Software Engineering 2

Clean Code – SOLID – Liskov Substitution Principle

Dr. Ahmed Hesham Mostafa

Liskov Substitution Principle
LSP

- Substitutability is a principle in object-oriented programming introduced by Barbara Liskov in a 1987 conference keynote stating that, if class B is a subclass of class A, then wherever A is expected, B can be used instead:
- Objects of a superclass should be replaceable with objects of its subclasses without breaking the system.
- The Liskov Substitution Principle states that any class that is the child of a parent class should be usable in place of its parent without any unexpected behavior.
- LSP implies that a subclass should not be more restrictive than the behavior specified by the superclass.

Liskov Substitution Principle LSP

- This principle ensures that inheritance (one of the OOP principles) is used correctly.
- If an override method does nothing or just throws an exception, then you're probably violating the LSP.
- It can apply to inheritance or interface.

The four conditions for abiding by the Liskov Substitution principle are as follows:

- **Condition 1 Contravariance:** Method signatures must match: Methods must take the same parameters; the parameters of the overriding methods should be the same or more generic than the types of the parent's method parameters (it is not allowed in java but allowed in PHP to generalize the parameters for overloaded method).
- **Condition 2:** **Preconditions cannot be strengthened in the subtype** The preconditions for any method can't be greater than that of its parent; Any inherited method should not have more conditionals that change the return of that method, such as throwing an Exception. For example, if the parent method in the parent class accepts an integer number while in an overridden method in the child class, you added a condition to accept the only positive integer number, that change in the child class violates LSP.

The four conditions for abiding by the Liskov Substitution principle are as follows:

- Condition 3: **Postconditions cannot be weakened in the subtype** Postconditions must be at least equal to that of its parent; Inherited methods should return the same type as that of their parent. For example, if the method in the parent class creates a connection to DB and at the end of the method, you close the connection, while in the child you keep the connection open, that changes in the child class violate LSP.
- Condition 4: **Exception types must match;** If a method is designed to return a FileNotFoundException in the event of an error, the same condition in the inherited method must return a FileNotFoundException too.

Liskov Substitution Principle LSP

- When you’re trying to understand the Liskov Substitution Principle, it can be hard to find a brief explanation of the rules.
- The rules are simple enough to state, but the implications can be tricky to wrap your head around.
- The impact of a violation of the Liskov Substitution Principle can be significant, and it can lead to a lot of time spent fixing bugs.
- That’s why Liskov Substitution Principle is important if your existing code contains a lot of supertype-subtype relationships.

Vehicle Example: Violate LSP

```
class Vehicle
{
  String name;
  String getName() { ... }
  void setName(String n) { ... }
  double speed;
  double getSpeed() { ... }
  void setSpeed(double d) { ... }

  Engine engine;
  Engine getEngine() { ... }
  void setEngine(Engine e) { ... }
  void startEngine() { ... }
}
```

```
class Car extends Vehicle
{
@Override
void startEngine() { ... }
}
```

```
class Bicycle extends Vehicle
{
@Override
void startEngine(){
throw new Expcetion("Not supported");
}
 /*problem!*/
}
```

- There is no problem here, right? A car **is a** Vehicle, and here we can see that it overrides the startEngine() method of its superclass.

- a bicycle **is a** Vehicle, however, it does not have an engine and hence, startEngine() method cannot be implemented. So, it violates LSP

Vehicle Example: Violate LSP

- **Now we create a list of objects of Car and** Bicycle in store it in list vehicleList
- **When the children call the method** startEngine(), the car object works fine while for the Bicycle object it shows an exception.

```
public class Befor_LiskovSubstitutionPrinciple {
public static void main(String[] args) {
      ArrayList< Vehicle > vehicleList =new ArrayList();
      Vehicle BMW =new Car();
      vehicleList.add(BMW);
      Vehicle bike =new Bicycle();
      vehicleList.add(bike);
      for(Vehicle v : vehicleList){
       v. startEngine();
}
}
}
```

Vehicle Example: Violate LSP

- These are the kinds of problems that violation of the Liskov Substitution Principle leads to, and they can most usually be recognized by a method that does nothing or even can’t be implemented.
- The solution to these problems is a correct inheritance hierarchy, and in our case, we would solve the problem by differentiating classes of vehicles with and without engines.
- Even though a bicycle is a vehicle, it doesn’t have an engine.

Vehicle Example: Following LSP

```
class Vehicle
{
   String name;
   String getName() { ... }
   void setName(String n) { ... }
   double speed;
   double getSpeed() { ... }
   void setSpeed(double d) { ... }

}
```

```
class VehicleWithEngines extends Vehicle
{
Engine engine;
Engine getEngine() { ... }
void setEngine(Engine e) { ... }
void startEngine() { ... }
}
```

```
class Car extends VehicleWithEngines
{
@Override
void startEngine() { ... }
}
```

```
class VehicleWithoutEngines extends Vehicle
{
void startMoving() { ... }
}
```

```
class Bicycle extends VehicleWithoutEngines{
@Override
void startMoving() { ... }
}
```

Object-Oriented Design can violate the LSP in the following situations

- If a subclass returns an object that is completely different from what the superclass returns.
- If a subclass throws an exception that is not defined in the superclass.
- There are any side effects in subclass methods that were not part of the superclass definition.

Example of LSP violation in the .NET framework

- Sometimes is hard to not violate the principle. Take Microsoft for example. They have some of the best developers in the world working on the C# language. But even they couldn’t foresee how the .NET framework will evolve.
- Today, in .NET you can call the Add() method arrays. It will not work; it will throw a NotSupportException. But why can we do it? Because the Array class implements the IList interface which defines the method.
- This problem appeared with .NET 2.0 (when generics were introduced) and since Microsoft didn’t want to break the backward compatibility, they made this compromise

Notes about override method in java

Notes about override method in java

- Before Java 5.0, when you override a method, both parameters and return type must match exactly.
- Java 5.0 it introduces a new facility called covariant or **Covariance** return type.
- You can override a method with the same signature but return a subclass of the object returned.
- In another words, a method in a subclass can return an object whose type is a subclass of the type returned by the method with the same signature in the superclass.

Notes about override method in java

- It can not change the return type for the override method in java except in some cases.
- It is possible. returns type can be different only if the parent class method return type is a supertype of the child class method return type. This is called covariant
- As in this example we can change return type for method1() from Circle to Square as class Circle is supertype of class Square

```
class ParentClass {
    public Circle method1(Circle c) {
        return new Circle();
    }
}
Class Circle {

}

class Square extends Circle {

}

class ChildClass extends ParentClass {
    @ Override
    public Square method1(Circle c) {
        return new Square();
    }}
```

Software Engineering 2

Clean Code – SOLID – Interface Segregation Principle

Dr. Ahmed Hesham Mostafa

Interface Segregation Principle (ISP)

- The Interface Segregation Principle was defined by Robert C. Martin while consulting for Xerox to help them build the software for their new printer systems. He defined it as:
- “Clients should not be forced to depend upon interfaces that they do not use.”
- The goal of this principle is to **reduce the side effects of using larger interfaces by breaking application interfaces into smaller ones**. It's like the Single Responsibility Principle, where each class or interface serves a single purpose.
- Precise application design and correct abstraction is the key behind the Interface Segregation Principle. **Though it'll take more time and effort in the design phase of an application and might increase the code complexity, in the end, we get a flexible code.**

Interface Segregation Principle (ISP)

- First, no class should be forced to implement any method(s) of an interface they don’t use.
- Secondly, instead of creating large or you can say fat interfaces, create multiple smaller interfaces with the aim that the clients should only think about the methods that are of interest to them.

Example Printer : Violate Interface Segregation Principle (ISP)

- As you can see in the above diagram, we have an interface i.e. IPrinterTasks declared with four methods.
- Now if any class wants to implement this interface, then that class should have to provide the implementation to all the four methods of the IPrinterTasks interface.
- HPLaserJetPrinter wants all the services provided by the IPrinterTasks.
- LiquidInkjetPrinter class the Fax and PrintDuplex methods are not required by the class but, still, it is implementing these two methods. This is violating the Interface Segregation Principle

```
public interface PrinterTasks {
    void Print(String PrintContent);
    void Scan(String ScanContent);
    void Fax(String FaxContent);
    void PrintDuplex(String PrintDuplexContent);
    }
```

```
public class HPLaserJetPrinter implements PrinterTasks{

   @Override
   public void Print(String PrintContent) {
         System.out.println(PrintContent);
   }

   @Override
   public void Scan(String ScanContent) {
         System.out.println(ScanContent);
   }

   @Override
   public void Fax(String FaxContent) {
         System.out.println(FaxContent);
   }

   @Override
   public void PrintDuplex(String PrintDuplexContent) {
         System.out.println(PrintDuplexContent);
   }

}
```

```
public class LiquidInkjetPrinter implements PrinterTasks{

   @Override
   public void Print(String PrintContent) {
         System.out.println(PrintContent);
   }

   @Override
   public void Scan(String ScanContent) {
         System.out.println(ScanContent);
   }

   @Override
   public void Fax(String FaxContent) {
      throw new UnsupportedOperationException("Not
supported yet.");
   }
   @Override
   public void PrintDuplex(String PrintDuplexContent) {
      throw new UnsupportedOperationException("Not
supported yet.");
   }
   }
```

Example Printer : Follow Interface Segregation Principle (ISP)

- we have split that big interface into four small interfaces. Each interface now has some specific purpose.
- Now if any class wants all the services, then that class needs to implement all the four interfaces as shown below.

```
public interface PrintingTasks {
        public void Print(String PrintContent);
}
```

```
public interface ScanTasks {
   void Scan(String ScanContent);
}
```

```
public interface FaxTasks {
   void Fax(String FaxContent);
}
```

```
public interface DuplexTasks {
        void PrintDuplex(String PrintDuplexContent);
}
```

```java
public class HPLaserJetPrinter implements PrintingTasks,
ScanTasks, FaxTasks, DuplexTasks {

    @Override
    public void Print(String PrintContent) {
        System.out.println(PrintContent);
    }

    @Override
    public void Scan(String ScanContent) {
        System.out.println(ScanContent);
    }

    @Override
    public void Fax(String FaxContent) {
        System.out.println(FaxContent);
    }

    @Override
    public void PrintDuplex(String PrintDuplexContent) {
        System.out.println(PrintDuplexContent);
    }
}
```

```java
public class LiquidInkjetPrinter implements
PrintingTasks, ScanTasks {

    @Override
    public void Print(String PrintContent) {
        System.out.println(PrintContent);
    }

    @Override
    public void Scan(String ScanContent) {
        System.out.println(ScanContent);
    }


}
```

References

- Liskov Substitution Principle - Spring Framework Guru
- Liskov Substitution Principle in Java with Example (javaguides.net)
- Liskov Substitution Principle in C#: Cash In on Subclassing (methodpoet.com)
- https://levelup.gitconnected.com/the-liskov-substitution-principle-made-simple-5e69165e7ab5
- SOLID: Part 3 - Liskov Substitution & Interface Segregation Principles (tutsplus.com)
- SEforSDL - Principles: Liskov Substitution Principle (se-education.org)
- GitHub - akayibrahim/Solid-Principles-Tutorial: Tutorial For SOLID Principles
- Liskov Substitution Principle in PHP (in Arabic) (linkedin.com)
- SOLID Design Principles Explained: Interface Segregation with Code Examples (stackify.com)
- Interface Segregation Principle in Java | Baeldung
- Interface Segregation Principle in C# with Examples - Dot Net Tutorials
- c# - Why does calling Add throw an exception in a collection cast to ICollection<T>? - Stack Overflow
- Robert C. Martin, “Clean Code: A Handbook of Agile Software Craftsmanship”.