오브젝트 10장

이번 장에서는 클래스를 재사용하기 위해 새로운 클래스를 추가하는 가장 대표적인 기법인 상속에 관해 살펴보기로 한다.

객체지향에서는 상속 외에도 코드를 효과적으로 재사용 할 수 있는 방법이 한가지 더 있는데 새로운 클래스의 인스턴스 안에 기존 클래스의 인스턴스를 포함시키는 방법으로 흔히 합성이라고 한다.

상속과 중복 코드

중복 코드는 사람들의 마음속에 의심과 불신의 씨앗을 뿌린다.

DRY 원칙

중복 코드는 변경을 방해한다. 이것이 중복코드를 제거해야 하는 가장 큰 이유다. 프로그램의 본질은 비즈니스와 관련된 지식을 코드로 변환하는 것이다. 안타깝게도 이 지식은 항상 변한다. 그 이유가 무엇이건 일단 새로운 코드를 추가하고 나면 언젠가는 변경될 것이라고 생각하는 것이 현명하다.

- 중복 여부를 판단하는 기준은 변경이다. 요구사항이 변경됐을 때 두 코드를 함께 수정해야 한다면 이 코드는 중복이다.
- 프로그래머들은 DRY 원칙을 따라야 한다. DRY는 '반복하지 마라'라는 뜻의 Don't Repeat Yourself의 첫 글자를 모아 만든 용어로 간단히 말해 동일한 지식을 중복하지 말라는 것이다.

> DRY 원칙
>
> 모든 지식은 시스템 내에서 단일하고, 애매하지 않고, 정말로 믿을 만한 표현 양식을 가져야 한다.

DRY 원칙은 한번, 단 한번 원칙 또는 단일 지점제어 원칙이라고 부른다. 원칙의 이름이 무엇이건 핵심은 코드 안에 중복이 존재해서는 안 된다는 것이다.

중복과 변경

중복 코드의 문제점을 이해하기 위해 한 달에 한 번씩 가입자별로 전화 요금을 계산하는 간단한 애플리케이션을 개발해 보자. 전화 요금을 계산하는 규칙은 간단한데 통화 시간을 단위 시간당 요금으로 나눠주면 된다.

요구사항은 항상 변한다. 그리고 우리의 애플리케이션 역시 예외일 수는 없다. 애플리케이션이 성공적으로 출시되고 시간이 흘러 '심야 할인 요금제'라는 새로운 요금 방식을 추가해야 한다는 요구사항이 접수됐다.

이 요구사항을 해결할 수 있는 쉽고도 가장 빠른방법은 Phone의 코드를 복사해서 NightlyDiscountPhone이라는 새로운 클래스를 만든 후 수정하는 것이다.

- 하지만 구현 시간을 절약한 대가로 지불해야 하는 비용은 예상보다 크다. 사실 Phone과 NightlyDiscountPhone 사이에는 중복 코드가 존재하기 때문에 언제 터질지 모르는 시한 폭탄을 안고 있는 것과 같다.

중복코드 수정하기

중복코드가 코드 수정에 미치는 영향을 살펴보기 위해 새로운 요구사항을 추가해 보자. 이번에 추가할 기능은 통화 요금에 부과할 세금을 계산하는 것이다. 현재 통화 요금을 계산하는 로직은 Phone과 NightlyDiscountPhone 양쪽 모두에 구현돼 있기 때문에 세금을 추가하기 위해서는 두 클래스를 함께 수정해야 한다.

1. 이 예제는 중복 코드가 가지는 단점을 잘 보여준다. 많은 코드 더미 속에서 어떤 코드가 중복인지를 파악하는 일은 쉬운 일이 아니다. 중복 코드는 항상 함께 수정돼야 하기 때문에 수정할 때 하나라도 빠트린다면 버그로 이어질 것이다.
2. 한발 양보해서 모든 중복 코드를 식별했고, 함께 수정했다고 하자. 더 큰 문제는 중복 코드를 서로 다르게 수정하기가 쉽다는 것이다.
3. 중복 코드는 새로운 중복코드를 부른다. 중복코드를 제거하지 않은 상태에서 코드를 수정할 수 있는 유일한 방법은 새로운 중복 코드를 추가하는 것 뿐이다. 새로운 중복코드를 추가하는 과정에서 코드의 일관성이 무너질 위험이 항상 도사리고 있다. 더 큰 문제는 중복 코드가 늘어날 수록 애플리케이션은 변경에 취약해지고 버그가 발생할 가능성이 높아진다는 것이다.
4. 민첩하게 변경하기 위해서는 중복 코드를 추가하는 대신 제거해야 한다. 기회가 생길 때 마다 코드를 DRY하게 만들기 위해 노력하라.

타입 코드 사용하기

두 클래스 사이의 중복 코드를 제거하는 한 가지 방법은 클래스를 하나로 합치는 것이다. 다음과 같이 요금제를 구분하는 타입 코드를 추가하고 타입 코드의 값에 따라 로직을 분기시켜 Phone과 NightlyDiscountPhone을 하나로 합칠 수 있다. 하지만 계속 강조했던 것 처럼 타입 코드를 사용하는 클래스는 낮은 응집도와 높은 결합도라는 문제에 시달리게 된다.

- 객체지향 프로그래밍 언어는 타입 코드를 사용하지 않고도 중복 코드를 관리할 수 있는 효과적인 방법을 제공한다. 상속이 바로 그것이다.

상속을 이용해서 중복 코드 제거하기

상속의 기본 아이디어는 매우 간단하다. 이미 존재하는 클래스와 유사한 클래스가 필요하다면 코드를 복사하지 말고 상속을 이용해 코드를 재사용하라는 것이다.

따라서 NightlyDiscountPhone 클래스가 Phone 클래스를 상속받게 만들면 코드를 중복시키지 않고도 Phone 클래스의 코드 대부분을 재 사용할 수 있다.

NightlyDiscountPhone 클래스의 calcuateFee 메서드를 자세히 살펴보면 이상한 부분이 눈에 띌 것이다. super 참조를 통해 부모 클래스의 Phone의 calcuateFee 메서드를 호출해서 일반 요금제에 따라 통화 요금을 계산한 후 이 값에서 통화 시작 시간이 10시 이후인 통화의 요금을 빼주는 부분이다.

이렇게 구현된 이유를 이해하기 위해서는 개발자가 Phone의 코드를 재사용하기 위해 세운 가정을 이해하는 것이 중요하다. NightlyDiscountPhone을 구현한 개발자는 Phone의 코드를 최대한 많이 재사용하고 싶었다. 개발자는 Phone이 구현하고 있는 일반 요금제는 1개의 요금 규칙으로 구성돼 있는 데 비해 NightlyDiscountPhone으로 구현할 심야 할인 요금제는 10시를 기준으로 분리된 2개의 요금제로 구성

돼 있다고 분석했다.

10시 이후의 통화 요금을 계산하는 경우에 대해서만 NightlyDiscountPhone에서 구현하기로 결정한 것이다.

- 이 예를 통해 알 수 있는 것처럼 상속을 염두에 두고 설계되지 않은 클래스를 상속을 이용해 재사용하는 것은 생각처럼 쉽지 않다. 개발자는 재사용을 위해 상속 계층 사이에 무수히 많은 가정을 세웠을지도 모른다. 그리고 그 가정은 코드를 이해하기 어렵게 만들뿐만 아니라 직관에도 어긋날 수 있다.
- 상속을 이용해 코드를 재사용하기 위해서는 부모 클래스의 개발자가 세웠던 가정이나 추론 과정을 정확하게 이해해야 한다. 이것은 자식 클래스의 작성자가 부모 클래스의 구현 방법에 대한 정확한 지식을 가져야 한다는 것을 의미한다.
- 상속은 결합도를 높인다. 상속이 초래하는 부모 클래스와 자식 클래스 사이의 강한 결합이 코드를 수정하기 어렵게 만든다.

강하게 결합된 Phone과 NightlyDiscountPhone

부모 클래스와 자식 클래스 사이의 결합이 문제인 이유를 살펴보자. 만약 세금을 부과하는 요구사항이 추가된다면 어떻게 될까?

NightlyDiscountPhone을 Phone의 자식 클래스로 만든 이유는 Phone의 코드를 재사용하고 중복 코드를 제거하기 위해서다. 하지만 세금을 부과하는 로직을 추가하기 위해 Phone을 수정할 때 유사한 코드를 NightlyDiscountPhone에도 추가해야 했다. 다시 말해서 코드 중복을 제거하기 위해 상속을 사용했음에도 세금을 계산하는 로직을 추가하기 위해 새로운 중복 코드를 만들어야 하는 것이다.

> 상속을 위한 경고 1
>
> 자식 클래스의 메서드 안에서 super 참조를 이용해 부모 클래스의 직접 호출할 경우 두 클래스는 강하게 결합된다. super 호출을 제거할 수 있는 방법을 찾아 결합도를 제거하라.

상속을 사용하면 적은 노력으로도 새로운 기능을 쉽고, 빠르게 추가할 수 있다. 하지만 그로 인해 커다란 대가를 치러야 할 수도 있다.

- 이처럼 상속 관계로 연결된 자식 클래스가 부모 클래스의 변경에 취약해지는 현상을 가리켜 취약한 기반 클래스 문제라고 한다. 취약한 기반 클래스 문제는 코드 재사용을 목적으로 상속을 사용할 때 발생하는 가장 대표적인 문제다.

취약한 기반 클래스 문제

지금까지 살펴본 것 처럼 상속은 자식 클래스와 부모 클래스의 결합도를 높인다.

이처럼 부모 클래스의 변경에 의해 자식 클래스가 영향을 받는 현상을 **취약한 기반 클래스 문제**라고 부른다. 이 문제는 상속을 사용한다면 피할 수 없는 객체지향 프로그래밍의 근본적인 취약성이다.

> 이제 결합도의 개념을 상속에 적용해보자. 구현을 상속한 경우, 파생 클래스는 기반 클래스에 강하게 결합되며, 이 둘 사이의 밀접한 연결은 바람직하지 않다. 설계자들은 이런 현상에 대해 "취약한 기반 클래스 문제" 라는 명칭을 붙였다. 겉으로 보기에는 안전한 방식으로 기반 클래스를 수정한 것처럼 보이더라도 이 새로운 행동이 파생 클래스에게 상속될 경우 파생 클래스의 잘못된 동작을 초래할 수 있기 때문에 기반 클래스는 "취약하다". 단순히 기반 클래스의 메서드들만을 조사하는

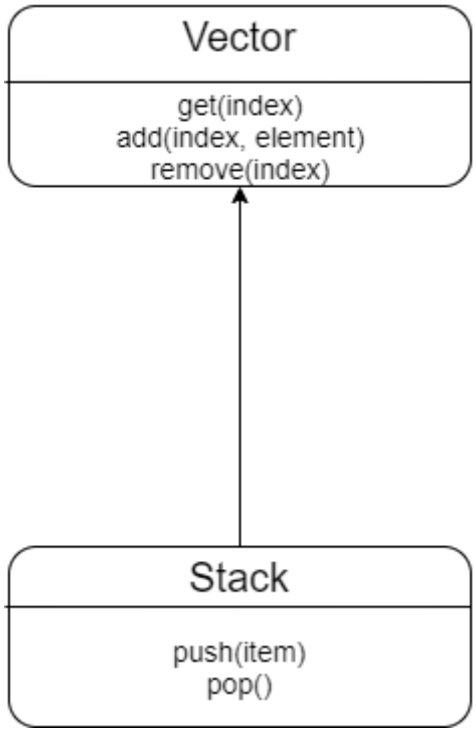
> 것만으로는 기반 클래스를 변경하는 것이 안전하다고 확신할 수 없다. 모든 파생 클래스들을 살펴봐야 한다. 나아가 기반 클래스와 파생 클래스를 사용하는 모든 코드가 새로운 코드로 인해 영향을 받지 않았는지 점검해야 한다.

- 취약한 기반 클래스 문제는 캡슐화를 약화시키고 결합도를 높인다.
- 객체를 사용하는 이유는 구현과 관련된 세부사항을 퍼블릭 인터페이스 뒤로 캡슐화할 수 있기 때문이다.
- 안타깝게도 상속을 사용하면 부모 클래스의 퍼블릭 인터페이스가 아닌 구현을 변경하더라도 자식 클래스가 영향을 받기 쉬워진다.

불필요한 인터페이스 상속 문제

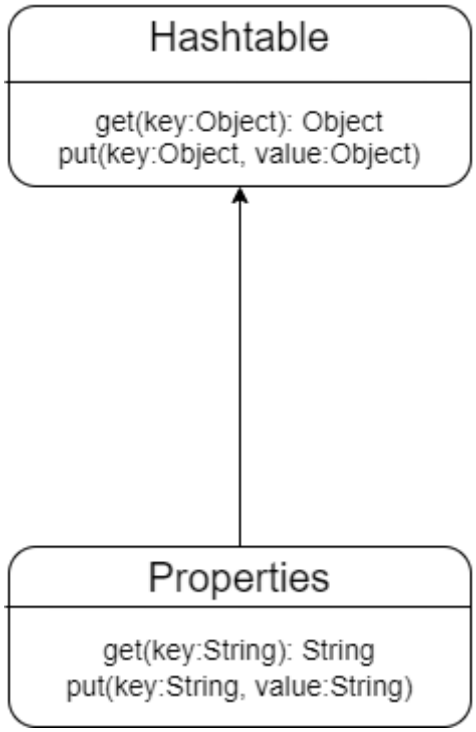
자바의 초기 버전에서 상속을 잘못 사용한 대표적인 사례는 Properties와 Stack이다. 두 클래스의 공통점은 부모 클래스에서 상속받은 메서드를 사용할 경우 자식 클래스의 규칙이 위반 될 수 있다는 것이다.

자바의 초기 컬렉션 프레임워크 개발자들은 요소의 추가, 삭제 오퍼레이션을 제공하는 Vector를 재사용하기 위해 Stack을 Vector의 자식 클래스로 구현했다.



- 안타깝게도 Stack이 Vector를 상속받기 때문에 Stack의 퍼블릭 인터페이스에 Vector의 퍼블릭 인터페이스가 합쳐진다. 따라서 Stack에게 상속된 Vector의 퍼블릭 인터페이스를 이용하면 임의의 위치에서 요소를 추가하거나 삭제할 수 있다. 따라서 맨 마지막 위치에서만 요소를 추가하거나 제거할 수 있도록 허용하는 Stack의 규칙을 쉽게 위반할 수 있다.

Properties 클래스는 잘못된 유산을 물려받는 또 다른 클래스다. Properties 클래스는 키와 값의 쌍을 보관한다는 점에서는 Map과 유사하지만 다양한 타입을 저장할 수 있는 Map과 달리 키와 값의 타입으로 오직 String만 가질 수 있다.



- 이 클래스는 Map의 조상인 Hashtable을 상속받는데 자바에 제네릭이 도입되기 이전에 만들어 졌기 때문에 키와 값의 타입이 String인지 여부를 체크할 수 있는 방법이 없었다. 따라서 Hashtable의 인터페이스에 포함돼 있는 put 메서드를 이용하면 String 타입 이외의 키와 값이라도 Properties에 저장할 수 있다.

- Stack 과 Properties의 예는 퍼블릭 인터페이스에 대한 고려 없이 단순히 코드 재사용을 위해 상속을 이용하는 것이 얼마나 위험한지를 잘 보여준다. **객체지향의 핵심은 객체들의 협력이다 단순히 코드를 재사용하기 위해 불필요한 오퍼레이션이 인터페이스에 스며들도록 방치해서는 안된다.**

> 상속을 위한 경고 2
>
> 상속 받은 부모 클래스의 메서드가 자식 클래스의 내부 구조에 대한 규칙을 깨트릴 수 있다.

메서드 오버라이딩의 오작용 문제

조슈아 블로치는 이펙티브 자바에서 HashSet의 구현에 강하게 결합된 InstrumentedHashSet 클래스를 소개한다.

```
public class InstrumentedHashSet<E> extends HashSet<E> {
    private int addCount = 0;

    @Override
    public boolean add(E e) {
        addCount++;
        return super.add(e);
    }

    @Override
```

```
    public boolean addAll(Collection<? extends E> c) {
        addCount += c.size();
        return super.addAll(c);
    }
}
```

InstrumentedHashSet 은 요소를 추가한 횟수를 기록하기 위해 addCount라는 인스턴스 변수를 포함한다.

InstrumentedHashSet 은 요소가 추가될 때마다 추가되는 요소의 개수마늨ㅁ addCount의 값을 증가시키기 위해 하나의 요소를 추가하는 add 메서드와 다수의 요소들을 한 번에 추가하는 addAll 메서드를 오버라이딩 한다.

InstrumentedHashSet 의 구현에는 아무런 문제가 없어 보인다. 적어도 다음과 같은 코드를 실행하기 전까지는 말이다.

```
InstrumentedHashSet<String> languages = new InstrumentedHashSet<>();
languages.addAll(Arrays.asList("Java", "Ruby", "Scala"));
```

대부분의 사람들은 위 코드를 실행한 후에 addCount의 값이 3이 될 거라고 예상할 것이다. 하지만 실제로 실행한 후의 addCount의 값은 6이다. 그 이유는 부모 클래스인 HashSet의 addAll 메서드 안에서 add 메서드를 호출하기 때문이다.

먼저 InstrumentedHashSet 의 addAll 메서드가 호출돼서 addCount에 3이 더해진다. 그 후 super.addAll 메서드가 호출되고 제어는 부모 클래스인 HashSet으로 이동한다. 불행하게도 HashSet은 각각의 요소를 추가하기 위해 내부적으로 add 메서드를 호출하고 결과적으로 InstrumentedHashSet 의 add 메서드가 세번 호출되어 addCount에 3이 더해지는 것이다. 따라서 최종 결과는 6이 된다.

> 상속을 위한 경고 3
>
> 자식 클래스가 부모 클래스의 메서드를 오버라이딩할 경우 부모 클래스가 자신의 메서드를 사용하는 방법에 자식 클래스가 결합될 수 있다.

조슈아 블로치는 클래스가 상속되기를 원한다면 상속을 위해 클래스를 설계하고 문서화 해야 하며, 그렇지 않은 경우에는 상속을 금지시켜야 한다고 주장한다.

객체지향의 핵심이 구현을 캡슐화 하는 것인데도 이렇게 내부 구현을 공개하고 문서화 하는 것이 옳은가?

> 잘된 API문서는 메서드가 무슨 일을 하는지를 기술해야 하고, 어떻게 하는지를 설명해서는 안 된다는 통념을 어기는 것은 아닐까? 그렇다. 어기는 것이다! 이것은 결국 상속이 캡슐화를 위반함으로써 초래된 불행인 것이다. 서브클래스가 안전할 수 있게끔 클래스를 문서화하려면 클래스의 상세 구현 내역을 기술해야 한다.

설계는 트레이드오프 활동이라는 사실을 기억하라. 상속은 코드 재사용을 위해 캡슐화를 희생한다. 완벽한 캡슐화를 원한다면 코드 재사용을 포기하거나 상속 이외의 다른방법을 사용해야 한다.

부모 클래스와 자식 클래스의 동시 수정 문제

요구사항이 변경되면 자식클래스가 부모 클래스의 메서드를 오버라이딩하거나 불필요한 인터페이스를 상속받지 않았음에도 부모 클래스를 수정할 때 자식 클래스를 함께 수정해야 할 수도 있다는 사실을 잘 보여준다.

결합도란 다른대상에 대해 알고 있는 지식의 양이다. 상속은 기본적으로 부모 클래스의 구현을 재사용한다는 기본 전제를 따르기 때문에 자식 클래스가 부모 클래스의 내부에 대해 속속들이 알도록 강요한다. 따라서 코드 재사용을 위한 상속은 부모 클래스와 자식 클래스를 강하게 결합시키기 때문에 함께 수정해야 하는 상황 역시 빈번하게 발생할 수 밖에 없는 것이다.

> 상속을 위한 경고 4
>
> 클래스를 상속하면 결합도로 인해 자식 클래스와 부모 클래스의 구현을 영원히 변경하지 않거나, 자식 클래스와 부모 클래스를 동시에 변경하거나 둘 중 하나를 선택할 수 밖에 없다.

Phone 다시 살펴보기

상속으로 인한 피해를 최소화할 수 있는 방법을 찾아보자. 취약한 기반 클래스 문제를 완전히 없앨 수는 없지만 어느 정도까지 위험을 완화시키는 것은 가능하다. 문제 해결의 열쇠는 바로 추상화다.

추상화에 의존하자

이 문제를 해결하는 가장 일반적인 방법은 자식 클래스가 부모 클래스의 구현이 아닌 추상화에 의존하도록 만드는 것이다. 정확하게 말하면 부모 클래스와 자식 클래스 모두 추상화에 의존하도록 수정해야 한다.

개인적으로 코드 중복을 제거하기 위해 상속을 도입할 때는 따르는 두가지 원칙이 있다.

- 두 메서드가 유사하게 보인다면 차이점을 메서드로 추출하라. 메서드 추출을 통해 두 메서드를 동일한 형태로 보이도록 만들 수 있다.
- 부모 클래스의 코드를 하위로 내리지 말고 자식 클래스의 코드를 상위로 올려라. 부모 클래스의 구체적인 메서드를 자식 클래스로 내리는 것보다 자식 클래스의 추상적인 메서드를 부모 클래스로 올리는 것이 재사용성과 응집도 측면에서 더 뛰어난 결과를 얻을 수 있다.

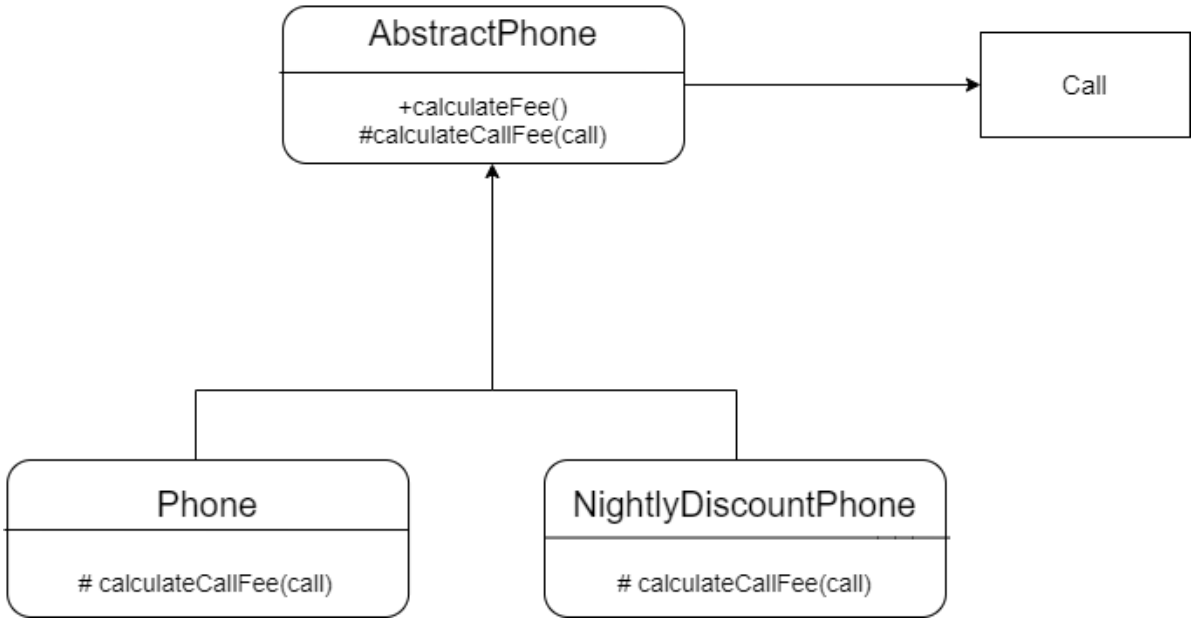
차이를 메서드로 추출하라

가장 먼저 할 일은 중복 코드 안에서 차이점을 별도의 메서드로 추출하는 것이다.

"변하는 것으로부터 변하지 않는 것을 분리하라" 또는 "변하는 부분을 찾고 이를 캡슐화하라"라는 조언을 메서드 수준에서 적용하자

1. 먼저 할일은 두 클래스의 메서드에서 다른부분을 별도의 메서드로 추출하는 것이다. 이 메서드는 하나의 Call에 대한 통화 요금을 계산하는 것이므로 메서드의 이름으로는 calculateCallFee가 좋을 것 같다.

2. 두 클래스의 calculateCallFee메서드를 동일하게 만들고 추출한 calculateCallFee 메서드 안에 서로 다른부분을 격리시켜 놓자
3. 이제 Phone과 NightlyDiscountPhone의 공통 부분을 부모 클래스로 이동시키자. 공통 코드를 옮길 때 인스턴스 변수보다 메서드를 먼저 이동시키는 게 편한데, 메서드를 옮기고 나면 그 메서드에 필요한 메서드나 인스턴스 변수가 무엇인지를 컴파일 에러를 통해 자동으로 알 수 있게 때문이다. 컴파일 에러를 바탕으로 메서드와 인스턴스 변수를 이동시키면 불필요한 부분은 자식 클래스에 둔 채로 부모 클래스에 꼭 필요한 코드만 이동시킬 수 있다.



리펙토링 후의 상속 계층

지금까지 살펴본 것처럼 자식 클래스들 사이의 공콩점을 부모 클래스로 옮김으로써 실제 코드를 기반으로 상속 계층을 구성할 수 있다. 이제 우리의 설계는 추상화에 의존하게 된다. 이 말의 의미를 살펴보자

> '위로 올리기' 전략은 실패했더라도 수정하기 쉬운 문제를 발생시킨다. 문제는 쉽게 찾을 수 있고 쉽게 고칠 수 있다. 추상화하지 않고 빼먹은 코드가 있더라도 하위 클래스가 해당 행동을 필요로 할 때가 오면 이 문제는 바로 눈에 띈다.
>
> 하지만 이 리팩터링을 반대 방향으로 진행한다면, 다시 말해 구체적인 구현을 아래로 내리는 방식으로 현재 클래스를 구체 클래스에서 추상 클래스로 변경하려 한다면 작은 실수 한 번으로도 구체적인 행동을 상위 클래스에 남겨 놓게 된다.

추상화가 핵심이다.

공통 코드를 이동시킨 후에 각 클래스는 서로 다른변경의 이유를 가진다는 것에 주목하라. AbstractPhone은 전체 통화 목록을 계산하는 방법이 바뀔 경우에만 변경된다. Phone은 일반 요금제의 통화 한건을 계산하는 방식이 바뀔 경우에만 변경된다. NightlyDiscountPhone은 심야 할인 요금제의 통화 한건을 계산하는 방식이 바뀔 경우에만 변경된다. 세 클래스는 각각 하나의 변경 이유만을 가진다. 클래스들은 단일 책임 원칙을 준수하기 때문에 응집도가 높다.

- 새로운 요금제를 추가하기도 쉽다는 사실 역시 주목하라. 새로운 요금제가 필요하다면 AbstractPhone을 상속 받는 새로운 클래스를 추가한 후 calculateCalFee 메서드만 오버라이딩하면 된다.

- 상속 계층이 코드를 진화시키는 데 걸림돌이 된다면 추상화를 찾아내고 상속 계층 안의 클래스들이 그 추상화에 의존하도록 코드를 리팩터링하라.

의도를 드러내는 이름 선택하기

한가지 아쉬운 점이 있다. 바로 클래스의 이름과 관련된 부분이다. NightlyDiscountPhone이라는 이름은 심야 할인 요금제와 관련된 내용을 구현한다는 사실을 명확하게 전달한다. 그에 비해 Phone은 일반 요금제와 관련된 내용을 구현한다는 사실을 명시적으로 전달하지 못한다.

따라서 AbstractPhone은 Phone으로, Phone은 RegularPhone으로 변경하는 것이 적절할 것이다.

세금 추가하기

인스턴스 변수의 목록이 변하지 않는 상황에서 객체의 행동만 변경된다면 상속 계층에 속한 각 클래스들을 독립적으로 진화시킬 수 있다. 하지만 인스턴스 변수가 추가되는 경우는 다르다. 사기 클래스는 자신의 인스턴스를 생성할 때 부모 클래스에 정의도니 인스턴스 변수를 초기화해야 하기 때문에 자연스럽게 부모 클래스에 추가된 인스턴스 변수는 자식 클래스의 초기화 로직에 영향을 미치게 된다.

결과적으로 책임을 아무리 잘 분리하더라도 인스턴스 변수의 추가는 종종 상속 계층 전반에 걸친 변경을 유발한다

하지만 인스턴스 초기화 로직을 변경하는 것이 두 클래스에 동일한 세금 계산 코드를 중복시키는 것보다는 현명한 선택이다.

- 지금까지 살펴본 것처럼 상속으로 인한 클래스 사이의 결합을 피할 수 있는 방법은 없다. 상속은 어떤 방식으로든 부모 클래스와 자식 클래스를 결합시킨다.

차이에 의한 프로그래밍

지금까지 살펴본 것처럼 상속을 사용하면 이미 존재하는 클래스의 코드를 기반으로 다른부분을 구현 함으로써 새로운 기능을 쉽고 빠르게 추가할 수 있다.

이처럼 기존 코드와 다른부분만을 추가함으로써 애플리케이션의 기능을 확장하는 방법을 **차이에 의한 프로그래밍**이라고 부른다. 상속을 이용하면 이미 존재하는 클래스의 코드를 쉽게 재사용할 수 있기 때문에 애플리케이션의 점진적인 정의가 가능해 진다.

- 차이에 의한 프로그래밍의 목표는 중복 코드를 제거하고 코드를 재사용하는 것이다.
- 객체지향 세계에서 중복 코드를 제거하고 코드를 재사용할 수 있는 가장 유명한 방법은 상속이다. 여러 클래스에 공통적으로 포함돼 있는 중복 코드를 하나의 클래스로 모은다. 원래 클래스들에서 중복 코드를 제거한 후 중복 코드가 옮겨진 클래스를 상속 관걔로 연결한다. 코드를 컴파일 하면 무대 뒤에서 마법이 일어나 상속 관계로 연결된 코드들이 하나로 합쳐진다. 따라서 상속을 사용하면 여러 클래스 사이에서 재사용 가능한 코드를 하나의 클래스 안으로 모을 수 있다.
- 상속은 강력한 도구다. 상속을 이용하면 새로운 기능을 추가하기 위해 직접 구현해야 하는 코드의 양을 최소화 할 수 있다. 상속은 너무나도 매력적이기 때문에 객체지향 프로그래밍에 갓 입문한 프로그래머들은 상속의 매력에 도취된 나머지 모든 설계에 상속을 적용하려고 시도한다.
- 상속의 오용과 남용은 애플리 케이션을 이해하고 확장하기 어렵게 만든다. 정말로 필요한 경우에만 상속을 사용하라

상속은 코드 재사용과 관련된 대부분의 경우에 우아한 해결 방법이 아니다. 객체지향에 능숙한 개발자들은 상속의 단점을 피하면서도 코드를 재사용할 수 있는 더 좋은 방법이 있다는 사실을 알고 있다. 바로 합성이다