Guia Completo de Testes Unitários em Java

Agosto 2025

Guia detalhado sobre testes unitários em Java com exemplos práticos, melhores práticas e principais bibliotecas

Sumário

1 Introdução

Testes unitários são uma prática essencial no desenvolvimento de software, garantindo que cada unidade de código funcione corretamente de forma isolada. Este guia aborda testes unitários em Java, explorando conceitos fundamentais, bibliotecas populares, exemplos práticos e melhores práticas para criar testes robustos e confiáveis.

1.1 O que são Testes Unitários?

Testes unitários verificam o comportamento de uma unidade específica de código, como um método ou uma classe, isoladamente de suas dependências. Eles são rápidos, automatizados e focam na lógica individual.

1.2 Objetivo do Guia

Este documento tem como objetivo:

- Explicar os fundamentos dos testes unitários.
- Apresentar as principais bibliotecas de teste em Java (JUnit, TestNG, Mockito).
- Fornecer exemplos práticos e detalhados.
- Detalhar melhores práticas para escrever testes eficazes.
- Explorar ferramentas complementares e cenários avançados.

2 Fundamentos de Testes Unitários

2.1 Princípios de Testes Unitários

- **Isolamento**: Testes devem verificar uma unidade sem dependências externas.
- **Repetibilidade**: Resultados consistentes em todas as execuções.
- **Rapidez**: Testes devem ser rápidos para feedback imediato.
- **Automação**: Testes devem rodar sem intervenção manual.

2.2 Estrutura de um Teste Unitário

Um teste unitário segue o padrão AAA (Arrange, Act, Assert):

- **Arrange**: Configura o ambiente do teste (objetos, mocks, dados).
- **Act**: Executa a ação a ser testada.
- **Assert**: Verifica o resultado esperado.

3 Principais Bibliotecas de Teste

3.1 JUnit 5

JUnit 5 é a biblioteca de testes mais popular para Java, conhecida por sua flexibilidade e suporte a recursos modernos.

3.1.1 Características

- Arquitetura modular (JUnit Platform, Jupiter, Vintage).
- Anotações como `@Test`, `@BeforeEach`, `@AfterEach`.
- Suporte a testes parametrizados e dinâmicos.
- Integração com ferramentas como Maven e Gradle.

3.1.2 Exemplo com JUnit 5

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

class CalculatorTest {
    private final Calculator calculator = new Calculator();

    @Test
    void testAddition() {
        // Arrange
        int a = 5, b = 3;
        // Act
        int result = calculator.add(a, b);
        // Assert
        assertEquals(8, result, "5 + 3 deve ser igual a 8");
    }

    @Test
    void testDivisionByZero() {
        // Arrange
        int a = 10, b = 0;
        // Act & Assert
        assertThrows(ArithmeticException.class, () ->
           calculator.divide(a, b),
            "Divisão por zero deve lançar ArithmeticException");
    }
}
```

Listing 1: Teste JUnit 5 para uma calculadora

3.2 TestNG

TestNG é uma alternativa ao JUnit, com foco em testes mais complexos, como testes de integração e paralelos.

3.2.1 Características

- Suporte a grupos de testes e dependências.
- Configuração via XML ou anotações.
- Relatórios detalhados.
- Execução paralela de testes.

3.2.2 Exemplo com TestNG

```java
import org.testng.annotations.Test;
import static org.testng.Assert.*;

public class UserServiceTest {
    private UserService userService = new UserService();

    @Test
    public void testCreateUser() {
        // Arrange
        User user = new User("Alice", "alice@example.com");
        // Act
        boolean created = userService.createUser(user);
        // Assert
        assertTrue(created, "Usuário deve ser criado com
            sucesso");
    }

    @Test(dependsOnMethods = "testCreateUser")
    public void testFindUser() {
        // Act
        User user = userService.findUser("alice@example.com");
        // Assert
        assertNotNull(user, "Usuário deve ser encontrado");
        assertEquals(user.getName(), "Alice");
    }
}
```

Listing 2: Teste TestNG para um serviço de usuário

3.3 Mockito

Mockito é uma biblioteca para criar mocks e simular dependências externas.

3.3.1 Características

- Criação de mocks para interfaces e classes.
- Verificação de interações com mocks.
- Suporte a anotações como `@Mock`, `@InjectMocks`.
- Integração com JUnit e TestNG.

3.3.2 Exemplo com Mockito

```java
import org.junit.jupiter.api.Test;
import org.junit.jupiter.api.extension.ExtendWith;
import org.mockito.InjectMocks;
import org.mockito.Mock;
import org.mockito.junit.jupiter.MockitoExtension;
import static org.mockito.Mockito.*;
import static org.junit.jupiter.api.Assertions.*;

@ExtendWith(MockitoExtension.class)
class OrderServiceTest {
    @Mock
    private PaymentGateway paymentGateway;

    @InjectMocks
    private OrderService orderService;

    @Test
    void testProcessOrder() {
        // Arrange
        Order order = new Order(1, 100.0);
        when(paymentGateway.processPayment(100.0)).thenReturn(true);

        // Act
        boolean result = orderService.processOrder(order);

        // Assert
        assertTrue(result, "Processamento do pedido deve ser
           bem-sucedido");
        verify(paymentGateway, times(1)).processPayment(100.0);
    }
}
```

Listing 3: Teste com Mockito para um serviço

4 Exemplo Prático Completo

Abaixo, um exemplo completo de uma aplicação Java com testes unitários, cobrindo uma classe de serviço que gerencia produtos em um sistema de e-commerce.

4.1 Classe de Modelo

```java
public class Product {
    private int id;
    private String name;
    private double price;

    public Product(int id, String name, double price) {
        this.id = id;
        this.name = name;
```

```
        this.price = price;
    }

    public int getId() { return id; }
    public String getName() { return name; }
    public double getPrice() { return price; }
}
```

Listing 4: Classe Product

4.2 Classe de Serviço

```
public class ProductService {
    private ProductRepository repository;

    public ProductService(ProductRepository repository) {
        this.repository = repository;
    }

    public Product addProduct(Product product) {
        if (product.getPrice() <= 0) {
            throw new IllegalArgumentException("Preço deve ser
                maior que zero");
        }
        return repository.save(product);
    }

    public Product findProduct(int id) {
        Product product = repository.findById(id);
        if (product == null) {
            throw new IllegalArgumentException("Produto não
                encontrado");
        }
        return product;
    }
}
```

Listing 5: Classe ProductService

4.3 Classe de Repositório (Interface)

```
public interface ProductRepository {
    Product save(Product product);
    Product findById(int id);
}
```

Listing 6: Interface ProductRepository

4.4 Testes Unitários

```java
import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.Test;
import org.junit.jupiter.api.extension.ExtendWith;
import org.mockito.InjectMocks;
import org.mockito.Mock;
import org.mockito.junit.jupiter.MockitoExtension;
import static org.junit.jupiter.api.Assertions.*;
import static org.mockito.Mockito.*;

@ExtendWith(MockitoExtension.class)
class ProductServiceTest {
    @Mock
    private ProductRepository repository;

    @InjectMocks
    private ProductService productService;

    private Product product;

    @BeforeEach
    void setUp() {
        product = new Product(1, "Laptop", 1500.0);
    }

    @Test
    void testAddProductSuccess() {
        // Arrange
        when(repository.save(product)).thenReturn(product);

        // Act
        Product result = productService.addProduct(product);

        // Assert
        assertNotNull(result);
        assertEquals("Laptop", result.getName());
        verify(repository, times(1)).save(product);
    }

    @Test
    void testAddProductInvalidPrice() {
        // Arrange
        Product invalidProduct = new Product(2, "Mouse", -10.0);

        // Act & Assert
        assertThrows(IllegalArgumentException.class, () ->
            productService.addProduct(invalidProduct),
            "Deve lançar exceção para preço inválido");
    }

    @Test
```

```
51      void testFindProductSuccess() {
52          // Arrange
53          when(repository.findById(1)).thenReturn(product);
54
55          // Act
56          Product result = productService.findProduct(1);
57
58          // Assert
59          assertNotNull(result);
60          assertEquals(1, result.getId());
61          verify(repository, times(1)).findById(1);
62      }
63
64      @Test
65      void testFindProductNotFound() {
66          // Arrange
67          when(repository.findById(999)).thenReturn(null);
68
69          // Act & Assert
70          assertThrows(IllegalArgumentException.class, () ->
71              productService.findProduct(999),
72              "Deve lançar exceção para produto não encontrado");
73      }
74  }
```

Listing 7: Testes para ProductService com JUnit 5 e Mockito

5 Melhores Práticas

5.1 Escrever Testes Claros

- Use nomes descritivos para métodos de teste: `testAddProductSuccess`.
- Siga o padrão AAA para organização clara.
- Evite lógica complexa nos testes.

5.2 Isolamento de Testes

- Use mocks para simular dependências externas.
- Evite compartilhar estado entre testes.
- Use `@BeforeEach` para inicializar estados.

5.3 Cobertura de Código

- Busque alta cobertura de código (80-90% é um bom alvo).
- Use ferramentas como JaCoCo para medir cobertura.
- Priorize cenários críticos (ex.: casos de erro).

5.4 Testes Parametrizados

Use testes parametrizados para verificar múltiplos cenários com o mesmo teste:

```
import org.junit.jupiter.params.ParameterizedTest;
import org.junit.jupiter.params.provider.CsvSource;

@ParameterizedTest
@CsvSource({"10, 20, 30", "5, 5, 10", "0, 0, 0"})
void testAddition(int a, int b, int expected) {
    Calculator calculator = new Calculator();
    assertEquals(expected, calculator.add(a, b));
}
```

Listing 8: Teste Parametrizado com JUnit 5

5.5 Manutenção de Testes

- Refatore testes regularmente para manter clareza.
- Evite duplicação de código com métodos auxiliares.
- Documente testes complexos com comentários.

6 Ferramentas Complementares

- **JaCoCo**: Gera relatórios de cobertura de código.
- **PowerMock**: Para mockar métodos estáticos e construtores.
- **AssertJ**: Biblioteca de asserções fluida.
- **Maven/Gradle**: Gerenciamento de dependências e execução de testes.
- **Surefire/Failsafe**: Plugins para executar testes no Maven.

7 Cenários Avançados

7.1 Testes com Bancos de Dados

Use bancos de dados em memória como H2 para testes:

```
import org.junit.jupiter.api.Test;
import org.springframework.jdbc.core.JdbcTemplate;
import
   org.springframework.jdbc.datasource.embedded.EmbeddedDatabaseBuilder;

class DatabaseTest {
    @Test
    void testDatabaseQuery() {
        var db = new EmbeddedDatabaseBuilder().build();
        var jdbcTemplate = new JdbcTemplate(db);
```

```
        jdbcTemplate.execute("CREATE TABLE users (id INT, name
            VARCHAR(50))");
        jdbcTemplate.update("INSERT INTO users VALUES (1,
            'Alice')");
        String name = jdbcTemplate.queryForObject(
            "SELECT name FROM users WHERE id = 1", String.class);
        assertEquals("Alice", name);
    }
}
```

Listing 9: Teste com H2

7.2 Testes Assíncronos

Teste métodos assíncronos usando `CompletableFuture`:

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

class AsyncServiceTest {
    private AsyncService service = new AsyncService();

    @Test
    void testAsyncOperation() throws Exception {
        CompletableFuture<String> future =
            service.performAsyncTask();
        String result = future.get();
        assertEquals("Task completed", result);
    }
}
```

Listing 10: Teste Assíncrono

7.3 Mocking de Métodos Estáticos

Use PowerMock para mockar métodos estáticos:

```
import org.junit.Test;
import org.powermock.api.mockito.PowerMockito;
import static org.mockito.Mockito.*;

public class StaticMethodTest {
    @Test
    public void testStaticMethod() {
        PowerMockito.mockStatic(Utility.class);
        when(Utility.staticMethod()).thenReturn("Mocked");
        String result = Utility.staticMethod();
        assertEquals("Mocked", result);
    }
}
```

Listing 11: Teste com PowerMock

8 Erros Comuns e Soluções

- **Testes Frágeis**: Evite acoplamento com implementações internas.
- **Testes Lentos**: Use mocks para evitar chamadas a recursos externos.
- **Falsos Positivos**: Sempre valide o estado esperado.
- **Cobertura Enganosa**: Foque em cenários relevantes, não apenas em percentuais.

9 Conclusão

Testes unitários são cruciais para a qualidade do software em Java. Este guia apresentou as principais bibliotecas (JUnit 5, TestNG, Mockito), exemplos práticos, melhores práticas e ferramentas complementares. Adotar testes unitários robustos melhora a confiabilidade e a manutenibilidade do código, reduzindo erros em produção.

10 Referências

- https://junit.org/junit5/
- https://testng.org/
- https://mockito.org/
- https://www.jacoco.org/
- Meszaros, Gerard. *xUnit Test Patterns: Refactoring Test Code*, 2007.