GeekBrains

Архитектура ПО

Методичка к уроку 8

Архитектура прикладных приложений (мобильные): MVC, MVP, MVVM

Оглавление

Введение

На этом уроке

1. Архитектурные и часто используемые паттерны прикладных приложений
   1.1. Паттерны MVC-семейства
   1.2. MVC (Model-View-Controller)
      1.2.1. Традиционный MVC в WEB
      1.2.2. MVC от Apple
      1.2.3. Пример MVC на Java
   1.2. MVP (Model-View-Presenter)
   1.3. MVVM
   1.4. MVPVM (Model-View-Presenter-ViewModel)
   1.5. MVI
2. Чистая архитектура прикладных приложений
   2.1. Чистая архитектура в iOS-разработке VIPER
   2.2. Чистая архитектура в Android-разработке
3. Server Driven UI
4. Кроссплатформенность приложений

Глоссарий

Дополнительные материалы

Используемые источники

Введение

С ростом мощности компьютеров росла и сложность ПО. Появилась необходимость структурировать программы — выделить отдельные части, отвечающие за работу с данными, бизнес-логику и интерфейс с пользователем. Это все еще были части одной программы, однако они становились все более независимыми и универсальными. Развитие этой тенденции привело к появлению архитектуры «Клиент-сервер».

В десктопных приложениях выделяются части: сервер и данные, клиент с интерфейсом. В таких приложениях части логики обработки данных могут быть распределены между сервером и клиентом. На сервере, например, могут выполняться ресурсоёмкие расчётные задачи, а на клиенте — более простые функции и управление с помощью интерфейса.

В разработке веб-приложений сейчас существует три основных подхода:

1. одностраничные SPA (Single Page Application),
2. многостраничные MPA (Multi Page Application),
3. прогрессивные приложения (PWA).

MPA появился раньше других, поэтому имеет классическую архитектуру. Пример MPA-приложения — wikipedia.org, где каждый запрос пользователя приводит к загрузке и отображению новой HTML-страницы. Но такие приложения неэффективны, если при запросе пользователя остается та же страница, но на ней, допустим, появляется новая картинка: клиенту придется снова целиком загружать страницу с появившейся картинкой. MPA больше подходит для крупных интернет-магазинов, бизнес-сайтов, каталогов. Имеет высокую скорость и производительность.

Одностраничное приложение (SPA) — это веб-приложение или веб-сайт, использующий единственный HTML-документ как оболочку для всех веб-страниц и организующий взаимодействие с пользователем через динамически подгружаемые HTML, CSS, JavaScript, обычно посредством AJAX. SPA обеспечивают более качественное и удобное взаимодействие пользователя с приложением. Сейчас это большинство интерактивных приложений в сети: маркетплейсы, финансовые приложения, порталы доступа к данным.

На этом уроке

1. Рассмотрим паттерны проектирования прикладных приложений семейства MV*.
2. Изучим паттерны MVC, MVP, MVVM и их производные.
3. Рассмотрим чистую архитектуру iOS-приложения.
4. Рассмотрим чистую архитектуру Android-приложения.
5. Разберём подход Server driven UI.
6. Исследуем кроссплатформенную разработку.

1. Архитектурные и часто используемые паттерны прикладных приложений

1.1. Паттерны MVC-семейства

Паттерны семейства MV* — одни из фундаментальных, используемых при разработке приложения с графическим интерфейсом. Без них значительно усложняется процесс тестирования и теряется возможность переиспользовать части приложения.

Разработка на основе паттернов MV* позволяет разделить код на независимые модули, тем самым упростить поддержку готового продукта и ускорить разработку нового функционала. Изначально паттерны появились в веб-технологиях, но со временем нашли применение как в настольных, так и в мобильных приложениях.

Основное отличие между паттернами — способ связи элементов между собой.

Прежде чем перейдем к описанию особенностей каждого паттерна, рассмотрим общие черты всего семейства — M(Model) и V(View).

Модель (Model) — часть программы, содержащая описание функциональной и бизнес-логики приложения. Модель должна быть полностью независимой от остальных частей продукта. Модельный слой ничего не должен знать об элементах графического интерфейса и того, каким образом он будет отображаться.

Признаки модели:

- модель — это бизнес-логика приложения;

- модель обладает информацией о себе самой и не знает о контроллерах и представлениях;
- для некоторых проектов модель — это просто слой данных (DAO, база данных, XML-файл);
- для других проектов модель — это менеджер базы данных, набор объектов или просто логика приложения.

Представление (View) отвечает за отображение данных, полученных от модели, но не может напрямую влиять на модель. Предполагается, что представление обладает доступом к данным «только на чтение».

Признаки представления:

- в представлении реализуется отображение данных, которые получаются от модели некоторым способом;
- в некоторых случаях представление может иметь простейший код, реализующий методы для изменения состояний объектов бизнес-логики.

Примеры представления: HTML-страница, WPF-форма, Android Views, Windows Form.

По историческим причинам аббревиатурой MVC (Model View-Controller) принято называть не единственный паттерн, а целое семейство паттернов, призванное отделить представление от модели. Произошло это отчасти потому, что MVC не просто паттерн, а объемное архитектурное решение, в котором разработчики часто видели что-то свое и, ставя во главу угла особенности своего проекта, реализовывали его по своему усмотрению. А отчасти и из-за возраста паттерна: во времена его изобретения и сами приложения, и графические интерфейсы были существенно проще, чем в наше время. С тех пор они сильно эволюционировали, и вместе с ними изменился сам паттерн.

Вариации MVC-паттернов — MVC, MVP, MVPM, MVVM, MVPVM, MVI, VIPER, RIBS (их рассмотрим позже). Основные идеи глобального подхода:

- Разделение на модель и представление, а также различные вариации их связывания (контроллер, представитель, модель представления и другие).

- Модель — это сущности системы, бизнес-логика, организация хранения данных. То есть в той или иной степени модель может покрывать все слои архитектуры.
- Представление обычно рассматривается на четвёртом внешнем слое.
- Слой связывания (контроллер, представитель) обычно рассматривается на третьем слое.

Ключевой паттерн в подходе проектирования MVC — непосредственно сам шаблон (паттерн) **модель-представление-контроллер**.

В отличие от готовых функций и библиотек, паттерны нельзя скопировать в программу.

Паттерны часто путают с алгоритмами. Оба понятия описывают типовые решения известных проблем, но алгоритм — это чёткий набор действий, а паттерн — это высокоуровневое описание решения, реализация которого может отличаться в зависимости от поставленной задачи.

1.2. MVC (Model-View-Controller)

Концепция MVC была описана в 1978 году Трюгве Реенскаугом, работавшем в научно-исследовательском центре Xerox PARC над языком программирования Smalltalk. Позже Стив Бурбек реализовал шаблон в Smalltalk-80. Окончательная версия концепции MVC была опубликована лишь в 1988 году в журнале Technology Object.

Название паттерна MVC расшифровывается как модель-представление-контроллер (от англ. Model-View-Controller). Это способ организации кода, который предполагает выделение блоков, отвечающих за решение разных задач. Один блок отвечает за данные и бизнес-логику приложения, другой — за внешний вид и способ взаимодействия с пользователем, третий контролирует работу приложения, связывая предыдущие блоки.

Модель (Model) предоставляет данные и методы работы с ними: запросы в базу данных, проверку на корректность. Модель не зависит от представления и контроллера, организовывая доступ к данным, управлению ими и к бизнес-логике. Модель строится таким образом, чтобы отвечать на запросы, изменяя своё состояние. Модель, за счёт независимости от визуального представления, может иметь несколько различных своих представлений.

Представление (View) отвечает за взаимодействие с пользователем. То есть код представления определяет внешний вид приложения и способы его использования.

Контроллер (Controller) отвечает за связь между моделью и представлением. Код контроллера определяет, как приложение реагирует на действия пользователя. Это связывающий, контролирующий слой.

Признаки контроллера:

- контроллер определяет, как представление должно быть отображено в данный момент;
- события представления могут повлиять только на контроллер, в то время как контроллер может «сообщить» модели, как нужно изменить её представление.

Основная идея паттерна в том, что контроллер и представление зависят от модели, но модель никак не зависит от этих двух компонентов.

Реализация: контроллер получает событие извне (например, от пользователя или по сети) и, в соответствии с заложенной в него логикой, реагирует на это, сообщая об изменениях модели посредством вызова соответствующих методов. После изменения модель использует информацию о том, что нужно изменить, и отправляет эту информацию всем заинтересованным сущностям представления. Получив их, представление изменяется в случае необходимости.

Первая компонента/модуль — модель. Она содержит всю бизнес-логику приложения. Модель предоставляет данные и методы работы с ними: запросы в базу данных, проверку на корректность. Модель не зависит от представления (не знает, как данные визуализировать) и контроллера (не имеет точек взаимодействия с пользователем), просто предоставляя доступ к данным и управлению ими.

Модель строится таким образом, чтобы отвечать на запросы, изменяя своё состояние. При этом может быть встроено уведомление наблюдателей.

Вторая часть системы — представление. Отвечает за получение необходимых данных из модели и отправляет их пользователю. Представление не обрабатывает введенные данные пользователя.

Третья часть — контроллер. Он обеспечивает связь между пользователем и системой, контролирует и направляет данные от пользователя к системе и наоборот. Использует модель и представление для реализации необходимого действия.

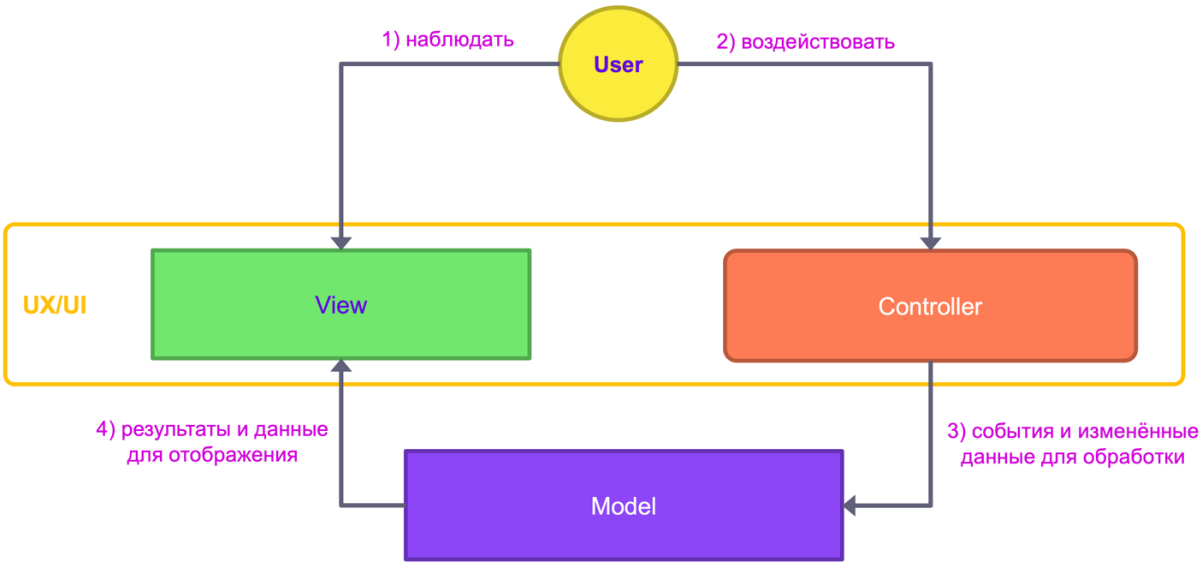


Схема традиционного MVC

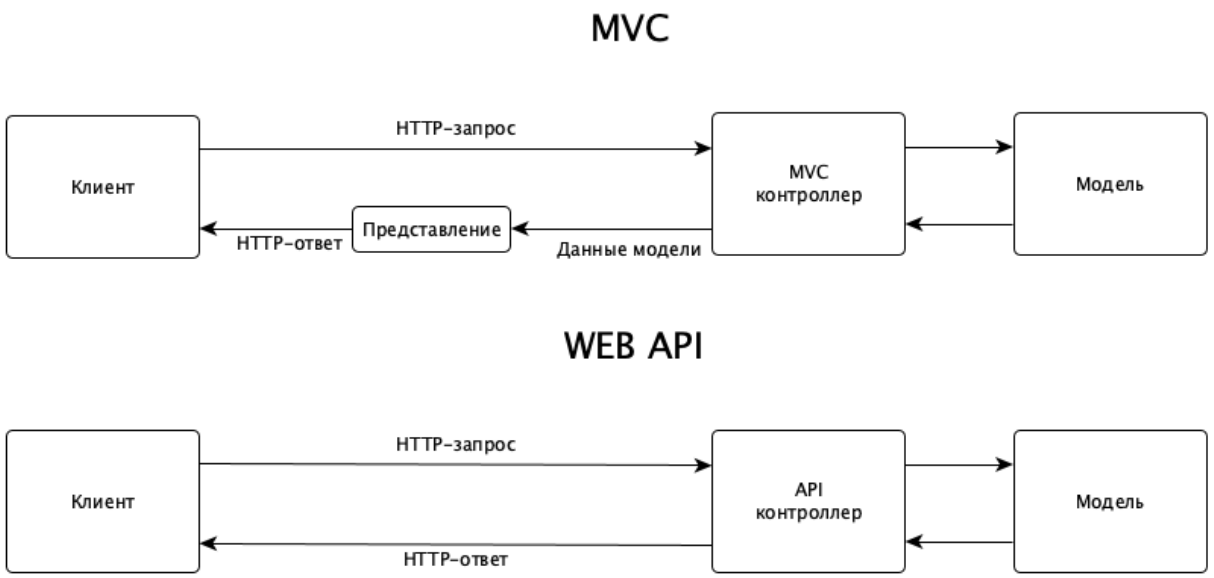
Основная цель применения концепции — в отделении бизнес-логики (модели) от её визуализации (представления). За счёт разделения повышается возможность повторного использования кода. Применение концепции наиболее полезно в случаях, когда пользователь должен видеть те же самые данные одновременно в различных контекстах и/или с различных точек зрения. В частности, решаются следующие задачи:

1. К одной модели можно подключить несколько представлений, при этом не затрагивая реализацию модели. Например, некоторые данные могут быть одновременно представлены в виде электронной таблицы, гистограммы и круговой диаграммы.

2. Не затрагивая реализацию представлений, можно изменить реакции на действия пользователя (нажатие на кнопку, ввод данных). Для этого достаточно подменить или модифицировать контроллер.

3. Ряд разработчиков специализируется только в одной из областей: либо разрабатывают графический интерфейс, либо бизнес-логику. Поэтому можно добиться того, что программисты, занимающиеся разработкой бизнес-логики (модели), вообще не будут осведомлены о том, какое представление будет использоваться, и как часто оно будет меняться.

На рисунке изображена схема работы приложения с использованием архитектурного паттерна MVC и его упрощенный вид для реализации API. В последнем случае контроллер выдает в качестве ответа на запрос просто сериализованные в одном из форматов данные (JSON, XML).



Паттерн MVC и его упрощенный вид для реализации API

При разработке систем с пользовательским интерфейсом, следуя паттерну MVC, нужно разделять систему на три части. Их, в свою очередь, можно называть **модулями** или **компонентами**.

У каждой составной компоненты будет свое предназначение [3]. Для его реализации можно воспользоваться фреймворками:

- **ASP.NET MVC Framework** — фреймворк для создания веб-приложений, который реализует шаблон Model-View-Controller.
- **Spring Framework** (коротко Spring) — универсальный фреймворк с открытым исходным кодом для Java-платформы.

- **Ruby on Rails** (RoR) — фреймворк, написанный на языке программирования Ruby. Реализует архитектурный шаблон Model-View-Controller для веб-приложений, а также обеспечивает их интеграцию с веб-сервером и сервером баз данных.
- **Django** — свободный фреймворк для веб-приложений на языке Python, использующий шаблон проектирования MVC.

1.2.1. Традиционный MVC в WEB

В традиционном MVC View не хранит состояния в себе. Controller просто рендерит View при изменениях Model. Например, веб-страница полностью перезагружается после того, как пользователь нажмет на ссылку для перехода в другое место.

Достоинства классической MVC:

1. К одной модели можно присоединить несколько представлений так, чтобы не менять код модели. Например, если табличные данные хочется показать и таблицей, и гистограммой, и пай-чартом.
2. Можно изменить реакцию на действия пользователя, не затрагивая реализацию представлений, просто подставив другой контроллер. Например, если разработчик хочет, чтобы по кнопке данные не отправлялись в БД, а сохранялись локально, ему достаточно заменить контроллер, не трогая при этом ни представление, ни модель.
3. Разработка бизнес-логики не зависит от разработки представлений и наоборот. Это значит, что несколько человек могут одновременно работать над одной задачей.
4. Однонаправленный поток данных (представление → контроллер → модель → представление) делает проще отладку и тестирование.

Традиционный MVC — это классический способ реализации MPA-приложений, но он кажется неприменимым к современной разработке мобильных приложений. В мобильной разработке он может трактоваться большим количеством способов, и это не всегда хорошо.

1.2.2. MVC от Apple

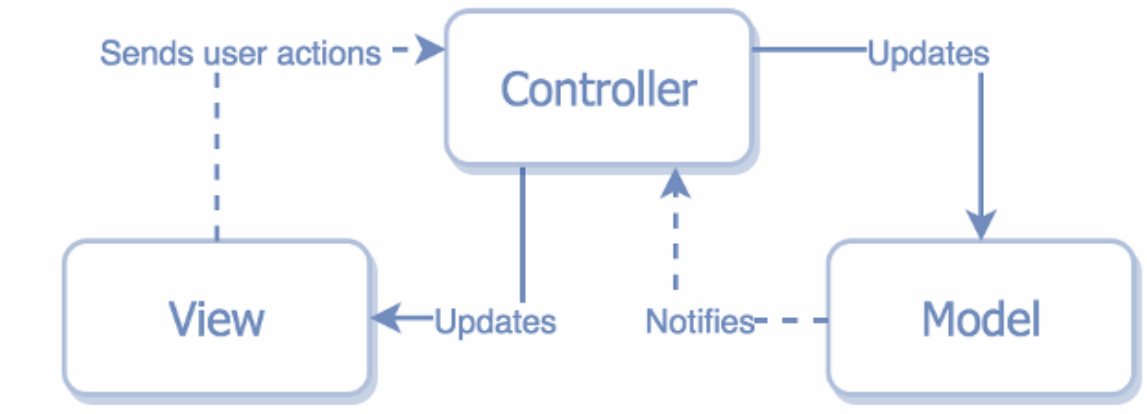


Схема Cocoa MVC в теории

Apple предлагают реализацию Cocoa MVC. Шаблон проектирования модель-представление-контроллер (MVC) назначает объектам в приложении одну из трех ролей: модель, представление или контроллер.

Шаблон определяет не только роли, которые объекты играют в приложении, но и способ взаимодействия объектов друг с другом. Каждый из трех типов объектов отделен от других абстрактными границами и взаимодействует с объектами других типов через эти границы. Коллекцию объектов определенного типа MVC в приложении иногда называют слоем — например, слоем модели.

MVC играет центральную роль в хорошем дизайне приложения Cocoa. Преимущества использования этого шаблона многочисленны. У многих объектов в этих приложениях есть тенденция к повторному использованию, а их интерфейсы, как правило, лучше определены. Приложения с дизайном MVC легче расширяются, чем другие приложения. Более того, многие технологии и архитектуры Cocoa основаны на MVC и требуют, чтобы настраиваемые объекты играли одну из ролей MVC.

Controller — это посредник между View и Model, следовательно, последние не знают о существовании друг друга. Поэтому Controller трудно повторно использовать, но появляется место для той хитрой бизнес-логики, которая не вписывается в Model.

Однако на практике Cocoa MVC выглядит следующим образом:

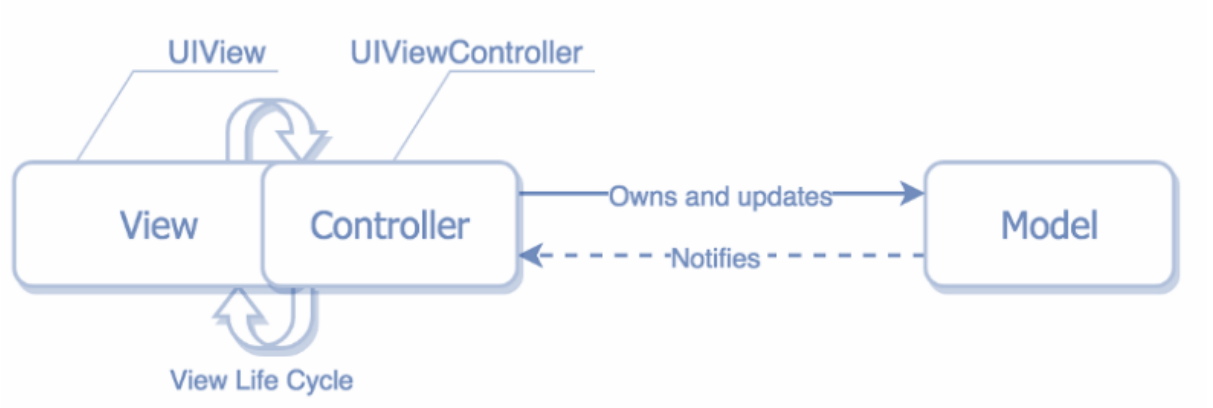


Схема Cocoa MVC на практике

Cocoa MVC поощряет вас писать Massive-View-Controller, потому что Controller настолько вовлечен в жизненный цикл View, что трудно сказать, что он является отдельной сущностью. Хотя у вас все еще есть возможность отгрузить часть бизнес-логики и преобразования данных в Model, когда дело доходит до отгрузки работы во View, у вас не так много вариантов. В большинстве случаев вся ответственность View состоит в том, чтобы отправить действия к контроллеру. В итоге все заканчивается тем, что View Controller становится делегатом и источником данных, а также местом запуска и отмены серверных запросов и, в общем-то, всего чего угодно.

Cocoa MVC обосновано расшифровывают как **Massive-View-Controller**.

Проблема не очевидна, пока дело не доходит до юнит-тестов. Так как View Controller тесно связана с View, его становится трудно тестировать, приходится идти изощренным путем, заменяя View Mock-объектами и имитируя их жизненный цикл, а также писать код View Controller таким образом, чтобы бизнес-логика была по максимуму отделена от кода View Layout.

Протестировать взаимодействие между View и ViewController — задача настолько нетривиальная, что в MVC-приложениях, как правило, тестируют только модель и network-слой (что во многих случаях бывает так же сложно разделить, как и UIView с UIViewController: представьте модель данных и интерфейс доступа к данным, например, к БД).

После всего сказанного может показаться, что Cocoa MVC — плохой выбор паттерна. Но давайте оценим его с точки зрения признаков хорошей архитектуры:

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

Cocoa MVC — это разумный выбор, если вы не готовы инвестировать много времени в свою архитектуру и чувствуете, что паттерн с более высокой стоимостью обслуживания вашему небольшому проекту или стартапу будет не по карману.

Cocoa MVC — лучший архитектурный паттерн с точки зрения скорости разработки.

1.2.3. Пример MVC на Java

controller.java

```
public class Controller {
   private final Model model;
   private final View view;


   public Controller(Model model, View view) {
      this.model = model;
      this.view = view;
   }


   public void insertStudent(Student student) {
      model.insertStudent(student);
```

```
    }

    public List<Student> getAllStudent() {
        return model.getAllStudents();
    }

    public void updateStudentById(int id, Student student) {
        model.updateStudentById(id, student);
    }

    public void insertStudents(List<Student> studentList) {
        model.insertCache(studentList);
    }

    public void updateStudents(List<Student> studentList) {
        model.updateCache(studentList);
    }

    public void updateView() {
        view.printStudents(model.getAllStudents());
    }
}
```

model.java

```
public class Model {
    private final HashMap<Integer, Student> cache = new HashMap<>();

    public void insertCache(List<Student> studentList) {
        for (Student student : studentList) {
            cache.put(student.getId(), student);
        }
    }

    public void insertStudent(Student student) {
        cache.put(student.getId(), student);
    }

    public void updateCache(List<Student> studentList) {
        cache.clear();
        insertCache(studentList);
    }

    public void updateStudentById(int id, Student student) {
        cache.replace(id, student);
    }

    public List<Student> getAllStudents() {
```

```
        return new ArrayList<>(cache.values());
    }
}
```

view.java

```
public class View {
    public void printStudent(Student student) {
        System.out.println(student.toString());
    }

    public void printStudents(List<Student> studentList) {
        for(Student student : studentList) {
            printStudent(student);
        }
    }
}
```

1.2. MVP (Model-View-Presenter)

Паттерн MVP — производный от паттерна MVC.

Элемент **Presenter** в этом шаблоне берёт на себя функциональность посредника по аналогии с контроллером в паттерне MVC и отвечает за управление событиями пользовательского интерфейса.

Обычно экземпляр представления создаёт экземпляр презентера, передавая ему ссылку на себя. При этом презентер работает с представлением в абстрактном виде, через его интерфейс.

Когда вызывается событие представления, оно вызывает конкретный метод презентера, не имеющий ни параметров, ни возвращаемого значения.

Презентер получает необходимые для работы метода данные о состоянии пользовательского интерфейса через интерфейс представления и через него же передаёт в представление данные из модели и другие результаты своей работы.

На рисунке ниже изображена схема паттерна MVP.

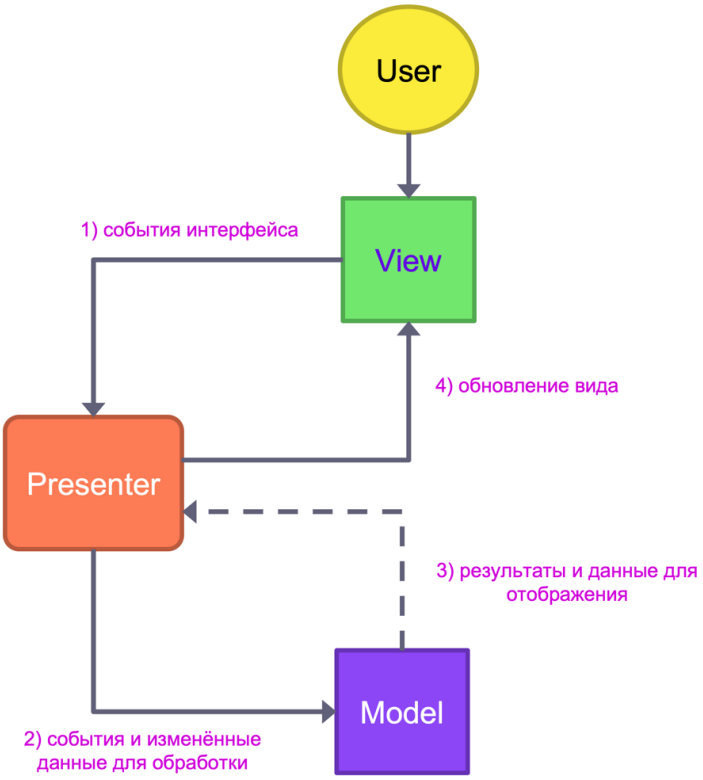


Схема MVP

MVP — шаблон проектирования пользовательского интерфейса, который был разработан для облегчения автоматического модульного тестирования и улучшения разделения ответственности, то есть отделения логики от отображения. Состоит из трёх компонент:

- **Model** — данные для отображения;
- **View** реализует отображение данных из модели, обращается к презентеру за обновлениями, перенаправляет события от пользователя в презентер;

- **Presenter** реализует взаимодействие между моделью и представлением и содержит в себе всю логику представления данных о предметной области; при необходимости получает данные из хранилища и преобразует для отображения во View.

Функции презентера:

- двухсторонняя коммуникация между презентером и представлением;
- представление взаимодействует напрямую с презентером путем вызова соответствующих функций;
- презентер взаимодействует с представлением через специальный интерфейс, реализуемый представлением.

Ключевые моменты архитектуры MVP:

- связь между представлением – презентером и представлением – моделью происходит через интерфейс, также называемый контрактом;
- один класс Presenter управляет только одним View, то есть между Presenter и View существует связь один-к-одному;
- класс модели и класс представления не знают о существовании друг друга.

Реализация: каждое представление должно реализовывать интерфейс, описывающий методы взаимодействия с презентером.

Презентер должен хранить ссылку на реализацию соответствующего интерфейса, которую обычно передают в конструкторе.

Логика представления должна иметь ссылку на экземпляр презентера. Обработка всех событий представления, как правило, делегируется для обработки в презентер и практически никогда не обрабатываются в представления.

С точки зрения MVP, подклассы UIViewController на самом деле есть View, а не Presenter. Это различие обеспечивает превосходную тестируемость, которая

идет за счет скорости разработки, потому что вы должны связывать вручную данные и события именно между View и Presenter.

MVP является первым паттерном, выявляющим проблему сборки, которая происходит из-за наличия трех действительно отдельных слоев.

Так как здесь не требуется, чтобы View знала о Model, выполнять сборку в презентующей View Controller (который на самом деле View) неправильно, следовательно, это нужно сделать в другом месте.

Например, можно создать сервис Router, который будет отвечать за выполнение сборки и презентацию View-to-View. Эта проблема возникает не только в MVP, ее также нужно решать во всех последующих паттернах.

Признаки хорошей архитектуры для MVP:

- **распределение:** большая часть ответственности разделена между Presenter и Model, а View ничего не делает;
- **тестируемость:** отличная, можно проверить большую часть бизнес-логики благодаря бездействию View;
- **простота использования:** количество кода для написания небольшого модуля в два раза больше по сравнению с MVC для аналогичного модуля, но в то же время идея MVP очень проста.

MVP в мобильной разработке означает превосходную тестируемость. Она экономит время и ресурсы на разработку. Реализуются главные функции и не надо тратить огромные деньги на маркетинговые исследования и инвестировать их в создание сложнейшего проекта, на который, возможно, уйдет много сил и времени, а результат впечатлять не будет.

MVP даёт гарантированное рабочее решение, выполняющее главные задачи. Приложение точно будет запущено, потому что разрабатываются только ключевые функции, а не огромный функционал, нуждающийся в постоянных переработках и стабилизации.

Пример MVP на Java

model.java

```
public class Model {
    public List<Student> getLocalStudents() {
        return Cache.listOfStudents();
    }

    public List<Student> getRemoteStudents() {
        return Cache.anotherListOfStudents();
    }
}
```

presenter.java

```
public class Presenter {
    private View viewContract;
    private final Model model;

    public Presenter(Model model) {
        this.model = model;
    }

    public void attachView(ViewContract viewContract) {
        this.viewContract = viewContract;
    }
```

```
    public void loadStudents() {
        viewContract.showStudents(model.getRemoteStudents());
        viewContract.showSeparator();
    }

    public void getLocalStudents() {
        viewContract.showStudents(model.getLocalStudents());
        viewContract.showSeparator();
    }
}
```

view.java

```
public class View {
    private final Presenter presenter;

    public View(Presenter presenter) {
        this.presenter = presenter;
    }

    public void showStudents(List<Student> students) {
        students.forEach(System.out::println);
```

```
    }


    public void showSeparator() {
        System.out.println(“----");
    }
}
```

1.3. MVVM (Model-View-ViewModel)

Паттерн MVVM также является производным от паттерна MVC. Он был представлен Джоном Госсманом (John Gossman) в 2005 году как модификация шаблона Presentation Model и первоначально был нацелен на разработку приложений в WPF.

MVVM расшифровывается как Model-View-ViewModel.

Сейчас паттерн вышел за пределы WPF и применяется в разных технологиях, в том числе при разработке под Android и iOS, а также как часть SPA-фреймворков. Тем не менее WPF — довольно показательная технология, которая раскрывает возможности этого паттерна.

MVVM состоит из трех компонентов:

- **Model** (модель) — как и в MVC, описывает используемые в приложении данные и содержит бизнес-логику.
- **View** (представление) определяет визуальный интерфейс, через который пользователь взаимодействует с приложением. Применительно к WPF представление — это декларативный код в xaml (строгий аналог html), который определяет интерфейс в виде кнопок, текстовых полей и прочих визуальных элементов.
- **ViewModel** (модель представления) — основное отличие MVVM от MVC. В отличие от контроллера в MVC модель представления в MVVM связывает модель и представление через механизм привязки

данных. Если в модели изменяются значения свойств, автоматически идет изменение отображаемых данных в представлении, хотя напрямую модель и представление не связаны.

На рисунке ниже изображена схема паттерна MVVM.

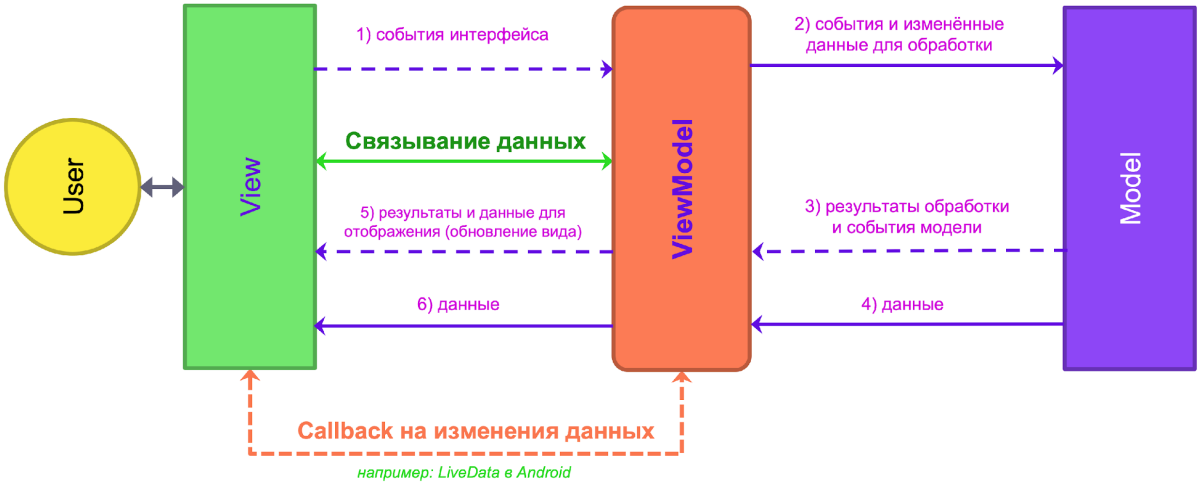


Схема паттерна MVVM

Представление выступает подписчиком на события, предоставляемые моделью представления (ViewModel).

Если в модели представления изменилось какое-либо свойство, она оповещает представление, а оно, в свою очередь, запрашивает обновлённое значение свойства из модели представления.

Если пользователь воздействует на какой-либо элемент интерфейса, представление вызывает соответствующую команду, предоставленную моделью представления.

MVVM — один из новейших MV*-паттернов. Он очень похож на MVP:

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

Кроме того, он делает биндинг (связывание) как надзирающая версия MVP, но не между View и Model, а между View и View Model.

Реализация: ViewModel представляет собой, с одной стороны, абстракцию над представлением, а с другой получает ссылки на свойства модели, подлежащие связыванию. То есть **ViewModel** содержит модель, преобразованную к представлению, а также команды, которыми может пользоваться представление, чтобы влиять на модель.

Особенности ViewModel:

- Двухсторонняя коммуникация модели представления с представлением.
- Модель представления — это абстракция представления. Обычно это означает, что свойства представления совпадают со свойствами модели представления.
- Модель представления не хранит ссылки на интерфейс представления. Изменение состояния модели представления автоматически изменяет представление и наоборот, благодаря механизму связывания данных (two-way binding).

При этом функции компонент строго разделены и немного отличаются от функций компонент в MVC/MVP.

Model — модуль не отличается от аналогичного в MVC. Он отвечает за создание моделей данных и может содержать в себе бизнес-логику. Вы также можете создать вспомогательные классы, например, manager-класс для управления объектами в Model и network manager для обработки сетевых запросов и парсинга.

Свойства Model:

- Содержит описание данных.
- Реализует CRUD.
- Оповещает ViewModel о произошедших изменениях в данных.
- Содержит бизнес-логику.
- Содержит логику валидации данных.
- MVVM никак не описывает реализацию Model, и она может быть реализована любым способом, даже VIPER модулем.

View — охватывает интерфейс, логику отображения (анимацию, отрисовку) и обработку пользовательских событий (нажатие кнопок и так далее) В MVC за это отвечают View и Controller. Это означает, что интерфейсы (Views) останутся неизменными, в то время как ViewController будет содержать малую часть того, что было в нем в MVC и, соответственно, сильно уменьшится.

ViewModel — слой запрашивает данные у Model (это может быть запрос к локальной базе данных или сетевой запрос) и передает их обратно во View уже в том формате, в котором они будут там использоваться и отображаться. Но это двунаправленный механизм, действия или данные, вводимые пользователем, проходят через ViewModel и обновляют Model. Поскольку ViewModel следит за всем, что отображается, то полезно использовать механизм связывания между этими двумя слоями.

Свойства **ViewModel:**

- Хранит состояние VIew.
- Знает о Model и может менять её состояние (вызывая соответствующие методы Model).
- Как правило, образует связь «один ко многим» с несколькими моделями.
- Преобразует данные, полученные от Model в формат, удобный для отображения View и обратно.
- Ничего не знает о View и может с ним коммуницировать только благодаря биндингам.
- Может содержать логику валидации данных.
- Может включать в себя бизнес-правила работы с действиями пользователя.

ViewModel знает обо всех компонентах и управляет ими. Это и достоинство, и недостаток MVVM:

- С одной стороны, ViewModel — удобный медиатор, оторванный от мира UI и Data Access, а значит её удобно тестировать и можно переиспользовать на других платформах.

- С другой стороны, ViewModel занимается и управлением состояниями своего View и зависимостей, и различного рода логикой. А значит её код разрастается, и мы возвращаемся к проблеме Massive-View-Controller. Чтобы этого избежать, необходимо крайне осторожно подходить к проектированию ViewModel.

ViewModel легко протестировать, но заниматься отладкой приложения на MVVM непросто: благодаря биндингам данные меняются мгновенно.

Оценка признаков хорошей архитектуры:

- **распределение:** в MVVM View имеет больше обязанностей, чем View из MVP, потому что первая обновляет свое состояние с ViewModel за счет установки биндингов, тогда как вторая направляет все события в Presenter и не обновляет себя (это делает Presenter);
- **тестируемость:** ViewModel не знает ничего о представлении, это позволяет нам с легкостью тестировать ее. View также можно тестировать;
- **простота использования:** объем кода для небольшого модуля примерно такой же, как в MVP для аналогичного модуля, но в реальном приложении, где вам придется направить все события из View в Presenter и обновлять View вручную, MVVM будет гораздо меньше по количеству кода, чем MVP (если вы используете биндинги).

MVVM сочетает в себе преимущества вышеупомянутых подходов и не требует дополнительного кода для обновления View в связи с биндингами на стороне View. Тем не менее тестируемость все еще находится на хорошем уровне.

MVVM хорошо подходит опытным командам среднего размера, которые работают над проектом с большим количеством представлений (экранов). Особенно хорошо MVVM себя показывает в проектах, где надо сильно отделить бизнес-логику от логики отрисовки представлений — такое часто встречается в кроссплатформенных приложениях или приложениях, которым в будущем может понадобиться клиент-сателлит на другую платформу.

Из-за сильного разделения MVVM позволяет нескольким разработчикам работать над одной задачей одновременно, практически не пересекаясь в коде.

Это самая «простая» для понимания и внедрения архитектура, которая обладает таким замечательным свойством.

Пример MVVM в Java

model.java

```
public class Model {
   private int listNumber = -1;


   public int getListNumber() {
      return listNumber;
   }


   public List<Student> getData() {
      return listNumber != 1 ? Cache.listOfStudents() : Cache.anotherListOfStudents();
   }
   public void updateListNumber() {
      listNumber = listNumber != 0 ? 0 : 1;
   }
}
```

viewmodel.java

```
public class AppViewModel extends ViewModel {
```

```
    private final Model model = new Model();


 private final MutableLiveData<List<Student>> students = new MutableLiveData<>();

 private final MutableLiveData<Integer> studentsListNumber = new
MutableLiveData<>(model.getListNumber());


    public LiveData<List<Student>> studentsLiveData() {
        return students;
    }


    public LiveData<Integer> listNumberLiveData() {
        return studentsListNumber;
    }


    public void updateData() {
        model.updateListNumber();
        studentsListNumber.postValue(model.getListNumber());
        students.postValue(model.getData());
    }
}
```

view.java

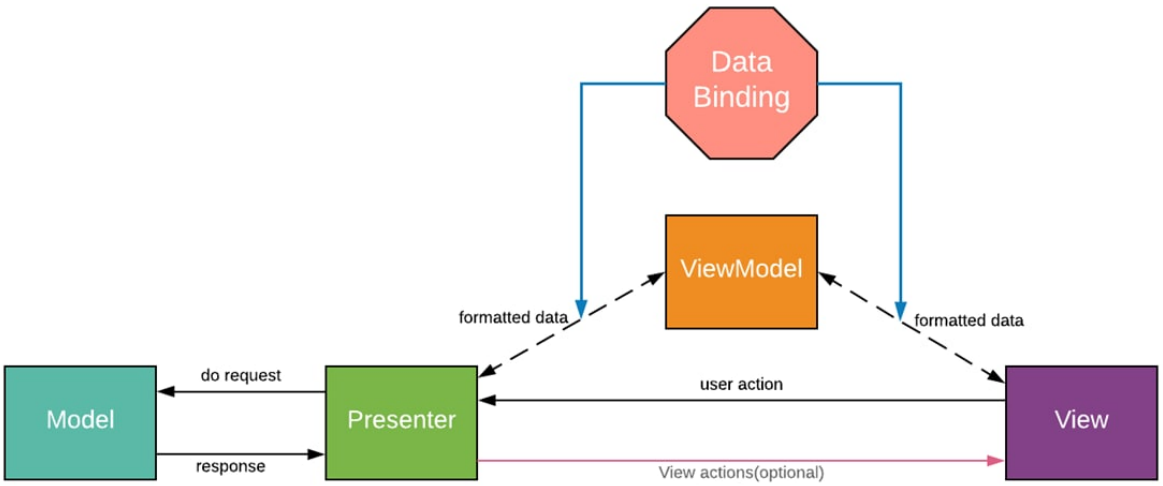
```
public class MainActivity extends AppCompatActivity {


  private AppViewModel viewModel;
  private ActivityMainBinding binding;
  private StudentArrayAdapter adapter;


  @Override
  protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    binding = ActivityMainBinding.inflate(getLayoutInflater());
    View root = binding.getRoot();
    viewModel = new ViewModelProvider(this).get(AppViewModel.class);
    initAdapter();
    initButton();
    initObserver();
    initListNumber();
    setContentView(root);
  }


  private void initAdapter() {
    adapter = new StudentArrayAdapter(this);
    binding.studentList.setAdapter(adapter);
  }
```

```
    private void initButton() {
        binding.updateList.setOnClickListener(view -> viewModel.updateData());
    }

    private void initObserver() {
        viewModel.studentsLiveData().observe(this, studentList -> {
            adapter.clear();
            adapter.addAll(studentList);
            adapter.notifyDataSetChanged();
        });
    }

    private void initListNumber() {
        viewModel.listNumberLiveData().observe(this, number ->
                binding.listNumber.setText(getString(R.string.list_number, number))
        );
    }
}
```

1.4. MVPVM (Model-View-Presenter-ViewModel)

Паттерн MVPVM — дальнейшее развитие идеи, заложенной в MVVM. MVPVM обеспечивает всю мощь и возможности MVVM, одновременно обеспечивая масштабируемость и расширяемость шаблона Model-View-Presenter (MVP). В

десктопных приложениях под ОС Windows существующий фреймворк Windows Presentation Foundation (WPF) упрощает реализацию MVPVM.



Архитектура паттерна MVPVM

Реализация: поведение модели схоже с классическим паттерном MVC. Представление (View) связывается с Presenter через интерфейс и в специальных методах сообщает о произошедших изменениях, реализуя стандартный паттерн «Наблюдатель» (Observer) аналогично классическому MVP.

При этом представление не хранит ссылку на модель представления (ViewModel), обеспечивая тем самым слабую связанность.

Presenter в рассматриваемом паттерне является центральным звеном, сообщающим модели о том, какие изменения необходимо произвести, получающий результаты изменений, по необходимости обновляя представление.

Presenter также хранит ссылку на модель представления, которой благодаря такому подходу не нужно, в свою очередь, хранить ссылку на модель, что позволяет уменьшить связанность и позволить переиспользовать модель представления для разных представлений.

При таком подходе появляется дополнительный слой, позволяющий отслеживать изменения представления, сообщая Presenter о произошедших изменениях и наоборот, по необходимости вызывать методы изменения у представления при обновлении данных в Presenter. Такое поведение возможно благодаря механизму биндингов.

Особенности MVPVM:

- отсутствие бизнес-логики в модели представления;
- дополнительный слой контроля за согласованностью данных в Presenter и представлении;
- возможность использования нескольких представлений с одним Presenter.

1.5. MVI (Model-View-Intent)

Архитектурный шаблон MVI (Model-View-Intent) был определен Андре Стальцем для фреймворка Javascript cycle.js. Он также подходит для Android или любого другого приложения на основе пользовательского интерфейса. Шаблон предназначен для создания удобочитаемой базы кода и более согласованно отделяет пользовательский интерфейс от бизнес-логики.

Для его реализации в Android-приложении можно воспользоваться библиотеками:

- Roxie
- Grox
- RxRedux

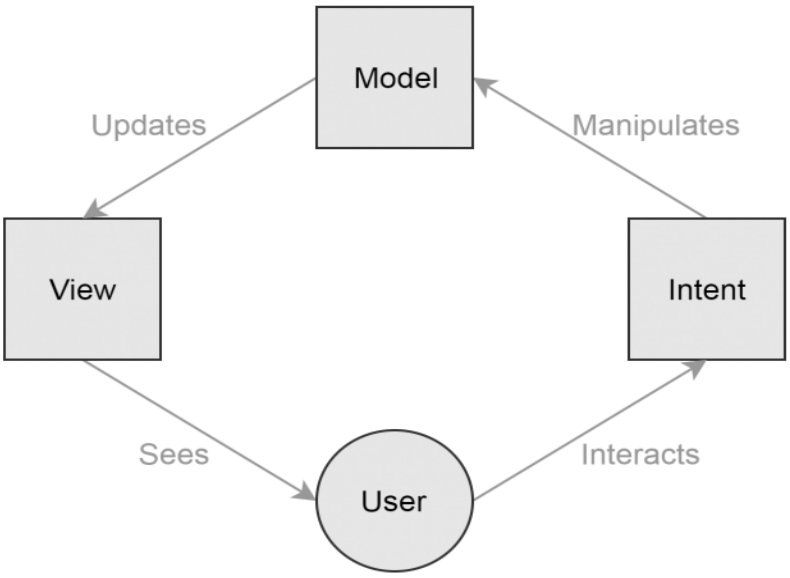


Схема MVI

MVP и MVVM — отличные шаблоны для проектов любого размера, при этом MVC редко используется в проектах бóльших, чем небольшие приложения, но ни один из них не был построен с учетом концепции реактивного взаимодействия. Наш код становится намного проще, поскольку мы используем множество расширений, уже присутствующих в большинстве реактивных фреймворков, также повышается удобочитаемость.

MVI еще больше разделяет компоненты, в итоге делая поток управления полностью однонаправленным. Когда все сделано правильно, зависимости MVI выстраиваются в виде круга.

Реализация: представление реагирует на пользовательский ввод и формирует событие (Actions), хранящее в себе информацию о том, что изменилось. Событие попадает в хранилище (в некоторых реализациях — сразу в модель), где происходит первичная обработка события. Также, если необходима генерация Action и отправка события в модель.

В случае успешного выполнения всех функций модели происходит генерация нового состояния и запрос представления на обновление внутреннего состояния в соответствии с предложенным.

В случае неудачного выполнения функций модели происходит откат к исходному состоянию представления с вызовом событий, если они необходимы.

Признаки событийного подхода:

- паттерн, как правило, поддерживает однонаправленный поток данных;
- чаще всего реализуется при помощи реактивного подхода.

Intent (намерение) представляет собой взаимодействие пользователя с представлением. Намерение доставляет изменения, которые должны быть обработаны моделью.

Model больше похожа на состояние, чем на исходную идею модели. Включает взаимодействие с бизнес-логикой. Бизнес-логика обновляет текущее состояние и, следовательно, наблюдающее представление.

View — представление, актуальное для пользовательского интерфейса. Наблюдает и отображает состояние модели. Перенаправляет каждое взаимодействие как намерение.

Модели MVI сами по себе являются состояниями. Вместо того чтобы ViewModel уведомлял наше представление о новых моделях, наша модель сама является состоянием.

Отсюда несколько преимуществ:

- модели больше не изменяемы;
- каждая модель — это состояние, в которое может войти представление;
- ошибки можно найти быстрее, так как мы можем сосредоточиться на состоянии, в котором они возникли;
- состояния можно легко сделать стойкими.

Намерение является взаимодействием, вызов которых пользователем, приводит к генерации Action. Компонент Intent часто называют Presenter, например, в MVP, но он реализован по-другому.

Технически модель все еще существует. Просто она не используется так же, как в MVP. MVP использует модели как контейнеры данных, а модели в MVI — это состояния, в которые может входить приложение. Например, у нас может быть модель, которая указывает, что приложение должно отображать полосу загрузки, в то время как другое состояние может содержать данные для отображения.

Оценка признаков хорошей архитектуры:

- **распределение:** модели больше не изменяемы, они стали состояниями. Однонаправленный поток позволяет нам работать без обратных вызовов между представлением и, в данном случае, намерением.
- **тестируемость:** в MVI компоненты легко тестируются.
- **простота использования:** использование реактивного программирования делает код очень простым для чтения и поддержки. Хотя MVI сложен для людей, не знакомых с реактивным

программированием, у него могут быть огромные преимущества в определенных проектах.

MVI — отличная архитектура. Она делает код читаемым, открывает возможности лёгкого расширения, просто добавлением новых состояний. MVI — архитектура, которую лучше всего использовать в качестве замены для MVVM.

2. Чистая архитектура прикладных приложений

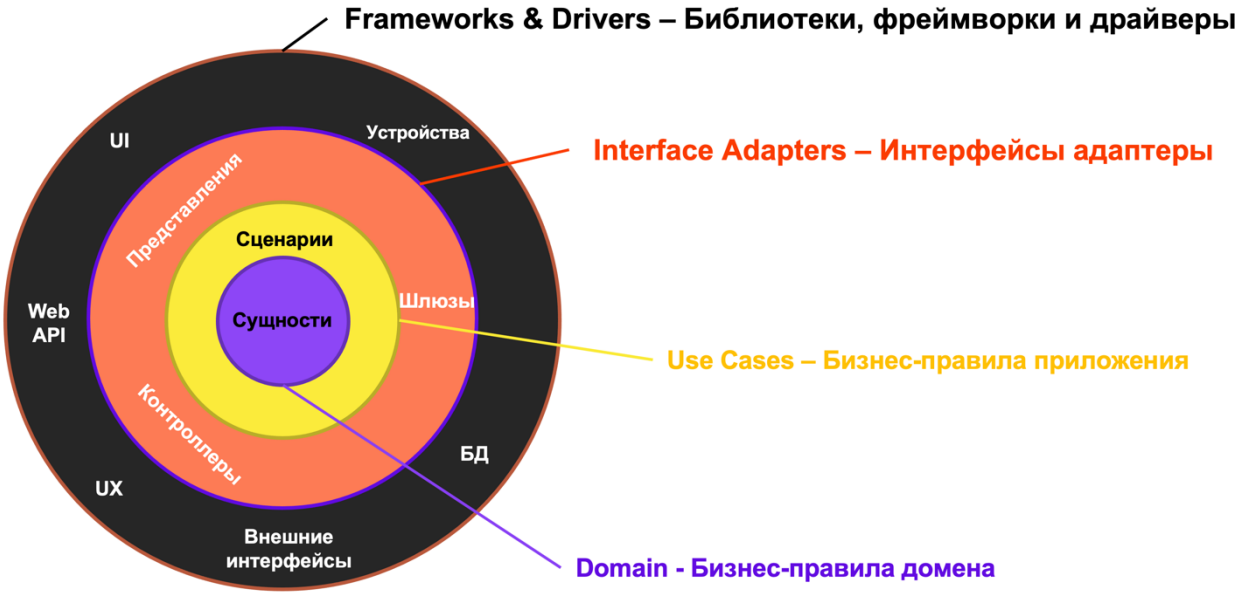
На больших проектах со сложной бизнес-логикой требуются более основательные подходы к проектированию кодовой базы приложения. Один из таких подходов — концепция Clean Architecture (чистая архитектура), которую мы детально рассматривали ранее.

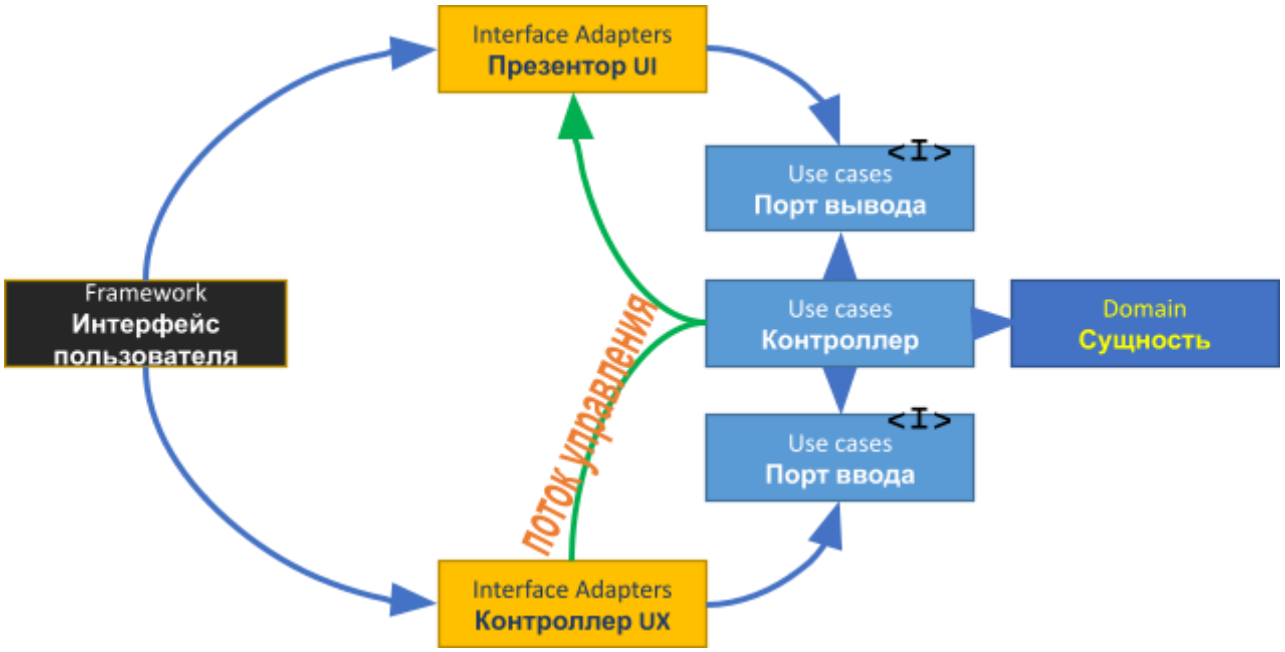
Clean Architecture делит логическую структуру приложения на различные уровни обязанностей. Это упрощает изолирование зависимости (например, ваша база данных) и тестирование взаимодействия на границах между уровнями.

Понятие «чистая архитектура» появилось в одноименной статье Роберта Мартина 2012 года. Кратко повторим её принципы:

- **Независимость от фреймворков.** Архитектура не должна полагаться на существование какой-либо библиотеки. Так вы сможете использовать фреймворки как инструменты, а не будете пытаться загнать свою систему в их ограничения.
- **Тестируемость.** Бизнес-логика должна быть тестируемой без внешних элементов вроде интерфейса, базы данных, сервера или любых других.
- **Независимость от интерфейса.** Интерфейс должен легко изменяться и не требовать изменения остальной системы. Например, веб-интерфейс должен заменяться на интерфейс консоли без необходимости изменения бизнес-логики.
- **Независимость от базы данных.** Ваша бизнес-логика не должна быть привязана к конкретным базам данных.
- **Независимость от любого внешнего агента.** Ваша бизнес-логика не должна знать вообще ничего о внешнем мире [12].

Отражение этих принципов в архитектуре программного обеспечения можно представить следующим образом:


Frameworks & Drivers – Библиотеки, фреймворки и драйверы
UI
Устройства
Представления
Сценарии
Сущности
Шлюзы
Web API
Контроллеры
БД
UX
Внешние интерфейсы
Interface Adapters – Интерфейсы адаптеры
Use Cases – Бизнес-правила приложения
Domain - Бизнес-правила домена


Interface Adapters
Презентор UI
Use cases
Порт вывода
<I>
Framework
Интерфейс пользователя
Use cases
Контроллер
Domain
Сущность
поток управления
Use cases
Порт ввода
<I>
Interface Adapters
Контроллер UX

Представление принципов Clean Architecture в архитектуре программного обеспечения

В этой схеме:

- **Domain** — бизнес-логика, общая для всех приложений. А в случае отдельного приложения — наиболее базовые бизнес-объекты.
- **Use Cases** — логика приложения. Сценарии применения, которые управляют потоком данных из предыдущего слоя.
- **Interface Adapters** — адаптеры между Use Cases и внешним миром. Этот слой конвертирует данные в формат, подходящий для внешних слоев, например, Web или базы данных, а также превращает внешние данные в формат для внутренних слоев.
- **Frameworks and Drivers** — внешний слой, содержащий фреймворки, инструменты, базы данных и так далее. В этом слое код должен связываться с предыдущим слоем, но не должен влиять в значительной степени на внутренние слои.

Все слои связаны правилом зависимости — **Dependency Rule**, которое гласит, что в исходном коде все зависимости могут указывать только вовнутрь. Например, ничто из внешнего круга не может быть упомянуто кодом из внутреннего круга. Это относится к функциям, классам, переменным или любым другим сущностям.

Сам Uncle Bob говорит, что эту схему можно изменять: добавлять или убирать слои, но основным правилом в архитектуре приложения должно всегда оставаться Dependency Rule.

2.1. Чистая архитектура в iOS-разработке VIPER

Для iOS-разработки чистая архитектура реализована в модели VIP или VIPER, которая позиционируется как замена шаблону MVC.

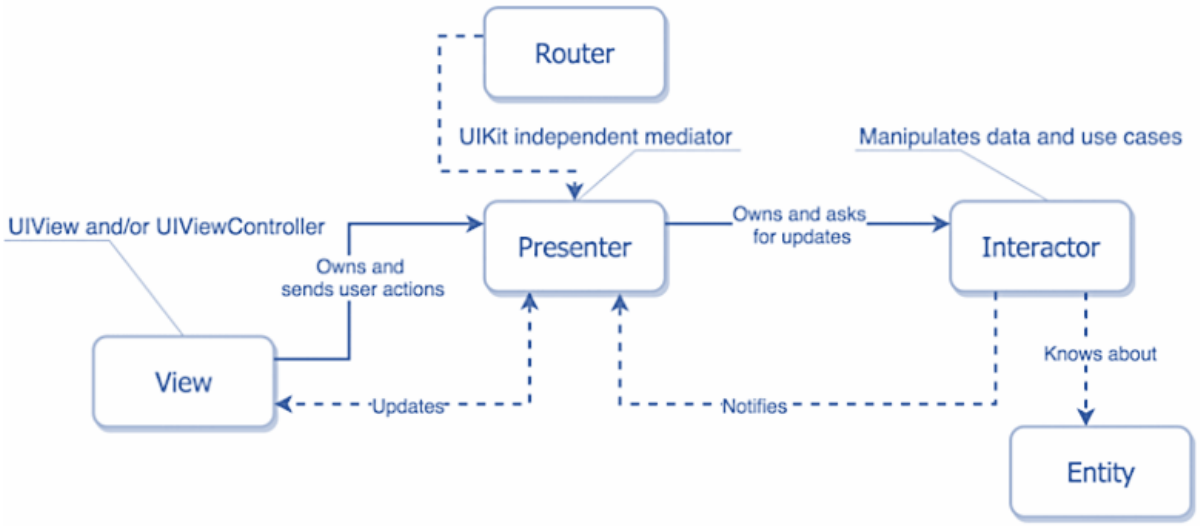


Схема VIPER

VIPER — для View, Interactor, Presenter, Entity и Routing. Но реальность такова, что в модуль входят не только эти компоненты, а Entity вообще может не входить в понятие модуля, так как является самодостаточным классом, который может использоваться в любом модуле или сервисе. На сложных экранах модуль можно делить на подмодули, где у каждого будут свои презентеры и интеракаторы.

Interactor содержит бизнес-логику для управления объектами (Entity), чтобы выполнить определенную задачу. Задача выполняется в Interactor независимо от любого UI. Тот же Interactor можно использовать в iOS-приложениях или консольных приложениях для Mac OS. Поскольку Interactor применяет свою бизнес-логику, он будет осуществлять выборку Entity из хранилища данных, управлять Entity и затем возвращать обновленные Entity назад в хранилище данных. Хранилище данных управляет персистентностью Entity. Entity не знают о хранилище данных, таким образом, они не знают, как сохраняться.

Entities — простые объекты данных. Не являются слоем доступа к данным, потому что это ответственность слоя Interactor.

Presenter в основном состоит из логики, для управления UI. Он собирает входные данные от взаимодействия с пользователем. Таким образом, он может отправлять запросы в Interactor. Presenter также получает результаты Interactor и преобразовывает их в наиболее эффективное состояние для отображения на View.

Entity никогда не передаются из интерактора к презентеру. Вместо них передаются простые структуры данных, у которых нет поведения. Это препятствует любой полезной работе в презентере. Presenter может только подготовить данные для отображения на View.

View является пассивной. Она ждет презентера, чтобы передать содержания для выведения на экран. Она никогда не запрашивает данные у презентера. Методы, определенные для представления (например, LoginView для экрана входа в систему), должны позволить презентеру общаться на более высоком уровне абстракции, выраженной с точки зрения его содержимого, а не то, как это содержимое будет отображаться. Presenter не знает о существовании UILabel, UIButton, и т.д. Presenter знает только о содержании, которое он поддерживает, и о том, когда его нужно вывести на экран. Presenter-у нужно определять, как содержание выводиться на экран.

View — это абстрактный интерфейс, определенный в Objective-C с помощью протокола. UIViewController или один из его подклассов реализуют протокол View. Например, это может быть экран входа в систему.

Router несет ответственность за переходы между VIPER-модулями.

Модулем VIPER может быть один экран или целая User Story вашего приложения (например, аутентификация может быть на один экран или несколько связанных экранов).

Если сравнить VIPER с паттернами MV*-вида, увидим несколько отличий в распределении обязанностей:

- логика из Model (взаимодействие данных) смещается в Interactor. Также есть Entities — структуры данных, которые ничего не делают.
- Из Controller, Presenter, ViewModel обязанности представления UI переехали в Presenter, но без возможности изменения данных.
- VIPER является первым шаблоном, который пробует решить проблему навигации — для этого есть Router.

В VIPER-модуле большое количество классов и протоколов. Это необходимо для обеспечения слабой связности и высокого зацепления, которые вместе делают код понятным, легко поддерживаемым и устойчивым к внешним изменениям, появляется возможность использовать код модуля несколько раз. Всё это облегчает работу над проектом, причем не важно, командная это

разработка или нет. VIPER, благодаря соблюдению этих принципов, позволяет легко менять части модуля, тем самым обеспечивая быстрое реагирование на изменение дизайна или требований.

Clean Swift (VIP)

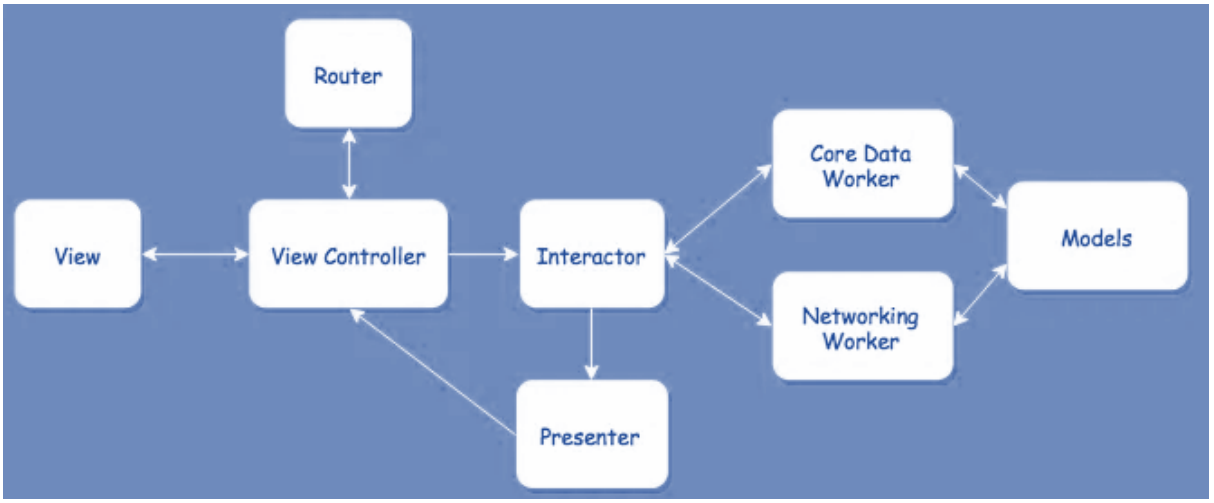


Схема VIP

Clean Swift, на первый взгляд, похож на VIPER. Отличия становятся видны после изучения принципа взаимодействия модулей.

В VIPER основу взаимодействия составляет Presenter — он передает запросы пользователя интерактору для обработки и форматирует полученные от него назад данные для отображения на View Controller.

В Clean Swift основными модулями так же, как и в VIPER, являются View Controller, Interactor, Presenter. Взаимодействие между ними происходит циклично. Передача данных основана на протоколах (аналогично VIPER), что позволяет при будущем изменении какого-то из компонентов системы просто подменить его на другой.

Процесс взаимодействия выглядит так: пользователь нажимает на кнопку, View Controller создает объект с описанием и отправляет его в Interactor. Interactor, в свою очередь, осуществляет какой-то конкретный сценарий в соответствии с бизнес-логикой, создает объект результата и передает его Presenter. Presenter формирует объект с отформатированными данными для отображения пользователю и отправляет его во View Controller.

Рассмотрим каждый модуль Clean Swift подробнее.

View Controller — как и в VIPER выполняет все конфигурации VIew, будь то цвет, настройки шрифта UILabel или Layout. Поэтому каждый UIViewController в данной архитектуре реализует Input-протокол для отображения данных или реакции на действия пользователя.

Interactor содержит в себе всю бизнес-логику. Он принимает действия пользователя от контроллера с параметрами (например, измененный текст поля ввода, нажатие той или иной кнопки), определёнными в Input-протоколе. После отработки логики Interactor при необходимости должен передать данные для их подготовки в Presenter перед отображением в ViewController. Однако Interactor принимает на вход запросы только от View, в отличие от VIPER, где эти запросы проходят Presenter.

Presenter обрабатывает данные для показа пользователю. Результат в этом случае — это Input-протокол ViewController'a. Здесь можно, например, поменять формат текста, перевести значение цвета из enum в rgb и так далее.

Worker — дополнительный элемент, который можно использовать, чтобы излишне не усложнять Interactor и не дублировать детали бизнес-логики. В простых модулях он не всегда нужен, но в достаточно нагруженных позволяет снять с Interactor часть задач. Например, в Worker может быть вынесена логика взаимодействия с базой данных, особенно если одни и те же запросы к базе могут использоваться в разных модулях.

Router ответственен за передачу данных другим модулям и переходы между ними. У него есть ссылка на контроллер, потому что в iOS, к сожалению, контроллеры, помимо всего прочего, исторически ответственны за переходы. При использовании segue можно упростить инициализацию переходов благодаря вызову методов Router из Prepare for segue, потому что Router знает, как передать данные, и сделает это без лишнего кода цикла со стороны Interactor/Presenter. Данные передаются, используя протоколы хранилищ данных каждого модуля, реализуемых в Interactor. Эти протоколы также ограничивают возможность доступа к внутренним данным модуля из Router.

Models — это описание структур данных для передачи данных между модулями. Каждая реализация функции бизнес-логики имеет свое описание моделей.

- **Request** — для передачи запроса из контроллера в интерактор.
- **Response** — ответ интерактора для передачи презентеру с данными.

- **ViewModel** — для передачи данных в готовом для отображения в контроллере виде.

Оценка признаков хорошей архитектуры для VIP и VIPER:

- **Распределение**: несомненно, VIPER и VIP — чемпионы в распределении обязанностей.
- **Тестируемость**: лучше распределение — лучше тестируемость.
- **Простота использования**: первые два преимущества идут за счет стоимости сопровождения. Вам придется писать огромное количество интерфейсов для классов с незначительными обязанностями.

2.2. Чистая архитектура в Android-разработке

Схему чистой архитектуры **Android-приложений** предложил разработчик Фернандо Сехас.

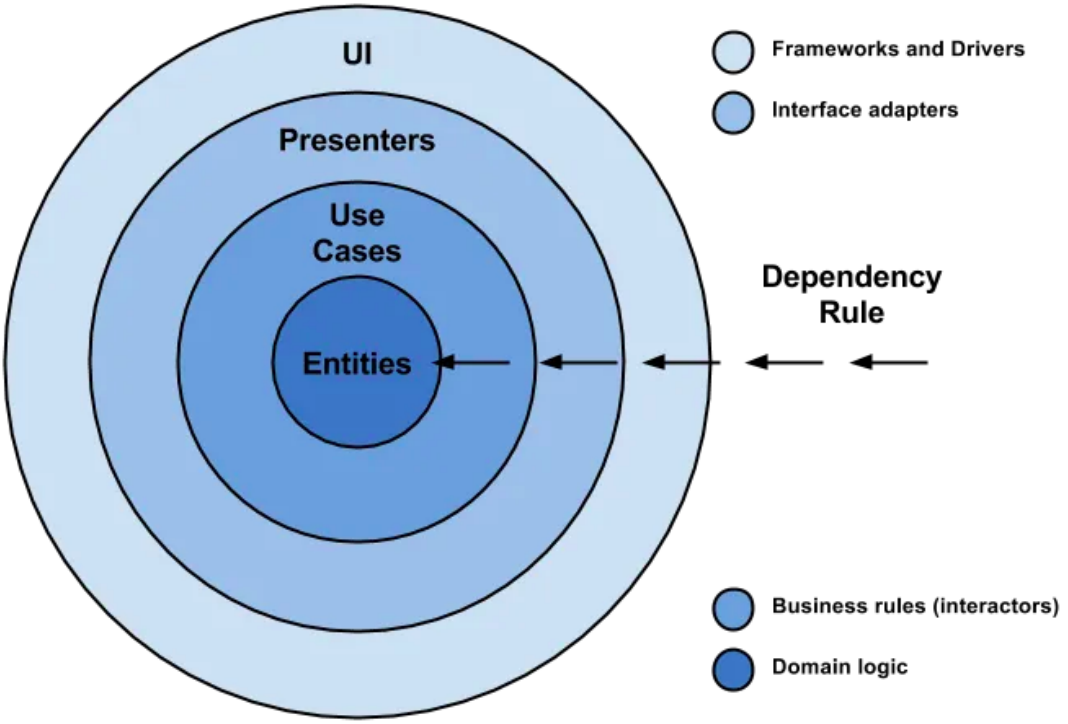
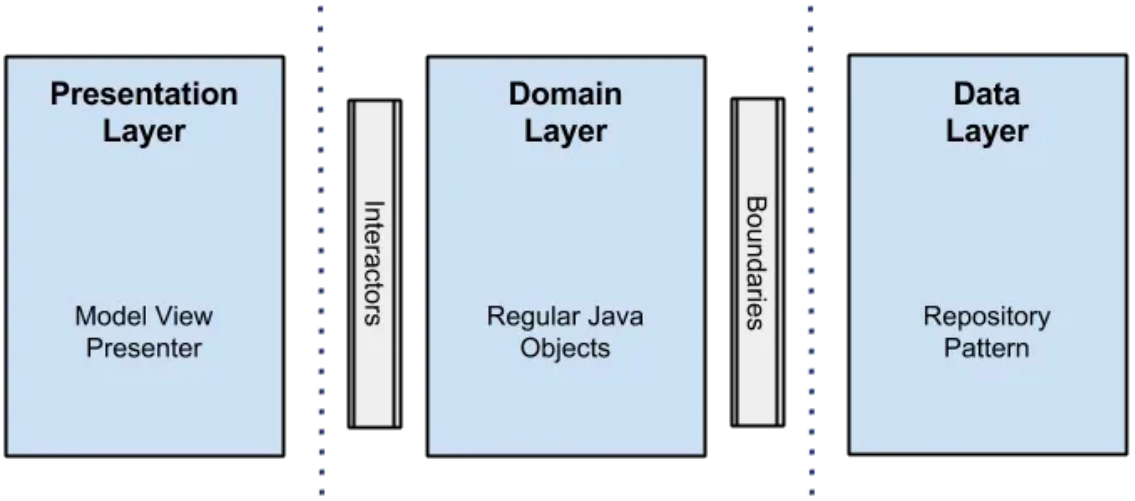


Схема чистой архитектуры Android-приложений

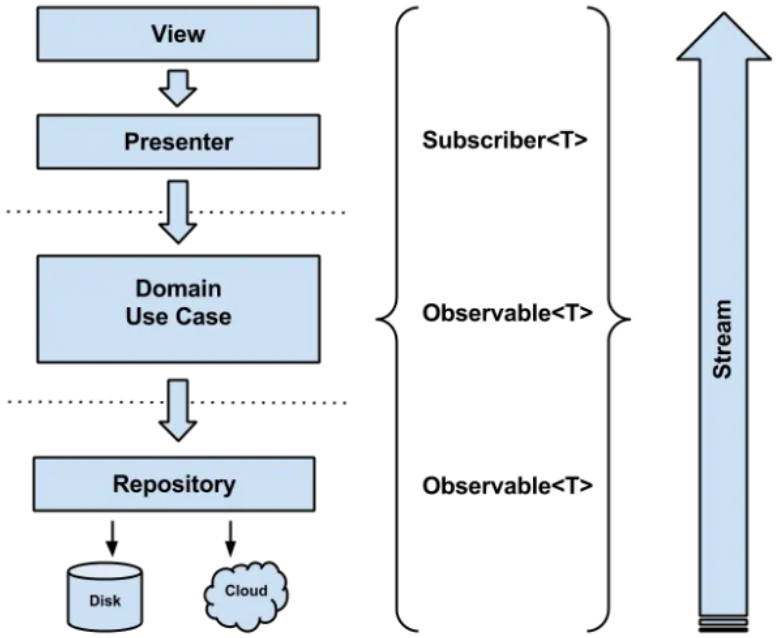
Внутренним слоем является Domain Layer, который хранит всю бизнес-логику. В этом же слое находятся и все Use Cases и реализации. Этот слой

является чистым модулем Java без Android-зависимостей. При связи с ним все внешние компоненты используют интерфейсы.



Разбиение на слои

В Presentation Layer происходит логика, связанная с представлениями и анимациями. Он использует модель Model-View-Presenter, но шаблон может быть другим, например, MVC или MVVM. Фрагменты и активности в этом слое — это представления, в которых происходит только UI-логика и изменение формата данных. Presenter в этом слое формируются из Interactors (Use Cases), которые производят работу вне основного потока UI Android, и возвращаются с данными, которые обрабатываются в View.



Фрагменты и активности Presentation Layer

Data Layer передает все данные через UserRepository, который использует Repository Pattern со стратегией, выбирающей разные источники данных в зависимости от условий. Например, при поиске пользователя по ID с условием существования пользователя в кэше, источником данных будет кэш, в ином случае для получения данных будет отправлен запрос в облако. Источник данных прозрачен для клиента, которого волнует, будут ли получены данные.

3. Server Driven UI

Server driven UI (SDUI) — это архитектура, в которой сервер решает, какие представления пользовательского интерфейса должны быть отображены на экране приложения.

Подход Server driven UI позволяет вносить изменения в программу и управлять ею со стороны сервера. Он открывает новые возможности и решает некоторые фундаментальные проблемы, связанные с разработкой нативных мобильных приложений. Так как реализация любого нововведения под разные операционные системы (Android и iOS) требует немалого количества времени и минимум двух разработчиков. К тому же оба приложения могут вести себя по-разному из-за разного понимания требований этими разработчиками.

Использование Server driven UI решает эту проблему, поскольку каждая бизнес-функция реализуется только один раз на серверной части.

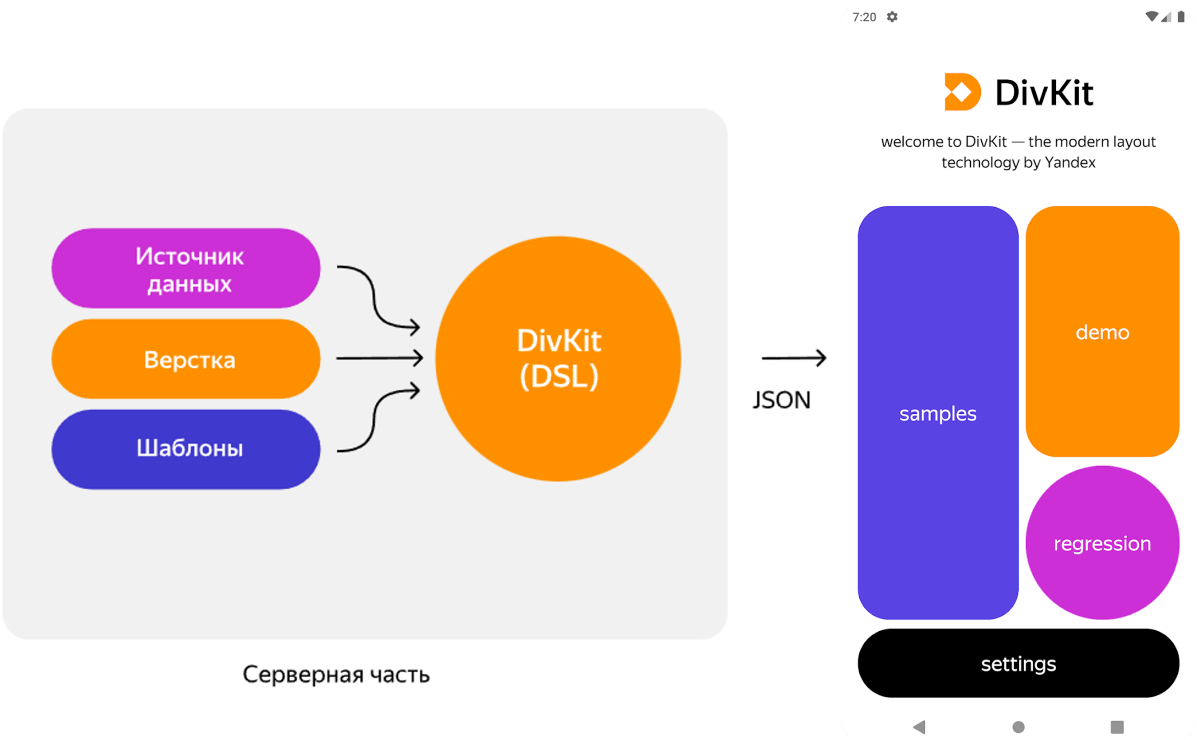
API сообщает клиенту, какие компоненты отображать и с каким содержимым. Это может быть реализовано на всех трех основных платформах: Android, iOS и в веб.

Как работает

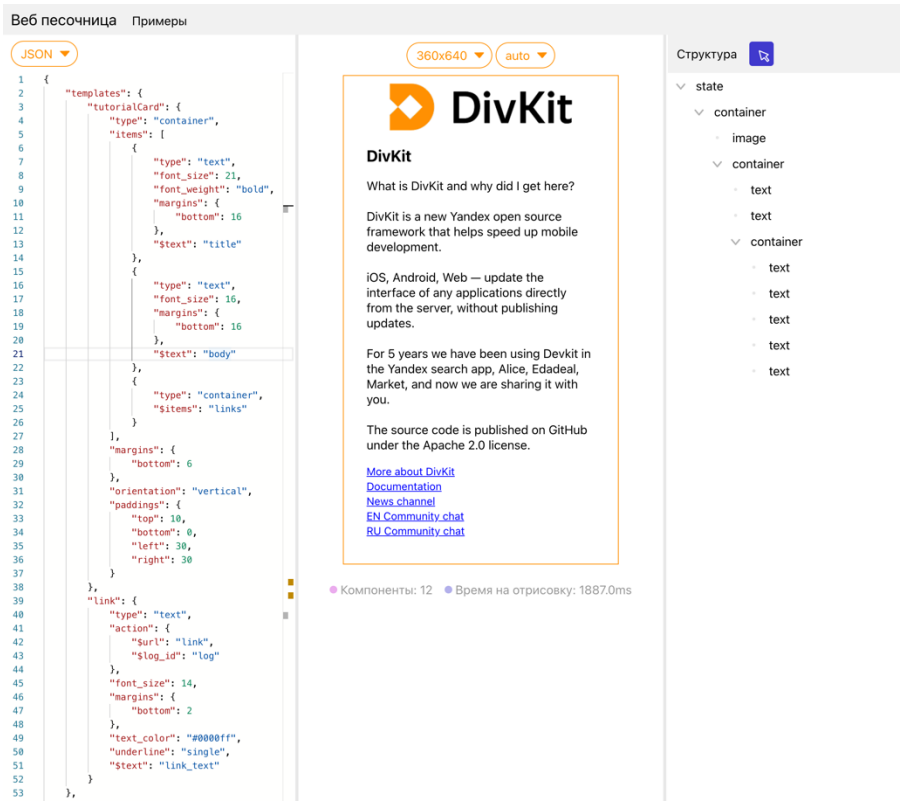
- Между приложением и сервером существует контракт. Основа этого контракта дает серверу контроль над пользовательским интерфейсом приложения.
- Сервер определяет список компонентов. Для каждого из компонентов, определенных на сервере, есть соответствующая реализация пользовательского интерфейса в приложении (UIComponent).
- С сервера будет приходить JSON в формате дерева. Там и будет описана вёрстка компонента.
- Потом из JSON нативно (на платформе) рендерится компонент.
- Если пользователь взаимодействует с кнопкой, то отправится новый HTTP-вызов: в тело этого вызова будет добавлен ID кнопки, в результате чего вернётся новый JSON с сервера и будет отображён.
- Сервер также может послать какие-то дополнительные данные, и они могут быть обработаны.

Пользовательский интерфейс может быть настроен на основе учетных данных пользователя, а веб-сервер вызывает информацию о пользователе с веб-сервера. В этом процессе изображение и содержимое извлекаются и анализируются в приложении в формате JSON.

Наконец, с помощью SDUI kit framework, приложение отображает окончательный обновленный пользовательский интерфейс для конечного пользователя.



Серверная часть



DivKit

Логика, определяющая способ представления данных для каждой строки, встроена в приложение, поэтому для внесения изменений в пользовательский интерфейс нам необходимо пройти полный цикл выпуска. Процесс выглядит примерно так:

1. Разработчики пишут код для внесения необходимых изменений в пользовательский интерфейс.
2. Изменения пользовательского интерфейса проверяют тестировщики.
3. Новая версия приложения отправляется в App/Play Store.
4. Apple/Google рассматривает и утверждает ее.
5. Пользователи обновляют приложение до новой версии.

Преимущества Server-Driven UI

1. Основная бизнес-логика остается неизменной для любого из профилей пользователей.
2. Сокращение времени, необходимого для внедрения той или иной функции без обновления приложения.
3. Возможность кастомизации пользовательского интерфейса для каждого профиля пользователей.
4. Возможность выпускать функции независимо от последовательности выпусков.
5. Развернутые изменения видны всем клиентам, независимо от того, какую версию приложения они используют.
6. Дает специалистам гораздо больше гибкости и позволяет экспериментировать и выполнять работу гораздо быстрее.
7. Можно деплоить изменения несколько раз в день и быстро откатываться на предыдущую версию, когда что-то пойдет не так.
8. Разработчикам больше не нужно знать принцип работы мобильных платформ.

9. Можно запускать тесты конкретных функций без обновления приложения.
10. Можно создавать больше повторно используемых компонентов.

Недостатки Server-Driven UI

1. Использование уникальных возможностей нативных компонент платформы становится сложной задачей.
2. Приложение уровня предприятия трудно координировать.
3. Снижение общей производительности сложных приложений.
4. «Тяжелые» операции клиента трудно перенести на сервер для обработки.
5. Производительность и стоимость. Ответы могут стать обременительными для сервера при наличии большого количества сложной бизнес-логики. Серверу приходится выполнять всю тяжелую работу, что приводит к увеличению затрат на поддержание работы серверов.
6. Создание составных компонентов, которые позволяют отображать множество возможных типов, приведет к быстрому созданию очень больших запросов от клиентов. Это также снижает общую производительность.
7. Сложность освоения данного подхода — разработчикам, дизайнерам, продукту придется дополнительно обучаться.

При всём этом SDUI нацелен на создание лучшего и более индивидуального опыта для конечных пользователей. Одновременно он предоставляет разработчикам столь необходимую гибкость настройки и гибкость при внедрении. С огромным спектром преимуществ для обеих сторон этот подход так же перспективен, как и технологии, стоящие за ним.

4. **Кроссплатформенность приложений**

Существует несколько видов разработки прикладных приложений:

- Нативный подход
- Кроссплатформенный подход
- Гибридный подход

Кроссплатформенность подразумевает создание приложений, которые могут работать в различных операционных системах.

После написания кода приложения его можно развернуть на разных устройствах и платформах, не беспокоясь о проблемах несовместимости. Это универсальный подход, который широко используется для экономии времени и денег на разработку. Часто для этого используются специализированные кроссплатформенные фреймворки.

Пример такой разработки — применение фреймворка Xamarin для создания приложений, работающих не только на Windows. Благодаря использованию Mono (Open Source реализации платформы .Net), проекты, написанные на C#, успешно запускаются на Unix-like системах: iOs, Android, Linux.

Нативный подход

Нативный подход заключается в использовании принятых для конкретной платформы языков программирования, библиотек, фреймворков или архитектурных паттернов. Пример:

- Swift, SwiftUI для iOS,
- Java и Kotlin для Android.

Плюсы:

- У нативных приложений есть доступ ко всем аппаратным ресурсам, службам и сервисам устройства: камере, микрофону, навигации, акселерометру, календарю, уведомлениям, блокировкам и так далее.
- Нативный подход позволяет получить более отзывчивые и производительные приложения.

Минусы:

- Нужно несколько команд разработки, что требует дополнительного бюджета.
- Некоторые различия в дизайне и возможности внедрения различий в аппаратных возможностях.
- Могут отличаться релизные циклы разных платформ.

Кроссплатформенный подход

Кроссплатформенный подход заключается в разработке исходного кода для нескольких мобильных платформ. Результат каждой отдельной сборки — отдельные исполняемые файлы. Пример:

- Flutter – SDK от компании Google для Android- и iOS-приложений,
- Kotlin Multiplatfrom (KMP) **—** SDK от компании JetBrains.

Плюсы:

- Подход оптимизирует стоимость разработки и поддержки приложения.
- Повышает скорость разработки и возможности масштабирования.

Минусы:

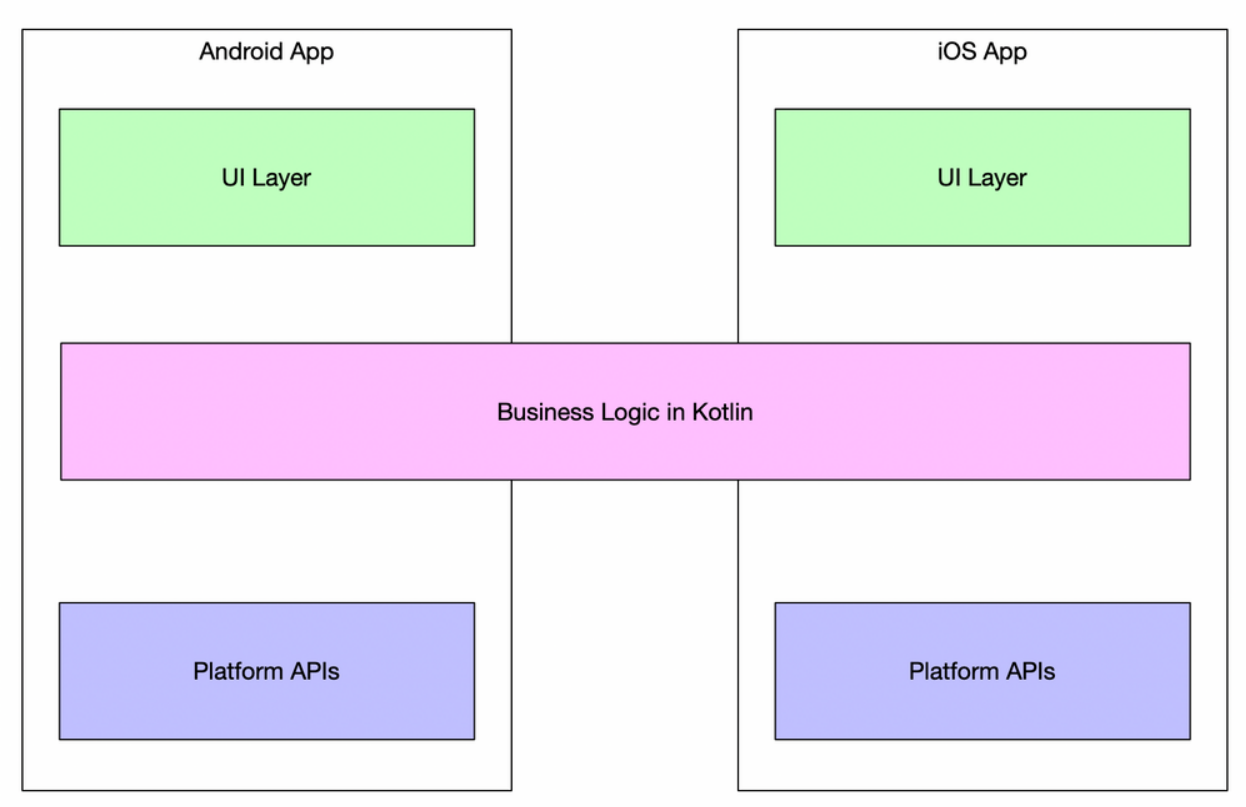
- Зачастую дает более низкую производительность.
- Меньше возможностей интеграции с возможностями устройства.
- Часто снижается качество пользовательского опыта за счёт неоптимальных решений. Ограничена реализация визуальных и графических элементов в приложениях, особенно анимации.

Гибридный подход

Гибридный подход заключается в использовании смеси нескольких языков программирования. Объединяет фреймворки с нативным пользовательским интерфейсом и общим кодом; фреймворки с общей кодовой базой и нативным кодом.

Само приложение может работать как кроссплатформенное, но для некоторых системных функций использует «родной код» системы. Пример:

- React Native,
- Xamarin,
- NativeScript,
- Flutter.



Пример KMP

Глоссарий

Web-приложение — приложение, реализованное с применением web-технологий: JavaScript, HTML, CSS, HTTP и других.

БЭМ (блок, элемент, модификатор) — компонентный подход к веб-разработке. В его основе лежит принцип разделения интерфейса на независимые блоки. Он позволяет легко и быстро разрабатывать интерфейсы любой сложности и повторно использовать существующий код.

Многостраничное приложение — традиционное веб-приложение, в котором с сервера запрашивается новая страница для отображения каждый раз, когда происходит обмен данными туда и обратно. Объем контента, который они несут, огромен, поэтому они, как правило, многоуровневые, со значительным количеством ссылок и сложными пользовательскими интерфейсами.

Одностраничное приложение (англ. single page application, SPA) — это веб-приложение или веб-сайт, использующий единственный HTML-документ как оболочку для всех веб-страниц и организующий взаимодействие с пользователем через динамически подгружаемые HTML, CSS, JavaScript, обычно посредством AJAX.

База данных — это организованная структура, предназначенная для хранения, изменения и обработки взаимосвязанной информации, преимущественно больших объёмов.

Бизнес-правила — совокупность правил, принципов, зависимостей поведения объектов предметной области. Последнее представляет собой область человеческой деятельности, которую система поддерживает.

Компонент — множество классов и языковых конструкций, объединённых по общему признаку и организованных в соответствии с определёнными правилами и ограничениями.

Пользовательский интерфейс — интерфейс, обеспечивающий передачу информации между пользователем-человеком и программно-аппаратными компонентами компьютерной системы.

Data Transfer Object (DTO) — один из шаблонов проектирования. Используется для передачи данных между подсистемами приложения. Data Transfer Object, в отличие от Business Object или Data Access Object, не должен содержать какого-либо поведения.

Object-Relational Mapping (ORM) – технология программирования, которая связывает базы данных с концепциями объектно-ориентированных языков программирования, создавая виртуальную объектную базу данных. Существуют как проприетарные, так и свободные реализации этой технологии.

Model-View-Controller (MVC) — схема разделения данных приложения, пользовательского интерфейса и управляющей логики на три отдельных компонента: модель, представление и контроллер таким образом, что модификация каждого компонента может осуществляться независимо.

Model-View-Presenter (MVP) — шаблон проектирования, производный от MVC. Используется в основном для построения пользовательского интерфейса. Элемент Presenter берёт на себя функциональность посредника (аналогично контроллеру в MVC) и отвечает за управление событиями пользовательского

интерфейса (например, использование мыши) так же, как в других шаблонах обычно отвечает представление.

Model-View-ViewModel (MVVM) — шаблон проектирования архитектуры приложения. Представлен в 2005 году Джоном Госсманом как модификация шаблона Presentation Model. Ориентирован на современные платформы разработки.

Дополнительные материалы

1. Выступление Jimmy Bogard на тему Solid Architecture in Slices not Layers.
2. Статья Component Based Architecture.
3. Статья GUI Architectures.
4. Книга Enterprise monorepo Angular patterns
5. Книга Effective React development with nx
6. Статья «Знакомство с паттерном MVC (Model-View-Controller)»
7. Статья MVPVM Design Pattern - The Model-View-Presenter-ViewModel Design Pattern for WPF

Используемые источники

1. Архитектурные паттерны в iOS. [Электронный ресурс] – Режим доступа: https://habr.com/ru/company/badoo/blog/281162/ – (дата обращения: 27.05.2021)
2. Мартин Р. Чистая архитектура. Искусство разработки программного обеспечения. — СПб.: Питер, 2018. — 352 с.: ил. — (Серия «Библиотека программиста»).
3. Знакомство с паттерном MVC. [Электронный ресурс] – Режим доступа: https://javarush.ru/groups/posts/2536-chastjh-7-znakomstvo-s-patternom-mvc-model-view-controller– (дата обращения: 27.05.2021)
4. Model-View-Controller. [Электронный ресурс] – Режим доступа: https://developer.apple.com/library/archive/documentation/General/Conceptual/DevPedia-CocoaCore/MVC.html – (дата обращения: 27.05.2021)
5. Архитектурные паттерны в iOS: страх и ненависть в диаграммах. MV(X). [Электронный ресурс] – Режим доступа: https://habr.com/ru/company/croc/blog/549590/ – (дата обращения: 27.05.2021)
6. Разработка мобильных приложений: почему следует начинать с минимально жизнеспособного продукта (MVP). [Электронный ресурс] – Режим доступа: https://smartum.pro/ru/blog-ru/mvp-v-razrabotke-mobilnykh-prilozheniy/– (дата обращения: 28.05.2021)
7. Сравнение архитектур Viper и MVVM: Как применить ту и другую. [Электронный ресурс] – Режим доступа: https://habr.com/ru/post/440904/ – (дата обращения: 28.05.2021)
8. Введение в VIPER. [Электронный ресурс] – Режим доступа: https://habr.com/ru/post/273061/ – (дата обращения: 28.05.2021)
9. Разбор архитектуры VIPER на примере небольшого iOS приложения на Swift 4. [Электронный ресурс] – Режим доступа: https://habr.com/ru/post/358412/ – (дата обращения: 28.05.2021)

10. Почему VIPER это хороший выбор для вашего следующего приложения. [Электронный ресурс] – Режим доступа: https://habr.com/ru/post/321590/ – (дата обращения: 28.05.2021)

11. Clean swift архитектура как альтернатива VIPER. [Электронный ресурс] – Режим доступа: https://habr.com/ru/post/415725/ – (дата обращения: 28.05.2021)

12. Clean Architecture https://blog.cleancoder.com/uncle-bob/2012/08/13/the-clean-architecture.html

13. Чистая архитектура на Android и iOS. [Электронный ресурс] – Режим доступа: https://apptractor.ru/develop/chistaya-arhitektura-na-android-i-ios.html – (дата обращения: 29.05.2021)

14. Model-View-Controller. [Электронный ресурс] – Режим доступа: https://wiki2.org/ru/Model-View-Controller – (дата обращения: 04.06.2021)

15. Model-View-Presenter. [Электронный ресурс] – Режим доступа: https://google-info.org/1331436/1/model-view-presenter.html – (дата обращения: 04.06.2021)

16. MVC — MVP — MVVM. [Электронный ресурс] – Режим доступа: https://nurlandroid.com/?p=366 – (дата обращения: 04.06.2021)

17. Model-View-Intent(MVI) in Android. [Электронный ресурс] – Режим доступа: https://www.novatec-gmbh.de/en/blog/mvi-in-android/ – (дата обращения: 04.06.2021)

18. Статья «Как выбрать мобильную кросс-платформу в 2021 году»

19. Статья Server Driven UI, Part 1: The Concept

20. Статья «Обновить UI за час: опенсорс фреймворк для быстрой разработки мобильных приложений Divkit»

21. Методология БЭМ

22. Статья Web frameworks, 2020 Developer Survey, Stackoverflow

23. Статья Web Components, MDN

24. Статья «Чем отличаются сайт и веб-приложение?»