión del código o cambiar su estructura y diseño y eliminar código muerto, para facilitar el mantenimiento en el futuro. Añadir nuevo comportamiento a un programa puede ser difícil con la estructura dada del programa, así que un desarrollador puede refactorizarlo primero para facilitar esta tarea y luego añadir el nuevo comportamiento.

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".

2.1 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 ver y entender, que es parte del objetivo de la refactorización.

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

Hoy en 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, etc.

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 siguiente 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 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 puede corregir la indentación del código con un simple atajo de teclado "Ctrl+I", en el caso de netbeans la combinación de teclas será "Alt+Shift+F".

```java
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;
    }

}
```

```java
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;
}

}
```

2.2 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 un 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 Designo f Existing Code”.

2.2.1 Extraer el método

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

```java
public void imprimirTodoNoRefactor() {
    imprimirBanner();

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

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

}

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

2.2.2 Encapsular atributo

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

```java
public String atributoNoEncapsulado;
```

```java
private String atributoEncapsulado;

public String getAtributoEncapsulado() {
    return atributoEncapsulado;
}

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

2.2.3 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.

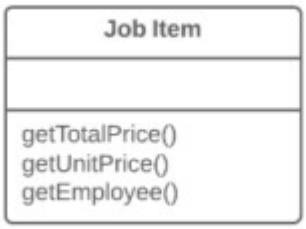
```
double energiaPotencial(double masa, double altura) {
    return masa * altura * 9.81;
}
```

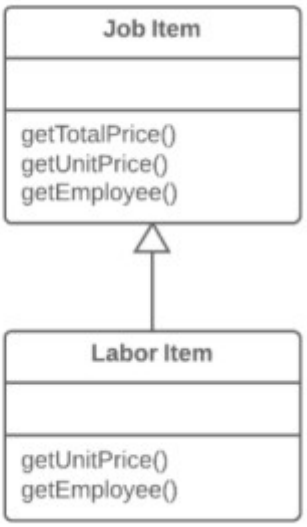
```
private static final double _CONSTANTE_GRAVITACIONAL = 9.81;

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

2.2.4 Extraer subclase

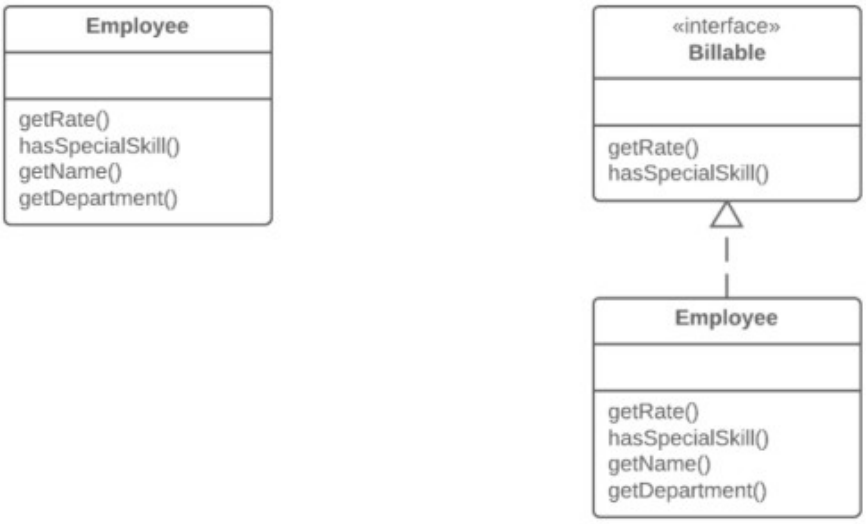
- Una clase tiene propiedades que solo son usadas en determinadas instancias
- Creamos una subclase para dicho conjunto de propiedades.





2.2.5 Extraer interfaz

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



2.3 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 con muchos if 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 fueran necesarias. Como resultado tenemos 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 a 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.

3 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 en aquel momento, 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.

3.1 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 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.

3.1.1 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 otro sistema de la época, como SVN o CVS, es que se trata de un sistema distribuido.

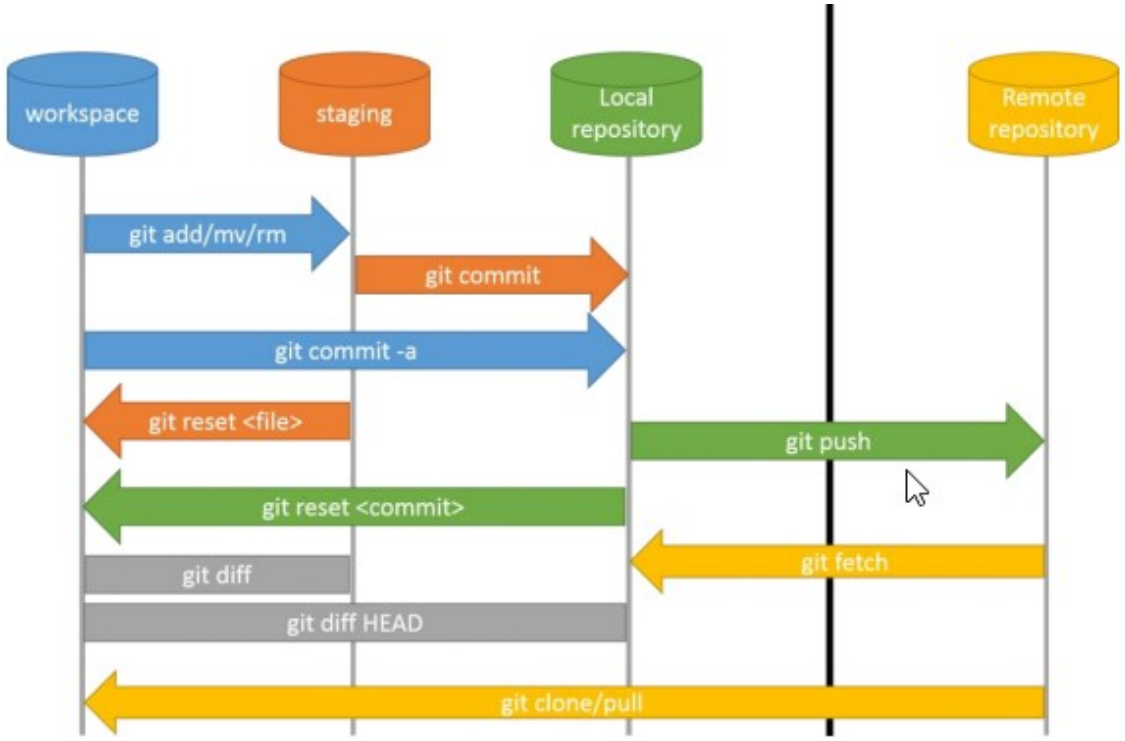
La principal ventaja de ser un sistema distribuido radica en la independencia que proporciona a los equipos de desarrollo, ya que al poseer cada integrante el proyecto de un servidor local puede seguir trabajando en caso de que el servidor centralizado falle.

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

- **Directorio de trabajo:** 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:** 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, comenzamos 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 acutalizar nuestro directorio de trabajo. O hacer uso de **git pull**, para acutalizar 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. Extraer 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.

3.1.2 GitHub

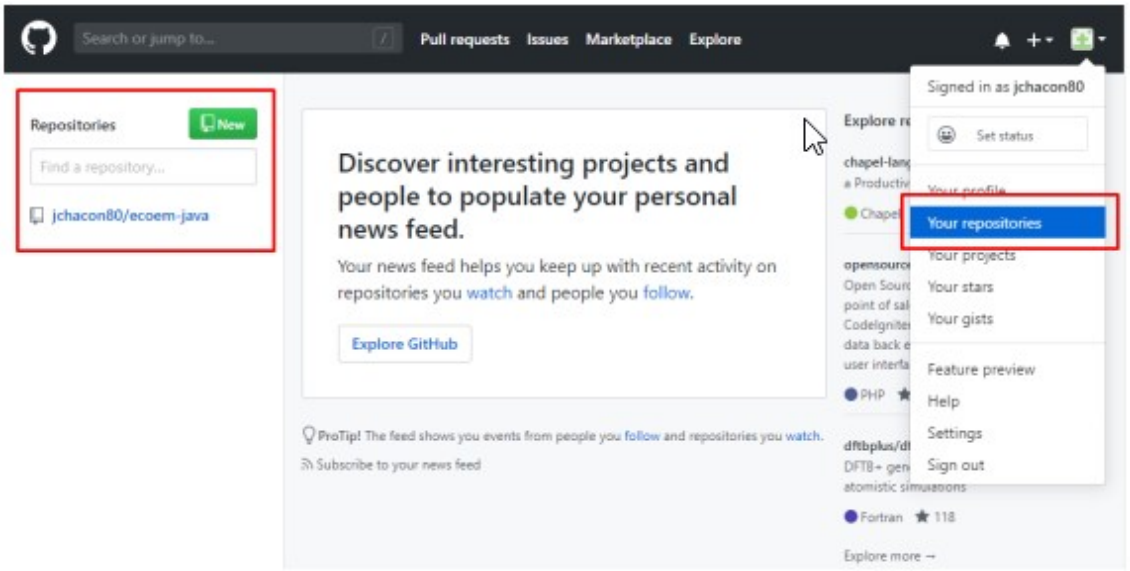
Para poder hacer uso del control de versiones, antes es necesario un repositorio con el que sincronizar nuestra copia de trabajo. En nuestro caso se hará uso de la plataforma GitHub (https://github.com/)

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

El primer paso será crear un usuario de GitHub para poder hacer uso de su plataforma. Habrá que acceder a su web y completar el formulario de registro. Dentro de los planes que tiene usaremos la opción FREE.

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

Ahora mismo nos centramos en la creación de nuestro primer repositorio, por lo que pulsaremos la opción de NEW en la parte izquierda, o accederemos a la opción YOUR REPOSITORIES desde nuestro perfil.



Si accedemos a la página de nuestros 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 de nuestro nuevo repositorio.

- Owner (propietario): por defecto será el usuario con el que estemos logado.
- 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 que tenemos un nuevo repositorio y pulsando sobre él podremos gestionarlo sin problemas.

3.2 TERMINOLOGÍA

La terminología empleada puede variar de sistema a sistema, pero a continuación se describen algunos de los términos de uso común.

- **REPOSITORIO:** El repositorio es el lugar en el que se almacenan los datos actualizados e históricos de cambios, a menudo en un servidor. A veces se le denomina depósito o depot. Puede ser un sistema de archivos en un disco duro, un banco de datos, etc.
- **MÓDULO:** Conjunto de directorios y/o archivos dentro del repositorio que pertenecen a un proyecto común.
- **REVISIÓN:** Una revisión es una versión determinada de la información que se gestiona. Hay sistemas que identifican las revisiones con un contador (ej. Subversión). Hay otros sistemas que identifican las revisiones mediante un código de detección de modificaciones. A la última versión se le suele identificar de forma especial con el nombre de HEAD. Para poner especial a una revisión en concreto se usan los rótulos o tags.
- **ROTULAR (TAG):** Darle a alguna versión de cada uno de los ficheros del módulo en desarrollo en un momento preciso un nombre común ("etiqueta" o "rótulo") para asegurarse de reencontrar ese estado de desarrollo posteriormente bajo ese nombre. En la práctica se rotula a todos los archivos en un momento determinado. Para eso el módulo se congela durante el rotulado para imponer una versión coherente. Pero bajo ciertas circunstancias puede ser necesario utilizar versiones de algunos ficheros que no coinciden temporalmente con las de los otros ficheros del módulo.

  Los tags permiten identificar de forma fácil revisiones importantes en el proyecto. Por ejemplo, se suelen usar tags para identificar el contenido de las versiones publicadas en el proyecto.

  En algunos sistemas se considera un tag como una rama en la que los ficheros no evolucionan, están congelados.
- **LÍNEA BASE (baseline):** Una revisión aprobada de un documento o fichero fuente, a partir de la cual se pueden realizar cambios subsiguientes.
- **ABRIR RAMA (BRANCH) O RAMIFICAR:** Un módulo puede ser branched o bifurcado en un instante de tiempo de forma que, desde momento en adelante se tienen dos copias (ramas) que evolucionan de

forma independiente siguiendo su propia línea de desarrollo. El módulo tiene entonces 2 (o más) “ramas”. La ventaja es que se puede hacer un “merge” de las modificaciones de ambas ramas, posibilitando la creación de “ramas de prueba” que contengan código para evaluación, si se decide que las modificaciones realizadas en la “rama de prueba” sean preservadas, se hace un “merge” con la rama principal. Son motivos habituales para la creación de ramas la creación de nuevas funcionalidades o la corrección de errores.

- **DESPLEGAR (CHECKOUT):** Un despliegue crea una copia de trabajo local desde el repositorio. Se puede especificar una revisión concreta, y por defecto se suele obtener la última.
- **PUBLICAR O ENVIAR (COMMIT, CHECK IN, INSTALL, SUBMIT):** Un commit sucede cuando una copia de los cambios hechos a una copia local es escrita o integrada en un repositorio.
- **CONFLICTO:** Un conflicto ocurre en las siguientes circunstancias:
    - Los usuarios X e Y despliegan versiones del archivo A en que las líneas n1 hasta n2 son comunes.
    - El usuario X envía cambios entre las líneas n1 y n2 al archivo A
    - El usuario Y no actualiza el archivo A tras el envío del usuario X.
    - El usuario Y realiza cambios entre las líneas n1 y n2.
    - El usuario Y intenta posteriormente enviar esos cambios al archivo A.
    - El sistema es incapaz de fusionar los cambios. El usuario Y debe resolver el conflicto combinando los cambios, o eligiendo uno de ellos para descartar el otro.
- **RESOLVER:** El acto de la intervención del usuario para atender un conflicto entre diferentes cambios al mismo documento.
- **CAMBIO (CHANGE, DIFF, DELTA):** Un cambio representa una modificación específica a un documento bajo control de versiones. La granularidad de la modificación considerada un cambio varía entre diferentes sistemas del control de versiones.

- **EXPORTACIÓN(EXPORT):** Una exportación es similar a un checkout, salvo porque crea un árbol de directorios limpio sin los metadatos de control de versiones presentes en la copia de trabajo. Se utiliza a menudo de forma previa a la publicación de los contenidos.
- **IMPORTACIÓN (IMPORT):** Una importación es la acción de copiar un árbol de directorios local (que o es en ese momento una copia de trabajo) en el repositorio por primera vez.
- **INTEGRACIÓN O FUSIÓN (MERGE):** Una integración o fusión une dos conjuntos de cambios sobre un fichero o un conjunto de ficheros en una revisión unificada de dicho fichero o ficheros.

  Esto puede suceder cuando un usuario, trabajando en esos ficheros, actualiza su copia en local con los cambios realizados, y añadidos al repositorio, por otros usuarios.

4 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 en el 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.

4.1 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 de nuestro 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 tiene sus propios modos de comentarios. En el caso de java tenemos dos modos de comentarios: en línea y en 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 lo que queremos comentar es toda una parte del código.

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

4.2 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 las clases, interfaces, constructores, métodos y campos.

Para hacer uso de la API de JavaDoc debemos usar etiquetas HTML o ciertas palabras reservadas precedidas por el carácter @.

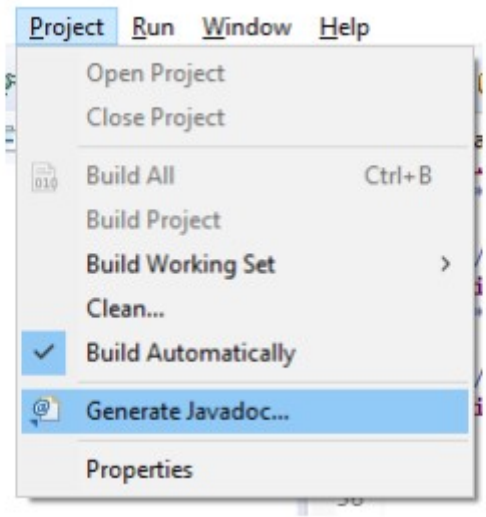
Estas etiquetas deben escribirse al principio de cada clase, miembro o método, dependiendo de qué objeto se desee describir, mediante un comentario iniciado con /** y acabado con */.

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

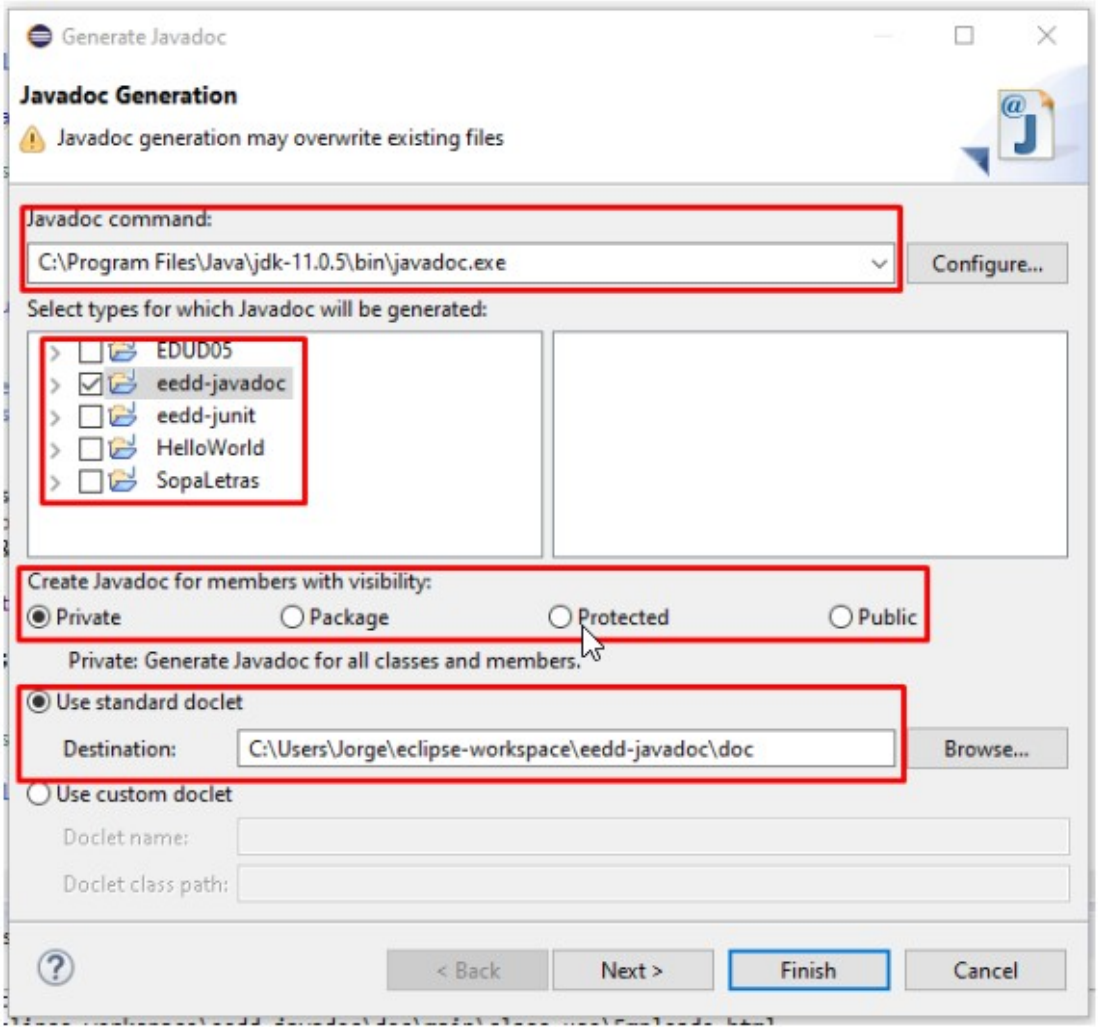
Y a continuación se puede observar su uso en una clase java

Una vez se ha documentado el código, la generación de la documentación HTML asociada es un proceso sencillo haciendo uso de nuestro IDE. Se debe acceder a la opción Project→Generate JavaDoc.



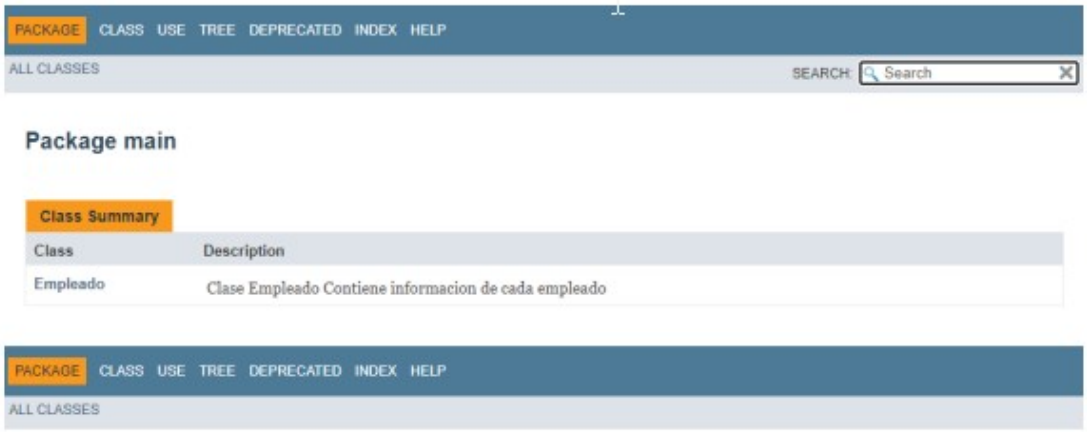
En la siguiente ventana tendremos que seleccionar la ruta donde se encuentra el javadoc (por defecto es la ruta de nuestro jdk), el proyecto, la

visibilidad de los elementos que queremos incluir en la documentación y la ruta donde almacenaremos la documentación.



En las dos ventanas siguientes dejaremos las opciones que vienen por defecto, y pulsaremos “Finish” Viendo en la consola 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: Search
Package main
Class Summary
Class
Description
Empleado
Clase Empleado Contiene informacion de cada empleado
PACKAGE CLASS USE TREE DEPRECATED INDEX HELP
ALL CLASSES