8장 의존성 관리하기

의존성 이해하기

변경과 의존성

- 어떤 객체가 협력하기 위해 다른 객체를 필요로 할 때 두 객체 사이에 의존성이 존재하게 된다. 의존성은 실행 시점과 구현 시점에 서로 다른 의미를 가진다.
- 실행 시점 : 의존하는 객체가 정상적으로 동작하기 위해서는 실행 시에 의존 대상 객체가 반드시 존재해야 한다.
- 구현 시점 : 의존 대상 객체가 변경 될 경우 의존하는 객체도 함께 변경된다.

```
public class PeriodCondition implements DiscountCondition {
  private DayOfWeek dayOfWeek;
  private LocalTime startTime;
  private LocalTime endTime;

  ...

  public boolean isSatisfiedBy(Screening screening) {
    return screening.getStartTime().getDayOfWeek().equals(dayOfWeek) &&
        startTime.compareTo(screening.getStartTime().toLocalTime()) <= 0 &&
        endTime.compareTo(screening.getStartTime().toLocalTime()) >= 0;
  }
}
```

의존성 역시 마찬가지다. 두 요소 사이의 의존성은 의존되는 요소가 변경될 때 의존하는 요소도 함께 변경될 수 있다는 의미한다.

따라서 의존성은 변경에 의한 영향의 전파 가능성을 암시한다.

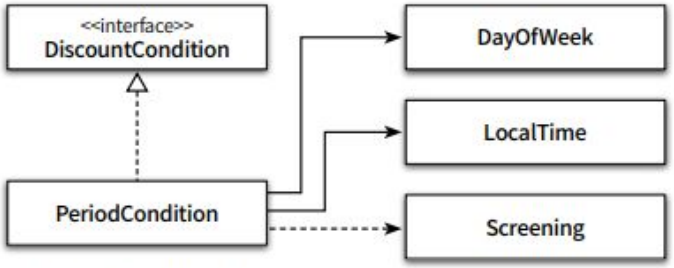

그림 8.3 PeriodCondition이 가지는 의존성의 종류를 강조

비록 의존성을 다른 방식으로 표기했지만 의존성이 가지는 근본적인 특성은 동일하다. PeriodCondition은 자신이 의존하는 대상이 변경될 때 함께 변경될 수 있다는 것이다.

UML과 의존성

UML(Unified Modeling Language)에 익숙한 사람이라면 여기서 설명하는 내용이 UML에서 정의하는 의존 관계와는 조금 다르다는 사실을 눈치챘을 것이다. UML에서는 두 요소 사이의 관계로 실체화 관계(realization), 연관 관계(association), 의존 관계(dependency), 일반화/특수화 관계(generalization/specialization), 합성 관계(composition), 집합 관계(aggregation) 등을 정의한다. 그림 8.3에는 이 중에서 실체화 관계, 연관 관계, 의존 관계가 포함돼 있다.

이번 장에서 다루고 있는 '의존성'은 UML의 의존 관계와는 다르다. UML은 두 요소 사이에 존재할 수 있는 다양한 관계의 하나로 '의존 관계'를 정의한다. 의존성은 두 요소 사이에 변경에 의해 영향을 주고받는 힘의 역학관계가 존재한다는 사실에 초점을 맞춘다. 따라서 UML에 정의된 모든 관계는 의존성이라는 개념을 포함한다. 예를 들어 PeriodCondition과 DiscountCondition 사이에는 UML에서 이야기하는 실체화 관계가 존재하지만 DiscountCondition에 대한 어떤 변경이 PeriodCondition에 대한 변경을 초래할 수 있기 때문에 두 요소 사이에는 의존성이 존재한다고 말할 수 있다.

이번 장에서 말하는 의존성을 단순히 UML에서 이야기하는 의존 관계로 해석해서는 안 된다. 의존성은 UML에서 정의하는 모든 관계가 가지는 공통적인 특성으로 바라봐야 한다.

의존성 전이

의존성은 전이될 수 있다.

의존성은 함께 변경될 수 있는 가능성을 의미하기 때문에 모든 경우에 의존성이 전이되는 것은 아니다.

의존성이 실제로 전이될지 여부는 변경의 방향과 캡슐화의 정도에 따라 달라진다.

의존성은 전이될 수 있기 때문에 의존성의 종류를 직접 의존성(direct dependency)과 간접 의존성(indirect dependency) 으로 나누기도 한다.

직접 의존성이란 말 그대로 한 요소가 다른 요소에 직접 의존하는 경우를 가리킨다.

간접 의존성이란 직접적인 관계는 존재하지 않는지만 의존성 전이에 의해 영향이 전파되는 경우를 가리킨다.

여기서는 클래스를 예로 들어 설명했지만 변경과 관련이 있는 어떤 것에도 의존성이라는 개념을 적용할 수 있다.

의존성의 대상은 객체일 수도 있고 모듈이나 더 큰 규모의 실행 시스템일수 도 있다. 의존성의 대상은 객체일 수도 있고 모듈이나 더 큰 규모의 실행 시스템일 수도 있다. 하지만 의존성의 본질은 변하지 않는다. 의존성이란 의존하고 있는 대상 변경에 영향을 받을 수 있는 가능성이다.

런타임 의존성과 컴파일 타임 의존성

- 의존성과 관련해서 다뤄야 하는 또 다른 주제는 런타임 의존성(run-time dependency)
- 컴파일타임 의존성 (compile-time dependency) 의 차이다..

- 런타임은 간단하다.
  말 그대로 어플리케이션이 실행되는 시점을 가리킨다.
  일반적으로 컴파일 타임이란 작성된 코드를 컴파일 하는 시점 가리키지만
  문맥에 따라서는 코드 그 자체를 가리키기도 한다.

- 컴파일타임 의존성이 바로 이런 경우에 해당한다.
  컴파일 타임의존성이라는 용어가 중요하게 생각하는 것은 시간이 아니라 우리가 작성한 코드의 구조이기 때문이다.
- 동적타입 언어의 경우에는 컴파일 타임이 존재 하지 않기 때문에 컴파일타임 의존성이라는 용어를 실제로 컴파일이 수행되는 시점으로 이해하면 의미가 모호해질 수 있다.

- 객체지향 어플리케이션에서 런타임의 주인공은 객체다.
  런타임 의존성이 다루는 주제는 객체 사이의 의존이다.
  반면 코드 관점에서 주인공은 클래스다. 따라서 컴파일 타임 의존성이 다루는 주제는 클래스사이의 의존성이다.

- 여기서 중요한 것은 런타임 의존성과 컴파일 타임의존성이 다를 수 있다.사실 유연하고 재사용 가능한 가능한 코드를 설계하기 위해서는 두 종류의 의존성을 서로 다르게 만들어야한다.

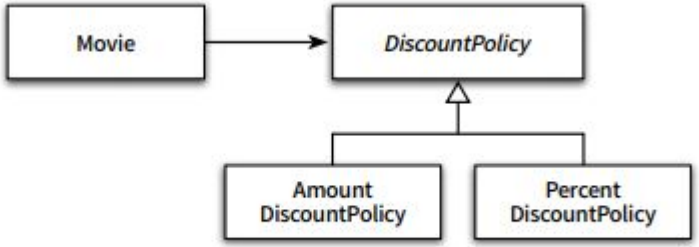

그림 8.5 코드 작성 시점의 Movie와 DiscountPolicy 사이의 의존성

Movie 는 가격을 계산하기 위해 비율 할인을 정책과 금액 할인 정책 모두를 적용할 수 있게 설계해야 한다.

=> Movie는 AmountDiscountPolicy와 PercentDiscountPolicy 모두와 협력할 수 있어야 한다.

=> AmountDiscountPolicy와 PercentDiscountPolicy가 추상 클래스인 DiscountPolicy를 상속받게 한 후 Movie가 이 추상클래스에 의존하도록 클래스 관계를 설계했다.

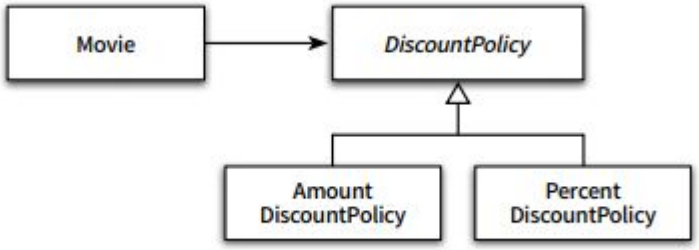

그림 8.5 코드 작성 시점의 Movie와 DiscountPolicy 사이의 의존성

Moive 클래스에서 AmountDiscointPolicy 클래스와 PercentDiscountPolicy 클래스로 향하는 어떤 의존성도 존재하지 않는다는 것이다.

Movie 클래스는 오직 추상 클래스인 DiscountPolicy클래스에만 의존한다.

Moive클래스의 코드를 살펴보면 AmountDiscountPolicy나 PercentDiscountPolicy에 대해서는 언급조차 하지 않는다.

```
public class Movie {
  ...
  private DiscountPolicy discountPolicy;

  public Movie(String title, Duration runningTime, Money fee, DiscountPolicy discountPolicy) {
    ...
    this.discountPolicy = discountPolicy;
  }

  public Money calculateMovieFee(Screening screening) {
    return fee.minus(discountPolicy.calculateDiscountAmount(screening));
  }
}
```

런타인 의존성을 살펴보면 조금 다르다?

금액할인 정책을 적용 =>AmountDiscountPolicy의 인스턴스와 협력해야 한다.

비율 할인 정책을 적용하기 위해서는 PercentDiscountPolicy의 인스턴스와 협력해야한다.

코드를 작성하는 시점의 Movie 클래스는 AmountDiscountPolicy 클래스와 PercentDiscountPolicy클래스의 존재에 대해 전혀 알지 못하지만 실행 시점의 Movie 인스턴스는 AmountDiscount 인스턴스와 PercentDiscountPolicy인스턴스와 협력할 수 있어야 한다.

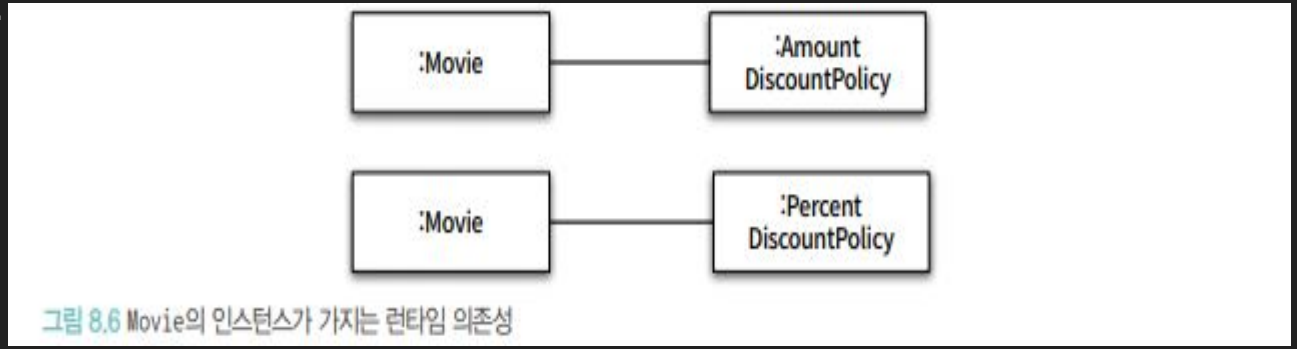

그림 8.6 Movie의 인스턴스가 가지는 런타임 의존성

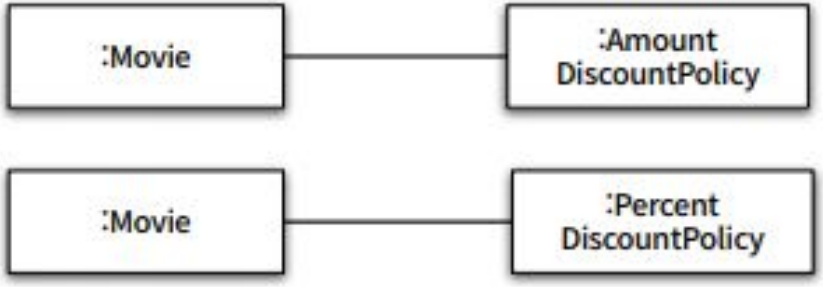

그림 8.6 Movie의 인스턴스가 가지는 런타임 의존성

만약 Movie클래스가 AmountDiscountPolicy 클래스에 데해서만 의존한다면?

-> PercentDiscountPolicy인스턴스와 협력하는 것은 불가능하다.

반대로 Movie클래스가 PercentDiscountPolicy 클래스에 대해서만 의존한다면
AmountDiscountPolicy인스턴스와 협력하는 것 역시 불가능.

Movie 의 인스턴스가 이 두 클래스의 인스턴스와 함께 협력할 수 있게 만드는 더 나은 방법은 Movie가 두 클래스 중 어떤 것도 알지 못하게 만드는 것이다.

대신 두 클래스 모두를 포괄하는 DiscountPolicy라는 추상 클래스에 의존하다록 만들고

이 컴파일타임 의존성을실행시에 PercentDiscountPolicy인스턴스나 AmountDiscountPolicy인스턴스에 대한 런타임 의존성으로 대체해야한다.

핵심 !

코드 작성 시점의 Movie클래스는 할인 정책을 구현한 두 클래스의 존재를 모르지만 실행 시점의 Movie객체는 두 클래스의 인스턴스와 협력할 수 있게 된다.

유연하고 재사용 가능한 설계를 창조하기 위해서는 동일한 소스코드 구조를 가지고 다양한 실행 구조를 만들 수 있어야 한다.

어떤 클래스의 인스턴스가 다양한 클래스의 인스턴스와 협력하기 위해서는 협력할 인스턴스의 구체적인 클래스를 알아서는 안 된다.

실제로 협력할 객체가 어떤 것인지는 런타임에 해결해야 한다.

클래스가 협력할 객체의 클래스를 명시적으로 드러내고 있다면 다른 클래스의 인스턴스와 협력할 가능성 자체가 없어진다. 컴파일타임 구조와 런타임 구조 사이의 거리가 멀면 멀수록 설계가 유연해지고 재사용 가능해진다.

컨텍스트 독립성

클래스는 자신과 협력할 객체의 구체적인 클래스에 대해 알아서는 안된다.

구체적인 클래스를 알면 알수록 그 클래스가 사용되는 특정한 문맥에 강하게 결합되기 때문이다.

구체 클래스에 대해 의존하는 것은 클래스의 인스턴스가 어떤 문맥에서 사용될 것인지를 구체적으로 명시하는 것과 같다. Movie클래스안에 PercentDiscountPolicy클래스에 대한 컴파일타임 의존성을 명시적으로 표현하는 것은 Movie가 비율할인 정책이 적용된 영화의 요금을 계산하는 문맥에서 사용될 것이라는 것을 가정하는 것이다.

Move 클래스에 추상클래스인 DiscountPolicy에 대한 컴파일 타임 의존성을 명시하는 것은 Movie가 할인 정책에 따라 요금을 계산하지만 구체적으로 어떤 정책을 따르는지는 결정하지 않았다고 선언하는 것이다.

이 경우 구체적인 문맥은 컴파일타임 의존성을 어떤 런타임 의존성으로 대체하느냐에 따라 달라질것이다.

====> 클래스가 특정 문맥에 강하게 결합될 수록 다른 문맥에서 사용하기는 더 어려워진다. 클래스가 사용될 특정한 문맥에 대해 최소한의 가정만으로 이뤄져 있따면 다른 문맥에서 재사용하기가 더 수월해진다.

이것이 컨텍스트 독립성이다.

설계가 유연해지기 위해서는 가능한 한 자신이 실행될 컨텍스트에 대한 구체적인 정보를 최대한 적게 알아야 한다.

컨텍스트에 대한 정보가 적으면 적을수록 더 다양한 컨텍스트에서 재사용될 수 있기 때문이다.

결과적으로 설계는 더 유연해지고 변경에 탄력적으로 대응할 수 있게 된다.

- 컨텍스트 독릭접이라는 말은 각 객체가 해당 객체를 실행하는 시스템에 관해 아무것도 알지 못한다는 의미이다.
- 이렇게 되면 행위의 단위를 가지고 새로운 상황에 적용할 수 있다.
- 컨텍스트 독립성을 따르면 다양한 컨텍스트에 적용할 수 있는 응집력이 있는 객체를 만들 수 있고 객체 구성 방법을 재설정해서 변경 가능한 시스템으로 나아갈 수 있다.

의존성 해결하기

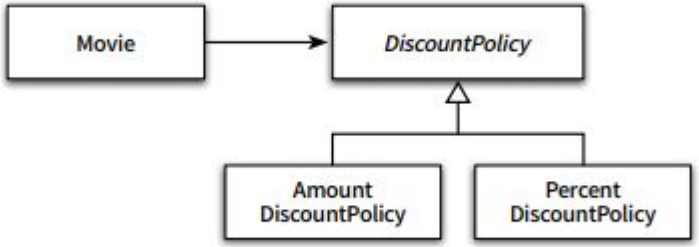

그림 8.5 코드 작성 시점의 Movie와 DiscountPolicy 사이의 의존성

컴파일 타임 의존성은 구체적인 런타임 의존성으로 대체되어야 한다.

Movie클래스는 DiscountPolicy 클래스에 의존한다.

이것은 컴파일타임 의존성이다.

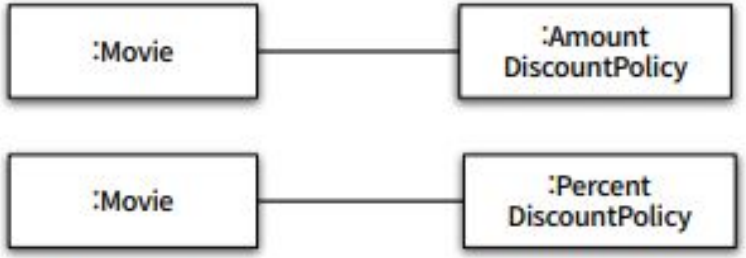

그림 8.6 Movie의 인스턴스가 가지는 런타임 의존성

Movie 인스턴스는 PercentDIscountPolicy 인스턴스나 AmountDiscountPolicy 인스턴스 중 하나에 의존한다.

이것은 Movie클래스와 DiscountPolicy 클래스 사이에 존재하는 컴파일 타임 의존성이 Movie인스턴스와 PercentDiscountPolicy 인스턴스 사이의 런타임 의존성이나 Movie 인스턴스와 AmountDiscountPolicy인스턴스 사이의 런타임 의존성으로 교체되어야 한다는 것을 의미

의존성해결하기

컴파일타임 의존성을 실행 컨텍스트에 맞는 적절한 런타임 의존성으로 교체하는 것을 의존성 해결 이라고 부른다.

- 객체를 생성하는 시점에 생성자를 통해 의존성 해결
- 객체 생성 후 setter 메서드를 통해 의존성 해결
- 메서드 실행 시 인자를 이용해 의존성 해결

예를 들어 어떤 영화 요금 계산에 금액 할인 정책을 적용하고 싶다고 가정해보자.

다음과 같이 Movie 객체를 생성할 때 AmountDiscountPolicy의 인스턴스를 Movie의 생성자에 인자로 전달하면 된다.

```
Movie avatar = new Movie("아바타",
    Duration.ofMinutes(120),
    Money.wons(10000),
    new AmountDiscountPolicy(...));
```

Movie의 생성자에 PercentDiscountPolicy의 인스턴스를 전달하면 비율 할인 정책에 따라 요금을 계산하게 될 것이다.

```
Movie starWars = new Movie("스타워즈",
    Duration.ofMinutes(180),
    Money.wons(11000),
    new PercentDiscountPolicy(...));
```

이를 위해 Movie 클래스는 PercentDiscountPolicy 인스턴스를 전달하면 비율 할인 정책에 따라 요금을 계산하게 될 것이다.

```
Movie starWars = new Movie("스타워즈",
    Duration.ofMinutes(180),
    Money.wons(11000),
    new PercentDiscountPolicy(...));
```

이를 위해 Movie클래스는

PercentDiscountPolicy인스턴스와 AmountDiscountPolicy 인스턴스 모두를 선택적으로 전달 받을 수 있도록 이 두 클래스의 부모 클래스인 DiscountPolicy 타입의 인자를 받는 생성자를 정의한다.

```
public class Movie {
  public Movie(String title, Duration runningTime, Money fee, DiscountPolicy discountPolicy) {
    ...
    this.discountPolicy = discountPolicy;
  }
}
```

Movie 의 인스턴스를 생성한 후에 메서드를 이용해 의존성을 해결하는 방법.

```
Movie avatar = new Movie(...));
avatar.setDiscountPolicy(new AmountDiscountPolicy(...));
```

이경우 Movie 인스턴스가 생성된 이후에도 DiscountPolicy를 설정할 수 있는 setter 메서드를 제공해야 한다.

```
public class Movie {
  public void setDiscountPolicy (DiscountPolicy discountPolicy) {
    this.discountPolicy = discountPolicy;
  }
}
```

setter 메서드를 이용하는 방식은 객체를 생성한 후에도 의존하고 있는 대상을 변경할 수 있는 가능성을 열어놓고 싶은 경우에 유용하다.

```
Movie avatar = new Movie(...));
avatar.setDiscountPolicy(new AmountDiscountPolicy(...));
...
avatar.setDiscountPolicy(new PercentDiscountPolicy(...));
```

setter 메서드를 이용하는 방법은 실행 시점에 의존 대상을 변경할 수 있기 때문에 설계를 좀 더 유연하게 만들 수 있다.

단점은 객체가 생성된 후에 협력에 필요한 의존 대상을 설정하기 때문에 객체를 생성하고 의존 대상을 설정하기 전까지는 객체의 상태가 불완전할수 있다는 점이다.

아래 코드에서 처럼 setter 메서드를 이용해 인스턴스 변수를 설정하기 위해내부적으로 해당 인스턴스 변수를 사용하는 코드를 실행하면 NullPointException 예외가 발생할 것이다.

```
Movie avatar = new Movie(...));
avatar.calculateFee(...);          // 예외 발생
avatar.setDiscountPolicy(new AmountDiscountPolicy(...));
```

더 좋은 방법은 생성자 방식과 setter 방식을 혼합하는 것이다.

항상 객체를 생성할 때 의존성을 해결해서 완전한 상태의 객체를 생성한 후 필요에 따라 setter 메서드를 이용해 의존 대상을 변경할 수 있게 할 수 있다.

이 방법은 시스템의 상태를 안정적으로 유지하면서 유연성을 향상 시킬 수 있어서

의존성 해결을 위해 가장 선호되는 방법이다.

```
Movie avatar = new Movie(..., new PercentDiscountPolicy(...));
...
avatar.setDiscountPolicy(new AmountDiscountPolicy(...));
```

Moive가 항상 할인 정책을 알 필요까지는 없고 가격을 계산할때만 일시적으로 알아도 무방하다면 메서드의 인자를 이용해 의존성을 해결할 수도 있다.

```
public class Movie {
  public Money calculateMovieFee(Screening screening, DiscountPolicy discountPolicy) {
      return fee.minus(discountPolicy.calculateDiscountAmount(screening));
    }
  }
```

메서드 인자를 사용하는 방식은 협력 대상에 대해 지속적으로 의존관계를 맺을 필요 없이 메소드가 실행되는 동안만 일시적으로 의존 관계가 존재해도 무방하거나 메서드가 실행될때마다 의존 대상이 매번 달라져야 하는 경우에 유용하다.

하지만 클래스의 메서드를 호출하는 대부분의 경우에 매번 동일한 객체를 인자로 전달하고 있다면 생성자를 이용하는 방식이나 setter 메서드를 이용해 의존성을 지속적으로 유지하는 방식으로 변경하는 것이 좋다.

유연한 설계

의존성과 결합도

- 객체지향 패러다임의 근간은 협력이다.
- 객체들이 협력갛기 위해서는 서로의 존재와 수행 가능한 책임을 알아야한다.
- 이런 지식들이 객체 사이의 의존성을 낳는다.
- 모든 의존성이 나쁜 것은 아니다.
- 의존성은 객체들의 협력을 가능하게 만드는 매개체라는 관점에서는 바람직한 것이다.
- 과하면 문제가 될 수 있다.

Movie가 비율 할인 정책을 구현하는 PercentDiscountPolicy에 직접 의존한다면

```
public class Movie {
  ...
  private PercentDiscountPolicy percentDiscountPolicy;

  public Movie(String title, Duration runningTime, Money fee,
      PercentDiscountPolicy percentDiscountPolicy) {
    this.percentDiscountPolicy = percentDiscountPolicy;
  }

  public Money calculateMovieFee(Screening screening) {
    return fee.minus(percentDiscountPolicy.calculateDiscountAmount(screening));
  }
}
```

비율 할인 정책을 적용하기 위해 Move가 PercentDiscountPolicy에 의존하고 있따는 사실을 코드를 통해 명시적으로 드러낸다.

Movie와 PercentDiscountPolicy사이에 의존성이 존재하는 것은 문제가 아니다.

오히려 이 의존성이 객체 사이의 협력을 가능하게 만들기 때문에 존재 자체는 바람직하다.

문제는 의존성 존재가아니라 의존성의 정도이다.

Movie를 PercentDisocountPolicy라는 구체적인 클래스에 의존하게 만들었기 때문에 다른 종류의 할인 정책이 필요한 문맥에서 Movie를 사용할 수 있는 가능성을 없앤다.

해결 방법은 의존성을 바람직하게 만드는 것이다.

Movie가 협력하고 싶은 대상이 반드시 PercentDiscountPolicy의 인스턴스일 필요는 없다.

Movie의 입장에서는 객체의 클래를 고정할 필요가 없다.

자신이 전송하는 calculateDiscountAmount 메시지를 이해할 수 있고 할인된 요금을 계산할 수만 있다면 어떤 타입의 객체와 협력하더라도 상관이 없다.

추상클래스인 DiscountPolicy는 calculateDiscountAmount 메시지를 이해할 수 있는 타입을 정의함으로써 이 문제를 해결한다.

AmountDiscountPolicy클래스와 PercentDiscountPolicy클래스가 DiscountPolicy를 상속받고 Movie클래스는 오직 DIscountPolicy에만 의존하도록 만듦으로써 DIscountPolicy클래스에 대한 컴파일 타임 의존성을 AmountDiscountPolicy인스턴스와 PercentDiscountPolicy인스턴스에 대한 런타임 의존성을 대체할 수 있다.

바람직한 의존성은 재사용성과 관련이 있다.

어떤 의존성이 다양한 환경에서 클래스를 재사용할 수 없도록 제한한다면

그 의존성은 바람직하지 못한 것이다.

어떤 의존성이 다양한 환경에서 재사용할 수 있다면 그 의존성은 바람직한것이다.

컨텍스트에 독립적인 의존성은 바람직한 의존성이고 특정한 컨텍스트에 강하게 결합된 의존성은 바람직하지 못한 의존성이다.

특정한 컨텍스트에 강하게 의존하는 클래스를 다른 컨텍스트에서 재사용할 수 있는 유일한 방법은 구현을 변경하는 것 뿐이다.

결합도 : 어떤 두 요소 사이에 존재하는 의존성이 바람직할 때 두 요소의 사이를

느슨한 결합도 또는 약합 결합도

혹은

단단할 결합도 또는 강한결합도를 가진다고 말한다.

일반적으로 의존성과 결합도를 동의어로 사용하지만 , 다른 관점에서 관계의 특성을 설명하는 용어다.

의존성은 두 요소 사이에 관계 유무를 설명한다. 따라서 의존성의 관점에서 의존성이 존재한다 또는 의존성이 존재하지 않는다라고 표현해야 한다.

그에 반해 결합도는 두 요소 사이에 존재하는 의존성의 정도를 상대적으로 표현한다.

따라서 결합도의 관점에서는 결합도가 강하다 또는 결합도가 느슨하다 라고 표현한다.

지식이 결합을 낳는다?

서로에 대해 알고 있는 지식의 양이 결합도를 결정한다.

이제 지식이라는 관점에서 결합도를 설명해보자.

Movie클래스가 PercentDiscountPolicy 클래스에 직접 의존한다고 가정하자.

이 경우 Movie는 협력 할 객체가 비율 할인 정책에 따라 할인 요금을 계산할 것이라는 것을 알고 있다.

반면 Movie 클래스가 추상 클래스인 DiscountPolicy 클래스에 의존하는 경우에는 구체적인 계산 방법은 알 필요가 없다.

그저 할인 요금을 계산한다는 사실만 알고 있다.

반면 Movie 클래스가 추상 클래스인 DiscountPolicy클래스에 의존하는 경우에는 구체적인 계산 방법은 알 필요가 없다.

그저 할인 요금을 계산한다는 사실만 알고 있을 뿐이다.

따라서 Movie가 PercentDiscountPolicy에 의존하는 것보다 DiscountPolicy에 의존하는 경우 알아야하는 지식의 양이 적기 때문에 결합도가 느슨해지는 것이다.

더 많이 알수록 더 많이 결합된다.

더 많이 알고 있다는 것은 더 적은 컨텍스트에서 재사용 가능하다는 것을 의미한다.

기존지식에 어울리지 않는 컨텍스트에서 클래스의 인스턴스를 사용하기 위해서 할 수 있는 유일한 방법은 클래스를 수정하는 것뿐이다.

그러니 적게 알아야 한다. 결합도를 느슨하게 만들기 위해서는 협력하는 대상에 대해 필요한 정보 외에는 최대한 감추는 것이 중요하다.

이 목적을 달성할 수 있는 가장 효과적인 방법은? 추상화

추상화에 의존하라

추상화란 어떤 양사, 세부사항, 구조를 좀더 명확하게 이해하기 위해 특정 절차나 물체를 의도적으로 생략하거나 감춤으로써 복잡도를 극복하는 방법이다.

DiscountPolicy 클래스는 PercentDiscountPolicy 클래스가 비율 할인 정책에 따라 할인 요금을 계산한다는 사실을 숨겨주기 때문에 PercentDiscountPolicy의 추상화다. 따라서 Movie 클래스의 관점에서 협력을 위해 알아야 하는 지식의 양은 PercentDiscountPolicy보다 DiscountPolicy 클래스가 더 적다. Movie와 DiscountPolicy 사이의 결합도가 더 느슨한 이유는 Movie가 구체적인 대상이 아닌 추상화에 의존하기 때문이다.

일반적으로 추상화와 결합도의 관점에서 의존 대상을 다음과 같이 구분하는 것이 유용하다. 목록에서 아래쪽으로 갈수록 클라이언트가 알아야 하는 지식의 양이 적어지기 때문에 결합도가 느슨해진다.

- 구체 클래스 의존성(concrete class dependency)
- 추상 클래스 의존성 (abstract class dependency)
- 인터페이스 의존성(interface dependency)

  구체 클래스에 비해 추상 클래스는 메서드의 내부 구현과 자식 클래스의 종류에 대한 지식을 클라이언트에게 숨길 수 있다.

  따라서 클라이언트가 알아야하는 지식의 양이 더 적기 때문에 구체 클래스보다 추상클래스에 의존하는 것이 결합도가 더 낮다.

  하지만 추상클래스의 클라이언트는 여전히 협력하는 대상이 속한 클래스 상속 계층이 무엇인지에 대해서는 알고 있어야 한다.

인터페이스에 의존하면 상속 계층을 모르더라도 협력이 가능해진다.

인터페이스 의존성은 협력하는 객체가 어떤 메시지를 수신할 수 잇는지에 대한 지식만을 남기기 때문에 추상 클래스의 의존성보다 결합도르 낮다.

이것은 다양한 클래스 상속 계층에 속한 객체들이 동일한 메시지를 수신할 수 있도록 컨텍스트를 확장하는 것을 가능하게 한다.

여기서 중요한 것은 실행 컨텍스트에 대해 알아야 하는 정보를 줄일수록 결합도가 낮아진다는 것이다. 결합도를 느슨하게 만들기 위해서는 구체적은 클래스보다 추상 클래스에 추상크랠스보다 인터페이스에 의존하도록 만드는것이 효과적이다.

대상이 더 추상적일 수록 결합도는 더 낮아진다.

명시적인 의존성

```
public class Movie {
  ...
  private DiscountPolicy discountPolicy;

  public Movie(String title, Duration runningTime, Money fee) {
    ...
    this.discountPolicy = new AmountDiscountPolicy(...);
  }
}
```

- Move의 인스턴 변수인 discountPolicy는 추상 클래스인 discountPolicy타입으러 선언되어 있다. Movie는 추상화에 의존하기 때문에 이 코드는 유연하고 재사용 가능할 것처럼 보인다.
- 하지만 안타깝게도 생성자를 보면 그렇지 않다. 구체클래스인 AmountDiscountPolicy의 인스턴스를 직접 생성해서 대입하고 있다.Move 는 추상클래스인 DIscountPolicy뿐만아니라 구체 클래스인 AmountDIcuntPolicy에도 의존하게 된다.

- 결합도를 느슨하게 만들기 위해서는 인스턴스 변수의 타입을 추상 클래스나 인터페이스로 선언하는 것만으로는 부족하다.
- 클래스 안에서 구체클래스에 대한 모든 의존성을 제거해야하만 한다.

하지만 런타임에 movie 는 구체 클래스의 인스턴스와 협력해야 하기 때문에 Movie의 인스턴스가 AmountDiscountPolicy의 인스턴스 인지 PercentDIscountPolicy의 인스턴스인지를 알려줄 수 있는 방법이 필요하다.

다시 말해서 Movie의 의존성을 해결해 줄수 있는 방법이 필요한 것이다.

의존성을 해결하는 방법에는

1.생성자

2. setter메서드

3.메서드 인자를 사용

여기서 트릭은 인스턴스 변수의 타입은 추상 클래스나 인터페이스로 정의하고 생성자 stter 메서드 메서드 인자로 의존성을 해결할 때는 추상 클래스를 상속받거나 인터페이스를 실체화한 구체 클래스를 전달하는 것이다.

```
public class Movie {
  ...
  private DiscountPolicy discountPolicy;

  public Movie(String title, Duration runningTime, Money fee, DiscountPolicy discountPolicy) {
    ...
    this.discountPolicy = discountPolicy;
  }
}
```

생성자를 사용해 의존성을 해결하는 경우이다.

앞의 코드와 다른 점이라면 생성자 안에서 인스턴스를 직접 생성하지 않고 생성자의 인자로 선언하고 있음을 알 수 있다.

인스턴스 변수의타입과 생성자의 인자타이봄두 추상클래스인DiscountPoloicy 로 선언되어 있다.

생성자의 인자가 추상 클래스 타입으로 선언됐기 때문에 이제 객체를 생성할 때 생성자의 인자로 DiscountPolicy의 자식 클래스 중 어떤 것이라도 전달가능하다.

Movie 인스턴스는 생성자의 인자로 전달된 인스턴스에 의존하게 된다.

의존성의 대상을 생성자의 인자로 전달받는 방법과 생성자 안에서 직접 생성하는 방법 사이의 가장 큰 차이점은 퍼블릭 인터페이스를 통해 할인 정책을 설정할 수 있는 방법의 제공 여부다. 생성자의 인자로 선언하는 방법은 Movie가 DIscountPolicy에 의존한다는 사실을 Movie의 퍼블릭 인터페이스에 드러내는 것이다.

이것은 setter 메서드를 사용하는 방식과 메서드 인자를 사용하는 방식의 경우에도 동일하다. 모든 경우에 의존성은 명시적으로 퍼블릭 인터페이스에 노출된다.

이를 명시적인 의존성이라고 부른다.

반면 Moive의 내부에서 AmountDiscountPolicy 의 인스턴스를 직접 생성하는 방식은 Movie가 DiscountPolicy에 의존한다는 사실을 감춘다.

다시 말해 의존성이 퍼블릭 인터페이스에 표현되지 않는다.

이를 숨겨진 의존성이라고 부른다.

의존성이 명시적이지 않으면 의존성을 파악하기 위해 내부 구현을 직접 살펴볼 수 밖에 없다.

커다란 클래스에 정의된 긴 메서드 내부 어딘가에서 인스턴스를 생성하는 코드를 파악하는 것은 어렵다.

의존성이 명시적이지 않으면 클래스를 다른 컨텍스트에 재사용하기 위해 내부 구현을 직접 변경해야한다는 것이다.

- 의존성은 명시적으로 표현되어야 한다.
  명시적인 의존성을 사용해야만 퍼블릭 인터페이스를 통해 컴파일타임 의존성을 적절한 런타임 의존성으로 교체할 수 있다.

new 는 해롭다

new 를 잘못사용하면 클래스 사이의 결합도가 극단적으로 높아진다.

결합도 측면에서 new가 해로운 이유는 크게 두가지다.

- new 연산자를 사용하기 위해서는 구체 클래스의 이름을 직접 기술해야한다.
  new 를 사용하는 클라이언트는 추상화가 아닌 구체 클래스에 의존할수밖에 없기때문에 결합도가 높아진다.
- new 연산자는 생성하려는 구체 클래스 뿐만아니라 어떤 인자를 이용해 클래스의 생성자를 호출해야 하는 지도 알아야한다.
  따라서 new 를 사용하면 클라이언트가 알아야하는 지식의 양이 늘어나기 때문에 결합도가 높아진다.

구체 클래스에 직접 의존하면 결합도가 높아진다는 사실을 기억하라.

결합도의 관점에서 구체 클래스는 협력자에게 너무 많은 지식을 알도록 강요한다.

여기에 new는 문제를 더 크게 만든다. 클라이언트는 구체 클래스를 생성하는데 어떤 정보가 필요한지에 대해서도 알아야 하기 때문이다.

AmoiuntDiscountPolicy의 인스턴스를 직접 생성하는 Movie클래스의 코드를 좀더 자세히 알아보자.

```
public class Movie {
  ...
  private DiscountPolicy discountPolicy;

  public Movie(String title, Duration runningTime, Money fee) {
    ...
    this.discountPolicy = new AmountDiscountPolicy(Money.wons(800),
                          new SequenceCondition(1),
                          new SequenceCondition(10),
                          new PeriodCondition(DayOfWeek.MONDAY,
                              LocalTime.of(10, 0), LocalTime.of(11, 59)),
                          new PeriodCondition(DayOfWeek.THURSDAY,
                              LocalTime.of(10, 0), LocalTime.of(20, 59))));
  }
}
```

Movie클래스가 AmountDiscountPolicy의 인스턴스르 생성하기 위해서는 생성자에 전달되는 인자를 알고 있어야한다.

이것은 Movie 클래스가 알아야 하는 지식의 양을 늘리기 떄문에 movie가 amountDiscountPolicy에게 더 강하게 결합되게 만든다.

엎친데 덮친 격으로 Movie가 AmountDiscountPolicy의 생성자에게 참조하는 두 구체 클래스인 SequenceCondition과 PeriodCondition에도 의존하도록 만든다.

그리고 다시 이 두 클래스의 인스턴스를 생성하는데 필요한 인자들의 정보에 대해서 movie를 결합시킨다.

결합도가 높으면 변경에 의해 영향을 받기 쉬워진다. Movie는 AmountDiscountPolicy의 생성자의 인자 목록이나 인자 순서를 바꾸는 경우에도 함께 변경될 수 있다. SequenceCondition과 PeriodCondition의 변경에도 영향을 받을 수 있다. Movie가 더 많은 것에 의존하면 의존할수록 점점 더 변경에 취약해진다. 이것이 높은 결합도를 피해야 하는 이유다.

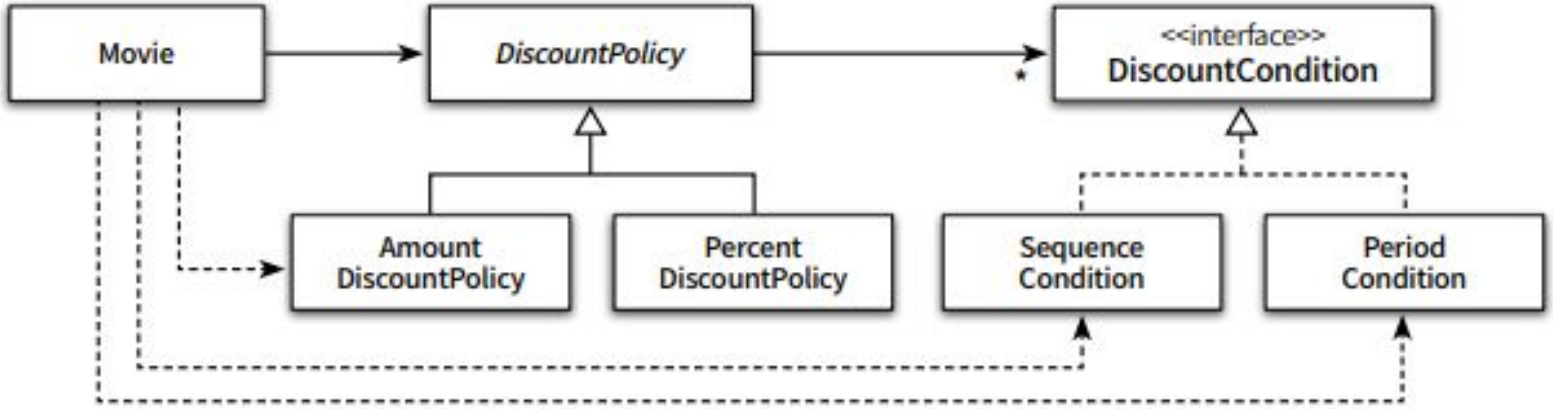

그림 8.8 new는 결합도를 높이기 때문에 해롭다

해결 방법은 인스턴스를 생성하는 로직과 생성성된 인스턴스를 사요하는 로직을 분리하는 것이다.

AmountDiscountPolicy를 사용하는 Movie는 인스턴스를 생성해서는 안된다.

단지 해당하는 인스턴스를 사용하기만 해야 한다.

이를 위해 Movie는 외부로부터 이미 생성된 AmountDiscountPolicy의 인스턴스를 전달받아야 한다.

외부에서 인스턴스를 전달받는 방법은 앞에서 살펴본 의존성 해결 방법과 동일하다.

생성자의 인자로 전달하거나 setter 메서드를 사영하거나 실행 시에 메서드의 인자로 전달하면 된다.
어떤 방법을 사용하건 Movie클래스에는 AmountDiscountPolicy의 인스턴스에 메시지를 전송하는 코드만 남아 있어야 한다.

생성자를 통해 외부의 인스턴스를 전달받아 의존성을 해결하는 Movie코드

```
public class Movie {
  ...
  private DiscountPolicy discountPolicy;

  public Movie(String title, Duration runningTime, Money fee, DiscountPolicy discountPolicy) {
    ...
    this.discountPolicy = discountPolicy;
  }
}
```

Movie는 AmountDIscountPolicy의 인스턴스를 직접 생성하지 않는다.
필요한 인스턴스를 생성자의 인자로 전달받아 내부의 인스턴스 변수에 할당한다.
Movie는 단지 메시지를 전송하는 단 하나의 일만 수행한다.
그렇다면 누가 AmountDiscountPolicy의 인스턴스를 생성하는가?
movie의 클라이언트가 처리한다. 이제 AmountDiscountPolicy의 인스턴스를 생성하는 책임은 Moive의 클라이언트로 옮겨지고 Movie는 AmountDiscountPolicy의 인스턴스를 사용하는 책임만 남는다..

```
Movie avatar = new Movie("아바타",
    Duration.ofMinutes(120),
    Money.wons(10000),
    new AmountDiscountPolicy(Money.wons(800),
        new SequenceCondition(1),
        new SequenceCondition(10),
        new PeriodCondition(DayOfWeek.MONDAY,
              LocalTime.of(10, 0), LocalTime.of(11, 59)),
        new PeriodCondition(DayOfWeek.THURSDAY,
              LocalTime.of(10, 0), LocalTime.of(20, 59))));
```

사용과 생성의 책임을 분리해서 Movie의 결합도를 낮추면 설계를 유연하게 만들 수 있다. ?movie의 생성자가 구체클래스인 AmountDiscountPolicy가 아니라 추상클래스인 DiscountPolicy를 인자로 받아들이도록 선언되어 있다.

생성의 책임을 클라이언트로 옮김으로써 이제 Movie는 DiscounPolicy의 모든 자식 클래스와 협력할 수있게 된다.

사용과 생성의 책임을 분리하고 의존성을 생성자에 명시적으로 드러내고

구체클래스가 아닌 추상 클래스에 의존하게 함으로써 설계를 유연하게 만들 수 있다. 그리고 그 출발은 객체를 생성하는 책임을 객체 내부가 아니라 클러이언트로 옮기는 것에서 시작했다는 점을 기억하라.

가끔은 생성해도 무방하다.

클래스 안에서 객체의 인스턴스를 직접 생성하는 방식이 유용한 경우도 있다.

주로 협력하는 기본 객체를 설정하고 싶은 경우가 여기에 속하낟.

예를들어 Movie가 대부분의 경우에는 AmountDiscountPolicy의 인스턴스와 협력하고 가끔쉭만 PercentDiscountPolicy의 인스턴스와 협력한다고 가정해보자.

이런 상황에서 모든 경우에 인스턴스를 생성하는 책임을 클라이언트로 옮긴다면 클라이언트들 사이에 중복코드가 늘어나고 Movie의 사용성도 나빠질 것이다.

이 문제를 해결하는 방법은 기본 객체를 생성하는 생성자를 추가하고

이 생성자에서 DiscountPolicy의 인스턴스를 인자로 받는 생성자를 체이닝하는 것이다.

```
public class Movie {
  private DiscountPolicy discountPolicy;

  public Movie(String title, Duration runningTime, Money fee) {
    this(title, runningTime, fee, new AmountDiscountPolicy(...));
  }

  public Movie(String title, Duration runningTime, Money fee, DiscountPolicy discountPolicy) {
    ...
    this.discountPolicy = discountPolicy;
  }
}
```

추가된 생성자 안에서 AmountDiscountPolicy 클래스의 인스턴스를 생성한다는 것을 알 수 있다.

첫번째 생성자의 내부에서 두번째 생성자를 호출한다는 것이다.

다시 말해 생성자가 체인처럼 연결되어 있다.

이제 클라이언트는 대부분의 경우에 추가된 간략한 생성자를 통해 AmountDiscountPolicy의 인스턴스와 협력하게 하면서도 컨텍스트에 적절한 DiscountPolicy의 인스턴스로 의존성을 교체할 수 있다.

이 방법은 메서드를 오버로딩 하는 경우에도 사용할 수 있다.

다음과 같이 DiscountPolicy의 인스턴스를 인자로 받는 메서드와 기본값을 생성하는 메서드를 함께 사용한다면 클레스의 사용성을 향상 시키면서도 다양한 컨텍스트를에서 유연하게 사용할 수 있는 여지를 제공할 수 있다.

```
public class Movie {
  public Money calculateMovieFee(Screening screening) {
    return calculateMovieFee(screening, new AmountDiscountPolicy(...));
  }

  public Money calculateMovieFee(Screening screening,
      DiscountPolicy discountPolicy) {
    return fee.minus(discountPolicy.calculateDiscountAmount(screening));
  }
}
```

설계가 트레이드 오프 활동이라는 것을 상기하자.

여기서 트레이드오프와 대상은 결합도와 사용성이다.

구체 클래스에 의존하게 되더라도 클래스의 사용성이 더 중요하다면 결합도를 높이는 방향으로 코드를 작성할 수 있다.

그럼에도 가급적 구체 클래스에 대한 의존성을 제거할 수 있는 방법을 찾아보기 바란다. 종종 모든 결합도가 모이는 새로운 클래스를 추가함으로써 사용성과 유연성이라는 두 마리 토끼를 잡을 수 있다는 경우도 있다.

표준 클래스에 대한 의존은 해롭지 않다.

의존성이 불편한 이유는 그것이 항상 변경에 대한 영향을 암시하기 때문.

변경될 확률이 거의 없는 클래스라면 의존성이 문제가 되지 않는다.

자바라면 JDK 에 포함된 표준 클래스가 이 부류에 속한다.

이런 클래스들에 대해서는 구체 클래스에 의존하거나 직접 인스턴스를 생성하더라도 문제가 없다.

JDK 표전 컬렉션 라이브러리에 속하는 ArrayList의 경우에는 직접 생성해서 대입하는 것이 일반적이다. ArrayList 의 코드가 수정될확률은 0에 가깝기 때문에 인스턴스를 직접 생성하더라도 문제가 되지 않는다.

비록 클래스를 직접 생성하더라도 가능한 한 추상적인 타입을 사용하는 것이 확장성 측면에서는 유리하다 위 코드에서 conditions 타입으로 인터페이스인 List를 사용한 것은 이 때문이다.

이렇게 하면 다양한 List 타입의 객체로 conditions를 대체할 수 있게 설계의 유연성을 높일 수 있다.

따라서 의존성에 의한 영향이 적은 경우에도 추상화에 의존하고 의존성을 명시적으로 드러내는 것은 좋은 설계 습관이다.

```
public abstract class DiscountPolicy {
  private List<DiscountCondition> conditions = new ArrayList<>();

  public void switchConditions(List<DiscountCondition> conditions) {
    this.conditions = conditions;
  }
}
```

컨텍스트 확장하기

실제로 Movie 가 유연하다는 사실을 입증하기 위해 지금까지와는 다른 컨텍스트에서 movie를 확장해서 재사용하는 두 가지 예를 살펴보겠다.

하나는 할인 혜택을 제공하지 않는 영화의 경우이고,

다른 하나는 다수의 할인 정책을 중복해서 적용하는 영화를 처리하는 경우다.

첫번째는 할인혜택을 제공하지 않는 영화의 예매 요금을 계산하는 경우다.

쉽게 생각할 수 있는 방법은 discountPolicy에 어떤 객체도 할당하지 않는것이다.

다음과 같이 discountPolicy에 null을 할당하고 실제로 존재하는지 판단하는 방법을 사용할 수 있다.

```
public class Movie {
  public Movie(String title, Duration runningTime, Money fee) {
    this(title, runningTime, fee, null);
  }

  public Movie(String title, Duration runningTime, Money fee, DiscountPolicy discountPolicy) {
    ...
    this.discountPolicy = discountPolicy;
  }

  public Money calculateMovieFee(Screening screening) {
    if (discountPolicy == null) {
      return fee;
    }

    return fee.minus(discountPolicy.calculateDiscountAmount(screening));
  }
}
```

앞에서 설명한 생성자 체이닝 기법으로 기본값을 null을 할당했다.

discountPolicy의 값이 null인 경우에는 할인정책을 적용해서는안되기 때문에 calculateMovieFee메서드 내부에서 discountPolicy의 값이null인지 여부를 체크한다.

동작은 하지만 문제가 있다.

지금까지의 movie와 discountPolicy사이의 협력 방식에 어긋나는 예외 케이스가 추가된 것이다. 그리고 이 예외 케이스를 처리하기 위해서 Movie의 내부 코드를 직접 수정해야 했다.

어던 경우든 코드 내부를 직접 수정하는 것은 버그의 발생 가능성을 높인다.

해결책은 할인 정책이 존재하지 않는다는 사실을 예외 케이스로 처리하지 말고 Movie 와 discountPolicy가 협력하던 방식을 따르도록 만든다.

할인 전책이 존재하지 않는다시실을 할인 정책의 한 종류로 간주한다.

할인할 금액으로 0을 반환하는 NoneDiscountPolicy 클래스를 추가하고 DiscountPolicy의 자식 클래스로 만드는것이다.

```
public class NoneDiscountPolicy extends DiscountPolicy {
  @Override
  protected Money getDiscountAmount(Screening Screening) {
    return Money.ZERO;
  }
}
```

간단히 NoneDiscountPolicy의 인스턴스를 Movie의 생성자에 전달하면 된다.

```
Movie avatar = new Movie("아바타",
    Duration.ofMinutes(120),
    Money.wons(10000),
    new NoneDiscountPolicy());
```

두번째 예는 중복 적용이 가능한 할인 정책을 구현하는 것이다.

중복할인이란 금액 할인 정책과 비율 할인 정책을 혼합해서 적용할 수 있는 정책이다.

할인 정책을 중복해서 적용하기 위해 Movie가 하나이상의 DiscountPolicy와 협력할 수 있어야 한다.

가장 간단하게 구현할 수 있는 방법은 Movie 가 DiscountPolicy의 인스턴스들로 구성된 List를 인스턴스 변수로 갖게 하는 것이다. 하지만 이 방법은 중복 할인정책을 구현하기 위해 기존의 할인 정책방식과는 다른 예외 케이스를 추가하여 만든다.

이문제 역시 NoneDiscountPolicy와 같은 방법을 사용해서 해결할 수 있다

중복 할인 정책을 할인 정책의 한가지로 간주하는 것이다.

중복 할인 정책을 구현하는 OverlappedDiscountPolicy를 DiscountPolicy의 자식클래스로 만들면 기존의 Movie와 DiscountPolicy사이의 협력 방식을 수정하지 않고도 여러개의 할인 정책을 적용할 수 있다.

```
public class OverlappedDiscountPolicy extends DiscountPolicy {
  private List<DiscountPolicy> discountPolicies = new ArrayList<>();

  public OverlappedDiscountPolicy(DiscountPolicy ... discountPolicies) {
    this. discountPolicies = Arrays.asList(discountPolicies);
  }

  @Override
  protected Money getDiscountAmount(Screening screening) {
    Money result = Money.ZERO;
    for(DiscountPolicy each : discountPolicies) {
      result = result.plus(each.calculateDiscountAmount(screening));
    }
    return result;
  }
}
```

```
Movie avatar = new Movie("아바타",
    Duration.ofMinutes(120),
    Money.wons(10000),
    new OverlappedDiscountPolicy(
        new AmountDiscountPolicy(...),
        new PercentDiscountPolicy(...)));
```

Movie 를 수정하지 않고도 할인 정책을 적용지 않는 새로운 기능을 추가하는 것이 얼마나 간단한지를 잘 보여준다.

우리는 단지 원하는 기능을 구현한 DiscountPoliy의 자식 클래스를 추가하고 이 클래스의 인스턴스를 Movie에 전달하기만 하면 된다.

Movie가 협력해야 하는 객체를 변경하는것만으로도 Movie를 새로운 컨텍스트에서 재사용할 수 있기 때문에 Movie는 유연하고 재사용 가능하다.

Movie가 DiscountPolicy라는 추상화에 의존하고 생성자를 통해 DiscountPolicy에 대한 의존성을 명시적으로 드러냈으며 new와 같이 구체 클래스를 직접적으로 다뤄야하는 책임을 Movie외부로 옮겨기 때문이다. 우리는 Movie가 의존하느 추상화인 DiscountPolicy클래스에 자식 클래스를 추가함으로써 Moive가 사용될 컨텍스트를 확장할 수 있었다. 결합도를 낮춤으로써 얻게 되는 컨텍스트의 확장이라는 개념이 유연하고 재사용 가능한 설계를 만드는 핵심이다.

조합 가능한 행동

어떤 객체와 협력하느냐에 따라 객체의 행동이 달라지는 것은 유연하고 재사용 가능한 설계가 가진 특징이다. 유연하고 재사용 가능한 설계는 응집도 높은 책임들을 가진 작은 개겢들을 다양한 방식으로 연결함으로써 어플리케이션의 기능을 쉽게 확장할 수 있다.

유연하고 재사용 가능한 설계는 객체가 어떻게(how)하는지를 장황하게 나열하지 않고도 객체들의 조합을 통해 무엇(what)을 하는지를 표현하는 클래스들로 구성된다.

따라서 클래스의 인스턴스를 생성하는 코드를 보는 것만으로 객체가 어떤 일을하는지를 쉽게 파악할 수 있다.

코드에 드러난 로직을 해석할 필요 없이 객체가 어떤 객체와 연결되었는지만을 보고 객체의 행동을 쉽게 예상하고 이해할 수있기 때문이다.

선언적으로 객체의 행동을 정의할 수있다.

Movie를 생성하는 아래 코드를 주의 깊게 살펴보기 바란다. 이 코드를 읽는 것만으로도 첫 번째 상영, 10번째 상영, 월요일 10시부터 12시 사이 상영, 목요일 10시부터 21시 상영의 경우에는 800원을 할인해 준다는 사실을 쉽게 이해할 수 있다. 그리고 인자를 변경하는 것만으로도 새로운 할인 정책과 할인 조건을 적용할 수 있다는 것 역시 알 수 있을 것이다.

```
new Movie("아바타",
    Duration.ofMinutes(120),
    Money.wons(10000),
    new AmountDiscountPolicy(Money.wons(800),
        new SequenceCondition(1),
        new SequenceCondition(10),
        new PeriodCondition(DayOfWeek.MONDAY, LocalTime.of(10, 0), LocalTime.of(12, 0)),
        new PeriodCondition(DayOfWeek.THURSDAY, LocalTime.of(10, 0), LocalTime.of(21, 0))));
```

유연하고 재사용 가능한 설계는 작은 객체들의 행동을 조합함으로써 새로운 행동을 이끌어낼수 있는 설꼐다.

훌륭한 객체지향 설계란 객체가 어떻게 하는지를 표현하는 것이 아니라 객체들의 조합을 선언적으로 표현함으로써 객체들이 무엇을 하는지를 표현하는 설계다.

핵심은 의존성을 관리하는 것이다.