

Константин

Backend Developer в Andersen

03.03.2020 9666 11

Правила написания кода: от создания системы до работы с объектами

Статья из группы Random
1728497 участников

Вы в группе

Всем доброго дня суток: сегодня мне хотелось бы поговорить с вами о правильном написании кода.

Когда я только начинал программировать, нигде не было чётко написано, что вот так писать можно, а если вот так напишешь, я найду тебя и.... В итоге в голове у меня было большое количество вопросов: а как правильно писать, каких принципов стоит придерживаться в том или ином участке программы и т.д.



Ну, не всем вот так сразу хочется вгрызаться в книги типа Clean Code, так как написано в них много, а поначалу понятно мало. Да и пока дочитаешь, всё желание кодить можно отбить.

Поэтому исходя из всего вышесказанного, сегодня я хочу предоставить вам небольшое руководство (свод небольших рекомендаций) для написания более высокоуровневого кода. В этой статье пройдемся по основным правилам и концепциям, которые касаются создания системы, работы с интерфейсами, классами и объектами.

Прочтение этого материала не займёт у вас много времени и, надеюсь, не даст заскучать. Я пойду “сверху вниз”, то есть, от общей структуры приложения к более узконаправленным деталям.



Система

Общие желательные характеристики системы таковы:

- минимальная сложность — необходимо избегать слишком усложненных проектов. Главное — это простота и понятность (лучшее=простое);
- простота сопровождения — при создании приложения необходимо помнить, что его нужно будет поддерживать (даже если это будете не вы), поэтому код должен быть понятным и очевидным;
- слабое сопряжение — это минимальное количество связей между разными частями программы (максимальное использование принципов ООП);
- возможность переиспользования — проектирование системы с возможностью переиспользовать её фрагменты в других приложениях;
- портируемость — система должна быть легко адаптирована к другой среде;
- единый стиль — проектирование системы в едином стиле в разных её фрагментах;
- расширяемость (масштабируемость) — улучшение системы без нарушения ее базовой структуры (если добавить или изменить какой-то фрагмент, это не должно влиять на остальные).

Построить приложение не требующее доработок, без добавления функционала, — фактически невозможно. Нам постоянно нужно будет вводить новые элементы, чтобы наше детище могло идти в ногу со временем. И тут на сцену выходит масштабируемость.

Масштабируемость — это по сути расширение приложения, добавление нового функционала, работа с большим количеством ресурсов (или, по-другому, с большей нагрузкой). То есть мы должны придерживаться некоторых правил, таких как уменьшение связанности системы за счёт увеличения модульности, чтобы было проще добавлять новую логику.

Этапы проектирования системы

1. Программная система — проектирование приложения в общем виде.
2. Разделение на подсистемы/пакеты — определение логически разделяемых частей и определение правил взаимодействия между ними.
3. Разделение подсистем на классы — разделение частей системы, на конкретные классы и интерфейсы, а также определение взаимодействия между ними.
4. Разделение классов на методы — полное определение нужных методов для класса, исходя из задачи этого класса.
Проектирование методов — детальное определение функциональности отдельных методов.

Обычно проектированием и занимаются рядовые разработчики, а теми пунктами, что описаны выше — архитектор приложения.

Главные принципы и концепции проектирования системы

Идиома отложенной инициализации

Приложение не тратит время на создание объекта до момента его непосредственного использования, что ускоряет процесс инициализации и уменьшает загрузку сборщика мусора. Но свою очередь, с этим не стоит перегибать, так как это может вести

к нарушению модульности. Возможно, стоит перенести все моменты конструирования в какую-то определенную часть, например, main, или в класс работающий по [принципу фабрики](#).

Один из аспектов качественного кода — отсутствие часто повторяющегося, шаблонного кода. Как правило, такой код выносится в отдельный класс, чтобы его можно было вызвать в нужный момент.

АОП

Отдельно хотелось бы отметить аспектно ориентированое программирование. Это программирование путём внедрения сквозной логики, то есть повторяющийся код выносится в классы — аспекты, и вызывается при достижении определенных условий. Например, при обращении к методу с определенным названием или обращение к переменной определенного типа.

Иногда аспекты могут путать, так как не сразу понятно, откуда вызывается код, но тем не менее, это очень полезный функционал. В частности, при кешировании или логгировании: мы навешиваем этот функционал, при этом не добавляя дополнительную логику в обычные классы.

Почитать больше про оап можно [тут](#).

4 правила проектирования простой архитектуры Согласно Кенту Беку

1. Выразительность — необходимость чётко выраженной цели класса, достигается путём правильного именования, небольшого размера и соблюдения принципа single responsibility (немного ниже рассмотрим подробнее).
2. Минимум классов и методов — в своём стремлении разбить классы на как можно более мелкие и однонаправленные можно зайти слишком далеко (антипатерн — стрельба дробью). Этот принцип призывает всё же сохранять компактность системы и не заходить слишком далеко, создавая по классу на каждый чих.
3. Отсутствие дублирования — лишний код, который путает, — признак не лучшего проектирования системы, выносится в отдельное место.
4. Выполнение всех тестов — система, прошедшая все тесты, контролируема, так как любое изменение может повлечь за собой падение тестов, что может показать нам — изменение внутренней логики метода потянуло и изменение ожидаемого поведения.

SOLID

При проектировании системы стоит учитывать и общеизвестные принципы SOLID:

- S — single responsibility — принцип единственной ответственности;
- O — open-closed — принцип открытости/закрытости;
- L — Liskov substitution — принцип подстановки Барбары Лисков;
- I — interface segregation — принцип разделения интерфейса;
- D — dependency inversion — принцип инверсии зависимостей;

Конкретно на каждом принципе останавливаться не будем (это немного выходит за рамки данной статьи, но [тут](#) можно ознакомиться подробнее

Interface

Пожалуй, один из самых важных этапов создания адекватного класса — это создание адекватного интерфейса, который будет представлять хорошую абстракцию, скрывающую детали реализации класса, и при этом будет представлять группу методов, чётко согласующихся между собой.

Рассмотрим подробнее один из принципов SOLID — interface segregation: клиенты (классы) не должны реализовывать ненужные методы, которые они не будут использовать. То есть, если речь идет о построении интерфейсов с минимальным количеством методов, которые направлены на выполнение единственной задачи этого интерфейса (как по мне, очень схоже с single responsibility), лучше вместо одного раздутого интерфейса создать пару более мелких. Благо, класс может реализовывать не один интерфейс, как в случае с наследованием.

Также нужно помнить о правильном именовании интерфейсов: название должно как можно более точно отображать его

задачу. И, конечно, чем оно будет короче, тем меньше путаницы вызовет.

Именно на уровне интерфейсов обычно пишут комментарии для документации, которые, в свою очередь, помогают нам подробно описать, что метод должен делать, какие аргументы принимает и что он вернёт.

Класс



Давайте рассмотрим внутреннюю организацию классов. А точнее, некоторые взгляды и правила, которых стоит придерживаться при построении классов.

Как правило, класс должен начинаться со списка переменных, расположенных в определенном порядке:

1. public static константы;
2. private static константы;
3. private переменные экземпляра.

Далее идут разнообразные конструкторы в порядке от меньшего количества аргументов к большему.

После них следуют методы от с более открытого доступа до самых закрытых: как правило приватные методы, скрывающие реализацию некоторого функционала, который мы хотим ограничить, стоят в самом низу.

Размер класса

Теперь хотелось бы поговорить о размере класса.



Вспомним один из принципов SOLID — single responsibility.

Single responsibility — принцип единственной ответственности. Он гласит, что у каждого объекта есть лишь одна цель (ответственность), и логика всех его методов направлена на ее обеспечение.

То есть исходя из этого, мы должны избегать больших, раздутых классов (что по своей природе — антипатерн — «божественный объект»), и если у нас очень много методов разнообразной, разнотипной логики в классе, нужно задуматься о том, чтобы разбить его на пару логичных частей (классов). Это, в свою очередь, повысит читаемость кода, так как нам не нужно много времени, чтобы понять цель метода, если мы знаем примерное назначение данного класса.

Также нужно и следить за именем класса: оно должно отображать содержащуюся в нём логику. Скажем, если у нас класс, в имени которого 20+ слов, нужно задуматься о рефакторинге.

У каждого уважающего себя класса должно быть не такое уж и большое количество внутренних переменных. Фактически каждый метод работает с одной из них или с несколькими, что вызывает большую связанность внутри класса (что и собственно и должно быть, так как класс должен быть как единое целое).

Как итог, повышение связанности класса приводит к уменьшению его как такового, ну и, понятное дело, у нас увеличивается количество классов. Некоторых это напрягает, так нужно больше ходить по классам, чтобы, увидеть как работает конкретная крупная задача.

Помимо всего прочего, каждый класс — это небольшой модуль, который должен быть минимально связан с другими. Подобная изолированность уменьшает количество изменений, которые нам нужно внести при добавлении дополнительной логики в какой-то класс.

Объекты



Инкапсуляция

Тут мы в первую очередь поговорим об одном из принципов ООП — инкапсуляции. Итак, скрытие реализации не сводится к созданию прослойки метода между переменными (бездумное ограничение доступа через одиночные методы, геттеры и сеттеры, что не есть хорошо, так как весь смысл инкапсулирования теряется). Скрытие доступа направленно на формирование абстракций, то есть класс предоставляет общие конкретные методы, посредством которых мы работаем с нашими данными. А знать, как именно мы работаем с этими данными, пользователю не обязательно — работает да и ладно.

Закон Деметры

Также можно рассмотреть закон Деметры: это небольшой набор правил, который помогает в управлении сложностью на уровне классов и методов.

Итак, предположим , что у нас есть объект `Car`, и у него есть метод — `move(Object arg1, Object arg2)`. Согласно закону Деметры, этот метод ограничивается вызовом:

- методов самого объекта `Car` (иначе говоря — `this`);
- методов объектов, созданных в `move`;
- методов переданных объектов в качестве аргументов — `arg1`, `arg2`;
- методов внутренних объектов `Car` (тот же `this`).

Иначе говоря, закон Деметры — это нечто вроде детского правила — разговаривать можно с друзьями, но не с чужаками.

Структура данных

Структура данных — это набор связанных элементов. При рассмотрении объекта как структуры данных — набор элементов данных, с которыми работают методы, существование которых подразумевается неявно.

То есть это объект, целью которого является хранение и работа (обработка) хранимых данных. Ключевое отличие от обычного объекта заключается в том, что объект — это набор методов, которые работают с элементами данными, существование которых подразумевается неявно. Понимаете? В обычном объекте главным аспектом являются методы, и внутренние переменные направлены на их правильную работу, а в структуре данных наоборот: методы поддерживают, помогают работать с хранимыми элементами, которые здесь и являются главными.

Одна из разновидностей структур данных — Data Transfer Object (DTO). Это класс с открытыми переменными и без методов (или только методами для чтения/записи), которые передают данные при работе с базами данных, работают с парсингом сообщений из сокетов и т. д.

Обычно в таких объектах данные долго не хранятся и почти сразу конвертируются в сущность, с которой и работает наше приложение. Сущность, в свою очередь, тоже получается структурой данных, но ее предназначение — участвовать в бизнес-

логике на разных уровнях приложения, а у DTO — транспортировать данные в/из приложения. Пример DTO:

```
1  @Setter
2  @Getter
3  @NoArgsConstructor
4  public class UserDto {
5      private long id;
6      private String firstName;
7      private String lastName;
8      private String email;
9      private String password;
10 }
```

Всё вроде бы понятно, но тут мы узнаем о существовании гибридов.

Гибриды — это объекты, которые содержат методы для обработки важной логики и хранят внутренние элементы и методы доступа к ним (get/set). Подобные объекты сумбурны, и усложняют добавление новых методов. Не стоит использовать их, так как непонятно, для чего они предназначены — для хранения элементов или выполнения какой-то логики.

Про возможные типы объектов можно почитать [тут](#).

Принципы создания переменных



Давайте немного порассуждаем на тему переменных, а точнее, подумаем: какие могут быть принципы их создания:

1. В идеале нужно объявлять и инициализировать переменную непосредственно перед её использованием (а не создали, и забыли про неё).
2. По возможности объявлять переменные как `final`, чтобы предотвратить изменение её значения после инициализации.
3. Не забывать про переменные-счётчики (обычно мы их используем в каком нибудь цикле `for`, то есть нужно не забывать их обнулить, иначе это может сломать нам всю логику).
4. Нужно стараться инициализировать переменные в конструкторе.
5. Если существует выбор между использованием объекта со ссылкой или без (`new SomeObject()`), делайте выбор в пользу без, так как этот объект после использования удалится во время следующей сборки мусора и не будет использовать ресурсы зря.
6. Делайте время жизни переменных как можно короче (расстояние между созданием переменной и последним обращением).
7. Инициализируйте переменные, используемые в цикле, непосредственно перед циклом, а не в начале метода, содержащего цикл.
8. Начинайте всегда с самой ограниченной области видимости и расширяйте её только при необходимости (нужно стараться делать переменную как можно более локальной).
9. Используйте каждую переменную только с одной целью.
10. Избегайте переменных со скрытым смыслом (переменная разрывается между двумя задачами, значит для решения одной из них её тип не подходит).

Научитесь программировать с нуля с JavaRush:

1200 задач, автопроверка решения и стиля кода

НАЧАТЬ ОБУЧЕНИЕ

Методы



Перейдём непосредственно к реализации нашей логики, а именно — к методам.

1. Первое правило — это компактность. В идеале один метод не должен превышать 20 строк, поэтому если, скажем, `public` метод значительно “разбухнет”, нужно задуматься о выносе отделяемой логики в приватные методы.
2. Второе правило — блоки в командах `if`, `else`, `while` и так далее, не должны иметь большую вложенность: это значительно понижает читаемость кода. В идеале вложенность должна быть не более двух блоков `{}`.

Код в этих блоках тоже желательно делать компактным и простым.

3. Третье правило — метод должен выполнять только одну операцию. То есть, если метод выполняет сложную разнообразную логику, мы его бьём на подметоды. В итоге сам метод будет фасадом, цель которого — вызов всех остальных операций в правильном порядке.

Но что если операция кажется слишком простой для создания отдельного метода? Да, иногда это может показаться пальбой из пушки по воробьям, но небольшие методы обеспечивают ряд преимуществ:

- облегченное чтение кода;
 - методы имеют свойство усложняться с течением разработки, и если метод изначально был простым, усложнение его функционала будет немного проще;
 - сокрытие деталей реализации;
 - облегчение повторного использования кода;
 - более высокая надёжность кода.
4. Правило понижения — код должен читаться сверху вниз: чем ниже — тем большее углубление в логику, и наоборот, чем выше — тем более абстрактные методы. Например, команды `switch` довольно-аки некомпактны и нежелательны, но если без использования переключателя никак, нужно постараться вынести его как можно ниже, в самые низкоуровневые методы.
 5. Аргументы метода — какое их количество идеально? В идеале их вовсе нет)) Но разве так бывает? Однако нужно стараться иметь их как можно меньше, ведь чем их меньше, тем проще использовать данный метод и легче его протестировать. Если сомневаешься, попробуй угадать все сценарии использования метода с большим количеством входящих аргументов.
 6. Отдельно хотелось бы выделить методы, имеющие входящим аргументом некий `boolean` флаг, так как это само собой подразумевает, что данный метод реализует более одной операции (если `true` то одна, `false` — другая),. Как я писал выше, это не есть хорошо и по возможности этого следует избегать.

7. Если у метода большое количество входящих аргументов (крайнее значение — 7, но стоит задумываться уже после 2-3), необходимо сгруппировать некоторые аргументы в отдельном объекте.
8. Если есть несколько похожих методов (перегруженных), то похожие параметры необходимо передавать в одном и том же порядке: это повышает читаемость и удобство использования.
9. Когда вы передаёте в метод параметры, вы должны быть уверены, что они все будут использованы, иначе зачем нужен этот аргумент? Выпилите его из интерфейса да и всё.
10. `try/catch` выглядит по своей природе не очень красиво, поэтому неплохим ходом было бы вынести его в промежуточный отдельный метод (метод для обработки исключений):

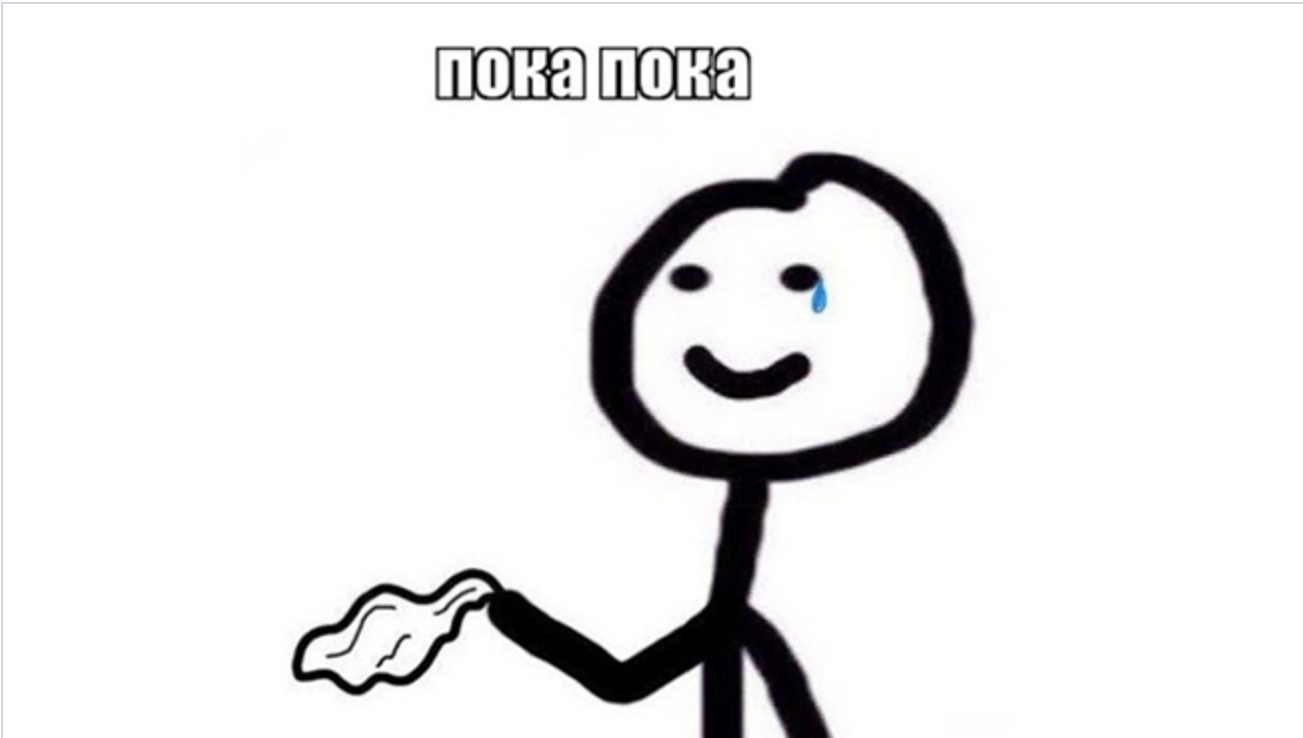
```
1 public void exceptionHandling(SomeObject obj) {
2     try {
3         someMethod(obj);
4     } catch (IOException e) {
5         e.printStackTrace();
6     }
7 }
```

Насчёт повторяющего кода я говорил выше, но добавлю и здесь:

Если у нас есть пара методов с повторяющимися частями кода, необходимо вынести его в отдельный метод, что повысит компактность как метода, так и класса.

И не стоит забывать о правильных наименованиях. Детали правильного именования классов, интерфейсов, методов и переменных я расскажу в следующей части статьи.

А нам этом у меня сегодня всё.



[Правила написания кода: сила правильных именований, хорошие и плохие комментарии](#)

Константин

Backend Developer в Andersen

-

+94

+

Комментарии (11)

популярные

новые

старые

JavaCoder

Введите текст комментария

Red Baron

Backend Developer

4 ноября 2021, 09:04

...

Заменил бы слово "простота" словом "лёгкость".
А так - да. Прочитал с удовольствием и пользой для себя.
Спасибо :)

Ответить

-

0

+

Igor

Java/Kotlin Developer

21 июля 2021, 13:22

...

Статья огонь. Всё кажется логичным и разумеющимся. Но как только доходит до написания кода, то мысли не о том, чтобы было по-фейншю читаемо, а чтобы это хоть как-то работало.

Ответить

-

+1

+

LuneFox

Уровень 41, Москва, Россия

EXPERT

21 марта, 22:33

...

Поэтому разработка ведётся в два этапа - сначала пишешь, чтобы это хоть как-то работало, а потом уже полируешь тряпочкой рефакторишь :)

Ответить

-

+1

+

Valua Sinicyn

Уровень 41, Харьков, Украина

4 марта 2021, 10:54

...

Блок try/catch не выглядит красиво (!), поэтому вынесите его в отдельный метод. Эстетика епта. За то создать +1 избыточный метод, это красиво.
А так, в целом, норм статья. Есть полезная инфа.

Ответить

-

+1

+

Хорс

Уровень 41, Харьков

9 октября 2020, 17:20

...

очень полезно. а то я всегда ломал голову- где надо оставлять геттеры/сеттеры. а оказывается, сразу после конструктора

Ответить

-

0

+

ram0973

Уровень 41, Набережные Челны, Россия

7 марта 2020, 19:30

...

К этой статье можно приложить список литературы. Я советую Стив Макконнелл - Совершенный код 2е издание. Ну и Мартин и Фаулер само собой

Ответить

-

+2

+

ram0973

Уровень 41, Набережные Челны, Россия

7 марта 2020, 19:29

...

Насчёт try/catch бы более подробный пример, и try with resources, он сразу увеличивает вложенность на 1. То же самое с DTO.

Ответить

+1

Андрей

Уровень 20, Винница, Украина

4 марта 2020, 12:50

...

Очень хорошая статья - вместо тысячи слов и десятка книг. Нашел свое - копай глубже! Ничего лишнего!

Ответить

0

Aleksey JavaWin

Уровень 15

4 марта 2020, 07:53

...

Привет, а мне статья понравилась. Не скажу, что особо зацепила, но мне как начинающему свой путь в разработке, она напомнила, что важно писать чистый код, а также принципы SOLID, и 4 правила проектирования простой архитектуры согласно Кенту Беку(такое впервые слышу, можно ссылочку на оригинал?).

Ответить

0

Nick

Уровень 31, Владивосток, Россия

4 марта 2020, 05:31

...

А кто целевая аудитория данной статьи? Как я понял, вместо "вгрызаться в книги типа Clean Code" начинающим товарищам предлагается конспект без подводок и примеров со смищными картинками (а по другому и никак в данном формате).

При выполнении любой задачи стараюсь задавать себе 3 вопроса:
1) Кому это надо?
2) Зачем это надо?
3) Что они будут с этим делать?

Соответственно, если автор писал этот конспектик для себя, то бесспорно красаучиг=)

Ответить

0

Koval

Salesforce Developer в success-craft

17 июля 2020, 14:55

...

по-моему неплохие рекомендации, я для себя несколько моментов вынес и теперь буду писать по-другому

Ответить

0

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

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

▼

