10장. 아키텍처 경계 강제하기

경계를 강제한다는 것

의존성이 올바른 방향을 향하도록 강제하는 것

- 계층 사이에 경계가 있다.
- 의존성 규칙에 따르면 계층 경계를 넘는 의존성은 항상 안쪽 방향으로 향해야 한다.

경계와 의존성

접근 제한자

Java에서 제공하는 가장 기본적인 도구

- 어떤 접근 제한자가 있고, 차이점이 무엇인가?

`package-private` (`default`) 제한자의 중요성

- Java 패키지를 통해 클래스들을 응집적인 모듈로 만들어주기 때문
- 모듈 내 클래스는 서로 접근 가능 / 패키지 바깥에서는 접근 불가능
- 모듈의 진입점이 될 클래스들만 public으로 만든다.

의존성 주입 메커니즘

- 리플렉션을 통해 클래스를 인스턴스화 함
- package-private 이더라도 인스턴스를 만들 수 있다.
- 클래스패스 스캐닝을 이용해야만 한다 ... !

컴파일 후 체크

코드가 컴파일된 후 런타임에 체크

- ArchUnit
  - 의존성 방향이 기대한 대로 잘 설정돼 있는지 체크할 수 있는 API 제공
  - 의존성 규칙 위반 시 예외 throw

```
class DependencyRuleTests {

    @Test
    void domainLayerDoesNotDependOnApplicationLayer() {
        noClasses()
            .that()
            .resideInAPackage("buckpal.domain..")
            .should()
            .dependOnClassesThat()
            .resideInAnyPackage("buckpal.application..")
            .check(new ClassFileImporter()
                    .importPackages("buckpal.."));
    }
}
```

  - **단점**
    - 패키지 이름에 오타 발생 시 찾기 어려움
    - 리팩토링에 취약

빌드 아티팩트

빌드 도구

- 주 기능 : 의존성 해결
- 코드베이스가 의존하고 있는 모든 아티팩트가 사용 가능한지 확인
- 이를 활용해서 모듈/아키텍처의 계층 간 의존성을 강제할 수 있다 !

모듈

- 모듈을 세분화할수록 모듈 간 의존성을 더 잘 제어할 수 있다.
- 더 작게 분리할수록 모듈 간 매핑을 더 많이 수행해야 한다.

빌드 모듈로 아키텍처 경계 구분하기

1. 빌드 도구는 순환 의존성을 싫어한다.
    - 하나의 모듈에서 일어나는 변경이 잠재적으로 다른 모듈을 변경하게 만든다.
    - 단일 책임 원칙을 위배한다.
2. 특정 모듈의 코드를 격리한 채로 변경할 수 있다.
    - 여러 개의 빌드 모듈은 각 모듈을 격리한 채로 변경할 수 있다.
3. 모듈 간 의존성이 빌드 스크립트에 명시되어 있다.
    - 새로운 의존성을 추가하는 일은 의식적인 행동이 된다.
    - 정말로 이 의존성이 필요할까?

결론

- 소프트웨어 아키텍처는 요소 간 의존성 관리가 전부다.
- 의존성이 올바른 방향을 가리키는지 지속적으로 확인하자.
- 새로운 코드 추가, 리팩토링까지 패키지 구조를 염두에 둬야 한다.
- package-private 가시성을 이용해 패키지 바깥에서 접근하면 안되는 클래스에 대한 의존성을 피하자.
- 하나의 빌드 모듈 안에서 아키텍처 경계를 강제해야 한다.
- 아키텍처가 안정적이라고 느껴진다면 아키텍처 요소를 독립적인 빌드 모듈로 추출하라.