Refactoring: otra práctica de la Programación eXtrema

En ediciones anteriores estuvimos hablando de TDD, sus beneficios y forma de implementarlo. En esta edición, nos concentraremos en el *Refactoring*: otra de las prácticas técnicas sugeridas por eXtreme Programming.

Escrito por: **Eugenia Bahit** (Arquitecta GLAMP & Agile Coach)

Eugenia es **Arquitecta de Software**, **docente** instructora de tecnologías **GLAMP** (GNU/Linux, Apache, MySQL, Python y PHP) y **Agile coach** (UTN) especializada en Scrum y eXtreme Programming. Miembro de la **Free Software Foundation** e integrante del equipo de **Debian Hackers**.

Webs:
Cursos de programación a Distancia: www.cursosdeprogramacionadistancia.com
Web personal: www.eugeniabahit.com

Redes sociales:
Twitter / Identi.ca: **@eugeniabahit**

Refactoring, es una de las prácticas técnicas sugeridas por XP[8] que consiste en mejorar el código fuente de una aplicación, sin que dichas modificaciones, afecten el comportamiento externo del sistema.

Cuando se programa con TDD[9] se hace imposible evitar *refactorizar* el código fuente ya que de los propios test deriva esta necesidad. Sin embargo, las técnicas de *Refactoring* pueden implementarse como una buena práctica de programación aunque no se programe con TDD.

Existen diferentes tipos de *refactorizaciones* que pueden ser necesarias implementar al código de nuestra aplicación. Cada tipo, representa una técnica diferente de *refactorización*. Por ejemplo, eliminar código redundante requiere de una técnica diferente a dividir los algoritmos de un método para crear métodos derivados.

8 Extreme Programming
9 Test Driven Development (Desarrollo guiado por pruebas)

Un problema no es un error...

Antes de continuar, habría que diferenciar el término "problema" de la palabra "error", para no generar confusiones.

El error en sí, es una falla en el código fuente que impide el correcto comportamiento del sistema. Mientras que el problema, puede definirse como "algo que huele mal en el código fuente"[10] pero sin embargo, no impide el correcto funcionamiento de la aplicación.

Los problemas que se pueden presentar en el código fuente de una aplicación, dependen de muchísimos factores y en gran parte de los casos, encuentran una relación directa con el paradigma de programación empleado así como en el lenguaje que se utilice.

Si se intentara abarcar todos los problemas posibles, la lista podría tornarse infinita, tediosa y hasta inútil o incluso confusa. Es por ello, que solo abarcaremos los problemas más frecuentes que puedan considerarse generales con independencia del lenguaje en que se programe. Trataré de incluir ejemplos tanto en Python como en PHP para hacer el tema más general.

La regla

En el mundo del *Refactoring*, haciendo una analogía con el béisbol, suele utilizarse la regla ***"Tres Strikes[11] y ¡refactoriza!"***. Esta regla puede traducirse como:

> *La primera vez que hagas algo, solo hazlo. La segunda vez que hagas algo similar, notarás que estás duplicando código, pero lo harás de todas formas. La tercera vez que te enfrentes al mismo caso, refactoriza.*

Cuando se está programando una aplicación con TDD, como hemos visto en ediciones anteriores, el proceso de desarrollo se está dividiendo en dos acciones concretas: programar y *refactorizar*. Esto es, a medida que vamos creando nuevos métodos, vamos *refactorizando* el código para eliminar redundancias y en definitiva, hacer el código -del test- más legible y así obtener un mejor rendimiento. Pero no estamos refactorizando el SUT constantemente (aunque sí lo *refactorizamos*), puesto que éste, tiene un momento y lugar para ser *refactorizado*.

---

10 Kent Beck, uno de los creadores de eXtreme Programming, es quien introdujo el término "bad smells" (malos olores) para referirse de manera global, a aquellas expresiones y algoritmos poco claros que generan confusión en el código fuente de un sistema, tornándolo más complejo de lo que debería ser.

11 En el béisbol, un strike es una anotación negativa para el bateador ofensivo, cuando la pelota no es lanzada hacia el diamante. Al tercer strike anotado, termina el turno del bateador.

La *refactorización* del SUT, implica que lo primero que debemos hacer es cumplir el objetivo (programar aquello que se necesita) y luego, *refactorizar* el código del SUT cuando:

- Se agregue un nuevo método;
- Se corrija un *bug*;
- Se haga una revisión de código;

Pero siempre, respetando la regla de "los tres *strikes*". Una vez identificado el momento, solo será cuestión de identificar el problema a fin de poder elegir la solución indicada.

Una solución a cada problema

Como comentamos anteriormente, no haremos una extensa lista de problemas, sino que nos centraremos en problemas generales. Muchas de las soluciones sugeridas en este artículo, pueden hallarse en SourceMaking.com[12], sitio donde se puede encontrar una completa clasificación de problemas[13] y sus respectivas soluciones[14].

PROBLEMA: Variables de uso temporal mal implementadas

En principio, definiremos a las variables de uso temporal, como aquellas variables que son asignadas en el ámbito local de un método de clase y son necesarias temporalmente, solo en ese método, sin ser llamadas o requeridas por otros métodos.

> Por favor, notar que cuando se habla de métodos de clase, el significado podría aplicarse también a funciones y procedimientos.

Generalmente representan un problema en tres casos, los cuáles veremos a continuación.

Caso 1: Variables de uso temporal que definen una acción concreta:

```
# Código PHP
$var = ($a * $b ) / (int)$c;
```

12 http://sourcemaking.com/refactoring. Nótese que algunas de las técnicas expuestas en el sitio Web referido, no se mencionan en este curso, por considerarlas poco apropiadas. Esto es debido a que algunas prácticas son más específicas de lenguajes como Java, mientras que a otras, las considero contrarias a las buenas prácticas de la programación orientada a objetos y por lo tanto, contraproducentes.
13 "Bad Smells in Code" http://sourcemaking.com/refactoring/bad-smells-in-code
14 Diferentes técnicas de refactorización: http://sourcemaking.com/refactoring

```
# Código Python
var = (a * b) / int(c)
```

En el ejemplo anterior, vemos una variable de uso temporal, que define una acción concreta: dividir el producto de dos factores. Esto representa un problema, ya que las acciones son responsabilidad de los métodos y no de las variables. En estos casos, la solución, es transferir la responsabilidad de la acción a un método:

```
# Código PHP
$var = dividir($a, $b, $c);

function dividir($a, $b, $c) {
    return ($a * $b ) / (int)$c;
}


# Código Python
var = dividir(a, b, c)

def dividir(a, b, c):
    return (a * b) / int(c)
```

Las variables de uso temporal que definen un valor directo: $var = 15; o por el retorno de la llamada a una función: $var = strlen($variable); no necesitan transferir su responsabilidad a otro método.

Caso 2. Variables de uso temporal son requeridas por más de un método:

```
# Código PHP
function metodo_a() {
    $a = 15;
    $b = 100;
    $c = 2;
    $var = self::dividir($a, $b, $c);
    // continuar...
}

private static function dividir($a, $b, $c) {
    return ($a * $b ) / $c;
}


# Código Python
def metodo_a(self):
    a = 15
    b = 100
    c = 2
    var = self._dividir(a, b, c)
    # continua...
```

```
def _dividir(self, a, b, c):
    return (a * b) / c
```

En el ejemplo, anterior, las variables temporales `$a`, `$b` y `$c`, son requeridas por dos métodos y se están definiendo como tales en un método, necesitando ser pasadas como parámetros. Aquí, la solución, será convertir las variables temporales, en propiedades de clase:

```
# Código PHP
function metodo_a() {
    self::$a = 15;
    self::$b = 100;
    self::$c = 2;
    $var = self::dividir();
    // continuar...
}

private static function dividir() {
    return (self::$a * self::$b ) / self::$c;
}


# Código Python
def metodo_a(self):
    self.a = 15
    self.b = 100
    self.c = 2
    var = self._dividir()
    # continua...

def _dividir(self):
    return (self.a * self.b) / self.c
```

Caso 3. Variables de uso temporal que reasignan parámetros:

```
# Código PHP
function foo($a) {
    $a = strtoupper($a);
    // continuar ...
}

# Código Python
def foo(a):
    a = a.upper()
    # continua...
```

En casos como éste, la confusión puede ser grande: un parámetro es un parámetro y una variable temporal, una variable temporal. Es entonces, cuando variables temporales no deben tener el mismo nombre que los parámetros:

```
# Código PHP
function foo($a) {
    $b = strtoupper($a);
    // continuar ...
}


# Código Python
def foo(a):
    b = a.upper()
    # continua...
```

PROBLEMA: Métodos que reciben parámetros

Aquí debe hacerse una notable distinción entre parámetros, variables de uso temporal y propiedades de clase. Y esta distinción, está dada por la finalidad de cada una:

- Las variables de uso temporal, como hemos visto antes, están destinadas a definir un valor concreto al cual se hará referencia solo en el ámbito donde se haya definido.
- Las propiedades de clase, son características inherentes al objeto a las cuales se hará referencia desde diversos ámbitos.
- Y finalmente, los parámetros, serán valores adicionales, que no pueden ser considerados propiedades del objeto pero que sin embargo, son requeridos para que una acción, por ejemplo, modifique las propiedades de un objeto.

Veamos algunos casos de mal uso de parámetros:

```
/*
    Parámetros cuya única finalidad es modificar de forma directa
    el valor de una o más propiedades
*/
class Producto {

    function __construct($nombre, $precio, $barcode) {
        $this->nombre = $nombre;
        $this->precio = $precio;
        $this->barcode = $barcode;
    }

}

$nombre = 'Agua mineral';
$precio = 5.75;
$barcode = 6543922234321;
$producto = new Producto($nombre, $precio, $barcode);

/*
    Métodos que solo recurren a sus parámetros actuando como funciones
*/
class Usuario(object):
```

```
    def validar_usuario(self, username, passwd):
        if username == 'pepe' and passwd == '123':
            return True
```

Como regla general, los parámetros deben ser evitados toda vez que sea posible, reemplazándolos por propiedades de clase. A no ser que una propiedad deba componerse de un objeto, los valores de la misma deberán ser modificados de forma directa evitando para ello el uso de parámetros:

```
/*
    Parámetros cuya única finalidad es modificar de forma directa
    el valor de una o más propiedades
*/
class Producto {

    function __construct() {
        $this->nombre = '';
        $this->precio = '';
        $this->barcode = '';
    }

}

$producto = new Producto();
$producto->nombre = 'Agua mineral';
$producto->precio = 5.75;
$producto->barcode = 6543922234321;


/*
    Métodos que solo recurren a sus parámetros actuando como funciones
*/
class Usuario(object):

    def validar_usuario(self):
        if self.username == 'pepe' and self.passwd == '123':
            return True
```

PROBLEMA: Expresiones extensas

Muchas veces, podremos encontrarnos con expresiones que debido a su extensión, se hacen difíciles de leer e inducen a una gran confusión:

```
# Código PHP
return ((in_array('abc', $array) || in_array('bcd', $array)) && (in_array('cde',
$array) || in_array('def', $array))) ? 'OK' : 'ERROR';

# Código Python
return 'OK' if (('abc' in lista or 'bcd' in lista) and ('cde' in lista or 'def' in
lista)) else 'ERROR'
```

Cuando estamos en presencia de expresiones tan extensas, lo mejor es -aquí sí- utilizar variables de uso temporal para simplificar dichas expresiones:

```
# Código PHP
$a = in_array('abc', $array);
$b = in_array('bcd', $array);
$c = in_array('cde', $array);
$d = in_array('def', $array);
$ab = ($a || $b);
$cd = ($c || $d);

return ($ab && $cd) ? 'OK' : 'ERROR';

# Código Python
a = 'abc' in lista
b = 'bcd' in lista
c = 'cde' in lista
d = 'def' in lista
ab = a or b
cd = c or d

return 'OK' if ab and cd else 'ERROR'
```