EEDD04

Optimización y documentación.

# Refactorización.

Técnica que consiste en realizar pequeños cambios para mejorar su estructura sin que cambie el comportamiento.

Algunos cambios pueden ser:

* Código duplicado.
* Métodos demasiado largos.
* Clases muy grandes o con demasiados métodos.
* Exceso o falta de comentarios.

## Concepto.

* **Refactorización:** cambio hecho en el código para hacerlo más fácil de entender y modificar.
* **Campos encapsulados:** Se aconseja crear métodos getter y setter para modificar los valores de los atributos de las clases.
* **Refactorizar:** reestructurar el programa.

## Limitaciones.

Cualquier modificación que requiera cambios en una base de datos siempre va a ser limitada por la cantidad de interdependencias que hay.

También es difícil refactorizar cuando hay fallos de diseño.

No siempre es recomendable refactorizar.

## Patrones de refactorización habituales.

Algunos de los patrones más habituales son:

* **Renombrar:** cambiar nombres a otros más significativos.
* **Encapsular campos.**
* **Sustituir bloques de código por métodos:** si un bloque de código se repite, es recomendable crear un método con ese bloque de código.
* **Modificar la extensión del código:** hacerlo más corto si se puede.
* **Reorganización de código condicional complejo:** no tener ifs kilométricos.
* **Mover la clase** de un paquete a otro, o de un proyecto a otro.
* **Borrado seguro:** eliminar el código que ya no es necesario.
* **Extraer la interfaz:** crear un nuevo interfaz de los métodos public non-static.

## Refactorización en Eclipse.

Click secundario -> refactorizar. Ahí hay varias opciones.

## Analizadores de código.

Tienen como objetivo evaluar el código sin ejecutarlo. Un ejemplo es PMD para Java, o Sonarcube.

### Instalación de PMD.

Se puede descargar como plugin de NetBeans.

<http://plugins.netbeans.org/plugin/57270/easypmd>

### Configuración.

# Control de versiones.

Los típicos: GIT, CVS, Subversion, Mercurial… Uno recomendado es BitBucket, basado en GIT.

## Tipos de herramientas de control de versiones.

* **Sistemas locales:** Control de versiones donde las distintas versiones se guardan en distintos directorios. La gestión recae sobre el responsable del proyecto y no tiene herramientas de automatización.
* **Sistemas centralizados:** arquitectura cliente-servidor. Un único equipo tiene los archivos de las versiones y los clientes replican esa información.
* **Sistemas distribuidos:** Cada sistema guarda una copia de todas las versiones.

## Estructura de herramientas de control de versiones.

El sistema de control de versiones está formado por un conjunto de componentes:

* **Repositorio:** donde se almacenan los proyectos.
* **Módulo:** directorio específico del repositorio.
* **Revisión:** cada versión parcial o cambios en los archivos o repositorio completo.
* **Etiqueta:** Información textual que se añade a un conjunto de archivos.
* **Rama:** Revisiones paralelas de un módulo para efectuar cambios sin tocar la evolución principal.

Los servicios que típicamente se proporcionan son:

* **Creación de repositorios.**
* **Clonación de repositorios.**
* **Descarga de la información del repositorio principal al local.**
* **Carga de información al repositorio principal desde el local.**
* **Gestión de conflictos.**
* **Gestión de ramas.**
* **Información sobre registro de actualizaciones.**
* **Comparativa de versiones.**

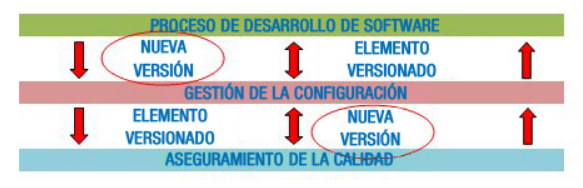
Las órdenes que se pueden ejecutar son:

* **Checkout:** Obtiene una copia del trabajo.
* **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.

### Repositorio

Almacenamiento central de las versiones.

## Gestión de versiones y entregas

* **Grafo de evolución simple:** los distintos cambios dan lugar a una secuencia lineal.
* **Variantes:** en este caso el grafo ya no es secuencial, sino que adopta forma de árbol.

Las terminologías que se usan para referirse al grafo son:

* Tronco: variante principal.
* Cabeza: última versión.
* Ramas (branches): variantes secundarias.
* Delta: el cambio de una revisión con respecto a la anterior.
* **Propagación de cambios:** cuando se tienen variantes que se desarrollan en paralelo, suele ser necesario aplicar un mismo cambio a varias variantes.
* **Fusión de variantes:** Cuando deja de ser necesario mantener un Branch, se junta con otra (merge).
* **Técnicas de almacenamiento:** hay varias formas de almacenas las distintas versiones:
  + **Deltas directos:** Se almacena la primera versión completa, y luego los cambios necesrios para construir cada nueva versión a partir de la anterior.
  + **Deltas inversos:** Se almacena la última versión y luego los cambios para reconstruir las versiones anteriores.
  + **Marcado selectivo:** se almacena el texto refundido de todas las versiones como una secuencia lineal.

## Herramientas de control de versiones:

* **SourceSafe:** forma parte del entorno de desarrollo de Visual Studio.
* **Visual Studio Team Foundation Server:** el sustituto de SourceSafe.
* **Darcs:** Sistema de control de versiones distribuido.
* **Git.**
* **Mercurial:** Permite que el desarrollo se haga distribuido.

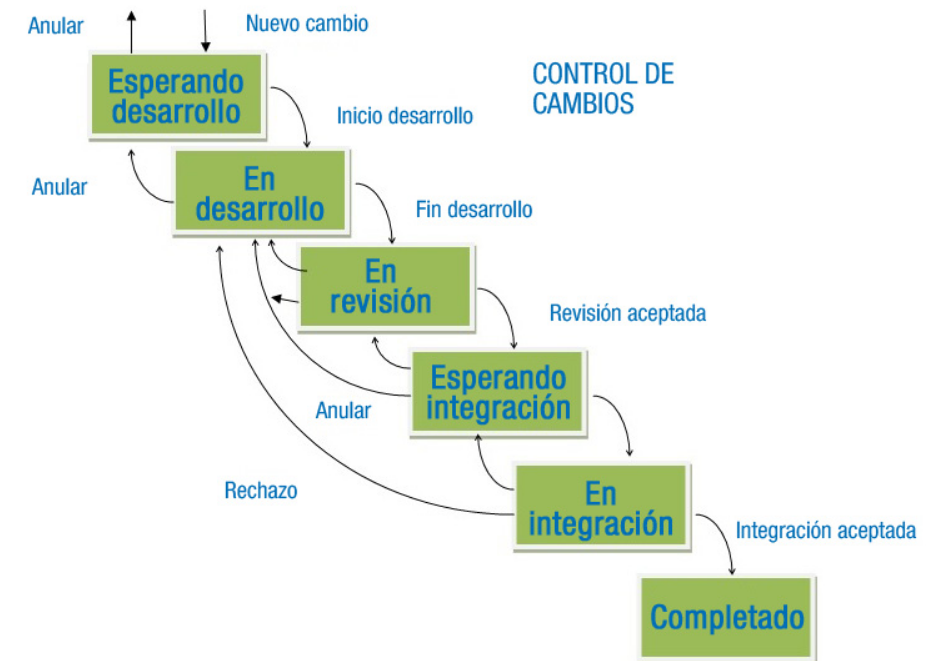
## Planificación de la gestión de configuraciones.

La gestión de configuraciones del software es un conjunto de actividades desarrolladas para gestionar los cambios a lo largo del ciclo de vida.

Se compone de 4 tareas básicas:

1. **Identificación:** establecer estándares de documentación y un esquema de identificación de documentos.
2. **Control de cambios:** evaluación y registro de todos los cambios que se hagan de la configuración.
3. **Auditorías de configuraciones:** sirve para garantizar que los cambios se han implementado correctamente.
4. **Generación de informes:** el software está compuesto por un conjunto de elementos que evolucionan de manera individual, y se debe garantizar la consistencia del conjunto del sistema.

## Gestión del cambio.



Pueden establecerse distintos tipos de control:

* **Control individual:** cuando un elemento está bajo control individual, el responsable puede generar documentación informal.
* **Control de gestión organizado:** hay un proceso de revisión y aprobación para cada cambio propuesto.
* **Control formal:** se realiza durante el mantenimiento. El impacto de cada tarea de mantenimiento se evalúa por el Comité de Control de Cambios que lo parueba.

# Documentación.

La documentación explica para qué sirve una clase, paquete, método etc… Es una parte fundamental.

## Uso de los comentarios.

Los comentarios tienen dos propósitos diferentes:

* Explicar el objetivo de las sentencias.
* Explicar qué realiza un método o clase, no cómo lo realiza.

# Documentación de clases

Hay formas de automatizar la documentación, entre ellas JavaDoc.

Para que JavaDoc pueda generar el HTML hay que hacer lo siguiente:

* Los comentarios se deben incorporar al principio de cada clase, método y variable de clase. Los comentarios serían así: /\*\* comentario \*/.
* Los comentarios pueden ser a nivel de clase, variable y método.
* Se genera para los métodos public y protected.

Hay una serie de etiquetas que fijan cómo se presentará la documentación:

|  |  |  |
| --- | --- | --- |
| Etiqueta y parámetros | Uso | Asociada a |
| @author nombre | Nombre del autor | Clase, interfaz |
| @version numero-versión | Indica el número de versión | Clase, interfaz |
| @since numero-versión | 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. |