Object oriented programming

Refactoring and software patterns

1. Code refactoring

In software engineering, *refactoring* often refers to a code update that does not affect its behavior, which is also known as *cleaning the code*. Refactoring is used as part of the software development process: developers sometimes add new functionalities and test cases, and sometimes refactor their code to make it clearer and more robust. In this case, running the tests again may prove that refactoring has not changed the behavior of the application.

So refactoring is a part of software maintenance that does not fix any bug nor add any new feature. The main target is to improve code comprehension or change its structure to ease its future maintenance. Sometimes it is difficult to add new functionalities to a program with its current code structure, so a developer can refactor it before adding more new code.

1.1. Why to refactor?

There are many reasons why we should use this technique:

- **Quality.** It is the main reason. Refactoring is a continuous thinking over our code in an environment where we don't have too much time to look back. Good code is simple and well structured, and anyone can read it and understand it without being part of the development team. We should not come back to those programs written in a single line, where conciseness was better than readability.
- **Efficiency.** Keeping a good design and a structured code is the most efficient way to develop a program. The effort that we put into avoiding duplicated code and simplifying the design will be retrieved when we need to make later modifications, either to fix some bugs or to add new features.
- **Evolutive Design.** Instead of an initial Big Design, sometimes the requirements are not clearly specified at the beginning, so we must face the design gradually. New functionalities can be added as we are implementing the project, so the initial design can become useless. Refactoring lets us make the design evolve as new functionalities join the old ones. This often implies important changes in the architecture of the project.
- **Avoid rewriting code.** In most cases, refactoring is better than rewriting. It is not easy to face a code that is not ours, and does not follow our own standards, but this is not a good reason to start from scratch. Especially in an environment where cost savings make it impossible. Anyway, updating our code is not always necessary. There must be a good reason, and we only need to make an update if there is a bad design that difficults future developments. Whenever we notice that our code is difficult to understand, or there is duplicated code, then we need to refactor it. Sometimes we may need to step back and rethink some aspects, so that we can move on quickly and easily.

1.2. When to refactor?

Refactoring code is not due to aesthetic reason. We must pay attention to some situations in which it is better to stop and rearrange our code. The elements that warn us that our code is in trouble are known as *Bad Smells*. An experienced programmer can determine that his/her program is starting to "stink" when there are:

- **Ambiguous identifiers**. They can be either variable, class or method names. We will need to rename them to clarify our code. For instance, we can rename a variable called `t` with another one called `waitTime`.
- **Duplicated code**. This is the main reason to refactor our code. If we detect the same piece of code in more than one place, we need to unify it.
- **Long method**. This is a very usual feature of structured programming, but in object oriented programming, the shorter a method is, the easier we can use it. So we can divide the main program in sub-programs.
- **Large class**. If a single class tries to solve too many problems, it may also have too many instance variables, which can lead to duplicated code.
- **Long parameter list**. We should not pass too many parameters to our methods. We should only use those really needed for the method to complete its task. Those methods who get too many input parameters change their behavior very often, and become very difficult to understand.
- **Feature envy**. It happens when a method uses more elements of another class than from its own class. This problem is usually solved by moving this method to the other class.

Therefore, we must take into account these factors as we are implementing the program, so that we can refactor it and avoid these situations. Let's have a look at this example:

```
public class NoRefactor
{
    public static void main(String[] args)
    {
        int sum1 = 0, sum2 = 0, result = 0;
        int array1[] = {1,2,3,4}, array2[] = {5,6,7,8};

        for(int i = 0; i < array1.length; i++)
        {
            sum1 += array1[i];
        }
        result += sum1/array1.length;

        for(int i = 0; i < array2.length; i++)
        {
            sum2 += array2[i];
        }
        result += sum2/array2.length;

        System.out.println("The result is: " + result);
    }
}
```

We can easily find duplicated code in this example, so we can refactor it like this:

```
public class Refactor
{
    public static int calculateResult(int []array)
    {
        int sum = 0;
        for(int i = 0; i < array.length; i++)
        {
            sum += array[i];
        }
        return sum/array.length;
    }

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

        result = calculateResult(array1);
        result += calculateResult(array2);

        System.out.println("The result is: " + result);
    }
}
```

Now let's have a look at this code:

```
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 input = new BufferedReader(
            new InputStreamReader(System.in));
        int result = 0;

        // Array initialization
        for(int i = 0; i < array.length; i++)
        {
            System.out.printf("Enter value for array[%d]", i);
            try
            {
                array[i] = Integer.valueOf(input.readLine());
            } catch (IOException e) {
                err.println("Error: " + e.getMessage());
            }
        }

        // Sum of the array
        System.out.println("Sum: ");
        for(int i = 0; i < array.length; i++)
        {
            result = result + array[i];
            System.out.print(array[i] + " " +
                (i==array.length-1?" ":"+ "));
        }
        System.out.println("= " + result);
    }
}
```

`main` method is too large, and we can divide it into functions.

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

public class NoLongMethod
{
    public static int[] initialize()
    {
        int array[] = new int[10];

        BufferedReader input = new BufferedReader(
            new InputStreamReader(System.in));

        for(int i = 0; i < array.length; i++)
        {
            System.out.printf("Enter value for array[%d]", i);
            try
            {
                array[i] = Integer.valueOf(input.readLine());
            } catch (IOException e) {
                err.println("Error: " + e.getMessage());
            }
        }

        return array;
    }

    public static void sumArray(int[] array)
    {
        int result = 0;

        System.out.println("Sum: ");

        // Sum of the array
        for(int i = 0; i < array.length; i++)
        {
            result = result + array[i];
            System.out.print(array[i] + " " +
                (i==array.length-1?" ":"+ "));
        }
        System.out.println("= " + result);
    }

    public static void main(String args[])
    {
        int[] array = initialize();
        sumArray(array);
    }
}
```

1.3. Refactoring in IntelliJ

In **IntelliJ**, we can find a *Refactor* main menu with some options to refactor our code. Also, we can right click over any element of the code (variable, method, class name...) and choose *Refactor* option. With these options we can:

- *Refactor* > *Rename*: Rename variables, classes, methods... and apply these changes to the whole code
- *Refactor* > *Move*: Move classes/members from one package/class to another.
- *Refactor* > *Introduce constant*: Convert a number or literal string into a constant.
- *Refactor* > *Introduce field*: Transform a local variable into a private attribute of the class.
- *Refactor* > *Extract interface*: Extract an interface from the methods of a class.
- *Refactor* > *Extract superclass*: Extract a superclass with methods that will belong to this superclass.
- ...

2. Design patterns

There are some useful techniques that we can apply when we need to refactor our code: the design patterns. They are a repeteable solution for a problem in software design.

The main advantages in using design patterns are:

- They constitute a wide catalog of solutions to problems
- They standardize how to resolve some given problems
- They simplify the learning of good programming practices
- They provide a common vocabulary among developers
- They avoid reinventing the wheel

2.1. Types of design patterns

According to the purpose, we can divide design patterns into these categories:

- **Creational patterns**: They encapsulate the logic of the object instantiation, hiding the concrete details of every object. This way, we can just work with abstractions.
- **Structural patterns**: They help us define how objects are composed.
- **Behavioral patterns**: They help us define how objects interact between them.

2.2. Factory Pattern

Factory pattern is one of the most used patterns in Java. It is a creational pattern that provides one of the best ways of creating objects. Using this pattern lets us create objects without showing the logic of the creation to the user. We just use a common interface.

For instance, let's go back to the *Shapes* example of previous documents. We had a *Shape* interface that was implemented by some classes, such as *Circle*, *Square* or *Rectangle*. Whenever we wanted to create an object, we had to call a concrete constructor:

```
Shape myShape = new Circle(3);
...
Shape myShape2 = new Square(5);
...
myShape.draw();
...
```

Therefore, developer must know the name of every class implementing *Shape* interface. But, if we use *Factory* pattern, we can encapsulate all the possible ways of creating a shape, so that we just need to specify which shape do we need, and the pattern will provide an instance of the concrete shape.

First step is to define the *shape factory*. This typically consists in creating a new class with a static method which is in charge of creating the objects of a given class or class hierarchy. In our case, we define a static method to create *Shape* subtypes:

```
public class ShapeFactory
{
    public static Shape getShape(ShapeType type, float param1, float param2)
    {
        if(type == ShapeType.CIRCLE){
            return new Circle(param1);
        } else if(type == ShapeType.RECTANGLE){
            return new Rectangle(param1,param2);
        } else if(type == ShapeType.SQUARE){
            return new Square(param1);
        }

        return null;
    }
}
```

This way, if we want to create an instance of any shape, we just need to call this factory with the name of the shape we want to create, and its parameters

```
Shape shape1 = ShapeFactory.getShape(ShapeType.CIRCLE,3, 0);
shape1.draw();
System.out.println(shape1.calculateArea());
```

Regarding `ShapeType`, it's just an enum where we can specify all the subtypes that we want to manage:

```
public enum ShapeType { CIRCLE, SQUARE, RECTANGLE }
```

From the developer's point of view, there's only one class (or interface, in this case), which is "Shape", and he doesn't need to know anything about the rest of implementing classes. This design pattern is really useful when there is a complex class hierarchy with a common instantiation pattern (similar parameters), or a huge bunch of classes which are really similar and implement the same parent interface or abstract class, as we can see in the following example.

We have a company that sells some products. We must apply the general VAT to some products, and a reduced VAT to some others. So we start by defining an abstract class called `Invoice` *with two subclasses to represent these two types of VAT.*

```
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 getAmountVAT();
}
```

These are the subclasses:

```
public class InvoiceVAT extends Invoice
{
    @Override
    public double getAmountVAT()
    {
        return getAmount()*1.21;
    }
}

public class InvoiceVATReduced extends Invoice
{
    @Override
    public double getAmountVAT()
    {
        return getAmount()*1.10;
    }
}
```

This way, we can define an `InvoiceType` enum to identify the enum types:

```
public enum InvoiceType { NORMAL, REDUCED }
```

And also an `InvoiceFactory` class to create one of the invoice types depending on the parameters:

```
public class InvoiceFactory
{
    public static Invoice getInvoice(InvoiceType type)
    {
        if (type == InvoiceType.NORMAL)
        {
            return new InvoiceVAT();
        } else if (type == InvoiceType.REDUCED) {
            return new InvoiceVATReduced();
        } else {
            return null;
        }
    }
}
```

Finally, we can use this factory to create the invoices:

```
Invoice myInvoice = FactoryInvoice.getInvoice(InvoiceType.NORMAL);
Invoice myInvoiceRed = FactoryInvoice.getInvoice(InvoiceType.REDUCED);

myInvoice.setId(1);
myInvoice.setAmount(1000);

myInvoiceRed.setId(2);
myInvoiceRed.setAmount(500);

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

> **Exercise 1:**
>
> Implement the *Factory* pattern in the Exercise 6 of this document as we have shown in previous example and test it.

> **Exercise 2:**
>
> Create a new project called **Invoices** with the packages `invoices.types` for the invoice types, and `invoices.main` with the main class. Implement the classes of the invoices example above, and add a new type of invoice with a super-reduced VAT (4%).

> **Exercise 3:**
>
> Update the *Animals* project of Exercise 1 of this document and add an *animal factory* so that we no longer use the constructor of each animal type.

2.3. Singleton pattern

This pattern is also a creational pattern that lets us manage just one single object of a given class. It is commonly used in classes that store configuration parameters for a given application, so that there should only be one object of this type for the whole application, and this object must be shared among all the components of the application.

Singleton pattern consists in having a static instance of the class itself, which will be returned every time we ask for a new instance from every part of the code. There will also be a private constructor that will be called only once to create the static attribute the first time it is needed.

For instance, if we have a class to count the total number of invoices:

```
public class InvoiceCounter
{
    private int counter;

    public void increaseCounter()
    {
        counter++;
    }

    public int getCounter()
    {
        return counter;
    }
}
```

We can add a (private) static attribute that points to the same class:

```
private static InvoiceCounter myCounter;
```

And a private constructor:

```
private InvoiceCounter()
{
    this.counter = 0;
}
```

In order to have an instance of this class, we need to add a public, static method, such as *getInstance* or, in this case, *getCounter*:

```
public static InvoiceCounter getInvoiceCounter()
{
    if(myCounter == null)
    {
        myCounter = new InvoiceCounter();
    }
    return myCounter;
}
```

Then, there will only be one single instance of this object in the system. If we want to use it to deal with some invoices, we can do it like this:

```
public static void main(String[] args)
{
    InvoiceCounter counter = InvoiceCounter.getCounter();
    Invoice myInvoice =
        InvoiceFactory.getInvoice(InvoiceType.NORMAL);
    Invoice myInvoiceRed =
        InvoiceFactory.getInvoice(InvoiceType.REDUCED);

    counter.increaseCounter();
    myInvoice.setId(counter.getCounter());
    myInvoice.setAmount(1000);

    counter.increaseCounter();
    myInvoiceRed.setId(counter.getCounter());
    myInvoiceRed.setAmount(500);

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

Exercise 4:

Add the `InvoiceCounter` class to the *Invoices* project started in previous exercises. Make the appropriate changes to automatically increase the invoice counter as soon as we instantiate a new invoice from `InvoiceFactory`, so we don't need to increase it manually from the *main* method, as we did in previous example.