



MiSta1984

15 сен 2022 в 21:37

# Принципы SOLID на примерах



9 мин



106K

Java\*, Проектирование и рефакторинг\*

Тutorial

Всем привет! Данная статья - эта попытка объяснить принципы SOLID на примерах псевдокода на Java. Статья будет полезна начинающим разработчикам понять данные принципы проектирования.

Вначале рассмотрим общее понятие, что такое SOLID и как расшифровывается каждая буква данной аббревиатуры.

**SOLID** - это принципы разработки программного обеспечения, следуя которым Вы получите хороший код, который в дальнейшем будет хорошо масштабироваться и поддерживаться в рабочем состоянии.

**S - Single Responsibility Principle - принцип единственной ответственности.** Каждый класс должен иметь только одну зону ответственности.

**O - Open closed Principle - принцип открытости-закрытости.** Классы должны быть открыты для расширения, но закрыты для изменения.

**L - Liskov substitution Principle - принцип подстановки Барбары Лисков.** Должна быть возможность вместо базового (родительского) типа (класса) подставить любой его подтип (класс-наследник), при этом работа программы не должна измениться.

**I - Interface Segregation Principle - принцип разделения интерфейсов.** Данный принцип обозначает, что не нужно заставлять клиента (класс) реализовывать интерфейс, который не имеет к нему отношения.

**D - Dependency Inversion Principle - принцип инверсии зависимостей.** Модули верхнего уровня не должны зависеть от модулей нижнего уровня. И те, и другие должны зависеть от абстракции. Абстракции не должны зависеть от деталей. Детали должны зависеть от абстракций.

Рассмотрим первый принцип - **принцип единственной ответственности** на примере.

Допустим у нас есть класс RentCarService и в нем есть несколько методов: найти машину по номеру, забронировать машину, распечатать заказ, получить информацию о машине, отправить сообщение.

```

public class RentCarService {

    public Car findCar(String carNo) {
        //find car by number
        return car;
    }

    public Order orderCar(String carNo, Client client) {
        //client order car
        return order;
    }

    public void printOrder(Order order) {
        //print order
    }

    public void getCarInterestInfo(String carType) {
        if (carType.equals("sedan")) {
            //do some job
        }
        if (carType.equals("pickup")) {
            //do some job
        }
        if (carType.equals("van")) {
            //do some job
        }
    }

    public void sendMessage(String typeMessage, String message) {
        if (typeMessage.equals("email")) {
            //write email
            //use JavaMailSenderAPI
        }
    }
}

```

У данного класса есть несколько зон ответственности, что является нарушением первого принципа. Возьмем метод получения информации об машине. Теперь у нас есть только три типа машин sedan, pickup и van, но если Заказчик захочет добавить еще несколько типов, тогда придется изменять и дописывать данный метод.

Или возьмем метод отправки сообщения. Если кроме отправки сообщения по электронной почте



+13



244



26

Одним словом, данный класс нарушает принцип единой ответственности, так как отвечает за разные действия.

Необходимо разделить данный класс RentCarService на несколько, и тем самым, следуя принципу единой ответственности, предоставить каждому классу отвечать только за одну зону или действие,

так в дальнейшем его будет проще дополнять и модифицировать.

Необходимо создать класс `PrinterService` и вынести там функционал по печати.

```
public class PrinterService {  
    public void printOrder(Order order) {  
        //print order  
    }  
}
```

Аналогично работа связанная с поиском информации о машине перенести в класс `CarInfoService`.

```
public class CarInfoService {  
    public void getCarInterestInfo(String carType) {  
        if (carType.equals("sedan")) {  
            //do some job  
        }  
        if (carType.equals("pickup")) {  
            //do some job  
        }  
        if (carType.equals("van")) {  
            //do some job  
        }  
    }  
}
```

Метод по отправке сообщений перенести в класс `NotificationService`.

```
public class NotificationService {  
    public void sendMessage(String typeMessage, String message) {  
        if (typeMessage.equals("email")) {  
            //write email  
            //use JavaMailSenderAPI  
        }  
    }  
}
```

А метод поиска машины в `CarService`.

```
public class CarService {  
    public Car findCar(String carNo) {  
        //find car by number  
        return car;  
    }  
}
```

И в классе RentCarService останется только один метод.

```
public class RentCarService {  
    public Order orderCar(String carNo, Client client) {  
        //client order car  
        return order;  
    }  
}
```

Теперь каждый класс несет ответственность только за одну зону и есть только одна причина для его изменения.

**Принцип открытости-закрытости** рассмотрим на примере только что созданного класса по отправке сообщений.

```
public class NotificationService {  
    public void sendMessage(String typeMessage, String message) {  
        if (typeMessage.equals("email")) {  
            //write email  
            //use JavaMailSenderAPI  
        }  
    }  
}
```

Допустим нам необходимо кроме отправки сообщения по электронной почте отправлять еще смс сообщения. И мы можем дописать метод sendMessage таким образом:

```
public class NotificationService {  
    public void sendMessage(String typeMessage, String message) {  
        if (typeMessage.equals("email")) {  
            //write email  
            //use JavaMailSenderAPI  
        }  
        if (typeMessage.equals("sms")) {
```

```
        //write sms
        //send sms
    }

}
```

Но в данном случае мы нарушим второй принцип, потому что класс должен быть закрыт для модификации, но открыт для расширения, а мы модифицируем (изменяем) метод.

Для того чтобы придерживаться принципа открытости-закрытости нам необходимо спроектировать наш код таким образом, чтобы каждый мог повторно использовать нашу функцию, просто расширив ее. Поэтому создадим интерфейс NotificationService и в нем поместим метод sendMessage.

```
public interface NotificationService {
    public void sendMessage(String message);
}
```

Далее создадим класс EmailNotification, который имплементирует интерфейс NotificationService и реализует метод отправки сообщений по электронной почте.

```
public class EmailNotification implements NotificationService{
    @Override
    public void sendMessage(String message) {
        //write email
        //use JavaMailSenderAPI
    }
}
```

Создадим аналогично класс MobileNotification, который будет отвечать за отправку смс сообщений.

```
public class MobileNotification implements NotificationService{
    @Override
    public void sendMessage(String message) {
        //write sms
        //send sms
    }
}
```

Проектируя таким образом код мы не будем нарушать принцип открытости-закрытости, так как мы расширяем нашу функциональность, а не изменяем (модифицируем) наш класс.

Давайте сейчас рассмотрим третий принцип: **принцип подстановки Барбары Лисков**.

Данный принцип непосредственно связан с наследованием классов. Допустим у нас есть базовый класс Счет (Account), в котором есть три метода: просмотр остатка на счете, пополнение счета и оплата.

```
public class Account {  
    public BigDecimal balance(String numberAccount){  
        //logic  
        return bigDecimal;  
    };  
    public void refill(String numberAccount, BigDecimal sum){  
        //logic  
    }  
    public void payment(String numberAccount, BigDecimal sum){  
        //logic  
    }  
  
}
```

Нам необходимо написать еще два класса: зарплатный счет и депозитный счет, при этом зарплатный счет должен поддерживать все операции, представленные в базовом классе, а депозитный счет - не должен поддерживать проведение оплаты.

```
public class SalaryAccount extends Account{  
    @Override  
    public BigDecimal balance(String numberAccount){  
        //logic  
        return bigDecimal;  
    };  
    @Override  
    public void refill(String numberAccount, BigDecimal sum){  
        //logic  
    }  
    @Override  
    public void payment(String numberAccount, BigDecimal sum){  
        //logic  
    }  
  
}
```

```

public class DepositAccount extends Account{
    @Override
    public BigDecimal balance(String numberAccount){
        //logic
        return bigDecimal;
    };
    @Override
    public void refill(String numberAccount, BigDecimal sum){
        //logic
    }
    @Override
    public void payment(String numberAccount, BigDecimal sum){
        throw new UnsupportedOperationException("Operation not supported");
    }
}

```

Если сейчас в коде программы везде, где мы использовали класс Account заменить на его класс-наследник (подтип) SalaryAccount, то программа продолжит нормально работать, так как в классе SalaryAccount доступны все операции, которые есть и в классе Account.

Если же мы такое попробуем сделать с классом DepositAccount, то есть заменим базовый класс Account на его класс-наследник DepositAccount, то программа начнет неправильно работать, так как при вызове метода payment() будет выбрасываться исключение new UnsupportedOperationException. Таким образом произошло нарушение принципа подстановки Барбары Лисков.

Для того чтобы следовать принципу подстановки Барбары Лисков необходимо в базовый (родительский) класс выносить только общую логику, характерную для классов наследников, которые будут ее реализовывать и, соответственно, можно будет базовый класс без проблем заменить на его класс-наследник.

В нашем случае класс Account будет выглядеть следующим образом.

```

public class Account {
    public BigDecimal balance(String numberAccount){
        //logic
        return bigDecimal;
    };
    public void refill(String numberAccount, BigDecimal sum){
        //logic
    }
}

```

Мы сможем от него наследовать класс DepositAccount.

```
public class DepositAccount extends Account{
    @Override
    public BigDecimal balance(String numberAccount){
        //logic
        return bigDecimal;
    };
    @Override
    public void refill(String numberAccount, BigDecimal sum){
        //logic
    }
}
```

Создадим дополнительный класс PaymentAccount, который унаследуем от Account и его расширим методом проведения оплаты.

```
public class PaymentAccount extends Account{
    public void payment(String numberAccount, BigDecimal sum){
        //logic
    }
}
```

И наш класс SalaryAccount уже унаследуем от класса PaymentAccount.

```
public class SalaryAccount extends PaymentAccount{
    @Override
    public BigDecimal balance(String numberAccount){
        //logic
        return bigDecimal;
    };
    @Override
    public void refill(String numberAccount, BigDecimal sum){
        //logic
    }
    @Override
    public void payment(String numberAccount, BigDecimal sum){
        //logic
    }
}
```

Сейчас замена класса PaymentAccount на его класс-наследник SalaryAccount не "поломает" нашу программу, так как класс SalaryAccount имеет доступ ко всем методам, что и PaymentAccount. Также все будет хорошо при замене класса Account на его класс-наследник PaymentAccount.



Принцип подстановки Барбары Лисков заключается в правильном использовании отношения наследования. Мы должны создавать наследников какого-либо базового класса тогда и только тогда, когда они собираются правильно реализовать его логику, не вызывая проблем при замене родителей на наследников.

Рассмотрим теперь **принцип разделения интерфейсов**.

Допустим у нас имеется интерфейс Payments и в нем есть три метода: оплата WebMoney, оплата банковской карточкой и оплата по номеру телефона.

```
public interface Payments {  
    void payWebMoney();  
    void payCreditCard();  
    void payPhoneNumber();  
}
```

Далее нам надо реализовать два класса-сервиса, которые будут у себя реализовывать различные виды проведения оплат (класс InternetPaymentService и TerminalPaymentService). При этом TerminalPaymentService не будет поддерживать проведение оплат по номеру телефона. Но если мы оба класса имплементим от интерфейса Payments, то мы будем "заставлять" TerminalPaymentService реализовывать метод, который ему не нужен.

```
public class InternetPaymentService implements Payments{  
    @Override  
    public void payWebMoney() {  
        //logic  
    }  
    @Override  
    public void payCreditCard() {  
        //logic  
    }  
    @Override  
    public void payPhoneNumber() {  
        //logic  
    }  
}
```

```
public class TerminalPaymentService implements Payments{  
    @Override  
    public void payWebMoney() {  
        //logic  
    }  
    @Override
```

```

    public void payCreditCard() {
        //logic
    }
    @Override
    public void payPhoneNumber() {
        //???????
    }
}

```

Таким образом произойдет нарушение принципа разделения интерфейсов.

Для того чтобы этого не происходило необходимо разделить наш исходный интерфейс Payments на несколько и, создавая классы, имплементировать в них только те интерфейсы с методами, которые им нужны.

```

public interface WebMoneyPayment {
    void payWebMoney();
}

```

```

public interface CreditCardPayment {
    void payCreditCard();
}

```

```

public interface PhoneNumberPayment {
    void payPhoneNumber();
}

```

```

public class InternetPaymentService implements WebMoneyPayment,
                                                CreditCardPayment,
                                                PhoneNumberPayment{

    @Override
    public void payWebMoney() {
        //logic
    }
    @Override
    public void payCreditCard() {
        //logic
    }
    @Override
    public void payPhoneNumber() {
        //logic
    }
}

```

```
}  
}
```

```
public class TerminalPaymentService implements WebMoneyPayment, CreditCardPayment{  
    @Override  
    public void payWebMoney() {  
        //logic  
    }  
    @Override  
    public void payCreditCard() {  
        //logic  
    }  
}
```

Давайте сейчас рассмотрим последний принцип: **принцип инверсии зависимостей**.

Допустим мы пишем приложение для магазина и решаем вопросы с проведением оплат. Вначале это просто небольшой магазин, где оплата происходит только за наличные. Создаем класс Cash и класс Shop.

```
public class Cash {  
    public void doTransaction(BigDecimal amount){  
        //logic  
    }  
}
```

```
public class Shop {  
    private Cash cash;  
    public Shop(Cash cash) {  
        this.cash = cash;  
    }  
    public void doPayment(Object order, BigDecimal amount){  
        cash.doTransaction(amount);  
    }  
}
```

Вроде все хорошо, но мы уже нарушили принцип инверсии зависимостей, так как мы тесно связали оплату наличными к нашему магазину. И если в дальнейшем нам необходимо будет добавить оплату еще банковской картой и телефоном ("100% понадобится"), то нам придется переписывать и изменять много кода. Мы в нашем коде модуль верхнего уровня тесно связали с модулем нижнего уровня, а нужно чтобы оба уровня зависели от абстракции.

Поэтому создадим интерфейс Payments.

```
public interface Payments {  
    void doTransaction(BigDecimal amount);  
}
```

Теперь все наши классы по оплате будут имплементировать данный интерфейс.

```
public class Cash implements Payments{  
    @Override  
    public void doTransaction(BigDecimal amount) {  
        //logic  
    }  
}
```

```
public class BankCard implements Payments{  
    @Override  
    public void doTransaction(BigDecimal amount) {  
        //logic  
    }  
}
```

```
public class PayByPhone implements Payments {  
    @Override  
    public void doTransaction(BigDecimal amount) {  
        //logic  
    }  
}
```

Теперь надо перепроектировать реализацию нашего магазина.

```
public class Shop {  
    private Payments payments;  
  
    public Shop(Payments payments) {  
        this.payments = payments;  
    }  
  
    public void doPayment(Object order, BigDecimal amount){
```

```
payments.doTransaction(amount);  
}  
}
```

Сейчас наш магазин слабо связан с системой оплаты, то есть он зависит от абстракции и уже не важно каким способом оплаты будут пользоваться (наличными, картой или телефоном) все будет работать.

Мы рассмотрели на примерах псевдокода принципы SOLID, надеюсь кому-то будет это полезно.

Спасибо Всем, кто дочитал до конца. Всем пока.

**Теги:** принципы проектирования, принципы разработки, solid

**Хэбы:** Java, Проектирование и рефакторинг

## Редакторский дайджест

Присылаем лучшие статьи раз в месяц

Электронпочта



21

Карма

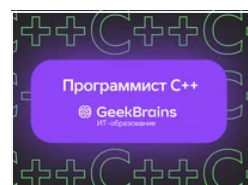
4

Рейтинг

@MiSta1984

Пользователь

Реклама



gb.ru РЕКЛАМА · 16+

## Обучение на программиста C++. 3 месяца Бесплатно

Программирование на C++ для начинающих. Наша цель - ваше трудоустройство!

Комментарии 26

## Публикации

ЛУЧШИЕ ЗА СУТКИ

ПОХОЖИЕ



dalerank

21 час назад

## Зачем в Switch SDK три разных sin?

 Простой

 8 мин

 4.9K

 +45

 16

 6



alizar

8 часов назад

## Первые агенты для самообучения сильного ИИ

 Средний

 6 мин

 2.3K

Мнение

 +28

 23

 8



EntityFX

20 часов назад

## Энтузиаст протестировал новейший процессор Loongson 3C5000

 Средний

 13 мин

 8K

 +21

 8

 18



OlegSivchenko

3 часа назад

## Долгая смерть Бетельгейзе и её научные аспекты

 9 мин

 1.7K

 +20

 7

 1



Bright\_Translate

4 часа назад

## Какие уроки я извлёк из создания расширения VSCode с помощью GPT-4

 Средний

 14 мин

 1.2K

Тutorial

Перевод

 +15

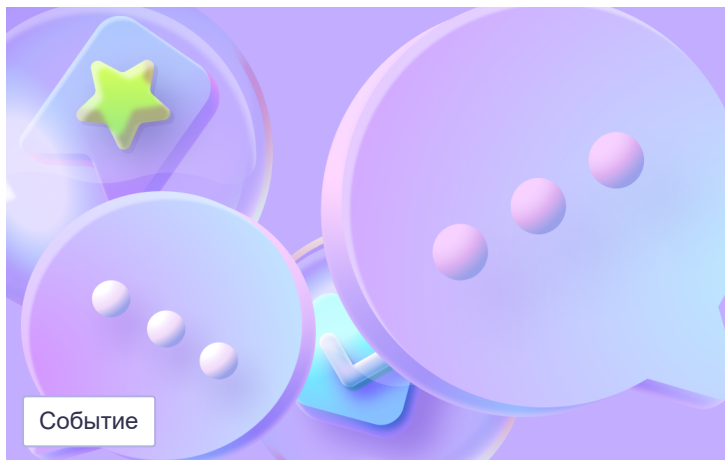
 11

 2

## Атлант исправил плечи: вторая линия поддержки IT-проектов ЕВРАЗа

Турбо

Показать еще





Налетай! У нас тут много подарков в честь обновления сервиса





Зачем дата-инженеру программировать


## КУРСЫ

 Java-разработчик с нуля  
16 июня 2023 · 109 500 Р · Нетология

 Java-разработчик  
19 июня 2023 · 147 000 Р · Яндекс Практикум

 Автоматизатор тестирования на Java  
22 июня 2023 · 84 000 Р · Яндекс Практикум

 DevOps для эксплуатации и разработки  
5 июля 2023 · 124 000 Р · Яндекс Практикум

 Офлайн-курс Java-разработчик  
21 августа 2023 · 59 900 Р · Бруноям

Больше курсов на Хабр Карьере

## ЧИТАЮТ СЕЙЧАС

На Reddit началась акция протеста против новых условий форума, многие подфорумы стали не доступны на 48 часов

 4.2K  22

Я — айтишник, я не хочу много знать

 871  1

СМИ: Twitter рискует потерять кон

1K 2

Я купил смартфон с камерой 41мп  
фотоаппаратом?

7.1K 69

Долгая смерть Бетельгейзе и её на

1.7K 1

Атлант исправил плечи: вторая ли

Турбо

ИСТОРИИ



Ваш аккаунт

- Войти
- Регистрация

Разделы

- Статьи
- Новости
- Хабы
- Компании
- Авторы
- Песочница

Информация

- Устройство сайта
- Для авторов
- Для компаний
- Документы
- Соглашение
- Конфиденциальность

Услуги

- Корпоративный блог
- Медийная реклама
- Нативные проекты
- Образовательные программы
- Стартапам
- Спецпроекты

gb.ru РЕКЛАМА · 16+

Программист С++

GeekBrains

ИТ-образование

Обучение  
на программиста С++.  
3 месяца Бесплатно

от 2 701 ₽ -69% 9 000 ₽

Программирование на С++  
для начинающих. Наша цель -  
ваше трудоустройство!

Скидка 70%

иц=Op

натъ б

Воспитай айтишника

Подборка статей о том,  
как научить детей  
программированию

КАК НАЙТИ ПЕРВУЮ РАБОТУ В ИТ

27Лабс

В ТЕМЕ:  
КАК БЫТЬ АДЕКВАТНЫМ  
и ПОЧЕМУ ЭТО ВАЖНО

Кулинарные лайфхаки  
от контент-команды  
Хабра



Настройка языка



Техническая поддержка

Вернуться на старую версию

© 2006–2023, Habr

**Обучение  
на программиста C++. 3  
месяца Бесплатно**

