

ОГЛАВЛЕНИЕ

Благодарности	14
Об авторе	15
Авторские права на иллюстрации	15
От издательства	15
Введение	16
Что нового в пятом издании	18
Целевая аудитория	19
Подход, примененный в этой книге	20
Глава 1. Введение в объектно-ориентированные концепции	21
Фундаментальные концепции	22
Объекты и унаследованные системы	23
Процедурное программирование в сравнении с объектно-ориентированным	24
Переход с процедурной разработки на объектно-ориентированную	29
Процедурное программирование	29
Объектно-ориентированное программирование	29
Что такое объект?	30
Данные объектов	30
Поведение объектов	31
Что такое класс?	35
Создание объектов	35

Атрибуты	37
Методы	37
Сообщения	38
Использование диаграмм классов в качестве визуального средства.....	38
Инкапсуляция и сокрытие данных.....	39
Интерфейсы.....	39
Реализации	40
Реальный пример парадигмы «интерфейс/реализация»	41
Модель парадигмы «интерфейс/реализация»	41
Наследование	43
Суперклассы и подклассы	44
Абстрагирование	45
Отношения «является экземпляром».....	46
Полиморфизм	47
Композиция	51
Абстрагирование	51
Отношения «содержит как часть»	52
Резюме	52
Глава 2. Как мыслить объектно	53
Разница между интерфейсом и реализацией	54
Интерфейс	56
Реализация	57
Пример интерфейса/реализации	57
Использование абстрактного мышления при проектировании классов	63
Обеспечение минимального интерфейса пользователя	65
Определение пользователей.....	66
Поведения объектов	67
Ограничения, налагаемые средой.....	67
Определение открытых интерфейсов.....	68
Определение реализации	69
Резюме	70
Ссылки	70

Глава 3. Прочие объектно-ориентированные концепции	71
Конструкторы	71
Когда осуществляется вызов конструктора?	72
Что находится внутри конструктора?	72
Конструктор по умолчанию	73
Использование множественных конструкторов	74
Перегрузка методов	75
Использование UML для моделирования классов	76
Как сконструирован суперкласс?	78
Проектирование конструкторов	79
Обработка ошибок	79
Игнорирование проблем	80
Проверка на предмет проблем и прерывание выполнения приложения	80
Проверка на предмет проблем и попытка устранить неполадки	80
Выбрасывание исключений	81
Важность области видимости	84
Локальные атрибуты	84
Атрибуты объектов	86
Атрибуты классов	88
Перегрузка операторов	89
Множественное наследование	90
Операции с объектами	91
Резюме	93
Ссылки	93
Глава 4. Анатомия класса	94
Имя класса	94
Комментарии	95
Атрибуты	97
Конструкторы	98
Методы доступа	101
Методы открытых интерфейсов	103
Методы закрытых реализаций	104
Резюме	105
Ссылки	105

Глава 5. Руководство по проектированию классов	106
Моделирование реальных систем	106
Определение открытых интерфейсов	108
Минимальный открытый интерфейс	108
Скрытие реализации	109
Проектирование надежных конструкторов (и, возможно, деструкторов)	110
Внедрение обработки ошибок в класс	111
Документирование класса и использование комментариев	111
Создание объектов с прицелом на взаимодействие	112
Проектирование с учетом повторного использования	113
Проектирование с учетом расширяемости	113
Делаем имена описательными	114
Абстрагирование непереносимого кода	115
Обеспечение возможности осуществлять копирование и сравнение	116
Сведение области видимости к минимуму	116
Проектирование с учетом сопровождаемости	117
Использование итерации в процессе разработки	118
Тестирование интерфейса	118
Использование постоянства объектов	120
Сериализация и маршалинг объектов	121
Резюме	122
Ссылки	122
Глава 6. Проектирование с использованием объектов	123
Руководство по проектированию	123
Проведение соответствующего анализа	128
Составление технического задания	128
Сбор требований	129
Разработка прототипа интерфейса пользователя	129
Определение классов	129
Определение ответственности каждого класса	130
Определение взаимодействия классов друг с другом	130
Создание модели классов для описания системы	130
Прототипирование интерфейса пользователя	130

Объектные обертки	131
Структурированный код	132
Обертывание структурированного кода	133
Обертывание непереносимого кода.....	135
Обертывание существующих классов	136
Резюме	137
Ссылки	138
Глава 7. Наследование и композиция	139
Повторное использование объектов	139
Наследование	141
Обобщение и конкретизация.....	145
Проектные решения	146
Композиция	148
Почему инкапсуляция является фундаментальной объектно-ориентированной концепцией	151
Как наследование ослабляет инкапсуляцию.....	151
Подробный пример полиморфизма	154
Ответственность объектов	154
Абстрактные классы, виртуальные методы и протоколы.....	158
Резюме	160
Ссылки	160
Глава 8. Фреймворки и повторное использование: проектирование с применением интерфейсов и абстрактных классов	162
Код: использовать повторно или нет?.....	162
Что такое фреймворк?.....	163
Что такое контракт?.....	166
Абстрактные классы.....	166
Интерфейсы.....	170
Связываем все воедино	172
Код, выдерживающий проверку компилятором.....	175
Заключение контракта	176
Системные «точки расширения».....	179

Пример из сферы электронного бизнеса	179
Проблема, касающаяся электронного бизнеса	179
Подход без повторного использования кода	180
Решение для электронного бизнеса	183
Объектная модель UML	183
Резюме	188
Ссылки	188
Глава 9. Создание объектов и объектно-ориентированное проектирование ...	189
Отношения композиции	190
Поэтапное создание	191
Типы композиции	194
Агрегации	194
Ассоциации	195
Использование ассоциаций в сочетании с агрегациями	196
Избегание зависимостей	197
Кардинальность	198
Ассоциации, включающие множественные объекты	200
Необязательные ассоциации	202
Связываем все воедино: пример	203
Резюме	204
Ссылки	204
Глава 10. Паттерны проектирования	205
Чем хороши паттерны проектирования?	206
Схема «Модель — Представление — Контроллер» в языке Smalltalk	207
Типы паттернов проектирования	209
Порождающие паттерны	210
Структурные паттерны	215
Паттерны поведения	218
Антипаттерны	219
Заключение	221
Ссылки	221

Глава 11. Избегание зависимостей и тесно связанных классов	222
Композиция против наследования и внедрения зависимостей	225
1. Наследование	225
2. Композиция.....	227
Внедрение зависимостей	230
Внедрение с помощью конструктора.....	232
Заключение.....	233
Ссылки	233
Глава 12. Принципы объектно-ориентированного проектирования SOLID	234
Принципы объектно-ориентированной разработки SOLID.....	236
1. SRP: принцип единственной ответственности	236
2. OCP: принцип открытости/закрытости	239
3. LSP: принцип подстановки Лисков.....	242
4. ISP: принцип разделения интерфейса	245
5. DIP: принцип инверсии зависимостей	246
Заключение.....	253
Ссылки	253
Об обложке	254

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.println("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.println("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.

Основопологающий 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 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;  
    }  
}  
  
class OutputAreas {  
    double areas=0;  
    public OutputAreas (double a) {  
        this.areas = a;  
    }  
  
    public void console() {
```

```

        System.out.println("Total of all areas = " + areas);
    }
    public void HTML() {
        System.out.println("<HTML>");
        System.out.println("Total of all areas = " + areas);
        System.out.println("</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.println("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) ;
        OutputAreas oAreas = 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 привести ее в класс `OutputAreas` без необходимости внесения изменений в класс `CalculateAreas`. В результате можно перераспределить класс `CalculateAreas` без какого-либо затрагивания других классов.

2. ОСР: 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 l, double w) {
        length = l;
        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.println("Hello World");

        Rectangle r = new Rectangle(1,2);

        CalculateAreas ca = new CalculateAreas ();

        System.out.println("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.