Architectural Patterns

MVC, MVVM, VIPER

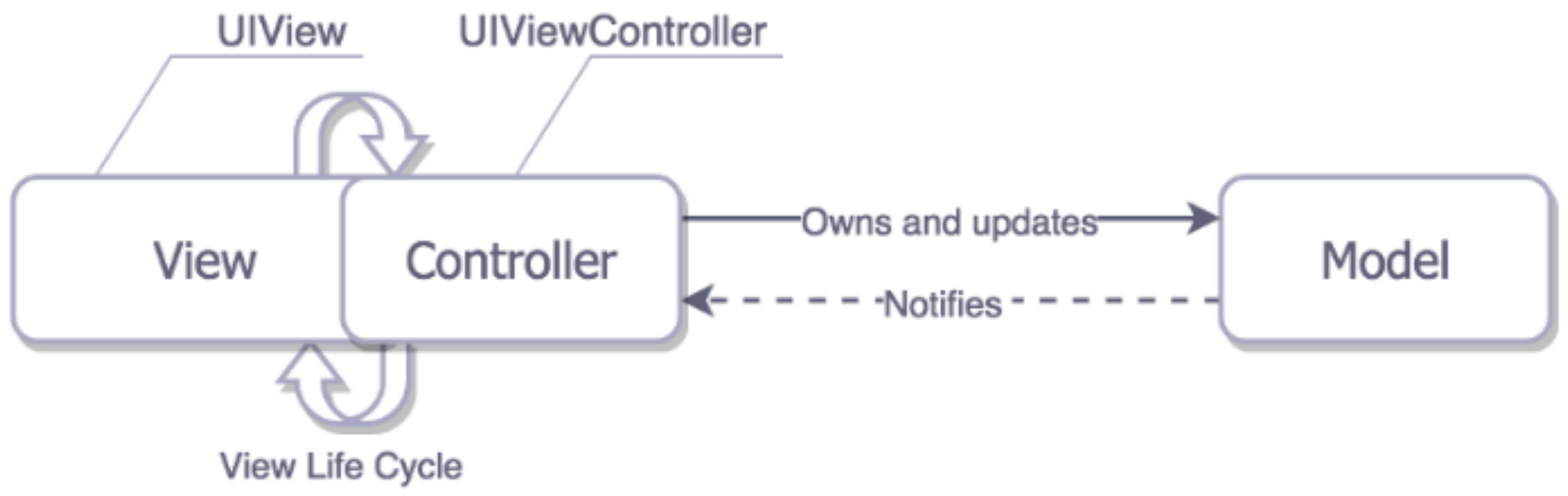
MVC

Model - View - Controller

MVC architecture is using for small projects and UI orientated applications. **MVC** is easy to learn, fast to implement and has small count of files and classes.

- **Model**—responsible for the domain data or a data access layer which manipulates the data.
- **View** —responsible for the presentation layer (**GUI**), for iOS environment think of everything starting with ‘**UI**’ prefix.
- **Controller**—the glue or the mediator between the **Model** and the **View**, in general responsible for altering the **Model** by reacting to the user’s actions performed on the **View** and updating the **View** with changes from the **Model.**

MVC


UIView
UIViewController
View
Controller
Owns and updates
Notifies
Model
View Life Cycle

MVC

```swift
class GreetingViewController : UIViewController { //View + Controller

    //MARK: - Private properties
```

views

```swift
    private let showGreetingButton = UIButton()
    private let greetingLabel = UILabel()
    private var greetingMessage: String {
        return "Hello" + " " + person.firstName + " " + person.lastName
    }

    //MARK: - Public properties
```

model

```swift
    var person: Person!

    //MARK: - Lifecycle
    override func viewDidLoad() {
        super.viewDidLoad()
        setupShowGreetingButtonAction()
    }

    //MARK: - Private functions
    private func setupShowGreetingButtonAction() {
        showGreetingButton.addTarget(self,
                                     action: #selector(showGreetingMessage),
                                     for: .touchUpInside)
    }

    @objc private func showGreetingMessage() {
        greetingLabel.text = greetingMessage
    }
}
```

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

```swift
class UserTableViewCell: UITableViewCell { // View

    @IBOutlet var userImage: UIImageView!

}
```

MVVM

Model - View - View Model

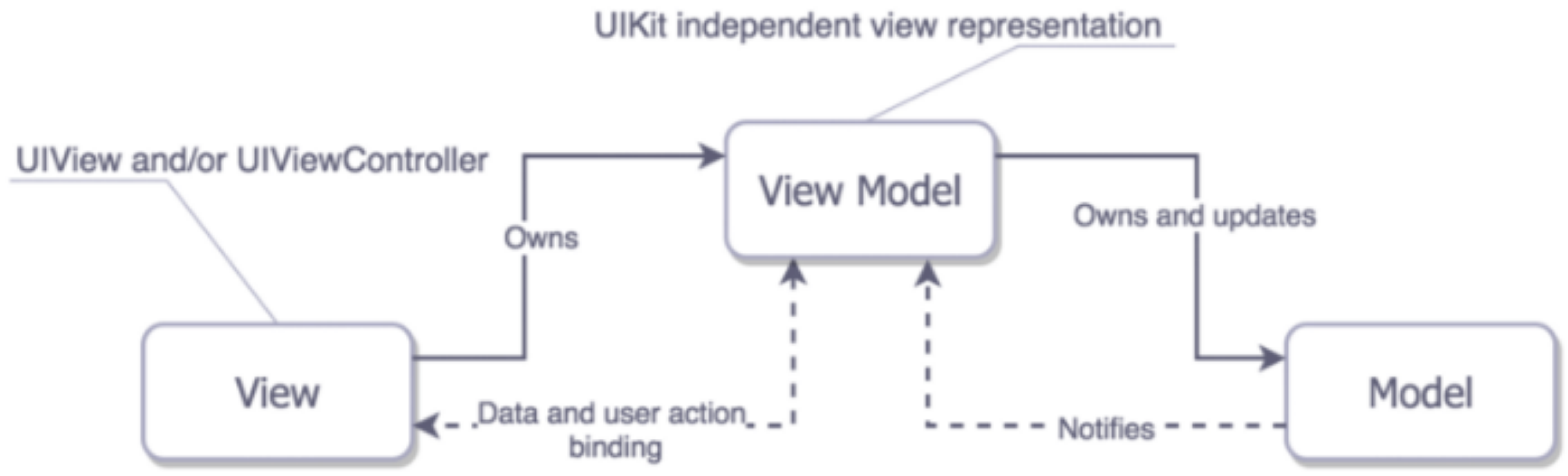
MVVM architecture is using when you need to transform models into another representation for a view.

- **Model**—responsible for the domain data or a data access layer which manipulates the data.

- **View/View Controller** —responsible for the presentation layer (**GUI**), for iOS environment think of everything starting with '**UI**' prefix.

- **View Model**—is the intermediary between the **View/View Controller** and **Model**. **View Model** invokes changes in the **Model** and updates itself with the updated **Model.**

MVVM


UIKit independent view representation
UIView and/or UIViewController
View Model
Owns
Owns and updates
View
Model
Data and user action binding
Notifies

MVVM

```swift
class GreetingViewController : UIViewController { //View + Controller

    //MARK: - Private properties
    private let showGreetingButton = UIButton()
    private let greetingLabel = UILabel()

    //MARK: - Public properties
    var viewModel: GreetingViewModelProtocol! {
        didSet {
            viewModel.greetingDidChange = { [weak self] greeting in
                guard let strongSelf = self else { return }

                strongSelf.greetingLabel.text = greeting
            }
        }
    }

    //MARK: - Lifecycle
    override func viewDidLoad() {
        super.viewDidLoad()
        setupShowGreetingButtonAction()
    }

    //MARK: - Private functions
    private func setupShowGreetingButtonAction() {
        showGreetingButton.addTarget(viewModel,
                                     action: Selector(("showGreeting")),
                                     for: .touchUpInside)
    }
}
```

```swift
protocol GreetingViewModelProtocol: class { //View Model Protocol
    var greetingDidChange: ((String) -> ())? { get set }
    init(person: Person)
    func showGreeting()
}
```

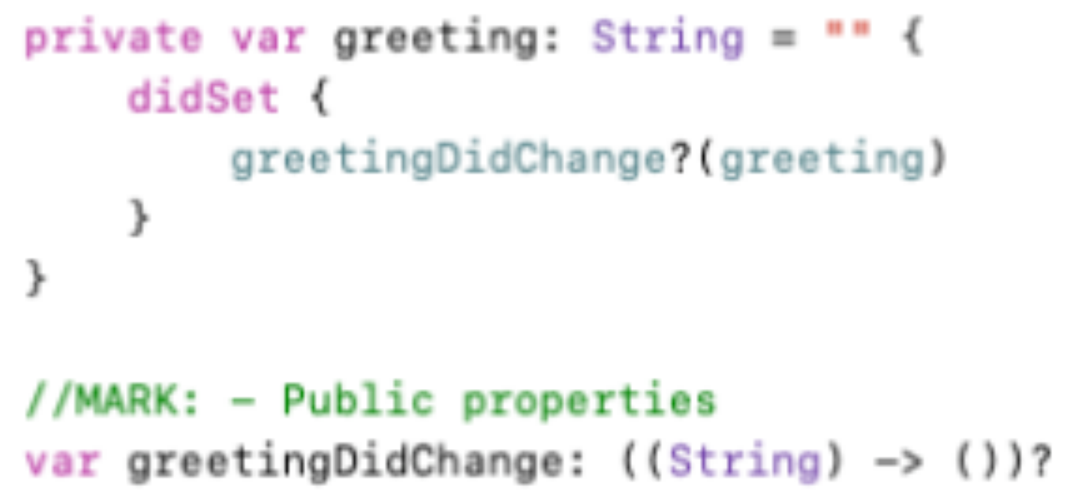
```swift
class GreetingViewModel : GreetingViewModelProtocol { //View Model
    //MARK: - Private properties
    private let person: Person
    private var greeting: String = "" {
        didSet {
            greetingDidChange?(greeting)
        }
    }

    //MARK: - Public properties
    var greetingDidChange: ((String) -> ())?

    required init(person: Person) {
        self.person = person
    }

    //MARK: - Public functions
    func showGreeting() {
        greeting = "Hello" + " " + person.firstName + " " + person.lastName
    }
}
```

MVVM

Binding refers to the flow of information between the **View** and the **View Model**.

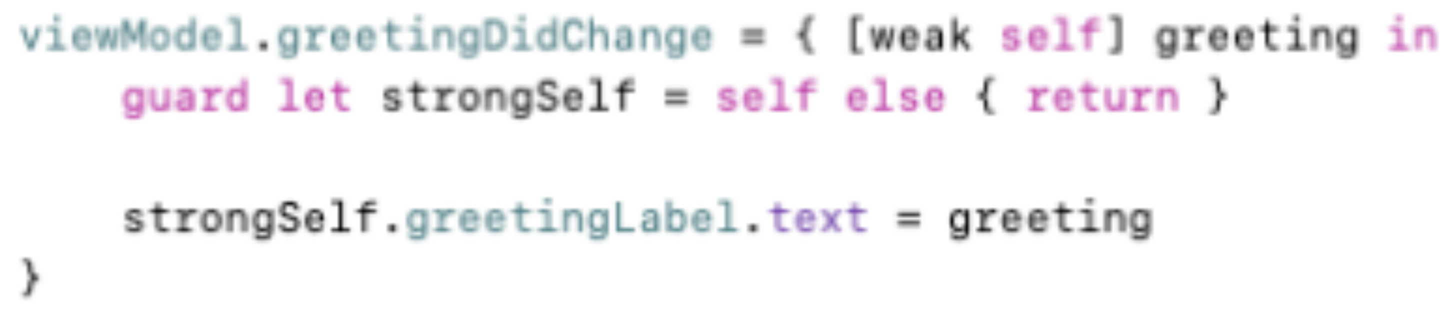
If you change your **View Model** properties then it is reflected on the **View**.

In **View Model**



```swift
private var greeting: String = "" {
    didSet {
        greetingDidChange?(greeting)
    }
}

//MARK: - Public properties
var greetingDidChange: ((String) -> ())?
```

In **View**



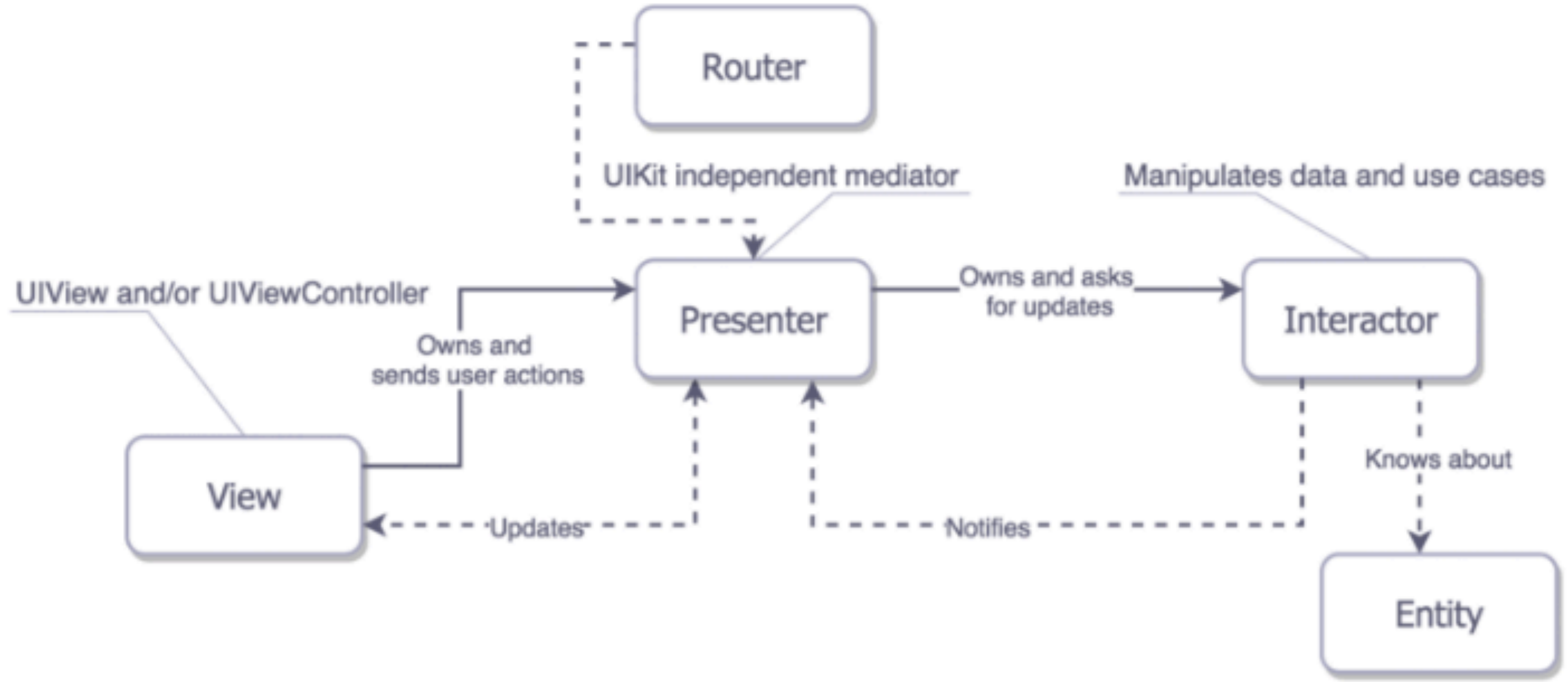
```swift
viewModel.greetingDidChange = { [weak self] greeting in
    guard let strongSelf = self else { return }

    strongSelf.greetingLabel.text = greeting
}
```

VIPER

VIPER is really segmented way to divide responsibilities.

- **View:** Shows the app interface to the user and get their responses. Upon receiving a response **View** alerts the **Presenter**.
- **Interactor:** Has the business logics of an app. Primarily make API calls to fetch data from a source.
- **Presenter:** Gets response from the **View**. Communicates with all the other components. Calls the **Router** for wire-framing, **Interactor** to fetch data, **View** to update the UI.
- **Entity:** Contains plain model classes used by the **Interactor**.
- **Router:** Does the wire-framing. Listens from the **Presenter** about which screen to present and executes that.

VIPER


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

	MVC	MVVM	VIPER
Responsibilities	Entangled	Dispersed	Dispersed
ViewController	Does everything	Passes actions and binds data for VM	Passes actions and binds data for Presenter
Data flow	Multi-directional	Multi-directional	Multi-directional
Testability	hard (too many responsibilities)	Better	Best
Entry	Easy	hard (when starting with reactive extensions)	Hardest
Reusability	Rather none	Rather small	Ok
Refactoring	Normal	Normal (affects VC - VM boundary)	Worse (can affect many boundaries due to multi-directional flow)
Number of files	Normal	Additional VM for every VC	Many
Lines of code in file	Too many	Many	Most-satisfactory