오브젝트

CHAPTER 04. 설계 품질과 트레이드오프

> 데이터 중심의 접근법을 취할 경우 직면하게 되는 다양한 문제점을 이야기

> 데이터 중심의 설계는 행동보다 데이터를 먼저 결정하고 협력이라는 문맥을 벗어나 고립된 객체의 상태에 초점을 맞추기 때문에
>
> 캡슐화를 위반하기 쉽고, 요소들 사이의 결합도가 높아지며, 코드를 변경하기 어려워진다

- 객체지향 설계의 핵심 ***역할, 책임, 협력***
  - 협력 : 기능 구현을 위해 메시지를 주고 받는 객체들 사이의 상호작용
  - 책임 : 다른 객체와 협력하기 위해 수행하는 행동
  - 역할 : 대체 가능한 책임의 집합
- 세가지 핵심 중 가장 중요한 것은 **책임** 이다
  - 책임이 적절하게 할당되지 못한다면 ⇢ 원활한 **협력 x**
  - 역할은 책임의 집합이기 때문에 ⇢ 역할 역시 **협력과 조화를 이루지 못한다**

  > 결국, 책임이 ***전체의 품질을 결정***

- 객체지향 설계란 올바른 객체에게 올바른 책임을 할당하면서 낮은 결합도와 높은 응집도를 가진 구조를 창조하는 활동

  위 정의에서 확인할 수 있는 설계에 관한 두 가지 관점
  - 객체지향 설계의 핵심 == 책임
  - 책임을 할당하는 작업이 응집도와 결합도 같은 설계 품질과 깊은 연관이 있다는 것
- 훌륭한 설계란 **합리적인 비용 안에서 변경을 수용할 수 있는 구조를 만드는 것**

  적절한 비용 안에서 쉽게 변경할 수 있는 설계는 **응집도가 높고** 서로 **느슨하게 결합** 돼 있는 요소로 구성된다
- 객체의 **행동** 에 초점을 맞추면 ⇢ 결합도와 응집도를 합리적인 수준으로 유지할 수 있다
- 객체의 **책임** 에 초점을 맞추면 ⇢ 객체지향 설계가 가능해 진다
  - 객체의 상태에서 행동으로, 객체 사이의 상호작용으로 설계의 중심을 이동시키고
  - 결합도가 낮고 응집도가 높으며
  - 구현을 효과적으로 캡슐화 하는 객체들을 창조할 수 있는 기반을 제공한다

> 2장에서 책임을 기준으로 구현한 영화 예매 시스템을 데이터 중심의 설계로 살펴보고 객체지향적으로 설계한 구조와 어떤 차이점이 있는지 살펴보자

1. 데이터 중심의 영화 예매 시스템

객체지향 설계에서의 객체 분할방법 두가지

1. 상태 중심 (책에서는 '데이터'를 '상태'와 동일한 의미로 사용)
    - 자신이 포함하고 있는 데이터(상태)를 조작하는데 필요한 오퍼레이션을 정의한다
    - 객체를 독립된 데이터(상태) 덩어리로 바라본다
2. 책임 중심
    - 다른 객체가 요청할 수 있는 오퍼레이션을 위해 필요한 상태를 보관한다
    - 객체를 협력하는 공동체 일원으로 바라본다

객체의 상태

- 객체의 상태는 구현에 속한다
- 구현은 불안정하기 때문에 변하기 쉽다
- 상태를 중심으로 객체를 분할하면 구현의 세부사항이 객체의 인터페이스에 스며들게 되어 캡슐화의 원칙이 무너진다
- 결과적으로 상태 변경은 인터페이스의 변경을 초래하며 이 인터페이스에 의존하는 모든 객체에게 변경의 영향이 퍼지게 된다

> 따라서 데이터에 초점을 맞추는 설계는 변경에 취약할 수밖에 없다.

객체의 책임

- 객체의 책임은 인터페이스에 속한다
- 책임을 드러내는 안정적인 인터페이스 뒤로 책임을 수행하는데 필요한 상태를 캡슐화함으로써 구현 변경에 대한 파장이 외부로 퍼져나가는 것을 방지한다

> 따라서 책임에 초점을 맞추면 상대적으로 변경에 안정적인 설계를 얻을 수 있게 된다.

1-1. 데이터를 준비하자

- 데이터 중심의 설계는 객체가 내부에 저장해야 하는 '데이터가 무엇인가'를 묻는 것으로 시작한다
- 기존에 책임 중심으로 설계할 때는 없었던 MovieType 이라는 할인 정책의 종류를 결정하는 타입을 생성하게 된다
    - Movie 가 할인 금액을 계산하는데 필요한 데이터는 무엇인가? ⇢ 구분하는 타입이 필요하고 각 타입에 관련된 데이터가 또다시 필요하다
    - 각 타입(금액 할인 정책과 비율 할인 정책)에 대한 데이터 ⇢ discountAmount, discountPercent 모두 필요
- 데이터 중심 설계는 객체가 포함해야 하는 데이터에 집중한다

1-2. 영화를 예매하자

- 기존에 책임 중심으로 설계할 때 없었던 ReservationAgency 클래스가 추가되었다
    - 데이터 클래스들을 조합해서 영화 예매 절차를 구현
    - 크게 두 부분으로 나눌 수 있다
        1. 할인 가능 여부 확인
        2. discountable 변수의 값을 체크하고 적절한 할인 정책에 따라 예매 요금을 계산하는 if 문

2. 설계 트레이드오프

> 데이터 중심 설계와 책임 중심 설계의 장단점 비교를 위해 **캡슐화**, **응집도**, **결합도** 를 사용하겠다.
>
> 본격적으로 두 가지 방법을 비교하기 전에 세가지 품질 척도의 의미를 살펴보자.

2-1. 캡슐화

- 상태와 행동을 하나의 객체 안에 모으는 이유는 객체의 **내부 구현을 외부로부터 감추기 위해서** 이다
- 객체지향이 강력한 이유는 한 곳에서 일어난 변경이 전체 시스템에 영향을 끼치지 않도록 **파급효과** 를 적절하게 조절할 수 있는 장치를 제공하기 때문이다
- 객체를 사용해 변경 가능성이 높은 부분(구현)은 내부에 숨기고(캡슐화) 외부에는 상대적으로 안정적인 부분(인터페이스)만 공개함으로써 변경의 여파를 통제할 수 있다
    - **구현** : 변경될 가능성이 높은 부분
    - **인터페이스** : 상대적으로 안정적인 부분

    > 객체를 설계하기 위한 가장 기본적인 아이디어는
    >
    > 변경의 정도에 따라 구현과 인터페이스를 분리하고 외부에서는 인터페이스에만 의존하도록 **관계를 조절하는 것** 이다. ⇠ 적절한 의존성 관리

- 캡슐화는 외부에서 알 필요가 없는 부분을 감춤으로써 대상을 단순화하는 **추상화의 한 종류**
    - 또는 ***변경 가능성이 높은 부분을 객체 내부로 숨기는 추상화 기법***
    - 변경될 수 있는 어떤 것이라도 캡슐화해야 한다 ⇠ 객체지향 설계의 중요한 원리 (안정적인 인터페이스 뒤로 불안정한 구현 세부사항을 캡슐화 하는 것)

> 설계가 필요한 이유는 요구사항이 변경되기 때문이고, 변경의 관점에서 설계의 품질을 판단하기 위해 캡슐화를 기준으로 삼을 수 있다.

2-2. 응집도와 결합도

응집도(Cohesion)

- 모듈에 포함된 내부 요소들이 연관돼 있는 정도를 나타낸다
- 모듈 내의 요소들이 하나의 목적을 위해 긴밀하게 협력한다면 그 모듈은 높은 응집도를 가진다

  반대로 모듈 내의 요소들이 서로 다른 목적을 추구한다면 그 모듈은 낮은 응집도를 가진다
- 객체지향 관점에서 응집도는 객체에 얼마나 관련 높은 **책임** 들을 할당했는지를 나타낸다

결합도(Coupling)

- 의존성의 정도를 나타내며 다른 모듈에 대해 얼마나 많은 지식을 갖고 있는지를 나타내는 척도다
  - 한 모듈이 변경되기 위해서 다른 모듈의 변경을 요구하는 정도
- 한 모듈이 다른 모듈에 대해 너무 자세한 부분까지 알고있다면 두 모듈은 **높은 결합도** 를 가진다
  - 꼭 필요한(적절한) 지식만 알고있어야 한다 ⇢ 낮은 결합도
- 객체지향 관점에서 결합도는 객체 또는 클래스가 **협력에 필요한 적절한 수준의 관계만을 유지하고 있는지** 를 나타낸다

좋은 설계와 응집도, 결합도의 관계

1. 좋은 설계란
   - 높은 응집도(High Cohesion)와 낮은 결합도(Low Coupling)를 가진 모듈로 구성된 설계를 의미한다
   - 오늘의 기능을 수행하면서 내일의 변경을 수용할 수 있는 설계다
2. 좋은 설계를 만들기 위해서는 높은 응집도와 낮은 결합도를 추구해야 한다

높은 응집도와 낮은 결합도를 가진 설계를 추구해야 하는 이유

- 설계를 **변경하기 쉽게** 만들기 때문
  - 변경과 응집도
    - **응집도가 높은 설계** 는 하나의 변경을 반영하기 위해 **하나의 모듈만 수정** 하는 반면,
    - **응집도가 낮은 설계** 는 하나의 원인에 의해 변경해야 하는 부분이 다수의 모듈에 분산되어 **여러 모듈을 동시에 수정해야 한다**
  - 변경과 결합도
    - 결합도가 높으면 높을수록 **함께 변경해야 하는 모듈의 수**가 늘어나기 때문에 변경이 어려워진다
    - 모듈 내부의 변경의 원인이 **다른 모듈에 영향을 미치는 경우에도 결합도가 높다** 고 표현한다
    - 퍼블릭 인터페이스를 수정했을 때만 다른 모듈에 영향을 미치는 경우 **결합도가 낮다** 고 표현한다
      - 클래스의 구현이 아닌 인터페이스에 의존하도록 코드를 작성해야 낮은 결합도를 얻을 수 있다는 이야기
      - 또는 변경될 확률이 매우 적은 안정적인 모듈에 의존하라(ex. 표준 라이브러리에 포함된 모듈이나 성숙한 프레임워크)

3. 데이터 중심의 영화 예매 시스템의 문제점

> 데이터 중심의 설계가 가진 문제점을 살펴보자

데이터 중심의 예매 시스템은 **기능적인 측면** 에선 2장에서 구현한 설계와 완전히 동일하지만, **설계 관점** 에서는 완전히 다르다

근본적인 차이는 **캡슐화를 다루는 방식** 이다

- 데이터 중심 설계 ⇢ 객체의 내부 구현을 인터페이스의 일부로 만든다 (ex. getter, setter)
- 책임 중심 설계 ⇢ 객체의 내부 구현을 안정적인 인터페이스 뒤로 캡슐화한다 (ex. 협력하는 메시지)

3-1. 캡슐화 위반

- 설계 시 협력에 관해 고민하지 않으면 캡슐화를 위반하는 과도한 접근자(getter)와 수정자(setter)를 가지게 되는 경향이 있다

  객체가 사용된 문맥을 추측할 수밖에 없는 경우 개발자는 어떤 상황에서도 해당 객체가 사용될 수 있게 최대한 많은 접근자 메서드를 추가하게 되는 것

- 객체가 사용될 협력(인터페이스, 메시지)을 고려하지 않고 객체가 다양한 상황에서 사용될 수 있을 것 이라는 막연한 추측을 기반으로 설계를 진행한다

  따라서 내부 상태를 드러내는 메서드를 추가하게 되고, 결과적으로 대부분의 내부 구현이 퍼블릭 인터페이스(메시지)에 그대로 노출될 수 밖에 없다

⇢ 결과적으로 캡슐화의 원칙을 위반하는 **변경에 취약한 설계** 를 얻게 된다.

3-2. 높은 결합도

- 캡슐화를 위반해서 객체 내부의 **구현**이 객체의 **인터페이스에 드러난다는 것** 은 **클라이언트가 구현에 강하게 결합** 된다는 것을 의미한다
- 객체의 내부 구현을 변경했음에도 이 인터페이스에 의존하는 모든 클라이언트들도 함께 변경해야 한다는 문제가 있다
- 데이터 중심 설계는 제어 로직이 특정 객체 안에 집중되기 때문에 하나의 제어 객체가 다수의 데이터 객체에 강하게 결합된다

  이 결합도로 인해 어떤 데이터 객체를 변경하더라도 제어 객체를 함께 변경할 수 밖에 없다

- 예제의 ReservationAgency 가 모든 데이터 객체에 의존하고 있기 때문에, 어떤 객체의 변도 ReservationAgency 의 변경을 유발하게 된다
  - 그림 4.4 참고 (p.115)
  - 해당 예제는 데이터 중심의 설계가 결합도와 관련해 가지는 치명적인 문제점을 잘 보여준다

데이터 중심의 설계는 전체 시스템을 하나의 거대한 의존성 덩어리로 만들어 버리기 때문에 어떤 변경이라도 시스템 전체가 요동칠 수밖에 없게 된다

3-3. 낮은 응집도

- 서로 다른 이유로 변경되는 코드가 하나의 모듈안에 공존할 때 모듈의 응집도가 낮다고 말한다

  코드를 수정하는 이유가 무엇인지와 연관되어 있다
- 설계 측면의 두 가지 문제점
  1. 변경의 이유가 서로 다른 코드들을 하나의 모듈 안에 뭉쳐놓았기 때문에 변경과 아무 상관이 없는 코드들이 영향을 받게 된다
  2. 하나의 요구사항 변경을 반영하기 위해 동시에 여러 모듈을 수정해야 한다

     응집도가 낮을 경우 다른 모듈에 위치해야 할 책임의 일부가 엉뚱한 곳에 위치하게 되기 때문
- 어떤 요구사항 변경을 수용하기 위해 하나 이상의 클래스를 수정해야 하는 것은 설계의 응집도가 낮다는 증거다

4. 자율적인 객체를 향해

4-1. 캡슐화를 지켜라

- 데이터 중심의 설계가 낮은 응집도와 높은 결합도라는 문제를 갖게 된 근본적인 원인은 바로 **캡슐화의 원칙을 위반** 했기 때문이다
- 객체는 스스로의 상태를 책임져야 하며 외부에서는 **인터페이스에 정의된 메서드(메시지)** 를 통해서만 상태에 접근할 수 있어야 한다
- 속성의 가시성을 private 으로 설정했다고 해도 접근자와 수정자를 통해 속성을 외부로 제공하고 있다면 캡슐화를 위반하는 것
- 객체가 스스로의 상태를 책임질 수 있게 외부의 책임을 객체 내부로 **책임을 이동** 시켜라

4-2. 스스로 자신의 데이터를 책임지는 객체

- 상태와 행동을 객체라는 하나의 단위로 묶는 이유 ⇢ 객체 스스로 자신의 상태를 처리할 수 있게 하기 위해
- 객체는 단순한 데이터 제공자가 아니다, 객체 내부에 저장되는 데이터보다 객체가 협력에 참여하면서 수행할 책임을 정의하는 오퍼레이션이 더 중요하다
  - 따라서 객체를 설계할 때 '이 객체가 어떤 데이터를 포함해야 하는가?'라는 질문을 두개의 개별적인 질문으로 분리해서 생각하자
    1. 이 객체가 어떤 데이터를 포함해야 하는가? (기존 구현을 따른다)
    2. 이 객체가 데이터에 대해 수행해야 하는 ***오퍼레이션*** 은 무엇인가? (이부분이 개선 포인트)
- 예제의 ReservationAgency 의 데이터에 대한 책임을 객체로 옮겨서 설계를 개선해본다
  - DiscountCondition 에 할인 조건을 판단할 수 있도록 isDiscountable 메서드를 추가한다

할인 조건이 두 가지 이기 때문에 두개의 isDiscountable 메서드를 추가한다

- Movie 에서 데이터를 처리하기위해 필요한 오퍼레이션으로 영화 요금을 계산하는 오퍼레이션과 할인 여부를 판단하는 오퍼레이션을 추가한다

  DiscountCondition 의 목록을 포함하기 때문에 할인 여부를 판단하는 오퍼레이션 역시 포함해야 하므로 isDiscountable 메서드를 추가하자

- Screening 도 동일한 방식으로 calculateFee 라는 메서드를 추가한다

> 그림 4.5 (p.125) 클래스 다이어그램과 그림 4.4 (p.115) 클래스 다이어그램의 비교
>
> 최소한의 결합도 측면에서 ReservationAgency 에 의존성이 몰려있던 첫 번째 설계보다 개선됐다.
>
> 두 번째 설계가 첫 번째 설계보다 내부 구현을 더 면밀하게 캡슐화하고 있기 때문이다.
>
> 두 번째 설계에서는 데이터를 처리하는데 필요한 메서드를 데이터를 가지고 있는 객체 스스로 구현하고 있다.
>
> 따라서 이 객체들은 스스로를 책임진다고 말할 수 있다.

5. 하지만 여전히 부족하다

> 설계를 개선했으나 첫 번째 설계에서 발생했던 대부분의 문제가 두 번째 설계에서도 발생하는 이유

5-1. 캡슐화 위반

문제점

객체가 자신의 상태를 스스로 관리하지만, 데이터 중심 설계를 함으로써 ***객체 내부의 정보를 인터페이스를 통해 외부에 노출*** 시키고 있다

- 내부의 속성이 변경될 경우 내부의 속성을 파라미터로 받는 메서드가 수정되고 해당 메서드를 사용하는 모든 클라이언트들도 수정되어야 한다
  - 내부의 구현의 변경이 외부로 퍼져나가는 **파급 효과(ripple effect)** ⇢ 캡슐화가 부족하다는 증거
- Movie 클래스 역시 내부 구현을 인터페이스에 노출시키고 있다
  - 할인 정책의 종류를 노출 (calculate**{TYPE}**DiscountedFee)
  - 새로운 할인 정책이 추가되거나 제거된다면 각 메서드들에 의존하는 모든 클라이언트가 영향을 받는다 ⇢ 할인 정책의 구현을 캡슐화하지 못 함

> **캡슐화의 진정한 의미**
>
> 구현과 관련된 변하는 어떤 것이든 감추는 것
>
> 내부 구현의 변경으로 인해 외부의 객체가 영향을 받는다면 캡슐화를 위반한 것

5-2. 높은 결합도

- 캡슐화 위반으로 인해 DiscountCondition 의 내부 구현이 외부로 노출 ⇢ Movie 와 DiscountCondition 의 결합도가 높다
- 두 객체의 결합도가 높을 경우 한 객체의 구현을 변경할 때 다른 객체에게 변경의 영향이 전파될 확률이 높아진다
- DiscountCondition 의 인터페이스가 아닌 '구현'이 변경하는 경우에도 Movie 가 변경된다는 것은 두 객체 사이의 결합도가 높다는 것을 의미

> 모든 문제의 원인은 **캡슐화** 원칙을 지키지 않았기 때문이다.
>
> DiscountCondition 의 내부 구현을 제대로 캡슐화 하지 못했기 때문에 다른 객체와의 **결합도** 도 함께 높아진 것이다.
>
> 유연한 설계를 하기위한 첫번째 목표는 **캡슐화** 이다.

5-3. 낮은 응집도

- 하나의 변경을 수용하기 위해 코드의 여러 곳을 동시에 변경해야 하는 것 ⇢ 응집도가 낮다는 증거

```
public class Screening {
  public Money calculateFee(int audienceCount) {
    switch (movie.getMovieType()) {
      case AMOUNT_DISCOUNT:
        if (movie.isDiscountable(whenScreened, sequence)) {
          return movie.calculateAmountDiscountFee().times(audienceCount);
        }
        break;
      case PERCENT_DISCOUNT:
        if (movie.isDiscountable(whenScreened, sequence)) {
          return movie.calculatePercentDiscountFee().times(audienceCount);
        }
      case NONE_DISCOUNT:
        return movie.calculateNoneDiscountFee().times(audienceCount);
    }

    return movie.calculateNoneDiscountFee().times(audienceCount);
  }
}
```

> 응집도가 낮은 이유 또한 **캡슐화를 위반** 했기 때문이다.

> DiscountCondition 과 Movie 의 내부 구현이 인터페이스에 그대로 노출되고 있고 Screening 은 노출된 구현에 직접적으로 의존하고 있다.

6. 데이터 중심 설계의 문제점

> 두 번째 설계가 변경에 유연하지 못한 이유는 **캡슐화를 위반** 했기 때문

캡슐화를 위반한 설계가 변경에 취약한 이유

캡슐화를 위반한 설계를 구성하는 요소들이 **높은 응집도** 와 **낮은 결합도** 를 가질 확률은 극히 낮다.

⇢ 높은 응집도와 낮은 결합도를 가지지 못했기 때문에 변경에 취약하다

데이터 중심의 설계가 변경에 취약한 이유

1. 너무 이른 시기에 **데이터에 관해 결정하도록 강요한다**
    - 설계 시작 시 '이 객체가 포함해야 하는 데이터가 무엇인가?' 와 같은 질문을 함으로써
        - 객체의 상태에 초점을 맞추게 되고
        - 같은 말로는 초점이 객체의 내부로 향해있는 것
2. 협력이라는 문맥을 고려하지 않고 **객체를 고립시킨 채 오퍼레이션을 결정** 한다
    - 데이터 중심 설계의 초점은 객체 내부로 향하기 때문에 객체가 관리할 데이터의 세부 정보를 먼저 결정한다

      객체의 구현이 이미 결정된 상태에서 다른 객체와의 협력 방법을 고민하기 때문에 이미 구현된 객체의 인터페이스를 억지로 끼워맞출 수밖에 없다

6-1. 데이터 중심 설계는 객체의 행동보다는 상태에 초점을 맞춘다

- 데이터를 먼저 결정하고 데이터를 처리하는 데 필요한 오퍼레이션을 나중에 결정하는 방식은 데이터에 관한 지식이 객체의 인터페이스에 고스란히 드러나게 된다 ⇢ 캡슐화 실패 ⇢ 변경에 취약해짐
- 객체의 내부 구현이 인터페이스를 어지럽히고 객체의 응집도와 결합도에 나쁜 영향을 미치기 때문에 변경에 취약한 코드를 만든다

6-2. 데이터 중심 설계는 객체를 고립시킨 채 오퍼레이션을 정의하도록 만든다

> 올바른 객체지향 설계의 무게 중심은 항상 객체의 내부가 아니라 외부에 맞춰져 있어야 한다.
>
> 중요한 것은 객체가 ***다른 객체와 협력하는 방법*** 이다.
>
> (개인적인 생각으로 '인터페이스를 먼저 생각한다' 는 점에서 TDD 가 생각나는 부분이었다)

- 데이터 중심 설계는 데이터의 세부 정보를 먼저 결정하기 때문에 **초점이 객체의 내부로 향해있는 것** ⇢ 문제점

올바른 객체지향 설계의 무게 중심은 항상 객체의 내부가 아니라 **외부에 맞춰져 있어야 한다** ⇢ 원활한 협력을 위해, 메시지를 먼저 생각해야 한다는 것

- 협력이 구현 세부사항에 종속되어 있음 (인터페이스에 구현이 노출) ⇢ 객체 내부 구현 변경 시 협력하는 객체 모두 영향을 받는 문제 발생

마무리

이번 장에서는 2장에서 만든 영화 예매 시스템을 **데이터 중심으로 설계** 하고 캡슐화, 응집도, 결합도 관점에서 문제점을 이야기하고 일부를 개선했다.

다음장에서 책임을 할당하는 방법을 적용하여 좀 더 객체지향적인 설계로 개선해 나간다.

정리

높은 응집도와 **낮은 결합도**를 이용해 **적절한 의존성**을 가진 **변경에 유연한 설계**를 해야 한다.

그러기 위해서 **캡슐화** 원칙을 지켜야 하며 구현보다 **인터페이스에 의존** 하는 설계를 해야 한다.

참고

[번역] TDD 변절자 : TDD는 설계 기법이 아니다

⇢ TDD 를 SOLID 원칙과 비교하며 설계관점에서의 TDD 를 이야기한다