JUnit

JUnit è un framework di test per il linguaggio di programmazione Java. È ampiamente utilizzato per scrivere test unitari che verificano il comportamento delle diverse parti di un'applicazione Java. JUnit supporta il concetto di "test-driven development" (TDD) e offre un ambiente strutturato per scrivere, eseguire e organizzare i test.

Ecco alcune informazioni fondamentali su JUnit:

1. **Annotazioni di Test:**
    - JUnit fa ampio uso di annotazioni per identificare i metodi di test e specificare il loro comportamento. Le annotazioni principali includono `@Test`, `@Before`, `@After`, `@BeforeClass`, e `@AfterClass`.

```
import org.junit.Test;
import static org.junit.Assert.*;

public class MyTestClass {

    @Before
    public void setUp() {
        // Codice eseguito prima di ogni metodo di test
    }

    @Test
    public void myTestMethod() {
        // Il tuo codice di test
        assertEquals(2, 1 + 1);
    }

    @After
    public void tearDown() {
        // Codice eseguito dopo ogni metodo di test
    }
}
```

2. **Assert Statements:**
    - JUnit fornisce una serie di metodi `assert` per verificare le asserzioni nei tuoi test. Alcuni esempi includono `assertEquals`, `assertTrue`, `assertFalse`, `assertNull`, `assertNotNull`, ecc.

```
import static org.junit.Assert.*;

@Test
public void testSum() {
    int result = Calculator.sum(2, 3);
    assertEquals(5, result);
}
```

3. **Test Suites:**
    - Puoi organizzare i tuoi test in suite per eseguire più test contemporaneamente. Puoi farlo utilizzando l'annotazione `@RunWith` e la classe `Suite`.

```
import org.junit.runner.RunWith;
import org.junit.runners.Suite;

@RunWith(Suite.class)
@Suite.SuiteClasses({Test1.class, Test2.class, Test3.class})
public class MyTestSuite {
    // La suite non richiede alcun codice aggiuntivo
}
```

4. **Parametrized Tests:**

- JUnit supporta test parametrizzati, che consentono di eseguire lo stesso test con diversi set di parametri.

```
import org.junit.Test;
import org.junit.runner.RunWith;
import org.junit.runners.Parameterized;
import java.util.Arrays;
import java.util.Collection;

@RunWith(Parameterized.class)
public class MyParameterizedTest {

    @Parameterized.Parameters
    public static Collection<Object[]> data() {
        return Arrays.asList(new Object[][]{
                {1, true},
                {2, false},
                {3, true}
        });
    }

    private int input;
    private boolean expected;

    public MyParameterizedTest(int input, boolean expected) {
        this.input = input;
        this.expected = expected;
    }

    @Test
    public void test() {
        // Il tuo codice di test parametrizzato
        assertEquals(expected, someMethod(input));
    }
}
```

5. **Mockito Integration:**
    - Mockito è un framework di mocking per test unitari in Java. Può essere utilizzato insieme a JUnit per simulare oggetti e comportamenti durante i test.

```
import static org.mockito.Mockito.*;

@Test
public void testSomethingWithMock() {
    // Creare un mock oggetto
    MyClass myMock = mock(MyClass.class);

    // Definire il comportamento del mock
    when(myMock.someMethod()).thenReturn("Hello");

    // Eseguire il test utilizzando il mock
    assertEquals("Hello", myMock.someMethod());
}
```

Queste sono solo alcune delle caratteristiche principali di JUnit. È uno strumento potente e flessibile che può essere utilizzato per garantire che il codice Java funzioni come previsto attraverso una suite di test automatizzati.

JUnit Test e Best Practices

Definizione di Unit Test Case:

- Un unit test case è una porzione di codice che verifica che un'altra parte di codice (generalmente un metodo) funzioni come previsto.
- Un test ben scritto ha un input noto e un output atteso stabiliti prima dell'esecuzione del test.
- Per ogni requisito (funzionalità implementate), è consigliabile avere almeno due test: uno positivo e l'altro negativo.
- JUnit consente l'identificazione dei metodi di test utilizzando annotazioni.
- Le asserzioni in JUnit sono utilizzate per confrontare i risultati ottenuti con quelli attesi.

Caratteristiche di JUnit:

- **Fixture:** La possibilità di impostare uno stato predefinito degli oggetti prima di eseguire un test. Ciò assicura un ambiente noto e ripetibile.
- **Test Runner:** Utilizzato per eseguire i test in modo trasparente all'utente.

Annotazioni in JUnit:

- `@Before`: Codice eseguito prima di ogni caso di test per creare la situazione iniziale corretta.
- `@After`: Codice eseguito dopo ogni caso di test per ripulire lo stato dell'ambiente.
- `@Test`: Codice per i casi di test veri e propri.

Asserzioni in JUnit:

- `assertEquals(expected, actual)`: Controlla che il valore restituito sia uguale a quello atteso.
- `assertSame(expected, actual)`: Controlla che i riferimenti siano identici.
- `assertTrue(actual) / assertFalse(actual)`: Utili per condizioni booleane.

Best Practices per Unit Testing:

- Test completamente automatizzati.
- Ogni caso di test dovrebbe coprire una sola funzionalità.
- L'insieme dei casi di test deve coprire tutte le funzionalità dell'unità.
- Evitare l'uso del costruttore del test case per impostare il test.
- Non assumere di conoscere l'ordine di esecuzione dei test case.
- Evitare test case con side effects.
- Utilizzare path relativi per leggere dati da locazioni del file system.
- Memorizzare i dati necessari per i test insieme ai test stessi.
- Assicurarsi che i nomi dei test siano time-independent.

Come testare metodi privati:

- Testare i metodi privati tramite i metodi pubblici che li utilizzano.
- Alternativamente, utilizzare la reflection.

Cosa testare:

- In generale, testare tanto più una parte di codice è visibile all'esterno, tanto più deve essere testata.

Esempio di Test:

```
import static org.junit.Assert.*;
import org.junit.Test;

public class CalculatorTest {

    @Test
    public void testAddition() {
        Calculator calculator = new Calculator();
        int result = calculator.add(2, 3);
        assertEquals(5, result);
    }

    @Test
    public void testSubtraction() {
        Calculator calculator = new Calculator();
        int result = calculator.subtract(5, 3);
        assertEquals(2, result);
    }
}
```

Questo esempio mostra come utilizzare JUnit con annotazioni e asserzioni per testare una classe `Calculator`. Implementare queste best practices migliorerà la qualità e la manutenibilità dei test unitari.