2. 스프링 핵심 원리 이해

Created	@October 30, 2022 2:07 PM
Progress	In Progress

예제 만들기 2.1. 2.2.

- 프로젝트 생성
- 비즈니스 요구사항과 설계
- 회원 도메인 설계/개발/실행과 테스트
- 주문과 할인 도메인 설계/개발/실행과 테스트

객체 지향 **원리 적용** 2.3.

- 새로운 할인 정책 개발
- 관심사의 분리
- AppConfig 리팩터링
- 새로운 구조와 할인 정책 적용
- 좋은 객체 지향 설계의 5가지 원칙(SOLID) 적용
- IoC, DI, 컨테이너
- 스프링으로 전환하기

학습 TODO list

- [ ] static, final

2.1. 예제 만들기

> *순수 자바를 사용해서 먼저 개발해보자.*

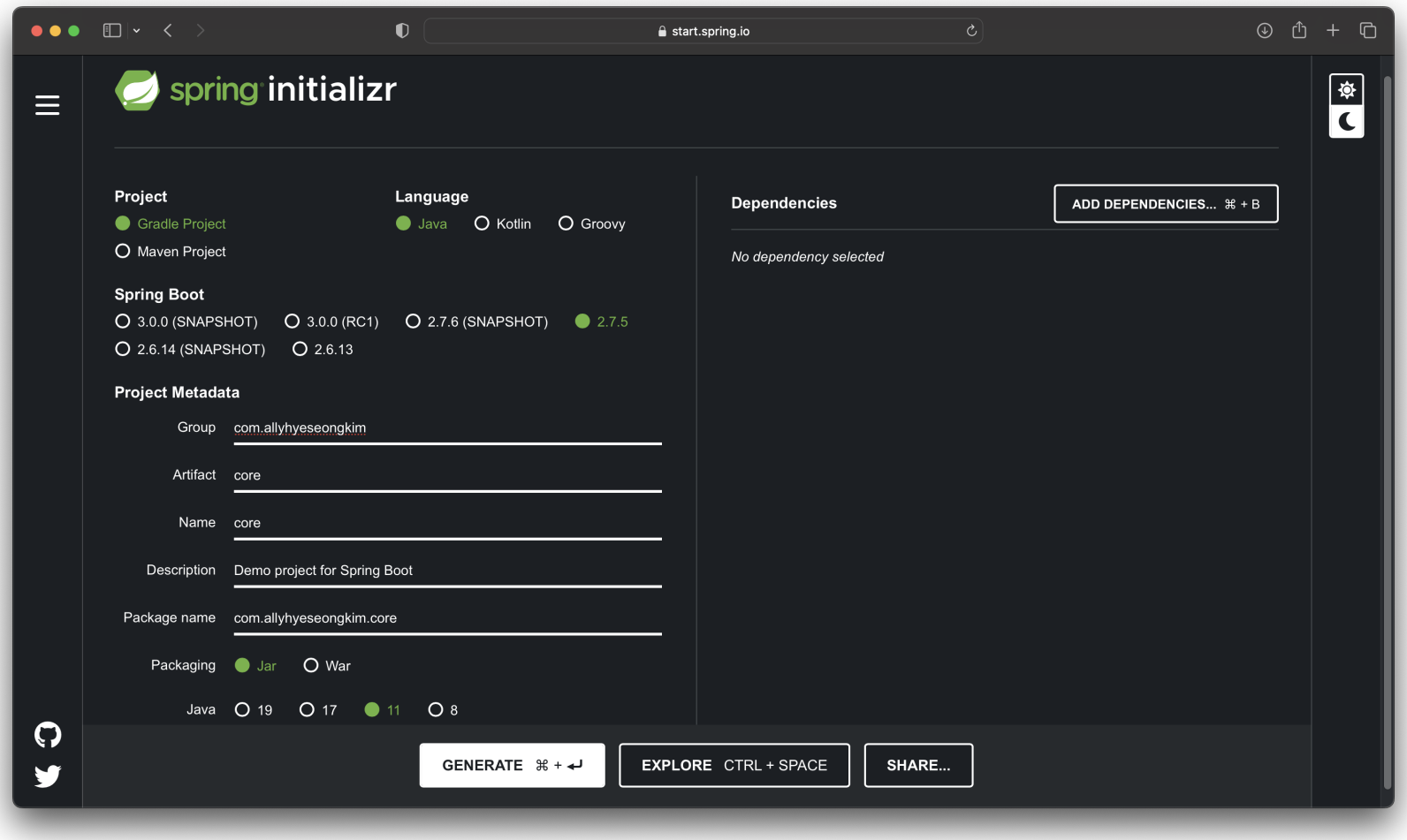
2.1.1. 프로젝트 생성

Spring Initializr

Initializr generates spring boot project with just what you need to start quickly!

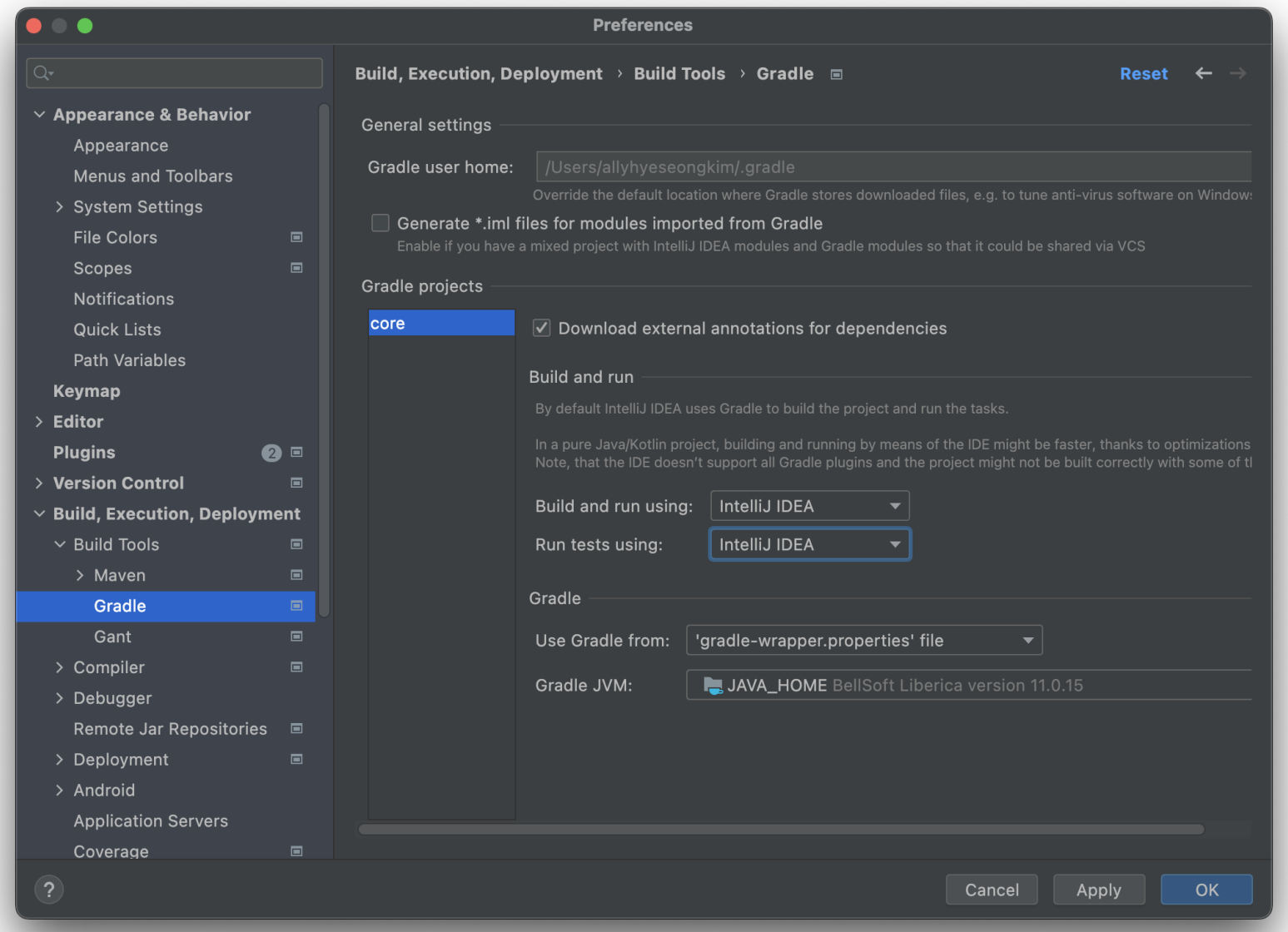
https://start.spring.io/

- 프로젝트 설정



Spring Boot 가장 안정된 최신 버전: 2.7.5 (SNAPSHOT 등이 없는 버전)

- 생성된 프로젝트 파일에서 `build.gradle`을 `IntelliJ`에서 `Open as Project`로 열어준다.
- `Preference > Build, Execution, Deployment > Build Tools > Gradle`에서 `Build, Run, Run test`를 `IntelliJ IDEA`를 사용하도록 변경한다.



- 최근 `IntelliJ` 버전은 `Gradle`을 통해서 실행 하는 것이 기본 설정이다. 하지만 현재는 `IntelliJ`를 통해서 자바로 바로 실행하는게 실행 속도가 더 빠르다.

2.2. 비즈니스 요구사항과 설계

> *요구사항을 보면 `회원` 데이터, `할인 정책` 같은 부분은 지금 결정하기 어렵다. 그렇다고 정책이 결정될 때까지 개발을 무기한 기다릴 수 없으므로 `객체 지향 설계` 방법을 적용해보자. → 인터페이스를 만들고 구현체를 언제들지 바꿀*

> *수 있도록 설계한다.*

👩‍💻 `회원` 도메인 요구사항

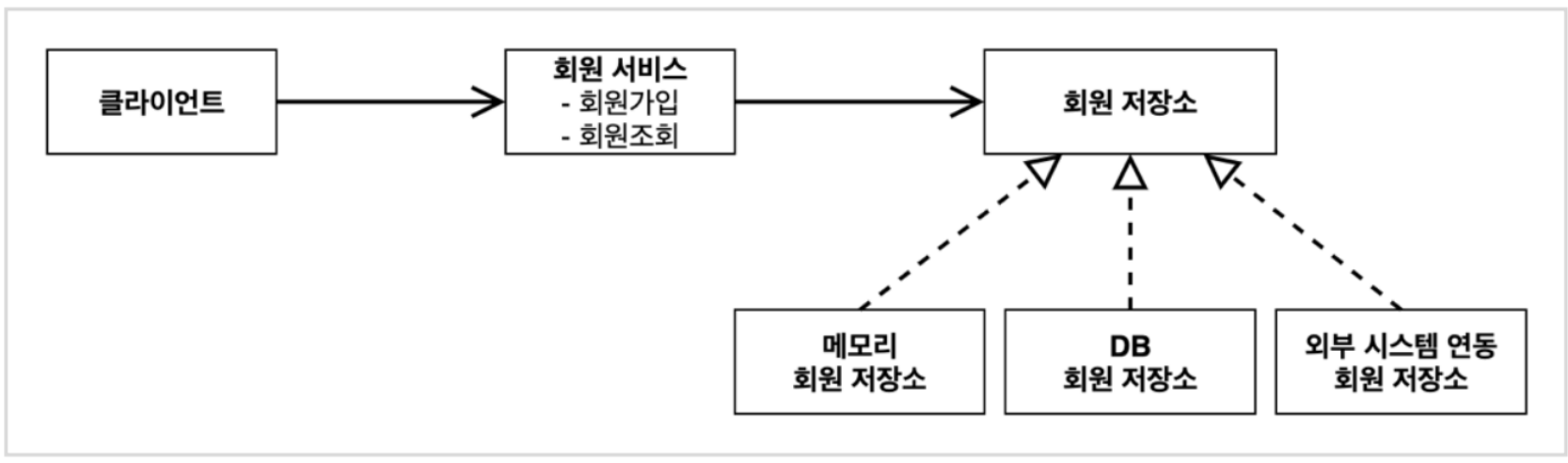
- `회원 가입`하고 `조회`할 수 있다.
- 회원은 `일반`과 `VIP` 두 가지 등급이 있다.
- 회원 데이터는 자체 DB를 구축할 수 있고, 외부 시스템과 연동할 수 있다.(미확정)

👩‍💻 `주문`과 `할인 정책` 도메인 요구사항

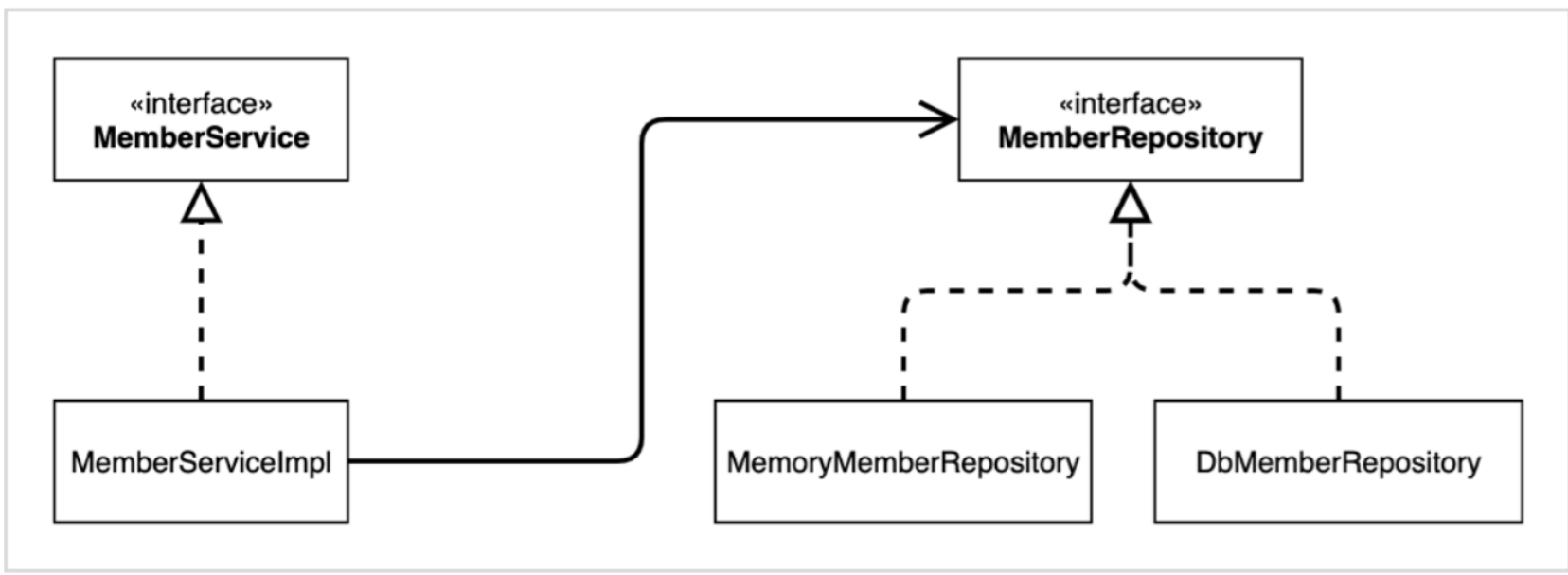
- 회원은 `상품`을 주문할 수 있다.
- 회원 등급에 따라 할인 정책을 적용할 수 있다.
- 할인 정책은 모든 `VIP`는 `1000원`을 할인해주는 고정 금액 할인을 적용한다.(나중에 변경될 수 있음)
  - 할인 정책은 변경 가능성이 높다. 회사의 기본 할인 정책을 아직 정하지 못했고, 오픈 직전까지 고민을 미루고 싶다. 최악의 경우 할인을 적용하지 않을 수도 있다.(미확정)

2.2.1. 회원 도메인 설계

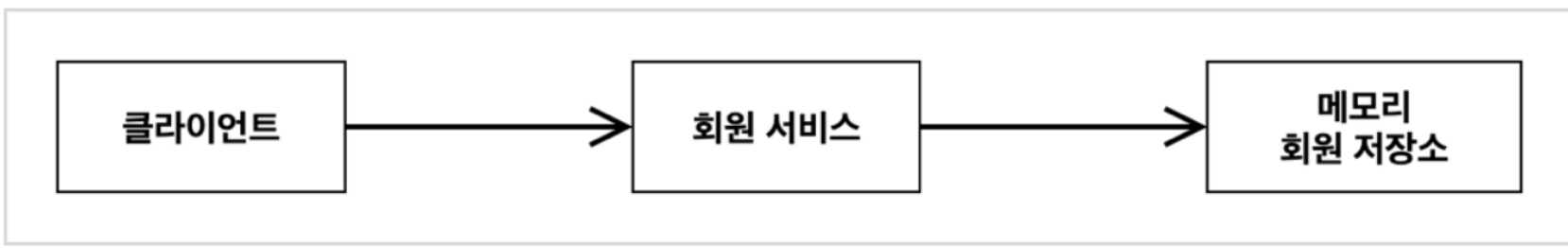
- `회원` 도메인 `협력` 관계



  - 메모리 회원 저장소: DB 결정 전까지 개발용
- `회원` 클래스 다이어그램



- `회원` 객체 다이어그램



회원 서비스 = MemberServiceImpl

2.2.2. 회원 도메인 개발

- `회원` 엔티티

```
package com.allyhyeseongkim.core.member;

public enum Grade {
  BASIC,
  VIP
}
```

```
package com.allyhyeseongkim.core.member;

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 this.id;
  }

  public void setId(Long id) {
    this.id = id;
  }

  public String getName() {
    return this.name;
  }

  public void setName(String name) {
    this.name = name;
  }

  public Grade getGrade() {
    return this.grade;
  }

  public void setGrade(Grade grade) {
    this.grade = grade;
  }
}
```

- 회원 저장소

```
package com.allyhyeseongkim.core.member;

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

```
package com.allyhyeseongkim.core.member;

import java.util.HashMap;
import java.util.Map;

public class MemoryMemberRepository implements MemberRepository {
  private static Map<Long, Member> store = new HashMap<>();
  // private static Map<Long, Member> store = new ConcurrentHashMap<>();

  @Override
  public void save(Member member) {
    store.put(member.getId(), member);
  }

  @Override
  public Memer findById(Long memberId) {
    return store.get(memberID);
  }
}
```

  - `HashMap`은 동시성 이슈가 발생할 수 있다. → `ConcurrentHashMap`을 사용하여 해결할 수 있다.(실무)
  - `static`으로 선언한 이유: 단 하나의 `store`를 모든 `MemoryMemberRepository` 인스턴스가 공유한다.
    - `static` field나 method는 **인스턴스 소유가 아닌 클래스 소유가 된다.**

- 회원 서비스

```
package com.allyhyeseongkim.core.member;

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

```
package com.allyhyeseongkim.core.member;

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

```
  public void join(Member member) {
    this.memberRepository.save(member);
  }

  public Member findMember(Long memberId) {
    return this.memberRepository.findById(memberId);
  }
}
```

- `final`로 선언한 이유: 불변 객체로 선언하여 생성자 주입 이후 변경이 불가능하도록 하여 추후 변경 여지를 막는다.

2.2.3. 회원 도메인 실행과 테스트

- `회원` 도메인

```
package com.allyhyeseongkim.core;

import com.allyhyeseongkim.core.member.Grade;
import com.allyhyeseongkim.core.member.Member;
import com.allyhyeseongkim.core.member.MemberService;
import com.allyhyeseongkim.core.member.MemberServiceImpl;

public class MemberApp {
  public static void main(String[] args) {
    MemberService memeberService = 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());
  }
}
```

> 출력 결과:
>
> *`new member = memberA`*
> *`find member = memberA`*

- 애플리케이션 로직으로 테스트하는 것은 좋은 방법이 아니다. JUnit 테스트를 사용해서 테스트해야 한다.

```
package com.allyhyeseongkim.core.member;

import org.assertj.core.api.Assertions;
import org.junit.jupiter.api.Test;

class MemberServiceTest {
  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);
  }
}
```

👩‍💻 `회원` 도메인 설계의 문제점

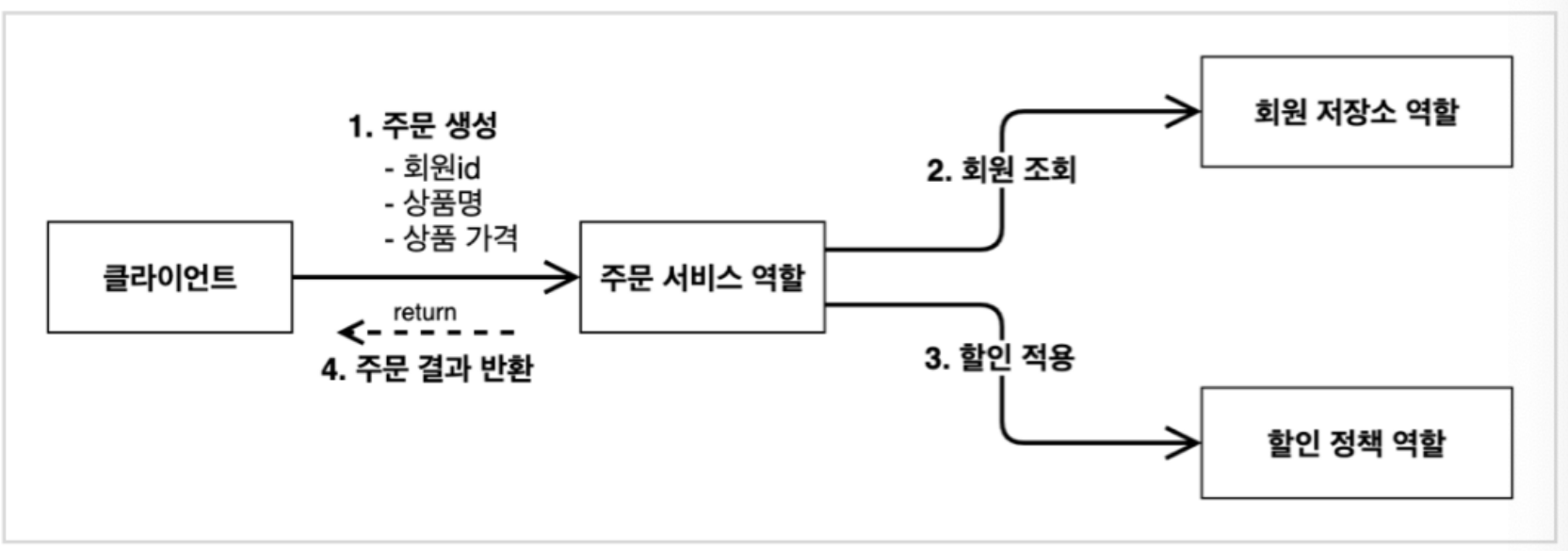
- 의존관계가 인터페이스 뿐만 아니라 구현까지 모두 의존하는 문제점이 있다.

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

  - `OCP`, `DIP 원칙`을 위반한다.

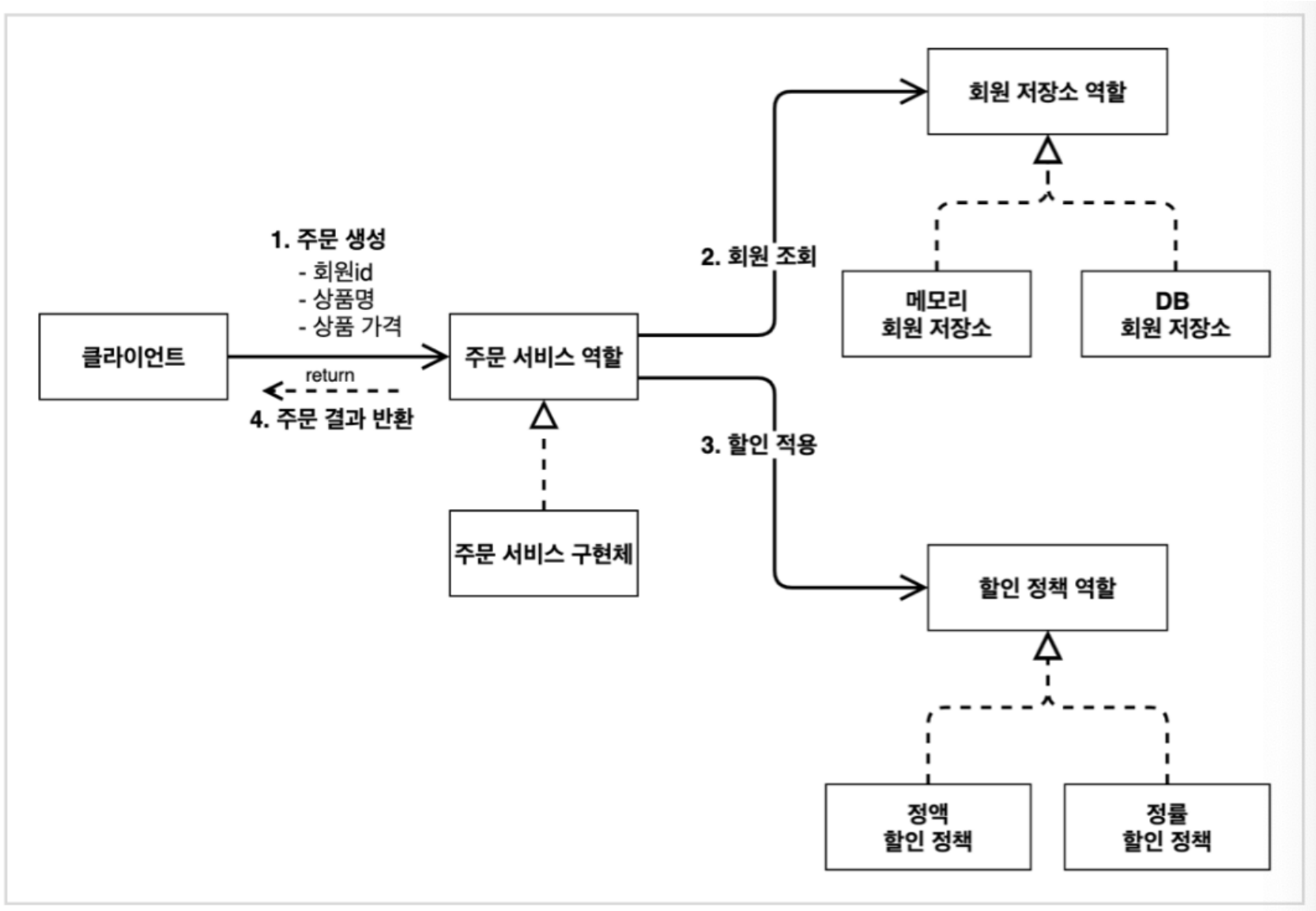
2.2.4. 주문과 할인 도메인 설계

- `주문` 도메인 `협력`, `역할`, `책임`

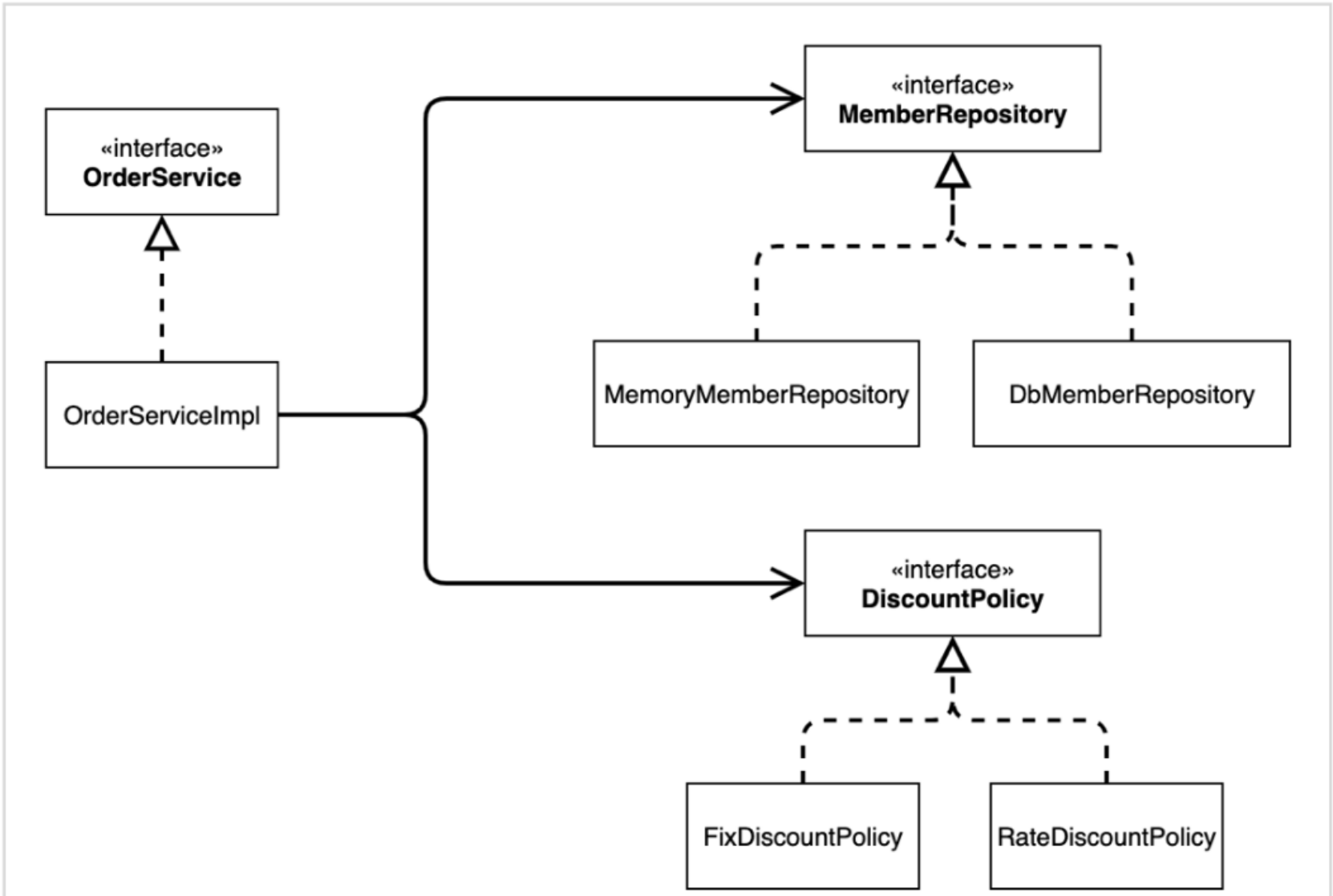


1. 클라이언트는 주문 서비스에 주문 생성을 요청한다.
2. 할인을 위해서는 회원 등급이 필요하다. 그래서 주문 서비스는 회원 저장소에서 회원을 조회한다.
3. 주문 서비스는 회원 등급에 따른 할인 여부를 할인 정책에 위임한다.
4. 주문 서비스는 할인 결과를 포함한 주문 결과를 반환한다.

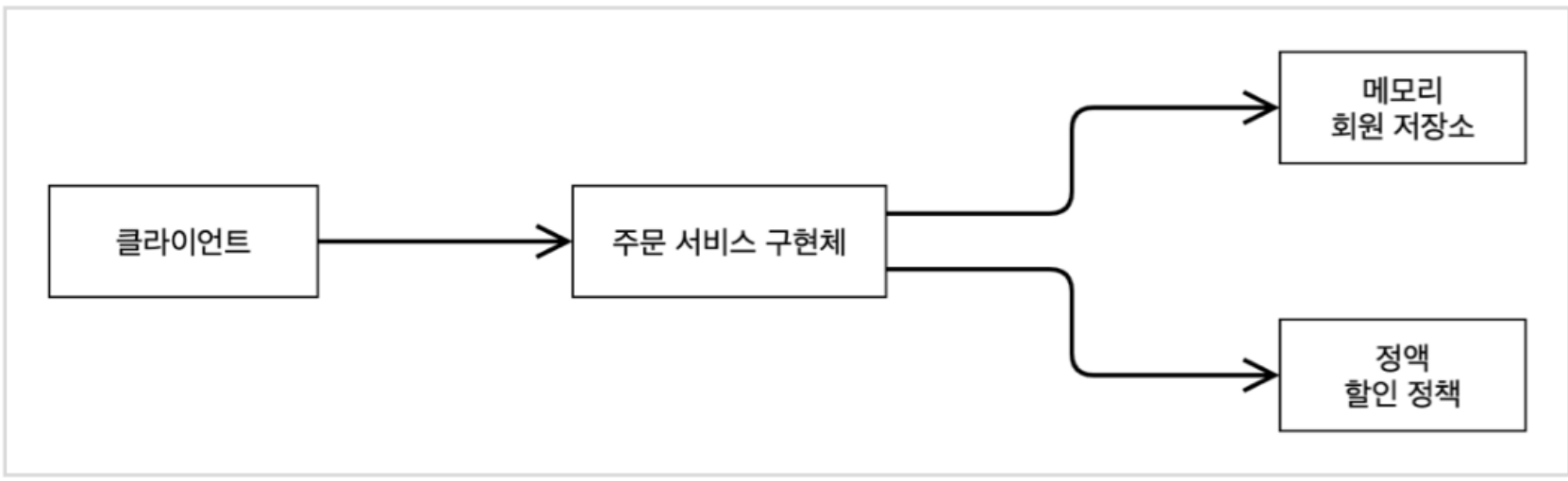
- 주문 도메인 전체

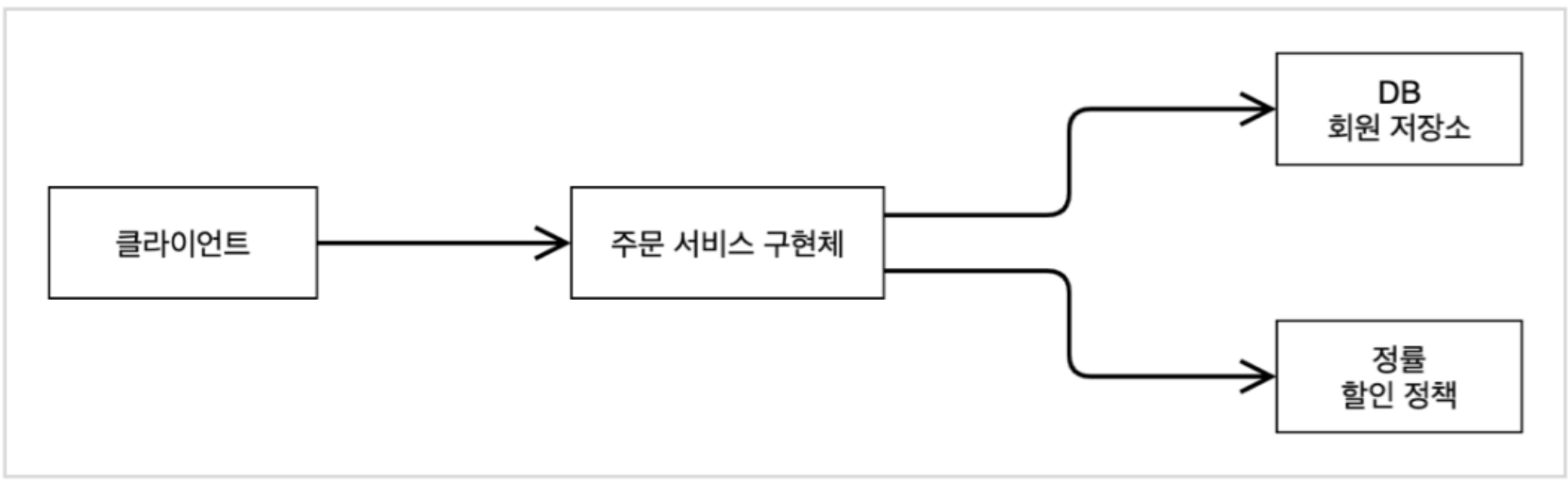


  - **역할**과 **구현**을 분리하여 자유롭게 구현 객체를 조립할 수 있다.
- 주문 도메인 클래스 다이어그램



- 주문 도메인 객체 다이어그램





역할들의 협력 관계를 그대로 재사용 할 수 있다.

2.2.5. 주문과 할인 도메인 개발

- 할인 정책 도메인

```
package com.allyhyeseongkim.core.discount;

import com.allyhyeseongkim.core.member.Member;

public interface DiscountPolicy {
  int discount(Member member, int price);
}
```

```
package com.allyhyeseongkim.core.discount;

import com.allyhyeseongkim.core.member.Grade;
import com.allyhyeseongkim.core.member.Member;

public class FixDiscountPolicy implements DiscountPolicy {
  int discountFixAmount = 1000;  //1000원 할인

  @Override
  public int discount(Member member, int price) {
    if (member.getGrade() == Grade.VIP) {
      return discountFixAmount;
    } else {
      return 0;
    }
  }
}
```

  - `enum` type은 `==`으로 비교할 수 있다.

- `주문` 도메인

```
package com.allyhyeseongkim.core.order;

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 this.itemPrice - this.discountPrice;
 }

  public Long getMemberId() {
    return this.memberId;
  }

  public String getItemName() {
    return this.itemName;
  }

  public int getItemPrice() {
    return this.itemPrice;
  }

  public int getdiscountPrice() {
    return this.discountPrice;
  }

  @Override
  public String toString() {
    return "Order{" + "memberId=" + memberId + ", itemName=" + itemName + "\'" + ", itemPrice=" + itemPrice + ", discountPrice=" + discountPrice + "}";
  }
}
```

- `주문` 서비스

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

```
package com.allyhyeseongkim.core.order;

import com.allyhyeseongkim.core.discount.DiscountPolicy;
import com.allyhyeseongkim.core.discount.FixDiscountPolicy;
import com.allyhyeseongkim.core.member.Member;
import com.allyhyeseongkim.core.member.MemberRepository;
import com.allyhyeseongkim.core.member.MemoryMemberRepository;

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 = this.memberRepository.findById(memberId);
    int discountPrice = this.discountPolicy.discount(member, itemPrice);

    return new Order(memberId, itemName, itemPrice, discountPrice);
  }
}
```

  - **`discount` 기능 변경이 필요할 때 `OrderServiceImpl`에서 바꾸지 않는다. → `SRP`를 잘 지켜서 설계했다.**

2.2.6. 주문과 할인 도메인 실행과 테스트

- `주문`과 `할인 정책` 실행

```
package com.allyhyeseongkim.core;

import com.allyhyeseongkim.core.member.Grade;
import com.allyhyeseongkim.core.member.Member;
import com.allyhyeseongkim.core.member.MemberService;
import com.allyhyeseongkim.core.member.MemberServiceImpl;
import com.allyhyeseongkim.core.order.Order;
import com.allyhyeseongkim.core.order.OrderService;
import com.allyhyeseongkim.core.order.OrderServiceImpl;

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);
  }
}
```

> 출력 결과:
>
> *`order = Order{memberId=1, itemName='itemA', itemPrice=10000, discountPrice=1000}`*

    - 애플리케이션 로직으로 테스트하는 것은 좋은 방법이 아니다. JUnit 테스트를 사용해서 테스트해야 한다.

- `주문`과 `할인 정책` 테스트

```
package com.allyhyeseongkim.core.order;

import com.allyhyeseongkim.core.member.Grade;
import com.allyhyeseongkim.core.member.Member;
import com.allyhyeseongkim.core.member.MemberService;
import com.allyhyeseongkim.core.member.MemberServiceImpl;
import org.assertj.core.api.Assertions;
import org.junit.jupiter.api.Test;

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(10000);
  }
}
```

2.3. 객체 지향 원리 적용

2.3.1. 새로운 할인 정책 개발

- 새로운 `할인 정책`을 확장해보자.

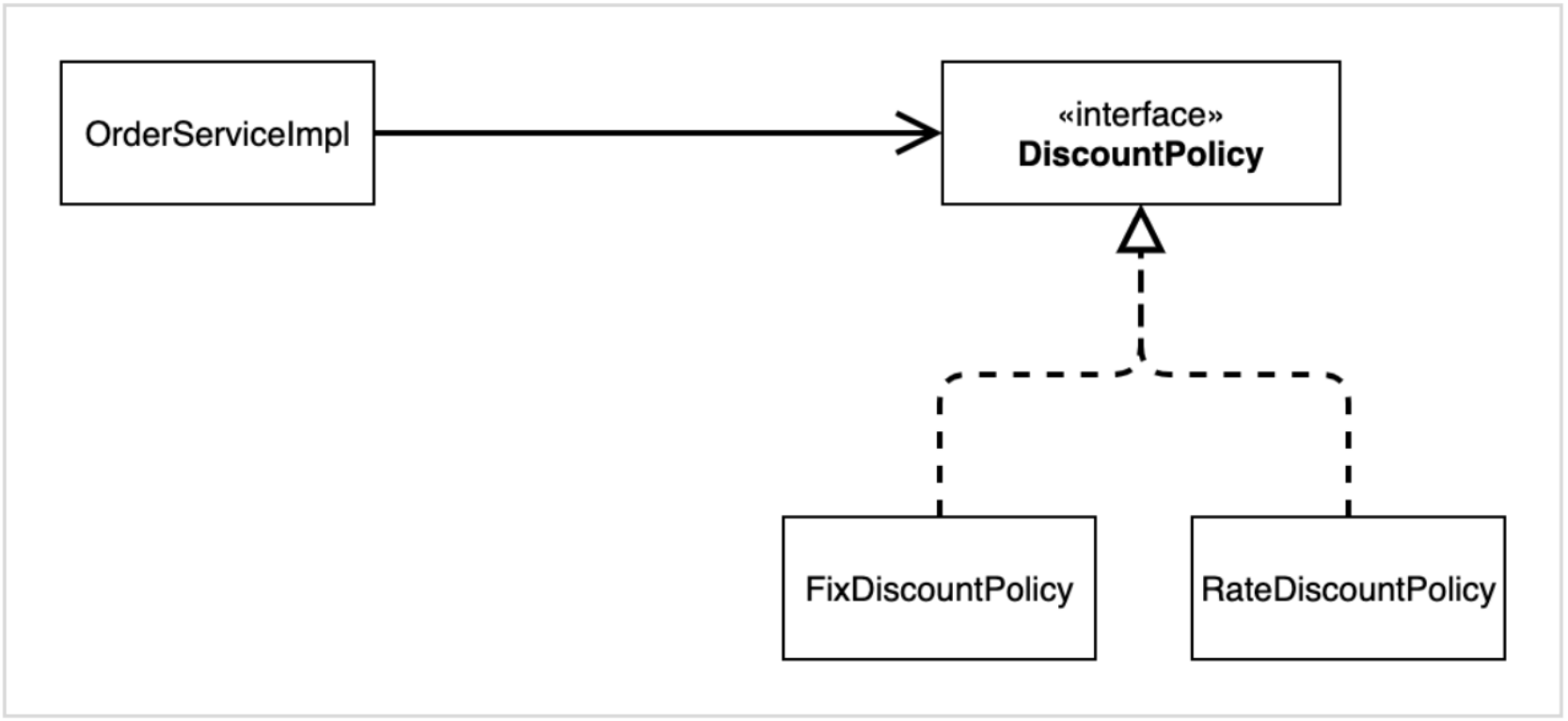
> 👩 서비스 오픈 직전에 할인 정책을 지금처럼 고정 금액 할인이 아니라 좀 더 합리적인 주문 금액당 할인하는 정률% 할인으로 변경하고 싶어요. 예를 들어서 기존 정책은 VIP가 10000원을 주문하든 20000원을 주문하든 항상 1000원을 할인했는데, 이번에 새로 나온 정책은 10%로 지정해두면 고객이 10000원 주문시 1000원을 할인해주고, 20000원 주문시에 2000원을 할인해주는 거예요!

> 👨‍💻 제가 처음부터 고정 금액 할인은 아니라고 했잖아요.

> 👩 애자일 소프트웨어 개발 선언 몰라요? “계획을 따르기보다 변화에 대응하기를”

> 👨‍💻 …(하지만 난 유연한 설계가 가능하도록 객체지향 설계 원칙을 준수했지 후후)

- 주문한 금액의 %를 할인해주는 새로운 정률 할인 정책을 추가하여 `객체지향 설계 원칙`을 잘 준수 했는지 확인해보자.
- `RateDiscountPolicy` 추가




```
package com.allyhyeseongkim.core.discount;
import com.allyhyeseongkim.core.member.Grade;
import com.allyhyeseongkim.core.member.Member;

public class RateDiscountPolicy implements DiscountPolicy {
  private int discountPercent = 10;  //10% 할인

  @Override
  public int discount(Member member, int price) {
    if (member.getGrade() == Grade.VIP) {
      return price * discountPercent / 100;
    } else {
      return 0;
    }
  }
}
```

- 테스트 작성

```

package com.allyhyeseongkim.core.discount;

import com.allyhyeseongkim.core.member.Grade;
import com.allyhyeseongkim.core.member.Member;
import org.junit.jupiter.api.DisplayName;
import org.junit.jupiter.api.Test;

import static org.assertj.core.api.Assertions.*;

class RateDiscountPolicyTest {
  DiscountPolicy discountPolicy = new RateDiscountPolicy();

  @Test
  @DisplayName("VIP는 10% 할인이 적용되어야 한다.")
  void vip_o() {
    //given
    Member member = new Member(1L, "memberVIP", Grade.VIP);
    //when
    int discount = this.discountPolicy.discount(member, 10000);
    //then
    assertThat(discount).isEqualTo(1000);
  }

  @Test
  @DisplayName("VIP가 아니면 할인이 적용되지 않아야 한다.")
  void vip_x() {
    //given
    Member member = new Member(2L, "memberBASIC", Grade.BASIC);
    //when
    int discount = this.discountPolicy.discount(member, 10000);
    //then
    assertThat(discount).isEqualTo(0);
  }
}
```

  - `Assertions`는 자주사용하므로 `static`으로 두는게 좋다.

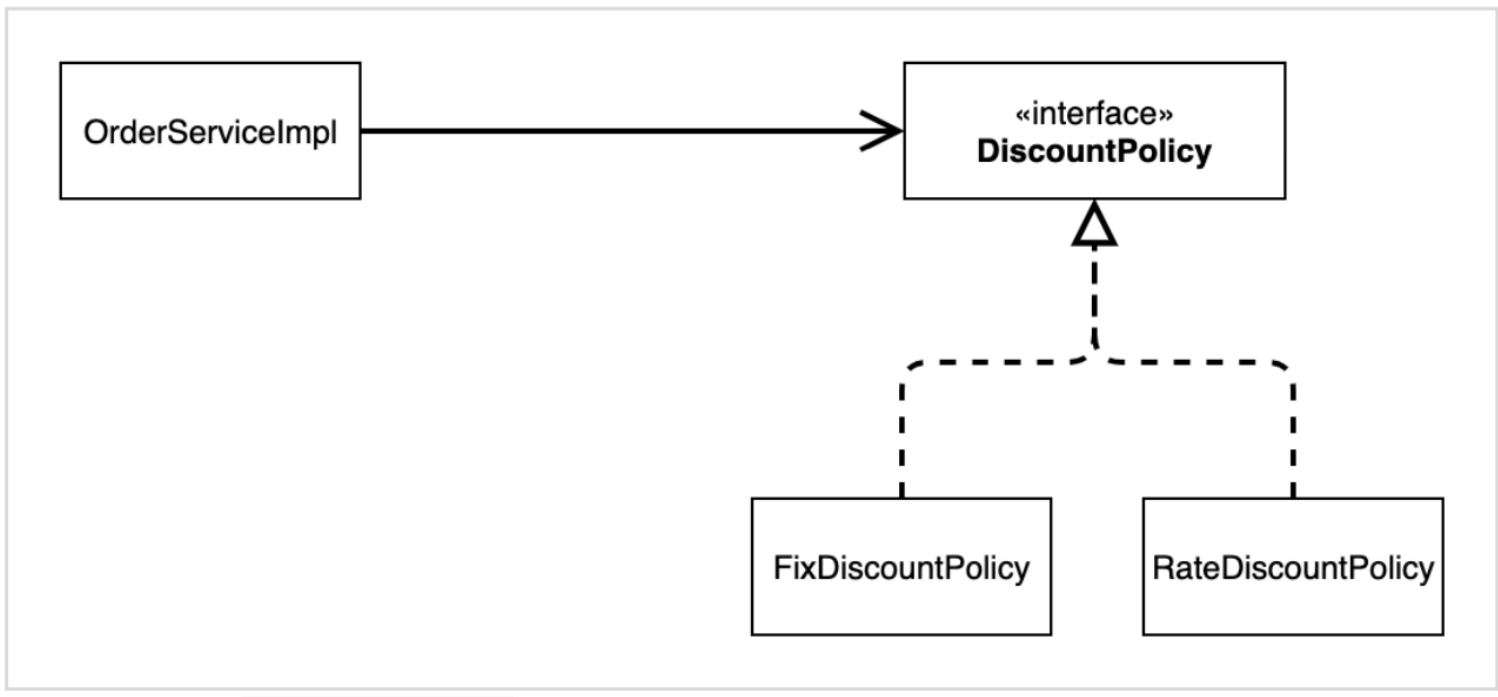
2.3.2. 새로운 할인 정책 적용과 문제점

- 할인 정책을 애플리케이션에 적용해보자.

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

👩‍💻 문제점

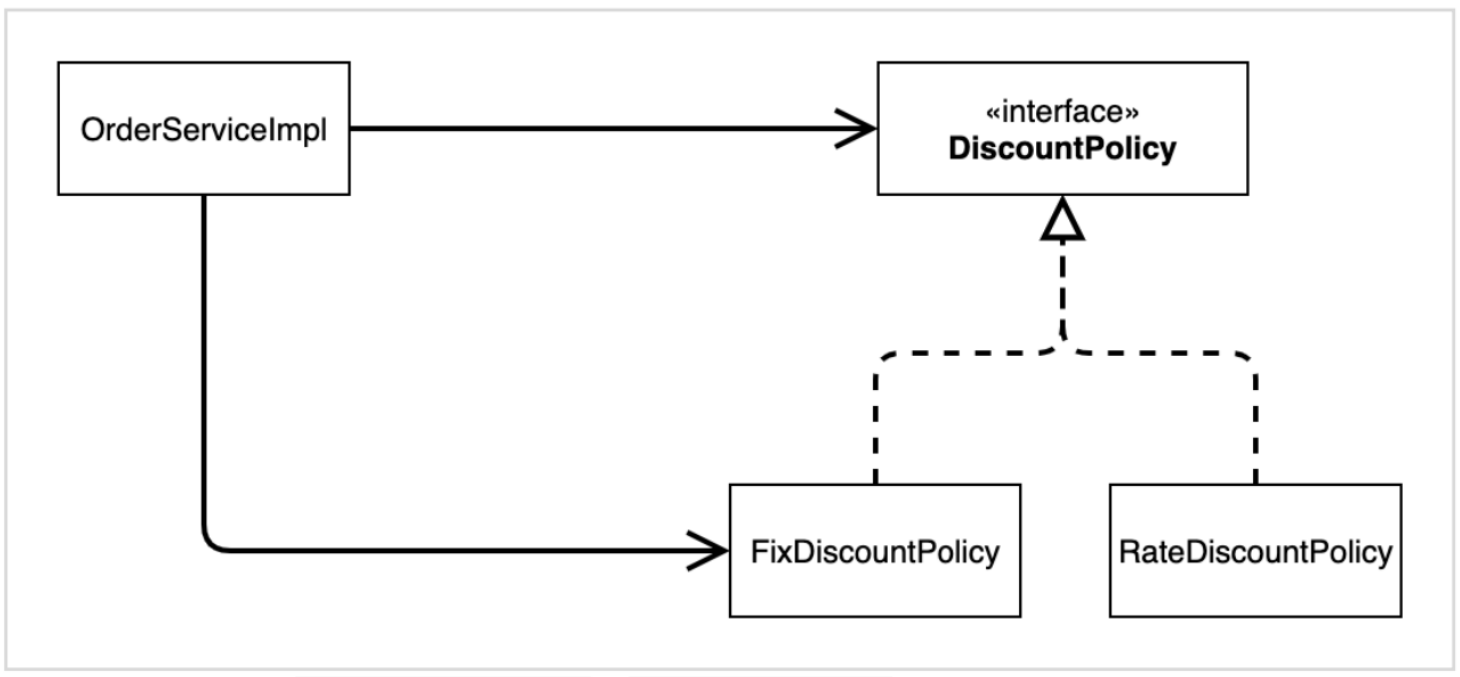
- 역할과 구현을 잘 분리했다.
- `다형성`을 활용하고, 인터페이스와 구현 객체를 분리했다.
- 하지만 `DIP`, `OCP`는 만족하지 못했다.
  - `DIP`: 주문서비스 클라이언트(`OrderServiceImpl`)는 `DiscountPolicy` 인터페이스에 의존하면서 DIP를 만족하는 것처럼 보인다.



기대했던 의존관계

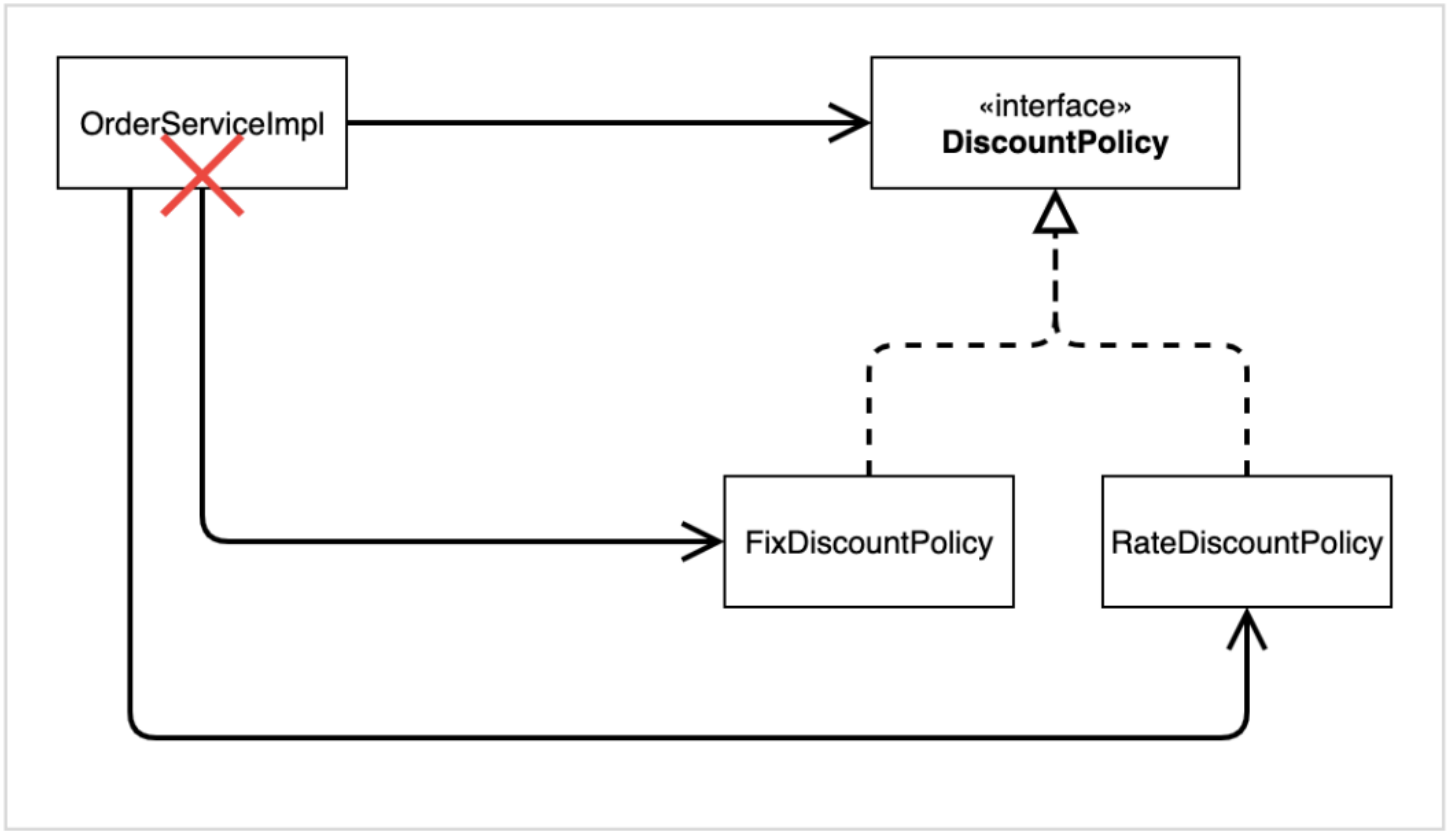
```
public class OrderServiceImpl implements OrderService {
  // private final DiscountPolicy discountPolicy = new FixDiscountPolicy();
  private final DiscountPolicy discountPolicy = new RateDiscountPolicy();
}
```

- 하지만 **구체(구현) 클래스도 의존하고 있다.** → `DIP` 위반



실제 의존관계

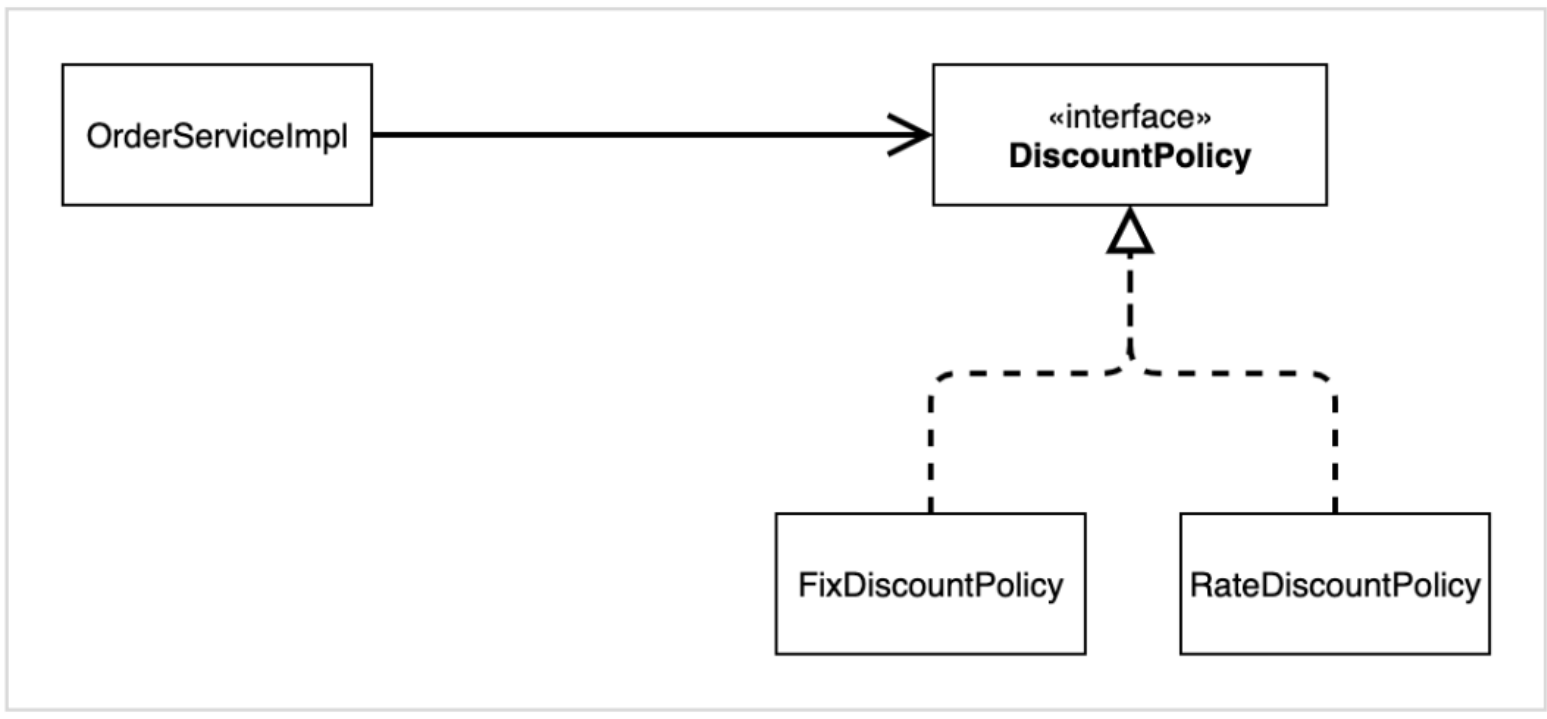
  - 추상(인터페이스) 클래스 의존: `DiscountPolicy`
  - 구체(구현) 클래스 의존: `FixDiscountPolicy`, `RateDiscountPolicy`
- `OCP`: 변경하지 않고 확장할 수 있는 것처럼 보인다.
  - 기능을 확장해서 변경하면, `2.3.2.`와 같이 클라이언트 코드에 영향을 준다. → `OCP` 위반



`FixDiscountPolicy`를 `RateDiscountPolicy`로 변경하는 순간 `OrderServiceImpl`의 소스 코드도 변경해야 한다.

👩‍💻 **해결방법**

- 클라이언트를 추상(인터페이스) 클래스에만 의존하도록 `의존관계`를 변경해야 한다.




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

  - 실제 실행 시 구현체가 없어 `NullPointerException`이 발생한다. → 클라이언트인 `OrderServiceImpl`에 구현 객체를 대신 생성하고 주입해주어야 한다.

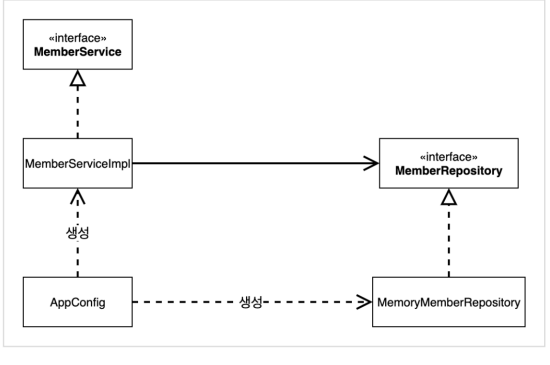
2.3.3. 관심사의 분리

- `AppConfig`의 등장: 애플리케이션의 전체 동작 방식을 구성(config)하기 위해 **구현 객체를 생성하고 연결하는 책임을 가지는 별도의 설정 클래스**를 만든다.

```
package com.allyhyeseongkim.core;

import com.allyhyeseongkim.core.discount.FixDiscountPolicy;
import com.allyhyeseongkim.core.member.MemberService;
import com.allyhyeseongkim.core.member.MemberServiceImpl;
import com.allyhyeseongkim.core.member.MemoryMemberRepository;
import com.allyhyeseongkim.core.order.OrderService;
import com.allyhyeseongkim.core.order.OrderServiceImpl;

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

```
  }
}
```

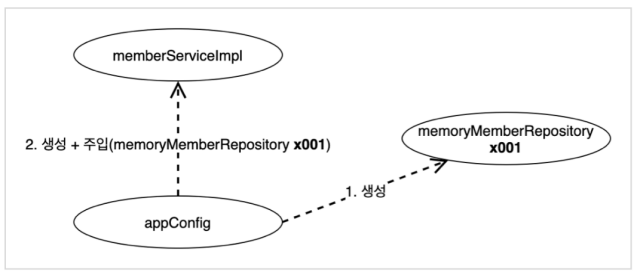
- 애플리케이션의 실제 동작에 필요한 **구현** 객체를 생성한다.
  - `MemberServiceImpl`
  - `MemoryMemberRepository`
  - `OrderServiceImpl`
  - `FixDiscountPolicy`
- 생성한 객체 인스턴스의 참조(레퍼런스)를 생성자를 통해 주입(연결)한다.
  - `MemberServiceImpl` → `MemoryMemberRepository`



클래스 다이어그램

```
package com.allyhyeseongkim.core.member;

public class MemberServiceImpl implements MemberService {
  private final MemberRepository memberRepository;

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

- `MemberServiceImpl`은 `MemoryMemberRepository`를 의존하지 않는다.
  - `MemberRepository` 인터페이스만 의존한다. → `DIP` 만족
- `MemberServiceImpl` 입장에서 생성자를 통해 어떤 구현 객체가 주입될지 알 수 없다.
  - 어떤 구현 객체를 주입할지는 외부(`AppConfig`)에서 결정된다.
  - 객체의 생성과 연결은 `AppConfig`에서 결정된다.
  - **의존관계(객체의 생성과 연결)에 대한 결정을 외부에 맡기고** 실행에만 집중한다. → **관심사의 분리**



객체 인스턴스 다이어그램

- `appConfig` 객체는 `memoryMemberRepository` 객체를 생성하고 참조값을 `memberServiceImpl`을 생성하면서 생성자로 전달한다.
- 클라이언트인 `memberServiceImp` 입장에서 **의존관계(`MemoryMemberRepository`)를 외부에서 주입해준다.** → `DI(Dependency Injection)`

- `OrderServiceImpl` → `MemoryMemberRepository`, `FixDiscountPolicy`

```
package com.allyhyeseongkim.core.order;

import com.allyhyeseongkim.core.discount.DiscountPolicy;
import com.allyhyeseongkim.core.member.MemberRepository;

public class OrderServiceImpl implements OrderService {
  private final MemberRepository memberRepository;
  private final DiscountPolicy discountPolicy;

  public OrderServiceImpl(MemberRepository memberRepository, DiscountPolicy discountPolicy) {
    this.memberRepository = memberRepository;
    this.discountPolicy = discountPolicy;
  }
  ...
}
```

- `OrderServiceImpl`은 `FixDiscountPolicy`를 의존하지 않는다.
  - `DiscountPolicy` 인터페이스만 의존한다. → `DIP` 만족
- `OrderServiceImpl` 입장에서 생성자를 통해 어떤 구현 객체가 주입될지 알 수 없다.

        - 어떤 구현 객체를 주입할지는 외부(`AppConfig`)에서 결정된다.
        - 객체의 생성과 연결은 `AppConfig`에서 결정된다.
        - **의존관계(객체의 생성과 연결)에 대한 결정을 외부에 맡기고** 실행에만 집중한다. → **관심사의 분리**
    - 클라이언트인 `OrderServiceImpl` 입장에서 **의존관계(`MemoryMemberRepository`, `FixDiscountPolicy`)를 외부에서 주입해준다.** → **`DI(Dependency Injection)`**
- `AppConfig` 실행

```
package com.allyhyeseongkim.core;

import com.allyhyeseongkim.core.member.Grade;
import com.allyhyeseongkim.core.member.Member;
import com.allyhyeseongkim.core.member.MemberService;

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

```
package com.allyhyeseongkim.core;

import com.allyhyeseongkim.core.member.Grade;
import com.allyhyeseongkim.core.member.Member;
import com.allyhyeseongkim.core.member.MemberService;
import com.allyhyeseongkim.core.order.Order;
import com.allyhyeseongkim.core.order.OrderService;

public class OrderApp {
  public static void main(String[] args) {
    AppConfig appConfig = new AppConfig();
    MemberService memberService = appConfig.memberService();
    OrderService orderService = appConfig.orderService();

    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);
  }
}
```

- 테스트 코드 오류 수정

```
import com.allyhyeseongkim.core.AppConfig;
import com.allyhyeseongkim.core.member.MemberService;
import org.junit.jupiter.api.BeforeEach;

class MemberServiceTest {
  MemberService memberService;

  @BeforeEach  // 각 테스트를 시작하기 전에 호출된다.
  public void beforeEach() {
    AppConfig appConfig = new AppConfig();
    this.memberService = appConfig.memberService();
  }
  ...
}
```

```
import com.allyhyeseongkim.core.AppConfig;
import com.allyhyeseongkim.core.member.MemberService;
package com.allyhyeseongkim.core.order.OrderService;
import org.junit.jupiter.api.BeforeEach;

class OrderServiceTest {
  MemberService memberService;
  OrderService orderService;

  @BeforeEach
  public void beforeEach() {
    AppConfig appConfig = new AppConfig();
    this.memberService = appConfig.memberService();
    this.orderService = appConfig.orderService();
  }
  ...
}
```

2.3.4. AppConfig 리팩터링

- 현재 `AppConfig`는 중복이 있고, **역할**에 따른 **구현**이 잘 안보인다.
    - **역할**: 주문 서비스 역할, 회원 저장소 역할, 할인 정책 역할
    - **구현**: 주문 서비스 구현체, 메모리 회원 저장소, 정액 할인 정책

```
package com.allyhyeseongkim.core;

import com.allyhyeseongkim.core.discount.FixDiscountPolicy;
import com.allyhyeseongkim.core.member.MemberService;
import com.allyhyeseongkim.core.member.MemberServiceImpl;
import com.allyhyeseongkim.core.member.MemoryMemberRepository;
import com.allyhyeseongkim.core.order.OrderService;
import com.allyhyeseongkim.core.order.OrderServiceImpl;

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

- `new MemoryMemberRepository()` → 중복

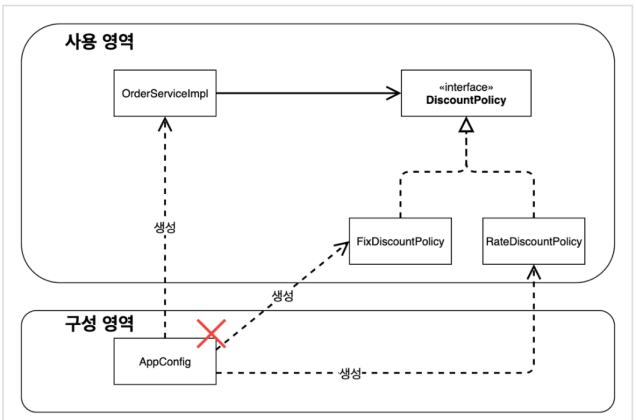
- 중복을 제거하고 **역할**에 따른 **구현**이 보이도록 `리팩터링`한다.

```
package com.allyhyeseongkim.core;

import com.allyhyeseongkim.core.discount.DiscountPolicy;
import com.allyhyeseongkim.core.discount.FixDiscountPolicy;
import com.allyhyeseongkim.core.member.MemberRepository;
import com.allyhyeseongkim.core.member.MemberService;
import com.allyhyeseongkim.core.member.MemberServiceImpl;
import com.allyhyeseongkim.core.member.MemoryMemberRepository;
import com.allyhyeseongkim.core.order.OrderService;
import com.allyhyeseongkim.core.order.OrderServiceImpl;

public class AppConfig {
  public MemberService memberService() {  //역할
    return new MemberServiceImpl(memberRepository());  //구현
  }
  public OrderService orderService() {  //역할
    return new OrderServiceImpl(memberRepository(), discountPolicy());  //구현
  }

  public MemoryRepository memoryRepository() {  //역할
    return new MemoryMemberRepository;  //구현
  }
  public DiscountPolicy discountPolicy() {  //역할
    return new FixDiscountPolicy;  //구현
  }
}
```

- 역할과 구현 클래스가 한눈에 들어온다. → 애플리케이션 전체 구성이 어떻게 되어있는지 빠르게 파악할 수 있다.

2.3.5. 새로운 구조와 할인 정책 적용

- `AppConfig`의 등장으로 애플리케이션이 **사용 영역과 객체를 생성하고 구성(`Configuration`)하는 영역으로 분리되었다.**



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

- 정률% 할인 정책으로 변경해보자.
  - `FixDiscountPolicy` → `RateDiscountPolicy`

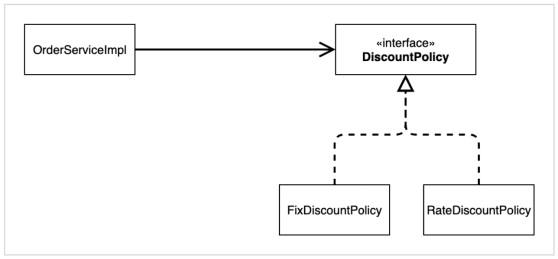
```
package com.allyhyeseongkim.core;

import com.allyhyeseongkim.core.discount.DiscountPolicy;
import com.allyhyeseongkim.core.discount.RateDiscountPolicy;
import com.allyhyeseongkim.core.member.MemberRepository;
import com.allyhyeseongkim.core.member.MemberService;
import com.allyhyeseongkim.core.member.MemberServiceImpl;
import com.allyhyeseongkim.core.member.MemoryMemberRepository;
import com.allyhyeseongkim.core.order.OrderService;
import com.allyhyeseongkim.core.order.OrderServiceImpl;

public class AppConfig {
  public MemberService memberService() {  //역할
    return new MemberServiceImpl(memberRepository());  //구현
  }
  public OrderService orderService() {  //역할
    return new OrderServiceImpl(memberRepository(), discountPolicy());  //구현
  }

  public MemoryRepository memoryRepository() {  //역할
```

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

2.3.6. 전체 흐름 정리

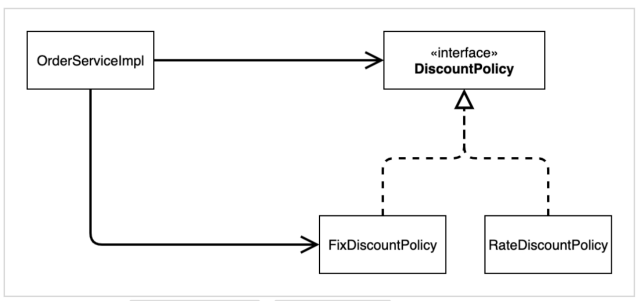
👩‍💻 새로운 할인 정책 개발

- `다형성`을 사용하여 개발할 수 있다.



👩‍💻 새로운 할인 정책 적용과 문제점

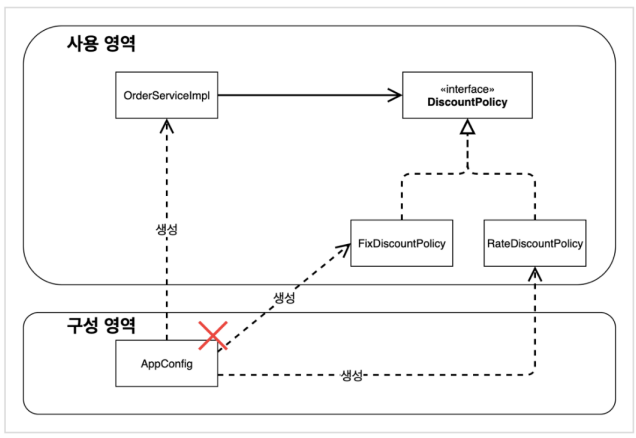
- 클라이언트 코드인 주문 서비스의 구현체도 함께 변경해야 한다.
  - 주문 서비스 클라이언트가 인터페이스인 `DiscountPolicy`뿐만 아니라, 구체 클래스인 `FixDiscountPolicy`도 같이 의존한다. → `DIP` 위반



실제 의존관계

👩‍💻 관심사의 분리

- `AppConfig`는 애플리케이션의 전체 동작 방식을 구성(config)하기 위해 **구현 객체를 생성하고 연결하는 책임을 갖는다.**



  - 클라이언트 객체는 자신의 역할을 실행하는 책임만을 갖는다.

👩‍💻 `AppConfig` `리팩터링`

- 구성 정보에서 **역할**과 **구현**을 명확하게 분리했다.

👩‍💻 새로운 구조와 할인 정책 적용

- `AppConfig`의 등장으로 애플리케이션이 **사용 영역과 객체를 생성하고 구성(Configuration)하는 영역으로 분리됐다.**
  - 할인 정책을 변경해도 `AppConfig`가 있는 구성 영역만 변경하면 된다.

2.3.7. 좋은 객체 지향 설계의 5가지 원칙의 적용

- `SRP`, `DIP`, `OCP`를 적용하였다.

👩‍💻 `SRP(단일 책임 원칙)`: 한 클래스는 하나의 책임만 가져야 한다.

- 클라이언트는 객체를 직접 **구현 객체를 생성하고 연결하고 실행하는 다양한 책임을 가지고 있었다.**
- **관심사를 분리하여** **구현 객체를 생성하고 연결하는 책임은** `AppConfig`가 담당하고 **객체를 실행하는 책임은 클라이언트 객체가 담당**하게 되었다. → `SRP` 만족

👩‍💻 `DIP(의존관계 역전 원칙)`: 추상화에 의존해야지, 구체화에 의존하면 안된다.

- 새로운 할인 정책을 적용할 때 클라이언트 코드를 같이 변경해야 했다.
  - 기존 클라이언트 코드(`OrderServiceImpl`)은 추상화 인터페이스(`DiscountPolicy`)와 구체 구현 클래스(`FixDiscountPolicy`)에 함께 의존했다.
- `AppConfig`에서 클라이언트 코드에서 사용하는 구현 객체를 주입하게 하여 **클라이언트 코드는 추상화 인터페이스만 의존하도록 하였다.** → `DIP` 만족

👩‍💻 `OCP`: 확장에는 열려 있으나 변경에는 닫혀 있어야 한다.

- 관심사를 분리하고 `AppConfig`가 클라이언트 코드에 구현 객체를 주입하도록 하여 **클라이언트 코드를 수정하지 않고 기능을 확장할 수 있다.** → `OCP` 만족

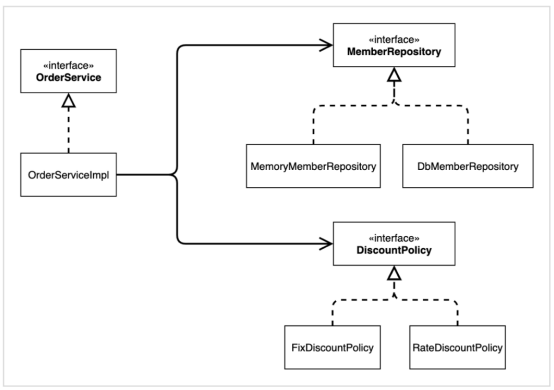
2.3.8. IoC, DI, 그리고 컨테이너

👩‍💻 `제어의 역전(Inversion of Control, IoC)`: **프로그램의 제어 흐름을 직접 제어하는 것이 아니라 외부에서 관리한다.**

- 기존 프로그램은 클라이언트 구현 객체가 서버 구현 객체를 생성하고 연결하고 실행했다.
  - **구현 객체가 프로그램의 제어 흐름을 조종한다.**
- `AppConfig`가 등장한 후 구현 객체는 자신의 로직을 실행하는 역할만 담당한다.
  - **프로그램의 제어 흐름은** `AppConfig`**가 가진다.**
  - ex. `OrderServiceImpl`은 필요한 인터페이스를 호출하지만 어떤 구현 객체들이 실행될지 모른다.
- `프레임워크` vs `라이브러리`
  - `프레임워크`: 내가 작성한 코드를 제어하고 대신 실행한다. ex. `JUnit`
  - `라이브러리`: 내가 작성한 코드가 직접 제어의 흐름을 담당한다.

👩‍💻 `의존관계 주입(Dependency Injection, DI)`: **애플리케이션의 `실행 시점(run-time)`에 외부에서 실제 구현 객체를 생성하고 클라이언트에 주입**해서 클라이언트와 서버의 의존관계가 연결된다.

- 객체 인스턴스를 생성하고 그 참조값을 주입한다.
- 클라이언트 코드를 변경하지 않고 클라이언트가 호출하는 대상의 타입 인스턴스를 변경할 수 있다.
- `정적인 클래스 의존관계`를 변경하지 않고 `동적인 객체 인스턴스 의존관계`를 쉽게 변경할 수 있다.



클래스 다이어그램 → 정적 클래스 의존관계는 애플리케이션을 실행하지 않아도 분석할 수 있다. `import` 코드만 보고 분석할 수 있다.

객체 다이어그램 → 동적 클래스 의존관계는 애플리케이션 실행 시점에 생성된 객체 인스턴스의 참조가 연결된 의존관계이다.

👩‍💻 `IoC 컨테이너`, `DI 컨테이너`: 객체를 생성하고 관리하면서 의존관계를 연결한다.

- **의존관계 주입**에 초점을 맞추어 최근에는 주로 `DI 컨테이너`라고 한다.
  - `어셈블러`, `오브젝트 팩토리` 등으로 불리기도 한다.
- ex. `AppConfig`

2.3.9. 스프링으로 전환하기

- `AppConfig` 스프링 기반으로 변경

```
package com.allyhyeseongkim.core;

import com.allyhyeseongkim.core.discount.DiscountPolicy;
import com.allyhyeseongkim.core.discount.RateDiscountPolicy;
import com.allyhyeseongkim.core.member.MemberRepository;
import com.allyhyeseongkim.core.member.MemberService;
import com.allyhyeseongkim.core.member.MemberServiceImpl;
import com.allyhyeseongkim.core.member.MemoryMemberRepository;
import com.allyhyeseongkim.core.order.OrderService;
import com.allyhyeseongkim.core.order.OrderServiceImpl;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

@Configuration
public class AppConfig {
  @Bean
  public MemberService memberService() {
    return new MemberServiceImpl(memberRepository());
  }

  @Bean
  public OrderService orderService() {
    return new OrderServiceImpl(memberRepository(), discountPolicy);
  }

  @Bean
  public MemberRepository memberRepository() {
    return new MemoryMemberRepository();
  }

  @Bean
  public DiscountPolicy discountPolicy() {
    return new RateDiscountPolicy();
  }
```

    - `@Configuration` : `AppConfig`에 설정을 구성한다.
    - `@Bean` : `스프링 컨테이너`에 `스프링 빈`으로 등록한다.
- `MemberApp`에 `스프링 컨테이너` 적용

```
package com.allyhyeseongkim.core;

import com.allyhyeseongkim.core.member.Grade;
import com.allyhyeseongkim.core.member.Member;
import com.allyhyeseongkim.core.member.MemberService;
import org.springframework.context.ApplicationContext;
import org.springframework.context.annotation.AnnotationConfigApplicationContext;

public class MemberApp {
  public static void main(String[] args) {
    ApplicationContext applicationContext = new AnnotationConfigApplicationContext(AppConfig.class);
    MemberService memberService = applicationContext.getBean("memberService", MemberService.class);

    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());
  }
}
```

- `OrderApp`에 `스프링 컨테이너` 적용

```
public class OrderApp {
  public static void main(String[] args) {
    ApplicationContext applicationContext = new AnnotationConfigApplicationContext(AppConfig.class);
    MemberService memberService = applicationContext.getBean("memberService", MemberService.class);
    OrderService orderService = applicationContext.getBean("orderService", OrderService.class);

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

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

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

👩‍💻 `스프링 컨테이너` : `ApplicationContext`

- 기존에는 개발자가 `AppConfig`를 사용해서 직접 객체를 사용하고 `DI`를 했지만 이제는 `스프링 컨테이너`를 통해 사용한다.
    - `스프링 컨테이너`는 `@Configuration`이 붙은 `AppConfig`를 설정(구성) 정보로 사용한다.
        - `@Bean`이 붙은 메서드를 모두 호출해서 반환된 객체를 `스프링 컨테이너`에 등록한다.
        - `스프링 컨테이너`에 등록된 객체를 `스프링 빈`이라 한다.
    - `스프링 빈`은 `@Bean`이 붙은 메서드 명을 `스프링 빈`의 이름으로 사용한다.
        - ex. `memberService`, `orderService`
- 기존에는 개발자가 필요한 객체를 `AppConfig`를 사용해서 직접 조회했지만 이제는 `스프링 컨테이너`를 통해 필요한 `스프링 빈`(객체)을 찾아야 한다.
    - `applicationContext.getBean()`
- 기존에는 개발자가 직접 자바 코드로 모든 것을 했다면 이제는 `스프링 컨테이너`에 객체를 `스프링 빈`으로 등록하고, `프링 컨테이너`에서 `스프링 빈`을 찾아서 사용한다.