

Overview sobre Testes Unitários

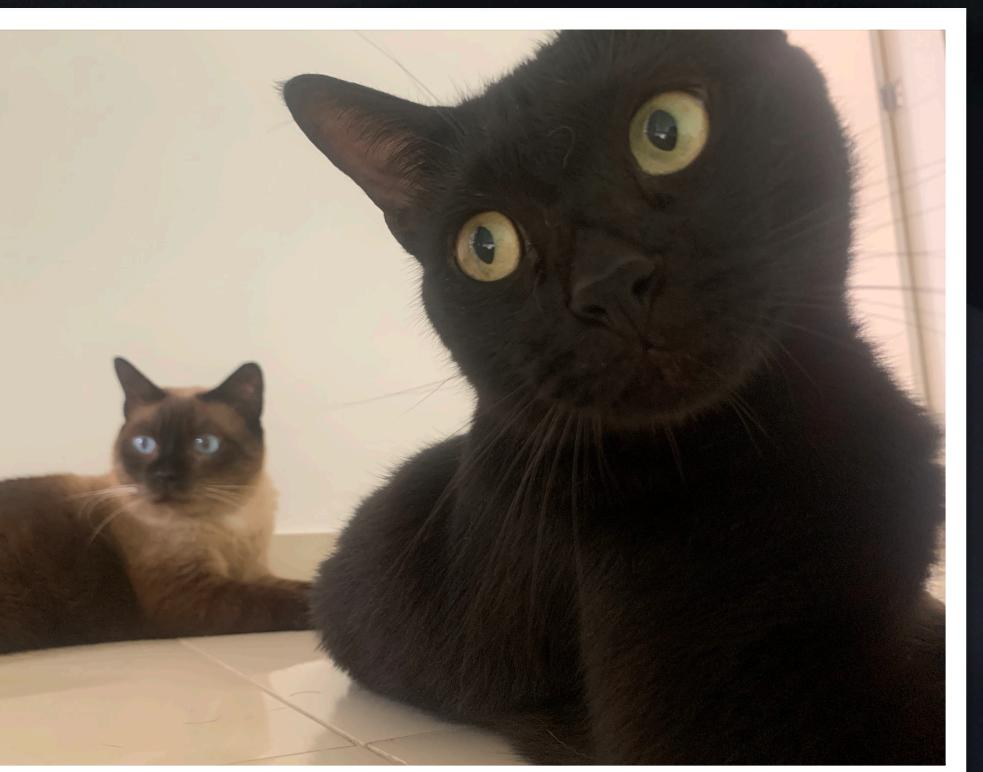
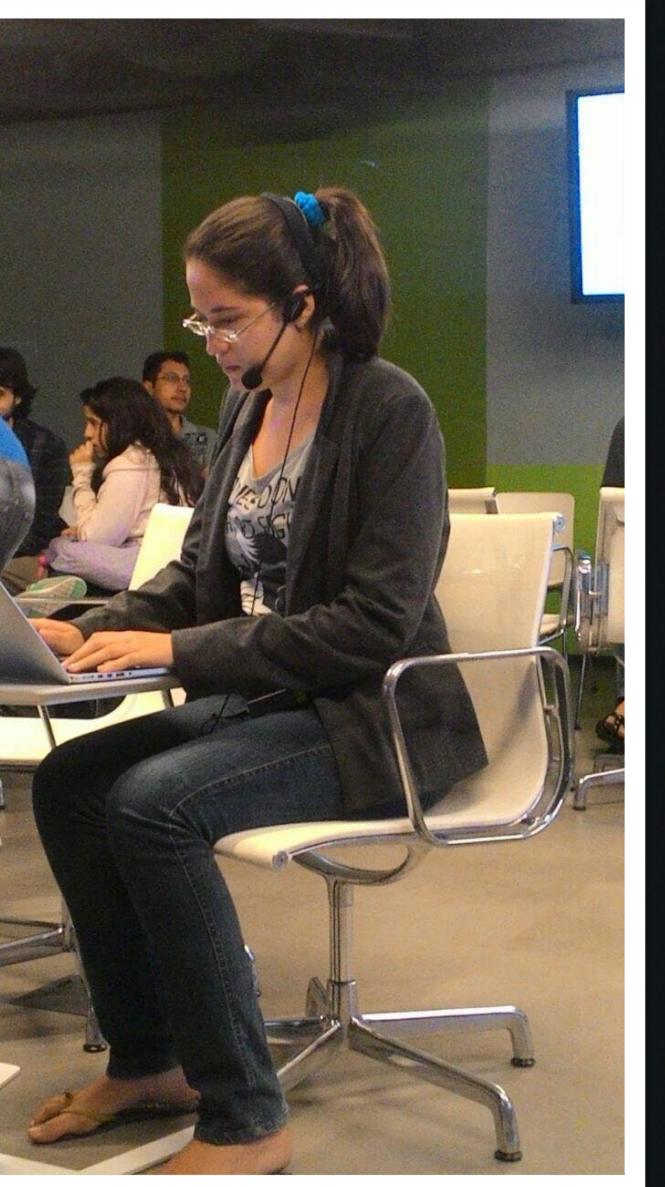


Uma abordagem geral sobre testes

Quem sou eu?

Anne Kariny

- Engenheira da Computação formada em 2019 no IFCE.
- Ex-Apple Developer Academy Alumni, turma de 2016/2017.
- Engenheira de Software iOS no iFood desde 2021.
- Animes, Nintendo, board games, pizza e gatos ❤️



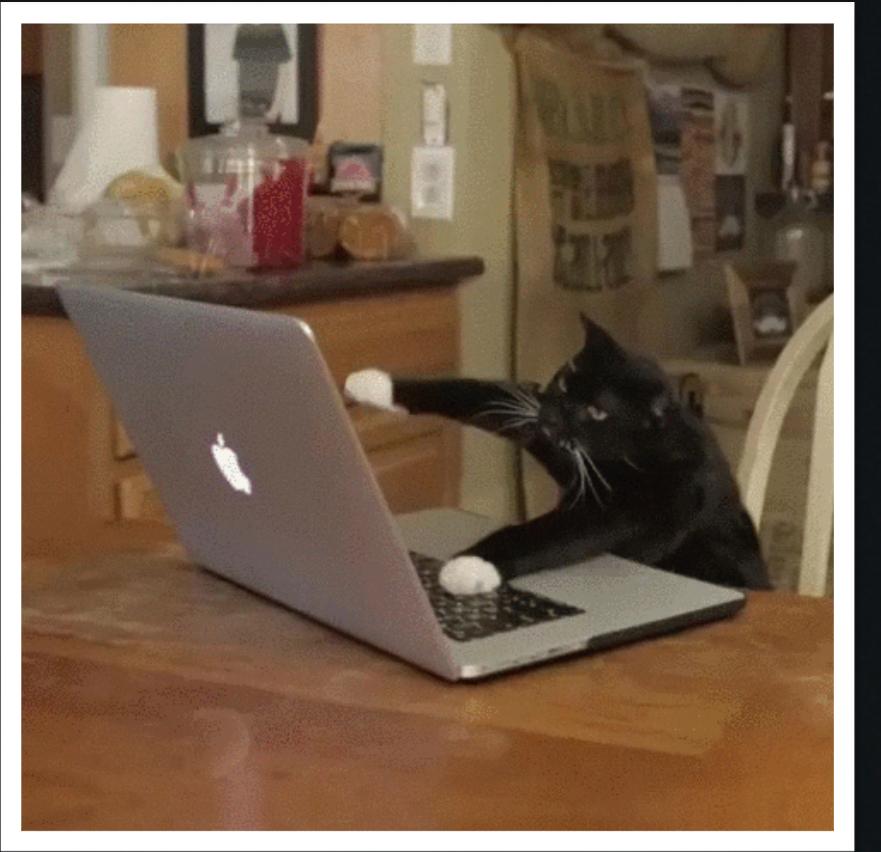
Agenda

1. Introdução: o que são, importância, uso no mercado, etc.
2. Utilizando arquiteturas e conceitos de SOLID ao nosso favor.
3. O que devo testar?
4. Como testar e dicas legais.
5. Hands-on: refatorando uma MassiveViewController para incluir testes.
6. Dúvidas?

Introdução

O que são? Para que servem?

- Forma de garantir que nosso código funciona como o esperado (por nós e pelo nosso cliente).
- Deixar claro os resultados esperados do código em cenários diferentes.
- Nos auxiliar durante refatorações.
- Aprofundar o entendimento sobre a funcionalidade implementada.
- Manter a qualidade do código identificando possíveis regressões (quando associado ao CI/CD).



Introdução

Uso no mercado

- ~Quase~ todas as empresas do mercado pedem conhecimentos sobre testes unitários.
- Uma pessoa desenvolvedora que tem experiência e valoriza a importância dos testes é “bem visto”.

- A importância dos testes pode variar em cada empresa.

Sobre a vaga:

Responsabilidade e atribuições

- Swift e UIKit;
- View Code;
- Gerenciamento de memória;
- SOLID;
- Injeção de dependência;
- Gerenciamento de dependências;
- Padrão de design (Ex:: MVC, Delegate, MVVM, VIP, etc);
- Arquitetura e modularização;
- Cultura de testes e testes unitários;
- Ferramentas de Debug;
- Programação orientada a protocolos;
- Conhecimento do ciclo de vida do aplicativo;

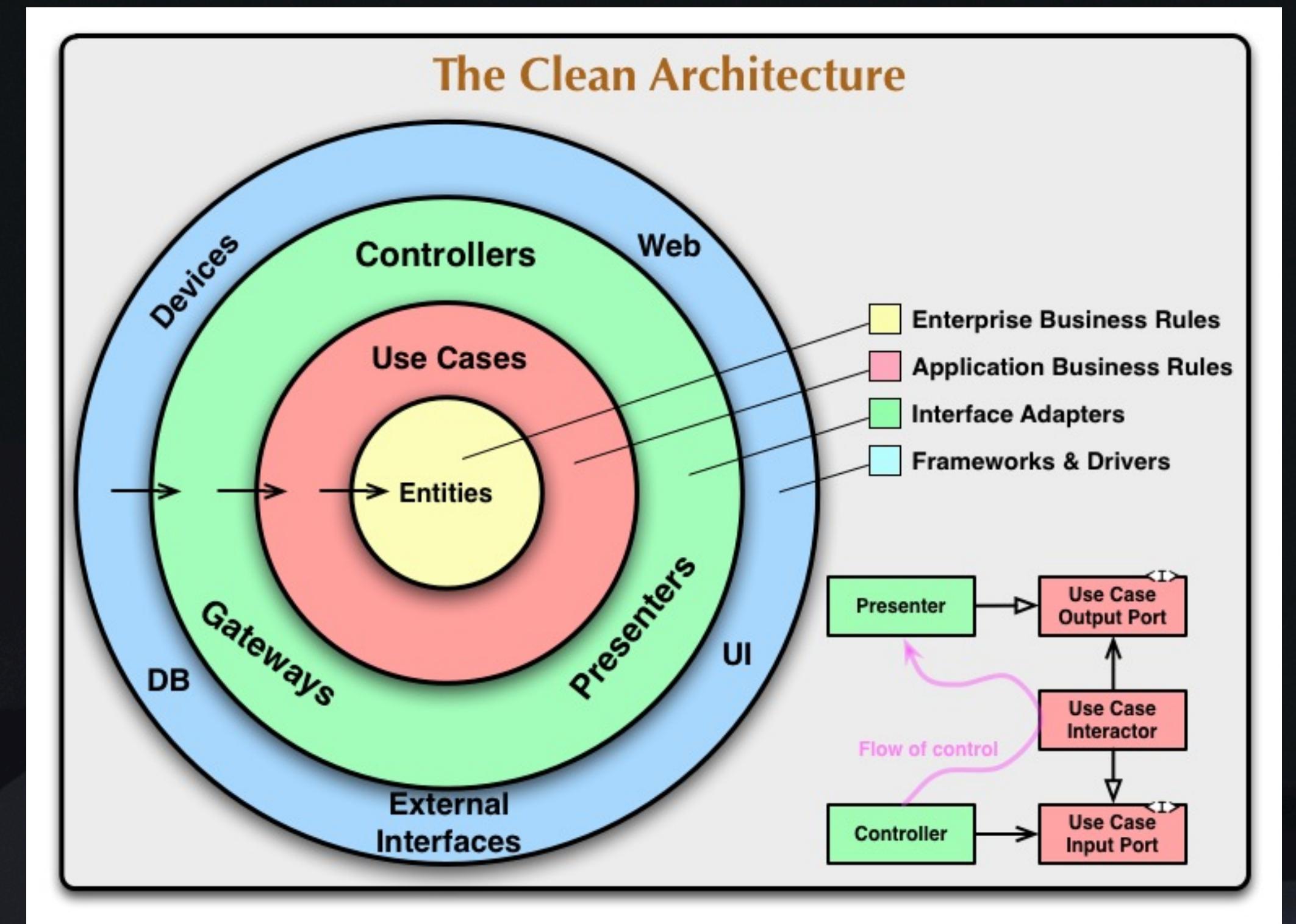
Requirements

- 5 years of programming experience in either Objective-C or Swift
- Experience developing, releasing and maintaining native iOS applications
- Experience with Java or willingness to learn
- Capacity to work efficiently as part of a team; able to give and accept feedback
- Experience designing clean and maintainable APIs
- Experience with multithreaded programming
- Experience writing unit tests and testable code

Utilizando arquiteturas e SOLID

Arquiteturas

- Ao separar as responsabilidades do código em diferentes camadas, conseguimos **isolar** e testar com facilidade.
- Facilita a modularização e reaproveitamento de testes também!



Utilizando UI Design Patterns

MVC, MVVM, MVP, etc. 😅

- Possuir camadas com as responsabilidades bem definidas facilita MUITO a criação de testes.
- Conseguimos testar cada uma dessas camadas ao separá-las.

Model

Controller

View

SOLID + Testes

- (S)ingle Responsibility Principle: cada classe e método deve ter um único propósito e função.
- (D)eependency inversion principle: sua classe deve depender de abstrações ao invés de referências concretas.
- Desta forma temos classes mais claras e independentes!

```
final class SessionHelper {

    private let validateUserName: ValidateUsername
    private let validatePassword: ValidatePassword
    private let getUserSession: GetUserSession
    private let setUserSession: SetUserSession

    init(validateUserName: ValidateUsername, validatePassword: ValidatePassword, getUserSession: GetUserSession, setUserSession: SetUserSession) {
        self.validateUserName = validateUserName
        self.validatePassword = validatePassword
        self.getUserSession = getUserSession
        self.setUserSession = setUserSession
    }

    func login(username: String, password: String) {
        if validateUserName.isValid(username) && validatePassword.isValid(password) {
            guard !getUserSession.isLoggedIn() else { return }

            setUserSession.logIn(username: username, password: password)
        }
    }

    func logout(username: String, password: String) {
        // Logout logic
    }
}
```



```
final class ValidateCredentialsUseCase {
    private let validateUserName: ValidateUsername
    private let validatePassword: ValidatePassword

    init(validateUserName: ValidateUsername, validatePassword: ValidatePassword) {
        self.validateUserName = validateUserName
        self.validatePassword = validatePassword
    }

    func execute(username: String, password: String) -> Bool {
        validateUserName.isValid(username) && validatePassword.isValid(password)
    }
}

final class LoginUseCase {
    private let getUserSession: GetUserSession
    private let setUserSession: SetUserSession

    init(getUserSession: GetUserSession, setUserSession: SetUserSession) {
        self.getUserSession = getUserSession
        self.setUserSession = setUserSession
    }

    func execute(username: String, password: String) {
        guard !getUserSession.isLoggedIn() else { return }
        setUserSession.logIn(username: username, password: password)
    }
}
```



```
final class HomeController {
    private let tableView = UITableView()

    func reloadData() {
        CoreDataStack().getUsers { users in
            tableView.reloadData()
        }
    }
}
```



```
protocol CoreDataProtocol {
    func getUsers(completion: (([User]) -> Void))
}

extension CoreDataStack: CoreDataProtocol {}

final class NewHomeController {
    private let tableView = UITableView()

    private let coreData: CoreDataProtocol

    init(coreData: CoreDataProtocol) {
        self.coreData = coreData
    }

    func reloadData() {
        coreData.getUsers { users in
            tableView.reloadData()
        }
    }
}
```



O que devo testar?

Tudo o que fizer sentido!

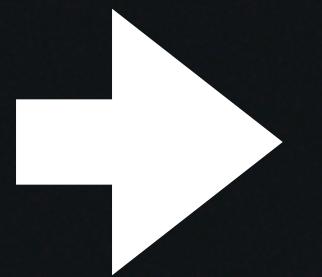


- Como o nome já diz “Teste Unitário”, ele deve testar cada unidade do seu código. O que são unidades? Funções, linhas, possibilidades.
- Ao escrever o teste, usar a mentalidade de: o que devo fazer para quebrar esse teste?
- Não se prender apenas ao happy path.
- Criar testes relevantes para o seu negócio! Um código com 100% de cobertura não necessariamente é um código funcional, apenas significa que os testes passaram por cada linha. Quem dita o que está certo e errado é você!

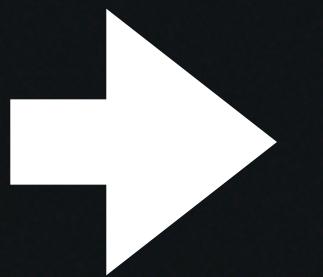
Como testar?

Estrutura

Dado um cenário hipotético
X,



Quando ocorrer determinados Y
eventos, ações, interações..



O resultado esperado deve
ser Z.

Como testar?

Nomenclaturas

- Given-When-Then (ou qualquer outra nomenclatura que faça sentido na sua empresa).
 - Ex.: *func test_givenUserIsAlreadyLoggedIn_whenExecute_shouldNotCreateSession() { ... }*
- SUT (System Under Test): normalmente utilizada para chamar a instância que está sendo utilizada para os testes.

Como testar?

Doubles (dublês)

- Stub: injetar dados e comportamentos que você deseja testar ->

```
class UserAPI {  
    func getUsers() -> [User] {  
        // Logic that get users from backend...  
    }  
}
```

```
class UserAPIStub: UserAPI {  
    override func getUsers() -> [User] {  
        [User(name: "User1"), User(name: "User2"), User(name: "User3")]  
    }  
}
```

- Spy: definir e observar chamadas ->

```
class UserAPISpy: UserAPI {  
  
    private(set) var getUsersCalledCount = 0  
    var getUsersCalledToBeReturned = [User]()  
  
    override func getUsers() -> [User] {  
        getUsersCalledCount += 1  
        return getUsersCalledToBeReturned  
    }  
}
```

Como testar o seguinte código?

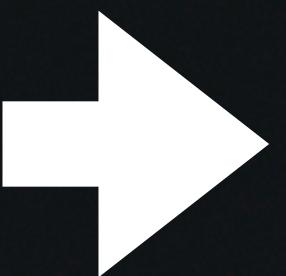
```
final class LoginUseCase {

    private let getUserSession: GetUserSession
    private let setUserSession: SetUserSession

    init(getUserSession: GetUserSession, setUserSession: SetUserSession) {
        self.getUserSession = getUserSession
        self.setUserSession = setUserSession
    }

    func execute(username: String, password: String) {
        guard !getUserSession.isLoggedIn() else { return }

        setUserSession.logIn(username: username, password: password)
    }
}
```



```
import XCTest
@testable import CocoaPresentation

final class LoginUseCaseTests: XCTestCase {
    private let getUserSessionSpy = GetUserSessionSpy()
    private let setUserSessionSpy = SetUserSessionSpy()

    private var sut: LoginUseCase {
        LoginUseCase(getUserSession: getUserSessionSpy, setUserSession: setUserSessionSpy)
    }

    func test_givenUserIsAlreadyLoggedIn_whenExecute_shouldNotSetNewLoginSession() {
        getUserSessionSpy.isLoggedInToBeReturned = true

        sut.execute(username: .random(), password: .random())

        XCTAssertEqual(setUserSessionSpy.logInCalledCount, 0)
    }

    func test_givenUserIsNotLoggedIn_whenExecute_shouldSetNewLoginSession() {
        getUserSessionSpy.isLoggedInToBeReturned = false

        let username = String.random()
        let password = String.random()

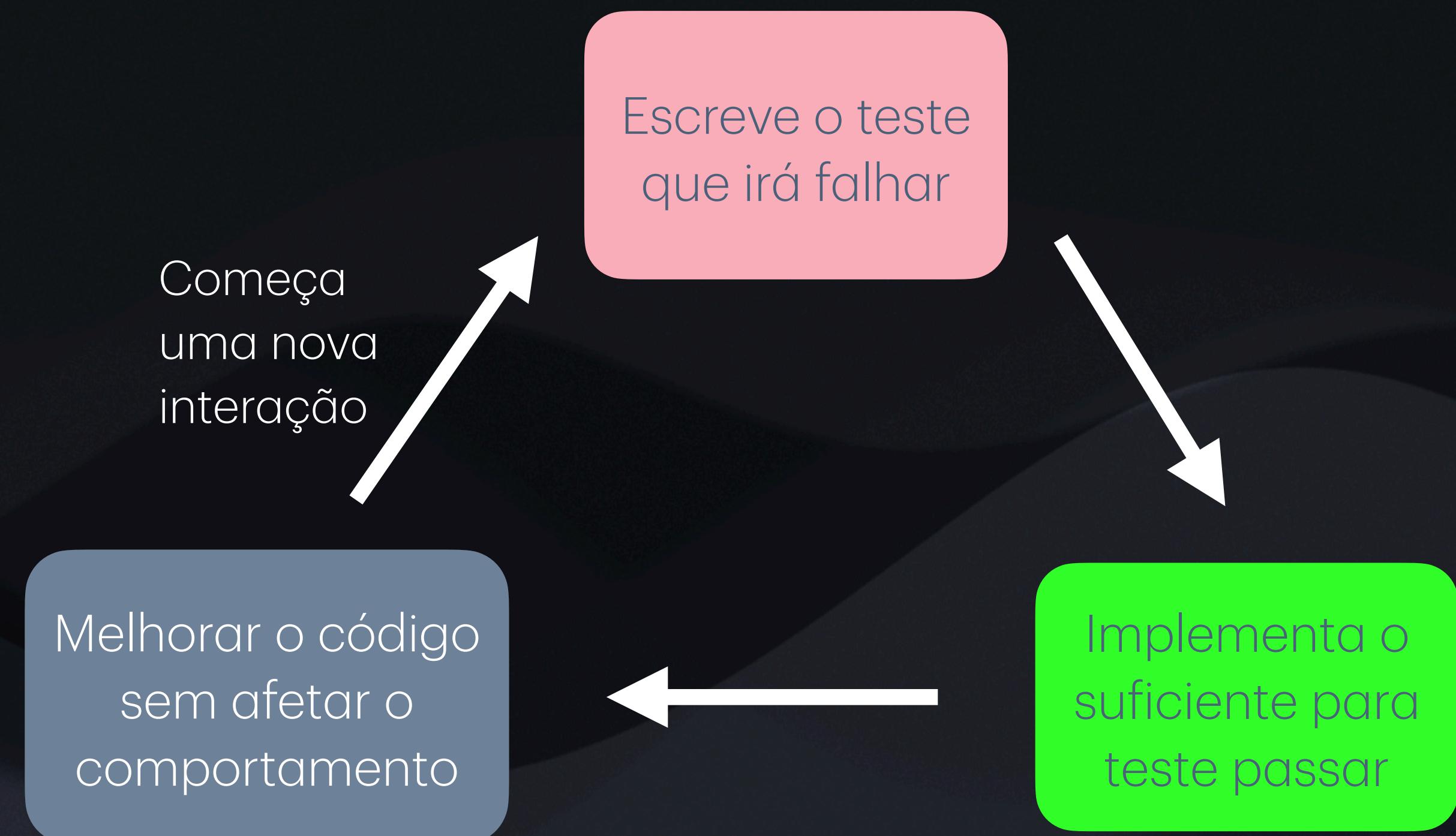
        sut.execute(username: username, password: password)

        XCTAssertEqual(setUserSessionSpy.logInCalledCount, 1)
        XCTAssertEqual(setUserSessionSpy.logInUsernamePassed, username)
        XCTAssertEqual(setUserSessionSpy.logInPasswordPassed, password)
    }
}
```

Como testar?

TDD (Test Driven Development)

- Técnica de programação onde começamos com a escrita dos testes, e a partir deles surge a implementação. Como isso funciona?



Dicas legais!

Compartilhando algumas dicas de cursos e experiências

- Teste para capturar memory leaks:

```
extension XCTestCase {
    func trackForMemoryLeaks(_ instance: AnyObject, file: StaticString = #file, line: UInt = #line) {
        tearDown { _ in
            XCTAssertNil(instance, "Instance should have been deallocated. Potential memory leak.", file: file, line: line)
        }
    }
}
```

```
func makeSUT(file: StaticString = #file, line: UInt = #line) -> (LoginUseCase, GetUserSessionSpy, SetUserSessionSpy) {
    let getUserSession = GetUserSessionSpy()
    let setUserSession = SetUserSessionSpy()
    let sut = LoginUseCase(getUserSession: getUserSession, setUserSession: setUserSession)

    trackForMemoryLeaks(getUserSession, file: file, line: line)
    trackForMemoryLeaks(setUserSession, file: file, line: line)
    trackForMemoryLeaks(sut, file: file, line: line)

    return (sut, getUserSession, setUserSession)
}
```

Dicas legais!

Compartilhando algumas dicas de cursos e experiências

- Como testar funções assíncronas? Podemos utilizar expectations:

```
func test_givenUsersAPIReturnsUsers_whenExecute_shouldReturnTheSameUsers() {  
    let users = [User(name: "User1"), User(name: "User2")]  
  
    let usersAPISpy = UserAPIClientSpy()  
    let sut = LoadUsers(userAPIClient: usersAPISpy)  
  
    usersAPISpy.loadUsersToBeReturned = users  
  
    var expectedUsers = [User]()  
    let expectation = self.expectation(description: "Load Users from API")  
  
    sut.execute { users in  
        expectedUsers = users  
        expectation.fulfill()  
    }  
  
    wait(for: [expectation], timeout: 1)  
    XCTAssertEqual(expectedUsers, users)  
}
```

Dicas legais!

Compartilhando algumas dicas de cursos e experiências

- Como testar códigos que fazem chamadas a 3rd-parties ou frameworks da Apple?

```
final class NotificationViewController: UIViewController {

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

    private func sendNotification() {
        let content = UNMutableNotificationContent()
        content.title = "Hello!"
        content.sound = UNNotificationSound.default

        let trigger = UNTimeIntervalNotificationTrigger(timeInterval: 5, repeats: false)
        let request = UNNotificationRequest(identifier: UUID().uuidString, content: content, trigger: trigger)

        UNUserNotificationCenter.current().add(request)
    }
}
```

Dicas legais!

Compartilhando algumas dicas de cursos e experiências

- Como testar códigos que fazem chamadas a 3rd-parties ou frameworks da Apple?
 - Uma opção é utilizar um protocolo com a mesma assinatura da função ou então criar uma camada de lógica para encapsular.

```
protocol UserNotificationCenter {
    func add(_ request: UNNotificationRequest, withCompletionHandler completionHandler: ((Error?) -> Void)?)  
}  
extension UNUserNotificationCenter: UserNotificationCenter {}
```

Dicas legais!

Compartilhando algumas dicas de cursos e experiências

- Experimentar diferentes tipos de XCTAsserts: dependendo do tipo de assertion, é possível obter erros com mensagens mais relevantes:

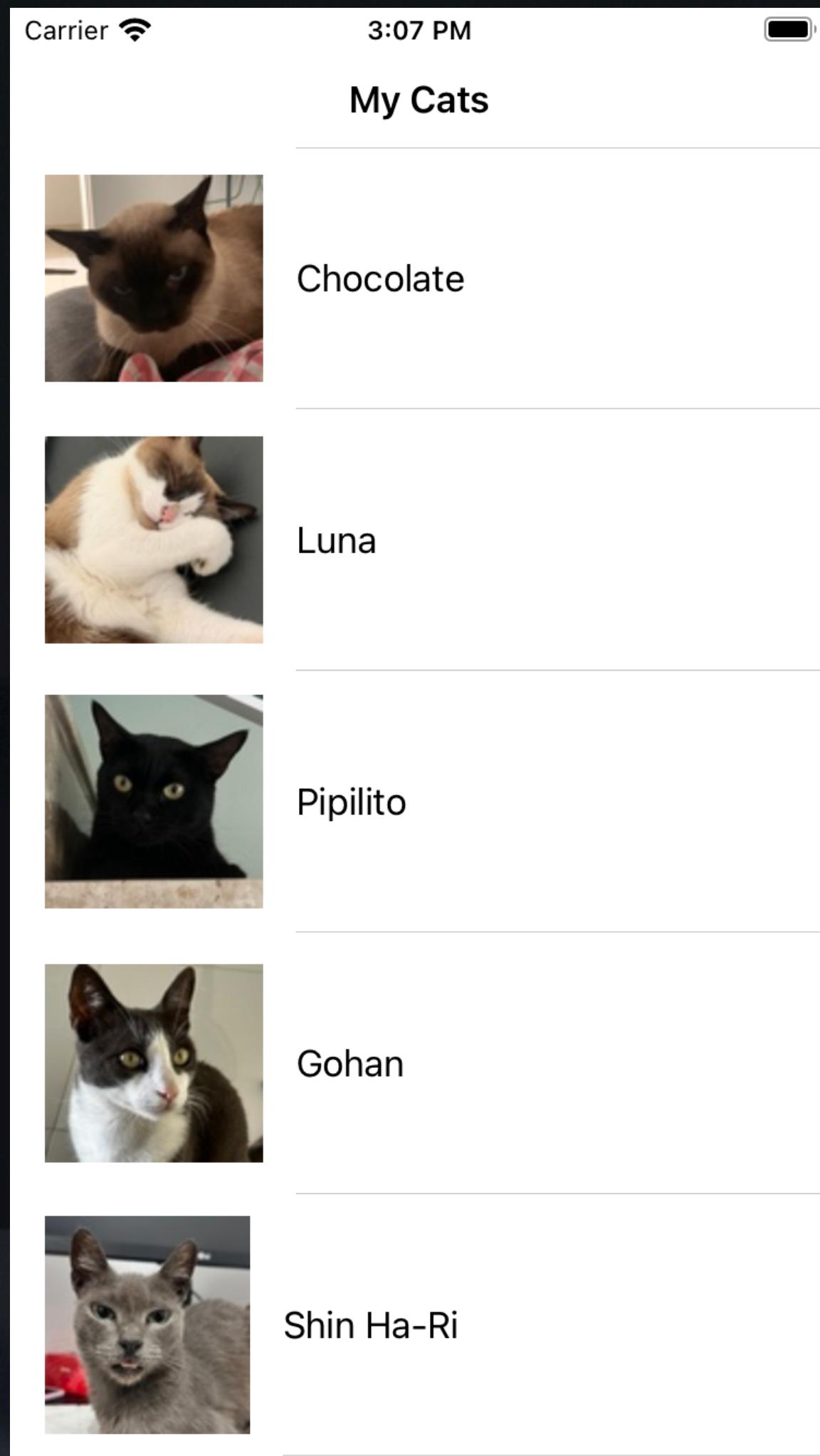
```
18
19 x func test_givenUserIsAlreadyLoggedIn_whenExecute_shouldNotSetNewLoginSession() {
20     getUserSessionSpy.isLoggedInToBeReturned = true
21
22     sut.execute(username: .random(), password: .random())
23
24     XCTAssertTrue(setUserSessionSpy.logInCalledCount == 1)
25 }
26
x test_givenUserIsAlreadyLoggedIn_whenExecute_shouldNotSetNewLoginSession(): XCTAssertTrue failed
```

```
18
19 x func test_givenUserIsAlreadyLoggedIn_whenExecute_shouldNotSetNewLoginSession() {
20     getUserSessionSpy.isLoggedInToBeReturned = true
21
22     sut.execute(username: .random(), password: .random())
23
24     XCTAssertEqual(setUserSessionSpy.logInCalledCount, 1)
25 }
26
x test_givenUserIsAlreadyLoggedIn_whenExecute_shouldNotSetNewLoginSession(): XCTAssertEqual failed: ("0") is not equal to ("1")
```

Hands-on



Um aplicativo que carrega uma lista de gatos



Referências

- WWDCs sobre testes. Ex.: Write tests to fail (2020), Testing tips & tricks (2018), etc.
- Curso iOS Lead Essential Developer
- Livro “iOS Unit Testing by Example: XCTest Tips and Techniques Using Swift” de Jon Reid.

Dúvidas?

Obrigada!



Instagram: @annekarinyf

LinkedIn: <https://www.linkedin.com/in/anne-kariny-freitas/>

Github: @annekarinyf