I.

모바일 아키텍처 개론

강 사 룡

여러 회사들을 컨설팅 해오면서 자주 발견하게 되는 것이 하나 있습니다. 바로 근본적으로 어떤 설계가 좋은 설계인가를 판단하는 원칙에 대한 합의가 개발팀 내에 없다는 점이었습니다. 구체적인 패턴으로 들어가기 전에 반드시 알아야만 하는 설계의 중요 원칙에 대해서 다뤄봅니다.

TABLE OF CONTENT (수정 필요).

SOLID 원칙

모바일 앱에서의 클린 아키텍처

부록:
Google에서는 어떻게 좋은 아키텍처를 만들어 내는가?

복잡성 제거:
좋은 설계를 위한 첫걸음 .

좋은 아키텍처란?

풀려는 문제에 잘 어울리는 설계

- 코드 구조 (예: 클래스)가 시스템이 어떻게 동작하는지를 잘 이해할 수 있는 보여준다
- 요구사항이 진화함에 따라 쉽게 변경 가능해야 한다

이런 게 쉬워야 한다

- 이해하기
- 왜 이렇게 되었는지 이유를 알기
- 유지하기
- 테스트하기

더 올바른 경향이 있다

성능 개선 작업을 더 부드럽게 해준다

좋은 아키텍처의 방해물: 복잡성

복잡성

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

복잡성 != 코드의 줄 수

징후

- **변경 증폭** (Change Amplification): 작은 변경 → 여러 곳의 많은 편집
- **인지적 부하** (Cognitive Load): 작은 변경 → 많은 선수 지식을 알아야 한다
- **알 수 없는 무지** (Unknown unknowns): 작은 변경 → 알 수 없는 결과들

복잡성을 조장하는 세 가지 요인

의존성 (Dependencies)

코드가 독립적으로 이해되고 수정될 수 없을 때

→ 없앨 수는 없으나, 최소화 되어야 함

불명확함 (Obscurity)

중요한 정보가 불명확할 때

→ 비용: 지금 10~20% 적게 듦, 일부 비용은 이자로 영원히 남음 (aka 기술 부채)

전술적 프로그래밍 (Tactical Programming)

빨리 완성하기

→ 비용: 10~20% 더 듦, 영구 무이자

복잡성을 낮추는 방법

불필요한 정보를 감춘다 (추상화)

- 깊은 모듈
- 범용 (general-purpose) 인터페이스
- 정보 은닉

깔끔한 추상화

- 복잡성을 아래로 끌어내리기
- 추상화 사이의 경계 찾기

깊은 모듈이란 무엇인가?

모듈	무언가를 구현하고 인터페이스를 제공하는 것 (예: 클래스, 퍼블릭 메소드)
추상화	중요하지 않은 세부사항을 감춰주는 간소화된 뷰
깊은 (deep) vs 얕은 (shallow)	하나의 간단한 요청으로 많은 작업이 완료된다 • 깊은 인터페이스의 예: file.open() • 얕은 것들: 래퍼 클래스, 나쁜 추상화들

범용 인터페이스는 더 깊다

범용인가, 특정용도인가?

범용은 지금 비용이 들고, 특정용도는 나중에 비용이 든다

모듈을 범용으로 만들려면?

- 인터페이스: 범용
- 구현: 현재 요구사항

판단을 위한 질문들

- 현재의 요구사항을 모두 만족하는 것 중에 무엇이 **가장 심플한** 인터페이스인가?
- 이 메소드가 **얼마나 많은 상황**에서 사용될 수 있는가?
  → 답이 하나라면 이는 적신호
- 이 **API**는 현재 나의 요구사항을 해결하는 데에 **실제로 사용하기 편한가?**

정보 은닉 vs 정보 유출

“

중요 원칙:
일반적인 케이스들을 최대한 심플하게 만들라

- 불필요한 정보를 감추기
- 정보 누출: 같은 지식이 여러 군데에서 사용된다
- 적신호: 과다 노출
  - 사용자가 일반적인 작업들을 위해서 명료하지 않은 기능들을 배워야 한다

복잡성을 아래로 끌어내리기

- 심플한 인터페이스는 심플한 구현보다 낫다
- 주의: 이를 남용하면 오히려 정보 유출로 이어질 수 있음
- 중요 원칙: 아래의 경우에 끌어내리기
  - 기존에 있는 기능과 밀접하게 연관되어 있다면
  - 앱의 여러부분들을 더 단순하게 만들어준다면
  - 클래스의 인터페이스를 단순하게 만들어 준다면

```
// bad
bufferedReader.getBufSize()
bufferedReader.adjustBufSize()
bufferedReader.readNext()

// good
bufferedReader.readNext()
```

추상화 사이의 경계 찾기

이럴 때는 합치기

- 정보가 공유된다
- 함께 있는 것이 인터페이스를 단순하게 만들어 준다
- 코드 중복을 없애준다
- 적신호: 코드 반복은 제대로 추상화가 이뤄지지 않았음을 의미할 수 있다

이럴 때는 나누기

- 특정 목적의 API가 범용 클래스 안에 있을 때
- 다른 종류의 범용 매커니즘이 함께 있을 때

그럼 재사용성은?

흔한 실수: 재사용성 확보를 DRY 원칙 차원에서만 바라봄

→ 이 경우, 성급하게 공통 요소를 부모 클래스로 만들 수 있음.
이 경우, 자식 클래스가 새롭게 만들어 질 때 이 부모의 공통 요소를 잘 이해하고 있어야 함.
(SOLID 원칙 중 리스코프 치환 원칙 참조) → 자식이 코드량 감소의 대가를 치르는 형국

**반대로, 중복 코드가 없는 한 하나의 클래스 / 메소드를
크게 만들어도 된다는 오해를 할 수 있음**

- 재사용성 != 하나의 기능 / 클래스가 앱의 여러 군데에서 재활용
- 현실적으로 앱 프로젝트 내에서 그런 경우가 얼마나 될까?
- 심지어 가장 많이 활용될 것 같은 유틸리티 클래스도 실제로는 활용되지 않는 경우가 많음 (멀티 모듈 참조)

재사용성은 독립적인 요소로 타 모듈에서 참조될 수 있음을 의미

SOLID 원칙.

SOLID 원칙이란?

S	Single Responsibility Principle	단일 책임 원칙
O	Open-Closed Principle	개방-폐쇄 원칙
L	Liskov Substitution Principle	리스코프 치환 원칙
I	Interface Segregation Principle	인터페이스 분리 원칙
D	Dependency Inversion Principle	의존성 역전 원칙

대전제

SOLID는 기본적으로 중간 규모(모듈)에 대한 원칙임을 기억하면
이해하기가 더 쉬움

목표: 중간 규모의 소프트웨어 구조가 아래와 같은 요구를 만족하도록..

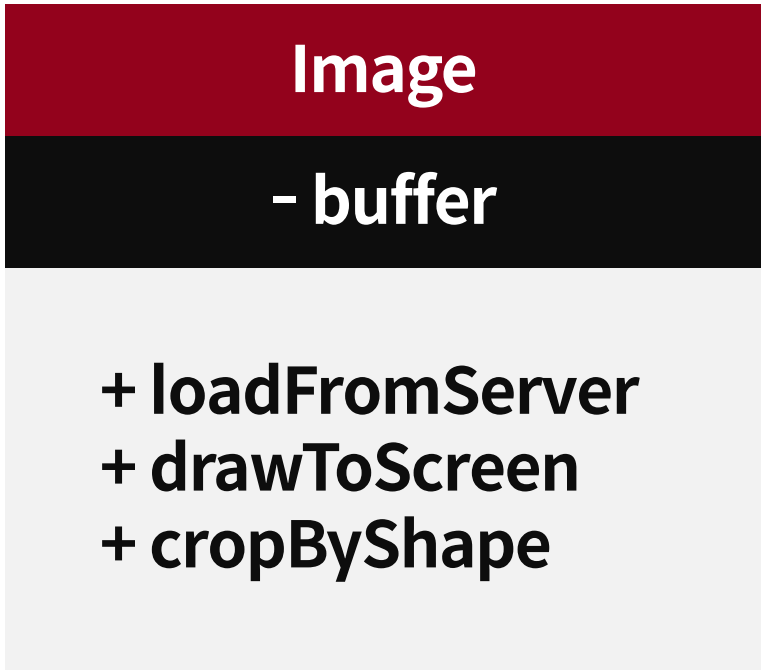
- **유지보수성**: 변경에 유연해야 한다
- **가독성**: 이해하기 쉬워야 한다
- **낮은 결합도(Loose couple), 높은 응집도(strong cohesion)**: 여러 소프트웨어 시스템에 사용될 수 있는 큰 컴포넌트의 기반이 되어야 한다

물론, 클래스 단위에서도 유용하게 사용됨

단일 책임 원칙 (Single Responsibility Principle)

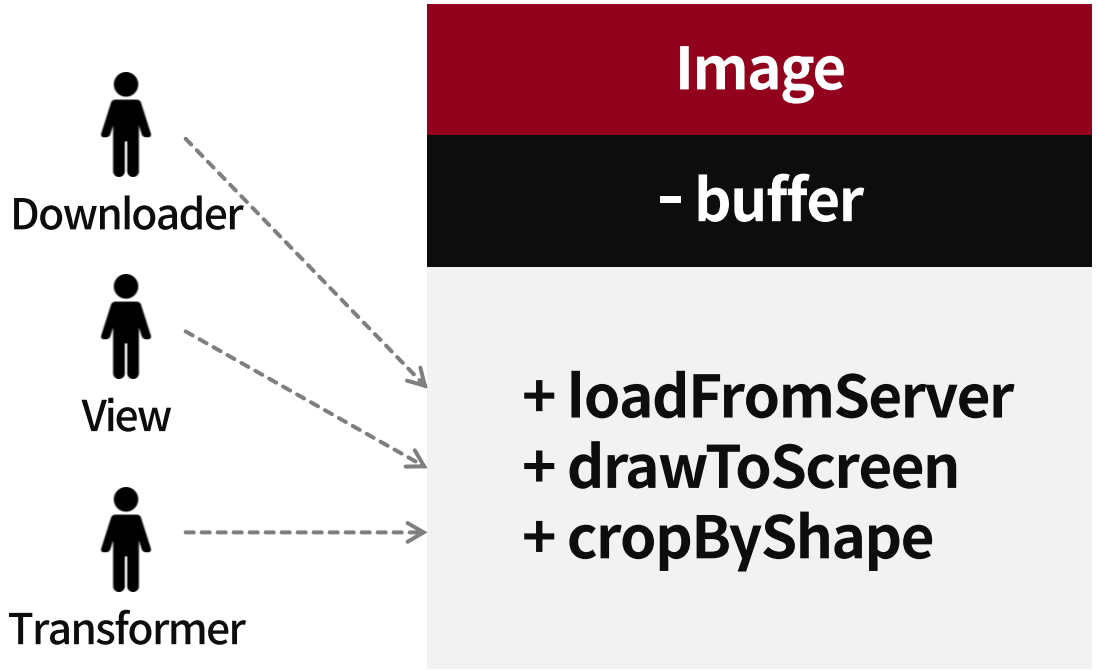
- 응집도(cohesion)에 대한 기준
- “각 소프트웨어 모듈은 변경의 이유가 단 하나여야만 한다”
  == “하나의 모듈은 오직 하나의 액터에 대해서만 책임져야 한다"
- 1974 년 다익스트라(Edsger W. Dijkstra)에 의해 제안된 개념인 관심사의 분리 (Separation of Concern)의 다른 표현
- 하나의 클래스가 하나의 일만 해야한다는 뜻이 아님에 유의 (메소드는 그래야 함)

이 클래스는 왜 잘못된 설계일까?



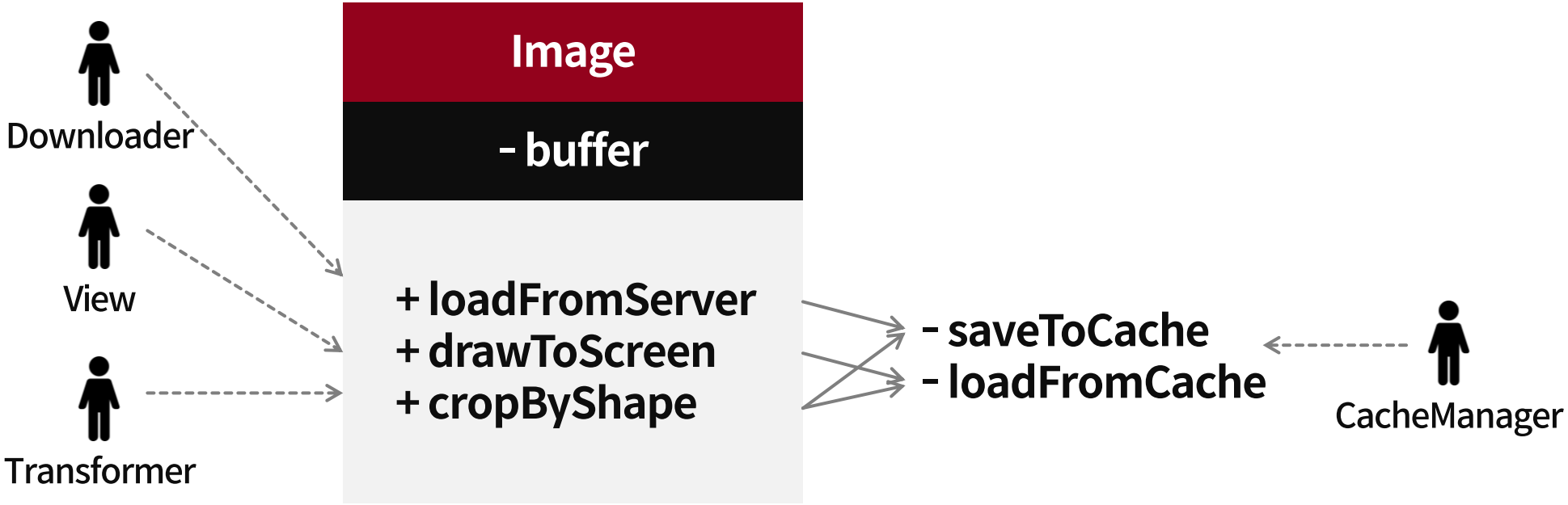
이 클래스는 왜 잘못된 설계일까?

> 세 개의 메소드가 서로 매우 다른 세 개의 액터를 책임지고 있기 때문

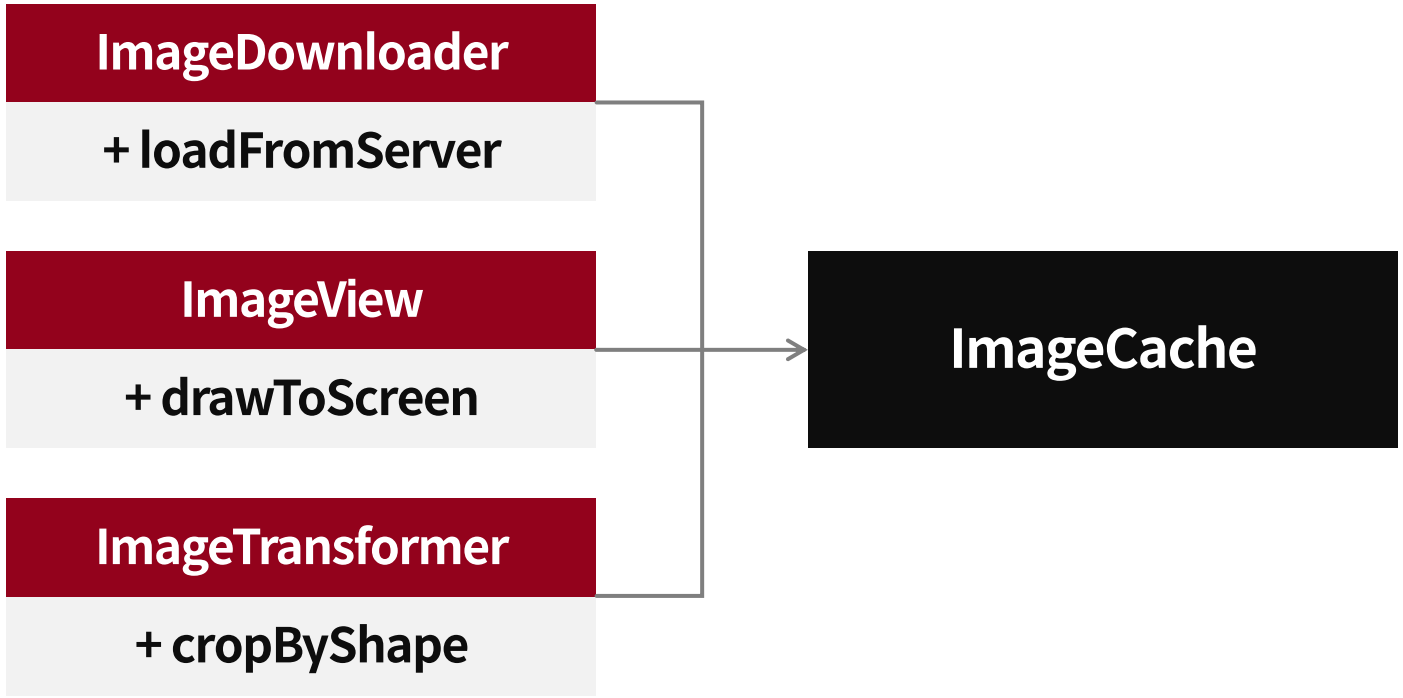


이 클래스는 왜 잘못된 설계일까?

- 실제로는 또 다른 액터가 더 있다면? 로컬 캐시 저장소
- 이 경우, 서버 로딩 로직 수정을 위해서 saveToCache 메소드를 수정하면, 같은 메소드를 호출하는 cropByShape 메소드까지 영향을 받을 수 있음

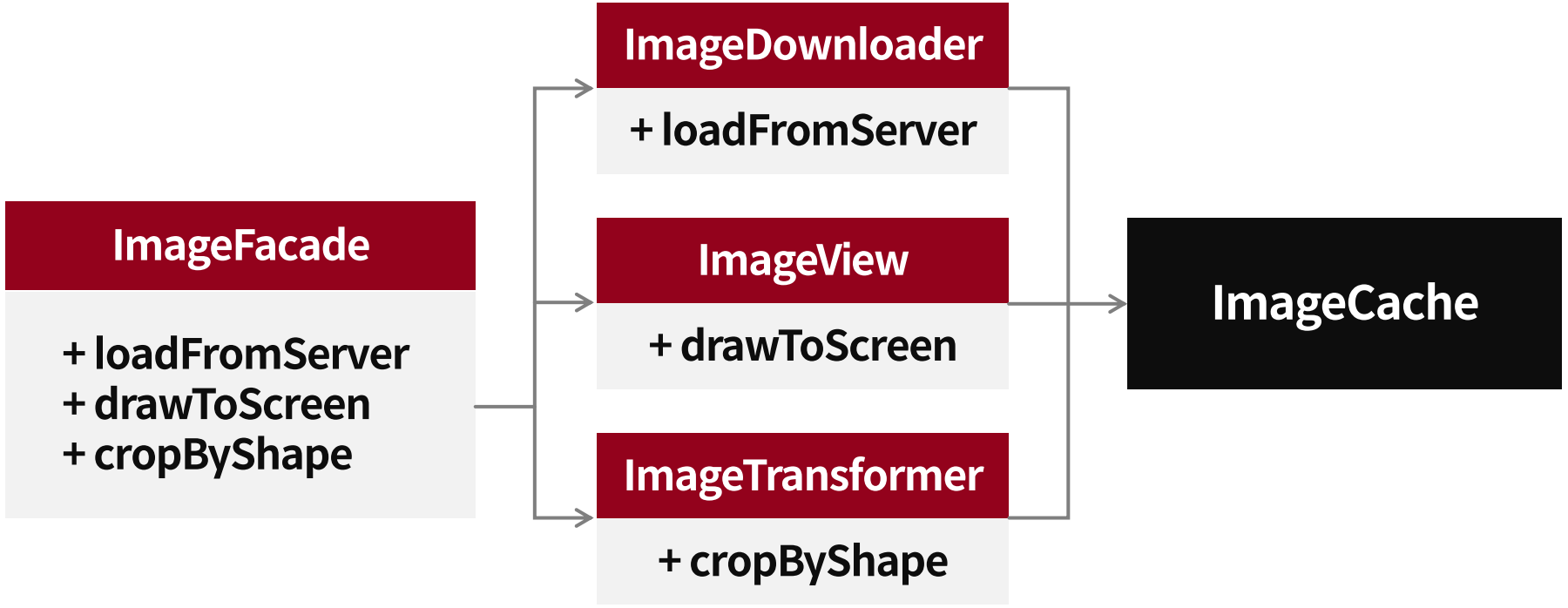


해결책: 세 개의 클래스로 분리하기



퍼사드(Facade) 패턴

사용자 입장에서 세 개의 클래스를 일일이 생성하는 것은 사용성이 떨어지므로



개방-폐쇄 원칙 (Open-Closed Principle)

1988년 버트란트 마이어(Bertrand Meyer)에 의해 제안된 원칙
- 원래는 라이브러리에 대한 규칙으로 제안

"소프트웨어 모듈은 확장에는 열려 있어야 하고, 변경에는 닫혀 있어야 한다"

"열려 있다"는 것은?	데이터 구조에 필드를 추가하거나, 함수에 새로운 요소를 추가하는 것이 가능해야 함
"닫혀 있다"는 것은?	내부 코드를 변경해도 이를 사용하는 외부 모듈에는 영향을 미치지 않아야 한다

클래스 단위의 개방-폐쇄 원칙

왜 모든 멤버변수는 private 혹은 protected 여야 하는가?

- 예를 들어, 앞에서 얘기했던 이미지 클래스가 이미지 데이터도 퍼블릭으로 노출하고 있다면?
- 내부 데이터의 변경이 클래스 사용자에게 의도하지 않게 영향을 미칠 수 있다

왜 글로벌 변수는 피해야 하는가?

이를 참조하는 모듈이나 클래스가 외부 값에 의존하게 되면 폐쇄 원칙이 깨진다. 예기치 못한 변경이 생길 수 있기 때문.

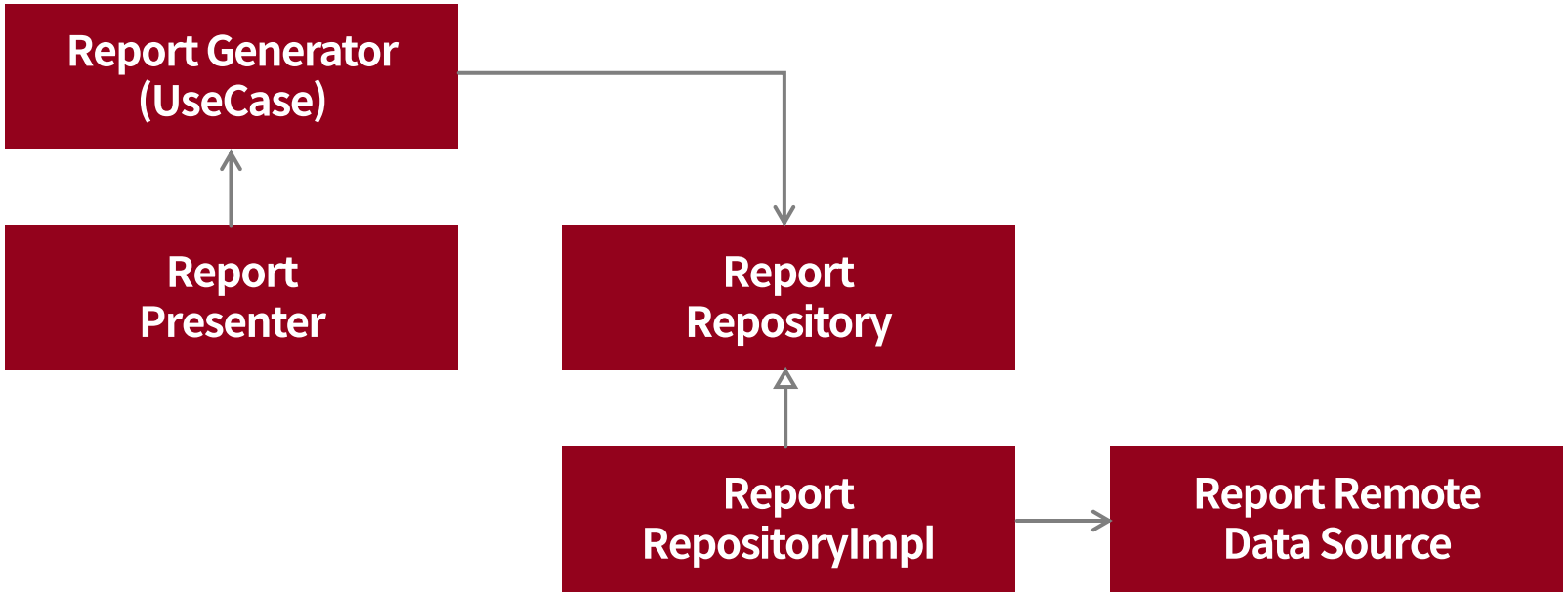
왜 추상 클래스 (abstract class / interface)를 굳이 따로 만드는 경우가 많은가?

사용자에게 필요한 것만 노출함으로 폐쇄 원칙을 만족하고, 구현 클래스의 내용을 감춤으로 수정도 용이해진다

모듈 단위의 개방-폐쇄 원칙

의존 관계의 방향이, 보호하려는 컴포넌트를 향하도록 그려진다
⇒ 결과적으로 높은 수준의 클래스/모듈이 하위 레벨의 변경으로부터 보호됨

예: 서버로부터 매출 정보를 받아와서 분석 보고서를 화면에 그려주는 앱



리스코프 치환 원칙 (Liskov Substitution Principle)

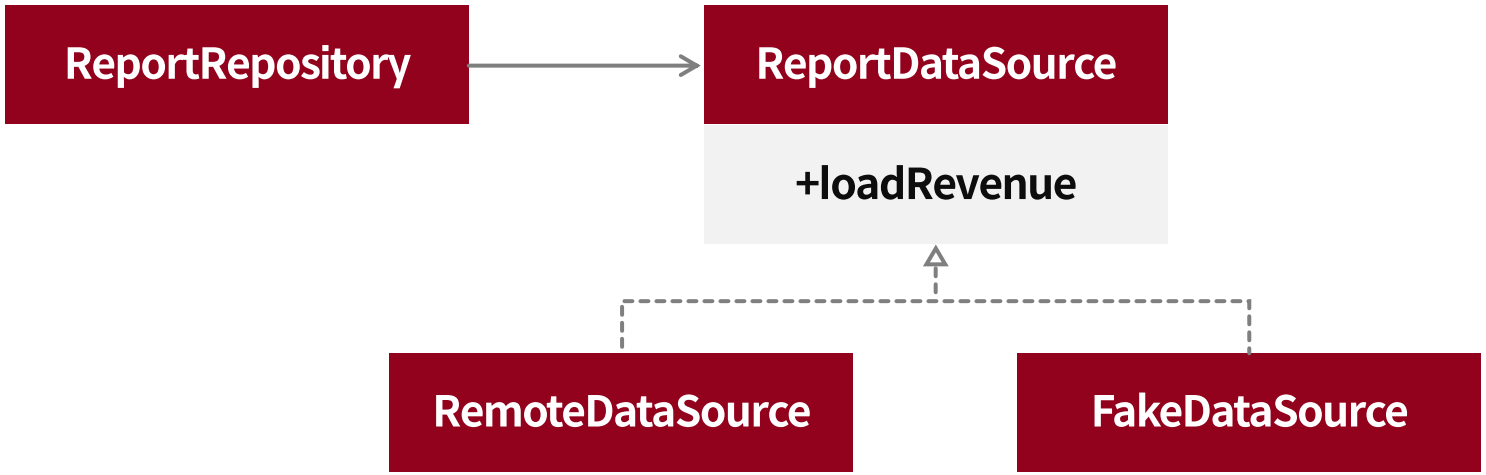
1988년
바바라 리스코프(Barbara Liskov)가
제안한 원칙

> 대체 가능한 컴포넌트들을
> 이용해 시스템을 만들 수 있으려면,
> 이들의 서브타입들은 반드시
> 서로 치환 가능해야 한다

리스코프 치환 원칙의 좋은 예

Repository의 행위가 DataSource의 자식들 중 어떤 것을 사용해도 동일한 역할을 함

→ 부모에서 정한 행위의 원칙을 자식에서 그대로 지키고 있기 때문에
행위 상속이라고도 부름



리스코프 치환 원칙을 위배하는 매우 일반적인 예

상속의 어려움

Rectangle이 area() 함수만을 갖고 있었으면
애초에 생기지 않았을 문제라고 생각할 수도 있겠지만,
처음에는 정사각형 같은 곳에 사용되리라고 전혀 예상하지 못했을 수도 있음.

```
val r: Rectangle = Square(...)
r.width = 5
r.height = 2
check(r.area() == 10) // Throws an IllegalStateException!
```

리스코프 치환 원칙의 또다른 교훈

왜 상속보다 조합(composition over inheritance)인가?

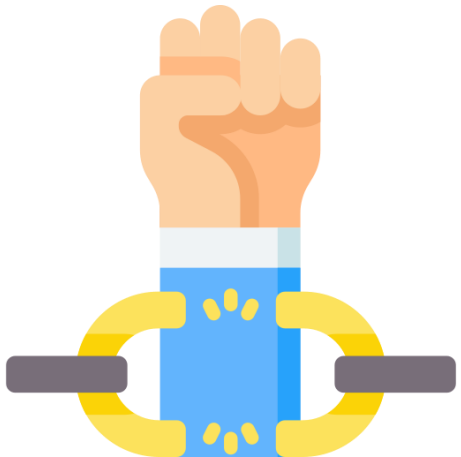
→ **상속으로 인한 부작용.**
자식은 부모가 갖고 있는 행위를 정확히 이해하고,
이 행위가 깨지지 않도록 할 의무가 있음

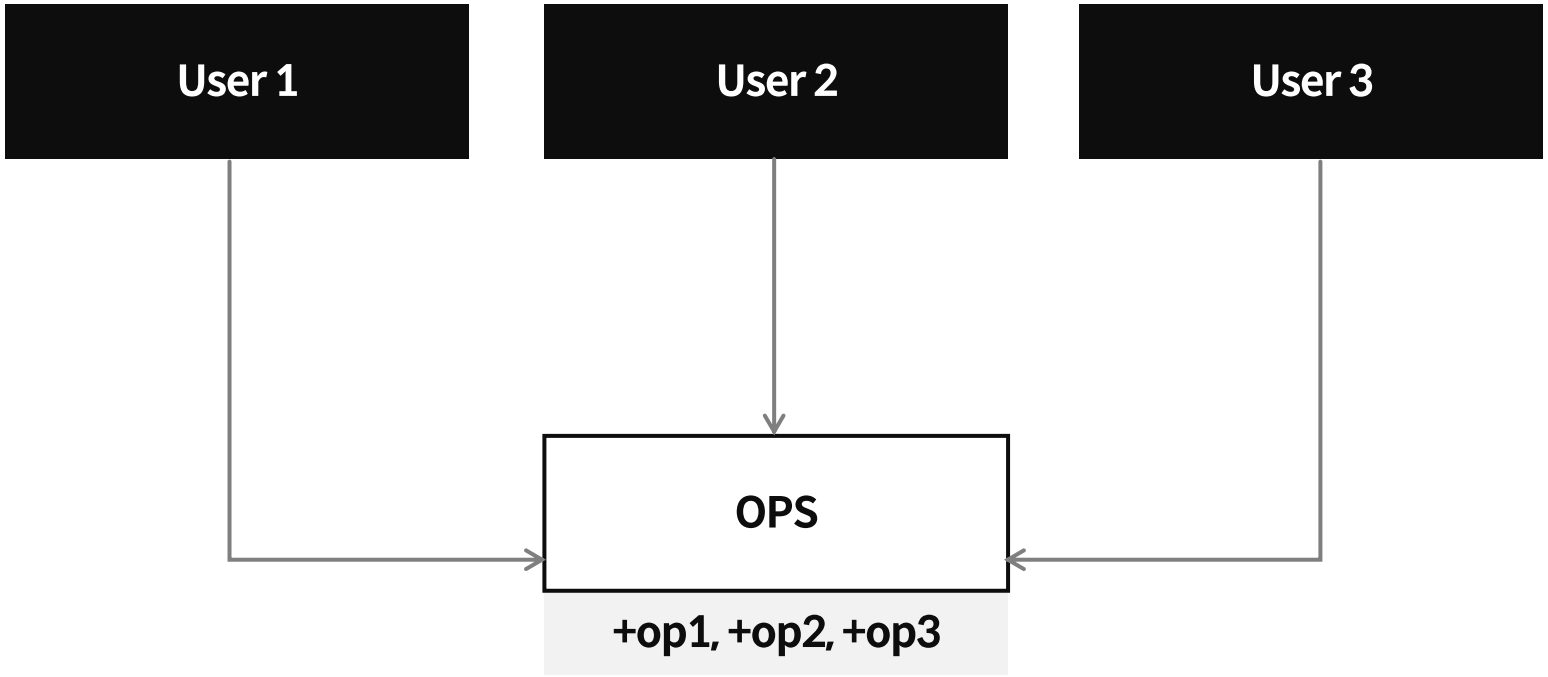
(여담이지만,) Jetpack Compose가 탄생한 배경에는 이런 이유도 있음

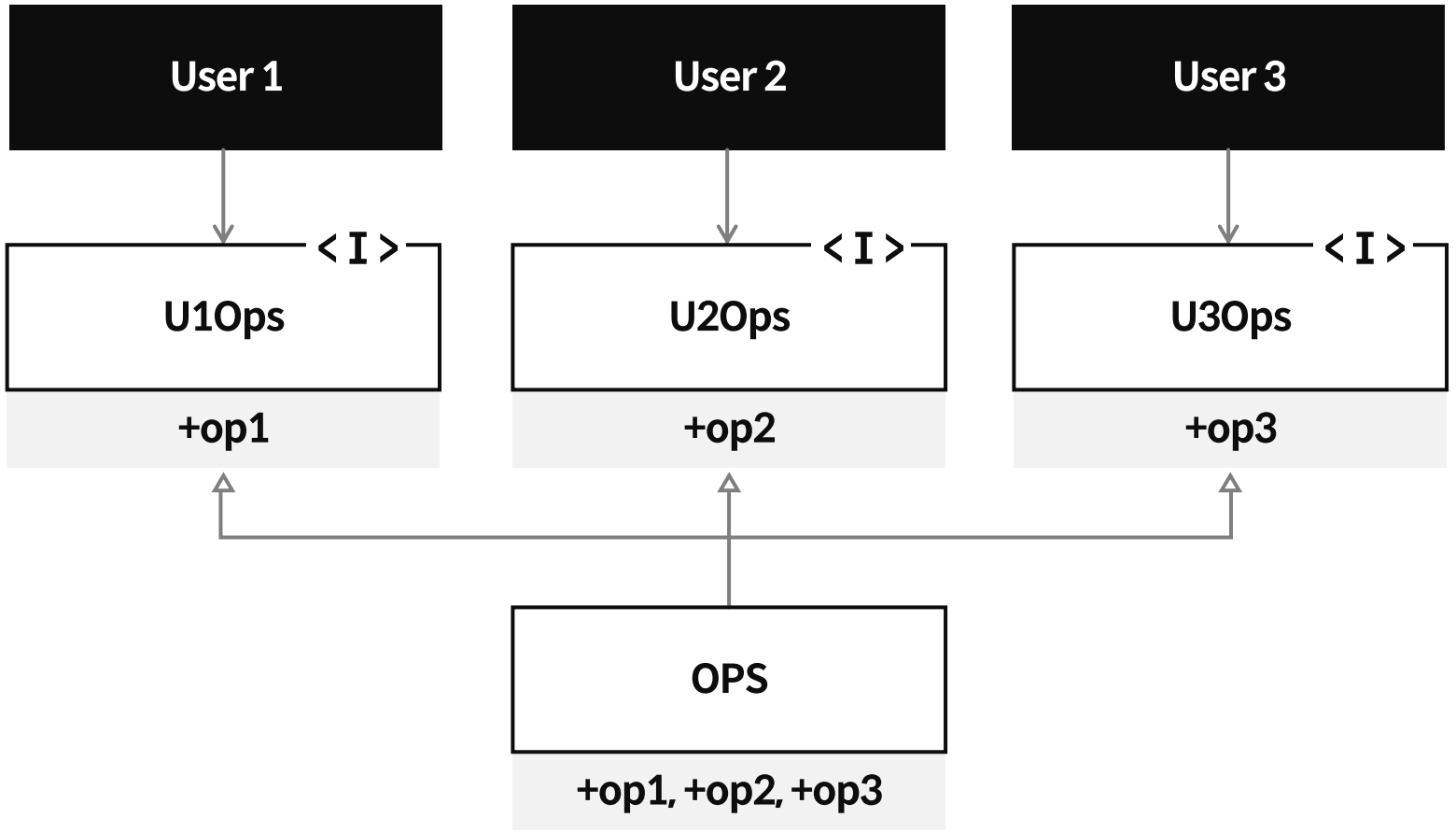
인터페이스 분리 원칙
(Interface Segregation Principle)

“

각 소프트웨어 모듈은
자신이 사용하지 않는 것에 의존하지 않아야 한다

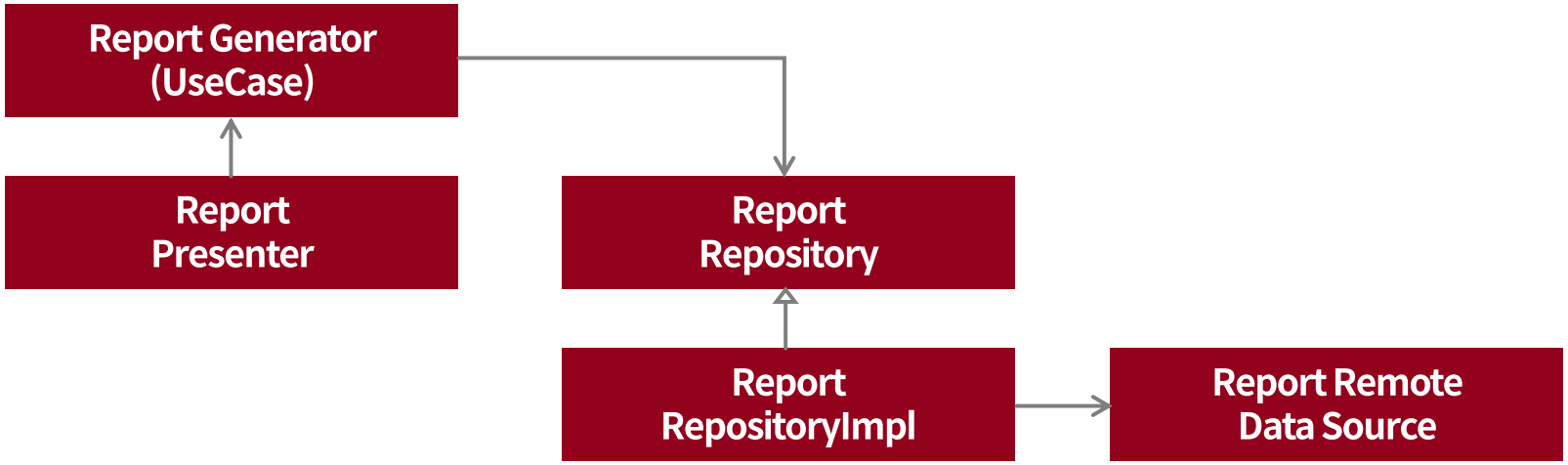



User 1
User 2
User 3
OPS
+op1, +op2, +op3


User 1
User 2
User 3
<I>
U1Ops
<I>
U2Ops
<I>
U3Ops
+op1
+op2
+op3
OPS
+op1, +op2, +op3

모듈 단위의 인터페이스 분리 원칙

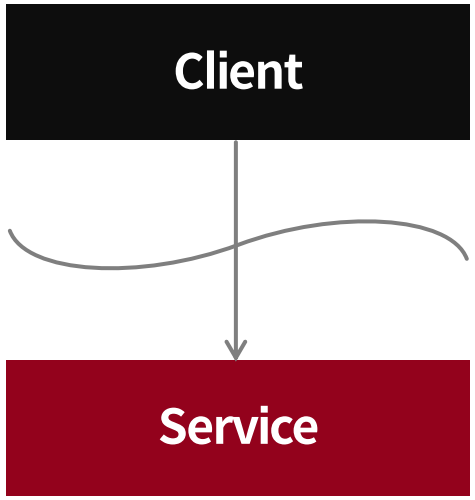
외부 모듈에 필요한 “인터페이스”만 노출한다

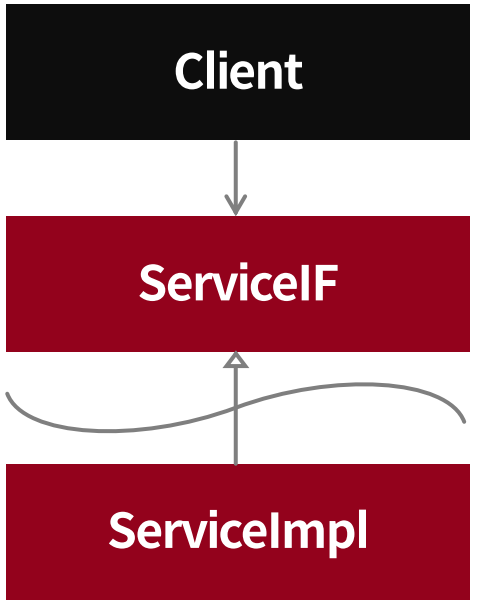


Q. 왜 인터페이스가 필요한가?
→ transitive dependency(추이 종속성)를 막기 위해서.
즉, Use Case가 DataStore에 종속성이 생기는 걸 막아줌

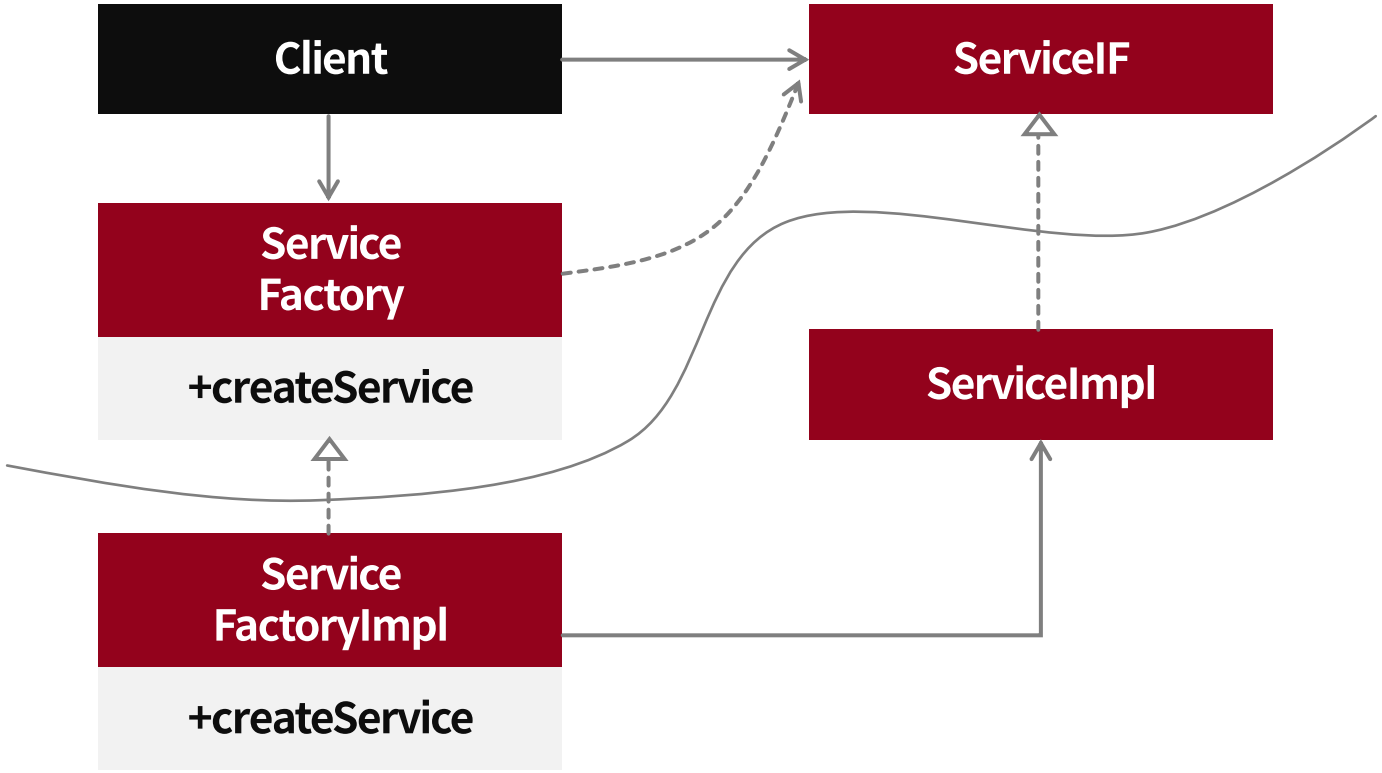
의존성 역전 원칙 (Dependency Inversion Principle)

높은 수준의 코드는 낮은 수준의 세부사항 구현에 절대로 의존해서는 안 되며,
그 반대여야 한다.





여기에 추상 팩토리 패턴을 추가하면 매우 유용



Mobile Clean Architecture.

“

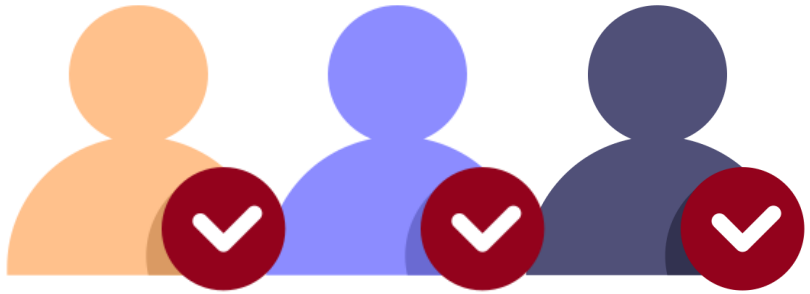
소프트웨어 아키텍처는 선을 긋는 기술이며,
나는 이러한 선을 경계(boundary)라고 부른다.
경계는 소프트웨어 요소를 서로 분리하고, 경계 한편에
있는 요소가 반대편에 있는 요소를 알지 못하도록 막는다.

- 로버트 C. 마틴, 클린 아키텍처 중에서

경계선의 구분

“
단일 책임 원칙

- 모바일 앱의 가장 큰 액터: 이용자, 비즈니스 로직 실행자, …
- 액터의 세분화를 통해 책임을 쪼갤 수 있음.
  예: 이용자 -> 회원가입을 원하는 이용자, 검색하는 이용자, …



공통 폐쇄 원칙 (Common Closure Principle)

“

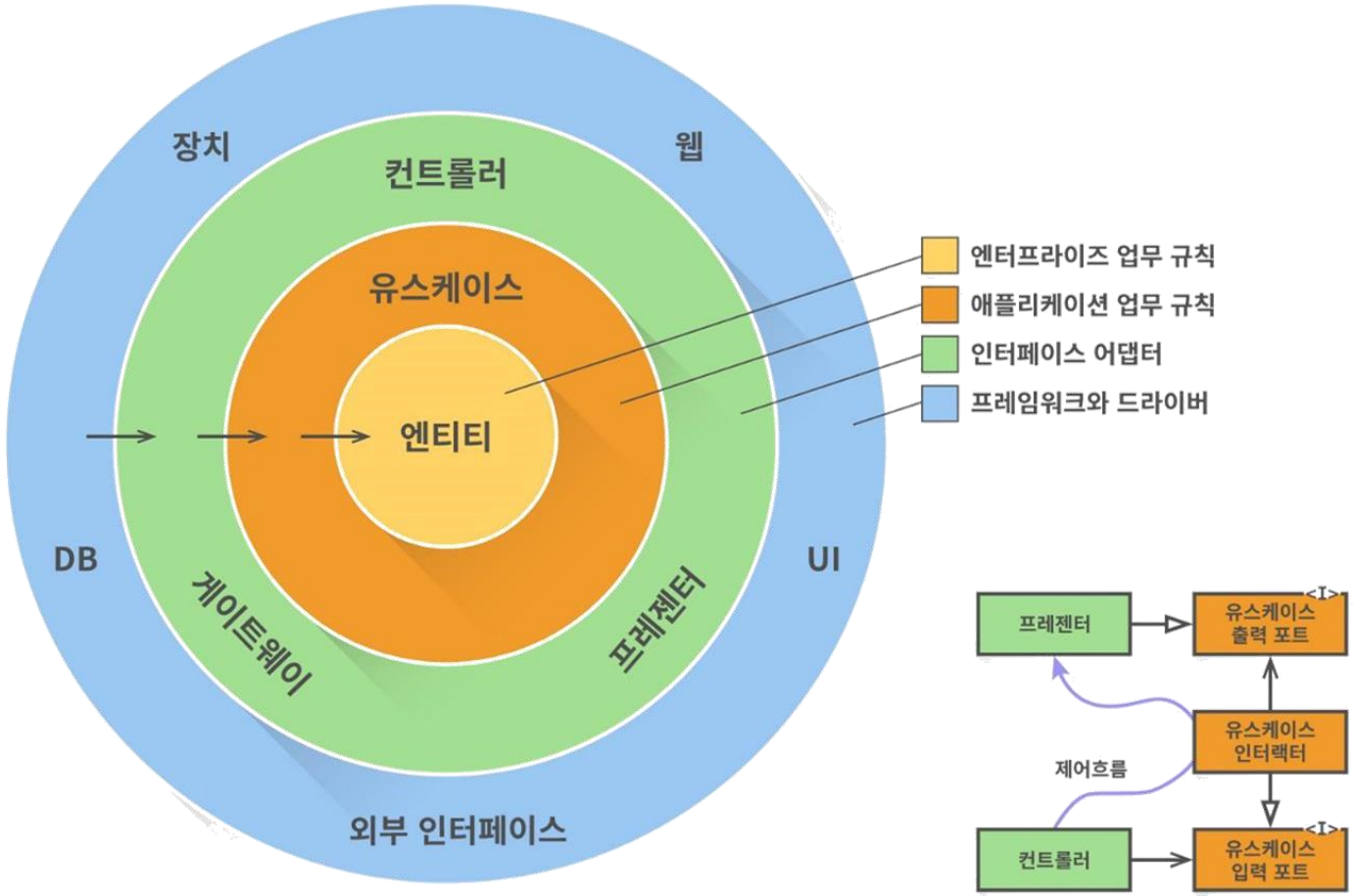
CCP = SRP(단일 책임 원칙) + OCP(개방-폐쇄 원칙)

- 각 계층은 하나의 (큰) 액터만을 가짐
- 가장 높은 수준의 계층은 그보다 하위의 계층의 변화로부터 보호되어야 함

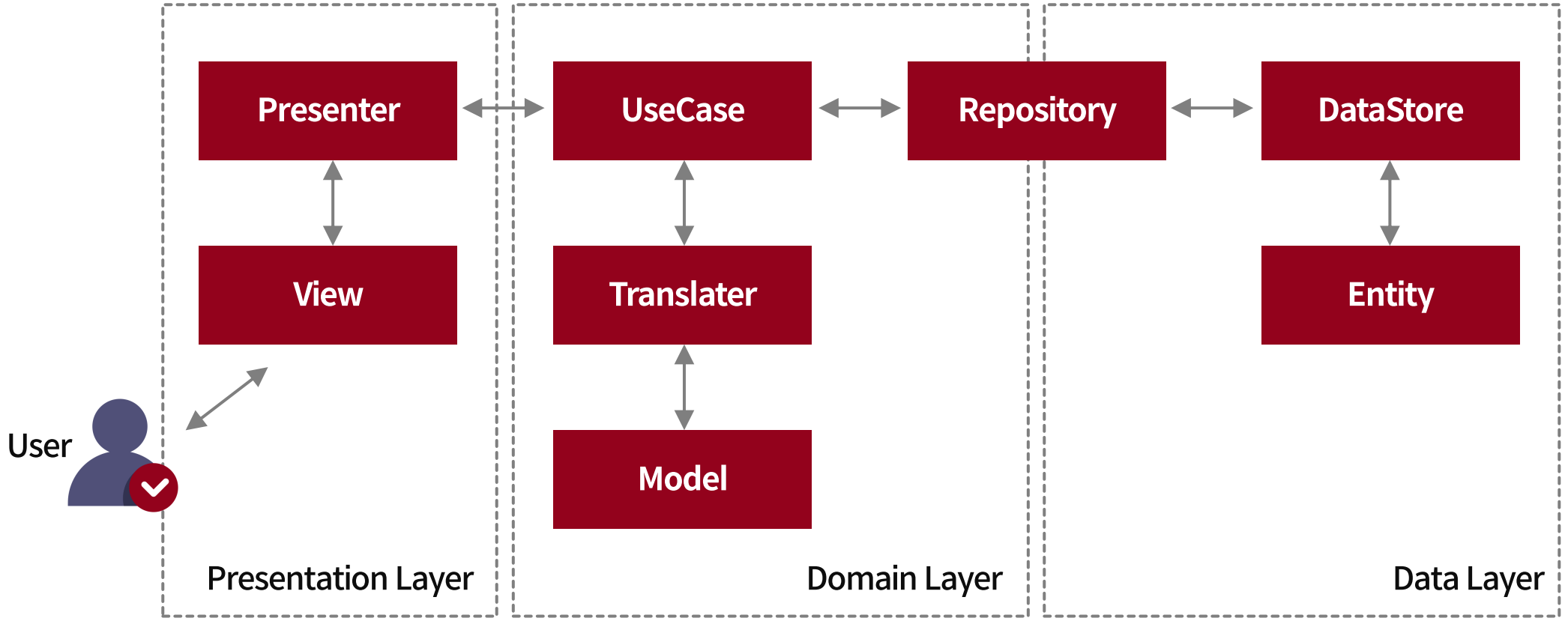
도메인 계층(그리고 유스케이스)의 지위

> 가장 높은 수준의 계층

클린 아키텍처



출처: 클린 아키텍처, 로버트 C 마틴, 도서출판 인사이트


Presenter
UseCase
Repository
DataStore
View
Translater
Entity
Model
User
Presentation Layer
Domain Layer
Data Layer

UI계층

화면의 표시, 애니메이션, 사용자 입력 처리 등 UI에 관련된 모든 처리

뷰(View)

- 직접적으로 플랫폼 의존적인 구현, 즉 UI 화면 표시와 사용자 입력만 담당
- 주의할 점은 View가 꼭 Activity/Fragment를 의미하지는 않는다는 점

프리젠터(Presenter, 혹은 ViewModel)

- 뷰와는 달리 OS의 렌더링 API 등에 직접적으로 의존하지 않음
- 뷰 관점의 비즈니스 로직을 담당

도메인 계층 (Domain Layer)

유스케이스(Use Case)

→ 도메인 관점의 비즈니스 로직

도메인 모델(Model)

→ 앱의 논리적인 엔티티 데이터

트랜스레이터(Translator)

→ 데이터 계층의 엔티티 - 도메인 모델을 변환하는 mapper의 역할

데이터 계층 (Data Layer)

리포지토리(Repository)

- 유스케이스가 필요로 하는 데이터 저장/수정 등의 기능을 제공.
- 데이터 소스를 인터페이스 형태로 참조하기 때문에 이 클래스에서 데이터소스 객체를 갈아끼우는 형태로, 외부 API 호출/로컬 DB 접근/mock object 출력을 전환할 수 있음.

데이터 소스(Data Source)

→ 실제 데이터의 입출력이 여기서 실행

엔티티(Entity)

- 데이터 소스에서 사용되는 데이터를 정의한 모델 (앞서 그림의 엔티티와는 다른 개념)
- REST API의 요청/응답을 위한 JSON, 로컬 DB에 저장된 테이블을 표현하는 data class 형태가 일반적

부록:
Google에서는 어떻게 좋은 아키텍처를 만들어 내는가?

모든 것은 문서에서 시작해서 문서로 끝난다

1-pager

- 군더더기 없이 문제점
- 내가 생각하는 방안1/2/3
- 각 방안의 장단점
- 그리고 feedback! → LGTM (Looks Good To Me)

TDR (Technical Design Document) 또 피드백 → LGTM (Looks Good To Me)
등으로 정제

Issue Tracker에 작업을 세분화 해서 등록

코드!

결과를 문서로 정리

- 최종 결과물 설명, 그리고 개선된 결과를 수치로 표시
- 인사 평가에서 그대로 첨부 자료(artifact)로 사용됨

문서의 중요성

막연하게 “이런 게 좋지 않을까?"와 글로 풀어내는 것과는 확연한 차이

다른 훌륭한 Google 엔지니어들의 의견을 듣고 더 발전시킬 수 있다

Google의 영향으로 테크 스펙 등의 이름으로 실리콘 밸리에서 일반적으로 사용됨

사용자 스토리의 중요성

모든 Googler에게 중요 사용자 경로(Critical User Journey)는 가장 중요한 개념

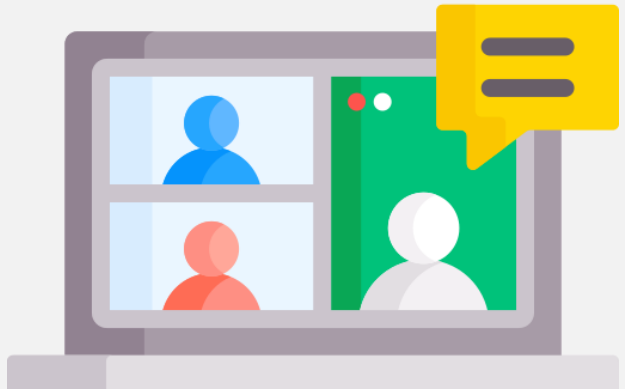
실제 사용자에게 helpful 하지 않다면 어떤 종류의 설계 개선이라도 의미가 없음

TRD의 중요 원칙: 두 번 설계하라!

- 첫 설계 초안은 대부분 최적이 아닐 뿐더러 원래 해결하고자 했던 문제점을 해결하기에 적합하기조차 못한 경우가 많음
- 최소한 두 개 이상의 설계를 제안할 것. 그리고 가급적이면 각자를 극단적으로 다른 방향으로 만들어 볼 것을 추천
- 각자의 장점과 단점을 (가중치와 함께) 적을 것. 특히 인터페이스에 대해 평가할 것
  - ✓ 어떤 인터페이스가 더 심플한가?
  - ✓ 어떤 인터페이스가 일반적인 시나리오에 더 적합한가?
  - ✓ 어떤 인터페이스가 더 고수준의 구현을 쉽게 해주는가?
- 만들어 본 설계들이 다 만족스럽지 않다면 또 다른 접근법을 고려해볼 것
- 중요: 두 번 설계하는 것은 똑똑한 사람들에게 더욱 어려움. 인내할 것!

TRD Review

- ✓ 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 TL이 있어야 함



수고하셨습니다!