객체지향의 5원칙(SOLID)

1. 단일 책임 원칙(Single responsibility principle) : SRP
2. 개방 폐쇄 원칙(Open/closed principle) : OCP
3. 리스코프 치환 원칙(Liskov substitution principle) : LSP
4. 인터페이스 분리 원칙(Interface segregation principle) : ISP
5. 의존관계 역전 원칙(Dependency inversion principle) : DIP

📌단일 책임의 원칙 : SRP (Single Responsibility Principle)

모든 클래스는 각각 하나의 기능만 가진다는 의미입니다.
해당 클래스가 제공하는 **모든 서비스는 단 하나의 책임을 수행하는데 집중 되어야 한다**는 원칙입니다.

SRP 원칙을 적용하면 다른 클래스들이 서로 영향을 미치는 연쇄작용을 줄일 수 있습니다.
응집도(cohesion)는 높이고 결합도(coupling)는 낮출 수 있습니다. 뿐만 아니라 책임을 적절하게 분배함으로써 코드의 가독성 향상, 유지보수 용 이라는 이점까지 누릴 수 있으며 다른 원칙들을 적용하는 기초가 됩니다.
사뭇 단순한 원칙이라고 생각될 수 있으나. 막상 실무에 적용하려면 프로젝트의 복잡하고 자주 변하는 성격 때문에 적용하기가 쉽지 않습니다.

📌개방폐쇄의 원칙 : OCP (Open Close Principle)

소프트웨어의 모든 구성요소(클래스, 모듈, 함수)는 **확장에 열려있고, 변경에는 닫혀있어야 한다는** 원칙입니다.
다시 말하면 요구사항의 변경이나 추가사항이 발생하더라도, 기존 구성요소는 수정이 일어나지 말아야하며 쉽게 확장이 가능하여 재사용할 수 있어야 한다는 뜻입니다.
OCP를 가능케 하는 가장 중요한 메커니즘은 추상화(Abstraction)와 다형성(Polymorphism) 입니다.
OCP는 객체지향의 장점을 극대화 하는 아주 중요한 원리입니다.

- 클래스를 설계할 때 변할 부분과 변하지 않을 부분을 명확히 구분해야 합니다.
- 변할 수 있는 부분은 추상화하여 상속하는 클래스가 의존할 수 있게 코드를 작성 해야합니다.
- 적당한 추상화 레벨을 선택해야합니다. 그래디 부치의 말에 의하면 추상화란 '다른 종류의 객체로부터 식별될 수 있는 객체의 본질적인 특징' 이라고 정의했습니다.
- Interface란 이런 변하지 않을 본질적인 특징에 관한 약속입니다.

ex) 게임 캐릭터들의 스킬이 있습니다. 캐릭터가 스킬을 습득하여 스킬 버튼이 활성화 되었을 경우 스킬이 어떤 내용인지는 모르더라도 버튼을 누를경우 캐릭터가 어떤 행동을 할 것이라는 사실은 자명합니다. 예를들어 앞으로 가거나 뒤로 가거나 점프도 할수 있을 것입니다. 스킬의 자세한 내용은 레벨업을 하거나 전직을 하거나 하면서 달라질 수 있겠지만 버튼을 누르면 스킬이 나간다는 사실은 변하지 않습니다.

코드로 예시를 들어보겠습니다.

```
class 캐릭터{
    public fun 베기(){
        print("🗡🗡🗡");
    }
    public fun 점프(){
        print("🤸");
    }
}

//Game
fun playGame(어떤캐릭터 : 캐릭터){
    어떤캐릭터.점프();
    어떤캐릭터.베기();
}
```

위 코드에서 [베기] 스킬을 없애고 [회전베기]로 변경사항이 생겼다고 가정해 보겠습니다.

```
class 캐릭터{
/*    public fun 베기(){
        print("🗡🗡🗡");
    }
*/
    public fun 회전베기(){
        print("⚔⚔⚔");
    }
    public fun 점프(){
        print("🤸");
    }
}
```

위 코드를 그대로 playGame에 적용하게 되면 캐릭터.베기()에서 컴파일 에러가 날 것입니다.

```
//Game
fun playGame(어떤캐릭터 : 전사){
    어떤캐릭터.점프();
    어떤캐릭터.베기(); //에러!!
}
```

이렇듯 스킬은 변하기 아주 쉬움에도 불구하고 그대로 상속해버리면 문제가 발생하기 쉽습니다.

스킬 두개가 각각 Q와 W버튼을 눌렀을 때 반응한다고 가정하면 Q버튼과 W버튼을 누르는 동작은 절대 변하지 않습니다. 이런 부분을 Interface로 만들어 줍니다.

```
interface 캐릭터 {
    fun QPressed()
    fun WPressed()
}
```

```
class 어떤캐릭터 : 캐릭터 {
    override fun QPressed(){
        회전베기();
    }

    override fun WPressed(){
        점프();
    }

    private fun 회전베기(){
        print("xxx");
    }
    private fun 점프(){
        print("🤸");
    }
}
```

위와 같이 Interface를 상속하여 변하는 부분과 변하지 않을 부분을 정확히 나누어 변할 수 있는 부분은 private로 선언하여 접근할 수 없게 만들었습니다.
또한 스킬에 변동 사항이 생기더라도 게임 진행에 전혀 문제가 없을 것입니다.

📌리스코브 치환의 원칙 : LSP (the Liskov Substitution Principle)

부모 클래스를 가리키는 포인터에 해당 클래스를 상속하는 자식 클래스를 할당하더라도 모든 기능이 정상 작동해야 하며 자식 클래스의 상세 내부를 부모 클래스는 알 필요가 없다는 뜻입니다.
한마디로, 부모 클래스를 상속한 자식 클래스는 부모 클래스의 역할을 정확히 해내야 한다는 뜻입니다.(정말 당연한 말이지만 잘 지켜지지 않는 원칙입니다.)

보통 부모 클래스의 메소드를 override하면서 문제가 발생합니다. 부모 클래스의 기존 메소드를 자식 클래스가 수정하면서 문제가 생깁니다.
LSP를 지키는 가장 간단한 방법은 상속은 하되 override를 안하는 것입니다.
하지만 이게 무조건 좋은 방법은 아닙니다.
상속을 할때 override가 필요하다면 기존 부모 클래스의 메소드가 하던 역할을 충실히 수행하고 기능의 추가만 신중하게 수행하면 됩니다.

LSP는 결국 상속의 과정 중 메소드의 재정의가 필요하다면 현재 자식 클래스가 부모 클래스의 기존 메소드의 의미를 해치지는 않는지 신중히 고민하고 올바르게 상속하라는 의미라고 생각합니다.

📌인터페이스 분리의 원칙 : ISP (Interface Segregation Principle)

자신이 사용하지 않는 인터페이스는 구현하지 말아야 한다는 원칙입니다. 다시말해, 하나의 큰 인터페이스를 상속 받기 보다는 인터페이스를 구체적이고 작은 단위들로 분리시켜 꼭 필요한 인터페이스만 상속하자 라는 의미입니다.
SRP가 단일 책임을 강조했다면 ISP는 인터페이스의 단일책임을 강조합니다.

인터페이스 하나의 크기가 크다는 것은 한번에 지켜야할 약속이 많아진다는 것을 의미합니다.

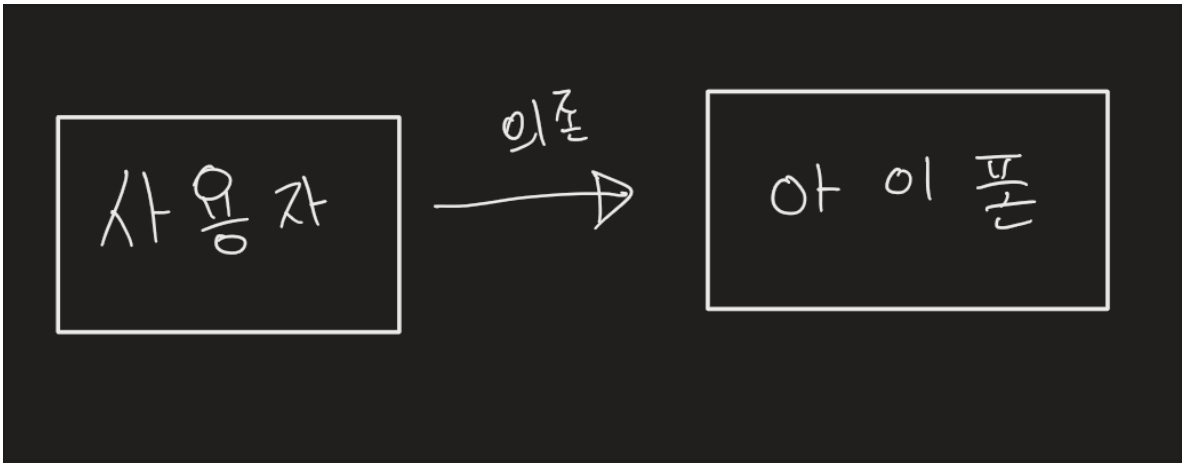
📌의존성 역전의 원칙 : DIP (Dependency Inversion Principle)

원문을 그대로 번역한다면 "상위 모듈은 하위 모듈에 의존해서는 안된다. 둘 다 추상화에 의존해야한다." , "추상화는 구체적인 것에 의존해서는 안된다. 구체적인 것은 추상화에 의존해야한다." 입니다.
글만 읽어서는 쉽게 이해되는 개념이 아닙니다.

클래스 사이에는 의존관계가 존재하기 마련입니다. 다만, 의존관계가 존재하되 구체적인 클래스에 의존하지 말고 최대한 추상화한 클래스에 의존하라는 뜻입니다. 다시말하면 interface를 적극적으로 활용하라는 의미이기도 합니다.

간단한 예시를 들어보겠습니다.

사용자가 존재하고 사용자는 아이폰을 사용합니다.

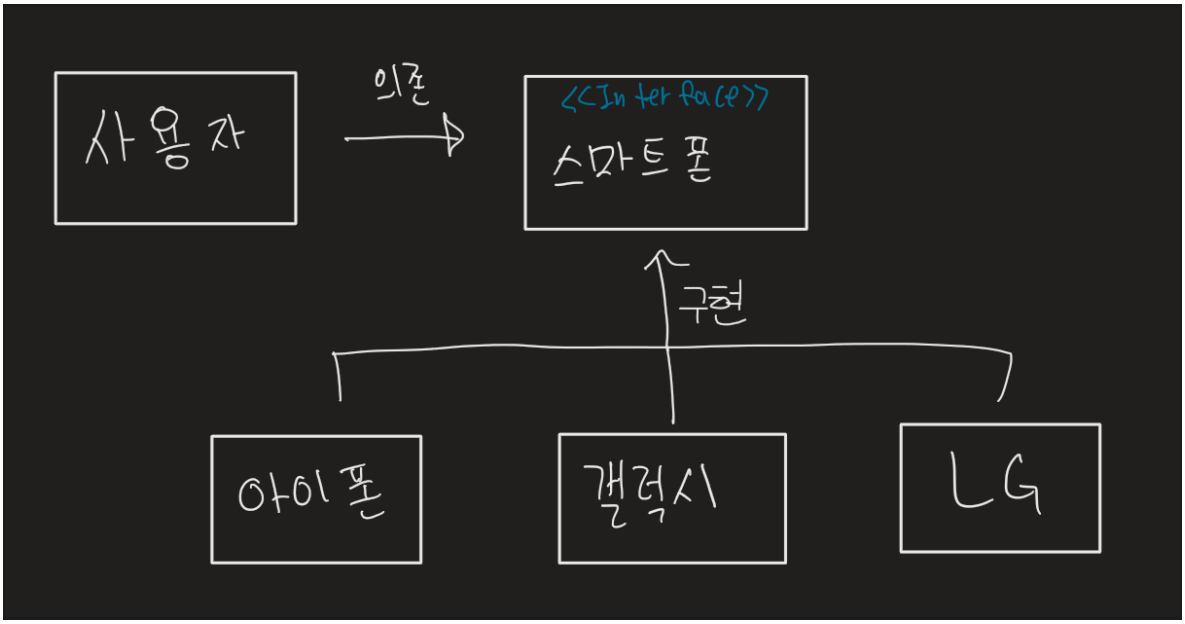


```
class 아이폰 {

fun 전화(){
     print("📞📞📞")
   }

fun 검색(){
     print("🔎🔎🔎")
   }
}

class 사용자 {

val 내폰 = 아이폰()

fun 전화(){
    내폰.전화()
   }

fun 검색(){
    내폰.검색()
   }
}
```

위 예시 코드에서 사용자는 아이폰 클래스에 의존하고 있습니다. 그리고 아이폰 클래스는 구체적인 클래스이기 때문에 변화에 취약합니다.

만약 사용자가 아이폰을 다른 스마트폰으로 바꾸고 싶어한다면 코드에 상당한 변화가 필요할 것입니다.

이 취약한 구조를 개선하기 위해 의존성을 역전 시킬 필요가 있습니다. 현재 사용자가 의존하고 있는 아이폰 클래스를 덜 구체적인 추상화된 클래스로 만드는 것입니다.



스마트폰이라는 추상화된 interface를 각종 구체화된 클래스들이 상속하고
사용자는 스마트폰 Interface에 의존합니다.
"구체적인 것은 추상화에 의존해야한다" 라는 말이 이해가 되실 것입니다.

기존에 사용자 클래스(상위 계층 = 정책 결정)가 아이폰 클래스(하위 계층 = 세부 사항)에 의존하던 상황을 반전시켜서 구현으로부터 독립 되었습니다. 이제 사용자와 아이폰 모두 추상화에 의존하는 상황으로 변경된 것입니다. 마침내 "상위 모듈은 하위 모듈에 의존해서는 안된다. 둘 다 추상화에 의존해야한다." 라는 말의 의미도 이해가 되실 것입니다.

코드는 다음과 같이 변경 가능할 것입니다.

```
interface 스마트폰 {
    fun 전화()
    fun 검색()
}

class 아이폰 : 스마트폰 {

override fun 전화(){
    print("📞📞📞")
    }

override fun 검색(){
    print("🔎🔎🔎")
    }
}

class 사용자(내폰 : 스마트폰){

fun 전화(){
    내폰.전화()
    }

fun 검색(){
    내폰.검색()
    }
}

//실제 사용
val 나 = 사용자(object : 아이폰())
나.전화()
나.검색()
```

위의 코드에서는 의존성 주입(DI : Dependency Injection) 이라는 개념도 사용되었는데 다른 포스팅에 설명 되어 있습니다. 간단히 말하면 의존관계가 있는 클래스를 외부에서 주입한다는 개념입니다.

이렇게 의존하는 클래스를 추상화하고 외부에서 주입함으로써 외부 변동에 유연하게 대처할 수 있는 코드로 개선되었습니다.

의존성 역전의 원칙을 적용하여 의존성 주입이라는 이점까지 취할 수 있게 되었습니다.