Refactorización

Ángel González M.

¿Qué es Refactorización?

Técnica disciplinada para efectuar cambios en la estructura interna de un código sin cambiar su comportamiento externo.

Limpiar requiere cambiar por dentro sin cambiar por fuera

Es fácil tener una idea complicada.
Es muy, muy complicado tener una idea simple.
-- Carver Mead

La refactorización, o en inglés refactoring, es:

- Una **limpieza** de código, básicamente
- La refactorización **no arregla errores, ni incorpora funcionalidades**
- Altera la estructura interna del código sin cambiar su comportamiento externo
- Si durante una refactorización se ha cambiado el comportamiento del software o web, es que has generado un error o bug

Importante

Antes de refactorizar es muy importante hacer los test

Sus objetivos pueden ser:

- Mejorar la **facilidad** de **comprensión** del código
- Cambiar su estructura y diseño
- Eliminar **código muerto**
- Facilitar el **mantenimiento** en el futuro
- Bajar **costes**

El código limpio

Ángel González M.

principios 5S

Seiri u organización: Es fundamental saber dónde están las cosas, mediante enfoques como el uso de nombres correctos.

• Seiton o sistematización: Existe un antiguo dicho norteamericano: un sitio para cada cosa y cada cosa en su sitio. **Un fragmento de código debe estar donde esperamos encontrarlo**; en caso contrario, refactorice hasta conseguirlo.

• Seiso o limpieza: Mantenga limpio el lugar de trabajo. ¿Qué dicen los autores sobre inundar el código de **comentarios** y líneas que **capturan historias o deseos futuros**? **Elimínelos**.
• Seiketsu o estandarización: El **grupo decide** cómo mantener limpio el lugar de trabajo.
• Shutsuke o disciplina: Significa **ser disciplinado** en la aplicación de las prácticas y reflejarlas en el trabajo y aceptar los cambios.

Crear código limpio es complicado

Aprender a crear código limpio es complicado:

- Requiere algo más que conocer principios y patrones.
- Debe practicarlo y fallar.
- Debe ver cómo otros practican y fallan.
- Debe observar cómo se caen y recuperan el paso.
- Debe ver cómo agonizan en cada decisión y el precio que pagan por tomar decisiones equivocadas.

¿Por Qué escribimos código sucio?

- ¿Tenía prisa?
- ¿Plazos de entrega? Seguramente. Puede que pensara que no tenía tiempo para hacer un buen trabajo; que su jefe se enfadaría si necesitaba tiempo para limpiar su código.
- O puede que estuviera cansado de trabajar en ese programa y quisiera acabar cuanto antes.
- O que viera el trabajo pendiente y tuviera que acabar con un módulo para pasar al siguiente. A todos nos ha pasado.
- Todos hemos visto el lío en el que estábamos y hemos optado por dejarlo para otro día.
- Todos hemos sentido el alivio de ver cómo un programa incorrecto funcionaba y hemos decidido que un mal programa que funciona es mejor que nada.
- Todos hemos dicho que lo solucionaríamos después.

Evidentemente, por aquel entonces, no conocíamos la ley de LeBlanc: **Después es igual a nunca.**

La regla del Boy Scout

Los Boy Scouts norteamericanos tienen una sencilla regla que podemos aplicar a nuestra profesión:

Dejar el campamento más limpio de lo que se ha encontrado

No basta con escribir código correctamente. El código debe limpiarse con el tiempo.

Todos hemos visto que el código se corrompe con el tiempo, de modo que debemos adoptar un papel activo para evitarlo.

Principales "malos olores del código"

Ángel González M.

Código duplicado

Sin duda el horror de los horrores. Encontrarse una misma funcionalidad en varios sitios. No hay nada más propenso a dar problema que código duplicado.

Solución: sacar código duplicado a un método/clase nueva.

Ejemplo

```
public static void main(String[] args) {
        Scanner lector = new Scanner(System.in);

        System.out.println("Inserta la base");
        int base = lector.nextInt();

        System.out.println("Inserta la altura");
        int altura = lector.nextInt();

        float r1 = base * altura / 2.0f;
        System.out.println("El area es "+ r1);



        System.out.println("Inserta la base");
        base = lector.nextInt();

        System.out.println("Inserta la altura");
        altura = lector.nextInt();

        float r2 = base * altura / 2.0f;
        System.out.println("El area es "+ r1);
        lector.close();
    }
```

```
static Scanner lector = new Scanner(System.in);

public static int leerValor(String mensaje) {
        System.out.println("Inserta la base");
        int valor = lector.nextInt();
        return valor;
}

public static float calcularArea(int base, int altura) {
        return base * altura / 2.0f;
}

public static void main(String[] args) {
        int base = leerValor("Inserta la base");
        int altura = leerValor("Inserta la altura");
        float r1 = calcularArea(base, altura);
        System.out.println("El area es "+ r1);

        base = leerValor("Inserta la base");
        altura = leerValor("Inserta la altura");
        float r2 = calcularArea(base, altura);
        System.out.println("El area es "+ r2);

        lector.close();
}
```

Ángel González M.

Código muerto

¿Cuántas veces nos hemos encontrado con código que no se usa? Los desarrolladores a veces pecamos de precavidos y vamos dejando clases y métodos “por si acaso esta funcionalidad vuelve a ser requerida” o “por si se usa”.

Solución: simplemente, borrar el código. Si hace falta más adelante, para eso está el sistema de control de versiones, no? 🙂

Ejemplo

```
int calcular (int x, int y) {
        int z = x/y;
        return x*y;
}
```

```
int calcular (int x, int y) {
        return x*y;
}
```

Métodos largos

Hay veces que nos encontramos con métodos o funciones cuyo fin no vemos a simple vista. ¿Qué hace este código? Si es tan largo, seguro que hace más de una cosa. Es difícil de leer y de mantener.

Solución: Buscar las diferentes responsabilidades dentro de dicho código, extraerlas en clases y métodos. Es preferible tener muchos métodos pequeños que un método enorme.

Ejemplo

```
public static void operar() {
        // Rellenar array
        int datos[] = {2, 9 ,3,4,1};
        // ordenar array
         int aux;
      for (int i = 0; i < datos.length - 1; i++) {
        for (int j = 0; j < datos.length - i - 1; j++) {
          if (datos[j + 1] < datos[j]) {
            aux = datos[j + 1];
            datos[j + 1] = datos[j];
            datos[j] = aux;
          }
        }
      }
     // pintar en pantalla el array
      for (int i = 0; i < datos.length; i++) {
        System.out.println(datos[i]);
      }
  }
  public static void main(String[] args) {
        operar();
  }
```

```
public static int[] rellenarArray() {
        return new int[] { 2, 9, 3, 4, 1 };
  }
  public static int[] ordenarArray(int[] datos) {
        int aux;
        for (int i = 0; i < datos.length - 1; i++) {
          for (int j = 0; j < datos.length - i - 1; j++) {
            if (datos[j + 1] < datos[j]) {
              aux = datos[j + 1];
              datos[j + 1] = datos[j];
              datos[j] = aux;
            }
          }
        }
        return datos;
  }
  public static void pintarArray(int[] datos) {
        for (int i = 0; i < datos.length; i++) {
          System.out.println(datos[i]);
        }
  }
  public static void operar() {
        int[] array = null;
        array = rellenarArray();
        array =  ordenarArray(array);
     pintarArray(array);
  }
  public static void main(String[] args) {
        operar();
  }
```

Ángel González M.

Clases largas

Estamos en las mismas que el caso anterior. Clases demasiado largas, seguro que tienen responsabilidades que no les corresponden. Igualmente difíciles de mantener y de interpretar.

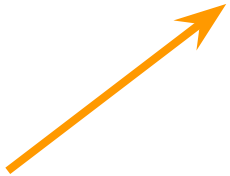
Solución: Mover los métodos según sus responsabilidades a otras clases, crear clases nuevas…

Ejemplo

```
class Coche {
        int velocidad;
        int numPuertas;

        void arrancar() {
                velocidad = 5;
        }
}
class Moto {
        int velocidad;

        void arrancar() {
                velocidad = 5;
        }
}
```



```
class Vehiculo {
        int velocidad;

        void arrancar() {
                velocidad = 5;
        }
}
class Coche extends Vehiculo{
        int numPuertas;

}
class Moto extends Vehiculo {

}
```

Ángel González M.

Lista larga de parámetros de una función

Según el libro “Clean Code”, si una función tiene más de 2 parámetros… algo estamos haciendo mal. Una larga lista de parámetros dificulta su uso y comprensión. Si los parámetros son del mismo tipo puede llegar a confundir. ¿El String este primero que era, el usuario o el password?…¿O era el apellido?.

Solución: En vez de un “addCustomer(String name, String surname, String phoneNumber, String….)” hacemos un “addCustomer(Customer newCustomer)” todo queda mucho más claro, ¿no?. La encapsulación puede ser útil en este tipo de olores.

Nota, en ocasiones si es necesario usar más de 2 parámetros

Ejemplo

```
public static void matricular(String nombre,
String apellidos, Date fechaAlta, String
asignatura) {
        // ...
    }
```

```
class Alumno{
    String nombre;
    String apellidos;
    Date fechaAlta;
    String asignatura;
}
```

….

```
public static void matricular(Alumno alumno) {
        // ...
}
```

Sentencias Switch

Los switch son malvados no los uses

Solución: Usar polimorfismos

Tampoco compruebas el tipo de objeto que estás tratando usando if's anidados, usa polimorfismo.

Ejemplo

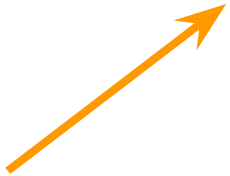
```
class Triangulo{
        int base;
        int altura;

        public Triangulo(int base, int altura) {
                super();
                this.base = base;
                this.altura = altura;
        }

        float getArea() {
                return base * altura / 2.0f;
        }
}
class Rectangulo{
        int lado1;
        int lado2;

        public Rectangulo(int lado1, int lado2) {
                super();
                this.lado1 = lado1;
                this.lado2 = lado2;
        }

        float getArea() {
                return lado1 * lado2;
        }
```



```
abstract class Figura{
        abstract float getArea();
}

class Triangulo extends Figura{
        int base;
        int altura;

        public Triangulo(int base, int altura) {
                super();
                this.base = base;
                this.altura = altura;
        }

        float getArea() {
                return base * altura / 2.0f;
        }
}

class Cuadrado extends Figura{
        int lado;

        public Cuadrado(int lado) {
                super();
                this.lado = lado;
        }

        float getArea() {
                return lado * lado;
        }
```

Ángel González M.

Comentarios

Demasiados comentarios en código pueden ocultar un olor. Si tienes que explicar demasiado un método o una clase, es que ese código no es claro. Es preferible que el código sea autoexplicativo a que pongas comentarios por cada método que crees.

Es más, ¿cuantas veces se os han quedado esos comentarios obsoletos? Seguro que más de una.

Solución: Crear código limpio y que se explique solo. Escribir comentarios en lugares donde no estés muy seguro del código o donde quieras recordar algo para futuros cambios.

Ejemplo

```
public float calculo(float a) {
        return 4/3 * 3.141 * Math.pow(r, 3);
}
```

```
public double volumenEsfera(float radio) {
        double radioAlCubo = Math.pow(radio, 3);
        final double PI =  3.1415f;

        return 4/3 * PI * radioAlCubo;
}
```

Resumen de todos los "malos olores" del código

Ángel González M.

Los "malos olores" del código

- **Código duplicado** (Duplicated Code): Si encontramos la misma estructura de código en varios sitios, lo mejor es unificarla en un único punto.

- **Método largo** (Long Method): Los métodos pequeños son más claros en lo que hacen y permiten compartir el código y que se pueda elegir el método a llamar según el caso. La sobrecarga en la llamada a un método es casi despreciable, por lo que no debe ser excusa para usar métodos lo más pequeños posible.

- **Clase grande** (Large Class): Una clase que hace demasiadas cosas suele tener muchas variables de instancia y, tras ellas, suele haber código duplicado. Si una clase no utiliza todas sus variables de instancia en todo momento, las que no se usen se deberían eliminar o extraer a otra clase.

- **Lista de parámetros larga** (Long Parameter List): De forma distinta a lo que ocurría con tecnologías anteriores, cuando se usan objetos, no hace falta pasar a un método toda la información que necesita para su ejecución, sino sólo aquella que es imprescindible para que puede obtener todo lo que necesita.

Los "malos olores" del código

- **Cambio divergente** (Divergent Change): Cuando hay que hacer un cambio, debemos ser capaces de identificar un único punto en el programa donde éste deba hacerse. Si tenemos una clase donde debamos cambiar 3 métodos por una razón y otros 4 por otra causa distinta, tal vez este objeto debería ser dividido en 2 con distintas responsabilidades.

- **Cura de un escopetazo** (Shotgun Surgery): Este caso es el contrario del anterior. Hay y que hacer un cambio y para ello deben modificarse varias clases desperdigadas por el código. Los métodos afectados deberían agruparse en una sola clase.

- **Envidia de capacidades** (Feature Envy): Si un método de una clase hace referencia más a métodos y parámetros de otra clase, tal vez sea en esa otra clase donde debería estar.

- **Agrupaciones de datos** (Data Clumps): Si un grupo de datos aparecen constantemente juntos y se usan siempre en los mismos momentos, estos datos deberían agruparse dentro de un objeto.

- **Comentarios** (Comments): A veces los comentarios lo que están enmascarando es uno de los "malos olores" que se han visto anteriormente. Es mejor invertir el tiempo en mejorar este código que en comentarlo.

Los "malos olores" del código

- **Obsesión por los tipos primitivos** (Primitive Obsession): Existen ciertos datos que están mejor agrupados en pequeñas clases, en lugar de usar primitivas, aunque esto último parezca más sencillo. Usar clases para estos datos ofrece nuevas capacidades como añadir nueva funcionalidad o realizar comprobaciones de tipo de datos.

- **Sentencias switch** (Switch Statements): A veces se encuentran las mismas sentencias switch repartidas por el código. Si se incluye un nuevo apartado clause se debe hacer para todas ellas y es fácil saltarse alguna. Normalmente esto es manejado mucho mejor a través de polimorfismo, cobre todo cuando la sentencia switch actúa en función de un valor que define el tipo de un objeto.

- **Jerarquías paralelas de herencia** (Parallel Inheritance Hierarchies): Esto ocurre cuando al añadir una clase en una jerarquía se hace evidente la necesidad de añadir una nueva clase en otra jerarquía distinta. Si las instancias de una jerarquía hacen referencia a las de la otra se puede eliminar una de ellas llevando sus métodos a la otra.

- **"Clase vaga"** (Lazy Class): Una clase que no hace nada está añadiendo una complejidad innecesaria y es mejor hacerla desaparecer.

Los "malos olores" del código

- **Generalización especulativa** (Speculative Generality): Cuando se añade funcionalidad especulando sobre lo que necesitaremos en el futuro, pero que en este momento es innecesario, el resultado es código más difícil de entender y mantener. Es mejor deshacerse de ello.

- **Campo temporal** (Temporary Field): Si existen variables de instancia que parecen usarse sólo en algunos casos, puede ser confuso entender en cuáles. Es mejor sacar estas variables a otra clase.

- **Cadenas de mensajes** (Message Chains): Estas cadenas aparecen cuando un cliente pide a otro objeto un objeto y a su vez llama a éste último para obtener otro objeto. De esta menra, el cliente se acopla a toda esta estructura de navegación y cualquier cambio en ella le afectará. Es mejor recuperar otros objetos mediante un delegado que encapsule dicha navegación y abstraiga al cliente de ella.

- "**Hombre en el medio**" (Middle Man): El caso anterior no es necesario que se cumpla a rajatabla. A veces, si existen muchos métodos que delegan a una misma clase, es mejor quitarnos el delegado y referirnos a ella directamente.

Los "malos olores" del código

- **Intimidad inapropiada** (Inappropiate Intimacy): Cuando una clase se dedica a hurgar en partes de otra clase que parecen ser privadas, ha llegado el momento de desacoplar dichas clases, por ejemplo, a través de una tercera que contenga la funcionalidad común de las dos anteriores.

- **Clases alternativas con interfaces diferentes** (Alternative Classes with Different Interfaces): Si tenemos dos métodos que hace los mismo pero se llaman de forma distinta, es mejor unificarlos para que se use siempre un único método.

- **Clase de librería incompleta** (Incomplete Library Class): A veces una clase de una librería de tercero no hace todo lo que necesitamos y debemos adaptar su uso a nuestras necesidades ya que no podemos modificarla.

- **Clase de datos** (Data Class): Las clases que solo tienen datos getters y setters son manipuladas en gran medida por otras clases. Podemos buscar este comportamiento para llevarlo a las clases que contienen los datos.

- **Legado rechazado** (Refused Bequest): A veces una subclase no usa todos los métodos que hereda de su clase padre. Esto puede significar que la jerarquía no es correcta y los métodos de las clases se deben mover a donde se usan realmente.

Cuando refactorizar

En cuanto a los mejores momentos para refactorizar, esto son:

- Cuando nos topemos por tercera vez con el mismo problema y debemos eliminar las duplicidades.
- Cuando estamos añadiendo una nueva funcionalidad y necesitamos entender lo que hay hecho, refactorizar ayuda a entender dicho código.
- Cuando estemos arreglando un error, la refactorización ayuda a clarificar el código y a encontrar la fuente del problema.
- Mientras se hace una revisión de código, puesto que en este momento es donde se hace evidente si el código que ha escrito un desarrollador es claro para los demás.

Cuando no refactorizar

También hay que tener en cuenta que hay momentos en los que la refactorización no ofrece más inconvenientes que ventajas:

- Cuando en realidad lo necesario es reescribir el código porque éste no funciona como debiera.
- Cuando estamos cerca de una fecha de entrega. La verdadera ganancia de la refactorización se hará evidente tras la entrega, lo que ya es demasiado tarde.

Ejercicios

Haz y repasa los ejercicios propuestos

https://docs.google.com/document/d/1XGiybOX1y97p-_w4UYFFXVS620s_ZSgZGWNV80ua1fc/edit?usp=sharing

MÁS EJERCICIOS

http://www.iwt2.org/alfresco/d/a/workspace/SpacesStore/50b15165-c473-4146-b77e-605be196011d/Tema%202.%20Ejercicios%20refactorizaci%C3%B3n%20resueltos.pdf?guest=true