Clean Code

Chapter 10. 클래스
Chapter 11. 시스템

10

클래스

클래스 체계

표준 자바 관례에 따른 내부 체계 순서

1. static public 상수
2. static private 변수
3. private instance 변수 (public 변수가 필요한 경우는 거의 없음)
4. public 함수
   - private 함수는 자신을 호출하는 public 함수 직후에 선언

클래스는 작아야 한다.

'작다'의 기준 = 클래스가 맡은 책임의 수

클래스 이름은 해당 클래스 책임을 기술해야 하는데,

모호하거나 간결한 이름이 떠오르지 않는 경우에는 책임이 많다는 의미!

예를 들어, Processor, Manager, Super 등과 같은 모호한 단어를 포함한 경우

클래스 설명은 if, and, or, but을 사용하지 않고 25단어 내외로 가능해야 한다.

예를 들어, "이 클래스는 ~하는 방법을 제공**하며**, 버전을 추적해준다."

클래스는 작아야 한다.

SOLID 원칙

1. **단일 책임 원칙 (Single Responsiblity Principle; SRP)**
   모든 클래스는 하나의 책임만 가져야 한다. 클래스는 그 책임을 캡슐화해야 한다.

2. **개방-폐쇄 원칙 (Open Closed Principle; OCP)**
   기존의 코드를 변경하지 않으면서, 기능을 추가할 수 있도록 설계되어야 한다.

3. **리스코프 치환 원칙 (Liskov Substitution Principle; LSP)**
   자식 클래스는 언제나 부모 클래스를 대체할 수 있어야 한다.

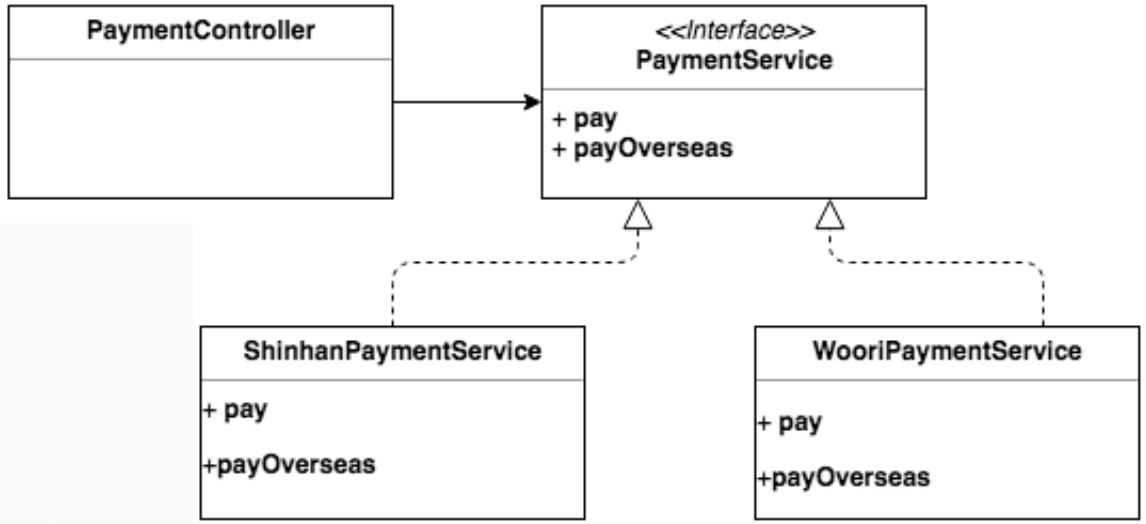
4. **인터페이스 분리 원칙 (Interface Segregation Principle; ISP)**
   구현 클래스는 자신이 사용하지 않을 인터페이스는 구현하지 않아야 한다.

5. **의존 역전 원칙 (Dependency Inversion Principle; DIP)**
   클라이언트는 구현 클래스가 아닌, 인터페이스에 의존해야 한다.

단일 책임 원칙 - SRP

클래스는 단 하나의 책임만 가져야 한다.

클래스가 변경되는 이유는 한 가지여야 한다.

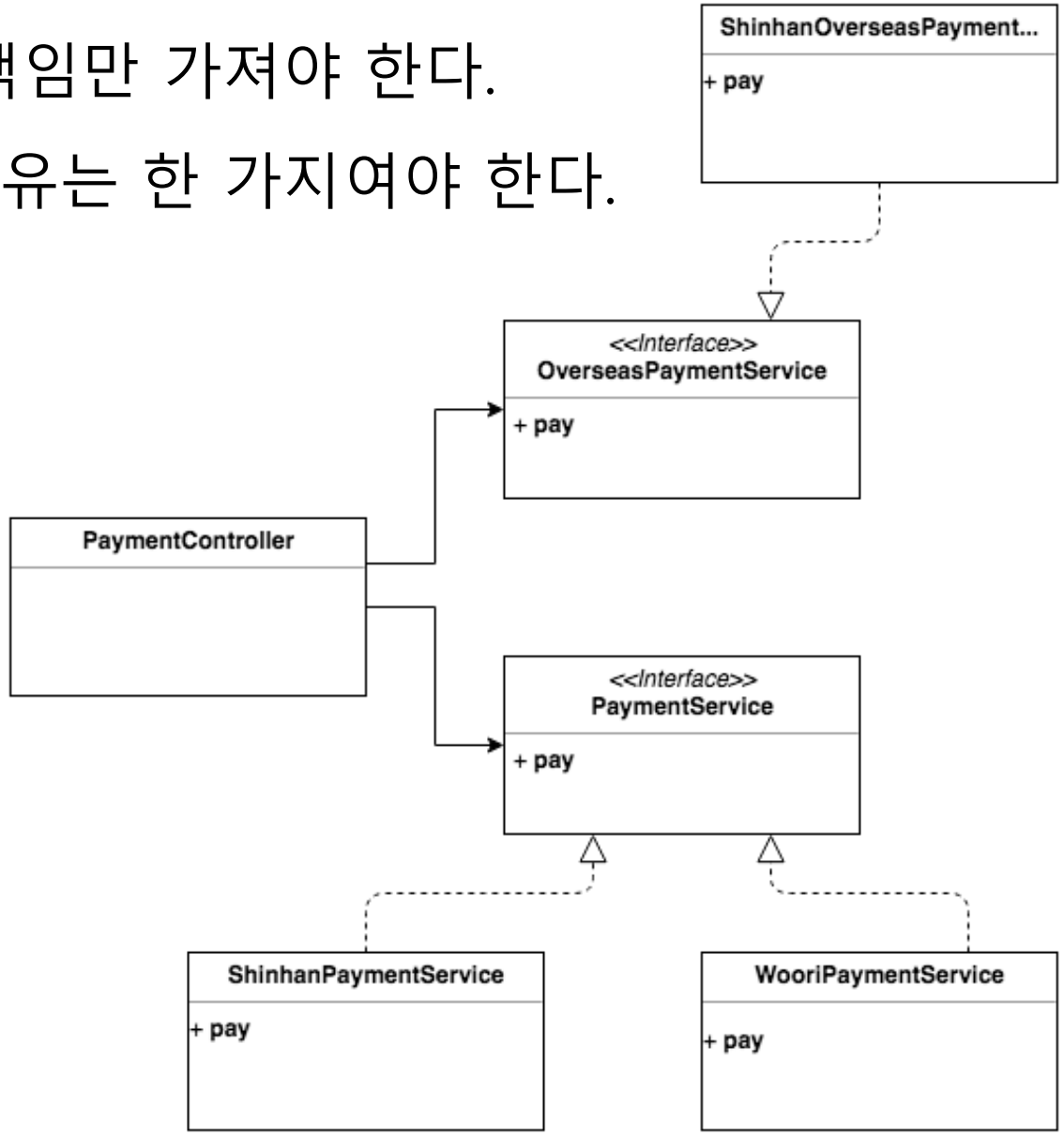



```java
public interface CardPaymentService {
    void pay(CardPaymentDto req);
    void payOverseas(CardPaymentDto req);
}

public class ShinhanCardPaymentService implements CardPaymentService {
    @Override
    public void pay(CardPaymentDto req) {...}
    @Override
    public void payOverseas(CardPaymentDto req) {...}
}

public class WooriCardPaymentService implements CardPaymentService {
    @Override
    public void pay(CardPaymentDto req) {...}
    @Override
    public void payOverseas(CardPaymentDto req) {...}
}
```

단일 책임 원칙 - SRP

클래스는 단 하나의 책임만 가져야 한다.

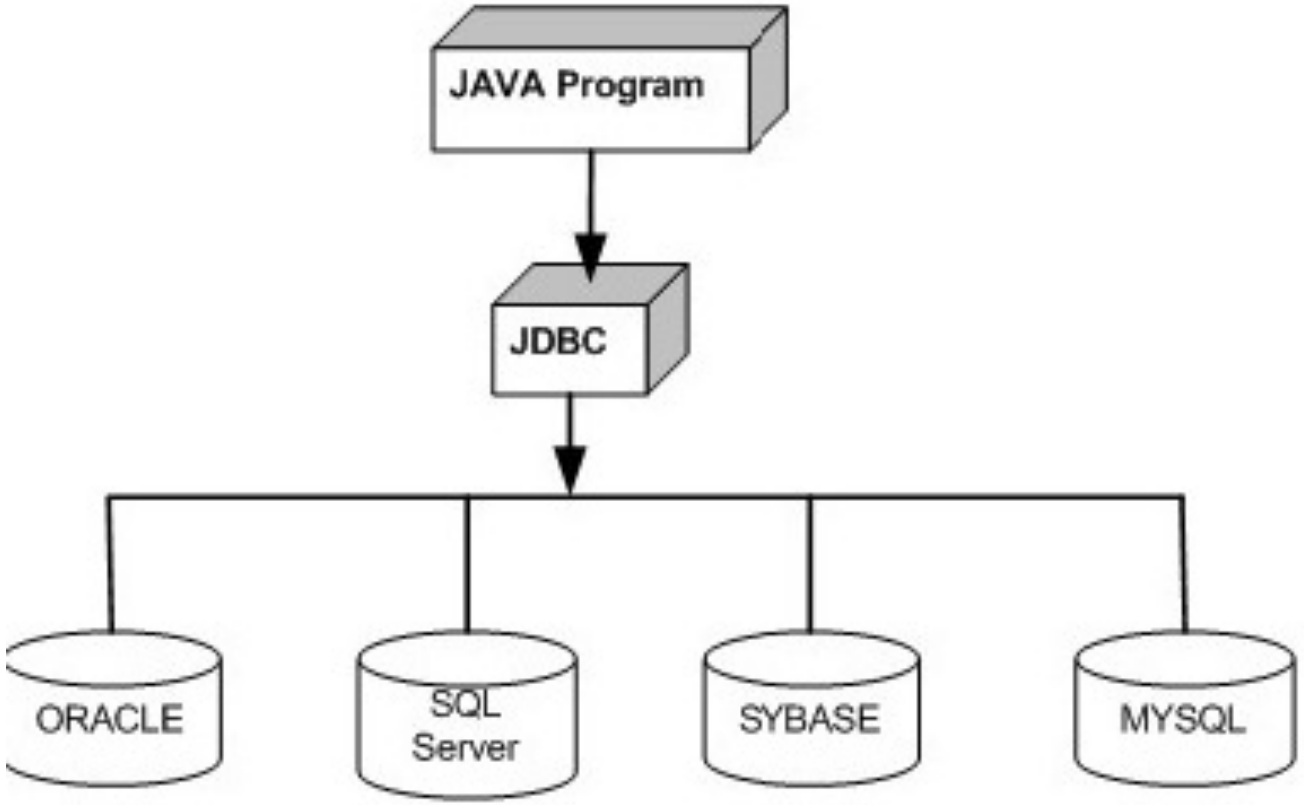
클래스가 변경되는 이유는 한 가지여야 한다.



개방-폐쇄 원칙 - OCP

확장에는 열려있어야 하며, 수정에는 닫혀있어야 한다.

기존의 코드 수정은 최소화하면서, 기능을 추가할 수 있어야 한다.



개방-폐쇄 원칙 - OCP

```
class Animal {
    String type;
    Animal(String type) {
        this.type = type;
    }
}

class AnimalSpeak {
    void speak(Animal animal) {
        if(animal.type.equals("Cat")) {
            System.out.println("냐옹");
        } else if(animal.type.equals("Dog")) {
            System.out.println("멍멍");
        }
    }
}
```

```
public class Main {
    public static void main(String[] args) {
        AnimalSpeak speaker = new AnimalSpeak();

        Animal cat = new Animal("Cat");
        Animal dog = new Animal("Dog");

        speaker.speak(cat); // 냐옹
        speaker.speak(dog); // 멍멍
    }
}
```

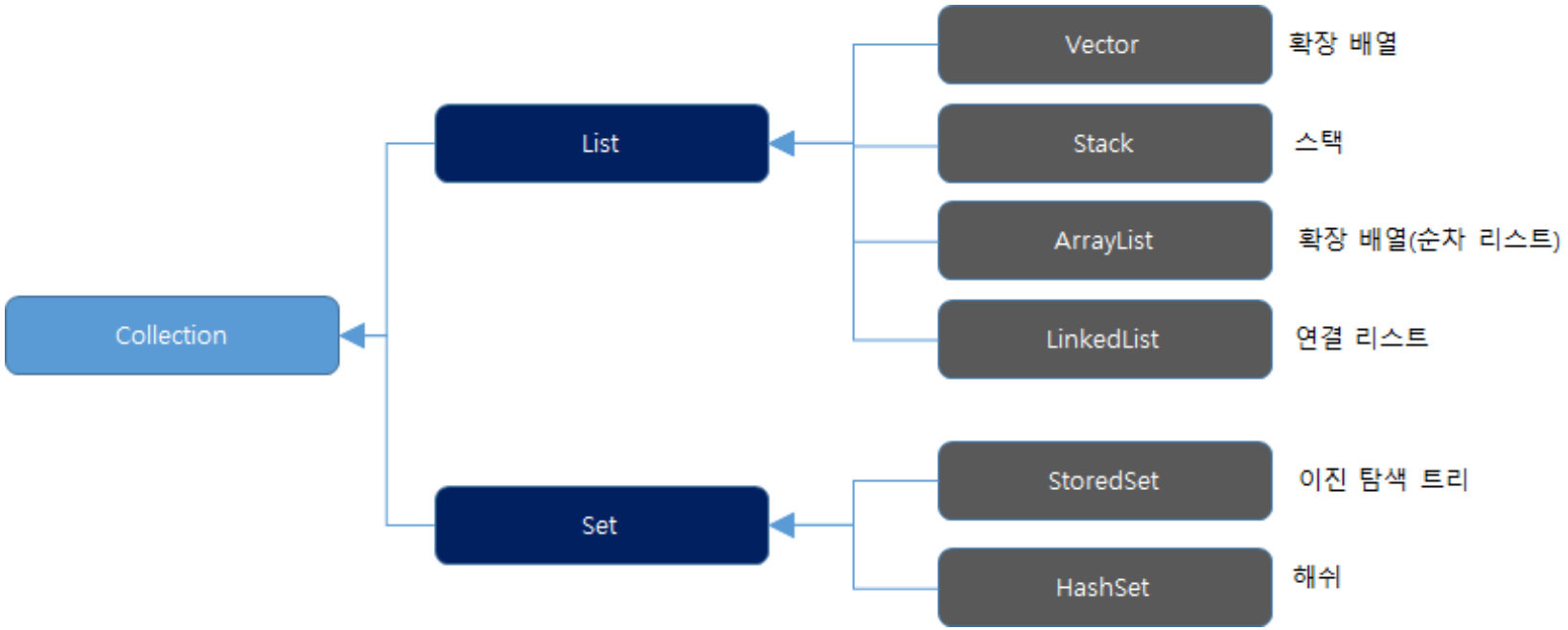
개방-폐쇄 원칙 - OCP

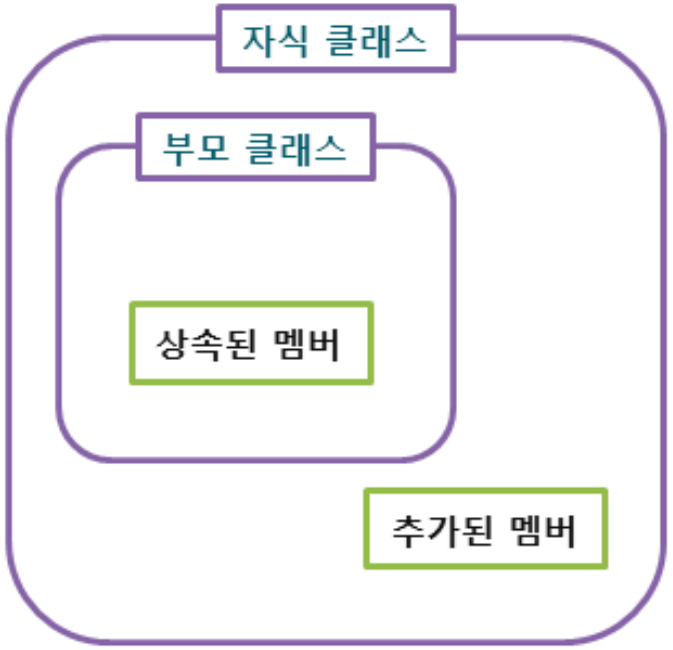
```
interface Animal {
            void speak();
}

class Cat implements Animal {
    void speak() {
        System.out.println("냐옹");
    }
}

class Dog implements Animal {
    void speak() {
        System.out.println("멍멍");
    }
}

class AnimalSpeak {
    void speak(Animal animal) {
        animal.speak();
    }
}
```

```
public class Main {
    public static void main(String[] args) {
        AnimalSpeak speaker = new AnimalSpeak();

        Animal cat = new Cat();
        Animal dog = new Dog();

        speaker.speak(cat); // 냐옹
        speaker.speak(dog); // 멍멍
    }
}
```

리스코프 치환 원칙 - LSP

서브 타입은 언제나 부모 타입을 대체할 수 있어야 한다.





리스코프 치환 원칙 - LSP

```java
public class Rectangle {
    private int width;
    private int height;

    // getter, setter
    ...
}
```

```java
public class Square extends Rectangle {

    @Override
    public void setWidth(final int width) {
        super.setWidth(width);
        super.setHeight(width);
    }

    @Override
    public void setHeight(final int height) {
        super.setWidth(height);
        super.setHeight(height);
    }
}
```

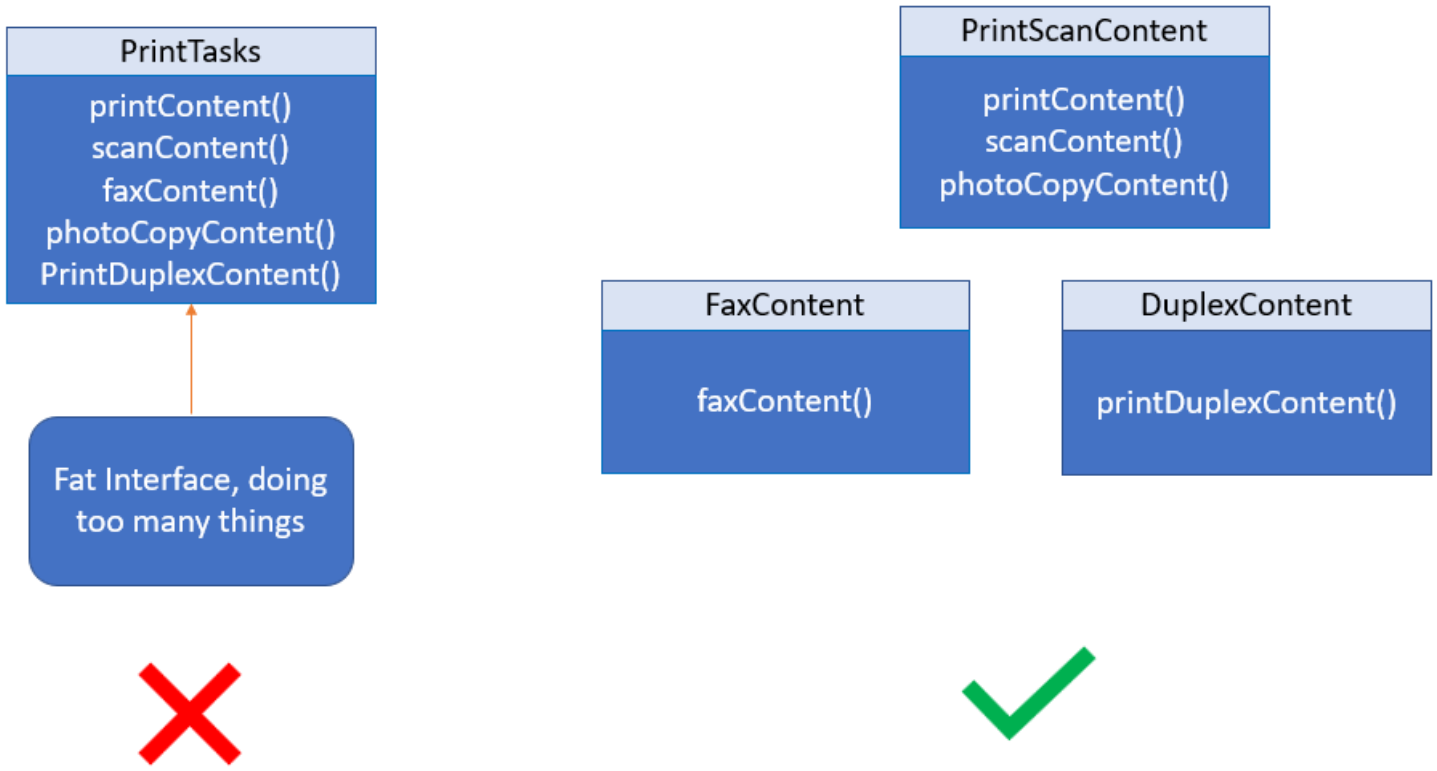
```java
public void increaseHeight(final Rectangle rectangle) {
  if (rectangle.getHeight() <= rectangle.getWidth()) {
      rectangle.setHeight(rectangle.getWidth() + 1);
  }
}
```

```java
public void increaseHeight(final Rectangle rectangle) {
  if (rectangle instanceof Square) {
      throw new IllegalStateException();
  }

  if (rectangle.getHeight() <= rectangle.getWidth()) {
      rectangle.setHeight(rectangle.getWidth() + 1);
  }
}
```

인터페이스 분리 원칙 – ISP

인터페이스의 단일 책임 원칙

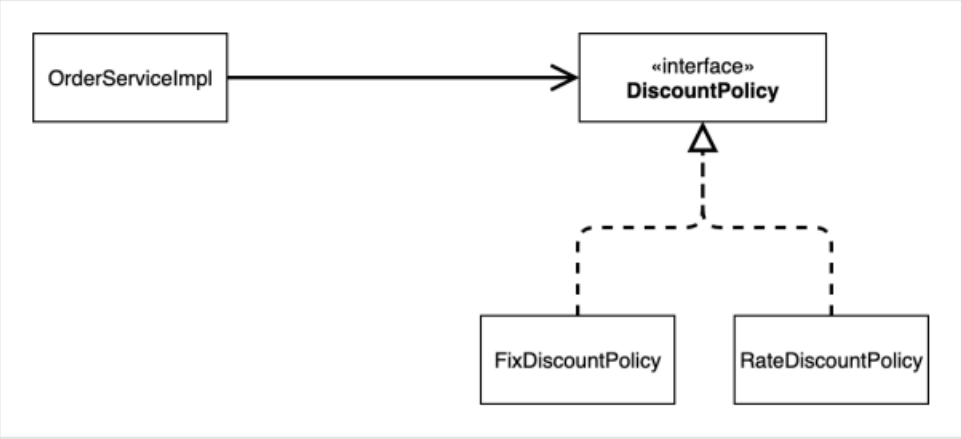
구현 클래스에서 자신이 사용하지 않는 메소드마저 **억지로 구현**시키지 말게하자



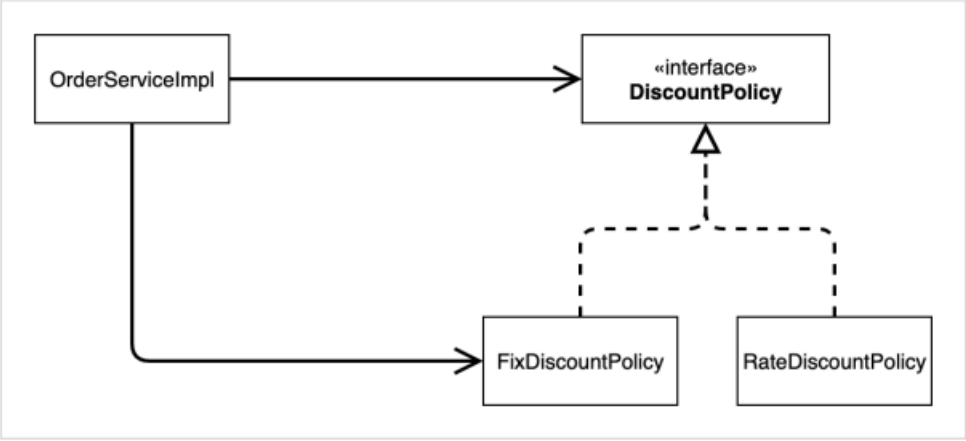
의존 역전 원칙 - DIP

클래스를 직접 참조하는 것이 아니라, 상위 요소(추상 클래스, 인터페이스)를 참조

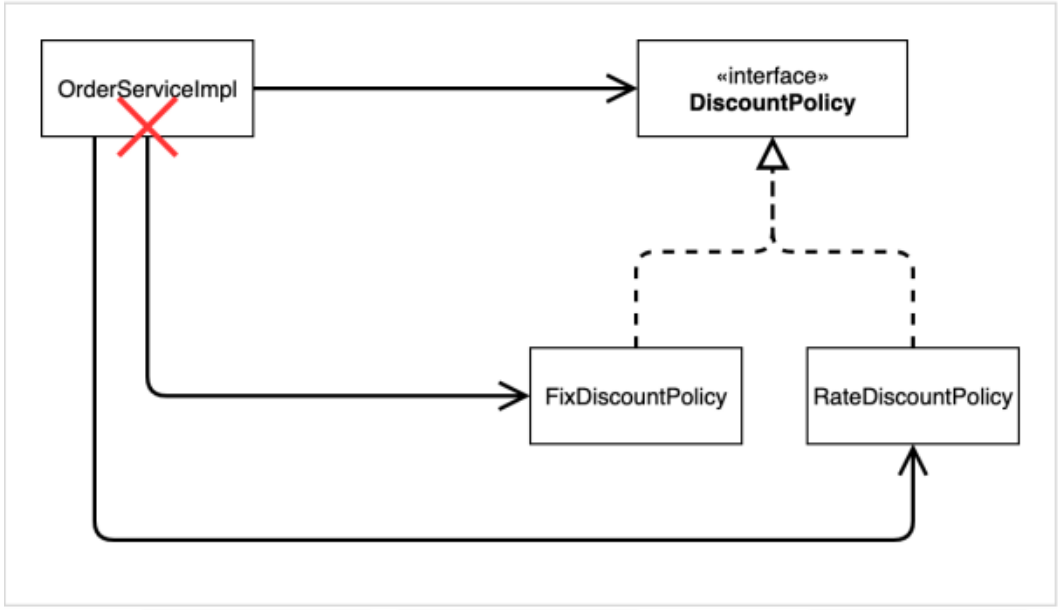
Expected



Actually : DIP 위반



정책 변경



의존 역전 원칙 - DIP

```java
public class OrderService {

    private final DiscountPolicy discountPolicy = new RateDiscountPolicy();

    public void discountPrice(int price) {
        System.out.println("= 주문 금액 : " + price);
        int actualPrice = discountPolicy.discountPrice(price);
        System.out.println("= 실결제 금액 : " + actualPrice);
    }
}
```

```java
public interface DiscountPolicy {
    int discountPrice(int price);
}
```

```java
public class RateDiscountPolicy implements DiscountPolicy {
    @Override
    public int discountPrice(int price) {
        return (int) (price * 0.9);
    }
}
```

```java
public class FixDiscountPolicy implements DiscountPolicy {
    @Override
    public int discountPrice(int price) {
        return price - 1000;
    }
}
```

의존 역전 원칙 - DIP

```java
public class OrderService {

    private final DiscountPolicy discountPolicy;

    public OrderService(DiscountPolicy discountPolicy) {
        this.discountPolicy = discountPolicy;
    }

    public void discountPrice(int price) {
        System.out.println("= 주문 금액 : " + price);
        int actualPrice = discountPolicy.discountPrice(price);
        System.out.println("= 실결제 금액 : " + actualPrice);
    }
}
```

Dependency Injection!


```java
public class Main {
    public static void main(String[] args) {
        OrderService orderService = new OrderService(new RateDiscountPolicy());
        orderService.discountPrice(10000);
    }
}
```

응집도

각 클래스의 메서드는 인스턴스 변수를 하나 이상 사용하게 만들어야 한다.

많이 사용할수록 메서드와 클래스의 응집도는 더 높다.

응집도가 높은 클래스를 지향하자.

'함수를 작게, 매개변수 목록을 짧게' 규칙을 지키다 보면, 응집도가 낮아지게 되는데

새로운 클래스로 쪼개야 한다는 신호!

응집도

큰 함수를 작은 함수 여럿으로 나누면 클래스 수가 많아질 수 있다.

1. 큰 함수 일부를 작은 함수 하나로 빼고 싶다.
2. 근데 빼내려는 코드가 큰 함수에 정의된 변수 4개를 사용한다.
3. 그럼 작은 함수의 인수를 4개로 넣어줘야 할까?
4. 클래스로 분리하고, 4개의 변수를 인스턴스 변수로 승격하면 된다!

11

시스템

시스템 제작과 시스템 사용을 분리하라

```
public Service getService(){
  if (service == null)
    service = new MyServiceImpl(...);
  return service;
}
```

장점

- 실제로 필요할 때까지 객체를 생성하지 않으므로 불필요한 부하가 걸리지 않는다.
- 어떤 경우에도 null 포인터를 반환하지 않는다.

단점

- MyServiceImpl의 생성자에 명시적으로 의존한다.
- service가 null인 경우와 아닌 경우, 두 방식을 모두 테스트해야한다.
- MyServiceImpl이 모든 상황에 적합한 객체인지 모른다.

시스템 제작과 시스템 사용을 분리하라

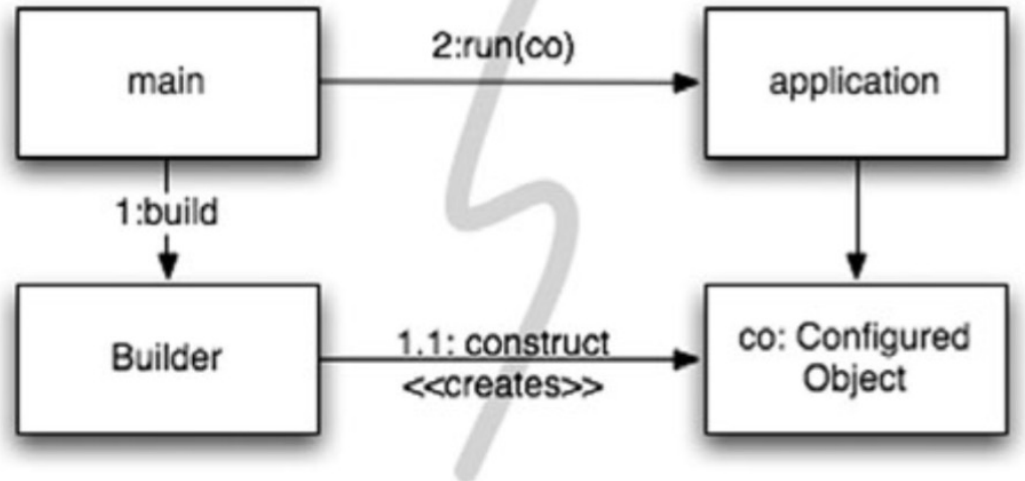


Figure 11-1

Separating construction in `main()`

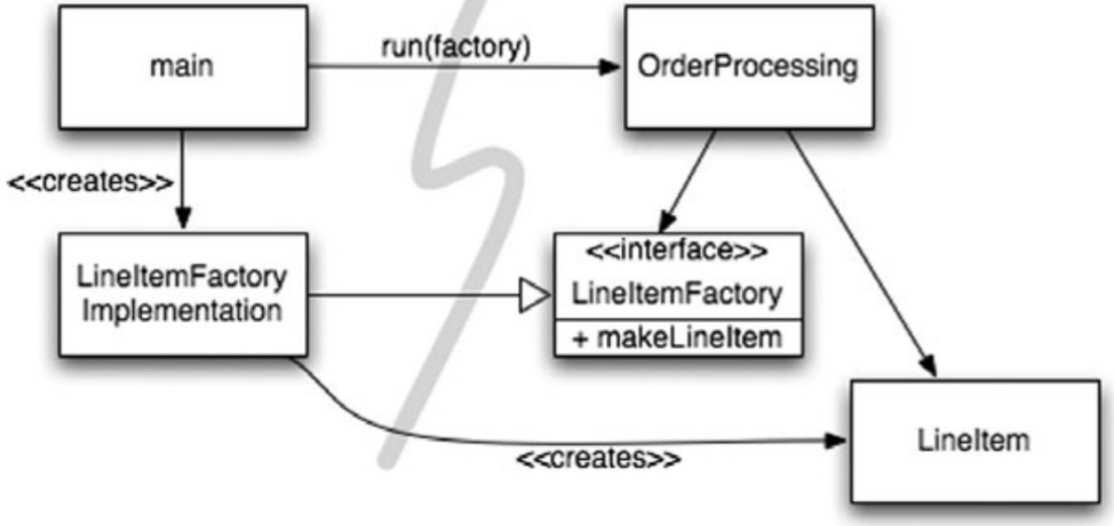


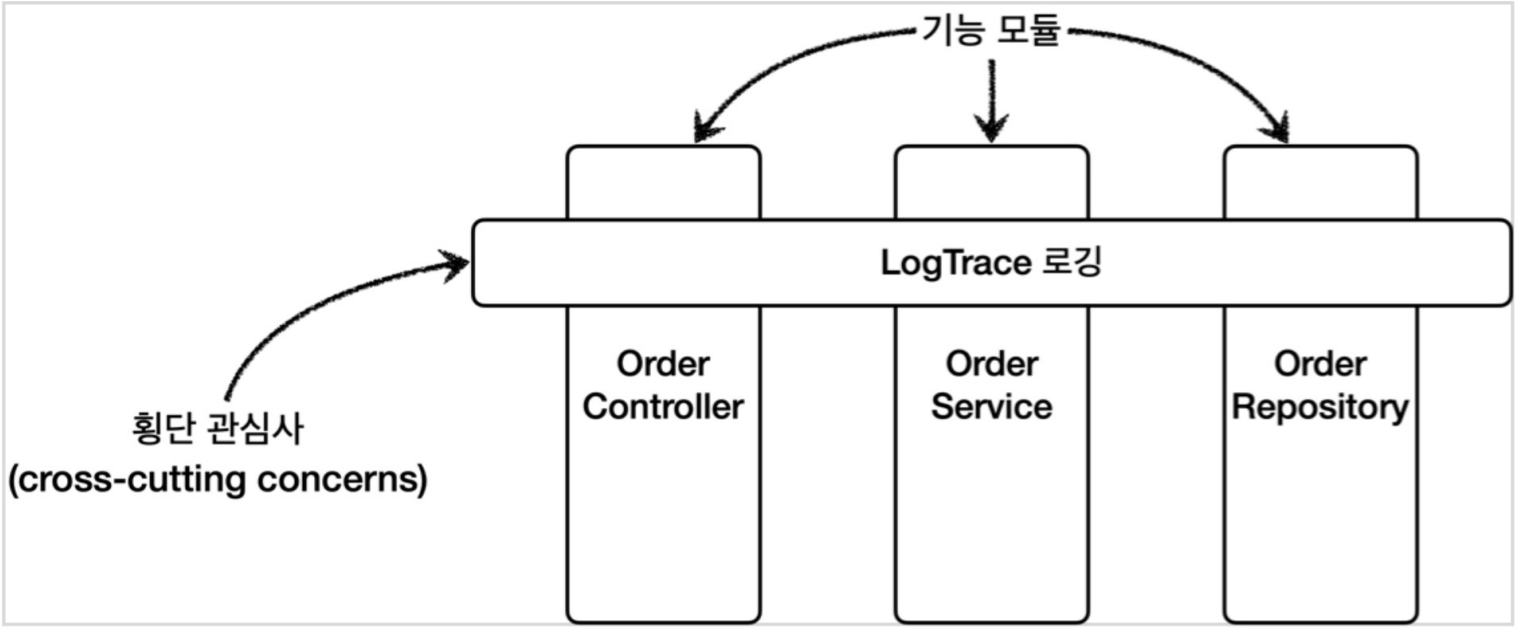
Figure 11-2

Separation construction with factory

횡단 관심사(Cross-cutting concerns)

핵심 기능은 해당 객체가 제공하는 고유의 기능
- EX) 주문로직, 회원가입, 로그인

부가 기능은 핵심 기능을 보조하기 위해 제공되는 기능
- EX) 로그 로직, 트랜잭션 기능 등
- 부가 기능은 단독으로 사용되지 않고, 핵심 기능과 함께 사용된다.



프록시(Proxy)

직접 호출 : Client -> Server

간접 호출 : Client -> **Proxy** -> Server

프록시 역할

권한에 따른 접근 차단, 캐싱, 지연로딩을 수행하는 **접근 제어**

서버의 기능에 다른 기능까지 추가해주는 **부가 기능 추가** (요청, 응답값을 변형, 로그 기록 등)

대리자가 또 다른 대리자를 호출하는 **프록시 체인**

프록시(Proxy) – JDK 동적 프록시

```
public interface AInterface {
  String call();
}
```

```
public class AImpl implements AInterface {
  public String call() {
    System.out.println("A 호출");
    return "a";
  }
}
```

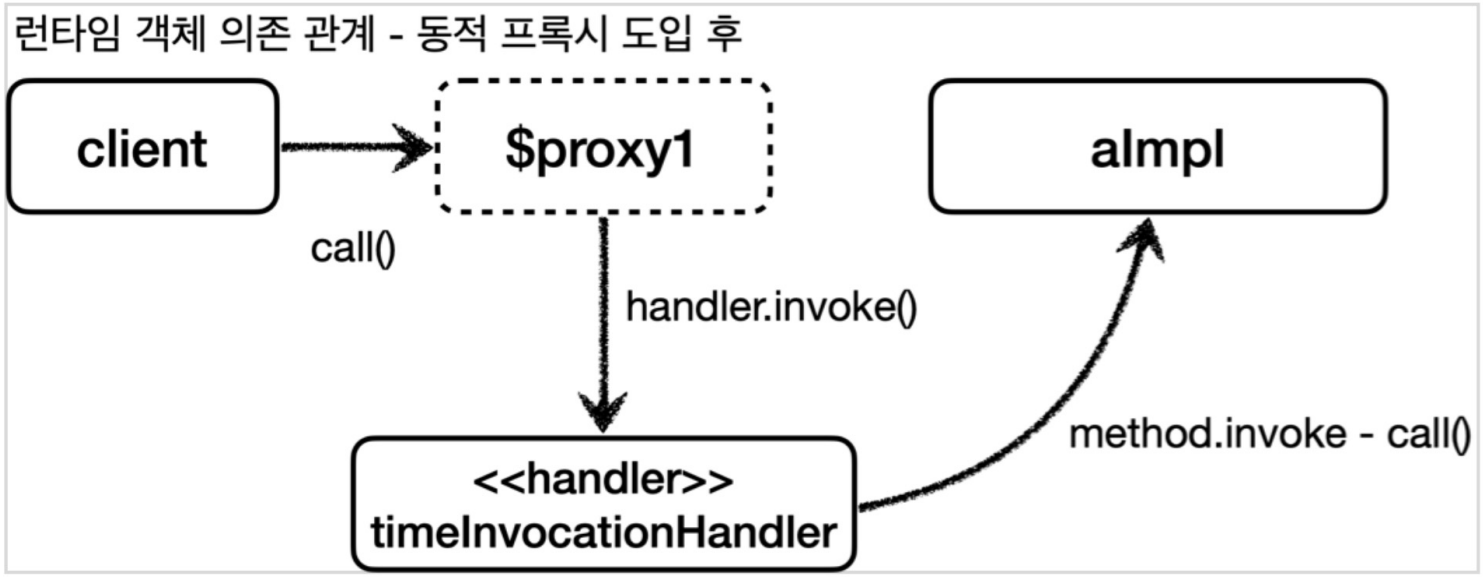
프록시(Proxy) - JDK 동적 프록시

```
public class TimeInvocationHandler implements InvocationHandler {
    private final Object target;

    public TimeInvocationHandler(Object target) {
        this.target = target;
    }

    @Override
    public Object invoke(Object proxy, Method method, Object[] args) throws Throwable {
        log.info("TimeProxy 실행");
        long startTime = System.currentTimeMillis();

        Object result = method.invoke(target, args);

        long endTime = System.currentTimeMillis();
        long resultTime = endTime-startTime;
        log.info("TimeProxy 종료 resultTime={}", resultTime);
        return result;
    }
}
```

프록시(Proxy) - JDK 동적 프록시

```
public class Main {
  public static void main(String[] args) {
    AInterface target = new AImpl();
    TimeInvocationHandler handler = new TimeInvocationHandler(target);

    AInterface proxy = (AInterface) Proxy.newProxyInstance(AInterface.class.getClassLoader(),
new Class[]{AInterface.class}, handler);
    proxy.call();
  }
}
```

```
// 실행 결과
TimeProxy 실행
A 호출
TimeProxy 종료 resultTime = 0
```



프록시(Proxy) - CGLIB

```
public class ConcreteService {
    public void call(){
        log.info("ConcreteService 호출");
    }
}
```

프록시(Proxy) - CGLIB

```
public class TimeMethodInterceptor implements MethodInterceptor {
    private final Object target;

    public TimeMethodInterceptor(Object target) {
        this.target = target;
    }

    @Override
    public Object intercept(Object obj, Method method, Object[] args, MethodProxy methodProxy) throws Throwable {
        log.info("TimeProxy 실행");
        long startTime = System.currentTimeMillis();

        Object result = methodProxy.invoke(target, args);

        long endTime = System.currentTimeMillis();
        long resultTime = endTime-startTime;
        log.info("TimeProxy 종료 resultTime={}", resultTime);
        return result;
    }
}
```

프록시(Proxy) - CGLIB

```
void cglib() {
    ConcreteService target = new ConcreteService();

    Enhancer enhancer = new Enhancer();
    enhancer.setSuperclass(ConcreteService.class);
    enhancer.setCallback(new TimeMethodInterceptor(target));
    ConcreteService proxy = (ConcreteService) enhancer.create();
    proxy.call();
}
```

```
//실행 결과
TimeMethodInterceptor - TimeProxy 실행
ConcreteService - ConcreteService 호출
TimeMethodInterceptor - TimeProxy 종료
resultTime=9
```