

TinyMVVM

Легковесный MVVM для iOS приложений

Алексей Артамонов

Senior iOS Developer, EPAM

10 лет в разработке мобильных
приложений
6 лет в разработке iOS приложений



С чего все началось?

Задача:

**Разработать iOS приложение в
соответствии с требованиями заказчика, с
возможностью покрытия юнит тестами**

Что имеем?

- Команда 3 iOS разработчика
- Требования и макеты приложения
- Довольно сжатые сроки для выпуска приложения
- Чистый лист (мы писали приложение с нуля)
- Свобода выбора архитектурных подходов
(Доверяющий заказчик)

Мы приступили к проектированию



Что такое архитектура?



Архитектура – это ...

... совокупность важнейших решений об организации программной системы

Какую архитектуру выбрать?

MVVM

MVP

MVC

VIPER

Другая

Лучшая архитектура - это?

А она существует?



Лучшая архитектура - это наиболее подходящая архитектура для выполнения поставленных на проекте целей

Как выбрать архитектуру?

Необходимо учесть



- Размер и сложность приложения
- Сроки разработки
- Требования к тестируемости
- Размер и зрелость команды
- Личные пристрастия архитектора

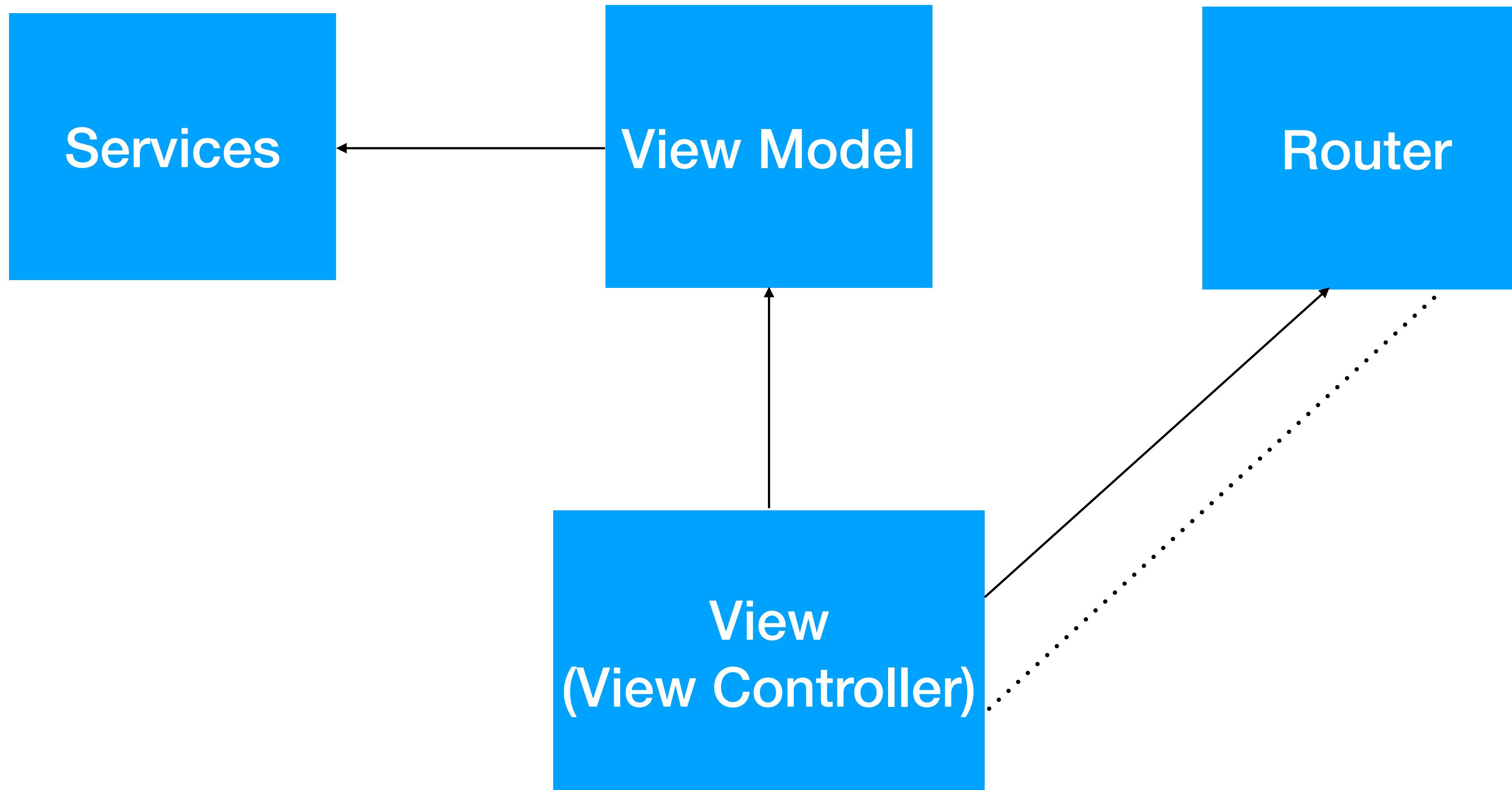
**Почему не подходят
существующие решения?**

Какие цели поставили мы?

Цели TinyMvvm

- Тестируемая бизнес-логика
- Параллельная разработка компонент
- Поддержка Storyboards
- Минимум внешних зависимостей

Компоненты TinyMVVM



Service

- Он же Model
- Протокол + реализация
- Используется ТОЛЬКО во View Model
- Передается во View Model в инициализаторе

Router

- Он же Coordinator
- Протокол + реализация
- Отвечает за переходы между экранами
- Знает многое о View
- ... возможно, даже слишком

BaseRouter

```
import UIKit

public class BaseRouter {
    weak var viewController: BaseViewController?

    init(with viewController: BaseViewController) {
        self.viewController = viewController
        viewController.segueHandler = { [weak self] (segue, sender) in
            self?.prepare(for: segue, sender: sender)
        }
    }

    func prepare(for segue: UIStoryboardSegue, sender: Any?) {
        // no-op
    }
}
```

View

- View и View Controller
- Владеет Router'ом и View Model
- Отображает данные из View Model
- Сообщает действия во View Model
- «Просит» Router перейти на другой экран

BaseViewController

```
import UIKit

class BaseViewController: UIViewController {
    public var segueHandler: ((UIStoryboardSegue, Any?) -> Void)?
        override func prepare(for segue: UIStoryboardSegue, sender: Any?) {
            segueHandler?(segue, sender)
        }
}
```

View Model

- Протокол + реализация
- *Избегает* классов UIKit
- Предоставляет данные в удобном виде
- Использует callback-блоки для уведомлений

Как мы проектируем View Model?

Идеальная View Model

- Посмотреть на View
- Выделить все элементы данных на экране
- Выделить все элементы, меняющие экран
- Выделить элементы, которые открывают новые экраны

Делаем биндинги

Можно ли сделать биндинг, не
используя реактивные
фреймворки?

ОТВЕТ:

```
public typealias PropertySubscriptionToken = Int

public class Property<T> {
    public typealias Listener = (T) -> Void
    public var value: T {
        didSet { listeners.values.forEach { $0(value) } }
    }

    private var listeners = [PropertySubscriptionToken: Listener]()

    @discardableResult
    /// method is not thread safe!
    public func addListener(skipCurrent: Bool = false,
                           _ listener: @escaping Listener
                           ) -> PropertySubscriptionToken

    public func removeListener(_ token: PropertySubscriptionToken)
}
```

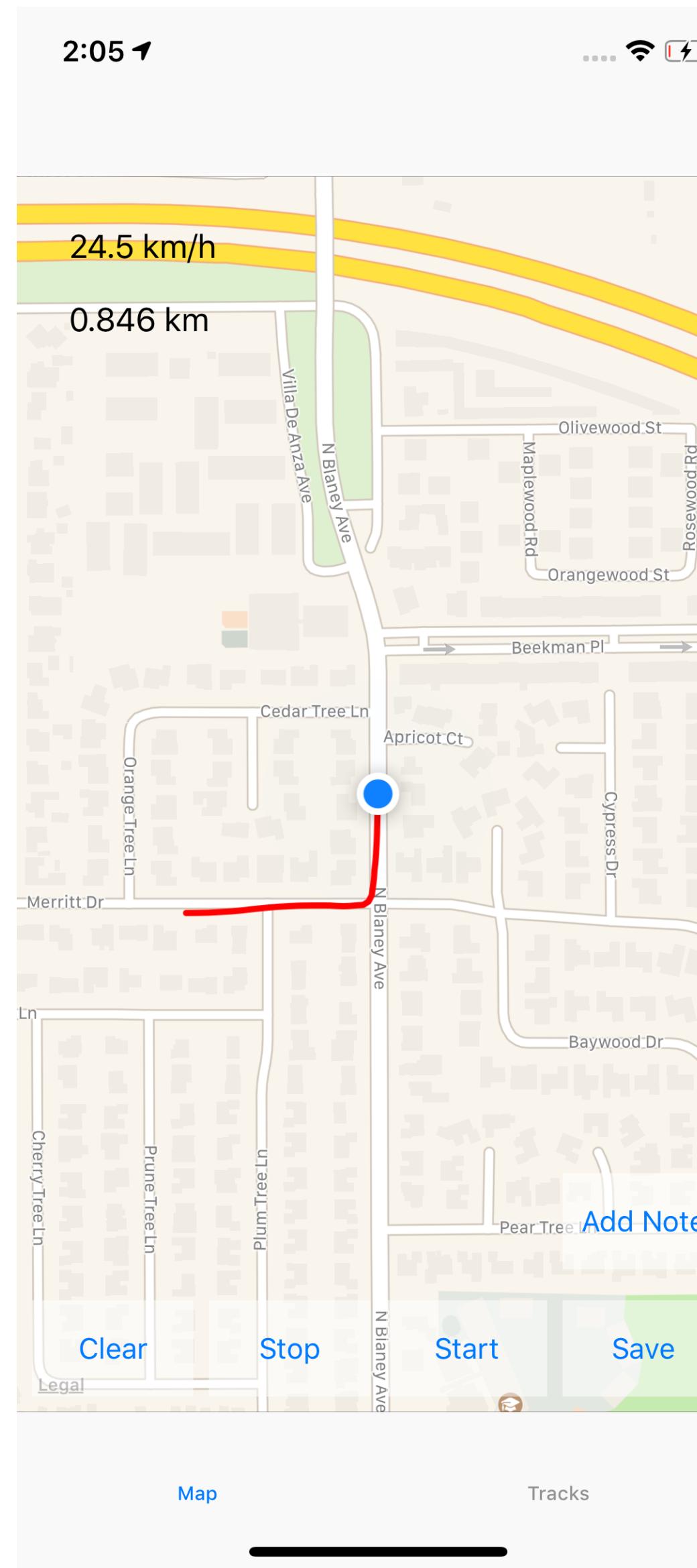
Пример

Элементы данных:

- Лейбл скорости
- Лейбл расстояния
- Trace
- MapView

Элементы изменяющие экран:

- Кнопка Start
- Кнопка Stop
- Кнопка Clear



Элементы, открывающие новые экраны

- Кнопка Save
- Кнопка Add Note

MapViewModel

```
public protocol MapViewModeling: class {
    var regionRadius: Property<Double> { get }
    var locations: Property<[CLLocation]> { get }
    var currentLocation: Property<CLLocation>? { get }
    var speed: Property<String> { get }
    var distanse: Property<String> { get }

    func startTrack()
    func stopTrack()
    func clearTrack()

    func getSaveTrackViewModel() -> SaveTrackViewModeling
}
```

MapViewController

```
class MapViewController: BaseViewController, MKMapViewDelegate {  
    @IBOutlet weak var distanceLabel: UILabel!  
    @IBOutlet weak var speedLabel: UILabel!  
    @IBOutlet weak var addNoteButton: UIButton!  
    @IBOutlet weak var clearButton: UIButton!  
    @IBOutlet weak var stopButton: UIButton!  
    @IBOutlet weak var startButton: UIButton!  
    @IBOutlet weak var saveButton: UIButton!  
    @IBOutlet weak var mapView: MKMapView!  
    private var route: MKOverlay?  
  
    public var router: MapViewRouting?  
    public var viewModel: MapViewModel?  
  
    override public func viewDidLoad() {  
        super.viewDidLoad()  
        if viewModel == nil {  
            viewModel = MapViewModel(locationService: LocationService.shared, dataStorageManager:  
DataStorageManager.sharedInstance)  
        }  
  
        if router == nil {  
            router = MapViewRouter(with: self)  
        }  
    }  
}
```

MapViewController

```
public var viewModel: MapViewModeling? {
    didSet {
        viewModel?.locations.addListener { [weak self] (locations) in
            guard self?.isViewLoaded ?? false else {
                return
            }
            self?.createRoute(with: locations)
        }

        viewModel?.regionRadius.addListener { [weak self] (radius) in
            guard self?.isViewLoaded ?? false else {
                return
            }
            guard let location = self?.mapView.userLocation.location else {
                return
            }
            self?.setMapCenter(with: location, radius: radius)
        }

        viewModel?.currentLocation?.addListener { [weak self] (location) in
            guard self?.isViewLoaded ?? false else {
                return
            }
            self?.setMapCenter(with: location, radius: self?.viewModel?.regionRadius.value ?? 500.0)
        }

        viewModel?.speed.addListener { [weak self] (value) in
            guard self?.isViewLoaded ?? false else {
                return
            }
        }
    }
}
```

MapViewController

```
@IBAction func handleStartButtonTap(_ sender: Any) {  
    viewModel?.startTrack()  
}  
  
@IBAction func handleStopButtonTap(_ sender: Any) {  
    viewModel?.stopTrack()  
}  
  
@IBAction func handleClearButtonTap(_ sender: Any) {  
    if let route = route {  
        mapView.removeOverlay(route)  
    }  
    viewModel?.clearTrack()  
}
```

MapViewRouter

```
public class MapViewRouter: BaseRouter {
    private enum Segue: String {
        case saveTrack
        case addNote
    }

    override func prepare(for segue: UIStoryboardSegue, sender: Any?) {
        guard let identifier = segue.identifier, let typed = Segue(rawValue: identifier) else {
            return
        }
        switch typed {
        case .saveTrack:
            guard let vc = segue.destination as? SaveTrackViewController else {
                fatalError("cannot construct SaveTrackViewController")
            }
            guard let navigation = viewController as? MapViewController else {
                return
            }
            vc.router = SaveTrackRouter(with: vc)
            vc.viewModel = navigation.viewModel?.getSaveTrackViewModel()

        case .addNote:
            //TODO:

        default:
            break
        }
    }
}
```

А как быть с таблицами и экранами нагруженными контролами?

Не получится ли Massive ViewModel?

- Любой экран можно разбить на части отвечающие за определенную логику и сделать View и ViewModel для этих частей
- В таблицах и коллекциях делаем ViewModel для ячеек
- Не обязательно делать всю логику по обработке данных во ViewModel, с некоторыми обязанностями могут справиться сервисы

Немного о юнит тестах

Цели достигнуты:

- Тестируемая бизнес-логика - ViewModel легко тестируются стандартными средствами (использование протоколов сервисов позволяет сделать заглушки для сервисов).
- Параллельная разработка компонент - многие компоненты были разработаны параллельно благодаря минерализации зависимостей модулей друг от друга.
- Поддержка storyboards - Router реализует метод `prepare(for segue: UIStoryboardSegue, sender: Any?)`, поэтому использование Storyboard не вызывает проблем.
- Минимум внешних зависимостей - мы не используем тяжеловесные и избыточные реактивные решения для реализации биндингов.

Сравнение архитектур

	TinyMVVM	MVVM(Reactive/RX)	MVP	VIPER	MVC
Распределение обязанностей в коде	Высокое	Среднее	Средний	Высокий	Низкий
Тестируемость	Высокая	Высокая	Средняя	Высокая	Низкая
Простота использования	Высокая	Низкая	Средняя	Низкая	Высокая
Порог входления	Средний	Высокий	Средний	Высокий	Низкий

Приемущества TinyMVVM

- Простота использования
- Невысокий порог вхождения
- Отличная тестируемость
- Архитектура позволяет хорошо распределить обязанности между компонентами

Спасибо за внимание

Презентацию и пример можно найти по ссылке:

<https://github.com/AlekseiArtamonov/TinyMVVM>

