Quality

Índice

INTRODUCCIÓN TEST UNITARIOS

Lenguajes de programación y frameworks de Testing

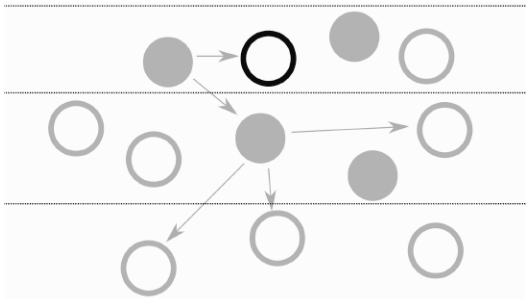
Si bien vamos a centrarnos en el lenguaje de Java y el framework de test JUnit, es bueno saber que todos los lenguajes de programación modernos poseen uno o varios frameworks de testing.

Test Unitarios

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

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.



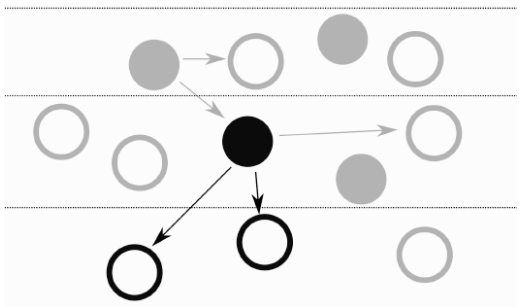
Alcance de un test unitario

Test de Integración

Validar la **interacción** de módulos de software dependientes entre sí probándolos 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.



Alcance de un test de integración

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 facil 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 Developement - TDD).

El principio F.I.R.S.T

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

¿QUÉ ES JUnit?

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).

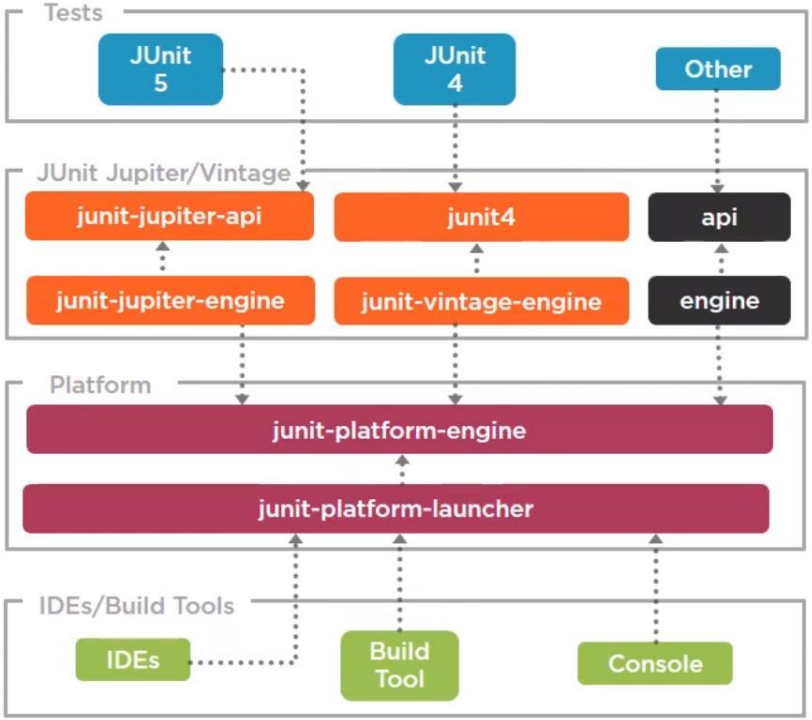
Es soportado por todas las IDEs (Eclipse, IntelliJ IDEA), build tools (Maven, Gradle) y por frameworks como Spring.

Arquitectura de Junit 5

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

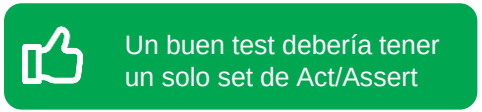
JUnit Jupiter: Api para escribir tests, motor para ejecutarlos.

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



¿Cómo escribir un Test Unitario?

Tenemos una clase Calculator con un método add() y queremos escribir un test unitario para comprobar que este método esté haciendo lo que esperamos de él.




```java
public class Calculator {

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

Creamos un método para comprobar que nuestra calculadora esté sumando dos enteros correctamente.

En primer lugar debemos escribir el **«Arrange»** que es dónde describimos las precondiciones y estado inicial. También se lo conoce como **test fixture.**

En segundo lugar agregamos el **«Act»,** es decir qué método se ejecutará para ser probado. En este caso es add().

Finalmente el **«Assert»** que describe el resultado esperado una vez tomadas las acciones y bajo determinadas condiciones.

En este caso asertamos que la variable *expected* es igual a la variable *sum* que es el resultado de ejecutar el método add().

```java
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);
    }
}
```

Anotaciones en JUnit

@Test: Es necesario anotar cada método para que Junit lo reconozca como un test y lo ejecute. (org.junit.jupiter.api.Test)

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

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

@Tag: Permite lanzar conjuntos de test en función de las etiquetas que especifiquemos.

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.

```
@BeforeEach
void setUp() {
    Calculator calculator = new Calculator();
    // calculator.setPropertiesIfNeeded();
}
```

Assertions en JUnit

JUnit provee una gran variedad de assertions que se encuentran ubicadas en org.junit.jupirter.api.Assertions.

También es posible utilizar librerías de Assertions externas como por ejemplo **AssertJ** o **Hamcrest.**

assertAll	assertNotSame
assertArrayEquals	assertNull
assertEquals	assertSame
assertFalse	assertThrows
assertIterableEquals	assertTimeout
assertLinesMatch	assertTimeoutPreemptively
assertNotEquals	assertTrue
assertNotNull	fail

```
assertEquals(4, Calculator.add(2, 2));

assertNotEquals(3, Calculator.add(2, 2));

assertNull(null);

assertNotNull("hola mundo");

assertNotSame(originalObject, otherObject);

assertTrue(trueBool);

assertFalse(falseBool);
```

TEST DOUBLES Y MOCKS

Dobles:

Usualmente la funcionalidad de un **SUT (System Under Test)** depende de otros componentes.

Stub	Mock
Contiene una mínima implementación.	Contiene una implementación dinámica.
Suele retornar valores constantes.	Puede ser configurado con un comportamiento específico.
Es un reemplazo controlado para una dependencia o colaborador en el sistema.	Es un objeto fake que decide si el unit test pasa o falla. Lo hace verificando si el SUT ha interactuado como es esperado con el objeto fake.
No pude fallar un test.	Puede fallar un test.

Mockito

Mockito http://mockito.org, es el framework de mocks más conocido del mundo java. Permite crear y configurar objetos mock.

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

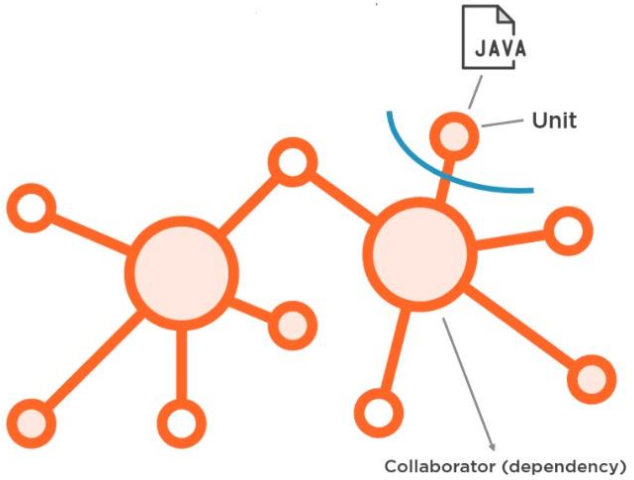
¿Por qué usar Mocks?

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.



Escribiendo un test con Mocks

Mockear las **dependencias** de la clase siendo testeada con **@Mock.**

En el setUp agregar el método **initMocks(this)** para que se inicialicen los mocks.

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.

No debería haber más de un mock por test.

```java
public class UserServiceTest {

    @Mock
    private UserRepository userRepository;

    UserService userService;

    @BeforeEach
    void setUp() {
        initMocks(this);
        userService = new UserServiceImpl(userRepository, emailService);
    }

    @Test
    void shouldGetAllUsers() {
        List<User> users = createUsers();

        when(userRepository.findAll()).thenReturn(users);

        List<User> returnedUsers = userService.getAllUsers();

        verify(userRepository, atLeast(1)).findAll();
        assertThat(returnedUsers).isEqualTo(users);
    }
}
```

¡Gracias! ☺