

UNIDAD 4: OPTIMIZACIÓN Y DOCUMENTACIÓN

Módulo Profesional: Entornos de Desarrollo

ÍNDICE

RESUMEN INTRODUCTORIO.....	3
INTRODUCCIÓN.....	3
CASO INTRODUCTORIO	3
1. REFACTORIZACIÓN.....	5
1.1 Patrones de refactorización más usuales	6
1.1.1 Extraer método	7
1.1.2 Métodos en línea	7
1.2 Refactorización y pruebas.....	8
1.3 Herramientas de ayuda a la refactorización	10
2. CONTROL DE VERSIONES	13
2.1 Tipos	13
2.2 Herramientas	14
2.3 Repositorio	15
3. DOCUMENTACIÓN	16
3.1 Uso de comentarios.....	17
3.2 Herramientas y alternativas.....	18
3.2.1 Gestión documental con software libre	19
3.2.2 Nuxeo.....	19
3.2.3 Alfresco	20
3.2.4 Athento.....	20
RESUMEN FINAL	22

RESUMEN INTRODUCTORIO

En la presente unidad se estudiarán conceptos relacionados con la refactorización, el control de versiones y el proceso de documentación. La refactorización es un proceso que debe ser tenido en cuenta, puesto que aporta técnicas para que el código de una aplicación quede más claro y libre de líneas indeseadas o que pueden ser sustituidas por otras más eficientes, reduciendo su número. El control de versiones es una técnica que permite a los miembros de un equipo de desarrollo controlar cada una de las versiones que se crean y se suben a un repositorio común. Además, el control de versiones permite a los usuarios tener siempre disponible tanto el código del aplicativo como ficheros que hacen referencia a la documentación del mismo. Por último, la propia documentación, es una tarea transversal a todas las fases del ciclo de vida de una aplicación. Es de suma importancia tener documentados todos los procesos desarrollados, para en un futuro solventar cualquier tipo de problema o anomalía. Además, se aconseja llevar la gestión del conocimiento a través de software dedicado a ello. Existen numerosos programas que permiten gestionar la documentación de un proyecto software, tanto de software libre como privativo. En esta unidad se presentan Nuxeo, Alfresco y Athento.

INTRODUCCIÓN

La refactorización es un concepto nuevo que se ha introducido en la terminología del mundo del desarrollo software apoyado por las metodologías ágiles. Como tal, es un proceso que va a permitir a los programadores mejorar sus funciones y automatizar sus procesos permitiendo, en este sentido, obtener ventajas en la realización de sus tareas diarias. Además, como se ha estudiado en las unidades anteriores, llevar una buena gestión de la documentación sobre todo lo relacionado con el desarrollo de software es crucial para el ciclo de vida del proyecto. Por todo ello es significativo para el alumnado adentrarse en estos nuevos conceptos y tomar consciencia de la importancia que conllevan.

CASO INTRODUCTORIO

El equipo de desarrollo de Juan ha establecido un nuevo patrón de trabajo. Todos los miembros del equipo deben tener disponible toda la documentación generada, así como el código resultante de que requisito implementado. Para ello, Juan decide llevar a cabo una buena práctica: instalar un software para el control de versiones.

Con este software instalado en cada equipo, tanto los programadores como analistas, diseñadores y tester son capaces de ver el trabajo de los demás y, dependiendo de los permisos asignados, poder modificar su contenido.

Por otro lado, Juan decide llevar a cabo una refactorización del código del aplicativo. ¿Cuál sería la mejor decisión a tomar? Reúne a sus programadores y asigna a cada uno de ellos un requisito funcional ya programado, para que pueda ser revisado y refactorizado, repartiendo la carga de trabajo. Al estar el código bajo un software para el control de versiones, todos tendrán notificaciones de las novedades existentes en el código.

Al finalizar la unidad el alumnado:

- Conocerá el concepto de refactorización y cómo aplicarlo.
- Identificará los documentos que se deben generar en cada fase del desarrollo del software.
- Conocerá las herramientas que existen en el mercado para la generación de documentación.

1. REFACTORIZACIÓN

Refactorizar (**refactoring**, en inglés) es realizar una transformación al software preservando su comportamiento, modificando sólo su estructura interna para mejorarlo. El término original es de Opdyke, quien lo introdujo por primera vez en 1992, en su tesis doctoral. En 2001, Tokuda y Batory lo definieron como una transformación parametrizada a un programa preservando su comportamiento, modificando automáticamente el diseño de la aplicación y el código fuente subyacente. Decía Fowler que eran cambios realizados en el software para hacerlo más fácil de modificar y comprender, por lo que no son una optimización del código, ya que esto en ocasiones lo hace menos comprensible, ni soluciona errores o mejora algoritmos. Las refactorizaciones pueden verse como un tipo de mantenimiento preventivo, cuyo objetivo es disminuir la complejidad del software en anticipación a los incrementos de complejidad que los cambios pudieran traer.

Una de las razones para refactorizar es ayudar al código a mantenerse en "buena forma", ya que con el tiempo los cambios en el software hacen que este pierda su estructura, y esto hace difícil ver y preservar el diseño. Refactorizar ayuda a evitar los problemas típicos que aparecen con el tiempo, como, por ejemplo, un mayor número de líneas para hacer las mismas cosas o código duplicado. Existen incluso posturas, como a la que comenta la metodología ágil XP, que afirma que la refactorización puede ser una alternativa a diseñar, codificando y refactorizando directamente, sin más. Sin llegar a extremos tan radicales y poco recomendables, sí que es cierto que una buena refactorización cambia el rol del tradicional del "diseño planificado", pudiendo relajar la presión por conseguir un diseño totalmente correcto en fases tempranas, buscando sólo una solución razonable. Otro aspecto importante es que ayuda a aumentar la simplicidad en el diseño. Sin el uso de refactorizaciones se busca la solución más flexible para el futuro, siendo estas soluciones, por lo general, más complejas y, por tanto, el software resultante más difícil de comprender y por ello de mantener. Además, ocurre que muchas veces toda esa flexibilidad no es necesaria, ya que es imposible predecir qué cambiará y dónde se necesitará la flexibilidad, forzado a introducir mucha más flexibilidad de la necesaria (síntoma de esto es la sobrecarga de patrones).



PARA SABER MÁS

Aunque hay varios catálogos de refactorizaciones el más famoso es el de Fowler, que se mantiene en la siguiente página:

- [Refactoring](#)

Tanto (Fowler et al, 1999) como (Piattini y García, 2003) coinciden en que las áreas conflictivas de la refactorización son las **bases de datos** y los **cambios de interfaces**. El cambio de base de datos tiene dos problemas: los sistemas están fuertemente acoplados a los esquemas de las bases de datos y el segundo de ellos radica en la migración tanto estructural como de datos. Se deben aplicar los cambios necesarios y luego migrar los datos existentes en la base de datos, lo que es muy costoso.

El problema del cambio de interface no tiene demasiada complejidad si se tiene acceso al código fuente de todos los clientes de la interface a refactorizar. Sí es problemático cuando ésta se convierte en lo que (Fowler et al, 1999) llama interface publicada (published interface) y no se dispone del código fuente modificable de los clientes de la misma.

1.1 Patrones de refactorización más usuales

Cualquier programador, al desarrollar una aplicación, siempre trata de conseguir que el código tenga la mayor calidad posible. El primer problema con el que se encuentra es lo que se entiende por calidad. Existen muchos artículos de calidad del software que relacionan la calidad con el número de errores que el software presenta. Es importante que el software no falle, pero es importante conocer que existen otras características que determinan si software es de calidad.

En este sentido han surgido estándares de codificación que permiten desarrollos más legibles y patrones para dar soluciones. La simplicidad y la claridad son características importantes de la calidad. Conseguir esa sencillez y claridad es complicado.

Los patrones, en ocasiones, presentan algunos problemas. Es sencillo aplicar ingeniería a los desarrollos intentando aplicar patrones que no serían necesarios. Los patrones en muchos casos enseñan a afrontar problemas de diseño generales, pero no son válidos para afrontar problemas más específicos, como por ejemplo cuando un método presenta demasiados parámetros o cuando sería mejor eliminar una herencia que no justificada.

Existen muchos patrones que ayudan a la refactorización, incluso algunos de ellos son utilizados a diario sin saber que son patrones como tal. A continuación, se muestran dos ejemplos:

1.1.1 Extraer método

Extraer método es una operación de refactorización que proporciona una manera sencilla para crear un nuevo método a partir de un fragmento de código de un miembro existente. Utilizando la extracción a método se puede crear un nuevo método extrayendo una selección del código dentro del bloque del mismo. El nuevo método que se ha extraído contiene el código seleccionado y el código seleccionado del miembro existente se reemplaza por una llamada al nuevo método. Convertir un fragmento de código en su propio método permite **reorganizar el código** de forma rápida y precisa para que sea posible volver a utilizarlo (reutilización) y lograr una mejor legibilidad.



ARTÍCULO DE INTERÉS

Este patrón es muy fácil de utilizar. Puede ver un ejemplo pinchando en el siguiente enlace:

- [Refactoring: Self Encapsulate Field](#)

Extraer método presenta las ventajas siguientes:

- Fomenta el uso de mejores métodos de codificación, dando énfasis a métodos discretos y reutilizables.
- Recomienda el código autodocumentado a través de una buena organización.
- Si se utilizan nombres descriptivos, los métodos de alto nivel se pueden leer como una serie de comentarios.
- Fomenta la creación de métodos más detallados para simplificar la sustitución.

1.1.2 Métodos en línea

Los **métodos en línea** es una técnica común utilizada en programación, siendo una aplicación directa de la definición de refactoring en cuanto a la optimización de código cambiando su estructura interna sin afectar su funcionalidad. El caso específico del método en línea se presenta cuando se tiene uno o más métodos cuyos códigos son lo suficientemente auto-explicativos como para poder prescindir de ellos. Bastará con sustituir las llamadas al método por el cuerpo de dicho método.



PARA SABER MÁS

Todos los patrones de refactorización y ejemplos los puedes encontrar en el siguiente enlace:

- [Refactorización](#)

1.2 Refactorización y pruebas

Para poder refactorizar de forma satisfactoria es indispensable contar con un buen lote de casos de prueba que validen el correcto funcionamiento del sistema. Estos casos de prueba deben cumplir con los siguientes requisitos:

- Deben ser automáticos de tal manera que se puedan ejecutar todos a la vez con simplemente hacer clic un botón.
- Deben ser auto verificables de forma que no se invierta tiempo en la verificación de los resultados de los mismos. Estos errores deben ser reportados por cada caso de prueba.
- Deben ejecutarse de manera independiente uno del otro, para que los resultados de uno no afecten a los resultados del resto.

Los casos de prueba permiten verificarse repetida e incrementalmente si los cambios introducidos han alterado el comportamiento observable del programa. El primer paso del proceso de refactorización es ejecutar las pruebas antes de haber efectuado cualquier cambio. Esta acción provee información sobre el comportamiento actual del sistema. Los resultados deben ser los mismos que se obtengan luego de la refactorización. El segundo paso, consiste en analizar los cambios a realizar y el tercero es, finalmente, la aplicación del cambio. Como fue mencionado anteriormente se vuelven a ejecutar las pruebas, de tal manera que, al contrastar los resultados iniciales con los resultados obtenidos tras efectuar la refactorización, deben ser exactamente los mismos (Fowler et al, 1999). Cabe aclarar que una optimización de código no es una refactorización ya que, si bien tienen en común la modificación del código fuente sin alterar el comportamiento observable del software, la diferencia radica en la forma de impactar el código fuente: la optimización suele agregarle complejidad.



ARTÍCULO DE INTERÉS

¿En qué consiste la optimización de código?

- [La optimización: una mejora en la ejecución de programas](#)

Es importante mencionar la relevancia de las pruebas unitarias para el desarrollo de software, ya que realizan otros aportes al margen de comprobar que el código funciona correctamente, entre los que se encuentran (Massol y Husted, 2003):

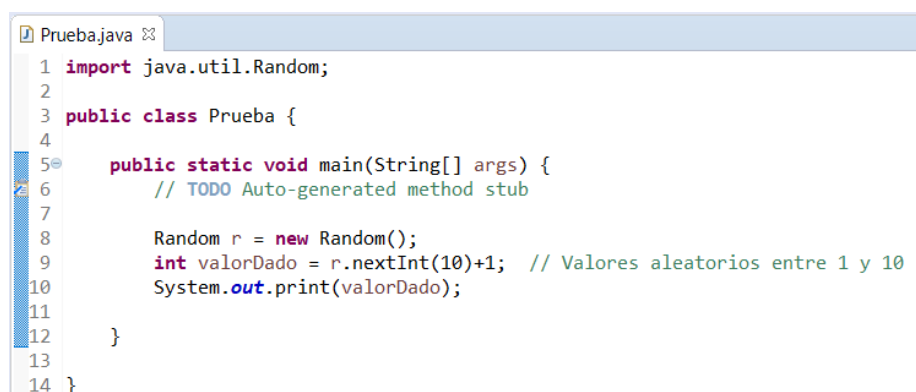
- **Previenen pruebas de regresión** y limitan el debug de código. En primera instancia demuestran que el código fuente funciona correctamente y, consecuentemente, aportan confianza a la hora de modificar el código ya que se pueden ejecutar nuevamente y verificar si la aplicación del cambio fue correcta. Un buen lote de casos de prueba permite detectar fallas temprano, con la ventaja de poder resolverlas con antelación y reducir el tiempo invertido en la detección de fallas.
- **Facilitan la refactorización.** Tal como se mencionó con anterioridad, es importante tener casos de prueba que verifiquen el correcto funcionamiento del sistema. El riesgo de introducir una falla luego de una refactorización queda reducido, ya que se pueden detectar a tiempo. Las pruebas unitarias proveen la seguridad suficiente como para facilitar la refactorización.
- **Mejoran el diseño de implementación.** Es la primera prueba a la que se somete el código fuente, por lo que necesitan que el sistema a probar sea flexible y factible de probar de forma unitaria aislada. Si el código no es fácil de probar de forma unitaria es una señal que indica que debe ser refactorizado.

Si bien los casos de prueba presentan ventajas, también tienen un costo asociado, deben ser mantenidos en línea conforme se modifica el código fuente. Esto significa que si se elimina funcionalidad del sistema deben eliminarse los casos de prueba de dicha funcionalidad, si se agrega funcionalidad se deberán agregar nuevos casos y, si se modificara funcionalidad existente, las pruebas unitarias deberán ser actualizadas de acuerdo al cambio realizado.

1.3 Herramientas de ayuda a la refactorización

Muchos editores y entornos de desarrollo presentan soporte automatizado refactorización. A continuación, se indican lo más conocidos: Eclipse, NetBeans, JDeveloper, Visual Studio, Visual Assist y DMS Toolkit Reingeniería de Software.

Las herramientas de Refactoring son especialmente útiles cuando se trata de realizar modificaciones o actualizaciones en las líneas código, que afecta a varios elementos del diseño. Eclipse incorpora desde versiones muy tempranas opciones para refactorizar nuestro código. En Eclipse, se puede acceder a las operaciones de Refactoring a través de la opción Refactor en el menú principal o en el menú pop-up del Editor. Como ejemplo de refactorización se propone una extracción a método mediante el IDE Eclipse. Para ello se parte de las siguientes 3 líneas de código, cuyo cometido es obtener por pantalla un valor numérico entero aleatorio entre el 1 y 10. Para ello se hace uso de la clase Random, importándola previamente:

A screenshot of the Eclipse IDE showing a Java file named 'Prueba.java'. The code is as follows:

```
1 import java.util.Random;
2
3 public class Prueba {
4
5     public static void main(String[] args) {
6         // TODO Auto-generated method stub
7
8         Random r = new Random();
9         int valorDado = r.nextInt(10)+1; // Valores aleatorios entre 1 y 10
10        System.out.print(valorDado);
11
12    }
13
14 }
```

Imagen: Código Java para número aleatorio

El siguiente paso consiste en seleccionar las 3 líneas de código a refactorizar y acceder a la opción de Eclipse Refactor → Extract Method, y en el nombre del método escribir, por ejemplo, aleatorio y pulsar OK:

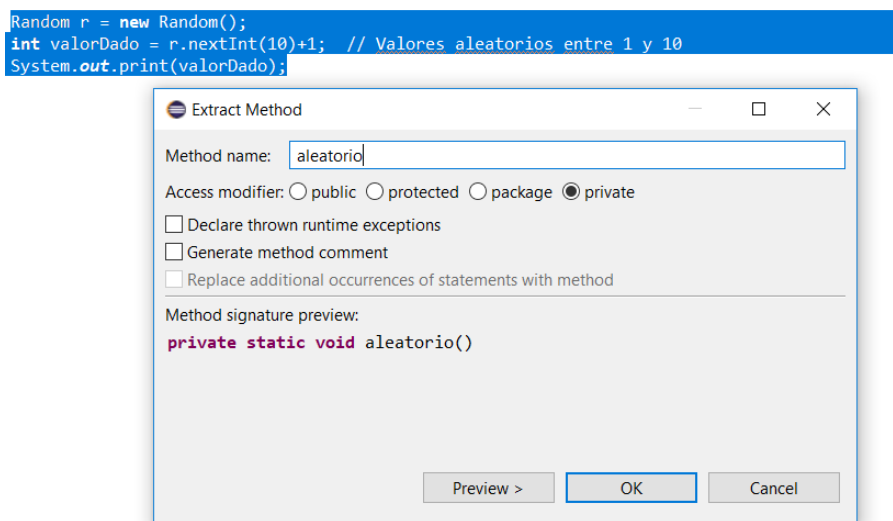


Imagen: Extracción de método con Eclipse

La refactorización se realiza automáticamente con la extracción de las 3 líneas de código seleccionadas añadiéndolas al nuevo método creado:

```

1  import java.util.Random;
2
3  public class Prueba {
4
5      public static void main(String[] args) {
6          // TODO Auto-generated method stub
7
8          aleatorio();
9
10     }
11
12     private static void aleatorio() {
13         Random r = new Random();
14         int valorDado = r.nextInt(10)+1; // Valores aleatorios entre 1 y 10
15         System.out.print(valorDado);
16     }
17
18 }

```

Imagen: Código refactorizado tras la extracción a método

La herramienta de Refactoring de Eclipse permite realizar otras refactorizaciones.



ARTÍCULO DE INTERÉS

En el siguiente enlace puede ver más ejemplos de refactorización con Eclipse:

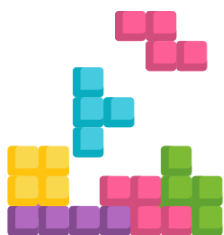
- [¿Qué es refactoring?](#)



PARA SABER MÁS

En la siguiente web se muestra una presentación sobre la refactorización en Visual Studio:

- [Refactorización](#)

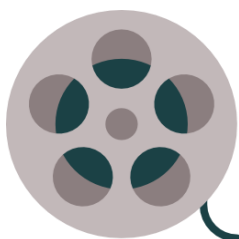


EJEMPLO PRÁCTICO

Se realizan una serie de pruebas en el código de una aplicación y se comprueba que el acceso a cierta información se realiza de forma lenta. Al analizar el código, aparentemente, no hay errores, pero siempre se comporta de esta manera. ¿Cuál será el procedimiento a aplicar para intentar solventar este problema? ¿Qué debería analizar el técnico para dar con la clave de esta lentitud tan frecuente?

SOLUCIÓN:

En primer lugar, es importante analizar bien las líneas de código que ejecutan y definen esa funcionalidad. Si se puede aplicar algún tipo de refactorización, este debe ser el primer paso. Una vez realizado, probar de nuevo la funcionalidad, para asegurar que no ha habido ningún cambio asociado a esta nueva versión. Si el problema persiste, la solución es optimizar el código que, a diferencia de la refactorización, permite analizar línea por línea el tiempo de ejecución correspondiente. De esta forma, será mucho más fácil ver en qué parte exacta del código se está produciendo esta lentitud en el acceso a la información.



VIDEO DE INTERÉS

Este videotutorial muestra a refactorizar usando NetBeans:

- [Tutorial de Refactor para Clases en Netbeans](#)

2. CONTROL DE VERSIONES

Un **sistema de control de versiones** (o sistema de control de revisiones) es una combinación de tecnologías y prácticas para seguir y controlar los cambios realizados en los ficheros del proyecto, en particular en el código fuente, en la documentación y en las páginas web. Lo normal hoy en día en las factorías de software es que al menos el código fuente del proyecto esté bajo un control de versiones. La razón por la cual el control de versiones es universal es porque ayuda virtualmente en todos los aspectos al dirigir un proyecto: comunicación entre los desarrolladores, manejo de los lanzamientos, administración de fallos, estabilidad entre el código y los esfuerzos de desarrollo experimental y atribución y autorización en los cambios de los desarrolladores. El sistema de control de versiones permite a una fuerza coordinadora central abarcar todas estas áreas.

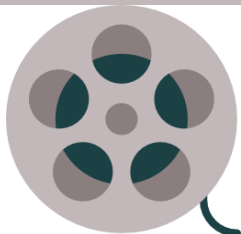
El núcleo del sistema es la gestión de cambios: identificar cada cambio a los ficheros del proyecto, anotar cada cambio con metadata como la fecha y el autor de la modificación y disponer esta información para quien sea y como sea. Es un mecanismo de comunicación donde el cambio es la unidad básica de información.

2.1 Tipos

Los sistemas de control de versiones se pueden clasificar en 2 grandes grupos:

- **Centralizados:** En un sistema de control de versiones centralizado todas las fuentes y sus versiones están almacenados en un único directorio (llamado repositorio de fuentes) de un ordenador (un servidor). Todos los desarrolladores que quieran trabajar con esas fuentes, deben pedirle al sistema de control de versiones una copia local para trabajar.
En ella realizan todos sus cambios y cuando están listos y funcionando, le piden al sistema de control de versiones que guarde las fuentes modificados como una nueva versión. Algunos ejemplos son **CVS** y **Subversión**.
- **Distribuidos:** En un sistema de control de versiones distribuido no hay un repositorio central. Todos los desarrolladores tienen su propia copia del repositorio, con todas las versiones y toda la historia. Por supuesto, según van desarrollando y haciendo cambios, sus fuentes y versiones van siendo distintas unas de otras.
Sin embargo, los sistemas de control de versiones distribuidos permiten que en cualquier momento dos desarrolladores cualesquiera puedan sincronizar sus repositorios.

Si uno de los desarrolladores ha modificado determinados ficheros fuentes y el otro no, los modificados se convierten en la versión más moderna. Ejemplos: **Git** y **Mercurial**.



VIDEO DE INTERÉS

En este vídeo se introduce el tema de sistemas de control de versiones distribuidos:

- [Repositorios de control de versiones distribuidos: El futuro del código fuente](#)

2.2 Herramientas

Aunque un sistema de control de versiones puede realizarse de forma manual, es muy aconsejable disponer de herramientas que faciliten esta gestión dando lugar a los llamados **sistemas de control de versiones** o **VCS** (del inglés Version Control System). Estos sistemas facilitan la administración de las distintas versiones de cada producto desarrollado, así como las posibles especializaciones realizadas (por ejemplo, para algún cliente específico). A continuación, se muestran algunos ejemplos de control de versiones:

- **CVS:** Ha estado vigente en las factorías de software durante bastante tiempo y muchos desarrolladores están ya familiarizados con él. En su día fue muy revolucionario: fue el primer sistema de control de versiones de código abierto con acceso a redes de área amplia para desarrolladores, y el primero que ofreció checkouts anónimos de sólo lectura, los que dieron a los desarrolladores una manera fácil de implicarse en los proyectos. CVS sólo versiona ficheros, no directorios. Ofrece ramificaciones, etiquetado y un buen rendimiento en la parte del cliente, pero no maneja muy bien ficheros grandes ni ficheros binarios. Tampoco soporta cambios atómicos.
- **Subversion:** Fue escrito para reemplazar a CVS, es decir, acceder al control de versiones de la misma manera que CVS lo hace, pero sin los problemas o falta de utilidades que más frecuentemente incomodan a los usuarios de CVS. Uno de los objetivos es encontrar la transición relativamente cómoda para los ya acostumbrados a CVS.
- **SVK:** Aunque se ha construido sobre Subversion, **SVK** se parece más a algunos de los anteriores sistemas descentralizados que a él. SVK soporta desarrollo distribuido, cambios locales, mezcla sofisticada de cambios y la habilidad de reflejar/clonar árboles desde sistemas de control de versiones que no son SVK.

- **Mercurial:** Es un sistema de control de versiones distribuido que ofrece, entre otras opciones, una completa indexación cruzada de ficheros y conjuntos de cambios, unos protocolos de sincronización SSH y HTTP eficientes respecto al uso de CPU y ancho de banda, una fusión arbitraria entre ramas de desarrolladores, una interfaz web autónoma integrada y portabilidad UNIX, MacOS X y Windows.
- **GIT:** Es un proyecto llevado a cabo por Linus Torvalds para manejar el árbol fuente del kernel de Linux. Al principio GIT se enfocó bastante en las necesidades del desarrollo del kernel, pero se ha expandido más allá de ello y en la actualidad es utilizado por otros proyectos aparte del kernel de Linux. Su página web dice que está diseñado para manejar proyectos muy grandes eficaz y velozmente. Se usa sobre todo en varios proyectos de código abierto. Cada directorio de trabajo de GIT es un repositorio completo con plenas capacidades de gestión de revisiones, sin depender del acceso a la red o de un servidor central.
- **Bazaar:** Está todavía en desarrollo. Será una implementación del protocolo GNU Arch, mantendrá compatibilidad con el protocolo GNU Arch a medida que evolucione y trabajará con el proceso de la comunidad GNU Arch para cualquier cambio de protocolo que fuera requerido a favor del agrado del usuario.



PARA SABER MÁS

Puedes conocer más información de los distintos controles de versiones:

- [Introduction to CVS](#)
- [Apache Subversion](#)
- [GIT](#)
- [Bazaar](#)

2.3 Repositorio

Subversión es un sistema centralizado para compartir información. En su núcleo está un **repositorio**, que es un almacén central de datos. Almacena información en forma de un árbol de archivos, una jerarquía típica de archivos y directorios. Un número de clientes se conectan al repositorio y luego leen o escriben esos archivos. Al escribir datos, el cliente hace que la información esté disponible para los otros. Al leer los datos, el cliente recibe información de los demás.

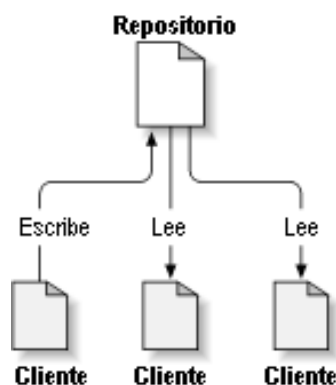


Imagen: Operaciones de los clientes sobre un repositorio

Lo que hace al repositorio de Subversión especial es que recuerda todos los cambios que se hayan escrito en él: cada cambio en cada archivo, e incluso los cambios en el propio árbol de directorios, como el añadir, borrar o reorganizar archivos y directorios.

Cuando un cliente lee datos de un repositorio, normalmente ve únicamente la última versión del árbol de archivos. Pero el cliente también tiene la capacidad de ver estados previos del sistema de archivos. Por ejemplo, un cliente puede hacer preguntas históricas, como ¿qué contenía este directorio el último miércoles?, o ¿quién fue la última persona que cambió este archivo, y qué cambios hizo? Esta es la clase de preguntas que forman el corazón de cualquier sistema de control de versiones: son sistemas que están diseñados para guardar y registrar los cambios a los datos a lo largo del tiempo.

3. DOCUMENTACIÓN

En la ejecución de un proyecto informático o desarrollo software se deben de seguir unas fases desde que se plantea el problema hasta que se dispone de la aplicación totalmente operativa en el ordenador. Los pasos son los siguientes:

- Análisis de factibilidad y análisis de requerimientos.
- Diseño del sistema.
- Implementación.
- Pruebas y validación.
- Explotación y mantenimiento.

Cada uno de estos pasos debe de llevar asociado un documento. Estos documentos son muy importantes ya que van a regir las fases del ciclo de vida del software y se recogen los pasos seguidos en cada fase para su ejecución.

No es viable la solución mostrada por algunos programadores de ir directamente a la implementación sin antes pararse en las fases 1, 2 y 3. Un trabajo deficiente en estas fases supone una mala definición del problema y por tanto el sistema no cumplirá seguramente con todos los requisitos. El diseño del sistema no será efectivo y los errores serán de difícil solución.



PARA SABER MÁS

En esta web se detalla cómo documentar un sistema de software:

- [Documentación de programas](#)

3.1 Uso de comentarios

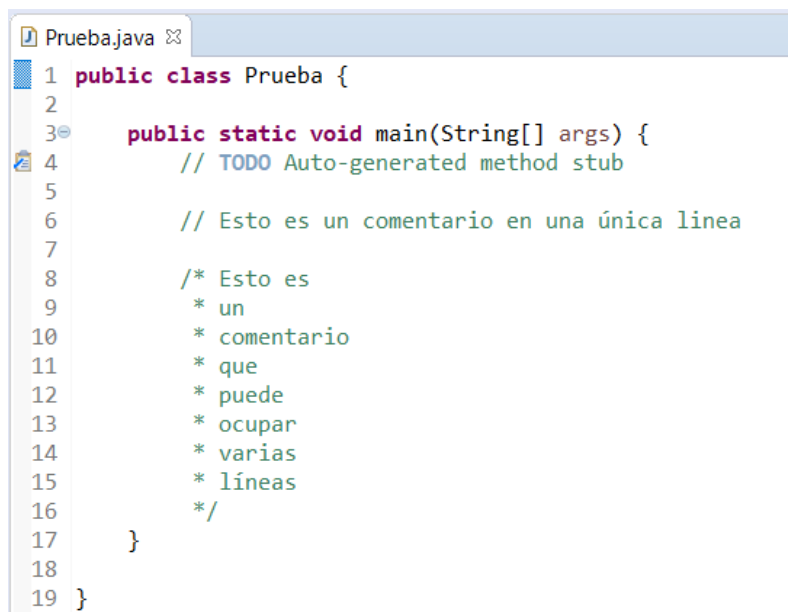
Durante la fase de implementación es necesario comentar convenientemente cada una de las partes del programa. Estos comentarios se incluyen en el código fuente con el objeto de clarificar y explicar cada elemento del programa. Por tanto, se deben comentar las clases, las variables, los módulos y, en definitiva, todo elemento que se considere importante.

Esta documentación tiene como objeto hacer más comprensible el código fuente a otros programadores que tengan que trabajar con él en el futuro, ya sea porque forman parte del grupo de desarrollo, el programa va a ser mantenido o modificado por otra persona distinta al programador inicial. También resulta muy útil durante la depuración y el mantenimiento del programa por el propio programador, al paso del tiempo las decisiones se olvidan y surgen dudas hasta en el propio programador de porqué se hicieron las cosas de una determinada manera y no de otra.



SABÍAS QUE...

En Java existen 2 tipos de comentarios, los que ocupan una única línea, delimitados por dos barras (//) y los que pueden ocupar varias líneas, delimitados por barra y asterisco (/*) seguido del comentario, y terminando con asterisco y barra (*//).



```

1 public class Prueba {
2
3     public static void main(String[] args) {
4         // TODO Auto-generated method stub
5
6         // Esto es un comentario en una única línea
7
8         /* Esto es
9          * un
10         * comentario
11         * que
12         * puede
13         * ocupar
14         * varias
15         * líneas
16         */
17     }
18
19 }

```

Imagen: Comentarios de código en Java



ARTÍCULO DE INTERÉS

Se debería comentar el código cuidadosamente y con buen gusto. Las directrices estilísticas se encuentran disponibles la siguiente guía de estilo de Java:

- [GUÍA DE ESTILO EN JAVA](#)

3.2 Herramientas y alternativas

Hasta cierto punto, la elección de la herramienta de documentación adecuada está determinada por el lenguaje con el que esté escrito el código, bien sea C++, C#, Visual Basic, Java o PHP, ya que existen herramientas específicas para estos y otros lenguajes. En otros casos, la herramienta a utilizar se determina dependiendo del tipo de documentación que sea necesario.

- Los procesadores de texto de Microsoft Word son adecuados para crear archivos de texto de la documentación. Para archivos de texto largos y complejos, se prefieren herramientas de documentación como Adobe FrameMaker.
- Se pueden crear archivos de ayuda para documentar el código fuente utilizando cualquier herramienta para crear archivos de ayuda, como RoboHelp, Help and Manual, Doc-To-Help, MadCap Flare o HelpLogix.



PARA SABER MÁS

Para conocer más sobre Adobe FrameMaker y RoboHelp puedes acceder desde estos enlaces:

- [Adobe FrameMaker](#)
- [Adobe RoboHelp](#)

3.2.1 Gestión documental con software libre

A la hora de implementar un sistema de gestión documental en un entorno profesional existen en la actualidad una gran variedad de soluciones, tanto de software libre como propietarias. Cabe destacar los proyectos **Nuxeo** y **Alfresco**. Los profesionales de la gestión documental consideran estas plataformas las grandes alternativas a soluciones propietarias como **Sharepoint** o **Documentum**, con las cuales se pueden evitar los inconvenientes ya señalados de costosas licencias e implementaciones complejas para adaptar la plataforma a los entornos de trabajo.

Optar por soluciones de software libre supone por tanto, también una forma más sencilla de trabajar, sin que ello implique una pérdida de eficacia y eficiencia, lo cual es otra ventaja añadida.

3.2.2 Nuxeo

Nuxeo es un software que permite implementar con gran funcionalidad un repositorio documental corporativo. Aporta soluciones a las necesidades primarias de gestión documental de las empresas, permitiendo gestionar cómodamente documentos mediante control de versiones, flujos de trabajo asociados, publicación remota o búsqueda avanzada a texto completo, además de integración con suite ofimáticas habituales como Microsoft Office y Open Office. A través de Nuxeo DAM se ofrece soporte para imágenes y vídeos.



PARA SABER MÁS

Para descargar el software y obtener más información sobre este sistema de gestión documental, visite:

- [Nuxeo](#)

3.2.3 Alfresco

Se trata de otro gran ejemplo de ECM open source. Es una solución versátil compatible con software tanto de la vertiente Microsoft, como de la rama Linux. Posibilita la creación y gestión de contenidos empresariales desde una gran cantidad de CMS, blogs y paquetes ofimáticos (Office y OpenOffice). Además, ofrece una gran variedad de herramientas colaborativas como calendarios individuales y de equipo, feeds de actividad, tableros de discusión, etc.

Alfresco es para las empresas ante todo colaboración, pero también constituye un gestor documental y un completo CMS corporativo. Su base de programación, junto a Nuxeo, es Java, lo que los convierte en soluciones multiplataforma adaptables a cualquier entorno, a diferencia de Sharepoint que depende de tecnología propietaria Windows, lo cual restringe sus posibilidades de despliegue.



PARA SABER MÁS

Para descargar el software y obtener más información sobre este sistema de gestión documental, visite:

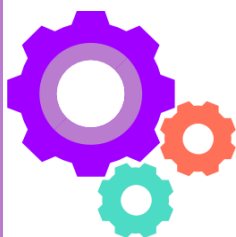
- [Alfresco](#)

3.2.4 Athento

Athento es una capa software que añade funcionalidad de última generación a sistemas de gestión de contenidos empresariales como Nuxeo y Alfresco, proporcionándoles interoperabilidad para conectarse con aplicaciones de terceros. Utiliza estándares abiertos e incorpora los últimos avances de la Sociedad de la información como tecnología semántica, procesamiento de la imagen mediante OCR, Redes Neuronales, Análisis del color y procesamiento del lenguaje natural (PNL), siendo fundamental su aportación en fases esenciales del ciclo de gestión de documental como la captura, la gestión y la distribución.



Imagen: Integración de la Gestión Documental



PARA SABER MÁS

Para descargar el software y obtener más información sobre este sistema de gestión documental, visite:

- [Athento](#)

RESUMEN FINAL

En esta unidad se presenta el concepto de refactorización y qué ventajas aporta a la fase de codificación. Entre varias operaciones se han visto 2: la extracción a método y el método en línea.

Otro concepto de en esta unidad es el control de versiones. Se trata de una técnica mediante la cual los miembros de un equipo pueden compartir las diferentes versiones tanto el código como de la documentación de un proyecto y tenerla siempre disponible. Existen diferentes herramientas que soportan este control.

Por último, destacar la importancia que adquiere la documentación en todas las fases del proyecto y disponer de un buen gestor documental para controlar toda la gestión del conocimiento del equipo.