команда iOS разработки Rambler&Co
КНИГА VIPER
Издание первое

Table of Contents

Предисловие	1.1
Об авторах	1.2
Об издании	1.3
Введение в VIPER	1.4
Структура модуля	1.5
Вопросы кодогенерации	1.6
Составные модули	1.7
Переходы между модулями	1.8
Путь от Massive ViewController до VIPER	1.9
NSFetchedResultsController	1.10
Работа с UIWebView	1.11
Тестирование модуля	1.12
Вопросы Code Style	1.13
Дополнительные материалы Rambler&Co	1.14
Подборка сторонних материалов	1.15

Предисловие

Отдел iOS разработки в Rambler&Co начинался с четырех человек. За первый год отдел вырос в четыре раза. Еще через год нас было уже двадцать пять. Параллельно нами разрабатываются более десяти проектов, причем достаточно разноплановых - почтовое приложение, клиент для блогоплатформы, медийные проекты. Одной из самых важных задач, которые мы должны были решить, была стандартизация подходов к разработке приложений. Решением этого вопроса стал такой архитектурный подход, как VIPER.

VIPER помог нам как в решении административных задач - единообразный подход к архитектуре позволял практически безболезненно переключать людей между разными проектами и легко интегрировать новых сотрудников, так и в вопросе повышения качества наших поектов - высокая модульность позволяла разрабатывать абсолютно независимые друг от друга модули, изменения которых не затрагивали остальную часть кода, а четкая декомпозиция `UIViewController` на слои помогала в вопросе увеличения покрытия кода тестами.

Конечно, внедрение VIPER в наши проекты происходило постепенно. Основная проблема, с которой мы столкнулись, заключалась в отсутствии единой базы знаний и практических примеров. Интернет был полон простых примеров использования этого подхода - экраны авторизации, списки задач. Эти приложения кратко показывали основные принципы взаимосвязи компонентов модуля, но не отвечали на более сложные вопросы: "как переходить от одного модуля к другому", "как реализовывать композицию модулей", "где писать общую бизнес-логику". Любая пустующая ниша когда-нибудь будет заполнена, и конкретно эту задачу мы решили взять на себя. Каждый раз, сталкиваясь с каким-нибудь вопросом по VIPER, мы собирались тесной группой заинтересованных, яростно спорили, исписывали маркерные доски и в итоге приходили к общему решению. Результаты наших споров фиксировались - и в итоге они переросли в серию докладов про использование VIPER и эту книгу.

Книгу условно можно разделить на три части: теоретическая, практическая и приложения. В теоретической части освещена общая структура модуля, как каноническая, так и с нашими дополнениями, история возникновения VIPER, вопросы кодогенерации и автоматизации создания модулей. Практическая часть раскрывает те самые секреты, о которых умалчивают остальные источники информации - переходы между экранами, различные варианты композиции модулей, покрытие кода тестами,

интеграцию VIPER с такими системными компонентами, как `UIWebView` и `NSFetchedResultsController`. А в приложения вошел наш взгляд на Code Style в рамках VIPER-модуля и подборки материалов для дальнейшего ознакомления с вопросом.

Как можно было заметить, во всем предисловии используется местоимение "мы" - и это не случайно. Создание этой книги - плод работы труда не одного человека, а всей нашей команды и большого количество внешних контрибьюторов. Практически все главы написаны разными людьми, которые, тем не менее, имеют общие взгляды на построение архитектуры мобильных приложений. Все идеи и подходы, предложенные нами, прошли проверку на десятках разноплановых приложений и используются командами как в России, так и по всему миру. Именно тот факт, что весь материал прошел проверку боем, является, на мой взгляд, одним из ключевых достоинств этой книги.

Об авторах

В работе над книгой принимало участие большое количество человек, как участников команды iOS разработки Rambler&Co, так и внешних контрибьюторов.

- Толстой Егор - etolstoy
- Крапивенский Сергей - serkrapiv
- Сычев Александр - Brain89
- Смаль Вадим - CognitiveDisson
- Цыганов Станислав - DevAlloy
- Попов Валерий - complexityclass
- Карпушин Артем - ArtemAK
- Сапрыкин Герман - mogol
- Зарембо Андрей - AndreyZarembo
- Мордань Константин - kmordan
- Коровкина Екатерина - k0rka
- Серов Антон - developer-anreal
- Савушкин Максим - maxsava
- garnett
- bimawa
- Samback
- youmee
- x0000ff
- iSevenDays

Визуальное оформление - обложка и схема VIPER-модуля:

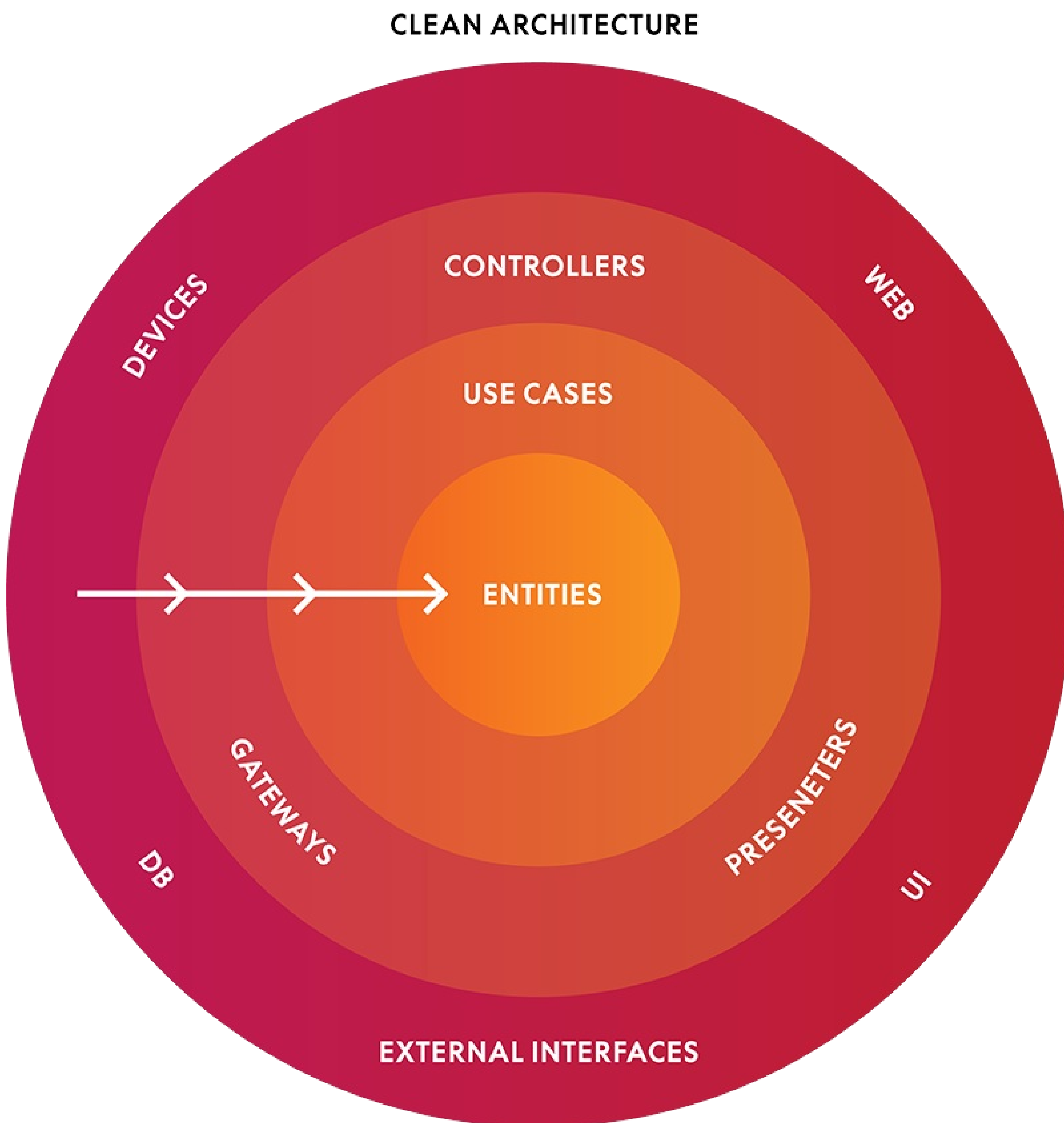
- Студия Рамблер.Инфографика

Об издании

Первое издание книги включает в себя все статьи нашего репозитория на GitHub по состоянию на сентябрь 2016 года. Книга доступна на русском языке, работы по переводу на английский активно ведутся. Если у вас есть вопросы, предложения или замечания - заводите issue на GitHub.

Введение в VIPER

VIPER - это подход к архитектуре мобильных приложений (в частности - iOS), основанный на идеях Роберта Мартина, изложенных им в статье The Clean Architecture.



Основные задачи, которые помогает решить VIPER

- Обеспечение более полного покрытия тестами слоя Presentation, обычно включающего в себя *Massive View Controllers*.

- Разбитие самых крупных классов наших приложений на набор элементов с более-менее четко определенными границами ответственности.

Важно сразу же отметить, что VIPER - это ни в коем случае не набор строгих шаблонов и правил. Скорее это перечень рекомендаций, следуя которым можно построить гибкую и переиспользуемую архитектуру мобильного приложения. Мы, iOS команда Rambler&Co, адаптировали некоторые из каноничных принципов и сформировали определенный набор Best Practices для разработки тех или иных юзкейсов.

Первоначально VIPER может ломать сознание, особенно разработчикам без опыта командной работы над крупными проектами - отсутствует понимание необходимости независимости модулей приложения друг от друга и максимально возможного покрытия их тестами. Тем не менее, весь набор решений оправдывает себя даже для небольших приложений.

Основные достоинства и недостатки VIPER

Плюсы:

- Повышение тестируемости Presentation-слоя приложений.
- Полная независимость модулей друг от друга - это позволяет независимо их разрабатывать и переиспользовать как в одном приложении, так и в нескольких.
- Передача проекта другим разработчикам, либо внедрение нового, дается намного проще, так как общие подходы к архитектуре заренее определены.

Минусы:

- Резкое увеличение количества классов в проекте, сложности при создании нового модуля.
- Некоторые из принципов не ложатся напрямую на UIKit и подходы Apple.
- Отсутствие в открытом доступе набора конкретных рекомендаций, best practices и примеров сложных приложений.

Остальные части нашего руководства в подробностях раскроют каждый из этих пунктов, в том числе расскажут о том, как избавиться от перечисленных недостатков.

Небольшой ликбез по истории вопроса

- **08.2012** - Статья The Clean Architecture от Роберта Мартина.
- **12.2013** - Статья Introduction to VIPER от компании MutualMobile.
- **06.2014** - Выпуск objc.io #13 со статьей Architecting iOS Apps with VIPER от тех же MutualMobile.
- **07.2014** - Выпуск подкаста iPhreaks Show, в котором MutualMobile рассказывают о

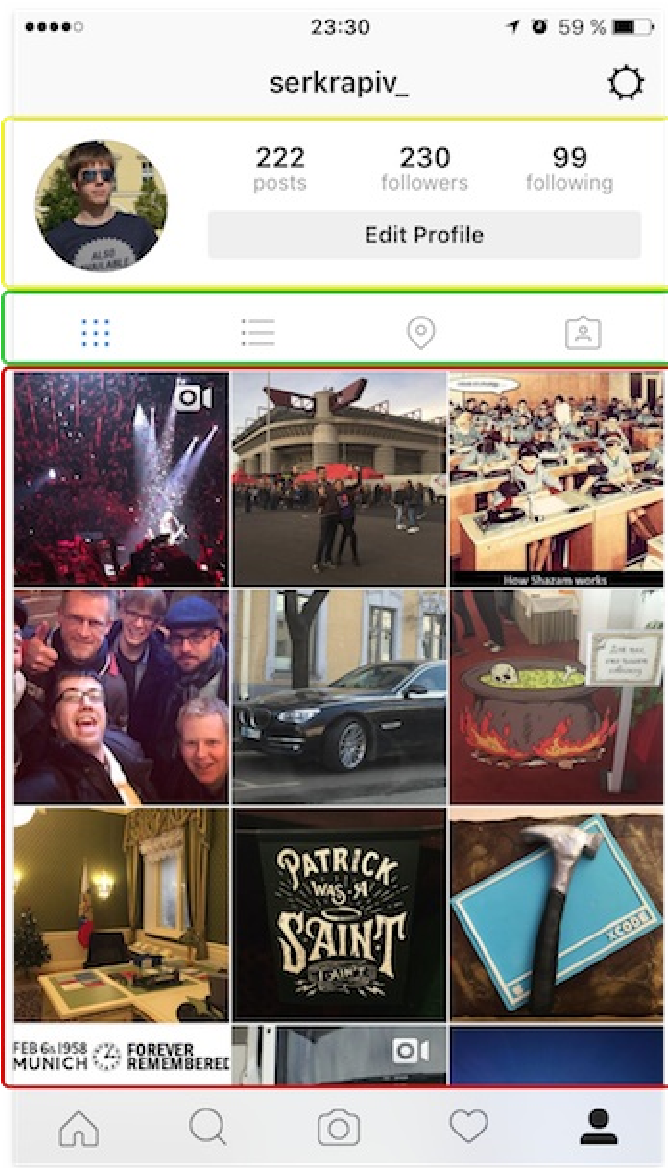
том, как появился VIPER, какие вопросы он решает, и как используется в их приложениях.

- **04.2015** - В рамках локального хакатона в Rambler&Co пишется первое приложение с использованием подходов VIPER.
- **12.2015** - У Rambler&Co больше десяти приложений на VIPER, как разрабатываемых в данный момент, так и выпущенных в AppStore.

Структура модуля

Любой проект можно разбить на несколько логических частей с четко определенной функциональностью. Возьмем, к примеру, Instagram. Основная часть приложения - лента новостей, в которой мы можем просматривать фотографии, ставить лайки, переходить к экрану создания комментария или к отправке сообщений. На другой вкладке находится список лайков, из которого мы можем переходить к конкретной фотографии. Каждый из этих экранов является самодостаточным элементом приложения, выполняющим четко поставленную задачу, и умеющим при необходимости инициировать переход на другие экраны. Такие элементы приложения мы называем модулями.

В простых случаях один экран соответствует одному модулю, но могут быть и сложные экраны, на которых одновременно находятся несколько модулей. Пример из того же Instagram - экран профиля. На нем можно выделить модуль фотографий **(1)**, модуль информации о пользователе **(2)**, вкладки для переключения списка фотографий на другие модули **(3)**.


23:30
59 %
serkrapiv_
222
posts
230
followers
99
following
Edit Profile
How Shazam works
PATRICK WAS A SAINT
FEB 6th 1958 MUNICH
FOREVER REMEMBERED
2
3
1

Структура VIPER-модуля

Для того, чтобы модуль выполнял свое предназначение, нужно решить ряд задач. Требуется реализовать бизнес-логику модуля, работу с сетью, базой данных, отрисовать пользовательский интерфейс. За все это должны отвечать отдельные компоненты, и VIPER описывает роль каждого и способы их взаимодействия между собой. Итак, VIPER-модуль состоит из следующих частей:

View: отвечает за отображение данных на экране и оповещает Presenter о действиях пользователя. Пассивен, сам никогда не запрашивает данные, только получает их от презентера.

Interactor: содержит всю бизнес-логику, необходимую для работы текущего модуля.

Presenter: получает от **View** информацию о действиях пользователя и преображает ее в запросы к **Router'у**, **Interactor'у**, а также получает данные от **Interactor'а**, подготавливает их и отправляет **View** для отображения.

Entity: объекты модели, не содержащие никакой бизнес-логики.

Router: отвечает за навигацию между модулями.

Что мы изменили

Так VIPER выглядит в своем первозданном виде от Mutual Mobile. Мы поработали с этим подходом и вскоре поняли, что в нем есть несколько не слишком удобных моментов:

1. В первоначальной версии VIPER за роутинг отвечает компонент под названием Wireframe. Но при этом он же отвечает за сборку модуля, переход к которому он осуществляет, и проставление всех зависимостей у этого модуля. Это плохо, поскольку нарушает принцип Single Responsibility.

   Мы решили разделить Wireframe на две части. Первая, Router, отвечает только за переходы между модулями. Вторая, Assembly, отвечает за сборку модуля и проставление зависимостей между всеми его компонентами. В наших проектах для этого используется Typhoon, замечательная библиотека для Dependency Injection, благодаря использованию которой вручную Assembly из кода не вызывается.

2. Интеракторы скрывают в себе бизнес-логику. Звучит довольно страшно, ведь за этим кроется много работы, которую стоит разделять между специализированными классами. К тому же часто один и тот же код нужно

переиспользовать в нескольких интеракторах. Нам было нужно общее понимание того, как это делать, чтобы не изобретать свои велосипеды в каждом проекте.

Мы решили ввести дополнительный слой сервисов. Сервис - объект, отвечающий за работу со своим определенным типом модельных объектов. Например, сервис новостей отвечает за получение списка новостей в определенной категории, а так же подробной информации о каждой новости. Сервис авторизации отвечает за, собственно, авторизацию, восстановление пароля, обновление сессии и так далее. У сервисов, в свою очередь, есть зависимости на объекты нижнего уровня, отвечающие за работу с сетью или базой данных.

Сервисы инжектируются в интерактор. В итоге интерактор в основном служит фасадом, взаимодействующим с сервисами и передающим полученные от них данные презентеру. Мы в команде договорились о том, что при работе с Core Data NSManagedObject'ы не выходят на уровни выше интерактора. Поэтому в интеракторах также происходит преобразование `NSManagedObject` в Plain Old NSObject, то есть простой наследник `NSObject`.

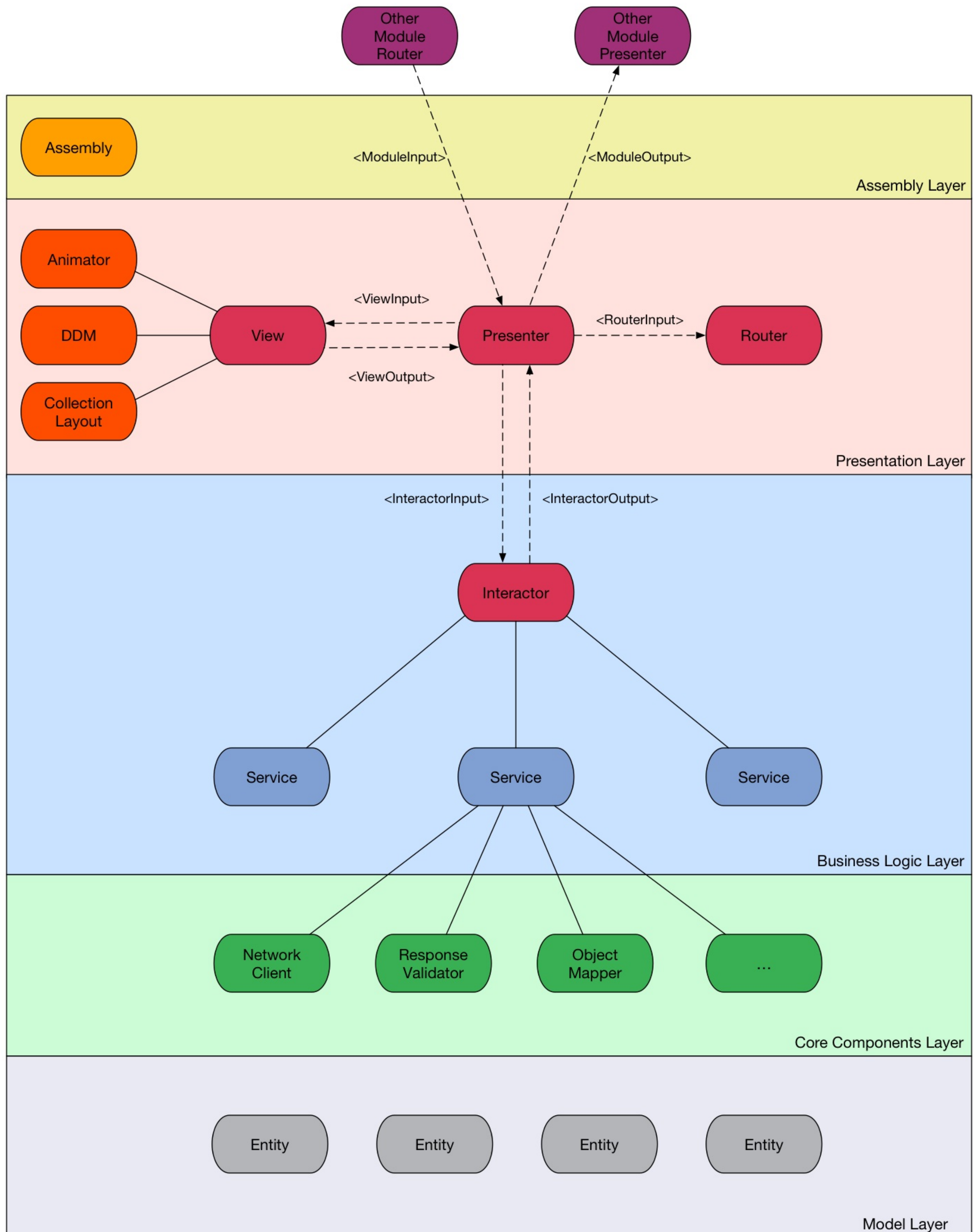
3. В модуле VIPER в качестве View чаще всего выступает `UIViewController`. А в контроллере иногда содержится код, не относящийся напрямую к задачам View. Пример: работа с таблицами и коллекциями. Не зря ведь для работы с этими объектами созданы протоколы, подразумевается, что их реализация должна быть вынесена в отдельный объект. Но в примере от Mutual Mobile код по работе с таблицами содержится прямо в `UIViewController`.

   Нам это не понравилось, и мы решили, что View в общем случае является не одним объектом, а слоем. Помимо контроллера этот слой может содержать дополнительные объекты, которые инжектируются в контроллер и берут на себя часть его работы. Примером такого объекта в наших проектах является DataDisplayManager, реализующий методы `UITableViewDataSource` и `UITableViewDelegate`, или их аналоги для коллекций.

4. Вопрос передачи данных между модулями в оригинальном VIPER не охвачен. Наше решение этой проблемы описано в главе “Переходы между модулями”, подробно на нем останавливаться не будем. Скажем лишь, что у каждого модуля могут быть интерфейсы входа и выхода - протоколы ModuleInput и ModuleOutput. Первый отвечает за передачу входных данных, например идентификатора статьи для экрана, отвечающего за отображение статьи. Второй отвечает за передачу результата работы модуля заинтересованному объекту. Например, при работе с модулем настроек мы можем передать наружу пункт меню, выбранный пользователем.

Итоговая схема модуля

Для иллюстрации всего описанного выше предлагаем ознакомиться с итоговой схемой модуля VIPER. Пугаться не стоит, далеко не каждый модуль должен содержать такое количество объектов. Целью этой схемы было максимально полно отобразить наш подход к архитектуре на примере сложного модуля.



Вопросы кодогенерации

Создание нового модуля - одно из самых узких мест в VIPER, особенно с точки зрения стороннего человека. Для того, чтобы создать новый модуль-экран, нужно как минимум:

- **Пять новых классов** (`Assembly` , `ViewController` , `Presenter` , `Interactor` , `Router`),
- **Пять новых протоколов** (`ViewInput` , `ViewOutput` , `InteractorInput` , `InteractorOutput` , `RouterInput`),
- **Пять новых тестов** (`AssemblyTests` , `ViewControllerTests` , `PresenterTests` , `InteractorTests` , `RouterTests`).

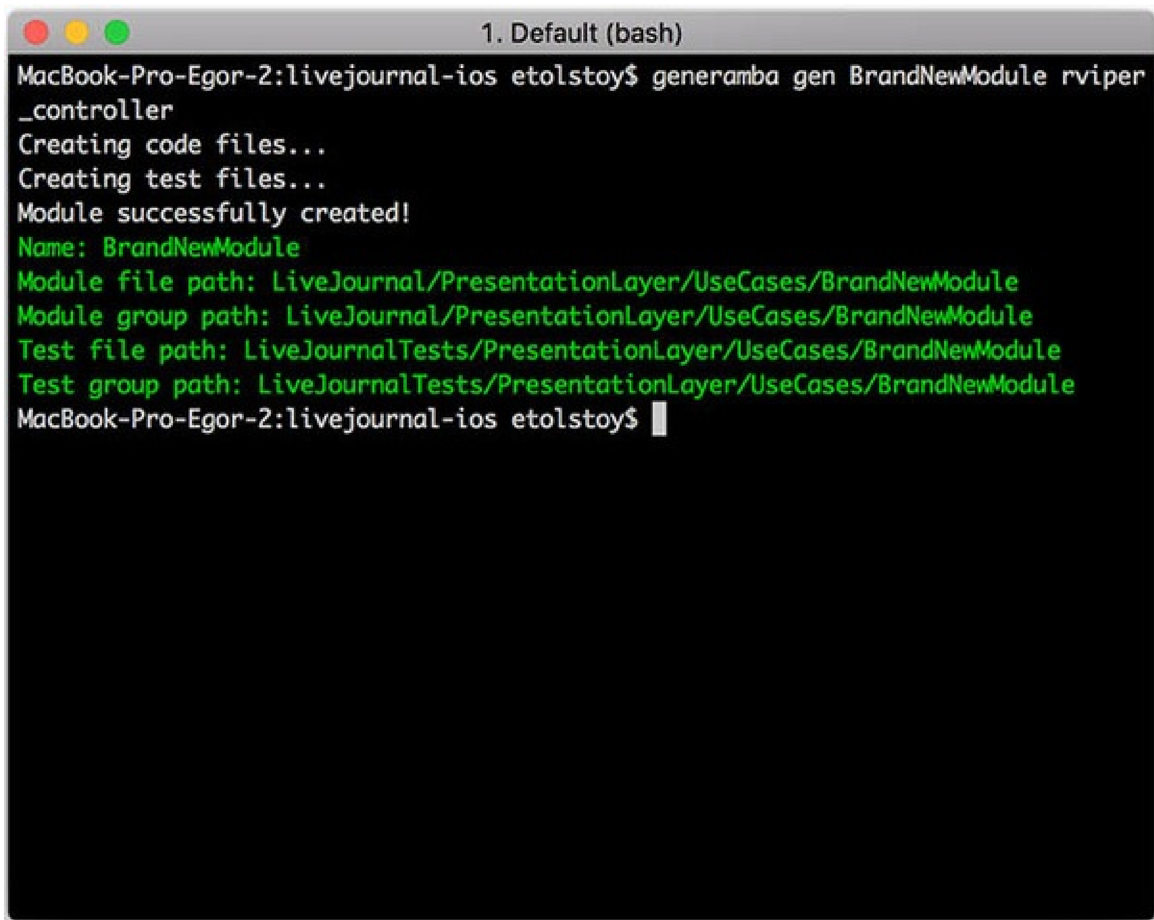
Кроме этого, нужно установить все необходимые связи, добавить реализацию протоколов, настроить dependency injection контейнер - и в частных случаях еще множество других действий. Такая сложность несет за собой две основные проблемы:

- Слишком много времени уходит на простую механическую работу,
- Повышается вероятность опечаток, которые могут повредить не только стилю кода, но и логике работы.

Один из способов решения проблемы, которым долгое время пользовались и мы в Rambler&Co — это создание собственных шаблонов для Xcode. Такой подход решает все обозначенные вопросы, но имеет ряд собственных недостатков:

- Создание нового темплейта - сложный процесс из-за достаточно громоздкого синтаксиса,
- Периодически при обновлении Xcode темплейты могут слететь,
- Нет удобного механизма добавления новых шаблонов в Xcode (Alcatraz не в счет),
- Принципиально отсутствует возможность добавления файлов шаблона в разные таргеты (к примеру, при автогенерации тестов),
- Настройка шаблонов и параметров кодогенерации во многом ориентирована на конкретного пользователя, а не на проект.

Чтобы не быть ограниченными деталями реализации и работы нашей IDE, мы решили вынести процесс кодогенерации на другой уровень и написали небольшую утилиту - Generamba.



Установка

`gem install generamba`

Функциональность

Настройка параметров проекта

Все параметры, нужные для кодогенерации, содержатся в основной папке проекта в файле `Rambafile`. Настраивается он полуавтоматически в результате вызова команды `generamba setup`. В нем находятся такие данные, как название проекта, префикс, компания, пути до папок модулей и тестов, перечисление используемых шаблонов. Этот файл держится в git и используется всеми разработчиками проекта.

Генерация нового модуля

Создание шаблона осуществляется командой `generamba gen ModuleName TemplateName`. В результате будут созданы все файлы, описанные в конкретном шаблоне - причем добавятся в Xcode, так и в файловую систему.

Работа с шаблонами

Все шаблоны, используемые в текущем проекте, описываются в `Rambafile`. При запуске команды `generamba template install` поочередно устанавливается каждый из указанных шаблонов - либо из локальной папки, либо из удаленного git-репозитория, либо из каталогов шаблонов (в том числе и общего). Все шаблоны хранятся в папке `Templates` текущего проекта.

С полным списком команд и их опций можно ознакомиться в нашей wiki.

Проект выложен в open source, поэтому каждый желающий может помочь нам в его развитии, сообщить об ошибках и оставить свои идеи.Кроме того, мы с радостью добавляем новые шаблоны в наш каталог через Pull Request'ы.

Другие кодогенераторы

Помимо Generamba есть и другие кодогенераторы - со своими достоинствами инедостатками.

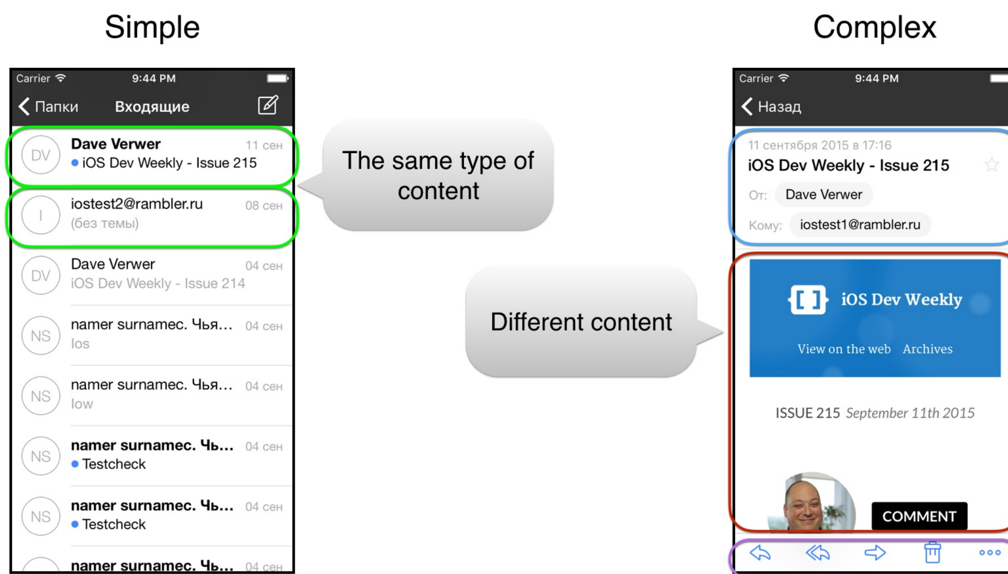
- vipergen
- boa

Составные модули

Простой vs Сложный модуль

Когда мы начинали использовать VIPER в своей работе, использовалась концепция *"Один экран - один модуль"*. Это прекрасно работало, потому что экраны в основном представляли собой простые таблицы. Но с появлением первого "сложного" экрана начались проблемы.

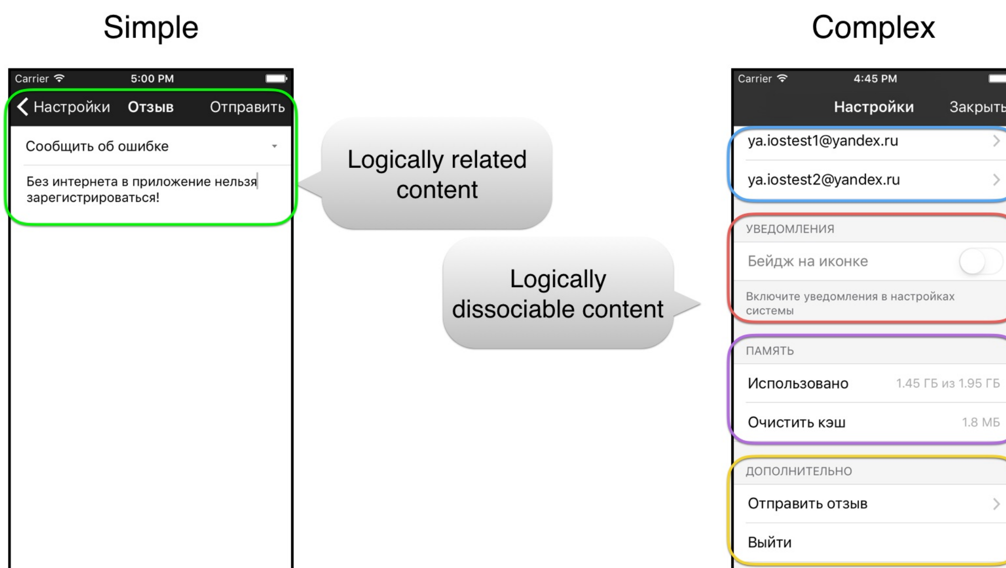
Примеры сложных модулей



Сложные экраны бывают разными. Например, экран настроек управляет множеством не связанных между собой элементов.

Экран просмотра сообщения внутри себя содержит шапку, коллекции для контактов, вложений, а также просмотр письма, для которого необходимо применить специальные преобразования.

Проблемы сложных модулей

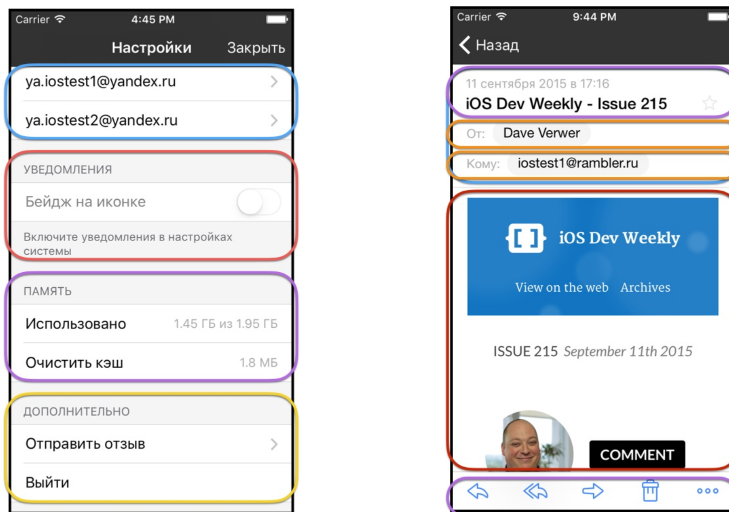


Какие проблемы создают сложные модули?

- Разнородные данные в одном модуле
- Сложная логика работы
- Затруднено тестирование
- Невозможность переиспользования
- Затруднено изменение функций и конфигурации

Например модуль настроек: он должен хранить информацию об имени с подписью, список подключенных ящиков, статус уведомлений. Каждая секция настроек может влиять на своих соседей по секции, но, теоретически, не должна трогать остальные. Хотя такая возможность у неё есть. Переиспользовать такой модуль не получится, все заточено под опции конкретного приложения. Добавление или изменение настроек требует изучения работы всего модуля.

Разбиение на подмодули



Достаточно очевидное разбиение настроек на подмодули - по секциям. Это логично и наглядно. Первый модуль - данные пользователя, второй модуль - список подключенных ящиков и так далее.

С модулем просмотра сообщения все тоже достаточно просто - шапка, контакты, вложения (а они ещё и сворачиваются) и просмотр тела письма.

Плюсы и минусы подмодулей

Плюсы

- Единая ответственность - каждый модуль может, а в идеале даже должен, отвечать за какую-то одну функцию.
- Тестируемость - маленькие модули легче тестировать.
- Переиспользуемость - модуль контактов или вложений может использоваться на экранах написания письма и даже в другом приложении, например мессенджере.
- Для добавления новой функции можно добавить новый подмодуль.
- Возможность создавать разные конфигурации, например добавить дополнительный пункт настроек только для разработчиков.

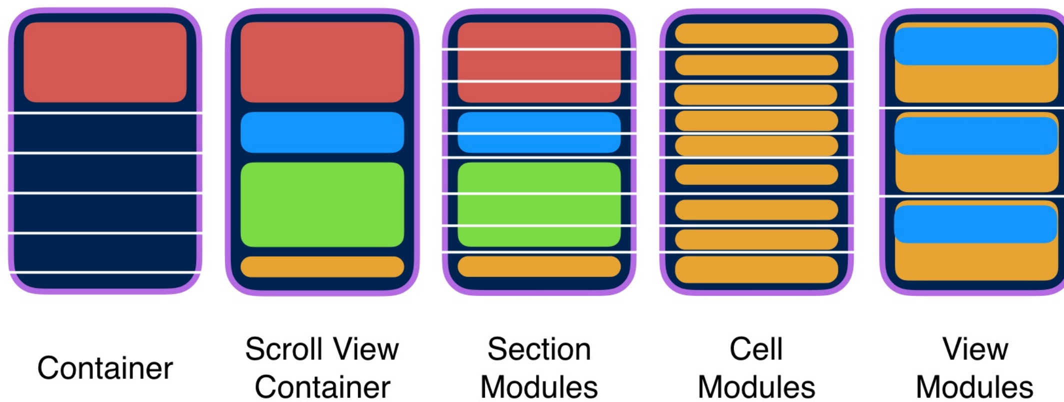
Минусы

- Дополнительный код. Он обеспечивает инициализацию и согласованную работу подмодулей. Его требуется хорошо протестировать.
- Опасность избыточного разделения. Система, разбитая на слишком маленькие модули, рассыпается. К примеру, в настройках не стоит делать модуль для каждого пункта.
- Усложнение потоков данных. Если подмодуль подмодуля должен вернуть какие-то данные в основной модуль, цепочка вызовов будет выглядеть гораздо сложнее, чем в случае монолитного модуля.

Поэтому разбиение на подмодули требует хорошего анализа.

Варианты разделения сложных модулей на подмодули

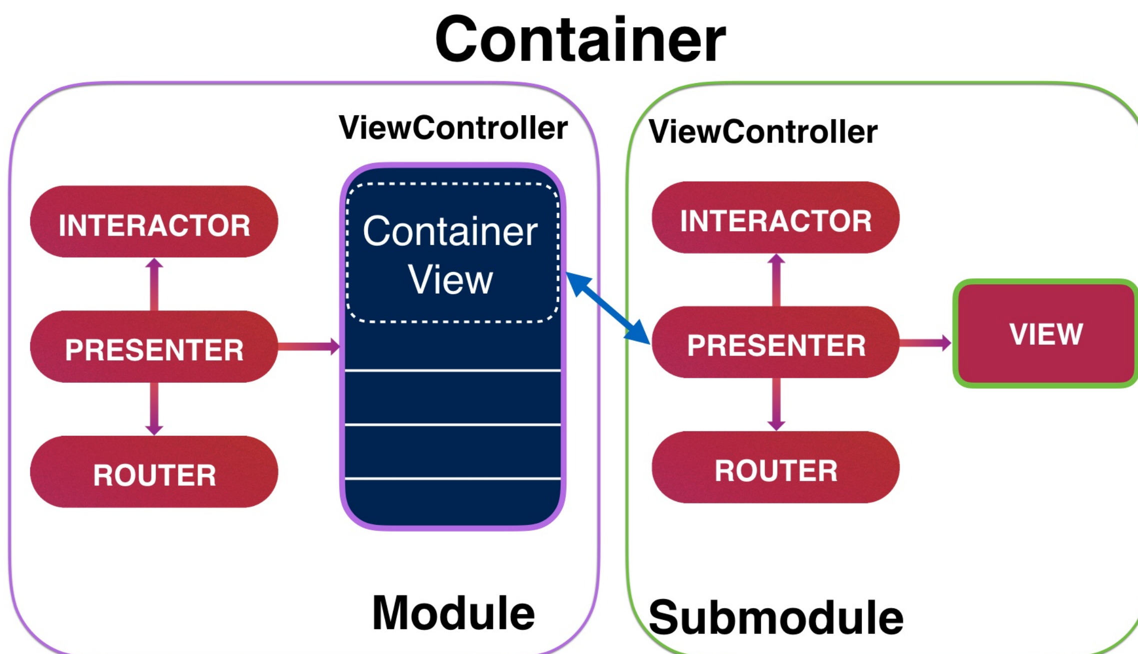
Methods for separation into sub-modules



В работе над проектами были использованы пять вариантов композитных модулей:

- Модуль-контейнер
- Scroll View Controller
- Таблица с группами ячеек
- Таблица с ячейками-модулями
- Модуль-View

Модуль-контейнер



Это аналог Container и EmbedSegue, но управляемый из модуля. Презентер просит роутер добавить дочерний модуль. Роутер инициализирует дочерний модуль, отдает ему данные для работы, добавляет ViewController подмодуля как дочерний для контроллера модуля. И аналогично View контроллера подмодуля добавляется во View-контейнер контроллера модуля.

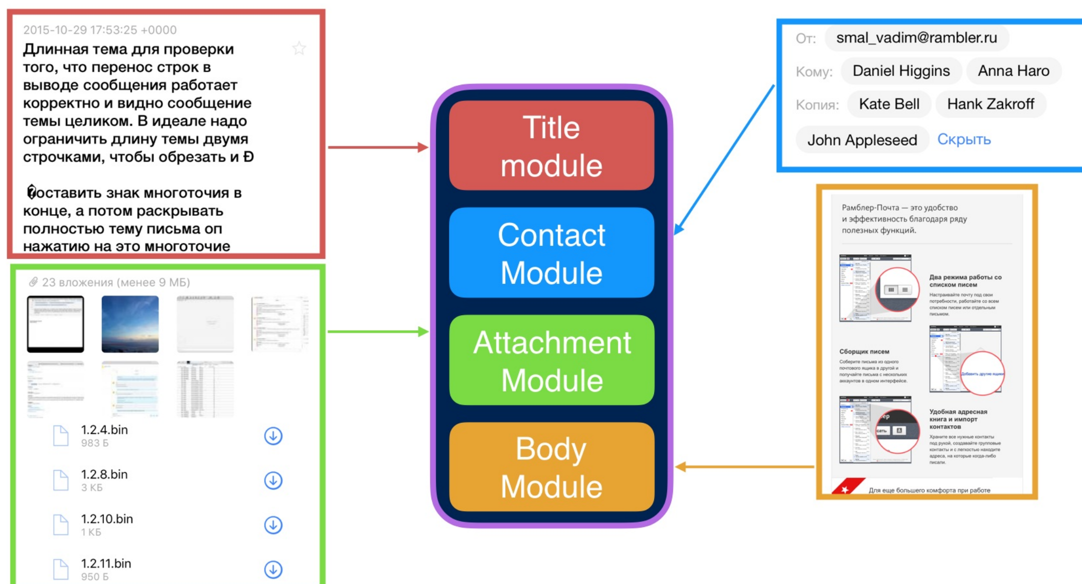
Такой подход хорошо использовать, когда требуется отдельная логика работы внутри модуля, например для таблиц, у которых есть сложная шапка.

Пример

Представьте список постов пользователя, над ними шапка с аватаркой и возможностью написать ему сообщение.

Весь модуль постов занимается только постами, он их загружает, отображает и обрабатывает нажание на ячейку поста с переходом на просмотр поста. Если сюда добавить ещё и загрузку профиля с переходом на написание сообщения, модуль заметно усложнится, поэтому их удобно вынести в отдельный встраиваемый подмодуль. Для работы ему требуется только идентификатор пользователя.

Scroll View Controller

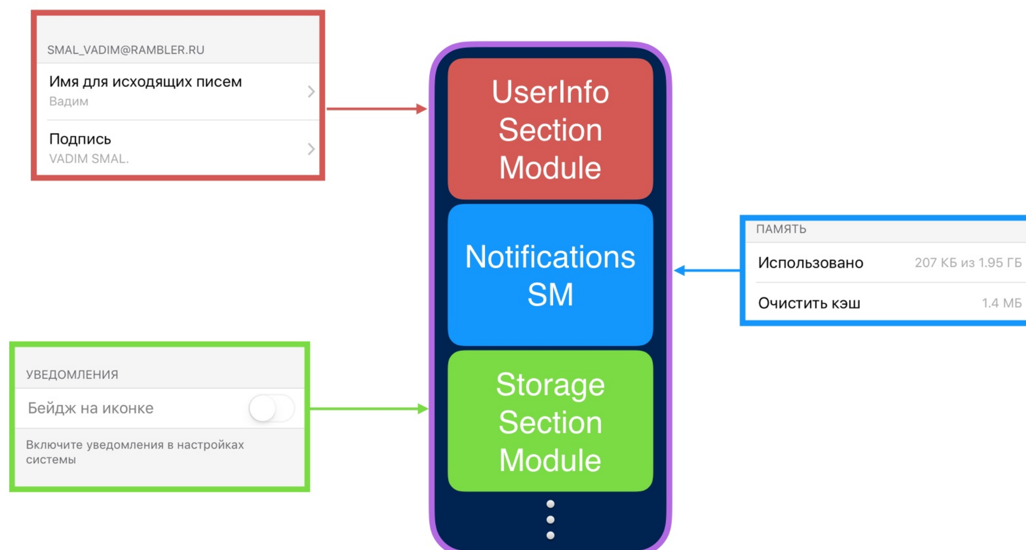


Сложный случай модуля-контейнера. Так реализован экран просмотра сообщений в почтовом клиенте Рамблер/почта. Внутри Scroll View находится несколько контейнеров, каждый из них независимо управляется своим подмодулем. Модуль контактов работает только с загрузкой и отображением контактов, отвечает за сокрытие/разворачивание списка. Модуль вложений разделяет вложения на картинки и документы, отвечает за скрытие и разворачивание списка. Модуль отображения сообщения обрабатывает письма, добавляет переносы в длинные строки, загружает и кеширует inline вложения с картинками.

Причем, как и положено в VIPER, все связанное с загрузкой, обработкой и бизнес-логикой выполяют интеракторы подмодулей. Для этого у них есть ссылки на сервисы. Технически, каждый такой подмодуль можно развернуть на весь экран.

Таблица с группами ячеек

TableView with section modules



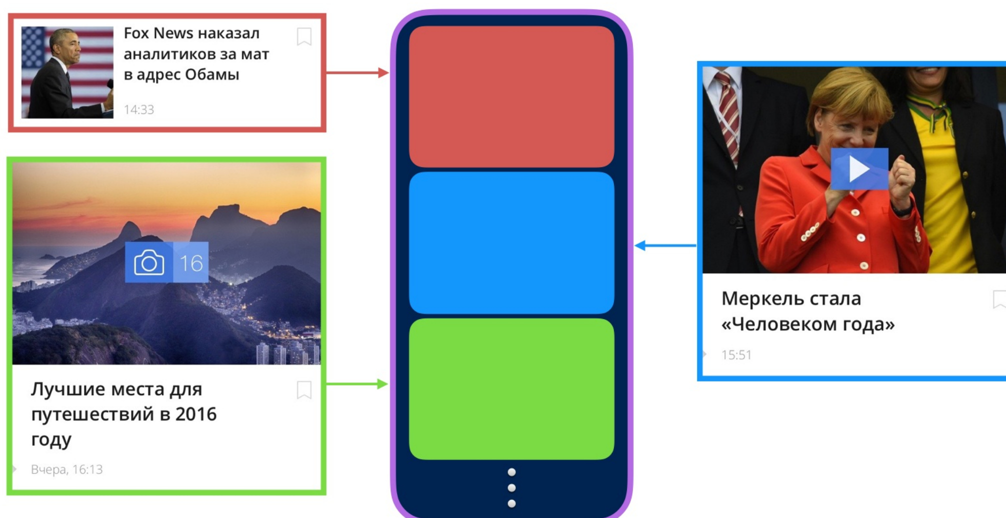
Это способ построения таблицы настроек. Интерактору при инициализации отдается список подмодулей для отображения. Он опрашивает каждый подмодуль и асинхронно получает массив view-model для каждого подмодуля, склеивает их в общий массив и передает своей таблице для отображения. Фабрика ячеек из cell-model получает все необходимые данные для создания и конфигурации ячейки, поэтому универсальна для всех подмодулей.

У подмодулей в качестве View используются фабрики cell-model, они преобразуют данные от presenter в подходящий для отображения в виде ячеек вид, транслируют события из ячеек в presenter, то есть полностью выполняют всю работу View.

Это позволяет модулю уведомлений работать только с уведомлениями, а модулю подключенных ящиков загрузить список ящиков из своего сервиса для отображения.

Таблица с ячейками-модулями

TableView with cell modules

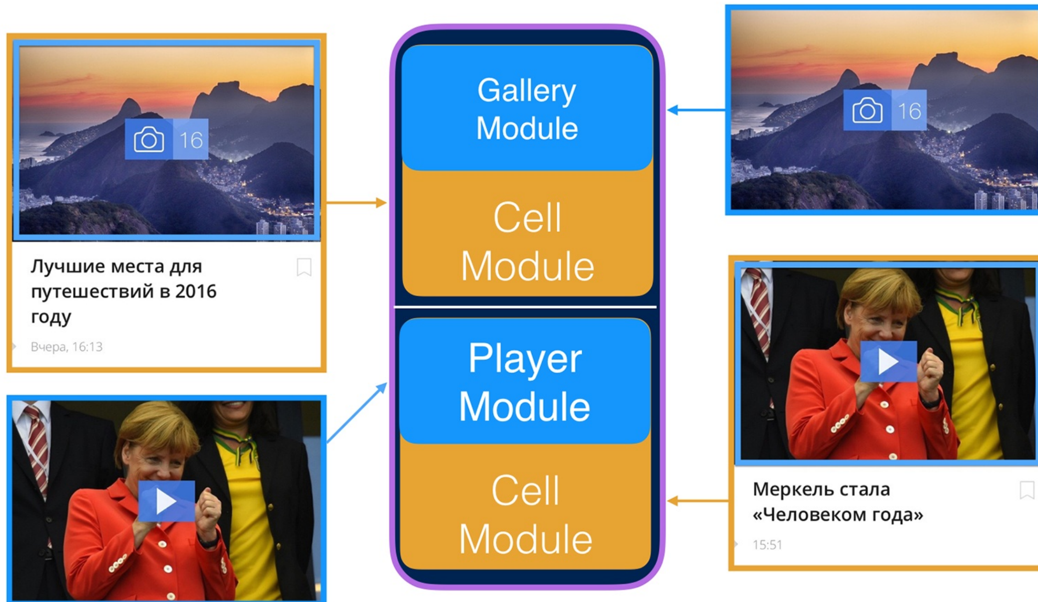


Бывают ситуации, когда отображаемый в таблице контент очень сложный, в ячейке обрабатывается много действий пользователя, ей для работы требуется загружать дополнительные данные или нужно отображать внутри себя CollectionView. В таком случае можно сделать каждую ячейку отдельным модулем.

Сложность в том, что необходимо сделать переиспользуемыми не только ячейки, но и модули VIPER. Фабрика ячеек должна настраивать состояние модуля при отображении ячейки. Например в случае с CollectionView внутри ячейки нужно передать ей не только список объектов для отображения, но и задать соответствующий ContentOffset.

Зато это позволяет обрабатывать все действия с события внутри такого подмодуля, в том числе связываться с сервером.

Модуль View



Самый очевидный способ - модуль View. Такой подход хорошо интегрируется с другими вариантами. Например, у нас может быть модуль ячейки, внутри которой есть подмодуль галереи или видео плеера. Такие подмодули могут быть легко переиспользуемы на других экранах.

Итоги

Когда подмодули помогают?

- Уменьшает сложность основного модуля
- Легкое переиспользование подмодулей
- Упрощает добавление новой функциональности
- Упрощает тестирование

Когда подмодули мешают?

- Значительно увеличивает объем кода
- Усложняет логику
- Усложняет отладку
- Тяжело поддерживать

Переходы между модулями

Подход к созданию модулей через фабрику, описываемый в статьях про канонический VIPER, достаточно неудобен. Мы решили попробовать родные `UIStoryboardSegue` для переходов между модулями. Такой подход открывал заманчивые перспективы - ведь для перехода в другой модуль необходимо было бы всего лишь указать SegueID и передать в модуль данные для работы. Кроме того, для конфигурации модулей мы используем Typhoon, поэтому все модульные ViewController после инициализации через Segue уже имеют связи с другими компонентами модуля.

Через ViewController

Это самый простой вариант. У роутера вызывающего модуля есть ссылка на свой ViewController, при переходе на другой модуль у ViewController вызывается метод `-prepareForSegue:`, где в `sender` передаются данные для следующего модуля. Внутри `-prepareForSegue:` вызывающего ViewController эти данные передаются вследующий модуль.

Такой подход работает, но есть и некоторые недостатки:

- Логика настройки следующего модуля размещается внутри View, а не в Router,
- Нет универсальности и переиспользования, этот метод нужно реализовывать в каждом модуле,
- Данные для работы следующего модуля попадают во View, а не в Presenter,
- Каждый модуль знает об устройстве другого модуля,
- Каждый роутер знает, что работает с классом `UIViewController`, и схема работает только для этого варианта.

Через ViewController с блоком конфигурации

Для решения первых двух проблем были использованы method-swizzling и блоки. В `-prepareForSegue:` в `sender` отправляется блок, в котором выполяется настройка модуля через `destinationViewController`. В альтернативном методе `-prepareForSegue:` блок вызывается с destinationViewController из segue в качестве параметра.

Это работает, логика настройки следующего модуля находится целиком внутри Router, для каждого модуля больше не требуется добавлять во ViewController метод `-prepareForSegue:`, но остаются три проблемы:

- Данные для работы следующего модуля попадают во View, а не в Presenter,

- Каждый модуль знает об устройстве другого модуля,
- Каждый роутер знает, что работает с ViewController и схема работает только для этого.

Много протоколов

Чтобы решить оставшиеся проблемы, были использованы протоколы. Много протоколов. А также swizzling и своя реализация promise. В итоге получилась система передачи данных между модулями без перечисленных недостатков, данные из презентера отдаются роутеру и он конфигурирует ими презентер следующего модуля. Но появились две новые проблемы:

- На освоение у нового разработчика уходило порядка 2х дней,
- Данные передавались только в одну сторону.

Вариант с ModuleInput

Текущий вариант, доступный в нашем GitHub под названием ViperMcFlurry стал гораздо проще в освоении. У каждого модуля теперь есть точка входа - ModuleInput, которая позволяет настроить модуль или вызывать методы. Этот moduleInput можно использовать внутри роутера для настройки модуля, можно вернуть презентеру, для постоянной связи с подмодулем.

У каждого модуля можно задать ModuleOutput, чтобы вернуть данные из модуля. ModuleInput/Output - это протоколы, которые задаются внутри модуля, то есть в них хранится контракт связи с ним. В большинстве модулей в роли ModuleInput выступает презентер этого модуля, а в качестве ModuleOutput - презентер вызывающего модуля.

Embed Segue

Поскольку для метода `-performSegue` требуется только имя перехода, `-prepareForSegue:` подвергся swizzle'ингу, а Typhoon настраивает модуль по ViewController, то мы можем использовать любые классы Segue и механизм переходов между модулями будет работать.

Поэтому для встраивания модулей был создан специальный тип `UIStoryboardSegue` EmbedSegue. Внутри `-performSegue` у SourceViewController вызывается метод, который возвращает View для идентификатора Segue. В эту View и встраивается модуль.

Путь от Massive ViewController до VIPER

Начать новый проект, заложив в его основу VIPER-архитектуру, несложно. Но программисты в своей практике постоянно сталкиваются с задачей поддержки и развития приложений, чья кодовая база изначально разрабатывалась хаотично, без применения жестких правил проектирования. Часто бывает, что требований к первой версии проекта немного, они умещаются на одном тетрадном листе, и, соответственно, при разработке не уделяется должное внимание архитектуре приложения. А техническое задание ко второй версии получается не менее объемным, чем "Война и мир" Толстого, что, естественно, требует кардинально изменить подход к развитию проекта. Поэтому более сложной, но и более интересной задачей является миграция уже в какой-то степени готового iOS-приложения со слабым фундаментом на прочное основание гибкой и надежной архитектуры, которой является VIPER.

Почему MVC становится Massive-View-Controller

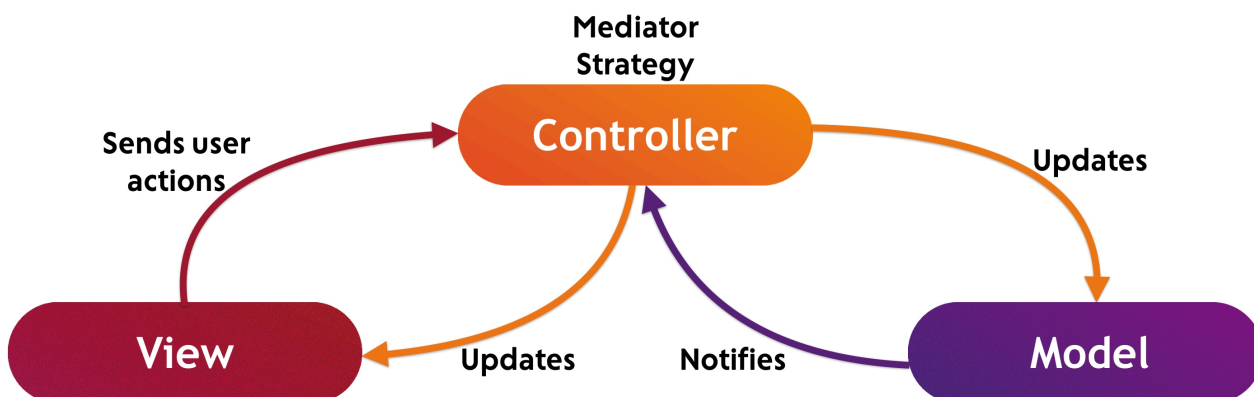
Базовый архитектурный шаблон iOS-приложений, предлагаемый Apple, - это **Model-View-Controller (MVC, "модель-представление-контроллер")**. Обычно этому паттерну явно или неявно следуют все начинающие разработчики. Шаблон Model-View-Controller, что очевидно, - трехслойный. При его использовании все объекты приложения в зависимости от своего назначения принадлежат одному из трех слоев: модели (Model), представлению (View) или управлению (Controller). Каждый архитектурный слой отделен от другого абстрактными границами, через которые осуществляется связь между объектами разных слоев. Основная цель применения этой архитектурной концепции состоит в отделении бизнес-логики (модели) от её визуализации (представления).

Модельный слой описывает данные приложения и определяет логику их обработки и хранения. Объекты модели не должны иметь явной связи с объектами представления; они не содержат информации, как данные можно визуализировать. Примерами объектов этого слоя являются объекты хранения данных, парсеры, сетевые клиенты и т.д.

Слой представления содержит объекты, которые пользователь может увидеть. Чаще всего, объекты этого слоя переиспользуемы. К ним относятся, например, **UILabel** и **UIButton**.

Слой управления выступает посредником при взаимодействии объектов слоя представления и модельных объектов: контролирует ввод данных пользователем и использует модель и представление для реализации необходимой реакции. Также

объекты этого слоя выполняют постановку и согласование задач приложения, управление жизненным циклом других объектов.



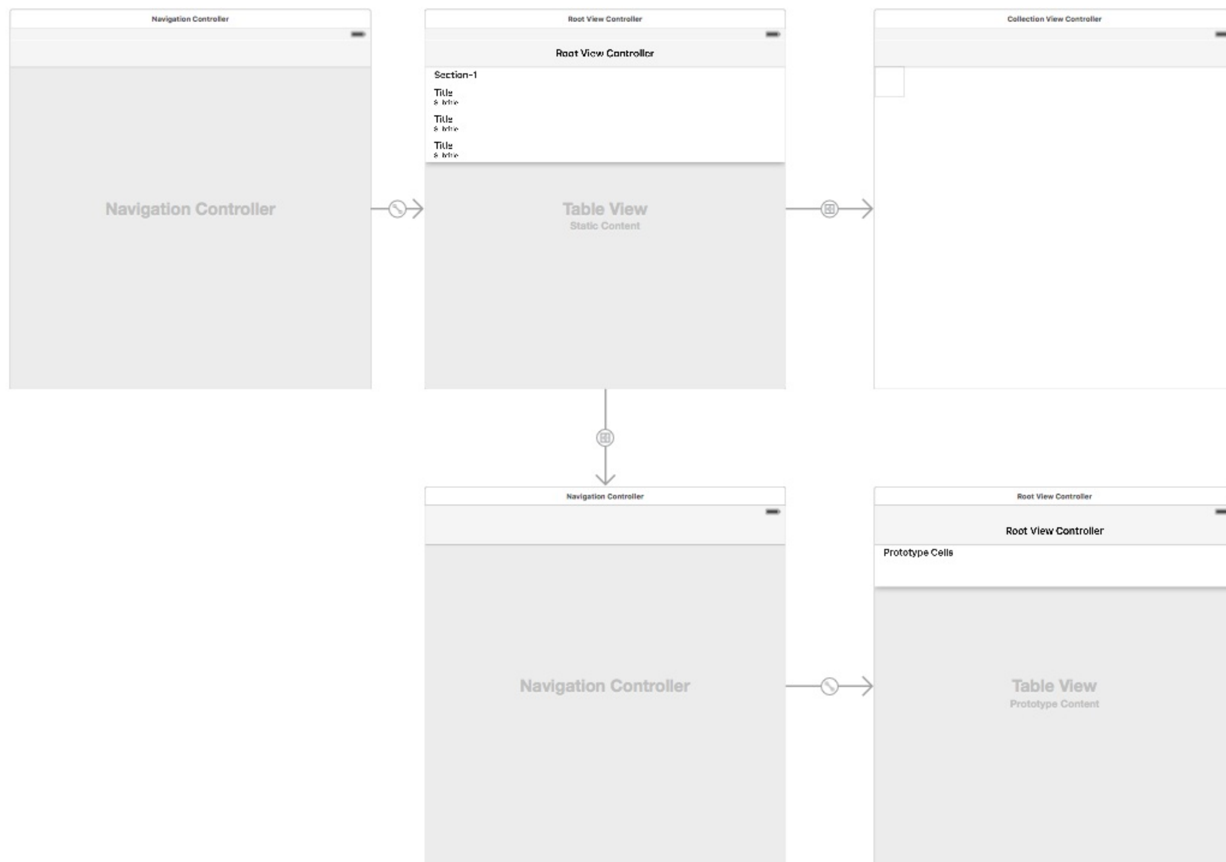
> **Замечание**. Несмотря на то, что в своей документации Apple называет описываемый архитектурный шаблон классическим Model-View-Controller, его правильное название - **Model–View–Adapter (MVA, "модель-представление-адаптер")** или **mediating-controller Model-View-Controller**.
>
> MVA и MVC решают одну и ту же проблему, но используют разный подход. Классический MVC представляется в виде треугольника, где вершинами являются слои: модель, представление и управление, - и разрешает обмен данными между моделью и представлением в обход контроллера. MVA же располагает три слоя на одной линии, исключая прямой обмен данными между моделью и представлением (как на рисунке выше).
>
> В статье здесь и дальше будет использоваться название Model-View-Controller для исключения расхождений с официальной документацией компании Apple.

Согласно рекомендациям Apple, ядром слоя управления в iOS-приложении выступают **контроллеры представления (view controllers)**. Каждый такой объект отвечает за:

- управление иерархией представлений;
- адаптацию размеров представлений к определяемому устройством пространству отображения;
- обновление содержимого представлений в ответ на изменение данных;
- обработку пользовательского ввода и передачу полученных данных в модельный слой;
- освобождение связанных ресурсов при нехватке доступной оперативной памяти.

Рекомендуемым способом описания view controller и связанных с ним view является **Storyboard editor**. Используя этот инструмент, можно не только указать, какие объекты представления какому view controller принадлежат, но также определить иерархические связи и переходы между различными контроллерами.

Рекомендуемым способом описания view controller и связанных с ним view является **Storyboard editor**. Используя этот инструмент, можно не только указать, какие объекты представления какому view controller принадлежат, но также определить иерархические связи и переходы между различными контроллерами.



Apple дает несколько советов по созданию контроллеров представления.

- **Используйте поставляемые со стандартным SDK классы view controllers.**

  В iOS SDK имеется множество контроллеров представления, решающих конкретные задачи: от доступа к списку контактов пользователя до отображения медиаданных. Хорошей практикой является использование в своих приложениях таких, поставляемых системными библиотеками, контроллеров.

- **Создавайте view controller максимально автономным.**

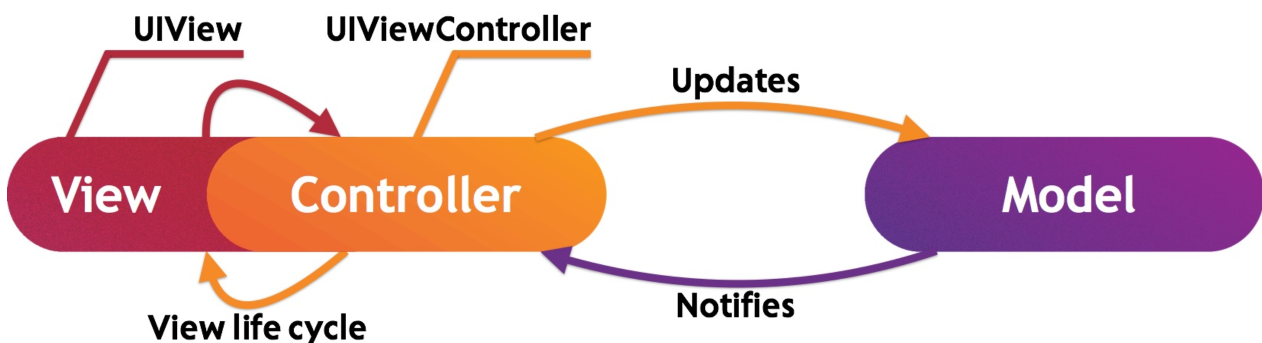
  Контроллер представления не должен знать о внутренней логике другого контроллера или о его иерархии view. Обмен данными между двумя контроллерами должен осуществляться через явно определенный публичный интерфейс.

- **Не храните во view controller данные.** View controller выступает посредником между модельным слоем и слоем представления при обмене данными. Он может кешировать некоторые данные для быстрого доступа, валидировать их, но его

- **Используйте view controller для реакции на внешние события.** К внешним событиям относятся: пользовательский ввод, системные уведомления (например, о появлении клавиатуры), делегатные методы различных обработчиков (например, CLLocationManager).

Приведенные рекомендации и советы, несмотря на свою полезность, слишком общие, описаны без конкретизации и поэтому имеют ряд недостатков. Главный из них - неконтролируемый рост сложности контроллеров представления. Их неразрывная связь с жизненным циклом представлений, являющаяся особенностью iOS SDK, требует формировать реакции на события этого цикла непосредственно во view controller. Реализация обработки внешних событий, пользовательского ввода также происходит во view controller. Помимо этого, модель часто становится слишком пассивной, используется исключительно для доступа к данным, а вся бизнес-логика оказывается вновь во view controller. Отдельно стоит отметить, что Apple практически не рассматривает важные вопросы организации двунаправленной передачи данных между контроллерами, конфигурации созданных контроллеров и т.п., что приводит большинство разработчиков к выводу, что за них опять же ответственен view controller.

В итоге, контроллеры представления становятся центром практически всего, что происходит в приложении и, как следствие, разрастаются до гигантских размеров, превращаясь в то, что называется **Massive-** или **Mega-View-Controller**. Подобный объект является отличным примером последствий необдуманной разработки, игнорирующей базовые принципы проектирования.



Можно выделить следующие недостатки Massive-View-Controller.

- **Высокая сложность поддержки и развития.** Код в Massive-View-Controller сложно модифицировать и расширять из-за его высокой связности и неочевидных потоков данных. Существует риск, что при внесении изменений можно сломать текущую функциональность приложения и до определенного момента не замечать этого.

- **Высокий порог вхождения.** Найти в большом объеме кода нужный метод может оказаться непростой задачей: структура кода неявна и требует время на изучение.

- **Высокий порог вхождения.** Найти в большом объеме кода нужный метод может оказаться непростой задачей: структура кода неявна и требует время на изучение.
- **Код слабо тестируем.** Из-за высокой связности код Massive-View-Controller не покрывается модульными тестами. Так, попытка проверить логику представления приводит к явному вызову методов жизненного цикла, которые неявно могут повлечь за собой загрузку всех связанных view, что нарушает принципы unit-тестирования.
- **Код практически невозможно переиспользовать.** Реализация новой функциональности приложения происходит непосредственно в контроллере представления. Соответственно, переиспользовать этот код без переиспользования всего контроллера не представляется возможным, а рефакторинг и декомпозиция объемного объекта может оказаться нетривиальной задачей.

Чтобы избежать перечисленных недостатков и не загнать в тупик разработку приложения, необходимо строго следовать требованиям хорошего, продуманного архитектурного шаблона. Model-View-Controller, несмотря на слабые места и неочевидность решения некоторых задач проектирования, вследствие простоты использования подойдет для небольших приложений или приложений, требующих высокую скорость разработки. Но для крупных и сложных, с долговременным циклом поддержки и развития лучше использовать более гибкую архитектуру - VIPER.

От плохой реализации к хорошей. Рефакторинг Massive-View-Controller

Столкнувшись с задачей развития приложения, код которого отягчен Massive-View-Controller, прежде всего необходимо улучшить структуру проекта. Как следствие, требуется миграция большого и трудноподдерживаемого Massive-View-Controller во что-то более удобное и гибкое, например, VIPER-модуль. В VIPER, в отличие от MVC, контроллер представления выступает ядром view-слоя, что позволяет не нагружать его дополнительной логикой и локализовать в нем только реализацию задач, непосредственно связанных с визуализацией данных. В этом случае view controller, получая различные внешние сигналы (пользовательский ввод, системные уведомления о нехватке памяти и т.д.), передает их на обработку нижележащему слою - презентеру. Вся остальная невизуальная логика модуля, реализуемая в MVC непосредственно во view controller (например, создание других модулей или передача данных между ними), также инкапуслирована в других слоях.

Примеры рефакторинга

Примеры рефакторинга

```
@protocol AGMainMenuControllerViewOutput <NSObject>
@required
- (void)showMenuSectionWithType:(MainMenuSectionType)sectionType fromViewControlle
r:(UIViewController *)viewController;
@end
```

2. Провести рефакторинг исходного метода из контроллера представления.

```
- (void)tableView:(UITableView *)tableView didSelectRowAtIndexPath:(NSIndexPath *)
indexPath {
   NSInteger index = indexPath.row;
   [self.output showMenuSectionWithType:index fromViewController:self];
}
```

   Теперь контроллер представления передает обработку пользовательского ввода из view-слоя в презентер, не делая ничего лишнего.

3. Расширить протокол входных данных для роутера формируемого VIPER-модуля.

   На каждый из трех исходных переходов следует определить свой метод.

```
@protocol AGMainMenuRouterInput <NSObject>
@required
- (void)showCityFromViewController:(UIViewController *)viewController;
- (void)showPlacesFromViewController:(UIViewController *)viewController;
- (void)showReferenceBookFromViewController:(UIViewController *)viewController;
@end
```

4. Реализовать добавленные методы протокола выходных данных view-слоя в презентере.

   Получив от слоя представления данные пользовательского ввода (т.е. номер выбранной ячейки таблицы), презентер выбирает вариант перехода и передает управление роутеру.

```
@protocol AGMainMenuControllerViewOutput <NSObject>
@required
- (void)showMenuSectionWithType:(MainMenuSectionType)sectionType fromViewControlle
r:(UIViewController *)viewController;
@end
```

2. Провести рефакторинг исходного метода из контроллера представления.

```
- (void)tableView:(UITableView *)tableView didSelectRowAtIndexPath:(NSIndexPath *)
indexPath {
   NSInteger index = indexPath.row;
   [self.output showMenuSectionWithType:index fromViewController:self];
}
```

   Теперь контроллер представления передает обработку пользовательского ввода из view-слоя в презентер, не делая ничего лишнего.

3. Расширить протокол входных данных для роутера формируемого VIPER-модуля.

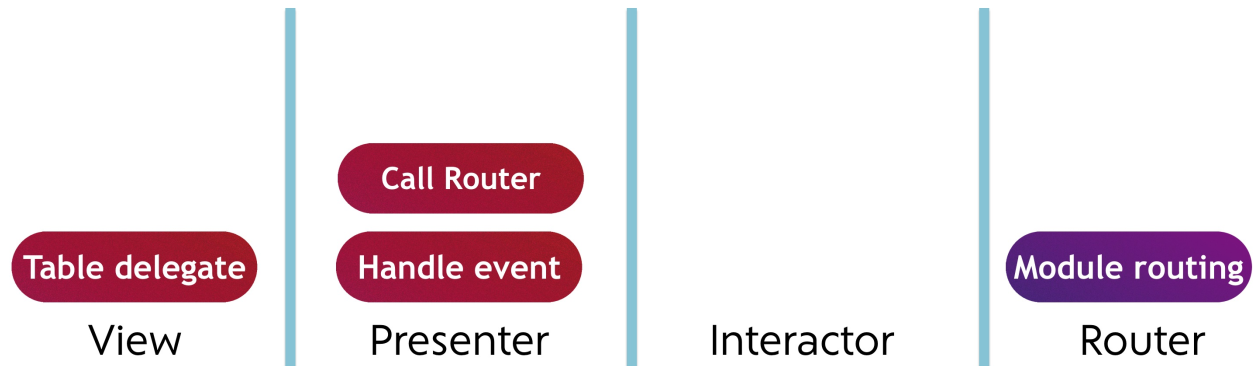
   На каждый из трех исходных переходов следует определить свой метод.

```
@protocol AGMainMenuRouterInput <NSObject>
@required
- (void)showCityFromViewController:(UIViewController *)viewController;
- (void)showPlacesFromViewController:(UIViewController *)viewController;
- (void)showReferenceBookFromViewController:(UIViewController *)viewController;
@end
```

4. Реализовать добавленные методы протокола выходных данных view-слоя в презентере.

   Получив от слоя представления данные пользовательского ввода (т.е. номер выбранной ячейки таблицы), презентер выбирает вариант перехода и передает управление роутеру.

примера, несмотря на свою необходимость, не затронет распределение логики по слоям VIPER-модуля. Результаты распределения логики по слоям VIPER-модуля представлены на рисунке ниже.



Чтение данных

В этом примере в методе `-viewDidLoad` рассматриваемого контроллера представления реализована загрузка списка кварталов города из локальной базы (для загрузки используется библиотека MagicalRecord). При этом после загрузки на обработку передаются managed objects, что может привести к непредсказуемому поведению приложения в дальнейшем.

```
- (void)viewDidLoad {
    [super viewDidLoad];
    NSManagedObjectContext *context = [NSManagedObjectContext MR_defaultContext];
    NSString *sortTerm = NSStringFromSelector(@selector(priority));
    NSArray *quarters = [AGQuarter MR_findAllSortedBy:sortTerm ascending:YES inContext
:context];
    [self handleLoadedQuarters:quarters];
}
```

При рефакторинге следует вынести логику чтения данных из view-слоя. Для этого:

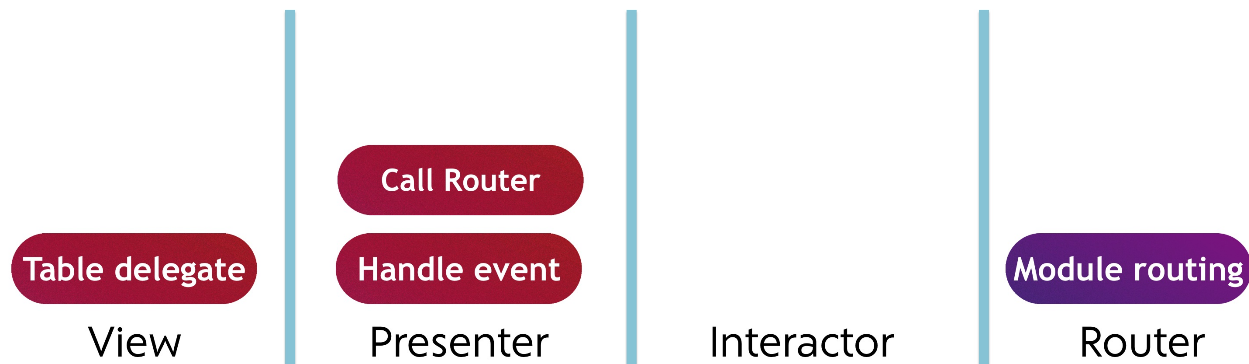
1. Необходимо добавить новый метод к выходным данным view-слоя.

   Это будет **«Получить список кварталов»**.

   ```
   @protocol AGMainMenuControllerViewOutput <NSObject>
   @required
   - (void)obtainQuarters;
   @end
   ```

2. Провести рефакторинг исходного метода из контроллера представления.

примера, несмотря на свою необходимость, не затронет распределение логики по слоям VIPER-модуля. Результаты распределения логики по слоям VIPER-модуля представлены на рисунке ниже.



Чтение данных

В этом примере в методе `-viewDidLoad` рассматриваемого контроллера представления реализована загрузка списка кварталов города из локальной базы (для загрузки используется библиотека MagicalRecord). При этом после загрузки на обработку передаются managed objects, что может привести к непредсказуемому поведению приложения в дальнейшем.

```
- (void)viewDidLoad {
    [super viewDidLoad];
    NSManagedObjectContext *context = [NSManagedObjectContext MR_defaultContext];
    NSString *sortTerm = NSStringFromSelector(@selector(priority));
    NSArray *quarters = [AGQuarter MR_findAllSortedBy:sortTerm ascending:YES inContext
:context];
    [self handleLoadedQuarters:quarters];
}
```

При рефакторинге следует вынести логику чтения данных из view-слоя. Для этого:

1. Необходимо добавить новый метод к выходным данным view-слоя.

   Это будет **«Получить список кварталов»**.

   ```
   @protocol AGMainMenuControllerViewOutput <NSObject>
   @required
   - (void)obtainQuarters;
   @end
   ```

2. Провести рефакторинг исходного метода из контроллера представления.

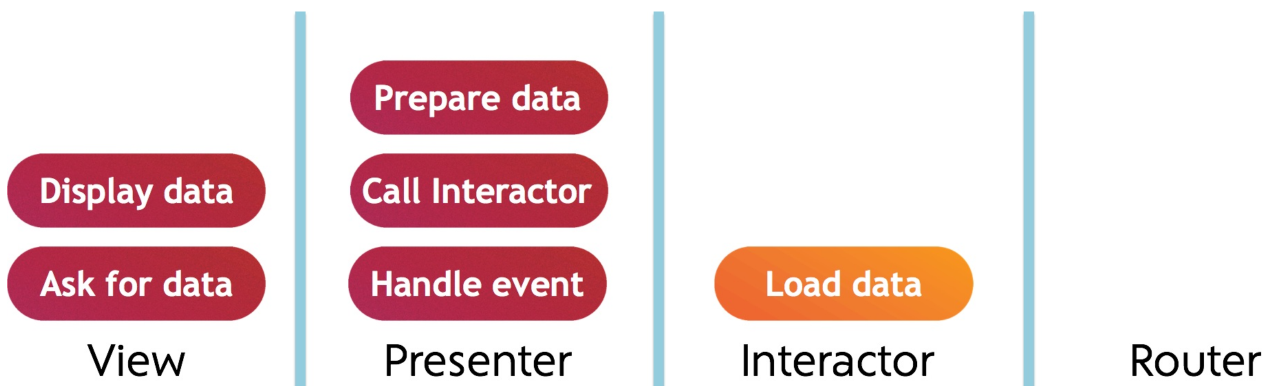
Презентер просит view-слой скрыть индикаторы загрузки и обработать полученные данные.

6. Отобразить полученные данные.

```
- (void)handleObtainedQuarters:(NSArray *)quarters {
   // Displaying quarters
}
```

Непосредственная реализация в рамках примера не рассматривается. В итоге, полученная реализация исправит потенциальную ошибку, связанную с использованием managed-объектов во view-слое, уменьшит связность кода и повысит его тестируемость.

Результаты распределения логики по слоям VIPER-модуля представлены на рисунке ниже.



Конфигурация объектов

В этом примере менеджер локального кеша изображений лениво создается в геттере, определенном непосредственно во view controller.

```
- (id<AGImageLocalCacheManager>)imageLocalCacheManager {
    if (_imageLocalCacheManager == nil) {
        NSFileManager *fileManager = [NSFileManager defaultManager];
        _imageLocalCacheManager = [AGImageLocalCacheManagerImplementation cacheManager
WithFileManager:fileManager];
    }
    return _imageLocalCacheManager;
}
```

Но объект сам не должен создавать требуемые ему вспомогательные объекты - это чревато дублированием кода по настройке зависимостей. Поэтому для инкапсуляции этой логики используются специальные конфигураторы, для реализации которых

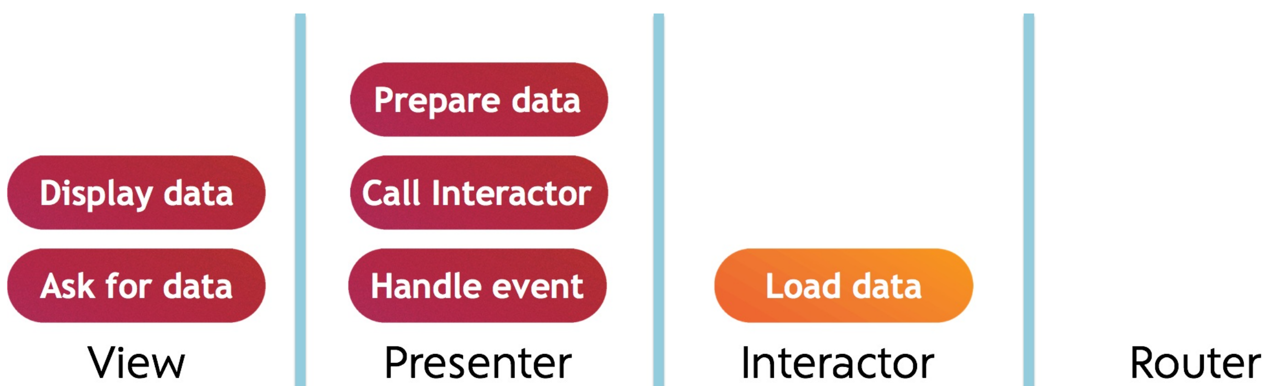
Презентер просит view-слой скрыть индикаторы загрузки и обработать полученные данные.

6. Отобразить полученные данные.

```
- (void)handleObtainedQuarters:(NSArray *)quarters {
   // Displaying quarters
}
```

Непосредственная реализация в рамках примера не рассматривается. В итоге, полученная реализация исправит потенциальную ошибку, связанную с использованием managed-объектов во view-слое, уменьшит связность кода и повысит его тестируемость.

Результаты распределения логики по слоям VIPER-модуля представлены на рисунке ниже.



Конфигурация объектов

В этом примере менеджер локального кеша изображений лениво создается в геттере, определенном непосредственно во view controller.

```
- (id<AGImageLocalCacheManager>)imageLocalCacheManager {
    if (_imageLocalCacheManager == nil) {
        NSFileManager *fileManager = [NSFileManager defaultManager];
        _imageLocalCacheManager = [AGImageLocalCacheManagerImplementation cacheManager
WithFileManager:fileManager];
    }
    return _imageLocalCacheManager;
}
```

Но объект сам не должен создавать требуемые ему вспомогательные объекты - это чревато дублированием кода по настройке зависимостей. Поэтому для инкапсуляции этой логики используются специальные конфигураторы, для реализации которых

В итоге, интерактор сможет использовать менеджер кеша и передавать полученные от него данные вышележащим слоям.

```
- (void)loadBackgroundImage {
   AGPaperGuide *guide = [AGPaperGuide MR_findFirstInContext:[NSManagedObjectConte
xt MR_defaultContext]];
   UIImage *guideBackgroundImage = [self.imageLocalCacheManager loadImageWithImage
Id:guide.backgroundImage.photoId];
   [self.output loadedBackgroundImage:guideBackgroundImage];
}
```

Полученная реализация устраняет потенциальное дублирование при создании вспомогательных объектов и повышает тестируемость кода. Результаты распределения логики по слоям VIPER-модуля представлены на рисунке ниже.

Общие советы

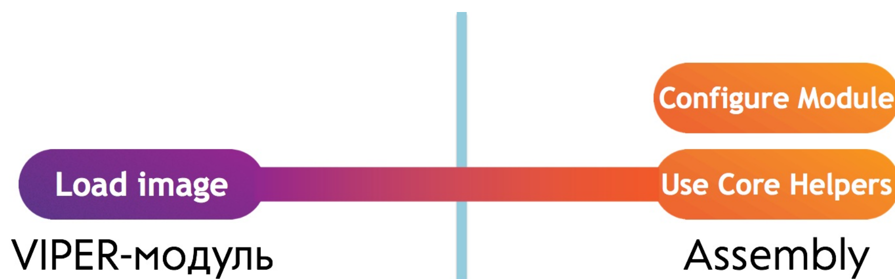
Ниже приведено несколько советов, полезных при переходе от Massive-View-Controller к VIPER-модулю.

- Воспринимайте view в VIPER-модуле не как объект, а как слой, в котором есть множество объектов. Например, хорошим тоном будет создание в этом слое отдельных объектов, реализующих делегаты таблицы и большинство других протоколов. Также стоит использовать вспомогательные сущности для анимаций.
- Классы, зависимые от **UIKit**, **Core Animation** или **Core Graphics** не должны выходить за пределы view-слоя.
- Используйте во view для взаимодействия с данными простые неизменяемые объекты, реализация которых никак не связана с модельным слоем. Это гарантирует, что данные, отданные во view для отображения, уже загружены и могут быть отображены.
- Сложные view, являющиеся частью иерархии view controller, должны сами реализовывать логику своего отображения. Так, кастомный date picker с дополнительными свойствами для визуализации данных может быть определен в отдельном вспомогательном классе.

В итоге, интерактор сможет использовать менеджер кеша и передавать полученные от него данные вышележащим слоям.

```
- (void)loadBackgroundImage {
   AGPaperGuide *guide = [AGPaperGuide MR_findFirstInContext:[NSManagedObjectConte
xt MR_defaultContext]];
   UIImage *guideBackgroundImage = [self.imageLocalCacheManager loadImageWithImage
Id:guide.backgroundImage.photoId];
   [self.output loadedBackgroundImage:guideBackgroundImage];
}
```

Полученная реализация устраняет потенциальное дублирование при создании вспомогательных объектов и повышает тестируемость кода. Результаты распределения логики по слоям VIPER-модуля представлены на рисунке ниже.



Общие советы

Ниже приведено несколько советов, полезных при переходе от Massive-View-Controller к VIPER-модулю.

- Воспринимайте view в VIPER-модуле не как объект, а как слой, в котором есть множество объектов. Например, хорошим тоном будет создание в этом слое отдельных объектов, реализующих делегаты таблицы и большинство других протоколов. Также стоит использовать вспомогательные сущности для анимаций.
- Классы, зависимые от **UIKit**, **Core Animation** или **Core Graphics** не должны выходить за пределы view-слоя.
- Используйте во view для взаимодействия с данными простые неизменяемые объекты, реализация которых никак не связана с модельным слоем. Это гарантирует, что данные, отданные во view для отображения, уже загружены и могут быть отображены.
- Сложные view, являющиеся частью иерархии view controller, должны сами реализовывать логику своего отображения. Так, кастомный date picker с дополнительными свойствами для визуализации данных может быть определен в отдельном вспомогательном классе.

- objc.io. Issue №1. Lighter view controllers.
- Model-View-Controller (MVC) in iOS: A Modern Approach by Rui Peres.

NSFetchedResultsController

Одной из наиболее удобных возможностей, которые нам предоставляет использование CoreData для работы с графом объектов, является `NSFetchedResultsController`. В рамках архитектуры MVC его использование достаточно очевидно - контроллер экрана реализует протокол `NSFetchedResultsControllerDelegate`:

```
- (void)controller:(NSFetchedResultsController *)controller didChangeObject:(id)anObje
ct atIndexPath:(nullable NSIndexPath *)indexPath forChangeType:(NSFetchedResultsChange
Type)type newIndexPath:(nullable NSIndexPath *)newIndexPath;
- (void)controllerWillChangeContent:(NSFetchedResultsController *)controller;
- (void)controllerDidChangeContent:(NSFetchedResultsController *)controller;
```

Эти методы действительно чрезвычайно удобны для того, чтобы прямо в них обновить стейт и вставить/перезагрузить/удалить некоторые из ячеек. За такую простоту, конечно, приходится платить:

- Еще одна ответственность у контроллера,
- Знание о CoreData выходит за пределы модельного/сервисного слоя,
- Мы жестко привязываемся к выбранному механизму уведомлений об изменениях состояния базы,
- *+100 строк* в и без того крупном классе.

Часть из поставленных проблем в некоторой степени решаются путем декомпозиции контроллера на составные объекты, в том числе и на элементы VIPER-стека.

В VIPER, в отличие от MVC, роль и место для `NSFetchedResultsController` не так четко определены. Однозначно не View - во-первых, она не может получать бизнес-сущности от кого-то из своего же слоя, во-вторых - в большинстве случаев стоит стараться вообще не использовать `NSManagedObject`'ы в чистом виде на верхнем уровне архитектуры.

Не подходит и Presenter - его ответственность - связывать между собой элементы модуля и держать стейт. FRC же - это отдельный источник данных, не связанный с интерактором.

Размещать на сервисном слое - тоже неправильно, поскольку сервисы - объекты пассивные, умеющие только реагировать на команды от верхнеуровневых компонентов, и не содержащие в себе никакого дополнительного стейта.

Оптимальный вариант для размещения ответственности FRC - а под ней мы понимаем слежение за состоянием графа объектов, это интерактор:

- Он уже работает с другими бизнес-сущностями,
- В большинстве случаев он знает о том, что мы используем CoreData,
- Он является источником данных текущего модуля - не важно, что послужило причиной их появления - прямой запрос из презентера или получение уведомления от базы.

Мы пришли к двум различным вариантам работы с FRC на уровне интерактора. Первый подходит для таблиц с ограниченным количеством контента. Второй - для infinite scroll'a.

Работа с таблицами с ограниченным количеством контента

Протокол CacheTracker

`CacheTracker` - протокол объекта, задачей которого является получение уведомлений об изменении состояния базы и формировании на их основе набора транзакций.

```
@protocol CacheTracker <NSObject>

/**
Метод настраивает трекер кеша

@param cacheRequest Запрос, описывающий поведение трекера
*/
- (void)setupWithCacheRequest:(CacheRequest *)cacheRequest;

/**
Метод формирует батч транзакций исходя из текущего состояния кеша

@return CacheTransactionBatch
*/
- (CacheTransactionBatch *)obtainTransactionBatchFromCurrentCache;

@end
```

Пример реализации `CacheTracker`

Приведенный вариант реализации построен как раз на работе с `NSFetchedResultsController`. Альтернативная имплементация протокола могла бы, к примеру, быть построенной на работе с `NSNotification`'ами, получаемыми от CoreData.

```
#pragma mark - Публичные методы

- (void)setupWithCacheRequest:(CacheRequest *)cacheRequest {
```

```
    NSManagedObjectContext *defaultContext = [NSManagedObjectContext MR_defaultContext
];
    NSFetchRequest *fetchRequest = [self fetchRequestWithCacheRequest:cacheRequest];
    self.controller = [[NSFetchedResultsController alloc] initWithFetchRequest:fetchRe
quest managedObjectContext:defaultContext sectionNameKeyPath:nil cacheName:nil];
    self.controller.delegate = self;
    [self.controller performFetch:nil];
}

- (NSFetchRequest *)fetchRequestWithCacheRequest:(CacheRequest *)cacheRequest {
    NSFetchRequest *fetchRequest = [NSFetchRequest fetchRequestWithEntityName:[cacheRe
quest.objectClass entityName]];
    [fetchRequest setPredicate:cacheRequest.predicate];
    [fetchRequest setSortDescriptors:cacheRequest.sortDescriptors];
    return fetchRequest;
}

- (CacheTransactionBatch *)obtainTransactionBatchFromCurrentCache {
    CacheTransactionBatch *batch = [CacheTransactionBatch new];
    for (NSUInteger i = 0; i < self.controller.fetchedObjects.count; i++) {
        id object = self.controller.fetchedObjects[i];
        NSIndexPath *indexPath = [self.controller indexPathForObject:object];
        id plainObject = [self.objectsFactory plainNSObjectForObject:object];
        CacheTransaction *transaction = [CacheTransaction transactionWithObject:plainO
bject oldIndexPath:nil updatedIndexPath:indexPath objectType:NSStringFromClass(self.ca
cheRequest.objectClass) changeType:NSFetchedResultsChangeInsert];
        [batch addTransaction:transaction];
    }

    return batch;
}

#pragma mark - Методы NSFetchedResultsControllerDelegate
- (void)controllerWillChangeContent:(NSFetchedResultsController *)controller {
    self.transactionBatch = [CacheTransactionBatch new];
}

- (void)controller:(NSFetchedResultsController *)controller didChangeObject:(NSManaged
Object *)anObject atIndexPath:(NSIndexPath *)indexPath forChangeType:(NSFetchedResults
ChangeType)type newIndexPath:(NSIndexPath *)newIndexPath {
    id plainObject = [self.objectsFactory plainNSObjectForObject:anObject];
    CacheTransaction *transaction = [CacheTransaction transactionWithObject:plainObjec
t oldIndexPath:indexPath updatedIndexPath:newIndexPath objectType:NSStringFromClass(se
lf.cacheRequest.objectClass) changeType:changeType];
    [self.transactionBatch addTransaction:transaction];
}

- (void)controllerDidChangeContent:(NSFetchedResultsController *)controller {
    if ([self.transactionBatch isEmpty]) {
        return;
    }
    [self.delegate didProcessTransactionBatch:self.transactionBatch];
}
```

CacheRequest

`CacheRequest` - объект, содержащий в себе полное описание параметров слежения за состоянием базы, необходимых для `CacheTracker`. По сути, это запрос, на основе которого `CacheTracker` формирует свое поведение.

```
@interface CacheRequest : NSObject

@property (strong, nonatomic, readonly) NSPredicate *predicate;
@property (strong, nonatomic, readonly) NSArray *sortDescriptors;
@property (assign, nonatomic, readonly) Class objectClass;
@property (strong, nonatomic, readonly) NSString *filterValue;

+ (instancetype)requestWithPredicate:(NSPredicate *)predicate sortDescriptors:(NSArray
 *)sortDescriptors objectClass:(Class)objectClass filterValue:(NSString *)filterValue;

@end
```

Классы транзакций

`CacheTransaction` - объект, описывающий изменение одного `NSManagedObject`.

```
@interface CacheTransaction : NSObject

/**
 Измененный объект
 */
@property (strong, nonatomic, readonly) id object;

/**
 IndexPath объекта до его изменения
 */
@property (strong, nonatomic, readonly) NSIndexPath *oldIndexPath;

/**
 IndexPath объекта после его изменения
 */
@property (strong, nonatomic, readonly) NSIndexPath *updatedIndexPath;

/**
 Тип измененного объекта
 */
@property (strong, nonatomic, readonly) NSString *objectType;

/**
 Тип изменения
 */
@property (assign, nonatomic, readonly) NSFetchedResultsChangeType changeType;

+ (instancetype)transactionWithObject:(id)object oldIndexPath:(NSIndexPath *)oldIndexP
ath updatedIndexPath:(NSIndexPath *)updatedIndexPath objectType:(NSString *)objectType
 changeType:(NSUInteger)changeType;

@end
```

`CacheTransactionBatch` объединяет набор транзакций в один объект, предназначенный для передачи между слоями прямо к таблице.

```
@interface CacheTransactionBatch : NSObject

@property (strong, nonatomic, readonly) NSOrderedSet *insertTransactions;
@property (strong, nonatomic, readonly) NSOrderedSet *updateTransactions;
@property (strong, nonatomic, readonly) NSOrderedSet *deleteTransactions;
@property (strong, nonatomic, readonly) NSOrderedSet *moveTransactions;

/**
 Метод добавляет в батч новую транзакцию

 @param transaction Транзакция
 */
- (void)addTransaction:(CacheTransaction *)transaction;

/**
 Метод сообщает, содержит ли батч хоть одну транзакцию

 @return YES/NO
 */
- (BOOL)isEmpty;

@end
```

Пример интерактора

Интерактор не делает практически никакой работы - ему нужно только правильно преднастроить `CacheTracker` и стать его делегатом.

```
@implementation PostListInteractor

- (void)setupCacheTrackingWithCacheRequest:(CacheRequest *)cacheRequest {
    [self.cacheTracker setupWithCacheRequest:cacheRequest];
    CacheTransactionBatch *initialBatch = [self.cacheTracker obtainTransactionBatchFro
mCurrentCache];
    [self.output didProcessCacheTransaction:initialBatch];
}

#pragma mark - Методы протокола CacheTrackerDelegate
- (void)didProcessTransactionBatch:(CacheTransactionBatch *)transactionBatch {
    [self.output didProcessCacheTransaction:transactionBatch];
}

@end
```

Работа с таблицей

И финальная часть схемы - обработка таблицей полученного батча транзакций. В текущем варианте мы работаем с `UITableView` не напрямую, а с помощью фреймворка `Nimbus` - но сути действий это не меняет.

```
- (void)updateDataSourceWithTransactionBatch:(CacheTransactionBatch *)transactionBatch
 {
    for (CacheTransaction *transaction in transactionBatch.insertTransactions) {
        PostListCellObject *cellObject = [self generateCellObjectForPost:transaction.o
bject];
        NSUInteger numberOfObjects = [self.tableViewModel lj_numberOfObjectsInSection:
PostListSectionIndex];
        NSUInteger updatedRow = transaction.updatedIndexPath.row;
        [self.tableViewModel insertObject:cellObject updatedRow inSection:PostListSect
ionIndex];
    }

    for (CacheTransaction *transaction in transactionBatch.updateTransactions) {
        PostListCellObject *cellObject = [self generateCellObjectForPost:transaction.o
bject];
        NSIndexPath *oldIndexPath = [NSIndexPath indexPathForRow:transaction.oldIndexP
ath.row inSection:PostListSectionIndex];
        [self.tableViewModel removeObjectAtIndexPath:oldIndexPath];
        [self.tableViewModel insertObject:cellObject atRow:transaction.updatedIndexPat
h.row inSection:PostListSectionIndex];
    }

    NSMutableArray *removeIndexPaths = [NSMutableArray array];
    for (CacheTransaction *transaction in transactionBatch.deleteTransactions) {
        NSIndexPath *removeIndexPath = [NSIndexPath indexPathForRow:transaction.oldInd
exPath.row inSection:PostListSectionIndex];
        [removeIndexPaths addObject:removeIndexPath];
    }
    [self.tableViewModel lj_removeObjectsAtIndexPaths:[removeIndexPaths copy]];

    for (CacheTransaction *transaction in transactionBatch.moveTransactions) {
        PostListCellObject *cellObject = [self generateCellObjectForPost:transaction.o
bject];
        NSIndexPath *oldIndexPath = [NSIndexPath indexPathForRow:transaction.oldIndexP
ath.row inSection:PostListSectionIndex];
        [self.tableViewModel removeObjectAtIndexPath:oldIndexPath];
        [self.tableViewModel insertObject:cellObject atRow:transaction.updatedIndexPat
h.row inSection:PostListSectionIndex];
    }
}
```

Работа с infinite scroll

При работе с бесконечными лентами держать в памяти все полученные объекты может быть очень накладно. В таком случае лучше использовать модифицированный первый вариант. Изменения следующие:

- `CacheTracker` не добавляет в транзакции модельные объекты, а передает только индексы.

- Таблица умеет запрашивать объект по индексу через цепочку `Presenter -> Interactor -> CacheTracker`.

Работа с `UIWebView`

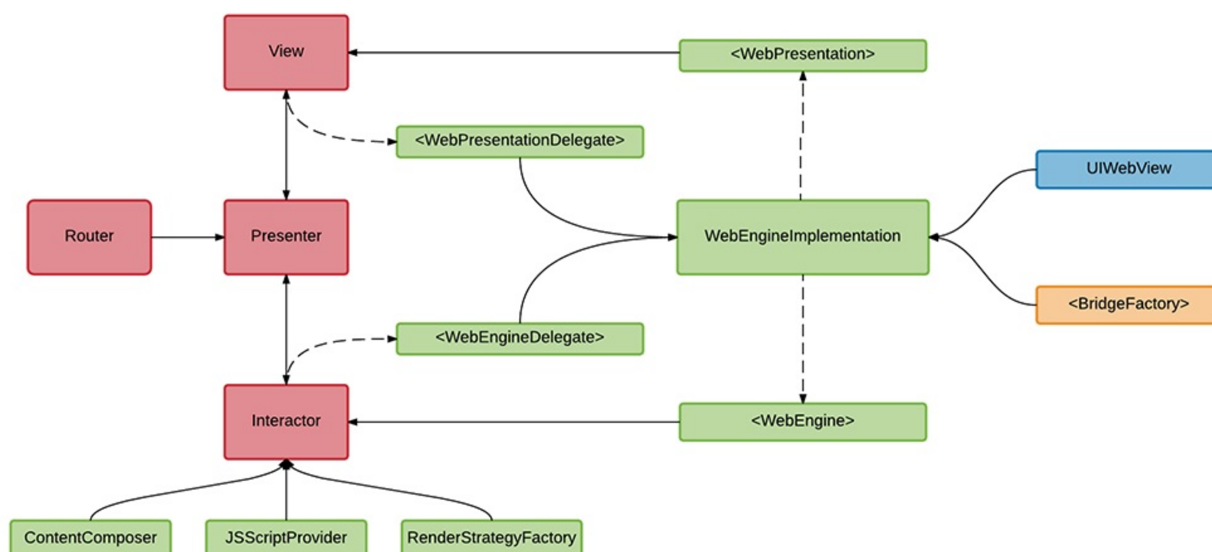
Основная идея - разбить ответственности `UIWebView` на две группы: `WebEngine`, с которым работает Interactor, и `WebPresentation`, с которым работает View.

Ответственности WebEngine

- Загрузка HTML.
- Исполнение JavaScript скриптов.
- Реализация кастомной стратегии рендеринга контента.
- Уведомление Интерактора о событиях окончания отрисовки, загрузки и рендеринга.

Ответственности WebPresentation

- Уведомление View о различных пользовательских действиях: нажатии на ссылки, изображения, видео.
- Предоставление интерфейса для получения информации о `UIWebView` как об объекте отображения, к примеру, размера ее контента.



Логика работы модуля

1. Assembly устанавливает двух делегатов для `WebEngineImplementation` - View и Interactor.
2. Presenter (или View, в случае ячейки) является входной точкой модуля, в которую приходят сырой html.
3. Presenter получает `WebEngine` у View и передает его в Interactor.
4. Presenter передает в Interactor html данные.
5. Interactor настраивает `WebEngine` путем передачи ему определенных скриптов на выполнение.
6. Interactor передает html в `WebEngine`.
7. `WebEngine` сообщает Interactor'у обо всех этапах загрузки данных.
8. Interactor передает эти callback'и в Presenter, который реализует остальную логику.

Каждый из этих шагов очень прост сам по себе - не больше пяти строк кода. Перейдем к рассмотрению реализации каждого из компонентов.

View

```
@interface PostContentCell : UITableViewCell <PostContentViewInput, WebPresentationDel
egate>

@property (weak, nonatomic) IBOutlet UIWebView *contentWebView;
@property (strong, nonatomic) id<PostContentViewOutput> output;
@property (strong, nonatomic) id<WebPresentation> webPresentation;


@end

@implementation PostContentCell

- (BOOL)shouldUpdateCellWithObject:(PostContentCellObject *)object {
    [self.output didTriggerModuleSetupEventWithHtmlContent:object.htmlContent];
    return YES;
}

#pragma mark - PostContentViewInput

- (void)setupInitialState {
    [self.webPresentation setupWithWebView:self.contentWebView];
}

#pragma mark - WebPresentationDelegate

- (void)webPresentation:(id<WebPresentation>)webPresentation didTapLink:(NSURL *)link
{
    [self.output didTriggerLinkTapEventWithURL:link];
}

- (void)webPresentation:(id<WebPresentation>)webPresentation didTapImageWithLink:(NSUR
L *)link {
    [self.output didTriggerImageTapWithURL:link];
}

@end
```

Presenter

```
@implementation PostContentPresenter

#pragma mark - PostContentViewOutput
- (void)didTriggerModuleSetupEventWithHtmlContent:(NSString *)htmlContent {
    self.rawHtmlContent = htmlContent;

    [self.interactor renderContent:self.rawHtmlContent];
}

- (void)didTriggerLinkTapEventWithURL:(NSURL *)url {
    [self.router showBrowserModuleWithURL:url];
}

- (void)didTriggerImageTapWithURL:(NSURL *)url {
    [self.router showFullscreenImageModuleWithImageURL:url];
}

#pragma mark - PostContentInteractorOutput

- (void)didCompleteRenderingContent {
    CGFloat contentHeight = [self.view obtainCurrentContentHeight];
    [self.moduleOutput didUpdatePostContentWithContentHeight:contentHeight];
}

@end
```

Interactor

```
@interface PostContentInteractor : NSObject <PostContentInteractorInput, WebEngineDele
gate>

@property (nonatomic, weak) id<PostContentInteractorOutput> output;
@property (nonatomic, strong) id<WebEngine> webEngine;
@property (nonatomic, strong) ContentHTMLComposer *contentComposer;
@property (nonatomic, strong) JSScriptProvider *scriptProvider;
@property (nonatomic, strong) ContentRenderStrategyFactory *renderStrategyFactory;

@end

@implementation PostContentInteractor

#pragma mark - PostContentInteractorInput

- (void)renderContent:(NSString *)content {
    NSArray *scripts = [self.scriptProvider obtainScriptsForPostEnvironmentSetup];
    for (NSString *script in scripts) {
        [self.webEngine executeScript:script];
    }

    NSString *processedContent = [self.contentComposer composeValidHtmlForRenderingFro
mRawHtml:content];
    [self.webEngine loadHtml:processedContent];
}

#pragma mark - WebEngineDelegate

- (void)didCompleteLoadingContentWithWebEngine:(id<WebEngine>)webEngine {
    ContentRenderStrategy *renderStrategy = [self.renderStrategyFactory postContentRen
derStrategy];
    [self.webEngine renderContentWithStrategy:renderStrategy];
}

- (void)didCompleteRenderingContentWithWebEngine:(id<WebEngine>)webEngine {
    [self.output didCompleteRenderingContent];
}

@end
```

WebEngineImplementation

```
@interface WebEngineImplementation : NSObject <WebPresentation, WebEngine>

@property (weak, nonatomic) id<WebEngineDelegate> webEngineDelegate;
@property (weak, nonatomic) id<WebPresentationDelegate> webPresentationDelegate;
@property (strong, nonatomic) WebBridgeFactory *bridgeFactory;

@end

@implementation WebEngineImplementation

#pragma mark - WebPresentation

- (void)setupWithWebView:(UIWebView *)webView {
    self.webView = webView;
    self.bridge = [self.bridgeFactory obtainBridgeForWebView:webView webViewDelegate:s
elf];
    [self setupJavascriptHandlers];
}

#pragma mark - Handlers Setup

- (void)setupJavascriptHandlers {
    @weakify(self);
    [self.bridge registerHandler:kImageTapHandler handler:^(id data, WVJBResponseCallb
ack responseCallback) {
        @strongify(self);
        NSString *imageURLString = data[kDataLinkKey];
        NSURL *imageURL = [NSURL URLWithString:imageURLString];
        [self.webPresentationDelegate webPresentation:self didTapImageWithLink:imageUR
L];
    }];
}

#pragma mark - WebEngine

- (void)loadHtml:(NSString *)html {
    [self.webView loadHTMLString:html baseURL:nil];
}

- (void)executeScript:(NSString *)script {
    [self.webView stringByEvaluatingJavaScriptFromString:script];
}

- (void)renderContentWithStrategy:(ContentRenderStrategy *)renderStrategy {
    dispatch_group_t group = dispatch_group_create();
    JSResponseBlock responseBlock = ^(NSDictionary *responseData) {
        NSString *logDescription = responseData[JSLogDescriptionKey];
        DDLogVerbose(@"%@", logDescription);
```

```
        dispatch_group_leave(group);
    };

    for (NSString *handlerKey in renderStrategy.scriptNames) {
        dispatch_group_enter(group);
        [self.bridge callHandler:handlerKey data:renderStrategy.scriptPayloads[handler
Key] responseCallback:responseBlock];
    }
    [self.webEngineDelegate didCompleteRenderingContentWithWebEngine:self];
}

#pragma mark - UIWebViewDelegate

- (void)webViewDidFinishLoad:(UIWebView *)webView {
    [self.webEngineDelegate didCompleteLoadingContentWithWebEngine:self];
}

- (BOOL)webView:(UIWebView *)webView shouldStartLoadWithRequest:(NSURLRequest *)reques
t navigationType:(UIWebViewNavigationType)navigationType {
    NSString *urlString = request.URL.absoluteString;
    if (navigationType == UIWebViewNavigationTypeLinkClicked) {
        [self.webPresentationDelegate webPresentation:self didTapLink:request.URL];
        return NO;
    }
    return YES;
}
```

Заключение

Помимо явного разделения ответственностей мы скрыли факт использования сторонней зависимости для связи нативного кода и JavaScript - это всего лишь детали реализации протокола `<WebEngine>`. Unit-тесты для всех компонентов получились очень простыми. Все, что нужно проверить - это правильность data flow.

Тестирование модуля

Кратко о том, что такое TDD

TDD - разработка через тестирование. Предполагается, что в начале пишутся тесты, потом реализация. В случае, если мы закрываем реальные классы протоколами, они идут первыми. После того, как все компоненты написаны - они могут быть интегрированы.

Благодаря такому подходу достигается полное описание ожидаемого поведения класса в тестах. Тесты могут служить в качестве примеров использования класса. Также TDD позволяет нам непредвзято посмотреть на класс с точки зрения пользователя данного класса до его написания.

Более подробно тема TDD в iOS была разобрана в статье Андрея Резанова.

VIPER и тесты

Обычно разделения на основные слои приложения достаточно для того, чтобы довольно неплохо протестировать сервисный и Core слой, но при наличии "толстого" ViewController возникают проблемы в тестировании Presentation слоя. Поэтому многие оставляют его без достаточного покрытия тестами. Давайте разберемся, как VIPER-модули могут помочь нам в покрытии View тестами.

Общим подходом для тестирования является следующий: окружаем объект тестируемого класса протокол-моками зависимостей. Вызываем методы интерфейса/манипулируем свойствами, проверяем вызовы методов моков.

Тестирование View

Существует некоторое количество библиотек, помогающих в тестировании UI слоя, а с недавних пор у нас появились еще и UI-тесты. Достаточно ли этого для полноценного покрытия View? На наш взгляд - нет.

UI-тесты могут служить неплохим способом написания acceptance, или приемочных тестов. В роли черного ящика выступает все боевое приложение, и мы через интерфейс приложения пытаемся получить тот или иной результат.

Проблема с UI-тестами в том, что в случае перегруженного VC весь View является для нас черным ящиком с множеством состояний, и для тестирования необходимо слишком большое количество тестов.

Чем нам тут может помочь VIPER? Из ViewController выносится логика по подготовке и представлению данных в Presenter. Соответственно на View ложится ответственность лишь за обработку и прокидывание событий в Presenter, а также за отображение UI. Отдельно необходимо заострить внимание на том, что View - это не обязательно UI, а класс, через который происходит взаимодействие внешнего мира с модулем. Что это означает для нас? IBOutlet и IBAction необходимо делать публичными. Как итог мы получаем возможность протестировать всевозможные нажатия, заполнение полей и прочее без симуляции нажатий, поиска нужных кнопочек по тексту на них и прочих ненадежных вещей.

Подытожим, выделив два основных вида тестов:

- Взаимодействуем с IBOutlet/вызываем IBAction -> проверяем, что были вызваны соответствующием методы мока Presenter.
- Вызываем методы протокола, через который общается Presenter -> проверяем, что меняются IBOutlet/View.

Отдельно можно выделить тестирование методов жизненного цикла ViewController/View, на которые нам так или иначе необходимо ориентироваться, так как зачастую Presenter не может начинать настройку View до `-viewDidLoad` или `-viewWillAppear`.

Тестирование Router

Методы роутера вызываются, когда необходимо совершить переход из текущего ViewController. Соответственно тестами покрываются методы переходов/закрытия текущего контроллера. Тестируем вызовы всевозможных аниматоров переходов.

Тестирование Interactor

Interactor является связующим звеном в работе с всевозможными сервисами. Именно через него работа идет работа со Storage, в нем создаются PONSO-объекты.

Большинство тестов касаются проверки вызовов одних сервисов в зависимости от ответов других.

Тестирование Presenter

Presenter можно назвать связающим звеном модуля, так как в нем происходит проксирование запросов от одной части модуля к другой. Тесты на подобную передачу составляют львиную долю от тестов Presenter.

Отдельно стоит отметить, что Presenter является входной точкой для модуля, именно в него передаются данные с предыдущего контроллера. На это опять же необходимо писать тесты.

Presenter является местом, где чаще всего находится максимальное число логических ветвлений. Это также необходимо учитывать, и на вход подавать все возможные принципиально отличающиеся друг от друга комбинации значений.

Тестирование Assembly

Assembly настраивает зависимости компонентов модуля. Ее тесты ответственны за то, чтобы проверять, что модуль состоит из правильных частей и все зависимости заполнены.

К счастью, при строгой структуре модуля, данные тесты могут быть созданы автоматически.

Вопросы Code Style

Использование всеми разработчиками команды одного code style не менее важно, чем единая точка зрения на архитектуру приложения и ответственности разных объектов. Наличие соглашений по работе с VIPER-стеком не исключение.

Общие правила для модуля

- Название модуля должно полностью отражать его назначение. Суффикс *Module* в название включать не следует.

  Пример: `MessageFolder`, `PostList`, `CacheSettings`.

- Все элементы модуля разбиты по подпапкам в рамках одной папки модуля.

  Пример:

```
/NewPostUserStory
    /NewPostModule
        /Assembly
        /Interactor
        /Presenter
        /Router
        /View
    /ChooseAvatarModule
        /Assembly
        /Interactor
        /Presenter
        /Router
        /View
```

- Если по итогу написания модуля какие-то из его элементов остались неиспользованными, будь то классы, протоколы или методы, они удаляются.

- Все хелперы располагаются в подпапке своего слоя.

  Пример:

```
/Interactor
    /UserInputValidator
        UserInputValidator.h
        UserInputValidator.m
    /PlainObjectMapper
        PlainObjectMapper.h
        PlainObjectMapperImplementation.h
        PlainObjectMapperImplementation.m
```

- Все методы, с помощью которых слои общаются друг с другом, должны быть синхронными.

  Пример:

```
@interface InteractorInput
- (void)obtainDataFromNetwork;
...

@interface InteractorOutput
- (void)didObtainDataFromNetwork:(NSArray *)data;
...
```

- Все методы протоколов, которыми закрыты элементы модуля, должны начинаться с глаголов - это помогает явно указать на то, что каждый из компонентов обладает поведением, а не состоянием.
- Методы, обозначающие начало действия, должны начинаться с глаголов, выражающих приказ или просьбу (глаголов повелительного наклонения).
- Методы, обозначающие завершение действия или процесса, констатацию факта должны начинаться с глагола прошедшего времени.

  Пример:

```
- (void)obtainImageForPostId:(NSString *)postId;
- (void)processUserInput:(NSString *)userInput;
- (void)invalidateCurrentCache;
```

- В публичных интерфейсах всех классов и протоколов стараемся использовать forward-declaration, используя `#import` лишь при необходимости.

  Пример:

```
#import <Foundation/Foundation.h>

#import "PostListViewOutput.h"
#import "PostListModuleInput.h"
#import "PostListInteractorOutput.h"

@protocol PostListViewInput;
@protocol PostListRouterInput;
@protocol PostListInteractorInput;
@class PostListViewModelMapper;

@interface PostListPresenter : NSObject <PostListModuleInput, PostListViewOutput,
PostListInteractorOutput>

@property (nonatomic, weak) id<PostListViewInput> view;
@property (nonatomic, strong) id<PostListRouterInput> router;
@property (nonatomic, strong) id<PostListInteractorInput> interactor;
@property (nonatomic, strong) PostListViewModelMapper *postListViewModelMapper;

@end
```

Слой Interactor

Класс `Interactor`

Наименование

`<ModuleName>Interactor.h / <ModuleName>Interactor.m`

Дополнительные правила

- Интерактор не держит состояния, только зависимости, расположенные в его открытом интерфейсе.

  Пример:

  ```
  @interface PostListInteractor : NSObject <PostListInteractorInput>

  @property (nonatomic, weak) id<PostListInteractorOutput> output;
  @property (nonatomic, strong) id<AccountService> accountService;

  @end
  ```

- Интерактор держит weak-ссылку на презентер. Переменная называется `output`.

  Пример:

`@property (nonatomic, weak) id<PostListInteractorOutput> output;`

Фасады над сервисами

Наименование

`<Feature>Facade.h / <Feature>Facade.m`

Описание

В случае, если в интеракторах нескольких модулей есть повторяющаяся логика по использованию сервисов определенным образом, она инкапсулируется в отдельный фасад над сервисами. В качестве примера можно привести объект, реализующий логику пагинации разных лент постов - он умеет запрашивать список новых элементов, сранивать их с ранее закешированными, вычислять смещения и дырки - и многое другое. Благодаря выделению этих связей в отдельную сущность, пагинация может быть достаточно легко подключена для любого модуля списка элементов.

Пример: `PagingFacade`.

Протокол `<InteractorInput>`

Наименование

`<ModuleName>InteractorInput.h`

Описание

Содержит методы для общения с интерактором. Этим протоколом закрыт интерактор с точки зрения презентера.

Примеры методов

```
- (NSArray *)obtainNewsFromCache;
- (void)obtainMessageWithId:(NSString *)messageId;
- (void)performLoginWithUsername:(NSString *)username password:(NSString *)password;
```

Общие паттерны методов

- Если в данном модуле нам нужно самостоятельно решать, в какой момент просить данные из кэша, а в какой - из сети, допустимо явно указывать это интерактору (`obtainFromNetwork/-fromCache`).

Протокол `<InteractorOutput>`

Наименование

`<ModuleName>InteractorOutput.h`

Описание

Содержит методы, при помощи которых интерактор общается с вышестоящим слоем модуля. Этим протоколом обычно закрыт презентер.

Примеры методов

```
- (void)didObtainMessage:(Message *)message;
- (void)didPerformLoginWithSuccess;
```

Общие паттерны методов:

- В большинстве случаев в качестве префикса каждого метода используем `did` - это указывает на пассивную роль интерактора, который умеет выполнять ряд действий по запросу и уведомлять об их окончании.- В случае отсутствия единой системы обработки ошибок заводятся пары методов (`didObtainWithSuccess/-withFailure`).

Слой Presenter

Класс `Presenter`

Наименование

`<ModuleName>Presenter.h / <ModuleName>Presenter.m`

Дополнительные правила

- В отличие от всех остальных элементов, презентер обладает состоянием. Оно находится в приватном extension.
- Презентер держит weak-ссылку на view. Переменная называется `view` .

  Пример:

  ```
  @property (nonatomic, weak) id<PostListViewInput> view;
  ```

- Презентер держит strong-ссылку на роутер. Переменная называется `router` .

  Пример:

  ```
  @property (nonatomic, strong) id<PostListRouterInput> router;
  ```

- Презентер держит strong-ссылку на интерактор. Переменная называется `interactor`.

  Пример:

  `@property (nonatomic, strong) id<PostListInteractorInput> interactor;`

- Если презентеру нужно держать объект, реализующий протокол `ModuleInput` дочернего модуля, переменная называется `<OtherModuleName>ModuleInput`.

Класс `State`

Наименование

`<ModuleName>State.h / <ModuleName>State.m`

Описание

В том случае, если текущий модуль обладает каким-либо состоянием, его можно выделить в отдельный объект, не обладающий никаким поведением и выступающий простым хранилищем данных.

Пример:

```
@interface PostListState

@property (nonatomic, assign) FeedType feedType;
@property (nonatomic, assign) BOOL hasHeader;
@property (nonatomic, strong) NSString *feedId;

@end
```

Протокол `<ModuleInput>`

Наименование

`<ModuleName>ModuleInput.h`

Описание

Содержит методы, при помощи которых с модулем могут общаться другие модули или его контейнер.

Дополнительные правила

- При использовании библиотеки ViperMcFlurry наследуется от протокола `<RamblerViperModuleInput>`.

Примеры методов

```
- (void)configureWithPostId:(NSString *)postId;
- (void)updateContentInset:(CGFloat)contentInset;
```

Протокол `<ModuleOutput>`

Наименование

`<ModuleName>ModuleOutput.h`

Описание

Содержит методы, при помощи которых модуль общается со своим контейнером или другими модулями.

Дополнительные правила

- При использовании библиотеки ViperMcFlurry наследуется от протокола `<RamblerViperModuleOutput>`.

Примеры методов

```
- (void)didSelectMenuItem:(NSString *)menuItem;
- (void)didPerformLoginWithSuccess;
```

Слой Router

Класс `Router`

Наименование

`<ModuleName>Router.h / <ModuleName>Router.m`

Дополнительные правила

- При использовании библиотеки ViperMcFlurry держит weak-ссылку на `ViewController`, отвечающий за переходы этого модуля. Ссылка представляет собой свойство, закрытое протоколом `<RamblerViperModuleTransitionHandlerProtocol>`. Обычно эта переменная называется `transitionHandler`

Класс `Route`

Наименование

`<ModuleName>Route.h / <ModuleName>Route.m`

Описание

Если несколько роутеров в рамках одного приложения реализуют повторяющуюся логику по переходам на один экран - ее можно инкапсулировать в отдельном объекте-маршруте, который будет подключаться к нужным модулям. К примеру, это может пригодиться в случае экрана авторизации, на который можно перейти из разных модулей - настроек, профиля, бокового меню. Благодаря инкапсуляции этой логики в отдельном объекте, мы избавляемся от необходимости в роутере каждого из этих модулей писать один и тот же код.

Примеры методов

```
- (void)openAuthorizationModuleWithTransitionHandler:(id<RamblerViperModuleTransitionHandlerProtocol>)transitionHandler;
```

Протокол `<RouterInput>`

Наименование

`<ModuleName>RouterInput.h`

Описание

Содержит методы переходов на другие модули, которые могут быть вызваны презентером.

Примеры методов

```
- (void)openDetailNewsModuleWithNewsId:(NSString *)newsId;
- (void)closeCurrentModule;
```

Общие паттерны методов

- Для консистентности все методы этого протокола начинаются либо на `open-` (открытие какого-либо модуля), `close-` (закрытие модуля) или `embed-` (встраивание дочернего модуля в контейнер).

Слой View

Классы отображения (ViewController, View, Cell)

Наименование

- `<ModuleName>View.h / <ModuleName>View.m`
- `<ModuleName>ViewController.h / <ModuleName>ViewController.m`
- `<ModuleName>Cell.h / <ModuleName>Cell.m` .

Дополнительные правила

- Все `IBOutlet` 'ы и `IBAction` 'ы (то есть все зависимости и интерфейс) View выносятся в его публичный интерфейс.
- View держит strong-ссылку на презентер. Переменная называется `output` .

  Пример:

  `@property (nonatomic, strong) id<PostListViewOutput> output;`

Класс DataDisplayManager

Наименование

`<ModuleName>DataDisplayManager.h / <ModuleName>DataDisplayManager.m`

Описание

Объект, закрывающий логику реализации `UITableViewDataSource` и `UITableViewDelegate` . Работает только с данными, ничего не знает о конкретных `UIView` экрана. Обычно протоколом не закрывается, потому что конкретному экрану чаще всего соответствует одна конкретная реализация DataDisplayManager.

Примеры методов

```
- (id<UITableViewDataSource>)dataSourceForTableView:(UITableView *)tableView;
- (id<UITableViewDelegate>)delegateForTableView:(UITableView *)tableView withBaseDeleg
ate:(id <UITableViewDelegate>)baseTableViewDelegate;
```

Класс CellObjectFactory

Наименование

`<ModuleName>CellObjectFactory.h / <ModuleName>CellObjectFactory.m`

Описание

Зачастую удобно бывает выносить логику по созданию моделей ячеек из DataDisplayManager'a в отдельный объект, который, по сути, преобразует обычные модели в CellObject'ы.

Протокол

Наименование

`<ModuleName>ViewInput.h`

Описание

Содержит методы, при помощи которых презентер может управлять отображением или получать введенные пользователем данные.

Примеры методов

```
- (void)updateWithTitle:(NSString *)title;
- (NSString *)obtainCurrentUserInput;
```

Протокол

Наименование

`<ModuleName>ViewOutput.h`

Описание

Содержит методы, при помощи которых View уведомляет презентер об изменениях своего состояния.

Дополнительные правила

- В этом же протоколе находятся и методы, при помощи которых View уведомляет презентер о событиях своего жизненного цикла.

Примеры методов

```
- (void)didTapLoginButton;
- (void)didModifyCurrentInput;
```

Общие паттерны методов

- В большинстве случаев в качестве префикса каждого метода используем `did` - это указывает на пассивную роль View, который умеет выполнять ряд действий по

запросу и уведомлять об их окончании.

Примеры:

```
- (void)didTriggerViewWillAppearEvent;
- (void)didTriggerMemoryWarningEvent;
```

Слой Assembly

Класс Assembly

Наименование

`<ModuleName>Assembly.h / <ModuleName>Assembly.m`

Дополнительные правила

- В интерфейс Assembly выносится только метод, конфигурирующий View. Установка всего VIPER-стека - это детали реализации assembly, которые не должны быть известны окружающему миру.

Примеры методов

```
- (PostListViewController *)viewPostListModule;
- (PostListPresenter *)presenterPostListModule;
```

Общие паттерны методов

- Для более удобной автоподстановки у всех методов, создающих стандартные компоненты, указывается префикс `<ComponentName><ModuleName>Module`.

Комментарии

- Все методы протоколов и конкретных классов обязательно покрываются подробными javadoc-комментариями.

  Пример:

```
@protocol PostListInteractorInput <NSObject>

/**
Метод возвращает модель поста по определенному индексу

@param index Индекс поста в рамках просматриваемой категории

@return Пост
*/
- (PostModelObject *)obtainPostAtIndex:(NSUInteger)index;

/**
Метод возвращает общее количество постов для текущей ленты

@return Количество постов
*/
- (NSUInteger)obtainOverallPostCount;

@end
```

- Комментарии пишутся к интерфейсам всех уникальных компонентов модуля (различные хелперы, ячейки).

  Пример:

```
/**
Ячейка, используемая для краткого отображения поста в списке
*/
@interface PostListCell : UITableViewCell

...

@end
```

- К интерфейсам всех неуникальных компонентов (интерактор, презентер, view), а также к их протоколам, пишется одинаковый комментарий, описывающий предназначение всего модуля.

  Пример:

```
/**
Модуль отвечает за отображение списка постов в любой из лент приложения. Использует
тся как embedded-модуль.
*/
@interface PostListAssembly : TyphoonAssembly

...

@end
```

- В имплементации классов группы методов различных протоколов разбиваются при помощи `#pragma mark -`.

  Пример:

```
@implementation PostListPresenter

#pragma mark - PostListModuleInput

- (void)configureModuleWithPostListConfig:(PostListConfig *)config {
    ...
}

#pragma mark - PostListViewOutput

- (void)didTriggerPullToRefreshEvent {
    ...
}

- (void)didTriggerPullToRefreshEvent {
    ...
}

#pragma mark - PostListInteractorOutput

- (void)didProcessCacheTransaction:(CacheTransactionBatch *)transaction {
    ...
}

@end
```

Тесты

- Для каждого компонента модуля создается отдельный тест-кейс с названием вида `<ModuleName>ViewControllerTests.m`.
- Стараемся придерживаться правила один тест - одна проверка.

- Для разделения реализации теста на логические блоки используем нотацию *given/when/then*.
- Тесты методов различных протоколов разбиваются при помощи `#pragma mark -`.
- Так как в интерфейсе Assembly объявлены не все методы, для их проверки создается отдельный extension `_Testable.h`. В этом случае мы тестируем приватные методы, но такой подход обусловлен деталями реализации библиотеки *Typhoon*. В случае написания фабрики вручную, возможно проверить результаты работы компонента через его публичный интерфейс.
- Файловая структура модуля в тестах максимально повторяет файловую структуру проекта.

Дополнительные материалы Rambler&Co

Rambler.iOS V - V is for VIPER

В Rambler&Co периодически проводятся встречи iOS разработчиков. Одна из них была полностью посвящена VIPER - и стала основой для этой книги.

- Вступление (Видео) - Егор Толстой
- VIPER a la Rambler (Видео | Слайды) - Сергей Крапивенский
- Кодогенерация и Генерамба (Видео | Слайды) - Егор Толстой
- Переходы между модулями (Видео | Слайды) - Вадим Смаль
- Сложные модули (Видео | Слайды) - Андрей Зарембо
- Разбиваем Massive View Controller (Видео | Слайды) - Александр Сычев
- Тестирование VIPER (Видео | Слайды) -Станислав Цыганов
- VIPER и Swift (Видео | Слайды) - Валерий Попов
- Секция вопросов и ответов (Видео) - Егор Толстой, Сергей Крапивенский

Rambler&IT

Теоретические материалы - это отлично, но одной из главных проблем, с которой мы столкнулись при знакомстве с VIPER, это отсутствие примеров его применения в приложениях сложнее обычного *Hello World*.

Мы постарались решить этот вопрос и выложили в Open Source приложение Rambler&IT. Его основные особенности:

- Разбито на три основных слоя: `Presentation` , `BusinessLogic` , `Core` .
- Слой `Presentation` целиком написан с использованием VIPER.
- Слой `BusinessLogic` написан с использованием Service Oriented Architecture.
- Слой `Core` написан с использованием концепции составных операций, вдохновленной сессией 226 WWDC 2015.
- Для Dependency Injection активно используется Typhoon.

Приложение получилось достаточно крупным и сложным и продолжает активно развиваться. На настоящий момент это наиболее полный открытый пример использования VIPER в боевом проекте.

Другие конференции

Помимо Rambler.iOS мы и на других конференциях рассказывали про использование VIPER.

- VIPER - или то, о чем все говорят, но никто не рассказывает (Видео | Слайды) - Егор Толстой
- Чистая архитектура с VIPER (Видео | Слайды) - Сергей Крапивенский
- VIPER: наш взгляд на вопрос (Видео | Слайды) - Екатерина Коровкина

Подборка сторонних материалов

На просторах сети можно найти достаточно много различных статей, презентаций и видео, так или иначе связанных с VIPER. Не все из них одинаково полезны, некоторые даже вредны - но со многими ознакомиться не только можно, но и нужно.

Важно: Наличие материала в этом списке не означает нашего молчаливого согласия со всеми изложенными в нем идеями. Возможно, нам просто понравились шуточки или шрифты.

Статьи

- objc.io #13 - Architecting iOS Apps with VIPER

  Авторы: Jeff Gilbert, Conrad Stoll.

  Рецензия: *Неувядающая классика, которую вы уже должны были прочесть. Знаменательна благодаря двум фактам: во-первых, это статья времен еще годного objc.io, во-вторых - авторство принадлежит автору идеи VIPER Jeff Gilbert.*

  Как и любой из прочих материалов, не стоит принимать слишком всерьез - предлагаемая реализация в целом неоптимальна, а во многом и вообще ошибочна. Но зато есть тестовые проекты на ObjC и Swift.

- Introduction to VIPER

  Авторы: Jeff Gilbert.

  Рецензия: *Еще один must-read, MutualMobile рассказывают о том, как они докатились до VIPER. Особенное внимание стоит обратить на первые абзацы, где Jeff говорит о том, что к необходимости использования такой архитектуры их подвела потребность в покрытии UI тестами. Неплохой вброс и про историю появления названия - про первоначальные буквы VIP и додумывание E и R.*

  Тем не менее, с некоторыми позициями мы не согласны - в том числе с концепцией Wireframe, тотальным запретом на передачу ManagedObject'ов выше интерактора и прямым использованием DataStore.

- Brigade’s Experience Using an MVC Alternative

  Авторы: Ryan Quan.

Рецензия: *На наш взгляд, это главный претендент на роль лучшего вступления в VIPER. Хороший язык, простые схемы, четкое объяснение основных идей и принципов. Единственные (из популярных туториалов) рекомендуют выносить бизнес-логику в сервисный слой. Рекомендуется использование в качестве мотивационного материала для своей команды, семьи и друзей.*

Конечно, здесь нас снова ожидает наш старый знакомый - Wireframe. Кроме того, выделение DataManager'a (а мы его называем ServiceFacade) из интерактора - это достаточно редкий кейс, чтобы рекомендовать его для использования на постоянной основе во всех модулях.

- The Clean Architecture

  Авторы: Robert Martin.

  Рецензия: *Хоть и не напрямую относится к VIPER, но однозначно достойно прочтения. Дядюшка Боб раскрывает всем глаза на то, что такое чистая архитектура, рисует кружочки, говорит про DI и делает кучу других интересных вещей.*

  Переложить архитектуру в чистом виде на нашу суровую реальность вряд ли получится - но именно идеи из этой статьи послужили для MutualMobile толчком к VIPER. - iOS Architecture Patterns

  Авторы: Bohdan Orlov **Рецензия:** *Отличный материал, в котором по полочкам разложены особенности MVC, MVVM, MVP и VIPER. Красивые схемки, четко изложенные плюсы и минусы каждого из подходов, есть даже ссылка на нашу горячо любимую Генерамбу. Из минусов - как всегда, рассматриваются слишком утрированные примеры из пары десятков строк, на которых достаточно тяжело увидеть как потенциальную пользу, так и сложности в использовании.*

Подкасты

- iPhreaks Show - VIPER

  Участники: Conrad Stoll, Jeff Gilbert.

  Рецензия: *Если бы все то, о чем говорится в этом подкасте, было упомянуто в* `той самой` *статье на objc.io - многое могло пойти по-другому. Создатели VIPER подробно рассказывают о своей мотивации, подходах к рефакторингу, реализации сложных композитных экранов, тестировании, и многом другом.*

Незаслуженно пропущенный широким кругом iOS-разработчиков, этот подкаст - чуть ли не лучшее из того, что можно прочитать/увидеть/услышать относительно VIPER.

Видео

- 250 Days Shipping With Swift and VIPER

  Докладчик: Brice Pollock.

  Рецензия: *Бодро, весело, про свифт. Разработчик из Coursera рассказывает об их опыте работы с VIPER. Как и нас, ребят не удовлетворила каноничная модель, и они своими силами расширили ее, включив туда ViewModel, EventHandler, FlowController. Выглядит интересно, но схема обмена данными в рамках одного модуля на 12 минуте вызывает благоговейный ужас.*

- Clean Architecture - VIPER

  Докладчик: Sergi Gracia.

  Рецензия: *Немного про ответственности элементов, немного про тестирование, немного про SOLID, много про офис и команду Redbooth - даже с элементами воркшопа. А шрифты в презентации - просто огонь. Ничего необычного - просто еще одно введение в концепцию VIPER.*

  А главная претензия - очень плохое качество видео, поэтому однозначно параллельно стоит посмотреть сопутствующие слайды.