RxSwift e VIPER

Denis Oliveira

Introdução

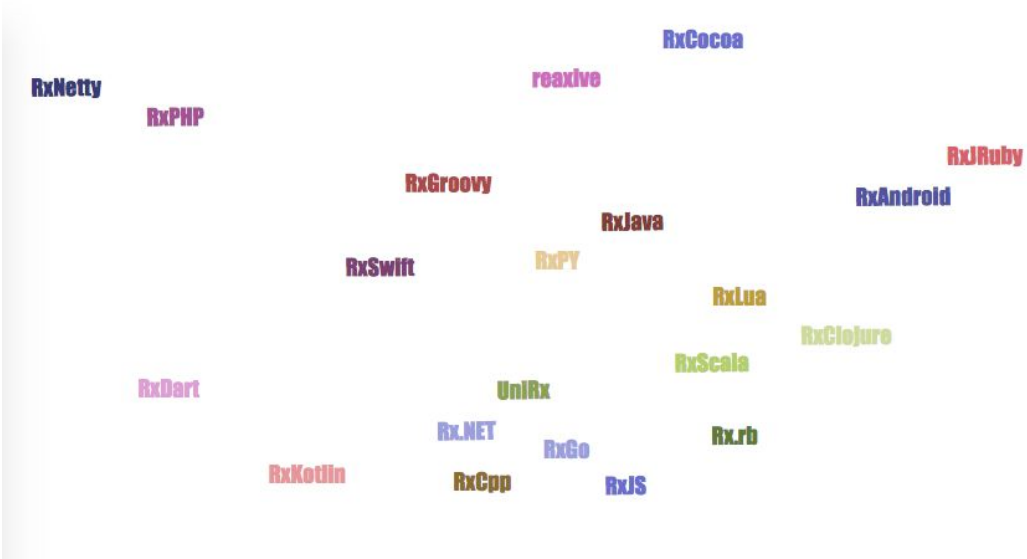
RX é um dos hottest topic em mobile

É um padrão multiplataforma

RXSwift é uma das bibliotecas que compõem Reactive.X

Reactive Extensions (RxNet)

As bibliotecas RX iniciaram como um projeto open source da Microsoft chamado Reactive Extensions



Introdução

O padrão reativo é a combinação de:

- Padrão Observer
- Padrão Iterator
- Programação Funcional

Padrão de Projeto de Software

São boas práticas para desenvolvimento de software.

Tem como objetivo solucionar problemas frequentes no desenvolvimento, são diferentes de padrões arquiteturais.

Ele não fornece a implementação mas a forma de como construí-la.

Padrão de Projeto de Software

Conforme o livro GoF "Gang of Four"

São abordados três tipos de padrão

- Criação
- Estrutural
- Comportamental

Padrão de Criação

Tem como objetivo abstrair o processo criação de novos (objetos, entidades, instâncias)

Exemplo:

- Factory
- Builder
- Singleton

Padrão Estrutural

Tem como objetivo facilitar a interação e comunicação de componentes dentro do sistema.

Exemplo:

- Adapter
- Bridge
- Composite

Padrão Comportamental

Tem como objetivo definir a responsabilidade de uma entidade ou seja qual comportamento ela vai exercer.

Exemplo:

- Command
- Observer
- Iterator

Padrão Observer

Tem como objetivo criar um relação de um-para-muitos, quando uma entidade emite uma mudança as entidades dependentes são notificadas automaticamente.

Padrão Iterator

Tem como objetivo fornecer uma maneira de acessar elementos de uma coleção de entidades de forma sequencial sem necessidade de conhecer a sua construção interna

Paradigmas de Linguagem de Programação

Fornece a abstração para

- Estruturar
- Executar

Exemplos

Estruturada, Imperativa, Orientada a objeto, Orientada a aspecto, Genética, Lógica, Funcional, Multiparadigma, dentre outras

Programação Funcional

Traz um viés matemático a programação

Evita a mudança de estado e dados são imutáveis

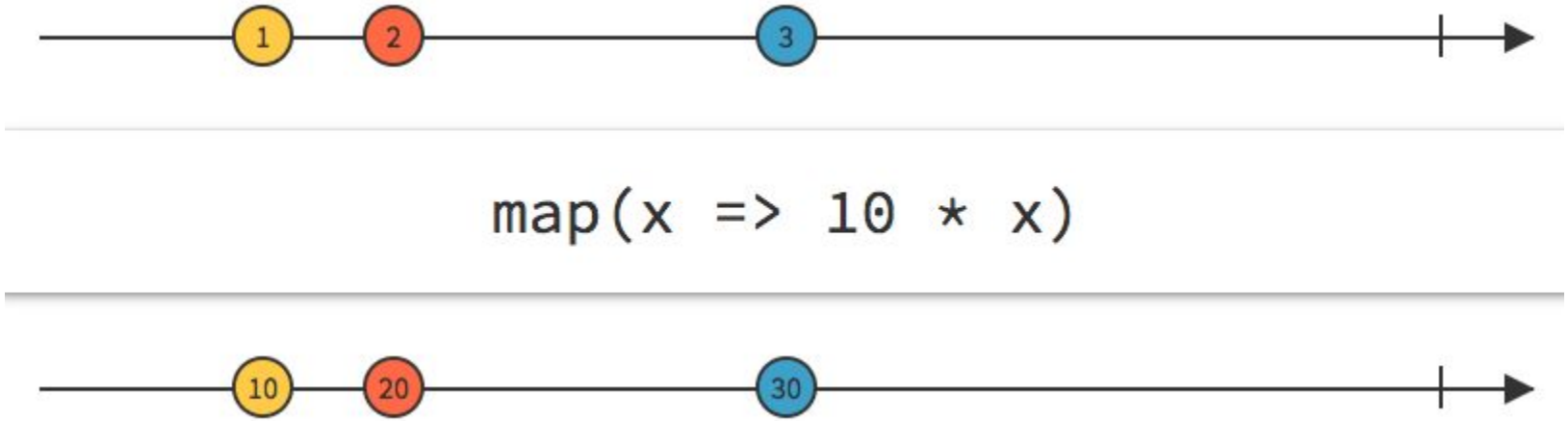
Funções de primeira classe

Recursão

Funções Lambdas

Programação Reativa

As alterações se propagam por todo o sistema automaticamente



Programação Reativa

Facilita a operações concorrentes porque não precisa realizar bloqueios enquanto uma entidade gera valor. As instruções são executadas em paralelo e seus resultados são capturados posteriormente, em ordem arbitrária. Ao invés de chamar um comportamento a ser executado você define um mecanismo para recuperar e transformar os dados, e por fim responder suas emissões sempre que estiverem prontas.

Programação Reativa

Observable

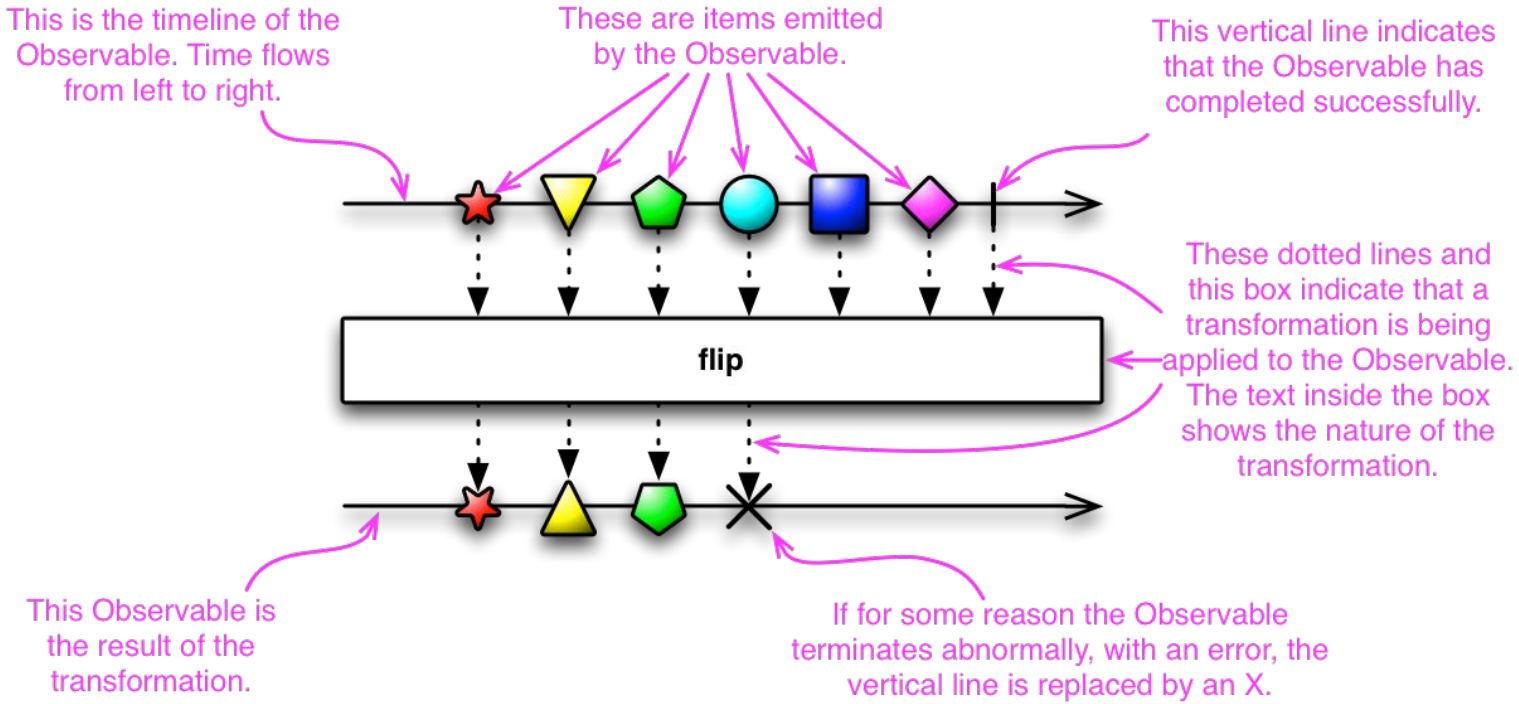
Operators

***Single**

Subject

***Schedulers**

Observable


This is the timeline of the Observable. Time flows from left to right.
These are items emitted by the Observable.
This vertical line indicates that the Observable has completed successfully.
These dotted lines and this box indicate that a transformation is being applied to the Observable. The text inside the box shows the nature of the transformation.
flip
This Observable is the result of the transformation.
If for some reason the Observable terminates abnormally, with an error, the vertical line is replaced by an X.

Observable

O método Subscribe é a conexão de um Observer a um Observable

- OnNext: chamado sempre que um observable emite um item
- OnError: chamado quando um observable não consegue emitir um item
- OnCompleted: chamado quando um observable passou por OnNext e não encontrou nenhum erro

Observable

Hot: Pode começar a emitir itens assim que é criado

Cool: Garante todos os itens da sequência desde o início

Operator

É considerado o core no ReactiveX, a maioria dos operadores operam sobre observable e retorna observable, dessa forma é possível encadear vários operadores, os operadores são categorizados por:

- Creating
- Transforming
- Filtering
- Combining
- Error Handling
- Utility Operators
- Conditional e Booleans
- Mathematical e Aggregate
- Backpressure
- Connectable
- Convert

Single

É uma variação do Observable, ao invé de usar as três respostas do Observer (onNext, onError, OnCompleted) usa somente dois:

- onSuccess
- onError

Subject

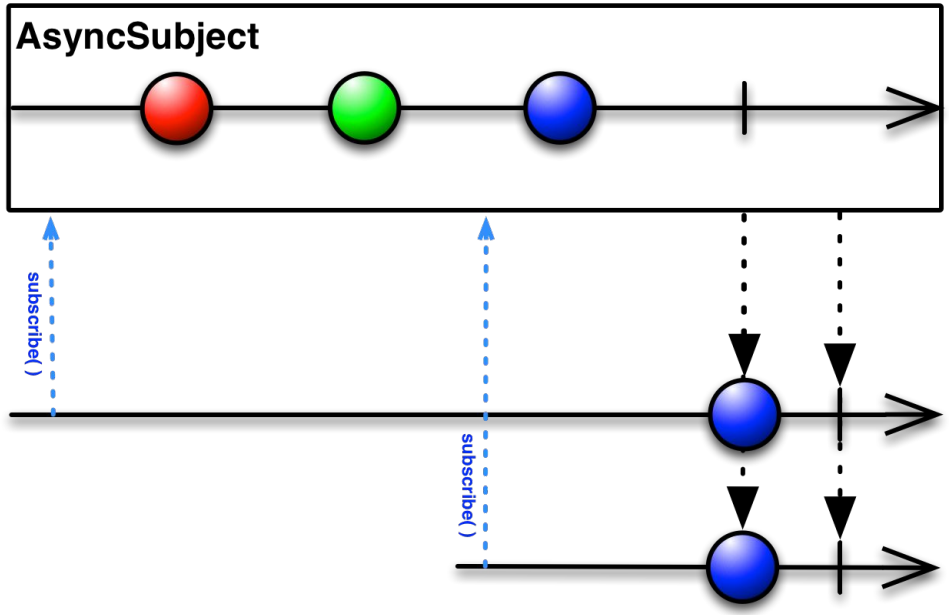
Atua como um proxy ou ponte pode ser tanto um observable quanto um observer

Existem quatro variantes:

- Async Subject
- Behaviour Subject
- Publish Subject
- Replay Subject

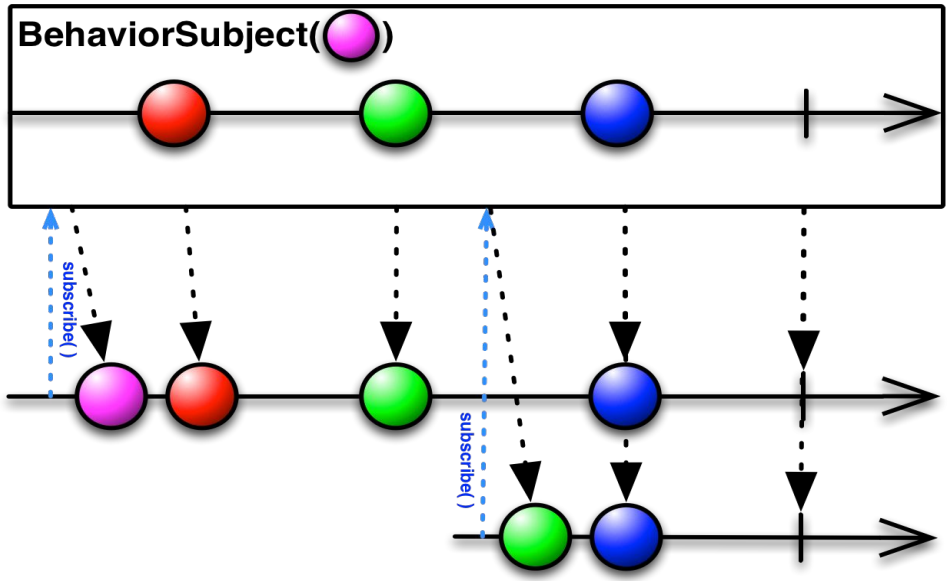
Async Subject

Emite somente o último valor emitido pelo observable



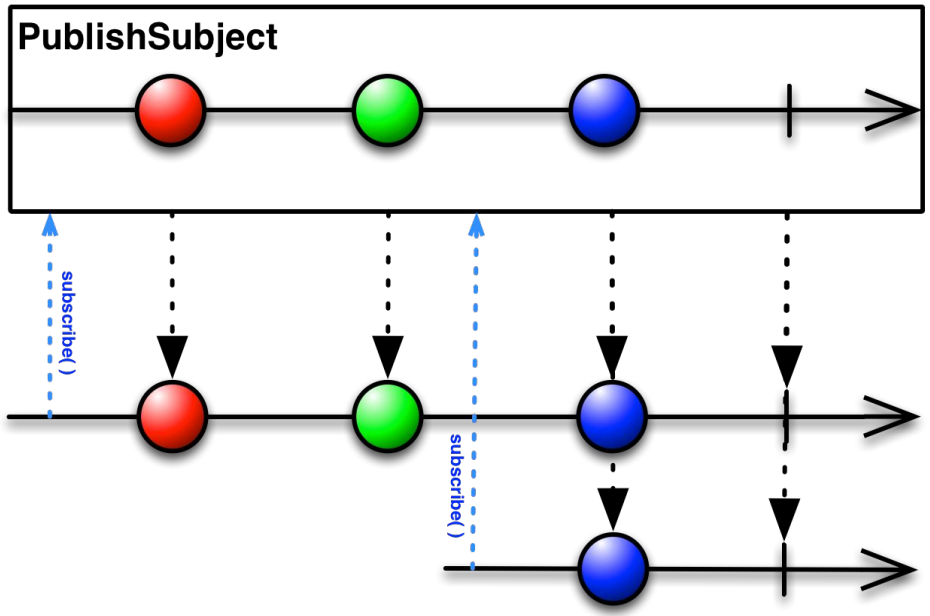
Behaviour Subject

Quando um observer subscribe ele emite os valores mais recentes



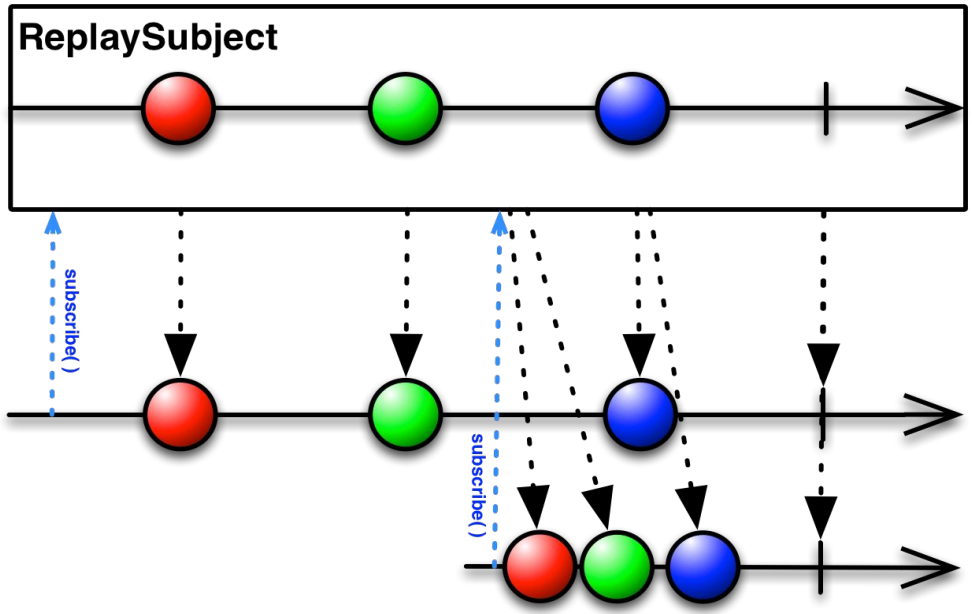
Publish Subject

Emite somente os dados que são emitidos depois do subscribe



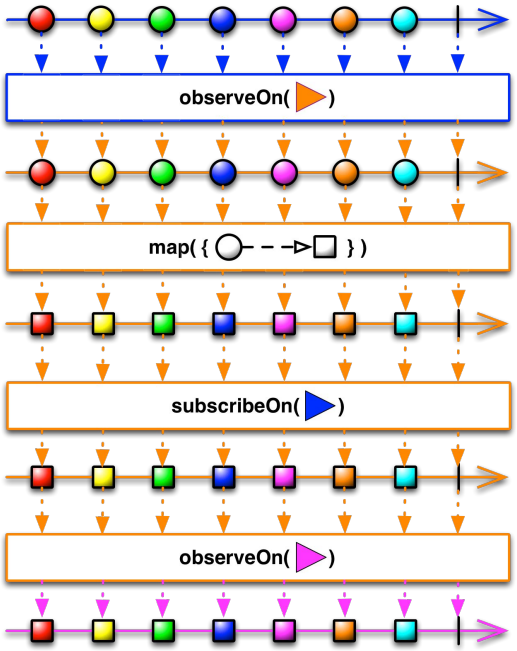
Replay Subject

Emite qualquer item que foi emitido pela fonte



Scheduler

Permite a operação dos operadores em cascata.



Diferenças

RXSwift é o framework que interage com a linguagem Swfit

RXCocoa é o framework que auxilia Cocoa APIs para usar técnicas reativas

ReativeCocoa é frameworks para a mesma finalidade. Nasceu de uma inspiração ao Microsoft Reactive Extensions para C# e tomou forma conforme o tempo.

RxSwift

```
let a = Variable(1)
let b = Variable(2)
let c = Observable.combineLatest(a.asObservable(), b.asObservable()) { $0 + $1 }
        .filter { $0 >= 0 }
        .map { "\($0) is positive" }
c.subscribe(onNext: { print($0) })
a.value = 4
b.value = -8
```

RxSwift

```
let disposeBag = DisposeBag()
let items = Observable.just(
        (0..<20).map { "\($0)" }
)

items.bind(to: tableView.rx.items(cellIdentifier: "Cell", cellType: UITableViewCell.self)) { (row, element, cell) in
        cell.textLabel?.text = "\(element) @ row \(row)"
}.disposed(by: disposeBag)

tableView.rx.modelSelected(String.self)
        .subscribe(onNext:  { value in
                DefaultWireframe.presentAlert("Tapped `\(value)`")
}).disposed(by: disposeBag)

tableView.rx.itemAccessoryButtonTapped
        .subscribe(onNext: { indexPath in
                DefaultWireframe.presentAlert("Tapped Detail @ \(indexPath.section),\(indexPath.row)")
        }).disposed(by: disposeBag)
```

RxSwift

http://reactivex.io/

https://github.com/ReactiveX/RxSwift

http://rxmarbles.com/

VIPER

Arquitetura de Software

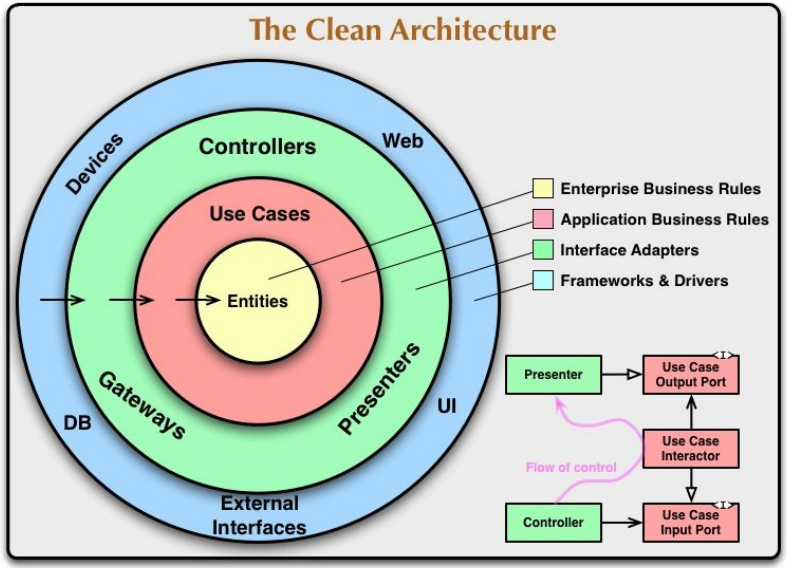
Tem como finalidade em alto nível estruturar o software tanto em sua construção e documentação, facilita decisões entre partes interessadas e captura decisões de alto nível. Após sua concepção é possível gerar componentes reutilizáveis para outros projetos.

Arquitetura de Software

Arquiteturas:

- Hexagonal
- Onion
- Screaming
- DCI
- BCE
- Clean

Clean Architecture



Arquitetura de Software - iOS

Algumas arquiteturas para desenvolvimento:

- MVC
- MVC - Lighter View Controllers
- VIPER

MVC

Também conhecida como Massive View Controller

PRÓS: é bom para softwares pequenos e para quem está iniciando.

CONTRA: Todo código fica nos controllers e os arquivos começam a ficar difíceis para manutenção, dificuldade em gerar partes testáveis

MVC - Lighter View Controllers

Realiza a separação de Data Sources entre outros Protocolos em arquivos separados.

PRÓS: Redução do controller e maior legibilidade

CONTRA: Mesmo removendo as implementações de protocolo, muita lógica fica no controller

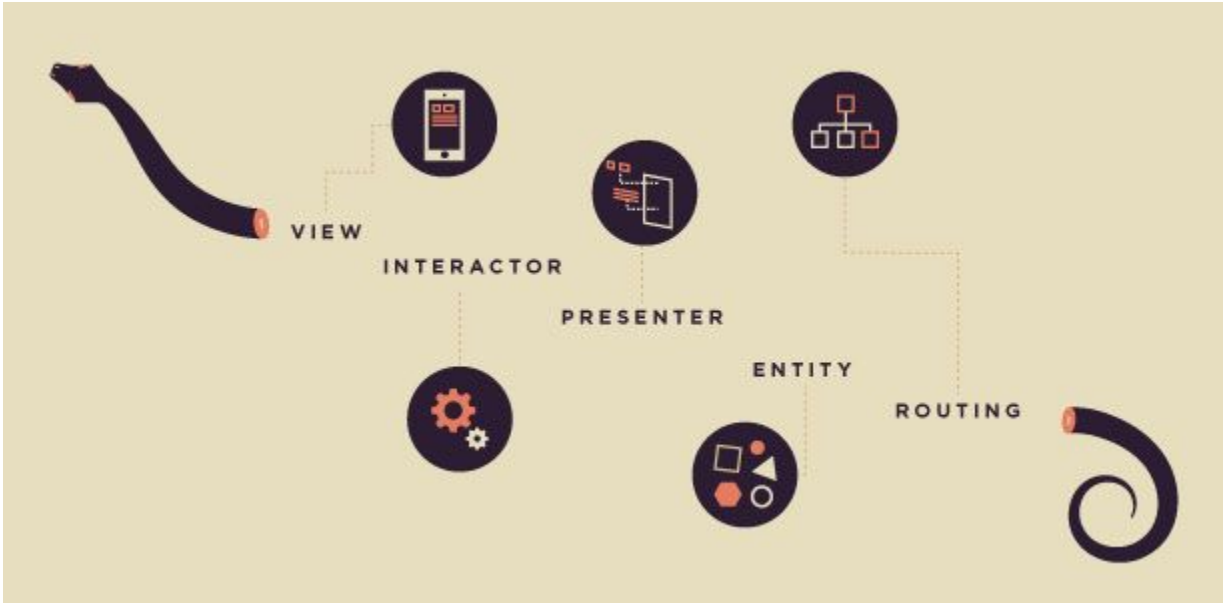
VIPER

É uma aplicação Clean Architecture para iOS.

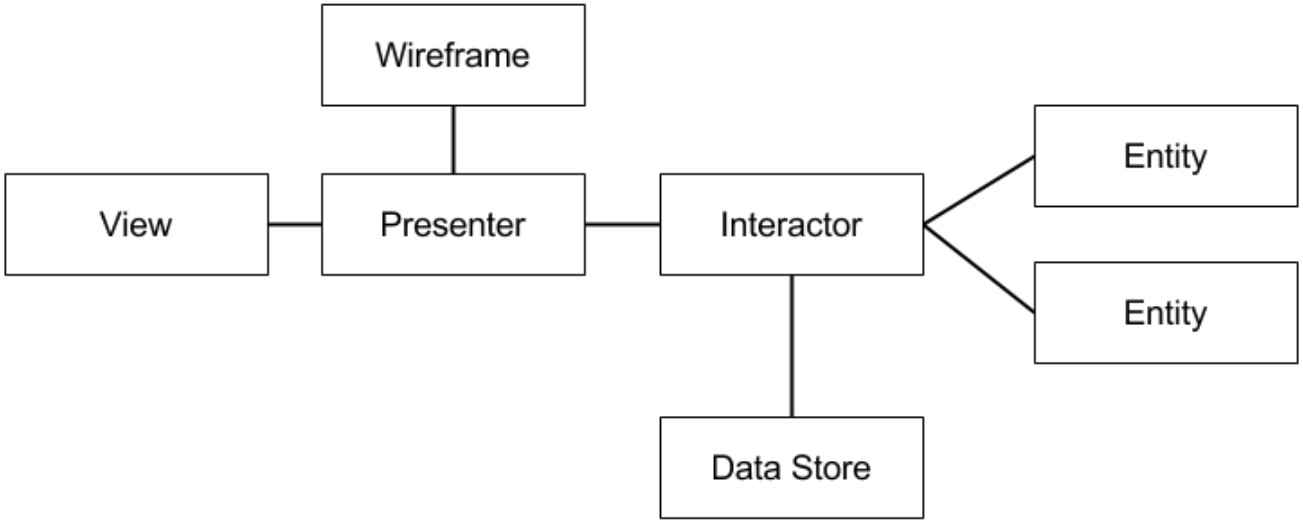
Substitui o tradicional MVC

É uma aplicação projetada para ser baseada em casos de uso

VIPER



VIPER


Wireframe
View
Presenter
Interactor
Entity
Entity
Data Store

VIPER

A separação das partes do VIPER garante a Single Responsibility Principle também conhecido como SOLID

- View é responsável pelo designer visual
- Interactor é responsável pela camada de negócio
- Presenter é representa a camada interação do designer

View

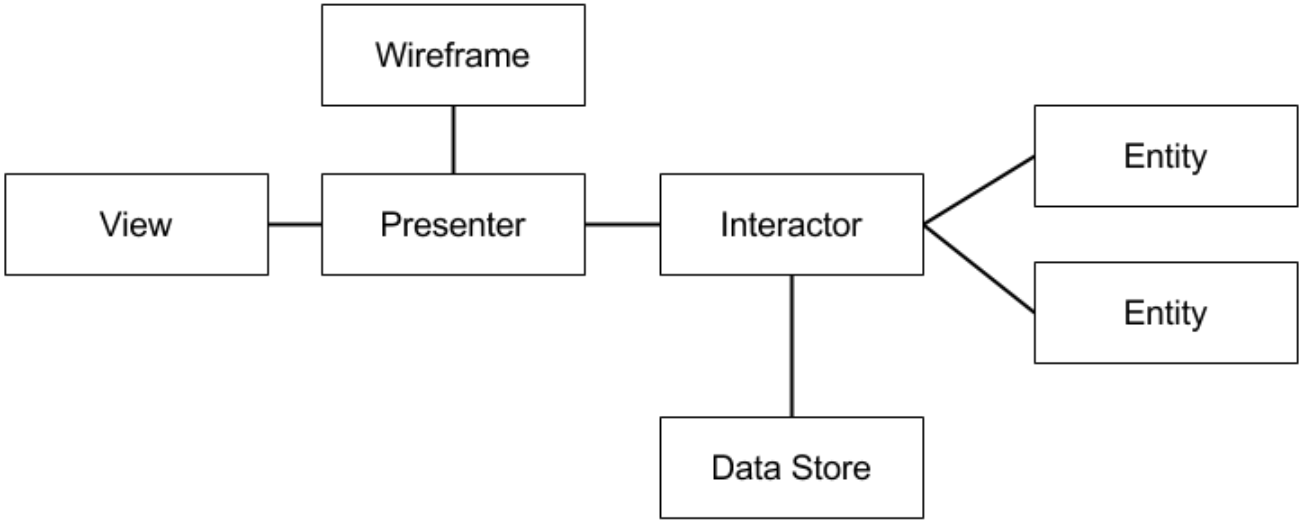
Exibe o que é dito pelo Presenter e retransmite a entrada do usuário de volta para o Presenter.

Abstrai a existência de UILabel, UIButton, etc, criando uma camada de alto nível para o Presenter.

O Presente determina quando o conteúdo deve ser exibido e o View como ele deve ser exibido.

Quanto o usuário interage a View emite um sinal em um método e o Presenter manipula este método criando uma ação.

VIPER


Wireframe
View
Presenter
Interactor
Entity
Entity
Data Store

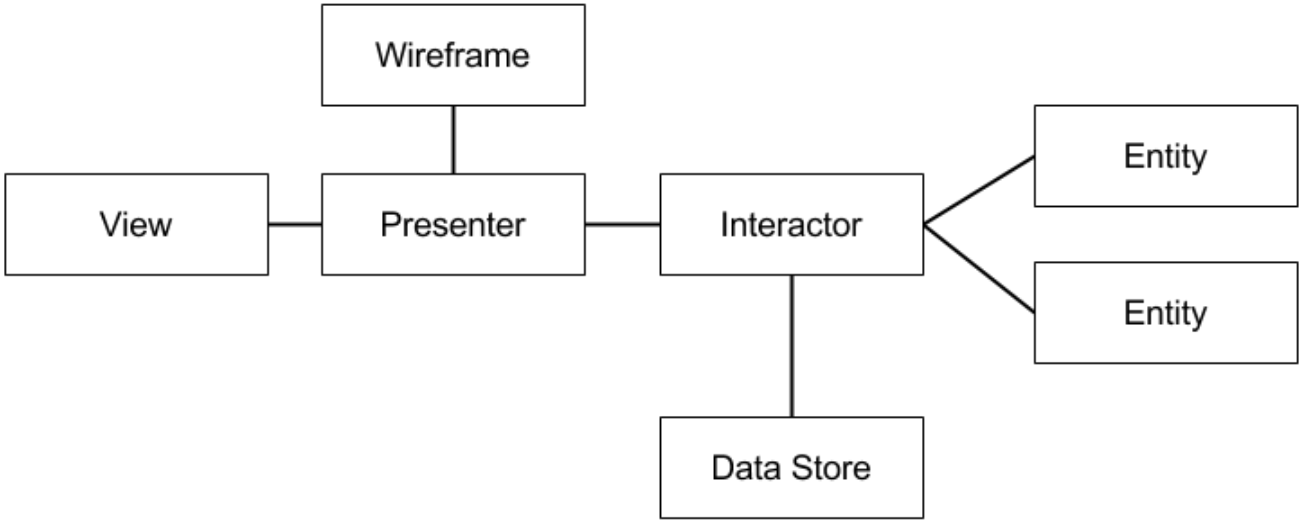
Interactor

Contém a lógica de negócio conforme especificado por um caso de uso.

Um Interactor representa um único caso de uso no aplicativo. Ele contém a lógica de negócios para manipular a Entity.

A atividade de um interactor é independente de qualquer UI

VIPER


Wireframe
View
Presenter
Interactor
Entity
Entity
Data Store

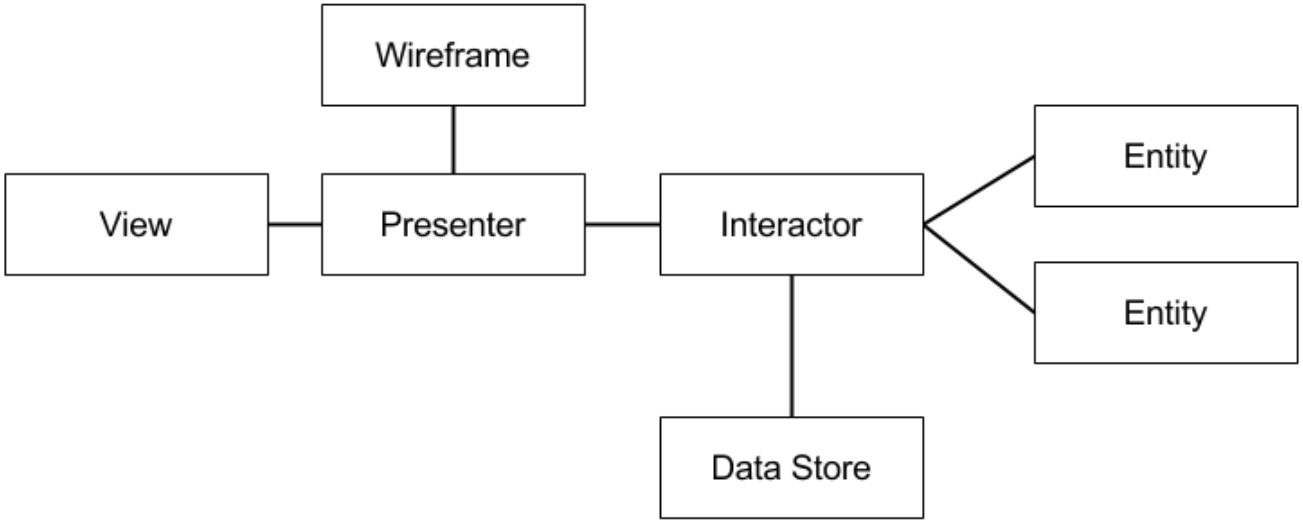
Presenter

Contém a lógica de visualização para preparar o conteúdo para exibição conforme recebido pelo Interactor e para reagir às entradas do usuário solicitando novos dados do interactor.

Consiste principalmente de lógica UI interagindo com o Wireframe para apresentar a interface e reunir todas as entradas de interação do usuário.

As entidades recebidas no Presenter pelo Interactor são estruturas de dados simples que não tem comportamento, isto impede qualquer processamento por parte do Presenter. Por fim os dados recebidos para a exibição.

VIPER


Wireframe
View
Presenter
Interactor
Entity
Entity
Data Store

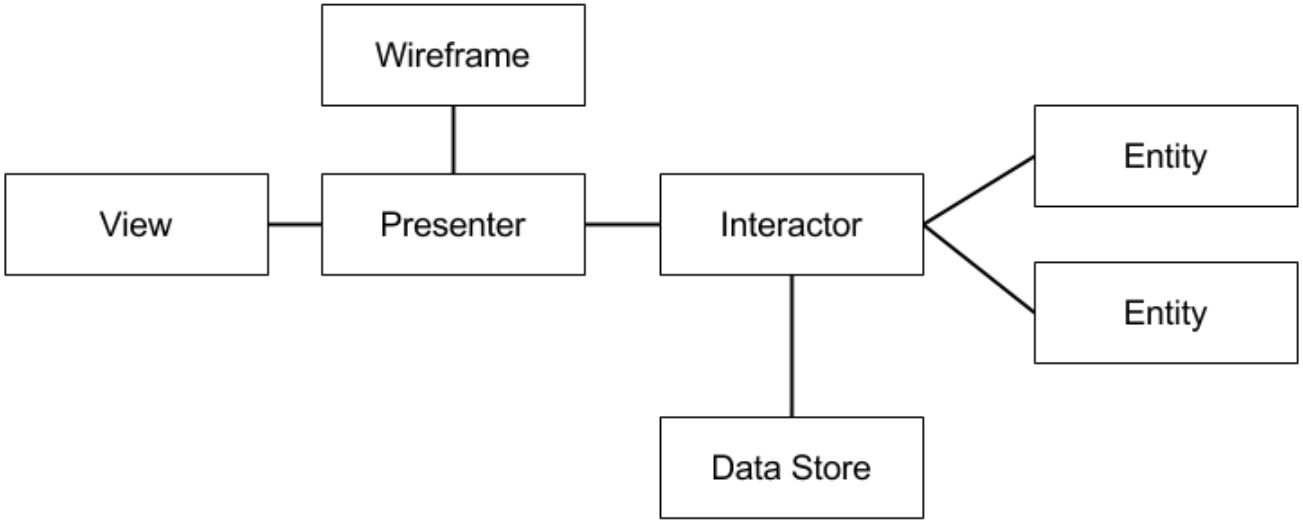
Entity

Contém objetos de modelo básicos usados pelo Interactor.

As entidades são manipuladas pelo interactor e somente por ele.

O Interactor nunca passa uma Entity para o Presenter.

VIPER


Wireframe
View
Presenter
Interactor
Entity
Entity
Data Store

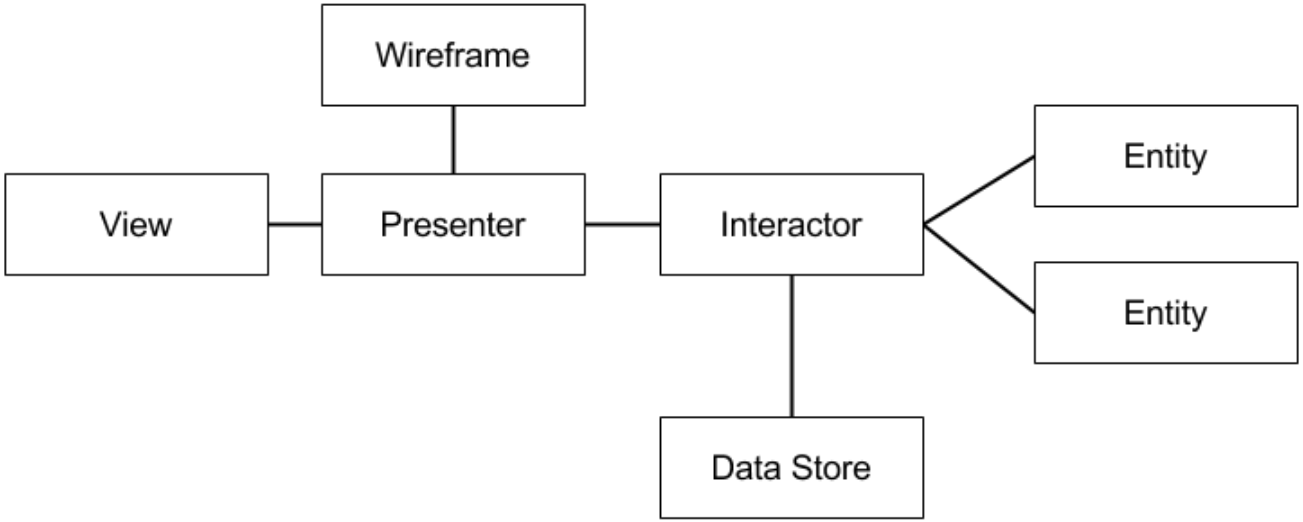
Routing

Contém a lógica de navegação para descrever quais telas são mostradas em qual ordem.

As rotas são definidas dentro do Wireframe, a responsabilidade do roteamento é compartilhada entre dois objetos Presenter e Wireframe.

Presenter sabe quando navegar para outra tela, enquanto o Wireframe sabe como navegar.

VIPER


Wireframe
View
Presenter
Interactor
Entity
Entity
Data Store

VIPER

http://mutualmobile.github.io/blog/2013/12/04/viper-introduction/

Obrigado

Denis Oliveira

Desenvolvedor iOS na Beblue

denis.oliveira@beblue.com.br