**📘 Política Institucional de Testes Unitários**

**1. Objetivo e Abrangência**

Esta política estabelece diretrizes claras para a criação, organização e manutenção de testes unitários no desenvolvimento de sistemas da instituição. Seu objetivo é:

* Promover consistência e legibilidade nos testes unitários.
* Garantir a qualidade, confiabilidade e rastreabilidade do código.
* Permitir que ferramentas automatizadas e agentes de IA possam gerar testes de acordo com padrões institucionais.
* Atender projetos desenvolvidos em múltiplas linguagens, com foco inicial em Java, .NET e JavaScript.

Esta política se aplica a todos os times de desenvolvimento e agentes automatizados que produzam ou mantenham testes unitários nos repositórios da instituição.

**2. Estrutura de Pastas e Convenção de Nomes**

**2.1 Estrutura de Pastas**

Os testes unitários devem estar organizados em pastas específicas separadas do código de produção:

/src

/<NomeDoProjeto>

/domain

/services

...

/tests

/<NomeDoProjeto>.Tests

/domain

ClasseAShould.js

/services

ClasseBShould.test.cs

*A separação clara facilita a manutenção, execução e versionamento.*

**2.2 Convenção de Nomes**

**Estilo adotado:** *sentenças descritivas legíveis por humanos*

* Nome da classe de teste: <NomeClasse>Should
  + Exemplo: BankAccountShould.java
* Nome do método de teste: sentença descritiva em inglês (ou português se o projeto for inteiramente em português), no formato:

Should\_return\_false\_when\_balance\_is\_insufficient

Deve\_retornar\_erro\_quando\_usuario\_nao\_autenticado

* Convenção AAA (Arrange, Act, Assert) deve ser adotada na estrutura do código.

**3. Avaliação da Qualidade dos Testes**

Todo teste unitário deve ser avaliado segundo os critérios abaixo:

| **Critério** | **Descrição** |
| --- | --- |
| Clareza e legibilidade | O nome do teste deve ser autoexplicativo e expressar a regra de negócio |
| Isolamento | Testes não devem depender de serviços externos (ex: banco, rede, tempo) |
| Execução rápida | Deve ser executável em milissegundos |
| Determinismo | Deve sempre retornar o mesmo resultado dado o mesmo código |
| Uso de mocks/fakes/stubs | Quando necessário, simular dependências para garantir isolamento |
| Cobertura relevante | Cobrir caminhos principais, bordas e exceções com testes distintos |
| Comentários | Apenas quando necessário para explicar contexto fora do escopo do nome |

**4. Exemplos por Linguagem**

**4.1 Java (JUnit)**

class BankAccountShould {

@Test

void should\_not\_allow\_withdrawal\_when\_balance\_is\_insufficient() {

BankAccount account = new BankAccount(50);

boolean result = account.withdraw(100);

assertFalse(result);

}

}

**4.2 C# (.NET xUnit)**

public class BankAccountShould {

[Fact]

public void Should\_return\_false\_when\_balance\_is\_insufficient() {

var account = new BankAccount(50);

var result = account.Withdraw(100);

Assert.False(result);

}

}

**4.3 JavaScript (Jest)**

describe('BankAccountShould', () => {

test('should return false when balance is insufficient', () => {

const account = new BankAccount(50);

const result = account.withdraw(100);

expect(result).toBe(false);

});

});