

## Caso práctico

### Situación

BK programación se encuentra desarrollando la aplicación de gestión hotelera.

Como parte del proceso de desarrollo surgen una serie de problemas: hay un código que presenta algunas implementaciones que se pueden mejorar aplicando refactorización, nos encontramos que los miembros del equipo de desarrollo están modificando constantemente métodos o clases, creando diferentes versiones de las mismas y falta una documentación clara y precisa del código.

**Ada** propone a **Juan** y a **María** (programadores principales de la aplicación) que usen los patrones generales de refactorización para conseguir un código de mejor calidad, más fácil de entender, más fácil de probar y con menor probabilidad de generar errores. **Ana** va a intentar ayudar a **Juan**, y **Carlos** a **María**, pero no conocen nada de refactorización, ni entiende la necesidad de realizar este trabajo "extra".

Como todos los miembros de BK Programación trabajan sobre los mismo proyectos, **Ada** debe coordinar el trabajo de todos ellos, por lo que propone que cada uno de ellos utilice un cliente de control de versiones, de forma que ella, de manera centralizada, pueda gestionar la configuración del software.

Los miembros con menor experiencia, **Carlos** y **Ana**, van a ir generando, utilizando herramientas de documentación automatizadas, la documentación de las clases y del código generado por **Juan** y **María**.

**Ada** va a encargarse de la Gestión de Configuraciones del Software, ya que es una labor fundamental para cualquier jefe de proyecto. Asimismo, debe garantizar que el código generado por **Juan** y **María** esté refactorizado.



**Materiales formativos de FP Online propiedad del Ministerio de Educación, Cultura y Deporte.**

[Aviso Legal](#)

# 1.- Refactorización.

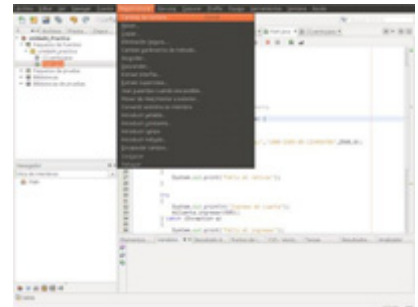
## Caso práctico



Gran parte de las clases, métodos y módulos que forman parte de la aplicación de Gestión Hotelera han sido implementados, surge ahora una pregunta. ¿Podemos mejorar la estructura del código y que sea de mayor calidad, sin que cambie su comportamiento? ¿Cómo hacerlo? ¿Qué patrones hay que seguir?

La refactorización es una disciplina 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 interna del código. Es una tarea que pretender limpiar el código minimizando la posibilidad de introducir errores.

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 nos ayuda a encontrar errores y a que nuestro programa sea más rápido.



Cuando se refactoriza se está mejorando el diseño del código después de haberlo escrito. Podemos partir de un mal diseño y, aplicando la refactorización, llegaremos a un código bien diseñado. Cada paso es simple, por ejemplo mover una propiedad desde una clase a otra, convertir determinado código en un nuevo método, etc. La acumulación de todos estos pequeños cambios pueden mejorar de forma ostensible el diseño.

## Para saber más

Puedes visitar la siguiente página web (en inglés), donde se presenta el proceso de refactorización de aplicaciones Java con Netbeans.

[Refactorización en Netbeans.](#)

## 1.1.- Concepto.

### Caso práctico



**Juan** va a empezar a refactorizar parte del código que ha generado. **Ana** no sabe que es refactorizar, así que **Juan** le va a explicar las bases del proceso de refactorización.

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

FACTORIZACIÓN DE POLINOMIOS

$$X^2 - 1 = (X + 1)(X - 1)$$

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 refactorización 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 hacía antes. Ningún usuario, ya sea usuario final u otro programador, podrá determinar que cosas han cambiado.

### Reflexiona

Con la refactorización de código, estamos modificando un código que funciona correctamente, ¿merece la pena el esfuerzo de refactorizar un código ya implementado?

## 1.2.- Limitaciones.

### Caso práctico

**Ana** se muestra muy interesada por conocer esta técnicas avanzadas de programación, sin embargo **Juan** le pone los pies en el suelo, explicándole que son técnicas con muchas limitaciones y poca documentación en la que basarse.



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.



### Autoevaluación

#### La refactorización:

- ☐ Se utiliza como técnica complementaria de realización de pruebas.
- ☐ Genera un código más difícil de entender y de mantener.
- ☐ Utiliza una serie de patrones, de aplicación sobre el código fuente.
- ☐ Es una técnica de programación no recogida por los entornos de desarrollo.

### 1.3.- Patrones de refactorización más habituales.

## Caso práctico



A pesar de la poca documentación y lo novedoso de la técnica, **Juan** le enseña a **Ana** algunos de los patrones más habituales de refactorización, que vienen ya integrados en la mayoría de los entornos de desarrollos más extendidos en el mercado.

En el proceso de refactorización, se siguen una serie de patrones preestablecidos, los más comunes son los siguientes:

**Renombrado** (rename): este patrón nos indica que debemos cambiar el nombre de un paquete, clase, método o campo, por un nombre más significativo.

**Sustituir bloques de código por un método:** este patrón nos aconseja sustituir un bloque de código, por un método. De esta forma, cada vez que queramos acceder a ese bloque de código, bastaría con invocar al método.

**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.

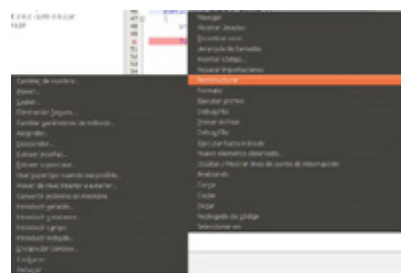
**Mover la clase:** si es necesario, se puede mover una clase de un paquete a otro, o de un proyecto a otro. La idea es no duplicar código que ya se haya generado. Esto impone la actualización en todo el código fuente de las referencias a la clase en su nueva localización.

**Borrado seguro:** se debe comprobar, que cuándo un elemento del código ya no es necesario, se han borrado todas la referencias a él que había en cualquier parte del proyecto.

**Cambiar los parámetros del proyecto:** nos permite añadir nuevos parámetros a 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.

**Mover del interior a otro nivel:** consiste en mover una clase interna a un nivel superior en la jerarquía.



## 1.4.- Analizadores de código.

### Caso práctico



**María** se va a centrar en el uso de analizadores de código con la ayuda de **Carlos**. **Carlos** no sabe lo que son los analizadores de código ni para qué sirven.

Cada **IDE** incluye herramientas de refactorización y analizadores de código. En el caso de software libre, existen analizadores de código que se pueden añadir como complementos a los entornos de desarrollo. Respecto a la refactorización, los **IDE** ofrecen asistentes que de forma automática y sencilla, ayudan a refactorizar el código.

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, pero sin que se modifique la semántica.

Los analizadores de código, son las herramientas encargadas de realizar esta labor. El analizador estático de código recibirá el código fuente de nuestro programa, lo procesará intentando averiguar la funcionalidad del mismo, y nos dará sugerencias, o nos mostrará posibles mejoras.

Los analizadores de código incluyen analizadores léxicos y sintácticos que procesan el código fuente y de un conjunto de reglas que se deben aplicar sobre determinadas estructuras. Si el analizador considera que nuestro código fuente tiene una estructura mejorable, nos lo indicará y también nos comunicará la mejora a realizar.

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 puede ser automático o manual. El automático, lo va a realizar un programa, que puede formar parte de la funcionalidad de un entorno de desarrollo, por ejemplo el **FindBugs** en Netbeans, o manual, cuando es una persona.

El análisis automático reduce la complejidad para detectar problemas de base en el código, ya que los busca siguiendo una serie de reglas predefinidas. El análisis manual, se centra en apartados de nuestra propia aplicación, como comprobar que la arquitectura de nuestro software es correcta.

Tomando como base el lenguaje de programación Java, nos encontramos en el mercado un conjunto de analizadores disponibles:

**PMD**. Esta herramienta 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.

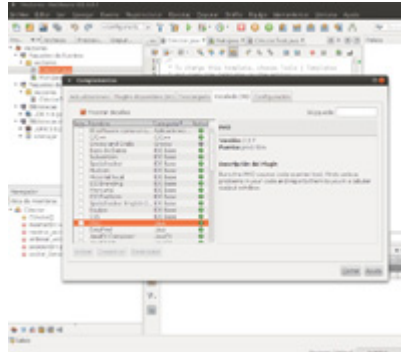
**CPD**. Forma parte del PMD. Su función es encontrar código duplicado.

### 1.4.1.- Uso.

---

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 6.9.1, vamos a instalar el plug-in para PMD. Para ello lo descargamos de sourceforge. Se descomprime el plug-in para Netbeans y se añade a los complementos de nuestro entorno de desarrollo. Para ello podemos utilizar el siguiente enlace:

[Descarga de PMD.](#)



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, pulsamos el botón derecho del ratón sobre el directorio que contiene los ficheros de código, en la vista de proyectos y elegimos: Herramientas - Ejecutar PMD.

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.

## 1.4.2.- Configuración.

La configuración de los analizadores de código, nos va a permitir definir nuestras propias reglas, añadiéndolas a las que el analizador nos proporciona por defecto.

En el caso del analizador de código PMD, lo primero que vamos a hacer, es conocer las reglas que son aplicables. Elegimos las reglas que son aplicables accediendo a Herramientas - Opciones - Varios - PMD

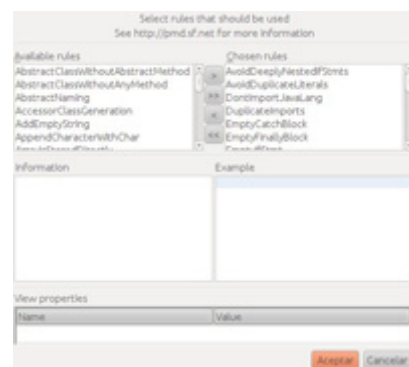
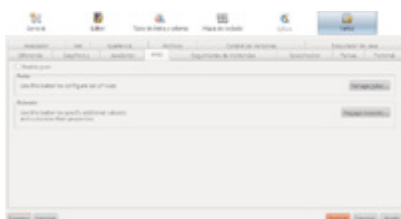
**Enable Scan** habilita el escaneo automático del proyecto, a intervalos regulares. Si pulsamos en **Manage Rules** nos iremos a un cuadro de diálogo que nos permitirá activar o desactivar reglas específicas.

Si seleccionamos el nombre de una regla, podemos ver una descripción del problema y un ejemplo de aplicación.

Si lo que queremos es introducir nuestras propias reglas, hacemos clic sobre **Manage Rulesets**. Se nos mostrará un nuevo formulario donde podemos importar un fichero **XML** o **JAR** con reglas.

Para crear ficheros XML con conjuntos de reglas, podemos obtener información en la siguiente página web.

[Creación de reglas para PMD, en inglés.](#)





## 1.5.- Refactorización y pruebas.

### Caso práctico



Juan ayudado por Ana, se ha dedicado a refactorizar algunas partes del código que había diseñado. Al modificar el código diseñado, se encuentra en la necesidad de realizar pruebas de nuevo.

En la actualidad, la refactorización y las pruebas, son dos aspectos del desarrollo de aplicaciones, que por sus implicaciones y su interrelación, se han convertido en conceptos de gran importancia para la industria. Muchas cuestiones y problemas siguen sin ser explorados en estos dos ámbitos. Uno de los enfoques actuales, que pretende integrar las pruebas y la refactorización, es el Desarrollo Guiado por Pruebas (TDD, Test Driven Development).

Con el Desarrollo Guiado por Pruebas (TDD), se propone agilizar el ciclo de escritura de código, y realización de pruebas de unidad. Cabe recordar, que el objetivo de las pruebas de unidad, es comprobar la calidad de un módulo desarrollado. Existen utilidades que permiten realizar esta labor, pudiendo ser personas distintas a las que los programan, quienes los realicen. Esto provoca cierta competencia entre los programadores de la unidad, y quienes tienen que realizar la prueba. El proceso de prueba, supone siempre un gasto de tiempo importante, ya que el programador realiza revisiones y depuraciones del mismo antes de enviarlo a prueba. Durante el proceso de pruebas hay que diseñar los casos de prueba y comprobar que la unidad realiza correctamente su función. Si se encuentran errores, éstos se documentan, y son enviados al programador para que lo subsane, con lo que debe volver a sumergirse en un código que ya había abandonado.



Con el Desarrollo Guiado por Pruebas, la propuesta que se hace es totalmente diferente. El programador realiza las pruebas de unidad en su propio código, e implementa esas pruebas antes de escribir el código a ser probado.

Cuando un programador recibe el requerimiento para implementar una parte del sistema, empieza por pensar el tipo de pruebas que va a tener que pasar la unidad que debe elaborar, para que sea correcta. Cuando ya tiene claro la prueba que debe de pasar, pasa a programar las pruebas que debe pasar el código que debe de programar, no la unidad en sí. Cuando se han implementado las pruebas, se comienza a implementar la unidad, con el objeto de poder pasar las pruebas que diseñó previamente.

Cuando el programador empieza a desarrollar el código que se le ha encomendado, va elaborando pequeñas versiones que puedan ser compiladas y pasen por alguna de las pruebas. Cuando se hace un cambio y vuelve a compilar también ejecuta las pruebas de unidad. Y trata de que su programa vaya pasando más y más pruebas hasta que no falle en ninguna, que es cuando lo considera listo para ser integrado con el resto del sistema.

Para realizar la refactorización siguiendo TDD, se refactoriza el código tan pronto como pasa las pruebas para eliminar la redundancia y hacerlo más claro. Existe el riesgo de que se cometan errores durante la tarea de refactorización, que se traduzcan en cambios de funcionalidad y, en definitiva, en que la unidad deje de pasar las pruebas. Tratándose de reescrituras puramente sintácticas, no es necesario correr ese riesgo: las decisiones deben ser tomadas por un humano, pero los detalles pueden quedar a cargo de un programa que los trate automáticamente.

## 1.6.- Herramientas de ayuda a la refactorización.

### Caso práctico

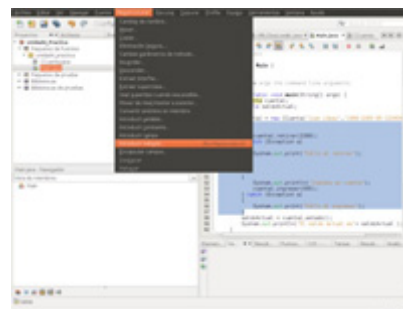


**Ana** ya conoce los principios y patrones básico de refactorización, ahora **Juan** le va a enseñar las herramientas de Netbeans para refactorizar de forma automática y sencilla, código Java.

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 Netbeans, la refactorización está integrada como una función más, de las utilidades que incorpora.

A continuación, vamos a usar los patrones más comunes de refactorización, usando las herramientas de ayuda del entorno.



**Renombrar.** Ya hemos indicado en puntos anteriores, que podemos cambiar el nombre de un paquete, clase, método o campo para darle un nombre más significativo. Netbeans nos permite hacerlo, de forma que actualizará todo el código fuente de nuestro proyecto donde se haga referencia al nombre modificado.

**Introducir método.** Con este patrón podemos seleccionar un conjunto de código, y reemplazarlo por un método.

**Encapsular campos.** Netbeans puede automáticamente generar métodos getter y setter para un campo, y opcionalmente actualizar todas las referencias al código para acceder al campo, usando los métodos getter y setter.



### Autoevaluación

¿Cuál no es un patrón de refactorización?

- ☐ Eliminar parámetros de un método.
- ☐ Renombrado.
- ☐ Sustitución de un bloque de sentencias por un método.
- ☐ Mover clase.

## 2.- Control de versiones.

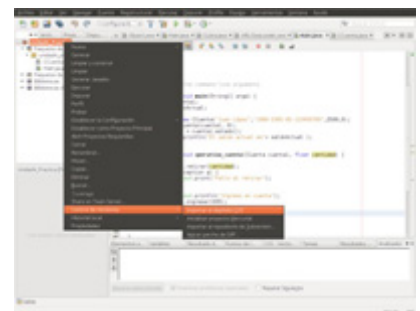
### Caso práctico



**Juan y María** están implementando la mayor parte de la aplicación. Continuamente está modificando el código generado, bien para mejorar algunos aspectos, o por qué han refactorizado. Hay diferentes versiones de una misma clase, de un mismo método. ¿Qué ocurre si se borra accidentalmente algún fichero? ¿Qué pasa si **María** modifica una clase que necesita **Juan**?

Con el término versión, se hace referencia a la evolución de un único elemento, o de cada elemento por separado, dentro de un sistema en desarrollo.

Siempre que se está realizando una labor, sea del tipo que sea, es importante saber en cada momento de qué estamos tratando, qué hemos realizado y qué nos queda por realizar. En el caso del desarrollo de software ocurre exactamente lo mismo. Cuando estamos desarrollando software, el código fuente está cambiando continuamente, siendo esta particularidad vital. Esto hace que en el desarrollo de software actual, sea de vital importancia que haya sistemas 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 desarrolladores trabajen en el mismo proyecto (incluso sobre los mismos archivos) de forma simultánea, sin que se pisen unos a otros. 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 una versión estable previa 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 sistemática u organizada.

En los entornos de desarrollo modernos, los sistemas de control de versiones son una parte fundamental, que van a permitir construir técnicas más sofisticadas como la Integración Continua.

En los proyectos Java, existen dos sistemas de control de versiones de código abierto, CVS y Subversion. La herramienta CVS es una herramienta de código abierto que es usada por gran cantidad de organizaciones. Subversion es el sucesor natural de CVS, ya que se adapta mejor que CVS a las modernas prácticas de desarrollo de software.

**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.- Estructura de herramientas de control de versiones.

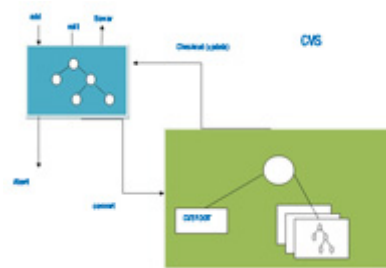
### Caso práctico



Para **Ana** el concepto de control de versiones le resulta muy abstracto. **Juan** va a intentar explicarle como funciona el mecanismo de control de versiones, tomando como ejemplo una herramienta que él conoce: CVS.

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 la 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 específico 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 añade 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.

Las órdenes que se pueden ejecutar son:

**checkout:** obtiene una 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.- Repositorio.

### Caso práctico



**Juan** le ha explicado a **Ana** los fundamentos del control de versiones, pero quiere profundizar más en el concepto de repositorio, ya que para él es la parte fundamental del control de versiones.

**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. En el siguiente enlace podemos ver acceder a un archivo que amplía la información sobre el repositorio CVS.

[Repositorio CVS](#)



### Autoevaluación

¿Qué afirmación sobre control de versiones es correcta?

- ☐ Solo puede existir una única versión de una clase.
- ☐ El almacenamiento de versiones es local a cada máquina.
- ☐ El repositorio centraliza el almacenamiento de los datos.



## 2.3.- Herramientas de control de versiones.

### Caso práctico



**María** quiere que **Carlos** conozca las herramientas de control de versiones que integra Netbeans, ya que es el Entorno que utilizan para su desarrollo. Dado que estamos utilizando un Entorno de Desarrollo Integrado, **Carlos** debe conocer las herramientas que incorpora Netbeans.

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 herramientas de control de versiones, se integran perfectamente en los entornos de desarrollo 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.



### Autoevaluación

¿Qué herramienta no es una herramienta de Control de Versiones?

- ☐ Subversion.
- ☐ CVS.
- ☐ Mercurial.
- ☐ PMD.



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

### Caso práctico

El equipo de desarrollo de BK Programación decide reunirse para planificar la gestión de configuraciones, ya que la aplicación de Gestión Hotelera es amplia y compleja, y continuamente se están diseñando nuevos módulos, clases o métodos.



**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.

La planificación de la Gestión de Configuraciones de Software va a comprender diferentes actividades:

1. Introducción (propósito, alcance, terminología).
2. Gestión de GCS (organización, responsabilidades, autoridades, políticas aplicables, directivas y procedimientos).
3. Actividades GCS (identificación de la configuración, control de configuración, etc.).
4. Agenda GCS (coordinación con otras actividades del proyecto).
5. Recursos GCS (herramientas, recursos físicos y humanos).
6. Mantenimiento de GCS.



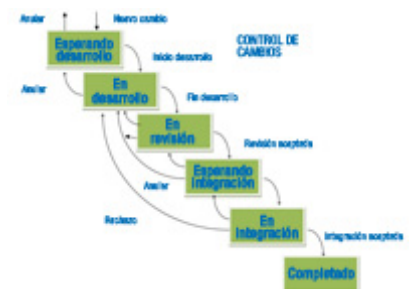
## 2.5.- Gestión del cambio.

### Caso práctico



Para gestionar el control de versiones de forma centralizada, **Ada** va a supervisar los cambios de versión que el equipo de **Juan** y de **María** están efectuando de forma independiente. **Ada** va a realizar una gestión del cambio centralizada y organizada.

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.



### Autoevaluación

¿Cuál de las siguientes no es una tarea básica de la Gestión de Configuraciones del Software?

- ☐ Control de cambios.
- ☐ Generación de informes.
- ☐ Auditorías de configuraciones.
- ☐ Gestión del repositorio.

## 2.6.- Gestión de versiones y entregas.

### Caso práctico



**María** ha desarrollado varias versiones del módulo de reservas de la aplicación de Gestión Hotelera. Le explica a **Carlos** como ha sido la evolución de cada versión del módulo, desde la primera versión, hasta la versión actual, que ella cree definitiva.

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 las 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.



## 2.7.- Herramientas CASE para la gestión de configuraciones.

### Caso práctico



Ada decide utilizar herramientas CASE, de código abierto, para la gestión de configuraciones. Decide utilizar Bugzilla, ya que puede integrarse con Netbeans.

Los procesos de gestión de configuraciones están estandarizados y requieren la aplicación de procedimientos predefinidos, ya que hay que gestionar gran cantidad de datos. Cuando se construye un sistema a partir de versiones de componentes, un error de gestión de configuraciones, puede implicar que el software no trabaje correctamente. Por todo ello, las herramientas CASE de apoyo son imprescindibles para la gestión de configuraciones.



Las herramientas se pueden combinar con entornos de trabajo de gestión de configuraciones. Hay dos tipos de entornos de trabajo de Gestión de Configuraciones:

**Entornos de trabajo abiertos:** las herramientas de Gestión de Configuraciones son integradas de acuerdo con procedimientos organizacionales estándar. Nos encontramos con bastantes herramientas de Gestión de Configuraciones comerciales y [open-source](#) disponibles para propósitos específicos. La gestión de cambios se puede llevar a cabo con herramientas de seguimiento (bug-tracking) como **Bugzilla**, la gestión de versiones a través de herramientas como **RCS** o **CVS**, y la construcción del sistema con herramientas como **Make** o **Imake**. Estas herramientas son open-source y están disponibles de forma gratuita.

**Entornos integrados:** estos entornos ofrecen facilidades integradas para gestión de versiones, construcción del sistema o seguimiento de los cambios. Por ejemplo, está el proceso de control de Cambios Unificado de **Rational**, que se basa en un entorno de Gestión de Configuraciones que incorpora **ClearCase** para la construcción y gestión de versiones del sistema y **ClearQuest** para el seguimiento de los cambios. Los entornos de Gestión de Configuraciones integrados, ofrecen la ventaja de un intercambio de datos sencillos, y el entorno ofrece una base de datos de Gestión de Configuraciones integrada.



### Autoevaluación

#### La Gestión de Configuraciones:

- ☐ Verifica la evolución de la implementación de una clase.
- ☐ Es una tarea que se puede gestionar con herramientas de Control de Versiones.
- ☐ Es un conjunto de actividades desarrolladas para gestionar los cambios a lo largo del ciclo de vida del software.

## 2.8.- Clientes de control de versiones integrados en el entorno de desarrollo.

### Caso práctico



**Juan** le va a enseñar a **Ana** los clientes de control de versiones que hay en Netbeans, para que aprenda a utilizarlos e integrarlos en los proyectos que realice de ahora en adelante.

Los entornos de desarrollo actuales integran de forma generalizada, clientes de control de versiones. En algunos casos, como puede ser el IDE Microsoft Visual Studio, se utiliza Visual Studio Team Foundation, de tal forma, que se realiza una gestión centralizada de un proyecto desarrollado por un equipo, incluyendo la gestión de configuraciones de software, la gestión de versiones, y demás aspectos de un desarrollo en equipo.

Los entornos de desarrollo Open-Source, como Eclipse y Netbeans, los integran de manera directa, o bien instalándolos como plug-in (complementos). Los clientes de control de versiones más destacados son:

CVS.  
Subversion.  
Mercurial.

### Debes conocer

Debes leer el siguiente documento para conocer las herramientas de control de versiones de Netbeans.

[Clientes de Control de Versiones en Netbeans](#)

También debes visitar el siguiente enlace donde se puede ver la guía de uso de subversión en Netbeans. Está en inglés pero es conveniente que le eches un vistazo para conocer cómo funciona Subversion.

[Guía de uso de Subversion en Netbeans.](#)



### Autoevaluación

¿Qué cliente de Gestión de Versiones no incorpora Netbeans?

- ☐ VS Team Foundation.
- ☐ CVS.
- ☐ Mercurial.

### 3.- Documentación.

#### Caso práctico



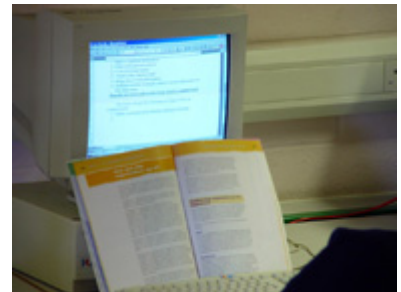
**Juan y María** saben de la importancia de tener documentado el código y todo el proceso de desarrollo de software. Al mismo tiempo que codifican y refactorizan, dejan constancia documental de lo que van haciendo. Como están desarrollando con Netbeans, y con el objetivo de facilitar su trabajo de documentación, van a utilizar JavaDoc. **Ana y Antonio** consiguen entender el funcionamiento de la aplicación, y la función de cada método, gracias a los comentarios que insertan en el código los dos programadores principales.

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.

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 lo 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.



#### Para saber más

El siguiente enlace nos muestra el estilo de programación a seguir en Java, así como la forma de documentar y realizar comentarios de un código. (En inglés)

[Documentación y comentarios en Java](#)

## 3.1.- Uso de comentarios.

### Caso práctico



**Carlos** está aprendiendo muchas cosas con **María**. Sin embargo hay algunos métodos y clases que ha implementado **María**, y que no logra entender. **María** le comenta que va a incluir comentarios en su código, y le va a enseñar la forma correcta de hacerlo.

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. 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.

```
/* Método para ingresar cantidades en la cuenta. Modifica el saldo.
 * Este método va a ser probado con JUnit
 */
// Comentario estilo javadoc
//
// @param cantidad
// @throws Exception
//
public void ingresar(double cantidad) throws Exception
{
    // el parámetro cantidad debe ser positivo. Comentario de una línea
    if (cantidad < 0)
        throw new Exception("No se puede ingresar una cantidad negativa");
    saldo = saldo + cantidad;
    // Este método realiza el ingreso de una cantidad de dinero
    // en la cuenta. Comentario multilínea
}
```

En el caso del lenguaje Java, C# y C, los comentarios, se implementan de forma similar. Cuando se trata de explicar la función de una sentencia, se usan los caracteres `//` seguidos del comentario, o con los caracteres `/*` y `*/`, situando el comentario entre ellos: `/* comentario */`

Otro tipo de comentarios que se utilizan en Java, son los que se utilizan para explicar qué hace un código, se denominan comentarios JavaDoc y se escriben empezando por `/**` y terminando con `*/`, estos comentarios pueden ocupar varias líneas. Este tipo de comentarios tienen que seguir una estructura prefijada.

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. 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.

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

## 3.2.- Alternativas.

### Caso práctico



**Juan** le explica a **María**, que para documentar el software, existen diferentes formas de hacerlo y distintas herramientas en el mercado que automatizan la tarea de documentación.

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

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.

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.



### Autoevaluación

#### Un comentario en formato JavaDoc.

- ☐ Utiliza los caracteres //
- ☐ Comienzan con /\* y termina por \*/
- ☐ Comienza por /\*\* y terminan por \*/



### 3.3.- Documentación de clases.

## Caso práctico



**Juan** va a utilizar JavaDoc para documentar las clases que ha desarrollado. **Ana** se da cuenta de la ayuda tan importante que ofrece, tanto para el futuro mantenimiento de la aplicación como para entender su funcionamiento, tener documentadas las clases.

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 transparente, en el diseño y documentación del código. Cuando se implementa una clase, se deben incluir comentarios. En el lenguaje Java, los criterios de documentación de clases, son los establecidos por JavaDoc.

Los comentarios de una clase deben comenzar con `/**` y terminar con `*/`. Entre la información que debe incluir un comentario de clase debe incluirse, al menos las etiquetas `@author` y `@version`, donde `@author` identifica el nombre del autor o autora de la clase y `@version`, la identificación de la versión y fecha.

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. Al menos se indican las etiquetas:

- @param:** seguido del nombre, se usa para indicar cada uno de los parámetros que tienen el constructor o método.
- @return:** si el método no es `void`, se indica lo que devuelve.
- @exception:** se indica el nombre de la excepción, especificando cuales pueden lanzarse.
- @throws:** se indica el nombre de la excepción, especificando las excepciones que pueden lanzarse.

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



### 3.4.- Herramientas.

## Caso práctico



Para documentar el código, el equipo de desarrollo de BK Programación, ha decidido utilizar JavaDoc, por lo que todos los componentes del equipo de desarrollo, debe familiarizarse con la herramienta.

Los entornos de programación que implementa Java, como Eclipse o Netbeans, 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 JavaDoc deben empezar por `/**` y terminar por `*/`.

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.



Los tags más habituales son los siguientes:

### Tags más habituales.

Tipo de tag	Formato	Descripción
Todos.	@see.	Permite crear una referencia a la documentación de otra clase o método.
Clases.	@version.	Comentario con datos indicativos del número de versión.
Clases.	@author.	Nombre del autor.
Clases.	@since.	Fecha desde la que está presente la clase.
Métodos.	@param.	Parámetros que recibe el método.
Métodos.	@return.	Significado del dato devuelto por el método
Métodos.	@throws.	Comentario sobre las excepciones que lanza.
Métodos.	@deprecated.	Indicación de que el método es obsoleto.

## 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 con **CVSR00T**. El IDE soporta diferentes formatos de **CVSR00T**, dependiendo de si el repositorio es local o remoto, y del método utilizado para conectarnos a él.

### Métodos de conexión a CVSR00T

Método	Descripción	Ejemplo
<b>pserver</b>	Contraseña de servidor remoto.	<code>:pserver:username@hostname:/repository_path</code>
<b>ext</b>	Acceso usando Remote Shell ( <u>RSH</u> ) o Secure Shell ( <u>SSH</u> ).	<code>:ext:username@hostname:/repository_path</code>
<b>local</b>	Acceso a un repositorio local.	<code>:local:/repository_path</code> (requiere un <u>CVS</u> externo ejecutable)
<b>fork</b>	Acceso a un repositorio local usando un protocolo remoto.	<code>:fork:/repository_path</code> (requiere un <u>CVS</u> 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 CVSR00T. 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.

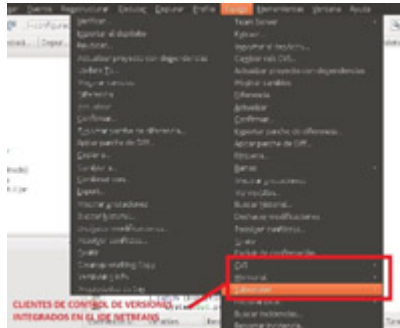


4. En el área de texto **Importar mensaje**, pondremos una descripción del proyecto que estamos importando.
5. Se especifica la localización del repositorio donde queremos importar el proyecto, escribiendo la ruta en **Carpeta del depósito**. De forma alternativa, se puede hacer clic en el botón **Examinar**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

### Licencias de recursos utilizados en la Unidad de Trabajo.

Recurso (1)	Datos del recurso (1)	Recurso (2)	Datos del recurso (2)
	Autoría: Oracle Corporation. Licencia: Copyright cita. Procedencia: Captura de pantalla de Netbeans.		Autoría: Oracle Corporation. Licencia: Copyright cita. Procedencia: Captura de pantalla de Netbeans.
	Autoría: Oracle Corporation. Licencia: Copyright cita. Procedencia: Captura de pantalla de Netbeans.		Autoría: Oracle Corporation. Licencia: Copyright cita. Procedencia: Captura de pantalla de Netbeans.
	Autoría: Oracle Corporation. Licencia: Copyright cita. Procedencia: Captura de pantalla de Netbeans.		Autoría: Oracle Corporation. Licencia: Copyright cita. Procedencia: Captura de pantalla de Netbeans.
	Autoría: Oracle Corporation. Licencia: Copyright cita. Procedencia: Captura de pantalla de Netbeans.		Autoría: Oracle Corporation. Licencia: Copyright cita. Procedencia: Captura de pantalla de Netbeans.
	Autoría: Gnome.org. Licencia: GNU. Procedencia: Captura de pantalla de gnome.org.		Autoría: Oracle Corporation. Licencia: Copyright cita. Procedencia: Captura de pantalla de Netbeans.
	Autoría: Oracle Corporation. Licencia: Copyright cita. Procedencia: Captura de pantalla de Netbeans.		Autoría: Oracle Corporation. Licencia: Copyright cita. Procedencia: Captura de pantalla de Netbeans.
	Autoría: Oracle Corporation. Licencia: Copyright cita. Procedencia: Captura de pantalla de Netbeans.		Autoría: Oracle Corporation. Licencia: Copyright cita. Procedencia: Captura de pantalla de Netbeans.

	<p>Autoría: Oracle Corporation. Licencia: Copyright cita. Procedencia: Captura de pantalla de Netbeans.</p>
--	---