helion.pl

Idź do

- Spis treści
- Przykładowy rozdział

Katalog książek

- Katalog online
- Zamów drukowany katalog

Twój koszyk

- Dodaj do koszyka

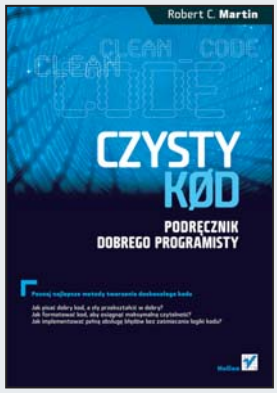
Cennik i informacje

- Zamów informacje o nowościach
- Zamów cennik

Czytelnia

- Fragmenty książek online

Czysty kod. Podręcznik dobrego programisty



Autor: Robert C. Martin
Tłumaczenie: Paweł Gonera
ISBN: 978-83-246-2188-0
Tytuł oryginału: Clean Code: A Handbook of Agile Software Craftsmanship
Format: 168×237, stron: 424

Poznaj najlepsze metody tworzenia doskonałego kodu

- Jak pisać dobry kod, a zły przekształcić w dobry?
- Jak formatować kod, aby osiągnąć maksymalną czytelność?
- Jak implementować pełną obsługę błędów bez zaśmiecania logiki kodu?

O tym, ile problemów sprawia niedbale napisany kod, wie każdy programista. Nie wszyscy jednak wiedzą, jak napisać ten świetny, "czysty" kod i czym właściwie powinien się on charakteryzować. Co więcej - jak odróżnić dobry kod od złego? Odpowiedź na te pytania oraz sposoby tworzenia czystego, czytelnego kodu znajdziesz właśnie w tej książce. Podręcznik jest obowiązkową pozycją dla każdego, kto chce poznać techniki rzetelnego i efektywnego programowania.

W książce „Czysty kod. Podręcznik dobrego programisty" szczegółowo omówione zostały zasady, wzorce i najlepsze praktyki pisania czystego kodu. Podręcznik zawiera także kilka analiz przypadków o coraz większej złożoności, z których każda jest doskonałym ćwiczeniem porządkowania zanieczyszczonego bądź nieudanego kodu. Z tego podręcznika dowiesz się m.in., jak tworzyć dobre nazwy, obiekty i funkcje, a także jak tworzyć testy jednostkowe i korzystać z programowania sterowanego testami. Nauczysz się przekształcać kod zawierający problemy w taki, który jest solidny i efektywny.

- Nazwy klas i metod
- Funkcje i listy argumentów
- Rozdzielanie poleceń i zapytań
- Stosowanie wyjątków
- Komentarze
- Formatowanie
- Obiekty i struktury danych
- Obsługa błędów
- Testy jednostkowe
- Klasy i systemy
- Współbieżność
- Oczyszczanie kodu

Niech stworzony przez Ciebie kod imponuje czystością!

Kontakt

Helion SA
ul. Kościuszki 1c
44-100 Gliwice
tel. 032 230 98 63
e-mail: helion@helion.pl
© Helion 1991-2008

SPIS TREŚCI

Słowo wstępne 13

Wstęp 19

1. Czysty kod 23

Niech stanie się kod... 24
W poszukiwaniu doskonałego kodu... 24
Całkowity koszt bałaganu 25
Rozpoczęcie wielkiej zmiany projektu 26
Postawa 27
Największa zagadka 28
Sztuka czystego kodu? 28
Co to jest czysty kod? 28
Szkoły myślenia 34
Jesteśmy autorami 35
Zasada skautów 36
Poprzednik i zasady 36
Zakończenie 36
Bibliografia 37

2. Znaczące nazwy 39

Wstęp 39
Używaj nazw przedstawiających intencje 40
Unikanie dezinformacji 41
Tworzenie wyraźnych różnic 42
Tworzenie nazw, które można wymówić 43
Korzystanie z nazw łatwych do wyszukania 44
Unikanie kodowania 45
Notacja węgierska 45
Przedrostki składników 46
Interfejsy i implementacje 46
Unikanie odwzorowania mentalnego 47
Nazwy klas 47
Nazwy metod 47
Nie bądź dowcipny 48
Wybieraj jedno słowo na pojęcie 48
Nie twórz kalamburów! 49
Korzystanie z nazw dziedziny rozwiązania 49
Korzystanie z nazw dziedziny problemu 49
Dodanie znaczącego kontekstu 50
Nie należy dodawać nadmiarowego kontekstu 51
Słowo końcowe 52

3. Funkcje **53**
Małe funkcje! 56
Bloki i wcięcia 57
Wykonuj jedną czynność 57
Sekcje wewnątrz funkcji 58
Jeden poziom abstrakcji w funkcji 58
Czytanie kodu od góry do dołu — zasada zstępująca 58
Instrukcje switch 59
Korzystanie z nazw opisowych 61
Argumenty funkcji 62
Często stosowane funkcje jednoargumentowe 62
Argumenty znacznikowe 63
Funkcje dwuargumentowe 63
Funkcje trzyargumentowe 64
Argumenty obiektowe 64
Listy argumentów 65
Czasowniki i słowa kluczowe 65
Unikanie efektów ubocznych 65
Argumenty wyjściowe 66
Rozdzielanie poleceń i zapytań 67
Stosowanie wyjątków zamiast zwracania kodów błędów 67
Wyodrębnienie bloków try-catch 68
Obsługa błędów jest jedną operacją 69
Przyciąganie zależności w Error.java 69
Nie powtarzaj się 69
Programowanie strukturalne 70
Jak pisać takie funkcje? 70
Zakończenie 71
SetupTeardownIncluder 71
Bibliografia 73

4. Komentarze **75**
Komentarze nie są szminką dla złego kodu 77
Czytelny kod nie wymaga komentarzy 77
Dobre komentarze 77
Komentarze prawne 77
Komentarze informacyjne 78
Wyjaśnianie zamierzeń 78
Wyjaśnianie 79
Ostrzeżenia o konsekwencjach 80
Komentarze TODO 80
Wzmocnienie 81
Komentarze Javadoc w publicznym API 81
Złe komentarze 81
Bełkot 81
Powtarzające się komentarze 82
Mylące komentarze 84
Komentarze wymagane 85
Komentarze dziennika 85

Komentarze wprowadzające szum informacyjny 86
Przerażający szum 87
Nie używaj komentarzy, jeżeli można użyć funkcji lub zmiennej 88
Znaczniki pozycji 88
Komentarze w klamrach zamykających 88
Atrybuty i dopiski 89
Zakomentowany kod 89
Komentarze HTML 90
Informacje nielokalne 91
Nadmiar informacji 91
Nieoczywiste połączenia 91
Nagłówki funkcji 92
Komentarze Javadoc w niepublicznym kodzie 92
Przykład 92
Bibliografia 95

5. Formatowanie 97
Przeznaczenie formatowania 98
Formatowanie pionowe 98
Metafora gazety 99
Pionowe odstępy pomiędzy segmentami kodu 99
Gęstość pionowa 101
Odległość pionowa 101
Uporządkowanie pionowe 105
Formatowanie poziome 106
Poziome odstępy i gęstość 106
Rozmieszczenie poziome 107
Wcięcia 109
Puste zakresy 110
Zasady zespołowe 110
Zasady formatowania wujka Boba 111

6. Obiekty i struktury danych 113
Abstrakcja danych 113
Antysymetria danych i obiektów 115
Prawo Demeter 117
Wraki pociągów 118
Hybrydy 118
Ukrywanie struktury 119
Obiekty transferu danych 119
Active Record 120
Zakończenie 121
Bibliografia 121

7. Obsługa błędów 123
Użycie wyjątków zamiast kodów powrotu 124
Rozpoczynanie od pisania instrukcji try-catch-finally 125
Użycie niekontrolowanych wyjątków 126
Dostarczanie kontekstu za pomocą wyjątków 127
Definiowanie klas wyjątków w zależności od potrzeb wywołującego 127

Definiowanie normalnego przepływu 129
Nie zwracamy null 130
Nie przekazujemy null 131
Zakończenie 132
Bibliografia 132

8. Granice 133
Zastosowanie kodu innych firm 134
Przeglądanie i zapoznawanie się z granicami 136
Korzystanie z pakietu log4j 136
Zalety testów uczących 138
Korzystanie z nieistniejącego kodu 138
Czyste granice 139
Bibliografia 140

9. Testy jednostkowe 141
Trzy prawa TDD 142
Zachowanie czystości testów 143
Testy zwiększają możliwości 144
Czyste testy 144
Języki testowania specyficzne dla domeny 147
Podwójny standard 147
Jedna asercja na test 149
Jedna koncepcja na test 150
F.I.R.S.T. 151
Zakończenie 152
Bibliografia 152

10. Klasy 153
Organizacja klas 153
Hermetyzacja 154
Klasy powinny być małe! 154
Zasada pojedynczej odpowiedzialności 156
Spójność 158
Utrzymywanie spójności powoduje powstanie wielu małych klas 158
Organizowanie zmian 164
Izolowanie modułów kodu przed zmianami 166
Bibliografia 167

11. Systemy 169
Jak budowałbyś miasto? 170
Oddzielenie konstruowania systemu od jego używania 170
Wydzielenie modułu main 171
Fabryki 172
Wstrzykiwanie zależności 172
Skalowanie w górę 173
Separowanie (rozcięcie) problemów 176
Pośredniki Java 177

Czyste biblioteki Java AOP 178
Aspekty w AspectJ 181
Testowanie architektury systemu 182
Optymalizacja podejmowania decyzji 183
Korzystaj ze standardów, gdy wnoszą realną wartość 183
Systemy wymagają języków dziedzinowych 184
Zakończenie 184
Bibliografia 185

12. Powstawanie projektu 187
Uzyskiwanie czystości projektu przez jego rozwijanie 187
Zasada numer 1 prostego projektu — system przechodzi wszystkie testy 188
Zasady numer 2 – 4 prostego projektu — przebudowa 188
Brak powtórzeń 189
Wyrazistość kodu 191
Minimalne klasy i metody 192
Zakończenie 192
Bibliografia 192

13. Współbieżność 193
W jakim celu stosować współbieżność? 194
Mity i nieporozumienia 195
Wyzwania 196
Zasady obrony współbieżności 196
Zasada pojedynczej odpowiedzialności 197
Wniosek — ograniczenie zakresu danych 197
Wniosek — korzystanie z kopii danych 197
Wniosek — wątki powinny być na tyle niezależne, na ile to tylko możliwe 198
Poznaj używaną bibliotekę 198
Kolekcje bezpieczne dla wątków 198
Poznaj modele wykonania 199
Producent-konsument 199
Czytelnik-pisarz 200
Ucztujący filozofowie 200
Uwaga na zależności pomiędzy synchronizowanymi metodami 201
Tworzenie małych sekcji synchronizowanych 201
Pisanie prawidłowego kodu wyłączającego jest trudne 202
Testowanie kodu wątków 202
Traktujemy przypadkowe awarie jako potencjalne problemy z wielowątkowością 203
Na początku uruchamiamy kod niekorzystający z wątków 203
Nasz kod wątków powinien dać się włączać 203
Nasz kod wątków powinien dać się dostrajać 204
Uruchamiamy więcej wątków, niż mamy do dyspozycji procesorów 204
Uruchamiamy testy na różnych platformach 204
Uzbrajamy nasz kod w elementy próbujące wywołać awarie i wymuszające awarie 205
Instrumentacja ręczna 205
Instrumentacja automatyczna 206
Zakończenie 207
Bibliografia 208

14. Udane oczyszczanie kodu **209**
Implementacja klasy Args 210
Args — zgrubny szkic 216
Argumenty typu String 228
Zakończenie 261

15. Struktura biblioteki JUnit **263**
Biblioteka JUnit 264
Zakończenie 276

16. Przebudowa klasy SerialDate **277**
Na początek uruchamiamy 278
Teraz poprawiamy 280
Zakończenie 293
Bibliografia 294

17. *Zapachy* kodu i heurystyki **295**
Komentarze 296
C1. Niewłaściwe informacje 296
C2. Przestarzałe komentarze 296
C3. Nadmiarowe komentarze 296
C4. Źle napisane komentarze 297
C5. Zakomentowany kod 297
Środowisko 297
E1. Budowanie wymaga więcej niż jednego kroku 297
E2. Testy wymagają więcej niż jednego kroku 297
Funkcje 298
F1. Nadmiar argumentów 298
F2. Argumenty wyjściowe 298
F3. Argumenty znacznikowe 298
F4. Martwe funkcje 298
Ogólne 298
G1. Wiele języków w jednym pliku źródłowym 298
G2. Oczywiste działanie jest nieimplementowane 299
G3. Niewłaściwe działanie w warunkach granicznych 299
G4. Zdjęte zabezpieczenia 299
G5. Powtórzenia 300
G6. Kod na nieodpowiednim poziomie abstrakcji 300
G7. Klasy bazowe zależne od swoich klas pochodnych 301
G8. Za dużo informacji 302
G9. Martwy kod 302
G10. Separacja pionowa 303
G11. Niespójność 303
G12. Zaciemnianie 303
G13. Sztuczne sprzężenia 303
G14. Zazdrość o funkcje 304
G15. Argumenty wybierające 305
G16. Zaciemnianie intencji 305
G17. Źle rozmieszczona odpowiedzialność 306

G18. Niewłaściwe metody statyczne 306
G19. Użycie opisowych zmiennych 307
G20. Nazwy funkcji powinny informować o tym, co realizują 307
G21. Zrozumienie algorytmu 308
G22. Zamiana zależności logicznych na fizyczne 308
G23. Zastosowanie polimorfizmu zamiast instrukcji if-else lub switch-case 309
G24. Wykorzystanie standardowych konwencji 310
G25. Zamiana magicznych liczb na stałe nazwane 310
G26. Precyzja 311
G27. Struktura przed konwencją 312
G28. Hermetyzacja warunków 312
G29. Unikanie warunków negatywnych 312
G30. Funkcje powinny wykonywać jedną operację 312
G31. Ukryte sprzężenia czasowe 313
G32. Unikanie dowolnych działań 314
G33. Hermetyzacja warunków granicznych 314
G34. Funkcje powinny zagłębiać się na jeden poziom abstrakcji 315
G35. Przechowywanie danych konfigurowalnych na wysokim poziomie 316
G36. Unikanie nawigacji przechodnich 317
Java 317
J1. Unikanie długich list importu przez użycie znaków wieloznacznych 317
J2. Nie dziedziczymy stałych 318
J3. Stałe kontra typy wyliczeniowe 319
Nazwy 320
N1. Wybór opisowych nazw 320
N2. Wybór nazw na odpowiednich poziomach abstrakcji 321
N3. Korzystanie ze standardowej nomenklatury tam, gdzie jest to możliwe 322
N4. Jednoznaczne nazwy 322
N5. Użycie długich nazw dla długich zakresów 323
N6. Unikanie kodowania 323
N7. Nazwy powinny opisywać efekty uboczne 323
Testy 324
T1. Niewystarczające testy 324
T2. Użycie narzędzi kontroli pokrycia 324
T3. Nie pomijaj prostych testów 324
T4. Ignorowany test jest wskazaniem niejednoznaczności 324
T5. Warunki graniczne 324
T6. Dokładne testowanie pobliskich błędów 324
T7. Wzorce błędów wiele ujawniają 324
T8. Wzorce pokrycia testami wiele ujawniają 325
T9. Testy powinny być szybkie 325
Zakończenie 325
Bibliografia 325

A Współbieżność II 327

Przykład klient-serwer 327
Serwer 327
Dodajemy wątki 329
Uwagi na temat serwera 329
Zakończenie 331

Możliwe ścieżki wykonania 331
Liczba ścieżek 332
Kopiemy głębiej 333
Zakończenie 336
Poznaj używaną bibliotekę 336
Biblioteka Executor 336
Rozwiązania nieblokujące 337
Bezpieczne klasy nieobsługujące wątków 338
Zależności między metodami mogą uszkodzić kod współbieżny 339
Tolerowanie awarii 340
Blokowanie na kliencie 340
Blokowanie na serwerze 342
Zwiększanie przepustowości 343
Obliczenie przepustowości jednowątkowej 344
Obliczenie przepustowości wielowątkowej 344
Zakleszczenie 345
Wzajemne wykluczanie 346
Blokowanie i oczekiwanie 346
Brak wywłaszczania 346
Cykliczne oczekiwanie 346
Zapobieganie wzajemnemu wykluczaniu 347
Zapobieganie blokowaniu i oczekiwaniu 347
Umożliwienie wywłaszczania 348
Zapobieganie oczekiwaniu cyklicznemu 348
Testowanie kodu wielowątkowego 349
Narzędzia wspierające testowanie kodu korzystającego z wątków 351
Zakończenie 352
Samouczek. Pełny kod przykładów 352
Klient-serwer bez wątków 352
Klient-serwer z użyciem wątków 355

B org.jfree.date.SerialDate 357

C Odwołania do heurystyk 411

Epilog 413

Skorowidz 415

Komentarze

Nie komentuj złego kodu — popraw go.

Brian W. Kernighan i P.J. Plaugher[1]

NIEWIELE JEST RZECZY TAK POMOCNYCH, jak dobrze umieszczony komentarz. Jednocześnie nic tak nie zaciemnia modułu, jak kilka zbyt dogmatycznych komentarzy. Nic nie jest tak szkodliwe, jak stary komentarz szerzący kłamstwa i dezinformację.

Komentarze nie są jak „Lista Schindlera". Nie są one „czystym dobrem". W rzeczywistości komentarze są w najlepszym przypadku koniecznym złem. Jeżeli nasz język programowania jest wystarczająco ekspresyjny lub mamy wystarczający talent, by wykorzystywać ten język, aby wyrażać nasze zamierzenia, nie będziemy potrzebować zbyt wielu komentarzy.

[1] [KP78], s. 144.

Prawidłowe zastosowanie komentarzy jest kompensowaniem naszych błędów przy tworzeniu kodu. Proszę zwrócić uwagę, że użyłem słowa *błąd*. Dokładnie to miałem na myśli. Obecność komentarzy zawsze sygnalizuje nieporadność programisty. Musimy korzystać z nich, ponieważ nie zawsze wiemy, jak wyrazić nasze intencje bez ich użycia, ale ich obecność nie jest powodem do świętowania.

Gdy uznamy, że konieczne jest napisanie komentarza, należy pomyśleć, czy nie istnieje sposób na wyrażenie tego samego w kodzie. Za każdym razem, gdy wyrazimy to samo za pomocą kodu, powinniśmy odczuwać satysfakcję. Za każdym razem, gdy piszemy komentarz, powinniśmy poczuć smak porażki.

Dlaczego jestem tak przeciwny komentarzom? Ponieważ one kłamią. Nie zawsze, nie rozmyślnie, ale nader często. Im starsze są komentarze, tym większe prawdopodobieństwo, że są po prostu błędne. Powód jest prosty. Programiści nie są w stanie utrzymywać ich aktualności.

Kod zmienia się i ewoluuje. Jego fragmenty są przenoszone w różne miejsca. Fragmenty te są rozdzielane, odtwarzane i ponownie łączone. Niestety, komentarze nie zawsze za nimi podążają — nie zawsze *mogą* być przenoszone. Zbyt często komentarze są odłączane od kodu, który opisują, i stają się osieroconymi notatkami o stale zmniejszającej się dokładności. Dla przykładu warto spojrzeć, co się stało z komentarzem i wierszem, którego dotyczył:

```
MockRequest request;
private final String HTTP_DATE_REGEXP =
   "[SMTWF][a-z]{2}\\,\\s[0-9]{2}\\s[JFMASOND][a-z]{2}\\s"+
   "[0-9]{4}\\s[0-9]{2}\\:[0-9]{2}\\:[0-9]{2}\\sGMT";
private Response response;
private FitNesseContext context;
private FileResponder responder;
private Locale saveLocale;
// Przykład: "Tue, 02 Apr 2003 22:18:49 GMT"
```

Pozostałe zmienne instancyjne zostały prawdopodobnie później dodane pomiędzy stałą `HTTP_DATE_REGEXP` a objaśniającym ją komentarzem.

Można oczywiście stwierdzić, że programiści powinni być na tyle zdyscyplinowani, aby utrzymywać komentarze w należytym stanie. Zgadzam się, powinni. Wolałbym jednak, aby poświęcona na to energia została spożytkowana na zapewnienie takiej precyzji i wyrazistości kodu, by komentarze okazały się zbędne.

Niedokładne komentarze są znacznie gorsze niż ich brak. Kłamią i wprowadzają w błąd. Powodują powstanie oczekiwań, które nigdy nie są spełnione. Definiują stare zasady, które nie są już potrzebne lub nie powinny być stosowane.

Prawda znajduje się w jednym miejscu: w kodzie. Jedynie kod może niezawodnie przedstawić to, co realizuje. Jest jedynym źródłem naprawdę dokładnych informacji. Dlatego choć komentarze są czasami niezbędne, poświęcimy sporą ilość energii na zminimalizowanie ich liczby.

Komentarze nie są szminką dla złego kodu

Jednym z często spotykanych powodów pisania komentarzy jest nieudany kod. Napisaliśmy moduł i zauważamy, że jest źle zorganizowany. Wiemy, że jest chaotyczny. Mówimy wówczas: „Hm, będzie lepiej, jak go skomentuję". Nie! Lepiej go poprawić!

Precyzyjny i czytelny kod z małą liczbą komentarzy jest o wiele lepszy niż zabałaganiony i złożony kod z mnóstwem komentarzy. Zamiast spędzać czas na pisaniu kodu wyjaśniającego bałagan, jaki zrobiliśmy, warto poświęcić czas na posprzątanie tego bałaganu.

Czytelny kod nie wymaga komentarzy

W wielu przypadkach kod mógłby zupełnie obejść się bez komentarzy, a jednak programiści wolą umieścić w nim komentarz, zamiast zawrzeć objaśnienia w samym kodzie. Spójrzmy na poniższy przykład. Co wolelibyśmy zobaczyć? To:

```
// Sprawdzenie, czy pracownik ma prawo do wszystkich korzyści
if ((employee.flags & HOURLY_FLAG) && (employee.age > 65))
```

czy to:

```
if (employee.isEligibleForFullBenefits())
```

Przeznaczenie tego kodu jest jasne po kilku sekundach myślenia. W wielu przypadkach jest to wyłącznie kwestia utworzenia funkcji, która wyraża to samo co komentarz, jaki chcemy napisać.

Dobre komentarze

Czasami komentarze są niezbędne lub bardzo przydatne. Przedstawimy kilka przypadków, w których uznaliśmy, że warto poświęcić im czas. Należy jednak pamiętać, że naprawdę dobry komentarz to taki, dla którego znaleźliśmy powód, aby go nie pisać.

Komentarze prawne

Korporacyjne standardy kodowania czasami wymuszają na nas pisanie pewnych komentarzy z powodów prawnych. Na przykład informacje o prawach autorskich są niezbędnym elementem umieszczanym w komentarzu na początku każdego pliku źródłowego.

Przykładem może być standardowy komentarz, jaki umieszczaliśmy na początku każdego pliku źródłowego w FitNesse. Na szczęście nasze środowisko IDE ukrywa te komentarze przez ich automatyczne zwinięcie.

```
// Copyright (C) 2003,2004,2005 by Object Mentor, Inc. All rights reserved.
// Released under the terms of the GNU General Public License version 2 or later.
```

Tego typu komentarze nie powinny być wielkości umów lub kodeksów. Tam, gdzie to możliwe, warto odwoływać się do standardowych licencji lub zewnętrznych dokumentów, a nie umieszczać w komentarzu wszystkich zasad i warunków.

Komentarze informacyjne

Czasami przydatne jest umieszczenie w komentarzu podstawowych informacji. Na przykład w poniższym komentarzu objaśniamy wartość zwracaną przez metodę abstrakcyjną.

```
// Zwraca testowany obiekt Responder.
protected abstract Responder responderInstance();
```

Komentarze tego typu są czasami przydatne, ale tam, gdzie to możliwe, lepiej jest skorzystać z nazwy funkcji do przekazania informacji. Na przykład w tym przypadku komentarz może stać się niepotrzebny, jeżeli zmienimy nazwę funkcji: `responderBeingTested`.

Poniżej mamy nieco lepszy przypadek:

```
// Dopasowywany format kk:mm:ss EEE, MMM dd, yyyy
Pattern timeMatcher = Pattern.compile(
   "\\d*:\\d*:\\d* \\w*, \\w* \\d*, \\d*");
```

W tym przypadku komentarze pozwalają nam poinformować, że użyte wyrażenie regularne ma dopasować czas i datę sformatowane za pomocą funkcji `SimpleDateFormat.format` z użyciem zdefiniowanego formatu. Nadal lepiej jest przenieść kod do specjalnej klasy pozwalającej na konwertowanie formatów daty i czasu. Po tej operacji komentarz najprawdopodobniej stanie się zbędny.

Wyjaśnianie zamierzeń

W niektórych przypadkach komentarze zawierają informacje nie tylko o implementacji, ale także o powodach podjęcia danej decyzji. W poniższym przypadku widzimy interesującą decyzję udokumentowaną w postaci komentarza. Przy porównywaniu obiektów autor zdecydował o tym, że obiekty jego klasy będą po posortowaniu wyżej niż obiekty pozostałych klas.

```
public int compareTo(Object o)
{
   if(o instanceof WikiPagePath)
   {
      WikiPagePath p = (WikiPagePath) o;
      String compressedName = StringUtil.join(names, "");
      String compressedArgumentName = StringUtil.join(p.names, "");
      return compressedName.compareTo(compressedArgumentName);
   }
   return 1; // Jesteśmy więksi, ponieważ jesteśmy właściwego typu.
}
```

Poniżej pokazany jest lepszy przykład. Możemy nie zgadzać się z rozwiązaniem tego problemu przez programistę, ale przynajmniej wiemy, co próbował zrobić.

```
public void testConcurrentAddWidgets() throws Exception {
   WidgetBuilder widgetBuilder =
   new WidgetBuilder(new Class[]{BoldWidget.class});
   String text = "'''bold text'''";
```

```
    ParentWidget parent =
       new BoldWidget(new MockWidgetRoot(), "'''bold text'''");
    AtomicBoolean failFlag = new AtomicBoolean();
    failFlag.set(false);

    //Jest to nasza próba uzyskania wyścigu
    //przez utworzenie dużej liczby wątków.
    for (int i = 0; i < 25000; i++) {
       WidgetBuilderThread widgetBuilderThread =
       new WidgetBuilderThread(widgetBuilder, text, parent, failFlag);
       Thread thread = new Thread(widgetBuilderThread);
       thread.start();
    }
    assertEquals(false, failFlag.get());
}
```

Wyjaśnianie

Czasami przydatne jest wytłumaczenie znaczenia niejasnych argumentów lub zwracanych wartości. Zwykle lepiej jest znaleźć sposób na to, by ten argument lub zwracana wartość były bardziej czytelne, ale jeżeli są one częścią biblioteki standardowej lub kodu, którego nie możemy zmieniać, to wyjaśnienia w komentarzach mogą być użyteczne.

```
public void testCompareTo() throws Exception
{
   WikiPagePath a = PathParser.parse("PageA");
   WikiPagePath ab = PathParser.parse("PageA.PageB");
   WikiPagePath b = PathParser.parse("PageB");
   WikiPagePath aa = PathParser.parse("PageA.PageA");
   WikiPagePath bb = PathParser.parse("PageB.PageB");
   WikiPagePath ba = PathParser.parse("PageB.PageA");

   assertTrue(a.compareTo(a) == 0); // a == a
   assertTrue(a.compareTo(b) != 0); // a != b
   assertTrue(ab.compareTo(ab) == 0); // ab == ab
   assertTrue(a.compareTo(b) == -1); // a < b
   assertTrue(aa.compareTo(ab) == -1); // aa < ab
   assertTrue(ba.compareTo(bb) == -1); // ba < bb
   assertTrue(b.compareTo(a) == 1); // b > a
   assertTrue(ab.compareTo(aa) == 1); // ab > aa
   assertTrue(bb.compareTo(ba) == 1); // bb > ba
}
```

Istnieje oczywiście spore ryzyko, że komentarze objaśniające są nieprawidłowe. Warto przeanalizować poprzedni przykład i zobaczyć, jak trudno jest sprawdzić, czy są one prawidłowe. Wyjaśnia to, dlaczego niezbędne są objaśnienia i dlaczego są one ryzykowne. Tak więc przed napisaniem tego typu komentarzy należy sprawdzić, czy nie istnieje lepszy sposób, a następnie poświęcić im więcej uwagi, aby były precyzyjne.

Ostrzeżenia o konsekwencjach

Komentarze mogą również służyć do ostrzegania innych programistów o określonych konsekwencjach. Poniższy komentarz wyjaśnia, dlaczego przypadek testowy jest wyłączony:

```
// Nie uruchamiaj, chyba że masz nieco czasu do zagospodarowania.
public void _testWithReallyBigFile()
{
   writeLinesToFile(10000000);
   response.setBody(testFile);
   response.readyToSend(this);
   String responseString = output.toString();
   assertSubString("Content-Length:
   ↪1000000000", responseString);
   assertTrue(bytesSent > 1000000000);
}
```

Obecnie oczywiście wyłączamy przypadek testowy przez użycie atrybutu `@Ignore` z odpowiednim tekstem wyjaśniającym. `@Ignore("Zajmuje zbyt dużo czasu")`. Jednak w czasach przed JUnit 4 umieszczenie podkreślenia przed nazwą metody było często stosowaną konwencją. Komentarz, choć nonszalancki, dosyć dobrze wskazuje powód.

Poniżej pokazany jest inny przykład:

```
public static SimpleDateFormat makeStandardHttpDateFormat()
{
   //SimpleDateFormat nie jest bezpieczna dla wątków,
   //więc musimy każdy obiekt tworzyć niezależnie.
   SimpleDateFormat df = new SimpleDateFormat("EEE, dd MMM yyyy HH:mm:ss z");
   df.setTimeZone(TimeZone.getTimeZone("GMT"));
   return df;
}
```

Można narzekać, że istnieją lepsze sposoby rozwiązania tego problemu. Mogę się z tym zgodzić. Jednak zastosowany tu komentarz jest całkiem rozsądny. Może on powstrzymać nadgorliwego programistę przed użyciem statycznego inicjalizera dla zapewnienia lepszej wydajności.

Komentarze TODO

Czasami dobrym pomysłem jest pozostawianie notatek „do zrobienia” w postaci komentarzy `//TODO`. W zamieszczonym poniżej przypadku komentarz `TODO` wyjaśnia, dlaczego funkcja ma zdegenerowaną implementację i jaka powinna być jej przyszłość.

```
//TODO-MdM Nie jest potrzebna.
// Oczekujemy, że zostanie usunięta po pobraniu modelu.
protected VersionInfo makeVersion() throws Exception
{
   return null;
}
```

Komentarze TODO oznaczają zadania, które według programisty powinny być wykonane, ale z pewnego powodu nie można tego zrobić od razu. Może to być przypomnienie o konieczności usunięcia przestarzałej funkcji lub prośba do innej osoby o zajęcie się problemem. Może to być żądanie, aby ktoś pomyślał o nadaniu lepszej nazwy, lub przypomnienie o konieczności wprowadzenia zmiany zależnej od planowanego zdarzenia. Niezależnie od tego, czym jest TODO, *nie może* to być wymówka dla pozostawienia złego kodu w systemie.

Obecnie wiele dobrych IDE zapewnia specjalne funkcje lokalizujące wszystkie komentarze TODO, więc jest mało prawdopodobne, aby zostały zgubione. Nadal jednak nie jest korzystne, by kod był nafaszerowany komentarzami TODO. Należy więc regularnie je przeglądać i eliminować wszystkie, które się da.

Wzmocnienie

Komentarz może być użyty do wzmocnienia wagi operacji, która w przeciwnym razie może wydawać się niekonsekwencją.

```
String listItemContent = match.group(3).trim();
// Wywołanie trim jest naprawdę ważne. Usuwa początkowe
// spacje, które mogą spowodować, że element będzie
// rozpoznany jako kolejna lista.
new ListItemWidget(this, listItemContent, this.level + 1);
return buildList(text.substring(match.end()));
```

Komentarze Javadoc w publicznym API

Nie ma nic bardziej pomocnego i satysfakcjonującego, jak dobrze opisane publiczne API. Przykładem tego może być standardowa biblioteka Java. Bez niej pisanie programów Java byłoby trudne, o ile nie niemożliwe.

Jeżeli piszemy publiczne API, to niezbędne jest napisanie dla niego dobrej dokumentacji Javadoc. Jednak należy pamiętać o pozostałych poradach z tego rozdziału. Komentarze Javadoc mogą być równie mylące, nie na miejscu i nieszczere jak wszystkie inne komentarze.

Złe komentarze

Do tej kategorii należy większość komentarzy. Zwykle są to podpory złego kodu lub wymówki albo uzasadnienie niewystarczających decyzji znaczące niewiele więcej niż dyskusja programisty ze sobą.

Bełkot

Pisanie komentarza tylko dlatego, że czujemy, iż powinien być napisany lub też że wymaga tego proces, jest błędem. Jeżeli decydujemy się na napisanie komentarza, musimy poświęcić nieco czasu na upewnienie się, że jest to najlepszy komentarz, jaki mogliśmy napisać.

Poniżej zamieszczony jest przykład znaleziony w FitNesse. Komentarz był faktycznie przydatny. Jednak autor śpieszył się lub nie poświęcił mu zbyt wiele uwagi. Bełkot, który po sobie zostawił, stanowi nie lada zagadkę:

```
public void loadProperties()
{
   try
   {
      String propertiesPath = propertiesLocation + "/" + PROPERTIES_FILE;
      FileInputStream propertiesStream = new FileInputStream(propertiesPath);
      loadedProperties.load(propertiesStream);
   }
   catch(IOException e)
   {
      // Brak plików właściwości oznacza załadowanie wszystkich wartości domyślnych.
   }
}
```

Co oznacza komentarz w bloku `catch`? Jasne jest, że znaczy on coś dla autora, ale znaczenie to nie zostało dobrze wyartykułowane. Jeżeli otrzymamy wyjątek `IOException`, najwyraźniej oznacza to brak pliku właściwości, a w takim przypadku ładowane są wszystkie wartości domyślne. Jednak kto ładuje te wartości domyślne? Czy były załadowane przed wywołaniem `loadProperties.load`? Czy też `loadProperties.load` przechwytuje wyjątek, ładuje wartości domyślne i przekazuje nam wyjątek do zignorowania? A może `loadProperties.load` ładuje wszystkie wartości domyślne przed próbą załadowania pliku? Czy autor próbował usprawiedliwić przed samym sobą fakt, że pozostawił pusty blok `catch`? Być może — ta możliwość jest nieco przerażająca — autor próbował powiedzieć sobie, że powinien wrócić w to miejsce i napisać kod ładujący wartości domyślne.

Jedynym sposobem, aby się tego dowiedzieć, jest przeanalizowanie kodu z innych części systemu i sprawdzenie, co się w nich dzieje. Wszystkie komentarze, które wymuszają zaglądanie do innych modułów w celu ich zrozumienia, nie są warte bitów, które zajmują.

Powtarzające się komentarze

Na listingu 4.1 zamieszczona jest prosta funkcja z komentarzem w nagłówku, który jest całkowicie zbędny. Prawdopodobnie dłużej zajmuje przeczytanie komentarza niż samego kodu.

LISTING 4.1. waitForClose

```
// Metoda użytkowa kończąca pracę, gdy this.closed ma wartość true. Zgłasza wyjątek,
// jeżeli przekroczony zostanie czas oczekiwania.
public synchronized void waitForClose(final long timeoutMillis)
  throws Exception
{
  if(!closed)
  {
    wait(timeoutMillis);
    if(!closed)
      throw new Exception("MockResponseSender could not be closed");
  }
}
```

Czemu służy ten komentarz? Przecież nie niesie więcej informacji niż sam kod. Nie uzasadnia on kodu, nie przedstawia zamierzeń ani przyczyn. Nie jest łatwiejszy do czytania od samego kodu.

W rzeczywistości jest mniej precyzyjny niż kod i wymusza na czytelniku zaakceptowanie braku precyzji w imię prawdziwego zrozumienia. Jest on podobny do paplania sprzedawcy używanych samochodów, który zapewnia, że nie musisz zaglądać pod maskę.

Spójrzmy teraz na legion bezużytecznych i nadmiarowych komentarzy Javadoc pobranych z programu Tomcat i zamieszczonych na listingu 4.2. Komentarze te mają za zadanie wyłącznie zaciemnić i popsuć kod. Nie mają one żadnej wartości dokumentującej. Co gorsza, pokazałem tutaj tylko kilka pierwszych. W tym module znajduje się znacznie więcej takich komentarzy.

LISTING 4.2. ContainerBase.java (Tomcat)

```
public abstract class ContainerBase
  implements Container, Lifecycle, Pipeline,
  MBeanRegistration, Serializable {

  /**
   * The processor delay for this component.
   */
  protected int backgroundProcessorDelay = -1;

  /**
   * The lifecycle event support for this component.
   */
  protected LifecycleSupport lifecycle =
    new LifecycleSupport(this);

  /**
   * The container event listeners for this Container.
   */
  protected ArrayList listeners = new ArrayList();

  /**
   * The Loader implementation with which this Container is
   * associated.
   */
  protected Loader loader = null;

  /**
   * The Logger implementation with which this Container is
   * associated.
   */
  protected Log logger = null;

  /**
   * Associated logger name.
   */
  protected String logName = null;

  /**
   * The Manager implementation with which this Container is
   * associated.
   */
  protected Manager manager = null;

  /**
   * The cluster with which this Container is associated.
   */
  protected Cluster cluster = null;
```

```
/**
 * The human-readable name of this Container.
 */
protected String name = null;

/**
 * The parent Container to which this Container is a child.
 */
protected Container parent = null;

/**
 * The parent class loader to be configured when we install a
 * Loader.
 */
protected ClassLoader parentClassLoader = null;

/**
 * The Pipeline object with which this Container is
 * associated.
 */
protected Pipeline pipeline = new StandardPipeline(this);

/**
 * The Realm with which this Container is associated.
 */
protected Realm realm = null;

/**
 * The resources DirContext object with which this Container
 * is associated.
 */
protected DirContext resources = null;
```

Mylące komentarze

Czasami pomimo najlepszych intencji programista zapisuje w komentarzu nieprecyzyjne zdania. Wróćmy na moment do nadmiarowego, ale również nieco mylącego komentarza zamieszczonego na listingu 4.1.

Czy Czytelnik zauważył, w czym ten komentarz jest mylący? Metoda ta nie kończy się, *gdy* `this.closed` ma wartość `true`. Kończy się ona, *jeżeli* `this.closed` ma wartość `true`; w przeciwnym razie czeka określony czas, a następnie zgłasza wyjątek, *jeżeli* `this.closed` nadal nie ma wartości `true`.

Ta subtelna dezinformacja umieszczona w komentarzu, który czyta się trudniej niż sam kod, może spowodować, że inny programista naiwnie wywoła tę funkcję, oczekując, że zakończy się od razu, gdy `this.closed` przyjmie wartość `true`. Ten biedny programista może zorientować się, o co chodzi, dopiero w sesji debugera, gdy będzie próbował zorientować się, dlaczego jego kod działa tak powoli.

Komentarze wymagane

Wymaganie, aby każda funkcja posiadała Javadoc lub aby każda zmienna posiadała komentarz, jest po prostu głupie. Tego typu komentarze tylko zaciemniają kod i prowadzą do powszechnych pomyłek i dezorganizacji.

Na przykład wymaganie komentarza Javadoc prowadzi do powstania takich potworów, jak ten zamieszczony na listingu 4.3. Takie komentarze nie wnoszą niczego, za to utrudniają zrozumienie kodu.

LISTING 4.3.

```
/**
 *
 * @param title Tytuł płyty CD
 * @param author Autor płyty CD
 * @param tracks Liczba ścieżek na płycie CD
 * @param durationInMinutes Czas odtwarzania CD w minutach
 */
public void addCD(String title, String author,
                  int tracks, int durationInMinutes) {
  CD cd = new CD();
  cd.title = title;
  cd.author = author;
  cd.tracks = tracks;
  cd.duration = duration;
  cdList.add(cd);
}
```

Komentarze dziennika

Czasami programiści dodają na początku każdego pliku komentarz informujący o każdej edycji. Komentarze takie tworzą pewnego rodzaju dziennik wszystkich wprowadzonych zmian. Spotkałem się z modułami zawierającymi kilkanaście stron z kolejnymi pozycjami dziennika.

```
* Changes (from 11-Oct-2001)
* --------------------------
* 11-Oct-2001 : Re-organised the class and moved it to new package
* com.jrefinery.date (DG);
* 05-Nov-2001 : Added a getDescription() method, and eliminated NotableDate
* class (DG);
* 12-Nov-2001 : IBD requires setDescription() method, now that NotableDate
* class is gone (DG); Changed getPreviousDayOfWeek(),
* getFollowingDayOfWeek() and getNearestDayOfWeek() to correct
* bugs (DG);
* 05-Dec-2001 : Fixed bug in SpreadsheetDate class (DG);
* 29-May-2002 : Moved the month constants into a separate interface
* (MonthConstants) (DG);
* 27-Aug-2002 : Fixed bug in addMonths() method, thanks to N???levka Petr (DG);
* 03-Oct-2002 : Fixed errors reported by Checkstyle (DG);
* 13-Mar-2003 : Implemented Serializable (DG);
* 29-May-2003 : Fixed bug in addMonths method (DG);
* 04-Sep-2003 : Implemented Comparable. Updated the isInRange javadocs (DG);
* 05-Jan-2005 : Fixed bug in addYears() method (1096282) (DG);
```

Dawno temu istniały powody tworzenia i utrzymywania takich dzienników na początku każdego modułu. Nie mieliśmy po prostu systemów kontroli wersji, które wykonywały to za nas. Obecnie jednak takie długie dzienniki tylko pogarszają czytelność modułu. Powinny zostać usunięte.

Komentarze wprowadzające szum informacyjny

Czasami zdarza się nam spotkać komentarze, które nie są niczym więcej jak tylko szumem informacyjnym. Przedstawiają one oczywiste dane i nie dostarczają żadnych nowych informacji.

```
/**
* Konstruktor domyślny.
*/
protected AnnualDateRule() {
}
```

No nie, *naprawdę*? Albo coś takiego:

```
/** Dzień miesiąca. */
private int dayOfMonth;
```

Następnie mamy doskonały przykład nadmiarowości:

```
/**
* Zwraca dzień miesiąca.
*
* @return dzień miesiąca.
*/
public int getDayOfMonth() {
   return dayOfMonth;
}
```

Komentarze takie stanowią tak duży szum informacyjny, że nauczyliśmy się je ignorować. Gdy czytamy kod, nasze oczy po prostu je pomijają. W końcu komentarze te głoszą nieprawdę, gdy otaczający kod jest zmieniany.

Pierwszy komentarz z listingu 4.4 wydaje się właściwy[2]. Wyjaśnia powód zignorowania bloku `catch`. Jednak drugi jest czystym szumem. Najwyraźniej programista był tak sfrustrowany pisaniem bloków `try-catch` w tej funkcji, że musiał sobie ulżyć.

LISTING 4.4. startSending

```
private void startSending()
{
  try
  {
    doSending();
  }
  catch(SocketException e)
  {
    // Normalne. Ktoś zatrzymał żądanie.
  }
  catch(Exception e)
```

[2] Obecny trend sprawdzania poprawności w komentarzach przez środowiska IDE jest zbawieniem dla wszystkich, którzy czytają dużo kodu.

```
    {
      try
      {
        response.add(ErrorResponder.makeExceptionString(e));
        response.closeAll();
      }
      catch(Exception e1)
      {
        // Muszę zrobić przerwę!
      }
    }
  }
```

Zamiast szukać ukojenia w bezużytecznych komentarzach, programista powinien zauważyć, że jego frustracja może być rozładowana przez poprawienie struktury kodu. Powinien skierować swoją energię na wyodrębnienie ostatniego bloku `try-catch` do osobnej funkcji, jak jest to pokazane na listingu 4.5.

LISTING 4.5. startSending (zmodyfikowany)

```
  private void startSending()
  {
    try
    {
      doSending();
    }
    catch(SocketException e)
    {
      // Normalne. Ktoś zatrzymał żądanie.
    }
    catch(Exception e)
    {
      addExceptionAndCloseResponse(e);
    }
  }

  private void addExceptionAndCloseResponse(Exception e)
  {
    try
    {
      response.add(ErrorResponder.makeExceptionString(e));
      response.closeAll();
    }
    catch(Exception e1)
    {
    }
  }
```

Warto zastąpić pokusę tworzenia szumu determinacją do wyczyszczenia swojego kodu. Pozwala to stać się lepszym i szczęśliwszym programistą.

Przerażający szum

Komentarze Javadoc również mogą być szumem. Jakie jest przeznaczenie poniższych komentarzy Javadoc (ze znanej biblioteki open source)? Odpowiedź: żadne. Są to po prostu nadmiarowe komentarze stanowiące szum informacyjny, napisane w źle pojętej chęci zapewnienia dokumentacji.

```
/** Nazwa. */
private String name;
/** Wersja. */
private String version;
/** nazwaLicencji. */
private String licenceName;
/** Wersja. */
private String info;
```

Przeczytajmy dokładniej te komentarze. Czy czytelnik może zauważyć błąd kopiowania i wklejania? Jeżeli autor nie poświęcił uwagi pisaniu komentarzy (lub ich wklejaniu), to czy czytelnik może oczekiwać po nich jakiejś korzyści?

Nie używaj komentarzy, jeżeli można użyć funkcji lub zmiennej

Przeanalizujmy poniższy fragment kodu:

```
// Czy moduł z listy globalnej <mod> zależy
// od podsystemu, którego jest częścią?
if (smodule.getDependSubsystems().contains(subSysMod.getSubSystem()))
```

Może to być przeorganizowane bez użycia komentarzy:

```
ArrayList moduleDependees = smodule.getDependSubsystems();
String ourSubSystem = subSysMod.getSubSystem();
if (moduleDependees.contains(ourSubSystem))
```

Autor oryginalnego kodu prawdopodobnie napisał komentarz na początku (niestety), a następnie kod realizujący zadanie z komentarza. Jeżeli jednak autor zmodyfikowałby kod w sposób, w jaki ja to wykonałem, komentarz mógłby zostać usunięty.

Znaczniki pozycji

Czasami programiści lubią zaznaczać określone miejsca w pliku źródłowym. Na przykład ostatnio trafiłem na program, w którym znalazłem coś takiego:

```
// Akcje ////////////////////////////////
```

Istnieją rzadkie przypadki, w których sensowne jest zebranie określonych funkcji razem pod tego rodzaju transparentami. Jednak zwykle powodują one chaos, który powinien być wyeliminowany — szczególnie ten pociąg ukośników na końcu.

Transparent ten jest zaskakujący i oczywisty, jeżeli nie widzimy go zbyt często. Tak więc warto używać ich oszczędnie i tylko wtedy, gdy ich zalety są wyraźne. Jeżeli zbyt często używamy tych transparentów, zaczynają być traktowane jako szum tła i ignorowane.

Komentarze w klamrach zamykających

Zdarza się, że programiści umieszczają specjalne komentarze po klamrach zamykających, tak jak na listingu 4.6. Choć może to mieć sens w przypadku długich funkcji, z głęboko zagnieżdżonymi strukturami, w małych i hermetycznych funkcjach, jakie preferujemy, tworzą tylko nieład. Jeżeli więc Czytelnik będzie chciał oznaczać klamry zamykające, niech spróbuje zamiast tego skrócić funkcję.

LISTING 4.6. wc.java

```
public class wc {
  public static void main(String[] args) {
    BufferedReader in = new BufferedReader(new InputStreamReader(System.in));
    String line;
    int lineCount = 0;
    int charCount = 0;
    int wordCount = 0;
    try {
      while ((line = in.readLine()) != null) {
        lineCount++;
        charCount += line.length();
        String words[] = line.split("\\W");
        wordCount += words.length;
      } //while
      System.out.println("wordCount = " + wordCount);
      System.out.println("lineCount = " + lineCount);
      System.out.println("charCount = " + charCount);
    } // try
    catch (IOException e) {
    System.err.println("Error:" + e.getMessage());
    } //catch
  } //main
}
```

Atrybuty i dopiski

/ Dodane przez Ricka */*

Systemy kontroli wersji świetnie nadają się do zapamiętywania, kto (i kiedy) dodał określony fragment. Nie ma potrzeby zaśmiecania kodu tymi małymi dopiskami. Można uważać, że tego typu komentarze będą przydatne do sprawdzenia, z kim można porozmawiać na temat danego fragmentu kodu. Rzeczywistość jest inna — zwykle zostają tam przez lata, tracąc na dokładności i użyteczności.

Pamiętajmy — systemy kontroli wersji są lepszym miejscem dla tego rodzaju informacji.

Zakomentowany kod

Niewiele jest praktyk tak nieprofesjonalnych, jak zakomentowanie kodu. Nie rób tego!

```
InputStreamResponse response = new InputStreamResponse();
response.setBody(formatter.getResultStream(), formatter.getByteCount());
// InputStream resultsStream = formatter.getResultStream();
// StreamReader reader = new StreamReader(resultsStream);
// response.setContent(reader.read(formatter.getByteCount()));
```

Inni programiści, którzy zobaczą taki zakomentowany kod, nie będą mieli odwagi go usunąć. Uznają, że jest tam z jakiegoś powodu i że jest zbyt ważny, aby go usunąć. W ten sposób zakomentowany kod zaczyna się odkładać jak osad na dnie butelki zepsutego wina.

Przeanalizujmy fragment z projektu Apache:

```
this.bytePos = writeBytes(pngIdBytes, 0);
//hdrPos = bytePos;
writeHeader();
writeResolution();
//dataPos = bytePos;
if (writeImageData()) {
   writeEnd();
   this.pngBytes = resizeByteArray(this.pngBytes, this.maxPos);
}
else {
   this.pngBytes = null;
}
return this.pngBytes;
```

Dlaczego te dwa wiersze kodu są zakomentowane? Czy są ważne? Czy jest to pozostałość po wcześniejszych zmianach? Czy też są błędami, które ktoś przed laty zakomentował i nie zadał sobie trudu, aby to wyczyścić?

W latach sześćdziesiątych ubiegłego wieku komentowanie kodu mogło być przydatne. Jednak od bardzo długiego czasu mamy już dobre systemy kontroli wersji. Systemy te pamiętają za nas wcześniejszy kod. Nie musimy już komentować kodu. Po prostu możemy go usunąć. Nie stracimy go. Gwarantuję.

Komentarze HTML

Kod HTML w komentarzach do kodu źródłowego jest paskudny, o czym można się przekonać po przeczytaniu kodu zamieszczonego poniżej. Powoduje on, że komentarze są trudne do przeczytania w jedynym miejscu, gdzie powinny być łatwe do czytania — edytorze lub środowisku IDE. Jeżeli komentarze mają być pobierane przez jakieś narzędzie (na przykład Javadoc), aby mogły być wyświetlone na stronie WWW, to zadaniem tego narzędzia, a nie programisty, powinno być opatrzenie ich stosownymi znacznikami HTML.

```
/**
 * Zadanie uruchomienia testów sprawności.
 * Zadanie uruchamia testy fitnesse i publikuje wyniki.
 * <p/>
 * <pre>
 * Zastosowanie:
 * &lt;taskdef name=&quot;execute-fitnesse-tests&quot;
 * classname=&quot;fitnesse.ant.ExecuteFitnesseTestsTask&quot;
 * classpathref=&quot;classpath&quot; /&gt;
 * LUB
 * &lt;taskdef classpathref=&quot;classpath&quot;
 * resource=&quot;tasks.properties&quot; /&gt;
 * <p/>
 * &lt;execute-fitnesse-tests
 * suitepage=&quot;FitNesse.SuiteAcceptanceTests&quot;
 * fitnesseport=&quot;8082&quot;
 * resultsdir=&quot;${results.dir}&quot;
 * resultshtmlpage=&quot;fit-results.html&quot;
 * classpathref=&quot;classpath&quot; /&gt;
 * </pre>
 */
```

Informacje nielokalne

Jeżeli konieczne jest napisanie komentarza, to należy upewnić się, że opisuje on kod znajdujący się w pobliżu. Nie należy udostępniać informacji dotyczących całego systemu w kontekście komentarzy lokalnych. Weźmy jako przykład zamieszczone poniżej komentarze Javadoc. Pomijając fakt, że są zupełnie zbędne, zawierają one informacje o domyślnym porcie. Funkcja jednak nie ma absolutnie żadnej kontroli nad tą wartością domyślną. Komentarz nie opisuje funkcji, ale inną część systemu, znacznie od niej oddaloną. Oczywiście, nie ma gwarancji, że komentarz ten zostanie zmieniony, gdy kod zawierający wartość domyślną ulegnie zmianie.

```
/**
 * Port, na którym działa fitnesse. Domyślnie <b>8082</b>.
 *
 * @param fitnessePort
 */
public void setFitnessePort(int fitnessePort)
{
   this.fitnessePort = fitnessePort;
}
```

Nadmiar informacji

Nie należy umieszczać w komentarzach interesujących z punktu widzenia historii dyskusji lub luźnych opisów szczegółów. Komentarz zamieszczony poniżej został pobrany z modułu mającego za zadanie sprawdzić, czy funkcja może kodować i dekodować zgodnie ze standardem base64. Osoba czytająca ten kod nie musi znać wszystkich szczegółowych informacji znajdujących się w komentarzu, poza numerem RFC.

```
/*
RFC 2045 - Multipurpose Internet Mail Extensions (MIME)
Part One: Format of Internet Message Bodies
section 6.8. Base64 Content-Transfer-Encoding
The encoding process represents 24-bit groups of input bits as output
strings of 4 encoded characters. Proceeding from left to right,
a 24-bit input group is formed by concatenating 3 8-bit input groups.
These 24 bits are then treated as 4 concatenated 6-bit groups, each
of which is translated into a single digit in the base64 alphabet.
When encoding a bit stream via the base64 encoding, the bit stream
must be presumed to be ordered with the most-significant-bit first.
That is, the first bit in the stream will be the high-order bit in
the first 8-bit byte, and the eighth bit will be the low-order bit in
the first 8-bit byte, and so on.
*/
```

Nieoczywiste połączenia

Połączenie pomiędzy komentarzem a kodem, który on opisuje, powinno być oczywiste. Jeżeli mamy problemy z napisaniem komentarza, to powinniśmy przynajmniej doprowadzić do tego, by czytelnik patrzący na komentarz i kod rozumiał, o czym mówi dany komentarz.

Jako przykład weźmy komentarz zaczerpnięty z projektu Apache:

```
/*
* Zaczynamy od tablicy, która jest na tyle duża, aby zmieścić wszystkie piksele
* (plus filter bajtów) oraz dodatkowe 200 bajtów na informacje nagłówka.
*/
this.pngBytes = new byte[((this.width + 1) * this.height * 3) + 200];
```

Co to są bajty `filter`? Czy ma to jakiś związek z wyrażeniem `+1`? A może z `*3`? Z obydwoma? Czy piksel jest bajtem? Dlaczego 200? Zadaniem komentarza jest wyjaśnianie kodu, który sam się nie objaśnia. Jaka szkoda, że sam komentarz wymaga dodatkowego objaśnienia.

Nagłówki funkcji

Krótkie funkcje nie wymagają rozbudowanych opisów. Odpowiednio wybrana nazwa małej funkcji realizującej jedną operację jest zwykle lepsza niż nagłówek z komentarzem.

Komentarze Javadoc w niepublicznym kodzie

Komentarze Javadoc są przydatne w publicznym API, ale za to niemile widziane w kodzie nieprzeznaczonym do publicznego rozpowszechniania. Generowanie stron Javadoc dla klas i funkcji wewnątrz systemu zwykle nie jest przydatne, a dodatkowy formalizm komentarzy Javadoc przyczynia się jedynie do powstania błędów i rozproszenia uwagi.

Przykład

Kod zamieszczony na listingu 4.7 został przeze mnie napisany na potrzeby pierwszego kursu XP Immersion. Był on w zamierzeniach przykładem złego stylu kodowania i komentowania. Później Kent Beck przebudował go do znacznie przyjemniejszej postaci na oczach kilkudziesięciu entuzjastycznie reagujących studentów. Później zaadaptowałem ten przykład na potrzeby mojej książki *Agile Software Development, Principles, Patterns, and Practices* i pierwszych artykułów *Craftman* publikowanych w magazynie *Software Development*.

Fascynujące w tym module jest to, że swego czasu byłby on uznawany za „dobrze udokumentowany". Teraz postrzegamy go jako mały bałagan. Spójrzmy, jak wiele problemów z komentarzami można tutaj znaleźć.

LISTING 4.7. GeneratePrimes.java

```
/**
 * Klasa ta generuje liczby pierwsze do określonego przez użytkownika
 * maksimum. Użytym algorytmem jest sito Eratostenesa.
 * <p>
 * Eratostenes z Cyrene, urodzony 276 p.n.e. w Cyrene, Libia --
 * zmarł 194 p.n.e. w Aleksandrii. Pierwszy człowiek, który obliczył
 * obwód Ziemi. Znany również z prac nad kalendarzem
 * z latami przestępnymi i prowadzenia biblioteki w Aleksandrii.
 * <p>
 * Algorytm jest dosyć prosty. Mamy tablicę liczb całkowitych
 * zaczynających się od 2. Wykreślamy wszystkie wielokrotności 2. Szukamy
```

```
 * następnej niewykreślonej liczby i wykreślamy wszystkie jej wielokrotności.
 * Powtarzamy działania do momentu osiągnięcia pierwiastka kwadratowego z maksymalnej wartości.
 *
 * @author Alphonse
 * @version 13 Feb 2002 atp
*/
import java.util.*;

  public class GeneratePrimes
  {
  /**
   * @param maxValue jest limitem generacji.
   */
  public static int[] generatePrimes(int maxValue)
  {
    if (maxValue >= 2) // Jedyny prawidłowy przypadek.
    {
      // Deklaracje.
      int s = maxValue + 1; // Rozmiar tablicy.
      boolean[] f = new boolean[s];
      int i;
      // Inicjalizacja tablicy wartościami true.
      for (i = 0; i < s; i++)
        f[i] = true;

      // Usuwanie znanych liczb niebędących pierwszymi.
      f[0] = f[1] = false;

      // Sito.
      int j;
      for (i = 2; i < Math.sqrt(s) + 1; i++)
      {
        if (f[i]) // Jeżeli i nie jest wykreślone, wykreślamy jego wielokrotności.
        {
          for (j = 2 * i; j < s; j += i)
            f[j] = false; // Wielokrotności nie są pierwsze.
        }
      }

      // Ile mamy liczb pierwszych?
      int count = 0;
      for (i = 0; i < s; i++)
      {
        if (f[i])
          count++; // Licznik trafień.
      }

      int[] primes = new int[count];

      // Przeniesienie liczb pierwszych do wyniku.
      for (i = 0, j = 0; i < s; i++)
      {
        if (f[i]) // Jeżeli pierwsza.
          primes[j++] = i;
      }
      return primes; // Zwracamy liczby pierwsze.
    }
    else // maxValue < 2
      return new int[0]; // Zwracamy pustą tablicę, jeżeli niewłaściwe dane wejściowe.
    }
}
```

Na listingu 4.8 zamieszczona jest przebudowana wersja tego samego modułu. Warto zauważyć, że znacznie ograniczona jest liczba komentarzy. W całym module znajdują się tylko dwa komentarze. Oba są z natury opisowe.

LISTING 4.8. PrimeGenerator.java (przebudowany)

```
/**
 * Klasa ta generuje liczby pierwsze do określonego przez użytkownika
 * maksimum. Użytym algorytmem jest sito Eratostenesa.
* Mamy tablicę liczb całkowitych zaczynających się od 2.
* Wyszukujemy pierwszą nieokreśloną liczbę i wykreślamy wszystkie jej
* wielokrotności. Powtarzamy, aż nie będzie więcej wielokrotności w tablicy.
*/

public class PrimeGenerator
{
  private static boolean[] crossedOut;
  private static int[] result;

  public static int[] generatePrimes(int maxValue)
  {
    if (maxValue < 2)
      return new int[0];
    else
    {
      uncrossIntegersUpTo(maxValue);
      crossOutMultiples();
      putUncrossedIntegersIntoResult();
      return result;
    }
  }

  private static void uncrossIntegersUpTo(int maxValue)
  {
    crossedOut = new boolean[maxValue + 1];
    for (int i = 2; i < crossedOut.length; i++)
      crossedOut[i] = false;
  }

  private static void crossOutMultiples()
  {
    int limit = determineIterationLimit();
    for (int i = 2; i <= limit; i++)
      if (notCrossed(i))
        crossOutMultiplesOf(i);
  }

  private static int determineIterationLimit()
  {
    // Każda wielokrotność w tablicy ma podzielnik będący liczbą pierwszą
    // mniejszą lub równą pierwiastkowi kwadratowemu wielkości tablicy,
    // więc nie musimy wykreślać wielokrotności większych od tego pierwiastka.
    double iterationLimit = Math.sqrt(crossedOut.length);
    return (int) iterationLimit;
  }

  private static void crossOutMultiplesOf(int i)
  {
    for (int multiple = 2*i;
         multiple < crossedOut.length;
         multiple += i)
      crossedOut[multiple] = true;
```

```
    }

    private static boolean notCrossed(int i)
    {
      return crossedOut[i] == false;
    }

    private static void putUncrossedIntegersIntoResult()
    {
      result = new int[numberOfUncrossedIntegers()];
      for (int j = 0, i = 2; i < crossedOut.length; i++)
        if (notCrossed(i))
          result[j++] = i;
    }

    private static int numberOfUncrossedIntegers()
    {
      int count = 0;
      for (int i = 2; i < crossedOut.length; i++)
        if (notCrossed(i))
          count++;

      return count;
    }
  }
```

Można się spierać, że pierwszy komentarz jest nadmiarowy, ponieważ czyta się go podobnie jak samą funkcję `genratePrimes`. Uważam jednak, że komentarz ułatwia czytelnikowi poznanie algorytmu, więc zdecydowałem o jego pozostawieniu.

Drugi komentarz jest niemal na pewno niezbędny. Wyjaśnia powody zastosowania pierwiastka jako ograniczenia pętli. Można sprawdzić, że żadna z prostych nazw zmiennych ani inna struktura kodu nie pozwala na wyjaśnienie tego punktu. Z drugiej strony, użycie pierwiastka może być próżnością. Czy faktycznie oszczędzam dużo czasu przez ograniczenie liczby iteracji do pierwiastka liczby? Czy obliczenie pierwiastka nie zajmuje więcej czasu, niż uda się nam zaoszczędzić?

Warto o tym pomyśleć. Zastosowanie pierwiastka jako limitu pętli zadowala siedzącego we mnie eksperta C i asemblera, ale nie jestem przekonany, że jest to warte czasu i energii osoby, która ma za zadanie zrozumieć ten kod.

Bibliografia

[KP78]: Kernighan i Plaugher, *The Elements of Programming Style*, McGraw-Hill 1978.