3. Design Pattern

- 생성
  - Builder
  - Dependency Injection
  - Singleton
- 구조
  - Adapter
  - Facade(퍼사드)
  - Facade 패턴은 다른 인터페이스 세트를 사용하기 쉽게 만드는 하이-레벨 인터페이스를 제공
- 행위
  - Command

EventBus in 3 steps
- Observer
- Model View Controller (MVC)
- Model View Presenter (MVP)
- Model View ViewModel (MVVM)

Model–view–viewmodel
- Contents
- Components of MVVM pattern[edit]
- 참고 문헌들

▪ 생성

▪ Builder

- 복잡한 인스턴스를 조합하여 만든 구조.

▪ NotificationCompat

```
Notification notification =new NotificationCompat.Builder
(this)
                                        .setSmallIcon(R.
drawable.ic_notification)
                                        .setContentIntent
(pendingIntent)
                                        .setTicker(message)
                                        .build();
```

AlertDialog

```
new AlertDialog.Builder(this)
    .setTitle("Builder Dialog")
    .setMessage("Builder Dialog Message")
    .setNegativeButton("Cancel", new DialogInterface.
OnClickListener() {
        @Override public void onClick(DialogInterface
dialog, int which) {
        }
    })
    .setPositiveButton("OK", new DialogInterface.
OnClickListener() {
        @Override public void onClick(DialogInterface
dialog, int which) {
        }
    })
    .show();
```

- Dependency Injection
  - 구성요소간 의존 관계가 소스코드 내부가 아닌 외부 설정파일등을 통해 정리.
  - 네트워크 클라이언트, 이미지 로더, SharedPreferences와 같이 앱의 다양한 지점에서 동일한 객체에 접근 해야할 때 사용하면 좋음
  - Android 는 Dagger2 프레임워크를 사용.
  - `@Module` 클래스 어노테이션을 달고 `@Provides` 메소드 어노테이션를 사용하여 주입하면 됨

▪ @Module

```
@Module
public class AppModule {
    @Provides
    SharedPreferences provideSharedPreferences
(Application app) {
        return app.getSharedPreferences("setting",
Context.MODE_PRIVATE);
    }
}
```

  - 위 `Module`은 필요한 객체를 생성하고 초기화함
  - 그런 다음 `Component` 인터페이스를 생성하여 모듈과 주입 할 클래스를 선언 ```java @Component (modules = AppModule.class) interface AppComponent {

}

- `Component` (Module) - `@Inject` ```java @Inject SharedPreferences sharedPreferences;

  - `Dependency Inject`을 사용하면 Activity에서 SharedPreferences 객체가 어떻게 생겼는지를 Activity가 알 필요없이 로컬 스토리지를 사용 할 수 있음
  - 자세한 구현 정보는 `Dagger2 Users Guide`를 참조

- Singleton
  - 클래스, 생성자가 여러 차례 생성되더라도 실제로 생성되는 객체는 하나
  - 최초 생성 이후에 호출된 생성자는 최초의 생성자가 생성한 객체를 리턴

▪ Singleton

```
public class SampleSingleton {
    private static SampleSingleton instance = null;

    private SampleSingleton() {

    }

    public static SampleSingleton getInstance() {
        if (instance == null) {
            synchronized(this) {
                if (instance == null) {
                    instance = new SampleSingleton();
                }
            }
        }

        return instance;
    }
}
```

- 구조
  - 구조패턴이란 기존에 생성되어 있는 클래스를 새롭게 구현하는 클래스에 맞지 않는 경우에 사용합니다. 작은 클래스의 합성을 통해 더 큰 클래스 구조를 형성하기 위한 패턴입니다.

- Adapter
  - Model, RecyclerView 연결하는 RecyclerView.Adapter

▪ SampleAdapter

```
public class SampleAdapter extends RecyclerView.
Adapter<SampleViewHolder> {
    private List<Test> tests;

    public SampleAdapter() {
        tests = new ArrayList();
    }

    @Override
    public SampleViewHolder onCreateViewHolder(ViewGroup
parent, int position) {
        // Create View Holder
    }

    @Override
    public void onBindViewHolder(SampleViewHolder
holder) {
        // View Bind
    }

    @Override
    public int getItemCount() {
        return tests.size();
    }

    public void add(Test item) {
        tests.add(item);
    }
}
```

- Facade(퍼사드)
  - Facade 패턴은 다른 인터페이스 세트를 사용하기 쉽게 만드는 하이-레벨 인터페이스를 제공
  - Retrofit Library가 대표적인 예.

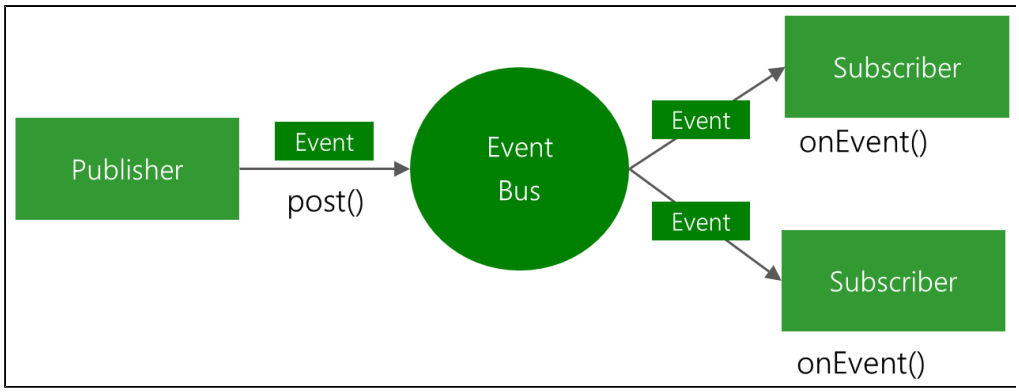
▪ Retrofit

```
public interface BooksApi {
    @GET("/books")
    void listBooks(Callback<List> callback);
}
```

▪ 행위

- 객체의 행위를 조직, 관리, 조합하는데 사용하는 패턴
- 객체들이 다른 객체와 상호작용하는 방식을 규정
- 각각 다른 객체들과 통신하는 방법과 객체의 책임을 규정하여 복잡한 행위들을 관리 할 수 있도록 함
- 두 객체 간의 관계에서부터 앱의 전체 아키텍처에까지 영향을 미침

- Command
  - 요청을 수행하는 객체가 별도의 수신자를 알지 못해도 요청을 실행하는 방식
  - 요청을 별도의 객체로 캡슐화하고 전송
  - EventBus가 대표적인 예 (https://github.com/greenrobot/EventBus)



- EventBus in 3 steps
  1. Define events:

     ```
     public static class MessageEvent { /* Additional fields if needed */ }
     ```

  2. Prepare subscribers: Declare and annotate your subscribing method, optionally specify a thread mode:

     ```
     @Subscribe(threadMode = ThreadMode.MAIN)
     public void onMessageEvent(MessageEvent event) {/* Do something */};
     ```

     Register and unregister your subscriber. For example on Android, activities and fragments should usually register according to their life cycle:

     ```
     @Override
      public void onStart() {
          super.onStart();
          EventBus.getDefault().register(this);
      }

      @Override
      public void onStop() {
          super.onStop();
          EventBus.getDefault().unregister(this);
      }
     ```

  3. Post events:

     ```
     EventBus.getDefault().post(new MessageEvent());
     ```

- Observer
  - 객체간 1:1 의존성을 정의
  - 하나의 객체가 상태가 변경되면 모든 종속된 객체에 자동 통지와 업데이트 수행
  - API 호출과 같이 불확실한 시간의 작업을 위해서 사용, 사용자 입력처리에도 사용

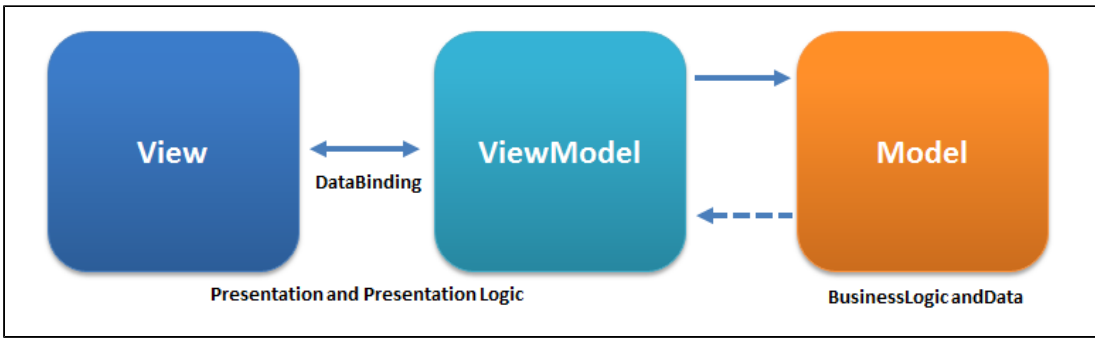
    - RxAndroid

    ```
    apiService.getData(someData)
        .observeOn(AndroidSchedulers.mainThread())
        .subscribe (/* an Observer */);
    ```

- Model View Controller (MVC)
  - UI와 비지니스 로직 분리
  - Model : 데이터 클래스, 실제 세계를 '모델링'
  - View : 시각적 클래스, 사용자에게 표시되는 모든 항목
  - Controller : Model과 View 사이의 접착제, View를 업데이트하고 사용자 입력을 받아 Model을 변경
  - Acticity(Controller)에 너무 많은 소스가 집약되기 때문에 좋은 방법이 아님.
- Model View Presenter (MVP)
  - 입력을 View에서 처리.
  - Presenter는 View의 인스턴스를 가지고 1:1 관계를 유지.
  - 동작 순서
    1. View에서 이벤트 발생하여 Presenter로 전달.
    2. Presenter는 이벤트에 따른 Model 조작.
    3. 결과 내용을 바인딩을 통해 View에게 통보
    4. View 업데이트
  - View를 추상화하여 View를 구현하지 않아도 UI로직에 대한 테스트 가능하고 유지보수 쉬움.
- Model View ViewModel (MVVM)
  - ViewModel은 Model과 View 사이의 접착제이지만 Controller와 다르게 동작

- View에 대한 명령을 표시하고 View를 Model에 바인딩
- Model이 업데이트되면 해당 View는 ViewModel을 통해 자신을 업데이트
- View가 없데이트되면 ViewModel은 Model을 통해 자신을 업데이트

Model-view-viewmodel



From Wikipedia, the free encyclopediaJump to navigationJump to search

Model-view-viewmodel (**MVVM**) is a software architectural pattern.

MVVM facilitates a separation of development of the graphical user interface – be it via a markup language or GUI code – from development of the business logic or back-end logic (the *data model*). The *view model* of MVVM is a value converter,[1] meaning the view model is responsible for exposing (converting) the data objects from the model in such a way that objects are easily managed and presented. In this respect, the view model is more model than view, and handles most if not all of the view's display logic.[1] The view model may implement a mediator pattern, organizing access to the back-end logic around the set of use cases supported by the view.

MVVM is a variation of Martin Fowler's Presentation Model design pattern.[2][3] MVVM abstracts a view's state and behavior in the same way,[3] but a Presentation Model abstracts a view (creates a *view model*) in a manner *not* dependent on a specific user-interface platform.

MVVM was invented by Microsoft architects Ken Cooper and Ted Peters specifically to simplify event-driven programming of user interfaces. The pattern was incorporated into Windows Presentation Foundation (WPF) (Microsoft's .NET graphics system) and Silverlight (WPF's Internet application derivative).[3] John Gossman, one of Microsoft's WPF and Silverlight architects, announced MVVM on his blog in 2005.[3]

Model-view-viewmodel is also referred to as **model-view-binder**, especially in implementations not involving the .NET platform. ZK (a web application framework written in Java) and KnockoutJS (a JavaScript library) use model-view-binder.[3][4][5]

Contents

Components of MVVM pattern[edit]

ModelModel refers either to a domain model, which represents real state content (an object-oriented approach), or to the data access layer, which represents content (a data-centric approach).[citation needed]ViewAs in the model-view-controller (MVC) and model-view-presenter (MVP) patterns, the view is the structure, layout, and appearance of what a user sees on the screen.[6] It displays a representation of the model and receives the user's interaction with the view (clicks, keyboard, gestures, etc.), and it forwards the handling of these to the view model via the data binding (properties, event callbacks, etc.) that is defined to link the view and view model.View modelThe view model is an abstraction of the view exposing public properties and commands. Instead of the controller of the MVC pattern, or the presenter of the MVP pattern, MVVM has a binder, which automates communication between the view and its bound properties in the view model. The view model has been described as a state of the data in the model.[7]The main difference between the view model and the Presenter in the MVP pattern, is that the presenter has a reference to a view whereas the view model does not. Instead, a view directly binds to properties on the view model to send and receive updates. To function efficiently, this requires a binding technology or generating boilerplate code to do the binding.[6]BinderDeclarative data and command-

binding are implicit in the MVVM pattern. In the Microsoft solution stack, the binder is a markup language called XAML.[8] The binder frees the developer from being obliged to write boiler-plate logic to synchronize the view model and view. When implemented outside of the Microsoft stack, the presence of a declarative data binding technology is what makes this pattern possible,[4][9] and without a binder, one would typically use MVP or MVC instead and have to write more boilerplate (or generate it with some other tool).

참고 문헌들

- 생성, 구조, 행위
    - https://chuumong.github.io/android/2017/01/16/%EC%95%88%EB%93%9C%EB%A1%9C%EC%9D%B4%EB%93%9C-%EB%94%94%EC%9E%90%EC%9D%B8-%ED%8C%A8%ED%84%B4
- mvc
    - uda 특성
        - https://academy.realm.io/kr/posts/eric-maxwell-uni-directional-architecture-android-using-realm/
- mvp
- mvi (model view intent)
    - http://hannesdorfmann.com/android/mosby3-mvi-1
    - http://hannesdorfmann.com/android/mosby3-mvi-2#model-view-intent-mvi
- mvvm
- di
- flux
    - https://github.com/lgvalle/android-flux-todo-app
- redux
    - https://medium.com/@trikita/writing-a-todo-app-with-redux-on-android-5de31cfbdb4f
- DiffUtil
    - https://medium.com/@ZakTaccardi/diffutil-one-way-data-flow-with-rxjava-and-kotlin-6e17f0cdef0c