Introducción al desarrollo dirigido por pruebas

Mucho se ha hablado de una de las doce prácticas técnicas más beneficiosas de la Programación eXtrema, pero a excepción del excelente libro *«Diseño Ágil con TDD*[18]*»* de *Carlos Blé Jurado*, muy poca es la bibliografía que podremos encontrar en español. Y si a esto le sumamos lo complejo que resulta asimilar el tema, se hace evidente la necesidad de sumar textos en nuestra lengua, que nos ayuden a comprender la importancia de esta técnica y aprender a implementarla.

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**

TDD (siglas de *Test-Driven Development* - Desarrollo conducido por pruebas) es una **técnica de programación** que consiste en guiar el desarrollo de una aplicación, por medio de **Test Unitarios**. Los Test Unitarios (Unit Test, en inglés) no son más que algoritmos que emulan lo que la aplicación se supone debería hacer, convirtiéndose así, en un modo simple de probar que lo que "piensas" programar, realmente funciona.

A la vez, esta técnica te permitirán saber: qué, cómo, cuáles y cuántos algoritmos necesitarás desarrollar para que tu aplicación haga lo que realmente debe hacer y no falle, ante ciertos casos que no son los que se esperaban.

18 http://www.dirigidoportests.com/el-libro

> *El objetivo del TDD no es "adivinar", sino probar y actuar, ya que los Test Unitarios serán una guía para entender como funciona el código, ayudándote a organizarlo de manera clara, legible y simple, a la vez de servir como "documentación"*

Carlos Blé Jurado en su libro **Diseño Ágil con TDD** nos define la técnica de TDD como:

> *"[...] la respuesta a las grandes preguntas: ¿Cómo lo hago? ¿Por dónde empiezo? ¿Cómo se qué es lo que hay que implementar y lo que no? ¿Cómo escribir un código que se pueda modificar sin romper funcionalidad existente? [...]"*

Según **Kent Beck** -uno de los co-fundadores de la Programación eXtrema (XP)-, implementar TDD nos otorga seis grandes ventajas:

1. **La calidad del software aumenta** disminuyendo prácticamente a cero, la cantidad de *bugs* en la aplicación;
2. Conseguimos **código altamente reutilizable** puesto que los test nos obligan a desarrollar algoritmos genéricos;
3. **El trabajo en equipo se hace más fácil**, une a las personas, ya que al desarrollar con test, nos aseguramos de no romper funcionalidades existentes de la aplicación;
4. Nos permite **confiar en nuestros compañeros de equipo** aunque tengan menos experiencia. Esto es, debido a que el hecho de tener que desarrollar test antes de programar el algoritmo definitivo, nos asegura -independientemente del grado de conocimiento y experiencia del desarrollador- que el algoritmo, efectivamente hará lo que se supone debe hacer y sin fallos;
5. Escribir el ejemplo (test) antes que el código **nos obliga a escribir el mínimo de funcionalidad necesaria, evitando sobre-diseñar**, puesto que desarrollando lo mínimamente indispensable, se obtiene un panorama más certero de lo que la aplicación hace y cuál y cómo es su comportamiento interno;
6. **Los tests son la mejor documentación técnica** que podemos consultar a la hora de entender qué misión cumple cada pieza del rompecabezas, ya que cada test, no es más que un "caso de uso" traducido en idioma informático.

Características de los Test Unitarios

Los Test Unitarios (o técnica de *Unit Testing*), representan el alma de la programación dirigida por pruebas. Son algoritmos independientes que se encargan de verificar -de manera simple y rápida- el comportamiento de **una parte mínima de código**, de forma **individual**, **independiente** y **sin alterar el funcionamiento de otras partes de la aplicación**.

Un Test Unitario posee **cuatro características particulares** que debe conservar a fin de considerarse "unitario". Estas son:

1. **Atómico:**

   Prueba una parte mínima de código.

   Dicho de manera simple, cada test unitario debe probar una -y solo una- "acción" realizada por un método.

   Por ejemplo, para un método que retorna el neto de un monto bruto más el IVA correspondiente, deberá haber un test que verifique recibir en forma correcta el importe bruto, otro que verifique el cálculo del IVA sobre un importe bruto y finalmente, un tercer test unitario que verifique el cálculo de un importe bruto más su IVA correspondiente.

2. **Independiente:**

   Cada Test Unitario DEBE ser independiente de otro.

   Por ejemplo, siguiendo el caso anterior, el test que verifique la suma de un importe bruto más su IVA correspondiente, no debe depender del test que verifica el cálculo del IVA.

3. **Inocuo:**

   Podría decirse que cada test unitario debe ser inofensivo para el sistema.

   Un test unitario DEBE poder correrse sin alterar ningún elemento del sistema, es decir, que no debe, por ejemplo, agregar, editar o eliminar registros de una base de datos.

4. **Rápido:**

   La velocidad de ejecución de un test unitario cumple un papel fundamental e ineludible en el desarrollo guiado por pruebas, ya que de la velocidad de ejecución de un test, dependerá de manera proporcional, la velocidad con la que una funcionalidad se desarrolle.

Anatomía de un Test

Los Test Unitarios se realizan, en cualquier lenguaje de programación, mediante herramientas que proveen un completo entorno de trabajo (*Frameworks* para *Unit Testing*). Estos entornos de trabajo, son desarrollados con un **formato** determinado, conocido como **xUnit**.

De allí, que los *frameworks* para *Unit Testing* que cumplen con dicho formato, suelen tener nombres compuestos por una abreviatura del lenguaje de programación, seguida del término "*unit*". Por ejemplo: **PyUnit** (Python), **PHPUnit** (PHP), **ShUnit** (Shell Scripting), etc.

Exceptuando el caso de Shell Scripting, **los frameworks xUnit de lenguajes que soportan la orientación a objetos, utilizan éste paradigma** tanto en su anatomía de desarrollo como para su implementación (creación de los test unitarios).

Por lo tanto, los Test Unitarios se agrupan en **clases**, denominadas **Test Case**, que heredan de una clase del *framework* xUnit, llamada **xTestCase**:

```
# Ejemplo con PyUnit
import unittest

class BalanceContableTestCase(unittest.TestCase):
    pass


# Ejemplo con PHPUnit
class BalanceContableTest extends PHPUnit_Framework_TestCase {
}
```

Los métodos contenidos en una clase Test Case, pueden o no, ser Test Unitarios. **Los Test Unitarios** contenidos en una clase Test Case, **deben contener el prefijo `test_` en el nombre del método** a fin de que el *framework* los identifique como tales:

```
# Ejemplo con PyUnit
import unittest

class BalanceContableTestCase(unittest.TestCase):

    def test_calcular_iva(self):
        pass


# Ejemplo con PHPUnit
class BalanceContableTest extends PHPUnit_Framework_TestCase {

    public function test_calcular_iva() {
    }

}
```

Otra ventaja que los *frameworks* xUnit nos proveen, es la facilidad de poder crear **dos métodos especiales** dentro de una clase Test Case **-que no son test-**, los cuales están

destinados a **preparar el escenario** necesario para correr los test de esa clase y **eliminar aquello que se desee liberar**, una vez que el test finalice. Estos métodos, son los denominados **`setUp()`** y **`tearDown()`** respectivamente:

```
# Ejemplo con PyUnit
class BalanceContableTestCase(unittest.TestCase):

    def setUp(self):
        self.importe_bruto = 100
        self.alicuota_iva = 21

    def tearDown(self):
        self.importe_bruto = 0
        self.alicuota_iva = 0

    def test_calcular_iva():
        pass


# Ejemplo con PHPUnit
class BalanceContableTest extends PHPUnit_Framework_TestCase {

    public function setUp() {
        $this->importe_bruto = 100;
        $this->alicuota_iva = 21;
    }

    public function tearDown() {
        $this->importe_bruto = 0;
        $this->alicuota_iva = 0;
    }

    public function test_calcular_iva() {
    }

}
```

Los métodos `setUp()` y `tearDown()` se ejecutan antes y después de cada test, respectivamente.

Cada Test estará dividido a la vez, en **tres partes** identificadas por las siglas **AAA** las cuáles representan a las tres "acciones" que son necesarias llevar a cabo, para dar forma a los Tests: ***Arrange***, ***Act*** *and* ***Assert*** (preparar, actuar y afirmar).

Preparar consiste en definir y configurar (dentro del test) los recursos necesarios para poder **actuar**, es decir, hacer la llamada al código del **Sistema cubierto por Test** (**SUT**) que se desea probar. Esto se conoce como "cobertura de código" o *Code Coverage* (o *Coverage* a secas).

Finalmente, **afirmar** el resultado de un test, se refiere a invocar al método `assert` del *Framework* xUnit, que sea necesario para "afirmar que el resultado obtenido durante la actuación, es el esperado".

Los métodos `assert` (métodos de afirmación), son métodos que vienen definidos por defecto en el *framework* xUnit, destinados a verificar un resultado. Los nombres de estos métodos suelen ser muy similares entre los diversos *frameworks*, como por

ejemplo `assertTrue()` para verificar que el valor del resultado de la actuación, devuelto por el SUT, sea `True`. Pero los veremos en detalle en la próxima entrega.

> Si quieres ir conociendo los métodos assert, puedes ir visitando las referencias oficiales: **PyUnit:** http://docs.python.org/2/library/unittest.html#test-cases y **PHPUnit:** http://www.phpunit.de/manual/current/en/writing-tests-for-phpunit.html#writing-tests-for-phpunit.assertions

Por favor, notar que mientras que PyUnit viene incluido como un módulo estándar de Python, PHPUnit, requiere de instalación por separado. En la próxima entrega, abarcaremos también la instalación de PHPUnit.

Algoritmo para las pruebas unitarias

Existe un algoritmo para escribir Test Unitarios, el cual consiste en:

- **PRIMER PASO:** Escribir el Test y hacer que falle (retornando en el SUT, un valor contrario al que se espera que devuelva en realidad)
- **SEGUNDO PASO:** Escribir la mínima cantidad de código necesaria (en el SUT) para que el test pase.
- **TERCER PASO:** Escribir un nuevo test (para el mismo SUT) y esperando que falle el nuevo test.
- **CUARTO PASO:** Escribir nuevamente, en el SUT, la mínima cantidad de código necesaria para que ambos test pasen (generalmente, la primera vez, para que el test pase, se "*hardcodea*" -se escribe "a mano"- el valor de retorno del SUT y, en esta segunda oportunidad, se escribe el mínimo algoritmo necesario para que el SUT retorne el mismo valor pero sin estar "hardcodeado").

*En la **siguiente entrega**, nos enfocaremos en los Test Unitarios con **PyUnit** y **PHPUnit**.*