INSTITUTO INFNET

ESCOLA SUPERIOR DE TECNOLOGIA

GRADUAÇÃO EM ANÁLISE E DESENVOLVIMENTO DE SISTEMAS

TP1

Aluno: Enzo Furtini

12 de mai. de 2025

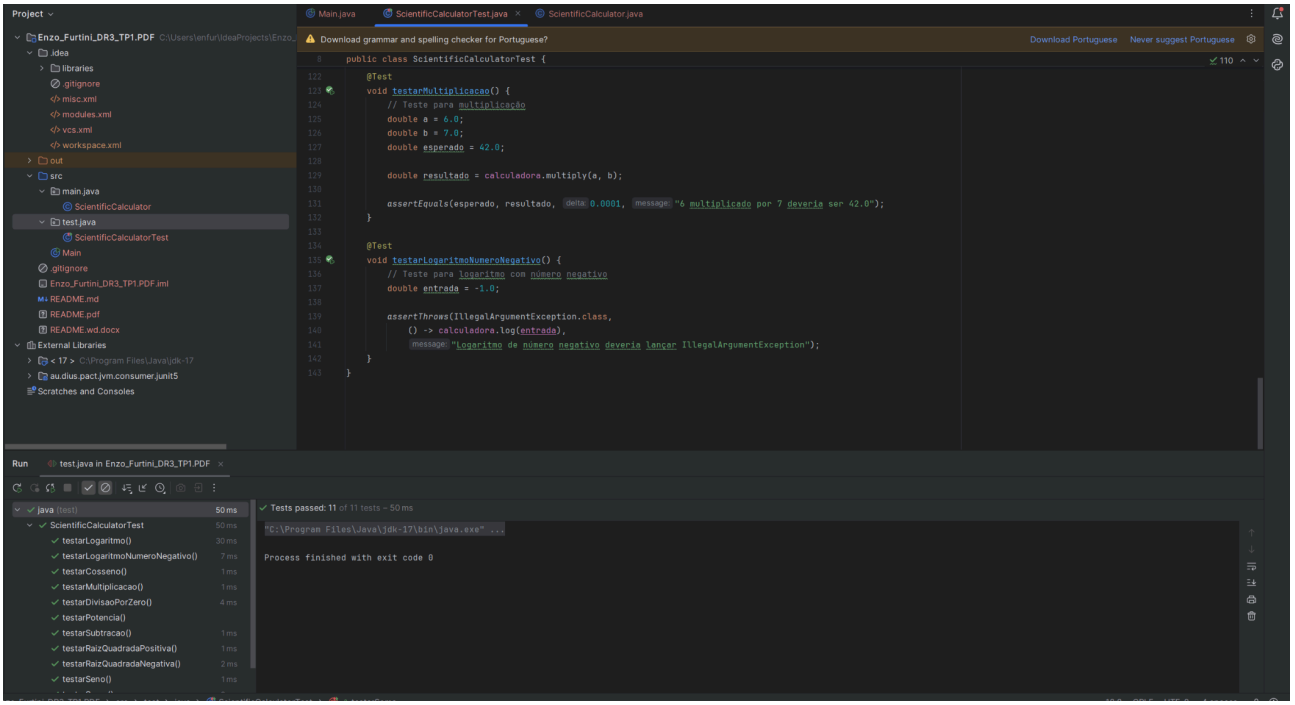
https://github.com/G-itch/Enzo_Furtini_DR3_TP1

Neste projeto, implementei testes unitários para uma calculadora científica usando JUnit 5. A ideia surgiu da necessidade de garantir que operações matemáticas complexas funcionem corretamente em diferentes cenários.

1 1. Configuração do Ambiente

Comecei configurando o ambiente de desenvolvimento:

- JUnit 5 para os testes (escolhi a versão mais recente por ter recursos interessantes)
- Estrutura de pastas seguindo o padrão Maven:
  - src/main/java para o código
  - src/test/java para os testes



2 2. Primeiro Teste

```
\@Test
void testAddition() {
    // Começando com números fáceis: 5 + 3
    double a = 5.0;
    double b = 3.0;
    double expected = 8.0;

    double result = calculator.add(a, b);

    // Usando delta de 0.0001 para lidar com imprecisões de ponto flutuante
    assertEquals(expected, result, 0.0001, "Ops! A soma deu errado!");
}
```

Quando executei, vi no painel:

- ✓ (verde) - Ufa, passou!
- Tempo: 0.023s - Bem rápido
- 1/1 testes passaram

3 3. As 4 Fases do Teste

```
\@Test
void testSubtraction() {
```

```
    // Setup: Preparando os números
    double a = 10.0;
    double b = 4.0;
    double expected = 6.0;

    // Execution: Rodando a subtração
    double result = calculator.subtract(a, b);

    // Assertion: Verificando se deu certo
    assertEquals(expected, result, 0.0001, "A subtração falhou!");
}
```

4 4. Reutilizando Código

Para não repetir código, usei o `\@BeforeEach`:

```
\@BeforeEach
void setUp() {
    // Criando uma calculadora nova antes de cada teste
    calculator = new ScientificCalculator();
}
```

5 5. Testando o Caminho Feliz

```
\@Test
void testSquareRootPositive() {
    // Testando com 16, que tem raiz exata
    double input = 16.0;
    double expected = 4.0;

    double result = calculator.squareRoot(input);

    assertEquals(expected, result, 0.0001, "A raiz quadrada deu errado!");
}
```

6 6. Testando Casos Problemáticos

```
\@Test
void testSquareRootNegative() {
    // Tentando calcular raiz de número negativo
    double input = -4.0;

    assertThrows(IllegalArgumentException.class,
        () -> calculator.squareRoot(input),
        "Deveria ter reclamado do número negativo!");
}
```

7 7. Lidando com Erros

```
\@Test
void testDivideByZero() {
    // Tentando dividir por zero
    double a = 10.0;
    double b = 0.0;

    assertThrows(IllegalArgumentException.class,
        () -> calculator.divide(a, b),
        "Deveria ter reclamado da divisão por zero!");
}
```

8 8. Testes Mais Complexos

- Logaritmo: Testei com e (2.71828...) que deve dar 1
- Seno: Usei 30° que deve dar 0.5
- Cosseno: 60° também deve dar 0.5

9 9. O Que Testar Primeiro?

Priorizei os testes assim:

1. Divisão por zero - erro crítico
2. Raiz de número negativo - erro comum
3. Log de número negativo - erro sutil
4. Funções trigonométricas - conversão de graus para radianos

10 10. Nomeando os Testes

Tentei deixar os nomes claros:

- `testSquareRootPositive` - diz exatamente o que testa
- `testDivideByZero` - indica o problema específico

11 11. Organizando Tudo

Separei os testes em grupos:

- Operações básicas (soma, subtração)
- Operações avançadas (potência, raiz)
- Funções trigonométricas
- Casos de erro

12 Conclusão

No final, aprendi que:

- Testes precisam ser claros e objetivos
- É importante testar tanto o que funciona quanto o que falha
- Organização faz diferença na manutenção
- Comentários ajudam muito quem vai ler depois