SOLID 원칙이란?

객체 지향 5대 원칙으로 불리는 SOLID 원칙에는 SRP(단일 책임 원칙),OCP(개방 폐쇄 원칙)

LSP(리스코프 치환 원칙), DIP(의존 역전 원칙), ISP(인터페이스 분리 원칙을 말한다). 이들의 앞자를 따서 만든 것으로, 프로그래머가 시간이 지나도 유지 보수와 확장이 쉬운 소프트웨어를 만드는데 도움을 주기 위해 고안 된 것.

1. 단일 책임원칙(Single Responsibility Principle): 단일 책임 원칙은 “클래스는 단 한 개의 책임을 가져야 한다.”를 의미한다. 클래스가 여러 책임을 갖게 되면 그 클래스는 각 책임마다 변경되는 이유가 발생하기 때문에 클래스가 한 개의 이유로만 변경되려면 클래스는 한 개의 책임만을 가져야한다고 한다. 이러한 이유로 이 원칙은 다른 말로 “클래스를 변경하는 이유는 단 한 개여야 한다.”

이 “책임”이라는 것은 무엇일까. 책임에 대해서 살펴보기 전에 단일 책임 원칙을 지키지 않았을 때의 문제점에 대해 살펴보자.

HTTP 프로토콜을 이용하여 데이터를 읽어와 화면에 보여주는 기능을 하는 코드가 있다고하자

Display() 메소드는 loadHtml()에서 읽어 온 HTML 응답 문자열을 updateGui()메소드 에 보낸다고 하자. UpdateGui()메소드는 parseDataToGuiData() 메소드를 이용해서 HTML 응답 메시지를 GUI에 보여주기 위한 GuiData 객체로 변한 뒤에 실제 tableUI를 이용하여 데이터를 보여준다 하자.

단일 책임 원칙을 지키지 않았을 때의 문제점. 데이터를 제공하는 서버가 HTTP로 유지된다면 여러 책임을 가져도 문제가 되지 않지만. 이 서버가 소켓 기반의 프로토콜로 변경되었다면 어떨까?

이 프로토콜은 응답 데이터로 byte 배열로 제공한다면 대대적으로 코드의 변화가 발생한다.

여러 데이터의 구조가 String에서 byte[] 로 변경되고, updateGui의 파라미터 타입이 변경되며

GuiData를 생성하는 parseDataToGuiData() 메소드의 코드가 변하게되었다.

단지 데이터를 제공하는 서버만 달라졌을 뿐인데, 연쇄적으로 코드가 수정된다. 책임의 개수가 많아질수록 한 책임의 기능 변화가 다른 책임에 주는 영향이 비례해서 증가하게되고, 결국 코드를 절차지향적으로 변하게하여 유지보수를 엉망으로 만들게 된다.

따라서 데이터 GUI를 보여주는 책임을 담당하는 객체와 데이터를 읽는 책임을 담당하는 객체,

그리고 데이터 자체를 추상화한 객체 3가지를 이용하여 책임을 분리하여야 한다.

단일 책임 원칙을 지키지 않았을 때 재사용이 어려워진다. DataViewer는 데이터를 읽기위한 클래스이나 그 안에서 HTTP 연동을 위해서 HttpClient 패키지를 사용하고, 화면에 데이터를 보여주기 위해 GuiComp라는 패키지를 사용한다고하자. 이때, 단순히 데이터만 읽어오려는 객체가 있다면,필요없는 GuiComp 패키지 까지 필요하게 되는 문제가 발생한다. 단일 책임 원칙에 따라 책임이 분리되었다면, HttpClient 패키지만 필요하게된다.

그렇다면 책임이란 대체 무엇일까. 한 책임의 구현 변경에 의해 다른 책임과 관련된 코드가 변경될 가능성이 높아진다는 것을 알게되었다. 하지만, 위코드에서 데이터의 읽어 오는 방식이 유지된다면 DataViewer 클래스는 수정할 필요가 없게된다. 이처럼 기능 변경 요구가 없을 때 수정에 대한 문제가 없다는 것은, 반대로 생각하면 책임의 단위는 변화되는 부분과 관련이 있다는 의미가 된다.

코드에서 DataViewr 클래스에서 데이터를 읽어오는 기능에 변화가 발생하였는데, 이런 변화를 통해 데이터를 읽어오는 기능이 별도로 분리되어야 할 책임이라고 이해할 수 있다.

또한 각각의 책임은 서로 다른 이유로 변경되어야 하므로 데이터를 읽어 오는 책임의 기능이 변경될 때 데이터를 보여주는 책임은 변하면 안된다.

그렇다면 어떻게 서로 다른 이유로 변경되는 것을 알 수 있을까? 그것은 바로 메소드를 실행하는 것이 누구인지를 살펴보는 것이다. 그리고 그 사용자들이 해당 클래스의 서로 다른 메소드를 사용한다면 그들 메소드는 각각 다른 책임에 속할 가능성이 높고, 책임 분리 후보로 판단할 수 있게 된다.

책임은 변화에 대한 것임을 알게되었다. 하지만 코드를 보면 단일 책임은 단순히 하나의 메소드가 동작하는 기능 같기도하다. 따라서 우리는 그 외에도 아래와 같은 의문을 가질 수 있다.

1. 클래스가 여러 가지의 public 메소드를 가진다면, 복수의 책임을 갖는가?
2. 클래스가 다중 상속(혹은 다중 구현)을 한다면, 복수의 책임을 갖는가?
3. 해당 클래스를 의존하는 사용자가 여럿이라면 변경되는 이유는 여러가지인가?

단일 책임은 하나의 메소드가 하는 일이라고 생각하면, 첫 번째 생각을 할 수 있을 것이고,

특정 클래스가 여러 클래스에 상속을 받는다면 단일 책임을 갖지 않는다고 생각할 수 있다.

마지막으로, 해당 클래스의 사용자가 여러명이면 변경되는 이유가 여러가지라고 판단하여

단일 책임을 갖지 않는다고 생각할 수 있다. 이 의문을 해결하는 훌륭한 정의는

“하나의 모듈은 하나의, 오직 하나의 액터에 대해서만 책임져야한다.”

SOLID를 창시한 로버트 C 마틴이 한말로 SRP를 새롭게 정의하였다. 여기서 ‘액터’는

시스템이 동일한 방식으로 변경되기를 원하는 사용자 집단을 의미한다.

즉 액터는 한 명일수도 있고, 여러 명이 될 수도 있는 것입니다. 이렇듯, SRP를 설계할 때는

거시적인 관점에서 해당 클래스에 어떤 액터가 의존하는지 생각하는 것이 바람직하다.

액터라는 것을 예시로 들어보면, ‘스마트폰’이라는 객체를 철수와 영희가 사용하고 있다고 가정해보자. 이때 철수는 스마트폰을 영상 시청을 위해서 사용하고 영희와 전화통화를 위해서 사용합니다.

위의 스마트폰 객체는 철수와 영희가 다른 방식으로 변경되기를 원할 수 있기 때문에 철수와 영희는 별 개의 액터입니다. 철수가 만약 영상 시청을 위해서 스마트폰의 액정 크기를 변경한다면 전화 통화 요구사항에는 맞지 않는 변경사항이 된다. 그러므로 해당 스마트폰이 단일 책임 원칙을 지키기 위해서는 다른 액터로 분리되어야한다.

하지만, 철수와 영희가 모두 같은 요구사항으로 스마트폰을 사용한다면 철수와 영희를 하나의 액터로 볼 수 있으므로 단일 책임 원칙을 준수한다고 할 수 있다.

이제 위의 의문점을 해결해보자, 먼저 두번째와 세번째 액터라는 개념을 통해서 쉽게 해결이 가능하다. 다중 상속을 받더라도 액터가 그 다중 상속한 것을 모두 사용한다면 단일 책임 원칙을 만족하는 것이고, 해당 클래스의 사용자가 여러명이어도 모두 동일한 요구 사항으로 해당 클래스를 사용한다면 단일 책임 원칙을 준수하는 것이다. 첫 번째 의문점도 마찬가지로 서로 다른 액터가 해당클래스의 여러 가지 메소드를 사용하는 것이 아니라면, 복수의 메소드여도 단일 책임 원칙을 지키고 있는 것입니다.

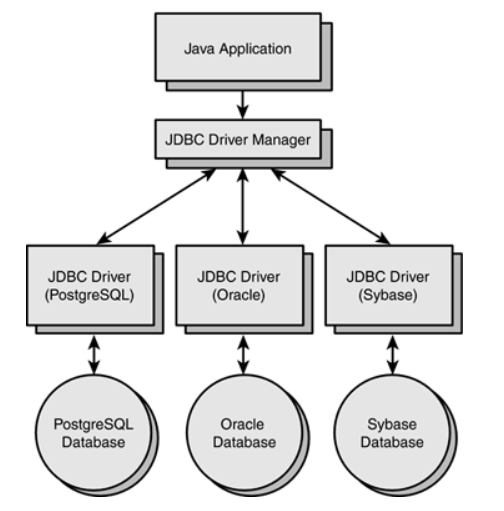
지금까지 단일 책임 원칙에 대해서 알아 보았습니다. 정의 자체는 간단하지만, 그 안에 ‘책임’이라는 기준을 세운다는 것이 굉장히 어렵다고 생각합니다. 이를 위해서는 단순히 해당 클래스만을 바라보기 보다면, 시야를 넓혀서 액터를 정의하는 것이 중요하겠습니다.

1. 개방 폐쇄의 원칙 정의

개방 폐쇄 원칙은 “확장에는 열려있어야 하고, 변경에는 닫혀 있어야한다.”를 의미한다.

조금 더 쉽게 풀어쓰자면, “ 기능을 변경하거나 확장할 수 있으면서 그 기능을 사용하는 코드는

수정하지 않는다.”를 뜻한다.



자바 어플리케이션에서 JDBC 매니저를 이용하기 위해서 설계된 구조이다. JDBC 매니저를 상속받는 PostgreSQL, Oracle, Sybase는 모두 변경에 확장적 이지만, 자바 어플리케이션은 수정에 폐쇄적이다. 더 쉽게 이야기하면, Oracle DB의 변화가 발생하여도 자바 어플리케이션에서 수정할 코드는 없다. 즉 개방 폐쇄 원칙은 하나의 변화가 다른 곳에도 연쇄적으로 변화를 일으키는 것을 방지하기 위해 만들어 졌다.

그렇다면, 개방 폐쇄 원칙을 어떻게 지킬 수 있을까?

2-2 개방 폐쇄 원칙을 지키는 방법

개방 폐쇄 원칙의 핵심은 변화하는 부분을 추상화 하는 것이다. 위의 3가지 DB는 기능의 이름은 같더라도 구체적으로 어떻게 동작하는지 다를 수 있다. 이 부분에 대해서 추상화 함으로써 기능을 고정시킬 수 있다. 주로, 인터페이스를 통해서 구현한다.

두번째는 상속을 이용한다. 클라이언트의 요청이 왔을 때 데이터를 HTTP 응답 프로토콜에 맞춰

데이터를 전송해 주는 ResponseSender가 있다고 하자.

ResponseSender 클래스의 send()메소드는 헤더와 몸체 내용을 전송하기 위해 sendHeader() 메소드와 sendBody() 메소드를 차례대로 호출하며, 이 두 메소드는 알맞게 HTTP 응답 데이터를 생성한다. 이때, 이 두 메소드는 protected 공개 범위를 갖고 있기 때문에 하위 클래스에서 오버라이딩이 가능하다.

ZippedResponseSender 클래스는 기존 기능에 압축 기능을 추가해 주는데, 이 기능을 추가하기 위해 ResponseSender 클래스의 코드는 바뀌지 않았다. 즉,ResponseSender 클래스는 확장에는 열려있으면서 변경에는 닫혀있는 것이다.

개방 폐쇄 원칙을 지키지 않았을 때의 문제점.

추상화와 다형성을 이용해서 개방 페쇄 원칙을 구현하기 때문에, 추상화와 다형성이 제대로 지켜지지 않은 코드는 개방 폐쇄 원칙을 어기게 된다. 이제 OCP 원칙을 어기는 특징을 살펴보자

1. 다운 캐스팅 한다.

예를들어, 슈팅 게임을 개발하는 경우 플레이어, 적, 미사일 등을 그리기 위해 Character 클래스의 draw()메소드를 정의한 후, Player, Enemy, Missile 클래스에 상속을 할 수 있다. 그런데 화면에 이들 캐릭터를 표시해 주는 코드가 아래와 같다면 어떨까요?

**public** **void** **drawCharacter**(Character character) {

**if**(character **instanceof** Missile) { *// 타입 확인*

Missile missile = (Missile) character; *// 타입 다운 캐스팅*

missile.drawSpecific();

} **else** {

character.draw();

}

}

다른 캐릭터는 그리는 방식이 동일하지만, 미사일만 다른 방식으로 표현해 주고 싶을 수도 있다.

가장 쉽게 할 수 있는 방법은 다운 캐스팅을 취하는 것이지만, Character 클래스에서 변경이 일어날 때마다 해당 메소드를 수정해 주어야 할 수도 있다. 즉 변경에 닫혀있지 않다.

따라서, instanceof와 같은 타입 확인 연산자가 사용된다면 해당 코드는 개방 폐쇄 원칙을 위반할 가능성이 높으므로, 타입 캐스팅 후 실행되는 메소드가 변화 대상인지 확인해야 한다.

가령, drawSpecific()메소드가 다른 객체들에서도 적용할만한 메소드라면, Character 타입에 추가하라는 것이다.

(2) 비슷한 if else 블록이 존재한다.

이번에는 character 클래스를 상속 받은 Enemy 클래스가 있다고 가정해보자. 정해진 패턴에 따라

경로를 이동하는 코드가 필요하다면, 아래와같이 작성된다.

Public class Enemy extends Character {

Private int pathPattern;

Public Enemy(int pathPattern) {

This.pathPattern = pathPattern;

}

Public void draw() {

If (pathPattern == 1) {

x += 4;

} else if (pathPattern == 2) {

y += 10;

} else if (pathPattern == 4) {

x+=4;

y+=10;

}

Enemy 클래스에 새로운 경로 패턴을 추가할 때마다 draw()메소드에 새로운 if 문이 생깁니다.

즉, 경로를 추가하는데 Enemy 클래스는 변경에 닫혀있지 않은 것이다. 따라서, (x,y)와 같은

경로 패턴을 추상화 해야한다.

**public** **class** **Enemy** **extends** **Character** {

**private** PathPattern pathPattern;

**public** **Enemy**(PathPattern pathPattern) {

**this**.pathPattern = pathPattern;

}

**public** **void** **draw**() {

**int** x = pathPattern.nextX();

**int** y = pathPattern.nextY();

...; *// 그려 주는 코드*

}

}

이렇게 추상화 하고나면, 새로운 이동 패턴이 생기더라도 draw() 메소드는 변경되지 않는다.

개방 폐쇄 원칙은 유연함에 관련된 원칙이다. 변화하는 부분을 추상화 함으로써 기존 코드를 수정하지 않고도, 확장을 할 수 있게 만들어 준다.

따라서, 우리는 개발을 하다가 코드에 대한 변화 요구가 발생하면, 변화와 관련된 구현을 추상화 해서 개방 폐쇄 원칙에 맞게 수정 할 수있는지 확인해 보는 습관을 기른다.

1. 리스코프 치환 원칙이란? LisKov substitution Principle의 정의

리스코프 치환 원칙은 “상위 타입의 객체를 하위 타입의 객체로 치환해도 상위 타입을 사용하는 프로그램은 정상적으로 동작해야한다”를 의미한다.

다시 말하면, 특정 메소드가 상위 타입을 인자로 사용한다고 할 때, 그 타입의 하위 타입도 문제 없이 정상적으로 작동해야 한다는 것을 말한다.

Public int calculate(final Item item) {

Return item.calculate();

}

상위 타입인 Item이 있고 그 하위타입이 Apple 이라면, 위 인자로 item이 아닌 apple을 넘겨도 코드가 동작하도록 한다.

여기서, 리스코프 치환 원칙이 제대로 지켜지지 않으면 다형성에 기반한 개방폐쇄 원칙 역시 위반하는 것이기 때문에, 리스코프 치환 원칙을 지키는 것이 중요하다.

리스코프 원칙을 지키지 않는 가장 대표적인 예가 ‘직사각형-정사각형 문제’이다. 우리는 직사각형은 정사각형이 아니지만, 정사각형은 직사각형이라는 사실을 알고 있다.

직사각형의 가로와 세로를 비교한 다음에, 세로가 가로보다 짧거나 같다면

가로의 길이에 1을 더한 만큼의 길이를 갖게 만드는 역할을 한다. 정사각형이 아닌 직사각형에 대해서는 위 메소드가 올바르게 작동한다. 예를 들어, 가로가 3, 세로가 2라고 한다면, 세로의 길이는 4가 되는 것이다.

하지만, 정사각형의 경우는 다르다. 정사각형은 항상 가로와 세로의 길이가 같으므로 위 메소드를 실행하게 되면 가로와 세로의 길이가 모두 1씩 증가하게 된다.

즉, 우리가 원하는 “메소드 실행 후, 직사각형의 길이는 가로보다 세로가 길어야 한다.”는 가정이 꺠지게 되는 것이다. 따라서 instanceof를 통해 타입 비교를 해야한다.

해당 도형이 정사각형일 경우 익셉션을 발생시키는 식으로 코드를 수정할 수 있지만 이것은 개방 폐쇄 원칙에 어긋나는 코드이다. 왜냐하면, increaseHeight()가 확장에는 열려 있지 않기 때문이다.

따라서, Square 클래스는 Rectangle 클래스를 상속받으면 안된다. 아무리 우리가 ‘직사각형은 정사각형이다.’라고 이야기를 해도 위처럼 문제가 발생했기 때문이다.

리스코프 치환 원칙은 기능의 명세와 확장에 대한 것.

-높이 값을 파라미터로 전달 받은 값으로 변경한다.

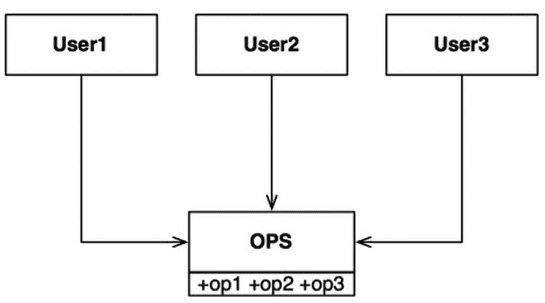
-폭 값은 변경되지 않는다.

즉, Rectangle 클래스의 setHeight()는 높이만 변경되고 폭은 그대로 유지될 것으로 가정하는데, Square 클래스의 setHeight()는 높이와 폭이 모두 바뀌게 된다. 따라서 상위 타입에서 정한 명세를 하위 타입에서도 그대로 지킬 수 있을 때 상속 해야한다.

이러한 개념은 우리가 확장을 하는데 어렵게 만든다. 그렇기에 우리는 상속을 잘 정의하여 치환 가능성을 위배되지 않도록 설계하여야한다.

인터페이스 분리 원칙 (Interface Segregation Principle) 정의

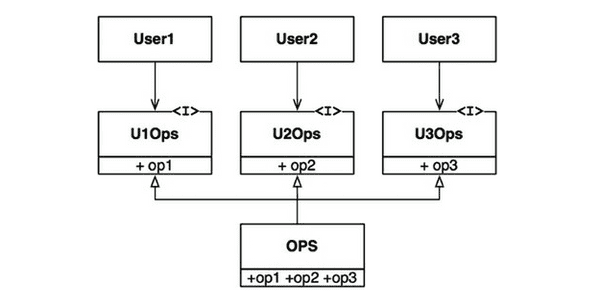
인터페이스 분리원칙은 “클라이언트 자신이 사용하는 메소드에만 의존해야한다”를 의미한다. 이를 쉽게 표현하면, “인터페이스는 그 인터페이스 를 사용하는 클라이언트를 기준으로 분리하여야 한다” 라고 바꿔 말할 수 있다.



위의 구조는 인터페이스 분리원칙을 위반한 예시이다. User1은 op1,User2는 op2, User3는 op3를 사용한다고 가정해보자.

이때, User1의 소스 코드는 op2와 op3를 사용하지 않음에도 이것들에 의존하며, op2의 소스 코드가 변경된다면 User1을 재 컴파일하여 배포해야 한다는 문제가 있다.

따라서, ISP를 지키기 위해서는 구조를 아래처럼 변경해야 한다.



OPS를 상속 받는 U1Ops, U2Ops, U3Ops로 분리한 다음, 적합한 클라이언트가 각각의 메소드를 사용할 수 있도록 바꾸면 된다.

정리.

인터페이스 분리 원칙은 클라이언트를 기준으로 인터페이스를 분리함으로써, 클라이언트로부터 발생하는 인터페이스의 여파가 다른 클라이언트에 미치는 영향을 최소화 하는 것을 목표로 한다고 할 수 있다.

1. 의존 역전 원칙 (Dependency Inversion Principle)의 정의

의존 역전 원칙은 “고수준 모듈은 저수준 모듈의 구현에 의존해서는 안된다. 저수준 모듈이 고수준 모듈에서 정의한 추상 타입에 의존해야한다”를 의미한다. 이것을 아주 쉽게 말하면 “자신보다 변하기 쉬운 것에 의존하지마라”라고 이해하면 된다.

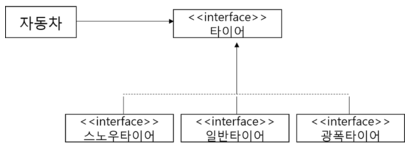
의존 역전 원칙의 예제 DIP하면 대표적인 예제라 할 수 있는 자동차와 스노우 타이어 이야기이다. 현재는 겨울이기 때문에 스노우 타이어를 구매하여 자동차에 끼도록 설계하엿다. 즉 고수준 모듈인 자동차가 저수준 모듈인 스노우 타이어에 의존하는 상태이다.

하지만 날씨가 따뜻해지면서 더 이상 스노우 타이어를 사용할 필요가 없어지게된다. 그래서 일반타이어로 교체하기로 결정한다.

그런데 단순히 스노우 타이어를 일반 타이어로 바꾼다고 코드가 끝나는것이 아니다. 이것에 의존하고 있던 자동차의 코드도 연쇄 적으로 영향을 끼치게 된다.

이것을 개방-폐쇄 원칙을 위반하는 것이므로 추상화나 다형성을 통해서 문제를 고쳐야한다. 의존 역전 원칙은 그 중에서도 추상화를 이용한다.

바로, 스노우 타이어나 일반 타이어를 ‘타이어’자체로 추상화 하는 것이다.



여기서 타이어는 저수준 모듈보다는 고수준 모듈인 자동차 입장에서 만들어지는데 이것은 고수준 모듈이 저수준 모듈에 의존했던 상황이 역전되어

저수준 모듈이 고수준 모듈에 의존하게 된다는 것을 의미한다. 이런 맥락에서 이 원칙의 이름이 의존 역전 원칙이라는 것이다.

다만, 소스 코드의 의존은 자동차가 ‘타이어’를 의존하지만, 런타임에서 객체 의존은 타이어가 아니라 하위 타이어 중 하나를 의존한다.

따라서 의존 역전 원칙은 런타임에서의 의존을 역전시키는 것이 아니라 소스코드 단계에서의 의존을 역전 시킨다는 것을 유의해야 한다.

## 정리

지금까지 SOLID 5가지 원칙을 모두 알아 보았습니다. 간단히 5가지 특징을 요약해 보겠습니다.

단일 책임 원칙과 인터페이스 분리 원칙은 객체가 커지지 않도록 막아줍니다. 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 줍니다.

리스코프 치환 원칙과 의존 역전 원칙은 개방 폐쇄 원칙을 지원합니다. 개방 폐쇄 원칙은 변화되는 부분을 추상화하고 다형성을 이용함으로써 기능 확장을 하면서도 기존 코드를 수정하기 않도록 만들어 줍니다. 여기서, 변화되는 부분을 추상화할 수 있도록 도와주는 원칙이 바로 의존 역전 원칙이고, 다형성을 도와주는 원칙이 리스코프 치환 원칙인 것입니다.