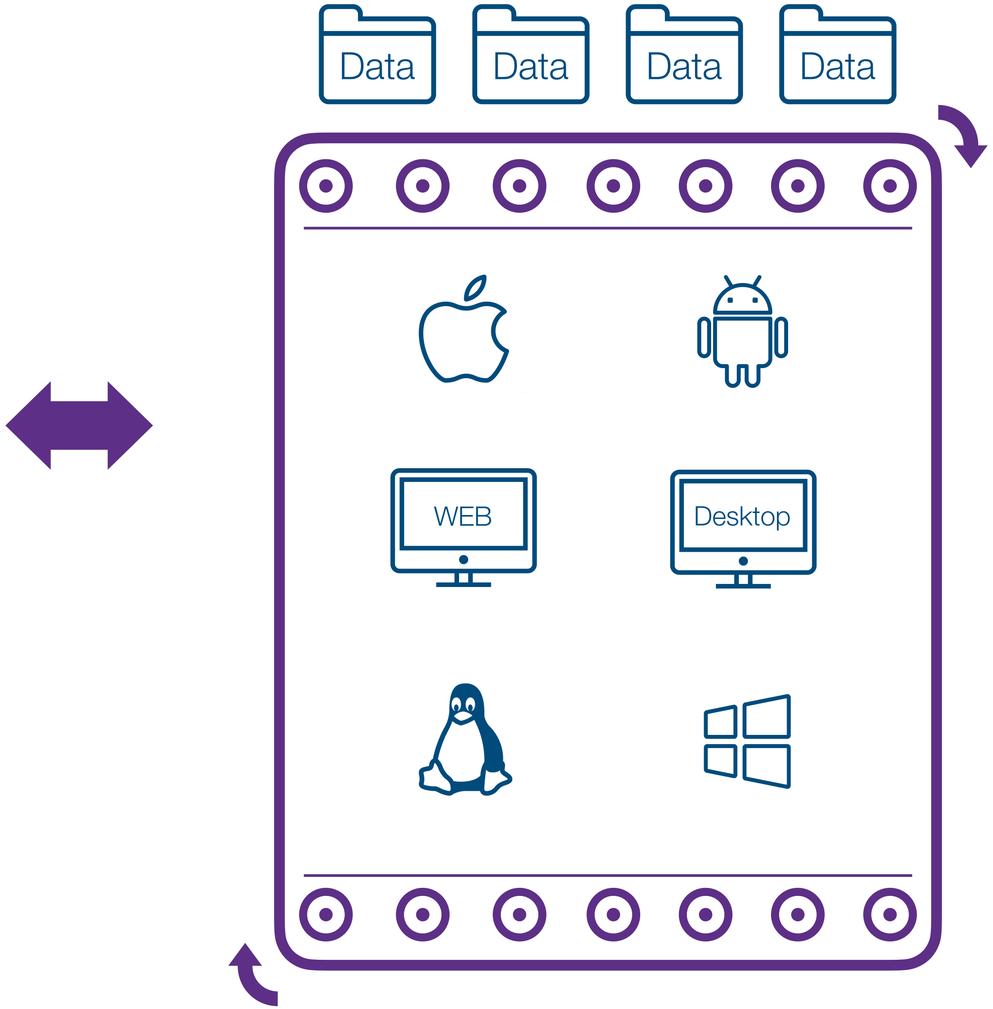
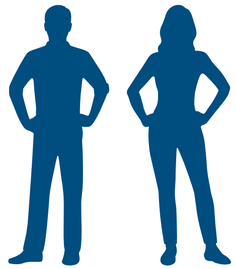
Архитектура сложных клиентских приложений

Юрий Дубовой Делимобиль

Клиентские приложения

Бизнес-требования не зависят от платформы.





Для бизнеса нет отдельных платформ. Есть **каналы обмена данными**.

Архитектура определяется бизнес-требованиями

Для простых проектов нет смысла в сложной архитектуре.

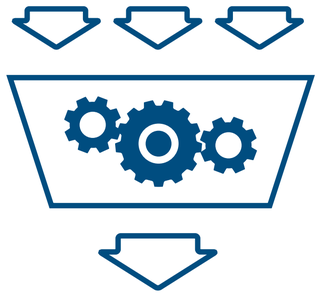
Не окупается.

Сложная архитектура демонстрирует преимущества только в **сложных** проектах.

Источники сложности

State на клиенте

Требуется синхронизация с бекендом



Несколько источников событий

Требуется устранять конфликты и состояния гонки

Удобный UX

Удобство и простота для пользователя обычно означают более сложную реализацию

Маркетинг

Трекинг, аналитика, акции, партнерки, feature toggles и т.п.

Архитектура развивается вместе с бизнес-требованиями

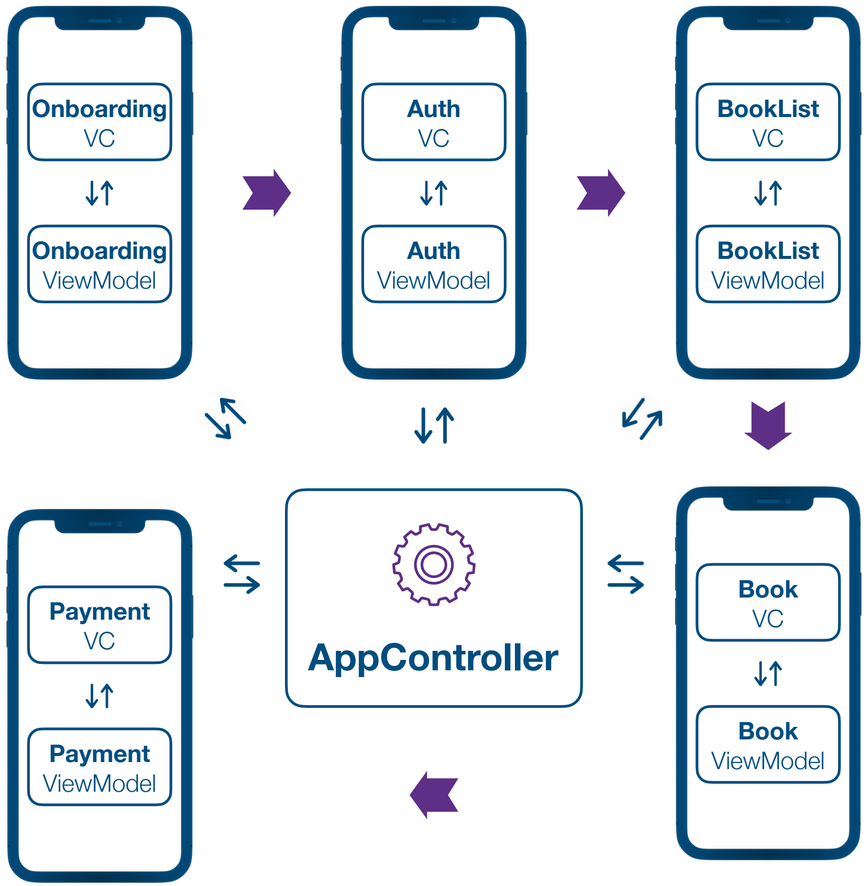
Бизнес растет

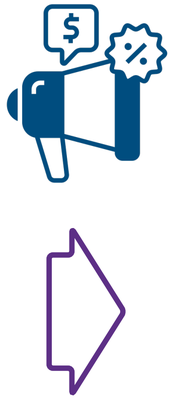
Требования уточняются

Архитектура усложняется

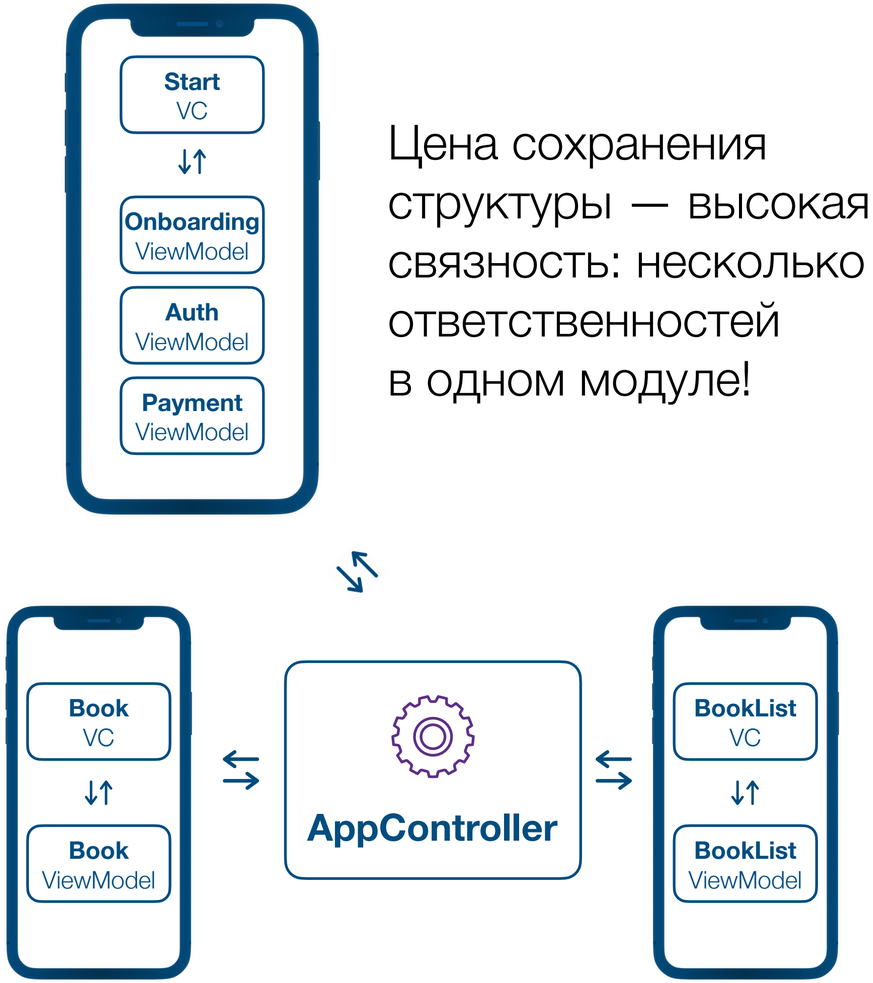
Пластичность: меняем форму, сохраняя структуру

Структура: один экран — один модуль





Маркетинг запрашивает платежи прямо на онбординге



Цена сохранения структуры — высокая связность: несколько ответственностей в одном модуле!

Эволюция проекта и тесты

Начало.
Кода немного

Дружно пишем код, баги ловит менеджер

⬇

Менеджер устал, подключаем тестировщиков

⬇

QA тестит быстрее, чем разработка выдает код

⬇

Нанимаем новых разработчиков =)

Рост.
Кода уже немало

Релизный цикл уменьшается до 1-2 недель

⬇

QA перестает вывозить регрессионное тестирование

⬇

Нам нужна автоматизация QA!

⬇

Нанимаем новых авто-тестировщиков и... разработчиков =)

Кризис.
Спагетти и уныние

Нам нужны тесты!
Разработчики, ваш выход!

⬇

Архитектура не подходит!
Переписываем весь проект

⬇

Переписали, пишем тесты.
Тесты получились хрупкие

⬇

Снова переписываем весь проект
и тесты в придачу =)

Тестируемость: Unit-тесты

Тестируемость — это возможность покрыть Unit-тестами любой класс.

Unit-тесты должны быть

Легкие

Быстро собираются и выполняются

Наглядные

Легко сопоставлять с документацией

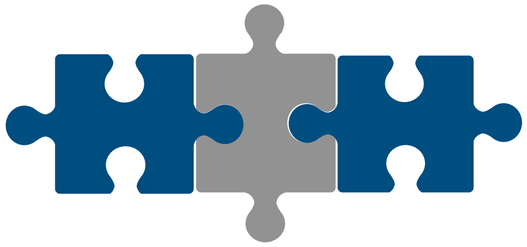
Прочные

Хрупкие тесты ломаются при рефакторинге тестируемого кода

Тестируемость: интеграционные тесты

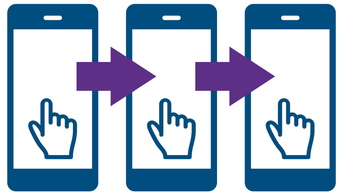
Тестируемость — это возможность применить интеграционное тестирование для любого UserFlow.

Требования к «честным» интеграционным тестам



Чистота модулей

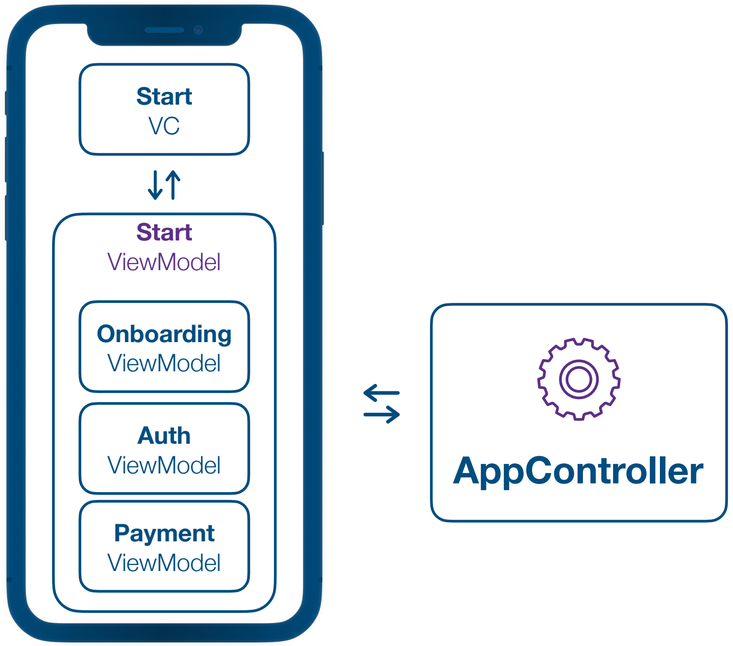
Тестируемые сущности ничего не знают о сценариях их использования, в том числе и о сценариях тестирования.



Не используем UI-тесты

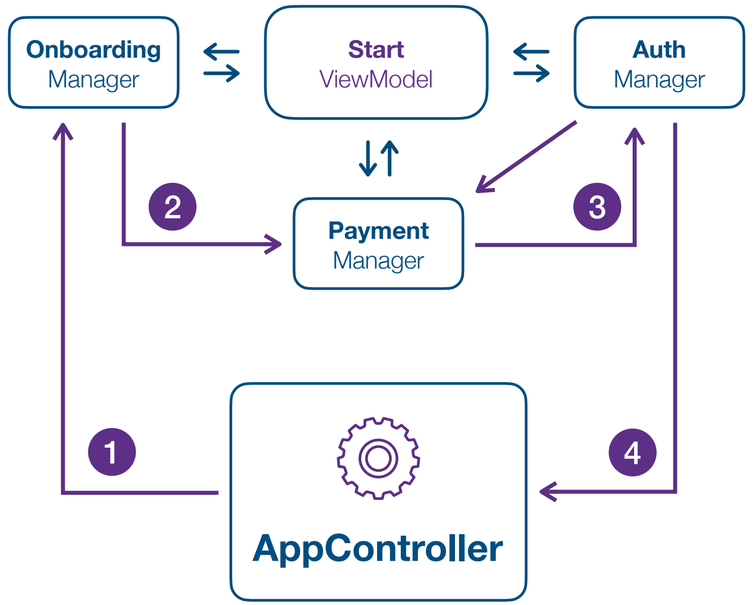
UI-тесты не позволяют проверить граничные случаи. Либо позволяют, но становятся хрупкими.

Декомпозиция объемного модуля. Шаг 1



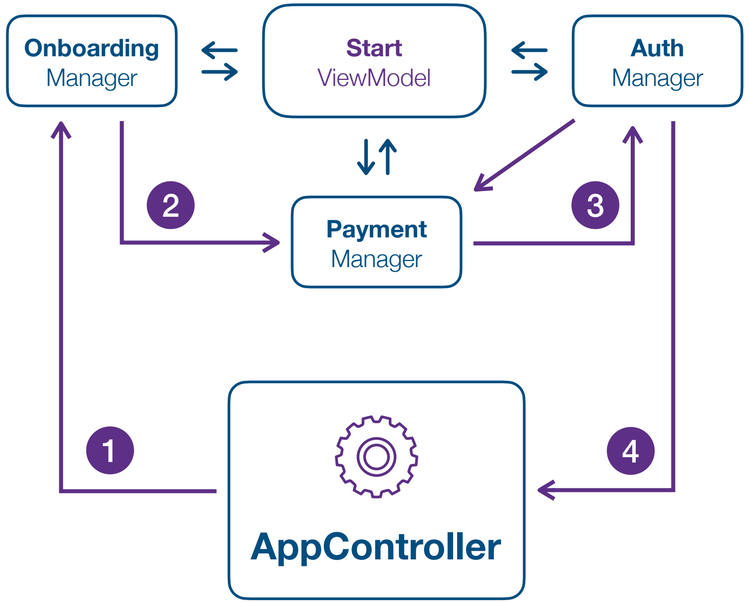
Проблема: в StartViewModel несколько различных ответственностей

Переносим ответственности StartViewModel в «менеджеры»



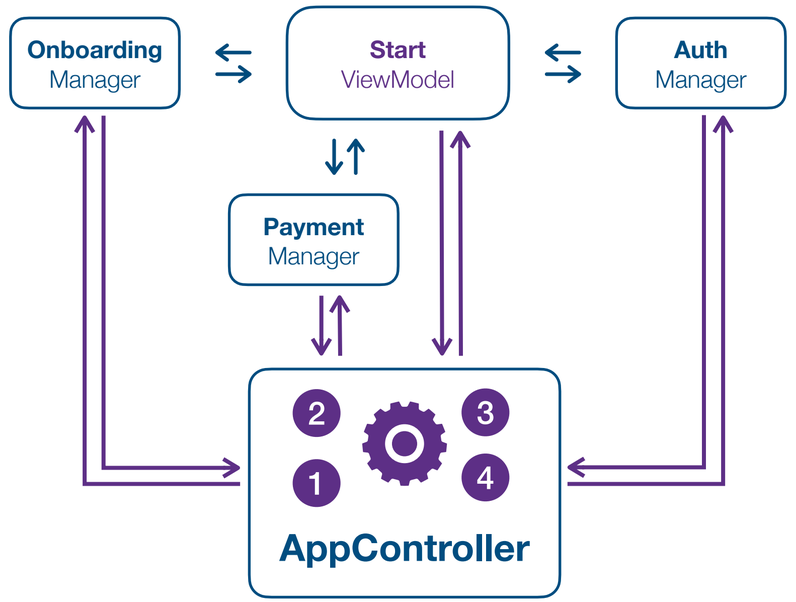
StartViewModel теперь занимается отображением данных, а за логику обмена данными отвечают менеджеры.

Декомпозиция объемного модуля. Шаг 2



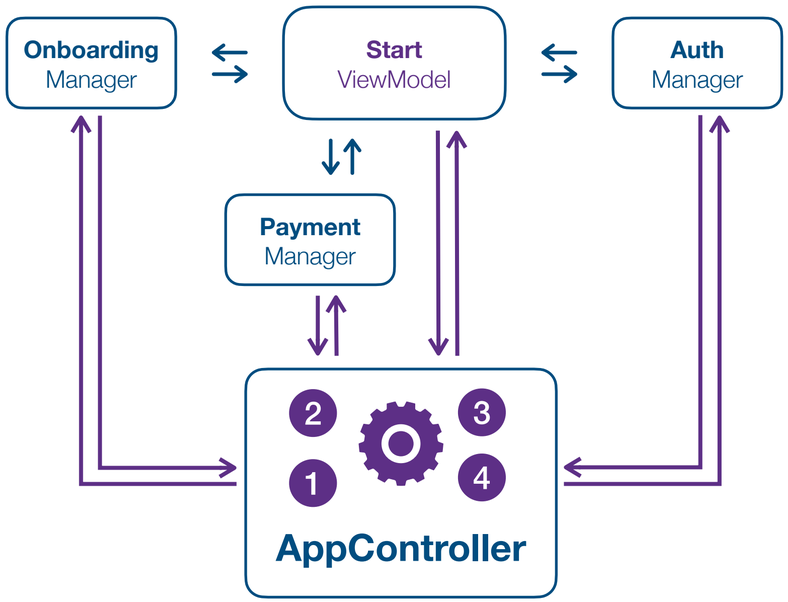
Проблема: менеджеры знают друг о друге. При изменении изменении UserFlow придется переписывать тесты менеджеров.

Переносим управление UserFlow в AppController



Менеджеры предоставляют интерфейсы для AppController. Обмен данными внутри AppController. Unit-тесты менеджеров устойчивы к изменению UserFlow.

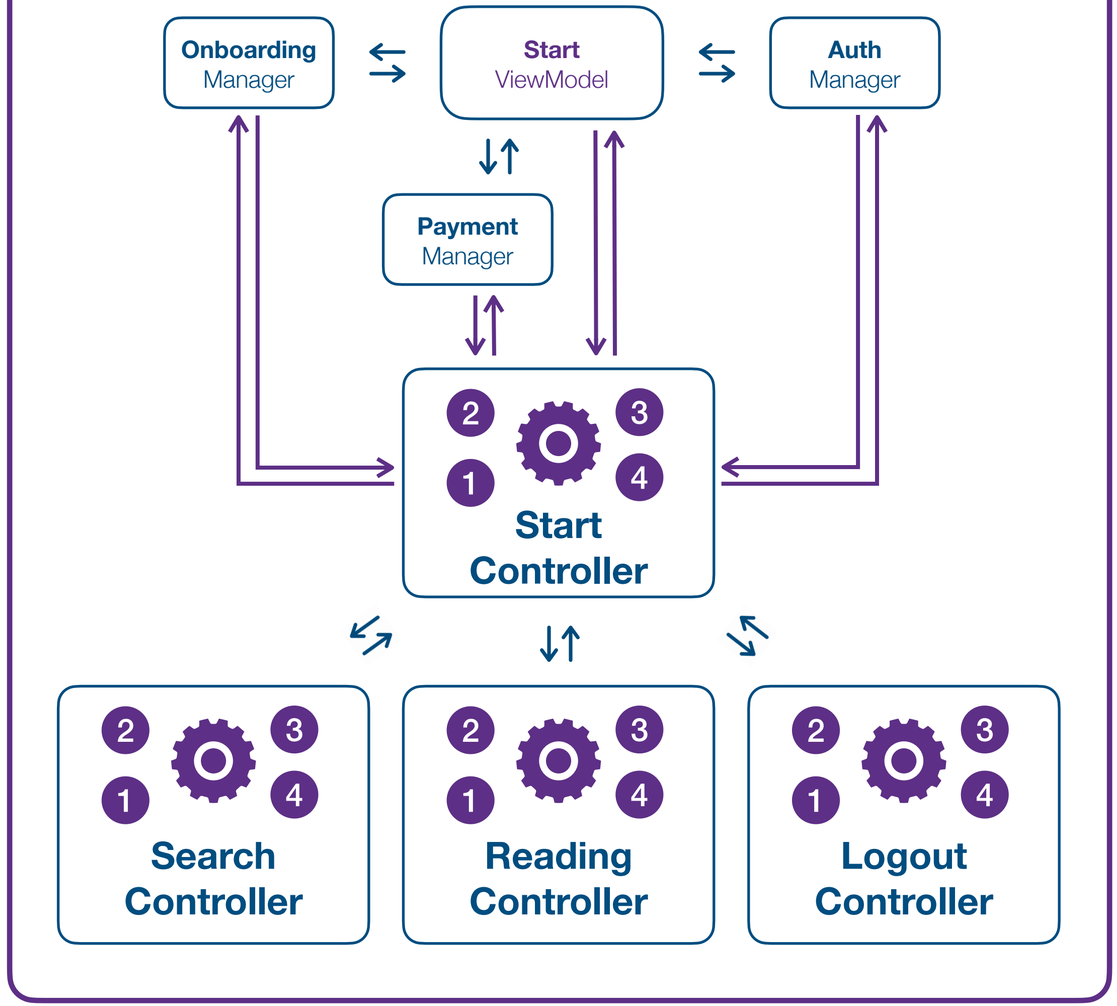
Декомпозиция объемного модуля. Шаг 3



Проблема: AppController теперь содержит не только общую логику, но и логику входа — превращается в God-объект.

Разбиваем AppController на группу контроллеров поменьше.

Каждый мини-контроллер отвечает за свой раздел логики.



Декомпозиция. Какова цена?

Что получили?

- ✅ Наличие Unit-тестов для всех классов
- ✅ Устойчивость Unit-тестов (нехрупкие)
- ✅ Наличие интеграционных тестов (тесты контроллеров)
- ❌ Читаемость кода
- ❌ Гарантия на соответствие кода документации

Проблема: контроллеры приложения содержат логику различного происхождения!

Бизнес-логика
(UserFlow)

Техническая логика
(ограничения платформы)

UI-логика
(навигация)

Маркетинг
(скидка дня 50%)

Наглядность

Наглядность — это возможность представить код графически, в виде диаграмм.

Для клиентских приложений достаточно 5 видов диаграмм.

Диаграмма Состояний

Описывает состояния сущностей и переходы между состояниями, т.е. данные.

Диаграмма деятельности

Описывает логические развилки в UserFlow. Отвечает на вопрос «Что делаем?»

Диаграмма классов

Описывает зависимости между сущностями.

Диаграмма последовательности

Описывает обмен сообщениями между сущностями. Отвечает на вопрос «Как делаем?»

Диаграмма прецедентов

Описывает совокупность всех имеющихся UserFlow.

Сервис

Сервис — это объект, который выполняет задачи, определенные в рамках своей зоны ответственности, реагируя на сообщения.

Сообщение — это обращение к методу или свойству того или иного сервиса.

Свойства сервиса:

1. Обладает императивным интерфейсом
   - синхронный доступ к состоянию
   - асинхронный режим "запрос-ответ"
   - подписки на обновления данных
2. Хранит состояние

Примеры сервисов

Storage — хранит локальный state приложения.

ApiProvider — предоставляет доступ к API сервера по сети

MapSDK — сторонняя библиотека для работы с картами

AppDelegate — посредник при общении между приложением и ОС

AppDelegate — это сервис. Тривиальный, нетестируемый

AppDelegate — это сервис-прародитель, который получает события от UIApplicationMain, т.е. от ОС.

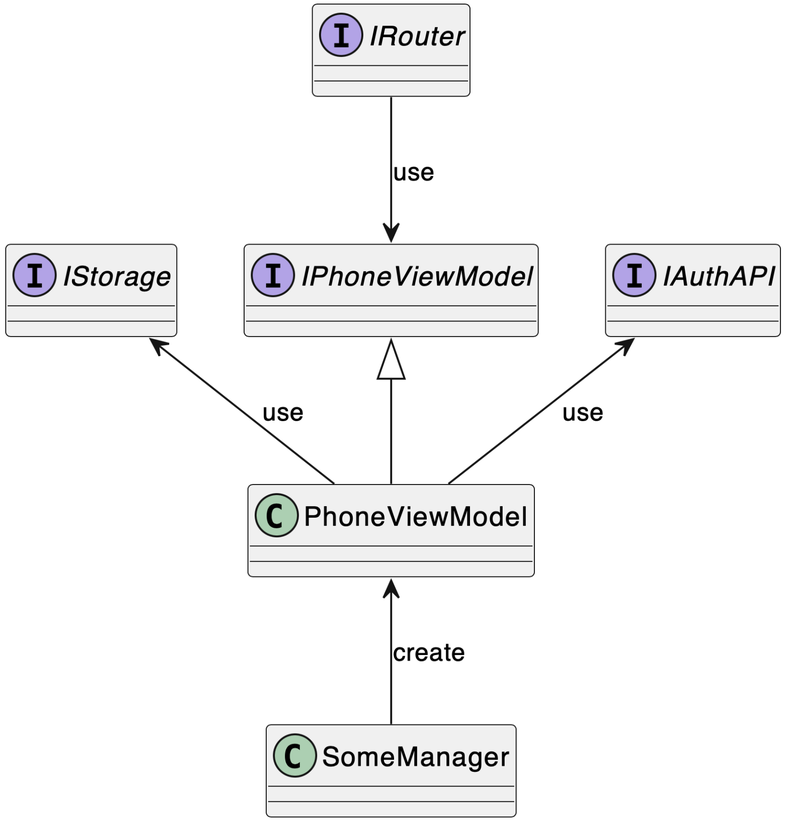
UIApplicationMain — это тоже сервис, который получает сообщения напрямую от ОС. Параметризируется через AppDelegate.

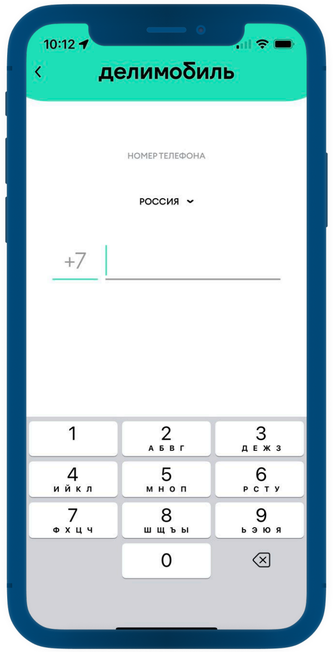
```objc
#import "AppDelegate.h"
int main(int argc, char *argv[])
{
    @autoreleasepool {
        return UIApplicationMain(
            argc,
            argv,
            nil,
            NSStringFromClass([AppDelegate class])
        );
    }
}
```

AppDelegate порождает все остальные сервисы.
Покрыть тестами нельзя, т.к. инициализатор недоступен.
Но можно поддерживать в тривиальном виде.

```swift
import UIKit

@UIApplicationMain
class AppDelegate: UIResponder, UIApplicationDelegate {

    var window: UIWindow?
    var appController: IAppController?

    func application(
        _ application: UIApplication, didFinishLaunchingWithOptions
        launchOptions: [UIApplication.LaunchOptionsKey: Any]?
    ) -> Bool {
        self.appController = AppController() // тривиальный
        self.window = appController?.start() // нетестируемый код
        return true
    }
}

protocol IAppController {
    func start() -> UIWindow
}

class AppController: IAppController {

    func start() -> UIWindow {
        let window = UIWindow()
        window.rootViewController = UINavigationController(
            rootViewController: StartViewController()
        )
        window.makeKeyAndVisible()
        return window
    }
}
```

Диаграмма сервисов

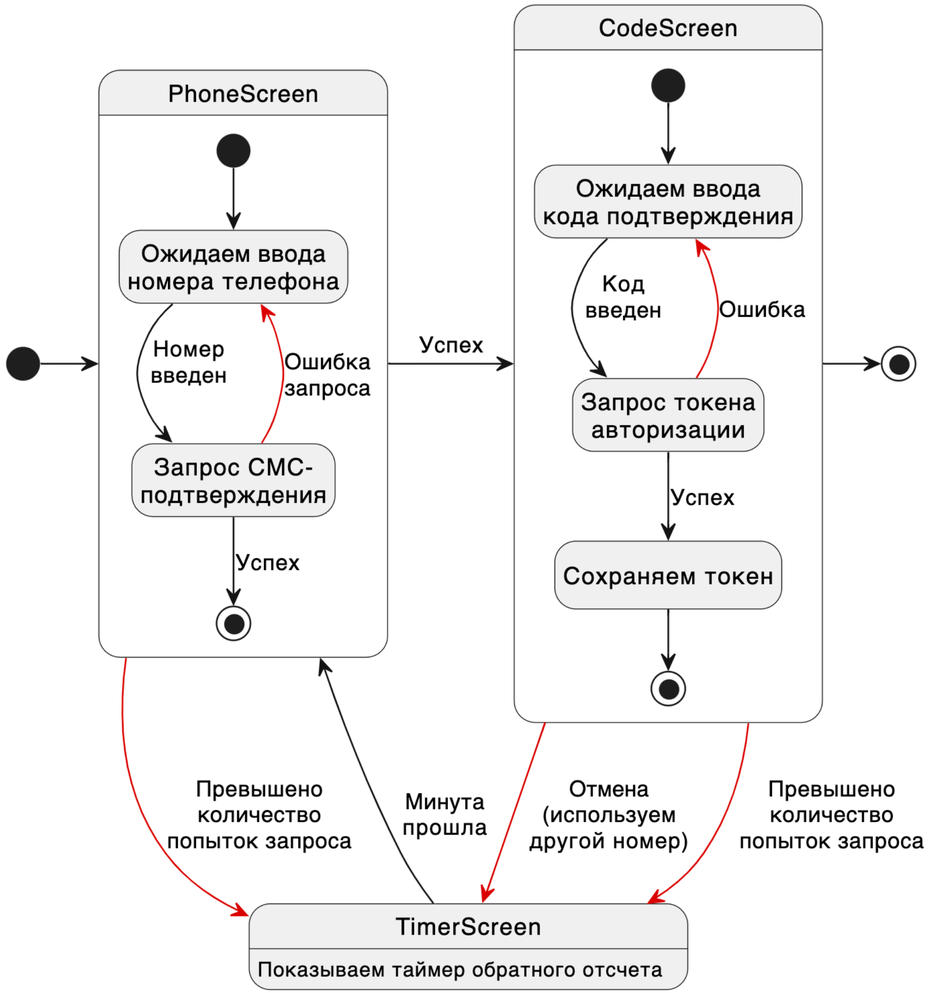




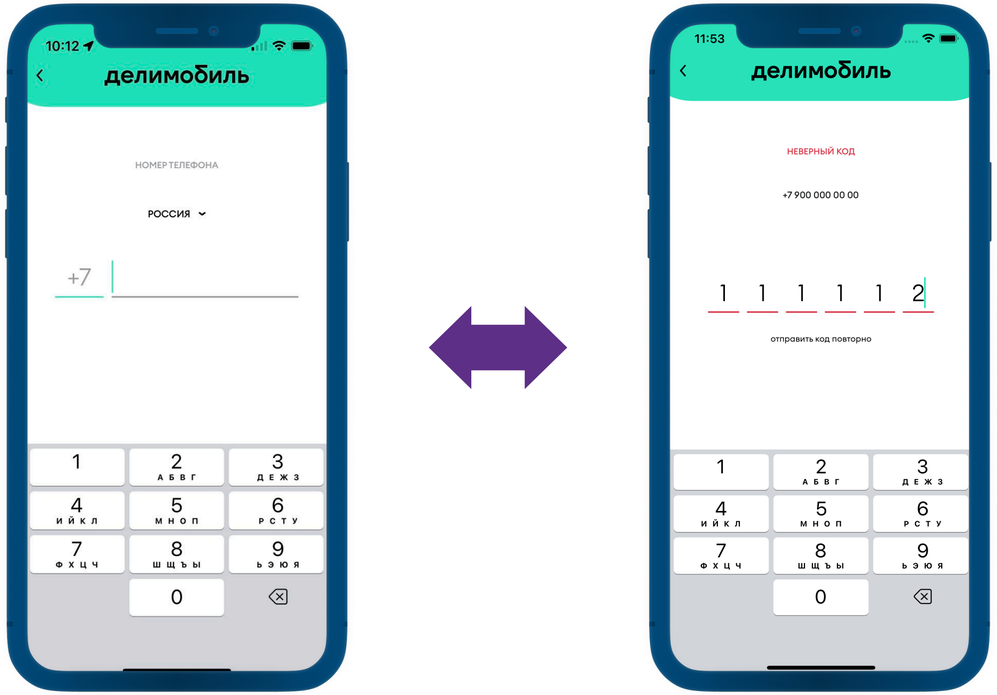

```swift
import Foundation

protocol ISomeManager {
    func goToPhoneScreen()
}

final class SomeManager: ISomeManager {

    private let authAPI: IAuthAPI
    private let router: IRouter
    private let storage: IStorage

    init(
        authAPI: IAuthAPI,
        router: IRouter,
        storage: IStorage
    ) {
        self.authAPI = authAPI
        self.router = router
        self.storage = storage
    }

    func goToPhoneScreen() {
        let phoneVM = PhoneViewModel(
            authAPI: authAPI,
            storage: storage
        )
        router.showPhone(viewModel: phoneVM)
    }
}

protocol IRouter {

    func showPhone(viewModel: IPhoneViewModel) {
    ...
```

Диаграмма сервисов — это упрощенная диаграмма классов, которая отображает зависимости между сервисами.

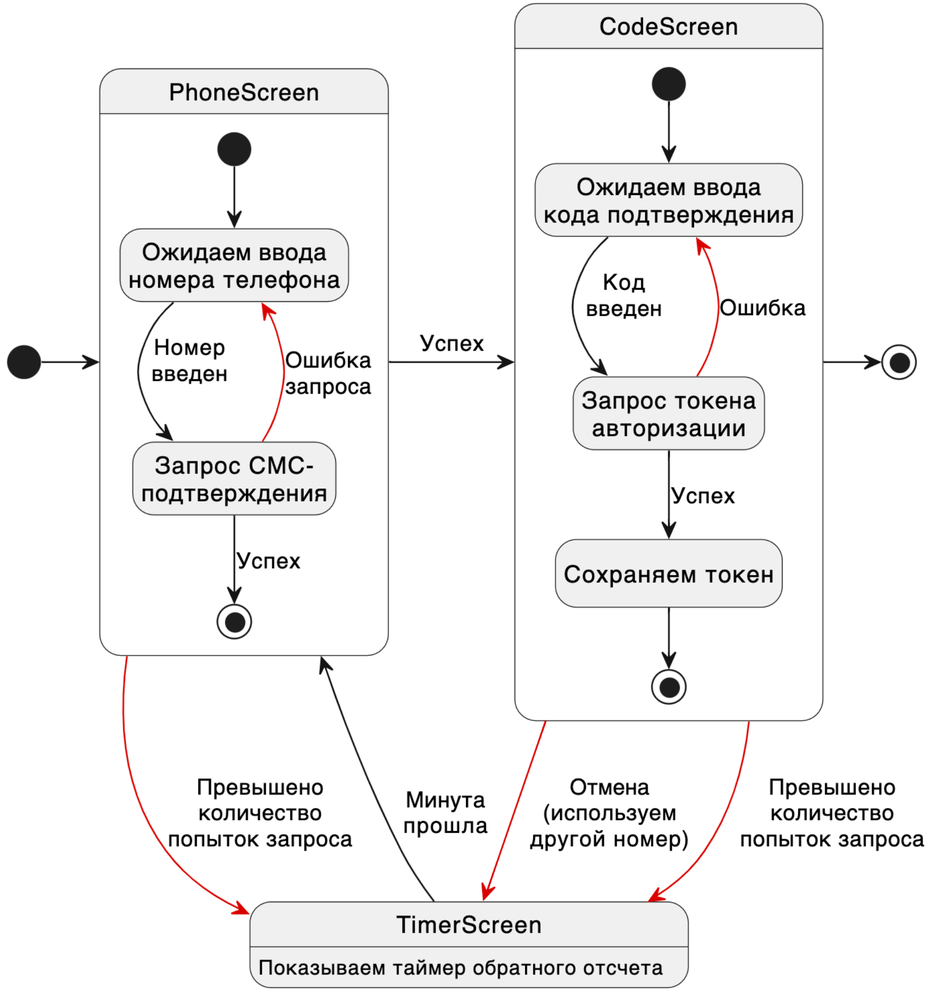
Диаграмма состояний



Сервисы хранят состояние приложения, поэтому описываются с помощью диаграммы состояний.



Сервис разрастается → снижается тестируемость



Проблема: сервис, отвечающий за вход, содержит логику различного происхождения!

Бизнес-логика (ограничение на количество SMS)

Техническая логика (токен авторизации)

UI-логика (обратный отсчет и навигация)

Как протестировать формирование подписок?

AppDelegate порождает сервисы AuthViewModel и UIWindow и настраивает обмен сообщениями (подписку).
Проблема: логика этого обмена в AppDelegate не тестируется.

```swift
import UIKit

@UIApplicationMain
class AppDelegate: UIResponder, UIApplicationDelegate {

    func application(_ application: UIApplication, didFinishLaunchingWithOptions
        launchOptions: [UIApplication.LaunchOptionsKey: Any]?) -> Bool {

        let window = UIWindow()
        let authAPI = AuthAPI()
        let storage = Storage()
        let authVM = AuthViewModel(authAPI: authAPI, storage: storage)
        authVM.onFinish = {
            // в обработчике нетестируемая логика!
            let booksVM = BooksViewModel()
            let booksVC = BooksViewController(viewModel: booksVM)
            let navigationVC = UINavigationController(rootViewController: booksVC)
            window.rootViewController = navigationVC
        }

        let phoneVC = PhoneViewController(viewModel: authVM)
        let navigationVC = UINavigationController(rootViewController: phoneVC)
        window.rootViewController = navigationVC
        window.makeKeyAndVisible()

        return true
    }
}
```

```swift
final class AuthViewModel: IAuthViewModel {
    private let authAPI: IAuthAPI
    private let storage: IStorage
    init(authAPI: IAuthAPI, storage: IStorage) {
        self.authAPI = authAPI
        self.storage = storage
    }
    var onFinish: VoidCompletion = nil
    private var phone = ""

    var isRequestAvailable: Bool { authAPI.isRequestAvailable }

    func apply(phone: String, completion: @escaping VoidHandler) {
        self.phone = phone
        authAPI.requestSMS(phone: phone, completion: completion)
    }

    func check(code: String, onError: ErrorCompletion) {
        authAPI.getAuthToken(
            phone: self.phone, smsCode: code
        ) { [weak self] result in
            switch result {
            case .success(let token):
                self?.storage.save(token: token)
                self?.onFinish?()
            case .failure(let error):
                onError?(error)
            }
        }
    }
```

Операция

Операция — это неделимая последовательность сообщений между сервисами.

```swift
import UIKit

@UIApplicationMain
class AppDelegate: UIResponder, UIApplicationDelegate {

    private var authOp: IAuthOp?

    func application(
        _ application: UIApplication,
        didFinishLaunchingWithOptions launchOptions:
    [UIApplication.LaunchOptionsKey: Any]?) -> Bool {

        let window = UIWindow()
        let authAPI = AuthAPI()
        let storage = Storage()

        self.authOp = AuthOp(
            window: window,
            authAPI: authAPI,
            storage: storage
        )

        self.authOp?.launch(onFinish: { [weak self] in
            self?.authOp = nil
        })

        return true
    }
}
```

```swift
import UIKit

protocol IAuthOp {
    func launch(onFinish: VoidCompletion)
}

final class AuthOp: IAuthOp {
    private let window: UIWindow
    private let authAPI: IAuthAPI
    private let storage: IStorage

    init(window: UIWindow, authAPI: IAuthAPI, storage: IStorage) {
        self.window = window
        self.authAPI = authAPI
        self.storage = storage
    }

    func launch(onFinish: VoidCompletion) {
        let authVM = AuthViewModel(authAPI: authAPI, storage: storage)
        authVM.onFinish = { [weak self] in
            guard let self = self else { return }
            let booksVM = BooksViewModel()
            let booksVC = BooksViewController(viewModel: booksVM)
            let navigationVC = UINavigationController(rootViewController: booksVC)
            self.window.rootViewController = navigationVC
            onFinish?()
        }

        let phoneVC = PhoneViewController(viewModel: authVM)
        let navigationVC = UINavigationController(rootViewController: phoneVC)
        window.rootViewController = navigationVC
        window.makeKeyAndVisible()
    }
}
```

Диаграмма сообщений

```
1  import UIKit
2
3  protocol IAuthOp {
4      func launch(onFinish: VoidCompletion)
5  }
6
7  final class AuthOp: IAuthOp {
8      private let window: UIWindow
9      private let authAPI: IAuthAPI
10     private let storage: IStorage
11
12     init(window: UIWindow, authAPI: IAuthAPI, storage: IStorage) {
13         self.window = window
14         self.authAPI = authAPI
15         self.storage = storage
16     }
17
18     func launch(onFinish: VoidCompletion) {
19         let authVM = AuthViewModel(authAPI: authAPI, storage: storage)
20  1      authVM.onFinish = { [weak self] in
21             guard let self = self else { return }
22             let booksVM = BooksViewModel()
23             let booksVC = BooksViewController(viewModel: booksVM)
24             let navigationVC = UINavigationController(rootViewController: booksVC)
25  4          self.window.rootViewController = navigationVC
26  5          onFinish?()
27         }
28
29         let phoneVC = PhoneViewController(viewModel: authVM)
30         let navigationVC = UINavigationController(rootViewController: phoneVC)
31  2      window.rootViewController = navigationVC
32  3      window.makeKeyAndVisible()
33     }
34 }
```

Операцию описывает диаграмма последовательности,
иначе – диаграмма сообщений.

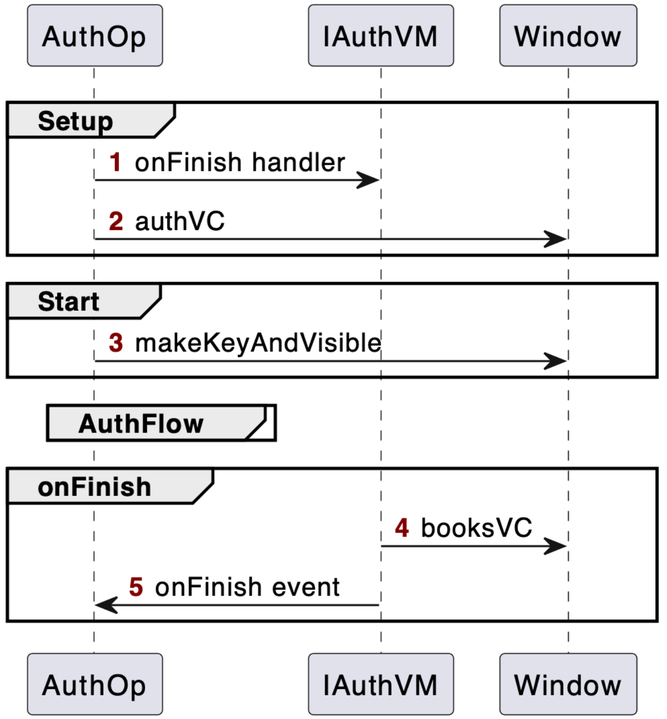
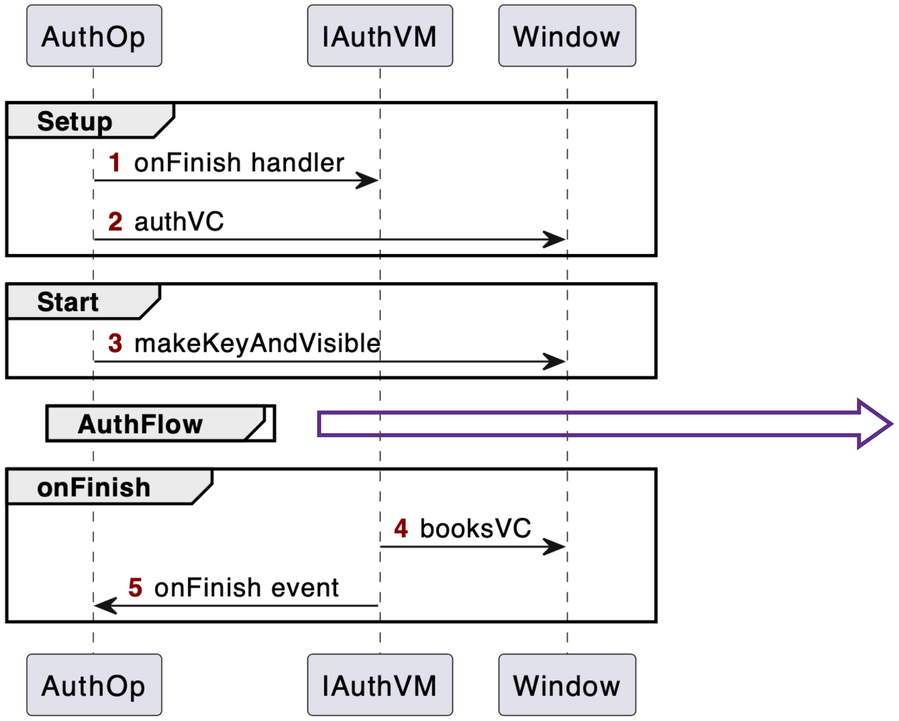
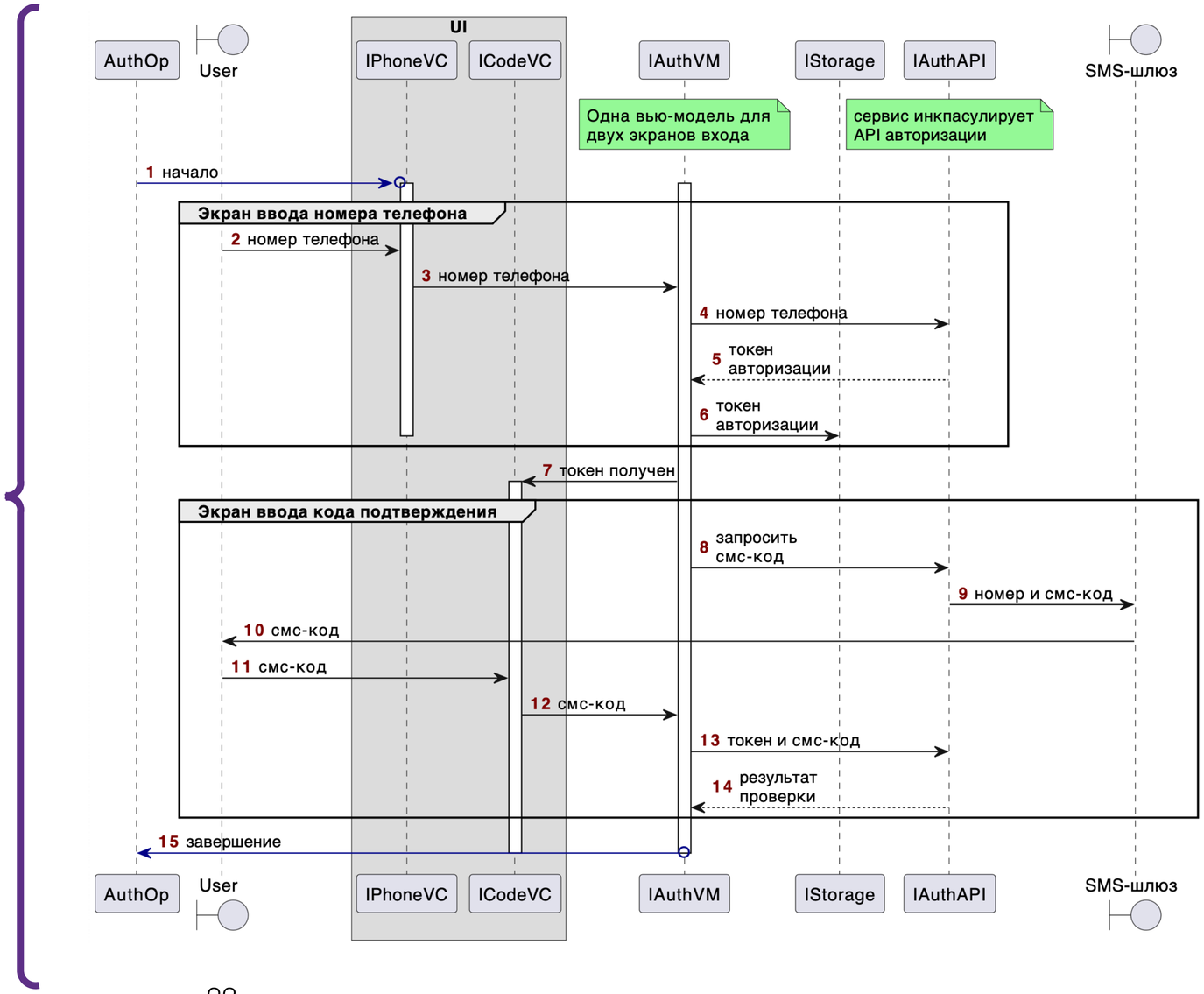


Диаграмма сообщений





Вариативную часть можно перенести в отдельную диаграмму.

Пластичность операций

Фабрика операций имеет доступ к контейнеру с сервисами – можно легко менять зависимости операции.

```
import Foundation

final class OpFactory: IOpFactory {

    private var services: ServiceContainer!

    func make_clearDataOp() -> IClearDataOperation {
        return ClearDataOperation(
            carDataService: services.carDataService,
            inAppStoryService: services.inAppStoryService,
            rentService: services.rentService,
            router: services.router,
            userService: services.userService,
            challengesService: services.challengesService
        )
    }

    func make_logoutOp() -> ILogoutOperation {
        return LogoutOperation(router: services.router,
                               userService: services.userService)
    }

    func make_handlePushTokenOp() -> IHandlePushTokenOp {
        ...

```

Двухэтапная инициализация позволяет обойти циклические зависимости.

```
import Foundation

@UIApplicationMain
class AppDelegate: UIResponder, UIApplicationDelegate {

    var window: UIWindow?

    private let opFactory: IOpFactory = OpFactory()

    // MARK: - Lifecycle

    func application(
        _ application: UIApplication,
        didFinishLaunchingWithOptions
        launchOptions:
    [UIApplication.LaunchOptionsKey: Any]?
    ) -> Bool {
        // создаем сервисы
        let services = opFactory
            .make_makeInitialServicesOp()
            .launch()
        opFactory.setInitialServices(services: services)
        // инициализируем сервисы
        opFactory.make_setupInitialServicesOp().launch()
        // запускаем приложение
        opFactory.make_startOp().launch()

        return true
    }
```

Системные зависимости снижают тестируемость операций

```swift
import UIKit

protocol IAuthOp {
    func launch(onFinish: VoidCompletion)
}

final class AuthOp: IAuthOp {
    private let window: UIWindow
    private let authAPI: IAuthAPI
    private let storage: IStorage

    init(window: UIWindow, authAPI: IAuthAPI, storage: IStorage) {
        self.window = window
        self.authAPI = authAPI
        self.storage = storage
    }

    func launch(onFinish: VoidCompletion) {
        let authVM = AuthViewModel(authAPI: authAPI, storage: storage)
        authVM.onFinish = { [weak self] in
            guard let self = self else { return }
            let booksVM = BooksViewModel()
            let booksVC = BooksViewController(viewModel: booksVM)
            let navigationVC = UINavigationController(rootViewController: booksVC)
            self.window.rootViewController = navigationVC
            onFinish?()
        }

        let phoneVC = PhoneViewController(viewModel: authVM)
        let navigationVC = UINavigationController(rootViewController: phoneVC)
        window.rootViewController = navigationVC
        window.makeKeyAndVisible()
    }
}
```

Внешние зависимости заменяются mock-объектами. Тест операции сводится к проверке состояния mock-ов после запуска теста.

Проблема:
UIKit не тестируется.

```swift
import XCTest

final class AuthTests: XCTestCase {

    final class AuthApiMock: IAuthAPI {
        func requestSMS(phone: String,
                        completion: @escaping VoidHandler) {}
        func getAuthToken(phone: String, smsCode: String,
                          completion: @escaping StringHandler) {}
        var isRequestAvailable: Bool { return true }
    }

    final class StorageMock: IStorage {
        func save(token: String) {}
    }

    func testWindowSetup() throws {
        // MARK: Setup
        let window = UIWindow()
        let authApi = AuthApiMock()
        let storage = StorageMock()
        let authOp = AuthOp(
            window: window,
            authAPI: authApi,
            storage: storage)

        // MARK: Action
        authOp.launch(onFinish: nil)

        // MARK: Test
        XCTAssertNotNil(window.rootViewController)
        XCTAssertTrue(window.isKeyWindow)
    }
}
```

Контекстные сервисы снижают тестируемость операции

```swift
import UIKit

protocol IAuthOp {
    func launch(onFinish: VoidCompletion)
}

final class AuthOp: IAuthOp {
    private let window: UIWindow
    private let authAPI: IAuthAPI
    private let storage: IStorage

    init(window: UIWindow, authAPI: IAuthAPI, storage: IStorage) {
        self.window = window
        self.authAPI = authAPI
        self.storage = storage
    }

    func launch(onFinish: VoidCompletion) {
        let authVM = AuthViewModel(authAPI: authAPI, storage: storage)
        authVM.onFinish = { [weak self] in
            guard let self = self else { return }
            let booksVM = BooksViewModel()
            let booksVC = BooksViewController(viewModel: booksVM)
            let navigationVC = UINavigationController(rootViewController: booksVC)
            self.window.rootViewController = navigationVC
            onFinish?()
        }

        let phoneVC = PhoneViewController(viewModel: authVM)
        let navigationVC = UINavigationController(rootViewController: phoneVC)
        window.rootViewController = navigationVC
        window.makeKeyAndVisible()
    }
}
```

ViewModel — это контекстный сервис, потому что его поведение зависит от контекста. Контекст определяется операцией, порождающей эту ViewModel.

Проблема: ViewModel создается внутри операции и мы не можем заменить его mock-объектом! Как протестировать корректность настройки ViewModel?

Роутер

```swift
import UIKit

final class Router: IRouter {

    var window: UIWindow?

    func setup() {
        let window = UIWindow()
        window.makeKeyAndVisible()
        self.window = window
    }

    func showOnboarding(viewModel: IOnboardingViewModel) {
        // Вью-контроллеры инстанцирует исключительно роутер!
        let onboardingVC = OnboardingViewController(viewModel: viewModel)
        let navVC = UINavigationController(rootViewController: onboardingVC)
        window?.rootViewController = navVC
    }

    func showPhone(viewModel: IAuthViewModel) {
        let phoneVC = PhoneViewController(viewModel: viewModel)
        // Роутер сам создал все вью-контроллеры, поэтому состояние и предысторию
        // UI-слоя знает лучше всех. Если онбординг уже показывался, роутер откроет
        // экран входа через уже готовый UINavigationController с анимацией
        if let navVC = window?.rootViewController as? UINavigationController {
            navVC.pushViewController(phoneVC, animated: true)
        } else {
            let navVC = UINavigationController(rootViewController: phoneVC)
            window?.rootViewController = navVC
        }
    }

    func showBooks(viewModel: IBooksViewModel) {

    ...
```

Router — это ключевой сервис, который отвечает за весь UI-слой.

Роутер и только роутер инстанцирует вью-контроллеры, поэтому роутер содержит UI-состояние и отвечает за UI-логику.

Роутер инкапсулирует UI. Снаружи роутер — это просто еще один сервис, один из многих. Сервис, который умеет выводить изображение на экран и воздействовать на вью-модели.

Роутер обеспечивает высокоуровневые интеграционные тесты и кроссплатформенность бизнес-логики.

Роутер обеспечивает тестируемость операций

```swift
import UIKit

protocol IAuthOp {
    func launch(onFinish: VoidCompletion)
}

final class AuthOp: IAuthOp {
    private let window: UIWindow
    private let authAPI: IAuthAPI
    private let storage: IStorage

    init(window: UIWindow, authAPI: IAuthAPI, storage: IStorage) {
        self.window = window
        self.authAPI = authAPI
        self.storage = storage
    }

    func launch(onFinish: VoidCompletion) {
        let authVM = AuthViewModel(authAPI: authAPI, storage: storage)
        authVM.onFinish = { [weak self] in
            guard let self = self else { return }
            let booksVM = BooksViewModel()
            let booksVC = BooksViewController(viewModel: booksVM)
            let navigationVC = UINavigationController(rootViewController: booksVC)
            self.window.rootViewController = navigationVC
            onFinish?()
        }

        let phoneVC = PhoneViewController(viewModel: authVM)
        let navigationVC = UINavigationController(rootViewController: phoneVC)
        window.rootViewController = navigationVC
        window.makeKeyAndVisible()
    }
}
```

```swift
import Foundation

protocol IAuthOp {
    func launch(onFinish: VoidCompletion)
}

final class AuthOp: IAuthOp {
    private let authAPI: IAuthAPI
    private let router: IRouter
    private let storage: IStorage

    init(authAPI: IAuthAPI, router: IRouter, storage: IStorage) {
        self.authAPI = authAPI
        self.router = router
        self.storage = storage
    }

    func launch(onFinish: VoidCompletion) {
        let authVM = AuthViewModel(authAPI: authAPI, storage: storage)
        authVM.onFinish = { [weak self] in
            self?.router.showBooks(viewModel: BooksViewModel())
            onFinish?()
        }
        router.setup()
        router.showPhone(viewModel: authVM)
    }
}
```

Роутер инкапсулирует UI (UIKit), поэтому все зависимости операции могут быть представлены mock-объектами.

Роутер инкапсулирует UI → обеспечивает кроссплатформу

Платформа
(iOS)

AppDelegate, APNs,
App Extensions,
InApp Purchases

Swift

Бизнес-логика
(UserFlow)

Маркетинг
(deeplinks)

Ключевые сервисы,
операции, вью-модели,
IRouter (интерфейс)

KMM (Кобол МультиплатформМ)

UI-логика
(навигация,
анимация)

Router (реализация),
Core Animation,
UIKit, UIEvent

Swift

Роутер = пластичность, наглядность и тестируемость

Тестируемость

Закрываем проблему тестирования связки между вью-моделью и операцией:
1. RouterMock воздействует на ViewModel →
2. ViewModel эмитирует события в замыкание-подписку (сформирована внутри операции) →
3. замыкание отправляет сообщения к сервисам-зависимостям операции →
4. mock-зависимости уведомляют тест.

Наглядность

Вью-модель в связке с роутером изолирует бизнес-логику от UI-логики, поэтому операция по-прежнему описывается диаграммой сообщений.

Пластичность

Можем запросить из любой операции показ любого экрана!

Масштабируемость?

Межоперационная логика и диаграмма операций

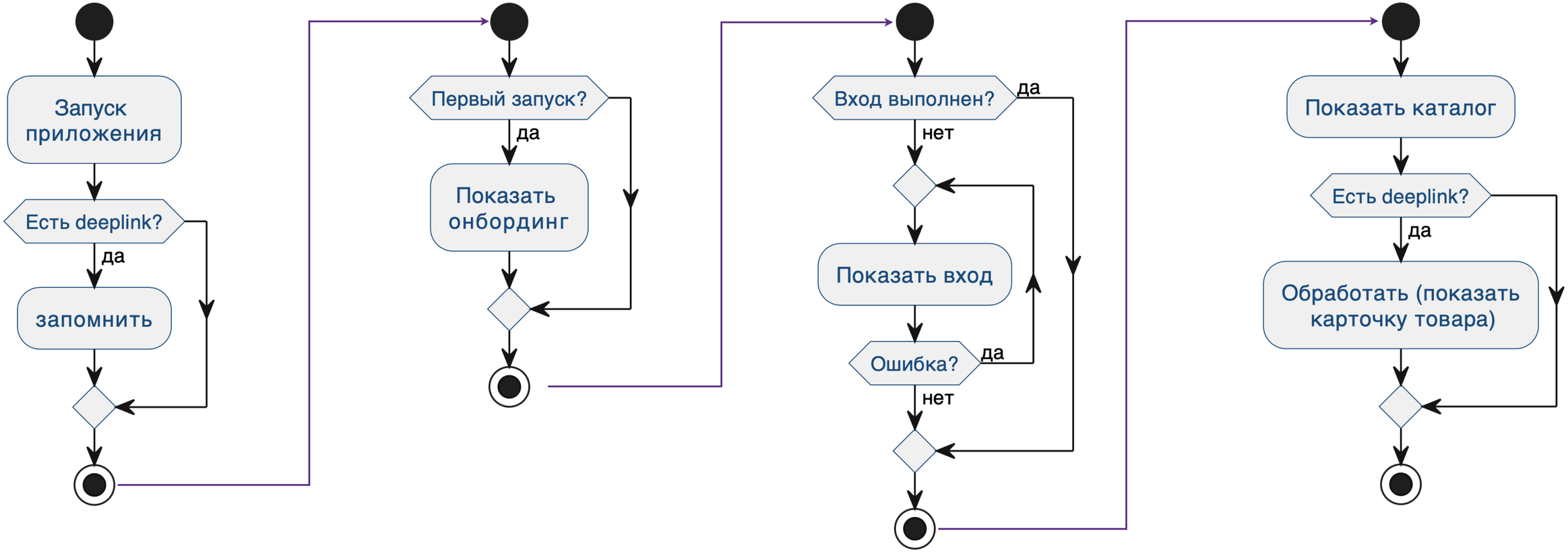


Диаграмма операций запуска приложения.
Ее может построить не только программист, но и бизнес-аналитик – на основе User Stories.

Сценарий

Сценарий — это последовательность операций.

Как только у операции возникает потребность обратиться к другой операции, она либо превращается в сценарий, либо делегирует задачу в какой-либо сервис. Операция неделима!

У сценария, как и у операции, нет императивного интерфейса и состояния.

```swift
import Foundation

final class LogoutScenario: ILogoutScenario {

    private let opFactory: IOpFactory

    private let retainer = Retainer()

    init(opFactory: IOpFactory) {
        self.opFactory = opFactory
        retainer.retain(link: self)
    }

    private func finish(_ onFinish: @escaping VoidCompletion) {
        onFinish()
        retainer.release()
    }

    func launch(type: LogoutType, onFinish: @escaping VoidCompletion) {

        switch type {
        case .regular:
            let logoutOperation = opFactory.make_logoutOp()
            logoutOperation.launch() { [weak self] in
                guard let self = self else { return }
                let clearDataOp = self.opFactory.make_clearDataOp()
                clearDataOp.launch()
                self.finish(onFinish)
            }
        case .forced:
            let clearDataOp = opFactory.make_clearDataOp()
            clearDataOp.launch()
            finish(onFinish)
        }
    }
}
```

Тестируемость сценария: фабрика операций

```swift
import Foundation

final class LogoutScenario: ILogoutScenario {

    private let opFactory: IOpFactory

    private let retainer = Retainer()

    init(opFactory: IOpFactory) {
        self.opFactory = opFactory
        retainer.retain(link: self)
    }

    private func finish(_ onFinish: @escaping VoidCompletion) {
        onFinish()
        retainer.release()
    }

    func launch(type: LogoutType, onFinish: @escaping VoidCompletion) {

        switch type {
        case .regular:
            let logoutOperation = opFactory.make_logoutOp()
            logoutOperation.launch() { [weak self] in
                guard let self = self else { return }
                let clearDataOp = self.opFactory.make_clearDataOp()
                clearDataOp.launch()
                self.finish(onFinish)
            }
        case .forced:
            let clearDataOp = opFactory.make_clearDataOp()
            clearDataOp.launch()
            finish(onFinish)
        }
    }
}
```

```swift
import XCTest
final class LogoutScenarioTests: XCTestCase {
    func testLaunch() {
        // MARK: Arrange
        let opFactory = OpFactoryMock()
        opFactory.onLogoutOp = {
            testStage1()
        }
        opFactory.onClearDataOp = {
            testStage2()
        }
        let scenario = LogoutScenario(opFactory: opFactory)
        let exp = expectation(description: "LogoutScenario")

        // MARK: Action
        scenario.launch(type: .regular, onFinish: {
            exp.fulfill()
        })

        // MARK: Test
        func testStage1() {
            XCTAssertTrue(opFactory.logoutOpMade)
            XCTAssertFalse(opFactory.clearDataOpMade)
        }
        func testStage2() {
            XCTAssertTrue(opFactory.logoutOpMade)
            XCTAssertTrue(opFactory.clearDataOpMade)
        }
        waitForExpectations(timeout: 1.0)
    }
}
```

Тестируемость сценария: фабрика сценариев и AppService

```swift
import Foundation

final class AppService: IAppService {

    private var scenarioFactory: IScenarioFactory!

    func setScenarioFactory(scenarioFactory: IScenarioFactory) {
        self.scenarioFactory = scenarioFactory
    }

    func logout(type: LogoutType) {
        let logoutScenario = self.scenarioFactory.makeLogoutScenario()
        logoutScenario.launch(type: type) { [weak self] in
            self?.appCoordinator.clear()
            self?.showAuth(isFromAppLaunch: false)
        }
    }

    func showAuth(isInitial: Bool) {
        let authScenario = scenarioFactory.makeAuthScenario()
        authScenario.launch(isFromAppLaunch: isInitial)
    }

    func showMain(context: ShowMainContext, onFinish: VoidCompletion?) {
        let showMainScenario = scenarioFactory.makeShowMainScenario()
        showMainScenario.launch(context: context, onFinish: onFinish)
    }

    func handle(deepLinkUrl: URL, context: HandleDeepLinkContext) {
        scenarioFactory
            .makeHandleDeepLinkScenario()
            .launch(url: deepLinkUrl, context: context)
    }

    ...
```

Сценарии формируются в фабрике сценариев. Доступ к фабрике есть у специального сервиса — AppService.

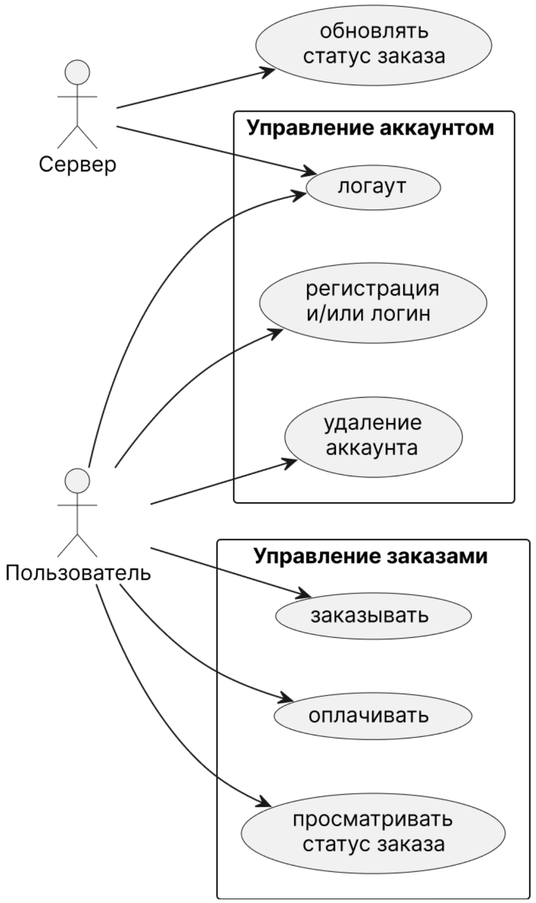
Сценарий ничего не знает о других сценариях и не может их запускать. Обмен данными между сценариями выполняется с помощью AppService.

Весь контекст, необходимый для работы, сценарий должен получить в методе launch.

Хороший сценарий в зависимости принимает только:
1) фабрику операций;
2) AppService.

Дополнительные сервисы-зависимости допустимы, но их тогда придется поддерживать в тестах сценария.

Пластичность сценария и диаграмма прецедентов



Доступ к AppService обеспечивает сценарию пластичность. Сценарий может в любой момент запросить запуск другого сценария, после чего завершить работу.

Диаграмма прецедентов — верхнеуровневое описание проекта. Отображает сценарии и отдельные операции, которые используются AppService.

Диаграмма отображает источники событий, воздействующие на приложение.

Три кита архитектуры

Тестируемость

обеспечена тем, что мы можем покрыть Unit-тестами все составляющие приложения: сервисы, операции, сценарии и даже UI-логику в роутере.

Пластичность

обеспечена возможностью перестраивать операции и сценарии, не прибегая к масштабному рефакторингу — зависимости всегда под рукой.

Наглядность

обеспечена диаграммами сообщений, операций, и прецедентов, которые однозначно отображаются в код. Диаграммы — это общий язык для аналитиков, проектировщиков, программистов и QA.

ROSS или POCC? Вот в чем вопрос! =)

Router
Pоутер

Отвечает за весь UI и UI-логику, в том числе навигацию. Позволяет разрабатывать приложение в headless-режиме.

Operation
Oперация

Описывает неделимую цепочку сообщений между сервисами. Обеспечивает интеграционное тестирование.

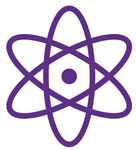
Service
Cервис

Основной строительный компонент. Обладает императивным интерфейсом и состоянием.

Scenario
Cценарий

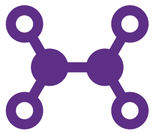
Отображение UserFlow в код. Состоит из операций и объединяющей их бизнес-логики. Обеспечивает высокоуровневую наглядность.

Правила ROSS

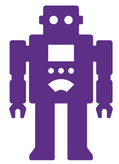


Операция неделима и не имеет доступа к другим операциям (фабрике операций)

Императивный интерфейс и состояние есть только у сервисов.



Сценарий не имеет доступа к другим сценариям. Используется сервис-посредник.



Любые манипуляции с UI выполняются через роутер.

Покрытие Unit-тестами всей кодовой базы на 100% невозможно и бессмысленно, но нетестируемую часть можно изолировать.

ROSS на практике

делимобиль — это **сложное клиентское** приложение

Сложные сценарии
проверка документов, осмотр авто, рассрочка, завершение аренды и др.

Контекстно-зависимый, композитный UI
пользователь одновременно работает с картой и управляющей панелью, где многие элементы зависят друг от друга

Синхронизируемые состояния
статус аренды может поменяться в любой момент, множество кешей

Самый большой каршеринг в мире

20 тысяч автомобилей

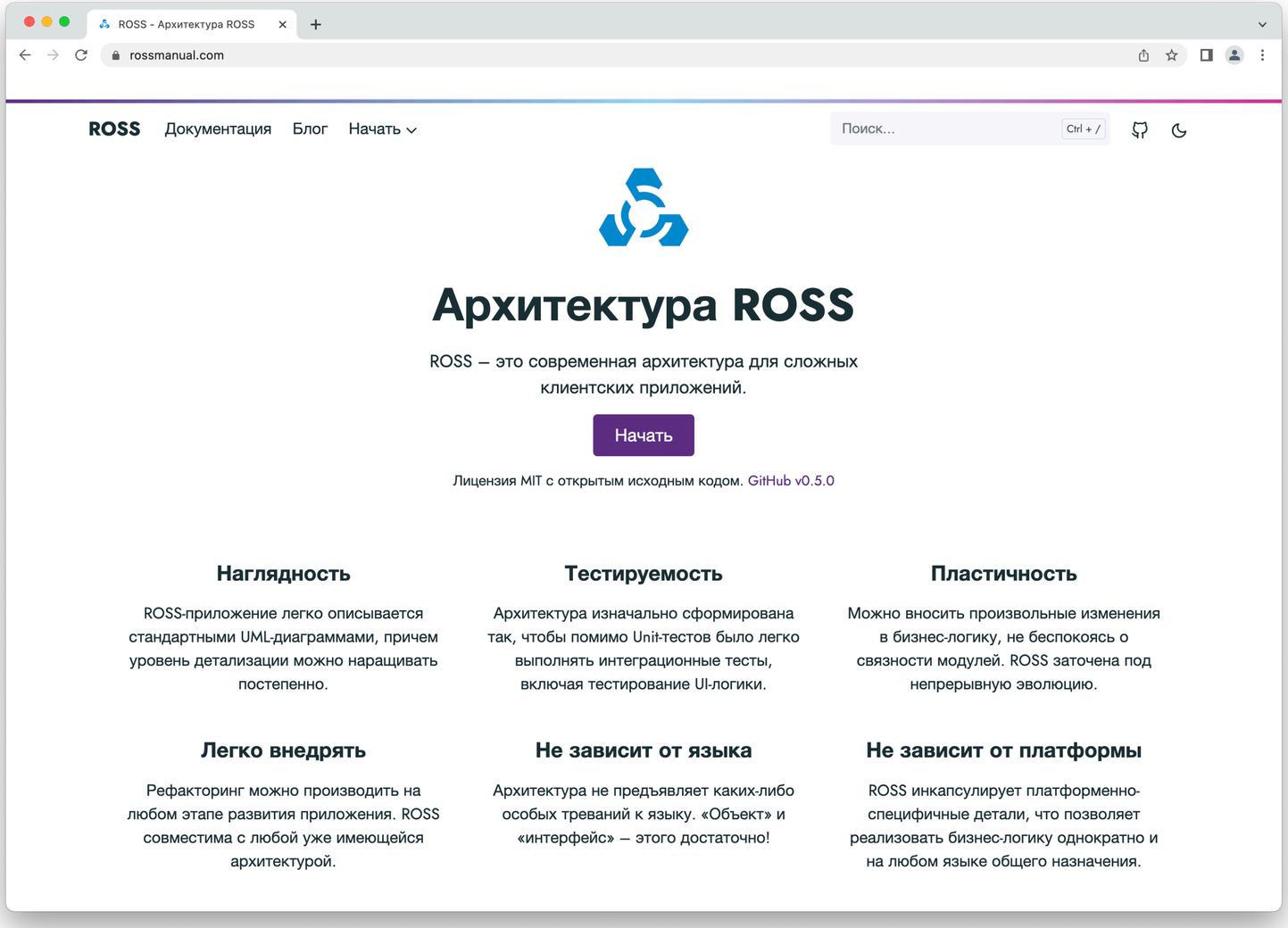
50% рынка по данным Департамента транспорта в Москве

3 страны и 9 городов присутствия в РФ

Компактная команда

4 разработчика
на каждую платформу
и при этом crash-free 99.9%

Документация



https://rossmanual.com

https://t.me/yury_dubovoy

https://t.me/rossmanual