Czysty kod w języku Java

Języki Programowania Wysokiego Poziomu 2021

Michał Forystek

13.04.2021r.

Plan Prezentacji

I Po co zajmować się czystym kodem?
II Nazwy
III Funkcje
IV Komentarze
V Formatowanie
VI Testy jednostkowe
VII Klasy

Po co zajmować się czystym kodem?

Czysty kod:

- poprawia czytelność kodu - więcej czytamy niż piszemy
- ułatwia pracę w grupie
- zapobiega kaskadowym zmianom
- ułatwia testowanie
- ułatwia szukanie i naprawę błędów
- ułatwia rozbudowę projektu

Podsumowując, czysty kod **przyśpiesza pracę.**

Nadawanie nazw

Nadawanie nazw

Nazywanie jest jedną z częstszych czynności podczas programowania, dlatego ważne jest aby robić to dobrze.

Nazwy muszą przedstawiać intencje

Nadawanie nazw 1

- Muszą mówić czym coś jest, dlaczego jest i co robi
- Jeśli wymagają komentarza to nie pokazują intencji

Nazwy muszą przedstawiać intencje

Nadawanie nazw 1

```
int t;
```

VS

```
int deliveryTimeInMinutes;
int timeSinceMidnightInSeconds;
```

lub

```
public int count(List<Account> list) { ... }
```

VS

```
public int countActiveAccounts(List<Account> listOfAccounts) { ... }
```

Unikamy dezinformacji

Nadawanie nazw 2

- Nie nazywamy rzeczy programistycznym słownictwem jeżeli może to wprowadzić w błąd np.:
  Map<String, String> listOfSomething ;
- Unikamy mało różniących się nazw np.:
  XYZControllerForEfficientHandlingOfStrings
  XYZControllerForEfficientStorageOfStrings
- Nie używamy małego L oraz dużego O jako nazw

Tworzenie nazw które da się wymówić

Nadawanie nazw 3

Prościej jest dyskutować o kodzie kiedy da się normalnie wymawiać nazwy. Ludzie dobrze operują na słowach. np.:

```
private Date genymdhms;

VS

private Date generationTimestamp;
```

Tworzenie nazw łatwych do wyszukania

Nadawanie nazw 4

Często wyszukujemy zmienne, funkcje itp.

- Korzystamy ze stałych "final" zamiast "magicznych liczb"

  np.: `private final int DAYS_IN_THE_WEEK;` zamiast `7`
- Dłuższe nazwy łatwiej jest wyszukać
- Krótkie nazwy są okej jeżeli występują lokalnie
- i czy j jako iterator są dobre - tradycja

Nazywamy dla innych a nie tylko dla siebie

Nadawanie nazw 5

Nadajemy nazwy które niosą znaczenie, nawet jeśli sami umiemy sobie z nimi poradzić.

```
np.: un -> nazwa użytkownika zawierająca
          2 małe litery na początku,
          znak specjalny w środku
          i 8 cyfr na końcu
```

Nazwy klas i metod

Nadawanie nazw 6

- Nazwy klas powinny być rzeczownikami np.: Account, Figure itp.
- Nazwy metod powinny być czasownikami np. startTheEngine() itp.
- W nazwach metod typu getter i setter należy używać przedrostków get, set, is jako ogólno przyjętej konwencji
- Gdy konstruktor jest przeciążony warto stosować metody fabryk opisujące argumenty konstruktora np.:

```
Complex point = Complex.FromRealNumber(23.0);

vs

Complex point = new Complex(23.0);
```

- Przy stosowaniu powyższej techniki trzeba pamiętać o ustawieniu konstruktora na prywatny

Unikamy nazw żartobliwych

Nadawanie nazw 7

- Żarty w nazwach wprowadzają niejednoznaczność i nieoczywistość np.:

```
castItIntoTheFire();

vs

sendRequestForShutdown();
```

- To samo tyczy się slangu albo lokalnego nazewnictwa

Nazwy spójne i nazwy rozróżnialne

Nadawanie nazw 8

- Metody robiące analogiczne rzeczy nazywamy za pomocą spójnego słownictwa np.: get / fetch
- Dla metod wykonujących różne rzeczy wybieramy różne słownictwo np.: add(dodawanie), append(dodawanie do kolekcji)

Funkcje

Funkcje

Funkcje są jedną z podstawowych struktur programu, dlatego ważne jest aby pisać je w dobry sposób

Funkcje powinny być małe

Empirycznie stwierdzono, że krótkie funkcje nie zawierające instrukcji zagnieżdżonych są lepsze

Funkcje powinny robić jedną rzecz na tym samym poziomie abastrakcji

- Funkcje powinny wykonywać jedną operację. Powinny robić to dobrze. Powinny robić tylko to.
- Funkcje powinny robić tylko rzeczy na poziomie abstrakcji o jeden niższym od ich nazwy np.:

```
public void printResultToConsole(Result result){
    System.out.println("Result " + result.getName());
    System.out.println("Data1 " + result.getData1());
    System.out.println("Data2 " + result.getData2());
}
```

- Jeśli funkcję da się podzielić na segmenty to znaczy że robi więcej niż jedną rzecz

Czytanie kodu od góry do dołu

Funkcje 3

- Pisanie kodu jak artykułu w gazecie pomaga w utrzymaniu jednego poziomu abstrakcji
- Piszemy od ogółu do szczegółu

Korzystanie z opisowych nazw

Funkcje 4

Dłuższe opisowe nazwy pozwalają szybciej zrozumieć co robi dana funkcja. Funkcjom krótkim, robiącym jedną rzecz łatwiej jest nadać opisową nazwę.

Argumenty

Funkcje 5

- Im mniej argumentów tym lepiej - mniejsza szansa, że coś się zepsuje
- 0 lub 1 argument - dobrze
- 2 lub 3 - okej
- >3 - unikamy
- Więcej argumentów utrudnia pisanie testów

Częste funkcje jednoargumentowe

Funkcje 5 Argumenty 1

Częste typy funkcji jednoargumentowych:

- Pytanie o element np.: boolean fileExists("MyFile.txt")
- Przekształcenie argumentu w coś innego i zwrócenie wyniku np.: InputStream fileOpen("NazwaPliku")

Argumenty typu boolean

Staramy się nie używać zmiennych typu boolean jako argumentów. Powodują one z definicji, że funkcja robi więcej niż jedną rzecz.

Funkcje dwuargumentowe

Funkcje 5 Argumenty 3

Dwa argumenty są dobre, gdy tworzą oczywistą całość
np.: Point(7.8, 2.9)

Funkcje wieloargumentowe

Funkcje 5 Argumenty 4

- Unikamy funkcji trzyargumentowych, aczkolwiek czasem nie da się inaczej
- Za dużo argumentów może sugerować potrzebę opakowania ich w obiekt
- Wiele podobnych argumentów traktujemy jako jeden argument np.: String.format()

Unikanie efektów ubocznych

Funkcje 6

- Funkcje powinny robić tylko to co jest określone w ich nazwie
- Unikamy używania argumentów wyjściowych
- Zamiast tego używamy this
- Funkcja powinna zmieniać co najwyżej obiekt którego jest częścią

Rozdzielenie poleceń i zapytań

Funkcje 7

- Funkcja powinna robić jedną rzecz, więc nie powinna jednocześnie odpowiadać na pytanie i coś wykonywać
- Lepiej jest napisać dwie osobne funkcje np.:

```
public boolean set(User user) { ... }
if (set(user1)) { ... }

VS

if (userExist(user1)) {
    setUser(user1);
    ...
}
```

Stosujemy wyjątki zamiast kodów błędów

Funkcje 8

- Wyjątki łatwo się obsługuje
- Obsługa wyjątku może dziać się w innym miejscu niż wyjątek się pojawił
- Warto wydzielać osobne funkcje mające tylko blok try-catch-finally i zajmujące się jedynie obsługą błędów
- Enumy z kodami błędów generują zależności

DRY - Don't Repeat Yourself

Funkcje 9

- Jest to jedna z najważniejszych zasad w programowaniu
- Powtarzający się kod to więcej okazji do błędów
- Powtarzanie kodu to marnowanie czasu
- Funkcje, klasy i programowanie obiektowe to między innymi niektóre ze sposobów zmniejszenia powtarzającego się kodu

Komentarze

Komentarze

- Dobrze napisane są bardzo pomocne
- Źle napisane robią zamieszanie w kodzie
- Dobry kod zwykle nie wymaga komentarzy
- Komentarze łatwo się przeterminowują

Niektóre przypadki dobrych komentarzy

Komentarze 1

- Komentarze prawne np. o prawach autorskich
- Ostrzeżenie o konsekwencjach uruchomienia kodu
- Komentarze TODO
- Wyróżnienie niepozornych, ale bardzo ważnych fragmentów kodu
- Komentarze Javadoc przy pisaniu publicznych API

Niektóre przypadki złych komentarzy

Komentarze 2

- Komentarze nieprzemyślane lub nieprecyzyjne
- Komentarze nielokalne
- Komentowanie samokomentującego się kodu
- Komentowanie długich funkcji
- Komentowanie źle nazwanych zmiennych lub funkcji
- Zakomentowany kod

Niektóre przypadki złych komentarzy

Komentarze 2

```
//Delivery time in minutes
int t;

VS

int deliveryTimeInMinutes;
```

Formatowanie

Formatowanie

- Główny element stanowiący o estetyce wizualnej kodu
- Sprawia że kod dobrze się czyta i łatwiej jest go zrozumieć
- **Formatowanie musi być spójne w całym projekcie**
- Przy pisaniu projektów należy się podporządkować decyzji większości

Formatowanie pionowe

Formatowanie 1

- Metafora gazety po raz kolejny
- Powiązane fragmenty kodu układamy blisko siebie
- Funkcje wywołujące nad wywoływanymi
- Fragmenty powiązane konceptualnie umieszczamy blisko siebie
- Zmienne lokalne deklarujemy jak najbliżej miejsca użycia
- Zmienne instancyjne w jednym widocznym miejscu

Formatowanie poziome

Formatowanie 2

- Spacjami oddzielamy mniej powiązane rzeczy
  np.: x = 5;
- Bardziej powiązane zostawiamy bez spacji
  np.: void funkcja(int jeden, int dwa) { ... }
- Za pomocą spacji można zaznaczać kolejność działań
  np.: 2*5 + 3/7
- Wcięcia informują o poziomach zagnieżdżenia instrukcji w kodzie i nadają mu hierarchiczność

Formatowanie - przykład

```
public class SomeClass{private String name;private String description;public SomeClass(String name,String description){this.name=name;this.description=description;}public void setNameAndDescription(String newName,String newDescription){this.name=newName;this.description=newDescription;}public String getName(){return this.name;}public String getDescription(){return this.description;}}
```

Formatowanie - przykład

```
public class SomeClass {
    private String name;
    private String description;

    public SomeClass(String name, String description) {
        this.name = name;
        this.description = description;
    }

    public void setNameAndDescription(String newName, String newDescription) {
        this.name = newName;
        this.description = newDescription;
    }

    public String getName() {
        return this.name;
    }

    public String getDescription() {
        return this.description;
    }
}
```

Testy jednostkowe

TDD - Test Driven Development

1. Nie piszemy kodu produkcyjnego, jeśli nie mamy niespełnionego testu.
2. Piszemy testy tylko do momentu napisania niespełnianego testu. Brak kompilacji również się liczy.
3. Piszemy tylko tyle kodu produkcyjnego ile trzeba by spełnić niepełniany test.

1. You may not write production code until you have written a failing unit test
2. You may not write more of a unit test than is sufficient to fail, and not compiling is failing
3. You may not write more production code than is sufficient to pass the currently failing test.

Po co pisać testy?

Testy jednostkowe 1

- Pozwalają w łatwy i szybki sposób stwierdzić czy kod działa
- Pozwalają bezpiecznie zmieniać i rozwijać kod
- Dobre testy dają pewność że nic nie zepsujemy

Jak pisać testy?

Testy jednostkowe 2

- Dbamy o czystość testów tak samo jak o kod produkcyjny (wyjątki to moc obliczeniowa i pamięć)
- Wzorzec BUILD-OPERATE-CHECK
- API do testów
- Możliwie mało asercji w teście
- Jedna koncepcja na test

F.I.R.S.T.

Testy jednostkowe 3

- Fast - wolnych testów nie chce nam się uruchamiać
- Independent - niezależne od innych testów
- Repeatable - możliwe do włączenia w każdym środowisku
- Self-validating - test mówi tylko czy coś się udało czy nie
- Timely - pisane bezpośrednio przed testowanym kodem

Klasy

Organizacja elementów w klasie

Klasy 1

- Stałe statyczne
  np.: public static final int MONTHS_IN_YEAR = 12;
- Zmienne statyczne
  np.: private static int amountOfInstances;
- Zmienne instancyjne
  np.: private String name;
- Metody publiczne i prywatne

Rozluźnianie zasad hermetyzacji traktujemy jako ostateczność.

Klasy powinny być małe

Klasy 2

- SRP - Single Responsibility Principle
- Spójność klasy
- Wysoka spójność prowadzi do powstawania małych klas

Ułatwianie zmian

Klasy 3

- Posiadanie wielu małych klas pomaga we wprowadzaniu zmian
- Rozdzielenie klas za pomocą dziedziczenia
- Używanie interfejsów i klas abstrakcyjnych prowadzi do luźnych powiązań między klasami

```
HashMap<String, String> map;

VS

Map<String, String> map;
```

Bibliografia

 Robert C. Martin, "Czysty kod. Podręcznik dobrego programisty", *Helion*, 2015, ISBN: 978-83-283-1399-6.

Koniec części teoretycznej

Dziękuję za uwagę