

## Actividad Presencial #5 TDD desde las trincheras

*"La civilización no suprime la barbarie, la perfecciona"*  
Voltaire

### Objetivos:

- Identificar correctamente las pruebas unitarias según requerimientos.
- Sistematizar buenas prácticas en la escritura de pruebas
- Conocer herramientas para automatizar pruebas en PHP y Java

### Contenido:

- Introducción
- Caso de prueba BankAccount
- Caso de prueba Factorial
- Conclusiones.

### Bibliografía

Beck, K. (2011). *Test Driven Development*. EE UU: Addison Wesley.  
Cohn, M. (2010). *Succeeding with Agile*. EE UU: Addison Wesley.  
Freeman, S., & Pryce, N. (2010). *Object oriented software guided by test*. EE UU: Addison Wesley.  
Jurado, C. B. (2010). *Diseño ágil con TDD*. safeCreatives.  
Koskela, L. (2008). *Test driven*. Manning Publications.

### Introducción

Hacer pruebas de funcionamiento de los diversos componentes de un proyecto es una parte vital del ciclo de desarrollo, consumiéndose en esta tarea una gran cantidad de tiempo y recursos. Por esta causa se hace imprescindible buscar alternativas que permitan automatizar estas pruebas, por lo menos a nivel unitario (cada componente).

Actualmente existen una amplia gama de herramientas para estos en casi todos los lenguajes de programación. Por solo mencionar algunos ejemplos relacionado con software libre se pueden citar:

- PHPUnit para PHP
- RSpec para Ruby
- JUnit para Java

Para comprender mejor la filosofía de trabajo con TDD se han seleccionado dos de las herramientas más difundidas en el mundo de la programación web de la actualidad: PHPUnit y JUnit, que serán aplicadas en dos ejercicios prácticos durante el transcurso de la clase.

## Ejercicio práctico #1. Cuenta bancaria en PHP

Se pretende implementar dentro de un sistema para la gestión de cuentas bancarias, una clase principal, cuya responsabilidad será manipular los datos básicos de la cuenta. La clase BankAccount no solo tendrá los métodos get/set para gestionar el balance, sino que incluirá métodos para depósitos y extracciones de dinero. También el cliente especifica dos condicionales que deben cumplirse:

- El balance inicial de la cuenta debe ser cero
- El balance inicial de la cuenta no puede ser negativo.

Las pruebas para la clase BankAccount se escribirán antes que el código, utilizando los términos establecidos por el cliente en su lenguaje natural. En primer lugar se planificará el resultado de la prueba y luego se implementará la clase base como sigue a continuación:

```
<?php
require_once 'BankAccount.php';

class BankAccountTest extends PHPUnit_Framework_TestCase
{
    protected $ba;

    protected function setUp()
    {
        $this->ba = new BankAccount;
    }

    //Prueba del método para controlar el balance inicial de la cuenta

    public function testBalanceIsInitiallyZero()
    {
        $this->assertEquals(0, $this->ba->getBalance());
    }

    //Prueba del método para controlar que el balance no sea negativo luego de una extracción

    public function testBalanceCannotBecomeNegative()
    {
        try {
```

```

        $this->ba->withdrawMoney(1);
    }

    catch (BankAccountException $e) {
        $this->assertEquals(0, $this->ba->getBalance());

        return;
    }

    $this->fail();
}

//Prueba del método para controlar que el balance no sea negativo antes de un depósito

public function testBalanceCannotBecomeNegative2()
{
    try {
        $this->ba->depositMoney(-1);
    }

    catch (BankAccountException $e) {
        $this->assertEquals(0, $this->ba->getBalance());

        return;
    }

    $this->fail();
}
}
?>

```

Posteriormente se escribe la cantidad mínima de código para que pase la primera prueba, `testBalanceInitiallyZero()`. En este ejemplo solo se implementará el método `getBalance()` como sigue a continuación:

```

<?php
class BankAccount
{
    protected $balance = 0;

```

```
public function getBalance()
{
    return $this->balance;
}
?>
```

La prueba para la primera condición pasa en el caso anterior, pero para la segunda falla porque aún no se han implementado los métodos llamados por las pruebas.

phpunit BankAccountTest  
 PHPUnit 3.7.0 by Sebastian Bergmann.

.Fatal error: Call to undefined method BankAccount::withdrawMoney()

Para que la prueba de la segunda condición pase es necesario implementar los métodos withdrawMoney(), depositMoney(), y setBalance(), como se muestra a continuación. Dichos procedimientos están escritos de manera que lanzan una excepción personalizada en la clase BankAccountException, cuando son invocados con valores no permitidos que violan las condiciones establecidas.

```
<?php
class BankAccount
{
    protected $balance = 0;

    public function getBalance()
    {
        return $this->balance;
    }

    protected function setBalance($balance)
    {
        if ($balance >= 0) {
            $this->balance = $balance;
        } else {
            throw new BankAccountException;
        }
    }
}
```

```

    }
}

public function depositMoney($balance)
{
    $this->setBalance($this->getBalance() + $balance);

    return $this->getBalance();
}

public function withdrawMoney($balance)
{
    $this->setBalance($this->getBalance() - $balance);

    return $this->getBalance();
}
}
?>

```

Los resultados esperados deben mostrarse de la siguiente forma:

```

phpunit BankAccountTest
PHPUnit 3.7.0 by Sebastian Bergmann.
Time: 0 seconds
OK (3 tests, 3 assertions)

```

Alternativamente puede ser utilizado un método **assert** estático provisto por la clase PHPUnit\_Framework\_Assert, para escribir las afirmaciones (assert) de las condiciones establecidas con estilo *design-by-contract* (diseño por restricciones). Cuando una de estas afirmaciones falla se lanza una excepción de la clase PHPUnit\_Framework\_AssertionFailedError. Con esta aproximación se puede escribir menos código para chequear la condición y que la prueba escrita sea más legible. De cualquier forma, en el proyecto se añade en tiempo de ejecución una dependencia de PHPUnit.

```

<?php
class BankAccount
{
    private $balance = 0;

```

```

public function getBalance()
{
    return $this->balance;
}

protected function setBalance($balance)
{
    PHPUnit_Framework_Assert::assertTrue($balance >= 0);

    $this->balance = $balance;
}

public function depositMoney($amount)
{
    PHPUnit_Framework_Assert::assertTrue($amount >= 0);

    $this->setBalance($this->getBalance() + $amount);

    return $this->getBalance();
}

public function withdrawMoney($amount)
{
    PHPUnit_Framework_Assert::assertTrue($amount >= 0);
    PHPUnit_Framework_Assert::assertTrue($this->balance >= $amount);

    $this->setBalance($this->getBalance() - $amount);

    return $this->getBalance();
}
}
?>

```

Al escribir las pruebas para las restricciones establecidas en el problema, se puede visualizar al menos de forma básica, la forma que tendrá la clase BankAccount que se debe escribir para que estas pruebas pasen. No obstante, en el caso anterior, no se contemplaron pruebas que invocaran a los métodos setBalance(), depositMoney(), and withdrawMoney() con valores correctos que no violaran las restricciones impuestas. Se debe señalar que una batería de pruebas debe contener

comprobaciones tanto positivas como negativas para abarcar la mayor cantidad de escenarios posibles en los que correrá cada método. Esto se verá con mayor detalle en próximas clases relacionadas con Desarrollo Guiado por Comportamientos (BDD).

## Ejercicio práctico #2. Factorial en Java

JUnit es un framework desarrollado para la realización de tests unitarios en Java. Tiene como principal característica poder definir Tests de forma rápida y sencilla a través de la utilización de etiquetas. El ejercicio que se verá a continuación se basa en dicho framework y consiste en implementar una clase cuya responsabilidad sea calcular el factorial de un número entero, como se muestra en los siguientes ejemplos:

$0! = 1$

$1! = 1$

$2! = 2 \times 1 = 2$

$3! = 3 \times 2 \times 1 = 6$

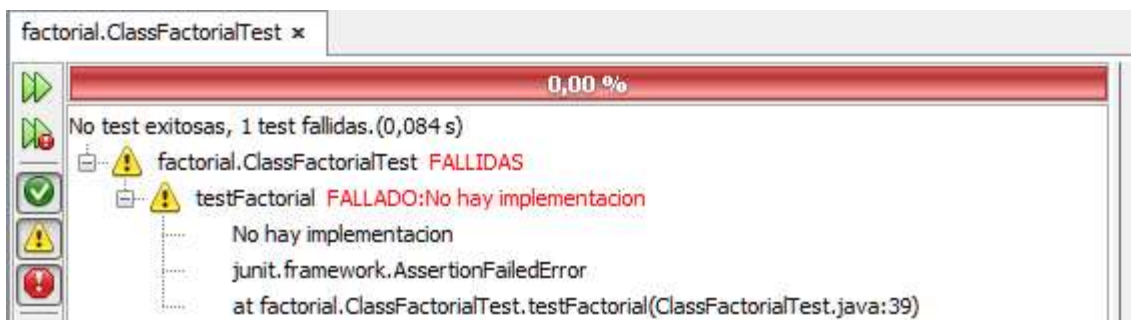
Este ejemplo se enfocará en el uso del IDE Netbeans que brinda la posibilidad de utilizar el framework JUnit pero además el framework TestNG basado en el primero pero con más funcionalidades. Primero se crea una clase base de prueba como sigue:

```
15  /*
16  public class ClassFactorialTest {
17
18  public ClassFactorialTest() {
19  }
20  @BeforeClass
21  public static void setUpClass() throws Exception {
22  }
23  @AfterClass
24  public static void tearDownClass() throws Exception {
25  }
26  /**
27   * Test of factorial method, of class ClassFactorial.
28   */
29  @Test
30  public void testFactorial() {
31      assertEquals(1, ClassFactorial.factorial(0));
32      assertTrue(1 == ClassFactorial.factorial(1));
33      assertEquals(24, ClassFactorial.factorial(4));
34      assertEquals(720, ClassFactorial.factorial(6));
35      boolean excepcionLanzada = false;
36      try {
37          ClassFactorial.factorial(-1);
38      } catch (IllegalArgumentException e) {
39          excepcionLanzada = true;
40      }
41      assertTrue(excepcionLanzada);
42  }
43  }
```

Como se puede apreciar los métodos setUp() y tearDown() se sustituyen por anotaciones @BeforeClass y @AfterClass, propias del lenguaje contenidas en los frameworks de prueba. Si se ejecutara la prueba en este momento no arrojará resultados positivos por ni siquiera la clase está implementada, por lo cual se escribe el código básico para que se pueda ejecutar la prueba:

```
10  * 1ero Definimos la clase para luego implementar la funcionalidad
11  */
12  public class ClassFactorial {
13
14      public static int factorial(int n) throws IllegalArgumentException {
15          return 0;
16          //Falta implementar funcionalidad
17      }
18  }
19
```

Aunque la prueba todavía no pase, se puede comenzar a implementar a partir de la visión que se obtiene del método de prueba implementado. El resultado que se observa en el la zona inferior del IDE se muestra a continuación:



Para que pase la prueba se implementa el método en cuestión como sigue:

```
10  * 3ro Desarrollar la funcionalidad
11  */
12  public class ClassFactorial {
13      public static int factorial(int n) throws IllegalArgumentException {
14          int factorial = 1;
15          if (n < 0) {
16              throw new IllegalArgumentException();
17          }
18          if (n > 0) {
19              factorial = n * factorial(n - 1);
20          }
21          return factorial;
22      }
23  }
24
```

La prueba pasa entonces y la barra ahora se muestra en verde:





Una vez que se ejecuta la prueba con éxito, se comprueba que el método funciona como se esperaba. Ahora, tras cualquier cambio en el código se ejecuta nuevamente JUnit para comprobar que no se ha “roto” nada. En vista de que el calcular el factorial de manera recursiva es ineficiente porque consume mucha memoria, se puede pensar en implementarlo de forma iterativa que consume menos y de luego de esta refactorización se ejecuta el SUT nuevamente para comprobar su funcionamiento.

```
10  * 5to Refactorizando
11  */
12  public class ClassFactorial {
13      public static int factorial(int n) throws IllegalArgumentException {
14          int factorial = 1;
15          if (n < 0) {
16              throw new IllegalArgumentException();
17          }
18          if (n > 0) {
19              /**factorial = n * factorial(n - 1);
20               * Aplicaremos el calculo del factorial de manera iterativa
21               en vista de que consume menos memoria*/
22              for(int i = 2; i<=n; i++)
23                  factorial*=i;
24          }
25          return factorial;
26      }
27  }
```

## Conclusiones

Durante el desarrollo de la aplicación pueden surgir numerosos casos de prueba, en tanto se añaden o modifican funcionalidades, ya que desde el principio es prácticamente imposible visualizar todos los escenarios posibles. Es importante y condición fundamental del programador saber parar en un punto en que se abarquen los casos de prueba fundamentales para cada funcionalidad, de lo contrario el desarrollo de la aplicación sería infinito.

TDD, implica pensar en la mayoría de las situaciones posibles y a medida que se escriben las pruebas se comprende mejor el problema. Aunque es una ayuda imprescindible para escribir mejor código, utilizar TDD no es garantía de escribir sin errores (bugs).

TDD es solo la punta del iceberg en la especialidad de pruebas de software. Además, es el cimiento de otras metodologías que se concentran más en las pruebas de unidad que incluyen los comportamientos de interfaz. En próximas clases, se verán los fundamentos de BDD, que terminará de demostrar la necesidad de entregar software con aval de pruebas automatizadas.