4. Optimización y documentación

Autor	Xerach Casanova
Clase	Entornos de desarrollo
Fecha	@Jan 22, 2021 7:20 AM

1. Optimización y documentación

La refactorización es una técnica que consiste en hacer pequeñas transformaciones en el código del programa para mejorar la estructura sin que cambie el comportamiento o funcionalidad con el objetivo de mejorar estructura, legibilidad o eficiencia.

De esta manera hacemos que el mantenimiento del software sea más sencillo y que el programa sea más rápido. Se basa en el concepto matemático de factorización de polinomios, así que (x+1)(x-1) se puede expresar con x2-1 sin que se altere el sentido.

Pistas para refactorizar:

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

1.1. Concepto

El concepto de refactorización se define como el cambio hecho en la estructura interna del software para hacerlo más fácil de entender y modificar sin alterar su comportamiento.

Se aconseja crear métodos getter y setter para cada campo que se defina en una clase para acceder o modificar su valor.

La refactorización y la optimización son conceptos distintos y la diferencia radica en que cuando se optimiza, se persigue una mejora de rendimiento y velocidad de ejecución, aunque el código resulte más complicado de entender.

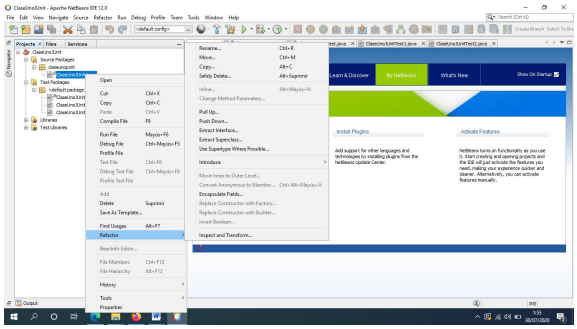
1.2. Limitaciones

La refactorización presenta problemas en algunos aspectos de su desarrollo al ser una técnica novedosa.

- Uno de estos problemas son las bases de datos, las cuales son difíciles de modificar al ser demasiado interdependientes. Cualquier modificación a las bases de datos (esquema, migración), suele ser tarea costosa, por ello, la refactorización en aplicaciones con bases de datos está limitada al diseño de la base de datos.
- Otro problema es el cambio de interfaces. Cuando modificamos la estructura interna de un programa o un método, no afecta al comportamiento de la interfaz, sin embargo, si renombramos un método genera un problema si la interfaz es pública y otros programas llaman a ese método. La solución pasa por mantener la interfaz vieja junta a la nueva.
- Es difícil refactorizar cuando existen errores de diseño o cuando la estructura a modificar es de vital importancia en el diseño de la aplicación.
- En los casos en los que no se debe refactorizar en absoluto es cuando es más fácil reescribirlo desde el principio que modificarlo.

1.3. Patrones de refactorizaciones más habituales

Los patrones más habituales de refactorizaciones ya vienen integrados en la mayoría de IDE. Estos son:



- **Renombrar:** Cambia el nombre de un paquete, clase, método o campo.
- **Encapsular campos.** Crear métodos getter y setter para los campos de clase.
- **Sustituir bloques de código por un método.** En ocasiones se observa que, al aparecer repetido en distintos sitios, podemos sustituir el bloque de código por un método, de manera que, cada vez que usemos ese bloque de código invoquemos al método.
- **Modificar la extensión del código**. Hacer un código más extenso si se gana en claridad o menos extenso solo si se gana eficiencia.
- **Crear código común** en una clase o método para evitar el código repetido.

- **Mover la clase de** un paquete a otro o de un proyecto a otro. Implica actualización en todo el código fuente de las referencias a la clase en su nueva ubicación.
- **Borrado seguro.** Garantizar que un elemento del código ya no es necesario, se borran todas las referencias a él en cualquier parte del proyecto.
- **Cambiar parámetros del método,** permite añadir, modificar o eliminar los parámetros de un método y cambiar modificadores de acceso.
- **Extraer la interfaz.** Crea una nueva interfaz de los métodos public non-static seleccionados en una clase o interfaz.

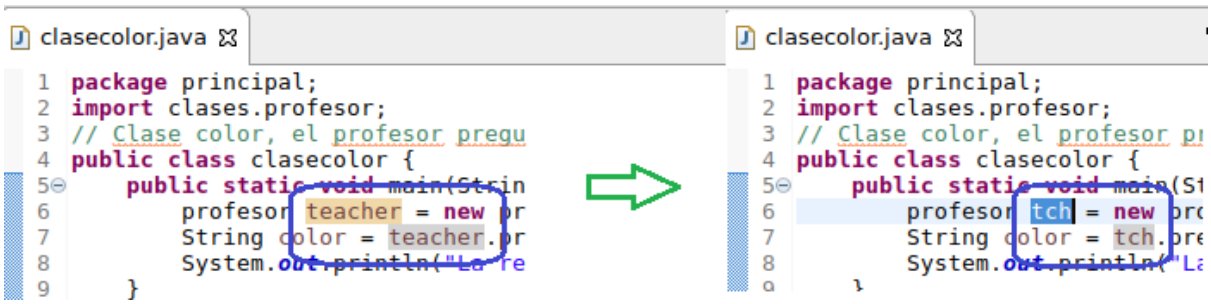
1.4. Refactorización en eclipse

Las herramientas de refactorización en los entorno de desarrollo ayudan a automatizar la refactorización, ahorrando tiempo y evitando redundancias o errores en el código que modificamos.

En eclipse podemos refactorizar haciendo click secundario en un fragmento de código y escogiendo la opción refactorizar. Según el código seleccionado, se mostrarán unas opciones u otras.

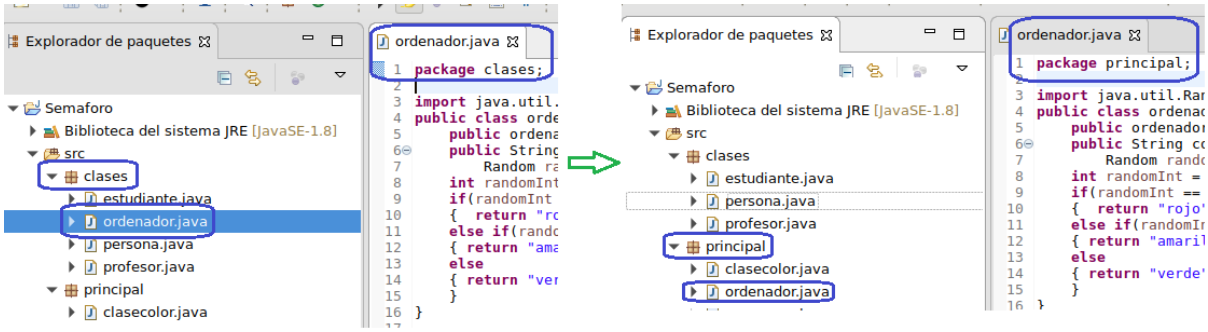
1.4.1. Renombrar

Modificar el nombre de cualquier elemento: variable, atributo, método o clase. Los cambios se realizan también sobre las referencias que haya a dicho elemento en todo el proyecto. Por ejemplo:



1.4.2. Mover

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

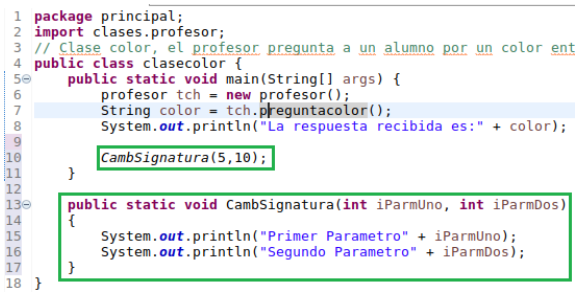


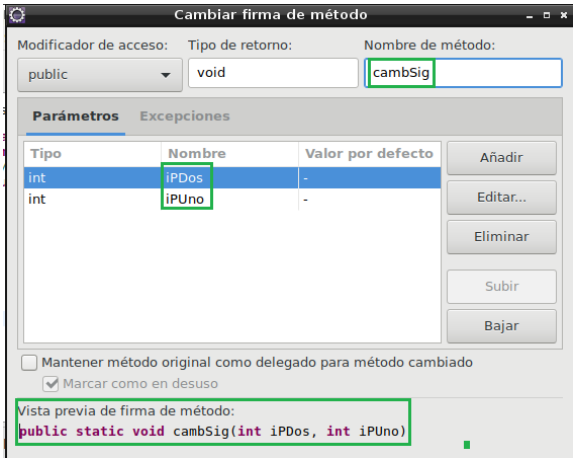
1.4.3. Cambiar signatura del método

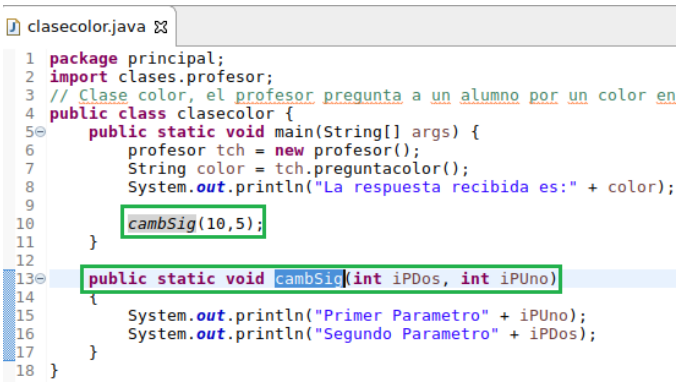
Modificar la firma o cabecera del método. Si este ya ha sido usado, cambiar el número o tipo de parámetros, o el tipo devuelto provoca fallos de compilación.

Es útil para cambiar el nombre de los parámetros o su orden (se modificará también el orden en todas las llamadas al método).

Ejemplo: Cambiar el nombre del método y los parámetros:

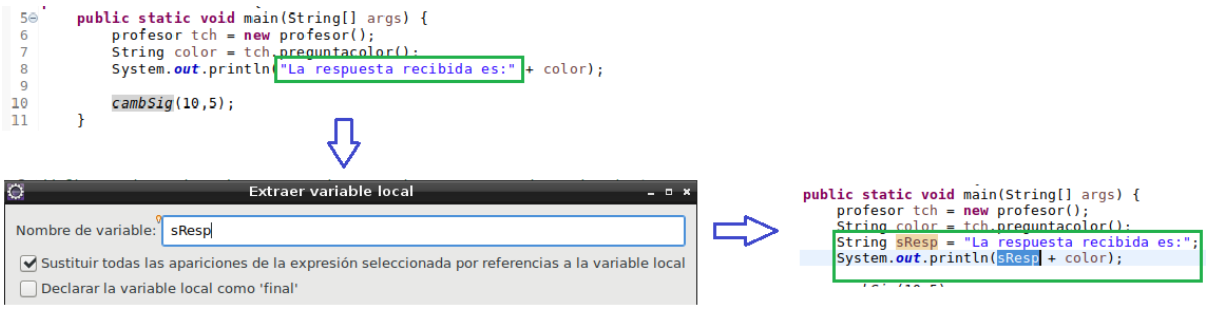






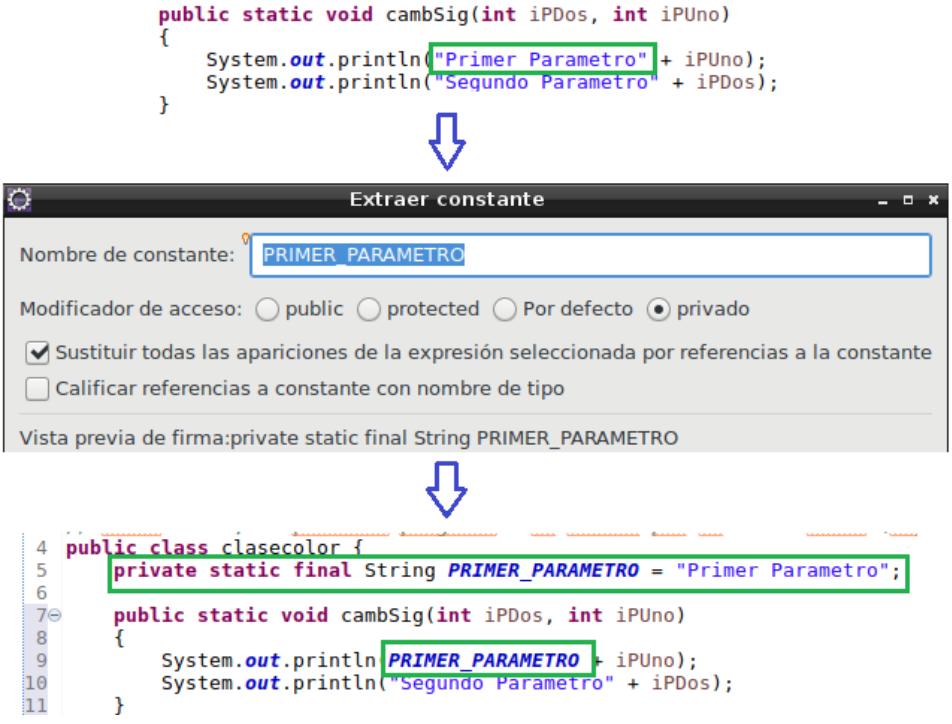
1.4.4. Extraer variable local

Crear una variable local inicializada con el valor de un literal. Las referencias a esa expresión se modifican por una referencia a la variable.



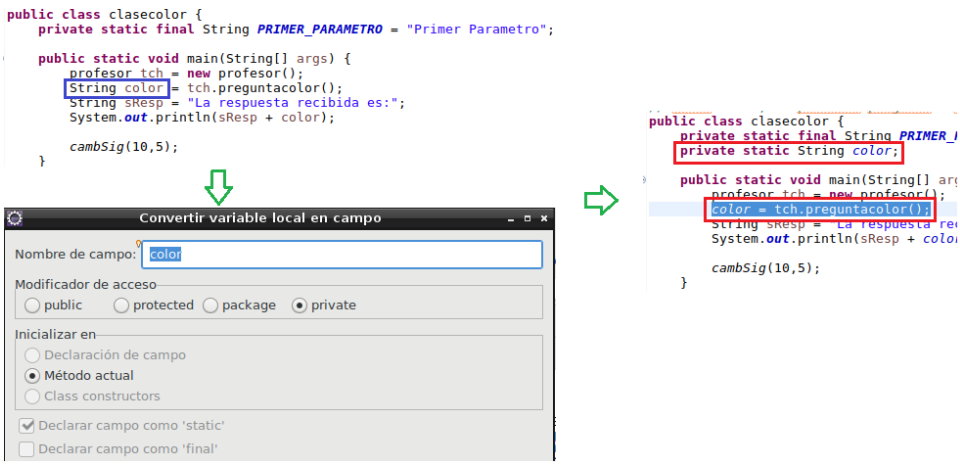
1.4.3. Extraer constante

Genera una constante a partir de la expresión seleccionada.



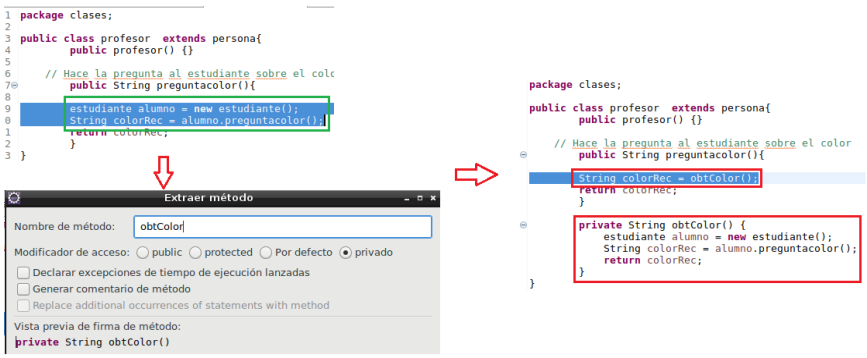
1.4.6. Convertir variable local en atributo

A veces se definen variables locales dentro de un método y luego decidimos usarlas como variables de clase.



1.4.7. Extraer método

Convierte el código seleccionado en un método, útil para reutilizarlo en varios sitios del programa.



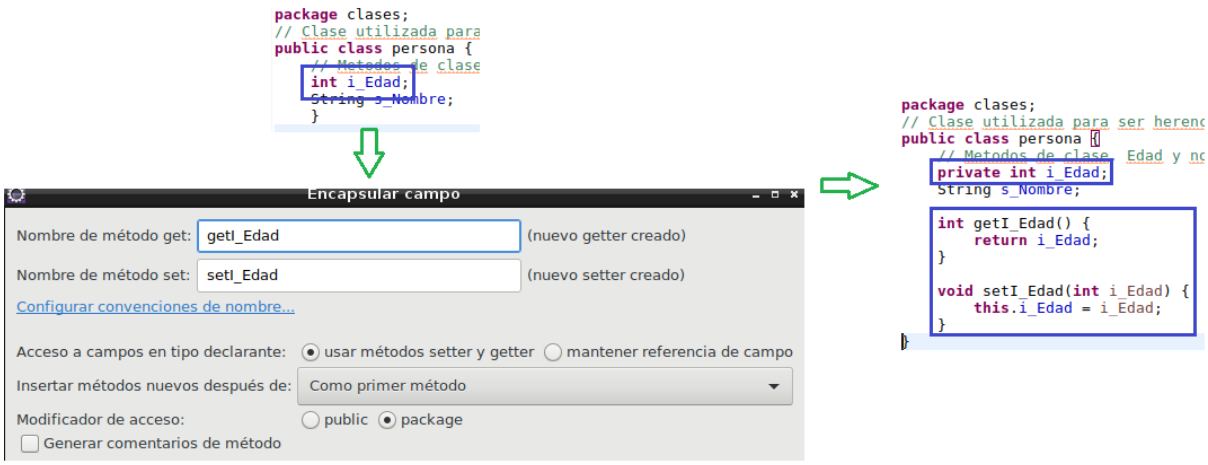
1.4.8. Incorporar

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 los lugares donde se referenciaban.

Útil si la variable, constante solo se utiliza una vez y no merece la pena almacenar su valor, o cuando el método solo se utiliza una o dos veces y no merece la pena aislarlo.

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.



2. Control de versiones

Cuando se desarrolla un software, el código fuente cambia constantemente, ya sea por el propio desarrollo o por el mantenimiento. Además en ocasiones se desarrollan por fases o entregas a cliente. Se hace necesario por tanto un sistema de control de versiones.

Un sistema de control de versiones bien diseñado facilita el desarrollo, permitiendo que varios programadores trabajen en el mismo proyecto e incluso sobre los mismos archivos de manera simultánea, gestionando los conflictos que puedan surgir de actualizaciones simultáneas sobre el mismo código. Las herramientas de control de versiones proveen de un sitio central donde se almacena el código de la aplicación y un historial de cambios realizado a lo largo de la vida del proyecto. También se permite volver a versiones estables previas si es necesario.

Una versión 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 para ello se debe disponer de un método par a designar las diferentes versiones de manera organizada.

Las alternativas de código abierto son: GIT, CVS, Subversion, Mercurial...

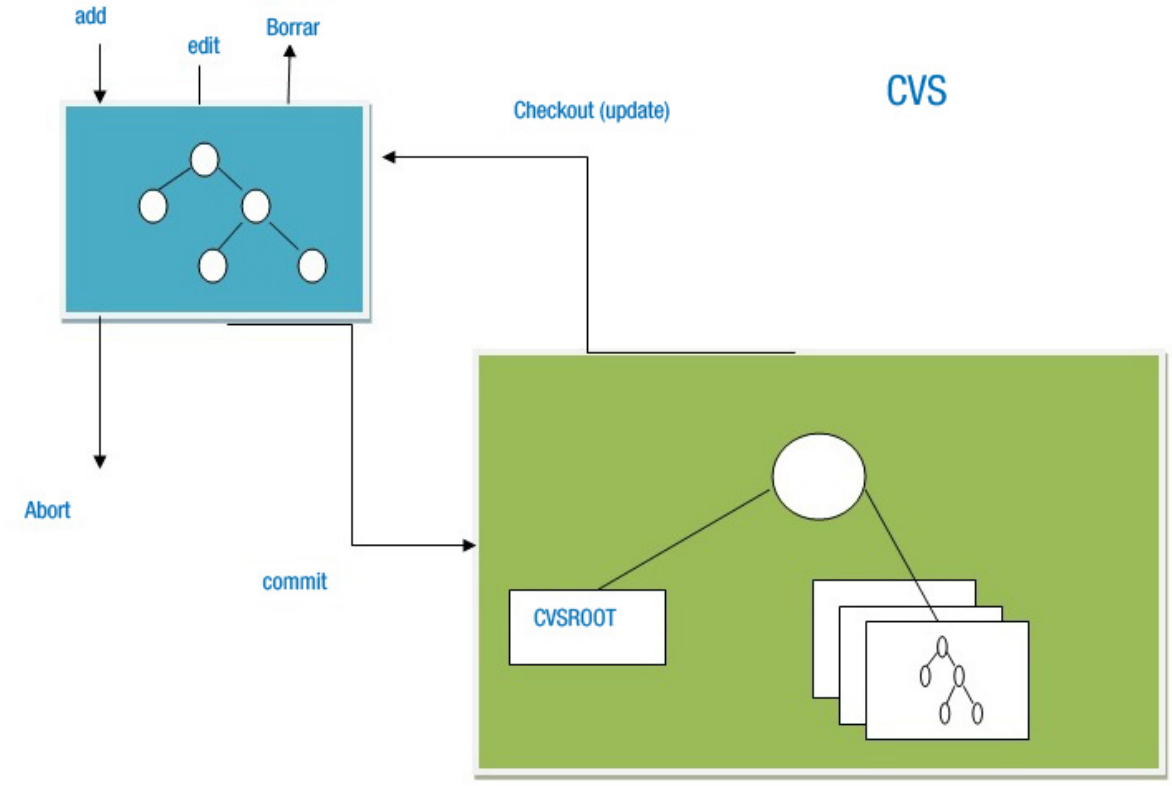
2.1. Tipos de herramientas de control de versiones

Se pueden clasificar las herramientas de control de versiones como:

- **Sistemas locales.** Control de versiones que se guarda en diferentes directorios en función de sus versiones. La gestión recae sobre el responsable del proyecto y no dispone de herramientas de automatización. Viable para pequeños proyectos de un único programador.
- **Sistemas centralizados:** arquitectura cliente - servidor. Un equipo contiene todos los archivos y sus diferentes versiones. los clientes replican la información en sus entornos de trabajo locales.
- **Sistemas distribuidos.** Cada sistema hace una copia completa de los ficheros de trabajo y de todas sus versiones. Todos los equipos tienen un rol de igual a igual y los cambios se pueden sincronizar entre cada par de copias disponibles. Habitualmente funcionan siendo uno el repositorio principal y el resto asumiendo papel de clientes sincronizando sus cambios a este.

2.2. Estructura de herramientas de control de versiones

Suelen estar formadas por un conjunto de elementos sobre los que se pueden ejecutar órdenes e intercambiar datos entre ellos. (Estudiamos la herramienta CVS).



Una herramienta de control de versiones es un sistema de mantenimiento de código fuente extraordinariamente útil para grupos de desarrolladores que trabajan cooperativamente usando alguna clase de red.

CVS permite trabajar y modificar ficheros organizados en proyectos. Dos personas pueden modificar un fichero sin perder el trabajo de ninguna.

CVS utiliza arquitectura cliente-servidor. El servidor guarda la versión actual y su historia. Los clientes conectan al servidor y sacan una copia completa del proyecto, trabajan en ella e ingresan sus cambios. El servidor utilizado es similar a UNIX, pero los clientes pueden funcionar en cualquier S.O.

Los clientes pueden comparar versiones diferentes de ficheros, solicitar historia completa de cambios o sacar foto histórica del proyecto en una fecha determinada o en un número de revisión determinado.

Muchos proyectos de código abierto permiten acceso de lectura anónima, de manera que se pueden descargar una copia del proyecto y solo necesitan contraseña para ingresar cambios.

También se utiliza el comando de actualización para tener sus copias al día de la última versión en servidor.

El sistema de control de versiones está formado por:

- **Repositiorio:** Lugar de almacenamiento de los datos de los proyectos. Suele ser un directorio en algún ordenador.
- **Módulo:** directorio específico del repositorio, puede identificar una parte del proyecto o el proyecto completo.
- **Revisión:** Cada una de las versiones parciales o cambios en los archivos o repositorio completo. La evolución del sistema se mide en revisiones y cada cambio se considera incremental.
- **Etiqueta:** información textual que se añade a un conjunto de archivos o a un módulo completo para aportar información importante.
- **Rama:** revisiones paralelas de un módulo para efectuar cambios sin tocar la evolución principal. Se emplea para pruebas o para obtener cambios en versiones antiguas.

Algunos servicios que se proporcionan son:

- **Creación de repositorios.** Esqueleto de un repositorio sin información inicial del proyecto.

- **Clonación de repositorios.** Crea un nuevo repositorio y vuelca la información de algún otro repositorio existente (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 local.** Actualiza los cambios realizados en la copia local en el repositorio principal.
- **Gestión de conflictos.** Si los cambios que se desean realizar en el repositorio principal entran en conflicto con otros cambios que se han subido por otro desarrollador, se trata de combinar automáticamente todo los cambios y, si no es posible por pérdida de información, se muestran al programador los conflictos para que se tome una decisión de como combinarlos.
- **Gestión de ramas.** Creación, eliminación, integración de diferencias entre ramas y selección de ramas de trabajo.
- **Información sobre registro de actualizaciones.**
- **Comparativa de versiones.** Genera información sobre las diferencias entre versiones del proyecto.

Las órdenes a ejecutar son:

- **chekout:** obtiene una copia del trabajo para 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 un almacén general de versiones, suele ser un directorio en la mayoría de herramientas.

El repositorio centraliza todos los componentes de un mismo sistema, incluyendo las distintas versiones de cada componente. Se ahorra espacio de almacenamiento ya que evitamos guardar por duplicado elementos comunes a varias versiones. Nos facilita el almacenaje de la info y la evolución del sistema.

Netbeans utiliza el control de versiones CVS, el cual tiene un componente principal: el repositorio, en el cual se deben almacenar todos los ficheros de los proyectos que puedan ser accedidos por varios desarrolladores.

Cuando usamos un sistema de control de versiones se trabaja de forma local y sincronizamos con el repositorio haciendo los cambios en nuestra copia local. Realizando el cambios se realiza también en el repositorio. En Netbeans se realiza de varias formas:

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

Teniendo un proyecto CVS versionado, podemos abrirlo en el IDE y acceder a las características del versionado. El IDE escanea nuestros proyectos abiertos y si contienen directorios CVS, se activan automáticamente el estado del archivo y la ayuda contextual para 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. Puede representarse en forma de grafo, donde los nodos son las versiones y los arcos son la creación de una versión a otra parte ya existente.

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

Variantes: existen varias versiones del componente, el grafo no es secuencia lineal y adopta forma de árbol. La numeración requiere varios niveles. El primer número designa la variante o línea de evolución, el segundo la revisión a lo largo de dicha variante.

La terminología para referirse a los elementos del grafo son:

- Tronco (trunk). Variante principal
- Cabeza (head). 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 es necesario aplicar un mismo cambio a varias variantes.

Fusión de variantes: Se puede fundir una rama con otra (Merge)

Técnicas de almacenamiento: las distintas versiones tienen en común parte del contenido. Para aprovechar el espacio se utilizan distintas técnicas:

- **Deltas directos:** se almacena la primera versión completa y luego los cambios mínimos 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 deltas directos.
- **Marcado selectivo:** se almacena el texto refundido de todas las versiones como secuencia lineal, marcando cada sección del conjunto con los números de versiones correspondiente.

Una entrega es una instancia de un sistema que se distribute a los usuarios externos al equipo de desarrollador.

La planificación de la entrega se ocupa de cuando emitir una versión del sistema como una entrega. La entrega es un conjunto de programas ejecutables, archivos de configuración, que definen como se configura la entrega para la instalación particular.

2.3. Herramientas de control de verisones

Además de CVS y subversion (sucesor natural de CVS), existen otras herramientas de amplia difusión:

- **SourceSafe:** forma parte de Microsoft Visual Studio.
- **Visual Studio Team Foundation Server:** es el sustituto de Sourse Safe, ofrece control de código fuente, recolección de datos, informes, seguimiento de proyectos... destinado a proyectos de colaboración.
- **Darcs:** Sistema de gestión de versiones distribuido: posibilidad de hacer commits locales (sin conexión), cada repositorio es una rama en sí misma, independencia de servidor central, posibilidad de renombrar ficheros, varios métodos de acceso como local: ssh, http y ftp, etc.

- **Git:** herramienta de control de versiones, diseñada por Linus Torvalds
- **Mercurial:** funciona en linux, windows y mac os x, programa en línea de comandos, permite desarrollo distribuido, capacidades avanzadas de ramificación e integración.

2.4. Planificación de la gestión de configuraciones

La gestión de configuraciones es un conjunto de actividades para gestionar los cambios a lo largo del ciclo de vida y está regulado por el estandar IEEE 828.

Una configuración es una combinación de versiones particulares de los componentes que forman un sistema consciente. 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 cambia porque se añaden o modifican elementos, o debido a la reorganización de los componentes sin que estos cambien.

La gestión de configuraciones de software compone cuatro tareas básicas:

- **Identificación:** se trata de establecer estándares de documentación y esquema de identificación de documentos.
- **Control de cambios:** evaluación y registro de todos los cambios que se hagan a la configuración software.
- **Auditoría de configuraciones:** garantizan que el cambio se haga correctamente.

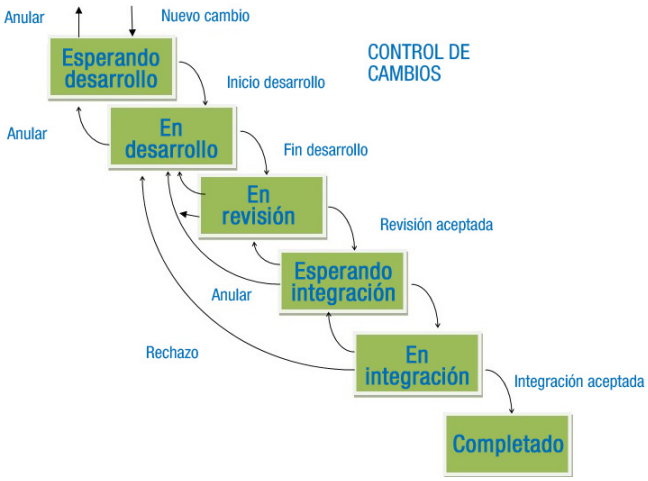
2.5. Gestión del cambio

Las herramientas de control de versiones no garantizan desarrollos razonables. Para garantizar que siempre disponemos de una línea base para continuar el desarrollo hay que aplicar controles al desarrollo e integración de los cambios. Se pueden establecer:

- **Control individual:** antes de aprobarse un nuevo elemento. El programador cambia la documentación cuando se requiere, el cambio se registra de manera individual pero no genera un documento formal.
- **Control de gestión organizado,** conduce a la aprobación de un nuevo elemento e implica procedimiento de revisión y aprobación para cada cambio propuesto en la configuración. Ocurre también durante el proceso

de desarrollo pero es usado después de haber sido aprobado en configuración de software.

- **Control formal,** durante el mantenimiento, el impacto de cada tarea se evalúa por un comité de control de cambios, que aprueba las modificaciones de configuración de software.



3. Documentación

El proceso de documentación de código es uno de los aspectos más importantes de un programador. El desarrollo rápido de aplicaciones va en detrimento de una buena documentación.

```
/* Método para ingresar cantidades en la cuenta. Modifica el saldo.
 * Este método va a ser probado con Junit
 */
/**      Comentarios 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
        la cuenta
     */                Comentario multilínea
}
```

Es fundamental para la detección de errores y para el mantenimiento posterior. Explica lo que no resulta evidente, cual es la finalidad de una clase, un paquete, un método o una variable, que se espera del uso de una variable, que algoritmo se usa, etc...

3.1. Uso de comentarios

Los comentarios tienen dos propósitos distintos:

- Explicar el objetivo de las sentencias, de manera que se sepa en todo momento la función de esa sentencia.
- Explicar qué realiza un método o clase, no cómo lo realiza. Se trata de explicar los valores que devolverá un método.

En el caso de comentarios de una sola línea, en Java, C# y C se utilian los caracteres // y en el caso de comentarios multilínea se utiliza /* */.

Es conveniente poner comentarios al principio de un fragmento de código, a lo largo de bucles o si hay alguna línea que no resulta evidente y puede llevar a confusión.

Cuando se modifica código, también se deben modificar los comentarios.

3.2. Documentación de clases

Existen distintas herramientas para automatizar, completar y enriquecer la documentación: JavaDoc, SchemeSpy, Doxygen, estos dos últimos incluyen modelos de bases de datos y diagramas.

Las clases que implementan una aplicación deben incluir comentarios, Debemos seguir una serie de pautas cuando implementamos una clase, para documentarla.

En los IDE de java, se incluye una herramienta que genera HTML de documentación a partir de los comentarios del código. Este es JavaDoc. Para ello se siguen una serie de normas:

- Los comentarios se hacen al principio de cada clase, cada método y cada variable de clase. Se escriben empezando por /** y terminando con */
- Los comentarios son a nivel de clase, nivel variable y nivel método.
- Se genera para métodos public y protected.
- Se pueden usar tags para documentar diferentes aspectos del código como los parámetros.

Las tags se añaden de forma automática, como la de @author y @version también se suele añadir @see para referenciar a otras clases y métodos.

Estas son 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* **@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.