Rambler&Co 85,47
Компания

Подпи

28 января 2016 в 16:41

Построение Android приложений шаг за шагом, часть первая

Тестирование мобильных приложений*, Разработка под Android*, Разработка мобильных приложений*, Блог компании Rambler&Co

В этой статье мы поговорим о проектировании архитектуры и создании мобильного приложения на основе паттерна MVP с использованием R Retrofit. Тема получилась довольно большой, поэтому подаваться будет отдельными порциями: в первой мы проектируем и создаем приложе второй занимаемся DI с помощью Dagger 2 и пишем тесты unit тесты, в третьей дописываем интеграционные и функциональные тесты, а так размышляем о TDD в реалиях Android разработки.

Содержание:

Полное содержание

- Введение
- Шаг 1. Простая архитектура
- Шаг 2. Усложненная архитектура

Введение

Для лучшего понимания и последовательного усложнения кода, разделим проектирование на два этапа: примитивная (минимально жизнеспособная) и обычная архитектура. В примитивной обойдемся минимальным количество кода и файлов, потом улучшим этот код.
Все исходники вы можете найти на github. Бранчи в репозитории соответствуют шагам в статье: Step 1 Simple architecture — первый шаг, St
Complex architecture — второй шаг.
Для примера попробуем получить список репозиториев для конкретного пользователя с помощью Github API.

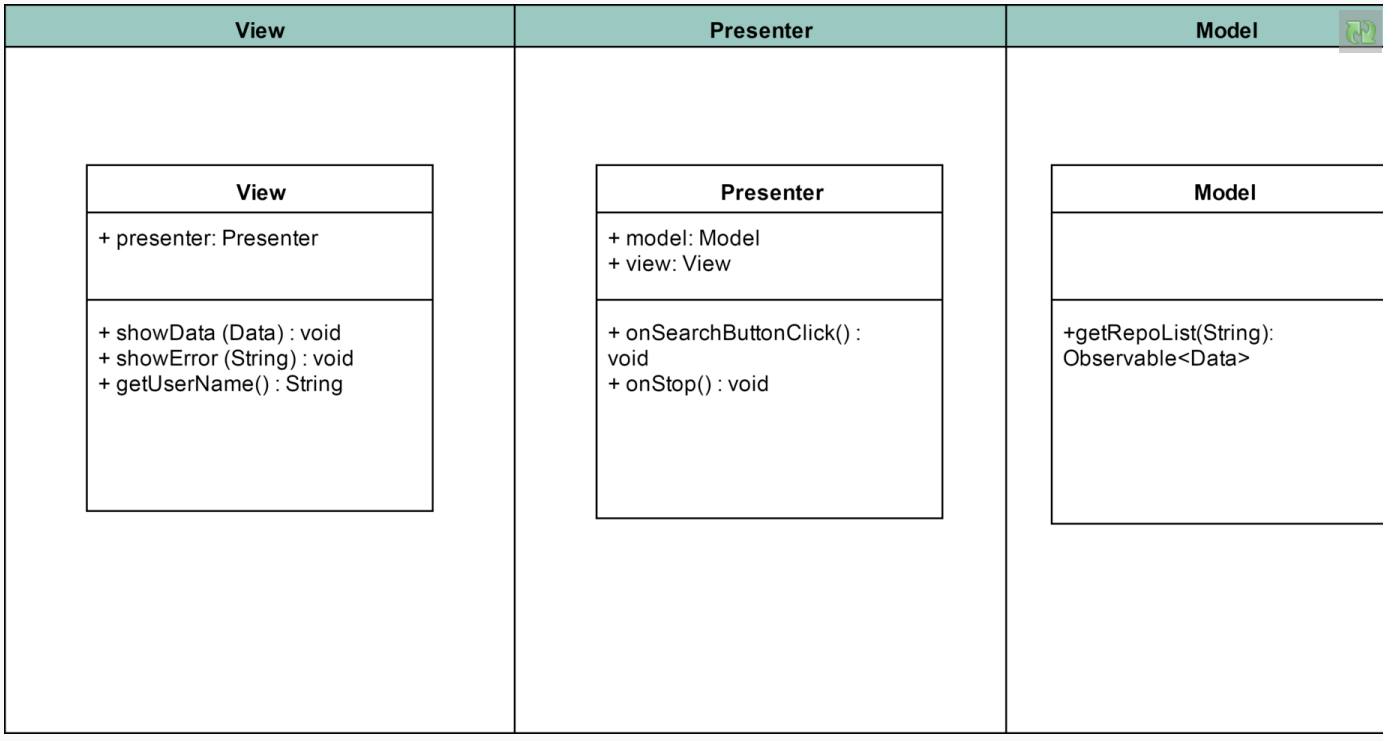
В нашем приложении мы будем использовать Rx, поэтому для понимания статьи необходимо иметь общее представление об этой технологии.Рекомендуем почитать серию публикаций Грокаем RxJava, эти материалы дадут хорошее представление о реактивном программировании.

Шаг 1. Простая архитектура

Разделение по слоям, MVP
При проектировании архитектуры будем придерживаться паттерна MVP. Более подробно можно почитать тут:
https://ru.wikipedia.org/wiki/Model-View-Presenter
http://habrahabr.ru/post/131446/

Разделим всю нашу программу на 3 основных слоя:
Model — тут получаем и храним данные. На выходе получаем Observable.
Presenter — в данном слое хранится вся логика приложения. Получаем Observable, подписываемся на него и передаем результат во view.
View — слой отображения, содержит все view элементы, активити, фрагменты и прочее.

Условная схема:



Model

Слой данных должен отдавать нам Observable<List<Repo>>, напишем интерфейс:

```
public interface Model {
    Observable<List<Repo>> getRepoList(String name);
}
```

Retrofit

Для упрощения работы с сетью используем Retrofit. Retrofit – библиотека для работы с REST API, она возьмет на себя всю работу с сетью, на остается только описать запросы с помощью интерфейса и аннотаций.

▾ Retrofit 2

Про Retrofit в рунете достаточно много материалов (http://www.pvsm.ru/android/58484, http://tttzof351.blogspot.ru/2014/01/java-retrofit.htm Основное отличие второй версии от первой в том, что у нас пропала разница между синхронными и асинхронными методами. Теперь мы получаем Call<Data> у которого можем вызвать execute() для синхронного или execute(callback) для асинхронного запроса. Также появила долгожданная возможность отменять запросы: call.cancel(). Как и раньше, можно получать Observable<Data>, правда теперь с помощью специального плагина

Интерфейс для получения данных о репозиториях:

```
public interface ApiInterface {
   @GET("users/{user}/repos")
   Observable<List<Repo>> getRepositories(@Path("user") String user);
}
```

▾ Реализация Model

```
public class ModelImpl implements Model {

    ApiInterface apiInterface = ApiModule.getApiInterface();

    @Override
    public Observable<List<Repo>> getRepoList(String name) {
        return apiInterface.getRepositories(name)
                .subscribeOn(Schedulers.io())
                .observeOn(AndroidSchedulers.mainThread());
    }
}
```

Работа с данными, POJO

Retrofit (и GSON внутри него) работают с POJO (Plain Old Java Object). Это значит, что для получения обьекта из JSON вида:

```
{
  "id":3,
  "name":"Andrey",
  "phone":"511 55 55"
}
```

Нам понадобится класс User, в который GSON запишет значения:

```
public class User {

    private int id;
    private String name;
    private String phone;

    public int getId() {
        return id;
    }

    public void setId(int id) {
        this.id = id;
    }

// etc

}
```

Руками генерировать такие классы естественно не нужно, для этого есть специальные генераторы, например: www.jsonschema2pojo.org.

Скармливаем ему наш JSON, выбираем:

Source type: JSON
Annotation style: Gson
Include getters and setters

и получаем код наших файлов. Можно скачать как zip или jar и положить в наш проект. Для репозитория получилось 3 обьекта: Owner, Perm
Repo.

▾ Пример сгенерированного кода

```
public class Permissions {

    @SerializedName("admin")
    @Expose
    private boolean admin;
    @SerializedName("push")
    @Expose
    private boolean push;
    @SerializedName("pull")
    @Expose
    private boolean pull;

    /**
     * @return The admin
     */
    public boolean isAdmin() {
        return admin;
    }

    /**
     * @param admin The admin
     */
    public void setAdmin(boolean admin) {
        this.admin = admin;
    }

    /**
     * @return The push
     */
    public boolean isPush() {
        return push;
    }

    /**
     * @param push The push
     */
    public void setPush(boolean push) {
        this.push = push;
    }

    /**
     * @return The pull
     */
    public boolean isPull() {
        return pull;
    }

    /**
     * @param pull The pull
     */
    public void setPull(boolean pull) {
        this.pull = pull;
    }
}
```

Presenter

Презентер знает что загрузить, как показать, что делать в случае ошибки и прочее. Т.е отделяет логику от представления. View в таком слу получается максимально «легкой». Наш презентер должен уметь обрабатывать нажатие кнопки поиска, инициализировать загрузку, отдават данные и отписываться в случае остановки Activity.

Интерфейс презентера:

```
public interface Presenter {
    void onSearchClick();
    void onStop();
}
```

Реализация презентера

```
public class RepoListPresenter implements Presenter {

    private Model model = new ModelImpl();

    private View view;
    private Subscription subscription = Subscriptions.empty();

    public RepoListPresenter(View view) {
        this.view = view;
    }

    @Override
    public void onSearchButtonClick() {

        if (!subscription.isUnsubscribed()) {
            subscription.unsubscribe();
        }

        subscription = model.getRepoList(view.getUserName())
                .subscribe(new Observer<List<Repo>>() {
                    @Override
                    public void onCompleted() {

                    }

                    @Override
                    public void onError(Throwable e) {
                        view.showError(e.getMessage());
                    }

                    @Override
                    public void onNext(List<Repo> data) {
                        if (data != null && !data.isEmpty()) {
                            view.showData(data);
                        } else {
                            view.showEmptyList();
                        }
                    }
                });
    }

    @Override
    public void onStop() {
        if (!subscription.isUnsubscribed()) {
            subscription.unsubscribe();
        }
    }
}
```

View

View реализуем как Activity, которое умеет отображать полученные данные, показывать ошибку, уведомлять о пустом списке и выдавать им пользователя по запросу от презентера. Интерфейс:

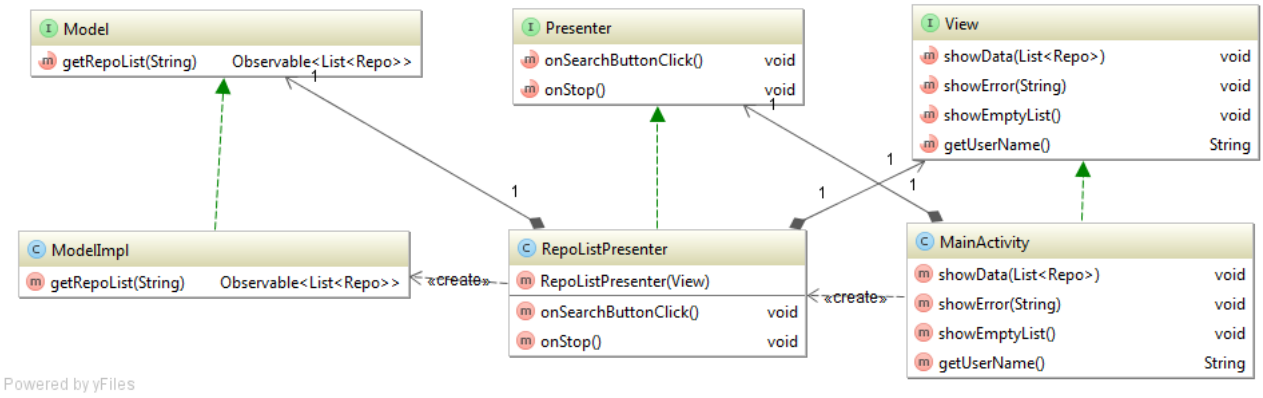
```
public interface IView {
    void showList(List<Repo> RepoList);
    void showError(String error);
    void showEmptyList();
    String getUserName();
}
```

Реализация методов View

```
@Override
public void showData(List<Repo> list) {
    adapter.setRepoList(list);
}

@Override
protected void onStop() {
    super.onStop();
    if (presenter != null) {
        presenter.onStop();
    }
}

@Override
public void showError(String error) {
    makeToast(error);
}

@Override
public void showEmptyList() {
    makeToast(getString(R.string.empty_repo_list));
}


@Override
public String getUserName() {
    return editText.getText().toString();
}
```

В результате у нас получилось простое приложение, которое разделено по слоям.

Схема:



Некоторые вещи требуют улучшения, однако, общая идея ясна. Теперь усложним нашу задачу, добавив новую функциональность.

Часть 2. Усложненная архитектура

Добавим новую функциональность в наше приложение, отображение информации о репозитории. Будем показывать списки branches и contr
они получаются разными запросами у API.

Retrolambda

Работа с Rx без лямбд — это боль, необходимость каждый раз писать анонимные классы быстро утомляет. Android не поддерживает Java 8 и

лямбды, но на помощь нам приходит Retrolambda (https://github.com/evant/gradle-retrolambda). Подробнее о лямбда-выражениях: http://habrahabr.ru/post/224593/

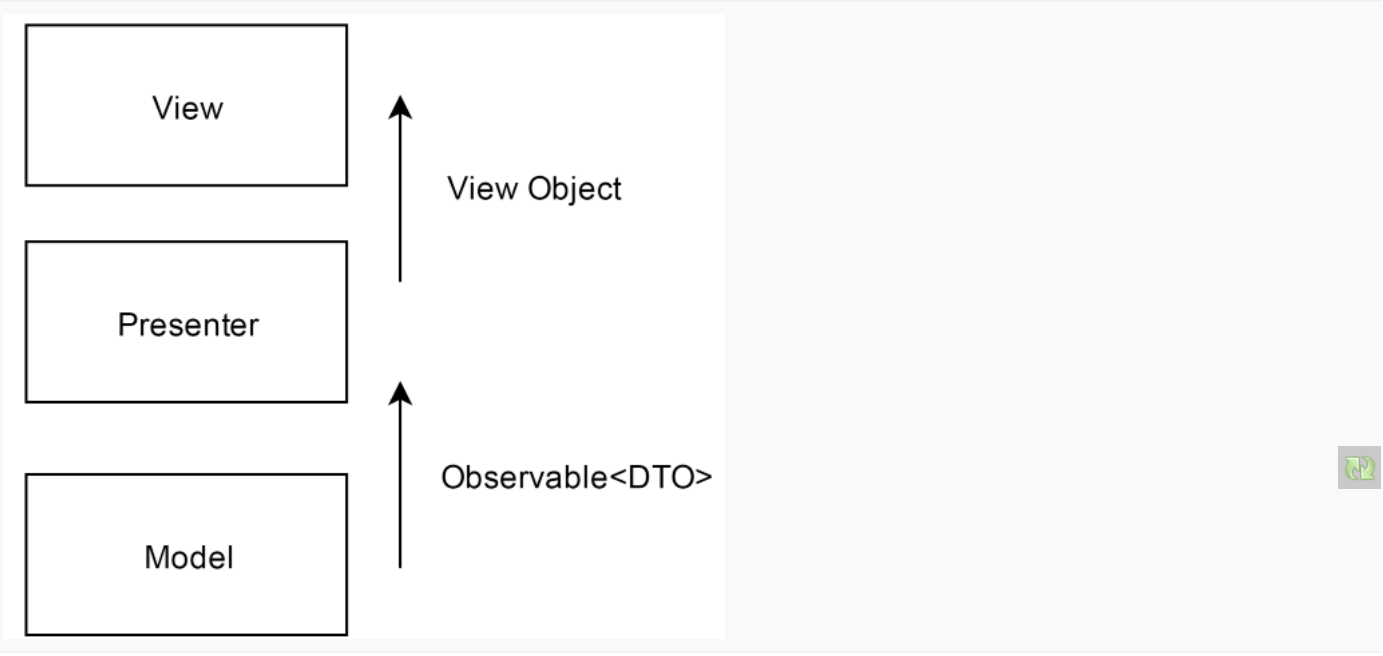
Разные модели данных для разных слоев.

Как видно, мы на всех трех слоях работаем с одним и тем же объектом данных Repo. Такой подход хорош для простых приложений, однако в реальной жизни мы всегда можем столкнутся со сменой API, необходимостью изменять объект или чем-то другим. Если над проектом работает несколько человек, то существует риск изменения класса в интересах другого слоя.

Поэтому зачастую применяется подход: один слой = один формат данных. И если изменятся какие-то поля в модели, это никак не повлияет на слой. Мы можем производить любые изменения в Presenter слое, но во View мы отдаем строго определенный объект (класс). Благодаря этом достигается независимость слоев от моделей данных, у каждого слоя своя модель. При изменении какой либо модели, нам нужно будет пере маппер и не трогать сам слой. Это похоже на контрактное программирование, когда мы точно знаем какой объект придет в наш слой и какой должны отдать дальше, тем самым защищая себя и коллег от непредсказуемых последствий.

В нашем примере нам вполне хватит двух типов данных, DTO — Data Transfer Object (полностью копирует JSON объект) и View Object (адаптированный объект для отображения). Если будет более сложное приложение, возможно понадобятся Business Object (для бизнес проц или например Data Base Object (для хранения сложных объектов в базе данных)

▾ Схематичное изображение передаваемых данных



Переименуем Repo в RepositoryDTO, создадим новый класс Repository и напишем маппер, реализующий интерфейс Func1<List<RepositoryDTO List<Repository>>
(перевод из List<RepositoryDTO> в List<Repository>)

▾ Маппер для объектов

```
public class RepoBranchesMapper implements Func1<List<BranchDTO>, List<Branch>> {

    @Override
    public List<Branch> call(List<BranchDTO> branchDTOs) {
        List<Branch> branches = Observable.from(branchDTOs)
                .map(branchDTO -> new Branch(branchDTO.getName()))
                .toList()
                .toBlocking()
                .first();
        return branches;
    }
}
```

Model

Мы ввели разные модели данных для разных слоев, интерфейс Model теперь отдает DTO объекты, в остальном все также.

```
public interface Model {
   Observable<List<RepositoryDTO>> getRepoList(String name);
   Observable<List<BranchDTO>> getRepoBranches(String owner, String name);
   Observable<List<ContributorDTO>> getRepoContributors(String owner, String name);
}
```

Presenter

В Presenter-слое нам необходим общий класс. Презентер может выполнять самые разные функции, это может быть простой презентер «загру покажи», может быть список с необходимостью подгрузки элементов, может быть карта, где мы будем запрашивать объекты на участке, а та множество других сущностей. Но всех их объединяет необходимость отписываться от Observable во избежание утечек памяти. Остальное зав типа презентера.

Если мы используем несколько Observable, то нам необходимо отписываться от всех разом в onStop. Для этого возможно использование CompositeSubscription: добавляем туда все наши подписки и отписываемся по команде.

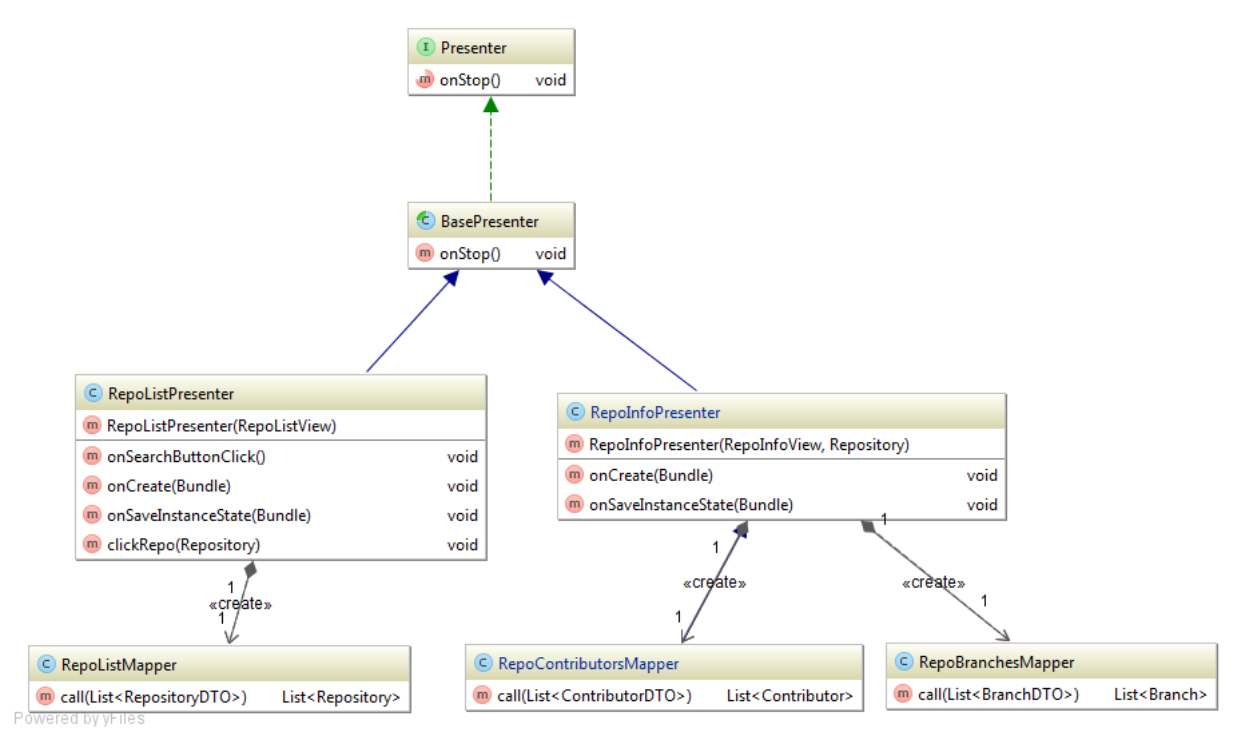
Также добавим в презентеры сохранение состояния. Для этого создаем и реализуем методы onCreate(Bundle savedInstanceState) и onSaveInstanceState(Bundle outState). Для перевода DTO в VO используем мапперы.

▼ Пример кода

```
public void onSearchButtonClick() {
        String name = view.getUserName();
        if (TextUtils.isEmpty(name)) return;

        Subscription subscription = dataRepository.getRepoList(name)
                        .map(repoListMapper)
                        .subscribe(new Observer<List<Repository>>() {
                                @Override
                                public void onCompleted() {
                                }

                                @Override
                                public void onError(Throwable e) {
                                        view.showError(e.getMessage());
                                }

                                @Override
                                public void onNext(List<Repository> list) {
                                        if (list != null && !list.isEmpty()) {
                                                repoList = list;
                                                view.showRepoList(list);
                                        } else {
                                                view.showEmptyList();
                                        }
                                }
                        });
        addSubscription(subscription);
}

public void onCreate(Bundle savedInstanceState) {
        if (savedInstanceState != null) {
                repoList = (List<Repository>) savedInstanceState.getSerializable(BUNDLE_REPO_LIST_KEY);

                if (!isRepoListEmpty()) {
                        view.showRepoList(repoList);
                }
        }

}

private boolean isRepoListEmpty() {
        return repoList == null || repoList.isEmpty();
}

public void onSaveInstanceState(Bundle outState) {
        if (!isRepoListEmpty()) {
                outState.putSerializable(BUNDLE_REPO_LIST_KEY, new ArrayList<>(repoList));
        }
```

```
}
```

Общая схем Presenter слоя:



View

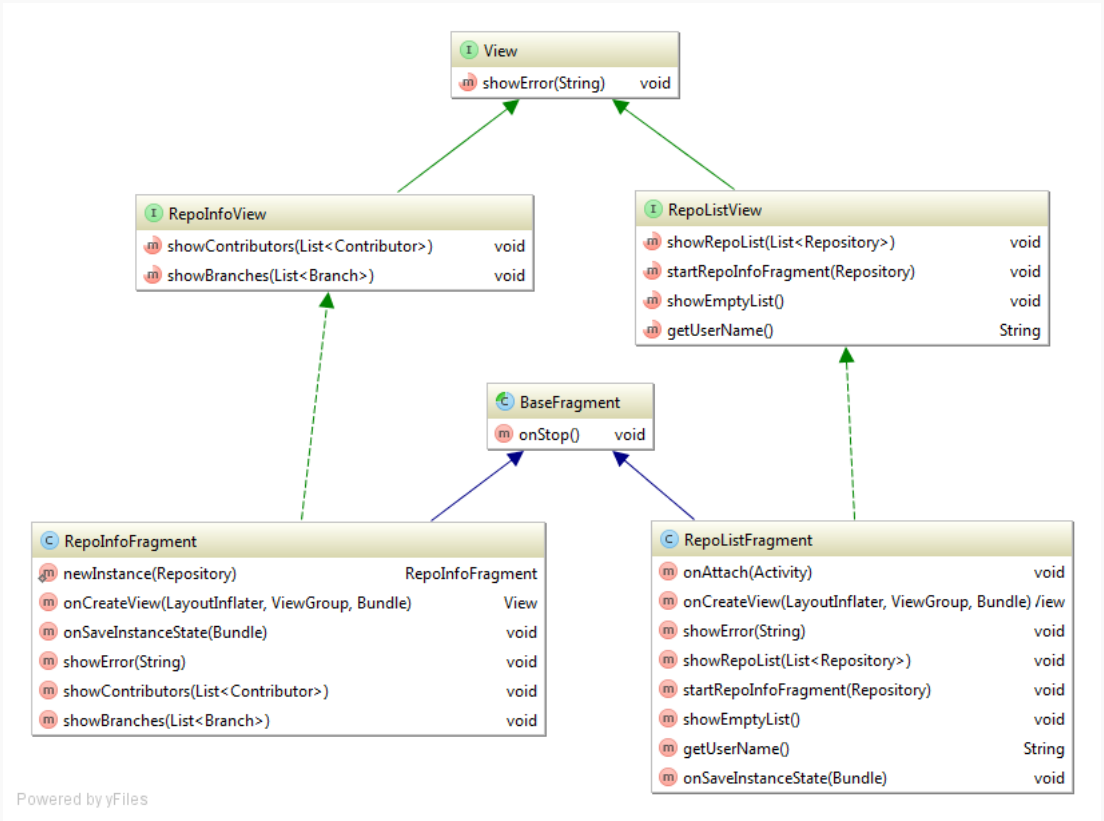
Будем использовать активити для управления фрагментами. Для каждой сущности свой фрагмент, который наследуется от базового фрагмен
Базовый фрагмент используя интерфейс базового презентера отписывается в onStop().

Также обратите внимание на восстановление состояния, вся логика переехала в презентер — View должно быть максимально простым.

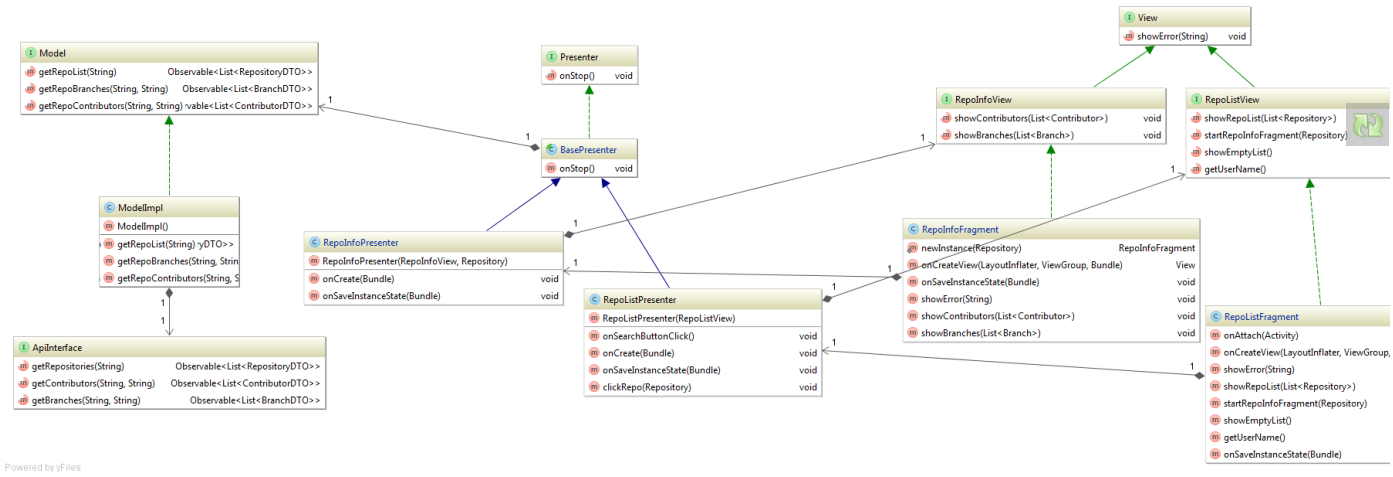
Код базового фрагмента

```
@Override
public void onStop() {
    super.onStop();
    if (getPresenter() != null) {
        getPresenter().onStop();
    }
}
```

Общая схема View слоя



Общая схема приложения на втором шаге (кликабельно):



Заключение или to be continued…

В результате у нас получилось работающее приложение с соблюдением всех необходимых уровней абстракции и четким разделением ответственности по компонентам (исходники). Такой код проще поддерживать и дополнять, над ним может работать команда разработчиков одним из главных преимуществ является достаточно легкое тестирование. В следующей статье рассмотрим внедрение Dagger 2, покроем тес существующий код и напишем новую функциональность, следуя принципам TDD.

UPDATE

Построение Android приложений шаг за шагом, часть вторая
Построение Android приложений шаг за шагом, часть третья

android, rxjava, архитектура приложений, mvp, архитектура android-приложений

ПОХОЖИЕ ПУБЛИКАЦИИ

12 февраля 2016 в 16:38
Android VIPER на реактивной тяге
+18 35,9k 202 35

26 октября 2015 в 19:21
Конвейерное производство Android приложений
+12 16,4k 137 18

16 сентября 2015 в 12:30
Тестирование на Android: Robolectric + Jenkins + JaCoCo
+20 20k 187 5

Комментарии (48)

отслеживать новые: в почте в трекере

Arturka 28 января 2016 в 21:17

Спасибо, очень хорошая статья, достаточно детально описано. Конечно, проблема MVP в том, что каждый понимает его по-своему (есть куча реализаций этого паттерна), предложенная схема в принципе одна из самых распространненых. Надеюсь, что вторая часть (особенно про тестирование будет также интересна)

Немного замечаний: не совсем понимаю выделение в интерфейс базового презентера — ведь как интерфейс он нигде не используется. Базовый класс презент это да, общая функциональность. А вот интерфейс — просто лишний.
И насчет извечной боли — как и утечек избежать, и запрос нормально сделать, тут есть спорные моменты. При вызове onStop у вас подписка отписывается, за отменяется. Отсюда следует, что при поворотах и прочем менеджить переподпиской придется самому. Альтернатива — использовать презентер для взаимодеі лоадерами или с сервисом, но тут свои минусы: в таком случае презентер уже будет знать о классах андроида. Впрочем, тестированию это особо не мешает.

P.S. Есть канал по паттернам, где 90% обсуждений посвящено MVP, можно присоединяться :)

ответить

andrey7mel 29 января 2016 в 11:41

Спасибо за комментарий и канал по паттернам! Менеджмент подписки и использование лоадеров тема для отдельной статьи, везде есть свои плюсы и мин

ответить

babylon 29 января 2016 в 07:46

Яркий пример как не надо программировать!

ответить

Newbilius 29 января 2016 в 08:25

А аргументы? :-)

ответить

Zeliret 29 января 2016 в 08:29

Поясните?

ответить

dev_troy 29 января 2016 в 09:00

Чего минусите то человека? Дело говорит. Посмотрите на количество файлов и связность.

ответить

Zeliret 29 января 2016 в 09:31

Ну, заявиться и сказать, что кг/ам (образно), — дело плевое.
А вот привести конкретную аргументацию на ресурсе, который посещают в том числе и новички, — силушки не хватило :)

ответить

babylon 29 января 2016 в 16:31 *(комментарий был изменён)*

Силушки, недостаточно, это правда. Я не отказался бы от поддержки, в том числе и от Rambler. Некоторые Mail.ru (например) двигаются по пути БЭМ им в руки.

ответить

agent10 29 января 2016 в 09:51

Нарисуете свою схему именно для этого примера?

ответить

dev_troy 29 января 2016 в 11:50

Я не умею рисовать схемы (шутка) =) Но на словах: зачем три файла/класса для презентера? Зачем интерфейс Presenter (если там lifecycle callback м то и назовите его LifecycleCallbacks по аналогии с ActivityLifecycleCallbacks)? А абстрактный BasePresenter, чтобы от CompositeSubscription отписывати Увеличение абстракции оправдано, если вы пишете какой-нибудь фреймворк или библиотеку. Или это делается потому что так написано в статьях по Задайте себе вопрос: как часто вам приходилось выкидывать один презентер и заменять его на другой? Мне, например, ни разу.

Что касается Model слоя: приложение вообще ничего не должно знать о сущностях DTO, DBO а так далее. Это все лучше вынести в, так называемый Repository слой. В котором выполняются мапинги данных, принятия решений о том, откуда эти данные брать (из кэша, БД или сети) и так далее. Нару торчит только Observable<List> getRepoList(String name);

При этом я ни в коем случае не говорю что статья плохая, статья отличная, особенно учитывая их скудное количество в русскоязычном сегменте.

ответить

agent10 29 января 2016 в 12:19

Вот это уже более аргументировано)

По поводу классов презентера соглашусь, но только для данного примера. В общем случае не известно как может пойти развитие.

А вот по сущностям в модели можно обсудить.
Вы предлагаете жёстко привязаться к модели данных.
Возьмём за пример модель RepositoryDTO — в реальности это большая модель с большим количеством полей.
В этом примере слою View надо всего лишь 4 поля. Для этого создаётся отдельный ViewObject — Repository(не путать с вашим паттерном) которы содержит только нужные поля, тем самым абстрагируясь от RepositoryDTO и от модели вообще.
Можно предположить, что в будущем измениться апи, скажем с GitHub на GitLab.
В текущем примере нужно поменять маппер Model <->ViewObject. В вашем менять View.

ответить

dev_troy 29 января 2016 в 13:16 *(комментарий был изменён)*

Вы меня неправильно поняли. Я как раз об этом и говорил. Есть некоторые сущности, которые являются ViewObject, при этом откуда она была получена, нам абсолютно неважно. Получение же этого ViewObject логично вынести в слой, который я назвал Repository, роль которого в контек статьи выполняет ApiInterface. Просто я предлагаю скрыть всю реализацию мапингов и прочих манипуляций внутри реализации ApiInterface и на выдавать уже готовые ViewObject модели. В статье же это делается на уровне презентера.

```
getRepoList() {
    return retrofit.getRepoList().flatMap({Model -> ViewObject});
}
```

ответить

agent10 29 января 2016 в 13:31 *(комментарий был изменён)*

Я понял, можно, но это если View чётко соответсвует одной модели, а если нет?
Например, есть 2 Активити, которым нужна одна модель, но разные поля из неё т.е. для разных View, ViewObject будут разные, хоть и будут абстрагировать одну модель.
У вас получится, что слой Repository будет иметь методы getViewObject1() и getViewObject2().
В примере автора, о конкретном ViewObject будет знать только соответствующая конкретная реализация Presenter, что на мой взгляд нескол правильнее.

ответить

andrey7mel 29 января 2016 в 13:31

Спасибо за замечания!
1) Интерфейс Presenter в данный момент получился лишним, с этим я полностью согласен.
2) Абстрактный BasePresenter содержит в себе функцию управления подпиской, реализовывать это в каждом презентере = дублировать код.
3) Model слой и DTO. Тут палка о двух концах, с одной стороны независимость приложения от DTO, DBO и прочих моделей (которая в данный мом достигается за счет мапперов), с другой стороны жесткая связка Model (DataRepository) и прикладного приложения.
В данный момент в слое Model нет ни одного импорта из Presenter или View, мы можем перенести весь слой в другое приложение, он полностью независим. Также, как было сказано выше, при изменении model нужно переписать мапперы, View Object (а вместе с ними и остальной код) при эт изменятся.

ответить

andrey7mel 29 января 2016 в 11:42

Расскажите как надо, с удовольствием посмотрим и обсудим!

ответить

babylon 29 января 2016 в 16:22

Надо всем внимательно посмотреть на такой проект Google как JSONNET. Использовать JSON в качестве шаблона для генерации кода классов это правильно Неправильно использовать классы и интерфейсы. Ещё более неправильно не программировать непосредственно в нотации JSON
Существующая JSON Schema как пример для подражания ужасна и непродуманна. Поэтому неудивительно, что её пытаются улучшать и расширять. Нотация позволяет проектировать системы снизу вверх, а использовать их сверху вниз. Можно собирать, разбирать и пересылать код «на лету». К сожалению Web пока заточен под JSON. Но думаю ситуация в ближайшие годы может резко измениться. И надо быть к этому готовыми. Можете еще отминусовать. Мне наплевать!

ответить

agent10 29 января 2016 в 16:42

Честно не понял, о чём вы?) А если поменять формат передачи данных в примере на XML, то к чему ваш комментарий будет относиться?

ответить

lair 9 февраля 2016 в 00:44

> Надо всем внимательно посмотреть на такой проект Google как JSONNET.

Маленький нюанс: Jsonnet — это **не** проект гугла. Вот вам цитата:

> Jsonnet is not an official Google product (experimental or otherwise), it is just code that happens to be owned by Google.

ответить

babylon 29 января 2016 в 16:56

XML отличается от JSON не в лучшую сторону. Я автор таких скриптов как ArrayToXML,ArrayToJSON,JsonToArray. Поэтому знаю между ними разницу:))). То что поняли нестрашно. Но лучше не менять. Точнее менять можно. Но автоматически. Вообще все должно еще и бинаризоваться :)

ответить

agent10 29 января 2016 в 17:00

Хорошо, я спрошу иначе, как формат передачи данных относится к архитектуре и взаимодействию между компонентами внутри приложения?

ответить

babylon 29 января 2016 в 17:07

Олично относиться. Архитектура строится на ссылках. В нодах JSON можно использовать поле id или использовать целочисленные имена нод как parent.

```
'$': {
// The pointer to the start of the scope node-context
// Указатель начала области действия ноды-контекста
'=>': 'content db',
'persons': {
'Dave': {
'#': 120,
'->': ['=>', '&', 20],
'20': [5, 1, 1, 1, 505, 3015, 5],
},
'Andy': {
'#': 130,
'21': [1, 2, 20, 2, 45, 400, 6],
'->': ['=>', '&', 21]
}
},
…
```

ответить

agent10 29 января 2016 в 17:16

Опишите идею в контексте, в данном случае, MVP и Android.

ответить

babylon 29 января 2016 в 17:31

Всё тоже. Архитектура приложения должна описываться в нотации JSON. В частности сервисы или вьюхи. Если мы используем парадигму событий, то должны эмититься события и к ним привязываться ссылки на ноды.События можно тоже выстраивать в иерархии. JSON напоминаю расшифровывается как JavaScript Notation. Слово object — ключевое. Когда программирование на классах назвали «ООП', то сильно погорячились.Пора возвращаться к корням. То есть к объект массивам. Хотя класс это тоже объект. И потуги в C# его объектизировать и „разобрать“ налицо. Есть разные подходы работы с событиями. С глобальным диспетчером событий или подход основанный на сигналах объекта или ноды.

ответить

agent10 29 января 2016 в 17:42

Для начала оффтоп — чтобы ответить на комментарий есть кнопка «Ответить», не нужно писать новый каждый раз.
Далее, вы куда-то уходите от идеи, C# тут появился уже:)… можете дать конкретный пример, ссылки?
Скажем есть база данных, простейшая. Есть экран со списком и с кнопкой, по нажатию идёт запрос в базу и заполняется список.
Какие ноды куда мне надо привязывать?

ответить

babylon 29 января 2016 в 18:35

Я не даю ссылки даже на свои примеры. Сами придумаете и опишите структуру в JSON формате. Создайте парсер и решите задачу на том языке на кот работаете.

ответить

Zeliret 29 января 2016 в 18:41

Шизофрения какая-то в комментах пошла…

ответить

agent10 29 января 2016 в 18:49

Ну я очень хотел понять, о чём говорил товарищ @babylon :) Но комментом выше он решил слиться.

ответить

babylon 29 января 2016 в 19:27

Я по прежнему здесь. Смотрю какой-то гад мне ещё наминусовал.

ответить

dev_troy 29 января 2016 в 23:46

Согласен, как ни старался, так и не понял причем тут JSON и архитектура.

ответить

babylon 30 января 2016 в 00:15

Думаю к концу года мы с Дейвом, что нибудь покажем, а пока отдыхайте и не забивайте этим мозги. Есть кто сразу втыкается, но это скорее исключение.

ответить

senneco 3 февраля 2016 в 09:15

@babylon похоже вы комментарии не в той статье пишите. Здесь разговор про Android+MVP, а вы говорите о JavaScript Object Notation, JSONNET и т. д.

ответить

babylon 3 февраля 2016 в 15:50

Я могу поговорить и об этом:) В топике рассматривается построение архитектуры. Причем для генерации классов используется JSON. Меня и удивляет, собственно генерить классы. Когда JSON как нельзя лучше подходит для построения архитектуры. Собственно мы чем мы оперируем. Есть нода-массив объект и нода-значение. Можно использовать целые числа в качестве адресации. То есть сделать нормальный биндинг. Ноды можно использовать как контейнеры для контента других нод. То есть имеются все составляющие для проектирования архитектуры. Я не говорил еще ничего о преимуществах JSONNEN. Топик звучит как «Построение Android приложений шаг за шагом». а не Android+MVP. Я хочу понять зачем почему всех склоняют к MVP. Прич очень не искусно даже на классах.

ответить

Bringoff 3 февраля 2016 в 18:00

Товарищ, вы пишете, совершенно не разбираясь в теме. Каким образом то, что возвращает rest api поможет построить архитектуру android-приложен предлагаете в java вместо классов для модели хранить все в json? Не говоря об отвратительной скорости работы с этим (по сравнению с POJO), как в json может заменить нормальные, человеческие java-классы?

ответить

agent10 3 февраля 2016 в 18:48

Дайте пример?) Или вы, сударь, трепло?)

ответить

babylon 3 февраля 2016 в 19:00

Да я аналитик или по вашему «трепло». Что в java-классах человеческого? И нормального??? Классы те же объекты, только избыточные. Плохо приспособленн разборки и сборки. Поэтому выдуваются всяческие костыли в виде DI. Я примеры не даю. Читайте официальную документацию. Причём тут REST API??? Скорость вполне сравнима. А для специально заточенных движков даже превос её. Вобщем оставайтесь на своей волне. Пока вас не смыли другой.

ответить

Bringoff 3 февраля 2016 в 20:12

> Классы те же объекты, только избыточные.

Wat??

> Плохо приспособленные для разборки и сборки. Поэтому выдуваются всяческие костыли в виде DI.

Как связаны dependency injection и сборка/расборка объектов? Что такое вообще сборка объектов?

> Причём тут REST API????

При том, что в статье идет речь о запросах к api, которые возвращают json.
Ощущение, что вы понятия не имеете о разработке на java и под android. Или просто троллите всех.
И да, используйте кнопку «Ответить».

ответить

babylon 3 февраля 2016 в 20:53

А у меня ощущение, что вы с утра не попрыгали. Что такое вообще архитектура, что такое XPath,JSONPath, дисплей листы? Хрень полная, да? Всё тако медленое. Фу-фу-фу.

ответить

agent10 3 февраля 2016 в 21:13

JSONPath этот ваще для школоты, а от XPath аж тошнит.

ответить

Bringoff 3 февраля 2016 в 21:25 *(комментарий был изменён)*

Стоит упомянуть, что пьяным заходить на хабр не стоит. Причем здесь дисплей-листы? Это вообще поприще компьютерной графики.

Скрытый текст

ответить

babylon 3 февраля 2016 в 21:39

А то что мы видим на экране Андроида это, что такое? Это не. Некомпьютерная графика.

ответить

Bringoff 3 февраля 2016 в 22:06

Экась вы ловко со слоя model на view перескочили. Точно тролль.
То есть, вы gui на чистом opengl реализовываете? Понятно, почему вы такой нарезанный. Вас остается только пожалеть.
Но все же вернитесь к нашим ~~баранам~~ json-ам, если вам есть, что сказать по существу. И загляните под спойлер предыдущего сообщения. Или так ловко обходите неприятные вам углы?

ответить

babylon 3 февраля 2016 в 22:42

На чистом канвасе. Но вы все равно не знаете, что это.

ответить

Bringoff 3 февраля 2016 в 22:59

Вот мне чисто для себя — вы разницу между Java и JavaScript улавливаете? Или canvas андроидовый? Ну, тогда вообще все очень плохо

ответить

babylon 3 февраля 2016 в 23:04

Что там не так, а да… Air в топку. Для меня самое сложное в Андроиде это кастомизация инпута. Остальное можно пережить. Кстати е
Java порт. Так что всё не плохо.

ответить

agent10 3 февраля 2016 в 19:35 *(комментарий был изменён)*

Вы не нервничайте — и не будете промахиваться по кнопке «Ответить» :) Это ведь не удобно, вот вам пример)
По делу, официальную документацию к чему надо читать? Открыл документацию по Андроиду — вашу тему не нашёл.

ответить

babylon 3 февраля 2016 в 19:44

Документация по Андроиду для птушников.

ответить

Error_403_Forbidden 10 февраля 2016 в 12:07

А ты, судя по всему, академик

ответить

kemsky 19 февраля 2016 в 01:21

Честно говоря, я давно так не смеялся читая комментарии :)

ответить

Вы не можете комментировать эту публикацию

Можно комментировать публикации, которые не старше 10 дней, а также те, которые вы уже комментировали ранее.

PhotoScan: как делать фотографии фотографий без бликов GT

1,6k 14 5

Электроника для ролевых игр в 2016 году GT

3k 6 10

Тысячная избранная статья. Как устроено рецензирование в Википедии GT

3,9k 17 44

SmartPoket. Часть 3, заключительная: мой личный органайзер GT

3,6k 7 12

Разработчик «умных» кредиток Plastc собрал предзаказов на $9 млн и объявил о банкротстве GT

9,4k 11 22