

# Tech-Talk Testes Unitários

| *How I Learned to Stop Worrying and Love Tests*

⚠ Atenção, as definições apresentadas podem variar de acordo com o autor

[alexandre.hausen@3778.care](mailto:alexandre.hausen@3778.care)

# Agenda

- Introdução e conceitos
- Q&As vs. devs?
- + Conceitos
  - unidade
  - *stub, fake e mock*
- Exemplos
- ++ Conceitos
  - critérios
  - aplicação dos critérios



# No início era o caos

- sem ferramentas de código (eg linters/formatadores/analisadores estáticos)
- sem controle de versão
- sem testes automatizados

E portanto sem CI, muito menos CD.

# O tempo passa, as pessoas aprendem (as vezes)

- ferramentas de código integradas às IDEs/CLI
- Git é padrão de fato

# E o teste?

## Pra que serve?

| Encontrar defeitos. 

A gente só corrige os defeitos que conhece.

## Pra que não serve?

| Para dizer que o sistema **não** tem defeitos.

Pode ser que os testes **ainda** não tenham encontrado defeitos.

# O que é teste?

- 📘 É a análise dos **artefatos e comportamentos** do software através dos processos de **verificação e validação**.

## E na prática?

1. verificação: "Nós construímos corretamente o software?" (*ie* ele implementa os requisitos)
2. validação: "Nós construímos o software correto?" (*ie* ele atende às expectativas do cliente)

# Como verificamos e validamos?

Da mesma forma, respondendo a pergunta:

- | Para uma dada entrada, a saída obtida é igual a saída esperada?

# Quem diz o que é o esperado?

- pessoa: especialista, analista de negócio, cliente, *product owner*, ...
- documento: requisitos, *user stories*, regras de negócio, diagramas, leis, normas, ...

# **Por que o desenvolvedor testa se temos Q&A?**

Porque os propósitos são diferentes e complementares.

## **Testes de sistema (Q&A)**

O objetivo é encontrar defeitos (funcionais e não-funcionais) no sistema usando testes manuais e ferramentas de automação.

Mas sempre com uma versão completa do sistema (*release*, *pre-release*, *prd*, *dev* ou *local*).

# Testes unitários (desenvolvedor)

O objetivo é encontrar defeitos (funcionais) nas unidades.

E também:

- orienta o desenvolvimento unidades novas
- auxilia a correção de *bugs*
- evita regressão (*ie* que o *bug* volte a ocorrer)
- dá confiança para realizar melhorias (*aka refactoring*)
- executado automaticamente (ambientes local e CI)

# O que é uma unidade?

É a menor parte do software a qual podemos verificar uma funcionalidade.

Em geral: uma função, um método, uma classe.



*User story:* Como um Analista de Cobrança, gostaria de saber quais pessoas de uma lista estão inadimplentes\*, para emitir cobranças.

\* débito não pago há 30 dias ou mais.

```

"""Caso de uso para listar pessoas inadimplentes"""
class Debito:
    def __init__(self, data, pago):
        self.data = data
        self.pago = pago

class ListaInadimplentesUseCase:
    def __init__(self, debito_repo):
        self.debito_repo = debito_repo

    def execute(self, cpfs):
        rows = self.debito_repo.lista_debitos_por_cpfs(cpfs)
        inadimplentes = set()
        for row in rows:
            if row['data'] and row['pago'] is not None: # 1
                debito = Debito(row['data'], row['pago'])
                dias_em_debito = date.today() - debito.data
                if not debito.pago and dias_em_debito.days > 30: # 2
                    inadimplentes.add(row['cpf'])
        return list(inadimplentes)

```

Executar exemplo python

```
# source ~/.virtualenvs/venv/bin/activate
# python run_app.py 15150373044 21426399090 39328534070 41513558048 57991403033
```

*Requisito funcional:* O sistema deve alertar o usuário ao digitar um CPF inválido.

```
function isValidCPF(cpf: string): boolean {
  cpf = cpf.replace(/\D/g, "");
  if (cpf.length !== 11) return false;
  const cpfDigits: number[] = cpf.split("").map((item) => +item);
  const rest = (count: number) =>
    ((cpfDigits
      .slice(0, count - 12)
      .reduce((soma, item, index) => soma + item * (count - index), 0) *
      10) %
     11) %
    10;
  return rest(10) === cpfDigits[9] && rest(11) === cpfDigits[10];
}
```

Inspirado em [gist.github.com](https://gist.github.com)

Executar exemplo typescript  
# npx ts-node ./src/index.ts 729.192.590-83

# Como testar a unidade?

Isolando-a do resto do sistema, substituindo as dependências e testando diferentes cenários de maneira determinística.

## E o que é uma dependência (no contexto de teste unitário)?

É tudo que você precisa configurar antes de executar o objeto do teste, *i.e.* aquilo que se quer testar (*System Under Test* ou SUT).

A dependência pode ser:

1. explícita: valores passados como entrada para o SUT (eg construtor, método ou função chamada)
2. implícita: instâncias construídas dentro do SUT e as fontes de não-determinismo

# Precisa isolar completamente para ser teste unitário?



Vale o bom senso, quanto mais isolado melhor você consegue controlar o teste da unidade, porém o *setup* das dependências será mais complicado.

# Como substituir as dependências?

## 1. Inversão de dependência

- Substituindo a instância passada por parâmetro
- Substituindo o criador da instância (eg *factory*)

## 2. Sobrescrever o objeto/classe

- *monkey patching* + *duck typing*
- *interface*/classe abstrata

## 3. Facilidades do *framework*/linguagem

# Substituir pelo quê?

Por algo que eu possa controlar o comportamento.

- *stub*
- *fake*
- *mock*



## Qual é a diferença entre *stub*, *fake* e *mock*?

- *stub*: uma implementação vazia (ou fixa) da dependência
- *fake*: um objeto com implementação simplificada (eg pseudo banco em memória)
- *mock*: um objeto/função com comportamento controlado (*ie* controle das expectativas de como deve ser chamado, dos valores de retorno e as exceções lançadas)

# Quais são os passos para testar uma unidade?

1. *setup* da unidade e das dependências

- definir retornos e exceções
- definir expectativas

2. executar objeto do teste com suas entradas

3. asserções

- comparar saída obtida com a esperada
- verificar as expectativas das dependências

# Exemplos de código

## 1. Exemplo de teste usando classe Stub

- `test_lista_cpfs_inadimplentes_stub()`

## 2. Exemplo de teste usando classe Fake

- `test_lista_cpfs_inadimplentes_fake()`

## 3. Exemplo de teste usando Mock

- `test_lista_cpfs_inadimplentes_mock()`

Executar testes básicos python:

```
# source ~/.virtualenvs/venv/bin/activate  
# python run_unit_tests.py
```

Atenção para não testar o *stub/fake/mock* ao invés da unidade.

```
def test_list_cpfs_inadimplentes():
    # setup
    debito_repo = Mock()
    debito_repo.lista_debitos_por_cpfs.return_value = [
        {'id': 1, 'cpf': '15150373044', 'pago': False, 'data': date(2022, 1, 1)}
    ]

    # execução
    mock_return = debito_repo.lista_debitos_por_cpfs(['15150373044', '21426399090'])

    # expectativas e asserções
    assert mock_return == [
        {'id': 1, 'cpf': '15150373044', 'pago': False, 'data': date(2022, 1, 1)}
    ]
```

# E o não-determinismo?

Algumas fontes de não-determinismo:

- base de dados, serviços externos ou estado global/variáveis de classe/*static*
- `Date.now()` e `datetime.date.today()`
- `Math.random()` e `random.random()`
- concorrência e paralelismo

# Como tratar o não-determinismo?

Quando for uma dependência implícita:

- transformar em dependência explícita
- criar abstrações e *factories* que possam ser substituídas por classes *mocks* ou *fakes*
- usar recursos dos *frameworks*, eg:
  - python `@freeze_time`
  - js `jest.setSystemTime`

Quando se tratar de concorrência/paralelismo:

- 😢

# E os requisitos não-funcionais?

Desempenho, tempo de resposta, quantidade de conexões, etc?

Não são avaliados por testes unitários, mas podem ser avaliados por **testes de sistema** ou **testes de integração**.

# Quanto mais testes melhor!!!

🔥 *Hot take* 🔥

Opa, não é bem assim!

A quantidade de entradas possíveis de um sistema é grande demais (praticamente ilimitada).

Testes demais vão ter um alto custo de manutenção.

Portanto temos que escolher quais testes fazer.

Vamos dar preferência a "testes bons".

## O que faz um teste "bom"?

Testes bons são aqueles com maior probabilidade de revelar algum defeito.  
Vários testes que testam a mesma coisa tem pouco valor.

# Como saber se o teste é bom?

Tendo critérios de escolha, por exemplo:

- Critérios funcionais (*aka black-box*): derivados da funcionalidade
- Critérios estruturais (*aka white-box*): derivados da estrutura do código

Bons testes variam:

- cenários
- tipos de erros
- fluxos de execução

## Critérios Funcionais

Os testes são criados cobrindo as funcionalidades do sistema: especificação, requisitos funcionais, *user stories*, *use cases*, regras de negócio...

Algumas técnicas de critérios funcionais ajudam na escolha:

- partições em classe de equilalência (*ie* separar a entrada em categorias)
- análise de valor limite
- tabelas de decisão
- máquina de estados
- erros comuns (*string* vazia, *undefined*, *None*, data inválida, valores fora da faixa...)

```
executar testes básicos typescript  
# npm test
```

## Critérios Estruturais

Critérios estruturais medem % de cobertura do código (*white-box*).

- todas as funções/métodos
- todas as linhas
- todos os fluxos de controle (*branches*)
- ...

Ajudam a descobrir testes interessantes que não estão sendo feitos.

- 1 - grafos dos exemplos: app.dot e cpf.dot
- 2 - observar HTML de cobertura dos testes triviais (py e ts)
- 3 - descomentar o critério '--cov-branch' do run\_unit\_tests.py

# Antes & depois



00 Observar o incremento da cobertura nos arquivos HTML.

- grafo exemplo: app-simples.dot
- descomentar testes criados usando critérios estruturais em py e ts
- python: branch 1F e 2F
- typescript: branch 1T

# **Qual é a cobertura ideal?**

**A cobertura ideal:**

**TL;DR 80%**





Não tem fórmula mágica 😞

Vale o bom senso e XP

Referência: <https://testing.googleblog.com/2010/07/code-coverage-goal-80-and-no-less.html>

# OK, mas conseguimos cobertura de 100%! pronto?



Mesmo com 100% de cobertura\* ainda podemos melhorar usando critérios funcionais complementares.

\* em qualquer critério

## Lá e de volta outra vez

- exemplo final python
- exemplo final typescript

*That's all Folks!*