Научная статья

Original article

АРХИТЕКТУРА VIPER ДЛЯ РАЗРАБОТКИ МОБИЛЬНЫХ ПРИЛОЖЕНИЙ

VIPER ARCHITECTURE FOR MOBILE APPLICATION DEVELOPMENT

Шкарин Егор Алексеевич, бакалавр, Московский государственный технический университет имени Н.Э. Баумана.

Shkarin E.A. schkarin.egor@yandex.ru

Аннотация

В этой статье рассматривается архитектура VIPER для разработки мобильных приложений под IOS. Одна из основных проблем при написании мобильных приложений является выбор правильной архитектуры. Компания Apple предлагает архитектуру MVC, однако, не смотря на всю ее простоту, в некоторых случаях она не является лучшим решением. Так как рано или поздно приложение будет расширятся, что приведет к увеличению кода и существенному его усложнению.

Annotation

This paper discusses the VIPER architecture for developing mobile applications for IOS. One of the main problems in writing mobile apps is choosing the right architecture. Apple offers MVC architecture, but despite its simplicity in some cases it is not the best solution. Sooner or later the application will be extended, which will make the code larger and more complex.

Ключевые слова: архитектура, мобильное приложение, Swift, чистая архитектура, принципы SOLID.

Keywords: architecture, mobile application, Swift, pure architecture, SOLID principles.

Введение

Архитектура VIPER является адаптацией Clean architecture для разработки под IOS. В отличие от MV(x) архитектур она обеспечивает отличную тестируемость и читаемость кода. Основная ее особенность заключается в том, что она максимально приближенно соблюдает принципы SOLID (в особенности принцип единой ответственности). VIPER позволяет максимально разгрузить View (который в UIKit представлен наследниками класса UIViewController), что сильно повышает его читаемость и опять же тестируемость.

Общие понятия

Каждая буква в названии VIPER означает слой (**V**iew–**I**nteractor–**P**resenter–**E**ntity–**R**outer). Каждая часть модуля отвечает за реализацию определённого функционала, что в точности соблюдает принцип единой ответственности. В интернете множество схем VIPER, однако я приведу эту:

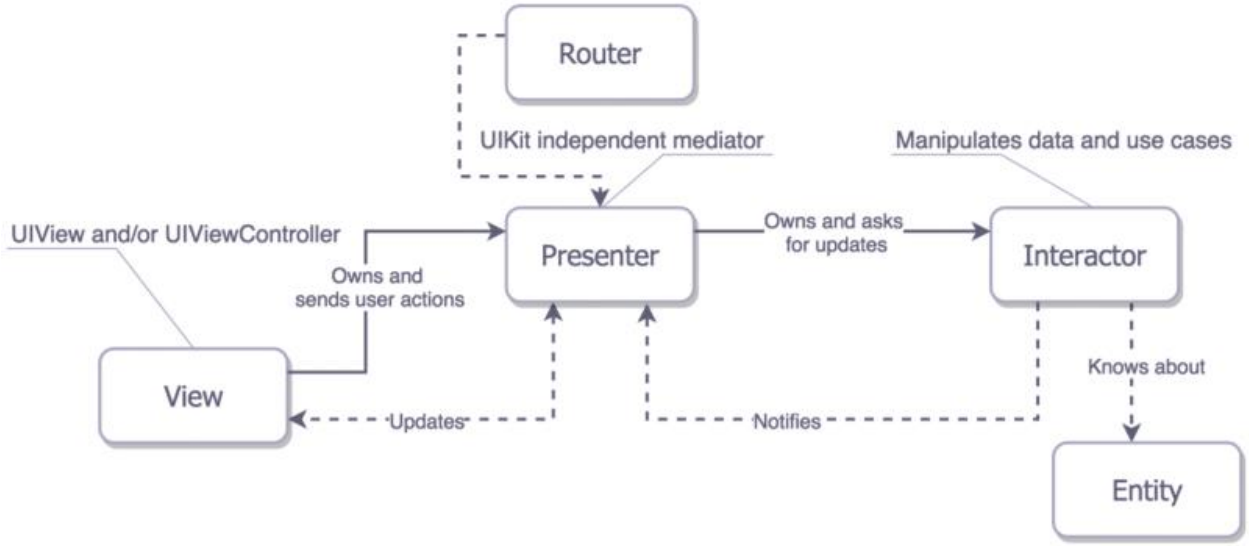

Рисунок 1 – Схема архитектуры VIPER

На этой схеме видно, как слои взаимодействуют между собой. Также немало важно то, что слои взаимодействуют между собой через протоколы, что соответствует принципу инверсии зависимостей. По моему мнению, хорошим решением будет использование input и output протоколов, которые я разберу в следующем разделе. Вот пример одного модуля VIPER:

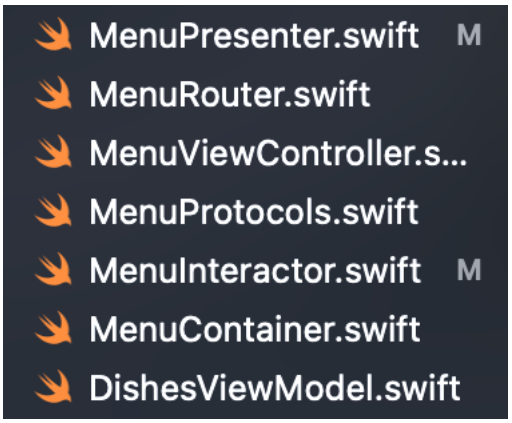

Рисунок 2 – Модуль VIPER

Здесь присутствуют еще несколько классов, которые не относятся к основной архитектуре. О них я расскажу далее в разделах.

Разбор слоев

В этом разделе будут представлены все слои VIPER с небольшими примерами из кода. Начнем с основного слоя.

- Presenter

  Presenter является «мозгом» всего модуля. Он знает о всех слоях, кроме Entity. Presenter принимает от view действия пользователя и решает, что с ними делать. Этот слой содержит логику подготовки данных для отображения во view. Важное правило написания Presenter, в нем не должно быть никакой работы с view, чтобы не путаться можно просто не импортировать библиотеку UIKit в этот файл. Ниже представлен пример Presenter:

```
import Foundation

final class MenuPresenter {
    weak var view: MenuViewInput?
    weak var moduleOutput: MenuModuleOutput?

    private let router: MenuRouterInput
    private let interactor: MenuInteractorInput
    private var numberOfDishCells: Int?
    init(router: MenuRouterInput, interactor: MenuInteractorInput) {
        self.router = router
        self.interactor = interactor
    }
}
```

```
extension MenuPresenter: MenuModuleInput {
}

extension MenuPresenter: MenuViewOutput {

    var advPhoto: String {
        return "photo2"
    }

    var numberOfCellsInAdvSection: Int {
        return 5
    }

    var numberOfCellsInDishSection: Int {
        return numberOfDishCells ?? 0
    }

    var numberOfSections: Int {
        return 2
    }

    func viewDidLoad() {
        interactor.loadData()
    }


    func goToProduct(viewModel: DishesViewModel) {
        router.gotToProductScreen(viewModel: viewModel)
    }

}

extension MenuPresenter: MenuInteractorOutput {
    func catchedError(error: Error) {
        view?.makeErrorNotification(error: error.localizedDescription)
    }

    func didLoadData(dishes: [Dish]) {
        let viewModels = makeViewModels(dishes: dishes)
        numberOfDishCells = viewModels.count
        view?.updateViewWithDishes(dishes: viewModels)
    }

}

private extension MenuPresenter {
    func makeViewModels(dishes: [Dish]) -> [DishesViewModel] {
        return dishes.map { dish in
            let nameOfDish = dish.title.capitalized
            let imageUrl = URL(string: dish.image)
            let ingredients = dish.extendedIngredients.joined(separator: ", ")
            let resource = KingFisherManager.setupResourceForCache(url: imageUrl)
            let htmlLessString = dish.instructions.replacingOccurrences(of: "<[^>]+>", with: "", options:
                .regularExpression, range: nil)
            return DishesViewModel(title: nameOfDish, image: imageUrl, resource: resource,
                extendedIngredients: ingredients, instructions: htmlLessString)
        }
    }
}
```

Рисунок 3 – Пример Presenter

Из примера видно, что Presenter принимает массив объектов Dish и преобразует их в массив viewModel, которые в свою очередь передаются для отображения во view.

- Interactor

Этот слой отвечает за работу с данными и содержит бизнес логику модуля. Interactor взаимодействует с сервисами приложения и моделями (Entity). К сервисам относятся различные менеджеры для работы с сетью, базами данных и т.д. Основная задача этого слоя заключается в получении данных и передачи их в Presenter. Ниже приведены примеры Interactor и менеджера для работы с сетью:

```swift
import Foundation

final class MenuInteractor {
    weak var output: MenuInteractorOutput?
    private var networkManager = NetworkManager()
}

extension MenuInteractor: MenuInteractorInput {
    func loadData() {
        networkManager.getData {[weak self] result in
            switch result {
            case .success(let data):
                guard let dataDict = try? JSONSerialization.jsonObject(with: data, options: []) as?
                    [String: Any],
                        let newData =  dataDict["recipes"] as? [[String: Any]]
                else {
                    return
                }
                var dishes = [Dish]()
                for (index, value) in newData.enumerated() {
                    guard let dish = Dish(dict: value, index: index) else { continue }
                    dishes.append(dish)
                }
                self?.output?.didLoadData(dishes: dishes)
                break
            case .failure(let error):
                self?.output?.catchedError(error: error)
            }
        }
    }
}
```

```swift
import Foundation

protocol NetworkManagerProtocol {

}

final class NetworkManager {
    func getData(completion: @escaping (Result<Data, Error>) -> ()) {
        guard let url = URL(string: "url with api key") else { return }
        let request = URLRequest(url: url)
        URLSession.shared.dataTask(with: request) { data, response, error in
            if let error = error {
                completion(.failure(error))
            }
            guard let data = data
            else {
                return
            }
            completion(.success(data))
        }.resume()
    }

}
```

Рисунок 4 – Пример Interactor

- View

Этот слой отвечает за отображение view с учетом данных полученных из Presenter. Важно чтобы в этом слое выполнялась только работа, связанная с конфигурацией view и его отображением. View может знать только о существовании presenter. Вот пример view:

```swift
import UIKit

final class MenuViewController: UIViewController {
    private let output: MenuViewOutput
    private var collectionView: UICollectionView?
    private var activityIndicator = UIActivityIndicatorView()

    //Views для обработки потери сети
    private let noConnectionLabel = UILabel()
    private let errorLabel = UILabel()
    private let errorButton = UIButton(type: .roundedRect)
    private let errorStackView = UIStackView()

    private var viewModels = [DishesViewModel]()

    init(output: MenuViewOutput) {
        self.output = output
        super.init(nibName: nil, bundle: nil)
    }

    required init?(coder aDecoder: NSCoder) {
        fatalError("init(coder:) has not been implemented")
    }

    override func viewDidLoad() {
        super.viewDidLoad()
        output.viewDidLoad()
        setupNavigationBar()
        setupCollectionView()
        collectionView?.isHidden = true
        setupWaitingIndicator()
    }
    override func viewWillAppear(_ animated: Bool) {
        super.viewWillAppear(animated)
        tabBarController?.tabBar.isHidden = false
    }
}
```

```swift
extension MenuViewController: UICollectionViewDataSource {
    func numberOfSections(in collectionView: UICollectionView) -> Int {
        return output.numberOfSections
    }
    func collectionView(_ collectionView: UICollectionView, numberOfItemsInSection section: Int) -> Int {
        switch section {
        case 0:
            return output.numberOfCellsInAdvSection
        case 1:
            return output.numberOfCellsInDishSection
        default:
            return 0
        }
    }

    func collectionView(_ collectionView: UICollectionView, cellForItemAt indexPath: IndexPath) -> UICollectionViewCell {
        switch indexPath.section {
        case 0:
            let cell = collectionView.dequeueCell(cellType: AdvsCell.self, for: indexPath)
            cell.configure(nameOfPhoto: output.advPhoto)
            return cell
        case 1:
            let cell = collectionView.dequeueCell(cellType: DishCell.self, for: indexPath)
            cell.configure(viewModel: viewModels[indexPath.row])
            return cell
        default:
            return UICollectionViewCell()
        }
    }
```

Рисунок 5 – Пример View

В этом примере видно, что view только вызывает функции из Presenter и конфигурирует view своими внутренними функциями.

- Router

Этот слой отвечает за навигацию в приложении и передачу данных между экранами. Его задача получать от presenter уведомления, о том, что нужно перейти на следующий экран. Пример Router ниже:

```swift
import UIKit

final class MenuRouter {
    var viewController: UIViewController?
}

extension MenuRouter: MenuRouterInput {
    func gotToProductScreen(viewModel: DishesViewModel) {
        let productContext = ProductContext(moduleOutput: nil)
        let productContainer =  ProductContainer.assemble(with: productContext)
        productContainer.input.reciveDish(viewModel: viewModel)
        viewController?.navigationController?.pushViewController(productContainer.viewController, animated: true)
    }
}
```

Рисунок 6 – Пример Router

- Entity

Этот слой, по сути, не относится к самому модулю. Entity это просто модель данных, которой манипулирует interactor и передает в presenter. В entity находятся сырые данных (по сути, это аналогия с Model в MVC или MVVM).

```
import Foundation
struct Dish: Codable {
    let id: Int
    let title: String
    let image: String
    let dishTypes: [String]
    let extendedIngredients: [String]
    let instructions: String

    init?(dict: [String: Any], index: Int) {
        guard let id = dict["id"] as? Int,
              let title = dict["title"] as? String,
              let image = dict["image"] as? String,
              let dishTypes = dict["dishTypes"] as? [String],
              let extendedIngredientsArray = dict["extendedIngredients"] as? [[String: Any]],
              let instructions = dict["instructions"] as? String
        else {
            return nil
        }
        self.id = id
        self.title = title
        self.image = image
        self.dishTypes = dishTypes
        self.extendedIngredients = extendedIngredientsArray.compactMap({ dict in
            return dict["name"] as? String
        })
        self.instructions = instructions
    }
}
```

Рисунок 7 – Пример Entity

На этом разбор основных слоев VIPER закончен. Но есть дополнительные слои, которые не входят в классический VIPER. К таким слоям относятся Container и Protocols.

- Container

Этот слой снимает ответственность по сборки модуля с Router. Это сильно облегчает по количеству кода Router и оставляет ему только одну ответственность по переключению между экранами. Ниже пример Container:

```swift
import UIKit

final class MenuContainer {
    let input: MenuModuleInput
    let viewController: UIViewController
    private(set) weak var router: MenuRouterInput!

    static func assemble(with context: MenuContext) -> MenuContainer {
        let router = MenuRouter()
        let interactor = MenuInteractor()
        let presenter = MenuPresenter(router: router, interactor: interactor)
        let viewController = MenuViewController(output: presenter)

        presenter.view = viewController
        presenter.moduleOutput = context.moduleOutput
        router.viewController = viewController
        interactor.output = presenter

        return MenuContainer(view: viewController, input: presenter, router: router)
    }

    private init(view: UIViewController, input: MenuModuleInput, router: MenuRouterInput) {
        self.viewController = view
        self.input = input
        self.router = router
    }
}
```

Рисунок 8 – Пример Container

- Protocols

По сути, это даже не является слоем. Это просто список протоколов, через которые общаются слои. Вот пример:

```swift
import Foundation

protocol MenuModuleInput {
    var moduleOutput: MenuModuleOutput? { get }
}

protocol MenuModuleOutput: AnyObject {

}

protocol MenuViewInput: AnyObject {
    func updateViewWithDishes(dishes: [DishesViewModel])
    func makeErrorNotification(error: String)
}

protocol MenuViewOutput: AnyObject {
    var numberOfSections: Int { get }
    var numberOfCellsInAdvSection: Int { get }
    var numberOfCellsInDishSection: Int { get }
    var advPhoto: String { get }
    func viewDidLoad()
    func goToProduct(viewModel: DishesViewModel)
}

protocol MenuInteractorInput: AnyObject {
    func loadData()
}

protocol MenuInteractorOutput: AnyObject {
    func didLoadData(dishes: [Dish])
    func catchedError(error: Error)
}

protocol MenuRouterInput: AnyObject {
    func gotToProductScreen(viewModel: DishesViewModel)
}
```

Рисунок 9 – Пример слоя Protocols

Теперь стоит прояснить с input и output протоколами. Это абстракции, которые отвечают за то, откуда и куда передаются данные. Output протоколы

отвечают за передачу данных из слоя в другой слой. Input протоколы отвечают за прием данных в слой.

Сборка стартовых модулей VIPER

В этом разделе стоит начать со старта приложения. Здесь можно идти двумя путями: собирать первый модуль в UISceneDelegate напрямую или работать через отдельный класс называемый Coordinator. При работе через UISceneDelegate напрямую возникают проблемы, так как мы загрязняем лишним кодом этот делегат и даем ему еще одну ответственность по сборке первого модуля. Поэтому, по моему мнению, лучше будет использовать отдельный класс, который будет собирать все необходимые нам модули через вызов функции. Вот пример координатора:

```
import Foundation
import UIKit

protocol CoordinatorProtocol {
    func start()
}

final class Coordinator: CoordinatorProtocol {
    private let window: UIWindow
    private lazy var tabBarController = UITabBarController()
    private lazy var navigationControllers = Coordinator.makeNavigationControllers()
    init(window: UIWindow) {
        self.window = window
    }

    func start() {
        setupMenu()
        setupContacts()
        setupProfile()
        setupBasket()

        let navigationControllers = NavControllerType.allCases.compactMap {
            self.navigationControllers[$0]
        }
        tabBarController.setViewControllers(navigationControllers, animated: true)
        tabBarController.tabBar.barTintColor = UIColor(named: "tabBarColor")
        window.rootViewController = tabBarController
        window.makeKeyAndVisible()
    }
}
```

```
extension Coordinator {

    private func setupMenu() {
        guard let navController = navigationControllers[.menu] else {
            fatalError("No navController")
        }
        let menuContext = MenuContext(moduleOutput: nil)
        let menuContainer = MenuContainer.assemble(with: menuContext)
        navController.setViewControllers([menuContainer.viewController], animated: true)
    }

    private func setupBasket() {
        guard let navController = navigationControllers[.basket] else {
            fatalError("No navController")
        }
        let basketContext = BasketContext(moduleOutput: nil)
        let basketContainer = BasketContainer.assemble(with: basketContext)
        navController.setViewControllers([basketContainer.viewController], animated: true)
    }

    private func setupProfile() {
        guard let navController = navigationControllers[.profile] else {
            fatalError("No navController")
        }
        let profileContext = ProfileContext(moduleOutput: nil)
        let profileContainer = ProfileContainer.assemble(with: profileContext)
        navController.setViewControllers([profileContainer.viewController], animated: true)
    }

    private func setupContacts() {
        guard let navController = navigationControllers[.contacts] else {
            fatalError("No navController")
        }
        let contactsContext = ContactsContext(moduleOutput: nil)
        let contactContainer = ContactsContainer.assemble(with: contactsContext)
        navController.setViewControllers([contactContainer.viewController], animated: true)
    }

    fileprivate static func makeNavigationControllers() -> [NavControllerType: UINavigationController] {
        var result: [NavControllerType: UINavigationController] = [:]
        NavControllerType.allCases.forEach { navControllerKey in
            let navigationController = UINavigationController()
            let tabBarItem = UITabBarItem(title: navControllerKey.title,
                                          image: navControllerKey.image,
                                          tag: navControllerKey.rawValue)
            navigationController.tabBarItem = tabBarItem
            result[navControllerKey] = navigationController

            navigationController.isNavigationBarHidden = true
        }
        return result
    }
}

fileprivate enum NavControllerType: Int, CaseIterable {
    case menu, basket, profile, contacts

    var title: String {
        switch self {
        case .menu:
            return "Меню"
        case .basket:
            return "Избранное"
        case .profile:
            return "Профиль"
        case .contacts:
            return "Контакты"
        }
    }

    var image: UIImage? {
        switch self {
        case .menu:
            return UIImage(named: "menu")
        case .basket:
            return UIImage(named: "basket")
        case .profile:
            return UIImage(named: "profile")
        case .contacts:
            return UIImage(named: "contacts")
        }
    }
}
```

Рисунок 10 – пример Coordinator

И вот пример UISceneDelegate при использовании Coordinator:

```swift
class SceneDelegate: UIResponder, UIWindowSceneDelegate {

    var window: UIWindow?
    var coordinator: CoordinatorProtocol?

    func scene(_ scene: UIScene,
               willConnectTo session: UISceneSession,
               options connectionOptions: UIScene.ConnectionOptions) {
        guard let windowScene = (scene as? UIWindowScene) else { return }
        window = UIWindow(windowScene: windowScene)
        guard let window = window else { return }
        window.overrideUserInterfaceStyle = .light
        coordinator = Coordinator(window: window)
        coordinator?.start()
    }
```

Рисунок 11 – UISceneDelegate при использовании Coordinator

Из примера видно, что код для сборки первой сцены очень большой, так как используется UITabBarController. Однако из-за переноса ответственности за сборку модулей на Coordinator UISceneDelegate сильно облегчается и улучшается его читаемость. Также при усложнении приложения можно использовать несколько координаторов. К примеру, если в приложении присутствует аутентификация, то можно использовать отдельный координатор для нее и отдельный координатор для старта основной части приложения, причем эти два координатора можно объединить в один AppCoordinator, что еще больше облегчит UISceneDelegate, так как у него будет меньше зависимостей.

Сравнение с архитектурами MV(x)

К архитектурам MV(x) можно отнести MVC, MVVM, MVP. Теперь о каждой по порядку.

- MVC

  Это классическая архитектура, рекомендуемая кампанией Apple для своих приложений. В ней есть 3 модуля Model, View, Controller. Несмотря на всю простоту, у этой архитектуры есть сильный недостаток - вся работа с навигацией, данными, сетью и бизнес логика находится в наследниках UIViewController. Это не очень хорошо, так как тестировать такой класс очень трудно, а порой даже невозможно. Здесь очевидно, что VIPER существенно упрощает наследники UIViewController, так как вся бизнес-логика находится в Interactor, за навигацию отвечает Router, а за работу с сетью, базами данных и т.д. отвечают различные дополнительные сервисы.

- MVVM и MVP

  Эти две архитектуры расшифровываются так - Model, View, ViewModel и Model, View, Presenter. MVP и MVVM очень схожи между собой, так как в

первой бизнес логика находится в слое Presenter, а во второй в ViewModel. По сути, эти два слоя аналогия Presenter в VIPER, однако их недостаток заключается в том, что как правило, в больших приложениях Presenter из MVP и ViewModel очень сильно разрастаются, так как делают всю работу с бизнес-логикой, данными, навигацией и т.д. Это основное отличие от Presenter в VIPER. В VIPER Presenter только готовит данные к презентации во view и принимает пользовательские действия, делегируя их выполнение другим слоям.

Заключение

Выбор архитектуры VIPER хорошее решение для большого приложения, так как она позволяет писать код, который потом будет легко читать, тестировать и расширять. Однако VIPER важно использовать только там, где это действительно необходимо. Для небольших приложений эта архитектура может быть излишеством, так как для того, чтобы создать один экран в приложении будет необходимо создавать целый модуль, как минимум из 6 файлом, что не всегда удобно и целесообразно.

Литература

1. Электронный ресурс удаленного доступа (https://web-creator.ru/articles/solid)
2. Architecting iOS Apps with VIPER [электронный ресурс] / Zafar Ivaev, январь 2020. - Режим доступа: https://betterprogramming.pub/how-to-implement-viper-architecture-in-your-ios-app-rest-api-and-kingfisher-f494a0891c43?gi=92e578aec16c, свободный.
3. Routing in VIPER Architecture [электронный ресурс] / Lyle Resnick, сентябрь 2018. Режим доступа: http://lyleresnick.com/blog/2018/09/19/Routing-in-VIPER-Architecture, свободный.
4. Электронный ресурс удаленного доступа (https://developer.apple.com/documentation/uikit/uiscenedelegate)
5. iOS App Architecture: Coordinators [электронный ресурс] / Brendon Justin, 23 июля 2017. Режим доступа: https://readyset.build/ios-app-architecture-coordinators-8faade1763cf, свободный.

Literature

1. Electronic resource of remote access (https://web-creator.ru/articles/solid)
2. Architecting iOS Apps with VIPER [electronic resource] / Zafar Ivaev, January 2020. - Access mode: https://betterprogramming.pub/how-to-implement-viper-architecture-in-your-ios-app-rest- api-and-kingfisher-f494a0891c43?gi=92e578aec16c, free.
3. Routing in VIPER Architecture [electronic resource] / Lyle Resnick, September 2018. Access mode: http://lyleresnick.com/blog/2018/09/19/Routing-in-VIPER-Architecture, free.
4. Remote Access Electronic Resource (https://developer.apple.com/documentation/uikit/uiscenedelegate)
5. iOS App Architecture: Coordinators [electronic resource] / Brendon Justin, July 23, 2017. Access mode: https://readyset.build/ios-app-architecture-coordinators-8faade1763cf, free.

© Шкарин Е.А., 2022 Научно-образовательный журнал для студентов и преподавателей «StudNet» №2/2022.

Для цитирования: Шкарин Е.А. АРХИТЕКТУРА VIPER ДЛЯ РАЗРАБОТКИ МОБИЛЬНЫХ ПРИЛОЖЕНИЙ. // Научно-образовательный журнал для студентов и преподавателей «StudNet» №2/2022.