UD 04 - OPTIMIZACIÓN Y DOCUMENTACIÓN

1. Refactorización

Refactorización: Cambio hecho en la estructura interna del software para hacerlo más fácil de entender y modificar sin que se altere su comportamiento externo (funcionalidad). Suelen ser cambios pequeños en el software que mejoran su mantenimiento (Ej.: "Extraer método", "Encapsular campos")

Refactorizar: Reestructurar el software aplicando refactorizaciones sin cambiar su comportamiento

La refactorización:

- Busca mejorar la estructura interna del código; limpiarlo sin introducir errores.
- Facilita que el software sea más fácil de entender y de modificar
- Que el mantenimiento sea más sencillo
- Que se encuentren errores
- Que el programa sea más rápido
- Utiliza una serie de patrones, de aplicación sobre el código fuente.

No es lo mismo que la optimización (que persigue una mejora del rendimiento aunque podría llegar a hacer el código más difícil de entender)

1.1. Problemas de la refactorización

La refactorización presenta problemas en:

- Las bases de datos: Tienen dificultades para ser modificadas debido a sus interdependencias y a conllevar modificación del esquema y migración de datos (muy costosa).
- Interfaces. Al modificar la estructura interna no se cambia comportamiento ni la interfaz. Pero si renombramos un método hay que modificar todos los que referencia a él lo cuál es un problema si la interfaz es pública. La solución es mantener las dos interfaces (nueva y vieja)
- Cambios en diseño difíciles de refactorizar como los debidos a errores de diseño o los que son de vital importancia en la aplicación.
- Si un código no funciona, no se refactoriza; se reescribe.

1.2. Patrones

- Renombrado: Cambiar nombre paquete, clase, método, atributo por otro más significativo
- Sustituir bloques de código por un método: Así cada vez que se quiera acceder al bloque de código solo hay que llamar a ese método
- Campos encapsulados: Mediante getters y setters
- Mover la clase: De un paquete a otro, de un proyecto a otro... Evitar duplicar código. Deben actualizarse las referencias a la clase.
- Borrado seguro: Cuando ya no sea necesario el elemento, que se hayan borrado todas las referencias a él en el proyecto.
- Cambiar los parámetros del proyecto: Añadir nuevos parámetros a un método, cambiar los modificadores de acceso.
- Extraer la interfaz
- Mover del interior a otro nivel: Se mueve una clase interna a un nivel superior en la jerarquía.

1.3. Analizadores de código

El analizador estático de código recibe directamente el código fuente y busca evaluarlo sin llegar a ejecutarlo. Intenta averiguar la funcionalidad del mismo y lo procesa dando sugerencias o posibles mejores.

- Tienen analizadores léxicos y sintácticos que procesan el código fuente y un conjunto de reglas que se deben aplicar sobre las estructuras.
- Las funciones de los analizadores son encontrar partes del código que puedan reducir el rendimiento, provocar errores en el software, tener excesiva complejidad, complicar el flujo de datos, crear problemas de seguridad.
- El análisis puede ser automático o manual. El automático lo realizan programas como el **Findbugs** de Netbeans.
- Analizadores como **EasyPMD, PMD** (nadie usa esa mierda, hoy en día SonarLint y SonarQube por favor) que detectan patrones que pueden ser posibles errores en tiempo de ejecución, unreacheable code, código susceptible de ser optimizado, expresiones lógicas que pueden ser simplificadas, malos usos del lenguaje...
- CPD: Forma parte del PMD. Busca código duplicado.

1.3.1. Uso

Los analizadores de código estático se suelen integrar en los Entornos de Desarrollo (muchas veces como plugins). En Netbeans por ejemplo podemos instalar EasyPMD.

En el caso del prehistórico Netbeans y el analizador PMD...

Como Herramientas > Ejecutar PMD se puede ejecutar el análisis y obtener el informe. Después el informe permite navegar por las clases y, en la línea que no cumple, se verá una marca PMD con un tooltip con la descripción del error.

En la configuración de los analizadores de código es posible definir las propias reglas sumándolas a las que el analizador propone por defecto.

Herramientas > Opciones > Varios > easyPMD.

- Con Enable Scan se habilita el escaneado automático del proyecto a intervalos regulares.
- En Rulestet (Manage Rules) pueden verse las existentes y activar-desactivar, añadir (Add standard) y personalizar (Add custom)
- En Reporting se puede establecer qué información visualizar (por cada regla puede verse descripción del problema y ejemplo de aplicación). Con las propias reglas puede usarse "Manage Rulesets" e importar ficheros XML o JAR con reglas.

1..4 Refactorización y pruebas

Refactorización y pruebas están fuertemente relacionadas en el **Test Driven Development**.

- Agilizar ciclo de escritura de código y realizar pruebas de calidad de los módulos
- Evitar conflictos y gasto excesivo entre developers y SQA (ideas y venidas del código...), en este caso el programador realiza las pruebas de unidad en su propio código y las implementa antes de escribir el código a ser probado.

Sería algo así

```
- Lee requerimiento
- Plantea las pruebas que va a tener que pasar el código a elaborar para ser
correcto
- Programa las pruebas
- Comienza a implementar la unidad de código para que pase esas pruebas
- Elabora pequeñas versiones que puedan ser compiladas y pasen por algunas de las
pruebas
- En cada cambio y recompilación reejecuta las pruebas de unidad tratando que cada
vez pase más hasta que no falle ninguna
```

La refactorización siguiendo TDD: Se refactoriza el código tan pronto como pasa las pruebas y hacerlo más claro. Podrían cometerse errores que provocarían que la unidad falle las pruebas. En reescrituras

sintácticas no es necesario correr el riesgo, las decisiones las toma un humano pero los detalles pueden quedar a cargo de programa que lo trate automáticamente.

1.5. Herramientas de ayuda a la refactorización

Herramientas de ayuda de netebans

- **Renombrar**: Actualiza todo el proyecto
- **Encapsular campos**: Genera getters y setters automáticamente y opcionalmente actualiza las referencias al código para usar esos getters y setters.
- **Cambiar parámetros del método**: Añadir o eliminar parámetros y cambiar el tipo
- **Eliminación segura**: Comprueba si hay referencias a elemento y si no la hay, lo borra. Si se borra con referencias, se advierte de que habrá errores.

(Eliminar parámetros del método, según el test, no seria un mecanismo de refactorizar)

2. Control de versiones

Es vital en el desarrollo un sistema de control de versiones porque este:

- Facilita al equipo de desarrollo su labor permitiendo que varios trabajen en mismo proyecto / mismos archivos
- Sitio central en el que almacenar el código fuente
- Historial de cambios realizados
- Volver a versiones estables previas del código
- Sobre ellos se construyen técnicas más sofisticadas como la Integración Continua

Versión: Forma particular de un elemento en un instante dado.
Revisión: Evolución del elemento en el tiempo en el tiempo.

Los IDEs proporcionan herramientas de control de versiones y facilitan el desarrollo en equipo de las aplicaciones.

En un instante dado pueden coexistir varias versiones y hay que disponer de métodos para designar las versiones de forma sistemática u organizada.

Podemos mencionar CSV y Subversión (O al menos eso diría alguien que vive en el año 2000...). Subversion sería el sucesor natural de CVS.

2.1. Estructura de las herramientas de control de versiones CSV (Concurrent Version System)

Sistema de mantenimiento de código fuente (grupos de archivos en general) para desarrolladores que trabajan y modifican concurrentemente ficheros organizados en proyectos.

- Arquitectura cliente-servidor: Servidor guarda versión actual del proyecto y su historia. Clientes conectan al servidor para sacar copia completa del proyecto trabajar en esa copia e ingresar sus cambios.
- Servidor y cliente se conectan por internet aunque podrían estar en la misma máquina. Servidor suele tener sistema operativo UNIX.
- Se permitiría el acceso de lectura anónimo (sin contraseña)

Componentes:

- **Repositorio**: Lugar de almacenamiento de los datos de los proyectos. Suele ser un directorio.

Centraliza los componentes, incluyendo versiones. Se ahorra espacio de almacenamiento (no se guarda por duplicado los elementos comunes a varias versiones). Se facilita el almacenaje de información de la evolución del sistema. Almacena los datos pero también información sobre versiones, temporización. Al usar sistema de control de versiones trabajamos de forma local,

sincronizándonos con el repositorio. Haciendo los cambios en nuestra copia local, se acomete el cambio en el repositorio.

- **Módulo**: Directorio específico del repositorio. Parte del proyecto o proyecto por completo.
- **Revisión**: Cada versión parcial o cambios en archivos o repositorio completo. La evolución del sistema se mide en revisiones.
- **Etiqueta**: Información textual que se añade a un conjunto de archivos o módulo completo
- **Rama**: Revisiones paralelas de un módulo para efectuar cambios sin tocar evolución principal. Pruebas, mantener cambios en versiones antiguas.

Órdenes:

- `checkout`: Copia del trabajo para trabajar en 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

Repositorio CSV

Para conectar con un repositorio en Netbeans -> Team -> CVS -> Checkout
Se introduce localización del repositorio (CVSROOT). Soporta diferentes según sea local o remoto y el método usado para conectarse a él.

Métodos

- `pserver`: contraseña de servidor remoto
- `ext`: acceso usando Remote Shell (RSH) o Secure Shell (SSH)
- `local`: acceso a repositorio local
- `fork`: acceso a repositorio local usando protocolo remoto

En módulos a extraer se especifica el módulo que se quiere extraer (o se hace con Examinar), indicando en Carpeta local donde se quieren extraer.

Para importar hacia el repositorio remoto se selecciona el proyecto no versionado y se elije Equipo - CVS - Importar al depósito. Se especifica la localización con CVSROOT, variando según el método usado para conectarse, y la carpeta local en la que se quiere colocar. Se especifica la localización del repositorio escribiendola en Carpeta del depósito (o con examinar para ir a algun lado dentro del repositorio)

2.2. Herramientas de control de versiones

- CVS.
- Subversion: Los archivos no tienen número de versión independiente. Todo el repositorio tiene único número de versión que identifica el estado común de todos en un instante. Se accede al repositorio a través de redes. Modifican el mismo conjunto de datos. Control de versiones.
- SourceSafe (Microsoft Visual Studio)
- Visual Studio Team Foundation Server. Sustituto de Source Safe. Ofrece código fuente, recolección de datos, seguimiento de proyectos.
- Darcs: Gestión de versiones distribuido. Cada repositorio es una rama en sí misma, independiente del servidor central.
- Git
- Mercurial: Programa de línea de comandos. Desarrollo distribuido. Capacidades avanzadas de ramificación e integración. Opciones dadas a su motor hg. Gran rendimiento y escalabilidad.

Más destacados en IDEs con plugins disponibles: CSV, Subversion, Mercurial. (Y Git ni lo menciona, hay que joderse). en Netbeans y Eclipse. Y otros como Visual Studio Team Foundation en Microsoft Visual Studio.

2.3. Planificación de la gestión de configuraciones

La configuración es una combinación de versiones particulares de componentes que hacen un sistema consistente. Desde la evolución en el tiempo es el conjunto de versiones de los componentes en un instante dado.

Gestión de configuraciones de software (GCS): Conjunto de actividades desarrolladas para gestionar cambios a lo largo del ciclo de vida. Su planificación se regula en estándar IEEE 828.
Se refiere a la evolución de todo un conjunto de elementos.

Debe disponerse de un método que designe las configuraciones de forma sistemática y planificada para facilitar desarrollo de software de forma evolutiva, por cambios sucesivos.

Tareas de la gestión de configuraciones de software

1. **Identificación** (Estándares de documentación. Esquema de identificación)
2. **Control de cambios** (Evaluación y registro de todos los cambios en la configuración)
3. **Auditorías de configuraciones**: Junto con revisiones técnicas formales garantiza que el cambio se ha implementado correctamente
4. **Generación de informes**: Garantizar la consistencia del conjunto del sistema (ya que los componentes evolucionan de forma individual)

Actividades de la planificación gestión de configuraciones de software

- Introducción (propósito, alcance, terminología)
- Gestión de GCS (organización, responsabilidades, políticas, directivas, procedimientos)
- Actividades GSC (identificación de configuración, control de configuración)
- Agenda GSC (coordinación con otras actividades)
- Recursos GCS (herramientas, recursos físicos y humanos)
- Mantenimiento GCS

2.4. Gestión del cambio

- Debe haber un control para que el desarrollo sea razonable (no es posible que cualquier componente del equipo de desarrollo pueda realizar cambios e integrarlos sin ningún control). --> GARANTIZAR LINEA BASE PARA CONTINUAR EL DESARROLLO, con controles al desarrollo y la integración.

Tipos de control:

1. **Control individual**: Antes de aprobarse un nuevo elemento, el programador cambia documentación cuando se requiere. Registra de forma informal el cambio, aunque no sea documento formal.
2. **Control de gestión u organizado**: Con procedimiento de revisión y aprobación de cada cambio propuesto. El cambio es registrado formalmente.
3. **Control formal, durante el mantenimiento**. El impacto de cada tarea de mantenimiento se evalúa por Comité de Control de Cambios, que aprueba las modificaciones de la configuración software.

2.5. Gestión de versiones y entregas (CICD)

De versiones... (""Continuous Integration"")

La evolución de las versiones pueden representarse en forma de grafo donde los nodos son versiones y los arcos son creación de una versión a partir de otra.

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

Variantes: Cuando hay varias versiones del componente el grafo no es lineal sino con forma de árbol. La numeración de versiones tiene dos niveles: Primer número (variante, línea de evolución) y segundo número (versión particular, revisión) a lo largo de dicha variante.

Terminología de los elementos del grafo

- Tronco (trunk): Variante principal
- Cabeza (head): Última versión del tronco
- Ramas (branches): Variantes secundarias
- Delta: Cambio de una revisión respecto a la anterior

Propagación de cambios: Aplicar mismo cambio a varias variables que están en paralelo.

Fusión: Dejar de mantener una rama independiente fundiéndola con otra (merge)

Técnicas de almacenamiento: Ya se ha comentado que suelen tener el mismo contenido las distintas ramas, no repitiendo esos datos comunes para no desaprovechar espacio.

- **Deltas directos**: Primera versión completa y luego cambios mínimos necesarios para reconstruir cada nueva versión a partir de la anterior.
- **Deltas inversos**: Última versión completa y cambios mínimos para reconstruir versión anterior a partir de la siguiente.
- **Marcado selectivo**: Texto refundido de todas las versiones como secuencia lineal. Marcando cada sección del conjunto con los números de versiones que le corresponden.

De entregas... (""Continuous Deployment"")

La entrega es una instancia del sistema que se distribuye a usuarios externos al equipo de desarrollo.

La planificación de la entrega se ocupa de cuándo emitir una versión del sistema como una entrega. La entrega está compuesta del conjunto de programas ejecutables, los archivos de configuración, los archivos de datos, el instalador, documentación, embalaje, publicidad...

2.6. Herramientas CASE para la gestión de configuraciones

CASE Compute Aided Software Engineering.

Abiertos: Seguimiento configuraciones (bug-tracking) como `Bugzilla`. Seguimiento versiones: `RCS`, `RVS`. Construcción del sistema `Make`, `Imake`

Integrados: Para la gestión de versiones, construcción del sistema y seguimiento de configuraciones (cambios). Proceso de control de cambios unificado de `Rational`, basado en Gestión de configuraciones de `ClearCase` para construcción y gestión de versiones y `ClearQuest` para seguimiento de cambios. Los entornos de Gestión de Configuraciones Integrado ofrecen la ventaja de intercambio de datos sencillos y el entorno ofrece Base de datos de gestión de configuraciones integrada.

3. Documentación

- Permite explicar su funcionamiento, punto por punto para que cualquier persona que lea el comentario pueda entender su finalidad.
- Es fundamental para detección de errores y mantenimiento posterior.
- Da explicaciones de la función del código, de las características de un método. . No debe explicar qué hace el código sino por qué lo hace. (Finalidad de clase, de paquete, qué hace un método, para qué sirve una variable, qué se espera, qué algoritmo se usa, por qué así y node otra forma, qué se podría mejorar)

3.1. Uso de comentarios

Anotación (Entre / * /) que se realiza en el código y que el compilador ignora. Sirve para indicar aspectos a los desarrolladores:

- Explicar objetivo de las sentencias
- Explicar qué realiza (no cómo lo realiza)

En Java los que se usan para explicar qué hace un código se llaman Comentarios Javadoc /** */
Los comentarios "son obligatorios" con JavaDoc. Deben incorporarse al principio de cada clase, método y variable de clase.

Si el código es modificado, los comentarios deben también.

3.2. Alternativas a la falta de buena documentación

Alternativas para documentar código son los comentarios. **JavaDoc, SchemeSpy, Doxygen...** permiten producir una documentación actualizada, precisa y utilizable en línea. (Con SchemeSpyy Doxygen se incluyen modelos de bases de datos gráficos y diagramas)

Con comentarios en código difícil de entender y documentación generada por estas herramientas, se puede ayudar al nuevo desarrollador a entenderlo.

3.3. Documentación de código con JavaDoc

DOCUMENTACIÓN DE CLASES:
Criterios de documentación establecidos por JavaDoc

Comenzar y terminar así
/**
*/

Incluir al menos `@author` y `@version`. Con los IDE se añaden de forma automática.
`@see` para referenciar a otras clases y métodos. También está `@since`, fecha desde la que está presente la clase.

DOCUMENTACIÓN DE MÉTODOS:
`@param` seguido del nombre, parámetro que se le pasa
`@return` si no es `void`, se indica lo que devuelve
`@exception` nombre de la excepción, especificando cual pueden lanzarse. Etiqueta antigua en versiones de Javadoc, se usa para describir excepciones que son subclases de `Throwable` como `RuntimeException` o `IOException`...
`@throws`, nombre de la excepción, más reciente y especificando CUALES excpeciones (en plural) pueden lanzarse
`@deperecated` método obsoleto

DOCUMENTACIÓN DE ATRIBUTOS:
Pueden contener comentarios aunque no hay etiquetas obligatorias en JavaDoc.