# 12. Архитектура, внедрение зависимостей, Dagger 2

## Архитектура мобильного приложения

Три базовые архитектуры:

1. MVC
2. MVVM
3. MVP

Разобрать пример:

<https://academy.realm.io/posts/eric-maxwell-mvc-mvp-and-mvvm-on-android/>

## MVC

**Model**

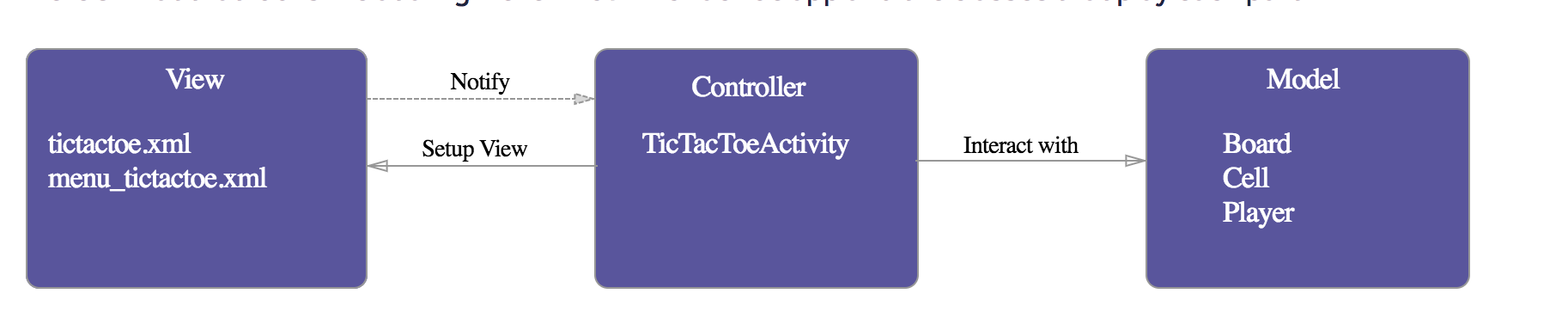
The model is the Data + State + Business logic of our Tic-Tac-Toe application. It’s the brains of our application so to speak. It is not tied to the view or controller, and because of this, it is reusable in many contexts.

**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.

**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.



#### Controller Concerns

* 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](https://martinfowler.com/bliki/AnemicDomainModel.html), more and more code starts getting transferred into the controllers, making them bloated and brittle.

## MVP

#### **Model**

Same as MVC / No change

#### **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.

#### **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.



## MVVM

#### **Model**

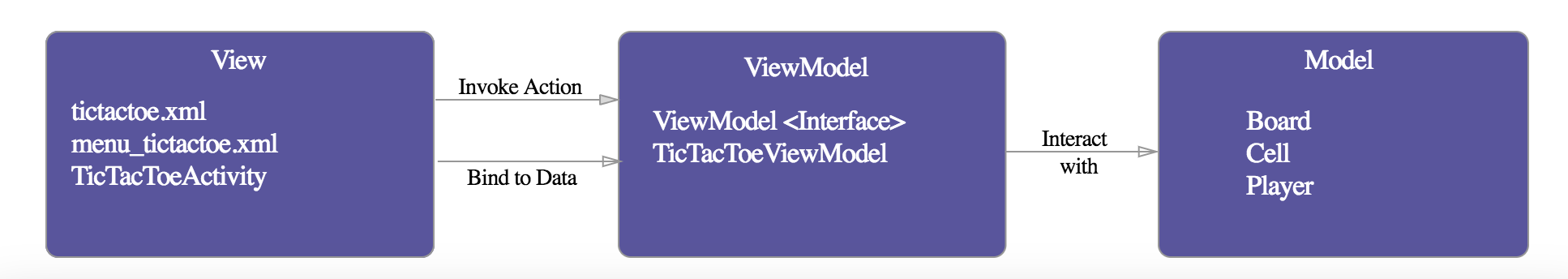
Same as MVC / No change

#### **View**

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

#### **ViewModel**

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.



<https://habr.com/post/278815/> - еще 1 пример MVP

## Внедрение зависимостей

<https://habr.com/post/343248/> - даггер

#### Чем плохи зависимости?

Большое количество зависимостей в классе приводит к проблемам сильных связей (*hard dependency*), что плохо по следующим причинам:

* Уменьшаются возможности повторного использования кода.
* Усложняется процесс тестирования.
* Ухудшается поддерживаемость (*maintainability*) кода при росте проекта.

## Dagger 2

#### Dagger 2 спешит на помощь

Dagger 2 — это один из фреймворков с открытым исходным кодом для внедрения зависимостей (далее буду использовать DI, от Dependency Injection), который генерирует большое количество шаблонного кода за вас. Почему он лучше остальных? Сейчас это единственный DI фреймворк, который генерирует полностью отслеживаемый Java код, имитирующий тот код, который вы могли написать вручную. Это означает, что в построении графа зависимостей нет никакой магии. Dagger 2 менее динамичен, чем другие (в нем не используется рефлексия), но простота и производительность сгенерированного кода находятся на том же уровне, что и у написанного вручную. Коротко, Dagger 2 генерирует весь шаблонный код для внедрения зависимостей за вас.

Другими словами, внедрение зависимостей основывается на [концепции инверсии контроля](https://en.wikipedia.org/wiki/Inversion_of_control), которая говорит о том, что класс должен получать свои зависимости извне. Говоря просто, ни один класс не должен создавать экземпляр другого класса, а должен получать все экземпляры из класса конфигурации.

Аннотация @Inject сообщит Dagger какие зависимости должны быть предоставлены зависимому объекту.

#### @Component

Это аннотация используется для интерфейса, который объединит все части процесса внедрения зависимостей. При использовании данной аннотации мы определяем из каких модулей или других компонентов будут браться зависимости. Также здесь можно определить, какие зависимости будут видны открыто (могут быть внедрены) и где компонент может внедрять объекты. @Component, в общем, что-то вроде моста между @Module (рассмотрим эту аннотацию позже) и @Inject.

Разобрать пример <https://github.com/ribot/android-boilerplate>

#### Полезные ссылки:

1. Обзор архитектур <https://academy.realm.io/posts/eric-maxwell-mvc-mvp-and-mvvm-on-android/>
2. Один из самых понятных циклов статей про даггер 2 <https://habr.com/post/343248/>

#### Задание:

Необходимо реализовать приложение с мостами из урока 7.  
Карту делать не нужно, напоминалку делать не нужно.   
Требования:  
1. Приложение должно содержать 2 экрана - список мостов и экран моста.

2. Список мостов получается из сети. Не забыть про три состояния интерфейса - загрузка, данные, ошибка.

3. **На экран моста передается только его айди**. Информацию о мосте получать отдельным запросом из сети уже на экране информации о мосте. также не забыть про три состояния интерфейса.

4. Использовать MVP архитектуру, разобранную на уроке. В качестве шаблона взять код из урока (<https://github.com/Tooto/android-school/tree/master/lesson-mvp> )

5. Можно использовать примитивное внедрение зависимостей, как на уроке.

6. Картинка на экране моста должна схлопываться в тулбар.

**Продвинутый уровень**: разобраться во внедрении зависимостей, с помощью dagger 2, и по аналогии с <https://github.com/ribot/android-boilerplate> сделать.