4. 애플리케이션 계층 (Application Layer) — 선택적

복잡한 유스케이스 처리 (Command, Use Case 중심)

🧩 1. 개념 정의

🔸 유스케이스(Use Case)란?

> 시스템이 클라이언트의 요청에 대해 수행해야 할 **하나의 기능 단위**.
> "사용자 입장에서 무슨 일이 일어나는가?"에 집중하는 비즈니스 흐름.

🔸 Command 기반 Use Case 처리란?

- **명령(Command)** 객체를 통해 외부 요청을 내부 도메인 로직과 **명확히 분리**
- Use Case는 이러한 Command를 받아 → **검증** → **도메인 객체 호출** → **결과 반환**하는 역할

📦 2. 주요 구성 요소

구성요소	설명
Command	클라이언트 요청을 캡슐화한 단순 입력 데이터 (DTO 역할)
UseCase(Application Service)	하나의 유스케이스를 실행. 흐름 조율, 트랜잭션 관리
Domain	실제 상태 변경, 규칙 적용 등의 핵심 로직을 담당
Result DTO	클라이언트에게 반환할 응답 데이터를 캡슐화

🧱 3. 구조 예시

```
[Controller]
    ↓
[Command 객체 생성]
    ↓
[UseCase 호출] → [Command 처리]
    ↓
[도메인 모델 조작]
    ↓
[결과 객체 반환 (DTO)]
```

🧠 4. 예제: 사용자 회원가입 유스케이스

1) 📩 Command 객체

```
public record RegisterUserCommand(String name, String email, String password) {}
```

2) 🧭 UseCase (Application Layer)

```
@Service
public class RegisterUserUseCase {

    private final UserRepository userRepository;
    private final PasswordEncoder passwordEncoder;

    @Transactional
    public void register(RegisterUserCommand cmd) {
        if (userRepository.existsByEmail(cmd.email())) {
            throw new EmailAlreadyExistsException();
        }

        User user = new User(cmd.name(), cmd.email(),
passwordEncoder.encode(cmd.password()));
        userRepository.save(user);
    }
}
```

> 📌 여기서 UseCase는 "회원가입"이라는 하나의 흐름을 책임지고 조율하는 클래스임

---

3) 🔧 도메인 엔티티

```
@Entity
public class User {
    @Id @GeneratedValue
    private Long id;

    private String name;
    private String email;
    private String password;

    protected User() {}

    public User(String name, String email, String encodedPassword) {
        this.name = name;
        this.email = email;
        this.password = encodedPassword;
    }
}
```

---

4) 🎮 Controller

```
@RestController
@RequestMapping("/api/users")
public class UserController {

    private final RegisterUserUseCase registerUserUseCase;

    @PostMapping
    public ResponseEntity<Void> register(@RequestBody @Valid RegisterUserRequest request) {
        RegisterUserCommand cmd = request.toCommand();
        registerUserUseCase.register(cmd);
        return ResponseEntity.status(HttpStatus.CREATED).build();
    }
}
```

5) DTO → Command 변환

```
public class RegisterUserRequest {
    @NotBlank private String name;
    @Email private String email;
    @NotBlank private String password;

    public RegisterUserCommand toCommand() {
        return new RegisterUserCommand(name, email, password);
    }
}
```

✅ 5. 왜 이렇게 분리할까?

목적	이유
▶️ 명확한 책임 분리	요청 → 명령 → 유스케이스 → 도메인
▶️ 테스트 용이성	UseCase 단독 테스트 가능 (Mock만 주입하면 됨)
▶️ 보안/검증 집중화	입력 검증, 정책 검증을 UseCase 내로 집중 가능
▶️ 단일 유스케이스 중심 설계	복잡한 서비스 메서드 혼란 제거

📚 6. 여러 Use Case를 구분할 때 기준

기준	예시 UseCase
의미 단위로 나눌 것	`RegisterUserUseCase`, `LoginUserUseCase`
단일 책임 원칙(SRP) 따를 것	결제와 알림은 분리

기준	예시 UseCase
트랜잭션 경계를 기준으로 나눌 것	하나의 DB 작업 묶음이면 하나의 UseCase

🛠️ 7. 고급 확장 요소 (선택 사항)

기능	설명
Result 객체	결과를 `ResultDto`, `Success/Fail` 객체로 감싸기
CQRS 적용	Query UseCase와 Command UseCase 분리
Validation Layer 추가	Command 검증기 (`CommandValidator`) 별도 생성
도메인 이벤트 발행	`userRegisteredEvent` 같은 이벤트 객체 발행
Clean Architecture와 연결	Interface → UseCase → Domain → Infra 방향 유지

✅ 요약

구성 요소	역할
`Command`	요청 파라미터를 캡슐화한 객체
`UseCase`	요청의 비즈니스 흐름을 실행하고 도메인을 조율
`Domain`	상태 변경, 규칙 적용 등 핵심 로직 담당
`Result`	클라이언트에 전달할 응답 데이터 모델
`Controller`	HTTP 요청을 수신하고 Command로 변환한 후 UseCase 호출

📦 디렉터리 구조 예시

```
com.example.user
├── controller
│   └── UserController.java
├── command
│   └── RegisterUserCommand.java
├── usecase
│   └── RegisterUserUseCase.java
├── domain
│   └── User.java
├── dto
│   └── RegisterUserRequest.java
└── repository
    └── UserRepository.java
```

도메인 로직 조합 및 트랜잭션 조율

🧩 1. 개념 정리

용어	의미
도메인 로직 조합	여러 도메인 객체 간의 작업을 **조율하고 연결**하는 비즈니스 흐름
트랜잭션 조율	여러 작업이 하나의 **트랜잭션 안에서 원자성**을 가지도록 관리하는 것 (`@Transactional` 사용)

🎯 2. 왜 분리하고 조합해야 하나?

- 도메인 객체는 **자기 책임 범위의 상태**만 변경해야 함.
- 도메인 객체 간 협력이 필요할 땐, **응용 계층(Application Layer)**에서 orchestrate 해야 함.
- DB 작업, 외부 API 호출, 이벤트 발행 등의 **경계 설정이 명확**해야 유지보수가 쉬움.

💡 3. 책임 분리 핵심 규칙

계층	책임
Domain	단일 객체의 불변성/규칙/상태 변경
Application / Service	여러 도메인 간 협력, 트랜잭션 조정, 흐름 제어
Repository	영속성 책임 (저장, 조회, 삭제)
Infra	외부 시스템, 이메일, 메시징 등 처리

📦 4. 예제: 주문(Order) + 재고(Inventory) + 결제(Payment)

유스케이스 시나리오:

> "상품을 주문하면 → 재고를 차감하고 → 결제를 수행하고 → 주문을 저장해야 한다."

✔️ 도메인 객체 (단일 책임만 수행)

Order.java

```
public class Order {
    private OrderStatus status;

    public void markAsPaid() {
        if (this.status != OrderStatus.REQUESTED) {
            throw new IllegalStateException("결제 상태 아님");
        }
        this.status = OrderStatus.PAID;
    }
```

```
}
```

Inventory.java

```
public class Inventory {
    private int quantity;

    public void decrease(int amount) {
        if (quantity < amount) throw new OutOfStockException();
        quantity -= amount;
    }
}
```

✔ UseCase or Service Layer (조합 + 트랜잭션 조율)

```
@Service
public class OrderService {

    private final OrderRepository orderRepository;
    private final InventoryRepository inventoryRepository;
    private final PaymentGateway paymentGateway;

    @Transactional
    public void placeOrder(OrderRequestDto dto) {
        Inventory inventory = inventoryRepository.findByItemId(dto.getItemId());
        inventory.decrease(dto.getQuantity());

        paymentGateway.pay(dto.getPaymentInfo()); // 외부 시스템

        Order order = new Order(dto.getUserId(), dto.getItemId(), dto.getQuantity());
        order.markAsPaid();

        orderRepository.save(order);
    }
}
```

여기서 중요한 조율 포인트:

위치	설명
`inventory.decrease(...)`	도메인 내부 검증 후 상태 변경
`paymentGateway.pay(...)`	외부 API 호출, 예외 발생 가능
`@Transactional`	전체 흐름이 하나의 트랜잭션으로 묶임
`order.markAsPaid()`	도메인 객체의 상태 변화 직접 호출

⚠️ 5. 트랜잭션 주의사항

항목	설명
트랜잭션 경계	반드시 UseCase/Service 계층에서만 `@Transactional` 선언
예외 발생	예외 발생 시 롤백 (`RuntimeException` 계열만 기본 롤백)
외부 시스템 호출	외부 결제 API 등은 트랜잭션 바깥에서 호출하는 것이 일반적 (보상 트랜잭션 필요)
중첩 트랜잭션	내부 호출 시 `@Transactional` 이 적용되지 않을 수 있음 (프록시 주의)

🔁 도메인 조합 시 패턴

패턴	설명
Orchestration (명령형)	UseCase에서 순서를 정의하고 각 도메인을 호출
Choreography (이벤트 기반)	도메인 간 이벤트를 발생시켜 비동기 협력 (예: Spring Event, Kafka)

✅ 요약 정리

항목	역할
도메인 로직 조합	여러 도메인 모델의 기능을 하나의 유스케이스로 통합
트랜잭션 조율	단일 흐름으로 묶어 DB 상태를 일관되게 유지
책임 위치	반드시 Application/Service Layer에서 조율
도메인	독립적 상태 변경만 수행 (외부 의존 없이)
@Transactional	조율의 핵심, rollback 처리 기준 확인 필수

📚 디렉터리 구조 예시 (조합 중심)

```
order/
├── domain/
|   ├── Order.java
|   ├── Inventory.java
|   └── Payment.java
├── repository/
|   ├── OrderRepository.java
|   └── InventoryRepository.java
├── application/
|   ├── PlaceOrderUseCase.java
|   └── CancelOrderUseCase.java
├── infra/
|   └── ExternalPaymentGateway.java
```

애플리케이션 서비스 vs 도메인 서비스

🔍 1. 핵심 정의 비교

구분	애플리케이션 서비스 (Application Service)	도메인 서비스 (Domain Service)
위치	응용 계층 (UseCase 계층)	도메인 계층
목적	유스케이스 조합, 트랜잭션 관리, 흐름 조율	도메인 규칙 표현 (단일 엔티티로 표현하기 어려울 때)
역할	여러 도메인 객체/외부 요소를 조합	도메인 객체 간 협력에 필요한 비즈니스 로직 수행
상태 보유	❌ (Stateless)	❌ (Stateless)
호출 위치	컨트롤러 → 애플리케이션 서비스	애플리케이션 서비스 or 도메인 엔티티 내부
기술 의존	있음 (트랜잭션, 리포지토리 등)	없음 (순수 자바 코드로 유지)

🧩 2. 애플리케이션 서비스란?

> 클라이언트 요청을 받아서
>
> - 유효성 검사
> - 트랜잭션 시작
> - 여러 도메인 객체와 외부 시스템 호출
> - 결과 반환
>
> 을 담당하는 **유스케이스 중심 계층**

📦 예시

```
@Service
public class TransferMoneyUseCase {

    private final AccountRepository accountRepository;
    private final EventPublisher eventPublisher;

    @Transactional
    public void transfer(TransferCommand cmd) {
        Account from = accountRepository.findById(cmd.fromId());
        Account to = accountRepository.findById(cmd.toId());

        from.withdraw(cmd.amount());
        to.deposit(cmd.amount());

        accountRepository.save(from);
        accountRepository.save(to);

        eventPublisher.publish(new MoneyTransferredEvent(cmd.fromId(), cmd.toId()));
    }
}
```

🧭 핵심: 흐름의 조율자. 도메인의 **로직은 호출만 함**, 스스로 가지지 않음.

🧠 3. 도메인 서비스란?

> 도메인 객체 하나로 표현하기 어려운 **비즈니스 규칙**을
> 도메인 계층 안에서 **순수 자바 로직**으로 표현하는 객체

사용 시점:

- 하나의 엔티티에 책임을 부여하기 애매한 경우
- 여러 엔티티/VO 간의 협업이 필요한 도메인 규칙이 있을 때

📦 예시

```
public class ShippingService {
    public boolean isFreeShipping(Order order, Member member) {
        return member.isVip() || order.totalPrice() > 100_000;
    }
}
```

> 📌 도메인 서비스는 상태를 가지지 않고, **순수 계산/판단 로직**만 가짐.

⚠️ 4. 잘못 쓰기 쉬운 사례 비교

❌ 나쁜 예 (비즈니스 로직을 애플리케이션 서비스에 넣음)

```
public void createDiscountedOrder(OrderRequestDto dto) {
    if (dto.price > 10000) {
        dto.discount = 0.1;
    }
    Order order = new Order(dto);
    ...
}
```

> 이런 비즈니스 조건은 **도메인 규칙**이며 → 도메인 or 도메인 서비스로 내려보내야 함.

📚 5. 실전 비교 예제

✔️ 도메인 서비스

```
public class PasswordStrengthChecker {
    public boolean isWeak(String password) {
        return password.length() < 8 || password.matches(".*1234.*");
    }
}
```

✔️ 애플리케이션 서비스에서 호출

```
public class UserRegistrationUseCase {

    private final PasswordStrengthChecker passwordChecker;

    public void register(RegisterUserCommand cmd) {
        if (passwordChecker.isWeak(cmd.password())) {
            throw new WeakPasswordException();
        }
        ...
    }
}
```

✅ 요약 정리표

항목	Application Service	Domain Service
목적	유스케이스 흐름 제어	도메인 규칙 정의
위치	응용 계층	도메인 계층
상태 보유	❌	❌

항목	Application Service	Domain Service
순수성	트랜잭션, 리포지토리 사용 가능	기술 독립, 순수 자바 로직
테스트	Mocking 필요	단위 테스트 매우 쉬움
의존성	외부 시스템 의존 가능	도메인 객체만 의존 가능

🧱 디렉터리 구조 예시

```
user/
├── application/
|   └── RegisterUserUseCase.java
├── domain/
|   ├── User.java
|   └── PasswordStrengthChecker.java  ← 도메인 서비스
├── command/
|   └── RegisterUserCommand.java
```

✅ 결론

상황	선택
흐름 조율, 외부 시스템 호출, 트랜잭션	Application Service
핵심 비즈니스 규칙, 엔티티 외부 판단	Domain Service

예: `UserRegistrationService`, `OrderApplicationService`

✅ 예시 1: `UserRegistrationService` (회원가입 유스케이스)

📌 요구사항

- 사용자가 이름, 이메일, 비밀번호를 입력하면
- 이메일 중복을 확인하고
- 비밀번호를 암호화해서
- 새 사용자를 등록하고
- 이메일 알림을 발송한다

📦 Command 객체

```
public record RegisterUserCommand(String name, String email, String password) {}
```

📦 도메인 객체

```
@Entity
public class User {
    @Id @GeneratedValue
    private Long id;

    private String name;
    private String email;
    private String encryptedPassword;

    protected User() {}

    public User(String name, String email, String encryptedPassword) {
        this.name = name;
        this.email = email;
        this.encryptedPassword = encryptedPassword;
    }
}
```

📦 도메인 서비스 (선택)

```
public class PasswordPolicyService {
    public boolean isWeak(String password) {
        return password.length() < 8 || password.contains("1234");
    }
}
```

⚙️ Application Service (UserRegistrationService)

```
@Service
public class UserRegistrationService {

    private final UserRepository userRepository;
    private final PasswordEncoder passwordEncoder;
    private final EmailSender emailSender;
    private final PasswordPolicyService passwordPolicy;

    @Transactional
    public void register(RegisterUserCommand cmd) {
        if (userRepository.existsByEmail(cmd.email())) {
            throw new DuplicateEmailException();
        }

```

```
        if (passwordPolicy.isWeak(cmd.password())) {
            throw new WeakPasswordException();
        }

        String encryptedPassword = passwordEncoder.encode(cmd.password());
        User user = new User(cmd.name(), cmd.email(), encryptedPassword);
        userRepository.save(user);

        emailSender.sendWelcome(cmd.email());
    }
}
```

✅ 구조 요약

구성요소	클래스
Command	`RegisterUserCommand`
Application Service	`UserRegistrationService`
Domain Entity	`User`
Domain Service	`PasswordPolicyService`
Infra 의존	`UserRepository`, `EmailSender`, `PasswordEncoder`

✅ 예시 2: `OrderApplicationService` (주문 처리 유스케이스)

📌 요구사항

- 사용자가 상품을 주문하면
- 재고를 차감하고
- 결제를 요청한 후
- 주문 상태를 "결제 완료"로 변경하고 저장한다

📦 Command 객체

```
public record PlaceOrderCommand(Long userId, Long itemId, int quantity, PaymentInfo
paymentInfo) {}
```

📦 도메인 엔티티

```
@Entity
public class Order {
    @Id @GeneratedValue
    private Long id;

    private Long userId;
    private Long itemId;
    private int quantity;
    private OrderStatus status;

    public Order(Long userId, Long itemId, int quantity) {
        this.userId = userId;
        this.itemId = itemId;
        this.quantity = quantity;
        this.status = OrderStatus.REQUESTED;
    }

    public void markAsPaid() {
        if (status != OrderStatus.REQUESTED) {
            throw new IllegalStateException("이미 처리된 주문입니다.");
        }
        this.status = OrderStatus.PAID;
    }
}
```

📦 도메인 서비스 (예: 배송 정책 판단 등)

```
public class ShippingPolicy {
    public boolean isFreeShipping(int quantity) {
        return quantity >= 5;
    }
}
```

⚙️ Application Service (OrderApplicationService)

```
@Service
public class OrderApplicationService {

    private final InventoryRepository inventoryRepository;
    private final OrderRepository orderRepository;
    private final PaymentGateway paymentGateway;

    @Transactional
    public void placeOrder(PlaceOrderCommand cmd) {
        Inventory inventory = inventoryRepository.findByItemId(cmd.itemId());
        inventory.decrease(cmd.quantity());

```

```
        paymentGateway.charge(cmd.paymentInfo());

        Order order = new Order(cmd.userId(), cmd.itemId(), cmd.quantity());
        order.markAsPaid();
        orderRepository.save(order);
    }
}
```

✅ 구조 요약

구성요소	클래스
Command	`PlaceOrderCommand`
Application Service	`OrderApplicationService`
Domain Entity	`Order`, `Inventory`
Domain Service	`ShippingPolicy` (선택)
Infra 의존	`OrderRepository`, `InventoryRepository`, `PaymentGateway`

📐 공통 설계 원칙

원칙	설명
유스케이스 단위로 분리	`register()`, `placeOrder()` 는 각각 독립 유스케이스
Command → Domain 전환	직접 `User`, `Order` 객체 생성 또는 팩토리 사용
트랜잭션 조율 위치 명확	항상 Application Service에 `@Transactional` 선언
외부 시스템 호출	결제, 이메일 발송 등은 도메인 밖에서 수행
도메인은 순수하게 유지	외부 API, DB 접근, Validation 책임 없음

DTO/Command 객체 패턴

🧩 1. 용어 정리

개념	설명
DTO (Data Transfer Object)	클라이언트 ↔ 서버 간 데이터 전달용 객체. API 입출력에 특화
Command	유스케이스를 실행하기 위한 **도메인 내부 전달 객체**, DTO보다 더 도메인 친화적
Result DTO / Response DTO	유스케이스 수행 결과를 클라이언트에 응답할 때 사용하는 객체

🎯 2. 왜 분리하는가?

이유	설명
🔐 도메인 모델 보호	외부 요청이 도메인 모델을 직접 건드리지 않도록 캡슐화
🧭 계층 간 책임 명확화	Presentation ↔ Application ↔ Domain 계층 간 경계 유지
🔁 재사용성 향상	Command는 유스케이스 내부에서 재사용 가능, DTO는 컨트롤러 전용
🧪 테스트 편의성	단위 테스트 시 Command만 주입하면 됨 (복잡한 컨트롤러 안 써도 됨)

🔗 3. 계층 흐름 구조

```
Client JSON
   ↓
@RequestBody DTO
   ↓ (toCommand)
Command 객체
   ↓
Application Service
   ↓
Domain Entity / Domain Service
   ↓
Result DTO or Response
   ↓
ResponseEntity<DTO>
   ↓
Client
```

🔨 4. 실전 예시: 사용자 회원가입 흐름

📥 요청 DTO (API 요청 수신)

```
public class RegisterUserRequest {

    @NotBlank private String name;
    @Email private String email;
    @NotBlank private String password;

    public RegisterUserCommand toCommand() {
        return new RegisterUserCommand(name, email, password);
    }
}
```

- 사용자가 JSON으로 보낸 데이터와 **1:1 대응**
- `@Valid` 등을 붙여 **입력 검증 전용**

- 도메인에서는 절대 사용되지 않음

📜 Command 객체 (Application/Domain 전달용)

```
public record RegisterUserCommand(
    String name,
    String email,
    String password
) {}
```

- 유스케이스에서 사용하는 **의미 있는 입력 데이터**
- 순수 Java 타입, 값만 갖고 있고 로직 없음
- 단위 테스트 시 직접 생성해서 바로 주입 가능

📤 응답 DTO (결과 반환용)

```
public class RegisterUserResponse {
    private Long userId;
    private String name;

    public RegisterUserResponse(User user) {
        this.userId = user.getId();
        this.name = user.getName();
    }
}
```

- 도메인 객체 → 응답으로 가공할 때 사용
- 클라이언트에 필요한 정보만 담음
- 보안상 노출하면 안 되는 데이터는 숨김

⚙️ Application Service

```
@Service
public class UserRegistrationService {

    private final UserRepository userRepository;
    private final PasswordEncoder passwordEncoder;

    @Transactional
    public User register(RegisterUserCommand cmd) {
        if (userRepository.existsByEmail(cmd.email())) {
            throw new DuplicateEmailException();
        }

        String encrypted = passwordEncoder.encode(cmd.password());
        User user = new User(cmd.name(), cmd.email(), encrypted);
        return userRepository.save(user);
```

```
    }
}
```

🎮 Controller

```
@RestController
@RequestMapping("/api/users")
public class UserController {

    private final UserRegistrationService userRegistrationService;

    @PostMapping
    public ResponseEntity<RegisterUserResponse> register(
        @RequestBody @Valid RegisterUserRequest request
    ) {
        User user = userRegistrationService.register(request.toCommand());
        return ResponseEntity.status(HttpStatus.CREATED)
                             .body(new RegisterUserResponse(user));
    }
}
```

✅ DTO vs Command vs Response 차이 요약표

구분	DTO (Request)	Command	Response DTO
위치	Controller	Application Layer 내부	Controller (응답)
목적	API 요청 매핑	유스케이스 실행	API 응답 가공
내용	클라이언트와 1:1	비즈니스 의미 중심	사용자에게 보여줄 데이터
검증	`@Valid`, `@NotBlank` 등	보통 없음 (이미 검증됨)	보안 필터링 포함 가능
테스트	Mock MVC에 적합	단위 테스트에 적합	JSON 스펙 문서화에 적합

🛠️ 실전 개발 팁

팁	설명
DTO에는 로직 넣지 말기	getter/setter 또는 toCommand만 허용
Command에는 검증 로직 넣지 말기	검증은 Controller 또는 Validator에서
DTO와 Command 이름 구분	`UserRequest`, `UserCommand`, `UserResponse`로 명확히 나눠야
Value Object와 Command 혼동 금지	Command는 입력값, VO는 도메인 속성

🧱 디렉토리 구조 예시

```
user/
├── controller/
|   └── UserController.java
├── dto/
|   ├── RegisterUserRequest.java
|   └── RegisterUserResponse.java
├── command/
|   └── RegisterUserCommand.java
├── application/
|   └── UserRegistrationService.java
├── domain/
|   └── User.java
```

✅ 마무리 핵심 요약

규칙	설명
API ↔ DTO ↔ Command ↔ Domain 순으로 흐른다	외부 요청은 DTO로 받고, 내부에서는 Command 객체로 전달한 뒤, 도메인 로직에서 처리함. **역방향으로 흐르면 계층 간 책임이 섞여 구조가 무너짐**.
Command는 Application Layer의 입력 모델이다	Command는 DTO가 아님. `유스케이스 실행에 필요한 필수 데이터` 만을 담은 **내부 전달 객체**이며, Application Layer가 도메인을 호출할 때 사용하는 **의미 중심의 모델**이다.
DTO는 Web 요청/응답에만 사용되어야 한다	DTO는 Controller에만 국한되어야 하며, 도메인 또는 Application Service에 직접 넘기면 **프레젠테이션 계층이 도메인을 침범**하는 구조가 된다.
응답 DTO는 도메인 엔티티를 그대로 노출하지 않도록 구성한다	`User`, `Order` 같은 도메인 객체를 JSON으로 직접 응답하면 * 보안 이슈 (비밀번호, 내부 상태 노출) * 순환 참조 문제 * API 버전 관리 어려움 이 생김. → 반드시 `Response DTO`를 별도로 만들어 사용해야 한다.
계층 간 객체는 절대 섞이지 말아야 한다 (순환, 결합 위험)	* `DTO`가 도메인 객체를 의존하거나 * `도메인 객체`가 DTO 또는 Controller를 알게 되면 → **의존 방향이 역전**되어 SRP/DI 원칙이 무너짐. 계층마다 **독립성과 책임의 경계를 철저히 지켜야 유지보수 가능한 시스템이 된다.**

✅ 요약 정리

계층	객체	역할
Controller	DTO (Request/Response)	요청 바인딩, 응답 생성 전용
Application Service	Command	유스케이스 실행용 입력 모델
Domain	Entity/VO	비즈니스 상태, 규칙 처리