UDC 004

MODERN APPROACHES TO ANDROID APP ARCHITECTURE: MVVM, MVP, VIPER

YAUHENI HARAVY, SERGEY SURTO
Polotsk State University, Belarus

Choosing the right approach to application architecture is an important and responsible solution that affects all subsequent application development; it needs to keep code clean and organized. The article describes some of Android app architectures.

During Google I/O, Google introduced architecture components which include LiveData and ViewModel which facilitate developing Android app using MVVM (Model-View-ViewModel) pattern. This article discusses how these components can serve an android app that follows MVVM. MVVM is one of the architectural patterns which enhances separation of concerns, it allows separating the user interface logic from the business (or the back-end) logic. Its target (with other MVC pattern goals) is to achieve the following principle "Keeping UI code simple and free of app logic in order to make it easier to manage".

MVVM has mainly the following layers:

– Model. Model represents the data and business logic of the app. One of the recommended implementation strategies of this layer, is to expose its data through observables to be decoupled completely from ViewModel or any other observer/consumer.

– ViewModel. ViewModel interacts with model and also prepares observable(s) that can be observed by a View. ViewModel can optionally provide hooks for the view to pass events to the model. One of the important implementation strategies of this layer is to decouple it from the View, i.e, ViewModel should not be aware of the view which it is interacting with.

– View. Finally, the view role in this pattern is to observe (or subscribe to) a ViewModel observable to get data in order to update UI elements accordingly [4].

LiveData. As said above, LiveData is one of the newly introduced architecture components. LiveData is an observable data holder. This allows the components in your app to be able to observe LiveData objects for changes without creating explicit and rigid dependency paths between them. This decouples completely the LiveData object producer from the LiveData object consumer. In addition to this, there is also a great benefit in LiveData, LiveData respects the lifecycle state of your app components (activities, fragments, services) and handles object life cycle management which ensures that LiveData objects do not leak.

As per Google Docs, If you are already using a library like Rx or Agera, you can continue using them instead of LiveData. But in this case, it is your responsibility to handle object allocation and de-allocation per Android components life cycle.

Since LiveData respects Android Lifecycle, this means it will not invoke its observer callback unless the LiveData host (activity or fragment) is in an active state (received onStart() but not received onStop() for example). In addition to this, LiveData will also automatically remove the observer when the its host receives onDestroy().

ViewModel is also one of the newly introduced architecture components. Architecture components provide a new class called ViewModel, which is responsible for preparing the data for the UI/View. ViewModel gives you a good base class for your MVVM ViewModel layer since ViewModel and AndroidViewModel extending classes are automatically having their holding data retained during configuration changes. This means that after configuration changes this ViewModel holded data is immediately available to the next activity or fragment instance [2].

MVP (Model View Presenter) pattern is a derivative from the well-known MVC (Model View Controller), which for a while now is gaining importance in the development of Android applications. The MVP pattern allows separating the presentation layer from the logic, so that everything about how the interface works is separated from how we represent it on screen. Ideally the MVP pattern would achieve the same logic and might have completely different and interchangeable views. The first thing to clarify is that MVP is not an architectural pattern; it's only responsible for the presentation layer. In any case it is always better to use it for your architecture than not using it at all [5].

In Android we have a problem arising from the fact that Android activities are closely coupled to both interface and data access mechanisms. We can find extreme examples such as CursorAdapter, which mix adapters, which are part of the view together with cursors, something that should be relegated to the depths of data access layer. For an application to be easily extensible and maintainable we need to define well separated layers. What do we do tomorrow if, instead of retrieving the same data from a database, we need to do it from a web service? We would have to redo our entire view. MVP makes views independent from our data source. We divide the application into at least three different layers, which let us test them independently. With MVP we are able to take most of logic out from the activities so that we can test it without using instrumentation tests.

There are many variations of MVP and everyone can adjust the pattern idea to their needs and the way they feel more comfortable. The pattern varies depending basically on the amount of responsibilities that we delegate to the presenter. Is the view responsible to enable or disable a progress bar, or should it be done by the presenter? And who decides which actions should be shown in the Action Bar? That's where the tough decisions begin.

The presenter is responsible to act as the middleman between view and model. It retrieves data from the model and returns it formatted to the view. But unlike the typical MVC, it also decides what happens when you interact with the view.

The view, usually implemented by an Activity (it may be a Fragment, a View… depending on how the app is structured), will contain a reference to the presenter. The presenter will be ideally provided by a dependency injector such as Dagger, but in case you don't use something like this, it will be responsible for creating the presenter object. The only thing that the view will do is calling a method from the presenter every time there is an interface action (a button click for example).

In an application with a good layered architecture, this model would only be the gateway to the domain layer or business logic. If we were using the Uncle Bob clean architecture, the model would probably be an interactor that implements a use case. For now, it is enough to see it as the provider of the data we want to display in the view.

VIPER is a clean architecture mainly used in iOS app development but now we can implement the same in Android environment. It helps to keep the code clean and organized, avoiding the Massive-View-Controller situation. VIPER stands for View Interactor Presenter Entity Router, which is a class that has a well-defined responsibility, following the Single Responsibility Principle [1].

MVVM makes a lot of sense if you use it alongside data binding. But as projects grow, the presenter can become a huge class with a lot of methods, making it hard to maintain and understand. This happens so because too many responsibilities are to be taken care of by presenters like UI Events, UI logic, business logic, networking and database queries. This type of working is a direct violation of Principle of Single Responsibility, something that VIPER can fix. To overcome this issue in big Android apps, MVP+ and VIPE were used in Android apps basically moving some of the responsibilities from the Presenter to the Interactor.

The presenter handles UI events and prepares for data that comes from the Interaction to be displayed on View. The business logic and fetching of data from DBs or APIs are then done by the Interactor. Interfaces for linking the modules are used. In that way, they can't access methods other than the ones declared on the interface. This allocates a single responsibility of each module by clearly defining the structure. The developer can work with a free mind and will not commit mistakes like putting the business logic in a wrong place.

In fact, the modules are created and linked together on startup. When the Activity is created, it initializes the Presenter, passing itself as the View on the constructor. Then Interactor is initialized by a presenter. UIViewController or Storyboard handles this in an iOS project and later they club all the modules together. But on Android we don't create the Activities ourselves: we have to use Intents, and we don't have access to the newly created Activity from the previous one. This helps to prevent memory leaks, but it can be a problem if you just want to pass data to the new module. The Presenter can't be put on the Intent's extras because it would need to be Parcelable or Serializable.

VIPE had almost resolved all the MVP's problems, splitting the responsibilities of the Presenter with the Interactor. However, the View isn't as passive as the iOS VIPER's View because an additional task of regular view and routing to other modules has to be performed on Android.

Here are the differences between VIPE and VIPER. Now we move the view routing logic to the Router. It only needs an instance of the Activity so we call it startActivity method. It is still not as powerful as in iOS apps but still the single responsibility feature can be met.

After working on MVP and then doing a project using VIPER, I can surely say that the architecture does work on Android and it's worth it. The classes become smaller and more maintainable. It also guides the development process, because the architecture makes it clear where the code should be written. I am planning to use VIPER on my new projects, so that I could have better maintainability and clearer code. Of course, this is an evolving adaptation, so nothing here is carved in stone.

REFERENCES

1 Android VIPER на реактивной тяге [Электронный ресурс]. – Режим доступа: https://habrahabr.ru/company/rambler-co/blog/277003/. – Дата доступа: 12.02.2018.
2 Guide to App Architecture [Electronic resource]. – Mode of access: https://developer.android.com/topic/libraries/architecture/guide.html. – Date of access: 10.02.2018.
3 ViewModel [Electronic resource] – Mode of access: https://developer.android.com/topic/libraries/architecture/viewmodel.html. – Date of access: 10.02.2018.
4 MVVM architecture, ViewModel and LiveData [Electronic resource]. – Mode of access: https://proandroiddev.com/mvvm-architecture-viewmodel-and-livedata. – Date of access: 10.02.2018.
5 MVP for Android: how to organize the presentation layer [Electronic resource]. – Mode of access: https://antonioleiva.com/mvp-android/. – Date of access: 12.02.2018.