## TODAY.....

도메인 영역의 구성요소와 Infrasturcture

※ 네 개의 영역

- 표현
  - User Interface & System Interface
  - 전달 & 응답
- 응용
  - **운전자**(운전대를 잡고 ~한다 =>**Call** 한다)
  - 로직수행을 위임한다 => 상호작용 GOOD
- 도메인
  - 도메일 모델 & 도메인의 핵심 로직 구현
- Infrastructure
  - 구현기술

## ※ 도메인 영역의 구성요소

- **엔티티**
  - 고유 식별자 가진다
    - ⇨ 레코드 구분 가능
  - 데이터 + **도메인 기능** 제공 (p.56)
    - ⇨ 이때 도메인 관점에서 기능을 구현하고, 캡슐화를 해서 데이터가 임의로 변경되는것을 막는다
  - **벨류 타입을 이용해서 표현 가능** (p.56 ,Order 엔티티, Orderer 벨류타입)
- **벨류**
  - DTO 같은 , **값을 담는 그릇**의미 -> 추상화 작업 (이름짓기 중요)
  - 사용이유 :
    - 두개 이상 데이터표현 (p.56 ,Orderer)
    - 의미를 명확히 하기위해서 (p.25 ,Money)
  - 불변으로 구현 (p.58) 권장
    - 데이터 변경시 , 객체 자체를 완전히 교체한다
    - **장점** : 안전한 코드 작성
      - 밸류 타입에는 set 메서드를 구현하지 않게된다.

- **애그리거트**
  - **발생** : 도메인이 커질수록 , 개발한 도메인 모델도 커지면서 많은 엔티티와 벨류가 출현
    - ⇨ 점점 복잡해진다
    - **문제** : 전체 구조가 아닌, 한개의 엔티와 벨류에만 집중하게됨
      - ⇨ 상위 수준에서 모델을 볼수 있어야 한다
  - **관련** 객체를 **하나**로 묶은 **군집** (그림 2.17)
    - ➔ 개념상 완전한 한 개의 도메인 모델을 표현
  - **특징**
    - **독립된** 객체 **군** = 경계를 가진다
      - 애그리거트 **자기 자신만** 관리한다
    - 군집에 속한 객체들을 관리하는 **루트엔티티**를 갖는다 (그림 2.18)
      - 애그리거트에 속해있는 엔티티,벨류를 이용해서 애그리거트가 구현해야 할 기능을 제공
      - 루트를 **통해**서 **간접적** 으로 애그리거트 내의 다른 엔티티,벨류에 접근
        - 내부 구현을 숨겨서 **애그리거트 단위로 구현을 캡슐화** 가능하게 한다
        - 루트(Oder)를 통하지 않고 shippingInfo 를 변경할수 없다 (p.60 코드)

- **경계 설정 기준** => 도메인 규칙 , 요구사항

  - 유사하거나 동일한 라이프 사이클을 갖는경우
    - 관련 모델을 하나로 모은 것 이기 때문에
  - 변경 주체가 같은 경우
    - 애그리거트 **자기 자신만** 관리한다

- **장점**
  - 복잡한 도메인 모델을 관리하는데 도움
    - 개별 객체가 아닌 , 애그리거트 간의 관계로 이해 하고 구현
      - 큰 틀에서 도메인 모델을 관리
  - 도메인 기능을 확장하고 변경하는데 필요한 시간 줄어든다
    - 애그리거트 단위로 일관성을 관리

> 복잡한 도메인을 단순한 구조로 만든다

## Repository

- **객체의 영속성을 처리한다**
- 애그리거트 단위로 존재한다 => 애그리거트 **루트 기준**으로 작성
  - 루트는 애그리거트에 속한 모든 객체를 포함하고 있으므로, 결과적으로 애그리거트 단위로 저장하고 조회한다
- 사용
  - **주체** : **응용 서비스**
  - EX ) p.62
    - 도메인 모델을 사용해야 하는 코드는 repository 를 통해서 도메인 객체를 구한뒤
    - 도메인 객체의 기능을 실행한다
  - repository 최소 **필요 메소드 2 개**
    - 아이디로 애그리거트 조회
    - 애그리거트 저장
- **모듈 구조**(그림 2.19)
  - **repository 인터페이스**는, 도메인 객체를 영속화 하는데 필요한 기능을 **추상화** 한것으로 **고수준** 모듈로 **도메인모델** 영역에 속한다
  - **실제 구현** 클래스는 **저수준** 모듈로 **인프라스트럭처** 영역에 속한다

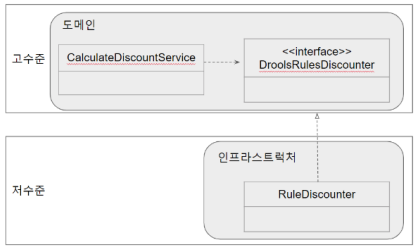
- **구현** : 애그리거트를 어떤 저장소 (RDMS,NOSQL)에 저장하느냐에 따라서, 구현 방법이 다르다 =>책에서는 JPA

## 도메인 서비스

- 특정 엔티티에 속하지 않은 도메인 로직을 제공
- 도메인 로직이 여러 엔티티와 벨류를 필요할 경우 사용한다

## ※ Infrasturcture

- 응용 영역과 도메인 영역에서 구현 기술에 대한 의존을 가진다
  - **문제** : 구현기술 변경 어려움 & 테스트 어려움
    - **방안** : DIP
      - 반대로 저수준 모듈이 고수준 모듈에 의존하도록 한다
      - 도메인 영역과 응용영역에서 정의한 인터페이스를 Infrasturcture 영역에서 구현 하는 방식으로 사용




( Gof 의 Strategy pattern 연결)

- 하지만, 무조건 Infrasturcture 에 대한 의존을 없애는 것은 좋은 것이 아니다
  - EX )
    - (p66) 스프링 @Transaction 사용

      -> 응용영역에서 Infrasturcture(스프링 프레임워크) 의존
    - (p67) JPA 전용 어노테이션

      -> 도메인 영역에서 Infrasturcture 의존

      > **장점** : 구현의 편리함

- DIP 의 장점을 해치지 않는 범위에서, 응용영역과 도메인 영역에서 구현 기술에 대한 의존을 가져가야한다

## ※ Gof 의 Strategy pattern

- 특징
  - 바뀌는 부분은 따로 뽑아서 **캡슐화** 하고 해당 기능을 **인터페이스에 위임**
    - 나중에 바뀌지 않는 부분에는 영향을 미치지 않은 채로 그 부분만 고치거나 확장 할수 있다
- 구현 방법
  - 변하는 부분 캡슐화
  - 인터페이스에 위임
  - 각 객체 구현
- 장점
  - If 분기문을 푸는 방식 중 한 개
  - 객체 또는 기능이 추가/변경 되더라도 쉽고 간단하게 적용 가능
  - 코드의 중복없이 재사용 가능
- 단점
  - 개수가 늘어난다