# Applying the Liskov Substitution Principle (LSP)

**Dan Geabunea**
PASSIONATE SOFTWARE DEVELOPER | BLOGGER

@romaniancoder www.romaniancoder.com

# Overview

What is the Liskov Substitution Principle?

Violations of the Liskov Substitution Principles

Fixing incorrect relationships between types

Demo: Refactor class to respect the Liskov Substitution Principle

# Liskov Substitution Principle

If S is a subtype of T, then objects of type T in a program may be replaced with objects of type S without modifying the functionality of the program.

# Liskov Substitution Principle

Any object of a type must be substitutable by objects of a derived typed without altering the correctness of that program.

# Relationships

## Is a

Square is a kind of rectangle

An ostrich is a bird

## Is substitutable by

Is the class rectangle fully substitutable by the class square?

Incorrect relationships between types cause unexpected bugs or side effects.

# Violations of the Liskov Substitution Principle

# Empty Methods/Functions

```
class Bird{
    public void fly(int altitude){
        setAltitude(altitude);
        // Fly logic
    }
}
```

# Empty Methods/Functions

```
class Ostrich extends Bird{
    @Override public void fly(int altitude){
        // Do nothing; An ostrich can't fly
    }
}

Bird ostrich = new Ostrich();
ostrich.fly(1000);
```

# Harden Preconditions

```
class Rectangle {
    public void setHeight(int height){}
    public void setWidth(int width(){}

    public int calculateArea(){
        return this.width * this.height;
    }
}
```

# Harden Preconditions

```
class Square extends Rectangle {
    public void setHeight(int height){
        // Width must be equal to height
        this.height = height;
        this.width = height;
    }
    public void setWidth(int width){} // Same logic w = h
}
```

# Harden Preconditions

```
Rectangle r = new Square();
r.setWidth(10);
r.setHeight(20);

r.calculateArea(); // Will return 400
```

# Partial Implemented Interfaces

```
interface Account{
    void processLocalTransfer(double amount);
    void processInternationalTransfer(double amount);
}
```

# Partial Implemented Interfaces

```
class SchoolAccount implements Account{
    void processLocalTransfer(double amount){
        // Business logic here
    }
    void processInternationalTransfer(double amount){
        throw new RuntimeException("Not Implemented");
    }
}
```

# Partial Implemented Interfaces

```
Account account = new SchoolAccount();
account.processInternationalTransfer(10000); // Will crash
```

# Type Checking

```
for (Task t : tasks){
    if(t instanceof BugFix){
        BugFix bf = (BugFix)t;
        bf.initializeBugDescription();
    }
    t.setInProgress();
}
```

# Fixing Incorrect Relationships

# Two Ways to Refactor Code to LSP

Eliminate incorrect relations between objects

Use “Tell, don’t ask!” principle to eliminate type checking and casting

# Empty Methods/Functions

```
class Ostrich extends Bird{
    @Override public void fly(int altitude){
        // Do nothing; An ostrich can't fly
    }
}
```

# Fixed Empty Methods/Functions

```
class Bird {
    // Bird data and capabilities
    public void fly(int altitude){…}
}
class Ostrich {
    // Ostrich data and capabilities. No fly method
}
```

# Partial Implemented Interfaces

```
class SchoolAccount implements Account{
    void processLocalTransfer(double amount){
        // Business logic here
    }
    void processInternationalTransfer(double amount){
        throw new RuntimeException("Not Implemented");
    }
}
```

# Fixed Partial Implemented Interfaces

```
class SchoolAccount implements LocalAccount{
    void processLocalTransfer(double amount){
        // Business logic here
    }
}
```

# Type Checking

```
for (Task t : tasks){
    if(t instanceof BugFix){
        BugFix bf = (BugFix)t;
        bf.initializeBugDescription();
    }
    t.setInProgress();
}
```

# Fixed Type Checking

```
class BugFix extends Task{
    @Override
    public void setInProgress(){
        this.initializeBugDescription();
        super.setInProgress();
    }
}
```

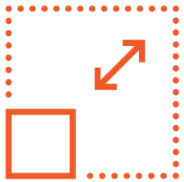
# Tell, Don't Ask

```
for (Task t : tasks){

    // Task should be replaceable by BugFix
    t.setInProgress();

}
```

# Apply the LSP in a Proactive Way



Make sure that a derived type can substitute its base type completely

Keep base classes small and focused

Keep interfaces lean

# Demo

## Applying the LSP

- Observe the problems of incorrect type hierarchies
- Refactor code using LSP

# Summary

Don’t think relationships in terms of “IS A”

Empty methods, type checking and hardened preconditions are signs that you are violating the LSP

LSP also applies for interfaces, not just for class inheritance

Most times you fix an incorrect type hierarchy by breaking it

Real life categories do not always map to OOP relationships.

“If it looks like a duck, quacks like a duck, but needs batteries – you probably have the wrong abstraction”

**SOLID Motivational Poster**