UNIVERSIDADE FEDERAL DE PERNAMBUCO

Centro de Informática
UFPE

Universidade Federal de Pernambuco
Centro de Informática - CIn
Disciplina: Fundamentos de Processo de Desenvolvimento de Software
Docente: Breno Miranda
Discente: Livya Karolinne Fonseca de Menezes

Avaliação - Fundamentos de Processo de Desenvolvimento de Software

Tutorial - TDD com Pytest

INTRODUÇÃO

O *Test Driven Development* (TDD) baseia-se em escrever testes antes da implementação, ocorrendo assim um desenvolvimento guiado por testes.

No TDD, antes da codificação, devem ser escritos testes que validem os requisitos, porém, como ainda não tem código implementado, a falha deste teste é esperada. Em seguida, é feita a codificação do requisito para que os testes sejam executados com sucesso. Após isso, podemos fazer refatoramento no código, como por exemplo, eliminar *code smells*. Esse ciclo se repete até que todos os requisitos estejam implementados.

Existem várias bibliotecas que possibilitam a aplicação do TDD através de testes automatizados dependendo da linguagem de programação utilizada. Neste tutorial será utilizada a biblioteca Pytest para realizar testes de uma classe que representa uma conta bancária.

PRÉ-REQUISITOS

Para a realização deste tutorial, será preciso

- python 3.7.4
- pip 20.3.1
- virtualenv 20.2.2

PREPARANDO O AMBIENTE

Para preparar o ambiente, será preciso executar os seguintes passos:

1. Criar o ambiente virtual:

```
$ virtualenv -p /usr/bin/python3 .tddenv
```

2. Ativar o ambiente virtual criado no passo anterior:

```
$ source .tddenv/bin/activate
```

3. Instalar o Pytest:

```
$ pip install pytest
```

4. Criar e acessar o diretório do projeto:

```
$ mkdir TDDPytest && cd TDDPytest
```

5. Criar o arquivo que conterá a implementação dos requisitos:

```
$ touch conta.py
```

6. Criar o arquivo que conterá os testes:

```
$ touch conta_test.py
```

7. Colocar o conteúdo inicial do conta.py, o qual conterá uma classe Conta e seu construtor, que recebe um saldo inicial. Essa implementação não vai alterar em nada na falha do teste:

```
class Conta:

   def __init__(self, saldo):
       self.saldo = saldo
```

8. Colocar o conteúdo inicial do conta_test.py:

```
import pytest
```

IMPLEMENTAÇÃO - TDD

Para a implementação do TDD, os componentes devem ser testados separadamente, sem os outros componentes de sistema. Então, como a ideia do TDD é testar com base em requisitos, neste tutorial serão utilizados três requisitos de um sistema de operações bancárias para serem exemplificados, são eles: realizar consulta de saldo, depósito bancário e saque. E para cada requisito, a realização dos testes em TDD será feita passo a passo.

[RF01] - Realizar consulta do saldo bancário: O sistema deve permitir a consulta do saldo bancário.

1. Escrever o teste de consulta de saldo:

```
from conta import Conta

def test_consulta_saldo():
    conta = Conta(100)
    assert conta.consultar_saldo() == 100
```

2. Executar o Pytest para obter a falha do teste, que será falha por não ter o método consultar_saldo definido:

```
$ pytest
```

3. Implementar o método/requisito (que é a quantidade de código mínima para fazer com que o teste passe) na classe Conta no arquivo conta.py que realiza a consulta do saldo:

```
def consultar_saldo(self):
    return self.saldo
```

4. Reexecutar o Pytest, que é esperado que seja executado com sucesso:

```
$ pytest
```

[RF02] - Realizar depósito bancário: O sistema deve permitir a realização de um depósito bancário.

1. Incrementar o arquivo conta_test.py adicionando o teste referente ao depósito bancário:

```
def test_deposito():
    conta = Conta(100)
    conta.deposito(10)
    assert conta.consultar_saldo() == 110
```

2. Executar o Pytest para rodar a suíte de testes e obter a falha do teste:

```
$ pytest
```

3. Implementar o método/requisito na classe Conta no arquivo conta.py que realiza o depósito bancário:

```
def deposito(self, valor):
    self.saldo = self.saldo + valor
```

4. Reexecutar o Pytest, que é esperado que seja executado com sucesso:

```
$ pytest
```

[RF03] - Realizar saque bancário: O sistema deve permitir o saque bancário, exceto caso o valor a ser sacado seja maior do que o saldo da conta.

1. Incrementar o arquivo conta_test.py adicionando o teste referente ao saque bancário:

```
def test_saque():
    conta = Conta(100)
    conta.saque(10)
    assert conta.consultar_saldo() == 90
```

2. Executar o Pytest para obter a falha do teste:

```
$ pytest
```

3. Implementar o método/requisito na classe conta.py que realiza o saque bancário:

```
def saque(self, valor):
    self.saldo = self.saldo - valor
```

4. Reexecutar o Pytest, que é esperado que seja executado com sucesso:

```
$ pytest
```

5. Para o mesmo requisito, implementar um teste que capture uma exceção caso o valor do saque seja maior que o valor do saldo:

```
def test_saque_exception():
    conta = Conta(100)
    with pytest.raises(SaldoInsuficienteException):
           conta.saque(500)
```

6. Executar o Pytest para obter a falha do teste:

```
$ pytest
```

7. Criar o arquivo saldo_insuficiente_exception.py que conterá a exceção que será lançada caso seja realizado um saque maior que o saldo disponível:

```
$ touch saldo_insuficiente_exception.py
```

8. Adicionar o conteúdo da exceção no arquivo saldo_insuficiente_exception.py:

```
class SaldoInsuficienteException(Exception):
    def __init__(self, message):
        self.message = message
        super().__init__(self.message)
```

9. Adicionar no início do arquivo conta.py:

```
from saldo_insuficiente_exception import SaldoInsuficienteException
```

10. Adicionar no início do arquivo conta_test.py:

```
from saldo_insuficiente_exception import SaldoInsuficienteException
```

11. Modificar o método na classe Conta do arquivo conta.py que realiza o saque bancário:

```
def saque(self, valor):
   if self.saldo >= valor:
       self.saldo = self.saldo - valor
   else:
       raise SaldoInsuficienteException("Você não possui saldo suficiente")
```

7. Reexecutar o Pytest, que é esperado que seja executado com sucesso:

```
$ pytest
```

Ao final da execução dos passos listados em cada requisito, foi retornado sucesso para todos os

quatro testes realizados. Dessa forma, não será necessário escrever mais nenhum código ou refatoração e os requisitos foram satisfeitos.

CONCLUSÃO

Com este tutorial, foi possível ver, na prática, as vantagens do TDD e suas utilidades, como focar nos problemas específicos do desenvolvimento e suas soluções, códigos podem ser escritos de forma mais legível e simples, possibilidade menor de retrabalho, mais facilidade em correção de bugs, etc. Tudo isso possibilita o desenvolvimento de um software com mais qualidade.