앱 안정성 및 확장성 강화를 위한

안드로이드 아키텍처

-UI 계층-

[Part 2. 테스트 구현 2](#_Toc117491256)

[1. 좋은 아키텍처를 위한 올바른 테스트 2](#_Toc117491257)

[참고 링크 3](#_Toc117491258)

# Part 3. UI 계층

## 1. MVx의 대원칙

1 UI계층(MVx)의 대원칙

1.1 어떤 경우이든 Model은 분리되어야 함

-적어도 데이터 계층(로컬 DB, remote API 접근 등)에서 처리되는 모든 로직은 UI레이어에서 독립

1.2 뷰의 역할을 할 수 있는 한 분리시켜야 한다는 공통의 문제 의식

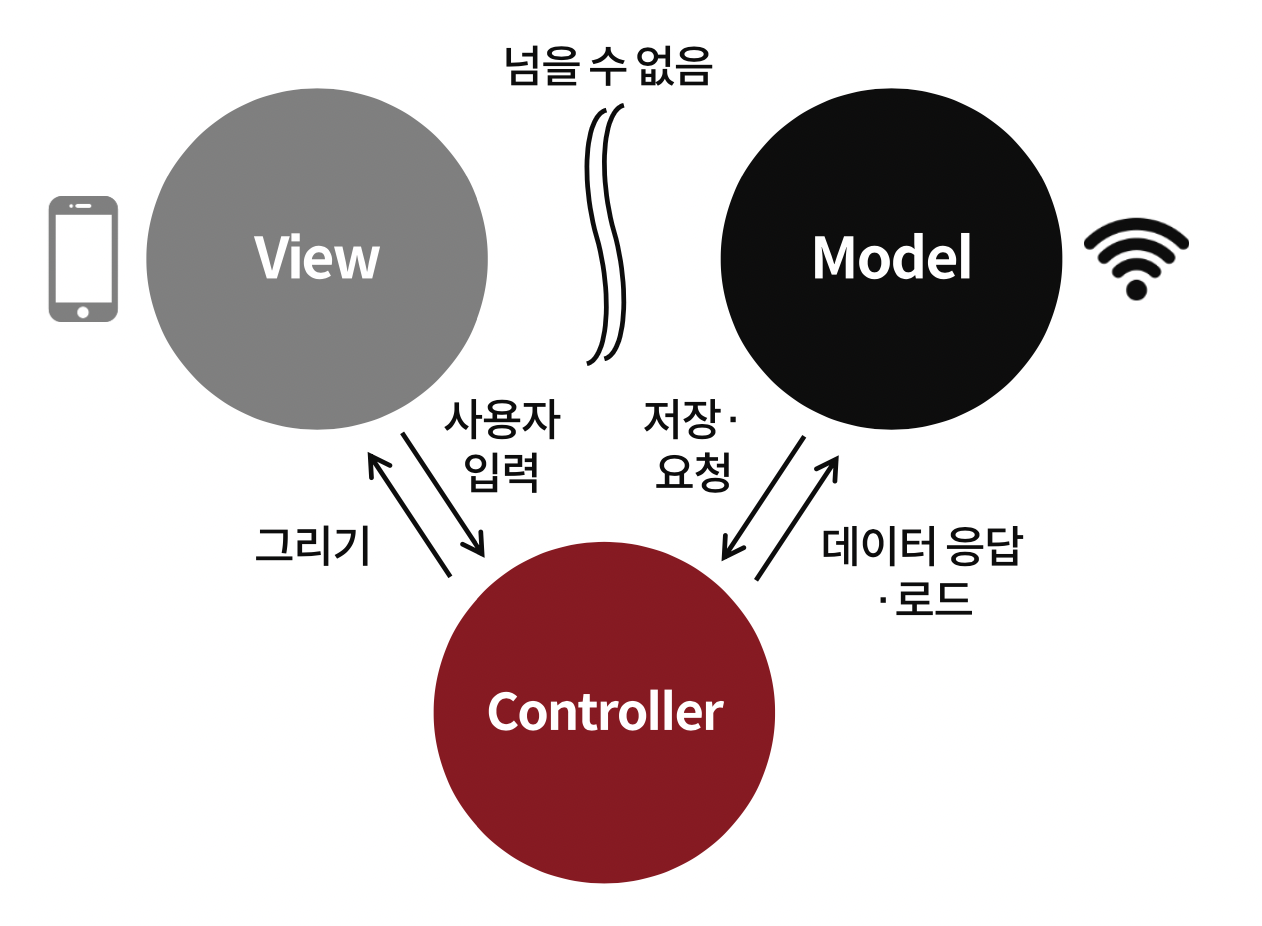
1.3 UI계층에서 Android에서 발생할 수 있는 특수한 상황들을 잘 처리할 수 있는 체계가 필요

-Context처리

-Life Cycle 이벤트 처리: 앱의 안정성을 담보할 필요가 있고, 이벤트 처리 과정에서 사용자에게 기대하지 않는 동작을 보여줘서는 안됨

-디바이스 종류가 다양하기 때문에 디바이스마다 테스트하는 것이 불가능함. 하지만 할 수 있는 한 많은 부분이 테스트 가능하도록 만들어 져야함

2 MVC 패턴



[그림 1.1 MVC 패턴]

2.1 모델(M)과 뷰(V)는 일종의 레고 블록. 컨트롤러(C)는 레고블럭의 조율을 담당

-모델(Data+Domain) 계층에는 비지니스 로직을, 뷰는 UI로직을 제공

-C는 어떤 V를 보여줄 것인가를 결정해 M에서 받은 데이터/에러를 V로 넘겨줌.

-플랫폼을 막론하고 유용하게 적용되는 패러다임. 특히 웹에서 잘 동작

4 왜 안드로이드에서 MVC 는 잘 동작하지 않는가?

4.1 모바일 환경의 문제

-복잡한 비동기 처리

-Life Cycle 처리

-UI 로직 분리의 어려움

cf. 웹의 html에서는 뷰가 컨트롤러와 완전히 독립된 형태로 UI로직을 구현 가능하지만, 모바일은 그렇지 못함

4.2 안드로이드 문제 : V-C 분리가 애매하다

-뷰: 안드로이드의 XML은 기본 레이아웃만을 제공. UI로직이 들어갈 여지가 없음

-컨트롤러: Activity/Fragment가 뷰, 컨트롤러 모두 담당하게 됨

->로직이 너무 많이 들어가게 되면Fat Activity 발생 -> 가독성, 유지보수성, 확장성 저하 -> Activty 자체가 Context기 때문에 Unit Test를 만들기가 매우 까다로움

5 MVC해결책과 한계점

5.1 뷰의 분화

-안드로이드는 <include> 태그를 통해 XML정의를 여러 개로 분리 가능

->나눈 View들은 Activity/Fragment가 아닌 별도의 컨트롤러를 통해 제어

5.2 ViewController

-Activity/Fragmet는 뷰도 컨트롤러도 아니도록 설계(화면 안 요소들을 생성, 라이프 사이클 처리, 그리고 Context에 밀접한 처리를 bridge해주는 역할만 남김)

-각 뷰마다 ViewController를 만들어 뷰의 동작에 관련된 로직 및 컨트롤러 로직을 여기에 구현. 단, ViewController는 설정 변경(configuration change)으로 인한 라이프사이클 변화에서 살아남도록 구현해야 함

ex. Hilt 의 @RetainedActivityScope 지정 //설정 변환(가로세로 전환/다크 모드 변환 등)이 있어도 살아남을 수 있는 scope

5.3 한계점

-5.1, 5.2로 설계해도 사용자 이벤트와 외부 이벤트 등의 효과적인 처리가 여전히 어려움

-ViewController의 상당수 동작을 위해 Context가 필요한데 Fragment에 연결된 ViewController라면 설정 변경에서 살아남게 만든다는 것이 쉽지 않음

-대부분의 테스트 케이스에 Context가 필요하므로 테스트 작성도 까다롭고 실행 속도도 느림

6 Non-MVC의 설계 전략

6.1 MVP 접근법

-Activity에서 뷰(V)와 컨트롤러(C)의 역할을 최대한 빼앗아 뷰(V)와 프레젠터(P)로 넘김

-Activity는 객체 생성 및 순수 흐름 관리 역할 위주

6.2 MVVM/MVI 접근법

-단방향 데이터 흐름(Uni-directional Data Flow): ‘컨트롤러->뷰’ 방향의 데이터 흐름을 이벤트를 수신하는 형태로 구현

-Activity는 최대한 일부 Context의존 가능만 하도록 설계

-뷰 로직은 최대한 dataBinding으로 구현

## 2 MVP, non-MVC 에서 공통적으로 고려해야할 것들

2.1 MVP

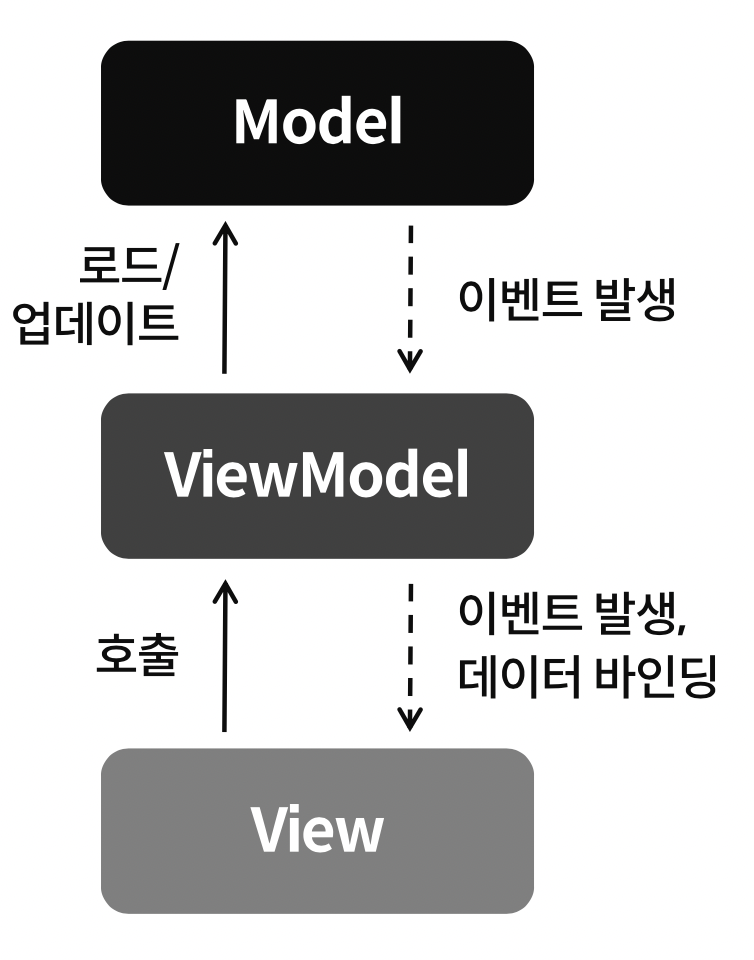
2.2 MVP 의 장점

2.3 MVP 의 단점

2.4 MVP 그리고 non-MVC에서의 주의사항

## 3 MVVM

1. M-V-ViewModel



[그림 3.1MVVM 패턴]

1.1 MVVM 주요 특징

1.1.1 단방향 데이터 흐름(UDF, Uni-directional Data Flow)

-뷰는 자신이 받은 사용자 입력을 전달만 할 뿐 직접적으로 결과를 받지 않는다. 대신 간접적으로 이벤트 형태로 상태 변경을 통보 받는다.

Cf. UDF 원칙

-상위객체(ex. State Holder)는 하위 객체로부터 상태를 읽을 수 없어야 한다.

-하위 객체(ex. View)는 상위 객체의 이벤트를 직접 읽을 수 없고, 상위 객체도 하위 객체로 직접 이벤트를 보낼 수도 없다.

1.1.2 VIewModel: Uni-directional Mediator

-프레젠터와 매우 유사하나, 뷰모델은 뷰를 제어하지 않음

-뷰는 뷰모델을 호출하되, 결과는 Callback / Observable 형태로 받음

-뷰모델은 그 자체로 독립적인 시스템. 뷰가 없어도 동작 가능하며 결과를 이벤트로 보낼 뿐 수신인의 존재를 모름

1.2 MVVM 단점

-학습 곡선 : 이벤트는 기본적으로 반응형(Reactive)로 받아야 하므로, 이 개념에 대한 숙지가 필요하며 RxJava, Coroutine 라이브러리에 대한 학습이 필요

-모든 비지니스로직을 ViewModel로 이전 시켰기 때문에 Fat ViewModel 이 될 위험성이 있음

-순환 이벤트 흐름을 완전히 막기 어려움 -> Databinding 을 사용해 최소화

1.3 Fat ViewModel 해결책

-뷰를 최대한 분리, 그리고 뷰마다 뷰모델도 역시 함께 분리

Activity/Fragment 의 사이즈가 작다면 ViewModel 과 1:1 구조가 맞으나, 사이즈가 크다면 일정 기준(ex 통신 서버)으로 ViewModel 을 분리해 1:N 구조로 설계할 수 있음. 또 ViewModel들 끼리 데이터를 공유할 수 있는 공유 ViewModel을 설계할 수 있음

-뷰모델의 비지니스 로직 중 도메인 로직에 해당하는 부분은 분리하여 도메인 계층으로 이동

2. 데이터 바인딩

-이벤트를 뷰로 바로 적용할 수 있는 매커니즘을 제공

-데이터 바인딩이 없다면 순환 이벤트 흐림이 발생

cf. 순환 이벤트

-뷰 이벤트 발생 -> 뷰모델이 처리 후, 이벤트로 알려 줌 -> 뷰가 다시 이를 받음

**3. AAC ViewModel**

3.1 AAC ViewModel 에서 제공하는 것

-생명주기 내에서 설정 변경과 프로세스 종료가 일어나도 ViewModel의 내용이 보존되는 구조를 제공한다.

-일반 변수, Flow, LiveData, Compose State 가 보존 될 수 있도록 하는 API 제공 – SavedStateHandle

-Coroutines 가 ViewModel 의 생명주기 내에서 동작될 수 있도록하는 환경을 제공 – ViewModel Scrope

3.2 구글은 왜 ViewModel만을 AAC에서 지원하는가?

-기존 Activity/Fragment 의 형태를 깨뜨리지 않으면서 재사용성/생산성 높은 아키텍처를 구현 가능하기 때문이다.

-ViewModel 은 UI에 중요한 상태 정보를 갖고 있기 떄문에 안드로이드 생존 주기 처리에 가장 잘 어울리는 구조다.

ex. 화면이 회전되어 생명 주기가 다시 시작된다고 해도 ViewModel 내부 데이터는 보존된다.

**4. ViewModel Antipatterns : 잘못된 사용**

4.1 상태가 유실 될 수 있는 공급자(Provider)를 통해 이벤트를 전달한다.

-Coroutine-Channel / RxJava-Flowable 등의 API 는 데이터 전달을 완전히 보장하지 않음

cf. Channel : 이전 정보는 유실되어도 상관없는 실시간 위치 정보를 다룰 때 사용

4.2 UI코드에서 액션까지 구현한다.

-UI는 현재 상태가 무엇이고, 어떻게 반영할지만 결정. 비지니스 로직, 네비게이션 등은 ViewModel의 영역

4.3 1회성 이벤트(ex. Dialog, SnackBar 등)를 즉시 처리하지 않는다.

-Channel.send(), MutableLiveData.postValue() 등의 API는 낮은 우선순위로 실행될 수 있음

4.4 UI 사태 처리에 Compose 의 State 대신 StateFlow/SharedFlow를 이용한다.

->불필요한 오버헤드 발생

cf. 오버헤드 :

4.5 ViewModel 생성자에서 초기화 이상의 일을 한다.

-단일 책임 원칙 위반

-테스크 구현이 어려움

-상속 시 자식에게 과도한 정보 습득 요구

-사용자에게 필요없는 초기화를 수행하지 않게 만들 선택권 박탈

cf. 생성자에서 하면 좋지 않은 일 : 다른 객체의 생성, 정적 메서드 호출, 필드에 단순 대입 이상의 초기화 로직 구현, if/when 분기 로직

## 4 MVI-Flux

## 5 기본적인 뷰모델 구현

# 참고 링크

Test Coverage

의존성 주입 라이브러리 Hilt

<https://developer.android.com/training/dependency-injection/hilt-testing>