REFACTORIZACIÓN

CODE UD6 – Tema 1

IES Plurilingüe Antón Losada Diéguez

Adrián Fernández González

Tabla de contenido

Refactorización

1. Introducción

La refactorización es el proceso en el que se reestructura el código para hacerlo más legible, reutilizable y manejable pero **sin cambiar la funcionalidad** del mismo.

Aunque pueda parecer algo innecesario, ayuda a entender mejor las partes de un sistema, agilizando el desarrollo, la detección de errores, la modificación y el mantenimiento del mismo, entre otras muchas cosas, por lo que siempre es recomendable hacer el código lo más legible posible.

En desarrollos destinados a ser públicos, esto se vuelve algo vital, pues es necesario que terceras personas entiendan el código.

La refactorización a veces colisiona con la optimización, ya que puede ser necesario ampliar el número de métodos y llamadas para que sea más legible.

Por tanto, dependiendo del objetivo del código, puede ser necesario sacrificar legibilidad para optimizar o gastar algunos recursos adicionales para hacerlo más legible.

Hay que tener en cuenta que el término legible es muy subjetivo, por lo que dos personas pueden no coincidir en el proceso de refactorización

1.1. Código espagueti

Se denomina código espagueti al código que tiene una estructura difícil de seguir, que su flujo se asemeja más a un plato de espaguetis que a una estructura secuencial.

Cuando el código llega a este punto, es de vital importancia refactorizarlo para hacerlo legible.

2. Cuándo refactorizar

Lo habitual en el proceso de desarrollo es realizar la refactorización tras haber escrito el código en cuestión, para hacerlo lo más legible posible.

Este proceso puede realizarse múltiples veces a lo largo del ciclo de vida del sistema, ya que el añadir nuevas funcionalidades, puede hacer que sea necesario aunar código, modificarlo, etc., por lo que será necesario volver a refactorizar.

Aunque todo depende del sistema, las necesidades en el desarrollo y hasta la política de la empresa o del cliente, existen ciertos momentos en los que es recomendable realizar un proceso de refactorización:

- Cuando aparece un mismo problema más de dos veces es recomendable aunar, eliminando duplicidades.
- Cuando se añade una nueva funcionalidad, se refactoriza para que sea más legible.

- Cuando se va a arreglar un error, se refactoriza para hacer más legible el código y así encontrar y entender el fallo.
- Mientras se revisa el código, ya que es cuando se evidencia o no la legibilidad del mismo.

Hay que tener en cuenta que no se debe refactorizar si el código no funciona o si la fecha de entrega está próxima, ya que, en ambos casos, se pierde tiempo.

3. Malos olores del código

Se denominan malos olores del código a ciertas líneas, bloques de código o estructuras que reducen la legibilidad, el manejo del código y/o su reutilización.

A continuación, se exponen los más comunes divididos por tipos, aunque existen muchos más que dependerán de un ámbito en concreto, técnicas utilizadas, lenguajes y tecnologías aplicadas.

3.1. Clases monstruosas

Las clases monstruosas son un tipo de mal olor de código en el que las clases se vuelven excesivamente grandes y complejas.

Este tipo de mal olor de código suele aparecer cuando el sistema va aumentando, se va desarrollando y no se ha realizado ninguna refactorización durante el proceso.

3.1.1. Método largo

Un método excesivamente largo puede ser difícil de seguir, por lo que es más legible separarlo en partes con una lógica independiente.

La sobrecarga a los métodos no tiene un excesivo impacto a no ser que sea una parte crítica del sistema, por lo que suele prevalecer la legibilidad.

Hay que tener en cuenta que un uso excesivo de la separación, puede conllevar lo opuesto, que se haga difícil de seguir.

```
switch (op) {
    case "0":
        System.out.println("~~~~~~~~~~~~~~~~~~~~
        con.status();
        break;
    case "1":
        newOrder(day);
        System.out.println("Pedido generado");
        break;
    case "2":
        Materiales pedidos = getOrders();
        if(pedidos != null)
        {
            con.construir(pedidos);
        }
        genReport(day);
        day++;
        break;
    default:
        run = false;
        System.out.println("~~~~~~~~~~~~~~~~~~~~
}
```

En este ejemplo se puede ver un segmento de un menú en el que, en vez de hacer toda la lógica en su interior, se separó en métodos que realizan las funcionalidades de cada segmento.

3.1.2. Clase grande

Cada clase ha de representar una entidad única y utilizar, en la medida de lo posible todos sus métodos y atributos, sobre todo los privados y protegidos, de una u otra forma.

Si una clase pierde su significado propio o representa diversas entidades, es mejor separarla en clases que sí cumplan esto.

3.1.3. Lista de parámetros larga

Todo método y función ha de tener los parámetros mínimos imprescindibles para su funcionamiento.

Si va a necesitar prácticamente todos los datos que tiene un objeto como atributos, es preferible pasar por parámetro el objeto que no todos los valores.

```
public void escribirEmpleado(String nombre, int edad, String dni, int sueldo)

public void escribirEmpleado(Empleado e)
```

Si los datos los puede obtener por su cuenta, de constantes de otras clases, atributos propios, etc., estos valores deberían obviarse.

Si dispone de valores por defecto, es recomendable crear métodos sobrecargados.

3.1.4 Obsesión por los tipos primitivos

Aunque lo más eficiente es el uso de datos primitivos, ciertos datos están mejor agrupados en una clase.

Cuando un conjunto de datos debe realizar una operación de forma conjunta o sean susceptibles de ampliación de funcionalidades en un futuro, es mejor que se agrupen bajo una clase a que se mantengan como tipos primitivos.

```
public int buscarDato(int[] datos, int buscar);
public void cambiarDato(int[] datos, int nuevo);
```

En este ejemplo, se puede ver como se realizan varias operaciones sobre el *array* datos desde fuera, por lo que sería recomendable crear una clase que guarde esos datos y realice las operaciones oportunas.

3.1.5. Aglomeraciones de datos (Data Clumps)

Si un grupo de datos se pasan constantemente juntos, es mejor crear una clase que los agrupe y pasar un objeto que todos los datos.

```
public double calcularDistancia(double xA, double yA, double zA,
                                double xB, double xB, double zB);

public double dibujarPunto(double xA, double yA, double zA);

public double calcularArea(double xA, double yA, double zA,
                           double xB, double xB, double zB,
                           double xC, double xC, double zC);
```

En este ejemplo, las coordenadas de cada punto siempre se pasan juntos, x, y, z, por lo que es recomendable aunarlas en una clase propia que, además, se puede utilizar para todos los conjuntos de puntos.

```
public double calcularDistancia(Coords A, Coords B);
public double dibujarPunto(Coords A);
public double calcularArea(Coords A, Coords B, Coords C);
```

En este ejemplo, se observa una mejora sustancial en la legibilidad y un mejor manejo de los datos, aunque el consumo de recursos es mayor.

3.2. Abusos de la orientación a objetos

Estos malos olores surgen por la aplicación incorrecta de la programación orientada a objetos.

3.2.1. Sentencias switch para funcionalidad

El uso de los switch debe estar orientado a operaciones relativamente sencillas, tomas de decisiones y no para realizar operaciones distintas en función de un tipo o estado de un objeto.

Esto ocasionaría que, fuese necesario incluir un switch cada vez que el objeto en cuestión fuese a realizar una operación que dependa de su estado.

A mayores, si algo cambia, como un estado adicional, sería necesario crear un nuevo case en cada switch del sistema.

```
public void actuar()
{
    switch(this.tipo)
    {
        case 'entrada':
            ...
        break;
        case 'procesando':
            ...
        break;
        ...
    }
}
```

En estos casos, es más adecuado usar clases distintas que heredan de un padre común o una interfaz implementada por todos.

3.2.2. Clases alternativas con diferentes interfaces

Si existen dos clases distintas que hacen lo mismo, pero con distintos nombres, es necesario unificar todo en una.

Esto también puede ser un indicativo de una mala documentación o comunicación, pues suele ocurrir cuando dos desarrolladores implementan una clase de forma independiente sin saber que ya existe en el sistema una que realiza lo mismo.

3.2.3. Legado rechazado (Refused Bequest)

Si una subclase no usa o se aprovecha de los métodos y atributos del padre, puede indicar que la clase no pertenece realmente a esa jerarquía y ha de ser independiente.

3.2.4. Campo temporal

Los atributos de una clase han de ser utilizados habitualmente por la clase, si existen varios que se usan únicamente en un método en concreto o bajo ciertas circunstancias, es recomendable separarlo en una clase independiente.

3.3. Prevencionistas de cambios

Estos malos olores del código son aquellos que, si se realiza un cambio en una parte del sistema, implica el cambio en otras partes.

Estos malos olores implican un número excesivo de cambios en el código cada vez, por lo que aumentan los tiempos de desarrollo y mantenimiento.

3.3.1. Cambio divergente

Si al realizar un cambio en el sistema es necesario cambiar múltiples cosas de una clase aparentemente inconexas. Esto es indicativo de un mal diseño y/o abuso de copiar-pegar métodos.

Cada cambio en una funcionalidad debería estar localizado en un único punto.

Lo más probable, es que sea necesario dividir la clase y, en ocasiones, crear un padre común para ambas.

3.3.2. Cirugía de escopeta/Cirugía a tiros (Shotgun Surgery)

Lo contrario al anterior, si al realizar un cambio en una clase concreta, es necesario realizar cambios en otras clases, puede ser un claro indicativo de que es necesario aunar todo en una misma clase.

3.3.3. Jerarquías paralelas de herencia

Si al crear una nueva clase hija en una jerarquía es necesario crear otra en otra jerarquía distinta, es un claro indicativo de que ambas jerarquías han de ser unificadas.

3.4. Dispensables

Un dispensable es algo superfluo, innecesario que no aporta nada.

Estos malos olores surgen cuando se realizan cambios o añadidos que no se llegan a usar, a veces por intentar adelantarse a los acontecimientos, gastando tiempo innecesariamente.

Siempre es recomendable dedicarse a hacer lo que es necesario ahora y no posibles futuros.

3.4.1 Código duplicado

Si un bloque de código se repite en varios sitios, lo mejor es unificarla en un único punto. Esto se puede conseguir con métodos auxiliares, clases helper, librerías, funciones, etc.

```
/**
 * Devuelve un número aleatorio entre el mínimo y el máximo.
 * @param min El mínimo valor. Incluído.
 * @param max El máximo valor. No incluído.
 * @return Un valor aleatorio entre <code>min</code> y <code>max</code>
 */
public static int getRandomNumber(int min, int max)
{
    return (int) ((Math.random() * (max - min)) + min);
}
```

En este ejemplo se puede ver el habitual método de obtención de un número aleatorio en un rango. Puesto que va a ser utilizada en distintas partes del sistema, es recomendable crear un método auxiliar, ya que, al hacerlo de forma genérica, puede ser reutilizada.

3.4.2. Clase vaga

Una clase que no hace nada o es ridículamente sencilla, es mejor eliminarla.

Aunque esto parezca algo obvio, hay casos en los que una clase se crea, pero nunca llega a usarse. Esto puede ocurrir si se preparó el sistema para una futura ampliación que nunca llegó a suceder o dejó de ser útil tras un proceso de refactorización, por ejemplo.

3.4.3. Generalización especulativa (Speculative Generality)

Esto es similar al anterior, cuando se crea un conjunto de clases, métodos y funcionalidades para futuras ampliaciones por si estas ocurren.

Si algo no se usa actualmente, se elimina, si en un futuro se necesita algo, se implementa.

Ojo, esto no es aplicable para sistemas creados con el objetivo de ser genéricos, utilidades para terceros en los que hay que ofrecer un abanico de utilidades aunque no lleguen a ser utilizadas por todos los usuario.

3.4.4. Clase de datos

Las clases de datos son aquellas que solo contienen datos públicos o accesibles mediante getters y setters. Estas clases carecen de funcionalidades añadidas, solo sirven como almacenes de estos datos.

Este tipo de clase es innecesaria, los datos han de estar en la clase que los utiliza.

No hay que confundir una clase de datos con una clase de acceso a datos (Patrón DAO), la primera simplemente almacena datos, la segunda ofrece un punto de entrada

3.4.5. Código muerto

Se denomina código muerto a aquel que ya no se usa y, por tanto, ha de ser eliminado.

El código muerto aparece cuando se ha cambiado parte del sistema, se ha dejado de usar una parte y no se ha verificado si ya no se necesita.

Este código lo único que hace es aumentar el tamaño del sistema y entorpecer su mantenimiento.

Ojo, no confundirlo con código obsoleto (Deprecated) que indica que hay una alternativa mejor a un código que sí era usado.

3.4.6. Comentarios excesivos

Si un bloque de código necesita de los comentarios para ser entendido, quiere decir que no es legible, por lo que es necesario refactorizarlo.

3.4.7. Complejidad artificial

El uso de patrones, algoritmos probados y bloques de código de terceros ha de estar justificado y ser necesario para una funcionalidad.

El abuso en su uso solo crea complejidad adicional y partes que no se llegan a usar.

3.5. Acopladores

Los malos olores de tipo acoplador son aquellos que establecen una unión excesiva entre varias clases.

3.5.1. Envidia de funcionalidad (Feature Envy)

Si un método de una clase hace referencia más a métodos y atributos de otra clase, puede indicar que el método debería pertenecer a la otra clase.

3.5.2. Cadenas de mensajes

Cuando se trabaja con mensajes entre objetos, si un objeto solicita un objeto a uno y este a su vez a otro y así sucesivamente, se crea una alta dependencia entre todas las clases de la cadena. Esto hace que, si una de esas clases se ve modificada, toda la cadena ha de ser modificada.

En estos casos, lo mejor es delegar en una clase que se encargue de realizar las comunicaciones, de tal forma que, si algo cambiase, solo el delegado tendría que ser modificado.

3.5.3 "Hombre en el medio"

Cuando toda funcionalidad de una clase está delegada a otra y únicamente sirve de fachada sin razón aparente, esa clase ha de ser eliminada, puesto que no hace nada en sí misma.

```
public int buscarDato(int[] datos, int buscar)
{
    Datos.buscarDato(datos, buscar);
}
```

En este ejemplo puede verse como el método *buscarDato* realmente llama a un método de otra clase, siendo simplemente un puente.

3.5.4. Clase de librería incompleta

Es habitual el uso de librerías externas, pero no siempre se mantienen, se actualizan a tiempo o disponen de todo lo que se necesita.

Si la librería es de código abierto, se podría modificar, pero si no lo es, es necesario añadir esos métodos en otra clase, ya sea una extensión o un encapsulador de la misma.

Lo que no debería hacerse es reescribir desde cero la librería, ya que implicaría mucho tiempo y que igual el problema es que no se escogió la adecuada o no es necesaria.

3.5.5. Intimidad inapropiada

Si un método de una clase utiliza los atributos y métodos en teoría privados de otras clases, quiere decir que ese método o partes del mismo pertenecen a esas clases.

3.6. Malas prácticas

Estos malos olores vienen dados por malas prácticas a la hora de programar y que reducen igualmente la legibilidad.

3.6.1. Nombres poco significativos

Una de las malas prácticas más habituales es el uso de nombres poco significativos en variables, métodos y clases.

Todo nombre que se use en el código para designar una variable, un método o una clase ha de ser lo suficientemente explicativo como para entender su uso.

3.6.2. Nombres excesivamente significativos

El caso contrario al anterior, el uso de un nombre excesivamente largo para intentar describir mejor algo suele causar todo lo contrario, que se entienda peor, además de que se hace más complicado escribirlo.

3.6.3. Código segmentado

Es habitual separar con un salto de línea ciertos bloques de código dentro de un mismo método para separar las distintas partes del mismo. Aunque útil, no se debe abusar de ello, ya

que, cuando el código está separado de forma excesiva, además de ocupar mucho más espacio, da la impresión de que son partes independientes cuando en realidad no.

3.6.4. Anidamiento de ifs

Además de ser más óptimo, el anidamiento de los if es mucho más legible, ya que se ve el flujo real del sistema, los caminos que tiene y las distintas opciones.

```
if(x == 1)
{
    ...
}
if(x > 5)
{
    ...
}
if(x < -1)
{
    ...
}
```

```
if(x == 1)
{
    ...
}else{
    if(x > 5)
    {
        ...
    }else{
        if(x < -1)
        {
            ...
        }
    }
}
```

3.6.5. Orden de if-else

Por convención, es habitual poner la condición del if la opción más común, la que más veces se repite y en el else la que menos, ya que se supone que será la que más se revise.

3.6.6. Returns excesivos

No siempre hay un único punto de salida de una función, por lo que sería necesario el uso de múltiples *return*, pero esto no es lo más adecuado en cuanto a legibilidad.

Para una mejor lectura, se suele limitar a dos el número de *returns* y, en caso de necesitar más, se utilizará una variable que almacene el resultado y será devuelta en el *return*.

```
private static int compare(int A, int B)
{
    if(A > B)
    {
        return -1;
    }else{
        if(A < B)
        {
            return 1;
        }else{
            return 0;
        }
    }
}
```

En este ejemplo, la función *compare* indica qué elemento es el mayor o si son iguales. El uso de los tres *returns* no es aconsejable, por lo que más adecuado es el uso de una variable.

```
private static int compare(int A, int B)
{
    int val = 0;
    if(A > B)
    {
        val = -1;
    }else{
        if(A < B)
        {
            val = 1;
        }
    }
    return val;
}
```

En este otro ejemplo, se usa una variable *val* que almacena el valor a devolver. Como se puede observar, al ponerle un valor por defecto, no hace falta el último *else*.

3.6.7. Uso de excesivos métodos auxiliares

Los métodos auxiliares pueden mejorar la legibilidad, además de volver el código más reutilizable, pero su uso excesivo puede hacer que sea difícil seguir el código, ya que es necesario ir a saltos.

3.6.8. Estructura en paquetes

La división del sistema en paquetes es fundamental para separar la lógica, las partes del mismo y mejorar su legibilidad.

No hay un estándar de los nombres o divisiones a realizar, eso dependerá del sistema y la política de empresa.