Refactory 기록

@Transactional(rollbackOn = RuntimeException.class)  
public Order createOrder(@Valid Account account, //*TODO: DTO 로 변경 예정* Map<@Valid Game, @Valid Optional<Coupon>> gameCouponMap, PaymentMethod method,  
 int point) {  
 if (!(method == PaymentMethod.*CARD* || method == PaymentMethod.*KAKAOPAY*)) {  
 throw new RuntimeException(); // 잘못된 결제 수단  
 }  
  
 List<Payment> payments = new ArrayList<>();  
 /\*List<OrdersGames> ordersGamesList = new ArrayList<>();  
 List<OrdersPayments> ordersPaymentsList = new ArrayList<>();\*/  
 List<CouponPayment.CouponPaymentBuilder> couponPaymentBuilders = new ArrayList<>();  
 List<DiscountPayment.DiscountPaymentBuilder> discountPaymentBuilders = new ArrayList<>();  
 PointPayment.PointPaymentBuilder pointPaymentBuilder = null;  
 int totalPrice = 0;  
 int remainTotalPrice = 0;  
 for (var gameCouponEntry : gameCouponMap.entrySet()) {  
 Game game = gameCouponEntry.getKey();  
 Optional<Coupon> couponOptional = gameCouponEntry.getValue();  
  
 totalPrice += game.getPrice();  
 int remainPrice = game.getPrice();  
 int discountPrice = (int) (remainPrice - Math.*round*(remainPrice \* (1 - game.getDiscount())));  
  
 if (discountPrice != 0) {  
 discountPaymentBuilders.add(  
 DiscountPayment.*builder*().game(game).price(discountPrice).status(PaymentStatus.*PENDING*));  
 remainPrice -= discountPrice;  
 }  
  
 if (couponOptional.isPresent()) {  
 Coupon coupon = couponOptional.get();  
 int couponPrice = -1;  
 if (coupon.getMinFulfillPrice() < remainPrice) {  
 throw new RuntimeException(); // 쿠폰 최소 충족 금액 : 충족 안함 다시 해야함  
 }  
 if (coupon.getMethod() == CouponMethod.*PERCENT*) { // improve: 전략패턴  
 couponPrice = (int) (remainPrice \* coupon.getAmount());  
 couponPrice = (int) Math.*min*(coupon.getMaxDiscountPrice(),  
 Math.*max*(coupon.getMinDiscountPrice(), couponPrice));  
 } else if (coupon.getMethod() == CouponMethod.*STATIC*) {  
 couponPrice = (int) coupon.getAmount();  
 }  
 couponPaymentBuilders.add(  
 CouponPayment.*builder*().game(game).coupon(coupon).price(couponPrice).status(PaymentStatus.*PENDING*));  
 }  
 remainTotalPrice += remainPrice;  
 }  
 if (remainTotalPrice != 0) {  
 point = Math.*min*(point, remainTotalPrice);  
 if (account.getPoint() < point) {  
 throw new RuntimeException(); // Wrong Point 포인트 부족  
 }  
 remainTotalPrice -= point;  
 pointPaymentBuilder = PointPayment.*builder*().price(point).status(PaymentStatus.*PENDING*);  
 }  
  
 Order order;  
 PaymentStatus paymentStatus = PaymentStatus.*PENDING*;  
 OrderStatus orderStatus = OrderStatus.*PENDING*;  
 if (remainTotalPrice == 0) { // 무료 빠른 주문 완성  
 paymentStatus = PaymentStatus.*SUCCESS*;  
 orderStatus = OrderStatus.*SUCCESS*;  
 }  
  
 for (var couponPaymentBuilder : couponPaymentBuilders) {  
 CouponPayment couponPayment = couponPaymentBuilder.status(paymentStatus).build();  
 couponPaymentRepository.save(couponPayment);  
 payments.add(couponPayment);  
 }  
 for (var discountPaymentBuilder : discountPaymentBuilders) {  
 DiscountPayment discountPayment = discountPaymentBuilder.status(paymentStatus).build();  
 discountPaymentRepository.save(discountPayment);  
 payments.add(discountPayment);  
 }  
 if (pointPaymentBuilder != null) {  
 PointPayment pointPayment = pointPaymentBuilder.status(paymentStatus).build();  
 pointPaymentRepository.save(pointPayment);  
 payments.add(pointPayment);  
 }  
  
 order = Order.*builder*().account(account).totalPrice(totalPrice).status(orderStatus).build();  
  
 if(remainTotalPrice != 0) {  
 if(method == PaymentMethod.*CARD*) {  
  
 }  
 else if(method == PaymentMethod.*KAKAOPAY*) {  
 KakaopayPayment.KakaopayPaymentBuilder kakaopayPaymentBuilder = KakaopayPayment.*builder*();  
 KakaopayPayment kakaopayPayment = kakaopayPaymentBuilder.price(remainTotalPrice).status(PaymentStatus.*PENDING*).build();  
 kakaopayPaymentRepository.save(kakaopayPayment);  
 }  
 }  
 for (var gameCouponEntry : gameCouponMap.entrySet()) {  
 Game game = gameCouponEntry.getKey();  
 ordersGamesRepository.save(OrdersGames.*builder*().game(game).order(order).build());  
  
 }  
 for (var payment : payments) {  
 ordersPaymentsRepository.save(OrdersPayments.*builder*().order(order).payment(payment).build());  
 }  
 return order;  
}

기존의 Order 생성 코드

모든 코드가 하나의 의존성이고 강하게 결합 되어있다.

* 수정 하려면 모든 코드의 의도를 알아야함
* 에러 발생시 모든 코드를 알아야 함
* 기능 추가 시 코드를 변경해야함

Refacory!

Payment 생성 하는 부분이 서로 연관이 없고 독립적으로 작동하는 것을 발견!

또한 Payment 생성하는 부분서 앞선 Payment 생성 후 남은 금액만이 계승되는 것을 발견

따라서

순차적인 PaymentFactory를 만드는 것을 목표로 Refactory

기대효과 :

추후 새로운 Payment 추가 시 코드의 변경 없이 확장 가능

로직들의 가독성 증가

순서를 로직으로 간편하게 볼 수 있음

결과

@Transactional(rollbackOn = RuntimeException.class)  
public Order createOrder(@Valid Account account, //*TODO: DTO 로 변경 예정* Map<@Valid Game, @Valid Optional<Coupon>> gameCouponMap, List<PaymentMethod> methods,  
 int point) {  
 int totalPrice = gameCouponMap.entrySet().stream()  
 .mapToInt(entry -> entry.getKey().getPrice()).reduce(0, (ingPrice, v) -> ingPrice + v);  
 List<Game> games = gameCouponMap.entrySet().stream().map(entry -> entry.getKey()).toList();  
 Order order = Order.*builder*()  
 .account(account)  
 .totalPrice(totalPrice)  
 .status(OrderStatus.*PENDING*)  
 .build();  
  
 List<Payment> payments = new ArrayList<>();  
 int totalRemainPrice = 0;  
 for (var game : games) {  
 Optional<Coupon> coupon = gameCouponMap.get(game);  
 SingleGameProductContext singleGameProductContext = SingleGameProductContext.*builder*()  
 .game(game)  
 .coupon(coupon == null ? Optional.*empty*() : coupon)  
 .account(account)  
 .remainPrice(game.getPrice())  
 .build();  
 for (var singleProductPaymentFactory : singleProductPaymentFactories) {  
 singleProductPaymentFactory.create(singleGameProductContext)  
 .ifPresent(payment -> payments.add(payment));  
 }  
 totalRemainPrice += singleGameProductContext.getRemainPrice();  
 }  
 {  
 AllGameProductContext allGameProductContext = AllGameProductContext.*builder*()  
 .games(games)  
 .account(account)  
 .point(point)  
 .totalRemainPrice(totalRemainPrice)  
 .build();  
 for (var allProductPaymentFactory : allProductPaymentFactories) {  
 allProductPaymentFactory.create(allGameProductContext)  
 .ifPresent(payment -> payments.add(payment));  
 }  
 totalRemainPrice = allGameProductContext.getTotalRemainPrice();  
 }  
  
 if (totalRemainPrice == 0) {  
 throw new RuntimeException(); //ROLLBACk  
 }  
 for (var payment : payments) {  
 ordersPaymentsRepository.save(  
 OrdersPayments.*builder*().payment(payment).order(order).build());  
 }  
 for (var game : games) {  
 ordersGamesRepository.save(OrdersGames.*builder*().game(game).order(order).build());  
 }  
 return order;  
}

딱 봐도 뭘 하는 코드인지 쉽게 보인다.

@Component  
@org.springframework.core.annotation.Order(100)  
public class DiscountPaymentFactory implements SingleProductPaymentFactory { //전략 패턴  
  
 public PaymentMethod getTargetPaymentMethod() {  
 return PaymentMethod.*DISCOUNT*;  
 }  
  
 @Autowired  
 private PaymentRepository<DiscountPayment> discountPaymentRepository;  
  
 @Override  
 public Optional<Payment> create(SingleGameProductContext context) {  
 int discountPrice = (int) (  
 context.getRemainPrice() \* (1 - context.getGame().getDiscount()));  
 DiscountPayment discountPayment = DiscountPayment.*builder*().game(context.getGame())  
 .price(discountPrice).status(PaymentStatus.*SUCCESS*).build();  
 discountPaymentRepository.save(discountPayment);  
 context.setRemainPrice(context.getRemainPrice() - discountPrice);  
 return Optional.*of*(discountPayment);  
 }  
}

@Component  
@org.springframework.core.annotation.Order(100)  
public class PointPaymentFactory implements AllProductPaymentFactory {  
  
 @Autowired  
 private PaymentRepository<PointPayment> pointPaymentRepository;  
  
 public PaymentMethod getTargetPaymentMethod() {  
 return PaymentMethod.*POINT*;  
 }  
  
 @Override  
 public Optional<Payment> create(AllGameProductContext context) {  
 int totalRemainPrice = context.getTotalRemainPrice();  
 int pointPrice = context.getPoint();  
 if (totalRemainPrice > context.getPoint()) {  
 throw new RuntimeException();  
 }  
 PointPayment payment = PointPayment.*builder*().status(PaymentStatus.*SUCCESS*)  
 .price(pointPrice).build();  
 pointPaymentRepository.save(payment);  
 context.setTotalRemainPrice(totalRemainPrice - pointPrice);  
 return Optional.*of*(payment);  
 }  
  
}

2. 순환 의존성

KakaoPaymentService 와

OrderService 에서 서로 호출하는 의존성 발생

KakaoPaymentService 에서

KakaoAPI완료 후 결제상태 변경 할 때 Order 상태 변경 update 호출

OrderService 에서는 KakaoPayment 생성에서 사용

생성 의존성

KakaopayService <-> OrderService

해결방법

옵저버 패턴을 사용하여 update 로직 회피

결과

OrderService 가 KakaoService 보다 생성 우선순위를 가진다는 것 보다는

옵저버 패턴을 사용하여 Service 레이어가 옵저버 구조보다 우선순위 에 있다는 것을 규정하는 것이 좋고

또한 옵저버 패턴을 사용하여 update 를 함수에 종속되는것이아닌 클래스에 종속되게 하여 구조적 의미를 가지게 할 수있었다.

결론

PaymentService를 Subscribe 로 하는 OrderService 를 참조하는 Observer 생성

public interface Observer<C extends ObserverContext> {  
  
 void onUpdate(C context);  
}

public interface Subject<C extends ObserverContext, O extends Observer<C>> {  
  
 void registerObserver(O observer);  
  
 void unregisterObserver(O observer);  
  
 void notifyObservers(C context);  
}

orderService.onUpdate(orderId)

->

notifyObservers(OrderUpdateObserverContext.*builder*().payment(kakaopayPayment).build());

**확장성 증가**

* **새로운 옵저버 클래스 만들어서 확장**

**가독성 증가**

* **기존의 함수 군대 군대 숨어있는 옵저버를 클래스로 구조화하여 노출**

**또한 최종 생성 의존성**

OrderService <- KakaoService, Observer

KakaoService <- Observer

따라서 스프링 생성 의존성 순환을 해결