BIUD Talks

TESTE UNITÁRIO

ALEX PRAXEDES - FAZEDOR DE CÓDIGO

Cronograma do Programa

🤖 22/04	**Wanderson**	Inteligência Artificial para Dev
✅ 29/04	**Praxedes**	Testes Unitários
🏗️ 06/05	**Luk**	Arquitetura de Software
🔗 13/05	**Regis**	Integração de Sistemas
🔍 20/05	**Rogério**	Root Cause Analysis (RCA)
⚡ 27/05	**Gustavo**	Metodologia Ágil
⚡ 03/06	**Gabriel Pereira**	Banco de Dados

Estrutura de cada Sessão

Apresentação

20-30 minutos

Conteúdo e expertise do palestrante

Debate & Discussão

30 minutos

Dúvidas, insights e networking

👤 Moderador: Rogério

Responsável por mediar o debate e garantir engajamento de todos os participantes

TESTE UNITÁRIO

- Por que testes importam
- O que é teste unitário
- Para que serve
- Boas práticas
- Erros comuns
- Exemplos

Por que testes importam

O software quebra, está quebrado ou quebrará!

- Software muda.
- Mudança cria risco.
- Risco vira bug.

Toda aplicação é um organismo vivo!

- Tudo que é vivo se transforma.
- Toda transformação gera impacto.
- Impacto sem validação gera instabilidade.

Testes reduzem o custo da mudança.

- Validam comportamentos importantes.
- Revelam quebras mais cedo.
- Dão confiança para evoluir o sistema.

O que é teste unitário

Validação automática de uma pequena parte do código.

- Uma função.
- Um método.
- Uma regra de negócio.

Verifica um comportamento esperado.

- Dado um cenário.
- Quando uma ação acontece.
- Então o resultado deve ser conhecido.

Ele deve ser rápido, previsível e independente.

- Sem banco real.
- Sem API externa.
- Sem dependências desnecessárias.

Então teste unitário é:

Testa a menor parte útil do sistema?

- ❑ SIM
- ❑ COM CERTEZA
- ❑ DEPENDE

Faz sentido testar:

- ➔ Uma regra de cálculo
- ➔ Uma validação importante
- ➔ Uma decisão do sistema
- ➔ Um comportamento com risco real

Não faz sentido testar:

- ➔ Obviedades sem valor
- ➔ Detalhes sem impacto real
- ➔ Código irrelevante para o comportamento
- ➔ Coisas que só aumentam cobertura sem confiança

Para que serve

Proteger comportamentos importantes.

- Garante que regras continuem funcionando.
- Evita regressões após mudanças.
- Revela quebras antes do usuário final.

Dar segurança para evoluir o sistema.

- Permite refatorar com mais confiança.
- Reduz medo de alterar código existente.
- Acelera manutenção e evolução do software.

Documentação viva do comportamento do software.

- Cenários e comportamentos esperados.
- O contexto funcional de uma lógica.
- As regras que o time quer preservar.

Boas práticas e erros comuns

Boas práticas

- Testar comportamento, não implementação.
- Manter testes claros, rápidos e previsíveis.
- Isolar dependências externas quando necessário.
- Cobrir regras, decisões e cenários relevantes.

Erros comuns

- Testar detalhes internos do código.
- Criar testes frágeis e difíceis de entender.
- Depender de banco, APIs ou ambiente externo sem necessidade.
- Escrever testes só para aumentar cobertura.

Exemplos

Lógica conceitual

- Cenário conhecido.
- Ação executada.
- Resultado esperado.

```
DADO uma compra de R$ 150
QUANDO o desconto for calculado
ENTÃO o total deve ser R$ 135
```

Exemplos

Regra em um serviço

- Testa uma regra em um serviço
- Usa entrada conhecida
- Valida a saída esperada

```typescript
it('deve aplicar 10% de desconto em compras acima de R$ 100', () => {
  const total = service.calculateTotal(150)

  expect(total).toBe(135)
})
```

Exemplos

Testa comportamento e não visual

- Testa comportamento da interface
- Simula uma ação do usuário
- Valida a resposta da tela

TypeScript

```typescript
it('deve exibir mensagem de erro quando o campo estiver vazio', () => {
  render(<LoginForm />)

  fireEvent.click(screen.getByText('Entrar'))

  expect(screen.getByText('Informe seu e-mail')).toBeInTheDocument()
})
```

Obrigado!

Até o próximo episódio da BIUD Talks:

🏗️ **06/05** | *Arquitetura de Software* - Luk

💡 Inovando juntos | 🚀 Crescendo juntos