Структура лекции

- SOLID
- MVC
- MVP
- MVVM
- VIPER

Почему стоит позаботиться о выборе архитектуры?

Потому что если вы этого не сделаете, то в один прекрасный день, отлаживая огромный класс с десятками различных методов и свойств, вы окажетесь не в состоянии найти и исправить в нем ошибки. Естественно, такой класс трудно держать в голове как единое целое, поэтому вы всегда будете терять из виду какие-нибудь важные детали.

Плохая архитектура

SOLID

SOLID


S
Принцип единой ответственности (SRP)
O
Принцип открытости/закрытости (OCP)
L
Принцип заменяемости (LCP)
I
Принцип разделения интерфейсов (ISP)
D
Принцип инверсии зависимостей (DIP)

S

МГТУ им. Н.Э. Баумана
ТЕХНОПАРК
Mail.Ru Group

Принцип единственной ответственности (Single Responsibility Principle)

Существует лишь одна причина, приводящая к изменению класса.

Один класс должен решать только какую-то одну задачу. Он может иметь несколько методов, но они должны использоваться лишь для решения общей задачи. Все методы и свойства должны служить одной цели. Если класс имеет несколько назначений, его нужно разделить на отдельные классы.

O

Принцип открытости/закрытости (Open-closed Principle)

Программные сущности должны быть открыты для расширения, но закрыты для модификации.

Программные сущности (классы, модули, функции и прочее) должны быть расширяемыми без изменения своего содержимого. Если строго соблюдать этот принцип, то можно регулировать поведение кода без изменения самого исходника.

Принцип подстановки Барбары Лисков (Liskov Substitution Principle)

Функции, использующие указатели ссылок на базовые классы, должны уметь использовать объекты производных классов, даже не зная об этом.

Подкласс/производный класс должен быть взаимозаменяем с базовым/родительским классом.

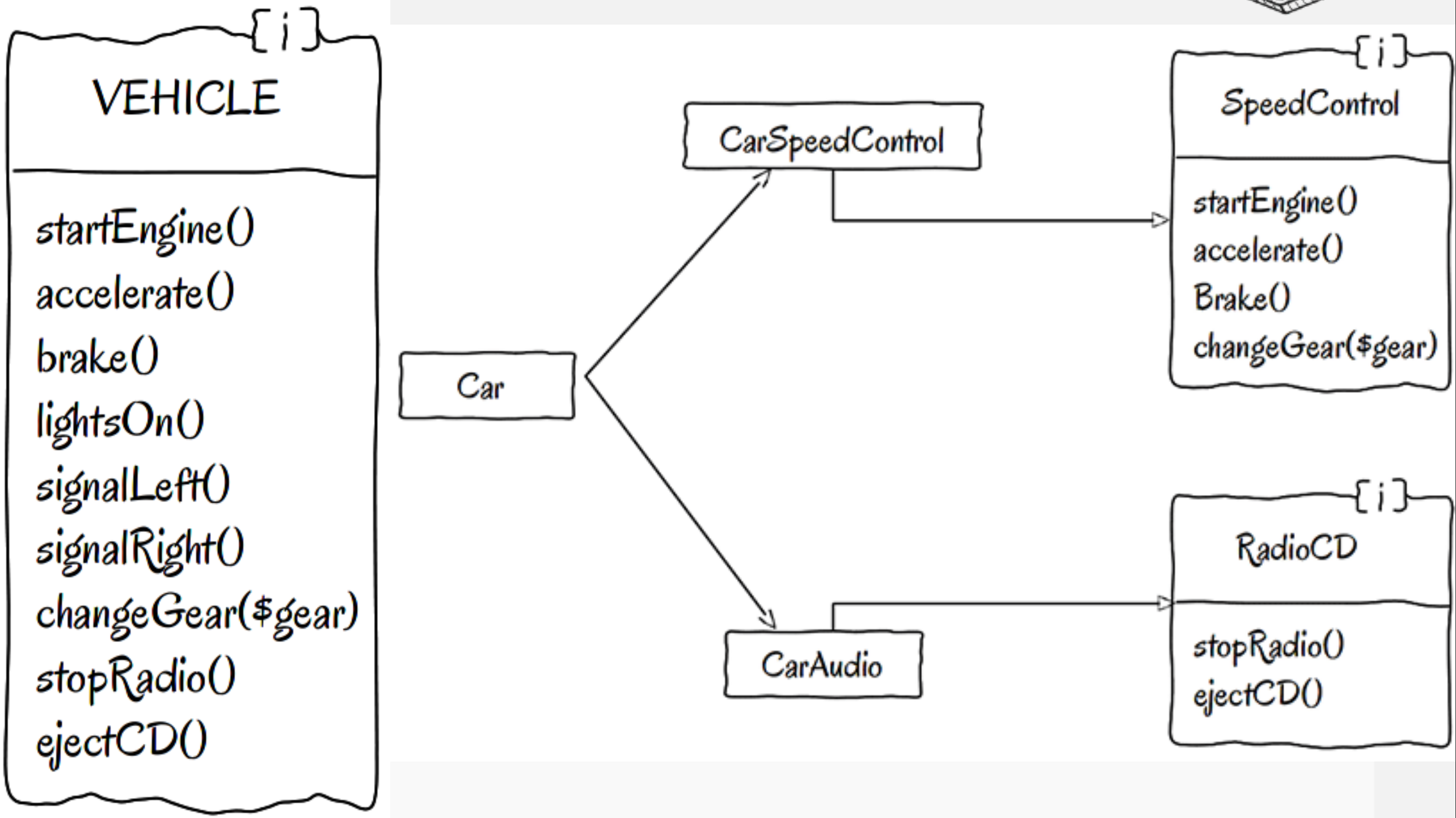
I

Принцип разделения интерфейса (Interface Segregation Principle)

Нельзя заставлять клиента реализовать интерфейс, которым он не пользуется.

Это означает, что нужно разбивать интерфейсы на более мелкие, лучше удовлетворяющие конкретным потребностям клиентов.

Цель принципа разделения интерфейса заключается в минимизации побочных эффектов и повторов за счёт разделения ПО на независимые части.


VEHICLE
startEngine()
accelerate()
brake()
lightsOn()
signalLeft()
signalRight()
changeGear($gear)
stopRadio()
ejectCD()
Car
CarSpeedControl
SpeedControl
startEngine()
accelerate()
Brake()
changeGear($gear)
CarAudio
RadioCD
stopRadio()
ejectCD()

D

Принцип инверсии зависимостей (Dependency Inversion Principle)

Высокоуровневые модули не должны зависеть от низкоуровневых. Оба вида модулей должны зависеть от абстракций.

Абстракции не должны зависеть от подробностей. Подробности должны зависеть от абстракций.

Проще говоря: зависьте от абстракций, а не от чего-то конкретного.

Применяя этот принцип, одни модули можно легко заменять другими, всего лишь меняя модуль зависимости, и тогда никакие перемены в низкоуровневом модуле не повлияют на высокоуровневый.

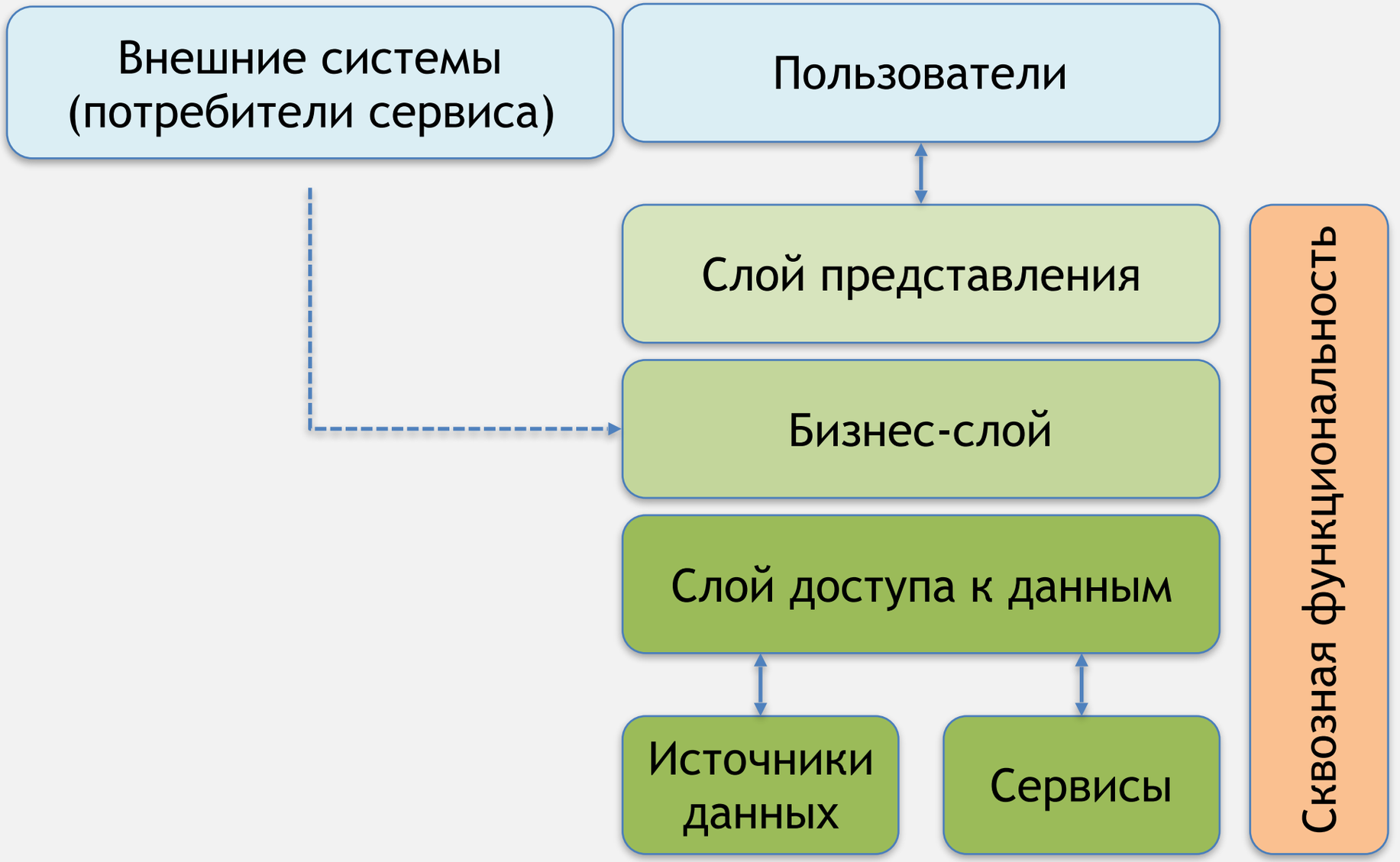
Признаки хорошей архитектуры

- сбалансированное распределение обязанностей между сущностями с жесткими ролями;
- тестируемость. Обычно вытекает из первого признака (без паники, это легкоосуществимо при соответствующей архитектуре);
- простота использования и низкая стоимость обслуживания.

Типовые составляющие приложения

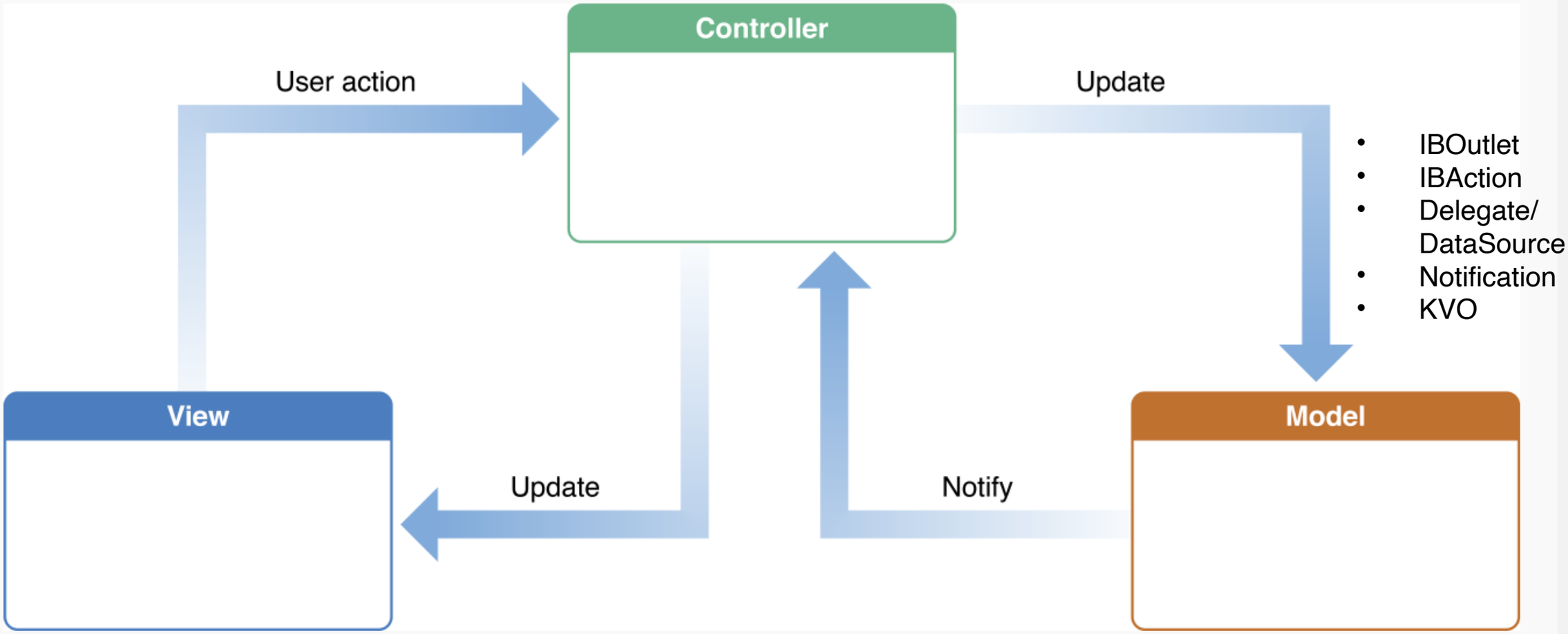
1. Пользовательский интерфейс (UI)
2. Бизнес-логика
3. Сетевое взаимодействие
4. Хранение/кеширование данных

Layered архитектура приложения



1. Models — ответственные за данные домена или слой доступа к данным, который манипулирует данными, например, класс Person или PersonDataProvider;
2. Views — ответственные за уровень представления (GUI); для окружающей среды iOS это все, что начинается с префикса UI;
3. Controller / Presenter / ViewModel — посредник между Model и View; в целом отвечает за изменения Model, реагируя на действия пользователя, выполненные на View, и обновляет View, используя изменения из Model.

Схема MVC



https://developer.apple.com/library/content/documentation/General/Conceptual/DevPedia-CocoaCore/MVC.html

View

- Стандартные элементы (view, label и т.п.)
- Показывают данные
- Общаются с пользователем
- Ничего не знают о модели

Controller

Координирует взаимодействие между View и Model (посредник)

- Заполняет view данными
- Обрабатывает события от view

- Передаёт данные в модель
- Берёт данные из модели (модель уведомляет об изменении данных)

Способы коммуникации между слоями

1. Делегат
2. Block / closure
3. Notifications
4. KVO
5. IBOutlet / IBAction (между View и Controller)

MVC: ожидания

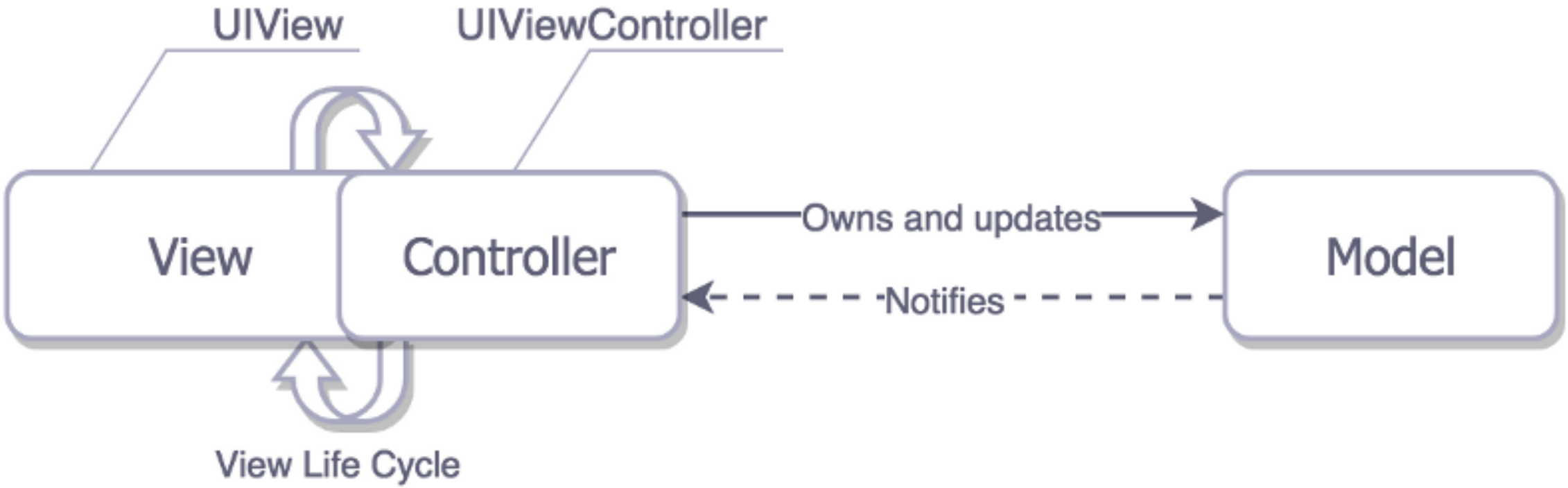
1. Controller является посредником между View и Model, следовательно, две последних не знают о существовании друг друга. Поэтому Controller трудно повторно использовать, но это, в принципе, нас устраивает, так как мы должны иметь место для той хитрой бизнес-логики, которая не вписывается в Model.

Massive View Controller

RAMBLER&Co Massive View Controller -> VIPER

MVC: реальность

1. Massive View Controller



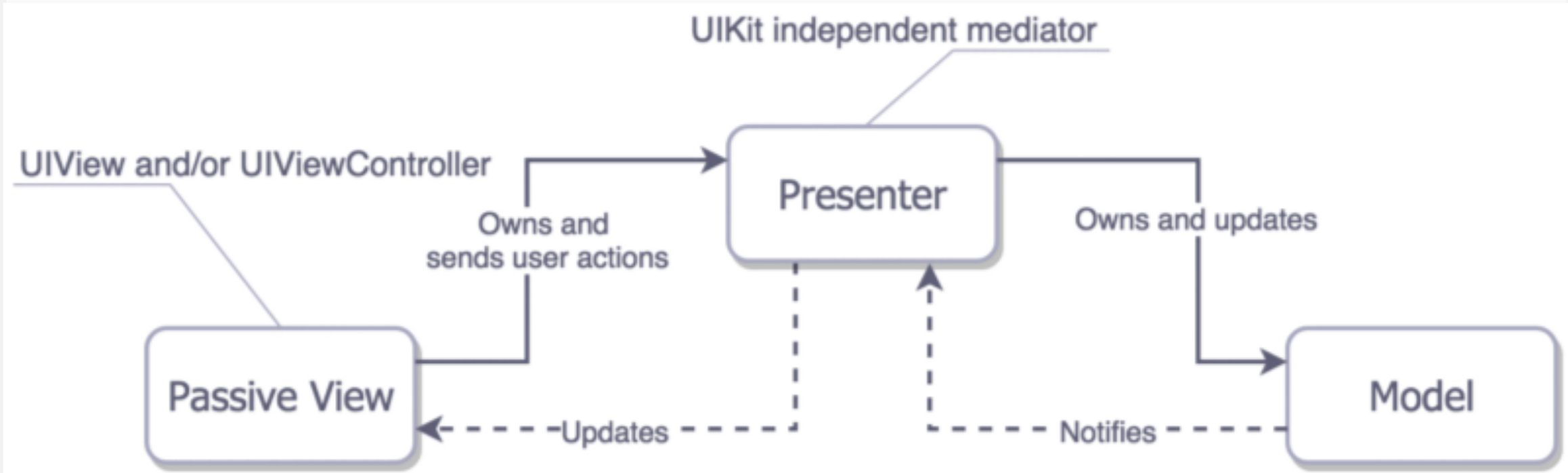
MVC: итоги

1. Распределение: View и Model на самом деле разделены, но View и Controller тесно связаны;
2. Тестируемость: из-за плохого распределения вы, вероятно, будете тестировать только Model;
3. Простота использования: наименьшее количество кода среди других паттернов. К тому же он выглядит понятным, поэтому его легко может поддерживать даже неопытный разработчик.

Архитектуры приложений

1. Предпосылки к возникновению разных архитектур
   - MVC не до конца учитывает специфику мобильных приложений
   - MVC может трактоваться кучей способов, и это плохо
2. Что решают другие архитектуры?
   - Унификация кода для того, чтобы можно было легче в нём разобраться
   - Уменьшение неопределённости при написании кода

MVP

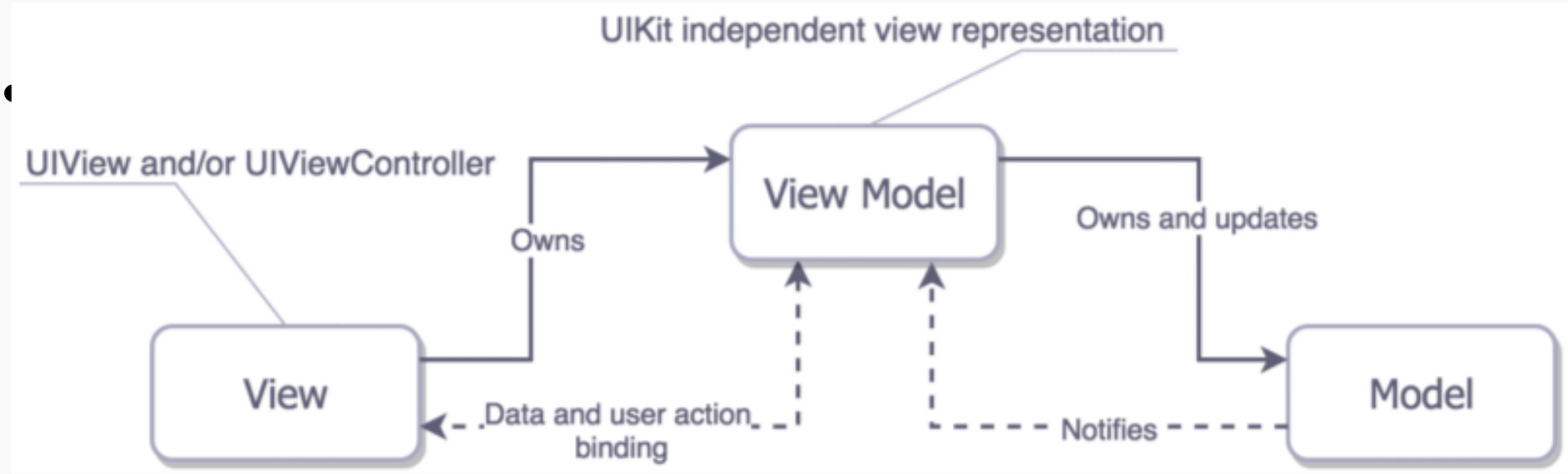

UIKit independent mediator
UIView and/or UIViewController
Presenter
Owns and
sends user actions
Owns and updates
Passive View
Updates
Notifies
Model

MVP: итого

1. Распределение: большая часть ответственности разделена между Presenter и Model, а View ничего не делает;
2. Тестируемость: отличная, мы можем проверить большую часть бизнес-логики благодаря бездействию View;
3. Простота использования: в нашем нереально простом примере количество кода в два раза больше по сравнению с MVC, но в то же время идея MVP очень проста

Архитектуры приложений (MVVM)

- MVVM (Model View ViewModel)
  - Отлично работает в связке с Rx* штуками
  - Отчасти решает проблему работы со сложными данными



Она очень похожа на MVP:

1. MVVM рассматривает View Controller как View;
2. в нем нет тесной связи между View и Model.

Так что такое View Model в среде iOS? Это независимое от UIKit представление View и ее состояния. View Model вызывает изменения в Model и самостоятельно обновляется с уже обновленной Model. И так как биндинг происходит между View и View Model, то первая, соответственно, тоже обновляется.

MVVM: итого

1. распределение: большая часть ответственности разделена между View Model и Model, а View ничего не делает;
2. тестируемость: View Model не знает ничего о представлении, это позволяет нам с легкостью тестировать ее. View также можно тестировать, но так как она зависит от UIKit, вы можете просто пропустить это;
3. простота использования: тот же объем кода, как в нашем примере MVP, но в реальном приложении, где вам придется направить все события из View в Presenter и обновлять View вручную, MVVM будет гораздо стройнее.

Основные архитектурные паттерны

1. MVC
   - чем плох?
2. MVP
   - похож на MVC, но
     - presenter не занимается layout
     - вместо view можно использовать mock-объекты
     - view controller относится к view
3. MVVM
   - похож на MVP, но связь не view и model, а view и view model (биндинг)
4. VIPER
   - ещё большее разделение обязанностей
     - view - UI
     - presenter - бизнес-логика, связанная с UI (всё инициализирует в модуле)
     - interactor - бизнес-логика, связанная с данными
     - router - переходы между модулями
     - entity - объекты данных

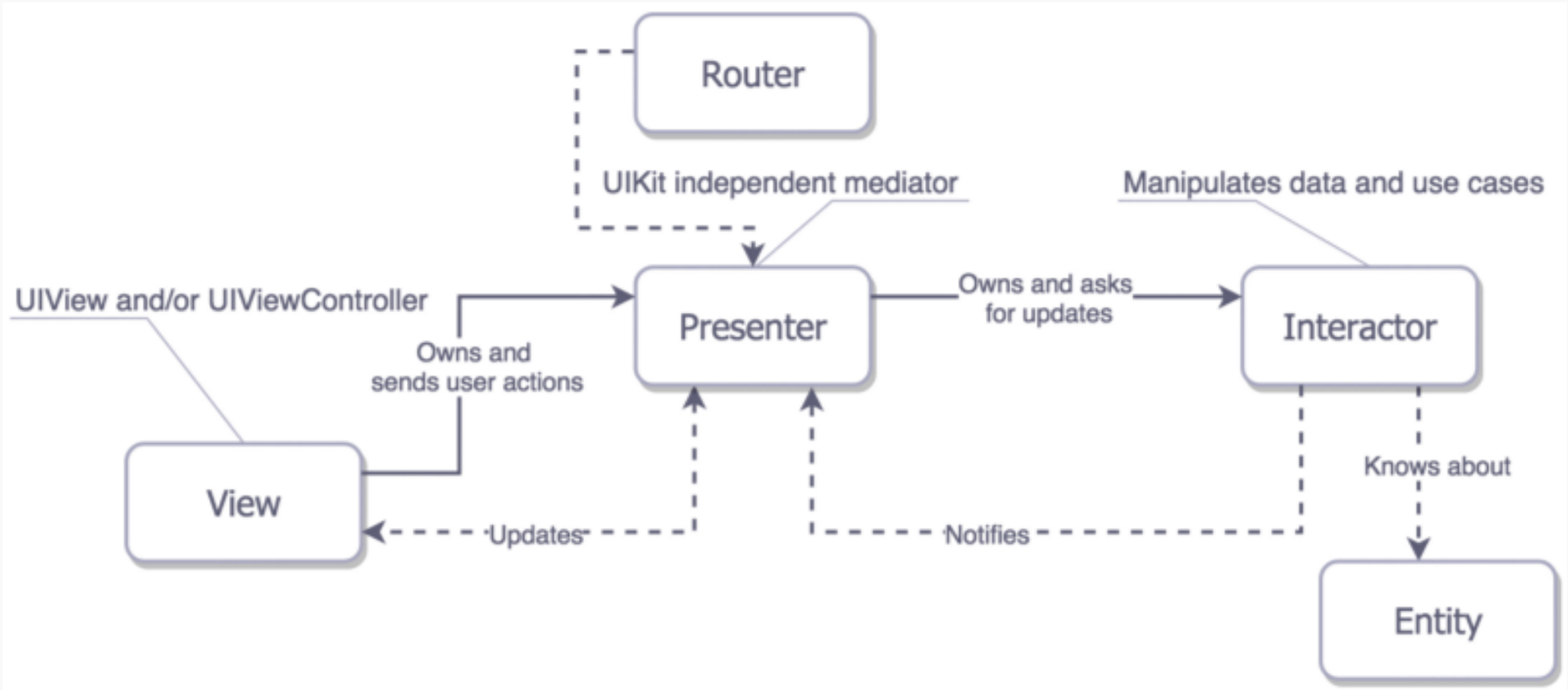
https://habrahabr.ru/company/badoo/blog/281162/

Архитектуры приложений

- VIPER
  - Стильно, модно, молодёжно
  - V - view (представление)
  - I - interactor (бизнес-логика)
  - P - presenter (связь всего со всем)
  - E - entity (сущности)
  - R - router (маршрутизатор)

Архитектуры приложений (VIPER)

- Основная единица — модуль
- Модули независимы друг от друга
- Модуль это не обязательно == экран, на сложном экране может быть несколько модулей
- Взаимодействие между модулями осуществляется через интерфейсы ModuleInput и ModuleOutput



Архитектуры приложений

- VIPER
  - Плюсы
    - Переиспользуемость модулей (хотя на практике это не часто применяется)
    - Тестируемость
  - Минусы
    - Многословность, на экран надо писать минимум 5 классов и кучу интерфейсов (отчасти решается кодогенерацией)

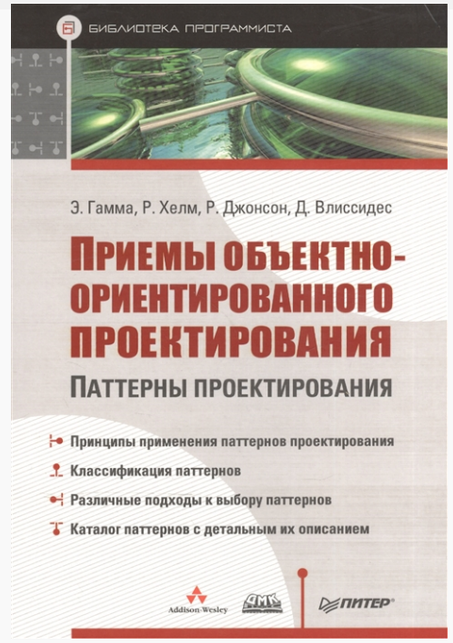
Какие еще бывают

1. VIP (View Interactor Presenter)
2. RIBs от Uber
3. Сбербанк, 100 разработчиков, ~1B строк кода
   - Все разбито на frameworks

Ссылки

- Банда четырех. Паттерны проектирования
  - Их применение в ios

- Про «тяжёлые» view controller’ы
  https://www.objc.io/issues/1-view-controllers/



- Про структуру проекта в Xcode:
  https://habrahabr.ru/post/261907/

- «Архитектурный дизайн мобильных приложений»
  часть 1: https://habrahabr.ru/company/redmadrobot/blog/246551/
  часть 2: https://habrahabr.ru/company/redmadrobot/blog/251337/

- Про архитектурные паттерны:
  https://habrahabr.ru/company/badoo/blog/281162/