**Curso de desarrollo de software**

**Técnicas de refactorización**

**Actividad grupal**

Hoy en día, el desarrollo de software ágil es literalmente imprescindible y los equipos ágiles mantienen y amplían mucho su código de iteración en iteración, y sin una refactorización continua, esto es difícil de lograr. Esto se debe a que el código no refactorizado tiende a degradarse: dependencias poco saludables entre clases o paquetes, mala asignación de responsabilidades de clase, demasiadas responsabilidades por método o clase, código duplicado y muchas otras variedades de confusión y desorden. Por lo tanto, las ventajas incluyen una mejor legibilidad del código y una menor complejidad capacidades que pueden mejorar la capacidad de mantenimiento del código fuente y crear una arquitectura interna más expresiva.

Algunos consejos para hacer bien las técnicas de refactorización de código

* La refactorización del código se debe realizar como una serie de pequeños cambios, cada uno de los cuales mejora un poco el código existente y deja el programa en condiciones de funcionamiento. No mezcles un montón de refactorizaciones en un gran cambio.
* Cuando refactorices, definitivamente debe hacerlo usando TDD y CI. Si no puedes ejecutar esas pruebas después de cada pequeño paso en una refactorización, creas el riesgo de introducir errores.
* El código debería volverse más limpio.
* No se debe crear una nueva funcionalidad durante la refactorización. No mezcles la refactorización y el desarrollo directo de nuevas características. Intenta separar estos procesos al menos dentro de los límites de las confirmaciones individuales.

**Beneficios de la refactorización de código**

1. Ver la imágen completa

Si tienes un método principal que maneja toda la funcionalidad, lo más probable es que sea demasiado largo e increíblemente complejo. Pero si se divide en partes, es fácil ver lo que realmente se está haciendo.

2. Legibilidad para tu equipo

Haz que sea fácil de entender para tus compañeros, no lo escribas para ti mismo, piensa a largo plazo.

3. Mantenibilidad

La integración de actualizaciones y actualizaciones es un proceso continuo que es inevitable y debe ser bienvenido. Cuando el código base no está organizado y se basa en una base débil, los desarrolladores a menudo dudan en realizar cambios. Pero con la refactorización de código, el código organizado, el producto se construirá sobre una base limpia y estará listo para futuras actualizaciones.

4. Eficiencia

La refactorización de código puede considerarse una inversión, pero obtienes buenos resultados. Reduce el esfuerzo requerido para futuros cambios en el código, ya sea por ti o por otros desarrolladores, mejorando así la eficiencia.

5. Reducir la complejidad

Haz que sea más fácil para ti y tu equipo trabajar en el proyecto.

**Ejercicios**

Hay muchas técnicas de refactorización de código y presentamos algunas de las más comunes y útiles. Tu trabajo es encontrar las soluciones y como se pueden refactorizar.

Presenta código funcional en el lenguaje de programación favorito código antes de refactorizar y luego de refactorizar , en un repositorio llamado **Técnicas-Refactorización** de las técnicas entregadas.

Sugerencia: utilizar el catálogo de Martin Fowler:<https://refactoring.com/catalog/index.html>

**Ejemplo:**

**Antes**

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) {

// ...

}

}

**Después**

void renderBanner() {

if ((platform.toUpperCase().indexOf("MAC") > -1) &&

(browser.toUpperCase().indexOf("IE") > -1) &&

wasInitialized() && resize > 0 )

{

// ...

}

}

**Refactorización preparatoria**

Como desarrollador, hay cosas que puedes hacer con tu código base para que la creación de tu próxima característica sea un poco más fácil. Esto se llama a esto refactorización preparatoria [[1]](#footnote-1). Esto nuevamente se puede ejecutar usando la técnica rojo-verde descrita anteriormente. La refactorización preparatoria también puede implicar el pago de la deuda técnica que se acumuló durante las fases anteriores del desarrollo de características. Aunque es posible que los usuarios finales no estén de acuerdo con el equipo de ingeniería en tales esfuerzos, los desarrolladores casi siempre aprecian el valor de un buen ejercicio de refactorización.

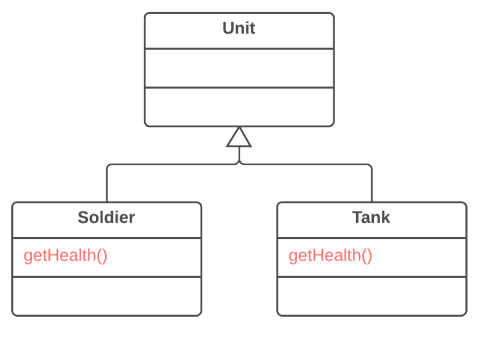
**Refactorización de abstracción [[2]](#footnote-2)**

La abstracción tiene su propio grupo de técnicas de refactorización, principalmente asociadas con el movimiento de la funcionalidad a lo largo de la jerarquía de herencia de clases, la creación de nuevas clases e interfaces y la sustitución de herencia por delegación y viceversa. Por ejemplo:

Pull up field, pull up method, pull up constructor body, push down field, push down method, extract subclass, extract superclass, extract interface, collapse hierarchy, form template method, replace inheritance with delegation, replace delegation with Inheritance, etc.

1. **Pull up method**

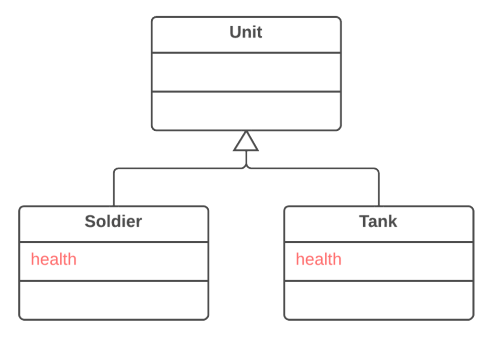
Problema: tus subclases tienen métodos que realizan un trabajo similar.



getHealth() en Uniy

1. **Pull Up Field**

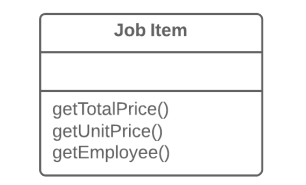
Problema: dos clases tienen el mismo campo



Health en Unit

1. **Extract Subclass**

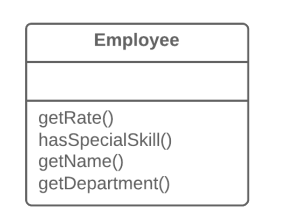
Problema: una clase tiene características que se usan solo en ciertos casos.



Solución: ………………………………………………………………..

1. **Extract Interface**

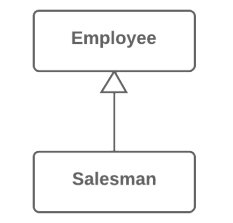
**Problema:** múltiples clientes están usando la misma parte de una interfaz de clase. Otro caso: parte de la interfaz en dos clases es la misma.

****

Solución: ………………………………………………………………..

1. **Collapse Hierarchy**

Problema: tienes una jerarquía de clases en la que una subclase es prácticamente igual a su superclase.



Solución: ………………………………………………………………..

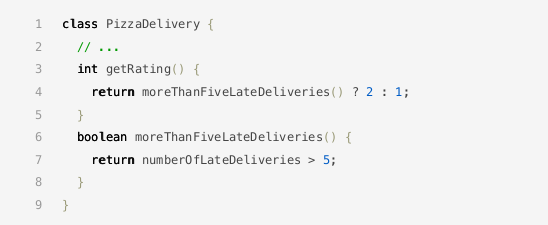
Hay dos tipos de esfuerzos de refactorización que se clasifican según el alcance y la complejidad. La ramificación por abstracción **es una técnica que algunos de los equipos utilizan para realizar refactorizaciones a gran escala**. La idea básica es construir una capa de abstracción que envuelva la parte del sistema que se va a refactorizar y la contraparte que eventualmente lo reemplazará. Por ejemplo: encapsulate field: fuerza el código para acceder al campo con métodos getter y setter, generalize type: crea tipos más generales para permitir más código compartido, etc.

**Refactorización de composición de métodos[[3]](#footnote-3)**

Gran parte de la refactorización se dedica a componer correctamente los métodos. En la mayoría de los casos, los métodos excesivamente largos son la raíz de todos los males. Los caprichos del código dentro de estos métodos ocultan la lógica de ejecución y hacen que el método sea extremadamente difícil de entender e incluso más difícil de cambiar. Las técnicas de refactorización de código de este grupo simplifican los métodos y eliminan la duplicación de código. Los ejemplos pueden ser: extract method, inline method, extract variable, inline Temp, replace Temp with Query, split temporary variable, remove assignments to parameters, etc.

1. **Inline method**

Problema: cuando el cuerpo de un método es más obvio que el propio método, utiliza esta técnica.



Solución:

class PizzaDelivery {

// ...

int getRating() {

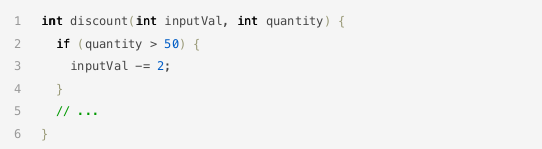
return numberOfLateDeliveries > 5 ? 2 : 1;

}

}

1. **Remove Assignments to Parameters**

Problema: se asigna algún valor a un parámetro dentro del cuerpo del método.



Solución:

int discount(int inputVal, int quantity) {

int result = inputVal;

if (quantity > 50) {

result -= 2;

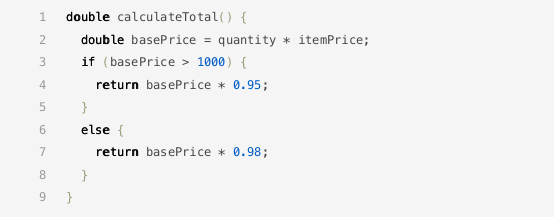
}

// ...

}

1. **Replace Temp with Query**

Problema:coloca el resultado de una expresión en una variable local para uso posterior en su código.



Solución:

double calculateTotal() {

if (basePrice() > 1000) {

return basePrice() \* 0.95;

}

else {

return basePrice() \* 0.98;

}

}

double basePrice() {

return quantity \* itemPrice;

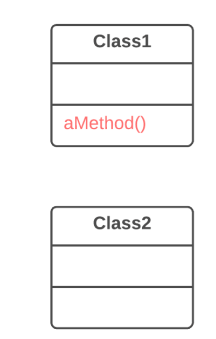
}

**Mover características entre objetos de refactorización**

Estas técnicas de refactorización de código muestran cómo mover de manera segura la funcionalidad entre clases, crear nuevas clases y ocultar los detalles de implementación del acceso público. Por ejemplo: move method, move field, extract class, inline class, hide delegate, remove middle man, introduce foreign method, introduce local extension, etc.

1. **Move method**

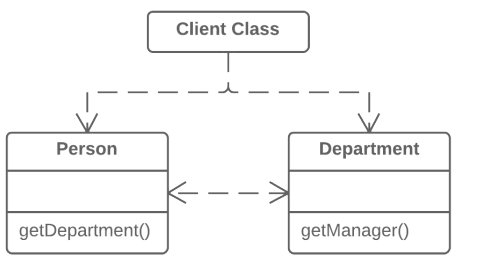
Problema: un método se usa más en otra clase que en su propia clase.



Solución: ………………………………………………………………..

1. **Hide delegate**

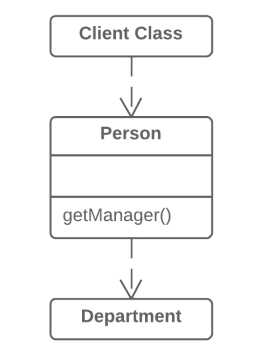
Problema: el cliente obtiene el objeto B de un campo o método del objeto А. Luego, el cliente llama a un método del objeto B.



Solución: ………………………………………………………………..

1. **Remove middle man**

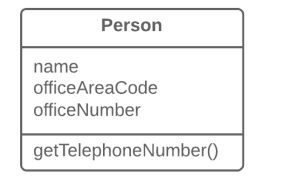
Problema: una clase tiene demasiados métodos que simplemente delegan a otros objetos.



Solución: ………………………………………………………………..

1. **Extract class**

Problema: cuando una clase hace el trabajo de dos, resulta incómodo



Solución: ………………………………………………………………..

1. **Introduce Foreign Method**

Problema: una clase de utilidad no contiene el método que necesita y no puede agregar el método a la clase.



Solución:

class Report {

// ...

void sendReport() {

Date newStart = nextDay(previousEnd);

// ...

}

private static Date nextDay(Date arg) {

return new Date(arg.getYear(), arg.getMonth(), arg.getDate() + 1);

}

}

1. **Introduce local extension**

Problema: una clase de utilidad no contiene algunos métodos que necesita. Pero no puedes agregar estos métodos a la clase.



Solución:

**Refactorización de expresiones condicionales**

Los condicionales tienden a volverse cada vez más complicados en su lógica con el tiempo, y también existen más técnicas para combatir esto. Por ejemplo: consolidate conditional expression, consolidate duplicate conditional fragments, decompose conditional, replace conditional with polymorphism, remove control flag, replace nested conditional with guard clauses,etc.

1. **Consolidate conditional expression**

Problema: tienes múltiples condicionales que conducen al mismo resultado o acción.



Solución:

double disabilityAmount() {

if (isNotEligibleForDisability()) {

return 0;

}

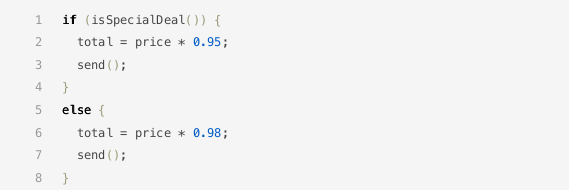
// Compute the disability amount.

// ...

}.

1. **Consolidate duplicate conditional fragments**

Problema: se puede encontrar un código idéntico en todas las ramas de un condicional.



Solución:

if (isSpecialDeal()) {

total = price \* 0.95;

}

else {

total = price \* 0.98;

}

send();

1. **Decompose conditional**

Problema: tienes complejos if-then/else o switch



Solución:

if (isSummer(date)) {

charge = summerCharge(quantity);

}

else {

charge = winterCharge(quantity);

}

1. **Replace Conditional with Polymorphism**

Problema: tienes un condicional que realiza varias acciones según el tipo de objeto o las propiedades.



Solución:

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();

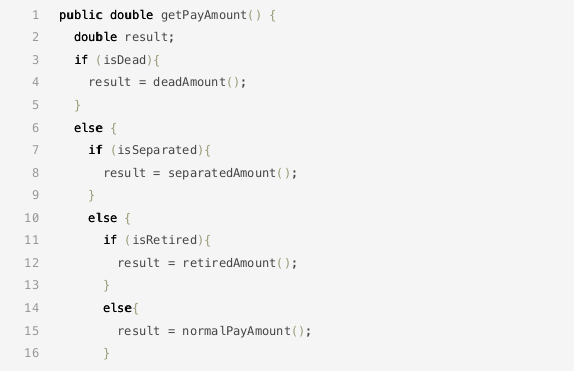
1. **Remove control Flag**

Problema: tienes una variable booleana que actúa como indicador de control para varias expresiones booleanas

Solución: break,continue,return

1. **Replace nested conditional with guard clauses**

Problema: tienes un grupo de condicionales anidados y es difícil determinar el flujo normal de ejecución del código.



Solución:

public double getPayAmount() {

if (isDead){

return deadAmount();

}

if (isSeparated){

return separatedAmount();

}

if (isRetired){

return retiredAmount();

}

return normalPayAmount();

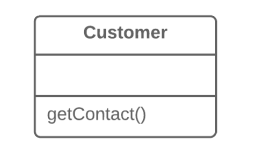
}

**Refactorización de llamadas a métodos**

Estas técnicas hacen que las llamadas a métodos sean más simples y fáciles de entender. Esto simplifica las interfaces para la interacción entre clases. Por ejemplo: add parameter, remove parameter, rename method, separate query from modifier, parameterize Method, introduce parameter object, preserve whole object, remove setting method, replace parameter with explicit methods, replace parameter with method call, etc.

1. **Add parameter**

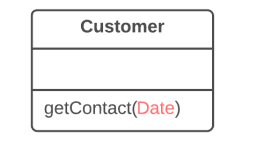
Problema: un método no tiene suficientes datos para realizar ciertas acciones.

****

Solución: getContact(Date)

1. **Remove parameter**

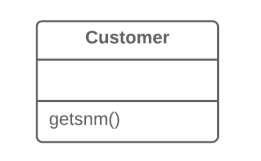
Problema: un parámetro no se usa en el cuerpo de un método.



Solución: getContact()

1. **Rename method**

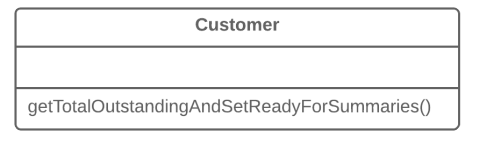
Problema: el nombre del método no explica lo que hace.



Solución: getSecondName()

1. **Separate query from modifier**

Problema: ¿tienes un método que devuelve un valor pero también cambia algo dentro de un objeto?



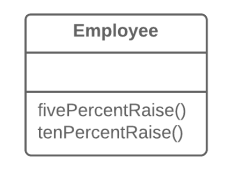
Solución:

getTotalOutstanding()

setReadyForSummaries()

1. **Parameterize Method**

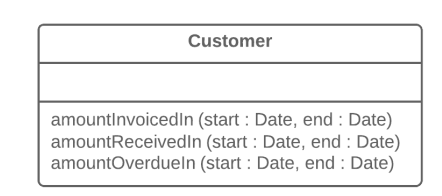
Problema: múltiples métodos realizan acciones similares que son diferentes solo en sus valores internos, números u operaciones.



Solución: raise(percentage)

1. **Introduce parameter object**

Problema: tus métodos contienen un grupo repetitivo de parámetros.



Solución:

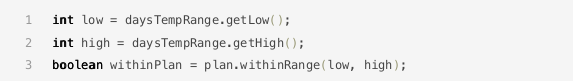
amountInvoicedln(date:DateRange)

amountReceivedln(date:DateRange)

amountOverdueln(date:DateRange)

1. **Preserve whole object**

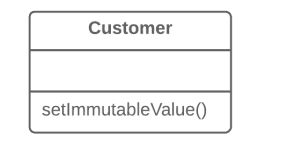
Problema: obtienes varios valores de un objeto y luego los pasa como parámetros a un método.



Solución: boolean withinPlan = plan.withinRange(daysTempRange);

1. **Remove setting method**

Problema: el valor de un campo debe establecerse solo cuando se crea, y no cambiará en ningún momento después de eso.



Customer solo

1. **Replace parameter with explicit methods**

Problema: un método se divide en partes, cada una de las cuales se ejecuta según el valor de un parámetro.



void setHeight(int arg) {

height = arg;

}

void setWidth(int arg) {

width = arg;

}

1. **Replace parameter with method call**

Problema: llama a un método de consulta y pasa tus resultados como parámetros de otro método, mientras que ese método podría llamar a la consulta directamente.



int basePrice = quantity \* itemPrice;

double finalPrice = discountedPrice(basePrice);

**Refactorización en la descomposición del código en piezas más lógicas**

La creación de componentes descompone el código en unidades semánticas reutilizables que presentan interfaces claras, bien definidas y fáciles de usar. Por ejemplo: **extract class** mueve parte del código de una clase existente a una nueva clase, **extract method**, para convertir parte de un método más grande en un nuevo método. Al dividir el código en partes más pequeñas, es más fácil de entender. Esto también es aplicable a las funciones.

1. **Extract method**

Problema: tienes un fragmento de código que se puede agrupar.



void printOwing() {

printBanner();

printDetails(getOutstanding());

}

void printDetails(double outstanding) {

System.out.println("name: " + name);

System.out.println("amount: " + outstanding);

}

1. **Preparatory refactoring** [↑](#footnote-ref-1)
2. **Branching by abstraction refactoring** [↑](#footnote-ref-2)
3. **Composing methods refactoring** [↑](#footnote-ref-3)