5. 책임 할당하기

- 0) 서론
  - 책임에 초점을 맞춰 설계할 때 어떤 객체에 어떤 책임을 할당할지 결정하기 쉽지 않다
    - 책임 할당 과정은 트레이드오프 활동
- 1) 책임 주도 설계를 위해
  - 책임 중심의 설계로 전환하기 위한 두 가지 원칙
    - 데이터보다 행동을 먼저 결정하라
    - 협력이라는 문맥 안에서 책임을 결정하라
  - 데이터보다 행동을 먼저 결정하라
    - 클라이언트 관점의 행동은 객체의 책임을 의미
    - 책임 중심의 설계는 객체의 책임을 결정한 후 책임을 수행하는데 필요한 데이터가 무엇인가 결정한다
    - 객체지향 설계에서 가장 중요한 것은 적절한 객체에게 적절한 책임을 할당하는 능력
      - 객체에 어떤 책임을 할당해야 하는지는 협력에서 찾을 수 있다
  - 협력하는 문맥 안에서 책임을 결정하라
    - 책임은 객체의 임무이 아니라 객체가 참여하는 협력에 책임해야 한다
      - 협력에 적합한 책임이냐 메시지 전송자에게 적합한 책임
        - 클라이언트의 의도에 적합한 책임을 할당해야 한다
    - 메시지를 결정한 후에 객체를 선택해야 한다
      - 메시지가 존재하기 때문에 그 메시지를 처리할 객체가 필요한 것
    - "이 클래스는 무엇을 해야하지?" 에서 "메시지를 누구에게 전송해야 하지" 라고 질문하는 것이 메시지 기반 설계로 향하는 첫걸음이다
    - 메시지를 먼저 결정하기 때문에 송신자는 메시지 수신자에 대한 가정을 할 수 없다
      - 메시지 전송자 관점에서 메시지 수신자가 잘 협력해 했을까 한다
  - 책임 주도 설계
    - 책임 주도 설계의 핵심은 책임을 정한 후에 책임을 수행할 객체를 결정하는 것
- 2) 책임 할당을 위한 GRASP 패턴
  - 도메인 개념에서 출발하기
    - 설계를 시작하는 단계에서 개념들의 의미와 관계가 정립되거나 완벽할 필요가 없다.
      - 책임을 할당받을 객체들의 종류와 관계에 대한 정보를 제공하는 출발점으로 도메인을 사용
  - 정보 전문가에게 책임을 할당하라
    - 애플리케이션이 제공해야 하는 기능을 애플리케이션의 책임으로 생각
      - 이 책임을 애플리케이션에 대해 전송된 메시지로간주하고 메시지를 책임을 완벽해 객체를 선택하는 것으로 설계 시작
    - 메시지를 전송할 객체는 무엇을 원하는가
    - 메시지를 수신할 적합한 객체는 누구인가
      - 정보 전문가 패턴
        - 책임을 수행할 정보를 알고 있는 객체에게 책임을 할당
        - 정보는 데이터와 다르다, 객체가 정보를 알고 있다 해서 정보를 저장할 필요는 없다
    - 메시지를 객체에 할당하고 객체 내부로 들어가 메시지를 처리하는 절차와 구현을 고민
      - 스스로 처리할 수 없는 작업이 있으면 역무에 도움 요청
        - 새로운 메시지가 되고 새로운 객체에 새로운 책임이 부여
          - 이 과정을 통해 협력 공동체가 구성된다
  - 높은 응집도와 낮은 결합도
    - 책임을 할당할 수 있는 다양한 대안들이 존재하면 높은 응집도와 낮은 결합도를 고려해 선택한다
    - 높은 응집도
      - 객체가 필요없는 책임을 떠안으면 하나의 변경에 생길때 같이 변경되므로 응집도가 낮아진다
    - 낮은 결합도
      - 협력을 잘못 구성하면 필요없는 의존이 생겨 결합도가 증가한다
  - 창조자에게 객체 생성 책임을 할당하라
    - 객체를 생성할 때 아래 조건을 최대한 많이 만족하는 객체에 객체 생성 책임을 할당한다
    - B가 A 객체를 포함하거나 참조한다
      - B가 A 객체를 기록한다
        - B가 A 객체를 긴밀하게 사용한다
          - B가 A 객체를 초기화 하는데 필요한 데이터를 가지고 있다
- 3) 구현을 통한 검증
  - 메시지를 결정하고 책임을 결정
    - 책임을 수행하는데 필요한 인스턴스 변수를 결정
      - 메시지는 송신자의 의도를 드러내도록 이름을 결정
  - 메시지를 기반으로 협력을 구성하는 책임 주도 설계 방식으로 설계하면 캡슐화가 낮은 결합도를 달성할 수 있다
  - DiscountCondition 개선하기
    - DiscountCondition 은 하나 이상의 변경 이유를 가지기 때문에 응집도가 낮다
      - 연관성 없는 기능이나 데이터가 하나의 클래스에 뒤섞였다
        - 변경의 이유에 따라 클래스를 분리해야 한다
      - 새로운 할인 조건 추가
      - 순번 조건을 판단하는 로직 변경
      - 기간 조건을 판단하는 로직 변경
    - 보통 설계를 개선하는 작업은 변경의 이유가 하나 이상인 클래스를 찾는가 부터 시작
    - 코드를 통해 변경의 이유를 파악할 수 있는 방법
      - 인스턴스 변수가 초기화되는 시점을 살펴본다
        - 응집도가 높은 클래스는 인스턴스를 생성할 때 모든 속성을 초기화 한다
          - 함께 초기화되는 속성을 기준으로 코드를 분리한다
      - 메서드들이 인스턴스 변수를 사용하는 방식을 살펴본다
        - 모든 메서드가 객체의 모든 속성을 사용하면 클래스의 응집도는 높다
          - 속성 그룹과 해당 그룹에 접근하는 메서드 그룹을 기준으로 코드를 분리한다
  - 타입 분리하기
    - 기간, 순번으로 클래스를 분리하면 Movie 클래스가 상속 모두에게 영향받다
      - 결합도가 높아져 새로운 할인 조건을 추가하기가 더 어렵다
  - 다형성을 통해 분리하기
    - 역할
      - 동일한 메일 수행 = 동일한 역할 수행
        - 역할은 협력 안에서 대체 가능성을 의미
          - 역할을 사용해 객체의 구체적인 타입을 추상화 할 수 있다
    - POLYMORTHISM 패턴
      - 객체에 정시적인 타입에 따라 행동을 분기해야 하면 정시적인 타입을 명시적인 클래스로 정의하고 행동을 나눠 응집도 문제를 해결할 수 있다
  - 변경으로부터 보호하기
    - PROTECTED VARIATIONS 패턴
      - 인터페이스를 설계해야는 클래스를 추가하는 것으로 기능을 확장할 수 있다
      - 변경을 캡슐화 하도록 책임을 할당
        - 변화가 예상되는 불안정한 지점들을 식별하고 그 주위에 안정된 인터페이스를 형성
  - 변경과 유연성
    - 설계를 주도하는 것은 변경
      - 변경을 수용하도록 코드를 더 유연하게 만들기 위해 상속 대신 합성을 사용
    - 상속을 이용해 구현했다면, 상위 층 변경을 수용하기 위해 새로운 인스턴스를 생성하고 정보를 복사해야 한다
    - 상속 대신 합성을 사용하면 객체를 연결된 인스턴스를 교체하는 단순한 작업으로 바꿀 수 있다
- 4) 책임 주도 설계의 대안