Введение в программирование на Java

Лекция 9. ООП (часть 4). Принципы SOLID.

Виталий Олегович Афанасьев

17 марта 2025

Принципы SOLID

- S
- O
- L
- I
- D

Принципы SOLID

- **S** — Single Responsibility Principle
- **O** — Open-Closed Principle
- **L** — Liskov Substitution Principle
- **I** — Interface Segregation Principle
- **D** — Dependency Inversion Principle

Single Responsibility Principle

Принцип единственной ответственности:

> *У программной сущности должна быть только одна причина для изменения.*

В другой формулировке:

> *То, что изменяется вместе, должно тоже храниться вместе.*

Single Responsibility Principle: Пример (1)

Представим, что мы описываем программу, собирающую информацию с датчиков.

В качестве информации служит температура и влажность. Недолго думая мы создаём следующий класс:

```
1 public class TempAndHumidity {
2     // 1. Какие-то поля и методы, связанные с температурой
3     // 2. Какие-то поля и методы, связанные с влажностью
4 }
```

Single Responsibility Principle: Пример (2)

Позже к нам приходит наш коллега Петя с просьбой переиспользовать наш класс, но только для работы с температурой.

Как это сделать? В данном случае — только copy-paste:

```
1 public class Temp {
2     // 1. Какие-то поля и методы, связанные с температурой
3 }
```

Single Responsibility Principle: Пример (2)

Позже к нам приходит наш коллега Петя с просьбой переиспользовать наш класс, но только для работы с температурой.

Как это сделать? В данном случае — только copy-paste:

```
1 public class Temp {
2     // 1. Какие-то поля и методы, связанные с температурой
3 }
```

Другой наш коллега Вася хочет воспользоваться только частью класса про влажность. Снова copy-paste:

```
1 public class Humidity {
2     // 2. Какие-то поля и методы, связанные с влажностью
3 }
```

Single Responsibility Principle: Пример (3)

Найдя баг в коде класса Васи тестировщики исправили его.

При этом про оригинальный класс `TempAndHumidity` тестировщики не догадываются, и баг в нём исправлен не будет.

Single Responsibility Principle: God Object

С понятием единственной ответственности тесно связан анти-паттерн **"God Object"**.

God object — объект, который берёт на себя слишком много ответственности, имеет большое число функций и хранит практически все данные.

В коде такого объекта разобраться довольно сложно, как и поддерживать его.

Практически при любом изменении в требованиях данный класс будет меняться.

Open-Closed Principle

Принцип открытости/закрытости:

> *Программные сущности должны быть открыты для расширения, но закрыты для изменения.*

Open-Closed Principle: Пример (1)

Допустим, что мы написали свой алгоритм сортировки. При этом при реализации мы упорядочиваем числа по неубыванию.

```
1 public int[] myCoolSort(int[] values) {
2     ...
3     boolean ordered = a <= b;
4     ...
5 }
```

Как можно воспользоваться данной сортировкой, если требуется отсортировать массив по невозрастанию?

Open-Closed Principle: Пример (2)

Правильным способом будет ввести абстракцию для упорядоченности чисел:

```
1  public interface Comparator {
2      /**
3       * Возвращает < 0, если first должен стоять левее second
4       *   (first 'меньше' second)
5       * Возвращает > 0, если first должен стоять правее second
6       *   (first 'больше' second)
7       * Возвращает 0, если first и second могут стоять в любом порядке
8       *   (first 'равно' second)
9       */
10     int compare(int first, int second);
11 }
```

Open-Closed Principle: Пример (3)

Тогда сортировка принимает следующий вид:

```
1 public int[] myCoolSort(int[] values, Comparator comparator) {
2     ...
3     boolean ordered = comparator.compare(a, b) <= 0;
4     ...
5 }
```

Open-Closed Principle: Пример (4)

Пример использования такой сортировки:

```
1 public class NonDescending implements Comparator {
2     @Override
3     public int compare(int first, int second) {
4         return first - second;
5     }
6 }
```

```
1 public class NonAscending implements Comparator {
2     @Override
3     public int compare(int first, int second) {
4         return second - first;
5     }
6 }
```

```
1 int[] values = ...;
2 int[] sortedValues = myCoolSort(values, new NonAscending());
```

Open-Closed Principle: Пример №2

Другой пример OCP — уже знакомая нам иерархия из фигур.

Добавление новых фигур подразумевает добавление новых классов.

Но при этом код функции `totalArea` не приходится менять.

```
1  public interface Shape {
2      double area();
3  }
4  public class Circle implements Shape { ... }
5  public class Rectangle implements Shape { ... }
6
7  public double totalArea(Shape... shapes) {
8      double sum = 0;
9      for(int i = 0; i < shapes.length; i++) {
10         sum += shapes[i].area();
11     }
12     return sum;
13 }
```

Liskov Substitution Principle

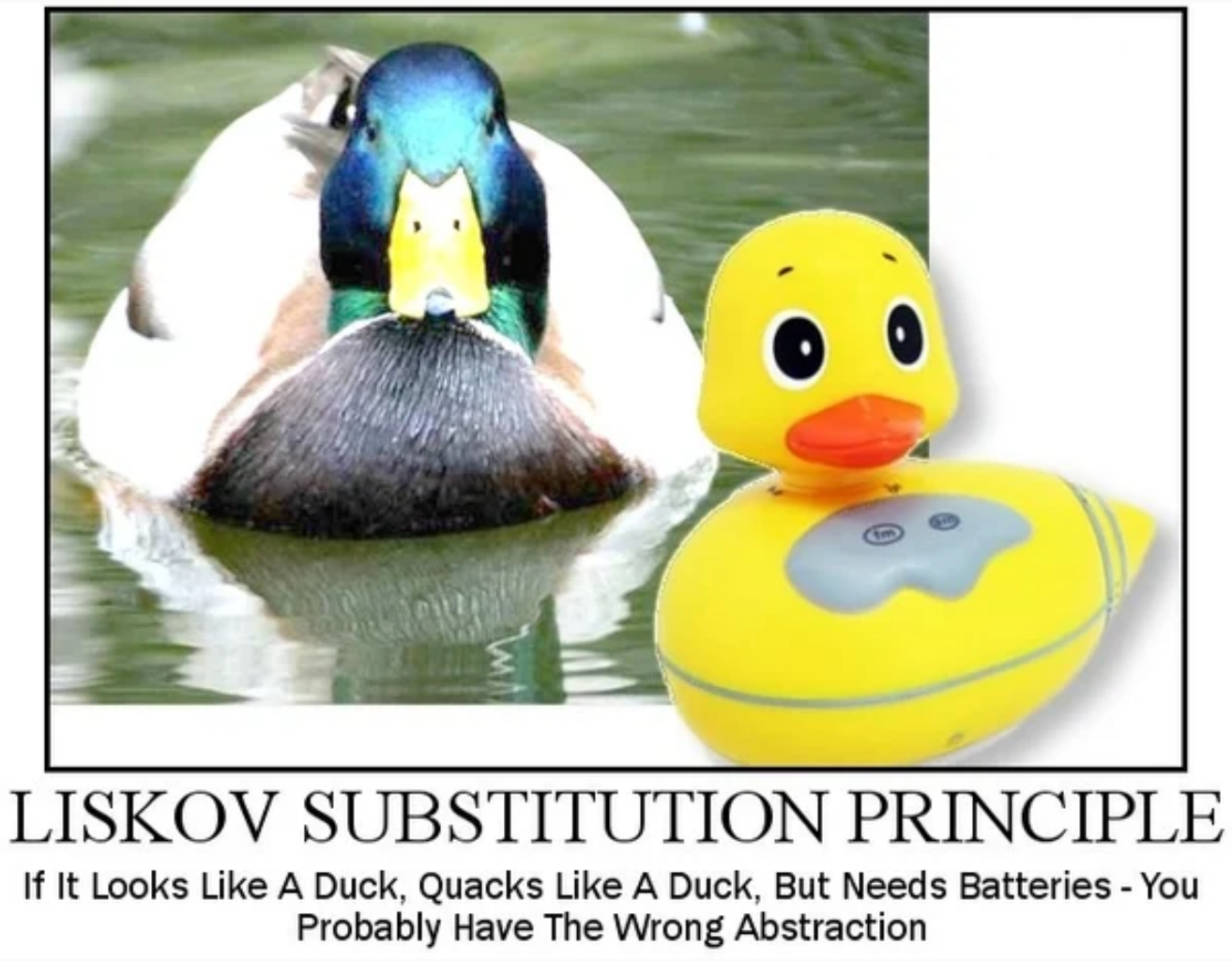
Принцип подстановки Барбары Лисков:

> *Функции, которые используют базовый тип, должны иметь возможность использовать подтипы базового типа, не зная об этом.*

В другой интерпретации (Г. Саттер и А. Александреску):

> *Подкласс не должен требовать от вызывающего кода больше, чем базовый класс, и не должен предоставлять вызывающему коду меньше, чем базовый класс.*

Liskov Substitution Principle



Liskov Substitution Principle: Проблема квадрата-прямоугольника (1)

Необходимо спроектировать классы Квадрат и Прямоугольник. У данных объектов должна быть возможность вычислять площадь и изменять длины сторон.

Т.к. классы похожи, попробуем реализовать их при помощи наследования.

Какой класс должен быть родительским, а какой — дочерним?

Liskov Substitution Principle: Проблема квадрата-прямоугольника (2)

Попробуем сделать Прямоугольник подклассом Квадрата:

```
1  public class Square {
2      private int size;
3      public Square(int size) { this.size = size; }
4      public int area() { return size * size; }
5      public int getSize() { return size; }
6      public void setSize(int size) { this.size = size; }
7  }
8  public class Rectangle extends Square {
9      private int height;
10     public Rectangle(int width, int height) {
11         super(width);
12         this.height = height;
13     }
14     @Override
15     public int area() { return getSize() * height; }
16     public void getHeight() { return height; }
17     public void setHeight(int height) { this.height = height; }
18 }
```

Liskov Substitution Principle: Проблема квадрата-прямоугольника (3)

Проблема: методы, взаимодействующие с Квадратом, ведут себя странно при передаче в них Прямоугольника.

```
1 public static int setSizeTo10AndGetArea(Square square) {
2     square.setSize(10);
3     return square.area(); // Этот метод вернёт 100, так ведь?
4 }
```

Liskov Substitution Principle: Проблема квадрата-прямоугольника (3)

Проблема: методы, взаимодействующие с Квадратом, ведут себя странно при передаче в них Прямоугольника.

```
1 public static int setSizeTo10AndGetArea(Square square) {
2     square.setSize(10);
3     return square.area();
4 }
5 public static void main(String[] args){
6     Rectangle rectangle = new Rectangle(5, 3);
7     int result = setSizeTo10AndGetArea(rectangle); // Вернёт 30
8     rectangle.getSize(); // Что такое size?
9 }
```

Liskov Substitution Principle: Проблема квадрата-прямоугольника (4)

Попробуем сделать Квадрат подклассом Прямоугольника:

```
1  public class Rectangle {
2      private int width, height;
3      public Rectangle(int width, int height) {
4          this.width = width;
5          this.height = height;
6      }
7      public int area() { return width * height; }
8      public void setSizes(int width, int height) {
9          this.width = width;
10         this.height = height;
11     }
12 }
13
14 public class Square extends Rectangle {
15     public Square(int size) {
16         super(size, size);
17     }
18 }
```

Liskov Substitution Principle: Проблема квадрата-прямоугольника (5)

Проблема: Квадрат унаследовал метод, который не имеет для него смысла.

```
1 public static void main(String[] args){
2     Square square = new Square(5);
3     square.setSizes(10, 8); // Что мы получим в результате этого вызова?
4 }
```

Liskov Substitution Principle: Проблема квадрата-прямоугольника (6)

Правильным подходом будет задать себе вопрос: в чём цель классификации и наследования?

У квадрата и прямоугольника лишь одна общая операция: вычисление площади. Поэтому более правильная абстракция — реализация общего интерфейса Фигура.

```
1  public interface Shape {
2      int area();
3  }
4  public class Rectangle implements Shape {
5      @Override
6      public int area() { return width * height; }
7  }
8  public class Square implements Shape {
9      @Override
10     public int area() { return size * size; }
11 }
```

Liskov Substitution Principle: Проблема квадрата-прямоугольника (7)

Тем не менее, существует ещё одна проблема (уже не в плоскости LSP):

```
1 Rectangle rectangle = new Rectangle(5, 5);
2 Square square = new Square(5);
3 // Чем rectangle отличается от square?
```

Liskov Substitution Principle: Композиция вместо наследования (1)

Т.к. удовлетворить требования LSP довольно сложно, вместо наследования рекомендуется использовать композицию.

При наследовании: класс `B` расширяет класс `A`, в итоге у класса `B` такой же интерфейс (т.е. набор публичных методов).

При композиции: классы `A` и `B` реализуют общий интерфейс `I`; при этом класс `B` хранит объект класса `A` и реализует при помощи него нужные методы.

Рекомендуется к прочтению: **J. Bloch. Effective Java. Favor composition over inheritance**.

Liskov Substitution Principle: Композиция вместо наследования (2)

```java
public class Rectangle implements Shape {
    private int width, height;
    public Rectangle(int width, int height) {
        this.width = width;
        this.height = height;
    }
    public int area() { return width * height; }
    public void setSizes(int width, int height) { ... }
}

public class Square implements Shape {
    private final Rectangle rectangle;
    public Square(int size) {
        this.rectangle = new Rectangle(size, size);
    }
    public int area() { return rectangle.area(); }
    public void setSize(int size) {
        rectangle.setSizes(size, size);
    }
}
```

Interface Segregation Principle

Принцип разделения интерфейса:

> *Программные сущности не должны зависеть от методов, которые они не используют.*

В другой формулировке:

> *Интерфейсы формируются исходя из потребностей пользовательского кода.*

Interface Segregation Principle: Пример (1)

```
1 public interface CoffeeMachine {
2     void turnOn();
3     void turnOff();
4     Coffee brewCoffee();
5 }
```

```
1 public void restart(CoffeeMachine coffeeMachine) {
2     coffeeMachine.turnOff();
3     coffeeMachine.turnOn();
4 }
```

Interface Segregation Principle: Пример (2)

```
1  public void restart(CoffeeMachine coffeeMachine) {
2      coffeeMachine.turnOff();
3      coffeeMachine.turnOn();
4  }
```

Метод `restart` довольно полезный, поэтому хотелось бы использовать его и для класса PC.

```
1  PC pc = new PC();
2  restart(pc);
3
4  public class PC implements CoffeeMachine {
5      public void turnOn() { ... }
6      public void turnOff() { ... }
7      public Coffee brewCoffee() {
8          // ???
9          // Как реализовать этот метод?
10     }
11 }
```

Interface Segregation Principle: Пример (3)

Метод `restart` не использует все возможности интерфейса `CoffeeMachine`, поэтому правильнее выделить необходимый набор методов в отдельный интерфейс.

```
1 public interface Switchable {
2     void turnOn();
3     void turnOff();
4 }
5
6 public void restart(Switchable switchable) {
7     switchable.turnOff();
8     switchable.turnOn();
9 }
```

При таком подходе мы сначала думаем о том, **как** будут использовать интерфейс его пользователи (метод `restart`), а только потом создаём реализации этого интерфейса (`CoffeeMachine` и `PC`).

Interface Segregation Principle: Пример (4)

Интерфейс `CoffeeMachine` и класс `PC` могут расширить/реализовать нужный интерфейс.

```
1 public interface CoffeeMachine extends Switchable {
2     Coffee brewCoffee();
3 }
4
5 public class PC implements Switchable {
6     public void turnOn() { ... }
7     public void turnOff() { ... }
8 }
```

Dependency Inversion Principle

Принцип инверсии зависимостей:

1. *Модули верхних уровней не должны зависеть от модулей нижних уровней. Оба типа модулей должны зависеть от абстракций.*
2. *Абстракции не должны зависеть от деталей. Детали должны зависеть от абстракций.*

Dependency Inversion Principle: Пример (1)

Возьмём следующий набор типов:

```
1 public interface IntStack {
2     void push(int value);
3     int pop();
4 }
5 public class DynamicArrayIntStack implements IntStack { ... }
```

В чём проблема кода ниже?

```
1 public class PostfixCalculator {
2     private final DynamicArrayIntStack stack =
3             new DynamicArrayIntStack();
4 }
```

Dependency Inversion Principle: Пример (2)

Для калькулятора не важна реализация стека. Важен лишь его интерфейс.

```
1 public class PostfixCalculator {
2     private final IntStack stack;
3     public PostfixCalculator(IntStack stack) {
4         this.stack = stack;
5     }
6 }
```

Реализации могут свободно меняться (без переписывания калькулятора!):

```
1 new PostfixCalculator(new DynamicArrayIntStack());
2 new PostfixCalculator(new LinkedListIntStack());
```

Важно о всех принципах проектирования

Разработка ПО находится на стыке формального и неформального.

Поэтому нельзя строго сказать, **как правильно** писать программы, а как — нет.

Принципы проектирования по типу SOLID — это не священные догмы!

Данные принципы нужно **использовать разумно**, заранее обдумав все плюсы и минусы, а не просто слепо следовать.

Материалы для изучения

Книги:

- Стив Макконнелл. Совершенный код.
- ~~Роберт Мартин. Чистый код.~~

Статьи:

- Что такое анти-паттерны?
- Single Responsibility Principle. Не такой простой, как кажется
- SOLID Design in C#: The Liskov Substitution Principle (LSP)
- Разбираемся с SOLID: Инверсия зависимостей
- Актуальность принципов SOLID