Spring 공부

객체 지향 설계와 스프링

▼ 이야기 - 자바 진영의 추운 겨울과 스프링의 탄생

이전에 별로였던 EJB를 보완하고 추운 겨울을 지나 좋은 프레임워크를 만들었다고 해서 스프링이구나

▼ 스프링이란?

스프링 프레임워크, 스프링 부트가 핵심

- 왜 만들었나?
  - 자바 언어 기반의 프레임 워크
  - 객체 지향 언어가 가진 강력한 특징을 살려내는 프레임워크
  - 좋은 객체 지향 애플리케이션을 개발할 수 있게 도와준다

단순한 기술에 초점을 맞추기 보다 왜 만들었는지가 중요하네

객체 지향 언어가 가진 강력한 특징을 살리는 프레임워크가 스프링이다

▼ 좋은 객체 지향 프로그래밍이란?

- 유연하고 변경이 용이하다
- 다형성 : 역할과 구현으로 세상을 구분
  - 자동차가 k3,아반떼, 테슬라로 바뀌어도 자동차의 역할만 바뀌는것
  - 운전자가 자동차를 운전하는 역할은 바뀌지 않음
  - 자동차를 무한으로 구현 가능하다

역할과 구현으로 세상을 구분하면 세상이 단순해지고 유연해지며 변경도 편리해진다

- 클라이언트는 대상의 역할(인터페이스)만 알면된다
- 클라이언트는 구현 대상의 내부 구조를 몰라도 된다
- 클라이언트는 구현 대상의 내부 구조가 변경되어도 영향을 받지 않는다

- 클라이언트는 구현 대상 자체를 변경해도 영향을 받지 않는다

역할 = 인터페이스

구현 = 인터페이스를 구현할 클래스 ,구현 객체

혼자 있는 객체는 없고, 클라이언트의 요청과 서버의 응답으로 협력관계를 가진다

역할과 구현을 분리하여 클라이언트를 변경하지 않고, 서버의 구현기능을 유연하게 변경할 수 있다.

스프링에서는

- 다형성이 가장 중요하다
- 다형성을 극대화해서 이용할 수 있게 도와준다
- 제어의 역전(ioc), 의존관계 주입(DI)는 다형성을 활용해서 역할과 구현을 편리하게 다룰 수 있게 지원한다
- 레고 블럭을 조립하듯이, 공연 무대의 배우를 선택하듯이 구현을 편리하게 변경할 수 있다.

SOLID 개념도 같이 알아야한다

스프링에선 다형성의 개념을 적용시키는데 중요하구나

▼ 좋은 객체 지향 설계의 5가지 원칙(SOLID)

SRP : 단일 책임 원칙(Single Responsibility Principle)

- 한 클래스는 하나의 책임만 가져야 한다
- 변경이 있을때 파급 효과가 적어야한다
- 하나의 클래스에 여러가지가 들어가있으면 안좋다, 깨진것이다
    - UI 변경, 객체의 생성과 사용을 분리한다

OCP : 개방-폐쇄 원칙 (Open/ Closed Principle)

- 확장에는 열려있으나 변경에는 닫혀 있어야 한다

```java
public class MemberService {

   private MemberRepository memberRepository = new MemoryMemberRepository();

}
```

```java
public class MemberService {

//   private MemberRepository memberRepository = new MemoryMemberRepository();
    private MemberRepository memberRepository = new JdbcMemberRepository();


}
```

MemberService 클라이언트가 구현 클래스를 직접 선택해서 다형성을 적용시켜야 하는데 이러면 클라이언트 코드를 변경이 되기 때문에 SOLID 중 OCP가 깨진다. 이것도 하면 안된다

- 객체를 생성하고, 연관관계를 맺어주는 별도의 조립, 설정자가 필요하다
- Spring의 DI등을 활용한다

LSP : 리스코프 치환 원칙(Liskov Substitution Principle)

- 프로그램 객체는 프로그램의 정확성을 깨뜨리지 않으면서 하위 타입의 인스턴스로 바꿀 수 있어야한다
- 다형성에서 하위 클래스는 인터페이스 규약을 다 지켜야하고 인터페이스 구현한 구현체가 믿고 사용하려면 이 원칙이 필요하다
- 컴파일에 성공하는 것을 넘어서는 이야기
    - 자동차 엑셀은 앞으로 가야하는데 뒤로 가도록 구현하면 LSP를 위반하게 된다.

ISP : 인터페이스 분리 원칙 (Interface Segregation Principle)

- 특정 클라이언트를 위한 인터페이스 여러개가 범용 인터페이스 하나보다 낫다
- 자동차 인터페이스 구현할때 운전 인터페이스, 정비 인터페이스 이 두개로 구현하는게 자동차 하나만 구현하는 것보다 훨씬 낫다
- 분리하면 정비 인터페이스 자체가 변해도 운전자 클라이언트에 영향을 주지 않기 때문에 인터페이스가 명확해지고 대체가능성이 높아진다
    - 인터페이스를 기능단위로 작게 분리하자.
        - 인터페이스가 명확해지고 대체가능성이 높아진다

DIP : 의존관계 역전 원칙 (Dependency Inversion Principle)

- 프로그래머는 추상화에 의존해야지, 구체화에 의존하면 안된다.
- 구현 클래스에 의존하지 말고 인터페이스에 의존하라는 뜻
- 역할에 의존해야 한다,

- MemberRepository m = new MemoryMemberRepository();

이건 MemberRepository 클라이언트가 구현 클래스를 의존한다, 코드를 알고 있다는 뜻이다.

DIP 위반하게 된다

다형성 만으로는 DIP를 지킬수가 없네?

객체 지향의 핵심은 다형성인데

- 다형성 만으로는 쉽게 부품을 갈아끼우듯이 개발할 수 없다
- 다형성 만으로는 구현 객체를 변경할 때 클라이언트 코드도 함께 변경된다
- 다형성 만으로는 OCP, DIP를 지킬 수 없다
- 뭔가 더 필요하다
  - 스프링을 사용하는 이유

다형성이 객체지향에서 제일 중요한데, 다형성 만으로는 SOLID 법칙중에 OCP, DIP를 지킬 수 없다.

변경할때 클라이언트에서 코드를 변경해야하고,

구현 클래스를 의존할 수 밖에없다

그래서 Spring을 사용한다

▼ 객체 지향 설계와 스프링

스프링은 다형성 + OCP, DIP를 가능하게 지원한다

- DI : 의존관계 , 의존성 주입
- DI 컨테이너 제공

모든 설계에 역할과 구현을 분리하자

▼ 정리

객체 지향의 SOLID 법칙중 OCP와 DIP는 다형성만으로 구현하기 힘든데 Spring은 이것을 가능케하는 프레임 워크이다.

스프링 핵심 원리 이해1- 예제 만들기

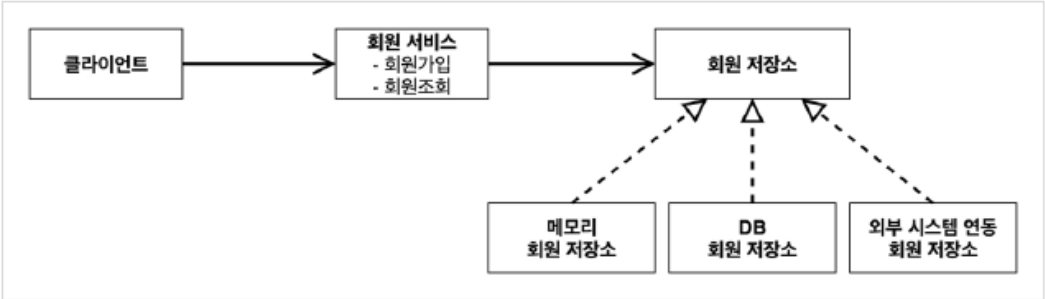
▼ 프로젝트 설계 과정

java로도 인터페이스 바로바로 구현하는게 체화가 안되어있어서 한번 정리

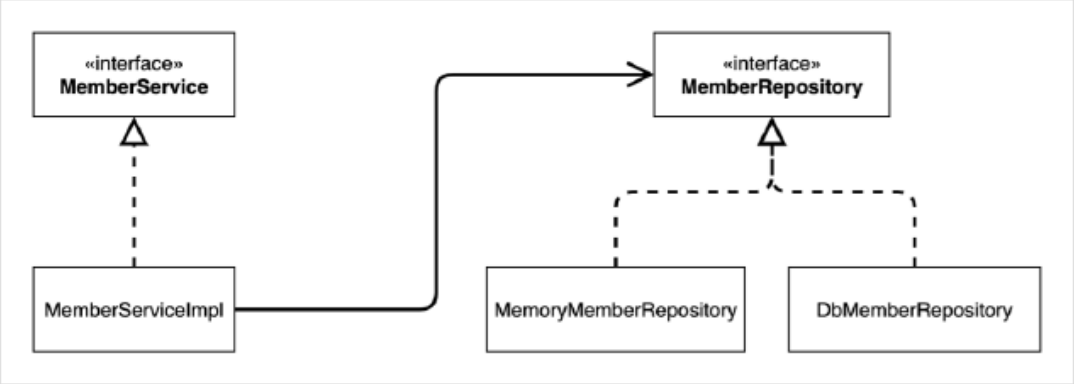
회원 도메인 설계

- 회원 도메인 요구사항
  - 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)

회원 도메인 협력 관계

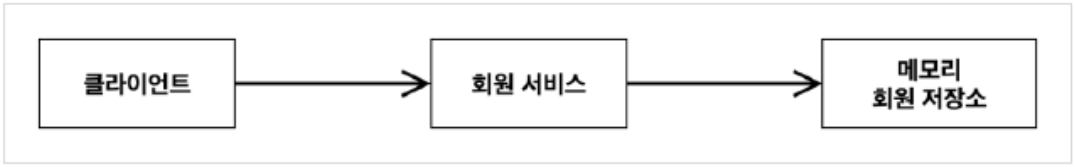


글로 작성된 것을 모두가 알아볼 수 있게 도메인 협력관계를 그린다



개발자를 위한 클래스 다이어그램을 만든다

회원 객체 다이어그램



- 회원 서비스: **MemberServiceImpl**

실제 객체가 사용되는 과정을 그린다

1. 등급이 나누어져있으니 grade enum을 생성

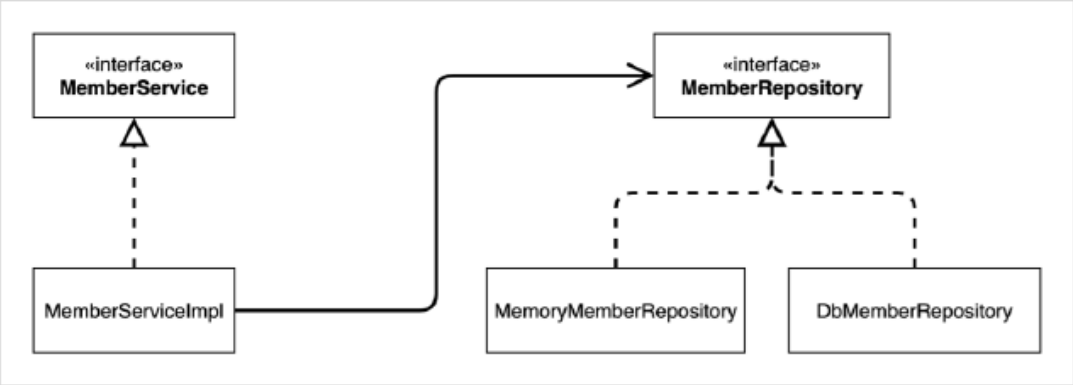
```
public enum Grade {
BASIC,
VIP
}
```

2. Member 클래스를 만든다 (회원 엔티티)

```
public class Member {
private Long id;
private String name;
private Grade grade;
public Member(Long id, String name, Grade grade) {
this.id = id;
this.name = name;
this.grade = grade;
}
```

```
public Long getId() {
return id;
}
public void setId(Long id) {
this.id = id;
}
public String getName() {
return name;
}
public void setName(String name) {
this.name = name;
}
public Grade getGrade() {
return grade;
}
public void setGrade(Grade grade) {
this.grade = grade;
}
}
```

- alt + insert로 생성자와 getter setter 생성



그림을 본 후에 인터페이스를 생성한다

3. 가장 디테일한 MemberRepository 인터페이스 부터 생성

```
public interface MemberRepository {
void save(Member member);
Member findById(Long memberId);
}
```

- 회원 정보를 저장
- 아이디로 회원을 찾는기능

4. 인터페이스에 대한 세부 구현체를 생성한다

```
public class MemoryMemberRepository implements MemberRepository {
private static Map<Long, Member> store = new HashMap<>();
@Override
public void save(Member member) {
store.put(member.getId(), member);
}
@Override
public Member findById(Long memberId) {
return store.get(memberId);
}
}
```

- store hashmap에 저장한 후에 save, findById구현

5. MemberService 인터페이스 구현

```
public interface MemberService {
void join(Member member);
Member findMember(Long memberId);
}
```

6. MemberSerivce 구현체 구현

```
public class MemberServiceImpl implements MemberService {
private final MemberRepository memberRepository = new
MemoryMemberRepository();

public void join(Member member) {
memberRepository.save(member);
}
public Member findMember(Long memberId) {
return memberRepository.findById(memberId);
}
}
```

MemberRepository memberRepository = new MemoryMemberRepository();

- 인터페이스 a = new 인터페이스구현체
- 근데 이렇게 하면 MemberServiceImpl가 인터페이스 뿐만아니라 구현체도 참조하게 된다 :
    - DIP 문제
- 다른 저장소로 변경을 할때 MemoryMemberRepository를 변경해야하는데, 이때 OCP 원칙을 위배하게 된다.

7. 회원가입 main

```
public class MemberApp {
public static void main(String[] args) {
MemberService memberService = new MemberServiceImpl();
Member member = new Member(1L, "memberA", Grade.VIP);
memberService.join(member);
Member findMember = memberService.findMember(1L);
System.out.println("new member = " + member.getName());
System.out.println("find Member = " + findMember.getName());
}
}
```

MemberService memberSerivce = new memberServiceImpl()

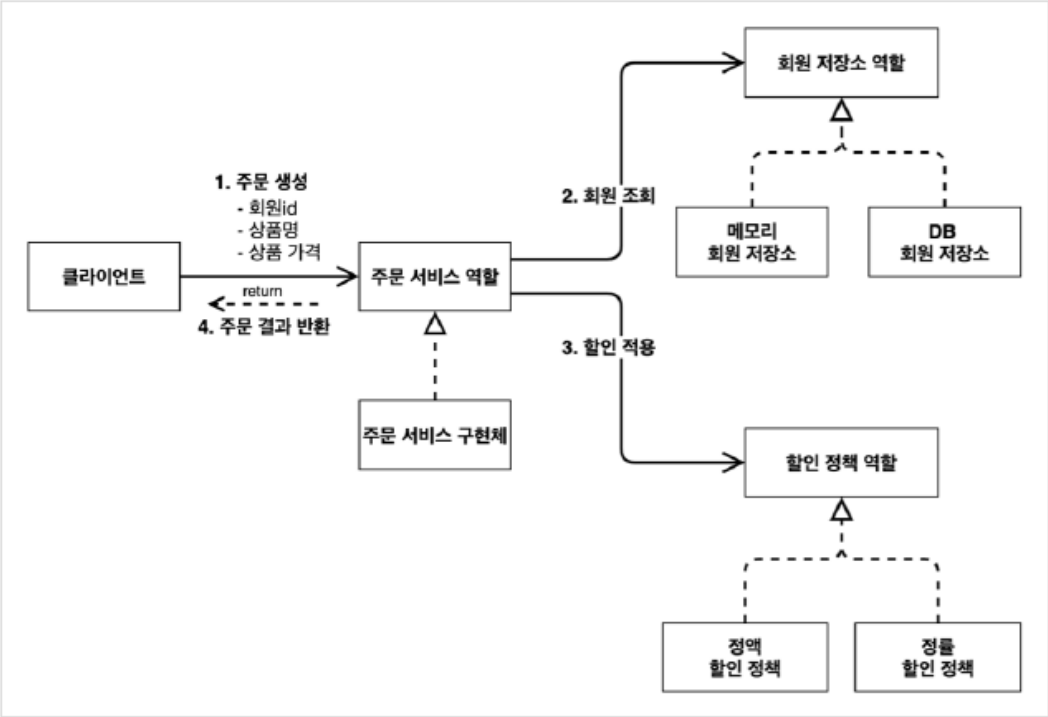
- 인터페이스 a = new 인터페이스구현체();

이렇게하면 좋은 코드가 아니다, 테스트를 별도로 구현

8. test코드 구현

```
class MemberServiceTest {
MemberService memberService = new MemberServiceImpl();
@Test
void join() {
//given
Member member = new Member(1L, "memberA", Grade.VIP);
//when
memberService.join(member);
Member findMember = memberService.findMember(1L);
//then
Assertions.assertThat(member).isEqualTo(findMember);
}
}
```

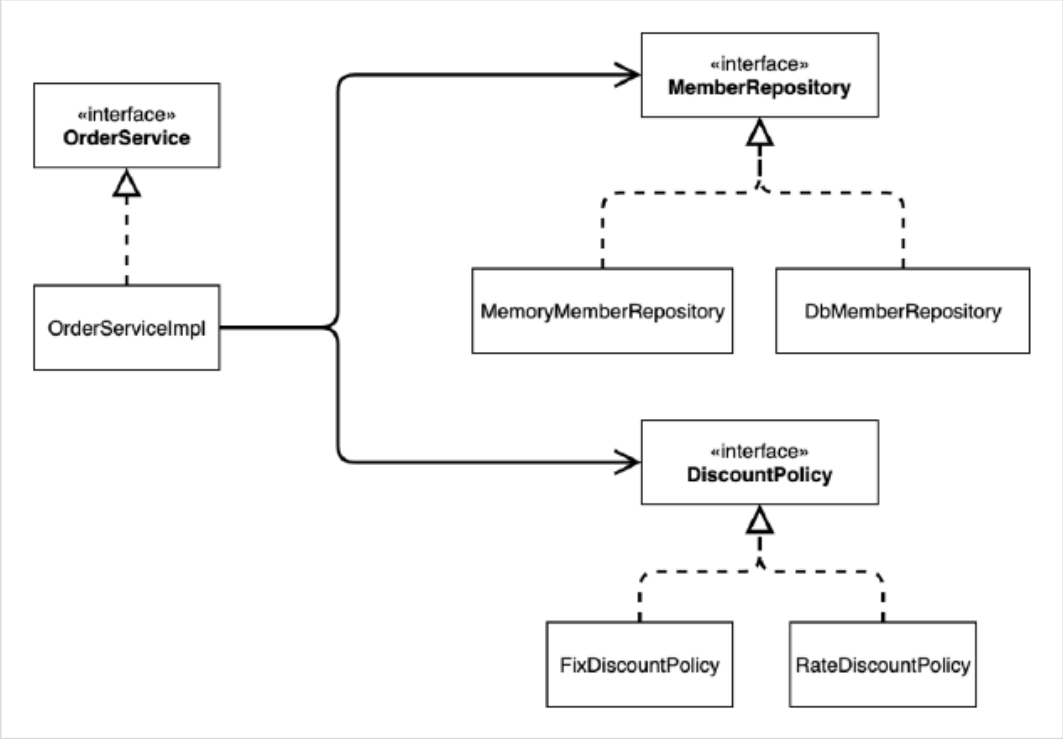
Order



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

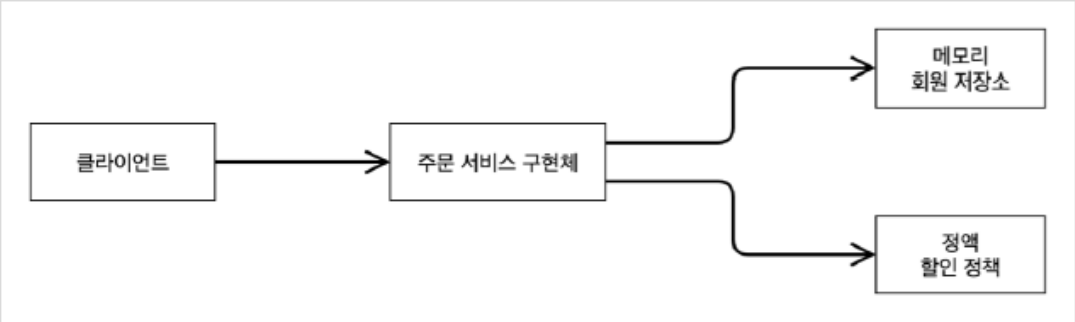
전체적인 다이어그램 구상, 역할과 구현 나누어서

주문 도메인 클래스 다이어그램

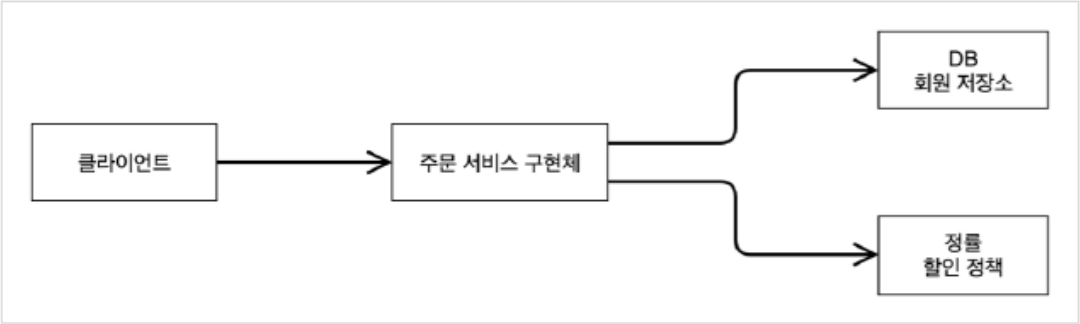


개발자를 위한 클래스 다이어그램

주문 도메인 객체 다이어그램1



주문 도메인 객체 다이어그램2



이렇게 역할과 구현을 나누게 되면 변동사항에 맞춰 쉽게 변환할 수 있다

1. 디테일한 Discount policy 인터페이스

```
public interface DiscountPolicy {
/**
* @return 할인 대상 금액
*/
int discount(Member member, int price);
}
```

2. Discount policy 인터페이스 구현체 FixDiscountPolicy

```
public class FixDiscountPolicy implements DiscountPolicy {
private int discountFixAmount = 1000; //1000원 할인
@Override
public int discount(Member member, int price) {
if (member.getGrade() == Grade.VIP) {
return discountFixAmount;
} else {
return 0;
}
}
```

3. Order 엔티티

```
public class Order {
private Long memberId;
private String itemName;
private int itemPrice;
private int discountPrice;
public Order(Long memberId, String itemName, int itemPrice, int
```

```
discountPrice) {
this.memberId = memberId;
this.itemName = itemName;
this.itemPrice = itemPrice;
this.discountPrice = discountPrice;
}
public int calculatePrice() {
return itemPrice - discountPrice;
}
public Long getMemberId() {
return memberId;
}
public String getItemName() {
return itemName;
}
public int getItemPrice() {
return itemPrice;
}
public int getDiscountPrice() {
return discountPrice;
}
@Override
public String toString() {
return "Order{" +
"memberId=" + memberId +
", itemName='" + itemName + '\'' +
", itemPrice=" + itemPrice +
", discountPrice=" + discountPrice +
'}';
}
}
```

4. OrderService 인터페이스

```
public interface OrderService {
Order createOrder(Long memberId, String itemName, int itemPrice);
}
```

5. OrderService 인터페이스 구현체 OrderServiceImpl

```
public class OrderServiceImpl implements OrderService {
private final MemberRepository memberRepository = new
MemoryMemberRepository();
private final DiscountPolicy discountPolicy = new FixDiscountPolicy();
@Override
public Order createOrder(Long memberId, String itemName, int itemPrice) {
Member member = memberRepository.findById(memberId);
int discountPrice = discountPolicy.discount(member, itemPrice);
return new Order(memberId, itemName, itemPrice, discountPrice);
}
}
```

- member을 memberRepository에서 findById로 찾고
- discountPolicy에서 찾은 member를 넣은 후에 할인정책 적용
- Order 반환

6. OrderApp main 구현

```
public class OrderApp {
public static void main(String[] args) {
MemberService memberService = new MemberServiceImpl();
OrderService orderService = new OrderServiceImpl();
long memberId = 1L;
Member member = new Member(memberId, "memberA", Grade.VIP);
memberService.join(member);
Order order = orderService.createOrder(memberId, "itemA", 10000);
System.out.println("order = " + order);
}
```

- orderService.createOrder로 order 반환

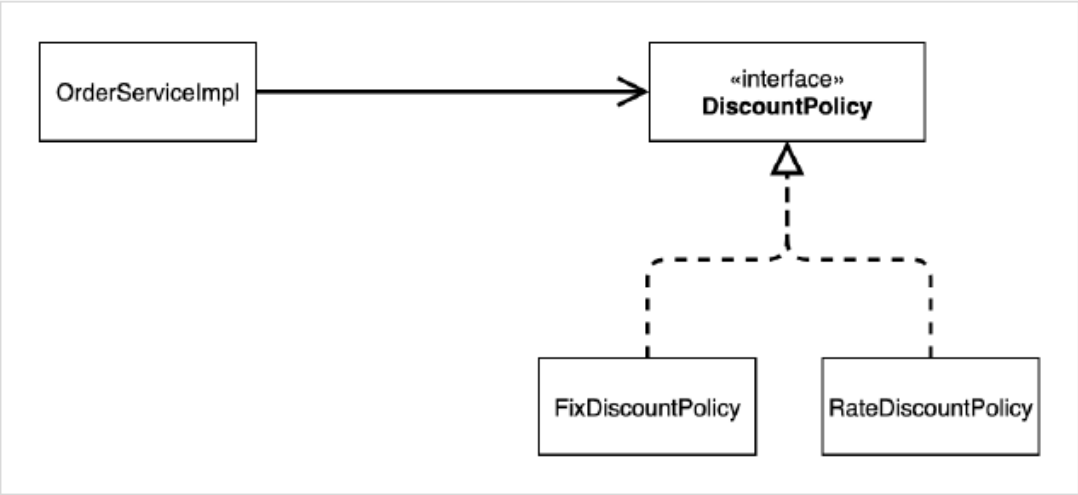
7. Test 코드

```
class OrderServiceTest {
MemberService memberService = new MemberServiceImpl();
OrderService orderService = new OrderServiceImpl();
@Test
void createOrder() {
long memberId = 1L;
Member member = new Member(memberId, "memberA", Grade.VIP);
memberService.join(member);
Order order = orderService.createOrder(memberId, "itemA", 10000);
Assertions.assertThat(order.getDiscountPrice()).isEqualTo(1000);
}
}
```

▼ 개발 사항의 변경, OCP, DIP 위반 문제

- 할인정책을 변경하려고한다

DiscountPolicy 인터페이스에 새로운 구현체를 만들어서 넣는다

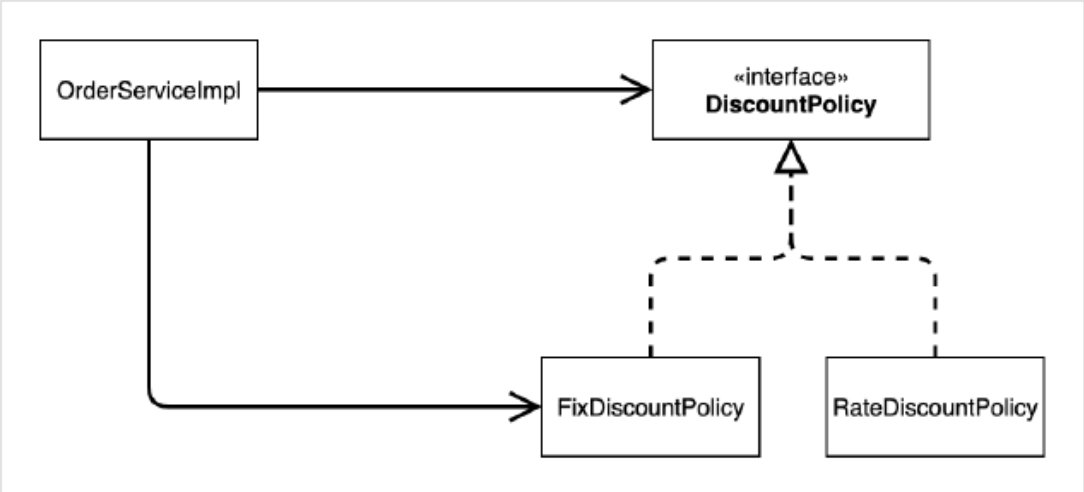


변경할때 OrderServiceImpl에서 변경을 해야함

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

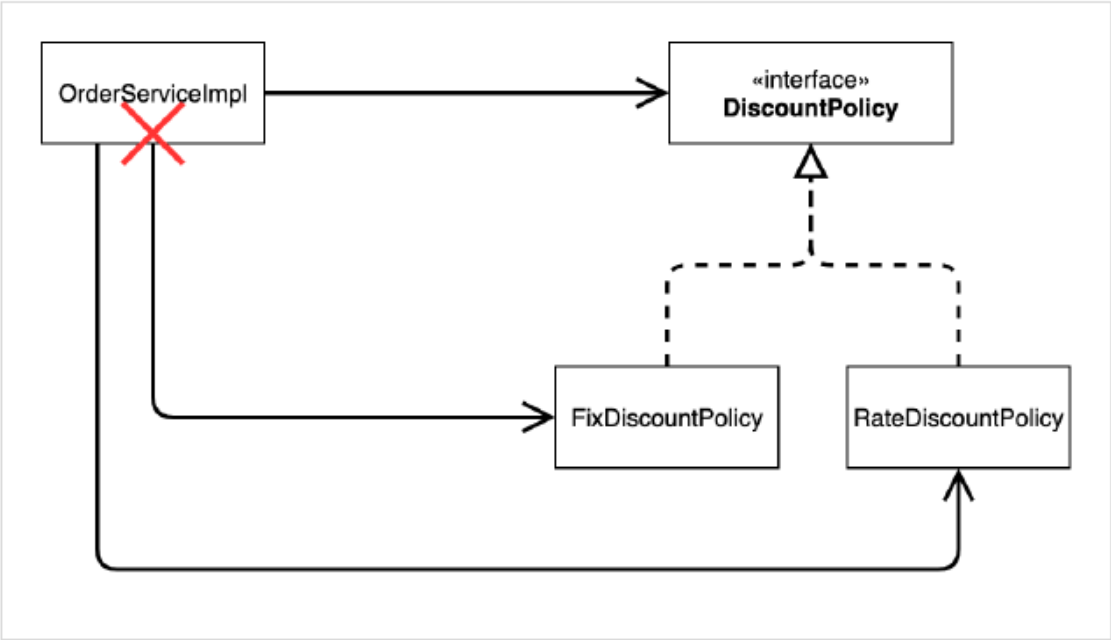
이때

실제 의존관계



기존 의존관계는 interface만 의존하는게 아닌, 구현체 실제 코드도 의존하게 된다

DIP 위반



변경에 닫혀있어야 하는 OCP 위반

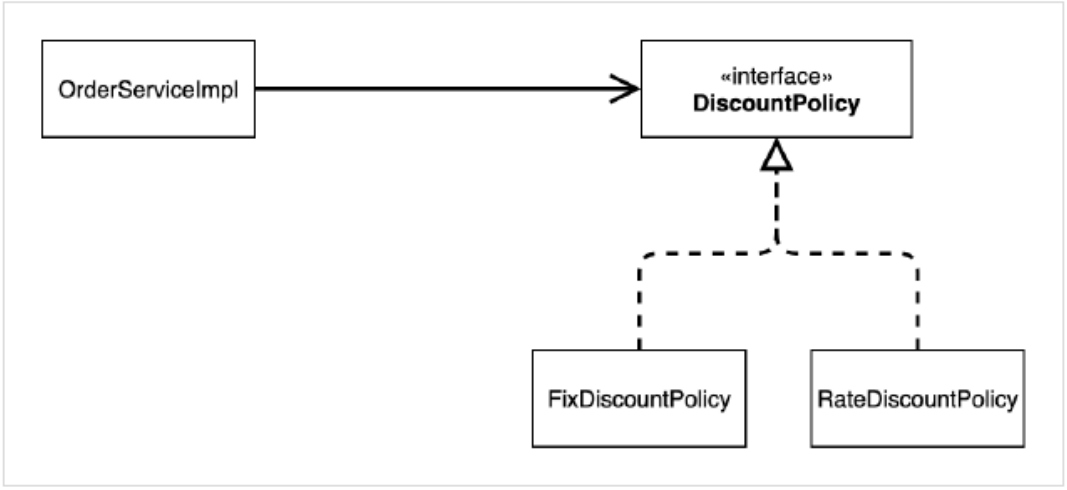
클라이언트 코드에 영향을 준다

어떻게 문제를 해결할까?

▼ DIP (의존관계 문제)의 해결 : 관심사의 분리

- DIP 위반 → 인터페이스에만 의존하도록 해야한다

인터페이스에만 의존하도록 설계를 변경하자



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

이렇게 하면 DIP는 지켜지는데 당연히 코드를 모르니 nullpointerException이 발생하겠지

어떻게 할까?

지금까지 발생한 문제는 디카프리오가 직접 여배우를 선택하는 것과 같다

공연 기획자가 원래 해야한다

공연도 해야하고 동시에 여자 주인공도 공연에 직접 초빙해야 하는 다양한 책임을 가진다

- 관심사의 분리
  - 공연 기획자를 만들고 배우가 공연 기획자의 책임을 확실히 분리하자

  의존관계 DI 개념

  ▼ 자원을 직접 명시하지 말고 의존 객체 주입을 사용하라 (DI)

  ▼ 정리 30p

  많은 클래스가 하나 이상의 자원에 의존할때, dictionary는 하나인데 이를 다양한 클래스에서 사용할때 static으로 자원을 사용하거나, 싱글턴으로 구현하는 경우는 안좋다

  사전을 예로 들어서 , 사전 하나로 이 모든 쓰임에 대응하는것은 불가능하다.

  사용하는 자원에 따라 동작이 달라지는 클래스에는 static 클래스나 싱글턴 방식은 적합하지않다

  → 인스턴스를 생성할때 생성자에 **필요한 자원**을 넘겨준다

  : 의존 객체 주입(Dependency Injection)

```
public class Main {

  public static void main(String[] args) {
    Controller controller = new Controller();
    controller.print();
  }
}

class Controller {

  private Service service;

  public Controller() {
    this.service = new Service();
  }
```

```
  public void print() {
    System.out.println(service.message());
  }
}

class Service {

  public String message() {
    return "Hello World!";
  }
}
출처: https://7942yongdae.tistory.com/177 [프로그래머 YD:티스토리]
```

이러한 방식은 Controller와 Service와 직접적인 강한 관계가 있고 이는 프로그램의 유연성을 떨어트린다.

controller와 service와의 관계를 직접적이 아닌 간접적인 관계로 만든다→ 의존 객체 주입

```
public class Main {

  public static void main(String[] args) {
    Controller controller = new Controller(new MessageService());
    controller.print();
  }
}

interface IService {

  String message();
}

class Controller {

  private IService service;

  public Controller(IService service) {
    this.service = service;
  }

  public void print() {
    System.out.println(service.message());
  }
}

class MessageService implements IService {

  public String message() {
    return "Hello World!";
  }
}
출처: https://7942yongdae.tistory.com/177 [프로그래머 YD:티스토리]
```

이번에는 controller가 service 객체를 생성하지 않고 그저 interface인 Iservice 객체를 사용했다

controller 객체는 messageservice 객체를 몰라도 iservice 객체를 이용하여 출력한다

→"객체(MessageService)는 내가(Main) 만들게 넌 사용(IService) 해"

main에 messageserivce 주입

https://7942yongdae.tistory.com/177

직접적인 관계를 갖던 클래스를 간접적인 관계를 갖게해주고

필요한 자원을 생성자에 넘겨주는 의존객체 주입을 이용하자.

유연성, 재사용성, 테스트 용이성 개선한다.

자바에서는 이렇게 활용이 됐는데 SOLID 법칙의 DIP를 지키기 위해선 DI가 필수적이다

Effective Java

AppConfig

- 구현객체를 생성하고 연결을 책임진다

```
package hello.core.member;

public class MemberServiceImpl implements MemberService {
    private final MemberRepository memberRepository;

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

    public void join(Member member) {
        memberRepository.save(member);
    }
    public Member findMember(Long memberId) {
        return memberRepository.findById(memberId);
    }
}
```

```
public class AppConfig {

    public MemberService memberService() {
        return new MemberServiceImpl(new MemoryMemberRepository());
    }
}
```

AppConfig 클래스

팩토리 메소드를 통하여 MemberServiceImpl에 MemoryMemberRepository를 주입시킨다

```
package hello.core.member;

public class MemberServiceImpl implements MemberService {
    private final MemberRepository memberRepository = new
            MemoryMemberRepository();
    public void join(Member member) {
        memberRepository.save(member);
    }
    public Member findMember(Long memberId) {
        return memberRepository.findById(memberId);
    }
}
--------
기존 코드
    private final MemberRepository memberRepository = new
            MemoryMemberRepository();
이렇게 MemoryMemberRepository()의 코드를 알게 되며 의존성이있는데

private final MemberRepository memberRepository;
로 의존성을 인터페이스에만 할당해주고
//설계 변경으로 MemberServiceImpl 은 MemoryMemberRepository 를 의존하지 않는다!

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

생성자를 생성하여
//MemberServiceImpl 입장에서 생성자를 통해 어떤 구현 객체가 들어올지(주입될지)는 알 수 없다.
외부 appconfig에서 memberRepository 를 MemoryMemberRepository로 주입
```

- private final MemberRepository memberRepository = new MemoryMemberRepository();

를

private final MemberRepository memberRepository;

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

생성자 만들기

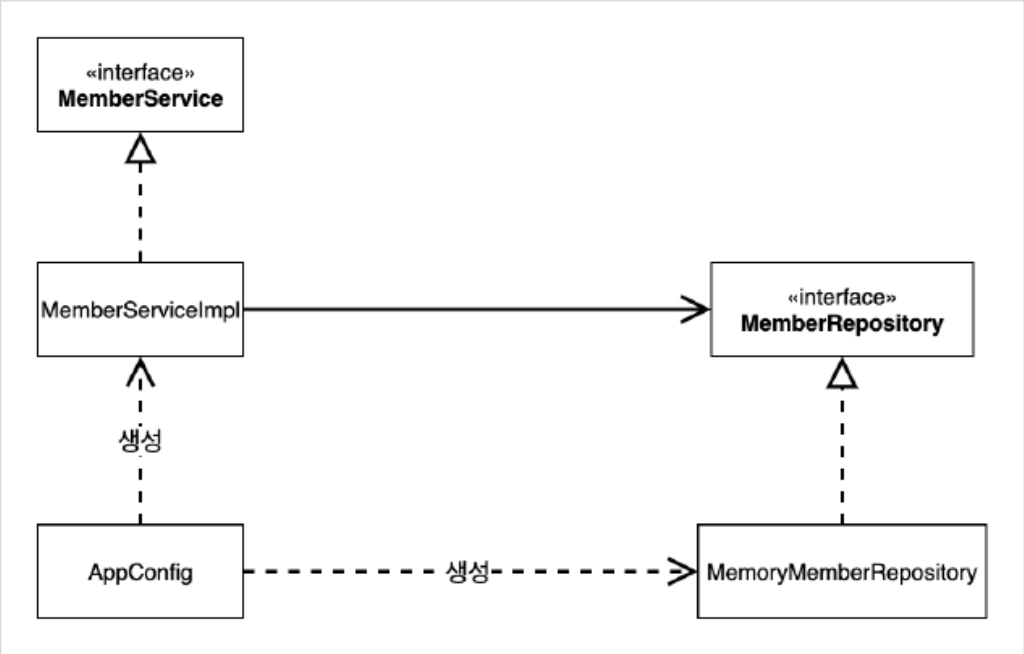
```
package hello.core.member;

public class MemberServiceImpl implements MemberService {
    private final MemberRepository memberRepository;

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

    public void join(Member member) {
        memberRepository.save(member);
    }
    public Member findMember(Long memberId) {
        return memberRepository.findById(memberId);
    }
}
```

```
public class AppConfig {
public MemberService memberService() {
return new MemberServiceImpl(new MemoryMemberRepository());
}
public OrderService orderService() {
return new OrderServiceImpl(
new MemoryMemberRepository(),
new FixDiscountPolicy());
}
}
```

- AppConfig는 애플리케이션의 실제 동작에 필요한 **구현 객체를 생성**한다.
  - `MemberServiceImpl`
  - `MemoryMemberRepository`
  - `OrderServiceImpl`
  - `FixDiscountPolicy`
- AppConfig는 생성한 객체 인스턴스의 참조(레퍼런스)를 **생성자를 통해서 주입(연결)**해준다.
  - `MemberServiceImpl` ➡ `MemoryMemberRepository`
  - `OrderServiceImpl` ➡ `MemoryMemberRepository`, `FixDiscountPolicy`

```
public class MemberServiceImpl implements MemberService {
private final MemberRepository memberRepository;
public MemberServiceImpl(MemberRepository memberRepository) {
this.memberRepository = memberRepository;
}
public void join(Member member) {
memberRepository.save(member);
}
public Member findMember(Long memberId) {
return memberRepository.findById(memberId);
}
}
```

- 설계 변경으로 `MemberServiceImpl` 은 `MemoryMemberRepository` 를 의존하지 않는다!
- 단지 `MemberRepository` 인터페이스만 의존한다.
- `MemberServiceImpl` 입장에서 생성자를 통해 어떤 구현 객체가 들어올지(주입될지)는 알 수 없다.
- `MemberServiceImpl` 의 생성자를 통해서 어떤 구현 객체를 주입할지는 오직 외부(`AppConfig`)에서 결정된다.
- `MemberServiceImpl` 은 이제부터 **의존관계에 대한 고민은 외부**에 맡기고 **실행에만 집중**하면 된다.



- 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.

AppConfig로 의존관계를 주입한다 (Dependency Injection)

```
public class OrderApp {

    public static void main(String[] args) {

        AppConfig appConfig = new AppConfig();
        MemberService memberService = appConfig.memberService();
        OrderService orderService= appConfig.orderService()
       // MemberService memberService = new MemberServiceImpl(null);
       // OrderService orderService = new OrderServiceImpl(null,null);

        Long memberId = 1L;
        Member member = new Member(memberId, "memberA", Grade.VIP);
        memberService.join(member);

        Order order = orderService.createOrder(memberId, "itemA", 10000);

        System.out.println("order = " + order);
    }
}
```

정리

- AppConfig를 통해서 관심사를 확실하게 분리했다.
- 배역, 배우를 생각해보자.
- AppConfig는 공연 기획자다.
- AppConfig는 구체 클래스를 선택한다. 배역에 맞는 담당 배우를 선택한다. 애플리케이션이 어떻게 동작해야 할지 전체 구성을 책임진다.
- 이제 각 배우들은 담당 기능을 실행하는 책임만 지면 된다.
- `OrderServiceImpl`은 기능을 실행하는 책임만 지면 된다.

▼ AppConfig 리팩터링

역할이 한눈에 보이는 게 중요하다. 이렇게 봐꿔보자

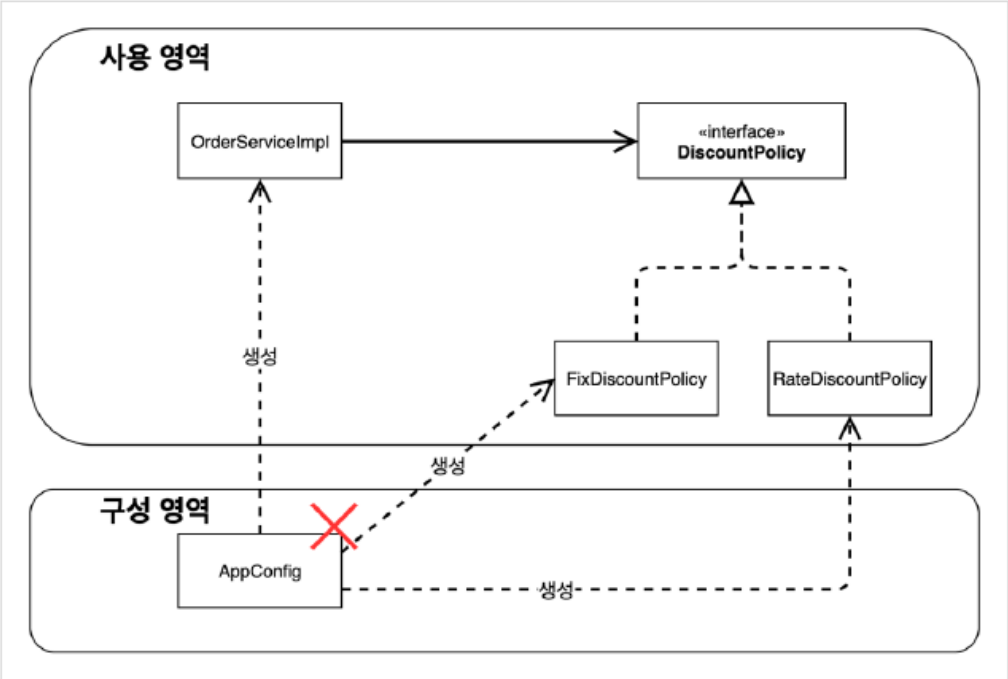
```
package hello.core;

import hello.core.discount.DiscountPolicy;
import hello.core.discount.FixDiscountPolicy;
import hello.core.member.MemberService;
import hello.core.member.MemberServiceImpl;
import hello.core.member.MemoryMemberRepository;
import hello.core.order.OrderService;
import hello.core.order.OrderServiceImpl;

public class AppConfig {

    public MemberService memberService() {
        return new MemberServiceImpl(memberRepository());
    }

    private MemoryMemberRepository memberRepository() {
        return new MemoryMemberRepository();
    }

    public OrderService orderService(){
        return new OrderServiceImpl(new MemoryMemberRepository(),new FixDiscountPolicy());
    }

    public DiscountPolicy discountPolicy(){
        return new FixDiscountPolicy();
    }
}
```

이렇게 구현하면 이것만 보고 설계에 대한 전체적인 것을 한눈에 알 수 있다

- `new MemoryMemberRepository()` 이 부분이 중복 제거되었다. 이제 `MemoryMemberRepository`를 다른 구현체로 변경할 때 한 부분만 변경하면 된다.
- `AppConfig`를 보면 역할과 구현 클래스가 한눈에 들어온다. 애플리케이션 전체 구성이 어떻게 되어있는지 빠르게 파악할 수 있다.

▼ 할인 정책 바꾸기 (OCP와 DIP를 만족시키는 AppConfig)

그림 - 할인 정책의 변경



- `FixDiscountPolicy` ➡ `RateDiscountPolicy`로 변경해도 구성 영역만 영향을 받고, 사용 영역은 전혀 영향을 받지 않는다.

```
appconfig 에서

public DiscountPolicy discountPolicy(){
       // return new FixDiscountPolicy();
        return new RateDiscountPolicy();
    }
```

- `AppConfig`에서 할인 정책 역할을 담당하는 구현을 `FixDiscountPolicy` ➡ `RateDiscountPolicy` 객체로 변경했다.
- 이제 할인 정책을 변경해도, 애플리케이션의 구성 역할을 담당하는 AppConfig만 변경하면 된다. 클라이언트 코드인 `OrderServiceImpl`를 포함해서 **사용 영역**의 어떤 코드도 변경할 필요가 없다.
- **구성 영역**은 당연히 변경된다. 구성 역할을 담당하는 AppConfig를 애플리케이션이라는 공연의 기획자로 생각하자. 공연 기획자는 공연 참여자인 구현 객체들을 모두 알아야 한다.

Appconfig만 바꾸면 된다 ~

이제 사용 영역에 있는 것은 전혀 손댈 필요가 없다

AppConfig를 통하여 클라이언트 코드를 손댈필요가 없으므로

이제 DIP와 OCP 둘다 지켜진다.

▼ SOLID 정리

SRP 단일 책임 원칙

- 하나의 클래스는 하나의 책임만 가져야한다

클라이언트 객체가 다양한 책임을 가지고 있었는데

- AppConfig 를 생성하여 구현 객체를 생성하고 연결
- 클라이언트 객체는 실행하는 책임만 가짐

DIP 의존관계 역전 원칙

- 프로그래머는 인터페이스에 의존해야하지 구체화에 의존하면 안된다
- AppConfig로 DI 개념을 활용하여 해결

OCP

소프트웨어 요소는 확장에는 열려있으나 변경에는 닫혀 있어야 한다

클라이언트 코드를 변경하지 않고 AppConfig의 변경을 통하여 이를 해결

▼ IoC, DI, 컨테이너

- IOC(Inversion Of Control) 제어의 역전

OrderServiceImpl 등 인터페이스만 실행하고 어떤 구현 객체들이 실행될지 모른다

>AppConfig가 제어의 흐름을 가져간다

프로그램을 직접 제어 하는것이 아닌 외부에서 관리하는 것을 제어의 역전IOC라고 부른다

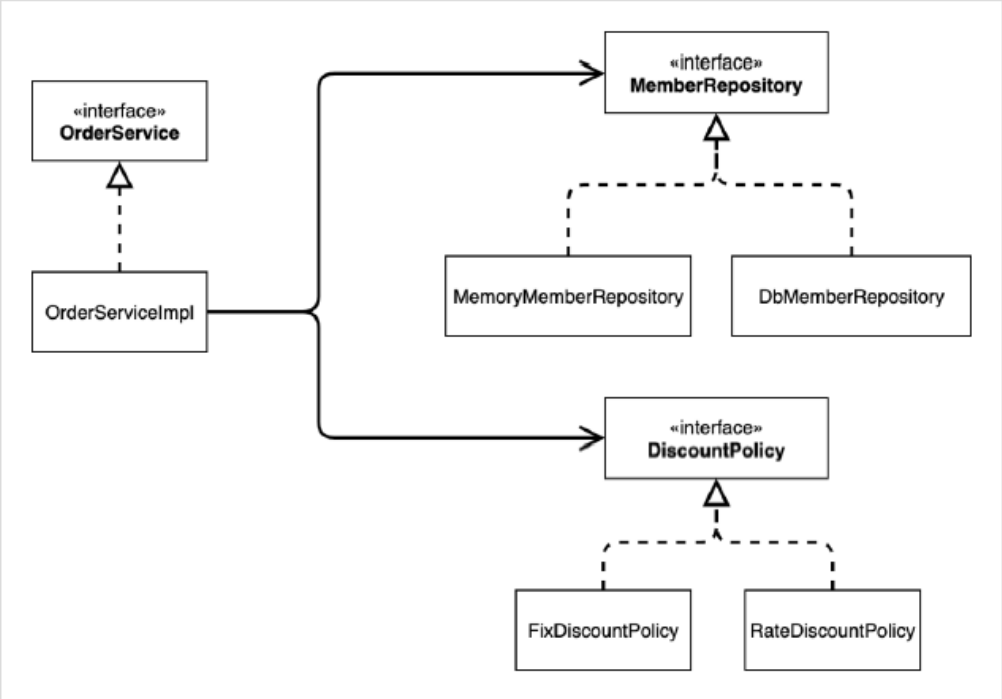
프레임워크

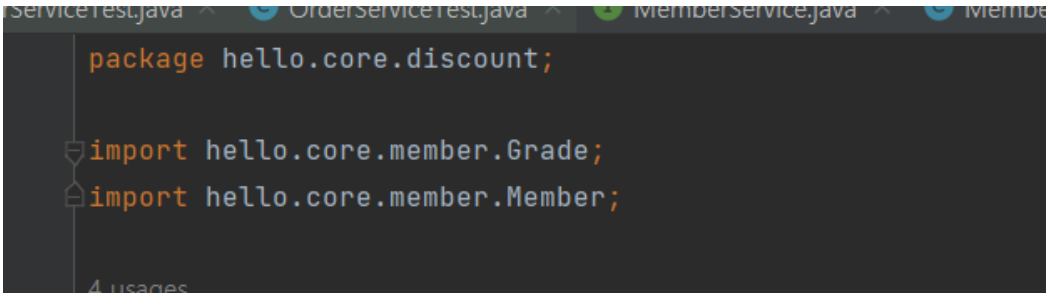
- 내가 작성한 코드를 제어하고 대신 실행 (JUnit)
- 프레임워크의 생태계 안에서 내 코드가 실행

라이브러리

- 내가 작성한 코드가 직접 제어의 흐름을 담당(JSoup)
- 가져다가 쓸때, 이용할때

- 의존관계 주입 DI(Dependency Injection)
  - 정적인 클래스 의존관계
    - 클래스가 사용하는 import 관계만 봐서 바로 쉽게 판단할수 있는것





- 동적인 객체 인스턴스 의존관계
  - 실행 시점(런타임)에서 객체 생성하고 연결되는것
  - 클라이언트 코드를 변경하지 않고 클ㄹ라이언트가 호출하는 대상의 타입의 인스턴스르 변경 가능
  - 정적인 클래스 의존관계를 변경하지 않고 (클래스 다이어그램을 손대지 않고) 동적인 객체 인스턴스 의존관계를 쉽게 변경할 수 있다

- IOC 컨테이너, **DI 컨테이너**
  - AppConfig 처럼 객체 생성하고 관리하면서 의존관계 연결해주는것

  어셈블러, 오브젝트 팩토리라고도 불린다

▼ 스프링으로 전환하기

```
//설정정보
@Configuration
public class AppConfig {

    //spring container에 등록된다
    @Bean
    public MemberService memberService() {
        return new MemberServiceImpl(memberRepository());
    }
    @Bean
    public MemoryMemberRepository memberRepository() {
        return new MemoryMemberRepository();
    }
    @Bean
    public OrderService orderService(){
        return new OrderServiceImpl(new MemoryMemberRepository(),new FixDiscountPolicy());
    }
    @Bean
    public DiscountPolicy discountPolicy(){
       // return new FixDiscountPolicy();
        return new RateDiscountPolicy();
    }
}
```

```
public class MemberApp {
    public static void main(String[] args) {
//        AppConfig appConfig = new AppConfig();
//        MemberService memberService = appConfig.memberService();


        //Appconfig에 있는 Bean에 있는것을 다 집어넣어서 관리해준다
        //원래는 직접찾아갔는데 이제는 context로
        ApplicationContext applicationContext = new
                AnnotationConfigApplicationContext(AppConfig.class);

        MemberService memberService =
                applicationContext.getBean("memberService", MemberService.class);
        //bean에 method이름으로 저장된다


        //memberServiceimpl이 들어있다 그 안에 의존관계
        Member member = new Member(1L, "memberA", Grade.VIP);
        memberService.join(member);
        Member findMember = memberService.findMember(1L);
        System.out.println("new member = " + member.getName());
        System.out.println("find Member = " + findMember.getName());
    }
}
```

ApplicationContext 를 스프링 컨테이너라 한다.

기존에는 AppConfig를 사용해서 직접 객체 생성하고 DI를 했지만 이제는 ApplicaitonContext를 이용하여 스프링 컨테이너 통해서 사용

@Configuration이 붙은 AppConfig를 설정정보, @Bean 적힌 메서드 모두 호출하여 스프링 컨테이너에 등록 : 스프링 빈

스프링 컨테이너를 통하여 빈 찾고 applicaitonContext.getBean()이용하여 찾는다

기존에 개발자 자바코드에서 모든것을 했다면 이제

- 스프링 컨테이너에 객체를 스프링 빈으로 등록하고
- 스프링 컨테이너에서 스프링 빈을 찾아서 사용

▼ 정리

JAVA 코드만을 사용하여 코드를 구현해보았고 SOLID 법칙중 DIP와 OCP가 지켜지지 않는 것을 확인하였다

이를 지키기 위하여 AppConfig를 만들어서 사용과 구성의 영역을 나눠서 효율적으로 코드를 짰고 DIP와 OCP를 지켰다

허나 Spring을 이용하여 스프링컨테이너에 객체를 넣고 사용하는게 AppConfig보다 더 좋다는데 왜 좋은걸까 ?

엄청나게 장점이 많다는데,

이후 공부할 내용