MVC VS MVP VS MVVM

Common patterns in Android development

TRAINING ROADMAP: OVERVIEW

- MVC
- MVP
- MVVM
- Summary
- Best practices

This training covers major aspects of MVC & MVP.

The goal of this training is to become familiar with MVP.

Pre-requisites:

- Android Studio
- Android device or emulator

SECTION 1:
MVC

MVC

- The model, view, controller approach separates your application at a macro level into 3 sets of responsibilities.

- MVC does a great job of separating the model and view. Certainly the model can be easily tested because it's not tied to anything and the view has nothing much to test at a unit testing level. The Controller has a few problems however.

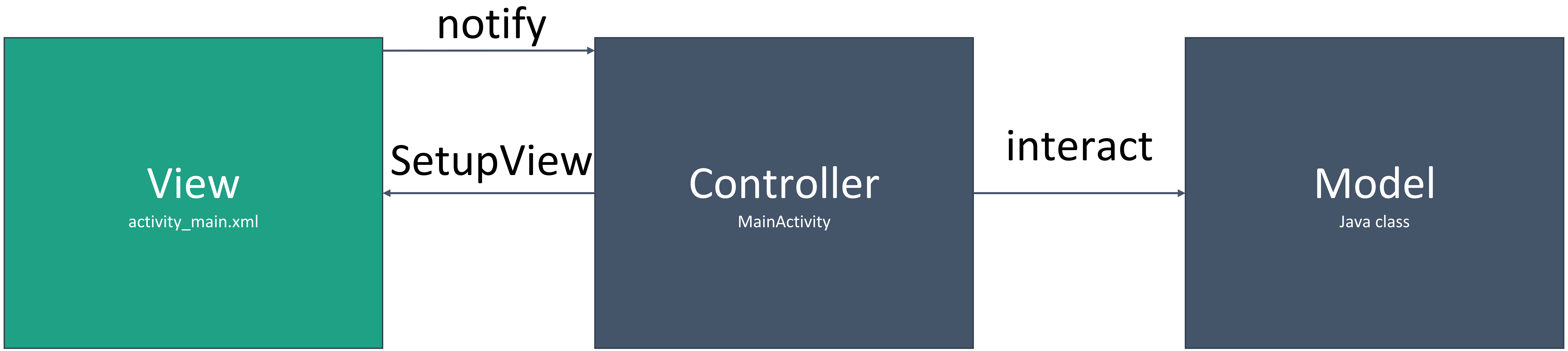
- *Testability* - The controller is tied so tightly to the Android APIs that it is difficult to unit test.

- *Modularity & Flexibility* - The controllers are tightly coupled to the views. It might as well be an extension of the view. If we change the view, we have to go back and change the controller.

- *Maintenance* - Over time, particularly in applications with anemic models, more and more code starts getting transferred into the controllers, making them bloated and brittle.

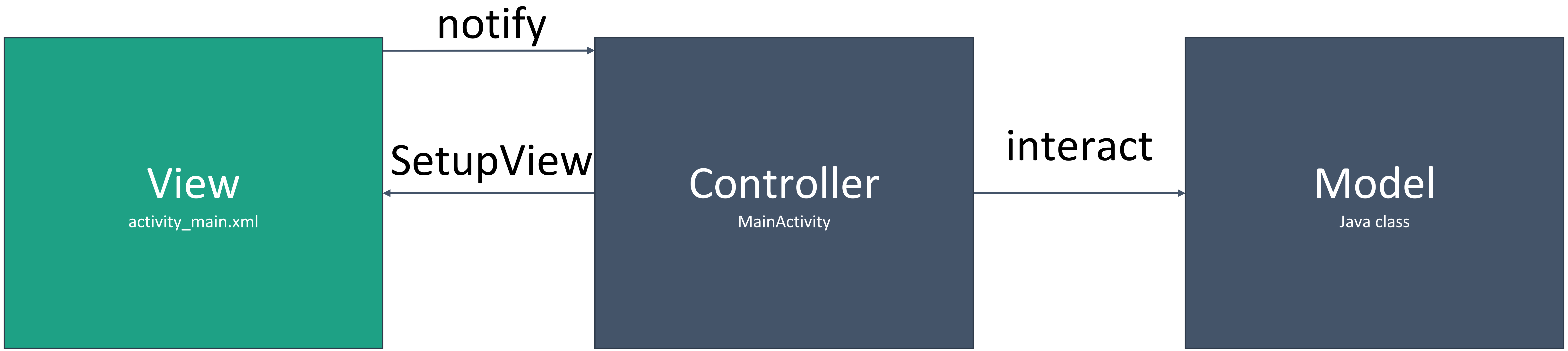
MVC: MODEL

- The model is the **Data + State + Business logic**
- It’s the brains of our application so to speak.
- It is not tied to the view or controller, and because of this, it is reusable in many contexts.



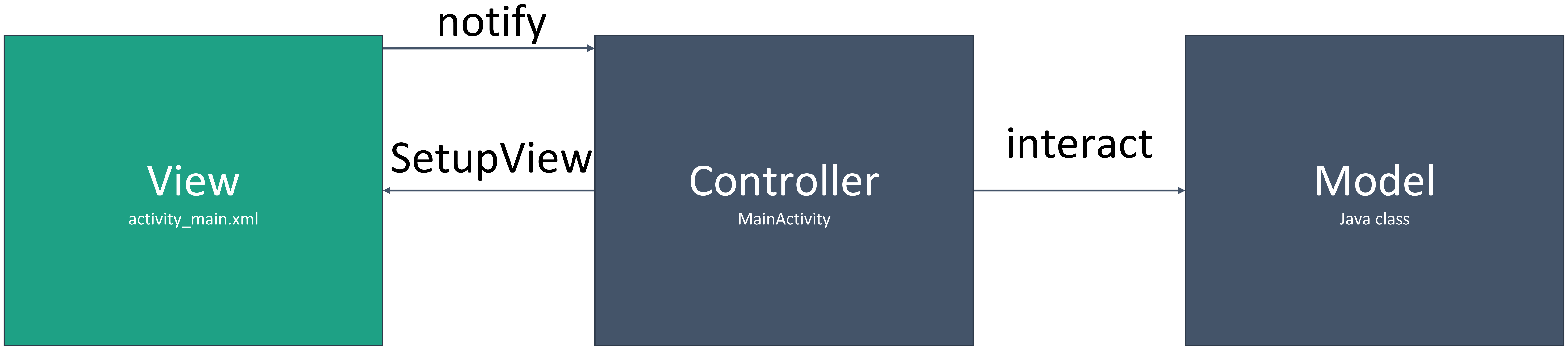
MVC: VIEW

- The view is the **Representation** of the Model.
- The view has a responsibility to render the User Interface (UI) and communicate to the controller when the user interacts with the application.
- In MVC architecture, Views are generally pretty “dumb” in that they have no knowledge of the underlying model and no understanding of state or what to do when a user interacts by clicking a button, typing a value, etc.
- The idea is that the less they know the more loosely coupled they are to the model and therefore the more flexible they are to change.



MVC: CONTROLLER

- The controller is **Glue** that ties the app together. It’s the master controller for what happens in the application. When the View tells the controller that a user clicked a button, the controller decides how to interact with the model accordingly. Based on data changing in the model, the controller may decide to update the state of the view as appropriate. In the case of an Android application, the controller is almost always represented by an Activity or Fragment.



SECTION 2:
MVP

MVP

- MVP breaks the controller up so that the natural view/activity coupling can occur without tying it to the rest of the “controller” responsibilities. More on this below, but let’s start again with a common definition of responsibilities as compared to MVC.

- This is much cleaner. We can easily unit test the presenter logic because it’s not tied to any Android specific views and APIs and that also allows us to work with any other view as long as the view implements an abstract interface.

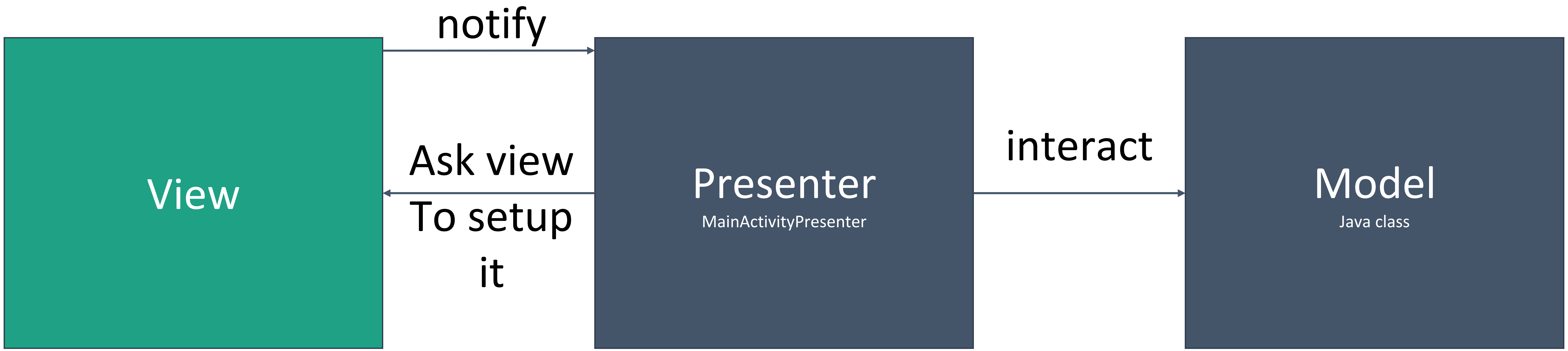
- *Maintenance* - Presenters, just like Controllers, are prone to collecting additional business logic, sprinkled in, over time. At some point, developers often find themselves with large unwieldy presenters that are difficult to break apart.

- Of course, the careful developer can help to prevent this, by diligently guarding against this temptation as the application changes over time. However, MVVM can help address this by doing less to start

MVP: MODEL

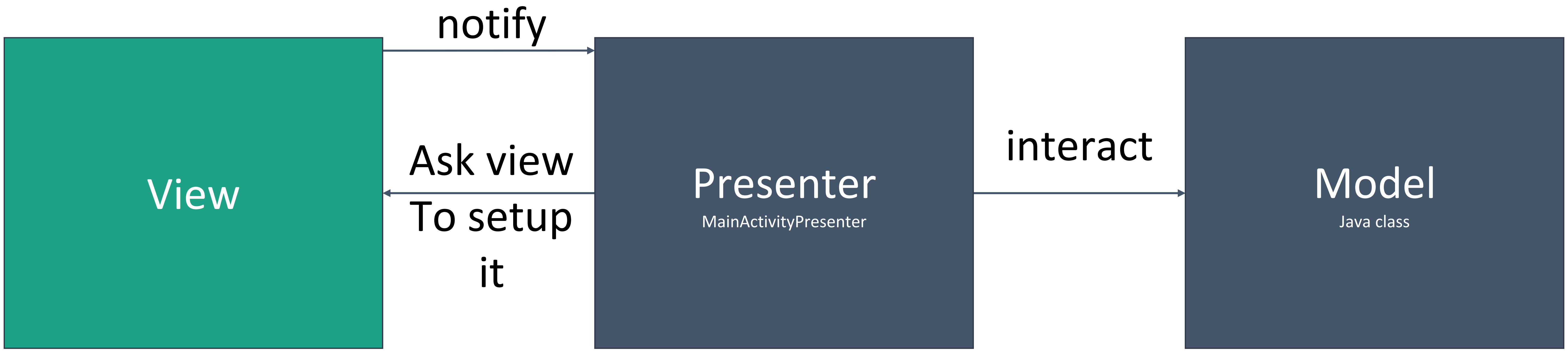
- Same as MVC / No change

- The model is the **Data + State + Business logic**. It's the brains of our application so to speak. It is not tied to the view or controller, and because of this, it is reusable in many contexts.



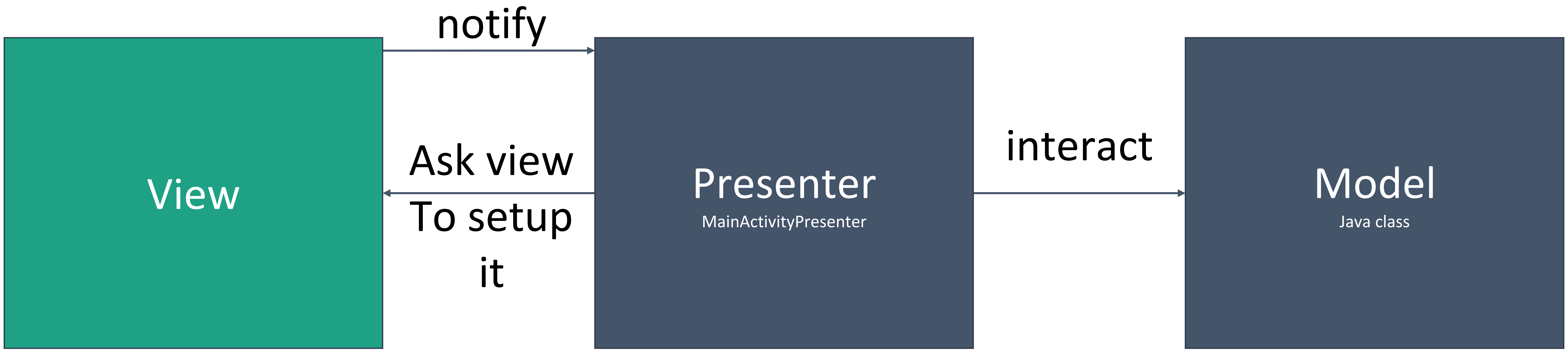
MVP: VIEW

- The only change here is that the Activity/Fragment is now considered part of the view. We stop fighting the natural tendency for them to go hand in hand. Good practice is to have the Activity implement a view interface so that the presenter has an interface to code to. This eliminates coupling it to any specific view and allows simple unit testing with a mock implementation of the view.



MVP: PRESENTER

- This is essentially the controller from MVC except that it is not at all tied to the View, just an interface. This addresses the testability concerns as well as the modularity/flexibility concerns we had with MVC. *In fact, MVP purists would argue that the presenter should never have any references to any Android APIs or code.*



SECTION 3:
MVVM

MVVM

- MVVM with Data Binding on Android has the benefits of easier testing and modularity, while also reducing the amount of glue code that we have to write to connect the view + model.

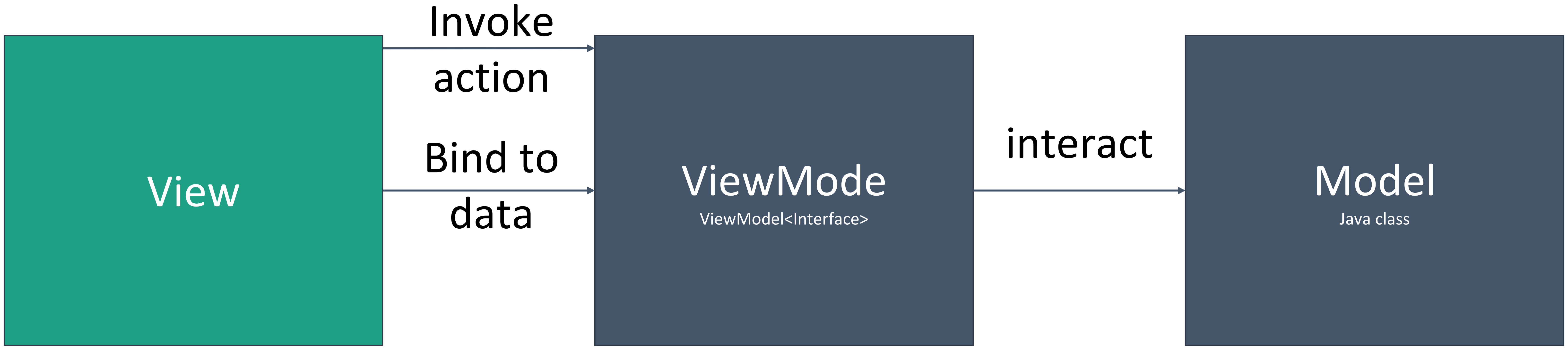
- Unit testing is even easier now, because you really have no dependency on the view. When testing, you only need to verify that the observable variables are set appropriately when the model changes. There is no need to mock out the view for testing as there was with the MVP pattern.

- *Maintenance* - Since views can bind to both variables and expressions, extraneous presentation logic can creep in over time, effectively adding code to our XML. To avoid this, always get values directly from the ViewModel rather than attempt to compute or derive them in the views binding expression. This way the computation can be unit tested appropriately.

MVVM: MODEL

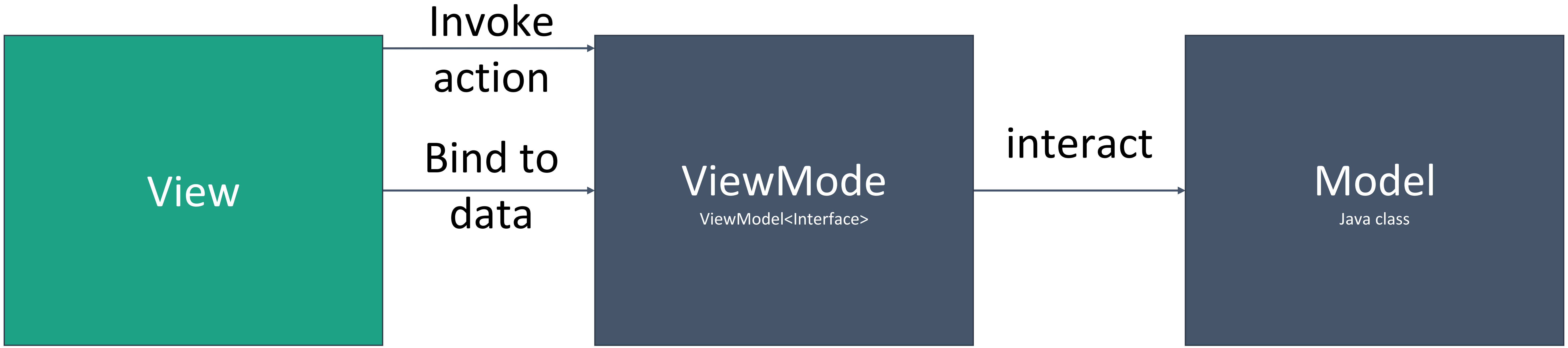
- Same as MVC / No change

- The model is the **Data + State + Business logic**. It’s the brains of our application so to speak. It is not tied to the view or controller, and because of this, it is reusable in many contexts.



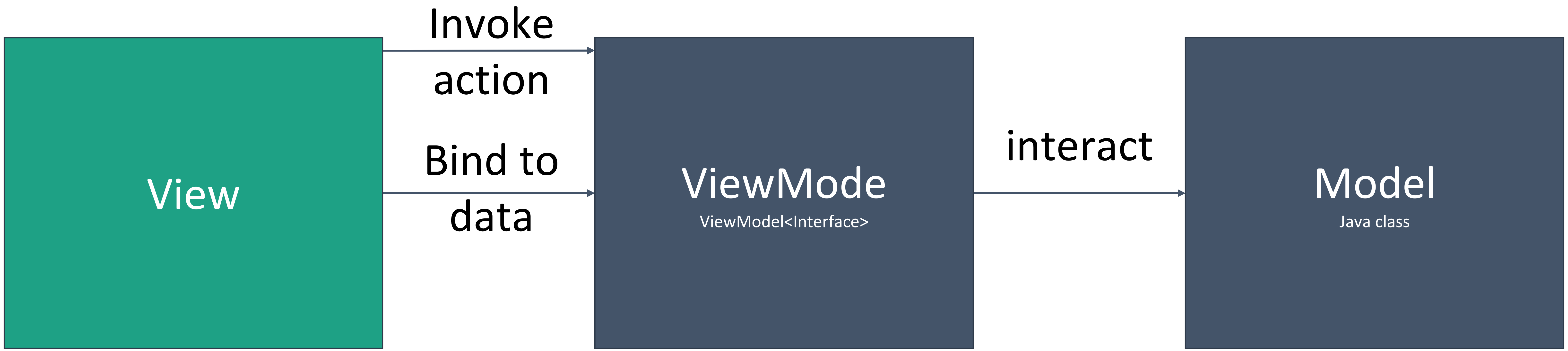
MVVM: VIEW

- The view binds to observable variables and actions exposed by the ViewModel in a flexible way. More on that in minute.



MVVM: VIEW MODEL

- The ViewModel is responsible for wrapping the model and preparing observable data needed by the view. It also provides hooks for the view to pass events to the model. The ViewModel is not tied to the view however.

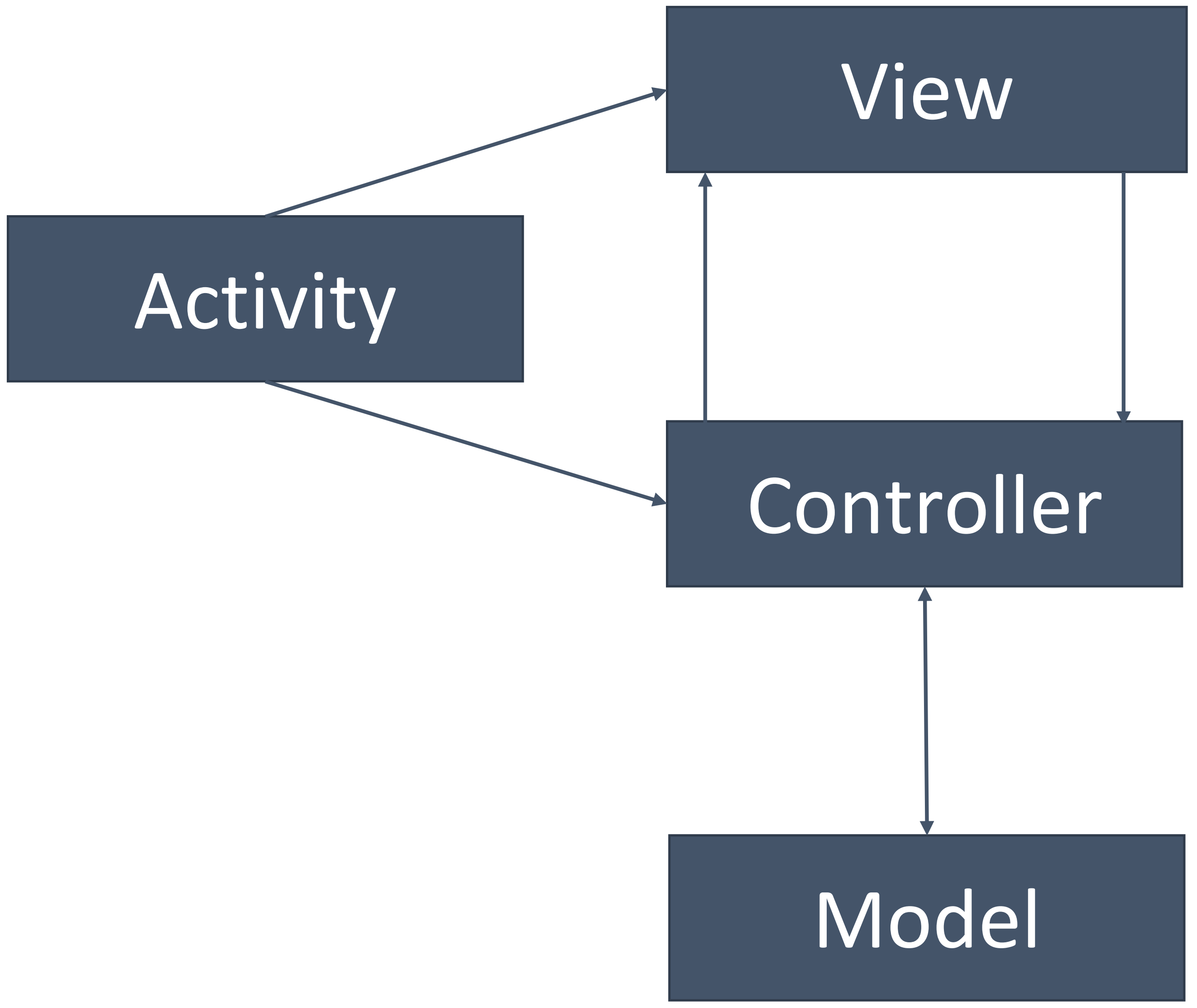


SECTION 4:
BEST PRACTICES ANDROID

CONCLUSION

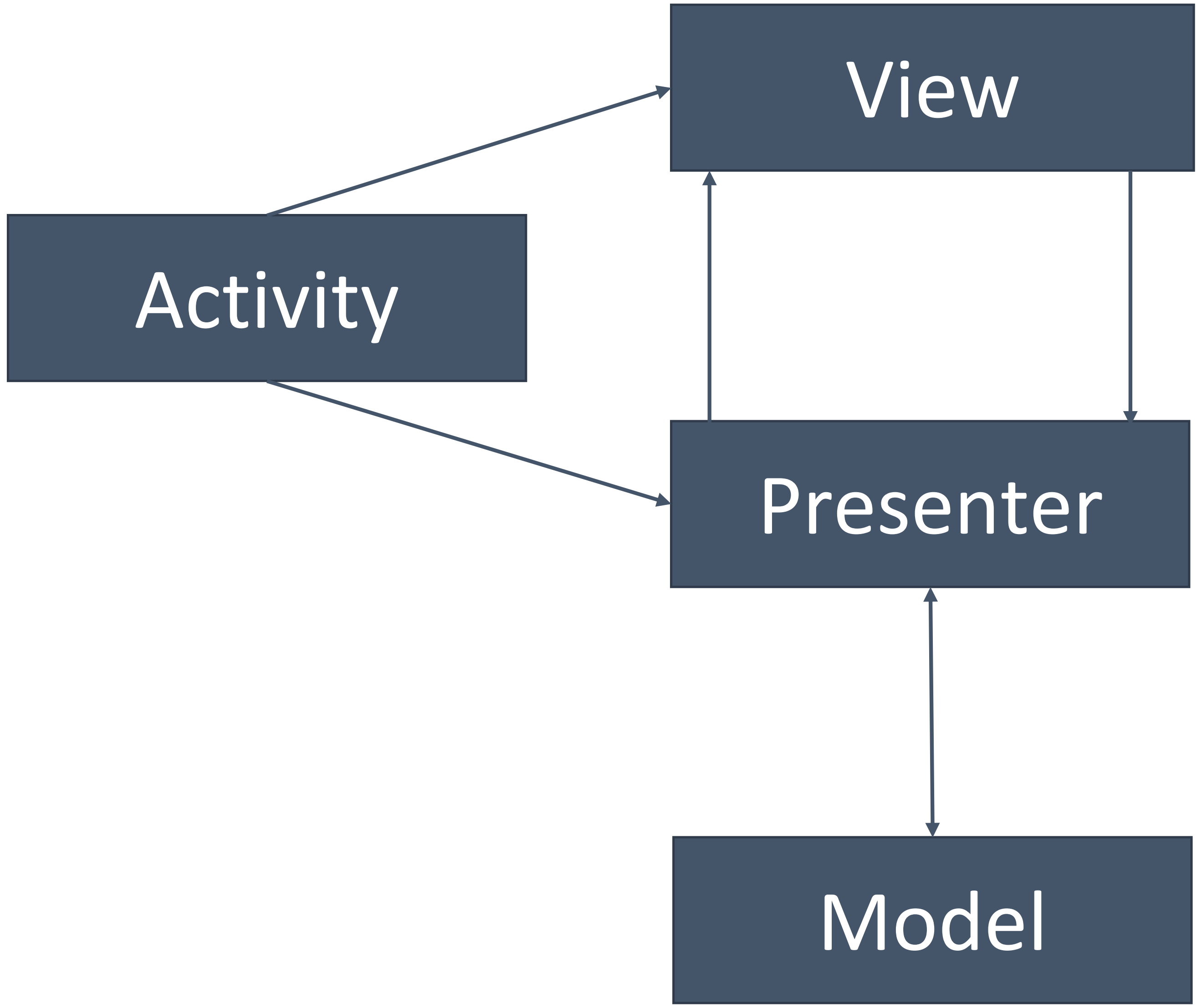
- Both MVP and MVVM do a better job than MVC in breaking down your app into modular, single purpose components, but they also add more complexity to your app. For a very simple application with only one or two screens, MVC may work just fine. MVVM with data binding is attractive as it follows a more reactive programming model and produces less code.

- So which pattern is best for you? If you're choosing between MVP and MVVM, a lot of the decision comes down to personal preference, but seeing them in action will help you understand the benefits and tradeoffs.



BEST PRACTICES

- Activity Classes for Dependency inversion
- Controller depends on abstractions (interfaces).
- Centralized places in code to control configuration and dependencies.
- Enables us to move to dependency injection in the future.
- Facades to decouple analytics, a/b tests etc.


View
Activity
Presenter
Model