

Helion 



Piękny kod

Tajemnice mistrzów programowania

O'REILLY®

Pod redakcją Andy'ego Orama i Grega Wilsona

Tytuł oryginału: Beautiful Code: Leading Programmers Explain How They Think

Tłumaczenie: Łukasz Piwko (wstęp, rozdz. 1 – 16),
Marcin Rogóż (rozdz. 17 – 33),
Projekt okładki: Radosław Pazdrijowski i Mateusz Obarek

ISBN: 978-83-283-3477-9

© Helion SA 2008, 2017

Authorized translation of the English edition of Beautiful Code © 2007 O'Reilly Media, Inc.
This translation is published and sold by permission of O'Reilly Media, Inc., the owner of all rights to publish and sell the same.

All rights reserved. No part of this book may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording or by any information storage retrieval system, without permission from the Publisher.

Wszelkie prawa zastrzeżone. Nieautoryzowane rozpowszechnianie całości lub fragmentu niniejszej publikacji w jakiegokolwiek postaci jest zabronione. Wykonywanie kopii metodą kserograficzną, fotograficzną, a także kopiowanie książki na nośniku filmowym, magnetycznym lub innym powoduje naruszenie praw autorskich niniejszej publikacji.

Wszystkie znaki występujące w tekście są zastrzeżonymi znakami firmowymi bądź towarowymi ich właścicieli.

Autor oraz Wydawnictwo HELION dołożyli wszelkich starań, by zawarte w tej książce informacje były kompletne i rzetelne. Nie biorą jednak żadnej odpowiedzialności ani za ich wykorzystanie, ani za związane z tym ewentualne naruszenie praw patentowych lub autorskich. Autor oraz Wydawnictwo HELION nie ponoszą również żadnej odpowiedzialności za ewentualne szkody wynikłe z wykorzystania informacji zawartych w książce.

Fotografia na okładce została wykorzystana za zgodą iStockPhoto Inc.

Wydawnictwo HELION
ul. Kościuszki 1c, 44-100 GLIWICE
tel. 032 231 22 19, 032 230 98 63
e-mail: helion@helion.pl
WWW: <http://helion.pl> (księgarnia internetowa, katalog książek)

Drogi Czytelniku!
Jeżeli chcesz ocenić tę książkę, zajrzyj pod adres
<http://helion.pl/user/opinie/szpppv>
Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

Pliki z przykładami omawianymi w książce można znaleźć pod adresem:
<ftp://ftp.helion.pl/przyklady/szpppv.zip>

Printed in Poland.

- [Kup książkę](#)
- [Poleć książkę](#)
- [Oceń książkę](#)

- [Księgarnia internetowa](#)
- [Lubię to! » Nasza społeczność](#)

Słowo wstępne	13
Wstęp	15
1. Wyrażenia regularne	19
Programowanie w praktyce	20
Implementacja	21
Omówienie	22
Alternatywy	24
Rozszerzanie	25
Podsumowanie	27
2. Edytor delty w Subversion — interfejs jako ontologia	29
Kontrola wersji i transformacja drzewa	30
Prezentacja różnic pomiędzy drzewami	34
Interfejs edytora delty	35
Ale czy to jest sztuka?	40
Abstrakcja jako sport widowiskowy	43
Wnioski	45
3. Najpiękniejszy kod, którego nigdy nie napisałem	47
Najpiękniejszy kod, jaki kiedykolwiek napisałem	47
Coraz więcej za pomocą coraz mniejszych środków	49
Perspektywa	54
Co to jest pisanie	57
Zakończenie	57
Podziękowania	59
4. Wyszukiwanie	61
Na czas	61
Problem — dane z pamiętnika sieciowego	62
Problem — kto zażądał, czego i kiedy	70
Wyszukiwanie na dużą skalę	75
Podsumowanie	77

5. Poprawny, piękny, szybki (w takiej kolejności)	79
— lekcje z projektowania weryfikatorów XML	
Znaczenie walidacji XML	79
Problem	80
Wersja 1. Naiwna implementacja	82
Wersja 2. Imitacja gramatyki BNF $O(N)$	83
Wersja 3. Pierwsza optymalizacja $O(\log N)$	84
Wersja 4. Druga optymalizacja — nie sprawdzaj dwa razy	85
Wersja 5. Trzecia optymalizacja $O(1)$	87
Wersja 6. Czwarta optymalizacja — buforowanie	91
Morał	93
6. Framework for Integrated Test — piękno poprzez delikatność	95
Acceptance Testing Framework w trzech klasach	96
Wyzwanie zaprojektowania środowiska	98
Otwarte środowisko	99
Jak prosty może być parser HTML	100
Podsumowanie	103
7. Piękne testy	105
To niesforne wyszukiwanie binarne	106
Wstęp do JUnit	109
Rozprawić się z wyszukiwaniem binarnym	111
Podsumowanie	122
8. Generowanie w locie kodu do przetwarzania obrazów	125
9. Kolejność wykonywania operatorów	147
JavaScript	148
Tablica symboli	149
Tokeny	150
Kolejność	151
Wyrażenia	152
Operatory wzrostkowe	152
Operatory przedrostkowe	154
Operatory przypisania	155
Stałe	155
Zakres	156
Instrukcje	157
Funkcje	160
Literały tablicowe i obiektowe	161
Rzeczy do zrobienia i przemyślenia	162
10. Poszukiwanie szybszych metod zliczania bitów w stanie wysokim	163
Podstawowe metody	164
Dziel i zwyciężaj	165
Inne metody	167

Suma i różnica liczb ustawionych bitów w dwóch słowach	169
Porównywanie liczby ustawionych bitów w dwóch słowach	169
Zliczanie jedynek w tablicy	170
Zastosowania	175
11. Bezpieczna komunikacja — technologia wolności	177
Początki	178
Rozwikłać tajemnicę bezpiecznego przesyłania wiadomości	180
Klucz to użyteczność	181
Podstawa	184
Zestaw testów	188
Działający prototyp	189
Oczyść, podłącz i używaj	190
Hakowanie w Himalajach	194
Niewidoczne ruchy ręką	199
Prędkość ma znaczenie	201
Prywatność komunikacji dla praw jednostki	202
Hakowanie cywilizacji	203
12. Hodowanie pięknego kodu w języku BioPerl	205
BioPerl i moduł Bio::Graphics	206
Proces projektowania modułu Bio::Design	210
Rozszerzanie modułu Bio::Graphics	228
Wnioski i lekcje	232
13. Projekt programu Gene Sorter	235
Interfejs użytkownika programu Gene Sorter	236
Podtrzymywanie dialogu z użytkownikiem przez internet	237
Nieco polimorfizmu	239
Filtrowanie w celu znalezienia odpowiedniego genu	242
Ogólna teoria pięknego kodu	243
Podsumowanie	246
14. Jak elegancki kod ewoluuje wraz ze sprzętem — przypadek eliminacji Gaussa	247
Wpływ architektury komputerów na algorytmy macierzowe	248
Metoda dekompozycyjna	250
Prosta wersja	251
Podprocedura DGEFA biblioteki LINPACK	252
Procedura LAPACK DGETRF	255
Rekursywna dekompozycja LU	257
Procedura ScaLAPACK PDGETRF	260
Wielowątkowość w systemach wielordzeniowych	265
Słowo na temat analizy błędów i liczby operacji	267
Przyszłe kierunki badań	268
Literatura zalecana	269

15. Długoterminowe korzyści z pięknego projektu	271
Moje wyobrażenie o pięknym kodzie	271
Wprowadzenie do biblioteki CERN	272
Zewnętrzne piękno	273
Piękno wewnętrzne	278
Podsumowanie	284
16. Model sterowników jądra systemu Linux — korzyści płynące ze współpracy	285
Skromne początki	286
Redukcja do jeszcze mniejszych rozmiarów	290
Skalowanie do tysięcy urządzeń	293
Małe, luźno połączone obiekty	294
17. Inny poziom pośredniości	297
Od kodu do wskaźników	297
Od argumentów funkcji do wskaźników argumentów	300
Od systemów plików do warstw systemów plików	303
Od kodu do języka konkretnej domeny	305
Multipleksacja i demultipleksacja	307
Na zawsze warstwy?	308
18. Implementacja słownika w Pythonie — być wszystkim dla wszystkich	311
Wewnątrz słownika	313
Warunki specjalne	314
Kolizje	316
Zmiana rozmiaru	317
Iteracje i zmiany dynamiczne	318
Podsumowanie	319
Podziękowania	319
19. Wielowymiarowe iteratory w NumPy	321
Kluczowe wyzwania w operacjach na N-wymiarowych tablicach	322
Modele pamięci dla tablicy N-wymiarowej	323
Początki iteratora NumPy	324
Interfejs iteratora	331
Wykorzystanie iteratora	332
Podsumowanie	336
20. System korporacyjny o wysokim stopniu niezawodności dla misji Mars Rover NASA	337
Misja i Collaborative Information Portal	338
Wymagania misji	339
Architektura systemu	340
Studium przypadku — usługa strumieniowa	343
Niezawodność	346
Solidność	353
Podsumowanie	355

21. ERP5 — projektowanie maksymalnej giętkości	357
Ogólne cele ERP	358
ERP5	358
Podstawowa platforma Zope	360
Założenia ERP5 Project	364
Pisanie kodu dla ERP5 Project	365
Podsumowanie	368
22. Łyżka dziegciu	371
23. Programowanie rozproszone z zastosowaniem MapReduce	389
Motywujący przykład	389
Model programistyczny MapReduce	392
Inne przykłady MapReduce	393
Implementacja rozproszonego MapReduce	394
Rozszerzenia modelu	398
Wnioski	399
Literatura zalecana	400
Podziękowania	400
Dodatek: przykład algorytmu zliczającego słowa	400
24. Piękna współbieżność	403
Prosty przykład: konta bankowe	404
Pamięć transakcyjna STM	406
Problem Świętego Mikołaja	414
Refleksje na temat Haskell'a	422
Wnioski	423
Podziękowania	424
25. Abstrakcja składniowa — rozszerzenie syntax-case	425
Krótkie wprowadzenie do syntax-case	429
Algorytm rozwijania	431
Przykład	443
Wnioski	445
26. Architektura oszczędzająca nakłady — obiektowy framework dla oprogramowania sieciowego	447
Przykładowa aplikacja — usługa rejestrowania	449
Zorientowany obiektowo projekt frameworku serwera rejestrowania	451
Implementacja sekwencyjnych serwerów rejestrowania	457
Implementacja współbieżnych serwerów rejestrowania	461
Wnioski	467
27. Integracja partnerów biznesowych z wykorzystaniem architektury REST	469
To projektu	470
Udostępnianie usług klientom zewnętrznym	470

Przekazywanie usługi za pomocą wzorca fabryki	473
Wymiana danych z użyciem protokołów e-biznesowych	475
Wnioski	480
28. Piękne debugowanie	481
Debugowanie debugera	482
Systematyczny proces	483
Szukany problem	485
Automatyczne wyszukiwanie przyczyny awarii	486
Debugowanie delta	488
Minimalizacja wejścia	490
Polowanie na usterkę	490
Problem prototypu	493
Wnioski	493
Podziękowania	494
Literatura zalecana	494
29. Traktując kod jako esej	495
30. Gdy ze światem łączy cię tylko przycisk	501
Podstawowy model projektu	502
Interfejs wejściowy	505
Wydajność interfejsu użytkownika	518
Pobieranie	518
Przyszłe kierunki rozwoju	519
31. Emacspeak — kompletne dźwiękowe środowisko pracy	521
Tworzenie wyjścia mówionego	522
Włączanie mowy w Emacsie	523
Bezbolesny dostęp do informacji online	534
Podsumowanie	541
Podziękowania	544
32. Kod w ruchu	545
O byciu „podręcznikowym”	546
Podobne wygląda podobnie	547
Niebezpieczeństwa wcięć	548
Poruszanie się po kodzie	549
Wykorzystywane przez nas narzędzia	550
Burzliwa przeszłość DiffMerge	552
Wnioski	554
Podziękowania	554
Literatura zalecana	554

33. Pisanie programów dla Księgi	557
Niekrólewska droga	558
Ostrzeżenie dla nawiasofobów	558
Trzy w rządzie	559
Śliskie nachylenie	561
Nierówność trójkąta	563
Meandrowanie	565
„No przecież!”, znaczy się „Aha!”	566
Wnioski	567
Zalecana literatura	568
Posłowie	571
Autorzy	573
Skorowidz	583

Najpiękniejszy kod, którego nigdy nie napisałem

Jon Bentley

KIEDYS SŁYSZAŁEM, ŻE PEWIEN MISTRZ PROGRAMOWANIA dawał taką oto pochwałę: „On dodaje funkcje poprzez usuwanie kodu”. Antoine Saint-Exupéry, francuski pisarz i lotnik, wyraził tę myśl bardziej ogólnie: „Projektant może uznać, że osiągnął perfekcję, nie wtedy, kiedy nie pozostało już nic do dodania, ale wtedy, gdy nie można już nic odjąć”. W oprogramowaniu najpiękniejszego kodu, najpiękniejszych funkcji i najpiękniejszych programów czasami w ogóle nie ma.

Oczywiście trudno dyskutować o rzeczach, których nie ma. Ten rozdział jest próbą wykonania tego przytłaczającego zadania poprzez zaprezentowanie nowatorskiej analizy czasu pracy klasycznego programu Quicksort. Pierwszy podrozdział zawiera ogólny opis programu z osobistego punktu widzenia. Następny — to już treść właściwa tego rozdziału. Zaczniemy od dodania jednego licznika do programu, a następnie będziemy manipulować kodem, żeby stawał się coraz mniejszy i potężniejszy, aż tylko kilka wierszy kodu w pełni będzie pokrywać jego średni czas działania. Trzeci podrozdział podsumowuje techniki i zawiera niezwykle zwięzłą analizę kosztów binarnych drzew poszukiwań. Wskazówki znajdujące się w dwóch ostatnich podrozdziałach, oparte na spostrzeżeniach zawartych w tym tekście, pomogą nam pisać bardziej eleganckie programy.

Najpiękniejszy kod, jaki kiedykolwiek napisałem

Kiedy Greg Wilson przedstawił mi pomysł na tę książkę, zadałem sobie pytanie, jaki był najpiękniejszy kod, który napisałem. Po prawie całym dniu kołatania się tego pytania w mojej głowie zdałem sobie sprawę, że ogólna odpowiedź jest niezwykle prosta: Quicksort. Jednak w zależności od tego, jak precyzyjnie sformuluje się to pytanie, można odpowiedzieć na nie na trzy sposoby.

Tematem mojej rozprawy naukowej były algorytmy typu „dziel i zwyciężaj”. Dzięki niej odkryłem, że algorytm Quicksort napisany przez programistę o nazwisku C. A. R. Hoare (*Quicksort*, „Computer Journal” nr 5) jest niezaprzeczalnie dziadkiem ich wszystkich. Jest to piękny algorytm rozwiązujący podstawowy problem, który można zaimplementować w eleganckim kodzie. Zawsze go uwielbiałem, ale trzymałem się z dala od jego najgłębiej zagnieżdżonej pętli. Kiedyś spędziłem dwa dni na debugowaniu programu opartego na niej i całymi latami kopiowałem skrupulatnie kod za każdym razem, kiedy musiałem wykonać podobne zadanie. Rozwiązywał moje problemy, ale nigdy tak *naprawdę* go nie rozumiałem.

W końcu nauczyłem się od Nico Lomuto eleganckiej metody dzielenia i nareszcie mogłem napisać program Quicksort, który byłby dla mnie zrozumiały, a nawet umiałbym udowodnić, że jest poprawny. Spostrzeżenie Williama Strunka Jr., że „piszący szybko piszą zwięźle”, ma zastosowanie zarówno do kodu, jak i języka angielskiego. W związku z tym, idąc za jego radą, „pomijałem zbędne słowa” (*The Elements of Style*). Udało mi się zredukować 40 wierszy kodu do równo 12. A więc jeśli pytanie brzmi: „Jaki jest najpiękniejszy mały fragment kodu, jaki w życiu napisałem?”, moja odpowiedź to: Quicksort z mojej książki pod tytułem *Perelki oprogramowania*¹. Ta funkcja Quicksort, napisana w języku C, została przedstawiona na listingu 3.1. W następnym podrozdziale zajmiemy się dalszym dostrajaniem i badaniem tego kodu.

LISTING 3.1. Funkcja Quicksort

```
void quicksort(int l, int u)
{
    int i, m;
    if (l >= u) return;
    swap(l, randint(l, u));
    m = l;
    for (i = l+1; i <= u; i++)
        if (x[i] < x[l])
            swap(++m, i);
    swap(l, m);
    quicksort(l, m-1);
    quicksort(m+1, u);
}
```

Kod ten sortuje globalną tablicę $x[n]$, kiedy jest wywoływany z argumentami `quicksort(0, n-1)`. Oba argumenty tej funkcji są indeksami podtablicy, która ma być posortowana. `l` oznacza dolną granicę (ang. *lower*), a `u` — górną (ang. *upper*). Wywołanie funkcji `swap(i, j)` powoduje zamianę zawartości elementów $x[i]$ i $x[j]$. Pierwsza funkcja `swap` losowo wybiera element podziału, który w taki sam sposób jest wybierany pomiędzy `l` i `u`.

Książka *Perelki oprogramowania* zawiera szczegółowy opis i dowód poprawności funkcji `quicksort`. Zakładam, że Czytelnik zna algorytm Quicksort na poziomie tamtego opisu i najbardziej podstawowych książek o algorytmach.

Jeśli zmienimy pytanie na: „Jaki jest najpiękniejszy powszechnie używany fragment kodu, który napisałeś?”, moja odpowiedź ponownie będzie brzmieć Quicksort. W artykule napisanym razem

¹ Jon Bentley, *Perelki oprogramowania*, wyd. 2, Wydawnictwa Naukowo-Techniczne, Warszawa 2001 — *przyp. red.*

z M. D. McIlroyem² omawiamy poważny błąd związany z wydajnością w nieco sędziwej już funkcji systemu Unix — `qsort`. Wzięliśmy się za pisanie nowej funkcji `sort` dla biblioteki języka C, biorąc pod uwagę wiele różnych algorytmów do wykorzystania, w tym *Merge Sort* i *Heap Sort*. Po porównaniu kilku możliwości implementacji zdecydowaliśmy się na wersję z algorytmem Quicksort. We wspomnianym artykule wyjaśniamy, w jaki sposób napisaliśmy nową funkcję, która była bardziej przejrzysta, szybsza i solidniejsza niż jej konkurentki — po części z racji swoich niewielkich rozmiarów. Mądra rada Gordona Bella okazała się słuszna: „Najtańsze, najszybsze i najbardziej niezawodne komponenty systemu komputerowego to te, których nie ma”. Funkcja ta jest już powszechnie używana od ponad dziesięciu lat i nie zgłoszono jeszcze żadnych błędów.

Biorąc pod uwagę korzyści płynące ze zmniejszania objętości kodu, zadałem sobie w końcu trzeci wariant pytania zamieszczonego na początku tego rozdziału: „Jaki jest najpiękniejszy fragment kodu, którego *nigdy* nie napisałem?”. Jak udało mi się osiągnąć bardzo dużo za pomocą tak małych środków? Odpowiedź i tym razem jest związana z Quicksort, a konkretnie z analizą jego wydajności. O tym opowiadałem w kolejnym podrozdziale.

Coraz więcej za pomocą coraz mniejszych środków

Quicksort to bardzo elegancki algorytm, który nadaje się do wykonywania wnikliwych analiz. Około roku 1980 odbyłem wspaniałą rozmowę z Tonym Hoarem na temat historii jego algorytmu. Powiedział mi, że kiedy go opracował, wydawało mu się, iż jest on zbyt prosty do opublikowania. Napisał więc tylko swój klasyczny artykuł *Quicksort*, kiedy udało mu się przeanalizować jego oczekiwany czas wykonywania.

Łatwo się zorientować, że posortowanie tablicy zawierającej n elementów algorytmowi Quicksort może w najgorszym przypadku zająć około n^2 czasu. W najlepszym natomiast przypadku wybiera on wartość średnią jako element dzielący, dzięki czemu sortuje tablicę za pomocą około $n \times \lg(n)$ porównań. A więc ilu średnio porównań potrzebuje w przypadku losowej tablicy n różnych wartości?

Analiza tego problemu dokonana przez Hoare’a jest piękna, ale niestety wykraczająca poza wiedzę matematyczną wielu programistów. Kiedy uczyłem zasady działania algorytmu Quicksort studentów, martwiło mnie, że wielu z nich nie mogło zrozumieć dowodu, nawet mimo mojego szczerego wysiłku. Spróbujemy teraz podejść do tego zagadnienia w eksperymentalny sposób. Zaczniemy od programu Hoare’a i stopniowo dojdziemy do analizy zbliżonej do jego własnej.

Naszym zadaniem jest zmodyfikować kod z listingu 3.1 przedstawiającego randomizujący kod Quicksort, aby drogą analizy sprawdzał średnią liczbę porównań potrzebnych do posortowania tablicy zawierającej unikatowe elementy. Spróbujemy też uzyskać jak najwięcej przy użyciu jak najmniejszej ilości kodu, czasu i miejsca.

Aby określić średnią liczbę porównań, najpierw rozszerzymy funkcjonalność programu o możliwość ich zliczania. W tym celu inkrementujemy zmienną `comps` przed porównaniem w wewnętrznej pętli (listing 3.2).

² J. Bentley, M. D. McIlroy, *Engineering a sort function*, „Software-Practice and Experience”, Vol. 23, No. 11 — *przyp. red.*

LISTING 3.2. Wewnętrzna pętla algorytmu Quicksort przystosowana do zliczania porównań

```
for (i = l+1; i <= u; i++) {
    comps++;
    if (x[i] < x[l])
        swap(++m, i);
}
```

Jeśli uruchomimy program tylko dla jednego n , dowiemy się, ile porównań to jedno uruchomienie potrzebuje. Jeśli powtórzymy tę operację wielokrotnie dla wielu wartości n i przeprowadzimy statystyczną analizę wyników, uzyskamy wartość średnią. Algorytm Quicksort potrzebuje około $1,4 n \times \lg(n)$ porównań do posortowania n elementów.

Nie jest to wcale zły sposób na uzyskanie wglądu w działanie programu. Dzięki 13 wierszom kodu i kilku eksperymentom można sporo odkryć. Znane powiedzenie przypisywane pisarzom takim jak Blaise Pascal i T. S. Eliot brzmi: „Gdybym miał więcej czasu, napisałbym Ci krótszy list”. My mamy czas, więc poeksperymentujemy trochę z kodem, aby napisać krótszy (i lepszy) program.

Zagramy w przyspieszanie eksperymentu, próbując zwiększyć statystyczną dokładność i wgląd w działanie programu. Jako że wewnętrzna pętla wykonuje dokładnie $u-1$ porównań, możemy nieco przyspieszyć działanie programu, zliczając te porównania za pomocą pojedynczej operacji poza pętlą. Po tej zmianie algorytm Quicksort wygląda jak na listingu 3.3.

LISTING 3.3. Algorytm Quicksort po przeniesieniu inkrementacji na zewnątrz pętli

```
comps += u-1;
for (i = l+1; i <= u; i++)
    if (x[i] < x[l])
        swap(++m, i);
```

Program ten sortuje tablicę i jednocześnie sprawdza liczbę potrzebnych porównań. Jeśli jednak naszym celem jest tylko zliczenie porównań, nie musimy sortować tablicy. Na listingu 3.4 zostało usunięte prawdziwe sortowanie i pozostał tylko szkielet różnych wywołań wykonywanych przez program.

LISTING 3.4. Szkielet algorytmu Quicksort zredukowany do zliczania

```
void quickcount(int l, int u)
{
    int m;
    if (l >= u) return;
    m = randint(l, u);
    comps += u-l;
    quickcount(l, m-1);
    quickcount(m+1, u);
}
```

Program ten działa dzięki losowemu wybieraniu przez Quicksort elementu dzielącego i dzięki założeniu, że wszystkie elementy są unikatowe. Jest on wykonywany w czasie proporcjonalnym do n . Podczas gdy program z listingu 3.3 wymagał proporcjonalnej do n ilości miejsca, teraz została ona zredukowana do stosu rekurencji, który średnio jest proporcjonalny do $\lg(n)$.

Mimo że indeksy (l i u) tablicy są niezbędne w prawdziwym programie, w tej wersji szkieletu nie mają znaczenia. Można je zastąpić jedną liczbą całkowitą (n), która będzie określała rozmiar podtablicy do posortowania (listing 3.5).

LISTING 3.5. Szkielet algorytmu Quicksort z jednym argumentem określającym rozmiar

```
void qc(int n)
{
    int m;
    if (n <= 1) return;
    m = randint(1, n);
    comps += n-1;
    qc(m-1);
    qc(n-m);
}
```

Bardziej naturalne teraz będzie przetworzenie tej procedury do postaci funkcji zliczającej porównania (ang. *comparison count* — *cc*), która zwraca liczbę porównań użytych przez jedno wykonanie algorytmu Quicksort. Funkcję tę przedstawia listing 3.6.

LISTING 3.6. Szkielet algorytmu Quicksort zaimplementowany jako funkcja

```
int cc(int n)
{
    int m;
    if (n <= 1) return 0;
    m = randint(1, n);
    return n-1 + cc(m-1) + cc(n-m);
}
```

Przykłady zamieszczone na listingach 3.4, 3.5 i 3.6 rozwiązują ten sam podstawowy problem i potrzebują na to tyle samo czasu i pamięci. Każda kolejna wersja ma poprawioną formę, dzięki czemu jest nieco bardziej przejrzysta i zwięzła od poprzedniej.

Definiując **paradoks wynalazcy** (ang. *inventor's paradox*), George Pólya oznajmia, że: „Bardziej ambitny plan może mieć więcej szans na powodzenie”³. Spróbujemy teraz wykorzystać ten paradoks w analizie Quicksort. Do tej pory zadawaliśmy sobie pytanie, ile porównań potrzebuje algorytm Quicksort do posortowania tablicy zawierającej n elementów. Teraz zadamy bardziej ambitne pytanie: ile średnio porównań potrzebuje algorytm Quicksort do posortowania losowej tablicy o rozmiarze n ? Możemy rozszerzyć kod z listingu 3.6, aby uzyskać pseudokod widoczny na listingu 3.7.

LISTING 3.7. Średnia liczba porównań algorytmu Quicksort jako pseudokod

```
float c(int n)
{
    if (n <= 1) return 0;
    sum = 0;
    for (m = 1; m <= n; m++)
        sum += n-1 + c(m-1) + c(n-m);
    return sum/n;
}
```

Jeśli dane wejściowe zawierają maksymalnie jeden element, Quicksort nie wykonuje żadnych porównań, jak w przykładzie z listingu 3.6. W przypadku n o większej wartości kod ten bierze pod uwagę każdą wartość dzielącą (od pierwszego do ostatniego elementu — każdy jest równie prawdopodobny)

³ George Pólya, *How to solve it*, Princeton University Press, 1945 — *przyp. red.*

i określa koszt podziału w każdym z tych miejsc. Następnie kod oblicza sumę tych wartości (w ten sposób rekursywnie rozwiązując jeden problem rozmiaru $m-1$ i jeden problem rozmiaru $n-m$) i dzieli ją przez n , uzyskując średnią.

Gdybyśmy mogli obliczyć tę liczbę, nasze eksperymenty byłyby znacznie bardziej potężne. Zamiast przeprowadzać wiele eksperymentów z jedną wartością n w celu oszacowania średniej, jeden eksperyment wystarczyłby do uzyskania prawdziwej średniej. Niestety, ta potęga ma swoją cenę: program działa w czasie proporcjonalnym do 3^n (interesującym ćwiczeniem jest analiza tego czasu przy użyciu technik opisanych w tym rozdziale).

Kod z listingu 3.7 potrzebuje właśnie tyle czasu, ponieważ oblicza pododpowiedzi wielokrotnie. W takim przypadku można zastosować **programowanie dynamiczne** w celu zapisywania tych pododpowiedzi, co pozwoli na uniknięcie ich ponownego obliczania. W tym przypadku wprowadzimy tablicę $t[N+1]$, w której element $t[n]$ przechowuje $c(n)$, i obliczymy jej wartości w kolejności rosnącej. N będzie oznaczać maksymalną wartość n , czyli rozmiar tablicy do posortowania. Rezultat jest widoczny na listingu 3.8.

LISTING 3.8. Obliczenia algorytmu Quicksort przy użyciu programowania dynamicznego

```
t[0] = 0
for (n = 1; n <= N; n++)
    sum = 0
    for (i = 1; i <= n; i++)
        sum += n-1 + t[i-1] + t[n-i]
    t[n] = sum/n
```

Program ten jest z grubsza transkrypcją kodu z listingu 3.7, w której zastąpiono $c(n)$ zapisem $t[n]$. Jego czas wykonywania jest proporcjonalny do N^2 , a ilość zajmowanego miejsca do N . Jedną z jego zalet jest to, że po zakończeniu wykonywania tablica t zawiera rzeczywiste wartości średnie (a nie przybliżoną wartość przykładowych średnich) dla elementów tablicy od 0 do N . Dzięki analizie tych liczb można uzyskać informacje na temat funkcjonalnej formy spodziewanej liczby porównań wykonanych przez algorytm Quicksort.

Teraz uprościmy nasz program jeszcze bardziej. Najpierw przeniesiemy człon $n-1$ poza pętlę, jak widać na listingu 3.9.

LISTING 3.9. Obliczenia Quicksort z kodem przeniesionym na zewnątrz pętli

```
t[0] = 0
for (n = 1; n <= N; n++)
    sum = 0
    for (i = 1; i <= n; i++)
        sum += t[i-1] + t[n-i]
    t[n] = n-1 + sum/n
```

Dalsze dostrajanie kodu będzie polegało na użyciu symetrii. Jeśli na przykład n wynosi 4, wewnętrzna pętla oblicza następującą sumę:

$$t[0]+t[3] + t[1]+t[2] + t[2]+t[1] + t[3]+t[0]$$

W tym szeregu par pierwsze elementy zwiększają się, podczas gdy mniejsze zmniejszają. Możemy zatem sumę tę zapisać tak:


```
2 * (t[0] + t[1] + t[2] + t[3])
```

Za pomocą tej symetrii otrzymamy algorytm widoczny na listingu 3.10.

LISTING 3.10. Obliczenia Quicksort przy użyciu symetrii

```
t[0] = 0
for (n = 1; n <= N; n++)
    sum = 0
    for (i = 0; i < n; i++)
        sum += 2 * t[i]
    t[n] = n-1 + sum/n
```

Kod ten jednak również nie jest w pełni efektywny, ponieważ wielokrotnie oblicza tę samą sumę. Zamiast dodawać wszystkie poprzednie człony, możemy zmienną `sum` zainicjalizować poza pętlą i dodać następny człon. Rezultat jest widoczny na listingu 3.11.

LISTING 3.11. Obliczenia Quicksort z usuniętą wewnętrzną pętlą

```
sum = 0; t[0] = 0
for (n = 1; n <= N; n++)
    sum += 2*t[n-1]
    t[n] = n-1 + sum/n
```

Ten niewielki program jest naprawdę użyteczny. W czasie proporcjonalnym do N tworzy tabelę rzeczywistych spodziewanych czasów wykonania algorytmu Quicksort dla każdej liczby całkowitej od 1 do N .

Kod z listingu 3.11 jest łatwy do użycia w arkuszu kalkulacyjnym, w którym wartości są natychmiast dostępne do dalszej analizy. Tabela 3.1 przedstawia początkowe wiersze.

TABELA 3.1. Wynik implementacji kodu z listingu 3.11 w arkuszu kalkulacyjnym

N	Suma	t[n]
0	0	0
1	0	0
2	0	1
3	2	2.667
4	7.333	4.833
5	17	7.4
6	31.8	10.3
7	52.4	13.486
8	79.371	16.921

Pierwszy wiersz liczb w tej tabeli jest inicjalizowany za pomocą trzech stałych z kodu. W notacji arkuszy kalkulacyjnych kolejny wiersz liczb (trzeci wiersz arkusza) jest obliczany przy użyciu następujących zależności:

$$A3 = A2+1$$

$$B3 = B2 + 2*C2$$

$$C3 = A3-1 + B3/A3$$

Kopiując poprzez przeciągnięcie te (względne) odwołania w dół, można uzupełnić arkusz. Ten arkusz jest moim poważnym kandydatem na „najpiękniejszy kod, jaki kiedykolwiek napisałem” w kategorii osiągnięcia jak najwięcej za pomocą tylko kilku wierszy kodu.

Co jednak, jeśli nie potrzebujemy tych wszystkich wartości? Gdybyśmy na przykład woleli przeanalizować tylko kilka z wartości (na przykład wszystkie potęgi cyfry 2 od 2^0 do 2^{32})? Mimo że kod z listingu 3.11 tworzy pełną tablicę `t`, używa on tylko jej najnowszej wartości.

Możemy zatem zastąpić liniową przestrzeń tablicy `t[]` stałą przestrzenią zmiennej `t`, jak na listingu 3.12.

Listing 3.12. Obliczenia Quicksort — ostateczna wersja

```
sum = 0; t = 0
for (n = 1; n <= N; n++)
    sum += 2*t
t = n-1 + sum/n
```

Można następnie wstawić dodatkowy wiersz kodu w celu sprawdzenia trafności `n` i w razie potrzeby wydrukować wyniki.

Ten niewielki program jest ostatnim etapem naszej podróży. Dobrą konkluzją w odniesieniu do niej mogą być słowa Alana Perlisa: „Prostota nie występuje przed złożonością, ale jest jej następstwem”⁴.

Perspektywa

Tabela 3.2 zawiera zestawienie programów analizujących Quicksort, prezentowanych w tym rozdziale.

TABELA 3.2. Ewolucja programu analizującego pracę algorytmu Quicksort

Numer przykładu	Liczba wierszy	Typ odpowiedzi	Liczba odpowiedzi	Czas trwania	Przestrzeń
2	13	Przykładowa	1	$n \times \lg(n)$	N
3	13	"	"	"	"
4	8	"	"	n	$\lg(n)$
5	8	"	"	"	"
6	6	"	"	"	"
7	6	Dokładna	"	3^N	N
8	6	"	N	N^2	N
9	6	"	"	"	"
10	6	"	"	"	"
11	4	"	"	N	"
12	4	Dokładna	N	N	1

⁴ Alan Perlis, *Epigrams on Programming*, „Sigplan Notices”, Vol. 17, Issue 9 — *przyp. red.*

Każdy etap ewolucji naszego kodu był bardzo prosty. Przejście od przykładu zamieszczonego na listingu 3.6 do dokładnej odpowiedzi na listingu 3.7 jest prawdopodobnie najbardziej subtelne. Kod w miarę kurczenia się stawał się coraz szybszy. W połowie XIX wieku Robert Browning zauważył, że „mniej oznacza więcej”. Ta tabela umożliwi ilościowe określenie jednego z przykładów tamtej minimalistycznej filozofii.

Widzieliśmy trzy zasadniczo różniące się typy programów. Przykłady z listingów 3.2 i 3.3 są działającymi algorytmami Quicksort przystosowanymi do zliczania porównań w trakcie sortowania prawdziwej tablicy. Listingi 3.4 do 3.6 implementują prosty model Quicksort — imitują jedno uruchomienie algorytmu, w rzeczywistości nie wykonując żadnego sortowania. Listingi 3.7 do 3.12 implementują bardziej wyrafinowany model — obliczają rzeczywistą średnią liczbę porównań, nie badania jakiegoś konkretnego uruchomienia algorytmu.

Oto podsumowanie technik zastosowanych do uzyskania każdego z programów:

- Listingi 3.2, 3.4, 3.7 — fundamentalna zmiana definicji problemu.
- Listingi 3.5, 3.6, 3.12 — nieduża zmiana definicji funkcji.
- Listing 3.8 — nowa struktura danych implementująca programowanie dynamiczne.

Techniki te są typowe. Program często można uprościć poprzez odpowiedzenie sobie na pytanie, jaki problem tak naprawdę trzeba rozwiązać oraz czy jest funkcja lepiej nadająca się do rozwiązania tego problemu.

Kiedy po raz pierwszy przedstawiłem tę analizę studentom, program w końcu skurczył się do 0 wierszy kodu i zniknął w tumanie matematycznego kurzu. Kod z listingu 3.7 można przedstawić za pomocą następującej zależności rekurencyjnej:

$$C_0 = 0 \quad C_n = (n-1) + (1/n) \sum_{1 \leq i \leq n} C_{i-1} + C_{n-i}$$

Jest to dokładnie metoda zastosowana przez Hoare’a i później przedstawiona przez D. E. Knutha w jego klasycznej monografii *Sztuka programowania. Tom 3: Sortowanie i wyszukiwanie*⁵. Sztuczki programistyczne polegające na wprowadzaniu odpowiedników i zastosowaniu symetrii, dzięki którym powstał kod zaprezentowany na listingu 3.10, umożliwiły uproszczenie części rekurencyjnej do następującej postaci:

$$C_n = n - 1 + (2/n) \sum_{0 \leq i \leq n-1} C_i$$

Technika Knutha polegająca na usunięciu symbolu sumy daje w wyniku (mniej więcej) kod widoczny na listingu 3.11, który można zastąpić układem dwóch zależności rekurencyjnych z dwiema niewiadomymi.

$$C_0 = 0 \quad S_0 = 0 \quad S_n = S_{n-1} + 2C_{n-1} \quad C_n = n - 1 + S_n / n$$

⁵ Wydawnictwa Naukowo-Techniczne, Warszawa 2002 — *przyp. red.*

Knuth uzyskuje wynik dzięki zastosowaniu matematycznej techniki czynnika sumującego (ang. *summing factor*):

$$C_n = (n+1)(2H_{n+1} - 2) - 2n \sim 1,386n \ln n$$

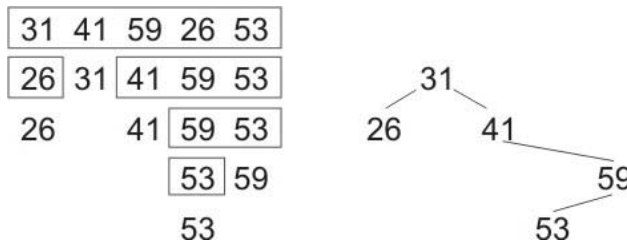
gdzie H_n oznacza n -tą liczbę harmoniczną $1 + \frac{1}{2} + \frac{1}{3} + \dots + \frac{1}{n}$. W ten sposób gładko przeszliśmy od eksperymentowania na programie poprzez wzbogacanie go dogłębną analizą do kompletnie matematycznej analizy jego działania.

Na tej formule kończymy naszą przygodę. Poszliśmy za słynną radą Einsteina, która brzmi: „Upraszczaj wszystko jak to tylko możliwe i ani trochę bardziej”.

Dodatkowa analiza

Słynne stwierdzenie Goethego mówi, że architektura to zamrożona muzyka. W dokładnie tym samym sensie twierdzę, że struktury danych to zamrożone algorytmy. Jeśli zamrozimy algorytm Quicksort, otrzymamy strukturę danych binarnego drzewa poszukiwań. Struktura ta jest zaprezentowana w publikacji Knutha. Czas jej wykonania został przeanalizowany za pomocą relacji rekurencyjnej podobnej do tej występującej w Quicksort.

Gdybyśmy chcieli przeanalizować średni koszt wstawienia elementu do binarnego drzewa wyszukiwania, moglibyśmy zacząć od kodu, który następnie wzbogacilibyśmy o zliczanie porównań. Potem moglibyśmy przeprowadzić eksperymenty na zgromadzonych danych. Następnie moglibyśmy uprościć kod (i zwiększyć jego funkcjonalność) w sposób bardzo podobny do tego z poprzedniego podrozdziału. Prostsza metoda polega na zdefiniowaniu nowego algorytmu Quicksort z użyciem metody **idealnego podziału** pozostawiającej elementy w tej samej względnej kolejności po obu stronach. Taki algorytm Quicksort jest izomorficzny z binarnymi drzewami poszukiwań, co widać na rysunku 3.1.



Rysunek 3.1. Algorytm Quicksort z idealnym podziałem i odpowiadające mu drzewo binarne poszukiwań

Ramki po lewej stronie prezentują algorytm Quicksort z idealnym podziałem w trakcie działania. Graf po prawej stronie przedstawia odpowiadające mu drzewo binarne, które zostało zbudowane z tych samych danych wejściowych. Oba te procesy wykonują nie tylko tę samą liczbę porównań, ale dokładnie takie same ich zestawy. A zatem nasza poprzednia analiza w celu sprawdzenia średniej efektywności randomizującego algorytmu Quicksort, działającego na zestawie unikatowych elementów, daje nam średnią liczbę porównań do wstawienia do binarnego drzewa wyszukiwań losowo ustawionych unikatowych elementów.

Co to jest pisanie

Tworząc kody z listingów od 3.2 do 3.12, najpierw zapisałem je w swoich notatkach, następnie na tablicy dla studentów i w końcu na kartkach tego rozdziału. Programy te powstawały stopniowo. Spędziłem dużą ilość czasu nad ich analizą i jestem przekonany o tym, że nie zawierają błędów. Jednak poza implementacją w arkuszu kalkulacyjnym przykładu z listingu 3.11 nigdy nie uruchomiłem żadnego z tych programów jako programu komputerowego.

W ciągu dwudziestu lat pracy w Bell Labs miałem okazję uczyć się od wielu nauczycieli (zwłaszcza od Briana Kernighana, którego rozdział o nauczaniu programowania pojawia się jako pierwszy w tej książce). Nauczono mnie, że pisanie programu do użytku publicznego to coś więcej niż tylko wpisywanie symboli. Po napisaniu kodu programu uruchamia się go w kilku przypadkach testowych, następnie buduje szczegółowe rusztowanie, sterowniki i bibliotekę przypadków systematycznie na nim uruchamianych. W idealnej sytuacji skompilowany kod źródłowy jest „włączany w tekst” bez interwencji człowieka. Przykład z listingu 3.1 (i wszystkie kody w książce *Perelki programowania*) napisałem właśnie w tym duchu.

Punktem honoru dla mnie było trzymanie się tytułu i nieimplementowanie przykładów z listingów 3.2 do 3.12. Prawie czterdzieści lat programowania komputerów nauczyło mnie głębokiego szacunku dla trudności tego rzemiosła (mówiąc dokładniej, panicznego strachu przed błędami). Skapitulowałem, implementując kod z listingu 3.11 w arkuszu kalkulacyjnym, i dorzuciłem dodatkową kolumnę, która dała zamkniętą formę rozwiązania. Wyobraź sobie moją radość, kiedy zobaczyłem, że dokładnie do siebie pasują! Tak więc prezentuję światu te piękne nienapisane programy z pewną dozą pewności, że są poprawne, ale w głębi będąc boleśnie świadom, że mogą zawierać jakieś nieodkryte błędy. Mam nadzieję, że głębokie piękno, które w nich widzę, nie zostanie przekreślone przez jakieś powierzchowne skazy.

Prezentując niepewnie te nienapisane programy, pocieszam się spostrzeżeniem Alana Perlisa, który powiedział: „Czy jest możliwe, że oprogramowanie nie jest podobne do niczego innego, że jest skazane na wyrzucenie, że cała filozofia polega na tym, aby postrzegać je jako mydlaną bańkę?”.

Zakończenie

Piękno ma wiele źródeł. Ten rozdział koncentruje się na pięknie zdobywanym dzięki prostocie, elegancji i zwięzłości. Poniższe stwierdzenia wyrażają tę najistotniejszą myśl:

- Staraj się dodawać funkcje poprzez usuwanie kodu.
- Projektant może uznać, że osiągnął perfekcję, nie wtedy, kiedy nie pozostało już nic do dodania, ale wtedy, gdy nie można już nic odjąć (Saint-Exupéry).
- W oprogramowaniu najpiękniejszego kodu, najpiękniejszych funkcji i najpiękniejszych programów czasami w ogóle nie ma.
- Piszący szybko piszą zwięźle. Pomijają niepotrzebne słowa (Strunk i White).

- Najtańsze, najszybsze i najbardziej niezawodne komponenty systemu komputerowego to te, których nie ma (Bell).
- Dąż do robienia coraz więcej za pomocą coraz mniejszej ilości kodu.
- Gdybym miał więcej czasu, napisałbym Ci krótszy list (Pascal).
- Paradoks wynalazcy: bardziej ambitny plan może mieć więcej szans na powodzenie (Pólya).
- Prostota nie występuje przed złożonością, ale jest jej następstwem (Perlis).
- Mniej oznacza więcej (Browning).
- Upraszczaj wszystko jak to tylko możliwe i ani trochę bardziej (Einstein).
- Oprogramowanie powinno być czasami postrzegane jako mydlana bańka (Perlis).
- Szukaj piękna w prostocie.

Na tym kończy się ta lekcja. Idź zatem i postępuj, jak tu napisano.

Dla tych, którzy potrzebują bardziej konkretnych wskazówek, poniżej przedstawiam listę koncepcji podzielonych na trzy kategorie.

Analiza programów

Jednym ze sposobów na zyskanie wglądu w działanie programu jest odpowiednie wyposażenie go i uruchomienie na reprezentatywnej próbie danych, jak w przykładzie z listingu 3.2. Często jednak bardziej niż całym programem zajmujemy się jednym jego fragmentem. W tym przypadku zajmowaliśmy się tylko średnią liczbą porównań wykonywanych przez Quicksort, a pomijaliśmy wiele innych aspektów. Sedgewick⁶ bada takie zagadnienia, jak wymagana przez niego przestrzeń i wiele innych komponentów wykonywania różnych wariantów Quicksort. Koncentrując się na najważniejszych problemach, możemy (przez chwilę) zapomnieć o innych aspektach programu. W jednym z moich artykułów, *A Case Study In Applied Algorithm Design*⁷, opisuję, jak zetknąłem się z problemem oszacowania wydajności **heurystyki paskowej** (ang. *strip heuristic*) do znalezienia przybliżonej drogi akwizytora przez N punktów w określonym kwadracie. Ocenilem, że kompletny program do rozwiązania tego zadania zajmie około 100 wierszy kodu. Po kilku etapach podobnych do tych opisanych powyżej uzyskałem dwunastowierszową symulację o znacznie większej dokładności (a po zakończeniu mojej małej symulacji odkryłem, że Beardwood i inni autorzy⁸ wyrazili moją symulację w postaci podwójnej liczby całkowitej, a więc rozwiązali matematycznie ten problem około dwudziestu lat wcześniej).

⁶ Robert Sedgewick, *The Analysis of Quicksort programs*, „Acta Informatica”, Vol. 7 — *przyp. red.*

⁷ „IEEE Computer”, Vol. 17, No. 2 — *przyp. red.*

⁸ J. Beardwood, J. H. Halton, J. M. Hammersley, *The Shortest Path Through Many Points*, „Proc. Cambridge Philosophical Soc.”, Vol. 55 — *przyp. red.*

Małe fragmenty kodu

Uważam, że programowanie komputerów to umiejętność praktyczna, i zgadzam się z Pólyą, iż „zdolności praktyczne nabywamy poprzez naśladownictwo i praktykę”. Programiści, którzy pragną pisać piękny kod, powinni zatem czytać piękne programy i naśladować zastosowane w nich techniki we własnych. Według mnie do takich ćwiczeń najlepiej nadają się niewielkie fragmenty kodu, składające się z 10 do 25 wierszy kodu. Przygotowywanie drugiego wydania książki *Perelki programowania* wymagało mnóstwa pacy, ale było też bardzo zabawne. Implementowałem każdy fragment kodu i pracowałem nad nim, aby zredukować jego rozmiar do niezbędnego minimum. Mam nadzieję, że inni będą mieli tyle samo radości z czytania tego kodu co ja z jego pisania.

Systemy oprogramowania

Opisałem niezwykle szczegółowo jedno małe zadanie. Wydaje mi się, że świetność tych zasad nie bierze się z małych fragmentów kodu, a z dużych programów i wielkich systemów komputerowych. Parnas opisuje techniki redukcji systemu do niezbędnego minimum⁹. Stosując je, nie zapomnij rady Toma Duffa: „Podkradaj kod, kiedy tylko jest taka możliwość”.

Podziękowania

Dziękuję za wnikliwe komentarze Danowi Bentleyowi, Brianowi Kernighanowi, Andy’emu Oramowi i Davidowi Weissowi.

⁹ David L. Parnas, *Designing software for ease of extension and contraction*, „IEEE T. Software Engineering”, Vol. 5, No. 2 — *przyp. red.*

.NET Common Language Runtime, 132
 __dict__, 315

A

abstrakcja, 43
 iterator, 324
 abstrakcja składniowa, 425
 makra Lisp, 426
 makra preprocesora C, 425
 syntax-rules, 428
 warunek higieniczności dla rozszerzania makr, 427
 Acceptance Testing Framework, 96
 ACE, 448, 454
 ACE_Handle_Set, 459
 Adapter, 448
 adnotacje, 206
 adresowanie otwarte, 316
 advice, 524
 AFL, 529
 Agile methodologies, 110
 algorytmy
 algebra liniowa, 248
 binarySearch, 116
 dziel i zwyciężaj, 48
 Euklidesa, 558
 filtr cyfrowy, 136
 gęsta algebra liniowa, 253
 higieniczne rozszerzanie makr, 428
 macierzowe, 248
 ogólne, 146
 podzielone na bloki, 248
 Quicksort, 48
 rozwijanie, 431
 rsync, 29
 współliniowość, 566
 wyszukujące, 67
 zliczające słowa, 400
 analiza błędów, 267
 analiza programów, 58
 analizator składniowy, 306
 AND, 126
 annotation, 206
 Ant, 89
 Apache Log4J, 347
 API, 45
 API JavaMail, 98
 API JDOM, 79, 80
 API JDOM 3, 86
 API Perl XS, 184
 aplikacje sieciowe, 228, 447
 Application Programming Interface, 447
 apply_textdelta(), 41
 architektura
 komputer, 248
 oszczędzająca nakłady, 447
 REST, 469
 RISC, 79, 249
 sieciowe usługi rejestrowania, 449
 wielokrotnego użytku, 448
 zorientowana na usługi, 340
 argumenty funkcji, 300
 arkusze właściwości, 361
 ArrayIndexOutOfBoundsException, 108
 Arrays, 116
 AS/400, 469, 470
 asembler, 126
 asercje
 blokady, 306
 JUnit, 120
 ASSERT_VOP_ELOCKED(), 306
 ASSERT_VOP_UNLOCKED(), 306
 assignment, 155
 AsTeR, 529
 AsTeR, Audio System For Technical Readings, 521
 asymetria w przepływie danych, 503
 Atom, 540
 atomically act, 410
 Audio Formatting Language, 529
 audyt modułu, 197
 Aural CSS, 529, 530, 535
 automatyczne wyszukiwanie przyczyny awarii, 486
 awk, 20, 70

B

- backtick operator, 198
- badania użyteczności, 182
- base pair, 206
- Basic Linear Algebra Communication Subprograms, 261
- Basic Multilingual Plane, 89
- baton, 36
- baza danych, 277
- Benchmark, 201
- bezpieczna komunikacja, 177
- bezpieczne przesyłanie wiadomości, 180
- białe znaki, 550
- biblioteki
 - CERN, 272
 - LAPACK, 272
- binarySearch, 113, 116, 121
- binarySearchComparisonCount, 121
- Bio::Graphics, 206
 - dodawanie nowych glifów, 231
 - fabryki glifów, 219
 - gęstość własności, 209
 - historijka, 210
 - interaktywne aplikacje sieciowe, 209
 - klasy obiektowe, 216
 - niezależność od formatu graficznego, 209
 - niezależność od schematów baz danych, 210
 - obsługa obrazów nadających się do publikacji, 230
 - opcje dynamiczne, 224
 - otwarta natura problemu, 208
 - powiększanie semantyczne, 225
 - proces projektowania, 210
 - projektowanie sposobu interakcji dewelopera z modulem, 210
 - przetwarzanie opcji, 218
 - rozszerzanie modułu, 228
 - skala, 209
 - ścieżki, 212
 - ustawianie opcji, 214
 - wspieranie programistów sieciowych, 228
 - wymagania, 208
 - wywołanie zwrotne, 227
 - zwracane dane, 207
- Bio::Graphics::Glyph, 232
- Bio::Graphics::Glyph::Factory, 219
- Bio::Graphics::Panel, 210, 216
- Bio::Graphics::Track, 216
- Bio::SeqFeature::Generic, 224
- Bio::SeqFeatureI, 224
- bioinformatyka, 205
- BioMoby, 471
- BioPerl, 205, 206, 213
- BitBlt, 126, 128, 132
- BLACS, 261
- BLAS, 253

- block, 158
- block-partitioned algorithms, 248
- blokady, 404, 405, 423
 - rogatkowa, 376
 - wątki, 375
- blokowanie, 413
 - wątki, 372
 - zasoby, 372
- błędy wyczerpania pamięci, 277
- BMP, 89
- BNF, 81
- boundary testing, 106
- bramki, 416
- branch, 140
- branch if less than, 142
- branching, 140
- brush, 128
- BT Trade, 364
- bufor wpisywania, 515
- buforowanie, 91
- Business Template, 364
- bycie podręcznikowym, 547

C

- C#, 131
- CA, 180
- Carry Save Adder, 171
- CAS, 412
- cd9660_read(), 299
- cele ERP, 358
- cele projektowe, 185
- CERN, 272
 - algorytmy, 273
- Certification Authority, 180
- CGI, 237
- chaining, 249
- Character, 82
 - isDigit(), 82, 83
 - isLetterOrDigit(), 82, 83
- checkXMLName(), 82, 83, 86
- CIP, 337
- CIP Middleware Monitor Utility, 352
- cmaild, 188
- CMF, 357, 358, 360
- CMF Types, 360
- Collaborative Information Portal, 337, 338
- Commercial off-the-shell, 340
- Common Lisp, 425
- commoning, 172
- Compare-And-Swap, 412
- constant, 155
- container_of(), 289
- Content Management Framework, 358
- cookie, 238

- core dump, 378
- corner cases, 112
- COTS, 340
- CPAN, 184
- CreateProcess(), 464
- Crypt::PGP5, 189
- Cryptonite, 178, 182
 - audyt modułu Crypt::GPG, 197
 - bezpieczne przysyłanie wiadomości, 180
 - cele projektowe, 185
 - cmaild, 188
 - Crypt::GPG, 191
 - Crypt::PGP, 190
 - Cryptonite::Mail::Config, 196
 - Cryptonite::Mail::Service, 196
 - DBD::Replication, 192
 - decyzje, 185
 - deszyfracja, 192
 - działający prototyp, 189
 - Edit Key, 194
 - folder cieni, 192
 - IMAP, 192
 - IPC::Run, 198
 - Mail::Cclient, 201
 - Mail::Folder, 192
 - Mail::Folder::Shadow, 192, 200
 - Mail::Folder::SQL, 192
 - mbox, 189
 - OpenPGP, 198
 - Params::Validate, 196
 - Persistence::Database::SQL, 190
 - Persistence::Object::Postgres, 190, 191
 - Persistence::Object::Simple, 190, 191
 - początkowy projekt systemu, 186
 - poczta odbierana, 189
 - projekt systemu, 186
 - prywatność komunikacji, 202
 - przechowywanie wiadomości e-mail, 191
 - przejście od prototypu do skalowalnego produktu, 190
 - reorganizacja kontenera wiadomości, 191
 - Replication::Recall, 192, 201
 - replikacja wiadomości e-mail, 192
 - serializacja, 187
 - szybkość działania, 201
 - trwałość deszyfracji, 192
 - uwierzytelnienie z kluczem, 180
 - użyteczność, 181
 - zabezpieczanie kodu, 195
 - zarządzanie kluczami, 194
 - zestaw testów, 188
- Cryptonite Mail Daemon, 188
- Cryptonite::Mail::Service, 186
- CSA, 171
- cyberpunk, 202
- cyfrowe filtry obrazu, 132

- cykl życia grup procesów, 465
- czas trwania testów, 118
- czynniki sumujący, 56
- czytanie programów, 495
- czytelność kodu, 114, 309
- czynniki kanałów, 540

D

- dane, 125
- dane z pamiętnika sieciowego, 62
- data display debugger, 482
- DAXPY, 254
- DBD::SQLite, 189
- DBI, 189
- DCOM, 391
- dd(), 488
- ddchange, 493
- ddd, 482
- debuger, 482
 - ddd, 482
- debugowanie, 378, 481
 - automatyczne wyszukiwanie przyczyny awarii, 486
 - dd(), 488
 - ddd, 482
 - debuger, 482, 485
 - efektywność, 481
 - gdb, 482
 - hipoteza przyczyny awarii, 484
 - in situ, 378
 - metoda naukowa, 483
 - odtworzenie awarii, 490
 - poawaryjne, 378
 - post mortem, 378
 - przewidywania, 484
 - systematyczny proces, 483
 - wyszukiwanie przyczyn, 483
 - zasady, 484
- debugowanie delta, 488
 - ddchange, 493
 - minimalizacja wejścia, 490
 - porównanie stanów programu, 491
 - problem prototypu, 493
 - różnice między dwoma stanami, 491
 - stan programu, 491
 - wyszukiwanie przyczyn w danych wejściowych, 490
- DECTalk, 522
- DECTalk Express, 522
- definiowanie interfejsu usługi, 471
- dekompozycja LU, 248, 280
- delta debugging, 488
- delta editor, 29
- demarshalling, 391
- demultipleksacja, 307
- dentry, 294

- depot, 374
 - deskryptor tablicy, 263
 - deszyfracja, 192
 - devfs, 286
 - device, 286, 291
 - DGEFA, 252, 254
 - DGEMM, 257
 - DGETF2, 257, 258
 - DGETRF, 255
 - dialog z użytkownikiem przez internet, 237
 - dictionary, 67
 - DiffMerge, 546, 547, 550, 552
 - digital filters, 132
 - długie kliknięcie, 507
 - długość wierszy tekstu, 547
 - DNA, 206
 - dobry projekt, 29
 - Document, 86
 - document-centric, 357
 - dodawanie funkcji poprzez usuwanie kodu, 47
 - dokumenty
 - XHTML, 91
 - XML, 79, 479
 - DOM, 479
 - Domain Specific Language, 498
 - doPost(), 472
 - dostarczanie informacji, 61
 - dostęp do informacji online, 534
 - do-while, 23
 - DRY, 497
 - DSCAL, 254
 - DTRSM, 257
 - Duff's Device, 29
 - DynamicMethod, 139, 146
 - dynamyczna rekonfiguracja, 354
 - dynamiczne obiekty, 149
 - dynamiczne wybieranie funkcji przechowującej, 315
 - dynamiczne zasiedlanie drzewa, 510
 - działający prototyp, 189
 - dziedziczenie priorytetów, 373
 - dziedziczenie prototypowe, 149
 - dział i zwyciężaj, 48, 164, 283
 - zliczanie bitów w stanie wysokim, 165
 - dziennik transakcji, 412
 - dźwiękowe środowisko pracy, 521
 - Emacspeak, 521
 - wyjście mówione, 522
- E**
- ed, 20
 - EDEADLK, 382
 - edytor delty, 29, 41
 - interfejs, 35
 - efekty uboczne, 407, 409
 - efektywność debugowania, 481
 - egrep, 20
 - EJB, 341, 344
 - eksponowanie obiektów OpenPGP, 182
 - ekspresywna notacja obiektowa, 148
 - ekstremalne przypadki testowe, 112
 - elastyczność kodu, 499
 - elegancki kod, 48
 - element, 359
 - eliminacja Gaussa, 247
 - analiza błędów, 267
 - DGETRF, 255
 - faktoryzacja dla wykonywania wielowątkowego, 265
 - język MATLAB, 251
 - LAPACK, 256
 - liczba operacji, 267
 - LINPACK, 253
 - metoda dekompozycyjna, 250
 - PBLAS, 263
 - rekursywna dekompozycja LU, 257
 - ScaLAPACK, 260, 261
 - eliminacja niepotrzebnych transferów danych przez sieć, 29
 - elocutor, 501
 - asymetria w przepływie danych, 503
 - bufor wpisywania, 515
 - Common Words, 514
 - częste słowa, 514
 - długie kliknięcie, 507
 - drzewo, 506
 - dynamiczne zasiedlanie drzewa, 510
 - edycja, 515
 - Favorites, 514
 - grupowanie słów, 504
 - implementacja pamięci podręcznej, 513
 - interfejs wejściowy, 505
 - makra, 517
 - model projektu, 502
 - następne słowo, 511
 - Next Word, 511
 - pamięć podręczna, 513
 - pobieranie, 518
 - proste wpisywanie, 511
 - przewidywanie, 511
 - przewijanie, 515
 - Replace, 512
 - schowek, 517
 - szablony, 512
 - śledzenie ścieżek, 515
 - Templates, 512
 - TreeView, 506
 - układ ekranu, 503
 - ulubione, 514
 - uzupełnianie słów, 511
 - wejście binarne, 506
 - Word Completion, 511

- wydajność interfejsu użytkownika, 518
- wyszukiwanie, 517
- zastępowanie, 512
- Emacs Calendar, 532
- Emacs W3, 535
- Emacspeak, 521
 - advice, 524
 - Aural CSS, 529
 - czytnik kanałów, 540
 - dostęp do informacji online, 534
 - emacspeak-auditory-icon, 532
 - emacspeak-calendar, 533
 - emacspeak-calendar-speak-date, 533, 534
 - emacspeak-minibuffer-setup-hook, 531
 - emacspeak-speak-line, 524
 - emacspeak-url-template, 539
 - emacspeak-w3-extract-table-by-match, 537
 - emacspeak-websearch, 535, 536, 539
 - formatowane dźwięku, 526
 - formatowanie wyjścia dźwiękowego na podstawie słuchowych list wyświetlania, 528
 - generowanie bogatego wyjścia mówionego, 525
 - gramatyka zawartości bufora, 525
 - ikony dźwiękowe, 530
 - implementacja, 523
 - internetowy wiersz poleceń, 539
 - kalendarz, 532
 - minibuffer-setup-hook, 531
 - odtwarzanie ikon dźwiękowych podczas wypowiedzania zawartości, 532
 - personality, 527
 - put-text-property, 526
 - RSS, 540
 - semantyka zależna od kontekstu, 532
 - serwer mowy, 522
 - stylizowanie wyjścia mówionego, 529
 - szablony URL, 539
 - tts-format-text-and-speak, 528, 529
 - tts-speak, 528
 - tworzenie dźwiękowych list wyświetlania, 526
 - voice-lock, 526
 - włączanie mowy w Emacsie, 523
 - wyszukiwanie zorientowane na zadania, 535
- emacspeak-auditory-icon, 532
- emacspeak-calendar, 533
- emacspeak-calendar-speak-date, 533, 534
- emacspeak-minibuffer-setup-hook, 531
- emacspeak-url-template, 539
- emacspeak-w3-extract-table-by-match, 537
- emacspeak-websearch, 535, 536, 539
- emacspeak-websearch-yahoo-map-directions-get-locations, 537
- embedded systems, 293
- Enterprise, 293
- Enterprise JavaBeans, 341
- Enterprise Resource Planning, 357
- EPR5 RAD, 366
- ERP5, 357, 358
 - arkusze właściwości, 361
 - BT Trade, 364
 - CMF, 357, 358, 360
 - CMF Types, 360
 - DCWorkflow, 358
 - egzemplarz zasobu, 359
 - element, 359
 - GUI, 366
 - implementacja zachowania zadań, 367
 - Item, 359
 - kategorie podstawowe, 363
 - Movement, 359
 - Node, 359
 - obieg Task, 367
 - obieg Task Report, 369
 - Path, 359
 - pisanie kodu, 365
 - Portal, 362
 - przemieszczenie, 359
 - Resource, 359
 - szablon biznesowy, 364
 - ścieżka, 359
 - Task, 367
 - UBM, 359
 - węzeł, 359
 - XML, 358
 - założenia projektu, 364
 - zasób, 359
 - ZODB, 358, 359
 - Zope, 360
 - ZPT, 358
 - zunifikowany model biznesowy, 359
- ERP5 Project, 364
- eseje, 495
- evaluation stack, 139
- ewoluowanie kodu wraz ze sprzętem, 247
- Exclusive-OR, 129
- exec*(), 464
- executeQuery(), 477
- expression, 152
- eXtreme Programming, 110

F

- fabryka, 473
- faktoryzacja
 - LU, 279
 - panelowa, 265
- falszywe zakleszczenie, 382
- fasada, 448, 454
 - ACE, 455
- features, 208
- ffs, 300

- fgrep, 20
- file baton, 41
- FilterMethodCS(), 134, 135, 137, 146
- FilterMethodTL(), 134, 138, 145, 146
- filtry, 242
 - grafika cyfrowa, 132
 - wyostrzające, 133
- FIT, 95
- fixture, 96
- Fixture, 99
- folder cieni, 192
- fork(), 464
- forkIO(), 408
- format XML, 358
- formatowane dźwięku, 526
- formaty wpisów dziennika, 450
- Fortran, 252
- fragmentowanie, 321
- Framework for Integrated Test, 95
 - ActionFixture, 97
 - architektura, 98
 - dokumenty, 96
 - fixture, 96
 - Fixture, 97, 99
 - otwarte środowisko, 99
 - Parse, 97
 - parser HTML, 100
 - projektowanie środowiska, 98
 - przetwarzanie kodu HTML, 100
 - rdzeń, 97
 - testy, 96
 - TypeAdapter, 97
- frameworki, 448, 452
- free(), 318
- FreeBSD, 307
- funkcja partycjonująca, 398
- funkcje, 160, 245
- funkcjonalność interfejsu użytkownika, 182

G

- gałęzie wydań DiffMerge, 552
- Gate, 416
- GBrowse, 229
- GD, 230
- gdb, 482
- Gene Sorter, 235
 - advFilter, 242
 - filterControls, 242
 - filtry, 242
 - interfejs użytkownika, 236
 - podtrzymywanie dialogu z użytkownikiem przez internet, 237
 - polimorfizm, 239
 - teoria pięknego kodu, 243

- generator XML, 479
- generowanie
 - bogate wyjście mówione, 525
 - kod do przetwarzania obrazów, 125
 - kod w locie, 126
- Genes, 209
- genom, 206
- getResponseXML(), 479
- gęstość własności, 209
- GHC, 421
- Glasgow Haskell Compiler, 421
- głęboko wcięty kod, 548
- gniazda, 447
- GNU debugger, 482
- Google, 75, 76
- Google Maps, 537, 538
- goto, 140
- graficzny interfejs użytkownika, 503
- grafika, 209
 - SVG, 230
- gramatyka BNF, 81, 83
- Grand Unified Debugger Emacs, 532
- grep, 20, 393
- grupowanie słów, 504
- GUI, 366
- gwarancje kolejności, 399

H

- haker, 202
- Hash, 67
- hash table, 67
- hash(), 399
- HashMap, 67
- Haskell, 404, 406, 422
 - akcje, 407, 409, 422
 - atomically act, 410
 - efekty uboczne, 407, 409
 - forkIO(), 408
 - kompilacja programu, 421
 - operacje STM, 414
 - pamięć transakcyjna, 423
 - STM, 410
 - struktury sterujące, definiowane przez użytkownika, 409
 - transakcje, 410
 - uruchamianie programu, 421
 - wątki, 408
 - wejście-wyjście, 407
- Heap Sort, 49
- Hello World, 496
- heurystyka paskowa, 58
- higieniczne rozszerzanie makr, 427
- hook methods, 453
- hot swapping, 355

HotkeyAdaptor, 474
HotkeyAdaptorFactory, 474
HTML, 100
HttpClient, 475

I

IDAMAX, 254
IDE, 185, 497
idealny podział, 56
identyfikator URI, 92
IETF, 180
if-else, 140
if-then-else, 548
ikony dźwiękowe, 530
IL Disassembler, 135, 137
ILGenerator, 139, 146
ILP, 174
image filters, 132
ImageClip, 134
ImageFilter, 135, 138
IMAP, 192, 199
imitacja gramatyki BNF O(N), 83
implementacja, 386

- kod, 272
- pamięć podręczna, 513
- pamięć transakcyjna, 412
- rozproszony MapReduce, 394
- sekwencyjny serwer rejestrowania, 457
- słownik, 311
- Święty Mikołaj, 420
- wyszukiwanie binarne, 109

infix, 153, 156
infixr, 155
informacje pośrednie, 323
informacyjny RNA, 236
information retrieval, 76
informowanie o postępie, 29
infrastruktura klucza publicznego, 180, 203
infrastruktura warstwy pośredniczącej, 448
inode, 294
Instruction-Level Parallelism, 174
instrukcje, 157

- rozgałęziające, 140

integracja partnerów biznesowych, 469
interakcja dewelopera z modulem, 210
interaktywne aplikacje sieciowe, 209
interfejs, 29
interfejs edytora delty, 35
interfejs usługi systemu zaplecza, 472
interfejs wejściowy, 505
internetowy wiersz poleceń, 539
inventor's paradox, 51
inwersja priorytetów, 373
IPC, 449, 450, 454

IPC::Run, 198
ISO X.509, 180
isXMLCombiningChar(), 83
isXMLExtender(), 83
isXMLLetter(), 83
isXMLNameCharacter(), 83
isXMLNameStartCharacter(), 83
Item, 359
iteracyjny serwer rejestrowania, 457
Iterative_Logging_Server, 457, 461
iterator NumPy, 324, 335
iteratory, 323

J

J2EE, 340
Java, 26
Java(), 26
JavaMail, 98
JavaScript, 148
jądro

- Linux, 285
- Solaris, 372

JDOM, 79, 80, 84
JDOM 1.0, 85
język

- BioPerl, 205
- funkcyjny, 409, 422
- Haskell, 404, 422
- imperatywny, 422
- Java, 26
- JavaScript, 148
- konkretnej domeny, 305
- LISP, 148
- MATLAB, 251
- Python, 311
- Rexx, 498
- Ruby, 64, 497
- TCL, 522
- XML, 80

JIT, 79, 132
JPL, 338
JSON, 149
jump, 140
JUnit, 109, 120

- asercje, 120
- dokumentacja, 110

Just-In-Time compiler, 132
Jython, 315

K

kanały RSS, 540
Key Ring, 183
KFFD, 428, 445

Kleene, Stephen, 19
 klient mowy Emacspeak, 523
 klient poczty e-mail, 184
 klient-serwer, 522
 klucz, 66
 kobject, 291
 kod, 125, 305, 496
 generowanie w locie, 126
 XML, 79
 zarządzany, 131
 kolejność wykonywania operatorów, 147
 funkcje, 160
 instrukcje, 157
 JavaScript, 148
 literały obiektowe, 161
 literały tablicowe, 161
 operator trójargumentowy, 153
 operatory przedrostkowe, 154
 operatory przypisania, 155
 operatory wrostkowe, 152
 podejmowanie decyzji, 151
 stałe, 155
 tablica symboli, 149
 technika parsowania, 147
 technika Pratta, 148
 tokeny, 150
 zakres, 156
 kolizje, 316
 komentarze, 549
 kompilacja, 26
 kompilatory, 492
 język C#, 131
 JIT, 79, 132, 139
 komponenty EJB, 344
 komunikacja, 177
 między procesami, 449
 konstruktor danych MkGate, 417
 konstrukty pętli, 425
 konstrukty warunkowe, 425
 konta bankowe, 404
 blokady, 404
 zmiennne warunkowe, 404
 kontrola wersji, 30
 konwencje pisania kodu, 546
 konwersja niedeterministycznego automatu skończonego
 na automat deterministyczny, 25
 kopia robocza, 34
 kopiowanie binarnej tablicy wyszukiwania, 89
 kref, 292
 kref_put(), 292
 krótki kod, 279
 kryptografia, 202
 kryptografia z kluczem publicznym, 180
 Księga, 557

L

LAPACK, 248, 256, 272
 left denotation, 151
 Level-1 BLAS, 253
 lex, 306
 liczba operacji, 267
 licznik referencji, 293
 LINPACK, 248, 252, 253, 255
 Linux, 285
 Lisp, 541
 LISP, 148
 list comprehension, 419
 lista, 318
 listy składane, 419
 literały
 obiektywne, 149, 161
 tablicowe, 149, 161
 Logging_Server, 450
 lokalność, 398
 lookdict(), 319
 lookdict_string(), 319
 losowanie, 115
 ludzki genom, 235
 luźne powiązania, 342

Ł

ładowanie
 binarne tablice wyszukiwania, 90
 wielkie tablice asocjacyjne, 71
 łańcuch blokowań, 374
 łatwość użycia, 179
 łączenie operacji w łańcuchy, 249

M

m4, 425
 macierz
 gęsta, 247
 rzadka, 247
 magazines, 374
 magazyn, 374
 Mail:Cclient, 201
 Mail::Folder::Shadow, 192, 200
 Mail::Folder::SQL, 192
 Mail:IMAPClient, 201
 MailVault beta 2, 178
 make, 498
 makra, 425, 517
 makroprocesor m4, 425
 malloc(), 318
 małe fragmenty kodu, 59

- małe systemy wbudowane, 293
- małe, luźno połączone obiekty, 294
- Map(), 392, 395
- MAPICS, 470, 477
- Maple, 67
- mapowanie, 313
- MapReduce, 389, 392
 - funkcja partycjonująca, 398
 - gwarancje kolejności, 399
 - implementacja, 394
 - kopia robocza, 396
 - lokalność, 398
 - Map(), 392, 395
 - model programistyczny, 392
 - odporność na błędy, 397
 - odwrócony graf odnośników internetowych, 393
 - odwrócony indeks, 394
 - porównywanie złych rekordów, 399
 - program główny, 396
 - przepustowość sieci, 398
 - Reduce(), 392
 - relacje między procesami, 396
 - rozproszone grep, 393
 - rozproszone sortowanie, 394
 - rozszerzenia modelu, 398
 - wektor terminów dla hosta, 394
 - wyważenie obciążenia, 397
 - zadania rezerwowe, 398
- Mars Exploration Rover, 337, 338
 - architektura CIP, 341
 - architektura systemu, 340
 - architektura usługi strumieniowej, 344
 - CIP Middleware Monitor Utility, 352
 - Collaborative Information Portal, 338
 - doFileDownload(), 348
 - dynamiczna rekonfiguracja, 354
 - funkcjonalność, 343
 - getDataFile(), 350
 - Jet Propulsion Laboratory, 338
 - monitorowanie, 352
 - niezawodność, 343, 346
 - przesyłanie plików, 346
 - readDataBlock(), 348, 351
 - registerNewReader(), 349
 - rejestrwanie, 347, 352
 - removeReader(), 349
 - SOA, 341
 - solidność, 353
 - StreamerServiceBean, 347
 - usługa strumieniowa, 343
 - usługi, 341
 - warstwa kliencka, 340
 - wymagania misji, 339
 - wymiana podczas pracy, 355
 - zarządzanie czasem, 339
 - zarządzanie danymi, 340, 343
 - zarządzanie personelem, 340
- marshalling, 391
- match(), 22
- Matcher, 26
- matchhere(), 23, 24
- matchstar(), 23, 24
- MATLAB, 251
- mbox, 189
- meandry, 565
- MER, 338
- Merge Sort, 49
- messenger RNA, 236
- metaznaki, 19
- metoda dekompozycyjna, 250
- metoda szablonu, 448, 453
- metody abstrakcyjne, 453
- metody zsynchronizowane, 404
- metodyki Agile, 110
- Microsoft .NET Framework, 131
- miejsca DNA, 224
- MIMD, 260
- MIME, 191
- minibuffer-setup-hook, 531
- minimalizacja wejścia, 490
- MkGate, 417
- MLB, 539
- model sterowników jądra systemu Linux, 285
- model współbieżności, 450
- modularność, 342, 423
- modyfikowalne zmienne, 408
- monitorowanie, 352
- mostkowanie, 172
- Movement, 359
- mRNA, 236
- Multiple Instruction Multiple Data, 260
- multipleksacja, 307
- mutable, 408
- mutex, 372
- mutex_vector_exit(), 383

N

- następne słowo, 511
- National Public Radio, 539
- nawiasofobia, 558
- nawracanie, 25
- Neomailbox, 199
- Next Word, 511
- NFS, 303
- nierówność trójkąta, 563
- nieustanne testowanie, 485
- niezależność od formatu graficznego, 209
- niezależność od języka, 342
- niezależność od schematów baz danych, 210

- niezawodność, 343, 346
- Node, 359
- notacja
 - # {}, 68
 - Backusa-Naura, 81
- nowy SMP, 268
- NPR, 539
- ntz(), 175
- null denotation, 151
- null_bypass, 302, 303
- nullfs, 302
- NullPointerException, 112
- NUMA, 395
- numer wersji, 30
- NumPy, 321
 - fragmentowanie, 321
 - generator liczb losowych, 335
 - implementacja rozgłaszania, 327
 - interfejs iteratora, 331
 - iteracja przez wszystkie wymiary oprócz jednego, 332
 - iteracja, 323
 - iteratory, 324
 - modele pamięci, 323
 - obiekt multi-iteratora, 334
 - operacje, 322
 - projekt iteratora, 325
 - przerwanie iteratora, 326
 - PyArray_IterAllButAxis(), 333
 - PyArrayIterObject, 330
 - rozgłaszanie, 334
 - rozwój iteratora, 325
 - struktura iteratora, 329
 - śledzenie licznika iteratora, 328
 - tablice nieciągłe, 323
 - ustawianie iteratora, 327
 - wiele iteracji, 333
 - wykorzystanie iteratora, 332
- N-wymiarowe tablice, 321
 - iteratory, 323
 - modele pamięci, 323
 - operacje, 322

O

- obciążenie, 386
- obiekt fixture, 96
- objektowy framework dla oprogramowania sieciowego, 447
- obiekty generyczne, 148
- Object Oriented, 449
- obliczanie powierzchni, 567
- obsługa obrazów nadających się do publikacji, 230
- odporność na błędy, 397
- odstęp Hamminga, 175
- odtworzenie ikon dźwiękowych podczas wypowiedzania zawartości, 532
- odtworzenie awarii, 490

- odwrócony graf odnośników internetowych, 393
- odwrócony indeks, 394
- ogólna teoria pięknego kodu, 243
- OO, 449
- opakowanie, 431
- opcje dynamiczne, 224
- OpCodes, 139
- OpenPGP, 178, 180, 198
- OpenPGP/MIME, 192
- operacje
 - rastrowe, 127
 - wektor-wektor, 249
- operator, 147
 - przedrostkowe, 154
 - przypisanie, 155
 - wrostkowe, 152
 - wsteczny cudzysłów, 198
- oprogramowanie, 371
 - objektowe, 449
 - wielokrotnego użytku dla aplikacji sieciowych, 447
- oprogramowanie sieciowe, 447
 - ACE, 448
 - API, 447
 - architektura wielokrotnego użytku, 448
 - biblioteki Javy dla programowania sieciowego, 448
 - frameworki, 448
 - gniazda, 447
 - projekt obiektowy, 448
 - trudności przypadkowe, 447
 - usługa rejestrowania, 449
 - wątki, 447
 - Windows, 447
 - wzorce, 448
- optymalizacja, 69, 277
- optymistyczne wykonanie, 412
- organ certyfikacji, 180
- orzeczenia, 76
- oszczędności, 280
- otwarta natura problemu, 208
- otwarte środowisko, 99

P

- PageRank, 77
- pakiet wyrażeń regularnych, 21
- pamięć
 - asocjacyjna, 66
 - dynamiczna, 277
 - podręczna, 504, 513
 - skojarzeniowa, 66
 - współdzielona, 403, 449
- pamięć transakcyjna, 406, 423
 - implementacja, 412
- pamiętnik sieciowy, 62
- panel factorization, 265

panika jądra, 382
 paradoks wynalazcy, 51
 Params::Validate, 196
 parasitic DNA, 206
 Parse, 101, 102
 parser, 80

- HTML, 100
- XML, 80, 86

 parsowanie, 147, 440

- XML, 476

 partycjonowany magazyn, 390
 partycjonowany procesy, 391
 Path, 359
 pattern, 128
 Pattern, 26
 PBLAS, 263
 PDGETRF, 260
 PDSxa, 129
 Perl, 20, 184
 Persistence::Object::Module, 189
 pęk kluczy, 183
 pętle, 23, 425
 PGP, 181, 189
 PGPSDK, 189
 piękna współbieżność, 403
 piękne debugowanie, 481
 piękne testy, 105
 piękno oszczędności, 280
 piękno prostoty, 279
 piękno przepływu sterowania, 283
 piękno wewnętrzne, 278
 piękno zwięzłości, 279
 piękny kod, 271, 279, 495, 497
 piękny projekt, 271
 PKI, 180, 203
 platforma uruchomieniowa .NET, 132
 platforma Zope, 360
 pliki

- cookie, 238
- źródłowe, 550

 poczta e-mail, 98
 podejście zorientowane na dokumenty, 357
 pod-podwłasności, 208
 podtrzymywanie dialogu z użytkownikiem

- przez internet, 237

 podwłasności, 208
 podział systemu plików na warstwy, 300, 303
 polimorfizm, 239
 pool, 35
 poprawność oprogramowania, 371
 population count, 163, 175
 porównywanie liczby ustawionych bitów

- w dwóch słowach, 169

 portal, 357
 Portal, 362
 poruszanie się po kodzie, 549
 POSIX, 464
 post mortem, 378
 posting, 76
 pośredniość, 297

- czytelność kodu, 309

 powielacz binarny, 171
 powierzchnia, 567
 powiększanie semantyczne, 225
 powtórne wykonanie, 412
 PPC_Logging_Server, 466
 prefix, 156, 161
 prezentacja różnic pomiędzy drzewami, 34
 problemy

- inwersja priorytetów, 373
- macierze gęste, 247
- prototyp, 493
- Święty Mikołaj, 414
- wspólniowość, 559

 procedury sterujące, 279
 proces-dla-połączenia, 464
 procesor wektorowy, 249
 Process, 465
 Process_Manager, 465
 Process_Options, 465
 procesy, 464
 program sekwencyjny, 408
 programowanie, 20, 243

- dynamiczne, 52
- ekstremalne, 110
- modularne, 406
- rozproszone, 389
- sieciowe, 448
- wspomagane przez testy, 111

 projekt, 271

- RAD, 519
- system, 186

 projektowanie, 185

- maksymalna giętkość, 357
- sposób interakcji dewelopera z modulem, 210
- środowisko, 98
- weryfikatory XML, 79
- wydajne algorytmy algebry liniowej, 248

 proste rozwiązania, 29
 prostota, 279, 498
 protokoły

- e-biznesowe, 475
- IETF, 180
- NFS, 303

 prototyp, 179, 189
 Prywatność komunikacji, 202
 przechowywanie informacji, 61

- informacje pośrednie, 323

 przechwycenie odwołania do zmiennej, 426
 przejście od prototypu do skalowalnego produktu, 190

przemieszczenie, 359
 przenośne fasady, 465
 przenośny język asemblacji, 306
 przepływ sterowania, 283
 przesyłanie

- dokumenty XML, 471
- wiadomości, 180

 przeszukiwanie

- sieć, 76
- tablica, 87

 przetwarzanie

- kod HTML, 100
- opcje, 218

 przewyciężanie wcięć, 549
 przykładowe programy, 278
 przypadki specjalne, 315
 przypadki testowe, 112
 przyspieszania obliczeń, 390
 przyswajanie zmienności, 453
 pthread_mutex_lock(), 382
 ptys, 197
 Public Key Infrastructure, 180
 put_device(), 291
 PyArray_ITER_DATA(), 331
 PyArray_ITER_NEXT(), 331
 PyArray_ITER_NOTDONE(), 331
 PyArray_IterAllButAxis(), 333
 PyArray_IterNew(), 331
 PyArrayIterObject, 330
 PyDict_SetItem(), 319
 PyDictionary, 315
 PyDictObject, 313, 315
 PyStringMap, 315
 Python, 67, 311

Q

QED, 19
 Quicksort, 47

- ewolucja programu, 54
- idealny podział, 56
- paradoks wynalazcy, 51
- programowanie dynamiczne, 52
- przeniesienie inkrementacji na zewnątrz pętli, 50
- symetria, 53
- technika Knutha, 55
- zliczanie porównań, 50

 quicksort(), 48

R

RAD, 519
 rake, 498
 Rake, 498

Rakefile, 498
 raster operation, 127
 RE, 26
 RE_match(), 26
 RE_new(), 26
 Reactive_Logging_Server, 459, 461
 read(), 298
 reaktywny serwer rejestrowania, 458
 Reduce(), 392
 regexprs, 63
 rejestrowanie, 347, 352
 rekurencja, 23, 25, 323
 rekursywna dekompozycja LU, 257
 rekursywna faktoryzacja LU, 258, 259
 relacyjna baza danych, 504
 reorganizacja kontenera wiadomości, 191
 replikacja wiadomości e-mail, 192
 repozytorium, 30
 REST, 469, 470

- definiowanie interfejsu usługi, 471
- przekazywanie usługi za pomocą wzorca fabryki, 473

 revision number, 30
 REXX, 498
 RISC, 79, 164, 249
 RNA, 206
 rogatka, 372
 Rosettanet, 475
 Rosettanet E-Business, 469
 rozgałęzienia, 140

- bezwarunkowe, 140

 rozkład

- LAPACK, 256
- LU, 256

 rozmiar tablicy mieszającej słownika, 317
 rozproszenie danych, 391
 rozproszone sortowanie, 394
 rozszerzanie, 25

- Emacs o tworzenie dźwiękowych list wyświetlania, 526

 rozwój funkcjonalności interfejsu użytkownika, 182
 równania matematyczne, 271
 równoległy program zliczający występowanie słowa, 390
 równoległy system, 377
 równowaga kodu, 499
 różnica liczb ustawionych bitów w dwóch słowach, 169
 RPG, 477
 RSS, 540
 RTF, 102
 Ruby, 64, 497

S

S/MIME, 180
 samolubny DNA, 206
 SAX, 86
 SAXBuilder, 86

ScaLAPACK, 248, 260, 261, 264
 Scheme, 426
 schowek, 517
 Secure Shell, 202
 Secure Sockets Layer, 202
 sekwencje DNA, 206
 sekwencyjny serwer rejestracji, 457
 semantic zooming, 225
 semantyka zależna od kontekstu, 532
 serializacja, 187
 Service, 187
 Service Oriented Architecture, 340
 Servlet, 472
 serwer IMAP, 192
 Session, 99
 SGBSV, 273, 276, 279
 SGBTRF, 279, 280
 SGBTRS, 279
 Shadow Folder, 192
 shards, 77
 sideways sum, 163
 sieciowa usługa rejestracji, 449

- Acceptor, 455
- ACE, 454
- architektura, 449
- cechy wspólne, 452
- fasada, 454
- formaty wpisów dziennika, 450
- framework, 452
- główna pętla serwera, 453
- implementacja sekwencyjnych serwerów rejestracji, 457
- implementacja współbieżnych serwerów rejestracji, 461
- IPC, 450, 454
- iteracyjny serwer rejestracji, 457
- Iterative_Logging_Server, 457, 461
- komunikacja między procesami, 449
- Logging_Server, 450, 455
- metoda szablonu, 453
- model współbieżności, 450
- Mutex, 455
- PPC_Logging_Server, 466
- Process, 465
- projekt obiektowego frameworku serwera, 450
- przyswajanie zmienności, 453
- Reactive_Logging_Server, 459, 461
- reaktywny serwer rejestracji, 458
- serwer, 450
- serwer rejestracji w technologii
 - proces-dla-połączenia, 464
- serwer rejestracji w technologii
 - wątek-dla-połączenia, 462
- strategie blokowania, 450
- synchroniczny demultiplekser zdarzeń, 460
- Thread_Args, 463
- TPC_Logging_Server, 462, 464
- współbieżny serwer rejestracji, 461, 467
- wzorce, 450
- zorientowany obiektowo projekt frameworku serwera, 451

sieć VPN, 180
 sieć zaufania, 180
 SIMD, 166, 260, 266
 Simple Object Access Protocol, 470
 Single Instruction Multiple Data, 260
 skalowalność, 343

- kod, 277

 skalowanie do tysięcy urządzeń, 293
 skanowanie wierszy danych tekstowych, 70
 składanie odpowiedzi XML, 479
 składnia, 425
 składy, 374
 skrypty CGI, 237
 słownik, 67, 311

- __builtin__, 312
- adresowanie otwarte, 316
- dynamiczne wybieranie funkcji przechowującej, 315
- implementacja w C, 315
- implementacja w Javie, 315
- implementacja w Pythonie, 312
- items(), 318
- iteracje, 318
- keys(), 318
- klucze, 312
- kolizje, 316
- lista, 318
- lookdict(), 319
- lookdict_string(), 319
- ma_fill, 313
- ma_mask, 314
- ma_table, 314
- ma_used, 313
- mapowanie, 313
- określanie nowego rozmiaru tablic, 317
- operacje, 311
- optymalizacja dla małych haszy, 315
- pobieranie kluczy, 312
- PyDictionary, 315
- PyDictObject, 313, 315
- PyStringMap, 315
- Python, 312
- rozmiar tablicy mieszającej, 317
- tworzenie przypadków specjalnych, 315
- usuwanie klucza, 318
- values(), 318
- wartości, 312
- zmiana rozmiaru tablicy, 317
- zmiany dynamiczne, 318

- smoke testing, 106
- SMP, 249
- SOA, 340, 341
- SOAP, 187, 470, 471
- SOBJ_TYPE(), 380
- sockets, 447
- Software Transactional Memory, 403
- Solaris, 372
- solidność, 353
- sort, 68
- sort_by_value, 68
- sortowanie, 70, 394
- sprawdzanie
 - nazwy XML, 81
 - poprawność kodu, 305
- sprzętowy syntezytor mowy, 522
- SQL, 192
- SSH, 202
- SSL, 202, 449
- stałe, 155
- stan programu, 491, 492
- standard Rosettanet, 475, 476
- standardy, 342
- statement, 158
- statements, 158
- std, 157
- sterowniki jądra systemu Linux, 285
 - container_of(), 289
 - dentry, 294
 - devfs, 286
 - device, 286, 291
 - inode, 294
 - kobject, 291
 - kref, 292
 - kref_put(), 292
 - licznik referencji, 293
 - programy wielowątkowe, 290
 - put_device(), 291
 - rdzeń sterownika, 288
 - skalowanie do tysięcy urządzeń, 293
 - sysfs, 287
 - urządzenia, 285
 - usb_interface, 288, 289
 - zliczanie referencji, 290, 292
- STM, 403, 410
 - blokowanie, 413
 - operacje, 414
 - pamięć transakcyjna, 406
 - problem Świętego Mikołaja, 414
 - wybór, 413
- stmt, 158
- Store, 99
- stos ewaluacyjny, 139
- strategie
 - blokowanie, 450
 - dziel i zwyciężaj, 164
- StreamerServiceBean, 347
- StretchBlt, 126, 128
- String.charAt(), 26
- String.substring(), 26
- StringBuffer, 479
- strip heuristic, 58
- stronicowanie pamięci RAM, 277
- struktury sterujące, definiowane przez użytkownika, 409
- stylizowanie wyjścia mówionego, 529
- subfeatures, 208
- sub-subfeatures, 208
- Subversion, 29
 - apply_textdelta(), 41
 - baton, 36
 - drzewo katalogów, 32
 - edycja delty drzewa, 43
 - edytor delty, 29, 41
 - file baton, 41
 - interfejs edytora delty, 35
 - interfejsy, 34
 - klient-serwer, 34
 - kontrola wersji, 30
 - kopia robocza, 34
 - łącze do nowego drzewa, 33
 - numer wersji, 30
 - pool, 35
 - prezentacja różnic pomiędzy drzewami, 34
 - przechodzenie w górę, 32
 - repozytorium, 30
 - serwer-klient, 34
 - svn_error_t, 35
 - text delta, 35
 - transformacja drzewa, 30
 - transmisja zmian pomiędzy dwiema stronami, 34
 - tworzenie katalogu nadrzędnego, 31
 - węzły, 31
 - window handler, 35
- suma liczb ustawionych bitów w dwóch słowach, 169
- sumator pełny, 171
- sumator z przechowywaniem przeniesień, 171
- summing factor, 56
- SVG, 230
- svn_delta_editor_t, 29
- svn_error_t, 35
- svnsync, 45
- symbole wieloznaczne, 19
- synchroniczny demultiplexer zdarzeń, 460
- synchronizacja, 372
- synchronized, 404
- SyncML, 358
- syntax-case, 425
 - abstrakcyjna reprezentacja kodu źródłowego, 432
 - algorytm rozwijania, 431
 - błędy składniowe, 434
 - define-record, 432

- define-syntax, 429
- ekspander, 437
- exp, 437
- exp-lambda, 439
- exp-let, 439
- exp-letrec-syntax, 440
- exp-macro, 438
- exp-quote, 439
- forma wejścia, 430
- forma wyjścia, 430
- free-identifier=?, 441
- identyfikator szablonu, 430
- identyfikatory, 441
- if, 439, 444
- KFFD, 431
- konwersje, 442
- lambda, 439
- let, 439, 444
- letrec-syntax, 439
- make-syntax-object, 432
- manipulowanie środowiskami, 435
- obiekty składni, 432
- opakowania, 435
- or, 444
- parsowanie, 440
- porównywanie identyfikatorów, 441
- predykaty strukturalne, 434
- quote, 433, 439
- rekurencja, 430
- reprezentacje, 432
- rozpoczynanie rozwijania, 442
- s-wyrażenie, 439
- syntax-car, 441
- syntax-cdr, 441
- syntax-object?, 432
- syntax-object-expr, 432
- syntax-object-wrap, 432
- środowisko początkowe, 443
- transformator, 429
- transformatory bazowe, 439
- translacja identyfikatora, 436
- tworzenie obiektów składni, 440
- tworzenie opakowań, 435
- tworzenie wyjścia ekspandera, 433
- with-syntax, 430
- wydobywanie obiektów składniowych, 433
- wyjście ekspandera, 433
- wyrażenia lambda, 439
- syntax-rules, 428
- sysfs, 287
- system klasy Enterprise, 293
- system kontroli wersji, 29
- system korporacyjny o wysokim stopniu niezawodności, 337
- system kryptograficzny z kluczem publicznym, 180
- system operacyjny, 447

- system oprogramowania, 59
- system planowania zasobów przedsiębiorstwa, 357
- system plików, 297
 - devfs, 286
 - umapfs, 300
- system syntax-case, 428
- system syntax-rules, 428
- system wielordzeniowy, 265
- System.Reflection.Emit, 138
- szablon biznesowy, 364
- szablony URL, 539
- szukanie liniowe, 316
- szybkie tworzenie aplikacji, 519
- szybkość działania aplikacji, 201
- szybkość wykonywania kodu, 79
- szzyfrowanie, 190, 192
 - z kluczem publicznym, 203

Ś

- ścieżka, 359
 - krytyczna, 265
 - spełniająca oczekiwania, 118
- śledzenie ścieżek, 515
- środowisko dźwiękowe, 521
- środowisko wielokomputerowe, 391
- Święty Mikołaj, 414
 - implementacja, 420

T

- t_wchan, 380
- tabele, 67
- tablica rogatek, 384
- tablica symboli, 149
- tablice asocjacyjne, 67, 70
- tablice nieciągłe, 323
- tablice N-wymiarowe, 321
- tablice syntaktyczne, 525
- TCL, 522
- TDD, 111
- techniki
 - generowanie kodu w locie, 126
 - Knutha, 55
 - Pratta, 148, 152
 - rekurencja, 25
 - testowanie, 106
- technologia
 - proces-dla-połączenia, 464
 - wątek-dla-połączenia, 462
- teorie testowe, 106
- ternary raster operation, 128
- test poprawności formy, 80
- test-driven development, 111

- testowanie, 106, 111
 - JUnit, 109
 - mutacyjne, 118
 - testy, 118
 - współliniowość, 566
 - wyszukiwanie binarne, 111
 - z losowaniem, 115
 - zakresy, 106, 112
- testowe programy, 278
- testy, 96, 105
 - integracji, 106, 111
 - JUnit, 109
 - obciążeniowe, 386
 - piękne ze względu na prostotę, 106
 - piękne ze względu na swoją wszechstronność, 106
 - wydajność, 201
 - wyszukiwanie binarne, 106
- text delta, 35
- Text::Template, 190
- Thread_Args, 463
- threads, 447
- TLI, 449
- tokeny, 150
- TPC_Logging_Server, 462, 464
- transakcje, 410, 423
- transformacja drzewa, 30
- transformata Fouriera, 271
- transformator, 429
- transpozony, 236
- TreeView, 506, 510
- trójskładnikowa operacja rastrowa, 128
- trudności przypadkowe, 447
- trwałość deszyfracji, 192
- tts-format-text-and-speak, 528, 529
- tts-speak, 528
- turnstile_block(), 372, 383, 384, 385
- turnstile_interlock(), 384, 385
- turnstile_lookup(), 384
- turnstile_pi_waive(), 384
- tworzenie przypadków specjalnych, 315

U

- UBM, 359
- udostępnianie usług klientom zewnętrznym, 470
- układ równań liniowych, 273
- umap_bypass, 303
- umapfs, 300, 301, 303
- Unicode, 81
- Unix, 447
- upibp, 380
- URI, 92
- urządzenia, 285
- usb_interface, 288, 289
- usługa rejestrowania, 449

- usługa strumieniowa, 343
- ustawianie opcji, 214
- uwierzytelnianie z kluczem, 180
- uzupełnianie słów, 511
- użyteczność, 181

V

- VCALL(ap), 302
- Verifier, 90
- ViaVoice, 522
- Visual Basic 6, 503
- vnode, 306
- vnodeop_desc, 304
- voice-lock, 526
- vop_generic_args, 303
- VOP_READ, 298, 301, 307
- vop_read_args, 301
- vop_vector, 298, 301, 302
- VPN, 180

W

- WABA alignments, 209
- wait_for_multiple_events(), 458, 459
- walidacja kluczy, 181
- walidacja XML, 79
- warstwy, 308
- warstwy pośredniczące, 299
- warstwy systemów plików, 303
- wartość, 66
- warunek higieniczności dla rozszerzania makr, 427
- warunki
 - brzegowe, 387
 - końcowe, 24
- wątek-dla-połączenia, 462
- wątki, 408, 447
 - Haskell, 409
- wcięcia, 548
- wdrażanie, 185
 - rozproszone, 203
- Web of Trust, 180
- wejście binarne, 506
- wektor terminów, 394
- weryfikatory XML, 79, 86
- wewnętrzna blokada jądra, 381
- węzeł, 359
- while, 23
- wiązanie argumentów, 527
- wieloprzetwarzanie symetryczne, 249
- wielowątkowa wersja algorytmu, 265
- wielowątkowość w systemach wielordzeniowych, 265
- wielowymiarowe iteratory, 321
- wielowymiarowe tablice, 321

- window handler, 35
- Windows, 447
- Windows 1.0, 126, 131
- wizualizacja genomu, 206
- własności, 208
- Word Completion, 511
- wpływ architektury komputerów na algorytmy
 - macierzowe, 248
- wrap, 431
- wskazniki, 297
- wskazniki argumentów, 300
- współbieżność, 403
 - blokady, 404
 - metody zsynchronizowane, 404
 - pamięć transakcyjna, 406
 - zmienne warunkowe, 404
- współbieżny serwer rejestrowania, 461
- współliniowość, 559
- współpraca, 285
- współzależność kodu i danych, 125
- wsteczny cudzysłów, 198
- wychodzenie z pętli, 75
- wyczerpanie pamięci, 277
- wydajność, 25, 120, 277
 - interfejs użytkownika, 518
- wydania, 552
- wygląd kodu, 545
 - białe znaki, 550
 - bycie podręcznikowym, 547
 - DiffMerge, 547
 - długość wierszy tekstu, 547
 - komentarze, 549
 - narzędzia, 550
 - podobnie funkcjonujący kod, 547
 - poruszanie się po kodzie, 549
 - porządek, 549
 - wcięcia, 548
 - wydania, 552
 - wytyczne, 546
 - zagnieżdżony kod, 549
- wyjście mówione, 522
- wykonywanie, 26
- wymiana danych z użyciem protokołów e-biznesowych, 475
- wymiana podczas pracy, 355
- wyrażenia, 152
- wyrażenia regularne, 19
 - algorytmy dopasowujące, 20
 - grep, 20
 - implementacja, 21
 - klasy znaków, 26
 - kolejność sprawdzeń, 24
 - kompilacja, 26
 - koniec łańcucha, 23
 - match(), 22
 - Matcher, 26
 - matchhere(), 23, 24
 - matchstar(), 23, 24
 - metaznaki, 19
 - nawracanie, 25
 - Pattern, 26
 - początek łańcucha, 22
 - RE, 26
 - rekurencja, 25
 - rozszerzanie, 25
 - symbole wieloznaczne, 19, 26
 - warunki końcowe, 24
 - wydajność, 25
 - wyszukiwanie, 63
 - wzorzec, 19
 - zatrzymanie rekurencji, 23
- wyszukiwanie, 61, 517
 - binarne, 72, 74, 106
 - czas, 61
 - Google, 75, 76
 - informacji, 76
 - na dużą skalę, 75
 - optymalizacja, 69
 - PageRank, 77
 - pamięć asocjacyjna, 66
 - pełnotekstowe, 76
 - sieciowe, 75
 - tablica asocjacyjna, 67
 - wychodzenie z pętli, 75
 - wyrażenia regularne, 63
 - Yahoo!, 75
 - z orzeczeniami, 76
 - zorientowane na zadania, 535
- wyszukiwanie przyczyn, 483
- wytyczne pisania kodu, 546
- wyważenie obciążenia, 397
- wywołanie, 145
 - metody, 213
 - zwrotne, 227
- wznawianie wątków, 372
- wzorce, 19, 425, 450
 - Adapter, 448
 - fasada, 448, 454
 - metoda szablonu, 448, 453
- wzór Euklidesa, 567

X

- XML, 79, 102, 358, 469, 476
 - buforowanie, 91
 - gramatyka BNF, 81
 - optymalizacja O(1), 87
 - optymalizacja O(log N), 84
 - parser, 86
 - weryfikacja znaków cyfrowych, 84
 - weryfikator znaków, 82

XML 1.0, 81
XOM, 79, 85, 87
XPath, 476
XSLT, 537

Y

yacc, 306
Yahoo!, 75
Yahoo! Maps, 535

Z

zabezpieczanie kodu, 195
zabezpieczenia handlu internetowego, 202
zagnieżdżony kod, 549
zakres, 156
zaokrąglanie, 271
zapis zmian, 552
zapisywanie binarnej tablicy wyszukiwania, 89
zarządzanie

- cykl życiowy grupy procesów, 465
- czas, 339
- dane, 340
- personel, 340
- złożoność kodu wraz z upływem czasu, 542

zasada DRY, 497
zasady pisania kodu, 245, 546
zasięg lokalny, 245
zasób, 359
zestaw testów, 188

- testy binarności, 121

ZFS, 375
zliczanie

- jedynki w tablicy, 170
- referencje, 291, 292
- żądania artykułu, 67

zliczanie bitów w stanie wysokim, 163

- CSA, 171
- dziel i zwyciężaj, 165
- ILP, 174
- metody, 164, 167
- odstęp Hamminga, 175
- population count, 175
- porównywanie liczby ustawionych bitów
 - w dwóch słowach, 169
- RISC, 164
- zliczanie jedynek w tablicy, 170

zmiany, 552
zmiennne, 213

- CGI, 238
- warunkowe, 404, 405

znaczniki bitowe, 89
ZODB, 359, 360
Zope, 360
Zope Content Management Framework, 357
Zope Object Database, 360
Zope Page Templates, 358
zorientowany obiektowo projekt frameworku serwera

- rejestracja, 451

ZPT, 358
zrozumiały kod, 245
zrównoleganie algorytmów, 390
zrzut jądra, 378
zunifikowany model biznesowy, 359
zwięzłość, 279, 496

Ż

żądania

- HTTP, 471
- XML, 476

PROGRAM PARTNERSKI

GRUPY WYDAWNICZEJ HELION



1. ZAREJESTRUJ SIĘ
2. PREZENTUJ KSIĄŻKI
3. ZBIERAJ PROWIZJĘ

Zmień swoją stronę WWW
w działający bankomat!

Dowiedz się więcej i dołącz już dzisiaj!

<http://program-partnerski.helion.pl>

GRUPA WYDAWNICZA

 **Helion SA**

Dołącz do grona mistrzów programowania!

Wbrew pozorom programowanie to nie tylko nauka ścisła, to także sztuka! Trudna sztuka! Napisanie kodu poprawnie działającego czy kodu spełniającego oczekiwania użytkowników programu to niewątpliwie wyzwanie! Wymaga bowiem doskonałego zaplanowania architektury, skutecznej optymalizacji kodu źródłowego oraz umiejętności przewidywania potencjalnych problemów i ich odpowiednio wczesnej eliminacji.

Właśnie w tej książce prawdziwi mistrzowie programowania podzielą się z Tobą swoimi doświadczeniami, przemyśleniami i spostrzeżeniami dotyczącymi tworzenia profesjonalnych rozwiązań. Znajdziesz tu wiele praktycznych porad dotyczących pisania kodu, rozwiązywania problemów programistycznych, projektowania architektury, tworzenia interfejsów użytkownika i pracy w zespole projektowym. Dowiesz się, kiedy należy postępować dokładnie według wskazań metodologii, a kiedy pójście na skróty może okazać się najlepszym rozwiązaniem. Poznasz sposób myślenia i zasady pracy najlepszych programistów świata, dzięki czemu użytkownikom Twoich aplikacji zapewnisz maksymalny komfort.

- Korzystanie z wyrażań regularnych
- Dobór odpowiedniego poziomu abstrakcji
- Ocena jakości kodu źródłowego
- Testowanie
- Techniki analizy składni
- Zabezpieczanie komunikacji sieciowej
- Dostosowywanie architektury systemu do architektury komputerów
- Praca zespołowa
- Projektowanie systemów na bazie komponentów *open source*
- Usuwanie błędów
- Ułatwianie pracy osobom niepełnosprawnym

Lista współautorów:

Brian Kernighan	Travis E. Oliphant
Karl Fogel	Ronald Mak
Jon Bentley	Rogério Atem de Carvalho
Tim Bray	oraz Rafael Monnerat
Elliotte Rusty Harold	Bryan Cantrill
Michael Feathers	Jeff Dean
Alberto Savoia	oraz Sanjay Ghemawat
Charles Petzold	Simon Peyton Jones
Douglas Crockford	R. Kent Dybvig
Henry S. Warren Jr.	William R. Otte
Ashish Gulhati	oraz Douglas C. Schmidt
Lincoln Stein	Andrew Patzer
Jim Kent	Andreas Zeller
Jack Dongarra	Yukihiro Matsumoto
oraz Piotr Luszczek	Arun Mehta
Adam Kolawa	T.V. Raman
Greg Kroah-Hartman	Laura Wingerd
Diomidis Spinellis	oraz Christopher Seiwald
Andrew Kuchling	Brian Hayes

Całkowity dochód z oryginalnego wydania tej książki zostanie przekazany na rzecz organizacji Amnesty International.

Helion	
księgarnia internetowa	Helion SA ul. Kościuszki 1c, 44-100 Gliwice tel.: 32 230 98 63 e-mail: helion@helion.pl http://helion.pl
http://helion.pl	
zamówienia telefoniczne	
0 801 339900	Sprawdź najnowsze promocje: ● http://helion.pl/promocje Książki najchętniej czytane: ● http://helion.pl/bestsellery Zamów informacje o nowościach: ● http://helion.pl/nowosci
0 601 339900	
Informatyka w najlepszym wydaniu	

O'REILLY	
ISBN 978-83-283-3477-9	
9 788328	334779
cena: 89,00 zł	

sięgnij po **WIĘCEJ**

KOD KORZYŚCI