

디자인 패턴

전략패턴

디자인 패 턴??

프로그램을 설계할 때 발생했던 문제점들을 객체 간의 상호관계 등을 이용하여 해결할 수 있도록 하는 하나의 규약!

=> 내가 겪었던 문제들을 이미 먼저 겪은 사람들이 만들어 둔 관행

전략 패턴

객체의 행위를 바꾸고 싶은 경우 직접 수행하지 않고 **전략**이라고 부르는 캡슐화한 알고리즘을 컨텍스트 안에서 바꿔주면서 상호 교체함

*) 컨텍스트 : 상황, 맥락, 문맥을 의미하며 개발자가 어떠한 작업을 완료하는데 필요한 모든 관련 정보



어느날....
너무 가지고 싶었던 물건이 눈앞에 있다...
네이버 스토어에서 나는 네이버페이를 이용하게 되는데...



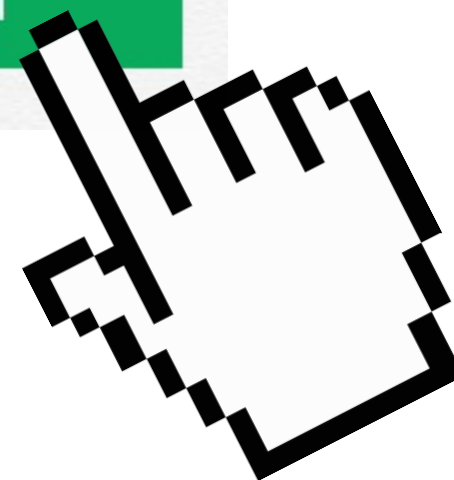
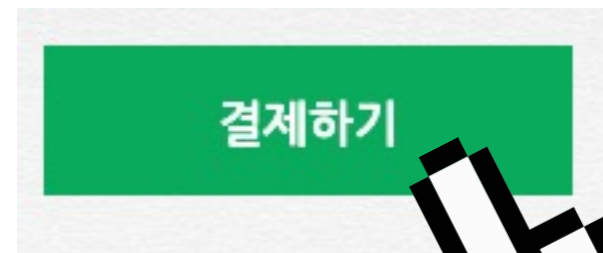
난 네이버페이, 나를 이용하는 방법은 여러 방법이 있지..


A screenshot of the Naver Pay payment method selection interface. At the top, there's a header '결제수단' (Payment Method) with a back arrow. Below it, three radio buttons are visible: '충전포인트 결제' (Recharge Point Payment) with a blue badge '최대 2.5% 적립' (Up to 2.5% discount), '계좌 간편결제' (Account Easy Payment), and '카드 간편결제' (Card Easy Payment). Under '충전포인트 결제', there's a green box showing '충전포인트 잔액' (Recharge Point Balance) as '4,480원'. Below these, three more radio buttons are shown: '일반결제' (General Payment) which is selected with a green dot, and two unselected ones. Under '일반결제', there are five buttons: '신용카드' (Credit Card) in green, '휴대폰' (Mobile Phone), '상품권' (Gift Certificate), 'T-money', and '문화누리카드' (Cultural Nuri Card). Below these buttons are two dropdown menus: '카드선택' (Card Selection) and '일시불' (Installment). At the bottom, there's a section '결제/할인혜택 안내' (Payment/Discussion Benefit Notice) with a note: '* 등록된 계좌/카드는 설정>계좌/카드 정보에서 확인 가능합니다.' (Registered account/card can be confirmed in Settings > Account/Card Information).

충전 포인트 결제
계좌 간편결제
카드 간편 결제
일반 결제...!



이미 눈이 돌아버린 저는.. 그냥 결제하기만 눌러서 결제를 진행하였습니다!





결제하기

여기서 결제 완료하는 상태로 변경시에는

네이버 포인트를 더 적립하기 위해서 **충전 포인트 결제**,

계좌에 바로 그 금액이 빠져나가는 걸 원하면 **계좌 간편결제**

신용카드를 써야할 때 **카드 간편 결제**

이렇게 다양한 전략 방식이 있고,

계좌에서 만원단위로 네이버 포인트로 충전해서 다시 그 금액중에서 결제하는..

이런 복잡한 행위는 충전 포인트 결제 전략으로 캡슐화된 알고리즘으로 존재해서 전 간단히 선택하고 결제합니다.

매번 결제 상황마다 다양한 전략 방식을 선택하여서 결제가 가능한 장점이 있쥬!

이걸.. 이제.. 실제.. 개발 환경에서 생각해 볼까요...?

전 물류회사 A사에서 일하는 엔지니어였어요...
어느날.. PM이 나에게 다가와서 말하더라구요...



PM

유니님, 유니님이 배송사에게 배송하는 프로세스 담당하시죠?

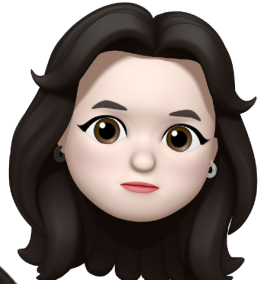
네! 혹시 에러가 발생했나요?



PM

아니요! 그건 아니고요... 다름이 아니라 저희 이제.. 일본 배송을 하기로 해서 ^^...
...그래서요...

네....(싸늘하다...)



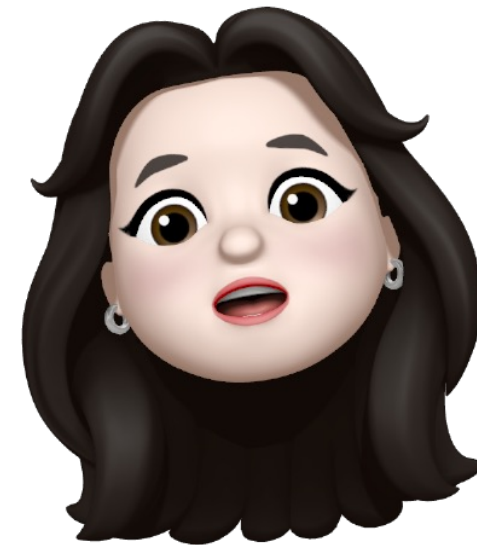
PM

일본 배송 프로세스가 필요할거 같아요
제가 전달 받기로는... 완전히 프로세스가 다르다고...
다음주중에... 미국도 진행해서.. 미국도 달라서 준비가 필요할거 같아요!!

(가슴에 비수가 날아와 쏘인다..)
ㅈㅈㅈㅈㅈ 왜 나라별 배송 프로세스가 다르죠??



그랬습니다.. 저는... 배송 프로세스 로직 하나만 믿고 있었던 월.루 개발자였습니다!!
그런데 이제.. 해외 배송들이 추가되면서 각 나라별로
배송사에게 전달하는 프로세스를 다르게 뒤야하는 상황에 놓였습니다..



```
class 발송 {  
  발송상품  
  
  발송(상품) {  
    발송상품 = 상품  
  }  
  
  택배사에게발송 = () => {}  
  
  발송취소 = () => {}  
  
  쿼드로보내기 = () => {}  
}
```

여기 제가 월루했던 코드입니다...
여기서 택배사에게 발송시 이제 국가 코드에 따라 택배사에게 발송이
달라져야 해요!!



흐음...전략 패턴을 이용해보야지!

일단 바뀌는 부분과 바뀌지 않는 부분을 분리해야지..!

바뀌는 부분은

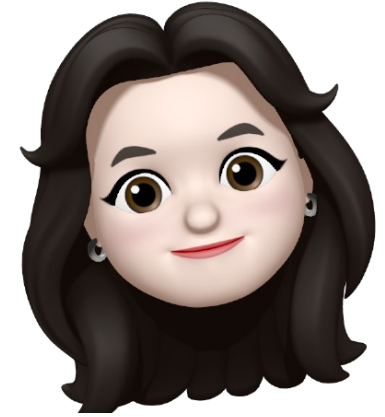
```
택배사에게발송 = ( ) => { }
```

이지!

또, 구현이 아닌 인터페이스에 맞춰서 프로그래밍을 해야혀!

```
interface 택배사에게발송행위 {  
    택배사에게발송 ( ) : boolean  
}  
  
class 일본으로발송하기 implements 택배사에게발송행위 {  
    택배사에게발송 = ( ) => {  
        //일본택배사에게 배송  
    }  
}
```

그러면... 이걸 이제 기존 발송 클래스와는 어떻게 사용해야할까?
상속이 아닌 두 클래스가 합치는 구성으로 진행해야겠지!



```
class 발송 {  
    발송상품  
    택배사에게발송행 : 택배사에게발송행위  
  
    발송(상품) {  
        발송상품 = 상품  
    }  
  
    택배사에게발송 = () => {  
        택배사에게발송행.택배사에게발송()  
    }  
}
```

```

class 국내발송 extends 발송 {

  발송(상품) {
    발송상품 = 상품
    택배사에게발송행 = new 국내택배사에게발송하기()
  }

  택배사에게발송 = () => {
    택배사에게발송행.택배사에게발송()
  }
}

class 일본발송 extends 발송 {

  발송(상품) {
    발송상품 = 상품
    택배사에게발송행 = new 일본으로발송하기()
  }

  택배사에게발송 = () => {
    택배사에게발송행.택배사에게발송()
  }
}

=====
let 발송Obj
if(상품.nationCode == 'KR') {
  발송Obj = new 국내발송()
}else {
  발송Obj = new 일본발송()
}

-----프로세스 -----

발송Obj.택배사에게로()

```

이렇게 하면 앞으로 중간에 발송Obj를 선언시
분기 시켜주는 부분만 계속 추가시켜주면된당!

이렇게 저는 문제를 해결했습니다!

중간에 **택배사에게발송행**의 setter함수를 만들게되면,
동적으로 행위를 지정할 수 있습니다!