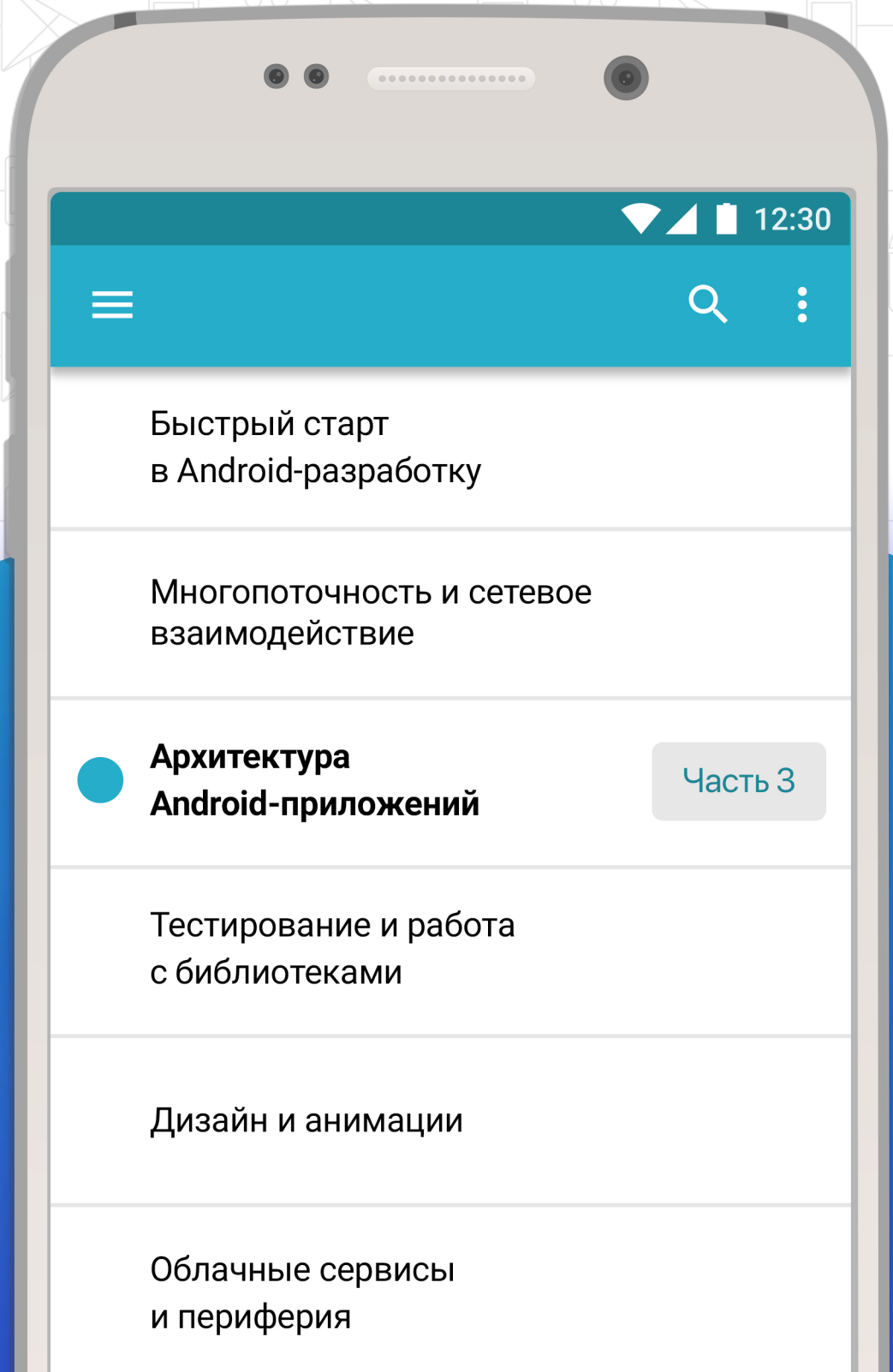
фонд развития
онлайн образования
eldf.ru

e·legion

academy.e-legion.com

Программа Android-разработчик

Конспект



Оглавление

Глава 1

НЕДЕЛЯ 3

1.1. Dependency Injection

1.1.1. Dependency Inversion и Inversion of Control

Привет. На этой неделе мы изучим сложный материал, посвященный внедрению зависимостей. Но такого зверя просто так не одолеть, поэтому мы начнем немного издалека. Сейчас попробуем разобраться, в чем разница между принципом инверсии зависимостей и инверсией управления.

Dependency Inversion – инверсия зависимостей – это пятый по счету из принципов объектно-ориентированного проектирования SOLID. Напомню формулировку:

- Модули верхних уровней не должны зависеть от модулей нижних уровней. Оба типа модулей должны зависеть от абстракций;
- Абстракции не должны зависеть от деталей. Детали должны зависеть от абстракций.

Как мы уже выяснили ранее, практическое использование принципа в том, чтобы сделать классы зависимыми от абстракций. Но самое главное, самое важное – зависимости должны быть на том же или на вышестоящем уровне абстракции, то есть не всякая переданная зависимость является правильной реализацией принципа инверсии зависимостей.

Выражаясь очень простым языком, зависимости класса должны быть интерфейсами или абстрактными классами. Например, внутри класса Car есть зависимость Engine. Engine – это интерфейс. Получается, что класс Car, во-первых, работает только с объявленными в интерфейсе

методами, то есть не зависит от деталей того или иного двигателя, и, во-вторых, может получить любую реализацию интерфейса Engine, что в идеале дает нам бесчисленное множество машин с различными двигателями – бензиновым, дизельным, электрическим, гибридным или даже на основе воды. То есть, иными словами, наша машина потенциально уже сейчас может работать с еще даже не изобретенным двигателем. И важный пункт. Engine – это двигатель, абстракция. Если бы мы передавали в класс Car низкоуровневую зависимость, например, материал проводки, то принцип инверсии зависимостей был бы нарушен, так как появилась бы зависимость от деталей.

Перейдем к более общему понятию – Inversion of Control, она же инверсия управления. Инверсия, как мы знаем из школьного курса русской литературы, это противопоставление, в нашем случае потока управления. И логично, что если есть инвертированное управление, то должно быть и прямое. Direct Control, или прямой поток управления – это самый обычный классический процедурный подход к программированию. Берем объект, вызываем его метод, используем результат. Команды выполняются просто и по порядку, как видно в этом примере:

```
List<Item> items = mDatabase.getSomeItems();
mAdapter.addItems(items);
```

Теперь рассмотрим другой вариант обращения к базе данных – с помощью callback'а, который мы передаем в метод getSomeItems. А что такое callback? Callback – это обратный вызов метода. Чувствуете разницу? База данных вызовет метод onItemsReceived, когда данные будут готовы. Держите в голове этот простой пример, сейчас мы перейдем к более сложному понятию.

```
class Database {

    List<Item> getSomeItems() {
        //get items
        return items;
    }

    void getSomeItems(Callback callback) {
        //get items
        callback.onItemsReceived(items);
    }

    interface Callback {
        void onItemsReceived(List<Item> items);
    }
```

```
16  }
```

Android, как вы уже знаете, – это фреймворк. Picasso, который мы используем для загрузки изображений, – это библиотека. В чем же разница между фреймворком и библиотекой, не конкретно Android и Picasso, а вообще? Библиотеки в большинстве своем дают нам функциональность, которую мы спокойно просто используем. Фреймворки дают нам контрольные точки, методы и интерфейсы, реализуя которые, мы добавляем свою функциональность. Проще будет в этом разобраться, если добавить еще одну сущность – приложения. Перефразирую: в чем разница между приложением, построенным на библиотеках и приложением, построенным на фреймворке?

Допустим, мы пишем приложение на чистой Java без какого-либо фреймворка. У нас есть метод main, точка старта, и мы сами решаем, как будет проходить процесс работы с приложением, порядок действий и обработка пользовательского ввода. Конечно, мы вправе не изобретать велосипеды и подключить какие-нибудь библиотеки для различных задач – для сетевых запросов, отображения графики и т. п. Но основной костяк работы в приложении мы пишем сами, ручками. С другой стороны, когда мы пишем приложение под Android, мы переопределяем методы onCreate, onResume, и этого достаточно, чтобы создать уникальное приложение. Мы не создаем активити через конструктор, мы используем интенты, которые скидываем в систему. Android берет на себя всю низкоуровневую работу, связанную с созданием и обработкой активити, то есть фреймворк не спрашивает нас, что нужно сделать, он это знает сам. Его интересуют только конкретные уникальные детали. Основной костяк работы приложения находится внутри фреймворка, вне нашего контроля.

Теперь давайте познакомимся с определением инверсии управления. Инверсия управления – это принцип проектирования, в котором части программы, написанные вручную разработчиком, получают поток управления из какого-либо общего фреймворка.

Какие задачи решает инверсия управления?

- Отделение вызова от реализации задачи;
- Концентрация на самой задаче, а не на обработке вызова;
- Отсутствие допущений о том, как система работает внутри, использование вместо этого общепринятых контрактов;
- Предотвращение сайд-эффектов от замены модуля.

В объектно-ориентированном проектировании различают несколько способов реализации инверсии управления. Первое – это DI – Dependency Injection, внедрение зависимостей. Второе – Service Locator, третье – контекстуальный поиск, четвертое и пятое – это паттерны «Абстрактная фабрика» и «Стратегия».

Мы только что выяснили, что внедрение зависимостей – это реализация инверсии управления. В следующем видео мы рассмотрим Dependency Injection подробнее.

1.1.2. Dependency Injection

Привет. В этом видео поговорим про Dependency Injection – внедрение зависимостей. Давайте расставим все по полочкам. Я буду повторяться, но, как говорится, повторение мать учения.

Допустим, у нас есть класс A, который внутри себя использует функциональность класса B. Класс A в этом случае называется клиентом, класс B – сервисом или зависимостью. Зависимость является частью состояния клиента – логично. В идеале зависимость должна быть абстракцией, интерфейсом, и клиент должен будет в какой-то момент получить реализацию этого интерфейса. Если зависимость создается внутри клиента, то говорят, что клиент сам разрешает свои зависимости. Если зависимость передается клиенту извне, то этот процесс называется внедрением.

Дальше, как мы помним из принципа единой ответственности, класс должен заниматься чем-то одним. DI-контейнер (инжектор) – класс или библиотека, которая занимается только тем, что готовит зависимости для других классов. Библиотеки и классы из других реализаций инверсии управления также называются контейнерами.

Внедрение зависимости – это техника программирования, в которой один объект (контейнер) снабжает зависимостями (сервисами) другие объекты (клиентов). Чтобы понять, зачем внедрение зависимостей вообще нужно, рассмотрим следующий пример: у нас есть класс Car, который имеет зависимости в виде объектов типа Engine и Tires. Класс Car сам разрешает свои зависимости, то есть сам выбирает и создает реализацию для двигателя машины и типа шин.

```
1  public class Car {
2
3      private Engine mEngine;
4      private Tires mTires;
5
```

```
6       public Car() {
7           mEngine = new AutoEngine("V2.0");
8           mTires = new Michelin();
9       }
10
11      public void start(Settings s) {
12          s.configure();
13          mEngine.run(s);
14      }
15  }
```

Какие могут возникнуть проблемы? Первая – мы не можем поменять реализацию AutoEngine и Michelin на другие, не изменив при этом код самого класса Car. Вторая – текущая реализация AutoEngine использует параметр V2.0, который сам является зависимостью и задан хардкодом. И третья – у нас не получится процитировать класс Car отдельно от классов AutoEngine и Michelin.

В качестве решения этих проблем передадим классы зависимостей напрямую в конструктор класса Car.

```
1   public Car(Engine engine, Tires tires) {
2       mEngine = engine;
3       mTires = tires;
4   }
```

Обратите внимание вот на что. Класс Car зависит от классов Engine и Tires. Engine, в свою очередь, зависит от строкового параметра. И тот класс, который будет использовать Car, должен знать и об Engine, и о Tires, и об engineCapacity. Задумайтесь: класс знает о зависимости, которую он сам использует не напрямую, а только для создания класса Car. Это, мягко говоря, неудобно. Что, если у нас будет 100 или 1000 зависимостей?

Единственный способ сейчас не захламлять класс внутренними зависимостями своих зависимостей – это получать эти самые зависимости из вызывающего кода, сверху, и так по цепочке вверх до самого начала инициализации программы. И находясь на таком высоком уровне, сложно вручную сориентироваться, какому классу какая зависимость нужна. Именно тут на сцене появляется DI-механизм.

Какие вопросы решает DI?

- Как сделать работу класса независимой от того, как были созданы его зависимости?
- Как перенести сборку зависимостей в отдельные конфигурационные файлы?
- Как сделать несколько различных конфигураций для приложения?
- Как сделать всю работу приложения независимой от того, как были созданы его компоненты?

Различают следующие способы внедрения зависимостей в объект:

- Constructor Injection – внедрение через конструктор обязательной зависимости;
- Property (Setter) Injection – переопределение через сеттер необязательной зависимости;
- Method Injection – передача в метод зависимости, которая используется только в этом методе.

У начинающих разработчиков может возникнуть вопрос: зачем мне все это? Как минимум, для упрощения тестирования. Данная схема наглядно демонстрирует возможности тестирования при подходе с DI. Если мы пишем тесты для какого-то одного модуля, то протестировать всю цепочку зависимостей, скорее всего, будет слишком трудозатратно. Мы можем предоставить тестовую зависимость с понятным и определенным поведением и спокойно протестировать модуль. Это широко распространенный прием в тестировании.

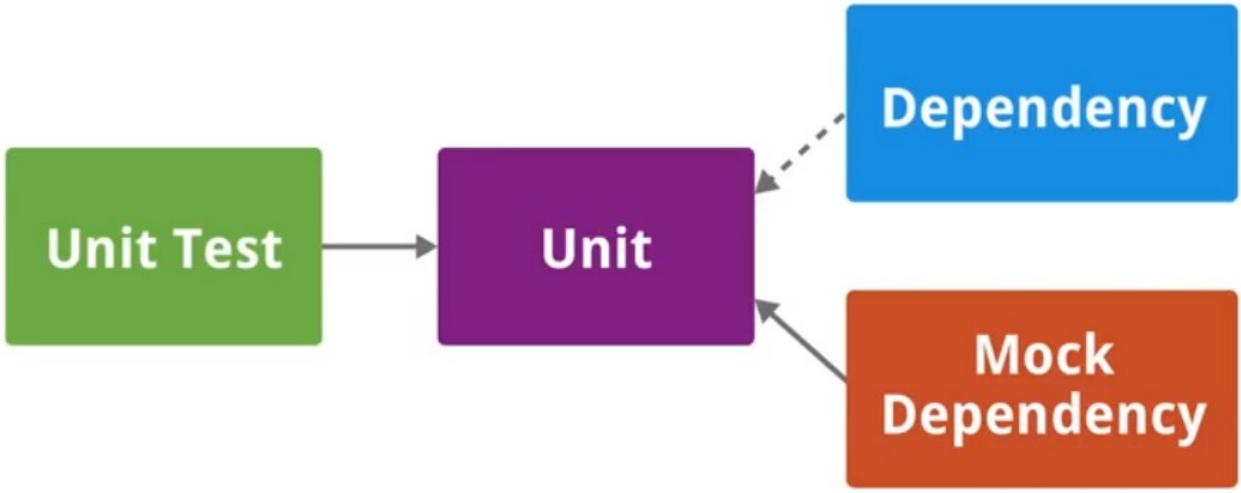


Рис. 1.1: Схема

Подведем итоги. Плюсы:

- Выбор реализации;
- Независимое тестирование;
- Слабая связность кода;
- Параллельная разработка функциональности.

Минусы:

- Сложность отладки из-за разделения мест создания и использования компонентов;
- Зависимость приложения от фреймворка для внедрения зависимости.

Это были основы DI. Я рекомендую вам также ознакомиться с тематическими сайтами, блогами, статьями, видео и проектами, так как тема сложная и распространенная. В ней нужно разбираться, если вы хотите преуспеть.

1.1.3. Service Locator vs DI

Dependency Injection, с которым вы познакомились, является одним вариантом реализации Инверсии контроля. Другой реализацией IoC является ServiceLocator.

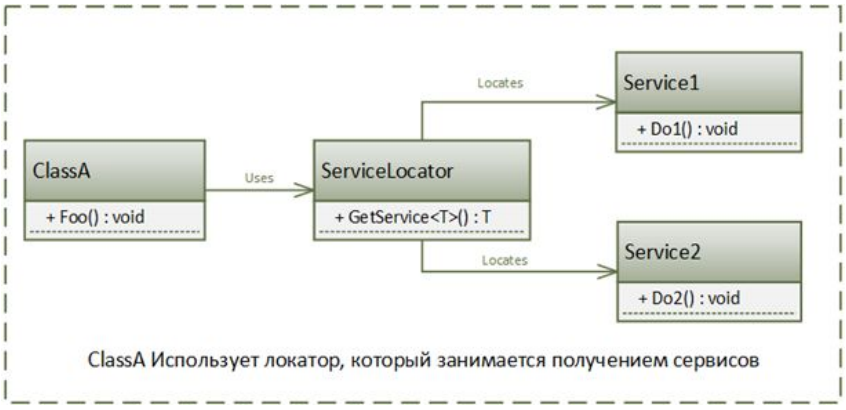


Рис. 1.2: Service Locator

Service Locator – это паттерн.

Суть его заключается в том, что вместо создания конкретных объектов – «сервисов» напрямую с помощью ключевого слова new, используется специальный «фабричный» объект сервислокейтор, который отвечает за создание, а точнее «нахождение» всех сервисов.

Реализовать данный паттерн можно 2 способами:

1. Локатор может быть синглтоном (в классическом виде или в виде класса с набором статических методов). Тогда доступ к нему может производиться из любой точки в коде;
2. Локатор может передаваться требуемым классам через конструктор или свойство в виде объекта класса или интерфейса.

Service Locator в статически типизированных, объектно-ориентированных языках считают антипаттерном. Причина – нарушение инкапсуляции: бизнес-логика не должна знать о нем и все зависимости должны пробрасываться явно.

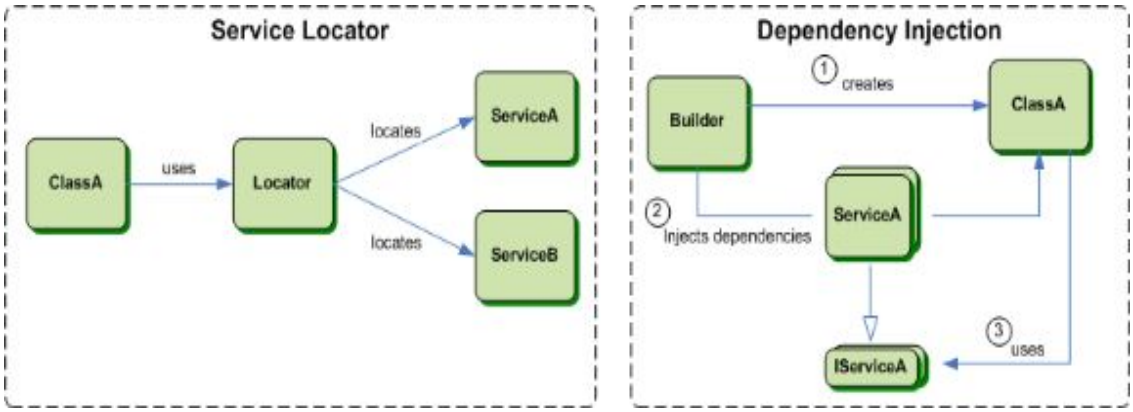


Рис. 1.3: Service Locator vs DI

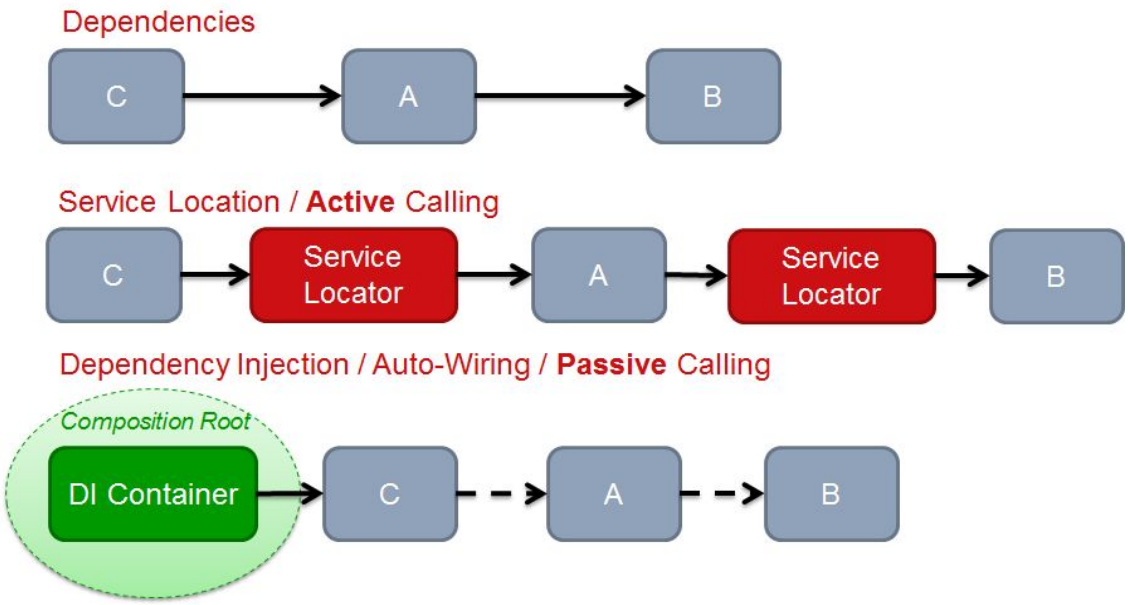
Давайте сравним Service Locator и Dependency Injection.

По всем правилам, использование контейнера должно быть ограничено минимальным количеством мест. В идеале, в приложении должна быть лишь одна точка, где производится вызов метода container.Resolve(); – определение, какому клиенту какие зависимости нужны. Этот код должен находиться либо в точке инициализации приложения, либо максимально близко к ней.

Различие в возможностях использования

Наличие в арсенале универсального объекта, способного получить любую зависимость, провоцирует к его использованию напрямую и в других частях приложения.

Мы можем протянуть нужную зависимость через конструктор, а можем просто передать этой вью модели сам контейнер, чтобы она получила требуемую зависимость самостоятельно. На рисунке приведены несколько случаев связи в зависимостях.



В 1 случае объекты последовательно зависят друг от друга. Во 2 случае продемонстрирована работа паттерна Service Locator, который на каждом этапе вычисляет и создаёт зависимости. В 3 случае демонстрируется работа внедрения зависимостей, в которой есть только одна-единственная точка входа.

Каждый раз, когда потребуется удовлетворить какую-то зависимость, ее можно просто получить из контейнера. При этом контейнер будет не только выдавать объекты, но и следить за их жизненным циклом: не плодить новые без необходимости и выгружать старые, когда они уже точно станут не нужны.

Удовлетворение зависимостей кода через умный контейнер – суть DI-паттерна. Одна из проблем – это долгий старт приложения. DI строит граф зависимостей в основном потоке.

Дополнительные материалы

В чем принципиальное отличие от Service Locator?

https://habrahabr.ru/post/166287/

http://blog.ploeh.dk/2010/02/03/ServiceLocatorisanAnti-Pattern/

1.1.4. О графе зависимостей

Граф зависимостей

Граф зависимостей – схема, которая отображает, каким образом элементы зависят друг от друга. Это позволяет нагляднее понимать, какой объект от какого зависит и кто кого использует.

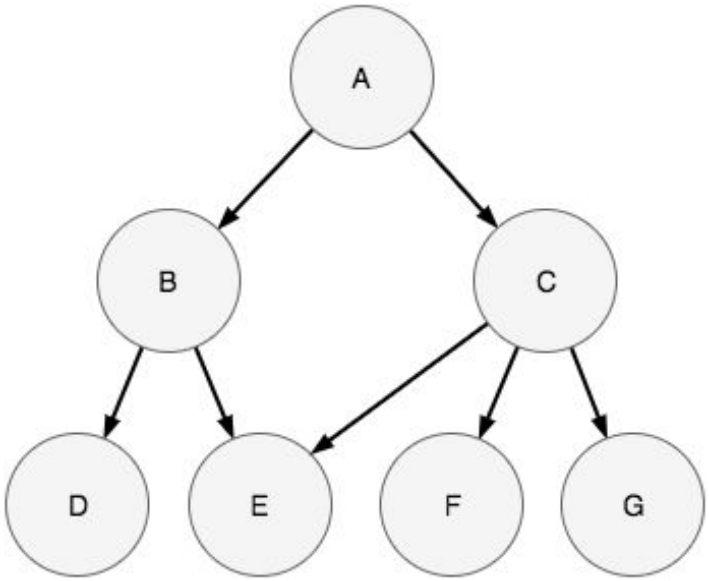


Рис. 1.4: Граф зависимостей

Мы уже знаем, что такое зависимость и насколько сложными они бывают. Для более простой и наглядной работы с зависимостями применяются графы зависимостей. Хотя в Android этот инструмент несколько ограничен по сравнению с .NET или же Enterprise Java, хотелось бы осветить инструменты и правила их создания.

Для чего?

Для чего же это нужно? Прежде всего для построения правильной (с точки зрения SOLID) и простой архитектуры.

Для визуализации можно использовать следующий инструмент: https://github.com/alexzaitsev/apk-dependency-graph/releases или https://github.com/dvdciri/daggraph – только для Dagger2. Данные инструменты отобразят вам структуру классов в созданном вами приложении.

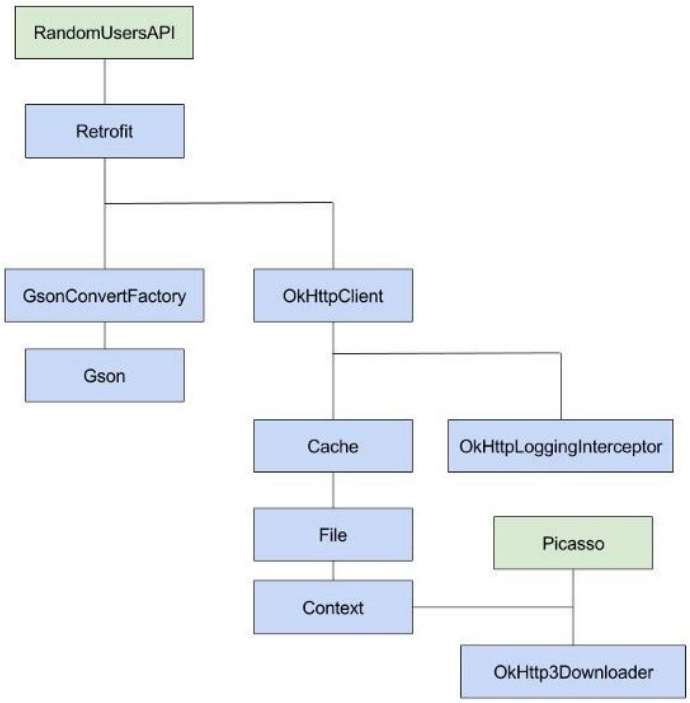


Рис. 1.5: Схема

А теперь давайте подумаем, какая структура у самого обычного Android-приложения? Допустим, это простое клиент-серверное приложение, которое получает и отдает какие-то данные. На верхнем уровне у нас расположены REST-методы, описанные через Retrofit. Retrofit, в свою очередь, получает транспортный уровень и какой-нибудь механизм маппинга своих классов. Чаще всего это OkHttp и Gson. OkHttp имеет зависимости от Cache и Picasso для отображения картинок и OkHttp3Downloader для их загрузки и логера.

Согласны, что схема гораздо нагляднее, чем словесное описание?

Теперь давайте рассмотрим некую идеальную структуру приложения с применением DI. Как видно из схемы, компоненты Android получают все зависимости из написанного нами AppComponent,

в котором мы внедряем всю сетевую работу, работу с БД и утилитарные классы.

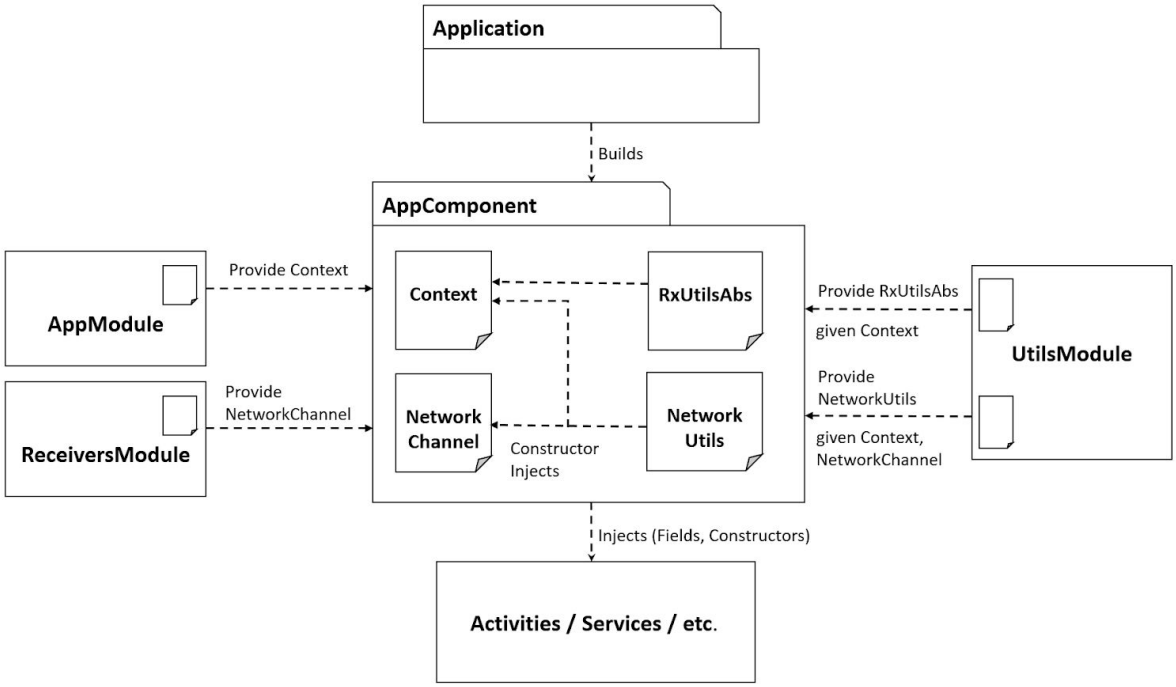


Рис. 1.6: Схема

У всех модулей линейная зависимость друг от друга, что дает нам возможность заменить на похожий по функциональности модуль, протестировать его отдельно от других и переиспользовать в других проектах.

1.1.5. Реализуем DI в приложении своими руками

Пример реализации

Наша задача – своими руками реализовать более-менее простейший DI контейнер и уйти от жестких зависимостей в приложении. Для этого нам нужно вынести интерфейс, который будет предоставлять базовый AbstractAccountAuthenticator при помощи метода getAuthenticator():

```
interface ISocialAuthManager {
    AbstractAccountAuthenticator getAuthenticator();
}
class AuthManager implements ISocialAuthManager {
    public AbstractAccountAuthenticator getAuthenticator() {
        //Реализация аутентификатора
    }
}
```

Реализация инкапсуляции

Инкапсуляция будет отвечать за предоставление нужного Authentificator для входа в мобильное приложение через определённую соц. сеть. Класс Auth будет иметь две зависимости (Vk и Instagram). Сами по себе классы не будут создаваться, а будут передаваться в конструкторе, тем самым мы всегда сможем их замокать и протестировать аутентификацию отдельно от остального кода. Также сам метод getAuthenticator() будет представлять из себя фабрику, которая возвращает соответствующий аутентификатор на основе класса, который мы ему передадим:

```
class Auth {
    private InstaAuth mInstaAuth;
    private VkAuth mVkAuth;
    // Внедрение зависимостей(DI) - получение зависимостей из другого
    места через конструктор
    public Auth(InstaAuth instaAuth, VkAuth vkAuth) {
        this.mInstaAuth = instaAuth;
        this.mVkAuth = vkAuth;
    }
    public void auth() {
        mInstaAuth.auth();
        mVkAuth.auth();
    }
}
```

Непосредственное внедрение зависимостей

Как может выглядеть место использования объекта Auth? Допустим, в активити у нас может быть такой код:

```
1  // AuthManager - это тоже зависимость!
2  AuthManager authManager = new AuthManager();
3  Auth auth = new Auth(authManager.getAuthenticator(InstaAuth.class),
4  authManager.getAuthenticator(VkAuth.class));
```

Но AuthManager – это тоже зависимость, а это означает, что по DI мы не можем создавать его внутри класса. Мы должны получать его извне.

Стартовая точка Android-приложения в подавляющем большинстве случаев - -это Application. И если масштабировать на все приложение, то в Application или недалеко от него будет код, который инициализирует все зависимости для всех классов и поставляет эти зависимости клиентам. Это рутинная работа. Для автоматизации этого процесса используются DI библиотеки.

Дополнительные материалы

Есть ли зависимость от контекста/жизненного цикла?

https://www.dataart.ru/news/dagger-2-lechim-zavisimosti-po-metodike-google/

В чем недостатки использования DI?

https://habrahabr.ru/post/321344/

1.1.6. Dagger 2. Часть 1

Привет. В этом видео мы познакомимся с библиотекой Dagger 2, реализующей внедрение зависимостей.

Как мы уже поняли, очень быстро точка входа приложения будет наполнена огромным количеством кода для инициализации всех зависимостей. Для создания одного класса, с которым

мы будем работать, нужно проинициализировать несколько других. Это рутинный процесс, и чтобы как-то его упростить, мы можем воспользоваться различными фреймворками, способными генерировать большое количество кода за нас, а нам остается только грамотно расставить аннотации. Производительность сгенерированного кода находится на том же уровне, что и написанного вручную, правда, о простоте и понимании такого не скажешь. Код, в свою очередь, генерируется при помощи annotation processor'ов – обработчиков аннотаций. Именно так работает Dagger.

Давайте разберем простой пример. Как обычно, первым делом добавляем зависимости в build.gradle уровня app. Не забываем, что нужно добавить еще и обработчик аннотаций. Хорошо. Допустим, мы хотим получить ссылку на репозиторий в активити, и для этого мы хотим воспользоваться Dagger. Предположим, что в этом репозитории нет каких-либо своих зависимостей. Первым делом нужно указать аннотацию Inject над конструктором репозитория. В активити над полем репозитория также добавляем аннотацию Inject.

```
1  @Inject
2  public SimpleRepository() { ... }
```

Для того, чтобы Dagger знал, куда инжектить зависимости, нужно создать интерфейс, помеченный аннотацией Component. В этом интерфейсе надо объявить метод, который принимает на вход объект клиента и ничего не возвращает. Также мы обязательно должны указать продолжительность жизни нашего компонента и зависимости, которую он предоставляет. Аннотация Singleton указывает на то, что зависимости и компонент будут жить, пока живо наше приложение.

```
1  @Singleton
2  @Component
3  public interface SimpleComponent {
4      void inject(SampleActivity activity);
5  }
```

Теперь нам нужно собрать проект – щелкаем на Build. После этого Dagger сгенерирует класс, который позволит с помощью билдера проинициализировать этот компонент. После создания нашего компонента нам нужно вызвать метод inject, чтобы Dagger внедрил все наши зависимости в SampleActivity. И уже после этого мы сможем работать с нашим репозиторием.

Если мы не можем получить доступ к конструктору и добавить над ним аннотацию Inject, то что делать? Тут в бой вступают модули. Модуль – это класс, который позволяет Dagger создать instance какой-нибудь зависимости с указанием ее реализации. Сначала мы создаем класс и помечаем его аннотацией Module. Внутри модуля определяем методы, возвращающие необходимые зависимости и помечаем эти методы аннотацией Provide. Чтобы компонент использовал модуль, его добавляют в качестве аргумента в аннотацию Component. Компонент может использовать несколько модулей. Продолжительность жизней зависимостей, которые предоставляет модуль, обязательно должна совпадать с таковыми у компонента, который этот модуль использует.

Вот мы и познакомились с Dagger на самом простом примере. Далее давайте рассмотрим другие его возможности.

1.1.7. Dagger 2. Часть 2

Продолжаем изучать Dagger 2. Довольно часто приходится использовать две или более различных реализации одного и того же интерфейса. Для того, чтобы Dagger смог отличить реализации друг от друга, используется аннотация Named. Например, есть различные источники данных и мы хотим обозначить, что у нас – сервер, а что – локальная база данных. Для обеих реализаций мы создаем Provide-методы и помечаем их с помощью аннотации Named, в качестве аргумента в которую передаем их строковые названия, по которым их можно будет отличить. Таким образом, в клиенте используется аннотация Inject вместе с аннотацией Named, в которую точно так же передаем нужную нам строку.

Аннотация Qualifier – это еще один способ определения уникальности связи. Для этого нам нужно создать уникальную аннотацию с помощью аннотации Interface, и уже далее использовать ее вместо аннотации Named. В примере мы создали аннотации ServerContext и LocalContext.

```
1  @Qualifier
2  public @interface ServerContext {}
3
4  @Qualifier
5  public @interface LocalContext {}
6
7  @Provide
8  @ServerContext
9  @Singleton
10 public DataSource provideRemoteDataSource() {
```

```
11  return new ApiImpl(); }
12
13  @Provide
14  @LocalContext
15  @Singleton
16  public DataSource provideLocalDataSource() {
17  return new DataBaseImpl(); }
18
19  //in client
20  @Inject @ServerContext
21  DataSource mDataSource;
22
23  @Inject @LocalContext
24  DataSource mDataSource;
```

Далее используем эти аннотации в модуле вместе с аннотацией Provide и в клиенте вместе с аннотацией Inject.

Теперь давайте поговорим про scope'ы. Scope – это время жизни и область видимости зависимостей. В прошлый раз в качестве scope мы использовали Singleton, который говорил о том, что наш компонент и зависимости должны жить на уровне приложения и существуют в одном экземпляре. А что если мы хотим сделать так, чтобы Dagger внедрял Presenter? В этом случае зависимость Presenter'а должна жить столько, сколько живет активити или фрагмент. Для Presenter'а Singleton использовать нерационально. Чтобы указать нужное нам время жизни зависимости, нам нужно создать свою Scope-аннотацию, как это показано в примере.

```
1   @Scope
2   @Retention(RetentionPolicy.RUNTIME)
3   public @interface FragmentScope {}
4
5   @FragmentScope
6   @Component(modules = {FragmentModule.class})
7   public interface FragmentComponent {
8       void inject(SampleFragment fragment);
9   }
10
11  @Module
12  public class FragmentModule {
13      @Provides
```

```
14      @FragmentScope
15      public Presenter provideSampleRpesenter() {
16          return new SamplePresenter();
17      }
18  }
```

Далее мы создаем компонент, и, если необходимо (ну, чаще всего) модуль и помечаем все зависимости внутри и сам компонент аннотацией Scope'а. В нашем случае это FragmentScope.

Стоит задуматься: а как Dagger понимает, когда зависимости, находящиеся в определенном Scope, должны умереть? Никак. Для этого нам самим нужно понять, когда мы точно перестанем использовать наш клиент – фрагмент или активити, в котором есть зависимости, и в этот момент уничтожить компонент, который эти зависимости предоставляет. И, если есть желание сохранить зависимости после переворота, то для этого нужно будет сохранить ссылку на компонент и использовать его после воссоздания активити или фрагмента.

Dagger 2 – сложная в освоении и понимании библиотека, что, однако, не мешает ей распространяться в Android-разработке.

1.1.8. Dagger 2 в MVP. Компоненты и модули

В данном занятии мы добавим библиотеку Dagger 2 для реализации паттерна Dependency Injection.

Для этого перейдем в проект. Gradle scripts, module app. Добавим три зависимости. Первая на Dagger, вторая на Dagger-Compiler, третья Dagger-Android-Support. Нажмем Sync now. Теперь нам нужно создать компонент и модуль для предоставления зависимостей. Жить они будут на уровне приложения, соответственно, они будут помечены аннотацией Singleton. Давайте перейдем к их реализации. Перейдем в проект, создадим package, назовем его di, создадим интерфейс AppComponent, пометим его аннотацией Singleton. Пометим его аннотацией Component, чтобы Dagger понимал, что это наш компонент. Далее добавим modules, поставим сразу фигурные скобки, эти модули мы реализуем чуть позже, а сейчас давайте добавим метод Inject для того, чтобы мы знали, куда будем инжектить наши зависимости. Инжектить мы будем в ProjectsFragment injector.

Теперь давайте перейдём к созданию модулей. Модулей у нас будет две штуки. Это app-модуль и network-модуль. App-модуль будет отвечать за общие зависимости, которые нам нужны в

приложении. Network-модуль, соответственно, будет отвечать за зависимости, которые работают с сетью.

Создадим app-модуль. Пометим его аннотацией Module. Создадим private final переменную AppDelegate mApp и создадим конструктор, в который будем передавать эту переменную. Далее добавим метод с аннотациями Provides и Singleton, который будет нам возвращать наш AppDelegate. Назовем его provideApp и будем возвращать наш mApp. Далее скопируем этот метод и перейдем в AppDelegate. Здесь скопируем всё, что связано с mStorage, больше у нас mStorage в AppDelegate не будет, можно удалить метод getStorage, у нас все зависимости будут предоставляться Dagger'у. Перейдём в AppModule, копируем этот код, вставим его в метод ProvideStorage. Возвращать он будет Storage соответственно. Здесь добавим return, return mApp сотрем. Вместо this у нас будет возвращаться mApp.

Теперь давайте добавим этот модуль в наш app-компонент. Перейдём в него, напишем AppModule.class, поставим запятую и создадим новый модуль. Скопируем AppModule и назовем его Network Module. Далее стираем конструктор и переменную AppDelegate, она нам больше не нужна. Перейдем в ApiUtils. ApiUtils теперь у нас практически не будет, в нём останется только NETWORK_EXCEPTIONS. И вставим весь этот код сюда. Скопируем аннотации Provides, Singleton. Вставим перед getClient. getClient переименуем на provideClient. Уберем private static переменные всех методов, эти модификаторы нам больше не нужны. Далее в provideClient стираем ненужный код, sClient у нас больше нет, теперь просто возвращаем return builder.build. Сотрём все переменные, продолжим делать то же самое с остальными зависимостями. Provides, Singleton. Get будем менять на provide. Provide, так же как и здесь. Теперь давайте добавим метод для gson'а. Gson provideGson, return new Gson. В provideRetrofit у нас будут передаваться два параметра. Это Gson и OkHttpClient. Далее стираем ненужный код. Здесь делаем return new Retrofit.Builder. Вместо getClient у нас будет client, вместо sGson у нас будет gson. Далее в BehanceApi нам нужна сущность Retrofit'а, передаём её сюда, Retrofit retrofit и возвращаем вот эту строку. Вместо getRetrofit у нас будет просто retrofit.

Добавим наш network-модуль в AppComponent. NetworkModule.class. Попробуем сбилдить наш проект. Разумеется, наш проект не сбилдится, потому что мы удалили всю логику работы с Storage и AppDelegate. Чтобы разрешить это, нам надо внедрять зависимости в те места, в которых он используется. Этим мы займемся в следующем занятии.

1.1.9. Dagger 2 в MVP. Используем зависимости

В данном занятии мы с вами будем использовать зависимости с помощью библиотеки Dagger 2 и с помощью модулей и компонентов, которые мы написали в предыдущем занятии. Для этого перейдем в проект. Откроем AppDelegate, сюда добавим private static AppComponent sAppComponent переменную, чтобы у нас постоянно хранилась ссылка на наш компонент, проинициализируем его в методе onCreate. sAppComponent = DaggerAppComponent. Если у вас нет DaggerAppComponent, просто попробуйте сбилдить ваш проект.

```
1  DaggerAppComponent.buider().appModule(new AppModule(mApp : this))networkModule(new
   ↪  NetworkModule()).build();
```

Если в вашем проекте app-модуль, network-модуль зачеркнуты – не пугайтесь, Dagger просто вам подсказывает, что методы, предоставляемые этими модулями, не используются. Далее создадим метод public static, который будет возвращать нам App Component, getAppComponent. Внутри него будем возвращать sAppComponent. Попробуем сбилдить проект и исправить ошибки, который допустили в предыдущем занятии. Начнем с getStorage. Весь интерфейс StorageOwner и все его реализации нам больше не нужны, соответственно, просто удаляем его. Далее ProjectsPresenter getApiService. Поскольку в api utils getApiService-метода больше нет, мы будем использовать mApi, который сейчас добавим. mStorage мы будем получать с помощью DI-паттерна. Inject BehanceApi mApi. В ProfileFragment метод getProfile, вернее, его реализацию я закомментирую, потому что вам это останется на домашнее задание, то есть api utils get api service вы должны будете получать с помощью паттерна DI. mStorage я также удаляю, он мне не нужен.

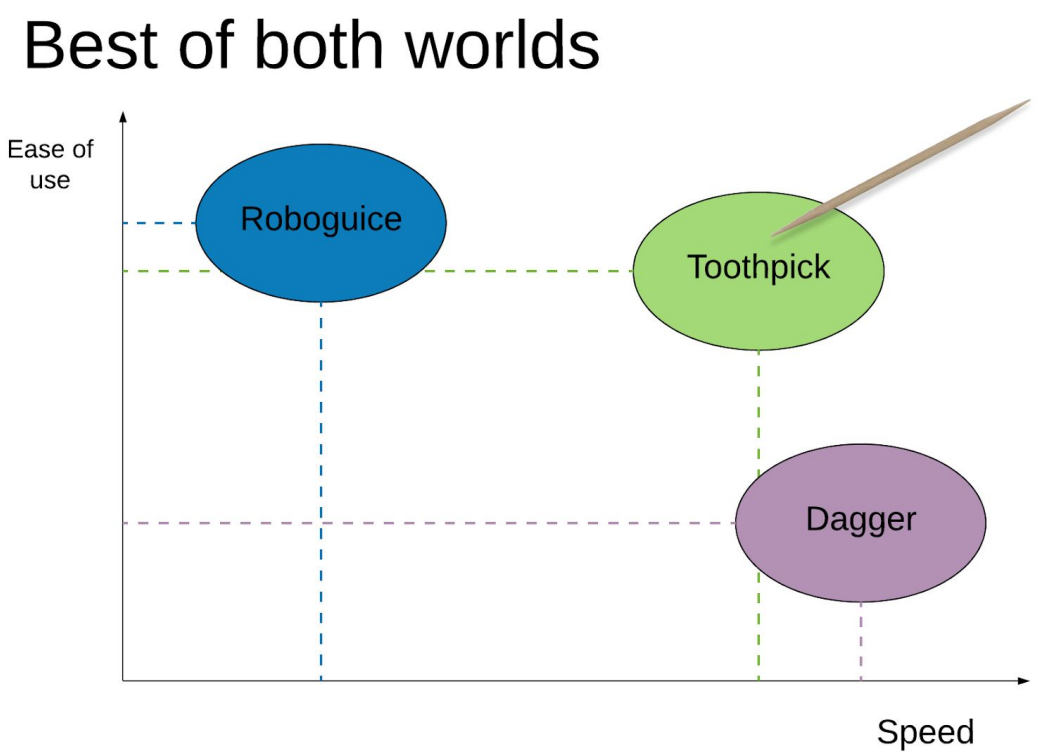
Далее перейдем в ProjectsFragment, удаляем все что связано с mStorage. Создаем метод onCreate, в нем AppDelegate.getAppComponent inject this. Это чтобы получить все наши зависимости, которые мы пометили аннотацией Inject. mStorage нам здесь не нужен, мы его будем получать с помощью DI внутри presenter'a. Соответственно, удаляем всё что с ним связано. Presenter мы будем получать с помощью DI, соответственно, добавим сюда аннотацию Inject. View теперь мы будем получать с помощью сеттера public void setView, в который мы будем передавать ProjectsView view. mView = view. Вообще это можно также реализовать с помощью паттерна DI и модулей. Это будет вам на домашнее задание. Теперь она у нас не final. И соответственно mView view отсюда уходит.

Далее идем в ProjectsFragment, убираем отсюда все параметры, которые передавали и здесь в mPresenter'e, также можно стереть создание presenter'a, потому что мы его будем получать с помощью аннотации Inject и здесь в mPresenter'e вызываем метод setView. Чтобы избежать вызова этого метода, вам нужно просто создать модуль, в который вы будете передавать вашу View

текущую, и Presenter поймет, какая View ему принадлежит. Попробуем сбилдить наш проект, посмотрим, нет ли ошибок, затем запустим эмулятор. Сбилдилось успешно, можно запускать. Как вы видите, ProjectsFragment успешно работает. В данном занятии мы успешно использовали зависимости с помощью библиотеки Dagger 2.

1.1.10. Обзор Toothpick

Давайте рассмотрим другую библиотеку, реализующую DI – Toothpick. Для начала, хотелось бы обратить ваше внимание на следующую диаграмму:



Мы видим, что Toothpick хорош с точки зрения простота/производительность.

С чего начать?

С зависимости в gradle.

```
implementation
'com.github.stephanenicolas.toothpick:toothpick-runtime:1.1.3'
implementation 'com.github.stephanenicolas.toothpick:smoothie:1.1.3'
annotationProcessor
'com.github.stephanenicolas.toothpick:toothpick-compiler:1.1.3'
```

Также добавим следующую директиву, она пригодится чуть позже:

```
annotationProcessorOptions {
    arguments = [ toothpick_registry_package_name :
    'com.elegion.test.behancer' ]
    }
}
```

Обратите внимание, что в качестве значения toothpick_registry_package_name нужно указать package вашего проекта. Далее в классе Application следует отключить рефлексию и установить MemberInjectorRegistry и FactoryRegistry.

```
Toothpick.setConfiguration(
Configuration.forProduction().disableReflection());

MemberInjectorRegistryLocator.setRootRegistry(
new com.elegion.test.behancer.MemberInjectorRegistry());

FactoryRegistryLocator.setRootRegistry(
new com.elegion.test.behancer.FactoryRegistry());
```

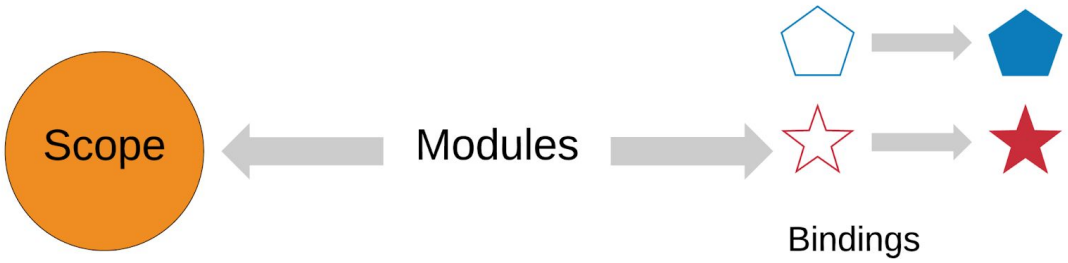
По сути стандартный код из документации, если копнуть чуть глубже, то в той же документации можно прочитать, что это нужно для поддержания полиморфизма при инъекции зависимостей. Если этого не сделать, то TP не сгенерирует классы наподобие этого, в котором зарегистрирует классы.

```java
public final class ToothPickActivity$$MemberInjector implements
MemberInjector<ToothPickActivity> {
    @Override
    public void inject(ToothPickActivity target, Scope scope) {
        target.title = scope.getInstance(String.class, "title");
    }
}
```

Пока у нас нет ни одной аннотации @Inject в коде, то FactoryRegistry() и MemberInjectorRegistry() не создаются, и проект не запустится. В этом одна из неточностей документации. Важный момент – если мы ранее не объявили javaCompileOptions, то будет ошибка, связанная с тем, что при сборке не будут найдены классы MemberInjectorRegistry и FactoryRegistry.

Схема работы

Давайте рассмотрим схему работы Toothpick.



Scope

Сначала разберемся с понятием «скоуп». В TP внедрение и создание компонентов всегда происходит в каком-либо скоупе. Скоуп – это некий интерфейс функциональности, чью реализацию

вы отдаёте модулям и далее у них запрашиваете на уровне инъекции готовый продукт.

Скоупы нужно открывать и закрывать. Единственным исключением является скоуп приложения – его закрывать не надо – он живет на протяжении жизни всего приложения.

```
Scope appScope = Toothpick.openScope(AppDelegate.class);
```

Также он определяет жизненный цикл этой функциональности.

Module

Модуль - класс TP, от которого мы наследуемся, чтобы реализовать в нём все наши зависимости. В документации по TP сказано, что модуль предоставляет небольшой DSL (Предметно-ориентированный язык) для того, чтобы сделать привязки наших зависимостей удобными.

Далее перейдем к модулям. Их надо создать и инициализировать в скоупе. Установление зависимостей в скоупе улучшено установкой модулей для скоупа:

```
appScope.installModules(
new SmoothieApplicationModule(this), new AppModule(this));
```

Рассмотрим создание самого простого модуля AppModule. Единственная функциональность, которую он нам несёт – это предоставление зависимости нашего Context нашего Application.

```
public class AppModule extends Module {
    public AppModule(Application application) {
        bind(Application.class).toInstance(application);
    }
}
```

Чтобы сообщить о том, что мы хотим добавить зависимость, надо использовать метод класса Module bind().

Давайте рассмотрим пример из документации:

```
class SimpleModule extends Module {
    SimpleModule() {
        bind(IFoo.class).to(Foo.class); // case 1
        bind(IFoo.class).toInstance(new Foo()); // case 2
        bind(IFoo.class).toProvider(FooProvider.class); // case 3
        bind(IFoo.class).toProviderInstance(new FooProvider()); // case 4
        bind(Foo.class); // case 5
    }
}
```

Есть несколько случаев использования bind:

1. Каждый @Inject IFoo будет ассоциирован с новой реализацией Foo.
2. Каждый @Inject IFoo будет ассоциирован с одной и той же реализацией Foo. Реализация определена в модуле.
3. Каждый @Inject IFoo будет ассоциирован с новой реализацией Foo, которую в свою очередь порождает создание нового объекта FooProvider.
4. Каждый @Inject IFoo будет ассоциирован с одной и той же реализацией Foo, её предоставляет один и тот же объект провайдера.
5. Каждый @Inject Foo создаст объект Foo, то же самое если мы напишем new.

Давайте предоставим простейшую зависимость – строку с названием нашего курса. Для этого создадим отдельный модуль TitleModule:

```
public class TitleModule extends Module {
    public TitleModule() {
        bind(String.class).withName("title").toInstance("Courcera");
    }
}
```

Так как у нас может быть много инстансов с классом String, мы должны её пометить с помощью метода withName() и при инжекте использовать @Named.

Далее, в Activity.

```
public class ToothPickActivity extends AppCompatActivity {
    private TextView mTextView;
    @Inject
    @Named("title")
    String title;
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        Scope appScope = Toothpick.openScope(App.class);
        appScope.installModules(new TitleModule());
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_tooth_pick);
        Toothpick.inject(this, appScope);
        mTextView = findViewById(R.id.sample_toothpick_dep);
        mTextView.setText(title);
    }
}
```

Мы должны открыть скоуп, предоставить ему модуль, в котором мы привязали эту зависимость и выполнить:

```
Toothpick.inject(this, appScope);
```

Если мы хотим, чтобы зависимость предоставлялась не в скоупе приложения, а, например, в скоупе, привязанном к Activity:

```
Scope scope = Toothpick.openScope(ToothPickActivity.class);
```

и не забываем закрыть этот скоуп:

```
@Override
protected void onDestroy() {
    Toothpick.closeScope(ToothPickActivity.class);
    super.onDestroy();
}
```

Если говорить о разработке в целом, по скоуп приложения служит прежде всего для предоставления зависимостей в виде синглтонов, то есть для работы с базой данных, сетевым слоем.

Скоупы, открытые в Activity и Fragment, предоставляют зависимости, завязанные на их жизненные циклы – это презентеры, адаптеры, листенеры.

Есть случаи, когда конструктор закрытый, плюс нам нужны и другие зависимости. Для создания таких зависимостей нужно использовать Provider. Provider<T> – это интерфейс, чей метод get() предоставляет зависимость. Рассмотрим ещё один пример:

```
public class RetrofitProvider implements Provider<Retrofit> {
    @Inject
    OkHttpClient mClient;

    @Inject
    Gson mGson;

    @Override
    public Retrofit get() {
        return new Retrofit.Builder()
        .baseUrl(BuildConfig.API_URL)
        .client(mClient)
        .addConverterFactory(GsonConverterFactory.create(mGson))
        .addCallAdapterFactory(RxJava2CallAdapterFactory.create())
        .build();
    }
}
```

Retrofit создается через билдер и требует 2 зависимости. Внутри провайдера тоже можно инжектить зависимости, например:

```
@Inject
OkHttpClient mClient;
```

при этом метод Toothpick.inject(this, scope); вызывать не надо.

Чтобы сделать эту зависимость синглтоном, надо вызвать метод providesSingletonInScope() в месте привязки.

```
bind(Retrofit.class).toProvider(RetrofitProvider.class)
.providesSingletonInScope();
```

Мы рассмотрели основы работы с Toothpick.

Ссылки на документацию:

https://github.com/stephanenicolas/toothpick/wiki

https://www.youtube.com/watch?v=EOFrA-MHbjUt=171s

1.1.11. Замена Dagger2 на Toothpick

В данном занятии мы с вами переделаем реализацию паттерна Dependency Injection с помощью библиотеки Dagger2 на библиотеку Toothpick. Поехали.

Откроем наш проект, перейдем в build.gradle (Module: app) и увидим, что я тут добавил несколько строчек в default config. Это javaCompileOptions, annotationProcessorOptions. Это нужно для того, чтобы наш код генерировался на основе аннотаций. Дальше. Я добавил три зависимости. toothpick-runtime, smoothie для андроида и toothpick-compiler. Посмотрим, что я переделал в AppDelegate. В AppDelegate я добавил несколько строчек кода. Они нужны для того, чтобы сконфигурировать наш Toothpick.

Далее можно переходить непосредственно к замене Dagger'а на Toothpick. Убираем sAppComponent, он нам больше не нужен, теперь у нас это будет sAppScope. Соответственно это будет Scope из toothpick'а. Далее возвращаться у нас здесь будет Scope и метод будет называться getAppScope. Метод installModules ругается на то, что мы ему передали неверные данные. Давайте посмотрим, что не так. Зайдем в NetworkModule, убираем аннотацию Module, она нам больше не нужна. Здесь добавляем строчку extends Module. Далее добавляем пустой конструктор и добавляем несколько private final переменных, сейчас поймете, зачем. mGson, OkHttpClient и Retrofit, соответственно mOkHttpClient, mRetroFit.

Далее в конструкторе NetworkModule делаем следующее. Метод bind, указываем ему class – в нашем случае это Gson, toInstance, и передаем ему mGson, но mGson должен быть чему-то равен. Он будет равен методу provideGson, соответственно, OkHttpClient будет равен provideClient, Retrofit будет равен provideRetrofit, который требует Gson и OkHttpClient пока что. Далее добавляем bind для OkHttpClient'а, для Retrofit'а, добавляем bind для BehanceApi. Здесь возвращаем OkHttpClient, здесь возвращаем Retrofit. BehancaApi возвращаем метод provideApiService. Метод аннотации Provide, Singleton просто стираем. Далее, он не принимает никаких параметров, а просто принимает instance Retrofit'а. Так же, как и provideClient, вернее, provideRetrofit. Client,

передаем ему mOkHttpClient, в gson передаем ему mGson. NetworkModule готов, едем дальше.

Переходим в AppDelegate, переходим в AppModule. Стираем аннотацию Module и добавляем ему extends Module. Здесь также добавляем методы bind, в нашем случае он должен предоставлять нам bind(AppDelegate.class).toInstance(mApp) и должен предоставлять Storage, который мы будем получать из метода provideStorage. Убираем аннотации Provides и Singleton. Работа над модулем AppModule завершена, едем дальше.

Переходим в ui, projects, ProjectsFragment и тут AppDelegate.getAppScope добавляем следующее. Toothpick.inject, this, и передаем ему наш Scope. Scope мы получаем из AppDelegate.getAppScope.

Далее просто попробуем сбилдить наш проект. Да, давайте удалим наш AppComponent, потому что он нам больше не нужен. Попробуем сбилдить еще раз. Проект успешно сбилдился, давайте теперь запустим его. Какая-то ошибка. Давайте посмотрим, какая. Откроем Logcat. NullPointerExeption, когда мы делали mPresenter, давайте зайдем в него. ProjectsPresenter. Inject mStorage. Inject mBehanceApi. С виду все выглядит правильно, что же не так? Разумеется, мы в AppDelegate не назначили AppScope, давайте исправим это. sAppScope, соответственно здесь также будет sAppScope. Запускаем, все должно работать. Как вы видите, приложение успешно запустилось. В данном занятии мы успешно совершили рефакторинг из библиотеки Dagger 2 на библиотеку Toothpick.

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