senneco 12 февраля 2016 в 15:56

# Moxy — реализация MVP под Android с щепоткой магии

Разработка под Android, Разработка мобильных приложений, Проектирование и рефакторинг

## Что такое *MVP*

*MVP* – это способ разделения ответственности в коде приложения. *Model* предоставляет данные для *Presenter*. *View* выполняет две функции: реагирует на команды от пользователя(или от элементов UI), передавая эти события в *Presenter* и изменяет gui по требо *Presenter*. *Presenter* выступает как связующее звено между *View* и *Model*. *Presenter* получает события из *View*, обрабатывает их(используя или не используя *Model*), и командует *View* о том, как она должна себя изменить.

У такого подхода к разделению ответственности есть ряд плюсов:

1. Сильно упрощается написание тестов к коду
2. Легко менять какую-то часть, не ломая при этом другую
3. Код разбивается на мелкие кусочки, за счёт чего он становится более понятным и читабельным

В то же время, конечно, есть и минусы:

1. Кода становится больше
2. К этому подходу нужно привыкать
3. На данный момент не сильно распространённый(*но известный*) подход, поэтому приходится всем рассказывать о нём

## *MVP* в Android

Activity в Android является God object. На ней обычно лежит следующая ответственность:

- Полное управление GUI
- Обработка взаимодействия с пользователем.
- Запуск асинхронных задач.
- Обработка результата асинхронной задачи.

Самое печальное, наш God Object не бессмертен – Activity ещё и умирает при смене конфигурации.

*MVP* снимает часть ответственности с Activity. Вся работа с асинхронными задачами уходит в *Presenter*. Вся бизнес-логика – в *Pre* и *Model*. Activity, в свою очередь, становится *View*. Она начинает просто отображать то, что скажет *Presenter* и передаёт события в *Presenter*, чтобы тот решал, как быть дальше.

Перед написанием своего решения мы изучили множество статей и реализаций концепции *MVP* в Android(см. ссылки в конце стат На основании анализа сложился список требований к решению:

1. *View* должна привязываться к уже имеющемуся *Presenter* при смене конфигурации
2. После привязывания *View* к уже имеющемуся *Presenter*, *View* должна отображать актуальное состояние *Presenter*
3. *Presenter* должен уметь(*при необходимости*) жить независимо от того, кто на него подписан или от него отписался

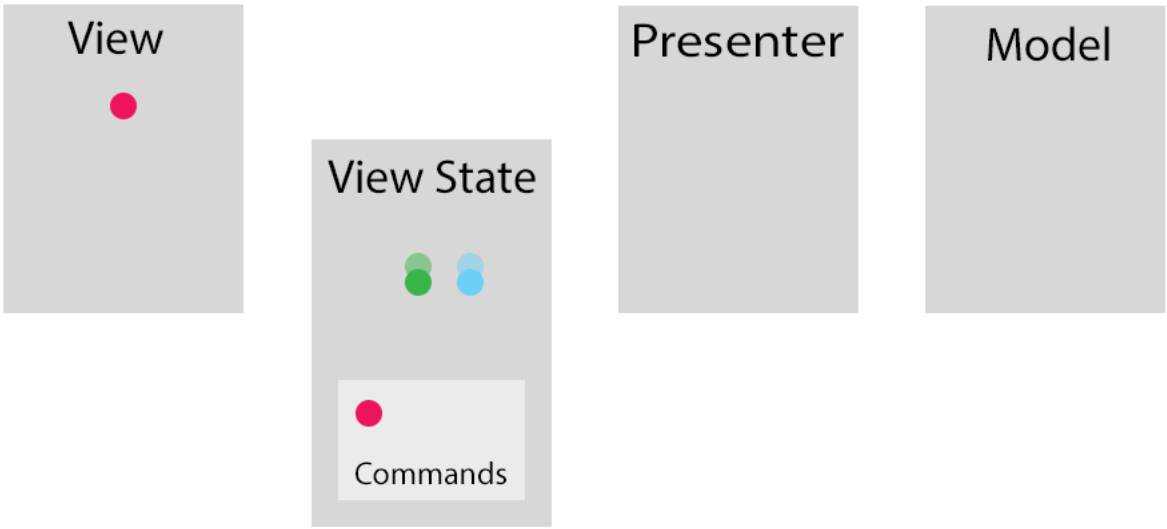
На данный момент ни одно из существующих решений не умеет делать все эти пункты одновременно. Как нам сперва показалось, больше всего нам подходила библиотека Mosby. Но позже выяснилось, что при её использовании, нам пришлось бы писать слишк много кода, каждый раз. Особенно, для реализации первых двух пунктов из нашего списка требований. Поэтому было принято ре разработать собственное решение.

## Moxy – теория

Наше решение сильно отличается от всех прочих(даже сама концепция *MVP* была модернизирована) тем, что между *View* и *Preser* затесался *ViewState*. Причём он там абсолютно необходим. Он отвечает за то, чтобы каждая *View* всегда выглядела именно так, ка хочет *Presenter*. *ViewState* хранит в себе список команд, которые были переданы из *Presenter* во *View*. И когда „новая" *View*

присоединяется к *Presenter*, *ViewState* автоматически применяет к ней все команды, которые *Presenter* выдавал раньше. Таким обр
получается, что не зависимо от того, что произойдёт со *View* по вине Android, *View* останется всё-равно в правильном состоянии. ,
этого вам нужно будет только привыкнуть изменять *View* исключительно командами из *Presenter*. Заметим, что это одно из основн
правил *MVP* и распространяется не только на Moxy.

Схематичная иллюстрация того, как это работает:




Что происходит на этой схеме:

1. Во *View* происходит событие , которое передаётся в *Presenter*
2. *Presenter* передаёт команду во *ViewState*
3. *Presenter* стартует асинхронный запрос в *Model*
4. *ViewState* складывает команду в очередь команд, после чего передаёт её во *View*
5. *View* приводит себя в состояние, указанное в команде
6. *Presenter* получает результат запроса из *Model*
7. *Presenter* передаёт во *ViewState* две команды и
8. *ViewState* сохраняет команды и в очередь команд и передаёт их во *View*
9. *View* приводит себя в состояние, указанное в командах и
10. Новая/пересозданная *View* присоединяется к уже имеющемуся *Presenter*
11. *ViewState* передаёт сохранённый список команд в новую/пересозданную *View*
12. Новая/пересозданная *View* приводит себя в состояние, указанное в командах , и

## Moxy – возможности

У Moxy есть несколько весомых преимущества перед другими решениями:

- *Presenter* не пересоздаётся при пересоздании Activity(это в разы упрощает работу с многопоточьностью)
- Автоматизация полного восстановления того, что видит пользователь при пересоздании Activity(в том числе при динамическо добавлении элементов Android View)
- Возможность из одного *Presenter* менять сразу несколько *View*(на практике оказалось чрезвычайно удобно)

Для этого в Moxy есть несколько механизмов, которые можно комбинировать между собой так, как вам будет угодно. Самыми весомыми механизмами являются аннотации, на основании которых генерируется код. А во время исполнения программы, инстру

под название `MvpDelegate` начинает полноценно использовать сгенерированный код.

Доступны следующие аннотации:

- @InjectPresenter – аннотация для управления жизненным циклом *Presenter*
- @InjectViewState – аннотация для привязывания *ViewState* к *Presenter*
- @StateStrategyType – аннотация для управления стратегией добавления и удаления команды из очереди команд во *ViewState*
- @GenerateViewState – аннотация для генерации кода *ViewState* для определенного интерфейса *View*

Обо всём этом далее.

### Moxy – MvpPresenter

Каждое приложение содержит в себе какую-то бизнес-логику. В концепции *MVP*, вся бизнес-логика располагается в *Presenter* и в *Model*. По факту это значит, что вы практически не программируете во *View*. Для того, чтоб ваш *Presenter* **не** превратился в God Ob нужно разделять каждый отдельный блок бизнес-логики в отдельный *Presenter*. В таком случае у вас получится много *Presenter*, н будут очень простыми и понятными. Например, если у вас на одном экране было две бизнес-логики, а затем они разошлись на 2 р экрана, то вы просто измените View. А *Presenter* какими были, такими и останутся. Так же, в этом случае вы сможете легко переиспользовать один *Presenter* в нескольких местах(например, BasketPresenter, сквозной через всё приложение). Ещё это упрос тестирование кода – вы просто проверите небольшой *Presenter*, что он всё делает правильно.

Для *Presenter* в Moxy заведен класс `MvpPresenter<View extends MvpView>`. В `MvpPresenter` содержится экземпляр *ViewState*, который тоже время должен реализовывать тот самый тип `View`, который пришёл в `MvpPresenter`. Доступ к этому экземпляру *ViewState* мож получить из метода `public View getViewState()`. А во время разработки вы не думаете, что работаете со *ViewState*, а просто даёте этот метод команды для *View*, как ей измениться. Так же есть методы для привязывания/отвязывания *View* от *Presenter*(`public voi attachView(View view)` и `public void detachView(View view)`). Обратите внимание на то, что к одному *Presenter* может быть привяза несколько *View*. Они будут всегда иметь актуальное состояние(за счёт *ViewState*). А если вы хотите, чтобы привязывание/отвязыва *View* проходило не через стандартное поле *ViewState*, то можете переопределить эти методы и работать с пришедшей *View* как хо *Например, вы можете захотеть использовать нестандартный ViewState, который не реализует интерфейс View, если вам нужно.*

В классе `MvpPresenter` так же есть интересный метод `protected void onFirstViewAttach()`. Очень важно понять, когда этот метод б вызван и зачем он нужен. Этот метод вызывается тогда, когда к конкретному экземпляру *Presenter* первый раз будет привязана л *View*. А когда к этому *Presenter* будет привязана *другая View*, к ней уже будет применено состояние из *ViewState*. И здесь уже не в эта *новая View* – совсем другая *View*, или пересозданная в результате смены конфигурации. Этот метод подходит для того, чтобы, например, загрузить список новостей при первом открытии экрана списка новостей.

В момент, когда во *View* пришла команда, вам может потребоваться понять, это новая команда, или это команда для восстановлен состояния? Например, если это свежая команда, то нужно применить команду с анимацией. А иначе не надо применять анимацию Можно это сделать через разные StateStrategy, или через сложные флаги в `Bundle savedState`. Но правильным решение будет использовать метод *Presenter*(или *ViewState*) `public boolean isInRestoreState(View view)`, который сообщит вам, в каком состоян находится конкретная *View*. Таким образом вы сможете понять, нужна ли вам анимация, или нет.

### Moxy – MvpView и MvpViewState

Самым простым компонентом *MVP* является *View*. Вам нужно завести интерфейс, который наследуется от интерфейса-маркера `Mv` и описать в нём методы, которые будет уметь выполнять *View*. В дополнение ко *View*, наша библиотека имеет сущность *ViewState*, которая непосредственно связана со *View*. *ViewState* является наследником `MvpViewState<View extends MvpView>`. Он управляет одн или несколькими, *View*(все одного типа `View`). И каждый раз, когда во *ViewState* приходит команда из *Presenter*, *ViewState* отправля всем *View*, о которых он знает. Также у `MvpViewState` есть метод `protected abstract void restoreState(View view)`, который будет вызван когда какая-нибудь *View* будет пересоздана, или когда к *Presenter*ко *ViewState* будет привязана новая *View*. Именно после как выполнится этот метод, „новая" *View* примет нужное состояние.

Стоит заметить, что `MvpViewState` хранит в себе список всех привязанных к нему *View*. И будет хорошо, если вы не будете забывать отвязывать *View*, которые уже уничтожены. Но если вы вдруг забудете это сделать, сильно не переживайте – в `MvpViewState` храня прямые ссылки на *View*, а `WeakReference`, что всё-таки поможет GC. А в случае, если вы используете такой механизм, как *MvpDeleg* то можете не беспокоиться об этом – он как привязывает *View* к *Presenter*, так и отвязывает их.

### Moxy – @GenerateViewState и @InjectViewState

Так как *ViewState* в большинстве случаев является довольно однообразной прослойкой между *View* и *Presenter*, был написан генер кода, который сделает за вас всю грязную работу. Применяя аннотацию @GenerateViewState к вашему интерфейсу View, вы получи сгенерированный класс *ViewState*. И чтобы вам не пришлось в *Presenter* самостоятельно искать и создавать экземпляр этого клас есть аннотация @InjectViewState. Достаточно просто применить её к классу вашего *Presenter*. Дальше `MvpPresenter` сам всё сделае

создаст экземпляр этого *ViewState*, сложит его себе в качестве поля и будет везде использовать его. Вам же просто останется ра с методом `public View getViewState()` из `MvpPresenter`.

В том случае, если вы не хотите использовать @GenerateViewState, но ваш *ViewState* реализует интерфейс *View*, вы можете по пре использовать аннотацию @InjectViewState. В таком случае, передайте в эту аннотацию, в качестве параметра, класс вашего *ViewS*

- Будьте аккуратны при применении аннотации @InjectViewState к типизируемому Presenter.
- Также учтите, что интерфейс, аннотированный @GenerateViewState должен быть не типизированным

### Moxy – StateStrategy для команд во *ViewState*

По умолчанию, все команды для *View* сохраняются во *ViewState* просто в том порядке, в котором они туда поступали. И после того команды были применены, они продолжают лежать в этой очереди. Но это поведение можно поменять, применяя аннотацию @StateStrategyType к интерфейсу *View* и к его методам. На вход эта аннотация получает параметр, в котором вы должны указать к `StateStrategy`, который вы хотите использовать. Если применить эту аннотацию ко всему интерфейсу *View*, то те методы, для кото стратегия не указана, будут использовать эту стратегию.

StateStrategy управляет очередью команд через два метода: `void beforeApply` и `void afterApply`. Первый метод будет вызван пере, как команда будет отправлена во *View*(метод `beforeApply` будет вызван сразу, как только поступит какая-то команда из *Presenter*). этом месте, в стратегии, указанной по умолчанию, и происходит добавление команды в очередь. Второй метод `afterApply` будет в каждый раз, когда команда будет применена ко *View*. И в первом, и во втором методе вы можете менять список команд как хотите

Давайте рассмотрим стратегии, которые уже реализованы в Moxy:

- AddToEndStrategy – добавит пришедшую команду в конец очереди. *Используется по умолчанию*
- AddToEndSingleStrategy – добавит пришедшую команду в конец очереди команд. Причём, если команда такого типа уже есть в очереди, то уже существующая будет удалена
- SingleStateStrategy – очистит всю очередь команд, после чего добавит себя в неё
- SkipStrategy – команда не будет добавлена в очередь, и никак не изменит очередь

Если же у вас какая-то специфичная логика и вам не хватает этих стратегий, то вы можете сделать свою стратегию. В этом случае поможет механизм тегирования методов. В аннотацию @StateStrategyType можно передать параметр tag(по умолчанию является названием метода). Затем, по этому тегу, вы сможете в методах `void beforeApply(List<ViewCommand<View>> currentState, ViewCommand<View> incomingCommand)` и `void afterApply(List<ViewCommand<View>> currentState, ViewCommand<View> incomingCommand)` понять, что за `ViewComand` вам пришли(из метода `ViewCommand String getTag()`).

Перед написанием своих стратегий, посмотрите на код уже реализованных – возможно он будет вам полезен.

### Moxy – MvpDelegate и жизненный цикл MvpPresenter

Сам по себе, *Presenter* нигде не создаётся, нигде не хранится и ниоткуда не достаётся. И чтобы вам не пришлось ничего придумыв для решения этих задач, мы сделали такой механизм, как `MvpDelegate`. Он следит за тем, чтобы там, где есть его экземпляр, были правильно инициализированы все *Presenter*. Для этого от вас требуется только передать в него все основные моменты жизненного цикла вашей *View*. Посмотреть какие методы когда вызывать, вы можете в классе `MvpActivity` или `MvpFragment`.

Для того, чтобы `MvpDelegate` нашел все *Presenter*, вы должны отметить их аннотацией @InjectPresenter. Эта аннотация очень мощна Через неё вы можете управлять тем, сколько времени будет жить *Presenter*. Если вы хотите, чтобы *Presenter* жил только пока есть в которой он содержится(+ пока происходит смена конфигурации), то просто добавьте эту аннотацию к полю *Presenter*. В случае, вы хотите, чтобы *Presenter* жил не зависимо от того, кто и когда на него подписан, вам нужно будет сделать две вещи. Первое – ну сообщить `MvpDelegate`, что *Presenter* не привязан к жизненному циклу того, кто его запросил. Для этого, нужно выставить значение параметра type аннотации @InjectPresenter как PresenterType.GLOBAL. Второе – вы должны передать `MvpDelegate` информацию, по которой он сможет найти нужный вам *Presenter* в хранилище всех *Presenter*. Есть два варианта, как это сделать:

Первый вариант. В аннотации @InjectPresenter вы выставляете значение для параметра tag. Тогда *MvpDelegate* попытается найти в глобальном хранилище *Presenter* с таким тэгом. Если он его найдёт, то просто установит его в это поле. Иначе он создаст подходя *Presenter*, сложит его в хранилище, и установит его в это поле. С учётом того, что к одному *Presenter* может быть привязано неско *View*, этот механизм открывает очень много возможностей перед вами.

Второй вариант(для парметризированного тэга). По сути, он похож на первый вариант. Отличие лишь в том, что во втором случае можете заранее знать, какой тэг будет у *Presenter*. Т.е. тэг должен генерироваться динамически. Тогда вам придётся немного постараться:

1. Создайте свою реализацию `PresenterFactory`

2. В аннотацию @InjectPresenter установите параметры:
   - В factory установите класс вашей `PresenterFactory`
   - В presenterId установите строковый идентификатор *Presenter*(это нужно для того, чтобы различать в одном классе *Presente* одинаковыми фабриками)
3. Заведите свой интерфейс, содержащий один и только один метод, который будет возвращать параметр для factory нужного ти
   - Аннотируйте этот интерфейс как @ParamsProvider(PresenterFactoryClass), передав аннотации, в качестве параметра, класс вашей `PresenterFactory`
   - Опишите метод, который будет возвращать параметр, должен на вход получать один параметр `String`(в этот параметр при тот самый параметр presenterId из аннотации @InjectPresenter)
4. Объект, который содержит *Presenter*, в аннотации @InjectPresenter которого указанна эта `PresenterFactory`, обязан реализовыв созданный в п.п. 3. интерфейс

Здесь вам стоит знать, что вам не показалось, что это место слишком запутанно. Так и есть, оно запутано. Просто знайте, что есл потребуется такая функциональность, следуйте этому небольшому списку правил, и вы сами всё поймёте и у вас всё получится.

Кроме указанной выше функциональности, `MvpDelegate` умеет быть родительским/дочерним делегатом для другого. Это необходи того, чтобы вы могли автоматизировать жизненный цикл *Presenter* не только внутри Activity/Fragment, но и внутри других элемент которых нет самостоятельного жизненного цикла(например, в адаптере или даже в `ViewHolder` элемента адаптера). Если вы устан для одного `MvpDelegate` в качестве родительского другой `MvpDelegate`, то делегат-потомок будет получать все события жизненного цикла делегата-родителя. Для этого просто вызовите у целевого `MvpDelegate` метод `public void setParentDelegate(MvpDelegate delegate, String childId)`. В качестве `delegate` он ожидает получить родительский `MvpDelegate`. В качестве `childId`, вы должны ука уникальный идентификатор, по которому локальные *Presenter* одного делегата-потомка будут отличаться от локальных *Presenter* другого делегата-потомка.

Отметим, что если у родительского `MvpDelegate` уже был вызван метод `onCreate`. то вам необходимо самостоятельно вызвать мето `onCreate` у делегата-потомка. Почему это важно? Чтобы это понять, разберёмся, как работает `MvpDelegate`.

`MvpDelegate` кроме того, что управляет инициализацией полей *Presenter*, он делает ещё одну очень важную вещь. Он привязывает и отвязывает *View* от *Presenter*. Привязывание *View* к *Presenter* происходит в методе `onStart`, а отвязывание – в методе `onDestroy`. *У `Fragment` немного по другому, см. на github.*

▸ Почему именно в этих методах?

`MvpDelegate` использует специальное хранилище для *Presenter*. Доступ к этому хранилищу он получает через `MvpFacade`. `MvpFacade` – содержит в себе хранилище *Presenter* и некоторые другие элементы, призванные помочь *MvpDelegate* делать его работу оптимальн Не смотря на то, что *MvpFacade* является синглтоном, будет здорово, если вы выполните его метод `public static void init()` например, в методе `onCreate()` вашего Application. Или вы можете наследовать ваш Application от `MvpApplication`, поставляемого в Тогда в момент, когда `MvpDelegate` обратится к этому синглтону, он уже будет готов к работе.

### Moxy – *Model*

Важным элементом *MVP* является *Model*. Но в Moxy эта часть *MVP* никак не затронута. Всё дело в том, что в этом нет смысла. В ка проекте свои требования к *Model*. Где-то *Model* это просто набор классов для работы с API и сама работа с API(например, через Retrofit). Где-то в *Model* входит ещё и дополнительная бизнес-логика. В каких-то проектах актуально использование подхода Clean Architecture. В таком случае внутри *Model* появляются дополнительные сущности, например, *Interactor* и *Repository*. А с учётом того *Presenter* полностью отвязан от жизненного цикла Activity, вы можете спокойно создавать экземпляр конкретной *Model* внутри *Pre* и работать с ним. Используя DI вы можете подключать нужную *Model* в *Presenter*. А в будущем, используя тот же DI, спокойно подм *Model* для тестов.

В любом случае, крайне удобно для работы с *Model* использовать Rx. Тогда вы можете сделать так, чтобы публичные методы *Mode* возвращали `Observable`. В таком случае будет легко сделать взаимодействие *Model⇒Presenter*, и в то же время *Model⇔Model*. Это д возможность легко сделать параллельное исполнение запросов из *Presenter* в *Model*.

## Moxy – итого

В результате мы имеем библиотеку, которая решает все проблемы жизненного цикла. Вы всегда будете показывать пользователю именно то состояние, которое для него актуально, и в то же время, вам не придётся делать ничего лишнего. Только опишите все команды для *View* отдельными методами. И избегайте изменения *View* из самого *View*. Если вы показали диалог командой из *Prese* то и при закрытии диалога, должна быть команда из *Presenter*. Иначе *ViewState* снова покажет вам диалог после смены конфигура

Хотелось бы заметить, что библиотека никак не ограничивает вас в выборе реализации многопоточности в вашем приложении. Вы

можете использовать Rx, AsyncTask, Thread, Executor. Главное, будьте аккуратны, работайте со *View* только с главного потока. Ещё Moxy не решит проблем с `commit()` фрагментов после выполнения `onSaveInstanceState()`. Поэтому не забудьте закрывать транзакц используя `commitAllowingStateLoss()`. Так же, она не решит проблем с утечкой памяти – если вы передадите ссылку на Context/Activity/Fragment в *Presenter*(а потом ещё и во *ViewState*), то память может утечь. Будьте аккуратны.

### Полезные материалы

Moxy не получилась бы такой, какой она получилась, если бы не многочисленные труды других людей. Вот некоторые из них:

- Android Application Architecture (Android Dev Summit 2015)
- Android Testing Codelab – тема *MVP* затронута не сильно, но можно что-то для себя почерпнуть. Так же можно посмотреть, как тестировать *MVP*.
- Nucleus – пример реализации *MVP*, с замашкой на обработку жизненного цикла.
- Mosby – лучшая реализация *MVP* до релиза Moxy. Отлично расписаны сами принципы *MVP*, которые критично понять.
- Old Mosby – руководство к первой версии Mosby. Крайне полезное для понимания того, что такое *MVP*.
- STINSON'S PLAYBOOK FOR MOSBY – набор советов, который очень поможет вам определиться с некоторыми понятиями *MVP*. Т дополнительно объяснит, какая часть программы, каким компонентом должна стать.
- Android Reactive MVP: практика – ещё одно видение, как должна выглядит структура Android-приложения, построенного на *MV*
- Andrtoid Clean Architecture – поможет понять, что такое модель, и на какие компоненты её можно разложить.
- Алексей Макаров. Speaker Clean Architecture и MVP – хороший доклад про Clean Architecture, и хорошие вопросы в конце.
- Mosby issues 85 – помогает понять, что из себя должен представлять Repository.

### Moxy – где брать

Чтобы подключить Moxy в свой проект, просто добавьте её в зависимостях. Moxy состоит из трёх частей. Одна из них отвечает за предоставление вам Moxy SDK. Её довольно просто подключить:

```
dependencies{
    ...
    compile 'com.arello-mobile:moxy:1.1.1'
}
```

Если вы хотите иметь доступ к таким вспомогательным классам, как `MvpApplication`, `MvpActivity` и `MvpFragment`, так же подключите `android`:

```
dependencies{
    ...
    compile 'com.arello-mobile:moxy-android:1.1.1'
}
```

Другая часть отвечает за обработку аннотаций и занимается генерацией кода. И здесь вам нужно определиться.

Если у вас нет никаких особых требований, ваш проект – обычный Android-проект, и вы не хотите, чтобы сгенерированный код был доступен из вашего когда, то подключите зависимость так:

```
dependencies{
    ...
    provided 'com.arello-mobile:moxy-compiler:1.1.1'
}
```

Если же вы хотите иметь прямой доступ к сгенерированному коду, то стоит использовать android-apt:

1. Модифицируйте build.gradle вашего проекта:

```
buildscript {
    dependencies {
        classpath 'com.neenbedankt.gradle.plugins:android-apt:1.4'
    }
}
```

2. Модифицируйте build.gradle вашего приложения:

```
apply plugin: 'com.neenbedankt.android-apt'

dependencies {
    ...
    apt 'com.arello-mobile:moxy-compiler:1.1.1'
}
```

Исходники библиотеки можно найти на Github: https://github.com/Arello-Mobile/Moxy

Полноценный пример приложения, использующего Moxy: https://github.com/Arello-Mobile/Moxy/tree/master/sample-github

В момент, когда мы соберём репрезентативный список вопросов по нашей библиотеке, по тому, как её использовать, по MVP в це будет сделана отдельная статья, в которой будут освещены самые популярные/интересные вопросы. Вопросы можно задавать зде комментариях, писать мне(@senneco ) и ещё одному автору библиотеки – @Xanderblinov. Или можете обращаться ко всему отделу Android-разработки Arello Mobile, написав на java-developers@arello-mobile.com.

От авторов библиотеки Moxy
@senneco и @Xanderblinov

**Метки:** mvp, android, android development

+15 274 64,3k 46

18,0 Карма 0,0 Рейтинг 10 Подписчики

Юрий @senneco
Android team leader

Поделиться публикацией

ПОХОЖИЕ ПУБЛИКАЦИИ

15 мая в 16:02
**Google I/O 2017: заметки с места событий от Android-разработчика**
+9 3,9k 15 0

29 августа 2016 в 14:35
**MVP на стероидах: заставляем робота писать код за вас**
+17 16,2k 102 9

9 марта 2016 в 11:06
**Расширяемый код Android-приложений с MVP**
+12 32,7k 177 9

## Комментарии 46

**forceLain** 12.02.16 в 19:12

Очень хороший подход! Очень забавно, что мы, создавая свою MVP либу, написали все до безобразия похоже. Я даже вижу проблемы, кото натолкнули вас на те или иные решения: тэгирование View, что бы иметь возможность переаттачить к ней презентер после поворота экран MVP-делегаты, что бы решить проблему отсутствия множественного наследования в Java и уже имеющиеся несопоставимые Fragment/DialogFragment, Activity/AppCompatActivity и т.п. (правда их мы подсмотрели в Nucleus), ViewState, что бы облегчить страдания при пересоздании View/Fragment/Activity из небытия, правда мы её назвали StateModel :)
Чем хочу поделиться из наших решений:

1) ViewState умеет сама записывать себя в Bundle savedState по умолчанию используя стандартный механизм серелизации java, но есть возможность повлиять на это переопределив в нужный метод в своей конкретной ViewState.

2) для связи между View -> Presenter -> ViewState -> View мы используем RxJava и её PublishSubject и BehaviorSubject, из которых легко строится очередь из событий и ожидание на их публикацию. Во-первых, это помогает вьюшке, подписавшись на изменения из View получи правильное состояние даже если ответ от сервера пришел как раз в тот момент, когда активити еще была в процессе пересоздания из-за поворота, например. Во-вторых, коммуникации между View, Presenter и ViewState защищены от эксепшенов в том смысле, что если что-то случится — мы то 100% вероятностью получим это в onError, даже если забыли поставить проверну на NPE. Ну и в-третьих, все подписки V на ViewState отписываются через делегат и по этому нигде ничего не течет :)

> **senneco** 12.02.16 в 19:22
>
> У вас видимо ViewState хранится во View, поэтому вы вынуждены сериализовывать его и складывать в Bundle?
>
> Мы решили развязать пользователю руки, и поэтому ссылка на ViewState хранится в Presenter. Presenter в свою очередь хранится не в Activity, а в статичном хранилище. Это позволяет не зависеть Presenter(а значит и ViewState) от жизненного цикла View. И поэтому даже команда во ViewState прилетела в то время, когда View не приаттачена к Presenter/к ViewState, как только View будет приаттачена, ViewS сообщит ей весь набор команд, которые она должна выполнить. За счёт этого можно из Presenter передавать в командах даже несериализуемые данные.
>
> А если вы это и говорили, то круто, что мы не одни так подумали =)
>
> И да, у нас идёт тэгирование не View, а Presenter ;)
>
> > **forceLain** 12.02.16 в 19:42
> >
> > Да, если вдаваться в детали, мы положили ViewState в активити для того, что бы иметь возможность «вернуть все как было» не только повороте экрана, но и при пересоздании всего процесса приложения (весьма частый кейс в Android 6 со своими новыми runtime пермишеннами). Конечно после таких издевательств над процессом в нем не останется никаких презентеров в статичном хранилище вот весь ActivityTask андроид нам любезно восстанавливает и отдает savedState, а там наша ViewState лежит себе :)
> > А для «легкого» пересоздания View, как в случае с поворотом, мы как раз так же используем хранилище презентеров. Вот только онс статичное, а создается и лежит внутри Application. А тегирование для View мы используем не только для того, что бы автоматически переаттачить тот же презентер к новой View, но и для того, что бы разделять одинаковые Activity в одном AcivityTask и вешать им разн презентеры. Это на примерно такой случай: открыть активити «чатик с другом», из неё открыть активити «список друзей друга», а и открыть еще одну активити «чатик с еще одним другом». В итоге получим первую и последнюю активити одного класса, но чатик там должен быть разный, соответственно, и презентеры тоже разные.
> >
> > > **senneco** 12.02.16 в 19:55
> > >
> > > Понятно, а мы решили, что раз процесс убился, и всё-равно потерялись все Presenter, то просто пусть заново будет создан Present всё начнётся сначала. Я замечал, что у стоковых Android-приложений именно такое поведение =)
> > >
> > > Да, у нас тоже легко сделать кейс что на другой активити такого же типа будет использоваться другой Presenter =) Вообще, изначa все Presenter – локальные. И, соответственно, на каждый экран свои Presenter. А вот если указать глобальный тэг, то будет использоваться везде один Presenter. Ну и спец. фишка – динамический тэг для глобального презентера. Например, открыли списо своих контактов → создался Presenter для нашего списка контактов. Затем открыли список контактов друга → создался Presenter д списка его контактов. Затем вернулись к своему списку контактов, и тут уже не создаётся новый Presenter, а берётся старый. Акту может быть, например, если эти Presenter очень долго отрабатывают и будет обидно потерять их.
> > >
> > > А ваше решение где-нибудь опубликовано? Было бы интересно посмотреть =)
> > >
> > > > **forceLain** 12.02.16 в 20:04
> > > >
> > > > Интересно, что вы еще написали себе аннотации. Лично мне не очень хотелось писать собственные кодогенерирующие аннотац для инжекта, на крайний случай хватает инжектов из даггера, но раз у вас есть, наверно стоит взглянуть на них тоже.
> > > >
> > > > > А ваше решение где-нибудь опубликовано?
> > > >
> > > > Ага, на внутрикомпанейском гитлабе :) Вероятно, когда нибудь оформим и в общий доступ

**senneco** 12.02.16 в 20:11

Да, мы очень хотели, чтоб пришлось писать минимум кода. И в то же время хотелось попробовать annotation processor =) Результат крайне порадовал – для полноценного сохранения состояния достаточно применить аннотацию @GenerateViewState MvpView и @InjectViewState к MvpPresenter. Когда видишь этот код и результат его работы, кажется что там есть магия =)

Правда, если можно обойтись без кодогенерации/рефлексии, используя только наследование/композицию, это наверное даж круче.

**senneco** 14.02.16 в 20:22

Вот как это можно сделать в moxy:

- в каждом методе View сохранять в Bundle какое-то описание состояния
- складывать этот Bundle в outState
- в onCreate передавать этот Bundle в Presenter
- в Presenter смотреть в метод onFistViewAttached, есть ли Bundle
- если есть Bundle, «парсить» его и давать команды во ViewState

У этого способа есть минус – он не автоматизирован. Но есть и плюс – лишний раз Bundle парситься не будет. А вы как-нибудь автоматизировали создание сериализуемого ViewState?

**forceLain** 15.02.16 в 07:15

У нас View только отражает состояние ViewState и передает клики и т.п. презентеру. ViewState изменяясь сообщает об этом Viev View уже показывает прогрессы или пезультаты или еще чего. В onSaveInstanceState базоый презентер серелизует ViewState в б стандартным ObjectOutputStream и потом достает из бандла в onCreate() стандартным же ObjectInputStream. Этого хватает для большинства экранов. Если на каком то экране во ViewState требуется положить что-то такое, чего не стоит серелизовать этими средствами, то можно переопределить серелизацию/десерелизацию конкретно для этого экрана и пары ViewState-Presenter.

**pavel_dolbik** 15.02.16 в 17:36

Подскажите, пожалуйста, как правильно передать данные из View в Presenter.

Предположим:

- есть ActivityTest
- из Intent-а получаем значение
- в Presenter

```
protected void onFirstViewAttach() {
    super.onFirstViewAttach();
    getViewState().start();
}
```

- во View

```
@Override
public void start() {
    presenter.start(getIntent().getExtras().getInt(Constants.VALUE));
}
```

Если так сделать, то при каждом перевароте экрана, будет каждый раз отрабатывать presenter.start(value), а нужно, только один раз.

**senneco** 15.02.16 в 18:28

Если вам нужно, чтоб команда отрабатывала исключительно один раз, значит она не должна быть сохранена во ViewState. Для этого у не должна быть стратегия SkipStrategy. Её можно указать, применив к методу `start` в интерфейсе View аннотацию: @StateStrategyType(SkipStrategy.class)

Ещё на заметку, ваш код можно изменить:

- в activity, в методе `onCreate` выполните ваш код `presenter.setStartValue(getIntent().getExtras().getInt(Constants.VALUE));`
- в presenter, в методе `setStartValue` сохарните пришедшее значение где-нибудь в presenter

- в методе `onFirstViewAttach` берёте это значение и работаете с ним

Но это не обязательно – ваш подход абсолютно так же будет работать. Просто имейте ввиду возможность такого способа =)

Учтите, что метод `onFirstViewAttach` будет вызван только при первом привязывании view. А после поворота девайса, он уже не будет вызван. Но похоже вы это и так поняли =)

 **pavel_dolbik** 23.02.16 в 08:06

Как правильно поступить в такой ситуации:

Есть ViewPager в котором находятся 2 фрагмента (Fragment 1, Fragment 2). UI и логика фрагментов идентичны, отличие только в выборке да из БД.

Вопрос: Можно как-то при переходе с одного фрагмента на другой обнулять/сбрасывать ViewState. Т.е. если на Fragment 1 было показано диалоговое окно и которое должно быть показано при перевороте устройства, то при переходе на другой фрагмент, Fragment 2 должен бы первоначальном состоянии.

В такой ситуации нужно делать один presenter и при переходе между фрагментами обрабатывать события. Или лучше сделать 2 разных presenters, но тогда получим дублирование кода.

 **senneco** 23.02.16 в 18:18

В большинстве случаев, здесь будет достаточно сделать так, чтоб на каждую страницу ViewPager был свой Presenter. Так вам будет прои всего – не нужно будет ничего разруливать.

В таком случае, каждый Fragment будет по-своему инициализировать свой Presenter, а Presenter будет уже доставать нужные данные. И будет очень просто обработать команды из Presenter, и оба фрагмента будут независимы друг от друга.

 **nwalker** 24.02.16 в 23:31

А вот предположим, мы хотим создать в своем коде отдельную сущность Router, тот самый, из соседней статьи про VIPER, и вызывать его методы из презентеров. Естественно, хочется обойтись без бойлерплейта, но не очень понятно как, роутеру для запуска активити нужен контекст — текущая активити.
Простое и некрасивое решение — явно вызывать Presenter.start(this) в OnCreate/OnStart. А есть ли решение красивее?

 **senneco** 25.02.16 в 07:43

Главный вопрос, который нам здесь нужно решить – а где будет жить экземпляр Router?

Если он будет жить и использоваться только во View, то никаких проблем нет – у нас есть напрямую доступ к Activity.

Сложней, если ссылка на Router нужна внутри Presenter. В таком случае вам не обойтись без явной передачи "чего-то" из View в Present Здесь вы встаёте перед другим выбором: что передать из View? Context? А если Router будет не стартовать Activity, а менять фрагменты Тогда придётся передавать что-то другое. Таким образом само собой напрашивается решение из соседней статьи: из View вы устанавливаете в Presenter непосредственно экземпляр Router, с которым в будущем будете работать из Presenter.

Мне кажется, если реализовывать VIPER, то нужно идти по второму пути и просто в onCreate передавать в Presenter экземпляр Router. В таком случае хотелось бы обратить внимание на две вещи:

1. Не забудьте убирать Router из Presenter, когда View уничтожается(иначе будет утечка памяти)
2. Вы можете расширить функционал MvpDelegate, добавив в метод `onCreate` указывание Router для Presenter, и очищая ссылку на Ro в Presenter внутри метода MvpDelegate `onDestroy`

PS: Router ломается, если вы начинаете строить приложение не на фрагментах, а на custom view, т.к. при смене конфигурации вы потеря все изменения лэйаута. В таком случае не используйте Router, а работайте прямыми командами во View из Presenter через ViewState. Тс вы не потеряете ваши изменения после изменения конфигурации.

 **nwalker** 25.02.16 в 18:57

Спасибо, вы мне помогли упорядочить мысли на тему навигации.

 **pavel_dolbik** 07.03.16 в 14:33

Подскажите пожалуйста.
Если использовать Presenter для Fragment и во Fragment установить setRetainInstance(true); то при переворроте экрана, Presenter не востанавливает состояние View.

Если же убрать setRetainInstance(true); то при перевороте, происходит утечка памяти GC не освобождает память

 **senneco** 07.03.16 в 15:32

Интересное замечание. Действительно, состояние не будет восстановлено, т.к. у фрагмента не будет вызван метод onCreate(). Значит, в случае с retain-фрагментом можно поступить например так: вызывать метод делегата onCreate() не в onCreate() фрагмента, а где-нибудь другом месте. Например, в методе onCreateView(). Правда, метод onAttach() может быть более подходящим местом, но я с ходу не могу ручаться за вызовы этого метода у retain-фрагментов.

А если делать не retain-фрагмент, то утечки памяти не должно быть – при вызове метода onDestroy() у фрагмента, он будет отвязан от презентера. В то же время в презентере хранятся weak references на View, поэтому утечки не должно быть. Может быть вы как-то самостоятельно храните ссылку на фрагмент где-нибудь в презентере?

**pavel_dolbik** 07.03.16 в 22:03

Спасибо за советы!

**pavel_dolbik** 11.03.16 в 10:44

Еще небольшой вопрос:

При загрузке данных — показываем диалоговое окно с прогрессом ( progress)
Что-то пошло не так, скрываем progress и показываем сообщение об ошибке ( error )
Пользователь прочитал сообщение об ошибке и закрыл error.

Так как это все делается через presenter, все сохраняется во view state
И при перевороте экрана отработают все 4 метода:

1. Показать progress
2. Скрыть progress
3. Показать error
4. Скрыть error

Отработает целых 4 метода ( при перевороте ), но по сути они отработают впустую, т.к. пользователь в такой ситуации не должен ничего увидеть.

Вопрос:
Можно и нужно ли как-то очищать очередь во view state?
Если будет вызвано гораздо больше методов, не приведет ли это к ненужной трате ресурсов?

**nwalker** 11.03.16 в 16:58

Для этого предусмотрены стратегии, применяются либо на весь view-интерфейс, либо на отдельные методы, можно писать свои. Подро в посте и исходниках.

**senneco** 11.03.16 в 17:28

Эти команды применятся очень быстро, что пользователь не почувствует, что применилось несколько команд.

Но в случае, если вы хотите, вы можете написать свою стратегию, которая будет удалять команду, к которой текущая является противодействием. Или же, если новая команда приводит View в такое состояние, что все предыдущие команды становятся точно не нужными, то можно применять стратегию SingleStateStrategy. Например если есть команда showData, и нет swipe to refresh, то можно к применить эту стратегию, т.к. после того, как установили данные, точно не нужно ни ошибку показывать, ни прогресс.

**pavel_dolbik** 11.03.16 в 14:48

@GenerateViewState — сейчас deprecated

**senneco** 11.03.16 в 17:14

Да, именно так. Когда вы применяете аннотацию @InjectViewState, annotation processor понимает, ViewState какой View вы хотите использовать, и генерирует его, если его ещё нет.

**pavel_dolbik** 04.05.16 в 11:38

День добрый.
Очень интересное поведение наблюдал на Android 6.

Есть TestActivity и Presenter. В приложении есть permission, например на использование камеры.
TestActivity запущена, метод onFirstViewAttach() — отработал.
Затем, пользователь заходит в Settings и включает/выключает permission.
Возращается в TestActivity и метод onFirstViewAttach опять отрабатывает.

Когда включаешь/отключаешь permission getMvpDelegate().onDestroy(); не отрабатывает.

Почему onFirstViewAttach() отрабатывает?

**senneco** 04.05.16 в 12:29

Видимо, Android полностью останавливает приложение в таком случае. Тут вам не помогут даже глобальные presenter. Но похоже, что у activity должен был быть вызван метод onSaveInstanceState – в нём вы можете сохранить какие-нибудь флаги для presenter. А в onCreate передавать эти флаги в presenter(банально сделать метод init(Bundle args) в presenter), и уже в presenter решать, делать что-нибудь со vi onFirstViewAttach, или нет.

**forceLain** 04.05.16 в 12:39

Переключение настроек полностью убивает процесс приложение и создает новый, по этому все презентеры тоже убиваются. onSaveInstanceState()/onCreate() действительно вызываются, но не те, к которым мы привыкли, а новые, те что с 21 API: onSaveInstanceState(android.os.Bundle, android.os.PersistableBundle) и onCreate(android.os.Bundle, android.os.PersistableBundle)

**agent10** 02.09.16 в 01:01

Может не внимательно прочитал, но не очень понял есть ли принципиальное отличие ViewState от механизма самого Андроида(onsaveinstancestate/onRestoreState) для случая связи одной View для одного Presenter'а?

**senneco** 02.09.16 в 05:55

Я даже не знаю, есть ли ясный ответ «да» или «нет». Могу только рассказать в чём разница, а вы уже сами определитесь =)

ViewState хранится в Presenter. Presenter хранится в static-хранилище. Поэтому нет никакой обходимости передавать в команды serializa объекты. Можно складывать хоть что. Но в случае, если процесс будет уничтожен, то и static-хранилище с Presenter будет уничтожено. значит и все ViewState будут уничтожены.

В то же время, в Bundle saveState можно складывать только serializable-объекты(ну и примитивы со String). Выигрыш, понятно, в том, что процесс будет уничтожен, а потом восстановлен, мы сможем запросто достать команды из savedState. Но вот какая проблема: у вас мо быть команда showProgress(). И если пользователь будет видеть прогресс, то наш Presenter обязан загрузить данные. А это значит мы должны во время применения команды, ещё и начать что-то делать в Presenter. Но велика вероятность, что для этого нам придётся сохранить оочень много информации в команде. А это чревато запутанным кодом. В то же время, если мы «лениво» сохраняем команды сперва из savesState будет получена команда showProgress(), а сразу после этого могут идти две команды hideProgress() и showData(). И придётся как-то очень сильно исхитриться, чтоб Presenter перестал грузить данные, т. к. они уже есть.

И такой подход с Bundle как раз используется в Mosby(ну или почти такой). И это мне в Mosby и не понравилось – нужда каждый раз рук разруливать восстановление состояния View.

Ещё, @forceLain говорит, что они используют save state для хранения ViewState. Может, у них как-то по другому. Но я именно так вижу е использование =)

**agent10** 02.09.16 в 09:46

И ещё вопрос близкий к этому.
Я пока сам Moxy или MVP не использовал, а читаю только теоретически, но есть подозрение, что не для всего UI и не для всех экрано есть смысл использовать MVP подход. Согласны? И вот хотелось бы понять для каких?
К примеру ведь Андроид умеет сам восстанавливать View у которых прописан id, сетевой фреймворк Robospice или даже те же Loade могут отдавать ответ асинхронной задачи в новую активити после смены конфигурации, вроде как DialogFragment умеет восстанавли свой показ.
Т.е. встроенные средства есть и часто даже не надо писать дополнительный код.

Поэтому хочется понять какова должна быть сложность UI и его бизнес логики, чтобы стоимость использования MVP оправдала себя. Есть ли у вас какой-то набор правил исходя из имеющегося опыта?
Может быть была бы полезная статья с живыми примерами, типа «раньше было так и такие-то проблемы», а после внедрения Moxy « так и проблем больше нет».

**senneco** 05.09.16 в 21:23

За время использования MVP/Moxy убедился, что MVP нужно использовать тогда, когда на вашем экране есть логика. В таком случ выделив её в Presenter и Model, View становится максимально простой. И в то же время можно легко тестировать Model и Presenter

В то же время, MVP помогает очень сочно производить изменения дизаайна, рефакторинг кода. Например, был у вас DailogFragmer стал BottomSheetDialog, Snackbar или Toast(в зависимости от содержимого). И если интерфейс View был сделан максимально независимо от того, как выглядит View, то вы это сделает максимально быстро и просто. Или же поменялись какие-нибудь условия бизнес-логики. У вас опять же готовый интерфейс View, который не придётся менять. И это действительно так происходит, даже на маленьких проектах(2-3 месяца).

Так же, «Андроид умеет сам восстанавливать View у которых прописан id», но вот незадача – visibility он не восстановит =( И динамически добавленная View пропадёт.

Про Loaders я молчу – мы от них как раз убегали, когда создавали Moxy :D Но опыт использования Loaders тоже очень полезен. Пол всё – AsyncTask, Loader, Robospice, Rx, MVP, Moxy =) Главное, почувствовать когда и где что лучше использовать.

MVP показал себя с лучшей стороны – чрезвычайно дешево завести интерфейс для View и Presenter для минимальной бизнес-логик профит очень приятен, даже при малейших изменениях.

Статья ещё будет, и наверное не одна. Но вряд ли там найдётся место тому, «как было раньше», потому что раньше было перепробовано слишком много всего :D

PS: другие библиотеки, реализующие MVP толком не пробовал – хватало детального изучения сорцов+сэмплов, чтобы в чём-нибуд расстроиться. Mosby понравился больше всего =)

**Xanderblinov** 05.09.16 в 23:26

С первого взгляда кажетя, что использование MVP на экранах с одной кнопкой избыточно. Но, в последствии, начинаешь восприни MVP как философию от которой не хочется отходить. Более того, использование одного подхода повсеместно делает приложение консистентным и легким для поддержки.

Основные проблемы MVP подхода уже решены

- boilerplates связи презентеров и вью решается статической кодогенерацией шаблонов
- хранение состояние в презентерах решается динамической кодогенерацией

Так что смело используйте!

**Arvalon** 07.11.16 в 12:40

А можно инжектить Presenter в View в поле типа не конкретного Presenter'a, а интерфейса, который данный презентер реализует?
Вот пример:
▸ Интерфейсы и т.д.

**senneco** 07.11.16 в 13:04

На данный момент такой возможности нет. И пока я не представляю, как сделать такую возможность, чтобы одновременно это было и удобно, и безопасно. Потому что в таком случае вам обязательно придётся делать provide-метод, а это может быть не очевидно, или еш чем-нибудь не красиво.

Можно подумать на досуге, как можно сделать =)

**Arvalon** 07.11.16 в 13:45

Спасибо, я думал что чего-то не нашёл в фреймворке.
Надо же ведь стараться проектировать через интерфейсы а не через конкретную реализацию. Так и для тестирования в будущем есл потребуется подставлять другие презентеры — интерфейс и потребуется.

**dmdev** 07.11.16 в 14:10

А как вам поможет интерфейс презентера в тестировании? Протестировать вьюху? Но смысл MVP в том, чтобы вынести всю логику презентер. Подставлять другие презентеры во вью тоже сомнительная идея. Другой презентер — другая вью — другой интерфейс вью.

**Xanderblinov** 23.11.16 в 20:40

@senneco можем сделать дополнительный слой абстракции:

В каждой вью нужно будет подключаться к PresenterStore через метод MvpFacade.getPresenterStore

Дефолтная map Presenter с PresenterImpl будет генериться статической кодогенерацией и подаваться как параметр к PresenterSt

MvpFacade будет инициализировать либо тестовым PresenterStore, либо живым с дефолтной мапой.

Остался вопрос, есть ли в этом потребность и на сколько станет все сложнее для понимания)

**Xanderblinov** 23.11.16 в 20:46

@terrakok нужно третье мнение

**terrakok** 23.11.16 в 21:24

В MVP не предполагается использование разных презентеров для одной вью. Вью имеет доступ к конкретному презентеру.
Зачем там интерфейс? Какие публичные методы он будет скрывать? Для кого?
Мокси реализует чистый архитектурный подход. Не надо ломать принципы

**senneco** 24.11.16 в 05:46

Я не могу понять, что ты предлагаешь, и зачем?) Для управления инжекцией презентера? Как-то очень сложно подход выгляд

**Xanderblinov** 30.11.16 в 17:06

Да, излишне запутанно. Это тот самый случай когда надо выбирать простоту

**anton9088** 27.12.16 в 02:30

Можно было не делать такие костыли с повтором команд для View, а просто для каждого View определить модельный класс, в котором буд храниться все данные нужные для отображения, и делать во View не много методов для установки данных, а один для установки модели, то при пересоздании View достаточно установить сохраненную модель, а не повторять все действия.

Например для экрана авторизации в таком модельном классе могли бы быть поля — логин, пароль, флаг для какого-нибудь чекбокса. При создании View, Presenter передает во View модельный класс со значениями по умолчанию если модель еще не была создана, или уже существующую.

**anton9088** 27.12.16 в 02:58

+ использовать Data Binding, для автоматического изменения модели при вводе текста пользователем и других действиях

**senneco** 27.12.16 в 07:17

Конечно можно. Просто это тогда называется MVVM. Но мне больше по душе MVP. Спорить о том, что лучше — не вижу смысла ;) Тем более про это есть целая статья, и её писал человек, который использовал и тот, и тот подход.

**anton9088** 27.12.16 в 13:19

Не обязательно называть это MVVM, можно использовать и ViewModel и Presenter вместе. Даже в доке гугла про Data Binding есть упоминания про Presenter https://developer.android.com/topic/libraries/data-binding/index.html

**senneco** 27.12.16 в 13:34

Ну, можно назвать вообще как угодно =) И да, MVP можно использовать с Data Binding, при желании. Даже есть желание попроб такой подход. В таком случае можно не использовать аннотацию `@InjectViewState`, а просто в момент аттача передавать бинди объект во вью, и всё. Для этого достаточно заоверрайдить метод `attachView`, и в нём передавать во вью свой объект.

**terrakok** 27.12.16 в 16:39

вы называете "костылем" особенность реализации. Это как назвать "костылем" аккумуляторы у Теслы, исправляющие баг с невозможнс ездить на бензине.

Только полноправные пользователи могут оставлять комментарии. Войдите, пожалуйста.