TEMA ~~11~~ – OPTIMIZACIÓN

AHORA TEMA 5

Y DOCUMENTACIÓN

11.1- INTRODUCCIÓN

El presente tema introduce los conceptos de Refactorización, Documentación y Control de Versiones, que aseguran que, por un lado, los proyectos de software mantengan una **trazabilidad** a lo largo de su vida, a pesar de los cambios que hayan podido sufrir en sus requisitos, en su código e independientemente del número de personas que hayan pasado por su construcción y depuración, y por otro, su código sea óptimo, limpio y de calidad.

11.2- REFACTORIZACIÓN

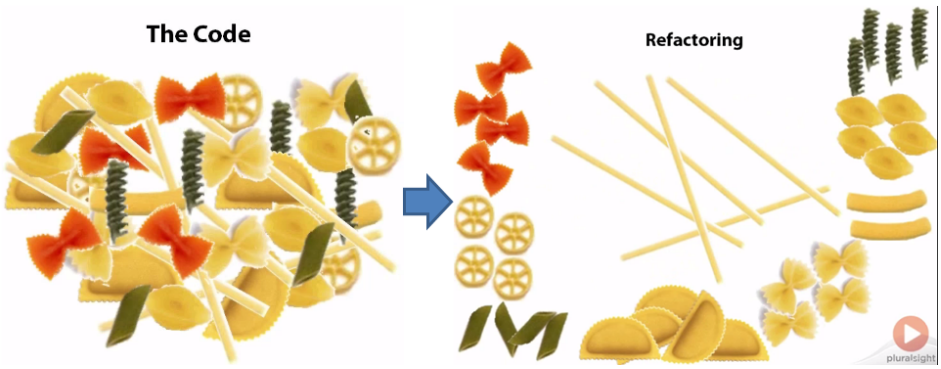
La refactorización es una disciplina técnica, que consiste en realizar pequeños ajustes y transformaciones en el código de un programa, para mejorar la estructura sin que cambie el comportamiento ni la funcionalidad del mismo.

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 trata de:

- mejorar el diseño del software
- que sea más fácil de entender
- que su mantenimiento sea más sencillo
- facilitar a encontrar errores
- que nuestros programas sean más rápidos en tiempo de ejecución

Cuando se refactoriza, se está mejorando el diseño del código **después** de haberlo escrito. Podemos incluso, en una situación extrema, partir de un mal diseño y, aplicando la refactorización, llegar a un código bien diseñado.



Cada paso es simple, por ejemplo:

- mover una propiedad desde una clase a otra (por que sea más lógico que esté allí)
- convertir determinado código en un nuevo método
- eliminar niveles excesivos de anidamiento

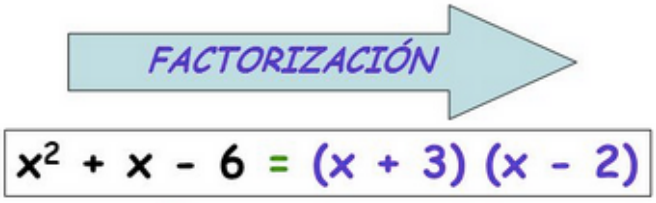
- dividir y reestructurar clases y/o funciones demasiado grandes
- corregir nombres incorrectos o confusos en clases, funciones y variables, etc…

Pero la acumulación de todos estos pequeños cambios pueden mejorar de forma ostensible el diseño y hacer el código más ligero, limpio y mantenible.

Hay una sentencia, tomada de los Boy Scouts que lo resume perfectamente:

"Deja un módulo en mejor estado que como lo encontraste"

El concepto de refactorización de código (refactoring), se basa en el concepto matemático de factorización de polinomios.



Pongamos un ejemplo de refactorización típica: "***Encapsular Campo***".

> *Para encapsular los campos, se deben 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. Ganamos de esta manera en ocultación y modularidad, ofreciendo un interfaz al exterior más limpio y reusable.*

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. Solo los cambios efectuados que redunden en conseguir un software más fácil de entender son refactorizaciones.

Hay que resaltar, por otra parte, que la refactorización no cambia el comportamiento observable del software desde fuera. El software sigue cumpliendo la misma función que hacia antes. Ningún usuario, ya sea usuario

final u otro programador, deben ser capaces de determinar qué cosas han cambiado.

Por supuesto, una refactorización debe crear un nuevo código exento de errores, de lo contrario podemos entrar en un bucle muy pernicioso. Para ello será muy importante incluir y manejar el concepto de **control de versiones**, que veremos más adelante.

Cuándo se debe refactorizar y cuándo no

Pese a las muchas ventajas que ofrece la refactorización, es necesario saber cuándo está recomendado hacer lo y cuando no.

Veamos el primer caso, cuando sí refactorizar:

- cuando desarrollamos usando TDD (Test Driven Development): tal y como vimos en el tema anterior, la refactorización es precisamente, la tercera fase: Red – Green – Refactor
- cuando desarrollamos usando PDD (Pain driven development): Si el código te causa dolor, entonces refactorizalo, porque, literalmente, no hay por donde cogerlo
- cuando se necesita agregar una nueva funcionalidad que puede adaptarse o empotrarse a las ya existentes. Reformar código ya escrito puede hacer que la nueva funcionalidad quede cubierta
- cuando corregimos bugs, ya que a veces los bugs anidan en código caótico, y al reordenarlo, saltan a la vista
- cuando hacemos la **revisión de código** (técnica estática de prueba)

Y entonces, ¿cuando no se debe refactorizar?

- cuando el coste sea muy alto y la única opción viable sea reescribir toda la aplicación. Mejor empezar de nuevo
- cuando no se usa un código (*death code*, fue escrito y comentado ó abandonado posteriormente), en este caso lo recomendable es borrarlo, antes de que el código se convierta en un cementerio zombie.
- cuando el fin del plazo de entrega esté cerca, ya que pueden aparecer nuevos bugs que no se llegan a detectar por falta de tiempo y el producto decepcione al cliente. En estos casos se prefiere el código funcionando mejorable al código no funcional

Principios de Refactorización

Hay una serie de principios que se enuncian de manera escueta:

- **KISS** (Keep it simple stupid), no uses más líneas de las necesarias
- **DRY** (Dont repeat yourself), no desarrolles código redundante
- **YAGNI** (You aren't gonna need it)", es decir, sólo debe desarrollarse lo que se vaya a usar en el momento
- Escribe código expresivo, autodocumentado
- Separa responsabilidades en módulos aparte
- Usa un nivel de abstracción apropiado

Herramientas de refactorización

Existen bastantes herramientas para ayudarnos en la tarea de refactorización:

- Visual studio (Microsoft)
- JustCode (Telerik)
- ReSharper (JetBrains)
- CodeRush (DevExpress)
- Visual Assist X (Whole Tomato)

Así como plug-ins para los IDE's más habituales: Netbeans, Eclipse,…

Otras refactorizaciones

Finalmente, comentar tan sólo que existen otras posibles refactorizaciones si miramos más allá del código de programación:

- **Refactorización de base de datos**: se realizan cambios en el esquema (Schema) de una base de datos con la finalidad de mejorar su diseño. Cuando el modelo ha ido evolucionando durante el proceso de desarrollo de software, algunos añadidos o parcheados pueden ensuciar y enrevesar el diseño de la base de datos

- **Refactorización de Interfaz de usuario**: se efectúan cambios de estandarización de la presentación, manteniendo la funcionalidad, por ejemplo alinear campos, alinear tamaños de botones, diseñar nuevos sistemas de navegación, modificar el layout, etc…

Con esta ampliación del enfoque, se extendería y adaptaría los principios de esta la técnica desde el **controlador** al **modelo** y a la **vista** de una aplicación.

Técnicas de refactorización

Existe todo un corpus de técnicas de refactorización recomendadas, que en su mayor parte, exceden los contenidos del presente curso.

Pero, enlazando con los contenidos del tema anterior, de testing, un concepto a tener muy claro para aplicar una Refactorización de manera segura, es que es muy importante que el comportamiento del sistema sea **conocido y verificado antes de aplicar la modificación**, con la finalidad que se pueda mejorar el sistema si temor a que este deje de funcionar adecuadamente.

Para ello, se requiere aplicar un conjunto completo de pruebas en ambiente de desarrollo antes de aplicar la modificación y después de aplicar la Refactorización, las pruebas en ambiente de desarrollo se deben ejecutar de nuevo, asegurando de esta manera que el software mantenga. el comportamiento adecuado.

Es fundamental que estas pruebas se realicen después de aplicar cada Refactorización pequeña, evitando aplicar muchas Refactorizaciones sin probar, así sabremos en caso de error, cual es la modificación de código que lo ocasionó.

Para más información, se recomienda visitar la siguiente dirección, de la cual se ha obtenido parte del texto que antecede:
https://code2read.com/2015/08/10/code-design-que-es-refactorizacion/

Pero especialmente interesante para los contenidos de esta asignatura, es el contenido de la siguiente página, que particulariza para código java:
http://entornos.codeandcoke.com/doku.php?id=apuntes:refactorizacion

11.3- CONTROL DE VERSIONES

Se denomina **control de versiones** a la gestión de los cambios que se realizan sobre los elementos de algún producto (en especial software) a medida que se va desarrollando o modificando a lo largo del tiempo, o sobre una configuración del mismo.

En el ámbito de esta asignatura, nos interesa únicamente hablar de código fuente, pero en realidad, cualquier tipo de archivo generado por ordenador, puede ponerse bajo control de versiones: diseños gráficos, sitios web, documentos administrativos, composiciones musicales, etc…

Una **versión**, **revisión** o **edición** de un producto, por otra parte, es el estado concreto en el que se encuentra el mismo en un momento dado de su desarrollo o modificación (la *foto fija*) y como veremos, éstas versiones no siempre se suceden según una secuencia temporal, sino que pueden bifurcarse y darse en paralelo.

Aunque un sistema de control de versiones puede realizarse de forma manual (con hábitos sistemáticos de registro y grabación), es muy aconsejable disponer de herramientas que faciliten esta gestión dando lugar a los llamados sistemas de control de versiones o **CVS** (del inglés Version Control System). Estos sistemas facilitan la administración y salvaguarda de las distintas versiones de cada producto desarrollado, así como las posibles especializaciones.

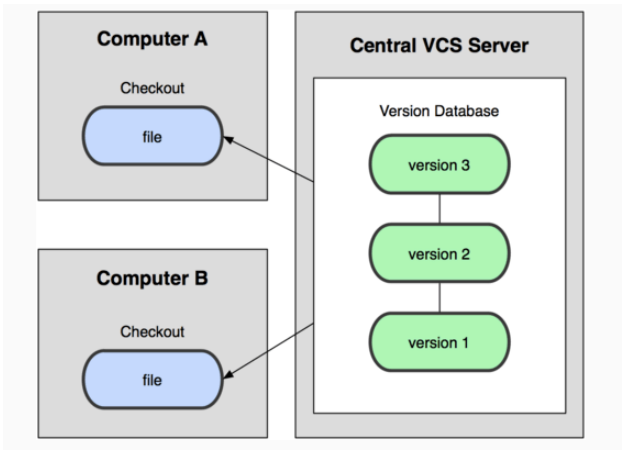
Hay bastantes herramientas de este tipo, p.ej: CVS, Subversion, SourceSafe, ClearCase, Darcs, Bazaar, Plastic SCM, Git, SCCS, Mercurial, Perforce, Fossil SCM, Team Foundation Server.

En particular, como el control de versiones tiene especial importancia en la industria informática, desde la perspectiva del control de versiones del **código fuente** en programación, aparecen sistemas de control específicos para código fuente, que reciben también el nombre más concreto de **SCM** (siglas del inglés Source Code Management).

El control de versiones tiene varios propósito importantes:

- ser utilizado como copia de seguridad
- revertir archivos a un estado anterior
- revertir el proyecto entero a un estado anterior
- comparar cambios a lo largo del tiempo
- ver quién modificó por última vez algo que puede estar causando un problema
- ver quién introdujo un error y cuándo

Por otra parte, se puede tener instalado un CVS **en local** o **en la nube**.



La segunda opción es la más utilizada, porque además de las funciones anteriores, se permite:

- el **trabajo colaborativo en equipo**
- la independencia de una instalación hardware concreta

Volviendo al tema de la seguridad, haremos un especial hincapié en que usar un CVS remoto significa también tener un respaldo o back-up de todo el trabajo realizado. Si se produjera algún problema en un equipo hardware local, se podrían rescatar el trabajo con bastante facilidad si está subido a la nube.

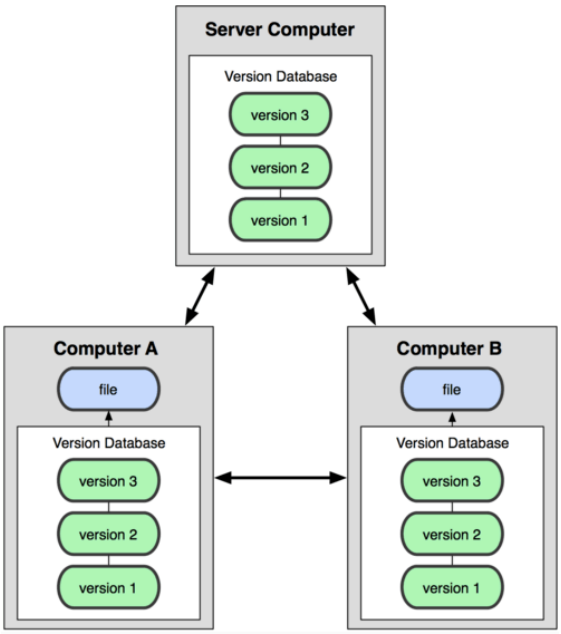
Además, todo el mundo puede saber (hasta cierto punto) en qué están trabajando los otros colaboradores del proyecto y los administradores tienen control detallado de qué puede hacer cada uno.

Sin embargo, esta configuración, si está centralizada, también tiene serias desventajas. La más obvia es el punto crítico de fallo que representa el servidor centralizado.

Si ese servidor se cae durante una hora, entonces durante esa hora nadie puede colaborar o guardar cambios versionados de aquello en que están trabajando. Si el disco duro en el que se encuentra la base de datos central se corrompe, y no se han llevado copias de seguridad, pierdes absolutamente.

Es aquí donde aparece la solución de los sistemas de control de versiones distribuidos (Distributed Version Control Systems o **DVCSs** en inglés).

En un DVCS (como Git, Mercurial, Bazaar o Darcs), los clientes no sólo descargan la última versión de los archivos, también replican completamente el repositorio. Así, si un servidor muere, y estos sistemas estaban colaborando a través de él, cualquiera de los repositorios de los clientes puede copiarse en el servidor para restaurarlo.



Las herramientas de control de versiones, suelen estar formadas por una serie de **componentes**, sobre los cuales, se pueden ejecutar órdenes y realizar intercambio datos entre ellos.

Típicamente, son:

- **Repositorio:** Es el lugar de almacenamiento de los datos de los proyectos. Suele ser un directorio en algún servidor
- **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.

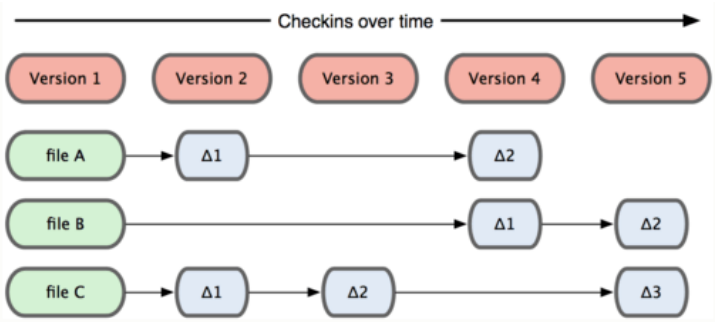
Git y Github (Tomado de la documentación oficial Git)

Git aparece en el año 2005 como una herramienta libre de control de versiones para el grupo de trabajo de la comunidad de desarrollo Linux. Hasta el momento, ésta utilizaba una herramienta propietaria llamada BitKeeper.

Manteniendo las ventajas que tenía BitKeeper, y mejorando los inconvenientes, se desarrolla esta nueva herramienta, buscando la velocidad, sencillez, apoyo al desarrollo no-lineal, arquitectura distribuida y escalabilidad

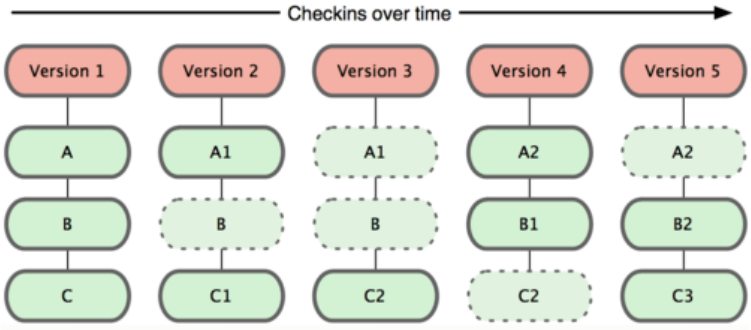
Como todo el software libre, desde 2005, Git ha sufrido una rápida evolución, siendo actualmente el CVS más utilizado.

La principal diferencia entre Git y cualquier otro VCS es que la mayoría de ellos almacenan los archivos originales y sus sucesivos cambios a lo largo del tiempo (CVS, Subversion, Perforce, Bazaar, etc.), según el siguiente esquema:



Por el contrario, Git almacena un conjunto de instantáneas de un sistema de archivos. Cada vez que se confirma un cambio, se hace una foto del aspecto de todos los archivos en ese momento, y se guarda una referencia

a esa instantánea. Si los archivos no se han modificado, para optimizar, Git almacena un enlace al archivo anterior idéntico que ya tiene almacenado:



Esta es la diferencia principal entre Git y prácticamente todos los demás VCSs. Hace que Git se parezca más a un mini sistema de archivos con algunas herramientas potentes construidas sobre él, más que a un VCS.

Por otra parte, **casi cualquier operación se puede realizar en local** sin acceder a otro ordenador de la red. La mayoría de los CVC no son así, y redunda en su velocidad y en la facilidad de trabajo sin necesitar estar conectado. Esto también significa que hay muy poco que no puedas hacer si estás desconectado o sin VPN (Virtual Private Network).

Git **mantiene la integridad mediante checksum** (suma de comprobación) Se trata de un número de 40 caracteres hexadecimales, generado mediante un algoritmo (SHA-1) que toma como entrada el código original, y que puede recalcularse en cualquier momento. Si hubiera una diferencia entre el checksum del repositorio y el obtenido en segunda instancia, es que los datos están corruptos.

En Git generalmente **sólo se añade información**, de manera que todo se puede deshacer cualquier cambio que se haya confirmado previamente.

Para poder trabajar con Git, es necesario, en primer lugar, descargarse el software de Git (lo hay para Linux, Windows y Mac) y abrirse una cuenta en https://github.com/

A partir de este momento, podemos configurar nuestro perfil adecuadamente, ya que, además de un CVS, funciona como un escaparate de nuestro trabajo (y el de toda la comunidad)

Una vez hecho esto, podemos:

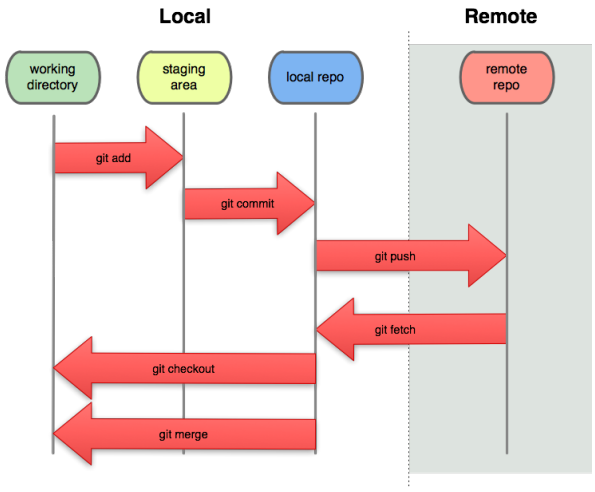
- Crear un repositorio para subir nuestros proyectos
- Navegar por los contenidos de otros autores
- Clonar otros repositorios disponibles con *git clone* (los bajamos así a nuestro equipo) y actualizarlos con *git pull* (poniendo al día los cambios)

Cuando trabajamos con un repositorio Git, hay tres estados principales en los que se pueden encontrar los archivos:

- preparado (staged): se ha marcado un archivo modificado en su versión actual para que vaya en la próxima confirmación
- modificado (modified): se ha modificado el archivo pero todavía no se ha confirmado a la base de datos
- confirmado (committed): los datos están almacenados de manera segura en la base de datos local

Esto nos lleva a las tres secciones principales de un proyecto de Git:

- el directorio de trabajo (working directory)
- el área de preparación (staging area)
- el directorio de Git (Git directory)



Para que un archivo quede subido al repositorio remoto, es necesario realizar sobre él 3 operaciones: add, commit y push

- **add**: añade al archivo al grupo de los que deben ser tenidos en cuenta en la próxima actualización
- **commit**: pasan al repositorio local, y quedan a disposición de ser subidos al repositorio
- **push**: son subidos de forma real al repositorio remoto

Una vez ejecutados los tres comandos, nuestros archivos estarán ya asegurados en el repositorio remoto.

Además de esta operatoria básica de subir y obtener información de Github, también se pueden realizar otras operaciones interesantes, como por ejemplo, **gestión de versiones** y realizar **ramificaciones** o branches.

Las versiones múltiples, nos permiten guardar imágenes de la evolución del software en distintas etapas de su evolución.

Para manejar diferentes versiones, es necesario etiquetarlas con el comando:

git tag –a *vX.X* –m "mensaje aclaratorio"

Para ver todas las versiones que tenemos:

git tag –n

Y para que se suban al repositorio:

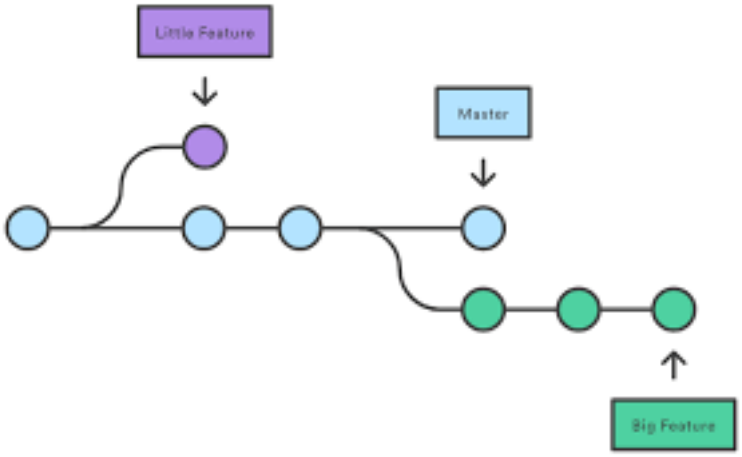
git tag -n

La ramificación permite el seguimiento de diferentes líneas de desarrollo. Siempre hay al menos una denominada **master**, que es la principal. A partir de ésta, se pueden crear otras adicionales, que eventualmente pueden volver a "fundirse" en la principal.

El comando para crear una rama y desplazar la referencia posterior a ella es:
git checkout –b *nombrerama*

Para combinar dos ramas en una (fundiendo la ramificación con su rama origen):

git merge *nombrerama (*y como siempre, git push)

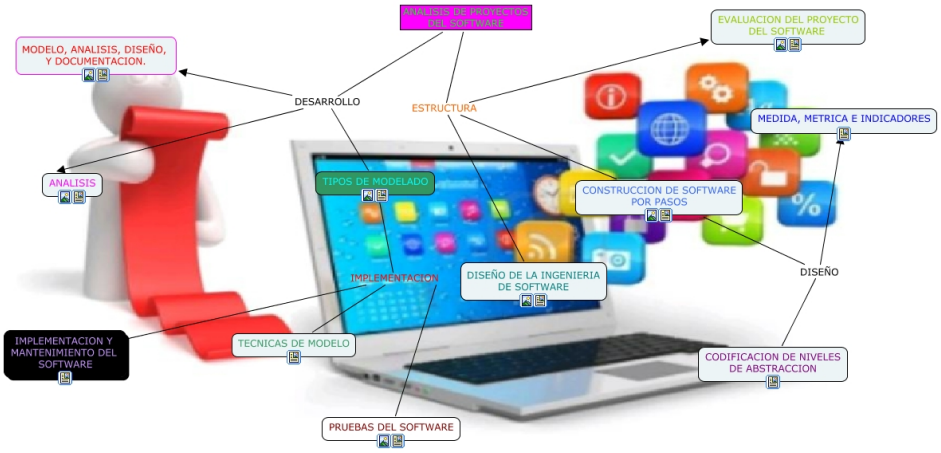


Como se muestra en la imagen, esta es la forma en que los desarrolladores hacen evolucionar el software en múltiples direcciones exploratorias.

11.4- DOCUMENTACIÓN

Por documentación se entiende el texto escrito que acompaña a un proyecto de cualquier tipo, en este caso de Software.

La documentación es un requisito importante en un proyecto comercial, ya que el cliente siempre va a solicitar que se documenten las distintas fases del proyecto. Podemos distinguir los siguientes tipos de documentación:



1- Documentación de las especificaciones

Este documento tiene como objeto asegurar que tanto el desarrollador como el cliente tienen la misma idea sobre las funcionalidades del sistema.

Es muy importante que esto quede claro y el documento esté consensuado, porque de lo contrario, el desarrollo de software no será aceptable.

En Ingeniería del Software existe una normativa referente a este tipo de documento: la norma IEEE 830 que recoge recomendaciones para la documentación de requerimientos del software, se indica que las especificaciones deben incluir:

- Introducción: fines y objetos del software
- Descripción de la información: se realiza una descripción detallada del problema, incluyendo el hardware y el software necesario
- Descripción funcional: descripción de cada función requerida en el sistema, incluyendo diagramas
- Descripción del comportamiento: comportamiento del software ante sucesos externos y controles internos
- Criterios de validación: documentación sobre límites de rendimiento, clases de pruebas, respuesta esperada del software, consideraciones especiales…

2- Documentación del diseño

En la fase de Diseño, se decide la estructura de datos a utilizar, la forma en que se van a implementar las distintas estructuras, el contenido de las clases, sus métodos y sus atributos, los objetos a utilizar. También se definen las funciones, sus datos de entrada y salida, qué tarea realizan, etc…

3- Documentación del código fuente

El proceso de documentación de código fuente, es uno de los aspectos más importantes de la labor de un programador.

Documentar el código sirve para explicar su funcionamiento, punto por punto, de forma que cualquier persona que lea el comentario, puede entender su finalidad.

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 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 en ningún caso 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 será ignorada por el compilador, sirve para indicar a los desarrolladores de código diferentes aspectos de éste que pueden ser útiles. En principio, los comentarios tienen dos propósitos diferentes:

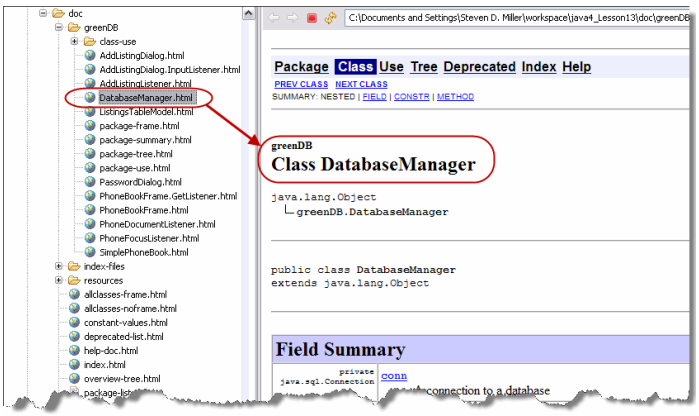
- Explicar el objetivo de las sentencias, de forma que el programador o programadora, sepa en todo momento la función de esa sentencia, tanto si lo diseñaron como si son otros los que quieren entenderlo o modificarlo.
- Explicar qué realiza un método, o clase, no cómo lo realiza, y en este caso, se trata 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 */

Hay que tener en cuenta siempre que si el código es modificado (por ejemplo refactorizado) también se deberán modificar los comentarios.

JavaDoc

Un tipo de comentarios muy interesante que se suele utilizar en Java, son los comentarios **JavaDoc**.



Los comentarios JavaDoc se escriben empezando por /** y terminando con */ y pueden ocupar varias líneas, en las cuales se sigue una estructura prefijada. Deben incorporarse, obligatoriamente:

- al principio de cada clase
- al principio de cada método
- al principio de cada variable de clase

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

Cuando se implementa una clase, se deben, por tanto, incluir comentarios. En el lenguaje Java, los criterios de documentación de clases, son los establecidos por JavaDoc.

Entre la información textual que debe incluir un comentario de clase deben ir, al menos, las etiquetas:

- **@autor** que identifica el nombre del autor o autora
- **@version**, la identificación de la versión y fecha.

Con el uso de los entornos de desarrollo, muchas veces las etiquetas se añaden de forma automática, estableciendo el @autor y la @version de la clase de forma transparente. También se suele añadir la etiqueta:

- **@see**, que se utiliza para referenciar a otras clases y métodos relacionados.

Dentro de la clase, también se documentan los constructores y los métodos, siempre que sean *public* o *protected*. 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

Otras etiquetas que se pueden usar dentro de una clase son:

- **@since**: fecha desde la que está presente la clase
- **@deprecated**: para indicar que el método está obsoleto

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

4- Documentación de usuario final

Es la documentación que se le entrega al usuario, tanto especializado, como no especializado. En ella se describirá como utilizar las aplicaciones del proyecto.

La documentación más típica de usuario final es el **manual**.

Un Manual de Usuario facilita el conocimiento detallado de la interfaz de usuario, de la operatoria de entrada y salida de datos, así como de los errores más comunes que se pueden presentar en el uso del software y como solucionarlos