

# DBStart

HTML, CSS e TypeScript

Instrutora: Profa. Andréa Aparecida Konzen ([andrea.konzen@pucrs.br](mailto:andrea.konzen@pucrs.br))

*Material cedido gentilmente pelo Prof. Júlio Pereira Machado*



# Teste de Software

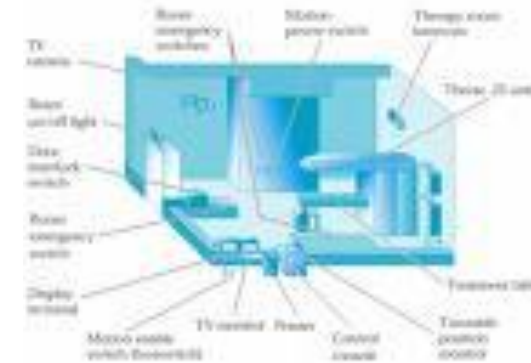


# Software e qualidade

- Software é um dos mais variados e complexos produtos produzidos de forma regular.
- Requisitos de qualidade dependentes:
  - Do domínio da aplicação;
  - Ambiente de execução;
  - Público alvo;
  - Etc.
- Exige técnicas sofisticadas e específica para cada produto desenvolvido.

# Software e qualidade

- Ariane 5:
  - Explosão em seu primeiro voo.
  - Causado pelo reuso de algumas partes de código de seu predecessor sem verificação adequada.
- Therac-25:
  - Máquina de terapia de radiação.
  - Devido a um erro de software, seis pessoas morreram de overdose.
- Pentium FDIV:
  - Erro de projeto na unidade de divisão de ponto-flutuante.
  - Intel foi forçada a oferecer substituição de todos os processadores defeituosos.



# Software e qualidade

- Por que é tão difícil garantir a qualidade do software?
  - Porque é algo dinâmico.
  - Porque envolve pessoas.
  - Porque as regras de negócio podem ser complexas.
  - Porque as tecnologias mudam rapidamente.
  - Porque as equipes mudam a toda hora.
  - Porque novas necessidades surgem a cada momento.
  - ...

# Software e qualidade

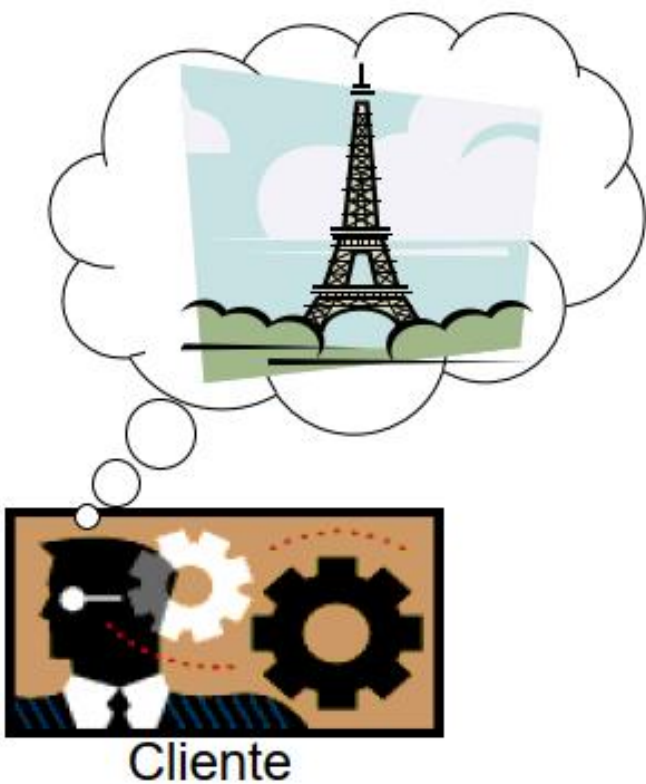


# Validação e Verificação

- **Validação** – o processo de avaliar o software ao final de uma etapa de seu desenvolvimento para garantir adequação ao seu propósito (necessidades do seu usuário).
- **Verificação** – o processo de determinar se o produto de uma determinada fase de desenvolvimento do software está de acordo com seus requisitos (sua especificação).

# Validação

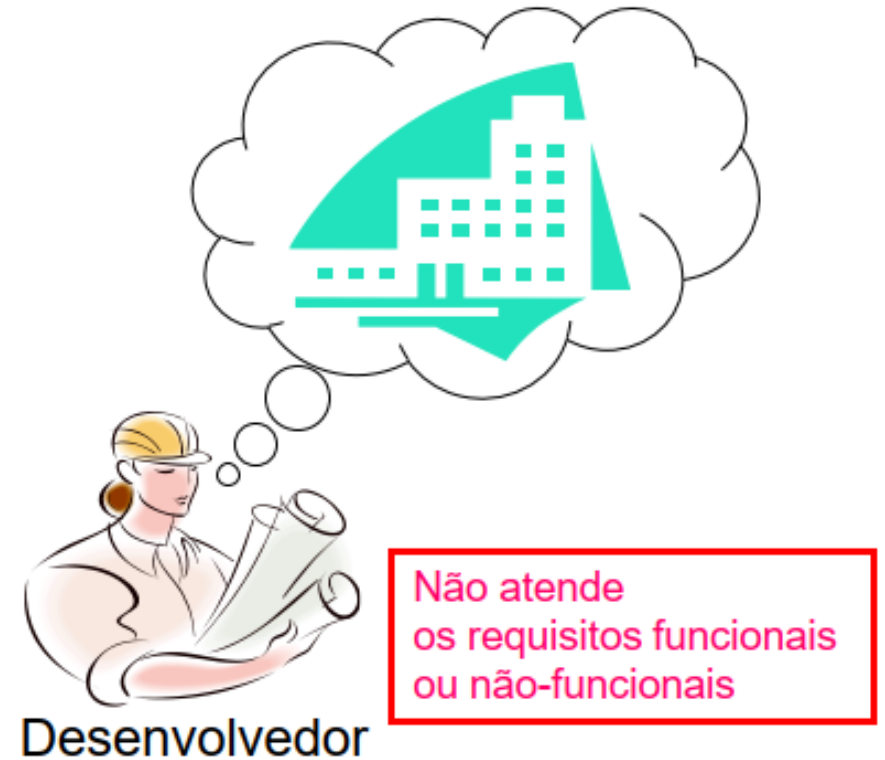
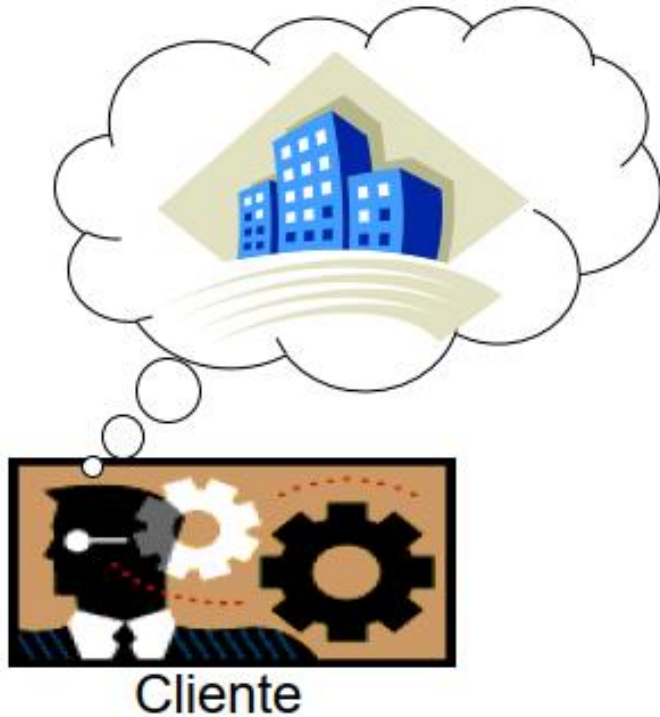
- “Estamos construindo o produto correto?”





# Verificação

- “Estamos construindo o produto corretamente?”



# Técnicas de V&V

- Existem dois tipos de técnicas de validação e verificação:



# Técnicas de V&V

- **Técnicas de V&V estáticas** não requerem que o sistema de software seja executado.
  - Exemplo: revisões de código
- **Técnicas de V&V dinâmicas** requerem trabalhar com uma representação executável do sistema de software.
  - Exemplo: teste

# Terminologia

- O termo *bug* é utilizado informalmente

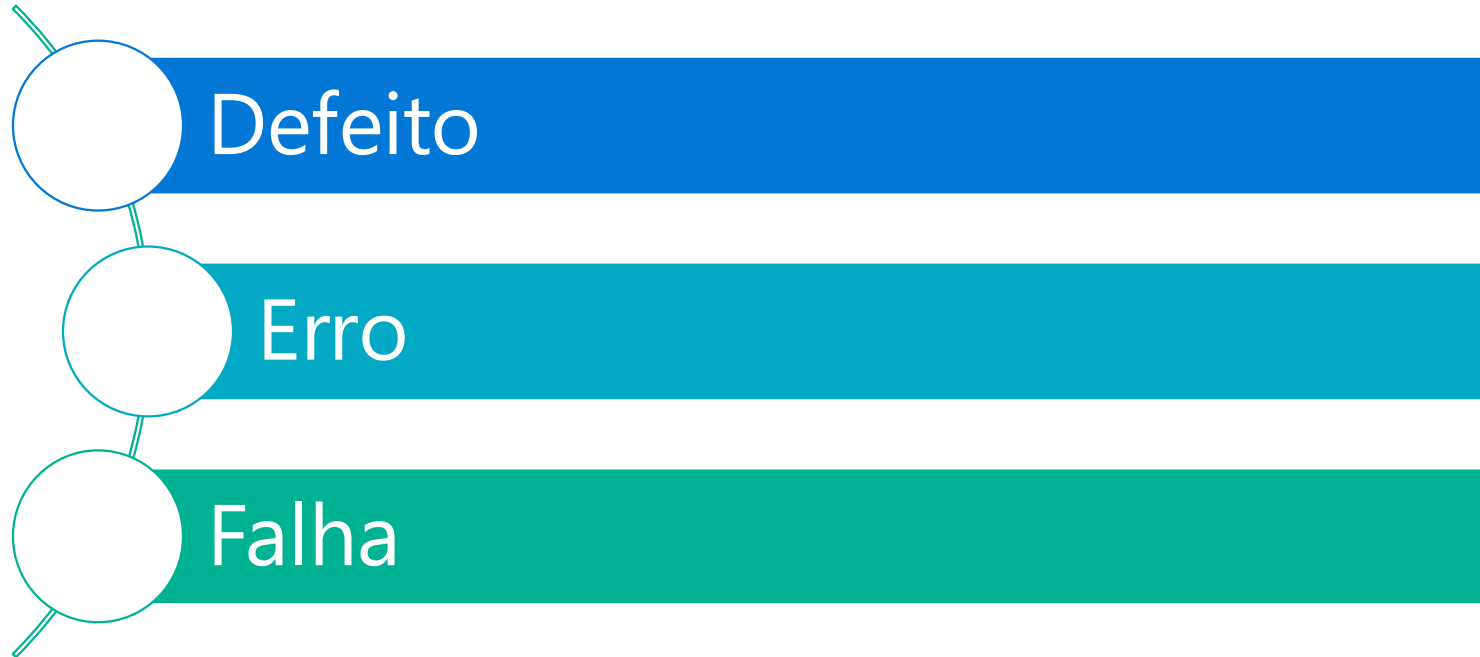


“It has been just so in all of my inventions. The first step is an intuition, and comes with a burst, then difficulties arise—this thing gives out and *[it is]* then that '**Bugs**'—as such little faults and difficulties are called—show themselves and months of intense watching, study and labor are requisite. . .” – Thomas Edison



“an analyzing process must equally have been performed in order to furnish the Analytical Engine with the necessary operative data; and that herein may also lie a possible source of **error**. Granted that the actual mechanism is unerring in its processes, the cards may give it wrong orders.” – Ada, Countess Lovelace (notas sobre Babbage’s Analytical Engine)

# Terminologia



# Terminologia

- **Defeito (*failure*):** é a manifestação externa do estado de erro quando este interfere na saída gerada.
- **Erro (*error*):** é o estado interno que diverge do estado correto.
- **Falha (*fault*):** problema no código que, ao ser executado, leva a um erro.

# Terminologia

- A execução de um teste leva a detecção de defeitos.
- A depuração do sistema permite então detectar o estado de erro.
- E levar, possivelmente, a descoberta da falha.

# Exemplo 1

Um programa deve imprimir um cupom fiscal com o valor total da compra. Executando-se um caso de teste percebe-se um defeito:

- o valor da soma está incorreto.

A depuração mostra um estado de erro:

- o preço do último produto não está sendo somado.

Uma análise do código detecta a falha:

- o método que percorre a lista de produtos está interpretando erroneamente o tamanho da lista de itens de venda.



# Exemplo 2

```
function quantidadeZeros(lista) {  
    let contagem = 0;  
    for(let i = 1; i < lista.length; i++) {  
        if (lista[i] === 0) {  
            contagem++;  
        }  
    }  
    return contagem;  
}
```

```
console.log(quantidadeZeros([2,7,0]));  
console.log(quantidadeZeros([0,2,7]));
```

Teste: [ 2, 7, 0 ]  
Esperado: 1  
Atual: 1

Teste: [ 0, 2, 7 ]  
Esperado: 1  
Atual: 0

# Exemplo 2

```
function quantidadeZeros(lista) {  
    let contagem = 0;  
    for(let i = 1; i < lista.length; i++) {  
        if (lista[i] === 0) {  
            contagem++;  
        }  
    }  
    return contagem;  
}
```

```
console.log(quantidadeZeros([2,7,0]));  
console.log(quantidadeZeros([0,2,7]));
```

Erro:  $i$  é 1 na primeira iteração.  
Erro se propaga para a variável *contagem*.  
Defeito: contagem é 0 no comando *return*.

Teste: [ 0, 2, 7 ]  
Esperado: 1  
Atual: 0

# Teste de software

- Quais os objetivos do teste?
  - Encontrar e prevenir defeitos;
  - Conhecer o grau de qualidade de um produto;
  - Prover informações para tomadas de decisões;
  - Validade e verificar produtos;
  - Etc.



# Categorias de testes



# Categorias de testes

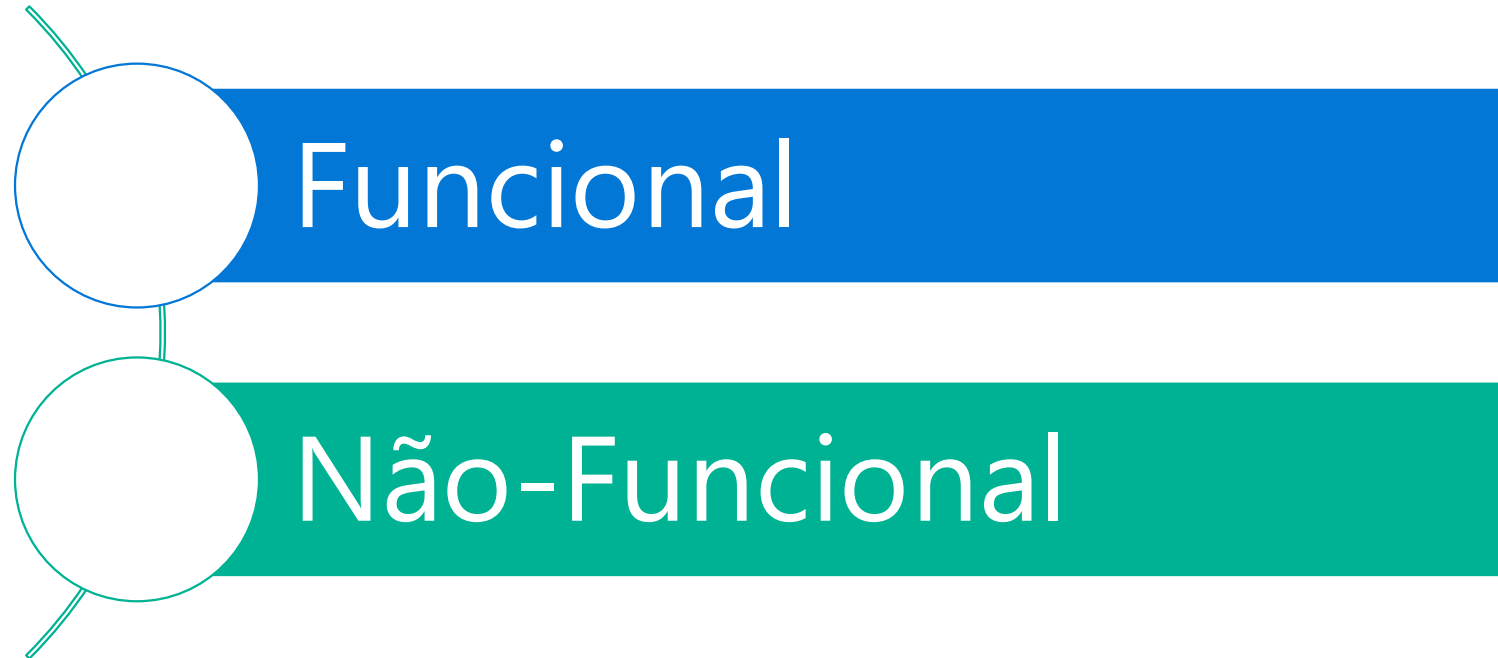
**Testes estatísticos** são utilizados para testar o desempenho e a confiabilidade do programa e como ele se comporta sob condições operacionais.

- Exemplo: verificar o comportamento de um servidor *web* com um grande número de acessos simultâneos.

**Testes de defeitos** se destinam a encontrar inconsistências entre o programa e sua especificação.

- Exemplo: verificar se a resposta de um servidor *web* está correta para uma determinada requisição.

# Categorias de testes



# Categorias de testes

- **Testes funcionais** são aqueles focados no teste das principais funcionalidades do sistema.
- **Testes não-funcionais** são aqueles focados no teste de outras características tais como tempo de resposta, quantidade de memória utilizada, e outros aspectos bastante relacionados ao desempenho e segurança do sistema.

# Níveis de testes

- De unidade
- De integração
- De sistema
- De aceitação
- De regressão
- De fumaça
- De exploração
- De robustez
- De desempenho
- De segurança
- De instalação
- De implantação
- De compatibilidade
- De concorrência
- De recuperação
- ...



# Níveis de testes

## Teste de unidade ou unitário:

- Busca identificar defeitos na lógica e implementação de cada “módulo” de software isoladamente.
- Possíveis unidades: funções, procedimentos, métodos, classes.

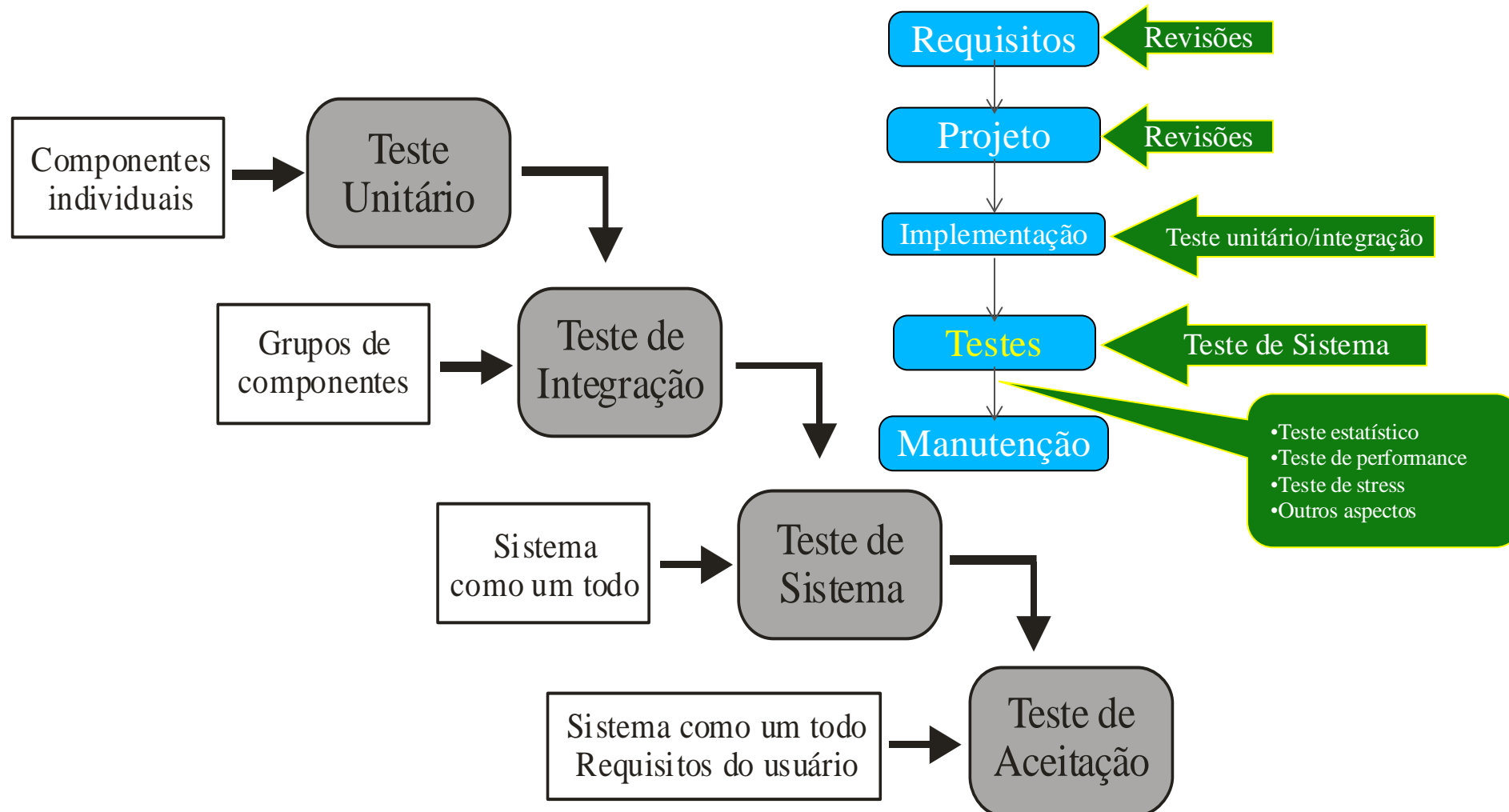
## Teste de integração:

- Visa descobrir defeitos em um grupo de “módulos” durante a integração da estrutura do programa.

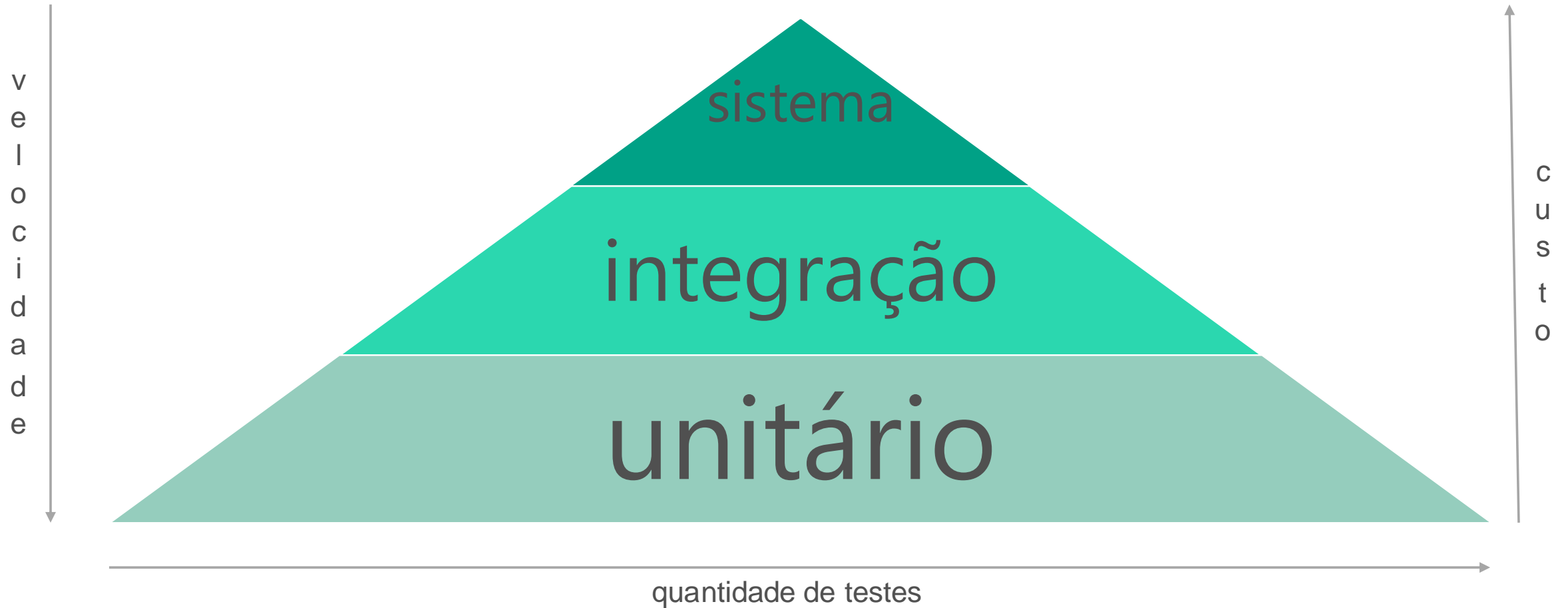
## Teste de sistema:

- Procura identificar defeitos nas funcionalidades do sistema como um todo.
- Envolve testes não-funcionais além dos funcionais.

# Níveis de testes



# Níveis de testes



# Testes em Javascript

- Grande variedade de bibliotecas e frameworks.
- Exemplos de frameworks para testes:
  - Jest <https://jestjs.io/>
  - Jasmine <https://jasmine.github.io/>
  - Mocha <https://mochajs.org/>
  - Cypress <https://www.cypress.io/>
  - Playwright <https://playwright.dev/>
- Exemplos de bibliotecas:
  - Chai <https://www.chaijs.com/> - asserções
  - Sinon <https://sinonjs.org/> - dublês (*spies, stubs, mocks*, etc)
  - Karma <https://karma-runner.github.io/> - executor

# JEST



# JEST

- Framework para realização de testes em JavaScript
- Suporta testes unitários, dublês, cobertura de código, etc
- Disponível em <https://jestjs.io/>



# JEST

- Qual executor de testes utilizar com JEST para TypeScript?
  - ts-jest <https://kulshekhar.github.io/ts-jest/>
  - @swc/jest <https://github.com/swc-project/jest>
  - etc

# Exemplo 1

```
export function somar(a,b) {  
    return a + b;  
}
```



# Jest: estrutura geral

```
describe('somar', () => {  
  it('deve retornar 2 para 1 + 1', () => {  
    expect(somar(1,1)).toBe(2);  
  });  
});
```

# Jest: estrutura geral

```
describe('somar', () => {
```

Define uma suíte/conjunto de testes.

```
  it('deve retornar 2 para 1 + 1', () => {
```

```
    expect(somar(1,1)).toBe(2);
```

```
  });
```

```
});
```

# Jest: estrutura geral

```
describe('somar', () => {
```

```
  it('deve retornar 2 para 1 + 1', () => {
```

```
    expect(somar(1,1)).toBe(2);
```

```
  });
```

```
});
```

Define um teste.

# Jest: estrutura geral

```
describe('somar', () => {  
  it('deve retornar 2 para 1 + 1', () => {  
    expect(somar(1,1)).toBe(2);  
  });  
});
```

Define o que está sendo verificado como esperado no teste através de um ou vários "matchers".

# Jest: *describe()*

- Uma suíte de testes é definida através da função global *describe()*.
- Parâmetros usuais:
  - Uma *string* descrevendo o que está sendo testado;
  - Uma *função* contendo a implementação da suíte de testes.
- Exemplo:

```
describe('somar', () => {  
  
});
```

# Jest: *it()/test()*

- A especificação de um teste é descrita pela função global *it()* ou *test()*
  - Ambos nomes são alias um do outro.
- Parâmetros usuais:
  - Uma *string* descrevendo o comportamento sendo testado;
  - Uma *função* contendo o corpo de implementação do teste individual.
- Exemplo:

```
it('deve retornar 2 para 1 + 1', () => {  
  });
```

```
test('deve retornar 2 para 1 + 1', () => {  
  });
```

# Jest: estrutura de um teste

- O código no corpo de um teste usualmente segue o padrão *arrange, act, assert*.
  - *Arrange*:
    - Estabelece as condições sobre as quais a unidade sob teste será exercitada.
    - É aqui que se configura dependências e dados do caso de teste.
  - *Act*:
    - Exercita a unidade sob teste.
    - Se apresenta como uma chamada de função ou método com os dados configurados na etapa anterior.
  - *Assert*:
    - Verifica que a unidade sob teste se comporta conforme esperado.

# Jest: estrutura de um teste

- Exemplo:

```
describe('somar', () => {  
  it('deve retornar 2 para 1 + 1', () => {  
    const a = 1;  
    const b = 1;  
    const resultadoEsperado = 2;  
    const resultadoAtual = somar(a,b);  
    expect(resultadoAtual).toBe(resultadoEsperado);  
  });  
});
```



# Jest: estrutura de um teste

- Exemplo:

```
describe('somar', () => {  
  it('deve retornar 2 para 1 + 1', () => {  
    const a = 1;  
    const b = 1;  
    const resultadoEsperado = 2;  
    const resultadoAtual = somar(a,b);  
    expect(resultadoAtual).toBe(resultadoEsperado);  
  });  
});
```

ARRANGE



# Jest: estrutura de um teste

- Exemplo:

```
describe('somar', () => {  
  it('deve retornar 2 para 1 + 1', () => {  
    const a = 1;  
    const b = 1;  
    const resultadoEsperado = 2;  
    const resultadoAtual = somar(a,b);  
    expect(resultadoAtual).toBe(resultadoEsperado);  
  });  
});
```



ACT

A red line connects the 'const resultadoAtual = somar(a,b);' line in the code block to the 'ACT' box.

# Jest: estrutura de um teste

- Exemplo:

```
describe('somar', () => {  
  it('deve retornar 2 para 1 + 1', () => {  
    const a = 1;  
    const b = 1;  
    const resultadoEsperado = 2;  
    const resultadoAtual = somar(a,b);  
    expect(resultadoAtual).toBe(resultadoEsperado);  
  });  
});
```

ASSERT

# Jest: *expectations*

- A estrutura de um teste busca observar se a unidade sob teste se comporta conforme esperado.
- Uma *expectation* é uma asserção sobre o estado esperado da unidade sob teste, a qual pode ser verdadeira ou falsa.
- Uma *expectation* é definida através da função global *expect()* que é o ponto de partida para a comparação entre o resultado atual e o resultado esperado através de um *matcher*...

# Jest: *matchers*

- *Matchers* são os elementos que permitem realizar asserções sobre o estado esperado de um teste.
- As APIs de teste possuem dezenas de métodos diferentes com *matchers* essenciais e ainda permitem a implementação de extensões.
  - Exemplos: `toBe()`, `toEqual()`, `toBeNull()`, `toBeFalsy()`, `toBeGreaterThan()`, `toBeCloseTo()`, `toMatch()`, ...
- Saiba mais: <https://jestjs.io/docs/expect>

# Exemplo 2

- Código a ser testado:

```
export function quantidadeZeros(lista) {  
  let contagem = 0;  
  for(let i = 1; i < lista.length; i++) {  
    if (lista[i] === 0) {  
      contagem++;  
    }  
  }  
  return contagem;  
}
```



# Exemplo 2

- Código do teste:

```
describe('quantidadeZeros', () => {  
  it('deve retornar 0 para array []', () => {  
    const array = [];  
    const quantidadeEsperada = 0;  
    const quantidadeAtual = quantidadeZeros(array);  
    expect(quantidadeAtual).toBe(quantidadeEsperada);  
  });  
  it('deve retornar 0 para array [1]', () => {  
    const array = [1];  
    const quantidadeEsperada = 0;  
    const quantidadeAtual = quantidadeZeros(array);  
    expect(quantidadeAtual).toBe(quantidadeEsperada);  
  });  
  it('deve retornar 1 para array [0]', () => {  
    const array = [0];  
    const quantidadeEsperada = 1;  
    const quantidadeAtual = quantidadeZeros(array);  
    expect(quantidadeAtual).toBe(quantidadeEsperada);  
  });  
});
```

# Jest: *describe()*

- Blocos *describe()* podem ser aninhados para a definição de uma estrutura de testes hierárquicos.
- Exemplo:

```
describe('stringBinParaNumber', () => {  
  describe('dado uma string binária inválida', () => {  
  
    });  
  describe('dado uma string binária válida', () => {  
  
    });  
});
```

- Uma suíte é composta por um ou mais testes individuais...



# Exemplo 3

- Código a ser testado:

```
export function stringBinParaNumber(binString) {  
  if (!/^[01]+$/.test(binString)) {  
    throw new Error('String em formato inválido');  
  }  
  return parseInt(binString, 2);  
}
```

# Exemplo 3

- Código do teste:

```
describe('stringBinParaNumber', () => {  
  describe('dados uma string binária inválida', () => {  
    it('contendo caracteres não numéricos 0 ou 1 deve lançar exceção', () => {  
      expect(() => stringBinParaNumber('abc')).toThrow();  
    });  
    it('contendo nenhum caractere deve lançar exceção', () => {  
      expect(() => stringBinParaNumber('')).toThrow();  
    });  
  });  
  describe('dados uma string binária válida', () => {  
    it('deve retornar 0 para "0"', () => {  
      expect(stringBinParaNumber('0')).toBe(0);  
    });  
    it('deve retornar 0 para "00"', () => {  
      expect(stringBinParaNumber('00')).toBe(0);  
    });  
  });  
});
```

# Jest: *it()/test()*

- Uma suíte de testes organiza o processo de testes de uma coleção de casos de testes. Assim, um mesmo teste é realizado com dados de entrada diferentes e resultados esperados diferentes.
- A API fornece meios de evitar a duplicação de código, tal como a função *it.each(tabela)(descrição,função)*.
- Parâmetros usuais:
  - Um *array* de *array* que define uma tabela onde cada linha representa os dados fornecido a cada teste;
  - Uma *string* que representa a descrição do teste, suportando parâmetros de formatação tais como %s (para strings), %i (para números inteiros), etc;
  - Uma *função* contendo o corpo de implementação do teste individual.

# Exemplo 4


- Código do teste:

```
describe('somar', () => {  
  it.each([  
    [0,0,0],  
    [1,0,1],  
    [-1,-1,0],  
    [0,-1,1]  
  ])( 'deve retornar %i para %i + %i', (r,x,y) => {  
    expect(somar(x,y)).toBe(r);  
  });  
});
```

# Exemplo 4

- Código do teste:

```
describe('somar', () => {  
  it.each([  
    [0,0,0],  
    [1,0,1],  
    [-1,-1,0],  
    [0,-1,1]  
  ])(  
    'deve retornar %i para %i + %i', (r,x,y) => {  
      expect(somar(x,y)).toBe(r);  
    });  
});
```



# Jest: *before...*, *after...*

- A implementação da suíte de testes pode fazer uso de funções globais aplicadas dentro de etapas da execução dos testes:
  - *beforeEach()* – executada antes de cada teste
  - *afterEach()* – executada após cada teste
  - *beforeAll()* – executada uma única vez antes de todos os testes
  - *afterAll()* – executada uma única vez após todos os testes
- São funções úteis para organizar a configuração dos testes, evitando a duplicação de código.

# Exemplo 5

- Classe a ser testada:
  - Parâmetro de *saldoinicial* do construtor deve ser  $\geq 0$ .
    - Gera exceção em caso de falha.
  - Parâmetro de *valor* para depositar/sacar deve ser  $> 0$ .
    - Gera exceção em caso de falha.

ContaCorrente
- saldo: number
+ constructor(saldoinicial: number) + depositar(valor: number) + sacar(valor: number) + get saldo(): number

# Exemplo 5

- Código do teste:

```
describe('ContaCorrente', () => {  
  let conta;  
  describe('construtor', () => {  
  
  });  
  describe('depositar', () => {  
    beforeEach(() => {  
      const saldoInicial = 0;  
      conta = new ContaCorrente(saldoInicial);  
    });  
    it('dado uma conta com saldo 0 deve depositar o valor 0.1', () => {  
    });  
    it('dado uma conta com saldo 0 deve depositar o valor 0.1 e depois 0.2', () => {  
    });  
  });  
});
```

Recria a conta corrente antes de cada teste para evitar efeitos colaterais do teste anterior.




# Exemplo 5

- Código do teste:

```
describe('depositar', () => {  
  beforeEach(() => {  
    const saldoInicial = 0;  
    conta = new ContaCorrente(saldoInicial);  
  });  
  
  it('dado uma conta com saldo 0 deve depositar o valor 0.1 e depois 0.2', () => {  
    conta.depositar(0.1);  
    conta.depositar(0.2);  
    expect(conta.saldo).toBe(0.3); //aqui irá falhar  
    expect(conta.saldo).toBeCloseTo(0.3, 2);  
  });  
});
```

Números em ponto flutuante NÃO devem ser testados de forma absoluta!  
Utilizar o *matcher toBeCloseTo()*.

# Jest: teste assíncrono

- É bastante comum encontrar APIs com código assíncrono.
- Jest possui suporte adequado para a execução de testes assíncronos:
  - *Callbacks*
  - *Promises*
  - *Async/await* 
- Saiba mais: <https://jestjs.io/docs/asynchronous>

# Exemplo 1

- Código a ser testado:

```
export async function somar(a, b) {  
  await new Promise(resolve => setTimeout(resolve, 1000));  
  if (typeof a !== 'number' || typeof b !== 'number') {  
    throw new Error('Argumentos devem ser valores numéricos');  
  }  
  return a + b;  
}
```

# Exemplo 1

- Código do teste: *promise* decidida com sucesso

```
it('deve retornar 2 para 1 + 1', async () => {  
    const resultado = await somar(1,1);  
    expect(resultado).toBe(2);  
});  
it('deve retornar 2 para 1 + 1', async () => {  
    expect.assertions(1);  
    await expect(somar(1,1)).resolves.toBe(2);  
});
```

# Exemplo 1

- Código do teste: *promise* decidida com falha

```
it('deve gerar exceção para 1 + "a"', async () => {
```

```
  expect.assertions(1);
```

```
  try {
```

```
    await somar(1, 'a');
```

```
  } catch (error) {
```

```
    expect(error).toBeInstanceOf(Error);
```

```
  }
```

```
});
```

```
it('deve gerar exceção para 1 + "a"', async () => {
```

```
  expect.assertions(1);
```

```
  await expect(somar(1, 'a')).rejects.toBeInstanceOf(Error);
```

```
});
```

Verifica se um determinado número de asserções foi executado no teste.

# Teste unitário X dependências

- Teste unitário pressupõe o teste de uma unidade sem dependências com outras unidades do sistema.
- Mas como realizar esse tipo de teste se existem dependências necessárias na implementação do sistema?
  - Exemplo: uma função necessita acessar a *web* para consultar dados.

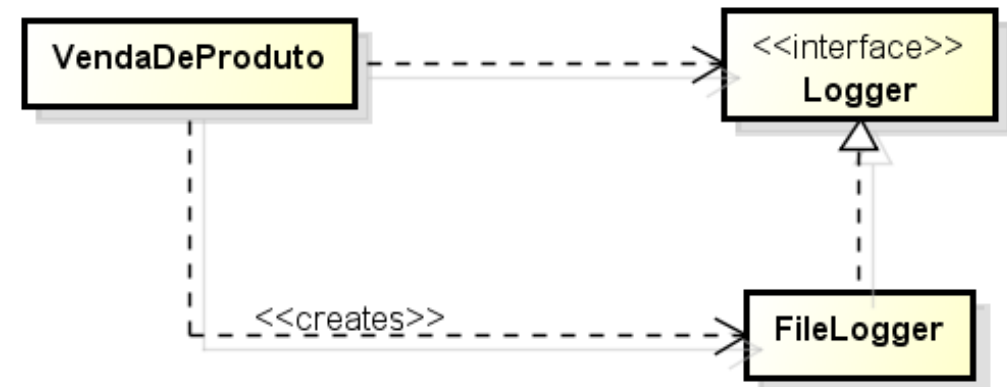
# Teste unitário X dependências

- Para realizar um teste unitário, é necessário “quebrar” de alguma forma as dependências.
- Espera-se que o código a ser testado tenha “boa qualidade” e “permita ser facilmente testado”.
- Então, como um código pode ter a característica de ser facilmente testado na presença de dependências?
- Solução: usar um padrão de Injeção de Dependências (*DI – Dependency Injection*)
- Saiba mais:
  - <https://martinfowler.com/articles/injection.html>
  - <https://martinfowler.com/articles/dipInTheWild.html>

# Injeção de dependências

- Exemplo:
  - Em um sistema é necessário gerar a gravação de um *log* toda vez que se realiza a venda de um produto.

```
class VendaDeProduto {  
    ...  
    venderProduto(produto) {  
        ...  
        logger = new FileLogger('log.txt');  
        logger.log(produto);  
    }  
    ...  
}
```





# Injeção de dependências

- Problemas:
  - Como fica a questão de mudança do nome do arquivo a ser realizado o log?
  - Como podemos trocar o mecanismo de log para um banco de dados, por exemplo?
  - Como realizar um teste unitário da classe *VendaDeProduto* sem a necessidade de realizar um operação de *log* "de verdade"?

```
class VendaDeProduto {  
    ...  
    venderProduto(produto) {  
        ...  
        logger = new FileLogger('log.txt');  
        logger.log(produto);  
    }  
    ...  
}
```

# Injeção de dependências

- Resumindo:
  - A classe *VendaDeProduto* sabe demais sobre a classe *FileLogger*, ou seja, existe uma dependência muito grande e indesejável!

```
class VendaDeProduto {  
    ...  
    venderProduto(produto) {  
        ...  
        logger = new FileLogger('log.txt');  
        logger.log(produto);  
    }  
    ...  
}
```

# Injeção de dependências

- Refatorando:
  - A classe *VendaDeProduto* irá receber um objeto sem a necessidade de se preocupar como e onde foi criado.
- Pontos de injeção:
  - Via **construtor** – utilizar um parâmetro no construtor para receber a referência ao objeto injetado.
  - Via setter – utilizar uma propriedade do tipo *set* para receber a referência ao objeto injetado.

```
class VendaDeProduto {  
    #logger;  
    ...  
    constructor(umLogger) {  
        ...  
        this.#logger = umLogger;  
    }  
    venderProduto(produto) {  
        ...  
        this.#logger.log(produto);  
    }  
    ...  
}
```

# Injeção de dependências

- Refatorando:
  - A classe *VendaDeProduto* irá receber um objeto sem a necessidade de se preocupar como e onde foi criado.
- Pontos de injeção:
  - Via construtor – utilizar um parâmetro no construtor para receber a referência ao objeto injetado.
  - Via **setter** – utilizar uma propriedade do tipo *set* para receber a referência ao objeto injetado.

```
class VendaDeProduto {  
    #logger;  
    ...  
    set logger(umLogger) {  
        ...  
        this.#logger = umLogger;  
    }  
    venderProduto(produto) {  
        ...  
        this.#logger.log(produto);  
    }  
    ...  
}
```

# Dublês

- Um dublê é uma estrutura de código que toma o lugar de uma dependência real no momento da realização de um teste.
- Exemplo:
  - uma função necessita acessar a *web* para consultar dados;
  - um dublê “simula” o acesso à *web* sem realmente acessar qualquer tipo de conexão de rede.

# Dublês

- Terminologia envolve a diferenciação entre muitos termos:
  - *Double*
  - *Spy*
  - *Stub*
  - *Mock*
  - Etc.
- Cuidado! Não existe consenso para a definição técnica de cada termo.
  - Não vamos entrar nesse nível de detalhe!
  - Vamos usar a API do Jest sem essa preocupação.
- Saiba mais: <https://martinfowler.com/bliki/TestDouble.html>

# Jest: dublês

- Na API do Jest, são utilizados os termos *mock* e *spy* de forma intercambiável. As construções permitem:
  - Substituir a implementação de funções, métodos, classes e módulos por dublês;
  - Capturar chamadas de funções e métodos para observar se e quantas vezes foram chamados;
  - Capturar chamadas de funções e métodos para observar os parâmetros recebidos;
  - Etc.
- Saiba mais:
  - <https://jestjs.io/docs/jest-object>
  - <https://jestjs.io/docs/mock-function-api>

# Jest: duplões para funções

- *jest.fn()* é responsável por criar duplões para funções.
  - Possui um parâmetro opcional que recebe uma implementação desejada para o duplão.
- Duplões podem injetar valores para testes via métodos:
  - *mockReturnValue()* – valor a ser retornado pelo duplão sempre quando for chamado
  - *mockResolvedValue()* – valor assíncrono a ser retornado pelo duplão sempre quando for chamado
  - *mockRejectedValue()* – valor rejeitado assíncrono a ser retornado pelo duplão sempre quando for chamado
  - Etc.



# Jest: dublês para funções

- Dublês possuem *matchers* customizados para observar o que aconteceu com a função chamada:
  - *toHaveBeenCalled()* – verificar se uma função dublê foi chamada
  - *toHaveBeenCalledTimes()* – verificar se uma função dublê foi chamada um determinado número de vezes
  - *toHaveBeenCalledWith()* – verificar se uma função dublê foi chamada com argumentos específicos
  - Etc.

# Exemplo 2

- Código a ser testado:

```
export async function converterRealPara(codigoMoeda, valor) {  
  //Obs.:  
  //chamada a um serviço que não existe  
  //retornaria um objeto Moeda {codigo, nome, cotacao}  
  const resposta = await fetch(`http://servicobancocentral.com.br/codigo=${codigoMoeda}`);  
  if (resposta.ok) {  
    const moeda = await resposta.json();  
    return valor * moeda.cotacao;  
  } else {  
    throw new Error(`GET status: ${resposta.status}`);  
  }  
}
```

Fetch é uma dependência externa que deve ser substituída por um dublê nos testes unitários.

# Exemplo 2

- Código do teste: configuração do dublê da função *fetch*

```
const moedaEsperada = {  
  codigo: 'USD',  
  nome: 'Dólar dos Estados Unidos',  
  cotacao: 0.1865881  
};  
globalThis.fetch = jest.fn().mockResolvedValue({  
  ok: true,  
  json: () => Promise.resolve(moedaEsperada)  
});
```

# Exemplo 2

- Código do teste: *promise* decidida com sucesso

```
it('realiza a conversão de valor da moeda de BRL para USD', async () => {  
  const resultado = await converterRealPara(codigoMoeda, valorParaConversao);  
  expect(resultado).toBeCloseTo(valorConvetidoEsperado);  
  expect(fetch).toHaveBeenCalledTimes(1);  
  expect(fetch).toHaveBeenCalledWith(`http://servicobancocentral.com.br/codigo=${codigoMoeda}`);  
});
```

# Exemplo 2

- Código do teste: *promise* decidida com falha

```
it('gera uma exceção em caso de falha de rede', async () => {  
  fetch.mockRejectedValueOnce(new TypeError('error'));  
  await expect(converterRealPara(codigoMoeda,  
valorParaConversao)).rejects.toBeInstanceOf(TypeError);  
  expect(fetch).toHaveBeenCalledTimes(1);  
  expect(fetch).toHaveBeenCalledWith(`http://servicobancocentral.com.br/codigo=${codigoMoeda}`);  
});
```

Substitui o resultado do  
dublê padrão por um  
novo somente para o  
teste atual.

# Exemplo 2

- Código do teste: *promise* decidida com falha

Substitui o resultado do dublê padrão por um novo somente para o teste atual.

```
it('gera uma exceção com código de moeda inválido', async () => {  
  fetch.mockResolvedValueOnce({  
    ok: false,  
    status: 404  
  });  
  await expect(converterRealPara('ZZZ',  
valorParaConversao)).rejects.toBeInstanceOf(Error);  
  expect(fetch).toHaveBeenCalledTimes(1);  
  expect(fetch).toHaveBeenCalledWith(`http://servicobancocentral.com.br/codigo=ZZZ`)  
;  
});
```

# Jest: dublês para módulos

- *jest.mock()* é responsável por criar dublês para módulos.
- IMPORTANTE:
  - *jest.unstable\_mockModule()* deve ser utilizado para a criação de dublês de módulos EcmaScript enquanto a API não for estabilizada em uma versão final.
  - Módulo dublê deve ser carregado via importação dinâmica *await import()*.
- Serão criados dublês padrão para cada unidade presente no módulo alvo.
  - Jest permite também que sejam criados dublês para um subconjunto do módulo, enquanto o restante mantém sua implementação original.

# Exemplo 3

- Código a ser testado:

```
import { buscarPorCodigo } from './moedasrepositorio';

export class Conversor {
  async converterRealPara(codigoMoeda, valor) {
    const moeda = await buscarPorCodigo(codigoMoeda);
    if (!moeda) {
      throw new Error('Código de moeda inexistente');
    }
    return valor * moeda.cotacao;
  }
}
```

Dependência de um  
módulo externo.



# Exemplo 3

- Código do teste: configuração do dublê do módulo *moedasrepositorio*

```
it('realiza a conversão de valor da moeda de BRL para USD', async () => {  
  //Dublê para o método de busca  
  const mockBuscarPorCodigo = jest.fn(() => moedaEsperada);  
  //Dublê para o módulo  
  jest.unstable_mockModule('./moedasrepositorio', () => {  
    return {  
      buscarPorCodigo: mockBuscarPorCodigo  
    };  
  });  
  await import('./moedasrepositorio');  
  const { Conversor } = await import('./conversor');  
  ...  
});
```

# Exemplo 3

- Código do teste: *promise* decidida com sucesso

```
it('realiza a conversão de valor da moeda de BRL para USD', async () => {  
  ...  
  //Conversor a ser testado com o dublê  
  const conversorComMock = new Conversor();  
  //Realizar a ação do teste  
  const resultado = await conversorComMock.converterRealPara(codigoMoeda,  
valorParaConversao);  
  expect(resultado).toBeCloseTo(valorConvetidoEsperado);  
  expect(mockBuscarPorCodigo).toHaveBeenCalledTimes(1);  
  expect(mockBuscarPorCodigo).toHaveBeenCalledWith(codigoMoeda);  
});
```