서브클래싱과 서브타이핑

제대로 상속하기

전철호

한밭대학교 모바일융합공학과

목차

상속의 첫 번째 용도: 타입 계층 구현

- 타입 계층 안에서
  - 부모 클래스는 일반적인 개념을 구현
  - 자식 클래스는 특수한 개념을 구현
- 타입 계층의 관점에서
  - 부모 클래스는 자식 클래스의 일반화(generalization)
  - 자식 클래스는 부모 클래스의 특수화(specialization)

상속의 두 번째 용도: 코드 재사용

- 부모 클래스의 코드를 재사용할 수 있음
- 하지만 두 클래스 간의 강한 결합이 발생해 코드를 변경하기 어려움

타입 계층 구현을 위해 상속을 사용하라

- 재사용을 목표로 상속을 사용하면 부모, 자식 클래스를 강하게 결합시킴
- 타입 계층을 목표로 상속을 사용하면 확장 가능하고 유연한 설계를 얻을 수 있음
- 단, 다형적인 객체를 구현하기 위해서는 객체의 행동을 기반으로 타입 계층을 구성해야함

올바른 타입 계층을 구성하는 원칙을 살펴보자

1. 타입

타입

01. 개념 관점의 타입

- 개념 관점에서 타입이란 우리가 인지하는 세상의 사물의 종류를 의미
- 다시 말해, 우리가 인식하는 객체들에 적용하는 개념이나 아이디어를 가리켜 타입이라 지칭
- 자바, 루비, C를 프로그래밍 언어라는 타입으로 분류할 수 있음

타입

01. 개념 관점의 타입

- 어떤 대상이 타입으로 분류될 때 그 대상을 타입의 인스턴스라고 칭함
- 일반적으로 타입의 인스턴스를 객체라고 부름

타입의 구성 요소: 심볼, 내연, 외연

- 심볼(symbol)
  - 타입에 이름을 붙인 것
  - 앞에서 ‘프로그래밍 언어’가 타입의 심볼에 해당함

타입의 구성 요소: 심볼, 내연, 외연

- 내연(intension)
  - 타입에 속하는 객체들이 가지는 공통적인 속성이나 행동
  - 타입에 속하는 객체들이 공유하는 속성이나 행동의 집합이 내연을 구성함

타입의 구성 요소: 심볼, 내연, 외연

- 외연(extension)
  - 타입에 속하는 객체들의 집합
  - ‘프로그래밍 언어’ 타입의 경우에는 자바, 루비, C가 속한 집합이 외연을 구성함

타입의 구성 요소: 심볼, 내연, 외연

- 심볼: 타입의 이름
- 내연: 타입에 속하는 객체들의 공통적인 속성과 행동의 집합
- 외연: 타입에 속하는 객체들의 집합

타입

02. 프로그래밍 언어 관점의 타입

- 프로그래밍 언어 관점에서 타입은 비트에 의미를 부여하기 위해 정의된 '제약'과 '규칙'을 가리킴
- 비트에 담긴 데이터를 문자열로 다룰지, 정수로 다룰지는 데이터의 타입에 의해 결정됨

프로그래밍 언어에서 타입의 두 가지 목적

- 타입에 수행될 수 있는 유효한 오퍼레이션의 집합을 정의
  - '+' 연산자는 숫자 타입이나 문자열 타입의 객체에는 사용할 수 있지만 다른 클래스의 인스턴스에 대해서는 사용할 수 없음
- 타입에 수행되는 오퍼레이션에 대해 미리 약속된 문맥 제공
  - '+' 연산자는 연산의 대상이 숫자 타입이면 '덧셈', 문자열 타입이면 '연결'을 수행함

타입

03. 객체지향 패러다임 관점의 타입

객체지향 패러다임 관점에서

- 오퍼레이션은 객체가 수신할 수 있는 메시지를 의미함
- 따라서 타입을 정의하는 것은 객체의 '퍼블릭 인터페이스'를 정의하는 것과 동일함
- 동일한 인터페이스를 가지는 객체들은 동일한 타입으로 분류할 수 있음

타입

03. 객체지향 패러다임 관점의 타입

객체지향 패러다임 관점에서

- 타입의 정의는 객체에게 중요한 것은 속성이 아니라 행동이라는 사실을 다시 한번 강조함
- 어떤 객체들이 동일한 상태를 가지고 있더라도, 퍼블릭 인터페이스가 다르다면 서로 다른 타입으로 분류됨
- 객체의 타입을 결정하는 것은 내부 속성이 아니라 객체가 외부에 제공하는 행동임

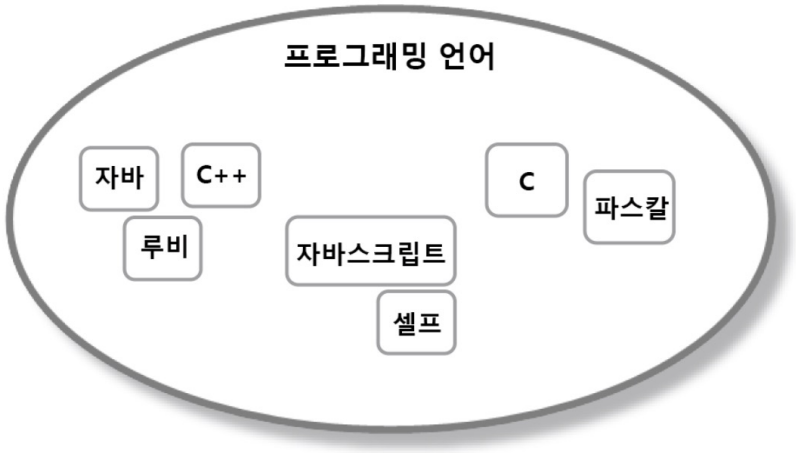
2. 타입 계층

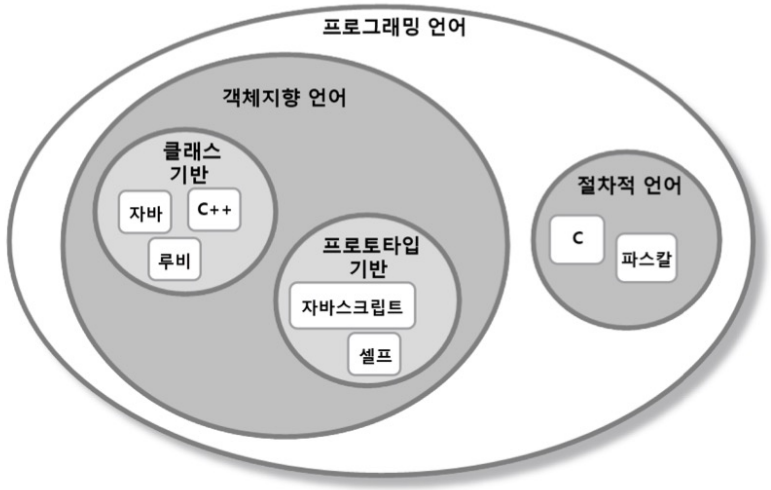
타입 계층

01. 타입 사이의 포함관계

- 타입은 객체들의 집합이기 때문에 다른 타입을 포함하는 것이 가능함
- 타입 안에 포함된 객체들을 좀 더 상세한 기준으로 묶어 새로운 타입을 정의하면

  이 새로운 타입은 자연스럽게 기존 타입의 부분집합이 됨

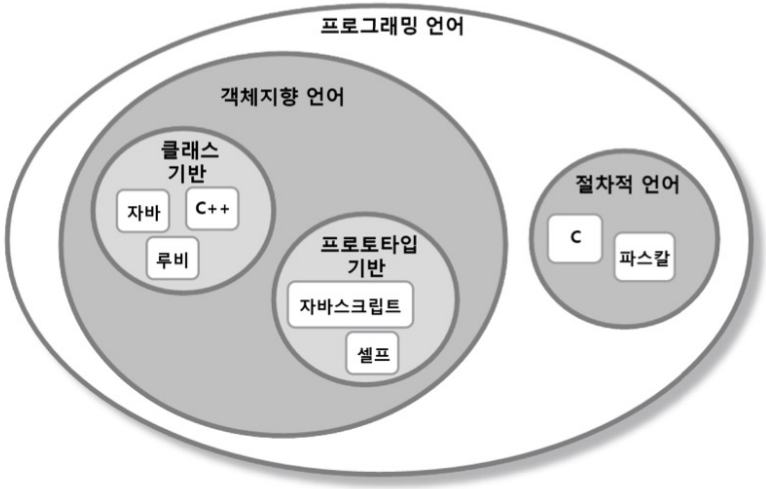




타입 계층

01. 타입 사이의 포함관계

- 타입이 다른 타입에 포함될 수 있기 때문에 동일한 인스턴스가 하나 이상의 타입으로 분류되는 것이 가능함
- 자바는 '프로그래밍 언어'인 동시에 '객체지향 언어'에 속하며 더 세부적으로 '클래스 기반 언어'에 속함
- 다른 타입을 포함하는 타입은 포함되는 타입보다 좀 더 일반화된 의미를 표현함
- 반면 포함되는 타입은 좀더 특수하고 구체적임

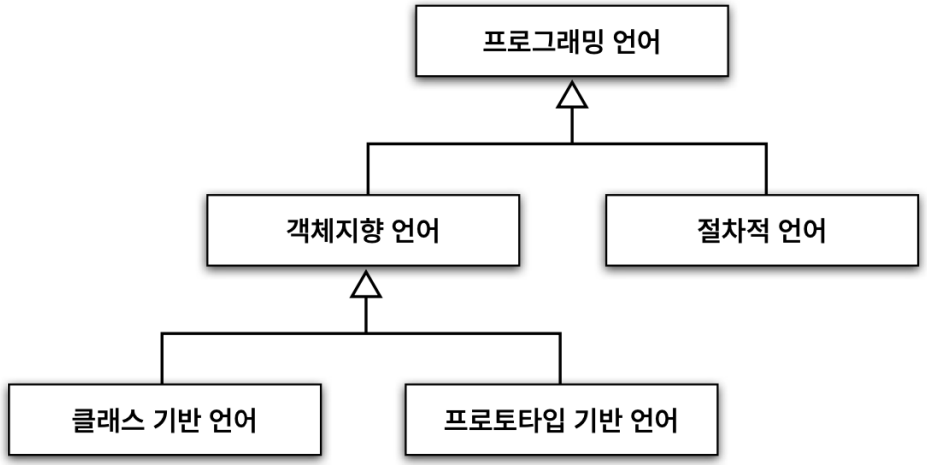


타입 계층

01. 타입 사이의 포함관계

- 특수화 관계를 가진 계층으로 표현할 수 있음
- 타입 계층을 구성하는 두 타입 간의 관계에서
  - 더 일반적인 타입을 슈퍼타입(supertype)이라고 부르고

    더 특수한 타입을 서브타입(subtype)이라고 부름



타입 계층

01. 타입 사이의 포함관계

내연과 외연 관점에서

- 슈퍼타입은
  - 집합이 다른 집합의 모든 멤버를 포함한다.
  - 타입 정의가 다른 타입보다 좀 더 일반적이다.
- 서브타입은
  - 집합에 포함되는 인스턴스들이 더 큰 집합에 포함된다.
  - 타입 정의가 다른 타입보다 좀 더 구체적이다.

타입 계층

- 객체의 타입을 결정하는 것은 퍼블릭 인터페이스
- 타입 계층 간의 관계를 형성하는 기준 또한 '퍼블릭 인터페이스'

- 서브타입의 인스턴스는 슈퍼타입의 인스턴스로 간주될 수 있음
- 이 사실이 상속과 다형성의 관계를 이해하기 위한 출발점

3. 서브클래싱과 서브타이핑

서브클래싱과 서브타이핑

상속의 올바른 용도는 타입 계층을 구현하는 것

그렇다면 올바른 상속의 기준은 무엇일까?

서브클래싱과 서브타이핑

01. 언제 상속을 사용해야 하는가?

마틴 오더스키는 아래의 두 질문에 모두 '예'라고 답할 수 있는 경우에만 상속을 사용하라고 조언함

1. 상속 관계가 is-a 관계를 모델링하는가?

- 이것은 애플리케이션을 구성하는 어휘에 대한 우리의 관점에 기반함
- "[자식 클래스]는 [부모 클래스]다"라고 말해도 이상하지 않다면 상속을 사용할 후보로 간주할 수 있음

서브클래싱과 서브타이핑

01. 언제 상속을 사용해야 하는가?

마틴 오더스키는 아래의 두 질문에 모두 '예'라고 답할 수 있는 경우에만 상속을 사용하라고 조언함

2. 클라이언트 입장에서 부모 클래스의 타입으로 자식 클래스를 사용해도 무방한가?

- 상속 계층을 사용하는 클라이언트의 입장에서 부모 클래스와 자식 클래스의 차이점을 몰라야함
- 이를 자식 클래스와 부모 클래스 사이의 행동 호환성이라고 부름

서브클래싱과 서브타이핑

01. 언제 상속을 사용해야 하는가?

첫 번째 질문보다는 두 번째 질문에 초점을 맞추는 것이 중요함

서브클래싱과 서브타이핑

02. Is-a 관계

- 마틴 오더스키의 조언에 따르면 두 클래스는 어휘적으로 is-a 관계를 만족해야 함
- “객체지향 언어는 프로그래밍 언어다”라고 표현할 수 있다면 is-a 관계를 만족하는 것
  - 자식: 객체지향 언어, 부모: 프로그래밍 언어
- 하지만 스콧 마이어스는 새와 펭귄의 예를 들어 is-a 관계가 직관을 쉽게 배신할 수 있다는 사실을 보여줌

서브클래싱과 서브타이핑

02. Is-a 관계

- 아래는 우리에게 익숙한 사실임

  A. 펭귄은 새다.

  B. 새는 날 수 있다.

- 두 가지 사실을 조합하면 오른쪽과 같은 코드를 얻게됨

```
public class Bird {
    public void fly() { ... }
    ...
}

public class Penguin extends Bird {
    ...
}
```

서브클래싱과 서브타이핑

02. Is-a 관계

- 이 코드는 반은 맞고 반은 틀림
  - 펭귄은 분명 새지만 날 수 없는 새다
- 하지만 코드는 분명히 “펭귄은 새고, 따라서 날 수 있다”

  라고 주장하고 있음

```
public class Bird {
    public void fly() { ... }
    ...
}

public class Penguin extends Bird {
    ...
}
```

서브클래싱과 서브타이핑

03. 행동 호환성

- 타입이 행동과 관련 있다는 사실에 주목해야함
- 타입의 이름 사이에 개념적으로 어떤 연관성이 있다고 하더라도 행동에 연관성이 없다면

  is-a 관계를 사용하면 안됨
- 어휘적으로 새와 펭귄은 is-a 관계를 만족하는 것처럼 보이지만,

  새와 펭귄의 서로 다른 행동 방식은 이 둘을 동일한 타입 계층으로 묶어서는 안된다고 경고하고 있음

서브클래싱과 서브타이핑

03. 행동 호환성

- 그렇다면 행동이 호환된다는 것은 무슨 의미일까?
- 중요한 것은 행동의 호환 여부를 판단하는 기준은 클라이언트의 관점이라는 것
- 클라이언트가 두 타입이 동일하게 행동할 것이라고 기대한다면 두 타입을 타입 계층으로 묶을 수 있음

서브클래싱과 서브타이핑

03. 행동 호환성

- Penguin이 Bird의 서브타입이 아닌 이유는 클라이언트 입장에서 모든 새가 날 수 있다고 가정하기 때문
- 타입 계층을 이해하기 위해서는 그 타입 계층이 사용될 문맥을 이해하는 것이 중요함

서브클래싱과 서브타이핑

03. 행동 호환성

```
public void flyBird(Bird bird) {
    bird.fly();
}
```

- 위와 같이 클라이언트가 날 수 있는 새만을 원한다고 가정
- 현재 Penguin은 Bird의 자식 클래스이기 때문에 컴파일러는 업캐스팅을 허용함
- flyBird 메서드 인자로 Penguin의 인스턴스가 전달되는 것을 막을 수 있는 방법이 없음
- Penguin은 클라이언트의 기대를 저버리기 때문에 Bird의 서브타입이 아님
- 이 둘을 상속 관계로 연결한 위 설계는 수정되어야함

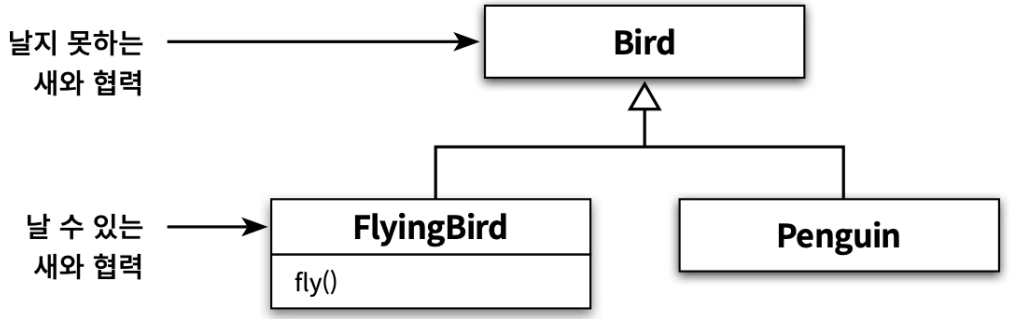
서브클래싱과 서브타이핑

04. 클라이언트의 기대에 따라 계층 분리하기 - 타입계층

- 행동 호환성을 만족시키지 않는 상속 계층을 그대로 유지한 채 클라이언트의 기대를 충족시킬 수 있는 방법은 찾기 쉽지 않음
- 문제를 해결할 수 있는 방법은 클라이언트의 기대에 맞게 상속 계층을 분리하는 것 뿐

서브클래싱과 서브타이핑

04. 클라이언트의 기대에 따라 계층 분리하기 - 타입계층



다음 코드처럼 날 수 있는 새와 날 수 없는 새를 명확하게

구분할 수 있도록 상속 계층을 분리하면,

서로 다른 요구사항을 가진 클라이언트를 만족시킬 수 있음

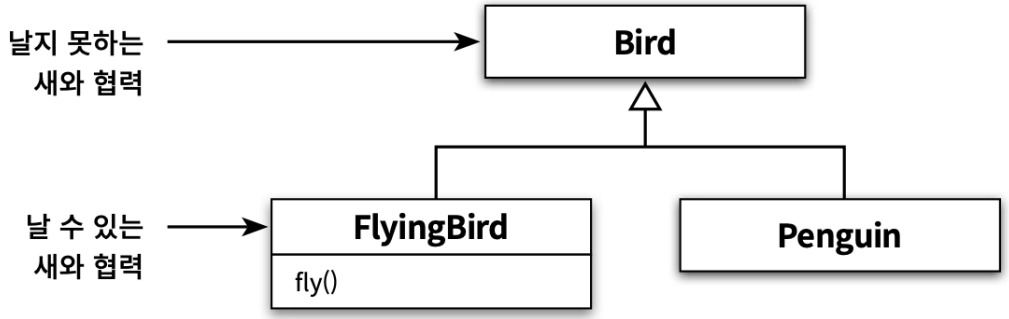
```
public class Bird {
    ...
}

public class FlyingBird extends Bird {
    public void fly() { ... }
    ...
}

public class Penguin extends Bird {
    ...
}
```

서브클래싱과 서브타이핑

04. 클라이언트의 기대에 따라 계층 분리하기 - 타입계층



만약 날 수 없는 새와 협력하는 메서드가 존재한다면

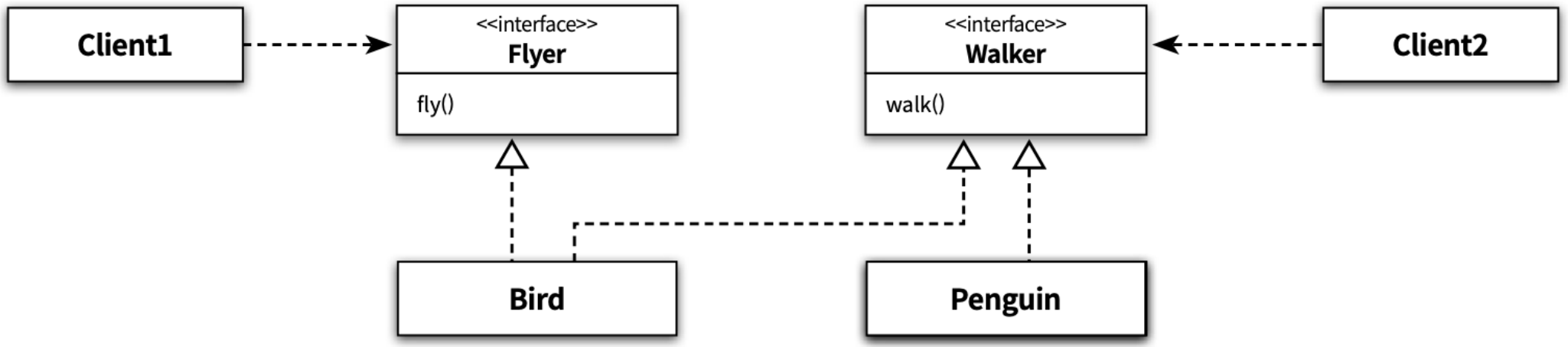
파라미터의 타입을 Bird로 선언하면 됨

```
public void flyBird(FlyingBird bird) {
    bird.fly();
}
```

서브클래싱과 서브타이핑

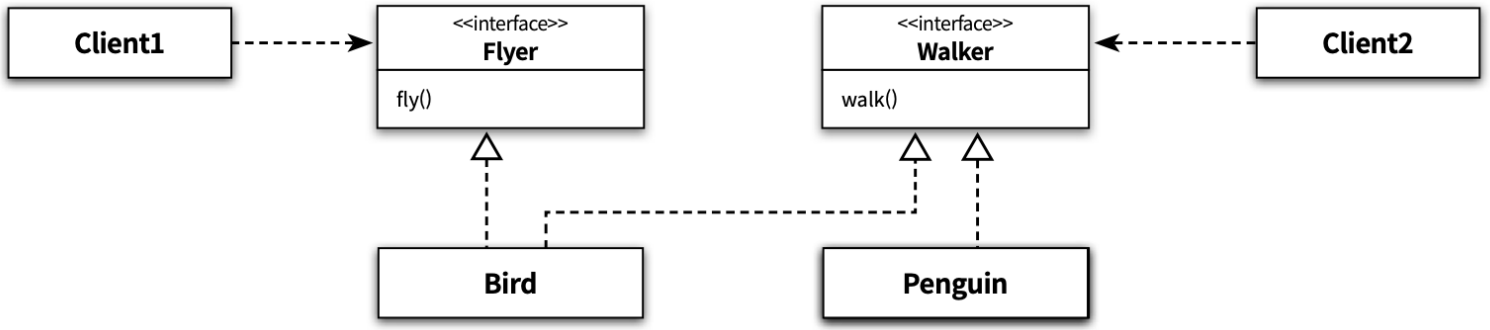
04. 클라이언트의 기대에 따라 계층 분리하기 - ISP

- 클라이언트에 따라 인터페이스를 분리할 수 도 있음
- 아래와 같이 인터페이스를 분리하면
- Bird와 Penguin은 자신이 수행할 수 있는 인터페이스만 구현하면 됨



서브클래싱과 서브타이핑

04. 클라이언트의 기대에 따라 계층 분리하기 - ISP



- 클라이언트에 따라 인터페이스를 분리하면 변경에 대한 영향을 더 세밀하게 제어할 수 있음
- 대부분의 경우 인터페이스는 클라이언트의 요구가 바뀜에 따라 변경됨
- 클라이언트에 따라 인터페이스를 분리(ISP)하면 각 클라이언트의 요구가 바뀌더라도

  영향의 파급효과를 효과적으로 제어할 수 있음

서브클래싱과 서브타이핑

상속을 사용하는 두 가지 목적에 따라 서브클래싱 혹은 서브타이핑이라고 칭함

서브클래싱(subsclassing)

- 다른 클래스의 코드를 재사용할 목적으로 상속을 사용하는 경우
- 자식 클래스와 부모 클래스의 행동이 호환되지 않기 때문에 자식 클래스의 인스턴스가 부모 클래스의 인스턴스를 대체할 수 없음
- **구현 상속**(implementation inheritance) 또는 **클래스 상속**(class inheritance)라고 부르기도 함

서브클래싱과 서브타이핑

상속을 사용하는 두 가지 목적에 따라 서브클래싱 혹은 서브타이핑이라고 칭함

서브타이핑(subtyping)

- 타입 계층을 구성하기 위해 상속을 사용하는 경우
- 서브타이핑에서는 행동 호환성을 만족하므로, 자식 클래스의 인스턴스가 부모 클래스의 인스턴스를 대체할 수 있음
- 이때 부모 클래스는 자식 클래스의 '슈퍼타입'이 되고 자식 클래스는 부모 클래스의 '서브타입'이 됨
- **인터페이스 상속(interface inferitance)**라고 부르기도함

서브클래싱과 서브타이핑

- 슈퍼타입과 서브타입 사이의 관계에서 가장 중요한 것은 퍼블릭 인터페이스임
- 슈퍼타입 인스턴스를 요구한 모든 곳에서 서브타입의 인스턴스를 대신 사용하기 위해 만족해야 하는 최소한의 조건은,
- 서브타입의 퍼블릭 인터페이스가 슈퍼타입에서 정의한 퍼블릭 인터페이스와 동일하거나 더 많은

  오퍼레이션을 포함해야 한다는 것
- 이것이 서브타이핑을 인터페이스 상속이라고 부르는 이유임

서브클래싱과 서브타이핑

- 서브타이핑 관계가 유지되기 위해서는 서브타입이 슈퍼타입이 하는 모든 행동을 동일하게 할 수 있어야함
- 즉, 어떤 타입이 다른 타입의 서브타입이 되기 위해서는 행동 호환성(behavioral substitution)을 만족해야 함

서브클래싱과 서브타이핑

- 자식 클래스가 부모 클래스를 대신할 수 있기 위해서는 자식 클래스는 부모 클래스가 사용되는 모든 문맥에서 동일하게 행동할 수 있어야함
- 다시 말해 두 클래스 사이의 행동 호환성은 부모 클래스에 대한 자식 클래스의 대체 가능성(substitutability)을 포함함

행동 호환성과 대체 가능성은 올바른 상속 관계를 구축하기 위한 지침이며,

리소코프 치환 원칙이라는 이름으로 정리되어 소개돼 왔음

4. 리스코프 치환 원칙

리스코프 치환 원칙

리소코프 치환 원칙을 한마디로 정리하면,

- “서브타입은 그것의 기반 타입에 대해 대체 가능해야 한다”는 것
- 클라이언트가 “차이점을 인식하지 못한 채 기반 클래스의 인터페이스를 통해 서브클래스를 사용할 수 있어야 한다”는 것

리스코프 치환 원칙

리스코프 치환 원칙은 행동 호환성을 설계 원칙으로 정리한 것

리스코프 치환 원칙

01. 리스코프 치환 원칙 사례

"정사각형은 직사각형이다"라는 *어휘적* is-a 관계 성립

```java
public class Rectangle {
    private int x, y, width, height;

    public Rectangle(int x, int y, int width, int height) {
        this.x = x;
        this.y = y;
        this.width = width;
        this.height = height;
    }

    getter, setter for width, height...

    public int getArea() {
        return width * height;
    }

}
```

```java
public class Square extends Rectangle {
    public Square(int x, int y, int size) {
        super(x, y, size, size);
    }

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

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

리스코프 치환 원칙

01. 리스코프 치환 원칙 사례

문제점: Rectangle과 협력하는 클라이언트는 직사각형의 너비와 높이가 다르다고 가정

```java
public void resize(Rectangle rectangle, int width, int height) {
    rectangle.setWidth(width);
    rectangle.setWidth(height);
}
```

리스코프 치환 원칙

01. 리스코프 치환 원칙 사례

- Square는 Rectangle의 구현을 재사용하고 있을 뿐
- 두 클래스는 LSP를 위반하기 때문에 서브타이핑 관계가 아닌 서브클래싱 관계임
- 이 예제는 is-a 라는 말이 얼마나 우리의 직관에서 벗어날 수 있는지를 잘 보여줌
- 중요한 것은 클라이언트 관점에서 행동이 호환되는지 여부임

리스코프 치환 원칙

02. 클라이언트와 대체 가능성

- Square가 Rectangle을 대체할 수 없는 이유는 클라이언트 관점에서 Square와 Rectangle이 다르기 때문
- 예상한 대로 작동하지 않으므로, 개발자는 밤을 새워 디버깅해야 하는 악몽과도 같은 상황에 빠질 수 있음
- 리스코프 치환 원칙은 자식 클래스가 부모 클래스를 대체하기 위해서는 부모 클래스에 대한 클라이언트의 가정을 준수해야 한다는 것을 강조함

리스코프 치환 원칙

02. 클라이언트와 대체 가능성

- LSP는 “클라이언트와 격리한 채로 본 모델을 의미 있게 검증하는 것이 불가능하다”는 아주 중요한 결론을 이끔
- 어떤 모델의 유효성은 클라이언트 관점에서만 검증 가능함

리스코프 치환 원칙

03. is-a 관계 다시 살펴보기

- 마틴 오더스키가 제안한 상속이 적합한지 판단하는 두 질문.
  - 어휘적으로 is-a 관계를 모델링한 것인가?
  - 클라이언트 입장에서 부모 클래스 대신 자식 클래스를 사용할 수 있는가?

리스코프 치환 원칙

03. is-a 관계 다시 살펴보기

- 사실 이 두 질문을 별개로 취급할 필요는 없음
- 클라이언트 관점에서 대체 불가능하다면 어휘적으로 is-a라고 말할 수 있다고 하더라도,

  그 관계를 is-a 관계라고 할 수 없음
- is-a는 클라이언트 관점에서 is-a일 때만 참일 수 있음
- is-a 관계로 표현된 문장을 볼 때마다 문장 앞에 “클라이언트 입장에서”라는 말이 빠져있다고 생각하면 됨

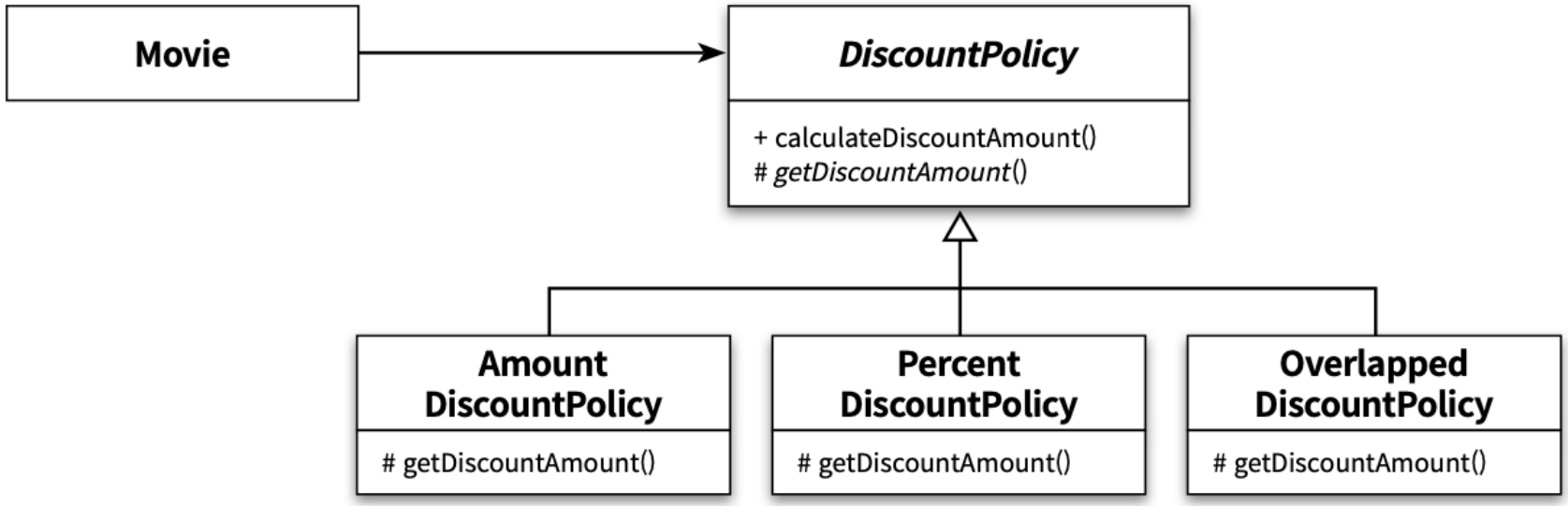
리스코프 치환 원칙

04. 리스코프 치환 원칙은 유연한 설계의 기반이다

- 클라이언트 입장에서 퍼블릭 인터페이스의 행동 방식이 변경되지 않는다면

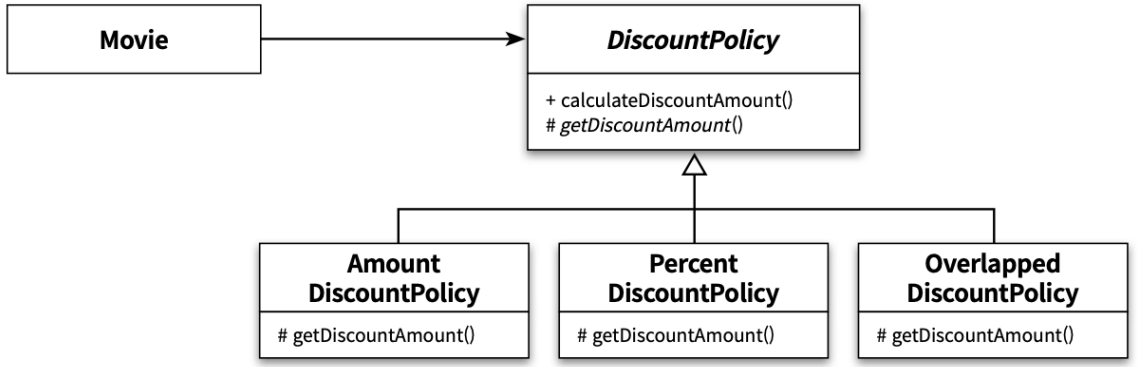
  클라이언트는 코드를 변경하지 않고도 새로운 자식 클래스와 협력할 수 있음

- 이전에 살펴봤던 할인 정책 예제에서 새로운 할인 정책을 추가하더라도 클라이언트를 수정할 필요가 없음



리스코프 치환 원칙

04. 리스코프 치환 원칙은 유연한 설계의 기반이다

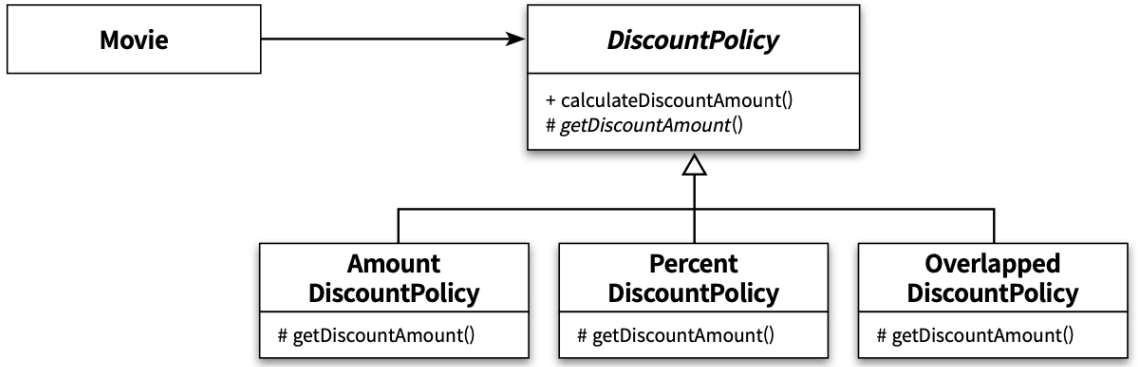


할인 정책 예제는 DIP, OCP, LSP가 한데 어우러져 설계를 확장하게 만듦

- 의존성 역전 원칙(DIP):
  - 구체 클래스인 Movie와 AmountDiscountPolicy 모두 추상 클래스인 DiscountPolicy에 의존함
  - ‘상위 수준의 모듈인 Movie’와 하위 ‘수준의 모듈인 AmountDiscountPolicy’는 모두 ‘추상 클래스인 DiscountPolicy’를 의존함

리스코프 치환 원칙

04. 리스코프 치환 원칙은 유연한 설계의 기반이다

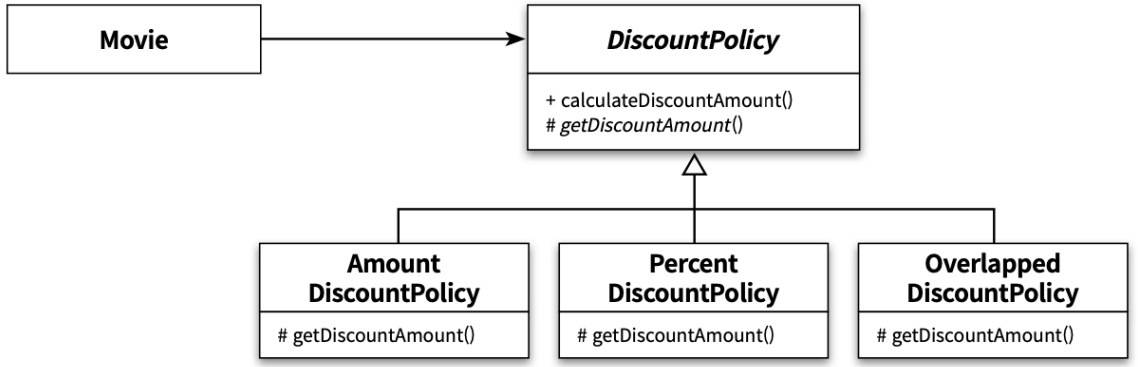


할인 정책 예제는 DIP, OCP, LSP가 한데 어우러져 설계를 확장하게 만듬

- 리스코프 치환 원칙(LSP):
  - Movie의 관점에서 DiscountPolicy 대신 AmountDiscountPolicy와 협력하더라도 아무런 문제가 없음
  - 다시 말해, AmountDiscountPolicy는 클라이언트에 대해 영향을 주지않고 DiscountPolicy를 대체할 수 있음

리스코프 치환 원칙

04. 리스코프 치환 원칙은 유연한 설계의 기반이다



할인 정책 예제는 DIP, OCP, LSP가 한데 어우러져 설계를 확장하게 만듬

- 개방-폐쇄 원칙(OCP):
  - 새로운 기능을 추가하기 위해 AmountDiscountPolicy를 추가하더라도 Movie에는 영향을 끼치지 않음
  - 다시 말해, 기능을 확장하면서 기존 코드를 수정할 필요가 없음

리스코프 치환 원칙

04. 리스코프 치환 원칙은 유연한 설계의 기반이다

- LSP이 어떻게 OCP를 지원하는지 눈여겨봐야함
- LSP는 OCP를 만족하는 설계를 위한 전제 조건임
- LSP 위반은 잠재적인 OCP 위반임

리스코프 치환 원칙

- 타입 계층을 구현할 수 있는 방법은 상속만 있는 것이 아님
- 자바의 인터페이스를 통해 서브타이핑 관계를 구현할 수 있음
- 핵심은 구현 방법과 무관하게 클라이언트의 관점에서 슈퍼타입에 대해 기대하는 모든 것이

  서브 타입에게도 적용되어야 한다는 것

감사합니다