**Android Architecture Patterns Part 1: Model-View-Controller**

[Florina Muntenescu](https://medium.com/@florina.muntenescu?source=post_page-----3baecef5f2b6----------------------)

[Florina Muntenescu](https://medium.com/@florina.muntenescu?source=post_page-----3baecef5f2b6----------------------)

[Nov 2, 2016](https://medium.com/upday-devs/android-architecture-patterns-part-1-model-view-controller-3baecef5f2b6?source=post_page-----3baecef5f2b6----------------------) · 6 min read

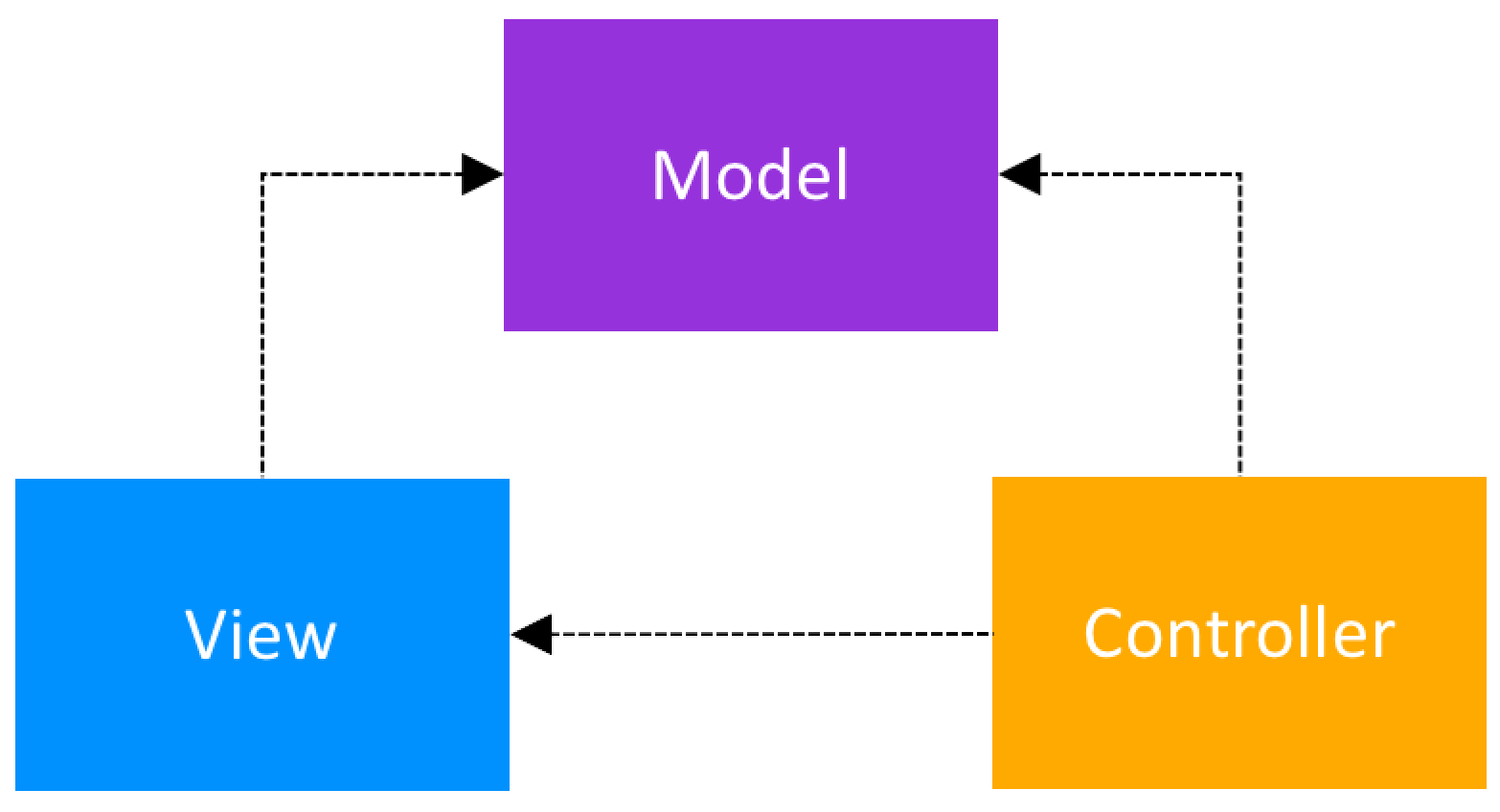
A year ago, when the majority of the current Android team started working at upday, the application was far from being the robust, stable app that we wanted it to be. We tried to understand why our code was in such bad shape and we found two main culprits: continuous changing of the UI and the lack of an architecture that supported the flexibility that we needed. The app was already at its fourth redesign in six months. The design pattern chosen seemed to be Model-View-Controller but was then already a “mutant”, far from how it should be.

Let’s discover together what the Model-View-Controller pattern is; how it has been applied in Android over the years; how it should be applied so it can maximise testability; and some of its advantages and disadvantages.

**The Model-View-Controller Pattern**

In a world where the user interface logic tends to change more often than the business logic, the desktop and Web developers needed a way of separating user interface functionality. The MVC pattern was their solution.

* **Model** — the data layer, responsible for managing the business logic and handling network or database API.
* **View** — the UI layer — a visualisation of the data from the Model.
* **Controller** — the logic layer, gets notified of the user’s behavior and updates the Model as needed.

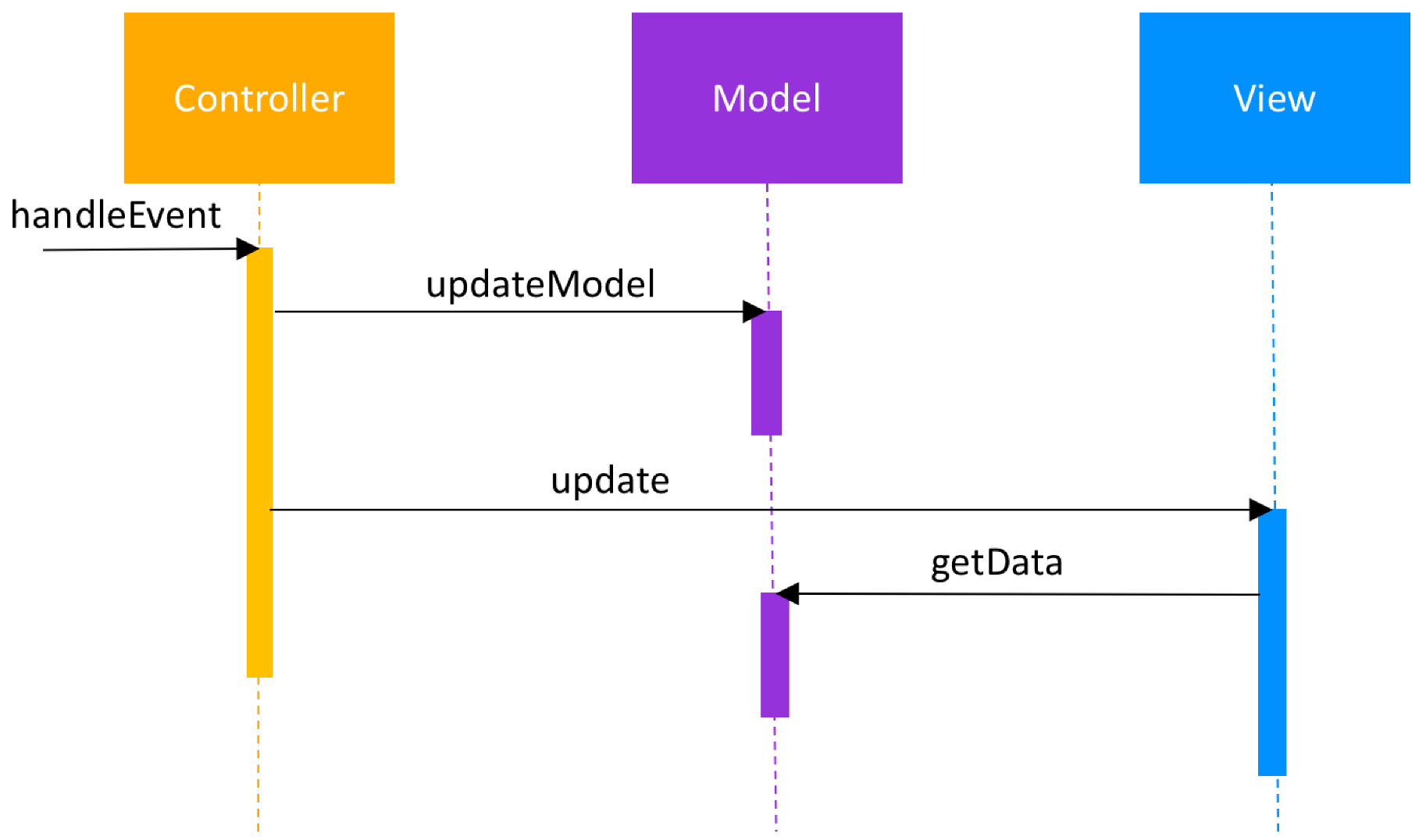


Model-View-Controller class structure

So, this means that both the Controller and the View depend on the Model: the Controller to update the data, the View to get the data. But, most important for the desktop and Web devs at that time: the Model was separated and could be tested independently of the UI. Several variants of MVC appeared. The best-known ones are related to whether the Model is passive or is actively notifying that it has changed. Here are more details:

**Passive Model**

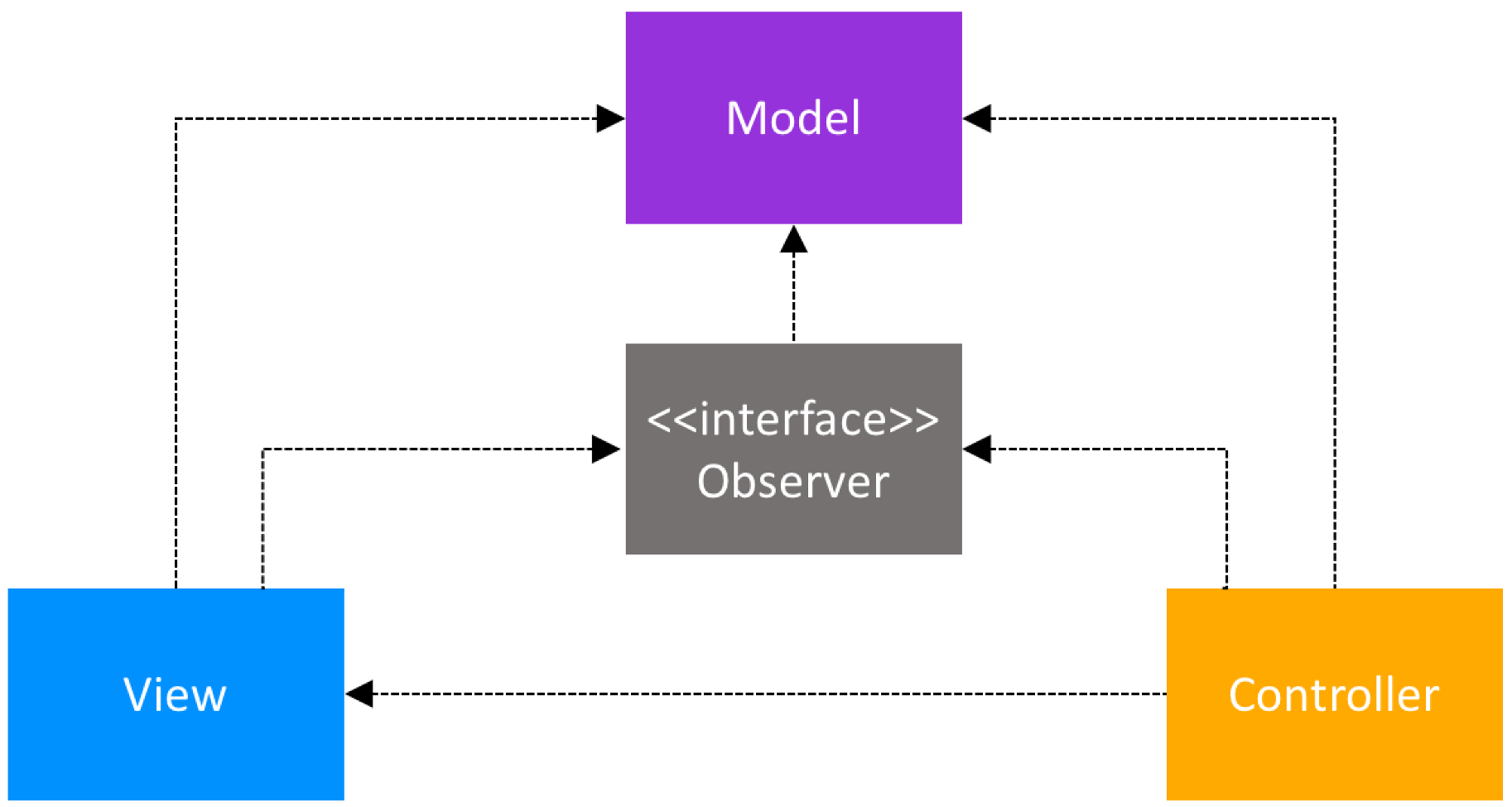
In the Passive Model version, the Controller is the only class that manipulates the Model. Based on the user’s actions, the Controller has to modify the Model. After the Model has been updated, the Controller will notify the View that it also needs to update. At that point, the View will request the data from the Model.



Model-View-Controller — passive Model — behavior

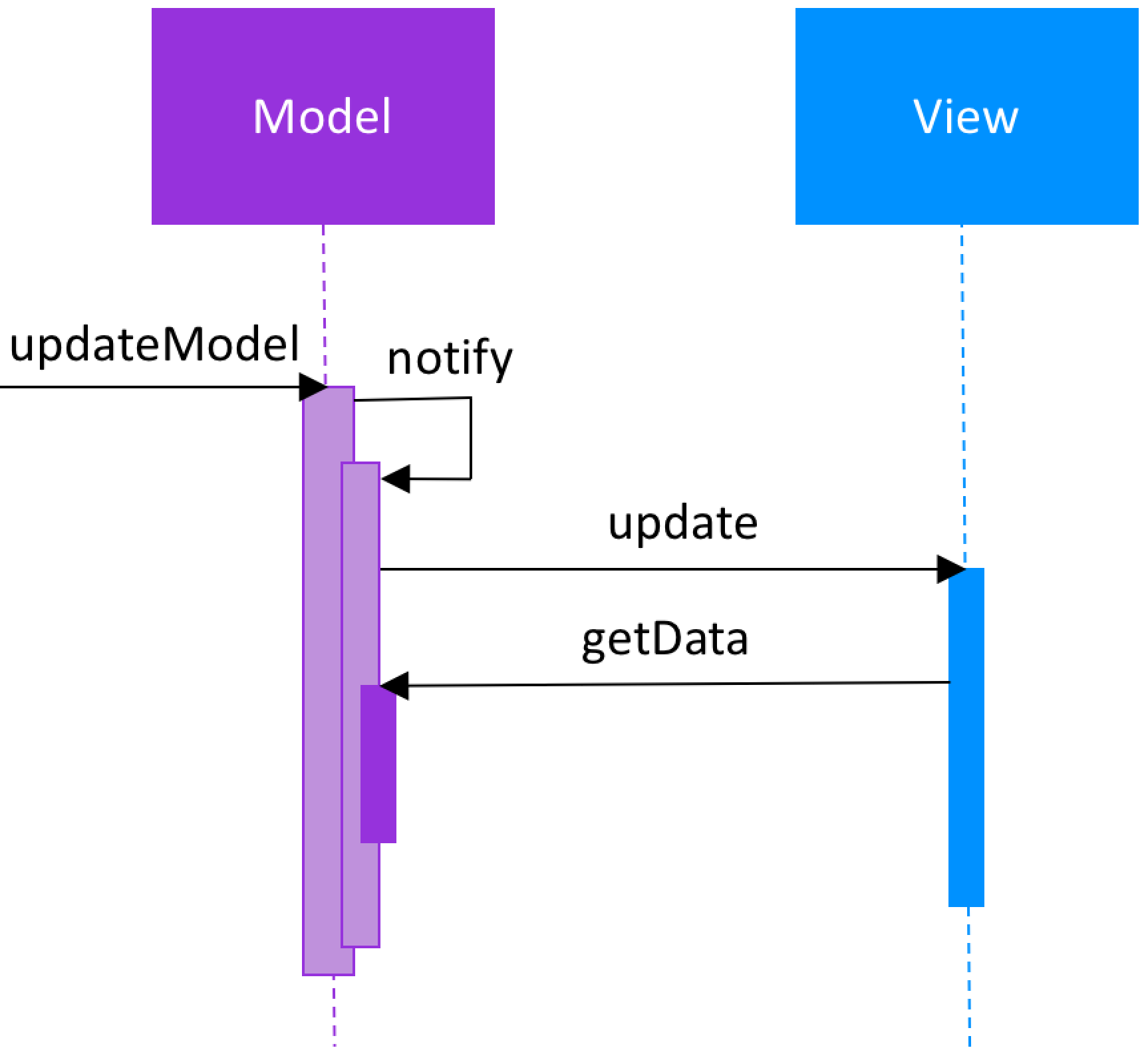
**Active Model**

For the cases when the Controller is not the only class that modifies the Model, the Model needs a way to notify the View, and other classes, about updates. This is achieved with the help of the Observer pattern. The Model contains a collection of observers that are interested in updates. The View implements the observer interface and registers as an observer to the Model.



Model-View-Controller — active Model — class structure

Every time the Model updates, it will also iterate through the collection of observers and call the update method. The implementation of this method in the View will then trigger the request of the latest data from the Model.



Model-View-Controller — active Model — behavior

**Model-View-Controller in Android**

In around 2011, when Android started to become more and more popular, architecture questions naturally appeared. Since MVC was one of the most popular UI patterns at that time, developers tried to apply it to Android too.

If you search on StackOverflow for questions like “How to apply MVC in Android”, one of the most popular answers stated that in Android, an Activity is both the View and the Controller. Looking back, this sounds almost crazy! But, at that point, the main emphasis was on making the Model testable and usually the implementation choice for the View and the Controller was dependent on the platform.

**How Should MVC Be Applied in Android**

Nowadays, the question of how to apply the MVC patterns has an answer that is easier to find. The Activities, Fragments and Views should be the Views in the MVC world. The Controllers should be separate classes that don’t extend or use any Android class, and same for the Models.

One problem arises when connecting the Controller to the View, since the Controller needs to tell the View to update. In the passive Model MVC architecture, the Controller needs to hold a reference to the View. The easiest way of doing this, while focusing on testing, is to have a BaseView interface, that the Activity/Fragment/View would extend. So, the Controller would have a reference to the BaseView.

**Advantages**

The Model-View-Controller pattern highly supports the separation of concerns. This advantage not only increases the testability of the code but it also makes it easier to extend, allowing a fairly easy implementation of new features.

The Model classes don’t have any reference to Android classes and are therefore straightforward to unit test. The Controller doesn’t extend or implement any Android classes and should have a reference to an interface class of the View. In this way, unit testing of the Controller is also possible.

If the Views respect the **single responsibility principle** then their role is just to update the Controller for every user event and just display data from the Model, without implementing any business logic. In this case, UI tests should be enough to cover the functionalities of the View.

**Disadvantages**

**The View Depends On The Controller And On The Model**

The dependence of the View on the Model starts being a downside in complex Views. In order to minimize the logic in the View, the Model should be able to provide testable methods for every element that gets to be displayed. In an active Model implementation, this exponentially increases the number of classes and methods, given that Observers for every type of data would be required.

Given that the View depends on both the Controller and the Model, changes in the UI logic might require updates in several classes, decreasing the flexibility of the pattern.

**Who Handles The UI Logic?**

According to the MVC pattern, the Controller updates the Model and the View gets the data to be displayed from the Model. But who decides on how to display the data? Is it the Model or the View? Consider the following example: we have a User, with first name and last name. In the View we need to display the user name as “Lastname, Firstname” (e.g. “Doe, John”).

If the Model’s role is to just provide the “raw” data, it means that the code in the View would be:

String firstName **=** userModel**.**getFirstName**();**   
String lastName **=** userModel**.**getLastName**();** nameTextView**.**setText**(**lastName **+** ", " **+** firstName**)**

So this means that it would be the View’s responsibility of handling the UI logic. But this makes the UI logic impossible to unit test.

The other approach is to have the Model expose only the data that needs to be displayed, hiding any business logic from the View. But then, we end up with Models that handle both business and UI logic. It would be unit testable, but then the Model ends up, implicitly being dependent on the View.

String name **=** userModel**.**getDisplayName**();** nameTextView**.**setText**(**name**);**

**Conclusion**

In the early days of Android the Model-View-Controller pattern seemed to have confused a lot of developers and led to code that was difficult, if not impossible to unit test.

The dependence of the View from the Model and having logic in the View steered our code-base to a state from which it was impossible to recover without refactoring completely the app. What was the new approach in architecture and why? Find out in by reading [this blog post](https://upday.github.io/blog/model-view-presenter/).

# Android Architecture Patterns Part 2: Model-View-Presenter

[](https://medium.com/@florina.muntenescu?source=post_page-----8a6faaae14a5----------------------)

[Florina Muntenescu](https://medium.com/@florina.muntenescu?source=post_page-----8a6faaae14a5----------------------)

[Nov 2, 2016](https://medium.com/upday-devs/android-architecture-patterns-part-2-model-view-presenter-8a6faaae14a5?source=post_page-----8a6faaae14a5----------------------) · 6 min read

It’s about time we developers start thinking about how we can apply good architecture patterns in our Android apps. To help with this, Google offers [Android Architecture Blueprints](https://github.com/googlesamples/android-architecture), where [Erik Hellman](https://github.com/erikhellman) and I worked together on the [MVP & RxJava](https://github.com/googlesamples/android-architecture/tree/todo-mvp-rxjava/) sample. Let’s have a look at how we applied it and the pros and cons of this approach.

## [googlesamples/android-architecture](https://github.com/googlesamples/android-architecture/tree/todo-mvp-rxjava/?source=post_page-----8a6faaae14a5----------------------)

### [android-architecture - A collection of samples to discuss and showcase different architectural tools and patterns for…](https://github.com/googlesamples/android-architecture/tree/todo-mvp-rxjava/?source=post_page-----8a6faaae14a5----------------------)

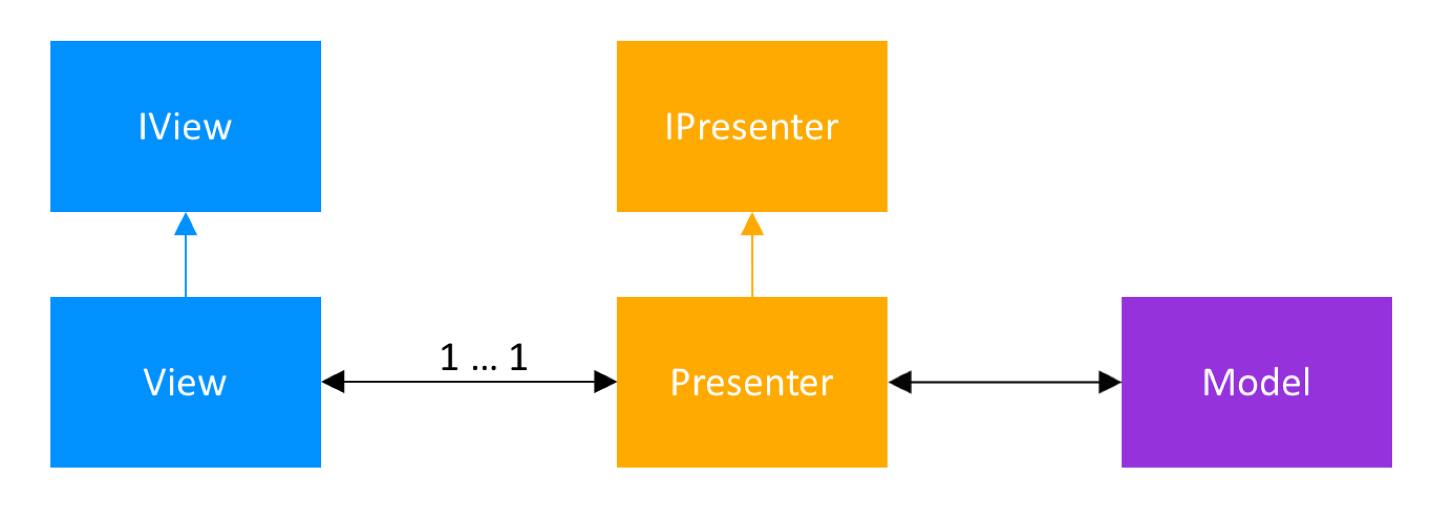
#### [github.com](https://github.com/googlesamples/android-architecture/tree/todo-mvp-rxjava/?source=post_page-----8a6faaae14a5----------------------)

# The Model-View-Presenter Pattern

Here are the roles of every component:

* **Model** — the data layer. Responsible for handling the business logic and communication with the network and database layers.
* **View** — the UI layer. Displays the data and notifies the Presenter about user actions.
* **Presenter** — retrieves the data from the Model, applies the UI logic and manages the state of the View, decides what to display and reacts to user input notifications from the View.

Since the View and the Presenter work closely together, they need to have a reference to one another. To make the Presenter unit testable with JUnit, the View is abstracted and an interface for it used. The relationship between the Presenter and its corresponding View is defined in a Contract interface class, making the code more readable and the connection between the two easier to understand.



Model-View-Presenter class structure

# The Model-View-Presenter Pattern & RxJava in Android Architecture Blueprints

The blueprint sample is a [”To Do” application](https://github.com/googlesamples/android-architecture/wiki/To-do-app-specification). It lets a user create, read, update and delete “To Do” tasks, as well as apply filters to the displayed list of tasks. RxJava is used to move off the main thread and be able to handle asynchronous operations.

# Model

The Model works with the remote and local data sources to get and save the data. This is where the business logic is handled. For example, when requesting the list of Tasks, the Model would try to retrieve them from the local data source. If it is empty, it will query the network, save the response in the local data source and then return the list.

The retrieval of tasks is done with the help of RxJava:

public Observable<List<Task>> **getTasks**(){   
 ...   
}

The Model receives as parameters in the constructor **interfaces of the local and remote data sources**, making the Model completely independent from any Android classes and thus easy to unit test with JUnit. For example, to test that getTasks requests data from the local source, we implemented the following test:

@Mock   
private TasksDataSource mTasksRemoteDataSource;   
@Mock   
private TasksDataSource mTasksLocalDataSource;   
...   
@Test   
public void **getTasks\_requestsAllTasksFromLocalDataSource**() {  
 // Given that the local data source has data available   
 setTasksAvailable(mTasksLocalDataSource, TASKS);  
 // And the remote data source does not have any data available   
 setTasksNotAvailable(mTasksRemoteDataSource);  
   
 // When tasks are requested from the tasks repository   
 TestSubscriber<List<Task>> testSubscriber =   
 new TestSubscriber<>();   
 mTasksRepository.getTasks().subscribe(testSubscriber); // Then tasks are loaded from the local data source   
 verify(mTasksLocalDataSource).getTasks();   
 testSubscriber.assertValue(TASKS);   
}

# View

The View works with the Presenter to display the data and it notifies the Presenter about the user’s actions. In MVP Activities, Fragments and custom Android views can be Views. Our choice was to use Fragments.

All Views implement the same BaseView interface that allows setting a Presenter.

public interface **BaseView**<T> {  
 void setPresenter(T presenter);   
}

The View notifies the Presenter that it is ready to be updated by calling the subscribemethod of the Presenter in onResume. The View calls presenter.unsubscribe() in onPause to tell the Presenter that it is no longer interested in being updated. If the implementation of the View is an Android custom view, then the subscribe and unsubscribe methods have to be called on onAttachedToWindow and onDetachedFromWindow. User actions, like button clicks, will trigger corresponding methods in the Presenter, this being the one that decides what should happen next.

The Views are tested with Espresso. The statistics screen, for example, needs to display the number of active and completed tasks. The test that checks that this is done correctly first puts some tasks in the TaskRepository; then launches the StatisticsActivity and checks content of the views:

@Before   
public void **setup**() {  
 // Given some tasks TasksRepository.destroyInstance();  
 TasksRepository repository = Injection.provideTasksRepository( InstrumentationRegistry.getContext()); repository.saveTask(new Task("Title1", "", false));  
 repository.saveTask(new Task("Title2", "", true)); // Lazily start the Activity from the ActivityTestRule Intent startIntent = new Intent();  
 mStatisticsActivityTestRule.launchActivity(startIntent);  
}@Test   
public void **Tasks\_ShowsNonEmptyMessage**() throws Exception {  
 // Check that the active and completed tasks text is displayed Context context = InstrumentationRegistry.getTargetContext();  
 String expectedActiveTaskText = context   
 .getString(R.string.statistics\_active\_tasks);  
 String expectedCompletedTaskText = context   
 .getString(R.string.statistics\_completed\_tasks); onView(withText(containsString(expectedActiveTaskText)))   
 .check(matches(isDisplayed()));  
 onView(withText(containsString(expectedCompletedTaskText)))   
 .check(matches(isDisplayed()));  
}

# Presenter

The Presenter and its corresponding View are created by the Activity. References to the View and to the TaskRepository - the Model - are given to the constructor of the Presenter. In the implementation of the constructor, the Presenter will call the setPresenter method of the View. This can be simplified when using a dependency injection framework that allows the injection of the Presenters in the corresponding views, reducing the coupling of the classes. The implementation of the ToDo-MVP with Dagger is covered in [another sample](https://github.com/googlesamples/android-architecture).

All Presenters implement the same BasePresenter interface.

public interface **BasePresenter** {  
 void subscribe();  
 void unsubscribe();  
}

When the subscribe method is called, the Presenter starts requesting the data from the Model, then it applies the UI logic to the received data and sets it to the View. For example, in the StatisticsPresenter, all tasks are requested from the TaskRepository - then the retrieved tasks are used to compute the number of active and completed tasks. These numbers will be used as parameters for the showStatistics(int numberOfActiveTasks, int numberOfCompletedTasks) method of the View.

A unit test to check that indeed the showStatistics method is called with the correct values is easy to implement. We are mocking the TaskRepository and the StatisticsContract.View and give the mocked objects as parameters to the constructor of a StatisticsPresenter object. The test implementation is:

@Test   
public void **loadNonEmptyTasksFromRepository\_CallViewToDisplay**() {  
 // Given an initialized StatisticsPresenter with // 1 active and 2 completed tasks setTasksAvailable(TASKS); // When loading of Tasks is requested mStatisticsPresenter.subscribe(); // Then the correct data is passed on to the view verify(mStatisticsView).showStatistics(1, 2);  
}

The role of the unsubscribe method is to clear all the subscriptions of the Presenter, thus avoiding memory leaks.

Apart from subscribe and unsubscribe, each Presenter exposes other methods, corresponding to the user actions in the View. For example, the AddEditTaskPresenter, adds methods like createTask, that would be called when the user presses the button that creates a new task. This ensures that all the user actions - and consequently all the UI logic - go through the Presenter and thereby can be unit tested.

# Disadvantages of Model-View-Presenter Pattern

The Model-View-Presenter pattern brings with it a very good separation of concerns. While this is for sure a pro, when developing a small app or a prototype, this can seem like an overhead. To decrease the number of interfaces used, some developers remove the Contract interface class, and the interface for the Presenter.

One of the pitfalls of MVP appears when moving the UI logic to the Presenter: this becomes now an all-knowing class, with thousands of lines of code. To solve this, split the code even more and remember to create classes that have only one responsibility and are unit testable.

# Conclusion

The [Model-View-Controller pattern](https://upday.github.io/blog/model-view-controller/) has two main disadvantages: firstly, the View has a reference to both the Controller and the Model; and secondly, it does not limit the handling of UI logic to a single class, this responsibility being shared between the Controller and the View or the Model. The Model-View-Presenter pattern solves both of these issues by breaking the connection that the View has with the Model and creating only **one class that handles** everything related to **the presentation of the View** — the Presenter: a single class that is easy to unit test.

What if we want an event-based architecture, where the View reacts on changes? Stay tuned for the next patterns sampled in the [Android Architecture Blueprints](https://github.com/googlesamples/android-architecture) to see how this can be implemented. Until then, read about our [Model-View-ViewModel pattern](https://upday.github.io/blog/model-view-viewmodel/) implementation in the upday app.

**Android Architecture Patterns Part 3:  
Model-View-ViewModel**

[Florina Muntenescu](https://medium.com/@florina.muntenescu?source=post_page-----e7eeee76b73b----------------------)

[Florina Muntenescu](https://medium.com/@florina.muntenescu?source=post_page-----e7eeee76b73b----------------------)

[Nov 4, 2016](https://medium.com/upday-devs/android-architecture-patterns-part-3-model-view-viewmodel-e7eeee76b73b?source=post_page-----e7eeee76b73b----------------------) · 6 min read

After four different designs in the first six months of the development of the upday app, we learned one important lesson: we need an architecture pattern that allows fast reaction to design changes! The solution we chose in the end was Model-View-ViewModel. Discover with me what MVVM is; how we are applying it at upday and what makes it so perfect for us.

**The Model-View-ViewModel Pattern**

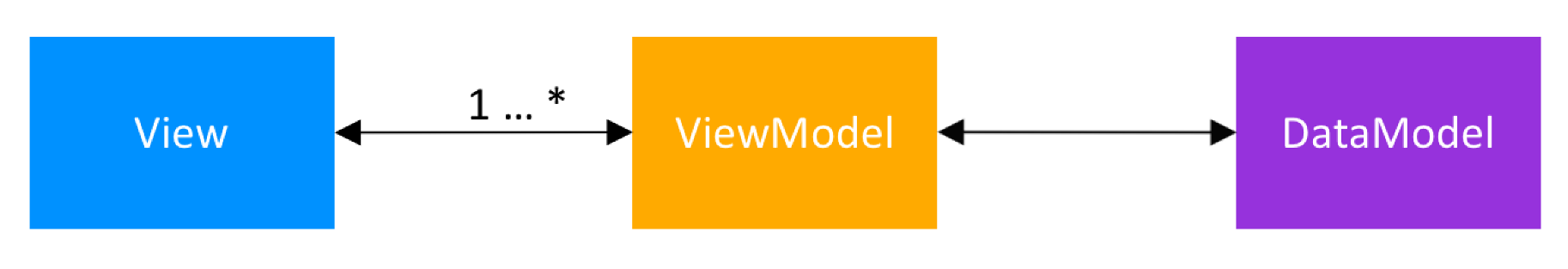
The main players in the MVVM pattern are:

* The *View* — that informs the ViewModel about the user’s actions
* The *ViewModel* — exposes streams of data relevant to the View
* The *DataModel* — abstracts the data source. The ViewModel works with the DataModel to get and save the data.

At a first glance, MVVM seems very similar to the [Model-View-Presenter pattern](https://upday.github.io/blog/model-view-presenter/), because both of them do a great job in abstracting the view’s state and behavior. The Presentation Model abstracts a View independent from a specific user-interface platform, whereas the MVVM pattern was created to simplify the **event driven** programming of user interfaces.

If the MVP pattern meant that the Presenter was telling the View directly what to display, in MVVM, **ViewModel exposes streams of events** to which the Views can bind to. Like this, the ViewModel does not need to hold a reference to the View anymore, like the Presenter is. This also means that all the interfaces that the MVP pattern requires, are now dropped.

The Views also notify the ViewModel about different actions. Thus, the MVVM pattern supports two-way data binding between the View and ViewModel and there is a many-to-one relationship between View and ViewModel. View has a reference to ViewModel but **ViewModel has no information about the View**. The consumer of the data should know about the producer, but the producer — the ViewModel — doesn’t know, and doesn’t care, who consumes the data.



Model-View-ViewModel class structure

**Model-View-ViewModel at upday**

A quick look at the Android posts on the upday blog will instantly reveal what our favorite library is: RxJava. So it’s no wonder, because RxJava is the backbone of upday’s code! The event driven part required by MVVM is done using RxJava’s Observables. Here’s how we apply MVVM in the Android app at upday, with the help of RxJava:

**DataModel**

The DataModel exposes data easily consumable through event streams — RxJava’s Observables. It composes data from multiple sources, like the network layer, database or shared preferences and exposes easily consumable data to whomever needs it. The DataModels hold the entire business logic.

Our strong emphasis on the single responsibility principle leads to creating a DataModel for every feature in the app. For example, we have an ArticleDataModel that composes its output from the API service and database layer. This DataModel handles the business logic ensuring that the latest news from the database is retrieved, by applying an age filter.

**ViewModel**

The ViewModel is a model for the View of the app: an abstraction of the View. The ViewModel retrieves the necessary data from the DataModel, applies the UI logic and then exposes relevant data for the View to consume. Similar to the DataModel, the ViewModel exposes the data via Observables.

We learned two things about the ViewModel the hard way:

* The ViewModel should expose states for the View, rather than just events. For example, if we need to display the name and the email address of a User, rather than creating two streams for this, we create a DisplayableUser object that encapsulates the two fields. The stream will emit every time the display name or the email changes. This way, we ensure that our View always displays the current state of the User.
* We should make sure that every action of the user goes through the ViewModel and that any possible logic of the View is moved in the ViewModel.

We wrote about these two topics in a blog post about [common mistakes in MVVM + RxJava](https://upday.github.io/blog/mvvm_rx_common_mistakes).

**View**

The View is the actual user interface in the app. It can be an Activity, a Fragment or any custom Android View. For Activities and Fragments, we are binding and unbinding from the event sources on onResume() and onPause().

private final CompositeSubscription mSubscription = new CompositeSubscription();  
  
@Override  
public void **onResume**() {  
 super.onResume();  
 mSubscription.add(mViewModel.getSomeData()  
 .observeOn(AndroidSchedulers.mainThread())  
 .subscribe(this::updateView,  
 this::handleError));  
}  
  
@Override  
public void **onPause**() {  
 mSubscription.clear();  
 super.onPause();  
}

If the MVVM View is a custom Android View, the binding is done in the constructor. To ensure that the subscription is not preserved, leading to [possible memory leaks](https://medium.com/square-corner-blog/android-leak-pattern-subscriptions-in-views-18f0860aa74c), the unbinding happens in onDetachedFromWindow.

private final CompositeSubscription mSubscription = new CompositeSubscription();public MyView(Context context, MyViewModel viewModel) {  
 ...  
 mSubscription.add(mViewModel.getSomeData()  
 .observeOn(AndroidSchedulers.mainThread())  
 .subscribe(this::updateView,  
 this::handleError));  
 }[@Override](http://twitter.com/Override)  
 public void **onDetachedFromWindow**() {  
 mSubscription.clear();  
 super.onDetachedFromWindow();  
 }  
}

**Testability Of The Model-View-ViewModel Classes**

One of the main reasons why we love the Model-View-ViewModel pattern is that it is so easy to test.

**DataModel**

The use of [inversion of control pattern](https://en.wikipedia.org/wiki/Inversion_of_control), heavily applied in our code, and the lack of any Android classes, facilitate the implementation of unit tests of the DataModel.

**ViewModel**

We see the Views and the unit tests as two different types of consumers of data from the ViewModel. The ViewModel is completely separated from the UI or any Android classes, therefore straightforward to unit test.

Consider the following example where the ViewModel just exposes some data from the DataModel:

public class **ViewModel** {  
 private final IDataModel mDataModel;public ViewModel(IDataModel dataModel) {  
 mDataModel = dataModel;  
 }public Observable<Data> getSomeData() {  
 return mDataModel.getSomeData();  
 }  
}

The tests for the ViewModel are easy to implement. With the help of Mockito, we are mocking the DataModel and we control the returned data for the methods used. Then, we make sure that when we subscribe to the Observable returned by getSomeData(), the expected data is emitted.

public class **ViewModelTest** {@Mock  
private IDataModel mDataModel;  
private ViewModel mViewModel;@Before  
public void setUp() throws Exception {  
 MockitoAnnotations.initMocks(this); mViewModel = new ViewModel(mDataModel);  
}@Test  
public void **testGetSomeData\_emitsCorrectData**() {  
 SomeData data = new SomeData();  
   
 Mockito.when(mDataModel.getSomeData())  
 .thenReturn(Observable.just(data));  
 TestSubscriber<SomeData> testSubscriber =   
new TestSubscriber<>(); mViewModel.getSomeData().subscribe(testSubscriber); testSubscriber.assertValue(data);  
 }  
}

If the ViewModel needs access to Android classes, we create wrappers that we call Providers. For example, for Android resources we created a IResourceProvider, that exposes methods like String getString(@StringRes final int id). The implementation of the IResourceProvider will contain a reference to the Context but, the ViewModel will only refer to an IResourceProvider injected.

As we have mentioned above, and in our [common mistakes](https://upday.github.io/blog/mvvm_rx_common_mistakes) blog post, we are creating model objects to hold the state of the data. This also allows a higher degree of testability and control of the data that is emitted by the ViewModel.

**View**

Given that the logic in the UI is minimal, the Views are easy to test with Espresso. We are also using libraries like DaggerMock and MockWebServer to improve the stability of our UI tests.

**Is MVVM The Right Solution?**

We have been using MVVM together with RxJava for almost a year now. We have seen that since the View is just a consumer of the ViewModel, it was easy to just replace different UI elements, with minimal, or sometimes zero changes in other classes.

We have also learned how important separation of concerns is and that we should split the code more, creating small Views and small ViewModels that only have specific responsibilities. The ViewModels are injected in the Views. This means that most of the times, we just add the Views in the XML UI, without doing any other changes. Therefore, when our UI requirements change again, we can easily replace some Views with new ones.

**Conclusion**

MVVM combines the advantages of separation of concerns provided by [MVP](https://upday.github.io/blog/model-view-presenter/), while leveraging the advantages of data bindings. The result is a pattern where the model drives as many of the operations as possible, minimizing the logic in the view.

After the design changes during the “infancy” of our app, we switched to MVVM in upday’s “adolescence” — a period of mistakes from which we learned a lot. Now, we can be proud of an app that has proven its resistance to another redesign. We are finally close to being able to call upday a mature app.

A simple example of the MVVM implementation can be found [here](https://github.com/florina-muntenescu/DroidconMVVM).A “Hello, World!” comparison between MVP and MVVM can be found [here](https://github.com/florina-muntenescu/MVPvsMVVM). Check out also our post on Model-View-Presenter.