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]. 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.

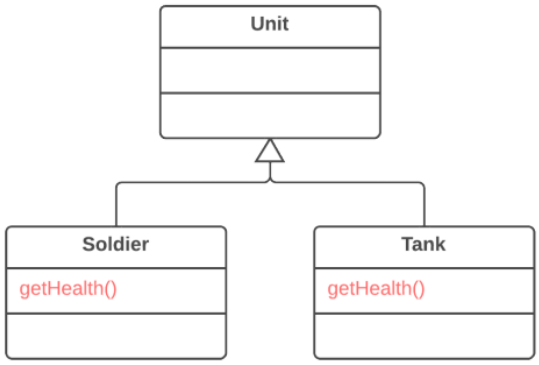
[1] **Preparatory refactoring**

Refactorización de abstracción [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:

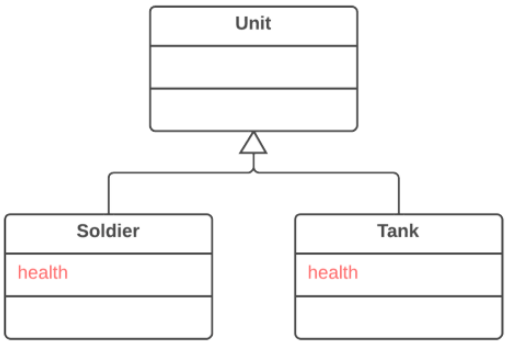
Pull up field, pull up method, pull up constructor body, push down field, push down method, extract subclass, extract superclass, extract interface, collapse hierarchy, form template method, replace inheritance with delegation, replace delegation with Inheritance, etc.

1. **Pull up method**
   Problema: tus subclases tienen métodos que realizan un trabajo similar.



   Solución : **Hacemos que esos metodos sean identicos y los movemos a la super clase UNIT**

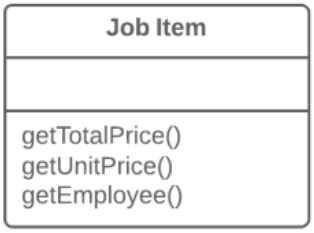
2. **Pull Up Field**
   Problema: dos clases tienen el mismo campo



   Solución: **Eliminamos ese campo de la SUBLCLASE y lo llevamos a la SUPERCLASE**

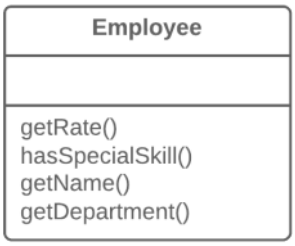
3. **Extract Subclass**
   Problema: una clase tiene características que se usan solo en ciertos casos.

[2] **Branching by abstraction refactoring**



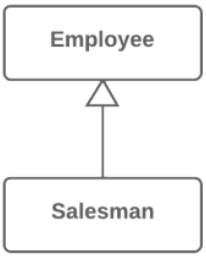
Solución: **Creamos una SUBCLASE y si requiero de esos métodos los llamo.**

4. **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: **movemos esos metodos a su propia interfaz, asi luego nos instanciamos.**

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



Solución: **Combinamos las dos clases, y eliminamos la subclase para quedarnos con la ClasePadre, si es el caso.**

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]

[3] **Composing methods refactoring**

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.

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

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

   Solución: **Reemplazamos esa llamada a moreThanFiveLateDeliveries() con numberOfLateDeliveries y eliminamos ese metodo porque es miuy obvio.**

7. **Remove Assignments to Parameters**
   Problema: se asigna algún valor a un parámetro dentro del cuerpo del método.

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

   Solución: **Primero creamos una variable y a esa variable le asignamos el “inputVal” , y ahí recien cambimos la variable, y asi no tocamos el parametro como variable, sino indirectamente.**

8. **Replace Temp with Query**
   Problema: coloca el resultado de una expresión en una variable local para uso posterior en su código.

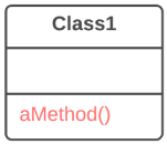
```
double calculateTotal() {
  double basePrice = quantity * itemPrice;
  if (basePrice > 1000) {
    return basePrice * 0.95;
  }
  else {
    return basePrice * 0.98;
  }
}
```

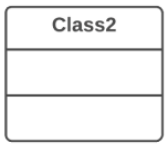
Solución: **creamos un metodo que nos devuelva “basePrice” y en el metodo “calculateTotal” llamamos a ese metodo nuevo con el valor del “basePrice”.**

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.

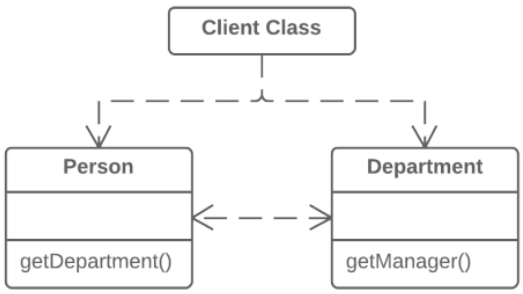
9. **Move method**
   Problema: un método se usa más en otra clase que en su propia clase.





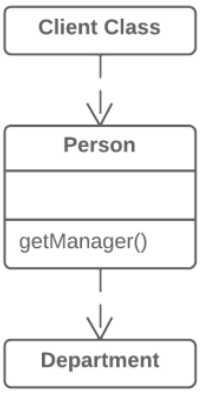
Solución: **creamos “aMethod” en la clase2 y copiamos el codigo de “aMethod” de la clase1 a la clase2.**

10. **Hide delegate**
    Problema: el cliente obtiene el objeto B de un campo o método del objeto A. Luego, el cliente llama a un método del objeto B.



Solución: **La claseCliente necesita “getManager()” de la claseDepartamento, pero podemos crear ese metodo “getManager()” en la clasePersona ya que ambos dependen (claseDepartamento con clasePersona), asi la clase Cliente solo depende de la clasePersona , ya no de claseDepartamento.**

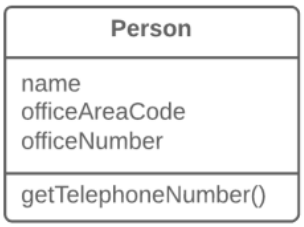
11. **Remove middle man**
    Problema: una clase tiene demasiados métodos que simplemente delegan a otros objetos.



Solución: ……..

12. **Extract class**

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



Solución: **Creamos una clase "Number" y ahí llevamos a los campos officeAreaCode, officeNumber.**

13. **Introduce Foreign Method**

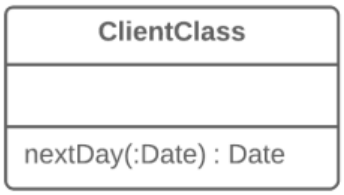
Problema: una clase de utilidad no contiene el método que necesita y no puede agregar el método a la clase.

```
class Report {
  // ...
  void sendReport() {
    Date nextDay = new Date(previousEnd.getYear(),
      previousEnd.getMonth(), previousEnd.getDate() + 1);
    // ...
  }
}
```

Solución: ……..

14. **Introduce local extension**

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



Solución: **Creamos una nueva clase que contenga esos metodos**

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.

15. **Consolidate conditional expression**
    Problema: tienes múltiples condicionales que conducen al mismo resultado o acción.

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

Solución: **Lo mejor seria colocar todos esos condiciones en un solo if, ya que todos me retornan 0.**
If (seniority <2 && monthsDisabled >12 && isPartTime==True) {
return 0;
}
O crear un metodo donde evalua todo eso.

16. **Consolidate duplicate conditional fragments**
    Problema: se puede encontrar un código idéntico en todas las ramas de un condicional.

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

Solución: **movemos ese send() fuera del condicional**

17. **Decompose conditional**
    Problema: tienes complejos if-then/else o switch

```
if (date.before(SUMMER_START) || date.after(SUMMER_END)) {
  charge = quantity * winterRate + winterServiceCharge;
}
else {
  charge = quantity * summerRate;
}
```

    Solución: **creamos un metodo donde evalue el cargo de verano y el cargo de invierno.**
    x
18. **Replace Conditional with Polymorphism**
    Problema: tienes un condicional que realiza varias acciones según el tipo de objeto o las propiedades.

```
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");
  }
}
```

    Solución: ……..

19. **Remove control Flag**
    Problema: tienes una variable booleana que actúa como indicador de control para varias expresiones booleanas

    Solución: ……..

20. **Replace nested conditional with guard clauses**
    Problema: tienes un grupo de condicionales anidados y es difícil determinar el flujo normal de ejecución del código.

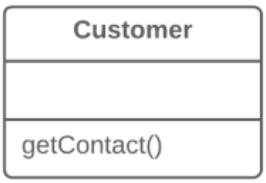
```
public double getPayAmount() {
  double result;
  if (isDead){
    result = deadAmount();
  }
  else {
    if (isSeparated){
      result = separatedAmount();
    }
    else {
      if (isRetired){
        result = retiredAmount();
      }
      else{
        result = normalPayAmount();
      }
```

Solución: ……..

Refactorización de llamadas a métodos

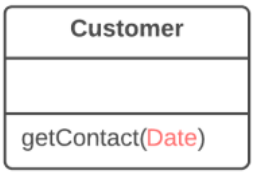
Estas técnicas hacen que las llamadas a métodos sean más simples y fáciles de entender. Esto simplifica las interfaces para la interacción entre clases. Por ejemplo: add parameter, remove parameter, rename method, separate query from modifier, parameterize Method, introduce parameter object, preserve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

21. **Add parameter**
    Problema: un método no tiene suficientes datos para realizar ciertas acciones.



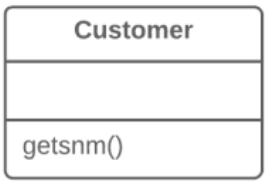
Solución: ……..

22. **Remove parameter**
    Problema: un parámetro no se usa en el cuerpo de un método.



Solución: ……..

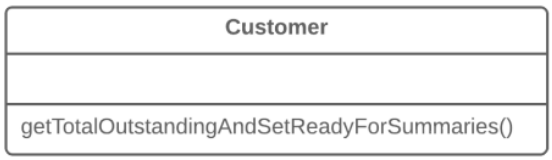
23. **Rename method**
    Problema: el nombre del método no explica lo que hace.



Solución: ……..

24. Separate query from modifier

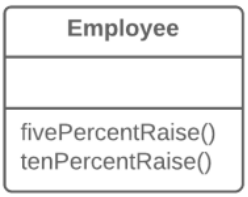
Problema: ¿tienes un método que devuelve un valor pero también cambia algo dentro de un objeto?



Solución: ……..

25. Parameterize Method

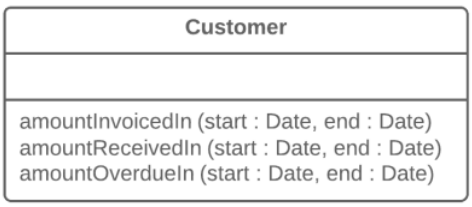
Problema: múltiples métodos realizan acciones similares que son diferentes solo en sus valores internos, números u operaciones.



Solución: ……..

26. Introduce parameter object

Problema: tus métodos contienen un grupo repetitivo de parámetros.



Solución: ……..

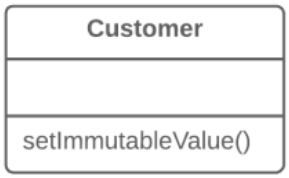
27. Preserve whole object

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

```
int low = daysTempRange.getLow();
int high = daysTempRange.getHigh();
boolean withinPlan = plan.withinRange(low, high);
```

Solución: ……..

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



    Solución: ……..

29. **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 setValue(String name, int value) {
  if (name.equals("height")) {
    height = value;
    return;
  }
  if (name.equals("width")) {
    width = value;
    return;
  }
  Assert.shouldNeverReachHere();
}
```

    Solución: ……..

30. **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 seasonDiscount = this.getSeasonalDiscount();
double fees = this.getFees();
double finalPrice = discountedPrice(basePrice, seasonDiscount, fees);
```

    Solución: ……..

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.

31. **Extract method**

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

```
void printOwing() {
  printBanner();

  // Print details.
  System.out.println("name: " + name);
  System.out.println("amount: " + getOutstanding());
}
```

Solución: ……..