Entornos de desarrollo

Optimización y Documentación

Índice

1. REFACTORIZACIÓN

La refactorización es una disciplina técnica, que consiste en realizar pequeñas transformaciones en el código de un programa, para mejorar la estructura sin que cambie el comportamiento ni funcionalidad de este. 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 mejora el diseño del software, hace que el software sea más fácil de entender, hace que el mantenimiento del software sea más sencillo, la refactorización nos ayuda a encontrar errores y a que nuestro programa sea más rápido.

Cuando se refactoriza se está mejorando el diseño del código después de haberlo escrito. Podemos partir de un mal diseño y, aplicando la refactorización, llegaremos a un código bien diseñado.

EJEMPLOS: Mover una propiedad desde una clase a otra. Convertir determinado código en un nuevo método, etc.

La acumulación de todos estos pequeños cambios puede mejorar de forma ostensible el diseño.

1.1 Concepto

El concepto de refactorización de código se base en el concepto matemático de factorización de polinomios.

Podemos definir el concepto de refactorización de dos formas:

- **Refactorización**: Cambio hecho en la estructura interna del software para hacerlo más fácil de entender y fácil de modificar sin modificar su comportamiento.

  Ejemplos de refactorización es "Extraer Método" y "Encapsular Campos". La refactorización es normalmente un cambio pequeño en el software que mejora su mantenimiento.

- **Campos encapsulados**: Se aconseja 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.

- **Refactorizar**: Reestructurar el software aplicando una serie de refactorizaciones sin cambiar su comportamiento.

Refactorizar no es lo mismo que Optimizar:

En ambos procesos, se pretende mejorar la estructura interna de una aplicación o componente, sin modificar su comportamiento. Sin embargo:

Refactorizar: consiste en **hacer el software más fácil de entender y modificar**. Se pueden hacer muchos cambios en el software que pueden hacer algún pequeño cambio en el comportamiento observable. Solo los cambios hechos para hacer el software más fácil de entender son refactorizaciones.

Optimizar: consiste en **mejorar el rendimiento**. Por ejemplo, mejorar la velocidad de ejecución, aunque el código sea más difícil de entender.

1.2 Limitaciones

La refactorización es una técnica lo suficientemente novedosa para conocer cuáles son los beneficios que aporta, pero falta experiencia para conocer el alcance total de sus limitaciones. Se ha constatado que la refactorización presente problemas en algunos aspectos del desarrollo.

Áreas problemáticas de la refactorización:

- **Las bases de datos**.
  Una base de datos presenta muchas dificultades para poder ser modificada, dado la gran cantidad de interdependencias que soporta. Cualquier modificación que se requiera de las bases de datos, incluyendo modificación de esquema y migración de datos, puede ser una tarea muy costosa.
  Entonces, la refactorización de una aplicación asociada a una base de datos siempre será limitada, ya que la aplicación dependerá del diseño de la base de datos.

- **Las interfaces**.
  Cuando refactorizamos, estamos modificando la estructura interna de un programa o de un método. El cambio interno no afecta al comportamiento ni a la interfaz.
  Sin embargo, si renombramos un método, hay que cambiar todas las referencias que se hacen a él. Siempre que se hace esto se genera un problema si es una interfaz pública. Una solución es mantener las dos interfaces, la nueva y la vieja, ya que, si es utilizada por otra clase o parte del proyecto, no podrá referenciarla.

Hay determinados cambios en el diseño que son difíciles de refactorizar. Es muy difícil refactorizar cuando hay un error de diseño o no es recomendable refactorizar, cuando la estructura a modificar es de vital importancia en el diseño de la aplicación.

Hay ocasiones en las que no debería refactorizar en absoluto. Nos podemos encontrar con un código que, aunque se puede refactorizar, sería más fácil reescribirlo desde el principio. Si un código no funciona, no se refactoriza, se reescribe.

1.3 Patrones de refactorización más habituales

En el proceso de refactorización, se siguen una serie de patrones preestablecidos, los más comunes son los siguientes:

- **Renombrado** (rename): este patrón nos indica que debemos cambiar el nombre de un paquete, clase, método o campo, por un nombre más significativo.
- **Sustituir bloques de código por un método**: este patrón nos aconseja sustituir un bloque de código, por un método. De esta forma, cada vez que queramos acceder a ese bloque de código, bastaría con invocar al método.
- **Campos encapsulados**: se aconseja 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.
- **Mover la clase**: si es necesario, se puede mover una clase de un paquete a otro, o de un proyecto a otro. La idea es no duplicar código que ya se haya generado. Esto impone la actualización en todo el código fuente de las referencias a la clase en su nueva localización.
- **Borrado seguro**: se debe comprobar, que cuándo un elemento del código ya no es necesario, se han borrado todas las referencias a él que había en cualquier parte del proyecto.
- **Cambiar los parámetros del proyecto**: nos permite añadir nuevos parámetros a un método y cambiar los modificadores de acceso.
- **Extraer la interfaz**: crea una nueva interfaz de los métodos public non-static seleccionados en una clase o interfaz.
- **Mover del interior a otro nivel**: consiste en mover una clase interna a un nivel superior en la jerarquía.

1.4 Analizadores de código

Cada IDE incluye herramientas de refactorización y analizadores de código. En el caso se software libre, existen analizadores de código que se pueden añadir como complementos a los entornos de desarrollo. Respecto a la refactorización, los IDE ofrecen asistentes qué de forma automática y sencilla, ayudan a refactorizar el código.

El análisis estático de código es un proceso que tiene como objetivo, **evaluar el software, sin llegar a ejecutarlo**. Esta técnica se va a aplicar directamente sobre el código fuente, para poder obtener información que nos permita mejorar la base de código, pero sin que se modifique la semántica.

Los analizadores de código son las herramientas encargadas de realizar esta labor. El analizador estático de código recibirá el código fuente de nuestro programa, lo procesará intentando averiguar la funcionalidad del mismo, y nos dará sugerencias, o nos mostrará posibles mejoras.

Los analizadores de código incluyen analizadores léxicos y sintácticos que procesan el código fuente y de un conjunto de reglas que se deben aplicar sobre determinadas estructuras. Si el analizador considera que nuestro código fuente tiene una estructura mejorable, nos lo indicará y también nos comunicará la mejora a realizar.

Las principales funciones de los analizadores es encontrar partes del código que puedan reducir el rendimiento, provocar errores en el software, tener una excesiva complejidad, complicar el flujo de datos, crear problemas de seguridad.

El análisis puede ser automático o manual.

El automático, los va a realizar un programa, que puede formar parte de la funcionalidad de un entorno de desarrollo, por ejemplo, el **pmd** en Eclipse.

El manual es realizado por una persona.

El análisis automático reduce la complejidad para detectar problemas de base en el código, ya que los busca siguiendo una serie de reglas predefinidas.

El análisis manual, se centra en apartados de nuestra propia aplicación, como comprobar que la arquitectura de nuestro software es correcta.

Tomando como base el lenguaje de programación Java, nos encontramos en el mercado un conjunto de analizadores disponibles:

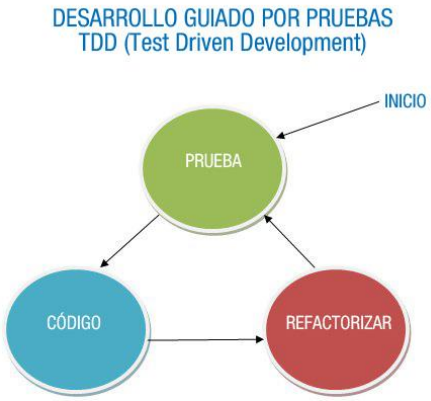
- **PMD**. Esta herramienta basa su funcionamiento en detectar patrones, que son posibles errores en tiempo de ejecución, código que no se puede ejecutar nunca porque no se puede llegar a él, código que puede ser optimizado, expresiones lógicas que pueden ser simplificadas, malos usos del lenguaje, etc.

  Más información en: https://github.com/pmd

- **FindBugs**. Es una herramienta de detección de defectos de código abierto diseñada para encontrar errores en los programas Java. FindBugs está buscando instancias de código que probablemente sean errores llamados "patrones de error". FindBugs utiliza el análisis sintáctico para examinar el código haciendo coincidir los códigos de bytes con una lista de más de 200 patrones de errores, como las referencias erróneas de punteros nulos, los bucles recursivos infinitos, los malos usos de las bibliotecas de Java y los puntos muertos.

  Más información en: http://findbugs.sourceforge.net/

1.5 Refactorización y Pruebas



En la actualidad, la refactorización y las pruebas son dos aspectos del desarrollo de aplicaciones, que, por sus implicaciones y su interrelación, se han convertido en conceptos de gran importancia para la industria. Muchas cuestiones y problemas siguen sin ser explorados en estos dos ámbitos. Uno de los enfoques actuales, que pretende integrar las pruebas y la refactorización, es el Desarrollo Guiado por Pruebas (TDD, Test Driven Development).

Con el Desarrollo Guiado por Pruebas (TDD), se propone agilizar el ciclo de escritura de código, y realización de pruebas de unidad. Cabe recordar, que el objetivo de las pruebas de unidad, es comprobar la calidad de un módulo desarrollado. Existen utilidades que permiten realizar esta labor, pudiendo ser personas distintas a las que los programan, quienes los realicen. Esto provoca cierta competencia entre los programadores de la unidad, y quienes tienen que realizar la prueba. El proceso de prueba supone siempre un gasto de tiempo importante, ya que el programador realiza revisiones y depuraciones del mismo antes de enviarlo a prueba. Durante el proceso de pruebas hay que diseñar los casos de prueba y comprobar que la unidad realiza correctamente su función. Si se encuentran errores, éstos se documentan, y son enviados al programador para que lo subsane, con lo que debe volver a sumergirse en un código que ya había abandonado.

Con el Desarrollo Guiado por Pruebas, la propuesta que se hace es totalmente diferente. El programador realiza las pruebas de unidad en su propio código, e implementa esas pruebas antes de escribir el código a ser probado.

Cuando un programador recibe el requerimiento para implementar una parte del sistema, empieza por pensar el tipo de pruebas que va a tener que pasar la unidad que debe elaborar, para que sea correcta. Cuando ya tiene claro la prueba que debe de pasar, pasa a programar las pruebas que debe pasar el código que debe de programar, no la unidad en sí. Cuando se han implementado las pruebas, se comienza a implementar la unidad, con el objeto de poder pasar las pruebas que diseñó previamente.

Cuando el programador empieza a desarrollar el código que se le ha encomendado, va elaborando pequeñas versiones que puedan ser compiladas y pasen por alguna de las pruebas. Cuando se hace un cambio y vuelve a compilar también ejecuta las pruebas de unidad. Y trata de que su programa vaya pasando más y más pruebas hasta que no falle en ninguna, que es cuando lo considera listo para ser integrado con el resto del sistema.

Para realizar la refactorización siguiendo TDD, se refactoriza el código tan pronto como pasa las pruebas para eliminar la redundancia y hacerlo más claro. Existe el riesgo de que se cometan errores durante la tarea de refactorización, que se traduzcan en cambios de funcionalidad y, en definitiva, en que la unidad deje de pasar las pruebas. Tratándose de reescrituras puramente sintácticas, no es necesario correr ese riesgo: las decisiones deben ser tomadas por un humano, pero los detalles pueden quedar a cargo de un programa que los trate automáticamente.

1.6 Herramientas de ayuda a la refactorización

Los entornos de desarrollo actuales nos proveen de una serie de herramientas que nos facilitan la labor de refactorizar nuestro código. En puntos anteriores, hemos indicado algunos de los patrones que se utilizan para refactorizar el código. Esta labor se puede realizar de forma manual, pero supone una pérdida de tiempo, y podemos inducir a redundancias o a errores en el código que modificamos.

En el Entorno de Desarrollo Netbeans, la refactorización está integrada como una función más, de las utilidades que incorpora.

A continuación, vamos a usar los patrones más comunes de refactorización, usando las herramientas de ayuda del entorno.

- **Renombrar**. Ya hemos indicado en puntos anteriores, que podemos cambiar el nombre de un paquete, clase, método o campo para darle un nombre más significativo. Netbeans nos permite hacerlo, de forma que actualizará todo el código fuente de nuestro proyecto donde se haga referencia al nombre modificado.
- **Introducir método**. Con este patrón podemos seleccionar un conjunto de código, y reemplazarlo por un método.
- **Encapsular campos**. Netbeans es capaz de generar de forma automática métodos getter y setter para un campo, y opcionalmente actualizar todas las referencias al código para acceder al campo, usando los métodos getter y setter.

2. DOCUMENTACIÓN

El proceso de documentación de código es uno de los aspectos más importantes de la labor de un programador. Documentar el código nos **sirve** para **explicar su funcionamiento**, punto por punto, de forma que cualquier persona que lea el comentario, **puede entender la finalidad del código**.

La labor de documentación es fundamental para la detección de errores y para su mantenimiento posterior, qué 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 todos los programas sufren modificaciones a lo 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 no es repetir lo que hace el código, sino explicar por qué se hace.

La documentación explicará cuál es la finalidad de una clase, de un paquete, qué hace un método, para qué sirve una variable, qué se espera del uso de una variable, qué algoritmo se usa, por qué hemos implementado de una manera y no de otra, qué se podría mejorar en el futuro, etc.

El siguiente enlace nos muestra el estilo de programación a seguir en Java, así como la forma de documentar y realizar comentarios de un código.

Mas información en:
https://www.oracle.com/technetwork/java/javase/documentation/index-137868.html

2.1 Uso y comentarios en Java

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 el compilador va a ignorar, sirve para indicar a los desarrolladores de código diferentes aspectos del código 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. 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* */

Otro tipo de comentarios que se utilizan en Java, son los que se utilizan para explicar qué hace un código, se denominan comentarios JavaDoc y se escriben empezando por / y terminando con */ , estos comentarios pueden ocupar varias líneas. Este tipo de comentarios tienen que seguir una estructura prefijada.**

```
/* Método para ingresar cantidades en la cuenta. Modifica el saldo.
 * Este método va a ser probado con Junit
 */
/**      Comentarios estilo Javadoc
 *
 * @param cantidad
 * @throws Exception
 */
public void ingresar(double cantidad) throws Exception
{
    // el parámetro cantidad debe ser positivo. Comentario de una línea
    if (cantidad<0)
        throw new Exception("No se puede ingresar una cantidad negativa");
    saldo = saldo + cantidad;
    /*  Este método realiza el ingreso de una cantidad de dinero
        la cuenta
     */          Comentario multilínea
}
```

Los comentarios son obligatorios con JavaDoc, y se deben incorporar:

- **Al principio de cada clase.**
- **Al principio de cada método.**
- **Al principio de cada variable de clase**.

No es obligatorio, pero en muchas situaciones es conveniente, poner los comentarios al principio de un fragmento de código que no resulta lo suficientemente claro, en bucles, o si hay alguna línea de código que no resulta evidente y pueda llevarnos a confusión.

Hay que tener en cuenta, que, si el código es modificado, también se deberán modificar los comentarios.

2.2 Alternativas

En la actualidad, el desarrollo rápido de aplicaciones, en muchos casos, va en detrimento de una buena documentación del código. Si el código no está documentado, puede resultar bastante difícil de entender, y por tanto de solucionar errores y de mantenerlo.

La primera alternativa que surge para documentar código, son los comentarios. Con los comentarios, documentamos la funcionalidad de una línea de código, de un método o el comportamiento de una determinada clase.

Existen diferentes herramientas que permiten automatizar, completar y enriquecer nuestra documentación. Podemos citar JavaDoc, SchemeSpy y Doxygen, que producen una documentación actualizada, precisa y utilizable en línea, incluyendo, además, con SchemeSpy y Doxygen, modelos de bases de datos gráficos y diagramas.

Insertando comentario en el código más difícil de entender, y utilizando la documentación generada por alguna de las herramientas citadas anteriormente, se genera la suficiente información para ayudar a cualquier nuevo programador o programadora.

2.3 Documentación de Clases

Las clases que se implementan en una aplicación deben de incluir comentarios.

Al utilizar un entorno de programación para la implementación de la clase, debemos seguir una serie de pautas, muchas de las cuales las realiza el IDE de forma trasparente, en el diseño y documentación del código. Cuando se implementa una clase, se deben incluir comentarios. En el lenguaje Java, los criterios de documentación de clases, son los establecidos por JavaDoc.

Los comentarios de una clase deben comenzar con /** y terminar con */ . Entre la información que debe incluir un comentario de clase debe incluirse, al menos las etiquetas @author y @version, donde @author identifica el nombre del autor o autora de la clase y @version, la identificación de la versión y fecha.

Con el uso de los entornos de desarrollo, las etiquetas se añaden de forma automática, estableciendo el @author y la @version de la clase de forma transparente al programador-programadora. También se suele añadir la etiqueta @see, que se utiliza para referenciar a otras clases y métodos.

Dentro de la clase, también se documentan los constructores y los métodos. 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.

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

2.4 Herramientas

Los entornos de programación que implementa Java, como Eclipse o Netbeans, incluyen una herramienta que va a generar páginas HTML de documentación a partir de los comentarios incluidos en el código fuente. La herramienta ya se ha indicado en los puntos anteriores, y es JavaDoc. Para que JavaDoc pueda generar las páginas HTML es necesario seguir una serie de normas de documentación en el código fuente, estas son:

- Los comentarios JavaDoc deben empezar por /** y terminar por */.
- Los comentarios pueden ser a nivel de clase, a nivel de variable y a nivel de método.
- La documentación se genera para métodos public y protected.
- Se puede usar tag para documentar diferentes aspectos determinados del código, como parámetros.

Los tags más habituales son los siguientes:

Tags más habituales.		
Tipo de tag	**Formato**	**Descripción**
Todos.	@see.	Permite crear una referencia a la documentación de otra clase o método.
Clases.	@version.	Comentario con datos indicativos del número de versión.
Clases.	@author.	Nombre del autor.
Clases.	@since.	Fecha desde la que está presente la clase.
Métodos.	@param.	Parámetros que recibe el método.
Métodos.	@return.	Significado del dato devuelto por el método
Métodos.	@throws.	Comentario sobre las excepciones que lanza.
Métodos.	@deprecated.	Indicación de que el método es obsoleto.