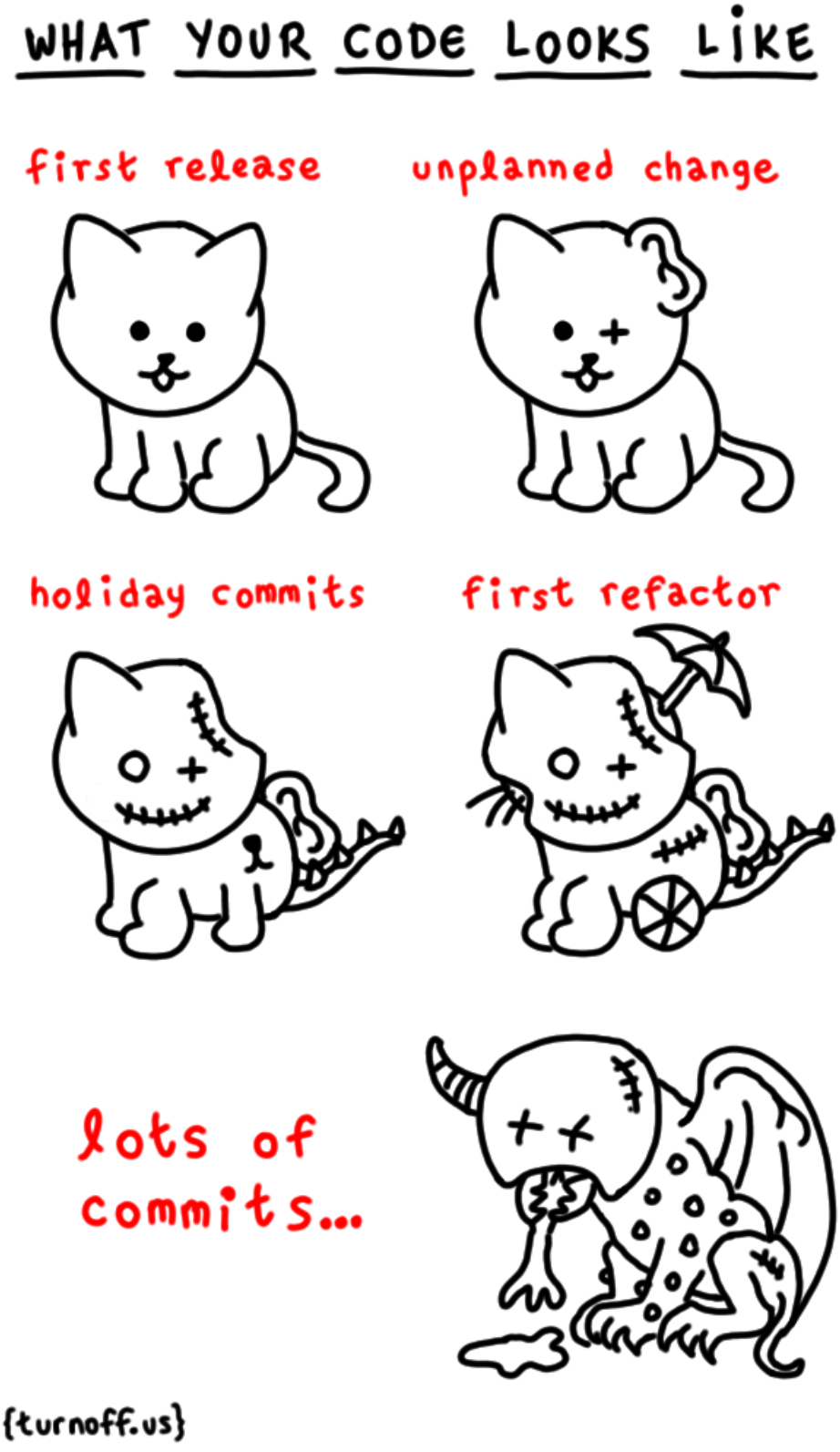
Refactorización



La refactorización es una técnica de la ingeniería de 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; 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 adición de alguna funcionalidad al software.

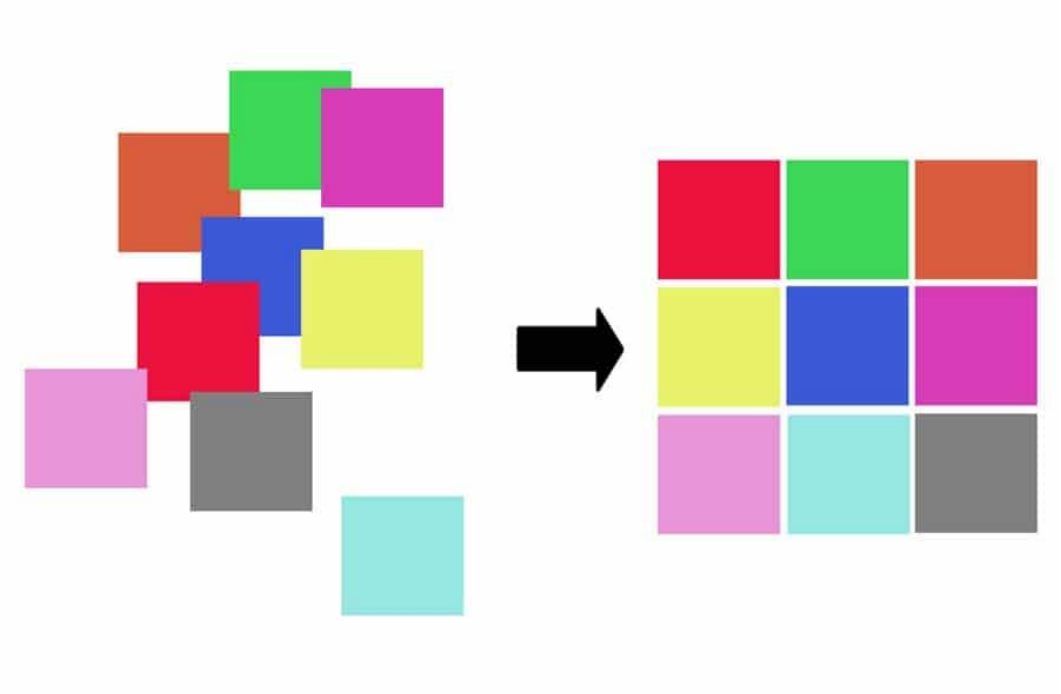
https://es.wikibooks.org/wiki/Refactorizaci%C3%B3n/Definici%C3%B3n

La refactorización tiene como objetivo limpiar el código para que sea más fácil de entender y de 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.

¿Qué hace la refactorización?

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



¿Cuándo refactorizar? Malos olores (bad smells)

La refactorización se debe ir haciendo mientras se va realizando el desarrollo de la aplicación, a los síntomas que indican la necesidad de refactorizar, se les suele llamar bad smells (malos olores).

Estos síntomas son los siguientes:

- Código duplicado (Duplicated code). 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.
- 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 y descomponer el método en otros más pequeños. En la programación orientada a objetos cuanto más corto es un método más fácil es 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 más pequeñas, de forma que cada una trate con un conjunto pequeño de responsabilidades bien delimitadas.
- Lista 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 unos con otros y suelen ir juntos siempre.

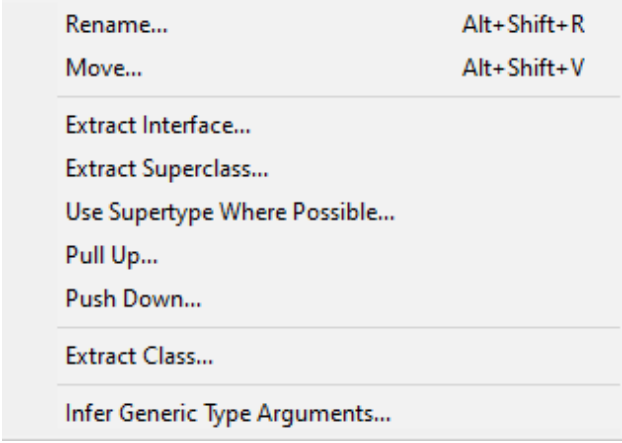
- Cambio divergente (Divergent change): una clase es frecuentemente modificada por diversos motivos, los cuales no suelen estar relacionados entre sí, a lo mejor conviene eliminar la clase. Este síntoma es el opuesto del siguiente.
- Cirugía a tiro pistola (Shotgun surgery): este síntoma se presenta 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.

El proceso de refactorización presenta algunas ventajas, entre las que se encuentran 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, principalmente las bases de datos, y los cambios de interfaces. Un cambio de base 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.

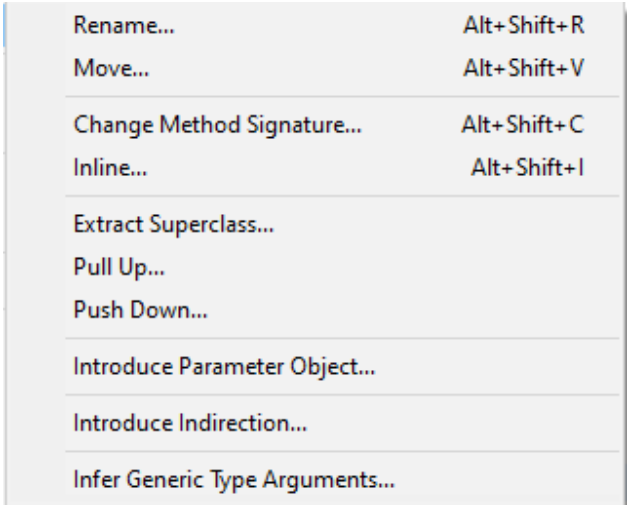
Refactorización en Eclipse

Eclipse tiene diversos métodos de refactorizar. Para refactorizar, elegiremos la opción “Refactor del menú contextual. Dependiendo de dónde invoquemos a la refactorización tendremos un menú contextual u otro con sus diferentes opciones de refactorización:

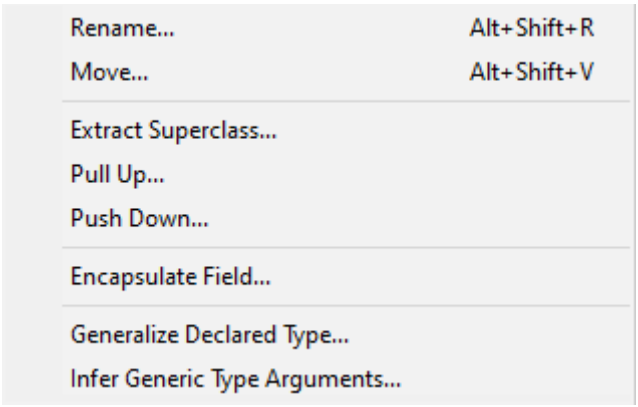
- Clase:



- Método:



- Atributo:



Métodos 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 podremos ver el antes y el después de refactorizar. A los métodos de refactorización también se les llama patrones de refactorización o catálogos de refactorización.

Para refactorizar se selecciona 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. A continuación, se muestran algunos de los métodos más comunes:

- Rename. Es una de las opciones más utilizadas. Cambia el nombre de 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.
- Extract Constant. Convierte un número o cadena literal en una constante. Al hacer la refactorización se mostrará dónde se van a producir los cambios, y se puede visualizar el estado antes de refactorizar y después de refactorizar. Tras la refactorización, todos los usos del literal se sustituyen por esa constante. El objetivo es modificar el valor del literal en un único lugar.
- Extract Local Variable. Asignar una expresión a variable local. Tras la refactorización, cualquier referencia a la expresión en el ámbito local se sustituye por la variable. La misma expresión en otro método no se modifica. En la figura se muestran los cambios que se producirán al extraer una variable local, se muestran dónde se realizarán los cambios, y el detalle antes y después (Figura 4.63).

- Convert Local Variable to Field. Convierte una variable local en un atributo privado de la clase. Tras la refactorización, todos los usos de la variable local se sustituyen por ese atributo.
- Extracl Method. Nos permite seleccionar un bloque de código y convertirlo en un método. El bloque de código no debe dejar llaves abiertas. Eclipse ajustará automáticamente los parámetros y el retorno de la función. Esto es muy útil para utilizarlo cuando se crean métodos muy largos, que se podrán dividir en varios métodos. También es muy útil extraer un método cuando se tiene un grupo de instrucciones que se repiten varias veces. Al extraer el método hay que indicar el modificador de acceso: público, protegido, privado, o sin modificador.
- Change Method Signature. Este método permite cambiar la firma de un método. Es decir, el nombre del método y los parámetros que tiene. De forma automática se actualizarán todas las dependencias y llamadas al método dentro del proyecto. En la ventana para cambiar la firma de un método, se indicará el nuevo nombre del método, el tipo de dato que devuelve, los nuevos parámetros, se pueden editar los parámetros y cambiarlos, o también asignar un valor por defecto. Si al refactorizar cambiamos el tipo de dato de retorno del método, aparecerán errores de compilación, por lo que debemos modificarlo manualmente.
- 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, dentro de la clase FicheroAleatorioVenlana nos encontramos con la declaración de una variable RandomAccessFile file.

  Por ejemplo, dentro de la clase FicheroAleatorioVenlana nos encontramos con esta declaración:

```
15          RandomAccessFile file;
16          file = new RandomAccessFile(fichero, "rw");
17          file.close();
```

Posicionamos el cursor en la referencia al método o variable, en este caso la variable fichero. Seleccionamos la opción "Inline" y el resultado es:

```
13          RandomAccessFile file;
14          file = new RandomAccessFile(new File("AleatorioDep.dat"), "rw");
```

Para verlo más claro, puedes usar la opción Preview.

- Member Type to Top Level. Convierte una clase anidada en una clase de nivel superior con su propio archivo de java. Si la clase es estática, la refactorización es inmediata. Si no es estática nos pide un nombre para declarar el nombre de la clase que mantendrá la referencia con la clase inicial.
- Extract Interface. Este método de refactorización nos permite escoger los métodos de una clase para crear una 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. Por ejemplo, se define la interface Animal, y los métodos comer y respirar porque todos los animales comen y respiran. Sin embargo, cada animal va a comer y respirar de forma diferente, por eso en cada animal se deben desarrollar esos métodos.

```
interface Animal {

    void comer();
    int respirar();
}
```

```
public class Perro implements Animal{

    @Override
    public void comer() {
        // Se define como come un perro
    }

    @Override
    public int respirar() {
        // Se define como respira un perro
        return 0;
    }

    public String ladrar() {
        // Este método es exclusivo  de los perros
        return "";
    }

}
```

- Extract Superclass. Este método permite extraer una superclase. Si la clase ya utilizaba una superclase, la recién creada pasará a ser su superclase. Se pueden seleccionar los métodos y atributos que forrnarán parte de la superclase. En la superclase, los métodos están actualmente allí, así que, si hay referencias a campos de clase original, habrá fallos de compilación.
- Convert Anonymous Class to Nested. Este método de refactorización permite convertir una clase anónima a una clase anidada de la clase que la contiene. Una clase anónima es una clase sin nombre de la que solo se crea un único objeto, de esta clase no se pueden definir constructores. Se utilizan con frecuencia cuando se crean ventanas, para gestionar los eventos de los distintos componentes de la interface gráfica. Puiedes leer más sobre clases anidadas aquí: https://javadesdecero.es/poo/clases-anidadas/