5장. 책임과 메시지

> 훌륭하고 성장 가능한 시스템을 만들기 위한 핵심은 모듈 내부의 속성과 행동이 어떤가보다는 **모듈이 어떻게 커뮤니케이션하는가**에 달려있다.

자율적인 책임

🗽 자율적인 객체

스스로의 판단에 따라 각자 맡은 책임을 수행하는 객체

🧨 자율적인 책임

객체가 `어떻게 해야 하는가`가 아닌 `무엇을 해야 하는가`를 설명하는 것

- 기준 문맥 안에서 의도를 명확하게 설명할 수 있는 수준으로 추상적이어야 함

메시지와 메서드

💬 메시지

객체는 다른 객체에 메시지를 보내는 것으로 협력한다.

- 송신자: 메시지 전송을 통해서만 다른 객체의 책임을 요청
- 수신자: 메시지 수신을 통해서만 자신의 책임을 수행

객체가 수신할 수 있는 모양이 객체가 수행할 책임의 모양을 결정한다.

🔠 메서드

객체가 메시지를 처리하기 위해 내부적으로 선택하는 방법

- 메시지에 대응되는 특정 메서드를 실행함으로써 책임을 수행한다.
- 객체지향에서는 메시지 수신 객체가 **런타임 시점**에 메서드를 선택한다. (절차지향은 컴파일 시점)

🚂 다형성

동일한 메시지를 수신했을 때 서로 다른 메서드로 메시지를 처리하는 메커니즘

- 서로 다른 객체들이 동일한 책임을 공유하는 것 (메시지:메서드=1:N)
- 수신자의 종류를 캡슐화
  - 송신자 → 수신자 : 어떤 타입인지 관심 X
- 객체지향이 유연하고 확장 가능하고 재사용성이 높은 이유 !

    - 메시지를 이해한다면 수신자가 누구든 상관 없다. (협력의 유연성)
    - 송신자에게 영향을 미치지 않고 협력의 수행 방식을 확장할 수 있다.
    - 다양한 문맥에서 협력을 재사용할 수 있다.

✨ 메시지의 중요성

- 송신자 - 수신자 간 결합도를 낮춘다.
    - 설계의 유연성
    - 확장 가능성
    - 재사용성
- 객체지향 시스템은 객체 생성 후 상호 메시지를 송신할 수 있게 조합함으로써 구축된다.
- 설계 품질을 위해 훌륭한 메시지를 선택해야 한다.

메시지를 따라라

🏰 객체지향의 핵심이 되는 메시지

연쇄적으로 메시지를 주고받는 객체들 사이의 협력 관계를 기반으로 유용한 기능을 제공하라

- 시스템을 메시지를 주고받는 동적인 객체들의 집합으로 바라봐야 한다.
- 협력 관점에서의 접근
    - 다른 객체에 무엇을 제공해야 하는가 ?
    - 다른 객체로부터 무엇을 얻어와야 하는가 ?

🫧 **메시지 중심의 협력 설계**

💡 책임-주도 설계의 기본 아이디어가 되는 메시지

주고받는 메시지를 기반으로 적절한 역할/책임/협력을 발견하라

- 시스템이 수행할 책임을 구현하기 위해 협력 관계를 시작할 객체를 찾아 책임을 할당한다.
- 책임 완수를 위해 다른 객체의 도움이 필요할 경우 어떤 메시지가 필요한지 결정한다.
- 메시지 결정 이후 메시지를 수신하기 위한 적합한 객체를 선택한다.

🫧 **메시지는 수신자의 책임을 결정**

🔄 What/Who 사이클

어떤 행위가 필요한가 ? → 어떤 객체가 수행할 것인가?

- 객체의 행위를 결정하는 것은 객체 자체의 속성이 아니다.
- 협력이라는 문맥 안에서 객체의 책임과 역할이 결정된다.
- 책임을 결정하는 것은 메시지다.

🫧 **어떤 메시지가 필요한지 먼저 고민**

🧙🏾 묻지 말고 시켜라

송신자는 수신자가 누군지 몰라도 믿고 메시지를 전송한다.

- 객체지향 어플리케이션이 자율적인 객체들의 공동체라는 사실을 강조한다.
  - 객체는 다른 객체의 결정에 간섭하지 말 것
  - 모든 객체는 자신의 상태를 기반으로 스스로 결정할 것

객체 인터페이스

🕹️ 인터페이스

두 사물의 경계에서 서로 상호작용할 수 있게 이어주는 방법이나 장치

- 내부 구조나 동작을 몰라도 쉽게 대상 조작 / 의사 전달이 가능
- 인터페이스 자체 변경이 없어도 내부 구성이나 작동 방식을 변경 가능
- 대상이 변경되어도 동일한 인터페이스를 제공하면 문제없이 상호작용 가능
- 객체가 어떤 메시지를 수신할 수 있는가 == 인터페이스의 모양
- 객체의 공용 인터페이스를 구성 == 객체가 외부로부터 수신할 수 있는 메시지의 목록

🤔 책임, 메시지, 그리고 인터페이스

1. 협력에 참여하는 객체의 책임은 **자율적**일 것
2. 객체가 메시지를 수신했을 때 **적절한 객체의 메서드가 수행**될 것

3. 외부로부터 메시지를 받기 위해 **인터페이스**를 사용할 것

인터페이스와 구현의 분리

💭 객체 관점에서 생각하기

1. 좀 더 추상적인 인터페이스

객체의 자율성을 보장하기 위해 추상적인 인터페이스 설계

2. 최소 인터페이스

외부에서 사용할 필요 없는 인터페이스는 최대한 노출하지 말 것

객체 내부 동작에 대해 가능한 적게 노출할 것

3. 인터페이스 - 구현 간 차이 인식

➡️ SOON

🏭 구현

- 객체의 상태와 메서드는 공용 인터페이스의 일부?
  - 🙅‍♀️ 이들은 구현에 해당한다.
- 객체의 외부와 내부를 분리하라 ?
  - `공용 인터페이스`와 `구현`을 분리하라

🦹 인터페이스와 구현의 분리 원칙

객체를 설계할 때 객체 외부에 노출되는 인터페이스와 내부에 숨겨지는 구현을 명확히 분리해야 한다.

- 소프트웨어는 항상 변경된다.
- 모든 것이 공개되면 작은 부분의 수정도 영향력이 있다.
- 변경에 대한 안전 지대를 만드는 것은 객체의 자율성을 높인다.
- 상태와 메서드 구현을 수정하더라도 외부에 영향을 미치면 안된다.
- 인터페이스와 구현의 분리는 송신자와 수신자가 느슨한 인터페이스에 대해서만 결합되도록 만든다.

- 외부 객체는 공용 인터페이스에만 의존하고 구현에 의존해서는 안된다.

💊 캡슐화

객체의 자율성을 보존하기 위해 구현을 외부로부터 감추는 것

상태와 행위의 캡슐화

- 데이터 캡슐화
- 상태와 행위를 묶고 외부에서 접근해야 하는 행위만 골라서 공용 인터페이스로 노출한다.
- 인터페이스와 구현을 분리하기 위한 전제조건

사적인 비밀의 캡슐화

- 외부 객체가 내부 상태를 직접 보거나 제어할 수 없게 막고 의사소통 가능한 특별한 경로만 노출한다.
- 구현과 관련된 세부 사항은 캡슐화를 통해 안정적인 인터페이스 뒤로 숨긴다.

책임의 자율성이 협력의 품질을 결정한다.

객체의 책임이 자율적일수록 협력이 이해하기 쉽고 변경에 유연해진다.

- 자율적인 책임은 협력을 단순하게 만든다.
- 자율적인 책임은 객체의 외부와 내부를 명확하게 분리한다. (인터페이스-구현의 분리)
- (캡슐화) 책임이 자율적인 경우 책임을 수행하는 내부 방법을 변경하더라도 외부에 영향을 미치지 않는다.
- 자율적인 책임은 협력의 대상을 다양하게 선택할 수 있는 유연성을 제공한다.
- 객체가 수행하는 책임들이 자율적일수록 객체의 역할을 이해하기 쉬워진다.

🤔 책임이 자율적이라는 건 ...

객체지향 ? 다른 패러다임보다 우월. 😎

- 적절한 추상화
- 높은 응집도
- 낮은 결합도

- 캡슐화 증진
- 인터페이스와 구현의 명확한 분리
- 설계의 유연성과 재사용성