

Refactoring - Cafe

Legacy Code Refactoring

목차

1. 목표
2. 실습 소개 - Cafe Service
3. Code Smell
4. Legacy Code 리팩토링

- ✓ Step 1. Extract Method
- ✓ Step 2. Feature Envy
- ✓ Step 3. Nested Condition
- ✓ Step 4. Polymorphism
- Step 5. Magic Number
- ✓ Step 6. Factory Pattern
- Step 7. Builder Pattern

- ✓ Step 8. DTO 적용 – Entity + DTO
- ✓ Step 9. DTO 적용 – Entity와 DTO 분리
- ✓ Step 10. 모듈 순환 참조

1. 목표

- 유지보수가 어려운 Legacy Code 내 코드 스멜을 찾는다.
- Legacy Code 리팩토링을 수행하면서 Test Code가 필요한 이유를 체감한다.
- Code Smell 별 리팩토링 접근 방법을 실습한다.
- Refactoring 목표와 방법을 이해한다.

2. 실습 소개 – Café Service

- Café Service

- 사내 카페에서 음료를 주문하여 결제하는 서비스
- 대상 : 새로운 주문 생성 서비스 (OrderService.create 메소드)
- Business 로직
 - 고객 Id로 사용자 정보 조회
 - 주문된 음료의 Total Cost를 계산하여 주문 대기 상태로 주문 내역 저장
 - 고객이 주문한 음료의 주문 목록 저장
 - 매월 마지막날 10% 할인 프로모션
 - 고객은 결제를 현금, 카드, 마일리지로 진행가능



2. 실습 소개 – Café Service

- 사내 Github repository
 - 사내 Git 주소 : <https://code.sdsdev.co.kr/act-edu/cafe>
- Branch 소개
 - **rf-initial** : 실습 시작 브랜치
 - **rf-step1 ~ rf-step10** : 실습 step별로 완성된 코드 브랜치
 - **rf-final** : 리팩토링 수행한 최종 브랜치

```
$ git clone https://code.sdsdev.co.kr/act-edu/cafe.git  
$ git checkout rf-initial  
$ git pull origin rf-initial
```

2. 실습 소개 – Café Service

- OrderService.create()

@Transactional

```
public Order create(int customerId, List<Map<String, Object>> orderItemList, int payment) {
    Customer customer = customerService.getCustomer(customerId);
    Order order = new Order();
    order.setStatus(OrderStatus.WAITING);
    order.setCustomer(customer);
    order.setPayment(payment);
    List<OrderItem> orderItems = new ArrayList<>();
    double totalCost = 0;

    // 1. 주문된 항목의 Total Cost 계산
    for(Map<String, Object> orderItemMap: orderItemList) {
        Beverage beverage = beverageRepository.findOne((Integer) orderItemMap.get("beverageId"));
        if(beverage == null) {
            continue;
        }

        OrderItem orderItem = new OrderItem();
        orderItem.setCount((Integer) orderItemMap.get("count"));
        orderItem.setBeverage(beverage);
        orderItem.setOrder(order);
        orderItems.add(orderItem);
        totalCost += orderItem.getCount() * orderItem.getBeverage().getCost();
    }

    // 2. 매월 마지막 날이면 10% 할인
    totalCost = this.getDiscountedTotalCost(totalCost);
    order.setTotalCost(totalCost);

    double mileagePoint = 0;
    switch(payment) {
        case 1: // cash 10%
            // 2021.1.1 현금 적립률 8% -> 10%로 변경
            // mileagePoint = order.getTotalCost() * 0.05;
            mileagePoint = totalCost * 0.1;
            break;
    }
}
```

```
        case 2: // credit card 5%
            mileagePoint = totalCost * 0.05;
            break;
        case 3: // mileage
            break;
    }

    if(payment == 3) { // pay mileage
        int customerMileage = mileageApiService.getMileages(customerId);
        if(customerMileage >= order.getTotalCost()) {
            Mileage mileage = new Mileage(customerId, order.getId(), order.getTotalCost());
            mileageApiService.minusMileages(customerId, mileage);
        } else {
            throw new BizException("mileage is not enough");
        }
    } else {
        Mileage mileage = new Mileage(customerId, order.getId(), mileagePoint);
        mileageApiService.saveMileages(customerId, mileage);

        if(payment == 1) {
            payWithCash(order, customerId);
        } else if(payment == 2) {
            payWithCard(order, customerId);
        }
    }

    order.setMileagePoint(mileagePoint);
    orderRepository.save(order);
    orderItemRepository.saveAll(orderItems);

    return order;
}
```

3. Café 서비스 Code Smell 찾기

- **Comments**

- 주석이 유지보수 대상이 될 수 있다.
- 주석이 필요하다는 것은 리팩토링이 필요하다는 의미이다.

- **Long Method**

- 비즈니스 로직이 복잡하다. -> order create 메소드가 너무 많은 역할을 수행하고 있다.
- 결제 유형에 따라 결제 로직처리가 이곳에 있어야 되는게 맞는가?

- **Feature Envy**

- 서로 분리되어야 할 로직이 한 곳에 있다.

- **Nested Condition**

- 로직을 복잡하게 하여 가독성을 떨어뜨린다.

- **Magic Number**

- paymentType이 어떤 의미인지 알 수 없다.

Legacy Code 1

```
/**
 * 주문 생성
 * 1. orderItem 생성 및 주문 총 비용 계산
 * 2. 할인 프로모션 : 매월 마지막 날이면 10% 할인
 * 3. 결제 유형에 따라 마일리지 적립
 * 4. order, orderItem DB 저장
 * @param customerId
 * @param orderItemList
 * @return
 */
@Transactional
public Order create(int customerId, List<Map<String, Object>> orderItemList, int payment) throws Exception {
    Customer customer = customerService.getCustomer(customerId);
    Order order = new Order();
    order.setStatus(OrderStatus.WAITING);
    order.setCustomer(customer);
    order.setPayment(payment);
    List<OrderItem> orderItems = new ArrayList<>();
    double totalCost = 0;

    // 1. 주문된 항목의 Total Cost 계산
    for(Map<String, Object> orderItemMap : orderItemList) {
        Beverage beverage = beverageRepository.findOne((Integer) orderItemMap.get("beverageId"));
        if(beverage == null) {
            continue;
        }

        OrderItem orderItem = new OrderItem();
        orderItem.setCount((Integer) orderItemMap.get("count"));
        orderItem.setBeverage(beverage);
        orderItem.setOrder(order);
        orderItems.add(orderItem);
        totalCost += orderItem.getCount() * beverage.getCost();
    }

    order.setTotalCost(totalCost);
}
```

여기 좀 구려요.

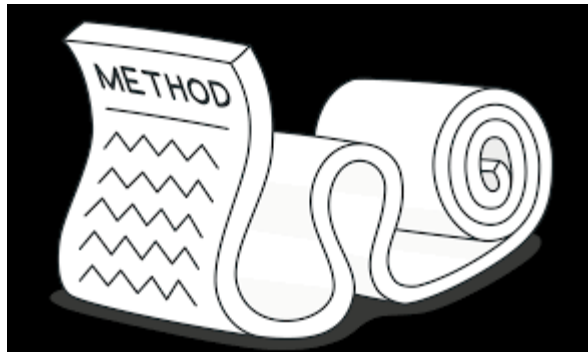
여기도 구려요

“리팩토링을 시작해봅시다.”

4. Legacy Code 리팩토링 – Long Method

Q) 왜 Long Method가 만들어지는 것일까?

- 업무가 복잡해서 **메소드가 너무 많은 기능을 수행**한다.
- 절차지향적으로 코드를 작성하는 경우
- 분기문의 남용
 - 새로운 요구사항 개발 시 기존 코드를 **copy & paste**로 새로운 **block**을 생성해서 개발 진행한다.
 - 유사하지만 조금씩 다른 코드가 만들어지고, 나중에 변경이 더 어려워지게 된다.



Step 1. Extract Method

메소드 추출 기법을 활용하여 Long Method를 제거하고 코드 가독성을 높여보자.

- 대상 : 주문 item 목록 생성 및 주문 금액(Total Cost) 계산
- Code smell
 - 코드 가독성이 떨어진다.
 - 소스 분석 없이는 무슨 일을 하는지 파악이 어렵다.
- Refactoring
 - 목표 : 메소드 추출로 코드 가독성을 높인다.
 - **for문 안에서 orderItem 목록 생성과 totalCost를 계산하는 부분을 분리**

Step 1. Extract Method

리팩토링을 통해 코드 가독성을 높이는 것이 성능 개선을 의미하는 것은 아니다.

- getOrderItems 메소드 추출
 - orderItemMap 목록을 가지고 orderItem 목록 생성

CTRL + ALT + M
(메소드 추출)

```
// 4. 주문된 항목의 Total Cost 계산
List<OrderItem> orderItems = new ArrayList<>();
for (Map<String, Object> orderItemMap: orderItemList) {
    Beverage beverage = beverageRepository.findOne((Integer)
orderItemMap.get("beverageId"));
    if (beverage == null) {
        continue;
    }

    OrderItem orderItem = new OrderItem();
    orderItem.setCount((Integer) orderItemMap.get("count"));
    orderItem.setBeverage(beverage);
    orderItem.setOrder(order);
    orderItems.add(orderItem);
}

double totalCost = 0;
for (OrderItem orderItem : orderItems) {
    totalCost += orderItem.getCount() * orderItem.getBeverage().getCost();
}
```

```
List<OrderItem> orderItems = getOrderItems(orderItemList, order);
```

```
double totalCost = 0;
for (OrderItem orderItem : orderItems) {
    totalCost += orderItem.getCount() * orderItem.getBeverage().getCost();
}
```

```
private List<OrderItem> getOrderItems(List<Map<String, Object>> orderItemList, Order order) {
    List<OrderItem> orderItems = new ArrayList<>();
    for (Map<String, Object> orderItemMap: orderItemList) {
        Beverage beverage = beverageRepository.findOne((Integer) orderItemMap.get("beverageId"));
        if (beverage == null) {
            continue;
        }

        OrderItem orderItem = new OrderItem();
        orderItem.setCount((Integer) orderItemMap.get("count"));
        orderItem.setBeverage(beverage);
        orderItem.setOrder(order);
        orderItems.add(orderItem);
    }
    return orderItems;
}
```

Step 1. Extract Method

리팩토링을 통해 코드 가독성을 높이는 것이 성능 개선을 의미하는 것은 아니다.

- getTotalCost 메소드 추출
 - **orderItem** 목록을 사용하여 총 금액을 계산

```
List<OrderItem> orderItems = getOrderItems(orderItemList, order);  
  
double totalCost = 0;  
for (OrderItem orderItem : orderItems) {  
    totalCost += orderItem.getCount() * orderItem.getBeverage().getCost();  
}  
  
totalCost = this.getDiscountedTotalCost(totalCost);  
order.setTotalCost(totalCost);  
  
double mileagePoint = 0;
```

CTRL + ALT + M
(메소드 추출)

```
List<OrderItem> orderItems = getOrderItems(orderItemList, order);  
  
double totalCost = this.getDiscountedTotalCost(getTotalCost(orderItems));  
order.setTotalCost(totalCost);  
  
double mileagePoint = 0;
```

```
private double getTotalCost(List<OrderItem> orderItems) {  
    double totalCost = 0;  
    for (OrderItem orderItem : orderItems) {  
        totalCost += orderItem.getCount() * orderItem.getBeverage().getCost();  
    }  
    return totalCost;  
}
```

Step 1. Extract Method

직관적 메소드명을 사용하면 코드만으로 어떤 기능인지 쉽게 파악할 수 있다.
리팩토링이 끝나면 반드시 이전의 테스트가 통과하는지 꼭 확인한다.

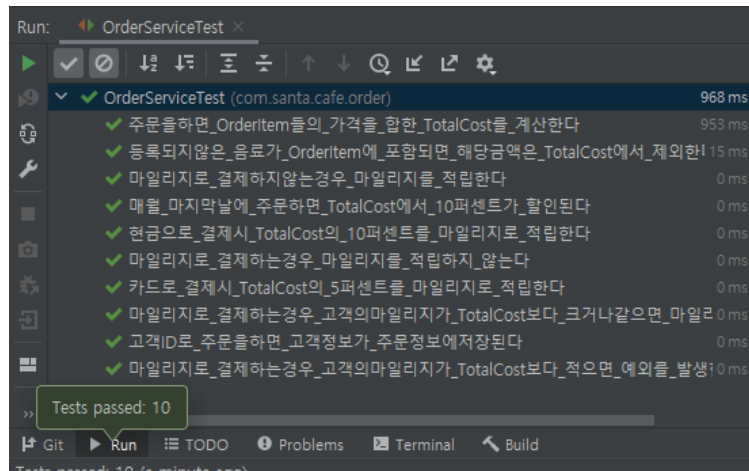
```
@Transactional
public Order create(int customerId, List<Map<String, Object>> orderItemList, int payment) {
    Customer customer = customerService.getCustomer(customerId);
    Order order = new Order();
    order.setStatus(OrderStatus.WAITING);
    order.setCustomer(customer);
    order.setPayment(payment);

    List<OrderItem> orderItems = getOrderItems(orderItemList, order);

    double totalCost = this.getDiscountedTotalCost(getTotalCost(orderItems));
    order.setTotalCost(totalCost);

    double mileagePoint = 0;
    switch(payment) {
        case 1:
            mileagePoint = totalCost * 0.1;
            break;
        case 2:
            mileagePoint = totalCost * 0.05;
            break;
        case 3:
            break;
    }
}
```

작업 후



Step 1. Extract Method

메소드 추출 기법을 활용하여 코드의 가독성을 높여보자.

- 대상 : 마일리지 계산 및 결제하는 로직
- Code smell
 - 기능을 수행하는지 직관적으로 파악하기 어렵다.
- Refactoring
 - 목표 : 메소드 추출로 코드 가독성을 높인다.
 - 마일리지 계산 메소드 추출
 - `getMileagePoint (int payment, double totalCost)`
 - 결제 유형에 따라 결제 로직 메소드 추출
 - `pay (int customerId, int payment, Order order, double mileagePoint)`

Step 1. Extract Method

getMileagePoint 메소드 추출 : 결제 유형에 따른 마일리지 계산 메소드 추출

CTRL + ALT + M
(메소드 추출)

```
double totalCost = this.getDiscountedTotalCost(getTotalCost(orderItems));
order.setTotalCost(totalCost);

double mileagePoint = 0;
switch(payment) {
    case 1:
        mileagePoint = totalCost * 0.1;
        break;
    case 2:
        mileagePoint = totalCost * 0.05;
        break;
    case 3:
        break;
}

if(payment == 3) {
    int customerMileage = mileageApiService.getMileages(customerId);
    if(customerMileage >= order.getTotalCost()) {
```

```
double totalCost = this.getDiscountedTotalCost(getTotalCost(orderItems));
order.setTotalCost(totalCost);
```

```
double mileagePoint = getMileagePoint(payment, totalCost);
```

```
if(payment == 3) {
    int customerMileage = mileageApiService.getMileages(customerId);
    if(customerMileage >= order.getTotalCost()) {
```

```
private double getMileagePoint(int payment, double totalCost) {
    double mileagePoint = 0;
    switch(payment) {
        case 1:
            mileagePoint = totalCost * 0.1;
            break;
        case 2:
            mileagePoint = totalCost * 0.05;
            break;
        case 3:
            break;
    }
    return mileagePoint;
}
```

Step 1. Extract Method

pay 메소드 추출 : 결제 유형에 따라 결제 및 마일리지 api 연계 메소드 추출

CTRL + ALT + M
(메소드 추출)

```
double mileagePoint = getMileagePoint(payment, totalCost);

if(payment == 3) {
    int customerMileage = mileageApiService.getMileages(customerId);
    if(customerMileage >= order.getTotalCost()) {
        Mileage mileage = new Mileage(customerId, order.getId(), order.getTotalCost());
        mileageApiService.minusMileages(customerId, mileage);
    } else {
        throw new BizException("mileage is not enough");
    }
} else {
    Mileage mileage = new Mileage(customerId, order.getId(), mileagePoint);
    mileageApiService.saveMileages(customerId, mileage);

    if(payment == 1) {
        payWithCash(order, customerId);
    } else if(payment == 2) {
        payWithCard(order, customerId);
    }
}

order.setMileagePoint(mileagePoint);
orderRepository.save(order);
orderItemRepository.saveAll(orderItems);
```

```
double mileagePoint = getMileagePoint(payment, totalCost);
```

```
pay(customerId, payment, order, mileagePoint);
```

```
order.setMileagePoint(mileagePoint);
orderRepository.save(order);
orderItemRepository.saveAll(orderItems);
```

```
private void pay(int customerId, int payment, Order order, double mileagePoint) {
    if(payment == 3) {
        int customerMileage = mileageApiService.getMileages(customerId);
        if(customerMileage >= order.getTotalCost()) {
            Mileage mileage = new Mileage(customerId, order.getId(), order.getTotalCost());
            mileageApiService.minusMileages(customerId, mileage);
        } else {
            throw new BizException("mileage is not enough");
        }
    } else {
        Mileage mileage = new Mileage(customerId, order.getId(), mileagePoint);
        mileageApiService.saveMileages(customerId, mileage);

        if(payment == 1) {
            payWithCash(order, customerId);
        } else if(payment == 2) {
            payWithCard(order, customerId);
        }
    }
}
```


Step 1. Extract Method

메소드 추출을 통해 기존 코드 대비 가독성이 월등히 향상되었다.

```
@Transactional
public Order create(int customerId, List<Map<String, Object>> orderItemList, int payment) {
    Customer customer = customerService.getCustomer(customerId);
    Order order = new Order();
    order.setStatus(OrderStatus.WAITING);
    order.setCustomer(customer);
    order.setPayment(payment);

    List<OrderItem> orderItems = getOrderItems(orderItemList, order);

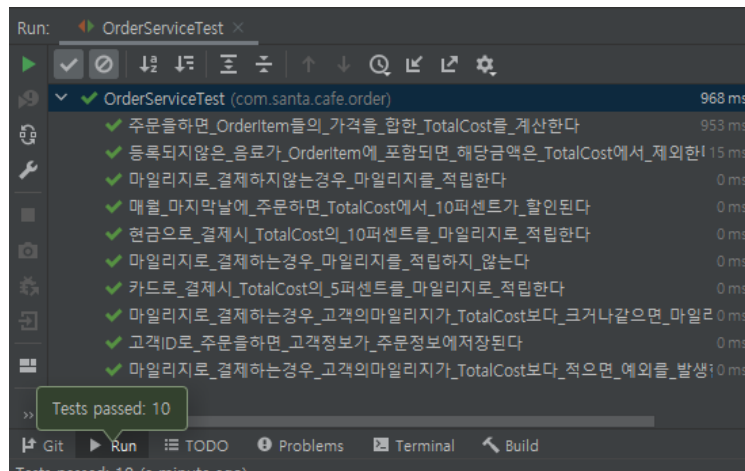
    double totalCost = this.getDiscountedTotalCost(orderItems);
    order.setTotalCost(totalCost);

    double mileagePoint = getMileagePoint(payment, totalCost);
    pay(customerId, payment, order, mileagePoint);

    order.setMileagePoint(mileagePoint);
    orderRepository.save(order);
    orderItemRepository.saveAll(orderItems);

    return order;
}
```

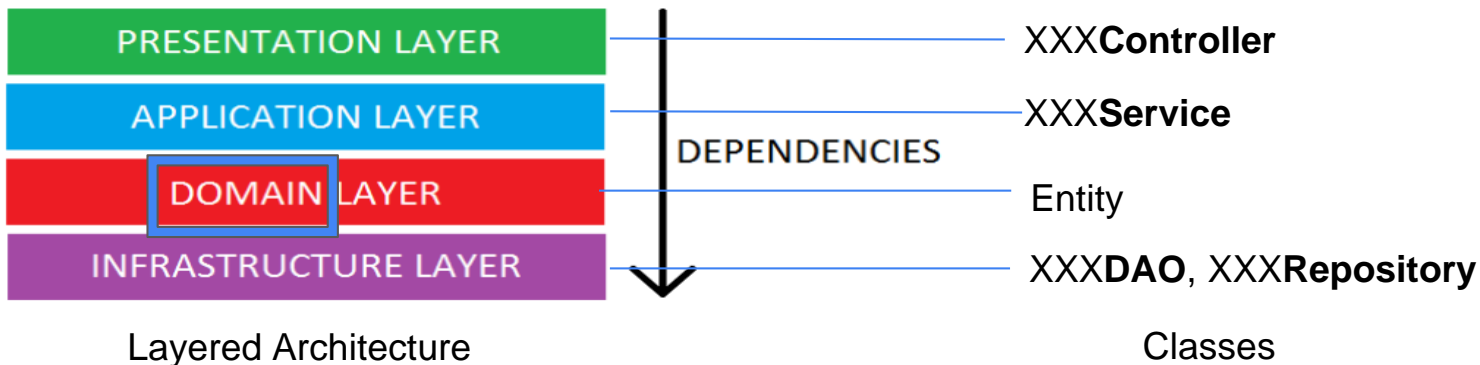
작업 후



Step 2. 클래스 분리 - Feature Envy

메소드가 자신이 속한 클래스보다 **다른 클래스에 관심을 가지고 있는 경우**를 말한다.

- 기본적인 규칙은 **같이 변화하는 것을 모으는 것**이다.
- 가장 흔한 Envy는 데이터에 대한 욕심이다.
- 메소드의 특정 부분만 고통받는 경우
 - 특정 부분을 **Extract method** 후, **Move method** 수행한다.



Step 2. 클래스 분리 - Feature Envy

클래스 추출 기법을 사용하여, 기존 도메인과 다른 역할을 수행하는 새로운 클래스를 생성하자.

- **대상 : 결제 유형에 따라 마일리지 포인트 계산, 결제 함수**
- Code smell
 - 결제 기능의 경우, 결제 기능을 담당하는 **PaymentService**로 기능이 옮기는 것이 더 낫다.
 - **OrderService** 객체는 현재 과도하게 주문 뿐만 아니라 결제 관련된 기능까지 수행하고 있다.
- Refactoring
 - **단일 책임 원칙**에 따라 객체는 각자 하나의 책임만 가지며, 각각의 클래스는 그 책임을 완전히 캡슐화 할 수 있다.
 - **Extract Class** 기법을 활용하여 **OrderService.create()**에서 결제 비즈니스 로직을 분리하여 **PaymentService**를 생성한다.
 - 마일리지 포인트 계산, 결제 함수를 **PaymentService**로 **Move method** 진행

Step 2. 클래스 분리 - Feature Envy

주문 관련 도메인에서 마일리지 계산 및 결제 로직까지 다루어야 할까?

```
@Transactional
public Order create(int customerId, List<Map<String, Object>> orderItemList, int payment) {
    Customer customer = customerService.getCustomer(customerId);
    Order order = new Order();
    order.setStatus(OrderStatus.WAITING);
    order.setCustomer(customer);
    order.setPayment(payment);

    List<OrderItem> orderItems = getOrderItems(orderItemList, order);

    double totalCost = this.getDiscountedTotalCost(orderItems);
    order.setTotalCost(totalCost);

    double mileagePoint = getMileagePoint(payment, totalCost);
    pay(customerId, payment, order, mileagePoint);

    order.setMileagePoint(mileagePoint);
    orderRepository.save(order);
    orderItemRepository.saveAll(orderItems);

    return order;
}
```

작업 전

- 마일리지 포인트 계산
- 결제
- 마일리지 API 연계



1. Extract Class
 - PaymentService 도메인 서비스를 생성
2. Move Method
 - 관련 메소드를 PaymentService로 이동

Step 2. 클래스 분리 - Feature Envy

PaymentService 클래스 생성 후, Move method 진행

1. 새로운 클래스 생성 : PaymentService.java
2. OrderService에서 PaymentService 빈을 의존성 주입 받도록 수정
3. payWithCard, payWithCash 메소드 이동
4. getMileagePoint, pay 메소드 이동
5. PaymentService에서 MileageApiService 빈을 주입 받도록 코드 추가
6. 에러 발생하는 테스트 코드를 수정한다.

Step 2. 클래스 분리 - Feature Envy

- PaymentService에서 외부에서 사용하는 메소드 접근제어자 변경 (private -> public)
- OrderService에서 PaymentService의 메소드를 호출하도록 변경

OrderService.create()

```
@Transactional
public Order create(int customerId, List<Map<String, Object>> orderItemList, int payment) {
    Customer customer = customerService.getCustomer(customerId);
    Order order = new Order();
    order.setStatus(OrderStatus.WAITING);
    order.setCustomer(customer);
    order.setPayment(payment);

    List<OrderItem> orderItems = getOrderItems(orderItemList, order);

    double totalCost = this.getDiscountedTotalCost(getTotalCost(orderItems));
    order.setTotalCost(totalCost);

    double mileagePoint = paymentService.getMileagePoint(payment, totalCost);

    paymentService.pay(customerId, payment, order, mileagePoint);

    order.setMileagePoint(mileagePoint);
    orderRepository.save(order);
    orderItemRepository.saveAll(orderItems);

    return order;
}
```

PaymentService

```
public class PaymentService {
    private final MileageApiService mileageApiService;

    public PaymentService(MileageApiService mileageApiService) {
        this.mileageApiService = mileageApiService;
    }

    public double getMileagePoint(int payment, double totalCost) {
        ...
    }

    public void pay(int customerId, int payment, Order order, double mileagePoint) {
        ...
    }

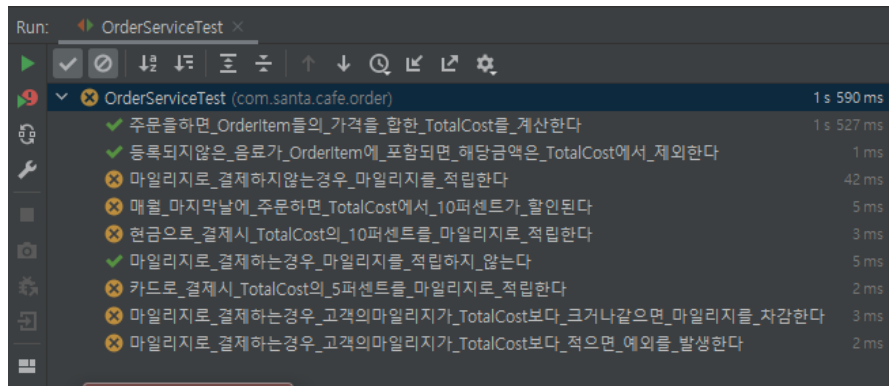
    private void payWithCard(Order order, int customerId) {
    }

    private void payWithCash(Order order, int customerId) {
    }
}
```

Step 2. 클래스 분리 - Feature Envy

Move method를 통해서 비즈니스 로직을 옮겼기 때문에 검증하는 테스트 부분도 함께 이동해야 한다.

- OrderServiceTest : 코드가 수정되면서 테스트 실패 발생. 해당 테스트 코드를 수정한다.
- PaymentServiceTest : OrderServiceTest에서 마일리지 및 결제 관련 테스트 케이스를 이동. 테스트 동작하도록 테스트 코드를 수정한다.



OrderServiceTest

1. PaymentService 빈 주입
2. 주문 관련 테스트 실패 건은 테스트가 통과하도록 테스트 코드 수정

PaymentServiceTest

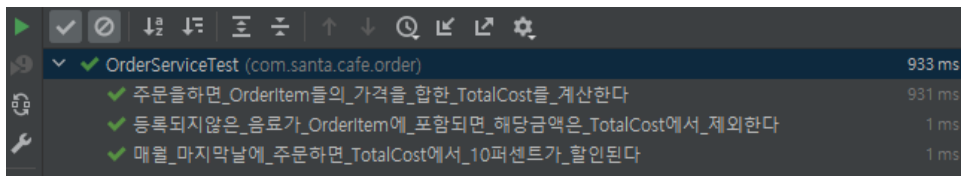
1. 마일리지 적립 및 결제 관련 테스트 코드 이동
2. 테스트가 통과하도록 테스트 코드 수정
3. 필요한 경우 테스트 케이스 추가

Step 2. 클래스 분리 - Feature Envy

PaymentService가 통과할 수 있도록 테스트 케이스를 수정해야 한다.

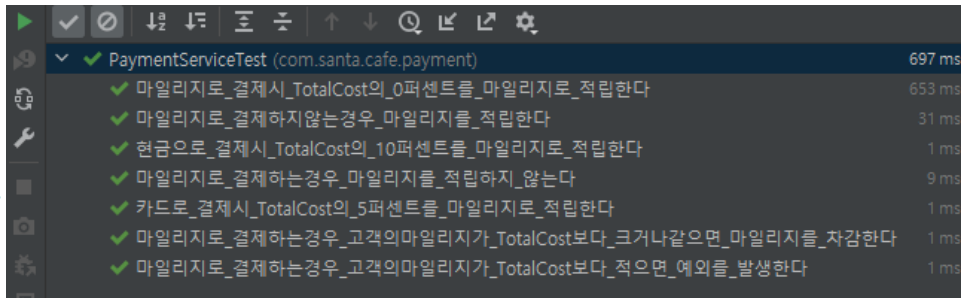
유지보수하기 쉬운 테스트를 작성해야 한다.

- 테스트 코드는 작은 기능 단위를 커버해야 한다.
- 테스트 코드 또한 리팩토링의 대상이다.
(예: 중복 제거, 메소드 추출, 상수로 분리)
- **Github 소스 확인 :**
<https://code.sdsdev.co.kr/act-edu/cafe/blob/rf-step2/src/test/java/com/santa/cafe/payment/PaymentServiceTest.java>



Test results for OrderServiceTest (com.santa.cafe.order):

Test Case	Duration
주문하면_Orderitem들의_가격을_합한_TotalCost를_계산한다	933 ms
등록되지않은_음료가_Orderitem에_포함되면_해당금액은_TotalCost에서_제외한다	931 ms
매월_마지막날에_주문하면_TotalCost에서_10퍼센트가_할인된다	1 ms



Test results for PaymentServiceTest (com.santa.cafe.payment):

Test Case	Duration
마일리지로_결제시_TotalCost의_0퍼센트를_마일리지로_적립한다	697 ms
마일리지로_결제하지않는경우_마일리지를_적립한다	653 ms
현금으로_결제시_TotalCost의_10퍼센트를_마일리지로_적립한다	31 ms
마일리지로_결제하는경우_마일리지를_적립하지_않는다	1 ms
카드로_결제시_TotalCost의_5퍼센트를_마일리지로_적립한다	9 ms
마일리지로_결제하는경우_고객의마일리지가_TotalCost보다_크거나같으면_마일리지를_차감한다	1 ms
마일리지로_결제하는경우_고객의마일리지가_TotalCost보다_적으면_예외를_발생한다	1 ms

Step 3. Nested Condition

Nested Condition으로 구성된 로직으로 코드 가독성이 떨어진다.
중첩된 구문을 최대한 제거해보자.

작업 전

```
public void pay(int customerId, int payment, Order order, double mileagePoint) {  
    if(payment == 3) {  
        int customerMileage = mileageApiService.getMileages(customerId);  
        if(customerMileage >= order.getTotalCost()) {  
            Mileage mileage = new Mileage(customerId, order.getId(), order.getTotalCost());  
            mileageApiService.minusMileages(customerId, mileage);  
        } else {  
            throw new BizException("mileage is not enough");  
        }  
    } else {  
        Mileage mileage = new Mileage(customerId, order.getId(), mileagePoint);  
        mileageApiService.saveMileages(customerId, mileage);  
  
        if(payment == 1) {  
            payWithCash(order, customerId);  
        } else if(payment == 2) {  
            payWithCard(order, customerId);  
        }  
    }  
}
```

Step 3. Nested Condition

IDE 기능을 활용하여 중첩된 if문을 밖으로 빼낸다.

1. 해당 소스를 블록지정하여 임시로 잘라내기(Ctrl + X)를 한다.

```
} else {  
    Mileage mileage = new Mileage(customerId, order.getId(), mileagePoint);  
    mileageApiService.saveMileages(customerId, mileage);  
  
    if(payment == 1) {  
        payWithCash(order, customerId);  
    } else if(payment == 2) {  
        payWithCard(order, customerId);  
    }  
}
```

2. ALT + Enter

```
} else {  
    ⚙ Invert 'if' condition  
    ⚙ Remove braces from 'else' statement  
    ⚙ Unwrap 'else' branch (changes semantics)  
    ⚙ Merge 'else if'  
    ⚙ Press Ctrl+Shift+I to open preview  
    Mileage mileage = new Mileage(customerId, order.getId(), mileagePoint);  
    mileageApiService.saveMileages(customerId, mileage);  
    payWithCard(order, customerId);  
}
```

3. ALT + Enter

```
if(payment == 3) {  
    int customerMileage = mileageApiService.getMileages(customerId);  
    if(customerMileage >= order.getTotalCost()) {  
        Mileage mileage = new Mileage(customerId, order.getId(), order.getTotalCost());  
        mileageApiService.minusMileages(customerId, mileage);  
    } else {  
        throw new IllegalArgumentException("Mileage is not enough");  
    }  
} else if {  
    ⚙ Invert 'if' condition  
    ⚙ Unwrap 'else' branch (changes semantics)  
    ⚙ Swap 'if' statements  
    ⚙ Split 'else if'  
    ⚙ Press Ctrl+Shift+I to open preview  
    Mileage mileage = new Mileage(customerId, order.getId(), mileagePoint);  
    mileageApiService.minusMileages(customerId, mileage);  
    payWithCard(order, customerId);  
}
```

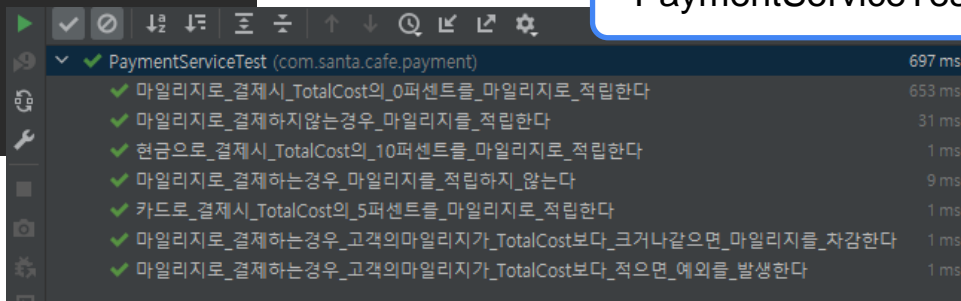
Step 3. Nested Condition

Nested Condition을 제거하면서 PaymentType에 따라 기능이 분리됨을 확인할 수 있다.

작업 후

```
if(payment == 1) {  
    Mileage mileage = new Mileage(customerId, order.getId(), mileagePoint);  
    mileageApiService.saveMileages(customerId, mileage);  
    payWithCash(order, customerId);  
} else if (payment == 2) {  
    Mileage mileage = new Mileage(customerId, order.getId(), mileagePoint);  
    mileageApiService.saveMileages(customerId, mileage);  
    payWithCard(order, customerId);  
} else if (payment == 3) {  
    int customerMileage = mileageApiService.getMileages(customerId);  
    if(customerMileage >= order.getTotalCost()) {  
        Mileage mileage = new Mileage(customerId, order.getId(), order.getTotalCost());  
        mileageApiService.minusMileages(customerId, mileage);  
    } else {  
        throw new BizException("mileage is not enough");  
    }  
}
```

PaymentServiceTest



Test Case	Duration
✓ PaymentServiceTest (com.santa.cafe.payment)	697 ms
✓ 마일리지로 결제시_TotalCost의_0퍼센트를_마일리지로_적립한다	653 ms
✓ 마일리지로 결제하지않는경우_마일리지를_적립한다	31 ms
✓ 현금으로 결제시_TotalCost의_10퍼센트를_마일리지로_적립한다	1 ms
✓ 마일리지로 결제하는경우_마일리지를_적립하지_않는다	9 ms
✓ 카드로 결제시_TotalCost의_5퍼센트를_마일리지로_적립한다	1 ms
✓ 마일리지로 결제하는경우_고객의마일리지가_TotalCost보다_크거나같으면_마일리지를_차감한다	1 ms
✓ 마일리지로 결제하는경우_고객의마일리지가_TotalCost보다_적으면_예외를_발생한다	1 ms

Step 4. 추상화 (Abstraction)

추상화를 통해서 확장에 유연하게 대응을 할 수 있다.

- 결제 유형에 따라 마일리지 포인트 계산, 결제 로직이 다르다.
- 다른 결제 유형이 추가가 된다면, 해당 기능에 조건문이 추가가 될 것이다.

어떤 방식으로 추상화를 진행할지는 고민이 필요하다.

: 인터페이스 vs 상속

```
public double getMileagePoint(int payment, double totalCost) {  
    double mileagePoint = 0;  
    switch(payment) {  
        case 1:  
            mileagePoint = totalCost * 0.1;  
            break;  
        case 2:  
            mileagePoint = totalCost * 0.05;  
            break;  
        case 3:  
            break;  
    }  
    return mileagePoint;  
}
```

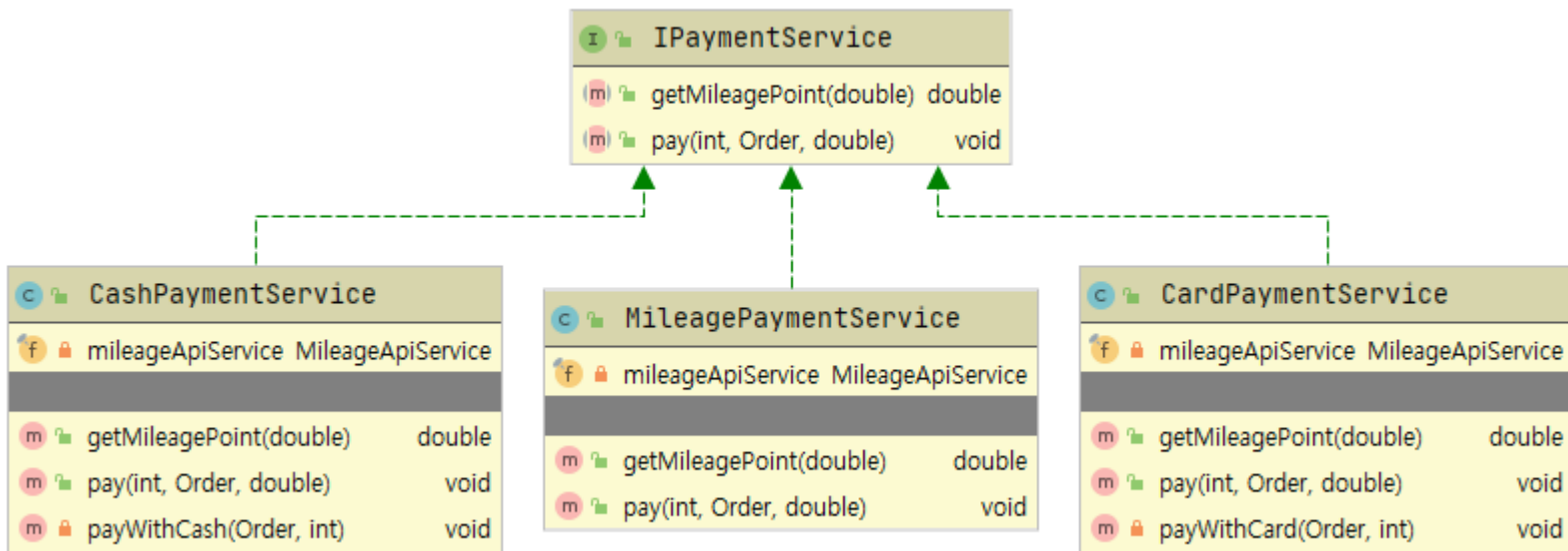
PaymentService.getMileagePoint

```
public void pay(int customerId, int payment, Order order, double mileagePoint) {  
    if(payment == 1) {  
        Mileage mileage = new Mileage(customerId, order.getId(), mileagePoint);  
        mileageApiService.saveMileages(customerId, mileage);  
        payWithCash(order, customerId);  
    } else if (payment == 2) {  
        Mileage mileage = new Mileage(customerId, order.getId(), mileagePoint);  
        mileageApiService.saveMileages(customerId, mileage);  
        payWithCard(order, customerId);  
    } else if (payment == 3) {  
        int customerMileage = mileageApiService.getMileages(customerId);  
        if(customerMileage >= order.getTotalCost()) {  
            Mileage mileage = new Mileage(customerId, order.getId(), order.getTotalCost());  
            mileageApiService.minusMileages(customerId, mileage);  
        } else {  
            throw new BizException("mileage is not enough");  
        }  
    }  
}
```

PaymentService.pay

Step 4. 추상화 (Abstraction)

- **IPaymentService**를 인터페이스로 하여 3개의 Payment 구현체가 추가된다.
- 작업이 완료되면 다음과 같은 클래스 구성을 가진다.



Step 4. 추상화 (Abstraction)

- 결제를 담당하는 Interface 서비스 생성 후, 확장에 유연하게 대응을 할 수 있다.
- IDE의 Refactor 기능을 사용하여 Interface 추출

```
public interface IPaymentService {  
  
    double getMileagePoint(double totalCost);  
    void pay(int customerId, Order order, double mileagePoint);  
  
}
```

```
@Service  
public class CashPaymentService implements IPaymentService {  
    @Override  
    public double getMileagePoint(double totalCost) {  
        return 0;  
    }  
  
    @Override  
    public void pay(int customerId, Order order, double mileagePoint) {  
    }  
}
```

```
@Service  
public class CardPaymentService implements IPaymentService {  
    @Override  
    public double getMileagePoint(double totalCost) {  
        return 0;  
    }  
  
    @Override  
    public void pay(int customerId, Order order, double mileagePoint) {  
    }  
}
```

```
@Service  
public class MileagePaymentService implements IPaymentService {  
    @Override  
    public double getMileagePoint(double totalCost) {  
        return 0;  
    }  
  
    @Override  
    public void pay(int customerId, Order order, double mileagePoint) {  
    }  
}
```

Step 4. 추상화 (Abstraction)

- 마일리지 계산 로직을 각 클래스 구현부로 옮긴다.
- 결제 유형 추가 시 더욱 유연하게 대처할 수 있다.

PaymentService.getMileagePoint

```
public double getMileagePoint(int payment, double totalCost) {  
    double mileagePoint = 0;  
    switch(payment) {  
        case 1:  
            mileagePoint = totalCost * 0.1;  
            break;  
        case 2:  
            mileagePoint = totalCost * 0.05;  
            break;  
        case 3:  
            break;  
    }  
    return mileagePoint;  
}
```

CashPaymentService

```
@Service  
public class CashPaymentService implements IPaymentService {  
    @Override  
    public double getMileagePoint(double totalCost) {  
        return 0;  
    }  
}
```

CardPaymentService

```
@Service  
public class CardPaymentService implements IPaymentService {  
    @Override  
    public double getMileagePoint(double totalCost) {  
        return 0;  
    }  
}
```

MileagePaymentService

```
@Service  
public class MileagePaymentService implements IPaymentService {  
    @Override  
    public double getMileagePoint(double totalCost) {  
        return 0;  
    }  
  
    @Override  
    public void pay(int customerId, Order order, double mileagePoint) {  
    }  
}
```

Step 4. 추상화 (Abstraction)

- 결제 로직에 필요한 private 함수(payWithCash, payWithCard)를 먼저 옮긴다.
- 결제 로직을 각 클래스 구현부로 옮긴다.

PaymentService.pay

```
public void pay(int customerId, int payment, Order order, double mileagePoint) {  
    if (payment == 1) {  
        Mileage mileage = new Mileage(customerId, order.getId(), mileagePoint);  
        mileageApiService.saveMileages(customerId, mileage);  
        payWithCash(order, customerId);  
    } else if (payment == 2) {  
        Mileage mileage = new Mileage(customerId, order.getId(), mileagePoint);  
        mileageApiService.saveMileages(customerId, mileage);  
        payWithCard(order, customerId);  
    } else if (payment == 3) {  
        int customerMileage = mileageApiService.getMileages(customerId);  
        if (customerMileage >= order.getTotalCost()) {  
            Mileage mileage = new Mileage(customerId, order.getId(), order.getTotalCost());  
            mileageApiService.minusMileages(customerId, mileage);  
        } else {  
            throw new BizException("mileage is not enough");  
        }  
    }  
}
```

CashPaymentService

```
@Service  
public class CashPaymentService implements IPaymentService {  
    @Override  
    public double getMileagePoint(double totalCost) {  
        return 0;  
    }  
}
```

CardPaymentService

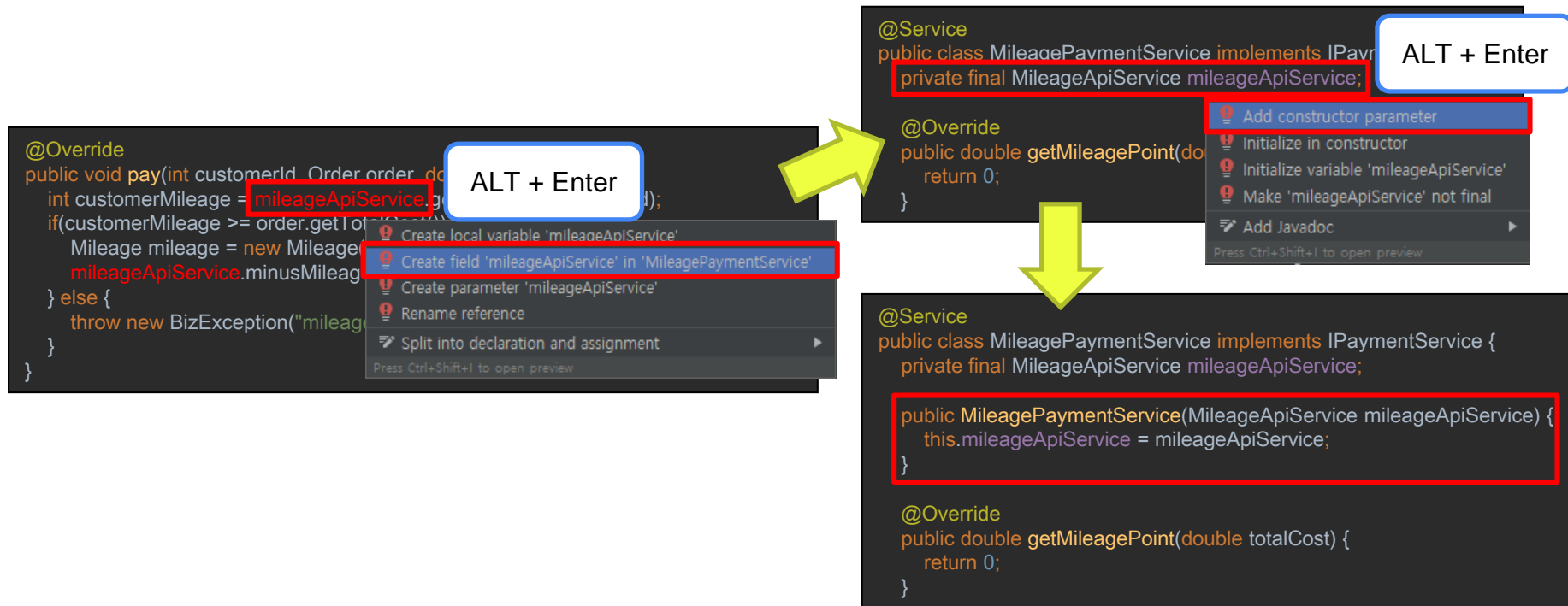
```
@Service  
public class CardPaymentService implements IPaymentService {  
    @Override  
    public double getMileagePoint(double totalCost) {  
        return 0;  
    }  
}
```

MileagePaymentService

```
@Service  
public class MileagePaymentService implements IPaymentService {  
    @Override  
    public double getMileagePoint(double totalCost) {  
        return 0;  
    }  
  
    @Override  
    public void pay(int customerId, Order order, double mileagePoint) {  
    }  
}
```


Step 4. 추상화 (Abstraction)

- 결제 로직에는 MileageApiService 빈 주입이 필요하다.
- 이를 위해 field(final 속성 추가)를 추가하고 생성자에 빈 주입 코드를 추가한다.
- PaymentType별 클래스에 모두 같은 작업을 수행한다.



Step 4. 추상화 (Abstraction)

PaymentService 리팩토링

- payment 값에 따라서 적절한 구현체를 사용하도록 리팩토링한다.
- PaymentService에 새로 생성된 의존성을 주입한다.
- 대상 : 생성자, getMileagePoint, pay 메소드

```
public double getMileagePoint(int payment, double totalCost) {  
    switch(payment) {  
        case 1:  
            return cashPaymentService.getMileagePoint(totalCost);  
        case 2:  
            return cardPaymentService.getMileagePoint(totalCost);  
        case 3:  
            return mileagePaymentService.getMileagePoint(totalCost);  
    }  
    return 0.0;  
}
```

```
public void pay(int customerId, int payment, Order order, double mileagePoint) {  
    if(payment == 1) {  
        cashPaymentService.pay(customerId, order, mileagePoint);  
    } else if (payment == 2) {  
        cashPaymentService.pay(customerId, order, mileagePoint);  
    } else if (payment == 3) {  
        cashPaymentService.pay(customerId, order, mileagePoint);  
    }  
}
```

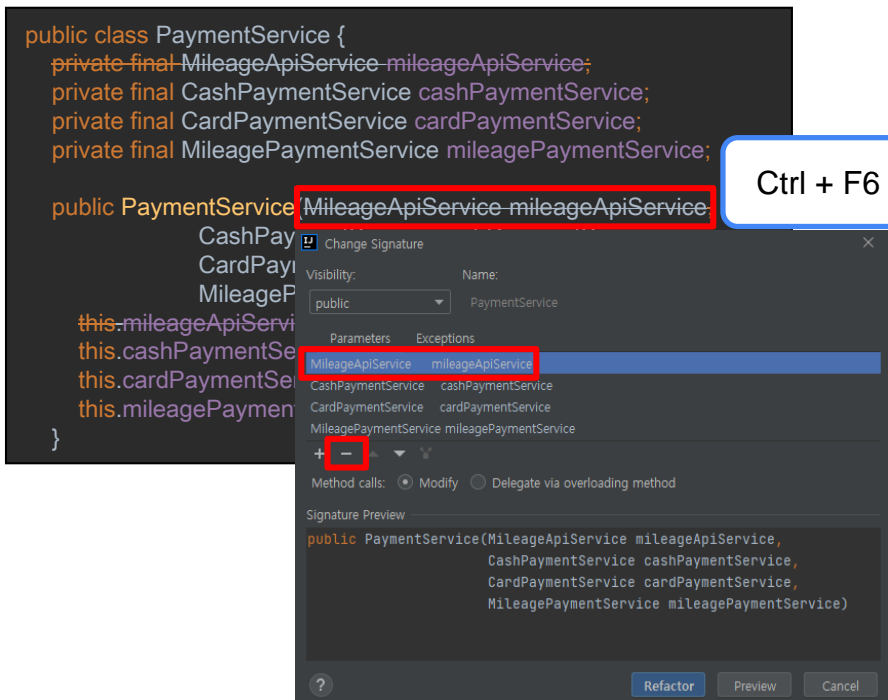
PaymentService

```
public class PaymentService {  
    private final MileageApiService mileageApiService;  
    private final CashPaymentService cashPaymentService;  
    private final CardPaymentService cardPaymentService;  
    private final MileagePaymentService mileagePaymentService;  
  
    public PaymentService(MileageApiService mileageApiService,  
        CashPaymentService cashPaymentService,  
        CardPaymentService cardPaymentService,  
        MileagePaymentService mileagePaymentService) {  
        this.mileageApiService = mileageApiService;  
        this.cashPaymentService = cashPaymentService;  
        this.cardPaymentService = cardPaymentService;  
        this.mileagePaymentService = mileagePaymentService;  
    }  
}
```

Step 4. 추상화 (Abstraction)

PaymentService 리팩토링

- MileageApiService는 사용되지 않으므로 삭제한다.(IDE 기능을 활용하면 편리함)

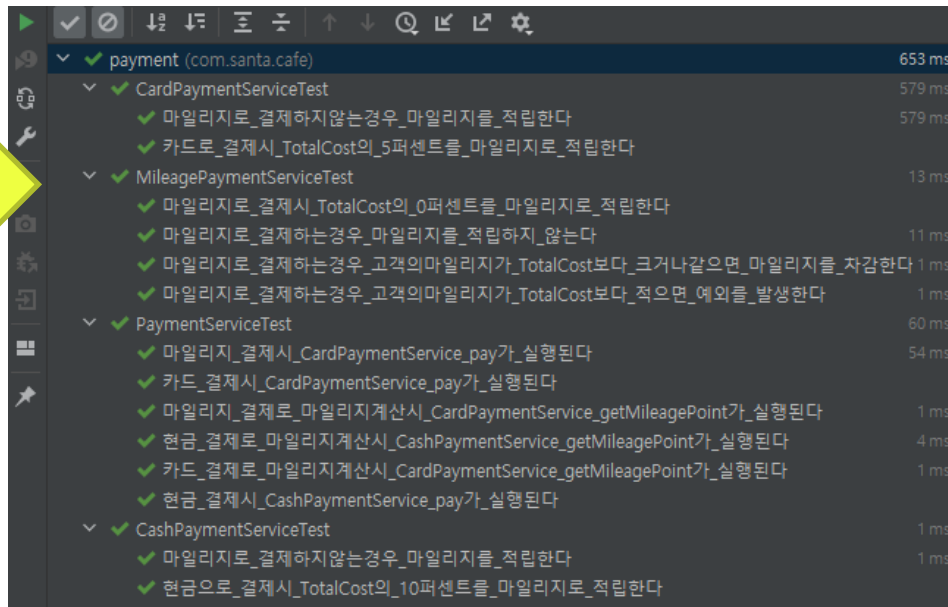
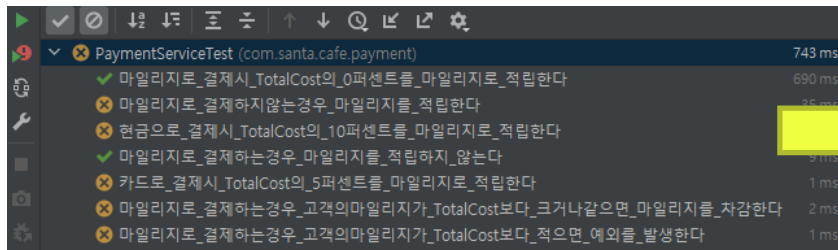


PaymentService

```
public class PaymentService {  
    private final CashPaymentService cashPaymentService;  
    private final CardPaymentService cardPaymentService;  
    private final MileagePaymentService mileagePaymentService;  
  
    public PaymentService(  
        CashPaymentService cashPaymentService,  
        CardPaymentService cardPaymentService,  
        MileagePaymentService mileagePaymentService) {  
        this.cashPaymentService = cashPaymentService;  
        this.cardPaymentService = cardPaymentService;  
        this.mileagePaymentService = mileagePaymentService;  
    }  
}
```

Step 4. 추상화 (Abstraction)

- **PaymentService** 생성자가 변경됐기 때문에 **PaymentServiceTest**에서 오류가 발생한다.
 - **CashPaymentService, CardPaymentService, MileagePaymentService** mock 추가
- **PaymentService** 변경으로 인해 기존 **Stubbing**이 동작하지 않는다.
 - **PaymentType**에 따라 테스트 코드를 분리해야 한다.



Step 5. Magic Number

의미를 파악하기 어려운 모호한 변수를 Enum을 이용한 상수로 추출

```
public double getMileagePoint(int payment, double totalCost) {  
    if (payment == 1) {  
        return cashPaymentService.getMileagePoint(totalCost);  
    } else if (payment == 2) {  
        return cardPaymentService.getMileagePoint(totalCost);  
    } else if (payment == 3) {  
        return mileagePaymentService.getMileagePoint(totalCost);  
    }  
    return 0.0;  
}
```

- int형 code 사용은 가독성을 저하시킨다.
 - 1: CASH
 - 2: CARD
 - 3: MILEAGE
- PaymentType enum 생성하여 리팩토링

```
public enum PaymentType {  
    CASH(1), CARD(2), MILEAGE(3);  
  
    private int code;  
  
    PaymentType(int code) {  
        this.code = code;  
    }  
  
    public int getCode() {  
        return code;  
    }  
  
    public static PaymentType fromCode(int code) {  
        return Stream.of(values())  
            .filter(type -> type.getCode() == code)  
            .findFirst()  
            .orElseThrow(() -> new NotFoundException("invalid PaymentType code"));  
    }  
}
```

Step 5. Magic Number

상수 생성 Refactoring 장점

- 상수를 생성함으로써 코드 가독성이 올라간다
- 추후에 payment값이 변경이 되더라도, 변경범위가 PaymentType enum으로 국한

```
public double getMileagePoint(int payment, double totalCost) {  
    if (payment == 1) {  
        return cashPaymentService.getMileagePoint(totalCost);  
    } else if (payment == 2) {  
        return cardPaymentService.getMileagePoint(totalCost);  
    } else if (payment == 3) {  
        return mileagePaymentService.getMileagePoint(totalCost);  
    }  
    return 0.0;  
}
```



```
public double getMileagePoint(PaymentType paymentType, double totalCost) {  
    if (paymentType == PaymentType.CASH) {  
        return cashPaymentService.getMileagePoint(totalCost);  
    } else if (paymentType == PaymentType.CARD) {  
        return cardPaymentService.getMileagePoint(totalCost);  
    } else if (paymentType == PaymentType.MILEAGE) {  
        return mileagePaymentService.getMileagePoint(totalCost);  
    }  
    return 0.0;  
}
```

PaymentService.getMileagePoint

```
public void pay(int customerId, int payment, Order order, double  
mileagePoint) {  
    if (payment == 1) {  
        cashPaymentService.pay(customerId, order, mileagePoint);  
    } else if (payment == 2) {  
        cardPaymentService.pay(customerId, order, mileagePoint);  
    } else if (payment == 3) {  
        mileagePaymentService.pay(customerId, order, mileagePoint);  
    }  
}
```



```
public void pay(int customerId, PaymentType paymentType, Order order,  
double mileagePoint) {  
    if (paymentType == PaymentType.CASH) {  
        cashPaymentService.pay(customerId, order, mileagePoint);  
    } else if (paymentType == PaymentType.CARD) {  
        cardPaymentService.pay(customerId, order, mileagePoint);  
    } else if (paymentType == PaymentType.MILEAGE) {  
        mileagePaymentService.pay(customerId, order, mileagePoint);  
    }  
}
```

PaymentService.pay

Step 5. Magic Number

OrderService의 호출부분 변경

- PaymentType.fromValue 메소드를 사용하여 int형을 enum type으로 변환


```
double totalCost = this.getDiscountedTotalCost(getTotalCost(orderItems));
order.setTotalCost(totalCost);

double mileagePoint = paymentService.getMileagePoint(payment totalCost);

paymentService.pay(customerId, payment order, mileagePoint);

order.setMileagePoint(mileagePoint);
orderRepository.save(order);
orderItemRepository.saveAll(orderItems);
```

OrderService.create



```
double totalCost = this.getDiscountedTotalCost(getTotalCost(orderItems));
order.setTotalCost(totalCost);

double mileagePoint = paymentService.getMileagePoint(PaymentType.fromCode(payment) totalCost);

paymentService.pay(customerId, PaymentType.fromCode(payment) order, mileagePoint);

order.setMileagePoint(mileagePoint);
orderRepository.save(order);
orderItemRepository.saveAll(orderItems);
```

Step 5. Magic Number

PaymentService 변경으로 인해 테스트 코드에서 오류가 발생한다.

- 오류 발생하는 테스트 코드 내 paymentType 사용하는 부분을 수정한다.

```
@Test
public void 현금_결제로_마일리지계산시_CashPaymentService_getMileagePoint가_실행된다() {
    //given

    //when
    subject.getMileagePoint(PAYMENT_CASH, 200.0);

    //then
    verify(mockCashPaymentService, times(1)).getMileagePoint(200.0);
}
```



PaymentServiceTest

```
@Test
public void 현금_결제로_마일리지계산시_CashPaymentService_getMileagePoint가_실행된다() {
    //given

    //when
    subject.getMileagePoint(PaymentType.CASH, 200.0);

    //then
    verify(mockCashPaymentService, times(1)).getMileagePoint(200.0);
}
```


Step 6. Factory Pattern

paymentType에 따라서 어떤 PaymentService를 사용할지를 Factory Method Pattern을 사용하여 결정할 수 있다.

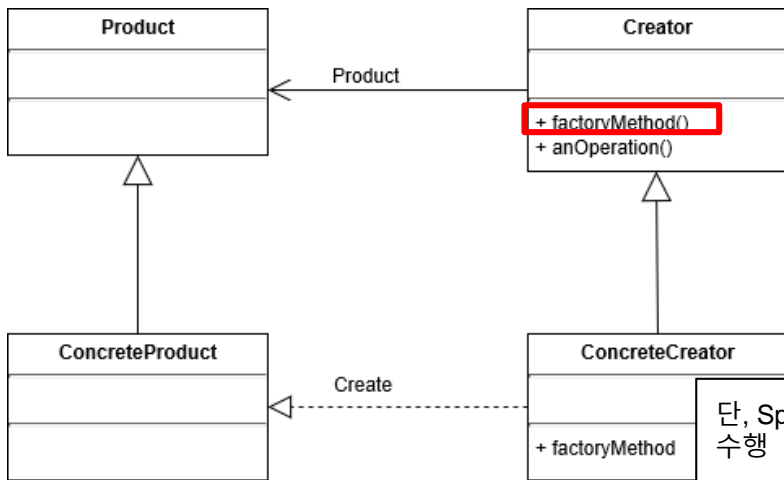
```
public double getMileagePoint(PaymentType paymentType, double totalCost) {  
    if (paymentType == PaymentType.CASH) {  
        return cashPaymentService.getMileagePoint(totalCost);  
    } else if (paymentType == PaymentType.CARD) {  
        return cardPaymentService.getMileagePoint(totalCost);  
    } else if (paymentType == PaymentType.MILEAGE) {  
        return mileagePaymentService.getMileagePoint(totalCost);  
    }  
    return 0.0;  
}
```

```
public void pay(int customerId, PaymentType paymentType, Order order, double mileagePoint) {  
    if (paymentType == PaymentType.CASH) {  
        cashPaymentService.pay(customerId, order, mileagePoint);  
    } else if (paymentType == PaymentType.CARD) {  
        cardPaymentService.pay(customerId, order, mileagePoint);  
    } else if (paymentType == PaymentType.MILEAGE) {  
        mileagePaymentService.pay(customerId, order, mileagePoint);  
    }  
}
```

Step 6. Factory Pattern

- Factory Pattern의 주요 목적은 Dependency Inversion하여 객체를 생성하는 것이다.
- Factory Method Pattern
 - 서브 클래스에서 어떤 클래스를 만들지 결정함으로써 객체 생성을 캡슐화 한다.

Factory Method Pattern



단, Spring 에서 Bean 생성 및 의존성 주입 역할 수행


실습에서 Bean을 생성하는 ConcreteCreator는 생성하지 않는다.

Step 6. Factory Pattern


Factory 패턴을 적용하기 위해 기존 IPaymentService 인터페이스 추가 작업이 필요하다.

- getPaymentType() 메소드를 추가
- 구현체별로 적절한 payment Type 값을 리턴하도록 구현한다.


```
public interface IPaymentService {  
    double getMileagePoint(int payment, double totalCost);  
  
    void pay(int customerId, int payment, Order order, double mileageP  
  
    int getPaymentType();  
}
```



```
@Service  
public class CashPaymentService implements IPaymentService {  
    @Override  
    public int getPaymentType() {  
        return 1;  
    }  
}
```



```
@Service  
public class CardPaymentService implements IPaymentService {  
    @Override  
    public int getPaymentType() {  
        return 2;  
    }  
}
```



```
@Service  
public class MileagePaymentService implements IPaymentService {  
    @Override  
    public int getPaymentType() {  
        return 3;  
    }  
}
```

Step 6. Factory Pattern

실습에서는 Factory Method Pattern를 활용

- 빈을 생성하는 역할을 스프링 프레임워크에서 수행
- 스프링 프레임워크의 의존성 주입 기능을 활용하여 Factory 구성

PaymentServiceFactory

```
@Component
public class PaymentServiceFactory {
    private final Map<PaymentType, IPaymentService> serviceMap = new HashMap<>();

    public PaymentServiceFactory(List<IPaymentService> listService) {
        if (CollectionUtils.isEmpty(listService)) {
            throw new IllegalArgumentException("parameter error");
        }

        for (IPaymentService service : listService) {
            serviceMap.put(service.getPaymentType(), service);
        }
    }

    public IPaymentService service(int paymentType) {
        if (map.get(paymentType) == null) {
            throw new IllegalArgumentException("payment type is not support");
        }
        return map.get(paymentType);
    }
}
```

• Spring 4.3 이후부터 같은 인터페이스를 가진 빈을 리스트로 주입 받을 수 있다.

• PaymentType을 사용하여 원하는 Service를 받아 사용할 수 있다.

Step 6. Factory Pattern

PaymentService에서 Factory 클래스를 사용하도록 리팩토링한다.
새로운 payment type이 추가되더라도 확장에 용이한 구조로 설계가 변경되었다.

PaymentService

```
@Service
public class PaymentService {
    private final PaymentFactory paymentFactory;

    public PaymentService(PaymentFactory paymentFactory) {
        this.paymentFactory = paymentFactory;
    }

    public double getMileagePoint(int payment, double totalCost) {
        return this.paymentFactory.service(payment).getMileagePoint(payment, totalCost);
    }

    public void pay(int customerId, int payment, Order order, double mileagePoint) {
        this.paymentFactory.service(payment).pay(customerId, payment, order, mileagePoint);
    }
}
```

- 생성자에서는 PaymentFactory bean만 의존성 주입을 받으면 된다.
- 각 비즈니스 method 내부에서는 service(payment) 메소드를 통해서 factory 메소드 내부에서 결제 유형에 따라 어떤 paymentService를 사용해야 하는지 결정한다.

Step 6. Factory Pattern

Factory pattern을 적용한 리팩토링이 정상적으로 동작하는지 테스트로 검증을 하기 위해서는 테스트 수정이 필요하다.

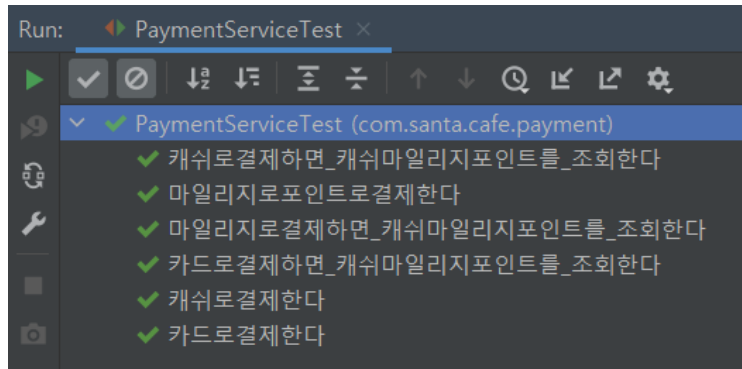
```
@RunWith(MockitoJUnitRunner.class)
public class PaymentServiceTest {
    private PaymentService subject;

    @Mock
    private CashPaymentService mockCashPaymentService;
    @Mock
    private CardPaymentService mockCardPaymentService;
    @Mock
    private MileagePaymentService mockMileagePaymentService;

    @Before
    public void setUp() {
        when(mockCashPaymentService.getPaymentType()).thenReturn(1);
        when(mockCardPaymentService.getPaymentType()).thenReturn(2);
        when(mockMileagePaymentService.getPaymentType()).thenReturn(3);

        PaymentFactory paymentFactory = new PaymentFactory(
            Lists.newArrayList(mockCashPaymentService,
                mockCardPaymentService,
                mockMileagePaymentService)
        );
        subject = new PaymentService(paymentFactory);
    }
}
```

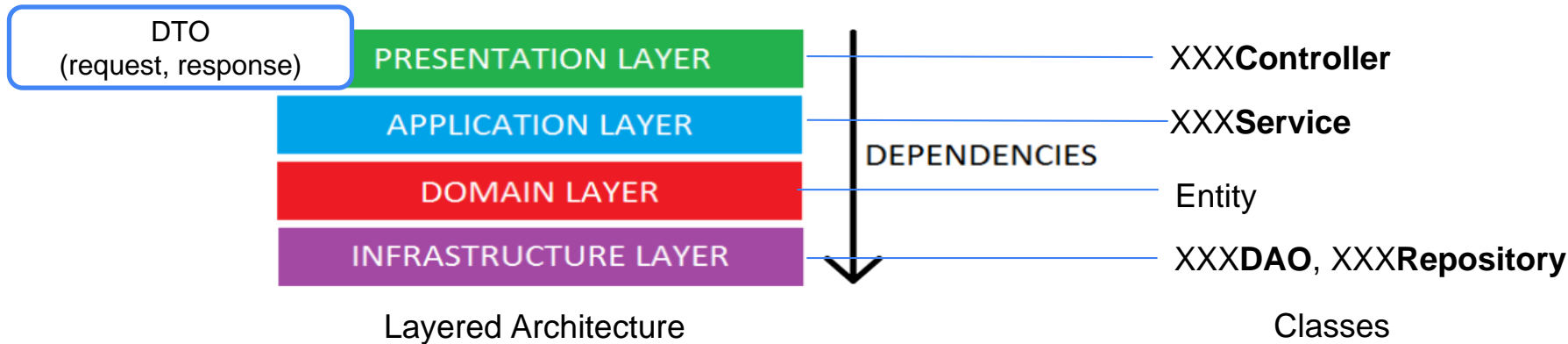
- 테스트 대상(SUT)을 setUp 메소드에서 직접 생성한다.
- PaymentFactory 객체가 실제로 동작하여, payment type에 따라 동적으로 필요한 paymentService를 가져오기 위함이다.



Step 8. DTO 적용

Entity와 DTO를 분리해서 관리해야 하는 이유?

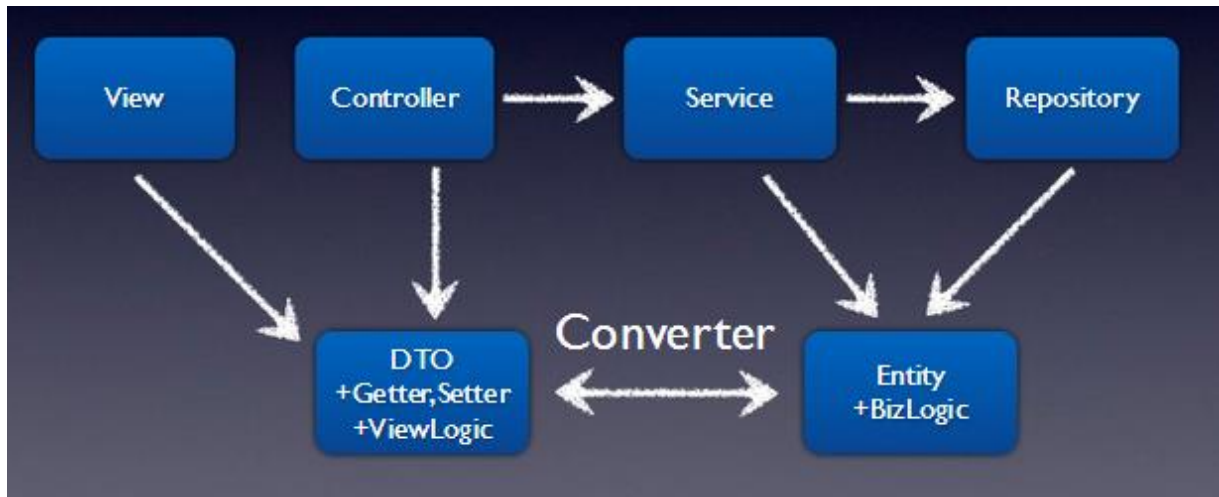
- DB Layer, View Layer 사이에 역할 분리
- Entity
 - 실제 테이블과 매핑, 변경하게 되면 다른 클래스에 영향을 끼친다.
- DTO (Data Transfer Object)
 - Request, Response 용 데이터 전달을 위한 View 용 클래스
 - 자주 변경될 수 있으므로, 클래스 분리를 해주는 것이 좋다.



Step 8. DTO 적용

DTO를 사용해야 하는 이유?

- Request, Response 객체가 항상 Entity와 동일할 것인가?
- 순환 참조가 발생할 수 있다.
- Entity 과 DB 테이블 스키마와 거의 동일한 구조를 가질 것인데, Client에게 노출하는 것이 적당한가?
- DTO를 생성하면서 어떤 값을 요청하고, 어떤 값을 응답 받을지에 대한 설계작업을 수행할 수 있다.



Step 8. DTO 적용 – Entity + DTO

목표 1. 하나의 Order, OrderItem 클래스가 DTO와 Entity 역할을 모두 담당하도록 변경해보자.

```
@Entity
public class OrderItem {
    ...

    @JsonIgnore
    @ManyToOne
    @JoinColumn(name = "beverage_id", referencedColumnName = "id")
    private Beverage beverage;

    @Transient
    private int beverageId;

    public OrderItem() {
    }

    public int getBeverageId() {
        return beverageId;
    }

    public void setBeverageId(int beverageId) {
        this.beverageId = beverageId;
    }

    ...
}
```

OrderItem

- DTO로써 View용으로 사용할
필드(beverageId)에 @Transient를 선언하여,
Entity와 연관이 없는 필드로 사용할 수 있다.
- DTO 용도로 추가한 필드에 getter, setter
method를 추가한다.

Step 8. DTO 적용 – Entity + DTO

OrderService의 create 메소드의 orderItemList 매개변수 타입을 변경한다.

(before) Map<String, Object> -> (after) List<OrderItem>

create 메소드

Method 명이나 Method 선언부에 커서를 두고, **CTRL + F6**를 클릭해서 “Change Signature”를 수행한다.

```
public Order create(int customerId, List<Map<String, Object>> orderItemList, int payment) {  
    Customer customer = cus  
    Order order = new Order  
    order.setStatus(OrderSt  
    order.setCustomer(custo  
    order.setPayment(payer  
  
    IPaymentService service  
    int customerId  
  
    List<OrderItem> orderIt  
    order.setTotalCost(this  
    order.setMileagePoint(s  
    int payment  
}
```

Change Signature

Visibility: public Return type: Order Name: create

Parameters

Type: List<Map<String, Object>> Name: orderItemList

Change Signature

Visibility: public Return type: Order Name: create

Parameters

Type: List<OrderItem> Name: orderItemList

Method calls: ☒ Modify ☐ Delegate via overloading method

Signature Preview

```
public Order create(int customerId,  
    List<OrderItem> orderItemList,  
    int payment)
```

Step 8. DTO 적용 – Entity + DTO

create 메소드 signature 변경에 따른 컴파일 에러를 수정한다.

1. **OrderService.getOrderItems** 메소드의 매개변수 타입 변경 및 리팩토링 진행한다.

- for문 orderItemMap 변수를 orderItem으로 변경
- for문 내부에서 불필요한 OrderItem 객체 생성 제거
- OrderItem에 count 세팅 로직 제거

```
private List<OrderItem> getOrderItems(List<Map<String, Object>> orderItemList, Order order) {
    List<OrderItem> orderItems = new ArrayList<>();
    for(Map<String, Object> orderItemMap: orderItemList) {
        Beverage beverage = beverageRepository.getOne(orderItemMap.get("beverageId"));
        if(beverage == null) {
            continue;
        }
        OrderItem orderItem = new OrderItem();
        orderItem.setCount((Integer) orderItemMap.get("count"));
        orderItem.setBeverage(beverage);
        orderItem.setOrder(order);
        orderItems.add(orderItem);
    }
    return orderItems;
}
```

Method 명이나 Method 선언부에
커서를 두고,
CTRL + F6를 클릭해서 “Change
Signature” 를 수행한다.

OrderService 변경 후

```
private List<OrderItem> getOrderItems(List<OrderItem> orderItemList, Order order) {
    List<OrderItem> orderItems = new ArrayList<>();
    for(OrderItem orderItem: orderItemList) {
        Beverage beverage = beverageRepository.getOne(orderItem.getBeverageId());
        if(beverage == null) {
            continue;
        }
        orderItem.setBeverage(beverage);
        orderItem.setOrder(order);
        orderItems.add(orderItem);
    }
    return orderItems;
}
```

Step 8. DTO 적용 – Entity + DTO

create 메소드 signature 변경에 따른 컴파일 에러를 수정한다.

2. OrderServiceTest에 create 메소드 테스트 케이스들 컴파일 에러를 수정한다.
 - create 메소드의 orderItemList 매개변수 타입을 Map에서 **OrderItem**으로 변경

```
@Test
public void 주문을하면_OrderItem들의_가격을_합한_TotalCost를_계산한다() {
```

OrderServiceTest 변경 후

```
@Test
public void 등록되지않은_음료가_OrderItem에_포함되면_해당금액을_계산한다() {
```

```
@Test
public void 매월_마지막날에_주문하면_TotalCost에서_10퍼센트가_감소한다() {
```

OrderServiceTest 편집 창에서
F2 단축키를 클릭해서 컴파일 에러
위치로 빠르게 이동 후 에러를 수정한다.

```
@Test
public void 주문을하면_OrderItem들의_가격을_합한_TotalCost를_계산한다() {
    //given
    OrderItem orderItem = new OrderItem();
    orderItem.setBeverageId(1);
    orderItem.setCount(2);

    //when
    Order result = subject.create(CUSTOMER_ID, Collections.singletonList(orderItem),
        PAYMENT_CASH);

    //then
    assertThat(result.getTotalCost(), is(2000.0));
}
```

Step 8. DTO 적용 – Entity + DTO

create 메소드 signature 변경에 따른 컴파일 에러를 수정한다.

3. **OrderController.create** 메소드의 `orderService.create` 호출부 변경한다.

- `List<Map<String, Object>> orderItems` 대신에
새롭게 만든 `List<OrderItem> orderItemList` 변수를 `create` 변수의 인자로 전달

```
@PostMapping
public Order create(@RequestBody Map<String, Object> orderMap) {
    int customerId = (int) orderMap.get("customerId");
    int payment = (int) orderMap.get("payment");

    List<Map<String, Object>> orderItems = (List<Map<String, Object>>) orderMap.get("orderItems");

    List<OrderItem> orderItemList = new ArrayList<>();
    for (Map<String, Object> orderItem : orderItems) {
        OrderItem item = new OrderItem();
        item.setBeverageId((Integer) orderItem.get("beverageId"));
        item.setCount((Integer) orderItem.get("count"));
        orderItemList.add(item);
    }

    return orderService.create(customerId, orderItemList, payment);
}
```

OrderController 변경 후

Step 8. DTO 적용 – Entity + DTO

목표 1. 하나의 Order, OrderItem 클래스가 DTO와 Entity 역할을 모두 담당하도록 변경해보자.

Order

```
@Entity
@Table(name = "\"ORDER\"")
public class Order {
    ...
    @OneToMany(mappedBy = "order", cascade = CascadeType.ALL)
    private final List<OrderItem> orderItems = new ArrayList<>();

    @ManyToOne
    private Customer customer;

    @Transient
    public int customerId;

    public Order() {
    }

    public int getCustomerId() {
        return customerId;
    }

    public void setCustomerId(int customerId) {
        this.customerId = customerId;
    }
    ...
}
```

- DTO로써 View용으로 사용할 필드(customerId)에 @Transient를 선언하여, Entity와 연관이 없는 필드로 사용할 수 있다.
- DTO 용도로 추가한 필드에 getter, setter method를 추가한다.

Step 8. DTO 적용 – Entity + DTO

OrderController.create 메소드에 RequestBody 값을 매핑할 객체를 Map에서 DTO 역할을 수행하는 Order 객체로 변경한다.

- create 메소드의 매개변수 타입을 Map<String, Object>에서 Order 객체로 변경
- orderService.create 호출을 위하여 map으로 부터 변수를 생성하는 로직은 불필요하므로 제거
- RequestBody 값이 Object로 잘 매핑되는지 확인하기 위해 OrderControllerSliceTest 소스 변경

사내 Github 주소 : OrderControllerSliceTest 소스 로컬로 복사하세요.

https://code.sdsdev.co.kr/act-edu/cafe/wiki/Step8_DTO_OrderControllerSliceTest

OrderControllerSliceTest
소스 복사

OrderController 변경 후

```
@PostMapping
public Order create(@RequestBody Order order) {
    return orderService.create(order.getCustomerId(),
        order.getOrderItems(),
        order.getPayment());
}
```

```
@Test
public void 주문정보로_POST_API를_요청하면_해당_주문을_생성한다() throws
Exception {
    final int NEW_ORDER_ID = 1;

    Order order = new Order();
    order.setId(NEW_ORDER_ID);

    OrderItem firstOrderItem = new OrderItem();
    firstOrderItem.setBeverageId(1);
    firstOrderItem.setCount(1);

    OrderItem secondOrderItem = new OrderItem();
    secondOrderItem.setBeverageId(2);
    secondOrderItem.setCount(2);

    when(mockOrderService.create(1, Lists.newArrayList(firstOrderItem,
        secondOrderItem), 3))
        .thenReturn(order);
}
```

Step 8. DTO 적용 – Entity + DTO

목표 1. 하나의 Order, OrderItem 클래스가 DTO와 Entity 역할을 모두 담당하도록 변경해보자.

- 장점
 - 하나의 Object로 Entity와 DTO 역할을 동시에 수행할 수 있다.
 - 생산성이나 유지보수에 좋다.
- 단점
 - API 가 복잡해 질 수록 하나의 Entity 객체로 커버하기가 어렵다.
 - Entity 객체가 테이블이 변경이 없는데도 View에 전달하기 위한 목적으로 잦은 변경이 생긴다.
 - 어노테이션(@Column, @Transient)들이 섞이면서 각 필드가 어떤 역할을 수행하는지 파악이 어려워진다.



“Entity가 DTO로써 기능을 동시에 수행할 때 유지보수가 힘들어지면, 객체 분리를 통해서 DTO와 Entity를 별도로 분리하자.”

Step 9. DTO 적용 – Entity 와 DTO 분리

목표 2. OrderDTO, OrderItemDTO를 별도로 생성해서, DTO와 Entity 역할을 분리하자.

```
public class OrderDTO {  
    private int customerId;  
    private int payment;  
    private List<OrderItemDTO> orderItems;  
    private int id;  
  
    public OrderDTO(int customerId, int payment, List<OrderItemDTO> orderItems) {  
        this.customerId = customerId;  
        this.payment = payment;  
        this.orderItems = orderItems;  
    }  
  
    public int getCustomerId() {  
        return customerId;  
    }  
  
    public void setCustomerId(int customerId) {  
        this.customerId = customerId;  
    }  
  
    public int getPayment() {  
        return payment;  
    }  
  
    public void setPayment(int payment) {  
        this.payment = payment;  
    }  
}
```

OrderDTO

- 패키지 경로 : com.santa.cafe.order.dto
- OrderDTO 클래스 생성
- 필드 추가
- request 매핑 : customerId, payment, orderItems
- response 매핑 : id
- 생성자, getter, setter 추가

참고. 기존에 OrderDTO는 삭제하고 진행하면 됩니다.

Step 9. DTO 적용 – Entity 와 DTO 분리

목표 2. OrderDTO, OrderItemDTO를 별도로 생성해서, DTO와 Entity 역할을 분리하자.

```
public class OrderItemDTO {  
    private int beverageId;  
    private int count;
```

OrderItemDTO

```
    public OrderItemDTO(int beverageId, int count) {  
        this.beverageId = beverageId;  
        this.count = count;  
    }
```

```
    public static OrderItem toEntity(OrderItemDTO orderItemDTO) {  
        OrderItem orderItem = new OrderItem();  
        orderItem.setBeverageId(orderItemDTO.getBeverageId());  
        orderItem.setCount(orderItemDTO.getCount());  
        return orderItem;  
    }
```

```
    public int getBeverageId() {  
        return beverageId;  
    }
```

```
    public void setBeverageId(int beverageId) {  
        this.beverageId = beverageId;  
    }
```

```
    public int getCount() {  
        return count;  
    }
```

```
    public void setCount(int count) {  
        this.count = count;  
    }
```

- 패키지 경로 : com.santa.cafe.order.dto
- 기존 소스에서 생성자 추가
 - Alt + Insert 단축키 > Constructor 선택
- toEntity 메소드 추가
 - DTO를 Entity로 변환하는 static 메소드

Step 9. DTO 적용 – Entity 와 DTO 분리

DTO를 어디서 어디 layer 까지 사용할 것인지 고민이 필요하다.

- Controller에서 DTO를 Entity로 변환시켜 Service에게 파라미터로 보내기.
- Controller에서 DTO 자체를 Service에게 파라미터로 보내고, Service가 메소드 상위에서 Entity로 변환 후 DTO는 사용하지 않기.
- Service가 Controller로 응답을 보낼 때, Entity를 반환하고 Controller가 이를 DTO로 변환하여 사용하기.
- Service가 Controller로 응답을 보낼 때, Entity를 DTO로 변환시켜 반환하기.



“위 방법 중 정답은 없다. 코드의 유지 보수성과 도메인 모델의 보호 관점에서 적합한 방법을 잘 사용하면 된다.

단 무작정 DTO를 만드는 것은 진짜 필요한지 고민해볼 필요가 있다.”

Step 9. DTO 적용 – Entity 와 DTO 분리

이번 실습에서는 다음 방법으로 DTO를 사용합니다.

[Code Convention]

Controller에서 DTO를 Entity로 변환시켜 Service에게 파라미터로 전달.

OrderController

```
@PostMapping
public Order create(@RequestBody Order order) {
    return orderService.create(order.getCustomerId(),
        order.getOrderItems(),
        order.getPayment());
}
```

Change Signature

Visibility:

public

Return type:

Order

Name:

create

Parameters

Exceptions

Type:

OrderDTO

Name:

order

- Change Signature 단축키 : Ct기 + F6
- Order -> OrderDTO로 변환
- 서비스 호출시 DTO -> Entity 변환이 필요하다.

```
@PostMapping
public Order create(@RequestBody OrderDTO order) {
    return orderService.create(order.getCustomerId(),
        order.getOrderItems(),
        order.getPayment());
}
```

Step 9. DTO 적용 – Entity 와 DTO 분리

[Code Convention]

Controller에서 DTO를 Entity로 변환시켜 Service에게 파라미터로 전달

OrderController

```
@PostMapping
public Order create(@RequestBody OrderDTO order) {
    List<OrderItem> orderItems = new ArrayList<>();
    for (OrderItemDTO orderItemDTO : order.getOrderItems()) {
        OrderItem orderItem = OrderItemDTO.toEntity(orderItemDTO);
        orderItems.add(orderItem);
    }

    return orderService.create(order.getCustomerId(),
        orderItems,
        order.getPayment());
}
```



OrderItem

```
public class OrderItemDTO {
    ...
    public static OrderItem toEntity(OrderItemDTO orderItem) {
        OrderItem orderItem = new OrderItem();
        orderItem.setBeverageId(orderItemDTO.getBeverageId());
        orderItem.setCount(orderItemDTO.getCount());
        return orderItem;
    }

    public static List<OrderItem> toEntity(List<OrderItemDTO> orderItemDTOList) {
        List<OrderItem> orderItems = new ArrayList<>();
        for (OrderItemDTO orderItemDTO : orderItemDTOList) {
            OrderItem orderItem = OrderItemDTO.toEntity(orderItemDTO);
            orderItems.add(orderItem);
        }

        return orderItems;
    }
}
```

- OrderService는 Entity를 매개변수로 전달받기 때문에, DTO -> Entity로 변환이 필요하다.
- 공통적으로 사용하기 때문에 **Entity(OrderItem)에 static method로 이동할 수 있다.**

OrderController

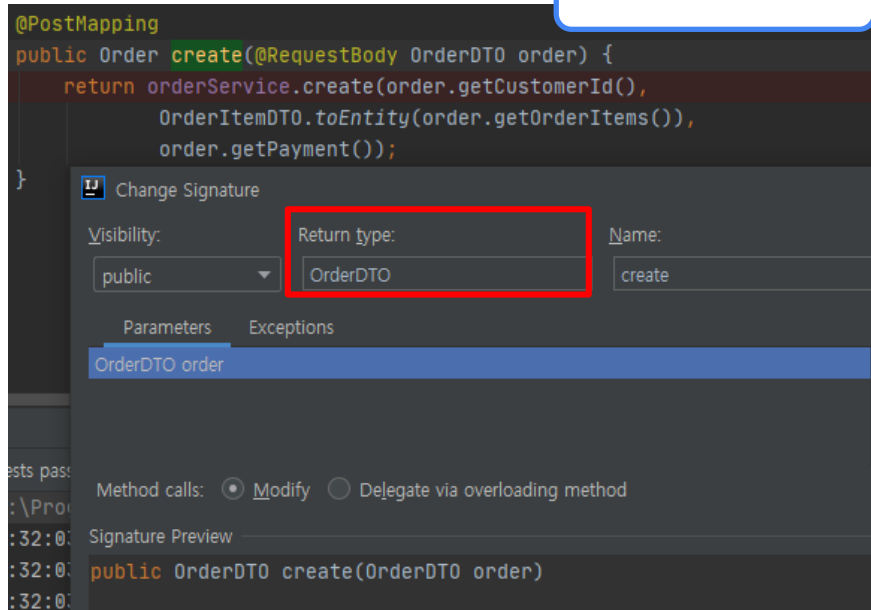
```
@PostMapping
public Order create(@RequestBody OrderDTO order) {
    return orderService.create(order.getCustomerId(),
        OrderItemDTO.toEntity(order.getOrderItems()),
        order.getPayment());
}
```

Step 9. DTO 적용 – Entity 와 DTO 분리

새로운 주문을 생성하는 API의 응답에는 “신규 주문 ID” 만 포함하면 되기 때문에 Order Entity의 전체 필드를 Client로 노출될 필요가 없다.

OrderService가 반환한 Entity를 OrderController에서 DTO로 변환하여 리턴한다.

OrderController



- Change Signature 단축키 : Ct기 + F6
- create 메소드의 리턴타입을 Order -> OrderDTO로 변환
- **Entity -> DTO로 변환하는 메소드가 필요하다.**

```
@PostMapping
public OrderDTO create(@RequestBody OrderDTO order) {
    return orderService.create(order.getCustomerId(),
        OrderItemDTO.toEntity(order.getOrderItems()),
        order.getPayment());
}
```

Step 9. DTO 적용 – Entity 와 DTO 분리

[Code Convention]

Service가 Controller로 응답을 보낼 때, Entity를 반환하고 Controller가 이를 DTO로 변환하여 사용하기.

Order

```
@Entity
@Table(name = "\"ORDER\"")
public class Order {
    ...

    public static OrderDTO toDTO(Order order) {
        OrderDTO orderDTO = new OrderDTO();
        orderDTO.setId(order.getId());
        return orderDTO;
    }

    ...
}
```

- Entity -> DTO로 변환하는 메소드 추가
- Order -> OrderDTO

OrderController

```
@PostMapping
public OrderDTO create(@RequestBody OrderDTO order) {
    return Order.toDTO(orderService.create(order.getId(),
        OrderItemDTO.toEntity(order.getOrderItems()),
        order.getPayment()));
}
```

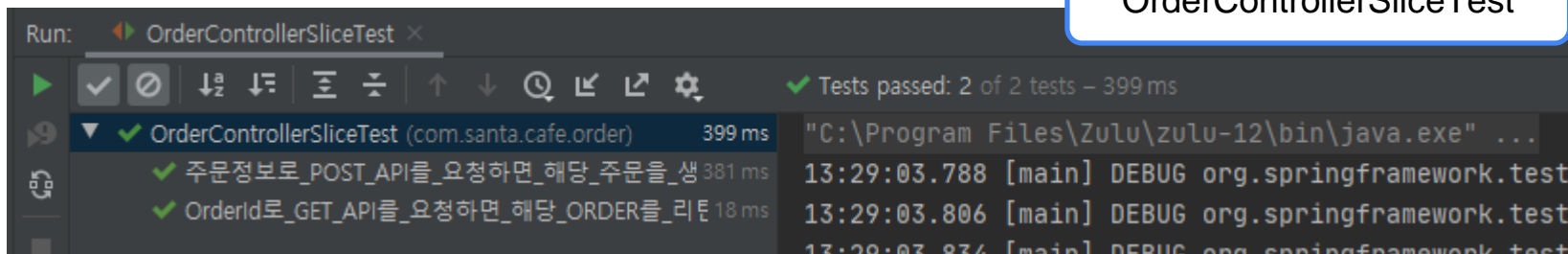
- OrderService.create 메소드의 결과값인 Entity를 Order.toDTO 메소드를 사용하여 DTO로 변환

Step 9. DTO 적용 – Entity 와 DTO 분리

DTO를 적용하여 리팩토링 후에, 리팩토링의 영향을 받는 모듈의 테스트가 정상적으로 통과하는 것을 확인한다.

테스트에서 검증하는 내용

- Request 값이 OrderDTO 객체로 정상적으로 deserialize 되는지
- OrderDTO 객체가 Response로 정상적으로 serialize 되는지



OrderControllerSliceTest

Step 9. DTO 적용 돌아보기

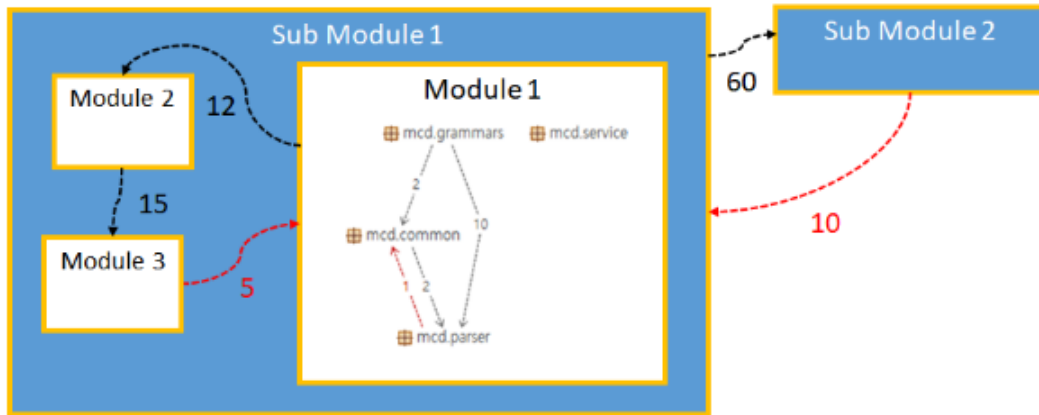
- DTO 사용이 권장되는 경우?
 - 큰 규모의 시스템
- DTO 사용이 큰 효과를 보지 못하는 경우?
 - 작은 규모의 시스템
- 장점
 - Presentation Layer와 Domain 모델간의 결합도 감소
 - 인터페이스 및 API의 안정성
 - 필요한 데이터만을 관리하는 DTO인 경우, Presentation Layer에서 최적화 실현
(예. Swagger 자동화)
- 단점
 - 코드 복제의 부담
 - 객체 추가에 따른 코드 분산의 어려움
 - DTO <-> Entity 매핑
- [참고] DTO와 Entity 변환 오픈 소스
 - ModelMapper : <http://modelmapper.org/getting-started/>
 - Mapstruct : <https://mapstruct.org/>

Step 10. 모듈 순환 참조

순환 참조 시 어떤 문제점이 존재하나?

- 기능 분리가 어려워서 모듈 재사용이 떨어진다.
- 기능 수정 시 참조하고 있는 모듈에서 부작용이 발생한다.
- 기능이 명확하게 분리되어 있지 않아서, 유지보수가 어렵다.

• Example

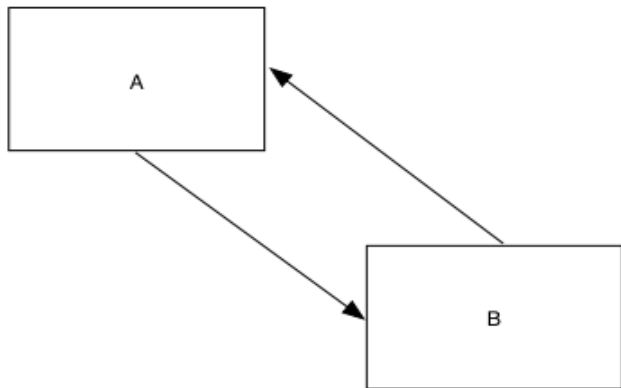


Step 10. 모듈 순환 참조

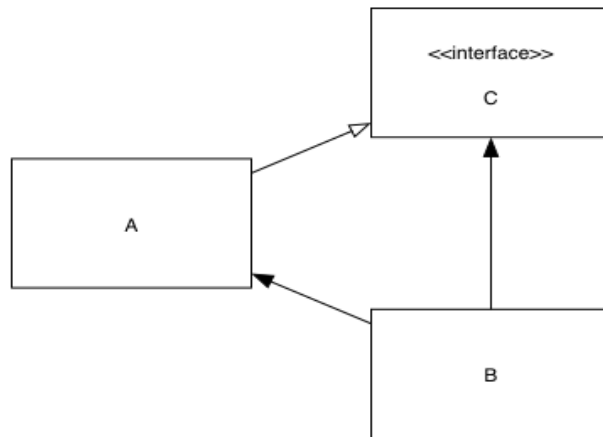
양방향 -> 단방향으로 변경하여, 사이클이 생기는 지점을 끊는 것이 핵심

[방법 1] 인터페이스 도출을 통한 사이클 제거

1. B의 메소드를 선언한 C 인터페이스는 만든다.
2. B는 C의 구현체로 한다.
3. A는 C를 의존한다.
4. B가 참조하는 A의 메소드는 유지한다.



[AS-IS]



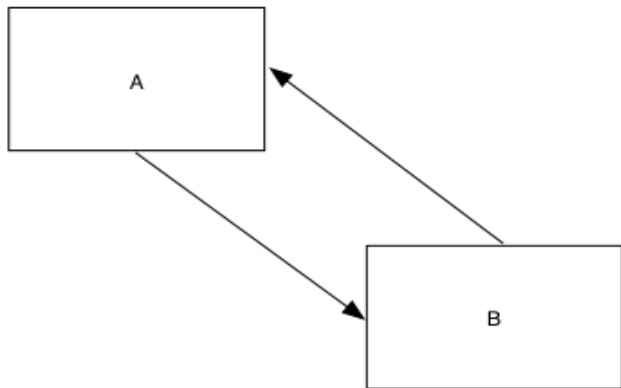
[TO-BE]

Step 10. 모듈 순환 참조

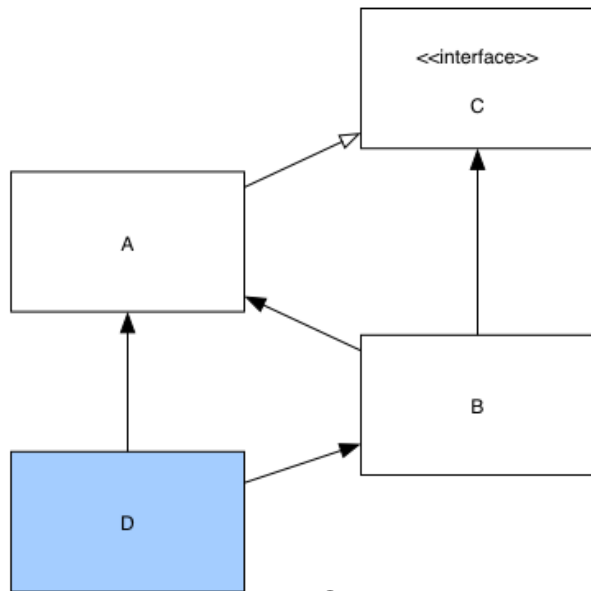
비즈니스가 복잡하면 A가 B를 참조해야 하는 경우가 생기기 때문에, 인터페이스 추상화로 문제를 해결할 수 없다.

[방법 2] 객체간의 기능 이동

1. D라는 새로운 클래스를 생성한다.
2. D가 A의 메소드를 참조한다.
3. D가 B의 메소드를 참조한다.
4. 기존에 A를 호출하는 부분은 D로 변경한다.



[AS-IS]



[TO-BE]

Step 10. 모듈 순환 참조

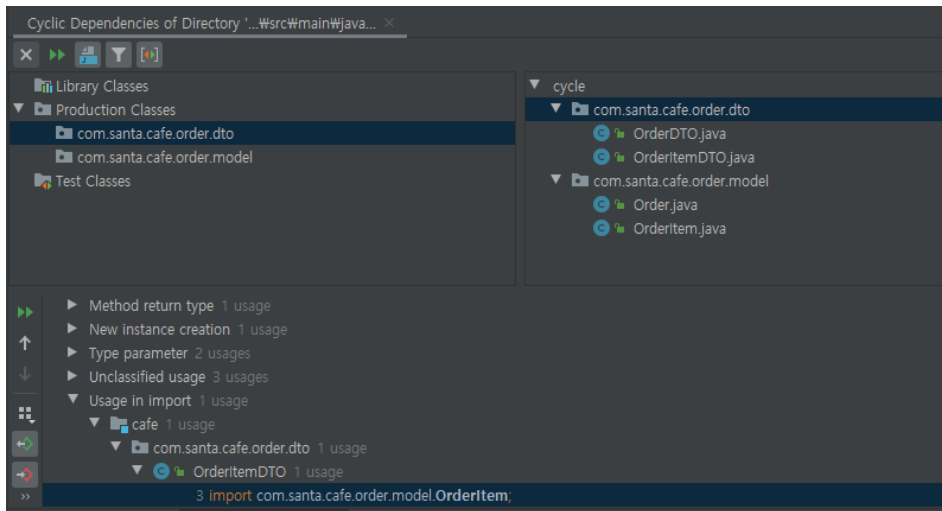
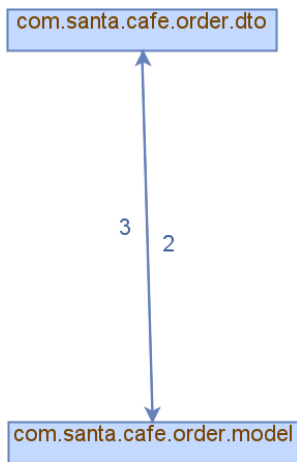
순환 참조는 **패키지 기준**으로 모듈 간의 사이클이 생기는지 확인한다.

IntelliJ의 Analyze 기능(Analyze Cyclic Dependencies)을 사용하여 어떻게 순환참조가 발생하는지 확인할 수 있다.

[목표]

현재 **DTO <-> Entity**로 변환하는 부분에서 순환 참조가 발생한다.

“객체간의 기능 이동” 방법을 사용하여 순환 참조를 해결하자.



Step 10. 모듈 순환 참조

Entity와 DTO를 서로 변환하는 역할을 ModelMapper 객체를 만들어서 관리하자.

- 패키지 : com.santa.cafe.order.mapper
- OrderModelMapper 신규 클래스 생성
- Order 객체에서 toDTO 메소드를 OrderModelMapper로 이동한다.

```
@Entity
@Table(name = "\"ORDER\"")
public class Order {

    public static OrderDTO toDTO(Order order) {
        OrderDTO orderDTO = new OrderDTO();
        orderDTO.setId(order.getId());
        return orderDTO;
    }
    ...
}
```

Order

```
public class OrderModelMapper {

    public static OrderDTO toDTO(Order order) {
        OrderDTO orderDTO = new OrderDTO();
        orderDTO.setId(order.getId());
        return orderDTO;
    }
}
```

OrderModelMapper

Step 10. 모듈 순환 참조

Entity와 DTO를 서로 변환하는 역할을 ModelMapper 객체를 만들어서 관리하자.

- 패키지 : com.santa.cafe.order.mapper
- OrderItemModelMapper 신규 클래스 생성
- OrderItemDTO 객체에서 toEntity 메소드를 OrderItemModelMapper로 이동한다.

```
public class OrderItemDTO {  
    ...  
    public static OrderItem toEntity(OrderItemDTO orderItemDTO) {  
        OrderItem orderItem = new OrderItem();  
        orderItem.setBeverageId(orderItemDTO.getBeverageId());  
        orderItem.setCount(orderItemDTO.getCount());  
        return orderItem;  
    }  
  
    public static List<OrderItem> toEntity(List<OrderItemDTO> orderItemDTOList) {  
        List<OrderItem> orderItems = new ArrayList<>();  
        for (OrderItemDTO orderItemDTO : orderItemDTOList) {  
            OrderItem orderItem = OrderItemDTO.toEntity(orderItemDTO);  
            orderItems.add(orderItem);  
        }  
  
        return orderItems;  
    }  
    ...  
}
```

OrderItemDTO

```
public class OrderItemModelMapper {  
    public static OrderItem toEntity(OrderItemDTO orderItemDTO) {  
        OrderItem orderItem = new OrderItem();  
        orderItem.setBeverageId(orderItemDTO.getBeverageId());  
        orderItem.setCount(orderItemDTO.getCount());  
        return orderItem;  
    }  
  
    public static List<OrderItem> toEntity(List<OrderItemDTO> orderItemDTOList) {  
        List<OrderItem> orderItems = new ArrayList<>();  
        for (OrderItemDTO orderItemDTO : orderItemDTOList) {  
            OrderItem orderItem = OrderItemModelMapper.toEntity(orderItemDTO);  
            orderItems.add(orderItem);  
        }  
  
        return orderItems;  
    }  
}
```

OrderItemModelMapper

Step 10. 모듈 순환 참조

OrderController에서 modelMapper를 사용하여 Entity와 DTO 변환을 하도록 변경하자.

- 변경 전 : Entity나 DTO 내부에 변환을 위한 static 메소드로 존재
- 변경 후 : DTO가 존재하는 Entity별로 modelMapper를 생성하여, Entity와 DTO 변환 메소드 관리

OrderController
변경 전

```
@PostMapping
public OrderDTO create(@RequestBody OrderDTO order) {
    return Order.toDTO(orderService.create(order.getCustomerId(),
        OrderItemDTO.toEntity(order.getOrderItems()),
        order.getPayment()));
}
```



OrderController
변경 후

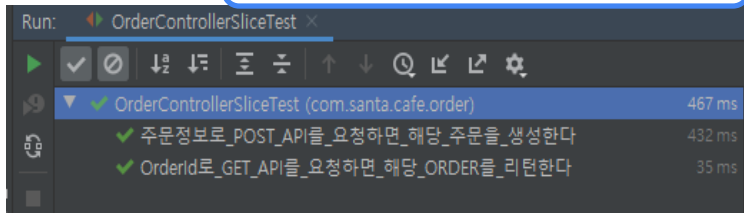
```
@PostMapping
public OrderDTO create(@RequestBody OrderDTO order) {
    return OrderModelMapper.toDTO(orderService.create(order.getCustomerId(),
        OrderItemModelMapper.toEntity(order.getOrderItems()),
        order.getPayment()));
}
```


Step 10. 모듈 순환 참조

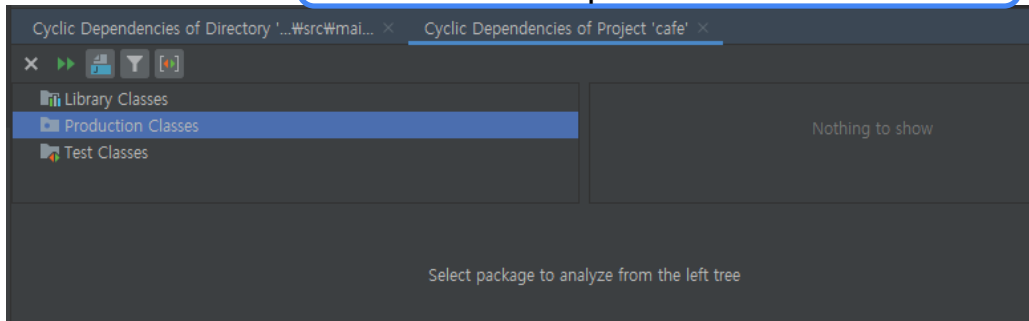
리팩토링 후에 테스트(OrderControllerSliceTest)가 정상적으로 통과하는 것을 통해서 기존 기능이 제대로 동작하는 것을 보장한다.

IntelliJ IDE 툴의 코드 분석을 통해서 기존에 존재하는 모듈 순환 참조 문제가 해결된 것을 확인할 수 있다.

OrderControllerSliceTest



Analyze -> Analyze Cyclic Dependencies



6. 리팩토링 결과 확인하기

현재 코드가 유지보수하기 좋은 최종 Clean code 인가?

- Long Method의 길이가 줄어들고, 가독성이 좋은 코드로 개선되었다.
- 하지만, 여전히 **Code smell**은 존재하며, **지속적인 개선**이 필요하다.

OrderService.create

```
@Transactional
public Order create(int customerId, List<OrderItem> orderItemList, int payment) {
    Customer customer = customerService.getCustomer(customerId);
    Order order = new Order();
    order.setStatus(OrderStatus.WAITING);
    order.setCustomer(customer);
    order.setPayment(payment);

    List<OrderItem> orderItems = getOrderItems(orderItemList, order);
    order.setTotalCost(getDiscountedTotalCost(getTotalCost(orderItems)));
    order.setMileagePoint(paymentService.getMileagePoint(payment, order.getTotalCost()));

    paymentService.pay(customerId, payment, order, order.getMileagePoint());

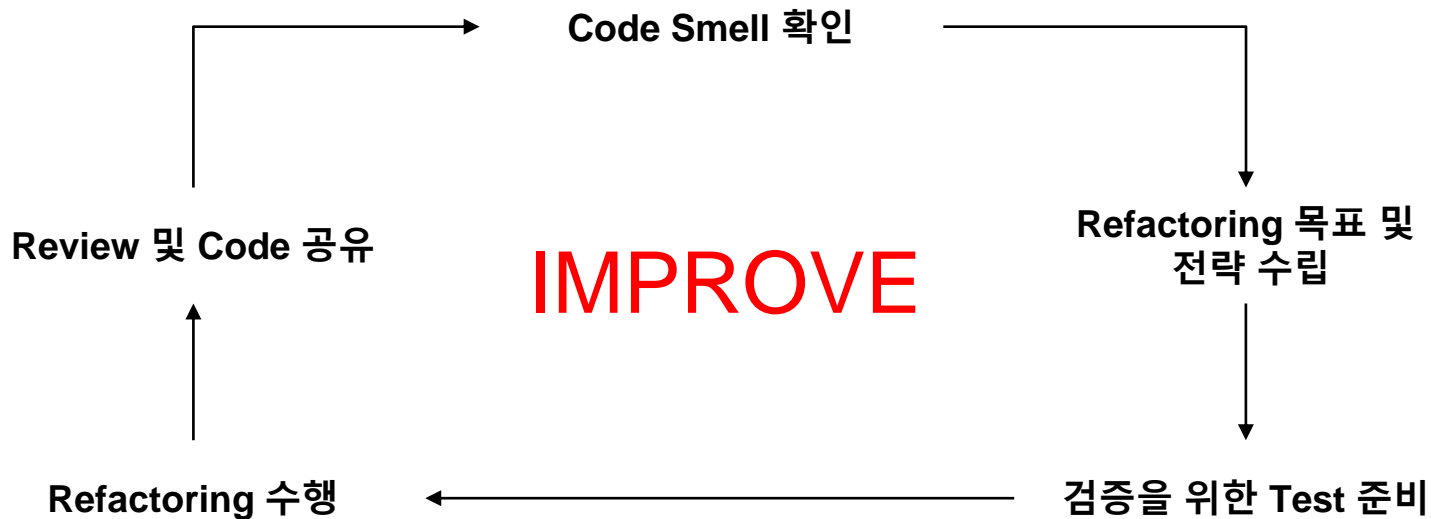
    orderRepository.save(order);
    orderItemRepository.saveAll(orderItems);

    return order;
}
```

- Naming : 메소드
- 객체 생성 Builder pattern 적용
- Extract method : 비즈니스 로직

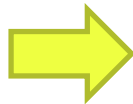
“리팩토링 이제 시작입니다.”

지속적인 개선으로써 Refactoring



Refactoring 시도해보기

- 좋은 Naming 습관을 통한 객체의 단순화
- 코드 중복 제거
- 생성자를 통한 의존성 주입 사용
- 메소드 및 객체의 단순화
- Interface 사용
- 단위 테스트 작성
- 테스트 또한 유지보수의 대상임을 인지



“**유지보수성을 높이기** 위해

코드 스멜을 인지하고,

소스를 개선하는 과정을

반복하는 것이

중요합니다.”

Q & A

코칭팀과 협업이 필요하면
언제든지 ACT로 연락주세요.

ACT 그룹 애자일 코칭팀

김경애 프로 (kyung50.kim@samsung.com)

김용진 프로 (yjn.kim@samsung.com)

감사합니다.

[참고] Step 7. Builder Pattern

Order 객체를 생성하는 부분을 좀 더 정리해 볼 수 있을까?

- Order 객체에 변수가 많아서 setter가 너무 많이 호출된다.
- Order 객체를 만드는 일반적인 new 사용은 create 메소드의 목적을 희석한다

```
@Transactional
public Order create(int customerId, List<Map<String, Object>> orderItemList, int payment) {
    Customer customer = customerService.getCustomer(customerId);
    Order order = new Order();
    order.setStatus(OrderStatus.WAITING);
    order.setCustomer(customer);
    order.setPayment(payment);

    List<OrderItem> orderItems = getOrderItems(orderItemList, order);
    order.setTotalCost(this.getDiscountedTotalCost(getTotalCost(orderItems)));
}
```

[참고] Step 7. Builder Pattern

객체를 생성하기 위한 다양한 패턴

- **Telescoping Constructor Pattern vs JavaBeans Pattern vs Builder Pattern**

```
Order order = new Order(payment, OrderStatus.WAITING, customer);
```

Telescoping Constructor Pattern

- 매개변수를 전달하여 생성자만을 사용한 객체 생성
- 쉽게 객체를 생성할 수 있지만, 파라미터 구성에 따라 생성자를 만들어줘야 함

```
Order order = new Order();  
order.setStatus(OrderStatus.WAITING);  
order.setCustomer(customer);  
order.setPayment(payment);
```

JavaBeans Pattern

- 생성자로 객체 생성 후 setter함수 사용하여 값 설정
- 객체를 불변으로 만들 수 없고, Thread 안정성을 확보하기 어려움

```
Order order = Order.builder()  
    .status(OrderStatus.WAITING)  
    .customer(customer)  
    .payment(payment)  
    .build();
```

Builder Pattern

- Build 함수를 통해 원하는 객체 생성
- 위 두개 패턴의 장점을 겸비한 객체 생성 패턴
- **가독성 및 객체 생성의 표현성을 높임**

[참고] Step 7. Builder Pattern

Builder Pattern을 직접 만들어보자.

- Order 클래스에 필요한 생성자를 추가한다.
- OrderBuilder 클래스를 만들고 객체 build를 위한 기능을 구현한다.
- OrderService의 객체 생성 부분을 수정한다.

Order

```
public Order(int id, double totalCost, double mileagePoint,
            int payment, OrderStatus status, Customer customer) {
    this.id = id;
    this.totalCost = totalCost;
    this.mileagePoint = mileagePoint;
    this.payment = payment;
    this.status = status;
    this.customer = customer;
}
```

@Transactional

```
public Order create(int customerId, List<Map<String, Object>> orderItemList, int payment) {
    Customer customer = customerService.getCustomer(customerId);
```

```
    OrderBuilder builder = new OrderBuilder();
    Order order = builder
        .customer(customer)
        .status(OrderStatus.WAITING)
        .payment(payment)
        .build();
```

```
    List<OrderItem> orderItems = getOrderItems(orderItemList);
```

OrderService.create()

OrderBuilder

```
public class OrderBuilder {
    private int id;
    ...
    private Customer customer;

    public OrderBuilder() {
    }

    public OrderBuilder id(double id) {
        this.id = id;
        return this;
    }
    ...
    public OrderBuilder customer(Customer customer) {
        this.customer = customer;
        return this;
    }

    public Order build() {
        return new Order(totalCost, mileagePoint, payment, status,
            customer);
    }
}
```

[참고] Step 7. Builder Pattern

Lombok 라이브러리는 객체에 Builder Pattern을 적용하기 최적의 툴이다.

- @Builder annotation을 사용하여 간편하게 Builder Pattern을 적용할 수 있다.

```
@Entity
@Table(name = "\"ORDER\"")
@Builder
public class Order {
    @Id
    @GeneratedValue
    private int id;

    @Column
    private double totalCost;
    ...
}
```

```
Order order = new Order();
order.setStatus(OrderStatus.WAITING);
order.setCustomer(customer);
order.setPayment(payment);
```

```
Order order = Order.builder()
    .status(OrderStatus.WAITING)
    .customer(customer)
    .payment(payment)
    .build();
```

[참고] Step 7. Builder Pattern

이외에도 Lombok은 다음과 같은 다양한 기능을 제공하고 있다.
Lombok을 사용하면 소스코드를 좀 더 깔끔한 상태로 유지할 수 있다.

@Getter @Setter

- 필드값에 대한 getters/setters 메소드가 자동 생성된다.

@ToString

- toString 메소드가 자동 생성된다.

@EqualsAndHashCode

- equals와 hashCode 메소드가 자동 생성된다.

@Data

- @Getter, @Setter, @NonNull, @EqualsAndHashCode, @ToString 에 대한 걸 모두 해주는 Annotation

[참고] Step 7. Builder Pattern

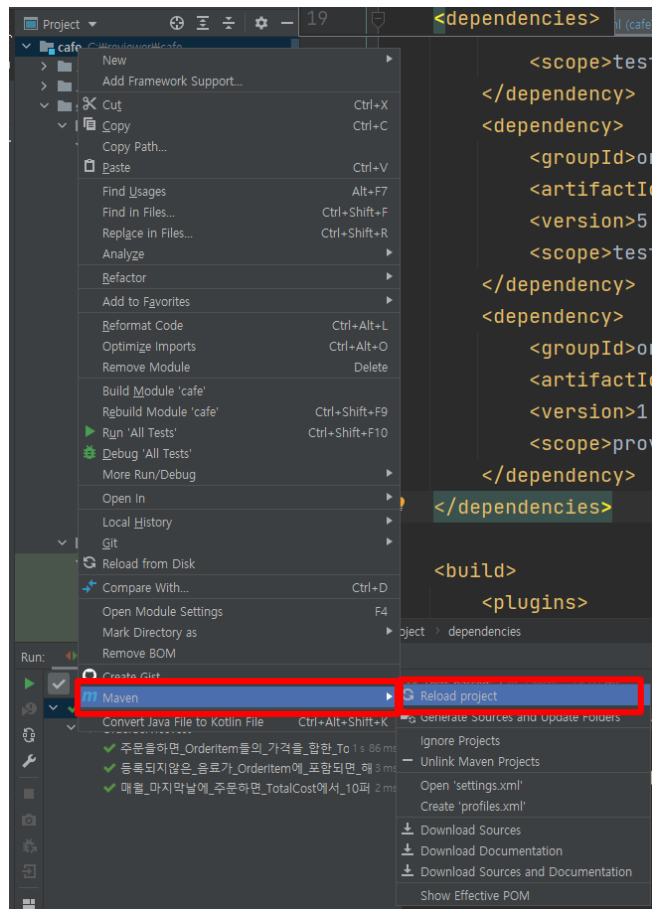
Lombok을 설치해보자.

- pom.xml에 필요한 dependency를 추가하자.
- Maven > Reload project를 통해 라이브러리를 갱신한다.

pom.xml

```
<dependency>
  <groupId>org.projectlombok</groupId>
  <artifactId>lombok</artifactId>
  <version>1.18.20</version>
  <scope>provided</scope>
</dependency>
```

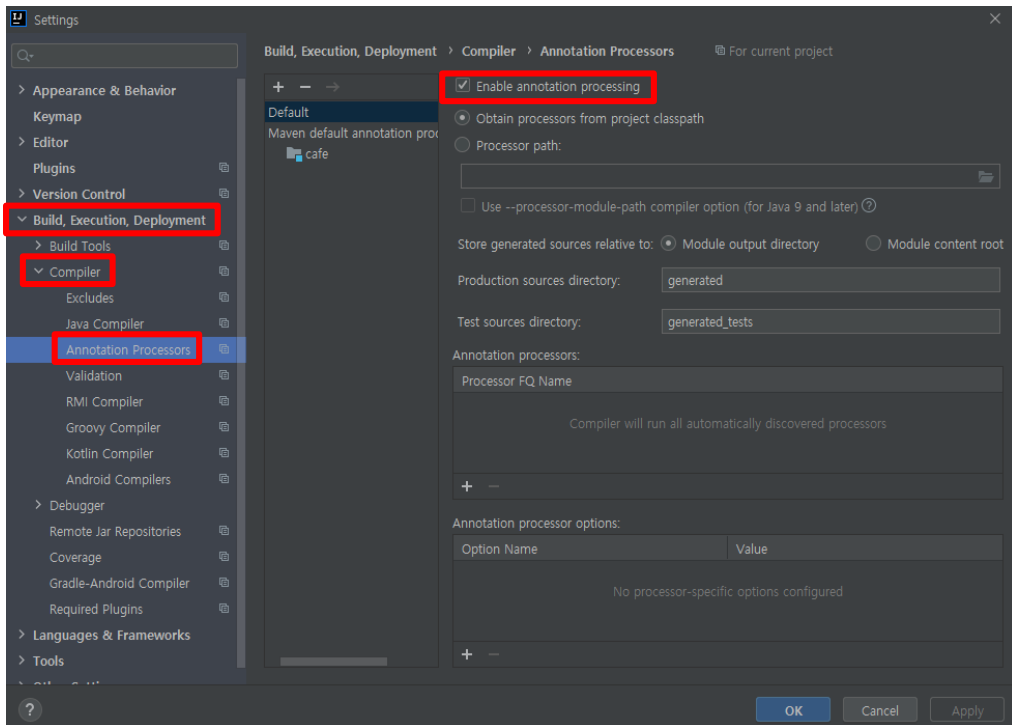
```
...
  <scope>test</scope>
</dependency>
<dependency>
  <groupId>org.junit.vintage</groupId>
  <artifactId>junit-vintage-engine</artifactId>
  <version>5.7.0</version>
  <scope>test</scope>
</dependency>
<dependency>
  <groupId>org.projectlombok</groupId>
  <artifactId>lombok</artifactId>
  <version>1.18.20</version>
  <scope>provided</scope>
</dependency>
</dependencies>
```



[참고] Step 7. Builder Pattern

IntelliJ에서 Lombok annotation을 처리할 수 있도록 다음과 같이 셋팅한다.

- Settings > Build, Execution, Deployment > Compiler > Annotation Processors 에서 [Enable annotation processing] 을 체크한다.



[참고] Step 7. Builder Pattern

Order 객체 생성하는 부분에 Builder Pattern을 적용해보자.

- Order 클래스에 @Builder Annotation을 추가한다.
- OrderService.create() 함수 내 Order 객체 생성 부분을 Builder Pattern으로 변경한다.
- @NoArgsConstructor, @AllArgsConstructor 사용하여 Order 생성자가 생략 가능하다.

```
@Entity
@Table(name = "ORDER")
@Builder
@NoArgsConstructor
@AllArgsConstructor
public class Order {
    ...
}
```

Order

```
@Transactional
public Order create(int customerId, List<Map<String, Object>> orderItemList) {
    Customer customer = customerService.getCustomer(customerId);

    Order order = new Order();
    order.setStatus(OrderStatus.WAITING);
    order.setCustomer(customer);
    order.setPayment(payment);

    List<OrderItem> orderItems = getOrderItems(orderItemList, order);
    order.setTotalCost(this.getDiscountedTotalCost(getTotalCost(orderItems)));
}
```

OrderService.craete

```
@Transactional
public Order create(int customerId, List<Map<String, Object>> orderItemList, int payment) {
    Customer customer = customerService.getCustomer(customerId);

    Order order = Order.builder()
        .customer(customer)
        .status(OrderStatus.WAITING)
        .payment(payment)
        .build();

    List<OrderItem> orderItems = getOrderItems(orderItemList, order);
    order.setTotalCost(this.getDiscountedTotalCost(getTotalCost(orderItems)));
}
```


[참고] Step 7. Builder Pattern

Order 객체를 생성하는 테스트 코드에도 Builder Pattern을 이용하도록 수정하자.

CashPaymentServiceTest

```
@Test
public void 마일리지로_결제하지않는경우_마일리지를_적립한다() {
    //given
    Order order = Order.builder()
        .totalCost(2000.0)
        .build();

    //when
    subject.pay(CUSTOMER_ID, order, 2
    ...
```

CardPaymentServiceTest

```
@Test
public void 마일리지로_결제하지않는경우_마일리지를_적립한다() {
    //given
    Order order = Order.builder()
        .totalCost(2000.0)
        .build();

    //when
    subject.pay(CUSTOMER_ID, order, 2
    ...
```

MileagePaymentServiceTest

```
@Test
public void 마일리지로_결제하는경우_마일리지를_적립하지_않는다() {
    //given
    Order order = Order.builder()
        .totalCost(2000.0)
        .build();
    when(mockMileageApiService.getMileages(CUSTOMER_ID)).thenReturn(3000);

    //when
    subject.pay(CUSTOMER_ID, order, 0.0);
    ...
```

[참고] Step 7. Builder Pattern

Lombok의 다른 기능들도 사용해볼까요?

```
@Entity
@Table(name = "\"ORDER\"")
@Builder
...
public class Order {
    ...
    public int getId() {
        return id;
    }
    ...
    public Customer getCustomer() {
        return customer;
    }
    ...
    public void addOrderItems(List<OrderItem> orderItems) {
        this.orderItems.addAll(orderItems);
    }
    @Override
    public boolean equals(Object o) {
        ...
    }
    @Override
    public int hashCode() {
        return Objects.hash(id, totalCost, status);
    }
    @Override
    public String toString() {
        ...
    }
    ...
    public double getMileagePoint() {
        return mileagePoint;
    }
}
```

Order

```
@Entity
@Table(name = "\"ORDER\"")
@Builder
...
@Data
public class Order {
    ...
    public void addOrderItems(List<OrderItem> orderItems) {
        this.orderItems.addAll(orderItems);
    }
    ...
    @Override
    public boolean equals(Object o) {
        if (this == o) return true;
        if (o == null || getClass() != o.getClass()) return false;
        Order order = (Order) o;
        return id == order.id &&
            totalCost == order.totalCost &&
            status == order.status;
    }
}
```

- @Data Annotation 추가
- Getter/Setter 메소드 삭제
- hashCode, toString 메소드 삭제
- equals 메소드는 로직이 있으므로 유지

[참고] Step 7. Builder Pattern

Refactoring을 통해서

- Long Method의 길이가 줄어들었다.
- 가독성이 좋아져서 유지보수하기 용이한 코드로 변경되었다.
- 최종적인 Clean code는 아니며, 지속적인 개선이 필요하다.

OrderService.craete

```
@Transactional
public Order create(int customerId, List<Map<String, Object>> orderItemList, int payment) {
    Customer customer = customerService.getCustomer(customerId);

    Order order = Order.builder()
        .customer(customer)
        .status(OrderStatus.WAITING)
        .payment(payment)
        .build();

    IPaymentService service = paymentServiceFactory.getService(PaymentType.fromCode(payment));

    List<OrderItem> orderItems = getOrderItems(orderItemList, order);
    order.setTotalCost(this.getDiscountedTotalCost(getTotalCost(orderItems)));
    order.setMileagePoint(service.getMileagePoint(order.getTotalCost()));
    service.pay(customerId, order, order.getMileagePoint());

    orderRepository.save(order);
    orderItemRepository.saveAll(orderItems);

    return order;
}
```