Other Architectures "VIPER"

What is VIPER?

It is an application architecture that reduces coding complexity, especially in large projects. The purpose of this architecture is to separate the operational codes with the project and modules and regularization. It is basically the use of protocol because interlayer communication is provided by Protocols in VIPER design pattern.

V = View
- Responsibility: Displays UI and delegates user interactions to Presenter
- Contains: UIView/UIViewController, UI setup, IBOutlets, IBActions
- Should Not: Contain business logic or know about data models

I = Interactor
- Responsibility: Contains business logic, fetches data from services
- Contains: Use cases, API calls, database operations
- Should Not: Know about UI or presentation logic

P = Presenter
- Responsibility: Mediates between View and Interactor, formats data for display
- Contains: Presentation logic, response handling
- Should Not: Directly access UIKit or make API calls

E = Entity
- Responsibility: Plain data objects (structs/classes)
- Contains: Data models, no business logic

R = Router
- Responsibility: Handles navigation between modules
- Contains: Navigation logic, dependency injection setup
- Should Not: Contain business or presentation logic

*Think of VIPER like building a restaurant, not coding an app:
View = Dining Room and Waiter
- What Customers see (tables, menu, decor)
- Takes order from customers
- Brings food from kitchen
- Does not cook or decide recipes

Presenter = Maiter D' (Restaurant Manager)
- Coordinates between dining room an kitchen
- Takes order from waiter, tells chef what to cook
- Makes food look nice before serving
- Does not cook or set tables

Interactor = Chef and Kitchen

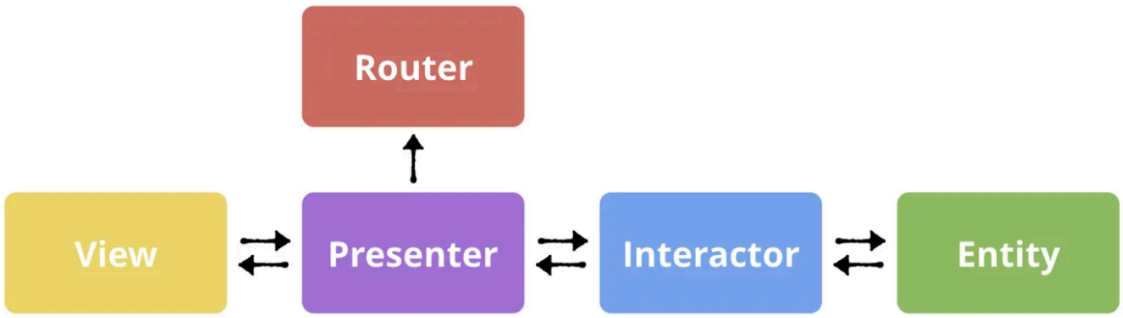
- Actually cooks the food
- Follow recipes (business logic)
- Manages ingredients (data)
- Does not serve or talk to customers

Entity = Ingredients

- Raw materials (tomatoes, chicken, spices)
- Just data, no logic
- Chef turns them into meals

Router = Host/Hostess

- Seats Customers at tables
- Guides them to restroom
- Shows them out when done
- Does not cook or take orders



Source

ADVANTAGES

1. Clean Separation of Concerns: Each component has a single, clear responsibility
2. High Testability: Business logic (Interactor) and presentation logic (Presenter) are easily testable without UI
3. Scalability: Easy to maintain and extend large codbases
4. Team Collaboration: Different team members can work on different layers simultaneously
5. Reusability: Interactors can be reused across different modules

DISADVANTAGES

1. Boilerplate Code: Requires more files and setup than MVC
2. Learning Curve: Steeper than traditional patterns
3. Over-engineering risk: Can be excessive for simple screens
4. Complex Communication: More components to wire together
5. Memory Management: Need careful handling of retain cycles between components

TYPICAL FILE STRUCTURE

```
Modules/
├── UserProfile/
│   ├── Contracts/
│   │   ├── UserProfileContracts.swift
│   ├── View/
│   │   ├── UserProfileViewController.swift
│   │   └── UserProfileView.xib
│   ├── Presenter/
│   │   └── UserProfilePresenter.swift
│   ├── Interactor/
│   │   └── UserProfileInteractor.swift
│   ├── Router/
│   │   └── UserProfileRouter.swift
│   └── Entity/
│       └── User.swift
```

BEST PRACTICES

1. Use Protocols for All Components: Enables easy testing and mocking
2. Inject Dependencies: Pass dependencies through initializer
3. Follow Unidirectional Data Flow: View->Presenter->Interactor->Entity
4. Keep Views Dumb: Views should only handle UI updates
5. Test Presenter and Interactor: These Contain the most important logic
6. Use Weak References: Avoid retain cycles between View-Presenter-Interactor
7. Create On Router per Module: Handles all navigation for that model

SAMPLES

1. Entity (Model)

swift Copy Download

```swift
// User.swift
struct User: Codable {
    let id: Int
    let name: String
    let email: String
    let profileImageURL: String

    var displayName: String {
        return "👤 \(name)"
    }
}
```

2. Contracts (Protocols)

swift Copy Download

```swift
// UserProfileContracts.swift
protocol UserProfileViewProtocol: AnyObject {
    func showUserProfile(_ user: User)
    func showError(_ message: String)
    func showLoading()
    func hideLoading()
}

protocol UserProfilePresenterProtocol {
    func viewDidLoad()
    func didTapEditProfile()
    func didTapLogout()
}

protocol UserProfileInteractorProtocol {
    func fetchUserProfile(userId: Int)
}

protocol UserProfileRouterProtocol {
    func navigateToEditProfile(for user: User)
    func navigateToLogin()
}
```

3. Interactor

swift Copy Download

```swift
// UserProfileInteractor.swift
class UserProfileInteractor: UserProfileInteractorProtocol {
    weak var presenter: UserProfileInteractorOutputProtocol?
    private let userService: UserServiceProtocol

    init(userService: UserServiceProtocol = UserService()) {
        self.userService = userService
    }

    func fetchUserProfile(userId: Int) {
        userService.fetchUser(id: userId) { [weak self] result in
            switch result {
            case .success(let user):
                self?.presenter?.userProfileFetched(user)
            case .failure(let error):
                self?.presenter?.userProfileFetchFailed(error)
            }
        }
    }
}
```

4. Presenter

swift Copy Download

```swift
// UserProfilePresenter.swift
class UserProfilePresenter: UserProfilePresenterProtocol {
    weak var view: UserProfileViewProtocol?
    var interactor: UserProfileInteractorProtocol
    var router: UserProfileRouterProtocol
    private let userId: Int

    init(view: UserProfileViewProtocol,
         interactor: UserProfileInteractorProtocol,
         router: UserProfileRouterProtocol,
         userId: Int) {
        self.view = view
        self.interactor = interactor
        self.router = router
        self.userId = userId
    }

    func viewDidLoad() {
        view?.showLoading()
        interactor.fetchUserProfile(userId: userId)
    }
```

5. View (ViewController)

swift Copy Download

```swift
// UserProfileViewController.swift
class UserProfileViewController: UIViewController, UserProfileViewProtocol {
    @IBOutlet private weak var nameLabel: UILabel!
    @IBOutlet private weak var emailLabel: UILabel!
    @IBOutlet private weak var profileImageView: UIImageView!
    @IBOutlet private weak var activityIndicator: UIActivityIndicatorView!

    var presenter: UserProfilePresenterProtocol!

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

    @IBAction private func editProfileTapped(_ sender: UIButton) {
        presenter.didTapEditProfile()
    }

    @IBAction private func logoutTapped(_ sender: UIButton) {
        presenter.didTapLogout()
    }
```

6. Router

swift Copy Download

```swift
// UserProfileRouter.swift
class UserProfileRouter: UserProfileRouterProtocol {
    weak var viewController: UIViewController?

    static func createModule(userId: Int) -> UIViewController {
        let view = UIStoryboard(name: "Main", bundle: nil)
            .instantiateViewController(withIdentifier: "UserProfileViewContro
ller") as! UserProfileViewController

        let interactor = UserProfileInteractor()
        let router = UserProfileRouter()
        let presenter = UserProfilePresenter(view: view,
                                             interactor: interactor,
                                             router: router,
                                             userId: userId)

        view.presenter = presenter
        interactor.presenter = presenter
        router.viewController = view

        return view
    }
```