MVP (Model-View-Presenter)

"MVP is a *user interface architectural pattern* engineered to facilitate automated unit testing and improve the *separation of concerns in presentation logic*" [1]. It makes code understandable, decoupled, testable and maintainable (easy to refactor). "Android bundles the UI and UI logic into the Activity class, which necessitates Instrumentation to test the Activity. Since Instrumentation is introduced, it is much more difficult (or impossible) to properly unit test your UI logic when the dependencies in the code cannot be mocked" [2]. In MVP, application is divided into at least 3 layers (Figure 1) and makes them independently testable from each other [3]. "With MVP we are able to take most of logic out from the activities so that we can test it without using instrumentation tests" [3]. Flow goes like **User-> View-> Presenter-> Model**

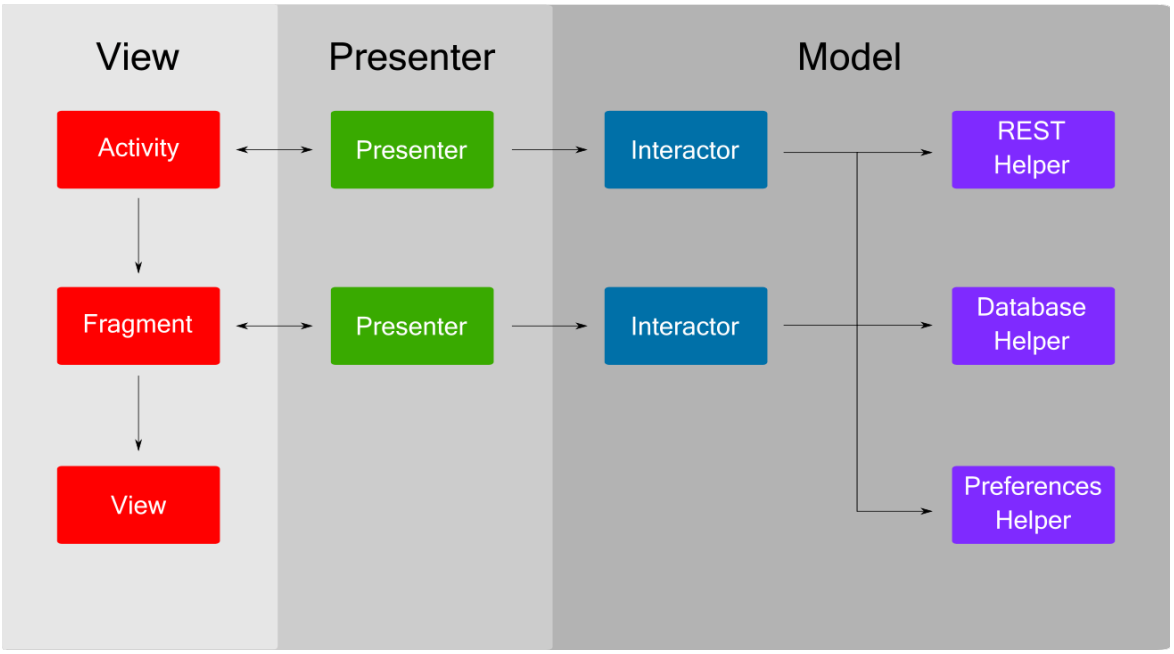

Figure 1: MVP Android Diagram [4]

"From our point of view *MVP is not a variant or enhancement of MVC* because that would mean that the Controller gets replaced by the Presenter. In our opinion **MVP wraps around MVC**. Take a look at your MVC powered app. The Presenter sits in the middle between controller and model like this:" [5]

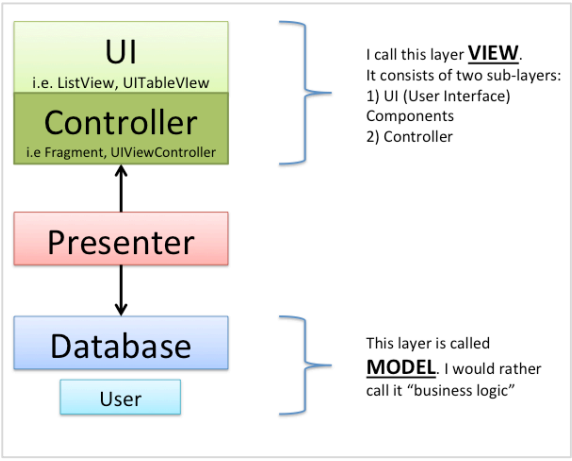

Figure 2: MVP difference from MVC [5]

"In our opinion the *Presenter does not replace the Controller*. Rather the Presenter coordinates or supervises the View which the Controller is part of. The Controller is the component that handles the click events and calls the corresponding Presenter methods. The Controller is the responsible component to control animations like hiding ProgressBar and displaying ListView instead. The Controller is listening for scroll events on the ListView i.e. to do some parallax item animations or scroll the toolbar in and out while scrolling the ListView. So all that UI related stuff still gets controlled by a Controller and not by a Presenter (i.e. *Presenter should not be an OnClickListener*). The Presenter is responsible to coordinate the overall state of the view layer (composed of UI widgets and Controller). So it's the job of the Presenter to tell the view layer that the loading animation should be displayed now or that the ListView should be displayed because the data is ready to be displayed" [5].

"When control goes from View to Presenter and then from Presenter to Model it is just a direct flow, it is easy to write code like this. You get an easy *User -> View -> Presenter -> Model -> Data* sequence. But when control goes like this: *User -> View -> Presenter -> View -> Presenter -> Model -> Data*, it just violates KISS principle. **Don't play ping - pong between your view and presenter**" [6].

Model

It is **Data of UI** [2] [7]. "The model is the data that will be displayed in the view. Please note that the word 'Model' is misleading. *It should rather be business logic that retrieves or manipulates a Model*" [5]. "**It is an interface responsible for managing data**. Model's responsibilities include using APIs, caching data, managing databases and so on. The model can also be an interface that communicates with other modules in charge of these responsibilities... *If you are using the Clean architecture, the Model could be an Interactor*" [8]. [6] [3].

Model knows nothing about other layers. "It is best to set up the Model Layer in such a way that the Presenter does not know whether it is getting the data from the disk, memory, or network" [9].

Why have different Entity (model) for each layer? We might get UserEntity which has name, surname from API but when displaying to UI, we may need to show full name and ranking etc. which does not come from API. Entity in Presentation must have such additional fields. Transformation is done asynchronously in Presenter [5].

Presenter

"Presenter is a layer that provides View with data from Model. Presenter also handles background tasks" [6]. It is **Logic of UI** [2]. "The Presenter is the 'middle-man' (played by the controller in MVC) and has references to both, view and model" [5]. "**It retrieves data from the model and returns it *formatted* to the view** [10]. But unlike the typical **MVC**, it also decides what happens when you interact with the view" [3]. "*Decides the actions to take when input events are received*" [10].

No Android code here [8], it will make platform independent and no instrumentation test is required. "As far as tests are concerned, most of the code that you absolutely need to test will be in the Presenter. What's great is that all this code doesn't need Android to run, as it just has a reference to the View interface, and not its Android-specific implementation. This

means that you can just mock the View interface, and write pure JUnit tests that make sure the right methods are called on that mock to test the integrity of your business logic" [10].

"*Lifecycle events* methods on the Presenter are simple and don't have to map the (overly complicated) Android lifecycle ones. You don't have to implement any of them, but you can implement as many as you want if the Presenter needs to take specific actions" [10] [8]. "Presenters do not need lifecycle callbacks from Activity, Fragment. It just needs to know if View is attached or not" [11].

"*Android Services* are a fundamental part of android app development. Services are clearly business logic. Therefore it's obvious that the Presenter is responsible to communicate with the Service" [12].

"**Don't retain presenter...** I think that presenter is not something we should persist, it is not a data class, to be clear... the presenter survives to orientation changes, but when Android kills the process and destroys the Activity, the latter will be recreated together with a new presenter. For this reason, this solution solves only half of the problem". You should cache in Model layer [8] [13]. But this way you won't save ongoing network requests. Presenter is only for UI logic, not data logic (cache).

View

Display of UI [2]. "View is a layer that *displays data* and *reacts to user actions*" [6] [5]. "On Android, this could be an Activity, a Fragment, an android.view.View or a Dialog" [6] [7]. "*The View doesn't know anything about the business logic or how to get the data* it has to show to the user... It will not contain any functional or business logic like access to database or web server for example" [14]. "View layer with MVP becomes so simple, so it does not even need to have callbacks when requesting for data" [6]. **View never calls Model directly**.

Responsibilities of View are as follows: informing presenter of relevant lifecycle events (attach, detach), informing presenter of user input events, laying out view and binding data to them, animations, navigating to other screens [10].

"The View is only doing what the Presenter tells the View to do. This variant of Model-View-Presenter is called **Passive View**. The view should be as dumb as possible. Let the Presenter control the view in an abstract way. For instance: Presenter invokes view.showLoading() but Presenter should not control view specific things like animations. So Presenter should not invoke methods like view.startAnimation() etc. By implementing MVP Passive View it's much easier to handle concurrency and multithreading. (returning data from Async calls)" [5]. Make View Passive, "if you have a username/password form and a "submit" button, you don't write the validation logic inside the view but inside the presenter. Your view should just collect the username and password and send them to the presenter" [8].

"The MVP pattern can also be implemented such that the View knows of the model. The view responds to state changes in the model for simple UI updates, while the presenter handles more complex UI logic. This more complex pattern is sometimes referred to as **Supervising Controller**. In Android, this can be accomplished by the Model using Java's Observable class and the View implementing the Observer interface; when something changes in the Model, it can call the Observable's notifyObservers method. It can also be implemented with Android's Handler class; when something changes in the Model, it can send a message to a handler that the View injects into it" [2].

"*The view should concern only about any necessary request parameters to restore the state*" [8]. It shouldn't save presenter or data itself.

"The methods to update the View (in View Interface) should be simple and targeted on a single element. This is better than having a single setMessage(Message m) method that will update everything, because formatting what should be displayed should be the responsibility of the Presenter. For example, you might want to start displaying 'You' instead of the user name if the current user is the author of the message, and this is part of your business logic" [10].

"Why do I need to define interfaces for the View ?" [5]

1. "*Since it's an interface you can change the view implementation*. You can simple move your code from something that extends Activity to something that extends Fragment." [5]
2. "Modularity: You can move the whole business logic, Presenter and View Interface in a standalone library project. Then you can use this library with the containing Presenter in various apps." [5]
3. "*You can easily write unit tests since you can mock views by implement the view interface*. One could also introduce a java interface for the Presenter to make unit testing by using mock Presenter objects even more easy." [5]
4. "Another very nice side effect of defining an interface for the view is that you don't get tempted to call methods of the activity / fragment directly from Presenter. You get a clear separation because while implementing the Presenter the only methods you see in your IDE's auto completion are those methods of the view interface. From our personal experiences we can say that this is very useful especially if you work in a team." [5]

Solutions for **Orientation Change** problem:

- Using static Presenter to retain Presenter
- Using Singleton PresenterManager class and retaining Presenters. Presenters store loaded models and returns them after orientation change [10] [6]
- Using LoaderManager to retain Presenter. This way ongoing network requests do not get lost [15]
- Retain Presenter until View totally gets destroyed
- Caching API response Observables in LRUCache [9]
- Storing API request Observables in somewhere (not to make requests twice)
- Caching API response data in cache, database, file...
- Request parameters for Presenter are saved in View (Activity...) [6]
- Save View's state (loading, showing error...) inside View (Activity...) itself

Code Repositories

- https://github.com/remind101/android-arch-sample *(Uses PresenterManager for retaining Presenter)*
- http://konmik.com/post/introduction_to_model_view_presenter_on_android/ *(Keeps loaded models in Presenter and returns it after orientation change. Presenter request parameters are stored in onSaveInstanceState() of Activity etc.)*
- https://github.com/antoniolg/androidmvp
- https://github.com/michal-luszczuk/tomorrow-mvp *(Uses LoaderManager to retain Presenter)*
- https://github.com/czyrux/MvpLoaderSample *(Uses Loader to retain Presenter)*
- https://github.com/macoscope/RoomBookerMVP
- https://github.com/teegarcs/RetroRx *(Caches API response Observables in LRUCache)*
- https://github.com/grandstaish/hello-mvp *(Retains Presenter until View gets destroyed once and for all)*
- https://github.com/MindorksOpenSource/android-mvp-architecture

Bonus

1. Configuration Change: [6]

	Case 1: Configuration Change, Activity restart	Case 2: An Activity Restart	Case 3: A Process Restart
Dialog	reset	reset	reset
Activity, View, Fragment	save/restore	save/restore	save/restore
Fragment with setRetainInstance(true))	no change	save/restore	save/restore
Static variables and threads	no change	no change	reset

"**Case 1:** A configuration change normally happens when a user flips the screen, changes language settings, attaches an external monitor, etc.
Case 2: An Activity restart happens when a user has set 'Don't keep activities' checkbox in Developer's settings and another activity becomes topmost.
Case 3: A process restart happens if there is not enough memory and the application is in the background." [6]

Simpler version: [6]

	A Configuration Change, An Activity Restart	A Process Restart
Activity, View, Fragment, DialogFragment	save/restore	save/restore
Static variables and threads	no change	reset

"Now it looks much better. We only need to write two pieces of code to completely restore an application in any possible case:

- save/restore for Activity, View, Fragment, DialogFragment
- restart background requests in case of a process restart" [6]

2. Presenter save/restore options [15]:

		Configuration change		State saved/restored on proces kill/recreation	Independent usage		Other comments
		State saved/retained	Long running tasks retained		with Activities only	with Fragments only	
1	Save state in bundle	YES saved/restored	NO	YES	YES	YES	NONE
2	Fragment with setRetainInstance	YES retained	YES	Yes, If additionally using onSaveInstanceState	NO	YES (but no for child fragments or back stack fragments)	NONE
3	Retain using **onRetainCustomNonConfigurationInstance**	YES retained	YES	Yes, If additionally using onSaveInstanceState	YES	NO	NONE
4	Hold presenter in some static singleton storage	YES retained	YES	Yes, If additionally using onSaveInstanceState	YES	YES	• Could be longer in memory than component it belongs to • Static is always bad
5	Retain presenter inside Loader object	YES retained	YES	Yes, If additionally using onSaveInstanceState	YES	YES	NONE

"Assumptions:

- Presenters should live in the memory only as long as the components they belong to
- The Presenter should be retained on configuration changes – i.e. if we want to retain long running tasks (even if an activity is being recreated in the same moment)
- The Presenter should be destroyed if a related component like activity or fragment is destroyed, either because of memory running out problem or just because of user action – we don't want to needlessly waste memory and store presenter in memory if related activity/fragment is currently not there
- The Presenter should be recreated after process recreation (if process of our app was killed because of low memory issues) – we must ensure reliability and possibility to restore presenter state later even if process is destroyed" [15]

References

[1] "Model-VIew-Presenter," [Online]. Available: https://en.wikipedia.org/wiki/Model%E2%80%93view%E2%80%93presenter. [Accessed 16 06 2017].

[2] J. Angelini, "An Mvp Pattern for Android," 10 04 2015. [Online]. Available: https://magenic.com/thinking/an-mvp-pattern-for-android. [Accessed 15 06 2017].

[3] A. Leiva, "MVP for Android: how to organize the presentation layer," 15 04 2014. [Online]. Available: https://antonioleiva.com/mvp-android/. [Accessed 14 06 2017].

[4] C. Ripple, "Applying MVP in Android," 26 05 2016. [Online]. Available: http://www.goxuni.com/673883-applying-mvp-in-android/. [Accessed 16 06 2017].

[5] H. Dorfmann, "Model-View-Presenter," 25 05 2015. [Online]. Available: http://hannesdorfmann.com/mosby/mvp/. [Accessed 14 06 2017].

[6] K. Mikheev, "Introduction to Model View Presenter on Android," 23 03 2015. [Online]. Available: http://konmik.com/post/introduction_to_model_view_presenter_on_android/. [Accessed 14 06 2017].

[7] P. Ziomacki, "Model-View-Presenter Architecture in Android Applications," 14 01 2016. [Online]. Available: http://macoscope.com/blog/model-view-presenter-architecture-in-android-applications/. [Accessed 15 06 2017].

[8] F. Cervone, "Model-View-Presenter: Android guidelines," 27 02 2017. [Online]. Available: https://medium.com/@cervonefrancesco/model-view-presenter-android-guidelines-94970b430ddf. [Accessed 15 06 2017].

[9] C. Teegarden, "A MVP Approach to Lifecycle Safe Requests with Retrofit 2.0 and RxJava," 11 02 2016. [Online]. Available: https://www.captechconsulting.com/blogs/a-mvp-approach-to-lifecycle-safe-requests-with-retrofit-20-and-rxjava. [Accessed 15 06 2017].

[10] N. Barraille, "Android Code That Scales, With MVP," remind.com, 12 2 2015. [Online]. Available: http://engineering.remind.com/android-code-that-scales/. [Accessed 14 06 2017].

[11] H. Dorfmann, "Presenters don't need lifecycle," 24 04 2016. [Online]. Available: http://hannesdorfmann.com/android/presenters-dont-need-lifecycle. [Accessed 14 06 2017].

[12] H. Dorfmann, "STINSON'S PLAYBOOK FOR MOSBY," 09 05 2015. [Online]. Available: http://hannesdorfmann.com/android/mosby-playbook. [Accessed 14 06 2017].

[13] M. Nakhimovich, "Presenters are not for persisting," 22 01 2017. [Online]. Available: https://hackernoon.com/presenters-are-not-for-persisting-f537a2cc7962. [Accessed 15 06 2017].

[14] D. Guerrero, "A BRIEF INTRODUCTION TO A CLEANER ANDROID ARCHITECTURE: THE MVP PATTERN," 13 10 2015. [Online]. Available: https://davidguerrerodiaz.wordpress.com/2015/10/13/a-brief-introduction-to-a-cleaner-android-architecture-the-mvp-pattern/. [Accessed 15 06 2017].

[15] M. Łuszczuk, "MVP for Android," 19 04 2016. [Online]. Available: http://blog.propaneapps.com/android/mvp-for-android/. [Accessed 15 06 2017].

Jemshit Iskenderov

17.06.2017