# UD.4 Optimización e documentación

## 1.- REFACTORIZACIÓN.

La **refactorización** es una técnica, que consiste en realizar pequeñas transformaciones en el código de un programa, para mejorar la estructura sin que cambie el comportamiento ni funcionalidad del mismo. Su objetivo es mejorar la estructura, la legibilidad o la eficiencia del código.

Con la refactorización se mejora el diseño del software, hace que el software sea más fácil de entender, hace que el mantenimiento del software sea más sencillo, la refactorización ayuda a que el programa sea más rápido.

La idea de refactorización de código, se basa en el concepto matemático de factorización de polinomios. Así, resulta que $(x + 1)(x - 1)$ se puede expresar como $x^2 - 1$ sin que se altere su sentido.

Algunas pistas que nos pueden indicar la necesidad de refactorizar un programa son:

- Código duplicado.
- Métodos demasiado largos.
- Clases muy grandes o con demasiados métodos.
- Métodos más interesados en los datos de otra clase que en los de la propia.
- Grupos de datos que suelen aparecer juntos y parecen más una clase que datos sueltos.
- Clases con pocas llamadas o que se usan muy poco.
- Exceso de comentarios explicando el código.

Hay que resaltar que la refactorización no cambia el comportamiento observable del software. El software sigue cumpliendo la misma función que hacía antes. Ningún usuario, ya sea usuario final u otro programador, podrá determinar qué cosas han cambiado.

### 1.1.- Concepto.

El concepto de refactorización de código, se base en el concepto matemático de factorización de polinomios.

Podemos definir el concepto de refactorización de dos formas:

- **Refactorización:** Cambio hecho en la estructura interna del software para hacerlo más fácil de entender y fácil de modificar sin modificar su comportamiento.
  Ejemplos de refactorización es "Extraer Método" y "Encapsular Campos". La refactorización es normalmente un cambio pequeño en el software que mejora su mantenimiento.

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

- **Refactorizar:** Reestructurar el software aplicando una serie de refactorizaciones sin cambiar su comportamiento.

El propósito de la refactorización es hacer el software más fácil de entender y de modificar. Se pueden hacer muchos cambios en el software que pueden hacer algún pequeño cambio en el comportamiento observable. Solo los cambios hechos para hacer el software más fácil de entender son refactorizaciones.

Hay que diferenciar la refactorizacón de la optimización. En ambos procesos, se pretende mejorar la estructura interna de una aplicación o componente, sin modificar su comportamiento. Sin embargo, cuando se optimiza, se persigue una mejora del rendimiento, por ejemplo mejorar la velocidad de ejecución, pero esto puede hacer un código más difícil de entender.

Hay que resaltar que la refactorización no cambia el comportamiento observable del software. El software sigue cumpliendo la misma función que hacia antes. Ningún usuario, ya sea usuario final u otro programador, podrá determinar que cosas han cambiado.

## 1.2.- Limitaciones.

La refactorización es un técnica lo suficientemente novedosa para conocer cuáles son los beneficios que aporta, pero falta experiencia para conocer el alcance total de sus limitaciones. Se ha constatado que la refactorización presente problemas en algunos aspectos del desarrollo.

Un área problemática de la refactorización son las bases de datos. Una base de datos presenta muchas dificultades para poder ser modificada, dado la gran cantidad de interdependencias que soporta. Cualquier modificación que se requiera de las bases de datos, incluyendo modificación de esquema y migración de datos, puede ser una tarea muy costosa. Es por ello que la refactorización de una aplicación asociada a una base de datos, siempre será limitada, ya que la aplicación dependerá del diseño de la base de datos.

Otra limitación, es cuando cambiamos interfaces. Cuando refactorizamos, estamos modificando la estructura interna de un programa o de un método. El cambio interno no afecta al comportamiento ni a la interfaz. Sin embargo, si renombramos un método, hay que cambiar todas las referencias que se hacen a él. Siempre que se hace esto se genera un problema si es una interfaz pública. Una solución es mantener las dos interfaces, la nueva y la vieja, ya que si es utilizada por otro clase o parte del proyecto, no podrá referenciarla.

Hay determinados cambios en el diseño que son difíciles de refactorizar. Es muy difícil refactorizar cuando hay un error de diseño o no es recomendable refactorizar, cuando la estructura a modificar es de vital importancia en el diseño de la aplicación.

Hay ocasiones en las que no debería refactorizar en absoluto. Nos podemos encontrar con un código que, aunque se puede refactorizar, sería más fácil reescribirlo desde el principio. Si un código no funciona, no se refactoriza, se reescribe.

## 1.3.- Patrones de refactorización más habituales.

Algunos de los patrones más habituales de refactorización, que vienen ya integrados en la mayoría de los entornos de desarrollos, son los siguientes:

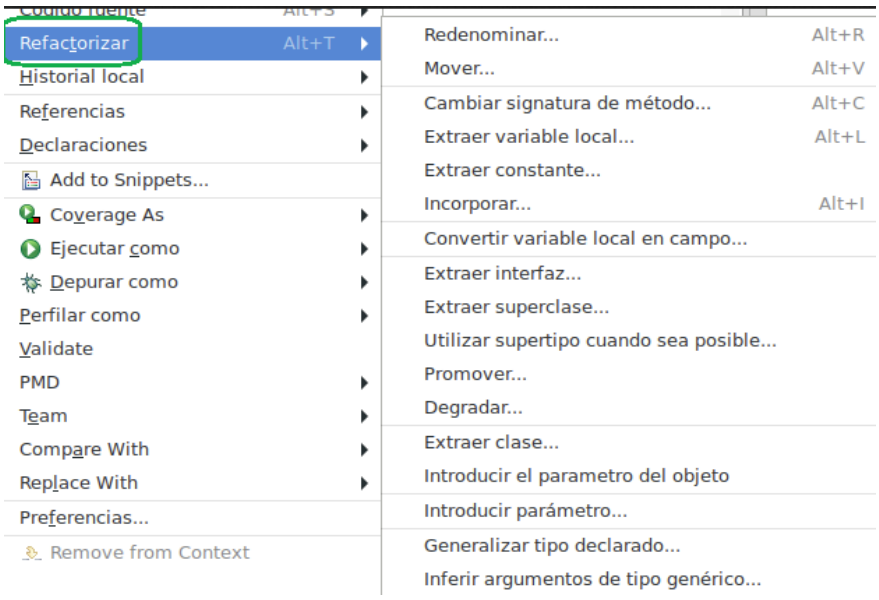
- **Renombrar.** Cambiar el nombre de un paquete, clase, método o campo, por un nombre más significativo.
- **Encapsular campos.** Crear métodos de asignación y de consulta (getters y setters) para los campos de la clase, que permitan un control sobre el acceso de estos campos, debiendo hacerse siempre mediante el uso de estos métodos.
- **Sustituir bloques de código por un método.** En ocasiones se observa que un bloque de código puede constituir el cuerpo de un método, dado que implementa una función por si mismo o aparece repetido en múltiples sitios. De esta forma, cada vez que queramos acceder a ese bloque de código, bastaría con invocar al método.
- **Modificar la extensión del código.** Hacer un código más extenso si se gana en claridad o menos extenso sólo si con eso se gana eficiencia.
- **Reorganizar código condicional complejo.** Patrón aplicable cuando existen varios if o condiciones anidadas o complejas.
- **Crear código común** (en una clase o método) para evitar el código repetido.
- **Mover la clase.** Mover una clase de un paquete a otro, o de un proyecto a otro. Esto implica la actualización en todo el código fuente de las referencias a la clase en su nueva localización.
- **Borrado seguro.** Garantizar que cuando un elemento del código ya no es necesario, se borran todas la referencias a él que había en cualquier parte del proyecto.
- **Cambiar los parámetros del método.** Permite añadir/modificar/eliminar los parámetros en un método y cambiar los modificadores de acceso.
- **Extraer la interfaz.** Crea un nueva interfaz de los métodos public non-static seleccionados en una clase o interfaz.

## 1.4.- Refactorización en Eclipse.

Los entornos de desarrollo actuales, nos proveen de una serie de herramientas que nos facilitan la labor de refactorizar nuestro código. En puntos anteriores, hemos indicado algunos de los patrones que se utilizan para refactorizar el código. Esta labor se puede realizar de forma manual, pero supone una pérdida de tiempo, y podemos inducir a redundancias o a errores en el código que modificamos.

En el Entorno de Desarrollo Eclipse, la refactorización está integrada como una función más, de las utilidades que incorpora.

A continuación se muestra el menú contextual disponible al hacer clic con el botón secundario sobre un fragmento de código en algunas versiones de Eclipse y escoger la opción Refactorizar. Aparece un menú con muchas opciones, de las que estudiaremos algunas.




Nota: el menú mostrado es contextual, por lo que la opción Refactorizar, podrá mostrar algunas opciones diferentes en función de la porción de código sobre la que sea llamado.

La mayor parte de estas funciones permanecen disponibles en las versiones más actuales de Eclipse. A continuación se van a explorar las funciones de refactorización.

### *1.4.1.- Renombrar.*

Es la opción más común, modifica el nombre a cualquier elemento (variable, atributo, método, clase…) y hace los cambios necesarios en las referencias que haya a dicho elemento en todo el proyecto.
Ejemplo: sustituir en el método main el nombre de la variable local teacher por tch. El cambio se realiza en todas sus apariciones.

### *1.4.2.- Mover.*

Cambia una clase de un paquete a otro, afectando a su declaración "package" y a su localización en el disco.

### *1.4.3.- Cambiar signatura del método.*

Modifica la "firma" o cabecera del método. Si el método ha sido ya usado, cambiar el número o tipo de parámetros (así como el tipo de valor devuelto) provocará fallos de compilación.
Es útil para cambiar el nombre de los parámetros, o su orden (Eclipse modificará también el orden de entrada de los parámetros en todas las llamadas al método).

### *1.4.4.- Extraer variable local.*

Crear una variable local inicializada con el valor de un literal (número, String…). Las referencias a esa expresión se modifican por una referencia a la variable.

### *1.4.5.- Extraer constante.*

Exactamente igual que el anterior, pero genera una constante con la expresión seleccionada.

### *1.4.6.- Convertir variable local en atributo.*

A veces se definen variables locales dentro de un método que luego resultan ser relevantes en el ámbito de la clase, por tanto debe ser considerada un atributo de la clase.

### *1.4.7.- Extraer método.*

Convierte el código seleccionado en un método, útil en código que es reutilizado en varios sitios del programa. También puede servir para aligerar un método que es demasiado largo.
Eclipse solo solicita el nombre del método pero descubre automáticamente los parámetros y tipo de retorno necesarios.

### *1.4.8.- Incorporar.*

Hace lo contrario que los "Extract": elimina una declaración de variable, método o constante y coloca su valor (en el caso de variables y constantes) o su código (en el caso de métodos) en aquellos lugares en que se referenciaba a esa variable, método o constante que ya no existirán.
Es muy útil cuando se comprueba que el contenido de una variable o constante se va a usar una sola vez y por tanto no merece la pena almacenarlo, sino que queda más limpio el código en una línea. También cuando se observa que un método sólo se llama una o dos veces, por lo que no merece la pena aislar ese código en un método.

### *1.4.9.- Autoencapsular atributo.*

Convierte una variable de clase en privada y genera los métodos Get y Set públicos para acceder a la misma. Opción también disponible desde el menú código fuente que se verá a continuación.

## 1.5.- Analizadores de código.

*Cada **IDE** incluye herramientas de refactorización y analizadores de código. En el caso se software libre, existen analizadores de código que se pueden añadir como complementos a los entornos de desarrollo.*

El análisis estático de código, es un proceso que tiene como objetivo, evaluar el software, sin llegar a ejecutarlo. Esta técnica se va a aplicar directamente sobre el código fuente, para poder obtener información que nos permita mejorar la base de código. Si el analizador considera que nuestro código fuente tiene una estructura mejorable, nos lo indicará y también nos comunicará la mejora a realizar. Por lo tanto, el uso de

analizadores de código proporciona información sobre algunos aspectos a considerar en la refactorización de los programas.

Las principales funciones de los analizadores es encontrar partes del código que puedan reducir el rendimiento, provocar errores en el software, tener una excesiva complejidad, complicar el flujo de datos, crear problemas de seguridad.

El análisis se realiza siguiendo una serie de reglas predefinidas.

Un ejemplo es **PMD**, una herramienta para Java que basa su funcionamiento en detectar patrones, que son posibles errores en tiempo de ejecución, código que no se puede ejecutar nunca porque no se puede llegar a él, código que puede ser optimizado, expresiones lógicas que pueden ser simplificadas, malos usos del lenguaje, etc.

Otro analizador de código disponible en el mercado es **Sonarcube**, herramienta Open-Source de análisis de calidad del código disponible para gran cantidad de lenguajes de programación.

### *1.5.1.- Instalación de PMD.*

Los analizadores de código estático, se suelen integrar en los Entornos de Desarrollo, aunque en algunos casos, hay que instalarlos como plug-in, tras instalar el IDE. En el caso de Netbeans vamos a instalar el plug-in para PMD. Para ello lo descargamos de este enlace:

http://plugins.netbeans.org/plugin/57270/easypmd

Cómo ya vimos en la unidad anterior, para instalar dicho plugin, en NetBeans vamos a Tools/plugins/Downloaded.

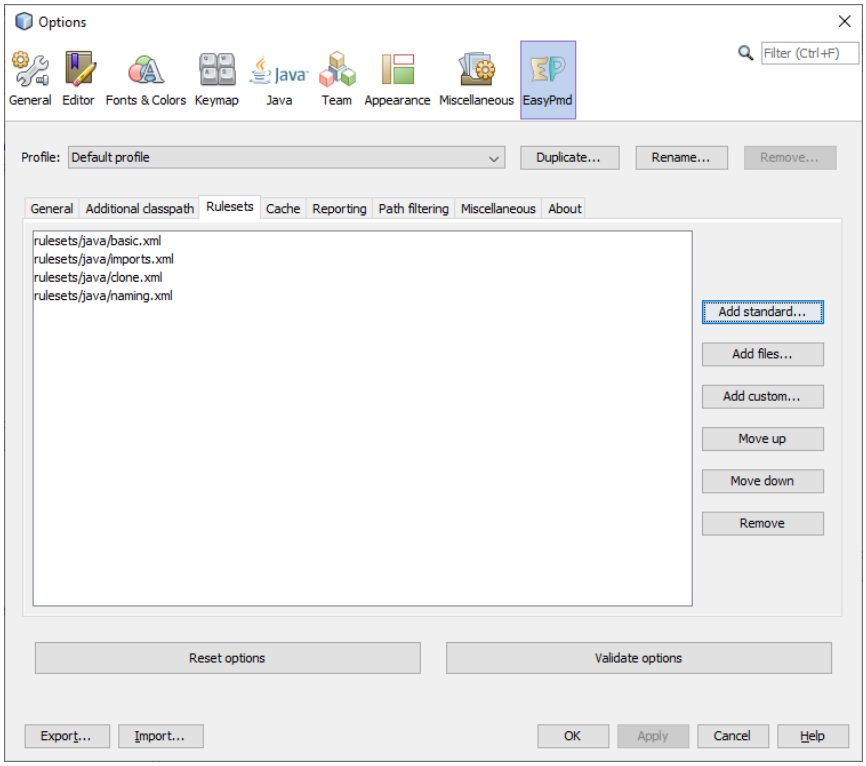
Pulsamos sobre "Add plugins" y buscamos el fichero que nos hemos descargado, lo abrimos y lo instalamos. Ya vimos en la unidad 3 como se hacía. Y reiniciamos NetBeans. Vamos a Plugins/Installed y en Serach ponemos "PMD". Nos aseguramos que ha quedado activado y, si no es así, lo hacemos.

### *1.5.2.- Configuración.*

Como se indicó en el apartado anterior, PMD es un analizador de código estático, capaz de detectar automáticamente un amplio rango de defectos y de inseguridades en el código. PMD se centra en detectar defectos de forma preventiva.

Una vez que tenemos desarrollado nuestro código, si queremos analizarlo con PMD, y obtener el informe del análisis, primero hay que pulsa en el menú Tools/Options y en "EasyPMD", en la pestaña "Rulesets".

Se indican las reglas que están activadas. Puedes activar más reglas. Para ello, pinchas en "Add standard" y añades las reglas que creas conveniente. Por ejemplo, vamos a añadir la regla "Naming". La seleccionamos y aceptar.




Mientras se analiza el código, el plug-in mostrará una barra de progreso en la esquina inferior derecha. El informe producido contiene la localización, el nombre de la regla que no se cumple y la recomendación de cómo se resuelve el problema.

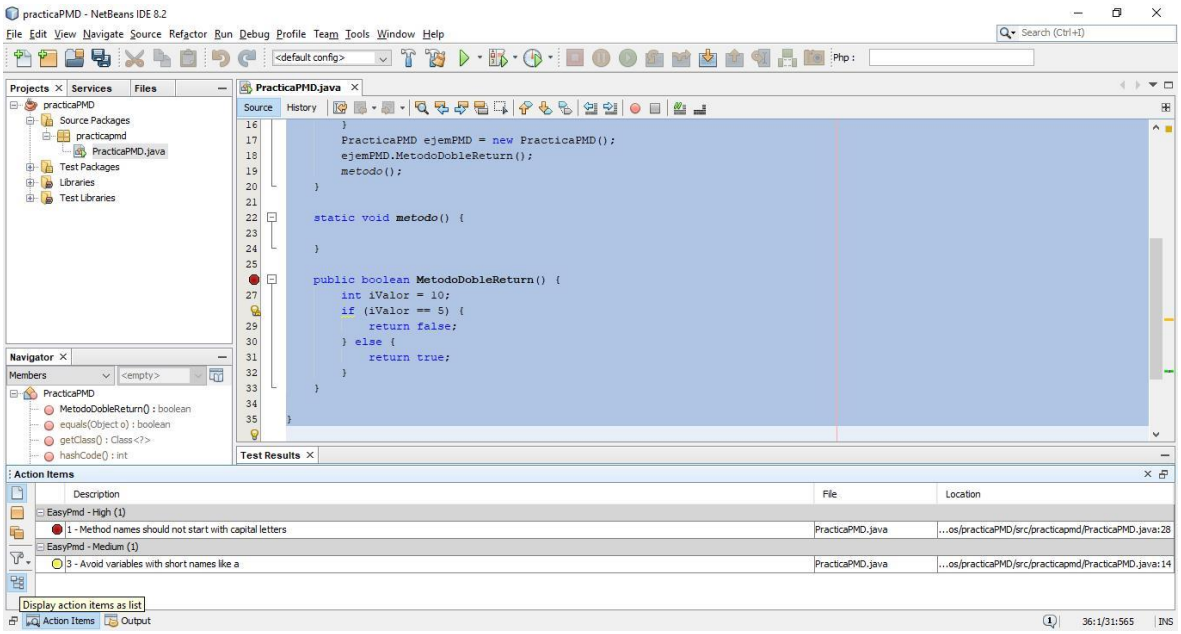
El informe PMD, nos permite navegar por el fichero de clases y en la línea donde se ha detectado el problema. En el número de línea, veremos una marca PMD. Si se posiciona el ratón encima de ella, veremos un tooltip con la descripción del error.

Para probarlo, crea un proyecto en NetBeans con este código:

```
/*
* To change this license header, choose License Headers in Project
Properties.
* To change this template file, choose Tools | Templates
* and open the template in the editor.
*/
package practicapmd;

public class PracticaPMD {

public static void main(String[] args) {
int a;

int altura = 89;
if (altura > 89) {
System.out.println("Eres muy alto");
}
PracticaPMD ejemPMD = new PracticaPMD();
```

```
ejemPMD.MetodoDobleReturn();
metodo();
}

static void metodo() {

}

public boolean MetodoDobleReturn() {
int iValor = 10;
if (iValor == 5)
return false;
else {
return true;
}
}

}
```

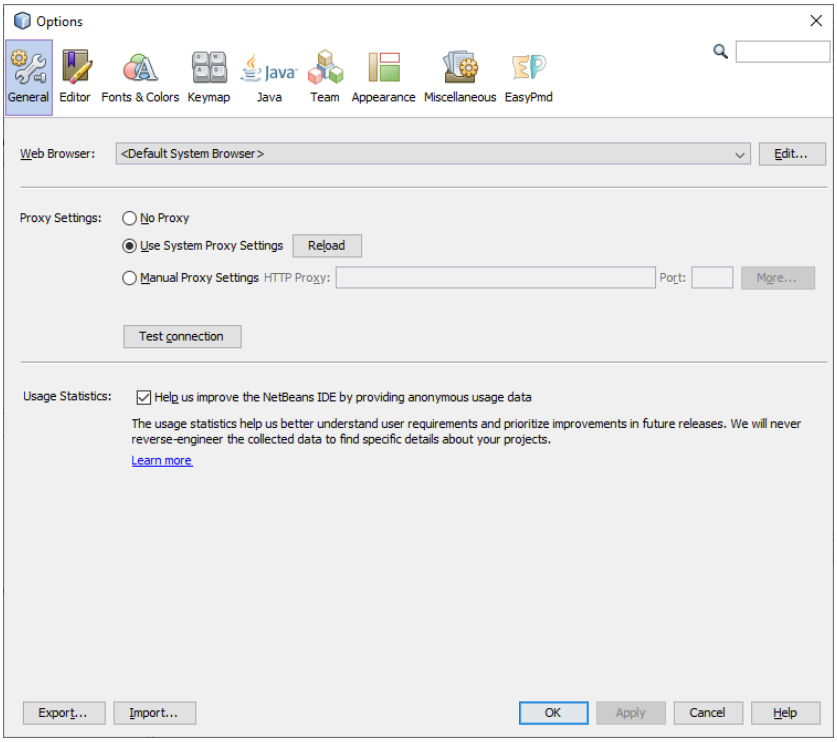
Pulsa sobre el menú "Windows" y sobre "Action Items":



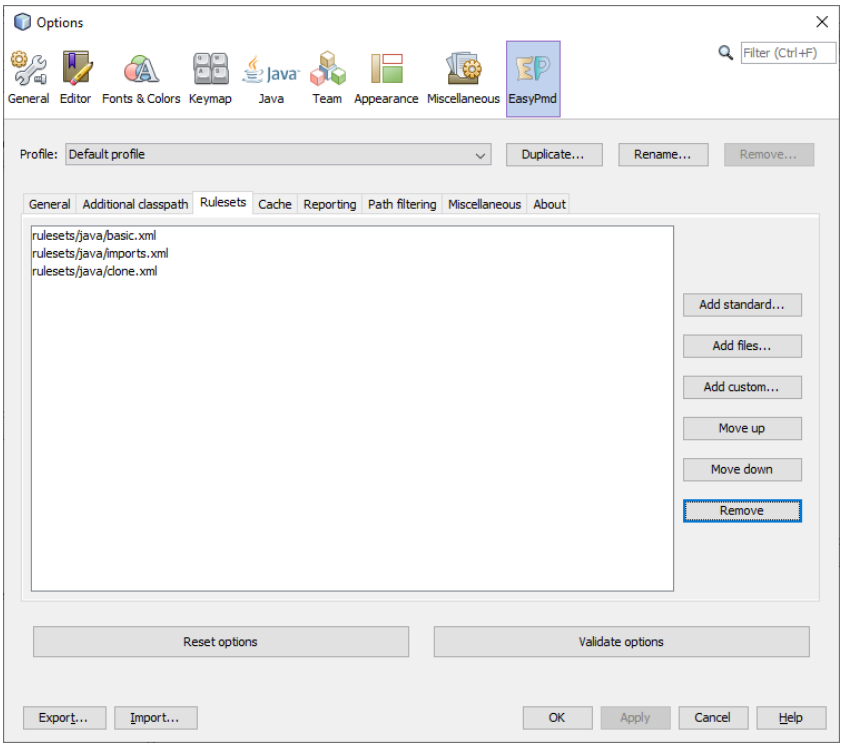

Y te da un informe de las reglas que no cumple tú código. En el ejemplo te avisa de:

- Un error muy grave y es poner a un método un nombre que empieza por una letra mayúscula.
- Un error leve que es el nombre de la variable "a". Un nombre muy corto.

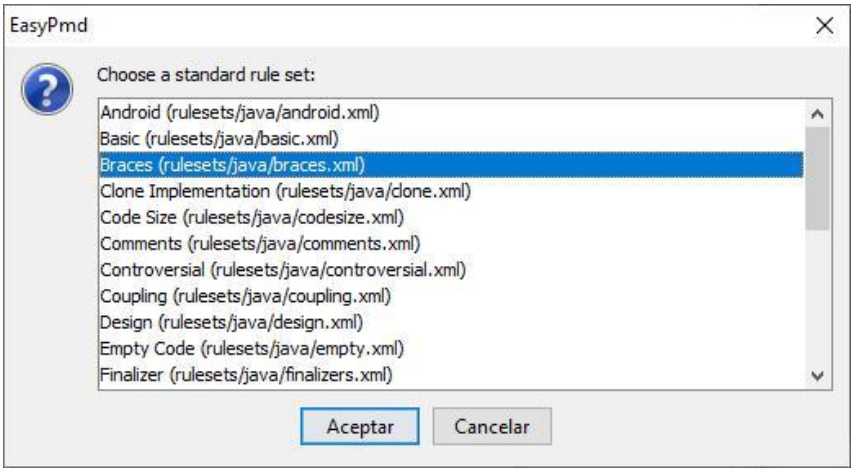
Así podemos ir añadiendo más reglas. Cómo, por ejemplo, que me avise si alguna estructura de control no tiene llaves, aunque tenga una sola instrucción. Para ello, hay que añadir la regla: "braces".
Para ello, pulsamos en el menú Tools/Options:



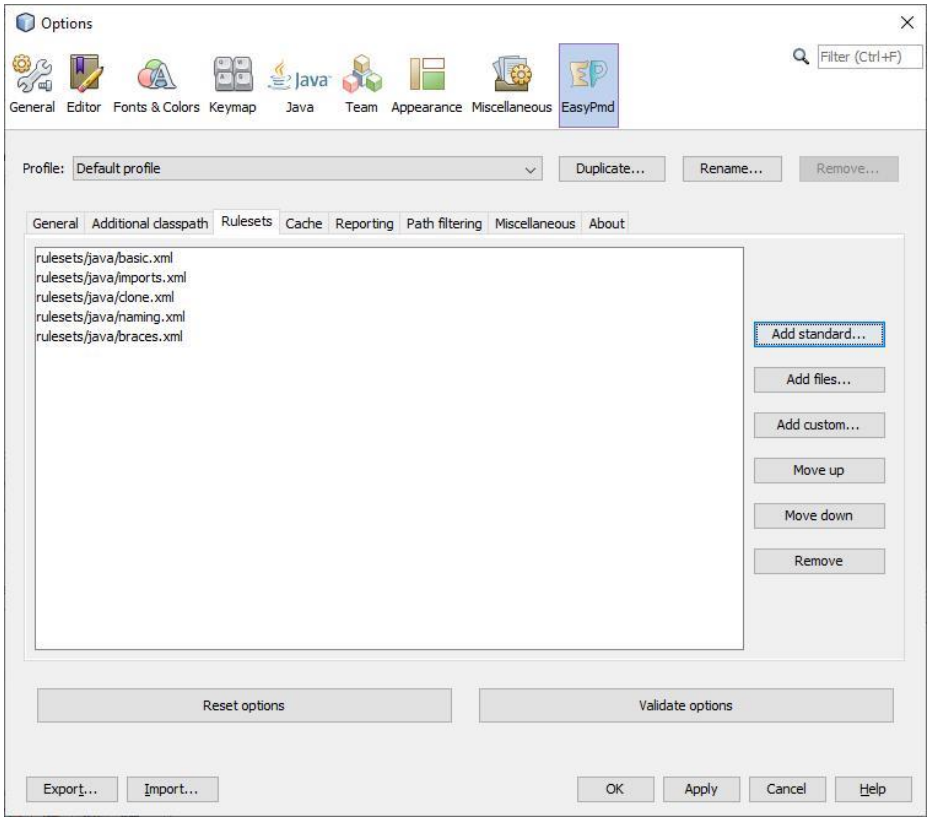

y en "EasyPMD", en la pestaña "Rulesets":



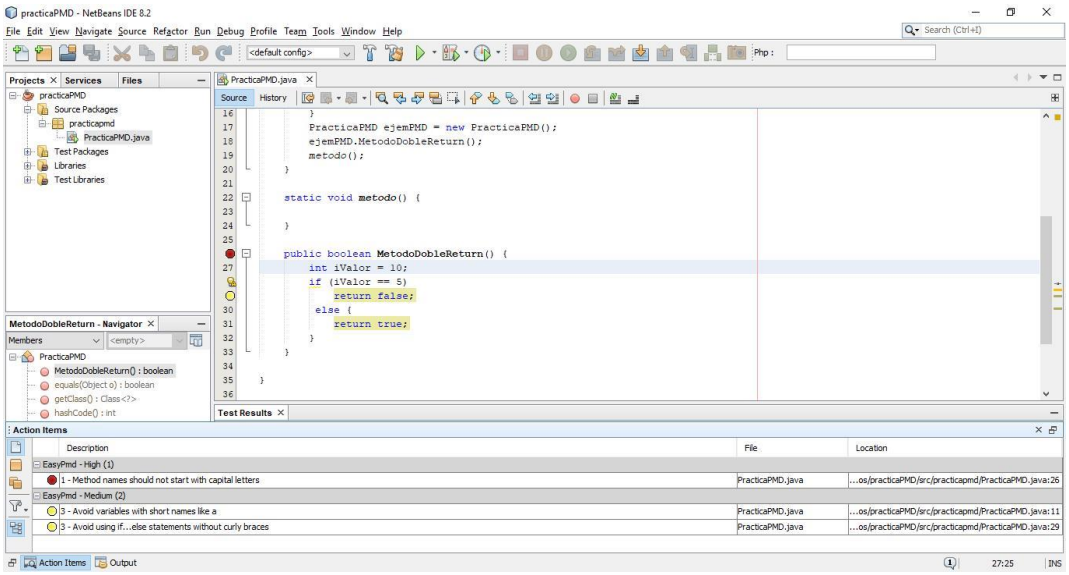

Pulsamos sobre "Add Standard" y añadimos la regla:



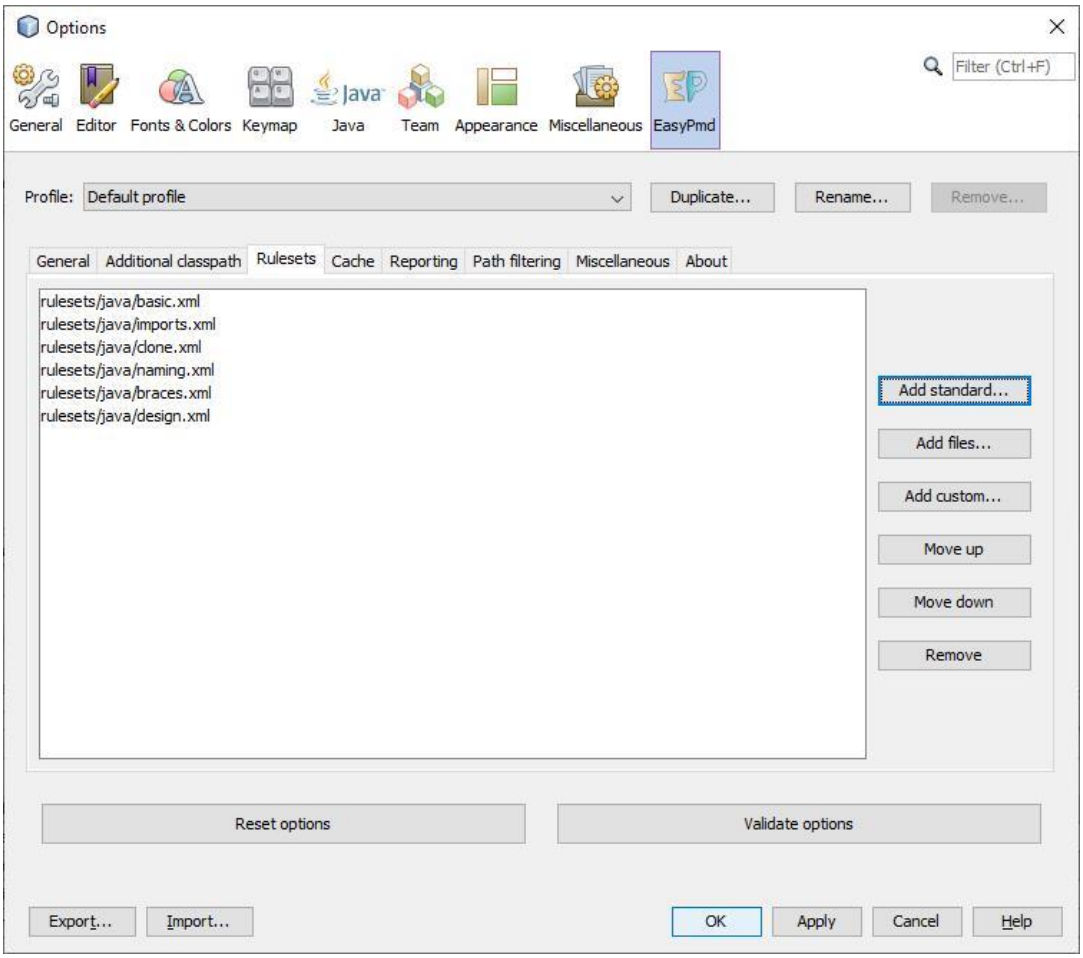

Aceptamos y tendremos esa regla añadida:



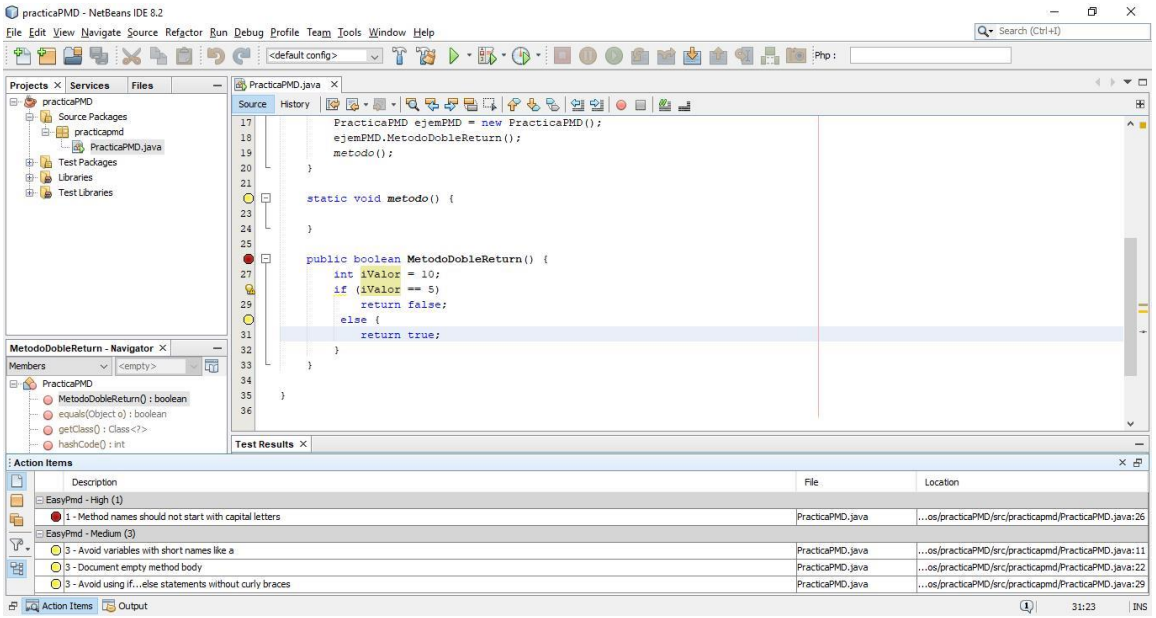

Damos a Ok y veremos como me avisa que hay un if que no tiene llaves:




Ahora añade la regla "Design" y veras como te avisa en el informe que tienes un método vacío.



Información del Action Items:




## 2.- CONTROL DE VERSIONES.

Cuando estamos desarrollando software, el código fuente está cambiando continuamente, bien por el propio desarrollo o por el mantenimiento a que se ve sometido. Además los proyectos en ocasiones se desarrollan por fases o se hacen diferentes entregas al cliente. Todos estos factores hacen necesario un sistema de control de versiones.

Las ventajas de utilizar un sistema de control de versiones son múltiples. Un sistema de control de versiones bien diseñado facilita al equipo de desarrollo su labor, permitiendo que varios programadores trabajen en el mismo proyecto (incluso sobre los mismo archivos) de forma simultánea, permitiendo gestionar los conflictos que se puedan producir por actualizaciones simultáneas sobre el mismo código. Las herramientas de control de versiones proveen de un sitio central donde almacenar el código fuente de la aplicación, así como el historial de cambios realizados a lo largo de la vida del proyecto.

También permite a los desarrolladores volver a versiones estables previas del código fuente si es necesario.

Una versión, desde el punto de vista de la evolución, se define como la forma particular de un objeto en un instante o contexto dado. Se denomina revisión, cuando se refiere a la evolución en el tiempo.

Pueden coexistir varias versiones alternativas en un instante dado y hay que disponer de un método, para designar las diferentes versiones de manera organizada.

Existen distintas alternativas de control de versiones de código abierto, GIT, CVS, Subversion, Mercurial... En ocasiones podrán ser motivo de gestión proyectos software en desarrollo o simplemente conjuntos de archivos de algún otro uso.

Un repositorio interesante para la gestión de proyectos es Bitbucket, está basado en Git y dispone de una versión gratuita para equipos de 5 personas. Ideal para gestionar el proyecto fin de ciclo.

**Para gestionar las distintas versiones que se van generando durante el desarrollo de una aplicación, los IDE, proporcionan herramientas de Control de Versiones y facilitan el desarrollo en equipo de aplicaciones.**

## 2.1.- Tipos de herramientas de control de versiones.

De acuerdo al modo de organizar la información, se pueden clasificar las herramienta de control de versiones como:

- **Sistemas locales:** se trata del control de versiones donde la información se guarda en diferentes directorios en función de sus versiones. Toda la gestión recae sobre el responsable del proyecto y no se dispone de herramientas que automaticen el proceso. Es viable para pequeños proyectos donde el trabajo es desarrollado por un único programador.
- **Sistemas centralizados:** responden a una arquitectura cliente-servidor. Un único equipo tiene todos los archivos en sus diferentes versiones, y los clientes replican esta información en sus entornos de trabajo locales. El principal inconveniente es que el servidor es un dispositivo crítico para el sistema ante posibles fallos.
- **Sistemas distribuidos:** en este modelo cada sistema hace con una copia completa de los ficheros de trabajo y de todas sus versiones. El rol de todos los equipos es de igual a igual y los cambios se pueden sincronizar entre cada par de copias disponibles. Aunque técnicamente todos los repositorios tienen la posibilidad de actuar como punto de referencia; habitualmente funcionan siendo uno el repositorio principal y el resto asumiendo un papel de clientes sincronizando sus cambios con éste.

## 2.2.- Estructura de herramientas de control de versiones.

Las herramientas de control de versiones, suelen estar formadas por un conjunto de elementos, sobre los cuales, se pueden ejecutar órdenes e intercambiar datos entre ellos. Como ejemplo, vamos a analizar ha herramienta CVS.

Una herramienta de control de versiones, como CVS, es un sistema de mantenimiento de código fuente (grupos de archivos en general) extraordinariamente útil para grupos de desarrolladores que trabajan cooperativamente usando alguna clase de red. CVS permite a un grupo de desarrolladores trabajar y modificar concurrentemente ficheros organizados en proyectos. Esto significa que dos o más personas pueden modificar un mismo fichero sin que se pierdan los trabajos de ninguna. Además, las operaciones más habituales son muy sencillas de usar.

CVS utiliza una arquitectura cliente-servidor: un servidor guarda la versión actual del proyecto y su historia, y los clientes conectan al servidor para sacar una copia completa del proyecto, trabajar en esa copia y entonces ingresar sus cambios. Típicamente, cliente y servidor conectan utilizando Internet, pero cliente y

servidor pueden estar en la misma máquina. El servidor normalmente utiliza un sistema operativo similar a Unix, mientras que los clientes CVS pueden funcionar en cualquier de los sistemas operativos más difundidos.

Los clientes pueden también comparar diferentes versiones de ficheros, solicitar una historia completa de los cambios, o sacar una "foto" histórica del proyecto tal como se encontraba en una fecha determinada o en un número de revisión determinado. Muchos proyectos de código abierto permiten el "acceso de lectura anónimo", significando que los clientes pueden sacar y comparar versiones sin necesidad de teclear una contraseña; solamente el ingreso de cambios requiere una contraseña en estos escenarios. Los clientes también pueden utilizar el comando de actualización con el fin de tener sus copias al día con la última versión que se encuentra en el servidor. Esto elimina la necesidad de repetir las descargas del proyecto completo.

El sistema de control de versiones está formado por un conjunto de componentes:

- **Repositorio:** es el lugar de almacenamiento de los datos de los proyectos. Suele ser un directorio en algún ordenador.
- **Módulo:** en un directorio especifico del repositorio. Puede identificar una parte del proyecto o ser el proyecto por completo.
- **Revisión:** es cada una de las versiones parciales o cambios en los archivos o repositorio completo. La evolución del sistema se mide en revisiones. Cada cambio se considera incremental.
- **Etiqueta:** información textual que se añada a un conjunto de archivos o a un módulo completo para indicar alguna información importante.
- **Rama:** revisiones paralelas de un módulo para efectuar cambios sin tocar la evolución principal. Se suele emplear para pruebas o para mantener los cambios en versiones antiguas.

Algunos de los **servicios** que típicamente proporcionan son:

- **Creación de repositorios.** Creación del esqueleto de un repositorio sin información inicial del proyecto.
- **Clonación de repositorios.** La clonación crea un nuevo repositorio y vuelca la información de algún otro repositorio ya existente. Crea una réplica.
- **Descarga de la información del repositorio principal al local**. Sincroniza la copia local con la información disponible en el repositorio principal.
- **Carga de información al repositorio principal desde el local.** Actualiza los cambios realizados en la copia local en el repositorio principal.
- **Gestión de conflictos.** En ocasiones, los cambios que se desean consolidar en el repositorio principal entran en conflicto con otros cambios que hayan sido subidos por algún otro desarrollador. Cuando se da esta situación, las herramientas de control de versiones tratan de combinar automáticamente todos los cambios. Si no es posible sin pérdida de información, muestra al

programador los conflictos acontecidos para que sea éste el que tome la decisión de como combinarlos.

- **Gestión de ramas.** Creación, eliminación , integración de diferencias entre ramas, selección de la rama de trabajo.
- **Información sobre registro de actualizaciones.**
- **Comparativa de versiones.** Genera información sobre las diferencias entre versiones del proyecto.

Las órdenes que se pueden ejecutar son:

- **Checkout:** obtiene un copia del trabajo para poder trabajar con ella.
- **Update:** actualiza la copia con cambios recientes en el repositorio.
- **Commit:** almacena la copia modificada en el repositorio.
- **Abort:** abandona los cambios en la copia de trabajo.

### *2.2.1.- Repositorio.*

**El repositorio es la parte fundamental de un sistema de control de versiones. Almacena toda la información y datos de un proyecto.**

El repositorio es un almacén general de versiones. En la mayoría de las herramientas de control de versiones, suele ser un directorio.

El repositorio centraliza todos los componentes de un mismo sistema, incluyendo las distintas versiones de cada componente. Con el repositorio, se va a conseguir un ahorro de espacio de almacenamiento, ya que estamos evitando guardar por duplicado, los elementos que son comunes a varias versiones. El repositorio nos va a facilitar el almacenaje de la información de la evolución del sistema, ya que, aparte de los datos en sí mismo, también almacena información sobre las versiones, temporización, etc.

El entorno de desarrollo integrado Netbeans usa como sistema de control de versiones CVS. Este sistema, tiene un componente principal, que es el repositorio. En el repositorio se deberán almacenar todos los ficheros de los proyectos, que puedan ser accedidos de forma simultánea por varios desarrolladores.

Cuando usamos una sistema de control de versiones, trabajamos de forma local, sincronizándonos con el repositorio, haciendo los cambios en nuestra copia local, realizado el cambio, se acomete el cambio en el repositorio. Para realizar la sincronización, en el entorno Netbeans, lo realizamos de varias formas:

- Abriendo un proyecto CVS en el IDE.
- Comprobando los archivos de un repositorio.
- Importando los archivos hacia un repositorio.

Si tenemos un proyecto CVS versionado, con el que hemos trabajado, podemos abrirlo en el IDE y podremos acceder a las características de versionado. El IDE escanea nuestro proyectos abiertos y si contienen directorios CVS, el estado del archivo y la ayuda-contextual se activan automáticamente para los proyectos de versiones CVS.

### *2.2.2.- Gestión de versiones y entregas.*

Las versiones hacen referencia a la evolución de un único elemento, dentro de un sistema software. La evolución puede representarse en forma de grafo, donde los nodos son las versiones y los arcos corresponden a la creación de una versión a partir de otra ya existente.

**Grafo de evolución simple:** las revisiones sucesivas de un componente dan lugar a una simple secuencia lineal. Esta evolución no presenta problemas en la organización del repositorio y las versiones se designan mediante números correlativos.

**Variantes:** en este caso, existen varias versiones del componente. El grafo ya no es una secuencia lineal, si no que adopta la forma de un árbol. La numeración de las versiones requerirá dos niveles. El primer número designa la variante (línea de evolución) y el segundo la versión particular (revisión) a lo largo de dicha variante.

La terminología que se usa para referirse a los elementos del grafo son:

- **Tronco** (trunk): es la variante principal.
- **Cabeza** (head): es la última versión del tronco.
- **Ramas (**branches): son la variantes secundarias.
- **Delta:** es el cambio de una revisión respecto a la anterior.

**Propagación de cambios:** cuando se tienen variantes que se desarrollan en paralelo, suele ser necesario aplicar un mismo cambio a varias variantes.

**Fusión de variantes:** en determinados momentos puede dejar de ser necesario mantener una rama independiente. En este caso se puede fundir con otra (MERGE).

**Técnicas de almacenamiento:** como en la mayoría de los casos, las distintas versiones tienen en común gran parte de su contenido, se organiza el almacenamiento para que no se desaproveche espacio repitiendo los datos en común de varias versiones.

- **Deltas directos:** se almacena la primera versión completa, y luego los cambios mínimos necesarios para reconstruir cada nueva versión a partir de la anterior.
- **Deltas inversos:** se almacena completa la última versión del tronco y los cambios necesarios para reconstruir cada versión anterior a partir de la siguiente. En las ramas se mantiene el uso de los deltas directos.
- **Marcado selectivo:** se almacena el texto refundido de todas las versiones como una secuencia lineal, marcando cada sección del conjunto con los números de versiones que corresponde.

En cuanto a la **gestión de entregas**, en primer lugar definimos el concepto de entrega como una instancia de un sistema que se distribuye a los usuarios externos al equipo de desarrollo.

La planificación de la entrega se ocupa de cuándo emitir una versión del sistema como una entrega. La entrega está compuesta por el conjunto de programas ejecutables, los archivos de configuración que

definan como se configura la entrega para una instalación particular, los archivos de datos que se necesitan para el funcionamiento del sistema, un programa de instalación para instalar el sistema en el hardware de destino, documentación electrónica y en papel, y, el embalaje y publicidad asociados, diseñados para esta entrega Actualmente los sistemas se entregan en discos ópticos (CD o DVD) o como archivos de instalación descargables desde la red.

## 2.3.- Herramientas de control de versiones.

Durante el proceso de desarrollo de software, donde todo un equipo de programadores están colaborando en el desarrollo de un proyecto software, los cambios son continuos. Es por ello necesario que existan en todos los lenguajes de programación y en todos los entornos de programación, herramientas que gestionen el control de cambios.

Si nos centramos en Java, actualmente destacan dos herramientas de control de cambios: CVS y Subversion. CVS es una herramienta de código abierto ampliamente utilizada en numerosas organizaciones. Subversion es el sucesor natural de CVS, está rápidamente integrándose en los nuevos proyectos Java, gracias a sus características que lo hacen adaptarse mejor a las modernas prácticas de programación Java. Estas dos herramienta de control de versiones, se integran perfectamente en los entornos de desarrollado para Java, como Netbeans y Eclipse.

Otras herramientas de amplia difusión son:

- **SourceSafe:** es una herramienta que forma parte del entorno de desarrollo Microsoft Visual Studio.
- **Visual Studio Team Foundation Server: es** el sustituto de Source Safe. Es un productor que ofrece control de código fuente, recolección de datos, informes y seguimiento de proyectos, y está destinado a proyectos de colaboración de desarrollo de software.
- **Darcs:** es un sistema de gestión de versiones distribuido. Algunas de sus características son: la posibilidad de hacer commits locales (sin conexión), cada repositorio es una rama en sí misma, independencia de un servidor central, posibilidad de renombrar ficheros, varios métodos de acceso como local, ssh, http y ftp, etc.
- **Git:** esta herramienta de control de versiones, diseñada por Linus Torvalds.
- **Mercurial:** esta herramienta funciona en Linux, Windows y Mac OS X, Es un programa de línea de comandos. Es una herramienta que permite que el desarrollo se haga distribuido, gestionando de forma robusta archivos de texto y binarios. Tiene capacidades avanzadas de ramificación e integración. Es una herramienta que incluye una interfaz web para su configuración y uso.

## 2.4.- Planificación de la gestión de configuraciones.

La Gestión de Configuraciones del software (**GCS**) es un conjunto de actividades desarrolladas para gestionar los cambios a lo largo del ciclo de vida. La planificación de las Gestión de Configuraciones del software, está regulado en el estándar **IEEE** 828.

Cuando se habla de la gestión de configuraciones, se está haciendo referencia a la evolución de todo un conjunto de elementos. Una configuración es una combinación de versiones particulares de los componentes que forman un sistema consistente. Desde el punto de vista de la evolución en el tiempo, es el conjunto de las versiones de los objetos componentes en un instante dado.

Una configuración puede cambiar porque se añaden, eliminan o se modifican elementos. También puede cambiar, debido a la reorganización de los componentes, sin que estos cambien.

Como consecuencia de lo expuesto, es necesario disponer de un método, que nos permita designar las diferentes configuraciones de manera sistemática y planificada. De esta forma se facilita el desarrollo de software de manera evolutiva, mediante cambios sucesivos aplicados a partir de una configuración inicial hasta llegar a una versión final aceptable del producto.

La Gestión de Configuraciones de Software se va a componer de cuatro tareas básicas:

1. **Identificación**. Se trata de establecer estándares de documentación y un esquema de identificación de documentos.
2. **Control de cambios.** Consiste en la evaluación y registro de todos los cambios que se hagan de la configuración software.
3. **Auditorías de configuraciones.** Sirven, junto con las revisiones técnicas formales para garantizar que el cambio se ha implementado correctamente.
4. **Generación de informes.** El sistema software está compuesto por un conjunto de elementos, que evolucionan de manera individual, por consiguiente, se debe garantizar la consistencia del conjunto del sistema.

## 2.5.- Gestión del cambio.

Las herramientas de control de versiones no garantizan un desarrollo razonable, si cualquier componente del equipo de desarrollo de una aplicación puede realizar cambios e integrarlos en el repositorio sin ningún tipo de control. Para garantizar que siempre disponemos de una línea base para continuar el desarrollo, es necesario aplicar controles al desarrollo e integración de los cambios. El control de cambios es un mecanismo que sirve para la evaluación y aprobación de los cambios hechos a los elementos de configuración del software.

Pueden establecerse distintos tipos de control:

1. **Control individual**, antes de aprobarse un nuevo elemento.

Cuando un elemento de la configuración está bajo control individual, el programador responsable cambia la documentación cuando se requiere. El cambio se puede registrar de manera informal, pero no genera ningún documento formal.

2. **Control de gestión u organizado**, conduce a la aprobación de un nuevo elemento.
   Implica un procedimiento de revisión y aprobación para cada cambio propuesto en la configuración. Como en el control individual, el control a nivel de proyecto ocurre durante el proceso de desarrollo pero es usado después de que haya sido aprobado un elemento de la configuración software. El cambio es registrado formalmente y es visible para la gestión.

3. **Control formal**, se realiza durante el mantenimiento.
   Ocurre durante la fase de mantenimiento del ciclo de vida software. El impacto de cada tarea de mantenimiento se evalúa por un Comité de Control de Cambios, el cuál aprueba la modificaciones de la configuración software.

## 3.- DOCUMENTACIÓN.

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

En la actualidad, el desarrollo rápido de aplicaciones, en muchos casos, va en detrimento de una buena documentación del código. Si el código no está documentado, puede resultar bastante difícil de entender, y por tanto de solucionar errores y de mantenerlo.

La labor de documentación es fundamental para la detección de errores y para su mantenimiento posterior, que en muchos casos, es realizado por personas diferentes a las que intervinieron en su creación. Hay que tener en cuenta que todos los programas tienen errores y todos los programas sufren modificaciones a los largo de su vida.

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

La documentación explicará cuál es la finalidad de un clase, de un paquete, qué hace un método, para qué sirve una variable, qué se espera del uso de una variable, qué algoritmo se usa, por qué hemos implementado de una manera y no de otra, qué se podría mejorar en el futuro, etc.

## 3.1.- Uso de comentarios.

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

Es la primera alternativa que surge para documentar código. Con los comentarios, documentamos la funcionalidad de una línea de código, de un método o el comportamiento de una determinada clase.

En principio, los comentarios tienen dos propósitos diferentes:

- **Explicar el objetivo de las sentencias.** De forma que el programador o programadora, sepa en todo momento la función de esa sentencia, tanto si lo diseñaron como si son otros los que quieren entenderlo o modificarlo.
- **Explicar qué realiza un método, o clase, no cómo lo realiza.** En este caso, se trata de explicar los valores que va a devolver un método, pero no se trata de explicar cómo se ha diseñado.

En el caso del lenguaje Java, C# y C, los comentarios, se implementan de forma similar. Cuando se trata de un comentario de una sola línea, se usan los caracteres // seguidos del comentario. Para comentarios multilínea, los caracteres a utilizar son /* y */, quedaría: /* comentario-multilínea */.

No es obligatorio, pero en muchas situaciones es conveniente, poner los comentarios al principio de un fragmento de código que no resulta lo suficientemente claro, a la largo de bucles, o si hay alguna línea de código que no resulta evidente y pueda llevarnos a confusión.

Insertando comentario en el código más difícil de entender, y utilizando la documentación generada por alguna de las herramientas citadas anteriormente, se genera la suficiente información para ayudar a cualquier nuevo programador o programadora.

Hay que tener en cuenta, que si el código es modificado, también se deberán modificar los comentarios.

## 3.2.- Documentación de clases.

Existen diferentes herramientas que permiten automatizar, completar y enriquecer nuestra documentación. Podemos citar JavaDoc, SchemeSpy y Doxygen, que producen una documentación actualizada, precisa y utilizable en línea, incluyendo además, con SchemeSpy y Doxygen, modelos de bases de datos gráficos y diagramas.

Las clases que se implementan en un aplicación, deben de incluir comentarios. Al utilizar un entorno de programación para la implementación de la clase, debemos seguir una serie de pautas, muchas de las cuales las realiza el IDE de forma trasparente, en el diseño y documentación del código.

Los entornos de programación que implementa Java, como Eclipse o Netbeanss, incluyen una herramienta que va a generar páginas HTML de documentación a partir de los comentarios incluidos en el código fuente. La herramienta ya se ha indicado en los puntos anteriores, y es JavaDoc.

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

- Los comentarios son obligatorios con JavaDoc, y se deben incorporar al principio de cada clase, al principio de cada método y al principio de cada variable de clase. Se escriben empezando por /** y terminando con */ , estos comentarios pueden ocupar varias líneas. Todos los comentarios hechos con // y /*comentario*/ no se incluirán en la documentación de la clase.
- Los comentarios pueden ser a nivel de clase, a nivel de variable y a nivel de método.
- La documentación se genera para métodos public y protected.
- Se puede usar tag para documentar diferentes aspectos determinados del código, como parámetros.

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

Dentro de la la clase, también se documentan los constructores y los métodos.

Existe una serie de etiquetas que fijan como se presentará la información en la documentación resultante JavaDoc. En la url https://en.wikipedia.org/wiki/Javadoc aparece una colección de palabras reservadas (etiquetas) definidas en Javadoc. A continuación se muestran algunas de las más utilizadas.

| Etiqueta y parámetros | Uso | Asociada a |
|---|---|---|
| **@author** *nombre* | Nombre del autor (programador) | Clase, interfaz |
| **@version** numero-*version* | Comentario con datos indicativos del número de versión. | Clase, interfaz |
| **@since** *numero-version* | Fecha desde la que está presente la clase. | Clase, interfaz, campo, método. |
| **@see** *referencia* | Permite crear una referencia a la documentación de otra clase o método. | Clase, interfaz, campo, método. |
| **@param** seguido del nombre del parámetro | Describe un parámetro de un método. | Método. |
| **@return** *descripción* | Describe el valor devuelto de un método. | Método. |
| **@exception** *clase descripción*<br>**@throws** *clase descripción* | Comentario sobre las excepciones que lanza. | Método. |
| **@deprecated** *descripción* | Describe un método obsoleto. | Clase, interfaz, campo, método. |

Los campos de una clase, también pueden incluir comentarios, aunque no existen etiquetas obligatorias en JavaDoc.

## Anexo I.- Repositorio CVS.

**Conectar con un repositorio.**

Si queremos conectar con un repositorio remoto desde el IDE, entonces chequearemos los ficheros e inmediatamente comenzará a trabajar con ellos. Se hace de la siguiente forma:

1. En el IDE NetBeans elegimos Herramientas - CVS - Extraer. El asistente para extraer el modulo se nos abrirá.
2. En el primer panel del asistente, se introduce la localización del repositorio definido conCVSROOT. El IDE soporta diferentes formatos de CVSROOT, dependiendo de si el repositorio es local o remoto, y del método utilizado para conectarnos a él.

**Métodos de conexión a CVSROOT**

| Método | Descripción | Ejemplo |
|---|---|---|
| pserver | Contraseña de servidor remoto. | :pserser:username@hotsname:/repository_path |
| ext | Acceso usando Remote Shell (RSH) o Secure Shell (SSH). | :ext:username@hostname:/repository_path |
| local | Acceso a un repositorio local. | :local:/repository_path<br>(requiere un CVS externo ejecutable) |
| fork | Acceso a un repositorio local usando un protocolo remoto. | :fork:/repository_path<br>(requiere un CVS externo ejecutable) |

3. En el panel de Módulos a extraer del asistente, especificamos el módulo que queremos extraer, en el campo Módulo. Si no sabemos el nombre del módulo, podemos extraerlo haciendo clic en el botón Examinar. Si lo que queremos es conectar a un repositorio remoto desde el IDE, debemos extraer los ficheros e inmediatamente trabajar con ellos, se hace de la siguiente forma:

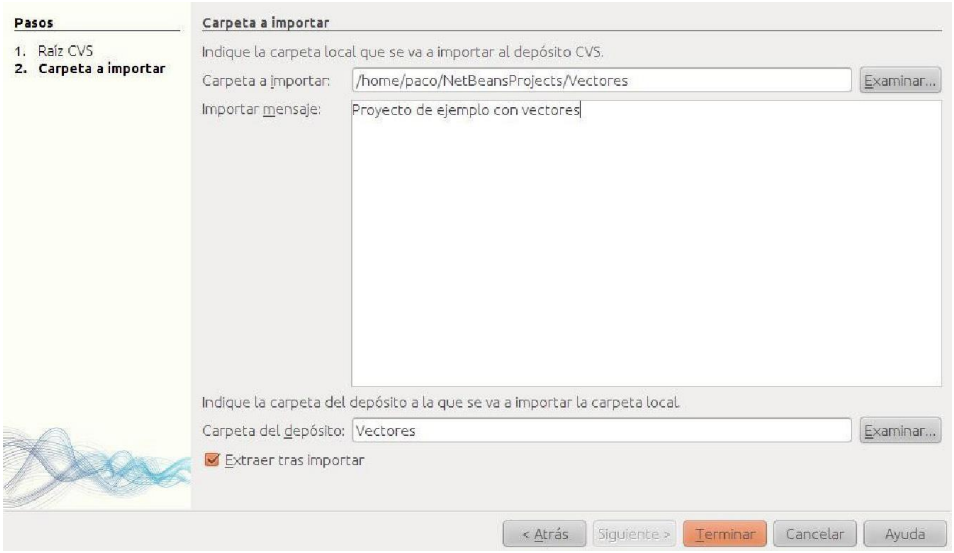

4. En el campo de texto, ponemos el nombre del Módulo que queremos extraer. Podemos usar el botón Examinar, para buscar aquellos módulos disponibles.
5. En el campo Carpeta local, pondremos la ruta de nuestro ordenador donde queremos extraer los archivos. Pulsaremos el botón Terminar para que el proceso comience.

**Importar archivos hacia un repositorio.**

Alternativamente, podemos importar un proyecto en el que estemos trabajando en el IDE hacia un repositorio remoto, seguiremos trabajando en el IDE después de que haya sido versionado con el repositorio CVS.

Para importar un proyecto a un repositorio:

1. Desde la ventana de proyectos, seleccionamos un proyecto que no este versionado, y elegimos Equipo - CVS - Importar al depósito. Se abrirá el asistente de importación CVS.
2. En el panel Raíz CVS del asistente, se especifica la localización del repositorio definido como CVSROOT. Dependiendo del método usado, necesitaremos utilizar más información, como la contraseña o configuración del proxy, para conectarnos a un repositorio remoto.
3. En Carpeta a importar, se especifica el directorio local donde queremos colocar el repositorio.

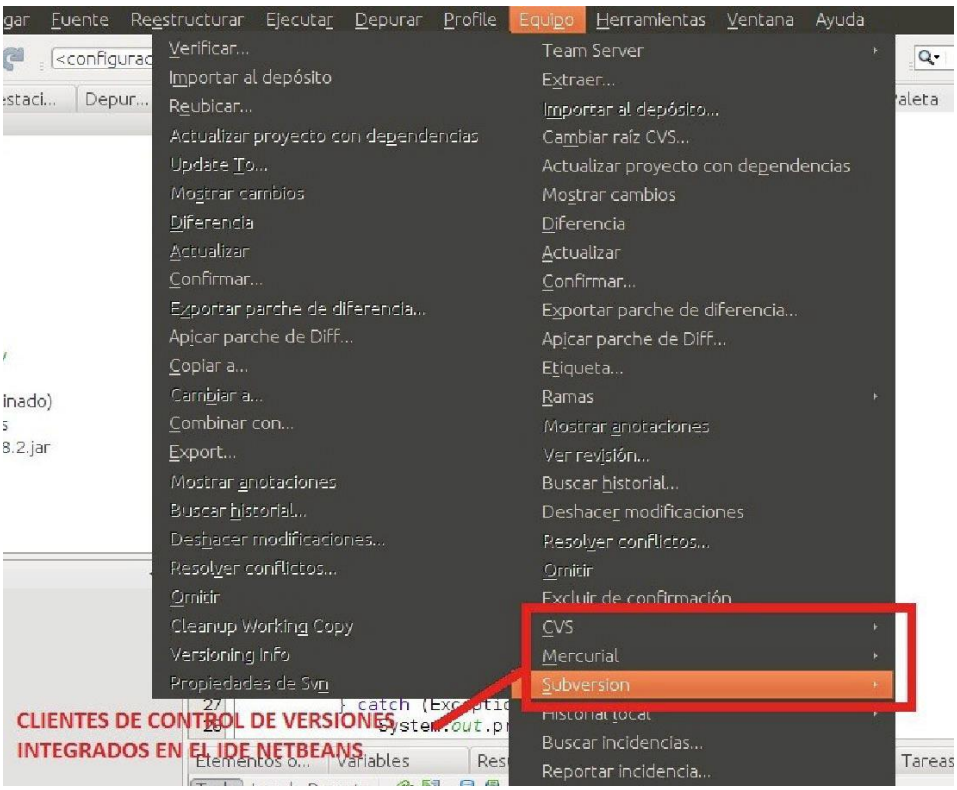



4. En el área de texto Importar mensaje, pondremos una descripción del proyecto que estamos importando.
5. Se especifica la localización del repositorio donde queremos importar el proyecto, escribiendo la ruta en Carpeta del depósito. De forma alternativa, se puede hacer clic en el botón Examinar para ir a una localización especifica en el repositorio. Al hacer clic en el botón Terminar se inicia la importación. El IDE copiara los ficheros del proyecto en el repositorio.

## Anexo II.- Clientes de Control de Versiones en Netbeans.

El entorno de desarrollo integrado NetBeans 6.9.1., incorpora tres clientes de control de versiones. Estos tres clientes son CVS, Subversion y Mercurial.

Cuando estemos desarrollando una aplicación, si queremos gestionar el control de versiones, podemos utilizar cualquiera de los tres.




### CVS

En el caso del IDE NetBeans, es importante su uso para mantener de forma segura, tanto los programas como las pruebas, en un repositorio de código. La configuración de CVS puede suponer unos cinco minutos, que dentro del tiempo de desarrollo de una aplicación, es un tiempo despreciable. Sin embargo, el uso de CVS en NetBeans, nos va a gestionar las distintas versiones del código que se desarrollen y nos va a evitar pérdidas de datos y de código.

Para utilizar CVS, es necesario la instalación de un cliente CVS, sin embargo, NetBeans nos va a evitar esta instalación, ya que incorpora un cliente CVS en Java.

CVS se puede utilizar con NetBeans de tres formas:

1. Desde la línea de comando CVS, donde se escribirán los comandos CVS y de esta forma se interactuará con el repositorio.
2. Si CVS está incorporado a NetBeans, dispondremos de un conjunto de clases Java que imitan los comandos clásicos de CVS.

3. Con fórmulas de línea de comandos CVS genéricas basadas en plantillas proporcionadas por el usuario, que son pasadas al shell del sistema operativo.

En los casos anteriores, donde se utilizan plantillas, las plantillas están parametrizadas, con la lógica NetBeans, que sustituye los operadores CVS por variables. Esto quiere decir, que NetBeans se va a encargar de decidir el nombre de los ficheros, que datos se importan o exportan del repositorio, sin que el usuario deba conocer toda la lógica interna de CVS. Nos evitamos tener que conocer todos los parámetros y opciones necesarias para realizar las operaciones con CVS.

## SUBVERSION

Subversion es un sistema de control de versiones de software libre, que se ha convertido en el sustituto natural de CVS. A diferencia de CVS, los archivos que se versionan no tienen un número de versión independiente, todo el repositorio tiene un único número de versión que identifica un estado común de todos los archivos del repositorio en un instante determinado.

Subversion puede acceder al repositorio a través de redes. Esto implica que varias personas pueden acceder, modificar y administrar el mismo conjunto de datos, con lo que se va a fomentar la colaboración. Se puede progresar más rápidamente sin un único conducto por el cual deban pasar todas las modificaciones. Y puesto que el trabajo se encuentra bajo el control de versiones, no se compromete la calidad del software que se desarrolla.

Recuerda visitar la guía de Subversion que te sugerimos en la unidad de trabajo.

## MERCURIAL

Mercurial es un sistema de control de versiones que utiliza sobre todo la línea de comandos. Todas las operaciones de Mercurial se invocan como opciones dadas a su motor hg. Las principales metas de Mercurial incluyen un gran rendimiento y escalabilidad, desarrollo completamente distribuido, sin necesidad de un servidor, gestión robusta de archivos de texto y binarios, y capacidades avanzadas de ramificación e integración.

## Anexo III.- GIT

En un proyecto de trabajo colaborativo, cada miembro del equipo realiza las tareas que le han sido asignadas, pero al terminar cada uno de ellos, habrá que preguntarse:

- ¿Cómo se integran todas las partes?.
- Si hay errores o cambios, ¿cómo se actualizan los nuevos cambios?.
- Si hay una nueva versión, ¿cómo se gestionan los conjuntos de módulos compatibles con cierta versión?, ¿cómo se recuperan versiones anteriores?, ¿cómo se fusionan versiones? ...

En el punto 2.3 hemos nombrado las diferentes herramientas de control de versiones. Vamos a ver en este anexo, en detalle, la herramienta "GIT".

Algunas de las características de GIT son:

- Gratis, de código abierto.
- Muy popular, disponible en múltiples plataformas.
- Distribuido. Las diferentes réplicas de los repositorios tienen una relación de igual a igual. Esta es una consideración técnica, en la práctica una de ellas suele adquirir el rol de repositorio principal.
- Basado en changesets. Si varios componentes del proyecto han cambiado respecto a la última versión, todos ellos son volcados al repositorio de una vez, esta entrega se considera atómica.
- No bloqueante. Es posible que varios repositorios locales estén trabajando de forma simultánea sobre los mismos componentes, posteriormente habrá que integrar las modificaciones efectuadas en todos ellos para obtener un versionado común.

### a.- Terminología

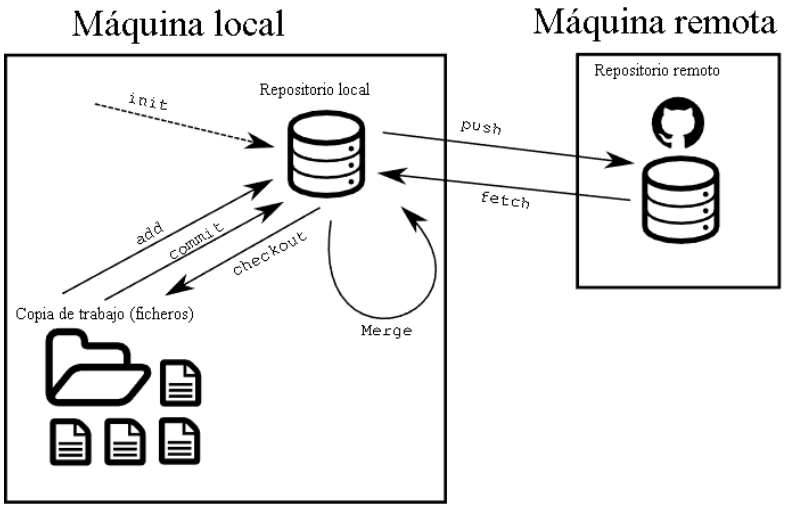
Algunos de los términos de uso habitual en GIT son:

- **Repositorio:** base de datos con las sucesivas versiones. Un repositorio podrá estar compuesta de diversas líneas de desarrollo o ramas. Típicamente habrá un repositorio común y tantos repositorios locales como participantes en la actividad colaborativa.
- **Rama master o trunk:** rama principal del desarrollo.
- **Rama (Branch):** línea de desarrollo paralela a la rama master. Podría recoger trabajos para diferentes clientes.
- **Área de trabajo:** ficheros sobre los que trabaja el programador.
- **Commit:** confirmación de una nueva versión (de la copia de trabajo al repositorio local).
- **Checkout:** (re)generación de la copia de trabajo a partir de la información del repositorio local.
- **Fusión (**Merge): acumular cambios en una misma versión . Permite combinar diferentes branchs en el repositorio local.
- **Comentario:** cada commit deberá documentar sus efectos y motivaciones.
- **Push:** actualización del repositorio remoto con la información del repositorio local.
- **Fetch:** actualización del repositorio local con la información del repositorio remoto.

- **Pull:** actualización del repositorio local y del área de trabajo con la información del repositorio remoto.

## b.- Escenario de trabajo con GIT.

En un proyecto colaborativo coordinado con GIT, un escenario típico de trabajo sería el siguiente.




En la figura se puede ver una máquina remota donde se encuentra el repositorio principal o remoto, sitio donde se pone en común todo el trabajo del conjunto de integrantes del equipo. Cada desarrollador tendrá su propio espacio de trabajo, en la figura se muestra una única máquina local.

Inicialmente, el entorno local tendrá un repositorio que será una réplica idéntica del repositorio remoto. Además el programador dispondrá de su área de trabajo, donde irá actualizando el código. Cuando unos cambios cobran suficiente entidad como para constituir una versión, éstos son guardados en el repositorio local. Finalmente, si el programador desea poner en común con los otros miembros del proyecto sus cambios, enviará sus actualizaciones desde el repositorio local al común.

## c.- Flujo de trabajo.

El flujo de trabajo normal del programador consistirá en las siguientes actuaciones:

- Modificar el área de trabajo. Actualizando el contenido de los ficheros del proyecto o creando/eliminando algunos de ellos.
- Seleccionar aquellos ficheros que se desea formen parte de la nueva versión del proyecto. Los ficheros que no sean seleccionados, aunque hayan sido cambiados no serán guardados en el repositorio.
- Comprobar cambios. Habrá que cerciorarse que los cambios introducidos en el proyecto son los deseados.
- Salvar las modificaciones introducidas en el área de trabajo al repositorio local.

Los cuatro puntos anteriores se podrán repetir tantas veces como sea necesario. A continuación habrá que:

- Conseguir los cambios de otros compañeros. Es muy probable que en el repositorio común existan actualizaciones que haya introducido algún otro colaborador, por lo que se hace necesario combinar esos cambios con los que queremos compartir nosotros.
- Actualizar el repositorio común desde el local para compartir nuestros cambios.

## d.- Resumen de comandos GIT.

A continuación se muestran algunos de los comandos más utilizados en GIT.

- **Init:** Crea un nuevo repositorio en el directorio inicialmente vacío. Se almacena en el directorio oculto .git.
- **Clone:** Crea un nuevo repositorio, copia de uno ya existente. El repositorio replicado se indica mediante su URL. La copia de trabajo inicial será la de la rama MASTER.
- **add, rm, mv:** Permite identificar los ficheros del área de trabajo que van a ser incluidos en el repositorio. Para seleccionar ficheros se usa el comando add, para deseleccionarlos el comando a utilizar será rm. El borrado de ficheros no tiene carácter retroactivo sobre las versiones anteriores.
- **Status:** Imprime un resumen del estado de los ficheros de la copia de trabajo. Los ficheros se pueden encontrar en los siguientes estados:
    - Untracked: no guardados en el repositorio. GIT ignora el fichero.
    - Unmodified: igual que en la rama activa del repositorio.
    - Modified: con cambios respecto al repositorio, pero que no está previsto que sean guardados al hacer commit.
    - Staged: el fichero está en el index. Será parte del próximo changeset que se añadirá al repositorio.
- **Commit:** Crea una nueva versión en el repositorio local, incluyendo los cambios en estado Staged procedentes del área de trabajo. Los ficheros pasan a estar seleccionados como parte de las nuevas versiones con el comando add. El comando git commit -a incluye en el index todos los ficheros del área de trabajo diferentes a los disponibles en el repositorio local, a continuación aplica un commit creando una nueva versión.
- **Push:** Sube las versiones del repositorio local a un repositorio remoto. El repositorio remoto puede establecerse con:
    - git clone (en este caso se denomina origin).
    - git remote add

  El repositorio debe tener todas las versiones del repositorio remoto, en otro caso, deberá realizarse un git pull previo.

- **Checkout:** Extrae versiones del repositorio local a la copia de trabajo, la acción se puede realizar sobre diferentes elementos del repositorio:
    - Un fichero: git checkout fichero.

- Una version: git checkout hash_de_versión.
- Un branch: git checkout branch.

También permite crear nuevas ramas, con git checkout -b nombre_rama.

- **Branch:** Lista, crea o elimina ramas en un repositorio.
- Otros comandos de interés son:
  - **git log:** historia de cambios de un repositorio.
  - **git diff:** Muestra los cambios de los ficheros en estado modified.
  - **git merge:** fusión de dos ramas.
  - **git gui:** interfaz gráfica para utilizar GIT.

## e.- Gitk

Gitk es un visor del repositorio local. Permite revisar ramas, commits y merges entre otros. Es una aplicación útil para:

- Recuperar versiones antiguas de un fichero.
- Visualizar las ramas de un repositorio.
- Hacer un seguimiento de la actividad en el repositorio.