UT4-OPTIMIZACIÓN

1. Introducción

Cuando el programados está comenzando, realiza los programas de la única forma que sabe, pero, cuando el desarrollador mejora, elige entre todas las posibles soluciones la más eficiente, efectiva, mantenible, etc.

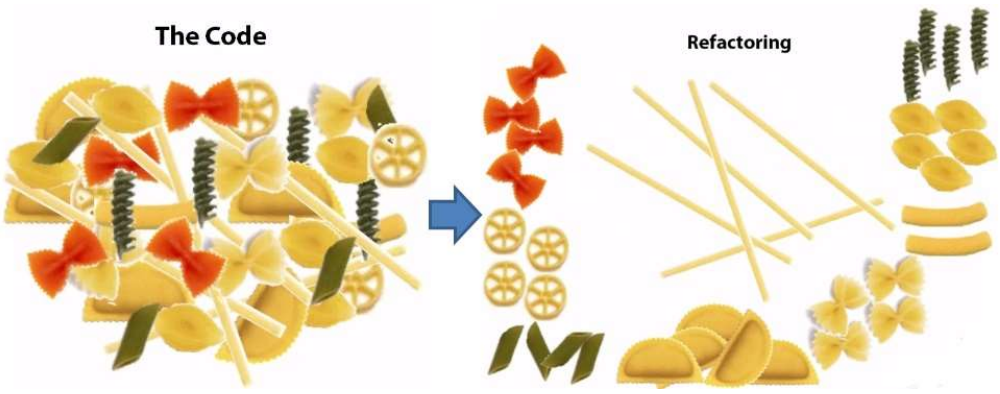
Además, actualmente, es raro participar en un proyecto sin utilizar un SCV o sistema de control de versiones. En esta unidad se abordarán sus principios básicos y se verá algo más en profundidad Git, que es una de las herramientas de control de versiones más utilizada actualmente.

2. REFACTORIZACIÓN

La refactorización es una técnica de la ingeniería del software que permite la optimización de un código previamente escrito, por medio de cambios en su estructura interna sin que esto suponga alteraciones en su comportamiento externo.

Por lo tanto, la refactorización nunca modificará el aspecto externo de una clase o programa

(el comportamiento del sw deberá ser siempre el mismo).



Dicho de otro modo, la refactorización no busca ni arreglar errores ni añadir nueva funcionalidad, sino mejorar la comprensión del código para facilitar así nuevos desarrollos, la resolución de errores o la adicción de alguna funcionalidad al software.

La refactorización tiene como objetivo limpiar el código para que sea más fácil de entender y modificar, permitiendo una mejor lectura para comprender qué es lo que se está realizando.

Después de refactorizar, el proyecto seguirá ejecutándose igual y obteniendo los mismos resultados.

Ventajas de la refactorización

a) Ayudará al programador a encontrar errores de una forma más rápida y sencilla. Muchas veces, una estrategia equivocada al programar implica errores y fallos internos a la larga.
b) Ayudará a programar más rápido. Seguramente, al encontrar menos errores, se perderá menos tiempo en corregir y arreglar código.
c) Los programas se leerán e interpretarán de una manera más fácil
d) Los diseños serán más robustos.
e) Los programas tendrán más calidad
f) Se evitará duplicar la lógica de los programas.

Diferencia entre refactorización y optimización

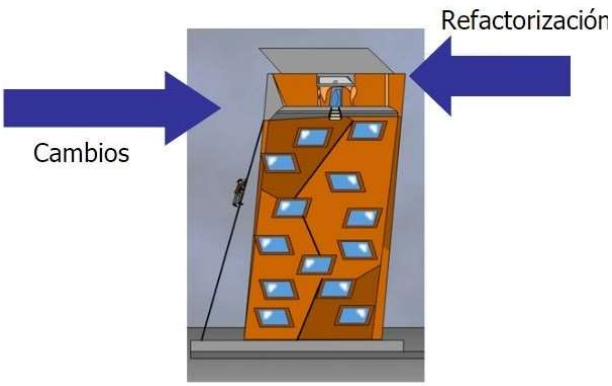
Hay que diferenciar la refactorización de la optimización. En ambos procesos, se pretende mejorar la estructura interna de una aplicación o componente, sin modificar su comportamiento. Sin embargo, cuando se optimiza, se persigue una mejora del rendimiento, por ejemplo, mejorar la velocidad de ejecución, pero esto puede hacer un código más difícil de entender.

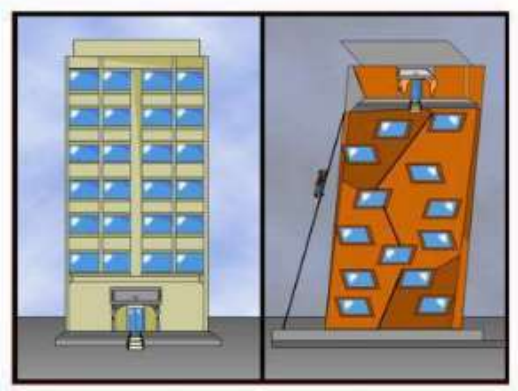
Hay que resaltar que la refactorización no cambia el comportamiento observable del software. El software sigue cumpliendo la misma función que hacía antes. Ningún usuario, ya sea usuario final u otro programador, podrá determinar qué cosas han cambiado.

¿Qué hace la refactorización?

- Limpia el código, mejorando la consistencia y la claridad.
- Mantiene el código, no corrige errores ni añade funcionalidades nuevas.
- Va a facilitar la realización de cambios en el código.
- Se obtiene un código limpio y altamente modularizado.

¿Si su software fuera un edificio, se parecería mas
a uno de la izqda. o de la dcha.?





Tipos de Refactorización

- Revisar constantemente el código
- Hacer un checklist de defectos a refactorizar
- Hacer refactorizaciones pequeñas, una a la vez
- Agregar casos de prueba o pruebas unitarias
- Revisar los resultados

3.1Cuando refactorizar. Malos olores (*bad smells*)

La refactorización se debe ir haciendo mientras se va realizando el desarrollo de la aplicación.

Los síntomas que indican la necesidad de refactorizar (o eliminar los malos olores) son los siguientes:

- **Código duplicado** *(duplicated code):* es la principal razón para refactorizar. Si se detecta el mismo más de un lugar, se debe buscar la forma de extraerlo y unificarlo.
- **Métodos muy largos** *(long method):* cuanto más largo es un método más difícil es de entender. Un método muy largo normalmente está realizando tareas que deberían ser responsabilidad de otros. Se deben identificar estos métodos y descomponerlos en otros más pequeños. En la programación orientada a objetos cuando más corto es un método más fácil es de reutilizarlo.
- **Clases muy grandes** (*large class*): si una clase intenta resolver muchos problemas, tendremos una clase con demasiados métodos, atributos o incluso instancias. La clase está asumiendo demasiadas responsabilidades. Hay que intentar hacer clases pequeñas, de forma que cada una trate con un conjunto pequeño de responsabilidades bien delimitadas.
- **Listas de parámetros extensa** *(long parameter list):* en la programación orientada a objetos no se suelen pasar muchos parámetros a los métodos, sino solo aquellos mínimamente necesarios para que el objeto involucrado consiga lo necesario. Tener demasiados parámetros puede estar indicando un problema de encapsulación de datos o la necesidad de crear una clase de objetos a partir de varios de esos parámetros, y pasar ese objeto como argumento en vez de todos los parámetros. Especialmente si esos parámetros suelen tener que ver con otros y suelen ir juntos siempre.

- **Cambio divergente** *(divergent change):* una clase es frecuentemente modificada por diversos motivos, los cuales no suelen estar relacionados entre sí, a lo mejor conviene eliminar la clase. Este síntoma es el opuesto del siguiente.
- **Cirugía a tiro pistola** (*shotgun surgery*): este síntoma se presenta cuando después de un cambio en una determinada clase, se deben realizar varias modificaciones adicionales en diversos lugares para compatibilizar dicho cambio.
- **Envidia de funcionalidad** (*feature envy):* se observa este síntoma cuando tenemos un método que utiliza más cantidad de elementos de otra clase que de la suya propia. Se suele resolver el problema pasando el método a la clase cuyos elementos utiliza más.
- **Clase de solo datos** *(data class):* clases que solo tienen atributos y métodos de acceso a ellos ("*get*" y "*set*"). Este tipo de clases deberían cuestionarse dado que no suelen tener comportamiento alguno.
- **Legado rechazado** *(refused bequest*): este síntoma lo encontramos en subclases que utilizan solo unas pocas características de sus superclases. Si las subclases no necesitan o no requieren todo lo que sus superclases les proveen por herencia, esto suele indicar que como fue pensada la jerarquía de clases no es correcto. La delegación suele ser la solución a este tipo de inconvenientes.

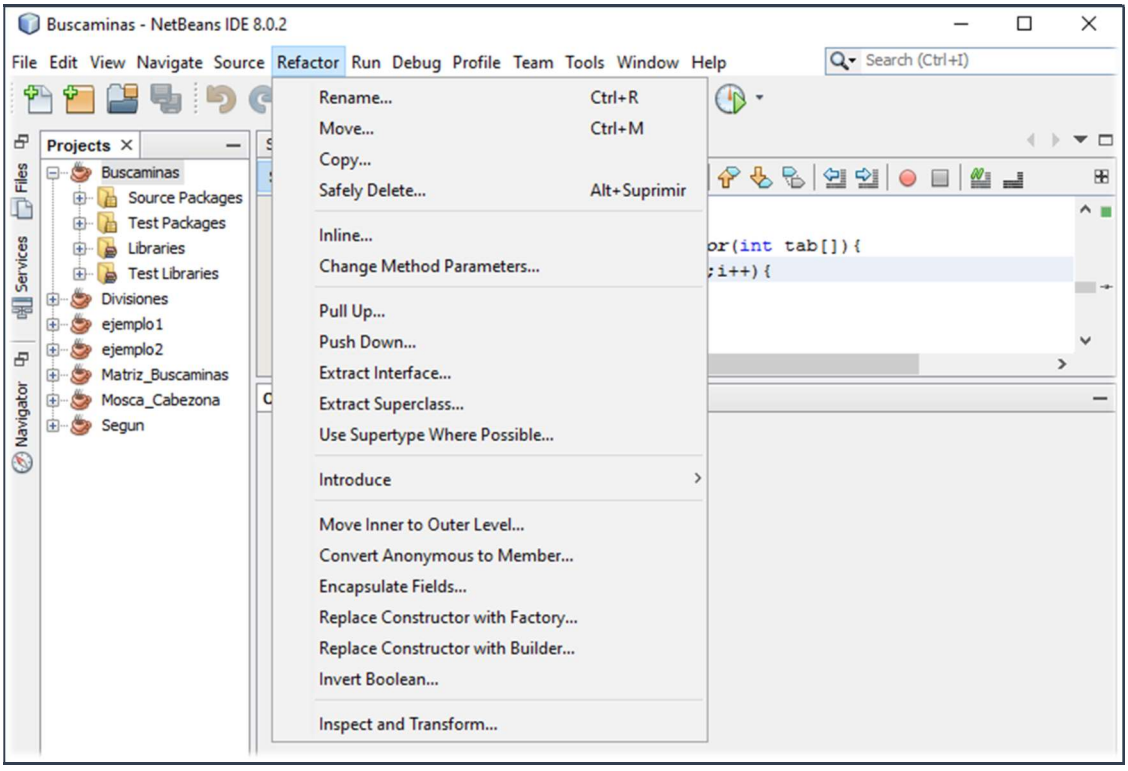
El proceso de refactorización presenta entre las ventajas mencionadas anteriormente, el mantenimiento del diseño del sistema, incremento de facilidad de lectura y comprensión del código fuente, detección temprana de fallos, aumento en la velocidad en la que se programa.

En cambio existen áreas conflictivas en la refactorización, como son las bases de datos y las interfaces. Un cambio de bases de datos es muy costoso pues los sistemas están muy acoplados a las bases de datos, y sería necesaria una migración tanto de estructura como de datos.

http://www.javamexico.org/blogs/oscarryz/refactorizacion_mejorando_el_diseno_del_codigo_existente

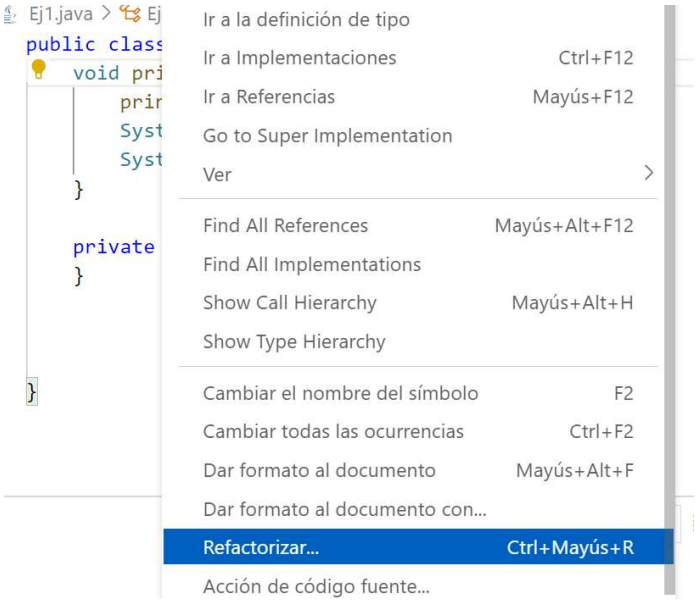
2.2 Refactorización en NetBeans

NetBeans tiene diversos métodos de refactorización o *refactoring*. Dependiendo de dónde invoquemos a la refactorización tendremos un menú contextual u otro con sus diferentes opciones, según sea una Clase, un Método o un Atributo. Para refactorizar, elegimos ***Refactor*** del menú o bien desde el menú contextual.



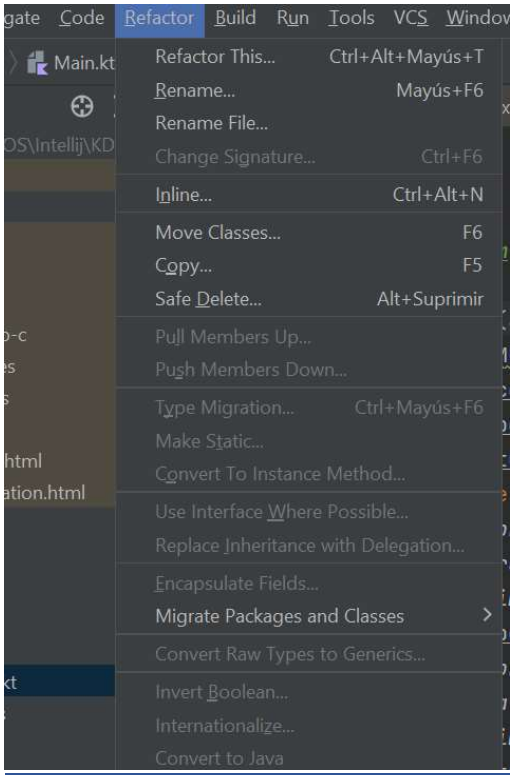
2.3 Refactorización en Visual Studio Code

En Visual Studio Code, podemos utilizar Ctr+mays+R o botón derecho sobre lo que queremos refactorizar y refactorización, como se muestra en la figura:



2.4 Refactorización en IntelliJ

Aparece en el menú principal



3.4MÉTODOS DE REFACTORIZACIÓN

Los ***métodos de refactorización*** son las prácticas para refactorizar el código, utilizando las herramientas podremos plantear casos para refactorizar y se mostrarán las posibles soluciones en las que podemos ver el antes y después de refactorizar. A los métodos de refactorización también se les llama ***patrones de refactorización*** o ***catálogos de refactorización***.

Para refactorizar el elemento (puede ser una clase, una variable, una expresión, un bloque de instrucciones, un método, etc.) se pulsa el botón derecho del ratón, se selecciona ***Refactor*** y, seguidamente se selecciona el método de refactorización.

Algunos de estos métodos son:

- **Rename:** es una de las opciones más utilizada. Cambia el nombre de las variables, clases, métodos, paquetes, directorios y casi cualquier identificador Java. Tras la refactorización, se modifican las referencias a ese identificador.
- **Move:** mueve una clase de un paquete a otro, se mueve el archivo .java a la carpeta, y se cambian todas las referencias. También se puede arrastrar y soltar una clase a un nuevo paquete, se realiza una refactorización automática.

- **Safely Delete**: borra un elemento de todo el proyecto, haciendo un rastreo de todos los lugares en los que aparece.
- **Inline:** nos permite ajustar una referencia a una variable o método con la línea en la que se utiliza y conseguir así una única línea de código. Cuando se utiliza, se sustituye la referencia a la variable o método con el valor asignado a la variable o la aplicación del método respectivamente. Por ejemplo pasar directamente los parámetros en vez de usar variables auxiliares.
- **Extract Interface**: permite escoger los métodos una clase para crear un interface. Una Interface es una especie de plantilla que define los métodos acerca de lo que puede o no hacer una clase. La Interface define los métodos pero no los desarrolla. Serán las clases que implementen la Interface quien desarrolle los métodos.
- **Extract Superclass**: este método permite extraer una superclase. Si la clase ya utilizaba una superclase, la recién creada pasara a ser su superclase. Se pueden seleccionar los métodos y atributos que formarán parte de la superclase. En la superclases, los métodos están actualmente allí, así que si hay referencias a campos de clase origen, habrá fallos de compilación.

Patrones de refactorización más usuales

En este apartado, se verán formas de refactorización con ejemplos concretos.

a) **Extract method o reducción lógica**

Código antes de refactorizar:

```
public class Ej1 {
    void printCuenta(String nombre, double cantidad){
        printLogo();
        System.out.println ("nombre:" +nombre);
        System.out.println("cantidad:"+ cantidad);
    }
```

Código después de refactorizar:

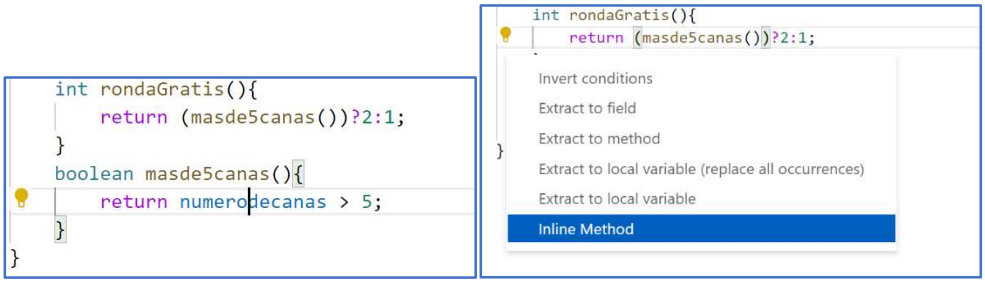
```
public class Ej1 {
    void printCuenta(String nombre, double cantidad){
        printLogo();
        extracted(nombre, cantidad);
    }

    private void extracted(String nombre, double cantidad) {
        System.out.println ("nombre:" +nombre);
        System.out.println("cantidad:"+ cantidad);
    }
```

Este tipo de refactorización es una de las más sencillas y típicas. Cuando se escriba código, lo que hay que crear son métodos que hagan lo que se dicen . Muchas veces, los métodos cortos tienen muchas ventajas:

- Realizan tareas específicas concretas (con lo cual pueden ser invocados por otros métodos de forma más sencilla)
- Permiten ir creando otros métodos con un nivel de complejidad mayor.

b) **Métodos inline o código embebido.**

Código antes de refactorizar:



Código después de refactorizar.

```
int rondaGratis(){
    return (numerodecanas > 5)?2:1;
}
```

Si lo que se pretende es escribir métodos y código que muestren la intención del código, muchas veces, ocultar la lógica del programa factorizando el código pude ser mala idea. En el ejemplo anterior, se observa que el código queda más simple y fácil de entender eliminando la función, puesto que se reduce su complejidad y es más legible.

De hecho, no debería utilizarse este tipo de formas de programar en Java cuando hay métodos que sobra cargan métodos de clases base.

Otro ejemplo de código embebido, que habría que evitar es el siguiente:

```
double preciobase =pedido.preciobase();
    return (preciobase > 100);
```

Sería mejor haber utilizado este código:

```
    return (pedido.preciobase(); > 100);
```

En este caso, se utiliza una variable temporal solamente para realizar la comparación. Si la variable solo se utiliza para realizar dicha comparativa, una forma más elegante y que evita problemas sería emplear la comparación directa a la hora de realizar el return.

Toma nota:
Evita utilizar variables temporales para almacenar resultados intermedios de las operaciones.

Para evitar utilizar variables temporales para almacenar resultados intermedios, por ejemplo en este código:

```
double preciobase =cantidad * preciounitario;
   if (preciobase > 1000)
       return preciobase * 0.90;
   else
       return preciobase * 0.95;
```

Es mejor utilizar este otro código:

```
   if (preciobase > 1000)
       return preciobase 0.90;
   else
       return preciobase  0.95;
…..
     double preciobase () {
     return cantidad *preciounitario;
     }
```

En el ejemplo anterior, desea realizarse un descuento al cliente. Si la cuenta supera una determinada cantidad, pagará un 90% (descuento de un 10%), en caso contrario, el descuento solamente será de un 5%.

Según lo visto anteriormente, so observa que se utiliza una variable temporal para calcular el precio base de una cuenta. Muchas veces, en vez de utilizar una variable temporal, es mejor utilizar un nuevo método para determinar el precio base. Este método podrá ser empleado por otros métodos de la clase, con lo cual su utilidad se incrementa.

c) Variables autoexplicativas

Los programadores se obcecan en escribir complejas líneas de código cuando se tienen recursos para hacer el código más legible. Por ejemplo, es fácil encontrar sentencias de código como la siguiente:

```
   if ((idioma.toUpperCase().index0f ("RUS")> -1)&&
   (idioma.toUpperCase().index0f ("ALE")> -1) &&
   (nivelingles >0)){
       //mensaje
   }
```

Utilizando variables autoexplicativas correctamente definidas (obsérvese que se ha definido como tipo final), el código resultaría mucho más legible y sencillo. Véase el código anterior corregido:

```
    Final boolean esEsp = idioma.toUpperCase().index0f ("esp")> -1);
      Final boolean esAleman= idioma.toUpperCase().index0f ("ALE")> -1);
    Final boolean ingles= nivelingles >0;
      if(esEsp && esAleman  && ingles){
        //mensaje
    }
```

e) **Mal uso de variables temporales**

Imagínese que se tiene el siguiente código:

```
double tmp = pi*radio*radio;
   System.out.println(tmp);
   tmp = 2*pi*radio;
   System.out.println (tmp);
```

En realidad, nuestro código debería haber tenido este aspecto:

```
Final double area = pi*radio*radio;
   System.out.println(area);
  Final double perímetro = 2*pi*radio;
   System.out.println (perimetro);
```

¿Por qué se hace esto? Porque una variable temporal, generalmente, se utiliza para ser instanciada o establecida solamente una vez por lo que, en caso de que tenga que cambiarse su valor varias veces durante la ejecución del método, hay que plantearse crear una segunda variable temporal. En el caso anterior, se ve claramente que habría que utilizar dos variables temporales en vez de una.

f) **Cambiar algoritmos**

Imaginemos que tenemos el siguiente código

```
String BuscaAnimal(String[] animales){
   for (int i = 0; i c animales.length; i++) {
      if (animales[i].equals ("Perro")){
      return "Perro";
      }
      if (animales[i].equals ("Tortuga")){
      return "Tortuga";
      }
      if (animales[i].equals ("Loro")){
      return "Loro";
      }
   }
   return "No encontrado";
}
```

El resultado de una refactorización que mejore la legibilidad y eficiencia del código anterior sería la siguiente:

```
String BuscaAnimal2(String[] animales){
    for (String animale: animales) {
        if (animale.equals("Perro")) {
            return "Perro";
        }
        if (animale.equals("Tortuga")) {
            return "Tortuga";
        }
        if (animale.equals("Loro")) {
            return "Loro";
        }
    }
    return "No encontrado";
}
```

Toma nota

Refactorización y pruebas

Todos los desarrolladores en su trabajo diario crean nuevo código a la vez que van haciendo refactorización de dicho código y van creando casos de prueba con el mismo. El objetivo es claro: mejorar eficiencia, consistencia, claridad y mantenimiento futuro. Si un proyecto va a sufrir una gran refactorización, lo mejor es crear casos de prueba con Junit u otra herramienta para asegurarse de que dicha refactorización no altera el comportamiento de código anterior.