4장. 유스케이스 구현하기

시작하기

- 헥사고날 아키텍처는 도메인 중심의 아키텍처에 적합
- 도메인 엔티티를 만드는 것으로 시작
- 도메인 엔티티를 중심으로 유스케이스 구현

유스케이스

⌨️ 역할

1. 입력을 받는다.
2. 비즈니스 규칙을 검증한다.

3. 모델 상태를 조작한다.
4. 출력을 반환한다.

💪 책임

- 비즈니스 규칙 검증
  - 도메인 엔티티와 책임 공유
- 모델 상태 변경
  - 다른 아웃고잉 어댑터 호출 가능
- 아웃고잉 어댑터에서 온 출력값을 반환할 출력 객체로 변환

🧐 입력 유효성 검증

- 애플리케이션 계층에서 실시
  - 유효하지 않은 입력값을 받아 모델의 상태를 해칠 수 있음
  - 유스케이스에 전달하기 전에 검증하면 잘 검증되었는지 신뢰하기 어려움
- 입력 모델의 생성자에서 유효성 검증 가능
  - 유효하지 않은 객체를 만드는 것이 불가능해짐
  - 빌더 패턴 사용 시 단점을 커버
    - 필드를 새로 추가하는 상황에서 위험 !
  - 유스케이스마다 다른 입력 모델을 사용하자
    - 특정 케이스의 분기 처리를 위해 코드 냄새를 만드는 일을 방지
    - 명확한 유스케이스에 도움
    - 유스케이스 간 결합 제거
- java의 `bean validation API`로 귀찮은 작업들을 애너테이션으로 표현 가능

🧐 비즈니스 규칙 검증

- 도메인 엔티티에서 실시
  - 여의치 않다면 유스케이스 코드에서 도메인 엔티티를 사용하기 전에 실시

- 검증 로직이 모델의 상태에 접근해야 하는지 집중

⚔️ 입력 유효성 검증 vs 비즈니스 규칙 검증

입력 유효성

- 구문상의 유효성을 검증
- 모델에 접근하지 않고도 검증되는 경우

비즈니스 규칙 유효성

- 의미적인 유효성을 검증
- 모델의 현재 상태에 접근하는 경우

도메인 모델

🐷 풍부한 도메인 모델

- 엔티티에서 가능한 한 많은 도메인 로직이 구현됨
- 유스케이스는 도메인 모델의 진입점으로 동작
  - 체계화된 도메인 엔티티 메서드 호출

🦴 빈약한 도메인 모델

- 얇은 엔티티
  - 상태를 표현하는 필드
  - getter/setter
- 도메인 로직은 유스케이스 클래스에 구현

유스케이스마다 다른 출력 모델

입력처럼 출력도 유스케이스에 맞게 구체화하자

- 호출자에게 꼭 필요한 데이터만 반환할 것
- 가능한 한 적게 반환할 것
- 같은 출력 모델을 공유하지 말 것
    - 유스케이스 간 강결합 발생
- 단일 책임 원칙을 적용하여 모델을 분리할 것
- 도메인 엔티티를 출력 모델로 사용하지 말 것

읽기 전용 유스케이스

- 유스케이스로 간주되지 않는다면 쿼리로 표현 가능
- 쿼리를 위한 인커밍 전용 포트 생성 후 쿼리 서비스에 구현
- CQRS 등의 개념에 적합

결론

- 도메인 로직을 입맛에 맞게 구현하고 입출력 모델을 독립적으로 모델링하면 사이드 이펙트를 피할 수 있다.
- 유스케이스별로 모델을 만들면 유효성 검증과 작업 효율성에 도움이 된다.