

# LSP

# Liskov 替换原则

OCP（开闭原则）的主要促成机制



正是使用了继承，我们才可以创建实现其基类中抽象方法的派生类。

# 定义

“

若对类型 S 的每一个对象 o1，都存在一个类型 T 的对象 o2，使得在所有针对 T 编写的程序 P 中，用 o1 替换 o2 后，程序 P 的行为功能不变，则 S 是 T 的子类型。

— Barbara Liskov, 1988

”

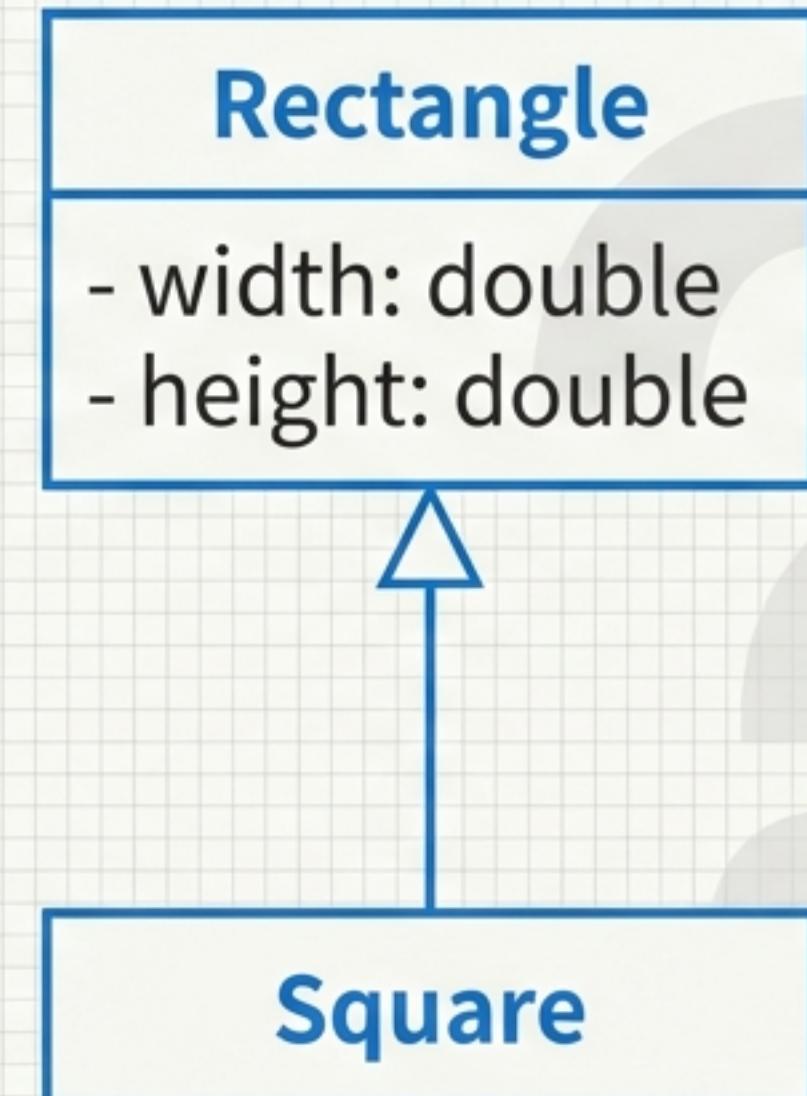
简而言之：子类型必须能够替换掉它们的基类型。

关键洞察：如果一个函数需要通过“运行时类型识别”（RTTI）来判断对象的具体类型，那么它就违反了 LSP。

# 经典的陷阱：正方形是一个矩形吗？

从一般意义上讲，一个正方形就是一个矩形。这种“IS-A”关系通常被认为是继承的基础。

然而，在软件设计中，这种直觉是错误的。



# 代码证据：LSP 违规导致的 OCP 违规

## LSP 违规：Square 重写

```
public class Square : Rectangle
{
    public override double Width
    {
        set
        {
            base.Width = value;
            base.Height = value; ←
        }
    }
    public override double Height
    {
        set
        {
            base.Height = value;
            base.Width = value; ←
        }
    }
}
```

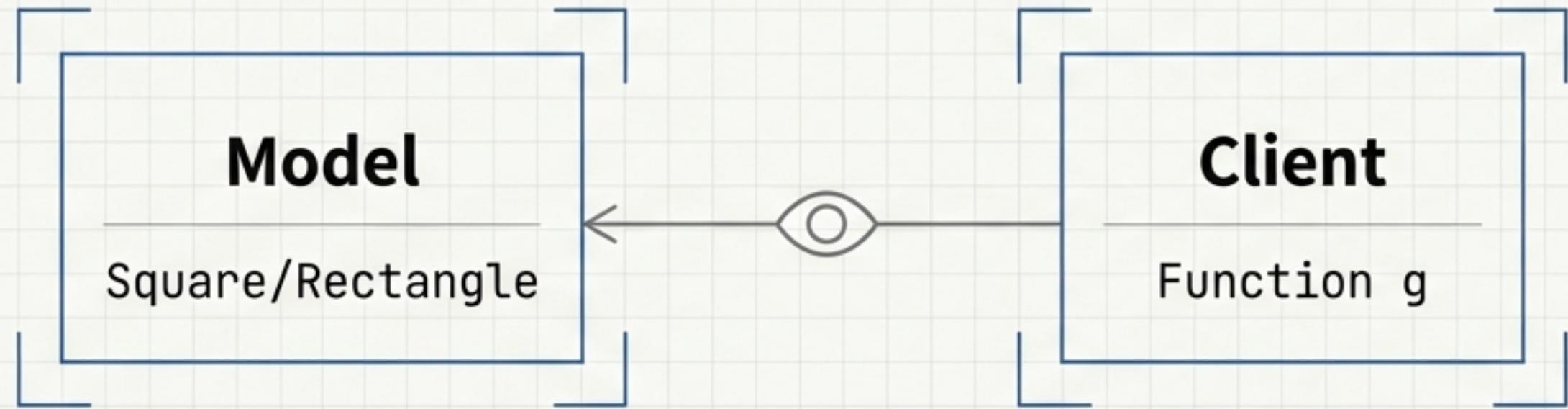
## OCP 违规：函数 g(Rectangle r) 失败

```
void g(Rectangle r)
{
    r.Width = 5;
    r.Height = 4;
    if (r.Area() != 20)
        throw new Exception("Bad area!"); ←
}
```

函数 g 假设宽度和高度是独立变化的。当传递 Square 时，断言失败。

结论：对于函数 g 来说，Square 不是 Rectangle。

# 行为重于结构



模型的有效性只能通过它的客户程序来表现。

1. 孤立地看，Square 和 Rectangle 是自相容的。
2. 但在 g 函数的上下文中，它们是不相容的。

**在 OOD 中，IS-A 关系是就行为方式而言的。**

# 解决方案：基于契约设计

## (Design by Contract - DbC, Bertrand Meyer)

### 类的契约 (Class Contract)



类作者显式地规定契约，客户端和实现端都必须遵守。

# 通过 DbC 验证 LSP

## 「规则 1：前置条件」

Noto Sans SC

派生类只能使用相等或更弱的前置条件。

Noto Serif SC

你不能对客户提出比基类更多的要求。

## 「规则 2：后置条件」

Noto Sans SC

派生类只能使用相等或更强的后置条件。

Noto Serif SC

你必须至少遵守基类承诺的所有规则。

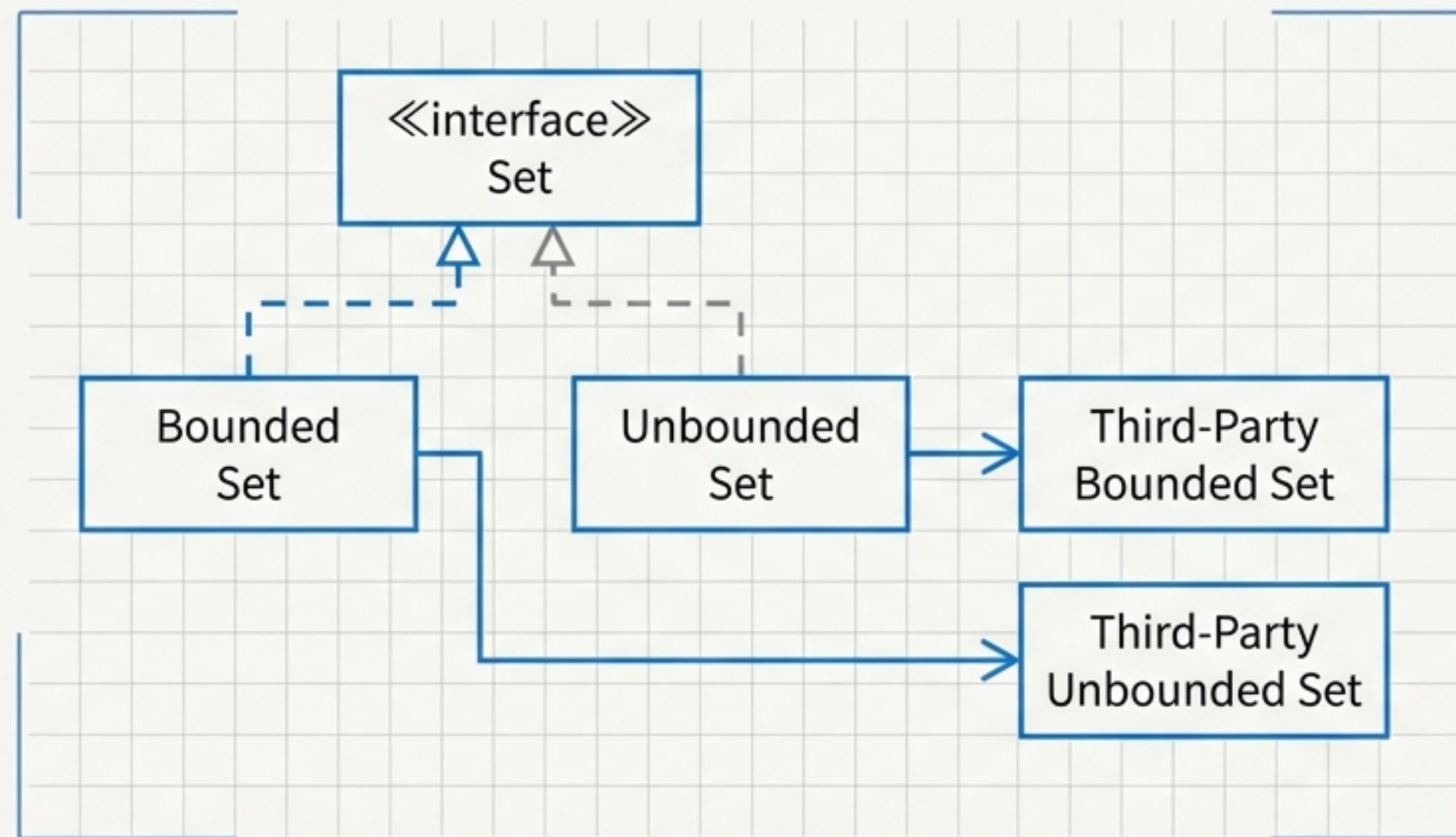
## 「Rectangle 违规案例：」

JetBrains Mono

Rectangle.setWidth 的后置条件承诺：`(width == w) && (height == old.height)`。

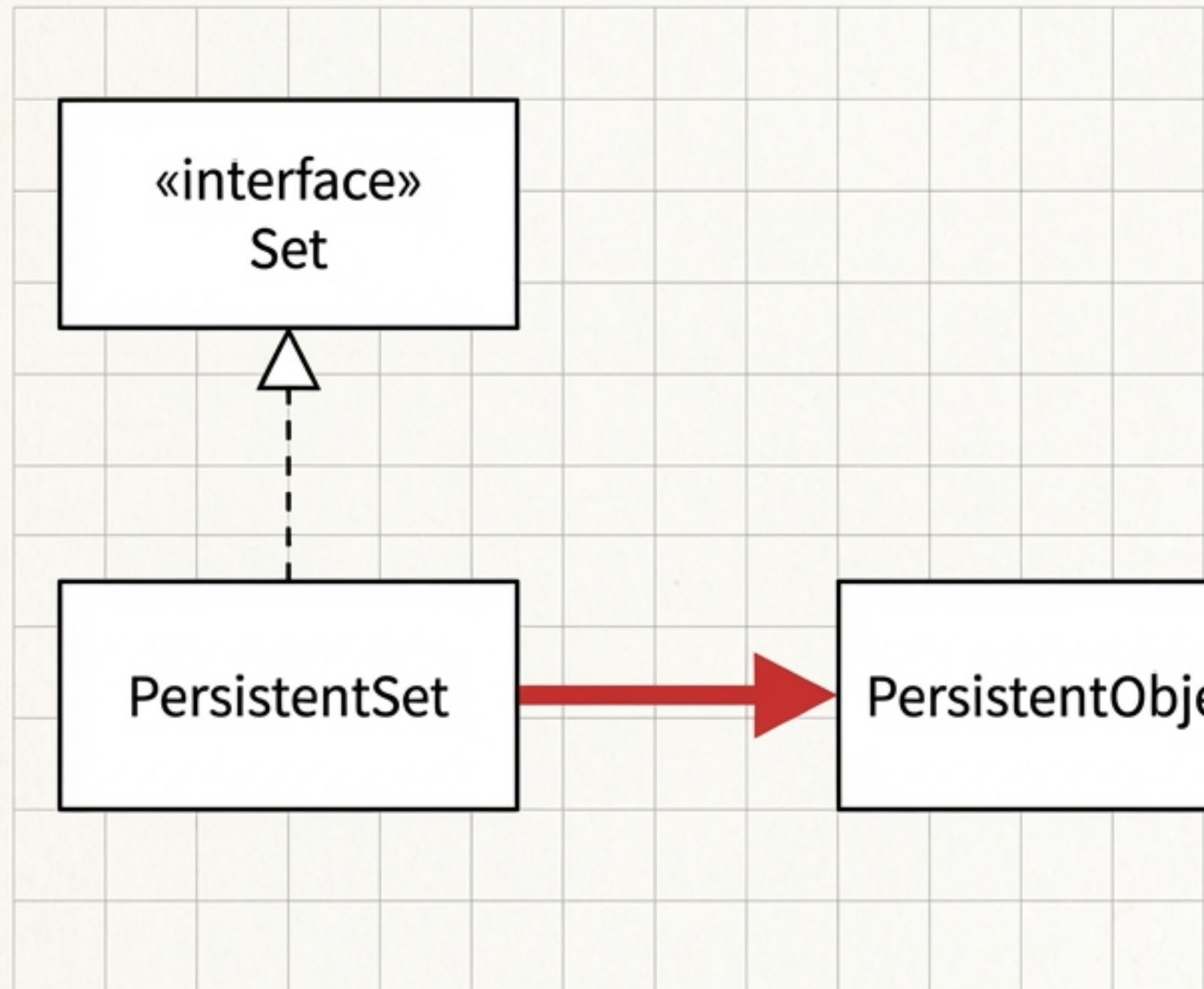
Square 改变了 height，因此违反了后置条件。

# 实战案例：第三方集合库



我们定义了一个抽象接口 Set 来统一访问。这符合 LSP：客户代码不需要知道使用的是哪种 Set。

# 问题出现：持久化集合

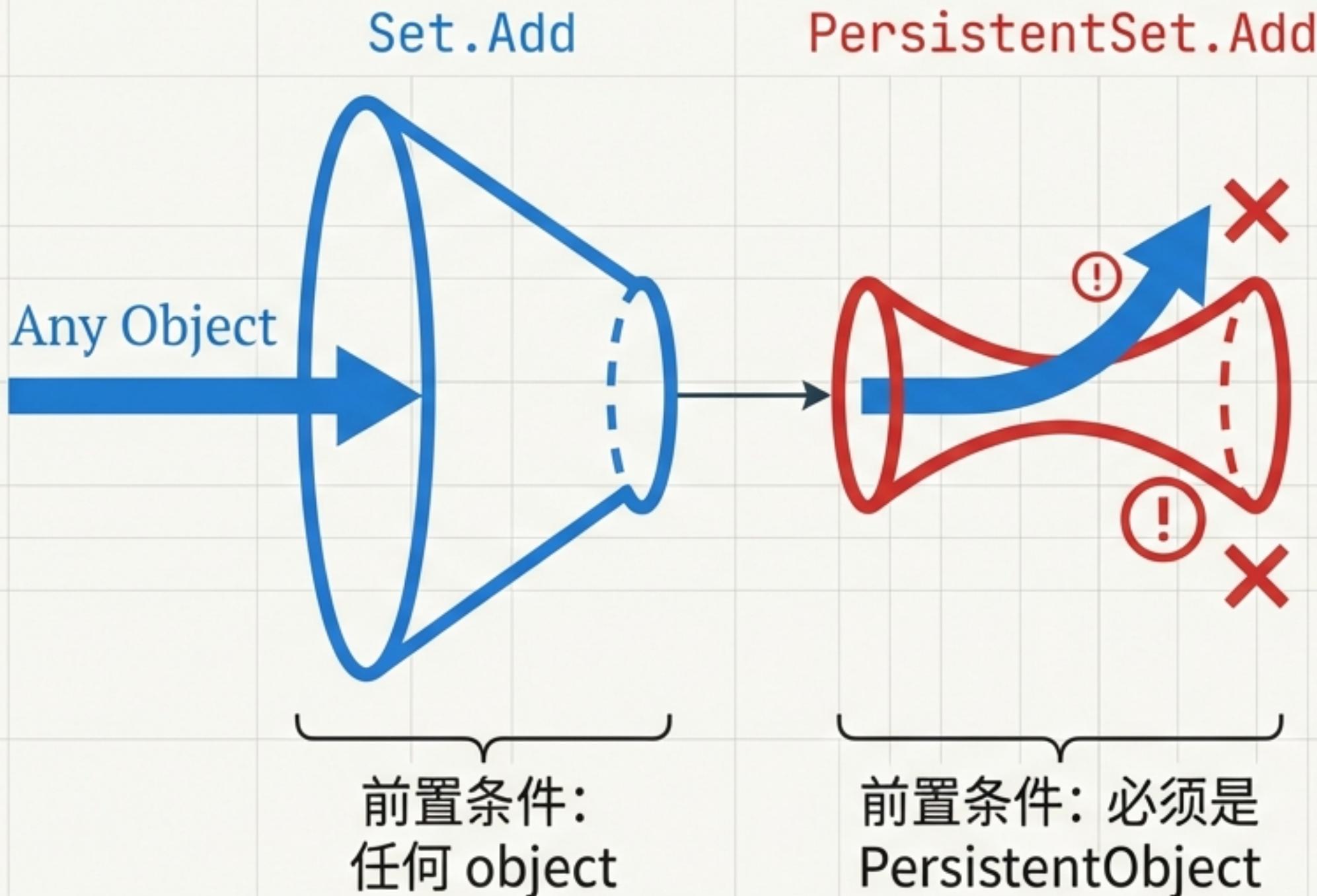


需求：添加 PersistentSet 以便写入流。

约束：PersistentSet 只能接受派生自 PersistentObject 的对象。

冲突：Set 接口允许添加任何 object。PersistentSet 破坏了这个约定。

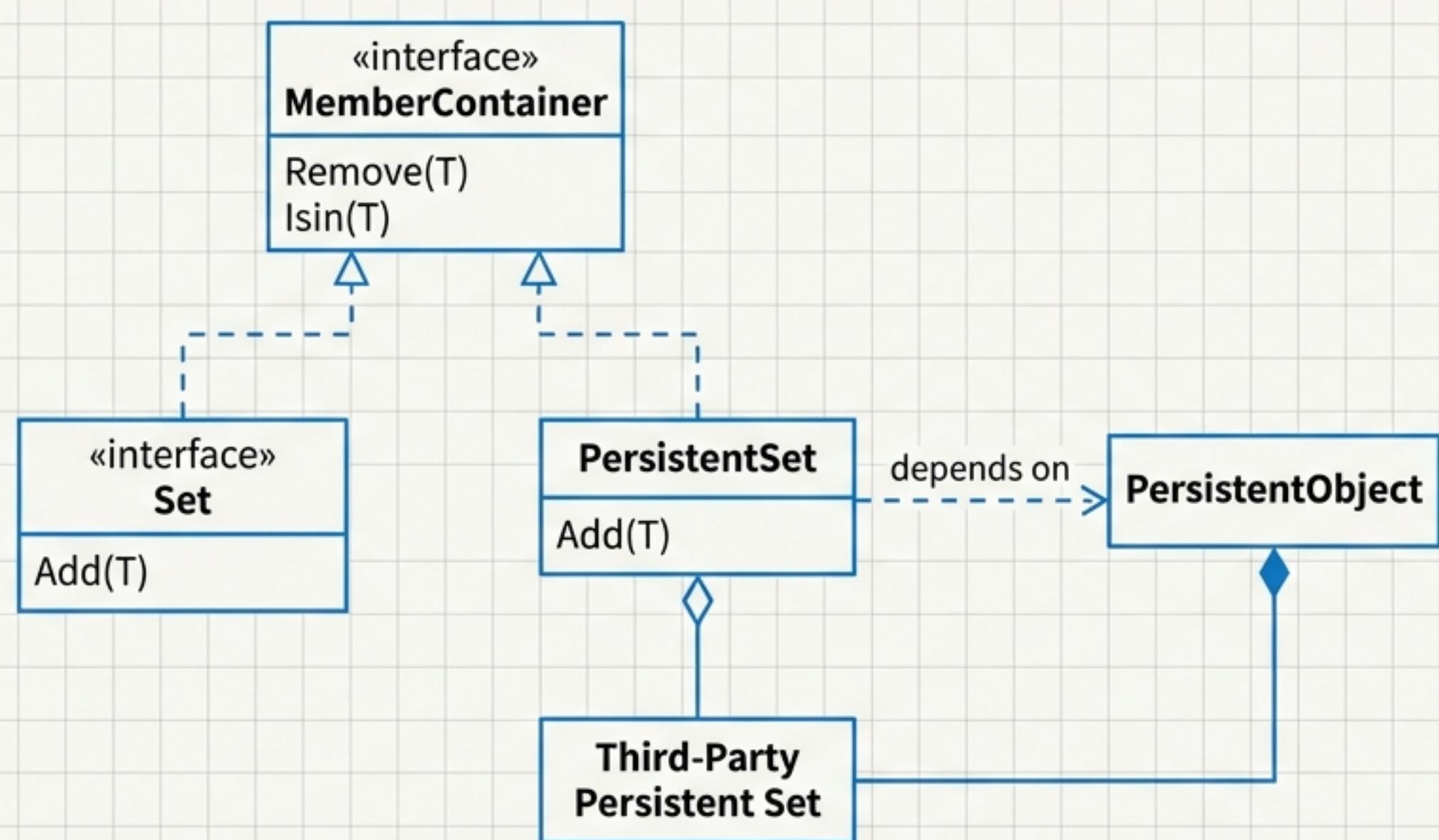
# 分析：前置条件的违规



1. 基类 `Set.Add` 前置条件：  
参数可以是任何 `object`。
2. 派生类 `PersistentSet.Add`  
前置条件：参数必须是  
`PersistentObject`。

**前置条件被加强了。  
这是 LSP 违规。**

# 解决方案：分离层次结构

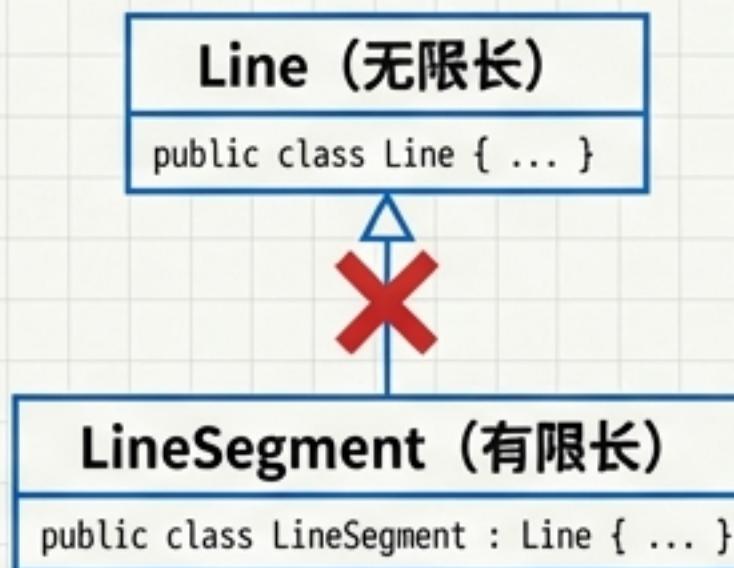


`PersistentSet` 和 `Set` 之间不存在 IS-A 关系。

- 策略：将它们完全分离。如果不必须通用，就不应强行继承。
- 我们可以提取一个更通用的接口，只有在它们确实有公共行为时。

# 另一种修复策略：提取公共部分

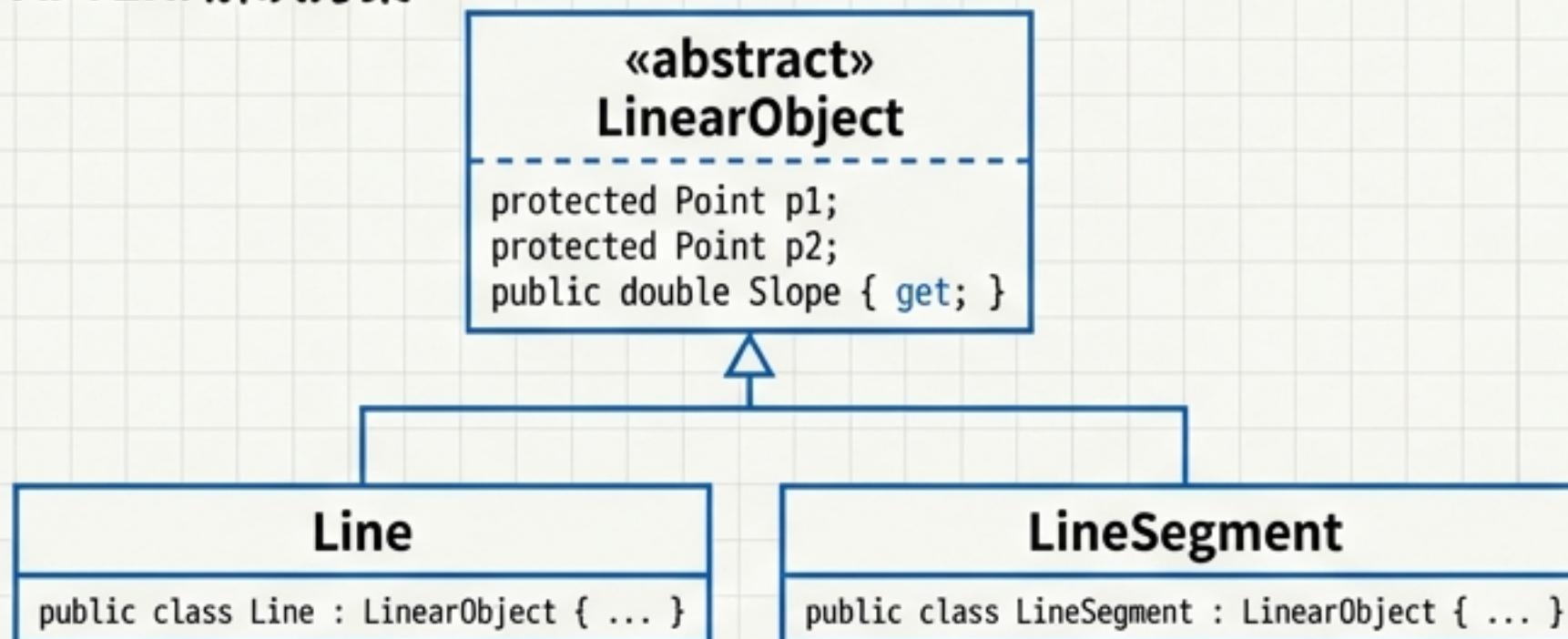
BEFORE: 问题分析



LineSegment 继承 Line 会导致计算混乱。

} Violates LSP: Finite cannot substitute Infinite.

AFTER: 解决方案



提取一个公共基类 **LinearObject**, 只包含两者共有的属性。

- **LinearObject**: 包含 Points, Slope
- Line 和 LineSegment 独立继承基类
- 避免了不合理的 IS-A 关系

} Refactored Hierarchy: Shared Structure Extracted.

# 启发式规则：如何发现 LSP 违规



## 退化函数

派生类中的函数什么也不做。



## 抛出异常

派生类的方法抛出  
NotImplementedException，  
而基类不抛出。



## 类型检查

代码中出现 `if (o is SomeType)` 或 `instanceof`。

---

“任何针对特定派生类的特殊处理都是 LSP 违规的迹象。”

---

# 结论 (Conclusion)



若移除 LSP, OCP 将崩溃

- OCP 是 OOD 的核心目标。
- LSP 是实现 OCP 的主要机制。
- 正是子类型的可替换性，使得使用基类类型的模块无需修改即可扩展。

**必须通过契约来定义子类型，而不仅仅是遵循语法上的继承。**

# 参考文献

**Liskov88:** Barbara Liskov, "Data Abstraction and Hierarchy," SIGPLAN Notices, 23(5) (May 1988).

**Meyer97:** Bertrand Meyer, Object-Oriented Software Construction, 2nd ed., Prentice Hall, 1997.

**Martin:** Robert C. Martin, Agile Principles, Patterns, and Practices.