



ARQUITETURA E
QUALIDADE DE SOFTWARE

MOBILE

EDUARDO LOURENÇO PINTO NETO

- ▶ Líder técnico iOS na TokenLab
- ▶ Desenvolvedor iOS desde 2013
- ▶ Formado em BCC pela USP São Carlos em 2013
- ▶ Entusiasta por computadores de alto desempenho e idealizador/builder da Nautilus Customs nas horas vagas



"THERE IS AN OLD JOKE, TOLD AMONGST MISCHIEVOUS DEVELOPERS, THAT IN ORDER TO BE CONSIDERED AN ARCHITECT YOU JUST NEED TO ANSWER EVERY TECHNICAL QUESTION WITH "IT DEPENDS"."

David Hill

"SOFTWARE APPLICATION ARCHITECTURE IS THE PROCESS OF DEFINING A STRUCTURED SOLUTION THAT MEETS ALL OF THE TECHNICAL AND OPERATIONAL REQUIREMENTS, WHILE OPTIMIZING COMMON QUALITY ATTRIBUTES SUCH AS PERFORMANCE, SECURITY, AND MANAGEABILITY."

Microsoft Application Architecture Guide, 2nd Edition

MAS SOU DEV IOS, POR QUE DEVO ME

PREOCUPAR?

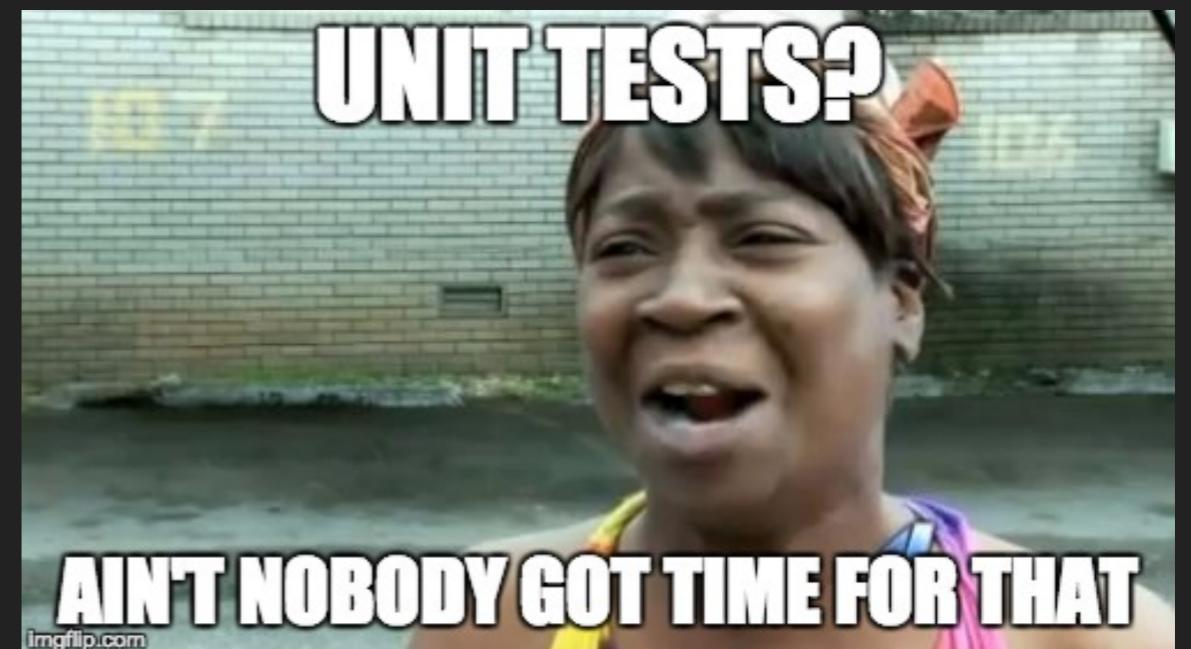
POR QUE ME PREOCUPAR COM ARQUITETURA DE SOFTWARE?

ARQUITETAR = ORGANIZAR = ESTRUTURAR → QUALIDADE

- ▶ Sem organização e estruturação, vai chegar o dia em que você vai se ver debugando uma classe com “dez” responsabilidades diferentes e falhando miseravelmente.
- ▶ Essa classe vai estar completamente fora de controle e você não vai conseguir “manter o fio da meada” ao fazer sua manutenção, esquecendo/deixando passar detalhes importantes.

SINAIS DE QUE ALGO ESTÁ ERRADO

- ▶ Essa classe com “dez” responsabilidades herda de UIViewController
- ▶ Seu *data set* reside em um UIViewController
- ▶ Seus testes unitários são difíceis de implementar e não cobrem casos importantes



APPLE MVC = APPLE “MASSIVE-VIEW-CONTROLLER”

- ▶ A culpa não é toda nossa, desenvolvedores (ufa!!!).
- ▶ A Apple não facilitou nosso trabalho ao “padronizar” um *design pattern* que não facilita:
 1. Modularização/divisão de responsabilidades
(`UIViewControllerAnimated`s, `Storyboards`)
 2. Integração com API’s nativas (Cores & Kits)

AGORA SABEMOS DOS “BUGS”... MAS E AS

FEATURES?

RESPONSABILIDADES

- ▶ Uma boa arquitetura deve começar com distribuição e balanceamento de responsabilidades
- ▶ Aquela classe com “dez” responsabilidades? O correto seriam “dez” classes com uma responsabilidade cada.

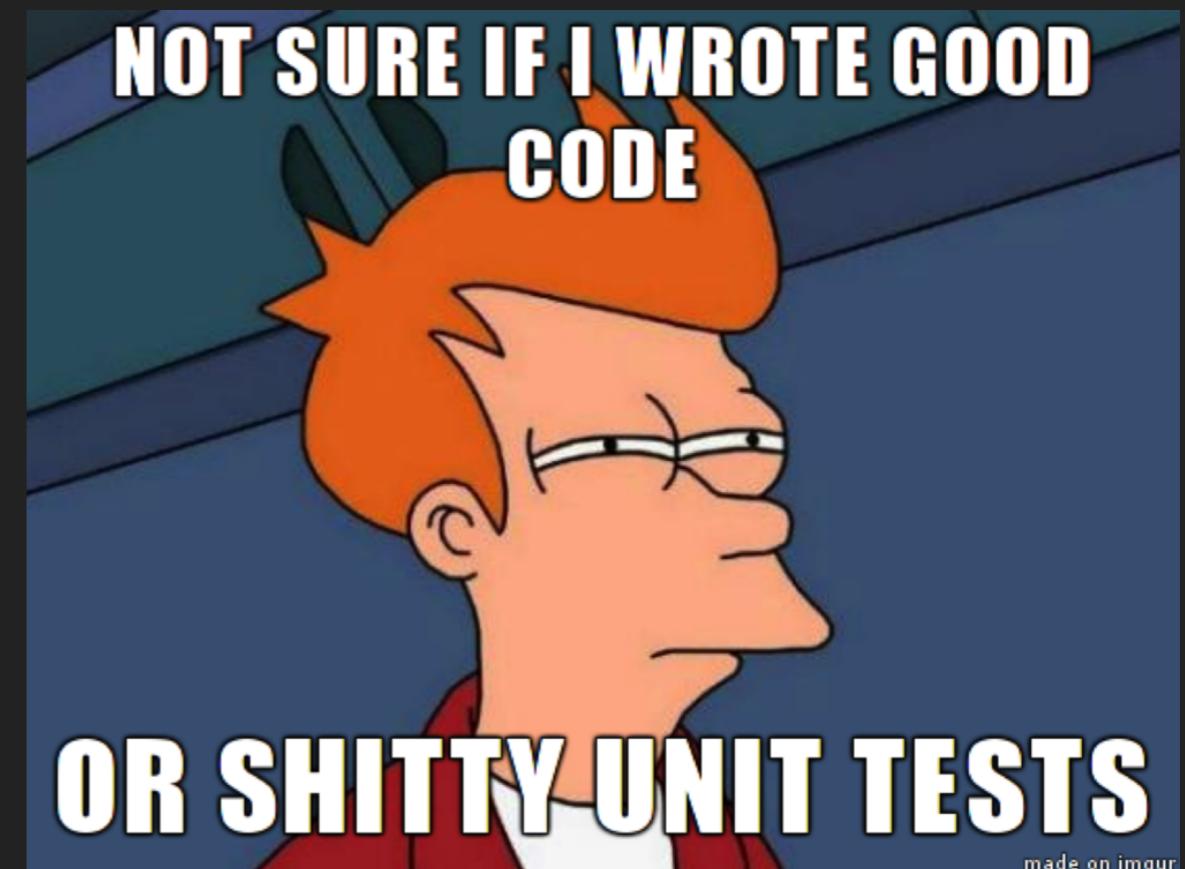


RESPONSABILIDADES - S.O.L.I.D.

- ▶ O raciocínio humano não escala de maneira linear com a complexidade dos cenários que enfrentamos
- ▶ Emprestamos alguns princípios de POO
 1. Responsabilidade Única
 2. Extender, não Modificar
 3. Inversão de Controle

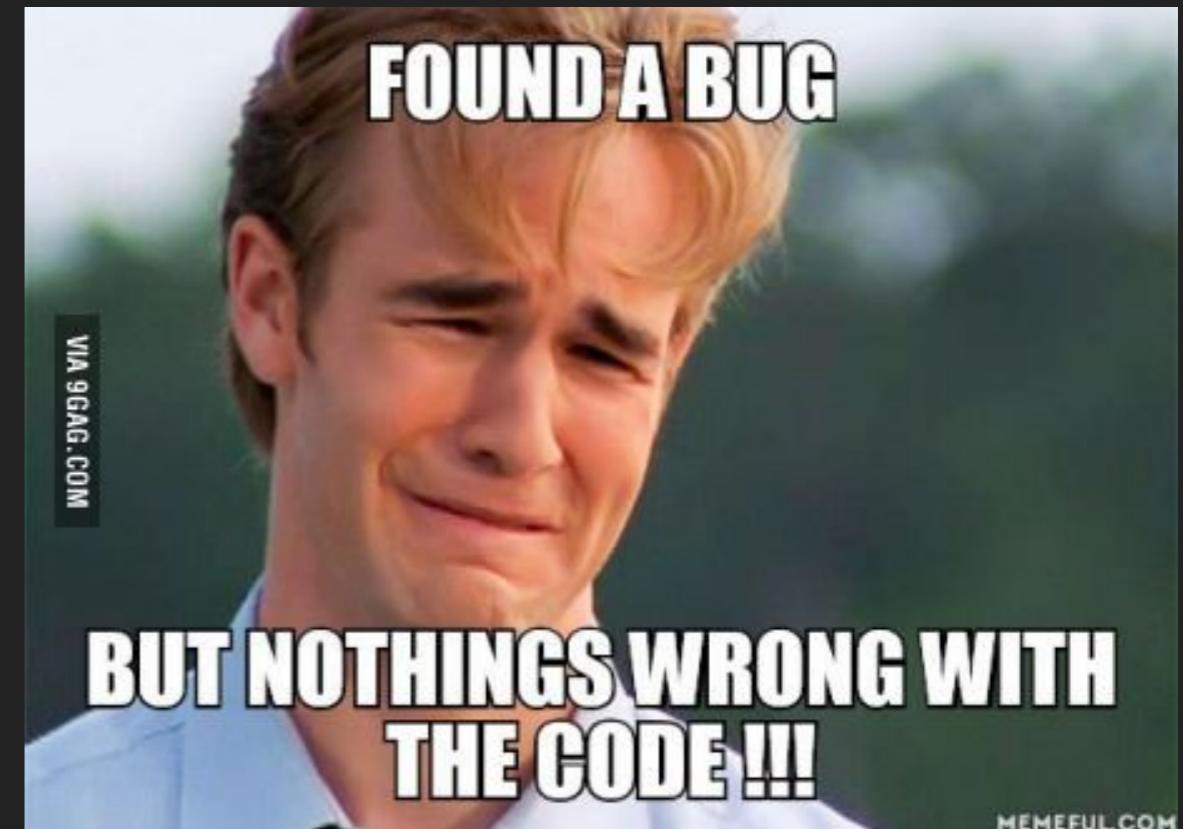
TESTABILIDADE

- ▶ Com responsabilidades unitárias, conseguimos testar de maneira unitária/encontrar bugs unitários
- ▶ Ótimo para quando cada alteração feita no software passa por todo um pipeline de aprovação para atingir nosso usuário (isso é para você iTunes Connect!).



MANUTENIBILIDADE

- ▶ Ao termos dependências e responsabilidades sob controle, sabemos em que lugar do código procurar quando recebemos um *bug report*.
- ▶ Encontrar mais rápido, corrigir mais rápido, enviar antes para review!

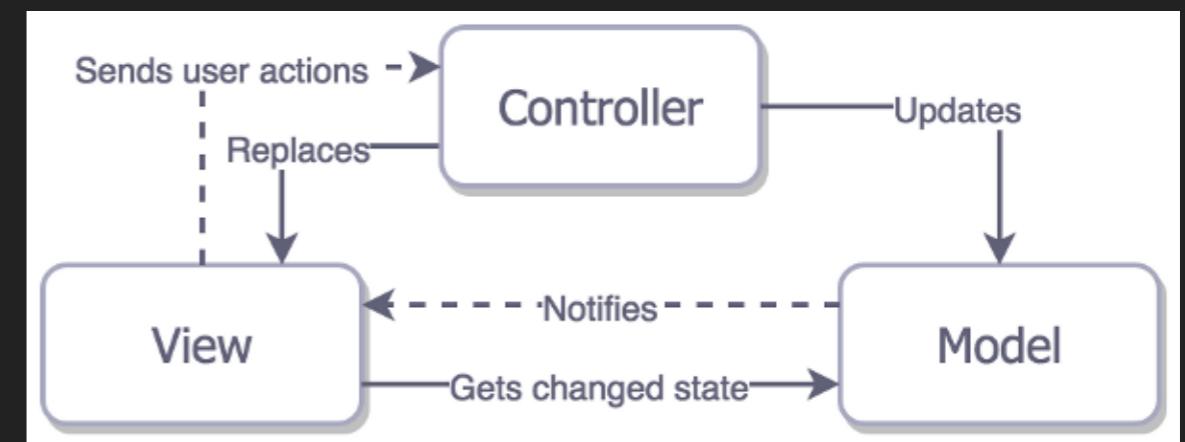


CADA CASO É UM CASO...

“IT DEPENDS”

MVC

- ▶ MVC stands for Model - View - Controller.
- ▶ A View é onde dados são apresentados
- ▶ O Model é aonde são armazenados e “adquiridos” os dados
- ▶ O Controller altera dados do Model de acordo com interações da View e também reflete alterações dos dados do Model na View.

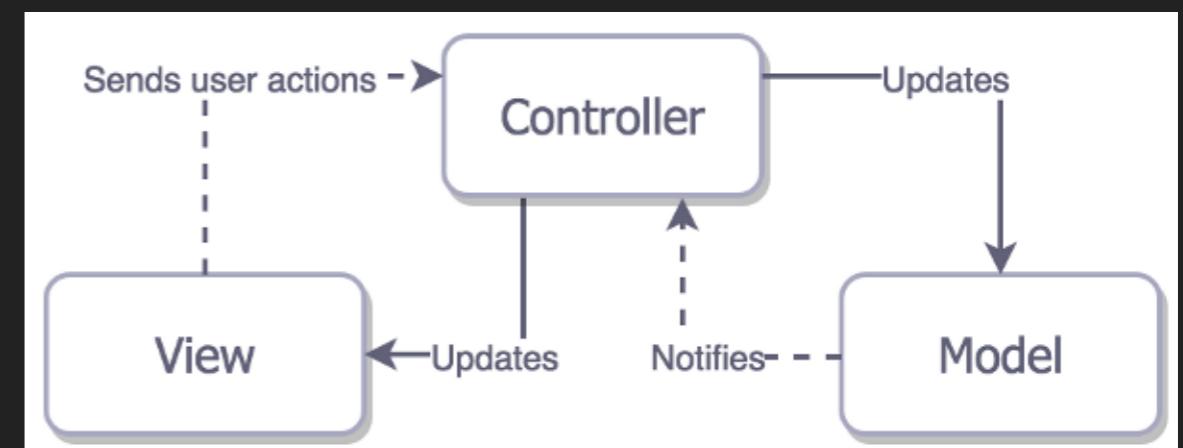


MVC

- ▶ Apesar de ser possível de se implementar um app iOS utilizando esse *architecture design pattern*, o fato das três entidades estarem fortemente **acopladas** diminui bastante sua reusabilidade (e não queremos isso, lembra?)

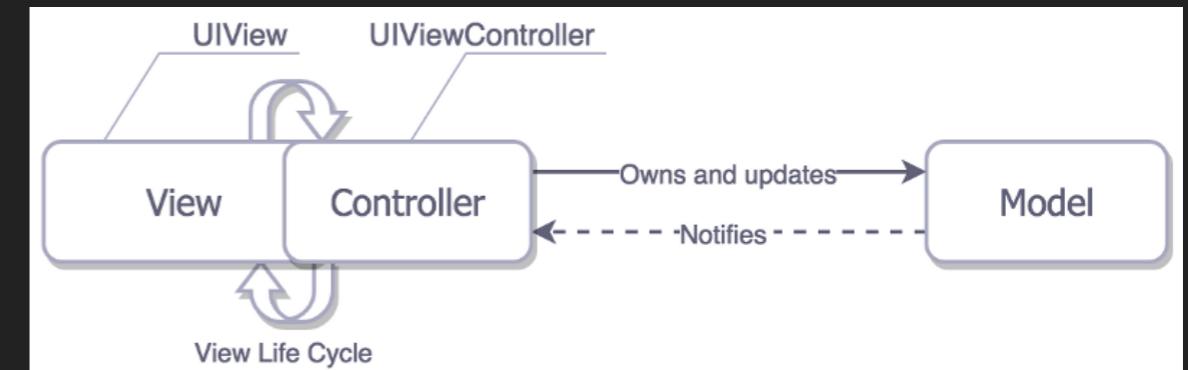
APPLE MVC - A EXPECTATIVA

- ▶ Também referido como **Cocoa MVC**, foi concebido como uma adaptação do MVC aos tempos modernos e à plataforma da Apple.
- ▶ Aqui, *View* e *Model* não interagem, **diminuindo o acoplamento**.
- ▶ O *Controller* **concentra** regras de negócios que não cabem no *Model* e faz a sua interface com a *View*.



APPLE MVC - A REALIDADE

- ▶ O resultado é que ao juntar a *View* e o *Controller* em uma só classe, fica **difícil de desacoplar View e Model**.
- ▶ Violamos o princípio da responsabilidade unitária.
- ▶ **UIViewController**s sendo *delegate* de outras *Views*, fazendo chamadas de *API's* externas, armazenando dados dos *Models*...



APPLE MVC

- ▶ Por ser o primeiro *architecture pattern* que a maioria de nós se acostumou, pode não parecer errado violar tantas regras.

```
if let userCell = tableView.dequeueReusableCell(withIdentifier: "identifier") as? UserCell {  
    userCell.configure(with: user)  
}  
  
//“MVCCellProblem.swift” by Bohdan Orlov + Eduardo Pinto revision
```

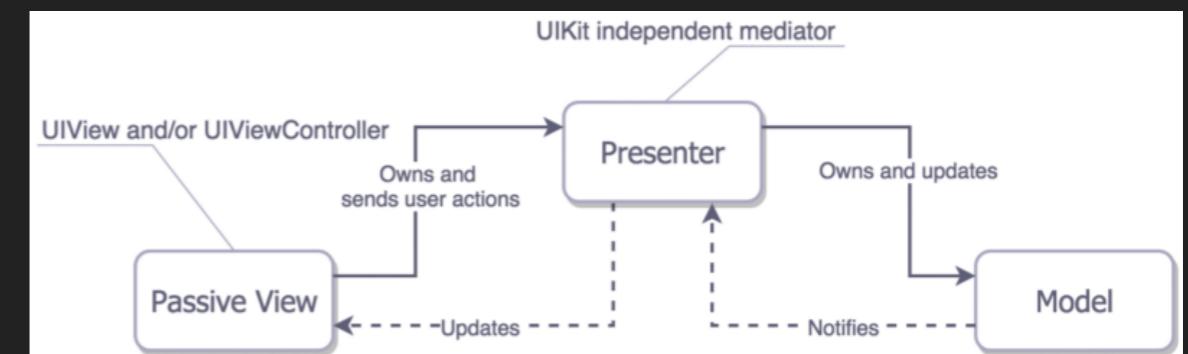
- ▶ Esse primeiro costume normalmente é incomodado quando é necessário ir contra o forte acoplamento (escrita de testes unitários, por exemplo)

APPLE MVC - AVALIAÇÃO

- ▶ **Responsabilidades:** *View* e *Model* com responsabilidades separadas, mas *View* e *Controller* **fortemente acoplados**.
- ▶ **Testabilidade:** Se bem cuidado, *Model* e algumas lógicas podem ser facilmente testados. O mesmo não acontece com *View* e *Controller*.
- ▶ **Manutenibilidade:** É fácil e rápido de se adotar, porém se mostra de difícil depuração em casos complexos.
- ▶ É uma opção a se considerar se seu projeto for simples e pequeno, onde velocidade e simplicidade são chaves.

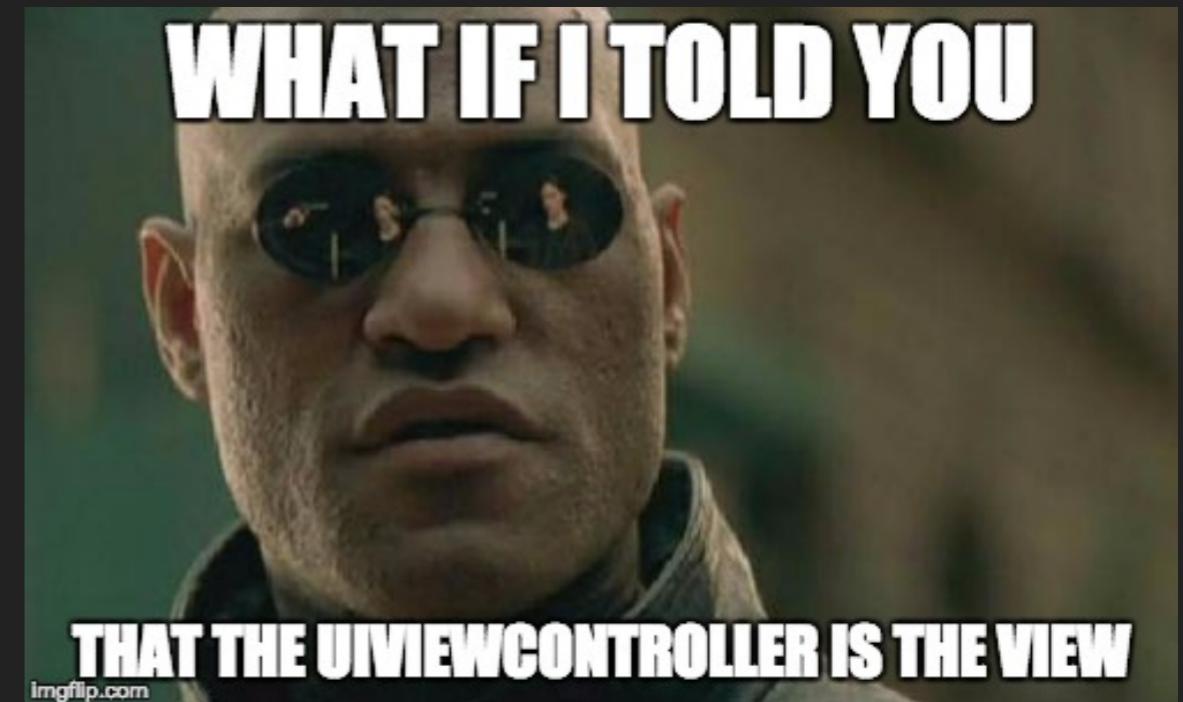
MVP - O COCOA MVC QUE DEU CERTO

- ▶ *MVP stands for Model - View - Presenter.*
- ▶ Diagrama se assemelha ao que esperávamos que o Apple MVC fosse.
- ▶ *View e Model* passam a interagir via *Presenter*, e não mais via *Controller*.



MVP

- ▶ *Presenter* é responsável apenas por alimentar e coletar ações da *View* (de fato, não há interação nenhuma entre o *Presenter* e o *UIKit*).
- ▶ **Desacoplamos** não apenas o *Model* da *View*, mas também tratamentos e interações com *API's*, residentes no *Presenter*, da lógica de apresentação, residente na *View*.



MVP - COMUNICAÇÃO ABSTRAÍDA

```
struct Person { // Model
    let firstName: String
    let lastName: String
}

protocol GreetingView: class {
    func setGreeting(greeting: String)
}

protocol GreetingViewPresenter {
    init(view: GreetingView, person: Person)
    func showGreeting()
}

// "MVP.swift" by Bohdan Orlov + Eduardo Pinto revision
```

MVP - PRESENTER

```
class GreetingPresenter : GreetingViewPresenter {
    weak let view: GreetingView
    let person: Person

    init(view: GreetingView, person: Person) {
        self.view = view
        self.person = person
    }

    func showGreeting() {
        let greeting = "Hello" + " " + self.person.firstName + " " + self.person.lastName
        self.view.setGreeting(greeting)
    }
}

//"MVP.swift" by Bohdan Orlov + Eduardo Pinto revision
```

MVP - VIEW + MONTAGEM

```
class GreetingViewController : UIViewController, GreetingView {
    var presenter: GreetingViewPresenter!
    let greetingLabel = UILabel()

    override func viewDidLoad() {
        super.viewDidLoad()
        self.presenter.showGreeting()
    }

    func setGreeting(greeting: String) {
        self.greetingLabel.text = greeting
    }
}

//Montagem do MVP: Note a inversão de controle/injeção de dependências
let model = Person(firstName: "Eduardo", lastName: "Pinto")
let view = GreetingViewController()
let presenter = GreetingPresenter(view: view, person: model)
view.presenter = presenter

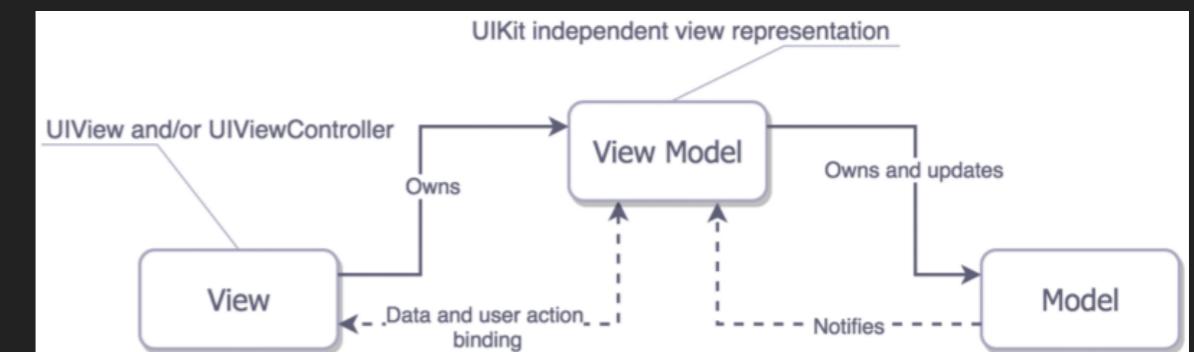
//MVP.swift" by Bohdan Orlov + Eduardo Pinto revision
```

MVP - AVALIAÇÃO

- ▶ **Responsabilidades:** Cada camada tem suas próprias responsabilidades, lógicas de negócio divididas entre *Model* e *Presenter* e apresentação concentrada na *View* favorecem um **menor acoplamento**.
- ▶ **Testabilidade:** *View* acumula apenas responsabilidades de apresentação, facilitando muito implementação de testes nas regras de negócio da aplicação.
- ▶ **Manutenibilidade:** Temos **mais código, mais boiler plate**, porém a separação de responsabilidades é clara e facilita deteção e correção de bugs.
- ▶ Muito importante entendermos que implementamos **muito mais código boiler plate** para atingirmos melhor manutenibilidade.

MVVM

- ▶ *MVVM stands for Model - View - View Model.*
- ▶ É o membro mais moderno da família MV(X), portanto aborda alguns dos pontos não abordados pelos *patterns* anteriormente citados.
- ▶ Assim como no MVP, mantemos a *View* (aqui também representada pelo *UIViewController*) e o *Model* desacoplados.



MVVM - O VIEW MODEL E OS BINDINGS

- ▶ O *View Model* é a entidade que altera o *Model* e reflete essas alterações na *View*, por meio de **bindings**.
- ▶ **Binding:** mecanismo de notificação automática de modificações
- ▶ OSX oferece mecanismo nativo para bindings.
- ▶ O que chega mais perto no iOS são notificações por *KVO*, porém apresentam suas limitações.

MVVM - BINDINGS

- ▶ O aspecto chave do MVVM são os bindings, portanto é essencial que não fiquemos presos à soluções precárias como *KVO's "caseiros"*.
- ▶ As alternativas mais comuns são **frameworks de programação funcional reativa** que implementam o chamado *Observer pattern*.
 - ▶ ReactiveX (RxSwift)
 - ▶ Reactive Cocoa



A FAMÍLIA “MV(X)”

MVVM

```
class IssueTrackerViewController: UIViewController {
    //ViewModel Reference
    var viewModel: IssueTrackerViewModel!

    //Outlets
    @IBOutlet weak var searchBar: UISearchBar!
    @IBOutlet weak var tableView: UITableView!
    @IBOutlet weak var loadingView: UIView!

    //Rx observable for search bar text reacting
    var searchBarText: Observable<String> {
        return searchBar.rx.text.orEmpty.debounce(0.5, scheduler: MainScheduler.instance).distinctUntilChanged()
    }

    //Rx dispose bag, for cleaning up
    var disposeBag = DisposeBag()

    override func viewDidLoad() {
        super.viewDidLoad()
        tableView.register(R.nib.issueTableViewCell)
        setupRx()
    }

    //GithubIssueTracker by Eduardo Pinto
```

A FAMÍLIA “MV(X)”

MVVM

```
func setupRx() {
    viewModel = IssueTrackerViewModel(with: searchBarText)

    viewModel.isLoading.map({ isLoading -> Bool in
        let isHidden = !isLoading
        return isHidden
    }).drive(loadingView.rx.isHidden).addDisposableTo(disposeBag)

    viewModel.isLoading.drive(UIApplication.shared.rx.isNetworkActivityIndicatorVisible).addDisposableTo(disposeBag)
    setupTableView()
}

func setupTableView() {
    viewModel.issues.drive(tableView.rx.items) { table, row, issue in
        let cell = table.dequeueReusableCell(withIdentifier: R.reuseIdentifier.issueTableViewCellIdentifier)!
        cell.configure(with: issue.cellText)
        return cell
    }.addDisposableTo(disposeBag)
}

//GithubIssueTracker by Eduardo Pinto
```

A FAMÍLIA “MV(X)”

MVVM

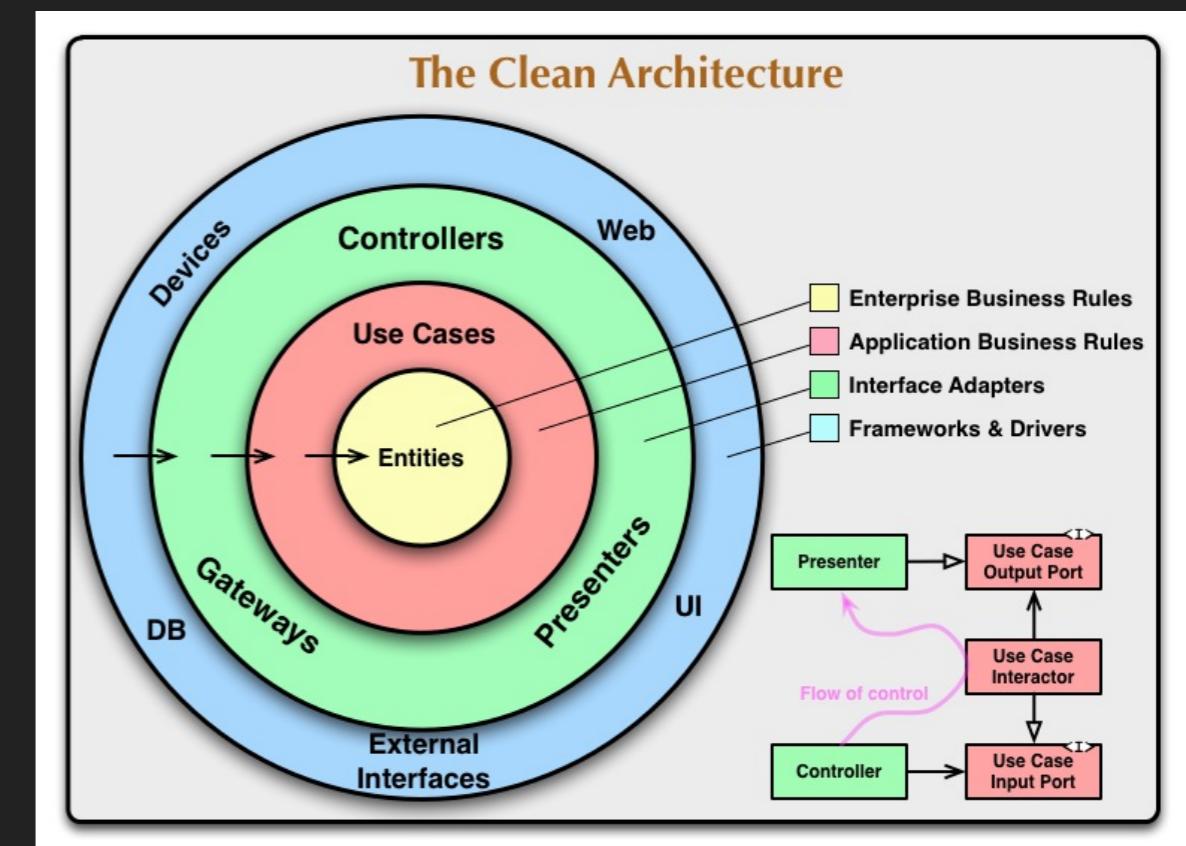
```
struct IssueTrackerViewModel {  
    var issues: Driver<[DisplayedIssue]>  
    var isLoading: Driver<Bool>  
    var issueTracker: IssueTracker  
  
    init(with repositoryName: Observable<String>) {  
        issueTracker = IssueTracker()  
  
        issues = issueTracker.trackIssues(for: repositoryName).flatMap({ (issues) -> Observable<[DisplayedIssue]> in  
            Observable<[DisplayedIssue]>.just(issues.map { DisplayedIssue(cellText:$0.title) })  
        }).asDriver(onErrorJustReturn: [])  
  
        isLoading = issueTracker.isLoading.asDriver()  
    }  
}  
  
//GithubIssueTracker by Eduardo Pinto
```

MVVM- AVALIAÇÃO

- ▶ **Responsabilidades:** Mantém *View* e *Model* **desacoplados**, porém introduz **mais responsabilidades** na *View* (setup de *bindings*), comparado ao MVP.
- ▶ **Testabilidade:** O *View Model* não conhece nenhum detalhe sobre a *View*, facilitando os testes.
- ▶ **Manutenibilidade:** *Bindings* e programação reativa introduzem dificuldade de depuração, porém apresenta distribuição de responsabilidades equivalente ao MVP e sem tanto *boiler plate*.
- ▶ Apresenta um paradigma com curva de aprendizado íngreme (programação reativa), porém atinge boa testabilidade e manutenibilidade sem introduzir excesso de código.

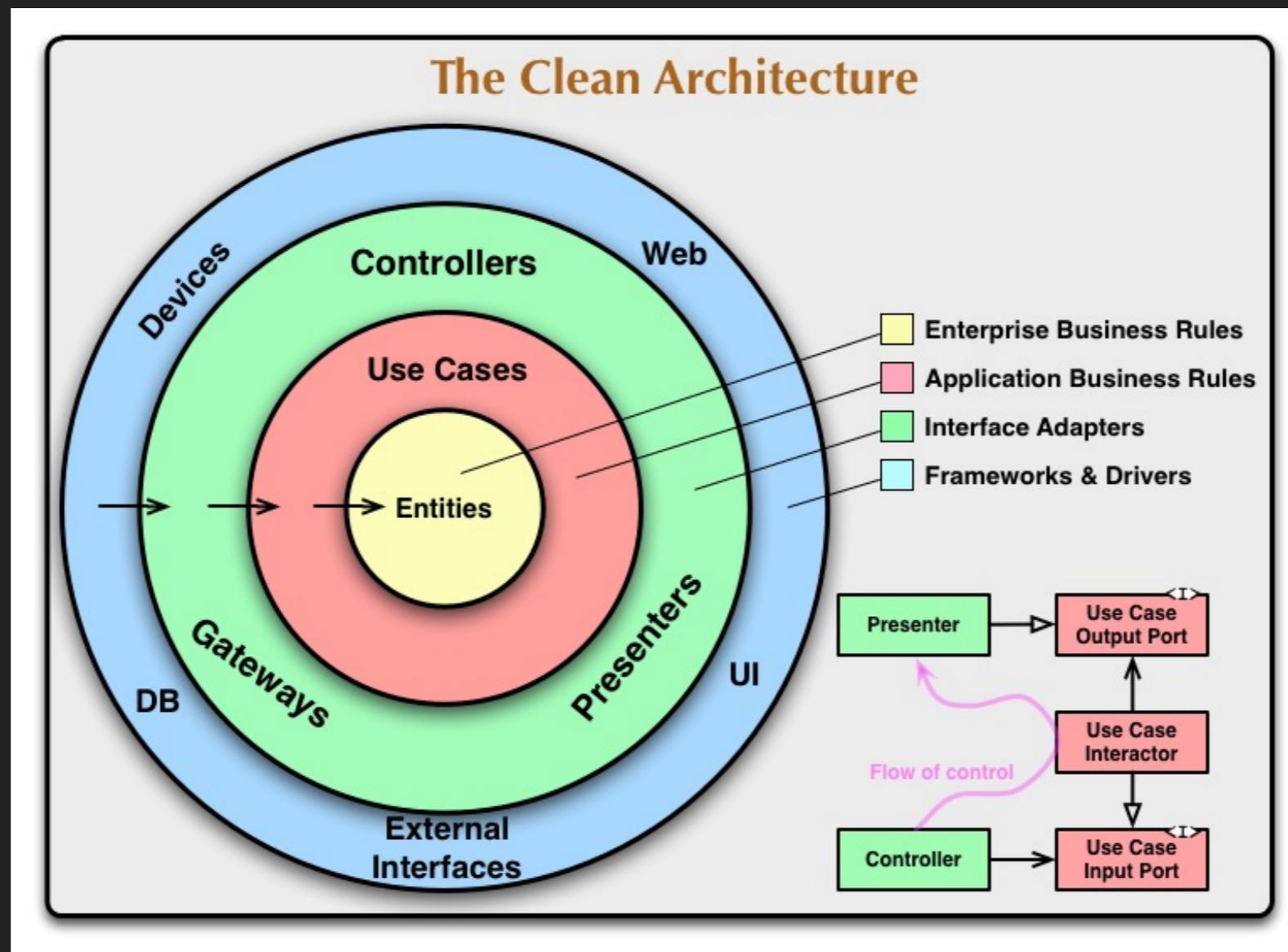
CLEAN ARCHITECTURE

- ▶ Idealizada por Robert “Uncle Bob” Martin no artigo “The Clean Architecture”.
- ▶ Inspirada por arquiteturas como *Hexagonal* e *Onion*, portanto é uma “nova família”, oposta à família MV(X).
- ▶ Busca separar as responsabilidades de um software por meio de camadas que o tornem “independente do mundo externo”.



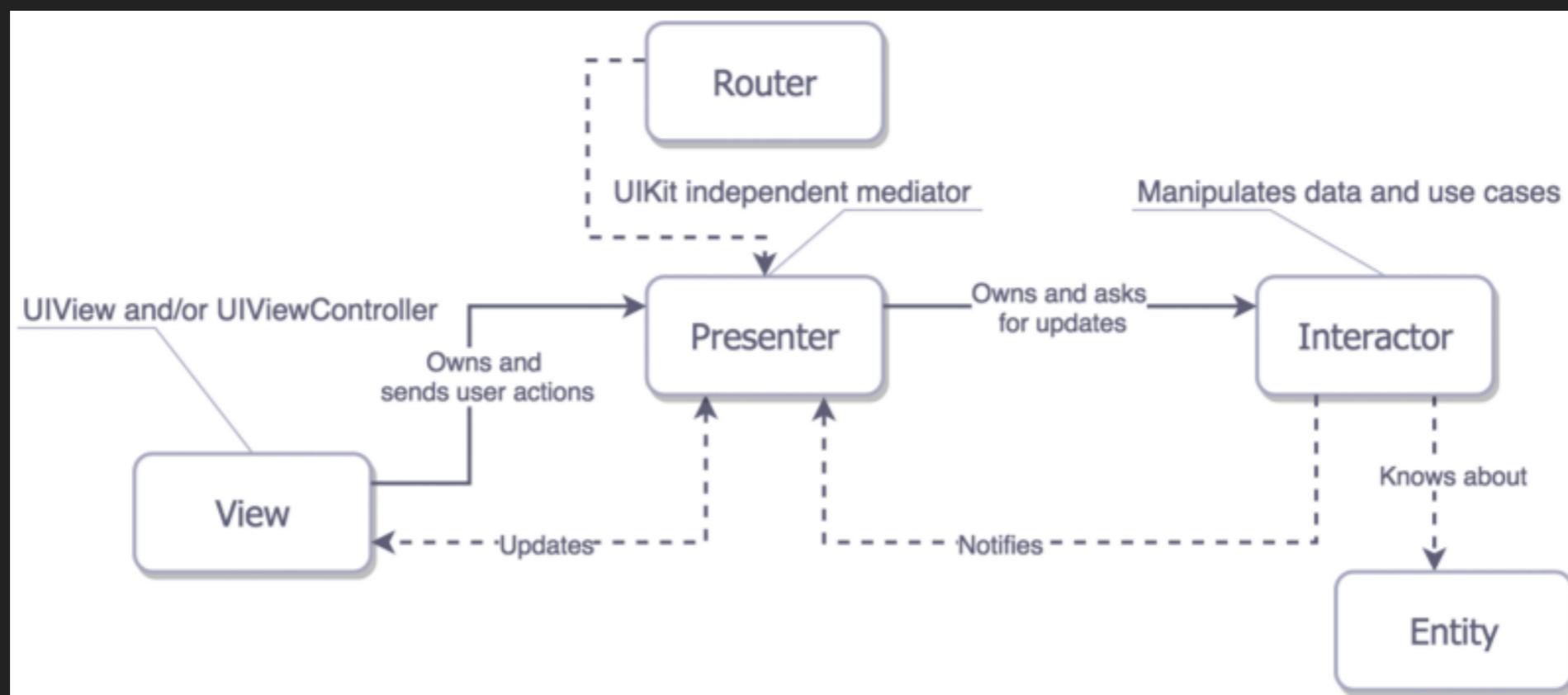
CLEAN ARCHITECTURE

- ▶ Regra de Dependência



VIPER - CLEAN TRADUZIDO PARA APLICAÇÕES IOS

- ▶ VIPER stands for View - Interactor - Presenter - Entity - Router.
- ▶ Foi a primeira interpretação de sucesso da *Clean Architecture* para iOS, idealizada por Jeff Gilbert e Conrad Stoll.



VIPER - AS CAMADAS

- ▶ **Interactor:** é aonde residem as lógicas de negócio relacionadas às *Entities*, como comunicação com *web services* e *caching*.
- ▶ **Presenter:** contém lógica de formatação e apresentação do que será apresentado na *View*. Recebe *input* da *View* e invoca métodos do *Interactor* para adquirir dados.
- ▶ **Entities:** objetos de armazenamento simples, sem lógica de negócios, que representam .
- ▶ **Router:** responsável pela navegação entre **módulos VIPER**.

“VIPER ADOPTS THE TRADITIONAL
STACK ARCHITECTURE APPROACH
WHERE COMMUNICATION
IS BIDIRECTIONAL.”



Paul Singer

VIPER - OS MÓDULOS

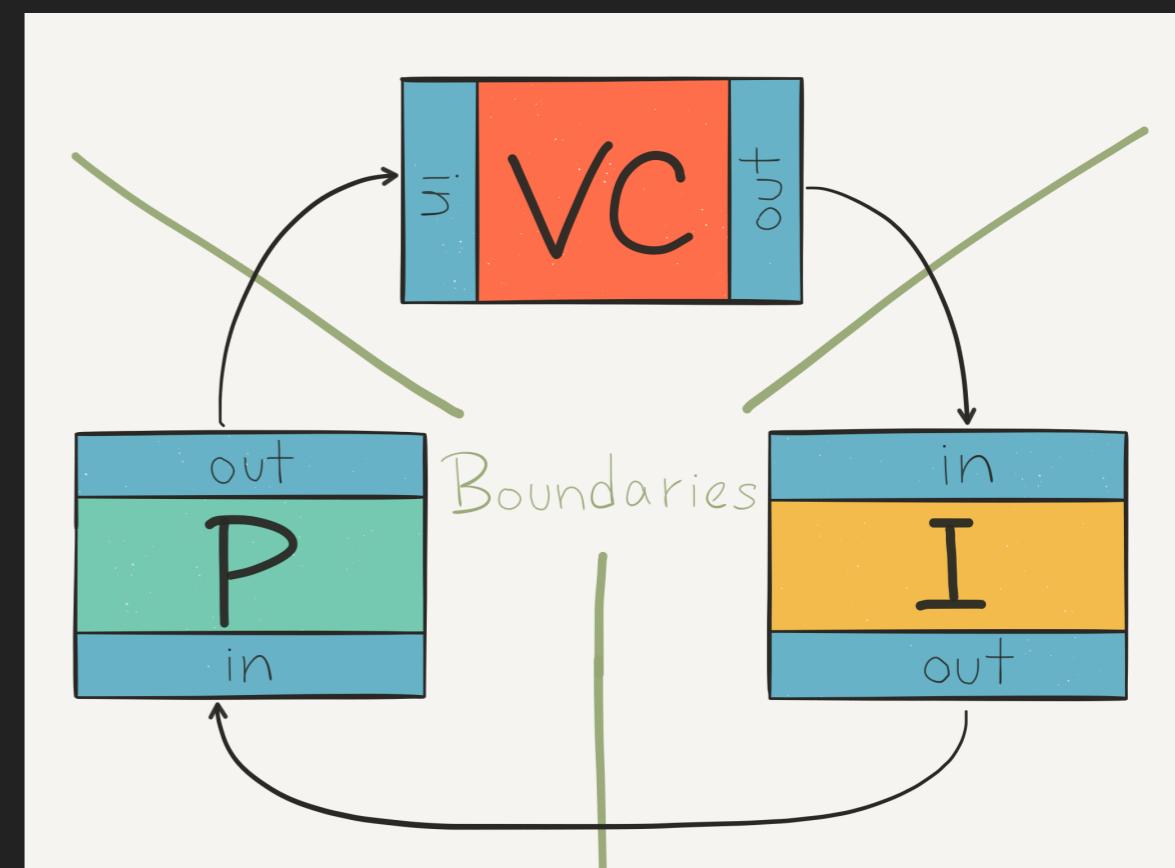
- ▶ Um **módulo** VIPER pode variar de um componente visual em uma tela até um caso de uso completo, composto de várias telas.
- ▶ Módulos navegam e interagem entre si pelo *Router*, podendo passar/receber parâmetros de um módulo para outro.
- ▶ Um módulo possui seus próprios *Interactor*, *Presenter*, *Entity* e *Router*.

VIPER- AVALIAÇÃO

- ▶ **Responsabilidades:** Cada uma das cinco camadas proporcionam **distribuição de responsabilidades maximizada e baixo acoplamento** entre as mesmas.
- ▶ **Testabilidade:** Maior distribuição implica em maior testabilidade.
- ▶ **Manutenibilidade:** A distribuição de responsabilidades facilita a depuração de bugs, porém introduz uma quantidade imensa de código de interface entre classes de altíssima modularidade.
- ▶ Deve haver muito cuidado ao considerar a utilizar VIPER em um projeto: a expectativa de uma futura facilidade na manutenção pode mascarar o fato de que a arquitetura pode ser *overkill* para seu projeto e que ele nunca irá alcançar uma manutenção complexa.

CLEAN SWIFT

- ▶ Também inspirada pela *Clean Architecture* de Uncle Bob.
- ▶ De certa maneira, tenta resolver o “problema” da comunicação bidirecional introduzida pelo VIPER
 - ▶ Ciclo VIP unidirecional
 - ▶ Modelos de comunicação



CLEAN SWIFT - AVALIAÇÃO

- ▶ **Responsabilidades:** Divisão por camadas **diminui o acoplamento.**
- ▶ **Testabilidade:** Maior distribuição implica em maior testabilidade.
- ▶ **Manutenibilidade:** Introdução do ciclo VIP unidirecional dificulta a criação de massive classes e simplifica todo o fluxo de dados.
- ▶ Alternativa de bom custo x benefício para fugir do Cocoa MVC, melhorando muito a qualidade do software, porém em alguns momentos aborda de maneira simplista os *drawbacks* da *Clean Architecture* original.

SAIBA A HORA E MANEIRA CERTA DE
VIOLAR A ARQUITETURA

“IT DEPENDS”

DUVIDAS?



REFERÊNCIAS

1. "iOS Architecture Patterns", Bohdan Orlov
2. "Getting a SOLID start.", Robert "Uncle Bob" Martin
3. Microsoft Application Architecture Guide, 2nd Edition
4. "The Clean Architecture", Robert "Uncle Bob" Martin
5. The Clean-er Architecture for iOS Apps, Paul Stringer
6. Clean Swift, Raymond Law
7. Architecting iOS Apps with VIPER, Jeff Gilbert e Conrad Stoll