Clean Architecture

16 ~ 19 장

16장.독립성

좋은 아키텍처는 시스템의 유스케이스, 운영, 개발, 배포를 지원해야 한다

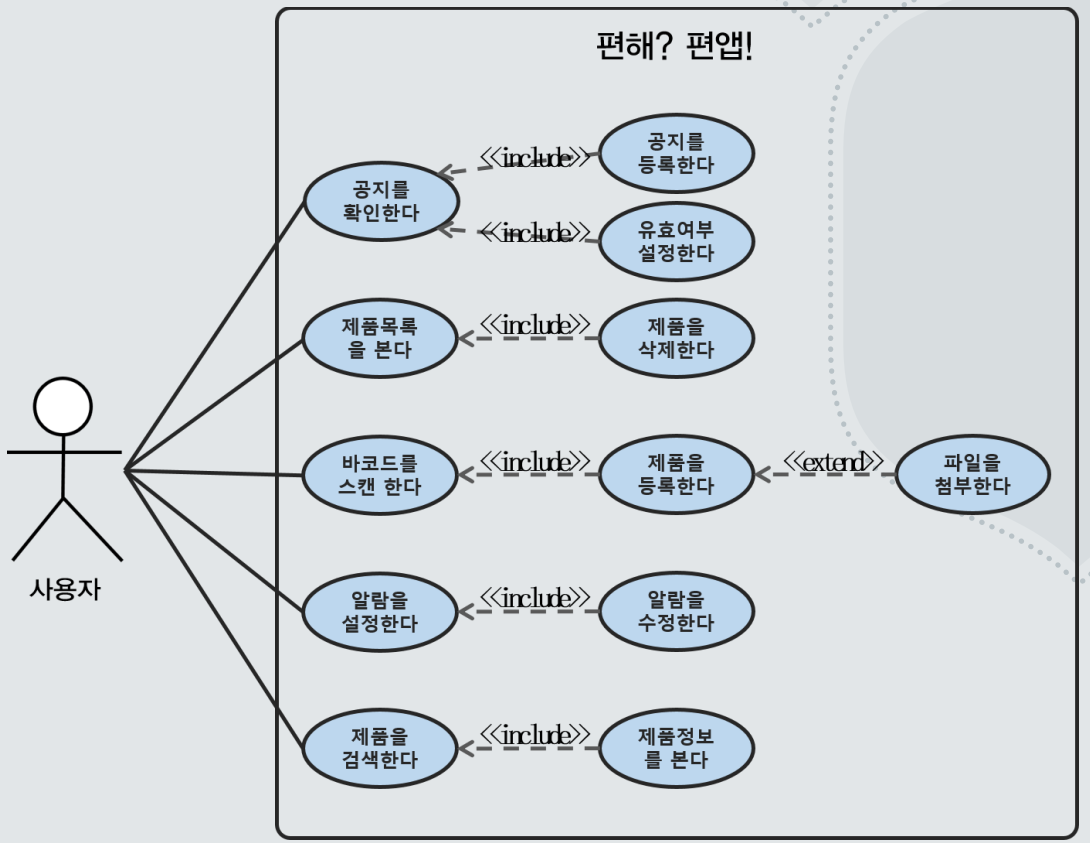
유스케이스

- 시스템 아키텍처는 시스템의 의도를 지원해야 한다는 의미
- 아키텍트의 최우선 관심사이자 아키텍처의 최우선 순위
- 아키텍처는 행위를 명확하게 하고 외부로 드러내야 함

➔ 아키텍처 수준에서 시스템의 의도를 파악할 수 있음

유스케이스(use case)란?

- 시스템의 동작을 사용자 입장에서 표현한 시나리오
- 시스템의 요구사항을 알아내는 과정



16장.독립성

좋은 아키텍처는 시스템의 유스케이스, 운영, 개발, 배포를 지원해야 한다

운영

- 상황에 맞는 형태로 아키텍처를 구조화해야 함
- 개선 가능성, 요구 사항 등을 고려해야 함
  Ex) 병렬프로그래밍 여부, 독립/공유 주소 공간 사용

개발

- 개발환경을 지원하는데 아키텍처는 핵심적인 역할을 수행함
- 콘웨이의 법칙이 작용하는 부분
- 팀 컬러/목적에 맞는 아키텍처를 확보하고 개발해야 함
  Ex) 컴포넌트간 독립성 보장

콘웨이의 법칙
시스템을 설계하는 조직이라면 어디든지 그 조직의 의사 소통 구조와 동일한 구조의 설계를 만들어 낼 것이다.

배포

- 아키텍처는 배포의 용이성을 결정하는데 중요한 역할을 함
- 절대적인 목표는 즉각적인 배포이다
  - 시스템 빌드 후 즉각 배포

16장.결합 분리

계층 결합 분리

- 유스케이스를 전부를 알지는 못하지만 기본적인 의도를 알고 있음

➔ 의도의 맥락에 따라 시스템을 나눔 By SRP & CCP
➔ 시스템을 수평적인 계층으로 분리할 수 있음

유스케이스 결합 분리

- 유스케이스 그 자체를 시스템을 분할하는 기준으로 사용

➔ 시스템을 수직적인 계층으로 분리할 수 있음
➔ 기존 요소에 지장을 주지 않고 새로운 유스케이스를 추가할 수 있음

단, 계층을 분리할 때 중복을 조심해야 함

운영 관점에서의 의미

분리된 시스템에 따라 최적화된 선택이 가능함 – ex) 서버 종류

+결합 & 배포 독립성 확보

16장.결합 분리

계층과 유스케이스의 결합 분리 방법

소스 수준 분리 모드 (모놀리틱 구조)

- 소스 코드 모듈 사이의 의존성 제어
- 모든 컴포넌트가 같은 주소 공간에서 실행
- 함수 호출을 통한 통신

배포 수준 분리 모드

- 배포 가능한 단위(jar, DLL, 공유라이브러리) 사이의 의존성 제어
- 많은 컴포넌트가 같은 주소 공간에서 실행
- 함수 호출을 통한 통신
- 결합이 분리된 컴포넌트가 독립적으로 배포할 수 있는 단위로 분할되어 있음

서비스 수준 분리 모드

- 데이터 구조 단위까지 의존성 제어
- 네트워크 패킷을 통한 통신

➔ 소스와 바이너리 변경에 대해 완전히 독립적인 실행

17장.경계: 선 긋기

- 선은 한편의 요소가 반대편의 요소를 알지 못하도록 만든다
- 초기에 그어지는 선들은 가능한 오랫동안 결정을 연기시키기 위해 그어진다
- 그 결과로, 핵심적인 업무 로직을 오염시키지 못하게 만드는 목적으로 쓰인다

어떻게 선을 그을까?

관련이 있는 것과 없는 것 사이에 선(경계)을 긋는다 – 비즈니스 규칙 | 데이터베이스 | GUI

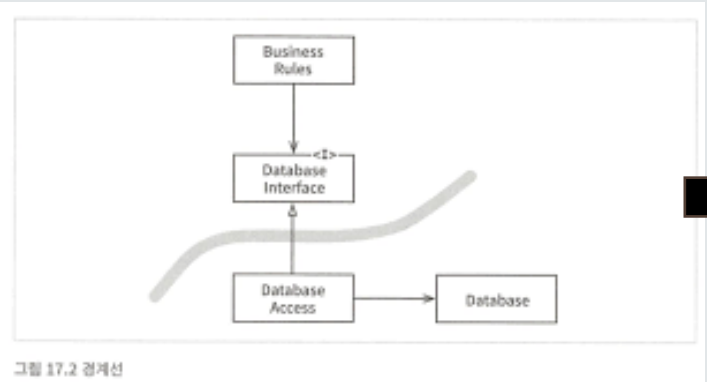


그림 17.2 경계선

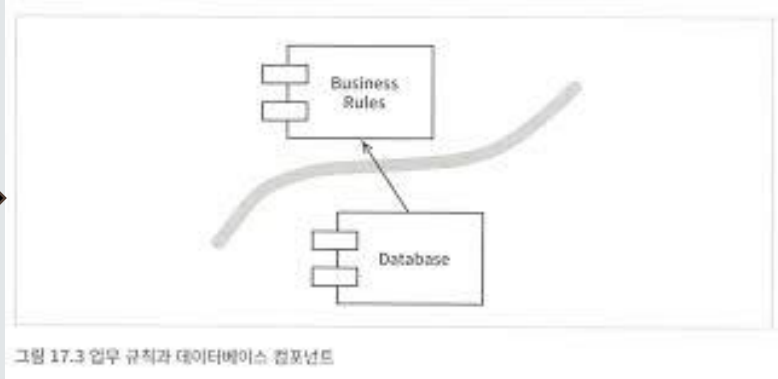


그림 17.3 업무 규칙과 데이터베이스 컴포넌트

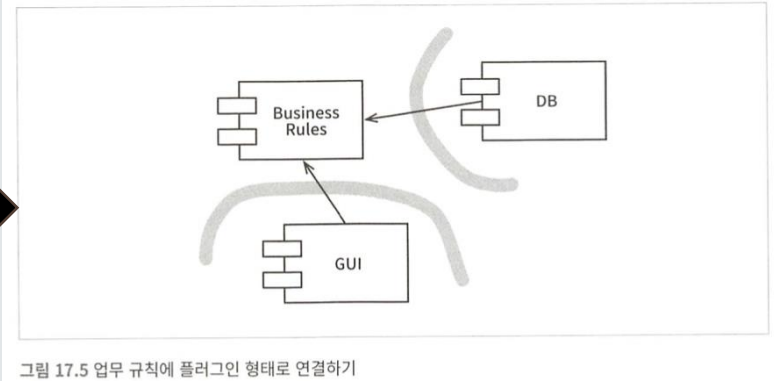


그림 17.5 업무 규칙에 플러그인 형태로 연결하기

- 업무 규칙과 DB 사이에 선을 그어서 DB에 대한 생각을 미룰 수 있게 되었다
- 업무 규칙에서 나오는 화살표가 없으므로 업무 규칙은 다양한 구현체를 교체할 수 있다

➔ 경계가 잘 그어지면 플러그인 구조를 띈다

➔ 생성/확장/유지보수가 편하다

18장.경계 해부학

런타임에 경계를 횡단한다: 경계 한쪽에 있는 기능이 반대편 기능을 호출하여 데이터를 전달하는 일

소스 코드가 변경되면 의존하는 코드들도 변경 및 컴파일이 필요할 수 있음
➔ 적절한 위치에서 경계를 횡단 해야 함 (By **소스 코드** 의존성 관리)
경계는 이러한 변경의 전파를 막고 관리하는 수단으로 존재함

두려운 단일체

- 배포 관점에서 경계가 들어나지 않음 (파일 하나만 배포하므로)
- 동적 다형성에 의존해서 내부 의존성 관리

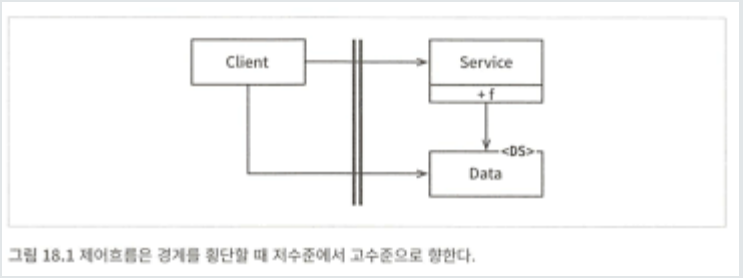


그림 18.1 제어흐름은 경계를 횡단할 때 저수준에서 고수준으로 향한다.

저수준 클라이언트와 고수준 서비스

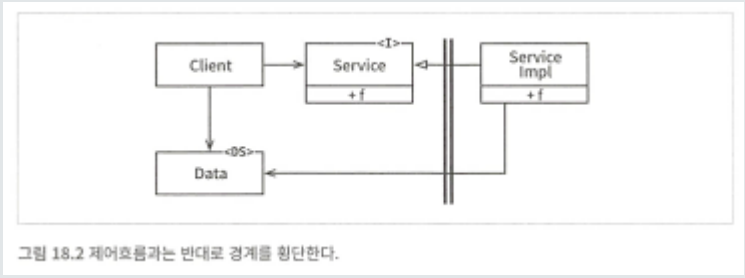


그림 18.2 제어흐름과는 반대로 경계를 횡단한다.

고수준 클라이언트와 저수준 서비스

위처럼 규칙적인 방식으로 구조를 분리하면 모노리틱 구조라도 개발,
테스트, 배포에 큰 도움이 된다

18장.경계 해부학

배포형 컴포넌트

- 아키텍처의 경계가 물리적으로 드러날 수 있다
- 배포 과정에서만 차이가 날 뿐, 배포 수준의 컴포넌트는 단일체와 동일하다
- 컴포넌트를 분리하거나 의존성을 관리하는 전략도 단일체와 동일하다
- 당연히 경계를 가로지르는 통신은 함수 호출이므로 값싸다

로컬 프로세스

- 아키텍처의 경계가 강한 물리적 형태를 띤다
- 소스코드 의존성인 화살표는 항상 고수준 컴포넌트를 향한다
- 경계를 지나는 통신은 비싼 작업이므로, 통신이 너무 빈번하게 이뤄지면 안된다

서비스

- 아키텍처의 경계가 가장 강한 물리적 형태를 띤다
- 모든 통신은 네트워크를 통해 이뤄지므로 함수 호출에 비해 느리다
- 고수준에서 지연에 따른 문제를 처리해야 한다
- 저수준 서비스는 고수준 서비스에 플러그인 되어야 한다

19장.정책과 수준

소프트웨어 시스템이란?

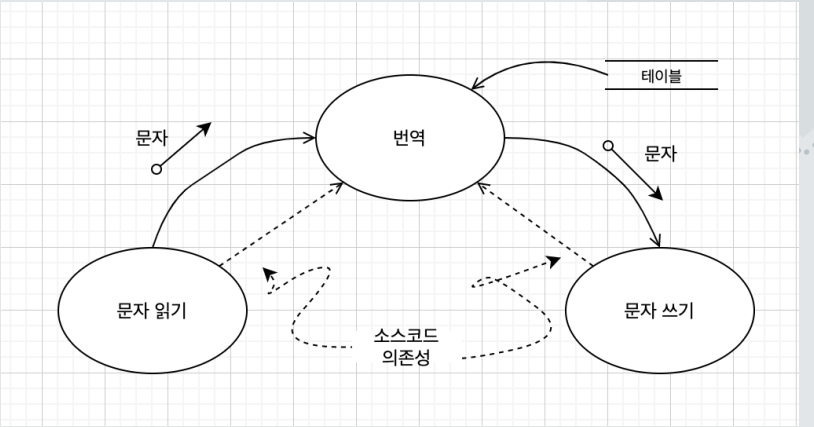
- 정책을 기술한 것
  +하나의 정책을 서술하는 여러 개의 작은 정책들이 있음

아키텍처 개발 기술

- 정책을 분리 & 재편성 하는 일을 포함함
- 컴포넌트들을 비순환 방향 그래프로 구성함
  - node: '동일 **수준**'의 정책을 포함하는 컴포넌트
  - Edge: 컴포넌트 간 의존성 & 다른 수준에 위치한 컴포넌트 연결

수준이란?

- 입력과 출력까지의 거리
- 즉, 입력과 출력 모두로부터 멀리 위치하면 정책의 수준은 높아짐
- 데이터의 흐름과 소스 코드 의존성이 항상 같은 방향을 가리키지 않음



19장.정책과 수준

정책을 컴포넌트로 묶는 기준은 정책이 변경되는 방식에 달려있다

```
function encrypt(){
  while(true)
    writeChar(translate(readChar());
}
```