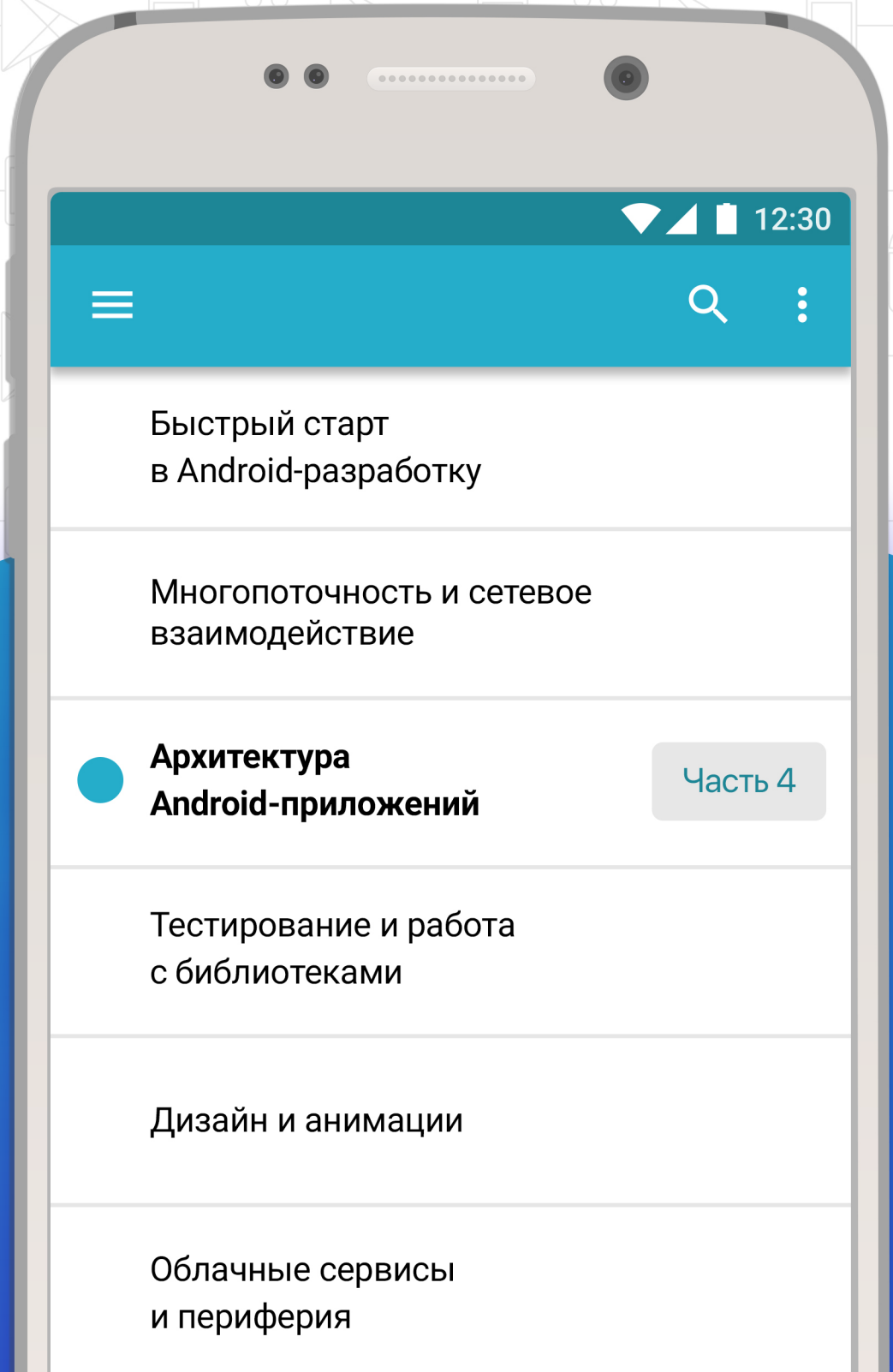
фонд развития
онлайн образования
eldf.ru

e·legion

academy.e-legion.com

Программа Android-разработчик

Конспект



Оглавление

Глава 1

НЕДЕЛЯ 4

1.1. Clean Architecture

1.1.1. Clean Architecture

Привет! Вы уже прекрасно знакомы с тем, как грамотно реализовать View-слой в любой архитектуре. Но как быть со слоем Module? Да, мы можем создать кучу утильных классов, сделать data-manager и там описывать всю бизнес-логику. Но кому понравится смотреть data-manager, который содержит огромную кучу плохо навигируемого кода. Тут нам на помощь приходит Clean Architecture, которая раскладывает всю бизнес-логику и взаимодействие с ней по полочкам. Давайте начнем.

В 2012 году Uncle Bob – Роберт Мартин – дядюшка Боб - опубликовал статью для Clean Architecture, которая и является основным описанием этого подхода. Разберем все термины, которые используются в этой диаграмме. Entities – Сущности – это бизнес-логика приложения. Use Cases – методы использования – эти методы организуют поток данных в Entities и из них. Также их называют interactors – посредники. Interface Adapters – Интерфейс-Адаптеры – этот набор адаптеров преобразует данные в формат, удобный для методов использования и сущностей. К этим адаптерам принадлежат презентеры и контроллеры. Frameworks and Drives – место скопления деталей – это у нас UI, инструменты, фреймворки, базы данных и так далее. Теперь к сути. Clean Architecture – это идеология, такая же, как и Material Design. Ее суть заключается в построении правильных зависимостей между слоями. Это называется Dependency Rule. Каждый слой архитектуры должен иметь только внутренние зависимости и ни одной внешней. Это касается функций, классов, переменных и так далее. Это правило позволяет строить системы, которые проще будет поддерживать, потому что изменения во внешних слоях не затронут внут-

ренние слои. На диаграмме изображены четыре круга, то есть четыре слоя архитектуры. Слоев не обязательно должно быть четыре: вы можете добавлять либо убирать слои на свое усмотрение, как вам будет угодно. У каждого слоя должна быть своя конкретная ответственность и он не может делать сразу всю работу приложения.

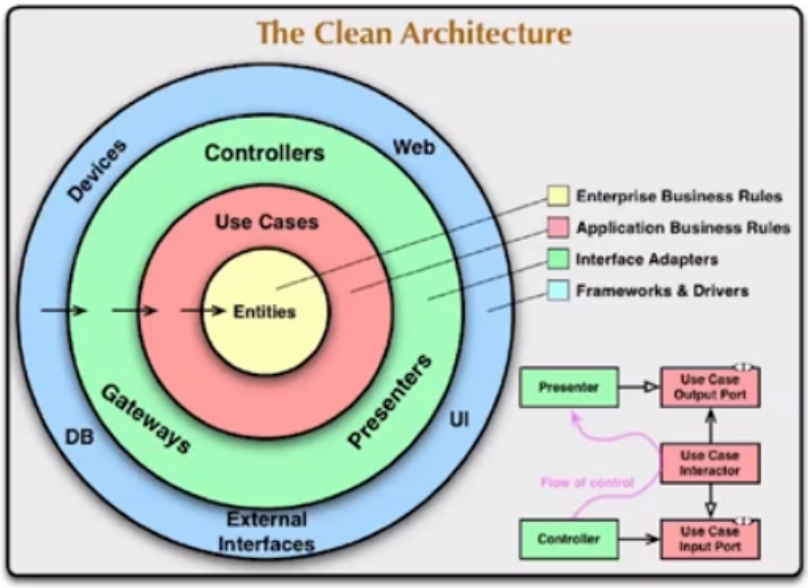


Рис. 1.1: Диаграмма

Clean Architecture объединила в себе идеи нескольких других архитектурных подходов, которые сходятся к тому, что архитектура должна:

- Быть тестируемой. Бизнес-правила могут быть протестированы без пользовательского интерфейса, базы данных, веб-сервера или любого другого внешнего компонента;
- Не зависеть от UI. Пользовательский интерфейс можно легко изменить, не изменяя остальную систему, например веб-интерфейс может быть заменен на консольный без изменения бизнес-правил;
- Не зависеть от БД. Вы можете поменять Oracle или SQL-сервер на MongoDB, Bigtable или что-то там еще. Ваши бизнес-правила не связаны с базой данных;

- Не зависеть от внешних фреймворков и библиотек. Архитектура не зависит от существования какой-либо библиотеки. Это позволяет использовать фреймворк в качестве инструмента, вместо того, чтобы втискивать свою систему в рамки его ограничений;
- Не зависеть от какого-либо внешнего сервиса. По факту ваши бизнес-правила ничего не знают о внешнем мире.

С теорией покончено. Надеюсь, суть вы уловили. Теперь перейдем к практике. Clean Architecture получили широкое распространение в Android после того как парень по имени Fernando Cejas выпустил в свет статью «Architecting Android...The clean way?» В ней он описывает примеры реализации идеологии Clean Architecture в Android. В этой статье Фернандо приводит такую схему слоев. То, что на этой схеме другие слои, а в domain-слое лежат еще какие-то boundaries, сбивает с толку. Проект разбит на три слоя. Это слой данных, Data layer, слой бизнес-логики, Domain Layer, слой представления, Presentation Layer. Каждый из этих слоев имеет свою цель и может работать независимо от остальных. Давайте разберем их.

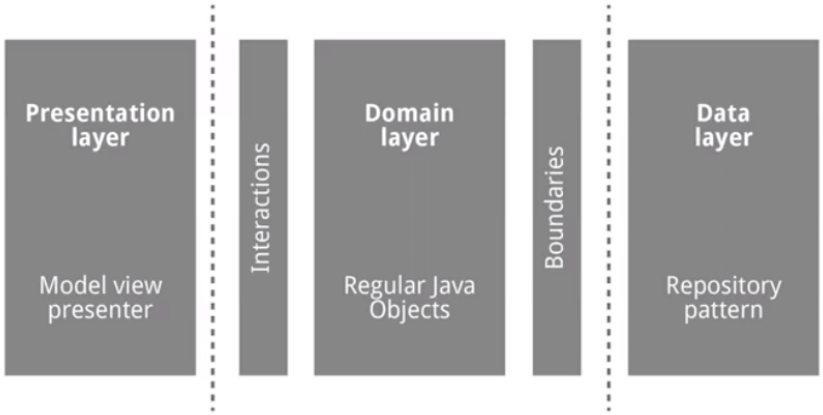


Рис. 1.2: Схема слоев

Тут все просто. Presentation – слой, в котором вы реализуете один из MV-паттернов на свой выбор. Что хотите, то и используйте: MVP, MVVM, любые другие. Фрагменты и Activity – это всего лишь View. В них нет никакой логики кроме логики UI и отрисовки этого самого отображения. Domain-слой – модуль на чистой Java без каких либо Android-зависимостей. Все внешние компоненты используют интерфейсы для связи с бизнес-объектами. Вся логика реализована в этом слое, тут у нас хранятся классы, с которыми мы работаем внутри приложения, различные интерфейсы и интеракторы, или, по-другому, use-кейсы нашего приложения. Интеракторы,

выражаясь простым языком, представляют из себя класс, который выполняет одно конкретное действие над данными, например, у нас есть объект класса User, нам нужно отредактировать его. Для этого создается интерактор, User-интерактор, внутри этого класса создается метод editUser и пишется обработка нашего юзера. Все данные, необходимые для приложения поставляются из Data-слоя через реализацию репозитория, который использует репозиторий-паттерн со стратегией, которая через фабрику выбирает различные источники данных в зависимости от определенных условий. Интерфейс находится в Domain-слое. Наверняка многие из вас слышали и даже использовали этот паттерн. На самом деле здесь у вас есть несколько вариантов того, как вы реализуете работу с различными хранилищами данных. Вы можете добавить логику работы с разными репозиториями, сервисы или интеракторы. Если же эта логика останется в репозиториях, тогда вам нужно будет просто обрабатывать данные в интеракторах.

Теперь давайте рассмотрим, как эти слои реализуются в Android-приложениях. Пряморукие и грамотные специалисты делают это вот так. Просто создают зависимости на модули, как указано на схеме, то есть domain не знает ни об одном модуле. Data знает только о Domain. Presentation знает и о Data, и о Domain. Все просто. Модуль Domain – это Java-модуль, Data и Presentation – это андроид-модули. Также все эти зависимости между модулями нужно прописать в gradle, как показано на схеме. Такая жесткая установка зависимостей и разделение модулей на Java и Android позволит не ошибиться неопытному специалисту. Если вы захотите написать Android-код Domain, у вас ничего не выйдет. Даже ничего не выйдет, если вы из Domain захотите обратиться к Data или Presentation-слою. Лично мое мнение, что этот подход красив и безопасен.

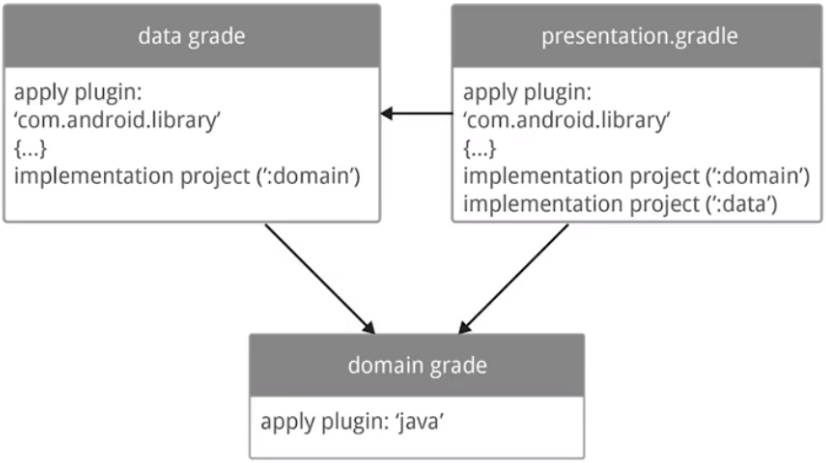


Рис. 1.3: Схема

Теперь давайте посмотрим, как гуляют данные по приложению в Clean Architecture, Допустим, мы нажали на кнопочку или просто зашли на экран и запрашиваем получение данных. Наш запрос из View идет в Presenter. В Presenter'е мы вызываем необходимый Use Case или интерактор. Он выполняет всю необходимую работу над данными, идет в Service, опять выполняет работу над данными, если необходимо. Service по своей логике определяет, с каким репозиторием или репозиториями он работает. И репозиторий уже предоставляет нам результаты запроса и все возвращается обратно. Схема проста, но напомню, что вы можете отказаться от каких-либо слоев и добавлять свои. Эта схема – всего лишь классический пример.

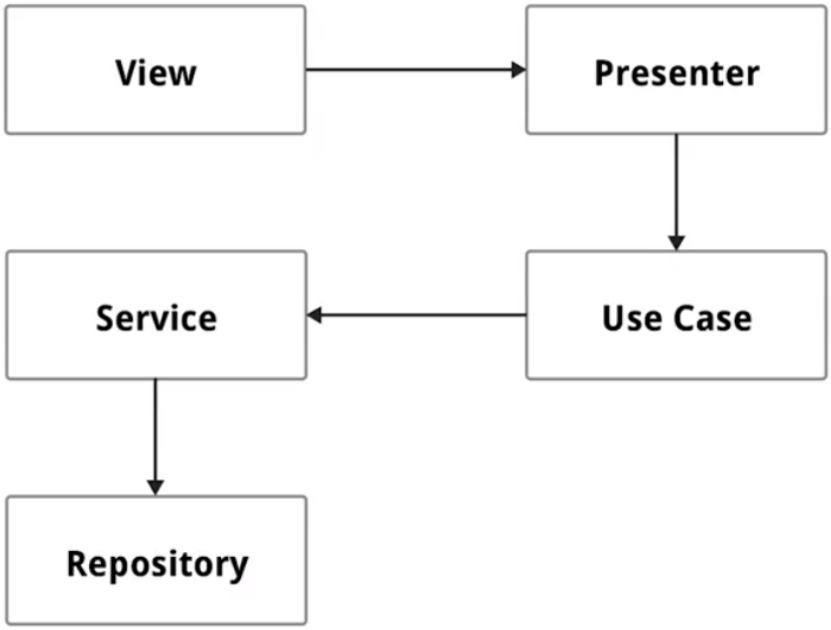


Рис. 1.4: Схема

Теперь давайте посмотрим немного кода. Вот так выглядит интерфейс интерактора. Все просто, есть интерфейс callback'а, который будет сообщать Presenter'у о результате, когда вернется ответ от репозитория и сервиса. В данном примере это может быть ошибка или успех. Есть метод execute, который соответственно нужно реализовать. Сейчас посмотрим, как это делается.

```
1  public interface Interactor<P, R> {
2
3      interface Callback<R> {
```

```
        void onSuccess(R object);
        void onError(Throwable throwable);
    }
    void execute (P parameter, Callback<R> callback);

}
```

Вот так работает Use Case. Это пример реализации метода execute. На вход он принимает какие-то данные и callback для response'а. В данном примере используется не самый красивый метод обработки ошибки. Лучше всего его переделать, потому что в каждом Use Case писать такой boiler plate – не лучшая практика. В наш успешный callback засовываем объект, который мы получаем. Далее он идет в Presenter – простая схема.

```
@Override
public void execute(final Integer feedId,
        final Callback<List<Article>> callback) {
    try {
        callback.onSuccess(feedRepository.getFeedArticles( FeedId));
    } catch (final Throwable throwable) {
        callback.onError(throwable);
    }
}
```

А теперь представьте, что вы пишете приложение, в котором как минимум 20 разных объектов, которые требуют обработки. С этими объектами нужно сделать кучу разных операций: сортировка, добавление, изменение каких-либо полей и так далее. Из этого получается 100 с лишним интерактоpов. Еще представьте, что вам нужно использовать 15-20 интерактоpов в одном Presenter'е. Представили? Это боль. Все это делает проект нереально огромным. Это неудобно. Уверен, что все, кто пользовался таким подходом, отказались от него и пришли к чему-то иному.

Так же поступили и мы. Гораздо удобнее объединять наши интерактоpы в один класс со своими методами для обработки данных. Это простой пример, как будут выглядеть методы внутри репозитория, который работает с разными типами хранилищ.

```
@Override
public Single<List<Post>> get(int offset, int count) {
    return mApi.getPosts(offset, count);
}
```

```
5
6   @Override
7   public Completable updateNotifications(String deviceId) {
8       String ids = mSharedPreferences.getString(SETTINGS_KEY, "0");
9       if (TextUtils.isEmpty(ids)) {
10          ids = "0";
11      }
12      return mApi.enableNotifications(deviceId, ids);
13  }
```

А вот так может выглядеть наше объединение интеракторов в один класс Interactor.

```
1   @Override
2   public Single<List<Post>> get(int offset, int count) {
3       return mPostServerRepository.getPosts(offset, count);
4   }
5
6   @Override
7   public Completable updateNotifications(String deviceId) {
8       String ids = mPostLocalRepository.getString(SETTINGS_KEY, "0");
9       if (TextUtils.isEmpty(ids)) {
10          ids = "0";
11      }
12      return mPostServerRepository.enableNotifications(deviceId, ids);
13  }
```

На этом все. От себя хотелось бы добавить, что лично я для себя выделил одно правило, к которому я пришел опытным путем: вы можете делать все что угодно со своим кодом и архитектурой. Главное, чтобы это было максимально удобно и красиво, но при этом всегда стоит задумываться о том, на какие грабли вы можете наткнуться и что может пойти не так. В первую очередь думайте о минусах и придумывайте, как их превратить в плюсы.

1.1.2. Плюсы и минусы Clean Architecture

Давайте рассмотрим плюсы и минусы Clean Architecture. Плюсы:

1. Модульное разделение. Благодаря этому можно не бояться за то, что неопытный разработчик напишет Android-код в domain слое. Также мы можем легко заменить любой из модулей на другой и это не вызовет никаких проблем;
2. Правило зависимостей, благодаря которому можно реализовать действительно чистый код;
3. Тестирование. Благодаря первым двум плюсам мы можем легко протестировать каждый модуль отдельно, что очень важно в крупных проектах;
4. Наличие Domain слоя, в котором мы пишем логику на чистой Java без зависимости от фреймворка;
5. Также вы можете использовать любой понравившийся вам MV-паттерн для UI-слоя;
6. Кастомизация. Вы можете кастомизировать вашу архитектуру так, как душе угодно, главное соблюдать Dependency Rule;
7. Хорошее комьюнити. Данная архитектура обладает большой аудиторией, которая всегда поможет, подскажет, если у вас возникнут проблемы;
8. Гайды и богатая база исходных кодов с примерами являются немаловажным фактором, благодаря этому вы можете легко освоить эту архитектуру самостоятельно или посмотреть, как реализовали некоторые кейсы другие люди.

Теперь к минусам.

1. Громоздкость. В большом проекте количество классов может вырасти до нереально больших размеров, из-за чего будет достаточно неудобно ориентироваться в проекте;
2. Порог вхождения. Для того, чтобы проникнуться Clean Architecture и ее особенностью, нужно немало поработать. Написать парочку своих проектов, изучить DI, реактивное программирование, разобраться с Dependency Rule и тем, как правильно разделять приложение на модули. Это уже немало.

Теперь давайте подведем итоги. Когда нам следует воспользоваться Clean Architecture? Если ваше приложение подразумевает наличие большого количества бизнес-логики, то для того, чтобы у вас не получилась каша, следует использовать Clean Architecture. Если ваше приложение разрастается до размеров хотя бы десяти экранов, то стоит задуматься о том, что работа с данными в проекте должна выглядеть красиво, соответственно для этого можно использовать Clean

Architecture. Заказчик требует покрыть все тестами? Вы хотите обезопаситься? Благодаря DI и Clean Architecure это можно сделать легко и просто, самое главное качественно. Если в планах есть расширять или поддерживать приложение, то с этой архитектурой это можно будет реализовать гладко и без особых проблем. Если приложение пишется на разные платформы, к примеру Android, iOS, Web, Desktop, то можно просто взять Domain-слой и внедрить его в другую архитектуру, при необходимости переписав на другой язык.

Теперь о том, когда следует воздержаться. Если у вас простое приложение, которое требует вводить данные с сервера, нет базы данных, не нужно мучиться с маппингом данных, не надо эти данные изменять или обрабатывать, то можно просто использовать какой-нибудь data-manager, или утильные классы, или что-то похожее. Я надеюсь, это поможет вам определиться с выбором.

1.1.3. Создание модулей data, domain и presentation

В данном занятии мы начнем с вами работать с Clean Architecture. В нем мы будем создавать три модуля. Data, Domain, Presentation. Перейдем к делу.

Зайдем в проект. Перейдем в Project Scope. Создадим новый модуль. Next. Называем его data. Выставляем минимальную версию SDK, в нашем случае это 15, точно такая же, как была до этого. Жмем Finish. Если гид спросит, добавлять ли файлы, пишем, да, добавлять. Далее создадим еще один модуль. Но уже Java Library. В нашем случае он будет называться domain. Жмем Finish. Подождем, пока студия сбилдит нам проект, выполнит необходимые операции. Далее сами жмем Sync на всякий случай, потому что студия не всегда видит с первого раза новые модули, только что добавленные.

Переходим в Android Scope, модуль app переименовываем в presentation. Теперь переходим в Gradle Scripts, build.gradle (Module:presentation). Здесь скопируем абсолютно все зависимости, которые есть в этом модуле. Вообще есть несколько способов разрешения зависимостей, но мы будем использовать самый простой, но не самый красивый. Как это сделать лучше и красивее, вы узнаете в следующем уроке. Скопировали все зависимости, переходим в build.gradle (Module:domain). Здесь удаляем абсолютно все, что связано с андроидом. Android test implementation также убираем. Все что связано с сетью мы тоже убираем, это Java Library, она вообще ничего не знает о том, как работать с сетью и с базой данных. Поэтому зависимость на Room мы сотрем, так же, как сотрем зависимость на Picasso. Далее переходим в module data. Здесь повысим compileSdkVersion и targetSdkVersion до 27. Далее вставим все зависимости, которые мы копировали до этого и удалим все зависимости, которые связаны с ui. В нашем случае это Picasso, RecyclerViewConstraintLayout и AppCompat. Добавим одну важную строчку, за-

висимость implementation project на domain. Скопируем ее и перейдем в build.gradle (Module: presentation). Здесь у нас уже будет две зависимости на domain и на data, ведь вы знаете, что domain не знает ничего ни об одном слое, data знает только о domain'е, а presentation знает и о data и о domain. Перейдем чуть выше, вырежем все build config field'ы из presentation и вставим их в data. Нажмем Sync now. Все верно.

Также удалим annotationProcesor из домена, так как это java library и android. Синхронизация прошла успешно, переходим в module presentation и вырезаем абсолютно все то, что есть в package data. Вырезали и вставляем в package com.elegion.data. Жмем Refactor. Удалим package из module presentation. Сделаем build, посмотрим, какие ошибки он нам покажет, а они точно будут, ведь рефакторинг никогда не бывает безошибочным. Он не видит данный package. Сделали автоимпорт, теперь наш build config из presentation заменился на data. Сделаем build еще раз. Не может найти API_QUERY, все верно, ведь мы убрали эти поля из build config'a presentation'a. Теперь они у нас находятся в data, как я предположил.

Жмем build и посмотрим, все ли работает верно. Проект успешно собрался. Теперь давайте запустим его на эмуляторе и посмотрим, ничего ли мы не сломали. Как вы видите, проект прекрасно работает. В данном занятии мы начали работать с Clean Architecture и успешно добавили три модуля. Это domain, data и presentation.

1.1.4. Создание ProjectServer/ProjectDBRepository

В данном занятии мы добавим с вами репозиторий для работы с моделью проекта. Давайте перейдем к делу, зайдем в module domain и создадим тут новый package, назовем его repository, создадим в нем новый интерфейс, назовем его ProjectRepository. Далее выберем интерфейс, нажмем OK.

Добавим сюда два метода. Первый будет возвращать Single<Project>. Как вы видите, module domain не видит объекты project. Все верно, потому что domain ничего не знает о data, а модели хранятся именно там. Но почему, спросите вы. Давайте разберемся. Перейдем в module data, java, com.elegion.data. Откроем model, project и посмотрим класс Project. Здесь вы увидите, что у нас используются аннотации из библиотеки Room. Почему? Потому что мы решили объединить обычные модели с моделью базы данных. Так удобнее. И не нужно ничего парсить, писать никаких сериалайзеров, десериалайзеров.

Как же нам перенести эти модели в домен-слой? Легко. Мы можем совершить очень плохой поступок – добавить зависимость на Room в module domain. Давайте этим займемся. Перейдем в

Gradle Scripts. build.gradle (Module: data). Скопируем зависимость на Room, вставим ее в домен, нажмем Sync Now. Синхронизация прошла успешно. Теперь перейдем в data-слой, вырежем package model и вставим его в домен. Далее можно удалить My Class. Это дефолтный класс, который создался при создании модуля. Проверим, все ли верно у нас работает, нет ли каких ошибок. Как вы видите, никаких ошибок нет, наши модели успешно перенеслись в другой слой.

Как вы можете заметить, теперь домен видит объект Project. Назовем наш метод getProjects, который ничего не будет принимать. Следующим методом, который мы добавим, будет void insertProjects. Это для сохранения моделей в базу данных, который будет принимать в себя список проектов projects. Далее переходим в модуль data, создадим здесь новый package, который назовем repository. New, Java Class, ProjectServerRepository. Enter. implements ProjectRepository. Заимплементим методы. Также он будет принимать в себя BehanceApi mApi. Добавим аннотацию Inject, обязательно создадим пустой конструктор. Alt+Insert, конструктор, пустой конструктор. Конструктор, Select None. Далее в getProjects добавляем mApi.getProjects(BuildConfig.API_QUERY точка map new function, который будет возвращать нам projectResponse.getProjects. Что-то пошло не так, давайте посмотрим, что. Он просит вернуть нам Project, а должен быть List проектов. List<Project>. Так же, как и в нашем интерфейсе, потому что мы будем возвращать список проектов, а не один проект. Соответственно, стираем этот метод и напишем его по-новой. map, new, fuction. projectResponse.getProjects. Теперь все работает верно.

insertProjects. На сервере нет функционала сохранения проектов, поэтому do nothing. Скопируем его и создадим database, ProjectDBRepository. Код мы будем брать из Storage, соответственно insertProjects, assemble и getProjects можно скопировать и вставить в ProjectDBRepository. OK. getProjects просто вырезаем и вставляем в наш getProject, только вместо return response у нас будет возвращаться return projects, который будет обернут в return Single.fromCallable, new Callable. Также нам нужен будет BehanceDao. Добавим его сюда, BehanceDao. Зависимость в модуле мы добавим чуть позже. List Project, Single List Project. Почему-то ему не нравится, что мы хотим сделать такой return, потому что у нас есть второй метод getProjects, который нужно просто удалить.

Далее в insertProjects просто копируем весь код, вставляем его сюда, удаляем первую строчку, и все будет работать. Теперь давайте добавим зависимость в модуль для того, чтобы у нас был BehanceDao. Откроем presentation, java, di, AppModule и здесь добавим даже две зависимости, одна будет возвращать BehanceDatabase, provideDatabase. Другая будет возвращать BehanceDao provideBehanceDao. Здесь мы будем делать return Room.databaseBuilder'a. Здесь мы будем получать BehanceDatabase в качестве входного параметра и здесь мы будем возвращать database.getBehanceDao. Здесь мы тоже будем получать BehanceDatabase, даже можно здесь получать BehanceDao, будем возвращать new Storage, которому мы передадим наш dao.

Теперь давайте попробуем сбилдить наш проект. Проект успешно сбилдился, добавим аннотации Inject над нашими конструкторами репозиториев, чтобы Dagger их правильно воспринимал. Теперь давайте добавим наши репозитории в module, чтобы мы могли их инжектить. Перейдем в presentation, di и создадим New module или лучше скопируем существующий. NetworkModule переименуем в RepositoryModule. Здесь для примера оставим одну зависимость и будем возвращать ProjectRepository обязательно с аннотацией Named и в ProjectRepository добавим public static final String, который будет называться SERVER = "SERVER" и DB = "DB". Чтобы мы могли использовать разные реализации одного и того же интерфейса нам необходима эта аннотация, поэтому мы будет различать ее по name'у Server и name'у Db. Далее, перейдем обратно в RepositoryModule и в Named добавим ProjectRepository.SERVER, provideServerProjectRepository или ProjectServerRepository, если точнее. Здесь мы просто будем возвращать new ProjectServerRepository. Далее, копируем это и SERVER меняем на DB, ProjectDBRepository. Здесь будет возвращаться return new ProjectDBRepository. Добавим наш RepositoryModule в AppComponent. RepositoryModule.class. Сбилдим для того, чтобы убедиться, что dagger не выявил никаких ошибок. Все сбилдилось успешно. В данном занятии мы добавили репозиторий для работы с моделью Project.

1.1.5. Создание ProjectService

В данном занятии мы создадим с вами сервис для работы с project-репозиториями, чтобы логика работы с базой данных и с репозиторием сервера хранилась не в Presenter'е, как это сейчас происходит, а хранилась в Service. Поехали.

Открываем проект. Идем в модуль domain. New Package service. Enter. Создадим новый интерфейс, назовем его ProjectService. Заметим, что он Interface. Нажмем OK. Теперь перейдем в ProjectRepository. Уберем public static final, потому что в интерфейсе переменные по умолчанию и так final static final. Скопируем отсюда оба этих метода. Далее в package service создадим новый Java Class, назовем его ProjectServiceImpl и укажем ему Interface, который он должен реализовать ProjectService. Жмем OK. Далее реализуем оба этих метода и добавляем два поля. Inject, Named. Первое поле у нас будет ProjectRepository, назовем его mServerRepository. В Named обязательно укажем ProjectRepository.SERVER. Скопируем, здесь поменяем на DB. mDBRepository. Добавим ему конструктор без входных параметров и добавим аннотацию Inject.

Далее. В методе getProjects мы должны описать логику взаимодействия с репозиторием базы данных и с репозиторием сервера. Эту логику мы возьмем из ProjectPresenter'а. Переходим в presentation, ui, ProjectPresenter, копируем все, что тут есть, пока ничего не удаляем, вставляем

в getProjects. CompositeDisposable нам не нужен, mApi в нашем случае это ServerRepository. Это просто стираем. В случае don't success'а мы должны сохранить их базы данных, то есть mDBRepository::insertProjects. Далее в случае возврата ошибки нам нужны network exceptions, которые сейчас хранятся в presentation-слое. Давайте перенесем ApiUtils в domain-слой.

Далее. Вместо mStorage у нас теперь есть mDBRepository, но он возвращает single, что делать в этом случае? А делать вот что, и тогда мы можем пользоваться костылями, мы просто будем использовать метод blockingGet, то есть блокирующий получение.observeOn, doOnSubscribe и subscribe мы стираем, нам это не нужно. return null мы также убираем. Здесь мы делаем return mServerRepository.getProjects. Далее в insertProjects мы просто делаем mDBRepository.insertProjects, передаем сюда projects, которые должны быть здесь.

Теперь добавим зависимости в модули. Создадим новый модуль, скопируем RepositoryModule, просто переименуем его в ServiceModule. Далее сотрем одну зависимость, уберем Named. Далее здесь добавим ProjectService, Repository уберем. provideProjectService. В качестве входного параметра ему можно будет передать ProjectService simple. Dagger автоматически создаст нужную зависимость и сам ее предоставит, вместо того, чтобы постоянно писать new. То же самое мы должны сделать в RepositoryModule, так как сейчас мы просто создаем new. Соответственно здесь у нас будет ProjectServerRepository repository. Здесь у нас будет ProjectDBRepository. Нажимаем Ctrl+S и переходим в ProjectsPresenter. Стираем mApi и mStorage. Сюда добавляем наш сервис ProjectService mService. Далее, Ctrl+C, mService вставляем сюда. doOnSuccess нам теперь не нужен. onErrorReturn на также не нужен. Теперь в качестве response у нас возвращается непосредственно List<Project>. Теперь мы можем избавиться от Storage, он нам больше не нужен, жмем OK, Delete Anyway, попробуем сбилдить проект, почистим ошибки, которые возникли при удалении Storage.

Cannot find symbol class Storage, он нам и не нужен, потому что он больше не используется. provideStorage тоже не нужен, так же как и тут. Попробуем сбилдить еще раз, вдруг Dagger укажет нам на какие-то ошибки. Все верно, какую-то ошибку он нашел. ProjectService cannot be provided without an @Provides-annotated method. Перейдем в ProjectService, ProjectServiceImpl. Здесь аннотация стоит, почему же он не может заинжектить наш ProjectService? Потому что мы скорее всего где-то допустили ошибку в модуле. Действительно, мы забыли добавить наш модуль в AppComponent. Давайте исправим это. RepositoryModule, ServiceModule.class. Билдим. Все успешно сбилдилось.

Теперь давайте попробуем запустить проект на эмуляторе. Как вы видите, все работает. В данном занятии мы успешно завершили работу с Clean-архитектурой. В нем мы узнали, как работать с модулями, как их создавать, какие ошибки можно допускать, как их исправлять.

1.1.6. Заключительное видео

Архитектурный блок окончен. Поздравляю! Давайте вспомним, что мы проходили. Мы изучили MVP и MVVM, два самых распространенных паттерна организации UI-слоя. Узнали об их минусах и плюсах, попробовали в проекте. На мой взгляд, MVP немного проще для восприятия, а MVVM с Databinding органичнее вписывается в Android-фреймворк. В MVP мы использовали Moxy для состояния. В MVVM для этой цели использовались архитектурные компоненты. Также мы изучили механизм внедрения зависимостей и специальные библиотеки для этого: Dagger2 и Toothpick. Отличное дополнение к вашему резюме. В конце концов мы познакомились с подходом CLEAN, то есть с чистой архитектурой, узнали о слоях, правиле зависимостей, а также попрактиковались в его создании и написании. Конечно, в рамках одного блока сложно расписать все нюансы, которые возникают в работе, но вы можете обратиться к нам в чате.

Что дальше? В прошлом блоке я просил вас начать работать над приложением для портфолио. Надеюсь, у вас появились кое-какие наработки. Теперь же я прошу вас применить новые знания и сделать ваш проект красивым и правильным в архитектурном плане. Удачи и до скорой встречи в следующем блоке, посвященному тестированию.

О проекте
Академия e-Legion – это образовательная платформа для повышения квалификации в мобильной разработке. Слушайте лекции топовых разработчиков, выполняйте практические задания и прокачивайте свои скиллы. Получите высокооплачиваемую профессию – разработчик мобильных приложений.

Программа "Архитектура Android-приложений"

Блок 1. Быстрый старт в Android-разработку

- Описание платформы Android
- Знакомство с IDE — Android Studio и системой сборки — Gradle
- Дебаг и логгирование
- Знакомство с основными сущностями Android-приложения
- Работа с Activity и Fragment
- Знакомство с элементами интерфейса — View, ViewGroup

Блок 2. Многопоточность и сетевое взаимодействие

- Работа со списками: RecyclerView
- Средства для обеспечения многопоточности в Android
- Работа с сетью с помощью Retrofit2/Okhttp3
- Базовое знакомство с реактивным программированием: RxJava2
- Работа с уведомлениями
- Работа с базами данных через Room

Блок 3. Архитектура Android-приложений

- MVP- и MVVM-паттерны
- Android Architecture Components
- Dependency Injection через Dagger2
- Clean Architecture

Блок 4. Тестирование и работа с картами

- Google Maps

- Оптимизация фоновых работ
- БД Realm
- WebView, ChromeCustomTabs
- Настройки приложений
- Picasso и Glide
- Unit- и UI-тестирование: Mockito, PowerMock, Espresso, Robolectric

Блок 5. Дизайн и анимации

- Стили и Темы
- Material Design Components
- Анимации
- Кастомные элементы интерфейса: Custom View

Блок 6. Облачные сервисы и периферия

- Google Firebase
- Google Analytics
- Push-уведомления
- Работа с сенсорами и камерой