

Dublês de testes

Jailma Januário da Silva

Leonardo Massayuki Takuno

Resumo

Os objetivos desta parte são: (I) Entender o que são dublês de testes e para que servem. (II) Entender e utilizar os diversos tipos de dublês de testes. (III) Realizar exemplos de testes usando dublês de testes. (IV) Organizar pastas em projetos com testes. (V) Utilizar a biblioteca mock do Python.

Introdução

Este texto apresenta o conceito de dublês de testes, que segundo FOWLER (2016), é um termo genérico para qualquer caso em que você substitua um objeto de produção para propósitos de testes. Aqui são tratados várias situações em que os dublês de testes são úteis e podem ser construídos. Ao final, o texto apresenta sobre a biblioteca unittest.mock que é bastante útil para construção de testes de unidade.

Dublês de testes

As partes anteriores trabalham testes de unidades de funções com responsabilidades bem definidas e bastante isoladas. Por **isolamento**, entende-se que essas unidades testadas não fazem uso de objetos que acessam fontes de recursos externos, os quais podem ser, por exemplo, arquivos, banco de dados ou serviços externos tais como APIs (do inglês, *Application Programming Interface*), que significa interface de programação de aplicação, que é um conjunto de funcionalidades disponíveis por um *software* para acesso por outras aplicações.

É natural que, ao construir os testes de unidade de um sistema de informação, o software utilize recursos externos, os quais podem não estarem disponíveis, ou então, que podem tornar os testes lentos. Contudo, o objetivo principal do teste de unidade, quase sempre, consiste em testar a funcionalidade a partir dos dados obtidos dessas fontes externas, não importando o local de origem em que esses dados estão armazenados.

Neste caso, torna-se interessante a utilização dos chamados dublês de testes, que são objetos que substituem as funcionalidades de determinadas classes, como por exemplo, as classes que usam acesso às fontes externas, entre outras. E com isso, o teste de unidade mantém a propriedade de isolamento, que é uma das características importantes para manter a qualidade dos testes.

Alguns autores, como MESZAROS (2007) e FOWLER (2006), classificam vários tipos de dublês de testes:

- **Dummy Objects** são objetos que não são utilizados nos testes mas são necessários para serem passados como parâmetros de funções. São dublês que não possuem funcionalidades, e apenas servem para preencher uma lista de parâmetros e viabilizar a execução de um teste;
- **Fake Objects** são objetos que possuem uma implementação simplificada de uma classe com dependência externa. Um exemplo bastante comum é utilizar um repositório de dados em memória para fins de teste, ao invés de acessar o banco de dados de fato.
- **Stubs** fornecem respostas prontas para chamadas realizadas nos testes de unidades, normalmente não respondem a nada diferente do que foi programado para responder nos testes. São dublês de

testes criados para retornar exatamente aquilo que é necessário para o teste passar.

- **Spies** são stubs que também registram algum tipo de informação baseado em como eles são chamados.

Mocks Objects são objetos que simulam o comportamento de objetos reais de maneira controlada. São dublês de testes pré-programados com informações que formam uma especificação das chamadas que esperam receber.

Criando objetos *fake*

No primeiro exemplo, considere o diagrama de classes conforme a Figura 6.1. Observe que existe uma classe **Livro** com atributos, `id`, `titulo`, `preco` e `data_publicacao`. A classe **LivroServico** contém serviços que encapsulam regras de negócios, que para este exemplo, consiste em apenas invocar operações de inserção e obtenção do tamanho do repositório de dados, representado pela classe abstrata **LivroRepositorio**. O motivo dessa classe ser abstrata, indica um contrato a ser seguido, e assim, a classe **LivroServico** depende uma abstração e não diretamente de uma classe concreta. Para os testes, o diagrama sugere um banco de dados que armazena dados em memória, um *fake object*, que é um pattern de teste útil para manter o teste isolado.

Figura 6.1. Diagrama de classes com *fake object*



Fonte: do autor, 2022.

Para o código da classe **Livro**, observe o arquivo `livro.py` conforme apresenta a Codificação 6.1.

Codificação 6.1. livro.py

```
class Livro:
    def __init__(self, id, titulo, preco, data_publicacao):
        self.id = id
        self.titulo = titulo
        self.preco = preco
        self.data_publicacao = data_publicacao
```

Fonte: do autor, 2022.

Como feito na parte 05, organize os arquivos em pastas src, conforme a Figura 6.2. Outros arquivos e pastas serão incluídas a seguir.

Figura 6.2. Organização de arquivos do projeto



Fonte: do autor, 2022.

Para o código da classe **LivroRepositorio**, observe o arquivo livroRepositorio.py de acordo com a Codificação 6.2. Grave esse arquivo livroRepositorio.py na pasta src, assim como o arquivo livro.py.

No Python, para que uma classe seja abstrata, ela deve herdar de uma classe da biblioteca abc (da sigla, *abstract base class*), e os métodos para serem abstratos devem ser decorados com a palavra reservada **@abstractmethod**, que também encontra-se na biblioteca abc.

Codificação 6.2. livroRepositorio.py

```
from abc import ABCMeta, abstractmethod
from src.livro import Livro
```

```
class LivroRepositorio(metaclass=ABCMeta):
```

```
    @abstractmethod
```

```
    def inserir(self, livro: Livro):
```

```
pass
```

```
@abstractmethod
```

```
def buscar_todos(self) -> list:
```

```
pass
```

```
@abstractmethod
```

```
def obter_tamanho(self) -> int:
```

```
pass
```

Fonte: do autor, 2022.

Para a classe **LivroServico**, observe o arquivo livroServico.py conforme apresenta a Codificação 6.3.

Codificação 6.3. livroServico.py

```
from src.livro import Livro
```

```
from src.livroRepositorio import LivroRepositorio
```

```
class LivroServico:
```

```
    def __init__(self, livroRepositorio: LivroRepositorio):
```

```
        self.livroRepositorio = livroRepositorio
```

```
    def insere_livro(self, livro: Livro):
```

```
        self.livroRepositorio.inserir(livro)
```

```
    def obter_tamanho(self):
```

```
        return self.livroRepositorio.obter_tamanho()
```

Fonte: do autor, 2022.

Perceba, pela Codificação 6.3, que ocorre o princípio da inversão de dependência em relação ao objeto da classe **LivroRepositorio**. O construtor da classe **LivroServico** recebe como parâmetro uma abstração, e assim, esse parâmetro pode ser uma instância da classe de produção para armazenar

dados de um banco de dados qualquer. Porém, para facilitar os testes, basta que essa instância seja um *fake object*.

Os três arquivos, `livro.py`, `livroRepositorio.py` e `livroServico.py` devem ser gravados na pasta `src`, conforme a Figura 6.3.

Figura 6.3. `livro.py`, `livroRepositorio.py` e `livroServico.py`



Fonte: do autor, 2022.

Agora, crie o arquivo `livroRepositorioFake.py` e grave dentro de uma pasta `tests`. Observe que, de acordo com a Codificação 6.4, a classe **`LivroRepositorioFake`** utiliza-se de uma lista em memória, e as operações do repositório consistem em manipulações simples da lista.

Codificação 6.4. `livroRepositorioFake.py`

```
from src.livro import Livro
from src.livroRepositorio import LivroRepositorio

class LivroRepositorioFake(LivroRepositorio):
    def __init__(self):
        self.livros = []

    def inserir(self, livro: Livro):
        self.livros.append(livro)

    def buscar_todos(self) -> list:
        return list(self.livros)

    def obter_tamanho(self) -> int:
        return len(self.livros)
```

Fonte: do autor, 2022.

Agora, crie o arquivo de teste, com o nome `test_fake_repositorio.py`, conforme a Codificação 6.5.

Codificação 6.5. `test_fake_repositorio.py`

```
from datetime import date

from livroRepositorioFake import LivroRepositorioFake
from src.livro import Livro
from src.livroServico import LivroServico


def test_inserir_dois_livros_deve_retornar_contagem_2():

    livroRepositorio = LivroRepositorioFake()
    livroServico = LivroServico(livroRepositorio)

    l1 = Livro('111', 'Python para iniciantes', 50.0, date.today)
    l2 = Livro('222', 'Testes automatizados', 70.0, date.today)

    livroServico.insere_livro(l1)
    livroServico.insere_livro(l2)

    assert livroServico.obter_tamanho() == 2
```

Fonte: do autor, 2022.

Para que os testes sejam executados adequadamente, é importante incluir, na raiz do projeto, o arquivo `pytest.ini`, com o conteúdo de acordo com a Figura 6.4.

Figura 6.4. arquivo `pytest.ini`



Fonte: do autor, 2022.

O arquivo `pytest.ini` contém em seu conteúdo a configuração para que o `pytest` possa encontrar os arquivos dentro do projeto.

Para se ter uma visão da organização dos arquivos no projeto, observe a Figura 6.5.

Figura 6.5. Organização de arquivos



Fonte: do autor, 2022.

Execute o pytest para verificar o teste de unidade utilizando-se de um *fake object*, conforme a Figura 6.6.

Figura 6.6. Execução dos testes de unidade



Fonte: do autor, 2022.

Criando objetos *dummy*

Objetos *dummy* são aqueles utilizados nos testes de unidades e que não possuem lógica de negócio. São necessários apenas para fazer o código do teste compilar, mas eles não são o foco principal do teste. Como exemplo de uso de objetos *dummy*, utilize o projeto da seção anterior e, agora, suponha que a classe **LivroRepositorio** necessite, em uma de suas regras de negócios, utilizar-se de um serviço de email. Crie uma classe abstrata para tratar serviços de email, como apresenta a Codificação 6.6.

Codificação 6.6. emailServico.py

```
from abc import ABCMeta, abstractmethod

class EmailServico(metaclass=ABCMeta):
    @abstractmethod
    def enviar_email(self, mensagem: str):
        pass
```

Fonte: do autor, 2022.

A classe **EmailServico** será acoplada à classe **LivroServico**, conforme apresenta a Figura 6.7.

Figura 6.7. Diagrama de classes com o objeto *dummy*



Fonte: do autor, 2022

Agora escreva o código da classe **EmailServicoDummy** de acordo com a listagem da Codificação 6.7.

Codificação 6.7. emailServicoDummy.py

```
from src.emailServico import EmailServico

class EmailServicoDummy(EmailServico):
    def enviar_email(self, mensagem: str):
        raise NotImplemented('Método não implementado!')
```

Fonte: do autor, 2022.

Observe que a classe de serviço **EmailServicoDummy** tem um método concreto denominado **enviar_email()**, porém este método gera uma exceção do tipo **NotImplemented()**, indicando que ele não foi implementado, pois, de fato, ela não deve ser invocada.

Para os testes de unidade sobre a classe **LivroServico**, observe o arquivo livroServico.py de acordo com a Codificação 6.8.

Codificação 6.8. livroServico.py

```
from src.livro import Livro
from src.livroRepositorio import LivroRepositorio
from src.emailServico import EmailServico

class LivroServico:
    def __init__(self, livroRepositorio: LivroRepositorio, \
        emailServico: EmailServico):
```

```

self.livroRepositorio = livroRepositorio
self.emailServico = emailServico

def insere_livro(self, livro: Livro):
    self.livroRepositorio.inserir(livro)

def obter_tamanho(self):
    return self.livroRepositorio.obter_tamanho()

```

Fonte: do autor, 2022.

Para uma visão geral da organização dos arquivos, observe a Figura 6.8.

Figura 6.8. Organização dos arquivos do projeto



Fonte: do autor, 2022

Falta apenas o arquivo `test_dummy_repositorio.py`, que pode ser observado na Codificação 6.9.

Codificação 6.9. `test_dummy_repositorio.py`

```

from datetime import date

from livroRepositorioFake import LivroRepositorioFake
from emailServicoDummy import EmailServicoDummy
from src.livro import Livro
from src.livroServico import LivroServico

def test_inserir_dois_livros_deve_retornar_contagem_2():

    livroRepositorio = LivroRepositorioFake()
    emailServico = EmailServicoDummy()
    livroServico = LivroServico(livroRepositorio, emailServico)

    l1 = Livro('111', 'Python para iniciantes', 50.0, date.today)
    l2 = Livro('222', 'Testes automatizados', 70.0, date.today)

```

```
livroServico.inserir_livro(l1)
```

```
livroServico.inserir_livro(l2)
```

```
assert livroServico.obter_tamanho() == 2
```

Fonte: do autor, 2022.

O objeto **emailServico** da classe **EmailServicoDummy** faz-se necessário, pois a classe **LivroServico** depende do **emailServico** no momento da instanciação. No entanto, o objeto dummy **emailServico** não faz parte do teste, ele está apenas para fazer o código de teste ser compilado.

Criando objetos *stubs*

Objetos *stubs* fornecem respostas pré-definidas para os métodos executados durante os testes de unidades. O comportamento desses objetos são definidos programaticamente de maneira a atender um determinado teste específico, ou seja, ao invés de chamar um serviço externo, o método do objeto *stub* é chamado e este devolve o resultado esperado.

Como exemplo de objetos stubs, observe a Codificação 6.10, sobre a classe abstrata **LivroRepositorio**, a qual acrescenta o método **obter_por_id**.

Codificação 6.10. livroRepositorio.py

```
from abc import ABCMeta, abstractmethod
```

```
from src.livro import Livro
```

```
class LivroRepositorio(metaclass=ABCMeta):
```

```
    @abstractmethod
```

```
    def inserir(self, livro: Livro):
```

```
        pass
```

```
@abstractmethod
```

```
def buscar_todos(self) -> list:  
    pass
```

```
@abstractmethod
```

```
def obter_tamanho(self) -> int:  
    pass
```

```
@abstractmethod
```

```
def obter_por_id(self, id: str) -> Livro:  
    pass
```

Fonte: do autor, 2022.

Agora, crie na pasta **tests** o arquivo `livroRepositorioStub.py` e escreva de acordo com a Codificação 6.11.

Codificação 6.11. `livroRepositorioStub.py`

```
from datetime import date
```

```
from src.livro import Livro
```

```
from src.livroRepositorio import LivroRepositorio
```

```
class ExcecaoLivroNaoEncontrado(Exception):  
    pass
```

```
class LivroRepositorioStub(LivroRepositorio):
```

```
    def __init__(self):  
        self.livros = []  
        l1 = Livro('111', 'Python para iniciantes', 50.0, date.today)  
        l2 = Livro('222', 'Testes automatizados', 70.0, date.today)  
        self.inserir(l1)  
        self.inserir(l2)
```

```
    def inserir(self, livro: Livro):  
        self.livros.append(livro)
```

```

def buscar_todos(self) -> list:
    return list(self.livros)

def obter_tamanho(self) -> int:
    return len(self.livros)

def obter_por_id(self, id: str) -> Livro:
    for livro in self.livros:
        if livro.id == id:
            return livro
    raise ExcecaoLivroNaoEncontrado

```

Fonte: do autor, 2022.

Para os testes de unidade, observe a Codificação 6.12 do arquivo test_stub_repositorio.py.

Codificação 6.12. test_stub_repositorio.py

```

import pytest

from tests.livroRepositorioStub import LivroRepositorioStub
from tests.livroRepositorioStub import ExcecaoLivroNaoEncontrado
from src.livroServico import LivroServico

@pytest.fixture
def servico():
    livroRepositorio = LivroRepositorioStub()
    livroServico = LivroServico(livroRepositorio)
    return livroServico

def test_objeto_stub_deve_devolver_contagem_igual_2(servico):
    assert servico.obter_tamanho() == 2

def test_busca_por_id_deve_devolver_livro_com_id_111(servico):
    l1 = servico.obter_por_id('111')

```

```
assert l1.id == '111'
```

```
def test_busca_por_id_deve_devolver_livro_com_id_222(servico):
```

```
    l1 = servico.obter_por_id('222')
```

```
    assert l1.id == '222'
```

```
def test_busca_por_id_333_deve_devolver_excecao(servico):
```

```
    with pytest.raises(ExcecaoLivroNaoEncontrado):
```

```
        l1 = servico.obter_por_id('333')
```

Fonte: do autor, 2022.

Como se pode observar pela Codificação 6.11, a classe **LivroRepositorioStub** ficou muito semelhante à classe **LivroRepositorioFake**, da seção 6.2.1, porém, é importante ressaltar que a classe **LivroRepositorioStub** contém valores fixos iniciados pelo construtor. Além disso, a classe **LivroRepositorioStub** poderia ter uma implementação mais simplificada e direcionada para os testes, como por exemplo, o método **obter_tamanho()** poderia devolver um número inteiro 2, diretamente. Ou então, o método **inserir** poderia ser suprimido, para estes testes. Para visualizar a organização dos arquivos observe a Figura 6.9.

Figura 6.9. Organização de arquivos para o exemplo de objetos *stubs*



Fonte: do autor, 2022.

Criando objetos *spies*

Os objetos *spies* são semelhantes aos objetos *stubs*, porém, eles registram informações sobre como eles são executados para serem utilizados em verificações nos testes de unidades. Para o exemplo de objetos *spies*, observe

a classe abstrata **LivroRepositorio**, que neste exemplo, contém apenas o método **inserir**, conforme a Codificação 6.13.

Codificação 6.13. livroRepositorio.py

```
from abc import ABCMeta, abstractmethod
from src.livro import Livro

class LivroRepositorio(metaclass=ABCMeta):

    @abstractmethod
    def inserir(self, livro: Livro):

        pass
```

Fonte: do autor, 2022.

Neste exemplo, a classe **LivroServico** também deve ser modificada de acordo com a Codificação 6.14.

Codificação 6.14. livroServico.py

```
from src.livro import Livro
from src.livroRepositorio import LivroRepositorio

class LivroServico:
    def __init__(self, livroRepositorio: LivroRepositorio):
        self.livroRepositorio = livroRepositorio

    def insere_livro(self, livro: Livro):

        self.livroRepositorio.inserir(livro)
```

Fonte: do autor, 2022.

Em seguida, crie a classe **LivroRepositorioSpy** de acordo com a Codificação 6.15.

Codificação 6.15. livroRepositorioSpy.py

```
from src.livro import Livro
from src.livroRepositorio import LivroRepositorio

class LivroRepositorioSpy(LivroRepositorio):
    def __init__(self):
```

```

self.__contagem_insercao = 0
self.__ultimo_livro = None

def inserir(self, livro: Livro):
    self.__contagem_insercao += 1
    self.__ultimo_livro = livro
def numero_de_insercoes(self):
    return self.__contagem_insercao

def verificar_ultimo_livro_inserido(self, id: str):

    return self.__ultimo_livro.id == id

```

Fonte: do autor, 2022.

De acordo com a Codificação 6.15, o método **inserir()** apenas conta o número de inserções e, também, armazena em uma variável privada o último livro inserido no repositório. A verificação é feita nos testes de unidade por meio dos métodos **numero_de_insercoes()** e **verificar_ultimo_livro_inserido()**, os quais não fazem parte do contrato definido pela classe abstrata **LivroRepositorio**. Para verificar os testes de unidade utilizando a classe **LivroRepositorioSpy**, observe a Codificação 6.16.

Codificação 6.16. test_spy_repositorio.py

```

import pytest
from datetime import date

from tests.livroRepositorioSpy import LivroRepositorioSpy
from src.livro import Livro
from src.livroServico import LivroServico

def test_inserir_2_livros_retornar_numero_de_insercoes_2():
    livroRepositorio = LivroRepositorioSpy()
    livroServico = LivroServico(livroRepositorio)
    l1 = Livro('111', 'Python para iniciantes', 50.0, date.today)
    l2 = Livro('222', 'Testes automatizados', 70.0, date.today)

```

```
livroServico.inserir_livro(l1)
livroServico.inserir_livro(l2)
assert livroRepositorio.numero_de_insercoes() == 2

assert livroRepositorio.verificar_ultimo_livro_inserido('222')
```

Fonte: do autor, 2022.

Observe a organização dos arquivos do exemplo de objetos *spies* pela Figura 6.10.

Figura 6.10. Organização de arquivos para o exemplo de objetos *spies*



Fonte: do autor, 2022.

Criando objetos *mocks*

Em testes de software, o termo *mock* é o mais conhecido e refere-se à capacidade de um objeto simular um objeto de produção cujos testes podem ser complexos. Alguns comportamentos podem ser substituídos por objetos *mocks*, tais como:

- objetos que geram resultados não determinísticos (ex. hora, temperatura, números aleatórios);
- objetos que possuem situações difíceis configurar, estados de erros difíceis de serem reproduzidos, ou que produzem algum tipo de efeito colateral (ex. falha de rede);
- objetos cujas classes ainda não foram construídos;
- objetos lentos, que necessitam que um banco de dados seja iniciado.

Perceba que todos esses comportamentos complexos são as situações tratadas por objetos *dublês* vistos nas seções anteriores, tais como, objetos *fake*, *stubs* ou até *spies*. Os objetos *mocks*, controlam fluxos, se colocam no lugar de outros objetos de produção, e ainda podem checar seus

resultados. Os objetos *mocks* têm comportamentos semelhantes a todos os outros objetos dupeiros, é possível afirmar que um objeto mock é uma mistura de *fake*, *stubs* e *spies* ao mesmo tempo. Para o exemplo de objetos *mocks*, observe a classe abstrata **LivroRepositorio**, que neste exemplo, contém apenas o método **obter_livro()**, de acordo com a Codificação 6.17.

Codificação 6.17. livroRepositorio.py

```
from abc import ABCMeta, abstractmethod
from src.livro import Livro

class LivroRepositorio(metaclass=ABCMeta):
    @abstractmethod
    def obter_livro(self, id: str) -> Livro:

        pass
```

Fonte: do autor, 2022.

Em seguida, crie a classe **LivroRepositorioMock** de acordo com a Codificação 6.18.

Codificação 6.18. livroRepositorioMock.py

```
from datetime import date

from src.livro import Livro
from src.livroRepositorio import LivroRepositorio

class ExcecaoLivroNaoEncontrado(Exception):
    pass

class LivroRepositorioMock(LivroRepositorio):
    def __init__(self):
        self.__contagem_chamadas = 0
        l1 = Livro('111', 'Python para iniciantes', 50.0, date.today())
        self.__livro = l1

    def obter_livro(self, id: str) -> Livro:
```

```

self.__contagem_chamadas += 1
if id == '111':
    return self.__livro
raise ExcecaoLivroNaoEncontrado

def numero_de_chamadas(self):
    return self.__contagem_chamadas

```

Fonte: do autor, 2022.

Para verificar os testes de unidade utilizando a classe **LivroRepositorioMock**, observe a Codificação 6.19.

Codificação 6.19. test_mock_repositorio.py

```

from tests.livroRepositorioMock import LivroRepositorioMock
from tests.livroRepositorioMock import ExcecaoLivroNaoEncontrado
from src.livro import Livro

```

```

def test_busca_livro_por_id():
    livroRepositorio = LivroRepositorioMock()
    livro = livroRepositorio.obter_livro('111')
    assert livro.id == '111'
    assert livro.titulo == 'Python para iniciantes'
    assert livroRepositorio.numero_de_chamadas() == 1

def test_busca_livro_por_id_inexistente_gera_excecao():
    with pytest.raises(ExcecaoLivroNaoEncontrado):
        livroRepositorio = LivroRepositorioMock()
        livroRepositorio.obter_livro('222')

```

Fonte: do autor, 2022.

Execute o pytest e verifique o resultado dos testes, conforme apresenta a Figura 6.11.

Figura 6.11. Execução do pytest para os objetos *mocks*



Fonte: do autor, 2022.

Uma biblioteca *mock* do Python

Em Python, existem várias bibliotecas que oferecem o recurso para criar objetos *mocks*, das quais uma das mais utilizadas é a biblioteca *mock* do arcabouço unittest. A utilização desta biblioteca *mock* facilita a criação de objetos *mocks*, e ajuda na construção de objetos que simulam objetos de produção que acessam recursos externos como mencionado nas seções anteriores. Para evitar confusão no texto, esta seção refere-se aos objetos *mocks* gerados pela biblioteca *mock* do unittest.

Para iniciar um objeto mock utilize a classe Mock, conforme apresenta a Figura 6.12.

Figura 6.12. Criando um objeto da classe Mock



Fonte: do autor, 2022.

Os objetos da classe Mock são flexíveis, podem assumir papel de qualquer entidade, tais como funções, classes ou métodos. Um mock pode interpretar qualquer objeto, como se pode observar na Figura 6.13. Observe, que pelas propriedades de tipagem dinâmica da linguagem Python, o objeto da classe Mock pode tomar a forma que for necessária para atender o teste de unidade, e sem a necessidade de definir a classe, como feito nas seções anteriores.

A Figura 6.13 indica que para criar um objeto da classe Mock é importante realizar a importação da biblioteca unittest.mock. Uma vez que o objeto mock é alocado na memória, ela possui uma identidade. A partir daí, é possível

definir um atributo, que neste exemplo, iniciou-se com o valor 10. Além disso, é possível indicar que o objeto mock pode invocar um método, que não foi definido o seu corpo, porém, possui uma identidade. Ao definir um valor de retorno para o método, por meio da propriedade **return_value**, é possível invocar o método simulado como se essa função tivesse implementação. Neste exemplo, o método **faz_algo()** devolve o valor 3.

Figura 6.13. Objeto da classe simulando um objeto



Fonte: do autor, 2022.

É possível determinar que o objeto mock simule uma função, como apresenta a Figura 6.14.

Figura 6.14. Objeto da classe simulando uma função



Fonte: do autor, 2022.

De acordo com a Figura 6.14, o exemplo realiza a importação da biblioteca `unittest.mock` com a classe `Mock`. Em seguida, declara uma variável mock que recebe uma instância da classe `Mock`. Após isso, determina que a propriedade `return_value` receba o valor 5. Ao invocar o objeto mock, com a sintaxe de uma função, o valor devolvido é o valor 5, como se havia previamente definido.

Utilizando objetos mocks em testes de unidade

Para ilustrar a utilização do objeto mock, crie o arquivo `test_mock_repositorio.py` de acordo com a Codificação 6.20.

Codificação 6.20. test_mock_repositorio.py

```
import pytest

from datetime import date
from unittest.mock import Mock
from src.livro import Livro

@pytest.fixture
def livro():
    l1 = Livro('111', 'Python para iniciantes', 50.0, date.today)
    return l1

def test_busca_livro_por_id(livro):
    livroRepositorio = Mock()
    livroRepositorio.obter_livro.return_value = livro
    livroRepositorio.numero_de_chamadas.return_value = 1
    resultado = livroRepositorio.obter_livro('111')

    assert resultado.id == '111'
    assert resultado.titulo == 'Python para iniciantes'
    assert livroRepositorio.numero_de_chamadas() == 1
```

Fonte: do autor, 2022.

A Codificação 6.20, apresenta a função `test_busca_livro_por_id`, que cria uma variável **livroRepositorio** e este recebe uma instância da classe `Mock`. Após isso, define-se que o **livroRepositorio** tem um método chamado **obter_livro**, que tem um valor de retorno um livro passado como fixture. Ainda, define-se um método **numero_de_chamadas** com valor de retorno igual a 1.

Para verificar como organizar os arquivos neste exemplo, observe a Figura 6.15.

Figura 6.15. Teste de unidade utilizando objetos mocks



Fonte: do autor, 2022.

A propriedade *side_effect* do objeto *mock*

A propriedade *side_effect* de um objeto *mock* aceita três tipos de objetos:

- *Exceptions*;
- Objetos *iterables*, como por exemplo as listas, tuplas, entre outros; e
- Funções (*callable*).

Para que o mock possa gerar uma exceção, basta atribuir à propriedade *side_effect* a exceção que deve ser gerada. Observe a Figura 6.16 para verificar essa situação.

Inicialmente, o exemplo executa a instrução para importar a biblioteca `unittest.mock` com a classe `Mock`. Em seguida, define-se uma variável para receber uma instância da classe `Mock`. Após isso, define-se que a propriedade *side_effect* da variável *mock* seja uma exceção do tipo **ValueError**. Por fim, ao executar o objeto mock, este lança a exceção `ValueError` que fora previamente definida.

Figura 6.16. O objeto *mock* gerando exceção



Fonte: do autor, 2022.

Caso a propriedade *side_effect* receba um objeto iterable, tal como uma lista, tupla, range ou algum outro objeto similar, o objeto mock produzirá como resultados os valores do objeto iterable. Observe a Figura 6.17 para essa situação.

De acordo com a Figura 6.17, o exemplo importa a biblioteca `unittest.mock` com a classe `Mock`. Define-se uma variável, de nome **funcao**, que recebe uma instância da classe `Mock`. Em seguida, a propriedade `side_effect` recebe uma lista de valores inteiros, que neste exemplo é `[4, 5, 6]`.

Ao executar o objeto *mock*, com sintaxe de chamada de função, este devolve o valor 4, que é o primeiro valor da lista. Ao executar novamente, o objeto *mock* devolve o valor 5, que é o segundo valor da lista. Ao executar uma terceira vez, o objeto *mock* devolve o número 6, que é o último valor da lista.

Perceba, que ao executar pela última vez, o Python lança uma exceção *StopIteration*, indicando que não há mais elementos da lista que fora atribuído à propriedade `side_effect` do objeto *mock*.

Figura 6.17. A propriedade `side_effect` com objetos *iterable*



Fonte: do autor, 2022.

A última situação, bastante útil, é quando a propriedade `side_effect` recebe uma função (*callable*). Observe esta situação na Figura 6.18.

Figura 6.18. A propriedade `side_effect` recebe uma função



Fonte: do autor, 2022.

De acordo com a Figura 6.18, o exemplo inicia, como nos exemplos anteriores, realizando a importação da biblioteca `unittest.mock` com a classe `Mock`. Em seguida, define-se uma função de nome **imprima_ola_mundo**. Na sequência, define-se um objeto da classe `Mock` e atribui à variável **funcao**. Então, define-se que a propriedade `side_effect` receba a função

imprima_ola_mundo. Ao executar o objeto *mock*, este comporta-se exatamente como a função **imprima_ola_mundo**.

Para um exemplo, mais complexo e interessante, observe a Figura 6.19.

Figura 6.19. A propriedade *side_effect* recebe uma função com parâmetros



Fonte: do autor, 2022.

De acordo com a Figura 6.19, o exemplo inicia, como nos exemplos anteriores, realizando a importação da biblioteca `unittest.mock` com a classe `Mock`. Em seguida, define-se uma função **imprime_numero**. Na sequência, define-se um objeto da classe `Mock` e atribui à variável **funcao**. Então, define-se que a propriedade *side_effect* receba a função **imprime_numero**. Ao executar o objeto *mock*, passando o valor 123 como parâmetro, este comporta-se exatamente como a função **imprime_numero** e imprime a mensagem 'Número: 123'.

Um outro exemplo, que trabalha com a ideia de função (*callable*), é apresentado na Figura 6.20.

Figura 6.20. A propriedade *side_effect* recebe uma função com parâmetros

2



Fonte: do autor, 2022.

De acordo com a Figura 6.20, o exemplo inicia realizando a importação da biblioteca `unittest.mock` com a classe `Mock`. Em seguida, define-se a classe de nome **MinhaClasse**, com o método construtor e o método **imprime_numero**. Na sequência, define-se um objeto da classe `Mock` e atribui à variável **classe_mock**. Então, define-se que a propriedade *side_effect* receba

a classe **MinhaClasse**. Após isso, define-se outro objeto mock de nome **obj_mock**, que recebe uma instância da classe **classe_mock**, com valor 123 que será transmitido ao construtor pela propriedade `side_effect`. Ao executar o método **imprime_valor** do objeto *mock* este imprime a mensagem 'Número: 123'.

Exemplo usando propriedade *side_effect*

Para utilizar a propriedade *side_effect* no projeto, observe a Codificação 6.21.

Codificação 6.21. `test_mock_repositorio.py`

```
import pytest

from datetime import date
from unittest.mock import Mock
from src.livro import Livro

class ExcecaoLivroNaoEncontrado(Exception):
    pass

@pytest.fixture
def livro():
    l1 = Livro('111', 'Python para iniciantes', 50.0, date.today())
    return l1

def test_busca_livro_por_id(livro):
    livroRepositorio = Mock()
    livroRepositorio.obter_livro.return_value = livro
    livroRepositorio.numero_de_chamadas.return_value = 1
    resultado = livroRepositorio.obter_livro('111')

    assert resultado.id == '111'
    assert resultado.titulo == 'Python para iniciantes'
    assert livroRepositorio.numero_de_chamadas() == 1
```

```
def test_busca_livro_por_id_inexistente_gera_excecao(livro):
    livroRepositorio = Mock()
    livroRepositorio.obter_livro.side_effect =
ExcecaoLivroNaoEncontrado

    with pytest.raises(ExcecaoLivroNaoEncontrado):
        livroRepositorio.obter_livro('222')
```

Fonte: do autor, 2022.

Observe a Figura 6.21, a função **test_busca_livro_por_id_inexistente_gera_excecao** inicia com a variável **livroRepositorio** recebendo uma instância de **Mock**. Em seguida, define-se um método **obter_livro** com propriedade *side_effect* recebendo a classe de exceção de nome **ExcecaoLivroNaoEncontrado**. Ao executar o método **obter_livro**, o objeto mock gera a exceção que fora previamente definida, e o teste será executado com sucesso, pois deve-se gerar a exceção caso o livro não seja encontrado.

Considerações finais

Conforme as técnicas de testes automatizados foram evoluindo e cada vez mais utilizados na indústria, surgiu a necessidade de utilização dos objetos duplês. É um processo natural, dado a complexidade dos testes e acoplamentos com dependências que utilizam-se de recursos externos. Utilize objetos mocks para tratar a propriedade de isolamento dos testes de unidade, eles sempre serão úteis em todos os casos.

Vamos praticar?

Neste exercício, utilize a biblioteca **requests** do python para realizar chamadas remotas a APIs (do inglês, *Application Programming Interface*).

1) Seja o código da classe **Blog** abaixo que realiza chamadas a uma API aberta, útil para testes, que devolve *posts* para um determinado *blog*.

blog.py

import requests

class **Blog**:

def **posts**(self):

 endereco = "https://jsonplaceholder.typicode.com/posts"

 response = requests.get(endereco)

 return response.json()

def **post_by_user_id**(self, **userId**: str):

 e = f"https://jsonplaceholder.typicode.com/posts/{userId}"

 response = requests.get(e)

 return response.json()

arquivo: **blog.py**

Perceba que a classe **Blog** acessa a API *jsonplaceholder*, que é um serviço API fake para testes. Para os testes de unidade, utilize objetos mock que simule o objeto da classe **Blog**. Crie testes para os métodos **posts** e **post_by_user_id**. Como sugestão para o desenvolvimento deste exercício, crie uma fixture que devolve uma lista de posts, no mesmo formato que o serviço API devolve, como apresenta o exemplo abaixo:

```
post = [{'userId' : 1,
        'Id' : 1,
        'title' : 'Titulo teste 1',
        'body' : 'Conteudo do blog 1'},
        {'userId' : 2,
        'Id' : 2,
        'title' : 'Titulo teste 2',
        'body' : 'Teste de conteudo do blog 2'}]
```

Referências

FOWLER, M. **Test double**. 2016. Disponível em: <<https://martinfowler.com/bliki/TestDouble.html>>. Acesso: 05 jul. 2022.

MESZAROS, G. **xUnit Test Patterns**: Refactoring Test Code. Addison-Wesley, 2007.

PYTEST. **Documentação versão de python 3.7+**. 2015. Disponível em: <<https://docs.pytest.org/en/7.1.x/index.html>>. Acesso em: 11 jun. 2022.

PYTHON. **Documentação versão de Python 3.10.5**. The python standard library. Development tools. unittest.mock - mock object library. Versão em inglês. Disponível em: <<https://docs.python.org/3/library/unittest.mock.html>>. Acesso: 08 jul. 2022.

SALE, D. **Testing python**: applying unit testing, TDD, BDD, and accepting testing. Wiley, 2014.