
Spis treści

Wstęp	11
Historia reguł	15
Jak nie zgadzać się z przedstawionymi tu regułami	19
1. Tak proste, jak to możliwe, lecz nie prostsze	21
Pomiar prostoty	23
...ale nie prostszy	24
Czasami lepiej jest uprościć problem niż rozwiązanie	26
Proste algorytmy	29
Nie trać z oczu celu	31
Jedna reguła, by rządzić innymi	32
2. Błędy są zaraźliwe	34
Nie polegaj na swoich użytkownikach	35
Zautomatyzowane testy mogą być kłopotliwe	36
Kod bezstanowy jest łatwiejszy do testowania	37
Badaj stan, którego nie możesz wyeliminować	42
Nie ufaj kodowi wywołującemu	43
Dbanie o poprawność kodu	48
3. Dobra nazwa jest najlepszą dokumentacją	49
Nie optymalizuj nazw pod kątem długości	50
Nie mieszaj konwencji	52
Nie strzelaj sobie samemu w stopę	54
Nie zmuszaj mnie do myślenia	57

4. Uogólnianie wymaga trzech przykładów	59
YAGNI	61
Oczywisty zarzut wobec tej strategii, w odpowiedzi na który powtórzę to samo	64
Pisanie kodu na wyrost to jeszcze pół biedy...	66
Nie tak wygląda sukces	71
5. Pierwsza lekcja optymalizacji: nie optymalizuj	73
Pierwsza lekcja optymalizacji	76
Druga lekcja optymalizacji	76
Sprawdzanie drugiej lekcji optymalizacji	77
Krok 1. Zmierz i przypisz czas procesora	77
Krok 2. Upewnij się, że nie ma błędu	78
Krok 3. Zmierz swoje dane	78
Krok 4. Planowanie i prototypowanie	78
Krok 5. Zoptymalizuj i powtórz	79
Stosowanie pięcioetapowego procesu optymalizacji	79
Nie ma żadnej trzeciej lekcji optymalizacji	83
Przerywnik, w którym poprzedni rozdział zostaje poddany krytyce	85
6. Przeglądy kodu są dobre z trzech powodów	89
Przeglądy kodu służą dzieleniu się wiedzą	91
Zabronione przeglądy kodu	93
Prawdziwa wartość przeglądów kodu	93
Przeglądy kodu mają społeczny charakter	93
7. Eliminuj przypadki niepowodzeń	95
Funkcja, która ułatwia strzał w stopę	96
Strzelanie sobie w stopę rykoszetem	97
Skorzystanie z pomocy kompilatora, by uniknąć strzelania sobie w stopę	99
Wycucie czasu jest kluczowe	100
Bardziej skomplikowany przykład	100
Uniemożliwianie popełniania błędów związanych z kolejnością	104
Stosowanie szablonów zamiast sekwencji wywołań metod	106
Skoordynowana kontrola stanu	106
Wykrywanie błędów jest dobre, ale jeszcze lepsze jest uniemożliwienie wyrażenia ich w kodzie	110

8. Kod, który nie jest wykonywany, nie działa	112
Krok 1. Prosty początek	113
Krok 2. Uogólnienie częstego wzorca	115
Krok 3. Dodawanie przebrań	116
Krok 4. Przyszła kryśka na Matyska	118
Kogo obwinić?	119
Ograniczenia testowania	120
9. Pisz kod, który można związać	122
Tak smakuje niepowodzenie	124
Rola pamięci krótkotrwałej	125
Gdzie narysować linię?	128
Koszt abstrakcji	130
Używaj abstrakcji, by ułatwiać zrozumienie kodu	131
Znaczenie pamięci długotrwałej	131
Wiedza powszechna jest darmowa, nowe koncepcje są kosztowne	133
Łącząc wszystko w całość	136
10. Gromadź złożoność w jednym miejscu	137
Prosty przykład	137
Ukrywanie szczegółów wewnętrznych	138
Rozproszony stan a złożoność	141
Zdolny do działania?	143
Widzieć jak przez mgłę	145
Ponowne przemyślenie podejścia	148
Złożoność w jednym miejscu — proste interakcje	150
11. Czy to jest dwa razy lepsze?	152
Trzy ścieżki ku przyszłości: ignoruj, ulepszaj, refaktoryzuj	153
Stopniowa ewolucja czy też ciągła rekonstrukcja	154
Prosta zasada	156
Radzenie sobie z niejasnymi korzyściami	157
Przeróbka jest dobrą okazją do rozwiązywania drobnych problemów	158
12. Duże zespoły wymagają silnych konwencji	159
Konwencje formatowania	160
Konwencje użycia języka	161
Konwencje rozwiązywania problemów	162
Efektywne zespoły myślą podobnie	167

13. Znajdź kamyk, który wywołał lawinę	169
Cykl życia błędu	169
Minimalizacja stanu	173
Radzenie sobie ze stanem, którego nie można uniknąć	177
Radzenie sobie z nieuniknionym opóźnieniem	180
14. Istnieją cztery rodzaje kodu	182
Łatwy problem, proste rozwiązanie	183
Łatwy problem, trzy skomplikowane rozwiązania	184
Koszt złożoności	188
Cztery (choć w zasadzie trzy) rodzaje programistów	188
Trudny problem, nieco skomplikowane rozwiązania, które nie działają	189
Trudny problem, nieco skomplikowane rozwiązanie	191
Trudny problem, łatwe rozwiązanie	194
15. Wyrwij chwasty	196
Identyfikacja chwastów	199
Jak w kodzie pojawiają się chwasty?	200
16. Kiedy rozwiązujesz problem, cofaj się i zaczynaj od wyniku, zamiast iść wprzód i wychodzić od kodu	202
Przykład	203
Pojawia się irytacja	206
Wybór jednej ze stron	208
Rozwiązuj problem, podążając wstecz	209
A teraz coś zupełnie innego	213
Praca poprzez podążanie wprzód lub wstecz	219
17. Czasami duży problem jest łatwiejszy do rozwiązania	221
Przeskok do wniosków	221
Znajdowanie czystej drogi do przodu	227
Rozpoznanie możliwości	230
18. Niech Twój kod opowie własną historię	232
Nie opowiadaj nieprawdziwych historii	233
Upewnij się, że historia będzie mieć sens	234
Opowiadanie dobrych historii	236

19. Przerabiaj równoległe	240
Przeszkody na drodze	240
Zamiast tego utwórz równoległy system	241
Konkretny przykład	242
Alokacja korzystająca ze stosu w praktyce	244
Chmura na horyzoncie	247
Nieco bardziej sprytnie konteksty stosu	248
Przejście ze starych kontekstów stosu na nowe	252
Przygotowania do migracji do klasy StackVector	254
Czas migrować	256
Rozpoznawanie, kiedy równoległe przerabianie jest dobrą strategią	258
20. Wykonaj obliczenia	259
Automatyzować czy nie automatyzować?	260
Poszukaj twardych ograniczeń	262
Kiedy obliczenia się zmieniają	266
Kiedy problem obliczeń ponownie staje się problemem Worda	267
21. Czasami będziesz musiał po prostu wbić gwoździe	269
Nowy argument	270
To nigdy nie jest tylko jeden błąd	272
Syreni zew automatyzacji	274
Zarządzanie wielkością plików	275
Nie ma żadnych skrótów	276
Wniosek: działaj na własnych zasadach	277
Użyj swojego najlepszego osądu	278
Przedyskutujcie to między sobą	279
Wypisuję się	279
A. Czytanie kodu C++ dla programistów Pythona	281
B. Czytanie kodu C++ dla programistów JavaScriptu	297