Эллеонора Керри

41 уровень

12.06.2019 14971 5

Как устроен рефакторинг в Java

Статья из группы Java Developer

44255 участников

Вы в группе

Во время обучения программированию много времени уделяется написанию кода. Большинство начинающих разработчиков считают, что в этом и состоит их будущая деятельность. Отчасти это так, но в задачи программиста также входят поддержка и рефакторинг кода. Сегодня поговорим о рефакторинге.

Рефакторинг в курсе JavaRush

В курсе JavaRush тема рефакторинга затрагивается дважды:

- Большая задача на уровне 5 квеста Multithreading;
- Лекция о рефакторинге в Intellij IDEA на 9 уровне квеста Java Collections.

Благодаря большой задаче, есть возможность познакомиться с настоящим рефакторингом на практике, а лекция о рефакторинге в IDEA поможет разобраться с автоматическими средствами, которые невероятно облегчают жизнь.

Что такое рефакторинг?

Это изменение структуры кода без изменения его функционала. Например, есть метод, который сравнивает 2 числа и возвращает *true*, если первое больше, и *false* в обратном случае:

```
1  public boolean max(int a, int b) {
2      if(a > b) {
3          return true;
4      } else if(a == b) {
```

```
        return false;
    } else {
        return false;
    }
}
```

Получился очень громоздкий код. Даже новички редко пишут подобное, однако такой риск есть. Казалось бы, зачем тут блок `if-else`, если можно написать метод на 6 строк короче:

```
public boolean max(int a, int b) {
     return a>b;
}
```

Теперь этот метод выглядит просто и элегантно, хотя выполняет то же действие, что и пример выше. Так и работает рефакторинг: меняет структуру кода, не затрагивая его суть. Существует множество методов и техник рефакторинга, которые рассмотрим подробнее.

Для чего нужен рефакторинг?

Существует несколько причин. Например, погоня за простотой и лаконичностью кода. Сторонники этой теории считают, что код должен быть максимально кратким, даже если для его понимания нужно несколько десятков строк комментарий. Другие разработчики уверены, что код должен подвергаться рефакторингу настолько, чтобы он был понятен с минимальным количеством комментариев.

Каждая команда выбирает свою позицию, но нужно помнить, что **рефакторинг — это не сокращение**.

Его главная цель — улучшить структуру кода.

В эту глобальную цель можно включить несколько задач:

1. Рефакторинг улучшает понимание кода, который написан другим разработчиком;
2. Помогает искать и устранять ошибки;
3. Позволяет повысить скорость разработки ПО;
4. В целом улучшает композицию программного обеспечения.

Если долгое время не проводить рефакторинг, могут возникнуть сложности в разработке вплоть до полной остановки работы.

“Запахи кода“

Когда код требует рефакторинга говорят, что он “пахнет”. Конечно, не буквально, но такой код действительно выглядит не совсем приятно. Ниже рассмотрим основные техники рефакторинга для начального этапа.

Неоправданно большие элементы

Существуют громоздкие классы и методы, с которыми невозможно эффективно работать именно из-за их огромного размера.

Большой класс

У такого класса есть огромное количество строк кода и много различных методов. Обычно разработчику легче добавить фичу в существующий класс, а не создавать новый, из-за чего он и растет. Как правило, функционал такого класса перегружен. В этом случае помогает выделение части функционала в отдельный класс. Об этом поговорим подробнее в разделе техник рефакторинга.

Большой метод

Этот “запах” возникает, когда разработчик добавляет в метод новый функционал. “Зачем мне выносить проверку параметров в отдельный метод, если я могу написать ее тут?”, “Для чего необходимо выделять метод поиска максимального элемента в массиве, оставим его тут. Так код яснее”, — и прочие заблуждения.

Есть два правила рефакторинга большого метода:

1. Если при написании метода хочется добавить комментарий в код, необходимо выделить этот функционал в отдельный метод;
2. Если метод занимает более 10-15 строк кода, следует определить задачи и подзадачи, которые он выполняет, и попробовать вынести подзадачи в отдельный метод.

Несколько способов устранить большой метод:

- Выделить часть функционала метода в отдельный метод;
- Если локальные переменные не дают вынести часть функционала, можно передать весь объект в другой метод.

Использование множества примитивных типов данных

Обычно такая проблема возникает, когда с течением времени в классе растет количество полей для хранения данных. Например, если использовать примитивные типы вместо маленьких объектов для хранения данных (валюта, дата, телефонные номера и т.д.) или константы для кодирования какой-либо информации.

Хорошей практикой в этом случае будет логическая группировка полей и вынос в отдельный класс (выделение класса). Также в класс можно включить методы для обработки этих данных.

Длинный список параметров

Достаточно распространенная ошибка, особенно в совокупности с большим методом. Обычно она возникает, если функционал метода перегружен, или метод объединяет несколько алгоритмов в себе.

В длинных списках параметров очень трудно разбираться, и использовать такие методы неудобно. Поэтому лучше передать объект целиком. Если у объекта нет достаточно данных, стоит использовать более общий объект или разделить функционал метода, чтобы он обрабатывал логически связанные данные.

Группы данных

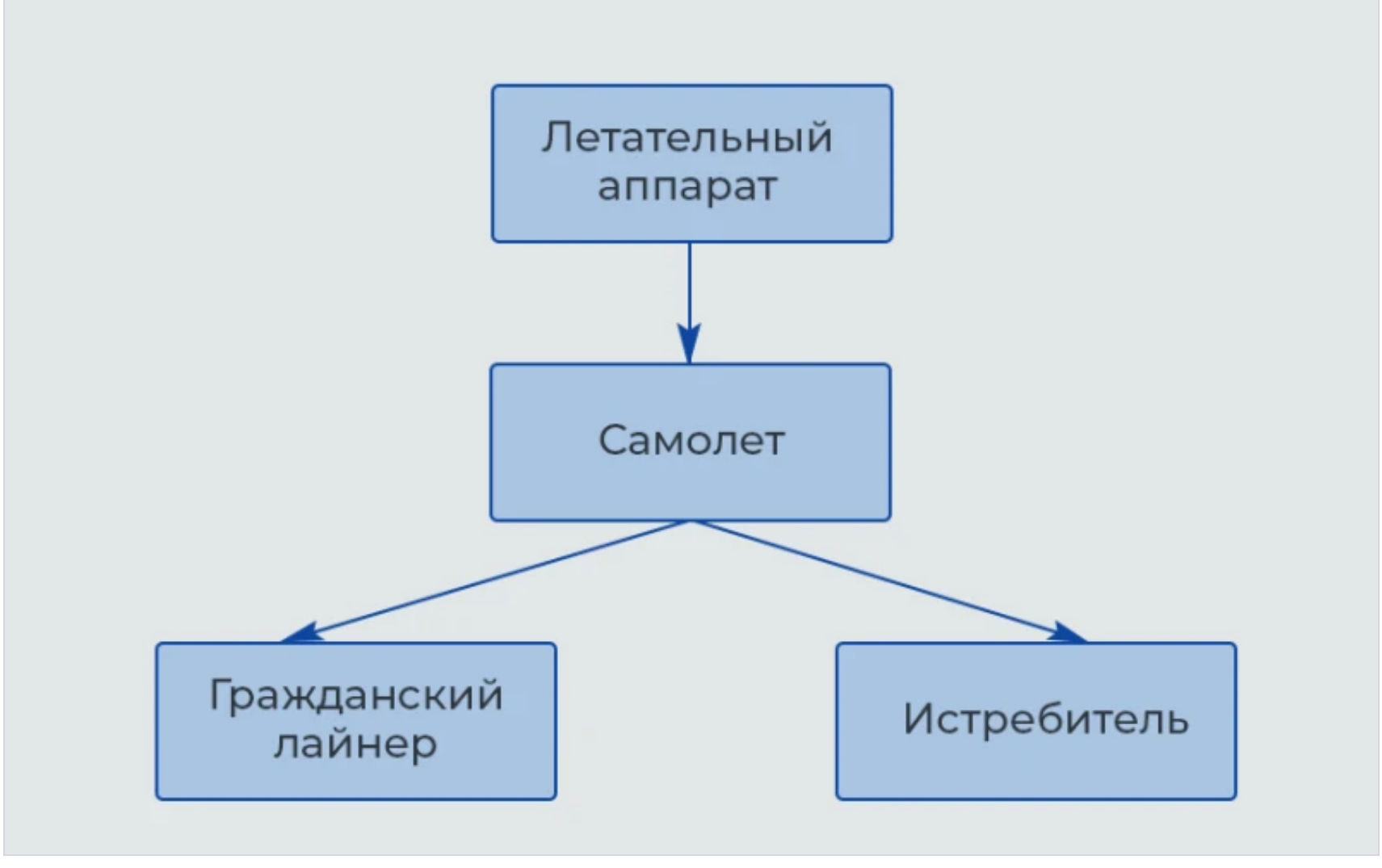
Часто в коде появляются логически связанные группы данных. Например, параметры подключения в БД (URL, имя пользователя, пароль, имя схемы и тд). Если из перечня элементов нельзя удалить ни одно поле, значит перечень — это группа данных, которую необходимо вынести в отдельный класс (выделение класса).

Решения, которые портят концепцию ООП

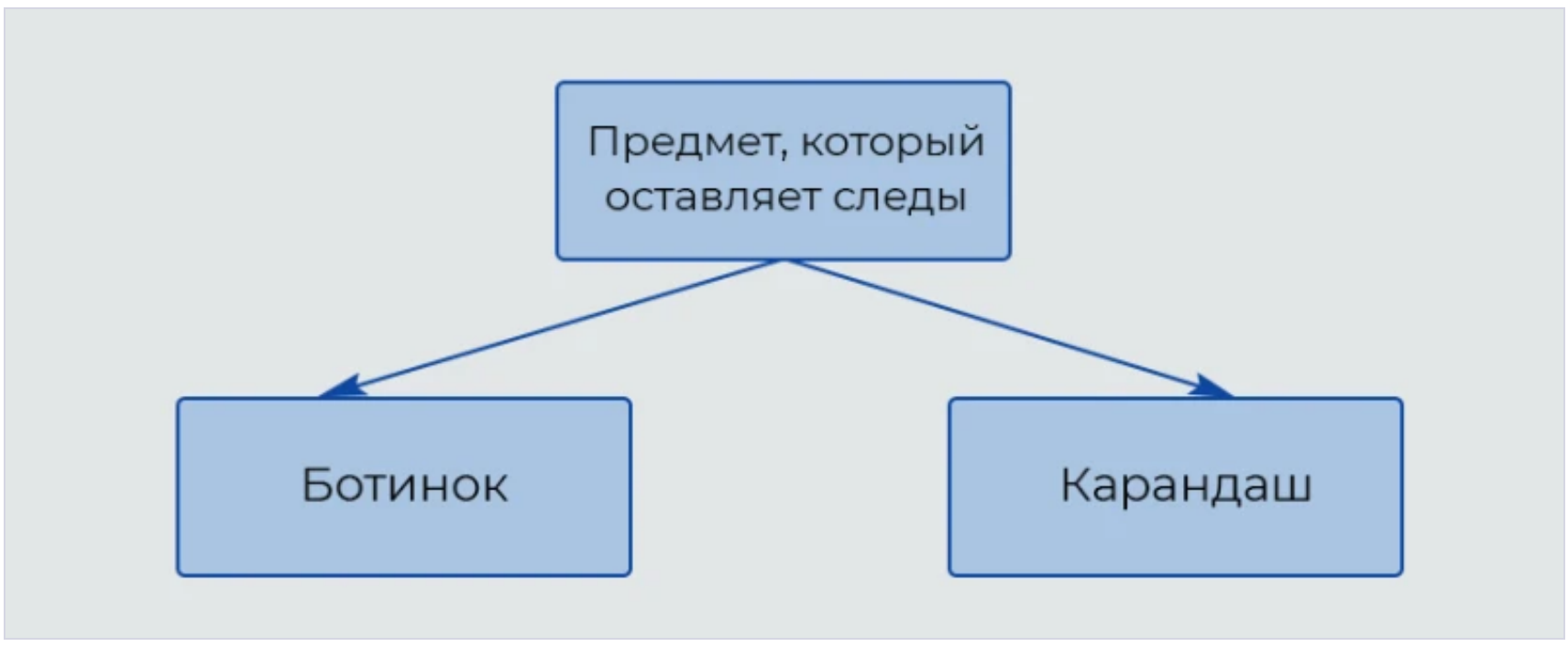
“Запахи” этого типа возникают, когда разработчик нарушает дизайн ООП. Такое происходит, если он не до конца понимает возможности этой парадигмы, использует их не до конца или неправильно.

Отказ от наследования

Если подкласс использует минимальную часть функций родительского класса, тут пахнет неправильной иерархией. Обычно в таком случае ненужные методы просто не переопределяются или выбрасывают исключения. Если класс унаследован от другого, это подразумевает под собой практически полное использование его функционала. Пример правильной иерархии:



Пример неправильной иерархии:



Оператор switch

Что плохого может быть в операторе `switch`? Он плох, когда его конструкция очень сложная. Также сюда относятся и множество вложенных блоков `if`.

Альтернативные классы с разными интерфейсами

Несколько классов фактически выполняют одно и то же, но их методы называются по-разному.

Временное поле

Если в классе заложено временное поле, которое нужно объекту лишь изредка, когда он заполняется значениями, а в остальное время — пустое или, не дай бог, `null`, значит код “попахивает”, а такой дизайн — сомнительное решение.

Запахи, которые затрудняют модификацию

Эти “запахи” более серьезные. Остальные в основном ухудшают понимание кода, тогда как эти не дают возможность его модифицировать. При внедрении каких-либо фич половина разработчиков уволится, а половина сойдет с ума.

Параллельные иерархии наследования

При создании подкласса какого-либо класса необходимо создавать еще один подкласс для другого класса.

Равномерное распределение зависимости

При выполнении любых модификаций приходится искать все зависимости (использования) этого класса и вносить множество мелких правок. Одно изменение — правки во множестве классов.

Сложное дерево модификаций

Этот запах противоположен предыдущему: изменения затрагивают большое количество методов одного класса. Как правило, зависимость в таком коде каскадная: изменив один метод, нужно поправить что-то в другом, а затем в третьем и так далее. Один класс — множество изменений.

“Мусорные запахи”

Достаточно неприятная категория запахов, которая вызывает головную боль. Бесполезный, ненужный, старый код. К счастью, современные IDE и линтеры научились предупреждать о таких запахах.

Большое количество комментариев в методе

У метода очень много поясняющих комментариев практически на каждой строке. Обычно это связано со сложным алгоритмом, поэтому лучше разделить код на несколько методов поменьше и дать им говорящие названия.

Дублирование кода

В разных классах или методах используются одинаковые блоки кода.

Ленивый класс

Класс берет на себя очень малый функционал, хотя планировался большой.

Неиспользуемый код

Класс, метод или переменная не используется в коде и являются “мертвым грузом”.

Излишняя связанность

Эта категория запахов характеризуется большим количеством неоправданных связей в коде.

Сторонние методы

Метод использует данные другого объекта гораздо чаще, чем собственные данные.

Неуместная близость

Класс использует служебные поля и методы другого класса.

Длинные вызовы классов

Один класс вызывает другой, тот запрашивает данные у третьего, тот у четвертого и так далее. Такая длинная цепь вызовов означает высокий уровень зависимости от текущей структуры классов.

Класс-таск-дилер

Класс нужен только для того, чтобы передать задание другому классу. Может быть, его стоит удалить?

Техники рефакторинга

Ниже пойдет речь о начальных техниках рефакторинга, которые помогут устранить описанные “запахи” кода.

Выделение класса

Класс выполняет слишком много функций, часть необходимо вынести в другой класс.

Например, имеется класс `Human`, в котором также содержится адрес проживания и метод, предоставляющий полный адрес:

```
class Human {
   private String name;
   private String age;
   private String country;
```

```
    private String city;
    private String street;
    private String house;
    private String quarter;

    public String getFullAddress() {
        StringBuilder result = new StringBuilder();
        return result
                        .append(country)
                        .append(", ")
                        .append(city)
                        .append(", ")
                        .append(street)
                        .append(", ")
                        .append(house)
                        .append(" ")
                        .append(quarter).toString();
    }
}
```

Хорошим тоном будет вынести информацию об адресе и метод (поведение обработки данных) в отдельный класс:

```
class Human {
    private String name;
    private String age;
    private Address address;

    private String getFullAddress() {
        return address.getFullAddress();
    }
}
class Address {
    private String country;
    private String city;
    private String street;
    private String house;
    private String quarter;

    public String getFullAddress() {
        StringBuilder result = new StringBuilder();
        return result
                        .append(country)
                        .append(", ")
                        .append(city)
                        .append(", ")
                        .append(street)
                        .append(", ")
                        .append(house)
                        .append(" ")
                        .append(quarter).toString();
    }
}
```

Выделение метода

Если в методе какой-либо функционал можно сгруппировать, следует вынести его в отдельный метод. Например, метод, который вычисляет корни квадратного уравнения:

```
public void calcQuadraticEq(double a, double b, double c) {
    double D = b * b - 4 * a * c;
    if (D > 0) {
        double x1, x2;
        x1 = (-b - Math.sqrt(D)) / (2 * a);
        x2 = (-b + Math.sqrt(D)) / (2 * a);
        System.out.println("x1 = " + x1 + ", x2 = " + x2);
    }
    else if (D == 0) {
        double x;
        x = -b / (2 * a);
        System.out.println("x = " + x);
    }
    else {
        System.out.println("Equation has no roots");
    }
}
```

Вынесем вычисление всех трех возможных вариантов в отдельные методы:

```
public void calcQuadraticEq(double a, double b, double c) {
    double D = b * b - 4 * a * c;
    if (D > 0) {
        dGreaterThanZero(a, b, D);
    }
    else if (D == 0) {
        dEqualsZero(a, b);
    }
    else {
        dLessThanZero();
    }
}

public void dGreaterThanZero(double a, double b, double D) {
    double x1, x2;
    x1 = (-b - Math.sqrt(D)) / (2 * a);
    x2 = (-b + Math.sqrt(D)) / (2 * a);
    System.out.println("x1 = " + x1 + ", x2 = " + x2);
}

public void dEqualsZero(double a, double b) {
    double x;
    x = -b / (2 * a);
    System.out.println("x = " + x);
}

public void dLessThanZero() {
    System.out.println("Equation has no roots");
}
```

Код каждого метода стал гораздо короче и понятнее.

Передача всего объекта

При вызове метода с параметрами иногда можно встретить такой код:

```
public void employeeMethod(Employee employee) {
    // Некоторые действия
    double yearlySalary = employee.getYearlySalary();
    double awards = employee.getAwards();
    double monthlySalary = getMonthlySalary(yearlySalary, awards);
    // Продолжение обработки
}

public double getMonthlySalary(double yearlySalary, double awards) {
     return (yearlySalary + awards)/12;
}
```

В методе `employeeMethod` целых 2 строки отводится на получение значений и сохранение их в примитивных переменных. Иногда такие конструкции занимают до 10 строчек. Гораздо проще передать в метод сам объект, откуда можно извлечь необходимые данные:

```
public void employeeMethod(Employee employee) {
    // Некоторые действия
    double monthlySalary = getMonthlySalary(employee);
    // Продолжение обработки
}

public double getMonthlySalary(Employee employee) {
    return (employee.getYearlySalary() + employee.getAwards())/12;
}
```

Просто, кратко и лаконично.

Логическая группировка полей и вынос в отдельный класс

Несмотря на то, что вышеописанные примеры — очень простые и при взгляде на них многие могут задаться вопросом “Да кто вообще так делает?”, многие разработчики от невнимательности, нежелания проводить рефакторинг кода или просто “И так сойдет” допускают подобные структурные ошибки.

Почему рефакторинг эффективен

Итог хорошего рефакторинга — программа, код которой легко читать, модификации логики программы не становятся угрозой, а внесение новых фич не превращается в ад разбора кода, а приятным занятием на пару дней.

Рефакторинг не стоит применять, если программу проще переписать с нуля. Например, команда оценивает трудозатраты на разбор, анализ и рефакторинг кода выше, чем на реализацию такого же функционала с нуля. Или у кода, который нужно отрефакторить, есть множество ошибок, сложных в отладке.

Знание, как улучшить структуру кода обязательно в работе программиста. Ну а изучать программирование на Java лучше на JavaRush — онлайн-курсе с акцентом на практику. 1200+ задач с мгновенной проверкой, около 20 минипроектов, задачи-игры — все это поможет почувствовать себя уверенно в кодинге. Лучшее время, чтобы начать — сейчас :)

Ресурсы для дополнительного погружения в рефакторинг

Самая известная книга о рефакторинге — это “Рефакторинг. Улучшение проекта существующего кода” Мартина Фаулера.

Также есть интересное издание о рефакторинге, написанное на основе предыдущей книги — “Рефакторинг с использованием шаблонов” Джошуа Кириевски.

Кстати о шаблонах. При рефакторинге всегда очень полезно знать основные паттерны проектирования приложений. В этом помогут эти отличные книги:

1. “Паттерны проектирования” — авторства Эрика Фримена, Элизабет Фримен, Кэтти Сьерра, Берта Бейтса из серии Head First;
2. “Читаемый код, или программирование как искусство” — Дастин Босуэлл, Тревор Фаучер.
3. “Совершенный код” Стива Макконнелла, в которой изложены принципы красивого и элегантного кода.

Ну и несколько статей о рефакторинге:

1. Адская задачка: приступаем к рефакторингу унаследованного кода;
2. Рефакторинг;
3. Refactoring for everyone.

Эллеонора Керри

+40

Комментарии (5) популярные новые старые

JavaCoder

Введите текст комментария

funbiscuit Уровень 41, Россия 29 января 2021, 13:00

Какой-то очень сомнительный пример с поиском корней. Да, суть понятна, что можно функционал вынести в отдельные методы. Но в конкретно данном примере это только ухудшило читаемость. У нас был метод всего лишь на 15 строк, в котором глаза видели сразу весь контекст и все было понятно. А после этого у нас три метода, одновременно сразу всё можно и не увидеть. Особенно весело выносить один sout в отдельный метод. В общем методы на 20 строк вряд ли стоит разбивать на отдельные. Именно так и получают большие классы на 100500 методов, которые по сути могли бы быть гораздо проще.

Ответить +5

LuneFox Уровень 41, Москва, Россия EXPERT 21 марта, 22:03

Я так понимаю, что несмотря на разные техники нужно придерживаться здравого смысла и добиваться максимальной читаемости кода интуитивно. Если через 2 недели после написания своего кода он кажется нечитаемым и неуютным - пора рефакторить.

Ответить +3

виктор Уровень 29, Москва, Россия 21 октября 2019, 10:37

"летальный аппарат")
спасибо, посмешили)

Ответить +2

Justinian Judge в **Mega City One** MASTER 23 июня 2019, 14:48

Само собой, чтобы стать гуру рефакторинга, для начала нужно ~~выучить~~ изучить основы/базу Java и прокачать навыки программирования. Для этих целей и нужен JavaRush — онлайн-кур***С*** по изучению программирования на Java, на 80% состоящий из практики.

п.с. Программирование,музыку, живопись, литературу, компьютер нельзя выучить, это предметные области, которыми можно овладеть на определенном уровне.

Ответить +6

Hanna Moruga Chief editor @ JavaRush 24 июня 2019, 09:57

Good point :) Поправили.

Ответить

+3

ОБУЧЕНИЕ

Курсы программирования

Курс Java

Помощь по задачам

Подписки

Задачи-игры

СООБЩЕСТВО

Пользователи

Статьи

Форум

Чат

Истории успеха

Активности

КОМПАНИЯ

О нас

Контакты

Отзывы

FAQ

Поддержка

RUSH

JavaRush — это интерактивный онлайн-курс по изучению Java-программирования с нуля. Он содержит 1200 практических задач с проверкой решения в один клик, необходимый минимум теории по основам Java и мотивирующие фишки, которые помогут пройти курс до конца: игры, опросы, интересные проекты и статьи об эффективном обучении и карьере Java-девелопера.

ПОДПИСЫВАЙТЕСЬ

ЯЗЫК ИНТЕРФЕЙСА

Русский