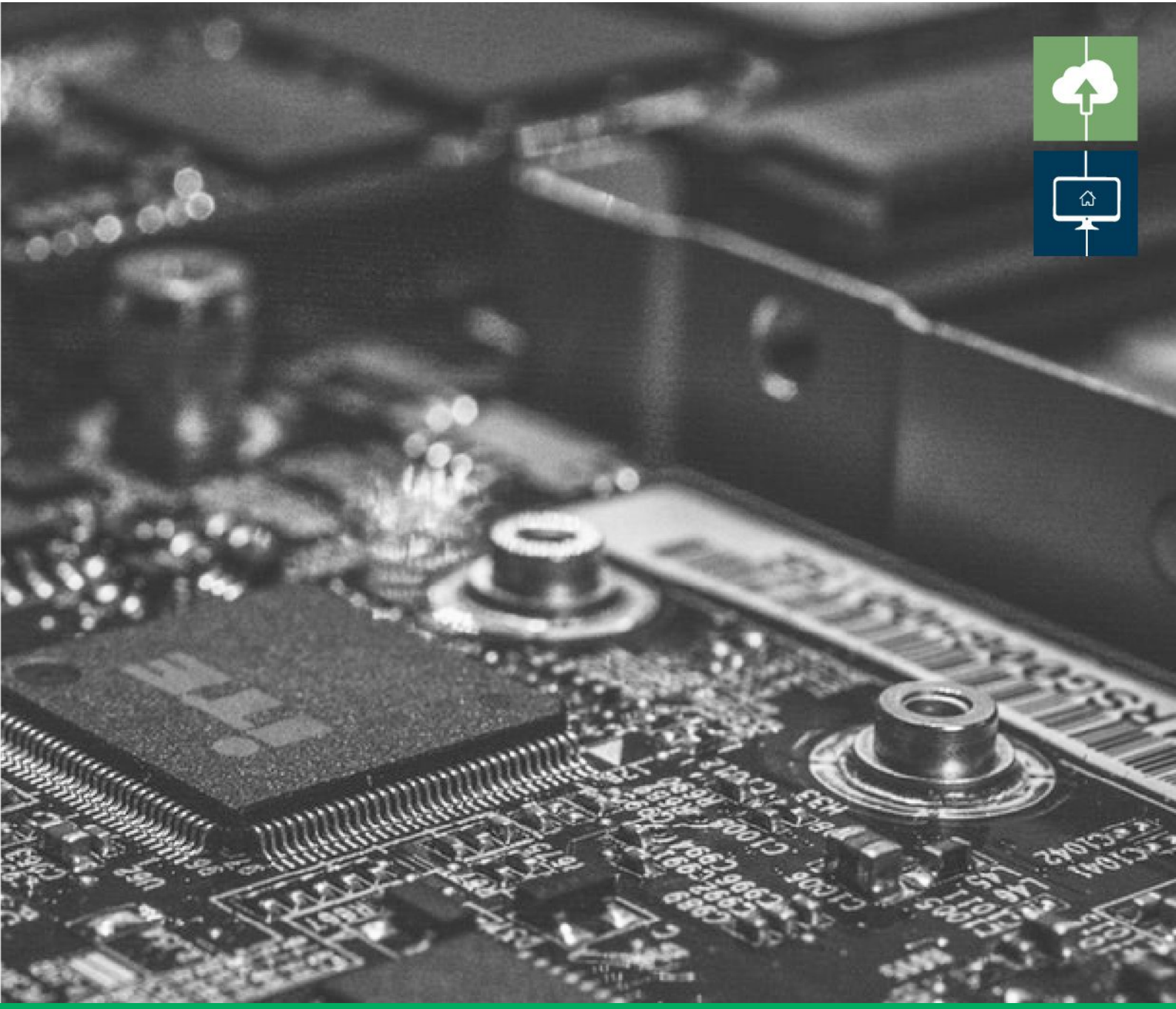


Técnico Superior en Desarrollo de Aplicaciones Multiplataforma

MP. ENTORNOS DE DESARROLLO

UD 4 - Refactorización y Documentación Desarrollo Software

Universidad Francisco de Vitoria
UFV Madrid
Ciclos Formativos de Grado Superior - CETYS

ÍNDICE

1. Refactorización
2. Control de versiones.
3. Control de versiones.

1. Refactorización

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.

Ejemplo

FACTORIZACIÓN DE POLINOMIOS
$X^2 - 1 = (X + 1)(X - 1)$

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:

a. **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.

b. **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.

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 qué cosas han cambiado.

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

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 otra 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.

Patrones de refactorización más habituales.

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

1. **Renombrado** (rename): Este patrón nos indica que debemos cambiar el nombre de un paquete, clase, método o campo, por un nombre más significativo.
2. **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.
3. **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.
4. **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 las 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.

Analizadores de código.

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, los 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.

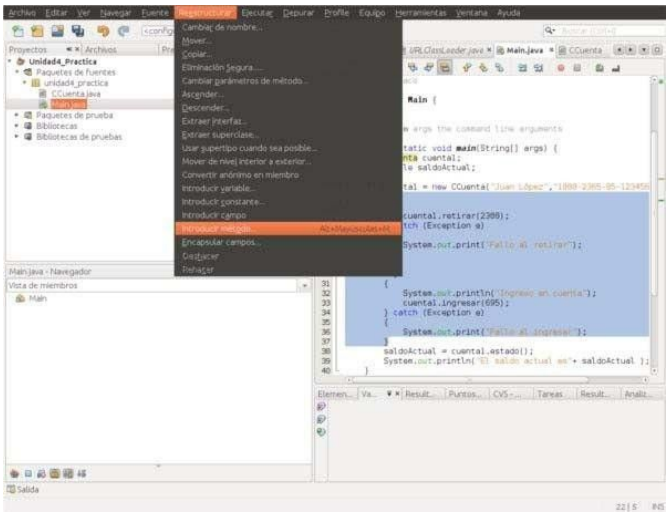
Herramientas de ayuda a la refactorización.

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.



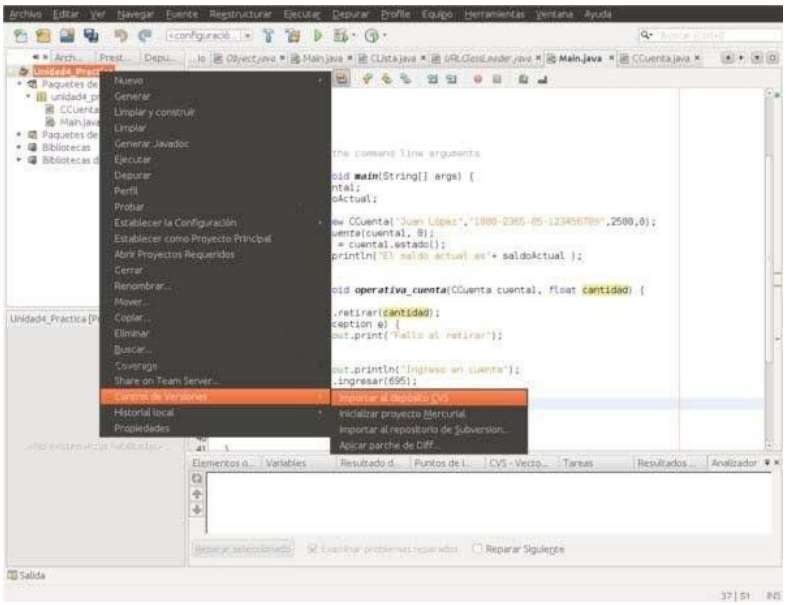
Introducir método. Con este patrón podemos seleccionar un conjunto de código y reemplazarlo por un método.

Encapsular campo. Para un campo, puede automáticamente generar métodos getter y setter para acceder al mismo. Opcionalmente actualizar todas las referencias al código para acceder al campo, usando los métodos getter y setter.

2. Control de versiones.

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 que 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 mismo archivos) de forma simultánea, sin que se que 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 un versión estable previa del código fuente si es necesario.

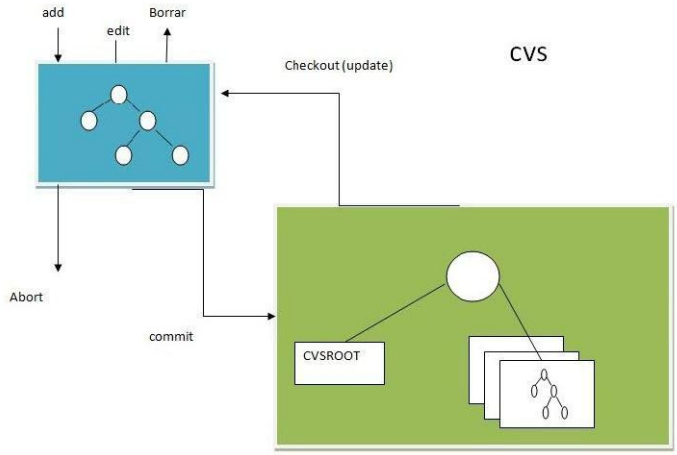
Una versión, desde el punto de vista de la evolución, se define como la forma particular de un objeto en un instante o contexto dado. Se denomina revisión, cuando se refiere a la evolución en el tiempo. Pueden coexistir varias versiones alternativas en un instante dado y hay que disponer de un método, 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.

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

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.

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.

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

Identificación. Se trata de establecer estándares de documentación y un esquema de identificación de documentos.
Control de cambios. Consiste en la evaluación y registro de todos los cambios que se hagan de la configuración software.
Auditorías de configuraciones. Sirven, junto con las revisiones técnicas formales para garantizar que el cambio se ha implementado correctamente.
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.

3. Control de versiones.

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 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 cual es la finalidad de un clase, de un paquete, qué hace un método, para que sirve una variable, qué se espera del uso de una variable, qué algoritmo se usa, por qué hemos implementado de una manera y de otro, qué se podría mejorar en el futuro, etc.

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. 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 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.

Alternativas.

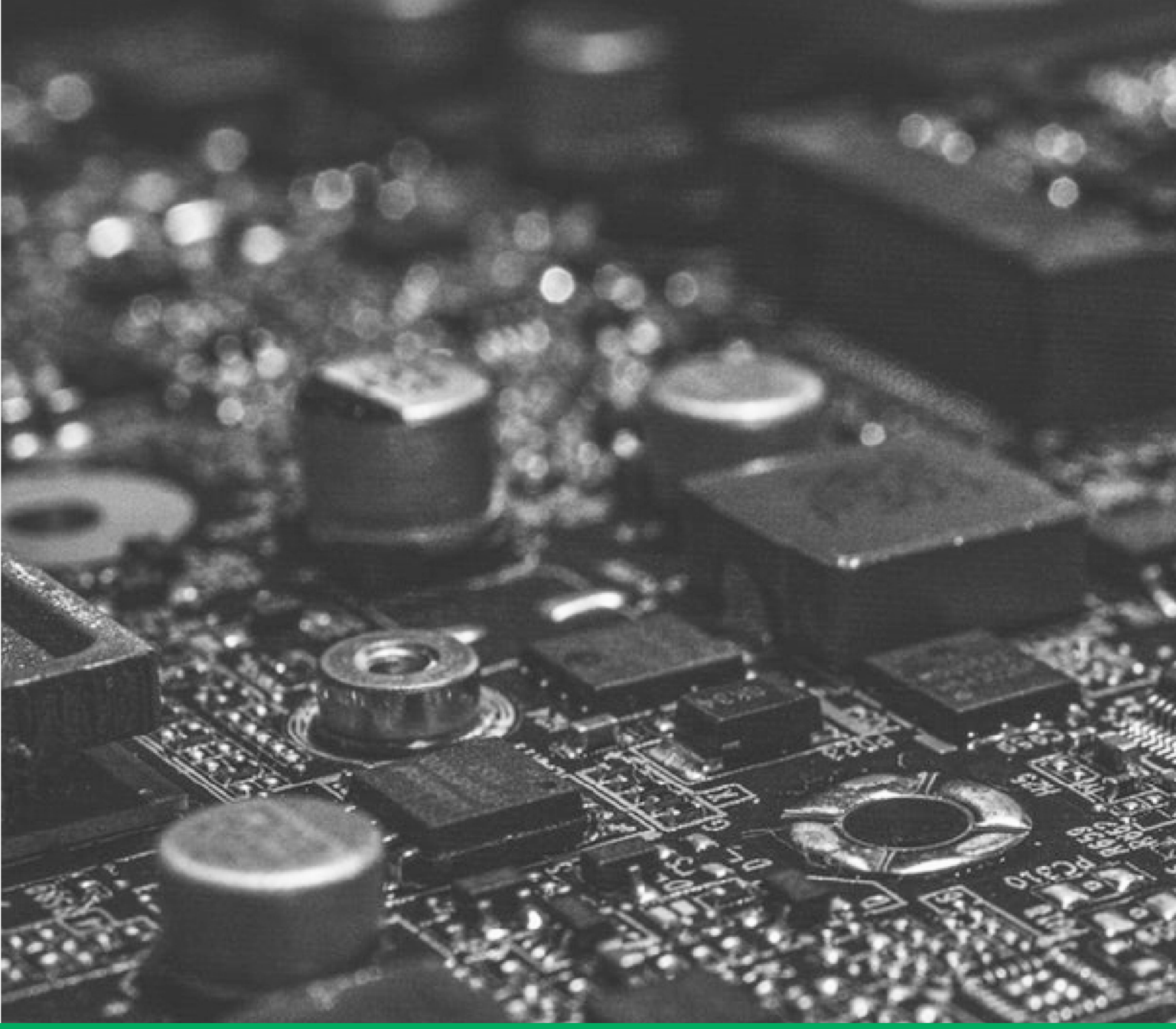
En la actualidad, el desarrollo rápido de aplicaciones, en muchos casos, va en detrimento de una buena documentación del código. Si el código no está documentado, puede resultar bastante difícil de entender, y por tanto de solucionar errores y de mantenerlo.

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



DAM

Universidad
Francisco de Vitoria
UFV Madrid
Ciclos Formativos de Grado Superior - CETYS