

### 3. SOLID

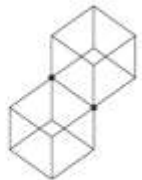
---



# JAVA

## 개체 지향 디자인 패턴

UML과 GoF 디자인 패턴 핵심 10가지로 배우는



# 학습목표

---

## 학습목표

- SOLID\*의 개념 이해하기
- SRP 이해하기
- OCP 이해하기
- LSP 이해하기
- DIP 이해하기
- ISP 이해하기

## 3.1 SRP(Single Responsibility Principle)

---

- ❖ 단일책임의 원칙
- ❖ 객체는 단 하나의 책임만을 가져야 한다



## 3.1 SRP(Single Responsibility Principle)

Keypoint\_ 책임 = 해야 하는 것

책임 = 할 수 있는 것

책임 = 해야 하는 것을 잘 할 수 있는 것



다음 직원 정보를 담당하는 Employee 클래스에는 4가지의 주요 메서드가 존재한다.

- ① calculatePay() : 회계팀에서 급여를 계산하는 메서드
- ① reportHours() : 인사팀에서 근무시간을 계산하는 메서드
- ① saveDatabase() : 기술팀에서 변경된 정보를 DB에 저장하는 메서드
- ① calculateExtraHour() : 초과 근무 시간을 계산하는 메서드 (회계팀과 인사팀에서 공유하여 사용)

너무 많은 책임

bad

## Employee 클래스 코드

```
class Employee {
    String name;
    String position;

    Employee(String name, String position) {
        this.name = name;
        this.position = position;
    }

    // * 초과 근무 시간을 계산하는 메서드 (두 팀에서 공유하여 사용)
    void calculateExtraHour() {
        // ...
    }

    // * 급여를 계산하는 메서드 (회계팀에서 사용)
    void calculatePay() {
        // ...
        this.calculateExtraHour();
        // ...
    }

    // * 근무시간을 계산하는 메서드 (인사팀에서 사용)
    void reportHours() {
        // ...
        this.calculateExtraHour();
        // ...
    }
}
```

# 변경

---

## ❖ 책임은 변경이유이다

- 책임이 많다는 것은 변경될 여지가 많다는 의미이다
- 책임을 많이 질수록 클래스 내부에서 서로 다른 역할을 수행하는 코드끼리 강하게 결합될 가능성이 높아진다.
- ❖ 회계팀에서 급여를 계산하는 방식을 변경함에 따라 초과 근무 시간을 계산하는 `calculateExtraHour()` 알고리즘 업데이트 필요
- ❖ 이 때 어떤 문제가 발생할까?
  - ❖ 인사팀의 `reportHours()` 메소드에 영향
  - ❖ 인사팀에서는 결과 데이터가 이상하다고 개발팀에 새로 요청을 보내게 될 것임

# 책임 분리

그림 3-1 변경의 영향

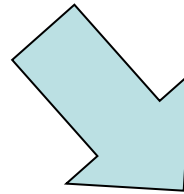
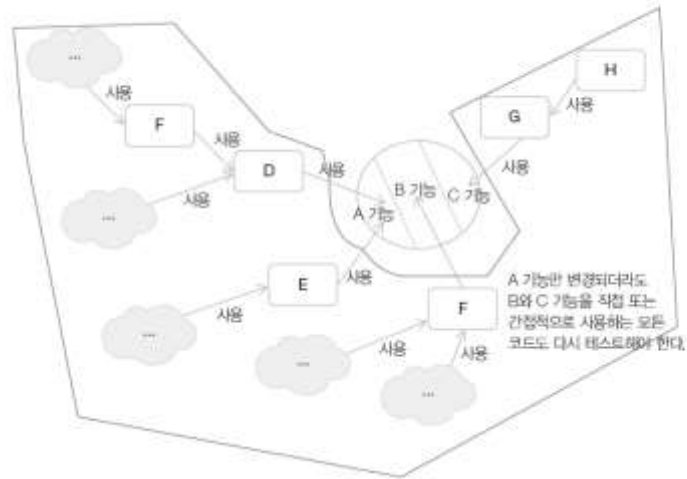


그림 3-2 책임 분리



## 코드 수정 결과

```
class PayCalculator {
    // * 초과 근무 시간을 계산하는 메서드
    void calculateExtraHour() {
        // ...
    }
    void calculatePay() {
        // ...
        this.calculateExtraHour();
        // ...
    }
}

class HourReporter {
    // * 초과 근무 시간을 계산하는 메서드
    void calculateExtraHour() {
        // ...
    }
    void reportHours() {
        // ...
        this.calculateExtraHour();
        // ...
    }
}

// * 기술팀에서 사용되는 전용 클래스
class EmployeeSaver {
    void saveDatabase() {
        // ...
    }
}
```

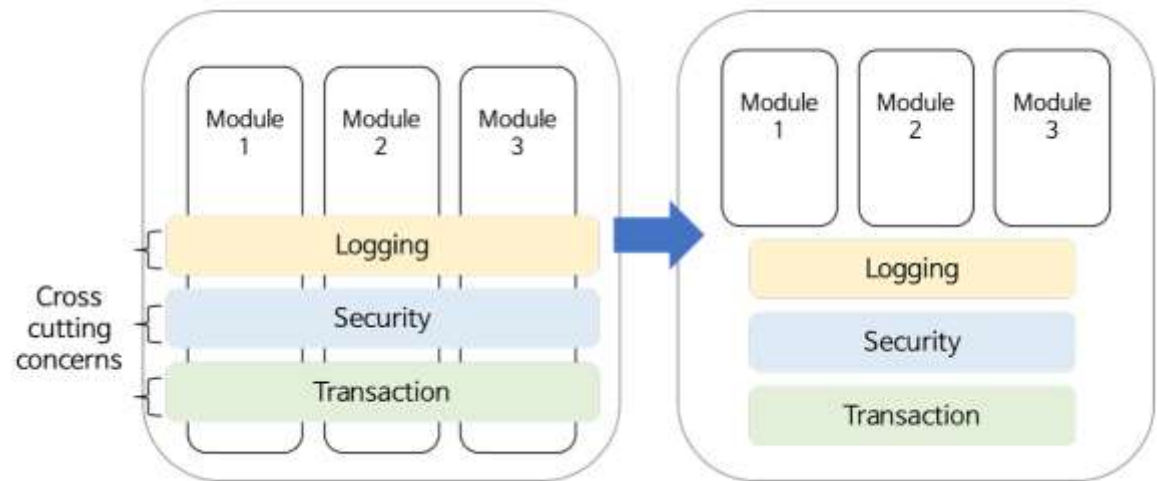
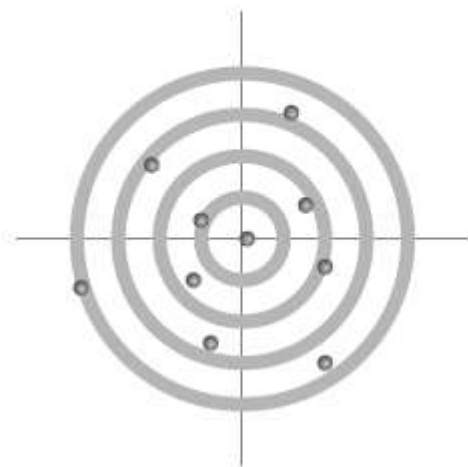


# 산탄총수술

❖ 하나의 책임이 여러 곳에 분산 bad

- 변경 이유가 발생했을 때 변경할 곳이 많음
- 변경될 곳을 빠짐 없이 찾아 일관되게 변경해야 함

### 그림 3-4 산탄총 수술



이미지 출처 : <https://inpa.tistory.com/entry/OOP-%F0%9F%92%A0-%EC%95%84%EC%A3%BC-%EC%89%BD%EA%B2%8C-%EC%9D>

## 또 다른 예제

### ❖ Book 클래스(전자책)를 살펴보자

```
public class Book {  
  
    private String name;  
    private String author;  
    private String text;  
  
    //constructor, getters and setters  
  
    // methods that directly relate to the book properties  
    public String replaceWordInText(String word, String replacementWord){  
        return text.replaceAll(word, replacementWord);  
    }  
  
    public boolean isWordInText(String word){  
        return text.contains(word);  
    }  
}
```

책의 텍스트 일부를  
교체하는 메소드

책의 텍스트 안에  
특정 단어가 있는지  
확인하는 메소드

## 그렇다면, 책을 출력하여 읽는 기능은?

---

❖ 아래와 같은 메소드를 Book 클래스에 추가하면 될까? *bad*

```
void printTextToConsole(){  
    // our code for formatting and printing the text  
}
```

# SRP를 지키기 위한 구현

---

❖ 책을 화면에 출력하는 기능만 별도로 처리하는 클래스 구현 *good*

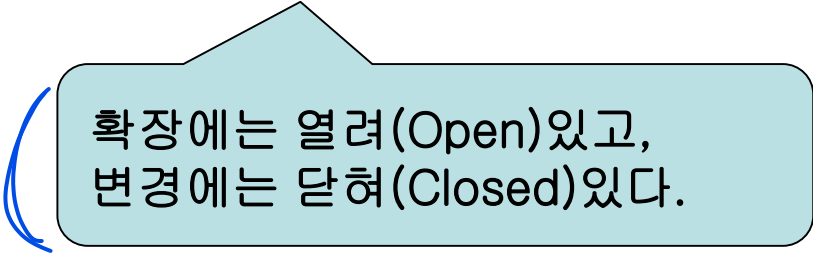
```
public class BookPrinter {  
  
    // methods for outputting text  
    void printTextToConsole(String text){  
        //our code for formatting and printing the text  
    }  
  
    void printTextToAnotherMedium(String text){  
        // code for writing to any other location..  
    }  
}
```

## 3.2 OCP(Open/Closed Principle)

---

### ❖ 개방폐쇄원칙

- 기존의 코드를 변경하지 않으면서 기능을 추가할 수 있도록 설계가 되어야 한다
- 클래스를 변경하지 않고도 <sub>closed</sub> 대상 클래스의 환경을 변경할 수 있도록 설계



확장에는 열려(Open)있고,  
변경에는 닫혀(Closed)있다.

# 기타를 만들어 봅시다

---

## ❖ Guitar 클래스

```
public class Guitar {  
  
    private String make;  
    private String model;  
    private int volume;  
  
    //Constructors, getters & setters  
}
```

# 로큰롤 느낌을 살리고 싶다면?

---

- ❖ Guitar 클래스에 불꽃 패턴을 직접 추가?
- ❖ 추가했을 때 기존 어플리케이션에서 어떤 오류가 발생할지 예측이 가능한가?

# OCP 원칙을 고수한 불꽃 패턴 추가

---

```
public class SuperCoolGuitarWithFlames extends Guitar {  
    private String flameColor;  
  
    //constructor, getters + setters  
}
```



# 체크포인트

- ❖ 다음 코드는 오후 10시가 되면 MP3를 작동시켜 음악을 연주한다. 그러나 이 코드가 제대로 작동하는지 테스트하려면 저녁 10시까지 기다려야 한다. OCP를 적용해 이 문제를 해결하는 코드를 작성하라

22시로  
가장제로  
바꿔서  
테스트  
하도록 했다

```
import java.util.Calendar;

public class TimeReminder {
    private MP3 m;

    public void reminder() {
        Calendar cal=Calendar.getInstance();
        m = new MP3();
        int hour = cal.get(Calendar.HOUR_OF_DAY);

        if (hour >= 22) {
            m.playSong();
        }
    }
}
```

## 또 다른 예제(2)

---

- ❖ Report를 요구받은 형식에 따라 생성하는 프로그램
- ❖ 만약 여기에 XML같은 다른 형식 생성이 필요하다면?

```
public class ReportGenerator {  
    public void generateReport(String type) {  
        if (type.equals("PDF")) {  
            System.out.println("Generating PDF report...");  
        } else if (type.equals("HTML")) {  
            System.out.println("Generating HTML report...");  
        }  
        // If we need to add another format, we have to modify this method.  
    }  
}
```

## 또 다른 예제(2)

- ❖ 문서 생성부는 interface로 껍데기만 만들어두고 필요할 때 구현

```
public interface Report {  
    void generate();  
}  
  
public class PDFReport implements Report {  
    @Override  
    public void generate() {  
        System.out.println("Generating PDF report...");  
    }  
}  
  
public class HTMLReport implements Report {  
    @Override  
    public void generate() {  
        System.out.println("Generating HTML report...");  
    }  
}  
  
public class XMLReport implements Report {  
    @Override  
    public void generate() {  
        System.out.println("Generating XML report...");  
    }  
}
```

## 3.3 LSP(Liskov Substitution Principle)

### ❖ 리스코프 치환 원칙

- LSP는 부모 클래스와 자식 클래스 사이의 행위가 일관성이 있어야 한다는 의미다

자식 클래스는 최소한  
부모 클래스가 하는 일은  
다 할 수 있어야 한다.



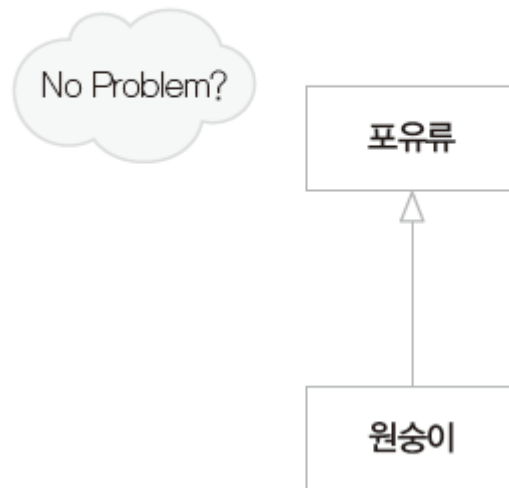
“A type hierarchy is composed of subtypes and supertypes. The intuitive idea of a subtype is one whose objects provide all the behavior of another type (the supertype) plus something extra. What is wanted here is something like the following substitution property: if for each object  $o_1$  of type  $S$  there is an object  $o_2$  of type  $T$  such that for all programs  $P$  defined in terms of  $T$ , the behavior of  $P$  is unchanged when  $o_1$  is substituted for  $o_2$ , then  $s$  is a subtype of  $T$ .”

– Barbara Liskov(1988)

# LSP

- ❖ LSP를 만족하면 프로그램에서 부모 클래스의 인스턴스 대신에 자식 클래스의 인스턴스로 대체해도 프로그램의 의미는 변화되지 않는다.

그림 3-8 원숭이 is a kind of 포유류



# LSP

## 포유류

- ❖ 포유류는 알을 낳지 않고 새끼를 낳아 번식한다.
- ❖ 포유류는 젖을 먹여서 새끼를 키우고 폐를 통해 호흡한다.
- ❖ 포유류는 체온이 일정한 정온 동물이며 털이나 두꺼운 피부로 덮여 있다.

## 원숭이

- ❖ 원숭이는 알을 낳지 않고 새끼를 낳아 번식한다.
- ❖ 원숭이는 젖을 먹여서 새끼를 키우고 폐를 통해 호흡한다.
- ❖ 원숭이는 체온이 일정한 정온 동물이며 털이나 두꺼운 피부로 덮여 있다.

그림 3-8 원숭이 is a kind of 포유류



# 원숭이를 구현하면?

---

## 포유류

```
public abstract class Mammal { 0개의 사용위치  
    public abstract void viviparity(); 0개  
    public abstract void breastfeeding();  
}
```

## 원숭이

```
public class Monkey extends Mammal { 0개의 사용위치  
  
    @Override 0개의 사용위치  
    public void viviparity() {  
        System.out.println("나는 엄마 뱃속에서 태어났어");  
    }  
  
    @Override 0개의 사용위치  
    public void breastfeeding() {  
        System.out.println("나는 태어나서 엄마젖을 먹고 자랐어");  
    }  
}
```

# 오리너구리

---

## 포유류

- ❖ 포유류는 알을 낳지 않고 새끼를 낳아 번식한다.
- ❖ 포유류는 젖을 먹여서 새끼를 키우고 폐를 통해 호흡한다.
- ❖ 포유류는 체온이 일정한 정온 동물이며 털이나 두꺼운 피부로 덮여 있다.

## 오리너구리

- ❖ 오리너구리는 새끼를 낳지 않고 알을 낳아 번식한다.
- ❖ 오리너구리는 젖을 먹여서 새끼를 키우고 폐를 통해 호흡한다.
- ❖ 오리너구리는 체온이 일정한 정온 동물이며 털이나 두꺼운 피부로 덮여 있다.

그림 3-9 오리너구리





# 오리너구리를 구현하면?

---

포유류

```
public abstract class Mammal { 0개의 사용위치  
    public abstract void viviparity(); 0개  
    public abstract void breastfeeding();  
}
```

오리너구리

bad

```
public class Platypus extends Mammal { 0개의 사용위치  
    @Override 0개의 사용위치  
    public void viviparity() {  
        throw new UnsupportedOperationException("오리너구리는 알에서 태어나!");  
    }  
  
    @Override 0개의 사용위치  
    public void breastfeeding() {  
        System.out.println("나는 태어나서 엄마젖을 먹고 자랐어");  
    }  
}
```

## 또 다른 예제

---

- ❖ 앞서 살펴본 포유류의 ‘태생’은 오리너구리에게 적용되지 않으므로 태생이라는 부분을 따로 빼서 인터페이스로 만들고 태생에 해당하는 원숭이만 받아서 구현하도록 코드 수정

# 다시 원숭이를 구현하면?

---

포유류

```
public interface Viviparity {  
    void viviparity(); 0개의 사  
}
```

```
public abstract class Mammal { 2개 사용 위치  
    public abstract void breastfeeding();  
}
```

원숭이

```
public class Monkey extends Mammal implements Viviparity { 0개  
  
    @Override 0개의 사용위치  
    public void viviparity() {  
        System.out.println("나는 엄마 뱃속에서 태어났어");  
    }  
  
    @Override 0개의 사용위치  
    public void breastfeeding() {  
        System.out.println("나는 태어나서 엄마젖을 먹고 자랐어");  
    }  
}
```

# 오리너구리를 구현하면?

---

## 포유류

```
public abstract class Mammal { 2개 사용 위치
    public abstract void breastfeeding();
}
```

## 오리너구리

```
public interface Ovoviviparity {
    void ovoviviparity(); 0개의 사
}
```

```
public class Platypus extends Mammal implements Ovoviviparity{
    @Override 0개의 사용위치
    public void breastfeeding() {
        System.out.println("나는 태어나서 엄마젖을 먹고 자랐어");
    }

    @Override 0개의 사용위치
    public void ovoviviparity() {
        System.out.println("나는 알에서 태어났어");
    }
}
```

## 또 다른 예제

- ❖ 참새를 구현하기 위해 새 클래스를 먼저 구현하고 그것을 상속받아 참새를 구현하였다.

```
public abstract class Bird { 1개 사용 위치 1개
    public abstract void ovoviviparity();
    public abstract void fly(); 0개의 사용위치
}
```

```
public class Sparrow extends Bird{ 0개의 사용위치
    @Override 0개의 사용위치
    public void ovoviviparity() {
        System.out.println("나는 알에서 태어났어");
    }

    @Override 0개의 사용위치
    public void fly() {
        System.out.println("나는 날 수 있어");
    }
}
```

## 또 다른 예제

---

- ❖ 그런데 만약, 펭귄 클래스를 만들어야 한다면 어떻게 해야 할까?
  - 펭귄은 날 수 없다
- ❖ 앞에서 배운 내용을 바탕으로 수정이 필요한 부분을 수정한 후 펭귄 클래스를 구현해보자.

## 3.5 ISP(Interface Segregation Principle)

### ❖ 인터페이스 분리 원칙

사용하지 않을 메소드를 구현하도록  
강요받지 않아야 한다

- 인터페이스를 클라이언트에 특화되도록 분리시키라는 설계 원칙
- 클라이언트의 관점에서 클라이언트 자신이 이용하지 않는 기능에는 영향을 받지 않아야 한다는 내용이 담겨 있다.

# 복합기 인터페이스

---

```
public interface MultiFuncDevice {  
    void print(); 0개의 사용위치 1개 구  
    void fax(); 0개의 사용위치  
    void copy(); 0개의 사용위치  
}
```



# 복합기 인터페이스로 프린터 구현

```
public class Printer implements MultiFuncDevice{ 0개의 사용위치
    public void print(){ 0개의 사용위치
        System.out.println("나는 프린트중이야");
    }

    @Override 0개의 사용위치 fax와 copier는 사용하지 않을 건데
    public void fax() {
        throw new UnsupportedOperationException("난 프린터라 팩스는 못해");
    }

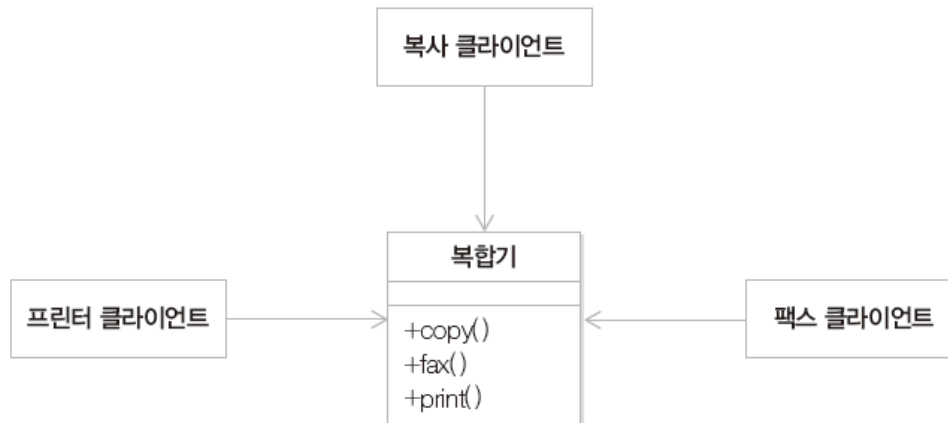
    @Override 0개의 사용위치 굳이 구현해야 할까?
    public void copy() {
        throw new UnsupportedOperationException("난 프린터라 복사는 못해");
    }
}
```

## 3.5 ISP(Interface Segregation Principle)

---

### ❖ 인터페이스 분리 원칙

그림 3-11 복합기의 클래스 다이어그램



# 각 기능을 별도의 interface로 구성

---

```
public interface Printer {  
    void print(); 0개의 사용위치  
}
```

```
public interface Fax {  
    void fax(); 0개의 사-  
}
```

```
public interface CopyMachine {  
    void copy(); 0개의 사용위치 2  
}
```

## 필요할 때 클라이언트에서 interface를 구현

---

```
public class MyCopyMachine implements CopyMachine{  
  
    @Override 0개의 사용위치  
    public void copy() {  
        System.out.println("복사를 시작합니다.");  
    }  
}
```

```
public class MyPrinter implements Printer{ 0개의 사용위치  
  
    @Override 0개의 사용위치  
    public void print() {  
        System.out.println("프린트를 시작합니다.");  
    }  
}
```

# 인터페이스는 복합기를 만들려면? 다중 상속 가능 (추상X)

```
public class MultifunctionDevice implements Printer, CopyMachine, Fax{

    @Override 0개의 사용위치
    public void copy() {
        System.out.println("난 복합기. 복사도 할 수 있지");
    }

    @Override 0개의 사용위치
    public void fax() {
        System.out.println("난 복합기. 팩스도 보내고 받을 수 있지");
    }

    @Override 0개의 사용위치
    public void print() {
        System.out.println("난 복합기. 프린트도 할 수 있지");
    }
}
```

## 3.5 ISP(Interface Segregation Principle)

### ❖ 인터페이스 분리 원칙

그림 3-11 복합기의 클래스 다이어그램

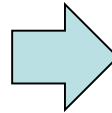
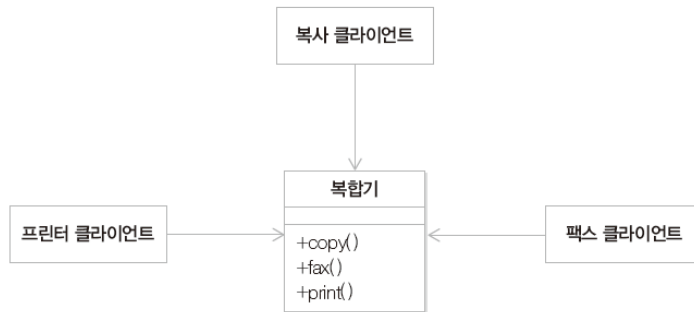
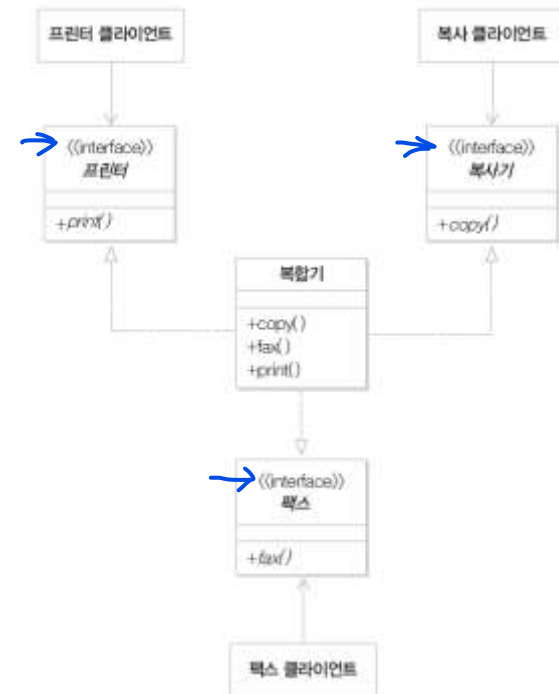


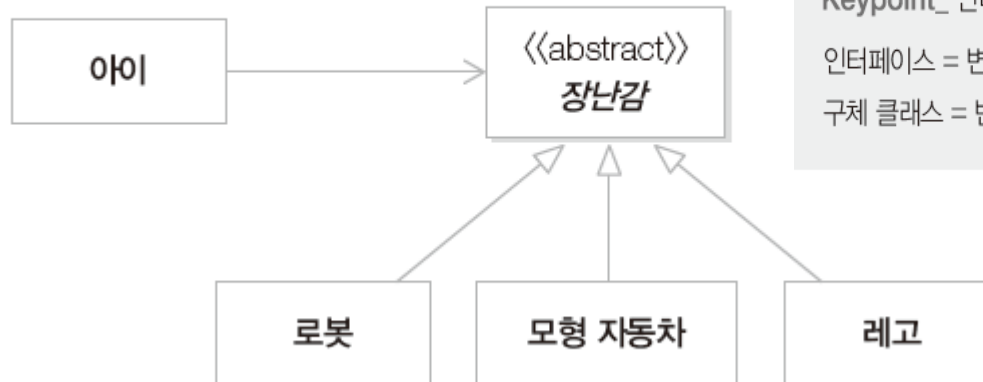
그림 3-12 복합기 클래스에 ISP를 적용한 예



## 3.4 DIP(Dependency Inversion Principle)

- ❖ DIP는 의존 관계를 맺을 때 변화하기 쉬운 것 또는 자주 변화하는 것 보다는 변화하기 어려운 것, 거의 변화가 없는 것에 의존하라는 원칙
  - 정책, 전략과 같은 어떤 큰 흐름이나 개념 같은 추상적인 것은 변하기 어려운 것에 해당하고 구체적인 방식, 사물 등과 같은 것은 변하기 쉬운 것으로 구분

그림 3-10 장난감 클래스에 DIP를 적용한 예



고수준 모듈이 저수준 모듈에 의존해서는 안된다

Keypoint\_ 인터페이스나 추상 클래스와 의존 관계를 맺도록 설계해야 한다.

인터페이스 = 변하지 않는 것

구체 클래스 = 변하기 쉬운 것

고수준: 추상적  
저수준: 구체적

# 의존성 주입

코드 3-5

```
public class Kid {  
    private Toy toy;  
  
    public void setToy(Toy toy) {  
        this.toy = toy;  
    }  
  
    public void play() {  
        System.out.println(toy.toStr  
    }  
}
```

코드 3-6

```
public class Robot extends Toy {  
    public String toString() {  
        return "Robot";  
    }  
}
```

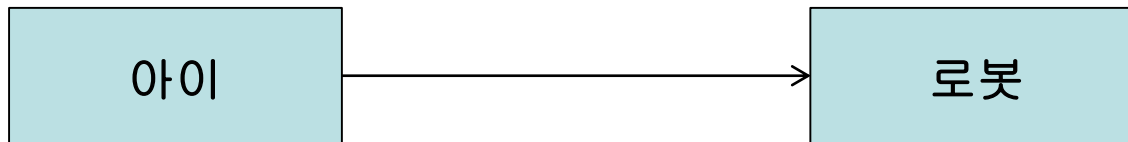
코드 3-7

```
public class Main {  
    public static void main(String[] args) {  
        Toy t = new Robot();  
        Kid k = new Kid();  
        k.setToy(t);  
        k.play();  
    }  
}
```



# 만약 Toy가 추상화되지 않은 상태라면?

---



## 코드 구현(잘못된 예시)

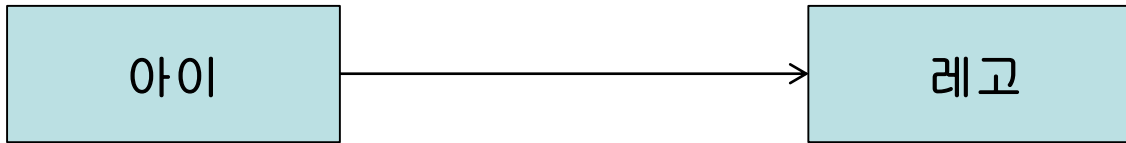
---

*bad*

```
public class Kid{  
    private Robot toy;  
  
    public void setToy(Robot toy){  
        this.toy=toy;  
    }  
  
    public void play(){  
        System.out.println(toy.toString());  
    }  
}
```

## 코드 구현(잘못된 예시)

❖ 만약, 아이가 레고를 가지고 놀다면?



*bad*

```
public class Kid{  
    private Lego toy;  
  
    public void setToy(Lego toy){  
        this.toy=toy;  
    }  
  
    public void play(){  
        System.out.println(toy.toString());  
    }  
}
```

## 또 다른 예제

- ❖ 다음 코드는 에어컨과 리모콘을 클래스로 구현한 것이다. 잘 살펴보고 수정이 필요한 부분을 찾아서 코드를 수정해보자.

```
public class AirConditioner { 2개 사용 위치
    public void airconditioning(){ 0개의 사용위치
        System.out.println("에어컨이 가동 중입니다.");
    }

    public void dehumidification(){ 0개의 사용위치
        System.out.println("제습운전 중입니다.");
    }
}
```

+ private RemoteControl rc;

에어컨에서 리모콘 담당하는 게 더 낫다

```
public class RemoteControl { 0개의 사용위치
    private AirConditioner ac; 1개 사용 위치

    public void RemoteControl(AirConditioner ac){
        this.ac=ac;
    }

    public void turnOn(){ 0개의 사용위치
        System.out.println("전원을 켭니다.");
    }

    public void turnOff(){ 0개의 사용위치
        System.out.println("전원을 끕니다.");
    }
}
```