Архитектуры и паттерны проектирования

MVC, MVP, MVVM, VIPER

Когда разработчики создают мобильные приложения, их код со временем становится сложнее. Если не применять структурированный подход, проект быстро превращается в **хаос**, где трудно разобраться, какая часть отвечает за что.

Чтобы избежать этого, используются **архитектурные паттерны — правила организации кода**, которые помогают:

- Разделить ответственность между разными частями приложения.
- Упростить поддержку и масштабирование проекта.
- Улучшить тестируемость кода.

Чем архитектуры отличаются от паттернов?

- **Архитектура** — это **глобальный подход** к организации приложения. Она определяет, как разделять логику, данные и пользовательский интерфейс. Например, **MVC, MVP, MVVM, VIPER**.

- **Паттерны проектирования** — это **локальные решения конкретных задач** в коде. Например, **делегирование**, **фабричный метод**, **синглтон**. Они применяются внутри архитектур.

Архитектура помогает построить структуру проекта, а паттерны помогают решать небольшие, но важные задачи внутри этой структуры.

Какие проблемы решают архитектуры?
Представьте, что в приложении логика данных, интерфейс и бизнес-правила смешаны в одном месте. Что случится?

- Контроллеры разрастаются, становятся трудно читаемыми.
- Невозможно повторно использовать код в разных частях проекта.
- Невозможно протестировать ключевую бизнес-логику без запуска всего приложения.

Поэтому **важно разделять логику приложения**.

Простое разделение кода по слоям

В большинстве архитектур код делится на три слоя:

- **Model (Модель)** — отвечает за данные и бизнес-логику.
- **View (Представление)** — отвечает за отображение информации на экране.
- **Controller / Presenter / ViewModel (Контроллер / Презентер / ВьюМодель)** — управляет связью между Model и View.

Каждая архитектура реализует эту концепцию по-разному, в зависимости от требований к тестируемости, удобству поддержки и гибкости.

Давайте теперь подробно разберем **MVC** — одну из самых базовых архитектур, предложенную Apple для разработки iOS-приложений. Эта архитектура используется уже много лет и остается актуальной, хотя и имеет свои ограничения, с которыми сталкиваются разработчики на более сложных проектах.

Что такое MVC?

MVC — это аббревиатура от **Model-View-Controller**. Эта архитектура делит приложение на три основных компонента:

1. **Model (Модель)** — слой, который управляет данными приложения, логикой и состоянием. Модель представляет бизнес-логику и общается с базой данных, API или другими источниками данных. Модель не зависит от представления и не знает ничего о пользовательском интерфейсе.
2. **View (Представление)** — отвечает за отображение данных пользователю. Это визуальная часть приложения, которая представляет собой все элементы интерфейса, такие как кнопки, метки, изображения и так далее. View не содержит бизнес-логики, а просто получает информацию от модели и отображает её пользователю.
3. **Controller (Контроллер)** — слой, который служит связующим звеном между Model и View. Контроллер управляет взаимодействием между моделью и представлением. Он обрабатывает действия пользователя, изменяет модель и обновляет представление. Контроллер знает о **View** и **Model**, но модель не знает о контроллере, и представление не должно содержать логику работы с данными.

Пример использования MVC в iOS-приложении

Давайте рассмотрим, как может выглядеть простое приложение на основе архитектуры MVC.

1. Model (Модель)

Модель в нашем примере будет простой моделью пользователя, которая хранит информацию о пользователе.

```
class UserModel {
    var name: String
    var age: Int

    init(name: String, age: Int) {
        self.name = name
        self.age = age
    }
}
```

Здесь UserModel хранит данные пользователя, такие как имя и возраст. Модель не знает, как отображать эти данные или что с ними делать в интерфейсе.

2. View (Представление)

Представление будет отображать имя и возраст пользователя на экране. В iOS это обычно может быть UILabel.

```
class UserView: UIView {
    var nameLabel: UILabel = UILabel()
    var ageLabel: UILabel = UILabel()

    func updateView(with user: UserModel) {
        nameLabel.text = user.name
        ageLabel.text = "\(user.age) years old"
    }
}
```

UserView — это представление, которое просто отображает данные пользователя. Оно не изменяет их, а только принимает их и отображает. В этом компоненте нет бизнес-логики, только визуальное представление.

3. Controller (Контроллер)

Контроллер управляет всем процессом. Он отвечает за получение данных от модели и обновление представления.

```
class UserController: UIViewController {
    var userModel: UserModel!
    var userView: UserView!

    override func viewDidLoad() {
        super.viewDidLoad()

        userModel = UserModel(name: "John Doe", age: 30)
        userView = UserView()
        userView.frame = view.bounds

        // Обновляем представление с данными из модели
        userView.updateView(with: userModel)

        view.addSubview(userView)
    }
}
```

В UserController мы создаем экземпляр модели UserModel, затем создаем и настраиваем UserView. Контроллер обновляет представление, передавая в него данные из модели. Это и есть основная задача контроллера — поддерживать связь между данными и интерфейсом.

Преимущества и ограничения MVC

Преимущества:

1. **Простота и понятность.** MVC — это базовый и очень интуитивно понятный подход. Вы можете легко понять, как работает ваш код, когда видите, что логика разделена на три компонента: данные, представление и контроллер.
2. **Легкость в тестировании.** Модель можно тестировать отдельно от представления и контроллера, так как она не зависит от UI.

Ограничения:

1. **Большие контроллеры.** В более сложных приложениях контроллеры могут становиться очень большими и трудными для понимания и поддержки. Это известная проблема, называемая **Massive View Controller**. Контроллеры, управляя большим количеством логики, становятся трудными для масштабирования.
2. **Трудности с многократными представлениями.** Если приложение включает несколько видов представлений (например, экраны с различным контентом), то код представлений может сильно дублироваться.
3. **Отсутствие чёткого разделения ответственности.** Если контроллер выполняет слишком много функций, это нарушает принцип единой ответственности.

MVP (Model-View-Presenter)

Что такое MVP?

MVP (Model-View-Presenter) — это архитектурный паттерн, который разделяет код на три основных компонента:

- **Model (Модель)** — отвечает за данные и бизнес-логику. Здесь работают сетевые запросы, базы данных и обработка информации.
- **View (Представление)** — отображает данные пользователю. В iOS это обычно **UIViewController**, который показывает интерфейс и получает пользовательский ввод.
- **Presenter (Презентер)** — посредник между View и Model. Он получает запросы от View, запрашивает данные у Model и передает обратно в View.

Ключевая идея MVP — **разгрузить View от логики и сделать код более тестируемым**.

Принцип работы MVP

Давайте представим ситуацию: мы разрабатываем экран профиля пользователя. Без архитектуры ViewController обычно берет на себя все роли сразу:

- Загружает данные из сети
- Обрабатывает их
- Настраивает UI

При изменении API или бизнес-логики придется вносить изменения в этот же класс, что усложняет поддержку. **MVP решает эту проблему разделением обязанностей:**

- **View** сообщает **Presenter**, что пользователь хочет загрузить профиль.
- **Presenter** запрашивает у **Model** данные о пользователе.
- **Model** возвращает результат (успешный или с ошибкой).
- **Presenter** передает полученную информацию обратно в **View**.

View ничего не знает о Model, а Model не взаимодействует с View напрямую. **Вся связь идет через Presenter**.

Как это выглядит в коде?

1. Интерфейс View
View должна уметь показывать данные или ошибку. Для этого мы создадим протокол, который будет реализован в ViewController.

```
protocol ProfileViewInput: AnyObject {
    func showProfile(name: String, age: Int)
    func showError(message: String)
}
```

ViewController будет работать с этим протоколом, а не напрямую с Presenter, что упрощает тестирование и замену компонентов.

2. Презентер
Презентер управляет логикой, но не знает деталей реализации View и Model.

```
class ProfilePresenter {
    private weak var view: ProfileViewInput?
    private let profileService: ProfileService

    init(view: ProfileViewInput, profileService: ProfileService) {
        self.view = view
        self.profileService = profileService
    }

    func loadProfile() {
        if let profile = profileService.getProfile() {
            view?.showProfile(name: profile.name, age: profile.age)
        } else {
            view?.showError(message: "Не удалось загрузить профиль")
        }
    }
}
```

Здесь **Presenter** не содержит UIKit, не обращается напрямую к UI и не знает, где именно показываются данные.

3. Модель (Model)
Здесь реализуется работа с данными.

```
struct Profile {
    let name: String
    let age: Int
}

class ProfileService {
    func getProfile() -> Profile? {
        return Profile(name: "Алексей", age: 25)
    }
}
```

Model не взаимодействует ни с View, ни с Presenter напрямую.

4. Реализация View (UIViewController)
ViewController теперь отвечает только за отображение информации, а Presenter управляет логикой.

```swift
class ProfileViewController: UIViewController, ProfileViewInput {
    private let presenter: ProfilePresenter

    private let nameLabel = UILabel()
    private let ageLabel = UILabel()

    init(presenter: ProfilePresenter) {
        self.presenter = presenter
        super.init(nibName: nil, bundle: nil)
    }

    required init?(coder: NSCoder) { fatalError("init(coder:) не поддерживается") }

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

    private func setupUI() {
        view.backgroundColor = .white
        view.addSubview(nameLabel)
        view.addSubview(ageLabel)

        // Устанавливаем фреймы (в реальном проекте лучше использовать Auto Layout)
        nameLabel.frame = CGRect(x: 20, y: 100, width: 200, height: 30)
        ageLabel.frame = CGRect(x: 20, y: 140, width: 200, height: 30)
    }

    func showProfile(name: String, age: Int) {
        nameLabel.text = "Имя: \(name)"
        ageLabel.text = "Возраст: \(age)"
    }

    func showError(message: String) {
        print("Ошибка: \(message)")
    }
}
```

Как это соединить?

В SceneDelegate или AppDelegate создаем зависимости и передаем их в ViewController:

let profileService = ProfileService()
let profileVC = ProfileViewController(presenter: ProfilePresenter(view: profileVC, profileService: profileService))

window?.rootViewController = profileVC
window?.makeKeyAndVisible()

Теперь все компоненты работают **взаимосвязанно, но независимо друг от друга**.

Плюсы MVP
- **Упрощенная поддержка** — View и логика отделены, что делает код более понятным.
- **Тестируемость** — Presenter можно тестировать отдельно от View, используя мок-данные.
- **Гибкость** — UI можно изменять без затрагивания логики.

Минусы MVP
- **Дополнительный код** — появляются дополнительные слои, что увеличивает объем кода.
- **Сложность для маленьких проектов** — в небольших приложениях MVP может выглядеть избыточно.

MVP — это мощная архитектура, позволяющая создавать гибкие и масштабируемые приложения. Она особенно полезна, если вы хотите отделить бизнес-логику от UI и сделать код тестируемым.

Однако, как и любая архитектура, MVP требует дисциплины и разумного применения. В небольших проектах можно обойтись более простыми решениями.

MVVM: Разбираем еще одну архитектуру для iOS

Представьте, что вам поручили разработать сложное приложение, в котором интерфейс динамически обновляется при изменении данных. Например, банковское приложение, где баланс должен обновляться без дополнительных действий пользователя.

Конечно, можно использовать MVP, но представьте, что ваш Presenter становится слишком громоздким, ведь он не только получает данные, но и форматирует их перед отправкой во View. Тут на помощь приходит **MVVM (Model-View-ViewModel)** — архитектурный паттерн, который решает эту проблему и делает код более чистым и удобным для тестирования.

В чем суть MVVM?

Главная идея MVVM — убрать логику из контроллера и вынести ее в специальный слой **ViewModel**, который будет обрабатывать данные и предоставлять их в удобном для UI формате.

В этой архитектуре есть три ключевых компонента:
1. **Model** — отвечает за данные и их обработку. Это может быть структура или класс, который получает и хранит информацию.
2. **View** — интерфейс пользователя. В iOS это UIViewController и его UIView-элементы.
3. **ViewModel** — связующее звено, которое форматирует данные и передает их в UI. Главное отличие от MVP — ViewModel не знает о View напрямую.

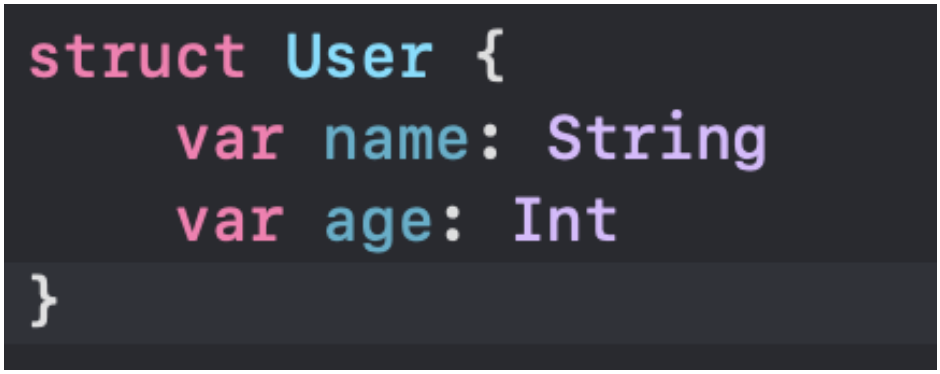
Давайте разберем на практике

Допустим, у нас есть экран профиля, где отображается имя пользователя и его возраст. Пользователь может нажать кнопку, чтобы увеличить возраст.

Как это реализовать в MVVM?

1. Создадим модель (Model).
2. Напишем ViewModel, которая подготовит данные для UI.
3. Настроим ViewController, который будет подписан на изменения в ViewModel.

1. Создаем Model



Просто структура, в которой хранятся данные пользователя.

2. Создаем ViewModel

ViewModel будет отвечать за логику работы с пользователем и обновление данных.

```
import Combine

class UserViewModel {
    @Published private(set) var name: String = ""
    @Published private(set) var age: String = ""

    private var user = User(name: "Алексей", age: 25)

    func loadUser() {
        name = user.name
        age = "Возраст: \(user.age)"
    }

    func increaseAge() {
        user.age += 1
        age = "Возраст: \(user.age)"
    }
}
```

• Используем @Published, чтобы View автоматически обновлялась при изменении данных.
• loadUser() загружает начальные данные.
• increaseAge() увеличивает возраст и обновляет UI.

3. Настраиваем ViewController

Теперь создадим экран и подпишем его на изменения в ViewModel.

```swift
class UserViewController: UIViewController {
    private let viewModel = UserViewModel()
    private var cancellables: Set<AnyCancellable> = []

    private let nameLabel = UILabel()
    private let ageLabel = UILabel()
    private let button = UIButton(type: .system)

    override func viewDidLoad() {
        super.viewDidLoad()
        setupUI()
        setupBindings()
        viewModel.loadUser()
    }

    private func setupUI() {
        view.backgroundColor = .white
        view.addSubview(nameLabel)
        view.addSubview(ageLabel)
        view.addSubview(button)

        nameLabel.frame = CGRect(x: 20, y: 100, width: 300, height: 30)
        ageLabel.frame = CGRect(x: 20, y: 140, width: 300, height: 30)
        button.frame = CGRect(x: 20, y: 180, width: 200, height: 50)

        button.setTitle("Добавить возраст", for: .normal)
        button.addTarget(self, action: #selector(didTapButton), for: .touchUpInside)
    }

    private func setupBindings() {
        viewModel.$name
            .receive(on: RunLoop.main)
            .sink { [weak self] name in self?.nameLabel.text = name }
            .store(in: &cancellables)

        viewModel.$age
            .receive(on: RunLoop.main)
            .sink { [weak self] age in self?.ageLabel.text = age }
            .store(in: &cancellables)
    }

    @objc private func didTapButton() {
        viewModel.increaseAge()
    }
}
```

Здесь важно, что ViewModel самостоятельно обновляет UI, а ViewController просто подписывается на ее изменения.

В чем преимущества MVVM?

1. **Чистый код** — ViewController остается простым, так как логика вынесена в ViewModel.
2. **Удобное тестирование** — ViewModel можно тестировать отдельно, без зависимости от UI.
3. **Автоматическое обновление UI** — если использовать Combine или RxSwift, ViewModel автоматически передает изменения в UI.

В приведённом выше примере архитектуры **MVVM** мы использовали свойство с аннотацией @Published. Давайте разберемся, что это и как работает в контексте **Combine**.

Combine и реактивное программирование

Combine — это фреймворк от Apple, предназначенный для работы с асинхронными событиями и реактивным программированием. Он позволяет обрабатывать потоки данных, связывая **издателей (Publishers)** и **подписчиков (Subscribers)**. Таким образом, Combine упрощает реактивную обработку изменений данных и обновление UI.

@Published

С помощью аннотации @Published, мы говорим, что переменная будет **издателем** (Publisher). Это значит, что как только её значение изменится, все **подписчики** (например, в представлении View) будут автоматически уведомлены и смогут отреагировать на эти изменения.

В примере MVVM, мы использовали @Published для свойства name в **ViewModel**:

```
class ViewModel: ObservableObject {
   @Published var name: String = ""
}
```

Здесь @Published гарантирует, что когда значение name изменится, все элементы UI, подписанные на это свойство, получат актуальное значение.

Вместо того, чтобы вручную обновлять интерфейс при изменении данных, с помощью **Combine** и @Published мы устанавливаем двустороннюю связь между **ViewModel** и **View**. **Combine** упрощает реактивное программирование, устраняя необходимость в большом количестве шаблонного кода и обеспечивая удобную работу с асинхронными данными.

VIPER

VIPER возник как адаптация принципов **Clean Architecture** Роберта Мартина (Uncle Bob) для iOS-разработки. Целью было создание структуры, которая обеспечивала бы чёткое разделение обязанностей, улучшенную тестируемость и масштабируемость приложений. Однако, несмотря на эти усилия, VIPER не стал массово применяться в iOS-разработке, поскольку его избыточность не оправдывает себя в большинстве проектов.

Актуальность VIPER в современном iOS-разработке

С развитием iOS-разработки и появлением новых архитектурных паттернов, таких как **MVVM** и **MVP**, VIPER стал менее популярным. Эти более простые и гибкие архитектуры часто оказываются более подходящими для большинства проектов, так как они обеспечивают хорошее разделение обязанностей без излишней сложности. В результате, VIPER используется реже, особенно в небольших и средних проектах, где его сложность может быть избыточной.

VIPER — это аббревиатура, которая обозначает несколько слоев, каждый из которых выполняет свою четко определенную задачу. Это архитектура, которая пытается максимизировать разделение ответственности и минимизировать зависимости между различными частями приложения. Каждый из слоев VIPER имеет свою роль:

- **V (View)** — Представление. Это UI-компоненты, которые отображают данные. В VIPER представление отвечает только за отображение, оно не содержит бизнес-логики.
- **I (Interactor)** — Интерактор. Этот слой управляет бизнес-логикой, обрабатывает данные и выполняет задачи, такие как запросы к базе данных или сети. Он работает с моделями данных, но не взаимодействует напрямую с UI.
- **P (Presenter)** — Презентер. Презентер управляет логикой, необходимой для обработки событий от пользователя, и решает, как эти события будут отображаться в представлении. Он действует как посредник между View и Interactor, а также обрабатывает логику для UI.

• **E (Entity)** — Сущность. Это модели данных, которые определяют структуру данных. Обычно эти сущности используются в слое Interactor для выполнения бизнес-логики.

• **R (Router)** — Роутер. Этот компонент отвечает за навигацию между экранами. Он решает, какой экран будет отображаться следующим, и как управлять переходами между экранами.

Рассмотрим, как может выглядеть приложение, использующее VIPER. Пусть это будет приложение, которое отображает список пользователей.

View (Представление)

```
protocol UserListView: AnyObject {
    func displayUsers(_ users: [User])
}
```

Представление (View) будет отображать список пользователей. В нашем примере представление будет получать данные от презентера и обновлять UI, но не будет заниматься бизнес-логикой.

Interactor

```
class UserService {
    // Запрос на сервер
    func fetchUsers(completion: @escaping (_ users: [User]) -> Void) { }
}

class UserListInteractor: UserListInteractorInput {
    let userService: UserService

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

    func fetchUsers() {
        userService.fetchUsers { users in
            // Возвращаем пользователей в презентер
        }
    }
}
```

Интерактор работает с бизнес-логикой — например, извлекает данные из сети или базы данных. В данном случае, интерактор вызывает сервис, который загружает список пользователей.

Presenter

```
protocol UserListPresenter {
    func viewDidLoad()
}

class UserListPresenterImpl: UserListPresenter {
    let view: UserListView
    let interactor: UserListInteractorInput

    init(view: UserListView, interactor: UserListInteractorInput) {
        self.view = view
        self.interactor = interactor
    }

    func viewDidLoad() {
        interactor.fetchUsers()
    }

    func usersFetched(_ users: [User]) {
        view.displayUsers(users)
    }
}
```

Презентер управляет логикой между View и Interactor. Он получает запросы от представления и передает их в интерактор, а затем обновляет представление с полученными данными.

Entity

```
struct User {
   let id: Int
   let name: String
}
```

Сущности представляют собой данные, которые используются в бизнес-логике приложения. В нашем случае сущность User представляет пользователя, с его идентификатором и именем.

Router

```
protocol UserListRouter {
    func navigateToUserDetails(user: User)
}

class UserListRouterImpl: UserListRouter {
    func navigateToUserDetails(user: User) {
        // Навигация на экран подробностей пользователя
    }
}
```

Роутер управляет навигацией. В данном примере роутер выполняет переход на экран с подробной информацией о пользователе.

Преимущества и ограничения VIPER

Преимущества:

1. **Четкое разделение ответственности.** Каждый компонент (View, Interactor, Presenter, Entity, Router) имеет свою собственную задачу, что позволяет легче тестировать и масштабировать приложение.
2. **Тестируемость.** Из-за четкого разделения логики и интерфейса, компоненты легко тестируемы.
3. **Поддержка сложных приложений.** VIPER подходит для сложных приложений с множеством экранов и сложной бизнес-логикой.

Ограничения:

1. **Избыточность.** Для многих приложений VIPER является избыточным и сложным в реализации. Простые приложения могут не требовать такой сложной архитектуры.
2. **Усложнение кода.** Из-за большого количества слоев код может стать трудным для понимания и поддержки, особенно в небольших проектах.
3. **Перегрузка проекта.** Внедрение VIPER требует множества протоколов, интерфейсов и классов, что увеличивает количество файлов и общую сложность проекта.

Архитектура VIPER предложила интересный подход к разделению обязанностей в iOS-приложениях, заимствованный из принципов Clean Architecture. Однако её сложность и избыточность привели к снижению популярности в пользу более простых и гибких архитектурных паттернов, таких как MVVM и MVP. Тем не менее, понимание VIPER может быть полезным для разработчиков, стремящихся к глубокому пониманию архитектурных принципов и готовых работать с более сложными структурами.

Clean Architecture

Теперь давайте рассмотрим **Clean Architecture**, её особенности, принципы и как она может быть использована в iOS-разработке.

Что такое Clean Architecture?

Clean Architecture — это архитектурный паттерн, предложенный Робертом Мартином (Uncle Bob), который фокусируется на разделении приложения на слои, каждый из которых отвечает за свою часть работы, и на обеспечении независимости от внешних зависимостей. Основной принцип заключается в том, чтобы логика приложения не зависела от технологий и фреймворков, которые могут изменяться со временем.

Основные принципы Clean Architecture

1. **Независимость от фреймворков**: Ваши бизнес-правила (модели, логика) не зависят от библиотек или фреймворков, таких как UIKit или SwiftUI. Например, вы можете легко изменить UI-библиотеку без изменения бизнес-логики.
2. **Независимость от UI**: Представление (UI) и взаимодействие с пользователем не должно быть связано с бизнес-логикой. UI может быть заменён или обновлён без изменения работы приложения.
3. **Независимость от базы данных**: Логика приложения не должна зависеть от того, как и где хранятся данные. Вы можете легко переключиться на другую базу данных, не затрагивая код приложения.
4. **Независимость от внешних агенств (например, сетевых сервисов)**: Приложение должно быть спроектировано так, чтобы в будущем не зависеть от сторонних сервисов.

Структура Clean Architecture

Clean Architecture делит приложение на несколько слоёв, что позволяет чётко разграничить ответственность каждого компонента:

1. **Entities (Сущности)**: Это бизнес-объекты или модели, которые содержат основные данные и логику. Например, в банковском приложении сущностью может быть Account с такими свойствами, как balance и методами для начисления или списания средств.
2. **Use Cases (Применение)**: Это слой бизнес-логики, который обрабатывает запросы от пользователя или других частей системы. Каждый use case выполняет определённую задачу, например, TransferMoney или GetUserProfile.
3. **Interface Adapters (Адаптеры интерфейса)**: Этот слой преобразует данные из формата, который понимает приложение, в формат, подходящий для UI или внешних систем (например, преобразование данных с сервера в формат, который требуется для UI).
4. **Frameworks and Drivers (Фреймворки и драйверы)**: Это внешний слой, в который входят такие компоненты, как UI (например, UIKit или SwiftUI), база данных,

сетевые запросы и прочие фреймворки. Этот слой зависит от всех слоёв выше, но сам не влияет на внутреннюю логику приложения.

Пример Clean Architecture для iOS

Представьте, что вы разрабатываете приложение для покупки билетов на концерт. В рамках **Clean Architecture** это будет выглядеть следующим образом:

- **Entities (Сущности)**: Это будут модели данных, такие как Ticket, User и Event, которые содержат информацию о концертах и пользователях.
- **Use Cases (Применение)**: Это будет логика для бронирования билетов, отображения доступных концертов, сохранения предпочтений пользователя и т.д. Например, use case BookTicket будет управлять процессом покупки билета.
- **Interface Adapters (Адаптеры интерфейса)**: Это слой, который отвечает за преобразование данных из use cases в формат, который может быть использован в UI, а также за обработку данных, полученных от UI и отправку их обратно в бизнес-логику. Например, здесь будет находиться преобразование моделей данных в объекты, которые отображаются на экране, как, например, список доступных билетов.
- **Frameworks and Drivers (Фреймворки и драйверы)**: Этот слой будет включать в себя UI (например, UIViewController и элементы управления), а также сетевые запросы, базу данных или другие внешние зависимости, которые необходимы для работы приложения.

Преимущества Clean Architecture

1. **Тестируемость**: Каждый слой можно тестировать независимо, так как они разделены на чёткие и независимые компоненты.
2. **Гибкость и масштабируемость**: Добавление новых фич и модулей в приложение не повлияет на другие части системы. Например, можно легко добавить новый способ оплаты или поменять UI, не затронув бизнес-логику.
3. **Независимость от технологий**: Код, написанный по принципам Clean Architecture, легче обновляется и адаптируется под новые технологии. Например, вы можете заменить UIKit на SwiftUI, не затрагивая основную логику работы приложения.

Clean Architecture предоставляет мощную и гибкую структуру для разработки масштабируемых и легко поддерживаемых приложений. Несмотря на свою сложность и потребность в строгом соблюдении принципов разделения ответственности, эта архитектура подходит для крупных и долгосрочных проектов, где важно поддерживать чистоту кода и его тестируемость. Однако для небольших проектов с ограниченными требованиями к масштабируемости, более простые архитектуры, такие как MVVM, могут быть более подходящими.

Clean Swift

Clean Swift — это специфическая адаптация принципов **Clean Architecture** для iOS-разработки, созданная с учётом особенностей мобильных приложений. Эта архитектура направлена на создание кода, который легко поддерживать, тестировать и масштабировать, разделяя ответственность между компонентами и гарантируя чёткое разграничение бизнес-логики и UI.

Цель **Clean Swift** — минимизировать зависимость между различными частями приложения, обеспечивая прозрачность и изоляцию логики. Это достигается благодаря разделению приложения на несколько слоёв, каждый из которых решает свою задачу, не мешая работе других. Основное отличие от классической **Clean Architecture** — это

использование структуры, более подходящей для iOS-приложений, включая работу с ViewControllers и представлениями данных.

Основные компоненты Clean Swift

1. **View**: Этот слой состоит из ViewController, который отвечает за отображение данных и обработку пользовательских действий. Он взаимодействует с Presenter для получения необходимых данных и обновления UI.
2. **Interactor**: Здесь размещается бизнес-логика приложения. В отличие от других архитектур, в Clean Swift именно Interactor выполняет все вычисления и обрабатывает запросы, поступающие от UI.
3. **Presenter**: Это промежуточный слой, который получает данные от Interactor и форматирует их в нужный для отображения вид. Он подготавливает информацию для UI, передавая её в ViewController.
4. **Router**: Слой для управления навигацией в приложении. Он отвечает за переходы между экранами, а также передачу данных между ними.

Преимущества Clean Swift

• **Чёткое разделение слоёв** помогает уменьшить зависимость между ними, что в свою очередь делает код легче поддерживаемым и расширяемым.

• **Тестируемость**: Благодаря разделению логики на компоненты, такие как Interactor и Presenter, можно легко создавать юнит-тесты, проверяя каждый слой в отдельности.

• **Масштабируемость**: Каждый слой можно развивать и изменять независимо от других. Это особенно важно в крупных проектах.

• **Управляемость**: Слой Router помогает чётко контролировать навигацию и логику переходов между экранами.

VIPER и **Clean Swift** имеют схожие цели и принципы, но между ними есть важные различия. Оба подхода — это архитектурные паттерны, нацеленные на создание чистого, поддерживаемого и легко тестируемого кода, но их организация и подходы могут немного различаться.

VIPER vs Clean Swift: Основные отличия

1. **Структура компонентов**:

• **VIPER** состоит из пяти основных компонентов:

• **View** — отображение данных и взаимодействие с пользователем.

• **Interactor** — бизнес-логика.

• **Presenter** — подготовка данных для отображения в UI.

• **Entity** — модели данных, которые используется в приложении.

• **Router** — навигация и управление переходами между экранами.

• В **Clean Swift** структура схожа, но немного упрощена и адаптирована для iOS. Основные компоненты — это:

• **View** (ViewController) — работа с UI.

• **Interactor** — бизнес-логика.

• **Presenter** — преобразование данных для отображения.

• **Router** — навигация.

В **Clean Swift** нет компонента, аналогичного **Entity** в VIPER, поскольку данные обычно передаются через слои с помощью структур или моделей.

2. **Навигация**:

• В **VIPER** навигация обычно осуществляется через **Router**, который управляет переходами между экранами, но также может использоваться для передачи данных.

• В **Clean Swift** навигация также делается через **Router**, но с более чётким разделением логики, где роутер часто взаимодействует с ViewController для инициирования переходов.

3. **Фокус**:

• **VIPER** ориентирован на создание гибкой и масштабируемой архитектуры, которая подходит для крупных приложений с множеством экранов и сложными взаимодействиями между компонентами.

• **Clean Swift** часто используется в контексте более простых приложений, где важна не только чистота кода, но и гибкость для работы с UI-компонентами, такими как ViewController.

Схожести между VIPER и Clean Swift:

• Оба паттерна разделяют приложение на несколько слоёв с чётким разграничением обязанностей. Это помогает улучшить тестируемость и поддерживаемость.

• Оба паттерна предлагают изоляцию бизнес-логики в **Interactor**, что позволяет легко тестировать логику независимо от UI.

• В обоих паттернах используется слой **Router** для навигации между экранами, что позволяет держать логику навигации отдельно от UI.

VIPER можно рассматривать как более подробную и комплексную версию **Clean Swift**, ориентированную на более крупные и сложные приложения. Clean Swift, в свою очередь, является адаптацией принципов **Clean Architecture** для мобильных приложений iOS, и в некоторых случаях может быть проще и гибче, особенно для небольших проектов.

Router и Coordinator

Router и **Coordinator** — это два важных паттерна, которые часто используются в архитектуре iOS-приложений для управления навигацией между экранами. Оба паттерна предназначены для улучшения структуры приложения и отделения логики навигации от других слоёв приложения. Однако они используются в разных контекстах и имеют свои особенности.

Router

Router — это компонент, который ответственен за управление навигацией между экранами в приложении. Обычно Router используется в архитектурах, таких как **VIPER**, и помогает изолировать логику навигации от остальных компонентов.

Основные обязанности Router:

1. **Навигация**: Router инициирует переходы между экранами. В VIPER, например, **Router** будет знать, как перемещаться между различными **ViewController** или открывать новые экраны.

2. **Передача данных**: Помимо управления переходами, Router может также передавать данные между экранами. Например, если один экран требует информации, которая должна быть получена с другого экрана, Router может управлять этой логикой.

Пример использования Router:

Router не будет заниматься бизнес-логикой или взаимодействием с UI. Взамен этого он выполняет навигацию. Пример кода на Swift:

```swift
class LoginRouter: LoginRouterProtocol {
    func navigateToHomeScreen(from view: LoginViewProtocol) {
        let homeViewController = HomeViewController()
        if let viewController = view as? UIViewController {
            viewController.navigationController?.pushViewController(homeViewController, animated: true)
        }
    }
```

В этом примере **Router** управляет навигацией с экрана логина на экран дома.

Coordinator

Coordinator — это более гибкий паттерн, который имеет более широкие возможности по организации навигации в приложении. Coordinator не только управляет переходами между экранами, но и определяет логику маршрутизации в рамках всей структуры приложения.

Coordinator чаще всего используется в архитектурах **MVVM** и **Clean Swift**. Он действует как управитель навигации на уровне приложения, а не только для отдельных экранов, как это происходит с **Router**.

Основные обязанности Coordinator:

1. **Управление навигацией**: Coordinator управляет всей навигацией внутри приложения, начиная от основного экрана и заканчивая всеми вложенными экранами.
2. **Создание экрана**: Coordinator может создать нужные экраны (или ViewControllers) и инициировать их.
3. **Отслеживание состояния навигации**: Coordinator может следить за состоянием навигации, обеспечивая переходы в зависимости от контекста (например, управление последовательностью экранов).
4. **Модулярность**: Используя Coordinator, можно разделить логику навигации на несколько независимых компонентов, что облегчает масштабирование приложения и поддержку.

Пример использования Coordinator:

Coordinator может быть более гибким и мощным, чем Router. Пример кода:

```swift
protocol Coordinator {
    func start()
}

class AppCoordinator: Coordinator {
    let window: UIWindow
    var navigationController: UINavigationController?

    init(window: UIWindow) {
        self.window = window
    }

    func start() {
        let viewController = LoginViewController()
        navigationController = UINavigationController(rootViewController: viewController)
        window.rootViewController = navigationController
        window.makeKeyAndVisible()
    }
```

В данном примере **AppCoordinator** управляет запуском приложения, создаёт навигационный контроллер и устанавливает его как корневой. Он может также инициировать переходы между экранами, создавая их по мере необходимости.

Отличия Router и Coordinator

1. **Обязанности**:

• **Router** ответственен за навигацию между экранами. Он работает в рамках одного экрана или модуля.

• **Coordinator** управляет всей навигацией внутри приложения или большого модуля. Он может работать с несколькими экранами и даже с несколькими Router'ами, если приложение сложное.

2. **Масштабируемость**:

• **Router** подходит для более мелких задач, когда необходимо изолировать логику навигации для одного модуля или экрана.

• **Coordinator** идеально подходит для управления всей навигацией приложения и использования сложных сценариев переходов и состояния.

3. **Гибкость**:

• **Router** обычно привязан к конкретной View (например, экрану).

• **Coordinator** обладает большей гибкостью и может быть использован для организации навигации на уровне всего приложения.

Когда использовать Router или Coordinator:

• Используйте **Router**, если ваше приложение имеет небольшую логику навигации, и вы хотите, чтобы навигация была организована в одном модуле или экране.

• Используйте **Coordinator**, если приложение имеет более сложную структуру навигации, где один экран может быть связан с несколькими другими экранами, и вы хотите централизовать логику навигации в одном месте.

Обе модели — **Router** и **Coordinator** — помогают изолировать навигацию от других компонентов приложения, что способствует более чистой и поддерживаемой архитектуре. Важно понимать, что **Router** — это более лёгкий и локализованный паттерн для навигации внутри одного экрана, а **Coordinator** — это более масштабируемая и централизованная система для управления навигацией во всём приложении.

SOLID

Принципы SOLID — это набор из пяти основных принципов объектно-ориентированного проектирования, которые помогают создавать гибкие, масштабируемые и легко поддерживаемые приложения. Эти принципы применимы не только для iOS-разработки, но и для программирования в целом, обеспечивая структурированный подход к разработке.

1. Принцип единой ответственности (Single Responsibility Principle, SRP)

Принцип единой ответственности утверждает, что класс должен иметь только одну причину для изменения. Это означает, что класс должен отвечать только за одну задачу. Если класс выполняет несколько различных функций, то изменение одной из этих функций может повлиять на другие части кода, что приведет к более сложному обслуживанию и тестированию.

Пример:

Предположим, у нас есть класс, который отвечает как за отображение информации, так и за сохранение данных. Согласно SRP, эти две ответственности нужно разделить на два класса.

```
// Неправильно: класс выполняет две роли
class UserInfo {
    func saveData() {
        // Сохраняет данные
    }

    func displayInfo() {
        // Отображает информацию
    }
}

// Правильно: разделяем обязанности
class UserInfoDisplay {
    func displayInfo() {
        // Отображает информацию
    }
}

class UserInfoSave {
    func saveData() {
        // Сохраняет данные
    }
}
```

2. Принцип открытости/закрытости (Open/Closed Principle, OCP)

Принцип открытости/закрытости гласит, что класс должен быть открыт для расширения, но закрыт для изменения. Это означает, что при необходимости добавления нового функционала в систему не следует изменять существующие классы, а лучше использовать наследование или композицию для расширения их возможностей.

Пример:

Если мы добавляем новый способ сохранения данных в приложение, лучше создать новый класс, чем изменять существующий.

```
// Открыто для расширения, закрыто для изменения
protocol DataSaver {
    func saveData()
}

class LocalDataSaver: DataSaver {
    func saveData() {
        // Сохраняет данные локально
    }
}

class CloudDataSaver: DataSaver {
    func saveData() {
        // Сохраняет данные в облаке
    }
}
```

3. Принцип подстановки Лисков (Liskov Substitution Principle, LSP)

Принцип подстановки Лисков гласит, что объекты подклассов должны быть взаимозаменяемыми с объектами их базовых классов, не нарушая правильности работы программы. Если класс A является подтипом класса B, то объект типа A должен вести себя так, как и объект типа B.

Пример:

Если мы создаем подкласс, который нарушает поведение родительского класса, это может привести к неправильной работе системы.

```
// Неправильно: подкласс нарушает поведение родителя
class Bird {
    func fly() {
        print("Я летаю")
    }
}

class Ostrich: Bird {
    override func fly() {
        // Странное поведение — страус не может летать
        print("Я не могу летать")
    }
}

// Лучше сделать так
class Bird {
    func move() {
        print("Я двигаюсь")
    }
}

class Sparrow: Bird {
    func fly() {
        print("Я летаю")
    }
}

class Ostrich: Bird {
    func run() {
        print("Я бегаю")
    }
}
```

4. Принцип разделения интерфейса (Interface Segregation Principle, ISP)

Принцип разделения интерфейса утверждает, что интерфейсы должны быть специфичными и разделёнными, а не слишком общими. Это предотвращает ситуации, когда клиент должен зависеть от методов, которые ему не нужны.

Пример:

Если интерфейс слишком громоздкий и включает методы, которые не нужны всем классам, то лучше разделить его на несколько меньших интерфейсов.

```
// Неправильно: класс зависит от ненужных методов
protocol Worker {
    func work()
    func eat()
}

class Employee: Worker {
    func work() {
        // Работает
    }

    func eat() {
        // Ест
    }
}

class Robot: Worker {
    func work() {
        // Работает
    }

    func eat() {
        // Не ест, но обязан реализовывать этот метод
    }
}
// Правильно: разделяем интерфейсы
protocol Workable {
    func work()
}

protocol Eatable {
    func eat()
}

class Employee: Workable, Eatable {
    func work() {
        // Работает
    }

    func eat() {
        // Ест
    }
}

class Robot: Workable {
    func work() {
        // Работает
    }
}
```

5. Принцип инверсии зависимостей (Dependency Inversion Principle, DIP)

Принцип инверсии зависимостей гласит, что высокоуровневые модули не должны зависеть от низкоуровневых, а оба должны зависеть от абстракций. Это помогает уменьшить связность между компонентами и повысить гибкость системы.

Пример:

Вместо того, чтобы класс зависел напрямую от реализации, он должен зависеть от интерфейса, а не от конкретных реализаций.

```
// Неправильно: класс зависит от конкретной реализации
class UserManager {
    var database: Database

    init() {
        self.database = MySQLDatabase() // Привязка к конкретной реализации
    }

    func save() {
        database.saveData()
    }
}

// Правильно: зависимость от абстракции
protocol Database {
    func saveData()
}
 // Правильно: зависимость от абстракции
 protocol Database {
     func saveData()
 }

 class MySQLDatabase: Database {
     func saveData() {
         // Сохраняет данные в MySQL
     }
 }

 class UserManager {
     var database: Database

     init(database: Database) {
         self.database = database
     }

     func save() {
         database.saveData()
     }
 }
```

Пример архитектуры MVC с принципами SOLID

Теперь давайте рассмотрим, как принцип **MVC** можно интегрировать с принципами **SOLID** на примере простого приложения.

1. Model: Класс модели должен иметь только одну задачу — хранение и обработку данных.

```
// Принцип SRP: класс модели имеет одну ответственность
class User {
    var name: String
    var age: Int

    init(name: String, age: Int) {
        self.name = name
        self.age = age
    }

    func updateUserName(_ name: String) {
        self.name = name
    }
}
```

2. View: Представление должно быть независимым от модели, только отображать данные.

```
// Принцип SRP: класс View занимается только отображением данных
class UserView {
    func displayUserInfo(name: String, age: Int) {
        print("User's name: \(name), age: \(age)")
    }
}
```

3. Controller: Контроллер отвечает за бизнес-логику, но он также не должен быть перегружен. Принцип OCP: контроллер открыт для расширений, закрыт для изменений.

```
// Принцип DIP: зависимость от абстракции
class UserController {
    var user: User
    var view: UserView

    init(user: User, view: UserView) {
        self.user = user
        self.view = view
    }

    func updateUser(name: String) {
        user.updateUserName(name)
        view.displayUserInfo(name: user.name, age: user.age)
    }
}
```

Здесь, мы видим, что каждый компонент отвечает за свою задачу, следуя принципам SOLID. В итоге, наша система гибкая и легко расширяемая, что позволяет легко добавлять новые функциональные возможности и поддерживать код.

Паттерны проектирования

Паттерны проектирования — это проверенные решения типичных задач в разработке программного обеспечения, которые помогают решить различные проблемы, повысить гибкость кода, улучшить поддержку и тестируемость приложений. Паттерны можно классифицировать по типам в зависимости от их назначения. Рассмотрим несколько популярных паттернов проектирования: один из каждого типа.

1. Порождающие паттерны (Creational Patterns)

Singleton (Одиночка)

Этот паттерн обеспечит создание только одного экземпляра класса в течение всего времени работы приложения. Singleton полезен в ситуациях, когда необходимо, чтобы в системе существовал только один объект для управления ресурсами или состоянием, например, для работы с базой данных или глобальным кэшем.

Как это работает:

Singleton создается с помощью статической переменной, которая будет хранить один экземпляр объекта. К этому экземпляру можно получить доступ через глобальный метод или свойство, что исключает возможность создания нескольких объектов.

Пример:

```
class Singleton {
    static let shared = Singleton()

    private init() { }

    func doSomething() {
        print("Singleton does something!")
    }
}

// Использование:
Singleton.shared.doSomething()
```

Что это решает:
Singleton исключает необходимость создания новых объектов в разных частях программы. Вместо этого мы используем единственный экземпляр класса, который доступен глобально.

2. Структурные паттерны (Structural Patterns)

Adapter (Адаптер)

Адаптер позволяет изменить интерфейс существующего класса, чтобы он подходил к другому интерфейсу. Такой подход часто применяется, когда необходимо интегрировать старый или сторонний код в вашу систему без изменения исходного кода.

В чем его суть:

Адаптер действует как посредник между двумя несовместимыми интерфейсами. Например, старый класс может иметь методы с устаревшим интерфейсом, а адаптер может вызвать их с новым интерфейсом, совместимым с текущей системой.

Пример:

```swift
// Старый интерфейс
protocol OldAPI {
    func oldMethod()
}

class OldClass: OldAPI {
    func oldMethod() {
        print("This is the old method")
    }
}

// Новый интерфейс
protocol NewAPI {
    func newMethod()
}

class Adapter: NewAPI {
    var oldAPI: OldAPI

    init(oldAPI: OldAPI) {
        self.oldAPI = oldAPI
    }

    func newMethod() {
        oldAPI.oldMethod()  // Используем старый метод
    }
}

// Использование
let oldClass = OldClass()
let adapter = Adapter(oldAPI: oldClass)
adapter.newMethod()  // Вызывает старый метод через адаптер
```

Как это помогает:
Использование адаптера позволяет интегрировать устаревшие компоненты в текущую архитектуру, избегая изменений в их коде. Это полезно при необходимости использования сторонних библиотек, которые имеют несовместимый интерфейс.

3. Поведенческие паттерны (Behavioral Patterns)

Observer (Наблюдатель)

Наблюдатель используется для создания отношения «один ко многим», где несколько объектов могут наблюдать за изменениями состояния другого объекта. Это эффективно для уведомления нескольких компонентов о состоянии изменений, например, в случае обновлений данных или событий.

Принцип работы:

Когда объект, который называется «Subject», изменяет состояние, он уведомляет всех подписанных на его изменения наблюдателей. Это решение часто используется в системах с множеством зависимых компонентов, например, в интерфейсах с динамическими обновлениями.

```swift
protocol Observer {
    func update(data: String)
}
class ConcreteObserver: Observer {
    var name: String

    init(name: String) {
        self.name = name
    }

    func update(data: String) {
        print("\(name) received update with data: \(data)")
    }
}
class Subject {
    var observers: [Observer] = []

    func addObserver(observer: Observer) {
        observers.append(observer)
    }

    func notifyObservers(data: String) {
        for observer in observers {
            observer.update(data: data)
        }
    }
}
// Использование
let subject = Subject()
let observer1 = ConcreteObserver(name: "Observer 1")
let observer2 = ConcreteObserver(name: "Observer 2")
subject.addObserver(observer: observer1)
subject.addObserver(observer: observer2)
subject.notifyObservers(data: "New data available")
```

Какие проблемы решает:

Наблюдатель полезен для реализации паттернов событий или реактивных систем. Он обеспечивает возможность оповещения множества объектов без необходимости вручную обновлять каждый из них.

4. Паттерны для работы с поведением пользователя (Behavioral UI Patterns)

Command (Команда)

Командный паттерн инкапсулирует запросы в виде объектов, что позволяет легко отслеживать, отменять или повторно исполнять действия. Это особенно полезно для реализации механизмов отмены, повторных операций и сценариев с отложенными действиями.

Механизм работы:

Каждый запрос или операция становится объектом, который реализует интерфейс Command. Это позволяет передавать команды как параметры, отменять действия или выполнять их в нужное время.

```swift
protocol Command {
    func execute()
}

class LightOnCommand: Command {
    var light: Light

    init(light: Light) {
        self.light = light
    }

    func execute() {
        light.turnOn()
    }
}

class Light {
    func turnOn() {
        print("Light is ON")
    }

    func turnOff() {
        print("Light is OFF")
    }
}

class RemoteControl {
    var command: Command

    init(command: Command) {
        self.command = command
    }

    func pressButton() {
        command.execute()
    }
}

// Использование
let light = Light()
let lightOnCommand = LightOnCommand(light: light)
let remoteControl = RemoteControl(command: lightOnCommand)

remoteControl.pressButton()  // "Light is ON"
```

Паттерны проектирования позволяют решить общие задачи и упростить архитектуру приложений, улучшая поддержку и расширяемость. Знание и умение правильно выбирать паттерны помогает создавать качественные, гибкие и легко поддерживаемые системы. У каждого паттерна есть своя ниша, и важно выбирать их в зависимости от того, какие задачи нужно решить в проекте.
Существует множество других паттернов проектирования, которые применяются в различных ситуациях в зависимости от потребностей проекта. Помимо тех, что были рассмотрены, вы можете встретить такие паттерны, как Factory, Prototype, Composite, Flyweight, Facade, State, Strategy и многие другие. Каждый из них решает определённую задачу и помогает в организации кода, улучшении масштабируемости, упрощении тестирования и поддерживаемости приложения.