

# Treinamento de Testes

#Concrete/Pessoal

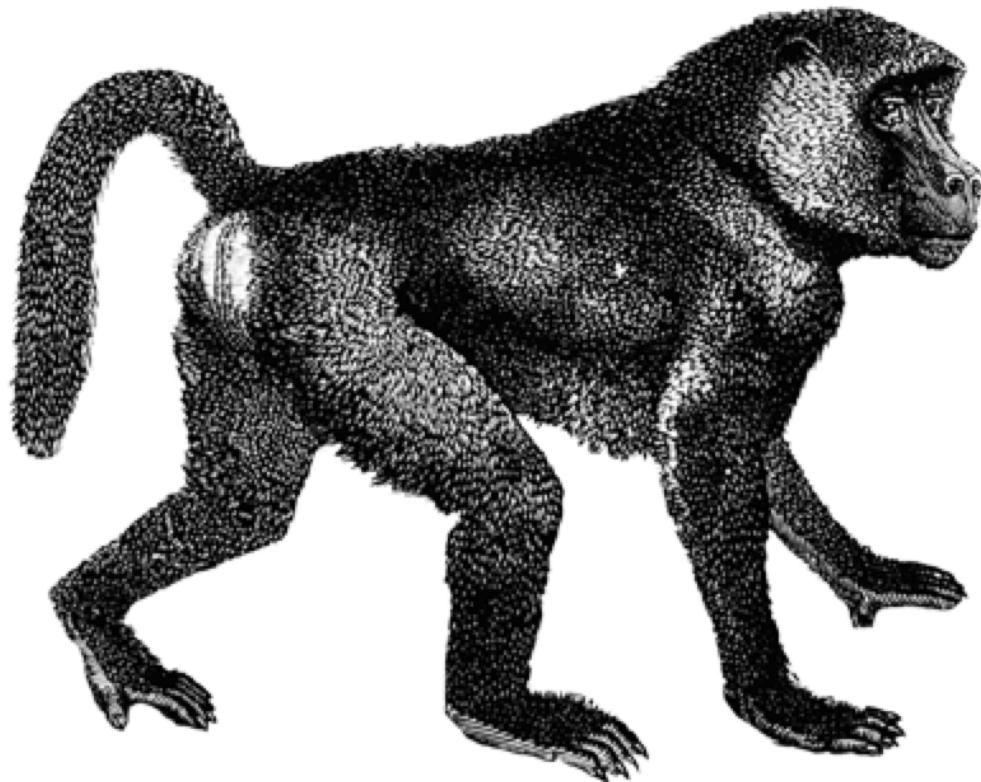
## O que é um teste

O que é um teste? Se olharmos em um dicionário iremos encontrar a definição de que um teste é qualquer meio para verificar ou testar a qualidade ou a veracidade de algo; prova, exame, verificação. Essa definição é legal, mas é recursiva, visto que para definir teste usamos a palavra testar. Mas de modo geral é uma verificação afim de garantir qualidade. Essa descrição muda pouco no contexto de engenharia de software, onde a verificação é feita em cima do código produzido afim de garantir qualidade do sistema.

## Frases ditas do porquê não testar

---

*Because Testing Sux*



# Excuses for Not Testing Software

*The Experts Guide*

**O RLY?**

*James Jeffery*

É comum muitos desenvolvedores não gostarem de testar ou acharem testes algo que não é muito útil. O que gera algumas frases bem conhecidas como:

Meu código funciona quando eu uso, por que eu testaria?

Não vejo beneficio em escrever testes.

Até App com testes tem bugs.

Não sei como, nem o que testar.

## Provas de que isso é errado

Vendo essas frases é possível até entender o porquê das pessoas não gostarem de escrever testes, mas será que de fato são pontos validos?

Afinal, como a definição pressupõe, um teste deve garantir qualidade, não testar é abrir mão de qualidade.

Por exemplo:

"Meu código funciona quando eu uso, por que eu testaria?" Essa frase até faz sentido, afinal você escreveu, você testou utilizando, mas o que garante que você validou todos os cenários? O que garante que esse código não irá quebrar se alguém mexer nele no futuro, ou mexer em algum código que usa ele?

"Não vejo beneficio em escrever testes." Essa frase é comum em projetos menores ou quando desenvolvedores estão começando a escrever testes, afinal, o que é essa qualidade que o teste esta garantindo? A ideia é que ao escrever testes, o que será entregue para o cliente foi validado em todos os cenários. Além disso, que no futuro, ao manter esse código ou atualizar ele saberemos que não estamos modificando o comportamento esperado dele. Desse modo, o que garantimos é que o produto foi de fato validado e é escalável.

"Até App com testes tem bugs." Essa é uma verdade dura que pode desaninar as pessoas de testar, mas afinal, quem escreve os testes são humanos, humanos erram. Por isso, por mais que o ideal seja cobrir todos os cenários possíveis, tem sempre aquele usuário que vai nos surpreender e achar um que não foi pensado. Mas isso não pode desaninar, pelo contrário, deve servir como motivação para evoluirmos e pensarmos cada vez mais em outros cenários e garantir a qualidade do projeto inteiro.

"Não sei como, nem o que testar." Essa é comum no começo, não é trivial começar a testar, teste por si só é algo que não é simples, afinal, é escrever algo que testa algo que você vai escrever. Mas não se preocupe, depois de ler esse artigo, isso irá mudar.

## Por que testar?

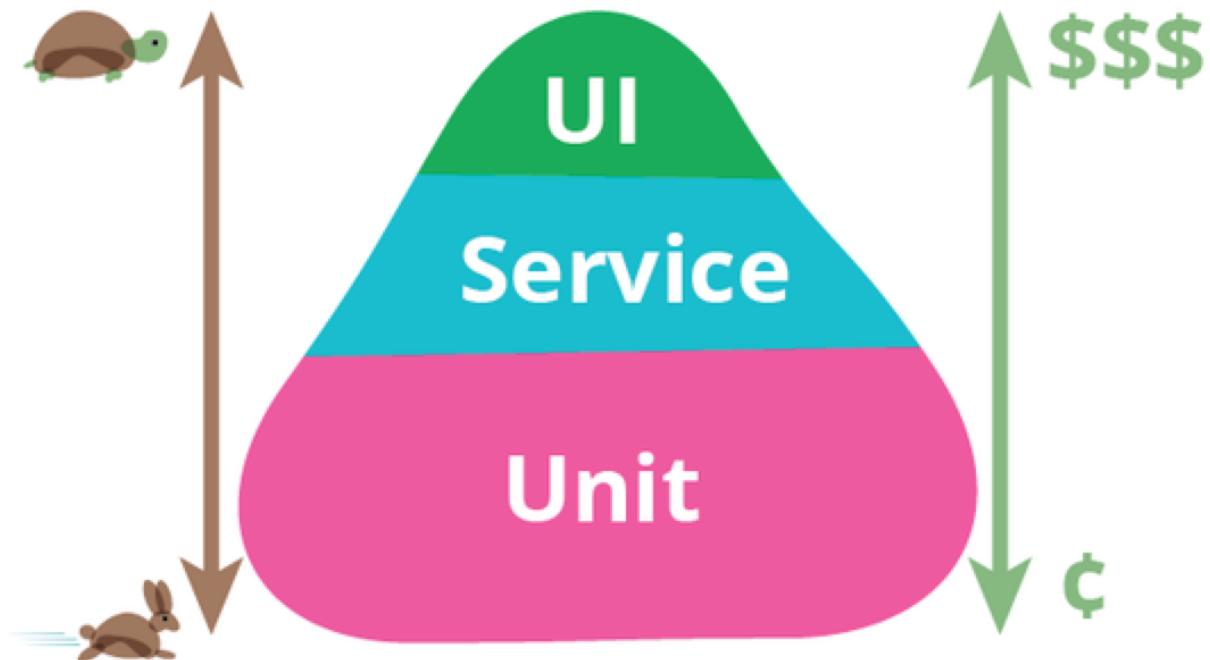
No desenvolvimento a criação de testes é essencial, pois através de testes garantimos qualidade, uma vez que um código testado é um código mais seguro para ir para produção; garantimos também que refatorações – que não tem o objetivo de modificar o comportamento do sistema – após serem feitas não quebrem o sistema; E além disso documenta o código, pois para entender o comportamento de alguma parte do sistema ou como utilizar ele basta olhar seu teste, pois ele irá mostrar como usar aquele código e o que ela faz nos seus possíveis cenários. No contexto de testes unitários ele serve também como uma boa forma de encontrar problemas antes de partir para testes mais lentos, como os de integração ou de UI.

## O que testar?

Existem diversas discussões acerca do que se deve testar, algumas dizem que toda linha de código deve ser testada, outras excluem algumas camadas. De modo geral, deve-se testar tudo que seja importante para o funcionamento de um sistema. Ou seja, tudo aquilo que em algum momento pode impactar o usuário final. O que eventualmente acontece dentro dos projetos é que tudo será testado, seja por você no desenvolvimento ou pelo QA na fase de validações.

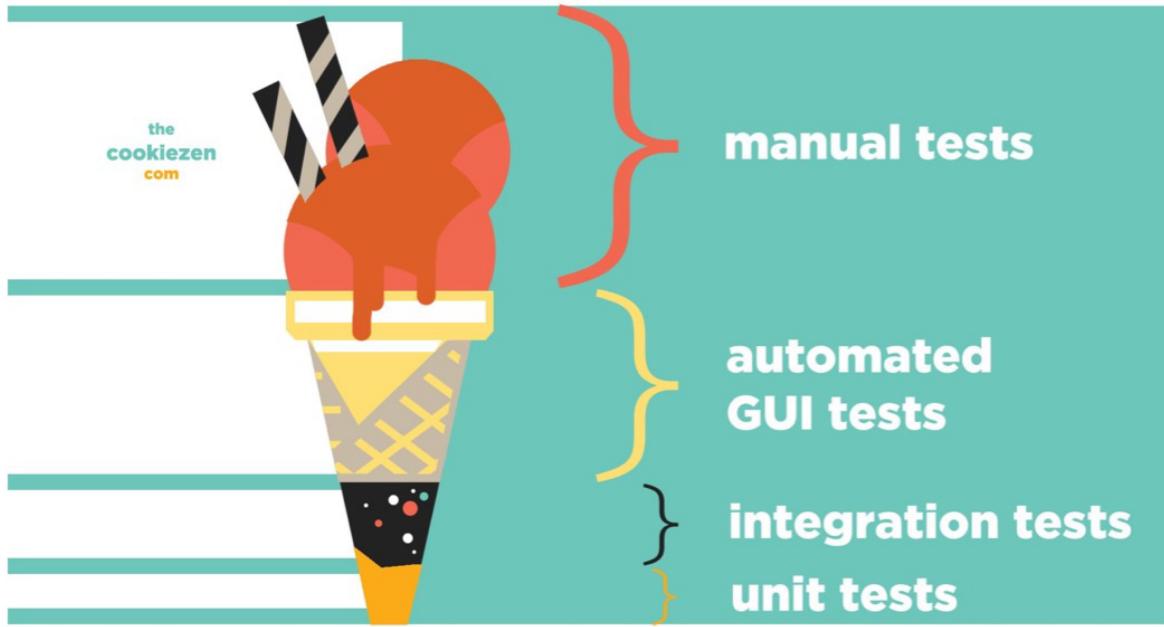
## Pirâmide de testes

Antes de começar a de fato escrever testes, vamos entender melhor o que é um teste na engenharia de software. Uma boa forma de entender é através da pirâmide de testes, um conceito introduzido por **Mike Cohn**, um grande unheiro de software que separou os testes em três camadas.



Segundo Mike Cohn, uma aplicação bem testada deve seguir a estrutura dessa pirâmide, onde no topo, em menor quantidade temos os testes de UI, também chamados de end-to-end (e2e) testes, esses testes estão em menor quantidade por serem caros por demorarem para serem escritos e são lerdos, pois eles inicializam a aplicação e rodam ela, simulando de fato um usuário. No meio temos os testes de integração, também chamados de testes de serviço, neles são testadas como as partes do sistema se comunicam, por exemplo validando contratos de APIs ou pequenos fluxos. Na base temos os testes que devem estar em maior quantidade, os testes unitários, esses são testes rápidos por testarem pequenas partes e além disso estão em maior quantidade por serem baratos, pois deveria ser rápido criar esses tipos de teste. Outro detalhe importante é que quanto mais subimos na pirâmide menos os testes precisam ser isolados, por exemplo um teste unitário nunca deve acessar uma API, enquanto um teste de UI, por simular o comportamento real deve com certeza fazer as chamadas de API.

Isso seria o ideal, assim temos um aplicação bem testada e com testes bem definidos. No entanto, com datas, pressão e outros fatores externos essa pirâmide pode acabar se invertendo e chegando o ao modelo de sorvete.



Nesse modelo, temos as mesmas características, no entanto a quantidade se inverte e ainda no topo temos os testes manuais, onde uma pessoa testa de fato. Esse modelo é considerado um *anti pattern* uma vez que com ele realizar um teste regressivo – onde roda-se todos os testes – demora muito e é muito custoso, no entanto é cômodo, uma vez que geralmente cabe ao QA fazer os testes acima dos unitários, fazendo com que o Dev apenas faça alguns testes unitários.

## Plano de testes

Agora que temos uma melhor ideia dos tipos de teste, entra um momento muito importante, a integração Dev e QA. Em modelos mais tradicionais essa integração é feita por um plano de testes, um documento onde é descrito cada caso possível e as estratégias e recursos que serão usados para fazer os testes desses cenários. Mas, assim como modelos tradicionais têm seus problemas com relação a ambientes caóticos, o plano de testes também tem os seus, com isso, no modelos agéis, temos o que chamamos de documentação viva.

A documentação viva é similar ao plano de teste com a diferença que ela está em constante evolução, a cada mudança ela é atualizada, adicionando, removendo ou atualizando os cenários e as estratégias que serão usadas para testar esses cenários.

## Teste unitário

Agora que temos uma noção dos tipos de testes e como definimos quais serão feitos

irei aprofundar no teste unitários, visto que esses serão os que você escreverá em maior quantidade.

## O que é?

O teste unitário é um código que testa a menor parte testável de um sistema, a chamada *unit*. Em linguagens orientadas a objetos como o Swift a *unit* costuma ser uma Classe. Por outro lado, em linguagens funcionais – lembrando que o Swift pode ter esse comportamento – a *unit* costuma ser uma função.

Esses testes quando agrupados dentro do contexto que estão se propondo a testar são chamados de suíte de testes.

## Características de um teste unitário

De modo geral uma suíte de testes deve seguir as seguintes características para ser considerada boa:

- Deve ser **isolada**: os testes não devem interferir em outros testes, assim como não devem depender de dados ou ambientes externos – como APIs e Bancos de Dados.
- Deve ser **rápida**: um teste deve ser capaz de rodar em segundos, entretanto isso não necessariamente quer dizer que a execução de todas as suítes de teste será rápida, uma vez que os tempos são somados. Porém, é necessário que o teste seja rápido para que a execução completa de todos eles não demore muito.
- Deve ser **coesa**: como a definição pressupõe um teste deve testar apenas a menor parte do sistema. Um teste que testa diversas partes fica complexo e com baixa coesão.

## Conceitos

- **SUT or OUT**: sistema ou objeto que se está testando.
- **Collaborator**: sistemas ou objetos auxiliares para o teste.

## Como escrever (XCTest)

De modo geral todo teste unitário passa por 3 fases, alguns lugares se referem a elas como AAA (Arrange, Act e Assert) e outros como given, when, it.

No iOS a Apple nos disponibiliza o framework **XCTest** para escrever os testes

unitários.

```
import XCTest
@testable import ModuleExempleUnderTest

class ExampleTestCase: XCTestCase {

    var sut: SomeObject!

    override class func setUp() {
        super.setUp()
        // Put global setup code here. This method is called before the
        invocation of suite
    }

    override func setUp() {
        super.setUp()
        // Put setup code here. This method is called before the
        invocation of each test method in the class.
    }

    override func setUp() throws {
        super.setUp()
        // Put setup code here. This method is called before the
        invocation of each test method in the class.
    }

    override class func tearDown() {
        // Put teardown code here. This method is called after the
        invocation of suite
        super.tearDown()
    }

    override func tearDown() {
        // Put teardown code here. This method is called after the
        invocation of each test method in the class.
    }
}
```

```

super.tearDown()

}

override func tearDown() throws {
    // Put teardown code here. This method is called after the
    invocation of each test method in the class.
    super.tearDown()
}

func testExample() {
    // This is an example of a functional test case.
    // Use XCTAssert and related functions to verify your tests
produce the correct results.
    XCTAssert(sut.x)
}

func testPerformanceExample() {
    // This is an example of a performance test case.
    self.measure {
        // Put the code you want to measure the time of here.
        sut.performCode()
    }
}

}

```

Analizando a anatomia de um teste vemos o `import` do framework de testes assim como o do modulo que será testado, esse deve conter a marcação de `@testable`. Após isso temos nossa classe que será a suite de testes que deve herdar de `XCTestCase` e ai começamos a parte de *Arrange* ou de *Given*, nela devemos preparar o ambiente para o teste, inicializando variáveis que os testes usarão e configurando o ambiente, por exemplo configurando as chamadas para serem todas isoladas, fazemos isso através do `setUp`, podendo ser algo único através do `setUp` de `class` ou para cada teste com o `setUp` padrão. É importante notar que temos o `tearDown` e nele devemos resetar todas as configurações feitas no `setUp` para garantir que uma próxima suite tenha o ambiente limpo.

Após realizar a preparação chegamos ao teste em si. No XCTest para criar um teste basta criar um função que tenha o seu começo com `test`. Dentro do teste costuma-se realizar o *Act* e o *Assert*, ou seja, nele realizamos alguma ação com o objeto que se está testando e depois validamos se realmente aconteceu o esperado.

## Algumas dicas

Por mais que seja necessária a utilização do *force* na declaração das variáveis, não é aconselhável seu uso em nenhum momento do teste, assim como o *force cast*. Sempre que possível evitar usar o *force* dentro dos testes pois se falhar ele irá gerar um Crash da suite de testes, o que é ruim pelos seguintes motivos:

- **Debug:** Um Crash é muito pior para se debugar o real motivo dele do que apenas um teste falhando, pois a mensagem não é sempre amigável como em um *Assert*
- **CI:** Se um teste crash na CI ele irá interromper a Pipeline dela por completo, e isso nos leva ao próximo motivo do porque é ruim.
- **Quantidade de erros:** Se um teste crashar ele interrompe a execução, o que significa que mesmo que você arrume ele, pode ser que outros testes crashem também ou falhem sem você saber até arrumar ele. Se evitar o uso do *force*, é possível saber todos os pontos de falha de testes

Para se evitar isso devemos seguir a mesma lógica de um código de produção, usando `if let` ou `guard let` com a adição de um `XCTFail`.

O `XCTFail` possibilita falhar o teste e ainda adicionar uma mensagem mais amigável para realizar o debug.

```
func testPropertie() {  
    guard let propertie = sut.someOptionalPropertie else {  
        XCTFail("Propertie not exist")  
        return  
    }  
    XCTAssertEqual(propertie, xpto)  
}
```

Outra estratégia, mais recente, é utilizar o `XCTUnwrap`, que foi disponibilizado a partir do **Xcode 11**. Ele faz basicamente o que o `guard + XCTFail` fazem, de maneira mais

enxuta.

```
func testPropertie() throws {
    let propertie = try XCTUnwrap(sut.someOptinalPropertie)
    XCTAssertEqual(propertie, xpto)
}
```

Para isso é preciso adicionar a palavra reservada `throws` para o teste pois o `XCTUnwrap` lança um erro caso o unwrap não funcione, porém, esse erro é lidado pelo `XCTest` através de um `XCTFail`.

## Testando métodos assíncronos

É comum sempre que temos um objeto de testes que possua funções com *closures* – como camada de rede – que tenhamos que lidar com funções assíncronas. Isso em um primeiro momento pode parecer um problema uma vez que um teste inicia uma thread, executa e finaliza a thread, assim sem esperar um retorno.

Porém, o `XCTest` disponibiliza uma forma de testar essas funções assíncronas de maneira bem tranquila através das `Expectations`. Existem diversas *expectations*, cada uma para um motivo e de modo geral todas servem para testes que saiam do escopo da thread do teste ou que sejam assíncronos.

De modo geral, basta utilizar a `XCTestExpectation` para um teste assíncrono.

```
func testSuccess() {
    let expectation = XCTestExpectation(description: #function)
    var data: Data?
    sut.execute { result in
        if case .success(let dataResponse) = result {
            data = dataResponse
            expectation.fulfill()
        }
    }
    wait(for: [expectation], timeout: 1)
}
```

```
XCTAssertNotNil(data)
}
```

É importante que o Assert não ocorra dentro do bloco assíncrono pois pode gerar falsos positivos, uma vez que caso a thread morra e não seja feito nenhum assert ou fail, o teste é dado como valido.

## Quando testar

Uma discussão muito grande no desenvolvimento é quando se testar. Afinal, devo primeiro fazer meu código e depois testa-lo ou primeiro criar seu teste e depois desenvolver até que o teste passe?

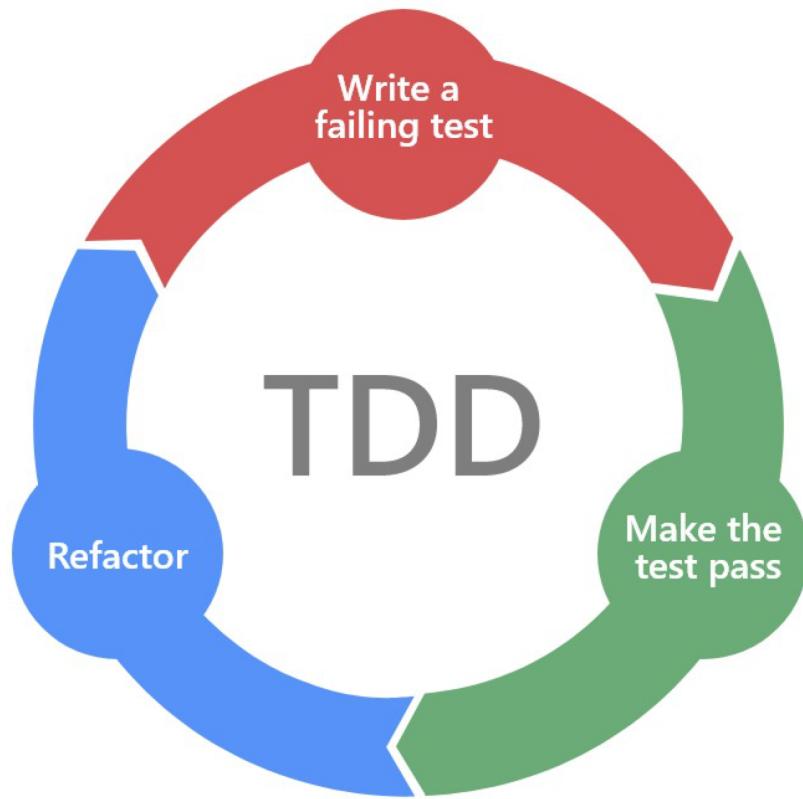
Na maioria dos casos os desenvolvedores preferem primeiro desenvolver e depois criar os testes, pois assim ele sabe exatamente o que precisa ser testado, porém isso abre algumas brechas.

Imagine que você primeiro desenvolveu alguma *feature* dentro do projeto e então irá fazer os testes dela, ao terminar os testes estarão passando e o código pode subir. Agora imagine que no futuro outro desenvolvedor precise mudar uma parte do seu código, mas não o comportamento dele, será que o teste que foi feito irá continuar passando?

A ideia de um teste é apenas quebrar se o comportamento de um sistema mudar, não a implementação dele. Ao entender isso é possível perceber a brecha. Ao escrever o teste se baseando na implementação já feita, é possível que seu teste se torne enviesado a sua implementação e que qualquer alteração nela, mesmo que não mude o comportamento, possa quebrar seu teste.

Esse cenário, no entanto, dificilmente irá acontecer se o teste for feito antes, pois assim o teste estará validando um comportamento e ele apenas irá passar com sucesso quando a implementação de fato tiver o comportamento esperado e caso alguém mude ela, caso não mude o comportamento, o teste não irá quebrar.

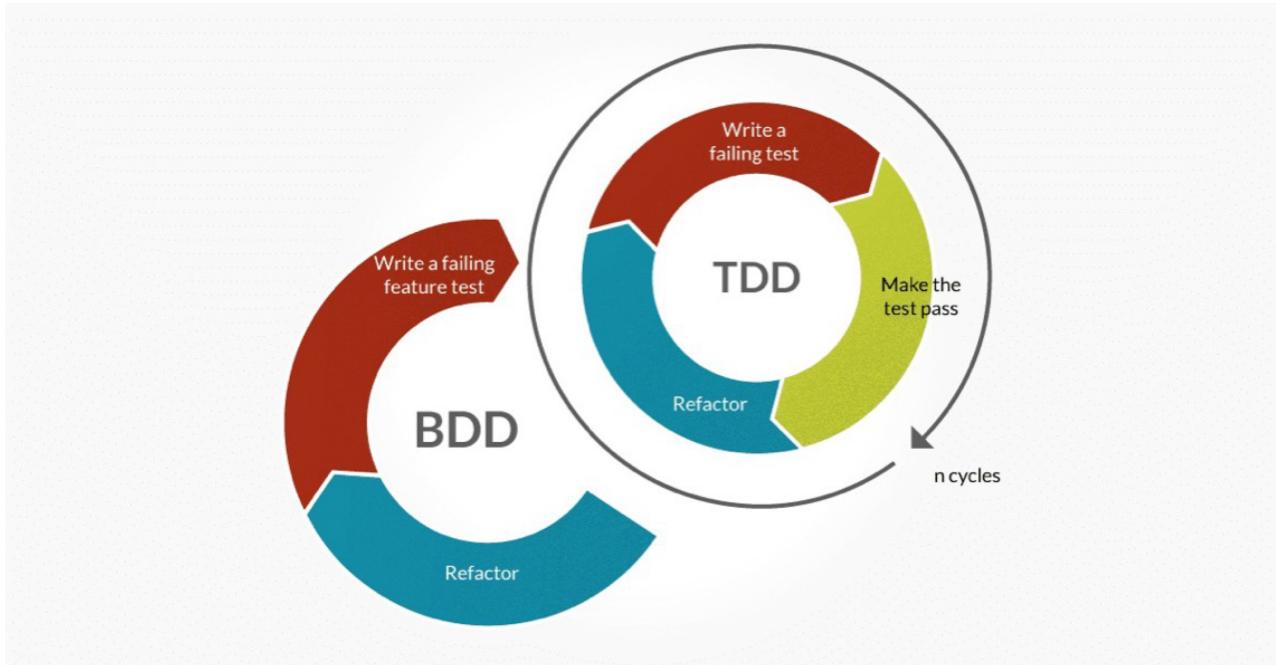
Essa abordagem é muito comum no processo de Test-driven development (TDD).



O TDD é um processo de desenvolvimento do software que tem como premissa a escrita de testes antes do código de produção. O processo começa com a escrita de testes para a feature planejada, nesse momento o teste irá falhar e então o desenvolvedor vai escrever o código até que o teste passe. Nesse momento, com o teste passando o código feito tem o mínimo possível para que a funcionalidade esteja certa, a partir disso o desenvolvedor pode refatorar esse código até chegar em um ponto de qualidade aceitável desde que os testes continuem passando. A ideia desse processo é que ao seguir ele o desenvolvedor irá fazer códigos mais simples e rápidos, uma vez que sabendo que o código chegou ao funcionamento esperado o desenvolvedor não precisa ficar fazendo over engineering do código. Além disso, ele garante que seu teste não tem ligação com a implementação, mas sim com uma funcionalidade.

## Outros Frameworks

O mundo de testes é imenso e existem diversas abordagens que podem ser utilizadas, uma das mais comuns no meio ágil é o Behavior-driven development (BDD).



Assim como no TDD o BDD também parte do pressuposto de escrever testes antes, na verdade o processo incorpora o TDD de certa forma, com a diferença que no BDD pensamos em escrever testes de feature, estamos preocupados com todo um contexto. De modo geral, o BDD preza por testes contextualizados e com fácil compreensão, sendo geralmente escrito utilizando ferramentas descritivas, para que tantos desenvolvedores como outros membros do time possam entender.

No universo do iOS temos dois frameworks terceiros muito conhecidos que disponibilizam maneiras de escrever testes descritivos seguindo um padrão de BDD. Os frameworks são o Quick e o Nimble.

## Nimble

O Nimble é um framework para Swift e Objective-C que se propõe a ajudar na escrita de testes para BDD através de funções de asserção mais descritivas. Tem suporte para CocoaPods, Carthage e Swift Package Manager.

[Repositório do Nimble](#)

```
expect(1 + 1).to(equal(2))
expect(1.2).to(beCloseTo(1.1, within: 0.1))
expect(3) > 2
expect("seahorse").to(contain("sea"))
expect(["Atlantic", "Pacific"]).toNot(contain("Mississippi"))
```

```
expect(ocean.isClean).toEventually(beTruthy())
```

## Quick

O Quick é um framework para Swift e Objective-C que se propõe a ajudar na escrita de testes para BDD. Tem suporte para CocoaPods, Carthage e Swift Package Manager. De modo geral seu funcionamento é atrelado ao Nimble.

[Repositório do Quick](#)

```
import Quick
import Nimble

class TableOfContentsSpec: QuickSpec {
    override func spec() {
        describe("the 'Documentation' directory") {
            it("has everything you need to get started") {
                let sections = Directory("Documentation").sections
                expect(sections).to(contain("Organized Tests with Quick Examples
and Example Groups"))
                expect(sections).to(contain("Installing Quick"))
            }

            context("if it doesn't have what you're looking for") {
                it("needs to be updated") {
                    let you = You(awesome: true)
                    expect{you.submittedAnIssue}.toEventually(beTruthy())
                }
            }
        }
    }
}
```

## Sempre tem algo novo

Como sair de um código não testável para um testável [\(EN\)](#)

- Uma abordagem de testes unitários e instrumentados [\(EN\)](#)
- Test Doubles e isolamento de testes em Swift [\(PT\)](#)
- Anúncio Test Plans WWDC [\(EN\)](#)
- Como utilizar test plans na prática [\(EN\)](#)
- Entendendo o life cycle de um TestCase [\(PT\)](#)
- Lista de artigos relacionados a testes no universo iOS [\(EN\)](#)
- Uma abordagem bem ampla sobre quase todos os tópicos de teste unitários em iOS [\(EN\)](#)
- Como testar uma camada de rede em swift [\(EN\)](#)
- Expandindo o assunto para entender melhor os tipos de teste além do unitário [\(EN\)](#)

| **Matheus de Vasconcelos Moura**