William Queiroz
Posted on 11 de mar. de 2022 • Updated on 21 de mar. de 2022

51 11

Testes Unitários: Fundamentos e Qualidade de Software!

#testing #braziliandevs #qualityassurance

Photo by Ferenc Almasi on Unsplash

Sumário

Introdução

E ae dev, tudo bem com você?

Hoje eu vim trazer um tema que eu gosto beeem pouco (sou apaixonado) que é sobre Testes Unitários e Qualidade de Software!

O post dessa vez está mais teórico que o normal por aqui, mas fica comigo, pega um café e garanta sua leitura!

Bora pro post?

Qualidade de Software

Antes de falarmos sobre testes, precisamos entender o que é que define a qualidade de um software. Da perspectiva de produto e negócios, do nosso "cliente", um software de qualidade é aquele que atende os requisitos funcionais, expectativa e necessidades do usuário final. E qual é a perspectiva do desenvolvedor quanto à qualidade?

Em 1977, James A. McCall propôs um modelo que define critérios de qualidade de software dividido em 3 pontos de vista distintos, sendo eles:

- **Operação**: que são características relativas ao uso do produto.
- **Revisão**: que é a capacidade do produto ser modificado e evoluído.
- **Transição**: que remete a adaptabilidade à novos ambientes.

Tais critérios, são elencados a partir de cada ponto de vista:

Operação:

- **Corretitude**: medida na qual o software satisfaz as especificações e objetivos visados pelo cliente.
- **Confiabilidade**: medida que se pode esperar que um programa execute sua função com a precisão exigida.
- **Eficiência**: é a quantidade de recursos computacionais exigidos para que um programa execute sua função, visando realizar a operação de forma 100% segura e performática.

- **Integridade**: medida na qual, controla-se o acesso ao software e aos dados, bloqueando assim o acesso de pessoas não autorizadas, para que não ocorra perda de dados ou de código.
- **Usabilidade**: mede a facilidade para a utilização do software.

Revisão:

- **Manutenção**: mede o esforço exigido para localizar e reparar erros em um programa.
- **Flexibilidade**: analisa o esforço utilizado para realizar uma alteração no software, isto é, qual o grau de facilidade que o software oferece para a sua alteração.
- **Testabilidade**: analisa se é possível testar o funcionamento do software, não só por vias automatizadas.

Transição:

- **Portabilidade**: mede a facilidade com que um produto pode ser movido para outra plataforma ou ambiente.
- **Reusabilidade**: medida na qual o software, ou parte dele, poder ser reusado em outros softwares, em outras palavras, o código do software deve ser reaproveitável.
- **Interoperabilidade**: capacidade do software ser acoplado à outro.

Dito isso, qualidade de software da perspectiva do desenvolvedor é uma área que visa garantir a qualidade do software por meio de normatizações e definições de processos de desenvolvimento. Testar é apenas **uma das maneiras** que temos para garantir a qualidade do nosso software. Ainda assim, vamos responder a seguinte pergunta.

Por que eu devo testar meu código?

Você em algum momento da sua carreira deve ter se perguntado: "*será que se eu alterar isso aqui, isso ainda vai continuar funcionando?* 🤔". Já sentiu aquela insegurança na hora de fazer aquela *feature* ou *refactor*? Seu código/projeto não trazia nenhuma **confiança** quando estava tomando proporções maiores?

Pois é... e como saber se aquilo que você implementou funciona **em conjunto** com aquilo já foi implementado? A melhor maneira de saber se algo funciona é: **testando**.

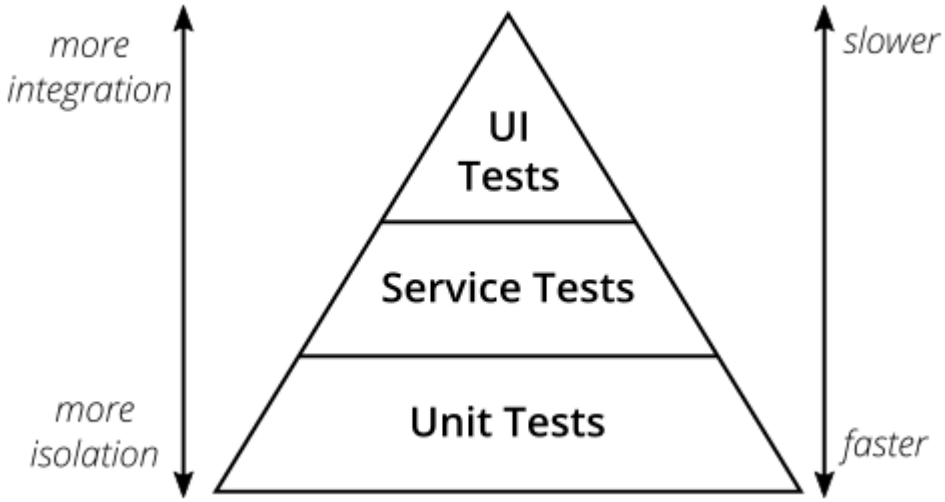
Ao testar o seu projeto, você garante mais:

- Confiabilidade;

- Integridade, evolução e melhorias no código;
- Velocidade nas entregas e demandas (uma vez que um código existente já possui testes, a preocupação nas mudanças e implementações futuras é menor, podendo-se ter mais agilidade e previsão ao decorrer do desenvolvimento); Isso vem em contrapartida de que se eu implementar testes durante a entrega eu vou levar mais tempo. Isso é gradual, à medida em que o software é contemplado com mais testes, mais demandas com qualidade são entregues.
- Qualidade no software (fácil manutenção, flexível a alterações, testabilidade e etc, tudo que falamos na introdução);
- Redução no tempo ao encontrar bugs;
- Implementações mais objetivas e códigos mais limpos.

Agora que você já sabe os motivos pelos quais você deve implementar testes, vamos conhecer alguns dos **tipos de testes**!

Tipos de testes



> Fonte: https://martinfowler.com/articles/practical-test-pyramid.html

A maioria das pessoas conhecem essa pirâmide graças à Mike Cohn que a descreveu no seu livro de 2009, Succeeding with Agile. No livro ele faz referência à "*Test Automation Pyramid*" (pirâmide de automação de testes) que popularmente ficou conhecida apenas como "*Test Pyramid*" (pirâmide de testes).

Ela é constituída por 3 camadas:

1. Testes Unitários
2. Testes de Serviço
3. Testes de Interface do Usuário

Pode-se observar que, à medida em que chegamos ao topo da pirâmide, menor é quantidade de testes que teremos e mais "caros" eles serão (pensando em processamento). Quanto menor é o isolamento, mais rápido será a execução dos nossos testes.

> 💡 Recebi uma excelente recomendação do Paulo Gonçalves, membro do Agile Testers de alternativas e materiais complementares à pirâmide de testes. Um deles é o artigo "*Abordagem de testes*" e a talk "*AT Talks: Triângulos de Teste, Não Mais*" da Samanta Cicilia. Os links você encontra na seção Referências e Links.

Testes Unitários

Estando na base da pirâmide, os testes unitários além de ser a fundamentação, eles são responsáveis por testar a menor unidade do nosso software de maneira isolada. "Tá, mas qual é a menor unidade do software?": podem ser métodos/funções, classes, módulos, etc. Esses testes validam o comportamento esperado de um método/função: dada uma determinada execução **espera-se** uma saída.

Como testamos as funções do software, podemos ter diversos comportamentos a cada execução com determinadas condições e, consequentemente, vários testes. O teste unitário nos possibilita criar um código mais objetivo, com métodos limpos e com uma única e determinada funcionalidade (daí, associa-se também o conceito de "unidade"), sendo possível ser testado de maneira isolada.

Vamos ver um exemplo de teste unitário com JavaScript! Imagine que temos uma calculadora, e que centralizamos as funções das operações em um arquivo. Precisamos testar o comportamento esperado de cada uma dessas funções. No exemplo, espera-se que a função `sum` retorne a soma de `a` e `b`. Como seria o nosso teste?

```
// calculator.js
const sum = (a, b) => a + b;

export { sum };

// calculator.spec.js
import { sum } from './calculator.js';

describe('calculator.js', () => {
  it('should return the sum of a and b', () => {
    const a = 1;
    const b = 2;
```

```
      const result = sum(a, b);

      expect(result).to.equal(3);
    });
  });
```

Na descrição do nosso teste, montamos o cenário de teste de acordo com a especificidade da função.

Você precisa ter a seguinte reflexão quando for escrever os seus testes unitários: "essa função/método deve fazer **isso** quando **aquilo**". Ou seja, para um determinado **comportamento**, é necessária uma **condição**.

Dicas para escrever bons testes unitários

- Escolha as melhores asserções para cada momento.
- Evite ruídos e dependências entre os testes, garanta o **isolamento**. Se o seu teste depender de quaisquer fatores externos ele **não é mais um teste unitário**.
- Utilize do teste unitário para definir o **design do seu código**. Durante a escrita dos testes, é comum identificarmos a necessidade de refatorar o código para melhorar o seu uso (e também isolá-lo). Aproveite desses momentos.
- Por último, mas não menos importante: evite "estressar" camadas que excedam a barreira de unidade: banco de dados, 3rd party APIs... Os testes unitários sempre superarão os testes das camadas superiores da pirâmide. Garanta que a execução dos testes unitários seja rápida, use *Test Doubles*!

Por falar nisso, você sabe o que são *Test Doubles?*

O que são Test Doubles

Segundo Martin Fowler, "*Test Doubles*" é um termo genérico para qualquer caso que você queira substituir um objeto de produção para fins de teste. Pense em "dublê", aqueles de cinema mesmo, que "simulam" a aparência e o comportamento dos atores em um filme. Você provavelmente deve conhecer o conceito de maneira generalizada como "mockar" ou pela palavra "Mock". O fato é que Mock é apenas um dos tipos de "dublês" que temos disponíveis.

Além dele temos Dummies, Fakes, Spies e Stubs.

Dummies, Fakes, Spies, Stubs e Mocks

Dummies: são objetos ou dados fictícios que substituem dados reais mas que não utilizados nos testes geralmente. São usados para satisfazer parâmetros. Com o uso

deles é possível diminuir a complexidade dos testes, focando no que importa.

Fakes: são objetos reais que implementam um comportamento que os torna inviável para serem utilizados em produção (um In-Memory Database, por exemplo).

Spies: são "espiões" para registrar as informações de uma determinada função. Pode ser usado para dizer se um específico método importante no escopo da função que está sendo testada foi chamado, quantas vezes, com quais argumentos e etc.

Stubs: são semelhantes aos spies porém, ele consegue substituir toda a implementação de um específico método, mudando o seu comportamento. São úteis para simular uma possível exceção, evitar estresse na camada de serviço: persistência de dados (evitando de gravar um registro no banco de dados, por exemplo); chamada à uma API ou integração a outros serviços possibilitando a diminuição no tempo de execução do teste.

Mocks: São similares ao stub porém, é a camada mais alta para tornar um comportamento falso, geralmente, são usados para emular um banco de dados, um *output* de dados ou até mesmo uma dependência, a diferença é que, com um stub, é possível testar diversas ramificações do seu código, já com o mock, é possível testar diversos comportamentos de uma só vez. Os Mocks tem expectativas sobre o jeito que ele deve ser chamado e, caso não seja chamado da forma correta, o teste deve falhar.

Diferenças entre Mocks e Stubs

Para entender a diferença entre mocks e stubs vamos imaginar que toda vez que um usuário criar uma conta no nosso software, um e-mail é disparado com uma mensagem de boas-vindas e dados referente à sua conta. No nosso cenário de testes não queremos enviar esse e-mail. Podemos usar mocks ou stubs, mas afinal... qual é a diferença?

Observemos a implementação abaixo retirada do artigo "*Mocks Aren't Stubs*" do Martin Fowler (sim, em Java mesmo hahaha). Criamos um stub que implementa o método `send` apenas com um contador (adicionando a mensagem para ser enviada na lista `messages`), ou seja, deixaremos de enviar o e-mail propriamente:

```
public interface MailService {
    public void send(Message msg);
}

public class MailServiceStub implements MailService {
```

```
    private List<Message> messages = new ArrayList<Message>();

    public void send(Message msg) {
        messages.add(msg);
    }

    public int numberSent() {
        return messages.size();
    }
}
```

Com nosso stub, podemos usar a verificação de estado no teste:

```
class OrderStateTester {

 public void testOrderSendsMailIfUnfilled() {
    Order order = new Order(TALISKER, 51);
    MailServiceStub mailer = new MailServiceStub();

    order.setMailer(mailer);
    order.fill(warehouse);

    assertEquals(1, mailer.numberSent());
  }
}
```

A verificação de estado ocorre através da asserção e utilização do método `numberSent` do nosso stub.

Usando mocks, nosso teste ficará um pouco diferente:

```
class OrderInteractionTester {
    // ...
    public void testOrderSendsMailIfUnfilled() {
        Order order = new Order(TALISKER, 51);

        Mock warehouse = mock(Warehouse.class);
        Mock mailer = mock(MailService.class);

        order.setMailer((MailService) mailer.proxy());

        mailer.expects(once()).method("send");
        warehouse.expects(once()).method("hasInventory")
            .withAnyArguments()
            .will(returnValue(false));
```

```
            order.fill((Warehouse) warehouse.proxy());
        }
    }
```

Lembra que falamos que os mocks tem expectativas sobre o jeito que ele deve ser chamado? Aqui criamos as expectativas (espera-se que o método `mailer.send` deve ser chamado ao menos 1 vez e o método `warehouse.hasInventory` seja chamado ao menos uma vez, com qualquer argumento e retornará `false` quando chamado. Se por ventura alguma dessas expectativas não forem atendidas, nosso teste falhará.

Em ambos os casos usamos *Test Doubles* ao invés do nosso serviço de e-mail (evitando o estresse externo). A principal diferença é que com stub utilizamos a **verificação por estado** e com o mock usamos a **verificação do comportamento**.

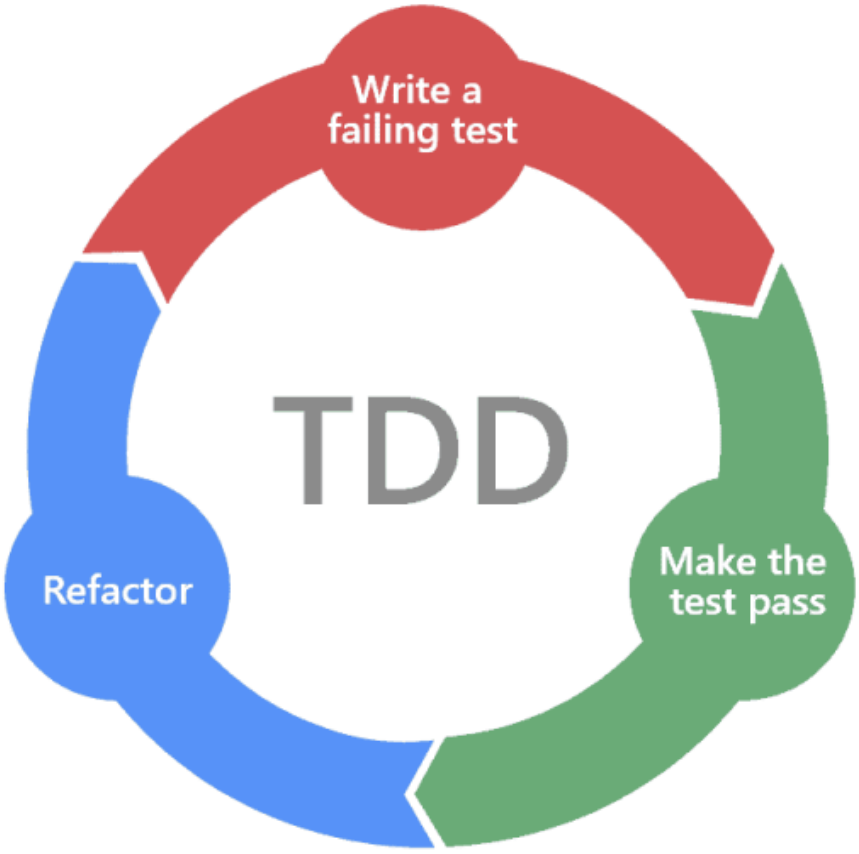
TDD: Test Driven Development

Agora que tivemos uma introdução ao universo dos testes unitários, vamos conhecer um método muito comum atualmente para a construção de testes unitários: o TDD!

TDD é uma sigla para "*Test Driven Development*", em português "Desenvolvimento Orientado à Testes. A técnica foi criada/descoberta em 2003 pelo engenheiro de software americano Kent Beck (um dos pais do *eXtreme Programming* ou simplesmente, XP) e declara que o TDD é um método que encoraja designs de código simples e inspira confiança.

O método se baseia em ciclos ou etapas de desenvolvimento que visam que uma implementação permita que um teste tenha um resultado positivo. Confuso não?

Calma que eu explico!



> Fonte: https://www.rcelebrone.com/2021/03/podcast-teste-unitario-automatizado-tdd.html

Como mencionei, a ideia do TDD é que você trabalhe em ciclos. Iniciando a partir da escrita de um teste para a sua solução. O TDD é composto em 3 ciclos, aplicados na seguinte ordem:

- **Red**: escreva um teste que irá falhar. Isso mesmo! Pense em como você teria um teste caso o seu código estivesse implementado.
- **Green**: a partir disso, implemente o código que satisfaça as condições do seu teste. O intuito nessa fase é que o código implementado, faça com que o teste criado no ciclo anterior passe.
- **Refactor**: na fase anterior, você precisava apenas fazer com que o teste passasse, nesse ciclo você deve focar em melhorar os pontos que não eram uma preocupação: legibilidade, reuso, duplicidade. Deixando-o funcional, e mais limpo. Consequentemente (e é comum), o seu teste criado no primeiro ciclo, pode vir a falhar com a refatoração. No próximo ciclo, você deve fazer com que o teste passe.

O ciclos são iterativos até que os testes e a implementação satisfaça as especificidades solução.

TDD vs Escrever o teste depois

Ressaltando que o TDD é uma metodologia para desenvolvimento e escrita de código, você deve estar se perguntando: "Po, mas o que eu ganho escrevendo o teste antes? Será que vale a pena?"

O que irá manter a qualidade do nosso software será a quantidade de testes produzidos durante a prática do TDD, como qualquer outra metodologia, você só conseguirá ver os reais ganhos colocando-a em prática, é no dia-a-dia que isso acaba ficando mais claro.

Uma das principais vantagens do uso de TDD é que o desenvolvedor acabar tendo mais feedback dos testes, e a quantidades de feedbacks também é maior!

Pense comigo: é melhor implementar testes em um sistema "legado" (onde a quantidade de código é muito maior para ser testado) ou aplicar TDD no inicio do projeto onde a base de código ainda é pequena? Quanto maior é a quantidade de código escrito, maior será o **custo da mudança**. Ao aplicar o TDD, o desenvolvedor recebe o feedback em um momento onde as mudanças ainda são baratas!

Análise de Qualidade de Software

Falamos muito sobre testes e qualidade de software, mas mais do que garantir, como podemos mensurar e analisar como anda a qualidade do nosso código-fonte?

Existem ferramentas poderosíssimas que podem nos auxiliar à deixar um código mais limpo e mais seguro para ir para produção. É comum as empresas adotarem táticas para evitar que um projeto com pouca qualidade (isso envolve segurança e todos os outros quesitos mencionados na introdução desse artigo) seja implantado no ambiente produtivo. Afinal é muito melhor prevenir do que remediar.

Uma excelente ferramenta utilizada amplamente no mercado é o SonarQube!

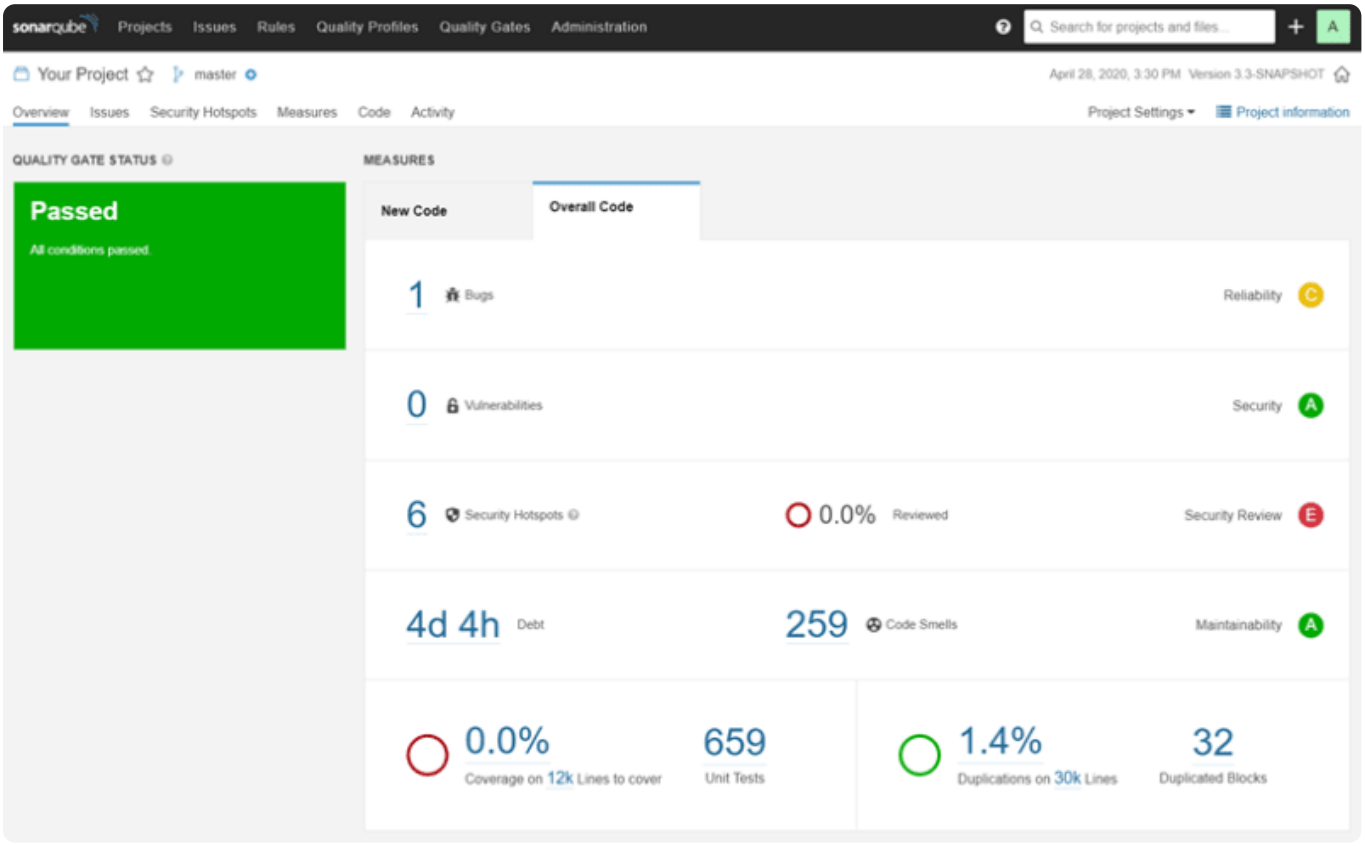
SonarQube

O SonarQube é uma ferramenta de revisão automática de código para detectar bugs, vulnerabilidades, falta de cobertura de teste e *code smells* em seu código. Ele pode ser integrado em esteiras de CI/CD para permitir a inspeção contínua de código em todas as *branchs* do projeto e *pull requests*.

O fluxo de análise do projeto é baseado no seguinte:

- O desenvolvedor abre um Pull Request com o código novo criado e testado.

- Num workflow de CI/CD, a pipeline de CI é acionada a partir do PR criado.
- A pipeline de CI executa os comandos do projeto para realizar a execução dos testes e extrair a cobertura do código.
- A pipeline de CI envia os metadados coletados para o SonarQube a partir de uma integração.
- O SonarQube recebe os metadados e disponibiliza uma dashboard com um overview completo sobre o seu projeto.



Existem diversas outras ferramentas e serviços (como Coveralls e Codecov), cabe você avaliar o que faz mais sentido utilizar no momento.

Finalizando...

Bem, é isso, por hoje, é só!

Quero te agradecer por chegar até aqui, e queria lhe pedir também para me encaminhar as suas dúvidas, comentários, críticas, correções ou sugestões sobre a publicação.

Deixe seu ❤️ se gostou ou um 🦄 se esse post te ajudou de alguma maneira! Não se esqueça de ver os posts anteriores e me siga para maaaais conteúdos.

Até!

Referências e Links

- Mocks Aren't Stubs
- Test Driven Development: TDD Simples e Prático
- Afinal, o que é TDD?
- Kent Beck - Wikipedia
- Extreme programming - Wikipedia
- SonarQube Documentation
- Code Smells
- O que é CI/CD?
- Abordagem de testes
- AT Talks: Triângulos de Teste, Não Mais

Top comments (12)

Jarod Mateus de Sousa Cavalcante • 11 de mar. de 22

Muito bom

William Queiroz • 11 de mar. de 22

Vlwww Jarod! Muito obrigado! o/

Mauro de Carvalho • 12 de mar. de 22

Parabéns pelo post, William! Sensacional, muito claro e direto ao ponto!

Eduardo Quintanilha • 11 de mar. de 22

Testes devem ser tão importantes quanto feature. Muito bom o artigo, William!

William Queiroz • 11 de mar. de 22

Exatamente Eduardo! Muitíssimo obrigado!

Mateus Patricio • 11 de mar. de 22 • Edited

Nossa, está excelente, por coincidência estou em um curso (Nubank-Alura) que está abordando esse tópico neste momento e sem sombras de dúvidas, esse artigo foi muito útil.

William Queiroz • 11 de mar. de 22

Fico muito feliz de poder ajudar Mateus! Bons estudos, uma abraço!

jpmoreira-ti • 8 de jun. de 22

Sensacional o conteúdo. Parabéns William.

Andrade Sampaio • 20 de abr. de 22

Parabéns pelo artigo! Ficou top.

Arthur Conrado de Lima • 13 de mar. de 22

Parabéns pelo artigo! Muito muito bommmm 👏👏

freitaseverson • 11 de mar. de 22

Excelente artigo! Conteúdo muito massa desde a teoria até a aplicação prática dos tipos e técnicas de teste.

William Queiroz • 14 de mar. de 22

Vlw @freitaseverson!! Tamo junto! Obrigado pelo apoio!

Code of Conduct • Report abuse

arm Arm PROMOTED

Learn How to Develop a Creative Mindset for Software Development

👦🏽 Tap into your child-inspired creativity.
🤝 Embrace fearless problem-solving.
💡 Create ground-breaking solutions.
🔑 Level up your design and development skills.
🏆 Learn from an award-winning engineer and innovator.

Register Now

William Queiroz

Salvando o mundo quando não estou jogando video-game 👾

LOCATION
São Paulo, Brazil

EDUCATION
Graduado em Análise e Desenvolvimento de Sistemas, MBA em Engenharia de Software

WORK
Sr. Software Engineer at Neon

JOINED
19 de nov. de 2019

More from William Queiroz

Drops #05: Meu guia para você aprender Go!
#go #braziliandevs #learning #beginners

Drops #04: Desmistificando ponteiros no Golang!
#go #braziliandevs

Configurando e publicando aplicações NodeJS no Nexus Repository Manager 3
#braziliandevs #node #npm

EthicalAds Reach backend, frontend, DataSci, or DevOps engineers with a content-targeted network

Ads by EthicalAds