Indo Mais Fundo em Teste

Testes Unitários:

Uma das verdades sobre testes unitários e que mesmo eles não são bons como indicadores de que o software tem qualidade, não se trata de serem ou terem o objetivo de serem um indicador de positivo, mas sim seu oposto. ele server muito melhor como indicador negativa que aponta os pontos de melhoria e fragilidade dentro de uma código, e seu objetivo, auxiliar num processo de desenvolvimento saudável

Em geral o desenvolvimento sem teste inicialmente e mais rápido mas com forme vai evoluindo o desenvolvimento do software, manutenção, refatoração, novas versões, o tempo de desenvolvimento e as horas gasta vão crescendo exponencialmente, enquanto um projeto onde os testes são modelados destes o inicio tem seu inicio muito mais lento, porém com o tempo ele vai se estabilizando.

Um copo de agua como software:

Dentro da Física temos um conceito básico da termodinâmica, a entropia, que diz a respeito da quantidade de desordem em um sistema, se fosse lembrar um pouco poderíamos lembrar do exemplo que eu acredito: primeiro temos um copo de agua em um estado original, em um segundo momento esquentamos essa agua então as moléculas de agua com forme vão se esquentando vão se agitando e causando desordem e temos nosso segundo estado, então para finalizar, depois tiramos a agua e esperamos ela esfriar a até a temperatura original, seria certo afirma que temos de volta o primeiro estado, não, uma das características da **Entropia é que quando gerada desordem dentro de um sistema, ele não consegui retornar estado original**, na verdade e gerado um novo estado, com uma nova organização ou melhor mais desordenado, no caso do copo de agua agora temos um terceiro estado que se parece com o primeiro mais quando olhado a fundo as moléculas se organizam de forma diferente.

Agora vamos trazer esses contexto para dentro de desenvolvimento de software:

- 1º Estado: O software funciona normalmente como esperado, considerado um estado estável
- 2º Estado: Um engenheiro, realiza uma implementação de uma nova feature, no software gerando mais desordem dentro do código
- 3º Estado: E aqui é onde os testes brilham os olhos, para facilitar o processo de um estado estável, que ainda não e como o primeiro ele tem mais desordem, ele e um código, é um código que tem mais código e mais réguas e por isso se torna mais desorganizado. Todavia os testes ajudam a chegar nesse terceiro estado mais estável em menor tempo. do que teríamos sem eles.

Eai Sabem o que é regressão?

Provavelmente, vão seja deve ter ouvido falar dos teste de regressão, se não tudo bem. também. Se fossemos explicar o que e testes de regressão seria: “peguei todos os testes que criamos antes dessa modificação e rode tudo de novo para garantir que tudo esta funcionando normalmente e não a bug novos no software’.

Porem gostaria de chamar a atenção para a palavra regressão, muitas pessoas conhecem os testes de regressão, mas não tem ideia do que significa a palavra regressão e isso e muito comum saibam vocês. Regressão é quando feature, para de funcionar após um determinado evento, muitas vezes poderemos ver referencias a regressões e bugs sendo usados, por que até então muitas vezes acabam sendo sinônimos.

Então quanto estamos falando de testes de regressão, estamos falando de testes contra regressões, e isso e uma característica muito importante para um desenvolvimento saudável.

A Armadilha nos Testes de Cobertura:

Primeira o que são os testes de cobertura, é um teste que avalia e retorna um porcentual de quanto do seu código e testado utilizando os testes unitários ou testes de integração. ou seja quanto do seu código realmente esta sendo testado

Mas os testes de cobertura são uma das grandes armadilhas, que muitas pessoas decidem acreditar, se eu perguntasse: você acredita que ter 100% dos de testes de cobertura para o seu código, é um indicativo que o seu código tem qualidade, você pode responder como a maioria que, sim, alias são 100%, isso tem que indicar algo positivo. E no caso chegamos a armadilha ter 100% de cobertura de teste, não diz muita coisa, no sentido positivo da coisa, na maioria dos casos pode indicar coisas ruins, Okay, Okay, então o que você imagina que poderiam ser essas coisa ruins? pense um pouco…E então já descobriu o que é?.

Mas antes de entregarmos a resposta, vamos falar um pouco sobre os testes de cobertura, para que você conheça um pouco mais os tipos. Ou seja vamos deixar você sofrendo um pouco, com essa crise existencial, voltadas a testes.(”-hihihi!” - nota do autor).

Conhecendo testes de Cobertura

O primeiro tipo de teste que conversaremos, é o mais comum conhecido como cobertura de código, avaliada a quantidade de código executado para a quantidade pela quantidade de linhas totais do codigo. com podemos ver pela seguinte formula:

$$X = \frac{\Delta l}{l}$$

onde temos:

- X - Porcentual de cobertura
- Δl - Linhas executadas
- l - linhas totais do arquivo

Vamos analisar o seguinte código:

```
def equal_number_four(number:int) -> bool:

  if int(number) == 4:
    return True
  else:
    return False
```

e temos os seguintes testes:

```
def test_equal_number_four_true():

  four= 4

  result = equal_number_four(number = four)
  assert(result, True)

# passa no testes
```

No nosso exemplo, qual seria a cobertura do nosso código:

temos que:

- um total de 5 linhas de códigos lembrando que não devemos considerar as linhas vazias ou linhas de comentário, pois as mesma não devem ser executadas. (ou pelo menos não deveriam)
- Δl - E um total de 3 linhas. pois como o testes retornou verdadeiro ele não passou pelo else,

então a nossa cobertura é de:

$$60\% = 0,6 = \frac{3}{5}$$

E claro que poderíamos realizar dois testes um que passar, pelo if quanto pelo else, mas entretanto a uma situação mais interessante que seria comprimir e estrutura e usara um operador ternário:

```
def equal_number_four(number:int) -> bool:

  return True if int(number) == 4 else False
```

▼ **entendo operador ternário python**

Operador ternário do python segue a seguinte estrutura:

resposta para se if condição else retorno para o se não

Veja que agora os nossos testes passariam com 100% das linhas de código, muitas vezes, ao invés de realizar vários testes, e necessário avaliar se no código de produção, não existe nenhum ganho de melhoria. e que todas as estruturas presentes, são realmente necessárias.

Cobertura por agencias

Diferente do método de cobertura por código, esse métrica é relativamente melhor, esses testes se foca em estruturas de controle como if e switch, e o seu calculo e similar outro

$$Y = \frac{\Delta f}{f}$$

onde temos:

Y- Porcentual de cobertura

Δf - total de estruturas que o teste passa

l - total de estruturas de controle no código

Vamos pegar o exemplo anterior para entender melhor:

```
def equal_number_four(number:int) -> bool:

  if int(number) == 4:
    return True
  else:
    return False
```

```
def test_equal_number_four_true():

  four= 4

  result = equal_number_four(number = four)
  assert(result, True)

# passa no testes
```

Para esses testes qual seria a cobertura pela métrica de agencia:

Temos:

- 2 estruturas de controle dentro do código ou seja dois caminhos, se for um ele vai para um caminho e se não for ele vai para outro, ou seja temos dois caminhos
- Desses 2 Caminhos o nosso testes só passa por um deles

Então o nossa cobertura fica?

$$50\% = 0,5 = \frac{1}{2}$$

O que não te contaram sobre os testes de cobertura:

Primeiro antes de jogar pedras no pobre testes de cobertura, tenho que aponta que os testes de cobertura com os testes de unidade, não são um bom indicador positivo, mas sim um ótimo indicador negativo, pois quanto temos níveis muito baixo por exemplo 50% ou 60% significa que grande parte ainda no nosso código ainda não foi testada, isso deveria ser alerta, para informar que ainda existam partes importantes no nosso código que ainda não foram devidamente testadas.

Agora se nos tivermos 100% de testes, por muitos motivos podemos dizer que não significa, muito pois primeiramente testes tem limitações, segundo eventualmente nem todos testes tem valor, os testes não nascem iguais, cada testes deve ter um valor especifico para o código. muitas vezes o indicativo 100% pode indicar que nos estamos realizam em muitos casos teses triviais e frágeis, que não muito valor, na verdade esses tipos de testes com o tempo podem problemas, acionando em falsos positivos e negativos. e ai esta a nossa resposta.

Outras das coisas que limitam os testes de cobertura essa e uma limitação dos teses unitários e até mesmo de integração, que num podemos testas todas as entradas e saída, existe um infinidade de entradas que um usuário pode colocar, e para cada um dessas um entrada uma resposta, também em alguns casos temos que avaliar a entrada como parte de uma de um caminho ou algo que define um caminho.

```
def equal_number_four(number:int) -> bool:

  if int(number) == 4:
    return True
  else:
    return False
```

Neste simples exemplo que vimos anteriormente, vamos explorar ele se por acaso o usuário colocar “” ou “abc” como entrada pra essa função, ele vai gerar um caminho onde retornara um ValueError e se ele colocasse [] ou None, como entrada ele nos retornaria TypeError, veja que mesmo para um testes simples. nos temos vários caminhos, e ainda a outros caminhos que podemos tentar com colocar um valor Float como entrada e etc. Outra das falhas do testes de cobertura e que ele não considera os possíveis caminhos gerados por bibliotecas externas.

OS testes como um todo devem ser parte do desenvolvimento e devem se dedicar as partes importantes do sistema, com o objeto de trazer o máximo de valor e custos mínimos de manutenção. (” -acho que uma hora você vão se cansar de quantas formas eu consigo dizer isso de varias maneiras, eu já estou, e cansativo, mas não tem o que fazer, né”).

OS Testes em geral devem se direcionara testar as partes mais importantes de código, que geralmente e o modelo de domínio, onde se encontra a parte logica e onde estão as regras de negocio do sistema. enquanto o resto deve ser testado brevemente ou de forma indireta.

Bons e Maus Testes

Para entender como realizar bons testes primeiro precisamos entender alguns conceitos de testes um dos principais conceitos são os pilares de testes que abordaremos em seguida:

Princípios de testes

Os testes tem 4 pilares básicos:

Proteção Contra Regressões

O primeiro pilar que iremos discutir é a o pilar de proteção contra regressões, já discutimos um pouco o que significa regressões, mas retomando é quando uma funcionalidade para de funcionar devido a um evento, muitas vezes a alteração de código; E este pilar fala especialmente sobre esse pilar como garantimos que o a cada nova modificação que realizarmos os testes que ja criamos seja o suficiente para avaliar que a funcionalidades as quais já tínhamos ainda se mantem, ou que caso não esteja a mesma seja apontada pelos testes.

Dentro dessa pilar nos temos que ter a ciência que quanto maior o código, bibliotecas internas ou externas, existira uma maior probabilidade de gerar uma regressão e bugs. devemos também avaliar a complexidade do código, no que diz respeito a regra de negocio e se as nossas asserções cobrem bem esses requisitos de negócios. de forma que exercitemos a maior quantidade de código possível. Entretanto é claro que existe um porém que e realização de testes triviais onde estes não tem muito valor, principalmente para esse pilar pois os mesmo não tem muita chance de ocasionar em regressões.

```
# exemplo de codigo trivial
def corta_nome_char20(nome:str) -> str:
  return nome[:20]
```

```
class Usuario:

  def __init__(self, nome):
    self._nome = nome

  # serua yn testes trivial testar
```

```
# um get simples como esse
def get_nome_user(self):
  return nome
```

Resistência á Refatoração

Primeiramente o que seria Refatoração?

significa realizar alteração em código já existente com o objetivo, diminuir a complexidade e melhorar legibilidade, sem alterar o comportamento da funcionalidades alteradas.

Agora podemos continuar, o pilar de resistência á refatoração, apresentar o grau a qual os testes suportam, as modificações das funcionalidades, sem que os testes quebrem, em geral os testes são feitos para que a cada modificação eles acusem o comportamento irregular pela funcionalidade, mesmo quando você não a quebre.

Uma das grandes características desse e garantir que principalmente que esses falsos positivos, não sejam capturados a cada refatoração. Quando alterado um funcionalidade e o mesmo mantem o comportamento intacto, e este não não são capturados pelos testes e denominado como uma testes unitário com resistência a refatoração.

E quais são os benefícios disso?, são essencialmente dois;

- Garantir que no caso de a funcionalidade seja quebra tenhamos um aviso preventivo, de um defeito e para que possamos corrigi-lo de maneira tempestiva, permitindo que não entre em produção
- E a confiança em relação as mudanças realizadas, poder confiar que as mudanças, realizadas estão muito menos propensas a gerar alguma regressão;

O Problema dos Falsos Positivos:

Um grande problemas dos falsos positivos, é quando eles se tornam parte da rotinas de teste, e é muito comum encontra-los, quando as pessoas se acostumam com as falhas, passam a ignora-las, também poderá acontecer de ignorar as falhas legitimas, apontadas pelos seus testes e isso pode se tornar muito, problema a curto ou a longo prazo. Um outra característicasdecorrente é o aumento da desconfiança em relação as mudanças e a hesitação e incerteza.

Manutenibilidade

Eventualmente avalia o custo de manutenção, primeiramente podemos apontar o tamanho do código de teste, o quão legível e simples para realizar mortificações, este código é. Porém não nos devemos nos preocupar em reduzir os nossos testes, pois isso afetaria de forma direta a qualidade dos Testes. Outro ponto é as dependência, em principal as foras do processo, o que e necessário para rodar os testes, conectividade, ter um banco de dados disponível, o quanto custa para manter essas dependências.

Feedback Rápido

Feedback rápido, tem uma caráter um tanto quanto descritivo, representa a velocidade que temos uma resposta dos teste, isso se reflete no tempo de execução dos teste, em principal nos testes de unidade e esperado que tenha um tempo menor que os outros.

Pesando os Testes

Entre os pilares de temos 2 deles são muitos importantes e geralmente esperamos máxima o grau de confiança neles, o primeiro é o de proteção contra e regressões e o segundo de de resistência a refutação

realização dos testes	Esperado Negativo	Esperado Positivo
Resposta Negativa	Negativo	False Negativo (falha proteção contra regressão)
Resposta Positiva	Falso positivo (falha na resistência a refatoração)	Positivo

Observando a tabela acima, podemos identificar que temos um relação entre a asserções dos testes e esses pilares, uma ponto importante e que, decisivamente é importante que primeiramente o código tenha uma proteção contra regressões, nas suas vazes iniciais enquanto, não houver ainda modificações e conforme for avançando no seu desenvolvimento também tenha um grau de resistência fatoração. Ou seja inicialmente os falsos positivos não são tão preocupantes quanto os falsos negativos, mas com o tempo pode se tornar um grande problema.

E uma da maneiras realizar o calculo do ruido gerados pelos testar, a precisão testes em relação ao ruido é realizando a diferença entre os bugs encontrados sobre os falsos alarmes

$$X = \frac{\sigma}{\epsilon}$$

σ = Quantidade de bugs encontrados

ε = Quantidade de falsos alarmes

X = Precisão dos testes em relação ao ruido

Estimando valor dos testes:

Para estimar os valores de testes devemos, utilizar os quatro pilares como parâmetros, para um testes ser valiosos, precisamos pontuar pelo menos alguma coisa em cada pilar, e para isso definimos uma grau de valor, entre 0 e 1, que o nosso destes pontua e o seu produto representa o valor estimado para o teste. Podemos representar pela seguinte equação:

$$valor = [0..1] * [0..1] * [0..1] * [0..1]$$

Entretanto eu não gosto muito bem dessa abordagem pois, basta uma métrica seja mais abaixo que a maioria e o valor do seu teste despenca e muitas vezes, para pessoas de negócios, que tem acompanhar não conseguiriam ver o real valor pelo resultado final. um dos fatores que contribui para eu não gostar dessa abordagem é que proteção contra regressões, resistência a refatoração e feedback rápido, são valores exclusivos então para potencializar dois deles e necessário defasar um outro. Isso eventualmente já seria um indicativo para o valor de testes decair. Para mim uma medida complementar que ajuda a visualizar melhor o equilíbrio entre os pilares, porém não exclui a necessidade do calculo do valor é a seguinte:

$$balançoteste = \frac{[0..1] + [0..1] + [0..1] + [0..1]}{4}$$

E apenas uma media simples entretanto nos permite ver o quanto nossos testes estão em equilíbrio em relação ao pilares de testes. Deve se pontuar que os sacrifícios em ralações aos testes devem apresentar uma redução nunca esperamos que os valores cheguem a zero. de forma que devemos sempre pontuar e tais reduções devem levar em consideração motivos estratégicos ou serem parciais. contribuindo para esse ponto, e necessário dizer que a resistência a refatoração e inegociável e deve tentar sempre conseguir performar melhor neste quesito.

Avaliando testes e pilares exclusivos

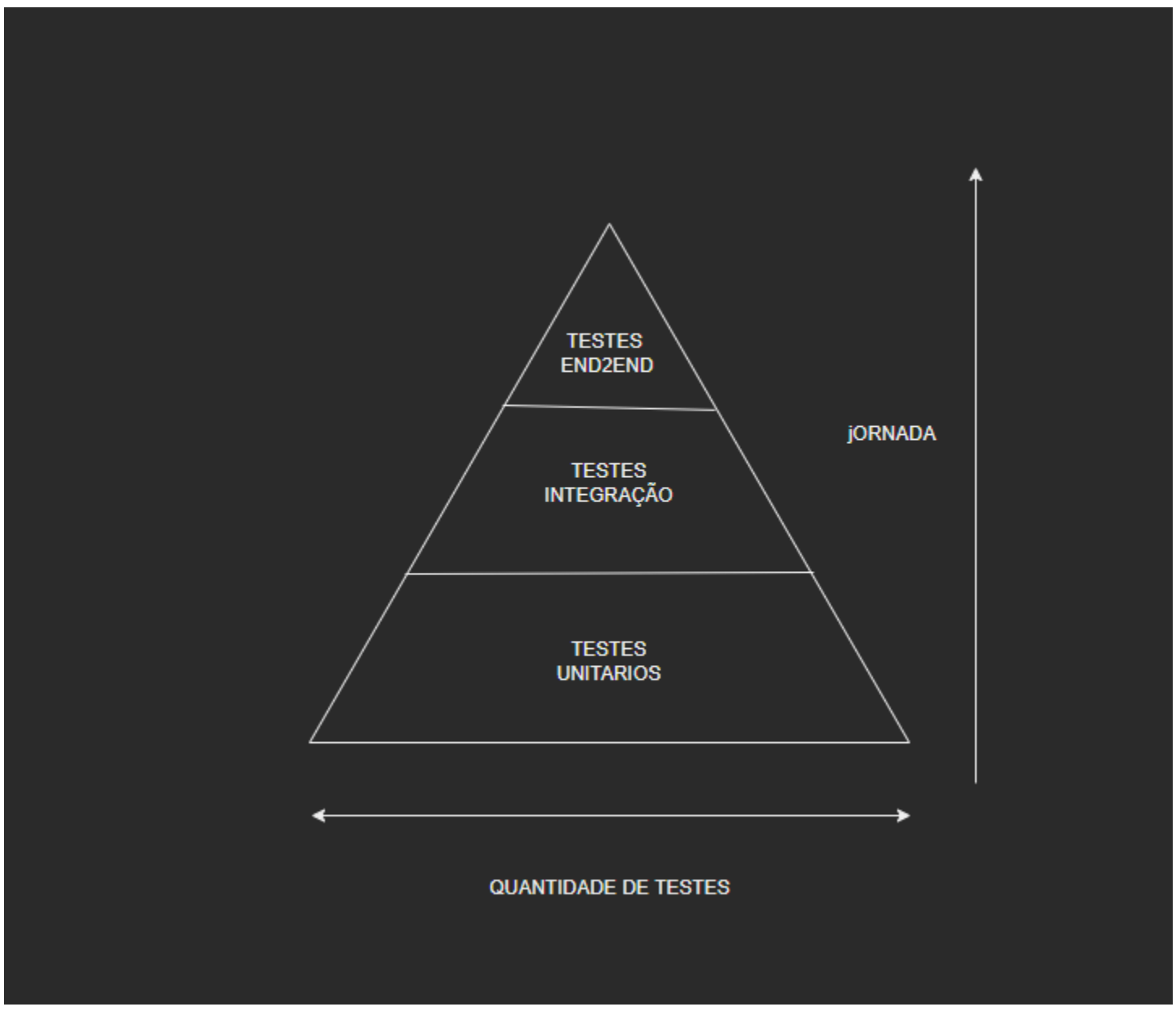
primeiramente vou mostra uma com alguns detalhes sobre testes e os 3 pilares, proteção contra regressões, resistência a refatoração e feedback rápido.

Testes	Resistencia a Refatoração	Proteção contra Regressões	Feedback Rapido
Testes Triviais	Alto - baixa chance de produzir falsos positivos	Inexistente - sem capacidade de revelar qualquer regressão	Alto - São rápidos
Testes Frágeis	Bem Baixo - Produz vários falsos positivos	Regular para Bom - tem chance de revelar regressões	Alto - executam rapidamente
Testes E2E	Alto - imune a falsos positivos	Alto - fornecem melhor proteção contra regressão	Bem Baixo - são lentos

Referente o pilar de manutenibilidade os testes triviais e testes frágeis, por serem mais fáceis também de escrever, porém os testes End-to-End, são mais robustos, e tem um custo maior em mantê-los.

Esclarecendo a Pirâmide de testes

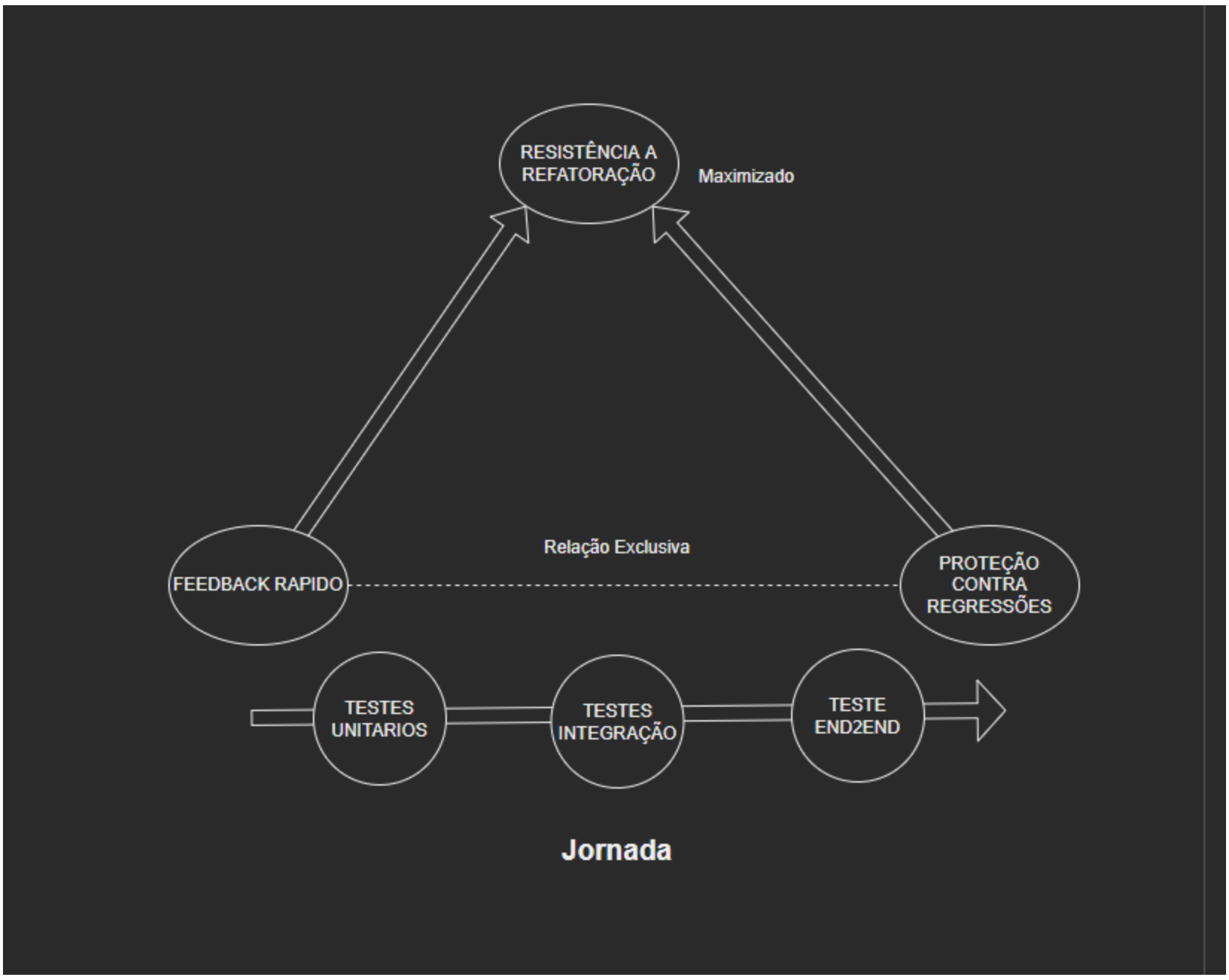
Uma das características da pirâmide de testes, e que ela e criar apartir de dois parâmetros o primeiro é a quantidade de testes realização e o segundo é a jornada do usuário.



uma pirâmide que tem três divisões, a primeira e em baixo testes unitários, do meio testes integração e no topo testes ponta a ponta, e tem uma flexa que relaciona essa ordem a jornada de testes, e a outra flexa embaixo que relaciona a quantidade de testes ao afunilamento da pirâmide conforme ela sobe

A pirâmide de testes tem uma relação, intrínseca com os pilares de testes, que conforme vamos avançando nos testes vão sendo favorecidos alguns pilares. por exemplo já foi comentado que sempre tentamos maximizar o pilar resistência contra a refatoração, é os outros pilares teríamos que decidir qual pois eles acabam sendo exclusivos entre si, no inicio da jornada temos maior favorecimento do pilar de feedback rápidos e conforme nos vamos avançando vamos perdendo,

enquanto para o proteção contra regressões temos uma relação inversa, onde temos maior favorecimento desce pilar no final da jornada do que no inicio.



Temos um triangulo formado por eclipses, onde cada um tem um nome de um pilar no topo resistência a refatoração , que tem um indicativo de esta maximizado, e tem flexa sendo apontada para eles dos outros eclipses, o pilar de feedback rápido que se encontra a esquerda e o da direita o proteção contra regressão, e a um linha tracejada que conecta também essas elipses, com o indicativo de relaçã0 exclusiva. embaixo desse triângulo, temos um flexa da esquerda para a direita e dentro da flexa a três círculos ao longo dela, o primeiro circulo, escrito testes unitários, o segundo testes de integração e o ultimo testes ponta a ponta.