INSTITUTO FEDERAL DE EDUCAÇÃO, CIÊNCIA E TECNOLOGIA DO RIO GRANDE DO SUL
CAMPUS FARROUPILHA

TÉCNICAS DE TESTE DE SOFTWARE

CURSO: Tecnologia em Análise e Desenvolvimento de Sistemas
DISCIPLINA: Qualidade e Teste de Software
PERÍODO LETIVO: 2025 / 2
PROFESSOR: William Moraes da Silva
DISCENTE: Luciano Magri

1) Particionamento de equivalência

Segundo VINCENZI (2011), o particionamento de equivalência é amplamente utilizada em testes de software, que consiste em dividir as entradas possíveis de um sistema em classes de equivalência, onde cada classe representa um conjunto de dados que deve ser tratado de forma semelhante pelo software. Essa abordagem permite reduzir a quantidade de casos de teste necessários, evitando testes redundantes e aumentando a eficiência da validação dos sistemas. Classes são definidas como válidas ou inválidas, e um caso de teste é gerado para representar cada classe, garantindo cobertura adequada com menor esforço de teste.

Essa técnica é amplamente empregada em testes funcionais de software, especialmente quando há a necessidade de reduzir o número de casos de teste, mantendo uma cobertura eficiente e eficiente de requisitos e regras de negócios.

O objetivo principal da partição de equivalência é gerar casos de teste que sejam representativos de grupos de entradas, reduzindo o esforço de testes sem comprometer a qualidade e a confiabilidade do produto. Assim, ela ajuda a detectar possíveis falhas de forma mais rápida e econômica, ao mesmo tempo em que assegura que diferentes comportamentos do sistema sejam validados.

Exemplo

1 - particinamento de equivalencia.py U

home > lmagri > Documents > coding > ADS > 2025.2 - Qualidade_e_teste_de_software > Atividade - Técnicas de Teste de Software > 1 - particinamento de e

```python
def pode_beber(idade: int ) -> bool:

    if idade >= 18:
        return True
    else:
        return False

def teste_particao_equivalencia():
    assert pode_beber(17) == False, "Erro: Pessoa menor de 18 não deveria poder beber."

    assert pode_beber(18) == True, "Erro: Pessoa com 18 anos deveria poder beber."

    assert pode_beber(25) == True, "Erro: Pessoa com mais de 18 anos deveria poder beber."

    print("Todos os testes passaram com sucesso.")

if __name__ == "__main__":
    teste_particao_equivalencia()
```

2) Análise de valores-limite (Boundary Value Analysis)

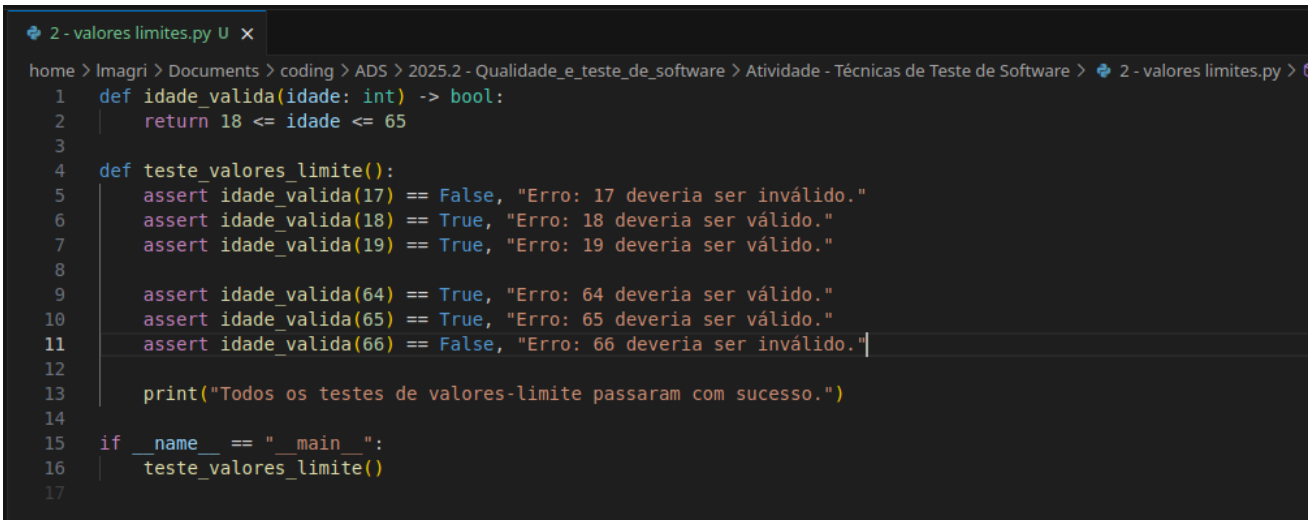
A Análise de Valores-Limite (Boundary Value Analysis - BVA) é uma técnica de teste de software que se concentra em verificar o comportamento do sistema nos limites dos intervalos de entrada permitidos. Ela é usada para identificar falhas ou erros que ocorrem frequentemente nas bordas ou limites desses intervalos, onde o programa pode se comportar de forma errada ou inesperada.

Essa técnica de teste funcional testa os dados de entrada nos limites superiores, inferiores e próximos a esses valores para garantir que o sistema lide corretamente com essas condições extremas. Por exemplo, se um sistema aceita idades de 18 a 65 anos, a técnica testará valores como 17, 18, 65 e 66, pois esses pontos têm maior probabilidade de apresentar erros.

Segundo SILVA et. al (2025), a análise de valores-limites é uma técnica de teste de software que complementa o particionamento de equivalência ao focar nos limites das classes de entradas e saídas, onde normalmente ocorrem mais erros. Essa técnica consiste em selecionar casos de teste nos valores extremos dos intervalos válidos, tanto mínimos quanto máximos, para verificar se o sistema trata corretamente essas fronteiras. Sua aplicação é importante para identificar falhas que não seriam detectadas com testes em pontos internos das classes de equivalência, garantindo maior robustez e confiabilidade ao sistema testado.

O objetivo da análise de valores-limite é identificar erros críticos que ocorrem nas fronteiras dos intervalos de entrada, melhorar a qualidade do software ao testar situações extremas e otimizar o processo de teste ao focar em valores que têm maior probabilidade de falhas. Assim, a técnica ajuda a economizar tempo e recursos, dando prioridade a testes críticos e relevantes.

Exemplo



3) Baseado em casos de uso

De acordo com FARIA (2025), os testes baseados em casos de uso são uma técnica de teste funcional que utiliza os casos de uso — representações dos requisitos funcionais de um sistema — como base para a criação dos casos de teste. Essa abordagem ajuda a garantir que todas as funcionalidades descritas nos casos de uso sejam devidamente testadas, permitindo verificar se o sistema implementa corretamente as operações descritas e os fluxos típicos de interação com o usuário. Além disso, os testes derivados de casos de uso facilitam a identificação das entradas, saídas e possíveis cenários a serem avaliados, aproximando a validação do software dos requisitos reais do usuário.

Testes baseados em casos de uso consistem na criação de casos de teste derivados diretamente dos casos de uso, que representam cenários reais de interação do usuário com o sistema. Cada fluxo principal e alternativo do caso de uso é transformado em um ou mais casos de teste, garantindo validação de cenários esperados e alternativos.

Essa técnica é usada quando há especificações claras em forma de casos de uso, frequentemente durante o desenvolvimento orientado a requisitos e design centrado no usuário. Ela é especialmente útil para garantir que todos os caminhos que um usuário pode seguir sejam testados, incluindo situações de exceção e variações de fluxo.

O objetivo é assegurar que todas as funcionalidades visíveis e acessíveis ao usuário estejam corretas, que os fluxos de trabalho sejam adequados e que o sistema atenda às expectativas do usuário

real. Além disso, auxilia na identificação precoce de erros e permite uma cobertura sistemática, promovendo maior qualidade no software e facilitando a comunicação entre equipes técnicas e clientes.

Os testes baseados em casos de uso transformam as interações do usuário em casos de teste práticos e organizados, focando na usabilidade e nos requisitos reais do sistema para garantir sua qualidade e adequação ao uso esperado.

Exemplo

3 - casos de uso.py U

home > lmagri > Documents > coding > ADS > 2025.2 - Qualidade_e_teste_de_software > Atividade - Técnicas de Teste de Software > 3 - casos de uso.py > ...

```python
def cadastrar_produto(nome: str, preco: float) -> bool:
    if not nome or preco < 0:
        return False
    return True

def teste_caso_de_uso_cadastro_produto():
    assert cadastrar_produto("Camiseta", 49.90) == True, "Erro: Produto válido não foi cadastrado."

    assert cadastrar_produto("", 49.90) == False, "Erro: Produto com nome vazio deveria ser inválido."

    assert cadastrar_produto("Calça", -10) == False, "Erro: Produto com preço negativo deveria ser inválido."

    print("Todos os testes de caso de uso para cadastro de produto passaram com sucesso.")

if __name__ == "__main__":
    teste_caso_de_uso_cadastro_produto()
```

4) Baseado em caminhos

Testes baseados em caminhos são uma técnica de teste estrutural que visa garantir a execução dos diferentes caminhos lógicos existentes no código-fonte de um programa, com o objetivo de verificar o comportamento do sistema em cada uma dessas rotas.

Segundo OLIVEIRA (2012), os testes baseados em caminhos são uma técnica de teste estrutural (caixa branca) que envolve a análise do fluxo de controle de um programa para identificar diferentes caminhos de execução. Cada caminho é uma sequência de instruções que pode ser seguida durante a execução do programa. A técnica seleciona caminhos independentes, que contêm ao menos uma nova aresta não percorrida anteriormente, garantindo cobertura da lógica do programa, especialmente das estruturas de decisão. O teste baseado em caminhos tem como objetivo exercitar todos os fluxos possíveis do sistema, focando na execução das decisões e no controle do fluxo do programa, utilizando a complexidade ciclomática para determinar o número mínimo de casos de teste necessários.

O principal objetivo dos testes baseados em caminhos é assegurar que todas as rotas possíveis (ou as mais importantes) dentro do fluxo do programa sejam testadas, permitindo encontrar falhas em decisões condicionais e fluxos que testes mais gerais podem não detectar. Isso melhora a confiança na robustez da aplicação e ajuda na identificação de defeitos lógicos.

Essa abordagem detalhada de testes foca na estrutura interna do código para cobrir os caminhos de execução e detectar bugs em fluxos específicos que poderiam passar despercebidos em outros tipos de teste.

Exemplo

4 - baseado em caminhos.py U

home > Imagri > Documents > coding > ADS > 2025.2 - Qualidade_e_teste_de_software > Atividade - Técnicas de Teste de Software > 4 - base

```python
def validar_numero(numero: int) -> str:
    if numero > 0:
        return "positivo"
    elif numero < 0:
        return "negativo"
    else:
        return "zero"

def testes_caminhos():
    assert validar_numero(10) == "positivo", "Erro no caminho 1: esperado 'positivo'"

    assert validar_numero(-5) == "negativo", "Erro no caminho 2: esperado 'negativo'"

    assert validar_numero(0) == "zero", "Erro no caminho 3: esperado 'zero'"

    print("Todos os caminhos de teste passaram com sucesso.")

if __name__ == "__main__":
    testes_caminhos()
```

5) Baseado em comandos (Statement Testing)

Testes baseados em comandos (Statement Testing) são uma técnica de teste estrutural focada em garantir que cada comando ou instrução do código seja executado pelo menos uma vez durante o teste.

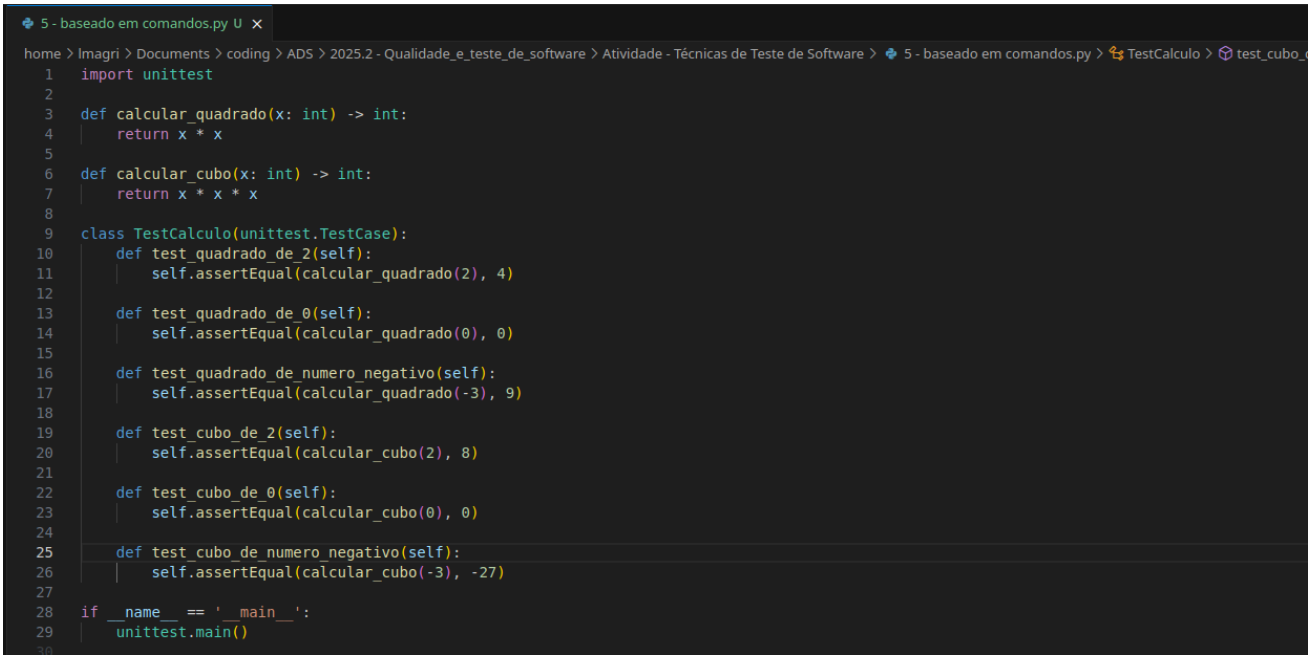
Teste baseado em comandos, também conhecido como Statement Testing, consiste em criar casos de teste que executem todas as linhas ou comandos do código-fonte ao menos uma vez, assegurando que nenhuma instrução permaneça não testada. Essa técnica é bastante utilizada para detectar erros simples de execução, como comandos nunca alcançados (código morto) ou linhas que apresentam exceções.

É usada durante as fases de teste estrutural (caixa branca), especialmente em projetos onde se tem controle do código e deseja-se assegurar cobertura mínima da implementação. É útil para encontrar partes do código que nunca são executadas por algum motivo, garantindo um nível básico de qualidade. Geralmente, é uma etapa inicial para testes mais aprofundados, como testes baseados em caminhos ou condições.

O objetivo principal é assegurar que todos os comandos sejam verificados, mesmo que brevemente, para identificar possíveis erros e melhorar a confiabilidade do código. Assim, melhora-se a qualidade geral do software, minimizando o risco de códigos não testados causarem falhas em produção.

De acordo com VISURE SOLUTIONS (2025), Os testes baseados em comandos são uma técnica de teste de software em que os casos de teste são gerados a partir dos comandos ou operações disponíveis no sistema, simulando entradas reais que o usuário pode executar. Essa abordagem é especialmente útil para sistemas interativos, onde cada comando representa uma ação que pode alterar o estado do sistema. O objetivo é verificar se a execução desses comandos produz os resultados esperados e se o sistema responde corretamente a diferentes sequências de comandos, assegurando o funcionamento das funcionalidades conforme especificado.

Exemplo



6) Baseado em ramos (Branch Testing)

Testes baseados em ramos (Branch Testing) são uma técnica de teste de software que foca em garantir que todos os possíveis ramos de decisão em um programa sejam testados.

Para VILELA (2012), os testes baseados em ramos são uma técnica de teste estrutural que visa verificar todas as ramificações (ou decisões) presentes no código-fonte do software. Cada ramo representa um caminho de execução possível gerado a partir de uma decisão (como uma instrução if ou switch). O objetivo desses testes é garantir que cada possível desvio lógico seja exercitado ao menos uma vez, assegurando que todas as condições e fluxos alternativos sejam testados. A técnica está relacionada à complexidade ciclomática, que ajuda a determinar a quantidade mínima de casos de teste para cobrir todos os ramos do programa, fundamental para identificar erros em decisões condicionais.

É usada principalmente em testes estruturais (caixa branca) quando o foco é verificar a lógica interna do código, especialmente em programas com múltiplos pontos de decisão. É essencial para sistemas críticos que exigem alta confiabilidade e para detectar bugs que podem surgir em ramos específicos que nem sempre são acionados em testes menos detalhados.

O objetivo do Branch Testing é garantir que todas as decisões no código são exercitadas, ajudando a revelar erros que ocorrem em caminhos alternativos e melhorando a qualidade e robustez do software. Ao alcançar cobertura de ramos, o teste oferece uma análise mais profunda do comportamento do programa do que o teste baseado apenas em comandos (Statement Testing).

Testes baseados em ramos asseguram que todas as decisões do código sejam testadas com seus resultados possíveis, aumentando a confiabilidade do software e evitando falhas em cenários específicos.

Exemplo

6 - baseado em ramos.py U

home > lmagri > Documents > coding > ADS > 2025.2 - Qualidade_e_teste_de_software > Atividade - Técnicas de Teste de Software >

```python
import unittest

def avaliar_credito(renda: float, historico_score: int) -> str  :
    if renda > 5000:
        if historico_score > 700:
            return "Crédito aprovado"
        else:
            return "Reprovado por score"
    else:
        if historico_score > 700:
            return "Aprovado com restrição"
        else:
            return "Crédito negado"

class TestAvaliacaoCredito(unittest.TestCase):
    def test_aprovado(self):
        self.assertEqual(avaliar_credito(6000, 750), "Crédito aprovado")

    def test_reprovado_por_score(self):
        self.assertEqual(avaliar_credito(6000, 650), "Reprovado por score")

    def test_aprovado_com_restricao(self):
        self.assertEqual(avaliar_credito(4000, 750), "Aprovado com restrição")

    def test_credito_negado(self):
        self.assertEqual(avaliar_credito(4000, 650), "Crédito negado")

if __name__ == "__main__":
    unittest.main()
```

7) Baseado em condições (Condition Testing)

Testes Baseados em Condições (Condition Testing) são um método de teste estrutural que foca na avaliação das condições booleanas ou lógicas dentro das decisões do código fonte.

De acordo com SALAMON (2019), testes baseados em condições são uma técnica de teste estrutural que foca na avaliação de cada condição lógica dentro das decisões do programa, verificando se todas as possíveis combinações de condições são testadas para assegurar que o software responde corretamente em diferentes cenários. Essa abordagem visa garantir que todas as condições tomadas isoladamente e em conjunto sejam cobertas pelos casos de teste, permitindo detectar erros específicos em decisões compostas e contribuindo para a robustez do sistema.

Essa técnica é usada em testes de caixa branca, principalmente em sistemas com decisões lógicas complexas, onde uma expressão condicional contém múltiplas partes (AND, OR, NOT). O objetivo é garantir que todas as condições, isoladamente, estejam cobertas nos testes, buscando maior precisão na validação lógica do programa.

O principal objetivo do Condition Testing é assegurar que cada componente condicional seja testado em seus dois valores verdadeiros e falsos, para identificar falhas que podem ocorrer quando certas condições não são corretamente avaliada ou isolada. Isso ajuda a encontrar defeitos difíceis relacionados a lógica e combinações de condições.

Testes baseados em condições aumentam a profundidade dos testes ao focar na granularidade dos componentes lógicos das decisões do programa, complementando técnicas como branch testing e path testing para uma cobertura mais detalhada.

Exemplo

7 - baseado em condicoes.py U

home > lmagri > Documents > coding > ADS > 2025.2 - Qualidade_e_teste_de_software > Atividade - Técnicas de Teste de Software > 7 - baseado em condicoes.py > pode_ser_can

```python
import unittest

def pode_ser_candidato(idade: int , cargo: str) -> bool:
    if cargo in ["Presidente", "Vice-Presidente", "Senador"]:
        return idade >= 35
    elif cargo in ["Governador", "Vice-Governador"]:
        return idade >= 30
    elif cargo in ["Deputado Federal", "Deputado Estadual", "Deputado Distrital", "Prefeito", "Vice-Prefeito", "Juiz de Paz"]:
        return idade >= 21
    elif cargo == "Vereador":
        return idade >= 18
    return False

class TesteIdadeMinimaCandidatura(unittest.TestCase):
    def test_presidente_apos_35(self):
        self.assertTrue(pode_ser_candidato(35, "Presidente"))
        self.assertFalse(pode_ser_candidato(34, "Presidente"))

    def test_governador_apos_30(self):
        self.assertTrue(pode_ser_candidato(30, "Governador"))
        self.assertFalse(pode_ser_candidato(29, "Governador"))

    def test_deputado_apos_21(self):
        self.assertTrue(pode_ser_candidato(21, "Deputado Federal"))
        self.assertFalse(pode_ser_candidato(20, "Deputado Federal"))

    def test_vereador_apos_18(self):
        self.assertTrue(pode_ser_candidato(18, "Vereador"))
        self.assertFalse(pode_ser_candidato(17, "Vereador"))

    def test_cargo_desconhecido(self):
        self.assertFalse(pode_ser_candidato(50, "Presidente de Clube"))

if __name__ == "__main__":
    unittest.main()
```

8) Baseado em condições múltiplas (Multiple Condition Testing)

Testes Baseados em Condições Múltiplas (Multiple Condition Testing) são uma técnica de teste estrutural que verifica todas as possíveis combinações de valores verdadeiros e falsos para as condições envolvidas numa decisão composta.

Multiple Condition Testing (ou Multiple Condition Coverage - MCC) assegura que cada combinação possível das condições componentes de uma expressão condicional seja testada. Por exemplo, se uma decisão depende de três condições booleanas, MCC testa todas as 8 combinações possíveis (2^3), garantindo que nenhuma interação entre condições passe despercebida.

É usada em sistemas com decisões lógicas complexas e críticas, onde bugs relacionados a combinações específicas de condições podem causar falhas graves. Normalmente é aplicada quando é necessário um alto nível de garantia na cobertura lógica, especialmente em software de segurança ou aplicações críticas.

Visa garantir máxima cobertura da lógica condicional, testando todos os efeitos possíveis das combinações das condições, não apenas isoladamente mas em sua interação. Isso evita situações onde uma condição individualmente correta gera comportamentos inesperados quando combinada com outras.

Para FARIA (2008), testes baseados em condições múltiplas são uma técnica avançada de teste estrutural que visa verificar todas as possíveis combinações das condições presentes em decisões compostas no software. Diferentemente dos testes baseados em condições simples, esta abordagem avalia as interações entre diversas condições em uma única decisão lógica, garantindo uma cobertura completa das variações possíveis e detectando falhas que podem ocorrer em conjunto. Esse método é fundamental para validar a correta implementação de lógicas complexas, como expressões booleanas com múltiplos operandos, elevando a confiabilidade do sistema.

Exemplo

8 - baseado em condicoes multiplas.py U

home > lmagri > Documents > coding > ADS > 2025.2 - Qualidade_e_teste_de_software > Atividade - Técnicas de Teste de Software > 8 -

```python
import unittest

def pode_passar(idade: int, possui_permissao: bool) -> bool:
    if idade >= 18 and possui_permissao:
        return True
    else:
        return False

class TestMultipleCondition(unittest.TestCase):
    def test_todas_combinacoes(self):
        test_cases = [
            (17, False, False),
            (17, True, False),
            (18, False, False),
            (18, True, True),
        ]
        for idade, permissao, esperado in test_cases:
            with self.subTest(idade=idade, permissao=permissao):
                self.assertEqual(pode_passar(idade, permissao), esperado)

if __name__ == "__main__":
    unittest.main()
```

REFERENCIAS BIBLIOGRÁFICAS

VINCENZI, Adriana Rocha. **Teste Funcional Sistemático Estendido:** Uma contribuição na aplicação de critérios de teste caixa-preta. 2011. Dissertação (Mestrado em Ciências da Computação) – Instituto de Informática, Universidade Federal de Goiás, Goiânia, 2011. Disponível em: https://repositorio.bc.ufg.br/tede/items/c9841cdb-860f-48f4-8fec-7b1e4084e2e4. Acesso em: 30 out. 2025.

SILVA, Cláudia Lins de; et al. **Investigando o Uso de Práticas Modernas de Testes de Software:** Partição de Equivalência e Análise de Valor Limite. 2025. Trabalho de Conclusão de Curso – Instituto de Computação, Universidade Federal do Ceará, Fortaleza, 2025. Disponível em: https://repositorio.ufc.br/bitstream/riufc/80606/1/2025_tcc_clsilva.pdf. Acesso em: 30 out. 2025.

FARIA, Jonatas M. **Partição de equivalência e Testes Baseados em Casos de Uso.** 2024. Disponível em: https://www.cin.ufpe.br/~processos/TAES3/Workshops_Qualidade/Workshop_Qualidade-2008-2/Testes_Baseados_em_Modelo_27112008.pdf. Acesso em: 30 out. 2025.

OLIVEIRA, Rafael. **Uma abordagem para o ensino de teste estrutural baseada em grafos de fluxo de controle.** Sociedade Brasileira de Computação, 2012. Disponível em: https://sol.sbc.org.br/index.php/erbase/article/download/8581/8482/. Acesso em: 1 nov. 2025.

VISURE SOLUTIONS. **Teste baseado em requisitos: guia de abordagem e melhores práticas.** 2025. Disponível em: https://visuresolutions.com/pt/guia-de-esmolas/teste-baseado-em-requisitos/. Acesso em: 1 nov. 2025.

VILELA, C. **Testes Funcionais de Software.** 2012. Disponível em: https://www.devmedia.com.br/testes-funcionais-de-software/23565. Acesso em: 2 nov. 2025.

SALAMON, João S. **Testes baseados em requisitos.** Universidade Federal do Espírito Santo, 2019. Disponível em: https://www.inf.ufes.br/~jssalamon/wp-content/uploads/disciplinas/engsoft/slides/Slide%208.1%20-%20Testes%20baseados%20em%20Requisitos.pdf. Acesso em: 2 nov. 2025.

FARIA, Jonatas M. **Testes Baseados em Modelo.** Universidade Federal de Pernambuco, 2008. Disponível em: https://www.cin.ufpe.br/~processos/TAES3/Workshops_Qualidade/Workshop_Qualidade-2008-2/Testes_Baseados_em_Modelo_27112008.pdf. Acesso em: 2 nov. 2025.