VIPER

architecture

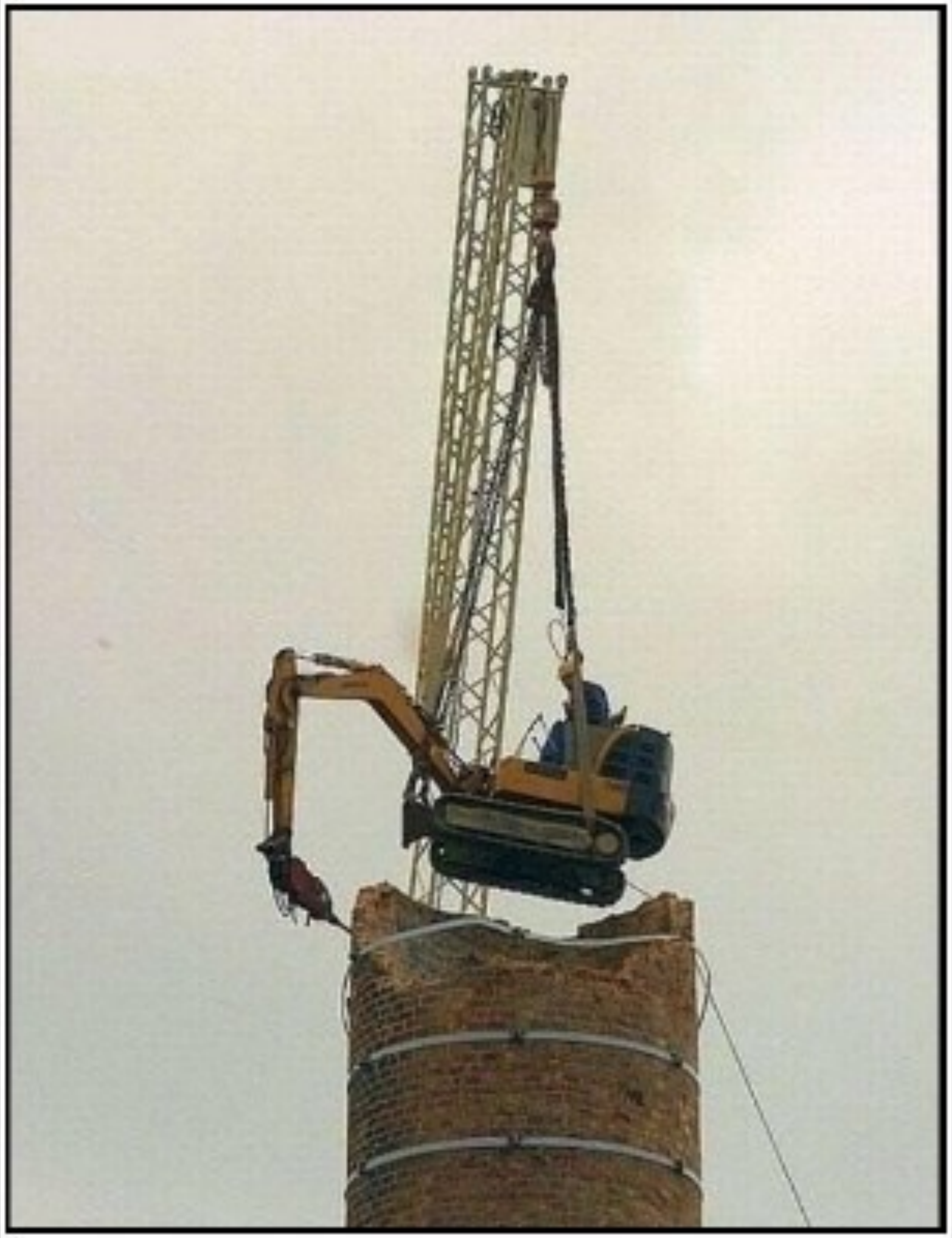
Options

- MVC (Apple-style)
- MVP
- MVVM

MV(C/P/VM) approaches

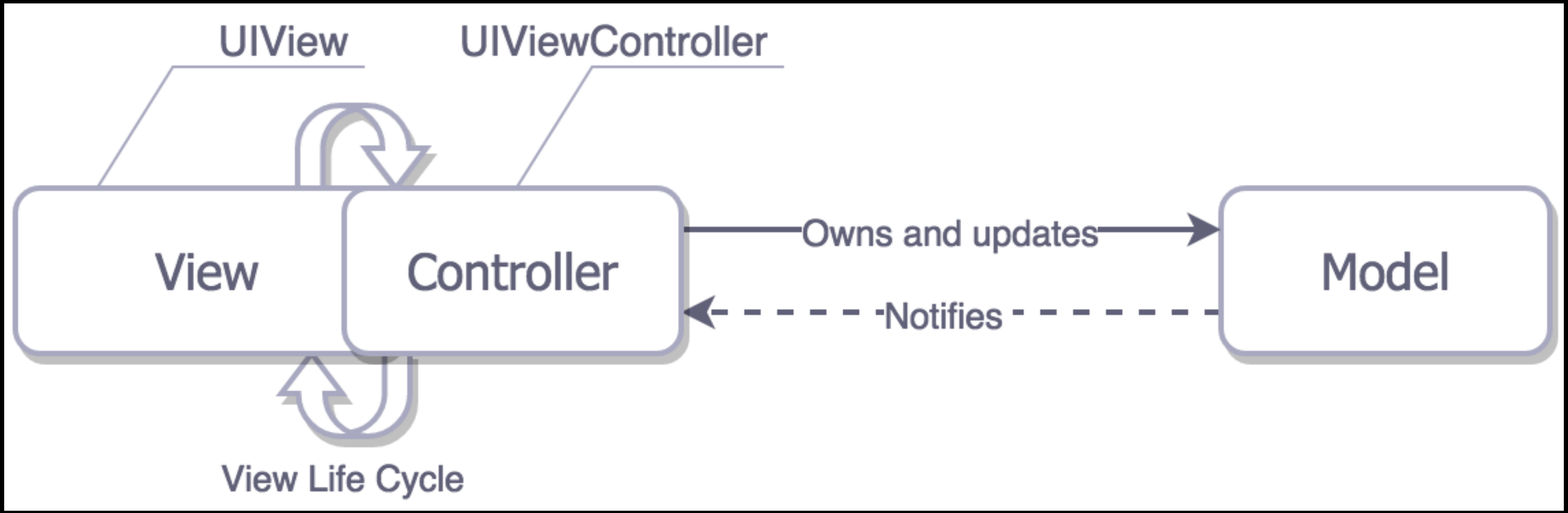
- Model – domain entities or data access layer (“Person” or “PersonDataProvider” classes)
- View – presentation tier (everything that begins with UI-)
- Controller/Presenter/View Model – “glue” or mediator that connects together View and Model

- a lot of unstructured code
- monster files (+1000 lines)
- baffling complexity
- untestable code



TRUST ME

I'm an Engineer



"Apple-style" MVC

Why should you think about architecture?

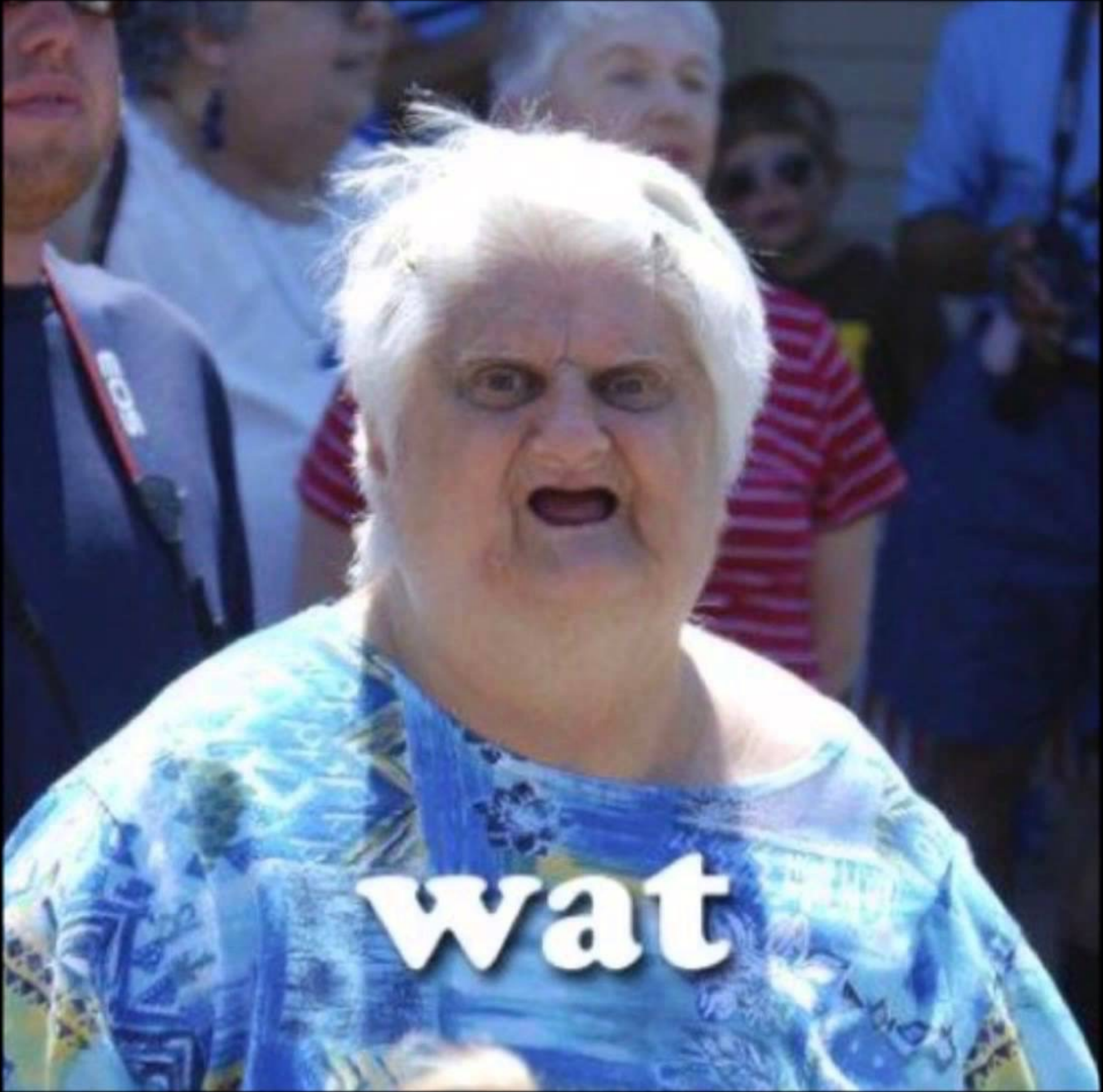
- balanced distribution of the responsibilities among entities with strictly defined roles
- testability
- implementation speed and support of the existing code

Clean architecture

Clean architecture

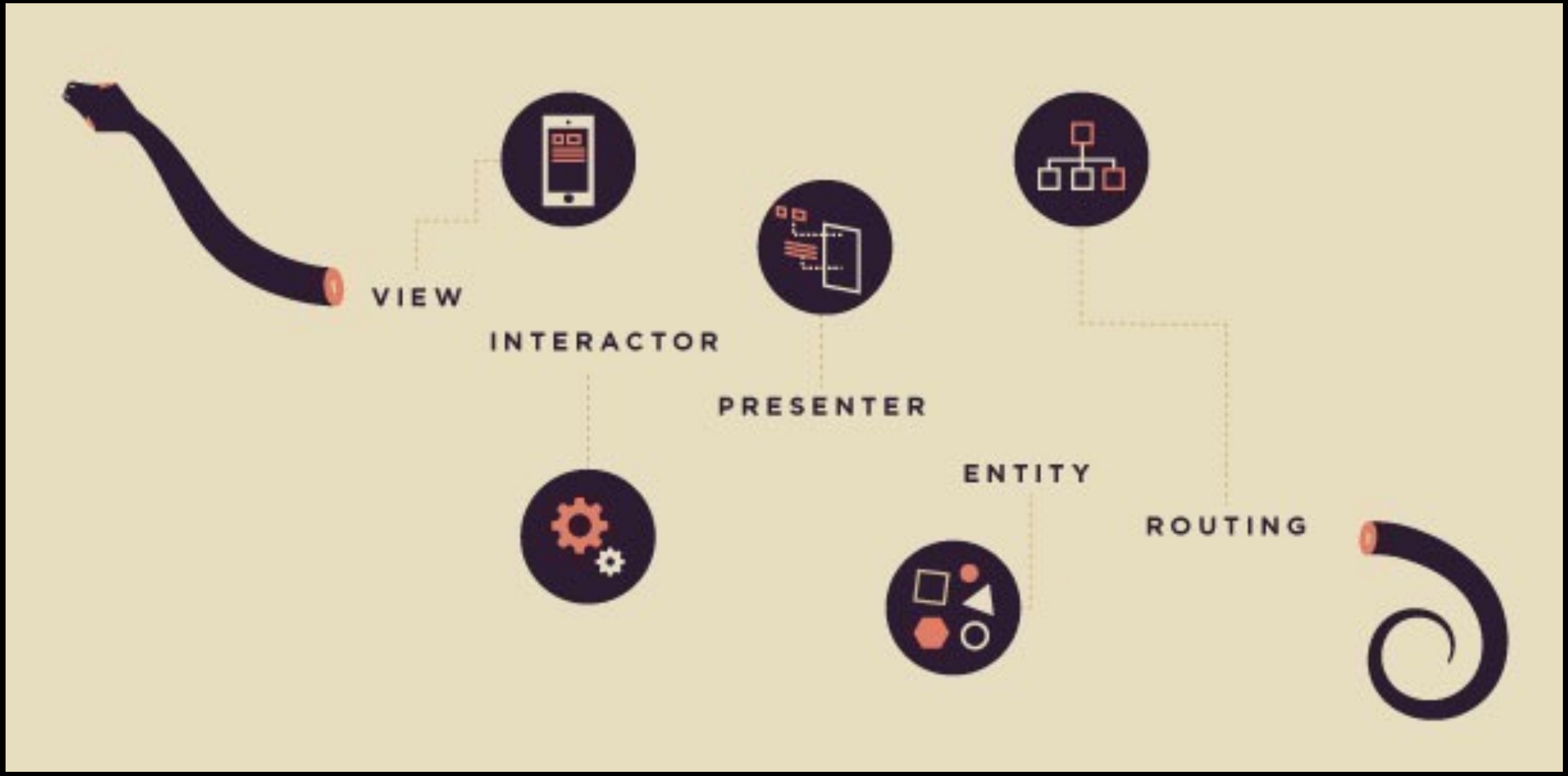
- independent from frameworks
- testable
- independent from UI
- independent from database
- independent from external entities

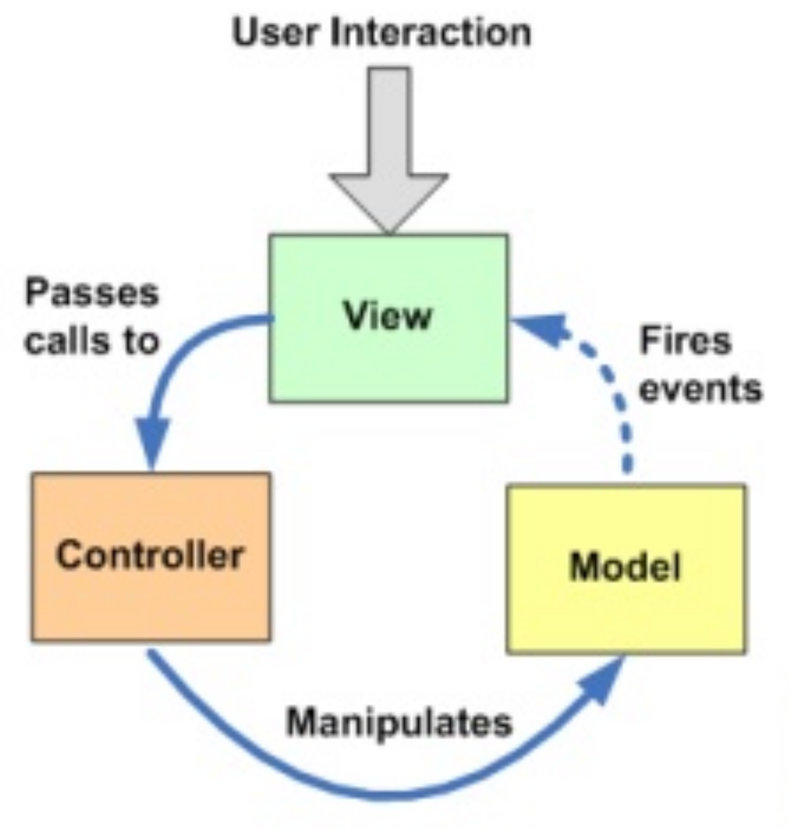
VIPER


wat

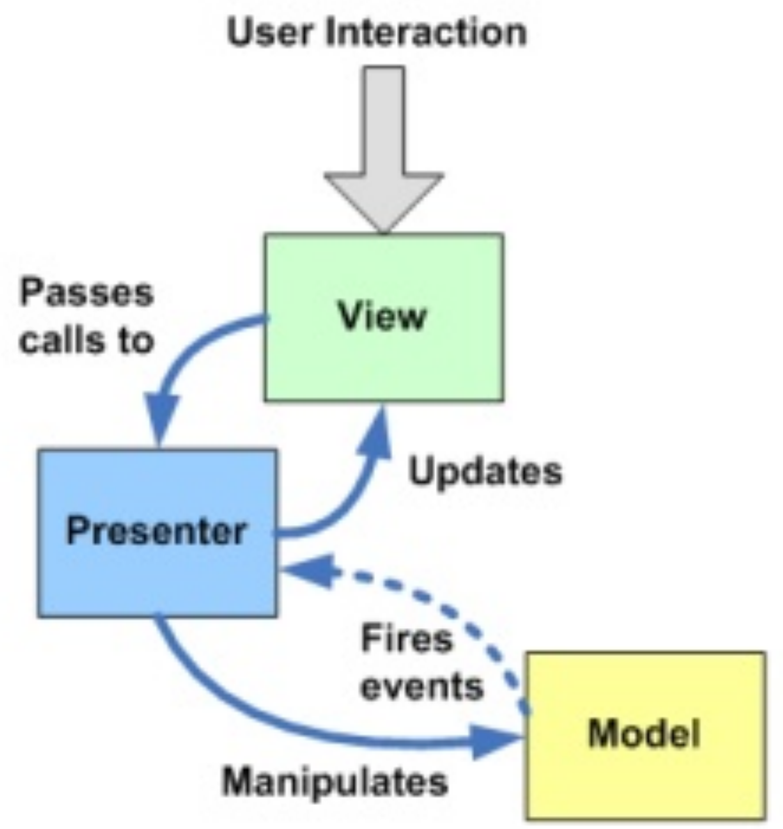
What IS Viper?

- View
- Interactor
- Presenter
- Entity
- Routing

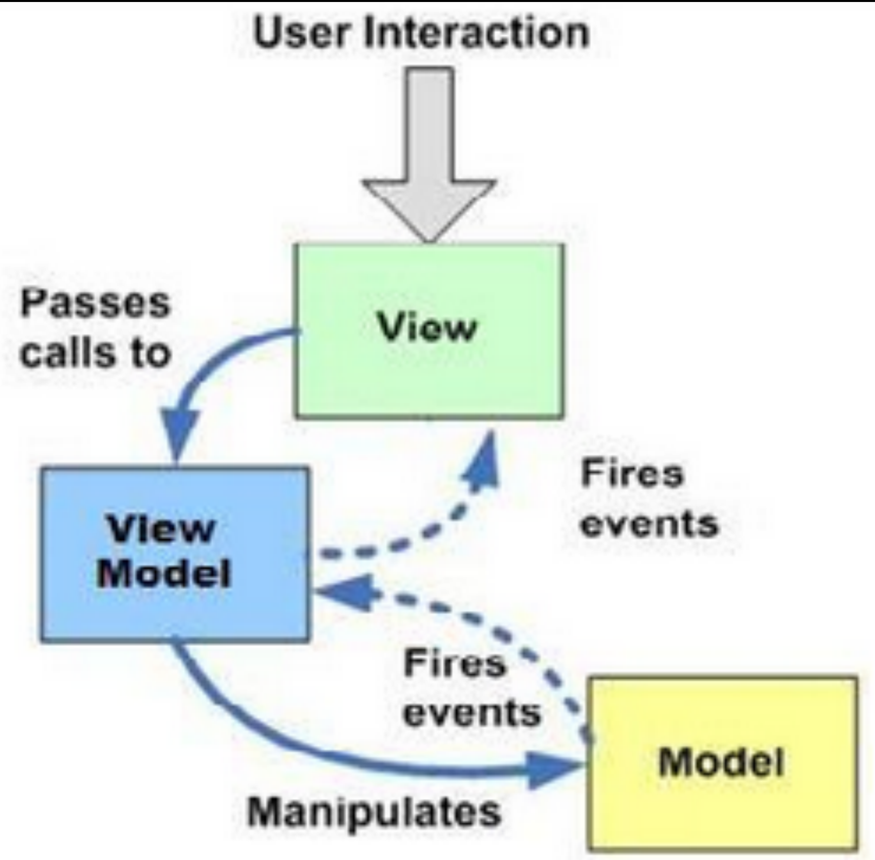




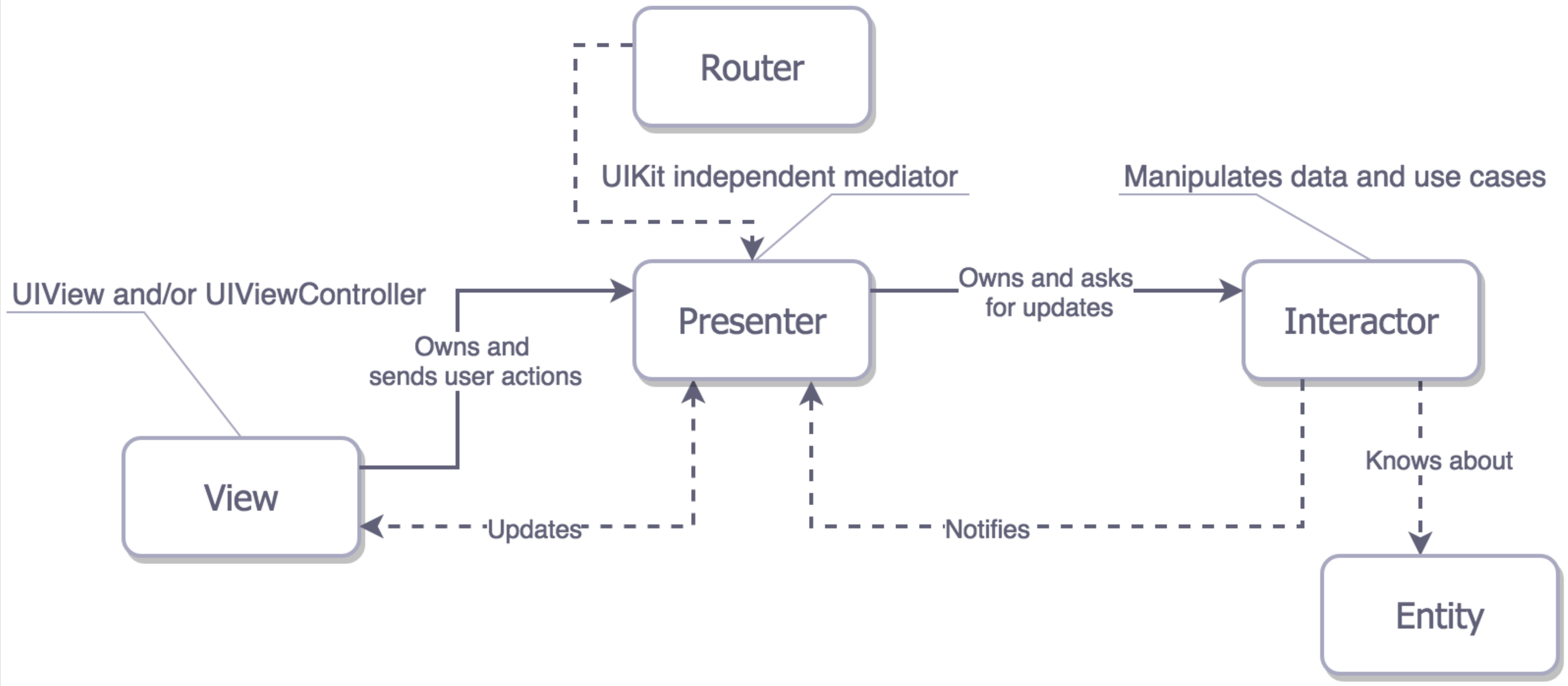
Model-View-Controller

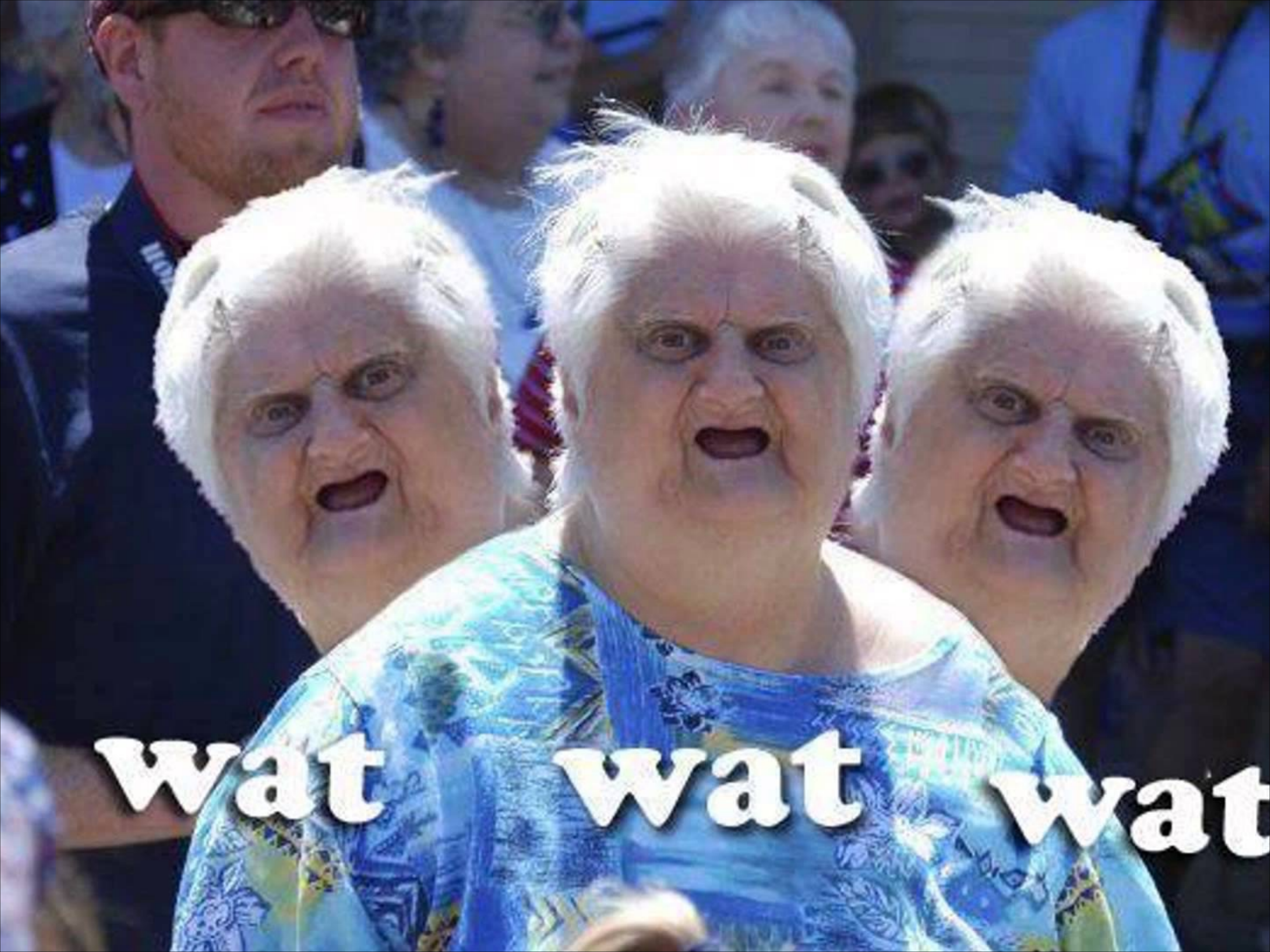


Model-View-Presenter



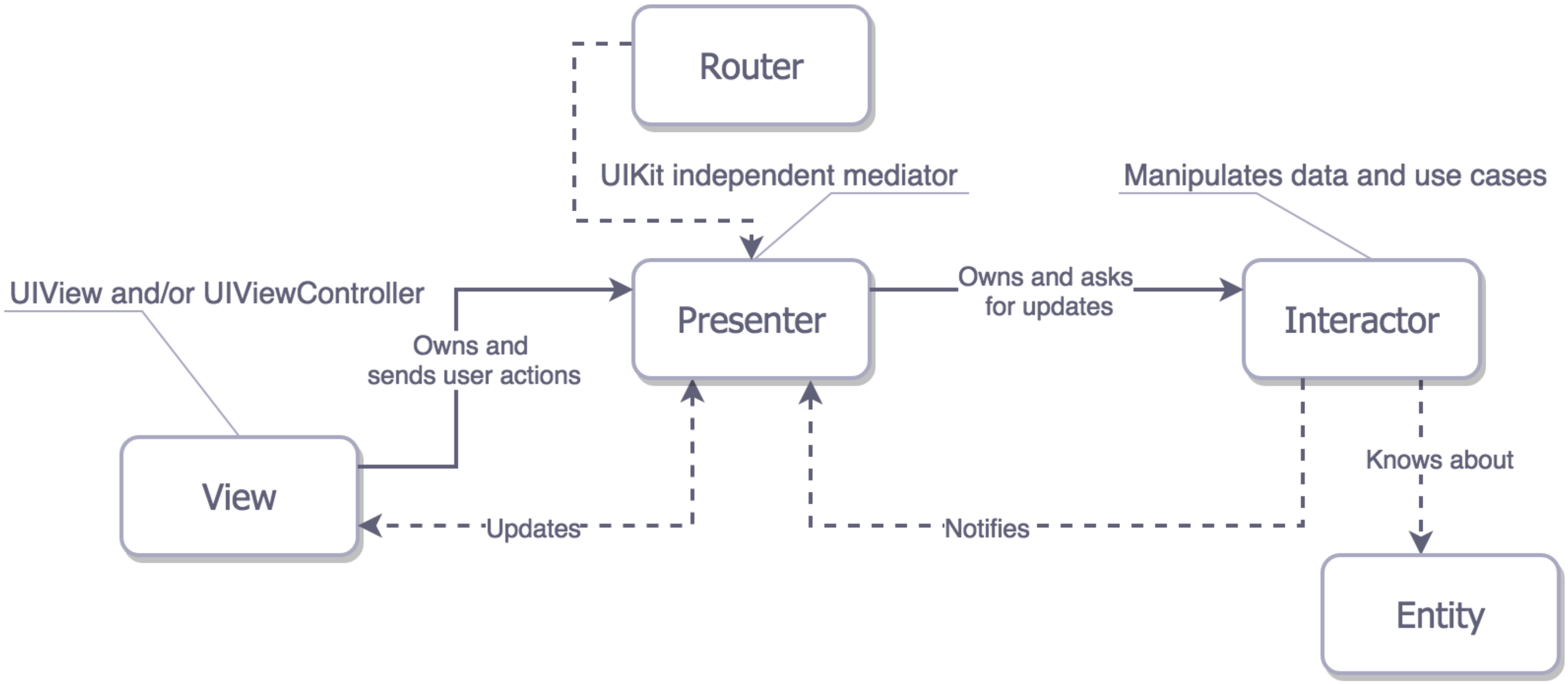
Model - View - ViewModel


Router
UIKit independent mediator
Manipulates data and use cases
UIView and/or UIViewController
Presenter
Owns and asks for updates
Interactor
Owns and sends user actions
Knows about
View
Updates
Notifies
Entity


wat
wat
wat

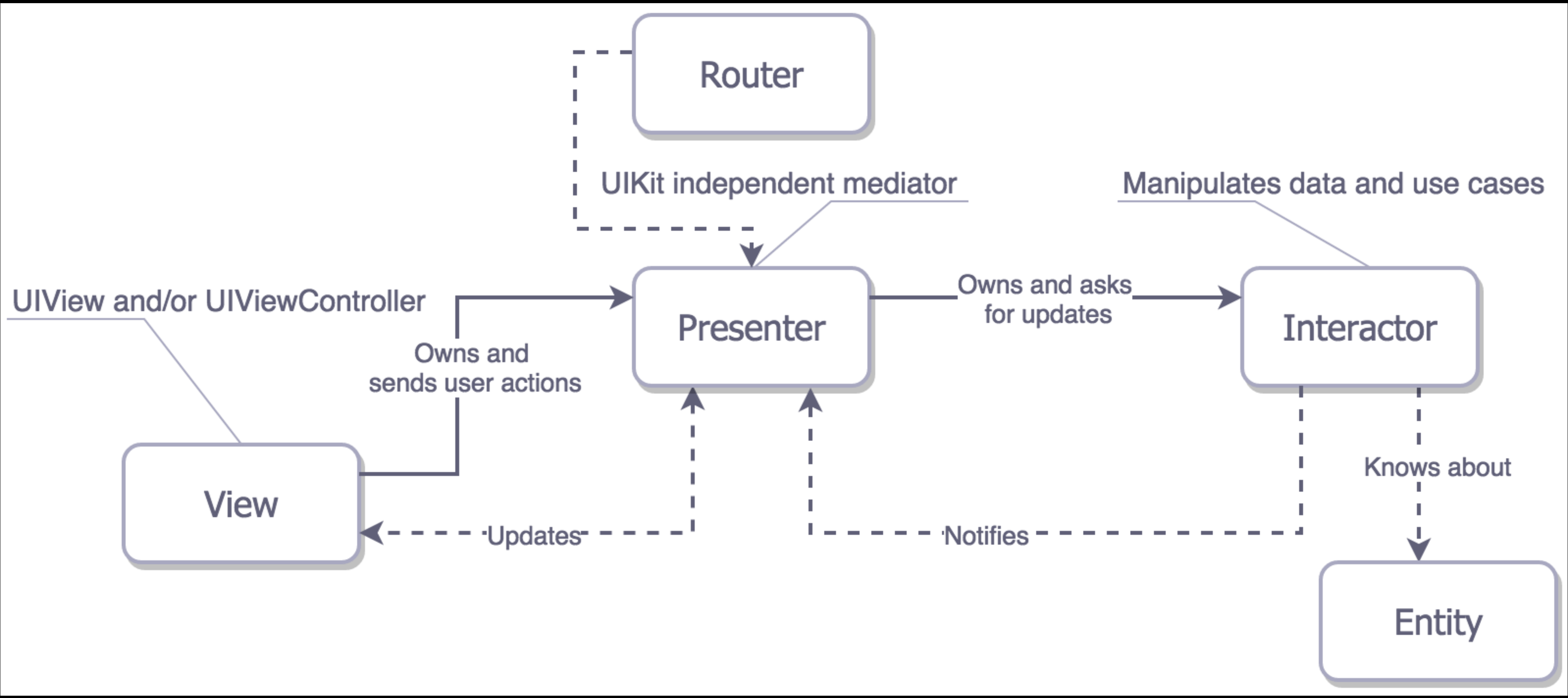
View

deals with data presentation and notifies the Presenter about user’s actions. It’s absolutely passive, never asks for the data by itself, only receives it from the Presenter


Router
UIKit independent mediator
Manipulates data and use cases
UIView and/or UIViewController
Presenter
Owns and asks for updates
Interactor
Owns and sends user actions
View
Updates
Notifies
Knows about
Entity

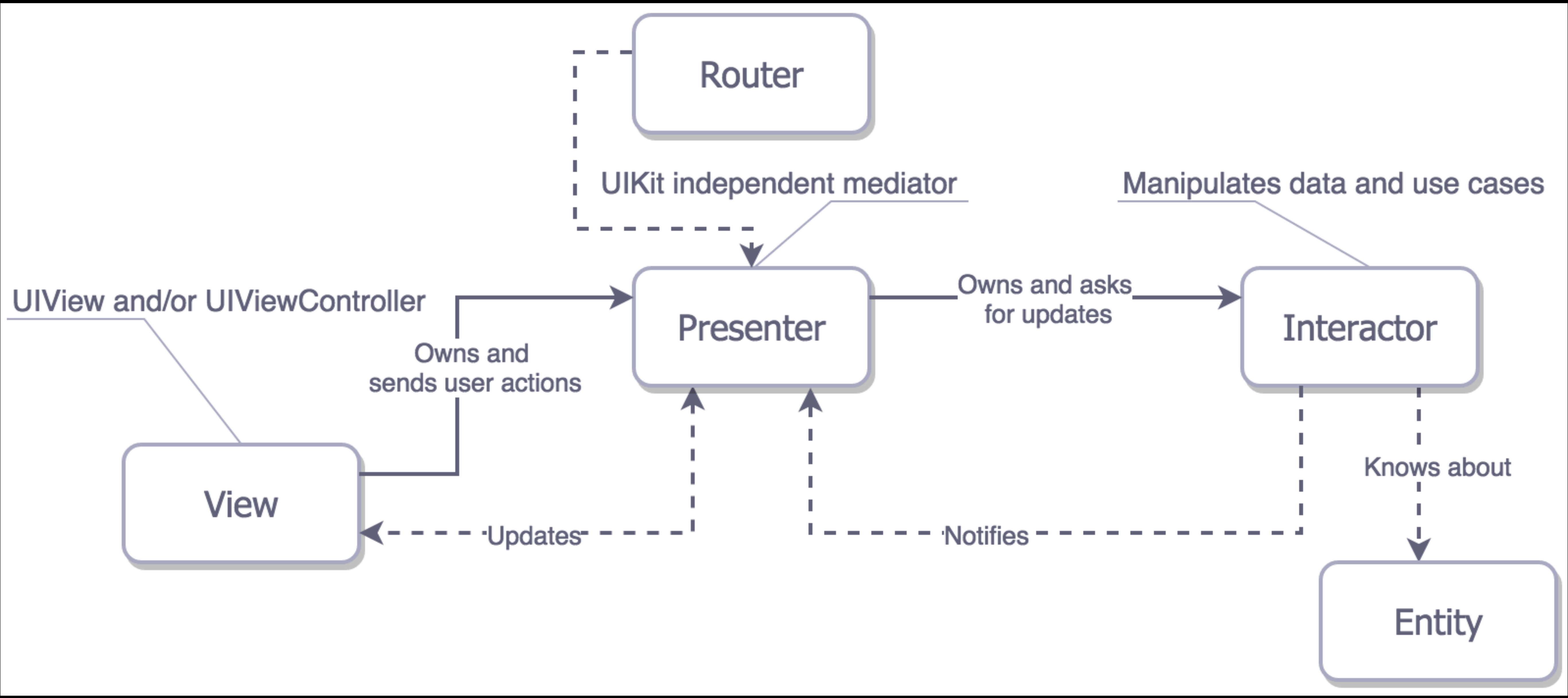
Interactor

contains all business logic needed for a module


Router
UIKit independent mediator
Manipulates data and use cases
UIView and/or UIViewController
Presenter
Owns and asks for updates
Interactor
Owns and sends user actions
Knows about
View
Updates
Notifies
Entity

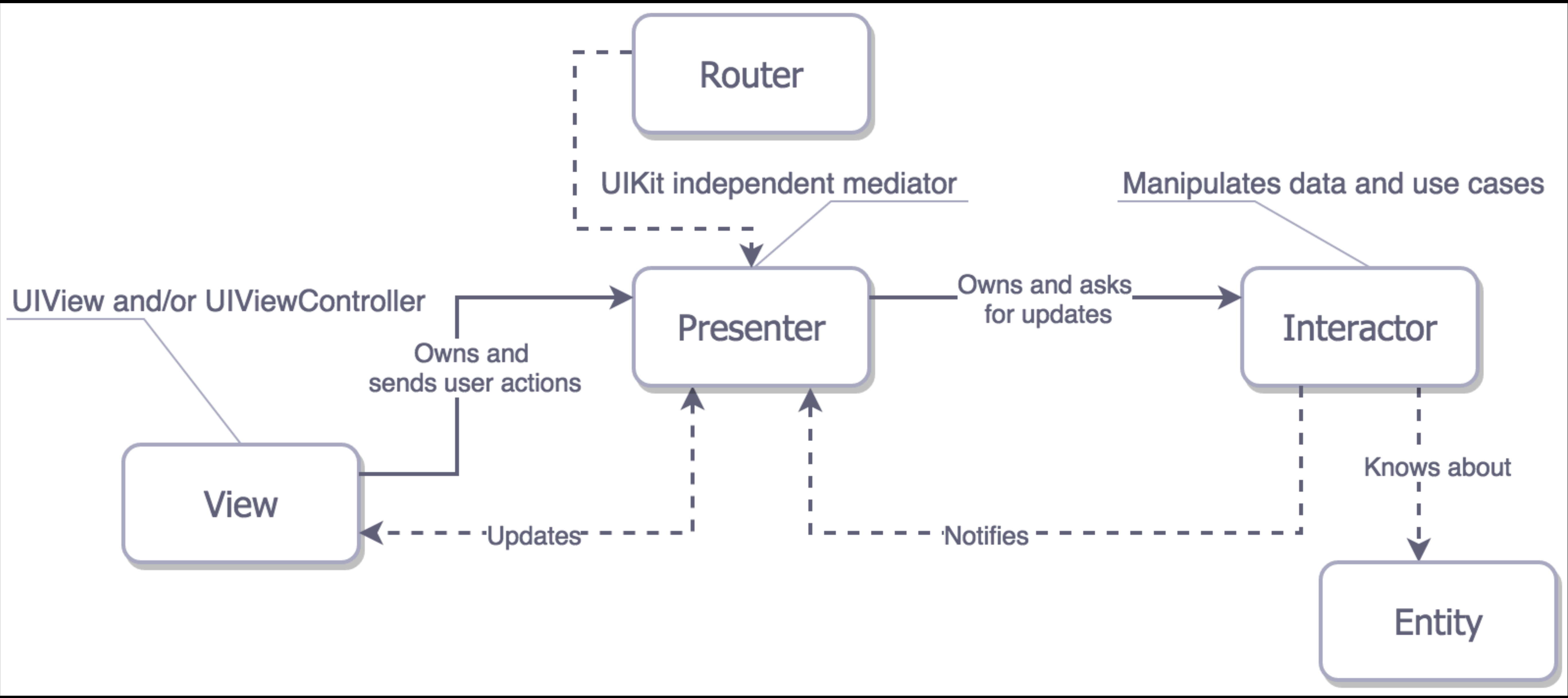
Presenter

receives information from View about user’s actions and transforms it into Router & Interactor requests as well as receives data from the Interactor, prepares it and sends to View for displaying


Router
UIKit independent mediator
Manipulates data and use cases
UIView and/or UIViewController
Presenter
Owns and asks for updates
Interactor
Owns and sends user actions
View
Knows about
Updates
Notifies
Entity

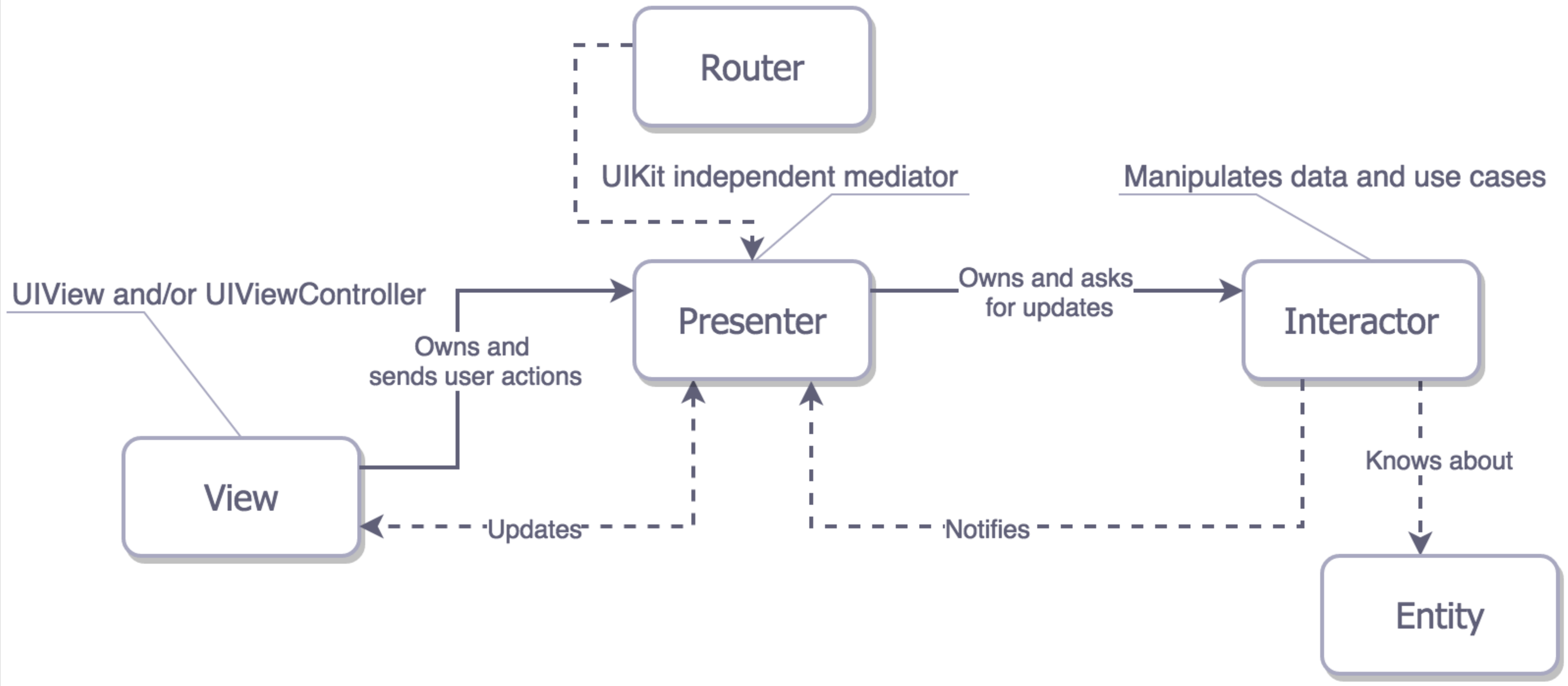
Entity

domain models that don’t contain any business logic


Router
UIKit independent mediator
Manipulates data and use cases
UIView and/or UIViewController
Presenter
Owns and asks for updates
Interactor
Owns and sends user actions
View
Knows about
Updates
Notifies
Entity

Router

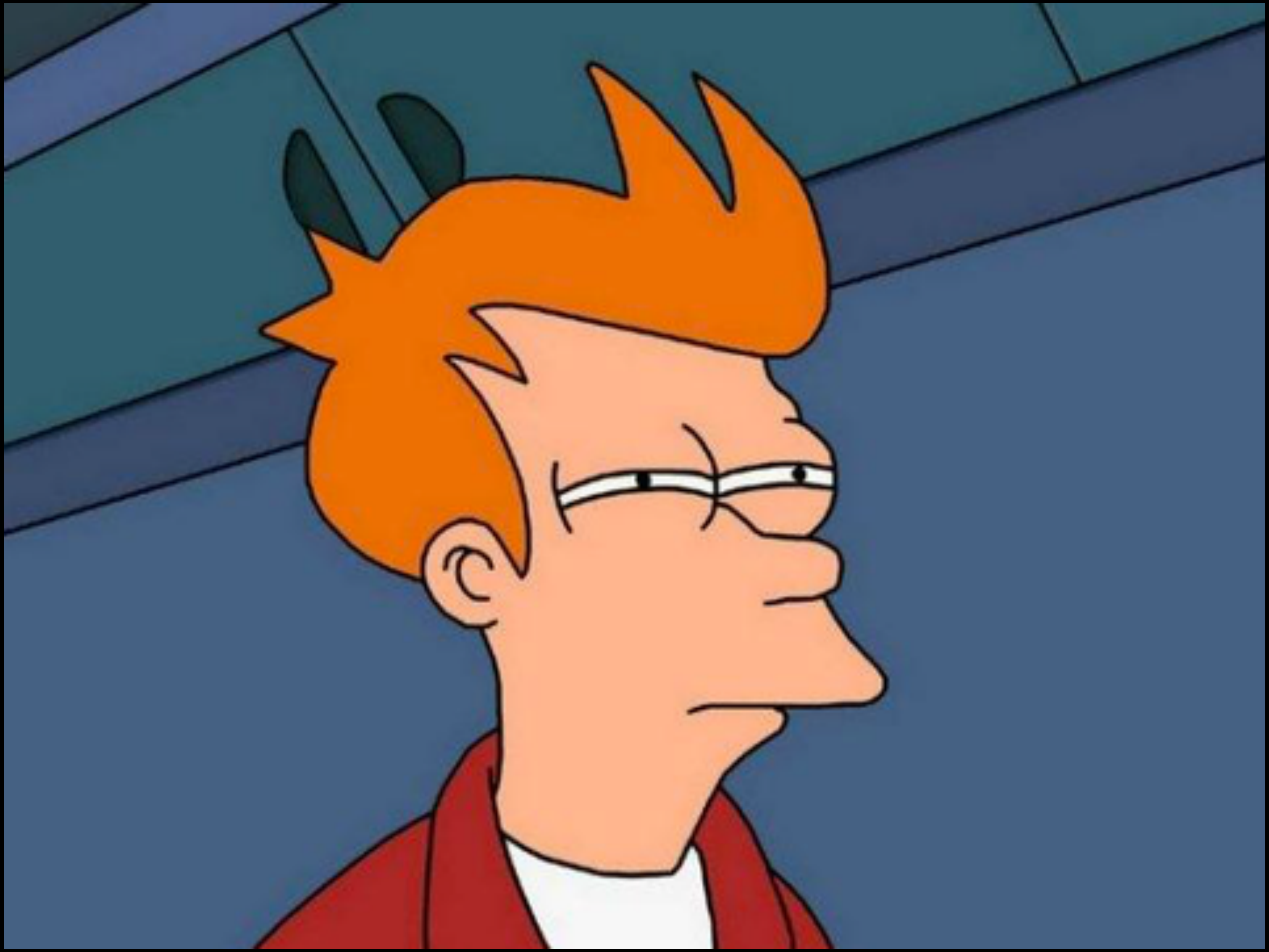
deals with navigation between modules



Router
UIKit independent mediator
Manipulates data and use cases
UIView and/or UIViewController
Presenter
Owns and asks for updates
Interactor
Owns and sends user actions
View
Updates
Notifies
Knows about
Entity

Main principles

- protocols
- dependency injection
- interface segregation



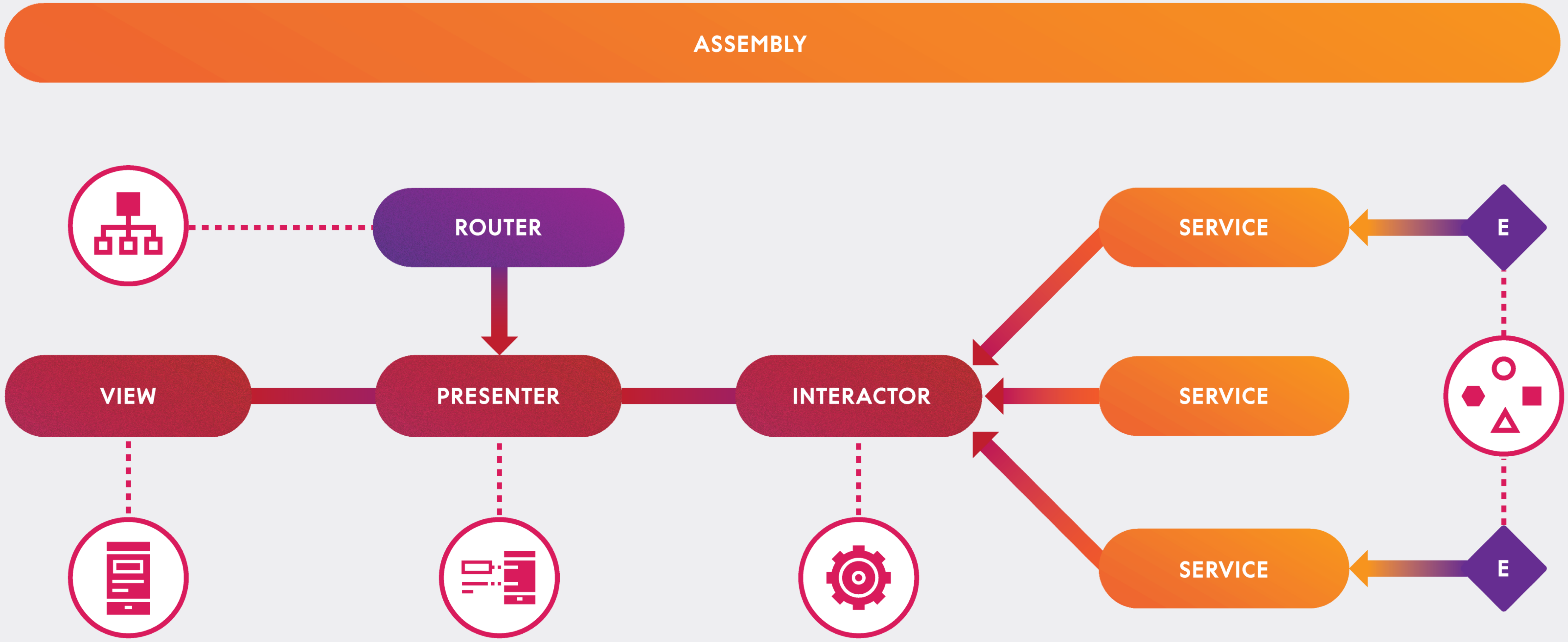
BUT…

We’ve lost something

- Wireframe knows too much
- Interactors are still difficult
- ViewControllers process tables and collections
- doesn’t have well defined communication between modules

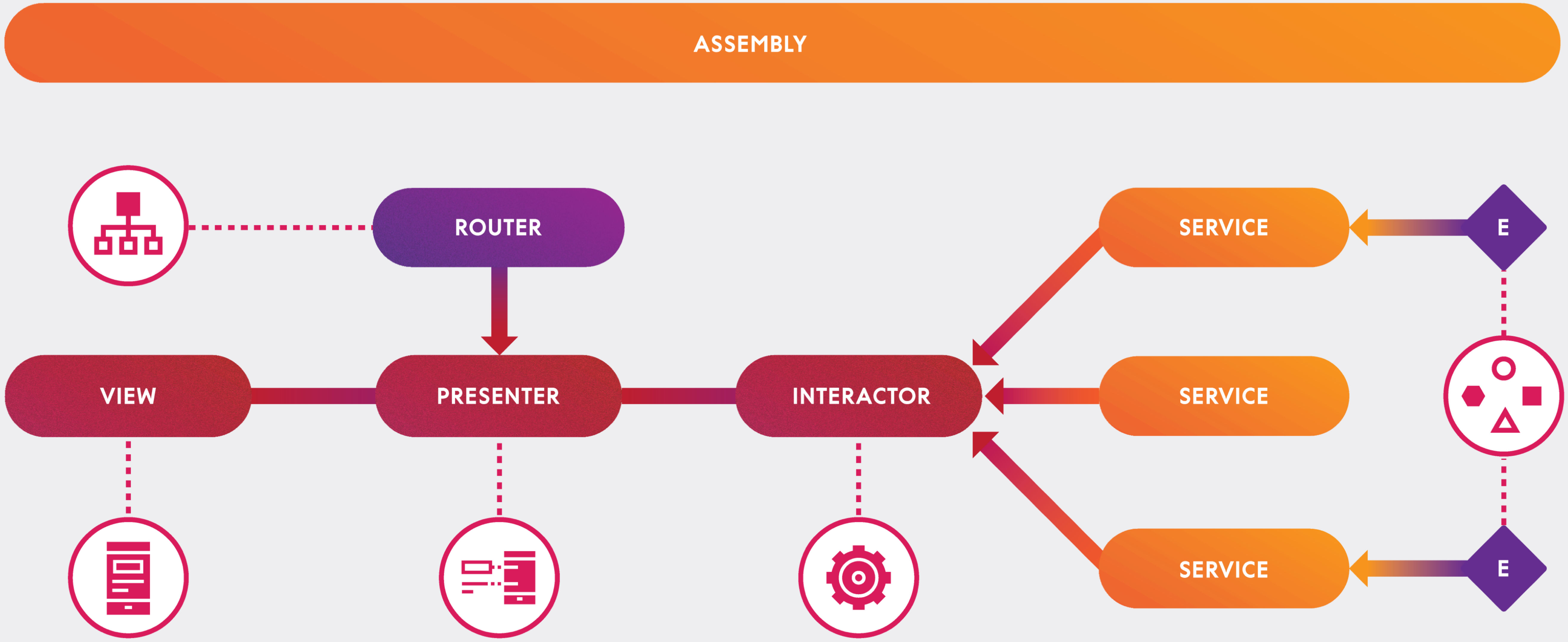
There is a solution

rambler-style VIPER


ASSEMBLY
ROUTER
SERVICE
E
VIEW
PRESENTER
INTERACTOR
SERVICE
SERVICE
E

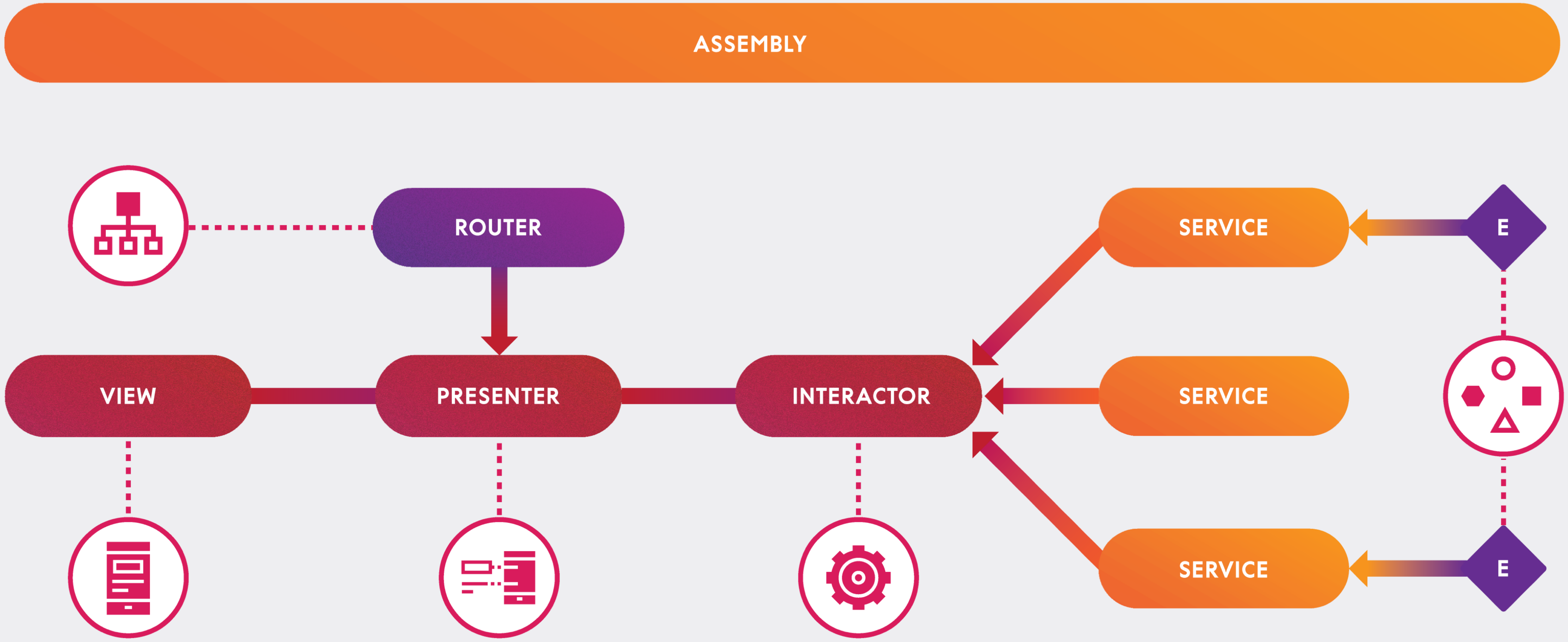
Problem No 1: Wireframe

- splits up into two entities: Router and Assembly
- Router - transitions between modules
- Assembly – gets components together with dependency injection


ASSEMBLY
ROUTER
VIEW
PRESENTER
INTERACTOR
SERVICE
SERVICE
SERVICE
E
E

Problem No 2: Interactor

- introduce the additional services tier
- each of the services is responsible for dealing with a certain type of domain models
- Interactor becomes a facade for services


ASSEMBLY
ROUTER
SERVICE
E
VIEW
PRESENTER
INTERACTOR
SERVICE
SERVICE
E

Problem No 3: ViewControllers

- remove a logic which is not suitable to the View role into separate tier called DataDisplay
- these objects implement the methods for UITableViewDelegate and UITableViewDataSource as well as their versions for collections

Problem No 4: Transferring data between modules

- two protocols **ModuleInput** and **ModuleOutput**

a LOT of files

Code generation

Generamba, VIPER gen, Boa

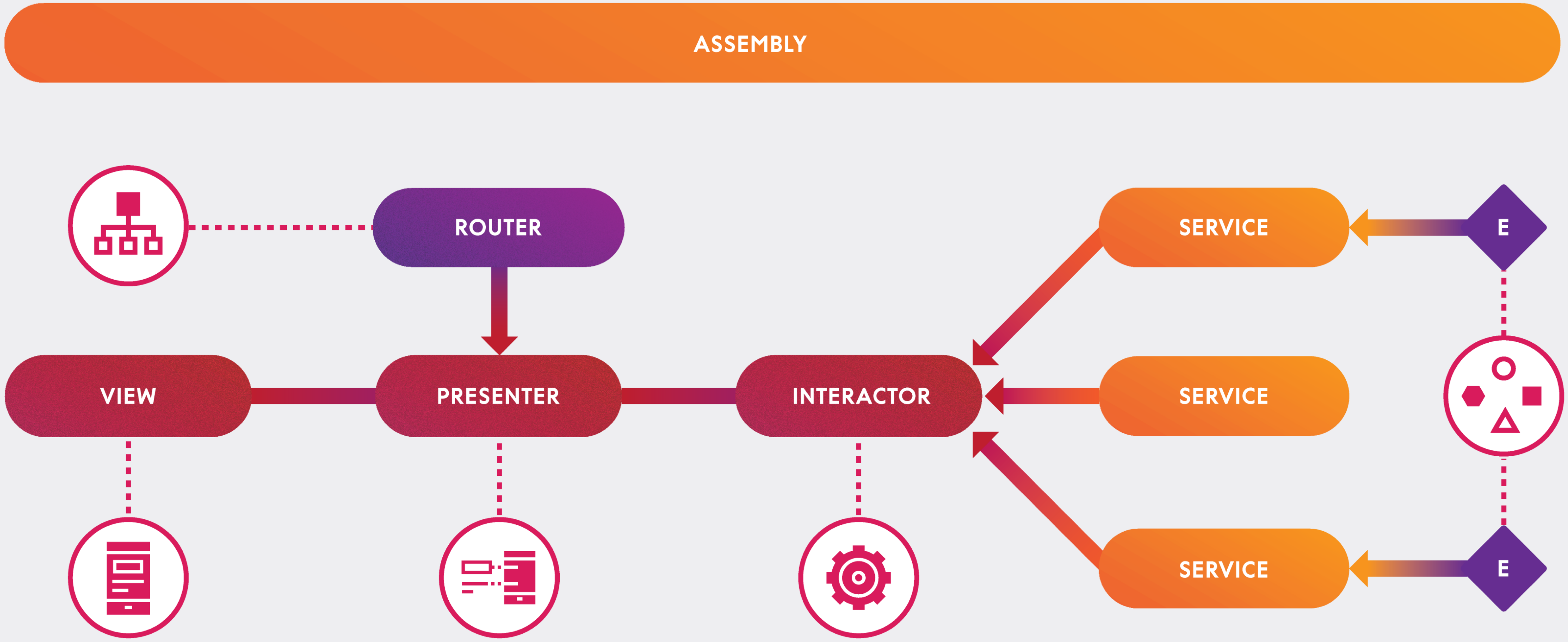
WAIT A MINUTE

This isnt my RPG

Testings

It is all about mocks

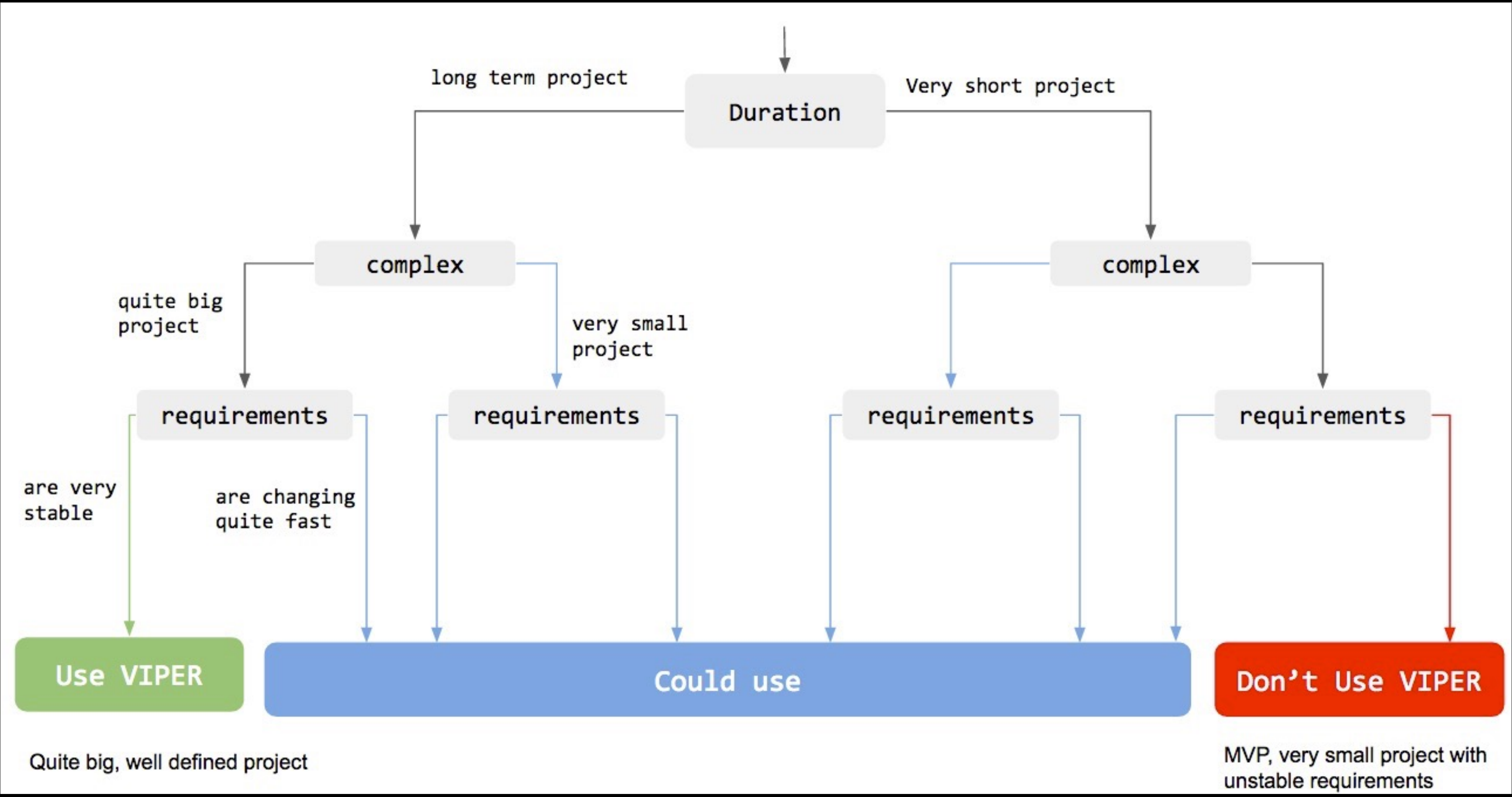
- services
- interactors
- presenters
- views
- routers
- assemblies


ASSEMBLY
ROUTER
SERVICE
E
VIEW
PRESENTER
INTERACTOR
SERVICE
SERVICE
E

Some conclusions

- “lighter”, stricter classes
- excellent scalability of the tasks among developers
- no excuse for making tests

VIPER vs. MV(C/P)



"Everything should be made as simple as possible, but no simpler"

–Albert Einstein

Resources

- architecture patterns
- objc.io/viper
- clean architecture
- rambler-viper
- viper

Questions?

Thank you!