Liskov Substitution Principle (LSP)

Introduction

The **Liskov Substitution Principle (LSP)** is one of the five SOLID design principles in object-oriented programming. It is named after Barbara Liskov and states that:

> "Objects of a superclass should be replaceable with objects of a subclass without affecting the correctness of the program."

This means that a derived class must extend the behavior of the base class without breaking its intended functionality.

Understanding LSP with an Example

Consider the following `Rectangle` class:

```
class Rectangle

{

protected:

  int width, height;

public:

  Rectangle(const int width, const int height)

    : width{width}, height{height} { }

  int get_width() const { return width; }

  virtual void set_width(const int width) { this->width = width; }

  int get_height() const { return height; }

  virtual void set_height(const int height) { this->height = height; }

  int area() const { return width * height; }

};
```

This class provides basic functionality to set and get dimensions while calculating the area.

Now, consider a `Square` class that inherits from `Rectangle`:

```
class Square : public Rectangle

{

public:

  Square(int size): Rectangle(size,size) {}

  void set_width(const int width) override {

    this->width = this->height = width;

  }

  void set_height(const int height) override {

    this->height = this->width = height;

  }

};
```

At first glance, this seems like a reasonable inheritance model, since a square is a special type of rectangle. However, this implementation **violates LSP** when used in a function that expects a `Rectangle` object.

The Problem with LSP Violation

Let's consider a function that processes a rectangle:

```
void process(Rectangle& r)
{
  int w = r.get_width();
  r.set_height(10);

  std::cout << "expected area = " << (w * 10)
    << ", got " << r.area() << std::endl;
}
```

Expected Behavior

When passing a `Rectangle` with width `5` and height `5`, we expect:

Expected area = 50, got 50

Unexpected Behavior with `Square`

If we pass a `Square` of size `5`, calling `set_height(10)` also sets the width to `10`, leading to:

Expected area = 50, got 100

This violates the principle because substituting a `Square` for a `Rectangle` **changes expected behavior**.

Solution: Avoid Inheriting Square from Rectangle

Instead of using inheritance, we can design a factory method that creates distinct shapes:

```
struct RectangleFactory

{

  static Rectangle create_rectangle(int w, int h) {  return Rectangle(w,
h);}

  static Rectangle create_square(int size) { return Rectangle(size, size);
}

};
```

This ensures that a `Rectangle` and a `Square` remain separate entities while adhering to their own constraints.

Key Takeaways

- **A subclass should not alter the expected behavior of its superclass.**
- **Inheritance should only be used when the subclass truly "is-a" type of its superclass.**
- **If modifying inherited behavior leads to unexpected results, consider composition or factory methods instead.**
- **LSP violations often occur when enforcing constraints that don't apply to all derived types.**

By following the **Liskov Substitution Principle**, we create more robust and maintainable object-oriented designs.

Solution: Avoid Inheriting Square from Rectangle

Instead of using inheritance, we can design a factory method that creates distinct shapes:

```cpp
#include <iostream>

class Shape
{
public:
    virtual int area() const = 0;
    virtual ~Shape() = default;
};

class Rectangle : public Shape
{
protected:
    int width, height;
public:
    Rectangle(const int width, const int height)
        : width{width}, height{height} { }

    int get_width() const { return width; }
    void set_width(const int width) { this->width = width; }
    int get_height() const { return height; }
    void set_height(const int height) { this->height = height; }

    int area() const override { return width * height; }
};

class Square : public Shape
{
private:
    int size;
public:
    Square(int size) : size{size} {}

    void set_size(int newSize) { size = newSize; }
    int area() const override { return size * size; }
};

struct ShapeFactory
{
    static Rectangle create_rectangle(int w, int h) { return Rectangle(w,
h); }
```

```cpp
    static Square create_square(int size) { return Square(size); }
};

void process(Shape& shape)
{
    std::cout << "Area = " << shape.area() << std::endl;
}

int main()
{
    Rectangle r = ShapeFactory::create_rectangle(5, 5);
    process(r);

    Square s = ShapeFactory::create_square(5);
    process(s);

    return 0;
}
```

This ensures that a Rectangle and a Square remain separate entities while adhering to their own constraints. The Shape base class enforces a common interface without forcing incorrect inheritance relationships.

Fix Explanation:

1. **Introduced a `Shape` base class** with a pure virtual `area()` method to ensure polymorphism.
2. **Separated `Rectangle` and `Square` into distinct entities**, avoiding incorrect inheritance.
3. **Modified `process()` to accept `Shape&`** instead of `Rectangle&`, ensuring it works correctly for both `Rectangle` and `Square`.
4. **Updated `RectangleFactory` to return separate `Rectangle` and `Square` instances**, preventing incorrect assumptions about inherited behavior.

This implementation adheres to **LSP** by ensuring that objects can be replaced without altering expected behavior.