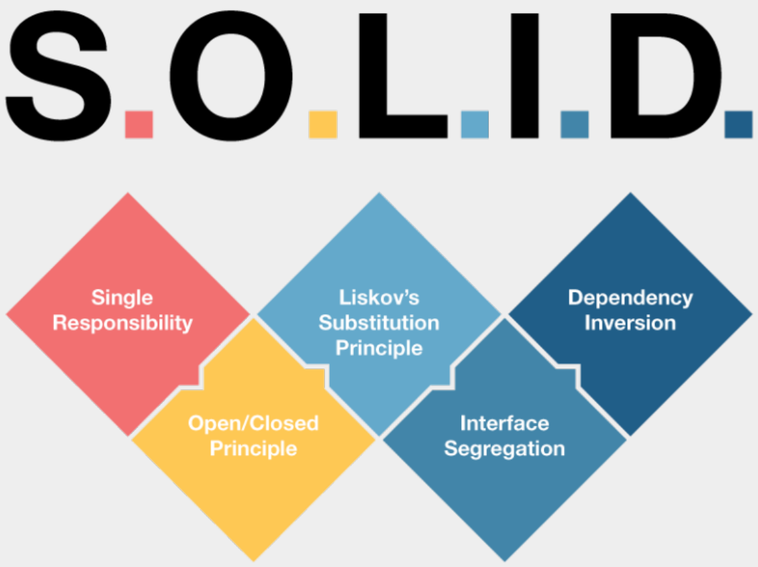
2022년 Wisoft 동계 워크샵

SOLID 원칙

20181745 윤진원

SOLID 원칙이란?

> 객체 지향 프로그래밍을 설계할 때, 유지보수와 확장이 쉬운 시스템을 만들고자 지향해야 할 5가지 원칙



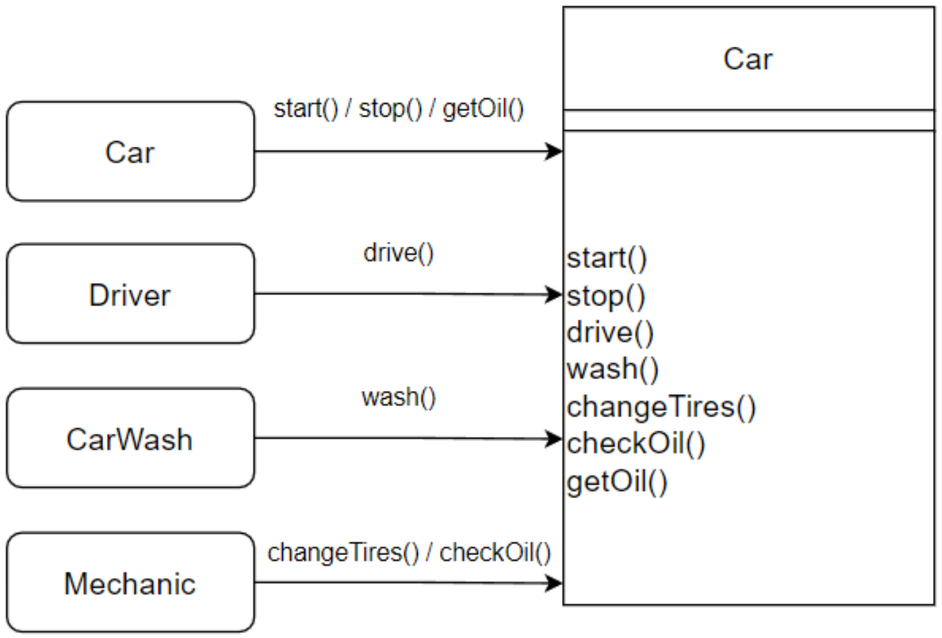
1. SRP(단일 책임 원칙)
2. OCP(개방-폐쇄 원칙)
3. LSP(리스코프 치환 원칙)
4. ISP(인터페이스 분리 원칙)
5. DIP(의존관계 역전 원칙)

SRP (Single Responsibility) : 단일 책임 원칙

> 클래스는 단 **한 개의 책임**을 갖고, 그 책임을 캡슐화해야 한다.
> 또한, **클래스를 변경하는 이유는 단 하나**여야 한다.

→ **메소드의 개수 ≠ 책임의 개수**

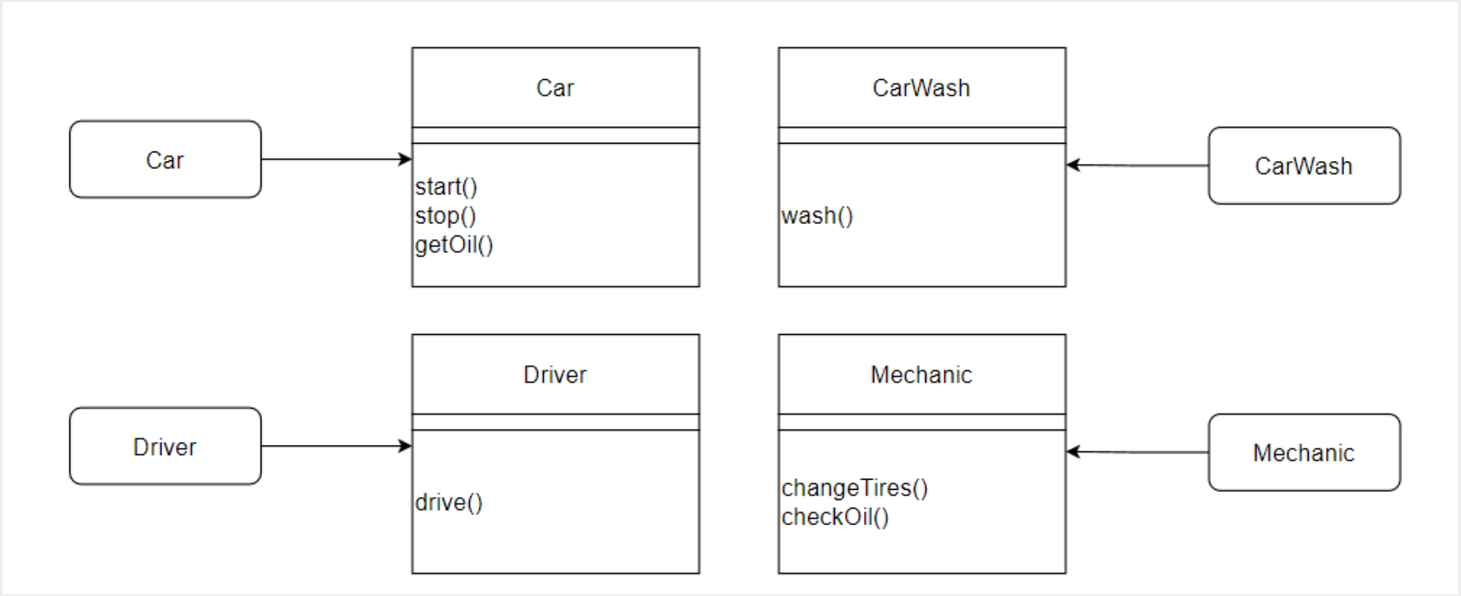
→ 클래스를 변경하는 이유가 한 가지이기 위해서는, 하나의 **액터**에 대한 책임만 가지고 있어야 한다.



적용 전 코드

```
class Car {

    private void start() { }
    private void stop() { }
    private void drive() { }
    private void wash() { }
    private void changeTires() { }
    private void checkOil() { }
    private void getOil() { }
}
```

SRP (Single Responsibility) : 단일 책임 원칙



단일 책임 원칙을 따랐을 때의 이점

- 하나의 책임만 가지고 있기에, 구현 및 이해가 쉽다.
- 변경의 연쇄작용에서 자유로울 수 있다.
- 유지보수가 용이해진다.

적용 후 코드

```
class Car {

    private void start() {}
    private void stop() {}
    private void getOil() {}
}

class Driver {

    private void drive() {}
}

class CarWash {

    private void wash() {}
}

class Mechanic {

    private void changeTires() {}
    private void checkOil() {}
}
```

OCP (Open-Closed) : 개방-폐쇄 원칙

> 소프트웨어 개체는 **확장**에 대해 열려 있어야 하고, **수정**에 대해서는 닫혀 있어야 한다.

→ 기능 추가 요청이 오면, 클래스를 **확장을 통해 손쉽게 구현**하면서, **확장에 따른 클래스 수정은 최소화** 하도록!

적용 전 코드

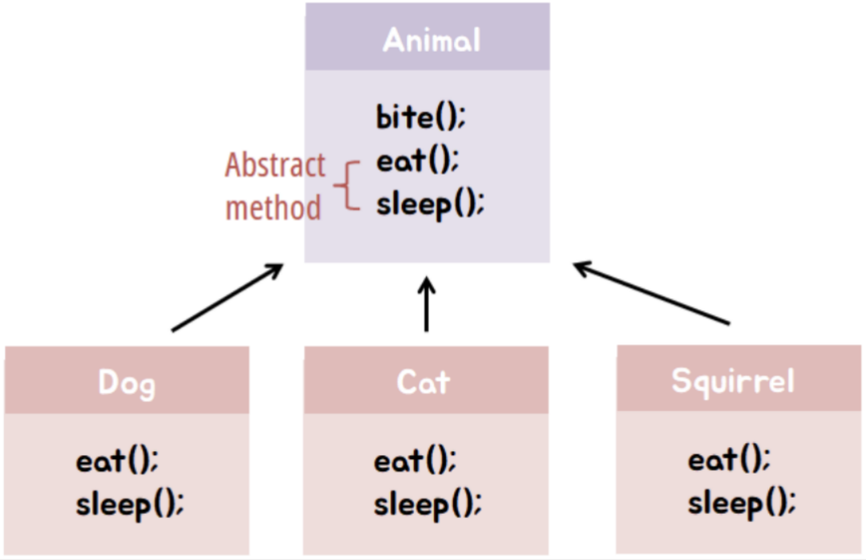
```java
class Animal {

    String type;

    Animal(String type) {
        this.type = type;
    }
}
```

```java
class HelloAnimal {

    void hello(Animal animal) {
        if (animal.type.equals("Cat")) {
            System.out.println("냐옹");
        } else if (animal.type.equals("Dog")) {
            System.out.println("멍멍");
        }
    }
}
```

```java
public class Main {

    public static void main(String[] args) {
        HelloAnimal hello = new HelloAnimal();

        Animal cat = new Animal("Cat");
        Animal dog = new Animal("Dog");

        hello.hello(cat);
        hello.hello(dog);
    }
}
```

OCP (Open-Closed) : 개방-폐쇄 원칙

> **추상화**를 이용한다! → 추상 클래스 or 인터페이스



1. 변경(확장)될 것과 변하지 않을 것을 엄격히 구분한다.
2. 이 두 모듈이 만나는 지점에 추상화(추상 클래스 or 인터페이스)를 정의한다.
3. 구현체에 의존하기보다 정의한 추상화에 의존하도록 작성 한다.

적용 후 코드

```java
abstract class Animal {

    abstract void speak();
}

class Cat extends Animal {

    void speak() {
        System.out.println("냐옹");
    }
}

class Dog extends Animal {

    void speak() {
        System.out.println("멍멍");
    }
}
```

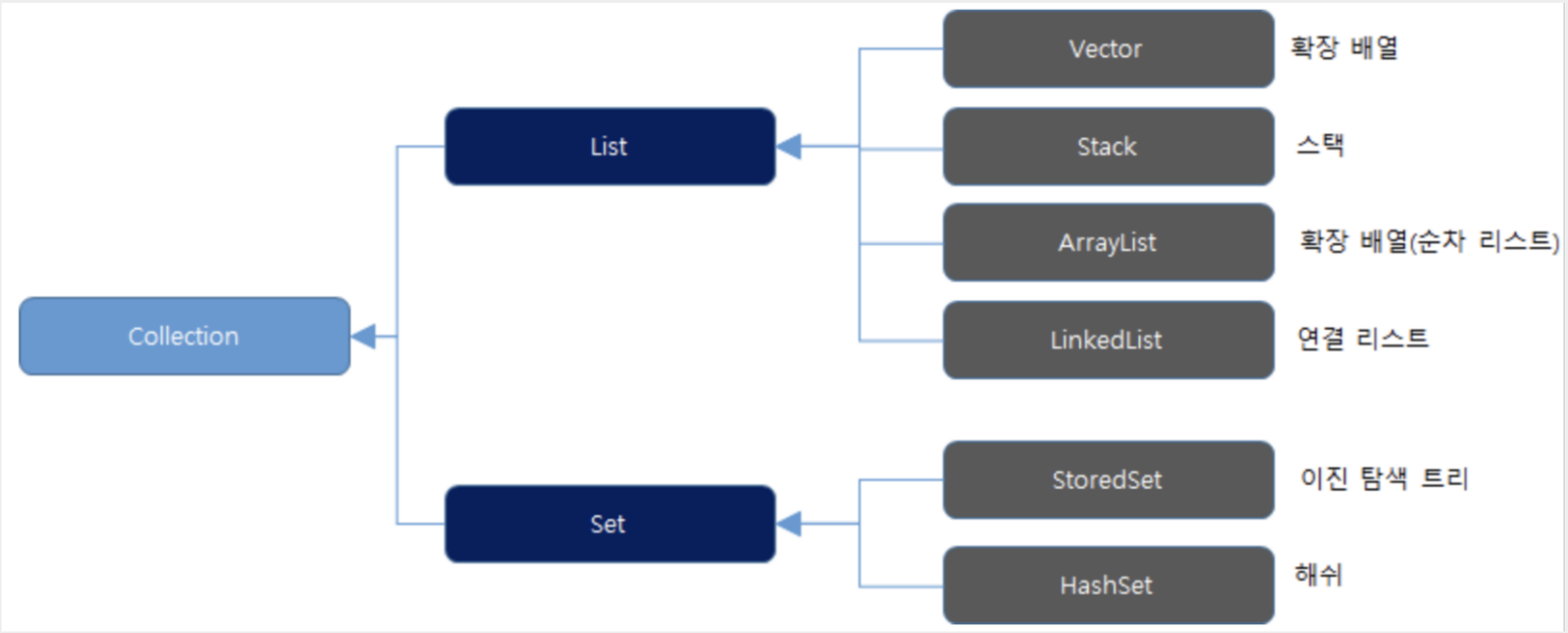
```java
class HelloAnimal {

    void hello(Animal animal) {
        animal.speak();
    }
}

public class Main {

    public static void main(String[] args) {
        HelloAnimal hello = new HelloAnimal();

        Animal cat = new Cat();
        Animal dog = new Dog();

        hello.hello(cat);
        hello.hello(dog);
    }
}
```

LSP (Liskov Substitution) : 리스코프 치환 원칙

> 서브 타입은 언제나 기반 타입으로 **교체**할 수 있어야 한다!

→ **부모 클래스의 인스턴스를 사용하는 위치에 자식 클래스의 인스턴스를 대신 사용했을 때** 코드가 원래 의도대로 작동해야 한다.

다형성을 이용한다!



LSP (Liskov Substitution) : 리스코프 치환 원칙

> 리스코프 치환 원칙이 지켜지지 않으면 개방 폐쇄 원칙을 위반하게 되므로 기능 확장을 위해 더 많은 부분을 수정해야 한다.
> 그럼 확장이 어렵게 되고, 따라서 상속을 잘 정의하여 치환 가능성을 위배되지 않도록 설계해야 한다.

```
void myData() {

        Collection data = new LinkedList();
        data = new HashSet();

        modify(data);
    }

    void modify(Collection data) {

        data.add(1);
        data.add(2);

        ...
    }
```

ISP (Interface Segregation) : 인터페이스 분리 원칙

> 인터페이스를 사용에 맞게끔 각기 **분리**해야 한다!
> 즉, 인터페이스를 잘게 분리함으로써, 클라이언트의 목적과 용도에 적합한 인터페이스 만을 제공한다.

변경 전 코드

```
interface ISmartPhone {

    void call(String number); // 통화 기능

    void message(String number, String text); // 문제 메세지 전송 기능

    void wirelessCharge(); // 무선 충전 기능

    void AR(); // 증강 현실(AR) 기능

    void biometrics(); // 생체 인식 기능
}
```

```
class S20 implements ISmartPhone {

    public void call(String number) { }

    public void message(String number, String text) { }

    public void wirelessCharge() { }

    public void AR() { }

    public void biometrics() { }
}

class S21 implements ISmartPhone {

    public void call(String number) { }

    public void message(String number, String text) { }

    public void wirelessCharge() { }

    public void AR() { }

    public void biometrics() { }
}
```

ISP (Interface Segregation) : 인터페이스 분리 원칙

→ 필요하지 않은 기능들을 구현해야 한다.

```
class S3 implements ISmartPhone {

    public void call(String number) { }

    public void message(String number, String text) { }

    public void wirelessCharge() {
        System.out.println("지원 하지 않는 기능 입니다.");
    }

    public void AR() {
        System.out.println("지원 하지 않는 기능 입니다.");
    }

    public void biometrics() {
        System.out.println("지원 하지 않는 기능 입니다.");
    }
}
```

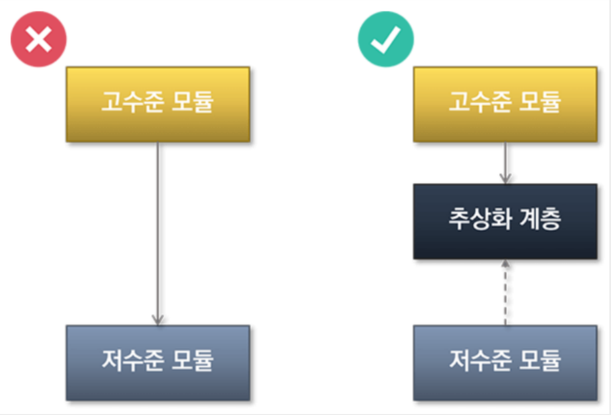
변경 후 코드

```
interface IPhone {

    void call(String number); // 통화 기능
    void message(String number, String text); // 문자 메세지 전송 기능
}

interface WirelessChargable {

    void wirelessCharge(); // 무선 충전 기능
}

interface ARable {

    void AR(); // 증강 현실(AR) 기능
}

interface Biometricsable {

    void biometrics(); // 생체 인식 기능
}
```

DIP (Dependency Inversion) : 의존관계 역전 원칙

> 의존 관계를 형성할 때 구체적인 것 (변하기 쉬운 것)에 의존하기 보단, **추상적인 것 (변하기 어려운 것)**에 의존하자!

→ 저수준 모듈이 변경되어도 고수준 모듈은 타격을 입지 않는 형태

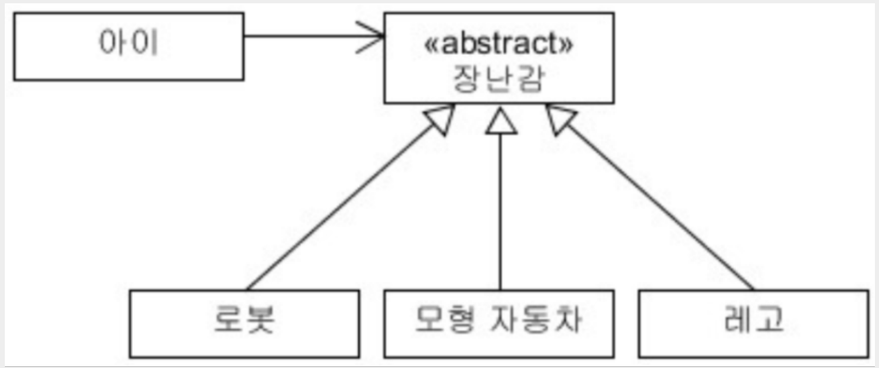



```
public class Kid {

    private Robot toy;

    public void setToy(Robot toy) {
        this.toy = toy;
    }
}
```

변경 전 코드

```
public class Kid {

    private Robot toy;
    private Lego toy; //레고 추가

    // 아이가 가지고 노는 장난감의 종류만큼 Kid 클래스 내에 메서드가 존재해야함.
    public void setToy(Robot toy) {
        this.toy = toy;
    }
    public void setToy(Lego toy) {
        this.toy = toy;
    }
}
```

DIP (Dependency Inversion) : 의존관계 역전 원칙




```java
public class Kid {

    private Toy toy;

    public void setToy(Toy toy) {
        this.toy = toy;
    }

    public void play() {
        System.out.println(toy.toString());
    }
}
```

변경 후 코드

```java
public class Robot extends Toy {

    public String toString() {
        return "Robot";
    }
}

public class Main {

    public static void main(String[] args) {
        Toy robot = new Robot();
        Kid k = new Kid();
        k.setToy(robot);
        k.play();
    }
}
```

```java
public class Lego extends Toy {

    public String toString() {
        return "Lego";
    }
}

public class Main {

    public static void main(String[] args) {
        Toy lego = new Lego();
        Kid k = new Kid();
        k.setToy(lego);
        k.play();
    }
}
```

결론

> **단일 책임 원칙**과 **인터페이스 분리 원칙**은 객체가 커지지 않도록 막아준다.
> 객체가 단일 책임을 갖게 하고 클라이언트마다 다른 인터페이스를 사용하게 함으로써 **한 기능의 변경이 다른 곳에까지 미치는 영향을 최소화할 수 있고, 이는 결국 기능 변경을 보다 쉽게 할 수 있도록** 만들어 준다.

> **리스코프 치환 원칙**과 **의존 역전 원칙**은 **개방 폐쇄 원칙**을 지원한다.
> 개방 폐쇄 원칙은 **변화되는 부분을 추상화하고 다형성을 이용함으로써 기능 확장을 하면서도 기존 코드를 수정하기 않도록** 만들어 준다.
> 여기서, 변화되는 부분을 추상화할 수 있도록 도와주는 원칙이 바로 의존 역전 원칙이고, 다형성을 도와주는 원칙이 리스코프 치환 원칙이다.

개인 일정

	1월	2월	3월	4월	5월	6월	7월~
SQL							
Vue.js							
OS							
Spring							
JPA							
정보처리기사							
알고리즘							
캡스톤							

감사합니다!