앱 안정성 및 확장성 강화를 위한

안드로이드 아키텍처

-모바일 아키텍처 개론-

[Part 0. Introduction 2](#_Toc115850962)

[1.왜 아키텍처가 중요한가 2](#_Toc115850963)

[Part1 모바일 아키텍처 개론 3](#_Toc115850964)

[1. 좋은 설계를 위한 첫걸음 : 복잡성 제거 3](#_Toc115850965)

[2. SOLID 원칙 5](#_Toc115850966)

[3. 모바일 클린 아키텍처 7](#_Toc115850967)

[4. Google에서는 어떻게 좋은 설계를 만들어내는가 9](#_Toc115850968)

[참고 링크 10](#_Toc115850969)

# Part 0. Introduction

## 1.왜 아키텍처가 중요한가

1앱의 생존을 위해서 필요

-앱에는 시스템 이벤트, 라이프사이클 이벤트 등의 수많은 이벤트가 발생하는데 잘못된 설계는 프로그램의 유지보수성을 극도로 떨어뜨림

-이벤트 처리 결과는 동시다발적으로 불시에 일어날 수 있으며 대응이 지연되면 앱이 정지 될 수 있음(반응형 프로그래밍 구조가 필요한 이유)

-웹 개발에 비해 빠른 배포가 어렵기 때문에 더 견고한 아키텍처 기반이 필요

-네트워크가 끊어졌어도 동작해야 함(유연한 데이터 계층 구현이 필요한 이유)

2비즈니스를 촉진

-성공한 앱을 더 빨리 확장시키려면 더 많은 모듈화, 더 유연한 아키텍처가 필요

-성공한 앱이 더 빨리 솟아오르려면 수많은 실험을 시도해야 하며, 실험 기반 시스템이 필요

-더 깊은 사용자화 위해서는 극도로 유연한 UI 구조가 필요

# Part1 모바일 아키텍처 개론

## 1. 좋은 설계를 위한 첫걸음 : 복잡성 제거

1좋은 아키텍처란?

-풀려는 문제에 잘 어울리는 설계

-클래스 등의 코드 구조가 시스템이 어떻게 동작하는지를 잘 보여줌

-요구사항이 진화함에 따라 쉽게 변경이 가능

-인과관계파악/유지보수/테스트가 쉬움

2복잡성이란?

-시스템을 이해하기 어렵고 수정하기 어렵게 만드는 소프트웨어 구조에 관련된 모든 것

-코드의 줄 수는 복잡성과 무관함

3복잡성이 나타나는 징후

-변경 증폭(Change Amplification) : 작은 수정 -> 여러 곳의 코드 수정

-인지적 부하(Cognitive Load) : 코드를 이해하기 위해서 작성자가 남긴 문서를 참고하는 등 많은 선수 지식을 알아야 하는 경우

-알 수 없는 무지(Unknown unknowns) : 결과가 나온 원인 파악이 어려운 경우

4복잡성을 조장하는 요인 : 낮추는 방법 : 추상화

-의존성(Dependencies) : 특정 코드, 모듈이 독립적으로 이해되지 않고 수정될 수 없는 경우 -> 의존성은 완전히 없앨 수는 없지만 최소화되어야 함

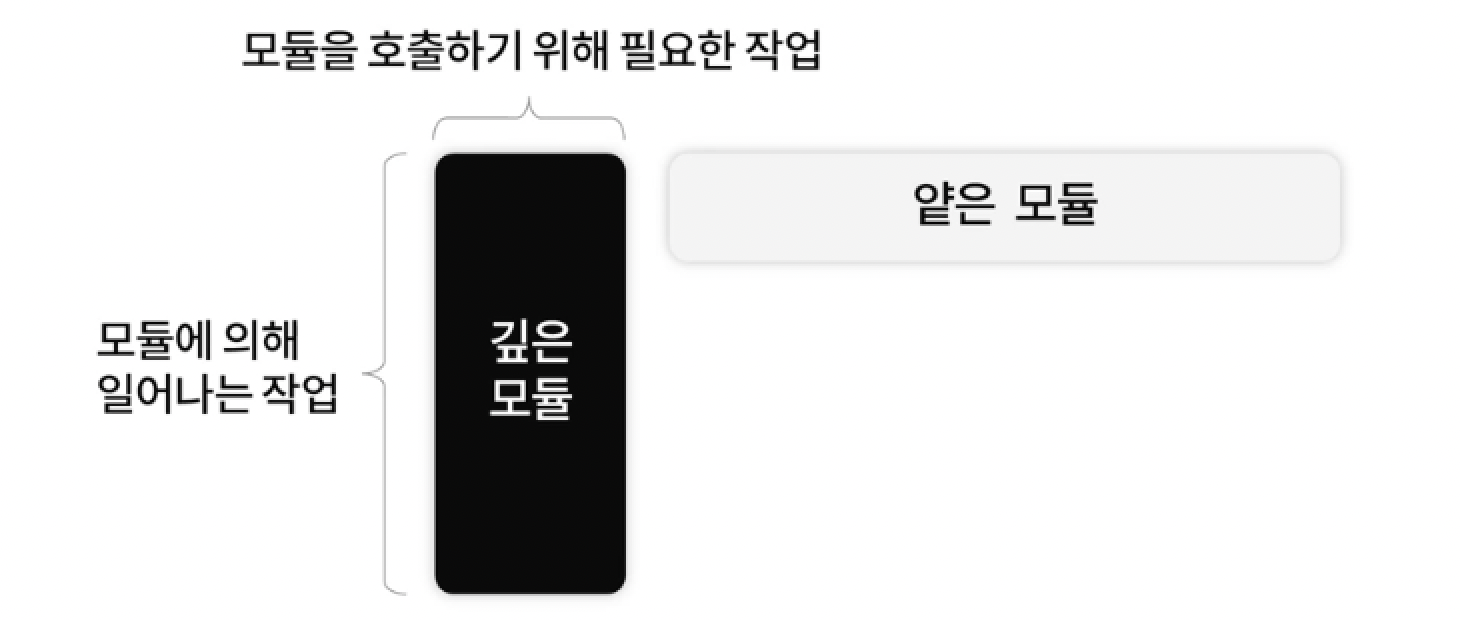
-불명확함(Obscurity) : 중요한 정보가 불명확해 중간 과정을 이해하지 못하면 결과를 이해할 수 없는 경우

-전술적 프로그래밍(Tactical Programming): 기능 구현만 목표로 두고 당장의 문제를 해결하는 경우. 이후 고도화를 위해서 더 많은 비용이 듦

5 추상화 방법

-추상화 : 사용자 관점의 불필요한 정보를 감춤으로써 기능 구현을 위해 최소한의 정보(인풋, 아웃풋, 기능 등)만 알아도 되는 간소화된 뷰

-깊은 모듈 : 인터페이스만 노출되어 있고 내부 코드는 깊게 있어 외부에서는 어떤 일이 일어나는지 알 수 없음. 하나의 간단한 요청으로 많은 작업이 완료됨. ex. file.open()(<-> 얕은 모듈 ex. 래퍼클래스, 나쁜 추상화 등)



[그림 1.1] 깊은 모듈/얕은 모듈

6범용 인터페이스 판단을 위한 질문

-범용 인터페이스 : 어떠한 입출력 장치에서도 연결하여 사용될 수 있는 인터페이스

-현재의 요구사항을 모두 만족하는 것 중에 무엇이 가장 심플한 인터페이스인가?

-이 메서드가 얼마나 많은 상황에서 사용될 수 있는가? -> 답이 하나라면 적신호

-이 API는 현재 나의 요구사항을 해결하는 데에 실제로 사용하기 편한가?

7정보 은닉

-사용자가 내부에서 어떤 일이 일어나는지 모르게 불필요한 정보를 감춤(인캡슐레이션)

-정보 은닉이 잘 되지 않으면 사용자가 일반적인 작업들을 위해서 명료하지 않은 기능들을 배워야함

8복잡성을 아래로(내부로) 끌어내리기 원칙

-전제 : 심플한 인터페이스는 심플한 구현보다 나음

-기존에 있는 기능과 밀접하게 연관되어 있는 경우

-앱의 여러부분들을 더 단순하게 만들어 주는 경우

-클래스의 인터페이스를 단순하게 만들어주는 경우

|  |
| --- |
| //bad case  bufferedReader.getBufSize()  bufferedReader.adjustBufSize()  bufferedReader.readNext()  //good case  //bad케이스의 bufferedReader.getBufSize()/.adjustBufSize() 가 bufferedReader.readNext() 내부로 끌어내려진 상황  bufferedReader.readNext() |

**[코드1 복잡성 끌어내리기 예시 코드]**

9 추상화 사이의 경계 찾기

9.1이럴 때는 합치기

-정보가 공유되는 경우

-함께 있는 것이 인터페이스를 단순하게 만드는 경우

-코드 중복을 없애주는 경우

-코드의 반복은 추상화가 이뤄지지 않았음을 의미하는 적신호일 수 있음

9.2이럴 때는 나누기

-특정 목적의 API가 범용 클래스 안에 있는 경우

-다른 종류의 범용 매커니즘이 함께 있는 경우

10 재사용성에 관한 흔한 실수

-재사용성 : 독립적인 요소로 타 모듈에서 참조될 수 있음을 의미

-재사용성 확보를 위해 DRY(Do not Repeat Yourself)원칙 차원에서만 바라봄.

이 경우 성급하게 공통 요소를 부모 클래스로 만들 수 있음 -> 자식 클래스가 새롭게 만들어 질 때 이 부모의 공통 요소를 잘 이해하고 있어야함

-중복 코드가 없는 한 하나의 클래스/메서드를 크게 만들어도 된다는 오해가 생길 수 있음

재사용성은 하나의 기능/클래스가 앱의 여러 군데에서 재활용되는 것을 의미하는 것이 아님.

## 2. SOLID 원칙

-중간 규모의 소프트웨어 구조(모듈)가 유지보수성, 가독성, 낮은 결합도, 높은 응집도 등 요구를 만족시키는 것이 목표

1단일 책임 원칙(Single Responsibility Principle)

-하나의 모듈은 오직 하나의 액터(사용자)에 대해서만 책임져야함(작동해야함)

-하나의 클래스가 하나의 일만 해야한다는 뜻이 아님에 유의

|  |
| --- |
| function Employee(name, pos, hours) {      this.name = name;      this.pos = pos;      this.hours = hours;  *//* 재정팀이사용할급여계산메서드      this.calculatePay = function() {          ...      }  *//* 인사팀이사용할근무시간계산메서드      this.calculateWorkHours = function() {          ...      }  *//* 시스템이사용할데이터갱신메서드      this.updateData = function() {          ...      }  } |

**[코드2 단일 책임 원칙 예외 예시]**

-[코드2]는 Employee 모듈을 설계하고, 모듈에서 사용할 여러 메서드를 한번에 정의한 모습인데, 메서드들이 각각 다른 사용자(재정팀, 인사팀, 시스템)를 위해 설계된 함수이므로 이는 단일 책임 원칙에 어긋나는 예시임

|  |
| --- |
| function EmployeeData(name, pos, hours) {      this.name = name;      this.pos = pos;      this.hours = hours;  }  *//* 재정팀이사용할기능을 *PayCalculator* 모듈로분리한다*.*  function PayCalculator(employeeData) {      this.data = employeeData;      this.calculatePay = function() {          ...      }  }  *//* 인사팀이사용할기능을 *WorkHourCalculator* 모듈로분리한다*.*  function WorkHourCalculator(employeeData) {      this.data = employeeData;      this.calculateWorkHour = function() {          ...      }  }  *//* 시스템이사용할기능을 *SystemDataUpdator* 모듈로분리한다*.*  function SystemDataUpdator(employeeData) {      this.data = employeeData;      this.updateData = function() {          ...      }  } |

**[코드3 코드2 리팩토링]**

-[코드3]의 모듈들을 저마다 다른 사용자를 위해 동작하므로, 단일 책임 원칙을 충족했다고 볼 수 있으며 유연한 설계가 가능함

2개방-폐쇄 원칙(Open-Closed Principle)

소프트웨어 모듈은 확장에는 열려 있어야하고, 변경에는 닫혀 있어야함

-‘열려 있다’ : 데이터 구조에 필드를 추가하거나, 함수에 새로운 요소를 추가하는 것이 가능해야 함

-‘닫혀 있다’ : 내부 코드를 변경해도 이를 사용하는 외부 모듈에는 영향을 미치지 않아야 함

2.1 클래스 단위의 개방-폐쇄 원칙

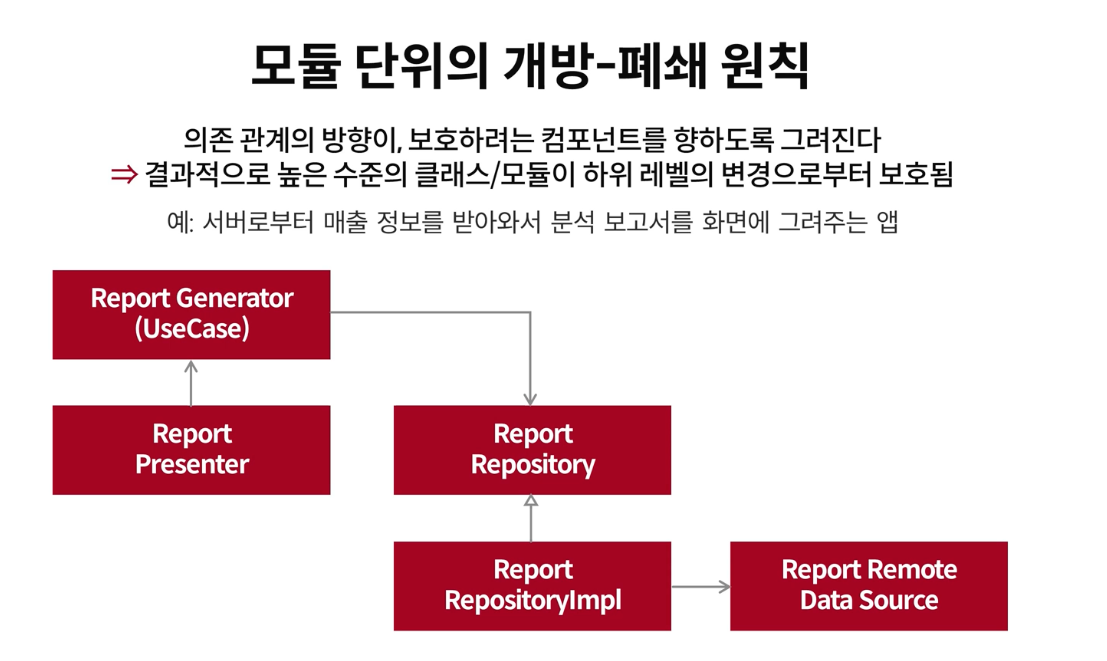
-내부 데이터의 변경이 클래스 사용자에게 의도하지 않게 영향을 미칠 수 있기 때문에 모든 멤버 변수는 private 이나 protected여야 함

-글로벌 변수를 참조하는 모듈이나 클래스가 외부 값에 의존하게 되면 폐쇄 원칙이 깨지기 때문에 예기치 못한 변경이 생길 수 있음

-추상 클래스(abstract/interface) 따로 만들어 사용하면 사용자에게 필요한 것만 노출함으로써 폐쇄 원칙을 만족하고, 구현 클래스 내용을 감춤으로 수정도 용이

2.2 모듈 단위의 개방-폐쇄 원칙

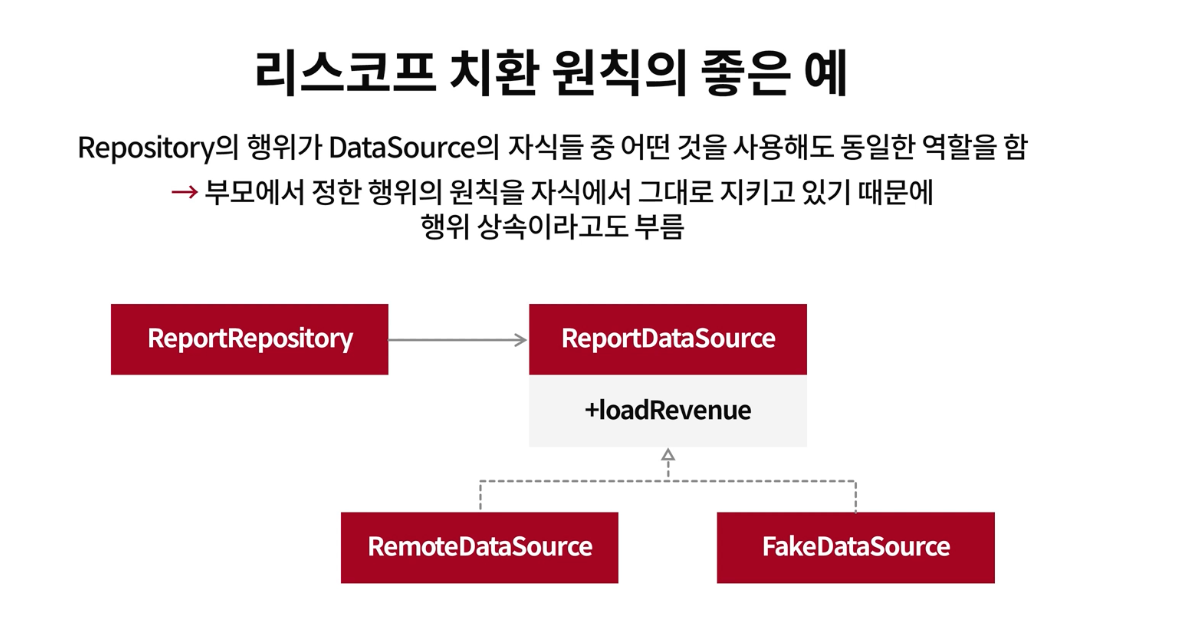
-의존 관계의 방향이, 보호하려는 컴포넌트를 향하도록 그림. 결과적으로 높은 수준의 클래스/모듈이 하위 레벨의 변경으로부터 보호됨



[그림 1.4] 서버로부터 매출 정보를 받아와 분석 보고서를 화면에 그려주는 앱

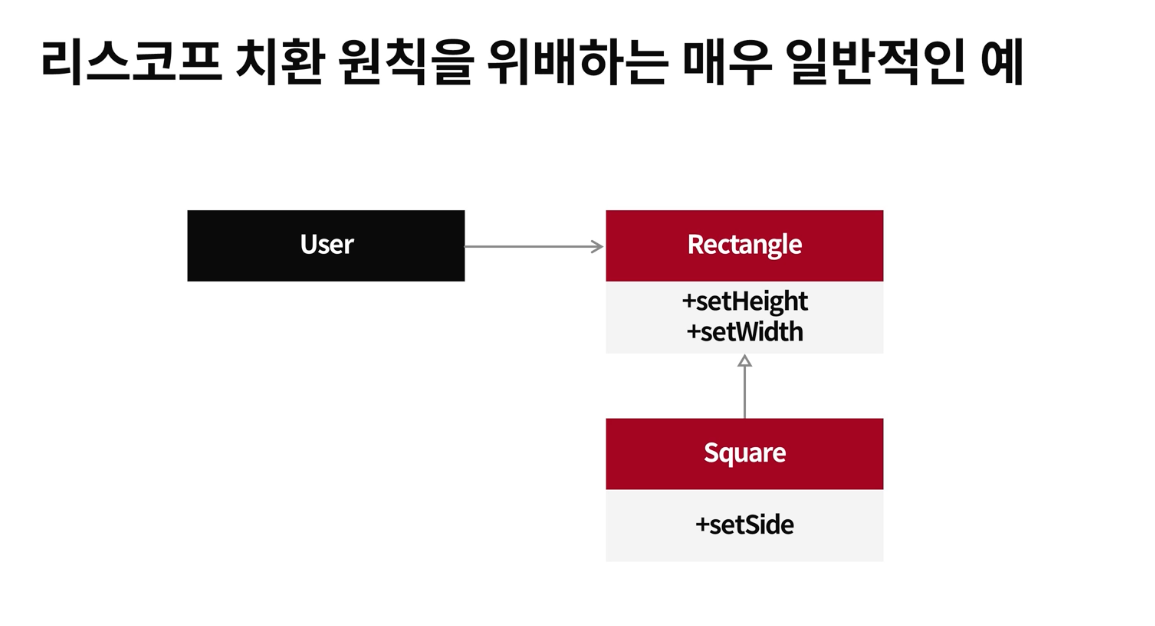
3리스코프 치환 원칙(Liskov Substitution Principle)

-부모 객체와 이를 상속한 자식 객체가 있을 때 부모 객체를 호출하는 동작에서 자식 객체가 부모 객체를 완전하게 대체할 수 있다는 원칙



[그림 1.5] 리스코프 치환 원칙의 좋은 예

-[그림 1.5] Repository의 행위가 DataSource의 자식들 중 어떤 것을 사용해도 동일한 역할을 함. 부모에서 정한 행위의 원칙을 자식에서 그대로 지키고 있기 때문에 행위 상속이라고도 부름



[그림 1.6] 리스코프 치환을 위배하는 일반적인 예

-[그림 1.6] 정사각형(Square)를 직사각형(Rectangle)이 상속하는 구조. 이론적으로는 정사각형은 직사각형의 한 종류지만 자식 객체인 정사각형이 부모 객체인 정사각형을 완전하게 대체할 수 없다.

4인터페이스 분리 원칙(Interface Segregation Principle)

-객체는 자신이 호출하지 않는 메서드에 의존하지 않아야 한다는 원칙

-추이 종속성(transitive dependency)를 막기 위해서, 유즈케이스가 DataStore에 종속성이 생기는 걸 막기 위해서 인터페이스가 필요하며, 외부 모듈에 필요한 인터페이스만 노출

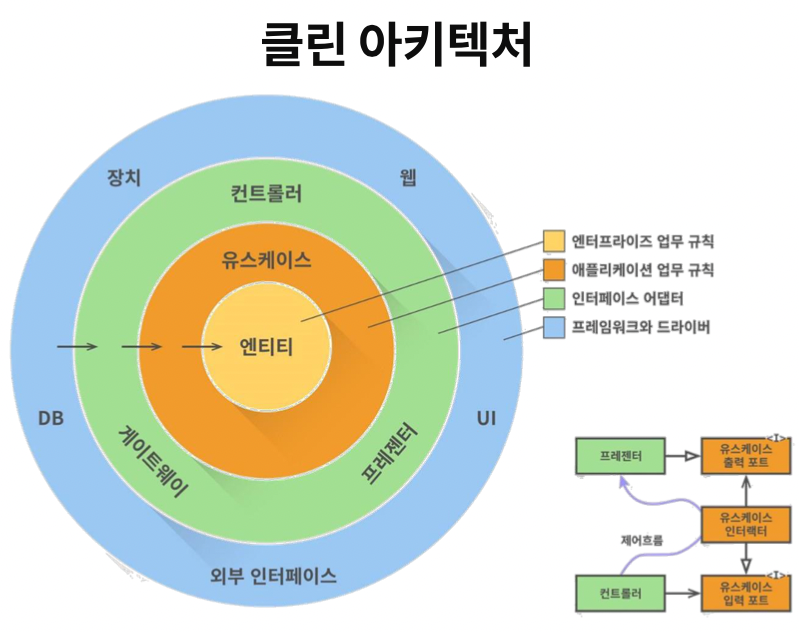
|  |
| --- |
| //스마트폰 객체  open class SmartPhone {      //통화 함수      fun call(number: String) {  *println*(**"**$number통화 연결**"**)      }      //문자 메시지 전송 함수      fun message(number: String, text: String) {  *println*(**"**$number **:** $text**"**)      }  }  //무선 충전 인터페이스  interface WirelessChargeable  {      fun wirelessCharge()  }  //생체 인식 인터페이스  interface Biometrics {      fun biometrics()  }  /./S20무선충전과 생체인식이 가능  class S20 : SmartPhone(), WirelessChargeable, Biometrics {      override fun wirelessCharge() {          //무선 충전 기능      }      override fun biometrics() {          //생체 인식 기능      }  }  //S2 스마트폰 기본 기능만 가능  class S2 : SmartPhone() {  //통화 및 메시지 송수신  } |

**[코드4 인터페이스 분리 원칙 예시]**

5의존성 역전 원칙(Dependency Inversion Principle)

-높은 수준의 코드는 낮은 수준의 세부사항 구현에 절대로 의존해서는 안되며, 그 반대여야 함

## 3. 모바일 클린 아키텍처



[그림 1.7] 클린 아키텍처

1 클린 아키텍처 원칙

1.1 단일 책임의 원칙

-하나의 모듈은 오직 하나의 액터에 대해서만 책임져야하는 단일 책임 원칙으로 경계선을 구분

-모바일 앱의 가장 큰 액터 : 이용자, 비지니스 로직 실행자 등

-액터의 세분화를 통해 책임을 쪼갤 수 있음 ex) 이용자 -> 회원가입을 원하는 이용자, 검색하는 이용자 등

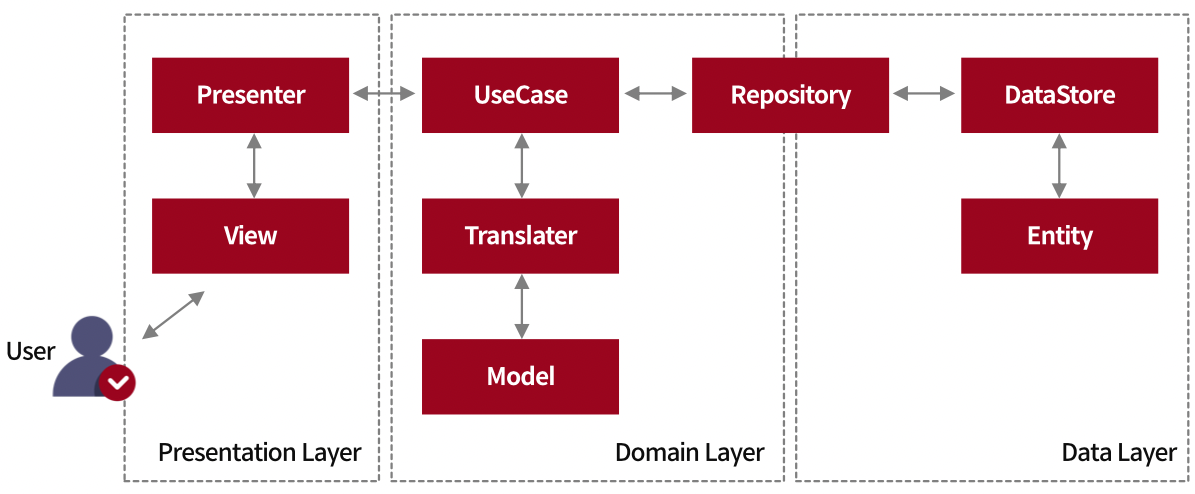
1.2 공동 폐쇄 원칙(Common Clousure Principle)

-CCP = SRP(단일 책임 원칙) + OCP(개방-폐쇄 원칙)

-각 계층은 하나의 큰 액터만 가짐

-가장 높은 수준의 계층은 그보다 하위 계층의 변화로 부터 보호되어야 함

2 클린 아키텍처 계층구조



[그림 1.8] 계층구조

2.1 UI 계층(UI Layer)

-화면의 표시, 애니메이션, 사용자 입력 처리 등 UI에 관한 모든 처리

2.1.1 뷰(View)

-직접적으로 플랫폼 의존적인 구현, 즉 UI화면 표시와 사용자 입력만 담당

-주의할 점은 View가 꼭 Activity/Fragment를 의미하지는 않음

2.1.2 프리젠터(Presenter, ViewModel)

-뷰와 달리 OS의 렌더링 API등에 직접적으로 의존하지 않음

-뷰 관점의 비즈니스 로직을 담당

2.2 도메인 계층(Domain Layer)

-유스케이스(Use Case) : 도메인 관점의 비즈니스 로직

-도메인 모델(Model) : 앱의 논리적인 엔티티 데이터

-트랜스레이터(Trarnslator): 데이터 계층의 엔티티, 도메인 모델을 변환하는 mapper의 역할

2.3 데이터 계층(Data Layer)

2.3.1리포지토리(Repository)

-유스케이스가 필요로 하는 데이터 저장/수정 등의 기능을 제공

-데이터 소스를 인터페이스 형태로 참조하기 때문에 이 클래스에서 데이터 소스 객체를 갈아끼우는 형태로, 외부 API 호출/로컬 DB 접근/Mock 객체 출력을 전환할 수 있음

2.3.2데이터 소스(Data source) : 실제 데이터의 입력이 여기서 실행

2.3.3엔티티(Entity)

-데이터 소스에서 사용되는 데이터를 정의한 모델(그림1.8과는 다른 개념)

-REST API의 요청/응답을 위한 JSON, 로컬 DB에 저장된 테이블을 표현하는 데이터 클래스 형태가 일반적

## 4. Google에서는 어떻게 좋은 설계를 만들어내는가

1. 모든 것은 문서에서 시작해서 문서로 끝난다.

1-Pager : 문제점과 내가 생각하는 방안 그리고 장단점을 군더더기 없이 한페이지로 정리

->TDR(Technical Design Document) 정제: 동료들의 피드백을 (+)(-)가 추가된 코멘트로 받고 리더급에서 LGTM(Looks Good To Me)가 나오면 다음 과정 진행

->Issue Tracker에 작업을 세분화 해서 등록

->코드 작성

->결과를 문서로 정리: 최종 결과물 설명과 개선된 결과를 수치로 표시. 이 문서는 인사 평가에서 그대로 첨부 자료로 사용됨

2. 코드 작성 전 문서의 중요성

-막연하게 “이런 게 좋지 않을까?” 와 글로 풀어내는 것과는 확연한 차이가 있음

-다른 엔지지어들의 의견을 듣고 더 발전 시킬 수 있음

3. 사용자 스토리의 중요성

-좋은 아키텍처란 비지니스에 도움이 되는 아키텍처로써 이를 측정하는 기준은 ‘사용자 경로(Critical User Journey)’

-모든 구글러에게 ‘사용자 경로’는 가장 중요한 개념

-실제 사용자에게 도움되지 않는 다면 어떤 종류의 설게 개선이라도 의미가 없음

4. TRD(Technical Design Document)의 중요 원칙: 두 번 설계하라

4.1 첫 설계 초안은 대부분 최적이 아닐 뿐 아니라 원래 해결하고자 했던 문제점을 해결하기에 적합하지 않은 경우도 많음

4.2 최소한 두 개 이상의 설계를 제안할 것. 그리고 가급적이면 각자를 극단적으로 다른 방향으로 만들어 보는 것을 추천

-> 반례를 통해 기존 설계의 장점이나 단점을 인식할 수 있음

4.3 각자의 장단점을 가중치와 함께 적을 것. 특히 인터페이스에 대해 평가할 것

-어떤 인테페이스가 더 심플한가?

-어떤 인터페이스가 일반적인 시나리오에 더 적합한가?

-어떤 인터페이스가 더 고수준의 구현을 쉽게 해주는가?

4.4 만들어 본 설계들이 다 만족스럽지 않다면 또 다른 접근 법을 고려해볼 것

4.5 두 번 설계하는 것은 똑똑한 사람들에게 더욱 어려움으로 인내할 것

5. 구글에서 TRD 리뷰

-누구라도 아키텍처에 대한 중요한 제안을 하고, 깊이 있는 기술 검토를 받을 수 있도록 만든 제도

-신청하면 보통 72시간 내에 리뷰 위원회가 열림

-그 사이에 기본적인 피드백이 TRD 문서 안에서 일어나고, 각 피드백에 대한 대응을 미리 해둬야 함

-참석자는 반드시 먼저 TRD를 숙지해야 함

-30분 시간 제한(전체 아키텍처에 큰 영향을 주는 경우에만 60분)

-관심 있는 누구라도 참석 가능. 단 적어도 한 명의 팀 리더(TL)가 있어야함

# 참고 링크

UML 기호 간단 정리

<https://blog.naver.com/PostView.naver?blogId=oh-mms&logNo=222117808952>

소프트웨어 개발 3대 원칙 : KISS, YAGNI, DRY

<https://blog.naver.com/complusblog/221163007357>

SOLID 원칙 정리 : 단일 책임 원칙, 개방 폐쇄 원칙

<https://merrily-code.tistory.com/201>

코드로 보는 객체지향 5가지 원칙

<https://blog.itcode.dev/posts/2021/08/15/liskov-subsitution-principle>