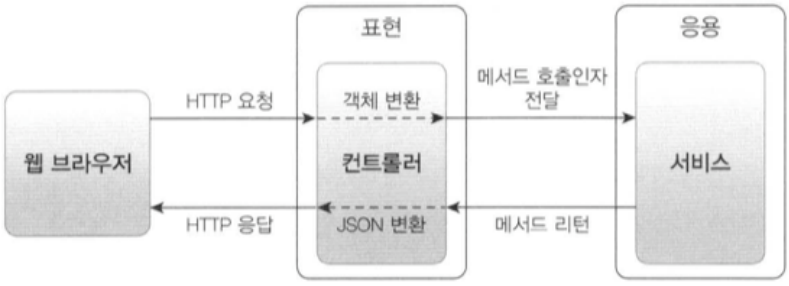
02. 아키텍처 개요

version 2018.35

네 개의 영역



[그림 2.1] 표현 영역은 사용자의 요청을 해석해서 응용 서비스에 전달하고 응용 서비스의 실행 결과를 사용자가 이해할 수 있는 형식으로 변환해서 응답한다.

아키텍처를 설계할 때 전형적인 영역이 '표현, '응용', '도메인', '인프라스트럭처' 네 개의 영역이다.

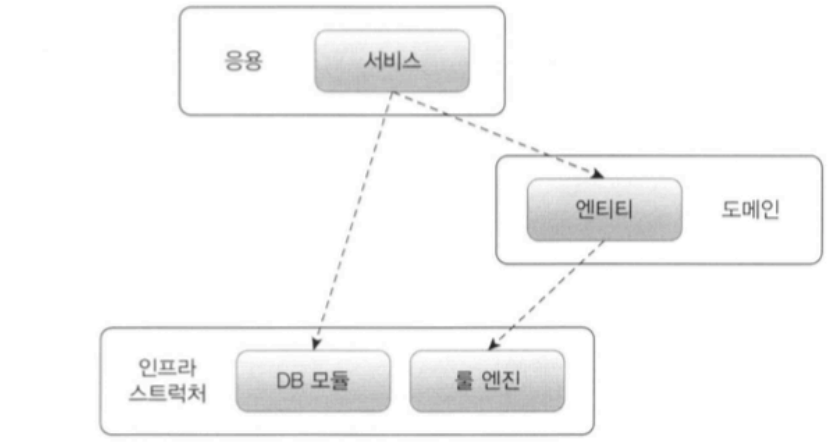
- 표현 영역: 사용자의 요청을 받아 응용 영역에 전달하고 처리 결과를 다시 사용자에게 보여주는 역할.
- 응용 영역: 시스템이 사용자에게 제공해야 할 기능을 구현. 기능을 구현하기 위해 도메인 영역의 도메인 모델을 사용. 로직을 직접 수행하기 보다는 도메인 모델에 로직 수행을 위임.
    - 주문등록, 주문 취소, 상품 상세 조회
- 도메인 영역: 도메인 모델을 구현. 도메인 모델은 도메인의 핵심 로직을 구현.

    - 배송지 변경, 결제완료, 주문 총액 계산
- 인프라스트럭처 영역: 구현 기술을 다룸. 논리적인 개념을 표현하기보다는 RDBMS 연동처리와 같은 실제 구현을 다룸.

계층 구조 아키텍처

> 표현 -> 응용 -> 도메인 -> 인프라스트럭처

네 영역을 구성할 때 많이 사용하는 아키텍처가 위와 같은 계층구조이다. 계층 구조는 특성상 상위 계층에서 하위 계층으로의 의존만 존재하고 반대는 존재하지 않는다. 하지만 구현의 편리함을 위해 계층 구조를 유연하게 적용하기도 한다. 예를 들어, 응용 계층은 바로 아래 도메인 계층에 의존하지만 외부 시스템과 연동을 위해 더 아래 계층인 인프라스트럭처 계층에 의존하기도 한다.



[그림 2.5] 전형적인 계층 구조상의 의존 관계

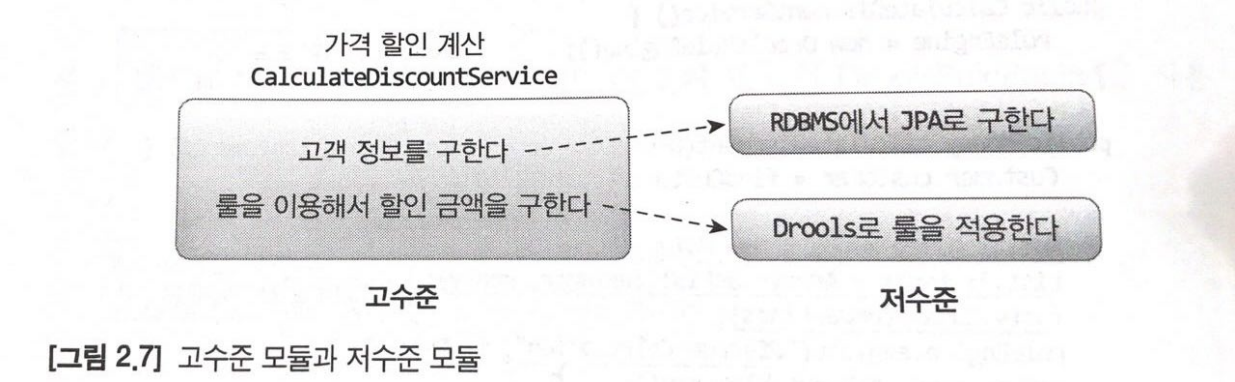
이러한 계층 구조를 사용하는 것은 직관적으로 이해하기 쉽지만 표현, 응용, 도메인 계층이 상세한 구현 기술을 다루는 인프라스트럭처 계층에 의존한다는 문제점이 있다. 아래의 CalculateDiscountService를 살펴보자.

```
public class CalculateDiscountService {
    private DroolsRuleEngine ruleEngine;

    public CalculateDiscountService() {
        ruleEngine = new DroolsRuleEngine();
    }

    public Money calculateDiscount(List<OrderLine> orderLines, String
customerId) {
        Customer customer = findCustomer(customerId);

        MutableMoney money = new MutableMoney(0);
        List<?> facts = Arrays.asList(customer, money);
        facts.addAll(orderLines);
        ruleEngine.evalute("discountCalculation", facts);
```

```
            return money.toImmutableMoney();
        }
    }
```

이렇게 의존관계가 형성됐을 때 첫번째 문제는 테스트가 어렵다는 것이고 두번째 문제는 구현 방식을 변경하기 어렵다는 점이다.

DIP(Dependency Inversion Principle)



[그림 2.7] 고수준 모듈과 저수준 모듈

> 고수준 모듈: 의미 있는 단일 기능을 제공하는 모듈.
>
> 저수준 모듈: 고수준 모듈의 기능을 구현하기 위한 하위 기능을 실제로 구현한 것.

DIP는 앞서 계층 구조 아키텍처에서 언급했던 두 가지 문제를 해결하기 위해 저수준 모듈이 고수준 모듈에 의존하도록 바꾼다.

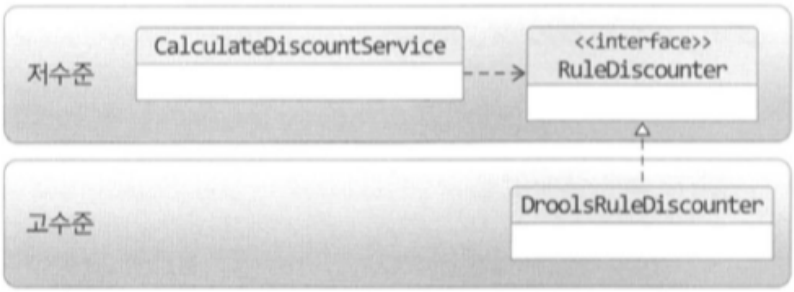
'룰을 이용해서 할인 금액을 구한다'라는 하위기능을 추상화한 인터페이스는 다음과 같다.

```
public interface RuleDiscounter {
    public Money applyRules(Customer customer, List<OrderLine> orderLines);
}
```

아래의 코드는 RuleDiscounter 인터페이스를 적용한 CalculateDiscountService이다.

```
public class CalculateDiscountService {
    private RuleDiscounter ruleDiscounter;

    public CalculateDiscountService(RuleDiscounter ruleDiscounter) {
        this.ruleDiscounter = ruleDiscounter;
    }

    public Money calculateDiscount(List<OrderLine> orderLines, String
customerId) {
        Customer customer = findCustomer(customerId);
```

```
        /*
        MutableMoney money = new MutableMoney(0);
        List<?> facts = Arrays.asList(customer, money);
        facts.addAll(orderLines);
        ruleEngine.evalute("discountCalculation", facts);
        return money.toImmutableMoney();
        */
        return ruleDiscounter.applyRules(customer, orderLines);
    }
}
```

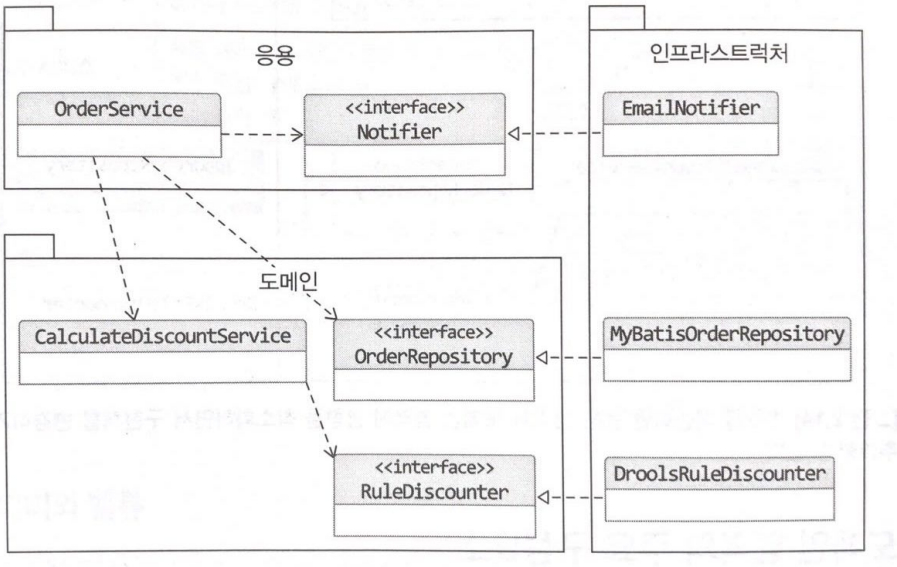


[그림 2.9] 저수준 모듈이 고수준 모듈에 의존(상속은 의존의 다른 형태)

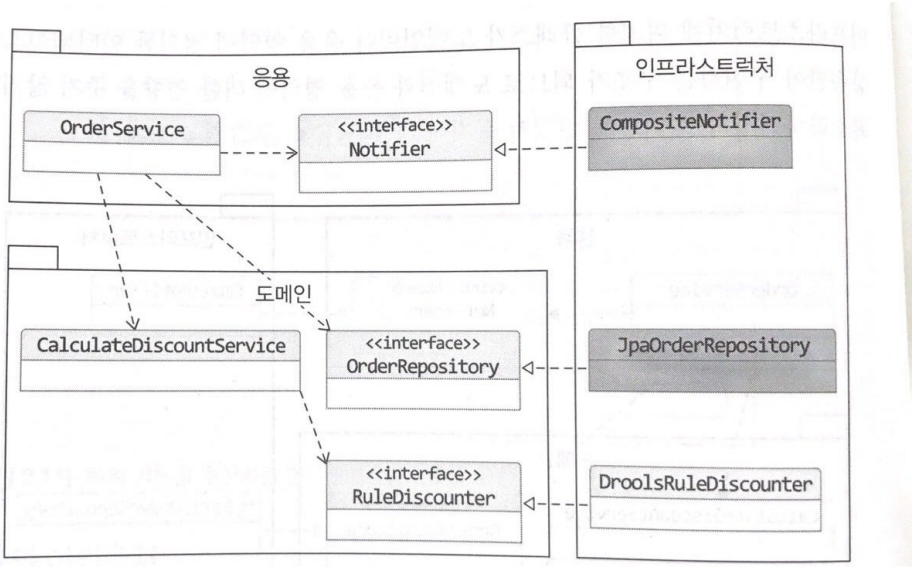
인터페이스를 분리하여 고수준 모듈이 저수준 모듈에 의존하지 않도록 변경한다.

DIP와 아키텍처

인프라스트럭처에 위치한 클래스가 도메인이나 응용 영역에 정의한 인터페이스를 상속받아 구현하는 구조가 되므로 도메인과 응용 영역에 대한 영향을 주지 않거나 최소화하면서 구현 기술을 변경하는 것이 가능하다.



[그림 2.13] DIP를 적용한 구조



[그림 2.14] DIP를 적용하면 응용 영역과 도메인 영역에 영향을 최소화하면서 구현체를 변경하거나 추가할 수 있다.

도메인 영역의 주요 구성요소

요소	설명
엔터티	고유의 식별자를 갖는 객체로 도메인 모델의 데이터를 포함하며 해당 데이터와 관련된 기능을 함께 제공한다.
밸류	고유의 식별자를 갖지 않는 객체로 개념적으로 하나인 도메인 객체의 속성을 표현할 때 사용된다. 엔터티의 속성으로 사용될 뿐만 아니라 다른 밸류 타입의 속성으로도 사용될 수 있다.
애그리거트	관련된 엔터티와 밸류 객체를 개념적으로 하나로 묶은 것이다. 주문과 관련된 Order, OrderLine, Orderer를 '주문' 애그리거트로 묶을 수 있다.
리포지터리	도메인 모델의 영속성을 처리한다. 예를 들어, DBMS 테이블에서 엔터티 객체를 로딩하거나 저장하는 기능을 제공한다.
도메인 서비스	특정 엔터티에 속하지 않은 도메인 로직을 제공한다. 도메인 로직이 여러 엔터티와 밸류를 필요로 할 경우 도메인 서비스에서 로직을 구현한다.

엔터티와 밸류

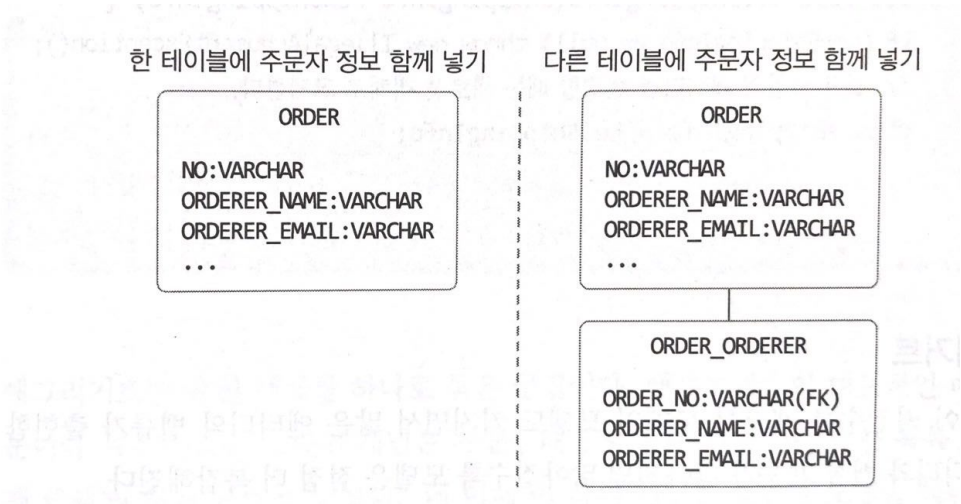
DB 관계형 모델의 엔터티와 비교 했을때 도메인 모델의 엔터티의 가장 큰 차이점은 데이터와 함께 도메인 기능을 함께 제공한다는 점이다. 도메인 관점에서 기능을 구현하고 기능 구현을 캡슐화해서 데이터가 임의로 변경되는 것을 막는다.

또다른 차이점은 도메인 모델의 엔터티는 두 개 이상의 데이터가 개념적으로 하나인 경우 밸류 타입을 이용해서 표현할 수 있다는 것이다.

```
public class Orderer {
    private String name;
    private String email;

    ...
}
```

```
public class Order {
    // 주문 도메인 모델의 데이터
    private OrderNo number;
    private Orderer orderer;
    private ShippingInfo shippingInfo;

    // 도메인 기능
    public void changeShippingInfo(ShippingInfo newShippingInfo) {

    }
}
```

RDBMS에서의 표현



[그림 2.15] RDBMS는 밸류를 제대로 표현하기 힘들다.

애그리거트

도메인이 커질 수록 엔터티와 밸류가 점점 많아진다. 갯수가 많아지면 점점 복잡도가 커진다. 도메인 모델이 복잡해지면 개발자가 전체 구조가 아닌 한 개의 엔터티와 밸류에만 집중하게 된다. 이를 해결하기 위해 애그리거트 라는 개념이 등장.

> 관련 객체를 하나로 묶은 군집

예를 들어 주문이라는 도메인 개념은 주문(루트 엔터티), 배송지정보, 주문자, 주문목록, 총결제금액의 하위 모델로 구성되는데 이 때 이 하위 개념을 표현한 모델을 하나로 묶어서 '주문' 이라는 상위 개념으로 표현할 수 있다.

루트 엔터티

- 애그리거트는 군집에 속한 객체를 관리하는 루트 엔터티를 갖는다.
- 루트 엔터티는 애그리거트에 속해 있는 엔터티와 밸류를 이용해서 애그리거트가 구현해야 할 기능을 제공.

리포지터리

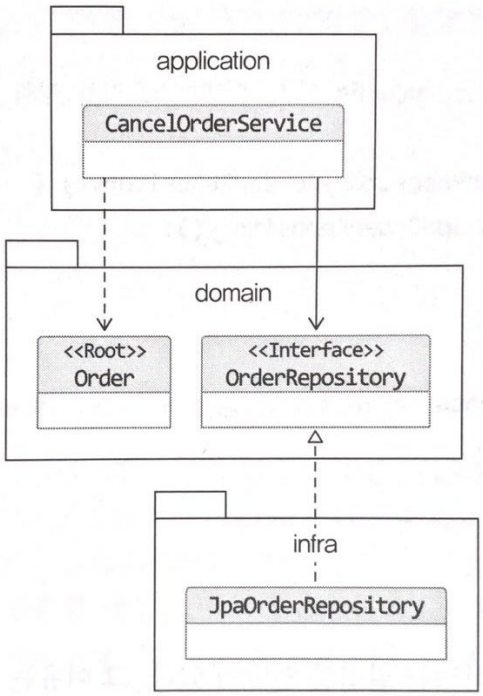
리포지터리는 RDBMS와 같은 물리적인 저장소에 애그리거트 단위로 도메인 객체를 저장하고 조회하는 기능을 정의한다. 예를 들어, 주문 애그리거트를 위한 리포지터리는 다음과 같이 정의할 수 있다.

```
public interface OrderRepository {
    public Order findByNumber(OrderNumber number);
    public void save(Order order);
    public void delete(Order order)
}
```

OrderRepository의 메서드를 보면 대상을 찾고 저장하는 단위가 애그리거트 루트인 Order인 것을 알 수 있다. 응용 서비스는 다음 코드처럼 OrderRepository를 이용해서 Order 객체를 구하고 해당기능을 실행한다.

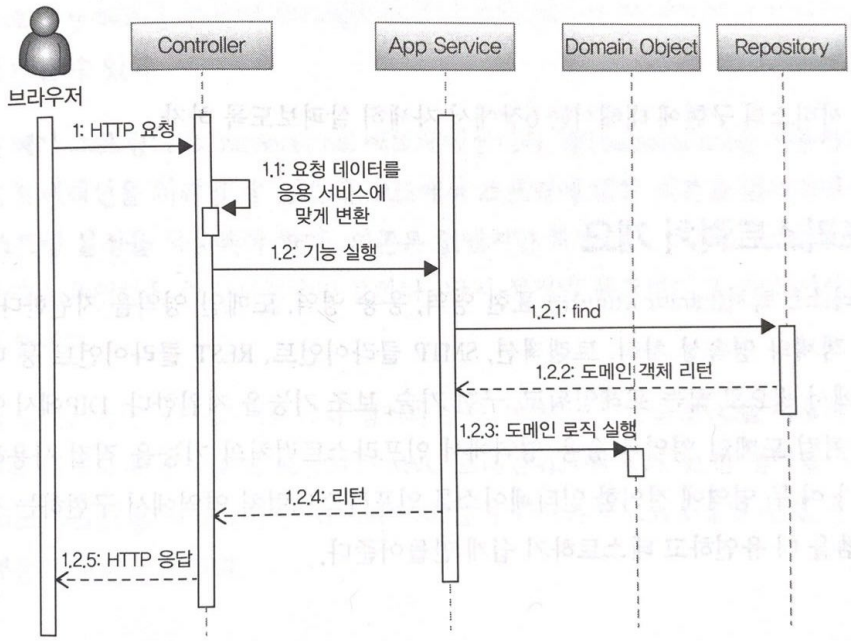
```
public class CancelOrderService {
    private OrderRepository orderRepository;

    public void cancel(OrderNumber number) {
        Order order = orderRepository.findByNumber(number);
        if(order == null) throw new NoOrderException(number);
        order.cancel();
    }
}
```

전체 모듈 구조는 아래와 같다.



[그림 2.19] 리포지터리 인터페이스는 도메인 모델 영역에 속하며, 실제 구현 클래스는 인프라스트럭처 영역에 속한다.

요청 처리 흐름



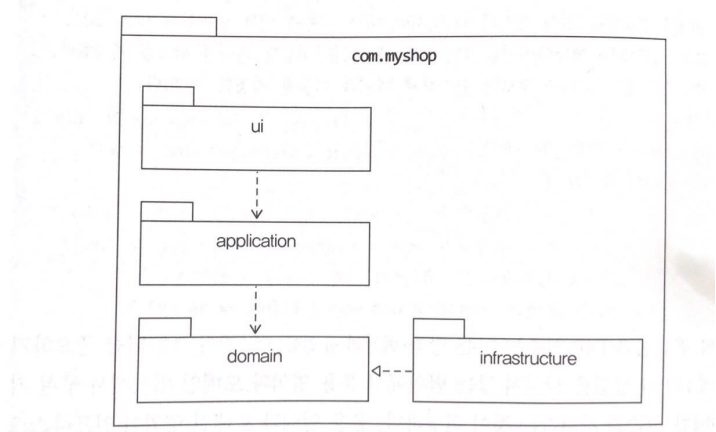
[그림 2.20] 요청 처리 흐름

인프라스트럭처 개요

DIP에서 언급한 것처럼 도메인 영역과 응용 영역에서 정의한 인터페이스를 인프라스트럭처 영역에서 구현하는 것이 시스템을 더 유연하고 테스트하기 쉽게 만들어준다. 하지만, 항상 인프라스트럭처에 대한 의존을 없애는 것이 좋은 것은 아니다. 구현의 편리함은 DIP가 주는 다른 장점(변경의 유연함, 테스트가 쉬움)만큼 중요하기 때문에 DIP의 장점을 해치지 않는 범위에서 응용 영역과 도메인 영역에서 구현 기술에 대한 의존을 가져가는 것이 현명하다. 그 예로 스프링의 @Transactional 애노테이션과 JPA의 @Entity, @Table 애노테이션이 있다.

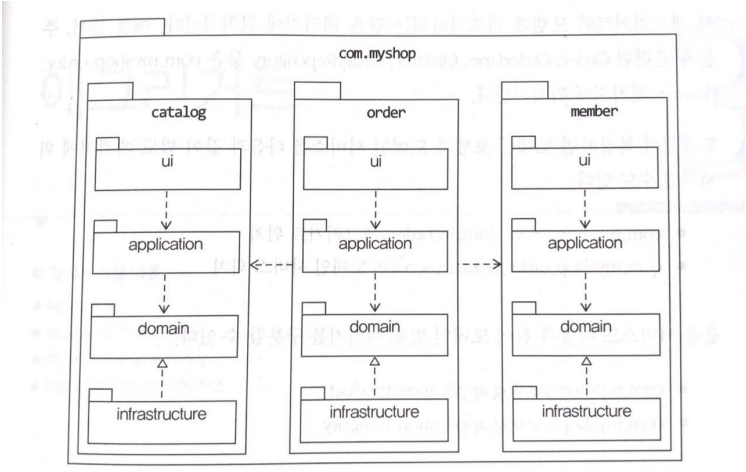
모듈 구성

아키텍처의 각 영역은 별도 패키지에 위치한다. [그림 2.21]과 같이 영역별로 모듈이 위치할 패키지를 구성 할 수 있을 것이다.



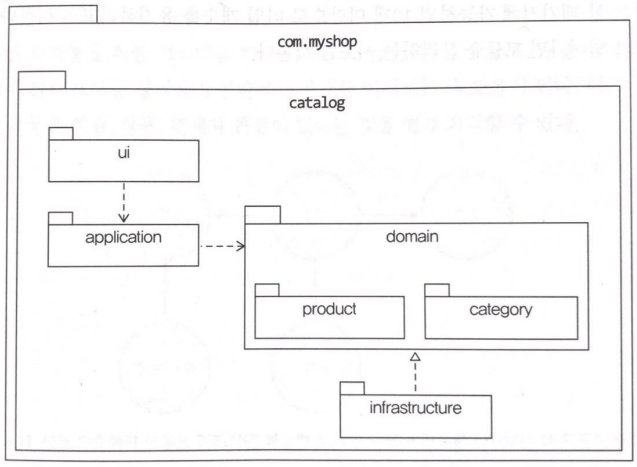
[그림 2.21] 영역별로 별도 패키지로 구성한 모듈 구조

도메인이 크면 [그림 2.22]와 같이 하위 도메인으로 나누고 각 하위 도메인마다 별도 패키지를 구성한다.



[그림 2.22] 도메인 크면 하위 도메인별로 모듈을 나눈다.

domain 모듈은 도메인에 속한 애그리거트를 기준으로 다시 패키지를 구성한다.



[그림 2.23] 하위 도메인을 하위 패키지로 구성한 모듈 구조

도메인이 복잡하면 도메인 모델과 도메인 서비스를 다음과 같이 별도 패키지에 위치시킬 수도 있다.

- com.myshop.order.domain.order: 애그리거트 위치
- com.myshop.order.domain.service: 도메인 서비스 위치

응용 서비스도 다음과 같이 도메인 별로 패키지를 구분할 수 있다.

- com.myshop.catalog.application.product
- com.myshop.catalog.application.category