Разработка под iOS. Взлетаем

Часть 4

Работа с архитектурой

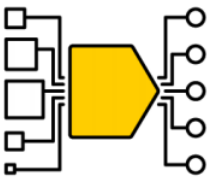
Оглавление

Глава 2

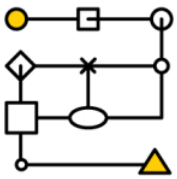
1 Разработка под iOS. Взлетаем

4.1. Классическая iOS архитектура

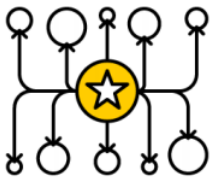
Привет! Меня зовут Дима, я занимаюсь разработкой iOS приложений в Яндексе. В этой части курса мы будем говорить про архитектуру.
Для начала обсудим, что же такое программная архитектура.



Слои
и модули
приложения

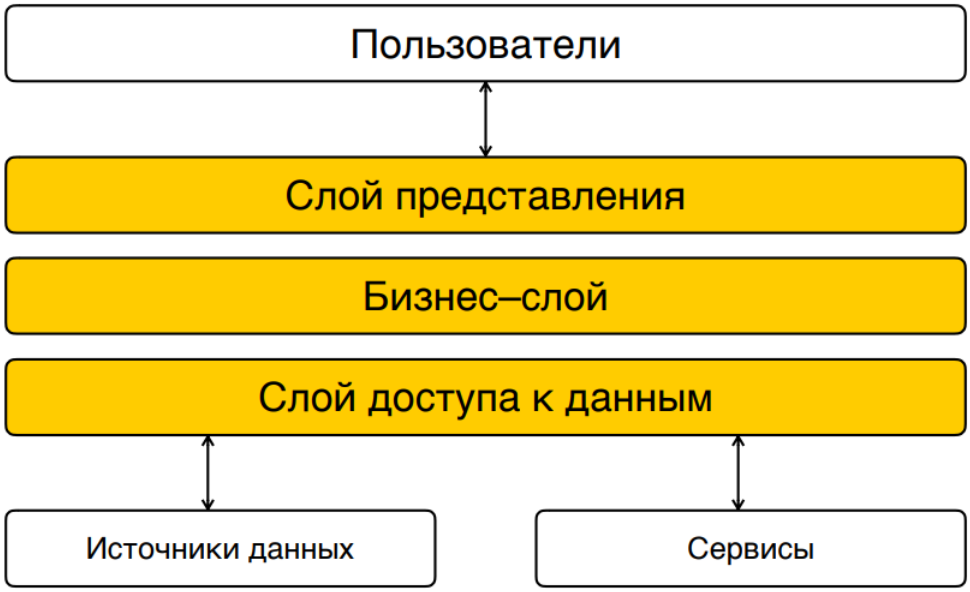


Связи
логических
блоков



Зависимости
между
сущностями

Это раздел программного дизайна, связанный со структурой приложений, то есть как код приложения разделен на различные интерфейсы и концептуальные слои, как потоки управления и потоки данных проходят в приложении между различными операциями и компонентами.
Возникает вопрос: что же такое концептуальный (либо архитектурный) слой? Минимально в любых приложениях можно выделить два слоя - модель и представление, то есть модель - это данные, а представление - это то, как эти данные отображаются пользователю и как пользователь может с ними взаимодействовать. Однако на практике, в современных приложениях, особенно в приложениях с богатым функционалом, архитектурных слоев часто около пяти или даже больше.
Чуть позже мы рассмотрим подробно, какими они бывают, но для общего представления сразу посмотрим на пример типичной многослойной архитектуры современного приложения.



Мы видим, сколько разных логических блоков может существовать в приложении, и нам предстоит разобраться, зачем они нужны и как и когда их уместно использовать. Таким образом, архитектура описывает компоненты приложения и связи между ними.
Теперь поговорим о том, на что же влияет архитектура приложения. Почему вообще так важно уделить внимание разработке хорошей архитектуры? Какие цели мы ставим перед собой, когда начинаем новый проект? (Ну, конечно, кроме того, что все функции приложения должны корректно работать и проект в целом был бы успешным). Важно:

1. чтобы стоимость расширения функционала не росла со временем;
2. чтобы работа могла быть удобно распараллелена между членами вашей команды;
3. чтобы вероятность случайно сломать работающий код была минимальна;
4. чтобы работа была прогнозируема (то есть мы могли более точно просчитывать, сколько же мы будем реализовывать ту или иную функцию);
5. чтобы предыдущие наши наработки эффективно использовались в будущем;
6. крайне важно, чтобы проект мог безболезненно пережить серьезные изменения (к примеру, замену любой библиотеки, которая работает, скажем, с базой данных или сетью, изменение формата данных и т.п.);
7. также очень важно, чтобы разработчик мог бы быстро разобраться в чужом коде либо в своем собственном коде, но который он написал достаточно давно;
8. ну, и еще один момент - тестируемость. Если компоненты приложения четко разделены между собой, и все их зависимости - явные и понятные, то их можно легко тестировать по принципу черного ящика - подавая данные на вход и проверяя их на выходе.

Эти пункты можно кратко сформулировать так:

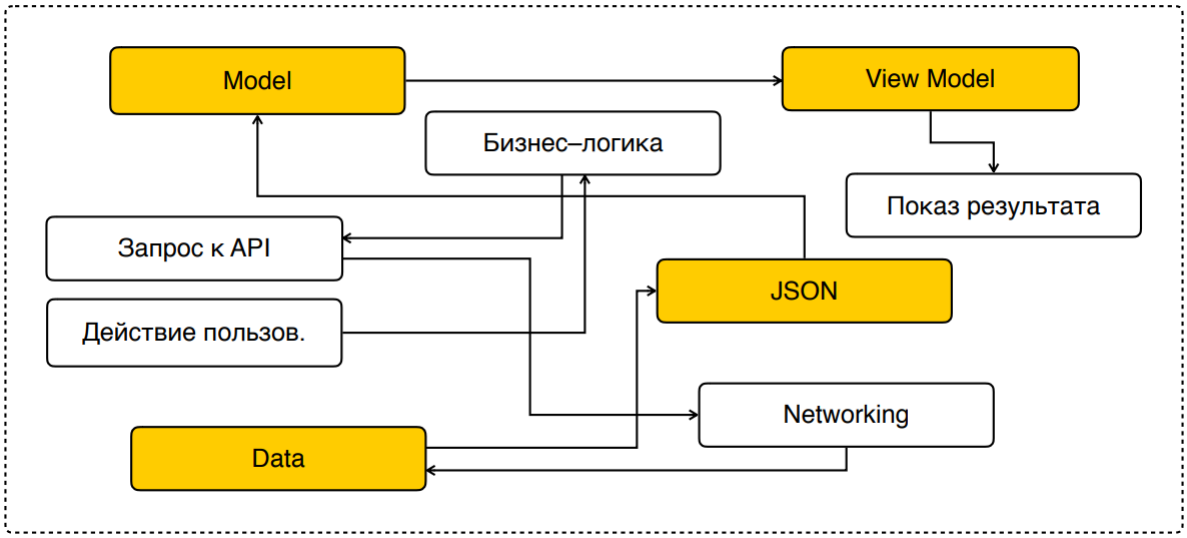
- Легкость развития, масштабируемость;

- Надежность, стабильность при изменениях;
- Прогнозируемость работы;
- Переиспользование наработок;
- Масштабируемость;
- Понятность кода;
- Тестируемость.

Этих целей нам предстоит научиться добиваться в условиях, когда приложение должно объединить в себе множество разных компонент: пользовательские события, сетевые запросы, кеширование данных, работу с графикой, фоновую обработку информации и так далее... Чтобы объединить эти компоненты и при этом обеспечить, чтобы данные надежно хранились и изменялись, а также корректно передавались между ними, требуется четкий набор правил, по которым эти компоненты будут между собой взаимодействовать.
Ну и прежде чем поговорить о том, как же добиться хорошей архитектуры, попробуем понять, что же такое плохая архитектура и какие у нее признаки.
Попробуем выделить несколько основных признаков плохой архитектуры приложения и дать им простые определения.

1. Первое: твердость кода. Это означает, что код сложно изменить, потому что каждое изменение вызывает много других изменений в других местах (причина этому - большое количество явных и неявных зависимостей в коде).
2. Второе: хрупкость. Это значит, что изменения вызывают поломки в других местах приложения, в том числе даже в концептуально несвязанных местах, что наиболее опасно!
3. Третье: неподвижность, то есть неизвлекаемость части кода для переиспользования в другом месте.
4. Четвертый термин тоже имеет интересное физическое название: вязкость. Тут речь о том, что написать правильный красивый код становится гораздо сложнее, чем написать какой-либо хак (по сути - грязный код, с которым будет неудобно работать в будущем).
5. Пятое: ненужные повторения. Это значит, что код содержит повторяющиеся структуры, которые могли бы быть унифицированы под некоей общей абстракцией и переиспользоваться.
6. И шестое: очень понятный термин - мутность. Это сложность чтения кода. То есть код не выражает явно то, для чего он предназначен, в него нужно долго вчитываться. Этого тоже нужно стараться избегать.

Сразу скажу, что одна из главных причин, по которым в приложениях появляются вот такие признаки плохой архитектуры (кроме незнания, конечно) - как правило, банальная лень! Ведь зачастую кажется, что реализовать какую-то функциональность максимально быстро (программисты иногда говорят "запилить") - это значит сделать хорошо. Но в долгосрочных проектах это определенно не так! Как мы уже говорили, грязный код очень быстро приводит к тому, что развитие проекта замедляется, разрабатывать его становится сложнее и дороже. Часто такие проекты через некоторое время приходится просто переписывать с нуля.
Рассмотрим самый типичный пример плохой архитектуры, очень специфичный для iOS-приложений, который даже получил имя собственное: массивные вью-контроллеры.

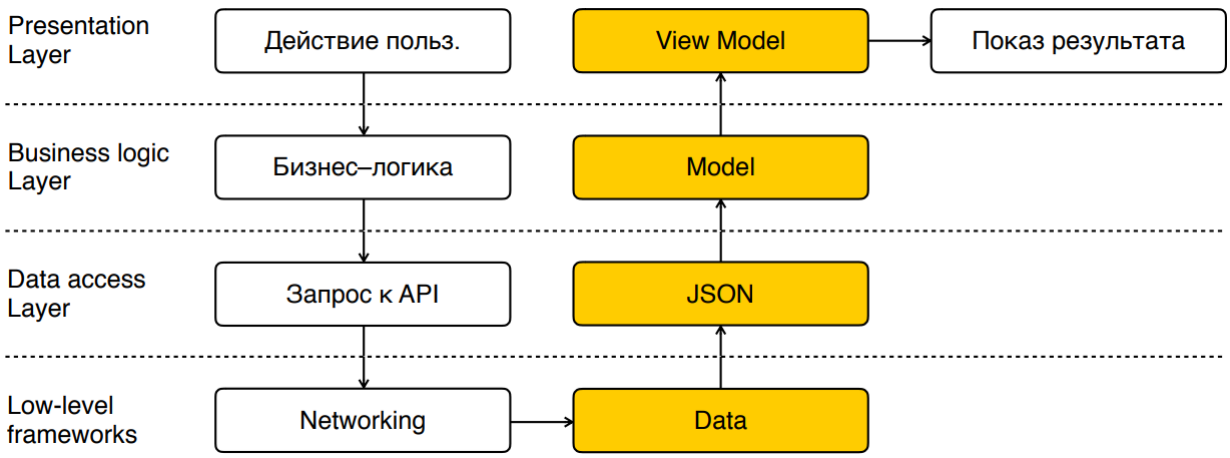


То есть это те классы, о которых мы подробно говорили в прошлых лекциях и которые часто разрастаются до огромных размеров, потому что начинают отвечать в приложении почти за все! И за отображение данных, и за их получение и обработку, и за принятие всех логических решений, и за реализацию алгоритмов и так далее! Именно такие классы, у которых слишком много ответственностей и большая внутренняя связность, и порождают те проблемы, о которых мы говорили. Код становится тяжело изменяемым и поэтому ненадежным, его становится трудно понять и еще сложнее - протестировать.

Как мы видим на картинке, массивный вью-контроллер содержит в себе всевозможную мешанину разнородных действий, которые объединяет только одно - общий экран в приложении. Все те проблемы, о которых мы говорили, обязательно возникнут и у разработчиков, создающих подобные классы.

Возможно, даже тестовое приложение, которые вы писали в процессе этого курса, имеет признаки чего-то подобного? Может быть, вы заметили, что писать задание для каждого последующего урока в рамках одного приложения становится сложнее и сложнее, потому что прошлый код начинает вам мешать?

И теперь давайте поговорим более конкретно, как же этих проблем избежать!

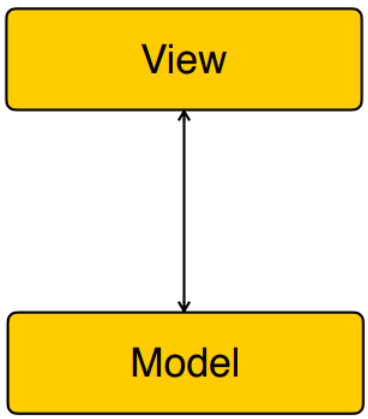


Посмотрите, насколько понятнее и логичнее выглядит схема, если разделить действия и соответствующие им данные по логическим слоям. Такие группировки и разделения могут быть разными и

зависеть от специфики вашего приложения и вашей команды.
По сути мы говорим о том, что в мире iOS разработки не существует единой "лучшей" архитектуры. Зато существует несколько видов архитектур, которые лучше работают для определенных видов проектов. А также существуют общие принципы, которые позволяют создать надежную, масштабируемую и удобную архитектуру.
То есть важно понять, что одну и ту же задачу можно реализовать при помощи разных архитектурных подходов. При этом при правильной реализации любого из них для пользователя результат будет одинаковым. Это означает, что фактически выбор и грамотная реализация архитектуры важна для нас самих, программистов. И этот выбор мы делаем, задавая себе примерно такие вопросы:

- какие задачи мы хотим, чтобы решались неявно, а какие явно прописывались в коде;
- где нам нужна гибкость, а где - единообразие;
- где мы хотим абстракции, а где - простоты.

Однако, когда выбор уже сделан, очень важно следовать ему во всем приложении: в противном случае появляется архитектурная раздробленность, которая снижает масштабируемость и повышает хрупкость кода.
Итак, мы с вами обсудили, почему важно уделить внимание архитектуре вашего приложения, поговорили о признаках плохой архитектуры и о том, что одну и ту же задачу можно успешно решить при помощи разных архитектурных подходов.
Далее мы рассмотрим несколько популярных в мире iOS разработки архитектур и разберем их особенности, преимущества и недостатки.
Мы с вами уже обсудили, почему так критично уделять внимание архитектуре ваших приложений, а также разобрали основные признаки плохой архитектуры.
А теперь поговорим, как же нам выстроить подходящую архитектуру, то есть обсудим, из каких логических блоков может состоять наше приложение, чтобы нам было удобно с ним работать, зачем эти блоки нужны и как они друг с другом взаимодействуют.
Мы будем с вами продвигаться от простых вариантов к более продвинутым и сложным. Все подходы, которые мы рассмотрим, на данный момент весьма широко распространены в мире iOS-разработки, поэтому современному iOS-разработчику весьма желательно в них ориентироваться.
Начнем с основы любой архитектуры мобильного приложения - двух слоев: модели и представления.



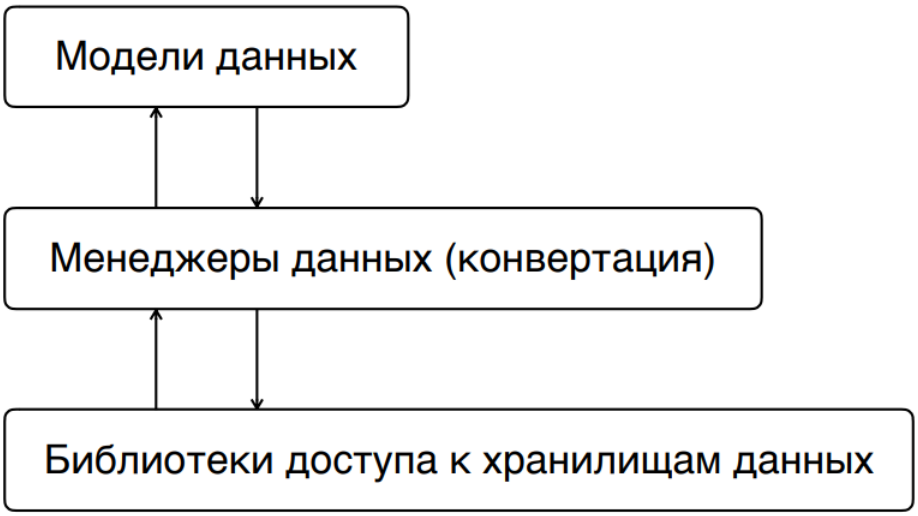
В прошлых уроках вы уже слышали о них, но давайте освежим это в памяти.
Итак, слой моделей - это абстрактное описание содержимого приложения, без всякой зависимости от

библиотек ввода-вывода, в частности от UIKit, то есть от классов, которые начинаются на UI - UIView, UIViewContoller и так далее. Как правило, слой моделей - это набор классов, которые описывают и обрабатывают данные, с которыми имеет дело пользователь. Например, если приложение - это социальная сеть, то модели будут описывать список друзей пользователя, его сообщения т.д.

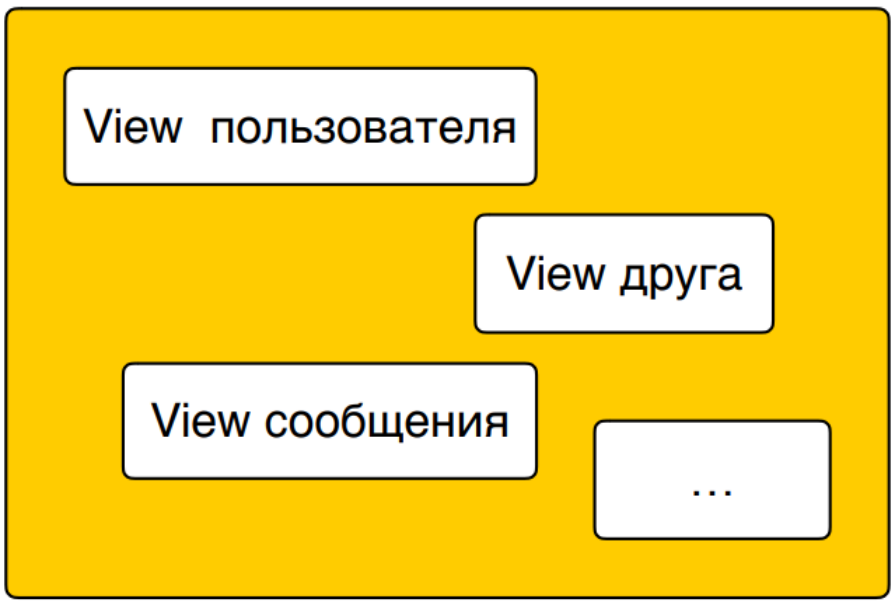
Модели, как правило, стараются реализовать как можно проще - в виде простых классов или структур. Иногда даже удобно объявить модель перечислением. С точки зрения разработки разница может быть весьма существенна - как вы знаете, в языке Swift структуры и классы по-разному передаются в функции: структуры копируются, а объекты классов передаются по ссылке. Но с архитектурной точки зрения нам важнее, что это наши собственные типы, описывающие объекты приложения.

```
public struct User {
    public let id: String
    public let firstName: String?
    public let lastName: String?
    public var avatarURL: String?
    public let birthDate: Date?
}
```

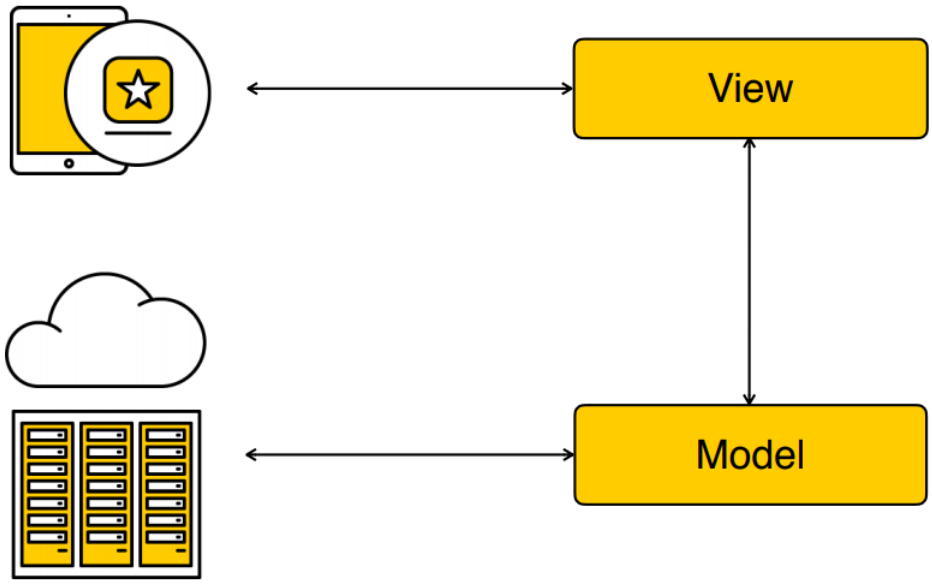
Данные же для объектов могут загружаться из сети или локального хранилища и сохраняться в оперативную память или на диск. Как правило процессами загрузки и сохранения управляют отдельные классы. Их часто называют менеджерами данных. Они отделяют объекты моделей от конкретных библиотек, отвечающих за загрузку и сохранение данных (например, URLSession, CoreData и т.д.).
Теперь несколько слов про слой представления - View.
Этот слой отвечает за то, чтобы сделать модели видимыми и интерактивными, то есть дать возможность пользователю взаимодействовать с данными в удобной для него форме. В отличие от слоя моделей, этот слой зависит от библиотек ввода-вывода, таких как UIKit. То есть в случае использования UIKit слой View будет содержать иерархию объектов UIView и другие UI-классы.



Когда мы говорим про модель или про представление, важно не запутаться. Эти слова, в зависимости от контекста, могут употребляться либо как названия целых слоев приложения, либо обозначать конкретные модели неких объектов в приложении или конкретные представления - например, view конкретной кнопки или view конкретной таблицы.



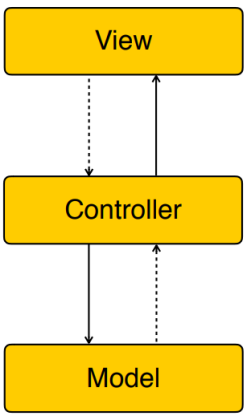
Далее. Слой моделей и слой view должны общаться между собой. Слой view должен отобразить данные слоя моделей, а слой моделей - измениться в ответ на действия пользователя, который с этим view как-то провзаимодействовал. Получается цикл обратной связи, то есть взаимодействие представления и модели друг с другом.



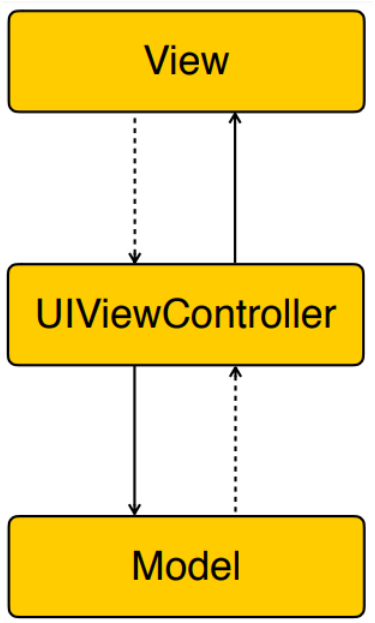
Это взаимодействие является базовым для всех видов архитектур. Можно сказать, что главная цель архитектуры - описать, как именно взаимодействуют view и модель между собой:

- связаны ли они напрямую или с помощью иных классов?
- кто создает сами модели и сами view?
- каким образом модель изменяется из-за действий во view?
- каким образом view реагирует на изменения в модели?

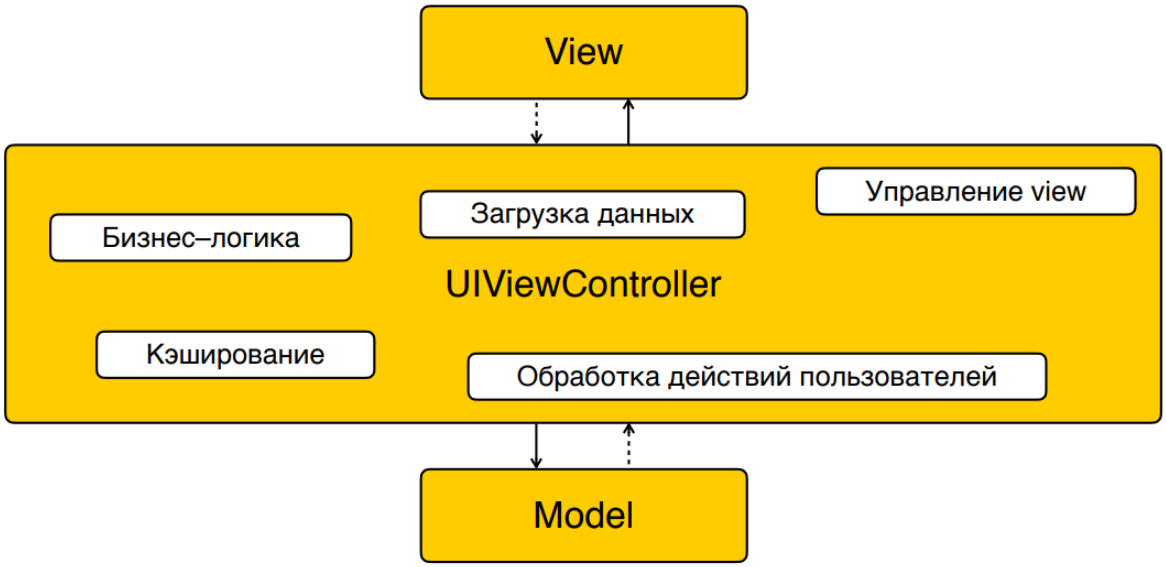
А дальше мы с вами рассмотрим, как же разные архитектуры отвечают на эти вопросы.
Итак, мы с вами узнали, что в основе любой архитектуры лежит взаимодействие модели и представления. Ну а теперь рассмотрим конкретные реализации.
И первой в нашем списке идет Cocoa MVC - архитектура, которой Apple предлагает пользоваться по умолчанию. Она содержит минимально возможное количество слоев для создания приложения, проста и удобна для начала разработки под iOS.



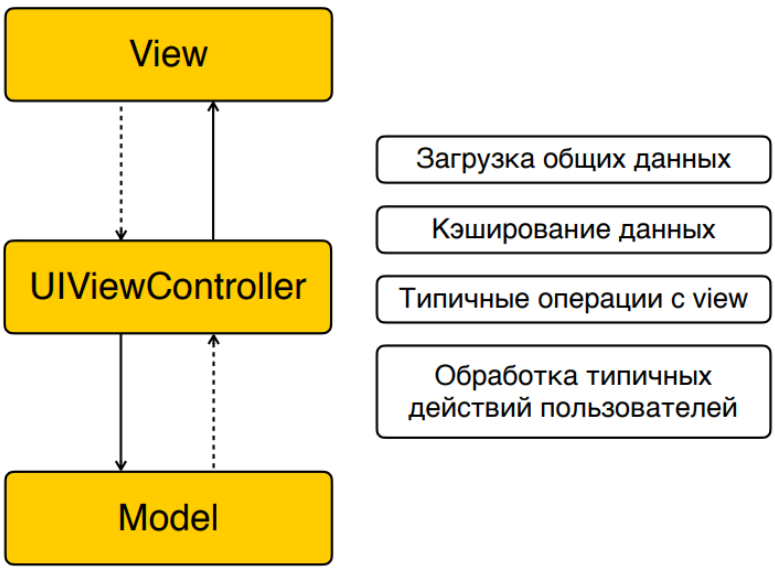
Главная идея этой архитектуры - наличие контроллера между view и моделью, который создает и настраивает их, дает им возможность общаться - то есть контролирует. По сути, контроллер описывает все поведение экрана приложения - и то, как пользователь может взаимодействовать с view, и как появляются и обрабатываются данные в моделях, и как эти данные отображаются во view.
Посмотрите на схему зависимостей в этой архитектуре: сплошные линии отражают отношения знания и владения, то есть контроллер знает про интерфейс view и моделей и имеет на них прямые ссылки. Пунктирные линии отражают отношения передачи данных, которые устанавливаются контроллером во время выполнения программы, то есть контроллер назначает себя принимающим эти данные от view и от моделей.
К примеру, если на view находится кнопка, контроллер назначает себя получателем действий от кнопки. Если на view лежит таблица - назначает себя делегатом, получающим данные о том, что с таблицей происходит.
Что касается обновления view при изменении данных моделей, то тут могут использоваться разные механизмы, например, KVO (key-value observing), или передача замыканий (closure) для последующих асинхронных обратных вызовов (так называемых колбэков).
Прямо из коробки Apple предоставляет нам класс UIViewController, который и используется в качестве контроллера в стандартном Cocoa MVC.



Для каждого из экранов нашего приложения мы создаем наследника - UIViewController - и описываем всю логику работы этого экрана. То есть на каждое событие (например, нажатие кнопки пользователем) в этом классе будет отдельная функция, которая решает, что должно произойти дальше.
Например, она может обратиться к базе данных за информацией, обработать ее, обновить некий объект модели в памяти и вызвать обновление view. Такие классы получают всю власть над поведением приложения. С одной стороны, это просто для реализации, с другой - довольно быстро такие классы превращаются в большие запутанные клубки логического и презентационного кода, и их становится сложно развивать и тестировать.



Но это еще не самое страшное, ведь если хорошо структурировать код, красиво назвать функции и переменные, писать исчерпывающие комментарии к коду, то его вполне можно будет поддерживать и развивать. Но давайте представим себе ситуацию, когда в нашем приложении - десятки экранов. И у нас обязательно возникнут ситуации, когда на разных экранах мы должны будем выполнять схожие действия с данными, например, обновлять информацию о пользователе из сети и сохранять ее в базу данных или проверять подключение к серверу.
И вот таких вещей, как дублирование идентичного кода в разных классах, быть уже точно не должно. Рано или поздно идентичный код поменяют в одном месте, но не поменяют в другом, и в приложении начнут появляться ошибки, чем дальше - тем больше. Поэтому общие вещи нужно обязательно выносить в отдельные классы, например, менеджер работы с базой данных, менеджер работы с сервером, сервис обновления статуса пользователя и так далее.



Сама по себе архитектура MVC не подскажет вам, как именно это сделать, поэтому тут есть определенная свобода творчества. Однако, когда вы выберете подход для таких вещей, то важно его придерживаться, чтобы вся ваша команда (или вы сами через некоторое время) могла легко ориентироваться в коде приложения. Какие тут могут быть приемы?
Допустим, есть функциональность, которая может понадобиться на любом экране (например, показать сообщение пользователю). Тогда можно создать extension для класса UIViewController с нужными для этого функциями, и вы автоматически наделяете такой функциональностью все ваши экраны. То есть в каждом вашем вью-контроллере появится доступ к функции показа сообщения пользователю так, как если бы она была написана в этом же классе.

```
//MARK: - Alerts
public extension UIViewController {
    public func showAlert(title: String, message: String?, completion: () -> Void?)
{
        //...
    }
}

//MARK: - Formatting
public extension UIViewController {
    public func shortDate(with: Date) -> String {
        //...
    }

    public func fullDate(with: Date) -> String {
        //...
    }
}
```

То же может касаться, например, общих функций форматирования данных для отображения на экране (например, мы хотим, чтобы время и даты во всем приложении показывались одинаково).
Ну а если вам нужна некая функциональность, которая требуется не везде, а только на нескольких экранах? Тогда лучше инкапсулировать ее в отдельных классах и обращаться к этим классам из вью-контроллеров. И тут тоже есть несколько вариаций.

- Если эта функциональность не требует хранения промежуточных данных (то есть классу не нужны будут переменные), то функции можно реализовать статическими (то есть со словом static), и тогда не нужно будет заботиться о том, что объект класса с нужной функциональностью нужно создать и хранить на него ссылку.

  ```
  public class AlertUtils {
      public static func showAlert(title: String, message: String?, completion:
  () -> Void?) {
          //...
      }

      //Usage example: AlertUtils.showAlert(title: "...", message: "...",
  completion: {...})
  }

  public class FormatUtils {
  ```

```
    public static func shortDate(with: Date) -> String {
        //...
    }

    public static func fullDate(with: Date) -> String {
        //...
    }

    //Usage example: view.label.text = FormatUtils.fullDate(with: date)
}
```

- Если же переменные все же нужны, то у нас тоже есть несколько вариантов, как создать и хранить этот объект:
  - Первый вариант - сделать его членом класса нашего вью-контроллера, используя директиву lazy, то есть этот вспомогательный объект будет создан при первой необходимости и существовать до конца жизни самого вью-контроллера.
    Это неплохой вариант, однако иногда за созданием таких объектов стоят большие накладные расходы - например, инициализация базы данных или зачитка данных в память. И тогда наш экран может притормаживать в момент создания таких объектов.

    ```
    class SomeViewController: UIViewController {
        private lazy var alertUtils: AlertUtils = {
            return AlertUtils(params: ...)
        }()

        private lazy var formatUtils: FormatUtils = {
            return FormatUtils(params: ...)
        }()

        //...
    }
    ```

  - Другой вариант - разделять такие объекты между всеми вью-контроллерами с помощью шаблона синглтон (одиночка), то есть, зная имя класса, мы можем обратиться к его единственному объекту с помощью метода, который, как правило, называется shared или current. Этот метод будет гарантировать, что объект будет создан лишь один раз и накладные расходы по его созданию нужно будет тратить лишь единожды за всю жизнь приложения.
    Этот вариант прост и даже популярен, однако нужно также понимать и минусы. Во-первых, этот объект может жить в памяти очень долго и накапливать в себе некие данные, которые могут со временем устаревать, поэтому для таких случаев может понадобиться реализовать механизм проверки актуальности данных. Кроме того, такие объекты могут предоставлять вью-контроллерам слишком много возможностей, которые им не принципиально нужны.

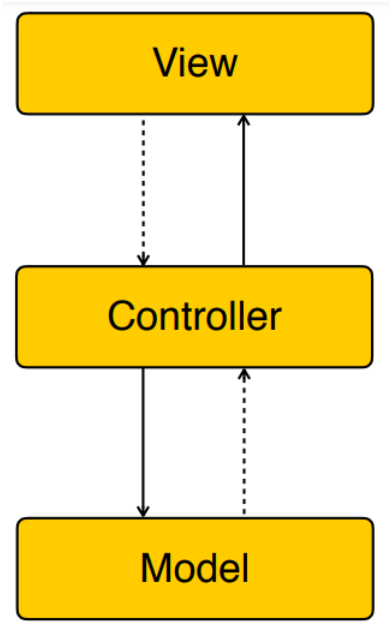
    ```
    final class AlertUtils {
        static let shared = AlertUtils()

        private init() { ... }

        public func showAlert(title: String, message: String?, completion:
    () -> Void?) {
    ```

```
        //...
    }

    //Usage example: AlertUtils.shared.showAlert(title: "...", message:
"...", completion: {...})
}

final class FormatUtils {
    static let shared = FormatUtils()

    private init() { ... }

    public func fullDate(with: Date) -> String {
        //...
    }

    //Usage example: view.label.text = FormatUtils.shared.fullDate(with:
date)
}
```

– Есть еще один вариант решения подобной задачи - так называемый локатор сервисов (service locator). Это некий общедоступный объект, который позволяет найти нужный сервис по требуемым критериям. То есть мы запрашиваем его не по имени класса, а по набору возможностей - например, сервис, который умеет читать данные о пользователе из базы данных. Этот набор возможностей, как правило, представляется в виде протокола - набора функций. Таким образом, объект, который мы получим, для нас не будет уметь лишнего и от нас будет скрыто, какой же класс по факту реализует эту функциональность. Это хорошо еще и тем, что со временем эту функциональность сможет реализовать другой класс, но нам не нужно будет переписывать код наших вью-контроллеров, потому что имя протокола останется прежним.

```
class SomeViewController: UIViewController {
    func showUserInfo() {
        //Loading user info:
        if let userInfoProvider: UserInfoProviderProtocol? =
ServiceLocator.shared.getService() {
            userInfoProvider.loadData { [weak self] userData in
                //Showing user info:
                self?.setupUserView(with: userData)
            }
        }
    }
}
```

Как мы с вами помним, у всякой архитектуры есть плюсы и минусы. Давайте рассмотрим, какие же они у классического MVC.

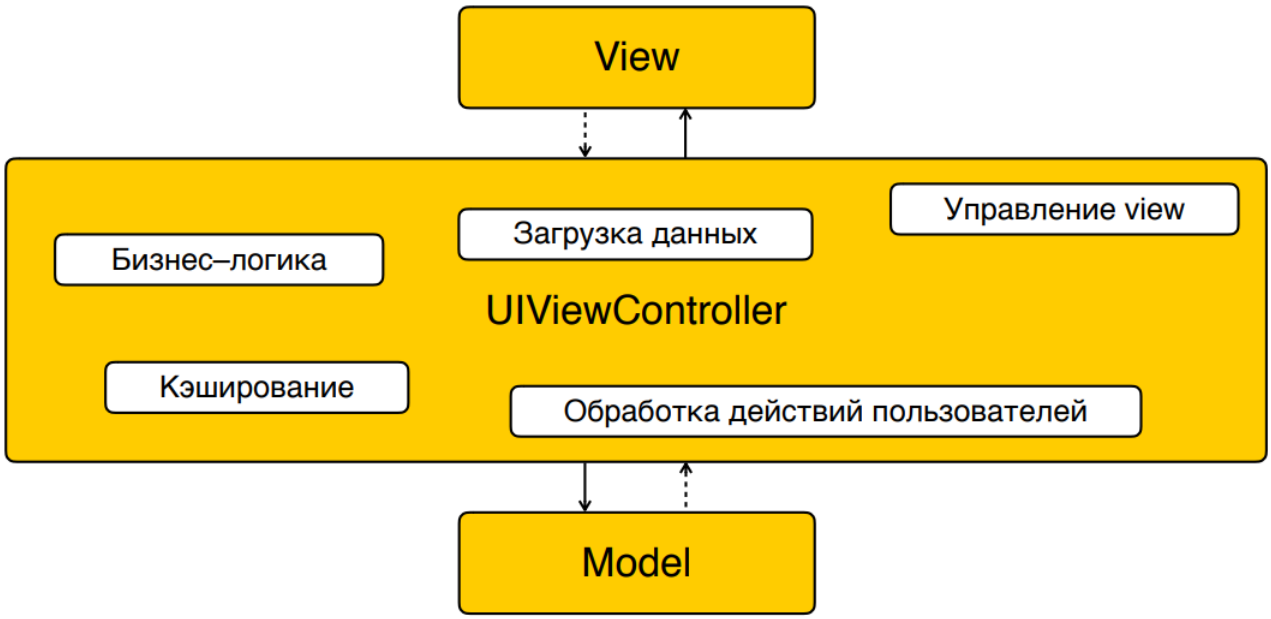


Итак, плюсы: безусловно, это быстрое начало разработки, отсутствие необходимости дополнительной подготовки, поскольку каждый iOS-разработчик хорошо понимает эту архитектуру. Подходит она хорошо для небольших коротких проектов, желательно с одним разработчиком в команде. При этом у архитектуры много минусов: массивные вью-контроллеры, плохая масштабируемость (то есть растущая сложность разработки), сложность тестирования, трудности при одновременной работе нескольких разработчиков над одним экраном, проблемы при необходимости серъезных изменений в функционале.

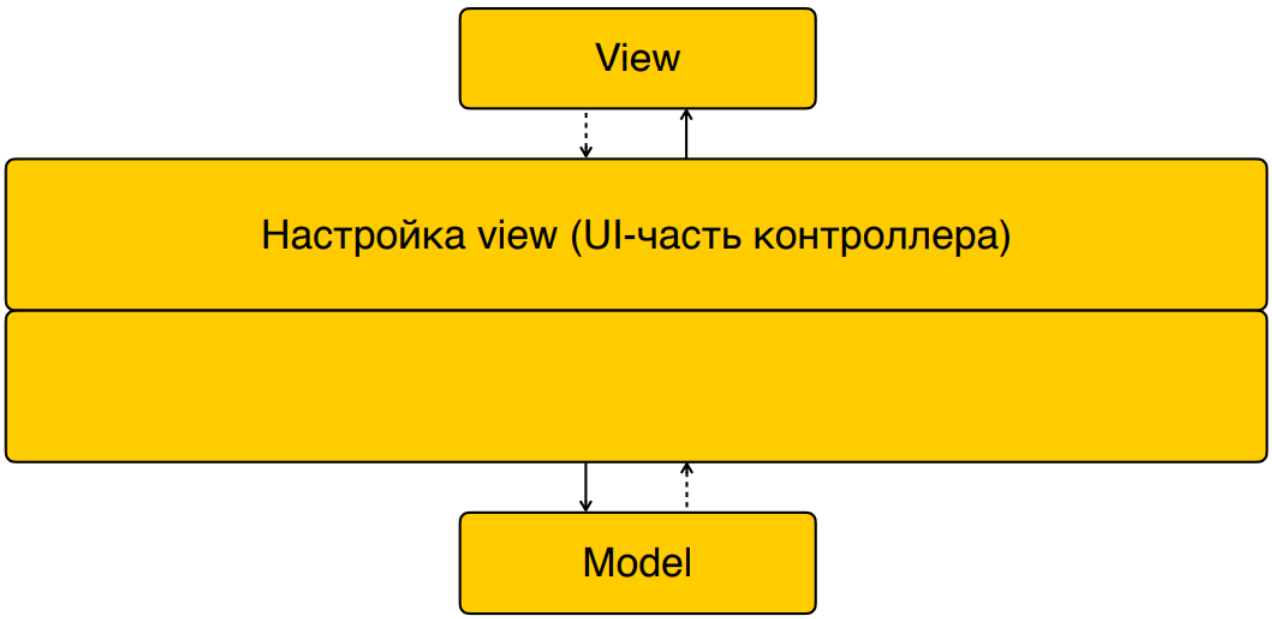
В следующих разделах я расскажу о различных модификациях MVC, которые призваны уменьшить эти проблемы.

4.2. Типичные улучшения и модификации

В прошлом разделе мы с вами поговорили про классическую iOS архитектуру Cocoa MVC и обозначили ее основные проблемы. Теперь поговорим о том, как же можно эти проблемы минимизировать. Будем решать одну из главных проблем - слишком большое количество ответственности у вью-контроллеров.

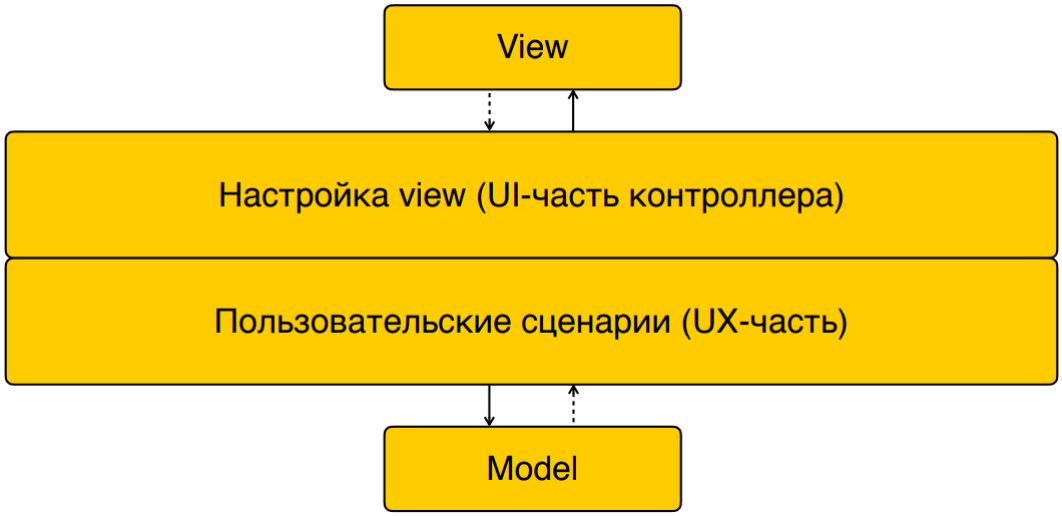


Поговорим о том, какие логические части могут быть вынесены из него, другими словами - абстрагированы от вью-контроллера.
Когда мы пишем код для какого-либо экрана, одна из основных вещей, которую мы должны описать - пользовательский интерфейс. И это описание состоит из двух частей. Первая - это настройка view, из которых состоит экран: позиция на экране, цвета, шрифты, надписи и т.д.



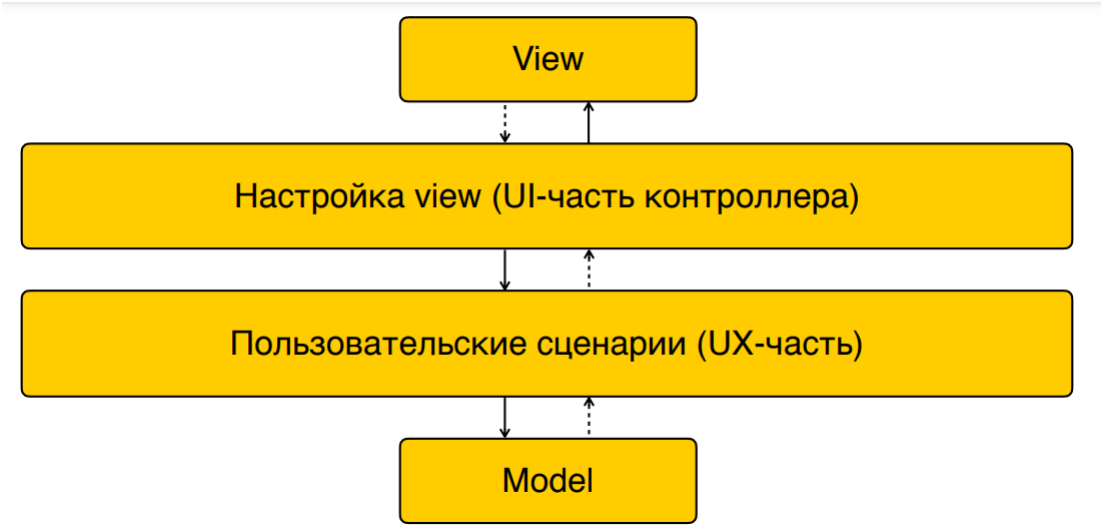
Эти вещи можно задать и в interface builder-е, но и в коде вью-контроллера нам зачастую нужно написать некоторые настройки. Это могут быть, например, динамическое конструирование надписей,

закругления углов у view, настройка таблиц, запуск анимаций и прочее... Это то, что называется настройка UI, и такой код очень хорошо подходит под сферу ответственности вью-контроллера.
Но есть и вторая часть описания интерфейса, которая называется UX - user experience - пользовательский опыт.

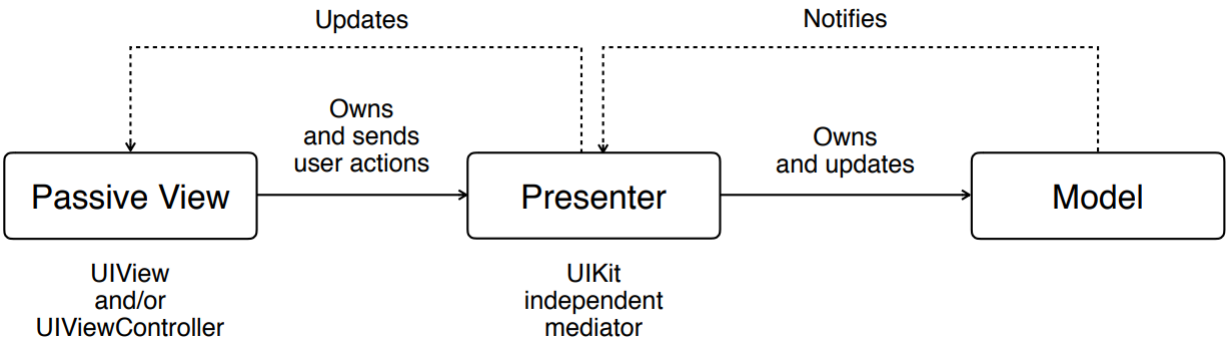


Грубо говоря, это реакции экрана на действия пользователя: как, например, нажатие на кнопку влияет на то, ЧТО будет показано пользователю дальше. То есть код UX - это содержимое методов, обрабатывающих все действия пользователя. Этот код может запрашивать какие-то данные у других классов и затем вызывать обновление view экрана. Либо, например, задать пользователю вопрос - действительно ли он хочет выполнить это действие. Либо инициировать показ другого экрана. Такой код называется презентационной логикой и может быть абстрагирован (то есть вынесен) в классы, которые называются презентеры.

Итак, модификация архитектуры MVC, в которой презентационная логика вынесена в презентеры, называется ModelViewPresenter - MVP. В ней контроллер разбивается на два класса - Презентер, который отвечает за то, что показывать пользователю, и вью-контроллер, который отвечает за то, как это показывать.



При таком подходе каждый вью-контроллер содержит сильную ссылку на объект соответствующего презентера, то есть владеет им, а презентер - слабую ссылку на объект вью-контроллера.



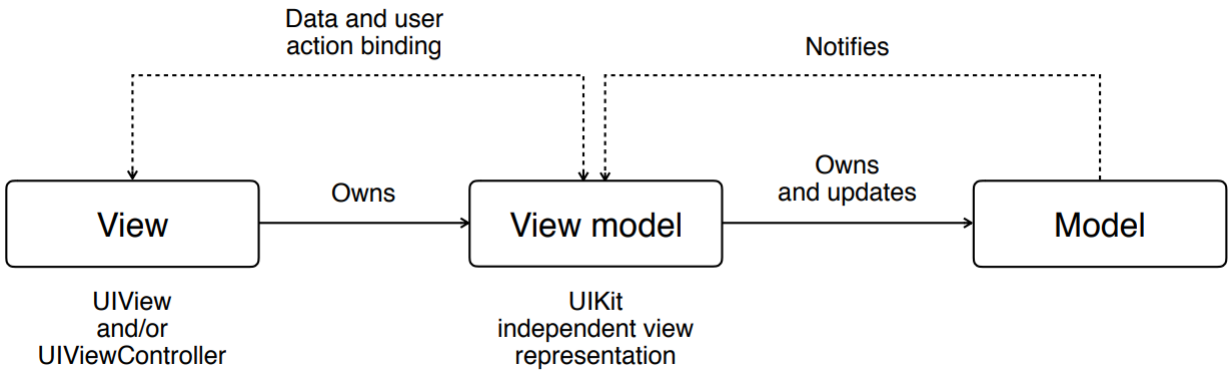
То есть, когда вью-контроллер уничтожается при закрытии экрана, презентер уничтожается сам собой. При этом очень желательно, чтобы эти ссылки были не на сами классы, а на протоколы, реализуемые этими классами. То есть вью-контроллер объявляет и реализует для презентера протокол с набором методов для обновления интерфейса данными, которые есть в распоряжении презентера. А презентер, в свою очередь, объявляет и реализует протокол, который вью-контроллер вызывает по действиям пользователя - нажатиям на кнопки и т.п.

```
protocol SomeViewProtocol {
    func setTitle(_ title: String, animated: Bool)
    func setSomeImage(_ image: UIImage?)
    func setSomeButton(hidden: Bool)
    func setActivityIndicator(visible: Bool)
    ...
}

protocol SomePresenterProtocol {
    func didTapSomeButton()
    func didEnterSomeText(_ text: String)
    func numberOfSomeRecords() -> Int
    func someRecordViewModel(at index: Int) -> SomeRecordViewModel
    ...
}
```

Таким образом, в MVP вью-контроллер становится пассивным и практически не принимает самостоятельных решений. Вместо него это активно делает презентер, поскольку только он знает, что нужно делать дальше.
Какие плюсы данного подхода:

- Разделяется ответственность, то есть то, что описано во вью-контроллере, инициирует дизайнер, а то, что в презентере - специалист по UX или менеджер вашего продукта.
- Более того, по тому, какой класс поменялся - вью-контроллер или презентер - вы сможете легко понять, изменился визуальный дизайн либо же поведение экрана.
- Также появляется возможность легкого юнит-тестирования логики презентера, поскольку презентер, как правило, независим от UIKit, поэтому можно протестировать исключительно наши собственные классы.

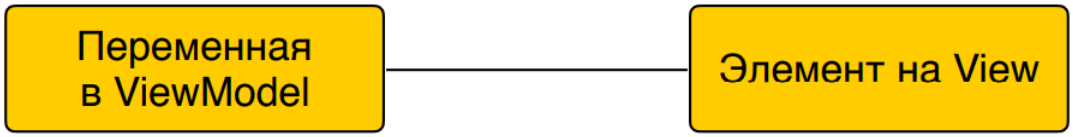
Минус же данного подхода в том, что он не является панацеей. Если применять исключительно его, нагружая презентеры не только презентационной логикой, но и загрузкой и обработкой данных, то презентеры скоро сами станут массивными и их станет сложнее и сложнее развивать. Поэтому для больших проектов этот подход также комбинируется с другими видами абстракций, о которых мы поговорим дальше.
Ну, а примеры использования MVP для iOS вы легко найдете на github в open-source проектах. Далее мы с вами посмотрим на еще одну вариацию MVC - MVVM.
Итак, мы узнали про архитектуру MVP, но есть еще одна похожая архитектура, которая также умеет абстрагировать презентационную логику, но делает это без презентеров. Она называется MVVM - Model-View-ViewModel. Давайте посмотрим на ее схему.



Тут место презентера занимает так называемая вью-модель - это класс, который хранит в себе данные, подготовленные для того, чтобы быть отображенными на экране. Вью-модель содержит в себе структуры, в которых находятся лишь такие типы данных, как строки, флаги либо простые перечисления для стилей отображения чего-либо. То есть, вью-модель нужна чтобы, например, преобразовать дату из типа Date в строку, которая будет показана на экране. Или некое логическое значение (например температуру воздуха) - в перечисление со стилем ее визуального отображения - холодным или теплым. А уже вью-контроллер преобразует стиль отображения в конкретный цвет.

```
struct UserProfileViewModel {
    let name: String
    let surname: String
    let avatarImage: UIImage
    let age: Int
    let status: String
    let roleName: String
    let roleViewType: UserRoleViewType
}
```

Как и презентеры, вью-модели, по-хорошему, не должны заниматься обработкой логических данных, они лишь должны конвертировать их и настраивать связи данных с view. Но в отличии от презентера, который активно управляет пассивным вью-контроллером, вью-модель использует так называемые binding'и для связки данных с элементами view. То есть когда изменение происходит на одном конце связки, второй конец моментально получает об этом информацию, и наоборот.



Например, binding может связать поле ввода строки с переменной модели, где хранится значение этой строки. Когда строка поменяется, в поле ввода автоматически отобразится новое значение, а когда пользователь сам поменяет значение в поле ввода - автоматически изменится значение переменной в модели. Такая связка обычно реализуется с помощью реактивных фреймворков, например RxSwift или ReactiveSwift. Эти фреймворки позволяют создавать так называемые сигналы, которые вызывают заранее определенные действия при срабатывании.
Binding :

```
// Bind the ‘name‘ property of ‘person‘ to the text value of an ‘UILabel‘.
// You’ll ReactiveCocoa and ReactiveSwift as well to make use of the <~ operator.
nameLabel.reactive.text <~ person.name
```

Загрузка данных из сети:

```
let searchResults = searchStrings
    .flatMap(.latest) { (query: String) -> SignalProducer<(Data, URLResponse),
AnyError> in
        let request = self.makeSearchRequest(escapedQuery: query)

        return URLSession.shared.reactive
            .data(with: request)
            .retry(upTo: 2)
            .flatMapError { error in
                print("Network error occurred: \(error)")
                return SignalProducer.empty
        }
    }
    .map { (data, response) -> [SearchResult] in
        let string = String(data: data, encoding: .utf8)!
        return self.searchResults(fromJSONString: string)
    }
    .observe(on: UIScheduler())
```

Сигналы можно объединять, фильтровать, передавать, используя приемы функционального программирования, делая код более декларативным, фокусирующимся на описании нужного результата. Этот подход может быть удачным для команд, в которых разработчики хорошо подготовлены к работе с реактивным кодом.

Какие еще есть нюансы при работе с данной архитектурой:

- В этой архитектуре может быть непросто изолировать логику представления от логики приложения. Если в вашем приложении сложная логика обработки данных, то ее смешение с логикой работы интерфейса может привести к жесткости и нарастающей сложности кода. И тогда чем дальше, тем сложнее будет разделить бизнес-логику от логики UX.

- Второй минус: при использовании реактивных фреймворков для байндинга часто возникает тенденция, при которой реактивный подход постепенно начнет использоваться повсеместно в

приложении, со временем запутывая и усложняя поддержку такого кода. А отладка реактивного кода действительно сложна. Часто очень непросто понять, почему тот или иной сигнал не сработал или сработал тогда, когда это не ожидалось.

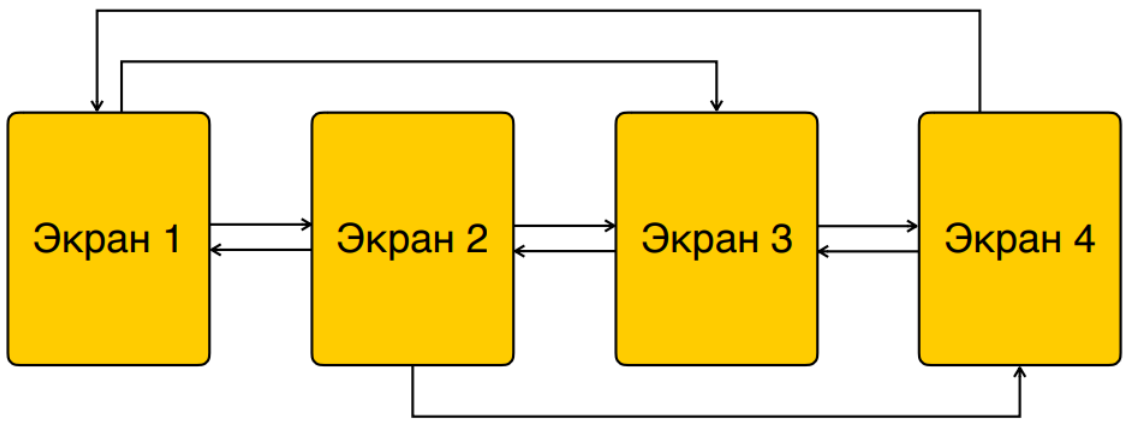
- Ну а если же ваша команда к реактивному программированию не готова, внедрение MVVM может ввести дополнительные расходы на обучение разработчиков. Если же отказаться от реактивного кода, то связки view и вью-моделей придется писать вручную, это тоже нужно учитывать.

Так или иначе, архитектура эта интересная и современная. Причем в открытом доступе находится масса примеров реализации MVVM для iOS. Поэтому предлагаю вам сейчас самостоятельно найти их и поизучать код. Также вам будет полезно немного ознакомиться с основными принципами функционального и реактивного программирования, например, самыми базовыми основами RxSwift, но это выходит за рамки данного курса, поэтому остается на ваше усмотрение. А сейчас главное - просто понять, что такое MVVM, и изучить пару простых примеров.
В помощь вам могу порекомендовать вот эти статьи:

- https://medium.com/flawless-app-stories/how-to-use-a-model-view-viewmodel-architecture-for-ios-46963c67be1b
- https://medium.com/@azamsharp/mvvm-in-ios-from-net-perspective-580eb7f4f129
- https://benoitpasquier.com/ios-swift-mvvm-pattern
- https://dev.to/eleazar0425/mvvm-pattern-sample-in-swiftios–58aj

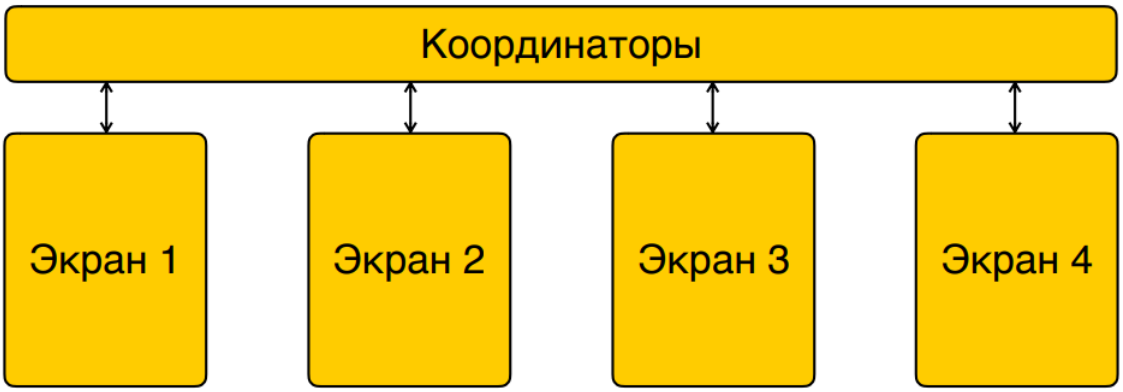
Итак, мы с вами узнали про архитектуру MVVM. Далее мы рассмотрим еще один интересный подход к тому, как можно улучшить стандартный MVC.
В прошлых видео мы с вами поговорили про то, как презентеры и вью-модели могут помочь изолировать логику UX. Теперь поговорим о том, какая еще логика может быть изолирована, чтобы еще лучше разделить между классами ответственность за разнородные вещи.

Почти каждое приложение содержит в себе множество экранов и некую общую центральную логику того, как пользователь перемещается между этими экранами.

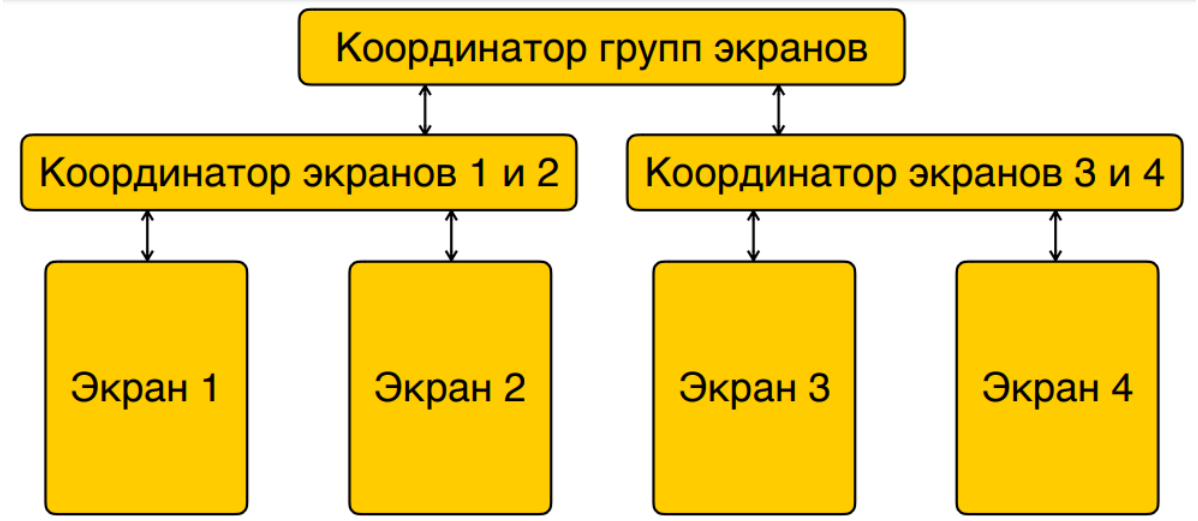


Вместе с пользователем, между экранами перемещаются и данные. Они нужны, чтобы на этих экранах что-то отобразить. То есть, существует некая логика - код - который описывает, какой экран должен показаться и какие данные нужно этому экрану передать. Кроме того, данные, сформированные на одном экране, зачастую нужно отправить на обработку перед тем, как на следующем

экране можно будет показать уже обработанные или расширенные данные.
Один из подходов, который снимает с вью-контроллеров, презентеров, вью-моделей ответственность за принятие таких решений (то есть решений на уровне всего приложения) - это слой координаторов.

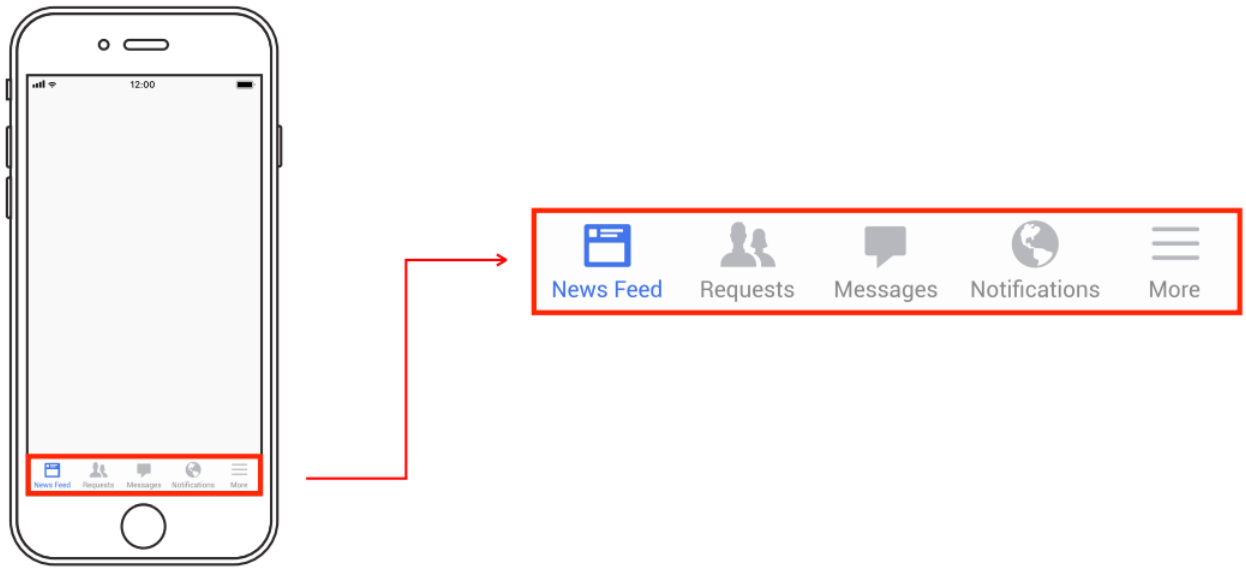


Координаторы - это объекты, которые могут принимать логические решения, хранить и передавать данные от экрана к экрану. То есть они выделяют общую для нескольких экранов логическую часть. Они знают, как устроена высокоуровневая структура приложения и из каких экранов оно состоит. Они как бы дирижируют классами, отвечающими за конкретные экраны. Могут создавать объекты этих экранов, подготавливать для них данные, а также отвечать за показ и скрытие этих экранов. Часто такие координаторы заменяют собой сигвеи в сториборде, то есть стрелки переходов между экранами.
Как правило, совокупность координаторов приложения представляет собой ДЕРЕВО, совпадающее с основными путями пользователя в приложении.



То есть, например, корневой координатор определяет происходящее на старте приложения - показать экран регистрации или основной рабочий экран приложения. Соответственно, дочерними координаторами будут - координатор регистрации и основной рабочий координатор. А основной рабочий координатор, в свою очередь, будет иметь дочерние координаторы, соответсвующие основным функциям приложения, как правило, соответствующие группам экранов в приложении.

Например, во многих популярных приложениях есть так называемый таб-бар внизу экрана, разделяющий приложение на разделы. Если использовать подход с координаторами, то у каждого раздела будет свой координатор, отвечающий за логику работы этого раздела.



Как и в случае взаимодействия между вью-контроллером и презентером, которое мы рассмотрели ранее, хорошо, если координаторы общаются между собой не по прямым ссылкам, а по ссылкам на их протоколы.

```
protocol MainCoordinatorProtocol {
    func openAvailableTasksList(animated: Bool, completion: VoidClosure?)
    func openAvailableTasksMap(animated: Bool, completion: VoidClosure?)
    func openMyTasks(animated: Bool, completion: VoidClosure?)
    func openActiveTasks(animated: Bool, completion: VoidClosure?)
    func openProfile(completion: VoidClosure?)
    func openMessages(completion: VoidClosure?)
    func openMore(completion: VoidClosure?)
}
```

Таким образом, координаторы не будут зависеть друг от друга, а будут лишь зависеть от перечня функций, которые им нужны.
Также, благодаря протоколам, координаторы могут легко протестированы с помощью юнит-тестов, потому что их можно будет подменить тестовым объектом, поддерживающим этот протокол, и возвращающим заранее предопределенные данные, чтобы результаты тестов координаторов были изолированы от других классов.
Итак, координаторы могут помогать абстрагировать общую логику приложения из вью-контроллеров и презентеров, более четко разделяя ответственность между классами, облегчая параллельную разработку, тестирование и понимание кода. Они добавляют в код приложения стройность и понятность - а значит, и надежность при развитии приложения.
Что характерно, координаторы могут успешно применяться к любой архитектуре: MVC, MVP, MVVM и тд. Более подробно про координаторы в iOS можно почитать в статьях в интернете, две из которых я рекомендую для ознакомления:

- https://readyset.build/ios-app-architecture-coordinators-8faade1763cf

- https://medium.com/sudo-by-icalia-labs/ios-architecture-mvvm-c-coordinators-3-6-3960ad9a6d85

Подумайте также, какими были бы координаторы в вашем приложении и какая у них была бы иерархия?

В следующем разделе мы немного отвлечемся от конкретных видов архитектур и поговорим про принципы общего характера, которые помогут сделать чище взаимодействие всех слоев приложения, а не только view и моделей.

4.3. Чистая архитектура

Во всех предыдущих видео мы с вами обсуждали так называемые MV-архитектуры. Все они рассматривают доставку данных из моделей на экран и обновление моделей по результатам взаимодействия пользователя с экраном. Также мы рассмотрели возможность использования координаторов, как инструмент передачи управления между экранами и частями приложения.
Но, возможно, у вас возник вопрос - а есть ли какие-то архитектурные принципы, которые можно применять к любому слою приложения? Есть ли какие-то правила, соблюдая которые мы можем обеспечить, что наша архитектура на всех уровнях не будет хрупкой, вязкой, неподвижной? Что работа сможет быть удобно распараллелена, функциональность - безболезненно расширяться, а код - легко тестироваться?
И такие принципы существуют. Один из самых известных на данный момент наборов таких принципов носит название SOLID. Впервые их сформулировал Роберт Мартин в 2000 году под названием Первые Пять Принципов. Впоследствии они получили название SOLID - по первым буквам этих пяти принципов:

- S - Single Responsibility Principle
- O - Open-Closed Principle
- L - Liskov Substitution Principle
- I - Interface Segregation Principle
- D - Dependency Inversion Principle

Давайте рассмотрим их подробно.
Первый принцип - Single Responsibility Principle. Принцип единственной ответственности.
Заключается в том, что класс должен иметь одну и только одну причину для изменений. Например, класс, отвечающий за представление, должен меняться, только если меняется дизайн. Класс, отвечающий за алгоритм обработки данных, должен меняться, только если изменилась бизнес-логика соответствующей функции приложения. И так далее. Таким образом уменьшается зависимость разнородных задач друг от друга, код безопасней менять, легче проверять и тестировать. Естественно, что классы не должны делиться на более мелкие классы бессистемно. Ваша архитектура должна выделять осмысленные абстракции в приложении и, исходя из этого, последовательно разделять код по классам. Пример такого разделения: View, ViewController-ы, Презентеры, Координаторы, Менеджеры данных, Вспомогательные сервисы и так далее.

- Выделение осмысленных абстракций в классы.

- Разделение кода на классы по сферам ответственности и по причинам для будущих изменений.
- Уменьшение сложности и запутанности кода.

Второй принцип - Open-Closed Principle. Принцип открытости/закрытости.
Класс должен быть по возможности открыт для расширения, но закрыт для изменения. Соответственно, вы должны иметь возможность расширять функциональность класса, не меняя код, который в нем уже был написан. Почему это важно? В работе мы постоянно добавляем какие-то новые возможности в приложение. Но если для добавления новой возможности нужно переделывать существующий код - это увеличение риска сломать существующие возможности, а также обязательная необходимость заново тестировать приложение. То есть, если бы мы хотели, например, достроить к дому чердак, а для этого нужно было бы перестроить еще и первый этаж или фундамент - такая пристройка была бы очень дорогостоящей.
Как можно добиться реализации этого принципа? На первый взгляд для расширения функциональности хорошо подходит наследование, однако нужно понимать, что наследование сильно связывает классы и они начинают зависеть друг от друга: наследник зависит от базового класса буквально, а базовый класс начинает зависеть от наследников в том плане, что, меняя базовый класс, нужно помнить о наследниках, чтобы не навредить им. Более того, в Swift'e, как и в objective c, нет множественного наследования, его можно только частично реализовать с помощью протоколов, имеющих реализации методов по умолчанию, однако повсеместное использование такого подхода может ухудшить понятность кода.
Более современный подход - использовать композицию, то есть включение объектов друг в друга, добавляя функциональность, но сохраняя независимость. То есть в класс, который реализует некий набор функциональности, мы добавляем переменные классов, которые умеют выполнять соответствующие функции. Таким образом, для добавления новой возможности нам фактически не придется менять код класса, а лишь передать в него новый объект и использовать его. Таким образом мы также повышаем возможности переиспользования нашего кода.

- Возможность развивать функциональность, не переделывая существующий код.
- Правильное переиспользование кода: композиция либо наследование (осторожно!).
- Безопасность изменений – определение точек роста приложения, отделение таких мест от неменяющегося кода.

Третий принцип - Liskov Substitution Principle. Назван по имени Барбары Лисков, которая сформулировала принцип подстановки, то есть возможности замещения базового типа его подтипом.
Принцип заключается в том, что функции, которые используют объект некого базового типа, должны иметь возможность использовать и объекты подтипов этого базового типа, не зная об этом. Это означает, что для них должно быть совершенно безопасно работать с любым наследником объекта, который они используют.
Более простыми словами можно сказать, что поведение наследуемых классов не должно противоречить поведению, заданному базовым классом. То есть поведение наследуемых классов должно быть ожидаемым для кода, использующего переменную базового типа. Принцип, вроде бы, очевидный, однако могут быть неочевидные случаи, когда он нарушается. Например, если дочерние классы генерируют исключения, которые не ожидаются от базового класса, либо подклассы ведут себя несвойственным для базового класса образом.
Классическим примером такой ситуации является наследование класса квадрата от класса прямоугольника. Очевидно, что в квадрате длина и ширина не могут отличаться, а в прямоугольнике - могут. Поэтому, если обращаться с квадратом как с прямоугольником - сначала инициализировать его длину, а потом - отличную от нее ширину,- то объект квадрата будет некорректным. Однако код, который работает с этим объектом, может даже не знать, что этот объект - квадрат, и ожидать от

него корректной работы. Таким образом, принцип Liskov Substitution призывает быть аккуратным с построением иерархии наследования в вашем приложении.

- Наследники должны строго соответствовать контракту с базовым классом.
- Наследники не могут противоречить ожидаемому поведению базового класса.
- Код может безопасно работать с любым наследником, не зная об этом (через интерфейс базового класса).

Четвертый принцип - Interface Segregation Principle. Разделение интерфейсов.
Простой и понятный принцип, который предлагает разделять большие интерфейсы на более мелкие, разбивая методы по группам, нужным разным клиентам этих интерфейсов. Говоря 'интерфейс', для Swift'а я имею в виду объявленный протокол - набор методов, который могут реализовать классы. Так вот, например, интерфейс презентера может быть разделен минимум на две группы: группа функций, отвечающих за реакцию на действия пользователя, и группа функций, отдающих представлению подготовленные данные.
То есть, как минимум, есть смысл разделять интерфейс команд от интерфейса запросов данных. Кроме того, команды и запросы тоже могут разделяться по группам, если клиенты используют только часть функций и им не нужно давать лишние возможности. Это добавляет безопасности в разработку, позволяет на этапе компиляции убедиться, что клиенты не вызывают методы, которые не должны вызывать. Также это дополняет принцип единственной ответственности, только применяя его уже не к классам, а к интерфейсам классов, то есть к группам возможностей, которые они представляют своим клиентам.
Пример реализации этого принципа, который вы можете вспомнить из других лекций этого курса - это делегаты класса UITableView: dataSource и delegate. То есть в UITableView реализация того, что отображает таблица, отделена от реализации того, как таблица это отображает.

- Клиенты класса должны как можно меньше зависеть от того, что им не нужно.
- Выделение ролей класса по видам его использования.
- Разделение интерфейса класса на несколько интерфейсов по выделенным ролям.

И пятый принцип - Dependency Inversion Principle. Инвертирование зависимостей.
Принцип очень важный и означает, что класс должен зависеть не от конкретной реализации, а от абстракции. По сути - от интерфейса, который скрывает реализацию и позволяет любому классу быть этой реализацией. То есть мы должны стараться связывать классы через ссылки на их интерфейсы, а не через ссылки на сами классы. Это особенно важно для случаев, когда классы должны быть взаимозаменяемыми.
Рассмотрим, например, случай, когда нам нужно реализовать чтение данных из базы. Есть некий класс нашей бизнес-логики, обрабатывающий данные и передающий их презентеру. Данные он должен вычитать из локального хранилища. Как вы уже знаете, сохранить данные на устройстве можно по-разному - можно использовать CoreData, можно сохранять их в простой файл и так далее. По сути, классу бизнес-логики не важно, какое хранилище используется, ему лишь важно получить нужный объект по его идентификатору. Возможно, в будущем понадобится оптимизировать хранение данных и заменить вид хранилища, но класс бизнес-логики не должен от этого пострадать. Ведь если его придется менять, это добавит дополнительный риск поломать алгоритм обработки данных.
Как мы можем в данном случае решить эту проблему? Класс бизнес-логики не должен иметь ссылку на конкретный инструмент, работающий с определенным хранилищем. Он должен иметь ссылку на интерфейс (то есть - на протокол), который умеет отдавать нужные нам объекты.
А вот за то, какой класс по факту будет реализовывать эту функциональность, будет отвечать создатель объектов, который знает, какое же хранилище нужно использовать. Этот принцип существенно

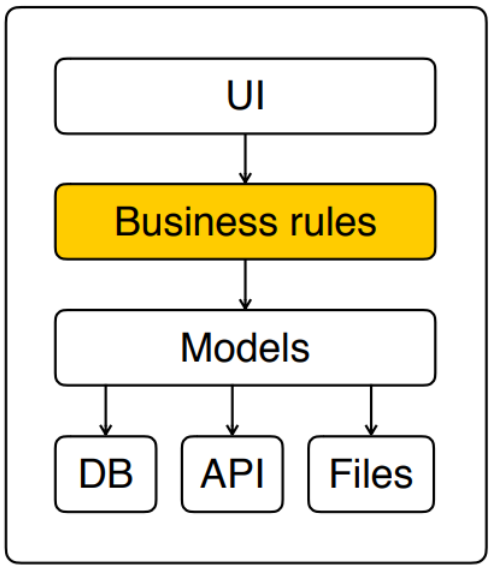
облегчает работу с долгосрочными проектами, в которых время от времени происходят замены вспомогательных механизмов - библиотек доступа в сеть, баз данных, систем сбора системных событий и так далее.

- Класс должен зависеть не от конкретной реализации, а от абстракции.
- Параметром класса/функции становится не ссылка на конкретный класс, а ссылка на интерфейс (протокол).
- Актуально для зависимостей, которые могут со временем меняться.
- Уменьшает число изменений во время развития функциональности.
- Упрощает юнит-тестирование (реальные зависимости могут быть заменены тестовыми заглушками).

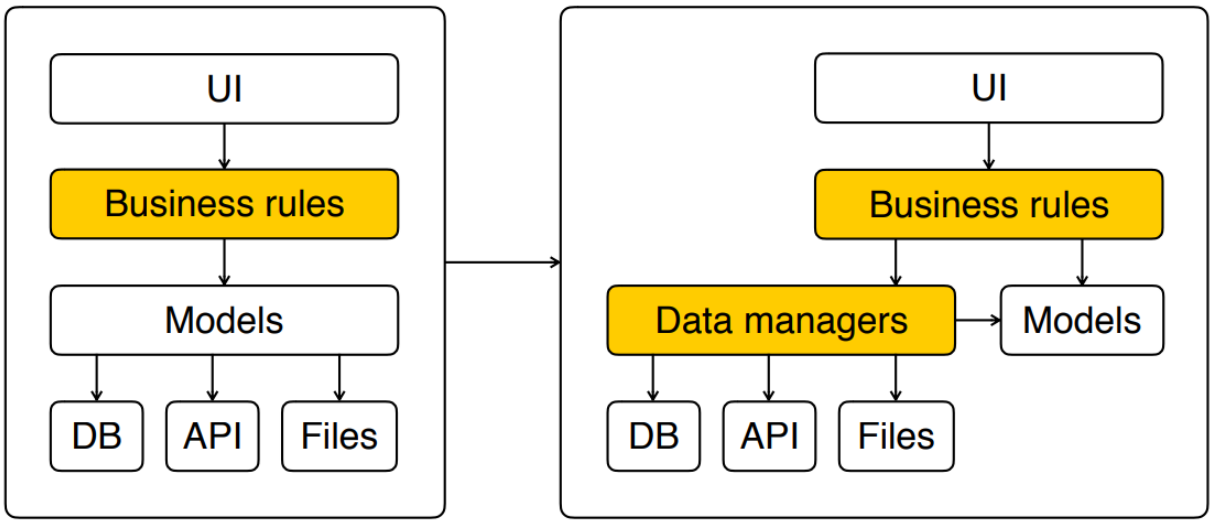
Итак, мы с вами рассмотрели принипы SOLID: Single Responsibility, Open-Closed, Liskov Substitution, Interface Segregation и Dependency Inversion.
Эти принципы лежат в основе так называемой Чистой архитектуры, о которой мы поговорим далее.
В прошлом видео мы с вами обсудили принципы SOLID, а теперь обсудим как их применять.
На самом деле, применять эти принципы можно к любой архитектуре. При этом, чем больше в архитектуре слоев, тем легче эти принципы реализовать на практике.

- Поскольку слои разделяют наш код на соответсвующие классы, мы можем добиться выполнения принципа единственной ответственности.
- Объявляя для этих классов протоколы, с помощью которых другие классы могут ими пользоваться, мы можем связывать классы друг с другом не по прямой ссылке на класс, а по ссылке на протокол. Таким образом мы выполним принцип инверсии зависимости, и классы будут зависеть не друг от друга, а от абстракции - набора возможностей, которые протокол предоставляет.
- Для классов, которые предоставляют разную функциональность для разных классов, мы можем разбить протокол на несколько более мелких, чтобы предоставлять другим классам только те возможности, которые нужны именно им. Это принцип разделения интерфейсов.
- Принцип open-closed заставляет задумываться о том, чтобы из-за новой функциональности не приходилось переписывать существующий код. В частности, можно инкапсулировать новую функциональность в отдельный класс и передавать в нужное место как ссылку на протокол, то есть ссылку на произвольный объект, который этот протокол реализует.
- Что касается принципа замещения Лисков - он обязательно должен применяться во всех архитектурах - нужно быть внимательным при создании иерархии классов. Дочерние классы всегда должны быть совместимы с базовыми.

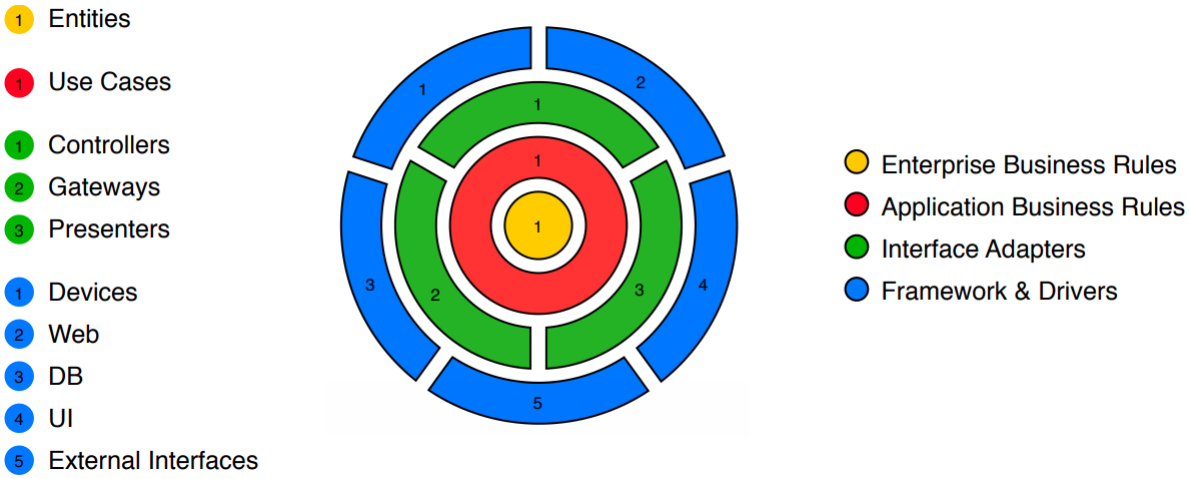
На основе принципов SOLID в 2012 году Роберт Мартин сформулировал архитектурный подход, который называется Чистая архитектура (Clean Architecture). Он применим к разработке под любую платформу - web, desktop, мобильные приложения. Этот подход помогает грамотно спроектировать архитектуру приложения: понять, какие сущности должны зависеть от других, а какие - нет.
Но для начала давайте рассмотрим типичную архитектурную проблему, которая часто встречается в коде приложений. Это зависимость слоя моделей от слоя ввода-вывода и сохранения данных.



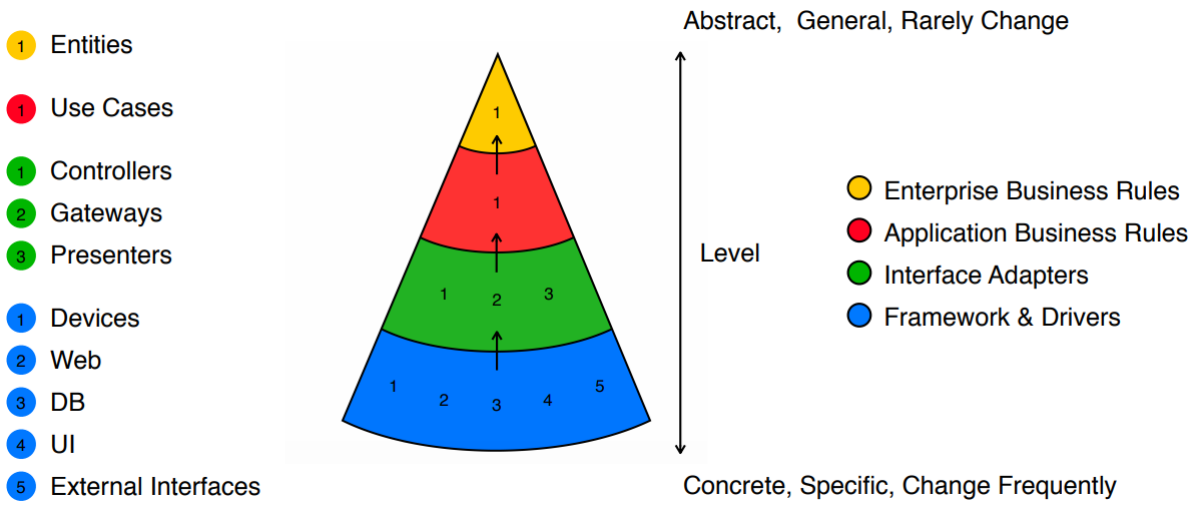
То есть модели данных зачастую напрямую базируются на том, как именно данные вычитываются из сети или локального хранилища. При таком подходе изменение формата получения данных сразу же означает необходимость менять сами модели. А дальше могут поменяться и остальные слои, по цепочке зависящие от моделей.
Решение этой проблемы - добавление конвертеров, которые занимаются преобразованием данных из внешних источников в модели и обратно. На схеме этот подход будет выглядеть вот так:



При этом сами модели никак не зависят от этих источников, то есть просто не знают об их существовании, а значит и не будут изменяться, если какая-либо конвертация поменяется. Этот простой прием позволяет существенно повысить стабильность кода при изменениях.
Если развернуть эту схему таким образом, чтобы модели оказались в центре, а остальные слои расположились вокруг нее, то мы получим схему чистой архитектуры, предложенную Робертом Мартином.



Давайте рассмотрим эту схему. Мы видим, что она выглядит как своеобразная луковица, где ядром являются модели. Остальные сущности разбиты по назначениям и сгруппированы по слоям. Эта схема наглядно показывает, что главный принцип тут в том, что внутренние слои этой луковицы не зависят от внешних слоев. То есть, как правило, они даже не знают об их существовании.
Модели - это, как правило, простые структуры, которые ни от чего не наследуются. И они полностью независимы, то есть не имеют в своем коде упоминаний про что-либо из других слоев. Еще раз подчеркну, что такой подход защищает модели от изменений во внешних слоях, делает их стабильными и надежными.
Далее располагается слой так называемых юз-кейсов, то есть сценариев работы приложения - его бизнес-логики. Этот слой, естественно зависит от моделей, пользуется ими для обработки данных, но при этом не зависит от того, как конкретно эти данные загружены и как эти данные будут показаны на экране. Про это знают только внешние слои. То есть слой юз-кейсов защищен от изменений, которые происходят в слоях ввода-вывода.
Если рассмотреть эту архитектуру в виде сектора этого круга, то получается такая картина:



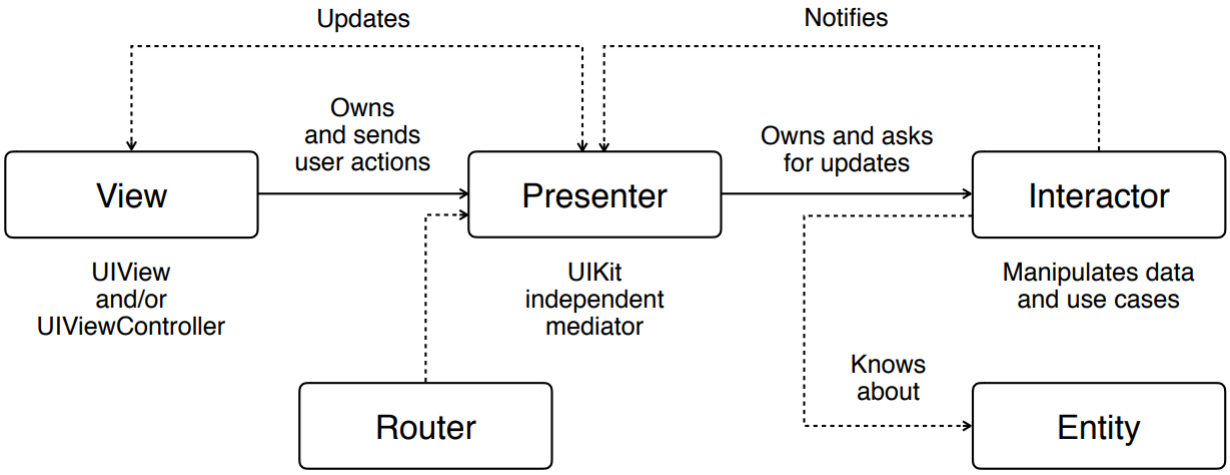
Стрелочками тут указаны направления зависимостей: от внешних слоев - к внутренним. Мы видим, что внутренние слои имеют более абстрактное, логическое представление и редко изменяются. В то

время как внешние слои - более специфичные и зависят от конкретных библиотек ввода-вывода. При этом части внешних слоев могут быть заменены на другие, и это будет безболезненно для внутренних слоев. Это значительно упрощает развитие приложения и улучшает его тестируемость.
Для того, чтобы такой подход можно было реализовать на практике, придется отказаться от некоторых приемов, которые уменьшают количество кода. Например, иногда для ускорения разработки прямо в моделях полям задают строковые идентификаторы, которые соответствуют названиям полей в базе данных и полям в структурах, присылаемых по сети. Также часто структура моделей полностью дублирует структуру ответов, присылаемых сервером. Это удобно ровно до того момента, когда структура сетевых ответов не начнет меняться, либо придется поддерживать сразу несколько версий сетевого API, либо перейти на использование базы данных с другим принципом хранения. В такие моменты (если модели жестко завязаны на формат их загрузки) может понадобится переписать значительную часть приложения, затрагивая и модели, и бизнес-логику, которая эти модели использует. Это приводит к задержкам в разработке, тестировании, риску появления ошибок в любом месте приложения.
Используя чистую архитектуру, этого можно избежать. Конечно, за надежность нужно будет заплатить некоторыми накладными расходами. Для каждого типа чтения или записи данных нужно написать свой конвертер данных. Даже если названия полей на данный момент совпадают в нескольких источниках - их нужно продублировать, чтобы они не зависели друг от друга.
Итак:

- В код структур моделей лучше не помещать названия полей API/БД.
- Для каждой версии API/БД лучше создать отдельный конвертер.
- Если код совпадает, но относится к независимым вещам – переиспользовать его нельзя.
- Важно разделить стабильные абстракции и изменяемые специфичные части кода.

В итоге очень важно понять следующее: переиспользовать можно только абстракции - то есть код, который написан в общем виде и не зависит от конкретики использования. Если же код просто совпадает, но принадлежит вещам, которые логически друг от друга не зависят - такой код выносить в некое общее место нельзя, даже если это кажется удобным на первый взгляд.
Итак, мы с вами рассмотрели подход к созданию архитектуры, который делает развитие приложения надежным и предсказуемым. Далее мы рассмотрим еще одну архитектуру, последнюю в нашем курсе, которая была создана специально для iOS и наиболее близка к чистой архитектуре по принципу своей работы.
Итак, мы с вами обсудили чистую архитектуру, где строгое применение принципов единственной ответственности и инверсии зависимостей позволяет, во-первых, хорошо декомпозировать код, а во-вторых, качественно изолировать модели и логический код от кода ввода-вывода и хранения данных.
В 2014 году Джеф Гилберт написал статью про архитектуру для iOS приложений, которая на тот момент стала наиболее полно соответствовать критериям чистой архитектуры. Она была названа VIPER. Как и MVC, MVP, MVVM, VIPER - это аббревиатура тех главных слоев, которые определяют ядро этой архитектуры:

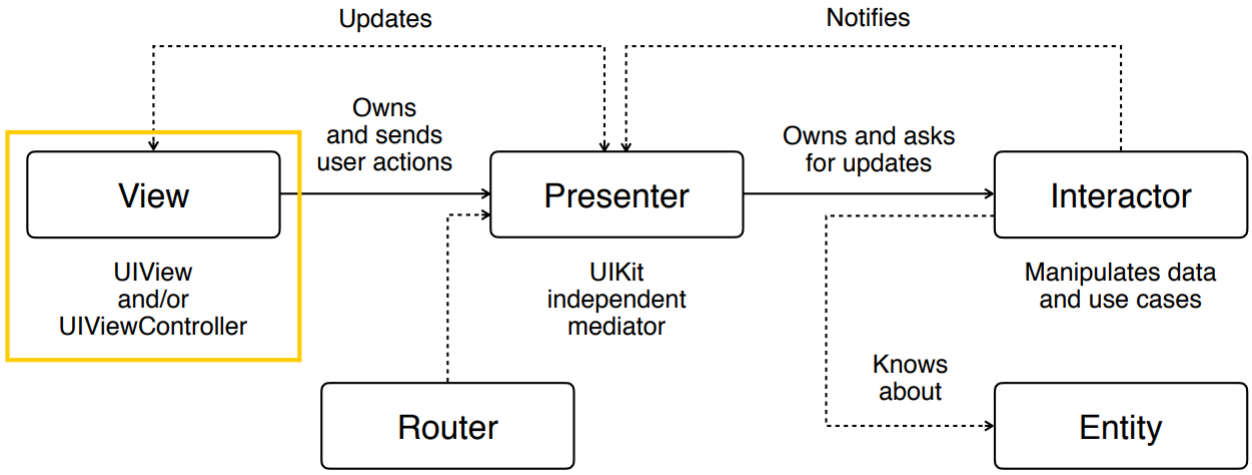
- V - View,
- I - Interactor,
- P - Presenter,
- E - Entity,
- R - Router.



Рассмотрим схему архитектуры. Если мы вспомним архитектуру MVP, то увидим тут сходство - слой презентеров. Но в MVP презентеры имеют довольно много ответственности - и по работе с данными, и по управлению происходящим на экране. В Вайпере же презентеры разбиты на три функциональных слоя:

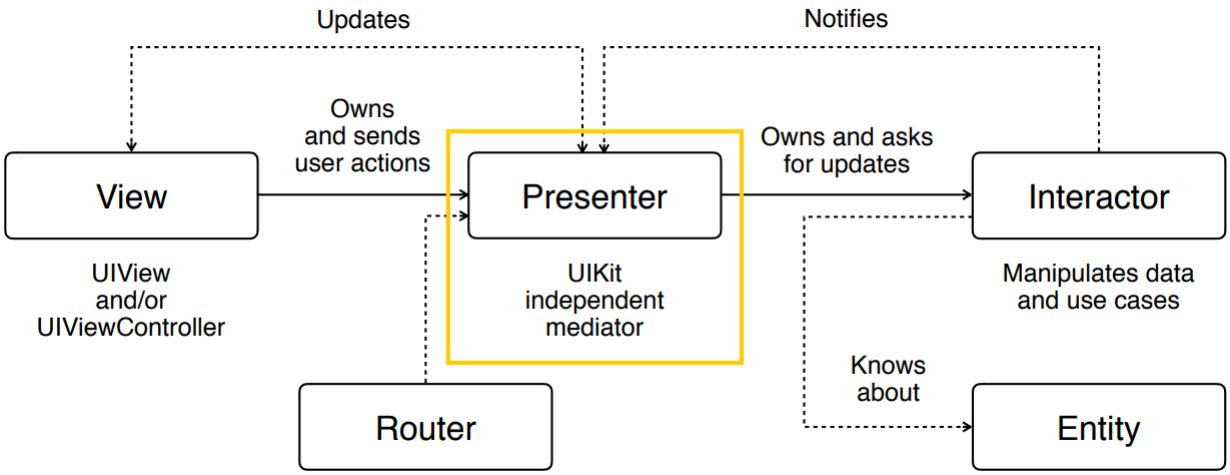
- первый - собственно презентеры, отвечающие за передачу данных на отрисовку во view и за получение сигналов от view на действия пользователя;
- второй - интерактор - слой бизнес-логики, принятия логических решений, обработки данных;
- третий - роутер - слой, отвечающий за логику смены экранов и передачи данных между ними.

Теперь давайте рассмотрим все части VIPER подробнее.
View.

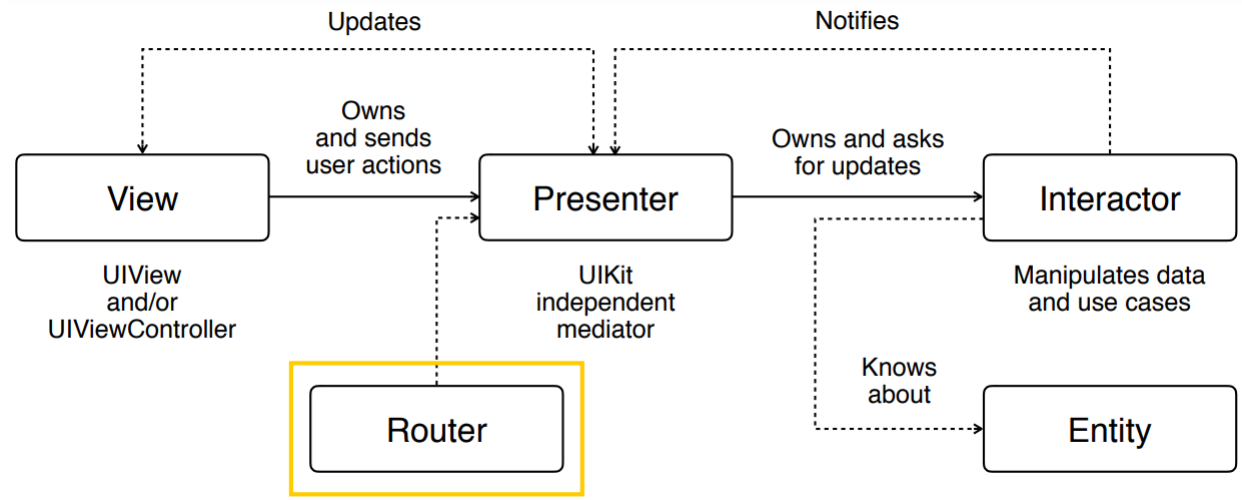


Как и в MVP, роль View играют ViewController-ы. В Вайпере View предельно пассивные, это означает, что View никогда не принимает никаких решений. За нее это делает презентер. Например, по нажатию на кнопку нужно запустить процесс загрузки данных и показать индикатор процесса - спиннер. Так

вот, вью-контроллер сам не решает, когда ему показывать и прятать спиннер. Вместо этого презентер отдает ему команды - покажи спиннер, спрячь спиннер. И когда отдать эти команды - решать только презентеру.

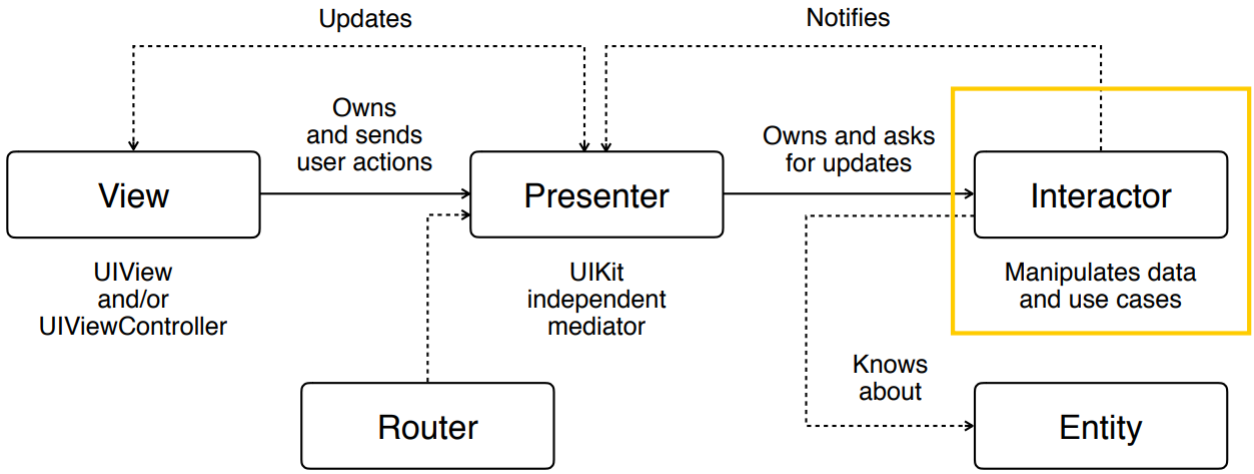


Презентер и view связаны друг с другом через протоколы. То есть у view есть свой вью-протокол, содержащий перечень команд для отрисовки интерфейса, которые оно может выполнять, а у презентера есть ссылка на этот протокол. У view, в свою очередь, есть ссылка на протокол презентера. Этот проткол содержит методы, которые view вызывает по действиям пользователя - например, по нажатой кнопке или другому действию.
Обратите внимание на стрелки: тут они означают направление владения, то есть сильных ссылок. Вью-контроллер владеет презентером, презентер - интерактором и т.д.
Роутер - это слой, управляющий поведением не конкретного экрана, а всего приложения, то есть переходами между экранами, созданием новых экранов и т.д.

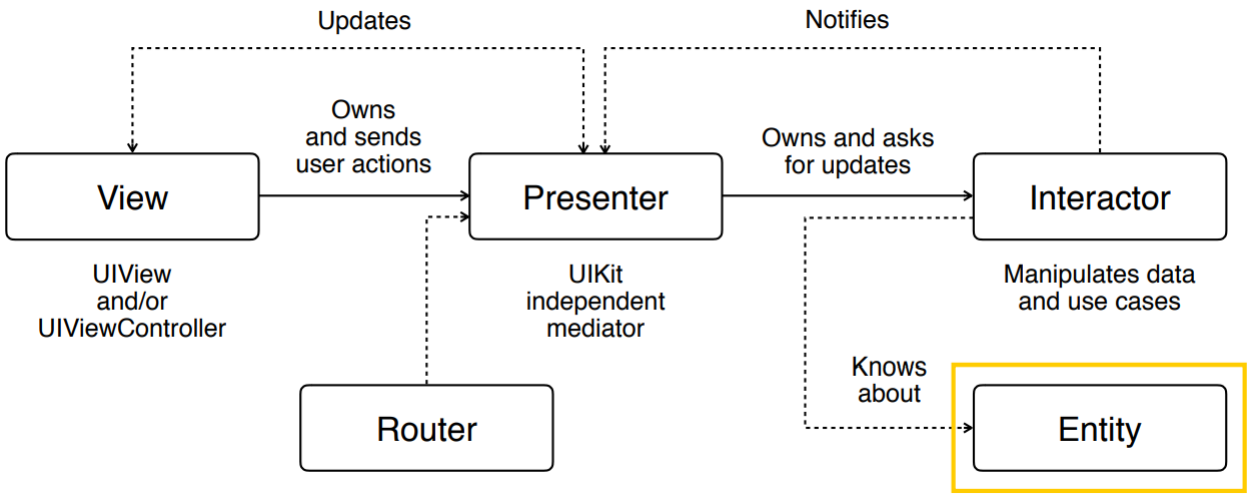


Роутеры могут выстраиваться в древовидную структуру подобно тому, как это делают координаторы, о которых мы говорили ранее. Но в отличие от координаторов роутеры не занимаются логическими

задачами, их роль более узкая - навигация между экранами и передача данных между ними. При использовании Вайпера разработчики зачастую отказываются от использования storyboard'а для реализации переходов между экранами, поскольку роутер с этой задачей справляется более гибко.
Несколько слов про интерактор.

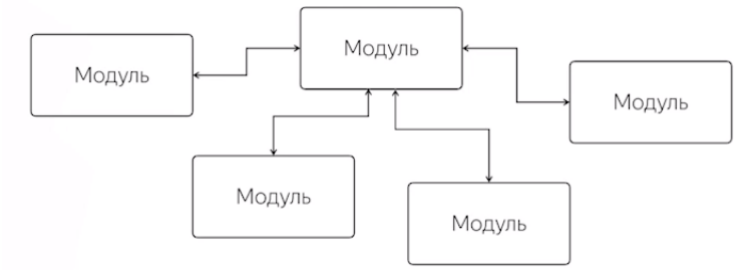


Это класс, где изолируется вся бизнес-логика для одного пользовательского сценария. Интерактор полностью скрывает от презентера то, откуда появляются данные. Презентер лишь просит интерактор дать ему данные, а интерактор дальше сам решает, откуда ему их брать - из базы данных, файлов, интернета и т.д. При этом интерактор, в свою очередь, ничего не знает про то, как эти данные будут использоваться, он ничего не знает про интерфейс приложения, экраны и их дизайн. То есть интерактор абстрактен. Если мы вспомним чистую архитектуру, то станет понятно, почему при приближении к моделям код теряет свою конкретику и специфику и описывает задачу логическими категориями.
Entity в Вайпере - это и есть модели.



Но модели - это не какой-то еще один слой, а всего лишь структуры или небольшие классы с наборами данных, с которыми работают интеракторы. Как правило, модели создаются в конвертерах данных,

которые вычитывают их из базы данных или интернета, потом интеракторы их анализируют, обрабатывают и в обработанном виде передают в презентеры. А там из них уже готовятся представления этих данных, которые могут быть показаны на экране. То есть, по сути, готовятся вью-модели.
В сложных приложениях такие вот наборы вайпер-классов могут формировать так называемые вайпер-модули, у которых есть свой интерфейс взаимодействия с другими модулями. Однако для небольших приложений такой подход может быть ненужным усложнением:



В целом нужно отметить, что Вайпер - неплохой выбор для крупных, долгосрочных приложений. У кода приложений, написанных при помощи Вайпера, отличная долгосрочная жизнеспособность. Однако поначалу такая разработка может показаться трудоемкой, ведь для каждого экрана нужно создавать гораздо больше классов. Некоторые разработчики даже используют генераторы кода для облегчения жизни. Однако плюсы от применения принципов SOLID и чистой архитектуры все же, как правило, оказываются весомее этих накладных расходов.
Более подробно про вайпер можно почитать в этих статьях, а также найти множество примеров использования:

- http://objc.io/issues/13-architecture/viper/
- http://medium.com/slalom-engineering/clean-architecture-for-ios-developmentusing-the-viper-pattern-fac30f5d29fc
- http://cheesecakelabs.com/blog/ios-project-architecture-using-viper/
- http://swifting.io/blog/2016/03/07/8-viper-to-be-or-not-to-be/
- http://habr.com/post/358412/

Подводя итог нашему обзору архитектурных подходов, хотелось бы отметить, что, несмотря на множество разных подходов, все они служат одной цели - облегчению разработки. Инвестиции в изучение различных архитектур, которые поначалу могут показаться ненужными или избыточно сложными, со временем окупаются. И чем сложнее проект, тем заметнее эффект от внедрения подходящего архитектурного подхода.
Выбор архитектуры - пожалуй, главный выбор разработчика при старте проекта. И чтобы не ошибиться в этом выборе, очень полезно ориентироваться в основных архитектурах, знать их плюсы и минусы.