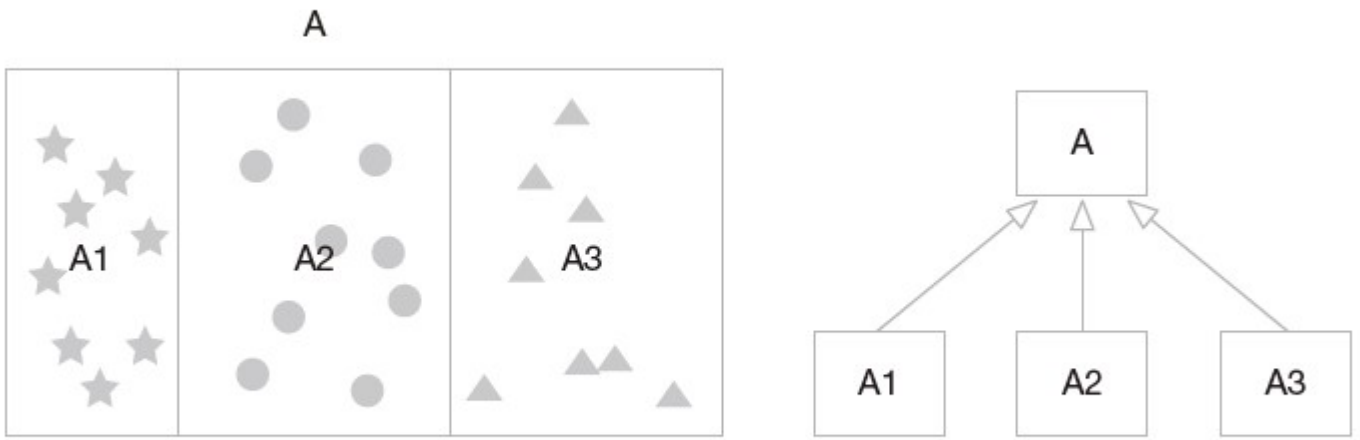
2.3.3 집합 관점으로 본 일반화 관계

❖ 그림 2-8

그림 2-8 집합과 일반화 관계

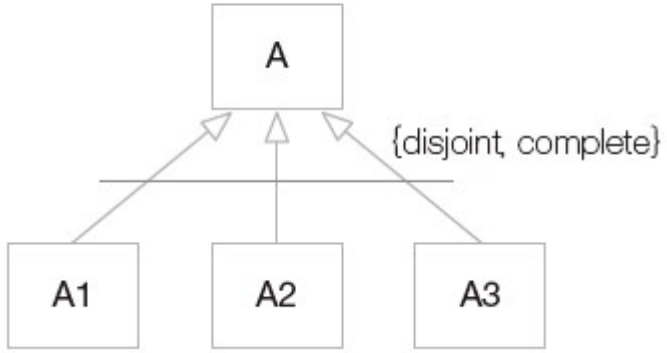


- $A = A1 \cup A2 \cup A3$
- $A1 \cap A2 \cap A3 = \varnothing$

집합과 일반화

❖ 그림 2-9

그림 2-9 일반화 관계에서의 제약 조건



❖ {disjoint}

- 자식 클래스 객체가 동시에 두 클래스에 속할 수 없다

❖ {complete}

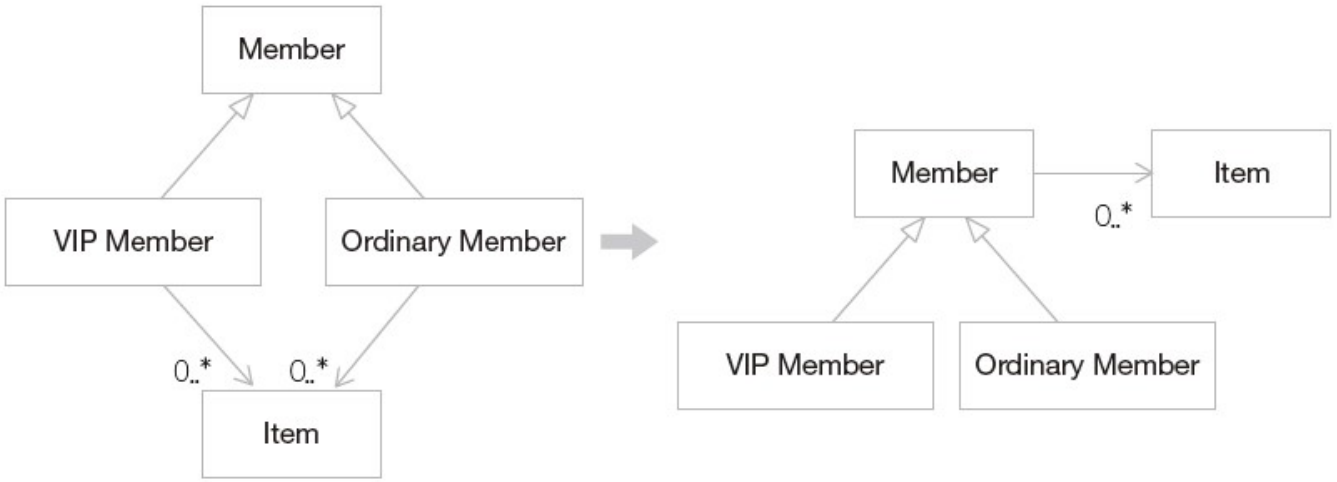
- 자식 클래스의 객체에 해당하는 부모 클래스의 객체와 부모 클래스의 객체에 해당하는 자식 클래스의 객체가 하나만 존재한다는 의미

연관 관계의일반화

- 일반화는 연관 관계를 단순하게 만들 수 있다

- 그림 2-10

그림 2-10 집합론을 통한 연관 관계의 일반화

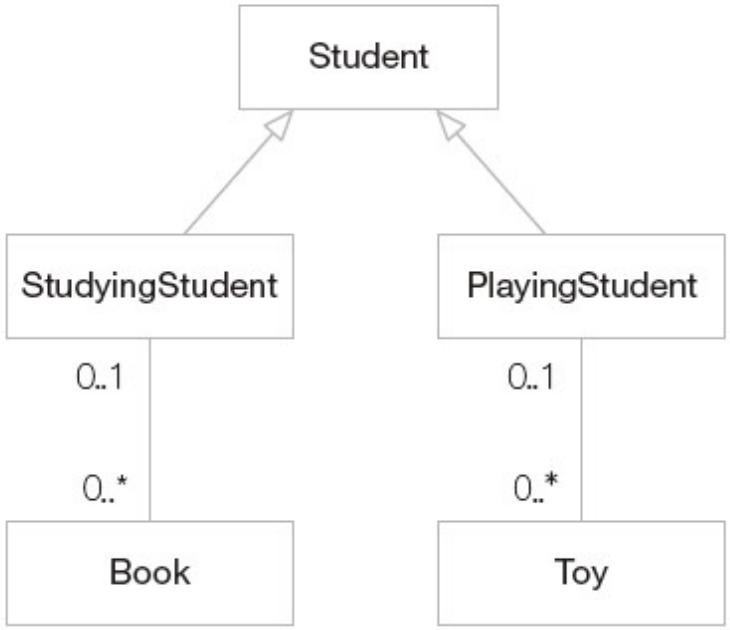


상호배타적 상태

❖ 집합론 관점에서 **일반화**는 **상호 배타적인 부분 집합으로 나누는 과정**

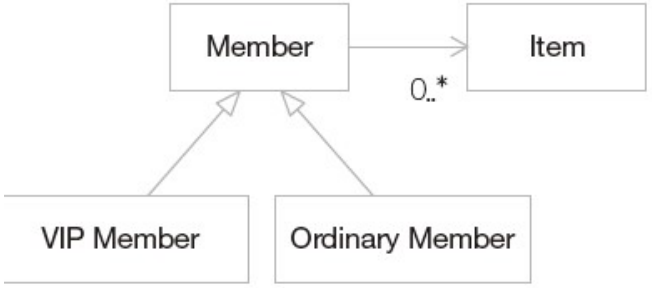
- 예) 공부하고 있는 중에는 책만 볼 수 있다. 장난감을 가지고 공부 할 수 없다
- 예) 노는 중에는 장난감만 갖고 놀 수 있다. 책을 가지고 놀지 않는다

그림 2-11 일반화 관계를 이용한 상호 배타적 관계 모델링

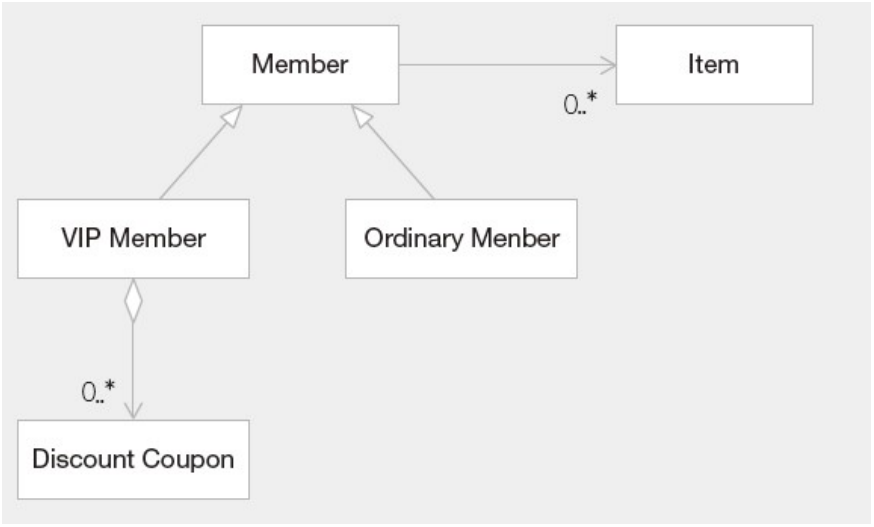


특수화(specialization)

- 그림 2-10에서는 Vip Member와 Ordinary Member로 구별하는 이유가 없음



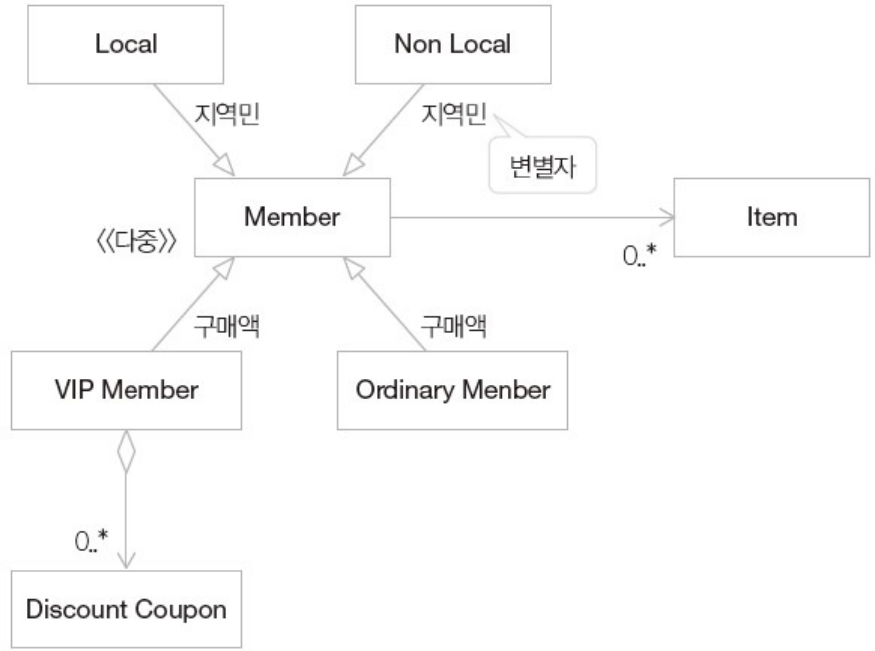
- VIP 멤버에게만 할인 쿠폰이 발행됨



변별자와 다중 분류

- 변별자(discriminator): 인스턴스 분류 기준
  - 지역민, 구매액
- 다중분류(multiple classification): 한 인스턴스가 동시에 여러 클래스에 속함

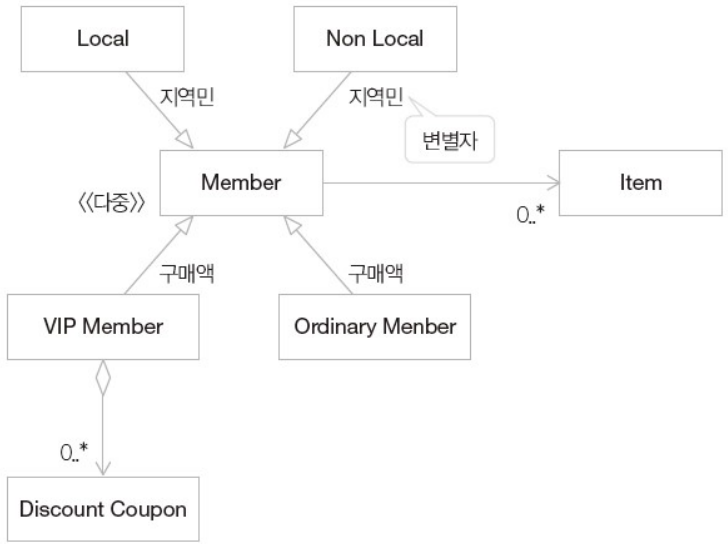
그림 2-12 변별자와 다중 분류



변별자와 다중 분류

- 지역주민이든 아니든 VIP 멤버에게는 할인 쿠폰이 지급된다
- How? 일반 회원이지만 지역 주민에게는 경품을 제공하도록 시스템에 새로운 요구사항을 추가

그림 2-12 변별자와 다중 분류



2.4 다형성

❖ 다형성

- 서로 다른 클래스의 객체가 같은 메시지를 받았을 때 각자의 방식으로 동작하는 능력'
- 자식 클래스를 개별적으로 다룰 필요가 없이 한번에 처리할수 있는 수단 제공
- 같은 애완동물이 "talk" 하지만 고양이는 야옹, 강아지는 멍멍, 앵무새는 안녕..
-

❖ 코드 2-15, 2-16, 2-17

❖ 코드 2-15 . 다형성을 적용한 코드

코드 2-15

```
public abstract class Pet {
  public abstract void talk();
}

public class Cat extends Pet {
  public void talk() {
    System.out.println("야옹");
  }
}

public class Dog extends Pet {
  public void talk() {
    System.out.println("멍멍");
```

```
  }
}
public class Parrot extends Pet {
  public void talk() {
    System.out.println("안녕엉");
  }
}
```

❖ 그림 2-16

- 다형성을 적용하지 않은 코드

코드 2-16

```
class Dog {
  public void bark() { ... }
}

class Cat {
  public void meow() { ... }
}

class Parrot {
  public void sing() { ... }
}

public class Main {
  public static void main(String[] args) {
    Dog d = new Dog();
    Cat c = new Cat();
    Parrot p = new Parrot();

    d.bark();
    c.meow();
    p.sing();
  }
}
```

❖ 그림 2-17

코드 2-17

```
abstrat class Pet {
  public abstract void talk();
}

class Dog extends Pet {
  public void talk() { ... }
}

class Cat extends Pet {
  public void talk() { ... }
}

class Parrot extends Pet {
  public void talk() { ... }
}

public class Main {
  public static void groupTalk(Pet[] p) {
    int i;

    for (i = 0; i < 3; i++)
      p.talk();
  }

  public static void main(String[] args) {
    Pet[] p = {new Cat(), new Dog(), new Parrot()};
    groupTalk(p);
  }
}
```

2.5 피터 코드의 상속 규칙

- 피터 코드의 상속 규칙 (아래 하나라도 만족하지 않는다면 상속을 사용해서는 안된다)
  - 자식 클래스와 부모 클래스 사이는 '역할 수행is role played by' 관계가 아니어야 한다.
  - 한 클래스의 인스턴스는 다른 서브 클래스의 객체로 변환할 필요가 절대 없어야 한다.
  - 자식 클래스가 부모 클래스의 책임을 무시하거나 재정의하지 않고 확장만 수행해야 한다.
  - 자식 클래스가 단지 일부 기능을 재사용할 목적으로 유틸리티 역할을 수행하는 클래스를 상속하지 않아야 한다.
  - 자식 클래스가 '역할role', '트랜잭션transaction', '디바이스device' 등을 특수화specialization해야 한다.

상속 규칙

그림 2-14 상속으로 표현한 역할 수행 관계

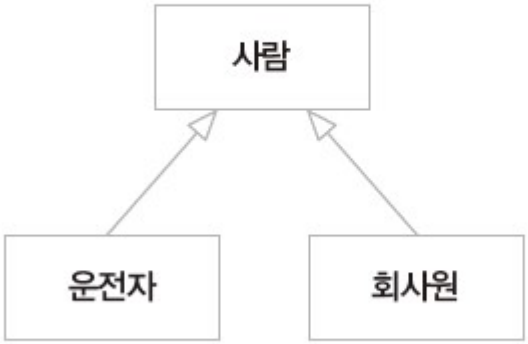
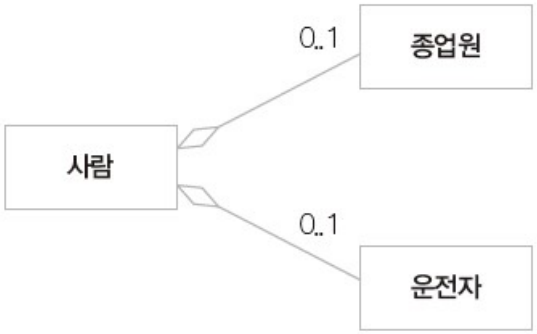
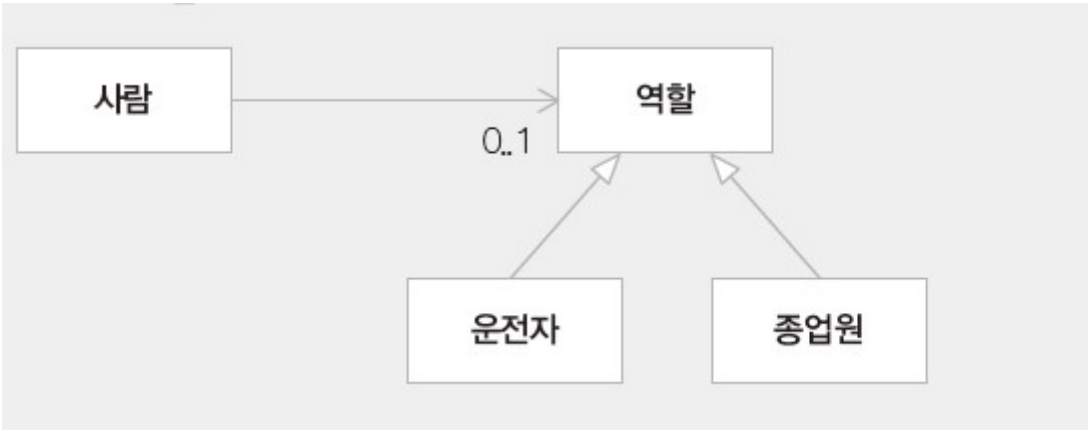


그림 2-15 집약 관계를 이용한 역할 수행 표현





첫 번째 규칙인 자식 클래스가 부모 클래스의 역할 중 하나를 표현하는지를 점검해보자. '운전자'는 어떤 순간에 '사람'이 수행하는 역할의 하나다. 마찬가지로 '회사원'도 사람이 어떤 순간에 수행하는 역할의 하나다. 따라서 사람과 운전자나 사람과 회사원은 상속 관계로 표현되어서는 안 되므로 규칙에 위배된다.

두 번째 규칙인 자식 클래스의 인스턴스들 사이에 변환 관계가 필요한지를 점검해보자. '운전자'는 어떤 시점에서 '회사원'이 될 필요가 있으며 '회사원' 역시 '운전자'가 될 필요가 있다. 가령 자신이 일하는 회사로 출퇴근하는 동안에는 '운전자'로서의 역할을 수행하며, 회사에 있을 때는 '회사원'으로서의 역할을 수행한다. 이런 경우 객체의 변환 작업이 필요하므로 규칙에 위배된다. 가령 평생 운전자 역할만 하던가 회사원 역할만 수행한다면 그림 2-14와 같이 표현하는 것도 나쁘지 않다. 그러나 대부분의 사람은 한 역할에 고정되지 않고 시점에 따라 다른 역할을 수행하는 경우가 많다.

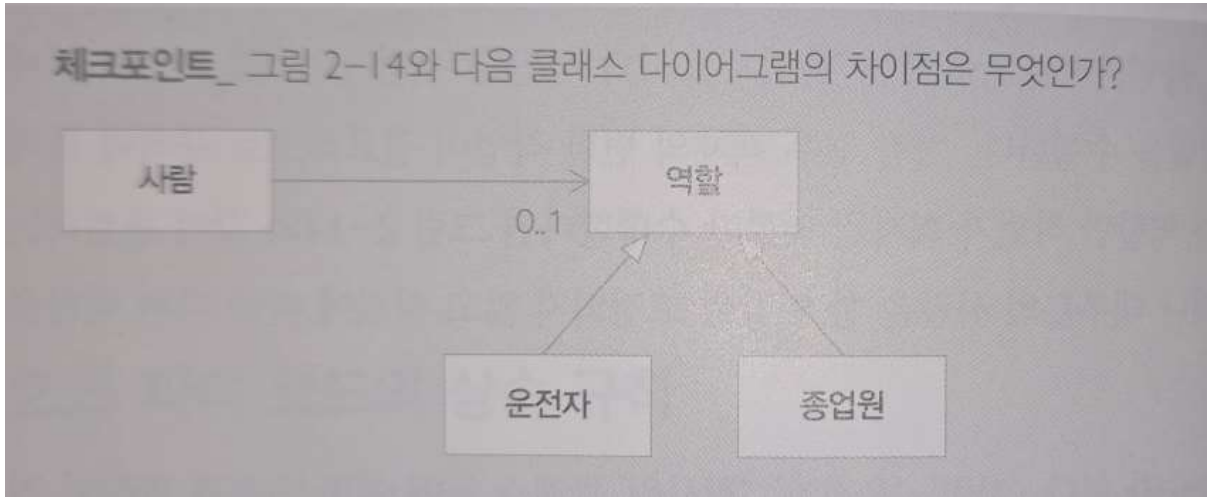
세 번째 규칙은 점검할 수가 없다. '사람', '운전자', '회사원' 클래스 등에 어떤 속성과 연산이 정의되었는지 정보가 없기 때문이다.

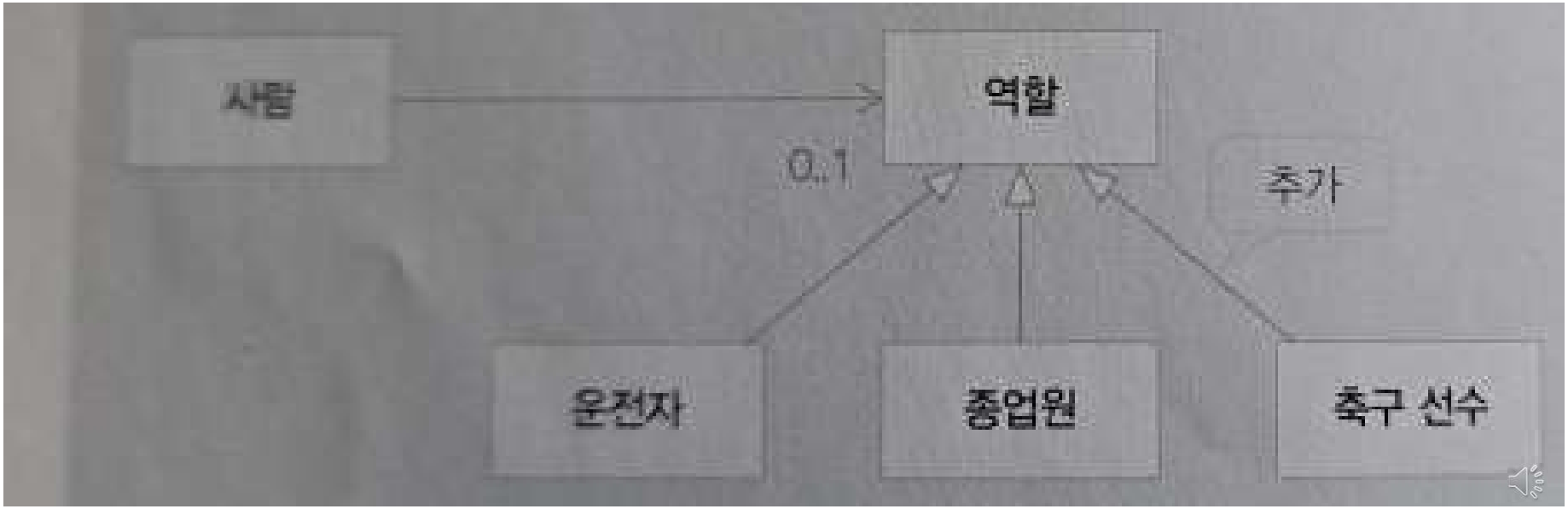
네 번째 규칙은 기능만 재사용할 목적으로 상속 관계를 표현하지는 않았으므로 규칙을 준수한다.

마지막 규칙은 슈퍼 클래스가 역할, 트랜잭션, 디바이스를 표현하지 않았으므로 규칙에 위배된다.

P. 88

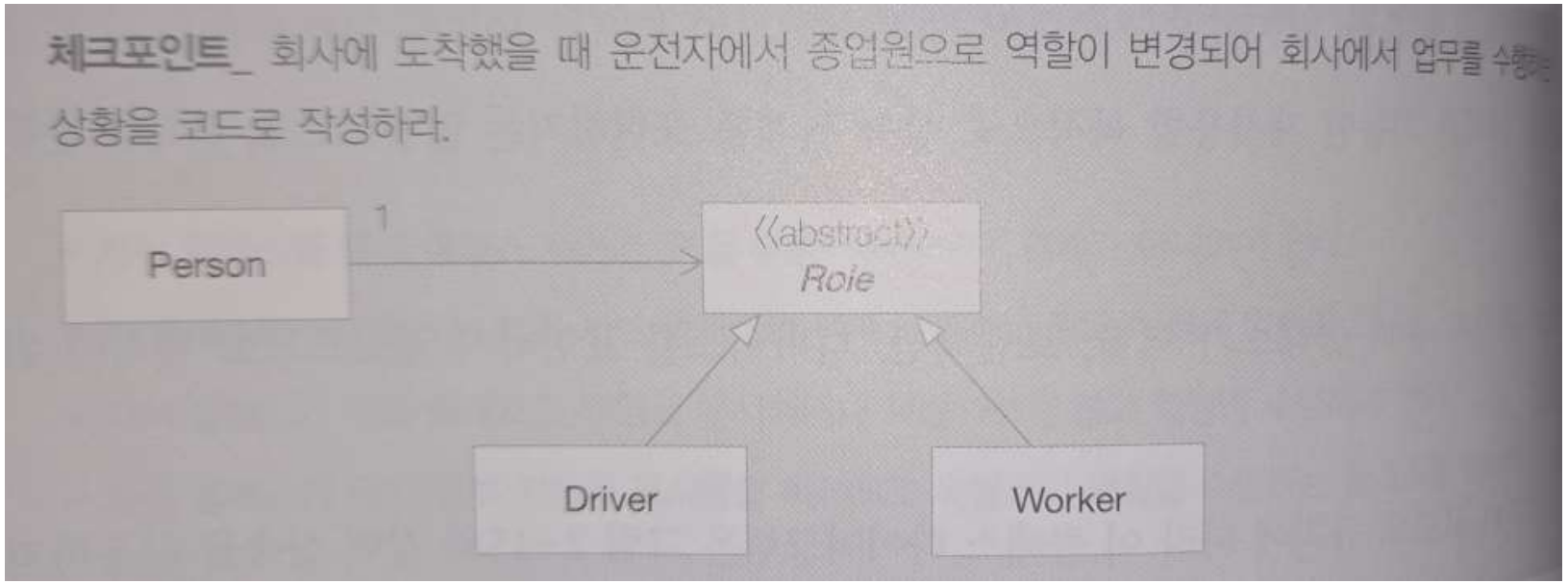
❖ ➔ p. 96

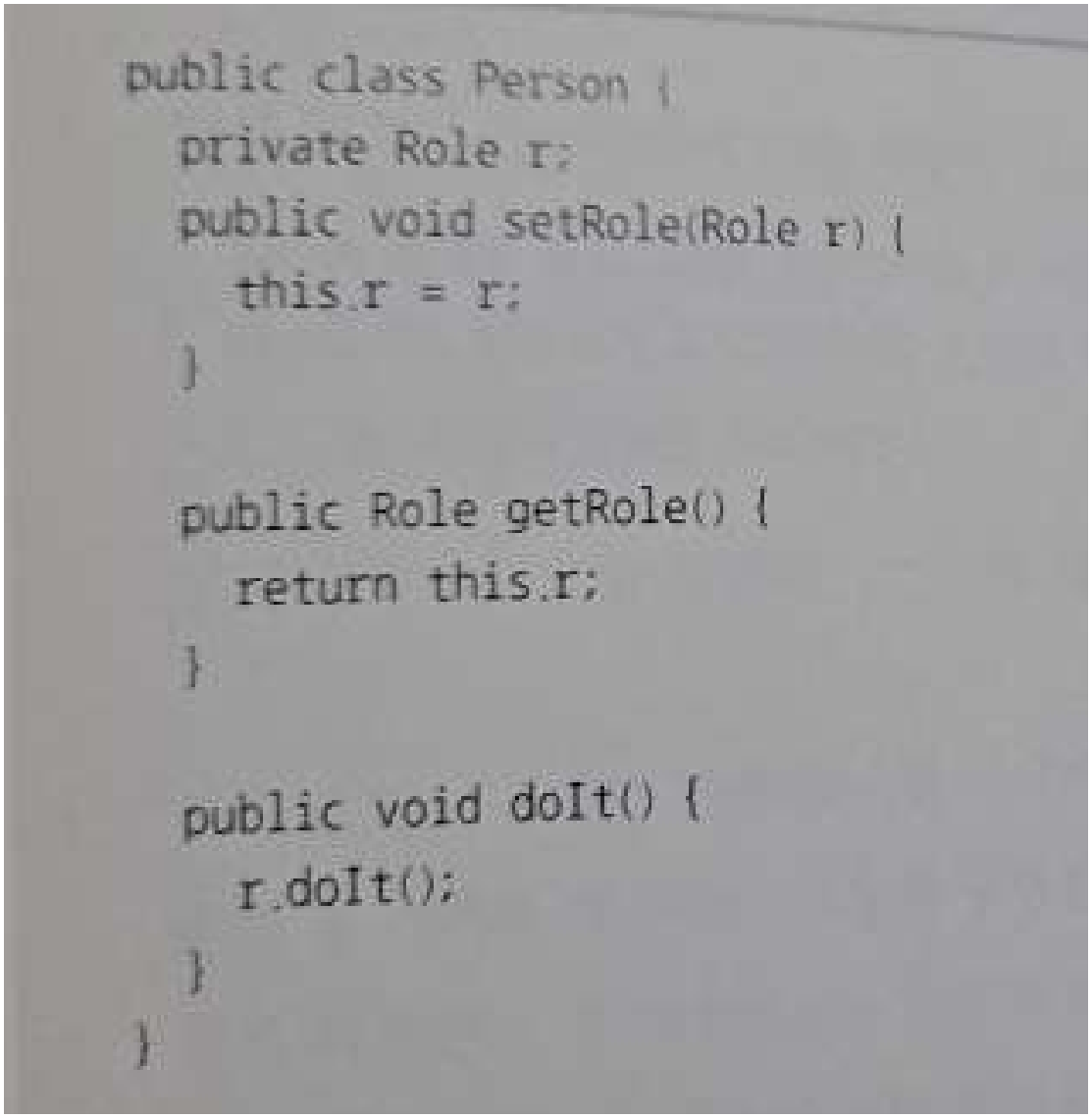

체크포인트_ 그림 2-14와 다음 클래스 다이어그램의 차이점은 무엇인가?
사람
역할
0..1
운전자
종업원


사람
역할
0..1
추가
운전자
종업원
축구 선수

P. 88

❖ ➔ p. 97

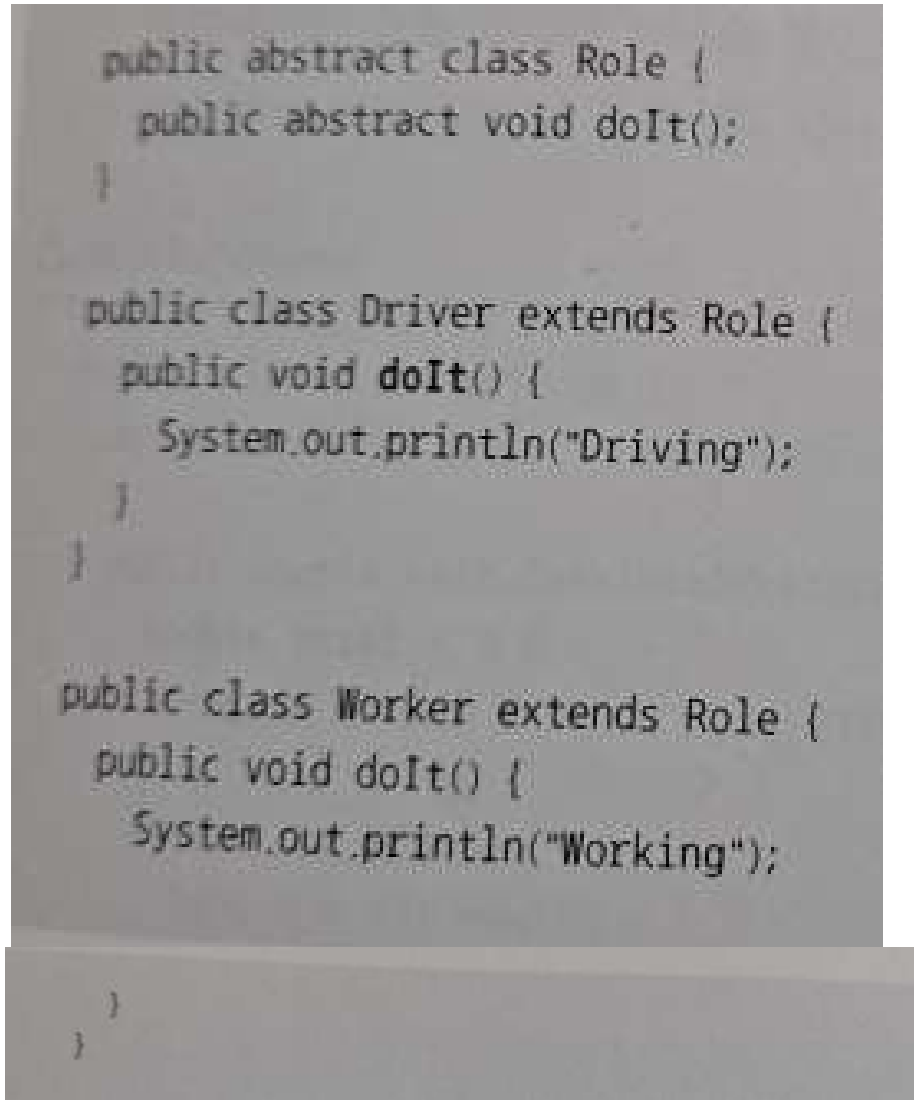

체크포인트_ 회사에 도착했을 때 운전자에서 종업원으로 역할이 변경되어 회사에서 업무를 수행
상황을 코드로 작성하라.
Person
1
〈〈abstract〉〉
Roie
Driver
Worker



```
public class Person {
  private Role r;
  public void setRole(Role r) {
    this.r = r;
  }

  public Role getRole() {
    return this.r;
  }

  public void doIt() {
    r.doIt();
  }
}
```

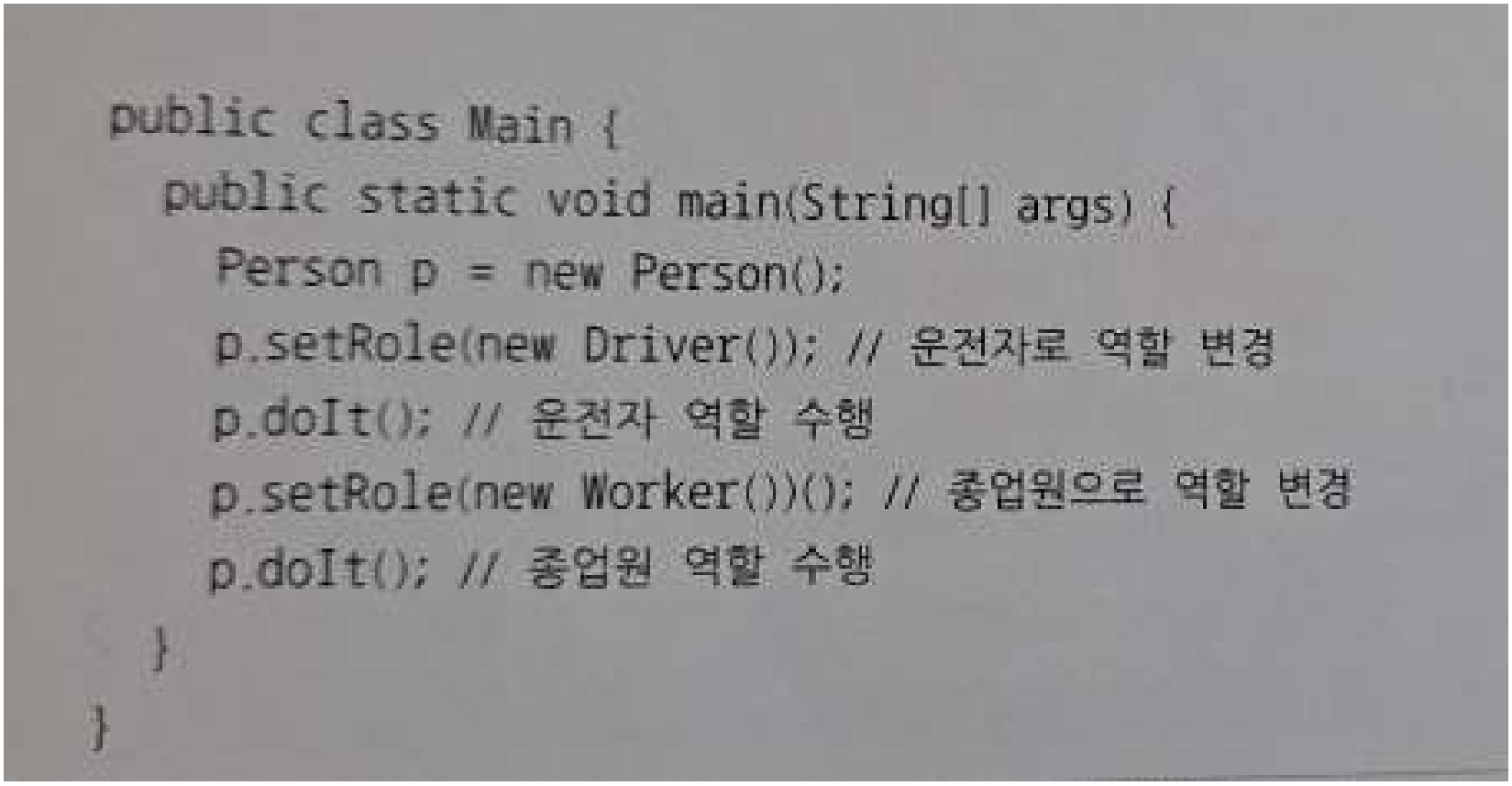


```
public abstract class Role {
  public abstract void doIt();
}

public class Driver extends Role {
  public void doIt() {
    System.out.println("Driving");
  }
}

public class Worker extends Role {
  public void doIt() {
    System.out.println("Working");

  }
}
```



```
public class Main {
    public static void main(String[] args) {
        Person p = new Person();
        p.setRole(new Driver()); // 운전자로 역할 변경
        p.doIt(); // 운전자 역할 수행
        p.setRole(new Worker())(); // 종업원으로 역할 변경
        p.doIt(); // 종업원 역할 수행
    }
}
```