Introduction

앞에서 추상화 및 다형성을 이용하여 OCP를 구현했습니다. 그런데 추상클래스에 의존하였으나 상속을 구현한 객체들을 사용할 수 있었습니다.

그렇다면 어떤 설계 원리가 이렇게 사용할 수 있도록하는 걸까요?

가장 좋은 상속 계층(inheritance hierarchies)의 특징은 무엇일까?

어떤 요소가 OCP에 부합하지 않는 계층 구조를 유발할까요?

로버트 마틴이 말하는 LSP

FUNCTIONS THAT USE POINTERS OR REFERENCES TO BASE CLASSES MUST BE ABLE TO USE OBJECTS OF DERIVED CLASSES WITHOUT KNOWING IT.

=> 베이스 클래스를 참조하거나, 베이스 클래스의 포인터를 사용하는 함수는 반드시 베이스 클래스로부터 파생된 클래스에 대해 모르고도 파생 클래스를 사용할 수 있어야 한다.

[참고]

Liskov가 실제 했던 말

Subtype Requirement: Let $\phi(x)$ be a property provable about objects x of type T. Then $\phi(y)$ should be true for objects y of type S where S is a subtype of T.

[Barbara Liskov and Jeannette Wing described the principle succinctly in a 1994 paper as follows[1]:]

object x가 type T, object y는 type S라 하자.

그리고 파이(x)를 x에 대해 증명할 수 있는 속성이라 정의하자.

만약에 S가 T의 서브타입이라면 파이(y)도 반드시 참이여야 한다.

이 원칙이 중요한 이유

이 원칙이 위배되었을 때 상황을 떠올리면 쉽습니다. 만약에 LSP를 따르지 않는다고 봅시다. 베이스 클래스를 참조하거나 포인터를 사용하는 함수는 반드시 베이스 클래스로부터 파생되는 함수를 모두 다 알아야 합니다. 이렇게 된다면 OCP를 위반하게 되는것이죠. 파생된 클래스가 새로 생성될 때마다 해당 함수를 수정하게 되니까요.

다음으로 볼 내용은 LSP를 위반한 예제입니다. 어떻게 문제를 해결해가는지, 그 과정에서 결국엔 어떤 결론에 도달하는지를 살펴보겠습니다.

A Simple Example of a Violation of LSP

```
void DrawShape(const Shape& s) {
  if (typeid(s) == typeid(Square))
  DrawSquare(static_cast<Square&>(s)); else if (typeid(s) == typeid(Circle))
  DrawCircle(static_cast<Circle&>(s));
}
```

이 예제에서 보시면 DrawShape함수는 Shape라는 기반 클래스를 참조합니다.하지만 LSP를 위배하였기에, s로부터 파생된 클래스인 정사각형과 원을 일일이 식별하며 도형을 그리는 함수를 실행하고 있습니다.

[참고] 이러한 함수의 구조를 OOD의 저주라고 부름(anathema to OOD)

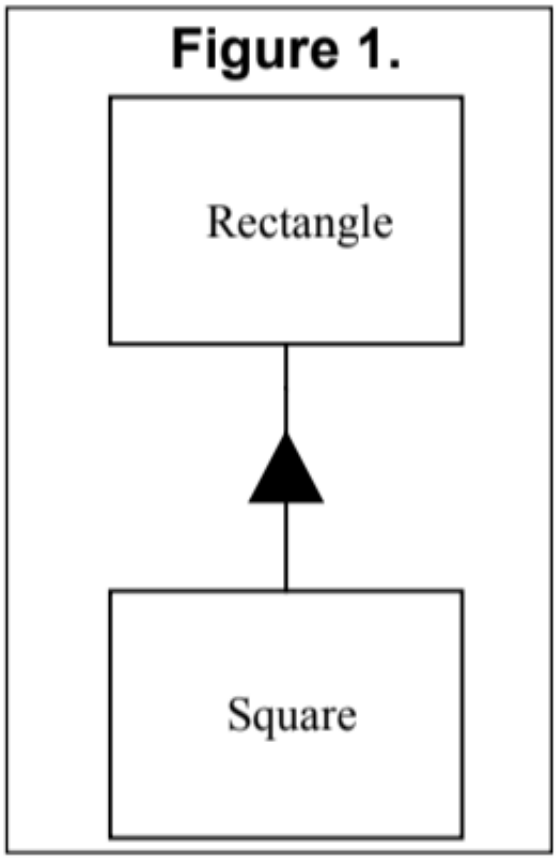
Square and Rectacgle, a More Subtle Violation

직사각형을 그리는 클래스가 있는데, 정사각형 클래스가 이를 상속받아 정사각형을 그리는 예제를 살펴보겠습니다. 그리고 발생하는 문제에 대해선, 트릭으로 해결해보겠습니다.

비껴가기 예제

방법1 관념에 맞게 적용

```
class Rectangle {
  public:
  void SetWidth(double w) {itsWidth=w;}
  void SetHeight(double h) {itsHeight=w;}
  double GetHeight() const {return itsHeight;}
  double GetWidth() const {return itsWidth;}
  private:
  double itsWidth;
  double itsHeight;
};
```



상속은 ISA관계입니다. [Figure 1]을 보시면, 직사각형 클래스에서 정사각형 클래스가 파생된 구조입니다. 그리고 정사각형은 직사각형이다라고 말할 수 있으니 수학적으로도 의미가 맞습니다.

하지만! **이 생각이 미묘하지만 심각한 문제를 유발할 수 있습니다.**

이 관계가 잘못된 이유

1. Square클래스는 itsHeight와 isWidth 두 개는 필요없습니다. 정사각형 정의에 따라 변 한 개만 주어져도 됩니다.
    - 이는 메모리 낭비로 이어집니다.
2. SetWidth, SetHeight는 정사각형은 w,h가 동일 하므로 매우 부적절한 함수입니다.
    - Square클래스에 맞지 않는 함수가 있으므로 잘못된 설계

방법2 - 오버라이드

1. 메모리는 충분하다 가정하여 1번 문제를 피해가고,
2. 만약에 상속할 때 SetWidth와 SetHeight를 다음과 같이 오버라이드한다면 2번 문제까지 피해갈 수 있습니다.

```
void Square::SetWidth(double w) {
  Rectangle::SetWidth(w);
  Rectangle::SetHeight(w);
}
void Square::SetHeight(double h) {
  Rectangle::SetHeight(h);
  Rectangle::SetWidth(h);
}
```

이와 같이 한다면, **수학적으로** 정사각형이라는 의미가 맞아 떨어지게 됩니다.

이것이 잘못된 이유

```
void f(Rectangle& r) {
  r.SetWidth(32); // calls Rectangle::SetWidth
}
```

이 함수에서 r에 Square 객체의 포인터가 들어간다면 Square 객체는 width와 height가 달라지기 때문입니다. (Rectangle의 SetWidth메소드 사용)

이것은 명백한 LSP 위반이지요.

방법3 - virtual

단순히 Rectangle의 SetWidth와 SetHeight를 virtual로 바꾸면 방법2의 문제를 해결 할 수 있게 됩니다.

[최종 코드]

```
class Rectangle {
  public:
  virtual void SetWidth(double w) {itsWidth=w;}
  virtual void SetHeight(double h) {itsHeight=h;}
  double GetHeight() const {return itsHeight;}
  double GetWidth() const {return itsWidth;}

  private:
  double itsHeight;
  double itsWidth;
};

class Square : public Rectangle {
  public:
  virtual void SetWidth(double w);
  virtual void SetHeight(double h);
};
void Square::SetWidth(double w) {
  Rectangle::SetWidth(w);
```

```
  Rectangle::SetHeight(w);
}
void Square::SetHeight(double h) {
  Rectangle::SetHeight(h);
  Rectangle::SetWidth(h);
}
```

이 코드가 맞아 보이는 부분

1. 수학적으로 정사각형이 생성
2. 수학적으로 직사각형 클래스
3. Rectangle을 참조하는 포인터를 사용하는 함수에 Square를 넘겨줄 수 있음

하지만, 근본적인 문제는 모델 그 자체가 자신을 잘 설명하느냐가 아닌(자기서술적), 이를 **사용하는 유저**에게 달려있습니다.

근본적인 문제를 보여주는 코드

```
void g(Rectangle& r) {
  r.SetWidth(5);
  r.SetHeight(4);
  assert(r.GetWidth() * r.GetHeight()) == 20);
}
```

Rectangle을 사용하는 유저입장에서 생각해봅시다.

r은 직사각형이기 때문에 위처럼 Width를 5로 설정하고, Height를 4로 설정한다면 20이 나온다고 생각할 수 있습니다.

하지만 정사각형이 들어가면 틀리게 되지요.

유저는 Height가 4로 들어갈 때 Height는 5로 고정되어 있다고 가정하게 됩니다.

결론 - LSP는 클래스의 행동에 달려있다.

수학적으로 정사각형이 직사각형이 아니기 때문에 LSP가 위반된 것이 아닙니다.

결국! 정사각형의 행동(behavior)이 직사각형의 행동과 일치하지 않기 때문에 LSP가 위반된 것입니다.

다시 말해서 LSP는 클래스 그 자체로 판단할 수 있는 것이 아닌, 유저가 사용하는 행태에 달려있습니다.

그리고 그 [근본적인 문제를 보여주는 코드]에서 볼 수 있 듯,

유저가 생각하길, 파생 클래스는 베이스 클래스의 행동을 그대로 따를 것이라고 판단합니다 (Design by Contract)

따라서 설계하는 사람은 유저의 행동을 잘 고려하여 클래스를 설계하여야 합니다.