Принципы объектно-ориентированной разработки SOLID

В главе 11 «Избегание зависимостей и тесно связанных классов» мы обсуждали некоторые фундаментальные концепции, постепенно подбираясь к обсуждению пяти принципов SOLID. В этой главе мы подробно рассмотрим каждый принцип SOLID. Все характеристики принципов SOLID взяты с сайта Дяди Боба: http://butunclebob.com/ArticleS.UncleBob.PrinciplesOfOod.

1. SRP: принцип единственной ответственности

Принцип единственной ответственности гласит о том, для внесения изменений в класс требуется только одна причина. Каждый класс и модуль программы должны иметь в приоритете одно задание. Поэтому не стоит вносить методы, которые могут вызвать изменения в классе более чем по одной причине. Если описание класса содержит слово «and», то принцип SRP может быть нарушен. Другими словами, каждый модуль или класс должен нести ответственность за одну какую-либо часть функционала программного обеспечения, и такая ответственность должна быть полностью инкапсулирована в класс.

Создание иерархии фигур — это один из классических примеров, иллюстрирующих наследование Этот пример часто встречается в обучении, а я использую его на протяжении этой главы (равно как и всей книги). В этом примере класс `Circle` наследует атрибуты от класса `Shape`. Класс `Shape` предоставляет абстрактный метод `calcArea()` в качестве контракта для подкласса. Каждый класс, наследующий от `Shape`, должен иметь собственную реализацию метода `calcArea()`:

```
abstract class Shape{
    protected String name;
    protected double area;
    public abstract double calcArea();
}
```

В этом примере класс `Circle`, наследующий от класса `Shape`, при необходимости обеспечивает свою реализацию метода `calcArea()`:

```
class Circle extends Shape{
    private double radius;

    public Circle(double r) {
        radius = r;
    }
    public double calcArea() {
        area = 3.14*(radius*radius) ;
        return (area);
    };
}
```

ПРЕДОСТЕРЕЖЕНИЕ

В этом примере мы только собираемся рассмотреть класс `Circle`, чтобы сосредоточиться на принципе единственной ответственности и сделать пример максимально простым.

Третий класс, `CalculateAreas`, подсчитывает площади различных фигур, содержащихся в массиве `Shape`. Массив `Shape` обладает неограниченным размером и может содержать различные фигуры, например квадраты и треугольники.

```
class CalculateAreas {
    Shape[] shapes;
    double sumTotal=0;
    public CalculateAreas(Shape[] sh) {
        this.shapes = sh;
    }
    public double sumAreas() {
        sumTotal=0;
        for (inti=0; i<shapes.length; i++) {
        sumTotal = sumTotal + shapes[i].calcArea() ;
        }
        return sumTotal ;
    }
    public void output() {
        System.out.printIn("Total of all areas = " + sumTotal);
    }
}
```

Обратите внимание, что класс `CalculateAreas` также обрабатывает вывод приложения, что может вызвать проблемы. Поведение подсчета площади и поведение вывода связаны, поскольку содержатся в одном и том же классе.

Мы можем проверить работоспособность этого кода с помощью соответствующего тестового приложения `TestShape`:

```
public class TestShape {
    public static void main(String args[]) {

        System.out.printin("Hello World!");

        Circle circle = new Circle(1);

        Shape[] shapeArray = new Shape[1];
        shapeArray[0] = circle;

        CalculateAreas ca = new CalculateAreas(shapeArray) ;

        ca.sumAreas() ;
        ca.output();
    }
}
```

Теперь, имея в распоряжении тестовое приложение, мы можем сосредоточиться на проблеме принципа единственной ответственности. Опять же, проблема связана с классом `CalculateAreas` и с тем, что этот класс содержит поведения и для *сложения* площадей различных фигур, а также для *вывода* данных.

Основополагающий вопрос (и, собственно, проблема) в том, что если нужно изменить функциональность метода `output()`, потребуется внести изменения в класс `CalculateAreas` независимо от того, изменится ли метод подсчета площади фигур. Например, если мы вдруг захотим осуществить вывод данных в HTML-консоль, а не в простой текст, нам потребуется заново компилировать и повторно внедрять код, который складывает площади фигур. Все потому, что ответственности связаны.

В соответствии с принципом единственной ответственности, задача состоит в том, чтобы изменение одного метода не повлияло на остальные методы и не приходилось проводить повторную компиляцию. «У класса должна быть одна, только одна, причина для изменения — единственная ответственность, которую нужно изменить».

Чтобы решить данный вопрос, можно поместить два метода в отдельные классы, один для оригинального консольного вывода, другой для вывода в HTML:

```
class CaiculateAreas {;
    Shape[] shapes;
    double sumTotal=0;

    public CalculateAreas(Shape[] sh) {
        this.shapes = sh;
    }

    public double sumAreas() {
        sumTotal=0;

        for (inti=0; i<shapes.length; i++) {

            sumTotal = sumTotal + shapes[i].calcArea();

        }

            return sumTotal;
        }
}
class OutputAreas {
    double areas=0;
    public OutputAreas (double a) {
        this.areas = a;
    }

    public void console() {
```

```
            System.out.printin("Total of all areas = " + areas);
        }
        public void HTML() {
            System.out.printIn("<HTML>") ;
            System.out.printin("Total of all areas = " + areas);
            System.out.printin("</HTML>") ;
        }
    }
```

Теперь с помощью недавно написанного класса мы можем добавить функциональность для вывода в HTML без воздействия на код для вычисления площади:

```
    public class TestShape {
        public static void main(String args[]) {

            System.out.printin("Hello World!");

            Circle circle = new Circle(1);

            Shape[] shapeArray = new Shape[1];
            shapeArray[0] = circle;

            CalculateAreas ca = new CalculateAreas(shapeArray) ;

            CalculateAreas sum = new CalculateAreas(shapeArray) ;
            OutputAreasoAreas = new OutputAreas(sum.sumAreas() ) ;

            oAreas.console(); // output to console
            oAreas.HTML() ; // output to HTML
        }
    }
```

Суть здесь заключается в том, что теперь можно послать вывод в различных направлениях в зависимости от необходимости. Если нужно добавить возможность другого способа вывода, например JSON, можно привнести ее в класс `OutputAreas` без необходимости внесения изменений в класс `CalculateAreas`. В результате можно перераспределить класс `CalculateAreas` без какого-либо затрагивания других классов.

2. OCP: принцип открытости/закрытости

Принцип открытости/закрытости гласит, что можно расширить поведение класса без внесения изменений.

Обратим снова внимание на пример с фигурами. В приведенном ниже коде есть класс `ShapeCalculator`, который берет объект `Rectangle`, рассчитывает площадь этого объекта и возвращает значения. Это простое приложение, но оно работает только с прямоугольниками.

```
class Rectangle{
    protected double length;
    protected double width;

    public Rectangle(double 1, double w) {
        length = 1;
        width = w;
    };
}
class CalculateAreas {

    private double area;

    public double calcArea(Rectangle r) {

        area = r.length * r.width;

        return area;

    }
}
public class OpenClosed {
    public static void main(String args[]) {

        System.out.printin("Hello World");

        Rectangle r = new Rectangle(1,2);

        CalculateAreas ca = new CalculateAreas ();

        System.out.printin("Area = "+ ca.calcArea(r));

    }
}
```

То, что это приложение работает только в случае с прямоугольниками, приводит к ограничению, которое наглядно объясняет принцип открытости/закрытости: если мы хотим добавить класс `Circle` к классу `CalculateArea` (изменить то, что он выполняет), нам нужно внести изменения в сам модуль. Очевидно, что это вступает в противоречие с принципом открытости/закрытости, который гласит, что мы не должны вносить изменения в модуль для изменения того, что он выполняет.

Чтобы соответствовать принципу открытости/закрытости, можно вернуться к уже проверенному примеру с фигурами, где создается абстрактный класс `Shape` а непосредственно фигуры наследуют от класса `Shape`, у которого есть абстрактный метод `getArea()`.

На данный момент можно добавлять столь много разных классов, сколько требуется, без необходимости внесения изменений непосредственно в класс `Shape` (например, класс `Circle`). Сейчас можно сказать, что класс `Shape` закрыт.