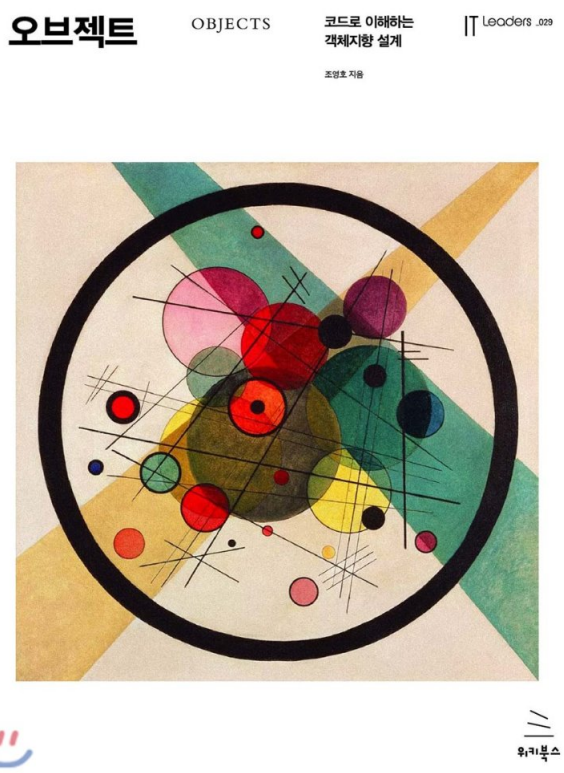


오브젝트 - 코드로 이해하는 객체지향 설계 를 정리한 자료입니다.

목차

Chapter 09 유연한 설계

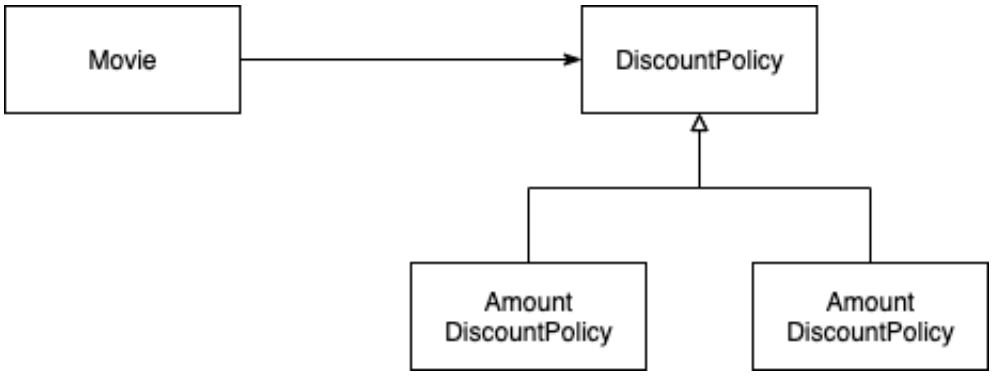
이번 장에서는 저번 장에서도 계속해서 설명했던 **유연한 설계를 만드는 기법들을 원칙으로 알아본다.**

원칙을 통해 기법들을 정리하면 머리에 더 오래 남으며 또렷하게 정리가 된다.

1 개방 - 폐쇄 원칙 OCP

유연한 설계의 대표적인 원칙이 바로 "개방 - 폐쇄 원칙"이다.

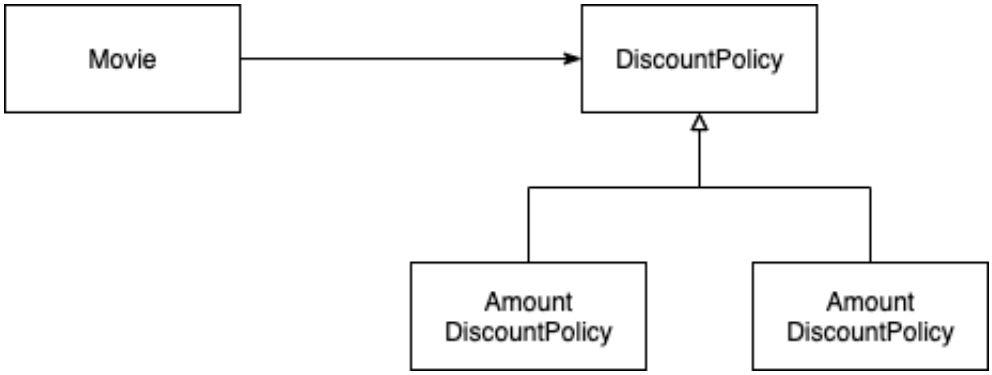
1-1 개방 - 폐쇄 원칙이란



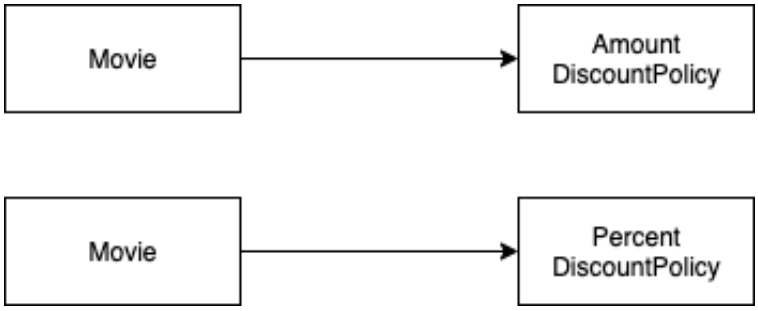
- 소프트웨어 개체 (클래스, 모듈, 함수 등등)는 **확장에 대해 열려 있어야 하고, 수정에 대해서는 닫혀 있어야 한다.**
    - **확장 -> Open** : 요구사항이 변경될 때 이 변경에 맞게 **새로운 '동작'을 추가해서** 기능을 **확장**할 수 있다.
    - **폐쇄 -> Closed** : 기존의 **'코드'**를 **수정하지 않고**도 **동작을 추가하거나 변경**할 수 있다.

1-2 컴파일타임 의존성을 고정시키고 런타임 의존성을 변경하라

사실 개방 - 폐쇄 원칙은 런타임 의존성과 컴파일타임 의존성에 관한 이야기이다.



- **런타임 의존성**
  - 실행시에 협력에 참여하는 객체들 사이의 관계



- **컴파일타임 의존성**
  - 코드에서 드러나는 클래스들 사이의 관계

변경을 하려면 전략을 만들어 주입해주기만 하면 된다

```
public class Movie {
  private DiscountPolicy discountPolicy; // 인터페이스 -> 다양한 환경에서 재사용할 수 있다.

  public Movie(..., DiscountPolicy discountPolicy) {
    this.discountPolicy = discountPolicy;
  }
}

new Movie(...,
          new PercentDiscountPolicy()); // 재사용 가능
```

- 변경을 하려면 `DiscountPolicy`의 구현체를 생성하여 주입해주기만 하면 된다.
  - 확장에 대해서는 열려 있고, 기존 코드는 수정할 필요 없이 새로운 클래스를 추가하는 것만으로 할인 정책을 확장할 수 있다.

1-3 추상화가 핵심이다

개방 - 폐쇄 원칙의 핵심은 추상화다

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

  public DiscountPolicy(DiscountCondition... conditions) {
    this.conditions = Arrays.asList(conditions);
  }

  public Money calculateDiscountAmount(Screening screening) {
    for(DiscountCondition each : conditions) {
      if(each.isSatisfiedBy(screening))
        return getDiscountAmount(screening);
    }
    return screening.getMovieFee();
  }

  abstract protected Monet getDiscountAmount(Screening screening); // 추상 메서드
-> 핵심 로직 (전략)
}
```

- 추상화란
  - 핵심적인 부분만 남기고 불필요한 부분은 생략함으로써 복잡성을 극복하는 기법.
  - 추상화과정을 거치면 문맥이 바뀌더라도 변하지 않는 부분만 남게 되고 문맥에 따라 변하는 부분은 생략된다.

전략패턴?

- 공통적인 부분은 문맥이 바뀌더라도 변하지 않아야 한다 -> **핵심 로직과 부가 로직을 분리** -> 전략(핵심 로직)을 컨텍스트 (부가 로직)에 주입한다.

추상화만 했다고 개방 - 폐쇄 원칙이 아니다 변하는 것이 무엇인지 이해해야 한다

- 추상화를 했다고 해서 모든 수정에 대해 설계가 폐쇄되는 것은 아니다.
  - 변하는 것과 변하지 않는 것이 무엇인지를 이해하고 이를 추상화의 목적으로 삼아야한다.
  - 즉, **변하는 것(핵심 로직)과 변하지 않는 것(부가 로직)을 분리해야 한다.**
- 추상화가 수정에 대해 닫혀 있을 수 있는 이유가 바로 이렇게 변하는 것과 변하지 않는 것을 결정하였기 때문이다.

2 생성 사용 분리

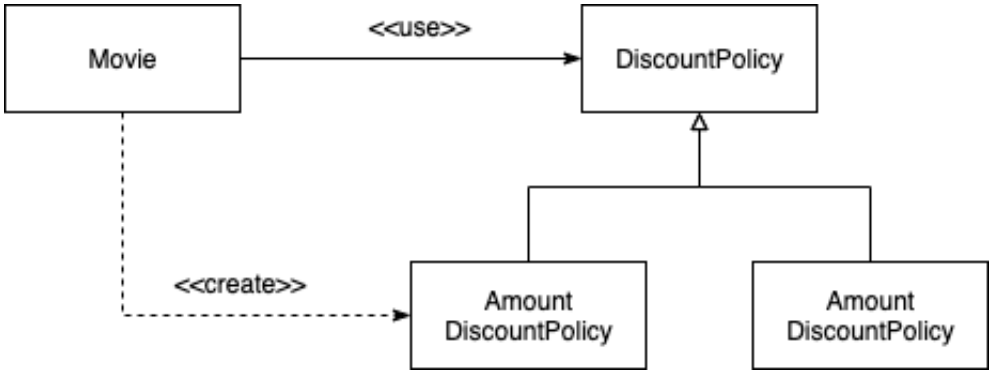
다른 객체에 의존하기 위해서는 객체를 생성해야 한다. 즉, **객체 생성을 피할 수 없다.**

어딘가에서는 반드시 객체를 생성해야 한다. 문제는 객체 생성이 아니다. **부적절한 곳에서 객체를 생성한다는 것이 문제다.**

2-1 생성과 사용을 분리하라

소프트웨어 시스템은 시작 단계와 실행 단계를 분리해야 한다.

생성과 사용의 책임을 함께 맡고 있는 객체



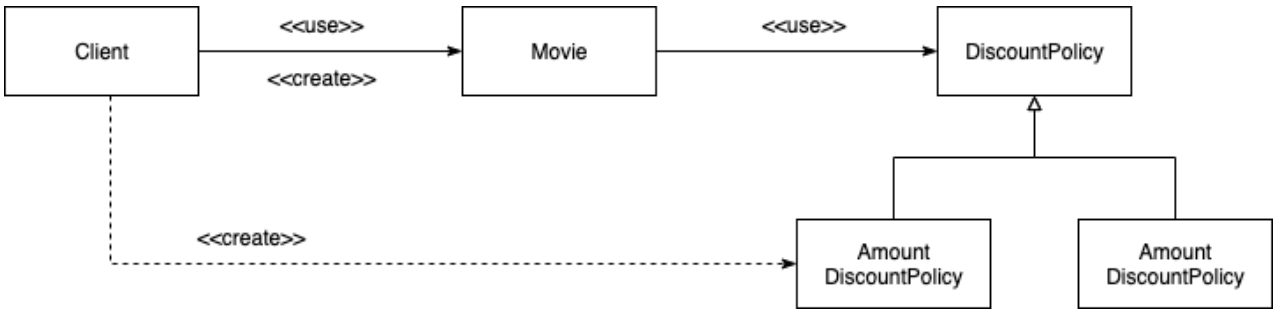

```
public class Movie {
  private DiscountPolicy discountPolicy;

  public Movie(String title, Duration runningTime, Monie fee) {
    ...
    this.discountPolicy = new AmountDiscountPolicy(...); // 생성 책임
  }

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

- `Movie`의 책임
  - 생성
  - 사용

생성과 사용의 책임을 분리한 객체



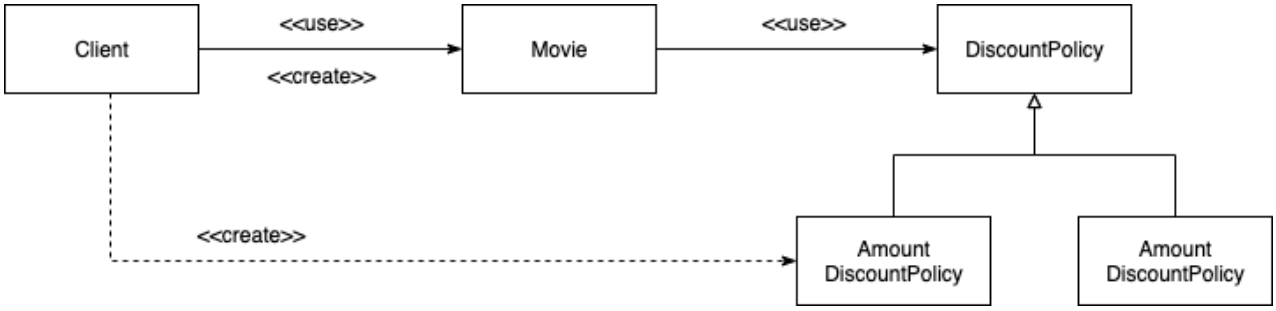

```
public class Client { // 클라이언트
  public Money getAvatarFee() {
    // 생성책임
    Movie avatar = new Movie("아바타",
                             Duration.ofMinutes(120),
                             Money.wons(1000),
                             new PercentDiscountPolicy(...));

    return avatar.getFee();
  }
}
```

- 생성 책임 - Client
  - 생성은 `Client`에게 위임한다
- 사용 책임 - Movie
  - `Movie`는 `DiscountPolicy`의 사용에만 집중한다

2-2 FACTORY 추가하기

만약 Client도 생성과 사용을 분리해야 한다면?

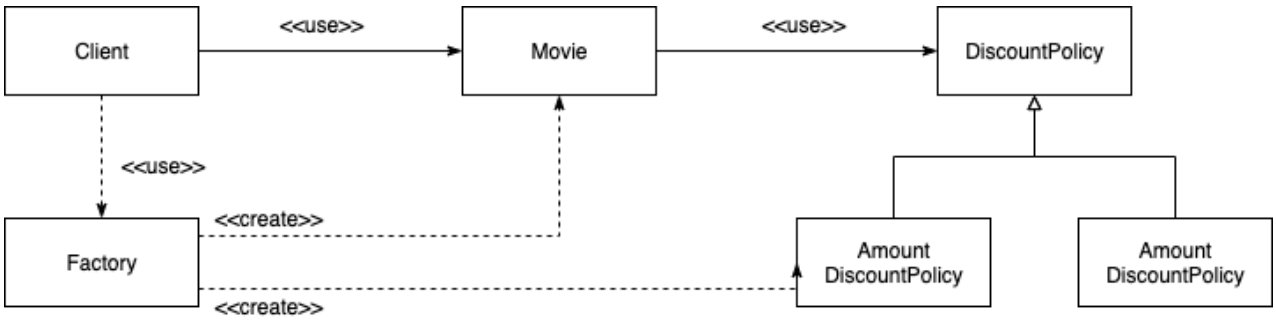


- `Client`의 책임
  - 생성 - `new Movie(...)`
  - 사용 - `return avatar.getFee()`

- `Client`도 생성과 사용이라는 두 가지의 책임을 가지고 있으며, **특정 컨텍스트에 종속적이다.**
  - 컨텍스트 1 - `AmountDiscountPolicy`를 만들어 주입 -> 종속
  - 컨텍스트 2 - `PercentDiscountPolicy`를 만들어 주입 -> 종속
- 만약 `Client`도 생성과 사용을 분리하고 싶다면??
  - FACTORY를 만들어주자.

FACTORY를 만들자

- FACTORY란?
  - 생성과 사용을 분리하기 위해 **객체 생성에 특화된 객체**를 FACTORY라고 한다.




```
// 생성 책임 (Movie, AmountDiscountPolicy를 생성)
public class Factory {
  public Movie createAvatarMovie() {
    return new Movie("아바타",
                     Duration.ofMinute(120),
                     Money.wons(10000),
                     new AmountDiscountPolicy(...));
  }
}
```

```
public class Client {
  private Factory factory;

  public Client(Factory factory) {
    this.factory = factory;
  }

  public Money getAvatarFee() {
    Movie avatar = factory.createAvatarMovie(); // 생성 책임은 FACTORY에게 위임
    return avatar.getFee(); // 사용 책임
  }
}
```

- Movie와 AmountDiscountPolicy 생성에 대한 책임을 모두 FACTORY가 가지게 되었다.

- **이제 Client와 Movie는 사용 책임만을 수행하면 된다.**
- **의존성에 변경이 필요하다면 FACTORY의 코드만 변경해주면 된다.**

2-3 순수한 가공물에게 책임 할당하기

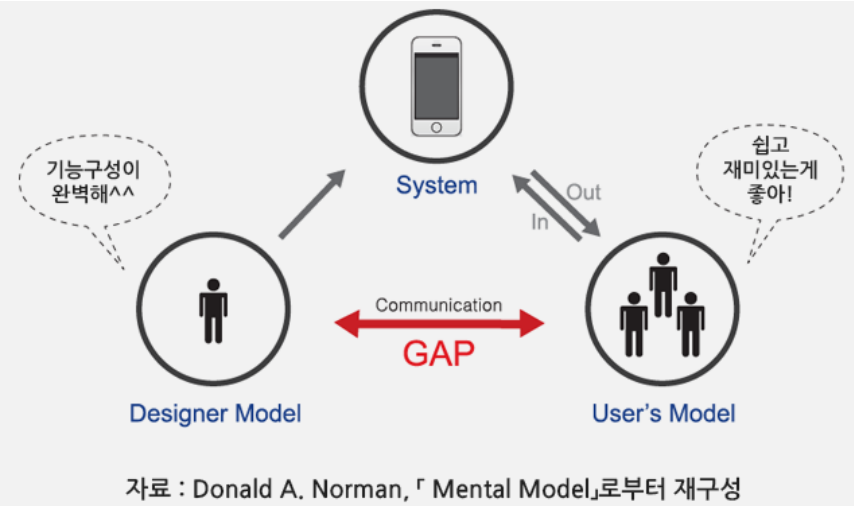
순수한 가공물 (PURE FABRICATION)이란?

- 눈치가 빠른 사람이라면 방금 만든 FACTORY가 도메인 모델에 속하지 않는다는 사실을 알아챘을 것이다.
  - **FACTORY를 추가한 이유는 순수하게 기술적인 결정이다. 전체적으로 결합도를 낮추기 위해 만들어진 순수한 가공물이다.**
- 이처럼 **책임을 할당하기 위해 창조되는 도메인과 무관한 인공적인 객체를 순수한 가공물**이라 부른다.
  - 즉, 도메인 모델을 기반으로 책임을 할당하는 과정에서 도움을 주는 객체.

표현적 분해와 행위적 분해

크레이그 라만은 시스템을 객체로 분해하는 데 크게 두 가지 방식이 존재한다고 설명한다.

- **표현적 분해**
  - **도메인에 존재하는 사물 또는 개념을 표현하는 객체들을 이용해 시스템을 분해하는 것**
  - 도메인 모델에 담겨 있는 개념과 관계를 따르며 도메인과 소프트웨어 사이의 표현적 차이를 최소화하는 것을 목적으로 한다.



자료 : Donald A. Norman, 「 Mental Model」로부터 재구성

- **행위적 분해**
  - 어떤 행동을 추가하려고 하는데 이 행동을 책임질 마땅한 도메인 개념이 존재하지 않는다면 순수한 가공물을 추가하고 이 객체에게 책임을 할당한다.
  - **이렇게 도메인 모델과 관련 없는 순수한 가공물을 생성하여 분해하는 것을 행위적 분해라 한다.**

> 이러한 측면에서 객체지향은 실세계의 모방이라는 말은 옳지 않다. (도메인 모델과는 관련 없는 순수한 객체를 많이 만들기 때문.)

만약 도메인 개념만으로 결합도를 낮출 수 없다면 인공적인 객체를 창조하라

- 만약 도메인 개념이 만족스럽지 못한다면 주저하지 말고 인공적인 객체를 창조하라
- FACTORY도 객체의 생성 책임을 할당할만한 도메인 객체가 존재하지 않을 때 선택할 수 있는 순수한 가공물이다.

3 의존성 주입

3-1 의존성 주입

의존성 주입이란?

- 사용하는 객체가 아닌 외부의 독립적인 객체가 인스턴스를 생성한 후 이를 전달해서 의존성을 해결하는 방법

의존성 주입 방법

- 생성자 주입 - 객체를 생성하는 시점에 생성자를 통한 의존성 해결
- setter 주입 - 객체 생성 후 setter 메서드를 통한 의존성 해결
- 메서드 주입 - 메서드 실행 시 인자를 이용한 의존성 해결

3-2 명시적인 의존성이 숨겨진 의존성보다 좋다

의존성을 주입해주는 또 다른 방법은 SERVICE LOCATOR다. 하지만 이 방법은 의존성을 숨긴다.

숨겨진 의존성 - SERVICE LOCATOR

```
public class Movie {
  private DiscountPolicy discountPolicy;

  public Movie(...) {
    ...
    this.discountPolicy = ServiceLocator.discountPolicy();
  }
}

public class ServiceLocator {
  private ServiceLocator(){}; // 객체 생성 불가

  private static ServiceLocator soleInstance = new ServiceLocator();
```

```
  private DiscountPolicy discountPolicy;

  public static DiscountPolicy discountPolicy() {
    return soleInstance.discountPolicy();
  }

  public static void provide(DiscountPolicy discountPolicy) {
    soleInstance.discountPolicy = discountPolicy;
  }
}

// 클라이언트
ServiceLocator.provide(new AmountDiscountPolicy(...)); // 의존성 추가 (만약 이 코드가 없다면 에러)
Movie avatar = new Movie("아바타",
                         Duration.ofMinutes(120),
                         Money.wons(120)); // 의존성이 숨겨진다.
```

- SERVICE LOCATOR (서비스 중개자 패턴)은 의존성을 해결해주는 좋은 도구처럼 보인다.
- 하지만, 의존성을 숨기기 때문에 안티패턴이라 불린다.
- **아무리 의존성을 해결해주는 좋은 도구여도, 의존성을 숨겨버리면 유지보수나 협업에 있어서 좋지 않다.**

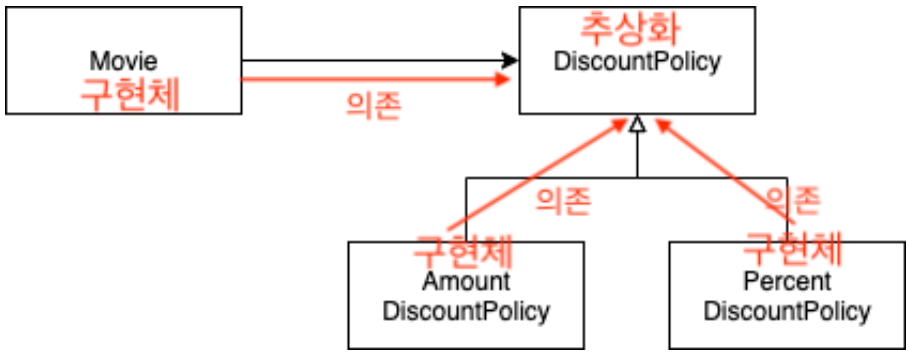
퍼블릭 인터페이스에 의존성을 명시적으로 드러내라

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

- 퍼블릭 인터페이스에 명시적으로 의존성이 나타나는 것이 숨겨진 의존성보다 좋다.
- 가급적 의존성을 객체의 퍼블릭 인터페이스에 노출하라.
    - 의존성을 구현 내부에 숨기면 숨길수록 코드를 이해하기도, 수정하기도 어려워진다.

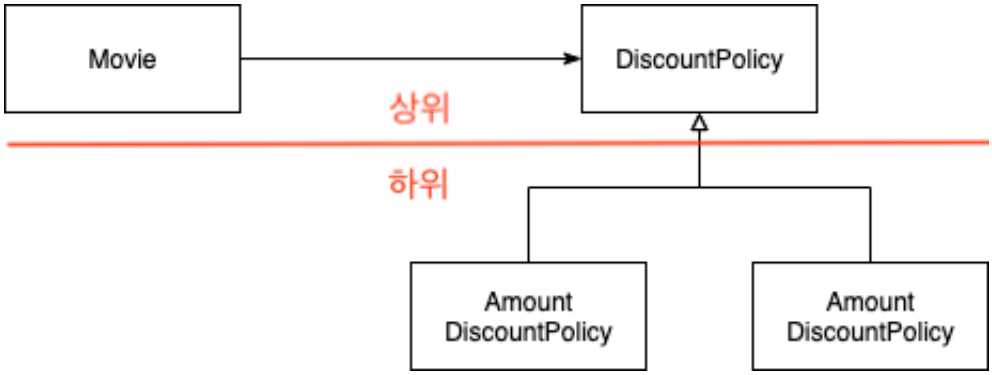
4 의존성 역전 원칙 DIP

4-1 의존성 역전 원칙



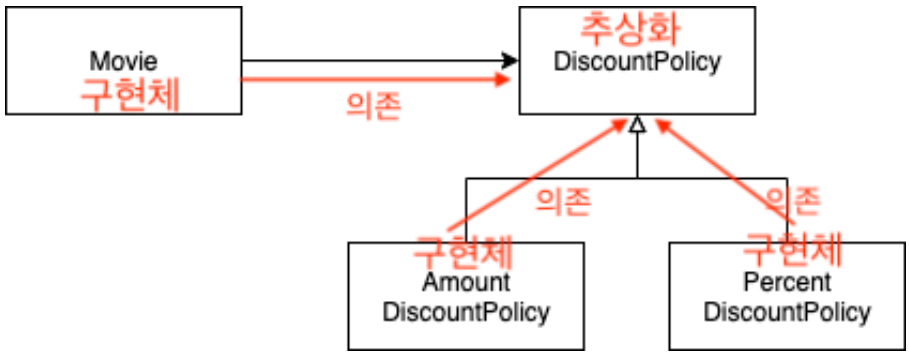
- 의존은 추상과 이루어져야 한다는 원칙
- **추상화된 것은 구체적인 것에 의존하면 안된다. 구체적인 것이 추상화된 것에 의존해야 한다 - 로버트 C. 마틴 -**
  - 자주 변하는 것보다 변화하기 어려운 것, 변화가 거의 없는 것에 의존하라는 것.
- DIP의 특징
  - 상위 수준의 모듈은 하위 수준의 모듈에 의존하면 안된다
  - 추상화는 구체적인 사항에 의존해서는 안 된다.

4-2 상위 수준의 모듈은 하위 수준의 모듈에 의존하면 안된다



- 상위 수준의 클래스는 어떤 식으로든 하위 수준의 클래스에 의존해서는 안된다
- **상위 수준의 변경에 의해 하위 수준이 변경되는 것은 납득할 수 있지만 하위 수준의 변경으로 인해 상위 수준이 변경돼서는 곤란하다.**
  - 인터페이스의 변경으로 인해 구현체가 변경되는 것은 납득 가능
  - 구현체가 변경되는 것이 인터페이스를 변경한다는 것은 납득 불가능 (말이 안된다)

4-3 추상화는 구체적인 사항에 의존해서는 안된다



- **상위 수준의 클래스와 하위 수준의 클래스 모두 추상화에 의존한다**
    - `Movie`는 추상 클래스인 `DiscountPolicy`에 의존한다
    - `AmountDiscountPolicy`도 `DiscountPolicy`에 의존한다
- **추상화에 의존하라.**
    - 유연하고 재사용 가능한 설계를 원한다면 모든 의존성의 방향이 추상 클래스나 인터페이스와 같은 추상화를 따라야 한다

4-4 역전의 의미

- DIP를 따르는 설계는 **의존성 방향이 전통적인 절차형 프로그래밍과는 반대 방향으로 나타나기 때문.**

5 유연성에 대한 조언

주의사항

5-1 유연한 설계는 유연성이 필요할 때만 옳다

유연하고 재사용 가능한 설계란

- 런타임 의존성과 컴파일타임 의존성의 차이를 인식하고 동일한 컴파일타임 의존성으로부터 다양한 런타임 의존성을 만들 수 있는 코드 구조를 가지는 설계.
    - 인터페이스 -> 구현체

유연성은 좋지만 항상 복잡성을 수반한다

- 유연성은 좋지만 항상 복잡성을 수반한다. 유연하지 않은 설계는 단순하고 명확하다.
- **불필요한 유연성은 불필요한 복잡성을 낳는다**
    - 단순하고 명확한 해법이 그런대로 만족스럽다면 유연성을 제거하라
    - 유연성은 코드를 읽는 사람들이 복잡함을 수용할 수 있을 때만 가치가 있다.

5-2 협력과 책임이 중요하다

객체의 협력과 책임이 더 중요하다

- 객체의 협력과 책임이 더 중요하다
- 먼저 역할, 책임, 협력에 초점을 맞춰야 한다.

역할, 책임, 협력 -> 생성

- 객체들의 역할, 책임, 협력을 생각하고 객체 생성 메커니즘을 생각하라.
- **마치 객체가 이미 존재하는 것처럼 객체들의 역할, 책임, 협력을 생각하고 마지막으로 객체 생성 메커니즘을 생각하자.**

6 정리

OCP - 구체적인 것이 아닌 추상적인 것에 의존하라

- 컴파일타임 의존성을 고정시키고 런타임 의존성을 변경하라
- 추상화가 핵심이다. 구체적인 것이 아닌 추상적인 것에 의존해야한다.

생성과 사용 책임을 분리하라

- 생성과 사용의 책임을 분리해라
- 생성에 관해서 도메인 모델에 존재하지 않는 순수한 객체를 만들어서 사용해라.
    - Factory

숨겨진 의존성은 나쁘다

- 아무리 생성 책임을 잘 수행하는 객체를 만들 수 있어도 의존성을 숨기면 안 좋다.
- 명시적인 의존성이 훨씬 좋다.
    - 의존성 주입 -> 전략패턴

DIP - 상위 수준의 모듈은 하위 수준의 모듈에 의존하면 안된다

- 상위 수준의 모듈은 하위 수준의 모듈에 의존하면 안된다.
    - 하위 수준의 모듈들을 추상화하여 상위 수준으로 만들어 의존성을 역전 시켜라

무엇보다 중요한 것은 역할, 책임 ,협력이다

- 유연성은 복잡성을 수반한다.
- 너무 유연성을 따지지말고 객체 생성의 책임을 무시한채 역할, 책임, 협력을 기반으로 객체를 먼저 만들고 나서 생성의 대한 매커니즘을 생각하라.

> 상속
>
> - 부모의 대한 정보를 알아야 한다.
>   - 부모가 변경되면 자식도 변경된다
>   - 부모와 자식의 퍼블릭 인터페이스가 합쳐진다.
>   - 상속의 경고
> - 부모의 내용을 그대로 물려준다.
>
> 합성
>
> -