por Mari Chelo Rubio

Entornos de Desarrollo
Bloque 3
Tema 3:Refactorización de código. Patrones más habituales

3.3.1. Introducción

En ingeniería del software, el término refactorización se usa a menudo para describir la modificación del código fuente sin cambiar su comportamiento, lo que se conoce informalmente por limpiar el código. La refactorización se realiza a menudo como parte del proceso de desarrollo del software: los desarrolladores alternan la inserción de nuevas funcionalidades y casos de prueba con la refactorización del código para mejorar su consistencia interna y su claridad. Los tests aseguran que la refactorización no cambia el comportamiento del código. La refactorización es la parte del mantenimiento del código que no arregla errores ni añade funcionalidad. El objetivo, por el contrario, es mejorar la facilidad de comprensión del código o cambiar su estructura y diseño y eliminar código muerto, para facilitar el mantenimiento en el futuro. Añadir nuevo comportamiento a un programa puede ser difícil con la estructura dada del programa, así que un desarrollador puede refactorizarlo primero para facilitar esta tarea y luego añadir el nuevo comportamiento.

3.3.2. Razones para la refactorización

Existen muchas razones por las que deberíamos adoptar esta técnica:

- **Calidad.** Es la razón primordial. Refactorizar es un proceso continuo de reflexión sobre nuestro código que permite que aprendamos de nuestros desarrollos en un entorno en el que no hay mucho tiempo para mirar hacia atrás. Un código de calidad es un código sencillo y bien estructurado, que cualquiera puede leer y entender sin necesidad de haber estado integrado en el equipo de desarrollo durante varios meses. Se debe terminar el tiempo en que imperaban los programas escritos en una sola línea en la que se hacía de todo, y en el que se valoraba la concisión aún a costa de la legibilidad.
- **Eficiencia.** Mantener un buen diseño y un código estructurado es sin duda la forma más eficiente de desarrollar. El esfuerzo que se invierta en evitar la duplicación de código y en simplificar el diseño se verá compensado cuando se tengan que realizar modificaciones, tanto para corregir errores como para añadir nuevas funcionalidades.
- **Diseño Evolutivo** en lugar de Gran Diseño Inicial. En muchas ocasiones los requisitos al principio del proyecto no están suficientemente especificados y debemos abordar el diseño de una forma gradual. Cuando tenemos unos requisitos claros y no cambiantes, un buen análisis de los mismos puede originar un diseño y una implementación brillantes; pero cuando los requisitos van cambiando según avanza el proyecto, y se añaden nuevas funcionalidades según se le van ocurriendo a los participantes o clientes, un diseño inicial deja de tener razón de ser. Refactorizar nos permitirá ir evolucionando el diseño según se incluyan nuevas funcionalidades, lo que implica muchas veces cambios importantes en la arquitectura.

- **Evitar la Reescritura de código.** En la mayoría de los casos, refactorizar es mejor que reescribir. No es fácil enfrentarse a un código que no conocemos y que no sigue los estándares que uno utiliza, pero eso no es una buena excusa para empezar de cero. Sobretodo en un entorno donde el ahorro de costos y la existencia de sistemas lo hacen imposible. De todas formas, no siempre es necesario modificar nuestro código, tiene que hacerse por una razón justificada. Solamente se necesitará refactorizar cuando se observe una mala estructuración de código o un mal diseño que impida futuros desarrollos. Cuando nos damos cuenta de que nuestro código está siendo duplicado y difícil de entender, entonces debemos recurrir a la refactorización. A veces, tenemos que dar algún paso atrás y replantear nuestro código, para posteriormente avanzar de manera rápida y eficaz.

3.3.3. Cuando refactorizar

No siempre es justificable modificar el código, no se reduce a una cuestión estética.

Tenemos que estar atentos a aquellas situaciones donde lo más inteligente es parar, para reorganizar el código. Se trata de dar algunos pasos hacia atrás que nos permitirán tomar carrerilla para seguir avanzando. Los síntomas que nos avisan de que nuestro código tiene problemas se conocen como *Bad Smells*.

Un programador experimentado puede intuir que su programa va camino a “oler mal” cuando hay:

- **Identifcadores ambiguos**. Ya sean nombres de variables, clases o métodos, los cuales será conveniente renombrarlos para clarificar el código. Por ejemplo, cambiar una variable que se llame 't' por una variable que se llame 'tiempo_de_espera'
- **Duplicated code (código duplicado)**. Es la principal razón para refactorizar. Si se detecta el mismo código en más de un lugar, se debe buscar la forma de extraerlo y unificarlo.
- **Long method (método largo)**: Legado de la programación estructurada. En la programación orientada a objetos cuando mas corto es un método más fácil de reutilizarlo es. Se suele dividir el programa en subprogramas.
- **Large class (clase grande)**. Si una clase intenta resolver muchos problemas, usualmente suele tener varias variables de instancia, lo que suele conducir a código duplicado.
- **Long parameter list (lista de parámetros extensa)**: en la programación orientada a objetos no se suelen pasar muchos parámetros a los métodos, sino sólo aquellos mínimamente necesarios para que el objeto involucrado consiga lo necesario. Éste tipo de métodos, los que reciben muchos parámetros, suelen variar con frecuencia, se tornan difíciles de comprender e incrementan el acoplamiento.
- **Feature envy (envidia de funcionalidad)**: un método que utiliza mas cantidad de elementos de otra clase que de la propia. Se suele resolver el problema pasando el método a la clase cuyos componentes son más requeridos para usar. Por tanto, es conveniente que a medida que se va desarrollando el código fuente del programa se tengan en cuenta si se están planteando estos síntomas para rediseñar nuestro código y adaptarlo a una solución que evite que se creen este tipo de situaciones. Por ejemplo:

```
package unit33;
import static java.lang.System.out;

public class SinRefactorizar
{
```

```
    public static void main(String[] args)
    {
        int suma1 = 0, suma2 = 0, resultado = 0;
        int array1[] = {1,2,3,4}, array2[] = {5,6,7,8};

        for(int i = 0; i < array1.length; i++){
            suma1 += array1[i];
        }
        resultado += suma1/array1.length;

        for(int i = 0; i < array2.length; i++){
            suma2 += array2[i];
        }
        resultado += suma2/array2.length;

        out.println("El resultado total es: " + resultado);
    }
}
```

En este código podemos detectar codigo duplicado, y podríamos refactorizarlo y dejarlo:

```
package unit33;
import static java.lang.System.out;

public class Refactorizada
{
    public static int calcularResultado(int []array){
        int suma = 0;
        for(int i = 0; i < array.length; i++){
            suma += array[i];
        }
        return suma/array.length;
    }
    public static void main(String args)
    {
        int array1[] = {1,2,3,4}, array2[] = {5,6,7,8}, resultado = 0;

        resultado = calcularResultado(array1);
        resultado += calcularResultado(array2);

        out.println("El resultado total es: " + resultado);
    }
}
```

Y si observamos el siguiente código:

```
ackage unit33;
import static java.lang.System.*;

import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;

public class LongMethod
{
    public static void main(String args[])
    {
        int array[] = new int[10];
        BufferedReader entrada = new BufferedReader(
                                    new InputStreamReader(System.in));
        int resultado = 0;

        // Inicialización del array
        for(int i = 0; i < array.length; i++)
        {
            out.printf("Introduce valor para array[%d]", i);
            try {
                array[i] = Integer.valueOf(entrada.readLine());
            } catch (IOException e) {
                err.println("Error: " + e.getMessage());
            }
        }
        out.println("Suma: ");
        // Suma del array
        for(int i = 0; i < array.length; i++)
        {
            resultado = resultado + array[i];
            out.print(array[i] + " " + (i==array.length-1?" ":"+ "));
        }
        out.println("= " + resultado);
    }
}
```

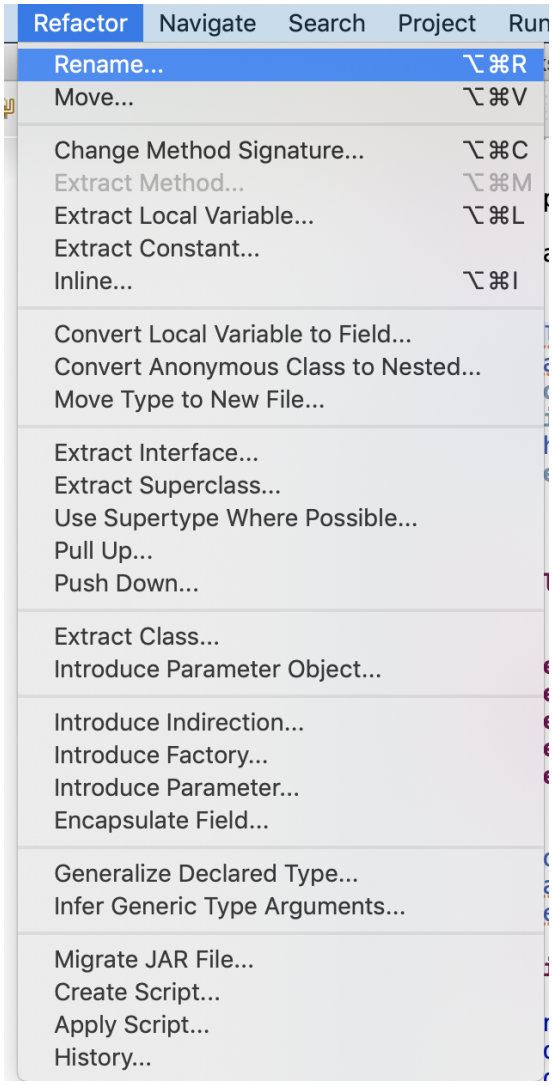
Vemos un método demasiado largo que puede dividirse en funciones.

```
package unit10;

import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import static java.lang.System.*;

public class NoLongMethod
{
```

```
    private static int array[] = new int[10];

    public static void inicializar()
    {
        BufferedReader entrada = new BufferedReader(new
InputStreamReader(System.in));

        for(int i = 0; i < array.length; i++)
        {
            out.printf("Introduce valor para array[%d]", i);
            try
            {
                array[i] = Integer.valueOf(entrada.readLine());
            } catch (IOException e) {
                err.println("Error: " + e.getMessage());
            }
        }
    }
    public static void sumaArray()
    {
        int resultado = 0;

        out.println("Suma: ");

        // Suma del array
        for(int i = 0; i < array.length; i++)
        {
            resultado = resultado + array[i];
            out.print(array[i] + " " + (i==array.length-1?" ":"+ "));
        }
        out.println("= " + resultado);
    }

    public static void main(String args[])
    {
        inicializar();
        sumaArray();
    }
}
```

3.3.4. Refactorizar en Eclipse

En el menú principal podemos encontrar la opción de ***Refactor*** con una serie funcionalidades para refactorizar nuestro código.



Mediante estas opciones podemos:

- Renombrar variables, clases, métodos ...
- Mover clases de un paquete a otro.
- Extraer una variable local a partir de una expresión.
- Convertir un número o cadena literal en una constante.
- Transformar una variable local en un atributo privado de una clase.
- Extraer un interface a partir de métodos de una clase.
- Extraer superclase con los métodos que pasarán a ser de la superclase.
- Extraer método nos permite convertir un trozo de código en un método.
- ...

3.3.5. Patrones de diseño

Existen unas herramientas muy útiles a la hora de refactorizar (o para evitar excesiva refactorización) que se llaman patrones de diseño.

Un Patrón de Diseño (design pattern) es una solución repetible a un problema recurrente en el diseño de software.

Las ventajas del uso de patrones son:

- Conforman un amplio catálogo de problemas y soluciones
- Estandarizan la resolución de determinados problemas
- Condensan y simplifican el aprendizaje de las buenas prácticas
- Proporcionan un vocabulario común entre desarrolladores
- Evitan “reinventar la rueda”

3.3.5.1. Tipos de patrones de diseño

Según la finalidad del patrón, estos se clasifican en tres tipos:

- **Patrones Creacionales**: Estos patrones hacen hincapié en la encapsulación de la lógica de la instanciación, ocultando los detalles concretos de cada objeto y permitiéndonos trabajar con abstracciones.
- **Patrones Estructurales**: Los patrones estructurales nos ayudan a definir la forma en la que los objetos se componen.
- **Patrones de comportamiento**: Los patrones comportamentales nos ayudan a definir la forma en la que los objetos interactúan entre ellos.

3.3.5.2. Algunos ejemplos: *Factory Pattern*

El patrón **Factory** es uno de los patrones más usados en Java. Este tipo de patrón de diseño correspondería a los creacionales y proporciona una de las mejores maneras de crear un objeto. Utilizando este patrón podemos crear un objeto sin exponer la lógica de la creación al cliente sino que se utiliza un interfaz común.

Por ejemplo si recordamos el ejercicio 2.7.2.1. donde teníamos un interfaz *Shape* que implementaban diferentes formas como *Circle*, *Square* o *Rectangle*. A la hora de crear uno de estos elementos debemos llamar al constructor concreto:

```
System.out.println(
            "Introduzca radio/lado/altura radio/lado/base y forma a
dibujar");
    par1=sc.nextFloat();
    par2=sc.nextFloat();
    shapeUser=sc.nextLine().trim();

    if(shapeUser.equalsIgnoreCase("CIRCLE"))
    {
        myShape=new Circle(par1);
    }
    else
        if(shapeUser.equalsIgnoreCase("SQUARE"))
        {
            myShape=new Square(par1);
        }
        else
            if(shapeUser.equalsIgnoreCase("RECTANGLE"))
```

```
            {
                myShape=new Rectangle(par1,par2);
            }
    sc.close();
    myShape.draw();
```

De esta forma el desarrollador de esta parte de la aplicación debe conocer el nombre de las clases de todas las formas y si se van agregando nuevas formas habría que ir añadiendo cada vez más opciones a este código.

Con la utilización del patrón Factory, encapsulamos todas las formas que implementan el interface Shape de forma que desde fuera se pasará qué forma queremos crear y este patrón nos proporciona una instancia del objeto correcto. Para ello primero construimos la *fábrica de formas*:

```
public class ShapeFactory {

    public Shape getShape(String shapeType,float par1,float par2){
            if(shapeType == null){
                return null;
            }
            if(shapeType.equalsIgnoreCase("CIRCLE")){
                return new Circle(par1);

            } else if(shapeType.equalsIgnoreCase("RECTANGLE")){
                return new Rectangle(par1,par2);

            } else if(shapeType.equalsIgnoreCase("SQUARE")){
                return new Square(par1);
            }

            return null;
        }

}
```

De esta forma cuando queramos construir alguna de estas formas no tenemos por qué saber su nombre de clase específico, sino pasarle que forma queremos instanciar.

```
ShapeFactory shapeFactory = new ShapeFactory();

    Shape shape1 = shapeFactory.getShape(
                        shapeUser.toUpperCase(),par1,par2);
    shape1.draw();
    System.out.println(shape1.calculateArea());
```

Para el usuario solo existe una clase (o interfaz en este caso) *Shape* y no tiene por qué conocer la existencia del resto de clases.

Como se puede observar este patrón es útil cuando hay una jerarquía de clases algo complicada o gran cantidad de clases parecidas que implementan un mismo interfaz como el ejemplo anterior e incluso descienden de la misma clase abstracta como podría ser el ejemplo siguiente.

Tenemos una empresa que vende productos a los que se le aplica el IVA general y otros productos a los que se les aplica el IVA reducido. Así que tenemos una clase abstracta Factura de quien heredan las facturas con iva reducido y con iva general.

```
public abstract class Invoice {

private int id;
private double amount;


public int getId()
{
    return id;
}

public void setId(int id)
{
    this.id = id;
}

public double getAmount()
{
    return amount;
}

public void setAmount(double amount)
{
    this.amount = amount;
}

    public abstract double getAmountIva();
}
```

Aquí tenemos la clase abstracta para todas las facturas, y heredando de ella tenemos:

```
public class InvoiceIva extends Invoice
{
    @Override
        public double getAmountIva()
        {
```

```
            return getAmount()*1.21;
        }
}

public class InvoiceIvaRed extends Invoice{

    @Override
        public double getAmountIva()
        {
            return getAmount()*1.10;
        }
}
```

De esta forma cuando queramos instanciar una nueva factura solo necesitamos conocer la existencia de la clase facturas y no la de los distintos tipos de facturas.

```
    Invoice myInvoice=FactoryInvoice.getInvoice("iva");
    Invoice myInvoiceRed=FactoryInvoice.getInvoice("red");

    myInvoice.setId(1);
    myInvoice.setAmount(1000);
    myInvoiceRed.setId(2);
    myInvoiceRed.setAmount(500);

    System.out.println(myInvoice.getAmountIva());
    System.out.println(myInvoiceRed.getAmountIva());
```

Ejercicios propuestos:

3.3.5.1. Implementa el *Factory pattern* al proyecto *Shapes* como está en el ejemplo y pruébalo.

3.3.5.2. Crea el ejemplo anterior en un proyecto llamado *Invoices* con los *packages invoices.types* para las clases de facturas y *invoices.main* con el main para probarlo. Añade a su vez un nuevo tipo de facturas con un tipo de iva superreducido (4%).

3.3.5.3. Modifica el ejercicio **2.7.1.1.** y añade una factoría de animales al proyecto de forma que no llamemos a los constructores de los distintos animales sino que le pidamos una instancia del animal en cuestión.

3.3.5.3. Algunos ejemplos: *Singleton*

Este patrón sirve para permitir crear un solo objeto de una clase determinada. Este patrón es útil por ejemplo para clases donde se guardan parámetros de configuración de las que sólo debe haber un objeto compartido por todo el sistema. Como la configuración de acceso a la base de datos, o datos como contadores de facturas compartidos por todos los programas que generan facturas nuevas.

El método consiste en tener en la clase una instancia estática de la propia clase que será la que se devolverá cuando se pida una instancia a esta clase y el constructor privado de forma que no se puedan crear más objetos de esta clase.

Por ejemplo si tenemos una clase de contadores de facturas donde se guardan las últimas facturas generadas:

```
public class GeneralCounter
{
    private int lastInvoice;
    private int lastDeliveryNote;
}
```

Añadimos como atributo estático una instancia a la misma clase;

```
    public static GeneralCounter myCounter;
```

Y el constructor privado

```
    private GeneralCounter(int lastInvoice,int lastDeliveryNote)
    {
        this.lastDeliveryNote=lastDeliveryNote;
        this.lastInvoice=lastInvoice;
    }
```

Y para poder obtener una instancia de esta clase:

```
    public static GeneralCounter getCounter(int lastInvoice,int
lastDeliveryNote)
    {
        if(myCounter==null)
        {
            myCounter=new GeneralCounter(lastInvoice,lastDeliveryNote);
        }
        return myCounter;
    }
```

De esta forma sólo existirá un objeto de esta clase en el sistema. Y luego si queremos utilizar sus datos:

```
public static void main(String[] args) {

    GeneralCounter counter=GeneralCounter.getCounter(0,0);
```

```
    Invoice myInvoice=FactoryInvoice.getInvoice("iva");
    Invoice myInvoiceRed=FactoryInvoice.getInvoice("red");

    counter.setlastInvoice(counter.getLastInvoice()+1);
    myInvoice.setId(counter.getLastInvoice());
    myInvoice.setAmount(1000);
    counter.setlastInvoice(counter.getLastInvoice()+1);
    myInvoiceRed.setId(counter.getLastInvoice());
    myInvoiceRed.setAmount(500);

    System.out.println(myInvoice.getAmountIva());
    System.out.println(myInvoiceRed.getAmountIva());
}
```

> **Ejercicios propuestos:**
>
> **3.3.5.4.** Añade a la clase *GeneralCounter* unas variables para guardar los tipos de iva. Y modifica después los métodos ***getAmountIva*** de los distintos tipos de factura para que accedan a este objeto a consultar el tipo de iva a utilizar en cada ocasión.

3.6 Más información

Patrones de diseño