

Gestão de Software

UC: Gestão e Qualidade de Software

Prof. Eliane Faveron Maciel

UNIFACS
Ecosistema ânima

11 de abril de 2024

Overview

1. Testes de Software

- 1.1 Técnicas de testes
- 1.2 Níveis de Testes

2. Testes de Unidade

3. Escrevendo os testes

- 3.1 Before e After
- 3.2 Mocks e Stubs

4. Referências

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.



Caixa branca vs Caixa preta

O teste **caixa-branca**, também chamado de teste da caixa-de-vidro ou teste estrutural, é uma loso a de projeto de casos de teste que usa a estrutura de controle descrita como parte do projeto no nível de componentes para derivar casos de teste.

O teste **caixa-preta** concentra-se nos testes com uma perspectiva de usuário. Não precisando entender todo o funcionamento do sistema.

Qual o objetivo do teste?

- Validar se um programa está em conformidade com os requisitos.
- Medir se a entrega possui qualidade.
- Localizar erros significativos dentro do prazo definido e verifica se as correções aplicadas estão em conformidade com os requisitos.
- Poder melhorar a qualidade e satisfação do cliente.

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.

Características do teste unitário

- É necessário ter uma plataforma e um ambiente de teste ou depuração para executar o código.
- Ferramentas de cobertura de código podem ajudar a verificar se todos os caminhos foram percorridos.
- Os caminhos de erro mais complexos costumam ser negligenciados, o que pode ter consequências dispendiosas mais tarde no ciclo de desenvolvimento.

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.

Escrevendo os testes

Testes em Python

Para criar os testes em java utilizaremos o JUnit que permite implementar classes que vão testar. As classes de teste têm o mesmo nome das classes testadas, mas com um sufixo Test.

Os métodos começam com o prefixo **test** e devem obedecer:

1. serem públicos, pois eles serão chamados pelo JUnit;
 2. não possuírem parâmetros;
 3. possuírem a anotação `@Test`, a qual identifica métodos que deverão ser executados durante um teste.
- Caso você não tenha o JUnit instalado, faça o download do arquivo junit.jar em www.junit.org, após inclua-o no classpath para compilar e rodar os programas de teste.
 - Porém o JUnit já vem configurado nas versões recentes de IDE's como Eclipse, NetBeans, JBuilder, BlueJ e outros.

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
```

Testes em Java

Para criar os testes em java utilizaremos o JUnit que permite implementar classes que vão testar. As classes de teste têm o mesmo nome das classes testadas, mas com um sufixo Test.

Os métodos começam com o prefixo **test** e devem obedecer:

1. serem públicos, pois eles serão chamados pelo JUnit;
 2. não possuírem parâmetros;
 3. possuírem a anotação `@Test`, a qual identifica métodos que deverão ser executados durante um teste.
- Caso você não tenha o JUnit instalado, faça o download do arquivo junit.jar em www.junit.org, após inclua-o no classpath para compilar e rodar os programas de teste.
 - Porém o JUnit já vem configurado nas versões recentes de IDE's como Eclipse, NetBeans, JBuilder, BlueJ e outros.

Exemplos

```
import org.junit.Test;
import static org.junit.Assert.assertTrue;

public class StackTest {

    @Test
    public void testEmptyStack() {
        Stack<Integer> stack = new Stack<Integer>();
        boolean empty = stack.isEmpty();
        assertTrue(empty);
    }
}
```

Before

As classes de testes podem conter uma função **Before**, esta por sua vez é chamada antes de qualquer outro método de teste.

```
1 @Before
2 public void init() {
3     stack = new Stack<Integer>();
4 }
```

```
1 import pytest
2
3 @pytest.fixture
4 def define_target():
5     return {'url': 'smtp.gmail.com', 'port': 587}
6
7 @pytest.fixture
8 def smtp_connection(define_target):
9     import smtplib
10    url = define_target['url']
11    port = define_target['port']
12    return smtplib.SMTP(url, port, timeout=5)
13
14 def test_ehlo(smtp_connection):
15     response, msg = smtp_connection.ehlo()
16     assert response == 250
17
18
19 #Exemplo baseado da documentação oficial https://docs.pytest.org/en/stable/fixture.html
```

Mocks e Stubs

Os **stubs** fornecem respostas prontas para chamadas feitas durante o teste. Os **mocks** são pré-programados com expectativas que formam uma especificação das chamadas que devem receber. Eles podem lançar uma exceção se receberem uma chamada inesperada e forem verificados durante a verificação para garantir que receberam todas as chamadas que esperavam.

Stubs e **mocks** diferem em seu propósito e escopo. O objetivo de um stub é fornecer dados consistentes e previsíveis para seu código consumir, enquanto o objetivo de um mock é afirmar expectativas e verificar comportamentos para seu código produzir. O escopo de um stub é limitado à saída da dependência, enquanto o escopo de um mock abrange tanto a saída quanto a entrada da dependência.

Mocks em Pytest

Mocks são substitutos para os métodos internos.

```
1 pip install pytest-mock
```

```
1 def test_foo(mocker):  
2     # all valid calls  
3     mocker.patch('os.remove')  
4     mocker.patch.object(os, 'listdir', autospec=True)  
5     mocked_isfile = mocker.patch('os.path.isfile')
```

```
1 import unittest
2 from unittest.mock import MagicMock
3
4 class Calculator:
5     def add(self, x, y):
6         return x + y
7
8 class TestCalculator(unittest.TestCase):
9     def test_add(self):
10         # Criar um mock para a classe Calculator
11         calculator_mock = MagicMock(spec=Calculator)
12         # Definir o comportamento esperado do mock
13         calculator_mock.add.return_value = 5
14         # Testar o método add
15         result = calculator_mock.add(2, 3)
16         # Verificar se o resultado está correto
17         self.assertEqual(result, 5)
18 if __name__ == '__main__':
19     unittest.main()
```

Spy em Pytest

A função de um spy em testes é permitir que você verifique não apenas se um método foi chamado, mas também quantas vezes foi chamado, com quais argumentos e em que ordem. Isso é útil quando você está interessado não apenas no resultado de uma chamada de método.

```
1 def test_spy_method(mocker):
2     class Foo(object):
3         def bar(self, v):
4             return v * 2
5
6     foo = Foo()
7     spy = mocker.spy(foo, 'bar')
8     assert foo.bar(21) == 42
9
10    spy.assert_called_once_with(21)
11    assert spy.spy_return == 42
```

Stub em Pytest

```
1 def test_stub(mocker):  
2     def foo(on_something):  
3         on_something('foo', 'bar')  
4  
5     stub = mocker.stub(name='on_something_stub')  
6  
7     foo(stub)  
8     stub.assert_called_once_with('foo', 'bar')
```

```
1 import org.junit.Test;
2 import static org.junit.Assert.assertEquals;
3 import static org.mockito.Mockito.*;
4
5 public class CalculatorTest {
6
7     @Test
8     public void testAdd() {
9         // Criar um mock para a classe Calculator
10         Calculator calculatorMock = mock(Calculator.class);
11
12         // Definir o comportamento esperado do mock
13         when(calculatorMock.add(2, 3)).thenReturn(5);
14
15         // Testar o método add
16         int result = calculatorMock.add(2, 3);
17
18         // Verificar se o resultado está correto
19         assertEquals(5, result);
20     }
```

Prática

- Crie um repositório no github para o projeto A3
- Crie um template para descrição dos Pull Requests - <https://docs.github.com/pt/communities/using-templates-to-encourage-useful-issues-and-pull-requests/creating-a-pull-request-template-for-your-repositoryadding-a-pull-request-template>
- <https://docs.github.com/pt/actions/using-workflows/creating-starter-workflows-for-your-organizationcreating-a-starter-workflow> - Crie um workflow que rode os testes das PRs - GitHub Workflows
- Integrar com ferramenta de Code Review IA
- Instalar no projeto ferramenta de pre-commit:
- <https://precommit.com>
- Husky

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
Ecosistema ânima

11 de abril de 2024