SOLID 객체지향 설계 원칙

목차

---

1. 단일 책임 원칙 (Single Responsibility Principle, SRP)

- 1-1. 정의
  - 한 클래스는 하나의 책임만 가져야 하고 클래스가 변경되어야 할 이유는 오직 하나여야 한다.

▼ 1-2. SRP 위반 예제

```
// SRP 위반 예시
class Employee {
    private String name;
    private double hourlyRate;
    private int hoursWorked;

    public Employee(String name, double hourlyRate, int hoursWorked) {
        this.name = name;
        this.hourlyRate = hourlyRate;
        this.hoursWorked = hoursWorked;
    }

    // 직원 정보 출력 (책임 1)
    public void printEmployeeInfo() {
        System.out.println("직원 이름: " + name);
        System.out.println("시급: " + hourlyRate);
        System.out.println("근무 시간: " + hoursWorked);
    }

    // 급여 계산 (책임 2)
    public double calculatePay() {
        return hourlyRate * hoursWorked;
    }
}

public class SRPViolationDemo {
    public static void main(String[] args) {
        Employee emp = new Employee("홍길동", 15000, 40);
        emp.printEmployeeInfo();
        System.out.println("이번 주 급여: " + emp.calculatePay() + "원");
    }
}
```

- class Employee

    - 직원 객체
    - 필드: name(이름), hourlyRate(시급), hoursWorked(근무 시간)
    - private
        - 외부에서 직접 접근할 수 없고, 클래스 내부 메서드를 통해서만 다룰 수 있음
- public Employee
    - 생성자
    - Employee 객체를 만들 때 이름/시급/근무시간을 초기화
    - this.name = name;
        - 매개변수와 필드 이름이 같을 때 자기 자신을 구분하는용도
- public void printEmployeeInfo() + public double calculatePay()
    - 직원 정보를 콘솔에 출력하는 기능 즉 출력(UI/프레젠테이션) 책임과 비즈니스 로직(급여 계산) 책임이 클래스 안에 포함되어있다.
    - Employee 클래스가 3가지 역할을 동시에 하게됨
        - 직원 데이터 보관
        - 출력 (UI 관련 로직)
        - 급여 계산 (비즈니스 로직)

▼ 1-3. Employee 클래스가 3가지 역할을 동시에 하게되는게 문제가되는이유?

- 기능이 섞여 있으면 한 영역을 고치면서 다른 영역에 코드 충돌 위험이 커진다.
    - 콘솔 대신 파일 출력으로 바꾸려고 printEmployeeInfo()를 수정했는데, 그 과정에서 calculatePay() 관련 필드를 잘못 건드려 급여 계산이 틀려버릴 수 있다
- 작은 변경에도 클래스 전체를 건드려야해서 유지보수성이 떨어진다.
    - 출력 로직 바꾸려고 Employee 클래스를 열면, 급여 계산과 데이터 관리 코드까지 다 보여서 코드 접근에 대한 불확실성이 커지고 불필요하게 큰리스크를 안게된다.
- 결론
    - 3가지 이유로 고친다고 해서 실행이 안되는 건 아니지만 실무에서는 한 클래스에는 하나의 책임만 부여하는 것이 장기적으로 안전하고 효율적이다.

2. 개방-폐쇄 원칙 (Open-Closed Principle, OCP)

- 2-1. 정의
    - 소프트웨어 요소(클래스, 모듈, 함수 등)는 확장에는 열려 있어야 하고, 수정에는 닫혀 있어야 한다.

▼ 2-2. OCP 위반 예제

```
// OCP 위반: 새로운 동물이 추가될 때마다 AnimalSound 클래스를 수정해야 함
class Animal {}

class Dog extends Animal {}
class Tiger extends Animal {}
class Cat extends Animal {}

class AnimalSound {
   public void makeSound(Animal animal) {
      if (animal instanceof Dog) {
         System.out.println("멍멍");
      } else if (animal instanceof Tiger) {
         System.out.println("어흥");
```

```
        } else if (animal instanceof Cat) {
            System.out.println("야옹");
        } else {
            System.out.println("알 수 없는 동물 소리");
        }
    }
}

public class OCPViolationExample {
    public static void main(String[] args) {
        AnimalSound soundMaker = new AnimalSound();

        soundMaker.makeSound(new Dog());   // 멍멍
        soundMaker.makeSound(new Tiger()); // 어흥
        soundMaker.makeSound(new Cat());   // 야옹
    }
}
```

- class Animal
  - 속성이나 메서드가 정의되지않음 (단순히 “동물”이라는 개념을 표현하기 위한 빈 클래스)
    - 구체 동물의 공통 행동을 추상화하지못함.
- class Dog extends Animal {}, class Tiger extends Animal {}, class Cat extends Animal {}
  - Animal을 상속받은 자식 클래스
  - 고유의 행동(makeSound)이 직접 구현되지 않음
    - “소리” 기능 같은 책임을 스스로 갖고 있지 않고, 나중에 다른 클래스(AnimalSound)에서 대신 처리
- class AnimalSound
  - 모든 자식 클래스의 “소리” 기능을 처리하는 클래스
    - 동물 객체가 전달되면, instanceof 연산자를 사용해 그 동물이 어떤 타입인지 검사하고 소리 출력.
  - 만약 새로운 동물 Horse 클래스를 추가하면?
    - AnimalSound 클래스의 makeSound 메서드 내부를 수정해서 else if (animal instanceof Horse) 같은 분기를 또 넣어야 함
    - 확장(새로운 동물 추가) 은 가능하지만, 수정(AnimalSound 변경)이 항상 따라오므로 OCP 원칙에 위배된다.

▼ 2-3. class AnimalSound에서 수정이 항상 따라온다는것의 의미?

- 새로운 동물이 추가될 때마다 기존에 이미 잘 동작하고 있던 AnimalSound 클래스의 코드를 바꿔야 한다.
  - 그러면 OCP의 핵심인 ‘새로운 기능은 추가할 수 있지만, 기존 기능은 그대로 두어야 한다’가 위배된다.
  - 새로운 동물을 추가해도 AnimalSound라는 기존 클래스의 내부 코드를 건드리지 않아도 되는 다음과 같은 형태여야 한다.

```
// OCP 준수 예시
interface Animal {
    void makeSound();
}

class Dog implements Animal {
    public void makeSound() { System.out.println("멍멍"); }
}

class Tiger implements Animal {
```

```
    public void makeSound() { System.out.println("어흥"); }
}

// 새로운 동물 추가 시 AnimalSound 수정할 필요 없음
class Horse implements Animal {
    public void makeSound() { System.out.println("히이잉"); }
}

class AnimalSound {
    public void makeSound(Animal animal) {
        animal.makeSound(); // 다형성 활용
    }
}

public class OCPCompliant {
    public static void main(String[] args) {
        AnimalSound soundMaker = new AnimalSound();
        soundMaker.makeSound(new Dog());
        soundMaker.makeSound(new Tiger());
        soundMaker.makeSound(new Horse()); // 기존 코드 수정 없이 확장 가능
    }
}
```

→ 이렇게 만들면 Horse 같은 새로운 동물이 추가되더라도 AnimalSound는 전혀 수정하지 않고 그대로 재사용할수 있음.

3. 리스코프 치환 원칙 (Liskov Substitution Principle, LSP)

- 3-1. 정의
  - 자식 클래스는 언제나 부모 클래스를 대체할 수 있어야 한다. 상속 관계에서 부모 타입으로 선언된 객체 자리에 자식 객체를 넣어도 프로그램이 정상적으로 동작해야 한다.

▼ 3-2. LSP 위반 예제

```
// LSP 위반 사례
class Bird {
    void fly() {
        System.out.println("새가 날아갑니다!");
    }
}

class Penguin extends Bird {
    @Override
    void fly() {
        // 펭귄은 날 수 없는데도 부모의 fly()를 오버라이드해야 함
        throw new UnsupportedOperationException("펭귄은 날 수 없어요!");
    }
}

public class LSPViolationExample {
    public static void main(String[] args) {
        Bird bird1 = new Bird();
        Bird bird2 = new Penguin(); // 부모 타입에 자식 객체 할당

        bird1.fly(); // 정상 동작
        bird2.fly(); // ❌ 실행 시 예외 발생 → LSP 위반
```

```
    }
}
```

- class Bird
  - 새를 추상화한 기본 클래스 (부모 클래스)
  - void fly() { System.out.println("새가 날아갑니다!"); }
    - 모든 Bird가 fly()를 가지도록 강제
- class Penguin extends Bird
  - Bird를 extends 해서 상속받은 자식 클래스.
  - 부모 클래스의 모든 메서드를 그대로 물려받게 되는데, 여기에는 fly()도 포함됨. 하지만 펭귄은 날 수 없으므로 fly()를 오버라이드하여 UnsupportedOperationException 처리를 함.
    - 부모 클래스가 정의한 계약을 지킬 수 없음. 부모가 "모든 Bird는 fly() 할 수 있다"고 했는데, 자식인 Penguin은 이 약속을 어기게 됨.
- public class LSPViolationExample
  - 실행 메인 클래스
  - Bird bird2 = new Penguin();
    - 부모 타입(Bird) 변수에 자식 객체(Penguin)를 넣음. LSP의 전제가 자식은 부모를 대체할 수 있어야 한다니까 가능해야함.
    - bird2의 타입은 Bird → 당연히 fly() 호출 가능해야 하는데 실제 객체는 Penguin 이라서 실행 시 UnsupportedOperationException 처리됨.
    - 부모 클래스의 계약(fly() 가능하다)을 자식 클래스가 어기므로 LSP 위반이다.

▼ 3-3. Bird bird2 = new Penguin();에서 펭귄 객체에 '인스턴스와 객체의 분리'가 어떻게 적용되는가? + 컴파일 시점 타입과 실행 시점 타입의 차이가 어떻게 LSP 위반으로 이어지는가?

- 객체는 클래스라는 설계도로부터 생성된 실체. 즉 new Penguin()으로 생성된 펭귄.
- 인스턴스는 어떤 클래스의 "구체적인 사례"라는 의미에서 바라본 객체. 즉 Penguin penguin = new Penguin();이면 penguin은?
  - "Penguin 클래스의 인스턴스"이자 동시에 "Bird 클래스의 인스턴스".
  - Penguin의 사례이자 Bird의 사례.
- 모든 인스턴스는 객체이지만 객체를 어떤 타입 관점에서 바라보느냐에 따라 인스턴스라고 부른다.
- Bird bird2 = new Penguin();에서
  - 실제로 만들어진 것은 Penguin 객체이고
  - 이 객체는 Penguin 클래스의 인스턴스임과 동시에 Penguin이 Bird를 상속했기 때문에 Bird 클래스의 인스턴스. 따라서 bird2라는 참조 변수는 Bird 타입을 기준으로 이 객체를 다룬다. 여기서 "타입은 Bird, 실제 객체는 Penguin" 이라는 분리가 발생한다.
- 컴파일 vs 실행
  - bird2 변수의 정적 타입(compile-time type) 은 Bird이므로 bird2.fly() 호출은 컴파일러가 허용한다. 하지만 실제 실행 시점(run-time type) 은 Penguin이므로 Penguin.fly() 가 실행되며 UnsupportedOperationException 처리된다 즉, Bird라는 부모 타입의 계약(fly() 가능하다) 은 Penguin 객체에서는 깨져버린다.
- 결론
  - 인스턴스와 객체의 분리는 "Bird 타입 인스턴스로서의 펭귄"이라는 다형성 상황을 만들어주지만 펭귄이 fly() 계약을 제대로 지키지 못하면서 LSP 위반이 발생했다.

4. 인터페이스 분리 원칙 (Interface Segregation Principle, ISP)

- 4-1. 정의

- 하나의 범용적인 큰 인터페이스보다는 여러 개의 구체적이고 작은 인터페이스로 나누는 것이 좋다.

▼ 4-2. ISP 위반 예제

```
// ISP 위반 예제
interface SmartMachine {
    void print();
    void fax();
    void scan();
}

// Printer는 사실 print만 필요하지만,
// 불필요한 fax(), scan()까지 억지로 구현해야 함
class Printer implements SmartMachine {
    @Override
    public void print() {
        System.out.println("문서를 출력합니다.");
    }

    @Override
    public void fax() {
        // 실제 프린터에는 필요 없는 기능
        System.out.println("❌ 프린터는 팩스를 지원하지 않습니다.");
    }

    @Override
    public void scan() {
        // 실제 프린터에는 필요 없는 기능
        System.out.println("❌ 프린터는 스캔을 지원하지 않습니다.");
    }
}

public class ISPViolationDemo {
    public static void main(String[] args) {
        SmartMachine printer = new Printer();
        printer.print();
        printer.fax();   // 의미 없는 기능 호출
        printer.scan();  // 의미 없는 기능 호출
    }
}
```

- interface SmartMachine
  - interface는 “이 클래스는 이런 기능을 반드시 제공해야 한다”라는 계약(Contract)을 정의
    - SmartMachine은 print(), fax(), scan()을 제공해야한다.
    - 이 인터페이스를 implements 하는 클래스는 세 메서드를 무조건 구현해야한다.
- class Printer implements SmartMachine
  - Printer는 SmartMachine을 구현해야 하므로 세 메서드를 모두 작성해야 해서 fax(), scan() 를 작성은 하되 안에서는 그냥 “지원하지 않는다”는 메시지만 출력.
  - Printer 클래스가 사용하지 않는 기능인데도 구현돼서 ISP 위반.

5. 의존 역전 원칙 (Dependency Inversion Principle, DIP)

- 5-1. 정의

- 고수준 모듈은 저수준 모듈에 의존하면 안 된다. 상위 비즈니스 로직이 하위 세부 구현에 직접 묶이지 않고, 추상화(인터페이스)에 의존해야 한다.

▼ 5-2. DIP 위반 예제

```
// 구체적인 구현체에 의존하는 예시 (DIP 위반)
class SnowTire {
    public void roll() {
        System.out.println("❄️ 눈길을 달리는 스노우 타이어");
    }
}

class Car {
    private SnowTire tire;

    public Car() {
        this.tire = new SnowTire(); // ❌ 특정 구현체에 직접 의존
    }

    public void drive() {
        tire.roll();
    }
}

public class DIPViolationExample {
    public static void main(String[] args) {
        Car car = new Car();
        car.drive();
    }
}
```

- class SnowTire
  - 타이어 구현체로써 roll() 메서드를 호출하면 단순히 콘솔에 "❄️ 눈길을 달리는 스노우 타이어"라는 메시지를 출력한다.
    - 이 클래스 자체가 인터페이스(추상화)가 아니라 구체적인 클래스이다.
- class Car
  - 자동차 구현체
- private SnowTire tire;
  - SnowTire tire;라는 구체 클래스 타입 필드를 가지고 있음.
- public Car() { this.tire = new SnowTire(); }
  - 생성자에서 new SnowTire()를 직접 만들어서 tire에 할당한다.
- public void drive()
  - drive() 메서드는 단순히 tire.roll()을 호출한다.

▼ 5-3. 이 코드가 DIP를 위반하는 이유? (고수준 모듈 vs 저수준 모듈)

- 의존 역전 원칙: 고수준 모듈은 저수준 모듈에 의존하지 말고, 둘 다 추상화에 의존해야 한다.
- 고수준 모듈인 Car는
  - “주행” = drive()이라는 목표만 있으면 되는데 new SnowTire()를 해 버리면서, 특정 부품인 스노우 타이어와 묶이게 됨
- 결론

- 이 코드가 DIP를 위반하는 이유는 고수준 모듈이 목적(drive())보다 수단(tire)에 자신을 종속시켜서.
- Car은 'Tire 인터페이스'에만 의존하고 실제 어떤 타이어를 쓸지는 외부에서 주입(Dependency Injection)받아야 한다.

▼ 5-4. "Car(고수준모듈)이 Tire 인터페이스에만 의존하고 실제 어떤 타이어를 쓸지는 외부에서 주입받아야 한다"의 의미?

- Tire 인터페이스
  - 타이어라면 반드시 roll() 기능을 제공해야 한다.
- 타이어
  - Tire 인터페이스를 지키면서 자기 방식대로 동작하는 타이어 (저수준 모듈/아래 코드에서 SnowTire, NormalTire)
- Car(고수준모듈)이 Tire 인터페이스에만 의존해야한다:

```
class SnowTire implements Tire {
    public void roll() {
        System.out.println("❄️ 눈길을 달리는 스노우 타이어");
    }
}

class NormalTire implements Tire {
    public void roll() {
        System.out.println("🚗 일반 도로를 달리는 일반 타이어");
    }
}
class Car {
    private Tire tire; // 인터페이스에만 의존

    public Car(Tire tire) { // 외부에서 주입
        this.tire = tire;
    }

    public void drive() {
        tire.roll(); // "굴러간다"는 사실만 사용
    }
}
```

  - 타이어 2종류: SnowTire, NormalTire
  - class Car에서 타이어 관련 코드를 보면 public Car(Tire tire) { // 외부에서 주입 this.tire = tire; } 니까 특정 타이어 종류랑 묶여있지 않음
- 실제 어떤 타이어를 쓸지를 외부에서 주입:

```
public class Main {
    public static void main(String[] args) {
        Car snowCar = new Car(new SnowTire());   // 겨울철엔 스노우 타이어
        snowCar.drive();

        Car normalCar = new Car(new NormalTire()); // 여름철엔 일반 타이어
        normalCar.drive();
    }
}
```

  - new Car(new SnowTire()) 에서 실제 어떤 타이어가 들어올지는 실행 시점에 외부에서 결정된다.
  - 그래서 Car는 본질적인 책임(주행)에만 집중할 수 있고 타이어의 종류가 바뀌어도 Car 클래스 자체는 수정할 필요가 없다.

- 결론
    - Car는 추상화(Tire 인터페이스)에만 의존하고, 구체적인 객체 생성과 선택은 외부(Main)에서 맡게 됨으로써 결합도를 낮추고 유연성을 확보한다.

6. 공통 특성: 응집도를 높이거나 결합도를 낮추는 설계

- SOLID 객체지향 설계 원칙은 모듈 간 결합도는 낮추고 각 모듈 내부의 응집도는 높여서 일관성있고 유연한 구조를 만드는게 목적.
    - SRP (단일 책임 원칙): 클래스가 한 가지 책임만 가지게 해서 응집도를 높임.
    - OCP (개방-폐쇄 원칙): 확장에는 열려 있고 변경에는 닫혀 있게 해서 코드 변경 없이 새로운 기능을 붙일 수 있게 해서 응집도를 유지하면서 변화에 유연하게 설계.
    - LSP (리스코프 치환 원칙): 부모 타입을 대체할 수 있는 자식 타입을 보장해서 결합도를 낮추면서 일관성 있게 설계.
    - ISP (인터페이스 분리 원칙): 불필요한 의존성을 줄이고 필요한 인터페이스만 사용하게 해서 결합도를 낮추고 응집도를 높임.
    - DIP (의존 역전 원칙): 고수준 모듈과 저수준 모듈이 추상화에 의존하도록 해서 결합도를 낮추고 응집도를 강화.