

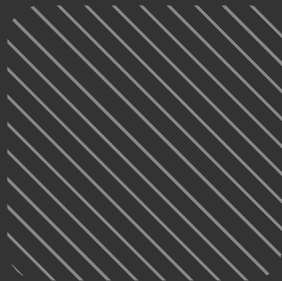
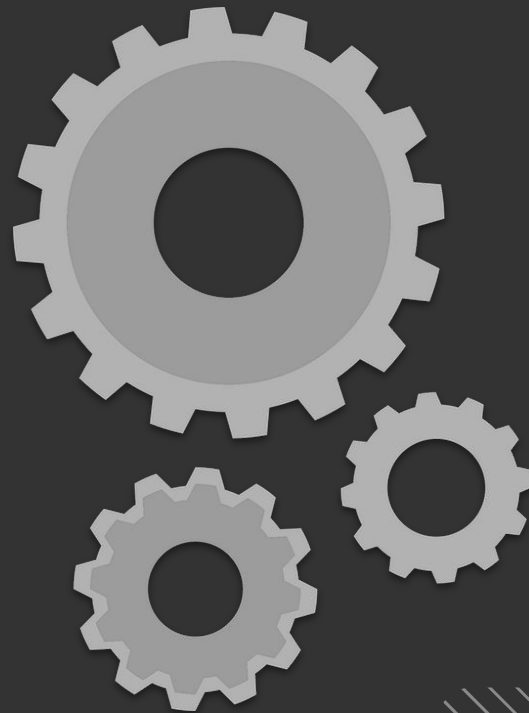
# Testing:

// Unit Test sin Mocks

// Unit Test con Mocks

IT BOARDING

**BOOTCAMP**



# Índice



**01**

Crear Código  
Testeable

**02**

Introducción a API Testing

**03**

Unit Test  
(Pruebas Unitarias)

**04**

Double or Fake

IT BOARDING

**BOOTCAMP**



## Spoiler Alert!

En esta clase vamos a aprender qué es un test, por qué es importante, cómo pensarlo, diseñarlo, construirlo, y sobre todo cómo saber si escribimos los suficientes.

Pero no vamos a enfocarnos en cómo hacerlos para ayudarnos a ejecutarlos cada vez que queremos agregar un cambio a nuestro repo, y cómo a través de la aplicación de prácticas como **CI/CD** nos ayuda a construir código de calidad y seguro.

Eso lo vamos a ver en 1 semana, en el **módulo de Quality**, por eso es importante que hoy se enfoquen en aprender a fondo qué es un test y cómo hacer los mejores tests :).



TESTING

# // Crear Código Testeable

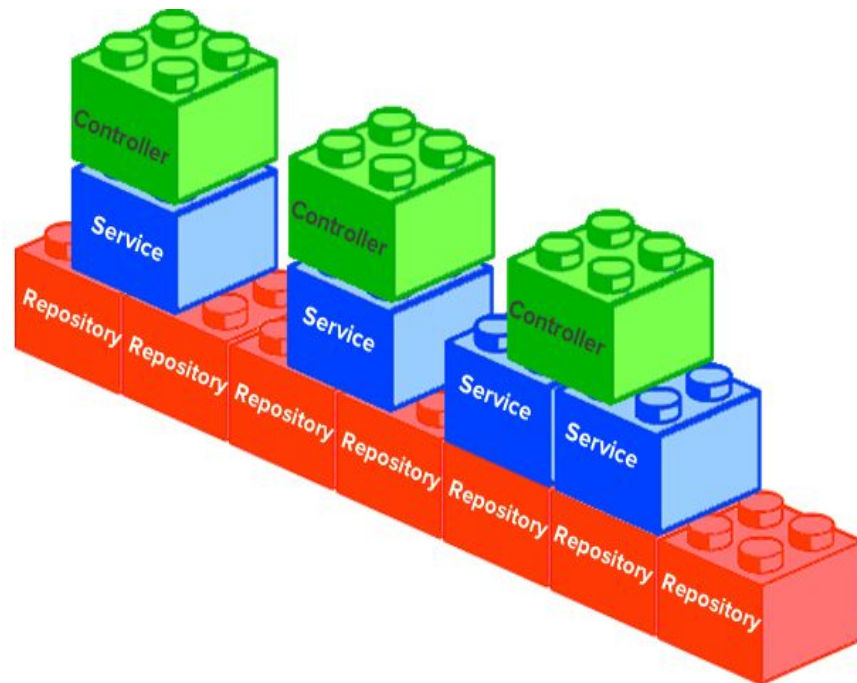
IT BOARDING

**BOOTCAMP**

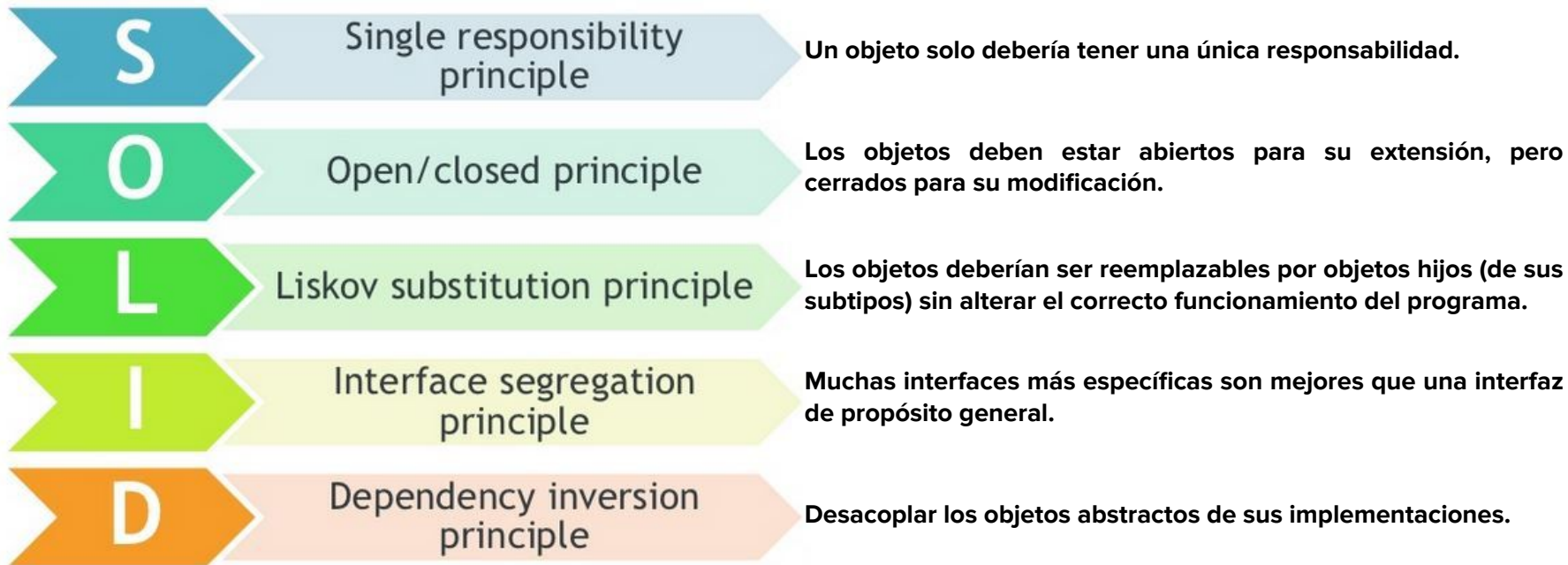


## ¿Que es un componente Testeable?

- Aquellos componentes que forman parte de la API que cumplan con los principios **S.O.L.I.D.**
- Hay componentes que no son necesario testear.
- Centrarse en componentes que posean funcionalidad:
  - Servicios
  - Controladores
  - Repositorios
  - Clases con funciones de Cálculo



## Principio S.O.L.I.D



TESTING

# // Introducción a API de Testing

IT BOARDING

**BOOTCAMP**

## Lenguajes, librerías y frameworks para Testing

Si bien vamos a centrarnos en el lenguaje de **Java** y el framework de test **JUnit**, es bueno saber que todos los lenguajes de programación modernos poseen uno o varios frameworks de testing.







## Tipos de Test y cuales estudiaremos

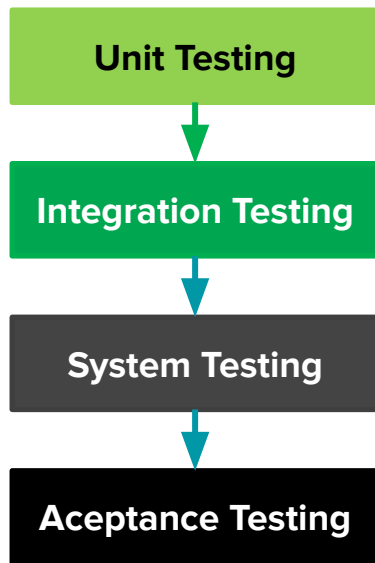
**01** | Pruebas Unitarias (Unit Test).

**02** | Pruebas de Integración

**03** | Pruebas de Funcionamiento

**04** | Pruebas de Aceptación

**05** | Pruebas de Estrés

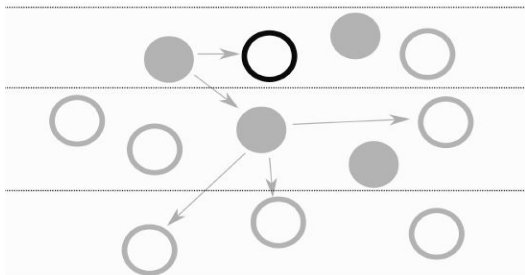


## Unit Test (Prueba de Unidad)

Validar que cada **unidad** del software funcione como se espera.

Toma una pieza testeable del código y prueba algunos supuestos sobre el comportamiento lógico de ese método o clase en **aislamiento**.

Cualquier **dependencia** del módulo bajo prueba debe sustituirse por un **mock** o un **stub**, para acotar la prueba específicamente a esa unidad de código.

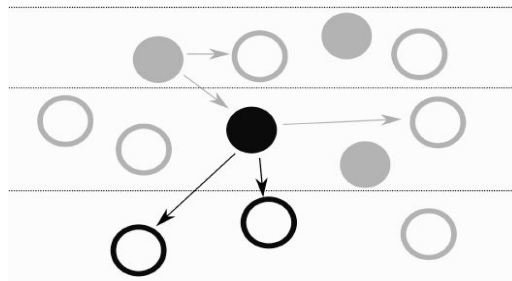


## Integration Test (Prueba de Integración)

Validar la **interacción** de módulos de software dependientes entre sí probandolos en **conjunto**.

Cubren un área mayor de código, del que a veces no tenemos control (como librerías de terceras partes), o una conexión a una base de datos, o a otro web service.

Corren más lento y suelen ser el paso siguiente a los tests unitarios.

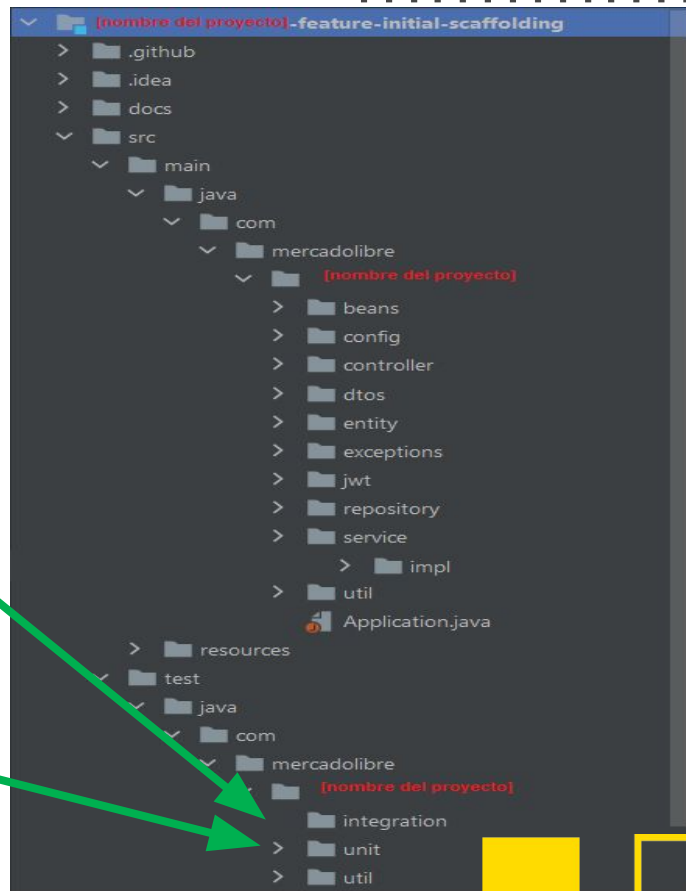


## Ubicación de los Test Unitario y de Integración

- Respetar la estructura del proyecto
- Los test se ubican dentro del paquete test

Crear un paquete “**integration**”, dentro de la ruta de test. Se pueden crear sub paquetes para organizarlos mejor

Crear un paquete “**unit**”, dentro de la ruta de test. Se pueden crear sub paquetes para organizarlos mejor.

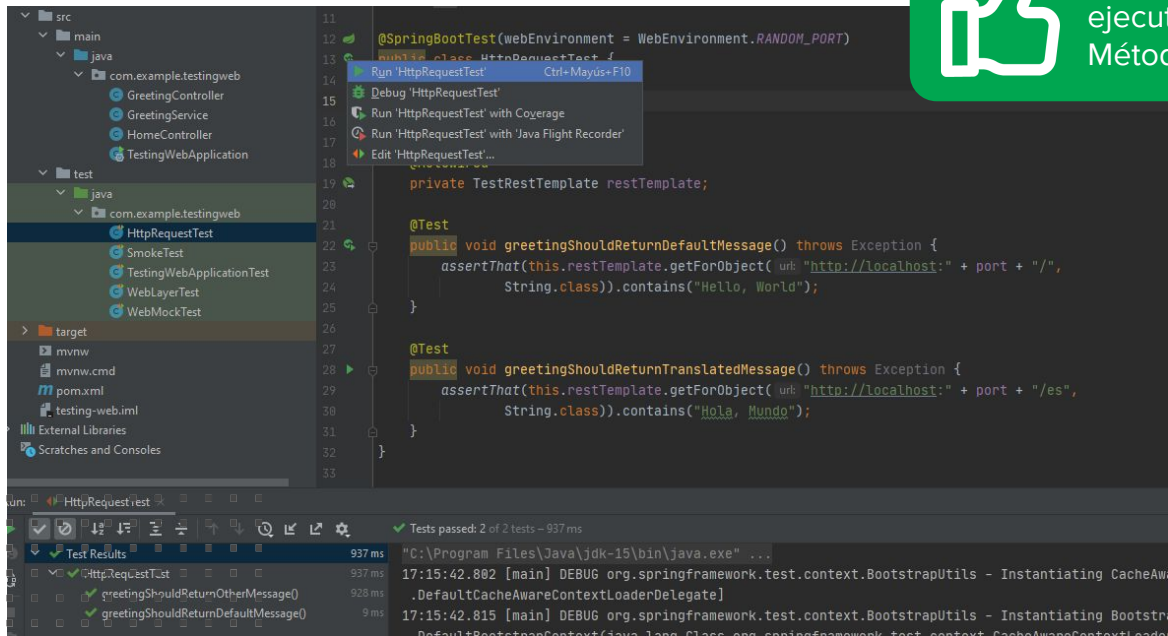


## Ejecutando Tests en IntelliJ

Al hacer click sobre el ícono de Test me muestra las opciones de ejecución. Las más importantes son “Run” para una ejecución de corrido y “Debug” para una ejecución evaluativa.



Si utilizo el ícono de la Clase se ejecutarán todos los Tests, si uso el del Método ejecutará solo el Test indicado.





## Consola de Ejecución de Tests

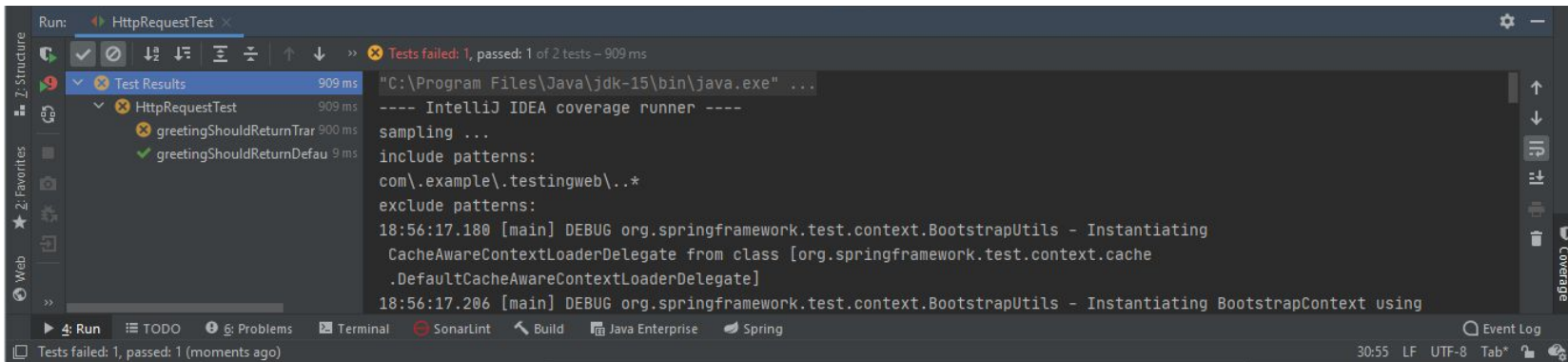
Nos muestra el resultado de la ejecución de los tests, indicando con un tilde verde los que pasaron y con una cruz amarilla los que fallaron. Además nos informa **la causa** en el caso de los que no pasaron y nos permite reiterar la ejecución.



### Resultados al ejecutar un Test

✓ greetingShouldReturnDefaultMessage()  
✗ greetingShouldReturnTranslatedMessage()

```
java.lang.AssertionError:  
Expecting:  
  <"Hola, Mundo">  
to contain:  
  <"Hola, Planeta">
```



TESTING

# // Unit Test (Pruebas Unitarias)

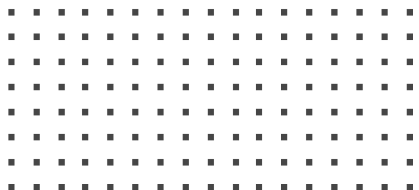
IT BOARDING

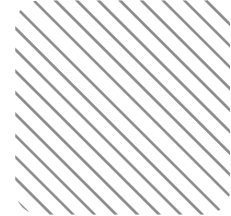
**BOOTCAMP**



## Beneficios de los Test Unitarios

- **Facilitar los cambios en el código** al detectar modificaciones que pueden romper el contrato en el caso de refactorizaciones. Es más fácil hacer un cambio y probar instantáneamente si está afectando alguna funcionalidad.
- **Encontrar bugs** probando componentes individuales antes de la integración, así los problemas pueden ser solucionados antes de que impacten otras partes del código. Reducen el tiempo de debugging.
- **Proveen documentación**, ayudan a comprender qué hace el código y cuál fue la intención al desarrollarlo.
- **Mejoran el diseño y la calidad del código** invitando al desarrollador a pensar en el diseño del mismo, antes de escribirlo (**Test Driven Development - TDD**).





## Principios F.I.R.S.T

Los Test Unitarios deben de cumplir con los principios F.I.R.S.T

|          |                             |                                                                                                                                                                                                                                                                 |
|----------|-----------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>F</b> | <b>Fast</b>                 | <b>Rápidos:</b><br>Es posible tener miles de tests en tu proyecto y deben ser rápidos de correr.                                                                                                                                                                |
| <b>I</b> | <b>Isolated/Independent</b> | <b>Aislados/Independientes:</b><br>Un método de test debe cumplir con los “ <b>3A</b> ” ( <b>Arrange, Act, Assert</b> ) o lo que es lo mismo: <b>Given, when, then</b> . Además no debe ser necesario que sean corridos en un determinado orden para funcionar. |
| <b>R</b> | <b>Repeatable</b>           | <b>Repetibles:</b><br>Resultados determinísticos. No deben depender de datos del ambiente mientras están corriendo (por ejemplo: la hora del sistema).                                                                                                          |
| <b>S</b> | <b>Self-Validating</b>      | <b>Auto-Validados:</b><br>No debe ser requerida una inspección manual para validar los resultados.                                                                                                                                                              |
| <b>T</b> | <b>Thorough</b>             | <b>Completos:</b><br>Deben cubrir cada escenario de un caso de uso, y no sólo buscar un coverage del 100%. Probar mutaciones, edge cases, excepciones, errores,                                                                                                 |





## JUnit

Es el framework open-source de testing para Java más usado, de él nos servimos para escribir y ejecutar tests automatizados (<http://junit.org>).

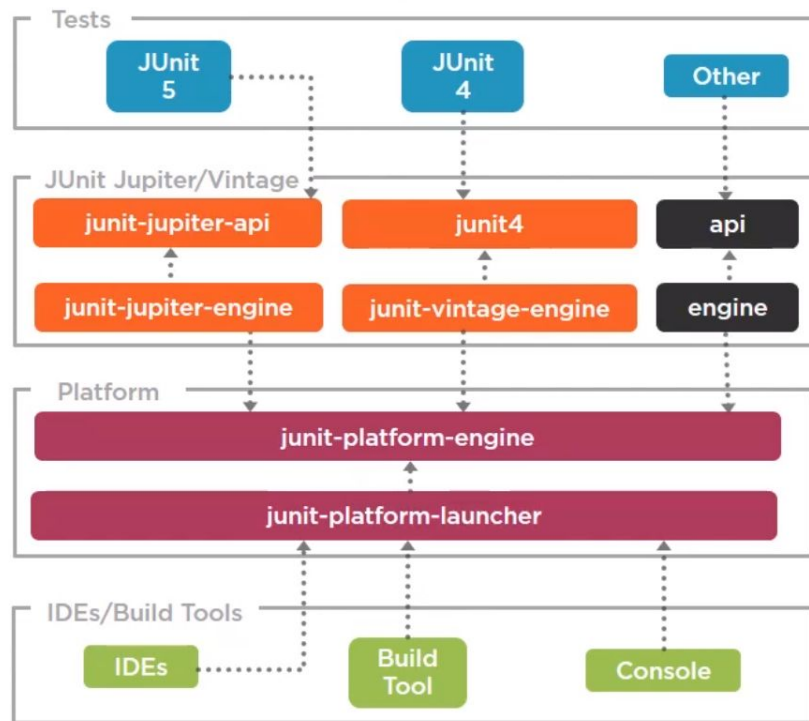
Es soportado por todas las IDEs (Eclipse, IntelliJ IDEA), build tools (Maven, Gradle) y por frameworks como Spring.

## Arquitectura de JUnit 5

**JUnit Platform:** Descubrir y ejecutar tests. La platform-launcher es usada por las IDEs y los build tools.

**JUnit Jupiter:** Api para escribir tests, motor para ejecutarlos.

**JUnit Vintage:** contiene el motor de Junit 3 y 4 para correr tests escritos en estas versiones.



## Escribiendo Unit Tests 1/5

### Spring Boot Starter Test

Existe un módulo específico de testing de **spring-boot: *spring-boot-starter-test*** que nos provee de algunas herramientas útiles para testing:

Incluye las librerías **Mockito**, **Hamcrest**, **Spring-test**, **junit**, y algunas clases de apoyo para testing.

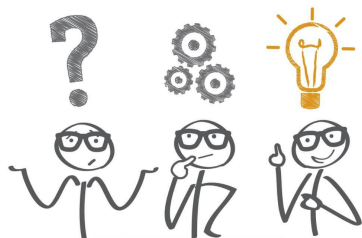
```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-test</artifactId>
  <scope>test</scope>
</dependency>
```

## Escribiendo Unit Tests 2/5

### Diseñar los escenarios de testeo a realizar

Dada una clase `Calculator` que cumple con los principios **S.O.L.I.D.** con un método `add()`, se desea escribir un test unitario para comprobar que este objeto se comporte de la forma esperada.

- Se realizan solo los Casos Borde.
- El nombre del test debe ser declarativo.



```
public class Calculator {  
    public Integer add(Integer a, Integer b) {  
        return a + b;  
    }  
}
```


```
class CalculatorTest {  
  
    @Test  
    void testAddOneNullNumber() {}  
  
    @Test  
    void testAddTwoPositiveNumbers() {}  
  
    @Test  
    void testAddOnePositiveOneNegative() {}  
  
    @Test  
    void testAddTwoNegativeNumbers() {}  
  
}
```

## Escribiendo Unit Tests 3/5

### Escribir los 3 pasos de un test

Cualquier método de testeo debe cumplir el principio **F.I.R.S.T.**

1. **«Arrange»** que es donde se describe el escenario inicial. Se crean las **precondiciones y estado inicial**.
2. **«Act»** es donde se hace la llamada efectiva al **método a ser testeado**. En este caso es `add()`.
3. **«Assert»** donde se compara el **resultado esperado** con el **resultado obtenido** una vez tomadas las acciones y bajo determinadas condiciones.



```
public class CalculatorTest {  
  
    @Test  
    public void shouldAddTwoPositiveNumbers() {  
        //arrange  
        Integer expected = 2;  
        Calculator calculator = new Calculator();  
  
        //act  
        Integer sum = calculator.add(1, 1);  
  
        //assert  
        assertEquals(expected, sum);  
    }  
}
```



Un buen test debería tener  
un solo set de Act/Assert



## Escribiendo Unit Tests 4/5

### Anotaciones en JUnit

**@Test:** Indica a Junit que el método es un test y debe ejecutarse.

**@ParameterizedTest:** Permite correr el test con múltiples argumentos. Puede tomar los parámetros de diferentes fuentes, como de otro método, de valores o un archivo csv.

**@Disable:** Deshabilita un test para que no se ejecute, un test anotado así, será ignorado.

**@Tag:** Permite lanzar conjuntos de test en función de las etiquetas especificadas.

Anotaciones de **ciclo de vida:** Sirven para establecer los fixtures. Pueden ser de método o de clase.

**@BeforeEach:** Ejecuta un método antes de la ejecución de **cada test**.

**@AfterEach:** Ejecuta un método después de la ejecución de **cada test**.

**@BeforeAll:** Ejecuta un método antes de la ejecución de todos los test **de la clase**.

**@AfterAll:** Ejecuta un método después de la ejecución de todos los test **de la clase**.



## Escribiendo Unit Tests 5/5

### Assertions en JUnit

|                           |                      |
|---------------------------|----------------------|
| assertAll                 | assertThrows         |
| assertArrayEquals         | assertIterableEquals |
| assertEquals              | assertNotEquals      |
| assertTrue                | assertFalse          |
| assertTimeoutPreemptively | assertTimeout        |
| assertNull                | assertNotNull        |
| assertLineMatch           | assertNotSame        |

```
assertEquals(4, Calculator.add(2, 2));
assertNotEquals(3, Calculator.add(2, 2));
assertNull(null);
assertNotNull("hola mundo");
assertNotSame(originalObject, otherObject);
assertTrue(trueBool);
assertFalse(falseBool);
```

TESTING

# // Double or Fake (Doble o Impostor)

IT BOARDING

**BOOTCAMP**



## Dobles

Se utilizan para simular a los componentes originales y así poder realizar una prueba de forma aislada.

### Tipos de Dobles o Impostores

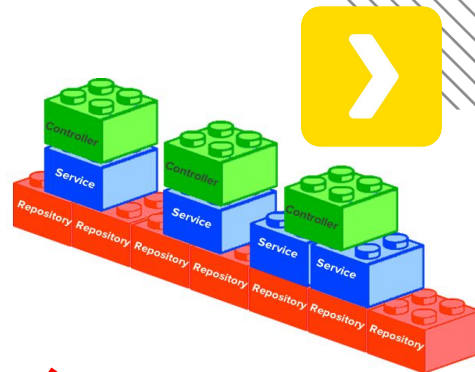
| Double or Fake | CONCEPTO                                                                                                                                                                                                                                                                        | Implementación | ¿puede fallar? |
|----------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|----------------|----------------|
| <b>MOCK</b>    | Cuando se necesitan verificar los resultados y las interacciones se utiliza un <b>mock</b> .<br>Se puede probar si un método ha sido llamado.<br>Se puede probar cuántas veces ha sido llamado un método.<br>Igual que los <i>stub</i> puede pre programarse su comportamiento. | Dinámica       | SI             |
| <b>STUB</b>    | Si solo se necesita verificar los resultados se suele utilizar un <b>stub</b> .<br>Reemplaza de forma controlada una dependencia o controlador.<br>Se reprograman sus valores de retorno, los cuales serán constantes                                                           | Mínima         | NO             |
| <b>SPY</b>     | Solo un subconjunto de métodos son <b>fake</b> .<br>A menos que sean explícitamente mockeados, el resto de métodos son los reales.                                                                                                                                              | Dinámica       | SI             |



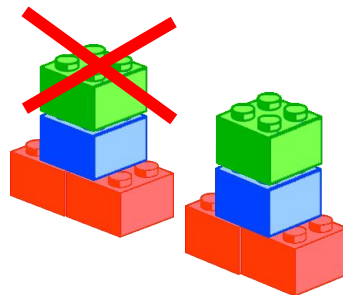
## Testear componentes usando DOBLE or FAKE

1. Identificar los componentes que van a ser testeados:
  - Controllers
  - Services
  - Repositories
2. Seleccionar **un solo componente** a ser testado.
  - Solamente interesa de quien depende inferiormente.
  - En este caso se analiza **un service** que posee **una dependencia a un repository**.
3. Reemplazar cada dependencia por un Doble or Fake, y comenzar a testear

1



2



3





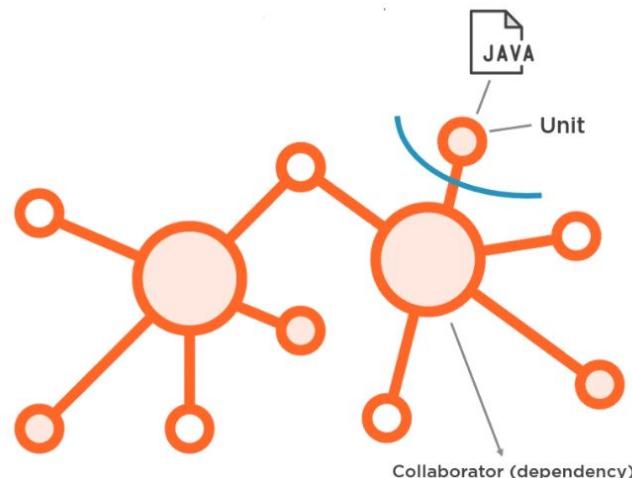
## ¿Por qué usar Mocks?

Permite razonar sobre una **unidad de código aislada**, sin tener que preocuparse por sus dependencias.

Es posible aislar a la clase de sus colaboradores, **reemplazando las dependencias por mocks**, y testeando todas las funcionalidades de esa unidad.

También es útil para mockear una dependencia que aún no fue creada y está en proceso de desarrollo.

¿Qué colaboradores suelen mockearse? Repositorios, Servicios, Librerías externas.



## Mockito

Es el framework de mocks más conocido del mundo java. Permite crear y configurar objetos mock. <http://mockito.org>

Otro framework bastante utilizado es **PowerMock**. <http://code.google.com/p/powermock/> ofrece la posibilidad de mockear métodos estáticos, entre otras funcionalidades.



## Escribiendo un test con Mocks

Anotar la clase con `@ExtendWith(MockitoExtension.class)` para inicializar los Mocks e inyectarlos en donde se indique.

Mockear las **dependencias** de la clase.

Injectar la dependencia Mockeada.

Dentro del test **definir el comportamiento** del mock: `when(methodCall).thenReturn(result)`.

**Ejecutar el método** de la clase siendo testeada.

**Verificar** que el método haya sido llamado y **retorne** los valores que esperábamos.

```
@ExtendWith(MockitoExtension.class)
public class UserServiceTest {

    @Mock
    private UserRepository userRepo;

    @InjectMocks
    private UserService userService;

    @Test
    void testFindAllUsers () {
        List<UserDTO> expectedUsers = createUsersList ();

        when (userRepo.findAll ()) .thenReturn (expectedUsers );

        List<UserDTO> currentUsers = userService.getAllUsers ();

        verify (userRepo, atLeast (1)).findAll ();
        assertThat (expectedUsers ).isEqualTo (currentUsers );
    }
}
```



**BUENA PRÁCTICA:**  
No debería haber más de un mock por test.



# Gracias.

IT BOARDING

**BOOTCAMP**

