Testando um App nunca testado

Eduardo Sanches Bocato

Sobre mim

- **Primeiro contato com desenvolvimento para iOS**
  - **2010**: Minicurso - "Desenvolvimento iPhone" no INPE

- **Até 2016, iOS era não era o meu foco**
  - Infra, Web, Backend (Java)
- **2016**:
  - Fui contratado pra ser backend, mas pediram pra "quebrar um galho no iOS"

- **2020:** Projeto pessoal - Medium
  - 1 artigo por semana
  - **https://medium.com/@bocat**o

Objetivos

O que eu pretendo mostrar:

- Alguns conceitos e dicas para um atingir testabilidade
- Conceitos importantes para testar um app
- Aplicação dos conceitos introduzidos

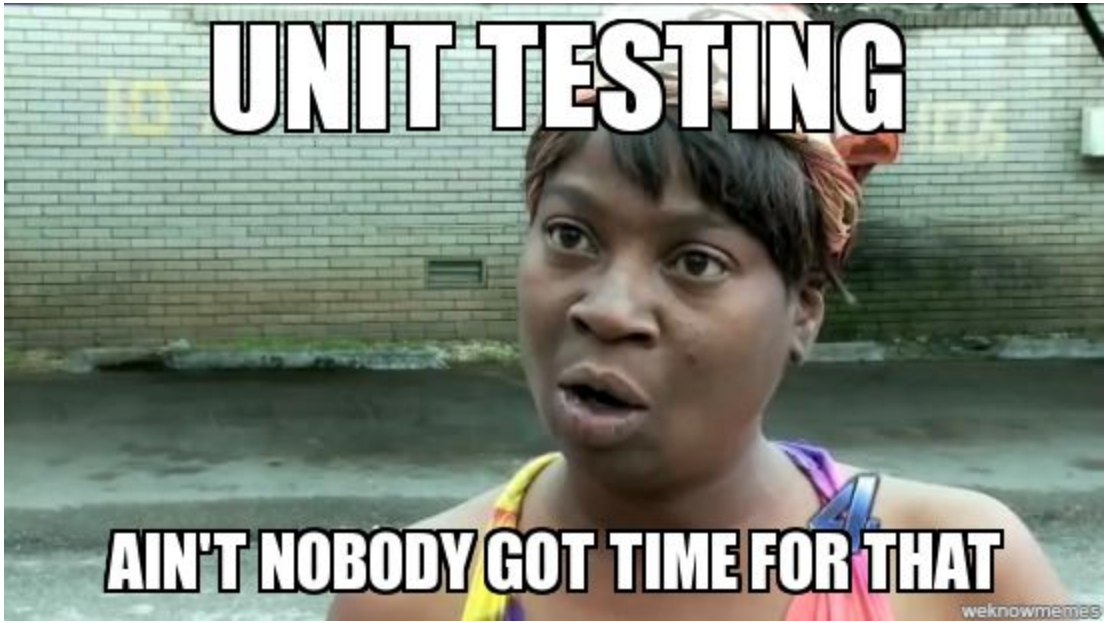
O que eu não estou preocupado:

- Testar o exemplo todo no live code.

A idéia é mostrar o processo, não o resultado final.

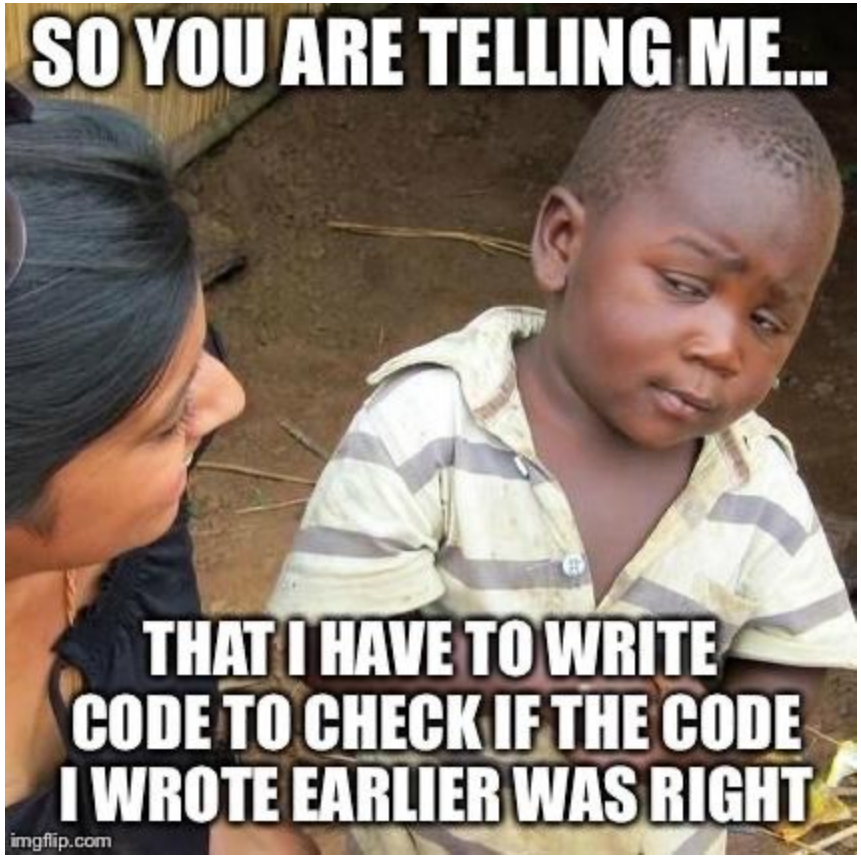
Roteiro

- Porque testar?
- Tipos de injeção de dependência
- Test Doubles
- Boas práticas em testes unitários
- Aplicação dos conceitos

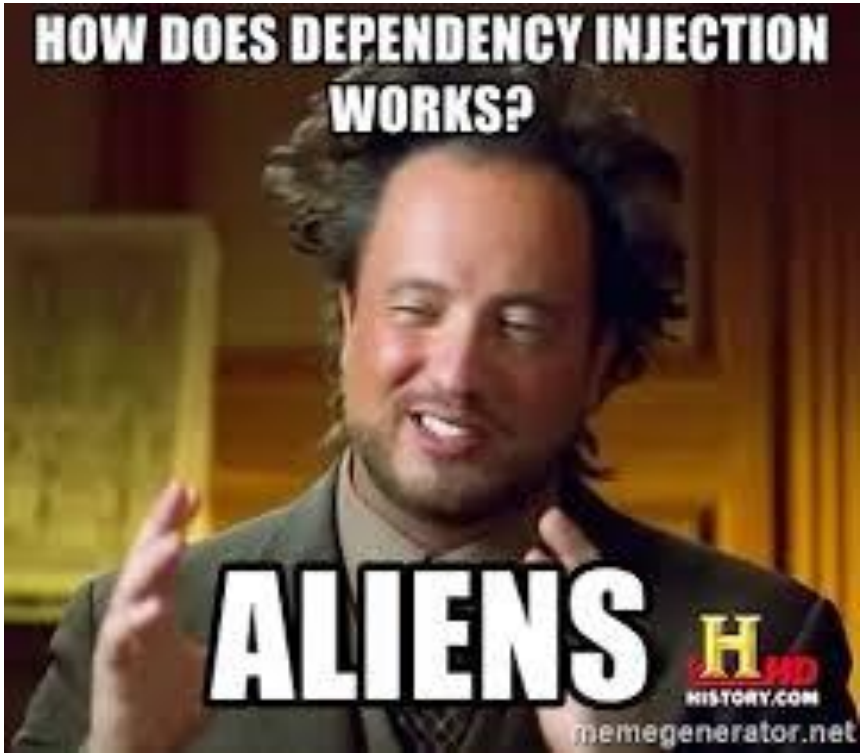

UNIT TESTING
AIN'T NOBODY GOT TIME FOR THAT
weknowmemes

Por que testar?

- **Garantir a qualidade do código**
- Garante corretude do código
- Acelera o processo de desenvolvimento (a médio/longo prazo)
  - Facilita debug
  - Expõe "edge cases"
  - ...
- Bugs são encontrados rapidamente
- Documentação
- Integração Contínua
- ...

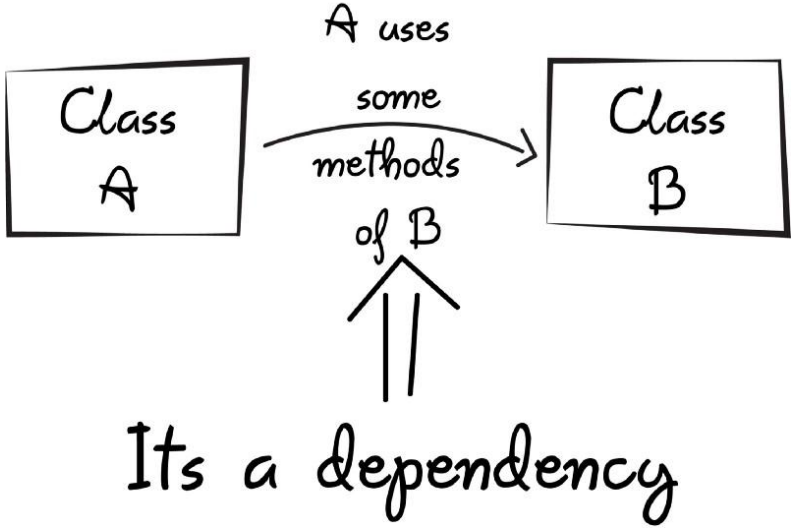

SO YOU ARE TELLING ME...
THAT I HAVE TO WRITE CODE TO CHECK IF THE CODE I WROTE EARLIER WAS RIGHT
imgflip.com

Injeção de Dependência



Injeção de Dependência

- Um objeto provê as dependências para outro, separando a criação da dependência do seu comportamento.
- O objeto que usa a dependência não precisa saber como construí-lo, só o consome.



Injeção de Dependência

Vantagens

- Baixo acoplamento
- Testabilidade
- Reusabilidade
- Legibilidade
- Separação de responsabilidades
- ...

Desvantagens

- Complexidade
  - Setups
  - Mais classes
- Forte acoplamento ao framework de injeção
- ...

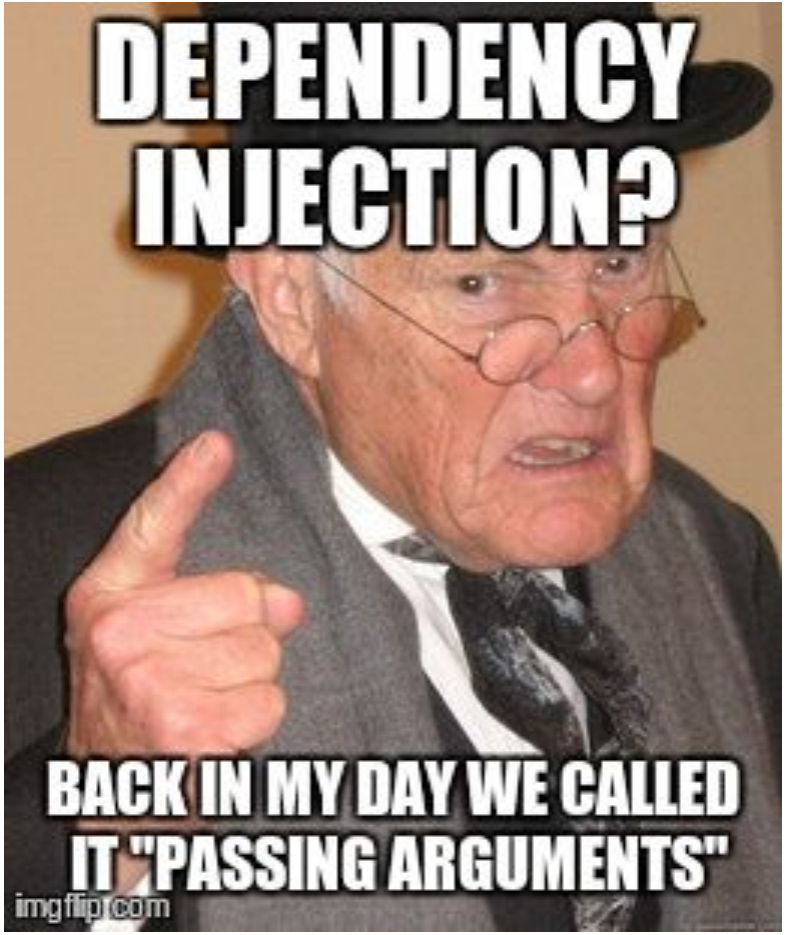
Tipos de Injeção de Dependência

- Initializer Based
- Property Based
- Parameter Based

Initializer Based

As dependências são providas no inicializador do objeto.

```
protocol NetworkProtocol {
    func getData(from url: URL, then: (Result<Data?, Error>) -> Void)
}
final class Network: NetworkProtocol {
    static let shared = Network()
    func getData(from url: URL, then: (Result<Data?, Error>) -> Void) { /* ... */ }
}

struct User { /* ... */ }
final class UserService {

    private let network: NetworkProtocol

    init(network: NetworkProtocol = Network.shared) {
        self.network = network
    }

    func getUserData(then: @escaping (Result<User, Error>) -> Void) {
        network.getData(from: URL(string: "www.data.com/some")!) { result in
            /* ... */
        }
    }

}
```


DEPENDENCY INJECTION?
BACK IN MY DAY WE CALLED IT "PASSING ARGUMENTS"
imgflip.com

Property Based

A dependência é injetada a partir de uma propriedade acessível externamente.

- Uma boa opção quando não se tem controle da inicialização do objeto.

* **Exemplo:** inicializando um ViewController a pela storyboard.

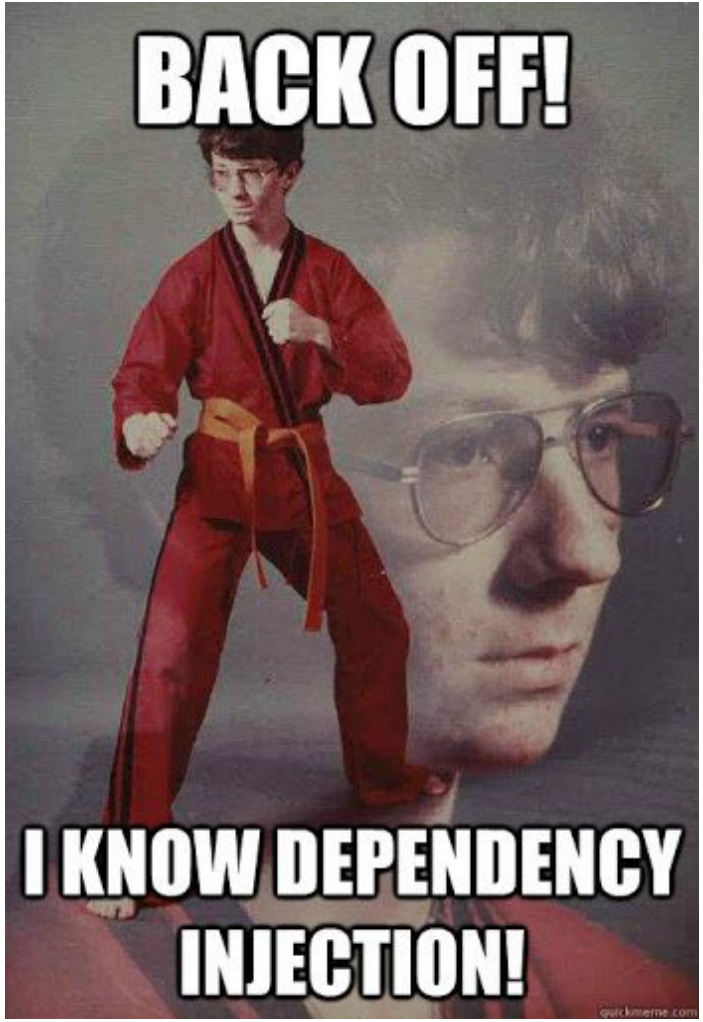
```swift
final class ViewController: UIViewController {

    var network: NetworkProtocol = Network.shared

    required init?(coder aDecoder: NSCoder) {
        fatalError("init(coder:) has not been implemented")
    }

    override func viewDidLoad() {
        super.viewDidLoad()
        loadData()
    }

    func loadData() {
        network.getData(from: URL(string: "www.data.com/some")!) { result in
            /* ... */
        }
    }

}
```

Parameter Based

A dependência é injetada no parâmetro da função.

* Bastante útil quando queremos testar algo legado, sem mudar muita coisa.

```
protocol NetworkProtocol {
    func getData(from url: URL, then: (Result<Data?, Error>) -> Void)
}
final class Network: NetworkProtocol {
    static let shared = Network()
    func getData(from url: URL, then: (Result<Data?, Error>) -> Void) { /* ... */ }
}

extension UIImageView {

    func setImageFromURL(_ url: URL, network: NetworkProtocol = Network.shared) {
        network.getData(from: url) { result in
            guard
                let data = try? result.get(),
                let remoteImage = UIImage(data: data)
            else { return }
            DispatchQueue.main.async { self.image = remoteImage }
        }
    }

}
```


BACK OFF!
I KNOW DEPENDENCY INJECTION!
quickmeme.com

Test Doubles

Test Double é um termo genérico para representar um objeto que substitui o objeto real na hora dos testes.

Tipos:

- **Dummy**
- **Stub**
- **Spy**
- **Fake**
- **Mock**

Dummy

Tem a intenção de "preencher" a dependência, sem afetar os testes.

```
protocol UserServiceProtocol {
    func login(_ user: User, then: (Result<Void, Error>) -> Void)
}

struct UserServiceDummy: UserServiceProtocol {
    func login(_ user: User, then: (Result<Void, Error>) -> Void) {
        // Do nothing.
    }
}
```

Stub

Retorna outputs pré-configurados para as chamadas feitas durante o teste.

```swift
struct Post {
    let title: String
    let text: String
}

protocol PostsServiceProtocol {
    func fetchAll(then: (Result<[Post], Error>) -> Void)
}

final class PostsServiceStub: PostsServiceProtocol {

    var fetchAllResultToBeReturned: Result<[Post], Error> = .success([])
    func fetchAll(then: (Result<[Post], Error>) -> Void) {
        then(fetchAllResultToBeReturned)
    }

}
```

Spy

Guarda informações relevantes sobre execução da dependência, para que possamos inspecioná-las posteriormente.

```swift
protocol SafeStorageProtocol {
    func storeUserData(_ user: User)
}

final class LoginViewModel {

    private let userService: UserServiceProtocol
    private let safeStorage: SafeStorageProtocol

    /* ... */

    func performLoginForUser(_ user: User) {
        userService.login(user) { [weak self] result in
            switch result {
            case .success:
                self?.safeStorage.storeUserData(user)
            case let .failure(error):
                // do something with the error
                debugPrint(error)
            }
        }
    }

}
```

```swift
final class SafeStorageSpy: SafeStorageProtocol {

    private(set) var storeUserDataCalled = false
    private(set) var userPassed: User?
    func storeUserData(_ user: User) {
        storeUserDataCalled = true
        userPassed = user
    }

}
```

Fake

Implementação simplificada da dependência real, que só serve pros propósitos do teste.

- Não poderia ser usado em produção.

```
struct User {
    let name: String
    let password: String
}

protocol UserServiceProtocol {
    func login(_ user: User, then: (Result<Void, Error>) -> Void)
}

final class UserServiceFake: UserServiceProtocol {

    func login(_ user: User, then: (Result<Void, Error>) -> Void) {
        if user.name == "Mock" {
            then(.success(()))
        }
    }

}
```

Mocks

"**Mocks** são objetos pré-programados com definições que especificam quais chamadas eles devem receber o que deve acontecer nesse caso. Eles podem lançar uma exceção caso recebam uma chamada que não era esperada, para garantir que tudo que era esperado ocorreu."
Tradução livre de: https://martinfowler.com/bliki/TestDouble.html

Resumindo:

- Mocks são objetos de teste mais "inteligentes", que podem conter validações, asserções e uma lógica mais complexa relacionada ao teste. Seja esta para simplificar os testes ou preparar algo específico do contexto.

Boas práticas em testes unitários

- Os testes devem ser **rápidos**, ou seja, se existe algo que deixaria seu teste muito lento, considere mockar isso (se for coerente).
- Testes n**ão devem acessar na camada de networking** real.
- Um teste não pode **depender** de outro, nem **afetá-lo**.
- Evite testar várias coisas em um mesmo test case.
- Evite **force unwrapping**, é preferível **falhar** um teste do que causar um **crash**.
- Dê **nomes bem descritivos** às funções de teste, independente do tamanho que ela vai ter no final.
- Use **AAA (A**rrange/**A**ssert/**A**ct**)** ou **Given/When/Then** para organizar suas funções de teste.

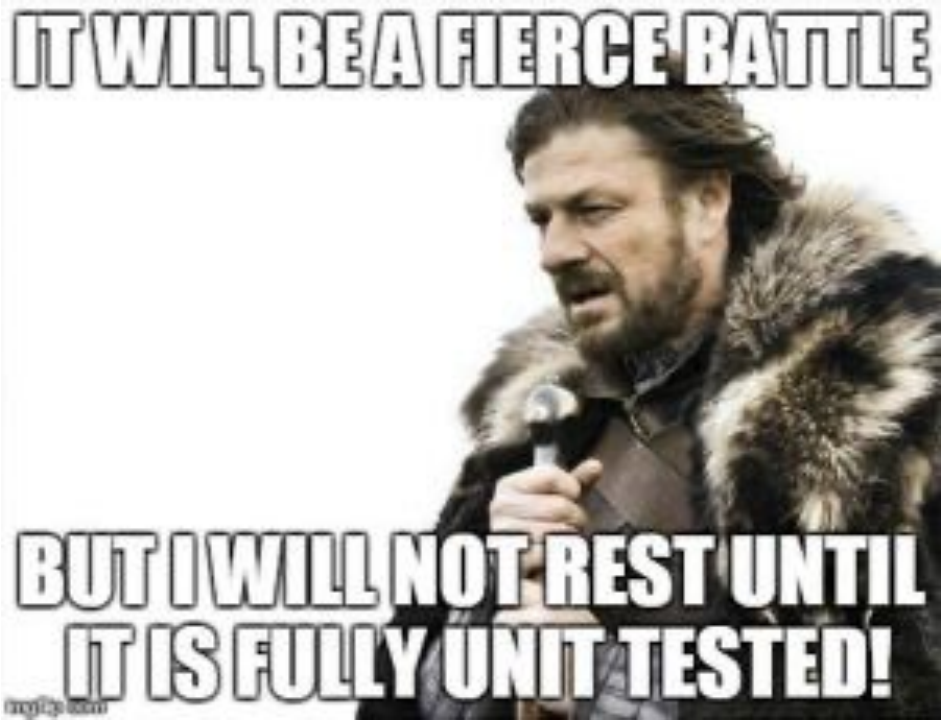
Aplicação dos conceitos

Cenário:

App em "MVC", daquele jeito que estamos bastante acostumados a ver nos legados de uns tempos atrás...

Onde está praticamente tudo acontecendo no ViewController, com alguns singletons, extensões, XIBs e Storyboards.

Aplicação dos conceitos



Aplicação dos conceitos

Parameter-based Injection + Stubs:
test_whenNetworkFails_errorImageShouldBeSet
Initializer-based Injection + Spy:
test_whenViewDidLoad_isCalled_favoritesShouldBeLoaded
Property-based Injection + Fakes + Dummy:
test_when_removeFromFavoriteSucceeds_tableViewShouldChangeNumberOfItems
Initializer-based Injection + Spy
test_whenAddIsCalledForAValidObject_userDefaultsShouldReceiveValue_andSyncronize
Initializer-based Injection + Spy
test_whenAddIsCalledForAValidObject_userDefaultsShouldReceiveValue_andSyncronize
Initializer-based Injection + Stubs + Dummy + Reflection:
test_whenAValidMovieIsSearched_successShouldBeHandled

Perguntas?

Material:

https://github.com/bocato/TestingAnUntestedApp

- Códigos dos Slides: no **Playground**
- App de Exemplo: na branch **master**
- LiveCoding: na branch **live-coding**

Contato:

Twitter: **@dubocato**

LinkedIn: **Eduardo Sanches Bocato**

E-Mail: **dubocato@gmail.com**