

# Teste de Software

UC: Gestão e Qualidade de Software

**Prof. Eliane Faveron Maciel**

UNIFACS  
ecossistema ânima

28 de março de 2024

# Overview

**1. Testes de Software**

**2. Testes de Unidade**

**3. Exemplos de teste**

**4. Referências**

## ... última aula ...

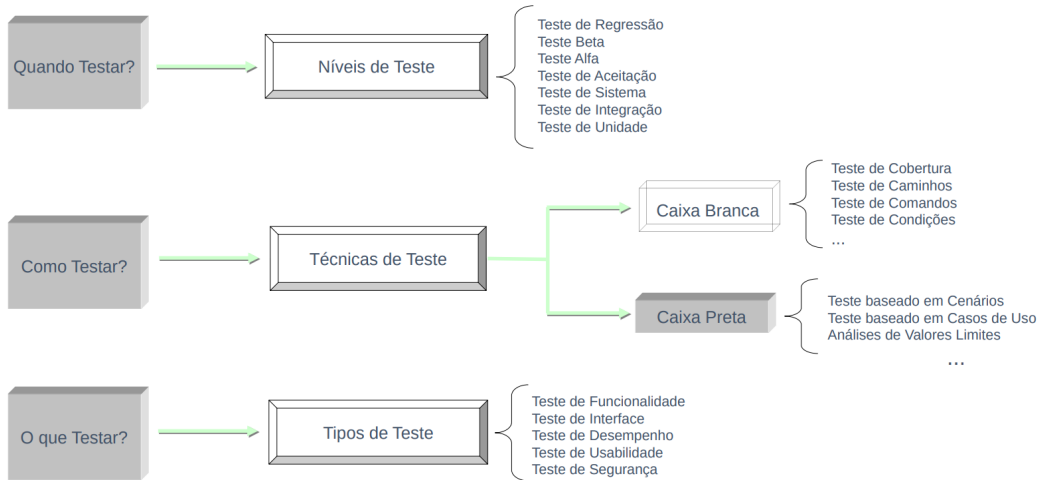
- Revisões técnicas formais
- Criação de Pull Request
- Definição da A3

# Testes de Software

---

## O que é?

O software é testado para revelar erros cometidos inadvertidamente quando ele foi projetado e construído. Uma estratégia de teste de componentes de software considera o teste de componentes individuais e a sua integração a um sistema em funcionamento.



# Níveis de Testes

1. Testes de Unidade
2. Testes de Integração
3. Testes de Sistema
4. Testes de Aceitação

# Estratégia para o sucesso do teste de software

No livro de [Pressman, 2021], cita que o teste de software terá sucesso se seguir os itens a seguir:

1. Especificar os requisitos do produto de uma maneira quantificável;
2. Definir os objetivos do teste;
3. Entender os usuários do software e desenvolverem um perfil para cada categoria de usuário;
4. Desenvolver um plano de teste;
5. Criar software “robusto” que seja projetado para testar-se a si próprio;
6. Usar revisões técnicas como filtro antes do teste;
7. Realizar revisões técnicas para avaliar a estratégia de teste e os próprios casos de teste;
8. Desenvolver abordagem de melhoria contínua para o processo de teste



# Testes de Unidade

---

# Teste de Unidade

*”O teste de unidade focaliza o esforço de verificação na menor unidade de projeto do software [Pressman, 2021].”*

O propósito principal dos testes é ajudar os desenvolvedores a descobrir defeitos antes desconhecidos. É importante elaborar casos de teste que exercitam as capacidades de manipulação de erros do componente.

**Casos de teste negativos:** Para descobrir novos defeitos, também é importante produzir casos de teste que testem que o componente não faz algo que não deveria fazer.

# Testes de unidade

Testes de unidade são implementados usando-se **frameworks** construídos especificamente para esse fim. [Valente, 2020].

- Python: Pytest ou unittest
- JavaScript: JestJS
- Java: JUnit

*(Os exemplos geralmente serão em python ou javascript, podendo ter algum retirado de livros em java.)*

# Conceitos importante

- **Teste**: método que implementa um teste.
- **Fixture**: estado do sistema que será testado por um ou mais métodos de teste, incluindo dados, objetos, etc.
- **Casos de Teste (Test Case)**: classe com os métodos de teste.
- **Suíte de Testes (Test Suite)**: conjunto de casos de teste, os quais são executados pelo framework de testes de unidade.
- **Sistema sob Teste (System Under Test, SUT)**: sistema que está sendo testado.

# Propriedade FIRST

- **Rápidos (Fast):** desenvolvedores devem executar testes de unidades frequentemente, para obter feedback rápido sobre bugs e regressões no código.
- **Independentes:** a ordem de execução dos testes de unidade não é importante. Para quaisquer testes T1 e T2, a execução de T1 seguida de T2 deve ter o mesmo resultado da execução de T2 e depois T1.
- **Determinísticos (Repeatable):** testes de unidade devem ter sempre o mesmo resultado.
- **Auto-verificáveis (Self-checking):** O resultado de um teste de unidades deve ser facilmente verificável. Adicionalmente, quando um teste falha, deve ser possível identificar essa falha de forma rápida, incluindo a localização do comando **assert** que falhou.
- **Escritos o quanto antes (Timely),** se possível antes mesmo do código que vai ser testado.

# Exemplos de teste

---

# Exemplos

```
1 # content of test_sample.py
2 def funcao(x):
3     return x + 1
4
5 # Testes
6 def test_answer():
7     assert funcao(3) == 4
8
9 def test_answer_two():
10    assert funcao(3) != 6
```

---

```
1 import pytest
2
3 class MyClass:
4     def __init__(self, name, age):
5         self.name = name
6         self.age = age
7
8 @pytest.fixture
9 def my_class():
10     return MyClass(name="pavol", age=39)
11
12 def test_name(my_class):
13     assert my_class.name == "pavol"
14
15 def test_age(my_class):
16     assert my_class.age == 39
```

---



# Cobertura de Testes

Um sistema com alta cobertura de código significa que foi mais exaustivamente testado e tem uma menor chance de conter erros, ao contrário de um sistema com baixa cobertura de código.

A cobertura de código, sendo uma métrica quantitativa, visa medir quanto (%) do software é coberto/exercitado ao executar um determinado conjunto de casos de testes.

# Cobertura de Testes – Métricas

- 
- Branchs: percentual de branches de um programa que são executados por testes; um comando if sempre gera dois branches: quando a condição é verdadeira e quando ela é falsa). Cobertura de comandos e de branches são também chamadas de Cobertura C0 e Cobertura C1, respectivamente.
- Statement: verifica quantas instruções do código são executadas;
- Conditional:

# Cobertura de Testes: Funções

Funções: percentual de funções que são executadas por um teste

```
int coverage (int x, int y)
{
    int z = 0;
    if ((x > 0) && (y > 0))
    {
        z = x;
    }
    return z;
}
```

# Cobertura de Testes: Statement

Verifica quantas instruções do código são executadas;

`coverage(1,0)`

```
int coverage (int x, int y)
{
    int z = 0;
    if ((x > 0) && (y > 0))
    {
        z = x;
    }
    return z;
}
```

Statement code coverage.

# Cobertura de Testes: Branch

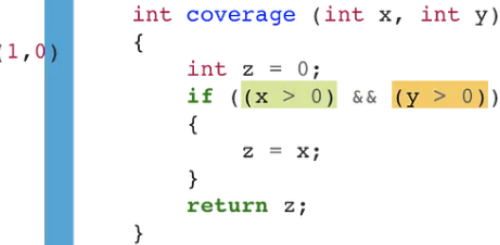
Branchs: percentual de branches de um programa que são executados por testes; um comando if sempre gera dois branches: quando a condição é verdadeira e quando ela é falsa). Cobertura de comandos e de branches são também chamadas de Cobertura C0 e Cobertura C1, respectivamente.

`coverage(1,1)`

```
int coverage (int x, int y)
{
    int z = 0;
    if ((x > 0) && (y > 0))
    {
        z = x;
    }
    return z;
}
```

# Cobertura de Testes

Verifica se cada sub-expressão booleana são avaliadas ambas como verdadeiras e falsas;



The diagram shows a blue laptop frame containing a white screen. On the screen, C code is displayed. The function `coverage` takes two integer arguments, `x` and `y`. It initializes `z` to 0 and then checks a condition `(x > 0) && (y > 0)`. The sub-expression `(x > 0)` is highlighted in light green, and `(y > 0)` is highlighted in light orange. If the condition is true, `z` is set to `x`. The function returns `z`. To the left of the screen, the text `coverage(1,0)` is written, with `1` in red and `0` in blue, indicating the test case values.

```
coverage(1,0)

int coverage (int x, int y)
{
    int z = 0;
    if ((x > 0) && (y > 0))
    {
        z = x;
    }
    return z;
}
```

# Material Auxiliar

<https://www.youtube.com/watch?v=4bubIRBCLVQ>

<https://medium.com/liferay-engineering-brazil/um-pouco-sobre-cobertura-de-c>

# Referências

---



# Referências



Pressman, Roger (2021)

Engenharia de Software: Uma abordagem Profissional

*AMGH Editora Ltda – 9. ed.*



Sommerville, Ian (2011)

Engenharia de Software

*Pearson Prentice Hall – 9. ed.*



Marco Tulio Valente (2020)

Engenharia de Software Moderna: Princípios e Práticas para Desenvolvimento de Software com Produtividade

*Editora: Independente*



# Obrigada

**Prof. Eliane Faveron Maciel**

UNIFACS  
ecossistema ânima

28 de março de 2024