Pruebas unitarias

Índice

JUNIT5

Objetivos

- Facilitar la **escritura** de pruebas útiles.
- Facilitar la creación de pruebas que mantengan su **valor** a lo largo del tiempo.
- Reducir el coste de escribir pruebas mediante la **reutilización de código.**

Características básicas

- Algunas de las **características básicas** de JUnit son

– Las pruebas se implementan como métodos (**métodos de prueba**).

– Cada prueba de unidad se ejecuta usando instancias de **clases independientes.**

– Anotaciones para facilitar el manejo del **contexto** (*fixture*) de las pruebas de unidad: @Before, @BeforeClass, @After y @AfterClass

– Un conjunto de métodos para hacer **aserciones** sobre el código.

Elementos principales

- **Pruebas**

– Se definen anotando métodos con @Test

– Por lo tanto, cada prueba será un método.

– Cada prueba debería probar una “unidad” (normalmente un método).

Elementos principales

- **Caso de pruebas**

– Un caso de pruebas es una clase que contiene una o varias pruebas.

– No necesitan ninguna anotación.

– Cada caso de prueba se ejecuta creando una nueva instancia de la clase y ejecutando sus pruebas.

Elementos principales

- ***Suite* de Pruebas**

– Las suites permiten agrupar clases de pruebas.

– Facilitan la estructuración de las pruebas de unidad, asociándolas a una funcionalidad superior a la que prueban directamente.

– Ejemplo: https://www.arquitecturajava.com/junit-test-suite-tdd-y-organizacion/

Elementos principales

Aserciones (*Assert*)

– Permiten definir las condiciones que se desea evaluar.

– La clase **`org.junit.Assert`** contiene una serie de métodos que facilitan su declaración.

- Si se superan las aserciones no ocurre nada.
- Si no se superan las aserciones se lanza una excepción.

Aserciones principales

- Algunas de las aserciones más utilizadas son
  - assertNull([“mensaje”,] A)
  - assertNotNull([“mensaje”,] A)
  - assertTrue([“mensaje”,] A)
  - assertFalse([“mensaje”,] A)
  - assertEquals([“mensaje”,] A, B)
  - assertSame([“mensaje”,] A, B)
  - assertNotSame([“mensaje”,] A, B)
  - assertArrayEquals([“mensaje”,] A, B)
  - fail([“mensaje”])
- Es recomendable que cada aserción lleve un mensaje descriptivo asociado.

Anotaciones

Anotaciones principales de Junit5:

– @After: Método que se ejecutará después de cada método de prueba.*

– @AfterClass: Método estático que se ejecutará después de todos los métodos de prueba y de contexto de una clase de pruebas.*

– @Before: Método que se ejecutará antes de cada método de prueba.*

– @BeforeClass: Método estático que se ejecutará antes de cualquier método de prueba o de contexto de una clase de pruebas.*

*Pueden definirse varios métodos, pero no se garantiza el orden de ejecución

Anotaciones

La configuración de JUnit4 se basa en **9 anotaciones**: (continuación)

– @Test: Indica que un método contiene un caso de prueba (es un método de prueba)

- @Nested: Indica que la clase anotada es una clase de prueba anidada no estática. Los métodos @BeforeAll y @AfterAll no se pueden usar directamente en una clase de prueba @Nested a menos que se use el ciclo de vida de la instancia de prueba por clase (@TestInstance(TestInstance.Lifecycle.PER_CLASS))

Buenas prácticas

- **Mismo paquete, distintos directorios**

– Las clases de dominio y las de prueba deben ir en el mismo paquete pero en directorios distintos.

– Al compartir el mismo paquete, las pruebas pueden acceder a los métodos protegidos.

– Al estar en distintos directorios, es más fácil diferenciar el código de pruebas del código de domino.

– El código de pruebas no debe ir en la distribución final.

Buenas prácticas

Usa nombres de pruebas descriptivos

– El nombre de los métodos debe facilitar la comprensión de qué se está probando.

– Es habitual utilizar el esquema de nombrado testXXXYYY, donde

- XXX es el nombre del método de dominio que se está probando.
- YYY (opcional) indica la diferencia entre distintos métodos de prueba que evalúan el mismo método de dominio.

Buenas prácticas

Una prueba de unidad, un método @Test

– No se debe agrupar varias pruebas en un método:

- Reducirá su legibilidad.
- Dificultará su comprensión.
- Será más probable es que exista un error en él (habrá que hacer *debug*).
- Una prueba que falla puede afectar a otras.

– Cuanto más conciso sea un método de prueba, mejor.

Buenas prácticas

Pruebas aisladas

– El éxito/fallo de una prueba no debe afectar al resto.
– El estado del sistema debería ser el mismo antes y después de ejecutar una prueba.
– Si no son aisladas, se puede producir una reacción en cadena que no permita saber qué pruebas han fallado realmente.
– La forma de conseguir esto es con una buena gestión en los métodos de gestión de *fixture*.
– Las bibliotecas de *mocking* (p.ej. EasyMock, Mockito, etc.) son muy útiles para llevar a cabo esta práctica.

Buenas prácticas

Explica el motivo por el cual falla una aserción

– Los método `assertX`, tienen una forma en la que
se puede especificar un mensaje informativo.
– Es recomendable utilizar dicho mensaje para describir el error detectado.
– Facilita el mantenimiento posterior y la documentación de las pruebas.

Buenas prácticas

Prueba todo lo que pueda fallar

– Con las pruebas de unidad se asegura que los métodos cumplen su especificación (contrato).

– Si, en algún momento, se cambia el método, podemos comprobar si sigue cumpliendo con su especificación.

Buenas prácticas

- **Haz las pruebas mantenibles**

– Las pruebas tienen un coste

- Tiempo de ejecución.
- Corrección cuando cambia el diseño/arquitectura del sistema.

– Cuida su código como si fuese el de la propia aplicación.

- Aplica los principios del buen diseño.

– No pruebes lo obvio.

- No hagas pruebas para bibliotecas de terceros.
- Prueba las partes relacionadas con la lógica de la aplicación.

Buenas prácticas

Deja que las pruebas mejoren tu código

– Cada prueba actúa como un cliente del API que se está probando.

– El hecho de tener que utilizar el API desde las pruebas facilita la identificación de errores de diseño o de interfaces demasiado complejas.

– Si es difícil de probar, entonces es probable que se pueda mejorar.

Buenas prácticas

Refactoriza

– En el desarrollo se añaden continuamente nuevas funcionalidades al sistema.

– Es probable que, cuando queramos añadir una nueva funcionalidad, el diseño del sistema no sea el adecuado.

– Mediante la refactorización podemos preparar el código para la nueva funcionalidad sin perder las anteriores.

– Al disponer de las pruebas de unidad podemos asegurar que la refactorización no afecta a las funcionalidades del sistema.

– Los IDEs actuales facilitan mucho el proceso de refactorización.

Mockito

¿Qué es Mockito?

Mockito es un objeto que simula ser otro para suplantarlo en un entorno determinado necesario para ejecutar el método que queremos probar. Esto nos permite centrarnos en el método que deseamos probar.

El comportamiento de un mockito debe ser programado previamente.

Anotaciones

@Mock: crea todos los objetos simulados

@InjectMocks: crea una instancia de la clase e inyecta los mocks que se crea en la anotación @Mock

No olvidarse de @ExtendWith(MockitoExtension.class)

Verificar el comportamiento

Comprobación de que las iteraciones se han realizado corretamente:

verify(“mock”). método(“parámetros”).

Stubbing

Programación de comportamiendo de los mocks, indicando qué deben devolver ciertos métodos:

When(“método”).thenReturn(“resultado esperado”);

Argument matchers

Permiten realizar llamadas a métodos mediante ‘comodines’ de forma que los parámetros a los mismos no se tengan que definir explícitamente.

Ejemplo:

```
when(mockedList.get(anyInt())).thenReturn(“element”);
```

Ejemplos Mockito

```java
@ExtendWith(MockitoExtension.class)
class Ejercicio80Test {
    @Mock
    DefaultOntimizeDaoHelper daoHelper;

    @InjectMocks
    MasterService service;


    @Nested
    @DisplayName("Test for Education queries")
    @TestInstance(TestInstance.Lifecycle.PER_CLASS)
    public class EducationQuery {

        @Test
        @DisplayName("Obtain all data from EDUCATION table")
        void when_queryOnlyWithAllColumns_return_allEducationData() {
            doReturn(getAllEducationData()).when(daoHelper).query(any(), anyMap(), anyList());
            EntityResult entityResult = service.educationQuery(new HashMap<>(), new ArrayList<>());
            assertEquals(EntityResult.OPERATION_SUCCESSFUL, entityResult.getCode());
            assertEquals(3, entityResult.calculateRecordNumber());
        }

        @Test
        @DisplayName("Obtain all data columns from EDUCATION table when ID is -> 2")
        void when_queryAllColumns_return_specificData() {
            HashMap<String, Object> keyMap = new HashMap<>() {{
                put("ID", 2);
            }};
            List<String> attrList = Arrays.asList("ID", "DESCRIPTION");
            doReturn(getSpecificEducationData(keyMap, attrList)).when(daoHelper).query(any(), anyMap(), anyList());
            EntityResult entityResult = service.educationQuery(new HashMap<>(), new ArrayList<>());
            assertEquals(EntityResult.OPERATION_SUCCESSFUL, entityResult.getCode());
            assertEquals(1, entityResult.calculateRecordNumber());
            assertEquals(2, entityResult.getRecordValues(0).get(EducationDao.ATTR_ID));
        }
```

Ejemplos Mockito

```java
public class ClientServiceTest {
	@Mock
	DefaultOntimizeDaoHelper daoHelper;

	@InjectMocks
	ClientService clientService;
	@Autowired
	ClientDao clientDao;

	@BeforeEach
	void setUp() {
		this.clientService = new ClientService();
		MockitoAnnotations.openMocks(this);
	}

	@Nested
	@DisplayName("Tests for Client inserts")
	@TestInstance(TestInstance.Lifecycle.PER_CLASS)
	public class Client_Insert {
		@Test
		@DisplayName("Insert a client successfully")
		void hotel_insert_success() {
			Map<String, Object> dataToInsert = new HashMap<>();
			dataToInsert.put("cl_nif", "98766789I");
			dataToInsert.put("cl_email", "alfredoperez@outlook.com");
			dataToInsert.put("cl_name", "Alfredo Pérez");
			dataToInsert.put("cl_phone", "985446789");
			EntityResult er = new EntityResultMapImpl(Arrays.asList("ID_CLIENT"));
			er.addRecord(new HashMap<String, Object>() {{put("ID_CLIENT", 2);}});
			er.setCode(EntityResult.OPERATION_SUCCESSFUL);
			HashMap<String, Object> keyMap = new HashMap<>();
			keyMap.put("ID_CLIENT", 2);
			when(daoHelper.insert(clientDao, dataToInsert)).thenReturn(er);
			EntityResult entityResult = clientService.clientInsert(dataToInsert);
			assertEquals(EntityResult.OPERATION_SUCCESSFUL, entityResult.getCode());
			int recordIndex = entityResult.getRecordIndex(keyMap);
			//assertEquals("INSERT_SUCCESSFULLY", entityResult.getMessage());
			assertEquals(2, entityResult.getRecordValues(recordIndex).get("ID_CLIENT"));
			verify(daoHelper).insert(clientDao, dataToInsert);

		}
```

Documentación oficial

Junit5: https://junit.org/junit5/docs/snapshot/user-guide/

Mockito4:
https://javadoc.io/doc/org.mockito/mockito-core/latest/org/mockito/Mockito.html

Curso recomendado

https://imatia.udemy.com/course/curso-completo-junit-mockito-spring-boot-test/#overview