Refactorización

Created	@March 3, 2022 11:55 AM
Tags	2doTrimestre
Formato	
Materia	
Tema	

La refactorización consiste en la limpieza del código. No arregla errores ni incorpora funcionalidades. Altera la estructura interna del código sin cambiar su comportamiento externo.

Importante: Hacer los test antes de la refactorización.

Objetivos:

- Facilitar la comprensión
- Eliminar código muerto
- Facilitar el mantenimiento en el futuro
- Bajar costes

Principios 5S

- Organización (Seiri): Uso de los nombres correctos, entre otras cosas.
- Sistematización (Seiton): Un fragmento de código debe estar donde esperamos encontrarlo.
- Limpieza (Seiso): Evitar comentarios y elementos innecesarios.
- Estandarización (Seiketsu): El grupo debe mantener el lugar de trabajo siguiendo un mismo método.
- Disciplina (Shutsuke): Ser disciplinado en la aplicación de las prácticas.

Ley de LeBlanc: Después es igual a nunca.

La regla del Boy Scout: Dejar el campamento más limpio de lo que se ha encontrado.

Indicativos de un mal código

▼ Código duplicado (Misma funcionalidad en varios sitios)

```
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();
}
```

▼ Solución: Sacar código duplicado a un método/clase nueva.

```
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();
  }
```

▼ Código muerto (Código que no se usa)

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

▼ Solución: Borrar el código.

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

▼ Métodos largos

```
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();
  }
```

▼ Solución: Buscar y extraer las diferentes responsabilidades.

```
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();
  }
```

▼ Clases largas

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

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

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

▼ Solución: Mover los métodos según sus responsabilidades a otras clases.

```
class Vehiculo {
  int velocidad;

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

}
class Moto extends Vehiculo {

}
```

▼ Lista larga de parámetros de una función

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

▼ Solución (En ocasiones sí es necesario usar más de dos parametros)

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

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

▼ Sentencias Switch

```
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;
  }
}
class Cuadrado{
  int lado;
```

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

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

public class Principal {

  public static void main(String[] args) {
    float resultado;
    Triangulo t = new Triangulo(4,5);
    resultado = t.getArea();
    System.out.println(resultado);

    Cuadrado c = new Cuadrado(4);
    resultado = c.getArea();
    System.out.println(resultado);

  }

}
```

▼ Solución: Hacer uso del polimorfismo.

```
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;
```

```
  }
}

class Rectangulo extends Cuadrado{

  int lado2;

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

  float getArea() {
    return super.lado * lado2;
  }
}

public class Principal {

  public static void main(String[] args) {
    float resultado;
    Figura f = new Triangulo(4,5);
    resultado = f.getArea();
    System.out.println(resultado);

    f = new Rectangulo(4,5);
    resultado = f.getArea();
    System.out.println(resultado);

  }
```

▼ Comentarios

(Imaginemos un código que se vale de comentarios para explicar qué hace)

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

▼ Solución: Crear código limpio y que se explique solo.

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

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

▼ Resumen Mal 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.
- **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.
- **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.
- **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 manera, 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.
- **Intimidad inapropiada** (Inappropriate 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

- 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

- 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.