



Estructura de Datos y Algoritmos

ITBA 2024-Q2

Test-Driven Development

- TDD es una metodología que comienza por los tests y luego pasa a la implementación.
- Permite enfocarse en la definición de la interfaz y del comportamiento y no en los detalles internos de implementación.
- Ve al sistema que se va a desarrollar como una caja negra ya que todavía no existe!
- Apunta a que todas las funcionalidades tengan algún test que las controle y defina.

Unit Testing

- Testear pequeñas unidades del código.
- Normalmente una clase o de una función aisladas.
- Corren automáticamente.
- Pueden ser ejecutados cada vez que se hacen cambios para comprobar que la funcionalidad anterior siga funcionando correctamente.
- Son esenciales para las metodologías Ágiles y para realizar Refactoring de código.

Unit Test - Características

- Automático
- Verifican un único caso por test
- Repetible
- Independientes de otros tests o de condiciones externas
- Mantenible y Documentado (comentado)
- Ejecuta en muy poco tiempo (muy deseable)

¿Qué testear en un Unit Test?

- En el caso de un método o función:
 - Casos Típicos
 - Casos de Borde
 - Casos de Error
 - Casos de Excepción
- En el caso de una clase:
 - Secuencias de llamadas válidas
 - Secuencias de llamadas inválidas
 - Chequeo de invariantes

JUnit 5

Introducción



JUnit

- Es un framework para realizar casos de prueba en aplicaciones Java
- Se pueden comparar resultados de las invocaciones de métodos con los valores esperados, o verificar si una excepción fue lanzada o no.
- Un caso de prueba es abortado ni bien falla alguna verificación o se lanza una excepción no esperada

<https://junit.org/junit5/>

Comparando resultados

- El **test unitario** más simple consiste en comparar el resultado obtenido con el resultado esperado.
- Para ello, se pueden utilizar los siguientes métodos estáticos:

`Assertions.assertEquals(valorEsperado, valorObtenido)`

`Assertions.assertTrue(valorObtenido)`

...

- Si un método lanza una excepción, el mismo se considera que falló. Para aquellos casos en que se espera que se lance esta excepción, se indica de la siguiente manera:

`Assertions.assertThrows(RuntimeException.class, () -> ...);`

Ejemplo

```
import org.junit.jupiter.api.DisplayName;
import org.junit.jupiter.api.Test;
import static org.junit.jupiter.api.Assertions.*;

class TimerFromScratchTest {
    @Test
    @DisplayName("Probar si el lapso de tiempo es correcto.")
    void getDurationTest() {
        TimerFromScratch timer = new TimerFromScratch();
        long expected = 2000; //ms
        long result = timer.getDuration( start: 0, stop: 2000);
        assertEquals(expected, result);
    }
}
```

Ejemplo

```
@Test
@DisplayName("Probar excepcion lanzado por: stop < start")
void getDurationExceptionTest() {
    TimerFromScratch timer = new TimerFromScratch();
    assertThrows(RuntimeException.class,
        ()->timer.getDuration( start: 2000, stop: 1000));
}
```

Ambiente de prueba

- Frecuentemente se desea tener un ambiente de prueba prefijado, por ejemplo con ciertas variables inicializadas.
- Para evitar repetir este código de inicialización en cada uno de los tests unitarios (en cada uno de los métodos `@Test`) se cuentan con varias anotaciones útiles:
 - `@BeforeAll`: Se ejecutará antes de todos los casos de prueba de la clase
 - `@BeforeEach`: Se ejecutará antes de cada `@Test`
 - `@AfterEach`: Se ejecutará después de cada `@Test`
 - `@AfterAll`: Se ejecutará después de todos los casos de prueba de la clase
 - y otras más

Ejemplo

```
@BeforeAll
static void initAll() {
    System.out.println("Empiezan los tests");
}
```

```
@BeforeEach
void init() {
    System.out.println("Empieza un test");
}
```

```
@AfterEach
void tearDown() {
    System.out.println("Termina un test");
}
```

```
@AfterAll
static void tearDownAll() {
    System.out.println("Terminaron todos los tests");
}
```

Ejemplo

```
class TimerFromScratchTest {  
    3 usages  
    TimerFromScratch timer;  
    @BeforeEach  
    void intanceTimer(){  
        this.timer = new TimerFromScratch();  
    }  
  
    @Test  
    @DisplayName("Probar si el lapso de tiempo es correcto.")  
    void getDurationTest(){  
        //TimerFromScratch timer = new TimerFromScratch();  
        long expected = 2000; //ms  
        long result = timer.getDuration( start: 0, stop: 2000);  
        assertEquals(expected,result);  
    }  
}
```

TP 1 – Ejer 7

Implementación de tests
unitarios para la versión

TimerFromScratch

usando Junit

Diseñar testeos!!!

Agregar plug-in y
dependencias

```
<dependencies>
```

```
  <dependency>
```

```
    <groupId>org.junit.jupiter</groupId>
```

```
    <artifactId>junit-jupiter-engine</artifactId>
```

```
    <version>5.8.0-M1</version>
```

```
    <scope>test</scope>
```

```
  </dependency>
```

```
</dependencies>
```

Y

```
<plugin>
```


```
  <groupId>org.apache.maven.plugins</groupId>
```

```
  <artifactId>maven-surefire-plugin</artifactId>
```

```
  <version>2.22.2</version>
```

```
</plugin>
```





Sólo con método toString() es complicado saber si la clase está generando bien los días, horas, minutos, segundos a partir de los ms...

¿Cómo sabremos cuál es el tiempo transcurrido para poder chequear si lo calculado es lo esperado?



Una Clase es **Testeable** si fue diseñada para poder realizar correctamente tests sobre ella.


Los métodos no deben tener efectos secundarios ni dependencias con componentes externos

TP 1- Ejer 8

Mejorar la implementación de
TimerFromScratch



Agregar métodos getters para
facilitar el testing.

- 
- ¿Validaron que no se obtengan 61 minutos? Etc
 - ¿Probaron si falla con `new Timer()` y ejecutamos con `stop()` anterior?



¿Qué es el coverage de un testeo de unidad?

<https://www.jetbrains.com/help/idea/running-test-with-coverage.html>