

Testando com Python

(Unitariamente)

@dunossauro

Roteiro

- Testes, pra que testes?
- Ciclo de feedback
- Quando escrever testes
- Quando rodar os testes
- O poder do assert
- xUnit-style
- unittest
- Definindo unidades
- SUT
- Fixtures
- Acoplamento
- Dublês de teste

Perguntas antes da gente começar

- O que são testes?
- Pra que servem os testes?
- Porque devo testar?
- Como devo testar?

Testes, pra que testes?

A ideia principal dos testes é garantir que a coisa funciona de fato.

Tá, isso é muito vago.

Então, para desenvolver um bom software ele precisa funcionar como esperado. E dependendo de como os testes são executados, mais rápido você vai ter um *feedback* de que ele está como deveria.

Ciclo de feedbacks

Então vamos imaginar que você escreva uma linha de código, pare de programar, execute o programa e veja se ele retorna o que precisa.

Sim, isso é um teste. Mas isso demora. Imagina toda linha de código escrita, uma média de 5 minutos pra subir e tudo e rodar. E quando a aplicação é mais complexa? Demora mais tempo.

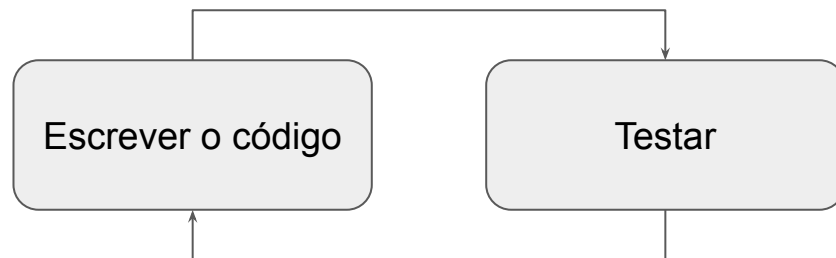
Ciclo de feedbacks

Temos outra situação,

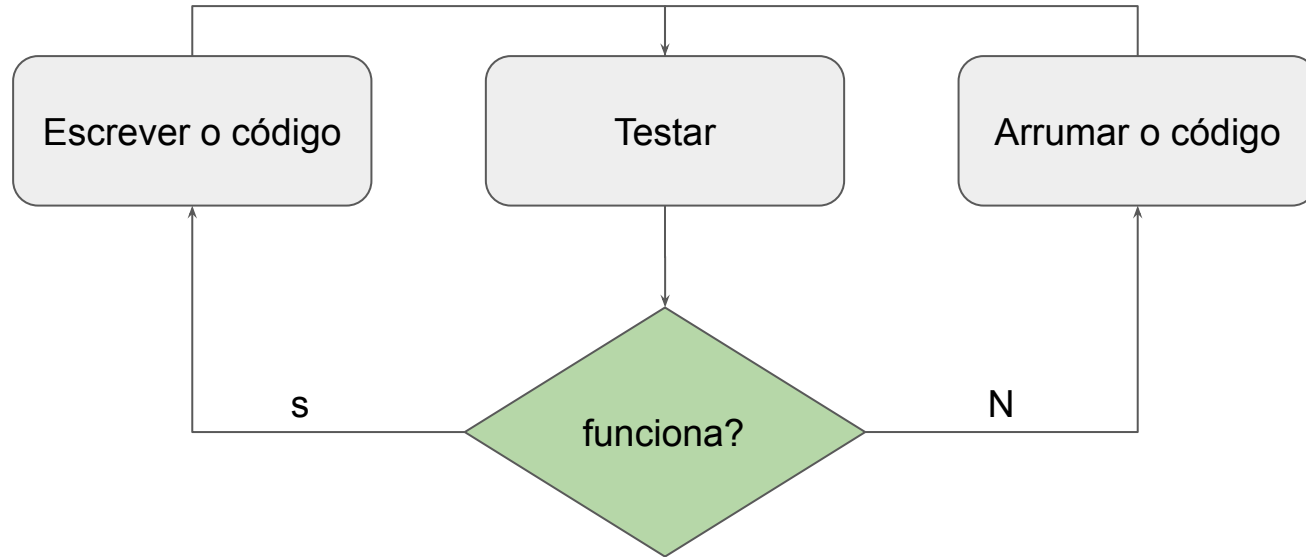
vamos imaginar que você tem um colega de trabalho que testa as coisas de todo mundo. Então você faz a modificação e passa pra ele. Só que ele tem outros testes para fazer. Logo você vai ter que testar algumas vezes antes dele testar de fato, pq você precisa ver se aqui pode ser mandado pra ele.

Mesmo que você não teste, esse ciclo vai demorar.

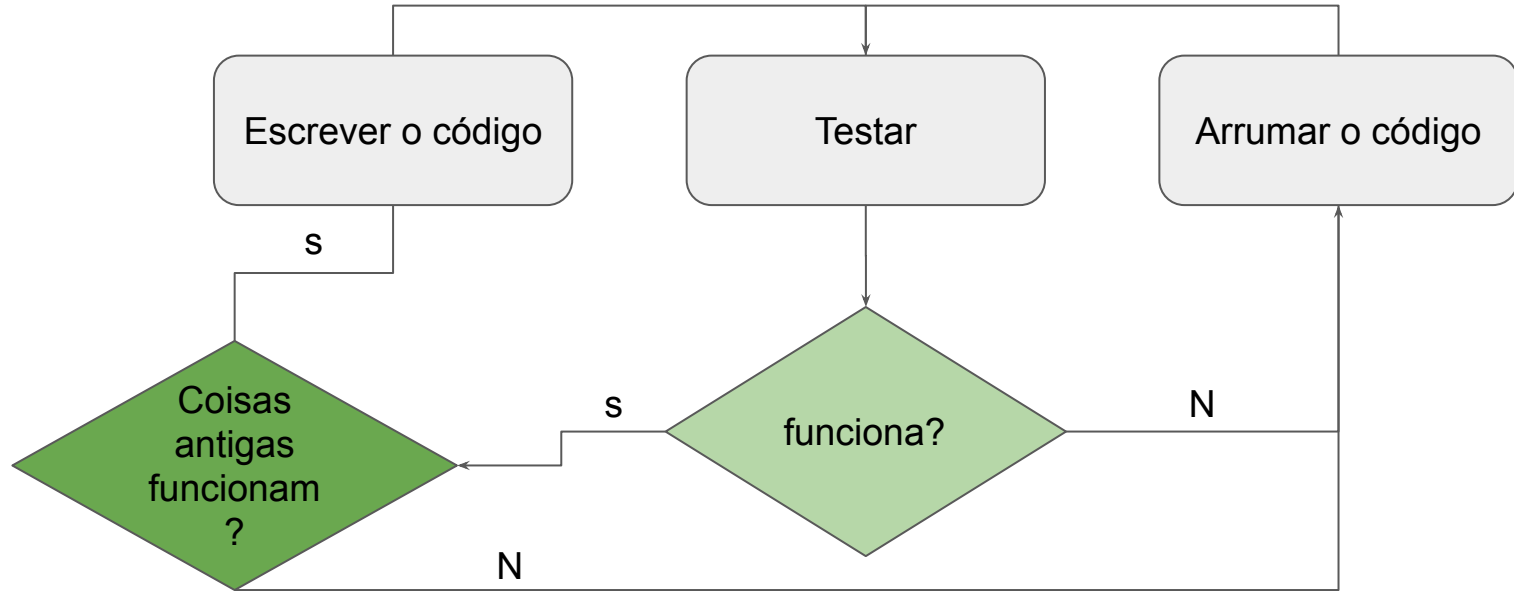
Ciclo de feedbacks



Ciclo de feedbacks



Ciclo de feedbacks



Ciclo de feedbacks

Vamos pensar que cada vez que você para tudo e testa você gasta 1 minuto pra fazer cada teste. Se você tem 30 unidades (funções, classes, etc..) Cada vez que você mudar algo vai levar 30m pra testar. Se você mudar bastante coisa, vai levar mais de um dia...

Espera... Computadores não são bons em executar tarefas trabalhosas e repetitivas?

Quando escrever testes?

Acho que a pergunta foi respondida, não é mesmo? O tempo todo...

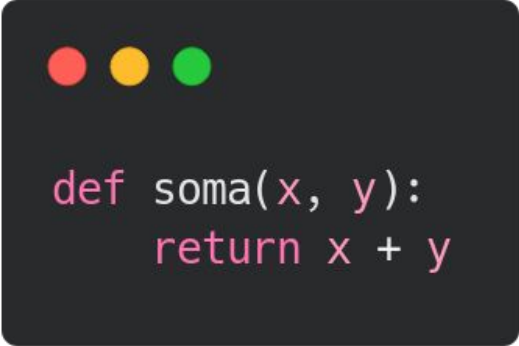
O legal de escrever testes é que diferente das tarefas manuais completamente chatas... Eles vão durar pra sempre.

Quando rodar seus testes?

Acho que também já foi respondido. Quanto mais rodar, mais feedback.

Problema #1

Dada uma função de soma:



```
def soma(x, y):  
    return x + y
```

Como ela poderia ser testada?


O poder da palavra assert

A palavra reservada **assert** faz uma “afirmação” em uma expressão em python. Ou seja, ela afirma que determinada expressão é verdadeira

```
1 """
2 Esquema de afirmação
3 """
4
5 # expressão completa
6 assert <exp_1> <cond> <exp_2>, "Error message"
7
8 # Forma mais simples
9 assert <exp>
10
11 # Exemplo: 2 é igual a 2
12 assert 2 == 2
13
14 # Exemplo: com chamada
15 assert soma(2, 3) != 0
```

Problema #2

Dada uma função de soma:



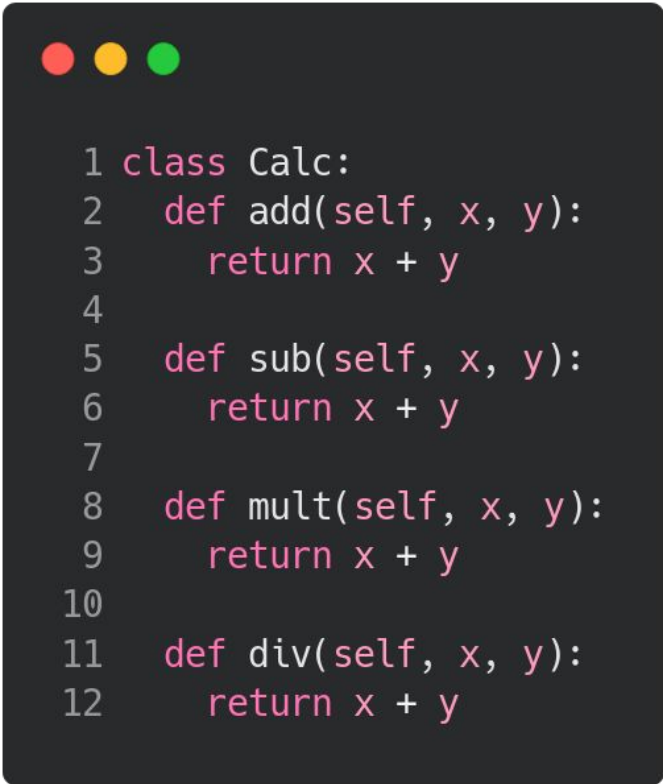
```
def soma(x, y):  
    return x + y
```

Faça assertivas de valores que você acredita serem coerentes com a função

Problema #3

Dado o código:

Faça assertivas em relação ao que funciona e não funciona nos seus casos de teste



```
1 class Calc:
2     def add(self, x, y):
3         return x + y
4
5     def sub(self, x, y):
6         return x + y
7
8     def mult(self, x, y):
9         return x + y
10
11     def div(self, x, y):
12         return x + y
```


xUnit-Style

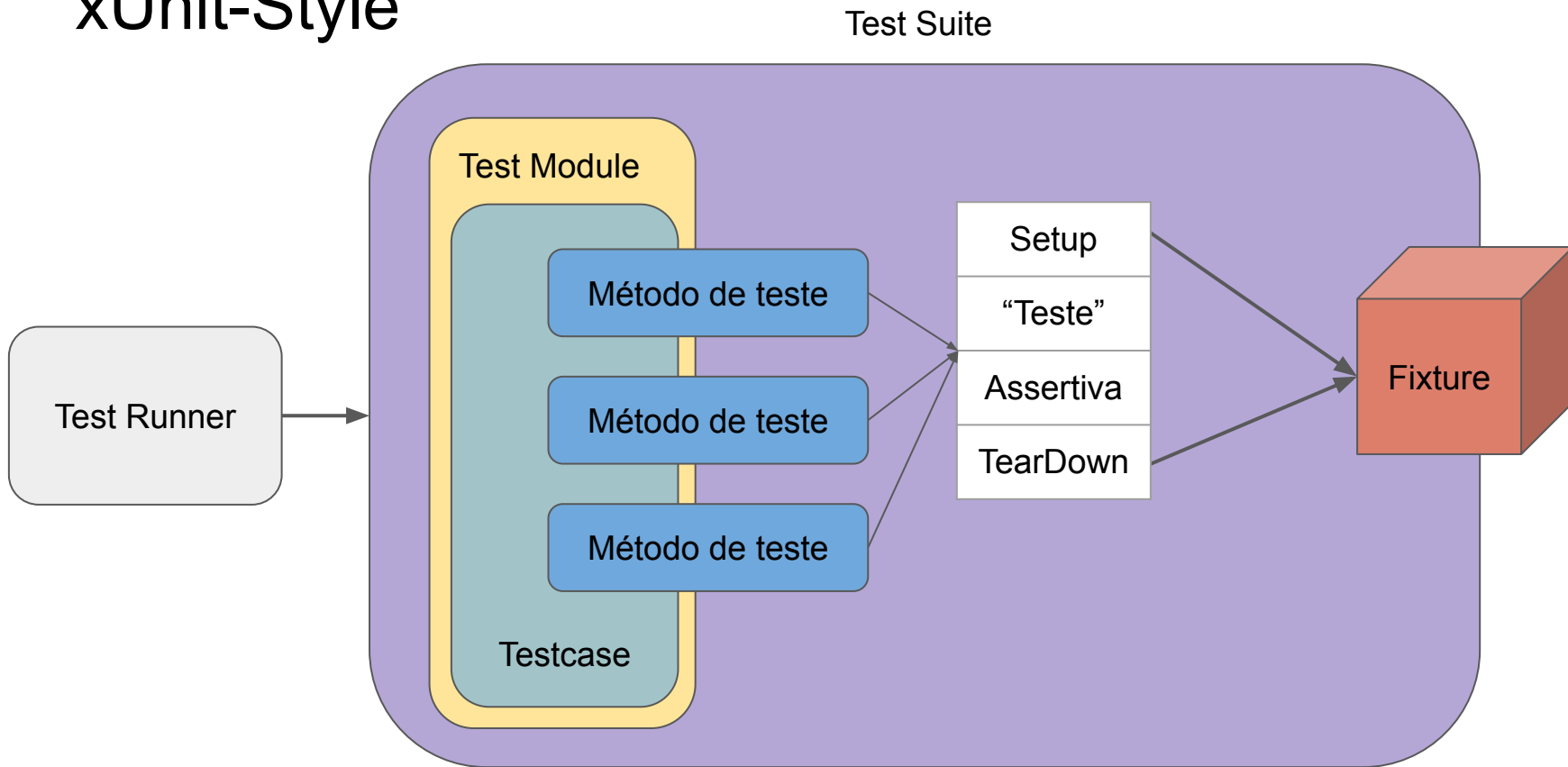
xUnit é uma filosofia de frameworks de testes. Ele nasceu no Smalltalk e lá se chamava SUnit.

O fato de ser baseado no smalltalk torna tudo MUITO burocrático. Mas isso é detalhe.

xUnit-Style - Glossário

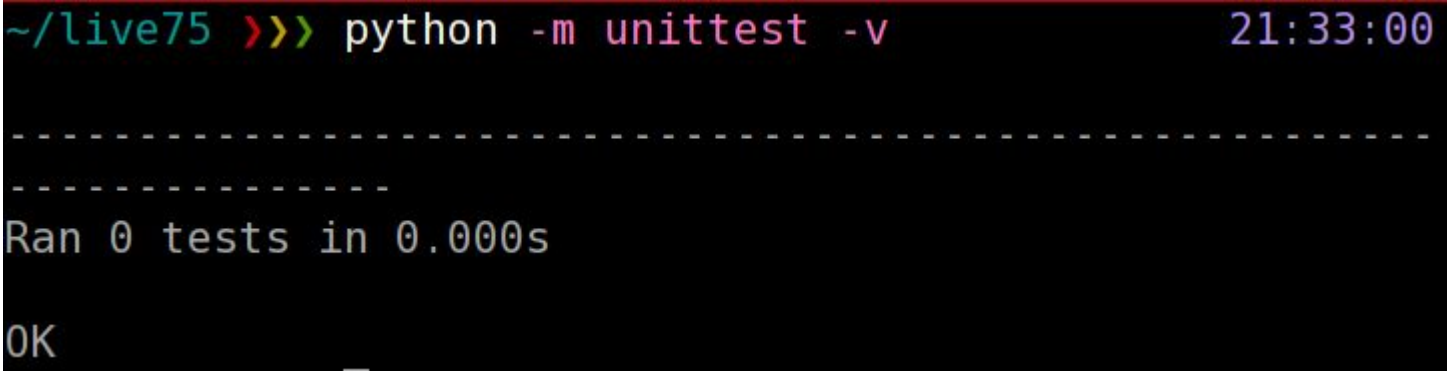
- **Test case class**
 - Classe base para todas as classes de teste
- **Test fixtures**
 - Funções ou métodos que são executados antes e depois da execução dos blocos do código de teste.
- **Assertions**
 - Funções ou métodos são usados para verificar o comportamento do componente que está sendo testado
- **Test runner**
 - Programa ou bloco de código que executa o conjunto de testes.

xUnit-Style



xUnit-Style

Test Runner



```
~/live75 >>> python -m unittest -v                                     21:33:00
-----
-----
Ran 0 tests in 0.000s

OK
```

A terminal window with a black background and colored text. The prompt is ~/live75 in green, followed by >>> in red and green. The command python -m unittest -v is in pink. The time 21:33:00 is in purple. The output shows two dashed lines, then 'Ran 0 tests in 0.000s' in white, and 'OK' in white at the bottom.

xUnit-Style

Testcase

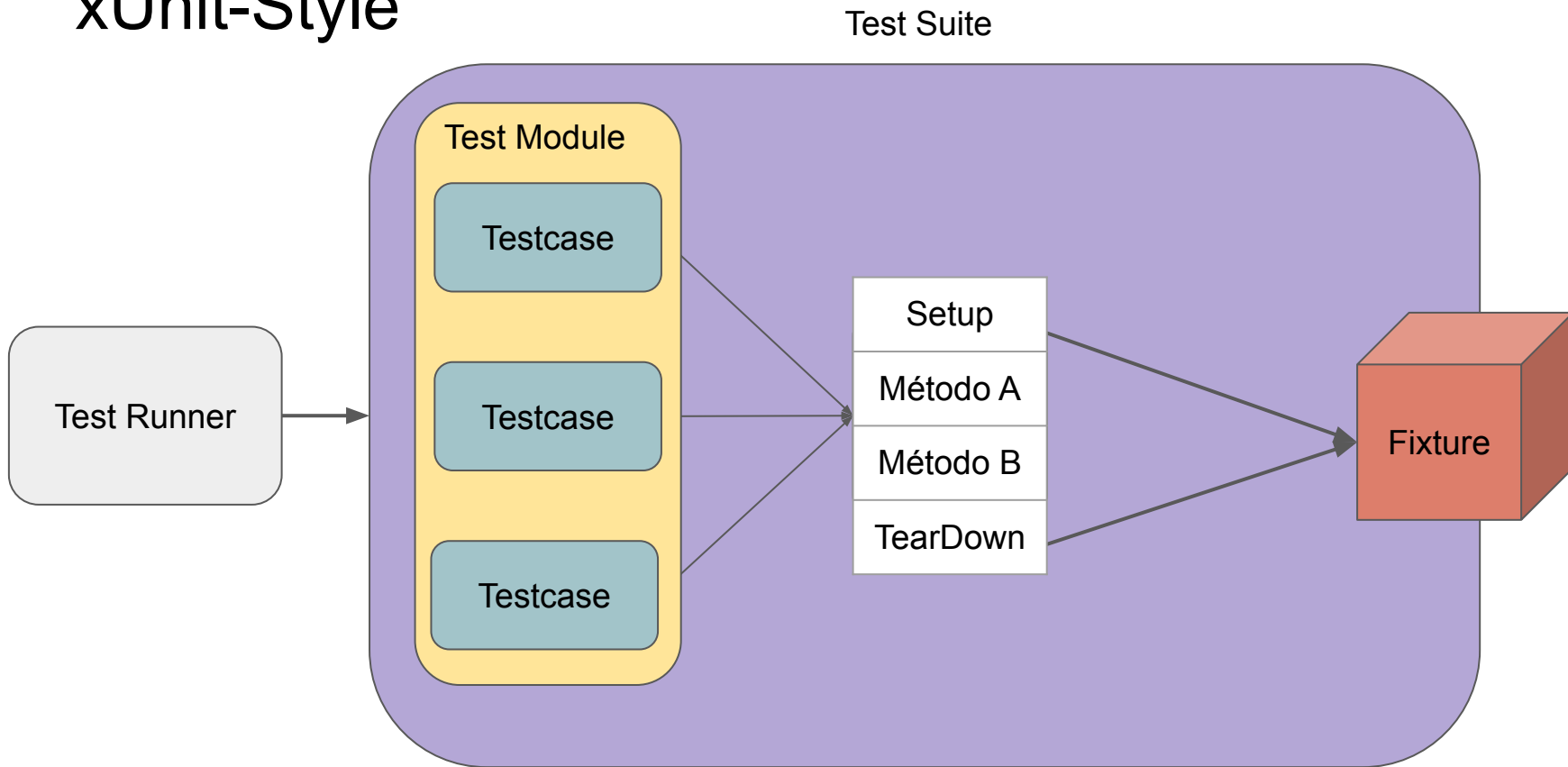
```
class TestCase(unittest.TestCase):  
    def setUp(self):  
        print("\nNo setUp()...")  
  
    def tearDown(self):  
        print("No tearDown()...")  
  
    def test_case01(self):  
        print("No test_case01()")  
        self.assertEqual('a', 'a', msg='a é igual a')
```

Fixtures

Método de teste

Assertiva

xUnit-Style



xUnit-Style

Testcase

```
class TestCase(unittest.TestCase):  
    @classmethod  
    def setUpClass(cls):  
        print("No setUpClass()...")  
  
    @classmethod  
    def tearDownClass(cls):  
        print("No tearDownClass()...")  
  
    def setUp(self):  
        print("\nNo setUp()...")  
  
    def tearDown(self):  
        print("No tearDown()...")  
  
    def test_case01(self):  
        print("No test_case01()")
```

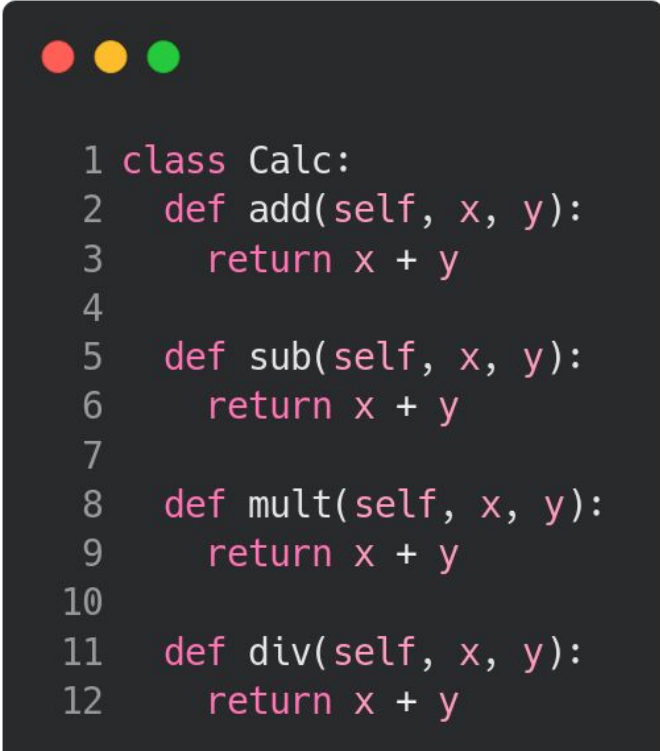
Fixtures
da
classe

Fixtures
dos
métodos

Método de teste

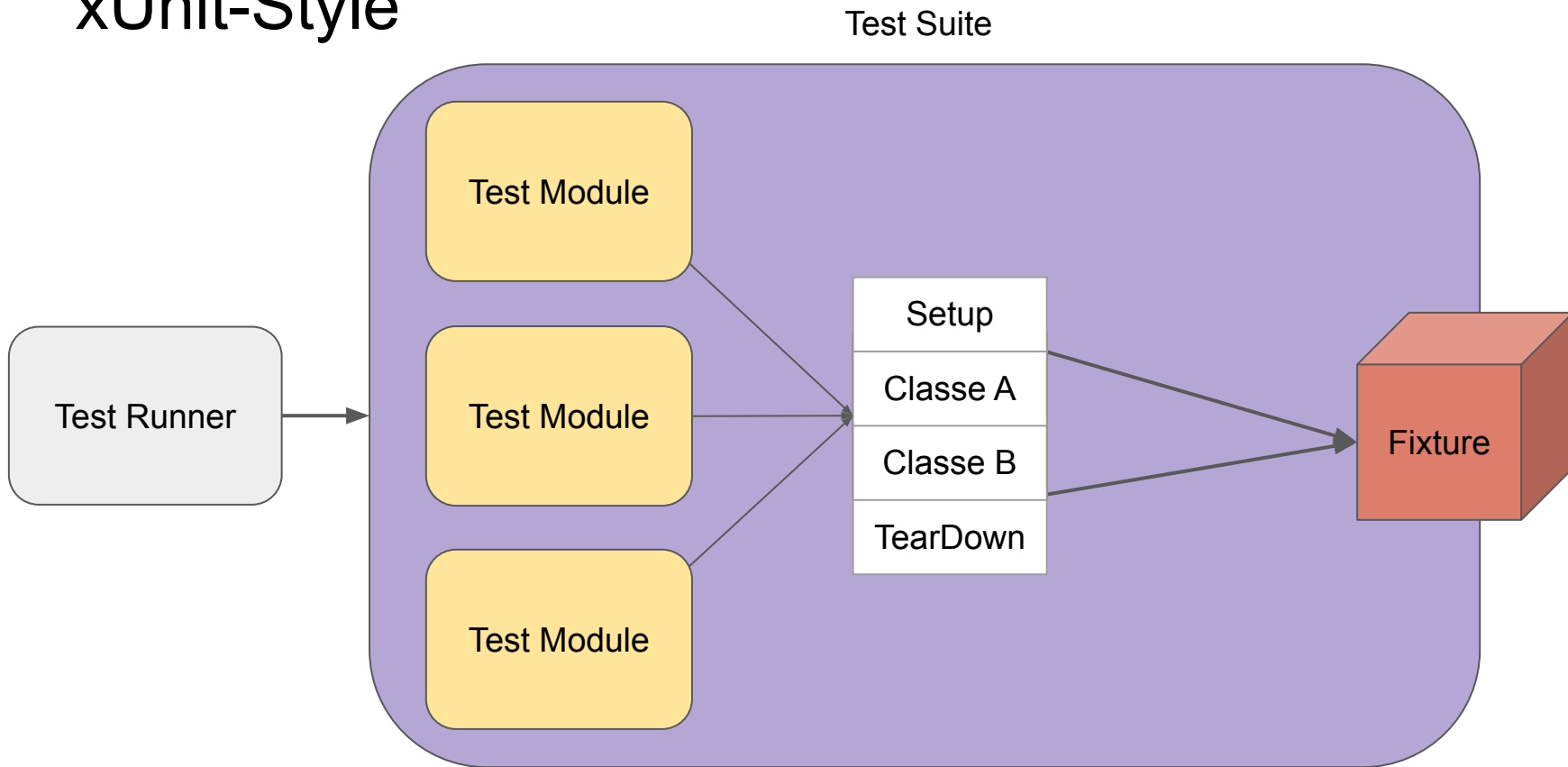
Problema #4

Crie sua primeira suite de testes para a classe da calculadora.



```
1 class Calc:
2     def add(self, x, y):
3         return x + y
4
5     def sub(self, x, y):
6         return x + y
7
8     def mult(self, x, y):
9         return x + y
10
11     def div(self, x, y):
12         return x + y
```


xUnit-Style



xUnit-Style

Test
Module

```
• def setUpModule():  
    print("No setUpModule()...")  
  
• def tearDownModule():  
    print("No tearDownModule()...")  
  
• class TestCase(unittest.TestCase):  
    @classmethod  
    • def setUpClass(cls):  
        print("No setUpClass()...")  
  
        @classmethod  
    • def tearDownClass(cls):  
        print("No tearDownClass()...")  
  
    • def setUp(self):  
        print("\nNo setUp()...")  
  
    • def tearDown(self):  
        print("No tearDown()...")  
  
    • def test_case01(self):  
        print("No test_case01()...")
```

Fixtures
do
módulo

Testcase

Fixtures
da
classe

Fixtures
dos
métodos

Método de teste

Definindo unidades

Quando se fala em unidades pode-se entender muita coisa dependendo do “estilo” de código que se faz. E essa definição é um tanto quanto complexa.

Em POO, a menor unidade é um objeto. Em funcional, a menor é uma função.

MAAAAASSSSSS....

Definindo unidades

Python é uma linguagem mista (multi paradigma). Então vou me referir a tudo como módulos¹.

Função -> Módulo

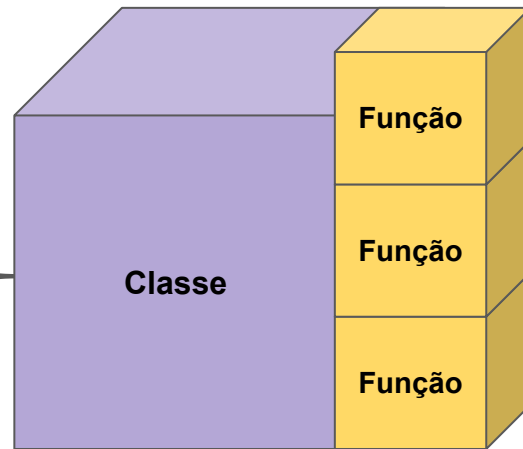
Objeto -> Módulo

São coisas que interagem entre si.

Definindo unidades

Em alguns casos, pode-se pensar em classes “maiores” que uma unidade.

Mas tudo que existe na classe está “acoplado” na mesma. É impossível isolar parte de classe



Definindo Unidades

Testes de unidade devem prezar por¹:

- **Isolado:**
 - O teste de unidade não pode conter dependências externas (bancos de dados, apis e etc)
- **Stateless:**
 - Não se pode guardar estados, ou seja, a cada teste todos os recursos que foram utilizados (instâncias, mocks e tudo mais) devem ser destruídos completamente e novos devem ser criados
- **Unitário:**
 - É um pouco redundante dizer isso, mas um teste de unidade deve apenas testar **uma** unidade, ou seja, se você começar a instanciar outras unidades já não é mais um teste unitário.

SUT (System Under Test)

Geralmente, “o que está sendo testado” na bibliografia aparece como SUT, mas pode aparecer com nomes diferentes, como²:

- SUT: Sistema em teste (A coisa em si)
 - CUT: Classe em teste
 - MUT: método em testes
 - AUT: Tudo em testes (All)
 - DOC: Componente de quem o SUT depende
-
- FUT: Função em teste
 - M2UT: Módulo em teste

SUT (System Under Test)

Geralmente, “o que está sendo testado” na bibliografia aparece como SUT, mas pode aparecer com nomes diferentes, como²:

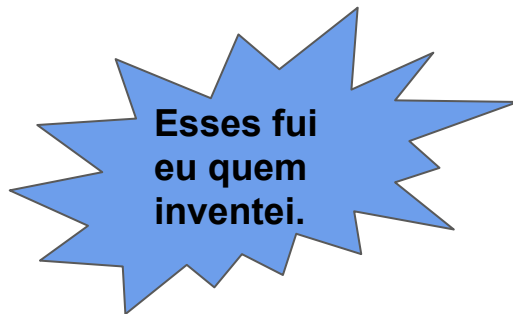
- **SUT**: Sistema em teste (A coisa em si)
- CUT: Classe em teste
- MUT: método em testes
- AUT: Tudo em testes (All)
- **DOC**: Componente de quem o SUT depende

- FUT: Função em teste
- M2UT: Módulo em teste

SUT (System Under Test)


Geralmente, “o que está sendo testado” na bibliografia aparece como SUT, mas pode aparecer com nomes diferentes, como²:

- SUT: Sistema em teste (A coisa em si)
- CUT: Classe em teste
- MUT: método em testes
- AUT: Tudo em testes (All)
- DOC: Componente de quem o SUT depende
- FUT: Função em teste
- M2UT: Módulo em teste



Problema #5

Data a classe da calculadora. Defina difentes SUTs e os teste individualmente.

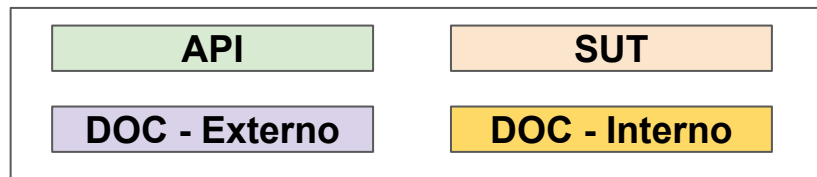


```
1 class Calc:
2     def add(self, x, y):
3         return x + y
4
5     def sub(self, x, y):
6         return x + y
7
8     def mult(self, x, y):
9         return x + y
10
11     def div(self, x, y):
12         return x + y
```

Problema #6

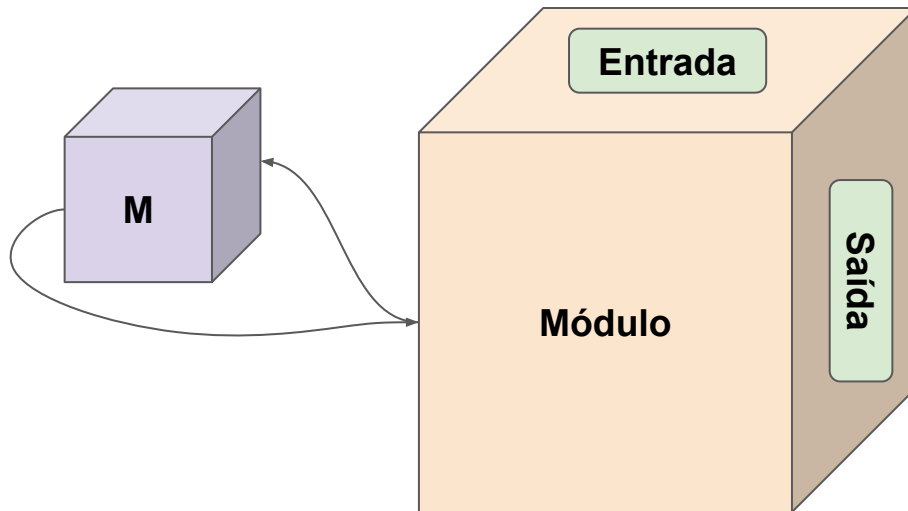
Agora que você tem os testes para garantir o comportamento dos seus métodos. Você poderia efetuar a correção da classe sem grandes problemas?

Acoplamento

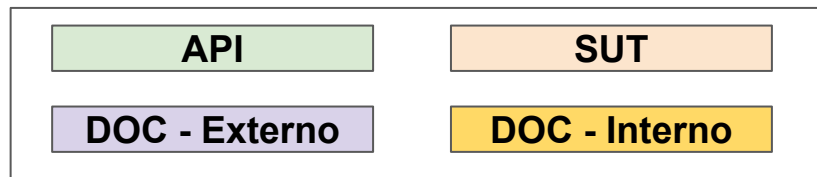


Não é tão complicado entender o sentido de acoplamento.

O quanto uma coisa depende de outra. Por exemplo o módulo A faz uma chamada para B que retorna para A que retorna pra quem chamou.

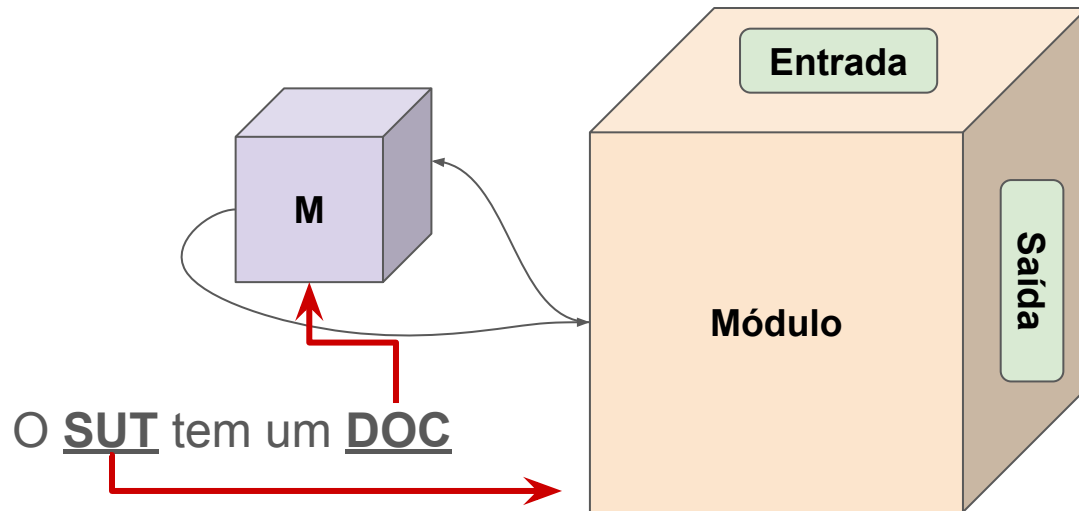


Acoplamento

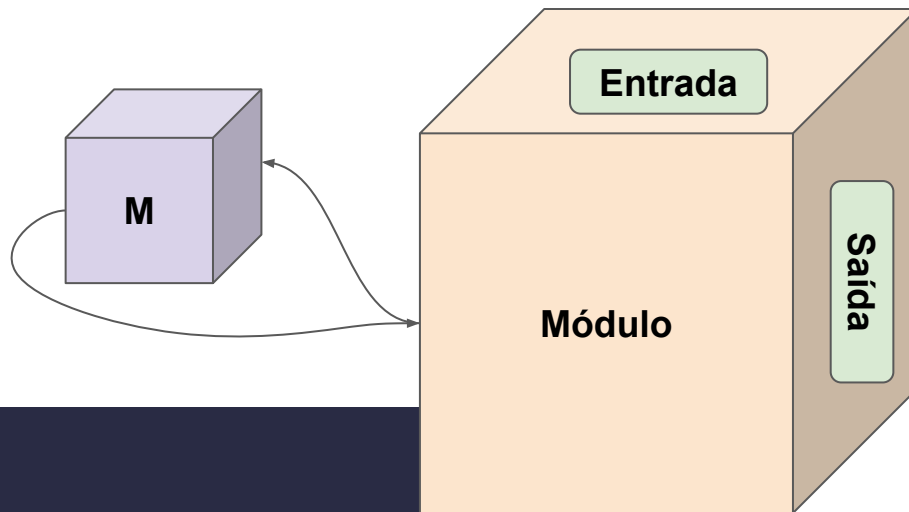
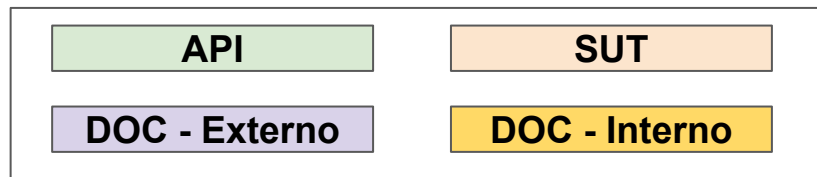


Não é tão complicado entender o sentido de acoplamento.

O quanto uma coisa depende de outra. Por exemplo o módulo A faz uma chamada para B que retorna para A que retorna pra quem chamou.



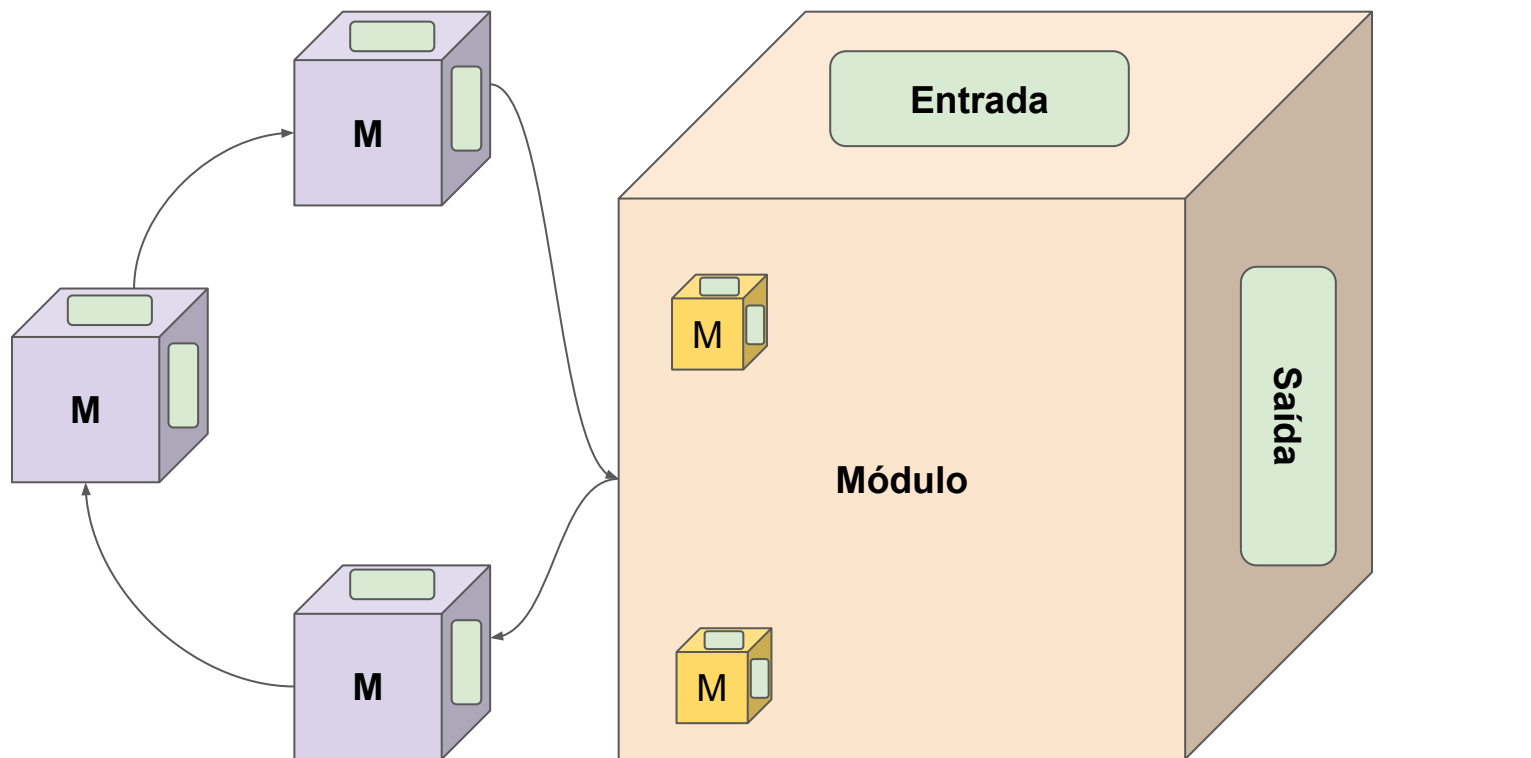
Acoplamento



```
from requests import get
```

```
def page_content(url: str, ssl: bool = False, *, params=None) -> str:  
    prefix = 'https' if ssl else 'http'  
    return get(f'{prefix}://{url}', params).content.decode()
```

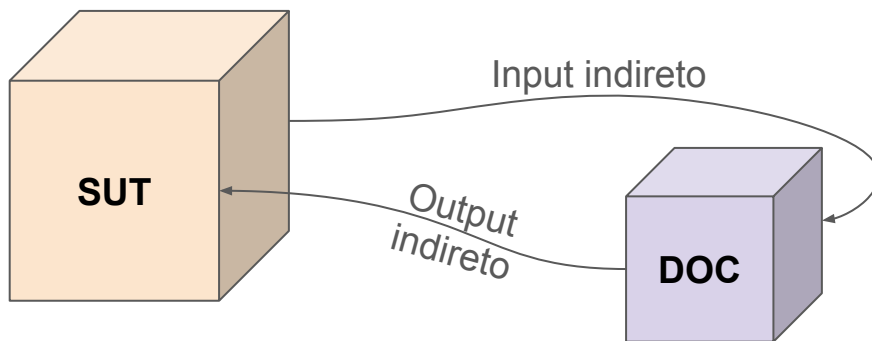
Acoplamento



Inputs e outputs **INDIRETOS**

Enquanto chamamos o SUT ele pode conversar com diversos DOCs.

Por exemplo, imagine que possa produzir algum tipo de efeito colateral?



Problema #7

Dada que exista uma função de soma e uma de subtração.

Crie uma função chamada 'exp' que receba x, y e z. E calcule:

$$f(x, y, z) \rightarrow x + y - z$$

Faça o teste dessa função, testando seus inputs e outputs indiretos

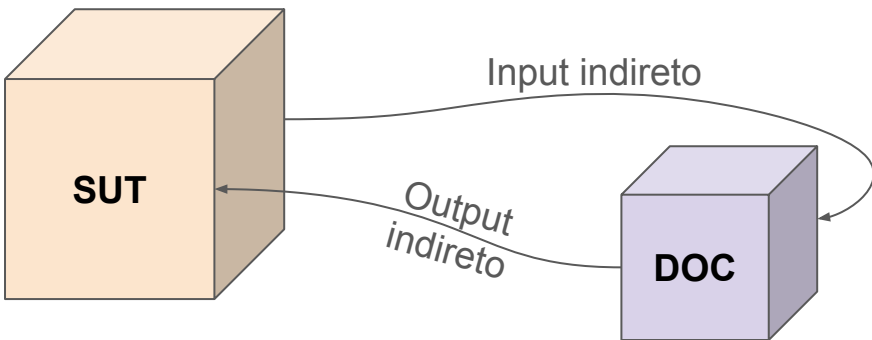
Problema #8

Agora que você já contruiu a função 'exp' que tal fazer uma chamada de uma expressão mais complexa?

$f(x, y, z) \rightarrow \exp(\exp(x, y, z), y, z)$

Agora tente efetuar os testes dessa função

Inputs e outputs **INDIRETOS**



Muito otimismo pensar que NADA vai dar errado no 'get'.

```
def endpoint_up(endpoint_url) -> tuple:
    if not endpoint_url.startswith('http'):
        endpoint_url = f'http://{endpoint_url}'

    request = get(endpoint_url)
    status_code = request.status_code

    # se retornar OK
    if status_code in [200, 201, 202, 302, 304]:
        return True, request.status_code

    # se der erro no endpoint
    elif request.status_code in range(500, 506):
        return False, 'bad request'

    # qualquer coisa não prevista
    else:
        return False, 'Deu ruim'
```

A pergunta de 1M de dólares

Como podemos verificar a lógica de um módulo, independentemente, quando ele depende de outro módulo? (Quando há acoplamento)

A resposta de 1M de dólares

Substituímos um componente do qual o SUT
depende com um
"equivalente específico de teste".

A resposta de 1M de dólares

Substituímos um componente do qual o SUT
depende com um

~~"equivalente específico de teste".~~

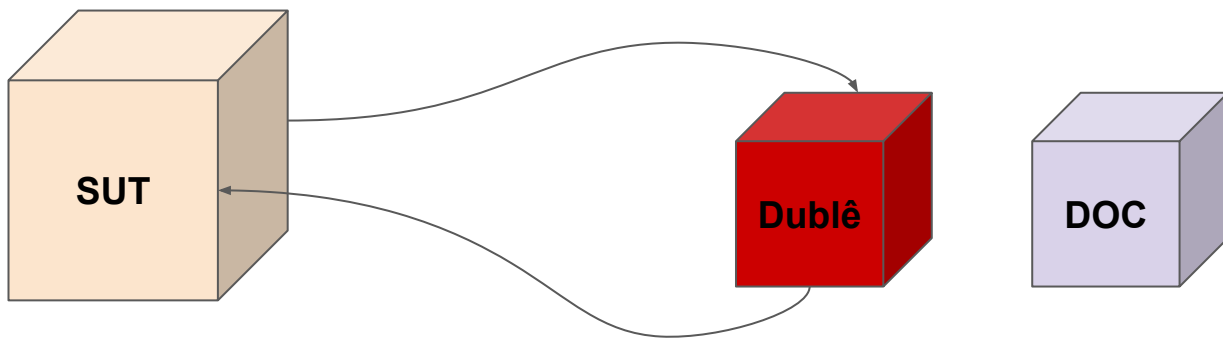
DUBLÊ

Dublês de teste



Dublês de teste

Então você já deve ter entendido que o dublê faz o trabalho para que os testes possam ser **determinísticos**. Ele vai de ajudar a construir o ambiente que você precisa, tirar o DOC da parada e colocar um cara que a gente sabe exatamente o que ele vai fazer.



Dublês de teste

Existem vários tipos de dublês

- **Dummy**
 - São os dublês mais simples. Imagine um parâmetro nulo, algo que o SUT tenha DOC, mas não entra no escopo do teste.
- **Fake**
 - A ideia do Fake é prover uma implementação de “mentira” para que o SUT consiga seguir o seu fluxo
- **Spy**
 - Tem a função de validar os inputs indiretos
- **Stub**
 - Substituem o DOC em um ambiente controlado
- **Mock:**
 - ???

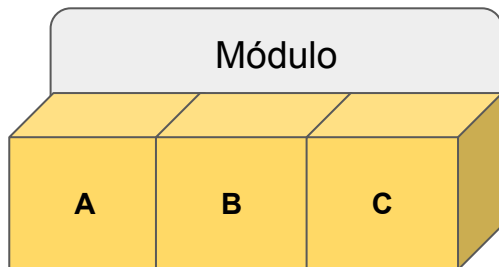
Dublês de teste

Pra que dublê nessa porra?

Vou testar tudo acoplado



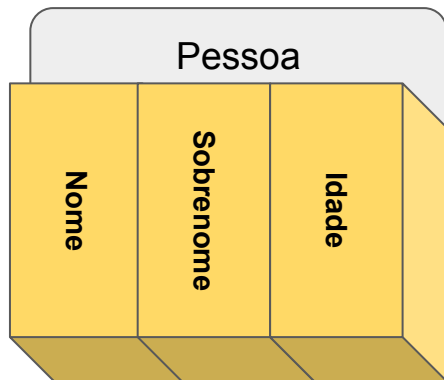
Dummy



Vamos supor que para que o módulo ser testado ele necessariamente precisa receber algo. Ou algo precisa existir para que ele funcione.

A ideia principal do Dummy não é que ele seja um objeto Nulo, puro e simplesmente, é que ele seja irrelevante para o SUT

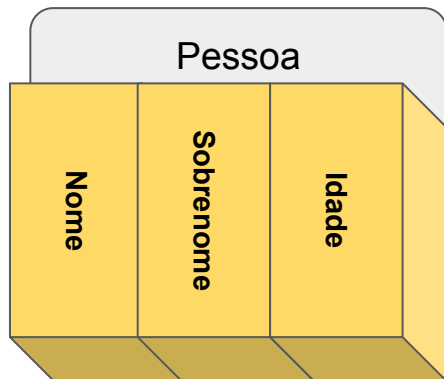
Dummy



```
class Pessoa:
    def __init__(self, nome, sobrenome, idade):
        self.nome = nome
        self.sobrenome = sobrenome
        self.idade = idade

Pessoa('Eduardo', 'Mendes')
```

Dummy

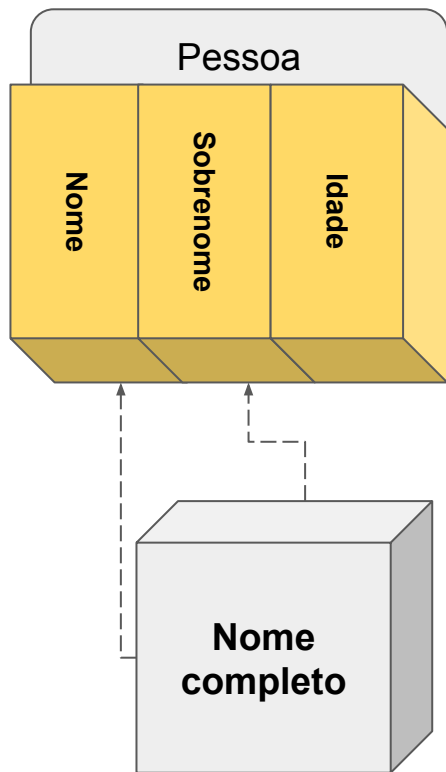


```
class Pessoa:
    def __init__(self, nome, sobrenome, idade):
        self.nome = nome
        self.sobrenome = sobrenome
        self.idade = idade

Pessoa('Eduardo', 'Mendes')
```

```
Traceback (most recent call last):
  File "dummy_exemplo.py", line 8, in <module>
    Pessoa('Eduardo', 'Mendes')
TypeError: __init__() missing 1 required positional argument: 'idade'
```

Dummy



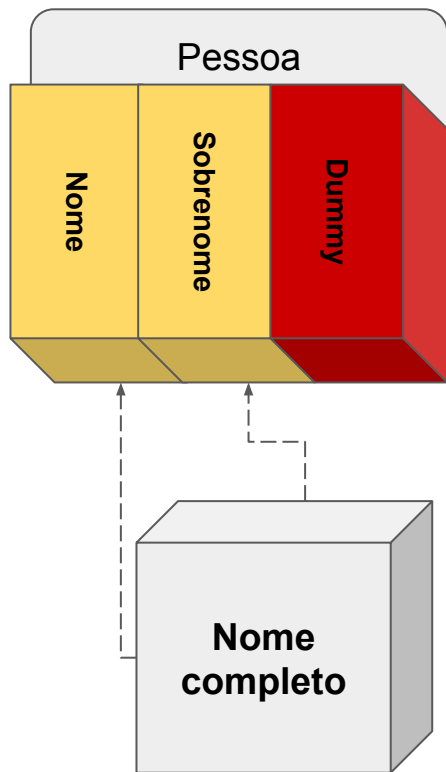
SUT / MUT

```
class Pessoa:
    def __init__(self, nome, sobrenome, idade):
        self.nome = nome
        self.sobrenome = sobrenome
        self.idade = idade

    @property
    def nome_completo(self):
        return f'{self.nome} {self.sobrenome}'

Pessoa('Eduardo', 'Mendes', '?????')
```

Dummy



SUT / MUT

```
from typing import NewType, Any

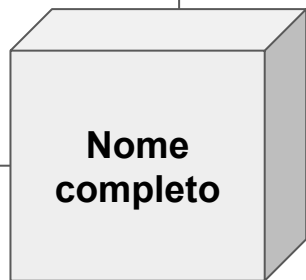
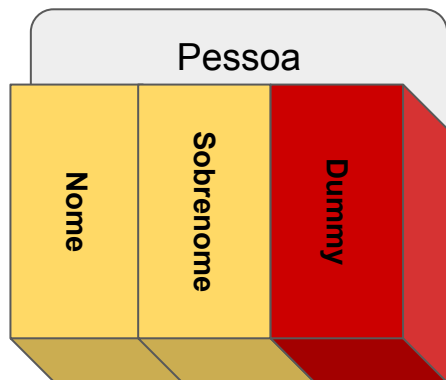
Dummy = NewType('Dummy', Any)

class Pessoa:
    def __init__(self, nome, sobrenome, idade):
        self.nome = nome
        self.sobrenome = sobrenome
        self.idade = idade

    @property
    def nome_completo(self):
        return f'{self.nome} {self.sobrenome}'

Pessoa('Eduardo', 'Mendes', Dummy(25))
```

Dummy



```
from typing import NewType, Any

Dummy = NewType('Dummy', Any)

class Pessoa:
    def __init__(self, nome, sobrenome, idade):
        self.nome = nome
        self.sobrenome = sobrenome
        self.idade = idade

    @property
    def nome_completo(self):
        return f'{self.nome} {self.sobrenome}'

Pessoa('Eduardo', 'Mendes', Dummy(25))
```


Problema #9

Usando a função 'exp' criada no problema 7. Teste a validação indireta da função de soma usando um dummy

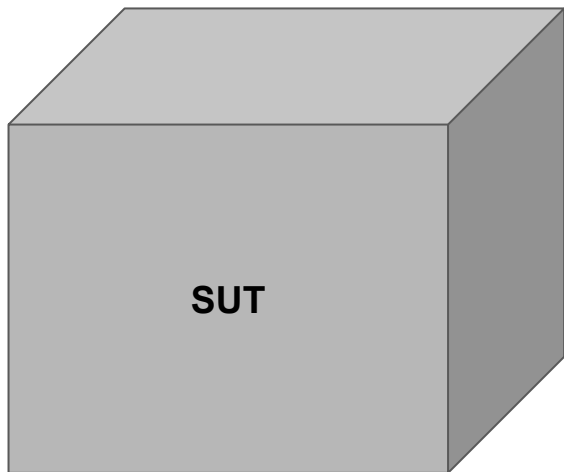
```
def exp(x, y, z):  
    return sub(soma(x, y), z)
```

Problema #10

Usando a função 'exp' criada no problema 8. Teste a validação indireta da função de subtração usando um dummy

```
def exp(x, y, z):  
    return sub(soma(x, y), z)
```

Spy



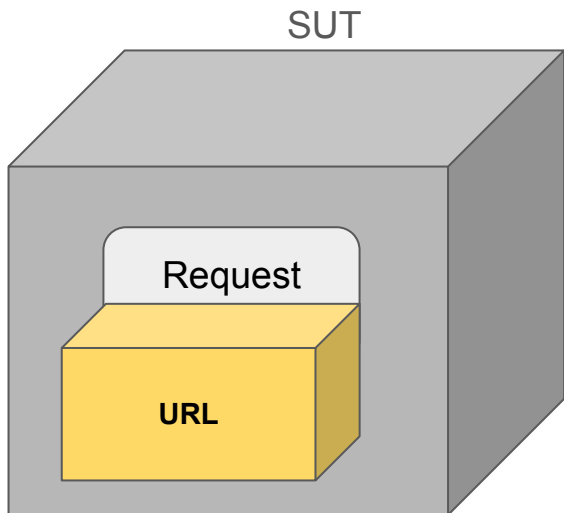
Spies são a KGB dos dublês. A ideia principal deles é saber quando o DOC foi chamado, com que valores ele foi chamado, quantas vezes foi chamado...

Você entendeu...

“Procedural Behavior Verification” é o termo técnico

A função deles é exercitar os inputs indiretos e validar se o DOC foi de fato executado

Spy



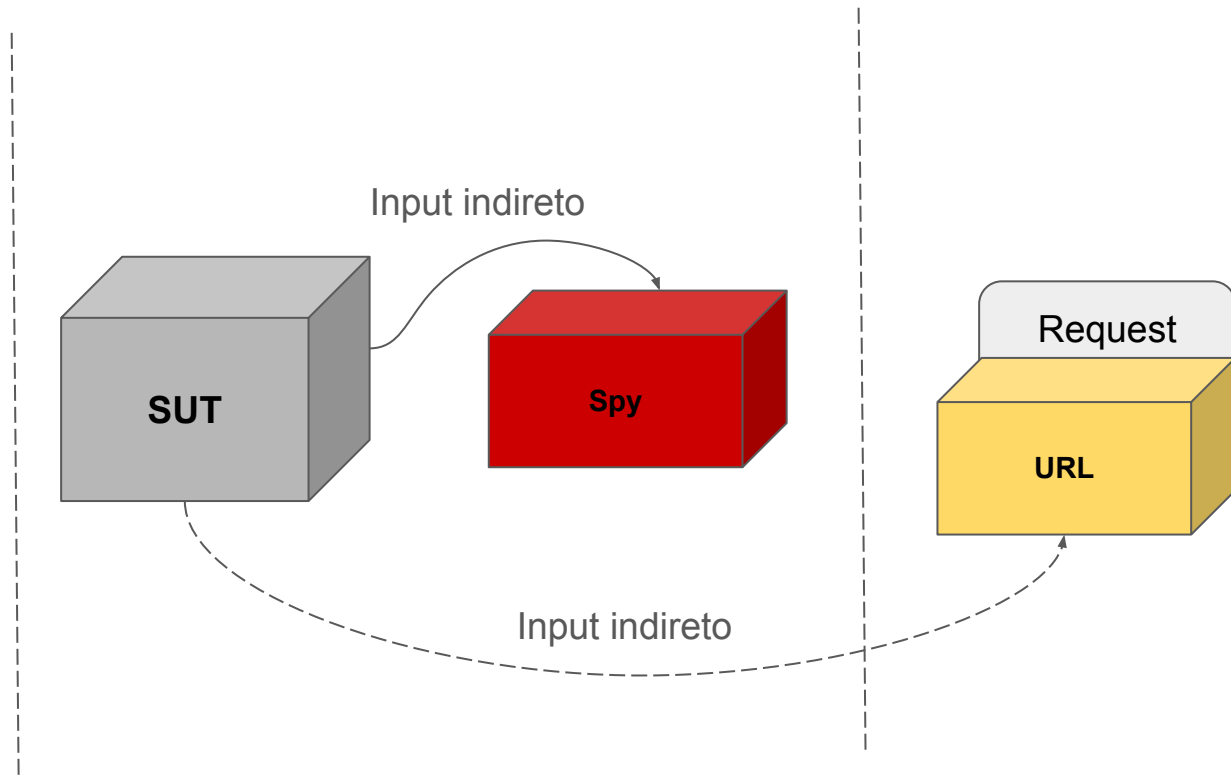
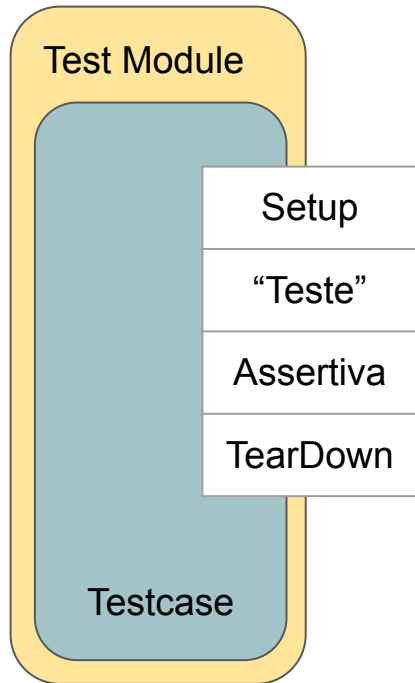
Spies são a KGB dos dublês. A ideia principal deles é saber quando o DOC foi chamado, com que valores ele foi chamado, quantas vezes foi chamado...

Você entendeu...

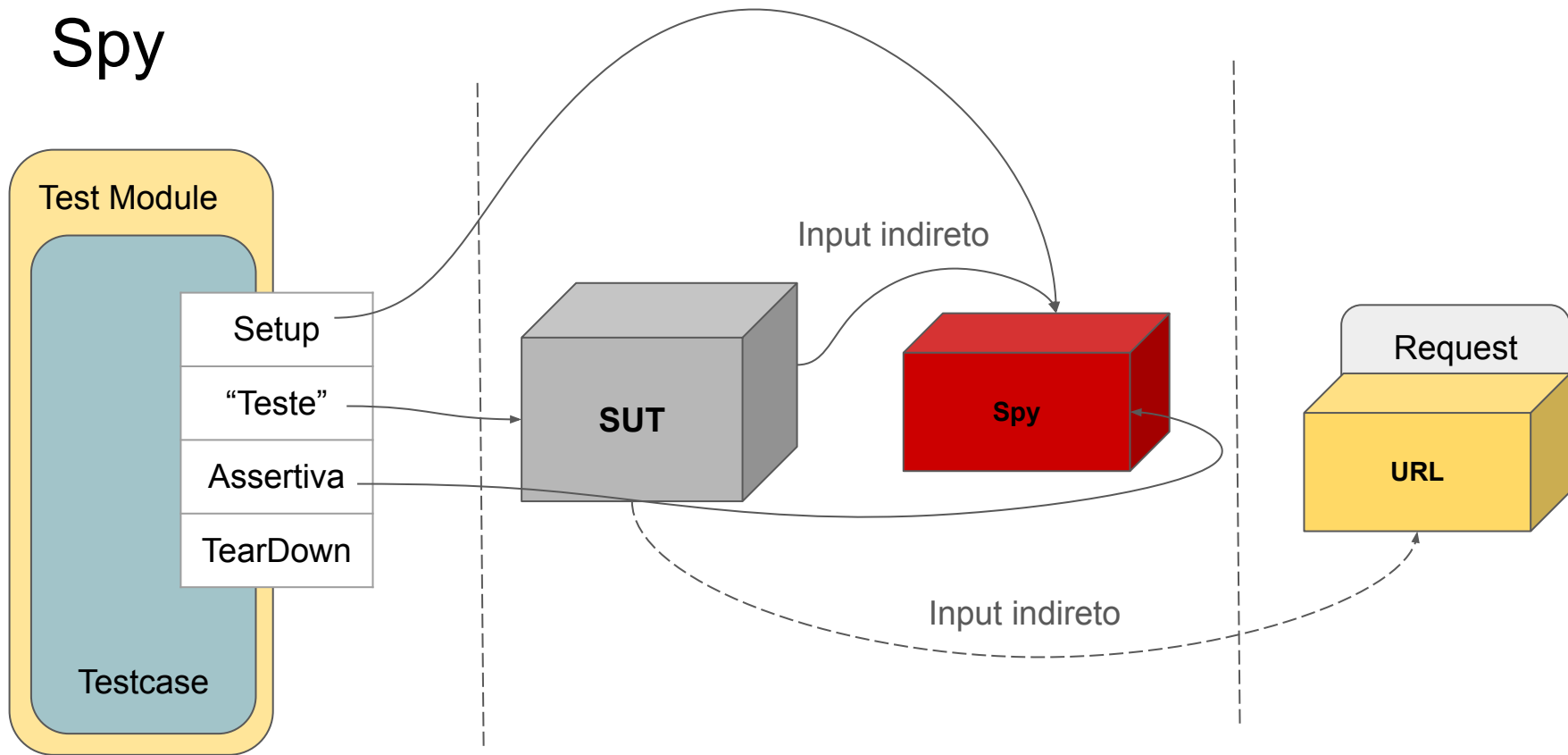
“Procedural Behavior Verification” é o termo técnico

A função deles é exercitar os inputs indiretos e validar se o DOC foi de fato executado

Spy



Spy



Spy

```
from requests import get

def page_content(url: str, ssl: bool = False, *, params=None) -> str:
    prefix = 'https' if ssl else 'http'
    return get(f'{prefix}://{url}', params).content.decode()
```

```
class TestPageContent(TestCase):
    def test_page_content_deve_ser_chamada_com_http(self):

        with mock.patch('acomplamento_exemplo.get') as spy:
            page_content('bababa', False)

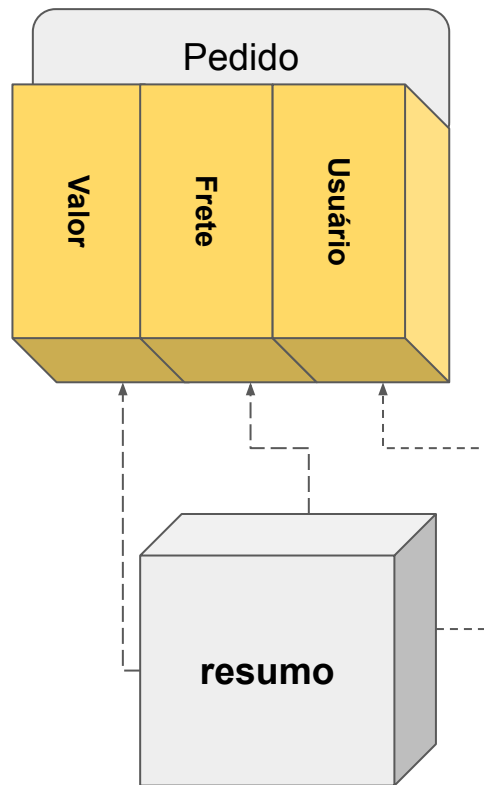
        spy.assert_called_with('http://bababa', None)
```

Problema #11

Usando a função 'exp' criada no problema 8. Faça a checagem dos inputs indiretos da função de soma em um teste. Em outro teste faça a função dos inputs indiretos da função de sub.

```
def exp(x, y, z):  
    return sub(soma(x, y), z)
```

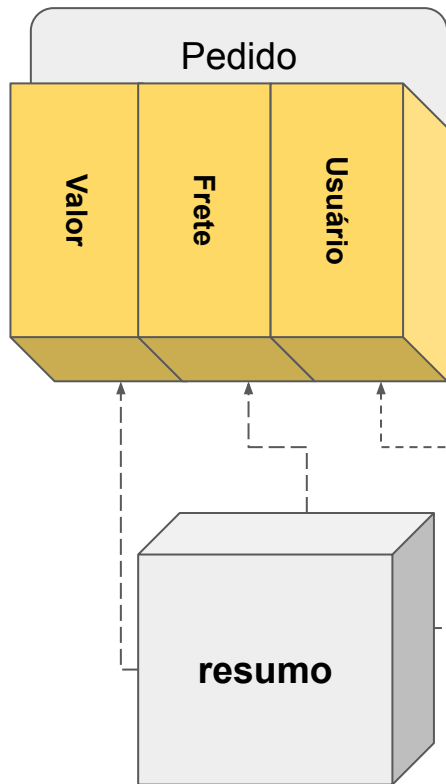

Fake



O fake é usado quando temos um módulo acoplado (DOC) e esse módulo é realmente chamado de alguma maneira.

Então o Fake tem que se comportar como o objeto, com a mesma API, na chamada do SUT.

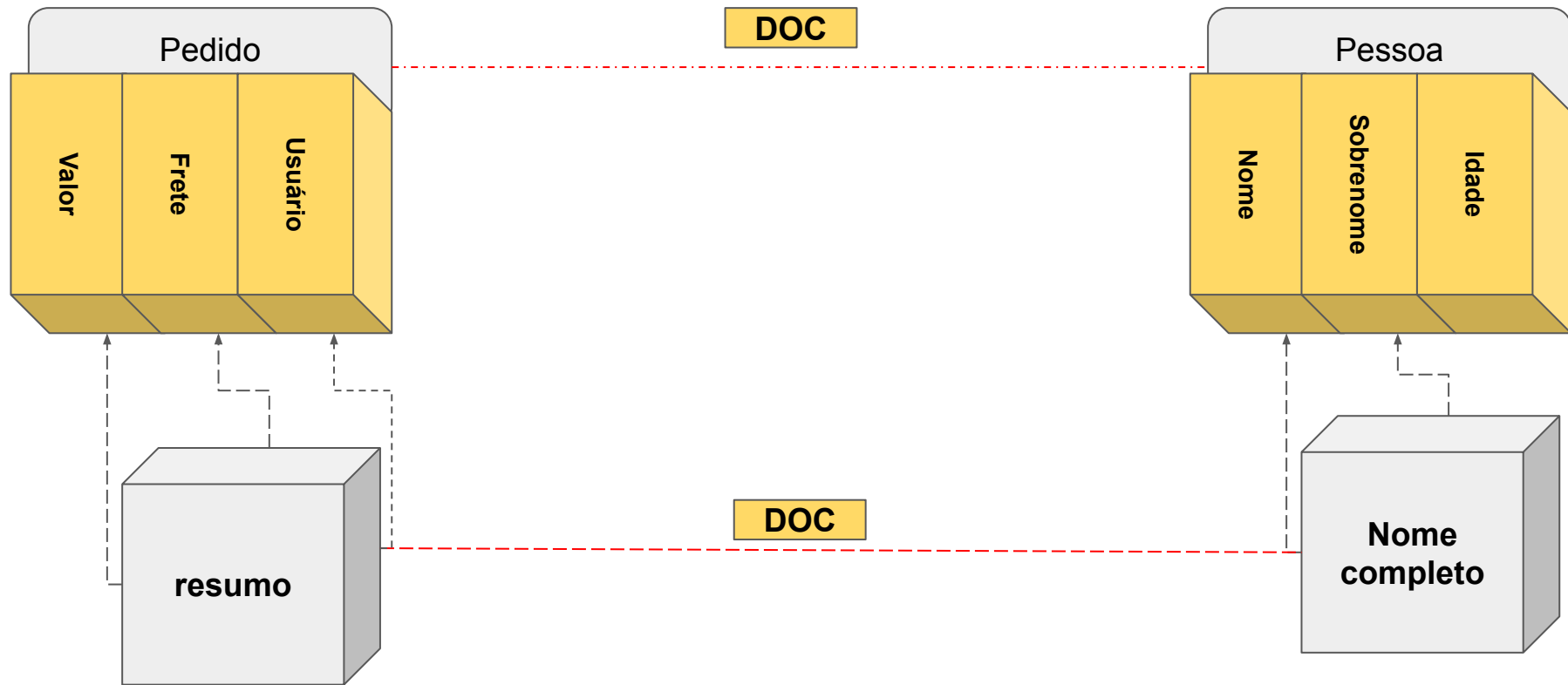
Fake



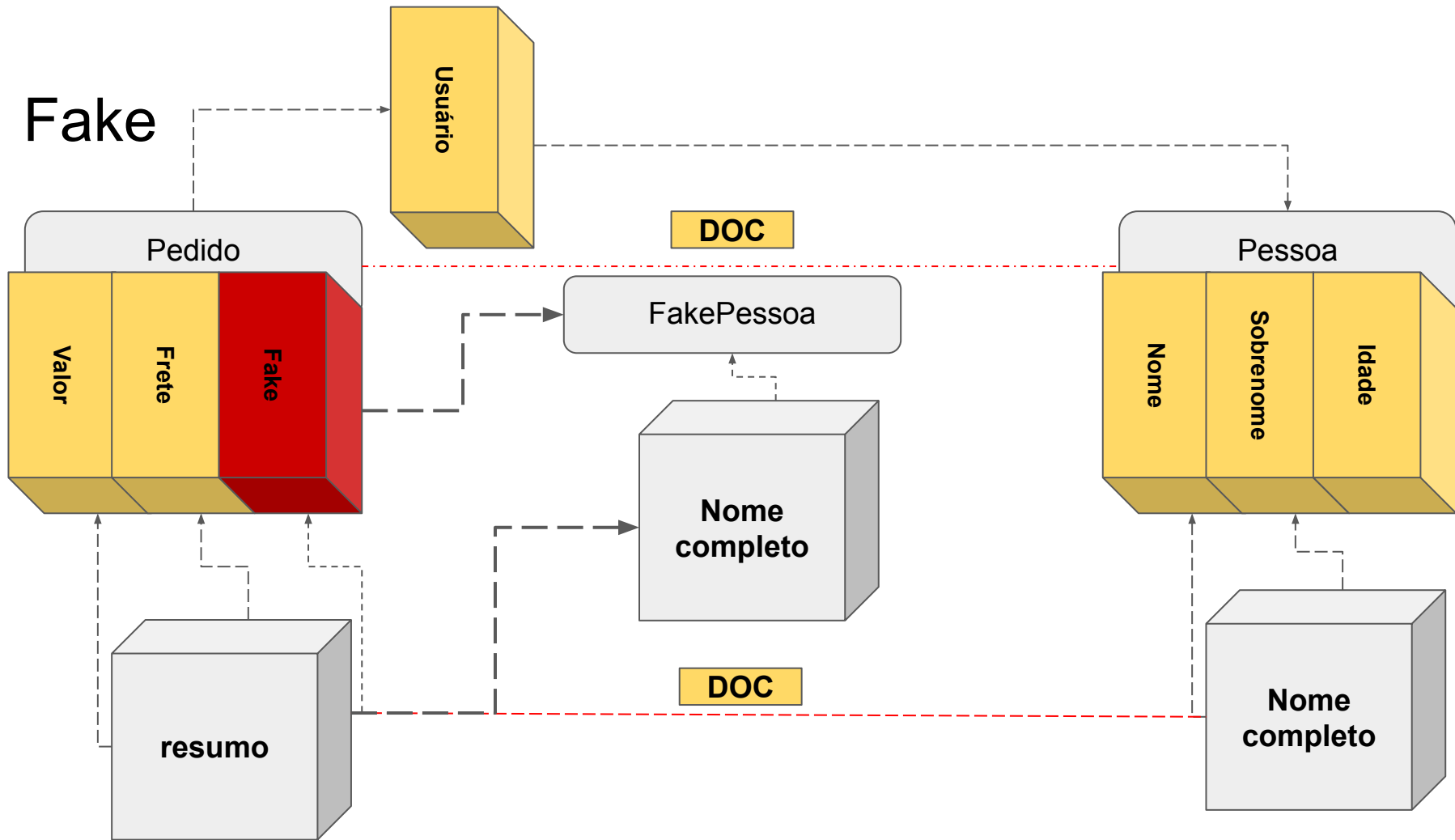
```
class Pedido:
    def __init__(self, valor, frete, usuario):
        self.valor = valor
        self.frete = frete
        self.usuario = usuario

    @property
    def resumo(self):
        """Informações gerais sobre o pedido."""
        return f'''
DOC Pedido por: {self.usuario.nome_completo}
        Valor: {self.valor}
        Frete: {self.frete}
        ...'''
```

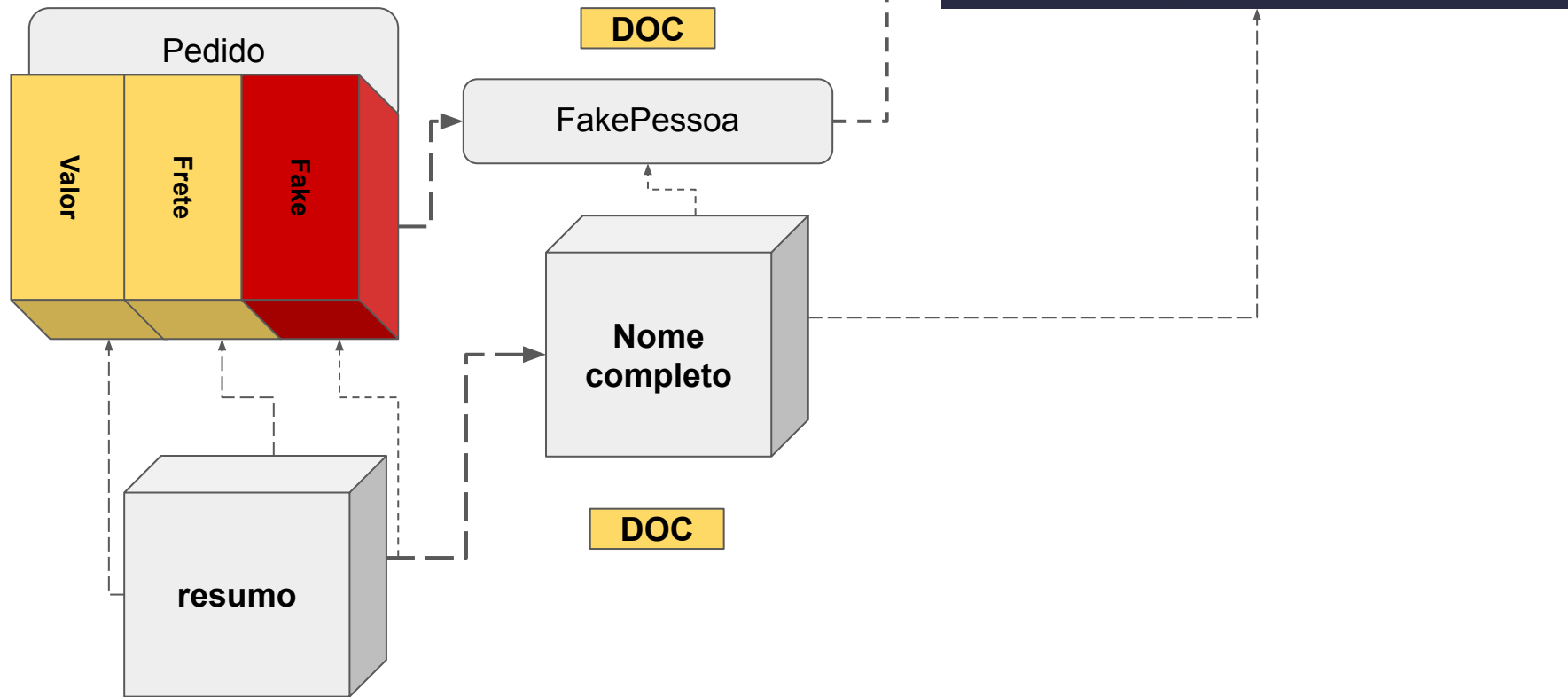
Fake



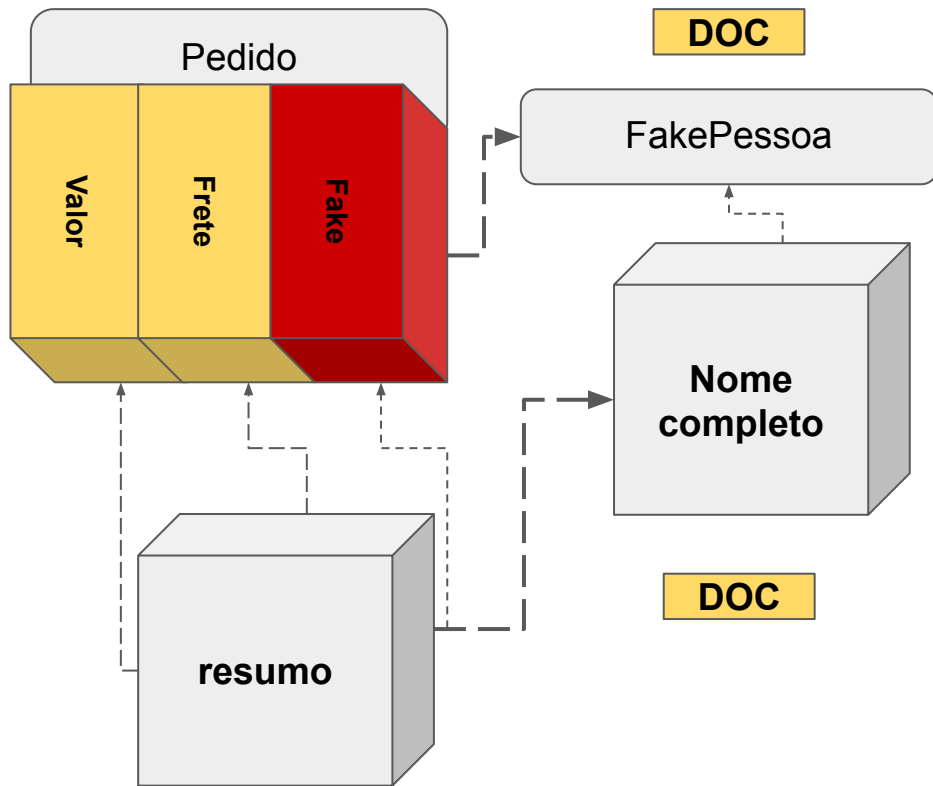
Fake



Fake



Fake



```
class Pedido:
    def __init__(self, valor, frete, usuario):
        self.valor = valor
        self.frete = frete
        self.usuario = usuario

    @property
    def resumo(self):
        """Informações gerais sobre o pedido."""
        return f'''
        Pedido por: {self.usuario.nome_completo}
        Valor: {self.valor}
        Frete: {self.frete}
        '''

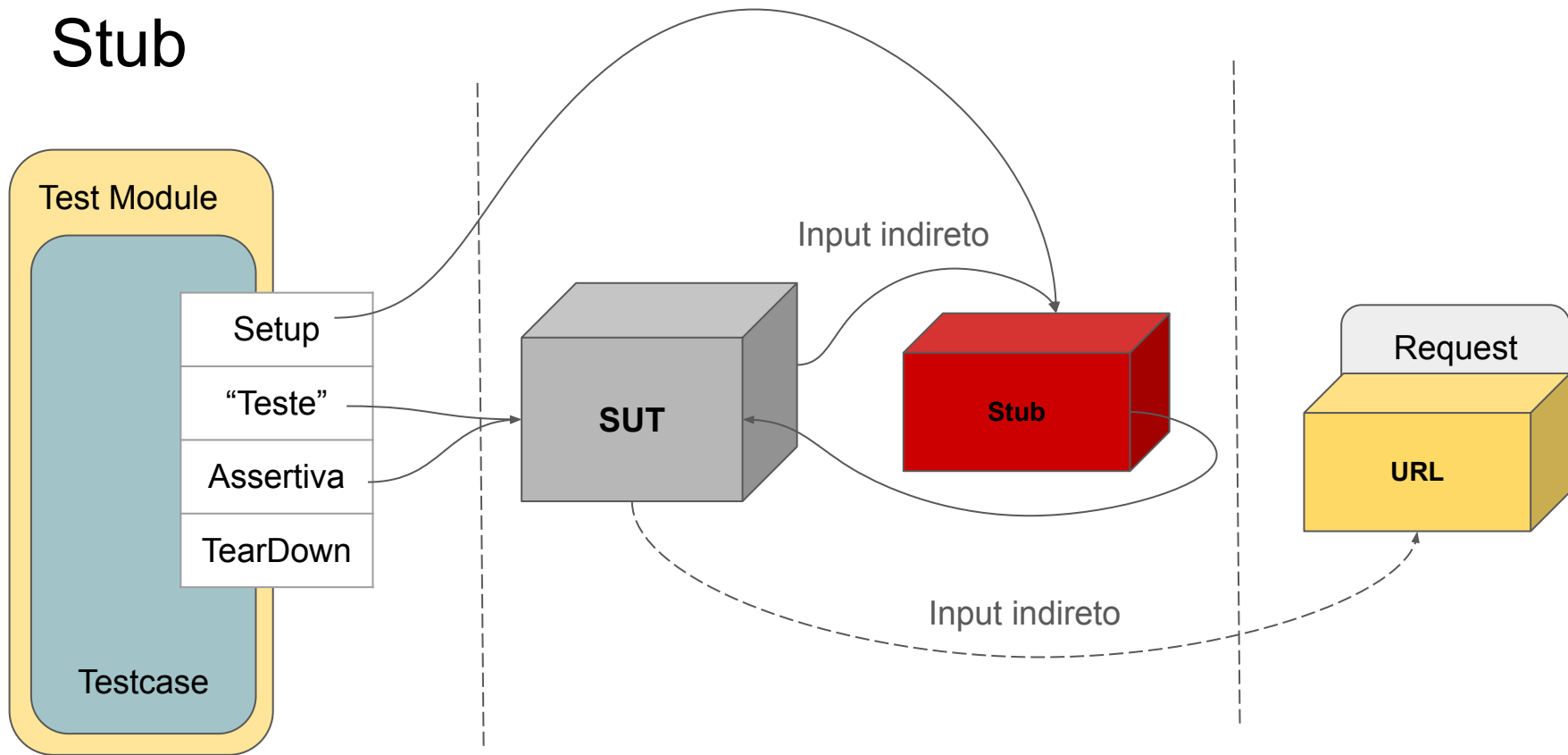
class FakePessoa:
    @property
    def nome_completo(self):
        return 'Eduardo Mendes'

Pedido(100.00, 13.00, FakePessoa()).resumo
```

Stub

Stub é um objeto que armazena dados predefinidos e os utiliza para atender chamadas durante os testes. Ele é usado quando não podemos ou não queremos envolver objetos que respondam com dados reais ou que tenham efeitos colaterais indesejáveis.

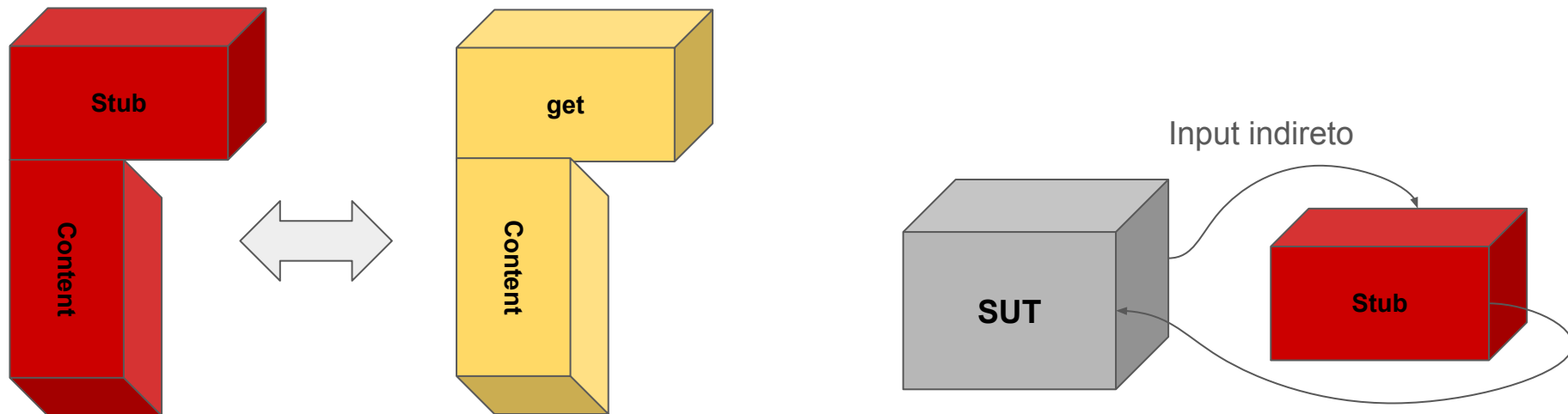
Stub



Stub

```
from requests import get

def page_content(url: str, ssl: bool = False, *, params=None) -> str:
    prefix = 'https' if ssl else 'http'
    return get(f'{prefix}://{url}', params).content.decode()
```

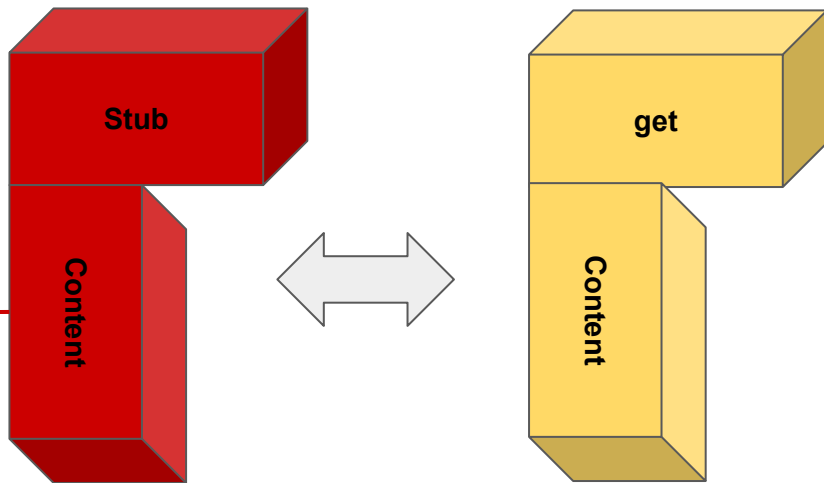


Stub

```
from requests import get

def page_content(url: str, ssl: bool = False, *, params=None) -> str:
    prefix = 'https' if ssl else 'http'
    return get(f'{prefix}://{url}', params).content.decode()
```

```
class StubGet:
    @property
    def content(self):
        return b'<html> ... </html>'
```



Stub

```
class StubGet:
    @property
    def content(self):
        return b'<html> ... </html>'
```

```
from requests import get

def page_content(url: str, ssl: bool = False, *, params=None) -> str:
    prefix = 'https' if ssl else 'http'
    return get(f'{prefix}://{url}', params).content.decode()
```

```
class TestPageContent(TestCase):
    def test_page_content_deve_ser_chamada_com_http(self):
        esperado = '<html> ... </html>'

        with mock.patch('acomplamento_exemplo.get', return_value=StubGet()):
            result = page_content('bababa', False)

        self.assertEqual(esperado, result)
```

Mock

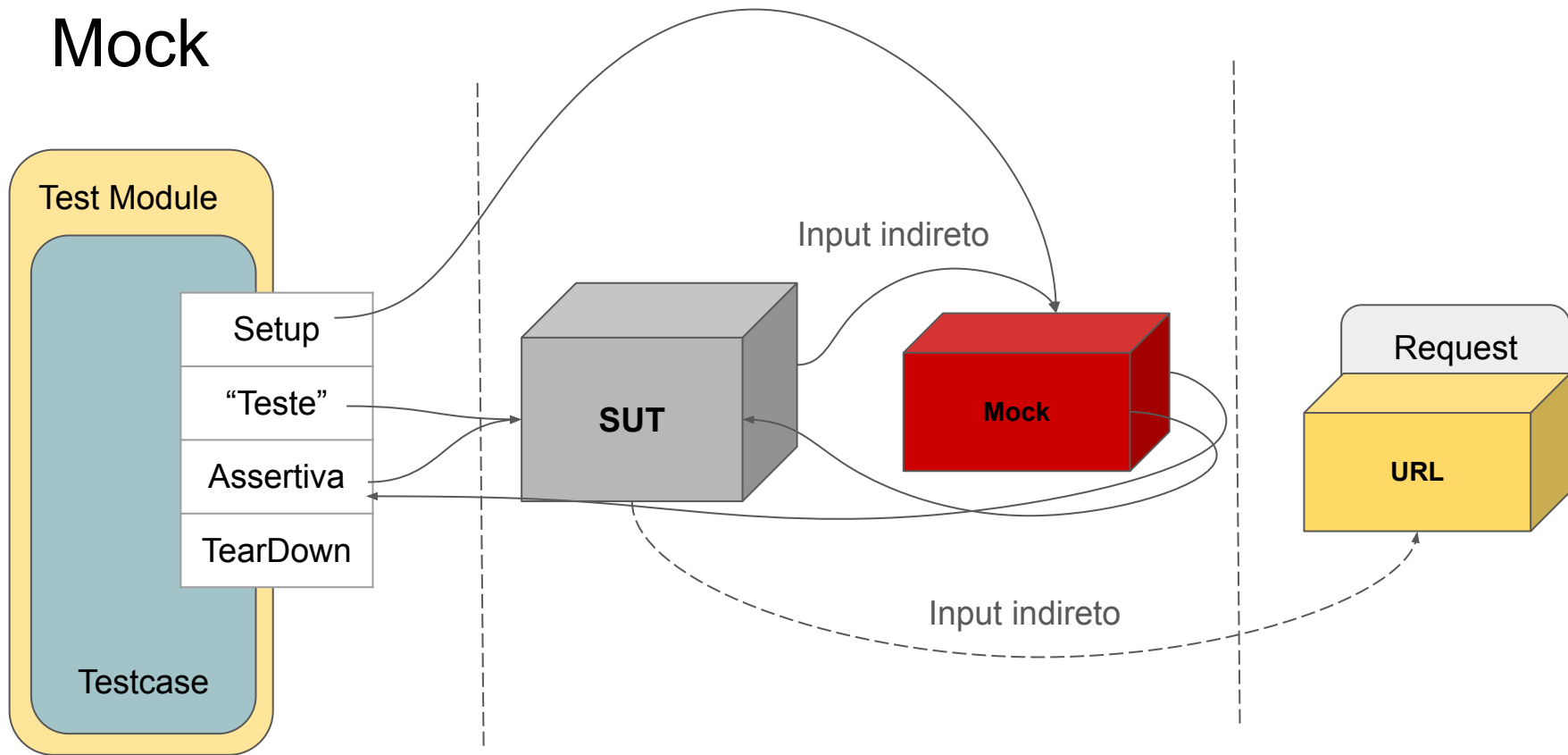
Os mocks são uma loucura que só. Sério, eles são de mais. Ninguém pode dizer o contrário.

Vamos esquecer da teoria por um pequeno minuto.

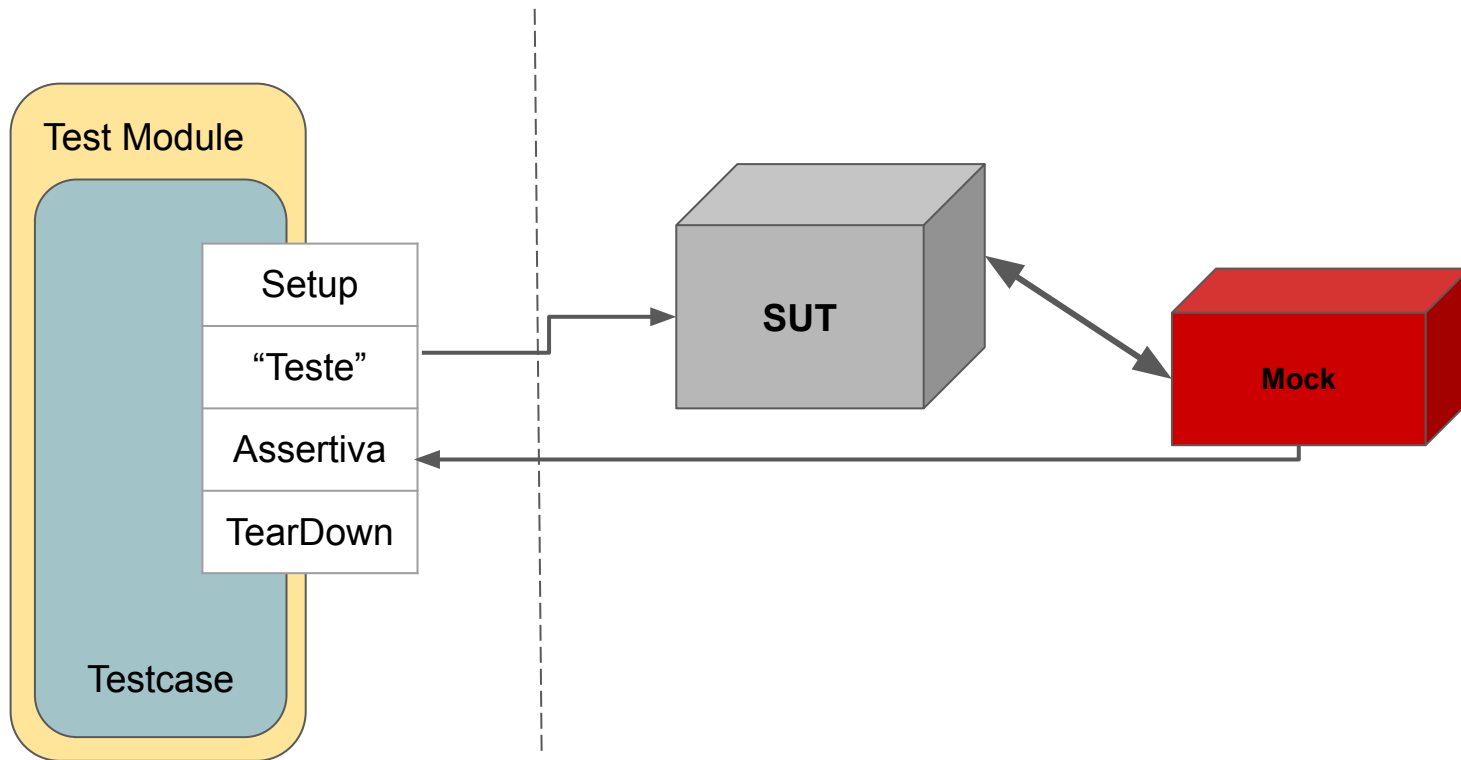
Mocks controlam o fluxo, se colocam no lugar de outras coisas, e ainda pode checar seus resultados.

Ele é um Stub e um Spy ao mesmo tempo.

Mock



Mock



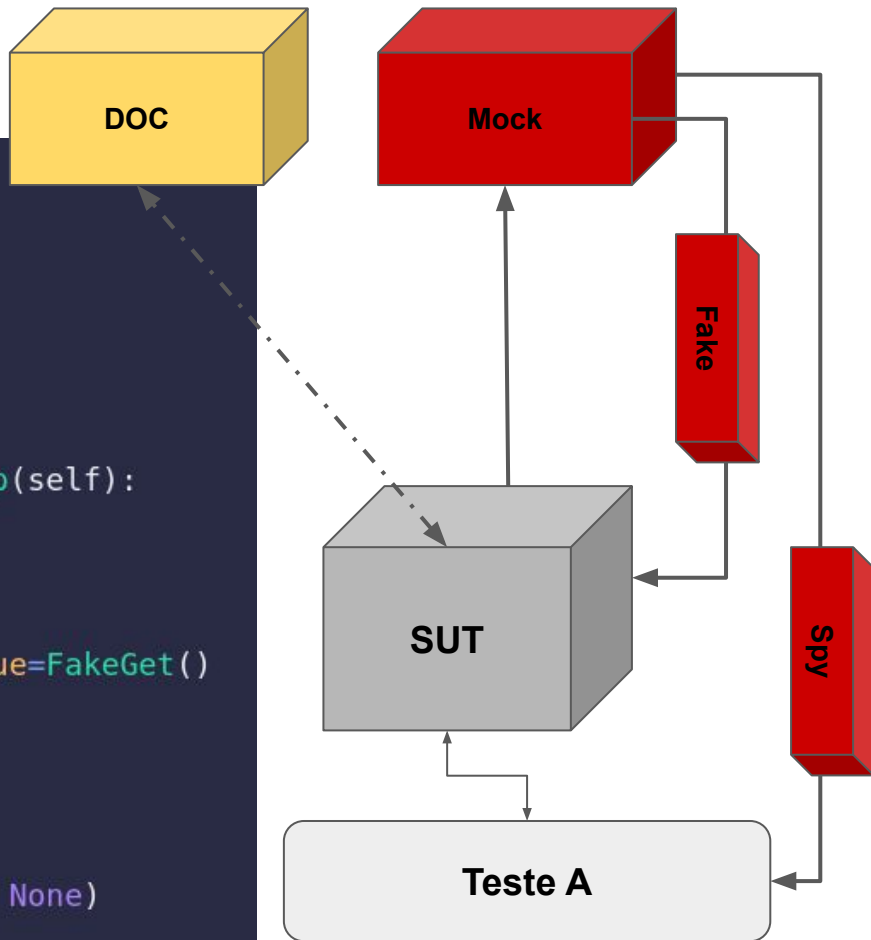
Mock

```
class FakeGet:
    @property
    def content(self):
        return b'<html> ... </html>'

class TestPageContent(TestCase):
    def test_page_content_deve_ser_chamada_com_http(self):
        esperado = '<html> ... </html>'

        with patch(
            'acomplamento_exemplo.get', return_value=FakeGet()
        ) as mocked:
            result = page_content('bababa', False)

        self.assertEqual(esperado, result)
        mocked.assert_called_with('http://bababa', None)
```



Mock

