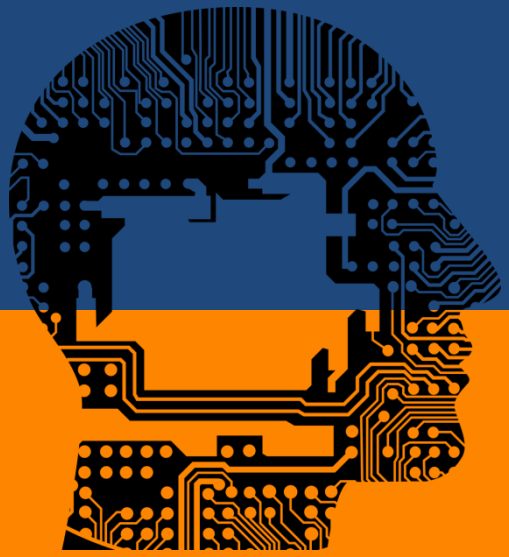
한국IT진흥부설
정보보호교육학원 아이섹

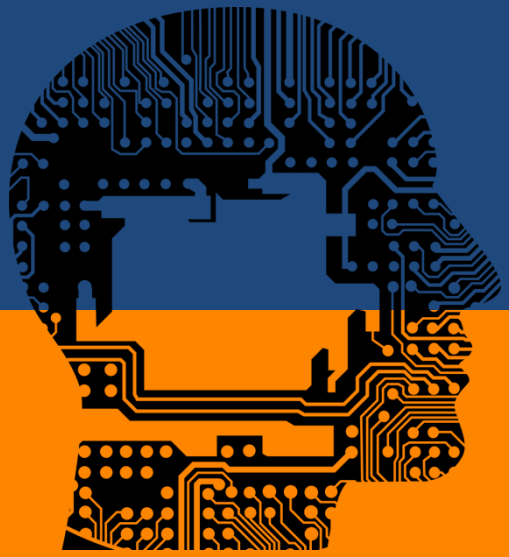


JAVA 웹 개발자 양성과정
Spring: Spring Core

2강 - 객체지향적 설계

By SoonGu Hong

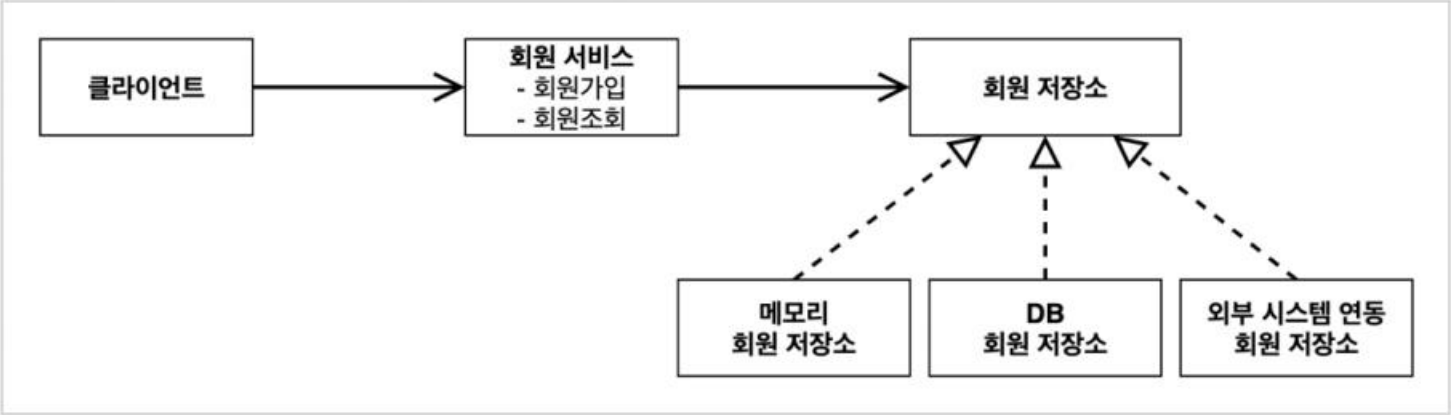
한국IT진흥부설
정보보호교육학원 아이섹



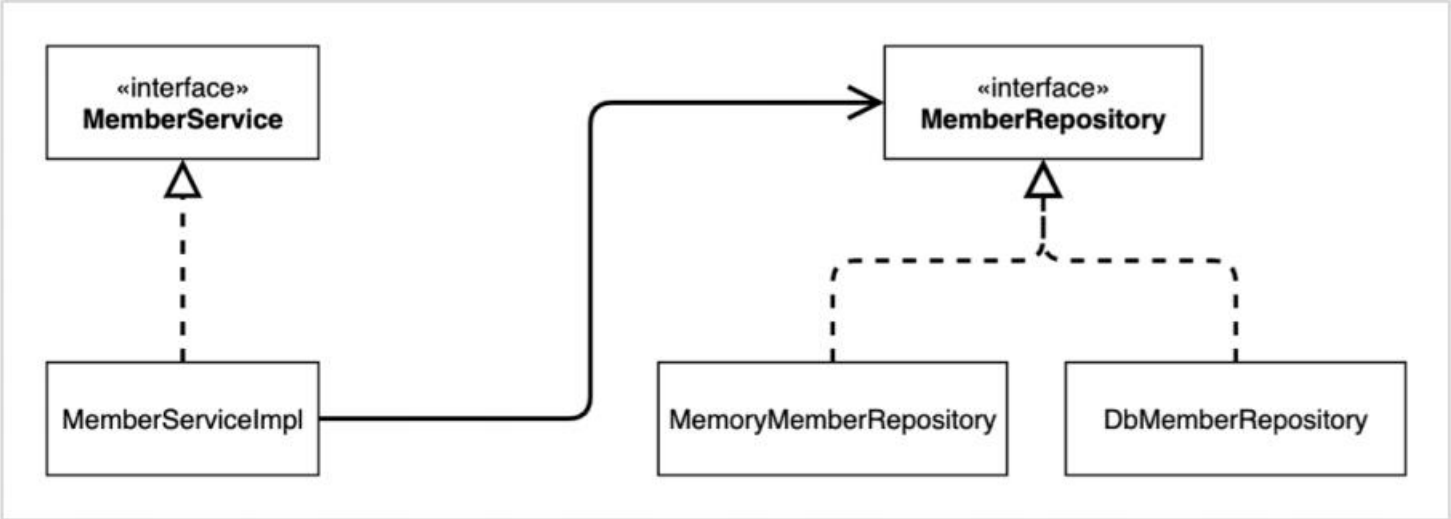
JAVA 웹 개발자 양성과정
Spring: Spring Core
1. 비즈니스 요구사항과 설계

- 회원
  - 회원을 가입하고 조회할 수 있다.
  - 회원은 일반과 VIP 두 가지 등급이 있다.
  - 회원 데이터는 자체 DB를 구축할 수 있고, 외부 시스템과 연동할 수 있다. (미확정)
- 주문과 할인 정책
  - 회원은 상품을 주문할 수 있다.
  - 회원 등급에 따라 할인 정책을 적용할 수 있다.
  - 할인 정책은 모든 VIP는 1000원을 할인해주는 고정 금액 할인을 적용해달라. (나중에 변경 될 수 있다.)
  - 할인 정책은 변경 가능성이 높다. 회사의 기본 할인 정책을 아직 정하지 못했고, 오픈 직전까지 고민을 미루고 싶다. 최악의 경우 할인을 적용하지 않을 수 도 있다. (미확정)

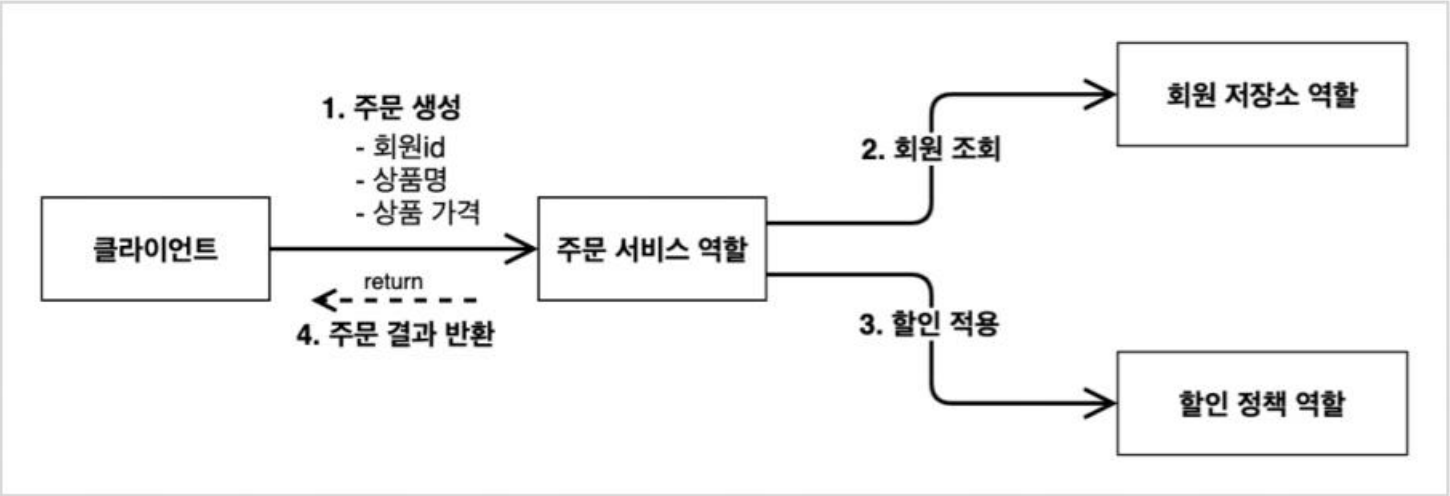
회원 도메인 협력 관계



회원 클래스 다이어그램



주문 도메인 협력, 역할, 책임

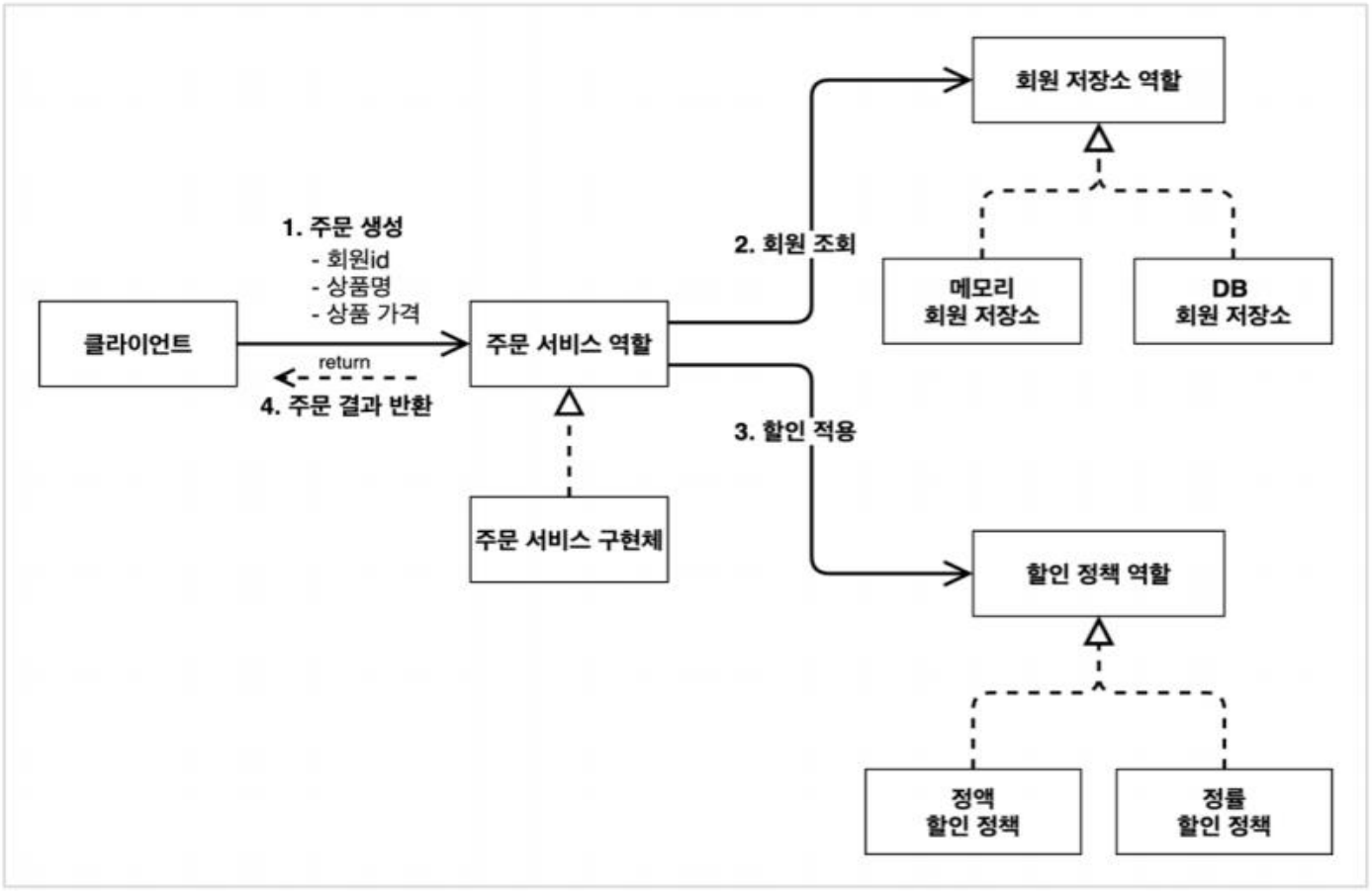


1. 주문 생성: 클라이언트는 주문 서비스에 주문 생성을 요청한다.

2. 회원 조회: 할인을 위해서는 회원 등급이 필요하다. 그래서 주문 서비스는 회원 저장소에서 회원을 조회한다.

3. 할인 적용: 주문 서비스는 회원 등급에 따른 할인 여부를 할인 정책에 위임한다.

4. 주문 결과 반환: 주문 서비스는 할인 결과를 포함한 주문 결과를 반환한다.



역할과 구현을 분리해서 자유롭게 구현 객체를 조립할 수 있게 설계했다. 덕분에 회원 저장소는 물론이고, 할인 정책도 유연하게 변경할 수 있다.

새로운 할인 정책 적용과 문제점

방금 추가한 할인 정책을 적용해보자.

할인 정책을 애플리케이션에 적용해보자.

할인 정책을 변경하려면 클라이언트인 `OrderServiceImpl` 코드를 고쳐야 한다.

```
public class OrderServiceImpl implements OrderService {

//    private final DiscountPolicy discountPolicy = new FixDiscountPolicy();
    private final DiscountPolicy discountPolicy = new RateDiscountPolicy();

}
```

문제점 발견

- 우리는 역할과 구현을 충실하게 분리했다. ➡ OK
- 다형성도 활용하고, 인터페이스와 구현 객체를 분리했다. ➡ OK
- OCP, DIP 같은 객체지향 설계 원칙을 충실히 준수했다
  - ➡ 그렇게 보이지만 사실은 아니다.
- DIP: 주문서비스 클라이언트(`OrderServiceImpl`)는 `DiscountPolicy` 인터페이스에 의존하면서 DIP를 지킨 것 같은데?
  - ➡ 클래스 의존관계를 분석해 보자. 추상(인터페이스) 뿐만 아니라 **구체(구현) 클래스에도 의존**하고 있다.
    - 추상(인터페이스) 의존: `DiscountPolicy`
    - 구체(구현) 클래스: `FixDiscountPolicy` , `RateDiscountPolicy`
- OCP: 변경하지 않고 확장할 수 있다고 했는데!
  - ➡ **지금 코드는 기능을 확장해서 변경하면, 클라이언트 코드에 영향을 준다!** 따라서 **OCP를 위반**한다.

어떻게 문제를 해결할 수 있을가?

- 클라이언트 코드인 `OrderServiceImpl`은 `DiscountPolicy`의 인터페이스 뿐만 아니라 구체 클래스도 함께 의존한다.
- 그래서 구체 클래스를 변경할 때 클라이언트 코드도 함께 변경해야 한다.
- **DIP 위반** → 추상에만 의존하도록 변경(인터페이스에만 의존)
- DIP를 위반하지 않도록 인터페이스에만 의존하도록 의존관계를 변경하면 된다.

인터페이스에만 의존하도록 설계를 변경하자

인터페이스에만 의존하도록 코드 변경

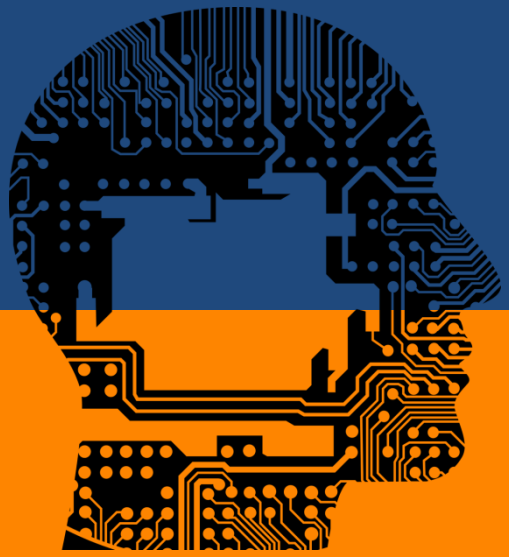
```
public class OrderServiceImpl implements OrderService {
    //private final DiscountPolicy discountPolicy = new RateDiscountPolicy();
    private DiscountPolicy discountPolicy;
}
```

- 인터페이스에만 의존하도록 설계와 코드를 변경했다.
- **그런데 구현체가 없는데 어떻게 코드를 실행할 수 있을까?**
- 실제 실행을 해보면 NPE(null pointer exception)가 발생한다.

해결방안

- 이 문제를 해결하려면 누군가가 클라이언트인 `OrderServiceImpl`에 `DiscountPolicy`의 구현 객체를 대신 생성하고 주입해주어야 한다.

한국IT진흥부설
정보보호교육학원 아이섹



JAVA 웹 개발자 양성과정

Spring: Spring Core

2. 관심사의 분리

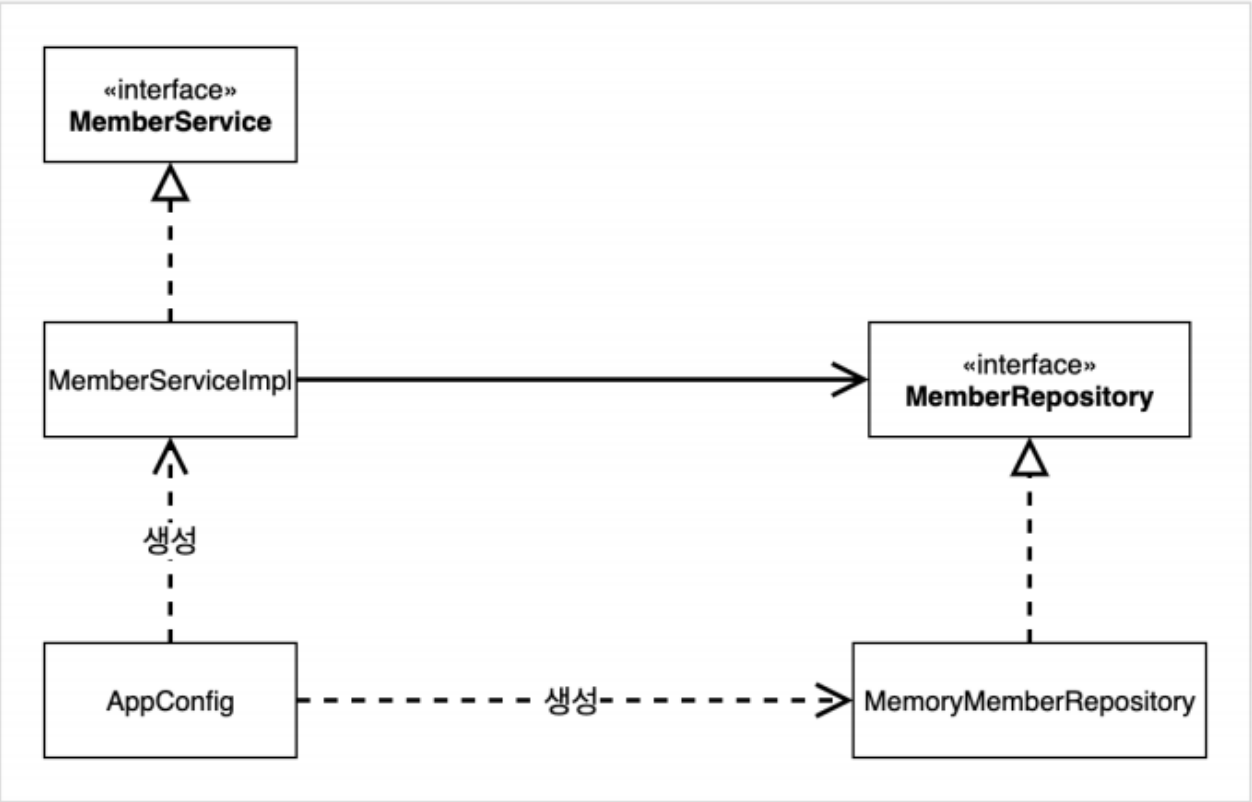
관심사를 분리하자

- 배우는 본인의 역할인 배역을 수행하는 것에만 집중해야 한다.
- 디카프리오는 어떤 여자 주인공이 선택되더라도 똑같이 공연을 할 수 있어야 한다.
- 공연을 구성하고, 담당 배우를 섭외하고, 역할에 맞는 배우를 지정하는 책임을 담당하는 별도의 **공연 기획자**가 나올시점이다.
- 공연 기획자를 만들고, 배우와 공연 기획자의 책임을 확실히 분리하자.

AppConfig 등장

- 애플리케이션의 전체 동작 방식을 구성(config)하기 위해, **구현 객체를 생성**하고, **연결**하는 책임을 가지는 별도의 설정 클래스를 만들자.

그림 - 클래스 다이어그램



- 객체의 생성과 연결은 `AppConfig` 가 담당한다.
- **DIP 완성:** `MemberServiceImpl` 은 `MemberRepository` 인 추상에만 의존하면 된다. 이제 구체 클래스를 몰라도 된다.
- **관심사의 분리:** 객체를 생성하고 연결하는 역할과 실행하는 역할이 명확히 분리되었다.

관심사의 분리

- 애플리케이션을 하나의 공연으로 생각
- 기존에는 클라이언트가 의존하는 서버 구현 객체를 직접 생성하고, 실행함
- 비유를 하면 기존에는 남자 주인공 배우가 공연도 하고, 동시에 여자 주인공도 직접 초빙하는 다양한 책임을 가지고 있음
- 공연을 구성하고, 담당 배우를 섭외하고, 지정하는 책임을 담당하는 별도의 **공연 기획자**가 나올 시점
- 공연 기획자인 AppConfig가 등장
- AppConfig는 애플리케이션의 전체 동작 방식을 구성(config)하기 위해, **구현 객체를 생성**하고, **연결**하는 책임

IoC, DI, 그리고 컨테이너

제어의 역전 IoC(Inversion of Control)

- 기존 프로그램은 클라이언트 구현 객체가 스스로 필요한 서버 구현 객체를 생성하고, 연결하고, 실행했다. 한마디로 구현 객체가 프로그램의 제어 흐름을 스스로 조종했다. 개발자 입장에서는 자연스러운 흐름이다.
- 반면에 AppConfig가 등장한 이후에 구현 객체는 자신의 로직을 실행하는 역할만 담당한다. 프로그램의 제어 흐름은 이제 AppConfig가 가져간다. 예를 들어서 `OrderServiceImpl` 은 필요한 인터페이스들을 호출하지만 어떤 구현 객체들이 실행될지 모른다.
- 프로그램에 대한 제어 흐름에 대한 권한은 모두 AppConfig가 가지고 있다. 심지어 `OrderServiceImpl` 도 AppConfig가 생성한다. 그리고 AppConfig는 `OrderServiceImpl` 이 아닌 OrderService 인터페이스의 다른 구현 객체를 생성하고 실행할 수 도 있다. 그런 사실도 모른체 `OrderServiceImpl` 은 묵묵히 자신의 로직을 실행할 뿐이다.
- 이렇듯 프로그램의 제어 흐름을 직접 제어하는 것이 아니라 외부에서 관리하는 것을 제어의 역전(IoC)이라 한다.

감사합니다
THANK YOU