

QUALIDADE E
TESTE
DE SOFTWARE



PROFº LUIZ CLÁUDIO

Introdução ao Desenvolvimento Guiado por Testes TDD

Desenvolvimento Guiado por Testes (Test-Driven-Development)

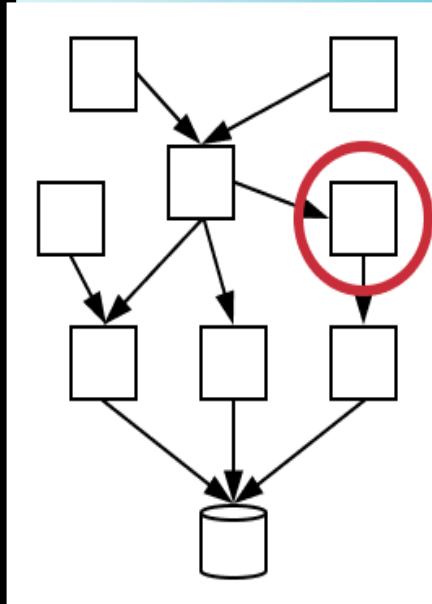
O desenvolvimento guiado por testes é um dos temas mais importantes para ser aprendido por um desenvolvedor profissional.

Ao se trabalhar com o desenvolvimento guiado por testes (TDD), a tendência é que a qualidade do sistema aumente consequentemente.

Tipos de testes automatizados

Testes de unidade (ou teste unitário):

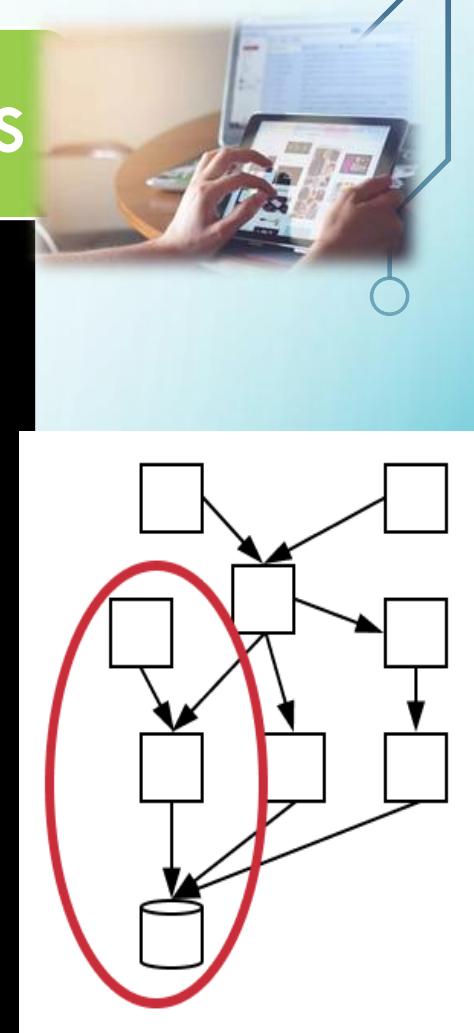
- Verifica individualmente as menores partes testáveis de uma aplicação, conhecidas como unidades. Essas unidades podem ser funções, métodos, classes ou procedimentos dentro de um programa. O objetivo principal dos testes de unidade é garantir que cada unidade de código funcione corretamente isoladamente.



Tipos de testes automatizados

Testes de integração:

- Verifica a interação entre diferentes módulos ou componentes de um sistema para garantir que eles funcionem corretamente em conjunto. Ao contrário dos testes de unidade, que testam partes isoladas do código, os testes de integração focam em como essas partes interagem entre si.



Tipos de testes automatizados



Test Driven Development TDD

- TDD é uma prática de desenvolvimento de software onde os testes são escritos antes do código funcional. Em resumo, funciona assim:
- Escreva um teste: Antes de implementar uma nova funcionalidade, escreva um teste. Esse primeiro teste deve falhar, pois a funcionalidade ainda não existe.



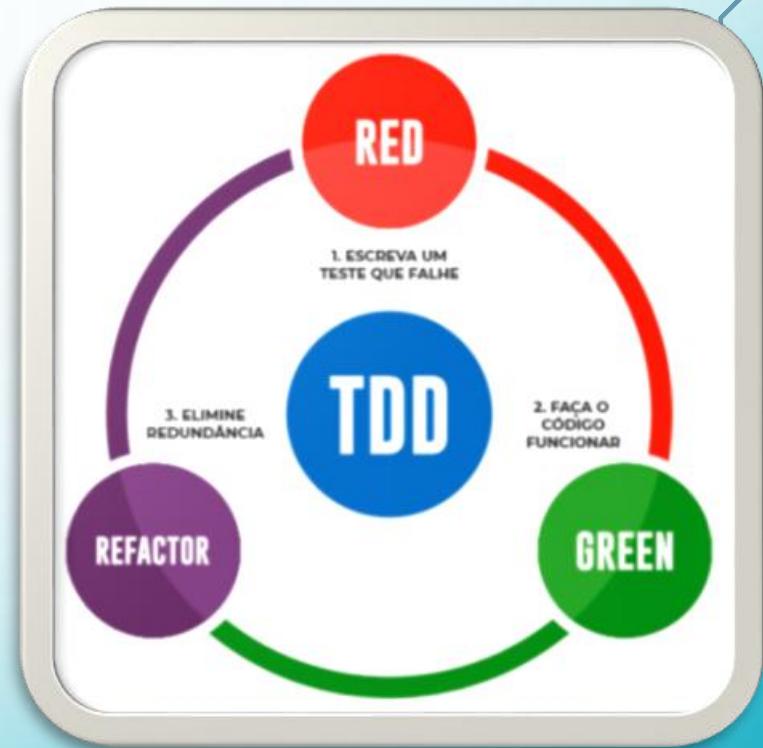
Implemente o código: Escreva o código necessário para passar no teste.

Refatore: Melhore o código, mantendo os testes passando.

Esse ciclo se repete para cada pequena funcionalidade, garantindo que o código seja continuamente testado e melhorado.

Test Driven Development TDD

- O ciclo básico do TDD é geralmente descrito como
- "Red-Green-Refactor":
- **Red:** Escrever um teste que falha (fase vermelha).
- **Green:** Implementar o código mínimo necessário para fazer o teste passar (fase verde).
- **Refactor:** Refatorar o código, melhorando a estrutura e a qualidade sem alterar o comportamento.



Esse ciclo garante que cada pedaço de funcionalidade é testado e implementado de forma incremental e controlada, resultando em um código de alta qualidade e bem testado.

Vantagens do TDD

- **-Qualidade do Código:** Aumenta a qualidade do código, promovendo design mais limpo e modular.
- **-Detecção de Erros:** Facilita a detecção de erros logo no início do desenvolvimento.
- **-Manutenção:** Código mais fácil de manter e refatorar.
- **-Confiança:** Proporciona mais confiança ao desenvolvedor, sabendo que o código funciona conforme o esperado.



Desvantagens do TDD

- **-Tempo Inicial:** Pode aumentar o tempo de desenvolvimento inicial, já que envolve a escrita de testes antes do código.
- **-Complexidade:** Requer uma mudança na mentalidade e disciplina dos desenvolvedores, o que pode ser difícil para alguns.
- **-Cobertura de Testes:** Pode não cobrir todos os casos, especialmente em sistemas complexos.
- **-Manutenção dos Testes:** Os testes precisam ser mantidos junto com o código, o que pode ser oneroso em projetos grandes.



Testes no JavaScript com JEST

- Jest é uma ferramenta de teste de JavaScript. Em termos simples, ele é usado para escrever e executar testes automatizados em aplicações JavaScript, especialmente em projetos que utilizam bibliotecas e frameworks como o React.



- O Jest ajuda a garantir que o código funcione conforme esperado, facilitando a detecção de erros durante o desenvolvimento.

Testes no JavaScript com JEST

- **1º passo: Escrever o teste e executá-lo.**
- **O primeiro teste deve falhar pois a função a ser testada ainda não existe. (fase RED)**



```
1 // Testando soma da calculadora
2 test("Deve retornar o valor 10 ao somar 5 + 5", () => {
3     let resultado = app.soma(5,5)
4     expect(resultado).toEqual(10)
5 })
```



Jest

Testes no JavaScript com JEST

```
> jest
```

```
FAIL test/calculadora.test.js
```

```
  Operações básicas
```

```
    × Deve retornar o valor 10 ao somar 5 + 5 (2 ms)
```

```
● Operações básicas > Deve retornar o valor 10 ao somar 5 + 5
```

```
TypeError: app.soma is not a function
```

```
16 |     // Testando soma da calculadora
17 |     test("Deve retornar o valor 10 ao somar 5 + 5", () => {
> 18 |         let resultado = app.soma(5,5)
          ^
19 |         expect(resultado).toEqual(10)
20 |
21 |     })
```

```
at Object.soma (test/calculadora.test.js:18:29)
```

```
Test Suites: 1 failed, 1 total
```

```
Tests:       1 failed, 1 total
```

```
Snapshots:  0 total
```

```
Time:        0.676 s, estimated 1 s
```

```
Ran all test suites.
```

Testes no JavaScript com JEST

- 2º passo: Implementar o código mínimo necessário para fazer o teste passar (**fase GREEN**).

```
1 // app.js:  
2  
3 module.exports = {  
4     soma: function(a,b){  
5         return a + b  
6     }  
7 }  
8
```



Jest

Testes no JavaScript com JEST

```
> jest
```

```
PASS ./calculadora.test.js
```

```
✓ Deve fazer algo quando receber X (2 ms)
```

File	% Stmt	% Branch	% Funcs	% Lines	Uncovered Line #s
All files	100	100	100	100	
index.js	100	100	100	100	

```
Test Suites: 1 passed, 1 total
```

```
Tests:       1 passed, 1 total
```

```
Snapshots:  0 total
```

```
Time:        0.409 s, estimated 1 s
```

```
Ran all test suites.
```

Testes no JavaScript com JEST

- 3º passo: Refatorar o código, melhorando a estrutura e a qualidade sem alterar o comportamento.

```
1 // app.js:  
2  
3 module.exports = {  
4     soma: (n1, n2) => {  
5         return n1 + n2  
6     }  
7 }
```





**TESTES FUNCIONAIS
E NÃO FUNCIONAIS**



Testes funcionais

- Os testes funcionais se concentram em verificar se o software realiza as funções que foram especificadas nos requisitos. Eles garantem que o sistema se comporta como esperado quando os usuários interagem com ele. Exemplos de testes funcionais incluem:
- 1. **Teste de Unidade:** Testa componentes individuais do software, como funções ou métodos, isoladamente.
- 2. **Teste de Integração:** Verifica se diferentes módulos/serviços do sistema funcionam corretamente juntos.



Testes funcionais

- **3. Teste de Sistema:** Avalia o sistema completo para garantir que ele atende aos requisitos especificados.



- **4. Teste de Aceitação:** Realizado para garantir que o sistema atende aos critérios de aceitação e está pronto para ser entregue ao cliente.

Testes não funcionais

- Os testes não funcionais se concentram em aspectos do software que não estão diretamente relacionados às funções específicas do sistema, mas que são igualmente importantes para a qualidade geral.

Exemplos de testes não funcionais incluem:

- **1. Teste de Usabilidade:** Avalia a facilidade de uso do sistema, incluindo a interface do usuário e a experiência do usuário.



Testes não funcionais

- **2. Teste de Desempenho:** Avalia a velocidade, escalabilidade e estabilidade do sistema sob diferentes cargas.
- **3. Teste de Segurança:** Verifica se o sistema é protegido contra ameaças e ataques, garantindo a integridade, confidencialidade e disponibilidade dos dados.
- **4. Teste de Compatibilidade:** Verifica se o sistema funciona em diferentes ambientes, como navegadores, sistemas operacionais e dispositivos.

