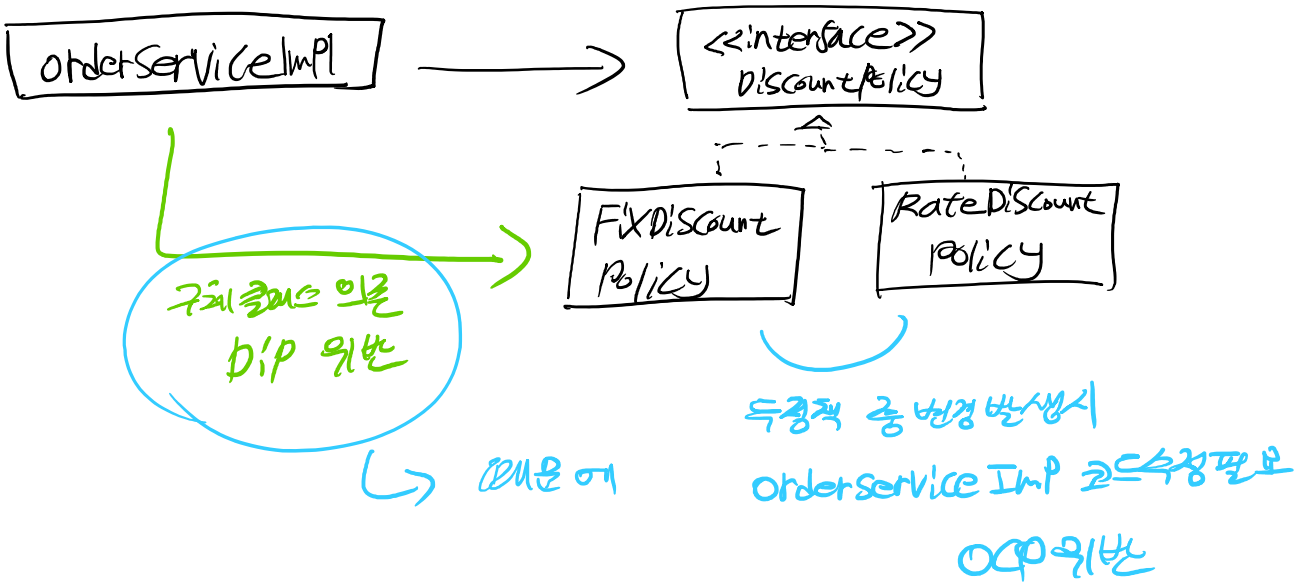
스프링핵심원리-객체지향원리적용

2021년 8월 4일 수요일 오후 4:08

새로운 할인 정책을 확장
=> 기존의 정액 할인에서 정률 할인으로 바꾸자

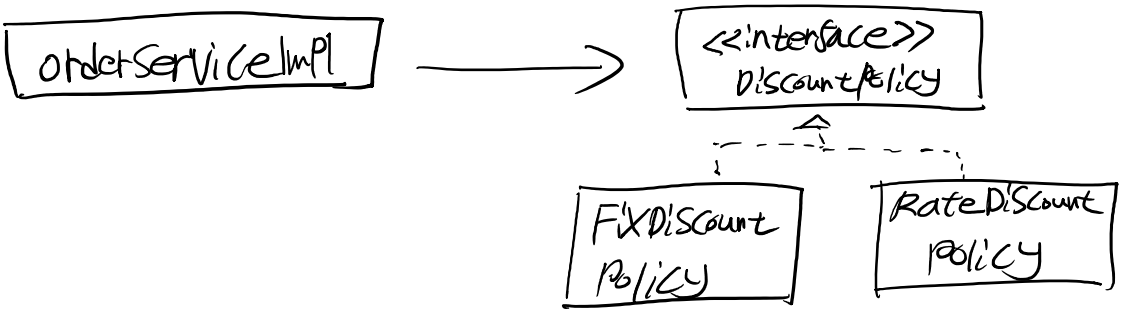
```
public class OrderServiceImpl implements OrderService{

    private final MemberRepository memberRepository = new MemoryMemberRepository();
    //private final DiscountPolicy discountPolicy = new FixDiscountPolicy(); //정액
    private final DiscountPolicy discountPolicy = new RateDiscountPolicy(); //정률
```

할인 정책을 변경하려면 클라이언트인 OrderServiceImpl 코드를 변경해야됨
OCP, DIP 같은 객체 지향 설계 원칙을 지킨 것처럼 보이지만 사실은 아님
why? => OrderServiceImpl이 추상(인터페이스) 뿐만이 아니라 구체(구현) 클래스에도 의존하고 있기 때문



어떻게 해결하는가?
인터페이스에만 의존 하도록 설계 변경

```
public class OrderServiceImpl implements OrderService{

    private final MemberRepository memberRepository = new MemoryMemberRepository();
    private DiscountPolicy discountPolicy;
```



=> 이러면 위의 객체지향 규칙을 지키는데에는 성공했지만,
구현체가 없어서 실제 실행이 안되버림 (Null포인트 Exception)
=> 그래서 누군가 대신 DiscountPolicy의 구현체를 만들어서 주입해주어야 한다

관심사의 분리

AppConfig 등장
어플리케이션의 전체 동작 방식을 구성하기 위해, 구현 객체를 생성하고 연결하는 책임을 가지는 별도의 설정 클래스

```java
public class AppConfig {

    public MemberService memberService() {

        return new MemberServiceImpl(new MemoryMemberRepository());
    }
```

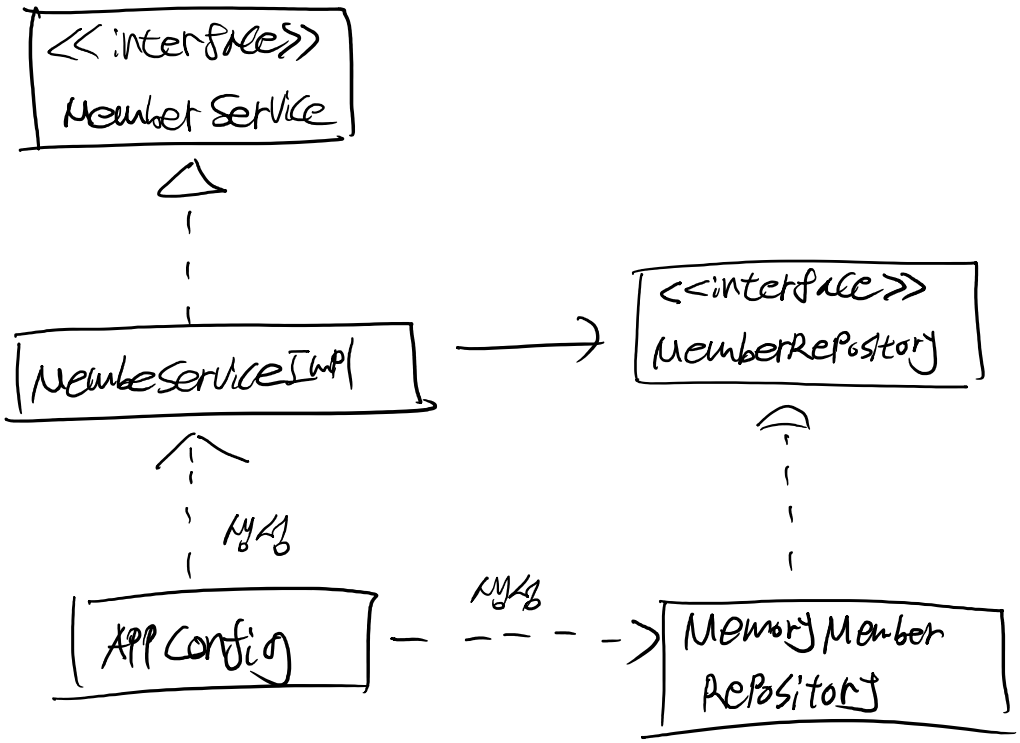
```java
public class MemberServiceImpl implements MemberService {

    private final MemberRepository memberRepository;

    4 related problems
    public MemberServiceImpl(MemberRepository memberRepository) {
        this.memberRepository = memberRepository;
    }
```

MemberServiceImpl 에서 MemoryMemberRepository에 대한 코드 없이 사용 (생성자를 통한 주입)
=> MemoryMemberRepository에 의존하지 않음, MemberRepository 인터페이스에만 의존

MemberServiceImpl 입장에서는 생성자를 통해 어떤 구현 객체가 들어올지 모름
이것을 결정하는 것은 외부(AppConfig)
고로 의존관계에 대한 고민 필요없고 실행에만 집중

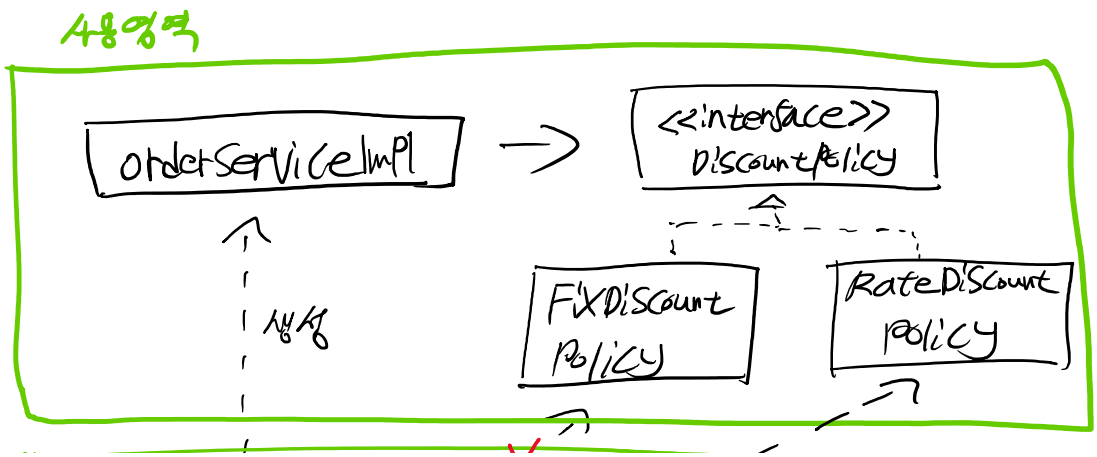
AppConfig 에서
실제 동작에 필요한 구현객체 생성
생성한 객체 인스턴스에 대한 레퍼런스를 생성자를 통해서 주입(연결)해준다 (의존관계 주입)

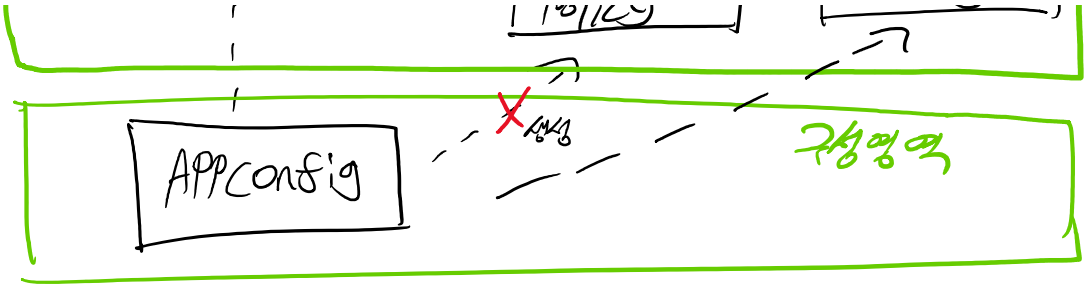
=> DIP 지킴 , 관심사의 분리(객체 생성,연결 하는 역활과 실행하는 역할 분리)



AppConfig 에 역활들이 들어나야...

AppConfig의 등장으로 어플리케이션이 크게 사용영역과 구성영역으로 나누어짐





FixDiscountPolicy에서 RateDiscountPolicy로 변경해도 구성영역만 영향받고,
사용영역은 영향 받지 않음

좋은 객체 지향 설계의 5가지 원칙의 적용

SRP 단일 책임 원칙 : 한 클래스는 하나의 책임만 가져야 한다

구현 객체를 생성하고 연결하는 책임은 AppConfig가 담당
클라이언트 객체는 실행하는 책임만 담당

DIP 의존관계 역전 원칙 : "추상화에 의존해야, 구체화에 의존X"
의존성 주입은 이 원칙을 따르는 방법 중 하나

AppConfig 가 FixDiscountPolicy 객체 인스턴스를 클라이언트 코드 대신 생성하여
클라이언트 코드에 의존관계 주입 = DIP 원칙 따르면서 새로운 할인 정책 적용

OCP : 소프트웨어 요소는 확장에는 열려 있으나 변경에는 닫혀 있어야 한다

AppConfig가 의존관계를 변경해서 클라이언트 코드에 주입하므로 클라이언트 코드는 변경할 필요X 즉, 소프트웨어 요소를 새롭게 확장해도 사용 영역의 변경은 닫혀 있다

제어의 역전 (IoC, Inversion of Control)

기존 : 클라이언트 구현 객체가 스스로 필요한 서버 구현 객체를 생성하고 연결하고 실행
즉, 구현 객체가 프로그램의 제어 흐름을 스스로 조종

AppConfig 등장 이후 : 프로그램에 대한 제어 흐름에 대한 권한 AppConfig가 모두 가지고 있음
구현객체는 자신의 로직을 실행하는 역활만 함
즉 프로그램의 제어흐름을 외부에서 관리 이것이 제어의 역전

프레임워크 vs 라이브러리

프레임 워크는 작성한 코드를 제어하고 대신 실행함

라이브러리는 작성한 코드가 직접 제어의 흐름을 담당

의존관계 주입 DI (Dependency Injection)

의존관계는 정적인 클래스 의존 관계와 실행 시점에 결정되는 동적인 객체 의존관계 둘을 분리해서 생각해야 함

정적인 클래스 의존 관계
클래스가 사용하는 import 코드만 보고 의존관계를 쉽게 판단할 수 있다. 정적인 의존관계는 어플리케이션을 실행하지 않아도 분석할 수 있다.

동적인 객체 인스턴스 의존 관계
어플리케이션 실행 시점에 실제 생성된 객체 인스턴스의 참조가 연결된 의존 관계이다

어플리케이션 실행시점(런타임)에 외부에서 실제 구현 객체를 생성하고 클라이언트에 전달해서 클라이언트와 서버의 실제 의존관계가 연결되는 것을 "의존관계 주입"이라고 함
객체 인스턴스를 생성하고 참조값을 전달해서 연결함
의존관계 주입 사용하면, 클라이언트 코드 수정없이 클라이언트가 호출하는 대상의 타입 인스턴스 변경 가능
의존관계 주입 사용하면, 정적인 클래스 의존관계 변경없이 동적인 객체 인스턴스 의존관계를 변경할 수 있음

IOC 컨테이너= DI 컨테이너
AppConfig 처럼 객체를 생성하고 관리하면서 의존관계를 연결해주는 것
어셈블러, 오브젝트 팩토리 등으로 불리기도 함