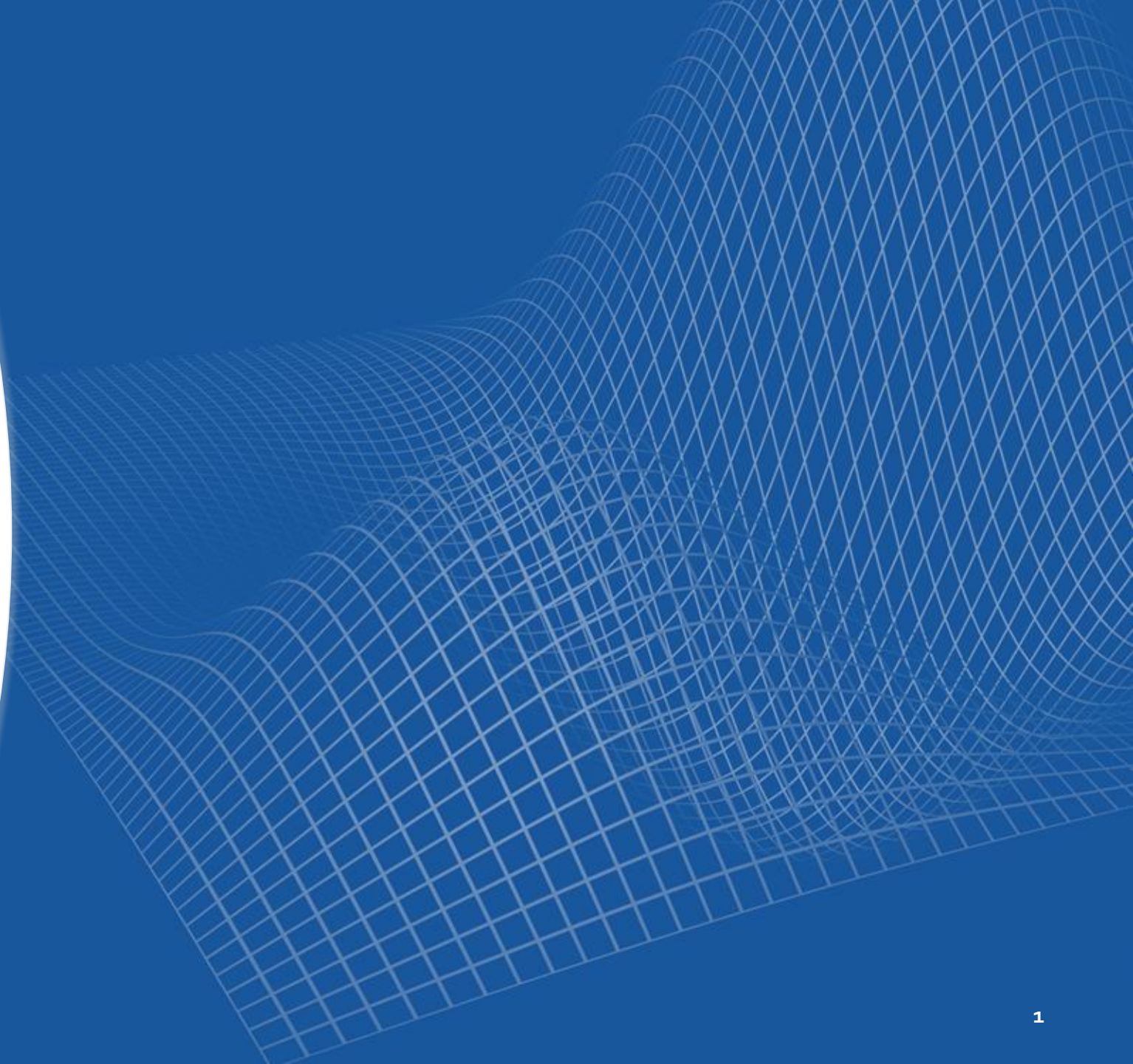
SOLID



SRP

- 단일책임의 원칙
- 객체는 단 하나의 책임만을 가져야 한다

Keypoint_ 책임 = 해야 하는 것
책임 = 할 수 있는 것
책임 = 해야 하는 것을 잘 할 수 있는 것

코드 3-1

```
public class Student {
  public void getCourses() { ... }
  public void addCourse(Course c) { ... }

  public void save() { ... }
  public Student load() { ... }
  public void printOnReportCard() { ... }
  public void printOnAttendanceBook() { ... }
}
```

너무 많은 책임

변경

- **책임은 변경이유이다**
  - 책임이 많다는 것은 변경될 여지가 많다는 의미이다
  - 책임을 많이 질수록 클래스 내부에서 서로 다른 역할을 수행하는 코드끼리 강하게 결합될 가능성이 높아진다.
  - 데이터베이스의 스키마가 변경된다면 Student 클래스도 변경되어야 하는가?
  - 학생이 지도 교수를 찾는 기능이 추가되어야 한다면 Student 클래스는 영향을 받는가?
  - 학생 정보를 성적표와 출석부 이외의 형식으로 출력해야 한다면 어떻게 해야 하는가?

책임분리

그림 3-1 변경의 영향

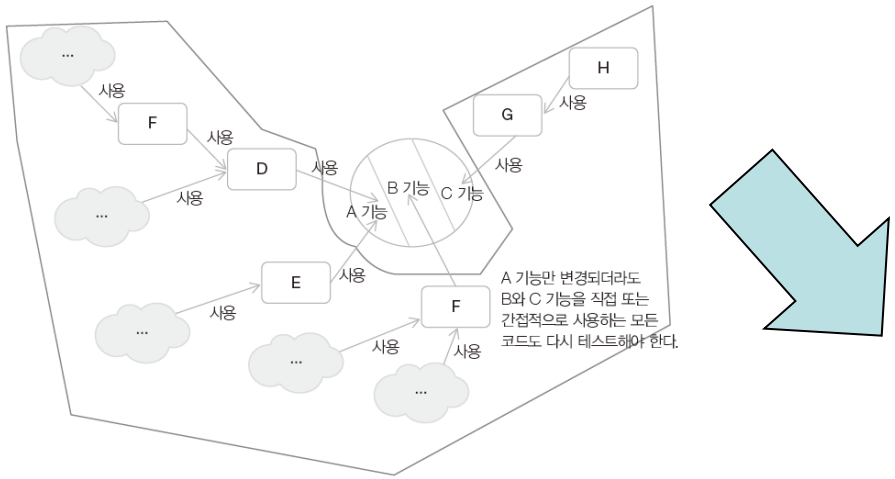
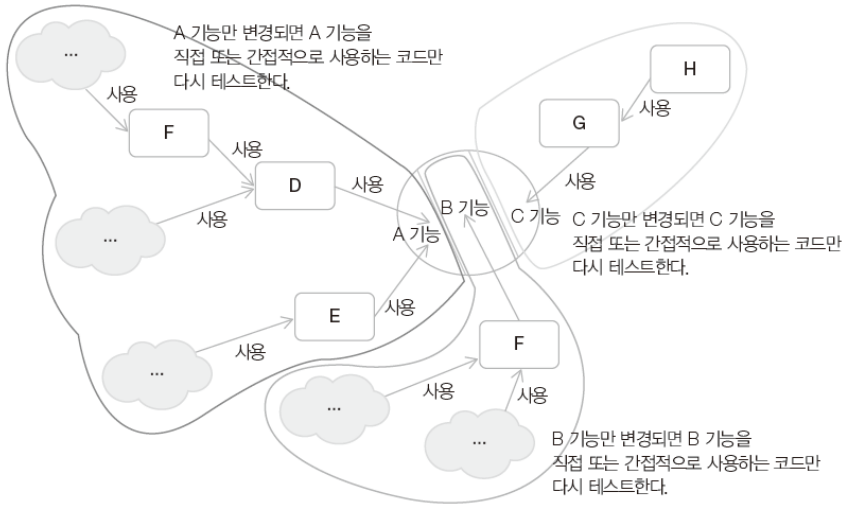


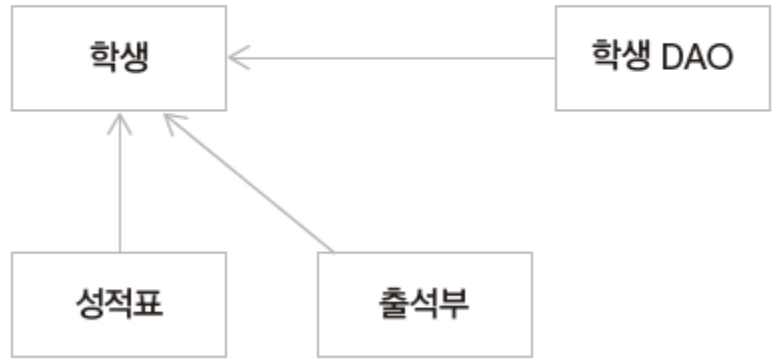
그림 3-2 책임 분리



학생 클래스

- **변경 사유**
  - 학생의 고유 정보,
  - 데이터베이스 스키마
  - 출력 형식의 변화

그림 3-3 개선된 디자인



산탄총 수술

- **하나의 책임이 여러 곳에 분산**
  - 변경 이유가 발생했을 때 변경할 곳이 많음
  - 변경될 곳을 빠짐 없이 찾아 일관되게 변경해야 함

그림 3-4 산탄총 수술

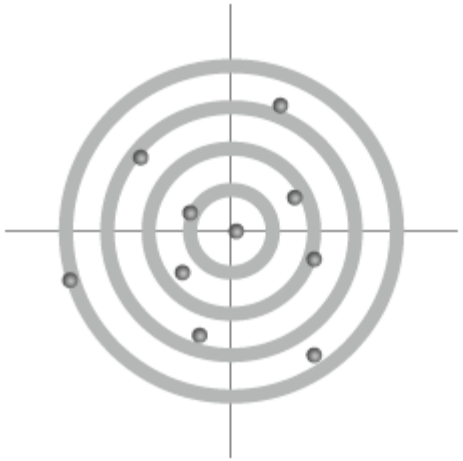


그림 3-5 횡단 관심

OCP

❖ 개방폐쇄원칙

- 기존의 코드를 변경하지 않으면서 기능을 추가할 수 있도록 설계가 되어야 한
- 클래스를 변경하지 않고도closed 대상 클래스의 환경을 변경할 수 있도록 설계

그림 3-6 성적표나 출석부에 학생을 출력하는 기능을 사용

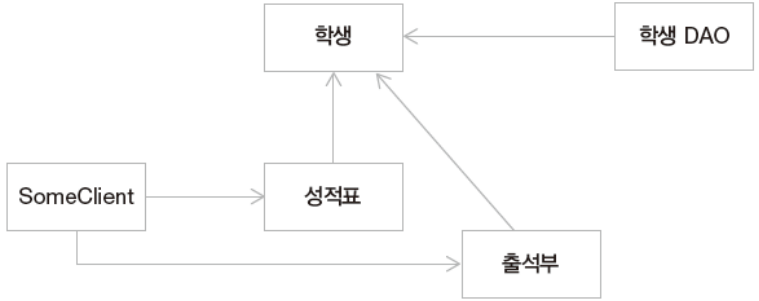
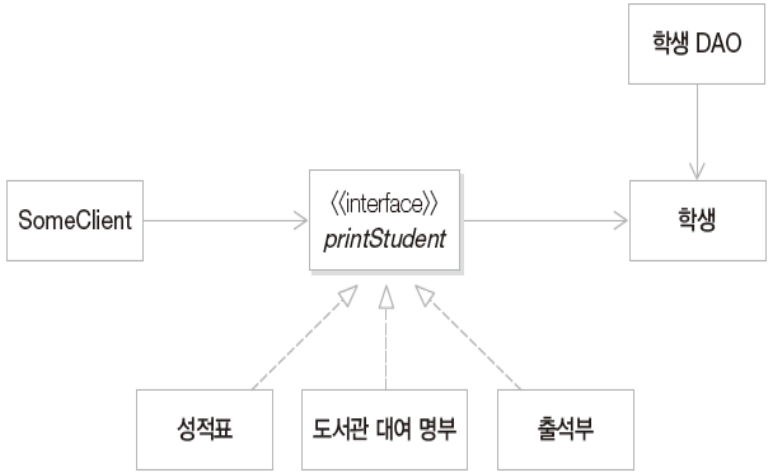


그림 3-7 OCP를 만족하는 설계



실습)

❖ 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.

```
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();
    }
  }
}
```

LSP

❖ 리스코프 치환 원칙

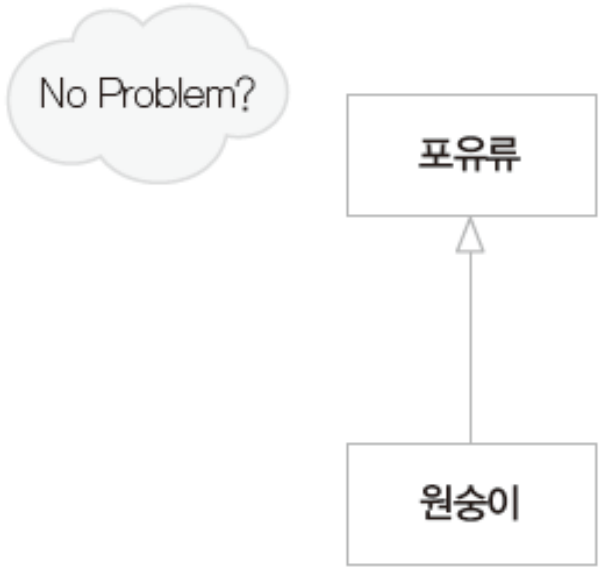
- LSP는 부모 클래스와 자식 클래스 사이의 **행위가 일관성**이 있어야 한다는 의미다

"A type hierarchy is composed of subtypes and supertypes. The intuitive idea of a subtytpe 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 o1 of type S there is an object o2 of type T such that for all programs P defined in terms of T, the behavior of P is unchanged when o1 is substituted for o2, then s is a subtype of T."

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

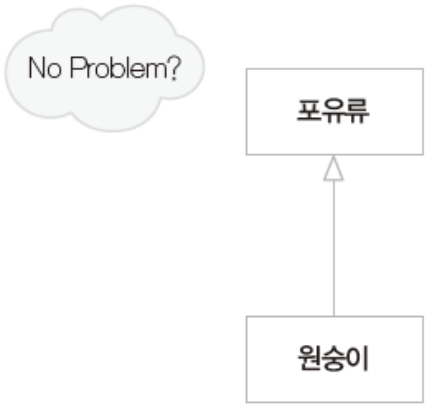
포유류

- 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



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

그림 3-9 오리너구리

행위일관성

❖ LSP를 만족하는 가장 단순한 방법은 재정의를 하지 않는 것이다

코드 3-2

```
public class Bag {
  private int price;

  public void setPrice(int price) {
    this.price = price;
  }

  public int getPrice() {
    return price;
  }
}
```

코드 3-3

```
public class DiscountedBag extends Bag {
  private double discountedRate = 0;

  public void setDiscounted(double discountedRate) {
    this.discountedRate = discountedRate;
  }

  public void applyDiscount(int price) {
    super.setPrice(price - (int)(discountedRate * price));
  }
}
```

가격은 설정된 가격 그대로 조회된다.

LSP

표 3-1 Bag 클래스와 DiscountedBag 클래스

Bag	DiscountedBag
Bag b1 = new Bag();	DiscountedBag b3 = new DiscountedBag();
Bag b2 = new Bag();	DiscountedBag b4 = new DiscountedBag();
b1.setPrice(50000);	b3.setPrice(50000);
System.out.println(b1.getPrice());	System.out.println(b3.getPrice());
b2.setPrice(b1.getPrice());	b4.setPrice(b3.getPrice());
System.out.println(b2.getPrice());	System.out.println(b4.getPrice());

행위일관성 X

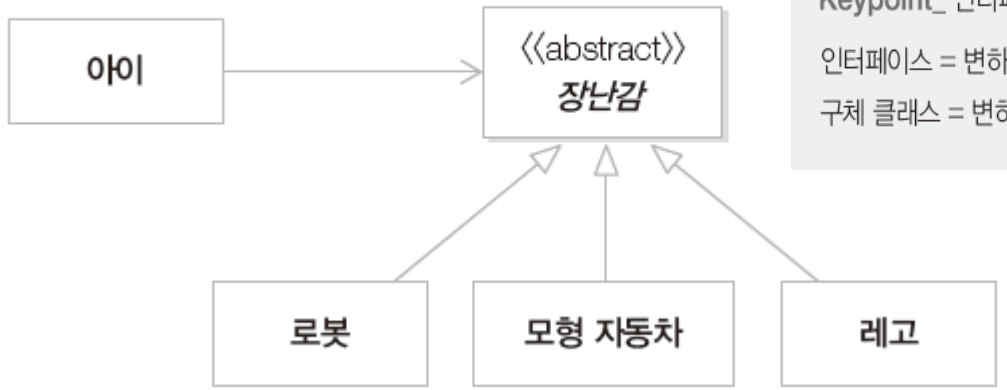
코드 3-4

```
public class DiscountedBag extends Bag {
  private double discountedRate;

  public void setDiscounted(double discountedRate) {
    this.discountedRate = discountedRate;
  }

  public void setPrice(int price) {
    super.setPrice(price - (int)(discountedRate * price));
  }
}
```

DIP

- DIP는 **의존 관계를 맺을 때 변화하기 쉬운 것 또는 자주 변화하는 것보다는 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



Keypoint_ 인터페이스나 추상 클래스와 의존 관계를 맺도록 설계해야 한다.

인터페이스 = 변하지 않는 것

구체 클래스 = 변하기 쉬운 것

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();
  }
}
```

ISP

❖ 인터페이스 분리 원칙

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

그림 3-11 복합기의 클래스 다이어그램

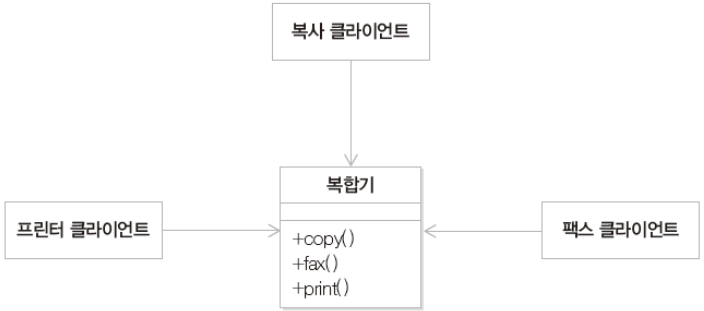


그림 3-12 복합기 클래스에 ISP를 적용한 예