PAC 2. UF2.

Optimización y documentación.

Aníbal Santos Gómez

1 - ¿Qué es la refactorización?

La refactorización es un proceso mediante el cual el programador optimiza el código del software realizado previamente, realizando cambios en la estructura interna sin afectar al comportamiento final del producto desarrollado.

Su objetivo fundamental es limpiar el código para que pueda ser fácil de mantel, es decir, para permitir una mejor compresión. La modificación realizada no alterará la ejecución ni los resultados conseguidos del software.

Es un proceso que puede resultar tedioso, por lo que para evitarlo en la medida de lo posible es necesario hacer un buen planteamiento del diseño del software. Modularizando lo máximo posible y encapsulando los componentes así como la lógica del programa lo permita, para optimizar su mantenimiento.

El diseño previo a la codificación del futuro producto es clave para evitar una refactorización larga y tediosa.

2 - ¿Cuándo consideras que es necesario refactorizar?

La refactorización se debe llevar a cabo a medida que desarrollamos el software, ya que encontraremos puntos en el desarrollo que se desconsolarán o simplemente que las finalidades del producto puedan ampliarse o complicarse. Llegado a este punto es importante el planteamiento de una refactorización del software que estemos realizando, y llevarla a cabo por la totalidad del equipo técnico, para que pueda ser coherente y evitemos así quebraderos de cabeza que acabarán siendo improductivos.

Pueden darse síntomas de refactorización en cualquier fase del desarrollo y es necesario que sepamos apreciar dichos síntomas, conocidos también como bad smells.

3 - ¿Qué son los bad smells?

Los bad smells son indicios de que el sistema no funciona como debería o como es de esperar. Los bad selles más típicos son:

- Código duplicado. Es una de las principales razones, aquí debemos unificar nuestro código.

- Métodos muy largos. Puede darse que tengamos métodos que alberguen otros métodos más pequeños, llegados a este punto deberemos dividirlos para poder reutilizarlos.

- Clases grandes. Esto dará lugar a tener muchos métodos y atributos. Siempre es mejor realizar clases más pequeñas y concretas, al igual que las funciones y métodos.

- Lista de parámetros extensa. Las funciones al igual que lo anteriormente dicho de las clases deben tener el mínimo de parámetros posible o tendremos un problema de encapsulación de datos.

- Cambio divergente. Es una situación a evitar, donde cometeremos fallos graves en las clases modificando su contenido y que podrá afectar al resto del programa.

- Cirugia a tio pistola. Es lo que se conoce poner un parche sobre otro parche, al final la lógica del programa acabará fallando cuando un método, una función o una clase no sean coherentes.

- Envidia de funcionalidad. Se da cuando un método usa más elementos de otra clase que de la suya propia.

- Clase de solo datos. Solo tiene atributos y métodos de acceso. No es habitual.

- Legado de rechazo. Subclases que usan características de la superclase, que puede dar lugar a un error en la jerarquía de clases.

4 - ¿Cuáles son las partes de un software más conflictivas a la hora de refactorizar?

Las partes más conflictivas a la hora de refactorizar serán todas aquellas clases, funciones o métodos que trabajen a su vez con otras partes similares y que se extiendan en una lógica larga y tediosa, como por ejemplo multitud de controles de flujo, o funciones recargadas de métodos que a su vez sean de las que dependen otras funciones.

Puede suponer un verdadero quebradero de cabeza por falta de diseño previo.

Todas aquellas partes que manejen datos o funcionalidades primarias o clave en la funcionalidad del software son las que entrarán en conflicto con el proceso de refactorización.

Deberemos prestar especial atención y sobre todo ir refactorizando en breves periodos de tiempo para evitar una sobrecarga y que nos juntemos finalmente en una situación en la que tengamos demasiado código espagueti.

5 - Nombra los métodos de refactorización que podemos encontrar en Eclipse.

Dentro de refactor tenemos los siguientes elementos más comunes:

- Rename.
- Move.
- Extract Constant.
- Extract Local Variable.
- Convert Local Variable to Field.
- Extract Method.
- Change Method Signature.
- Inline.
- Member Type to top Level.

- Extract Interface.
- Extract Superclass.
- Convert Anonymous class to Nested.

6 - ¿Qué es el control de versiones?

El control de versiones es un sistema que permite el registro de todos los cambios realizados en la estructura de directorios y el contenido de archivos de un software a lo largo de su desarrollo.

Es una gestión de los diversos cambios que se realizan sobre los elementos de algún producto o configuración del mismo. Una versión es el estado en que se encuentra dicho producto en un momento dado de su desarrollo o modificación.

Estos sistemas facilitan la administración de las distintas versiones de cada producto desarrollado, así como las posibles especializaciones realizadas. Existen numerosos sistemas de control de versiones como Subversion, SourceSafem Bazaar, o Git.

7 - Describe en qué consisten los siguientes términos del control de versiones:

A. **Repositorio.**

   Es el lugar donde se almacenan los datos y cambios realizados, es la ruta donde tendremos alojados nuestros archivos.

B. **Revisión.**

   Es una versión concreta de los datos almacenados. La última versión se denomina head.

C. **Etiquetar.**

Es una acción para localizar y recuperar una versión concreta en un momento determinado.

D. **Trunk.**

Es el tronco del proyecto. Por lo general se desarrolla en árbol, Trunk es el tronco del árbol y luego tenemos ramas para según que propósitos donde se va desarrollando.

E. **Crear un branch.**

Branch es una rama del proyecto, por lo general Trunk suele ser master, y las ramas se van desarrollando e integrando en master conforme se avanza en la funcionalidad del programa.

F. **Checkout.**

En Git por ejemplo checkout cambia entre ramas o crea ramas. En otros sistemas puede suponer la copia del proyecto en el repositorio local.

G. **Commit.**

Commit confira los cambios en el repositorio local para integrarlos posteriormente en el remoto con un push (Git).

H. **Importación.**

Supone la subida de carpetas y archivos al repositorio, se realiza después de un commit. En git se denomina push.

I. **Update.**

Se realiza cuando queremos integrar cambios realizados en el repositorio. En git se denomina pulo y sirve para traernos el trabajo realizado en otras ramas o en master (trunk).

J. **Merge.**

Se lleva a cabo para fusionar cambio, o ramas para unir elementos entrelazados.

K. **Conflicto.**

Ocurre cuando se cambia de un archivo a otro y no se actualiza y se realizan cambios sobre el mismo archivo.

8 - ¿Qué es subversion? ¿Cómo funciona el ciclo de vida de esta herramienta?

Es una herramienta de control de versiones open source basada en un repositorio. Utiliza el concepto de revisión para guardar los cambios producidos en el repositorio. Entre dos versiones solo guarda el conjunto de modificaciones, optimizando el uso de espacio en disco.

Subversion puede acceder al repositorio a través de redes lo que permite ser usado por personas que se encuentran en distintas computadoras.

El proyecto tendrá que verse como un árbol con su trunk (master), que será la línea principal y sus branch (ramas), que añaden o corrigen nuevas funciones, (lo ideal es desarrollar en ramas e integrar en master. Solo realizaríamos integración en master cuando está comprobado que la funcionalidad del código de una rama es correcta. Está mucho más indicado desarrollar nunca directamente en trunk o master). Y sus tags que marcan situaciones importantes o versiones acabadas (logs).

9 - ¿Qué tipo de documentación se realiza de un proyecto?

La documentación es el texto escrito que acompaña al proyecto. Existen tipos de documentación:

- Documentación de especificaciones, que sirve para comprobar que las ideas del desarrollador, como las del cliente son las mismas, ya que sino el proyecto no será aceptable.

- Documentación del diseño, aquí se describe la estructura interna del programa, formas de implementación, contenido de las clases, métodos, etc…

- Documentación del código fuente, que será irá desarrollando conforme se codifica, y consiste en ir comentando el código por partes.

- Documentación de usuario final, que no es otra más que la que se entrega al cliente y donde se describe como usar las aplicaciones del proyecto

10 - ¿Qué es JavaDoc?

Es una herramienta de Orcale, que sirve para extraer y generar documentación básica de APIs para el programador a partir del código fuente en formato HTML.

Javadoc es el estándar de la industria para documentar clases de Java, la mayor parte de los IDEs los generan automaticamente.

La escritura de comentarios se realizará mediante comentarios en línea //, comentarios de tipo C /* …. */ y comentarios de documentación de Javadoc /***….*/.

Además se utilizarán etiquetas de documentación precedidas por el carácter @, @author, @version, @see, @param, @return, @throws, @since.

Un ejemplo sería el siguiente:

```java
 /**
  * Inserta un título en la clase descripción.
  * Al ser el título obligatorio, si es nulo o vacío se lanzará
  * una excepción.
  *
  * @param titulo El nuevo título de la descripción.
  * @throws IllegalArgumentException Si titulo es null, está vacío o contiene sólo espacios.

  */
 public void setTitulo (String titulo) throws IllegalArgumentException
 {
   if (titulo == null || titulo.trim().equals(""))
   {
       throw new IllegalArgumentException("El título no puede ser nulo o vacío");
   }
   else
   {
       this.titulo = titulo;
   }
 }
```