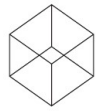
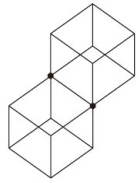
3. SOLID



JAVA 객체 지향 디자인 패턴

UML과 GoF 디자인 패턴 핵심 10가지로 배우는



- SOLID란 로버트 마틴[1][2]이 2000년대 초반[3]에 명명한 객체 지향 프로그래밍 및 설계의 다섯 가지 기본 원칙을 마이클 페더스가 두문자어 기억술로 소개한 것이다. 프로그래머가 시간이 지나도 유지 보수와 확장이 쉬운 시스템을 만들고자 할 때 이 원칙들을 함께 적용할 수 있다

두문자	약어	개념
S	SRP	**단일 책임 원칙 (Single responsibility principle)** 한 클래스는 하나의 책임만 가져야 한다.
O	OCP	**개방-폐쇄 원칙 (Open/closed principle)** "소프트웨어 요소는 확장에는 열려 있으나 변경에는 닫혀 있어야 한다."
L	LSP	**리스코프 치환 원칙 (Liskov substitution principle)** "프로그램의 객체는 프로그램의 정확성을 깨뜨리지 않으면서 하위 타입의 인스턴스로 바꿀 수 있어야 한다." 계약에 의한 설계를 참고하라.
I	ISP	**인터페이스 분리 원칙 (Interface segregation principle)** "특정 클라이언트를 위한 인터페이스 여러 개가 범용 인터페이스 하나보다 낫다."[4]
D	DIP	**의존관계 역전 원칙 (Dependency inversion principle)** 프로그래머는 "추상화에 의존해야지, 구체화에 의존하면 안된다."[4] 의존성 주입은 이 원칙을 따르는 방법 중 하나다.

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 (단위는 **객체**)

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
수강과목 조회와 추가(ㅇ)
데이타베이스에 학생정보 저장(X)
성적표와 출석부 출력 (X)

3.1.2 변경 (p. 105)

❖ 좋은 설계

- 설계원칙을 학습하는 이유는 예측하지 못한 변경사항이 발생하더라도, 유연하고 확장성 있도록 시스템 구조를 설계하기 위해서
- 좋은 설계란… 새로운 요구사항이나 변경사항이 있을때 가능한한 영향 받는 부분을 줄여야 함
- 클래스가 잘 설계되었는지를 판단하려면.. 언제 변경되어야 하는지를 사전에 검토 필요

❖ Keypoint : 책임 = 변경이유

- 책임은 변경이유
- 책임이 많다는 것은 변경될 여지가 많다는 의미
- 책임을 많이 질수록 클래스 내부에서 서로 다른 역할을 수행하는 코드끼리 강하게 결합될 가능성이 높아진다.

- Student Class의 변경 이유를 미리 생각해본다면 ??
  - 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?

3.1.3 책임 분리 (p. 106)

- 회귀(Regression) 테스트
  - 어떤 변화가 있을때 해당 변화가 기존 시스템의 기능에 영향을 주는지 평가하는 테스트
  - 회귀 테스트 비용을 줄이는 방법 하나는 시스템에 변경 사항이 발생했을때 영향을 받는 부분을 적게 하는 것
  - 한 클래스에 너무 많은 책임을 부여하지 말고, 단 하나의 책임만 수행하도록 함
    → 변경 사유가 될 수 있는 것을 하나로 만들어야 함 --> 책임 분리

그림 3-1 변경의 영향

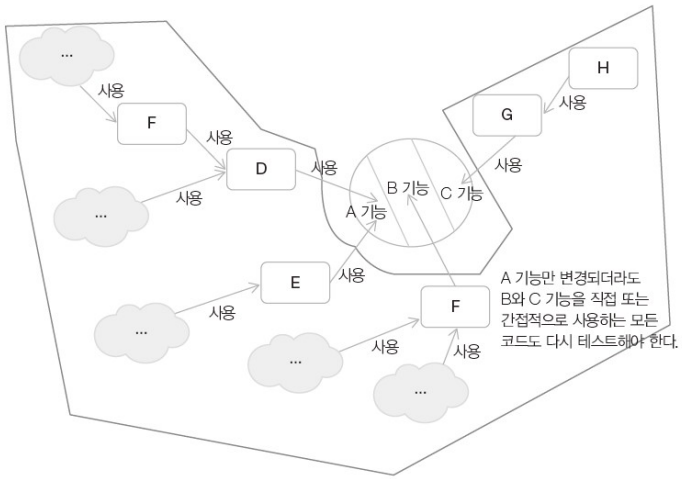
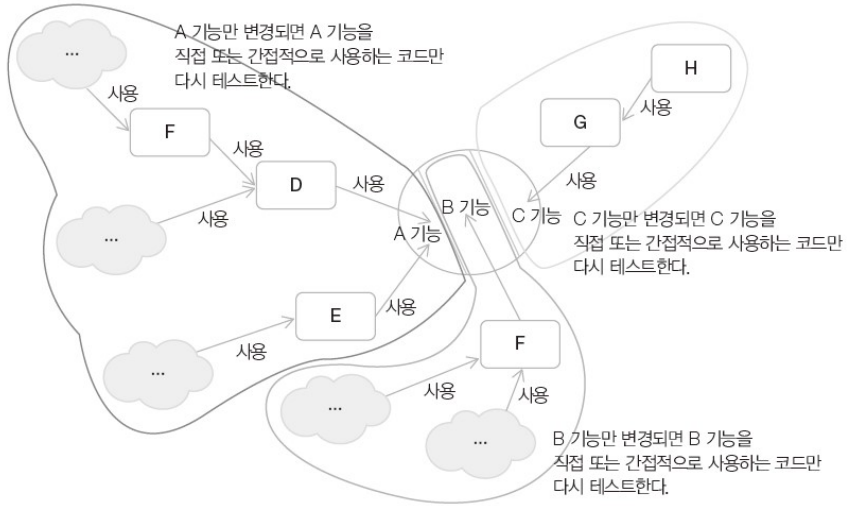


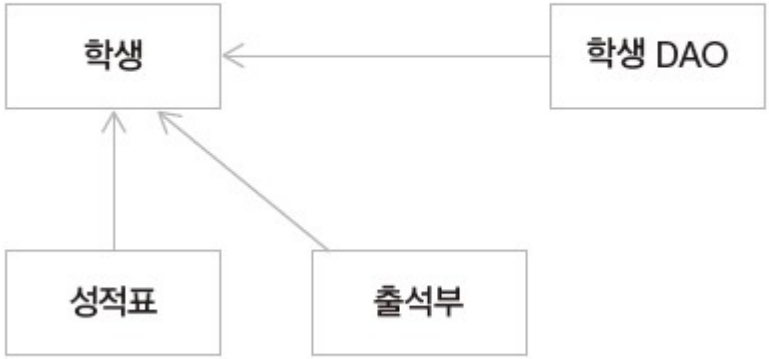
그림 3-2 책임 분리



- Student Class의 변경 사유 3가지
  - 학생의 고유 정보,
  - 데이터베이스 스키마
  - 출력 형식의 변화

- 학생 DAO : 학생 클래스의 인스턴스를 데이터베이스에 저장하거나 읽어들이는 역할
  - 데이타베이스 스키마가 변화되면
  - 학생 DAO 클래스나
  - 이를 사용하는 클래스만 영향.

그림 3-3 개선된 디자인

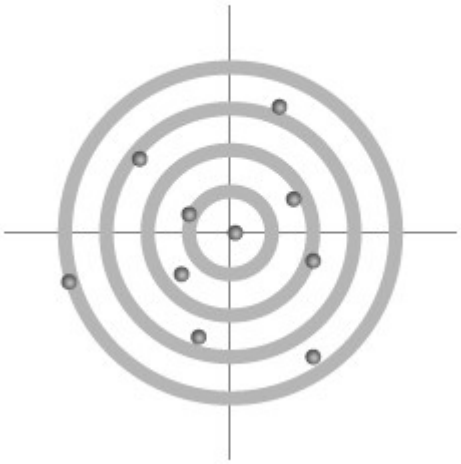


3.1.4 산탄총 수술(shotgun surgery) (p. 108)

❖ 하나의 책임이 여러 곳에 분산

- 변경 이유가 발생했을 때 변경할 곳이 많음
- 변경될 곳을 빠짐 없이 찾아 일관되게 변경해야 함
- 산탄총을 맞은 동물을 치료하는 상황

그림 3-4 산탄총 수술



❖ 횡단 관심 (cross-cutting concern)

- 하나의 책임이 여러 개의 클래스로 분리되어 있는 예
- 로깅, 보안, 트랜잭션

그림 3-5 횡단 관심

```
public void helloA() {
        System.out.println("Start......");
        AAA();
        System.out.println("End......");
}
```

```
public void helloB() {
        System.out.println("Start......");
        BBB();
        System.out.println("End......");
}
```


3.1.5 관심지향 프로그래밍과 횡단관심문제

- 관심지향 프로그래밍(AOP, Aspect-oriented Programming)
  - 횡단관심문제를 해결하는 방법
  - 횡단관심을 수행하는 코드를 애스팩트(aspect)라는 특별한 객체로 모듈화
  - 위빙(weaving)이라는 작업을 통해 모듈화한 코드를 핵심 기능에 끼워 넣음
  - 횡단 관심에 변경이 생긴다면 해당 애스펙트만 수정

3.2 개방-폐쇄 원칙(OCP-Open Closed Principle)

❖ 개방폐쇄원칙

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

그림 3-6 성적표나 출석부에 학생을 출력하는 기능을 사용

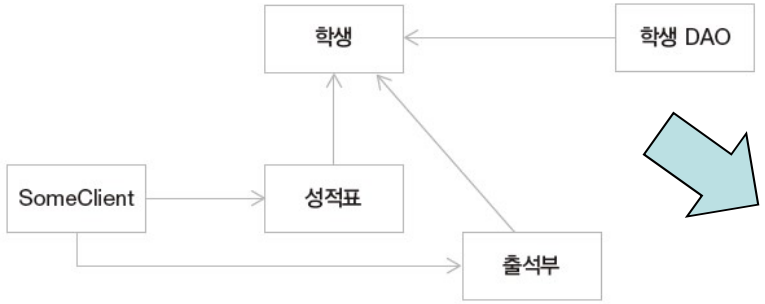
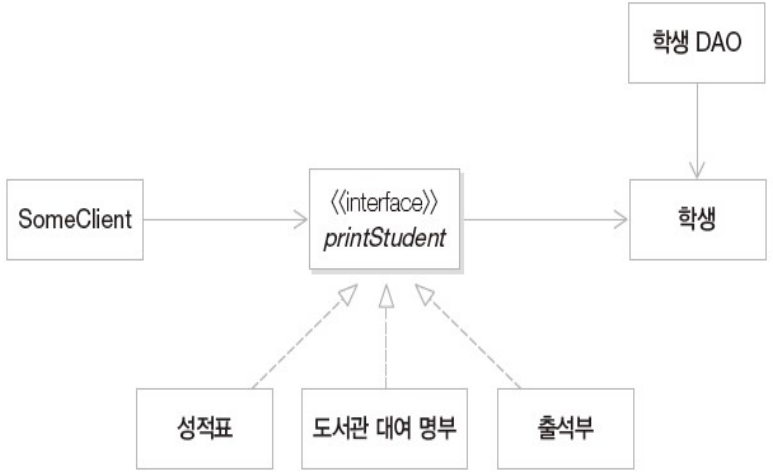


그림 3-7 OCP를 만족하는 설계

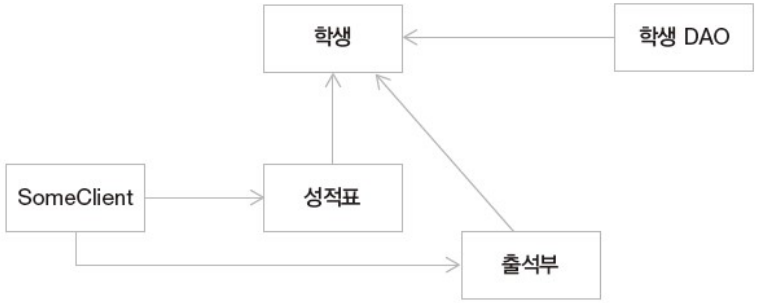


❖ 그림 3-6의 문제점

- **만약, 도서관 대여 명부와 같은 , 학생의 대여기록을 출력하기를 원할때**
- **방법.**
  - 도서관 대여명부 클래스를 만들고
  - SomeClient Class가 그 도서관대여명부클래스를 활용하도록

- **문제점**
  - SomeClient Class를 수정해야만 한다.
  - OCP를 위반

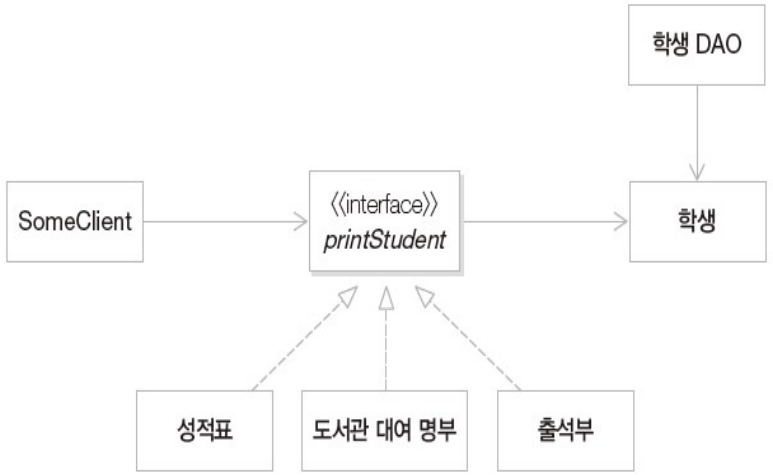
그림 3-6 성적표나 출석부에 학생을 출력하는 기능을 사용



❖ 그림 3-7의 해결책

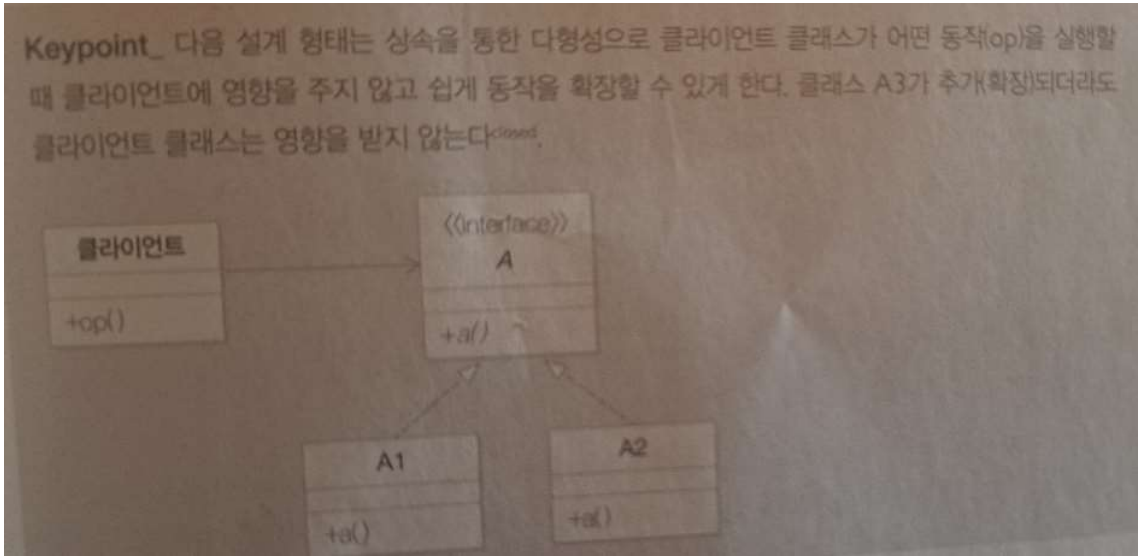
- 출력과 관련된 모든 클래스를 PrintStudent Interface의 하위에 위치
- 구체적인 다양한 출력 포맷은 interfac에 의하여 캡슐화
- SomeClient 를 수정할 필요가 없음

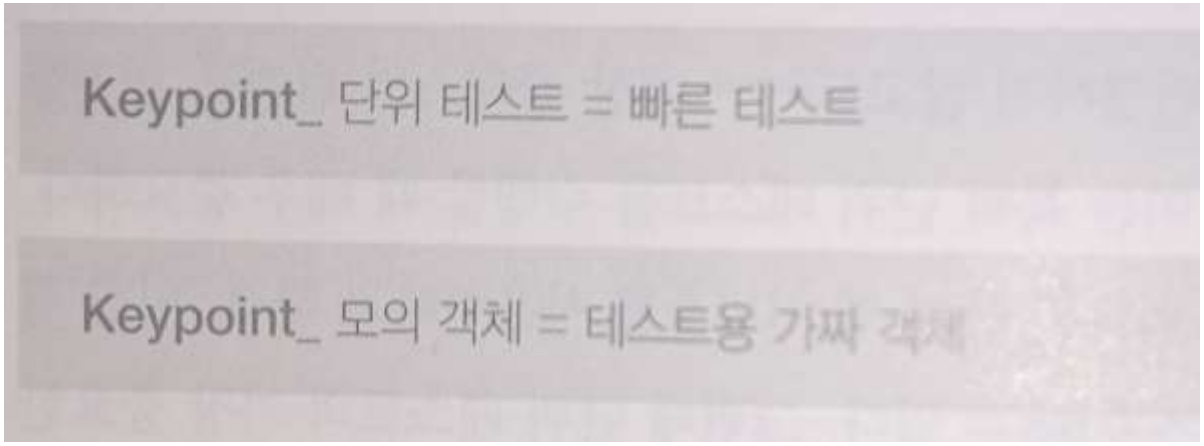
그림 3-7 OCP를 만족하는 설계



❖ OCP

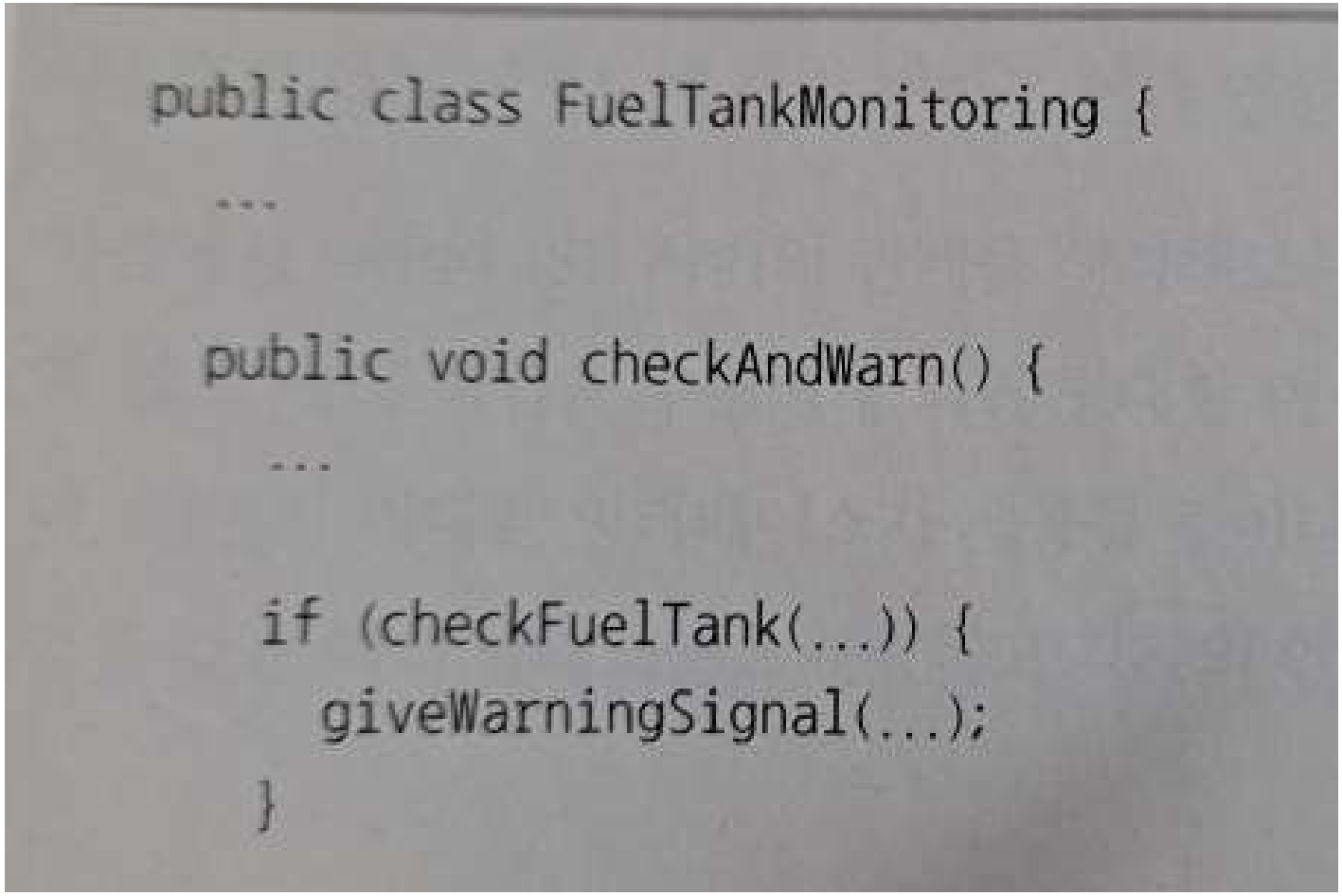
- **클래스를 변경하지 않더라도 (Closed)**
- **대상 클래스의 환경을 변경할수 있는 (open) 설계가 되어야 한다.**
- **➔ 단위테스트시 적용 원리**
  - 테스트 대상 기능이 사용하는 실제 외부의 서비스를 흉내내는 가짜 객체를 만들어 테스트의 효율성을 높일 필요
  - 예) 동시에 착륙하려는 비행기 1000대의 경우를 테스트할때 모의 객체를 이용



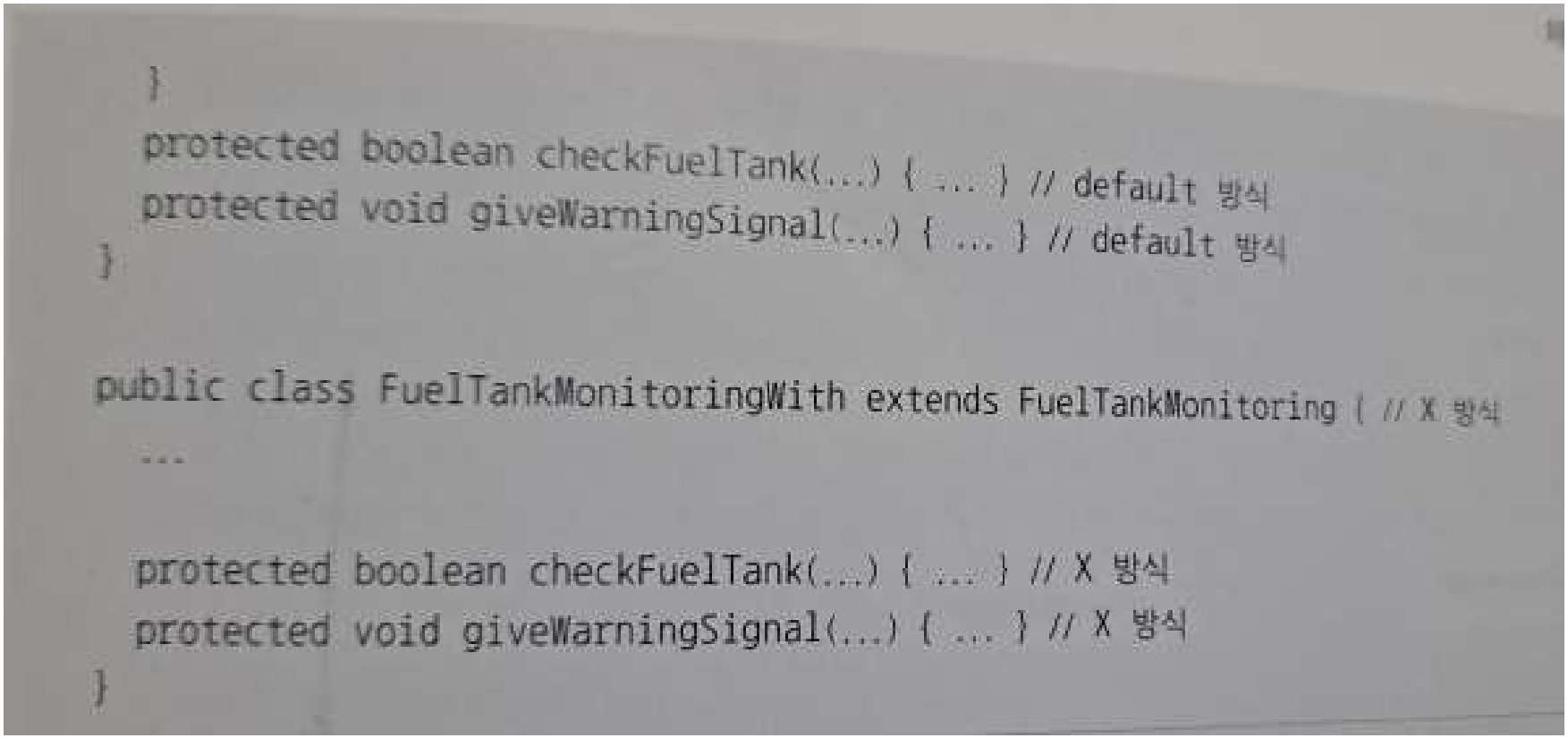

Keypoint_ 단위 테스트 = 빠른 테스트
Keypoint_ 모의 객체 = 테스트용 가짜 객체

체크포인트_ 다음 FuelTankMonitoring 클래스는 로켓의 연료 탱크를 검사해 특정 조건에 맞지 않으면 관리자에게 경고 신호를 보내주는 기능이 있다. 연료 탱크를 검사하는 방식과 경고를 보내는 방식이 변경될 가능성이 큰 경우에 대비해 다음 코드를 수정하라.

```
public class FuelTankMonitoring {
  ...

  public void checkAndWarn() {
    ...

    if (checkFuelTank(...)) {
      giveWarningSignal(...);
    }
  }
  private boolean checkFuelTank(...) { ... }
  private void giveWarningSignal(...) { ... }
}
```



```
public class FuelTankMonitoring {
  ...

  public void checkAndWarn() {
    ...

    if (checkFuelTank(...)) {
      giveWarningSignal(...);
    }
```



```
    }
    protected boolean checkFuelTank(...) { ... } // default 방식
    protected void giveWarningSignal(...) { ... } // default 방식
}

public class FuelTankMonitoringWith extends FuelTankMonitoring { // X 방식
    ...

    protected boolean checkFuelTank(...) { ... } // X 방식
    protected void giveWarningSignal(...) { ... } // X 방식
}
```

체크포인트(p. 115)

❖ 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();
    }
  }
}
```


3.3 리스코프 치환 원칙
LSP

❖ 리스코프 치환 원칙

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

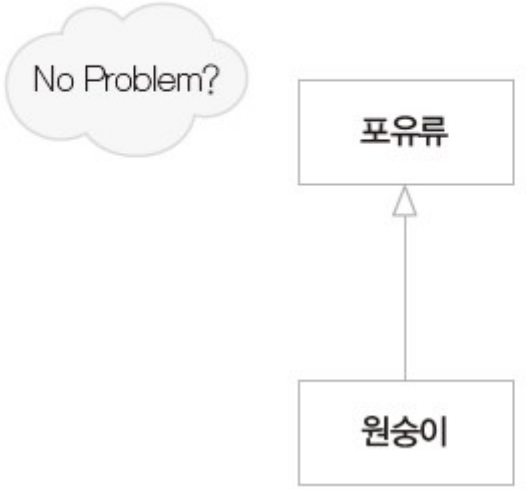
"A type hierarchy is composed of subtypes and supertypes. The intuitive idea of a subtytpe is one whose objects provide all the behavior of another type(the supertype) plus something extra. What is wanted here is something like the following substitution property: if for each object o1 of type S there 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."

- 자식클래스는 최소한 부모 클래스에서 가능한 행위는 수행할수 있어야 한다.

LSP

- LSP를 만족하면 프로그램에서 부모 클래스의 인스턴스 대신에 자식 클래스의 인스턴스로 대체해도 프로그램의 의미는 변화되지 않는다.
- 일반화 관계 : is a kind of 관계

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

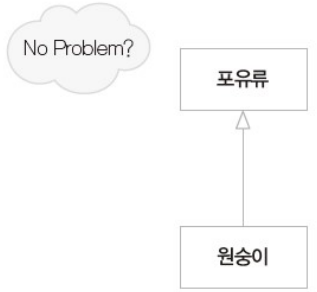


그림 3-9 오리너구리

오리너구리와 포유류의 관계

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.(X)
- 오리너구리는 젖을 먹여서 새끼를 키우고 폐를 통해 호흡한다.
- 오리너구리는 체온이 일정한 정온 동물이며 털이나 두꺼운 피부로 덮여 있다.

- 만약 오리너구리가 포유류로 분류된다면 상기 포유류의 정의가 잘못된 것

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

- 아래 양쪽의 답이 같다.

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

❖ 표 3-4

- 아래 Child class에서 setPrice를 재정의 (Override)
- 이는 LSP를 만족하지 않음

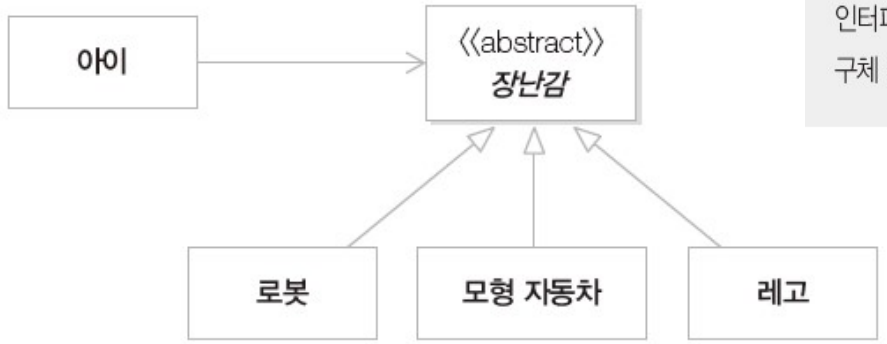
코드 3-4

```
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));
  }
}
```


3.4 의존역전원칙(DIP, Dependency Inversion Principle, p. 121)

- DIP는 의존 관계를 맺을 때 변화하기 쉬운 것 또는 자주 변화하는 것보다는 변화하기 어려운 것, 거의 변화가 없는 것에 의존하라는 원칙
  - 변하기 어려운 것 : 정책, 전략과 같은 어떤 큰 흐름이나 개념 같은 추상적인 것
  - 변하기 쉬운 것 : 구체적인 방식, 사물 등과 같은 것
- 그림 3-10
  - 아이가 장난감을 갖고 논다 (변하지 않음)
  - 어떤 때는 로봇을 갖고 놀고 어떤 때는 자동차를 갖고 논다 (변하기 쉬움)

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(Dependency injection)

❖ 의존성주입

- 클래스 외부에서 의존되는 것을 대상 객체의 인스턴스 변수에 주입하는 기술
- Dip를 만족하면 의존성주입이라는 기술로 변화를 쉽게 수용할수 있는 코드를 작성할수 있음

- 코드 3-5. Kid 클래스를 변경하지 않고도, 아이가 노는 장난감을 바꿔줄수 있음

코드 3-5

```
public class Kid {
  private Toy toy;

  public void setToy(Toy toy) {
    this.toy = toy;
  }

  public void play() {
    System.out.println(toy.toString());
  }
}
```

❖ 일단, 아이가 로봇을 갖고 놀도록

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();
  }
}
```

아이가 레고를 갖고 놀도록 변경

코드 3-8

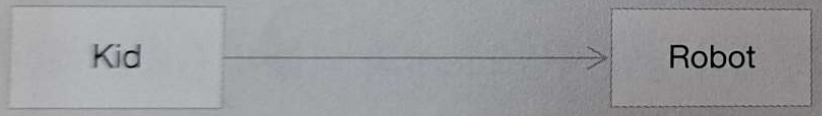
```
public class Lego extends Toy {
  public String toString() {
    return "Lego";
  }
}
```

코드 3-9

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

❖ 체크 포인트 ➔ p. 132

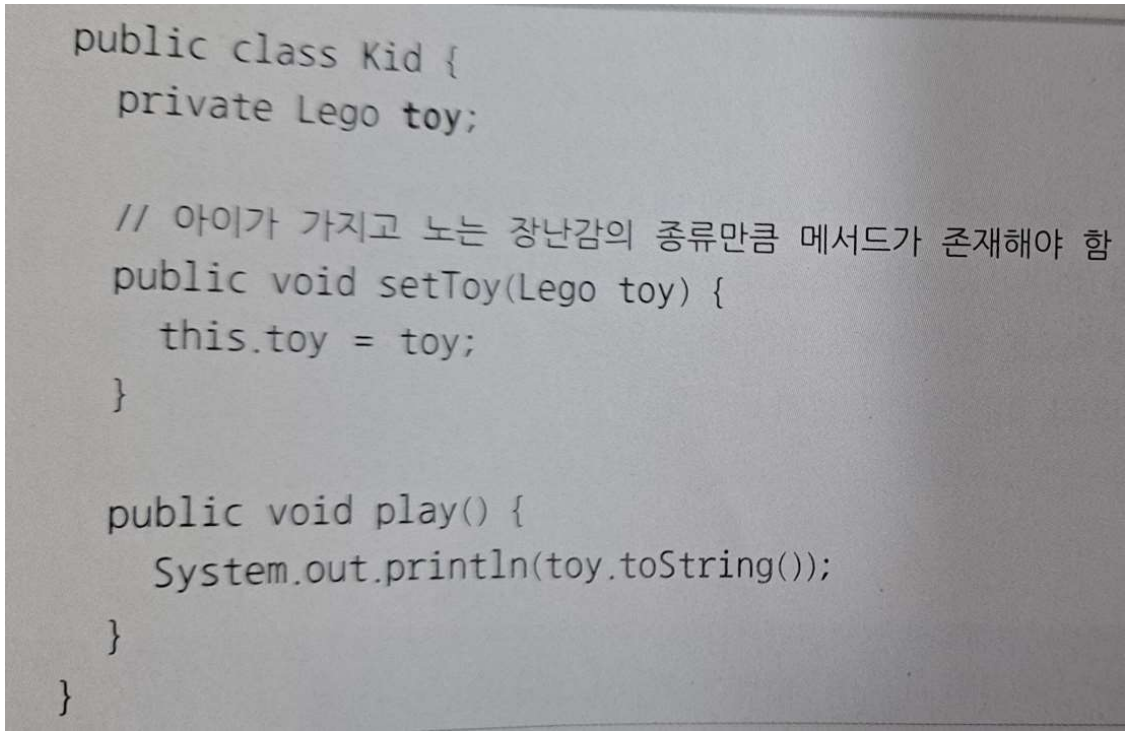
체크포인트_ 만약 Kid 클래스가 다음 클래스 다이어그램처럼 Robot 클래스와 연관 관계를 맺는다면 어떤 일이 발생할까?




```
public class Kid {
  private Robot toy;

  public void setToy(Robot toy) {
    this.toy = toy;
  }

  public void play() {
    System.out.println(toy.toString());
  }
}

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

- **이 코드에서 만약 레고로 바뀌면 ??**

❖ 레고를 갖고 놀게 하기 위해서 바뀐 코드
- OCP 위배
- 코드 3-5와 비교하라



```
public class Kid {
  private Lego toy;

  // 아이가 가지고 노는 장난감의 종류만큼 메서드가 존재해야 함
  public void setToy(Lego toy) {
    this.toy = toy;
  }

  public void play() {
    System.out.println(toy.toString());
  }
}
```

코드 3-5

```
public class Kid {
  private Toy toy;

  public void setToy(Toy toy) {
    this.toy = toy;
  }

  public void play() {
    System.out.println(toy.toString());
  }
}
```

3.5 인터페이스 분리 원칙 (ISP, Interface Segerigation Principle)

❖ 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

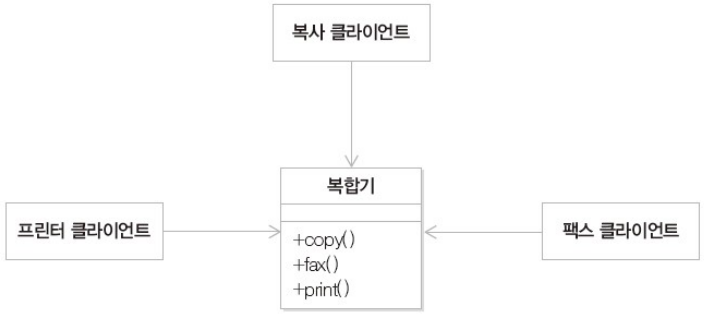
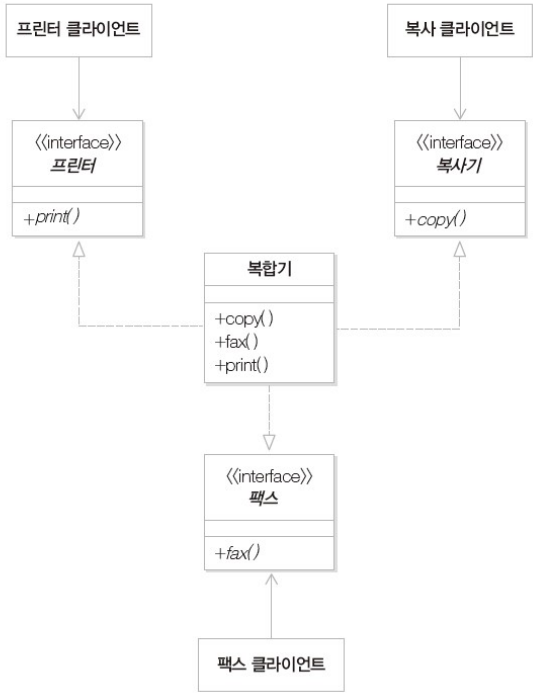


그림 3-12 복합기 클래스에 ISP를 적용한 예



SRP와 ISP

Keypoint_ SRP를 만족하더라도 ISP를 반드시 만족한다고는 할 수 없다.

그림 3-12 복합기 클래스에 ISP를 적용한 예