2️⃣

개방-폐쇄의 원칙 (OCP - Open-Closed Principle)

정의

> 소프트웨어 요소는 확장에는 열려 있으나 변경에는 닫혀 있어야 한다.

확장이란?

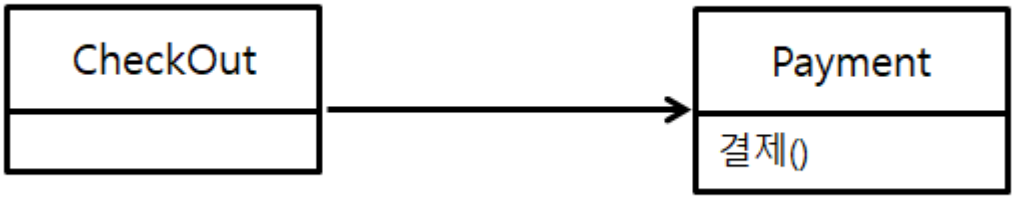
💡 새로운 타입을 추가하여 extends 하는 것을 확장이라고 하며, 단순히 메소드의 기능이 늘어나는 것을 확장이라고 하지 않는다. 확장을 위해 소스코드를 수정해야 한다면 Fragility가 발생하기 때문에 코드에 악취가 발생하게 된다.

변경이란?

💡 확장된 타입에 대한 코드 수정을 변경이라고 하는 것이 아닌, 새로운 타입을 활장할 시 상위 타입의 변경이 없는 것을 의미한다.

OCP를 준수하지 않는 경우 발생할 수 있는 문제점

변경에 취약한 코드가 된다



💡 CheckOut에서 Payment의 결제 메소드를 호출하는 코드를 호출하는 것으로 기능이 구현되어 있다고 가정한다.

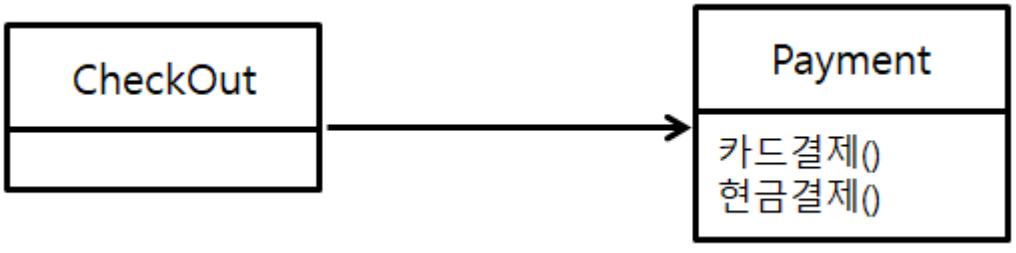
`CheckOut`

```
public class CheckOut {
    public void paymentProduct(Payment payment) {
        payment.결제();
    }
}
```

`Payment`

```
public class Payment {
    public void 결제() {
        System.out.println("결제를 진행합니다.");
    }
}
```

메소드 하나만 추가했을 뿐인데 클라이언트의 코드도 변경해야 한다



💡 요구사항이 변경되어 결제를 카드 결제와 현금 결제가 다른 프로세스로 동작하도록 변경해달라고 요청이 들어오게 되었다. 가장 쉽게 고려할 수 있는 부분은 Payment 메소드에 결제() 메소드를 카드결제() 메소드와 현금결제() 메소드로 나누어서 작성하는 것이다.

`Payment`

```
public class Payment {
    public void 카드결제() {
        System.out.println("카드 결제를 진행합니다.");
    }

    public void 현금결제() {
        System.out.println("현금 결제를 진행합니다.");
    }
}
```

💡 손쉽게 메소드를 하나 추가해서 카드 결제와 현금 결제를 진행하도록 변경했더니 Payment에 의존하는 CheckOut 클래스에 컴파일 에러가 발생을 했다. 더 이상 결제 메소드는 존재하지 않기 때문이다. 이처럼 다른 클래스에도 코드 수정의 영향을 주는 것을 Fragility가 있다고 한다.

`CheckOut`

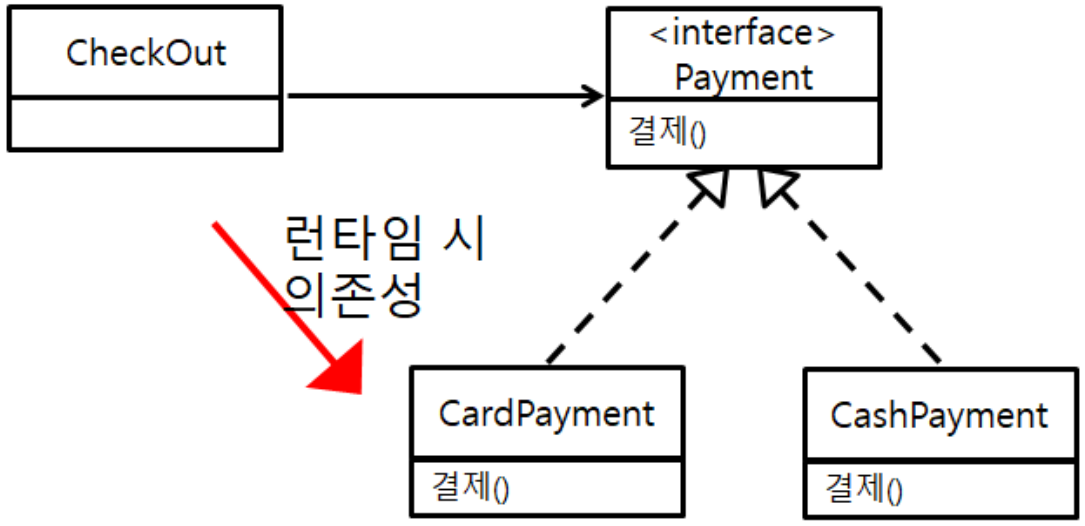
```
public class CheckOut {
    public void paymentProduct(Payment payment) {
        payment.결제();   //컴파일 에러 발생
    }
}
```

`CheckOut 수정`

```
public void paymentProduct(Payment payment, String payMethod) {
    if(payMethod.equals("카드") {
        payment.카드결제();
    } else if(payMethod.equals("현금") {
        payment.현금결제();
    }
    ...
```

```
    //향후 추가되는 결제 방식이 존재하는 경우 매번 로직을 수정해야 한다.
}
```

해결 방법



컴파일 시에는 클래스에 직접 의존하지 않고 인터페이스에 의존하게 한 뒤 런타임 시 클래스에 의존하도록 한다. 즉 컴파일 의존성을 런타임 의존성으로 바꾸게 되면 확장에 열려있게 된다. 또 다른 결제 방식이 추가될 때 마다 확장으로 Payment 인터페이스만 구현하게 되면 손쉽게 결제 방식을 변경할 수 있으며 CheckOut에도 영향을 주지 않게 된다.
따라서 확장에 대해서는 개방되어 있으며, 변경에 의해서는 닫혀있게 작성할 수 있다.

한계점

이론적으로는 이상적 이지만 현실적으로는 적용 불가능 한 케이스가 많이 존재한다. 확장에 필요한 모든 것을 인터페이스로 두는 것을 예측하기 힘들 뿐 아니라, 예측한다 하더라도 현재의 요구사항에 비해 지나치게 복잡성이 증가하게 된다.

Appendix

확장성을 가지도록 설계하는 기법

BDIF(Big Design Up Front) 기법을 이용하여 고객과 문제 영역을 고찰하고, 고객의 요구사항의 변경 가능성 또한 모든 경우의 수를 대비하여 도메인 모델을 만든다. 그 결과 OCP가 가능하도록 설계할 수 있지만 미처 예측할 수 없던 다른 변경사항이 발생하게 되면 결국 도메인 모델을 수정해야 하며, 완벽한 도메인 모델을 세우기 위해 들인 시간은 무의미한 시간이 되어버린다.
Agile Design 기법은 이와 상반되는 기법으로 최대한 빨리 고객의 요구사항을 끌어낼 수 있도록 실행 가능한 결과물을 먼저 보여준 후 피드백을 받는 내용을 통해 변경 요구사항을 끌어내 다시 실행 결과물을 만드는 Iteration을 한다.
가장 좋은 방법은 이 두 가지의 중간 지점을 찾는 것이다.