Refactorización de código

¿Cuál es la definición de refactorización de código?

Podríamos decir que la refactorización de código es como darle un “retoque” al código que tenemos para hacerlo mas bonito, ordenado y fácil de entender. Se trata de reorganizar tu código y quitar las partes que ya no son necesarias. Después de la refactorización todos los apartados del código deberán quedar ya en su lugar correcto.

Básicamente diremos que se trata de mejorar la estructura interna del código sin cambiar su funcionalidad externa.

¿Cuál es la motivación principal detrás de la refactorización de código?

La motivación principal seria la de hacer el código más fácil de entender y mantener en el tiempo. Poniendo un ejemplo cotidiano seria como limpiar tu habitación, todo es más fácil de encontrar y de mantener en su sitio.

En el código, la refactorización nos ayuda a reducir la complejidad del mismo, eliminando duplicados y mejorando su estructura en general. Esto facilita a ti y otros desarrolladores a trabajar en un futuro sobre este código de forma mas sencilla ya que tienen “el camino despejado”.

Describe dos tipos de refactorización de métodos mencionados en el texto (pdf) y proporciona un ejemplo para cada uno de ellos.

Extracción de método (Extract Method)

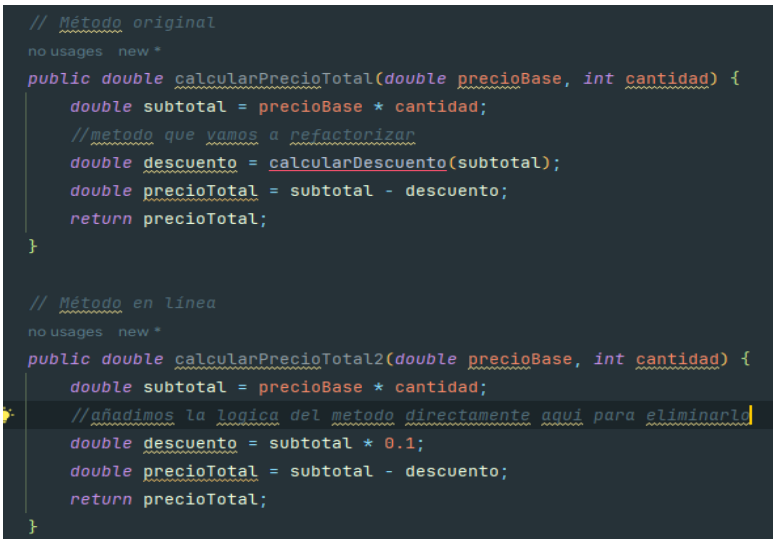
- Se trata de tomar un fragmento de código dentro de un método y convertirlo en un nuevo método separado, esto nos ayuda a mejorar la legibilidad y la reutilización del código.
- Ejemplo de extracción de método:

```java
// Método original
no usages  new *
public double calcularPrecioTotal(double precioBase, int cantidad) {
    double subtotal = precioBase * cantidad;
    double descuento = calcularDescuento(subtotal); // Extracción de método
    double precioTotal = subtotal - descuento;
    return precioTotal;
}

// Método extraído
1 usage  new *
public double calcularDescuento(double subtotal) {
    // Cálculo del descuento...
    return descuento;
}
```

Médodo en línea (Inline Method)

- Es un proceso inverso a la extracción de método. Consiste en eliminar un método pequeño y en línea en el lugar donde lo estamos llamando. Esto hace el código mas claro al evitar el exceso de métodos mas pequeños.
- Ejemplo de método en línea:



```
// Método original
no usages  new *
public double calcularPrecioTotal(double precioBase, int cantidad) {
    double subtotal = precioBase * cantidad;
    //metodo que vamos a refactorizar
    double descuento = calcularDescuento(subtotal);
    double precioTotal = subtotal - descuento;
    return precioTotal;
}

// Método en línea
no usages  new *
public double calcularPrecioTotal2(double precioBase, int cantidad) {
    double subtotal = precioBase * cantidad;
    //añadimos la logica del metodo directamente aqui para eliminarlo
    double descuento = subtotal * 0.1;
    double precioTotal = subtotal - descuento;
    return precioTotal;
}
```

¿Qué principios se sugieren para aplicar la refactorización de métodos de manera efectiva?

- Principio de responsabilidad única (SRP)

  Cada método deber tener solo una función ya hacer una sola cosa.

- Principio abierto/Cerrado (OCP)

  Los métodos deben poder extenderse pero deben estar cerrados a la hora de modificarlos.

- Principio de sustitución (LSP)

  Los métodos deben poder sustituirse por sus subtipos sin alterar la coherencia del programa.

- Principio de segregación de interfaces (ISP)

  Los métodos deben depender de interfaces especificas no de interfaces generales.

- Principio de inversión de dependencias (DIP)

  Los métodos deben depender de abstracciones en lugar de implementaciones concretas para mejorar su flexibilidad y reutilización.

Explica el concepto de "Principio Tell-Don't-Ask" y cómo se aplica en la refactorización de código.

El principio Tell-Don’t-Ask se trata de una guía de diseño que en lugar de preguntar a un objeto del código sobre su estado para tomar luego una decisión, deberíamos decirle directamente que hacer.

Se trata de delegar las operaciones necesarias al mismo objeto y no intentar obtener los datos nosotros mismos.

Cuando hablamos de refactorización se trata de reestructurar el código para que los métodos realicen las acciones directamente en lugar de solicitar los datos y intentar hacer las operaciones fuera del objeto. Los objetos deben ser responsables de su propio comportamiento

¿Cuál es la diferencia entre el método “Extraer una Variable” y el método “Renombrar un Método” en términos de su objetivo y aplicación práctica?

La diferencia entre “Extraer una Variable y “Renombre un Método” se encuentra en sus objetivos y aplicaciones prácticas. Extraer una Variable implica asignar una expresión compleja a una variable con un nombre que la describa dentro de un método, así haremos que el código sea mas claro y legible. En cambio, Renombrar un Método implica cambiar el nombre a un método existente para dejar más clara su función y su comportamiento, esto mejorara la comprensión y la coherencia del código en general.

Podríamos decir que el primero trata de simplificar expresiones complejas y el otro de hacer el código mas entendible con nombres sencillo que identifiquen cada parte.

¿Cuál es el papel del Test-Driven Development (TDD) en el proceso de refactorización y cómo ayuda a garantizar la calidad del código?

El Test-Driven Development (TDD) nos guía el proceso de refactorización al establecer una base de pruebas automatizadas que nos ayudaran a definir el comportamiento que esperamos del código. Cuando refactorizamos, estas pruebas sirven como garantía de que los cambios realizados mantienen su funcionalidad original. Esto asegura la calidad de nuestro código evitando fallos.

Describe al menos tres ejemplos de "code smells" comunes y explica cómo cada uno de ellos puede indicar la necesidad de refactorización.

- Duplicación de código: Se refiere a tener fragmentos de código parecidos o idénticos en diferentes partes del código. Esto aumenta la complejidad y el riesgo de errores en el código ya que deberemos realizar los cambios que queramos en múltiples lugares.

- Métodos largos y complejos: Esto ocurre cuando un método realiza demasiadas tareas o tiene una lógica demasiado compleja dificultando la comprensión y mantenimiento de código.

-Clases con demasiadas responsabilidades: Sucede cuando una clase hace demasiadas cosas, esto hace que no se cumpla el principio de responsabilidad única y dificulta la reutilización y la comprensión de nuestro código.

¿Cómo se define la deuda técnica en el contexto del desarrollo de software y cuáles son las posibles consecuencias de no abordarla de manera adecuada?

La deuda técnica en el desarrollo de software es la acumulación de compromisos técnicos, como código desordenado y falta de pruebas, que afectan a la calidad del software. No abordarla adecuadamente puede conllevar dificultades a la hora de realizar cambios, aumento de costes de mantenimiento, mayor riesgo de errores y deterioro en la calidad del software. Es importante saber gestionar la deuda técnica para garantizar la calidad y sostenibilidad del producto a largo plazo.

Explique el proceso de "Red, Green, Refactor" en Test-Driven Development (TDD) y cómo se integra con la refactorización del código.

El proceso "Red, Green, Refactor" en TDD implica escribir primero pruebas que fallen (Rojo), luego escribir el código mínimo para que las pruebas pasen (Verde) y finalmente refactorizar el código para mejorarlo sin cambiar su comportamiento. La refactorización es esencial en este proceso ya que garantiza que el código permanezca limpio, mantenible y escalable a medida que se desarrollan nuevas funcionalidades.

Hable sobre la importancia de la automatización de pruebas en el contexto de la refactorización, y cómo puede ayudar a reducir la deuda técnica y mejorar la calidad del código.

La automatización de pruebas desempeña un papel esencial en la refactorización del código, ya que proporciona una red de seguridad que detecta rápidamente posibles problemas después de los cambios realizados. Al ejecutar pruebas automatizadas después de cada modificación, podemos identificar cualquier regresión o comportamiento inesperado, lo que ayuda a garantizar la integridad funcional del software. Además, las pruebas automatizadas son fundamentales para reducir la deuda técnica al validar continuamente el código refactorizado, asegurando que cumpla con los requisitos y la funcionalidad esperada sin introducir nuevos errores.

Al mejorar la calidad del código y proporcionar una validación continua, la automatización de pruebas promueve un desarrollo más robusto y confiable. Actúa como una barrera de protección al garantizar que las modificaciones en el código no comprometan su funcionalidad. Esto permite a los equipos de desarrollo refactorizar con confianza, sabiendo que las pruebas automatizadas detectarán cualquier problema potencial. En resumen, la automatización de pruebas es esencial en el proceso de refactorización, ya que garantiza la integridad del código y ayuda a mantener un software de alta calidad y fácilmente mantenible.

¿Cuál es la contribución más destacada de Martin Fowler al campo de la refactorización de código y cómo ha influenciado su trabajo en la industria del desarrollo de software?

La contribución más destacada de Martin Fowler al campo de la refactorización de código es su libro "Refactoring: Improving the Design of Existing Code", publicado en 1999. Este libro ha establecido un estándar en la industria del desarrollo de software al proporcionar técnicas detalladas para mejorar el diseño y la mantenibilidad del código existente sin cambiar su comportamiento externo. La obra de Fowler ha influenciado ampliamente la práctica de la refactorización, convirtiéndola en una herramienta fundamental para mejorar la calidad del software en la industria del desarrollo de software.