

Módulo profesional entornos de
desarrollo

UD 7 –
Optimización y
refactorización

contenido

- Refactorización
 - concepto
 - ventajas y limitaciones
- ¿Qué refactorizar? Categorías de cosas que huelen mal→
 - inflados
 - abusadores de la orientación a objetos
 - obstaculizadores de cambios
 - prescindibles
- Ciclo de refactorización
- Herramientas

refactorización

Refactorización, el camino hacia la calidad

Refactorización

¿Qué es más **importante**, dedicar el menor tiempo posible en el desarrollo de una aplicación informática (y ahorrarnos todo el coste posible de un programador), o bien desarrollar la misma aplicación con un código fuente mucho más preparado para futuros cambios (habiendo dedicado más esfuerzo)?

Refactorización

Al desarrollar una aplicación hay que tener muy presentes algunos aspectos del código de programación que se irá implementando.

Hay pequeños aspectos que permitirán que este código sea considerado más óptimo o que facilitarán su mantenimiento.

Refactorización

En múltiples ocasiones durante el transcurso de un proyecto de grandes proporciones o de larga duración es frecuente encontrarse con que necesitamos reevaluar o modificar código creado anteriormente o ponerse a trabajar con un proyecto otra persona para lo cual antes deberá comprender el código.

A pesar de los comentarios o la documentación del proyecto en cuestión, la refactorización ayuda a tener un código que es más sencillo de comprender, más compacto, más limpio y por supuesto más fácil de modificar.

Ejemplo de refactorización

Por ejemplo, si hay un valor que se utilizará varias veces a lo largo de un determinado programa, es mejor utilizar una constante que contenga este valor, de esta manera, si el valor cambia sólo deberá modificar la definición de la constante y no será necesario irlo buscando por todo el código desarrollado ni recordar cuántas veces se ha utilizado.

Ejemplo de refactorización

A continuación, se muestra un ejemplo muy sencillo para entender a qué se hace referencia cuando se habla de refactorizar el código fuente:

```
Class CalculaCosto {  
    public static double costoTrabajadores(double numTrabajadores) {  
        return 1200 * numTrabajadores;  
    }  
}
```

En el código anterior se muestra un ejemplo de cómo sería la codificación de una clase que tiene como función el cálculo de los costes laborales totales de una empresa. Se puede ver que el **coste por trabajador** no se encuentra en ninguna variable ni a ninguna constante, sino que el método

Ejemplo de refactorización

El método `costoTrabajadores` devuelve el valor que ha recibido por parámetro (`numTrabajadores`) por un número que considera el salario bruto por trabajador (en este caso supuesto 1200 euros).

Probablemente, este importe se utilizará en más clases a lo largo del código de programación desarrollado o, como mínimo, más veces dentro de la misma clase.

Como quedaría el código una vez aplicada la refactorización? Se puede ver a continuación:

```
Class CalculaCostos {  
    final double SALARIO_BRUTO = 1.200;  
  
    public static double costoTrabajadores (double numTrabajadores) {  
        Return SALARIO_BRUTO * numTrabajadores;  
    }  
}
```

Actividad guiada

1. Con lo que has aprendido has
· ahora en programación...
· ¿Cambiarías algo? ¿Qué?

```
import java.util.*;
class Programa
{
    static double radeon;

    public static void main(String[] args)
    {
        //Programa que calcula el area de un triangulo
        Scanner sc = new Scanner(System.in);

        System.out.println("Bienvenido");
        System.out.println("Ingrese base del triángulo: ");
        double amd= sc.nextInt();

        System.out.println("Ingrese altura del triángulo: ");
        double nvidia = sc.nextInt();

        GTX(amd, nvidia);
        System.out.println("El área de el triángulo es: " + radeon);
    }

    static void GTX(double x, double y)
    {
        radeon = (x * y) / 2;
    }
}
```

Refactorización

Antes de refactorizar

```
import java.util.*;
class Programa
{
    static double radeon;

    public static void main(String[] args)
    {
        //Programa que calcula el area de un triangulo
        Scanner sc = new Scanner(System.in);

        System.out.println("Bienvenido");
        System.out.println("Ingrese base del triángulo: ");
        double amd= sc.nextInt();

        System.out.println("Ingrese altura del triángulo: ");
        double nvidia = sc.nextInt();

        GTX(amd, nvidia);
        System.out.println("El área de el triángulo es: " + radeon);
    }

    static void GTX(double x, double y)
    {
        radeon = (x * y) / 2;
    }
}
```

Después de refactorizar

```
import java.util.Scanner;
class ProgramaRefactorizado
{
    public static void main(String[] args)
    {
        //Programa que calcula el area de un triangulo
        Scanner sc = new Scanner(System.in);

        System.out.println("Bienvenido");
        System.out.println("Ingrese base del triángulo: ");
        double base= sc.nextDouble();

        System.out.println("Ingrese altura del triángulo: ");
        double altura = sc.nextDouble();

        double area = calcularArea(base, altura);
        System.out.println("El área de el triángulo es: " + area);
    }

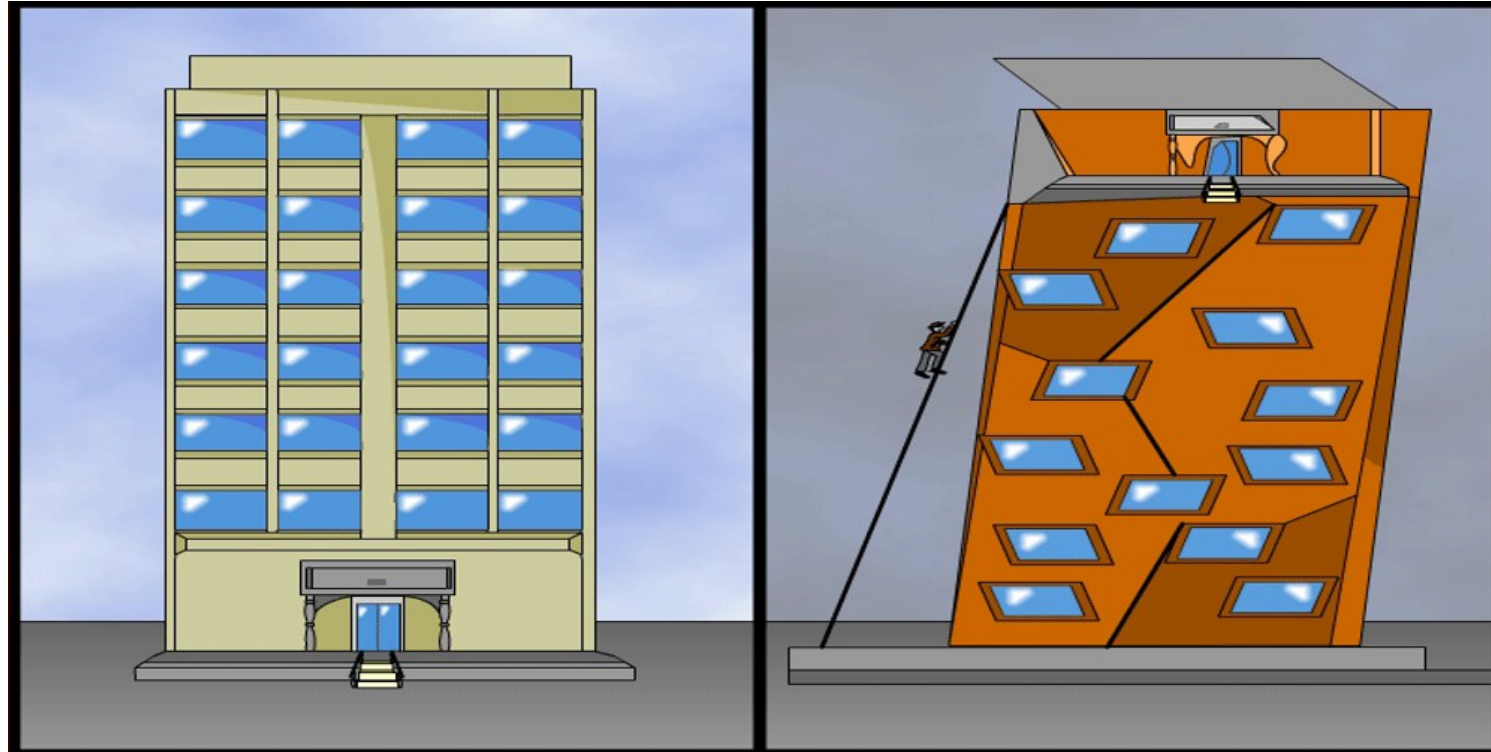
    static double calcularArea(double b, double a)
    {
        return (b * a) / 2;
    }
}
```

Refactorización

Como se puede observar las refactorizaciones son cambios que se hacen al código para que sea más legible, flexible, extendible y modificable.

Refactorización

¿Si tu software fuera un edificio, se parecería mas a uno de la izquierda o de la derecha?



Refactorización

Reestructurar el código fuente alterando su estructura sin cambiar su comportamiento

El término *refacción* hace referencia a los cambios efectuados en el código de programación desarrollado, sin implicar ningún cambio en los resultados de su ejecución. Es decir, se transforma el código fuente manteniendo intacto su comportamiento, aplicando los cambios sólo en la forma de programar o en la estructura del código fuente, buscando su optimización.

Refactorizar no es lo mismo que optimizar

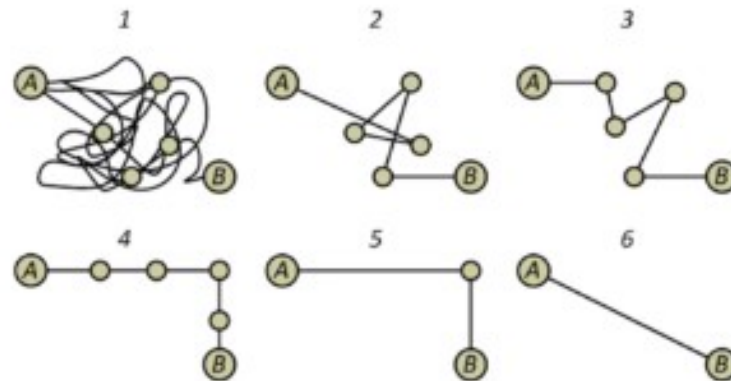
Refactorizar vs. optimizar: son cosas distintas porque tienen objetivos distintos.

En un desarrollo lo aconsejable es refactorizar primero y optimizar sólo cuando necesite (y no siempre necesito pensar en performance).

Cuando refactorizo el código queda más claro y mantenible, cuando optimizo no necesariamente pasa eso.

Refactorización

El "**Refactoring**" es el nombre que se le da al conjunto independiente de pequeñas técnicas que podemos aplicar para *mejorar el código existente*".



El código ya existe y está funcionando antes de mejorarlo

Ventajas y limitaciones de la refactorización

La utilización de la refactorización puede aportar algunas ventajas a los desarrolladores de software, pero hay que tener en cuenta que tiene limitaciones que hay que conocer antes de tomar la decisión de utilizarla.

Ventajas de la refactorización

Hay muchas ventajas en la utilización de la refactorización, aunque también hay inconvenientes y algunas limitaciones.

¿Por qué los programadores dedican tiempo a la refactorización del código fuente?

Ventajas y limitaciones de la refactorización

Una de las respuestas a esta pregunta es el aumento de la calidad del código fuente. Un código fuente sobre el que se han utilizado técnicas de refactorización se mantendrá en un estado mejor que un código fuente sobre el que no se hayan aplicado.

A medida que un código fuente original se ha ido modificando, ampliando o manteniendo, habrá sufrido modificaciones en la estructura básica sobre la que se diseñó, y es cada vez más difícil efectuar cambios evolutivos y aumenta la probabilidad de generar errores.

¿Por qué refactorizar?

Para mejorar el diseño del software

- Para hacerlo más entendible
- Para ayudar a encontrar errores (se pierde mucho menos tiempo en depurar y entender el código)
- Permite programar más rápido

La refactorización informalmente es llamada limpieza de código

¿Cuándo refactorizar?



Metáfora de los dos sombreros de Kent Beck

Cuando uno empieza a programar se pone el sombrero de “hacer código nuevo”, una vez terminado una parte del programa, lo ha compilado, lo ha probado que funciona y deja esa parte y sigue con la codificación de el programa.

Mientras sigues desarrollando te das cuenta que tienes un trozo de código que podrías reutilizar en lo que estás haciendo o simplemente ves algo que hecho de otra manera facilitaría el trabajo, en este momento te pones el sombrero de “limpiar el código”.

Ahora **no estás introduciendo nada nuevo**. Estás extrayendo operaciones en métodos, moviendo código de un lugar a otro, dejando métodos más pequeños con una nomenclatura más comprensible. Una vez que terminas, el código funciona exactamente igual que antes pero está hecho de otra manera que facilita el trabajo.

Metáfora de los dos sombreros



Añadir funcionalidad

- ⊗ No modifica el código existente
- ⊗ Añade nueva funcionalidad al sistema
- ⊗ Añade nuevos test
- ⊗ Obtener los resultados de test



Refactorizar

- ⊗ Modifica el código existente
- ⊗ No añade nuevas características
- ⊗ No añade test (pero podría modificar alguno)
 - ✗ Cambio de interfaz
- ⊗ Reestructura el código existente para eliminar redundancia

Realizar intercambios de sombreros, pero llevar puesto sólo uno cada vez

¿Cuándo refactorizar? Ejemplo

El concepto es combinar la creación de código nuevo y la mejora y refactorización de manera alternativa, siempre en plazos pequeños.

Es decir refactorizar en modo sistemático como parte de las fases de desarrollo y mantenimiento de código y no cómo una etapa planificada.

¿Qué no es la refactorización?

La refactorización no es una forma de encontrar y resolver defectos. Ninguna de estas técnicas que veamos de refactorización ayudarán a resolver los defectos actuales de una aplicación.

1. El “*Refactoring*” **NO** es resolver Defectos.



¿Qué no es la refactorización?

La refactorización no es para mejorar el desempeño de una aplicación. Para mejorar el desempeño de una aplicación se usan técnicas y herramientas diferentes a las de la refactorización. La refactorización si mejora la velocidad de algo y ese algo no es la aplicación sino nos hace más veloces a nosotros, a los programadores. Ya que al tener un código bien diseñado seremos más efectivos al entenderlo y al modificarlo

2. El “*Refactoring*” **NO** es para mejorar el desempeño de la aplicación.



La refactorización no es agregar nueva funcionalidad. Son dos pasos separados. Muchas veces debemos hacer refactorización para facilitar agregar nueva funcionalidad pero primero refactorizamos para reordenar el código sin afectar la funcionalidad y después agregamos la nueva funcionalidad

3. El “Refactoring” **NO** es para agregar nueva funcionalidad.

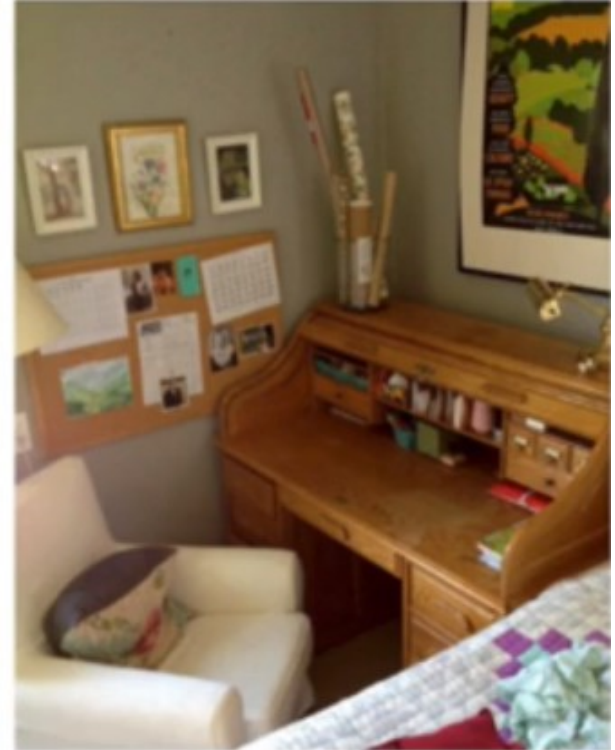
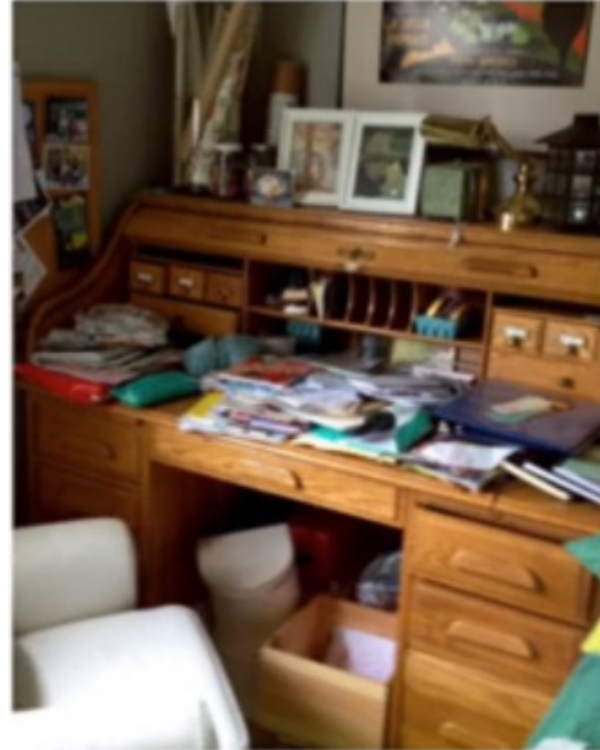


¿Cuándo refactorizar?

¿**Cuándo** se debe refactorizar?

- Antes de escribir un nuevo código
 - Para hacerlo más entendible
 - Para simplificar la implementación de las nuevas funcionalidades
- Antes de corregir un error
- Cuando se revisa el código y algo huele mal

El tiempo necesario para refactorizar hay que incluirlo como una actividad normal en el desarrollo del software



¿Cuándo no debes refactorizar?

No refactoricemos nunca cuando no sabemos exactamente lo que estamos haciendo



¿Cuándo no debes refactorizar?

Hay veces que no debes refactorizar.

Hay ocasiones en que el código existente es un desastre que aunque queramos refactorizarlo, sería más fácil comenzar desde el principio.

Esta decisión no es fácil de tomar.

Un signo claro de la necesidad de volver a escribir es cuando el código actual simplemente no funciona.

Puedes descubrir esto al intentar probarlo y descubrir que el código está tan lleno de errores que no puedes estabilizarlo. Recuerda, el código tiene que funcionar correctamente antes de refactorizar.

¿Cuándo no debes refactorizar?

Debes evitar la refactorización cuando estés cerca de una fecha límite de entrega. En ese punto, la ganancia en productividad de la refactorización aparecería después de la fecha límite y, por lo tanto, sería demasiado tarde.

Sin embargo, aparte de estar cerca de una fecha límite, no debes posponer la refactorización porque no tienes tiempo. La experiencia con varios proyectos ha demostrado que la refactorización resulta en una mayor productividad. No tener suficiente tiempo por lo general es una señal de que necesitamos hacer algo de refactorización.

Problemas con la refactorización

- Convencer a directivos
 - Directivos motivados por la calidad -> mejora la calidad del proceso
 - Otros -> Aumento de velocidad de desarrollo
- Trabajar sin casos de prueba definidos
 - Para refactorizar el código debe funcionar correctamente sino reescribirlo

Refactorización y Buenos programadores

Los **buenos programadores** pasan algún tiempo limpiando su código.

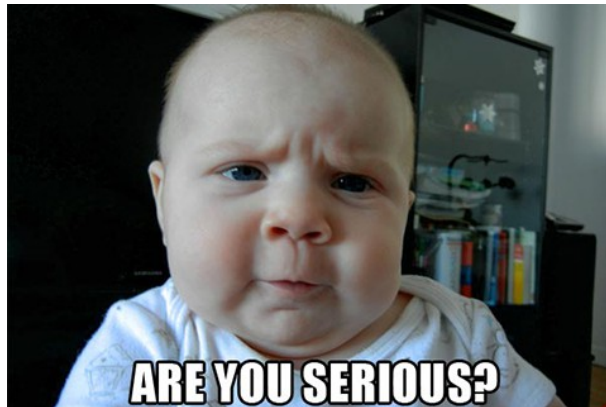
Lo hacen porque aprendieron que el código limpio es más fácil de cambiar que el código complejo y desordenado, y los buenos programadores saben que rara vez escriben código limpio la primera vez.

Buenos programadores

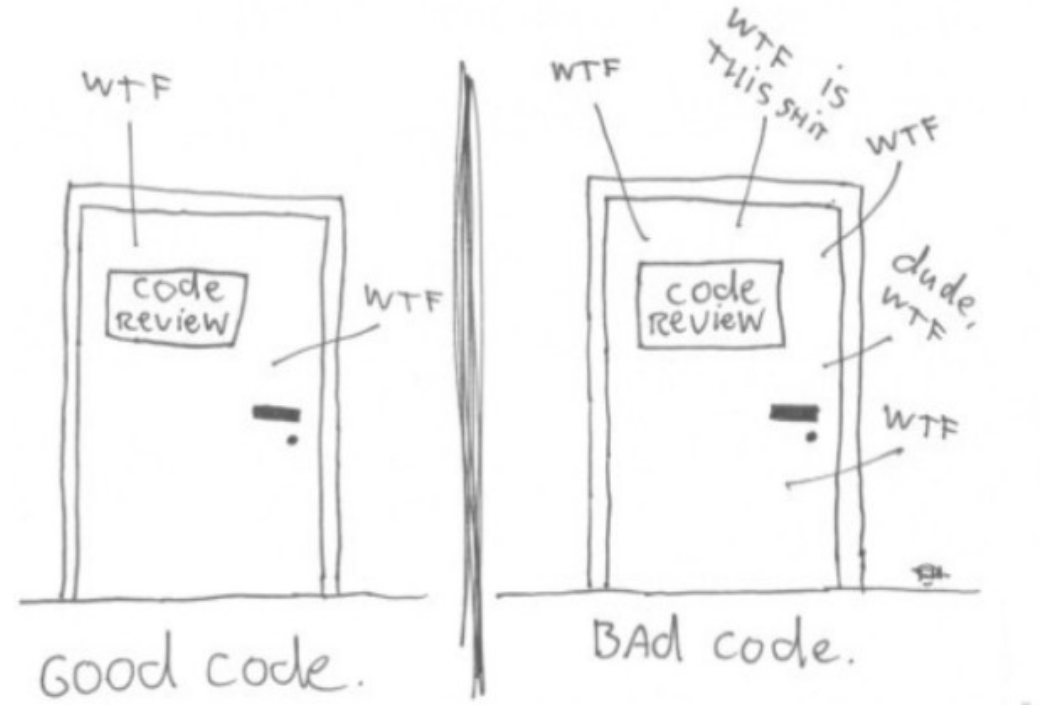
**¿QUÉ IMPLICA EL
CÓDIGO MALO?**

Buenos programadores

La cantidad de **WTFs/minute** emitidos cuando se examina tu código



The ONLY valid measurement
of code quality: WTFs/minute



Nota: WTF -**What the fuck**- ,vulgarismo inglés, frecuentemente usado en chats y foros para mostrar estupefacción, asombro o desentendimiento (en ocasiones, desacuerdo), y cuya traducción al castellano podría ser: «¿Pero qué rayos?», «¿Qué pasó?», «¿Qué diablos?» o «¿Qué demonios?» (o una expresión similar, algo soez, de todos conocida en España), «¿Pero qué me estás contando?», "¿Es en serio?!".

Buenos programadores

**¿QUÉ ENTENDEMOS
POR CÓDIGO LIMPIO?**

Cuando algo en el código no huele bien

Hay un término muy utilizado en la práctica de refactorización que se conoce como **“code smell”**.

Este término es utilizado cuando algo en el código **no huele bien**, hay algo que no nos gusta, que ese patrón en el código ya lo hemos visto y nos ha causado problemas. Por ejemplo, el ver código duplicado.

No importa que el código de el resultado esperado, eso no huele bien va a causar problemas a quien tenga la mala fortuna de modificarlo.

CUANDO ALGO EN EL CÓDIGO “NO HUELE BIEN”

Un “*Code Smell*” no es en sí un Defecto, es un problema potencial.



Patrones de refactorización más usuales

Cosas que huelen mal (bad smells)



Cosas que huelen mal

¿Qué? ¿Cómo puede el código "oler"?!!!!

Sí, el código puede oler y si huele mal hay que cambiarlo.

Lo que huele es señal de que hay algo en el código que no anda bien que podemos mejorar.

Veamos algunas técnicas...

Cosas que huelen mal

Veamos estos malos olores en detalle y las técnicas que se pueden utilizar para combatirlos...



INFLADOS

Inflados son código, métodos y clases que han aumentado a proporciones tan gigantescas que son difíciles de trabajar.

Cosas que huelen mal

Categoría Inflados

Inflados son código, métodos y clases que han aumentado a proporciones tan gigantescas que son difíciles de trabajar.

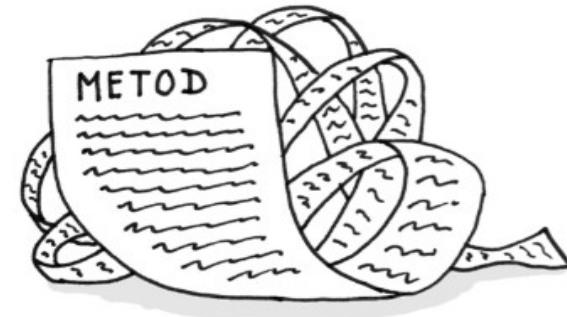
Por lo general, estos olores no aparecen de inmediato, sino que se acumulan con el tiempo a medida que el programa evoluciona (y especialmente cuando nadie hace un esfuerzo para erradicarlos).

- métodos largos
- clases grandes
- abusar de atributos primitivos
- lista larga de parámetros
- grupos de datos

¿Qué podemos refactorizar? inflados

Métodos largos

Un método contiene demasiadas líneas de código. En general, cualquier método de más de diez líneas debe hacer que comience a hacer preguntas.



Razones del problema

Siempre se está agregando algo a un método, pero nunca se saca nada. Ya que es más fácil escribir código que leerlo, este "olor" permanece inadvertido hasta que el método se convierte en una bestia fea y sobredimensionada.

¿Qué podemos refactorizar? inflados

Tratamiento

Como regla general, si siente la necesidad de agregar algo dentro de un método, debe tomar este código y ponerlo en un método nuevo. Y si el método tiene un nombre descriptivo, nadie tendrá que mirar el código del método para ver qué hace.



¿Qué podemos refactorizar? inflados

Ganancia

Entre todos los elementos de código orientado a objetos, las clases con métodos cortos viven más tiempo. Cuanto más largo sea un método o función, más difícil será comprenderlo y mantenerlo.

Además, los métodos largos son el escondite perfecto para el código duplicado no deseado.

¿Qué podemos refactorizar? inflados

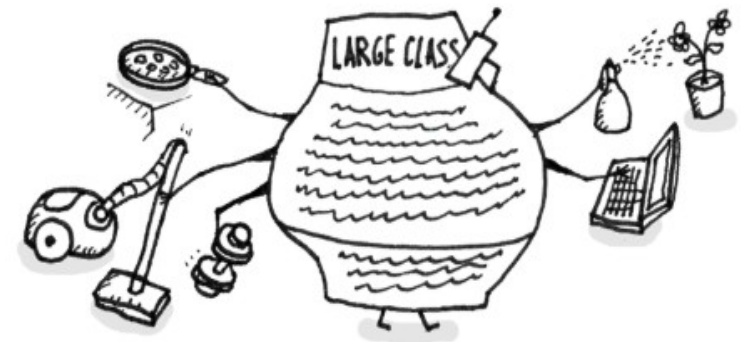
Clases grandes

Una clase contiene muchos campos / métodos / líneas de código.

Razones del problema

Las clases suelen comenzar poco a poco. Pero con el tiempo, se hinchan a medida que el programa crece.

Como ocurre también con los métodos largos, a los programadores generalmente les resulta mentalmente menos exigente colocar una nueva característica en una clase existente que crear una nueva clase para la característica.



¿Qué podemos refactorizar? inflados



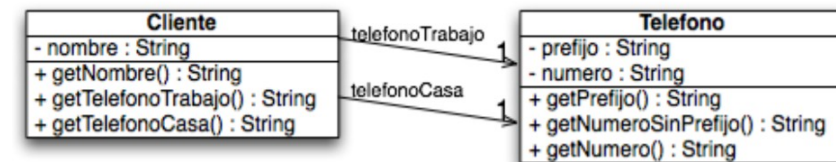
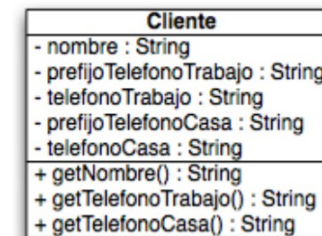
Tratamiento

Una clase con muchas responsabilidades conoce información de muchos objetos, y se ve rápidamente impactada ante cualquier cambio. Cualquier agregado o corrección de estas clases suele representar un dolor de cabeza. Cuando una clase hace demasiadas “cosas” piensa en dividirla.

Hay varias formas, una de ellas es mediante la técnica de

Extraer a clase ayuda si parte del comportamiento de la clase grande se puede dividir en un componente separado.

Otras formas son extraer subclase o extraer interface.

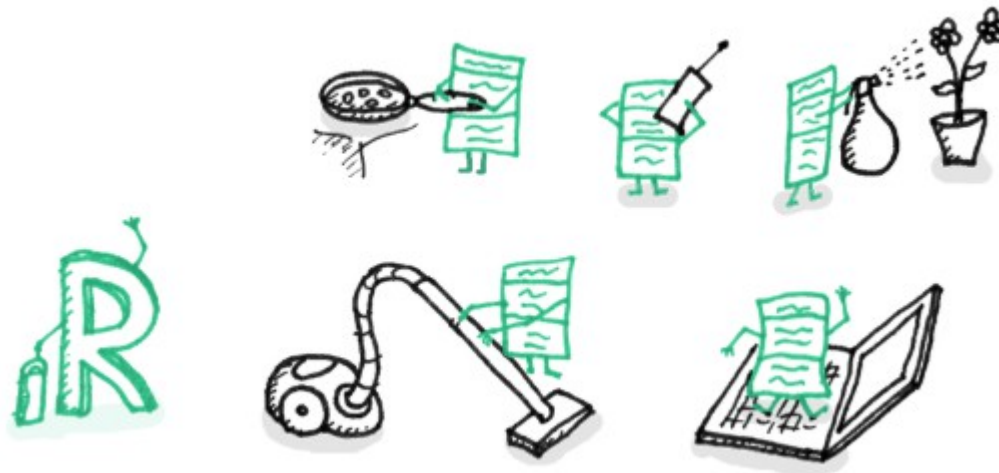


¿Qué podemos refactorizar? inflados

Ganancia

La refactorización de estas clases evita que los desarrolladores necesiten recordar una gran cantidad de atributos para una clase.

En muchos casos, dividir clases grandes en partes evita la duplicación de código y funcionalidad.



¿Qué podemos refactorizar? inflados

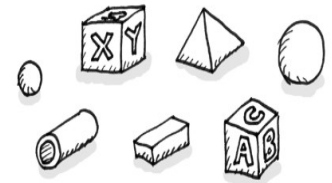
Obsesión primitiva

Signos y síntomas

Uso excesivo de tipos primitivos en lugar de objetos pequeños para tareas simples (como moneda, rangos, cadenas especiales para números de teléfono, etc.)

Razones del problema

Como la mayoría de los otros olores, las obsesiones primitivas nacen en momentos de debilidad. “¡Solo un campo para almacenar algunos datos!” Dijo el programador. Crear un campo primitivo es mucho más fácil que crear una clase completamente nueva, ¿verdad? Y así se hizo. Luego se necesitó otro campo y se agregó de la misma manera. He aquí que la clase se vuelve enorme y difícil de manejar.

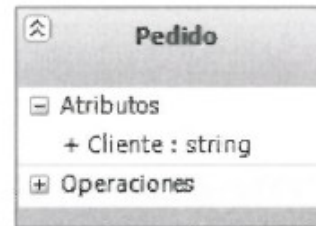


¿Qué podemos refactorizar? inflados



Tratamiento

Si tiene una gran variedad de campos primitivos, puede ser posible agrupar lógicamente algunos de ellos en su propia clase. Aún mejor, mueva el comportamiento asociado con estos datos (métodos) a la clase también. Para esta tarea, intenta reemplazar valor de datos con un objeto.



Refactorizamos



¿Qué podemos refactorizar? inflados

Ganancia

- El código se vuelve más flexible gracias al uso de objetos en lugar de primitivos.
- Mejor comprensibilidad y organización del código. Las operaciones en datos particulares están en el mismo lugar, en lugar de estar dispersas.
- Más fácil encontrar el código duplicado.



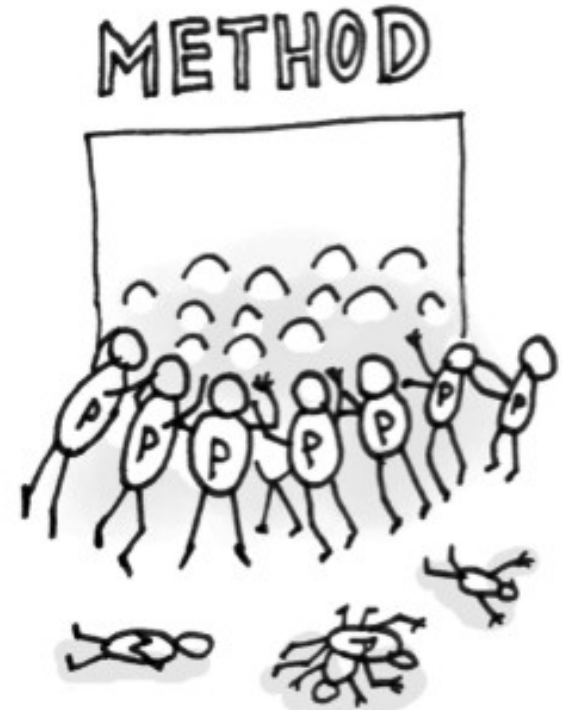
¿Qué podemos refactorizar? inflados

Lista grande de parámetros

Signos y síntomas

Más de tres o cuatro parámetros para un método.

En lugar de una larga lista de parámetros, un método puede usar los datos de su propio objeto. Si el objeto actual no contiene todos los datos necesarios, se puede pasar otro objeto (que obtendrá los datos necesarios) como parámetro del método.



¿Qué podemos refactorizar? inflados

Tratamiento

Si algunos de los parámetros son solo resultados de llamadas de método de otro objeto, reemplaza el parámetro con la llamada al método.

```
int precioBase= cantidad* precioUnitario;
```

```
double descuento= this.getDescuento();
```

```
double impuesto= this.getImpuesto();
```

```
double precioFinal= precioConDescuento(precioBase, descuento, impuesto);
```

Reemplazar por

```
int precioBase= cantidad* precioUnitario;
```

```
double precioFinal= precioConDescuento (precioBase);
```



¿Qué podemos refactorizar? inflados

Ganancia

- Más legible, código más corto.
- La refactorización puede revelar un código duplicado previamente desapercibido.

¿Qué podemos refactorizar? inflados

Grupos de datos

Signos y síntomas

A veces, diferentes partes del código contienen grupos idénticos de variables que se arrastran juntos (en atributos de clases, en parámetros, etc). Estos grupos deben convertirse en sus propias clases.



- x,y está representando un objeto Punto
- calle, altura, piso, departamento, etc. forman parte de un objeto Dirección

¿Qué podemos refactorizar? inflados

Razones para el problema

A menudo, estos grupos de datos se deben a una estructura de programa deficiente o "programación copiar y pegar".

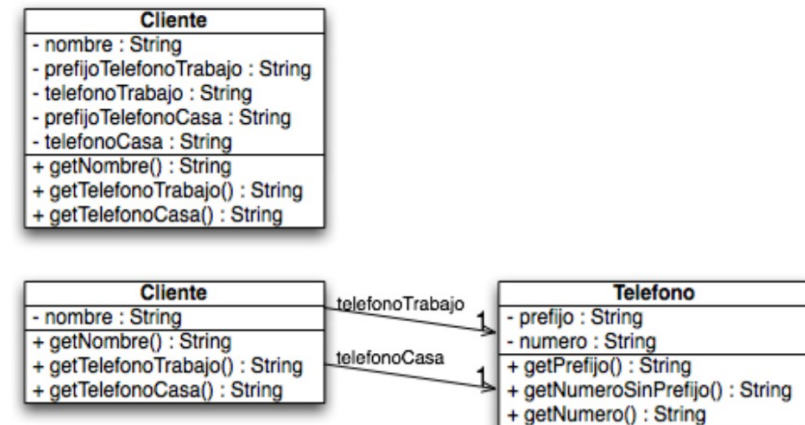
Si desea ver si algunos datos forman o no un grupo de datos, simplemente elimine uno de los datos y vea si los otros datos aún tienen sentido. Si este no es el caso, es una buena señal de que este grupo de variables debe combinarse en un objeto.

¿Qué podemos refactorizar? inflados

Tratamiento

Si los datos que se repiten comprenden los atributos de una clase, use crea una clase para mover los campos a su propia clase.

Si se pasan los mismos grupos de datos en los parámetros de los métodos, use usa un objeto como parámetro para establecerlos como una clase. Si algunos de los datos se pasan a otros métodos, piense en pasar el objeto de datos completo al método en lugar de solo campos individuales.



¿Qué podemos refactorizar? inflados

Ganancia

- Mejora la comprensión y organización del código. Las operaciones en datos particulares ahora se reúnen en un solo lugar, en lugar de aleatoriamente en todo el código.
- Reduce el tamaño del código.



Cuando ignorar

Pasar un objeto completo en los parámetros de un método, en lugar de pasar solo sus valores (tipos primitivos), puede crear una dependencia indeseable entre las dos clases.



ABUSADORES DE LA OO

Todos estos olores son una aplicación incompleta o incorrecta de los principios de programación orientados a objetos.

¿Qué podemos refactorizar?

Abusadores de la orientación a objetos

Categoría abusadores de la orientación a objetos

Todos estos olores son una aplicación incompleta o incorrecta de los principios de programación orientados a objetos.

- sentencias switch
- campos temporales
- legado rechazado
- clases alternativas con diferentes interfaces



¿Qué podemos refactorizar?

Abusadores de la orientación a objetos

Sentencias switch **Signos y síntomas**



Tienes un operador de switch complejo o una secuencia de instrucciones if.

Razones para el problema

El uso relativamente raro de switch y case es una de las características del código orientado a objetos. A menudo, el código con un switch puede dispersar en diferentes lugares del programa. Cuando se agrega una nueva condición, debe encontrar todo el switch y modificarlo.

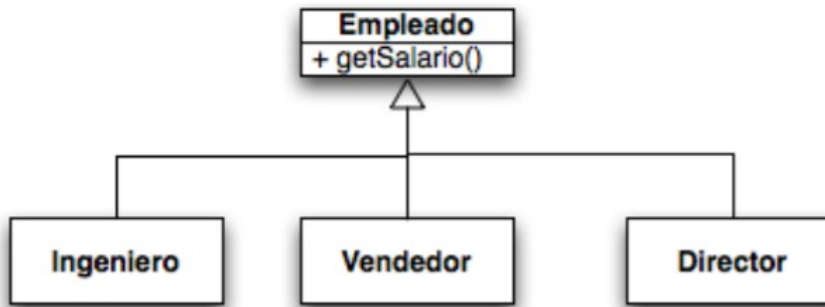
Como regla general, cuando veas un switch debería pensar en el **polimorfismo**.

¿Qué podemos refactorizar?

Abusadores de la orientación a objetos

Tratamiento

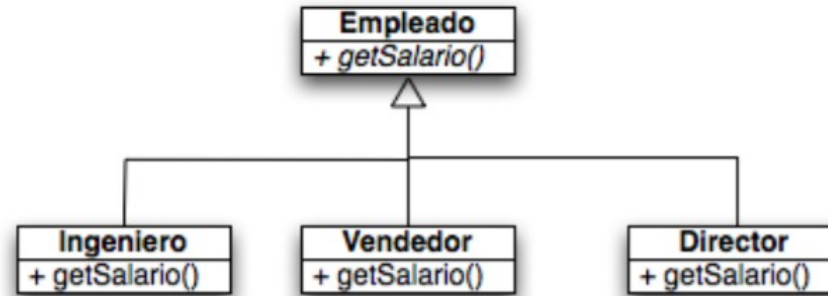
Un posible tratamiento es **Reemplazar condicionales por polimorfismo**



```
double getSalario() {
    switch(tipoEmpleado()) {
        case INGENIERO: return salarioBase + productividad; break;
        case VENDEDOR: return salarioBase + ventas*comision; break;
        case DIRECTOR: return salarioBase + bonificacion+ dietas; break;
        default : throw new RuntimeException("Tipo de empleado incorrecto");
    }
}
```

¿Qué podemos refactorizar?

Abusadores de la orientación a objetos



```
class Empleado {
    abstract double getSalario(); ...
}
```

```
class Ingeniero {
    @Override double getSalario() { return salarioBase + productividad; }
    ...}
class Vendedor {
    @Override double getSalario() { return salarioBase + ventas*comision; }
    ...}
class Director {
    @Override double getSalario() {return salarioBase+bonificacion+dietas;}
    ...}
```

¿Qué podemos refactorizar?

Abusadores de la orientación a objetos

Ganancia

Mejora la organización del código



¿Qué podemos refactorizar?

Abusadores de la orientación a objetos



Campo temporal

Los campos temporales obtienen sus valores (y, por lo tanto, son necesarios para los objetos) solo bajo ciertas circunstancias. Fuera de estas circunstancias, están vacíos. Es decir, algunos objetos tienen atributos que solo usan en ciertas circunstancias.

Ejemplo:

Una clase Factura que tiene un variable privada double total, otra totalConIVA y otra totalSinIVA,

Es decir que “almacena” valores que podría calcular.

¿Qué podemos refactorizar?

Abusadores de la orientación a objetos

Razones del problema

A menudo, los campos temporales se crean para su uso en un algoritmo que requiere una gran cantidad de entradas. Entonces, en lugar de crear parámetros en el método, el programador decide crear campos para estos datos en la clase. Estos campos se usan solo en el algoritmo y no se utilizan el resto del tiempo.

Este tipo de código es difícil de entender. Espera ver valores en campos de objetos, pero por alguna razón casi siempre están vacíos.



¿Qué podemos refactorizar?

Abusadores de la orientación a objetos

Tratamiento

Una posible solución es que los campos temporales y todos los códigos que operan en ellos se pueden colocar en una clase separada.



```
class Order {  
    //...  
    public double price() {  
        double primaryBasePrice;  
        double secondaryBasePrice;  
        double tertiaryBasePrice;  
        // Long computation.  
        //...  
    }  
}
```

```
class Order {  
    //...  
    public double price() {  
        return new PriceCalculator(this).compute(  
        }  
}  
  
class PriceCalculator {  
    private double primaryBasePrice;  
    private double secondaryBasePrice;  
    private double tertiaryBasePrice;  
  
    public PriceCalculator(Order order) {  
        // copy relevant information from order o  
        //...  
    }  
  
    public double compute() {  
        // Long computation.  
        //...  
    }  
}
```

Ganancia

Esto mejora la claridad y organización del código

ejemplo

Tenemos una variable temporal que se asigna una vez, reemplaza la variable temporal con la expresión.

```
double basePrice = anOrder.basePrice();  
return (basePrice > 1000)
```



```
return (anOrder.basePrice() > 1000)
```

Actividad


3. Refactoriza el siguiente código

```
double basePrice = _quantity * _itemPrice;  
if (basePrice > 1000)  
    return basePrice * 0.95;  
else  
    return basePrice * 0.98;
```

Actividad

Se está utilizando una variable temporal para contener el resultado de una expresión. Extrae la expresión en un método. Reemplaza todas las referencias a la variable temporal con la expresión. El nuevo método puede ser utilizado en otros métodos.

```
double basePrice = _quantity * _itemPrice;
if (basePrice > 1000)
    return basePrice * 0.95;
else
    return basePrice * 0.98;
```



```
...
if (basePrice() > 1000)
    return basePrice() * 0.95;
else
    return basePrice() * 0.98;

...
double basePrice() {
    return _quantity * _itemPrice;
}
```

¿Qué podemos refactorizar?

Abusadores de la orientación a objetos

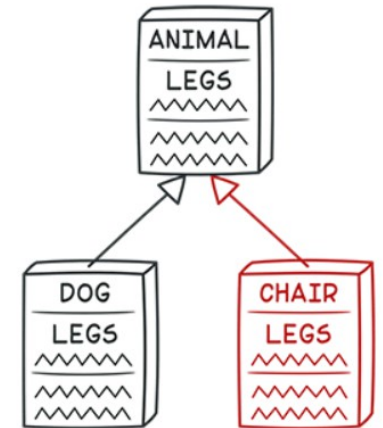
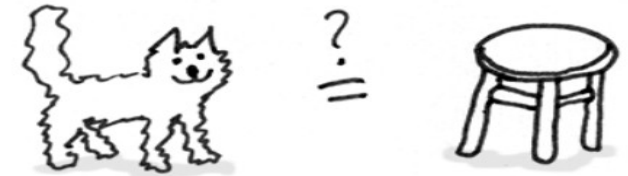
Legado rechazado

Signos y síntomas

Subclases que usan sólo 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.

Razones para el problema

Alguien estaba motivado para crear una herencia entre clases solo por el deseo de reutilizar el código en una superclase. Pero la superclase y la subclase son completamente diferentes.

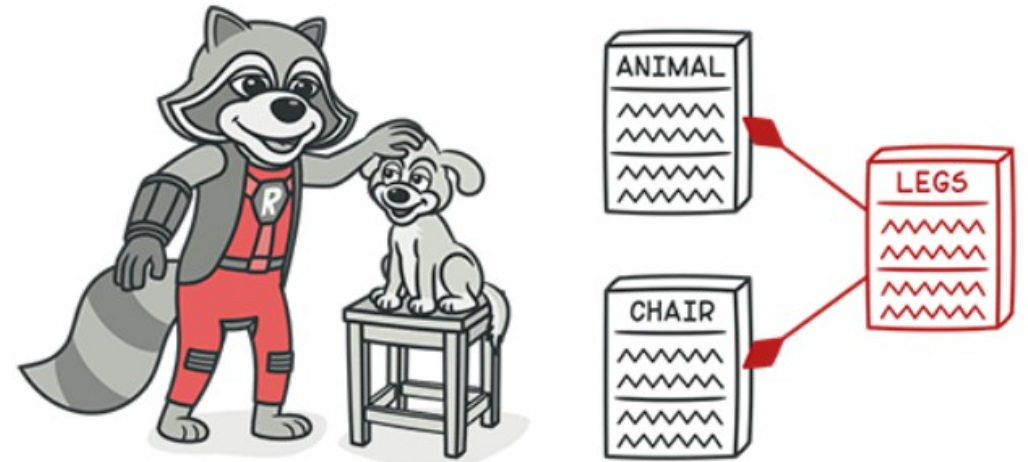


¿Qué podemos refactorizar?

Abusadores de la orientación a objetos

Tratamiento 1

Si la herencia no tiene sentido y la subclase realmente no tiene nada en común con la superclase, elimina la herencia y usa **reemplazar herencia por delegación (composición)**.



¿Qué podemos refactorizar?

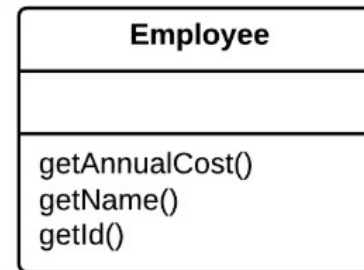
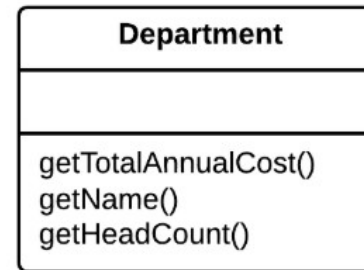
Abusadores de la orientación a objetos

Tratamiento 2

Si la herencia es apropiada, elimina los campos y métodos innecesarios en la subclase. Extrae todos los campos y métodos necesarios para la subclase de la clase principal, colóquelos en una nueva subclase y configure ambas clases para heredar de ella (**extraer a superclase**).

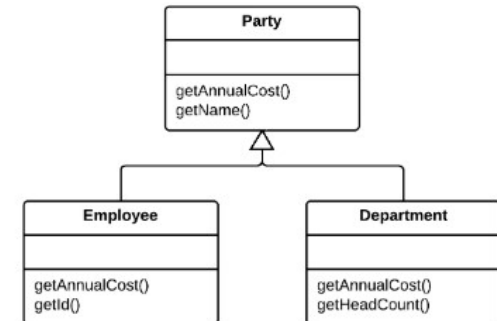
Problema

Tienes dos clases con campos y métodos comunes.



Solución

Cree una superclase compartida para ellos y muévelos todos los campos y métodos idénticos.



¿Qué podemos refactorizar?

Abusadores de la orientación a objetos

Ganancia

Mejora la claridad y la organización del código. Ya no tendrás que preguntarse por qué la clase `Dog` se hereda de la clase `Chair` (a pesar de que ambos tienen 4 patas).

actividad

4. A la vista del siguiente código, identifica y aplica las refactorizaciones que considere más convenientes.

```
public class Persona {
    String numeroDeTelefono;

    public Persona(String numeroDeTelefono) {
        super();
        this.numeroDeTelefono = numeroDeTelefono;
    }

    public String getNumeroDeTelefono() {
        return numeroDeTelefono;
    }

    public void setNumeroDeTelefono(String numeroDeTelefono) {
        this.numeroDeTelefono = numeroDeTelefono;
    }
}

public class Profesor extends Persona {

    String str;
    int edad;
    String numeroDeTelefono;
    List<Prestamo> prestamos;

    public Profesor(String numeroDeTelefono) {
        super(numeroDeTelefono);
    }

    public void printInformacionPersonal() {
        System.out.println("Nombre: " + str);
        System.out.println("Edad: " + edad);
        System.out.println("Teléfono: " + numeroDeTelefono);
    }

    public void printTodaLaInformacion() {
        System.out.println("Nombre: " + str);
        System.out.println("Edad: " + edad);
        System.out.println("Teléfono: " + this.numeroDeTelefono);
        for (Prestamo p: prestamos) {
            System.out.println(p);
        }
    }
}
```

¿Qué podemos refactorizar?

Abusadores de la orientación a objetos

Clases alternativas con diferentes interfaces

Signos y síntomas

Dos clases realizan funciones idénticas pero tienen nombres de métodos diferentes.

Razones para el problema

El programador que creó una de las clases probablemente no sabía que ya existía una clase funcionalmente equivalente.



¿Qué podemos refactorizar?

Abusadores de la orientación a objetos

Tratamiento

Intenta cambiar la interfaz de clases en términos de un denominador común.

- Cambia el nombre de método para hacerlos idénticos en todas las clases alternativas.
- Usa **mover método**, **agregar parámetro** y **parametrizar método** para que la cabecera y la implementación de los métodos sean las mismas.
- Si solo una parte de la funcionalidad de las clases está duplicada, intente usar **extraer a superclase**. En este caso, las clases existentes se convertirán en subclases.

¿Qué podemos refactorizar?

Abusadores de la orientación a objetos

Ganancia

- Se deshace del código duplicado innecesario, lo que hace que el código resultante sea menos voluminoso.
- El código se vuelve más legible y comprensible (ya no tiene que adivinar el motivo de la creación de una segunda clase que realiza exactamente las mismas funciones que la primera).





OBSTACULIZADO RES DE CAMBIOS

Estos olores significan que si necesita cambiar algo en un lugar en su código, también tiene que hacer muchos cambios en otros lugares.

¿Qué podemos refactorizar?

Obstaculizadores de cambios

Categoría obstaculizadores de cambios

Estos olores significan que si necesita cambiar algo en un lugar en su código, también tiene que hacer muchos cambios en otros lugares. El desarrollo del programa se vuelve mucho más complicado y costoso como resultado.

- cambio divergente
- cirugía de escopeta
- jerarquías paralelas



¿Qué podemos refactorizar?

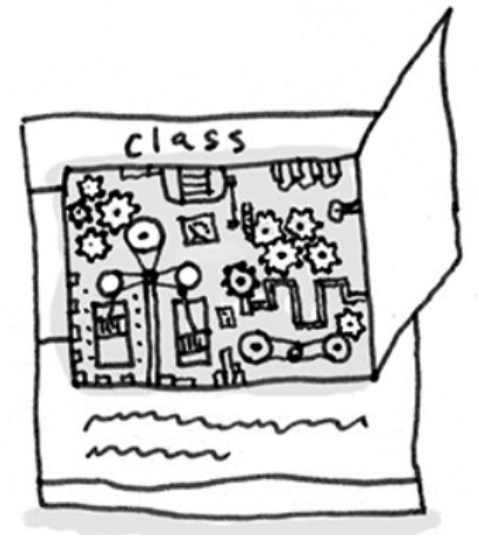
Obstaculizadores de cambios

Cambio divergente

El **cambio divergente** se parece a la **cirugía con escopeta**, pero en realidad es el olor opuesto. El cambio divergente es cuando se realizan muchos cambios en una sola clase. La cirugía de escopeta se refiere a cuando se realiza un solo cambio a varias clases simultáneamente.

Signos y síntomas

Te encuentras teniendo que cambiar muchos métodos no relacionados cuando haces cambios en una clase. Por ejemplo, al agregar un nuevo tipo de producto, tienes que cambiar los métodos para encontrar, mostrar y ordenar productos.



¿Qué podemos refactorizar?

Obstaculizadores de cambios

Razones para el problema

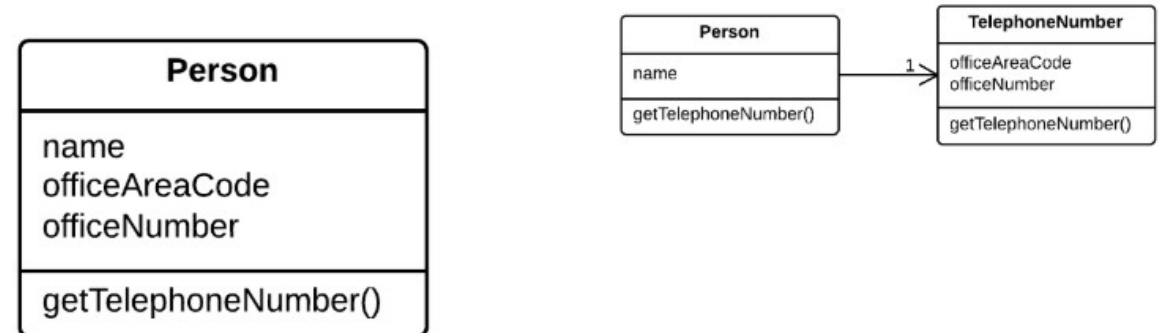
A menudo, estas modificaciones divergentes se deben a una estructura de programa deficiente o "programación copiar y pegar".

Tratamiento

Divide el comportamiento de la clase en otra clase. Si diferentes clases tienen el mismo comportamiento, es posible que desee combinar las clases a través de la herencia (**extraer a superclase** y **extraer a Subclase**).

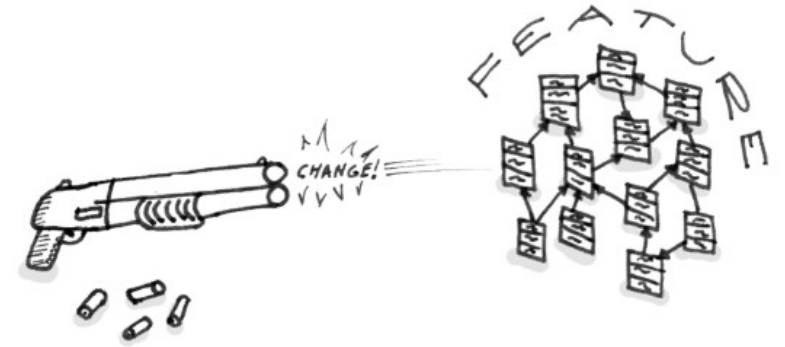
Ganancia

- Mejora la organización del código.
- Reduce la duplicación de código.
- Simplifica el soporte.



¿Qué podemos refactorizar?

Obstaculizadores de cambios



Cirugía de escopeta

Se parece al cambio divergente, pero en realidad es el olor opuesto. El cambio divergente es cuando se realizan muchos cambios en una sola clase. La cirugía de escopeta se refiere a cuando se realiza un solo cambio a varias clases simultáneamente.

Signos y síntomas

Realizar modificaciones requiere que realices muchos cambios pequeños en muchas clases diferentes.

¿Qué podemos refactorizar?

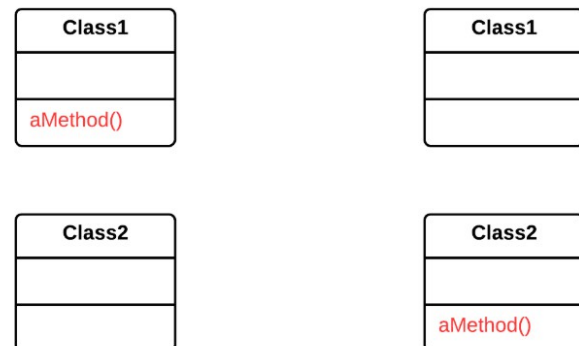
Obstaculizadores de cambios

Razones para el problema

Una sola responsabilidad se ha dividido entre un gran número de clases. Esto puede suceder después de una aplicación exagerada de cambio divergente.

Tratamiento

Usa **mover método y mover atributo** para mover los comportamientos de clase existentes a una sola clase. Si no hay una clase apropiada para esto, crea una nueva.



¿Qué podemos refactorizar?

Obstaculizadores de cambios

Ganancia

- Mejor organización
- Menos duplicación de código.
- Mantenimiento más sencillo.

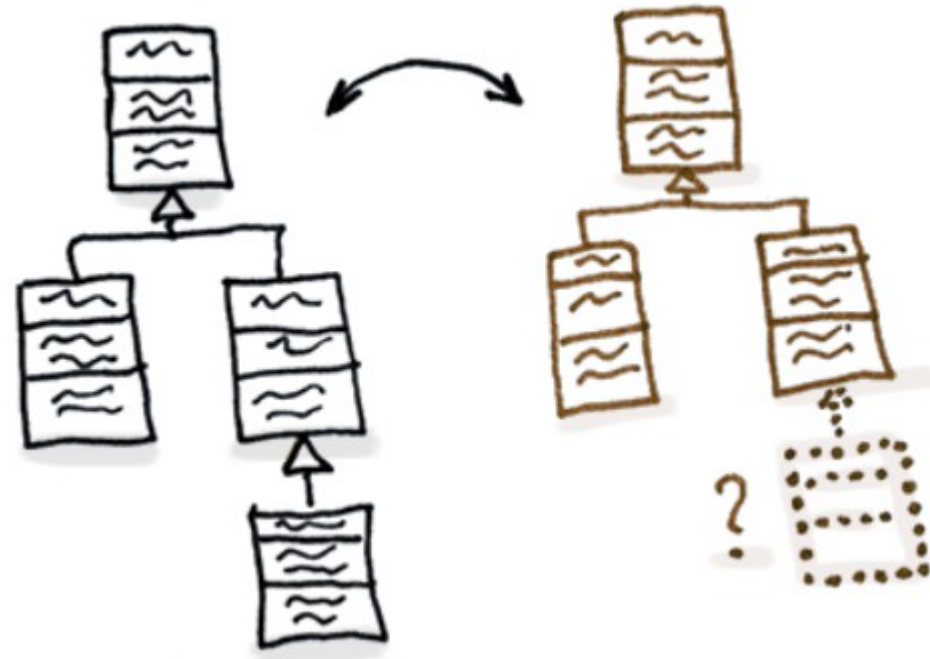


¿Qué podemos refactorizar?

Obstaculizadores de cambios

Jerarquías paralelas

Cuando creas una subclase para una clase, necesitas crear una subclase para otra clase.



¿Qué podemos refactorizar?

Obstaculizadores de cambios (poner ejemplo)

Razones para el problema

Todo iba bien mientras la jerarquía permaneciera pequeña. Pero con la incorporación de nuevas clases, hacer cambios se ha vuelto cada vez más difícil.

Tratamiento

Puede desduplicar las jerarquías de clases paralelas en dos pasos. Primero, haz que las instancias de una jerarquía se refieran a instancias de otra jerarquía. Luego, elimine la jerarquía en la clase referida, utilizando **mover método** y **mover atributo**

Ganancia

- Reduce la duplicación de código.
- Puede mejorar la organización del código.

actividad

5. A continuación tienes un fragmento de un juego, en concreto la clase que se encarga de ver el movimiento que se desea y mover las coordenadas del jugador en dicha dirección (considerando que el punto 0,0 está arriba a la izquierda).

Identifica qué refactorizaciones puedes realizar en ambas clases.

```
public class Game {  
    Player p;  
    //...  
    public void movement(String m) {  
        if (m.equalsIgnoreCase("Derecha")) {  
            p.setX(p.getX()+1);  
        }  
        if (m.equalsIgnoreCase("Izquierda")) {  
            p.setX(p.getX()-1);  
        }  
        if (m.equalsIgnoreCase("Arriba")) {  
            p.setY(p.getY()-1);  
        }  
        if (m.equalsIgnoreCase("Abajo")) {  
            p.setY(p.getY()+1);  
        }  
    }  
}
```

```
public class Player {  
    int x, y;  
    public int getX() {  
        return x;  
    }  
    public void setX(int x) {  
        this.x = x;  
    }  
    public int getY() {  
        return y;  
    }  
    public void setY(int y) {  
        this.y = y;  
    }  
}
```

```
public class Game {  
    Player p;  
    //...  
    public void movement(String m) {  
        if (m.equalsIgnoreCase("Derecha")) {  
            p.mueveDerecha();  
        }  
        if (m.equalsIgnoreCase("Izquierda")) {  
            p.mueveIzquierda();  
        }  
        if (m.equalsIgnoreCase("Arriba")) {  
            p.mueveArriba();  
        }  
        if (m.equalsIgnoreCase("Abajo")) {  
            p.mueveAbajo();  
        }  
    }  
}
```

```
public class Player {  
    int x, y;  
    public int getX() {  
        return x;  
    }  
    public void setX(int x) {  
        this.x = x;  
    }  
    public int getY() {  
        return y;  
    }  
    public void setY(int y) {  
        this.y = y;  
    }  
    public void mueveDerecha() {  
        x+=1;  
    }  
    public void mueveIzquierda() {  
        x-=1;  
    }  
    public void mueveArriba() {  
        y -= 1;  
    }  
    public void mueveAbajo() {  
        y += 1;  
    }  
}
```



PRESCINDIBLES

Un prescindible es algo inútil e innecesario cuya ausencia haría que el código sea más limpio, más eficiente y más fácil de entender.

¿Qué podemos refactorizar? prescindibles

Categoría prescindibles

Un prescindible es algo inútil e innecesario cuya ausencia haría que el código sea más limpio, más eficiente y más fácil de entender.

- comentarios
- código duplicado
- código muerto
- código especulativo
- clases de datos



¿Qué podemos refactorizar? prescindibles

Comentarios

Signos y síntomas

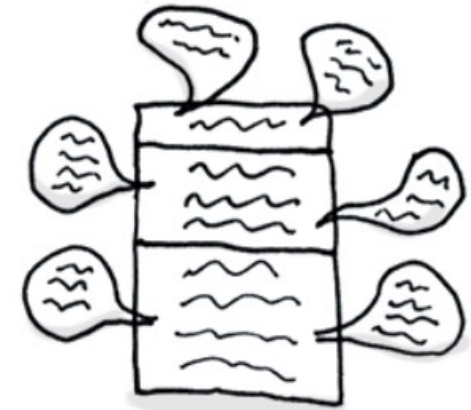
Un método está lleno de comentarios explicativos.

Razones para el problema

Los comentarios generalmente se crean con las mejores intenciones, cuando el autor se da cuenta de que su código no es intuitivo ni obvio. En tales casos, los comentarios son como un desodorante que enmascara el olor a pescado del código que podría mejorarse.

El mejor comentario es un buen nombre para un método o clase.

Si cree que un fragmento de código no se puede entender sin comentarios, intente cambiar la estructura del código de manera que los comentarios sean innecesarios.



¿Qué podemos refactorizar? prescindibles

Tratamiento

- Si se pretende que un comentario explique una expresión compleja, la expresión se debe dividir en subexpresiones comprensibles
- Si un comentario explica una sección del código, esta sección se puede convertir en un método separado. El nombre del nuevo método se puede tomar del texto del comentario, lo más probable.
- Si ya se ha extraído un método, pero aún son necesarios los comentarios para explicar qué hace el método, déle un nombre al método que se explique por sí mismo.

¿Qué podemos refactorizar? prescindibles

Ganancia

El código se vuelve más intuitivo y obvio.

Cuando ignorar

Los comentarios a veces pueden ser útiles:

- Al explicar por qué algo se está implementando de una manera particular.
- Cuando se explican algoritmos complejos (cuando se han probado todos los demás métodos para simplificar el algoritmo y se quedan cortos).

¿Qué podemos refactorizar? prescindibles

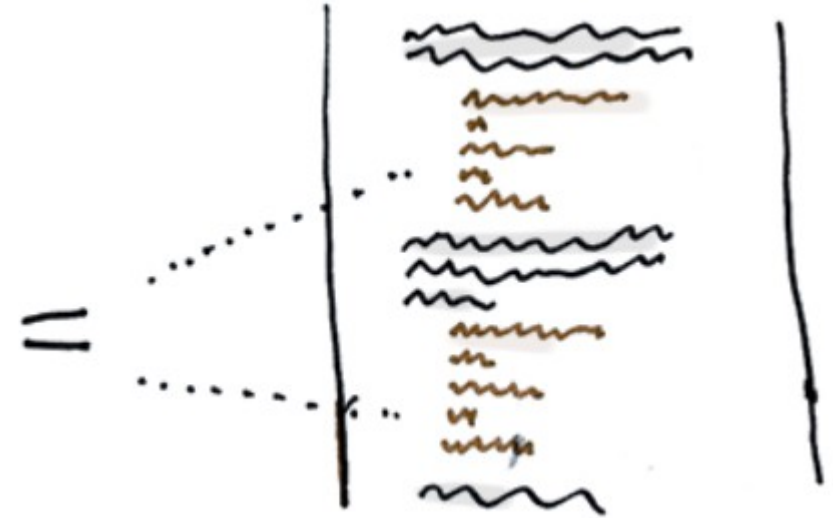
Código duplicado

Signos y síntomas

Dos fragmentos de código parecen casi idénticos.

Razones para el problema

La duplicación usualmente ocurre cuando múltiples programadores están trabajando en diferentes partes del mismo programa al mismo tiempo. Ya que están trabajando en diferentes tareas, puede que no sepan que su colega ya ha escrito un código similar que podría ser reutilizado para sus propias necesidades.



¿Qué podemos refactorizar? prescindibles

También hay una duplicación más sutil, cuando **partes específicas del código se ven diferentes pero en realidad realizan el mismo trabajo**. Este tipo de duplicación puede ser difícil de encontrar y corregir.

A veces la duplicación es útil. Cuando se apresura a cumplir con los plazos y el código existente es "casi correcto" para el trabajo, los programadores novatos no pueden resistir la tentación de copiar y pegar el código relevante. Y, en algunos casos, el programador es simplemente demasiado perezoso para refactorizar.

¿Qué podemos refactorizar? prescindibles

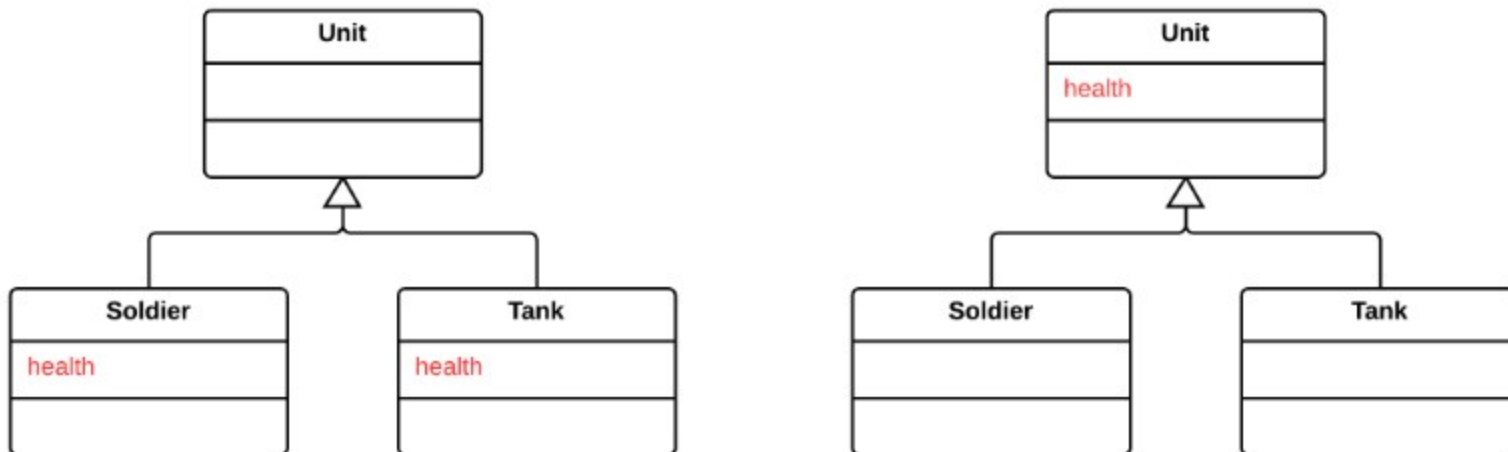
Tratamiento

Si se encuentra el mismo código en dos o más métodos en la misma clase: use un nuevo método para ello y realice llamadas para el nuevo método en ambos lugares.



¿Qué podemos refactorizar? prescindibles

- Si el mismo código se encuentra en dos subclases del mismo nivel: utiliza **extraer a método** en ambas clases, y mover a la superclase los atributos utilizados en el método



¿Qué podemos refactorizar? prescindibles

- Si hay código duplicado dentro del constructor de cada subclase: crea un constructor en la superclase y mueve el código que está repetido en las subclases. Llama al constructor de la superclase en los constructores de la subclase.

```
class Manager extends Employee {  
    public Manager(String name, String id, int grade) {  
        this.name = name;  
        this.id = id;  
        this.grade = grade;  
    }  
    // ...  
}
```

```
class Manager extends Employee {  
    public Manager(String name, String id, int grade) {  
        super(name, id);  
        this.grade = grade;  
    }  
    // ...  
}
```

¿Qué podemos refactorizar? prescindibles

- Si hay una gran cantidad de expresiones condicionales y ejecutan el mismo código (diferenciándose únicamente en sus condiciones), combine estos operadores en una sola condición utilizando **extraer a método** para colocar esa condición en el método y pon al mismo un nombre fácil de entender

```
double disabilityAmount() {  
    if (seniority < 2) {  
        return 0;  
    }  
    if (monthsDisabled > 12) {  
        return 0;  
    }  
    if (isPartTime) {  
        return 0;  
    }  
    // Compute the disability amount.  
    // ...  
}
```

```
double disabilityAmount() {  
    if (isNotEligableForDisability()) {  
        return 0;  
    }  
    // Compute the disability amount.  
    // ...  
}
```

¿Qué podemos refactorizar? prescindibles

- Si se realiza el mismo código en todas las ramas de una expresión condicional: coloque el código idéntico fuera del árbol de condiciones

```
if (isSpecialDeal()) {  
    total = price * 0.95;  
    send();  
}  
else {  
    total = price * 0.98;  
    send();  
}
```

```
if (isSpecialDeal()) {  
    total = price * 0.95;  
}  
else {  
    total = price * 0.98;  
}  
send();
```

¿Qué podemos refactorizar? prescindibles

Ganancia

- Fusionar código duplicado simplifica la estructura de su código y lo hace más corto.
- Simplificación + brevedad = código que es más fácil de simplificar y más económico de soportar.

¿Qué podemos refactorizar? prescindibles

Código muerto

Signos y síntomas

Una variable, parámetro, atributo, método o clase ya no se usa (generalmente porque está obsoleto).

Razones para el problema

Cuando los requisitos para el software han cambiado o se han realizado correcciones, nadie tuvo tiempo de limpiar el código anterior.

Dicho código también podría encontrarse en condiciones condicionales complejas, cuando una de las ramas se vuelve inaccesible (debido a un error u otras circunstancias).



¿Qué podemos refactorizar? prescindibles

Tratamiento

La forma más rápida de encontrar un código muerto es usar un buen IDE.
Eliminar código no utilizado y archivos innecesarios.



¿Qué podemos refactorizar? prescindibles

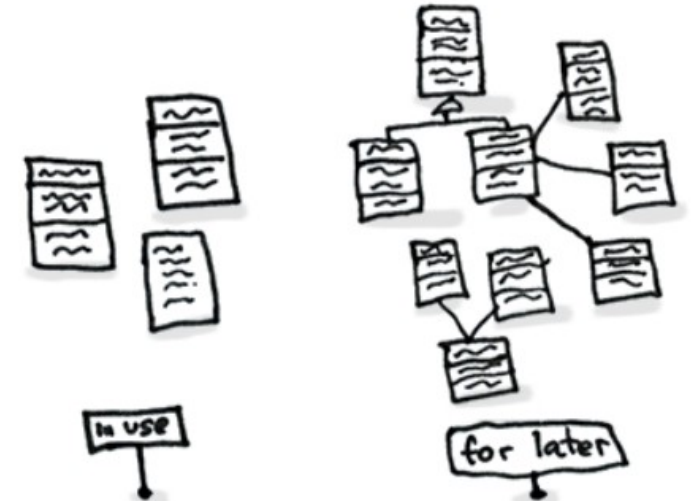
Código especulativo

Signos y síntomas

Hay una clase, método, campo o parámetro no utilizado

Razones para el problema

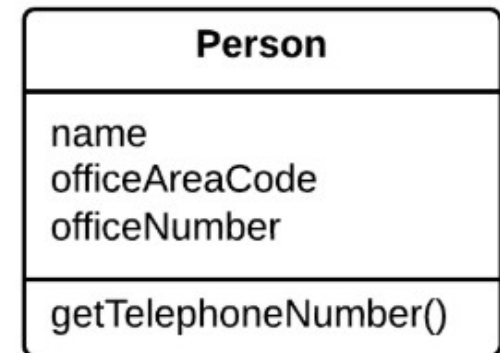
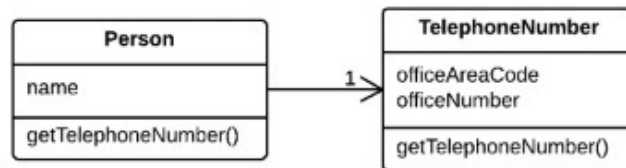
A veces, el código se crea "por si acaso" para admitir funciones futuras anticipadas que nunca se implementan. Como resultado, el código se vuelve difícil de entender y mantener.



¿Qué podemos refactorizar? prescindibles

Tratamiento

- La delegación innecesaria de funcionalidad a otra clase: una clase no hace casi nada y no es responsable de nada, y no se planifican responsabilidades adicionales para ella. Mueve todas las características de la clase a otra.



¿Qué podemos refactorizar? prescindibles

- ¿Métodos no utilizados? Cuando el cuerpo de un método es más obvio que el método en sí, reemplaza las llamadas al método con el contenido del método y elimine el método en sí.

```
class PizzaDelivery {  
    //...  
    int getRating() {  
        return moreThanFiveLateDeliveries() ? 2 : 1;  
    }  
    boolean moreThanFiveLateDeliveries() {  
        return numberOfLateDeliveries > 5;  
    }  
}
```

```
class PizzaDelivery {  
    //...  
    int getRating() {  
        return numberOfLateDeliveries > 5 ? 2 : 1;  
    }  
}
```

- Los métodos con parámetros no utilizados: se deben eliminar aquellos parámetro sin utilizar
- Los atributos no utilizados pueden ser simplemente eliminados.

¿Qué podemos refactorizar? prescindibles

Ganancia

- Código más delgado.
- Soporte más fácil

Cuando ignorar

Si está trabajando en un framework, es sumamente razonable crear una funcionalidad que no se utiliza en el marco mismo, siempre que los usuarios del marco la necesiten.

Antes de eliminar elementos, asegúrese de que no se usen en pruebas unitarias. Esto sucede si las pruebas necesitan una forma de obtener cierta información interna de una clase o realizar acciones especiales relacionadas con las pruebas.

¿Qué podemos refactorizar? prescindibles

Clases de datos

Signos y síntomas

Una clase de datos se refiere a una clase que contiene solo atributos y métodos básicos para acceder a ellos (getters y setters). Estos son simplemente contenedores de datos utilizados por otras clases. Estas clases no contienen ninguna funcionalidad adicional y no pueden operar de forma independiente con los datos que poseen.



¿Qué podemos refactorizar? prescindibles

Razones para el problema

Es algo normal cuando una clase recién creada contiene solo unos pocos atributos públicos (y tal vez incluso un puñado de getters/setters). Pero el verdadero poder de los objetos es que pueden contener comportamiento u operaciones sobre sus datos.

Tratamiento

- Si una clase contiene atributos públicos, hágalos privados/protegidos y solicite que el acceso se realice solo a través de getters y setters.
- Revise el código que utiliza la clase. En él, puede encontrar una funcionalidad que se ubicaría mejor en la propia clase de datos. Si este es el caso, utiliza **mover método** y **extraer a método** para migrar esta funcionalidad a la clase de datos.

¿Qué podemos refactorizar? prescindibles

Ganancia

- Mejora la comprensión y organización del código. Las operaciones en datos particulares ahora se recopilan en un solo lugar, en lugar de al azar en todo el código.
- Le ayuda a detectar la duplicación del código del cliente.

actividad

6. El siguiente código obviamente se ha actualizado varias veces a lo largo de los años, pero los cambios no han mejorado su estructura. ¿Cómo lo refactorizarías?

```
if (estado == TEXAS) {
    tasa = TX_RATE;
    amt = base * TX_RATE;
    calc = 2 * base (amt) + extra (amt) * 1.05;
}
más si ((estado == OHIO) || (estado == PRINCIPAL)) {
    tasa = (estado == OHIO)? OH_RATE: MN_RATE]
    amt = tasa base *;
    calc = 2 * base (amt) + extra (amt) * 1.05;
    if (estado == OHIO)
        puntos = 2;
}
más {
    tasa = 1;
    amt = base;
    calc = 2 * base (amt) + extra (amt) * 1.05;
}
```

Podríamos sugerir una reestructuración como sigue:

Si la expresión $2 * \text{base} (...) * 1.05$ aparece en otros lugares en el programa, debes probablemente hacer un método.

```
if (state == TEXAS) {
    rate = TX_RATE;
    amt = base * TX_RATE;
    calc = 2*basis(amt) + extra(amt)*1.05;
}
else if ((state == OHIO) || (state == MAINE)) {
    rate = (state == OHIO) ? OH_RATE : MN_RATE;
    amt = base * rate;
    calc = 2*basis(amt) + extra(amt)*1.05;
    if (state == OHIO)
        points = 2;
}
else {
    rate = 1;
    amt = base;
    calc = 2*basis(amt) + extra(amt)*1.05;
}
```

actividad

7. Analiza el siguiente código ¿Cómo lo refactorizarías?

```
if (esAcuerdoEspecial()) {  
    total = precio * 0.95;  
    enviar();  
}else {  
    total = precio * 0.98;  
    enviar();  
}
```


Actividad

```
if (esAcuerdoEspecial()) {  
    total = precio * 0.95;  
    enviar();  
}else {  
    total = precio * 0.98;  
    enviar();  
}
```

Refactorizamos

```
if (esAcuerdoEspecial())  
    total = precio * 0.95;  
else  
    total = precio * 0.98;  
enviar();
```

actividad

8. Analiza el siguiente código ¿Cómo lo refactorizarías?

```
if (fecha.antes (EMPIEZA_VERANO) || fecha.despues (FIN_VERANO) )  
    cargo = cantidad * _tasaInvierno + _cargoServicioInvierno;  
else cargo = cantidad * _tasaVerano;
```

ACTIVIDAD

Tenemos un complicado condicional, extrae a métodos la condición y el cuerpo

```
if (fecha.antes (EMPIEZA_VERANO) || fecha.despues (FIN_VERANO))  
    cargo = cantidad * _tasaInvierno + _cargoServicioInvierno;  
else cargo = cantidad * _tasaVerano;
```

Refactorizamos

```
if (noEsVerano(fecha))  
    cargo = cargoInvierno(cantidad);  
else charge = cargoVerano (cantidad);  
  
double cargoInvierno(int cantidad) {  
    return cantidad * _tasaInvierno + _cargoServicioInvierno;  
}  
double cargoVerano(int cantidad) {  
    return cantidad * _tasaVerano;  
}
```

actividad

9. Analiza el siguiente código ¿Cómo lo refactorizarías?

```
double cuantiaPorDiscapacidad() {  
    if (_antiguedad < 2) return 0;  
    if (_mesesDiscapacitado > 12) return 0;  
    if (_esTiempoParcial) return 0;  
    // calculamos la cantidad por discapacidad
```


ACTIVIDAD

Tenemos una secuencia de condicionales con el mismo resultado, combina en una sola expresión y extrae el método

```
double cuantiaPorDiscapacidad() {  
    if (_antiguedad < 2) return 0;  
    if (_mesesDiscapacitado > 12) return 0;  
    if (_esTiempoParcial) return 0;  
    // calculamos la cantidad por discapacidad
```

Refactorizamos

```
double cuantiaPorDiscapacidad () {  
    if (esNoElegibleParaDiscapacidad()) return 0;  
    // calculamos la cantidad por discapacidad
```



ACOPLADORES

Todos los olores en este grupo contribuyen al acoplamiento excesivo entre clases o muestran lo que sucede si el acoplamiento se reemplaza por una delegación excesiva.

Nota: el acoplamiento se refiere al grado en que una clase se basa en el conocimiento de los aspectos internos de otra clase.

¿Qué podemos refactorizar?

ACOPLADORES

Categoría acopladores

Todos los olores en este grupo contribuyen al acoplamiento excesivo entre clases o muestran lo que sucede si el acoplamiento se reemplaza por una delegación €

- Característica de la envidia
- Intimidad inapropiada
- Cadenas de mensajes
- Hombre en el medio
- Clase de biblioteca incompleta



¿Qué podemos refactorizar?

ACOPLADORES

Característica de la envidia

Signos y síntomas

Un método accede a los datos de otro objeto más que a sus propios datos.

Razones para el problema

Este olor puede ocurrir después de que los campos se muevan a una clase de datos. Si este es el caso, es posible que también desee mover las operaciones en los datos a esta clase.

Tratamiento

Como regla básica, si las cosas cambian al mismo tiempo, debe mantenerlas en el mismo lugar. Por lo general, los datos y las funciones que usan estos datos se cambian juntos (aunque son posibles excepciones).



¿Qué podemos refactorizar?

ACOPLADORES

Si un método claramente se debe mover a otro lugar, usa **mover método**.

- Si solo parte de un método accede a los datos de otro objeto, use el **extraer a método** para mover la parte en cuestión.
- Si un método usa funciones de varias otras clases, primero determine qué clase contiene la mayoría de los datos utilizados. Luego, coloque el método en esta clase junto con los otros datos. Alternativamente, usa **extraer a método** para dividir el método en varias partes que se pueden colocar en diferentes lugares en diferentes clases.

¿Qué podemos refactorizar?

ACOPLADORES

Intimidad inapropiada

Una clase usa los atributos y métodos internos de otra clase.

Razones para el problema

Esté atento a las clases que pasan demasiado tiempo juntas. Las buenas clases deben saber lo menos posible unas de otras. Dichas clases son más fáciles de mantener y reutilizar.

Tratamiento

La solución más simple es usa **mover método** y **mover atributo** para mover partes de una clase a la clase en la que se usan esas partes. Pero esto solo funciona si la primera clase realmente no necesita estas partes.



¿Qué podemos refactorizar?

ACOPLADORES

Cadenas de mensajes

En el código ves una serie de llamadas que se asemejan `$a->b()->c()->d()`

Razones para el problema

Una cadena de mensajes ocurre cuando un objeto solicita otro objeto, ese objeto solicita otro, y así sucesivamente. Estas cadenas significan que el objeto depende de la navegación a lo largo de la estructura de clases. Cualquier cambio en estas relaciones requiere modificar el objeto.



¿Qué podemos refactorizar?

ACOPLADORES

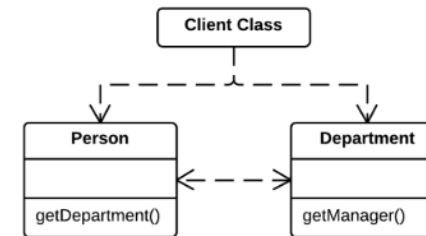
Tratamiento

Para eliminar una cadena de mensajes una forma es usar **ocultar delegado**

A veces es mejor pensar por qué se está utilizando el objeto final. Quizás tenga sentido usar el **extraer a método** para esta funcionalidad y moverlo al comienzo de la cadena, usando **mover método**.

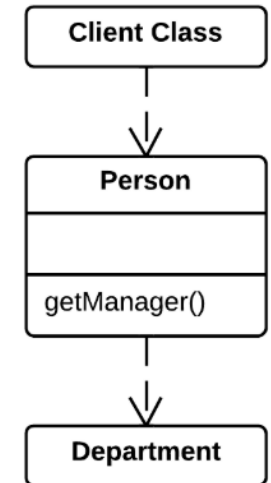
Problema

El cliente obtiene el objeto B de un campo o método del objeto A. Entonces el cliente llama a un método del objeto B.



Solución

Cree un nuevo método en la clase A que delegue la llamada al objeto B. Ahora el cliente no conoce ni depende de la clase B.



¿Qué podemos refactorizar?

ACOPLADORES

Hombre en el medio

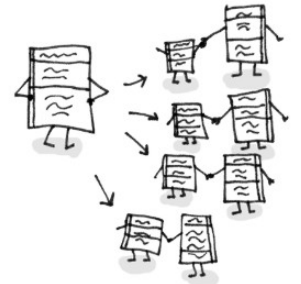
Signos y síntomas

Si una clase realiza solo una acción, delegando el trabajo a otra clase, ¿por qué existe?

Razones para el problema

Este olor puede ser el resultado de la eliminación excesivamente celosa de las cadenas de mensajes .

En otros casos, puede ser el resultado del trabajo útil de una clase que se traslada gradualmente a otras clases. La clase permanece como un caparazón vacío que no hace nada más que delegar.



¿Qué podemos refactorizar?

ACOPLADORES

Tratamiento

Si la mayoría de las clases de un método delegan a otra clase, usa **eliminar hombre en el medio**

Ganancia

Código menos voluminoso.

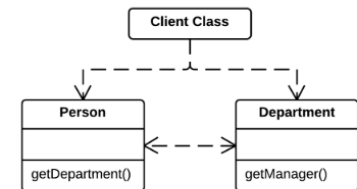
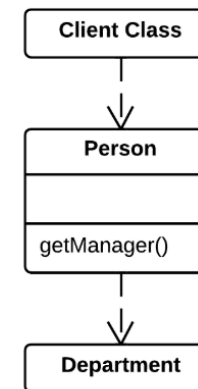


Problema

Una clase tiene demasiados métodos que simplemente delegan a otros objetos.

Solución

Elimine estos métodos y obligue al cliente a llamar a los métodos finales directamente.



¿Qué podemos refactorizar?

ACOPLADORES

Cuando ignorar

No elimine intermediarios que se hayan creado por algún motivo:

Es posible que se haya agregado un intermediario para evitar dependencias entre clases.

Algunos patrones de diseño crean intermediarios a propósito (como Proxy y Decorator).

¿Qué podemos refactorizar?

ACOPLADORES

Clase de librería incompleta

Signos y síntomas

Tarde o temprano, las librería dejan de satisfacer las necesidades de los usuarios. La única solución al problema, cambiar la biblioteca, a menudo es imposible ya que la biblioteca es de solo lectura.

Razones para el problema

El autor de la librería no ha proporcionado las características que necesita o se ha negado a implementarlas.

¿Qué podemos refactorizar?

ACOPLADORES

Tratamiento

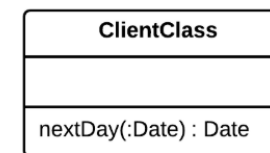
- Para introducir algunos métodos a una clase de librería, usa **introducir método extranjero**
- Para grandes cambios en una librería de clases, usa **introducir extensión local**

Cuando ignorar

Ampliar una biblioteca puede generar trabajo adicional si los cambios en la biblioteca implican cambios en el código.

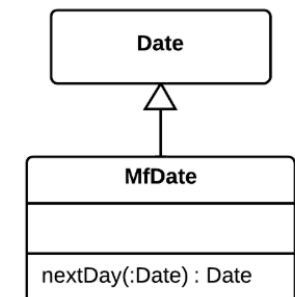
Problema

Una clase de utilidad no contiene algunos métodos que necesita. Pero no puede agregar estos métodos a la clase.



Solución

Cree una nueva clase que contenga los métodos y conviértala en hija o contenedor de la clase de utilidad.





FIN
MALOS OLORES

Ciclo de refactorización

¿ **Cómo** llevar a cabo la refactorización?

1. Antes de comenzar la refactorización, verifica que tengas un conjunto sólido de pruebas (unitarias y funcionales). Si no se cuenta con pruebas automatizadas hay que crearlas.
2. Identifica los “bad smell”
3. Elige la técnica de refactorización para resolver ese “bad smell” en particular
4. Ejecuta las pruebas antes del cambio
5. Aplica la técnica de refactorización. Utiliza siempre que sea posible herramientas especializadas. Con herramientas especializadas estas refactorizaciones se realizan automáticamente y sin riesgo.
6. Ejecuta nuevamente las pruebas. Y que son positivas

Herramientas de refactorización

No todas las herramientas que nos ofrece un IDE se basan únicamente en completar lo que escribimos, sino que también incluye patrones básicos de refactorización para su uso.

Los patrones más habituales que uno se puede encontrar en un IDE serían, renombrar, extraer a método, encapsular campo, eliminar parámetros, renombrar parámetros, etc.

Para acceder a estas funcionalidades, simplemente hay que seleccionar el código que se quiera refactorizar, y elegir la refactorización deseada y el propio IDE se encarga de realizarla. Esto **aumenta la velocidad del proceso de refactorización**

La importancia de los test

- Incluso con la herramienta, las pruebas son importantes
- Es tan importante codificar los test cómo escribir el código
 - Para preservar el comportamiento en los cambios
- Los test deben comprobarse automáticamente (self-checking)
 - Incluir los resultados esperados dentro del propio test
 - retornar “OK” si los resultados obtenido = resultados esperado
- Testear con cada compilación

vídeo

Veamos la aplicación de algunas de las técnicas de refactorización

Usando IDE en el extraer a método [https://](https://www.youtube.com/watch?v=ZeuQype6c2o&list=PL97WzFMf575j7B2qMEsDhHoMt_Cj4b)

www.youtube.com/watch?v=ZeuQype6c2o&list=PL97WzFMf575j7B2qMEsDhHoMt_Cj4b

Refactorización de extraer a clases

[https://](https://www.youtube.com/watch?v=5jtEPRm7n2w&index=12&list=PL97WzFMf575j7B2qMEsDh)

www.youtube.com/watch?v=5jtEPRm7n2w&index=12&list=PL97WzFMf575j7B2qMEsDh

Trabajando con cúmulos de datos

[https://](https://www.youtube.com/watch?v=H5GkOL908Ww&list=PL97WzFMf575j7B2qMEsDhHoMt_Cj)

www.youtube.com/watch?v=H5GkOL908Ww&list=PL97WzFMf575j7B2qMEsDhHoMt_Cj

actividad

1. Actividad guiada de refactorización Videoclub
2. Actividad ~~guiada~~ de refactorización CostoPersonal.java
3. ~~Actividad autónoma de refactorización EjercicioArrays.java~~

actividad

Analiza el siguiente código ¿Cómo lo refactorizarías?

```
public boolean max(int a, int b) {  
    if(a > b) {  
        return true;  
    } else if (a == b) {  
        return false;  
    } else {  
        return false;  
    }  
}
```


actividad

Analiza el siguiente código ¿Cómo lo refactorizarías?

Este es un código bastante difícil de manejar. ¿Por qué usar un bloque if-else si puedes escribir el método de 6 líneas de manera más concisa?

```
public boolean max(int a, int b) {  
    if(a > b) {  
        return true;  
    } else if (a == b) {  
        return false;  
    } else {  
        return false;  
    }  
}
```

```
public boolean max(int a, int b) {  
    return a > b;  
}
```

actividad

Analiza el siguiente código ¿Cómo lo refactoriza

```
class Human {  
    private String name;  
    private String age;  
    private String country;  
    private String city;  
    private String street;  
    private String house;  
    private String quarter;  
  
    public String getFullAddress() {  
        StringBuilder result = new StringBuilder();  
        return result  
            .append(country)  
            .append(", ")  
            .append(city)  
            .append(", ")  
            .append(street)  
            .append(", ")  
            .append(house)  
            .append(" ")  
            .append(quarter).toString();  
    }  
}
```

actividad

Analiza el siguiente código
¿Cómo lo refactorizarías?

Es una buena práctica
colocar la información de la
dirección y el método
asociado (comportamiento
de procesamiento de datos)
en una clase separada

Extraer a clase

```
class Human {
    private String name;
    private String age;
    private String country;
    private String city;
    private String street;
    private String house;
    private String quarter;

    public String getFullAddress() {
        StringBuilder result = new StringBuilder();
        return result
            .append(country)
            .append(", ")
            .append(city)
            .append(", ")
            .append(street)
            .append(", ")
            .append(house)
            .append(" ")
            .append(quarter).toString();
    }
}
```

```
class Human {
    private String name;
    private String age;
    private Address address;

    private String getFullAddress() {
        return address.getFullAddress();
    }
}

class Address {
    private String country;
    private String city;
    private String street;
    private String house;
    private String quarter;

    public String getFullAddress() {
        StringBuilder result = new StringBuilder();
        return result
            .append(country)
            .append(", ")
            .append(city)
            .append(", ")
            .append(street)
            .append(", ")
            .append(house)
            .append(" ")
            .append(quarter).toString();
    }
}
```

actividad

Analiza el siguiente código ¿Cómo lo refactorizarías?

```
public void employeeMethod(Employee employee) {  
    // Some actions  
    double yearlySalary = employee.getYearlySalary();  
    double awards = employee.getAwards();  
    double monthlySalary = getMonthlySalary(yearlySalary, awards);  
    // Continue processing  
}  
  
public double getMonthlySalary(double yearlySalary, double awards) {  
    return (yearlySalary + awards)/12;  
}
```

actividad

Analiza el siguiente código
¿Cómo lo refactorizarías?

EmployeeMethod tiene 2 líneas enteras dedicadas a recibir valores y almacenarlos en variables primitivas. A veces, tales construcciones pueden tomar hasta 10 líneas. Es mucho más fácil pasar el objeto en sí y usarlo para extraer los datos necesarios.

Pasar el objeto

```
public void employeeMethod(Employee employee) {  
    // Some actions  
    double yearlySalary = employee.getYearlySalary();  
    double awards = employee.getAwards();  
    double monthlySalary = getMonthlySalary(yearlySalary, awards);  
    // Continue processing  
}  
  
public double getMonthlySalary(double yearlySalary, double awards) {  
    return (yearlySalary + awards)/12;  
}
```

```
public void employeeMethod(Employee employee) {  
    // Some actions  
    double monthlySalary = getMonthlySalary(employee);  
    // Continue processing  
}  
  
public double getMonthlySalary(Employee employee) {  
    return (employee.getYearlySalary() + employee.getAwards())/12;  
}
```

actividades

Ejemplos de la web de Refactoring.Guru



actividad

Analiza el siguiente código ¿Cómo lo refactorizarías?

```
class Order {  
    // ...  
  
    public double calculateTotal() {  
        double total = 0;  
        for (Product product : getProducts()) {  
            total += product.quantity * product.price;  
        }  
  
        // Apply regional discounts.  
        switch (user.getCountry()) {  
            case "US": total *= 0.85; break;  
            case "RU": total *= 0.75; break;  
            case "CN": total *= 0.9; break;  
            // ...  
        }  
  
        return total;  
    }  
}
```

actividad

Analiza el
siguiente código
¿Cómo lo
refactorizarías?

```
class Order {
    // ...

    public double calculateTotal() {
        double total = 0;
        for (Product product : getProducts()) {
            total += product.quantity * product.price;
        }

        // Apply regional discounts.
        switch (user.getCountry()) {
            case "US": total *= 0.85; break;
            case "RU": total *= 0.75; break;
            case "CN": total *= 0.9; break;
            // ...
        }

        return total;
    }
}
```

```
class Order {
    // ...

    public double calculateTotal() {
        double total = 0;
        for (Product product : getProducts()) {
            total += product.quantity * product.price;
        }
        total = applyRegionalDiscounts(total);
        return total;
    }

    public double applyRegionalDiscounts(double total) {
        double result = total;
        switch (user.getCountry()) {
            case "US": result *= 0.85; break;
            case "RU": result *= 0.75; break;
            case "CN": result *= 0.9; break;
            // ...
        }
        return result;
    }
}
```


actividad

Analiza el siguiente código ¿Cómo lo refactorizarías?

```
void renderBanner() {  
    if ((platform.toUpperCase().indexOf("MAC") > -1) &&  
        (browser.toUpperCase().indexOf("IE") > -1) &&  
        wasInitialized() && resize > 0 )  
    {  
        // do something  
    }  
}
```

Extraer a variable

```
void renderBanner() {  
    final boolean isMacOs = platform.toUpperCase().indexOf("MAC") > -1;  
    final boolean isIE = browser.toUpperCase().indexOf("IE") > -1;  
    final boolean wasResized = resize > 0;  
  
    if (isMacOs && isIE && wasInitialized() && wasResized) {  
        // do something  
    }  
}
```

actividad

Analiza el siguiente código ¿Cómo lo refactorizarías?

```
void renderBanner() {  
    // Render banner only if we're in fullscreen mode and  
    // a change is requested either in frame or target, or  
    // experiment is active.  
    if (((frame.isChanged || target.isChanged) ||  
        experiment.isRunning()) &&  
        (frame.getSize() == screen.getSize())) {  
        // Print banner.  
    }  
}
```

actividad

Analiza el siguiente código ¿Cómo lo refactorizarías?

```
void renderBanner() {  
    // Render banner only if we're in fullscreen mode and  
    // a change is requested either in frame or target, or  
    // experiment is active.  
    if (((frame.isChanged || target.isChanged) ||  
        experiment.isRunning()) &&  
        (frame.getSize() == screen.getSize())) {  
        // Print banner.  
    }  
}
```

Extraer a variable

```
void renderBanner() {  
    boolean isChanged = frame.isChanged || target.isChanged;  
    boolean mustRedraw = isChanged || experiment.isRunning();  
    boolean isFullScreen = frame.getSize() == screen.getSize();  
  
    if (isFullScreen && mustRedraw) {  
        // print banner  
    }  
}
```

actividad

Analiza el siguiente código ¿Cómo lo refactorizarías?

```
boolean hasDiscount(Order order) {  
    double basePrice = order.basePrice();  
    return basePrice > 1000;  
}
```

actividad

Analiza el siguiente código ¿Cómo lo refactorizarías?

```
boolean hasDiscount(Order order) {  
    double basePrice = order.basePrice();  
    return basePrice > 1000;  
}
```

Eliminar variable temporal

```
boolean hasDiscount(Order order) {  
    return order.basePrice() > 1000;  
}
```

actividad

Analiza el siguiente código ¿Cómo lo refactorizarías?

```
class Manager extends Employee {  
    public Manager(String name, String id, int grade) {  
        this.name = name;  
        this.id = id;  
        this.grade = grade;  
    }  
    // ...  
}
```

actividad

Analiza el siguiente código ¿Cómo lo refactorizarías?

```
class Manager extends Employee {  
    public Manager(String name, String id, int grade) {  
        this.name = name;  
        this.id = id;  
        this.grade = grade;  
    }  
    // ...  
}
```

```
class Manager extends Employee {  
    public Manager(String name, String id, int grade) {  
        super(name, id);  
        this.grade = grade;  
    }  
    // ...  
}
```

actividad

Analiza el siguiente código ¿Cómo lo refactorizarías?

```
int discount(int inputVal, int quantity) {  
    if (inputVal > 50) {  
        inputVal -= 2;  
    }  
    // ...  
}
```

Eliminar asignaciones a parámetros

```
int discount(int inputVal, int quantity) {  
    int result = inputVal;  
    if (inputVal > 50) {  
        result -= 2;  
    }  
    // ...  
}
```


actividad

Analiza el siguiente código ¿Cómo lo refactorizarías?

```
class Customer {  
    private String name = "name";  
    private String lastName = "lastname";  
  
    /**  
     * Method returns customer's lastname.  
     */  
    String getLnm() {  
        return lastName;  
    }  
  
    // other methods ...  
}
```

actividad

Analiza el siguiente código ¿Cómo lo refactorizarías?

```
class Customer {  
    private String name = "name";  
    private String lastName = "lastname";  
  
    /**  
     * Method returns customer's lastname.  
     */  
    String getlnm() {  
        return lastName;  
    }  
  
    // other methods ...  
}
```

Renombrar método

```
class Customer {  
    private String name = "name";  
    private String lastName = "lastname";  
  
    String getLastName() {  
        return lastName;  
    }  
  
    // other methods ...  
}
```

actividad

Analiza el siguiente código ¿Cómo lo refactorizarías?

```
class Bird {  
    // ...  
    double getSpeed() {  
        switch (type) {  
            case EUROPEAN:  
                return getBaseSpeed();  
            case AFRICAN:  
                return getBaseSpeed() - getLoadFactor() * numberOfCoconuts;  
            case NORWEGIAN_BLUE:  
                return (isNailed) ? 0 : getBaseSpeed(voltage);  
        }  
        throw new RuntimeException("Should be unreachable");  
    }  
}
```

actividad

Analiza el siguiente código ¿Cómo lo refactorizarías?

```
class Bird {  
    // ...  
    double getSpeed() {  
        switch (type) {  
            case EUROPEAN:  
                return getBaseSpeed();  
            case AFRICAN:  
                return getBaseSpeed() - getLoadFactor() * numberOfCoconuts;  
            case NORWEGIAN_BLUE:  
                return (isNailed) ? 0 : getBaseSpeed(voltage);  
        }  
        throw new RuntimeException("Should be unreachable");  
    }  
}
```

Reemplazar condicional por polimorfismo

```
abstract class Bird {  
    // ...  
    abstract double getSpeed();  
}  
  
class European extends Bird {  
    double getSpeed() {  
        return getBaseSpeed();  
    }  
}  
  
class African extends Bird {  
    double getSpeed() {  
        return getBaseSpeed() - getLoadFactor() * numberOfCoconuts;  
    }  
}  
  
class NorwegianBlue extends Bird {  
    double getSpeed() {  
        return (isNailed) ? 0 : getBaseSpeed(voltage);  
    }  
}  
  
// Somewhere in client code  
speed = bird.getSpeed();
```

actividad

Analiza el siguiente código ¿Cómo lo refactorizarías?

```
int withdraw(int amount) {  
    if (amount > _balance) {  
        return -1;  
    }  
    else {  
        balance -= amount;  
        return 0;  
    }  
}
```

actividad

Analiza el siguiente código ¿Cómo lo refactorizarías?

```
int withdraw(int amount) {  
    if (amount > _balance) {  
        return -1;  
    }  
    else {  
        balance -= amount;  
        return 0;  
    }  
}
```

Reemplazar código de error por Excepción

```
void withdraw(int amount) throws BalanceException {  
    if (amount > _balance) {  
        throw new BalanceException();  
    }  
    balance -= amount;  
}
```

actividad

Analiza el siguiente código ¿Cómo lo refactorizarías?

```
double getValueForPeriod(int periodNumber) {  
    if (periodNumber >= values.length) {  
        return 0;  
    }  
    return values[periodNumber];  
}
```

actividad

Analiza el siguiente código ¿Cómo lo refactorizarías?

```
double getValueForPeriod(int periodNumber) {  
    if (periodNumber >= values.length) {  
        return 0;  
    }  
    return values[periodNumber];  
}
```

```
double getValueForPeriod(int periodNumber) {  
    try {  
        return values[periodNumber];  
    } catch (ArrayIndexOutOfBoundsException e) {  
        return 0;  
    }  
}
```


actividad

Analiza el siguiente código ¿Cómo lo refactorizarías?

```
public double getPayAmount() {  
    if (isDead){  
        return deadAmount();  
    }  
    if (isSeparated){  
        return separatedAmount();  
    }  
    if (isRetired){  
        return retiredAmount();  
    }  
    return normalPayAmount();  
}
```

actividad

Analiza el siguiente código ¿Cómo lo refactorizarías?

```
public double getPayAmount() {  
    if (isDead){  
        return deadAmount();  
    }  
    if (isSeparated){  
        return separatedAmount();  
    }  
    if (isRetired){  
        return retiredAmount();  
    }  
    return normalPayAmount();  
}
```

```
public double getPayAmount() {  
    double result;  
    if (isDead){  
        result = deadAmount();  
    }  
    else {  
        if (isSeparated){  
            result = separatedAmount();  
        }  
        else {  
            if (isRetired){  
                result = retiredAmount();  
            }  
            else{  
                result = normalPayAmount();  
            }  
        }  
    }  
    return result;  
}
```

actividad

Analiza el siguiente código ¿Cómo lo refactorizarías?

```
double calculateTotal() {  
    double basePrice = quantity * itemPrice;  
    if (basePrice > 1000) {  
        return basePrice * 0.95;  
    }  
    else {  
        return basePrice * 0.98;  
    }  
}
```

actividad

Analiza el siguiente código ¿Cómo lo refactorizarías?

```
double calculateTotal() {  
    double basePrice = quantity * itemPrice;  
    if (basePrice > 1000) {  
        return basePrice * 0.95;  
    }  
    else {  
        return basePrice * 0.98;  
    }  
}
```

```
double calculateTotal() {  
    if (basePrice() > 1000) {  
        return basePrice() * 0.95;  
    }  
    else {  
        return basePrice() * 0.98;  
    }  
}  
  
double basePrice() {  
    return quantity * itemPrice;  
}
```

Reemplazar la variable temporal por la llamada al método
aue hace el cálculo

actividad

Analiza el siguiente código ¿Cómo lo refactorizarías?

```
double temp = 2 * (height + width);  
System.out.println(temp);  
temp = height * width;  
System.out.println(temp);
```

actividad

Analiza el siguiente código ¿Cómo lo refactorizarías?

```
double temp = 2 * (height + width);  
System.out.println(temp);  
temp = height * width;  
System.out.println(temp);
```

Dividir la variable temporal

```
final double perimeter = 2 * (height + width);  
System.out.println(perimeter);  
final double area = height * width;  
System.out.println(area);
```

actividad

Analiza el siguiente código ¿Cómo lo refactorizarías?

```
String foundPerson(String[] people){  
    for (int i = 0; i < people.length; i++) {  
        if (people[i].equals("Don")){  
            return "Don";  
        }  
        if (people[i].equals("John")){  
            return "John";  
        }  
        if (people[i].equals("Kent")){  
            return "Kent";  
        }  
    }  
    return "";  
}
```

actividad

Analiza el siguiente código ¿Cómo lo refactorizarías?

```
String foundPerson(String[] people){  
    for (int i = 0; i < people.length; i++) {  
        if (people[i].equals("Don")){  
            return "Don";  
        }  
        if (people[i].equals("John")){  
            return "John";  
        }  
        if (people[i].equals("Kent")){  
            return "Kent";  
        }  
    }  
    return "";  
}
```

```
String foundPerson(String[] people){  
    List candidates =  
        Arrays.asList(new String[] {"Don", "John", "Kent"});  
    for (int i=0; i < people.length; i++) {  
        if (candidates.contains(people[i])) {  
            return people[i];  
        }  
    }  
    return "";  
}
```

Sustituye el algoritmo

referencias

- Libro: Refactoring improving the design of existing code. Martin Fowler
- Libro: Clean Code: A Handbook of Agile Software Craftmanship. Robert C. Martin



- Vídeos: Refactorización de código https://www.youtube.com/playlist?list=PL97WzFMf575j7B2qMEsDhHoMt_Cj4bJLh
- Web: <https://refactoring.guru/>
- Web: <https://sourcemaking.com/refactoring/>