Spring 핵심 기본

객체 지향 프로그래밍이란

객체 지향 프로그래밍이란 객체들의 모임과 메시지를 주고받는 협력으로 파악하고자 하는 것

- 어떤 클래스가 필요한지를 고민하기 전에 어떤 객체들이 필요한지를 고민하라
  - Why? 클래스는 공통적인 상태와 행동을 공유하는 객체들을 추상화 한 것
  - 클래스의 윤곽을 잡기 위해서는 어떤 상태와 행동을 가지는 객체가 필요한지를 생각해 봐야 한다.

- 객체는 독립적인 존재가 아니라 기능을 구현하기 위해 협력하는 공동체의 일원으로 봐야한다.
  - 객체는 홀로 존재하는게 아니다
  - 객체는 다른 객체에게 도움을 주거나 의존하면서 살아가는 협력적인 존재다.
  - 협력을 하기 위해서는 그 객체의 역할이 무엇인지 명확해야한다.

다형성(Polymorphism)

객체 지향 프로그래밍은 유연하고 변경이 용이하게 만든다

유연하고 변경이 용이하다
- 컴포넌트를 쉽고 유연하게 갈아 끼울 수 있다
- 새로운 기능을 쉽게 추가할 수 있다
- 기능을 바꿀 때 수정해야 할 부분이 적다.

유연하고 변경이 용이하게 개발하기 위해서 필요한 것이 다형성

다형성을 위해서는 역할과 구현으로 세상을 구분해야한다.
- 실제 예시로 자동차와 K3, 테슬라
  - 새로운 자동차가 나와도 나는 운전할 수 있다.
- 역할은 인터페이스
- 구현은 실제 구현 클래스
- 유연하고 변경이 용이하기 위해서는 역할에 의존해야한다.
  - 언제든지 다른 걸로 갈아끼울 수 있도록
  - 내부 구현은 알지 몰라도 된다 (추상화)
- 컴파일 시점의 의존성과 실행 시점의 의존성이 다를 수 있다.
  - 컴파일 시점에는 인터페이스에 의존하지만
  - 실행 시점에는 실제 인스턴스에 의존한다.
  - 그러므로 실행시점에는 동일한 메시지를 수신했을 때 객체에 따라 다르게 응답할 수 있다.

객체 설계시 역할(인터페이스)를 먼저 부여하고 그 역할을 수행하는 객체를 만들자!

객체 지향 설계의 5가지 원칙 (SOLID)

SOLID: 클린 코드로 유명한 로버트 마틴이 좋은 객체 지향 설계의 5가지 원칙을 정리

SRP(Single responsibility principle): 단일 책임의 원칙

OCP(Open / Closed principle): 개방-패쇄 원칙

LSP(Liskov substitution principle): 리스코프 치환 원칙

ISP (Interface segregation principle): 인터페이스 분리 원칙

DIP (Dependency inversion principle): 의존관계 역전 원칙

객체 지향 설계의 5가지 원칙 (SOLID)

SRP(Single responsibility principle): 단일 책임의 원칙
- 하나의 클래스는 하나의 책임만을 가져야 한다 == 어떤 클래스의 변경 이유는 오직 하나 뿐이어야 한다

책임이라는 건 하는 것(doing)과 아는 것(knowing)의 두가지 범주로 세분화 된다

하는 것
- 객체를 생성하거나 계산을 수행하는 것
- 다른 객체의 행동을 시작하는 것
- 다른 객체의 활동을 제어하고 조절하는 것

아는 것
- 자신이 유도하거나 계산할 수 있는 것에 대해 아는 것

Example) Order (주문) 객체
- 하는 것: 주문 가격을 계산한다
- 아는 것: 주문에 대한 정보를 알고 있다.

하나의 객체가 수행할 수 있다고 생각했던 책임이 여러 객체가 협력해야만 하는 커다란 책임일 수 있다

적절한 책임을 적절한 객체에게 할당해주는게 중요하다.
- 적절하게 책임을 할당했다면 변경할 때 파급효과가 적다

잘못된 example) 하나의 JSP 파일에 DB 접근 정보까지 다 들어있는 경우

객체 지향 설계의 5가지 원칙 (SOLID)

OCP(Open / Closed principle): 개방-패쇄 원칙

소프트웨어는 확장에는 열려있으나 변경에는 닫혀 있어야 한다 == 기능이 추가되거나 변경될 때 이미 제대로 된 코드를 변경하지 않고 새로운 코드를 추가하는 것으로 변경과 추가가 가능하다.

어떻게? 다형성을 생각해보면 된다
- 역할과 구현을 구분해서 생각해보면 새로운게 나오면 구현에 추가하면 된다 - 이건 변경이 아님

Example

상황) MemoryMemberRepository를 쓰다가 JDBCMemberRepository로 변경해야 하는 경우

```
private final MemberRepository memberRepository;

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

OCP를 위해서는 객체를 생성하고 연관관계를 맺어주는 별도의 조립, 생성자가 필요하다
- 객체에 대한 생성과 사용을 분리해야 한다
    - 소프트웨어 시스템에서는 객체를 생성하고 의존성을 서로 연결하는 시작 단계와 실행 단계를 분리해야 한다

LSP(Liskov substitution principle): 리스코프 치환 원칙

프로그램의 객체는 프로그램의 정확성을 깨면 안된다 == 인터페이스 구현 객체는 인터페이스 규약을 지켜야 한다

부모 클래스에서 가능한 모든 행위는 자식 클래스에서 수행이 가능해야 한다.

- Overriding을 모두 해야한다.

ISP (Interface segregation principle): 인터페이스 분리 원칙

자신이 이용하지 않는 메소드에 의존하면 안된다

하나의 범용적인 인터페이스를 사용하기 보다는 적절하게 쪼개서 클라이언트에 특화된 인터페이스를 사용하도록 해야한다

ISP (Interface segregation principle): 인터페이스 분리 원칙

자신이 이용하지 않는 메소드에 의존하면 안된다

하나의 범용적인 인터페이스를 사용하기 보다는 적절하게 쪼개서 클라이언트에 특화된 인터페이스를 사용하도록 해야한다

- 인터페이스가 명확해지고 대체 가능성이 좋아진다

DIP (Dependency inversion principle): 의존관계 역전 원칙

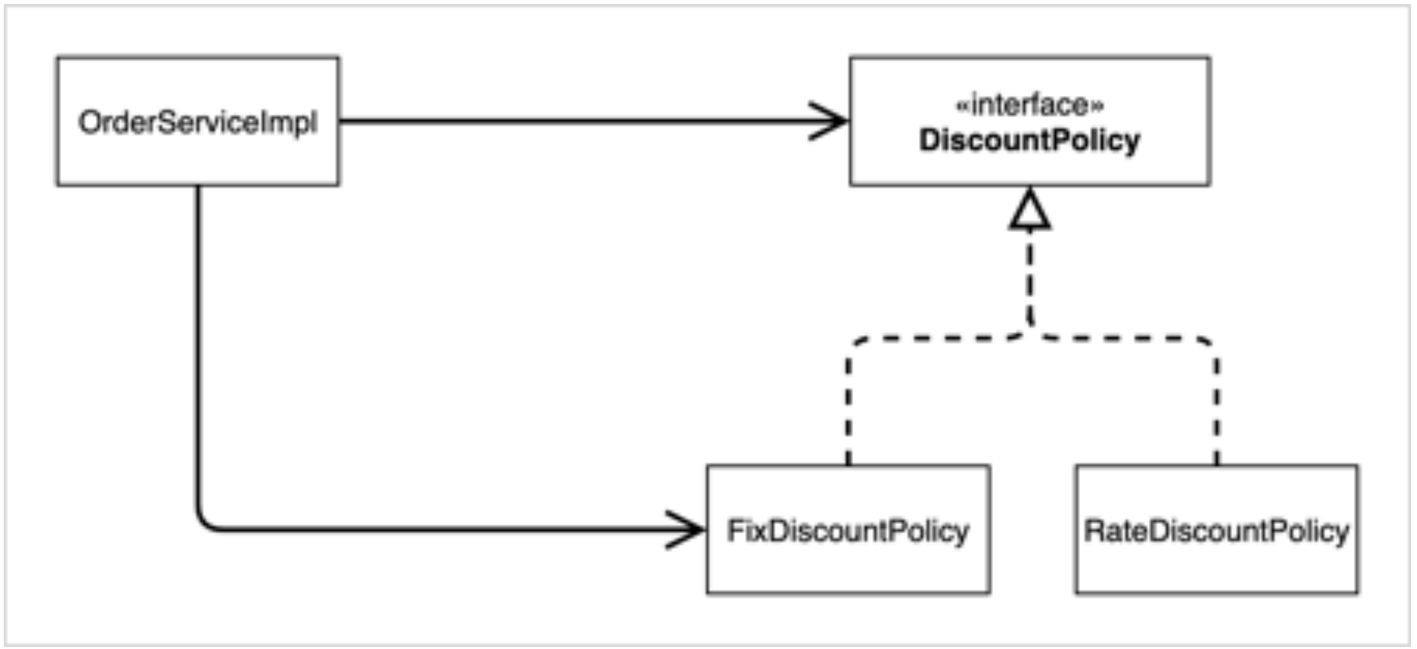
프로그래머는 추상화에 의존해야지 구체화에 의존하면 안된다

- 상위 수준의 모듈은 하위 수준의 모듈을 의존하면 안된다 둘 모두 구체화에 의존해야한다.
- 앞에서 이야기한 역할에 의존해야한다.

DiscountPolicy - int discount()

FixDiscountPolicy - int discount() **RateDiscountPolicy - int discount()**

Example - OrderService

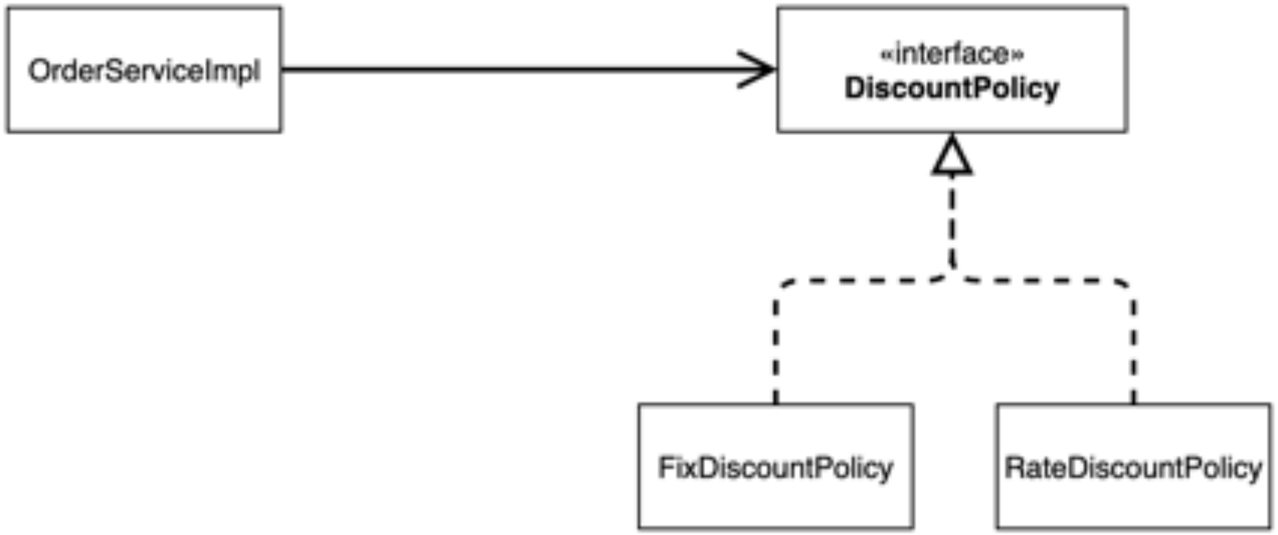


Legacy

OrederServiceImpl이 DiscountPolicy뿐 아니라 FixDiscountPolicy인 구체 클래스에도 의존하는 문제점이 있다.

- **FixDiscountPolicy라는 구체화에 의존하고 있으므로 DIP 위반**
- **할인 정책을 바꿔야 할 때 OrderServiceimpl의 코드를 변경해야 하므로 OCP 위반**

Example - OrderService

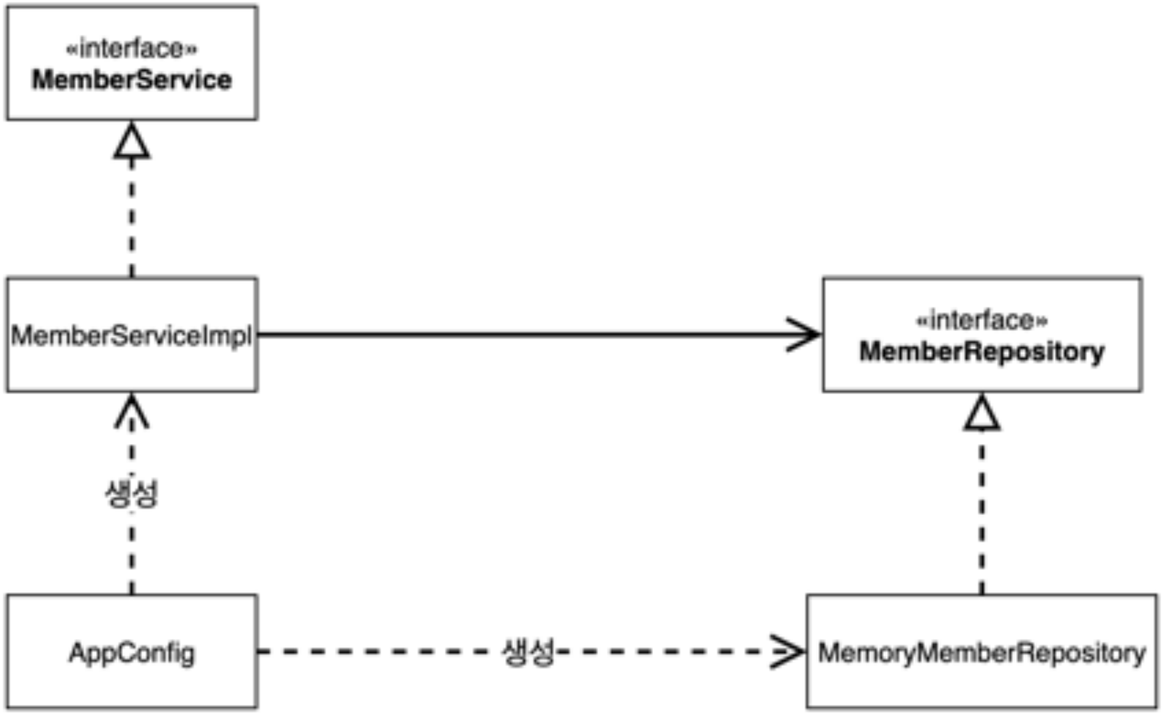


New - 인터페이스만 의존하도록 설계를 바꾸자

이렇게 바꿀경우 구현체가 없는데 어떻게 코드를 실행시키는가?

- **OCP를 위해서 객체의 생성과 사용을 분리해야한다.**
  - **누군가 클라이언트인 OrderServiceImpl에 DiscountPolicy의 구현 객체를 대신 생성해주고 주입해주어야 한다.**
    - **이걸 위해 AppConfig (스프링 컨테이너)의 등장**

Example - OrderService



AppConfig (스프링 컨테이너)

- **AppConfig는 애플리케이션의 실제 동작을 수행하는 구현객체를 생성하고 의존성을 넣어준다**
  - **MemoryServiceImpl**
  - **MemoryMemberRepository**
  - **OrderServiceImpl**

- **이렇게 함으로써 OrderServiceImpl, MemberServiceImpl은 실행에만 집중하면 된다**
- **OrderServiceImpl, MemberServiceImpl은 컴파일 시점에 인터페이스(역할)에만 집중하면 되고**
- **실제로 어떤 객체를 사용할건지는 런타임시점에 외부에서 결정해준다.**