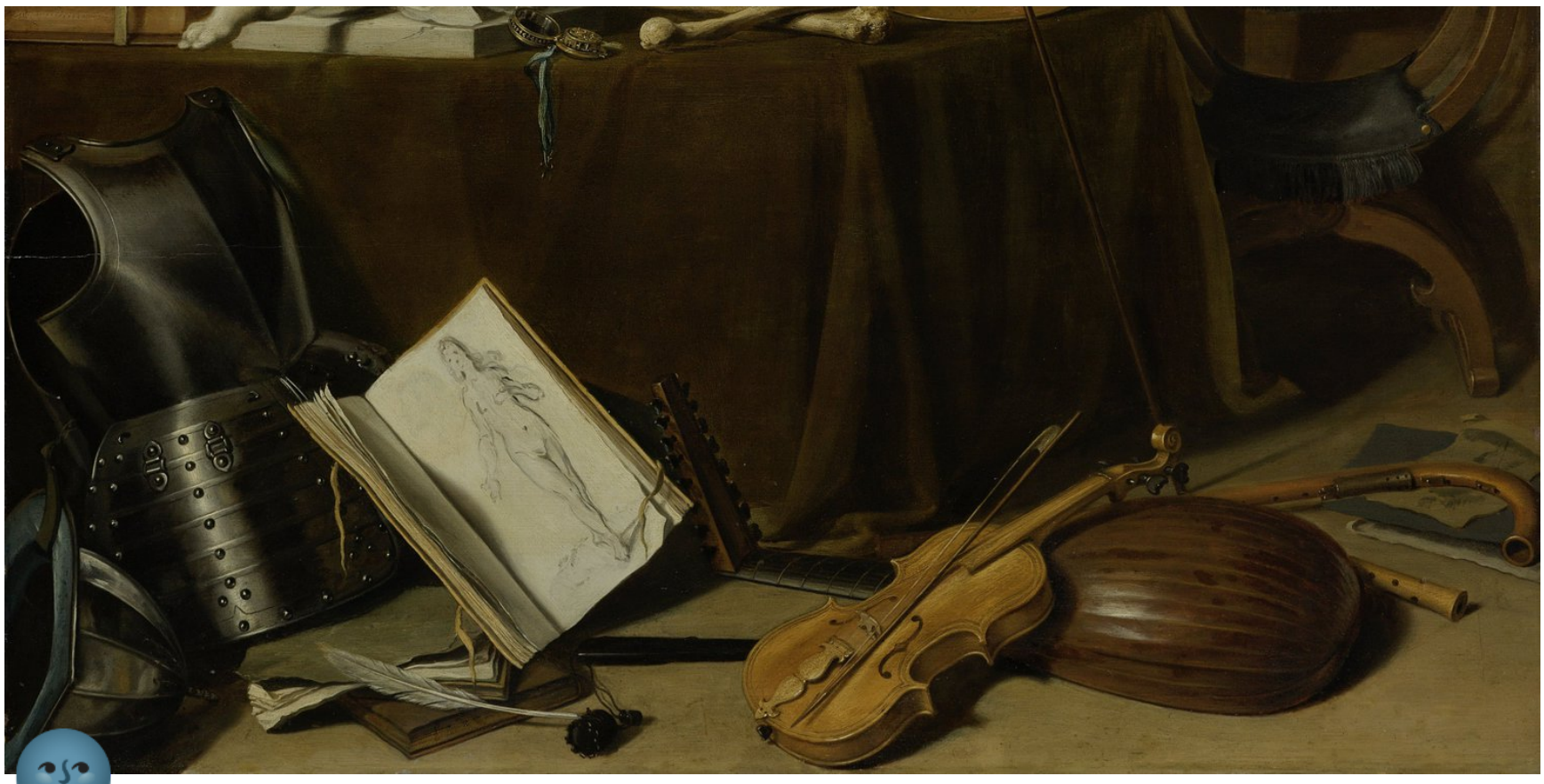

O Universo dos Testes unitários

A primeira coisa que temos que falar sobre testes unitários, e que temos que entender que testes unitários, são testes que deveria verificar unidades de comportamento, ao invés de unidades de código , de forma rápida e isola. Veja que temos três características simples, sendo que as duas primeiras são subjetivas pedaço de código (unidade) e rápido, e a ultima isolada, e neste nos temos uma divergência. Dentro dos Testes nos temos duas escolas a Escola Clássica (ou Detroit) e a Escola Mockista(ou Londres) .

Testa uma unidade de maneira isolada para a escola de Londres, representa que se classe que você esta testando e dependente de outras classes e necessário que sejam substituído por dubles de testes, para que assim só o que seja testado seja o comportamento de classes ou método em teste.

▼ **O que é um duble de testes:**

> Um dublê de teste é um objeto que se parece e se comporta como sua contraparte pretendida, mas na verdade é uma versão simplificada que reduz a complexidade e facilita o teste. Esse termo foi introduzido. por Gerard Meszaros em seu livro xUnit Test Patterns: Refactoring Test Code (Addison-Wesley, 2007).

Nesta abordagem temos uma complexidade, para realizar a separação da dependência, pois e necessário tem um conhecimento prévio do comportamento da dependência, de forma que possa se simular. A vantagem e que na eliminação das dependências em caso de erro, não a como negar que o erro esta na logica da função, pois já que não á outros suspeitos.

Agora na escola de testes clássica, como e de se imaginar não existe o isolamento das dependências, a escola Clássica considera que mesmo as dependência fazem parte de código, então devem ser testadas como um bloco. E o isolamento referencia e referente aos testes, cada testes deve ser isolado dos outros testes de forma que possam ser rodados de forma paralela, sem afetar o comportamento dos outros testes.

Conversando sobre dependências

Eventualmente temos três tipos de dependências são as:

- Dependências Compartilhadas

  São as dependências que são compartilhadas entre vários testes, com talvez um serviço de arquivos.

- Dependências Privadas

  São todas as dependência que não são compartilhadas

- Dependências Fora do Processo

  São dependências que não são iniciadas pelos sistemas, e geralmente, também são dependências compartilhadas , um dos exemplos, seria banco de dados, eles é uma dependência fora do sistema e pode ser compartilhada, se você destrói e remonta o banco de dados a cada testes ele deixa de ser compartilhada mais ainda não fora do processo

- Dependências Voláteis

  São dependências que tem configurações adicionais no ambiente, para serem executados como o banco de dados e outra característica é a resposta não determinística, como API que trabalham com data e hora.

- Dependências Imutáveis

  Imutáveis são com de espera são objetos e valores que não mudam, um valor fixo dentro de classe pode ser considerado como uma dependência imutável, uma função de elevar ao quadrado qualquer numero, onde temos um valor para uma variável que é dois que representa para elevar ao quadrado. essa variável e imutável e privada.

- Colaboradores

  Colaborados são classes que realizam o meio termo entre uma classe compartilhada que gerencia uma classe compartilhada fora de processo, como uma classe que gerencia os processos de interação com o banco de dados.

Dependências em testes:

Vamos falar brevemente sobre as dependências de testes, dentro para as duas escolas:

Para a escola londrina, todas as dependência deveriam ser substituídas, por dubles de testes, menos as dependências imutáveis, essas são teriam necessidade de serem substituídas por conta da natureza estática. Enquanto para a escola clássica só as dependências compartilhadas deveriam ser substituídas.

Padrões de Teste

Temos um padrão de testes conhecido como os 3As, que nos ajuda e nos orienta na escrita de um testes, qual a ideia é orientar a como realizamos testes. A estrutura dos 3As e bem simples são Arrange (arranjo), Act (ação) e Assert (afirmação), essa estrutura e muito semelhante a outra conhecida como GWT ou Given (dado), When (quando) e THEN(então), realmente a grande diferença entre elas é que GWT e mais legível, por pessoas não técnicas e é comumente usado em BDD. agora vamos entender com um contexto de negocio como essa estrutura funciona.

Dado (given\arrange) uma identificado de cliente

quando(when\act) executar o sistema e passado o identificar

Então(then\assert) me retorne os dados unicamente desse cliente

No Given, é onde prepararemos as variáveis, as nossas entradas que passaremos para a sistema. Na parte do When é quando é se refere a chamada do objeto em alguns casos temos algumas condições, e isso esperado quando estamos valando de IU, mas eventualmente será uma referencia direta a execução do método, função, classe, sistema. E o Then

por sua fez relaciona o resultado esperado sendo igual resposta do sistema. Mas eventualmente como isso dentro de um código vamos ao seguinte exemplo

```
def elevate_number_squart(
  number: float or int) -> float or int:

  """
  Eleva qualquer numero que forma
  passado para o quadrado.

  Args:
    number(float or int): numero que vai ser elevado
    ao quadrado.

  Return:
    flaot or number: retorna o numero elevado ao quadrado.
  """

  return float(number) ** 2

def test_elavate_number_sqrt_sixteen():

  """
  Given: Dado o numero 4
  When: Quando chmada a função de elevar ao quadrado
  When: Deve retornar valor igual a 16
  """

  # given or arrange
  # onde são preparados os dados

  number = 4 # numero de entrada
  expect = 16 # numero esperado

  # when or act
  # a chamada para a função ou sistema
  result = elevate_number_squart(number= number)

  # then or assert
  # compara a resposta e o resultado
  assert(result == expect)
```

Tamanhos esperados para essas funções:

Arranjo or Given:

Geralmente é esperado que essa seja o maior do três, até mesmo maior que o when e o then, junto. muitas vezes e mais vantagioso abstrair esses dado para uma classe privada dentro do conjunto de testes, ou abstrair para uma classe fabrica. Dois padrões famosos que ajudam a organizar e reutilizar código são Object Mother e Test Data Builder.

Act or When:

Quando um testes tende de utilizar duas ou mais linhas e um ponto de alerta, por exemplo, teste que quer avaliar se ao realizar um compra, o sistema ele também atualiza as informações, então quando chegamos no ponto da ação, você tem que chamar duas funções a primeira realiza a compra e a segunda realiza o update da informação, e tudo bem os testes passam. Mas se essa é um parte importante do seu código, e exige um testes assim, e de se esperar que nesse caso deve ser feito é uma função publica abstrair essas funções ou encapsula ou chama essas duas funções dentro dela.

Esse problema de termos uma ação em duas linhas significa que seu código esta sujeito a possíveis inconsistências. o cliente ou o sistema pode chamar a primeira função, mas não garante que ele chame a segunda. Esse contexto é denominado de **violação invariável,** é o ato de proteger contra essas possíveis inconsistência e nomeada como **encapsulamento**.

Assert or Then:

Uma informação importante é que geralmente esperado que um testes de unidade, tenha uma única afirmação, enquanto os testes de integração, que são testes mais lentos, para agilizar possamos ter mais de uma afirmação. E por que é

esperado os testes de unidade, tenham unicamente uma afirmação, e é bem simples os testes de unidade devem verificar uma unidade de comportamento, e quando você tem duas afirmações você sai um pouco do domínio, e você deixa de teste uma unidade de comportamento.

No caso da afirmação enquanto a tamanho, se elas ficam muito grande pode representar um sinal que talvez, a uma falha na abstração dentro do método, pois identifica que a uma dificuldade, em se realizar a igualdade.

Evite Usar IF

Quando usamos if dentro de um testes para verificar a saída, significa que parte do comportamento, mas para detalhes de implementação estão vazando, é você já não, s quando utilizamos if nos acabamos também quebrando a regra te testarmos um comportamento pois não temos certeza do comportamento.

No caso do comportamento, não ser totalmente explicito por conta de uma dependência volátil, a melhor proposta seria mockar essa dependência e controlar o funcionamento da função. Por que o mais importante dos testes e que eles seja um sequencia simples, que verifica um caminho, então não deve haver ramificações.

Desmontar - E um 3º Padrão de Testes

Desmontar e montar e uma outra etapa que pode ser incluída dentro das estruturas de testes; Se formos pensar a testes que são mais complexos, por exemplo um API que escreve dados em um banco de dados e necessário que nos testes tenhamos um banco de dados, é em alguns testes criar um banco de dados e só ir injetando informações pode enviesar os teste e aqui, então nesse ponto nos temos o nova etapa que é **desfazer** o que foi realizado, ou seja para utilizar um banco de dados, você vai criar um banco de dados, e assim que terminar o testes você destrói e na próximas vez que você for realizar o teste você vai criar, testar e deletar o banco de dados.

Para referencia essa nova estrutura nos temos um terceiro padrão de teste, que é muito comum de se ser encontrado:

> Setup - Given | Dado
> Exercise - When | Quando
> Verify - Then | Então
> TearDown - Desmonta

Reutilização de Código

Em muitos testes repetimos as mesmas coisas, os mesmo processos e uma forma de eliminar redundância de código e abstrair eles para outras entidades como classes privadas e etc. porem a reutilização de código dependendo da forma que for realizada pode implementa o acoplamento entre os testes quando um classes e utilizada em mais de um testes, uma solução para isso seria realizar a fase de desmontar os dados a cada testes, entretanto essa técnica também tem sua desvantagens pois adiciona tempo extra para deletar e criar as instancias novamente. Um dos outros porem da reutilização de código e que ao realizar nos dificultamos a legibilidade dos testes. pois faz que quem estiver olhando tenha que retornar a classe ou método para entender o seu objetivo dentro do testes,

Legibilidade

Uma das coisas de favorecem a legibilidade dos testes e primeiramente um bom nome, que permita quem olhar entender o que esta sendo executado e qual é a expectativa. e fato que existem padrões ou recomendações de nome como o abaixo, entretanto não devemos nos ficar presos a padrões, o objetivo e a adoção de um padrão de seja legível para os desenvolvedores e em muitas vezes esses nomes ficam sujeitos a algum padrão corporativo e etc.

Padrão nome: [Método sendo testado]_[cenário]_[resultado esperado]

se fossemos escrever do testes que escrevemos acima:

```
def test_elavate_number_sqrt_sixteen():

  """
  Given: Dado o numero 4
  When: Quando chmada a função de elevar ao quadrado
  When: Deve retornar valor igual a 16
  """

  # given or arrange
  # onde são preparados os dados

  number = 4 # numero de entrada
  expect = 16 # numero esperado

  # when or act
  # a chamada para a função ou sistema
  result = elevate_number_squart(number= number)

  # then or assert
  # compara a resposta e o resultado
  assert(result == expect)
```

se fossemos refratora esse nome como ele ficaria aplicando o padrão acima:

test_elevateNumberSquart_numberFour_sixteen

Uma das outras coisa que nos ajudam a aumentar a legibilidade dentro de uma código é a utilização de comentários. Os testes devem contar uma historia e os comentários auxiliam, nessa função. A utilização recomentada e indicar dependendo do padrão que você adotar onde estão os act, arrange ou given ou etc, dentro do seu código.

Testes Parametrizados

Testes parametrizados e uma funcionalidade que permite injetar valores em um mesmo testes, para realizar o mesmo testes com parâmetros dizentes, por exemplo:

```
from pytest import mark

# given or arrange
# onde são preparados os dados
@mark.parametrize(
    'entrada,esperado',
    [(4, 16), (12, 144), (1,1)]
)
def test_elavate_number_sqrt_paramtrize(entrada, esperado):

  """
  Given: Dado respesctivamente os numeros 4, 12, 1
  When: Quando chmada a função de elevar ao quadrado
  When: Deve retornar respectivamente o valor 16, 144, 1
  """

  # when or act
  # a chamada para a função ou sistema
  result = elevate_number_squart(number= entrada)

  # then or assert
  # compara a resposta e o resultado
  assert(result == esperado)
```

Veja que no exemplo acima eu modifico eu tenho uma decoretor “@mark.paratrize” que executa para mim essa para metrização e nele eu especifico as entradas da função e depois passo uma lista de valores o em cada parênteses ei tenho dois valores , o primeiro é o valor que eu passo para minha função executar e o segundo é o resultado esperado. ou seja para cada parênteses ele vai executar um testes, o primeiro ele utilizar o 4 como entrada na função e espera como resposta o 16, no segundo ele vai passar o 12 e esperar o 144 como resposta e o ultimo vai passar o 1 e esperar 1 com resposta.

Muitas bibliotecas de testes permitem a realização da parametrização dos testes.