Как заменить архитектуру в iOS приложении?

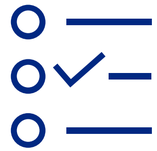
Иван Будников

IT-лид

Про что поговорим

С какими проблемами столкнулись

Как заменили архитектуру



Что получили в итоге

Стек проекта

- MVC
- Swift
- CocoaPods
- CoreData
- URLSession

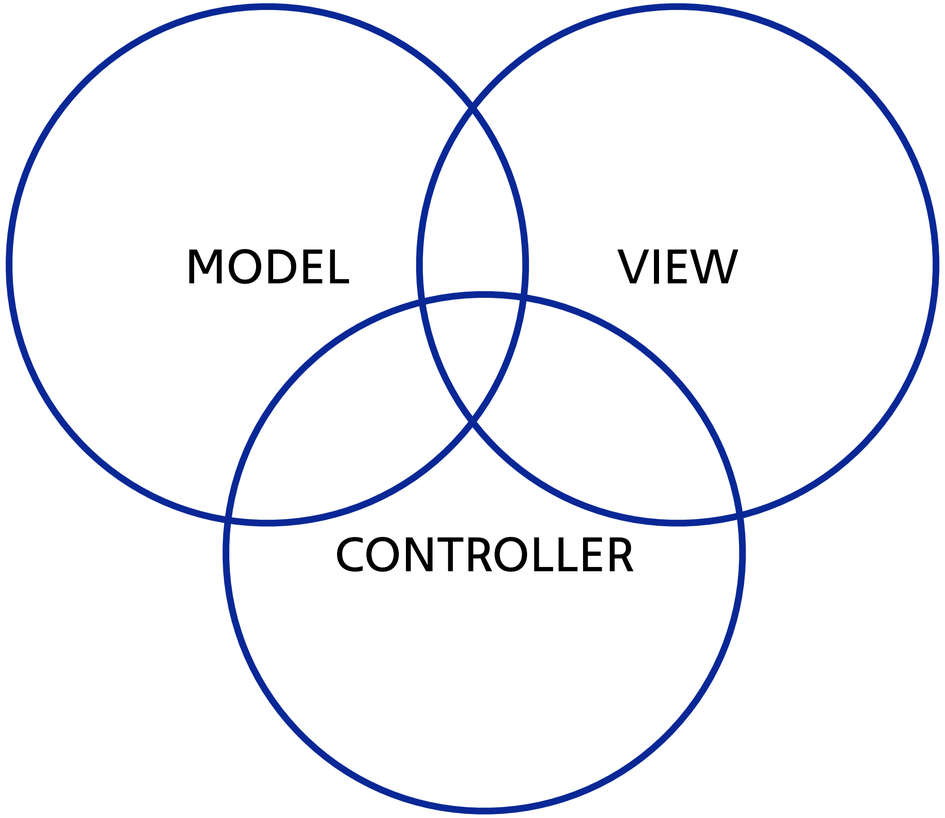
MVC превратился в Massive view controller

К чему это привело

Высокая связность кода

Запутанность логики

Много багов



Долгая разработка новых функций

Повторные регрессы

Нерегулярные релизы

В качестве примера

```
class ViewController: UIViewController {
    // some code

    func makeSomeLogic() {
        let newUIElement = getUIElement()
        self.view.addSubview(newUIElement)

        let networkService = ServiceForThisCase()
        networkService.makeRequest { result in
            let dataBaseService = DataBaseService()
            dataBaseService.updateEntities(with: result)

            self.updateUIElements(with: result)
        }
    }

    // some code
}
```

Почему решили менять архитектуру

Сложность поддержки

Существующий код запутанный
Много legacy

Исправление старого кода занимало много времени

Редизайн

Редизайн также включал в себя создание дизайн-системы

Большая часть UI перерисовывалась

Появлялась связность UI

Много новых функций

Новые функции были совсем новые

Как выбирали новую архитектуру

Есть опыт работы

Вызывают положительный отклик у команды

Позволяют быстро изменять UI

MVVM+R

Подготовка

Network

Выделили отдельный слой для работы с сетью

DataBase

Выделили отдельный слой для работы с базой данных

PlainObjects для моделей

Для переиспользования сущностей текущей базы данных

Покрыли слои протоколами

Для последующего DI

Models

Реализовали Model для текущих сущностей и добавили необходимую бизнес-логику

Было

```
class ViewController: UIViewController {
    // some code

    func makeSomeLogic() {
        let newUIElement = getUIElement()
        self.view.addSubview(newUIElement)

        let networkService = ServiceForThisCase()
        networkService.makeRequest { result in
            let dataBaseService = DataBaseService()
            dataBaseService.updateEntities(with: result)

            self.updateUIElements(with: result)
        }
    }

    // some code
}
```

Стало

```
class ViewController: UIViewController {

    private let networkService: NetworkProtocol = ServiceForThisCase()
    private let dataBaseService: DataBaseProtocol = DataBaseService()

    // some code

    func makeSomeLogic() {
        let newUIElement = getUIElement()
        self.view.addSubview(newUIElement)

        networkService.makeRequest { result in
            self.dataBaseService.updateEntities(with: result)

            self.updateUIElements(with: result)
        }
    }

    // some code
}
```

Кровь, пот и слёзы

Правила

Старая реализация View удаляется сразу

Во всех местах её использования делается заглушка

Новая View оценивается в качестве кандидата на переиспользование

Создаются необходимые ViewModels

Реализуется логика взаимодействия

Тестируются затронутые области

Обновление отправляется в AppStore

Каждый экран переезжает отдельно

Создаем роутер для экрана

Выделяем Views

Реализуем ViewModel

Прикидываем биндинги

Тестируем

Минусы

- Несколько раз crash-free «проседало» (с 98% до 93%)
- «Маленькие» экраны дольше реализовывались
- Bus factor стал «плавающим»
- 4 планируемых месяца превратились в 7

Плюсы

- Связность уменьшилась
- Проще переиспользовать UI
- Слои Network и DataBase инкапсулировались
- Логика работы приложения стала понятна
- Сократилось время внедрения новых функций с 8 до 4,5 дней
- Crash-free стабилизировался около 98,5%
- Количество регрессов сократилось с 3 до 1 плюс смоук-тест

Бонус: план продажи бизнесу смену архитектуры

Сделали анализ времени ключевых показателей:

- внедрение новых функций
- починки багов
- тестирования

Составили план и оценили время переезда

Примерно оценили уменьшение времени на разработку новых и изменение текущих функций

Сделали красивую презентацию

Много разговаривали

Иван Будников

IT-лид

Telegram: @ivan.budnikov