***Documentação Projeto Integrador***

***Aluno:*** *Yuri Samuel Gomes Dos Santos*

***Tema:*** *JUNIT + MOCKITO :: TESTE UNITÁRIO*

## **Sumário**

1. Introdução
2. O que são Testes Unitários?
3. JUnit
   1. 3.1 O que é?
   2. 3.2 Estrutura Básica de Teste com JUnit
4. Mockito
   1. 4.1 O que é?
   2. 4.2 Por que usar Mockito?
5. JUnit + Mockito
   1. 5.1 Exemplo Prático
6. Benefícios de JUnit e Mockito
7. Conclusão

## **1. Introdução**

JUnit e Mockito são ferramentas essenciais para o desenvolvimento de testes unitários em aplicações Java. Elas trabalham juntas para permitir que você teste unidades de código de forma isolada, garantindo que sua aplicação seja confiável e de fácil manutenção.

Nesta documentação, vamos explorar o que são testes unitários, como usar JUnit e Mockito, e quais são os benefícios dessa combinação.

## **2. O que são Testes Unitários?**

### **Definição**

Testes unitários verificam partes isoladas do seu código — chamadas **unidades** — para garantir que funcionam como esperado. Uma unidade geralmente é um método ou uma função.

### **Objetivo**

* Detectar erros cedo no desenvolvimento.
* Garantir que mudanças no código não quebrem funcionalidades existentes (**regressão**).
* Melhorar a qualidade do software.

### **Benefícios**

* Facilita a manutenção do código.
* Reduz custos com correção de bugs.
* Proporciona confiança em alterações e refatoramentos.

## **3. JUnit**

### **3.1 O que é?**

JUnit é um **framework de testes unitários para Java**. Ele fornece ferramentas para criar, organizar e executar testes automatizados.

### **Principais Funcionalidades**

* Anotações como @Test para identificar métodos de teste.
* Assertivas (assertEquals, assertTrue, etc.) para verificar resultados.
* Agrupamento de testes em suites.

### **3.2 Estrutura Básica de Teste com JUnit**

#### **Exemplo**

Classe a ser testada:

public class Calculadora {  
 public int soma(int a, int b) {  
 return a + b;  
 }  
}

Teste unitário:

import org.junit.jupiter.api.Test;  
import static org.junit.jupiter.api.Assertions.assertEquals;  
  
public class CalculadoraTest {  
  
 @Test  
 public void testSoma() {  
 // Setup  
 Calculadora calc = new Calculadora();  
  
 // Execução  
 int resultado = calc.soma(2, 3);  
  
 // Verificação  
 assertEquals(5, resultado, "A soma de 2 e 3 deve ser 5");  
 }  
}

## **4. Mockito**

### **4.1 O que é?**

Mockito é uma biblioteca de **mocking** (simulação) que permite criar dependências simuladas para testar o comportamento de classes de forma isolada.

### **4.2 Por que usar Mockito?**

* **Isolamento**: Simula dependências externas, como APIs ou bancos de dados.
* **Controle**: Permite definir como as dependências devem responder.
* **Verificação**: Confirma se as dependências foram chamadas corretamente.

## **5. JUnit + Mockito**

### **5.1 Exemplo Prático**

#### **Cenário**

Uma classe UserService depende de um UserRepository para buscar dados:

public class UserService {  
 private UserRepository userRepository;  
  
 public UserService(UserRepository userRepository) {  
 this.userRepository = userRepository;  
 }  
  
 public String getUserFullName(int userId) {  
 User user = userRepository.findById(userId);  
 return user.getFirstName() + " " + user.getLastName();  
 }  
}

#### **Teste com JUnit e Mockito**

import static org.mockito.Mockito.\*;  
import org.junit.jupiter.api.Test;  
import static org.junit.jupiter.api.Assertions.assertEquals;  
  
public class UserServiceTest {  
  
 @Test  
 public void testGetUserFullName() {  
 // 1. Criar um mock para UserRepository  
 UserRepository userRepositoryMock = mock(UserRepository.class);  
  
 // 2. Configurar comportamento do mock  
 User mockUser = new User("John", "Doe");  
 when(userRepositoryMock.findById(1)).thenReturn(mockUser);  
  
 // 3. Instanciar o UserService com o mock  
 UserService userService = new UserService(userRepositoryMock);  
  
 // 4. Testar o método  
 String fullName = userService.getUserFullName(1);  
  
 // 5. Verificar o resultado  
 assertEquals("John Doe", fullName);  
  
 // 6. Verificar interação com o mock  
 verify(userRepositoryMock).findById(1);  
 }  
}

#### **Detalhamento**

1. **Mock do UserRepository**: Simula o repositório para evitar dependências reais.
2. **Configuração**: Define o que o mock deve retornar quando chamado.
3. **Teste**: Valida o comportamento do UserService.
4. **Verificação**: Confirma que o repositório foi chamado corretamente.

## **6. Benefícios de JUnit e Mockito**

* **Agilidade**: Testes são rápidos e fáceis de repetir.
* **Isolamento**: Permite testar unidades de forma independente.
* **Confiabilidade**: Garante que o código funciona como esperado.
* **Facilidade de Manutenção**: Testes bem escritos facilitam refatoramentos.

## **7. CONCLUSÃO:**

JUnit e Mockito são ferramentas fundamentais para criar testes unitários robustos. Eles ajudam a garantir que cada parte do código funcione conforme o esperado, promovendo qualidade e confiança no software. O uso combinado permite testar com precisão até mesmo classes que têm dependências externas complexas.