SWE 4743:
Object-Oriented Design

Jeff Adkisson

Liskov Substitution Principle

Designing Correct Subtypes Without Surprise

Designing Correct Subtypes Without Surprise

- The Liskov Substitution Principle is about **trust**. When a client uses a base type, it should not need to know—or care—which subtype it receives.
  - Subtypes must honor the **semantic promises** of their base types, not reinterpret them.
  - When they do not, inheritance becomes a source of bugs and surprises rather than a means of reuse.
  - #### What is "Semantic"?

Table of Contents

Positioning LSP Within SOLID

- The **Liskov Substitution Principle (LSP)** is the principle that most clearly explains *why bad inheritance hurts so badly*.
  - If **SRP** is about *why code changes*<br>and **OCP** is about *where changes should land*<br>
  - then **LSP** is about *whether abstractions can be trusted at all*.
  - Without LSP:
  - Polymorphism becomes dangerous
  - Abstractions can mislead when their guarantees are unclear.

The Original Definition of LSP

- Barbara Liskov (1987):
  - In plain language:

What LSP Is Really About

- LSP is **not** about:
  - Syntax
  - Method signatures
  - Inheritance trees
  - LSP *is* about:
  - **Behavior**

LSP as a Contract, Not Inheritance

- Inheritance is a *mechanism*.
  - LSP is a *promise*.
  - classDiagram
  - direction TB
  - class BaseType {
  - +operation()

Classic Bad Example: Rectangle / Square

- ### Base Class
  - public class Rectangle
  - {
  - public virtual int Width { get; set; }
  - public virtual int Height { get; set; }
  - public int Area() => Width * Height;

Why the Rectangle Example Fails

- The base class contract implies:
  - Width and height are independent
  - Setting one does not affect the other
  - **`Square` violates that contract**.
  - Inheritance was legal.<br>Substitution was not.

A Corrected Design

- The fix is **not** clever overriding.
  - The fix is modeling that **preserves substitutability**.
  - classDiagram
  - direction TB
  - class Shape {
  - <<interface>>

Behavioral Subtyping Rules

- A subtype must:
  - 1. **Not strengthen preconditions**
  - 2. **Not weaken postconditions**
  - 3. **Preserve invariants**
  - These rules define behavioral compatibility.

Preconditions, Postconditions, and Invariants

- ### Preconditions
    - What must be true *before* a method runs
    - ### Postconditions
    - What is guaranteed *after* the method completes
    - ### Invariants
    - What must *always* remain true

Strengthening Preconditions (Bad)

- public class FileLogger
  - {
  - public virtual void Log(string? message)
  - {
  - // accepts any string
  - }

Weakening Postconditions (Bad)

- public class OrderProcessor
  - {
  - // Contract:
  - // - Processes the order
  - // - Persists the result
  - // - Returns true on success

Exception-Based LSP Violations

- Throwing **new exceptions** from overrides is often an LSP violation.
  - #### Base Class
  - public class UserRepository
  - {
  - // Contract: returns null if user is not found
  - public virtual User? FindById(Guid id)

LSP Across Abstraction Levels

- LSP applies to:
  - Classes
  - Interfaces
  - APIs
  - Services
  - Microservices

LSP and Single Responsibility Principle

- Single Responsibility makes LSP possible.
  - If a class has multiple responsibilities:
  - Some subtypes satisfy one responsibility
  - Others satisfy another
  - No subtype satisfies all
  - Multiple responsibilities almost guarantee LSP violations.

LSP and Open-Closed Principle

- The Open-Closed Principle *depends* on LSP.
  - You cannot safely extend behavior if subtypes cannot be reliably substituted.

Practical Heuristics for LSP

- Ask these questions:
  - Would I be surprised by this behavior?
  - Does this subtype restrict valid inputs?
  - Does it remove guarantees?
  - Does client code need `if (x is SubType)`?
  - If yes → LSP is violated.

Code Smells That Signal LSP Violations

- Overridden methods that throw
  - Subtypes with unused methods
  - Boolean flags in subclasses
  - Client-side type checks
  - “This works except when…”
  - `NotImplementedException`

Conclusion: Substitutability Is Trust

- LSP is about **trust**.
  - When you depend on an abstraction:
  - You trust its promises
  - You assume consistency
  - You rely on behavior, not type
  - Breaking LSP breaks that trust.