객체 지향 프로그래밍과 SOLID 원칙

김가영

INDEX

1

객체 지향적 프로그래밍

OOP
(Object Oriented Programming)

1

OOP

객체 지향적 프로그래밍이란?

컴퓨터 프로그래밍 패러다임 중 하나.

필요한 데이터를 **추상화**시켜 여러 개의 **객체**를 만들고,
여러 단위로 나누어진 객체들이
유기적으로 **상호작용해**
로직을 구성하는 프로그래밍 이론

1

OOP

절차 지향 프로그래밍의 문제를 개선

- 절차 지향 프로그래밍의 문제 :

- **절차와 데이터가 분리**되어 있어 매개 변수를 통해 데이터를 함수에 전달한다.

- 객체 지향 프로그래밍은 **클래스를 이용해 데이터와 절차를 묶어** 한 개의 자료형으로 취급할 수 있다.
- 코드의 재사용성 측면에서 유리하다.

1

OOP

객체 지향 프로그래밍의 특성

추상화

캡슐화

상속

다형성

1

OOP

객체 지향 4대 특성

1. 추상화

추상화란, **객체의 공통적인 속성과 기능을 추출하여 정의**하는 것

실제로 존재하는 객체들을 프로그램으로 만들기 위하여
공통 특성을 파악해 **필요 없는 특성을 제거**하는 과정

쉽게 말해 **Modeling**을 의미

1

OOP

객체 지향 4대 특성

2. 캡슐화

캡슐화란, **접근 제어자**를 사용해
데이터와 코드의 형태를 **외부로부터 알 수 없게** 하여
사용자가 사용할 수 있는 부분만 보이도록 하고 나머지는 내부에
감추는 것

객체의 속성을 보호하기 위해 사용. **(정보 은닉)**

1

OOP

객체 지향 4대 특성

캡슐화로 인해 얻을 수 있는 장점

- 객체의 오용 방지

- 재사용성, 유지 보수 효율성 향상

- 객체간의 결합도가 낮아진다.

- 데이터에 쉽게 접근할 수 없고 외부에서 감춰져 있으므로 데이터의 무결성 보장

1

OOP

객체 지향 4대 특성

3. 상속

상속이란, 상위 클래스에 있는 **기존 기능을 재사용할 수도 있고,**

상위 클래스와 상속 관계에 있는 **하위 클래스에 새로운 기능을 추가할 수 있는 것**
(객체 지향에서의 상속은 하위로 갈수록 구체화 됨)

1

OOP

객체 지향 4대 특성

상속으로 인해 얻을 수 있는 장점

- 재사용성 향상
- 확장성 향상
- 유지 보수 효율성 향상

1

OOP

객체 지향 4대 특성

4. 다형성

다형성이란, 한 객체가 **기능을 확장하거나 변경**하여

다른 형태로 재구성 되는 것

1

OOP

객체 지향 4대 특성

overriding, overloading 등을 통해 구현할 수 있다.

다형성으로 인해 얻을 수 있는 장점

- 부모 클래스로 선언된 변수에서 자식 클래스에서 오버라이딩 된 함수를 호출할 수 있다.
- 변화에 유연한 소프트웨어를 만들 수 있다

1

OOP

객체 지향적 프로그래밍

장점

- 코드 재사용에 용이
- 보안성 향상
- 자연적인 모델링
- 유지 보수가 편리하다

1

OOP

객체 지향적 프로그래밍

단점

- 설계 시 많은 시간과 노력 필요
- 상대적으로 느린 처리(실행) 속도
- 객체가 많으면 용량이 커질 수 있다

1

OOP

정리

객체 지향 프로그래밍의 구현 방식을 종합하면,

어떤 객체들을 추상화하여 공통점을 찾고,
그것을 클래스에 캡슐화하여 한 곳으로 모은다.

이를 새로운 클래스가 상속받아 재사용이 가능하며,

상속받은 클래스는 다형성을 통해 기능을 수정하거나 추가할 수 있다.

2

객체 지향 설계 5대 원칙

SOLID 원칙

2

SOLID

객체 지향 설계 5대 원칙

- 시간이 지나도 **유지보수**에 좋고, **확장성**이 뛰어난 시스템을 만들기 위해 적용시킬 수 있는 원칙
- 객체지향적인 프로그래밍을 위해 지켜야 하며 이를 통해 객체 간의 **응집도는 높이고, 결합도는 낮출 수 있다.**

SOLID

객체 지향 설계 5대 원칙의 앞글자를 따
SOLID 원칙이라고도 한다.

Single Responsibility Principle - 단일 책임 원칙

Open-Closed Principle - 개방 폐쇄 원칙

Liskov Subsitution Principle - 리스코프 치환 원칙

Interface Segregation Principle - 인터페이스 분리 원칙

Dependency Inversion Principle - 의존성 역전 원칙

2

SOLID

1. Single Responsibility Principle

단일 책임 원칙

의미

**모든 클래스(모듈)들은 각각
한 가지의 역할만 가져야 한다.**

**즉, 어떤 클래스를 변경해야 하는 이유는
오직 하나 뿐이어야 한다.**

2

SOLID

1. Single Responsibility Principle

단일 책임 원칙

단일 책임 원칙을 만족하는 방법

클래스의 역할을 한 가지로 구성한다.

-> 클래스 역할을 한 가지로 구성하면 모듈의 **응집도를 높일 수 있다**.

클래스(모듈)들은 그 책임을 캡슐화 시켜야 한다.

-> 관련있는 변수와 함수를 하나의 클래스로 묶고,
외부에서 쉽게 접근하지 못하도록 은닉하여 **접근을 제한**

SRP**를 만족하는 프로그램의 장점** : 유지보수나 확장 시 오류 최소화,
객체 내의 정보 손상과 오용을 방지, 데이터 독립성 증가.

2

SOLID

1. Single Responsibility Principle

단일 책임 원칙

클래스의 역할이 한 가지로 구성된다면,
클래스를 변경해야 하는 이유도 한 가지이다.

-> 한 클래스가 여러 역할에 대해 책임을 가지고 있다면,
해당 클래스는 여러 역할들에 의해 변경 요구가 생길 수 있어
수정하는 이유가 여러 가지가 될 수 있다.

-> 클래스가 가진 역할이 여러 가지라면, 한 클래스가 가진 **책임이 커지고,**
클래스 내부 함수끼리 **결합도가 높아지게** 되어
유지 보수 비용이 증가하게 되는 등, 변화에 손쉽게 대응할 수 없게 만든다.

2

SOLID

1. Single Responsibility Principle

단일 책임 원칙

장점

- 클래스 당 맡은 한 가지 역할에 대해서만 수정하면 되므로 확장성과 유지보수성 증가
- 적절하게 책임과 관심이 다른 코드를 분리하고, 서로 영향을 주지 않도록 추상화함으로써 애플리케이션의 변화에 손쉽게 대응 가능
- 시스템이 커질 수록 서로 많은 의존성을 갖게 되므로 위의 장점 극대화

2

SOLID

2. Open-Closed Principle

개방 폐쇄 원칙

의미

확장에는 열려있고,
수정에는 닫혀있어야 한다.

즉, 기본 코드를 변경하지 않으면서
기능을 추가할 수 있도록
설계되어야 한다는 원칙이다.

2

SOLID

2. Open-Closed Principle

개방 폐쇄 원칙

“ 확장에 대해 열려 있다.”

-> 요구사항이 변경될 때 새로운 동작을 추가하여
애플리케이션의 기능을 확장할 수 있어야 한다.

“ 수정에 대해 닫혀 있다.”

-> 기존의 코드를 수정하지 않고
애플리케이션의 동작을 추가하거나 변경할 수 있어야 한다.

자주 변경되는 내용은 수정하기 쉽게 설계 하고, **변경되지 않아야 하는 것은 수정되는 내용에 영향을 받지 않게** 하는 것

2

SOLID

2. Open-Closed Principle

개방 폐쇄 원칙

- 개방 폐쇄 원칙을 **지키지 않는다면**, 새로운 로직을 추가해야 하는 상황에서 새로운 **로직과 무관한 코드까지 수정**해야 하는 일이 생길 수 있다.

- 개방 폐쇄 원칙을 지키기 위해서는 **추상화**에 의존해야 한다. 변하지 않는 부분은 고정하고 변하는 부분을 생략하여 추상화함으로써, 변경이 필요한 경우 생략한 부분을 수정해 개방 폐쇄 원칙을 지켜야 한다.

- 개방 폐쇄 원칙을 만족했을 때의 **장점** :

  기존의 코드 및 클래스들을 수정하지 않은 채로 애플리케이션을 확장할 수 있다. 코드의 **유연성, 재사용성, 유지보수성이 증가**한다.

3. Liskov Subsitution Principle

리스코프 치환 원칙

의미

자식 클래스는 언제나
부모 클래스를 대체할 수 있어야 한다.

즉, 부모 클래스의 인스턴스 대신
자식 클래스의 인스터스를 사용해도
문제가 없어야 한다

SOLID

3. Liskov Subsitution Principle

리스코프 치환 원칙

- **상속**에 관한 원칙.
- 리스코프 치환 원칙은 개방 폐쇄 원칙을 받쳐주는 다형성에 관한 원칙을 제공한다. 따라서 리스코프 치환 원칙을 제대로 지키지 않으면, 다형성에 기반한 **개방 폐쇄 원칙을 위반**할 수 있으며, 프로그램에 모순이 생길 수 있다.
- 일반적으로 부모와 자식 클래스 사이의 상속 관계를 선언하기 위해서는 **일반화 관계(IS-A)**, 즉 일관성 있는 관계가 성립해야 한다.
- 리스코프 치환 원칙을 제대로 지키지 않으면, 자식 클래스가 부모 클래스의 **메소드를 오버라이딩을 하는 경우** 또는 **잘못된 상속을 했을 경우**에 문제가 될 수 있다. 해당 경우에는 상위 클래스의 의도에 맞게 오버라이딩을 바로 하거나 상속 관계를 끊어야 한다.

4. Interface Segregation Principle

인터페이스 분리 원칙

의미

**자신이 사용하지 않는 인터페이스는
구현하지 말아야 한다.**

**즉, 자신이 사용하지 않는 기능
(인터페이스)에는 영향을 받지 말아야 한다.**

SOLID

4. Interface Segregation Principle

인터페이스 분리 원칙

- 클래스의 책임을 덜어주고, **기능에 의존하지 않게** 해주는 원칙이다.
- 많은 추상 메소드를 가진 거대한 인터페이스 하나를, **관련된 추상 메소드들만 모여있도록 작은 크기의 인터페이스로 분리하는 방식으로 구현**
- 인터페이스 분리 원칙을 지켜 설계된 소프트웨어는 **시스템의 내부 의존도가 낮아져 확장성이 높아지고 리팩토링, 수정, 재배포에 용이**하다.
- 하나의 일반적 인터페이스보다 여러개의 구체적인 인터페이스가 객체 지향석 설계에 가깝다.

SOLID

4. Interface Segregation Principle

인터페이스 분리 원칙

- 인터페이스 분리 원칙은 모든 인터페이스를 다 분리하라는 것이 아니다.
- 너무 작아진 인터페이스는 자식 클래스가 **너무 많은 부모 클래스를 상속** 받아야 해서 **문제**가 생긴다.
- 인터페이스를 분리할 때 어떤 **기능이나 역할을 중심**으로 서로 **관련 있는 추상 메소드들은 모으고 관련 없는 추상 메소드들은 분리시키는 방식으로 구현해야 한다**.
- 인터페이스 분리 원칙을 지켜 설계한 프로그램은 각 클래스가 **본인의 역할에 맞는 인터페이스를 상속** 받을 수 있게 하여 더 **효율적이다**.
- 또한 각 클래스의 **기능을 쉽게 파악**할 수 있어 **코드의 가독성**을 높인다.

5. Dependency Inversion Principle

의존성 역전 원칙

의미

의존 관계를 맺을 때
변화 가능성이 낮은 것에 의존해야 한다.

즉, 변하기 쉬운 것의 변화에
영향을 받지 않게 해야 한다.

SOLID

5. Dependency Inversion Principle

의존성 역전 원칙

- 의존성 역전 원칙을 만족했을 때의 장점:
  - '**의존성 주입(DI)**' 이라는 기술로 **변화에 유연한 설계**를 할 수 있다.
  - 의존성이 줄어든다.
  - 재사용성이 높은 코드가 된다.
  - 가독성이 높아진다.
- **상위 클래스일수록, 인터페이스일수록, 추상 클래스일수록** 변하지 않을 가능성이 높다.
- 변하기 쉬운 클래스의 구현 내용에 의존하면 안 되고 변하지 않을 가능성이 높은 추상화된 클래스에 의존해야 한다.
- **변하기 쉬운 클래스에 의존하게 된다면,** 의존하는 클래스에 변화가 있을 때마다 **코드를 수정해야 하는 번거로움이 생긴다.**

3

정리

개발자가 SOLID 원칙을 지켜서 프로그램을 설계한다면,

시간이 지나도 **변경이 용이**하고, **유지보수와 확장이 쉬운** 소프트웨어를 개발할 수 있다.

객체 지향적 프로그래밍의 장점을 최대화 시키기 위해 반드시 SOLID원칙을 알맞게 고려하여 설계해야 한다.

감사합니다.