<객체지향 프로그래밍>

객체지향 프로그래밍(Object-Oriented Programming) 이란, 조립식 프로그래밍이다. 객체를 조립하여 전체 프로그램을 만드는 것이다.

- ◆ 장점 : 고장이 나도 해당 부분만 수리하면 되고, 기능을 추가할 때 만들어서 끼워 넣으면 된다.

클래스란, 객체를 만들기 위한 설계도이고, 해당 설계도를 통해 만들어진 것이 객체 또 다른 표현으로는 인스턴스라고 한다.

가장 중요한 점은 객체 내부에 자료형(필드)와 함수(메소드)가 같이 존재하는 것이다.

객체 지향으로 구현하게 되면, 객체 간의 독립성이 생기고 중복 코드의 양이 줄어드는 장점이 있다. 또한 독립성이 확립되면 유지보수에도 도움이 된다.

1. 특징

객체지향의 패러다임이 생겨나면서 크게 4가지 특징을 갖추게 되었다.

이 4가지 특성을 잘 이해하고 구현해야 객체를 통한 효율적인 구현이 가능해진다.

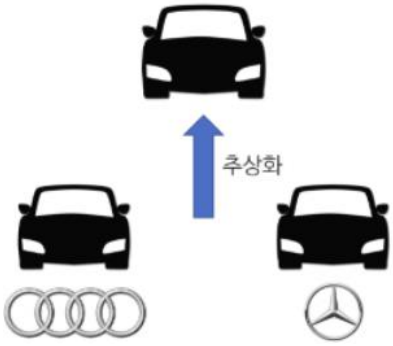
1) 추상화(Abstraction)

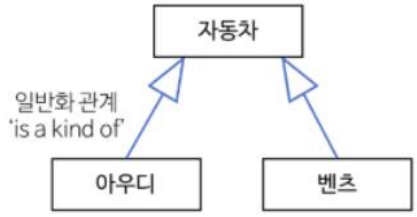
필요로 하는 속성이나 행동을 추출하는 작업

추상적인 개념에 의존하여 설계해야 유연함을 갖출 수 있다.

즉, **세부적인 사물들의 공통적인 특징을 파악한 후 하나의 집합으로 만들어내는 것**이 추상화다

ex. 아우디, BMW, 벤츠는 모두 '자동차'라는 공통점이 있다.





자동차라는 추상화 집합을 만들어 두고, 자동차들이 가진 공통적인 특징들을 만들어 활용한다.

'왜 필요하죠?'

예를 들면, '현대'와 같은 다른 자동차 브랜드가 추가될 수도 있다. 이때 추상화로 구현해두면 다른 곳의 코드는 수정할 필요 없이 추가로 만들 부분만 새로 생성해주면 된다.

2) 캡슐화(Encapsulation)

낮은 결합도를 유지할 수 있도록 설계하는 것

쉽게 말하면, **한 곳에서 변화가 일어나도 다른 곳에 미치는 영향을 최소화 시키는 것**을 말한다.

(객체가 내부적으로 기능을 어떻게 구현하는지 감추는 것!)

결합도가 낮도록 만들어야 하는 이유가 무엇일까? 결합도(coupling)란, 어떤 기능을 실행할 때 다른 클래스나 모듈에 얼마나 의존적인가를 나타내는 말이다.

즉, 독립적으로 만들어진 객체들 간의 의존도가 최대한 낮게 만드는 것이 중요하다. 객체들 간의 의존도가 높아지면 굳이 객체 지향으로 설계하는 의미가 없어진다.

우리는 소프트웨어 공학에서 객체 안의 모듈 간의 요소가 밀접한 관련이 있는 것으로 구성하여 응집도를 높이고 결합도를 줄여야 요구사항 변경에 대처하는 좋은 설계 방법이라고 배운다.

이것이 바로 캡슐화와 크게 연관된 부분이라고 할 수 있다.

그렇다면, 캡슐화는 어떻게 높은 응집도와 낮은 결합도를 갖게 할까?

바로 **정보 은닉**을 활용한다.

외부에서 접근할 필요가 없는 것들은 private으로 접근하지 못하도록 제한을 두는 것이다.

(객체안의 필드를 선언할 때 private으로 선언하라는 말이 바로 이 때문!!)

3) 상속

일반화 관계(Generalization)라고도 하며, 여러 개체들이 지닌 공통된 특성을 부각시켜 하나의 개념이나 법칙으로 성립하는 과정

일반화(상속)은 또 다른 캡슐화다.

자식 클래스를 외부로부터 은닉하는 캡슐화의 일종이라고 말할 수 있다.

아까 자동차를 통해 예를 들어 추상화를 설명했었다. 여기에 추가로 대리 운전을 하는 사람 클래스가 있다고 생각해보자. 이때, 자동차의 자식 클래스에 해당하는 벤츠, BMW, 아우디 등은 캡슐화를 통해 은닉해둔 상태다.

사람 클래스의 관점으로는, 구체적인 자동차의 종류가 숨겨져 있는 상태다. 대리 운전자 입장에서는 자동차의 종류가 어떤 것인지는 운전하는데 크게 중요하지 않다.

새로운 자동차들이 추가된다고 해도, 사람 클래스는 영향을 받지 않는 것이 중요하다. 그러므로 캡슐화를 통해 사람 클래스 입장에서는 확인할 수 없도록 구현하는 것이다.

이처럼, 상속 관계에서는 단순히 하나의 클래스 안에서 속성 및 연산들의 캡슐화에 한정되지 않는다. 즉, 자식 클래스 자체를 캡슐화하여 '사람 클래스'와 같은 외부에 은닉하는 것으로 확장되는 것이다.

이처럼 자식 클래스를 캡슐화해두면, 외부에선 이러한 클래스들에 영향을 받지 않고 개발을 이어갈 수 있는 장점이 있다.

- **상속 재사용의 단점**

상속을 통한 재사용을 할 때 나타나는 단점도 존재한다.

- 상위 클래스(부모 클래스)의 변경이 어려워진다.

  부모 클래스에 의존하는 자식 클래스가 많을 때, 부모 클래스의 변경이 필요하다면?

  이를 의존하는 자식 클래스들이 영향을 받게 된다.

- 불필요한 클래스가 증가할 수 있다.

  유사기능 확장 시, 필요 이상의 불필요한 클래스를 만들어야 하는 상황이 발생할 수 있다.

- 상속이 잘못 사용될 수 있다.

  같은 종류가 아닌 클래스의 구현을 재사용하기 위해 상속을 받게 되면, 문제가 발생할 수 있다. 상속받는 클래스가 부모 클래스와 IS-A 관계가 아닐 때 이에 해당한다.

- **해결책**

객체 조립(Composition), 컴포지션이라고 부르기도 한다.

객체 조립은, **필드에서 다른 객체를 참조하는 방식**으로 구현된다.

상속에 비해 비교적 런타임 구조가 복잡해지고, 구현이 어려운 단점이 존재하지만 변경 시 유연함을 확보하는데 장점이 매우 크다.

따라서 같은 종류가 아닌 클래스를 상속하고 싶을 때는 객체 조립을 우선적으로 적용하는 것이 좋다.

- **그럼 상속은 언제 사용할까?**
  - IS-A 관계가 성립할 때
  - 재사용 관점이 아닌, 기능의 확장 관점일 때

4) 다형성(Polymorphism)

서로 다른 클래스의 객체가 같은 메시지를 받았을 때 각자의 방식으로 동작하는 능력

객체 지향의 핵심과도 같은 부분이다.

다형성은, 상속과 함께 활용할 때 큰 힘을 발휘한다. 이와 같은 구현은 코드를 간결하게 해주고, 유연함을 갖추게 해준다.

즉, **부모 클래스의 메소드를 자식 클래스가 오버라이딩해서 자신의 역할에 맞게 활용**하는 것이 다형성이다.

이처럼 다형성을 사용하면, 구체적으로 현재 어떤 클래스 객체가 참조되는 지는 무관하게 프로그래밍하는 것이 가능하다.

상속 관계에 있으면, 새로운 자식 클래스가 추가되어도 부모 클래스의 함수를 참조해오면 되기 때문에 다른 클래스는 영향을 받지 않게 된다.

- **객체 지향 설계 과정**
  - 제공해야 할 기능을 찾고 세분화한다. 그리고 그 기능을 알맞은 객체에 할당한다.
  - 기능을 구현하는데 필요한 데이터를 객체에 추가한다.
  - 그 데이터를 이용하는 기능을 넣는다.

- 기능은 최대한 캡슐화하여 구현한다.
- 객체 간에 어떻게 메소드 요청을 주고받을 지 결정한다.

-

2. 객체 지향 설계 원칙

SOLID라고 부르는 5가지 설계 원칙이 존재한다.

1) SRP(Single Responsibility) - 단일 책임 원칙

클래스는 단 한 개의 책임을 가져야 한다.

클래스를 변경하는 이유는 단 한 개여야 한다.

이를 지키지 않으면, 한 책임의 변경에 의해 다른 책임과 관련된 코드에 영향이 갈 수 있다.

2) OCP(Open-Closed) - 개방-폐쇄 원칙

확장에는 열려 있어야 하고, 변경에는 닫혀 있어야 한다.

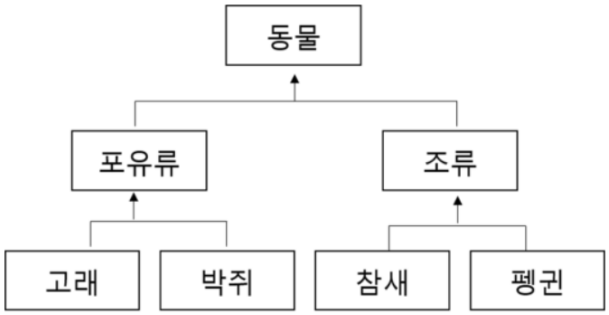
기능을 변경하거나 확장할 수 있으면서, 그 기능을 사용하는 코드는 수정하지 않는다.

3) LSP(Liskov Substitution) - 리스코프 치환 원칙

상위 타입의 객체를 하위 타입의 객체로 치환해도, 상위 타입을 사용하는 프로그램은 정상적으로 동작해야 한다.

상속 관계가 아닌 클래스들을 상속 관계로 설정하면, 이 원칙이 위배된다.

- 리스코프 치환 원칙 적용 사례 : 분류도



결국 리스코프 치환 원칙은 객체 지향의 상속이라는 특성을 올바르게 활용하면 자연스럽게 얻게 되는 것이다.

4) ISP(Interface Segregation) - 인터페이스 분리 원칙

인터페이스는 그 인터페이스를 사용하는 클라이언트를 기준으로 분리해야 한다.

각 클라이언트가 필요로 하는 인터페이스들을 분리함으로써, 각 클라이언트가 사용하지 않는 인터페이스에 변경이 발생하더라도 영향을 받지 않도록 만들어야 한다.

5) DIP(Dependency Inversion) - 의존 역전 원칙

고수준 모듈은 어떤 의미있는 단일 기능을 제공하는 모듈이고, 저수준 모듈은 고수준 모듈의 기능을 구현하기 위해 필요한 하위 기능의 실제 구현하는 모듈이다.

고수준 모듈은 저수준 모듈의 구현에 의존해서는 안된다.

저수준 모듈이 고수준 모듈에서 정의한 추상 타입에 의존해야 한다.

즉, 저수준 모듈이 변경돼도 고수준 모듈은 변경할 필요가 없는 것이다.

Reference

https://github.com/gyoogle/tech-interview-for-developer/blob/master/Computer%20Science/Software%20Engineering/Object-Oriented%20Programming.md