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

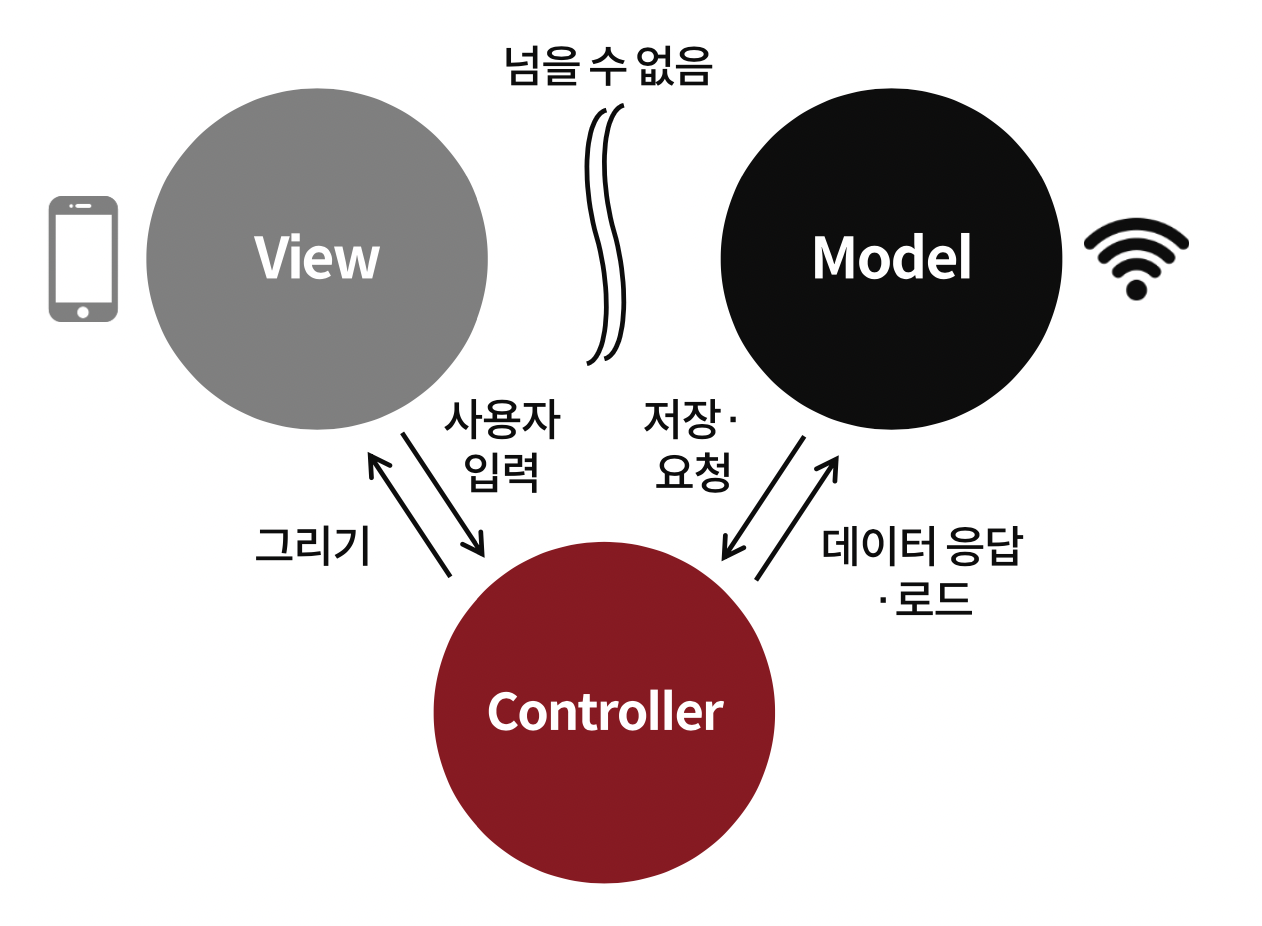
1.3 UI계층에서 Android에서 발생할 수 있는 특수한 상황들을 잘 처리할 수 있는 체계가 필요

-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 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

## 4 MVI-Flux

## 5 기본적인 뷰모델 구현

# 참고 링크

Test Coverage

의존성 주입 라이브러리 Hilt

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