Clean Architecture

소프트웨어 구조와 설계의 원칙

30장 - 데이터베이스는 세부사항이다 &
31장 - 웹은 세부사항이다 &
32장 - 프레임워크는 세부사항이다 &
33장 - 사례 연구: 비디오 판매 &
34장 - 빠져있는 장

Date
2023.09.22

Presenter
황유란

데이터베이스와 아키텍처

- 30~34장은 6장 세부사항의 내용
- 아키텍처 관점에서 데이터베이스는 **엔티티가 아니다**.
- 데이터베이스는 **데이터 모델이 아니다**.

관계형 데이터베이스

- **데이터를 행 단위로 배치한다**는 자체는 아키텍처적으로 볼 때 **전혀 중요하지 않다.**
- **유스케이스는 이를 알아서도 안되고 관여해서도 안된다.**
- 관계형 데이터베이스라는 사실은 **최하위의 수준의 유틸리티 함수만 알아야** 한다.

데이터베이스 시스템은 왜 이렇게 널리 사용되는가?

- 디스크 때문
- 데이터는 디스크에 존재 - 느림 - 최적화를 위해 색인,캐시,쿼리 같은 걸 사용 - 이를 사용하려면 데이터들이 어떤 데이터인지 알 수 있어야 한다 - 데이터 접근 및 관리 시스템이 필요 - RDBMS 등장

디스크가 없다면?

- 데이터들은 트리, 스택, 해시 테이블 같은 **데이터 구조로 변경해서 사용**할 것
    - 데이터를 파일이나 테이블 형태 그대로 사용하지 않는다.

세부사항

- **데이터베이스는** 메커니즘일뿐, **데이터를 담는 공간**에 지나지 않는다.
- 아키텍처 관점에서는 데이터가 존재하기만 하면 데이터가 어떻게 저장되고 있는지 신경써서는 안된다.

하지만 성능은?

- 데이터 저장소의 성능은 업무 규칙과 분리할 수 있는 관심사
- 전반적인 아키텍처와는 아무런 관련이 없다.

결론

- 체계화된 **데이터 구조와 데이터 모델은 아키텍처적으로 중요**하다.
- 데이터베이스는 세부사항이다.

끝없이 반복되는 추

- 웹은 아무것도 바꾸지 않았다.
- ex. Q사, 스마트폰 A사
- 마케팅 귀재는 어디나 존재하며, **아키텍트는 UI와 업무규칙을 서로 격리했어야** 한다.

요약

- GUI는 세부사항이다. -> 웹은 GUI다. -> 따라서, **웹은 세부사항**이다.
- **애플리케이션과 GUI의 상호작용은 빈번**하다.
- 하지만, UI와 애플리케이션 사이에는 추상화가 가능한 경계가 존재
- 업무로직은 다수의 유스케이스로 존재
- 유스케이스는 사용자를 대신해서 함수를 수행
- 유스케이스는 입력 데이터, 수행할 처리 과정, 출력 데이터를 기반으로 동작
- **UI와 애플리케이션 사이에 입력 데이터가 구성됨 → 유스케이스 실행 가능**

결론

- 이러한 추상화를 제대로 만들려면 수차례의 반복 과정을 거쳐야 한다.
- **웹은 세부사항**이다.

프레임워크 제작자

- **프레임워크는 아키텍처가 될 수 없다.**
- 프레임워크가 풀려는 문제와 당신의 문제는 꽤 많이 겹칠 것
- 겹치는 영역이 클수록 프레임워크는 더 인기를 끌고 유용해진다.
- **프레임워크는 당신의 문제를 해결하기 위함이 아니다.**

혼인 관계의 비대칭성

- 당신과 프레임워크 제작자 사이 관계는 **비대칭적**
- 당신 → 프레임워크에 헌신하는 구조
- 당신이 import를 해서 프레임워크를 사용하고 많이 결합될수록 프레임워크 제작자는 자신의 프레임워크가 효과적임을 증명하는셈

위험 요인

- 프레임워크 아키텍처는 그다지 **깔끔하지 않은 경우**가 많다.
- 프레임워크는 애플리케이션 초기 기능을 만드는 데 도움이 될 것이지만 **제품이 성숙되면 그 틀을 벗어나게 될 것**
- 프레임워크는 당신에게 **도움되지 않는 방향으로 진화**할 수 있다.
- **새로 나온 프레임워크**로 갈아타고 싶다.

해결책

- 프레임워크가 결혼하지 말라!
- **프레임워크를 사용할 수는 있되 세부사항 그 안쪽으로는 들어오지 못하게 하라.**
- C++ & STL, JAVA & 표준 라이브러리는 인정

결론

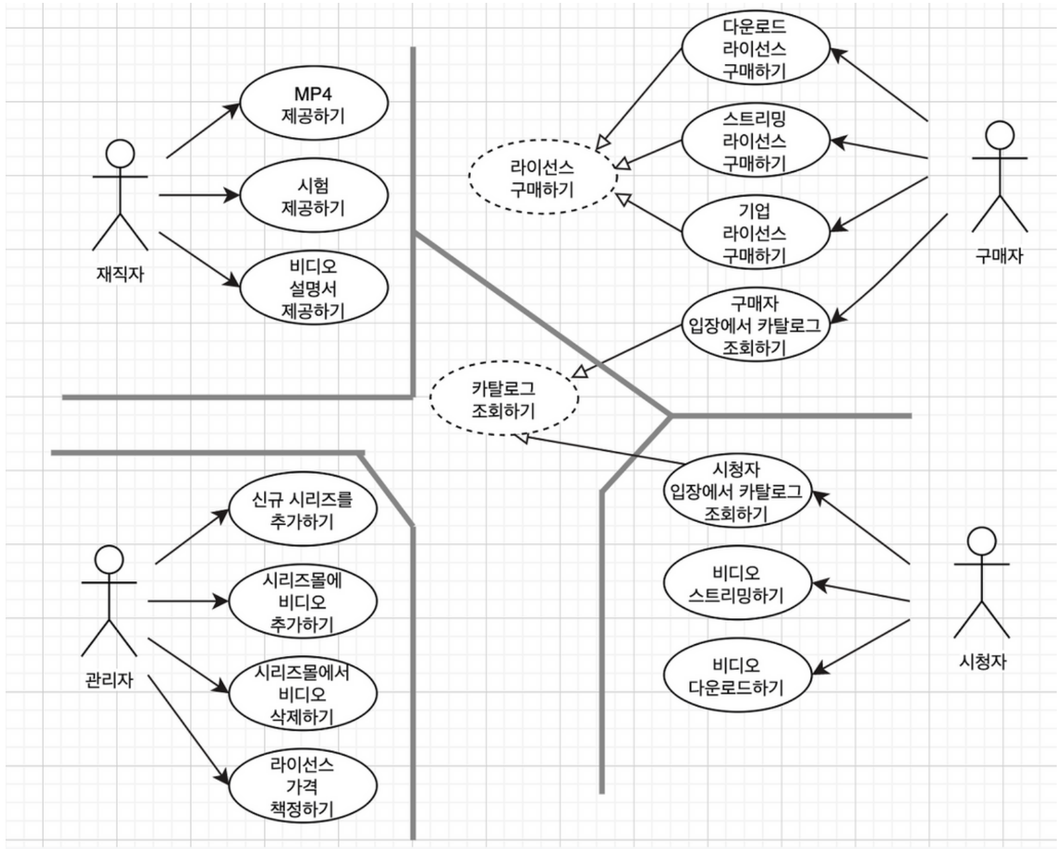
- 프레임워크를 가능한 한 **오랫동안 아키텍처 경계 너머**에 두자

비디오 판매

- 웹사이트에서 비디오를 판매하는 소프트웨어를 만들자.
- 초기 아키텍처 결정 첫단계 - **액터와 유스케이스를 식별**

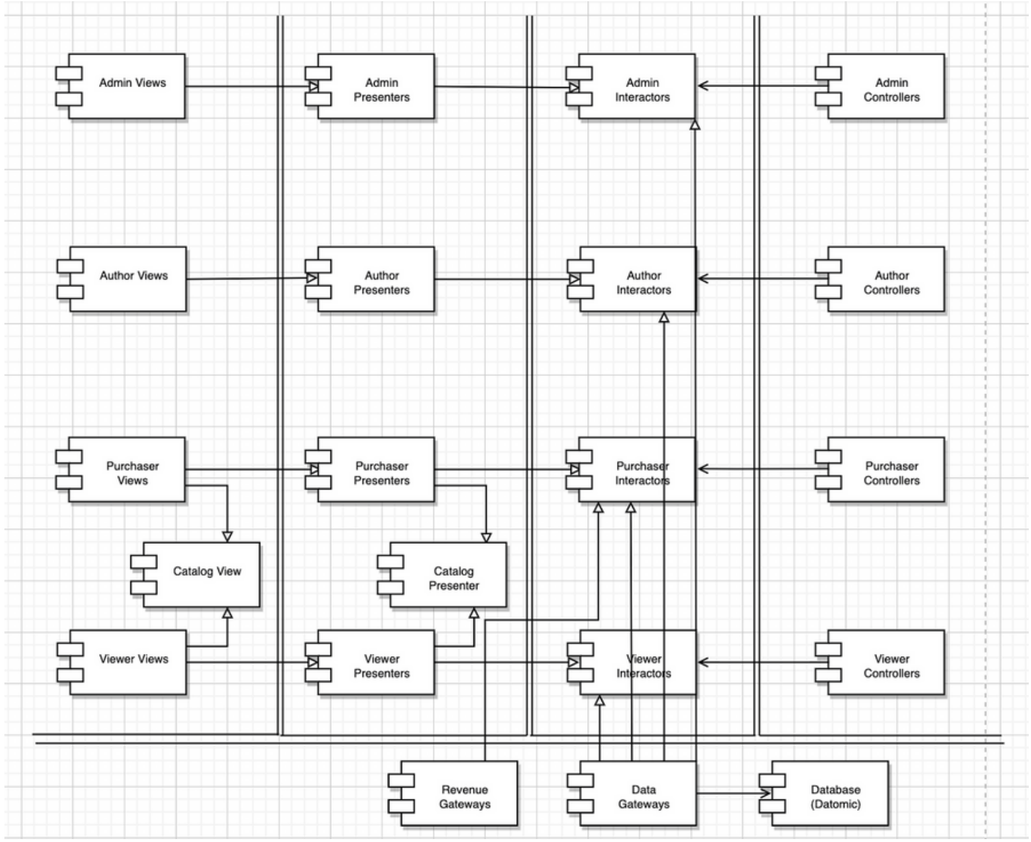
유스케이스 분석

- 시스템의 변경 이유는 네 액터가 주요 원인
- 특정 액터를 위한 변경이 다른 액터에게 영향을 미치지 못하게 하자.
- 점선 유스케이스는 추상케이스



컴포넌트 아키텍처

- 액터와 유스케이스를 식별했으면, 예비 단계의 컴포넌트 아키텍처를 만들어 볼 수 있다.
- 이중선 - 아키텍처 경계
- 액터에 따라 카테고리 분리
- 이 컴포넌트를 몇개의 jar로 합쳐서 관리할지는 열어둔다.
  - 배포 방식에 따라서 조정



의존성 관리

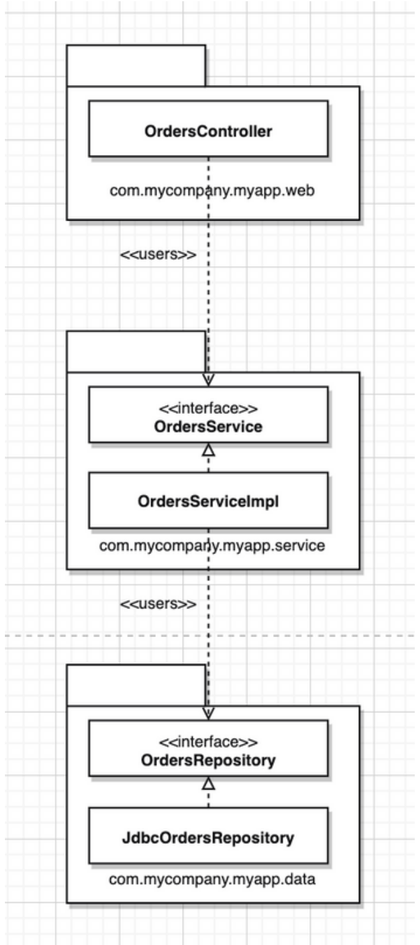
- 제어흐름은 오른쪽에서 왼쪽으로 이동
- 하지만, **모든 화살표가 오른쪽에서 왼쪽을 가리키지는 않음**

결론

- 코드를 한 번 구조화하고 나면 시스템을 실제로 배포하는 방식은 다양하게 선택할 수 있게 된다.

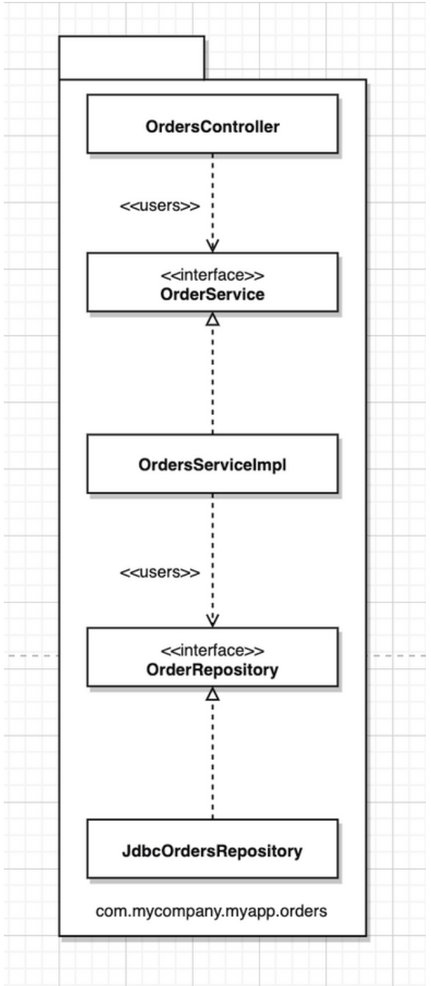
계층 기반 패키지

- **수평 계층형** 아키텍처
- 해당 코드가 하는 일에 기반해 코드를 분할
- **의존성은 모두 아래를** 향한다.
- **가장 빠르게** 무언가를 작동시켜주는 방법
- 문제
  - 소프트웨어가 커지고 복잡해지면 세 그릇가지고는 코드는 다 담기에 부족함을 느낄 것
  - **업무 도메인에 대해 아무것도** 말해주지 않는다.
    - 전혀 다른 도메인이더라도 아키텍처가 비슷해보인다.



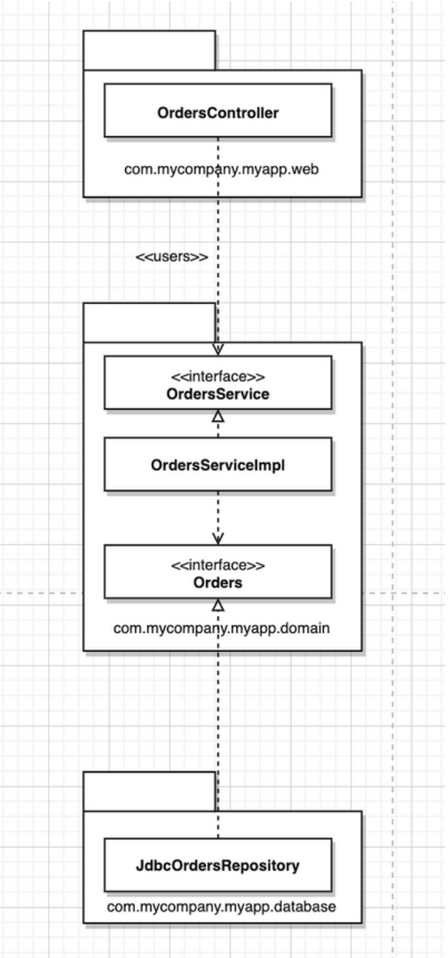
기능 기반 패키지

- **서로 연관된 기능**, 도메인 개념 또는 Arggregate Root (데이터 변경 단위 연관 객체의 묶음) 에 기반하여 수직으로 코드를 나누는 방식
  - **모두가 하나의 패키지**에 속하게 된다.
- 장점
  - 코드의 상위 수준 구조가 **업무 도메인에 대해 무언가**를 알려주게 된다.
    - 이게 이 코드가 주문 관련 일을 한다는 걸 볼 수 있음
  - '주문 조회하기' 유스케이스가 변경될 경우 **변경해야 할 코드를 모두 찾는 작업이 쉬워짐**
    - **모두 다 한 패키지에 있으니까**



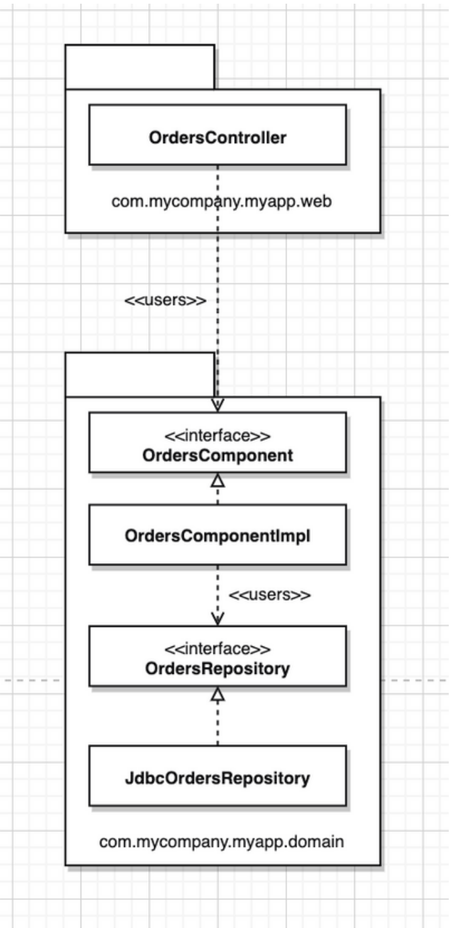
포트와 어댑터

- ‘포트와 어댑터’, ‘육각형 아키텍처’, ‘경계,컨트롤러, 엔티티’ 등의 방식으로 접근하는 이유는 **업무/도메인에 초점을 둔 코드가 프레임워크나 데이터베이스 같은 세부 내용과 독립적이고 분리된 아키텍처를 만들기 위함**
- 코드 베이스는 **내부(도메인)과 외부(인프라)로 구성**되어 있다.
- 주요 규칙
  - **외부 → 내부를 의존**해야 한다.
- com.mycompany.myapp.domain이 내부, 나머지는 외부
- OrderRepository가 Orders로 변경
  - 도메인 기준으로는 주문에 대해 말하는 것이지 OrderRepository를 말하는게 아님



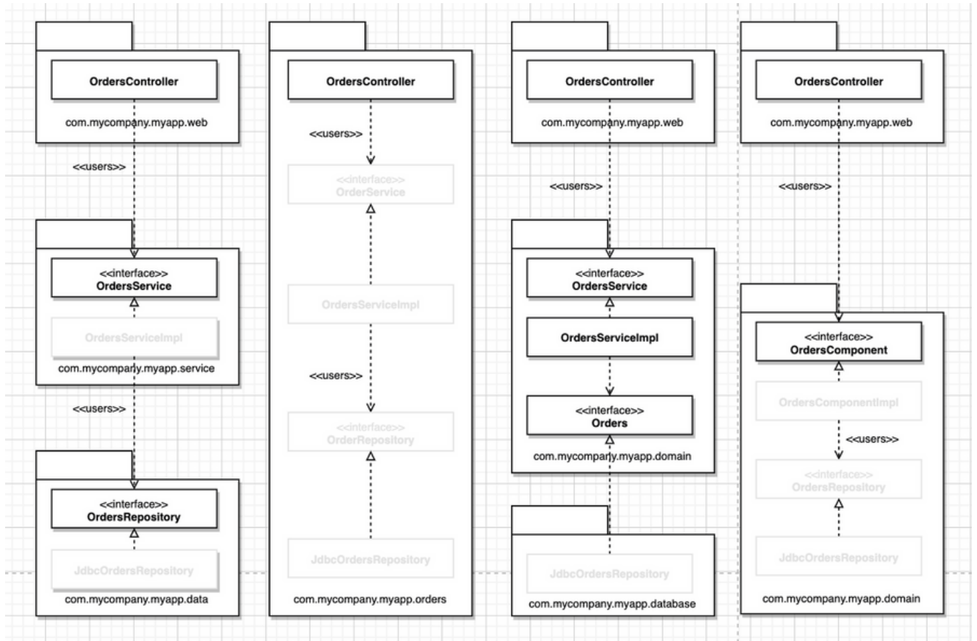
컴포넌트 기반 패키지

- **계층형 아키텍처**
  - 의존하려면 인터페이스를 public으로 선언해야
  - Controller가 Service를 건너뛰로 바로 Repository를 의존하게 구현하게 될 수도 있다.
  - “**/web 패키지에 있는 타입은 절대 **/data에 있는 타입에 접근하면 안 된다” 같은 규칙을 만들어야 한다.
- 컴포넌트 기반 패키지는 **큰 단위의 단일 컴포넌트와 관련된 모든 책임을 하나의 자바 패키지로** 묶는다.
- 업무 로직과 관련된 코드를 하나로 묶음
  - 코딩해야 할 때 OrdersComponent만 둘러보면 된다



구현 세부사항에는 항상 문제가 있다.

- **조직화 vs 캡슐화**
- public을 사용하면 패키지는 단순히 조직화를 위한 매커니즘으로 전락한다.
- 앞에서 설명한 네가지 아키텍처는 본질적으로 같아진다.
- **표시되어 있는 부분만 public으로 선언해야**한다.



다른 결합 분리 모드

- 프로그래밍 언어가 제공하는 방법 외에 **모듈로 소스 코드 의존성을 분리**하는 방법 존재
- 소스 코드 수준에서 서로 다른 **소스 코드로 트리로 분리하여 의존성을 분리**하는 방법도 존재
    - 도메인 코드 트리, 인프라 코드 트리 두개로 나누기

결론: 빠져 있는 조언

- **최적에 설계를 했어도 구현 전략에 얽힌 복잡함을 고려하지 않으면 설계가 망가질 수 있다.**
- 설계를 어떻게해야 원하는 코드 구조로 매핑할 수 있을지 고민하라
- 해당 코드를 어떻게 조직화 할 지 고민하라
- 런타임, 컴파일타임에 어떤 결합 분리모드를 적용할지를 고민하라
- 가능하면 선택사항은 열어두되 실용주의적으로 행하라
- 팀규모, 기술 수준, 해결책의 복잡성을 일정과 예산이라는 제약과 동시에 고려해라
- 선택한 아키텍처 스타일을 강제하는 데 컴파일러의 도움을 받을 수 있는지 고민하라
- 데이터 모델과 같은 영역에 결합되지 않도록 주의하라