

Toby's Clean Spring

**Domain Model Pattern and
Hexagonal Architecture**

토비의 클린 스프링

도메인 모델 패턴과 헥사고날 아키텍처

Part 1

강사 소개

Toby 이일민

토비의 스프링 3.0, 3.1 저자

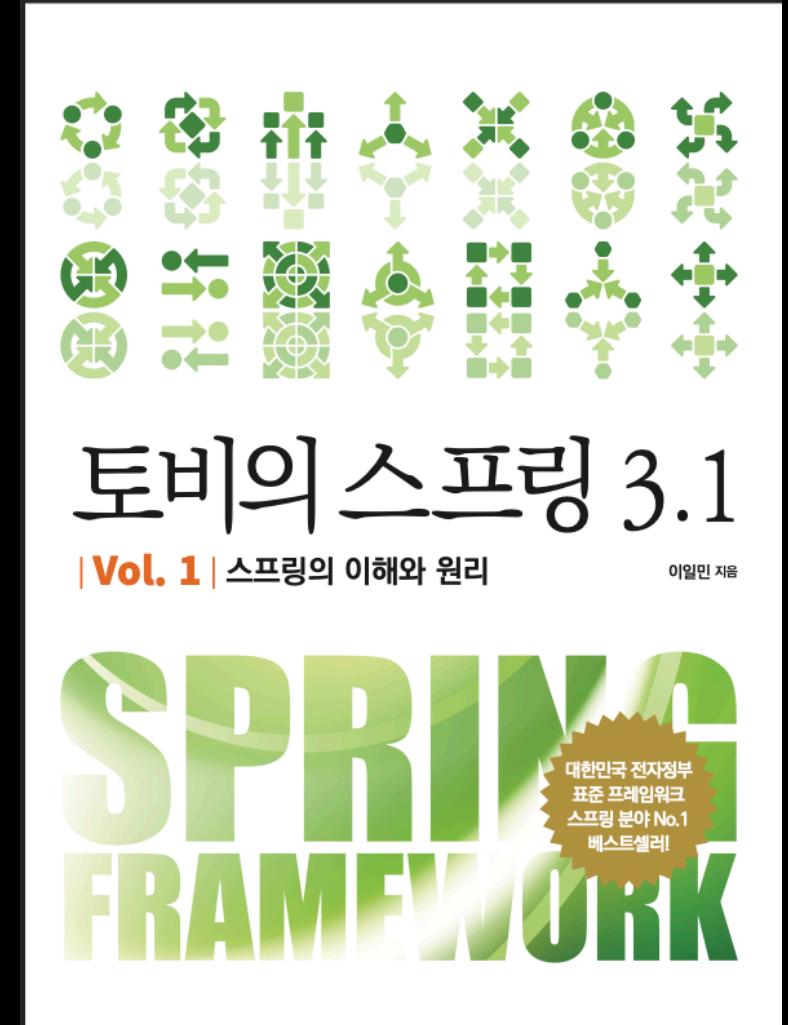
한국스프링사용자모임(KSUG) 설립자

스프링 기반의 시스템 개발 20년 경력

유튜브 기술 채널 (www.youtube.com/@tobyspring)

디스코드 서버 (<https://discord.gg/x4eT5HYk6X>)

이메일: tobyilee@gmail.com



강의 소개

클린 스프링 강의 시리즈의 첫 번째 주제인 도메인 모델 패턴과 헥사고날 아키텍처의 Part 1입니다

- **스프런(Splearn)**이라는 가상의 스타트업 개발팀이 온라인 교육 서비스를 개발하고 운영하는 과정의 시나리오를 바탕으로 진행됩니다
- 여러분은 토비와 함께 이 개발팀의 일원이 되어, 스타트업 서비스가 처음 만들어지고 성장하는 과정에서 각 단계별로 어떤 것을 고려하고 결정하며 빠르게 적용해 나가는지 체험할 수 있습니다
- 주어진 상황에서 최선의 선택과 트레이드오프(trade-off)가 어떻게 이루어지는지 함께 고민하고 경험하는 것이 목표입니다

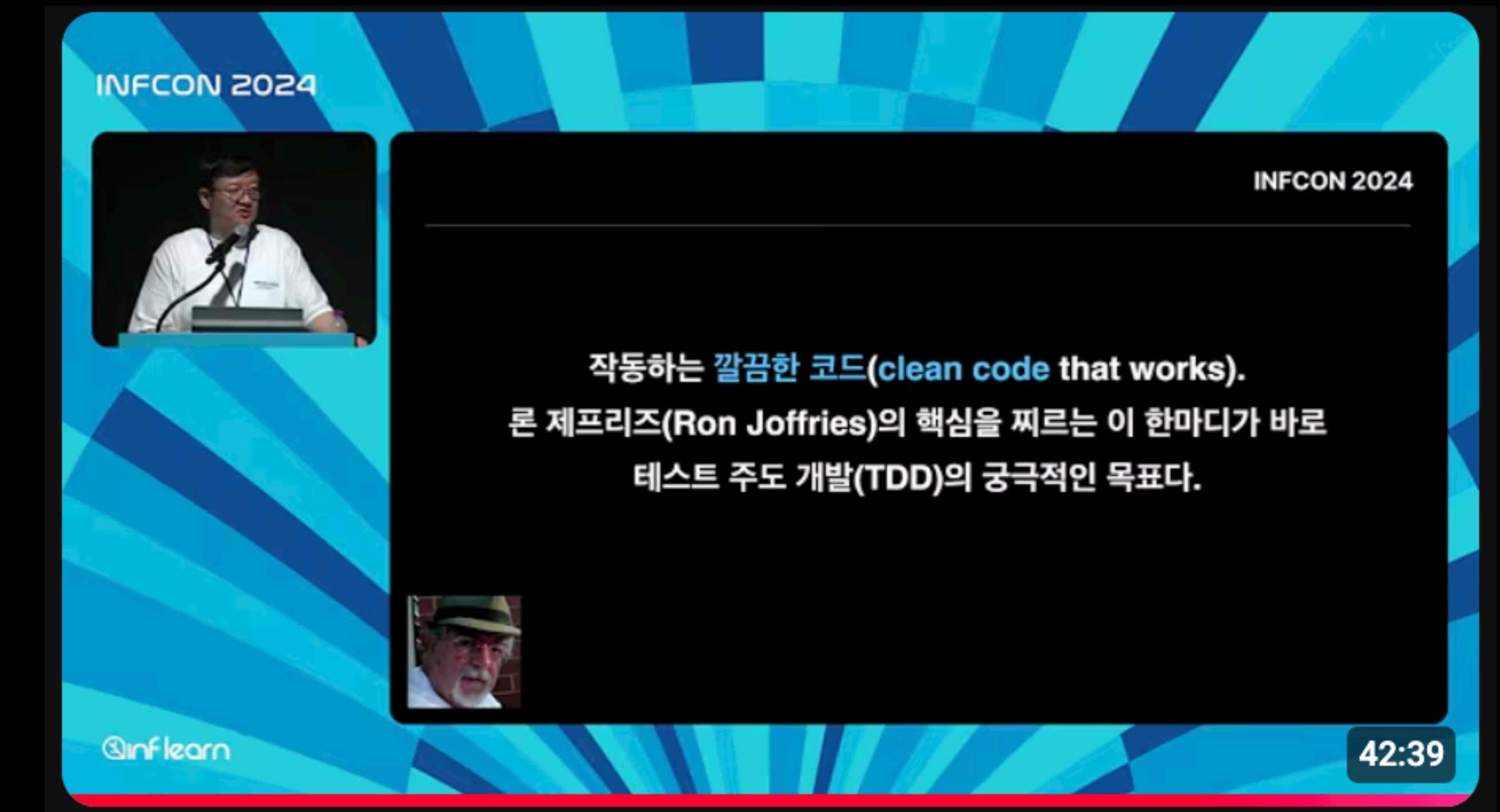
도메인 모델 패턴과 헥사고날 아키텍처

- 도메인 모델을 만들고 발전시키면서 모델이 주도하는 설계와 구현 전략을 사용합니다
- 핵심 기능을 담은 헥사고날 아키텍처 애플리케이션을 중심으로 개발하고 테스트와 운영 환경에 자유롭게 적용하는 방법을 설명합니다
- 모든 작성하는 코드에 대한 테스트를 만들면서 진행합니다

클린 스프링

클린 코드(Clean Code That Works)가 추구하는 유지보수성과 이를 통한 생산성을
스프링 애플리케이션 개발에 적용하는 원칙과 패턴, 그리고 실천법

- 인프콘 2024 - 클린 스프링(<https://www.youtube.com/watch?v=d3krJ4el8Hg>)
- 인프런 토비의 클린 스프링 로드맵(<https://www.inflearn.com/roadmaps/4858>)



클린 스프링의 선순환

리팩터링

유지보수성이 좋은 코드는 변경가능성이 좋다

빠르게 변경되는 코드는 개발 생산성이 좋다

도메인 모델 패턴과 헥사고날 아키텍처 - 파트 1과 파트 2

- **파트 1**

- 회원 기능 개발을 중심으로 핵심 개발 전략과 원칙, 실천방법을 설명
- 도메인 모델을 만들고 모델이 중심이 되는 개발 진행
- 도메인 모델 패턴을 이용해서 도메인 코드 작성
- 헥사고날 아키텍처로 응집도가 높고 적응성이 뛰어나며 테스트 용이성을 가진 애플리케이션 개발
- JPA 기술을 효율적으로 활용하는 방법과 애그리거트 기반의 모듈 분리
- 안전한 리팩터링을 가능하게 하는 테스트 코드 작성

- **파트 2**

- 초기 도메인 모델에서 분석한 주요 개념에 대한 애플리케이션 구현
- 도메인 모델을 발전시키면서 더 나은 설계를 발견하고 적용
- 복잡한 도메인 로직을 도메인 모델 패턴을 이용해서 정확하고 빠르게 개발
- 다양한 리팩터링 활용과 테스트 코드 지원 기능 개발

학습 방법

- 비록 토비 혼자서 진행하고 코딩하는 영상을 담은 강의이지만 여러분도 같은 팀으로 개발을 함께 한
다고 생각해주세요
- 강의 중간에 질문이 나오거나, 여러 선택지를 가지고 고민하는 부분이 있을 때 잠시 멈추고 나라면
어떤 것을 어떤 이유로 선택할지 먼저 생각해보세요
- 강의에서 선택한 방법과 작성한 코드를 다르게 접근해보고 싶다고 생각되면 예제를 만들면서 자신
만의 방식으로 접근해보고, 시간이 지나면서 각각 코드가 어떻게 변화하는지, 그때의 선택이 어떤
영향이 있는지 생각해보세요
- 여러분들의 의견이나 반론, 아이디어 등은 수강평, Q&A, 디스코드 채널 등에 남겨주시면 클린 스프
링 시리즈의 다음 강의를 제작할 때 참고하겠습니다
- 처음에는 강의를 보기만 해도 좋습니다. 그런 다음에라도 꼭 예제를 직접 작성해보세요

개발 환경 구성

Development Environment Setup

개발 환경 구성

- JDK
- IDE / IntelliJ
- HTTPie
- MySQL
- Docker

AI 개발 지원 도구

- GitHub Copilot (IntelliJ Plugin, github.com)
- ChatGPT
- Claude
- DeepSeek
- Perplexity
- Cursor, VSCode+Copilot

새로운 기술 사용법을 익히거나 코딩 연습을 할 때는 AI를 꺼두세요

Splearn 프로젝트 생성

Spring Initializr

스프링 프로젝트를 생성하는 방법

- Spring Boot Starter로 구성된 프로젝트로 생성
- Spring Initializr를 이용
 - IntelliJ Ultimate의 New Project - Spring Boot
 - start.spring.io 웹사이트
 - Spring CLI

Splearn 프로젝트

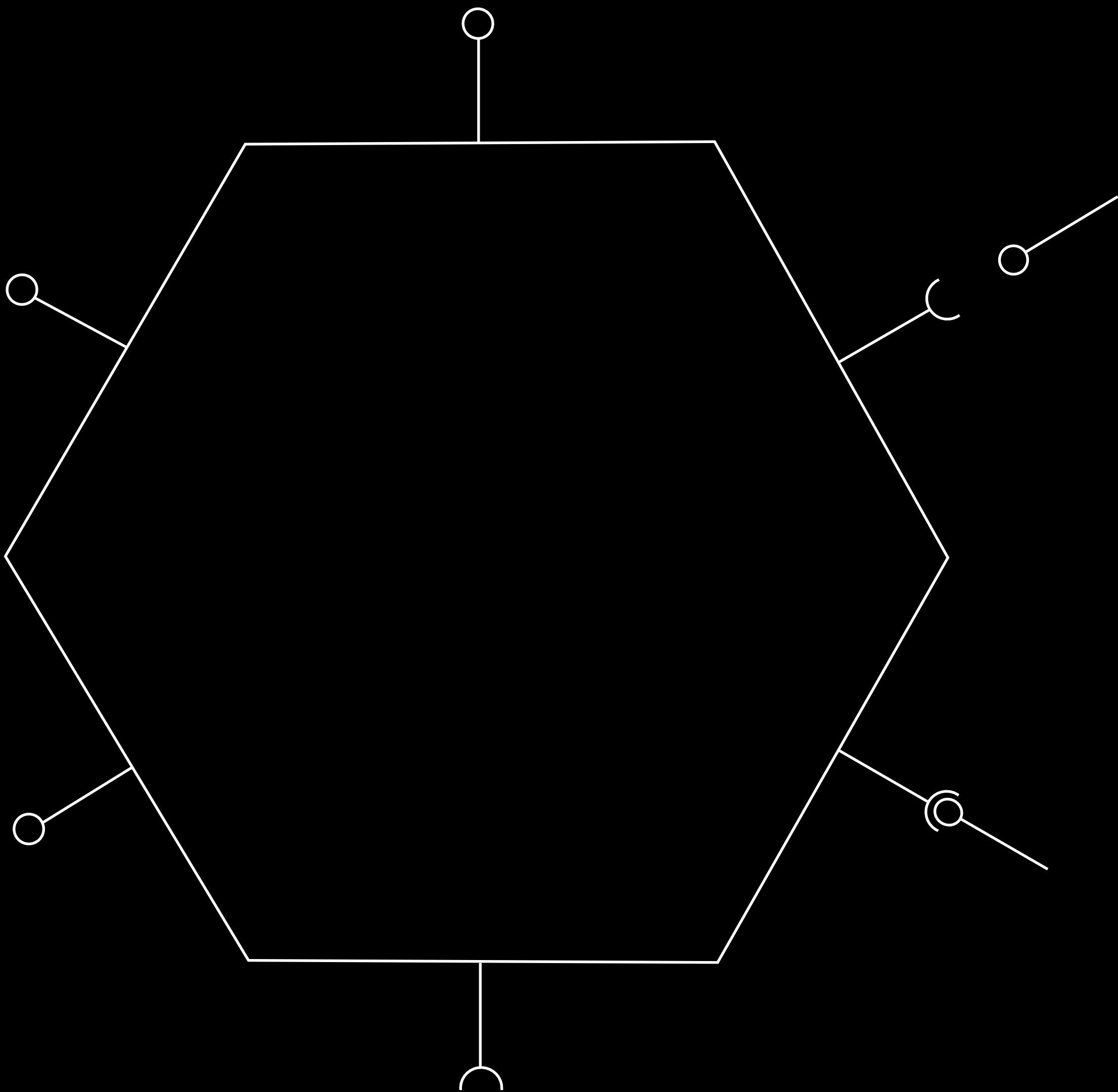
- Name: **splearn**
- Language: **Java**
- Type: **Gradle-Kotlin**
- Version: **3.4.3+**
- Group: **tobyspring**
- Artifact: **splearn**
- Package Name: **tobyspring.splearn**
- Java: **21**
- Packaging: **jar**

Splearn 프로젝트 의존라이브러리

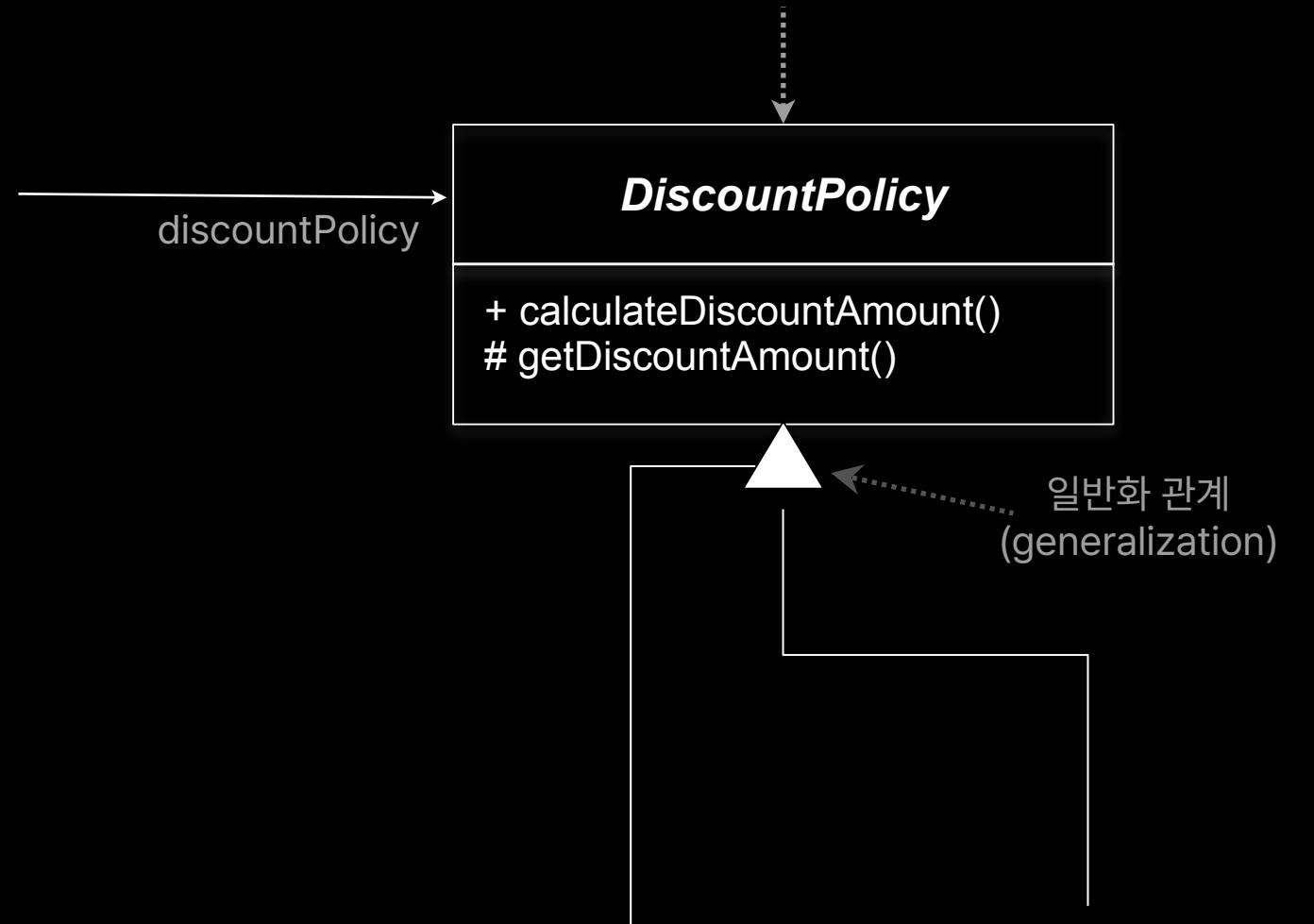
- **Lombok**
- **Docker Compose Support**
- **Spring Web**
- **Spring Data JPA**
- **H2 Database**
- **MySQL Database**

헥사고날 아키텍처

Hexagonal Architecture



추상 클래스
(abstract class)



도메인 모델이 이끄는 개발

도메인 모델

도메인(Domain)이란?

- 영토, 영역, 소유지
- 지식, 영향, 활동 영역
- 도메인 네임



소프트웨어 개발에서 도메인

- 소프트웨어는 이를 사용하는 사용자들의 활동이나 관심사와 관련이 있다
- 사용자가 프로그램, 또는 소프트웨어 서비스를 적용하는 주제 영역을 도메인이라고 한다
- 소프트웨어가 도메인을 반영하도록 만들어야 한다

도메인 모델

- 소프트웨어는 도메인의 핵심 개념과 요소들을 통합하고, 그 관계를 정확하게 구현해야 한다
- 소프트웨어는 도메인을 모델링해야 한다
- 도메인은 현실 세계의 일부이고, 단순히 코드로 직접 옮길 수 없다
따라서, 도메인의 추상화인 도메인 모델을 만들어야 한다
- 도메인 모델은 소프트웨어가 해결하려는 특정 문제 영역(도메인)의 핵심지도
- 도메인에 존재하는 중요한 개념과 이들 사이의 관계, 그리고 규칙을 표현

도메인 주도 설계(DDD)

- 도메인의 복잡성이 주는 문제를 해결하기 위한 접근 방법
- 도메인 모델을 개발 과정의 중심에 두는 방법이다
- 개발자 뿐 아니라 도메인을 가장 잘 아는 현업 전문가, 이해 관계자가 모두 참여해서 함께 도메인 모델을 만들고 계속 발전시켜야 한다
- 도메인 모델이 설계와 코드까지 이어져야 한다 (모델 주도 설계)
- 팀 안에서 도메인 모델에 기반한 단일 어휘체계를 만들고, 이를 문서, 회의, 대화, 그리고 코드까지 일관되게 사용한다 (보편 언어)

그러면 DDD로 개발할 것인가?

도메인에 집중하고
코드와 모델을 일치시키고
명확한 언어를 사용한다

도메인 모델과 보편 언어를 프로젝트에 기록

- 용어사전.md
- 도메인 모델.md
- 도메인 모델.drawio

Splearn 도메인 모델

도메인 모델 만들기

- 듣고 배우기
- '중요한 것'들 찾기 (개념 식별)
- '연결 고리' 찾기 (관계 정의)
- '것'들을 설명하기 (속성 및 기본 행위 명시)
 - 그려보기 (시각화)
 - 이야기 하고 다듬기 (반복)

도메인 전문가가 없다면

- 창업가, 기획자, PO를 가상의 도메인 전문가로 활용
 - 다양한 탐색과 집단적인 학습 - 이벤트 스토밍
 - 가설 기반의 점진적인 모델링
 - 보편 언어의 의식적인 구축과 진화
 - 베타 테스터, 초기 사용자와의 적극적인 소통
 - 도메인 모델의 발전과 진화, 공유를 더욱 적극적으로 해야 한다
-
- 경쟁 업체 서비스 분석

도메인 모델 패턴과 헥사고날 아키텍처

도메인 모델 패턴으로 회원 도메인 개발

도메인 모델 패턴

- 도메인/비즈니스 로직을 구성하는 아키텍처 패턴의 한 가지
 - 도메인 모델의 속성과 행위를 모두 포함하는 **도메인의 오브젝트 모델**이다
 - 오브젝트 모델이기 때문에 복잡한 연관 관계, 커스텀 속성, 상속 등을 사용할 수 있다
-
- 트랜잭션 스크립트는 하나의 업무절차(TX)를 처리하기 위한 **스크립트(메소드)**를 만들고 비즈니스 로직을 순서대로 코드로 작성하는 방법

엔티티(Entity)

- 도메인 모델을 만들 때 사용하는 패턴
- 도메인 안에 있는 대상이나 개념
- 고유한 식별자(identity)를 가지고 이를 통해서 개별적으로 구분된다
- 생명주기(life cycle)를 가진다. 시간의 흐름에 따라 상태가 변경될 수 있다

값 객체(Value Object)

- 도메인 모델에서 식별자가 필요하지 않고 속성/값으로만 구별되는 오브젝트
- 엔티티가 너무 많은 책임을 가지는 것을 방지하고
특정 속성 관련 행위를 분리해서 엔티티를 더 집중된 상태로 유지하게 한다
- 원시 타입보다는 도메인 개념을 더 명시적으로 나타내서 모델의 명확성을 높인다
- 생성 이후에 상태가 변하지 않고 변경이 필요하면 새로운 객체로 교체한다
- 풍부한 기능을 가진다
- 자체 유효성 검사도 가능하다
- 개념에 포함된 여러 개의 속성을 가질 수 있다

원시 타입을 객체로 교체 - 리팩터링

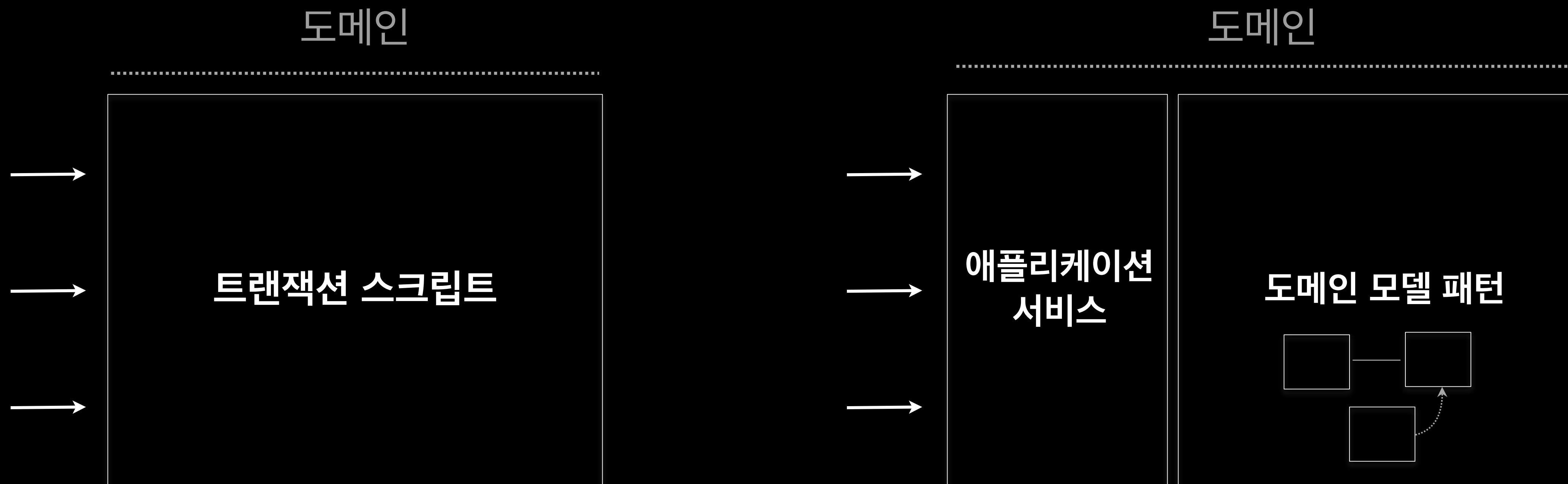
- 원시 타입의 남용을 개선하고 코드의 명시성과 안정성을 높이는 리팩터링 방법
- 값 객체는 도메인 모델의 핵심 구성 요소로 값 그 자체의 의미를 담아내고
불변성, 동등성 등의 도메인 로직을 포함하여 설계하는 방식

도메인 모델 패턴의 두 가지 스타일

- **단순 도메인 모델(Simple Domain Model):**
DB 설계와 유사, 테이블에 대해 하나의 도메인 오브젝트
- **풍성한 도메인 모델(Rich Domain Model):**
상속, 전략, GoF 디자인 패턴, 연관 관계
복잡한 로직에 적합하지만 DB 매팅이 어려울 수 있다

도메인 로직의 API 개발

- 도메인 모델 패턴은 트랜잭션 스크립트 처럼 작업 단위의 절차형 API를 만들기가 어렵다
- 도메인 로직의 명확한 작업 단위 API를 제공하는 애플리케이션 서비스가 필요



계층형 아키텍처(Layered Architecture)

- 특정 책임을 가진 여러 개의 계층으로 구성하는 아키텍처
- 관심사의 분리(separation of concerns) 원칙을 따른다
- 상위 계층은 (바로) 아래 계층의 기능에만 의존해야 한다
- 종속성을 제한하도록 명시적인 인터페이스를 통해서 기능을 캡슐화하고 모든 접근이 인터페이스를 통하여도록 만든다
- View, Domain, Data 3계층 구조가 대표적이다

헥사고날 아키텍처로 회원 애플리케이션 개발

헥사고날 아키텍처의 사실과 오해

Hexagonal Architecture

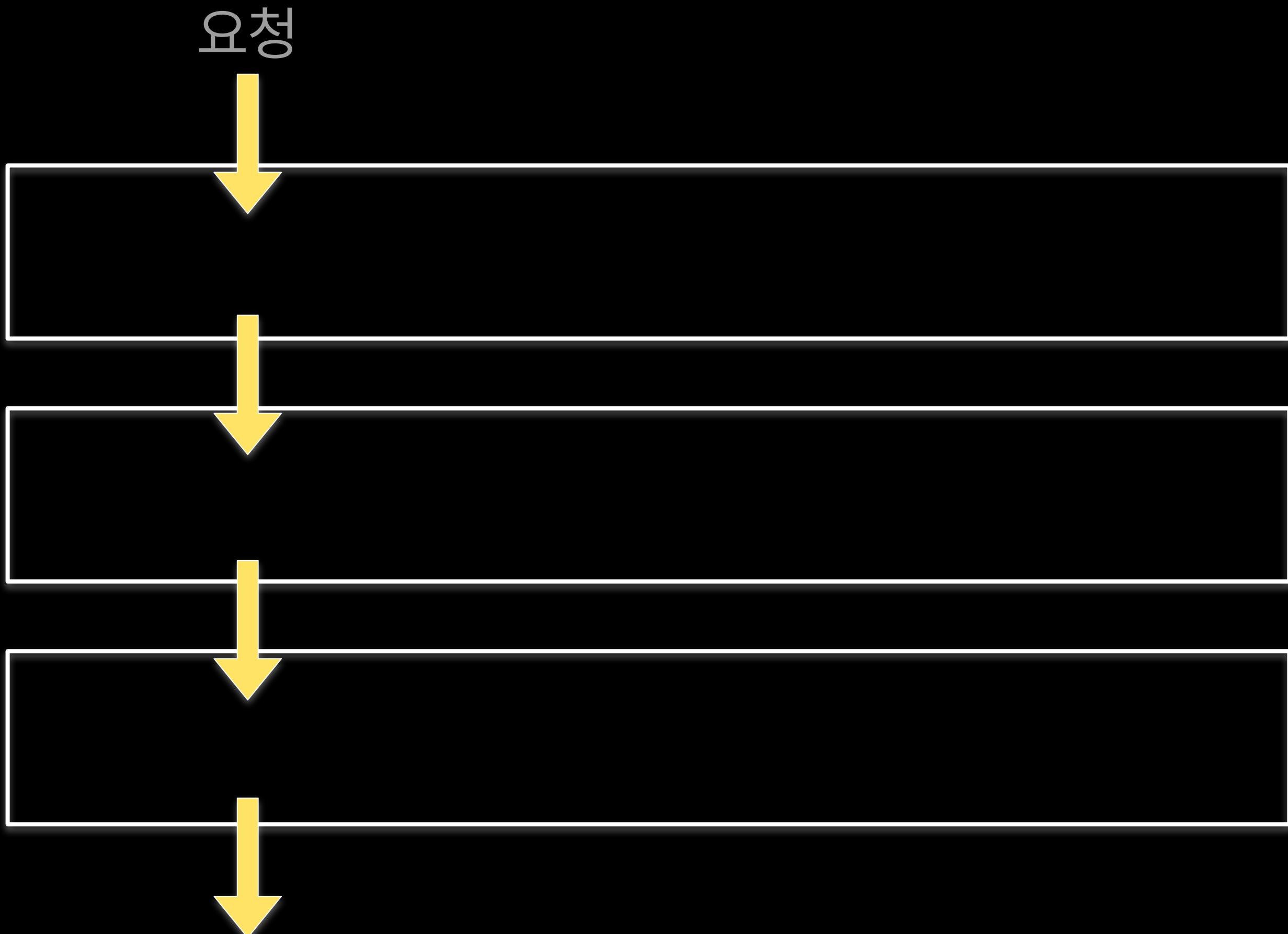
아키텍처

- 시스템의 **기본적인 구조**를 정의한다
- 시스템의 중요한 **품질 속성**에 큰 영향을 미친다
- 설계 결정의 **기반이 되는 핵심적인 개념**이다
- 기본 구성 요소와 상호 관계, 제약 조건, 원칙 등을 포함한다

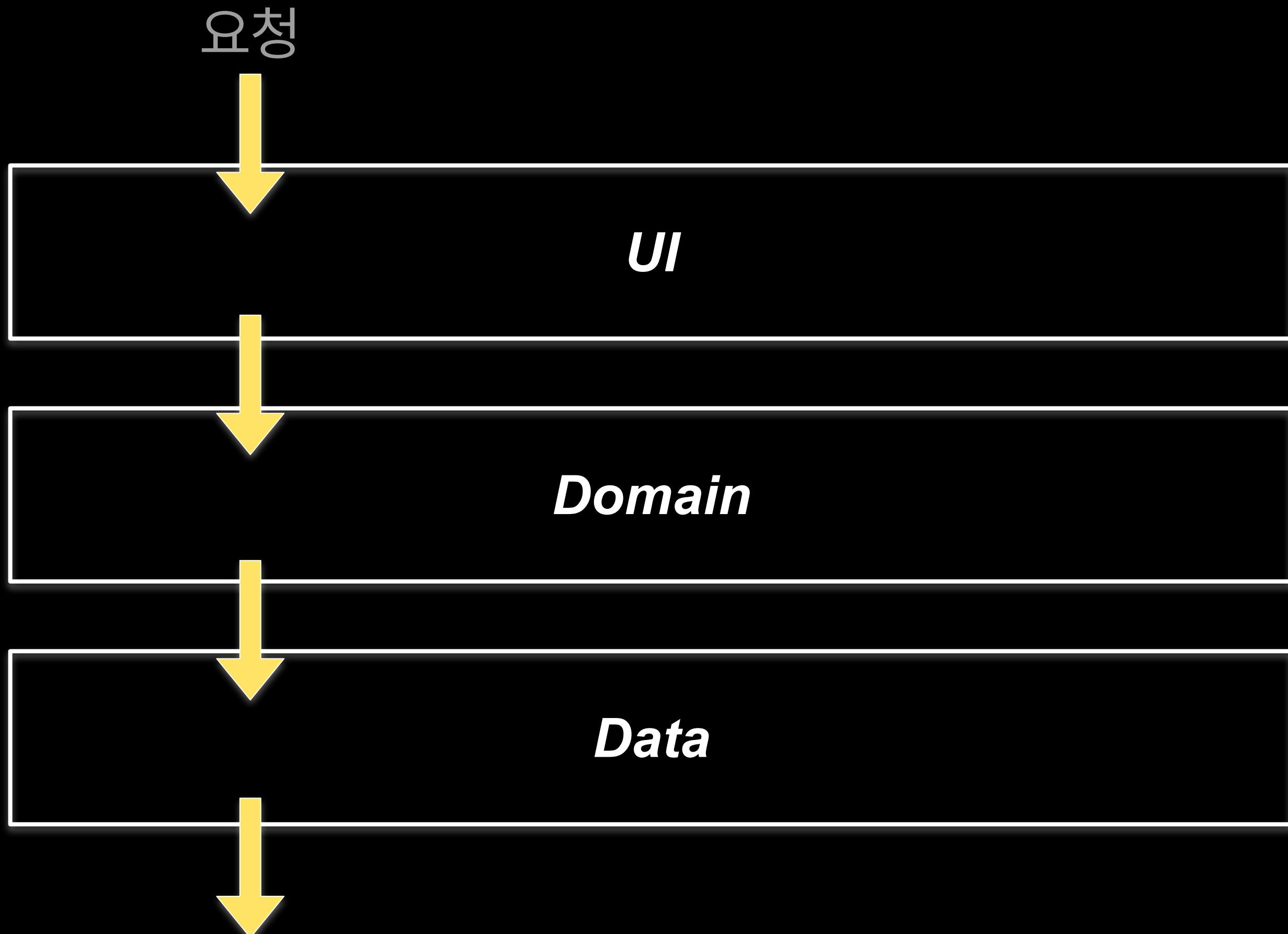
계층형 아키텍처(Layered Architecture)

- 서브 시스템을 계층(layers)으로 구조화 하는 아키텍처 스타일이다
- 계층은 사용 관계로 연결된다
- 사용 관계는 일반적으로 단방향이어야 한다는 핵심 제약이 있다
 - 상위 계층이 하위 계층의 서비스를 사용하는 하향식 흐름을 가진다
- 각 계층이 하위 계층의 내부 작동 방식을 알지 못하고 제한된 인터페이스만 사용하도록 한다
(계층 격리 Layers of Isolation)
 - 어떤 레이어의 변경이 다른 레이어의 컴포넌트에게 가능한 영향을 주지 않도록 해야 한다

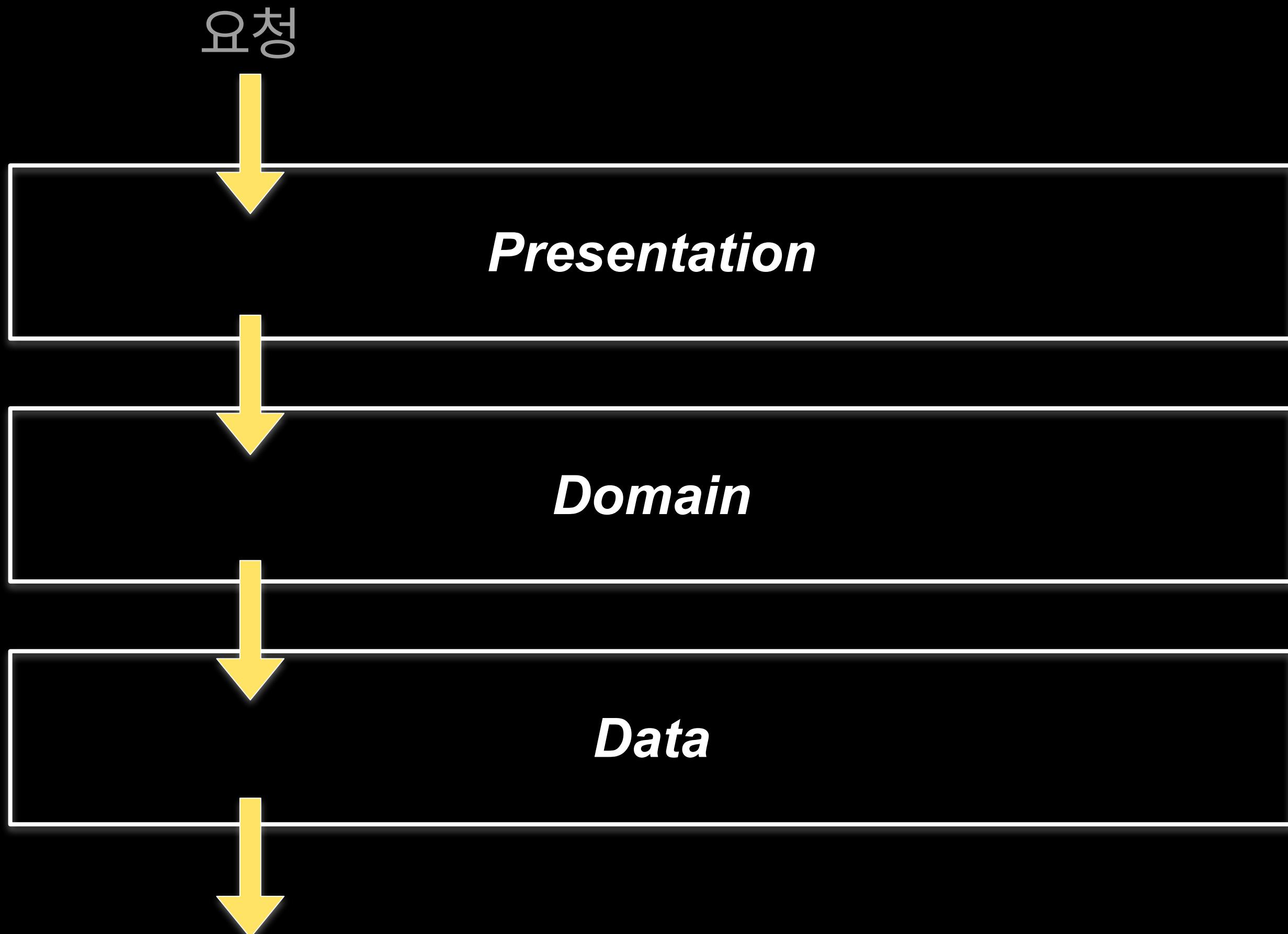
3계층 아키텍처



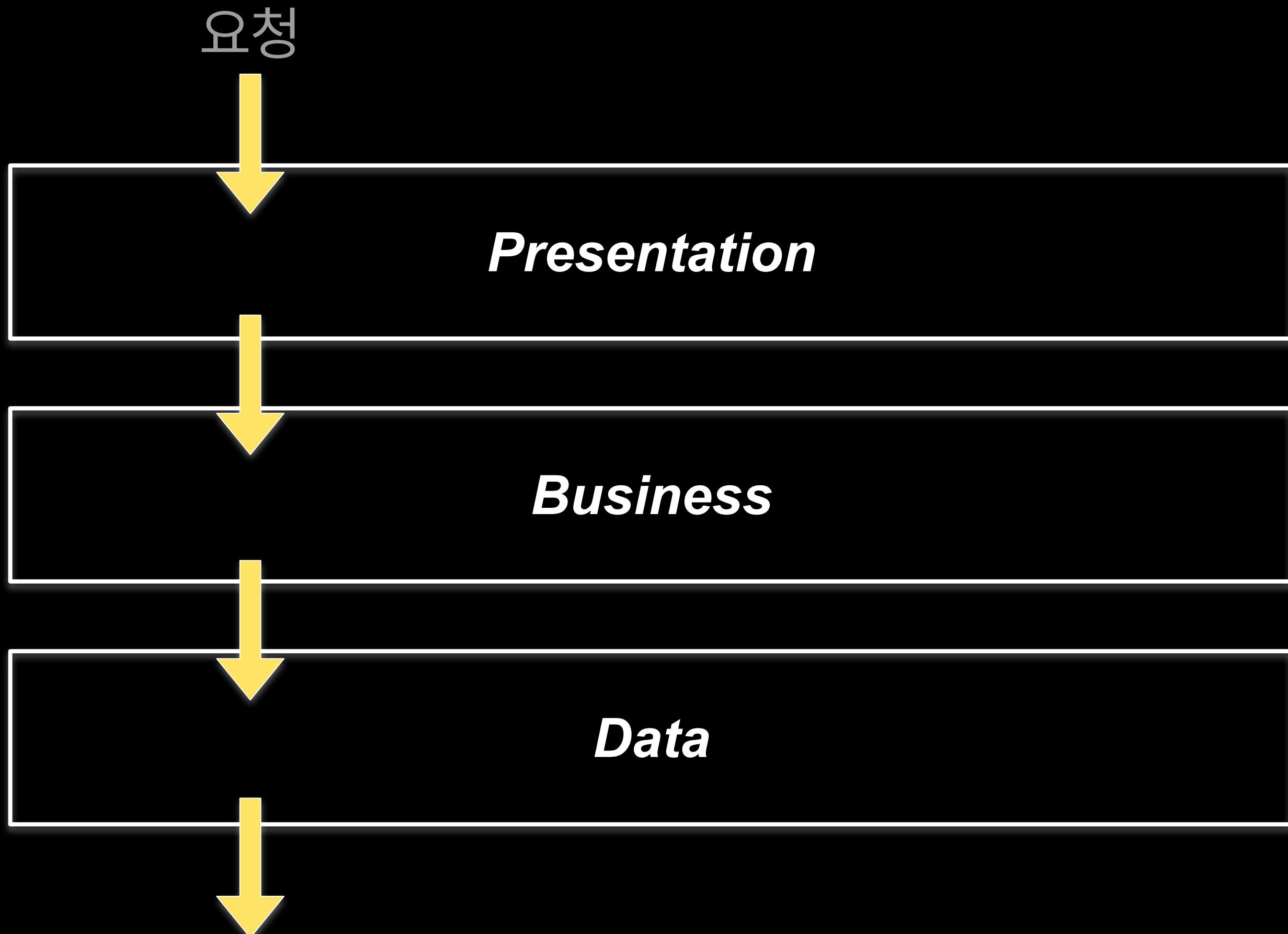
3계층 아키텍처



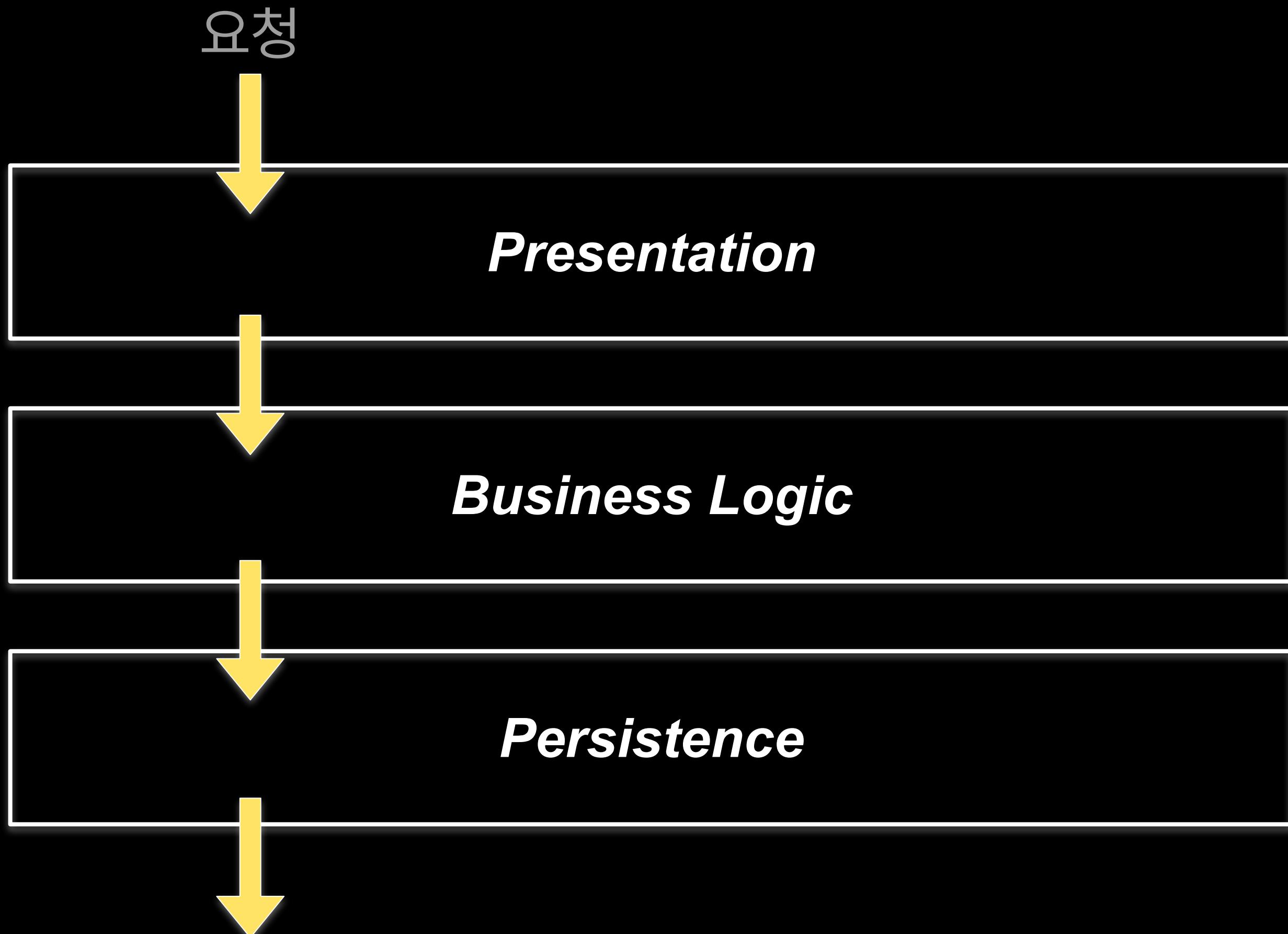
3계층 아키텍처



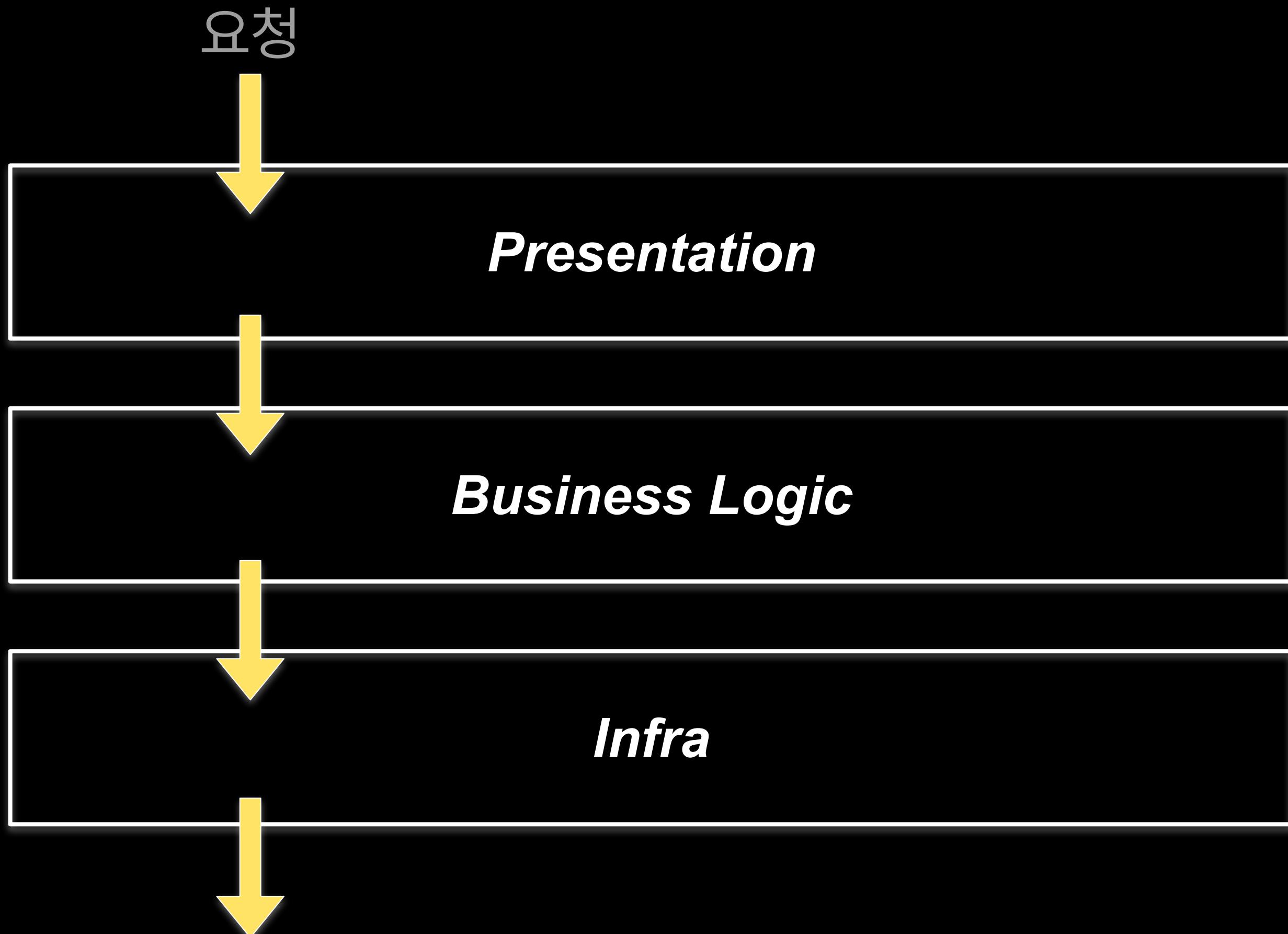
3계층 아키텍처



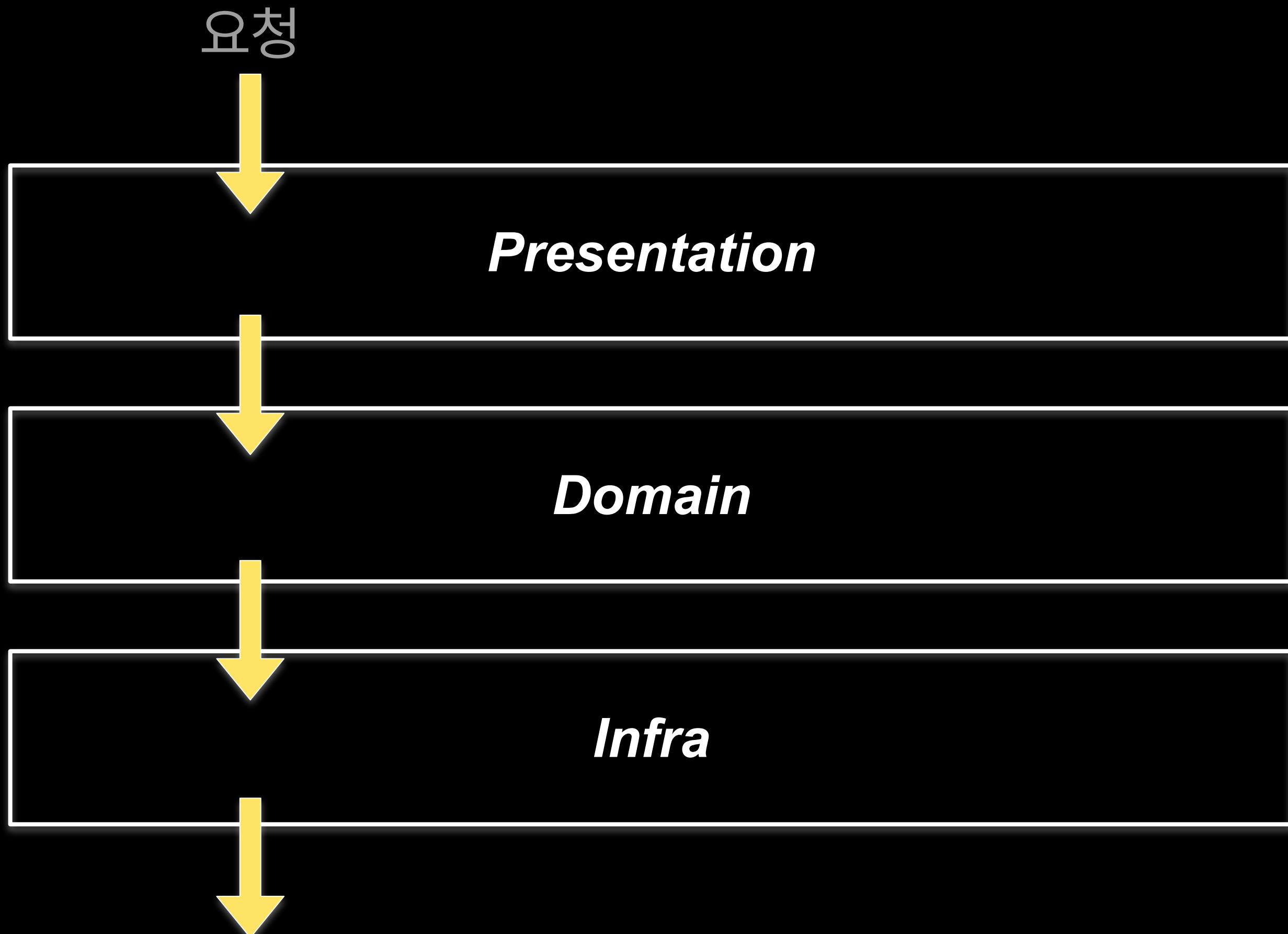
3계층 아키텍처



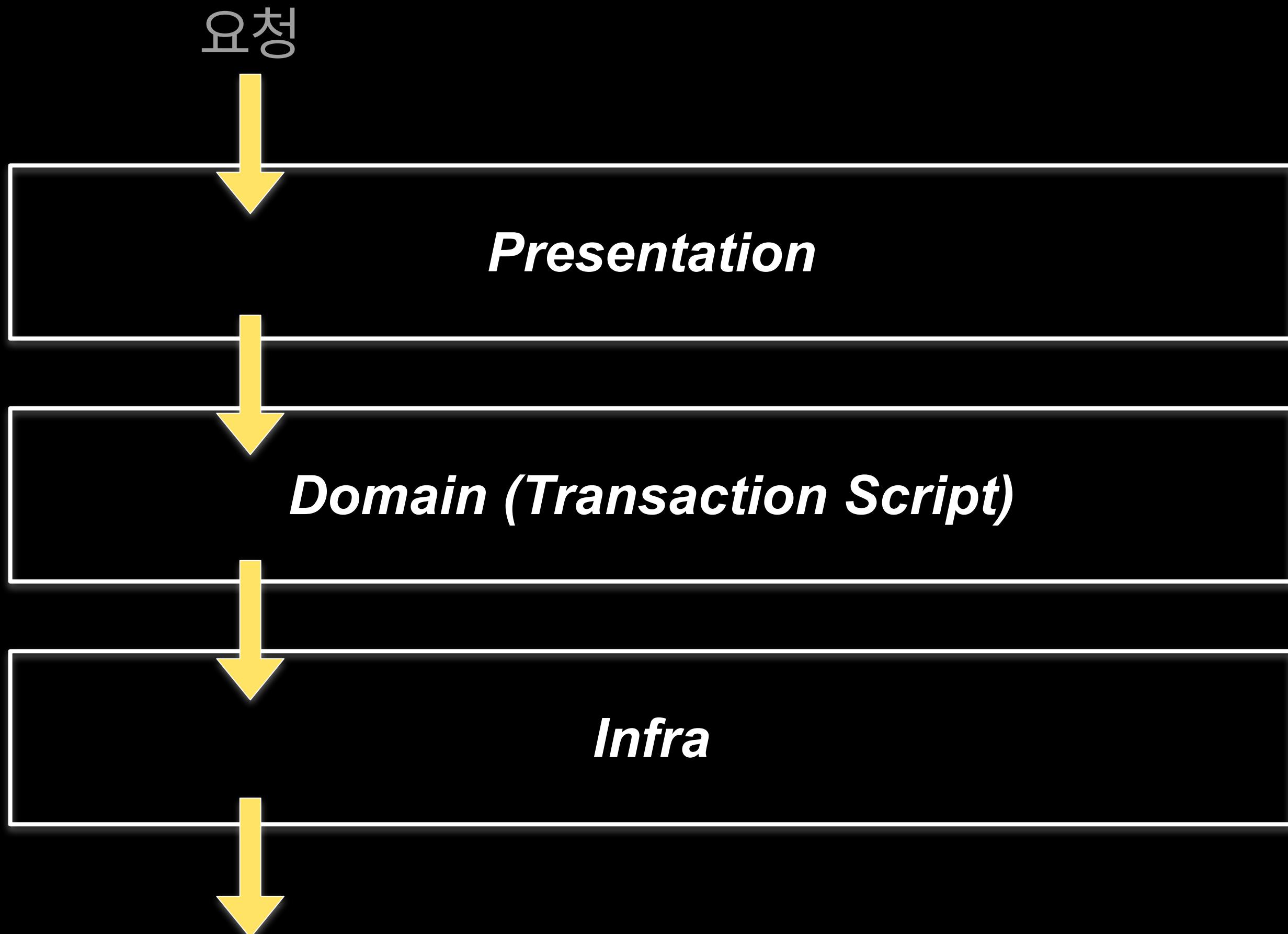
3계층 아키텍처



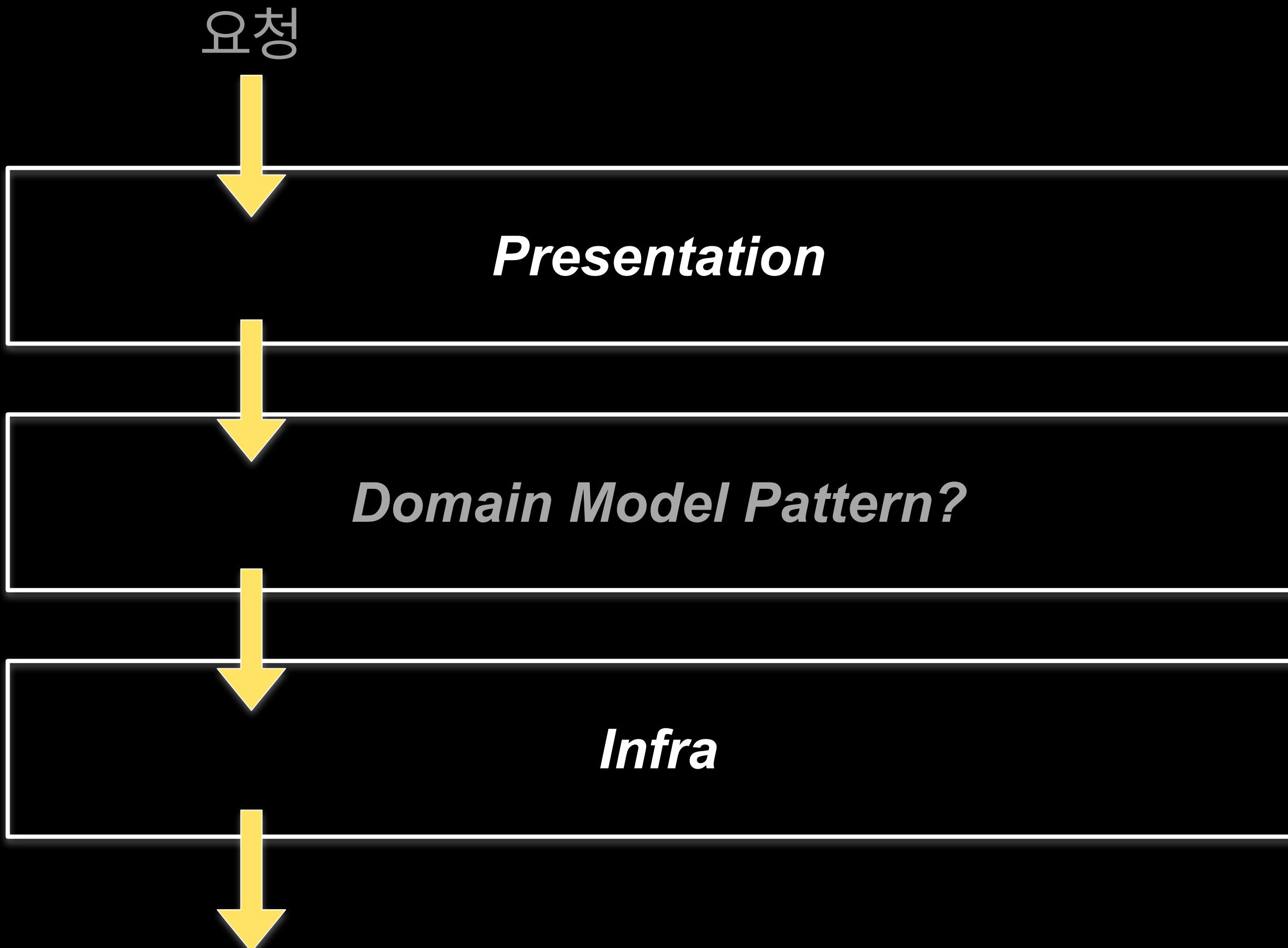
3계층 아키텍처



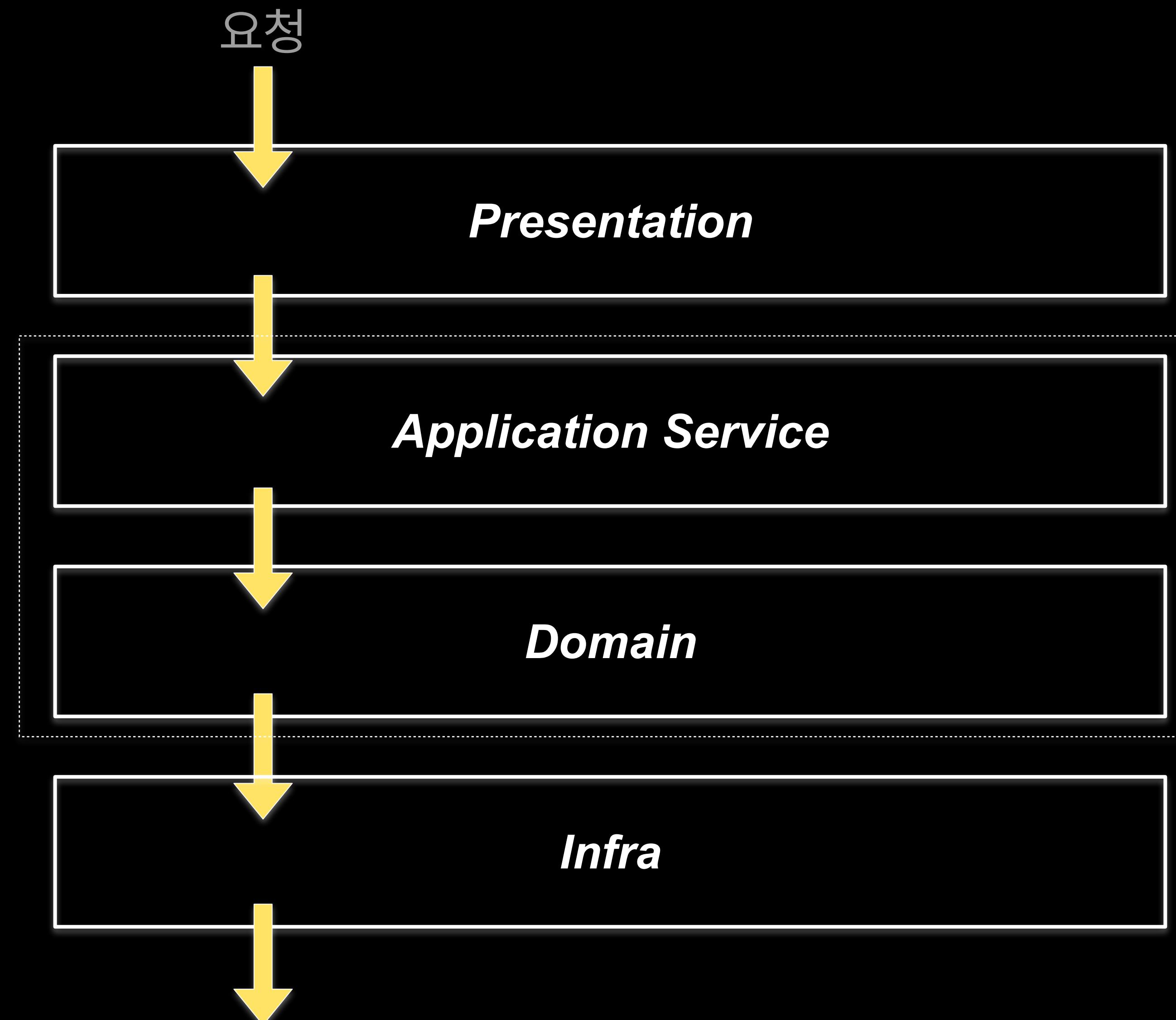
3계층 아키텍처



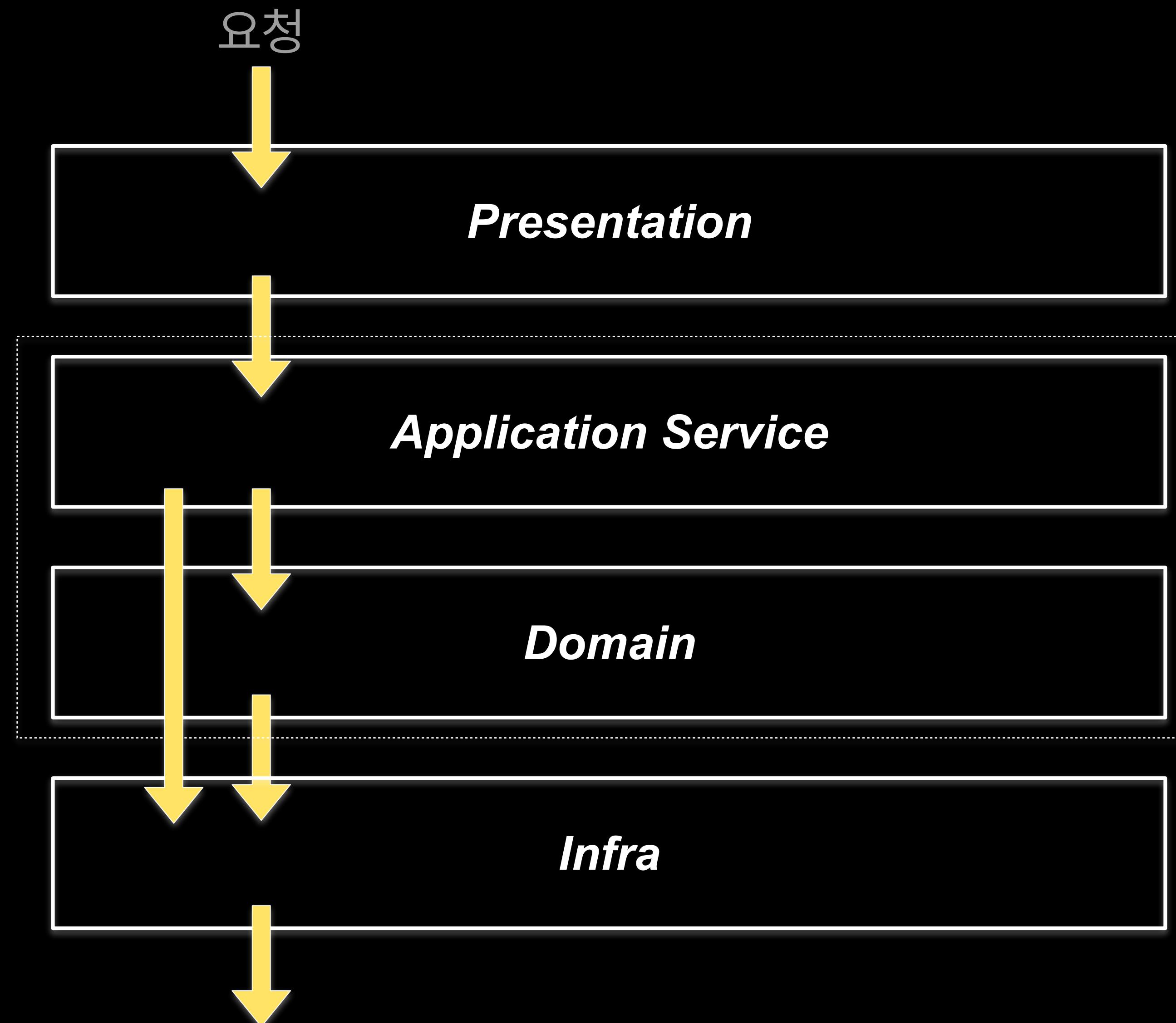
3계층 아키텍처



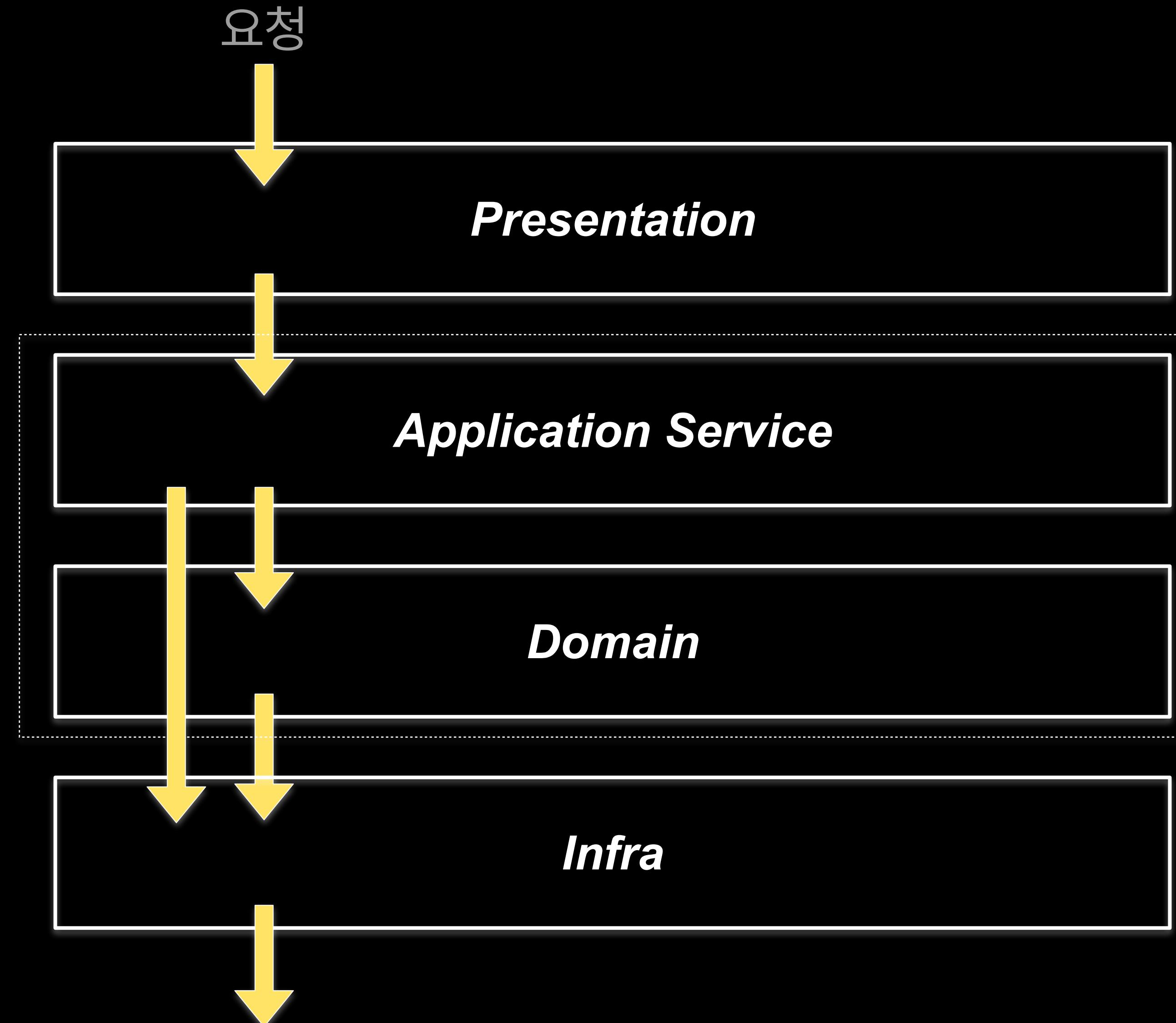
3 or 4 계층 아키텍처



3 or 4 계층 아키텍처



계층형 아키텍처

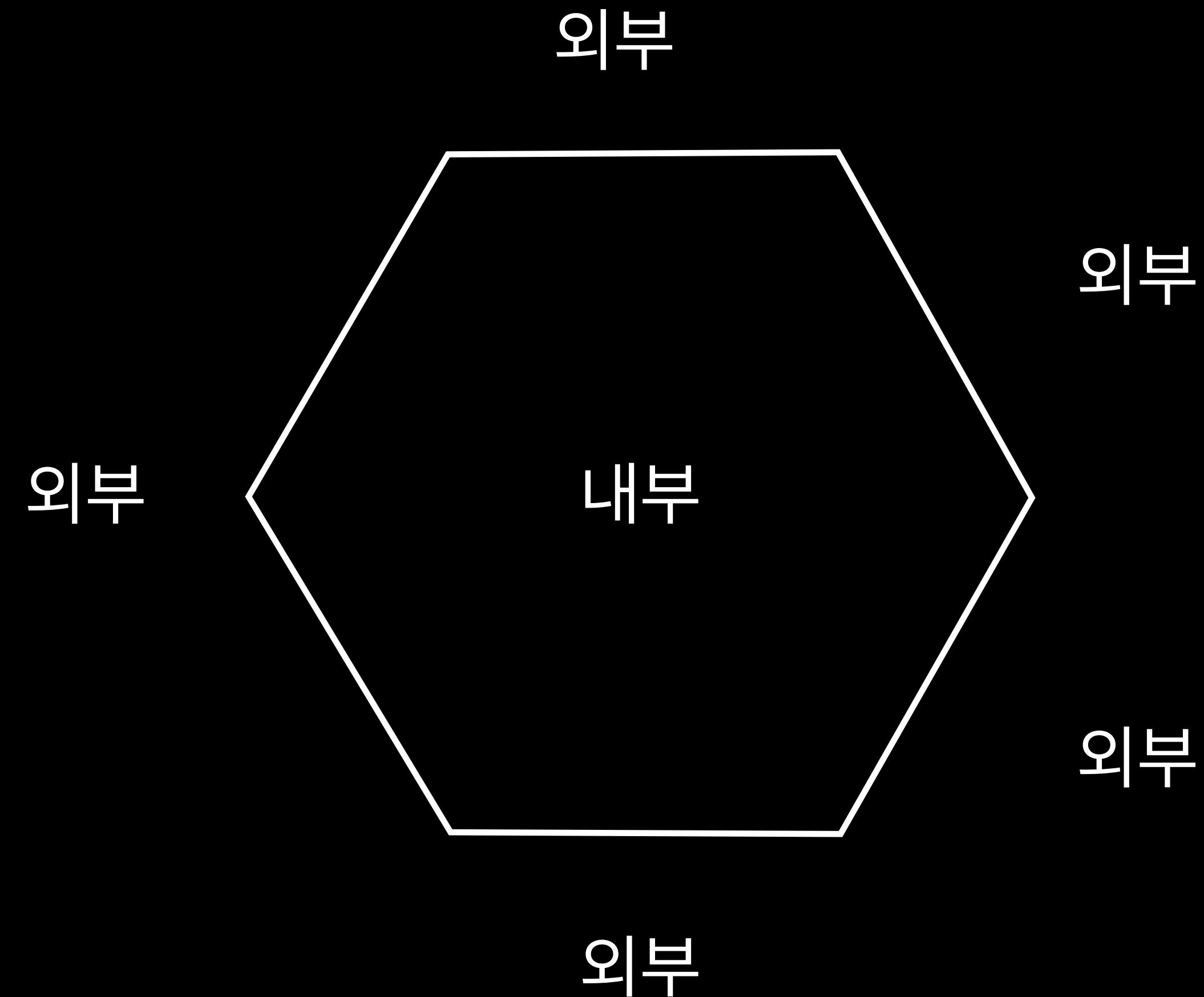


헥사고날 아키텍처

- 2005년 앤리스터 코번(Alistair Cockburn)이 제안한 아키텍처
- 계층형 아키텍처의 단방향 비대칭 구조가 아닌 대칭형(symmetric) 아키텍처
- 위 아래, 좌 우가 아닌 애플리케이션의 내부와 외부 세계라는 대칭 구조를 가진다

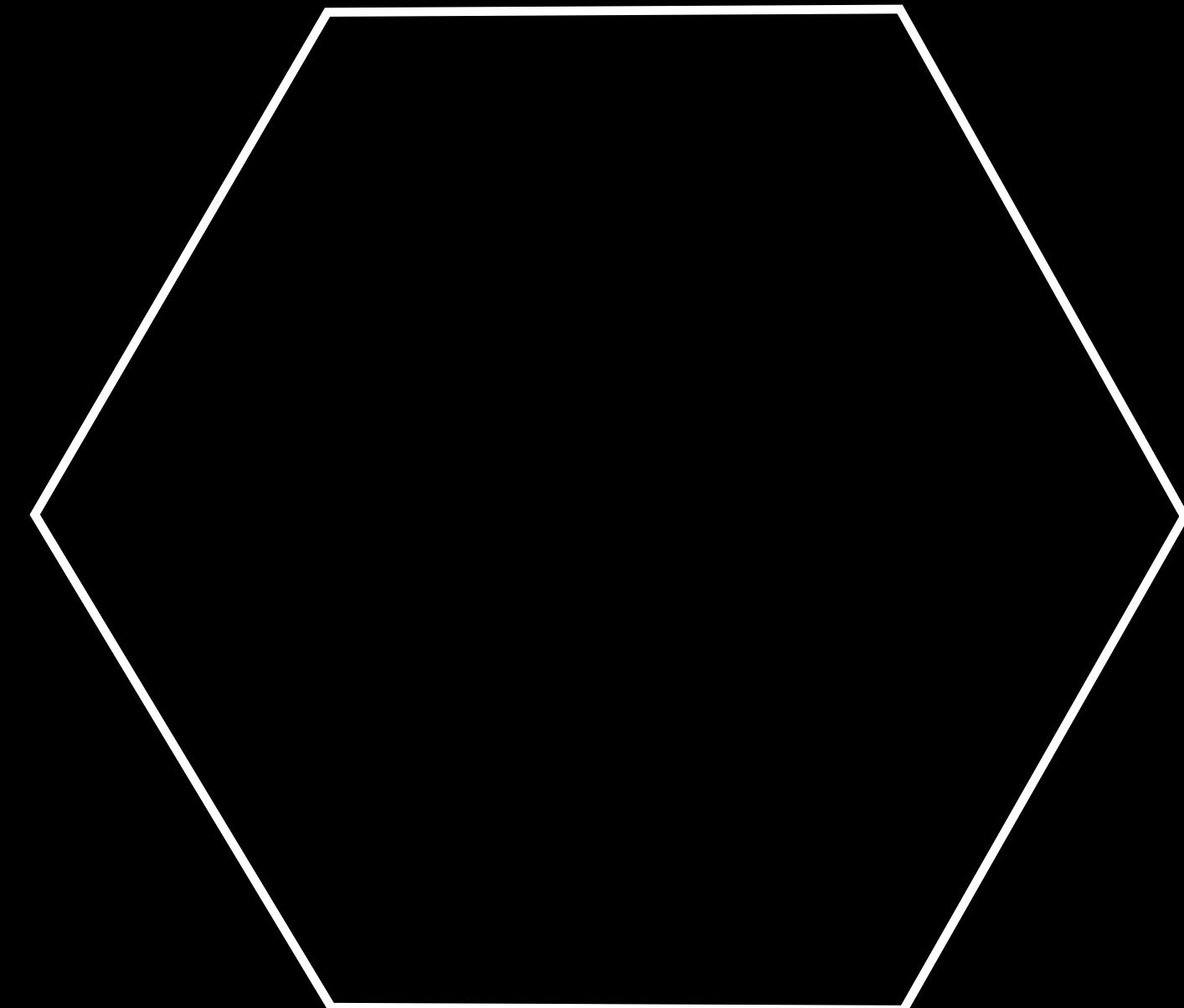
헥사고날 아키텍처

- 아키텍처의 대칭성을 가진 구조를, 그리기 쉬운 대표적인 도형인 육각형(hexagonal)으로 설명
- 육각형, 6개의 면을 가졌다는 게 특별한 의미를 가지는 건 아니다



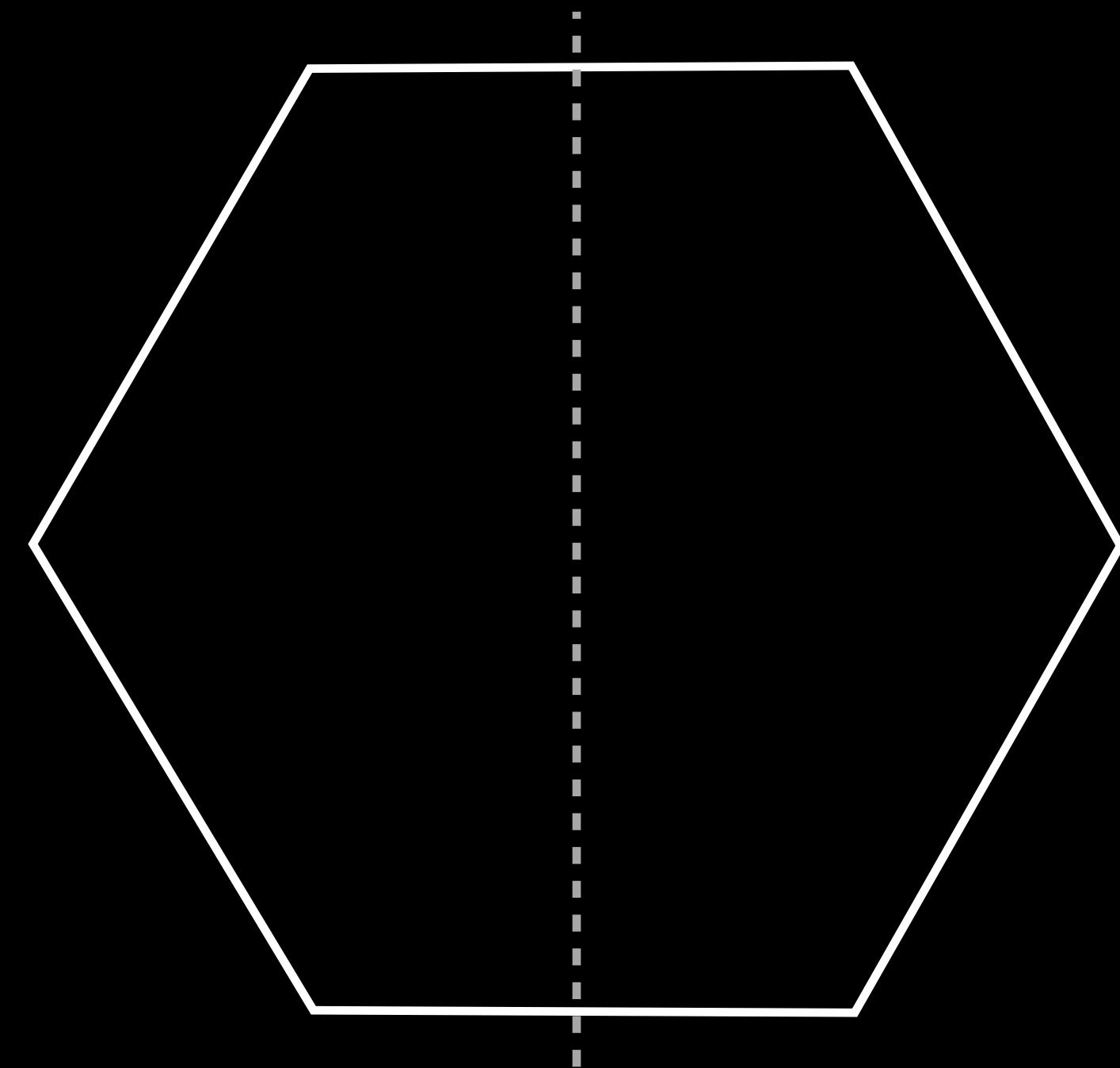
헥사고날 아키텍처

- 아키텍처의 대칭성을 가진 구조를, 그리기 쉬운 대표적인 도형인 육각형(hexagonal)으로 설명
- 육각형, 6개의 면을 가졌다는 게 특별한 의미를 가지는 건 아니다



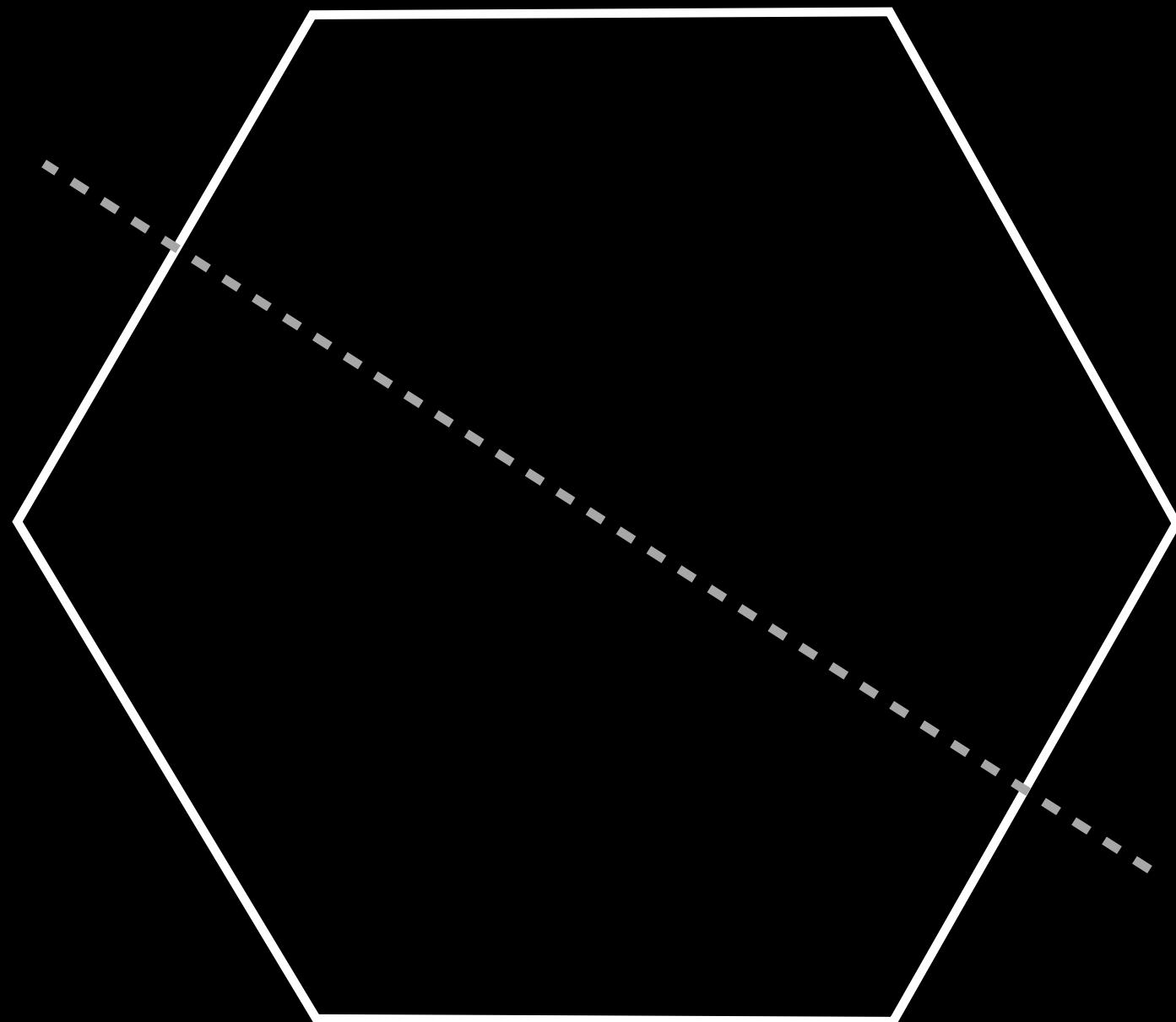
헥사고날 아키텍처

- 아키텍처의 대칭성을 가진 구조를, 그리기 쉬운 대표적인 도형인 육각형(hexagonal)으로 설명
- 육각형, 6개의 면을 가졌다는 게 특별한 의미를 가지는 건 아니다



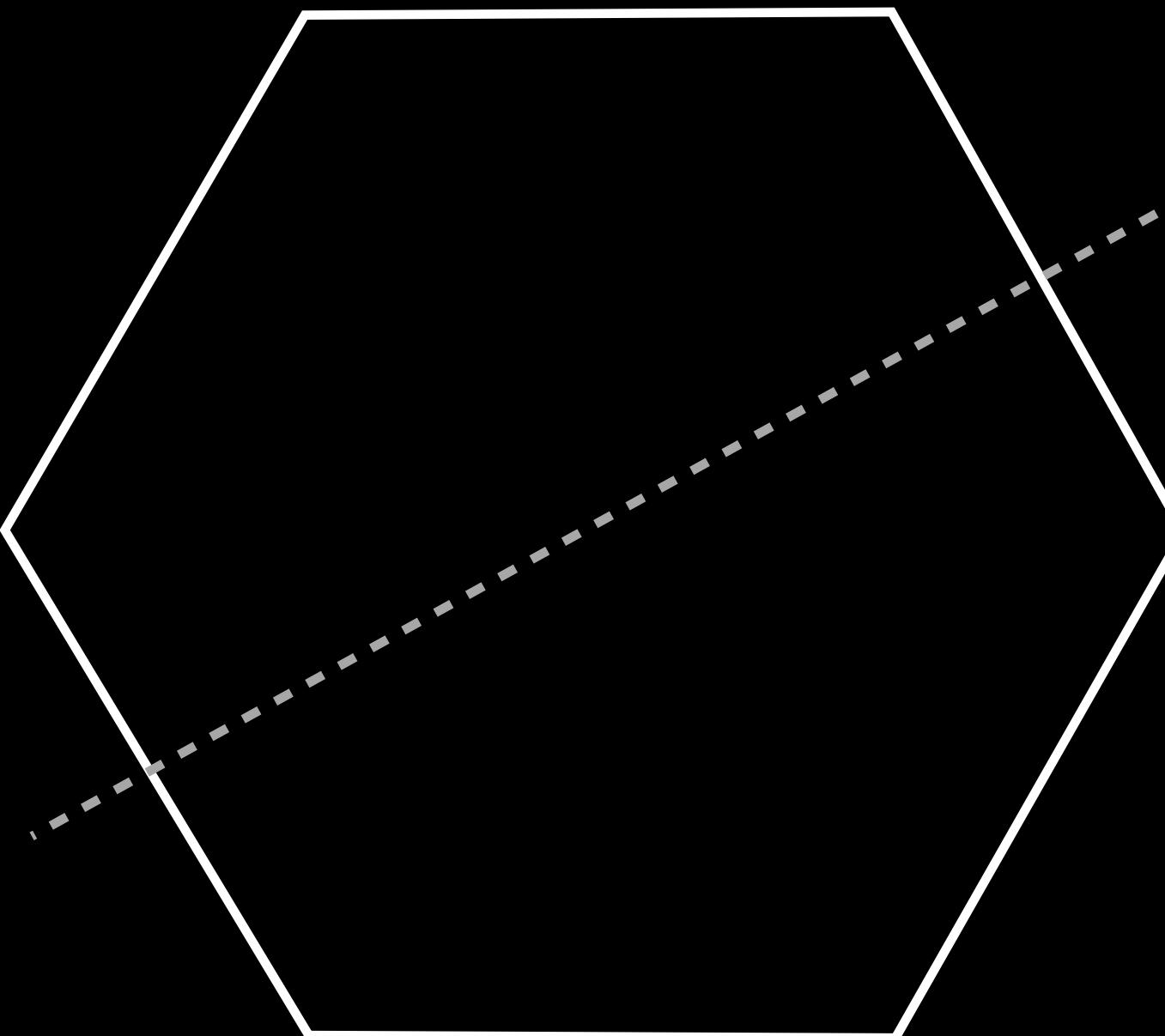
헥사고날 아키텍처

- 아키텍처의 대칭성을 가진 구조를, 그리기 쉬운 대표적인 도형인 육각형(hexagonal)으로 설명
- 육각형, 6개의 면을 가졌다는 게 특별한 의미를 가지는 건 아니다



헥사고날 아키텍처

- 아키텍처의 대칭성을 가진 구조를, 그리기 쉬운 대표적인 도형인 육각형(hexagonal)으로 설명
- 육각형, 6개의 면을 가졌다는 게 특별한 의미를 가지는 건 아니다



헥사곤의 내부

- 쉽게 변하지 않는, 중요한 도메인 로직을 담은 코어 애플리케이션
 - 도메인 로직을 가진 트랜잭션 스크립트
 - 애플리케이션 서비스와 도메인 모델 패턴을 따라서 만든 도메인

헥사곤의 외부

- 헥사곤과 상호작용(interaction)하는 모든 것 - 액터(Actor)
- 사용자, 브라우저, CLI 명령, 기계, 다른 시스템
- 운영 환경, DB, 메시징 시스템, 메일 시스템, 원격 서비스
- 테스트

헥사고날 아키텍처의 특징과 혜택

- 테스팅! 운영 시스템에 연결되지 않고 애플리케이션 테스트
- 애플리케이션과 상호작용하는 액터가 바뀌더라도 다시 빌드하지 않고 테스트
- UI 디테일이나 기술 정보가 도메인 로직 안으로 노출되지 않도록 보호한다
반대도 마찬가지
- 컴포넌트를 각각 개발하고 연결하는 방식으로 큰 시스템을 분리할 수 있다
- 시간이 지나면서 외부 연결을 다른 것으로 변경할 수 있다
- 기술 요소를 제거했기 때문에 도메인 설계에 집중할 수 있다

헥사곤을 부르는 여러가지 이름

- 헥사곤
- 애플리케이션
- 앱(**App**)
- 코어 시스템
- **SuD(System under Development)**
- **SuT(System under Test)**

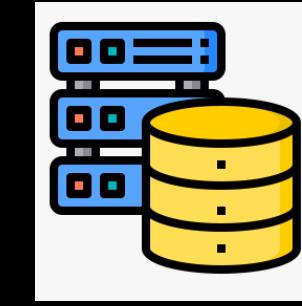
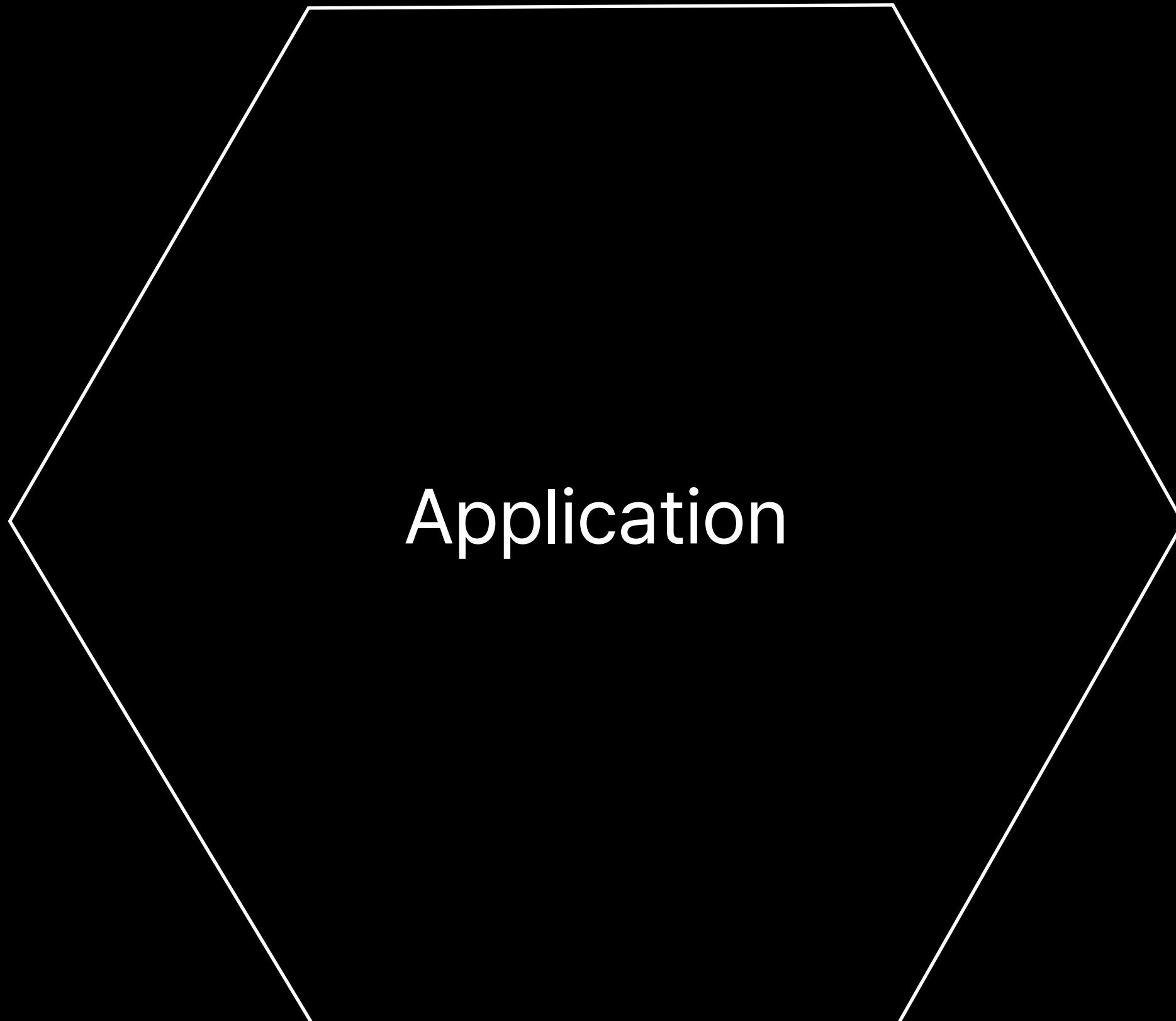
헥사곤을 부르는 여러가지 이름

- 헥사곤
- 애플리케이션
- 앱(App)
- 코어 시스템
- SuD(System under Development)
- SuT(System under Test)

액터와 애플리케이션의 상호작용은 어떻게?

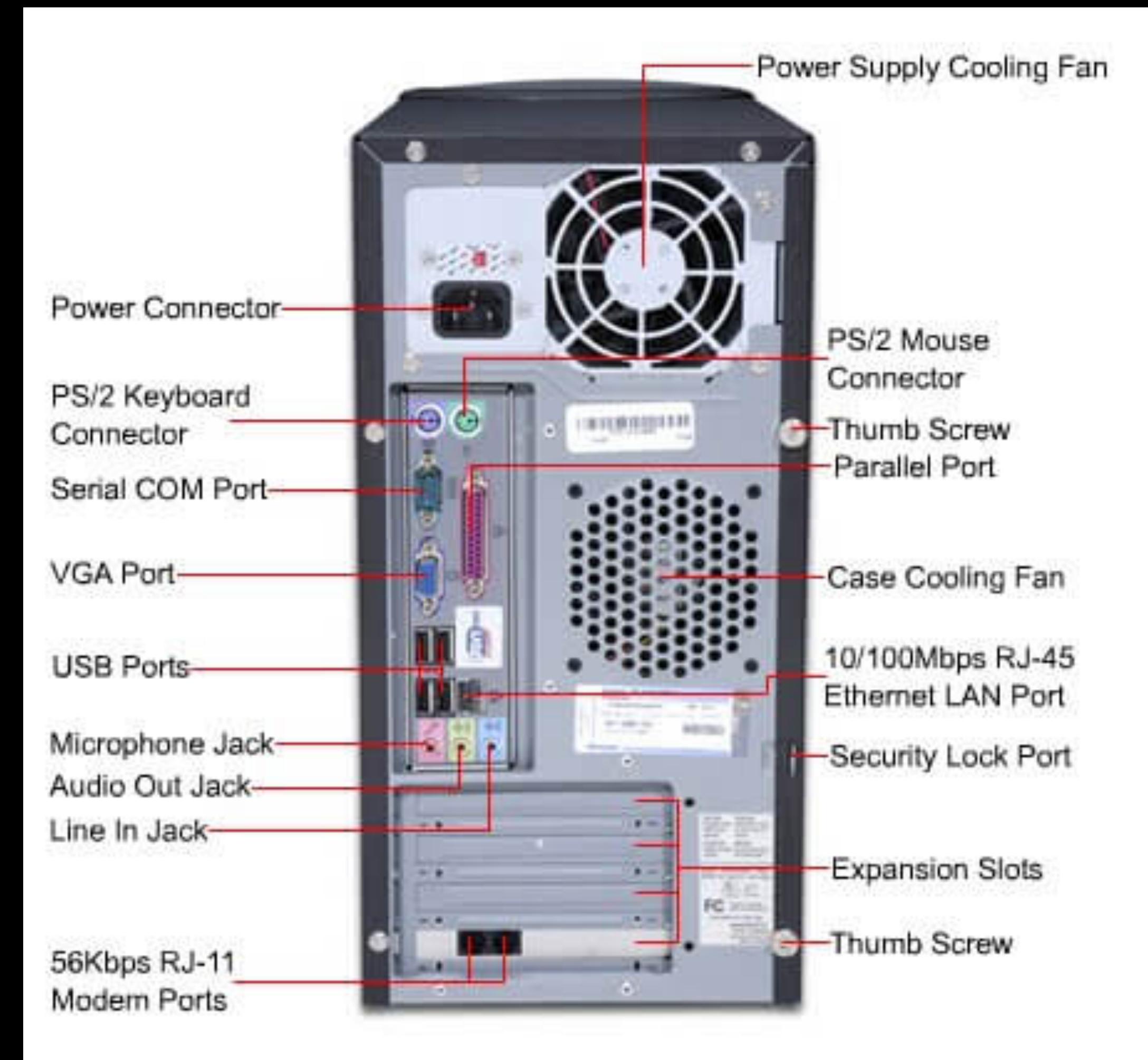


Test



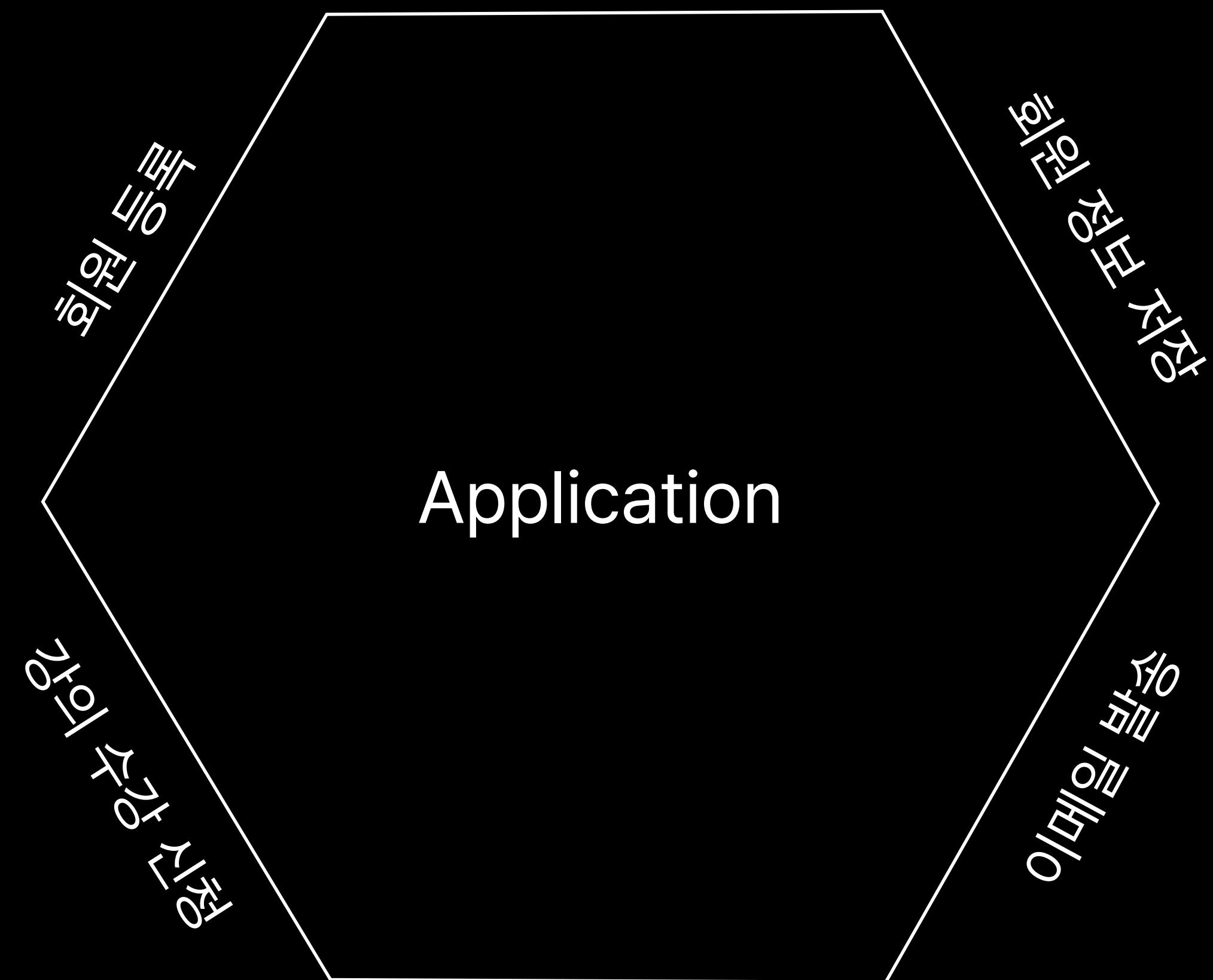
mock

포트(port)



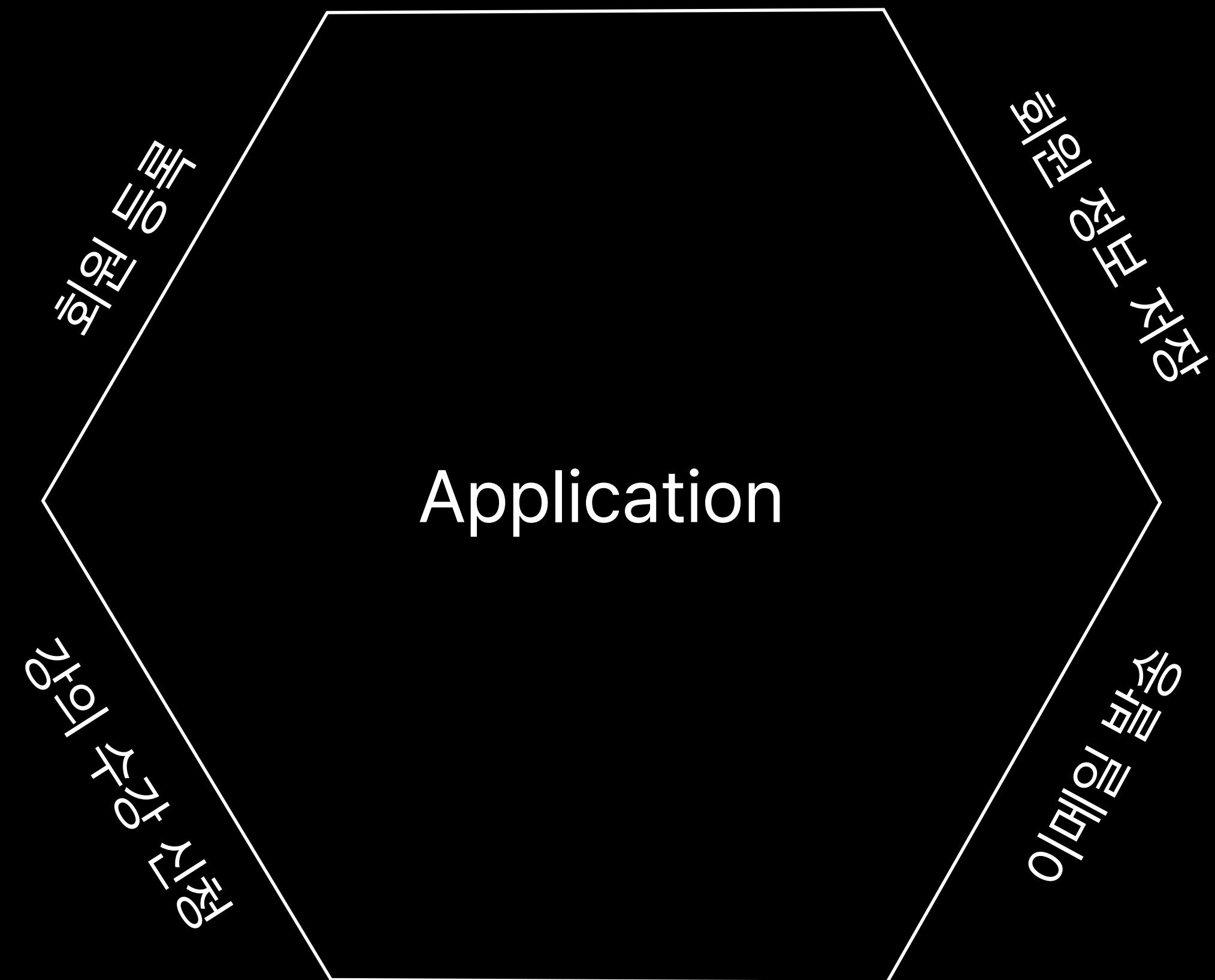
포트(port)

- 애플리케이션이 외부 세계와 **의도(intention)**를 가지고 상호작용 하는 아이디어를 캡처한 것
- 단순히 데이터를 주고받는 것이 아니라, **명확한 목적과 방향을 가지고 외부와 연결된다**



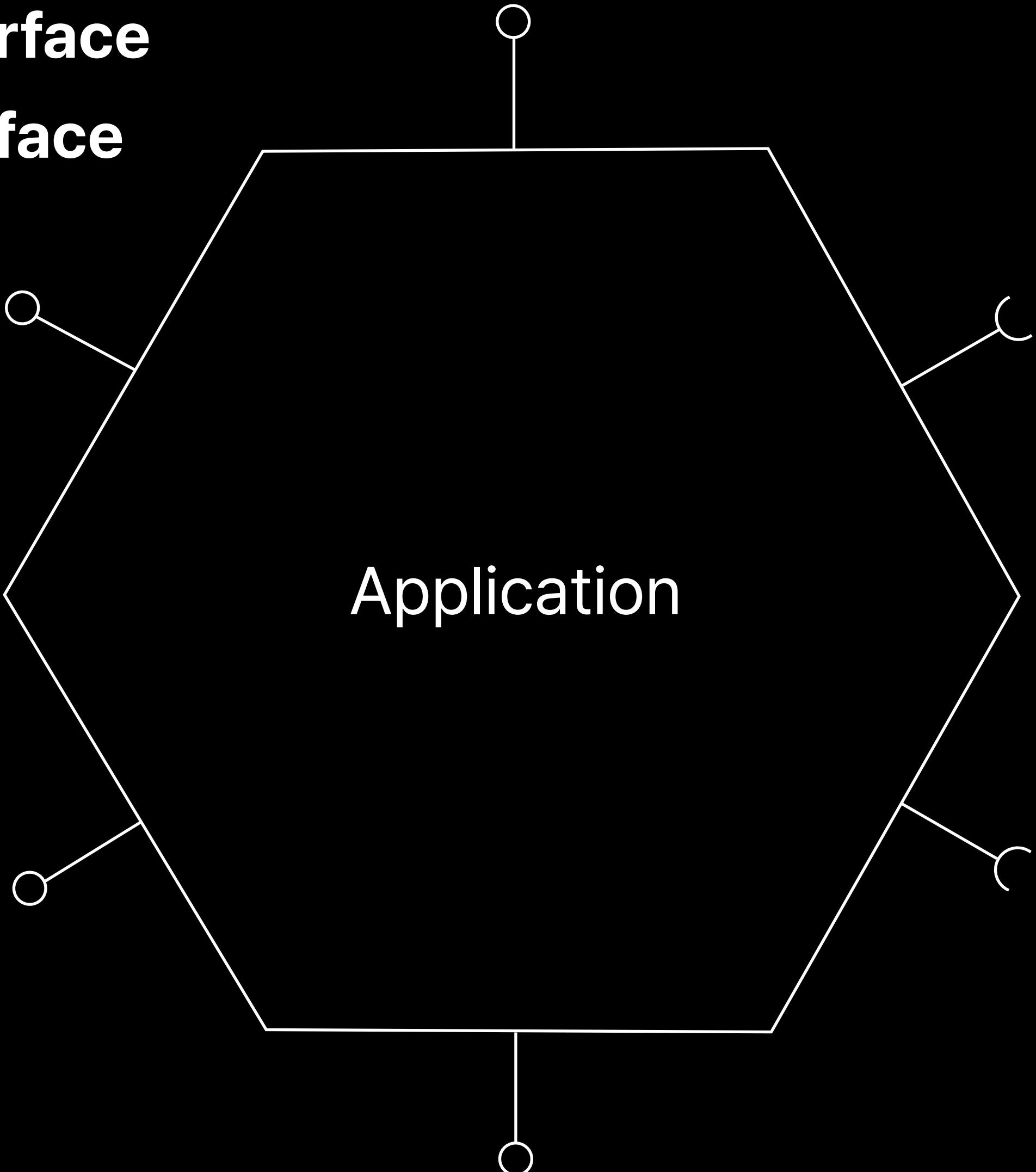
포트(port)

- 애플리케이션이 정의한 **인터페이스**로 만들어진다

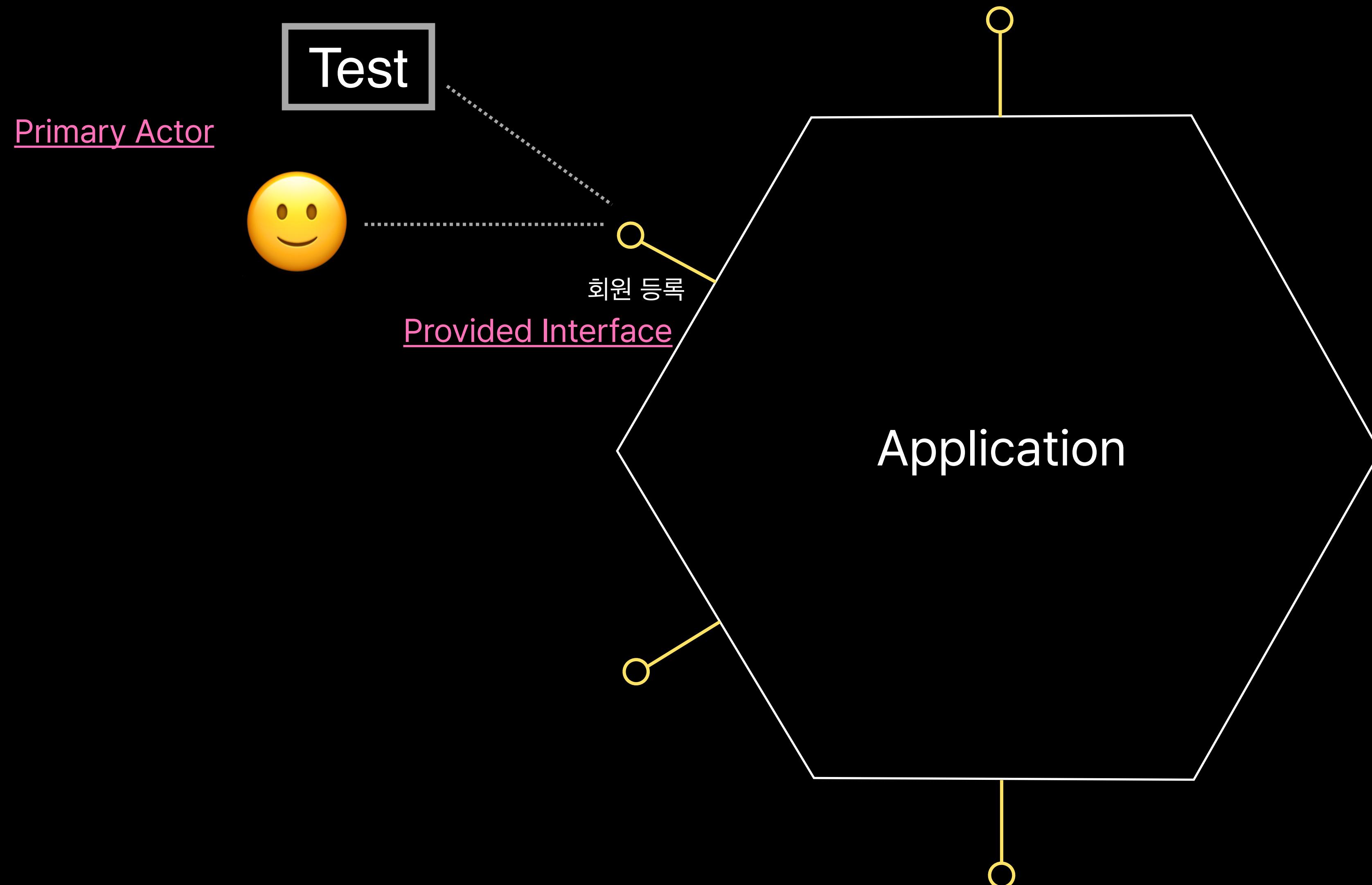


인터페이스

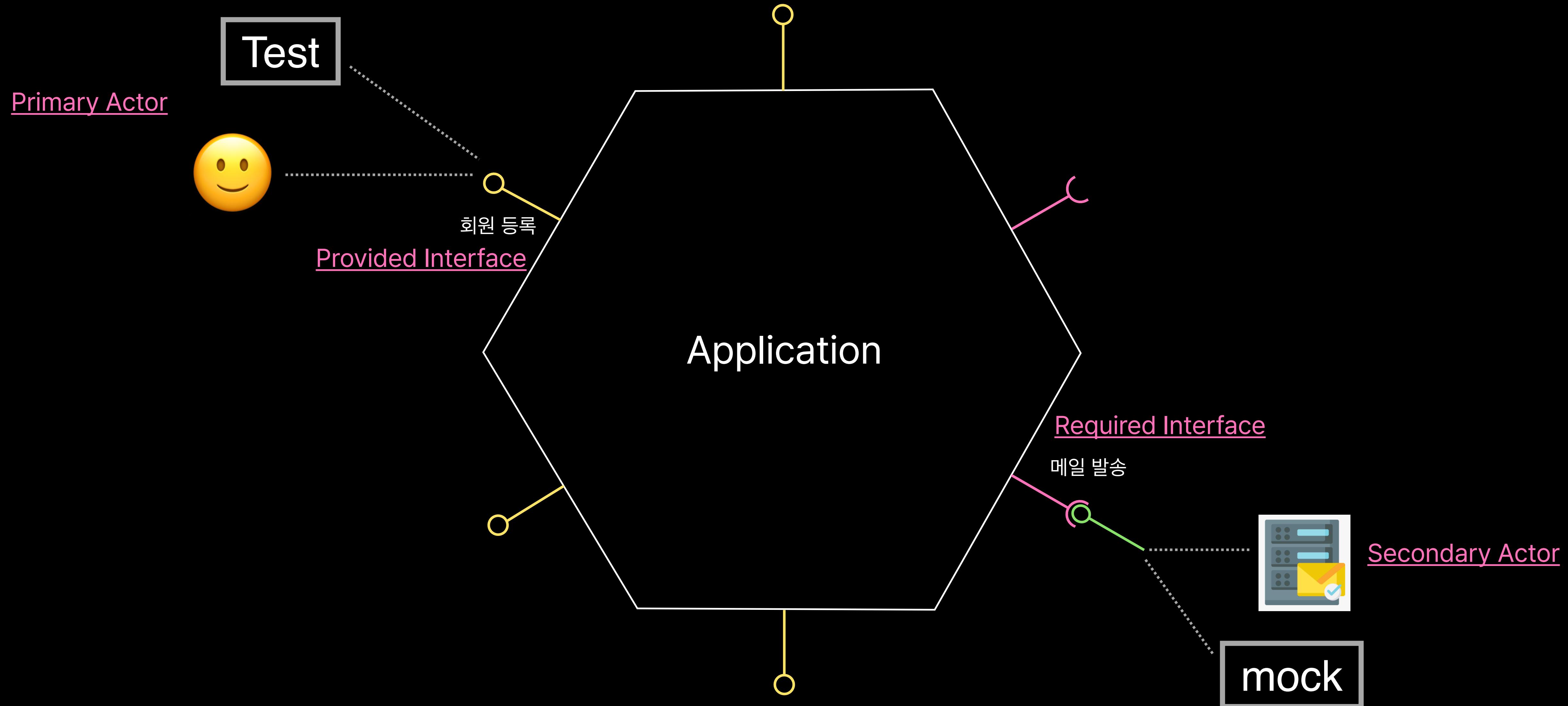
- **Lollipop: Provided Interface**
- **Socket: Required Interface**



기능 제공 인터페이스(Provided Interface)



기능 요구 인터페이스(Required Interface)



어댑터(Adapter)

- 애플리케이션의 포트를 액터가 직접 연결할 수 없다면 인터페이스의 변환을 위한 어댑터를 도입
- 브라우저를 통해서 애플리케이션의 회원 가입 포트의 기능 제공 인터페이스를 사용하려면?
 - 회원 가입 기능 제공 인터페이스를 사용하는 웹 컨트롤러 어댑터를 만든다
- 애플리케이션이 가진 회원 정보 저장 포트의 기능 요구 인터페이스로 DB와 직접 연결할 수 없다면?
 - 기능 요구 인터페이스를 구현한 리포지토리 어댑터를 만든다



포트와 어댑터 아키텍처

- 헥사고날 아키텍처의 특징을 담은 새로운 이름
- 하지만 여전히 헥사고날 아키텍처라는 이름이 더 많이 사용된다

헥사고날 아키텍처의 비대칭성

- 애플리케이션이 제공하는 기능을 사용하는 액터와 이를 위한 어댑터
 - **primary actor, primary adapter**
 - **driving actor, driving adapter**
- 애플리케이션이 동작하는데 필요한 기능을 제공하는 액터와 이를 위한 어댑터
 - **secondary actor, secondary adapter**
 - **driven actor, driven adapter**

오해: 애플리케이션 내부에 도메인 계층을 만들어야 한다

- 헥사고날 아키텍처는 애플리케이션 **내부 구현에 대한 원칙이나 요구사항이 없다**
 - 스파게티 코드로 만들어도 된다
 - 트랜잭션 스크립트, 도메인 모델 패턴과 애플리케이션 서비스 등도 가능하다
 - 도메인 계층을 포함하는 아키텍처는 클린 아키텍처이다
- 헥사고날 아키텍처는 클린 아키텍처, 어니언(onion) 아키텍처가 아니다

오해: 헥사고날 아키텍처 패키지 구조를 따라야 한다

- 헥사고날 아키텍처가 요구하는 패키지 구조는 없다
- 애플리케이션과 어댑터 패키지를 분리하는 것은 바람직하다
- 포트를 구분된 패키지에 두는 것을 권장한다

오해: 포트는 UseCase라는 접미사를 사용한다

- 포트의 의도를 담은 이름을 사용하면 된다
- For+~ing 스타일의 권장 네이밍이 있지만 이를 따를 필요는 없다

오해: 애플리케이션에는 도메인 모델만 넣고 JPA 엔티티 등은 어댑터에 둬야 한다

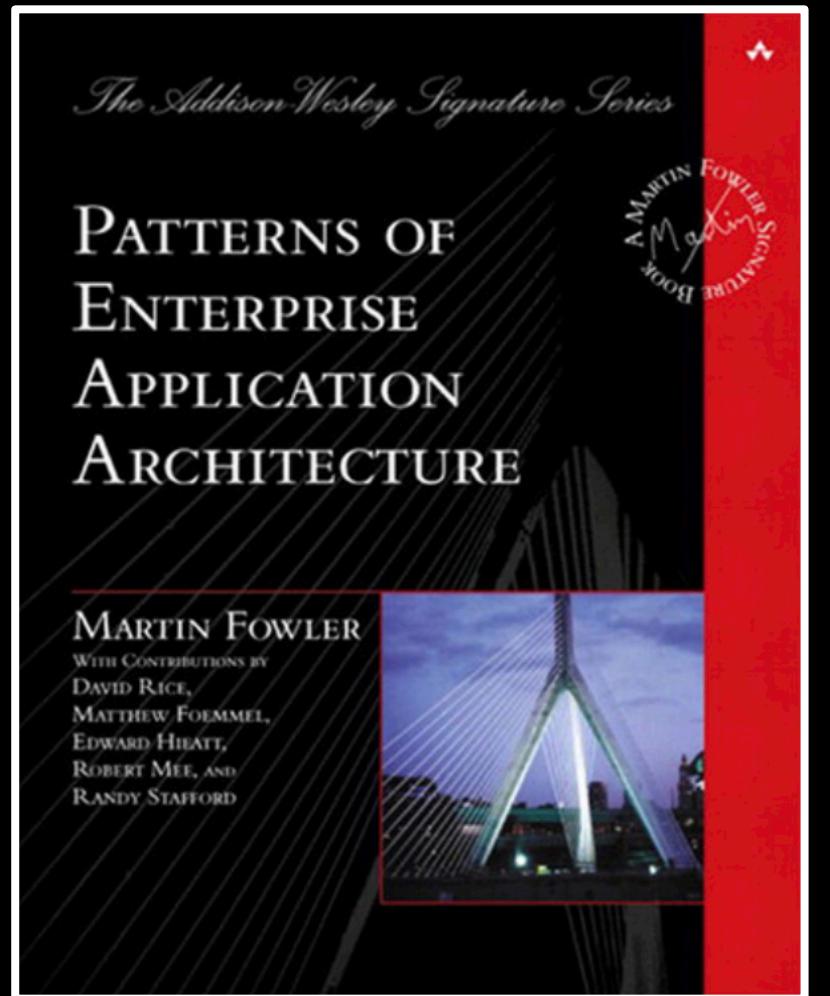
- 애플리케이션 코드와 포트 인터페이스가 외부 기술에 의존하지 않으면 된다
- 이건 할 말이 많습니다

사실: 헥사고날 아키텍처가 요구하는 것

- 애플리케이션은 모든 외부와의 상호작용을 위해서 **provided interface**와 **required interface**를 정의한다
- 애플리케이션과 상호작용 하는 액터는 런타임에 구성되어야 한다
- 애플리케이션은 액터에 대한 코드 의존성을 가지면 안 된다
- 액터는 정의된 포트를 통해서만 연결해야 한다
- 포트의 인터페이스에는 기술 의존성을 가지지 않는다

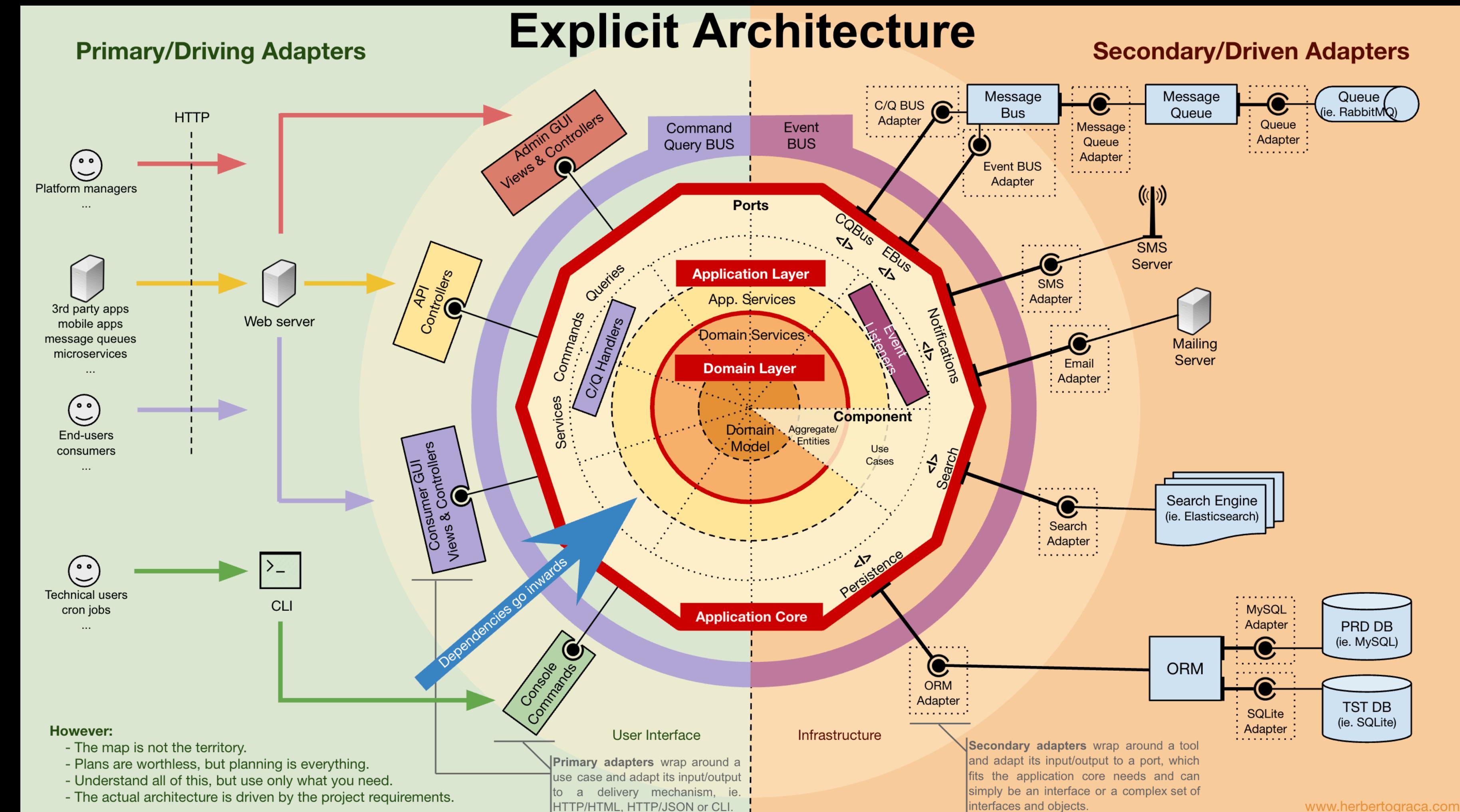
Splearn 개발에 적용할 아키텍처 패턴

- 헥사고날 아키텍처
- 도메인 모델 패턴



헥사고날과 도메인 모델 패턴을 적용한 대칭형 계층 구조

- 도메인 계층
- 애플리케이션 계층
- 어댑터 계층



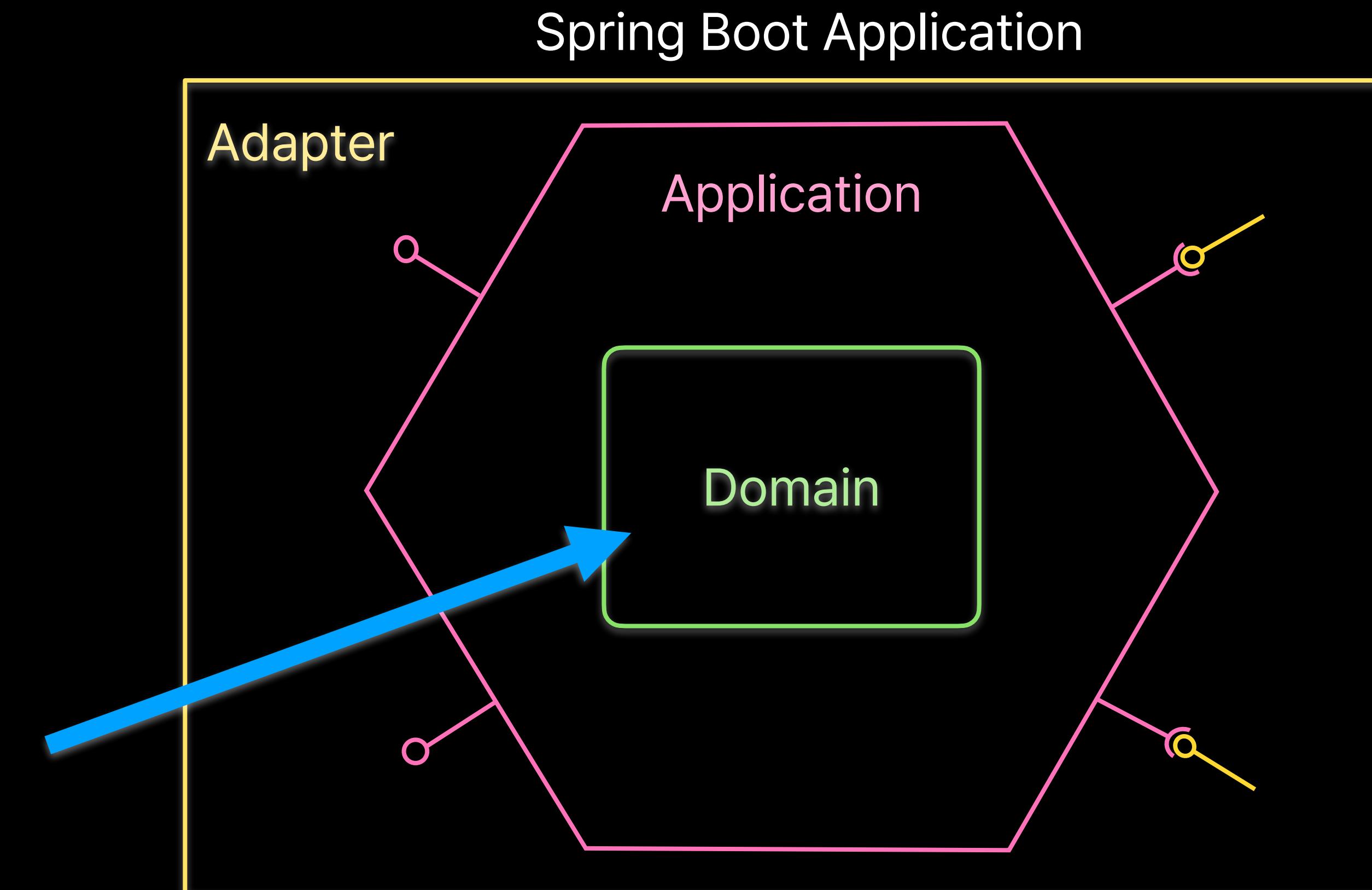
헥사고날과 도메인 모델 패턴을 적용한 대칭형 계층 구조

- 외부에서 내부로 향하는 일종의 계층 구조

- 코드의 의존 방향은 내부로만 향한다

[어댑터] -> [애플리케이션] -> [도메인]

- 단, 사용의 흐름은 비대칭적이다



도메인 주도 개발(DDD)의 아키텍처는?

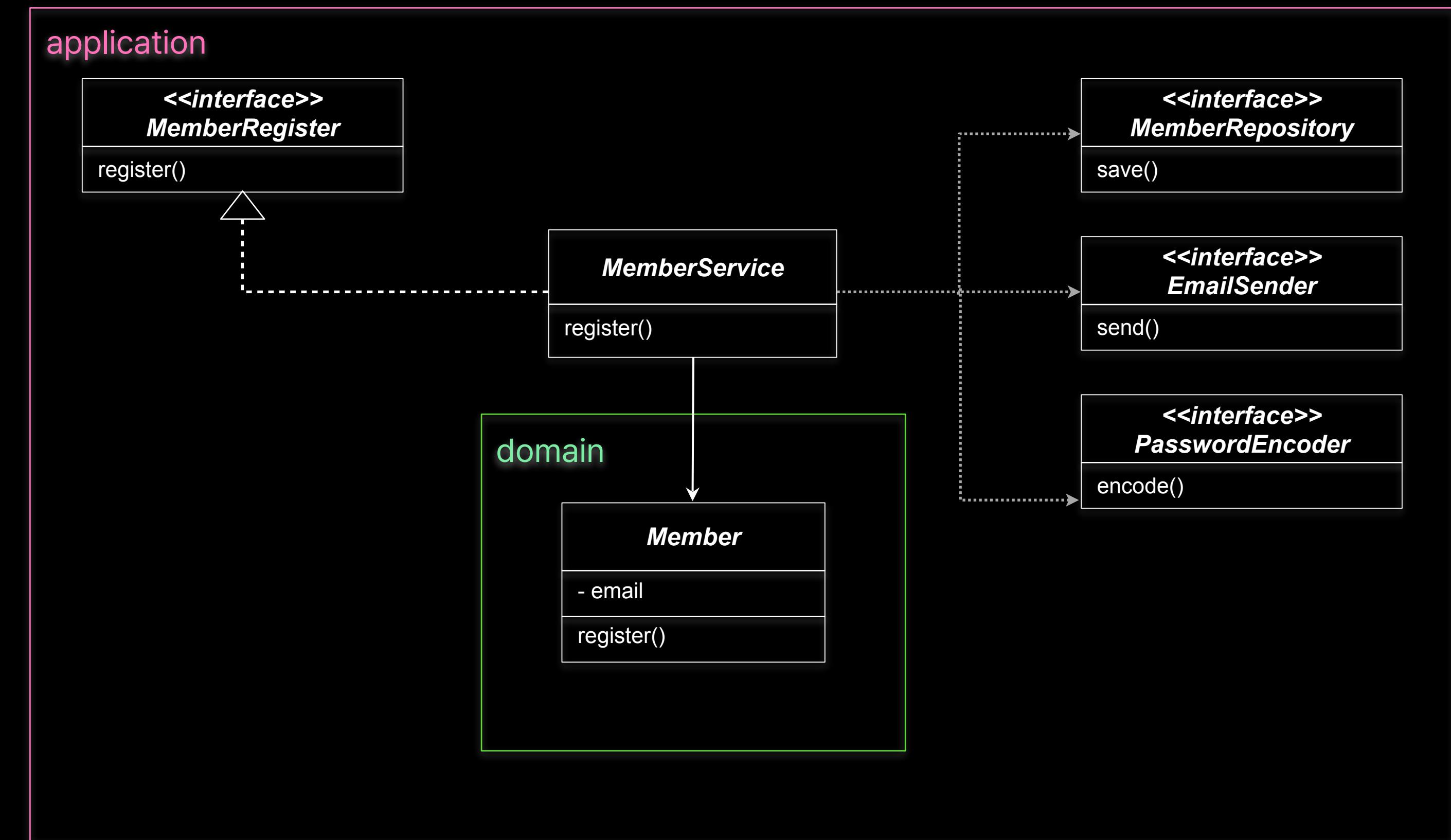
- 4개의 계층형 아키텍처: Eric Evans
- 헥사고날 아키텍처: Vaughn Vernon

회원 애플리케이션 서비스 개발

부제

엔티티 식별자

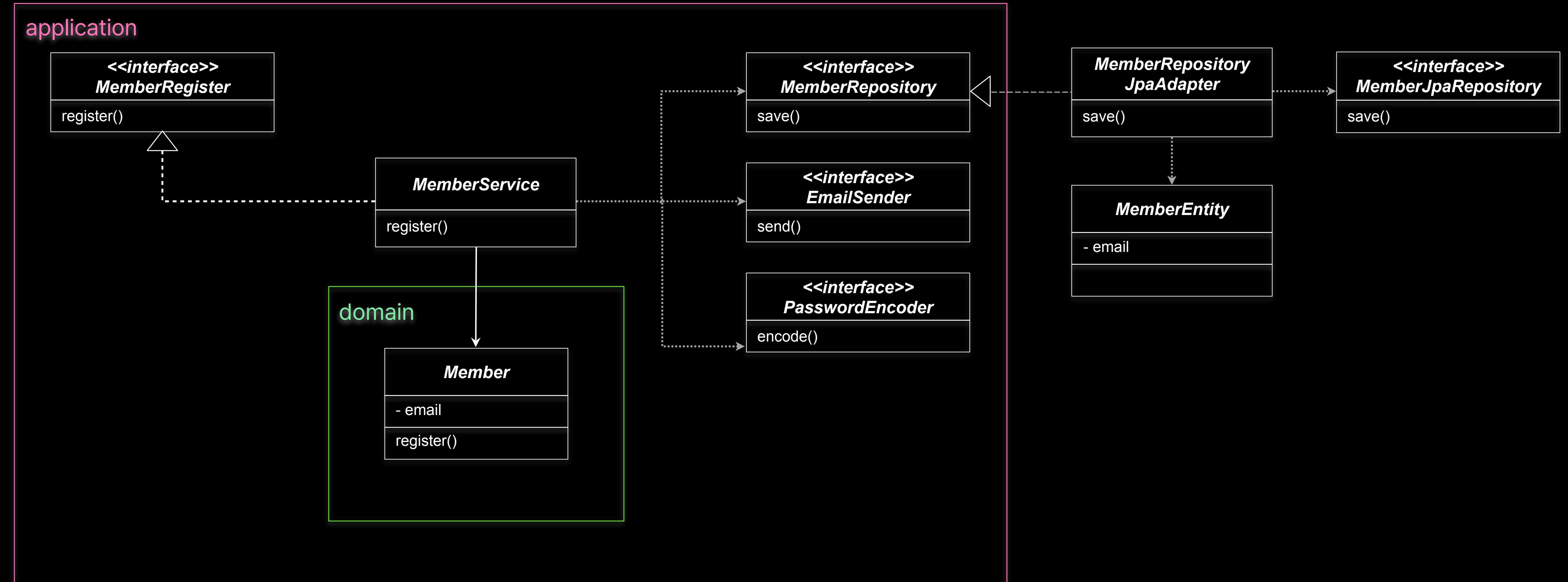
- 고유성: 두 개의 엔티티가 같은 값을 가지면 안 된다
- 불변성: 한 번 값이 할당되면 엔티티의 생명주기 동안 절대 변경되면 안 된다
 - 참조 정합성을 훼손하고, 연관관계를 깨뜨린다
- 비즈니스적인 의미로부터 디커플링 되는 것이 낫다
 - 자연 키 대신 인조 키(대리 키) 사용



JPA와 도메인 모델 패턴

JPA 모델과 도메인 모델은 다른 것인가?

- 도메인 모델은 DB와 매핑되는 데이터 모델과 다르며 이를 분리해야 한다
 - 어댑터 계층에 JPA 엔티티 등의 모델을 따로 만들고
 - Repository를 구현한 어댑터를 이용해서 도메인 오브젝트와 JPA 오브젝트를 매핑해준다
 - Member외에 MemberEntity 클래스를 만들어 JPA 관련 설정은 모두 이곳으로 옮긴다
 - MemberRepository를 구현한 MemberRepositoryJpaAdapter에서 이 두가지 오브젝트를 서로 매핑해주는 코드를 작성한다



이런 접근 방법을 선호하는 이유

- 데이터 모델과 도메인 모델이 너무 다른 경우
 - 레거시 DB에 도메인 모델 설계를 적용하는 경우
- 복잡한 도메인 모델이 데이터 모델과 간단히 매핑되지 않는 경우
 - JPA 모델과는 다른 도메인 모델이 존재한다면
- 데이터 저장 기술이 바뀌는 경우
 - Spring Data 프로젝트 존재 이유
- 코드에 등장하는 JPA 애노테이션은 기술 의존적이니까
 - 애노테이션이 코드의 실행에 영향을 줄까?
- 도메인 코드에 관심사가 다른 JPA 매핑 애노테이션, DB 정보가 들어가니까
 - 가독성 또는 취향

JPA 기술의 정체성

- ORM: 패러다임이 다른 관계형 DB와 객체지향 모델의 불일치를 해결하는 기술
 - SQL 매피ング과는 다르다
- JPA의 기술적 목표는 자바 애플리케이션 개발자가 관계형 데이터베이스를 관리하기 위해 자바 도메인 모델을 활용할 수 있는 객체/관계 매피ング 기능을 제공하는 것
- JPA의 엔티티는 경량 영속 도메인 오브젝트

도메인 모델 패턴

- 단순 도메인 모델은 테이블과 클래스가 1:1로 매핑된다
- 복잡한 도메인 모델은 DB 매핑이 어렵다는 문제가 있다
 - 이걸 해결해주는 것이 JPA(ORM) 기술
- JPA(ORM)이 매핑을 통해서 해결하려는 패러다임 불일치 문제
 1. 세분성(Granularity) 불일치
 2. 상속(Subtype) 불일치
 3. 정체성(Identity) 불일치
 4. 연관(Association) 불일치
 5. 데이터 탐색(Navigation) 불일치

스프링 데이터 프로젝트

- 다양한 데이터 저장소(관계형 또는 비관계형 DB, 클라우드 기반 데이터 서비스)에 대한 데이터 접근을 단순하고 일관된 프로그래밍 모델로 제공
 - 일관된 프로그래밍 모델: 저장소의 종류와 관계없이 동일한 방식으로 데이터에 접근하도록 한다
 - 보일러 플레이트 코드 감소
 - 데이터 저장소의 특성 유지
 - 확장성과 유연성
- Repository<T, ID>
 - T: 도메인 타입 = 엔티티 = 애그리거트 루트

도메인 모델과 JPA 모델을 반드시 분리해야 한다는 주장에 대한 반박

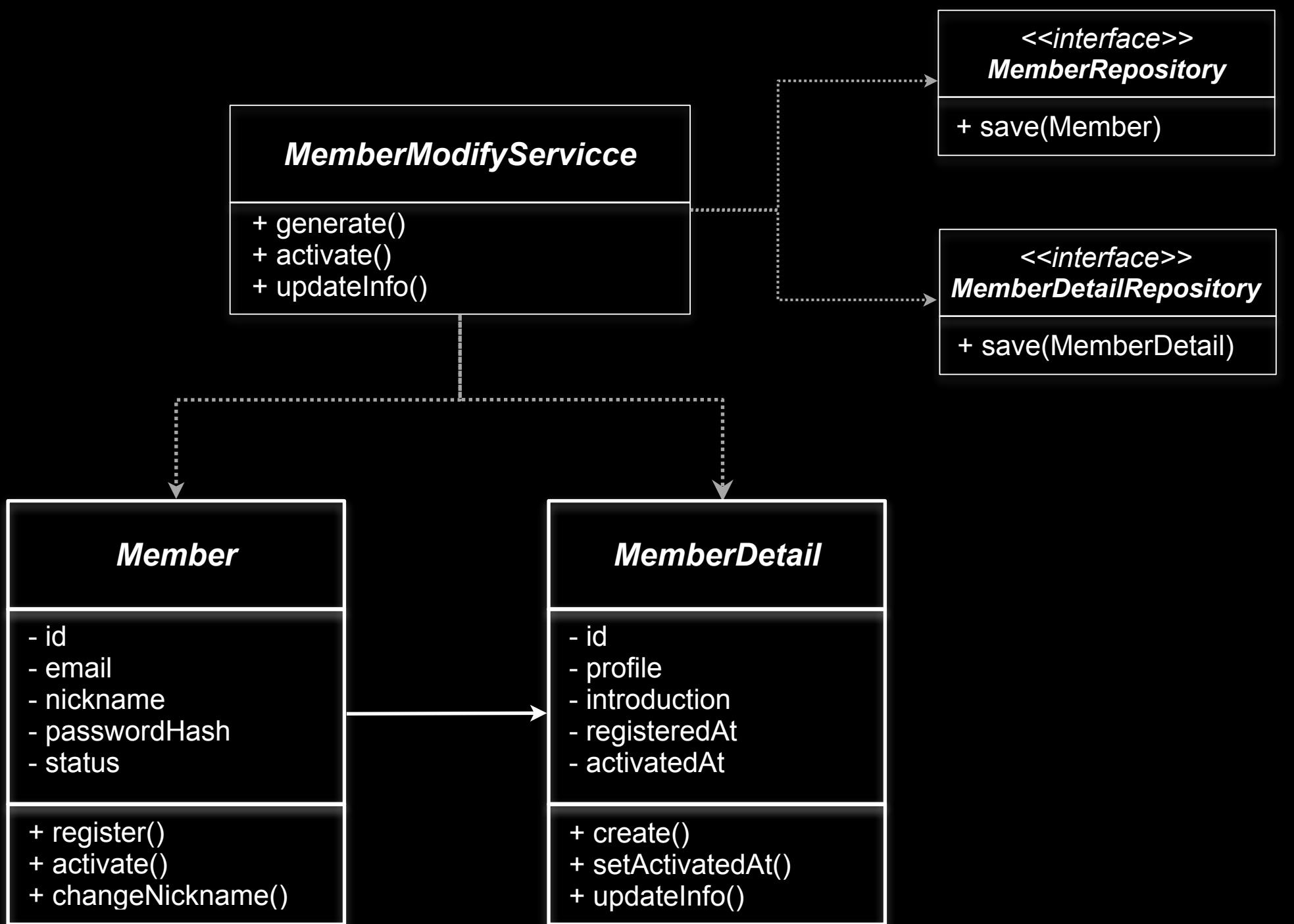
- 대부분 데이터 모델과 도메인 모델이 **다르지 않음**
- 복잡한 도메인 모델의 매핑은 **JPA가 충분히 지원**
- 모델 변환 로직과 유사한 두 가지 클래스로 인한 **불필요한 복잡성 증가**
- JPA는 근본적으로 **도메인 오브젝트의 매핑**을 위해서 설계된 기술
- JPA는 도메인 계층을 침범하지 않음
- 복잡한 쿼리 로직은 **커스텀 리포지토리와 어댑터 구현**을 통해서 개발 가능
- 도메인 계층과 데이터 계층의 결합은 불가피하다
 - 이 둘이 완전히 독립적인 경우는 매우 드물다
- JPA는 충분히 유연하고 막강하며, **Spring Data JPA는 놀라운 수준의 도메인 중심 개발 지원**

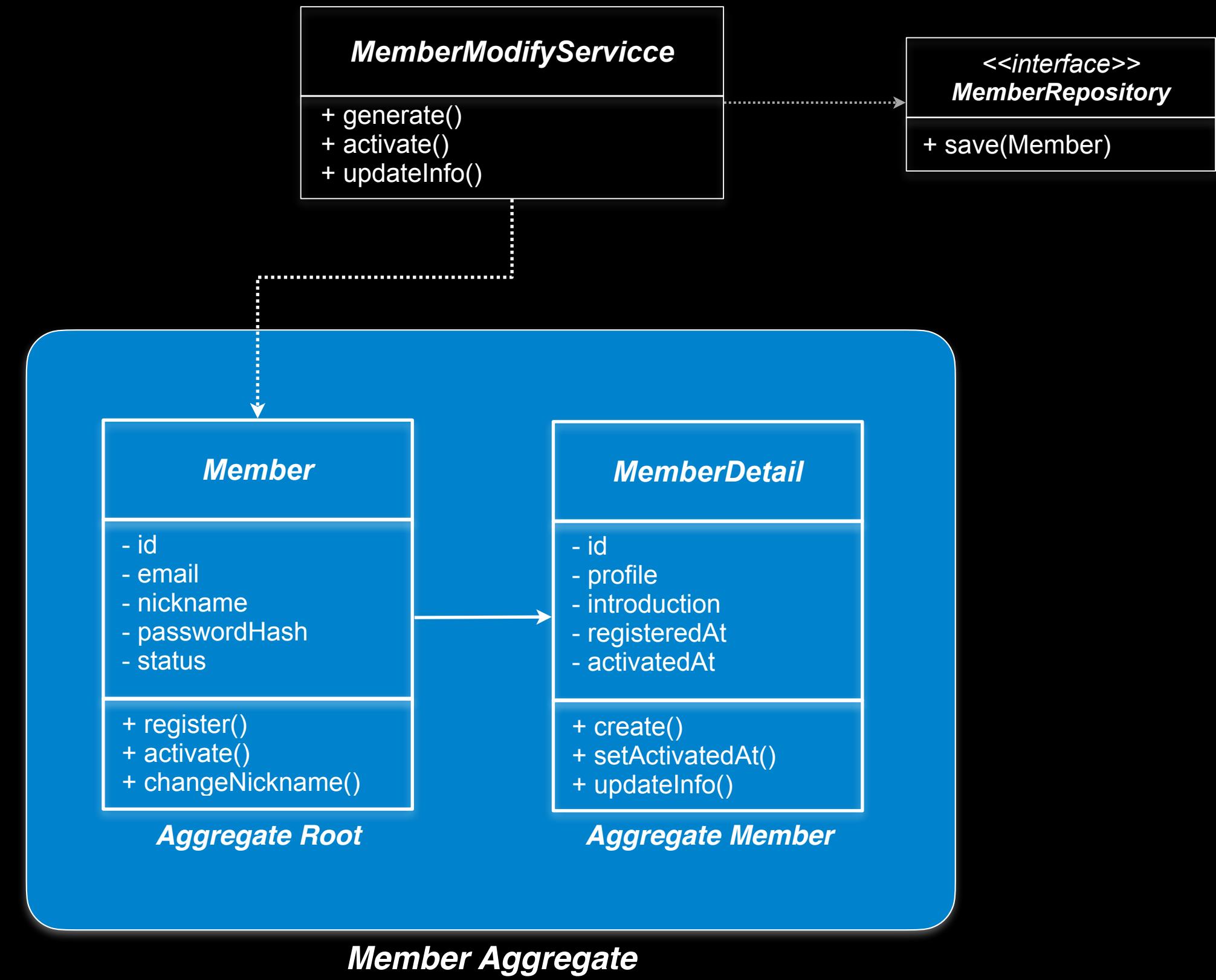
성급한 분리는 오히려 **복잡성만 초래할 뿐**이고
JPA는 **도메인 모델 중심의 개발**을 충분히 잘 지원한다

하지만 도메인 모델에 등장하는
도메인 관심사가 아닌 각종 JPA 매핑 애노테이션은
어떻게 제거할 방법이 없을까?

애그리거트를 이용한 일관성 있는 도메인 모델 설계

Aggregate





애그리거트(Aggregate)

- 도메인 주도 개발(DDD)에 소개된 도메인 모델 구성 요소/패턴의 하나이다
- 데이터 변경의 목적을 위해 하나의 단위로 취급되는 연관된 객체들의 클러스터이다
- 루트(root)와 경계(boundary)를 가진다
- 경계 내부에는 엔티티와 값 객체가 하나 또는 여러 개가 존재할 수 있다
- 애그리거트 루트는 내부에 포함된 단일 엔티티이다

애그리거트의 특징

- 데이터 변경 시 하나의 단위로 취급: 데이터 변경의 일관성을 유지한다
- 루트를 통한 접근 제어: 외부 객체는 루트 엔티티에만 참조를 가질 수 있다
- 데이터 일관성 유지:
 - 경계 내의 어떤 변경 사항이 있을 때 전체 애그리거트의 모든 불변식이 충족되어야 한다
 - 애그리거트를 넘어서는 불변식은 이벤트나 배치 등을 통해 특정 시간 내에 해결할 수 있다
- 검색 및 접근 방식: 리포지토리를 통해서 애그리거트 루트만 직접 얻을 수 있다
 - 내부 엔티티는 루트로부터 연관관계를 통해서 접근한다
- 생명주기 관리 캡슐화: Factory와 Repository를 이용해서 객체들을 생명주기에 걸쳐 체계적이고 의미있는 단위로 조작

애그리거트의 목표

- 일관성 유지: 객체 그룹에 적용되는 불변식을 유지하는 수단
- 이해 용이성: 객체의 시작과 끝을 명확히 해서 모델을 더 쉽게 이해하게 한다
- 트랜잭션 및 동시성 관리: 트랜잭션 범위와 데이터 일관성 유지 방법 제공
- 모델 단순화: 연관 관계 탐색을 제한하고 루트를 통해서만 접근하도록 해준다
- Factory와 Repository가 복잡한 생명주기 전환을 캡슐화하는 단위가 되도록 한다

애그리거트 적용 방법

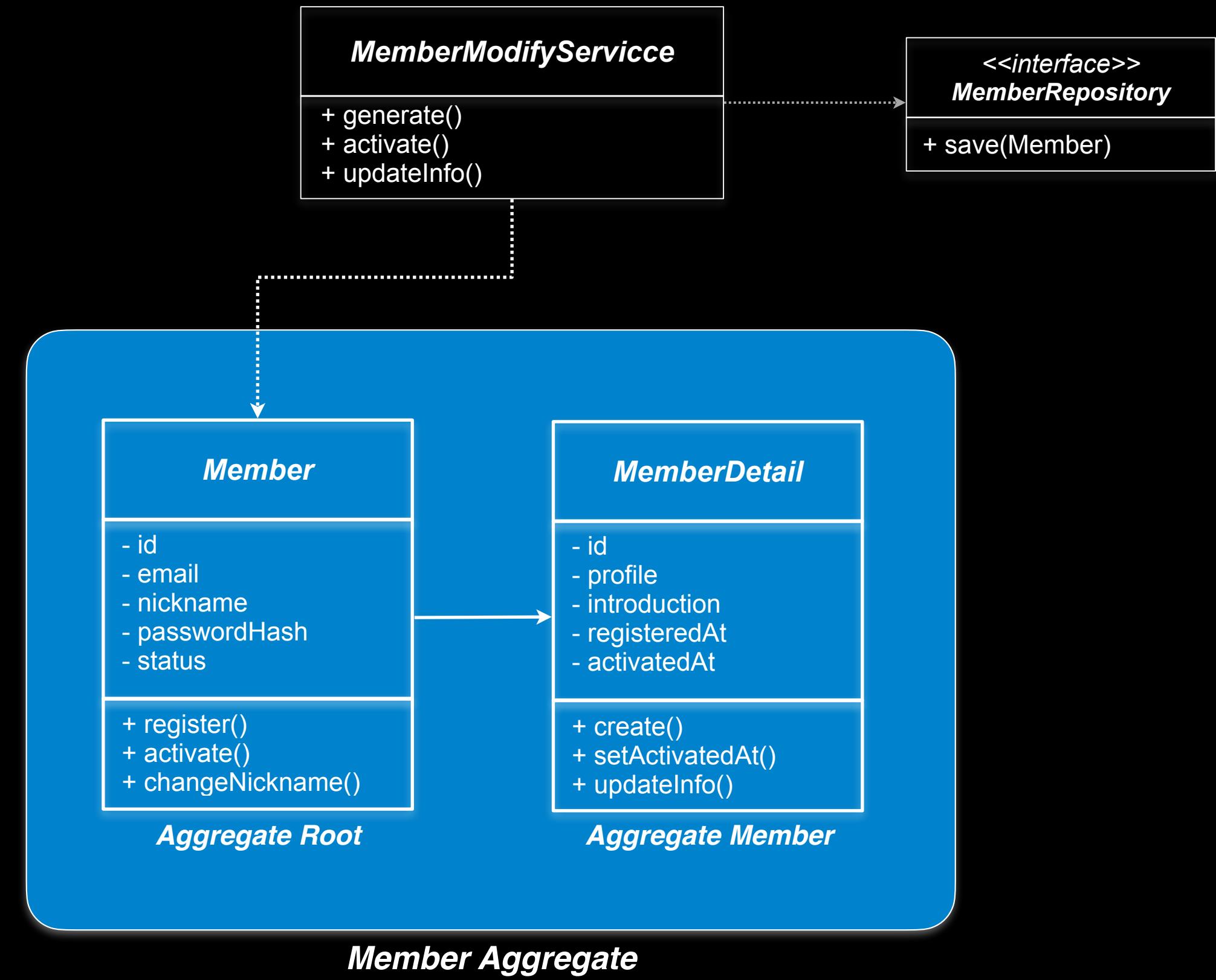
- JPA의 **cascading**을 적절하게 활용한다
- 리포지토리는 애그리거트 단위로 만든다
 - 스프링 데이터 JPA의 핵심 원칙
 - **Repository<T, ID>** : T = Aggregate Root
 - 리포지토리 리턴 타입은 애그리거트 루트
- 가능하다면 하나의 트랜잭션에 하나의 애그리거트만 변경한다
- 다른 애그리거트의 **참조는 애그리거트 루트**에 대해서만 한다
 - 연관관계 애그리거트 루트의 레퍼런스 대신 루트의 ID 값만 저장하기도 한다

애그리거트 설계와 적용의 어려움

- 적절한 애그리거트 경계를 선택하는 것은 꽤 어려운 결정이다
- 개발하면서 애그리거트의 범위가 달라지기도 한다
 - 대체로 작은 애그리거트가 될 가능성이 높다
- 성능에 부담을 주게 된다 (**lazy loading**의 도움이 필요)
- 내부 엔티티로의 직접 접근이나 여러 애그리거트를 한번에 조회하는 기능이 필요한 경우가 있다
- 도메인 이벤트와 최종적 일관성(**eventual consistency**)의 사용이 요구된다
- 완벽한 애그리거트가 아니어도 괜찮다
 - 우리는 DDD 하지 않는다고 하면 된다 😊

헥사고날 아키텍처와 애그리거트

- 애그리거트 단위로 애플리케이션(헥사곤)을 구성하는 방법이 유용하다
- 다른 애그리거트로의 접근은 애플리케이션 **포트**를 통해서 **ID**를 전달하는 방식으로
 - 애플리케이션 내부 리포지토리에서 루트 엔티티를 조회하는 방식으로 이루어지게 강제할 수 있다
- 도메인 이벤트와 리스너를 이용해서 애그리거트 사이의 작업을 연결할 수도 있다
 - 이벤트에 필요한 애그리거트 루트 ID를 전달한다
- 애그리거트를 설계하고 각각을 독립적인 애플리케이션으로 분리하자



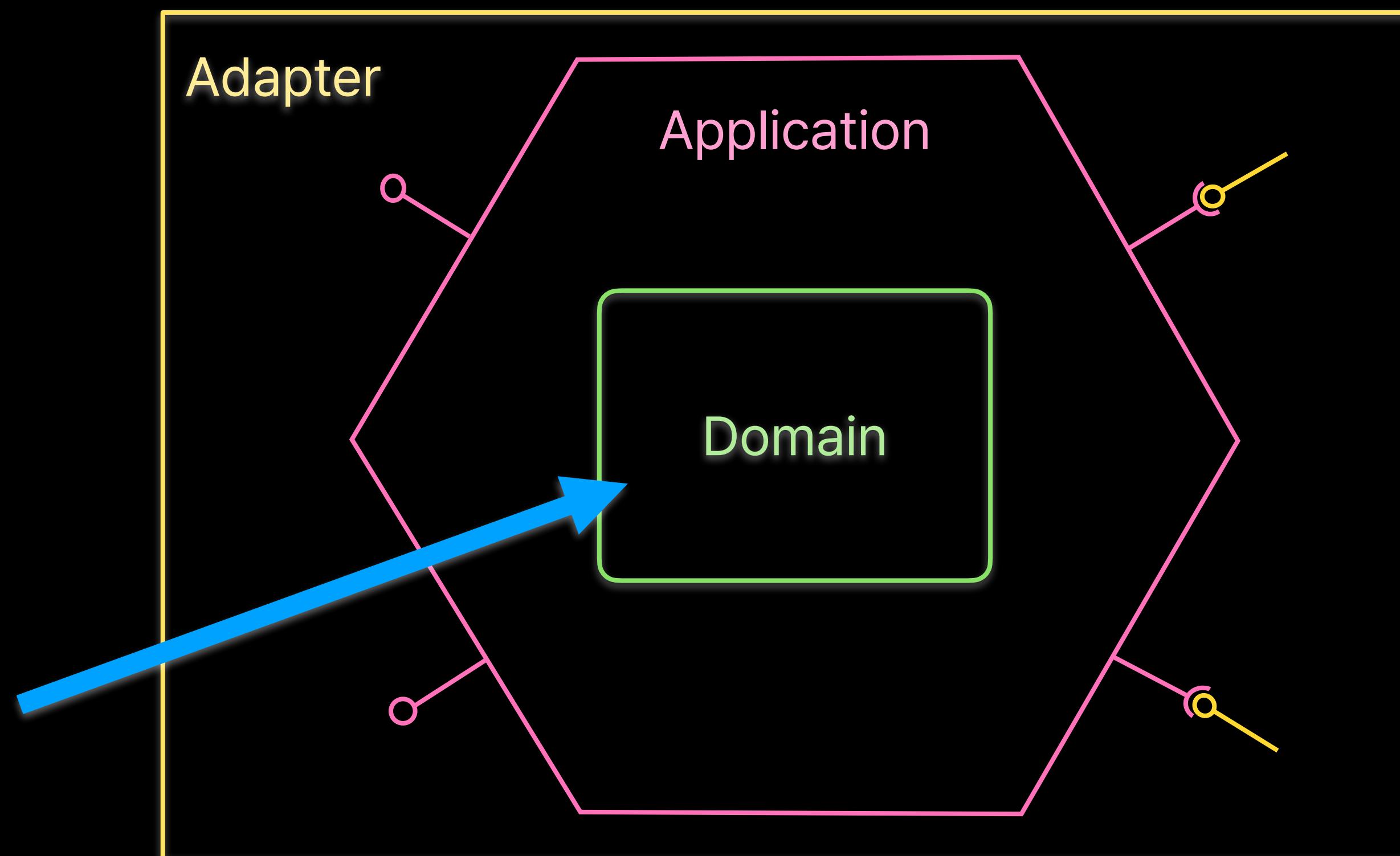
웹 API 어댑터 개발

Web API Adapter

애플리케이션 포트의 리턴 타입은? Entity vs DTO

- 애플리케이션에서 도메인 로직을 적용하고 엔티티의 정보를 돌려줄 때 어떤 타입을 사용하는가?
 - 엔티티 (애그리거트 루트)
 - DTO

Spring Boot Application



1. 애플리케이션(서비스) 계층의 리턴 타입은 DTO여야 한다?

- 계층형 아키텍처에서 각 계층은 자기 바로 **다음(하위)** 계층에만 의존해야 한다는 제약이 있다
- 도메인 계층에 존재하는 **엔티티는 애플리케이션 계층에서만 사용하고,**
그 밖으로 전달할 때는 필요한 값만 뽑아서 **DTO로 만들어서 전달해서** 아키텍처 제약을 따른다
- 도메인 로직을 가지고 있는 엔티티가 프레젠테이션 로직을 가진 어댑터(컨트롤러)로 유출되면
도메인 로직을 담은 메소드를 실행할 수 있는 문제가 있다
 - OSIV와 결합하면 컨트롤러에서 DB 변경이 일어날 수도 있다
- 헥사고날 아키텍처(또는 클린 아키텍처, 계층형 아키텍처)에서 도메인/애플리케이션 계층 밖으로
보낼 때는 **반드시 DTO에 담아서 전달한다**

2. 애플리케이션(서비스) 계층의 리턴 타입은 가능하다면 엔티티로 한다

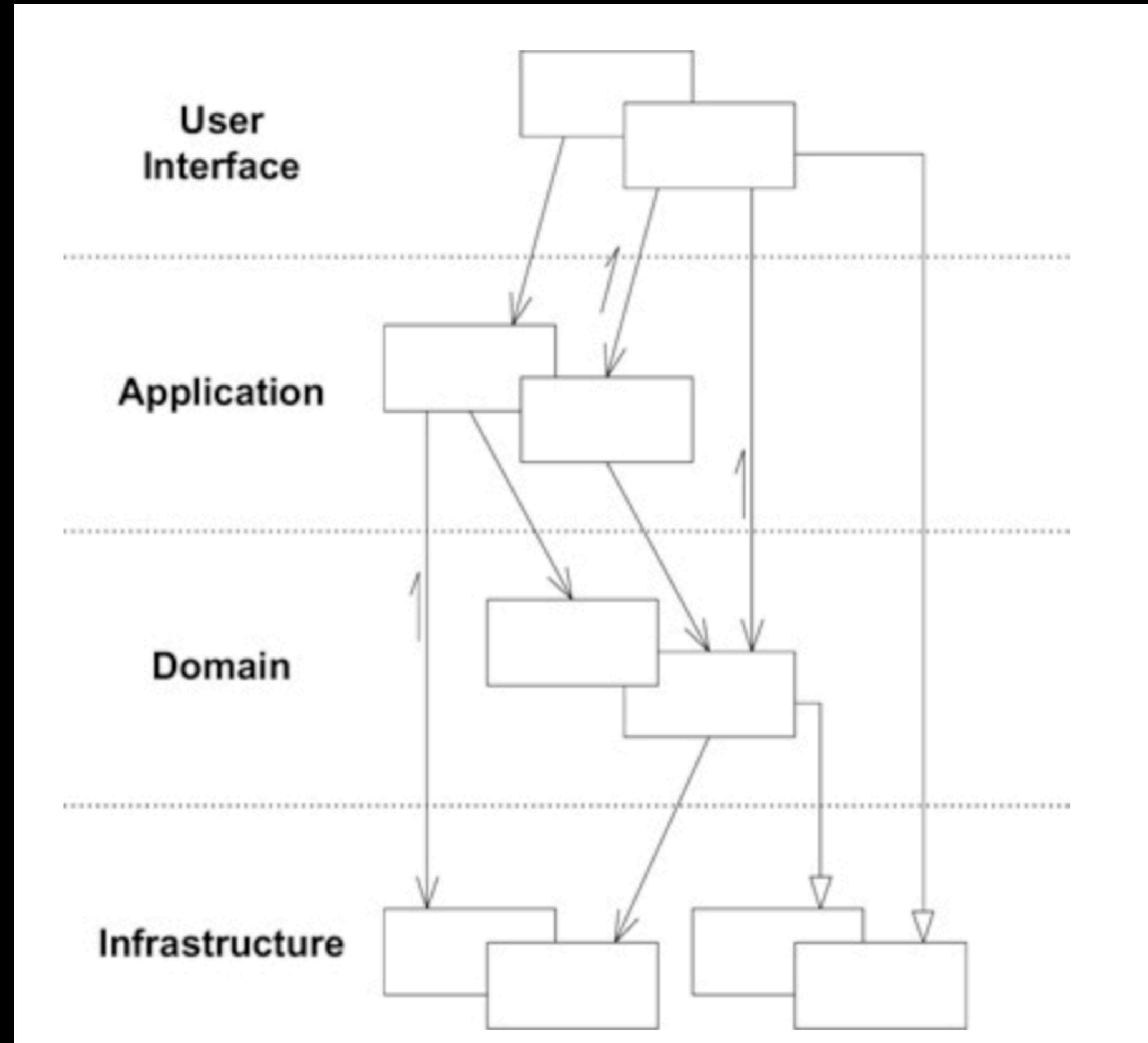
- 계층형 아키텍처에서 각 계층은 자기 바로 다음 계층에만 의존해야 한다는 제약이 있다
- 현대적인 엔터프라이즈 애플리케이션은 이 제약을 완화시킨

완화된 계층형 아키텍처(relaxed layered architecture)를 주로 사용한다

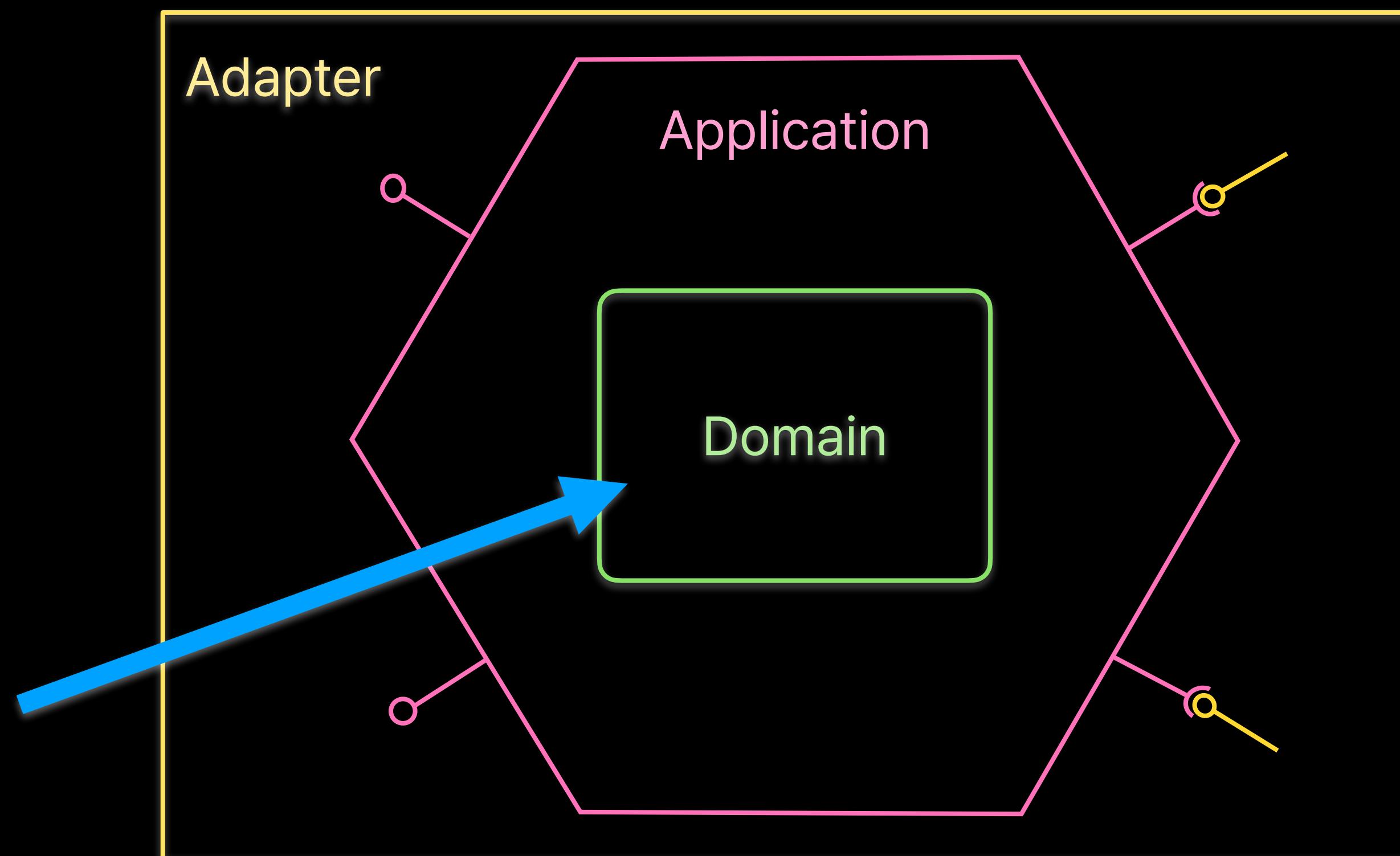
- 계층형 아키텍처의 목적은 시스템의 관심사를 분리(separation of concerns) 하는 것이다
- 이를 통해 전체적인 결합도를 낮추고 인지 과부하를 방지해서 재사용성과 유지보수성을 높인다

2. 애플리케이션(서비스) 계층의 리턴 타입은 가능하다면 엔티티로 한다

- 완화된 계층형 아키텍처(relaxed layered architecture)를 사용한다



Spring Boot Application



2. 애플리케이션(서비스) 계층의 리턴 타입은 가능하다면 엔티티로 한다

- 도메인 로직을 가지고 있는 엔티티가 프레젠테이션 로직을 가진 어댑터(컨트롤러)로 유출되면 도메인 로직을 담은 메소드를 실행할 수 있는 문제가 있다
- 엔티티가 가진 정보 조회를 넘어서 엔티티 로직을 사용하므로 도메인 로직이 유출되는 문제는 엔티티에서만 발생하는 것이 아니다.
DTO를 가지고도 얼마든지 컨트롤러에서 "도메인 로직"을 구현할 수 있다.
- 어짜피 세컨더리 어댑터에서도(리포지토리 구현 어댑터, 메일 발송 어댑터)에도 엔티티가 전달되거나 생성되어 리턴된다.
- 근본적으로 리포지토리 어댑터에서 도메인 엔티티를 제거할 방법은 없다. 여기서 도메인 로직을 실행할 수 있는 것은 문제가 아닌가?

2. 애플리케이션(서비스) 계층의 리턴 타입은 가능하다면 엔티티로 한다

- OSIV와 결합하면 컨트롤러에서 DB 변경이 일어날 수도 있다
- OSIV가 적용됐다고 하더라도 애플리케이션에서 리턴할 때 트랜잭션이 종료된다
- 따라서 이후의 엔티티 변경사항은 DB에 커밋되지 않는다

2. 애플리케이션(서비스) 계층의 리턴 타입은 가능하다면 엔티티로 한다

- DTO를 리턴하는 방식의 가장 큰 문제는 **프레젠테이션 로직이 애플리케이션 레이어로 침투**하는 것이다
- Web API에서 요구하는 정보가 무엇인지, 즉 **뷰(view)** 로직에 따라 엔티티에서 복제되는 DTO의 구성이 달라진다. DTO와 이를 매핑하는 코드는 모두 애플리케이션 계층이다
- 따라서 DTO를 통해 애플리케이션 계층이 어댑터 계층의 로직에 의존한다
- 상위(외부) 계층의 로직을 하위(내부) 계층이 의존하는 것은 계층형과 혁사고날 아키텍처 모두의 근간을 깨는 행위이다
- 엔티티(애그리거트 루트)를 돌려주고 어떤 정보를 사용자에게 전달할지 선택하는 **뷰** 로직은 **프레젠테이션 관심사를 가진 웹 API 어댑터**가 담당한다
- 엔티티 단위가 아닌 복잡한 **리포트성 조회 결과는 DTO를 리턴**한다. 이 자체가 도메인 로직이다

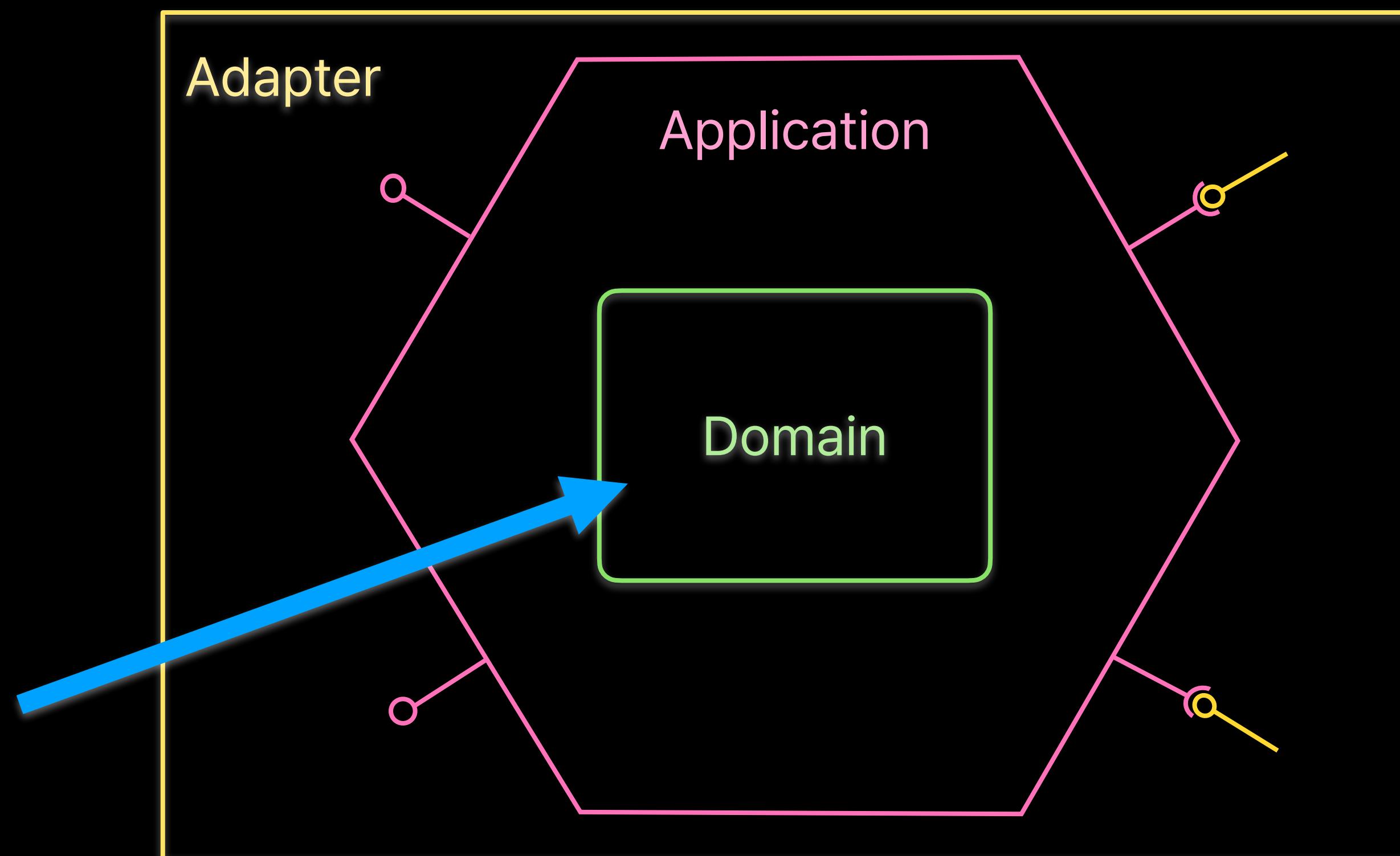
2. 애플리케이션(서비스) 계층의 리턴 타입은 가능하다면 엔티티로 한다

- 개발 가이드와 코드 리뷰를 통해서 내부의 로직이 외부에 새어나가는 코드를 작성하지 않도록 주의한다
- 정적 분석 도구나 아키텍처 테스트의 도움을 받아서 엔티티의 조회 메소드 외의 메소드가 사용되는 것을 체크할 수도 있다

2. 애플리케이션(서비스) 계층의 리턴 타입은 가능하다면 엔티티로 한다

- OSIV는 프레젠테이션/API 로직이 애플리케이션과 세컨더리 어댑터 계층(리포지토리 구현)까지 결합되는 것을 막아주는 좋은 도구이다
- 하지만 대용량 트래픽과 높은 성능을 위한 최적화가 요구되는 경우에는 사용하지 않는 것이 좋다 DB 커넥션과 JPA 영속 컨텍스트가 유지되는 것이 영향을 줄 수 있기 때문이다
- 이럴 때는 엔티티 리턴했을 때 간혹 lazy loading이 실패하는 문제를 막기 위해 최소한의 선제적인 조회 작업이 필요할 수도 있다
- 애그리거트 한번에 로딩, 이거 폐칭, 폐치 조인, 엔티티 그래프의 사용
Hibernate를 이용한 프록시 데이터 초기화 수행 등

Spring Boot Application



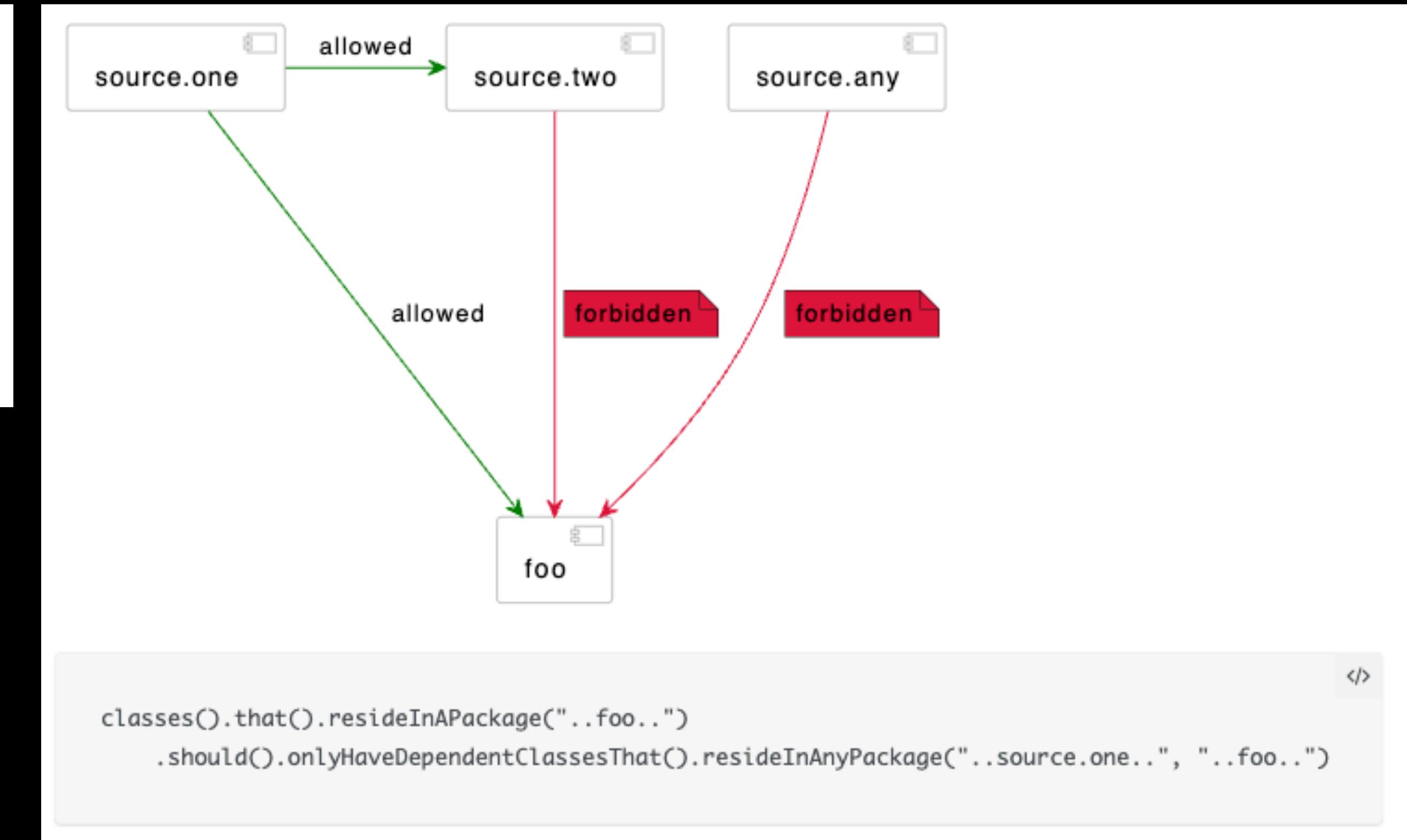
애플리케이션 기능 제공 엔터페이스의 리턴 타입은 엔티티와 애그리거트,
혹은 그 이상의 연관관계로 애플리케이션 로직 수행 결과를 리턴하기
어려운 경우에만 DTO 클래스/레코드를 만들어 사용한다

아키텍처 테스트

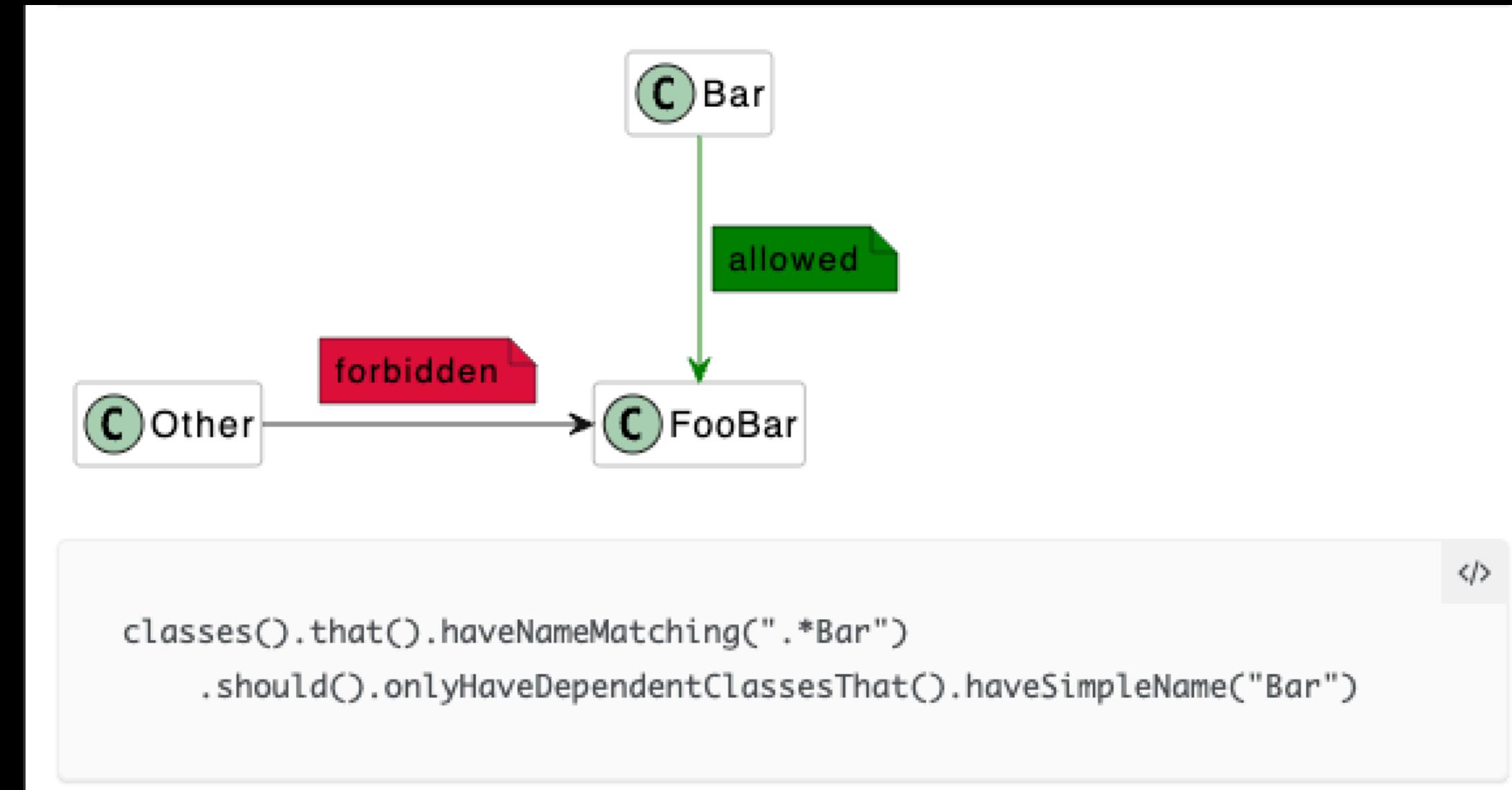
ArchUnit

- 자바 아키텍처를 검증하는 단위 테스트 도구
- 테스트 코드에 아키텍처 규칙을 선언하고
- 지정된 패키지, 클래스, 레이어, 슬라이스 사이의 의존 관계가 이 규칙에 맞는지 확인한다
 - 계층형 아키텍처의 의존관계
 - 패키지와 클래스의 복잡한 순환 의존성
 - 클래스나 메서드 이름의 형식
 - 애노테이션의 적용 패턴
 - 의존관계와 사용관계를 구분해서 검증

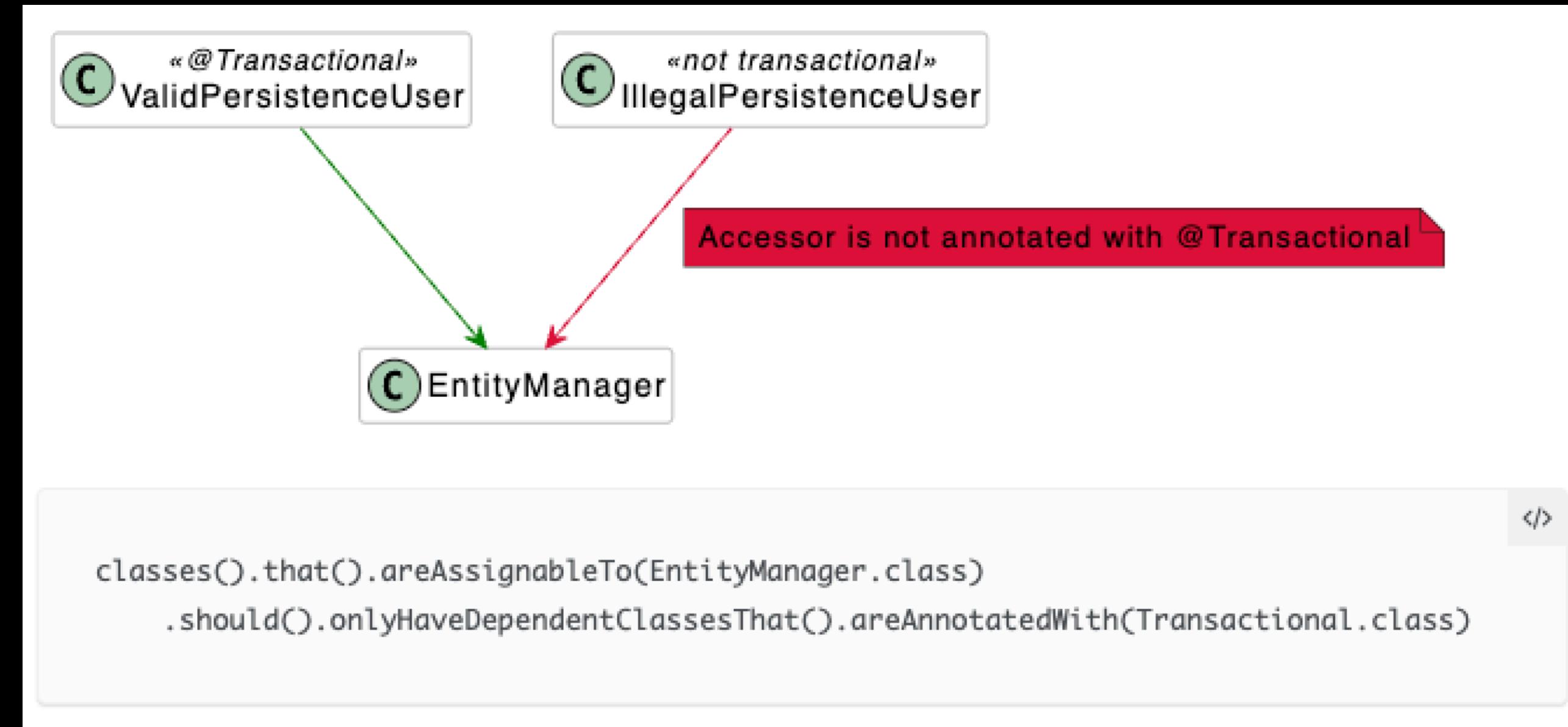
패키지 의존관계 확인



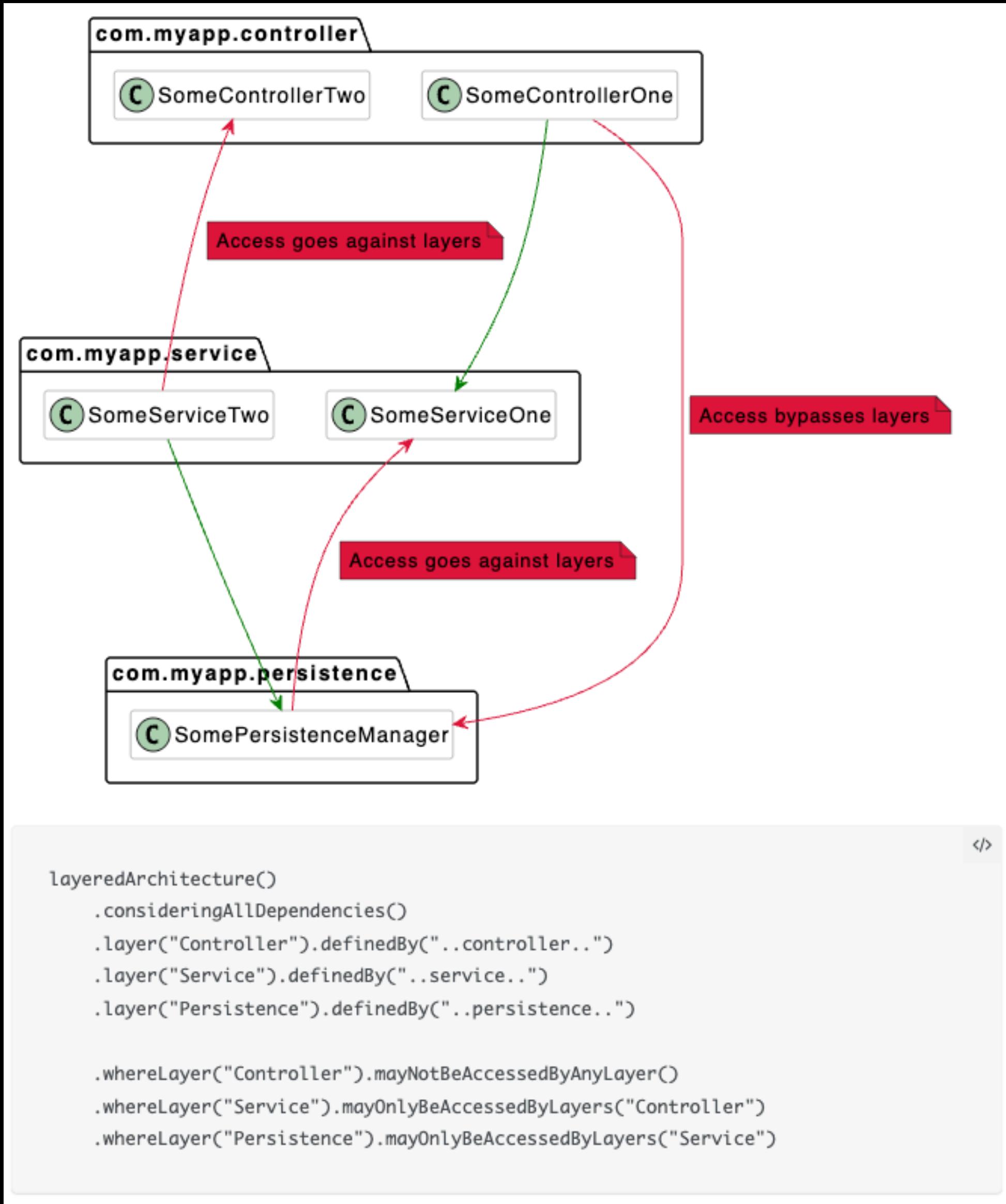
클래스 의존관계 확인



애노테이션 확인



계층 의존관계 확인



학습 테스트 Learning Test

- 테스트 주도 개발(**TDDBE**), 클린 코드에서 소개된 테스트
- 테스트 코드를 통해서 라이브러리, 레거시 시스템과 같은 외부 코드의 사용방법을 익히는 기법
- 새로운 기술의 사용방법에 대한 코딩 가이드 역할을 해준다
- 외부 코드의 버전이 바뀌었을 때 기존 사용방식에서 변경이 필요한지도 확인이 가능하다
- 학습 테스트에 투자하는 노력보다 얻는 성과가 크다