Transcrição do curso de testes automatizados de Python 3

O curso não texto nos vídeos.

O curso começa com o código abaixo (download):

class Usuario:  
  
 def \_\_init\_\_(self, nome):  
 self.\_\_nome = nome  
  
 @property  
 def nome(self):  
 return self.\_\_nome  
  
  
class Lance:  
  
 def \_\_init\_\_(self, usuario, valor):  
 self.usuario = usuario  
 self.valor = valor  
  
  
class Leilao:  
  
 def \_\_init\_\_(self, descricao):  
 self.descricao = descricao  
 self.\_\_lances = []  
  
 @property  
 def lances(self):  
 return self.\_\_lances

Depois foi criado uma novo arquivo chamado principal.py com o seguinte código:

from src.leilao.dominio import Usuario, Lance, Leilao  
  
# instanciando usuário  
dudu = Usuario('Eduardo')  
fer = Usuario('Fernando')  
amor = Usuario('Aline')  
  
#instanciando lance  
lance\_dudu = Lance(dudu, 100.0)  
lance\_fer = Lance(fer, 150.0)  
lance\_amor = Lance(amor, 200.0)  
  
#instanciando leilão  
leilao = Leilao('Carro')  
  
leilao.lances.append(lance\_dudu)  
leilao.lances.append(lance\_fer)  
leilao.lances.append(lance\_amor)  
  
for lance in leilao.lances:  
 print(f'O usuário {lance.usuario.nome} deu um lance de R${lance.valor}.')

como alteração foi criada uma nova classe chamada Avaliador

import sys

class Avaliador:  
 def \_\_init\_\_(self):  
 # método de conseguir o menor número do sitema  
 self.menor\_lance = sys.float\_info.max  
  
 # método de conseguir o maior número do sitema  
 self.maior\_lance = sys.float\_info.min  
  
 def avalia(self, leilao: Leilao):  
 for lance in leilao.lances:  
  
 if lance.valor > self.maior\_lance:  
 self.maior\_lance = lance.valor  
  
 elif lance.valor < self.menor\_lance:  
 self.menor\_lance = lance.valor

propositalmente ele usa elif no lugar de if, causando erro no principal:

#instanciando leilão  
leilao = Leilao('celular')  
  
leilao.lances.append(lance\_dudu)  
leilao.lances.append(lance\_fer)  
leilao.lances.append(lance\_amor)  
  
for lance in leilao.lances:  
 print(f'O usuário {lance.usuario.nome} deu um lance de R${lance.valor}.')  
  
avaliador = Avaliador()  
avaliador.avalia(leilao)  
  
print(f'O maior lance do leilão foi de {avaliador.maior\_lance} e o menor lance foi de {avaliador.menor\_lance}.')

Vídeo 6 – Implementação de testes unitários

Na IDE PyCharm para montar um teste de uma classe, o atalho para implementar esse teste é Ctrl+shift+t e aparecerá a opção de criar um novo teste, respeitando o padrão de boas práticas da comunidade Python:

from unittest import TestCase  
  
  
class TestAvaliador(TestCase):  
 def test\_avalia(self):  
 self.fail()

para nossa aplicação iremos substituir o fail() pela situação de teste e depois fazer as importações necessárias para a execução da classe

from unittest import TestCase  
from src.leilao.dominio import Usuario, Lance, Leilao, Avaliador  
  
#criado automaticamente com o comando Ctrl+shift+t com o cursor na classe Avaliador  
  
class TestAvaliador(TestCase):  
 def test\_avalia(self):  
 dudu = Usuario('Eduardo')  
 fer = Usuario('Fernando')  
 amor = Usuario('Aline')  
  
 lance\_dudu = Lance(dudu, 100.0)  
 lance\_fer = Lance(fer, 150.0)  
 lance\_amor = Lance(amor, 200.0)  
  
 leilao = Leilao('celular')  
  
 leilao.lances.append(lance\_dudu)  
 leilao.lances.append(lance\_fer)  
 leilao.lances.append(lance\_amor)  
  
 avaliador = Avaliador()  
 avaliador.avalia(leilao)  
  
 valor\_esperado\_maior = 200  
 valor\_esperado\_menor = 100  
  
 self.assertEqual(valor\_esperado\_menor, avaliador.menor\_lance)  
 self.assertEqual(valor\_esperado\_maior, avaliador.maior\_lance)

Executar depois:

Ran 1 test in 0.002s

OK

Process finished with exit code 0

Texto do site:

Um debate muito comum quando falamos sobre testes de software é se escrever testes aumenta a produtividade.

Muitas pessoas falam que quando escrevemos testes de software, garantimos o comportamento daquele trecho de código. Logo, no futuro, não precisaremos nos preocupar em arrumar um bug no sistema. Os testes já garantem, ou pelo menos deveriam, garantir que o código está correto.

Outras pessoas dizem que perdemos muito tempo escrevendo testes e que poderíamos utilizar esse tempo para implementar novas funcionalidades no sistema.

O fato é que ser produtivo é **relativo**! Cada um tem uma conclusão diferente das afirmações acima. O fato é que escrevendo testes ou não, o sistema tem que estar funcional e com o mínimo de erros para não afetar as pessoas e outros sistemas que dependam dele.

Logo, se escrever testes nos deixa mais produtivo ou não, depende! Depende do que enxergamos como produtividade.

Vamos começar a escrever a classe Avaliador. Portando, vamos até o módulo de domínio e começamos e declará-la:

class Avaliador:

Essa classe vai nos informar qual o maior e qual o menor lance do leilão. Vamos inicializá-la com esses dois atributos com o menor e com o maior valor para um número de ponto flutuante no sistema:

class Avaliador:

def \_\_init\_\_(self):

self.maior\_lance = sys.float\_info.min

self.menor\_lance = sys.float\_info.max

Bacana! Agora precisamos criar o método avalia. Esse método percorrerá a lista de lances no leilão e verá se o valor deste lance é maior ou menor do que o valor que está nos atributos:

class Avaliador:

def \_\_init\_\_(self):

self.maior\_lance = sys.float\_info.min

self.menor\_lance = sys.float\_info.max

def avalia(self, leilao):

for lance in leilao.lances:

if lance.valor > self.maior\_lance:

self.maior\_lance = lance.valor

elif lance.valor < self.menor\_lance:

self.menor\_lance = lance.valor

Vamos voltar ao arquivo principal.py e testar essa classe:

from src.leilao.dominio import Leilao, Usuario, Lance, Avaliador

# restante do código omitido

avaliador = Avaliador()

avaliador.avalia(leilao)

print(f'Maior lance: {avaliador.maior\_lance}')

print(f'Menor lance: {avaliador.menor\_lance}')

Quando executamos esse código, obtemos a seguinte saída:

Lance do usuario Gui com o valor de 200.0

Lance do usuario Yuri com o valor de 100.0

Maior lance: 200.0

Menor lance: 100.0

Porém quando trocamos a ordem que os lances são inseridos no sistema obtemos essa saída:

Lance do usuario Yuri com o valor de 100.0

Lance do usuario Gui com o valor de 200.0

Maior lance: 200.0

Menor lance: 1.7976931348623157e+308

Ou seja, temos um bug no sistema. Nosso código tem uma falha! Com isso, vimos o porquê é importante testar um software. Quando testamos o código, pensamos nos possíveis casos de uso que envolvem aquela regra de negócio. Por isso, garantimos que, para aqueles casos de uso testados, o código funciona corretamente.

A IDE Pycharm já possui alguns atalhos para a criação de testes. Para isso, basta colocar o cursor em cima da classe Avaliador e digitar o atalho **Ctrl+Shit+T**. Vamos chamar esse teste de test\_avaliador.py e selecionar o método avalia para ser testado.

O Pycharm já cria um método para a gente com o nome test\_avalia. Esse prefixo test\_\* o Python utiliza para reconhecer que este é um método de testes.

Vamos escrever o teste do avaliador nesse método. Para isso, vamos instanciar os objetos das classes de Leilão, Usuário, Lance e Avaliador.

Para fazer a asserção dos valores podemos utilizar o método assertEqual que herdamos da classe TestCase. Vemos que na primeira vez que rodamos o teste ele falha. Vamos arrumar o código na classe Avaliador. O problema se encontra na parte que fazemos a checagem dos valores no if. Basta trocar o elif por um if que conseguimos resolver esse problema:

def avalia(self, leilao):

for lance in leilao.lances:

if lance.valor > self.maior\_lance:

self.maior\_lance = lance.valor

if lance.valor < self.menor\_lance:

self.menor\_lance = lance.valor

Quando rodamos o teste agora, ele passa. :)

O nome das funções de teste deve ser bem explicado, não tendo problema de ser longo.

Vimos na aula o método setUp que é executado antes de cada método de testes. Este não é o único método que herdamos da classe TestCase, além deste, temos outros métodos como o tearDown, setUpClass e o tearDownClass.

Vamos imaginar que estamos realizando um teste que faz a conexão com um serviço externo, envia um e-mail ou então faz uma modificação no banco de dados.

Nesses casos, precisamos abrir uma conexão, executar os testes e fechar a conexão em seguida. Fechar a conexão é algo em comum com esses tipos de testes.

Podemos então declarar o método tearDown que é executado logo após a execução do teste. Ou seja, caso abrimos uma conexão, podemos utilizar o método tearDown para fechá-la após os testes.

Analogamente ao método setUp e tearDown, temos o setUpClass e o tearDownClasss. Ambos funcionam de uma forma parecida com o setUp e tearDown, porém, ao invés de serem executados antes de cada método, são executados apenas uma vez - o setUpClass no momento que a classe é criada e o tearDownClass após o último teste da classe ser rodado.

Os métodos tearDown e tearDownClass são muito utilizados em testes que integram duas partes do sistema - banco de dados, sistemas externos, ou então desejamos fechar uma conexão que foi aberta.

Esses tipos de testes, que verificam como duas partes do sistema se integram, são chamados de **testes de integração**.

Como vimos na aula, os métodos de testes precisam ser semânticos. Em muitos casos, os testes são utilizados como a documentação do código no projeto.

Existem algumas formas de nomeação, contudo, em muitos casos os times definem qual padrão utilizar. O importante é que o nome dos métodos sejam expressivos e que o padrão escolhido seja seguido.

Caso queiram conhecer outras formas comuns de nomear os testes, neste link <https://dzone.com/articles/7-popular-unit-test-naming> de um artigo em inglês mostra algumas formas de nomeação.

Vamos criar um teste que avalia os lances quando esses são propostos em ordem decrescente. Por isso, vamos começar a criar o método de teste:

test\_avalia2(self):

# implementação omitida

Esse testes, será bem parecido com o primeiro, teremos dois usuários e dois lances. Esses lances serão propostos a um leilão e, ao final, teremos um objeto do tipo Avaliador que usaremos para avaliar o leilão.

Bacana! O teste foi implementado, mas qual o problema com essa classe?

O que acontecerá quando uma pessoa olhar essa classe e ver dois métodos: test\_avalia e test\_avalia2? Ou então, se precisarmos testar outro caso de uso do método avalia, o que faríamos?

Com isso vimos que essa forma de nomeação não é tão boa, já que os nomes dos métodos não são semânticos.

Existem diversas formas para se nomear um método de teste. Uma forma muito utilizada é começar pela asserção, isto é, por aquilo que é esperado no teste, por exemplo: deve retornar o maior e o menor valor de um lance. Seguido pelo comportamento que está sendo testado - quando adicionados em ordem crescente, por exemplo.

Seguindo esse padrão, podemos renomear o test\_avalia para algo como:

def test\_deve\_retornar\_o\_maior\_e\_o\_menor\_valor\_de\_um\_lance\_quando\_adicionados\_em\_ordem\_crescente(self):

# implementação omitida

Lembrando que o teste deve ter o prefixo test\_\* para que a biblioteca unittest a reconheça como um teste.

Apesar de esse ser um estilo de nomeação de testes muito utilizada, essa não é a única forma de se fazer isso. Existem diferentes estilos e padrões que podem ser utilizados, tais padrões são definidos pela equipe. O importante é que ele seja semântico, diga qual a funcionalidade que está sendo testada. Além disso, o padrão precisa ser seguido, para evitar diferentes estilos de nomenclatura no projeto e possíveis confusões.

Vamos criar um novo teste para caso o leilão ter apenas um lance. Como já definimos um padrão de nomeação para o projeto, podemos utilizá-lo. Com isso, teremos um teste como:

def test\_deve\_retornar\_o\_mesmo\_valor\_para\_maior\_e\_menor\_lance\_quando\_leilao\_tiver\_apenas\_um\_lance(self):

# implementação omitida

O próximo passo é criar um teste que verifica o funcionamento quando o leilão possuir três lances. Podemos nomeá-lo com algo como:

def test\_deve\_retornar\_o\_maior\_e\_o\_menor\_valor\_de\_um\_lance\_quando\_o\_leilao\_tiver\_tres\_lances(self):

# código omitido

Não precisamos implementar testes para caso o leilão tiver quatro, cinco ou dez lances. Já temos testes que garantem que esse código funciona nesses casos, ou seja, já temos classes de equivalência para esses testes.

Já temos um código de testes bem estruturado. Qual o problema com ele ainda?

Temos muito código repetido! Muito copia e cola, quase todos os métodos de testes utilizam os mesmos objetos. Código repetido, geralmente, é considerado um mal cheiro (code smells), vamos resolver isso mais como?

**Isolando a criação do cenário**

A maior parte dos testes que escrevemos precisam dos mesmos objetos, para funcionar, isto é, o cenário entre eles é o mesmo. Por isso, podemos isolar a criação desses objetos em um método. Na biblioteca unittest esse método é chamado setUp:

class TestAvaliador(TestCase):

def setUp(self):

self.gui = Usuario('Gui')

self.lance\_do\_gui = Lance(self.gui, 150.0)

self.leilao = Leilao('Celular')

Antes de um teste ser rodado, o método setUp é executado criando os objetos. Isso garante que sempre teremos novos objetos antes de cada execução dos testes. Agora, basta utilizarmos esses objetos nos métodos de teste:

def test\_deve\_retornar\_o\_maior\_e\_o\_menor\_valor\_de\_um\_lance\_quando\_adicionados\_em\_ordem\_crescente(self):

yuri = Usuario('Yuri')

lance\_do\_yuri = Lance(yuri, 100.0)

self.leilao.lances.append(lance\_do\_yuri)

self.leilao.lances.append(self.lance\_do\_gui)

avaliador = Avaliador()

avaliador.avalia(self.leilao)

menor\_valor\_esperado = 100.0

maior\_valor\_esperado = 150.0

self.assertEqual(menor\_valor\_esperado, avaliador.menor\_lance)

self.assertEqual(maior\_valor\_esperado, avaliador.maior\_lance)

Nesta aula, vimos mais alguns conceitos do mundo de testes. Vimos como nomear os testes, entendemos o que são classes de equivalência e como isolar a criação do cenário no método setUp.

Ao final dessa aula, seu código deve estar parecido com este:

from unittest import TestCase

from src.leilao.dominio import Usuario, Lance, Leilao, Avaliador

class TestAvaliador(TestCase):

# test\_quando\_adicionados\_em\_ordem\_crescente\_deve\_retornar\_o\_maior\_e\_o\_menor\_valor\_de\_um\_lance

def setUp(self):

self.gui = Usuario('Gui')

self.lance\_do\_gui = Lance(self.gui, 150.0)

self.leilao = Leilao('Celular')

def test\_deve\_retornar\_o\_maior\_e\_o\_menor\_valor\_de\_um\_lance\_quando\_adicionados\_em\_ordem\_crescente(self):

yuri = Usuario('Yuri')

lance\_do\_yuri = Lance(yuri, 100.0)

self.leilao.lances.append(lance\_do\_yuri)

self.leilao.lances.append(self.lance\_do\_gui)

avaliador = Avaliador()

avaliador.avalia(self.leilao)

menor\_valor\_esperado = 100.0

maior\_valor\_esperado = 150.0

self.assertEqual(menor\_valor\_esperado, avaliador.menor\_lance)

self.assertEqual(maior\_valor\_esperado, avaliador.maior\_lance)

def test\_deve\_retornar\_o\_maior\_e\_o\_menor\_valor\_de\_um\_lance\_quando\_adicionados\_em\_ordem\_decrescente(self):

yuri = Usuario('Yuri')

lance\_do\_yuri = Lance(yuri, 100.0)

self.leilao.lances.append(self.lance\_do\_gui)

self.leilao.lances.append(lance\_do\_yuri)

avaliador = Avaliador()

avaliador.avalia(self.leilao)

menor\_valor\_esperado = 100.0

maior\_valor\_esperado = 150.0

self.assertEqual(menor\_valor\_esperado, avaliador.menor\_lance)

self.assertEqual(maior\_valor\_esperado, avaliador.maior\_lance)

def test\_deve\_retornar\_o\_mesmo\_valor\_para\_o\_maior\_e\_menor\_lance\_quando\_leilao\_tiver\_um\_lance(self):

self.leilao.lances.append(self.lance\_do\_gui)

avaliador = Avaliador()

avaliador.avalia(self.leilao)

self.assertEqual(150.0, avaliador.menor\_lance)

self.assertEqual(150.0, avaliador.maior\_lance)

def test\_deve\_retornar\_o\_maior\_e\_o\_menor\_valor\_quando\_o\_leilao\_tiver\_tres\_lances(self):

yuri = Usuario('Yuri')

lance\_do\_yuri = Lance(yuri, 100.0)

vini = Usuario('Vini')

lance\_do\_vini = Lance(vini, 200.0)

leilao = Leilao('Celular')

leilao.lances.append(lance\_do\_yuri)

leilao.lances.append(self.lance\_do\_gui)

leilao.lances.append(lance\_do\_vini)

avaliador = Avaliador()

avaliador.avalia(leilao)

menor\_valor\_esperado = 100.0

maior\_valor\_esperado = 200.0

self.assertEqual(menor\_valor\_esperado, avaliador.menor\_lance)

self.assertEqual(maior\_valor\_esperado, avaliador.maior\_lance)

# Lei de Demeter, ou o Princípio do menor conhecimento

Vamos analisar a classe Leilao:

class Leilao:

def \_\_init\_\_(self, descricao):

self.descricao = descricao

self.\_\_lances = []

self.maior\_lance = sys.float\_info.min

self.menor\_lance = sys.float\_info.max

# restante do código

Nesta classe, existe uma lista de lances. Ou seja, uma composição com uma lista. O que esperamos que ocorra com ela?

Esperamos que sejam adicionados os lances pertencentes ao leilão. Algo como leilao.lances.append(lance) é uma má prática, já que estamos muito acoplados com a implementação da classe Leilao.

Além de ferir o princípio do **Diga, não pergunte** (o Tell don't ask), estamos ferindo outro princípio chamado **Lei de Demeter**, ou o **Princípio do menor conhecimento** (Principle of least knowledge).

Esse princípio diz que devemos ter o menor conhecimento sobre a implementação da classe. Dessa forma, evitamos o acoplamento entre as classes do sistema.

Claro, o acoplamento sempre existirá. Quando estamos utilizando uma lista, string, dicionário, estamos acoplados com essa classe. Porém, isso é chamado de "acoplamento bom". A chance de uma dessas classes mudarem e afetarem o código são muito pequenas.

Acoplar com classes estáveis é muito melhor do que com classes instáveis

Então só devemos nos acoplar com classes do sistema?

Não! No código que escrevemos existem classes mais estáveis do que outras. São essas que devemos optar por acoplar.

Seguindo esses princípios como a Lei de Demeter e o Diga, não pergunte, escrevemos um código mais simples de ser alterado e escalado.

Quem quiser ver um pouco mais sobre a Lei de Demeter, deixo [esse texto](https://en.wikipedia.org/wiki/Law_of_Demeter), em inglês, da Wikipédia.

# Para saber mais - Copia Profunda

No curso, devolvemos uma cópia da lista de lances quando utilizamos a property:

class Leilao:

# código omitido

@property

def lances(self):

return self.\_\_lances[:]

Foi dito que isso é uma cópia rasa, mas o que isso significa?

Quando utilizamos uma cópia rasa, apenas a referência da lista, neste caso, é diferente. Todos os outros objetos dentro dessa lista compartilham a mesma referência. Ou seja, apesar da lista de lances ser uma cópia, os lances dentro da lista copiada são os mesmos lances do leilão.

Para que os lances sejam diferentes, precisamos copiar a lista profundamente. Por isso os termos cópia rasa e cópia profunda.

Poderia dar exemplos e falar sobre o funcionamento das cópias, porém, o Yan, um dos escritores do [blog da Alura](https://blog.alura.com.br) já escreveu um excelente post sobre cópias de lista no Python. Recomendo muito a leitura desse post. Para quem estiver interessado, pode encontrar a leitura neste link: <http://blog.alura.com.br/como-fazer-copia-de-lista-python/>

(post na pasta com o mesmo nome)

Enquanto escrevíamos os testes, vimos que havia algo de estranho com o design das classes, mas o que está estranho?

Para adicionar um leilão na lista, primeiro pedimos a lista de lances e depois utilizamos o método append da lista:

leilao.lances.append(lances)

Qual o problema com essa abordagem? O que acontece se trocarmos a implementação da classe Leilao? O código provavelmente quebrará.

Ou seja, estamos muito acoplados com esse código da classe Leilao. O que fazer então? Uma forma é isolar a adição de um novo leilão na lista. Podemos fazer isso criando um novo método:

def propoe(self, lance: Lance):

self.\_\_lances.append(lance)

O método propoe fica sendo o responsável por adicionar um lance a lista. Dessa forma, escondemos, isto é, encapsulamos a implementação da classe. Mas ainda conseguimos adicionar um lance sem utilizar o método propoe. Isso acontece porque estamos devolvendo a mesma lista de lances da classe Leilao. Ou seja, precisamos devolver outra lista. Para isso, podemos devolver uma cópia da lista de lances na propriedade:

@property

def lances(self):

return self.\_\_lances[:]

Bacana! Agora nos testes, basta substituir a adição de um leilão na lista pelo método propõe.

Vamos dar uma olhada na classe Avaliador, qual o papel dela? Essa classe trabalha com um leilão e descobre qual o valor do maior e qual o valor do menor lance. Faz sentido este comportamento estar aí? Ou é uma tarefa do Leilao conhecer o maior e o menor lance?

A resposta para essa pergunta é depende! As vezes é preciso separar o estado do comportamento. Alguns design patterns fazem isso. Outras vezes não é tão necessário.

No nosso caso, faz sentido o Leilao conseguir saber quem é o maior e o menor lance. Portanto, vamos mover o comportamento da classe Avaliador para a classe Leilao:

class Leilao:

def \_\_init\_\_(self, descricao):

self.descricao = descricao

self.\_\_lances = []

self.maior\_lance = sys.float\_info.min

self.menor\_lance = sys.float\_info.max

def propoe(self, lance: Lance):

if lance.valor > self.maior\_lance:

self.maior\_lance = lance.valor

if lance.valor < self.menor\_lance:

self.menor\_lance = lance.valor

self.\_\_lances.append(lance)

@property

def lances(self):

return self.\_\_lances[:]

Após isso, basta removermos o Avaliador dos testes e apagarmos esta classe.

Nesta aula, entramos em um debate sobre boas práticas de código e orientação a objetos. Ao final, teremos o módulo de domínio parecido com este:

import sys

class Usuario:

def \_\_init\_\_(self, nome):

self.\_\_nome = nome

@property

def nome(self):

return self.\_\_nome

class Lance:

def \_\_init\_\_(self, usuario, valor):

self.usuario = usuario

self.valor = valor

class Leilao:

def \_\_init\_\_(self, descricao):

self.descricao = descricao

self.\_\_lances = []

self.maior\_lance = sys.float\_info.min

self.menor\_lance = sys.float\_info.max

def propoe(self, lance: Lance):

if lance.valor > self.maior\_lance:

self.maior\_lance = lance.valor

if lance.valor < self.menor\_lance:

self.menor\_lance = lance.valor

self.\_\_lances.append(lance)

@property

def lances(self):

return self.\_\_lances[:]

Na próxima aula, vamos adicionar mais regras de negócio e ver como testar exceções.

Até agora os códigos estão assim:

Domínio:

import sys  
  
class Usuario:  
 def \_\_init\_\_(self, nome):  
 self.\_\_nome = nome  
  
 @property  
 def nome(self):  
 return self.\_\_nome  
  
  
class Lance:  
 def \_\_init\_\_(self, usuario, valor):  
 self.usuario = usuario  
 self.valor = valor  
  
  
class Leilao:  
 def \_\_init\_\_(self, descricao):  
 self.descricao = descricao  
 self.\_\_lances = []  
 self.menor\_lance = sys.float\_info.max  
 self.maior\_lance = sys.float\_info.min  
  
 def propoe(self, lance: Lance):  
 if lance.valor > self.maior\_lance:  
 self.maior\_lance = lance.valor  
 if lance.valor < self.menor\_lance:  
 self.menor\_lance = lance.valor  
  
 self.\_\_lances.append(lance)  
  
 @property  
 def lances(self):  
 return self.\_\_lances[:] #devolve uma cópia da lista. Não é a original

test\_avaliador:

from unittest import TestCase  
from src.leilao.dominio import Usuario, Lance, Leilao  
  
#criado automaticamente com o comando Ctrl+shift+t com o cursor na classe Avaliador  
  
class TestAvaliador(TestCase):  
  
 def setUp(self): # criação de um cenário único para os teste rodarem  
 self.dudu = Usuario('Eduardo')  
 self.fer = Usuario('Fernando')  
 self.amor = Usuario('Aline')  
  
 self.lance\_dudu = Lance(self.dudu, 200.0)  
 self.lance\_fer = Lance(self.fer, 150.0)  
 self.lance\_amor = Lance(self.amor, 100.0)  
  
 self.leilao = Leilao('celular')  
  
 def test\_deve\_retornar\_o\_maior\_e\_o\_menor\_valor\_de\_um\_lance\_quando\_adicionados\_em\_ordem\_crescente(self):  
  
 self.leilao.propoe(self.lance\_dudu)  
 self.leilao.propoe(self.lance\_fer)  
 self.leilao.propoe(self.lance\_amor)  
  
 valor\_esperado\_maior = 200  
 valor\_esperado\_menor = 100  
  
 self.assertEqual(valor\_esperado\_menor, self.leilao.menor\_lance)  
 self.assertEqual(valor\_esperado\_maior, self.leilao.maior\_lance)  
  
 def test\_deve\_retornar\_o\_mesmo\_valor\_para\_maior\_e\_menor\_lance\_quando\_leilao\_tiver\_apenas\_um\_lance(self):  
  
 self.leilao.propoe(self.lance\_dudu)  
  
 valor\_esperado\_maior = 200  
 valor\_esperado\_menor = 200  
  
 self.assertEqual(valor\_esperado\_menor, self.leilao.menor\_lance)  
 self.assertEqual(valor\_esperado\_maior, self.leilao.maior\_lance)  
  
  
 def test\_deve\_retornar\_o\_maior\_e\_o\_menor\_valor\_de\_um\_lance\_quando\_adicionados\_em\_ordem\_decrescente(self):  
  
 self.leilao.propoe(self.lance\_dudu)  
 self.leilao.propoe(self.lance\_fer)  
 self.leilao.propoe(self.lance\_amor)  
  
 valor\_esperado\_maior = 200  
 valor\_esperado\_menor = 100  
  
 self.assertEqual(valor\_esperado\_menor, self.leilao.menor\_lance)  
 self.assertEqual(valor\_esperado\_maior, self.leilao.maior\_lance)

# É preciso fazer baby steps?

Vimos uma abordagem de escrita de testes, os baby steps, ou passos de bebê. Nessa abordagem, começamos primeiro pelos testes e fazemos o mínimo possível para o teste passar. Quando ele passa, refatoramos o teste e o código e vamos mantendo esse ciclo no desenvolvimento.

Apesar dos baby steps ajudarem na escrita do código de testes e no entendimento das regras de negócio, não precisamos fazê-lo sempre.

Quando já temos conhecimento claro das regras de negócio ou então quando já conhecemos o problema, não precisamos realizar baby steps.

Existe um debate na comunidade de como praticar o TDD realizando os baby steps e escrevendo primeiro os testes ajudam a escrever um código mais simples, com um melhor algoritmo. Porém, isso também é relativo.

A verdade é que escrever o código é algo muito pessoal. Existem pessoas que começam pelo código e pessoas que começam pelos testes. Existem pessoas que fazem baby steps, outras que não. O importante é o código estar funcionando sem erros quando for solicitado.

Temos que implementar duas novas regras de negócio. O mesmo usuário não pode dar dois lances seguidos e o próximo lance sempre deve ser maior que o anterior.

Vamos começar por uma abordagem diferente, por isso, começamos pelos testes. Vamos começar implementando um teste para um caso de uso que é quando o leilão não possui lances:

def test\_deve\_permitir\_propor\_um\_lance\_caso\_o\_leilao\_nao\_tenha\_lances(self):

# código omitido

Ao rodarmos esse teste, ele passa. Vamos agora a outro caso de uso. Deve permitir um lance quando o usuário é diferente do anterior:

def test\_deve\_permitir\_propor\_um\_lance\_caso\_o\_ultimo\_usuario\_seja\_diferente(self):

# implementação omitida

Esse teste também já passa com o código que implementamos. Vamos para o terceiro teste. Não deve permitir que o mesmo usuário proponha dois lances seguidos:

def test\_nao\_deve\_permitir\_propor\_lance\_caso\_o\_usuario\_seja\_o\_mesmo(self):

# implementação omitida

Dessa vez o teste falhou! O que temos que fazer? Vamos para a classe de domínio e adicionar a verificação sobre se é o mesmo usuário:

def propoe(self, lance: Lance):

if not self.lances or self.lances[-1].usuario != lance.usuario:

if lance.valor > self.maior\_lance:

self.maior\_lance = lance.valor

if lance.valor < self.menor\_lance:

self.menor\_lance = lance.valor

self.\_\_lances.append(lance)

Além de conferir se o último usuário é diferente, precisamos conferir se o leilão não está vazio. Senão é lançado uma exceção.

Bacana! Agora o teste está passando, mas qual o problema dessa abordagem? Não estamos passando um feedback para o usuário. Precisamos que ele saiba que deu um problema na transação, por isso, uma forma é laçar uma exceção:

def propoe(self, lance: Lance):

if not self.\_\_lances or self.\_\_lances[-1].usuario != lance.usuario:

if lance.valor > self.maior\_lance:

self.maior\_lance = lance.valor

if lance.valor < self.menor\_lance:

self.menor\_lance = lance.valor

self.\_\_lances.append(lance)

else:

raise ValueError('Erro ao propor lance')

Legal! Só precisamos alterar o código de testes para testar o lançamento da exceção. Ao herdar da classe TestCase, já existe um método que podemos utilizar para testar exceções. Esse método é o self.assertRaises:

def test\_nao\_deve\_permitir\_propor\_lance\_caso\_o\_usuario\_seja\_o\_mesmo(self):

lance\_do\_gui200 = Lance(self.gui, 200.0)

with self.assertRaises(ValueError):

self.leilao.propoe(self.lance\_do\_gui)

self.leilao.propoe(lance\_do\_gui200)

Todos os testes voltam a passar! Agora vamos implementar a outra regra de negócio. Um lance deve ser maior que o anterior. Como essa regra é um pouco mais simples, já sabemos como implementá-la, não vamos começar pelos testes, partiremos para a implementação.

Para essa regra funcionar, basta adicionarmos uma condição na classe de domínio:

def propoe(self, lance: Lance):

if not self.\_\_lances or self.\_\_lances[-1].usuario != lance.usuario and lance.valor > self.\_\_lances[-1].valor:

# implementação omitida

Com essa alteração, um dos testes falha! O teste que verifica a inserção em ordem decrescente não passa. Faz sentido a gente ainda ter esse teste no sistema? A regra de negócio mudou, ou seja, não tem mais porque manter esse deste dessa maneira. O que podemos fazer é refatorar esse teste para que ele atenda a nova regra de negócio:

def test\_nao\_deve\_permitir\_propor\_um\_lance\_em\_ordem\_decrescente(self):

with self.assertRaises(ValueError):

yuri = Usuario('Yuri')

lance\_do\_yuri = Lance(yuri, 100.0)

self.leilao.propoe(self.lance\_do\_gui)

self.leilao.propoe(lance\_do\_yuri)

Agora todos os nossos testes estão passando!

Para instalar a biblioteca “pytest”:

No Windows -> no cmd digitar: pip install pytest

# Pytest fixtures e classes de testes

No vídeo, vimos um pouco sobre as fixtures do pytest. Existem outras fixtures que podemos utilizar além daquelas. Neste post <https://blog.alura.com.br/montando-cenarios-de-testes-com-o-pytest> são mostrados algumas outras fixtures.

É interessante notar que utilizamos funções para escrever os testes. Porém, se quiséssemos, poderíamos utilizar classes de testes também. A implementação do mesmo módulo de testes como uma classe, ficaria parecida com essa:

from src.leilao.dominio import Usuario, Leilao

import pytest

from src.leilao.excecoes import LanceInvalido

class TestUsuario:

@pytest.fixture

def vini(self):

return Usuario('Vini', 100.0)

@pytest.fixture

def leilao(self):

return Leilao('Celular')

def test\_deve\_subtrair\_valor\_da\_carteira\_do\_usuario\_quando\_este\_propor\_um\_lance(self, vini, leilao):

vini.propoe\_lance(leilao, 50.0)

assert vini.carteira == 50.0

def test\_deve\_permitir\_propor\_lance\_quando\_o\_valor\_eh\_menor\_que\_o\_valor\_da\_carteira(self, vini, leilao):

vini.propoe\_lance(leilao, 1.0)

assert vini.carteira == 99.0

def test\_deve\_permitir\_propor\_lance\_quando\_o\_valor\_eh\_igual\_ao\_valor\_da\_carteira(self, vini, leilao):

vini.propoe\_lance(leilao, 100.0)

assert vini.carteira == 0.0

def test\_nao\_deve\_permitir\_propor\_lance\_com\_valor\_maior\_que\_o\_da\_carteira(self, vini, leilao):

with pytest.raises(LanceInvalido):

vini.propoe\_lance(leilao, 200.0)

Mãos à obra

Antes de começar a escrever os testes, precisamos instalar a biblioteca. Por isso, vamos ao terminal e pedimos para o pip instalar a pytest para a gente:

pip install pytest

Com a biblioteca instalada, vamos começar a escrever nossos arquivos de testes. Mas antes, vamos criar um diretório chamado tests e mover o test\_leilao.py para lá. Nesse diretório, criamos outro arquivo chamado test\_usuario.py. É neste arquivo que vamos colocar o código de testes da classe usuário.

Temos que implementar as novas regras de negócio, mas antes disso, vamos começar pelos testes já para ir conhecendo a biblioteca. Logo, criamos uma função para testar a funcionalidade de subtrair um valor da carteira do usuário:

def test\_deve\_subtrair\_valor\_da\_carteira\_do\_usuario\_quando\_este\_propor\_um\_lance():

# implementação omitida

O teste falha! Vamos começar a mexer na classe usuário, antes de tudo, vamos colocar um parâmetro que representará a carteira no método que inicializa os atributos da classe, além disso, vamos criar um método propoe\_lance, que recebe o leilão e o valor do lance que está sendo proposto:

class Usuario:

def \_\_init\_\_(self, nome, carteira):

self.\_\_nome = nome

self.\_\_carteira = carteira

def propoe\_lance(self, leilao, valor: float):

self.carteira -= valor

lance = Lance(self, valor)

leilao.propoe(lance)

O teste está passando. Vamos começar a implementar outros testes:

def test\_deve\_permitir\_propor\_lance\_quando\_o\_valor\_eh\_menor\_que\_o\_valor\_da\_carteira():

# implementação omitida

def test\_deve\_permitir\_propor\_lance\_quando\_o\_valor\_eh\_igual\_ao\_valor\_da\_carteira():

# implementação omitida

Bacana! Com o código atual, esses dois testes já passam. Vamos implementar um novo testes, dessa vez, o usuário não pode dar um lance maior do que o valor da carteira. Para esse teste, podemos esperar uma exceção:

def test\_nao\_deve\_permitir\_propor\_lance\_com\_valor\_maior\_que\_o\_da\_carteira():

with pytest.raises(ValueError):

# implementação omitida

Como esse teste não passa, vamos implementar na classe de domínio as regras para fazê-lo passar:

def propoe\_lance(self, leilao, valor: float):

if self.carteira >= valor:

self.carteira -= valor

lance = Lance(self, valor)

leilao.propoe(lance)

else:

raise ValueError('Nao pode dar lance maior que o valor da carteira')

Legal, todos os testes estão passando! Mas, temos alguns códigos repetidos nos métodos de teste. Sabemos que podemos isolar isso de alguma forma. Na biblioteca unittest nós utilizamos o método setUp herdado da classe TestCase. O que fazemos na biblioteca pytest já que não herdamos de nenhum lugar?

Na biblioteca pytest, basta criarmos uma função e decorar ela dizendo que ela é uma fixture;

@pytest.fixture

def vini():

return Usuario('Vini', 100.0)

@pytest.fixture

def leilao():

return Leilao('Celular')

Bacana! Agora, basta passar uma parâmetro no método de testes com o mesmo nome da função:

def test\_deve\_subtrair\_valor\_da\_carteira\_do\_usuario\_quando\_este\_propor\_um\_lance(vini, leilao):

# implementação omitida

O Código do arquivo “test\_usuario.py” ficará até agora assim:

Lembrando que é necessário configurar o pycharm para usar o ‘pytest’ no lugar do ‘unittest’.

import pytest  
from src.leilao.dominio import Usuario, Leilao  
  
  
@pytest.fixture  
def usuario():  
 return Usuario('Dudu', 100.0)  
  
  
@pytest.fixture  
def leilao():  
 return Leilao('Celular')  
  
  
def test\_deve\_subtrair\_valor\_da\_carteira\_do\_usuario\_quando\_este\_propor\_um\_lance(usuario, leilao):  
 usuario.propoe\_lance(leilao, 50.0)  
 assert usuario.carteira == 50.0  
  
  
def test\_deve\_permitir\_propor\_lance\_quando\_o\_valor\_eh\_menor\_que\_o\_valor\_da\_carteira(usuario, leilao):  
 usuario.propoe\_lance(leilao, 1.0)  
 assert usuario.carteira == 99.0  
  
  
def test\_deve\_permitir\_propor\_lance\_quando\_o\_valor\_eh\_igual\_ao\_valor\_da\_carteira(usuario, leilao):  
 usuario.propoe\_lance(leilao, 100.0)  
 assert usuario.carteira == 0.0  
  
  
def test\_nao\_deve\_permitir\_propor\_lance\_com\_valor\_maior\_que\_o\_da\_carteira(usuario, leilao):  
 with pytest.raises(ValueError):  
 usuario.propoe\_lance(leilao, 200.0)

O arquivo “domínio.py”

import sys  
  
  
class Usuario:  
 def \_\_init\_\_(self, nome, valor\_carteira):  
 self.\_\_nome = nome  
 self.\_\_carteira = valor\_carteira  
  
 @property  
 def nome(self):  
 return self.\_\_nome  
  
 @property  
 def carteira(self):  
 return self.\_\_carteira  
  
 def propoe\_lance(self, leilao, valor\_lance):  
 if valor\_lance > self.\_\_carteira:  
 raise ValueError('Não pode propor um lance maior que o valor de carteira')  
  
 lance = Lance(self, valor\_lance)  
 leilao.propoe(lance)  
  
 self.\_\_carteira -= valor\_lance  
  
  
class Lance:  
 def \_\_init\_\_(self, usuario, valor):  
 self.usuario = usuario  
 self.valor = valor  
  
  
class Leilao:  
 def \_\_init\_\_(self, descricao):  
 self.descricao = descricao  
 self.\_\_lances = []  
 self.menor\_lance = sys.float\_info.max  
 self.maior\_lance = sys.float\_info.min  
  
 def propoe(self, lance: Lance):  
 if not self.\_\_lances or self.\_\_lances[-1].usuario != lance.usuario and lance.valor > self.\_\_lances[-1].valor:  
 if lance.valor > self.maior\_lance:  
 self.maior\_lance = lance.valor  
 if lance.valor < self.menor\_lance:  
 self.menor\_lance = lance.valor  
  
 self.\_\_lances.append(lance)  
 else:  
 raise ValueError('Erro ao propor lance')  
  
 @property  
 def lances(self):  
 return self.\_\_lances[:] #devolve uma cópia da lista. Não é a original

# Refatorando o domínio

Vamos começar a refatorar o código. Antes de tudo, vamos criar uma exceção que representa um lance inválido. Por isso, vamos criar um novo arquivo do Python chamado de excecoes.py e nele colocamos o código da classe da nossa exceção:

class LanceInvalido(Exception):

pass

Agora basta substituir o lançamento das exceções pela exceção que acabamos de criar:

raise LanceInvalido()

E temos que alterar o código de testes para que ele espere essa exceção:

def test\_nao\_deve\_permitir\_propor\_lance\_com\_valor\_maior\_que\_o\_da\_carteira(vini, leilao):

with pytest.raises(LanceInvalido):

# código omitido

def test\_nao\_deve\_permitir\_propor\_um\_lance\_em\_ordem\_decrescente(self):

with self.assertRaises(LanceInvalido):

# Código omitido

Bacana! Vamos agora começar a isolar as funções. Podemos criar um método \_lance\_eh\_valido e dentro desse método, temos as condições que avaliam um lance:

class Leilao:

# código omitido

def propoe(self, lance: Lance):

if self.\_lance\_eh\_valido(lance):

if not self.\_tem\_lances():

self.menor\_lance = lance.valor

self.maior\_lance = lance.valor

self.\_\_lances.append(lance)

def \_lance\_eh\_valido(self, lance):

return not self.\_tem\_lances() or (self.\_usuarios\_diferentes(lance) and

self.\_valor\_maior\_que\_lance\_anterior(lance))

Esse método tem a função de encapsular os métodos que definem as regras de um lance ser válido ou não. A codificação desses métodos fica próximo a isso:

def \_tem\_lances(self):

return self.\_\_lances

def \_usuarios\_diferentes(self, lance):

if self.\_\_lances[-1].usuario != lance.usuario:

return True

raise LanceInvalido('O mesmo usuário não pode dar dois lances seguidos')

def \_valor\_maior\_que\_lance\_anterior(self, lance):

if lance.valor > self.\_\_lances[-1].valor:

return True

raise LanceInvalido('O valor do lance deve ser maior que o lance anterior')

É importante notar também, que refatoramos o código que pegam o valor do maior e do menor lance, como a regra de negócio mudou, o código também teve que ser modificado.

Além da classe leilão, a classe usuário também pode ser refatorada:

def propoe\_lance(self, leilao, valor):

if not self.\_valor\_eh\_valido(valor):

raise LanceInvalido('Não pode propor um lance com o valor maior que o valor da carteira')

lance = Lance(self, valor)

leilao.propoe(lance)

self.\_\_carteira -= valor

Criamos um método para isolar a condição que checa se um valor é válido ou não. O código do método fica dessa forma:

def \_valor\_eh\_valido(self, valor):

return valor <= self.\_\_carteira

Conseguimos refatorar bem as classes de domínio. Com essa refatoração, conseguimos deixar o código mais legível, o que ajuda em manutenções futuras no sistema, rodando os testes, todos devem continuar passando.