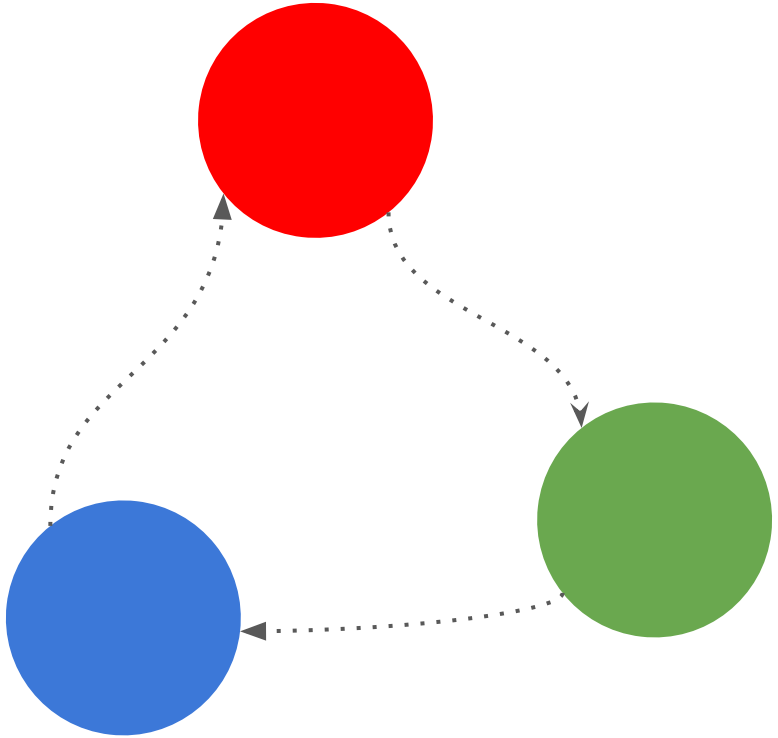
Desenvolvendo software com uma

mentalidade de testes



Oi,

meu nome é **Ciço**

Nem todo Ciço é de Juazeiro do Norte

Cícero Viana

Desenvolvedor front-end

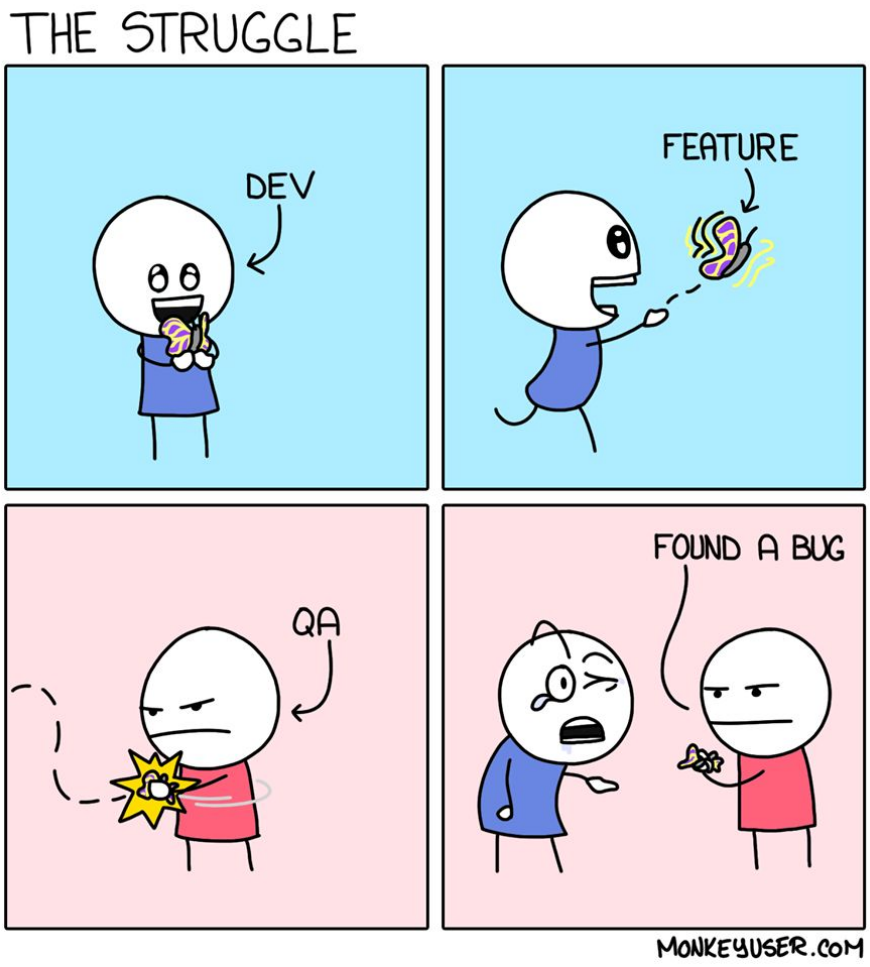
linktr.ee/cicerohen

Antonov an-225 - aeronave cargueira ucraniana produzida na década de 80

280 toneladas | 88 metros de largura

O que acontece antes de uma **máquina gigante**, mais pesada que o ar e com uma engenharia complexa ir para a “produção”?

Por que testar?


THE STRUGGLE
DEV
FEATURE
QA
FOUND A BUG
MONKEYUSER.COM

Testes ajudam a
identificar possíveis **bugs**
nos estágios iniciais do
desenvolvimento

Testes ajudam a manter
qualidade do **código**

Testes **facilitam ajustes**
no código

Testes **ajudam** no **entendimento** do design e **regras de negócio**

Testes **documentam** os
fluxos e **comportamentos**
do software

Por que não testamos?

Falta de **tempo**(?)

Complexidade das ferramentas e conceitos(?)

Pouco entendimento
sobre os fluxos e **regras de negócio**(?)

Time imaturo ou desalinhado em relação aos benefícios da cultura de teste(?)

O que é preciso para começar a aplicar testes?

Entendimento dos **conceitos**

Entendimento mínimo das API’s das **ferramentas**

Mas somente
ferramentas e conceitos
não são suficientes

Testes **nem sempre serão divertidos**, por isso...

Resiliência, **persistência** e **paciência** são palavras essenciais para a manutenção da cultura de testes

Conceitos

Conceitos

Mock

É um objeto com comportamento pré-programado usado na simulação de cenários

```
import { RANDOM_TEXT_URL } from "./config";

const fetchRandomText = async () => {
  try {
    const response = await fetch(RANDOM_TEXT_URL);
    const result = await response.json();
    return result;
  } catch (err) {
    throw err;
  }
};

export default fetchRandomText;

```

```
1 import { RANDOM_TEXT_URL } from "./config";
2
3 const fetchRandomText = async () => {
4   try {
5     const response = await fetch(RANDOM_TEXT_URL);
6     const result = await response.json();
7     return result;
8   } catch (err) {
9     throw err;
10  }
11 };
12
13 export default fetchRandomText;
14
```

O resultado da chamada de **fetch();** define o retorno da função **fetchRandomText();**

Conceitos

Mock

Simulando o **cenário de sucesso**

Define previamente o valor e o status HTTP que será retornado pelo **fetch();**

Executa a função a ser testada

```
1  import { RANDOM_TEXT_URL } from "../config";
2  import fetchRandomText from "../fetchRandomText";
3
4  describe("fetchRandomText() test", () => {
5      it("deve retornar com sucesso", async () => {
       const body = ['TEXTO'];
7      fetchMock.get(RANDOM_TEXT_URL, JSON.stringify(body), {
8        delay: 200
9      });
       const result = await fetchRandomText();
11     expect(result).to.deep.equal(body);
12   });
13 });
14
```

Verifica se para o cenário de sucesso da requisição, a função que está sendo testada retorna um body correto

Conceitos

Mock

Simulando o **cenário de erro**

Define previamente que fetch(); retornará um erro

Executa a função a ser testada

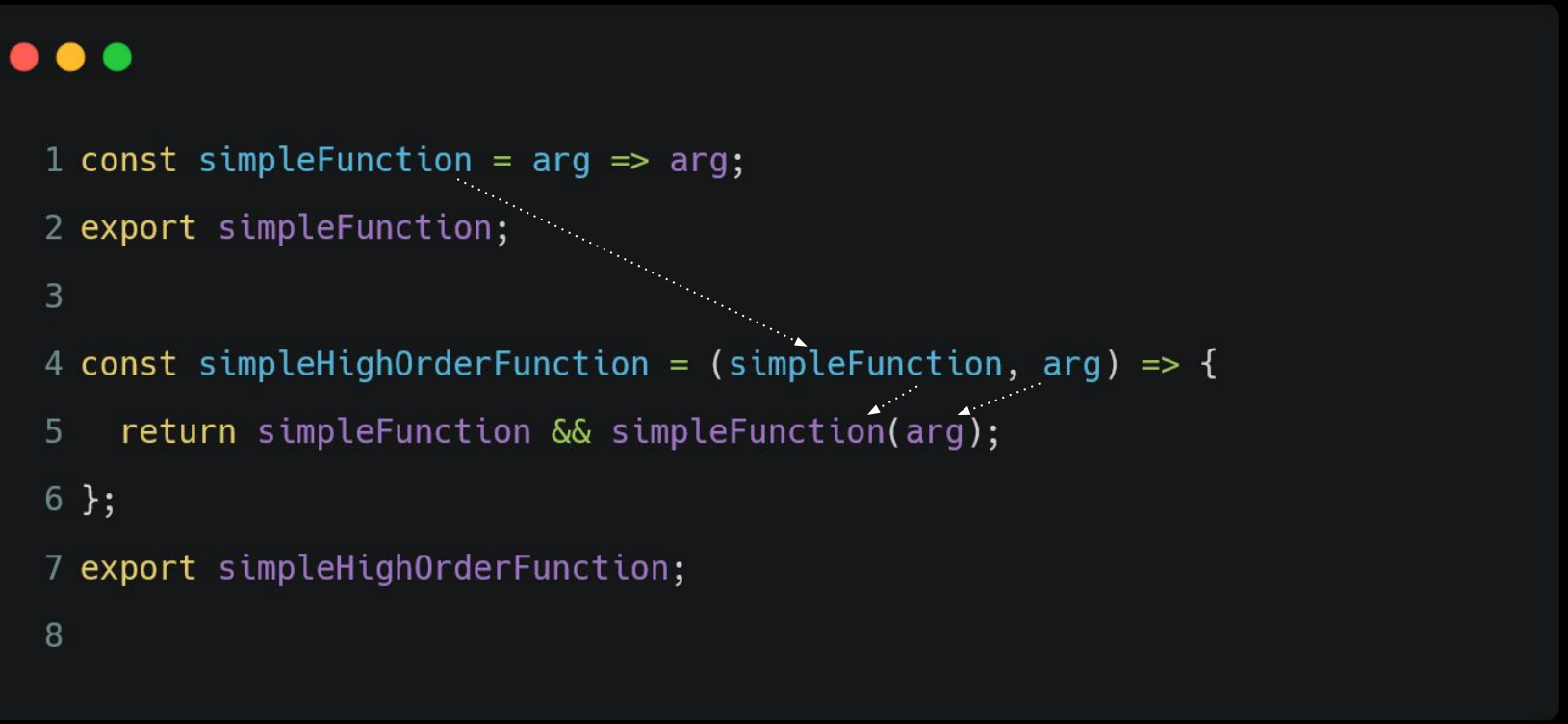
```
1 import { RANDOM_TEXT_URL } from "../config";
2 import fetchRandomText from "../fetchRandomText";
3
4 describe("fetchRandomText() test", () => {
5    it("deve retornar com erro", async () => {
6     const message = "Error";
7     fetchMock.mock(RANDOM_TEXT_URL, {
8       throws: new Error(message)
9     });
10    try {
11      await fetchRandomText();
12    } catch (err) {
13     expect(err.message).to.be.equal(message);
14    }
15  });
16 });
17
```

Espera que para o cenário de erro da requisição, a função testada retorne um erro.

Conceitos

Spy

É um objeto/função que armazena o estado das suas interações ocorridas dentro de outros contextos


1 const simpleFunction = arg => arg;
2 export simpleFunction;
3
4 const simpleHighOrderFunction = (simpleFunction, arg) => {
5 return simpleFunction && simpleFunction(arg);
6 };
7 export simpleHighOrderFunction;
8


Define previamente uma função spy que será passada como parâmetro.
Define previamente um valor para o segundo argumento da função a ser testada
Executa a função a ser testada
Espera que a função spy seja chamada uma única vez dentro da função a ser testada
1 import sinon from "sinon";
2 import simpleHighOrderFunction from "../simpleHighOrderFunction";
3
4 describe("simpleHighOrderFunction() test", () => {
it("deve executar simpleFunction() corretamente", () => {
6 const simpleFunctionSpy = sinon.spy();
7 const arg = 20;
8 simpleHighOrderFunction(simpleFunctionSpy, arg);
9 expect(simpleFunctionSpy).to.have.been.calledOnceWith(arg);
10 });
11 });
12

Ferramentas

Executar os testes

Criar objetos mock

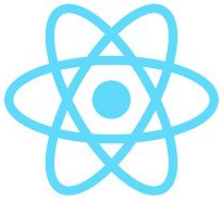
Criar objetos spy

Realizar asserções


Jest


chai


SINON.JS




Jest


chai


SINON.JS


JS


Jest


chai


SINON.JS

O que define uma unidade de teste?

Etapas de uma unidade de teste

Etapas de uma unidade de teste

Identificação

Identificar o objeto ou cenário a ser testado

Etapas de uma unidade de teste

Simulação

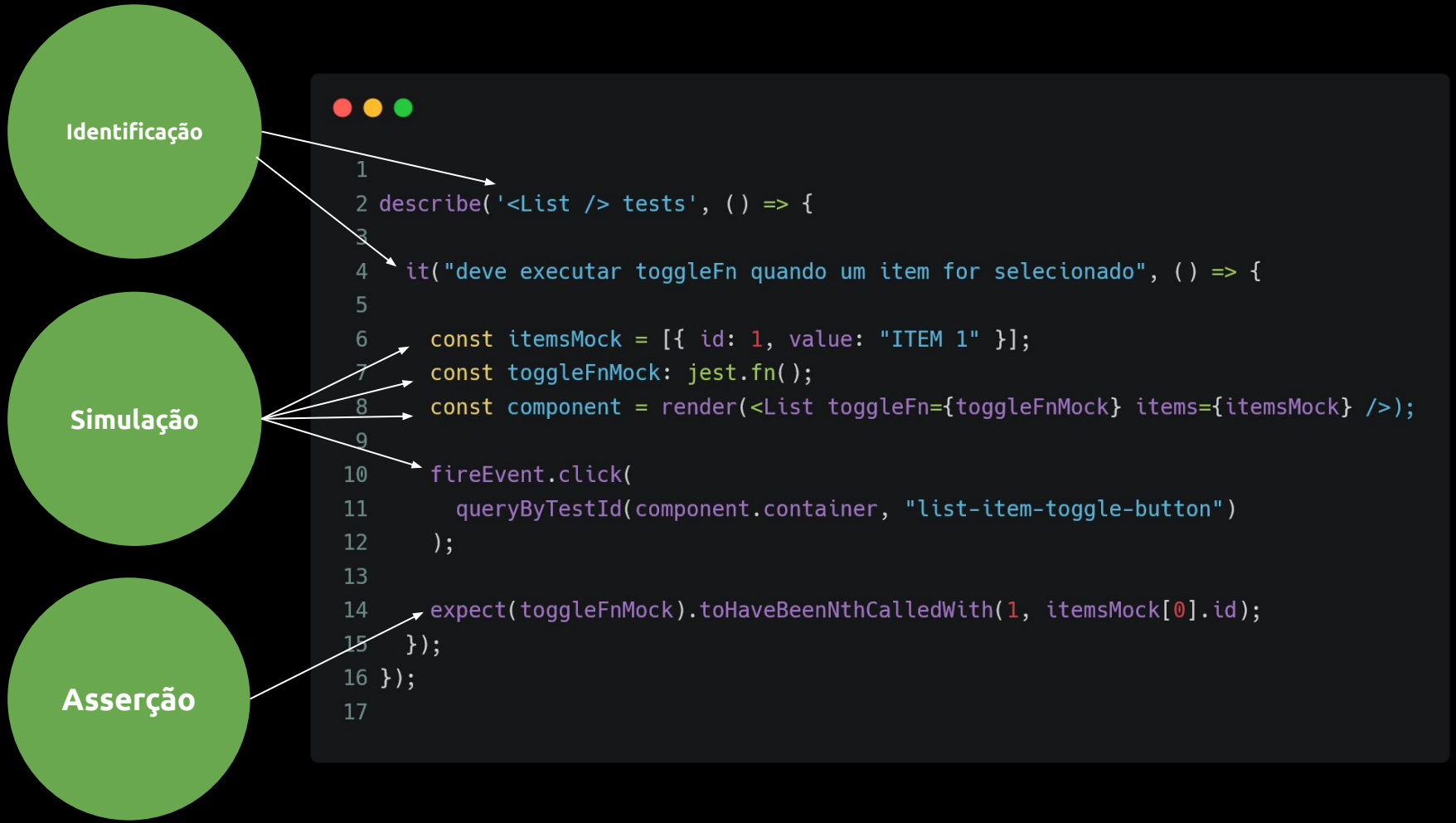
Simular o ambiente real e as interações do objeto a ser testado

Etapas de uma unidade de teste

Asserção

Fazer asserções no objeto a ser testado baseadas no ambiente e interações simuladas

```
describe('<List /> tests', () => {

  it("deve executar toggleFn quando um item for selecionado", () => {

    const itemsMock = [{ id: 1, value: "ITEM 1" }];
    const toggleFnMock: jest.fn();
    const component = render(<List toggleFn={toggleFnMock} items={itemsMock} />);

    fireEvent.click(
      queryByTestId(component.container, "list-item-toggle-button")
    );

    expect(toggleFnMock).toHaveBeenNthCalledWith(1, itemsMock[0].id);
  });
});

```


Identificação
Simulação
Asserção
describe('<List /> tests', () => {
it("deve executar toggleFn quando um item for selecionado", () => {
const itemsMock = [{ id: 1, value: "ITEM 1" }];
const toggleFnMock: jest.fn();
const component = render(<List toggleFn={toggleFnMock} items={itemsMock} />);
fireEvent.click(
queryByTestId(component.container, "list-item-toggle-button")
);
expect(toggleFnMock).toHaveBeenNthCalledWith(1, itemsMock[0].id);
});
});

Como saber o que testar?

Não tem mágica.

É extremamente necessário ler e entender o código, especificações e regras de negócio

As ferramentas oferecem comandos para

análise de cobertura de código

```
jest  --coverage
```

```
PASS  src/components/ListItem/__specs__/ListItem.spec.js
PASS  src/components/List/__specs__/List.spec.js
PASS  src/components/Button/__specs__/Button.spec.js
```

File	% Stmts	% Branch	% Funcs	% Lines	Uncovered Line #s
All files	65.79	80.77	66.67	66.67	
src	0	0	0	0	
App.js	0	0	0	0	... 30,32,36,37,40
index.js	0	100	100	0	5
src/components/Button	100	100	100	100	
Button.jsx	100	100	100	100	
index.js	0	0	0	0	
src/components/List	100	0	100	100	
List.jsx	100	0	100	100	11
index.js	0	0	0	0	
src/components/ListItem	100	84.62	100	100	
ListItem.jsx	100	84.62	100	100	37,43
index.js	0	0	0	0	

```
Test Suites: 3 passed, 3 total
Tests:       17 passed, 17 total
Snapshots:   0 total
Time:        6.917s
Ran all test suites.

Watch Usage
 › Press f to run only failed tests.
 › Press o to only run tests related to changed files.
 › Press q to quit watch mode.
 › Press p to filter by a filename regex pattern.
 › Press t to filter by a test name regex pattern.
 › Press Enter to trigger a test run.
```

Porcentagem de cobertura do projeto

Linhas não cobertas pelos testes

Bom, é isso.

Vamos a alguns exemplos práticos

Obrigado :)