

## 2. 전체 계층 구조 개요 (Layered Architecture)

### 계층형 아키텍처란?

#### 1. 개념 정의

계층형 아키텍처란, 애플리케이션을 기능이나 책임에 따라 수직적으로 계층화하여 분리하는 소프트웨어 아키텍처 패턴이다. 각 계층은 자신보다 하위 계층에만 의존하고, 상위 계층과는 의존하지 않도록 설계된다.

- 목적: 책임의 분리 (Separation of Concerns)
- 장점: 유지보수성, 테스트 용이성, 확장성, 재사용성

#### 2. 기본 계층 구성 (5계층 구조 기준)

계층명	대표 어노테이션/구성	주요 책임
프레젠테이션 계층 (Controller)	@Controller, @RestController	사용자 요청 수신, 응답 반환, UI 또는 API 출력
애플리케이션 계층 (UseCase Layer)	Application Service (선택)	유스케이스 실행, 도메인 객체 호출, 외부 연동 조합
서비스 계층 (Business Logic)	@Service	도메인 조합, 트랜잭션 관리, 외부 API/DAO 처리
도메인 계층 (Entity, VO)	@Entity, 도메인 서비스	핵심 비즈니스 규칙 보유, 도메인 상태 변경
데이터 접근 계층 (Repository, DAO)	@Repository, JpaRepository	DB 접근, 영속성 처리, 쿼리 수행

#### 3. 계층 간 흐름 예시

1 | 사용자 요청 → Controller → Application/Service → Domain → Repository → DB

##### 예시: 주문 생성 흐름

```
1 | OrderController
2 |   ↓
3 | OrderService (비즈니스 트랜잭션, 재고 체크)
4 |   ↓
5 | Order (도메인 엔티티, 상태 변경)
6 |   ↓
7 | OrderRepository (DB 저장)
```

## 4. 의존성 방향 원칙

- 상위 계층 → 하위 계층 으로만 의존함.
- 하위 계층 → 상위 계층 의존은 금지.  
(→ 의존 역전이 필요한 경우, 인터페이스 주입 등으로 해결)

1	✗ Repository → Service
2	✓ Service → Repository

## 5. 계층형 아키텍처의 장점

장점	설명
🎯 모듈화	각 계층을 독립적으로 수정 및 교체 가능
🎯 테스트 용이성	단위 테스트 작성이 쉽고 의존성 주입 구조에 적합
🎯 유지보수성	기능별 책임이 분리되어 이해하기 쉬움
🎯 확장성	기능 추가 시 계층별로 명확히 위치시킬 수 있음

## 6. 계층형 아키텍처의 단점

단점	설명
🏢 수직적 구조로 인한 호출 복잡성	모든 요청이 위→아래로 단계별 통과해야 하므로 느릴 수 있음
📄 지나친 계층 분리는 과도한 보일러플레이트	간단한 기능도 여러 계층 거쳐야 함
⚠️ 도메인과 애플리케이션 로직이 섞일 위험	역할 구분이 흐려지면 유지보수가 어려워짐

## 7. 계층형 아키텍처 vs 다른 구조

아키텍처	특징
Layered (계층형)	가장 기본, 단순 명료
Hexagonal (Ports & Adapters)	입출력과 도메인을 완전 분리
Clean Architecture	도메인 중심 설계, 의존성 완전 반전
Onion Architecture	도메인을 중심으로 원형 구조

## 8. Spring에서 계층 예시

```
1 // Controller (Presentation Layer)
2 @RestController
3 public class OrderController {
4     private final OrderService orderService;
5     public OrderController(OrderService orderService) { this.orderService =
6         orderService; }
7
8     @PostMapping("/orders")
9     public ResponseEntity<Void> placeOrder(@RequestBody OrderRequest request) {
10         orderService.placeOrder(request);
11         return ResponseEntity.ok().build();
12     }
13 }
14
15 // Service Layer
16 @Service
17 public class OrderService {
18     private final OrderRepository orderRepository;
19     public void placeOrder(OrderRequest request) {
20         Order order = new Order(request.getItem(), request.getQuantity());
21         orderRepository.save(order);
22     }
23 }
24
25 // Repository Layer
26 @Repository
27 public interface OrderRepository extends JpaRepository<Order, Long> {}
```

## 전통적 5계층 구조의 역할과 경계

### ✳ 1. 프레젠테이션 계층 (Presentation Layer)

**역할:** 사용자 요청 수신, 응답 반환

**책임:** UI 또는 API 입출력, HTTP 처리

**주요 구성:**

- @Controller, @RestController
- DTO 변환 (@RequestBody, @ResponseBody)
- 검증 (@Valid, BindingResult)
- View 렌더링 (Thymeleaf, JSP) or JSON 응답

```

1  @RestController
2  @RequestMapping("/users")
3  public class UserController {
4      private final UserService userService;
5
6      @PostMapping
7      public ResponseEntity<Void> createUser(@RequestBody @Valid UserCreateRequest dto) {
8          userService.createUser(dto);
9          return ResponseEntity.ok().build();
10     }
11 }

```

⚠ **경계:** 절대 비즈니스 로직을 포함하지 않음.  
서비스 계층에게 유스케이스를 위임하고, DTO 변환에 집중함.

## ⚙ 2. 애플리케이션 계층 (Application Layer) (선택적)

**역할:** 유스케이스 조립 및 실행

**책임:** 도메인 객체 조합, 트랜잭션 관리, 흐름 제어

**주요 구성:**

- 유스케이스 서비스 (UserUseCase, OrderApplicationService)
- 외부 API 또는 DB 연동 조합
- 명령 객체(Command), 결과 객체(Result DTO)

```

1  @Service
2  public class RegisterUserUseCase {
3      private final UserRepository userRepository;
4
5      @Transactional
6      public void register(UserCreateCommand cmd) {
7          User user = new User(cmd.name(), cmd.email());
8          userRepository.save(user);
9      }
10 }

```

⚠ **경계:** 도메인 로직은 직접 수행하지 않으며, **조율자 역할**만 수행  
(※ Application Layer를 분리하지 않는 경우, 이 역할을 Service Layer가 겸함)

## 🔧 3. 서비스 계층 (Service Layer)

**역할:** 도메인 모델과 외부 세계의 연결

**책임:** 트랜잭션 경계 설정, 복합 작업 실행, 도메인/Repository 호출

**주요 구성:**

- @Service 클래스
- @Transactional 선언 위치
- 도메인 규칙 호출, 리포지토리 사용

```

1  @Service
2  public class OrderService {
3      private final OrderRepository orderRepository;
4      private final PaymentGateway paymentGateway;
5
6      @Transactional
7      public void placeOrder(OrderRequestDto dto) {
8          Order order = new Order(dto.getItemId(), dto.getQuantity());
9          paymentGateway.pay(dto.getPaymentInfo());
10         orderRepository.save(order);
11     }
12 }

```

🚫 **경계:** 도메인 객체를 사용해 로직을 실행하지만, 직접 비즈니스 규칙을 구현하지 않음.  
그건 **도메인 계층의 책임**임.

## 🧠 4. 도메인 계층 (Domain Layer)

**역할:** 핵심 비즈니스 규칙을 표현하는 중심 계층

**책임:** 상태 변경, 검증, 비즈니스 규칙 자체

**주요 구성:**

- `@Entity` 클래스
- 도메인 서비스 (`OrderValidator`, `ShippingPolicy`)
- 밸류 오브젝트 (`Money`, `Address`)
- 도메인 이벤트

```

1  @Entity
2  public class Order {
3      @Id @GeneratedValue
4      private Long id;
5
6      private int quantity;
7
8      public void increase(int amount) {
9          if (amount <= 0) throw new IllegalArgumentException();
10         this.quantity += amount;
11     }
12 }

```

🚫 **경계:** DB, 웹, 외부 API 등과 무관한 **순수 Java 로직**으로 구성됨.  
도메인은 **불변성, 상태 변경, 자기 책임 원칙**을 준수해야 함.

## 📁 5. 데이터 접근 계층 (Data Access Layer)

**역할:** DB, 파일 등 영속 계층 접근

**책임:** 데이터 저장, 조회, 수정, 삭제

**주요 구성:**

- `@Repository`
- `JpaRepository<T, ID>`
- MyBatis `@Mapper`
- Native Query, QueryDSL, `@Query`

```
1 @Repository
2 public interface UserRepository extends JpaRepository<User, Long> {
3     Optional<User> findByEmail(String email);
4 }
```

⚠ **경계:** 단순 CRUD 책임만 있음.

도메인 규칙 판단은 하지 않음.

서비스 또는 도메인 계층이 쿼리 결과를 판단하여 로직 실행.

## 🔗 계층 간 관계 요약

1 사용자 → Controller → Service/Application → Domain → Repository → DB

- Controller는 Service에게 위임
- Service는 Domain 객체를 사용하고 Repository로 저장
- Domain은 스스로 상태를 바꿈
- Repository는 DB 작업만 담당

## 🎯 설계 핵심 포인트

계층	핵심 원칙
Presentation	요청/응답 처리에만 집중
Application	흐름 조율, 유스케이스 단위의 비즈니스 트랜잭션
Service	비즈니스 서비스 중심, 도메인 호출 및 조합
Domain	비즈니스 규칙의 주체. 상태 변경의 중심
Repository	기술적 세부 구현 캡슐화. DB 작업 전담

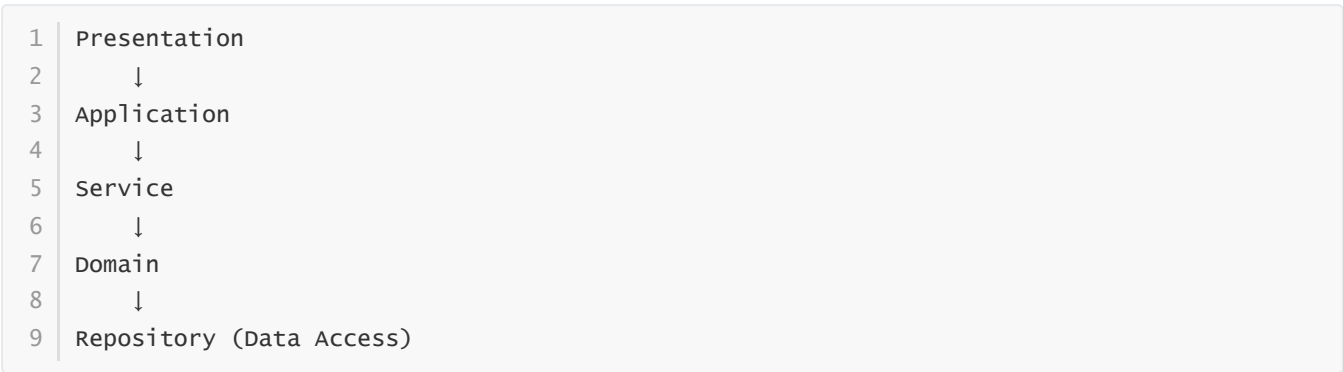
# 각 계층 간 의존성 방향과 책임

## 1. 의존성 방향이란?

계층 간 의존성 방향은

“어떤 계층이 어떤 계층에 의존하는가?”  
를 나타내는 방향성을 의미해.

### 기본 원칙 (상위 → 하위로만 의존)



- 상위 계층은 하위 계층을 사용할 수 있음
  - 하위 계층은 상위 계층을 몰라야 함
- 👉 이렇게 되면 의존성 역전이 필요 없음, 구조가 단순해짐.

## 각 계층의 책임 및 의존 방향


계층	의존 대상	책임
<b>Presentation</b> (컨트롤러)	Application / Service	사용자 요청 처리, DTO 매핑, 유효성 검증
<b>Application</b> (Use Case)	Service, Domain, Repository	유스케이스 조립, 트랜잭션 흐름 제어
<b>Service</b> (비즈니스 서비스)	Domain, Repository	비즈니스 흐름 조정, 외부 API 조합
<b>Domain</b> (비즈니스 모델)	없음 (가능하면 순수하게 유지)	상태 관리, 도메인 규칙 실행
<b>Repository</b> (데이터 접근)	DB, JPA API 등 외부 기술	데이터 저장/조회/삭제, 기술 캡슐화

## 예시 흐름: 회원 가입

```
1 [1] 사용자가 POST /users 요청
2   ↓
3 [2] UserController → UserService 호출
4   ↓
5 [3] UserService → User 도메인 객체 생성
6   ↓
7 [4] UserService → UserRepository.save()
8   ↓
9 [5] UserRepository → DB 저장
```

## 계층 간 의존 방향 위반 사례

### Repository → Service (역방향 호출)

```
1 // 잘못된 예시 (절대 금지)
2 public class UserRepository {
3     private final UserService userService; //  의존성 방향 위반
4 }
```

문제점:

- 순환 참조 발생 위험
- 관심사의 분리 원칙 위반
- 테스트와 유지보수 어려움

## 인터페이스를 활용한 의존 역전 (필요한 경우만)

- 도메인 → 외부 시스템에 의존할 경우,  
인터페이스를 정의해서 외부 구현체를 주입하는 구조로 만들어야 해.

예시:

```
1 // 도메인 계층에 정의된 인터페이스
2 public interface NotificationSender {
3     void sendWelcomeEmail(User user);
4 }
```

```
1 // 인프라 계층 구현
2 @Service
3 public class EmailNotificationSender implements NotificationSender {
4     public void sendWelcomeEmail(User user) {
5         // SMTP or kafka
6     }
7 }
```



## 📌 계층별 책임 요약

계층	책임	호출 대상	호출 주체
Presentation	요청 처리, DTO 변환	Application/Service	사용자의 직접 요청
Application	유스케이스 조립	Service, Domain	Controller
Service	비즈니스 흐름, 트랜잭션 경계	Domain, Repository	Application 또는 Controller
Domain	규칙과 상태의 중심	내부 메서드	Service
Repository	DB 작업만 수행	DB	Service 또는 Application

## 🔒 테스트 관점에서의 이점

- 각 계층이 **단일 책임**만 가지기 때문에, mocking, stub, fake 등을 통해 **계층별 테스트 분리** 가능
- 단위 테스트 작성이 훨씬 용이함.

## 📁 계층 간 의존성 관리 도구

- Spring DI (자동 주입)
- Interface 설계
- Bean 스코프 관리
- `@ComponentScan`, `@Configuration` 통한 모듈화

## ✅ 핵심 정리

핵심 원칙	설명
📁 단방향 의존성	위 → 아래만 의존, 아래 → 위 금지
✂️ 책임 분리	Controller는 비즈니스 X, Service는 View X
🚫 순환 참조 방지	계층 간 서로 호출 X
🧪 테스트 편의성	계층 별 mocking 가능
📦 확장성	계층 단위로 교체/재설계 가능

# 도메인 주도 설계(Domain-Driven Design) 관점에서 본 계층 구조

## 1. 도메인 주도 설계란?

DDD는 소프트웨어를 설계할 때, 핵심 비즈니스 개념(도메인)을 코드로 직접 모델링하고, 개발자와 비즈니스 전문가가 동일한 언어(Ubiquitous Language)를 사용하여 시스템을 만들어가는 접근 방식이다.

## 2. DDD의 전형적인 계층 구조 (4 Layer Architecture)

DDD에서는 전통적인 계층 아키텍처를 다음 네 가지로 재구성한다.

1	[1] Presentation Layer (사용자 인터페이스 계층)
2	[2] Application Layer (응용 계층)
3	[3] Domain Layer (도메인 계층)
4	[4] Infrastructure Layer (인프라 계층)

### 구조 시각화

1	Presentation	← 사용자 요청/응답 (API, UI)
2		
3	Application	← 유스케이스 실행, 흐름 조율
4		
5	Domain	← 핵심 비즈니스 로직, 모델, 규칙
6		
7	Infrastructure	← DB, 외부 시스템, 구현 기술
8		
9		

## 3. 각 계층의 DDD 관점 정의 및 책임

### ✦ [1] Presentation Layer — 사용자 인터페이스 계층

- 역할: 사용자와 시스템 사이의 인터페이스 (API, 웹, CLI 등)
- 책임:
  - 입력을 DTO로 받고, 출력을 DTO로 변환하여 응답
  - Application Layer의 유스케이스를 호출
- DDD적 사고:
  - "컨트롤러는 유스케이스를 트리거할 뿐이며, 도메인에 직접 관여하지 않는다."

## ⚙️ [2] Application Layer — 응용 계층

- **역할:** 유스케이스 단위로 작업을 조율 (순서 제어, 트랜잭션 경계 등)
- **책임:**
  - 도메인 모델 조작 순서 정의
  - 도메인 객체와 인프라스트럭처 객체 조합
- **DDD적 사고:**
  - "자기 자신은 상태를 가지지 않고, 비즈니스 규칙도 정의하지 않는다."
  - "단순히 작업의 흐름을 조율한다."

예:

```
1 @Transactional
2 public void registerUser(RegisterUserCommand cmd) {
3     if (userRepository.existsByEmail(cmd.getEmail())) {
4         throw new DuplicateEmailException();
5     }
6     User user = new User(cmd.getName(), cmd.getEmail());
7     userRepository.save(user);
8     emailSender.sendWelcome(user);
9 }
```

## 🧠 [3] Domain Layer — 도메인 계층 (핵심)

- **역할:** 비즈니스의 본질을 표현하는 모든 로직의 중심
- **책임:**
  - 도메인 모델 (@Entity, VO)
  - 도메인 서비스
  - 도메인 이벤트
  - 불변성, 규칙, 상태 전이
- **DDD적 사고:**
  - "도메인 모델은 반드시 스스로 유효성을 검증하고 스스로 상태를 변경해야 한다."
  - "모든 규칙은 도메인 모델 안에 들어가야 하며, 외부에서 강제되지 않는다."

예:

```
1 public class Order {
2     public void complete() {
3         if (this.status != Status.PAID) {
4             throw new IllegalStateException("결제되지 않은 주문은 완료할 수 없습니다.");
5         }
6         this.status = Status.COMPLETED;
7     }
8 }
```

## 📖 [4] Infrastructure Layer — 인프라 계층

- **역할:** 기술적 세부 구현 담당 (DB, 이메일, 파일 저장, Kafka 등)
- **책임:**
  - Repository 구현체
  - 외부 API 호출, 메시지 발행
  - Persistence, Messaging, I/O
- **DDD적 사고:**
  - "기술적인 구현은 도메인을 보조할 뿐이며, 핵심 모델과는 완전히 분리되어야 한다."
  - "Repository 인터페이스는 도메인 또는 응용 계층에서 선언하고, 실제 구현은 Infrastructure에서 한다."

예:

```
1 @Repository
2 public class JpaUserRepository implements UserRepository {
3     private final SpringDataJpaRepo jpaRepo;
4     public void save(User user) { jpaRepo.save(user); }
5 }
```

## 🔄 계층 간 의존성 방향 (DDD 원칙)

호출자	호출 대상
Presentation → Application	
Application → Domain	
Domain → 외부 기술과 직접 의존 ❌ (역전 필요)	
Infrastructure → Domain (인터페이스 구현을 통해 의존)	

🔴 도메인 계층은 의존이 없어야 하며, 가장 순수한 형태여야 한다.

## 💬 예: 회원 가입 흐름 (DDD 관점)

```
1 UserController
2   ↓
3 RegisterUserUseCase
4   ↓
5 User 도메인 객체 생성 및 상태 변경
6   ↓
7 UserRepository.save(user) ← interface
8   ↓
9 JpaUserRepository implements UserRepository
```

## DDD 관점에서의 좋은 설계 예시

요소	예시	설명
Entity	User, Order	식별자(ID)를 가진 변경 가능한 객체
Value Object	Address, Money	불변 객체, 의미 기반 비교
Domain Service	DiscountPolicyService	복수 엔티티 간의 규칙
Application Service	RegisterUserService	유스케이스 흐름 조율
Repository	UserRepository	도메인 객체 저장소 인터페이스
Infrastructure	JpaUserRepository, EmailSenderImpl	기술적 구현 세부

## DDD 계층 구조의 핵심 요약

계층	위치	책임	기술 의존성
Presentation	가장 바깥	요청/응답 처리	HTTP, JSON
Application	중간	유스케이스 실행, 흐름 조율	Spring, 외부 API
Domain	중심	비즈니스 규칙, 상태 관리	✗ (기술 무관해야 함)
Infrastructure	바깥쪽	DB, 메시징 등 구현	✓

## ✓ 최종 요약

핵심 원칙	설명
🎯 도메인이 중심이다	모든 규칙과 상태는 도메인에서 시작되어야 함
💻 기술은 도메인을 보조한다	도메인이 기술에 종속되면 안 됨
⬆ 의존성은 바깥에서 안쪽으로	바깥 계층이 안쪽 계층을 의존해야 함 (역전 구조)
💬 Ubiquitous Language	모델과 코드가 하나의 언어를 공유해야 함