3장. 코드 구성하기

패키지 구조 만들기

🚂 계층으로 구성하기

웹, 도메인, 영속성 계층에 대해 전용 패키지 두기 (`web`, `domain`, `persistence`)

단점

1. 애플리케이션의 기능 조각이나 특성을 구분 짓는 패키지 경계가 없다.
   - 추가적인 구조가 없다면 서로 연관되지 않은 기능들끼리 예상하지 못한 side effect를 일으키게 됨
2. 애플리케이션이 어떤 유스케이스들을 제공하는지 파악할 수 없다.
   - 어떤 기능을 찾기 위해서 어떤 서비스가 이를 구현했는지 추측해야 함

🕹️ 기능으로 구성하기

기능을 기준으로 패키지를 만들고 계층 패키지는 없애기

특징

- 패키지 외부에서 접근되면 안되는 클래스는 package-private 접근 수준으로 패키지 간 경계를 강화한다.
    - 각 기능 간 불필요한 의존성을 방지한다.

단점

- 계층으로 구성한 패키징 방식보다 아키텍처의 가시성을 훨씬 떨어뜨린다.
    - 어댑터를 나타내는 패키지명이 없음
    - 인커밍 포트 / 아웃고잉 포트 확인 불가
    - 도메인-영속성 코드 간 의존성을 역전시켜도 package-private 접근 수준을 이용해 실수로 영속성 코드에 의존하는 것을 막을 수 없음

👷🏻 아키텍처적으로 표현력 있는 패키지 구성하기

헥사고날 아키텍처의 핵심 요소

- 엔티티
- 유스케이스
- 인커밍/아웃고잉 포트
- 인커밍/아웃고잉 어댑터

장점

- `아키텍처-코드 갭` or `모델-코드-갭`을 효과적으로 다룰 수 있다.
    - 패키지 구조가 아키텍처를 반영할 수 있다 !
- 작업 중인 코드를 어떤 패키지에 넣어야 할지 계속 생각하게 해준다.
- 필요할 경우 하나의 어댑터를 다른 구현으로 쉽게 교체할 수 있따.
- DDD 개념에 직접적으로 대응시킬 수 있다.

🤔 패키지가 많으면 다 public으로 해야 할까?

적어도 어댑터 패키지에서는 그렇지 않다.

- 모든 클래스들은 application 패키지 내 포트 인터페이스를 통하지 않으면 바깥에서 호출되지 않는다.

의존성 주입의 역할

클린 아키텍처의 가장 본질적 요건은 애플리케이션 계층이 인커밍/아웃고잉 어댑터에 의존성을 갖지 않는 것이다.

어댑터

인커밍 어댑터

- 제어 흐름의 방향이 어댑터 - 도메인 코드 간 의존성 방향과 같다. (=쉽다.)

아웃고잉 어댑터

- 제어 흐름의 반대 방향으로 의존성을 돌리기 위해 의존성 역전 원칙을 이용해야 한다.
- 애플리케이션에 계층에 인터페이스(포트)를 만들고 어댑터에 인터페이스 구현 클래스를 둔다.

포트 구현체를 누가 애플리케이션 계층에 제공할까?

1. 포트를 애플리케이션 계층 안에서 수동으로 초기화한다.
    - 애플리케이션 계층에 어댑터에 대한 의존성이 추가된다.
2. ✨ 의존성 주입을 활용한다.
    - 모든 계층에 의존성을 가진 중립적 컴포넌트를 도입한다.
    - 아키텍처를 구성하는 대부분의 클래스를 초기화한다.

결론

> 완벽한 방법은 없다. 그러나 표현력 있는 패키지 구조는 적어도 `아키텍처-코드 갭`을 줄인다.

- 코드에서 아키텍처의 특정 요소를 찾기 위해 아키텍처 다이어그램의 박스 이름을 따라 패키지를 탐색한다.
    - 의사소통, 개발, 유지보수가 편해진다 👍🏻