Seminário: Testes de Unidade com JUnit e Mockito no Java

Alunos: Andrielson Leonardo Teza Orientador: EVERTON COIMBRA DE

ARAUJO

Sumário

- 1 Capa 3 Introdução 4 O que são testes de unidade? 5 Por que são importantes no desenvolvimento moderno? 6 Comparação rápida 7 Anotações do JUnit 5 9 Exemplo: .NET (xUnit + Moa)
- 10 Criando o Projeto com
 Maven
 12 Exemplo: Java (JUnit + Mockito)
 13 Execução do código.
 19 Boas Práticas
 20 Conclusão



O que são testes de unidade?

Por que são importantes no desenvolvimento moderno?

Comparação breve: .NET (xUnit/Moq) vs Java (JUnit/Mockito)



O que são testes de unidade?

São testes que:

Avaliam unidades independentes de código.

São rápidos de executar.

Evitam dependências externas (banco de dados, APIs, arquivos etc.).

Usam simulações (mocks) para isolar componentes.

© Por que são importantes no desenvolvimento moderno?

Detecção precoce de erros: problemas são encontrados ainda na fase de codificação.

Facilitam refatorações: você pode melhorar o código com confiança.

Documentação viva: mostram como o código é esperado a se comportar.

Integração com CI/CD: testes automatizados são executados a cada push, garantindo qualidade contínua.

Reduzem custos de manutenção: falhas são localizadas mais rápido e com menor impacto.

Comparação rápida: .NET (xUnit/Moq) vs Java (JUnit/Mockito)

Característica	.NET (xUnit + Moq)	Java (JUnit + Mockito)
Framework de teste	xUnit (ou NUnit, MSTest)	JUnit (mais comum é o JUnit 5)
Mocking	Moq	Mockito
Integração com IDEs	Visual Studio, JetBrains Rider	IntelliJ IDEA, Eclipse
	Intuitivo, especialmente com Visual	Muito maduro e bem suportado em
Facilidade de uso	Studio	Java
Sintaxe de testes	[Fact], [Theory], uso com Assert	@Test, Assertions.assertEquals, etc.
	Moq:	Mockito:
	Mock <iservice>.Setup().Returns(</iservice>	when(service.method()).thenReturn(.
Simulação de dependências	.))
		Amplamente usado em todo o
Popularidade	Padrão em projetos .NET modernos	ecossistema Java

- Anotações do JUnit 5
- @Test

Marca um método como um teste de unidade.

@BeforeEach

Executa um método antes de cada teste. Serve para preparar o ambiente de teste.

```
@Test
void deveSomarDoisNumeros() {
     int resultado = 2 + 2;
    assertEquals(4, resultado);
@BeforeEach
void configurar() {
   System.out.println("Executando antes de cada teste");
```

Anotações do JUnit 5

@AfterEach

Executa um método depois de void limpar() {
cada teste. Serve para limpar
ou encerrar recursos.
}

```
@AfterEach
void limpar() {
    System.out.println("Executando depois de cada teste");
}
```

@DisplayName

Permite dar um nome mais legível ao teste (útil para relatórios e leitura).

```
@Test
@DisplayName("Deve retornar verdadeiro para e-mail válido")
void emailValidoTest() {
    assertTrue("teste@email.com".contains("@"));
}
```

📍 Exemplo: .NET (xUnit + Moq)

```
public class UserServiceTests
    [Fact]
    public void DeveCadastrarUsuarioComEmailValido()
        // Arrange
        var repoMock = new Mock<IUserRepository>();
        var service = new UserService(repoMock.Object);
       // Act
        bool resultado = service.CadastrarUsuario("teste@email.com");
        // Assert
        Assert.True(resultado);
        repoMock.Verify(r => r.Save(It.IsAny<User>()), Times.Once);
```

Criando o Projeto com Maven

No terminal, execute:

mvn archetype:generate -DgroupId=com.exemplo -DartifactId=teste-usuario -DarchetypeArtifactId=maven-archetype-quickstart -DinteractiveMode=false

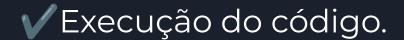
Criando assim a estrutura base do projeto:

Projeto com Maven

TESTE-USUARIO ✓ STC main\java\com\exemplo App.java User.java UserRepository.java UserService.java test\java\com\exemplo AppTest.java J UserServiceTest.java > target pom.xml

📍 Exemplo: Java (JUnit + Mockito)

```
@Test
void deveCadastrarUsuarioComEmailValido() {
    UserRepository repo = mock(UserRepository.class);
    UserService service = new UserService(repo);
    boolean resultado = service.cadastrarUsuario(email:"teste@email.com");
    assertTrue(resultado);
    verify(repo).save(any(User.class));
```



Para fazer a execução é usado o comando mvn test

C:\Users\andri\Documents\teste-usuario>mvn test

Picked up JAVA_TOOL_OPTIONS: -Dstdout.encoding=UTF-8 -Dstderr.encoding=UTF-8

[0.022s][warning][cds] This file is not the one used while building the shared archive file: C:\Users\andri\AppData\Roaming\Code\User\globalStorage\pleiades.java-extension-pack-jdk\java\latest\lib\modules

[0.023s][warning][cds] This file is not the one used while building the shared archive file: C:\Users\andri\AppData\Roaming\Code\User\globalStorage\pleiades.java-extension-pack-jdk\java\latest\lib\modules

[0.023s][warning][cds] C:\Users\andri\AppData\Roaming\Code\User\globalStorage\pleiades.java-extension-pack-jdk\java\latest\lib\modules

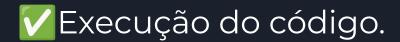
[0.023s][warning][cds] C:\Users\andri\AppData\Roaming\Code\User\globalStorage\pleiades.java-extension-pack-jdk\java\latest\lib\modules

size has changed.

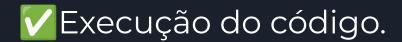
✓ Execução do código.

A saída como resultado será do comando mon test

```
[INFO] Results:
[INFO]
[INFO] Tests run: 2, Failures: 0, Errors: 0, Skipped: 0
```



A saída como resultado será do comando mon test



O comando mvn clean verify é usado no Maven, uma ferramenta de automação de build para projetos Java. Ele executa duas fases do ciclo de vida do Maven: clean e verify.

mvn clean

Objetivo: Limpar o projeto.

O que faz: Remove a pasta target/, que contém os artefatos compilados,

testes, relatórios, etc.



mvn verify

Objetivo: Verificar se o projeto atende todos os critérios de qualidade, testes e validações.

O que faz: Executa todas as fases anteriores até verify:

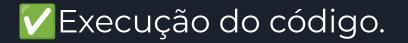
validate – Valida a estrutura do projeto.

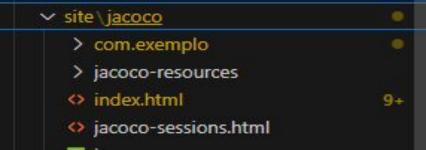
compile – Compila o código-fonte principal.

test - Executa os testes unitários.

package – Empacota o código (geralmente um .jar ou .war).

verify – Verifica se o build está correto (por exemplo, checa a cobertura de testes se você usa JaCoCo).







Testes independentes e repetíveis Nomes descritivos:

calcularDesconto_ValorMaiorQueCem_DeveAplicar10PorCento()

Não testar código trivial

Mocks para dependências externas



Testes de unidade ajudam a identificar erros rapidamente, garantindo que o código funcione corretamente mesmo após mudanças.

O JUnit 5 fornece uma estrutura robusta e moderna para escrever testes claros e organizados.

O Mockito permite simular dependências externas, permitindo testes realmente isolados.

Referência

Araújo, E. C. de. (2025). Testes de unidade. Universidade Tecnológica Federal do Paraná. Recuperado de https://moodle.utfpr.edu.br/

JUnit 5 – Documentação Oficial

https://junit.org/junit5/docs/current/user-guide/

Mockito - Documentação Oficial

https://site.mockito.org/