нетология

Свойства хорошего кода. SOLID

Филипп
Воронов

Филипп Воронов

Teamlead, Поиск VK

План занятия

1. Магические числа
2. Don’t Repeat Yourself
3. SOLID
4. Single-Responsibility principle
5. Open-Closed principle
6. Liskov substitution principle
7. Interface segregation principle
8. Dependency inversion principle
9. Итоги

Магические числа

Магические числа

У программы может быть множество количественных параметров: количество товаров в магазине, размер команды в онлайн-игре, минимальная плата за обслуживание. Все эти значения мы так или иначе используем в нашем коде.

Правило Magic: не используй числа напрямую в коде

Не используйте значение параметра в коде напрямую, заведите для этого отдельную константу и обращайтесь к ней по имени. Можете также вместо константы обратиться к другому осмысленному выражению в вашей программы. Это сделает ваш код *понятнее* для чтения, *адекватнее* по содержанию и поможет *избежать частых ошибок* при её редактировании. Числа не должны появляться в коде “магически”!

Магические числа

```
class Main {
  static void main(String[] args) {
    String[] products = {"Молоко", "Хлеб", "Горчица"};
    int[] prices = {100, 50, 40};

    int[] basket = new int[3];
    Scanner scanner = new Scanner(System.in);
    while (scanner.hasNextInt()) {
      int productNumber = scanner.nextInt();
      int productCount = scanner.nextInt();
      basket[productNumber - 1] += productCount;
    }

    for (int i = 0; i < 3; i++) {
      if (basket[i] > 0)
        System.out.println("Вы купили " + products[i] +
            " за " + (basket[i] * prices[i])
        );
    }
  }
}
```



```
class Main {
  static void main(String[] args) {
    String[] products = {"Молоко", "Хлеб", "Горчица"};
    int[] prices = {100, 50, 40};

    int[] basket = new int[products.length];
    Scanner scanner = new Scanner(System.in);
    while (scanner.hasNextInt()) {
      int productNumber = scanner.nextInt();
      int productCount = scanner.nextInt();
      basket[productNumber - 1] += productCount;
    }

    for (int i = 0; i < basket.length; i++) {
      if (basket[i] > 0)
        System.out.println("Вы купили " + products[i] +
            " за " + (basket[i] * prices[i])
        );
    }
  }
}
```

Теперь добавление новых товаров в массивы products и prices не приведут к ошибке, тк не требуют исправлений во всех местах числа 3 на 4 (и во всех ли? в каждом месте нам надо было бы ещё призадуматься, означает ли число 3 тут количество товаров или что-нибудь ещё!).

Don’t Repeat Yourself

Don’t Repeat Yourself

Часто наша программа в разных местах делает либо одни и те же вещи, либо очень похожие вещи. Особенно это актуально, когда мы пишем новый кусок кода через копирование и вставки другой части нашего же кода, возможно с несколькими поправками.

Правило DRY (Don’t Repeat Yourself): не повторяй свой код

Видите повторяющиеся или похожие вещи? Возможно, пришло время создать новый метод! Вынесите туда повторяющийся кусок кода. Если код повторяется не с точностью, то стоит подумать о создании параметров к вашему методу, которые заключали бы в себя эту вариативность.

Don’t Repeat Yourself

Теперь повторяющаяся логика вывода матрицы на экран вынесена в отдельный метод - код выглядит компактней, понятнее и в любой момент когда мы захотим вывести матрицу нам достаточно будет лишь вызвать метод, а не копировать кусок кода. Изменение логики вывода на экран теперь тоже удобнее: изменить придётся лишь в одном месте - метод - а не в каждом месте использовании как было до.

```
class Main {
  static void main(String[] args) {
    final Random RANDOM = new Random();
    final int SIZE = 5;

    int[][] matrix = new int[SIZE][SIZE];
    for (int i = 0; i < matrix.length; i++)
      for (int j = 0; j < matrix[i].length; j++)
        matrix[i][j] = RANDOM.nextInt(10);

    System.out.println("_________________");
    for (int i = 0; i < matrix.length; i++) {
      for (int j = 0; j < matrix[i].length; j++) {
        System.out.printf("%3d", matrix[i][j]);
      }
      System.out.println();
    }
    System.out.println("^^^^^^^^^^^^^^^^");

    int[][] flipped = new int[SIZE][SIZE];
    for (int i = 0; i < matrix.length; i++)
      for (int j = 0; j < matrix[i].length; j++)
        flipped[i][j] = matrix[j][i];

    System.out.println("_________________");
    for (int i = 0; i < flipped.length; i++) {
      for (int j = 0; j < flipped[i].length; j++) {
        System.out.printf("%3d", flipped[i][j]);
      }
      System.out.println();
    }
    System.out.println("^^^^^^^^^^^^^^^^");
  }
}
```

```
class Main {
  static void printMatrix(int[][] matrix) {
    System.out.println("_________________");
    for (int i = 0; i < matrix.length; i++) {
      for (int j = 0; j < matrix[i].length; j++) {
        System.out.printf("%3d", matrix[i][j]);
      }
      System.out.println();
    }
    System.out.println("^^^^^^^^^^^^^^^^");
  }

  public static void main(String[] args) {
    final Random RANDOM = new Random();
    final int SIZE = 5;

    int[][] matrix = new int[SIZE][SIZE];
    for (int i = 0; i < matrix.length; i++)
      for (int j = 0; j < matrix[i].length; j++)
        matrix[i][j] = RANDOM.nextInt(10);

    printMatrix(matrix);

    int[][] flipped = new int[SIZE][SIZE];
    for (int i = 0; i < matrix.length; i++)
      for (int j = 0; j < matrix[i].length; j++)
        flipped[i][j] = matrix[j][i];

    printMatrix(flipped);
  }
}
```

SOLID

SOLID

SOLID - это аббревиатура, которая обозначает пять принципов ООП и проектирования.

S - принцип единственной ответственности (Single Responsibility Principle)
Класс должен выполнять только те функции, для которых он логически предназначен.

O - принцип открытости/закрытости (Open Closed Principle)
Программные сущности должны быть открыты для расширения, но закрыты для модификации.

L - принцип замены Барбары Лисков (Liskov Substitution Principle)
Наследуй только тогда, когда можешь играть роль за предка.

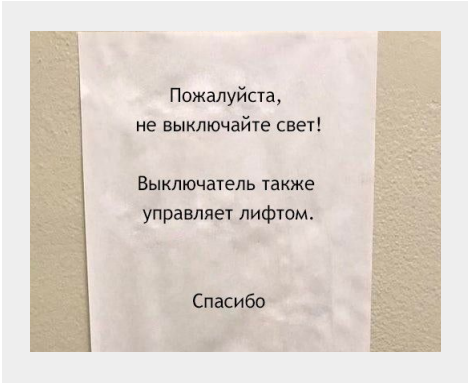
I - принцип сегрегации (разделения) интерфейса (Interface Segregation Principle)
Много интерфейсов, специально предназначенных для клиентов, лучше, чем один интерфейс общего назначения.

D - принцип инверсии зависимостей (Dependency Inversion Principle)
Всё зависит от абстракций (интерфейсов), а не от деталей реализации друг друга.

Single-Responsibility principle

Принцип единственной ответственности

Single-responsibility principle



Мы уже знаем, что не обязаны писать весь код нашей программы в одном файле - мы можем создавать сколько угодно разных классов, каждый в своём собственном файле. Это здорово, мы же не хотим хранить тысячи строк нашей программы в одном гигантском файле. Но это ли единственная мотивация и принцип разделения программы на классы?

Принцип единственной ответственности - каждый делает только то, для чего он предназначен

Мы разделяем программу на пакеты, классы и прочие блоки не только для сокращения объёмов отдельных файлов, но для логического разделения. Например, класс Bird описывает то, что умеет птица, но не описывает то, на какой этаж едет человек, чтобы взять из дома хлеба ей покрошить.
Класс должен выполнять только те функции, для которых он логически был создан, и все относящиеся логически к нему функции должны находиться в нём.

Single-responsibility principle

```
class Calculator {
    int calculate(String formula) {
        //...
    }

    void sendResultByEmail(int result, String
email) {
        //...
    }
}
```



```
class Calculator {
    int calculate(String formula) {
        //...
    }
}
```

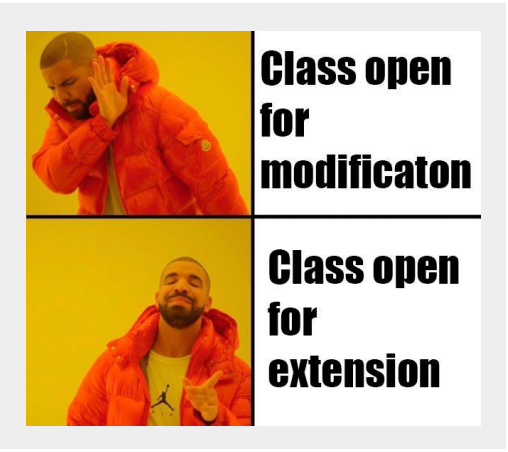
```
class EmailNotifier {
    void sendResultByEmail(int result, String
email) {
        //...
    }
}
```

Задача калькулятора считать математические формулы, выдавая в качестве результата число, но уж точно не отправлять этот результат по почте - эту логику необходимо вынести из класса калькулятора в отдельный класс, ведь логически это разные задачи.

Open-Closed principle

Принцип открытости / закрытости

Open-closed principle



С помощью наследования мы можем научить новые классы всему тому, что умел исходный, но также добавить новые поля и методы - и всё это не меняя исходный класс. Например, нет необходимости мерчендайзера учить выставлять йогурты отдельно для каждого конкретного производителя, можно научить выставлять йогурты *в принципе*, тогда нам не нужно будет его переучивать если ассортимент производителей расширится.

Принцип открытости/закрытости - программные сущности должны быть открыты для расширения, но закрыты для модификации

Сущности (например, классы) должны быть открыты для расширения и закрыты для изменения. Мы должны *стремиться* писать код так, чтобы добавить новую функциональность было легко (например, наследованием), *не меняя* исходные классы.

Open-closed principle

Теперь добавление нового типа фигур с площадью не требует изменений в классе подсчёта суммы площадей: мы *можем расширять* функциональность сумматора *не изменяя его* код, а добавляя новые классы реализаций.

```
class Square {
   double a;
   Square(int a) { this.a = a; }
   double area() { return a * a; }
}

class AreaSummator {
   double area = 0;
   void add(Square s) {
      area += s.area();
   }
   double sum() { return area; }
}
```

Добавляем Circle

```
class Square { ... }

class Circle {
   double r;
   Circle(int r) { this.r = r; }
   double area() {
      return Math.PI * r * r;
   }
}

public static class AreaSummator {
   Придётся менять для поддержки Circle
}
```

```
interface Areable {
   double area();
}

class Square implements Areable { ... }

class AreaSummator {
   double area = 0;
   void add(Areable s) { area += s.area(); }
   double sum() { return area; }
}
```

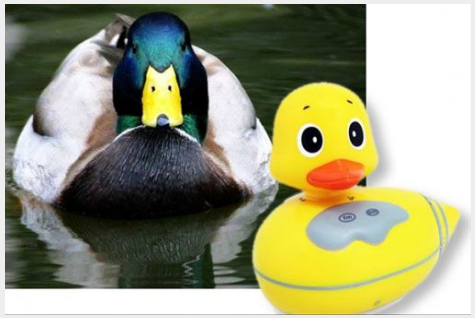
Добавляем Circle

```
class Main {

interface Areable {
  double area();
}

class Square implements Areable { … }
class Circle implements Areable { ... }

class AreaSummator {
   Не требует изменений!
}
```

Liskov substitution principle

Принцип замены Барбары Лисков

Liskov substitution principle



Наследование в ООП - это не просто “технический хак” копирования полей и методов, это целая философия. *Наследники расширяют и/или модифицируют возможности классов своих предков*, наследники могут заменить своих предков в местах, где требуется предок.

Принцип замены Барбары Лисков - наследуй только тогда, когда можешь играть роль за предка

Наследники должны *логически* иметь возможность сыграть роль своих предков там, где требуются объекты предков - ведь они наследники! Поэтому если игрушка похожа на утку, крякает как утка, но нуждается в батарейках - возможно вам надо дважды подумать прежде чем считать её наследником утки, птицы и животного вообще.

Liskov substitution principle

```
class Logger {
    void log(int value) {
        System.out.println("Logging value to console: "
                + value);
    }
}

class Calculator extends Logger {
    void sum(int a, int b) {
        System.out.println("+: " + (a + b));
    }

    @Override
    void log(int value) {
        System.out.println("log: " + Math.log(value));
    }
}
```



```
class Logger {
    void log(int value) {
        System.out.println("Logging value to console: "
                + value);
    }
}

class Calculator {
    void sum(int a, int b) {
        System.out.println("+: " + (a + b));
    }

    @Override
    void log(int value) {
        System.out.println("log: " + Math.log(value));
    }
}
```

У обоих классов есть метод `log(int value)`, но в случае калькулятора это подсчёт логарифма переданного значения с выводом результата на экран, а в случае логгера это логирование числа в консоль - совпадение сигнатур методов чисто случайны, логически они никак не связаны, логгер не является расширением понятия кулькулятор, как и калькулятор не является расширением понятия логгера, поэтому они не должны быть связаны отношением наследования.

Interface segregation principle

Принцип сегрегации (разделения) интерфейса

Interface segregation principle

Представим себе, что мы купили зарядку, которая подходит для разных моделей телефона, ноутбука и так далее. Будет ли правильно если мы не сможем зарядить наш андроид-телефон если контакт зарядки ноутбука поломался?

Принцип сегрегации (разделения) интерфейса - разделяй большие интерфейсы на маленькие

Много интерфейсов, специально предназначенных для клиентов, лучше, чем один интерфейс общего назначения. Если у вас есть сложный объект, умеющий много разных функций, то будет неправильно заключить всех их в один большой интерфейс: клиент не должен зависеть от функций, которых он не использует, поэтому лучше создать набор различных логически разделённых интерфейсов вместо одного большого.

Interface segregation principle

```
interface Device {
    void setAlarm(String time);
    void sendSMS(String msg, String number);
    void playMusic(String query);
}

class Mobile implements Device {
    ...
}
```



```
interface Clock {
    void setAlarm(String time);
}
interface SMSer {
    void sendSMS(String msg, String number);
}
interface MusicPlayer {
    void playMusic(String query);
}

class Mobile implements Clock, SMSer,
MusicPlayer {
    ...
}

class SmartWatch implements Clock, MusicPlayer {
    …
}
```

Теперь вместо одного гигантского интерфейса есть несколько различных логически независимых.

Теперь, если мы захотим в метод получить объект, умеющий выставлять будильник, нам не надо будет требовать от него умения отправлять SMS!

Dependency inversion principle

Принцип инверсии зависимостей

Dependency inversion principle

Сложные системы состоят из частей попроще - например, работающая лампа состоит из самого устройства, провода, проводки. Но правильно ли будет подключить утюг к электричеству спаивая его провода к проводке? Или достаточно просто знать, что если мы воткнём штекер в розетку, то пойдёт электричество?

Принцип инверсии зависимостей - зависьте от абстракций, а не от имплементаций

Традиционно, представляя сложные системы состоящими из более простых частей, мы добавляем зависимость высокому уровню взаимодействия от низкого. Но введя список требований (*интерфейс*), которым должен соответствовать низкий уровень мы меняем это отношение: теперь всё зависит от абстракций (*интерфейсов*), а не деталей реализации друг друга.

Dependency inversion principle

```
class Main {
  static class ConsoleLogger {
    void printInConsole(String msg) {
      System.out.println(msg);
    }
  }

  static void main(String[] args) {
    ConsoleLogger logger = new ConsoleLogger();
    logger.printInConsole("Hello!");
  }
}
```



```
public interface Logger {
  void log(String msg);
}

static class ConsoleLogger implements Logger {
  void printInConsole(String msg) {
    System.out.println(msg);
  }

  @Override
  public void log(String msg) {
    printInConsole(msg);
  }
}

public static void main(String[] args) {
  Logger logger = new ConsoleLogger();
  logger.log("Hello!");
}
}
```

Теперь вместо того, чтобы нашей логике в `main` зависеть от деталей реализации класса логгирования (в данном случае логгирования в консоль), наша логика тут будет зависеть только от абстракции процесса логгирования (от интерфейса `Logger`), как и сам консольный логгер будет зависеть от этой абстракции (т.к. он реализует этот интерфейс).

Сделав так, нам больше не надо беспокоиться о многих деталях реализации логгера, а поменять один логгер на другой (например, на отправку лога по почте) будет стоить нам лишь указания нужного конструктора, ничего другого нам менять не придётся.

Итоги

Итоги

- Мы познакомились с базовыми принципами магических чисел, DRY и SOLID, которые помогут сделать ваш код лучше: чище, читабельнее, гибче и с меньшими ошибками при разработке и внесении изменений.
- Не все принципы являются строгими законами: иногда стараясь на 100% исполнить каждый из них можно переборщить и сделать хуже
- Баланс между разными принципами написания кода есть искусство проектирования программного кода и приходит с опытом и расширением теоретической подготовки

Домашнее задание

Давайте посмотрим ваше домашнее задание.

- Вопросы по домашней работе задавайте **в чате** мессенджера .
- Задачи можно сдавать **по частям**.
- Зачёт по домашней работе проставляется после того, как **приняты все задачи**.

нетология

Задавайте вопросы и пишите отзыв о лекции!

Филипп Воронов