📘 Aula 01: Fundamentos de Testes de Software

Disciplina: Testes Automatizados (TSPI - IFTM)
Módulo: 01 - Introdução e Testes de Unidade
Professor: Dr. Bruno Queiroz Pinto

---

1. Introdução: Por que testar?

No desenvolvimento de software moderno, os testes deixaram de ser apenas uma etapa final de "caça aos bugs" para se tornarem uma ferramenta de design e garantia de qualidade contínua.

Verificação vs. Validação

Embora pareçam sinônimos, estes conceitos respondem a perguntas distintas na engenharia de software:

Conceito	Pergunta Chave	Foco	Exemplo Prático
Verificação	"Estamos construindo o produto **corretamente**?"	**Código & Lógica.** O software atende à especificação técnica? Não tem erros de sintaxe ou lógica?	O cálculo de juros está exato segundo a fórmula matemática?
Validação	"Estamos construindo o produto **certo**?"	**Requisitos & Valor.** O software resolve o problema real do usuário?	O fluxo de compra é intuitivo e permite que o usuário finalize o pedido?

---

2. A Estratégia da Pirâmide de Testes

Não é viável testar tudo manualmente, nem automatizar tudo via interface gráfica. A estratégia eficiente (proposta por Mike Cohn) é distribuir os testes em camadas.

Detalhando as Camadas

1. **Testes de Unidade (Base):**
- Testam a menor parte da lógica (um método/função) isoladamente.
- São a fundação da pirâmide pois são rápidos (milissegundos) e baratos de manter.

2. **Testes de Integração (Meio):**
- Verificam se as unidades funcionam juntas.
- Exemplo: Sua classe Conta salvando dados no Banco de Dados real ou comunicando com uma API externa.

3. **Testes E2E / Interface (Topo):**
- Simulam o comportamento do usuário final navegando no sistema.
- São vitais, mas **lentos** e **frágeis** (quebram facilmente com mudanças visuais).

---

3. Mergulhando em Testes de Unidade

O que é uma Unidade?

Em linguagens Orientadas a Objetos (como Java), a unidade é geralmente um **Método** dentro de uma **Classe**.

🔌 **A Analogia da Engenharia Elétrica**

> Antes de conectar um equipamento complexo, a engenharia testa seus componentes isoladamente:

1. **O Plugue Macho:** Verifica-se se os pinos estão alinhados e se não há curto interno.
2. **A Tomada Fêmea:** Verifica-se se a tensão está correta (127v/220v) e se há aterramento.

No Software: Se testarmos apenas o sistema integrado ("ligar o aparelho na tomada") e algo falhar, não saberemos se o erro está no plugue, na fiação ou na companhia elétrica. O teste de unidade isola o problema.

O que testar em uma classe?

Ao analisar uma classe para criar testes, devemos focar em 5 aspectos fundamentais:

1. **Interfaces:** Os parâmetros de entrada e saída estão corretos?
2. **Estruturas de Dados Locais:** As variáveis internas mantêm sua integridade durante a execução?

3. **Condições de Limite:** O método funciona nos extremos? (Ex: Testar com valor 0, valor negativo, ou valor exatamente igual ao saldo).
4. **Caminhos Independentes:** Todos os fluxos de controle (IFs, ELSEs, Loops) foram percorridos pelo menos uma vez?
5. **Tratamento de Erros:** O sistema lança a exceção correta quando algo inválido acontece?

---

4. O Padrão AAA (Arrange, Act, Assert)

É a estrutura universal para escrita de testes de unidade legíveis. Organiza o código em três etapas lógicas:

1. **Arrange (Preparar):** Inicializa os objetos, cria variáveis e define o cenário necessário.
2. **Act (Agir):** Executa a ação que queremos testar (chama o método).
3. **Assert (Validar):** Verifica se o resultado obtido ("Actual") é igual ao resultado esperado ("Expected").

---

5. Exemplo Prático: O "Teste Manual"

Antes de utilizarmos frameworks como o JUnit, é importante entender a lógica estrutural criando um teste manual com um método `main`.
Cenário: Validar a soma de dois números em uma classe `Calculadora`.
Java

```
package br.edu.iftm.tspi;

public class CalculadoraTesteManual {

    public static void main(String[] args) {

        // --- 1. ARRANGE (Preparação) ---
        Calculadora calc = new Calculadora();
        double numeroA = 10.0;
        double numeroB = 20.0;
        double resultadoEsperado = 30.0;

        // --- 2. ACT (Ação) ---
        double resultadoObtido = calc.somar(numeroA, numeroB);

        // --- 3. ASSERT (Validação) ---
        if (resultadoObtido != resultadoEsperado) {
            System.err.println("❌ FALHA: Esperava " + resultadoEsperado + " mas obteve " +
resultadoObtido);
        } else {
```

```
                System.out.println("✅ SUCESSO: O cálculo está correto.");
            }
        }
    }
```

Limitações do Teste Manual (main)

Embora funcione para exemplos triviais, esta abordagem não é escalável profissionalmente:

1. **Escalabilidade:** Se tivermos 500 testes, teremos 500 métodos main?
2. **Execução:** Se o teste falhar (lançar exceção), o programa para e não executa os próximos.
3. **Relatórios:** Não gera métricas ou relatórios de cobertura automáticos.
   *Para resolver isso, utilizaremos o framework **JUnit 5** nas próximas aulas.*

---

6. Boas Práticas (Princípios F.I.R.S.T.)

Para garantir a qualidade dos seus testes:

- **F (Fast):** Devem ser rápidos. Testes lentos desencorajam a execução.
- **I (Independent):** Um teste não pode depender do outro.
- **R (Repeatable):** Devem funcionar em qualquer ambiente (local, servidor, CI).
- **S (Self-validating):** O teste deve dizer se passou ou falhou (sem inspeção manual).
- **T (Timely):** Devem ser escritos no momento certo (idealmente junto com o código).

Outras Dicas:

- **Cenário Único:** Teste apenas uma coisa por vez. Se precisar testar o saque com saldo e sem saldo, crie dois testes diferentes.
- **Nomes Descritivos:** O nome do teste deve explicar o que ele faz (Ex: deveLancarErroAoSacarValorNegativo).

📚 Referências Bibliográficas

- **PRESSMAN, Roger S.** *Engenharia de Software: uma abordagem profissional*.
- **FOWLER, Martin.** *The Test Pyramid*.
- **SOMMERVILLE, Ian.** *Software Engineering*.