Testes unitários

- Testes unitários em Java são fundamentais para garantir a qualidade e a robustez do código. Eles permitem que desenvolvedores verifiquem individualmente partes específicas do código (métodos ou funções) para assegurar que funcionam corretamente de forma isolada. Aqui está uma explicação detalhada sobre testes unitários em Java:

Objetivos dos Testes Unitários

- Verificar o Comportamento: Garantir que métodos ou classes funcionem conforme o esperado.

- Identificar Bugs Precoce: Encontrar erros no início do ciclo de desenvolvimento.

- Facilitar Refatorações: Permitir a modificação do código sem medo de quebrar funcionalidades existentes.

- Documentação: Servir como uma forma de documentação que mostra como o código deve funcionar.

Frameworks Populares

- JUnit: O framework de teste unitário mais popular em Java. Atualmente, a versão mais usada é o JUnit 5.

- TestNG: Outra opção poderosa que oferece funcionalidades avançadas como paralelismo e configurações de grupos de testes.

Estrutura de um Teste Unitário

Dependências

- Certifique-se de adicionar as dependências do framework de teste ao seu projeto. Para JUnit 5 em um projeto Maven, por exemplo:

```
<dependency>
    <groupId>org.junit.jupiter</groupId>
    <artifactId>junit-jupiter-api</artifactId>
    <version>5.8.1</version>
    <scope>test</scope>
</dependency>
<dependency>
    <groupId>org.junit.jupiter</groupId>
    <artifactId>junit-jupiter-engine</artifactId>
    <version>5.8.1</version>
    <scope>test</scope>
</dependency>
```

Estrutura Básica

- Um teste unitário básico consiste em:

1. Setup: Inicialização de objetos e preparação do ambiente de teste.

2. Execução: Execução do método ou função a ser testada.

3. Verificação: Verificação dos resultados esperados.

4. Teardown (Opcional): Limpeza de recursos usados no teste.

- Exemplo básico usando JUnit 5:

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

class CalculatorTest {

    private Calculator calculator;

    @BeforeEach
    void setUp() {
        calculator = new Calculator();
    }

    @Test
    void testAddition() {
        assertEquals(5, calculator.add(2, 3));
    }

    @Test
    void testSubtraction() {
        assertEquals(2, calculator.subtract(5, 3));
    }
}
```

Anotações Comuns em JUnit

- @Test: Marca um método como um caso de teste.

- @BeforeEach: Executa um método antes de cada teste.

- @AfterEach: Executa um método após cada teste.

- @BeforeAll: Executa um método antes de todos os testes na classe.

- @AfterAll: Executa um método após todos os testes na classe.

- @Disabled: Ignora um teste.

Testando Exceções

- Você pode verificar se um método lança uma exceção esperada:

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

class CalculatorTest {

    private Calculator calculator = new Calculator();

    @Test
    void testDivisionByZero() {
        assertThrows(ArithmeticException.class, () -> calculator.divide(1, 0));
    }
}
```

Teste de Classes Dependentes

- Para testar classes que dependem de outras classes, pode ser necessário criar mocks das dependências usando bibliotecas como Mockito.

- Exemplo com Mockito:

```java
import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.Test;
import org.mockito.Mockito;

import static org.mockito.Mockito.*;
import static org.junit.jupiter.api.Assertions.*;

class ServiceTest {

    private Service service;
    private Dependency dependency;

    @BeforeEach
    void setUp() {
        dependency = Mockito.mock(Dependency.class);
        service = new Service(dependency);
    }

    @Test
    void testServiceMethod() {
        when(dependency.someMethod()).thenReturn("Mocked Result");

        String result = service.serviceMethod();

        assertEquals("Mocked Result", result);
        verify(dependency).someMethod();
    }
}
```

Assertions comuns (JUnit)

1) assertEquals(expected, actual): verifica se dois valores são iguais

2) assertNotEquals(unexpected, actual): verifica se dois valores não são iguais

3) assertTrue(condition): verifica se a condição é verdadeira

4) assertFalse(condition): verifica se a condição é falsa

5) assertNull(object): verifica se um objeto é nulo

6) assertNotNull(object): verifica se um objeto não é nulo

7) assertThrows(expectedType, executable): verifica se a exceção definida é lançada

8) assertArrayEquals(expectedArray, actualArray): verifica se dois arrays são iguais

Princípios de Boas Práticas

1. Testar Apenas uma Coisa por Teste: Cada teste deve verificar uma única condição ou comportamento.

2. Isolamento: Testes devem ser independentes e não devem depender da ordem de execução.

3. Nomear Testes Claramente: Use nomes de métodos descritivos para indicar o que está sendo testado e qual é a condição.

4. Cobertura de Testes: Procure cobrir todas as paths lógicas, mas lembre-se de que cobertura de testes não é um substituto para bons testes.

Execução de Testes

- IDE: A maioria das IDEs (Eclipse, IntelliJ IDEA) oferece suporte integrado para executar testes JUnit.

- Linha de Comando: Ferramentas de build como Maven e Gradle podem ser usadas para executar testes da linha de comando.

  - Maven: `mvn test`

  - Gradle: `gradle test`

Relatórios de Testes

- Ferramentas como Surefire (para Maven) podem ser usadas para gerar relatórios detalhados de execução de testes, o que é útil para integração contínua (CI).

Conclusão

- Testes unitários são uma parte essencial do desenvolvimento de software em Java, garantindo que o código funcione corretamente e facilitando a manutenção e evolução do sistema. Usando frameworks como JUnit e bibliotecas de mocking como Mockito, os desenvolvedores podem criar testes robustos e abrangentes para suas aplicações.