



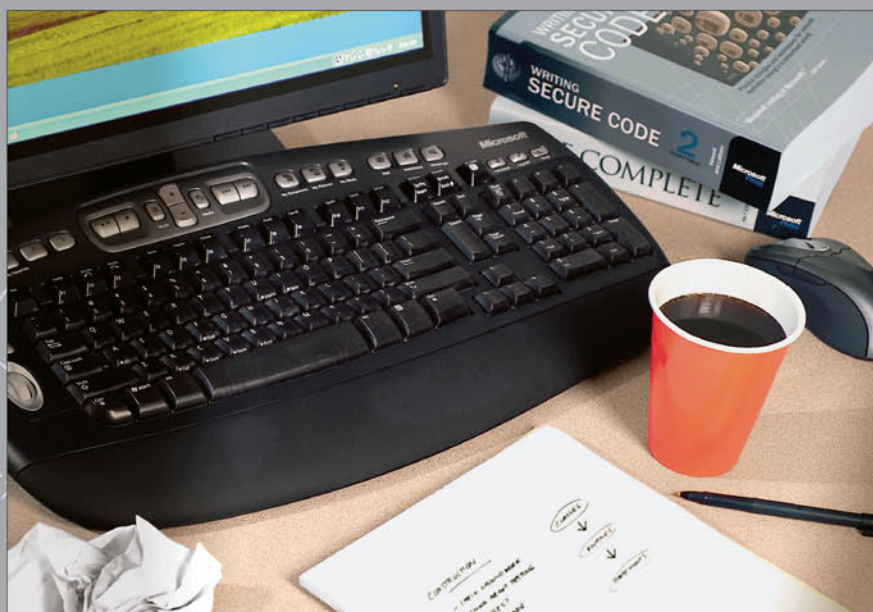
Microsoft®

2

Wydanie drugie

KOD DOSKONAŁY

Jak tworzyć oprogramowanie
pozbawione błędów



Kultowy podręcznik tworzenia doskonałego oprogramowania!

- Twórz wolny od błędów, najwyższej jakości kod
- Utrzymuj stałą kontrolę nad złożonymi projektami
 - Wcześnie wykrywaj i rozwiązuj problemy
- Sprawnie rozwijaj i poprawiaj oprogramowanie

Steve McConnell

Helion

Tytuł oryginału: Code Complete

Tłumaczenie: Paweł Koronkiewicz

ISBN: 978-83-283-3489-2

© 2010, 2017 Helion SA

Authorized translation of the English edition of Code Complete, Second Edition © 2004, Steven C. McConnell.

This translation is published and sold by permission of O'Reilly Media, Inc., which owns or controls 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.

Wydawnictwo HELION

ul. Kościuszki 1c, 44-100 GLIWICE

tel. 32 231 22 19, 32 230 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/koddov>

Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

Printed in Poland.

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

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

Spis treści

Wstęp	15
Podziękowania	23
Listy kontrolne	25
Tabele	27
Rysunki	29
Część I Proces budowy oprogramowania	35
1. Budowa oprogramowania	37
1.1. Czym jest budowa oprogramowania	37
1.2. Znaczenie procesu budowy oprogramowania	40
1.3. Jak korzystać z tej książki	41
2. Metafory procesu programowania	43
2.1. Znaczenie metafor	43
2.2. Jak korzystać z metafor w programowaniu	46
2.3. Popularne metafory programowania	47
3. Przed programowaniem — przygotowania	57
3.1. Przygotowania i ich znaczenie	58
3.2. Określanie rodzaju budowanego oprogramowania	65
3.3. Definicja problemu	70
3.4. Określenie wymagań	72
3.5. Architektura	77
3.6. Ilość czasu poświęcanego na przygotowania	89
4. Kluczowe decyzje konstrukcyjne	95
4.1. Wybór języka programowania	95
4.2. Konwencje programowania	100
4.3. Twoje położenie na fali technologii	101
4.4. Wybór podstawowych praktyk programowania	103
Część II Pisanie dobrego kodu	107
5. Projektowanie	109
5.1. Podstawowe problemy projektowania	110
5.2. Podstawowe pojęcia projektowania	113
5.3. Heurystyki — narzędzia projektanta	122
5.4. Techniki projektowania	146
5.5. Uwagi o popularnych metodykach pracy	155

6.	Klasy z klasą	161
	6.1. Abstrakcyjne typy danych	162
	6.2. Dobry interfejs klasy	169
	6.3. Problemy projektowania i implementacji	179
	6.4. Przesłanki dla utworzenia klasy	188
	6.5. Specyfika języka	192
	6.6. Pakiety klas	192
7.	Procedury wysokiej jakości	197
	7.1. Przesłanki utworzenia procedury	200
	7.2. Projektowanie na poziomie procedur	204
	7.3. Dobra nazwa procedury	207
	7.4. Jak długa może być procedura?	209
	7.5. Jak używać parametrów procedur	211
	7.6. Używanie funkcji	217
	7.7. Makra i procedury inline	218
8.	Programowanie defensywne	223
	8.1. Zabezpieczanie programu przed niewłaściwymi danymi wejściowymi	224
	8.2. Asercje	225
	8.3. Mechanizmy obsługi błędów	230
	8.4. Wyjątki	234
	8.5. Ograniczanie zasięgu szkód powodowanych przez błędy	239
	8.6. Kod wspomagający debugowanie	241
	8.7. Ilość kodu defensywnego w wersji finalnej	245
	8.8. Defensywne podejście do programowania defensywnego	246
9.	Proces Programowania w Pseudokodzie	251
	9.1. Budowanie klas i procedur krok po kroku	251
	9.2. Pseudokod dla zaawansowanych	253
	9.3. Budowanie procedur metodą PPP	256
	9.4. Alternatywy dla pseudokodu	269
Część III	Zmienne	273
10.	Zmienne w programie	275
	10.1. Podstawowa wiedza o danych	276
	10.2. Deklarowanie zmiennych	277
	10.3. Inicjalizowanie zmiennych	278
	10.4. Zakres	282
	10.5. Trwałość	289
	10.6. Czas wiązania	290

10.7. Związek między typami danych i strukturami sterowania	292
10.8. Jedno przeznaczenie każdej zmiennej	293
11. Potęga nazwy zmiennej	297
11.1. Wybieranie dobrej nazwy	297
11.2. Nazwy a rodzaje danych	303
11.3. Potęga konwencji nazw	308
11.4. Nieformalne konwencje nazw	310
11.5. Standardowe prefiksy	317
11.6. Nazwy krótkie a czytelne	319
11.7. Nazwy, których należy unikać	322
12. Podstawowe typy danych	327
12.1. Liczby	327
12.2. Liczby całkowite	329
12.3. Liczby zmiennoprzecinkowe	331
12.4. Znaki i ciągi znakowe	333
12.5. Zmienne logiczne	336
12.6. Typy wyliczeniowe	338
12.7. Stałe nazwane	343
12.8. Tablice	345
12.9. Tworzenie własnych typów (aliasy)	346
13. Inne typy danych	355
13.1. Struktury	355
13.2. Wskaźniki	359
13.3. Dane globalne	371
Część IV Instrukcje	383
14. Struktura kodu liniowego	385
14.1. Instrukcje, które wymagają określonej kolejności	385
14.2. Instrukcje, których kolejność nie ma znaczenia	388
15. Instrukcje warunkowe	393
15.1. Instrukcje if	393
15.2. Instrukcje case	398
16. Pętle	405
16.1. Wybieranie rodzaju pętli	405
16.2. Sterowanie pętlą	410
16.3. Łatwe tworzenie pętli — od wewnątrz	422
16.4. Pętle i tablice	424

17.	Nietypowe struktury sterowania	427
	17.1. Wiele wyjść z procedury	427
	17.2. Rekurencja	429
	17.3. Instrukcja goto	434
	17.4. Nietypowe struktury sterowania z perspektywy	444
18.	Metody oparte na tabelach	449
	18.1. Metody oparte na tabelach — wprowadzenie	449
	18.2. Tabele o dostępie bezpośrednim	451
	18.3. Tabele o dostępie indeksowym	462
	18.4. Tabele o dostępie schodkowym	464
	18.5. Inne metody wyszukiwania w tabelach	467
19.	Ogólne problemy sterowania	469
	19.1. Wyrażenia logiczne	469
	19.2. Instrukcje złożone (bloki)	480
	19.3. Instrukcje puste	481
	19.4. Praca z głębokimi zagnieżdżeniami	482
	19.5. Programowanie strukturalne	490
	19.6. Struktury sterujące i złożoność	493
Część V Sprawna praca z kodem		497
20.	Jakość oprogramowania	499
	20.1. Składowe jakości	499
	20.2. Metody podwyższania jakości	502
	20.3. Skuteczność metod podwyższania jakości	505
	20.4. Kiedy przeprowadzać kontrolę jakości	509
	20.5. Ogólna Zasada Jakości Oprogramowania	509
21.	Programowanie zespołowe	513
	21.1. Przegląd metod programowania zespołowego	514
	21.2. Programowanie w parach	517
	21.3. Formalne inspekcje	519
	21.4. Inne metody programowania zespołowego	526
22.	Testowanie	533
	22.1. Rola testów programisty	534
	22.2. Zalecane podejście do testów programisty	537
	22.3. Praktyczne techniki testowania	539
	22.4. Typowe błędy	550
	22.5. Narzędzia wspomagające testowanie	556

22.6. Usprawnianie testów	561
22.7. Gromadzenie informacji o testach	563
23. Debugowanie	569
23.1. Wprowadzenie	569
23.2. Wyszukiwanie defektu	574
23.3. Usuwanie defektu	585
23.4. Debugowanie a psychologia	588
23.5. Narzędzia debugowania — oczywiste i mniej oczywiste	591
24. Refaktoryzacja	597
24.1. Ewolucja oprogramowania i jej odmiany	598
24.2. Refaktoryzacje — wprowadzenie	599
24.3. Wybrane refaktoryzacje	605
24.4. Bezpieczne przekształcanie kodu	613
24.5. Strategie refaktoryzacji	615
25. Strategie optymalizacji kodu	621
25.1. Wydajność kodu	622
25.2. Optymalizowanie kodu	625
25.3. Rodzaje otyłości i lenistwa	632
25.4. Pomiar	637
25.5. Iterowanie	639
25.6. Strategie optymalizacji kodu — podsumowanie	640
26. Metody optymalizacji kodu	645
26.1. Struktury logiczne	646
26.2. Pętle	651
26.3. Przekształcenia danych	660
26.4. Wyrażenia	665
26.5. Procedury	674
26.6. Reimplementacja w języku niskiego poziomu	675
26.7. Im bardziej świat się zmienia, tym więcej zostaje bez zmian	677
Część VI Środowisko programowania	681
27. Jak rozmiar programu wpływa na jego budowę	683
27.1. Wielkość projektu a komunikacja	684
27.2. Skala rozmiarów projektów	684
27.3. Wpływ wielkości projektu na liczbę błędów	685
27.4. Wpływ wielkości projektu na efektywność pracy	687
27.5. Wpływ wielkości projektu na wykonywaną pracę	687

28.	Zarządzanie w programowaniu	695
	28.1. Zachęcanie do budowy dobrego kodu	696
	28.2. Zarządzanie konfiguracją	698
	28.3. Budowanie harmonogramu	705
	28.4. Pomiar	712
	28.5. Ludzkie traktowanie programistów	715
	28.6. Współpraca z przełożonymi	721
29.	Integracja	725
	29.1. Znaczenie metod integracji	725
	29.2. Częstość integracji — końcowa czy przyrostowa?	727
	29.3. Przyrostowe strategie integracji	730
	29.4. Codzienna kompilacja i test dymowy	738
30.	Narzędzia programowania	747
	30.1. Narzędzia do projektowania	748
	30.2. Narzędzia do pracy z kodem źródłowym	748
	30.3. Narzędzia do pracy z kodem wykonywalnym	754
	30.4. Środowiska narzędzi programowania	758
	30.5. Budowanie własnych narzędzi	759
	30.6. Narzędzia przyszłości	761
Część VII Rzemiosło programisty		765
31.	Układ i styl	767
	31.1. Wprowadzenie	768
	31.2. Techniki formatowania	774
	31.3. Style formatowania	776
	31.4. Formatowanie struktur sterujących	782
	31.5. Formatowanie instrukcji	789
	31.6. Formatowanie komentarzy	800
	31.7. Formatowanie procedur	802
	31.8. Formatowanie klas	804
32.	Kod, który opisuje się sam	813
	32.1. Zewnętrzna dokumentacja programu	813
	32.2. Styl programowania jako dokumentacja	814
	32.3. Komentować czy nie komentować	817
	32.4. Zasady pisania dobrych komentarzy	821
	32.5. Metody pisania komentarzy	828
	32.6. Normy IEEE	849

33.	Cechy charakteru	855
	33.1. Czy osobowość jest bez znaczenia?	856
	33.2. Inteligencja i skromność	857
	33.3. Ciekawość	858
	33.4. Uczciwość intelektualna	862
	33.5. Komunikacja i współpraca	865
	33.6. Kreatywność i dyscyplina	865
	33.7. Lenistwo	866
	33.8. Cechy, które znaczą mniej, niż myślisz	867
	33.9. Nawyki	869
34.	Powracające wątki — przegląd	873
	34.1. Walka ze złożonością	873
	34.2. Wybierz swój proces	875
	34.3. Pisz programy dla ludzi, nie tylko dla komputerów	877
	34.4. Programuj do języka, a nie w nim	879
	34.5. Konwencje jako pomoc w koncentracji uwagi	880
	34.6. Programowanie w kategoriach dziedziny problemu	881
	34.7. Uwaga, spadające odłamki!	884
	34.8. Iteruj, iteruj i jeszcze raz iteruj	886
	34.9. Nie będziesz łączył religii z programowaniem	887
35.	Gdzie znaleźć więcej informacji	891
	35.1. Programowanie	892
	35.2. Szersze spojrzenie na budowę oprogramowania	893
	35.3. Perodyki	895
	35.4. Plan czytelniczy programisty	896
	35.5. Stowarzyszenia zawodowe	898
	Bibliografia	899
	Skorowidz	918

Programowanie defensywne

cc2e.com/0861

W tym rozdziale

- 8.1. Zabezpieczanie programu przed niewłaściwymi danymi wejściowymi — strona 224
- 8.2. Asercje — strona 225
- 8.3. Mechanizmy obsługi błędów — strona 230
- 8.4. Wyjątki — strona 234
- 8.5. Ograniczanie zasięgu szkód powodowanych przez błędy — strona 239
- 8.6. Kod wspomagający debugowanie — strona 241
- 8.7. Ilość kodu defensywnego w wersji finalnej — strona 245
- 8.8. Defensywne podejście do programowania defensywnego — strona 246

Podobne tematy

- Ukrywanie informacji: „Ukrywaj tajemnice (ukrywanie informacji)” w podrozdziale 5.3
- Przygotowywanie projektu na przyszłe zmiany: „Identyfikuj obszary potencjalnych zmian” w podrozdziale 5.3
- Architektura: podrozdział 3.5
- Projektowanie: rozdział 5.
- Debugowanie: rozdział 23.



Programowanie defensywne nie oznacza przyjmowania postawy obronnej w trakcie omawiania kodu („To przecież doskonale działa!”). Jest to nawiązanie do defensywnej jazdy samochodem, która sprowadza się do przyjęcia założenia, że nigdy nie można być pewnym zachowania innych kierowców. Dzięki temu założeniu, gdy na drodze wydarzy się coś niebezpiecznego, defensywny kierowca ma duże szanse wyjść z przygody bez szwanku. Bez względu na to, komu zostanie przypisana wina za spowodowanie zagrożenia, na każdym spotyka odpowiedzialność za chronienie swojego zdrowia i życia. W programowaniu defensywnym podstawowym celem jest to, by przekazanie do procedury złych danych nie powodowało żadnych szkód, nawet jeżeli winę za doprowadzenie do takiej sytuacji będzie ponosiła inna część programu. Bardziej ogólnie, punktem wyjścia do programowania defensywnego jest przyznanie, że program będzie zmieniany i będą pojawiać się problemy. Dobry programista bierze to w trakcie pisania kodu pod uwagę.

W tym rozdziale piszę o tym, jak chronić się przed zimnym, okrutnym światem błędnych danych, zdarzeń, które „nigdy” nie nastąpią, i błędów innych programistów. Jeżeli masz duże doświadczenie, możesz pominąć pierwszy podrozdział — poświęcony zabezpieczeniu programu przed wadliwymi danymi wejściowymi — i przejść od razu do podrozdziału 8.2, w którym omawiane jest zagadnienie właściwego stosowania asercji.

8.1. Zabezpieczanie programu przed niewłaściwymi danymi wejściowymi

Być może spotkałeś się w szkole ze sformułowaniem „garbage in, garbage out” (śmieci na wejściu, śmieci na wyjściu). Jest to zasadniczo przeniesiona na grunt programowania zasada *caveat emptor*¹: użytkownicy, strzeżcie się!



W przypadku oprogramowania, które ma być wdrażane i aktywnie wykorzystywane, zasada „garbage in, garbage out” nie jest wystarczająca. Dobry program nigdy nie wyprowadza na wyjście śmieci, bez względu na przekazane mu dane. Może on działać zgodnie z zasadą „śmieci na wejściu, nic na wyjściu”, „śmieci na wejściu, komunikat błędu na wyjściu” lub „śmieci nie są dozwolone”. Według współczesnych standardów działanie na zasadzie „śmieci na wejściu, śmieci na wyjściu” znamionuje niedopracowaną i niebezpieczną w użyciu aplikację.

Można wyróżnić trzy techniki radzenia sobie ze śmieciami na wejściu programu:

Sprawdzanie wartości wszystkich danych ze źródeł zewnętrznych. Przy pobieraniu danych z pliku, od użytkownika, z sieci lub za pośrednictwem jakiegokolwiek innego interfejsu zewnętrznego należy sprawdzić, czy mieszczą się one w dopuszczalnym zakresie. W przypadku wartości liczbowej ważne jest, aby dane były liczbą i aby mieściła się ona w określonym przedziale. W przypadku ciągów znakowych problemem może być ich długość. Jeżeli ciąg ma reprezentować pewien szczególny zakres wartości (na przykład identyfikator transakcji lub klienta), należy dołożyć wszelkich starań, aby zweryfikować użyteczność odczytanych danych. Mając do czynienia z aplikacjami wymagającymi zabezpieczeń, warto zwrócić szczególną uwagę na niepożądane dane, które mogą posłużyć do zaatakowania systemu: celowe przepełnienia bufora, „wstrzyknięcia” SQL, HTML lub XML, błędy przepełnienia liczb całkowitych, dane przekazywane do wywołań systemowych itp.

Sprawdzanie wartości wszystkich parametrów wejściowych procedury. Zasada sprawdzania wartości parametrów wejściowych procedury jest powtórzeniem zasady weryfikowania danych pobieranych ze źródeł zewnętrznych, z tą różnicą, że miejsce interfejsu zewnętrznego zajmuje interfejs procedury. W podrozdziale 8.5, „Ograniczanie zasięgu szkód powodowanych przez błędy”, przedstawie praktyczną metodę określania, które z procedur wymagają sprawdzania wartości wejściowych.

¹ Klauzula handlowa nakazująca kupującemu sprawdzić towar przy zakupie i zwalniająca sprzedawcę z odpowiedzialności — *przyp. tłum.*

Określanie zasad obsługi złych danych. Co robisz po wykryciu błędnego parametru? W zależności od projektu możesz zdecydować się na jeden z kilkunastu schematów działania, które opisuję szczegółowo w podrozdziale 8.3 „Mechanizmy obsługi błędów”.

Programowanie defensywne jest świetnym uzupełnieniem innych opisywanych w tej książce technik podnoszenia jakości kodu, a jego najlepszą formą jest unikanie błędów od pierwszej wersji programu. Iteracyjne projektowanie, pisanie pseudokodu przed rozpoczęciem pracy z właściwym kodem, pisanie testów przed rozpoczęciem pisania kodu i niskopoziomowe inspekcje konstrukcyjne to działania, które pomagają unikać wprowadzania błędów — warto poświęcić im więcej uwagi niż samej idei programowania defensywnego. Jest ono jednak koncepcją, którą można bez przeszkód łączyć z każdą inną metodą pracy.

Jak ilustruje to rysunek 8.1, zabezpieczanie się przed pozornie drobnymi problemami może być w rzeczywistości istotniejsze, niż się początkowo wydaje. W dalszej części tego rozdziału opiszę różne techniki sprawdzania danych ze źródeł wewnętrznych, weryfikowania parametrów wejściowych i obsługi niepoprawnych wartości.



Rysunek 8.1. Część pływającego mostu na drodze I-90 w Seattle zatonała w trakcie burzy, ponieważ nie zamknięto pływaków, które utrzymywały go na powierzchni wody. Deszcz zalał je i most stał się zbyt ciężki. W trakcie budowy oprogramowania zabezpieczanie się przed drobiazgami ma większe znaczenie niż się wydaje.

8.2. Asercje

Asercja to kod stosowany w trakcie pracy nad oprogramowaniem — zazwyczaj procedura lub makro — który działa jako mechanizm automatycznej kontroli działania programu w trakcie jego wykonywania. Jeżeli asercja jest spełniona (ma wartość „prawda”), to znaczy, że kod działa zgodnie z oczekiwaniami. Jeżeli

nie jest spełniona, oznacza to wystąpienie nieoczekiwanego błędu. Na przykład gdy działanie systemu opiera się na założeniu, że plik z informacjami o klientach nigdy nie będzie przechowywał więcej niż 50 tysięcy rekordów, program może zawierać asercję mówiącą, że ich liczba jest mniejsza lub równa 50 tysięcy. Dopóki warunek ten będzie spełniony, asercja nie będzie wpływać na działanie kodu. Gdy jednak okaże się, że plik zawiera więcej niż 50 tysięcy rekordów, przypomni ona o swoim istnieniu zgłoszeniem wystąpienia błędu.



Asercje są szczególnie praktyczne w przypadku programów dużych i skomplikowanych oraz takich, które wymagają wysokiego poziomu niezawodności. Umożliwiają wtedy szybkie wykrywanie nietrafionych założeń interfejsu, nowych błędów pojawiających się przy wprowadzaniu modyfikacji itp.

Asercja wymaga zazwyczaj dwóch argumentów: wyrażenia logicznego opisującego założenie, które powinno być prawdziwe, oraz komunikatu, który będzie wyświetlany, w przypadku gdy wyrażenie logiczne będzie miało wartość „fałsz”. Oto przykład asercji w języku Java, która sprawdza, czy wartość zmiennej denominator (mianownik) jest różna od zera:

Przykład asercji (Java)

```
assert denominator != 0 : "Mianownik ma nieoczekiwaną wartość 0.";
```

Asercja ta bada założenie, że wartość denominator jest różna od 0. Pierwszy argument, `denominator != 0`, to wyrażenie logiczne, czyli o wartości „prawda” lub „fałsz”. Drugi to komunikat, który zostanie wypisany, gdy pierwszy argument będzie miał wartość „fałsz”.

Warto używać asercji do opisywania założeń przyjętych przy pisaniu kodu i wykrywania nieoczekiwanych sytuacji. Oto warunki, które mogą one sprawdzać:

- wartość parametru wejściowego (lub wyjściowego) mieści się w oczekiwanym zakresie;
- plik lub strumień jest otwarty (lub zamknięty), gdy procedura rozpoczyna pracę (lub gdy kończy pracę);
- wskaźnik pliku lub strumienia jest na jego początku (lub końcu), gdy procedura rozpoczyna pracę (lub gdy kończy pracę);
- plik lub strumień jest otwarty w trybie tylko-do-odczytu, tylko-do-zapisu lub trybie odczytywania i zapisywania;
- wartość zmiennej wejściowej nie została zmieniona w procedurze;
- wskaźnik nie jest wskaźnikiem pustym;
- tablica lub inny obiekt kontenerowy przekazany do procedury ma pojemność co najmniej X elementów danych;
- tablica została zainicjalizowana prawdziwymi danymi;
- obiekt kontenerowy jest pusty (lub pełny), gdy procedura rozpoczyna pracę (lub kończy pracę);
- wyniki wysoce zoptymalizowanej, złożonej procedury są zgodne z wynikami procedury wolniejszej, ale bardziej przejrzystej i lepiej sprawdzonej.

Oczywiście to tylko najprostsze przykłady — procedury mogą opierać swoje działanie na dużo bardziej szczegółowych założeniach. Można je opisywać między innymi za pomocą asercji.

W typowej sytuacji wyświetlanie komunikatów asercji przez kod przekazywany użytkownikom nie jest pożądane. Są one narzędziem przeznaczonym do stosowania tylko podczas pisania i modyfikowania kodu, normalnie są więc kompilowane podczas pracy z programem i pomijane w kompilacji wersji finalnej. W trakcie pracy z programem asercje zwracają uwagę na sprzeczne założenia, nieoczekiwane sytuacje, złe wartości przekazywane procedurom i inne podobne problemy. Pominięcie ich w kompilacji końcowej pozwala uniknąć ich niekorzystnego wpływu na wydajność.

Budowanie własnego mechanizmu asercji

Patrz też: Budowanie własnej procedury asercyjnej to dobry przykład programowania do języka (zamiast tylko w języku). Więcej na ten temat w podrozdziale 34.4 „Programuj do języka, a nie w nim”.

Wiele języków standardowo zapewnia możliwość korzystania z asercji — należą do nich C++, Java i Microsoft Visual Basic. Jeżeli stosowany język nie został wyposażony w procedury asercyjne, łatwo napisać je samodzielnie. Przykładowo, standardowe makro C++ `assert` nie daje możliwości korzystania z komunikatów tekstowych. Oto ulepszona wersja procedury `ASSERT`, również zaimplementowana jako makro C++:

Przykładowe makro asercji (C++)

```
#define ASSERT( condition, message ) { \
    if ( !(condition) ) {                \
        LogError( "Błąd asercji: ",    \
                #condition, message ); \
        exit( EXIT_FAILURE );          \
    }                                    \
}
```

Stosowanie asercji

Oto porady dotyczące korzystania z asercji:

Dla zdarzeń, których wystąpienia oczekujesz, stosuj kod obsługujący błędy; używaj asercji tylko dla tych sytuacji, które nigdy nie powinny mieć miejsca. Asercje sprawdzają, czy wystąpiła sytuacja, która *nigdy* nie powinna się zdarzyć. Kod obsługi błędów wykrywa natomiast wszystkie nietypowe okoliczności i zapewnia odpowiednie dla nich przetwarzanie. Nie muszą one zdarzać się często, ale zostały przewidziane przez programistę i kod przekazywany użytkownikowi musi zapewniać ich obsługę. Kod ten odpowiada za kontrolę danych wejściowych, podczas gdy zadaniem asercji jest wykrywanie błędów w programie.

Kod obsługujący błędy radzi sobie z nietypową sytuacją, pozwalając programowi zareagować w możliwie niekłopotliwy sposób. Jeżeli nietypowe okoliczności powodują uaktywnienie asercji, niekłopotliwa reakcja nie jest odpowiedzią — w tym przypadku wymagana jest zmiana kodu źródłowego, rekompilacja i udostępnienie nowej wersji oprogramowania.

Dobrym podejściem do asercji jest traktowanie ich jako „wykonywalnej dokumentacji” — nie sprawią one, że kod będzie działał, ale mogą być aktywną formą opisu zastępującą lub uzupełniającą komentarze.

Unikaj kodu wykonywalnego w asercjach. Umieszczenie w asercji kodu może skutkować tym, że zostanie on wyeliminowany z programu przy wyłączeniu jej mechanizmu. Przypuśćmy, że w programie znajduje się asercja:

Patrz też:

Przedstawione tu zagadnienie można także rozpatrywać jako przykład jednego z licznych problemów związanych z umieszczaniem wielu instrukcji w jednym wierszu. Więcej takich przykładów można znaleźć w punkcie „Nie więcej niż jedna instrukcja w wierszu” w podrozdziale 31.5.

Niebezpieczna forma asercji (Visual Basic)

```
Debug.Assert( PerformAction() ) 'Nie można wykonać operacji
```

Problem polega tu na tym, że gdy asercje nie zostaną skompilowane, nie zostanie skompilowany również kod wykonujący operację, czyli wywołanie `PerformAction()`. Tego typu instrukcje należy zawsze umieszczać w odrębnych wierszach — ich wynik można zapisać w zmiennej stanu i to jej wartość powinna podlegać badaniu. Oto przykład bezpiecznego użycia asercji:

Bezpieczne użycie asercji (Visual Basic)

```
actionPerformed = PerformAction()
Debug.Assert( actionPerformed ) 'Nie można wykonać operacji
```

Patrz też: O warunkach wstępnych i końcowych można przeczytać w książce *Programowanie zorientowane obiektowo* (Meyer 2005).

Używaj asercji do opisywania i weryfikowania warunków wstępnych i końcowych. Warunki wstępne i końcowe są elementem podejścia do projektowania i programowania znanego jako „projektowanie kontraktowe” (Meyer 2005). Gdy zostają one określone, procedura lub klasa zawiera rodzaj umowy z innymi częściami programu.

Warunki wstępne to charakterystyki, których przygotowanie kod wywołujący musi zapewnić, zanim wywoła procedurę lub utworzy obiekt. Są one zobowiązaniami kodu klienckiego wobec kodu wywoływanego.

Warunki końcowe to charakterystyki, które procedura lub klasa „obiecuje” osiągnąć w chwili zakończenia swojej pracy. Są to zobowiązania procedury lub klasy wobec kodu wywołującego.

Asercje to dobre narzędzie do dokumentowania warunków wstępnych i końcowych. Warunki te mogą być też opisywane w komentarzach, jednak asercje mają tę przewagę, że dynamicznie sprawdzają, czy są one spełnione.

W poniższym przykładzie asercje zostały użyte do opisanego warunków wstępnych i końcowych procedury `Velocity`.

Przykład wykorzystania asercji do opisu warunków wstępnych i końcowych (Visual Basic)

```
Private Function Velocity ( _
    ByVal latitude As Single, _
    ByVal longitude As Single, _
    ByVal elevation As Single _
) As Single
    ' szybkość
    ' szerokość geograficzna
    ' długość geograficzna
    ' wysokość

    ' warunki wstępne
    Debug.Assert ( -90 <= latitude And latitude <= 90 )
    Debug.Assert ( 0 <= longitude And longitude < 360 )
```



```

Debug.Assert ( -500 <= elevation And elevation <= 75000 )
...

' warunki końcowe
Debug.Assert ( 0 <= returnVelocity And returnVelocity <= 600 )

' zwracana wartość
Velocity = returnVelocity
End Function

```

Gdyby wartości `latitude`, `longitude` i `elevation` były pobierane z zewnątrz, wykrywanie w nich nieprawidłowości i odpowiednie reakcje programu powinny zapewniać kod obsługi błędów, a nie asercje. W przypadku gdy dane pochodzą z zaufanego, wewnętrznego źródła, a konstrukcja procedury bazuje na założeniu, że wartości będą mieściły się w dopuszczalnych zakresach, użycie asercji jest właściwe.

Patrz też: Więcej o niezawodności w punkcie „Poprawność a odporność” w podrozdziale 8.3.

Aby zapewnić wysoką niezawodność, używaj asercji, a potem zapewniaj obsługę błędów. W przypadku wystąpienia błędu procedura może użyć asercji lub kodu do jego obsługi, ale nie może skorzystać z obu tych mechanizmów jednocześnie. Niektórzy eksperci twierdzą nawet, że stosowanie tylko jednego z nich jest w zupełności wystarczające (Meyer 2005).

Spotykane w codziennym życiu programy i projekty nie są jednak zazwyczaj na tyle uporządkowane, aby można było ograniczyć się do samych tylko asercji. W dużych, rozwijanych latami systemach różne części mogą być projektowane przez różne osoby na przestrzeni 5 lub 10 lat, a czasem nawet dłuższego okresu. Projektantów dzieli wtedy czas i wiele ewoluujących wersji kodu. Często są oni ukierunkowani na stosowanie zupełnie innych technologii. Dodatkowo, jeżeli część systemu pochodzi ze źródeł zewnętrznych, pojawia się separacja geograficzna. W różnych fazach czasu życia systemu programiści stosują odmienne konwencje pisania kodu. Ponadto w dużych zespołach zawsze pojawiają się programiści bardziej i mniej sumienni, więc różne części kodu są bardziej lub mniej rygorystycznie przeglądane. Niektórzy dbają o dokładniejsze testy jednostkowe, natomiast gdy zespoły testujące pracują w różnych stronach świata i podlegają presji natury ekonomicznej, wynikiem jest niejednolita w poszczególnych wersjach jakość przeprowadzanych testów. Nie można też liczyć na pełne testowanie regresyjne na poziomie systemu.

W takich warunkach za wykrywanie tego samego błędu może odpowiadać zarówno asercja, jak i kod obsługi błędów. W kodzie źródłowym programu Microsoft Word warunki, które powinny być zawsze spełnione, są opisane asercjami, ale istnieje dla nich także kod obsługujący błędy, który może zostać uruchomiony, gdyby asercja zawiodła. W wyjątkowo dużych, złożonych i długo rozwijanych aplikacjach takich jak Word asercje, jako narzędzie pozwalające wykrywać jak największą liczbę błędów programistycznych, są bardzo pomocne. Przy takim stopniu złożoności aplikacji (miliony wierszy kodu) i po tylu generacjach zmian trudno jednak realistycznie oczekiwać, że każdy możliwy błąd zostanie wykryty i poprawiony przed oddaniem wersji przeznaczonej dla użytkowników. Stąd potrzeba zapewnienia obsługi nieoczekiwanych nieprawidłowości także w tej wersji programu.

Oto przykład takiej konstrukcji dla funkcji *Velocity*:

Przykład wykorzystania asercji do opisu warunków wstępnych i końcowych (Visual Basic)

```
Private Function Velocity ( _                'szybkość
    ByVal latitude As Single, _            'szerokość geograficzna
    ByVal longitude As Single, _          'długość geograficzna
    ByVal elevation As Single _          'wysokość
) As Single
```

' warunki wstępne

```
Debug.Assert ( -90 <= latitude And latitude <= 90 )
Debug.Assert ( 0 <= longitude And longitude < 360 )
Debug.Assert ( -500 <= elevation And elevation <= 75000 )
...
```

Kod asercji.

' Oczyszczanie danych wejściowych. Wartości powinny mieścić się w zakresach opisanych
' przez asercje, ale gdy tak nie jest, zostają zmienione na najbliższą dopuszczalną wartość.

```
If ( latitude < -90 ) Then
    latitude = -90
ElseIf ( latitude > 90 ) Then
    latitude = 90
End If
If ( longitude < 0 ) Then
    longitude = 0
ElseIf ( longitude > 360 ) Then
    ...
```

Kod, który obsługuje
złe dane wejściowe
w czasie wykonywania.

8.3. Mechanizmy obsługi błędów

Asercje mają zapewniać obsługę błędów, które nigdy nie powinny wystąpić. Co należy robić z błędami, które są oczekiwane? W zależności od sytuacji można zwracać wartość neutralną, podstawić następny element poprawnych danych, powtarzać wcześniejszą odpowiedź, wstawiać najbliższą wartość w dopuszczalnym zakresie, rejestrować ostrzeżenie w pliku, zwracać kod błędu, wywoływać procedurę albo obiekt obsługi błędów, wyświetlać komunikat lub przerwać pracę programu. Można też stosować połączenia powyższych technik.

Oto opisy tych podstawowych schematów obsługi błędów:

Zwracanie wartości neutralnej. Czasem najlepszą reakcją na złe dane jest kontynuowanie pracy i zwrócenie wartości, która nie wywoła żadnych szkód. Obliczenie numeryczne może zwracać 0. Operacja na ciągach znakowych — ciąg pusty. Operacja wskaźnikowa może zwracać pusty wskaźnik. Procedura rysująca, która otrzymuje niepoprawną wartość opisującą kolor w grze wideo, może używać standardowego koloru tła lub rysunku. Taka sama procedura, która jednak wyświetla dane z prześwietlenia chorego na raka pacjenta, nie powinna zwracać „wartości neutralnej”. W takim przypadku rozwiązaniem lepszym niż wyświetlenie niepoprawnych danych jest przerwanie pracy programu.

Podstawianie następnego poprawnego elementu danych. Przy przetwarzaniu strumienia danych najlepszym wyjściem może być kontynuowanie zwracania tych, które są poprawne. Jeżeli przy odczytywaniu rekordów z bazy okaże się,

że jeden z nich jest uszkodzony, rozwiązaniem jest kontynuowanie odczytu aż do znalezienia rekordu poprawnego. Jeżeli sto razy na sekundę odczytujesz temperaturę i pojedyncza wartość okazuje się niepoprawna, dopuszczalne może być ograniczenie obsługi błędów do oczekiwania na kolejny odczyt.

Powtarzanie wcześniejszej odpowiedzi. Jeżeli program odczytujący temperaturę nie otrzymuje pojedynczego odczytu, może zwracać ostatnią znaną wartość — choć zależy to od konkretnego zastosowania, na ogół można liczyć na to, że temperatura nie ulegnie znacznej zmianie w ciągu jednej setnej sekundy. Również w grze wideo, gdy pojawia się żądanie pokrycia części ekranu błędnym kolorem, można pozostać przy kolorze wykorzystywanym wcześniej. Gdy jednak masz do czynienia z autoryzowaniem transakcji bankomatu, rozwiązanie polegające na powtórzeniu ostatniej odpowiedzi z oczywistych względów nie jest dopuszczalne.

Podstawianie najbliższej dopuszczalnej wartości. W niektórych sytuacjach można zdecydować się na zwracanie najbliższej wartości mieszczącej się w dopuszczalnym zakresie. Tak postąpiliśmy w ostatniej wersji funkcji `Velocity`. Rozwiązanie to sprawdza się zazwyczaj przy rejestrowaniu odczytów różnych instrumentów. Termometr może być skalibrowany do pracy w zakresie od 0 do 100 stopni. Przy odczycie wartości niższej można podstawiać 0, czyli najbliższą wartość w dopuszczalnym przedziale. Podobnie, przy odczycie wartości wyższej od 100 można podstawiać 100. W przypadku operacji na ciągach znakowych, gdy liczba mająca opisywać ich długość jest mniejsza od 0, można podstawiać 0. Mój samochód stosuje takie podejście przy cofaniu. Ponieważ na skali szybkościomierza nie ma wartości ujemnych, pokazuje on wtedy prędkość 0 — najbliższą w dopuszczalnym zakresie.

Rejestrowanie ostrzeżenia w pliku. Odpowiedzią na wykrycie błędnych danych może być zapisanie w pliku stosownego ostrzeżenia i kontynuowanie pracy. Podejście takie można łączyć z innymi, na przykład z podstawianiem najbliższej dopuszczalnej wartości lub następnego poprawnego elementu danych. Gdy korzystasz z pewnego rodzaju dziennika (logu), musisz rozważyć, czy może on zostać upubliczniony — niektóre aplikacje mogą zmuszać do użycia szyfrowania lub innego rodzaju ochrony.

Zwracanie kodu błędu. Możesz wybrać części systemu, w których będzie implementowana obsługa błędów, i ograniczyć reakcję na nieprawidłowości w innych częściach do zgłaszania ich wystąpienia. Wówczas procedury na pewnym poziomie oczekują, że to procedury stojące wyżej w hierarchii wywołań zapewnią odpowiednią obsługę błędów. Można wyróżnić kilka sposobów powiadamiania innych elementów systemu o ich wystąpieniu:

- przypisywanie pewnej wartości zmiennej stanu,
- zwracanie wartości opisującej stan jako wartości funkcji,
- zgłaszanie wyjątku przy użyciu standardowego mechanizmu wyjątków.

Wybór mechanizmu zgłaszania błędów nie jest tak istotny jak decyzja o tym, które części systemu będą obsługiwać je bezpośrednio, a które będą jedynie

zgłaszać ich wystąpienie. Jeżeli istotne jest bezpieczeństwo, należy zwrócić szczególną uwagę na każdorazowe sprawdzanie kodów stanu w procedurach wywołujących.

Wywoływanie procedury (obiektu) obsługi błędu. Nieco innym podejściem jest dążenie do centralizacji obsługi błędów w globalnych procedurach lub obiektach. Zaletą tej techniki jest scentralizowanie odpowiedzialności, które ułatwia debugowanie. Wadą jest natomiast to, że o takim wspólnym mechanizmie wie cały program i cały program jest z nim powiązany. Jeżeli kiedykolwiek pojawi się potrzeba użycia kodu w innym systemie, wymagane będzie przeniesienie razem z nim pełnego mechanizmu obsługi błędów.

Metoda ta ma istotny związek z bezpieczeństwem. W przypadku przepełnienia bufora atakujący może uzyskać dostęp do adresu procedury lub obiektu obsługującego błędy. Rozwiązanie to staje się więc zagrożeniem od chwili, gdy w pracującej aplikacji wystąpi takie przepełnienie.

Wyświetlanie komunikatu błędu bez względu na miejsce wystąpienia problemu. To podejście minimalizuje ilość pracy, którą trzeba włożyć w zaprogramowanie mechanizmu obsługi błędów. Może ono jednak zarazem prowadzić do rozproszenia komunikatów interfejsu użytkownika po całej aplikacji, co utrudnia zapewnienie mu spójności, separowanie UI od reszty systemu i lokalizowanie oprogramowania. Należy też uważać, aby komunikaty błędów nie zawierały informacji pomocnych osobom zainteresowanym przełamaniem zabezpieczeń systemu. Ich zawartość może być dużą pomocą dla doświadczonego włamywacza.

Lokalna obsługa błędów. W niektórych projektach najlepszym rozwiązaniem jest lokalne obsługiwane błędów. Decyzję o wyborze określonej metody podejmuje programista projektujący i implementujący tę część systemu, w której błąd może wystąpić.

Podejście takie zapewnia poszczególnym programistom zespołu bardzo dużą swobodę, ale tworzy istotne zagrożenie tym, że działanie całego systemu nie będzie spełniać wymagań dotyczących poprawności lub odporności (więcej na ten temat za chwilę). Zależnie od wybieranych przez programistów metod na mniejszą lub większą skalę może wystąpić rozproszenie kodu interfejsu użytkownika po całym systemie. Naraża to program na wszelkie problemy związane z niejednorodnym systemem wyświetlania komunikatów błędów.

Przerwanie pracy programu. Niektóre systemy w chwili napotkania błędu przerywają pracę. Rozwiązanie takie może być najlepsze, gdy ich działanie ma wpływ na bezpieczeństwo ludzi. Przykładowo, jaka powinna być reakcja systemu, który steruje aparaturą naświetlającą stosowaną w leczeniu chorych na raka, jeżeli otrzyma on złe dane dotyczące dawkowania? Czy może użyć poprzedniej poprawnej wartości? Czy może zastosować najbliższą wartość poprawną? Czy może skorzystać z wartości neutralnej? W tym przypadku przerwanie pracy jest najlepszym rozwiązaniem. Lepiej uruchomić urządzenie ponownie, niż ryzykować poddanie pacjenta niewłaściwie dobranej dawce promieniowania.

Podobne podejście można wykorzystać dla poprawienia bezpieczeństwa systemu Microsoft Windows. Normalnie, gdy dziennik zabezpieczeń jest pełny, Windows kontynuuje pracę. Można jednak wprowadzić taką konfigurację, w której zapełnienie dziennika spowoduje zatrzymanie serwera. Rozwiązanie to może być najlepszym, gdy bezpieczeństwo środowiska jest na pierwszym miejscu.

Poprawność a odporność

Jak pokazują przykłady gry wideo i urządzenia naświetlającego, wybór metody obsługi błędów w dużej mierze zależy od rodzaju oprogramowania. Przykłady te zwracają też uwagę na fakt, że pewne metody sprzyjają zachowaniu poprawności kodu, podczas gdy inne prowadzą do wyższej odporności programu. Programiści używają tych terminów dość nieformalnie, ale poprawność i odporność to w istocie dwa przeciwieństwa. **Poprawność** (ang. *correctness*) oznacza, że program nigdy nie zwraca wyniku, który nie jest dokładny, i jego brak jest uznawany za lepszy niż jakikolwiek jego substytut. **Odporność** (ang. *robustness*) to pojęcie oznaczające, że zawsze podejmowane są działania mające na celu podtrzymanie funkcjonowania systemu, nawet jeżeli prowadzi to czasem do niedokładnych wyników.

W aplikacjach, których praca ma związek z bezpieczeństwem ludzi, zazwyczaj preferowana jest poprawność. Lepiej nie zwracać wyniku, niż zwracać błędny. Urządzenie naświetlające jest dobrym przykładem takiego systemu.

W programach użytkowych często bardziej ceniona jest odporność — jakkolwiek wynik jest lepszy niż całkowite zamknięcie aplikacji. Używany przeze mnie edytor tekstu od czasu do czasu wyświetla obcięty fragment wiersza przy dolnej krawędzi okna. Czy wystąpienie takiej sytuacji powinno spowodować zamknięcie edytora? Nie. Wiem, że gdy tylko przewinę zawartość okna w górę lub w dół, ekran zostanie odświeżony i wszystko wróci do normy.

Obsługa błędów a projekt wysokiego poziomu



Gdy dostępnych jest tak wiele możliwości, trzeba zwracać szczególną uwagę na spójność reguł obsługi błędów w całym programie. Powyższe mechanizmy mają wpływ na to, w jakim stopniu oprogramowanie spełnia wymagania w zakresie poprawności, odporności i innych cech нефункциональных. Wybór ogólnego schematu postępowania w przypadku złych parametrów to decyzja podejmowana na poziomie architektury lub projektu wysokiego poziomu.

Po dokonaniu wyboru zasad obsługi błędów należy ich konsekwentnie przestrzegać. Jeżeli decydujesz, że ich obsługę zapewnia kod wysokiego poziomu, a kod niskiego poziomu tylko je zgłasza, musisz zadbać o to, aby pierwszy z nich faktycznie pracował z błędami. Niektóre języki pozwalają ignorować fakt, że funkcja zwraca kod błędu — w C++ zwracana wartość w ogóle nie musi zostać użyta — nie jest to jednak cecha, z której warto korzystać. Zawsze sprawdzaj wartość zwracaną przez funkcję i wszelkie inne kody błędów. Nawet gdy nie oczekujesz, by funkcja mogła kiedykolwiek zgłosić błąd, sprawdzaj. Ochrona przed nieoczekiwanymi błędami to właśnie istota koncepcji programowania defensywnego.

Porady te w równej mierze stosują się do funkcji systemowych, jak i do funkcji, które definiujesz samodzielnie. O ile nie wprowadziłeś w architekturze ogólnej zasady, że błędy wywołań systemowych nie będą wykrywane, sprawdzaj wartość kodu błędu po każdym wywołaniu. W przypadku wystąpienia nieprawidłowości programiście natychmiast powinna zostać udostępniona informacja o numerze błędu i jego standardowy opis.

8.4. Wyjątki

Wyjątki to specyficzny mechanizm, który umożliwia przekazywanie informacji o błędach lub nietypowych zdarzeniach do kodu wywołującego. Gdy w trakcie wykonywania procedury rozpoznana zostaje nieoczekiwana sytuacja, do której obsługi nie jest ona przygotowana, procedura „wyrzuca” (ang. *throws*) wyjątek — staje w miejscu i zaczyna krzyczeć: „Nie wiem, co z tym zrobić! Mam nadzieję, że ktoś się tym zajmie!”. Kod nieznaący kontekstu, w którym wystąpił błąd, może w ten sposób przekazać kontrolę innym częściom systemu, posiadającym wiedzę, która pozwoli zinterpretować sytuację i podjąć rozsądne działania.

Wyjątki można także wykorzystać do porządkowania zawilej logiki na pewnym odcinku kodu. Ilustruje to przykład „Przepisać kod z użyciem try-finally” w podrozdziale 17.3. Ogólnie rzecz biorąc, mechanizm wyjątku polega na tym, że procedura używa polecenia `throw` (wyrzucić), aby go **zgłosić**, i przekazuje jednocześnie obiekt wyjątku. Kod w innej procedurze, wyżej w hierarchii wywołań, używa bloku `try-catch` (próbuj-przechwyć), aby wyjątek **przechwyć**.

Implementacje mechanizmu wyjątków w najpopularniejszych językach nie są jednolite. W tabeli 8.1 zaprezentowane zostało zestawienie podstawowych różnic między trzema z nich.

Programy, które używają wyjątków jako jednego z mechanizmów zwykłego przetwarzania, sprawiają takie same problemy z czytelnością oraz przy modyfikacji i rozbudowie jak tradycyjny kod spaghetti.
— Andy Hunt
i Dave Thomas

Jest coś, co łączy wyjątki z dziedziczeniem: oba te mechanizmy, odpowiednio stosowane, pozwalają zmniejszyć złożoność, jednak lekkomyślne ich traktowanie szybko prowadzi do powstania kodu, który jest niemal zupełnie niezrozumiały. Na kolejnych stronach zebrane zostały wskazówki zwracające uwagę na korzyści płynące ze stosowania wyjątków oraz problemy, które mogą wystąpić przy nadużywaniu tego mechanizmu.

Używaj wyjątków do powiadamiania innych części programu o błędach, które nie powinny być ignorowane. Największą zaletą wyjątków jest dawana przez nie możliwość sygnalizowania błędów w sposób niepozwalający na ich zignorowanie (Meyers 1996). W przypadku innych mechanizmów obsługi błędów zawsze należy liczyć się z ryzykiem propagacji błędu w programie bez jego wykrycia. Wyjątki eliminują to zagrożenie.

Zgłaszaj wyjątek tylko w sytuacjach naprawdę wyjątkowych. Wyjątki powinny być zarezerwowane dla sytuacji faktycznie nietypowych, innymi słowy dla takich, do których nie można dostosować kodu innymi metodami. Stosuje się je podobnie jak asercje — nie dla zdarzeń rzadkich, ale tych, które *nigdy* nie powinny wystąpić.

Tabela 8.1. Wyjątki w trzech popularnych językach

Cecha	C++	Java	Visual Basic
Blok try-catch	tak	tak	tak
Blok try-catch-finally	nie	tak	tak
Dane wyjątku	obiekt <code>Exception</code> lub obiekt klasy pochodnej; wskaźnik do obiektu; odwołanie do obiektu; typ taki jak <code>string</code> lub <code>int</code>	obiekt <code>Exception</code> lub obiekt klasy pochodnej	obiekt <code>Exception</code> lub obiekt klasy pochodnej
Skutek nieprzechwycenia wyjątku	wywołanie procedury <code>std::unexpected()</code> , standardowo wywołującej procedurę <code>std::terminate()</code> , która standardowo wywołuje <code>abort()</code>	przerwanie wątku wykonania, jeżeli wyjątek jest „wyjątkiem kontrolowanym”; brak skutków, jeżeli jest to „wyjątek czasu wykonania”	przerwanie programu
Zgłaszane wyjątki muszą być definiowane w interfejsie klasy	nie	tak	nie
Przechwytywane wyjątki muszą być definiowane w interfejsie klasy	nie	tak	nie

Używając wyjątków, warto pamiętać o tym, że jest to trudny kompromis między wprowadzeniem mechanizmu dającego ogromne możliwości a zwiększeniem stopnia komplikacji kodu. Osłabiają one hermetyzację, bo wymagają od kodu wywołującego procedurę, aby znał wyjątki, które może zgłosić kod wywoływany. Zwiększa to złożoność, a więc zaprzecza temu, co w rozdziale 5., „Projektowanie”, określone zostało jako Główny Imperatyw Techniczny Oprogramowania: Zarządzanie Złożonością.

Nie używaj wyjątków jako metody odkładania rozwiązania problemu na później. Jeżeli obsługę błędów można zapewnić lokalnie, należy to zrobić. Użycie `throw` nie jest alternatywą, w przypadku gdy problem można rozwiązać na miejscu.

Unikaj zgłaszania wyjątków w konstruktorach i destruktorach, jeżeli nie zostaną przechwycone w tym samym miejscu. Gdy konstruktory i destruktory mogą zgłaszać wyjątki, reguły przetwarzania komplikują się bardzo szybko. Przykładem może być fakt, że w języku C++ destruktor nie zostanie wywołany przed ukończeniem budowania obiektu. W efekcie zgłoszenie wyjątku w konstruktorze powoduje pominięcie destruktoru. To może z kolei prowadzić do sytuacji, w której pewne zasoby nie zostają zwolnione (Meyers 1996, Stroustrup 2010). Podobne problemy pojawiają się przy zgłaszaniu wyjątków w destruktorach.

Adwokat tego czy innego języka mógłby stwierdzić, że zasady tego rodzaju są „trywialnie proste” i pamiętanie o nich nie jest żadnym problemem, jednak programista jest tylko człowiekiem i każda reguła zostaje wcześniej czy później

zapomniana (czy też przeoczona). Lepszą praktyką jest proste unikanie wprowadzania dodatkowej złożoności, będącej konsekwencją takiego kodu, i powstrzymanie się od zgłaszania wyjątków poza „zwykłymi” procedurami.

Patrz też: Więcej o utrzymywaniu spójności abstrakcji interfejsu w punkcie „Dobra abstrakcja” w podrozdziale 6.2.

Zgłaszaj wyjątki na właściwym poziomie abstrakcji. Procedura, tak jak i klasa, powinna reprezentować poprzez swój interfejs spójną abstrakcję. Podobnie jak wykorzystywane typy danych, zgłaszane wyjątki są częścią jej interfejsu.

Gdy podejmujesz decyzję o przekazaniu wyjątku procedurze wywołującej, upewnij się, że poziom jego abstrakcji odpowiada abstrakcji interfejsu procedury. Oto przykład tego, czego nie należy robić:



UWAGA,
ZŁY KOD!

Deklaracja wyjątku
na złym poziomie
abstrakcji.

```

Klasa, która zgłasza wyjątek o niewłaściwym poziomie abstrakcji (Java)
class Employee {
    // pracownik
    ...
    public TaxId GetTaxId() throws EOFException { // wyjątek EOF
        ...
    }
    ...
}

```

Kod `GetTaxId()` przekazuje niskopoziomowy wyjątek `EOFException` do kodu wywołującego. Procedura nie przejmuje za niego odpowiedzialności i ujawnia szczegóły swojej implementacji, przekazując niskopoziomowy obiekt tego wyjątku. Prowadzi to do ścisłego powiązania kodu klienta procedury nie z klasą `Employee`, ale z kodem na niższym poziomie, tym, który zgłasza wyjątek `EOFException`. Hermetyzacja zostaje złamana i funkcjonalność pojęciowa kodu maleje.

Procedura `GetTaxId()` powinna przekazywać wyjątek spójny z interfejsem klasy, której jest częścią, na przykład taki:

Deklaracja wyjątku
na dopasowanym
poziomie abstrakcji.

```

Klasa, która zgłasza wyjątek o właściwym poziomie abstrakcji (Java)
class Employee {
    // pracownik
    ...
    public TaxId GetTaxId() throws EmployeeDataNotAvailable {
        // dane pracownika niedostępne
        ...
    }
    ...
}

```

Kod obsługi wyjątku wewnątrz `GetTaxId()` może po prostu mapować wyjątek `EOFException` do `EmployeeDataNotAvailable`. To wszystko, co jest potrzebne do zachowania abstrakcji interfejsu.

Zawieraj w komunikacie wyjątku informacje o wszystkim, co doprowadziło do jego zgłoszenia. Wyjątek pojawia się w określonych okolicznościach, które zostają rozpoznane w chwili jego zgłaszania. Informacje o tych okolicznościach są bezcenne dla osoby, która czyta komunikat błędu. Dbaj o to, aby zawierał on wszystkie dane niezbędne do zrozumienia, dlaczego nastąpiło zgłoszenie wyjątku. Jeżeli został on zgłoszony w wyniku wykrycia błędnego indeksu tablicy, komunikat powinien zawierać wartość tego indeksu, jak również informacje o ograniczeniach, którym podlega tablica.

Unikaj pustych bloków catch. Pokusa zignorowania wyjątku, z którym nie bardzo wiadomo, co zrobić, może być czasem duża. Oto przykład:



Niepoprawny sposób ignorowania wyjątku (Java)

```
try {
    ...
    // duża ilość kodu
    ...
} catch ( AnException exception ) {
}
```

Taka konstrukcja sugeruje, że albo niepoprawny jest kod wewnątrz bloku try, bo zgłasza wyjątek bez powodu, albo błędny jest kod w bloku catch, bo nie obsługuje poprawnego wyjątku. Należy ustalić, co jest źródłem problemu, i skorygować blok try lub blok catch.

Może się zdarzyć, że wyjątek na niższym poziomie nie reprezentuje wyjątku na poziomie abstrakcji wywołującej procedury. Gdy faktycznie tak jest, należy przynajmniej opisać kod, wyjaśniając, dlaczego pusty blok catch jest właściwym rozwiązaniem. Rolę takiego opisu może pełnić odpowiedni komentarz lub polecenie zarejestrowania komunikatu wyjątku w dzienniku, na przykład:

Poprawny sposób ignorowania wyjątku (Java)

```
try {
    ...
    // duża ilość kodu
    ...
} catch ( AnException exception ) {
    LogError( "Nieoczekiwany wyjątek" );
}
```

Poznaj wyjątki swoich bibliotek. Jeżeli pracujesz w języku, który nie wymaga, aby procedury lub klasy definiowały zgłaszane wyjątki, powinieneś znać każdy z wyjątków zgłaszanych przez używane biblioteki. W przypadku gdy wyjątek wygenerowany przez kod biblioteki nie zostanie przechwycony, program nie będzie działał poprawnie. Jeżeli w kodzie tym nie zostały opisane wyjątki, utwórz kod prototypu, który sprawdzi działanie biblioteki i pozwoli je poznać.

Rozważ zbudowanie scentralizowanego mechanizmu informowania o wyjątkach. Jedną z metod ukierunkowanych na zapewnienie spójności obsługi wyjątków jest użycie scentralizowanego mechanizmu komunikatów. Służy on jako repozytorium wiedzy o tym, jakie rodzaje wyjątków mogą wystąpić, jak powinny być obsługiwane, w jaki sposób należy je formatować itd.

Oto przykład prostej procedury obsługi wyjątków, której działanie sprowadza się do wypisania komunikatu z podstawowymi informacjami diagnostycznymi:

Scentralizowany mechanizm informowania o wyjątkach, część 1. (Visual Basic)

```
Sub ReportException( _
    ByVal className, _
    ByVal thisException As Exception _
)
    // raportuj wyjątek
    // nazwa klasy
    // ten wyjątek
```

Patrz też: Szczegółowe omówienie tej techniki można znaleźć w książce *Practical Standards for Microsoft Visual Basic .NET* (Foxall 2003).

```

Dim message As String           // komunikat
Dim caption As String          // tytuł
message = "Wyjątek: " & thisException.Message & ". " & ControlChars.CrLf & _
        "Klasa: " & className & ControlChars.CrLf & _
        "Procedura: " & thisException.TargetSite.Name & ControlChars.CrLf
caption = "Wyjątek"
MessageBox.Show( message, caption, MessageBoxButtons.OK, _
        MessageBoxIcon.Exclamation )
End Sub

```

Ta ogólna procedura obsługująca wyjątki jest wykorzystywana w kodzie w następujący sposób:

```

Scentralizowany mechanizm informowania o wyjątkach, część 2.
(Visual Basic)
Try
    ...
Catch exceptionObject As Exception
    ReportException( CLASS_NAME, exceptionObject )
End Try

```

Kod przedstawionej procedury `ReportException()` jest stosunkowo prosty. W prawdziwej aplikacji taka prostota może być pożądana, ale procedurę można też w dowolny sposób rozbudowywać, dostosowując ją do konkretnych potrzeb.

Jeżeli decydujesz się na zbudowanie scentralizowanego mechanizmu informowania o wyjątkach, nie zapomnij wziąć pod uwagę bardziej ogólnych kwestii związanych ze scentralizowaną obsługą wyjątków, które zostały opisane w punkcie „Wywoływanie procedury (obiektu) obsługi błędu” w podrozdziale 8.3.

Wprowadź standardy pracy z wyjątkami w całym projekcie. Aby obsługa wyjątków była możliwie funkcjonalna i przejrzysta, możesz ujednoczyć zasady korzystania z nich na kilka sposobów:

- Jeżeli pracujesz w języku takim jak C++, który pozwala przekazywać jako dane wyjątku różne obiekty, dane i wskaźniki, określ standardy stosowanych typów. Aby zachować zgodność z innymi językami, możesz przyjąć zasadę przekazywania wyłącznie obiektów dziedziczących po klasie `Exception`.
- Rozważ utworzenie klasy wyjątku specyficznej dla projektu, która posłuży jako klasa bazowa dla wszystkich zgłaszanych w jego ramach wyjątków. Rozwiązanie to można łatwo połączyć ze scentralizowanym i ustandaryzowanym rejestrowaniem, informowaniem o błędach oraz innymi podobnymi mechanizmami.
- Określ sytuacje, w których kod ma prawo używać konstrukcji `throw-catch`, aby przetwarzać błędy lokalnie.
- Określ okoliczności zezwalające na zgłoszenie przez kod wyjątku, który nie będzie obsługiwany lokalnie.
- Zdecyduj o tym, czy będzie stosowany scentralizowany mechanizm informowania o wyjątkach.

- Określ, czy będzie dopuszczalne zgłaszanie wyjątków w konstruktorach i destruktorach.

Patrz też: Wiele alternatywnych metod obsługi błędów zostało opisanych w podrozdziale 8.3 „Mechanizmy obsługi błędów”.

Nie zapominaj o alternatywach dla wyjątków. Wiele języków programowania daje możliwość korzystania z wyjątków już od 5 – 10 lat lub dłużej, ale wiedza o zasadach bezpiecznego ich stosowania wciąż nie jest duża.

Niektórzy programiści używają ich po prostu dlatego, że język został wyposażony w ten mechanizm obsługi błędów, tymczasem należy brać pod uwagę cały arsenał alternatyw: lokalną obsługę błędów, propagowanie błędu przez kod, rejestrowanie go w pliku dziennika, przerywanie pracy i inne. Zapewnianie obsługi błędów poprzez wyjątki tylko dlatego, że język został w nie wyposażony, to klasyczny przykład programowania w języku zamiast *do* języka (więcej na ten temat w podrozdziale 4.3 „Twoje położenie na fali technologii” i w podrozdziale 34.4 „Programuj do języka, a nie w nim”).

Zastanów się zawsze, czy Twój program wymaga mechanizmu wyjątków. Jak zwraca uwagę Bjarne Stroustrup, czasem najlepszą reakcją na poważny błąd czasu wykonania jest zwolnienie wszystkich zasobów i przerwanie pracy. Niech użytkownik uruchomi program ponownie z poprawnymi danymi wejściowymi (Stroustrup 2010).

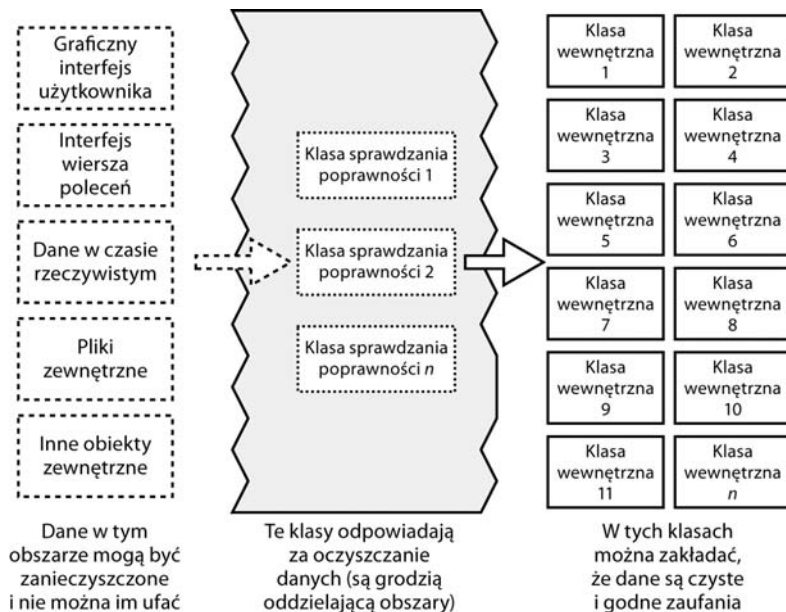
8.5. Ograniczanie zasięgu szkód powodowanych przez błędy

Ograniczanie zasięgu powstałych w wyniku błędów szkód ma na celu opanowanie trudnej sytuacji. Jest to działanie podobne do dzielenia statku szczelnie izolowanymi gradziami. Gdy zderza się on z górą lodową i jego poszycie ulega przerwaniu, zalane przedziały zostają odcięte, a praca załogi w pozostałych jest niezakłócona. Podobną rolę mają specjalne ściany przeciwpożarowe w budynkach (ang. *firewall*). Zapobiegają one rozprzestrzenianiu się ognia na ich kolejne części.

Jedną z metod ograniczania zasięgu szkód jest przypisywanie wybranym interfejsom funkcji granic obszarów „bezpiecznych”. Na takich granicach odbywa się wtedy sprawdzanie poprawności danych, zapewniane są też odpowiednie reakcje. Rysunek 8.2 ilustruje tę technikę.

To samo podejście można stosować na poziomie klasy. Jej metody publiczne mogą zakładać, że dane nie są godne zaufania, a zarazem odpowiadać za ich sprawdzanie i oczyszczanie. Po zaakceptowaniu przez nie danych metody prywatne mogą korzystać z założenia, że są one bezpieczne.

Dobrą analogią dla takiego podejścia może być także sala operacyjna. Przed znalezieniem się w niej dane muszą zostać poddane sterylizacji. Najważniejszą decyzją przy wprowadzaniu takiego schematu jest określenie, co powinno znaleźć się w sali operacyjnej, co poza nią i gdzie należy umieścić drzwi — które procedury będą w bezpiecznej strefie, które poza jej obrębem, a które będą odpowiadały za oczyszczanie danych. Najprostszą metodą jest zazwyczaj



Rysunek 8.2. Wyznaczenie części programu pracującej z zanieczyszczonymi danymi i części operującej na danych, które zostały zweryfikowane, to efektywna metoda zwalniania dużych fragmentów kodu z odpowiedzialności za ich sprawdzanie

oczyszczanie danych zewnętrznych od razu po ich odebraniu. Czasem jednak musi być ono realizowane na różnych poziomach — wtedy pojawia się wiele etapów sterylizacji.

Bezwzględnie konwertuj dane wejściowe na właściwy typ. Dane wejściowe mają zazwyczaj postać ciągu znaków lub liczby. Czasem są one mapowane do typu logicznego, czyli wartości „tak” lub „nie”, a czasem wykonywane jest mapowanie do typu wyczerpującego (na przykład `Color_Red`, `Color_Green` i `Color_Blue`). Utrzymywanie w programie nieprzekształconych danych wejściowych, choćby przez krótki czas, zwiększa złożoność i prawdopodobieństwo, że pewnego dnia pracę programu zakłóci wprowadzenie na przykład koloru „tak”. Dane takie należy jak najszybciej konwertować na właściwą postać.

Ograniczanie zasięgu szkód a asercje

Zagadnienie ograniczania zasięgu szkód pozwala jasno wyznaczyć granicę między stosowaniem asercji a obsługą błędów. Procedury poza chronionym obszarem powinny używać obsługi błędów, ponieważ nie mogą bezpiecznie przyjmować żadnych założeń dotyczących danych. Procedury wewnątrz chronionego obszaru powinny używać asercji, gdyż oczekują, że przekazywane im dane są już oczyszczone. Jeżeli procedura w obszarze chronionym wykrywa błędne dane, jest to błąd w programie, a nie w nich.

Kwestia ograniczania zasięgu szkód zwraca także uwagę na znaczenie podejmowanej na poziomie architektury decyzji o tym, jak realizowana będzie obsługa błędów. Wydzielenie w kodzie obszaru chronionego jest bowiem decyzją dotyczącą architektury systemu.

8.6. Kod wspomagający debugowanie

Kolejnym ważnym aspektem programowania defensywnego jest wykorzystywanie kodu wspomagającego debugowanie. Jest on potężnym sojusznikiem w wykrywaniu błędów.

Nie przenoś każdego ograniczenia wersji finalnej na wersję roboczą

Więcej informacji:

Więcej na temat stosowania kodu wspomagającego debugowanie jako techniki programowania defensywnego można znaleźć w książce *Writing Solid Code* (Maguire 2000).

Zasadą, o której programiści często zapominają, jest to, że wersja robocza oprogramowania nie musi kopiować wszystkich ograniczeń wersji przekazywanej użytkownikom. Program finalny musi pracować szybko, podczas gdy wersja robocza może zazwyczaj działać wolniej. Wersja finalna musi oszczędzać zasoby — robocza może pozwalać sobie na większe ekstrawagancje. Produkt nie powinien udostępniać użytkownikowi możliwości wykonywania niebezpiecznych operacji, natomiast wersja robocza może pozwalać na dodatkowe operacje, których użycie nie jest specjalnie ograniczone.

Przykładowo, jeden z programów, nad którymi pracowałem, szeroko wykorzystywał listy poczwornie powiązane. W kodzie listy nagminnie pojawiały się błędy i często ulegała ona zniszczeniu. Dodałem więc polecenie menu umożliwiające szybką weryfikację jej integralności.

W trybie debugowania program Microsoft Word wykonuje pracujący w pętli kod, który sprawdza integralność obiektu Document co kilka sekund. Pomaga to szybko wykrywać uszkodzenia danych, przez co usprawniony zostaje proces diagnozowania błędów.



W trakcie pracy z kodem warto być przygotowanym na poświęcenie szybkości i zasobów w zamian za możliwość korzystania z dołączanych do kodu narzędzi usprawniających proces programowania i debugowania.

Wcześniej wprowadzaj kod wspomagający debugowanie

Im wcześniej wprowadzisz kod wspomagający debugowanie, tym więcej przyniesie on korzyści. Wielu programistów nie podejmuje takiego wysiłku, dopóki brak ułatwień nie staje się istotnym problemem. Jeżeli jednak napiszesz odpowiedni kod od razu albo zaczniesz stosować narzędzia z innego projektu, będą one pomocą dobrze wykorzystaną i bardzo wartościową.

Stosuj programowanie ofensywne

Patrz też:

Więcej o obsłudze nieoczekiwanych przypadków w punkcie „Budowanie instrukcji case” w podrozdziale 15.2.

Przypadki wyjątkowe powinny być obsługiwane w sposób, który zwróci na nie uwagę w trakcie pracy nad programem, a pozwoli kontynuować wykonywanie kodu przekazanego użytkownikowi. Michael Howard i David LeBlanc nazywają takie podejście programowaniem ofensywnym (Howard i LeBlanc 2003).

Załóżmy, że masz do czynienia z instrukcją case, która ma zapewnić obsługę pięciu rodzajów zdarzeń. W trakcie pracy z kodem należy wtedy wykorzystać

przypadek domyślny do wyświetlania komunikatu: „Hej! Jeszcze jeden przypadek! Napraw program!”. W wersji finalnej ten sam przypadek powinien wywoływać łagodniejszą reakcję, na przykład zapisanie komunikatu do dziennika błędów.

Martwy program powoduje zazwyczaj mniej szkód niż kaleki.
— Andy Hunt i Dave Thomas

Oto kilka technik programowania ofensywnego:

- Zadbaj o to, aby asercje przerywały program. Nie pozwól, aby programiści popadli w nawyk wciskania klawisza *Enter*, aby pominąć znany problem. Niech będzie on na tyle uciążliwy, by musiał zostać naprawiony.
- Wypełnij do końca przydzieloną pamięć, aby wykryć ewentualne błędy alokacji.
- Wypełnij do końca przydzielone pliki i strumienie, aby wykryć błędy formatu pliku.
- Dbaj o to, aby kod w każdej klauzuli `default` lub `else` instrukcji `case` był uciążliwy (przerywał program lub w inny sposób zwracał na siebie uwagę).
- Bezpośrednio przed usunięciem obiektu wypełnij go przypadkowymi danymi.
- Jeżeli jest to dopuszczalne, wyposaż program w mechanizm przesyłania pocztą elektroniczną dzienników błędów, aby mieć dostęp do nieprawidłowości pojawiających się u użytkownika.

Czasem najlepszą obroną jest atak. Pozwól na dotkliwe błędy w trakcie debugowania, a program będzie łagodniejszy dla użytkownika.

Przygotuj się na usuwanie kodu wspomagającego

Jeżeli piszesz program do własnego użytku, pozostawienie elementów wspomagających debugowanie może być dopuszczalne. W przypadku programów komercyjnych wymagania dotyczące ich rozmiarów i szybkości pracy mogą wykluczać takie podejście. Nie można jednak kodu wspomagającego na przemian wstawiać do programu i usuwać. Oto kilka rozwiązań tego problemu:

Patrz też: Więcej o mechanizmach kontroli wersji w podrozdziale 28.2 „Zarządzanie konfiguracją”.

Używaj mechanizmów kontroli wersji i narzędzi kompilacji takich jak *ant* i *make*. Mechanizmy kontroli wersji pozwalają kompilować różne wersje programu z tych samych plików źródłowych. W trybie generowania wersji roboczej narzędzie kompilacji może włączać do kodu wszystkie elementy związane z debugowaniem, natomiast w trybie tworzenia wersji finalnej kod wspomagający można pominąć.

Używaj standardowego preprocesora. Jeżeli stosowane środowisko programowania ma preprocesor — jak na przykład język C++ — kod wspomagający debugowanie może być włączany do kompilacji lub wyłączany z niej przez prostą zmianę parametru kompilatora. Preprocesora można używać bezpośrednio lub za pośrednictwem makra pracującego z definicjami. Oto przykład kodu, który wykorzystuje go bezpośrednio:

Aby włączyć do programu kod wspomagający, używasz `#define` w celu zdefiniowania symbolu `DEBUG`. Aby wyłączyć kod wspomagający, nie definiujesz tego symbolu.

Bezpośrednie wykorzystanie preprocesora do zarządzania kodem wspomagającym debugowanie (C++)

```
#define DEBUG
...

#if defined( DEBUG )
// kod wspomagający debugowanie
...

#endif
```

Ta metoda ma kilka odmian. Zamiast ograniczać się do samego definiowania `DEBUG`, można przypisać makru pewną wartość, a wtedy sprawdzanie, czy definicja istnieje, zastępuje sprawdzanie wartości. Pozwala to różnicować poziomy debugowania. Część kodu wspomagającego je może być potrzebna zawsze — takie fragmenty poprzedzasz instrukcją w rodzaju `#if DEBUG > 0`. Pozostały kod wspomagający może być użyteczny tylko w wyjątkowych przypadkach — wówczas stosujesz instrukcję typu `#if DEBUG == POINTER_ERROR`. Można też wprowadzić hierarchię poziomów debugowania i stosować instrukcje takie jak `#if DEBUG > LEVEL_A`.

Jeżeli nie odpowiada Ci idea kodu upstrzonego instrukcjami `#if defined()`, możesz napisać makro preprocesora o podobnej funkcji. Oto przykład:

Użycie makra preprocesora do zarządzania kodem wspomagającym debugowanie (C++)

```
#define DEBUG
#if defined( DEBUG )
#define DebugCode( code_fragment ) { code_fragment }
#else
#define DebugCode( code_fragment )
#endif
...

DebugCode(
    statement 1;
    statement 2;
    ...
    statement n;
);
...
```

Kod włączany do programu w zależności od tego, czy zdefiniowano `DEBUG`.

Podobnie jak pierwsza technika, również i ta może być modyfikowana na różne sposoby w celu zapewnienia kontroli nad tym, które fragmenty kodu wspomagającego zostaną włączone do kompilacji, a które nie.

Patrz też: Więcej informacji na temat preprocesorów i listę publikacji zawierających porady dotyczące pisania własnych narzędzi tego rodzaju można znaleźć w punkcie „Preprocesory” w podrzdziale 30.3.

Napisz własny preprocesor. Jeżeli język nie został wyposażony w preprocesor, stosunkowo łatwo można napisać własny mechanizm tego rodzaju, który pozwoli włączać do kompilacji i wyłączać z niej kod wspomagający. Można wtedy wybrać niemal dowolną konwencję oznaczania kodu — na przykład w języku Java preprocesor może reagować na słowa kluczowe `//#BEGIN DEBUG` i `//#END DEBUG`. Niezbędnym uzupełnieniem jest skrypt wywołujący preprocesor,

a następnie kompilujący wygenerowany kod. Na dłuższą metę skrypt taki pozwala zaoszczędzić wiele czasu, a jednocześnie zabezpiecza przed omyłkowym skompilowaniem nieprzetworzonego kodu.

Patrz też: Więcej o procedurach zastępczych w punkcie „Budowanie rusztowania do testów pojedynczych klas” w podrozdziale 22.5.

Używaj procedur zastępczych. W wielu sytuacjach operacje związane z debugowaniem może przeprowadzać procedura. W trakcie pracy z programem wykonuje ona kilka takich działań i zwraca kontrolę procedurze wywołującej, natomiast w kodzie finalnym programu zostaje zastąpiona procedurą pustą, która przekazuje kontrolę wywołującej natychmiast lub po wykonaniu jedynie kilku szybkich operacji. Rozwiązanie takie w minimalnym stopniu wpływa na szybkość pracy programu i jest początkowo dużo prostsze niż pisanie własnego preprocesora. Obie wersje procedury muszą być w pewien sposób przechowywane, aby można było wymieniać je w zależności od celu kompilacji.

Oto przykład procedury, która sprawdza przekazywane do niej wskaźniki:

Przykład użycia procedury wspomagającej debugowanie (C++)

```
void DoSomething(
    SOME_TYPE *pointer;
    ...
) {
    // sprawdzanie przekazywanych parametrów
    CheckPointer( pointer );
    ...
}
```

Wywołanie procedury sprawdzającej wskaźnik.

W czasie pracy z kodem procedura `CheckPointer()` może wykonywać kompleksową weryfikację wskaźnika. Proces ten może być swobodnie rozbudowywany, nawet jeżeli ma znaczący wpływ na szybkość pracy programu. Procedura ta może wyglądać tak:

Procedura sprawdzająca wskaźniki w czasie pracy z kodem (C++)

```
void CheckPointer( void *pointer ) {
    // test 1 -- różny od NULL
    // test 2 -- sprawdzanie znacznika (dogtag)
    // test 3 -- sprawdzanie, czy obiekt nie jest uszkodzony
    ...
    // test n -- ...
}
```

Ta procedura sprawdza przekazany do niej wskaźnik. Jest wykorzystywana w czasie pracy z kodem do wykonywania dowolnie rozbudowanej serii testów.

Gdy kod jest gotowy do przekazania użytkownikom, obciążanie programu złożonym mechanizmem sprawdzania wskaźników jest niepożądane. Można wtedy zastąpić procedurę `CheckPointer()` jej uproszczoną wersją:

Ta sama procedura w kodzie wersji finalnej (C++)

```
void CheckPointer( void *pointer ) {
    // procedura pusta, natychmiastowy powrót do miejsca wywołania
}
```

Procedura, która nie wykonuje żadnych operacji.

Nie jest to wyczerpujące omówienie wszystkich możliwych sposobów łączenia elementów wspomagających debugowanie z podstawowym kodem, ale powinno być wystarczające, aby samodzielnie znaleźć rozwiązanie, które będzie dopasowane do potrzeb konkretnego środowiska.

8.7. Ilość kodu defensywnego w wersji finalnej

Jednym z paradoksów programowania defensywnego jest to, że w trakcie pracy nad programem błąd powinien rzucać się w oczy — lepiej, żeby był kłopotliwy, niż żeby został przeoczony. Jednocześnie w wersji przekazywanej użytkownikowi błędy nie powinny sprawiać kłopotów — powinny umożliwiać kontynuowanie pracy lub w uporządkowany sposób ją przerywać. Oto kilka porad pomocnych przy podejmowaniu decyzji o tym, które elementy kodu defensywnego należy pozostawić w kodzie wersji finalnej:

Zostaw kod wykrywający ważne błędy. Określ, w których obszarach programu niewykryte błędy są dopuszczalne, a w których nie. Jeżeli na przykład piszesz program arkusza kalkulacyjnego, możesz pozwolić na ich występowanie w części programu zajmującej się odświeżaniem ekranu, bo będzie to skutkowało wyłącznie problemami z wyświetlanym obrazem. Niedopuszczalne jest jednak niewykrycie błędu w mechanizmie obliczeniowym, ponieważ może on doprowadzić do trudnych do zauważenia zmian w arkuszu danych użytkownika. Zakłócenia w wyświetlaniu danych mają charakter przejściowy i są dla użytkującego program mniej dotkliwe niż wydruk z błędnymi wynikami obliczeń albo zniekształconymi danymi.

Usuń kod wykrywający banalne błędy. Jeżeli konsekwencje błędu są mało istotne, można usunąć wykrywający go kod. Odwołując się do wcześniejszego przykładu, można pozbyć się kodu weryfikującego poprawność odświeżania arkusza. Oczywiście nie chodzi o usuwanie kodu w znaczeniu dosłownym. Należy użyć systemu kontroli wersji, kodu preprocesora lub innej metody, która pozwala na skompilowanie programu bez zbędnego kodu. Jeżeli pojemność pamięci masowej nie jest problemem, można pozostawić kod sprawdzający błędy, ale skierować komunikaty do pliku dziennika.

Usuń kod, który gwałtownie przerywa pracę programu. Jak wspominałem, w trakcie pracy z kodem wykrywane błędy powinny być jak najbardziej widoczne, aby przypominały o konieczności wprowadzenia poprawek. Często najlepszą metodą osiągnięcia tego celu jest wypisanie komunikatu i przerwanie pracy aplikacji. Jest to praktyczne nawet w przypadku drobnych błędów.

Kod przekazywany użytkownikom ma inne wymagania. Użytkownik musi mieć szansę zapisania swojej pracy przed zakończeniem programu i zapewne chętnie zgodzi się na najróżniejsze zakłócenia jego funkcjonowania, jeżeli zyska dzięki temu możliwość zabezpieczenia danych. Żadne ułatwienie debugowania i wynikająca z niego przysza poprawa jakości aplikacji nie wynagrodzą utraty pracy w wyniku nagłej awarii. Jeżeli w programie znajdują się fragmenty, które mogą doprowadzić do zniszczenia, skasowania lub niezapisania danych, należy je z wersji finalnej usunąć.

Zostaw kod, który pozwala łagodnie przerwać pracę programu. Jeżeli program zawiera kod wspomagający debugowanie, który wykrywa błędy mogące uniemożliwić dalszą pracę, pozostaw elementy zapewniające łagodne jej zakończenie. Inżynierowie projektujący sondę marsjańską Pathfinder zostawili w jej programie część elementów wspomagających debugowanie. Gdy po wylądowaniu sondy został wykryty błąd, informacje, które zapewnił pozostawiony kod, umożliwiły zdiagnozowanie problemu i przesłanie do niej nowej wersji oprogramowania, dzięki czemu misja zakończyła się sukcesem (March 1999).

Rejestruj błędy istotne dla personelu pomocy technicznej. Weź pod uwagę pozostawienie w wersji finalnej kodu pomocniczego i wprowadzenie jedynie zmiany jego działania w taki sposób, aby obsługa błędów była niekłopotliwa dla użytkownika. Jeżeli w kodzie jest dużo asercji, które w trakcie pracy z nim przerywają wykonywanie programu, możesz rozważyć zmodyfikowanie procedury asercji tak, aby jedynie rejestrowała komunikaty w pliku.

Dbaj o to, aby komunikaty były zrozumiałe i praktyczne. Gdy pozostawiasz w programie wewnętrzne komunikaty błędów, zadбай o to, aby były sformułowane w języku czytelnym dla użytkownika. W toku jednego z pierwszych w moim życiu projektów programistycznych zadzwonił do mnie użytkownik z wiadomością, że wyświetlił się komunikat „Zła alokacja wskaźnika, spadaj!”. Całe szczęście miał poczucie humoru. Popularnym i praktycznym podejściem jest powiadomienie użytkownika o wystąpieniu „błędu wewnętrznego” i zawarcie w komunikacie adresu e-mail lub numeru telefonu, pod którym można zgłosić wystąpienie problemu.

8.8. Defensywne podejście do programowania defensywnego

Nadmiar jest zawsze zły, ale nadmiar whiskey jest w sam raz.
— Mark Twain

Nadmiar elementów programowania defensywnego może doprowadzić do zupełnie nowych, specyficznych problemów. Jeżeli przekazywane za pomocą parametrów dane podlegają drobiazgowej kontroli w każdym możliwym miejscu, program staje się rozwlekły i powolny. Co gorsza, dodatkowy kod to zawsze dodatkowa złożoność. Ponadto kod ten nie jest bardziej odporny na błędy niż główny kod programu. W nim też mogą wystąpić błędy, a jest to tym bardziej prawdopodobne, że najczęściej nie jest on pisany i rozwijany z równą uwagą co podstawowy kod operacji. Zakres defensywnego programowania musi być przemyślany — nadmiar jest równie szkodliwy jak niedostatek.

cc2e.com/0868

Lista kontrolna: Programowanie defensywne

Programowanie defensywne ogólnie

- Czy procedura jest chroniona przed niepoprawnymi danymi wejściowymi?
- Czy użyłeś asercji do opisu przyjętych założeń, przede wszystkim warunków wstępnych i końcowych?

- Czy stosujesz asercje wyłącznie do opisywania sytuacji, które nigdy nie powinny się zdarzyć?
- Czy architektura lub projekt wysokiego poziomu wyznaczają określony zbiór mechanizmów obsługi błędów, które powinny być stosowane?
- Czy architektura lub projekt wysokiego poziomu definiują, czy obsługa błędów powinna być ukierunkowana na odporność, czy na poprawność?
- Czy wprowadziłeś między obszarami programu podziały zapewniające ograniczenie szkód powodowanych przez błędy i redukujące ilość kodu, w którym niezbędne jest sprawdzanie poprawności danych?
- Czy używałeś w programie kodu wspomagającego debugowanie?
- Czy kod wspomagający debugowanie jest wprowadzany w taki sposób, aby jego aktywowanie i dezaktywowanie nie było kłopotliwe?
- Czy ilość kodu defensywnego jest odpowiednia — nie jest go za dużo ani za mało?
- Czy używałeś technik programowania ofensywnego, aby zabezpieczyć się przed przeoczeniem błędów?

Wyjątki

- Czy w projekcie określone zostało spójne podejście do stosowania wyjątków?
- Czy rozważyłeś alternatywy dla użycia wyjątku?
- Czy lokalna obsługa błędów ma zawsze pierwszeństwo przed zgłaszaniem nielokalnych wyjątków?
- Czy unikasz zgłaszania wyjątków w konstruktorach i destruktorach?
- Czy wszystkie wyjątki zostały dostosowane do poziomu abstrakcji zgłaszających je procedur?
- Czy dane wyjątków obejmują wszystkie potrzebne informacje?
- Czy w kodzie nie ma pustych bloków catch? Jeżeli ich stosowanie jest naprawdę uzasadnione, to czy kod zawiera odpowiednie wyjaśnienie?

Zabezpieczenia

- Czy kod, który sprawdza dane wejściowe, próbuje wykrywać ataki ukierunkowane na przepełnienie bufora, wstrzyknięcie kodu SQL lub HTML, przepełnienie wartości całkowitych lub inne?
- Czy sprawdzane są wyjściowe kody błędów procedur?
- Czy wszystkie wyjątki są przechwytywane?
- Czy komunikaty błędów nie zawierają informacji, które mogłyby zostać wykorzystane przez osobę zainteresowaną włamaniem do systemu?

Więcej informacji

cc2e.com/0875 W doskonaleniu umiejętności programowania defensywnego pomocne będą następujące lektury:

Zabezpieczenia

Howard, Michael, i David LeBlanc. *Writing Secure Code, 2nd Ed.* Redmond, WA, USA, Microsoft Press 2003. Howard i LeBlanc omawiają skutki, jakie może mieć dla bezpieczeństwa systemów nadmierne zaufanie do danych wejściowych. Książka otwiera oczy na świat niezliczonych możliwości przełamywania zabezpieczeń programów. Nie wszystkie, ale wiele z nich ma związek z metodami programowania. Opis obejmuje pełne spektrum zagadnień związanych z wymaganiami, projektowaniem, kodem i testami.

Asercje

Maguire, Steve. *Writing Solid Code.* Redmond, WA, USA, Microsoft Press 1993. Rozdział 2. zawiera doskonale omówienie tematu stosowania asercji ilustrowane ciekawymi przykładami z dobrze znanych produktów firmy Microsoft.

Stroustrup, Bjarne. *Język C++*, Warszawa, WNT 2010. W punkcie 24.3.7.2 autor opisuje kilka metod implementowania asercji w C++ i porusza temat związku między nimi a warunkami wstępnymi i końcowymi.

Meyer, Bertrand. *Programowanie zorientowane obiektowo.* Gliwice, Helion 2005. Książka ta zawiera pełne omówienie tematu warunków wstępnych i końcowych.

Wyjątki

Meyer, Bertrand. *Programowanie zorientowane obiektowo.* Gliwice, Helion 2005. W rozdziale 12. znajduje się szczegółowe omówienie tematu wyjątków.

Stroustrup, Bjarne. *Język C++*, Warszawa, WNT 2010. W rozdziale 14. szczegółowo omówiono mechanizm wyjątków języka C++. W podrozdziale 14.11 znajduje się doskonałe zestawienie 21 porad dotyczących pracy z nimi.

Meyers, Scott. *More Effective C++: 35 New Ways to Improve Your Programs and Designs.* Reading, MA, USA, Addison-Wesley 1996. Techniki o numerach od 9. do 15. to zarazem opis specyficznych cech mechanizmu wyjątków w C++.

Arnold, Ken, James Gosling i David Holmes. *The Java Programming Language, 3rd Ed.* Boston, MA, USA, Addison-Wesley 2000. W rozdziale 8. opisany został mechanizm obsługi wyjątków w języku Java.

Bloch, Joshua. *Effective Java Programming Language Guide.* Boston, MA, USA, Addison-Wesley 2001. W punktach od 39. do 47. opisywane są specyficzne cechy mechanizmu wyjątków języka Java.

Foxall, James. *Practical Standards for Microsoft Visual Basic .NET*. Redmond, WA, USA, Microsoft Press 2003. W rozdziale 10. omówiona została obsługa wyjątków w języku Visual Basic.

Podsumowanie

- Kod przekazywany użytkownikom powinien obsługiwać błędy w bardziej wyszukany sposób niż „śmieci na wejściu, śmieci na wyjściu”.
- Techniki programowania defensywnego ułatwiają wyszukiwanie błędów i poprawianie ich oraz ograniczają powodowane przez nie szkody w finalnej wersji kodu.
- Asercje ułatwiają wczesne wykrywanie błędów, zwłaszcza w systemach dużych, wymagających wysokiej niezawodności i takich, których kod ulega częstym zmianom.
- Wybór sposobu pracy z błędnymi danymi wejściowymi to podstawowa decyzja dotycząca obsługi błędów i jedna z kluczowych decyzji wysokiego poziomu.
- Wyjątki to metoda obsługi błędów operująca w wymiarze innym niż normalny przebieg wykonywania programu. Są cennym narzędziem programowania, o ile używane są w sposób ostrożny. Zawsze warto wziąć pod uwagę inne metody obsługi błędów.
- Ograniczenia wersji finalnej nie zawsze obowiązują w roboczej wersji kodu. Można to wykorzystać, dodając kod, który ułatwi szybkie wykrywanie błędów.

Skorowidz

#define, 479

#if, 243

&&, 479

||, 479

==, 479

A

abstract data type, 162

Abstract Factory, 140

abstrakcja, 125, 127, 174, 193

abstrakcyjna procedura przesłania, 181

abstrakcyjne typy danych, 162, 319

interfejs, 164

klasy, 169

korzyści ze stosowania, 163

niskopoziomowe typy danych, 166

poprawność programu, 164

ukrywanie szczegółów implementacji, 163

wiele instancji danych w środowisku

nieobiektowym, 167

wydajność, 164

zastosowanie, 162

zmiany, 164

Ada, 97, 349

Adapter, 139, 140

adres, 359

ADT, 162, 319

aktualizowanie testu dymowego, 739

akumulatory, 281

Algol, 626

algorytmy, 46, 259, 641

algorytm głosowania, 84

algorytm quicksort, 624

algorytm rekurencyjny poruszania się po

labiryncie, 430

aliasowanie, 372, 373

aliasy typów danych, 346

alokacje wskaźników, 367

alternatywy konstrukcyjne, 123

analiza jakości kodu, 752

kontrolery semantyczne, 752

kontrolery składniowe, 752

narzędzia pomiarowe, 752

analiza wartości granicznych, 547

analiza wymagań, 40

ant, 242

APL, 424

architektura oprogramowania, 77, 91

bloki konstrukcyjne programu, 79

decyzje o wykorzystaniu istniejącej bazy, 85

internacjonalizacja, 82

lista kontrolna, 87

lokalizacja, 82

nadmiar konstrukcyjny, 84

niezawodność, 84

ogólna jakość architektury, 86

organizacja danych, 79

organizacja programu, 78

podstawowe klasy programu, 79

projekt interfejsu użytkownika, 80

przetwarzanie błędów, 82

reguły biznesowe, 80

skalowalność, 81

składniki, 78

strategia zmian, 85

wejście-wyjście, 82

wprowadzanie zmian, 78

współdziałanie, 81

wydajność, 81

wykonalność, 84

zabezpieczenia, 81

zabezpieczenia przed awariami, 83

zarządzanie zasobami, 80

ARRAY_LENGTH(), 346

ASCII, 676

assembler, 96, 97, 676

asercje, 225, 248

budowanie mechanizmu asercji, 227

ograniczanie zasięgu szkód

powodowanych przez błędy, 240

stosowanie, 227

warunki końcowe, 228

warunki wstępne, 228

wskaźniki, 367

assert, 227

atrybuty obiektów, 123

audyty zewnętrzne, 503

auto_ptr, 369

automatyczne inicjalizowanie wszystkich

zmiennych, 281

automatyzacja codziennej kompilacji i testu, 739

automatyzacja testów, 562

autor, 520

awaria, 83

AWK, 750

B

badanie zgodności liczby nawiasów, 475
 Basic, 100
 baza danych, 121
 bazy błędów, 561
 BDUF, 155
 begin, 777
 Beizer Boris, 543, 552, 562
 Bentley Jon, 626, 859
 bezpieczne przekształcanie kodu, 613
 bezpieczne refaktoryzowanie, 618
 białe znaki, 774

- grupowanie, 774
- puste wiersze, 775
- wcięcia, 775

 biblioteki kodu, 755
 Big Bang, 727
 Big Design Up Front, 155
 binding time, 290
 bity znaczników, 839
 bloki, 480
 bloki konstrukcyjne programu, 79
 błędne wskaźniki, 282
 błędy, 82, 550, 635, 862

- błędy programisty, 552, 553
- błędy składniowe, 583, 594
- błędy w procesie testowania, 555
- dzielenie przez zero, 328
- kategorie błędów, 551
- literówki, 553
- obsługa błędów, 230
- ograniczanie zasięgu szkód, 239
- wskaźniki, 366
- zaokrąglanie liczb, 332

 Boehm Barry, 626
 Boeing Computer Services, 527
 BOOLEAN, 338
 bounds checker, 561
 braki w przygotowaniu projektu, 59
 break, 407, 409, 417, 418, 607, 647

- etykiety, 418
- redukowanie głębokiego zagnieżdżenia, 483

 breakpoint, 282, 561
 Bridge, 139, 140
 Brooks Fred, 710
 brudny test, 538
 brute force, 582, 583
 budowa domów, 52
 budowa oprogramowania, 37, 50

- czynności, 38

budowanie harmonogramu, 705

- cele, 706
- metody szacowania ilości pracy, 706
- opóźnienia projektu, 710
- prawo Freda Brooksa, 710
- szacowanie ilości pracy przy programowaniu, 707
- szacowanie wielkości projektu, 711
- techniki szacowania, 707
- wpływ różnych czynników
 - na harmonogram pracy, 708
- wymagania, 706
- zmniejszenie zakresu projektu, 710

 budowanie klas, 251, 252
 budowanie pętli od wewnątrz, 423
 budowanie procedur, 251, 252, 253
 budowanie systemu, 690
 budowanie własnych narzędzi, 759
 buforowanie, 651, 663
 bug, 569
 build tool, 754
 ByRef, 844
 ByVal, 844

C

C, 96, 98

- ciągi znakowe, 335
- konwencje nazw, 313, 316
- tablice, 346
- wskaźniki, 370
- wyrażenia logiczne, 479

 C#, 98
 C++, 96, 98, 195

- assert, 227
- case, 401
- globalna przestrzeń nazw, 301
- instrukcje puste, 481
- klasy, 192
- konwencje nazw, 313, 315
- makra, 218
- preprocesor, 242
- procedury inline, 220
- tworzenie typów danych, 347
- typy wyliczeniowe, 339
- wskaźniki, 362, 368
- wyjątki, 235
- wyrażenia logiczne, 479

 caching, 663
 case, 183, 292, 398, 403, 492, 648

- budowanie instrukcji, 399

 C++, 401, 402

- default, 400
 - formatowanie kodu, 788
 - głębokie zagnieżdżenia, 485
 - Java, 401
 - przechodzenie do następnego przypadku, 401, 402
 - wybór optymalnej kolejności przypadków, 399
 - wykrywanie błędów, 401
 - CASE, 748
 - catch, 237
 - caveat emptor, 224
 - cechy charakteru programisty, 855
 - ciekawość, 858
 - doświadczenie, 868
 - dyscyplina, 865
 - gorączka programowania, 869
 - inteligencja, 857
 - komunikacja, 865
 - kreatywność, 865
 - lenistwo, 866
 - nawyki, 869
 - osobowość, 856
 - skromność, 857
 - uczciwość intelektualna, 862
 - współpraca, 865
 - wytrwałość, 867
 - cechy projektu, 115
 - centralizacja punktów kontroli, 143, 189
 - charakter programisty, 856
 - check out, 702
 - ciągi znakowe, 333, 351
 - C, 335
 - deklaracja, 335
 - internacjonalizacja, 334
 - konwersja między typami ciągów znakowych, 334
 - lokalizacja, 334
 - pomyłki o jedną pozycję, 334
 - string, 335
 - Unicode, 334
 - wskaźniki, 335
 - ciekawość, 858
 - class-responsibility-collaboration, 154
 - COBOL, 98, 569
 - Cocomo II, 96
 - codzienna kompilacja, 738, 743
 - projekty, 742
 - wykrywanie nieudanych kompilacji, 739
 - zasady, 738
 - codzienny test dymowy, 739
 - collaborative software construction, 514
 - Composite, 140
 - computer-aided software engineering, 748
 - const, 220, 281
 - Constantine Larry, 204
 - continue, 417, 418
 - corectness, 233
 - CppUnit, 558
 - czarna skrzynka, 127
 - czas aktywności zmiennych, 284
 - czas wiązania, 290
 - czas wykonywania operacji, 636
 - czasopisma dla programistów, 895
 - częstość integracji, 727
 - czynniki wpływające na ilość pracy, 709
 - czyste bloki, 776
 - czysty test, 538
 - czytanie kodu, 528
 - czytelność kodu, 773, 877
- ## D
- dane, 79, 276
 - dane iteracyjne, 292
 - dane sekwencyjne, 292
 - dane selektywne, 292
 - dane składowe, 185
 - dane Unicode, 334
 - dane wędrowne, 375
 - dane globalne, 131, 371, 839
 - aliasowanie, 372
 - alternatywy stosowania, 375
 - często używane dane, 375
 - dane wędrowne, 375
 - dostęp bezpośredni, 378
 - efekty uboczne, 372
 - emulacja stałych nazwanych, 374
 - emulacja typów wyliczeniowych, 374
 - inicjalizowanie, 373
 - jednolite operacje na złożonych danych, 379
 - kod wielobieżny, 373
 - kontrola dostępu, 377
 - konwencja nazw, 379
 - mechanizm blokowania, 377
 - niejednolite operacje na złożonych danych, 379
 - niepewność kolejności, 373
 - niszczenie modularności, 374
 - ponowne użycie kodu, 373
 - problemy, 371
 - procedury dostępowe, 375, 378
 - przesłanki stosowania, 374
 - redukcja liczby problemów, 379
 - ujawnianie złożoności, 374
 - Data Encryption Standard, 639

- deasembler, 758
- DEBUG, 243
- Debug.Assert(), 228
- debugger, 268, 592
 - debugger symboliczny, 560
 - debugger systemowy, 593
- debugowanie, 241, 534, 569
 - błędy składniowe, 583
 - brute force, 582, 583
 - debugger, 592
 - debugowanie oparte na przesądach, 574
 - dystans psychologiczny, 590
 - dział i rząd, 584
 - efektywność, 570
 - integracja przyrostowa, 581
 - jakość oprogramowania, 570
 - limit czasu dla szybkiego debugowania, 583
 - lista kontrolna, 594
 - lokalizowanie źródła błędu, 577
 - metody usuwania defektów, 572
 - narzędzia, 579, 591, 757
 - narzędzia porównujące kod, 591
 - naukowa metoda debugowania, 574
 - negatywne wyniki testów, 580
 - nieefektywne debugowanie, 573
 - ostatnio zmieniany kod, 581
 - ostrzeżenia kompilatora, 591
 - platformy testowania, 592
 - podjęty obszar kodu, 580
 - poszukiwanie hipotez, 580
 - programy profilujące, 592
 - psychologia, 588
 - replikacja błędów, 579
 - rusztowanie, 592
 - sprawdzanie składni, 592
 - stabilizacja błędów, 576
 - tworzenie hipotez, 578
 - usuwanie defektu, 585
 - wpływ nastawienia psychicznego
 - na dostrzeżenie błędów, 589
 - wykorzystanie defektów, 571
 - wyszukiwanie defektu, 574
- Decorator, 139, 140
- decyzje konstrukcyjne, 95
- default, 400
- defekty programistyczne, 554
- defensywne podejście do programowania
 - defensywnego, 246
- definicja problemu, 51, 70, 71
- definicja produktu, 70
- definicje typów, 311
- definiowanie typów danych w komunikatach, 457
- deklaracja
 - dane, 797, 830
 - procedury, 262
 - wskazniki, 361
- deklaracja zmiennych, 277
 - niejawne, 277
 - zmienne stanu, 305
- Dekorator, 140
- dekrementacja, 796
- delete, 289, 366
- Delphi, 676
- DES, 639
- diagramy, 144
- diagramy UML, 146, 154
- Diff, 591, 750, 757
- Dijkstra Edsger, 114, 490
- długość instrukcji, 789
- długość nazw zmiennych, 300
- długość pętli, 422
- długość procedury, 209
- dni miesiąca, 451
- dobra hermetyzacja, 175
- dobra nazwa procedury, 207
- dobry komentarz, 852
- dobry interfejs klasy, 169
- dotądki ad hoc, 86
- dotawianie
 - liczby całkowite, 660
 - liczby zmiennoprzecinkowe, 660
- dog-and-pony show, 529
- dokument opisujący projekt szczegółowy, 814
- dokumentacja, 153, 813, 851
 - klasy, 815
 - komentarze, 817
 - nazwy danych, 816
 - normy IEEE, 849
 - organizacja danych, 816
 - procedury, 816
 - projekt, 813, 817
 - sterowanie, 816
 - styl programowania, 814
 - układ kodu, 817
 - zasady pisania dobrych komentarzy, 821
 - zewnętrzna dokumentacja programu, 813
- Don't Repeat Yourself, 599
- done, 306
- DoNothing(), 481
- dostęp do bazy danych, 121
- dostęp do zmiennych globalnych, 377
- doświadczenie, 868
- DRY, 599

Dużo projektowania na początku, 155
 dyscyplina, 865
 dystans psychologiczny, 590
 dziedziczenie, 126, 180, 194

- dziedziczenie procedur, 181
- dziedziczenie prywatne, 179
- dziedziczenie publiczne, 180
- dziedziczenie wielokrotne, 184
- głębokie drzewo dziedziczenia, 183
- reguły stosowania, 185
- wielodziedziczenie, 184
- zasada podstawienia Liskov, 180

 dziel i rządź, 147, 148, 584
 dzielenie całkowite, 329
 dzielenie programu na poziomy abstrakcji, 882
 dzielenie się wiedzą, 516

E

efekty uboczne, 796

- modyfikacja zmiennej globalnej, 372

 efektywne inspekcje, 526
 efektywne komentowanie kodu, 824
 efektywność debugowania, 570
 efektywność pracy, 687
 efektywność procedury, 258
 eklektyzm, 888
 eksperymenty, 858, 889
 elastyczność, 136
 elastyczny format komunikatu, 454
 elementy typów wyliczeniowych, 312
 eliminowanie wspólnych części wyrażień, 673
 else, 395, 492
 emulacja stałych nazwanych, 374
 emulacja typów wyliczeniowych, 374
 emulowane czyste bloki, 777
 end, 777
 Enough Design Up Front, 155
 ENUF, 155
 enum, 220
 enumeracja, 338
 EOFException, 236
 error, 306
 errors per K lines of code, 686
 ErrorType_Fatal, 134
 ErrorType_None, 134
 ErrorType_Warning, 134
 ewolucja oprogramowania, 598
 Exception, 238
 exit, 427
 Exit, 427
 Extreme Programming, 507

F

Fabryka Abstrakcyjna, 140
 fabryka obiektów, 489
 Facade, 139, 140
 Factory Method, 139, 140
 false, 469, 470
 Fasada, 140
 filozofia wygody, 288
 filozofia złożoności, 288
 final, 180, 281
 finally, 440
 firewall, 239
 for, 289, 292, 303, 409, 412
 foreach, 410
 For-Each, 410
 for-in, 410
 formalizacja dokumentów, 684
 formalizacja projektu, 152
 formalne inspekcje, 519, 525

- autor, 520
- cele, 524
- ego, 524
- kierownictwo, 521
- kontrola, 523
- lista kontrolna, 526
- moderator, 520
- optymalizacja inspekcji, 524
- pisarz, 521
- planowanie, 521
- procedura inspekcji, 521
- przebudowa kodu, 523
- przeгляд, 522
- przygotowania, 522
- raport z inspekcji, 523
- recenzent, 521
- role, 520
- spotkanie, 522
- trzecia godzina spotkania, 523
- zastosowanie, 520

 formalne przeglądy techniczne, 503
 formatowanie deklaracji danych, 797
 formatowanie instrukcji, 789

- deklaracja danych, 797
- długość instrukcji, 789

 efekty inkrementacji i dekrementacji, 796
 instrukcja przypisania, 794
 kontynuacja instrukcji przypisania, 795
 kontynuacja instrukcji sterujących, 794
 kontynuacja wiersza, 791
 kontynuacja wywołań procedur, 792

- formatowanie instrukcji
 - niekompletność instrukcji, 791
 - silnie powiązane elementy, 792
 - spacje, 790
 - typy wskaźnikowe, 799
 - uporządkowanie deklaracji, 799
 - wiele instrukcji w jednym wierszu, 795
 - wskaźniki, 799
 - zakończenie kontynuacji, 793
 - zmiennie, 799
 - zwiększanie przejrzystości, 790
 - formatowanie klas, 804
 - formatowanie podziałów między klasami, 805
 - układ implementacji klasy, 804
 - układ interfejsu klasy, 804
 - układ kodu w plikach, 807
 - układ kodu w programie, 807
 - formatowanie kodu, 768
 - białe znaki, 774
 - cele, 773, 774
 - czyste bloki, 776
 - czytelność kodu, 773
 - emulowane czyste bloki, 777
 - gęstość informacji, 774
 - głębokie wcięcia, 780
 - logiczna struktura kodu, 773
 - narzędzia, 750
 - nawiasy, 775
 - nawiasy klamrowe, 779
 - puste wiersze, 775
 - style formatowania, 776
 - techniki formatowania, 774
 - wcięcia, 775
 - wyrównywanie z głębokimi wcięciami, 780
 - wyróżnianie granic bloków parami begin-end, 779
 - formatowanie komentarzy, 800
 - wcięcia, 800
 - formatowanie procedur, 802
 - wcięcia dla argumentów, 802
 - formatowanie struktur sterujących, 782
 - bloki z jedną instrukcją, 785
 - case, 788
 - goto, 787
 - pary begin-end, 783
 - podwójne wcięcia, 783
 - puste wiersze, 784
 - składowe złożonych warunków, 786
 - Fortran, 98, 761
 - found, 306
 - foundation classes, 53
 - function, 217
 - Function, 427
 - funkcje, 197, 217
 - funkcje inline, 481
 - stosowanie, 217
 - wartość zwracana, 218
 - funkcje składowe, 185
- ## G
- garbage in, garbage out, 224
 - Gates Bill, 870
 - generatory danych testowych, 558
 - generatory hierarchii klas, 751
 - generatory kodu, 756
 - generatory odnośników, 751
 - generatory opisu interfejsu, 751
 - generowanie kodu, 754
 - gęstość defektów, 686
 - gęstość komentarzy, 828
 - globalna przestrzeń nazw, 301
 - głębokie drzewo dziedziczenia, 183
 - głębokie kopiowanie obiektów, 187
 - głębokie wcięcia, 780
 - głębokie zagnieżdżenia, 482
 - break, 483
 - case, 485
 - if, 482
 - if-then-else, 484
 - metody redukowania zagnieżdżeń, 482, 490
 - podejście obiektowe, 488
 - procedury, 486
 - Główny Imperatyw Techniczny Oprogramowania, 113, 114
 - gorączka programowania, 869
 - goto, 407, 434, 443, 445, 447, 787
 - argumenty przeciwko stosowaniu, 434
 - argumenty za stosowaniem, 435
 - przetwarzanie błędów, 437
 - stosowanie, 443
 - try-finally, 440
 - wspólna klauzula else, 442
 - zmiennie stanu, 439
 - granice tablic, 345
 - gromadzenie informacji o testach, 563
 - prywatne zbiory informacji o testach, 564
 - grupowanie powiązanych instrukcji, 390
 - grupy powiązanych instrukcji, 288
 - GUI, 101

H

hack, 268
 haki, 268
 Hansen W.J., 301
 hard-coding, 290
 harmonogram, 705
 hermetyzacja, 175, 193, 236, 601
 hermetyzacja kolekcji, 606
 hermetyzacja szczegółów implementacji, 126
 heurystyczne wskazówki, 46
 heurystyki, 46, 122, 141, 144

- abstrakcje, 125
- centralizacja punktów kontroli, 143
- czas wiązania, 143
- diagramy, 144
- dziedziczenie, 126
- hermetyzacja szczegółów implementacji, 126
- hierarchia, 141
- klasy bazowe, 125
- kohezja, 141
- kontrakty klas, 142
- luźne powiązania, 135
- modularność projektu, 144
- obiekty świata rzeczywistego, 123
- obszary potencjalnych zmian, 133
- projektowanie pod kątem testów, 142
- stosowanie, 145
- ukrywanie informacji, 127
- unikanie niepowodzeń, 143
- użycie najbardziej prymitywnych środków, 143
- wzorce projektowe, 139
- zakresy odpowiedzialności, 142

 hierarchia, 141
 high-tech, 715
 high-touch, 715
 hipoteza Sapira-Whorfa, 97
 hodowanie systemu, 48
 hodowla ostryg — przyrastanie systemu, 49
 homonimy, 323
 Hopper Grace, 569
 housekeeping, 414
 Howard Michael, 241
 Humphrey Watts, 555
 Hunt Andrew, 599
 hybrid coupling, 294

I

I18n, 82
 IDE, 748
 idea wykonywania małych kroków, 49

idealny projekt informatyczny, 41
 identyfikacja

- artefakty projektu, 698
- obiekty, 123
- obszary potencjalnych zmian, 133

 IEEE, 849
 if, 292, 393

- czytelność kodu, 394
- głębokie zagnieżdżenia, 482
- łańcuchy instrukcji if-then-else, 396
- typowe przypadki, 397

 if-then, 393, 403, 492
 if-then-else, 292, 395, 396, 403, 440, 484, 492, 648
 ilość czasu poświęcanego na przygotowania, 89
 ilość kodu defensywnego w wersji finalnej

- programu, 245

 implementacja określania wysokości

- stawki ubezpieczenia, 452

 IN, 212
 indeks długości ciągu, 663
 indeksowanie, 462
 indeksy pętli, 303, 425
 indeksy pomocnicze, 662
 indeksy tablic, 345
 inicjalizacja

- dane globalne, 373
- pamięć robocza, 282
- składowe klasy, 281
- w czasie kompilacji, 667
- wskaźniki, 361
- zmienne, 278, 287, 295

 inkrementacja, 796
 inline, 220
 inspection, 519
 inspekcja, 519, 531
 instrukcje maszynowe, 882
 instrukcje puste, 481

- C++, 481
- DoNothing(), 481

 instrukcje sekwencyjne, 292
 instrukcje warunkowe, 393

- case, 398
- if, 393
- lista kontrolna, 403
- switch, 398

 instrukcje złożone, 480

- nawiasy klamrowe, 480

 integracja, 725, 737, 743

- Big Bang, 727
- codzienna kompilacja, 738
- częstość integracji, 727

- integracja
 - integracja ciągła, 742
 - integracja funkcjonalna, 735
 - integracja końcowa, 727
 - integracja przyrostowa, 728
 - integracja typu T, 737
 - integracja warstwowa, 734
 - integracja według ryzyka, 735
 - integracja wstępująca, 733
 - integracja zstępująca, 730
 - lista kontrolna, 743
 - lokalizacja błędów, 729
 - metody integracji, 725
 - monitorowanie postępów w pracy, 729
 - przyrostowe strategie integracji, 730
 - relacje z klientem, 730
 - test dymowy, 738
 - testowanie jednostek systemu, 730
 - top-down, 732
 - integrated development environment, 748
 - inteligencja, 857
 - interakcje z systemem operacyjnym, 624
 - interfejs klasy, 169, 600
 - interfejs użytkownika, 80, 121
 - internacjonalizacja, 82, 334
 - internationalization, 82
 - interpretacja programu, 770
 - interpretowany język skryptowy, 99
 - inżynieria oprogramowania, 894
 - iteracyjne uszczegóławianie, 255
 - Iterator, 140
 - iterowanie, 147, 492, 639, 886
 - izolowanie złożoności, 188
- J**
- jakość oprogramowania, 499, 570
 - adaptacyjność, 500
 - audyty zewnętrzne, 503
 - cele, 502, 504
 - czytelność, 500
 - dokładność, 500
 - efektywność, 499
 - elastyczność, 500
 - formalne przeglądy techniczne, 503
 - funkcjonalność, 499
 - integralność, 499
 - kontrola jakości, 509
 - łatwość konserwacji, 500
 - metody podwyższania jakości, 502
 - możliwość ponownego użycia, 500
 - nieformalne przeglądy techniczne, 503
 - niezawodność, 499
 - normy, 512
 - odporność, 500
 - Ogólna Zasada Jakości Oprogramowania, 509
 - określanie celów, 504
 - poprawność, 499
 - proces budowy oprogramowania, 503
 - przeñośność, 500
 - skuteczność metod podwyższania jakości, 505
 - strategia testowania, 502
 - testowalność, 500
 - wyodrębnienie kontroli jakości, 502
 - zasady budowy oprogramowania, 503
 - zrozumiałość, 500
 - jakość wymagań, 76
 - jamming, 653
 - Java, 96, 99, 195, 631
 - case, 401
 - const, 281
 - final, 281
 - inicjalizowanie zmiennych, 280
 - konwencje nazw, 314, 315
 - nazwy zmiennych, 298
 - testowanie przepływu danych, 545
 - wyjątki, 235
 - wrażenia logiczne, 480
 - Javadoc, 843
 - JavaScript, 99
 - jawna konwersja typu, 370
 - język interpretowany, 634
 - język naturalny, 253
 - język niskiego poziomu, 675
 - język obiektowy, 99
 - język programowania, 95
 - Ada, 97
 - Algol, 626
 - assembler, 97
 - C, 98
 - C#, 98
 - C++, 98
 - Cobol, 98
 - Fortran, 98, 761
 - Java, 99
 - JavaScript, 99
 - Perl, 99
 - PHP, 99
 - Python, 99
 - SQL, 100
 - Visual Basic, 100
 - język skryptowy, 99

język translacji wzorów, 761
 język UML, 154
 Jones Capers, 551, 828
 JUnit, 558
 JVM, 631

K

karty class-responsibility-collaboration, 154
 kategorie błędów, 551
 Kernighan Brian, 282
 kierownictwo, 521
 klasa-odpowiedzialność-współpraca, 154
 klasy, 121, 122, 161, 169, 194, 310
 abstrakcja, 174
 bezpośrednie połączenia, 186
 błędy projektowania, 191
 budowanie, 251, 252
 budowanie procedur, 252
 C++, 192
 centralizacja punktów kontroli, 189
 dane składowe, 185
 dziedziczenie, 180
 erozja abstrakcji interfejsu w toku modyfikacji, 173
 formatowanie kodu, 804
 funkcje składowe, 185
 głębokie drzewo dziedziczenia, 183
 hermetyzacja, 175
 inicjalizacja składowych danych, 281
 interfejs, 169
 interfejs programistyczny, 173
 izolowanie złożoności, 188
 klasy bazowe, 125, 182
 klasy pochodne, 181, 182
 klasy zaprzyjaźnione, 177
 kohezja, 174
 komentarze, 846
 konstruktory, 186
 kontrakty, 142
 lista kontrolna, 193
 łączenie pokrewnych operacji, 190
 modelowanie obiektów abstrakcyjnych, 188
 modelowanie obiektów świata rzeczywistego, 188
 nazwy plików z kodem, 808
 ograniczanie dostępności, 175
 ograniczanie wpływu zmian, 189
 pakiety, 192
 planowanie pod kątem rodzin programów, 190
 ponowne użycie kodu, 190
 problemy implementacji, 179
 problemy projektowania, 179
 procedury komplementarne, 173
 przesłanki dla utworzenia klasy, 188
 publiczne procedury, 172
 refaktoryzacja, 191
 refaktoryzacja interfejsu, 609
 relacja „jest”, 180
 relacja „ma”, 179
 semantyczne łamanie hermetyzacji, 177
 specyfika języka, 192
 spójny poziom abstrakcji w interfejsie, 171
 testowanie, 252
 tworzenie, 252
 tworzenie ogólnego projektu klasy, 252
 ujawnianie danych składowych, 175
 ujawnianie szczegółów implementacji, 176
 układ kodu w plikach, 807
 ukrywanie danych globalnych, 189
 ukrywanie informacji, 127
 ukrywanie szczegółów implementacji, 176, 189
 usprawnianie przekazywania parametrów, 189
 wielodziedziczenie, 184
 wygoda „czasu czytania”, 177
 zasada podstawienia Liskov, 180
 zawieranie, 179
 zmniejszanie złożoności, 188
 źle dobrana abstrakcja, 170
 klasy dobrych danych, 549
 klasy fundamentowe, 53
 klasy złych danych, 548
 klasyfikacja znaków, 450
 KLOC, 686
 klucze wyszukiwania, 461
 kluczowe decyzje konstrukcyjne, 95
 język programowania, 95
 konwencje programowania, 100
 podstawowe praktyki programowania, 103
 technologia, 101
 Knuth Donald, 626
 kod błędu, 231
 kod defensywny w wersji finalnej programu, 245
 kod liniowy, 385
 kod programu, 696
 kod rekurencyjny, 430
 kod spaghetti, 435, 753
 kod wielobieżny, 373
 kod wspomagający debugowanie, 241
 kod źródłowy, 41
 kohezja, 141, 174, 204, 600
 kohezja czasowa, 205
 kohezja funkcjonalna, 204
 kohezja komunikacyjna, 205

- kohezja
 - kohezja logiczna, 206
 - kohezja proceduralna, 206
 - kohezja przypadkowa, 207
 - kohezja sekwencyjna, 204
- kolejność czytania kodu, 389
- kolejność instrukcji, 385
- komentarze, 255, 602, 800, 817
 - algorytmy, 845
 - bity znaczników, 839
 - bloki instrukcji, 840
 - błędy, 836
 - case, 840
 - cel bloku kodu, 831
 - dane globalne, 839
 - dane wejściowe, 844
 - dane wyjściowe, 844
 - deklaracja danych, 830
 - efektywne komentowanie kodu, 824
 - gęstość komentarzy, 828
 - globalne skutki wykonania procedury, 845
 - grupy wierszy, 830
 - if, 840
 - informacje niemożliwe do wyrażenia w kodzie, 824
 - interfejs klasy, 846
 - Javadoc, 843
 - jednostki danych numerycznych, 838
 - kategorie, 822
 - komentarze do akapitów kodu, 831
 - komentarze do deklaracji danych, 838
 - komentarze do klas, 846
 - komentarze do plików, 846
 - komentarze do pojedynczych wierszy, 828
 - komentarze do procedur, 841
 - komentarze do programów, 846
 - komentarze do struktur sterujących, 840
 - komentarze informujące o przeznaczeniu kodu, 823
 - komentarze na końcu wierszy, 829, 830
 - komentarze podsumowujące, 823
 - metody pisania komentarzy, 828
 - nadmiar komentarzy, 834
 - narzędzia wspomagające opisywanie kodu, 843
 - niespodzianki, 834
 - nota copyright, 847
 - objaśnienia do kodu, 822
 - odstępstwa od dobrego stylu programowania, 837
 - ograniczenia dotyczące danych wejściowych, 839
 - ograniczenia procedur, 844
 - opisy klas, 846
 - opisywanie pojedynczych wierszy, 829
 - optymalna liczba komentarzy, 828
 - oznaczanie części programu, 845
 - paradygmat książki w opisie programu, 848
 - parametry, 842
 - pętle, 840
 - powtórzenia kodu, 822
 - proces programowania, 827
 - przeznaczenie pliku, 846
 - różnicowanie między poziomami komentarzy, 835
 - skrót, 835
 - sposób projektowania klasy, 846
 - struktury sterujące, 840
 - wydajność, 827
 - wyszukany kod, 837
 - zakodowane wartości, 838
 - zakończenia bloków, 831
 - zakończenia pętli, 841
 - zakresy wartości liczbowych, 838
 - założenia interfejsu, 844
 - zasady pisania dobrych komentarzy, 821
 - zawartość pliku, 846
 - znacznik wersji, 847
 - znaczniki w kodzie, 823
- komentarze nagłówkowe, 259
- kompilacja, 624
- kompilacja procedury, 267
- kompilator, 754
- kompletność wymagań, 77
- Kompozyt, 140
- komputer mainframe, 101
- komputerowe wspomaganie inżynierii
 - oprogramowania, 748
- komunikacja, 684, 865
- komunikaty kompilatora, 584
- koncentracja na programowaniu, 40
- koncentracja uwagi, 880
- konfiguracje komputerów, 703
- konstruktory, 186, 281
 - prywatny konstruktor, 187
 - singleton, 187
- kontrakty klas, 142
- kontrola dostępu do pamięci, 282
- kontrola dostępu do zmiennych globalnych, 377
- kontrola jakości, 509
- kontrola wersji, 702, 753
- kontrola zmian, 698, 700
- kontrolery semantyczne, 752
- kontrolery składniowe, 752
- kontynuacja wiersza, 791
- konwencje nazw, 308, 325
 - C, 313, 316
 - C++, 313, 315
 - Java, 314, 315

- konwencje specyficzne dla języka, 313
- nieformalne konwencje, 310
- programowanie w wielu językach, 314
- stopnie formalizmu, 310
- Visual Basic, 314, 316
- wprowadzanie, 309
- zbiór konwencji nazw, 315
- zmienne globalne, 379
- konwencje obsługi błędów, 83
- konwencje programowania, 100, 880
- konwersja typów, 328, 370
 - konwersja między typami ciągów znakowych, 334
- kopiowanie głębokie, 187
- kopiowanie obiektów, 187
- koszt typowych operacji, 635
- koszt usuwania defektów, 63, 508
- koszt wyszukiwania defektów, 508
- koszt zmian, 701
 - zmiany w specyfikacji wymagań, 74
- kreatywność, 865
- krokowe wykonywanie kodu, 560
- krótkie nazwy, 319, 325
- kryteria oceny zależności między modułami, 136
- Kuhn Thomas, 859
- kultura organizacji, 516
- kwalifikatory, 301
- kwalifikatory wyników obliczeń, 301

L

- L10n, 82
- lazy evaluation, 651
- LDUF, 155
- leanness, 116
- LeBlanc David, 241
- lenistwo, 866
- leniwe obliczanie, 651
- liczba ścieżek komunikacji, 684
- liczby, 327, 351
 - błędy dzielenia przez zero, 328
 - konwersja typów, 328
 - magiczne liczby, 328
 - ostrzeżenia kompilatora, 329
- liczby całkowite, 329, 351
 - dzielenie całkowite, 329
 - przepelnienie, 329
 - przepelnienie w wynikach pośrednich, 330
- liczby Fibonacciego, 433
- liczby zmiennoprzecinkowe, 331, 351
 - błędy zaokrągleń, 332
 - sprawdzanie równości, 331

- licznik kontrolny, 416
- liczniki, 281
- linker, 754
- linker nakładkowy, 754
- Lint, 752
- Liskov Barbara, 180
- Liskov Substitution Principle, 180
- lista kontrolna
 - architektura oprogramowania, 87
 - bezpieczne refaktoryzowanie, 618
 - dane, 295
 - dane globalne, 380
 - debugowanie, 594
 - formatowanie, 809
 - inspekcje, 526
 - instrukcje warunkowe, 403
 - integracja, 743
 - jakość klasy, 193
 - kod, który opisuje się sam, 815
 - komentarze, 852
 - metody oparte na tabelach, 467
 - metody optymalizacji kodu, 678
 - najważniejsze praktyki programowania, 103
 - narzędzia programowania, 763
 - nazwy zmiennych, 325
 - nietypowe struktury sterowania, 446
 - optymalizacja kodu, 642
 - organizacja kodu liniowego, 391
 - pętle, 425
 - plan kontroli jakości, 511
 - podstawowe typy danych, 351
 - procedury wysokiej jakości, 221
 - Proces Programowania w Pseudokodzie, 271
 - programowanie defensywne, 246
 - programowanie w parach, 519
 - projektowanie, 158
 - przesłanki refaktoryzacji, 604
 - przygotowania, 93
 - refaktoryzacja, 611
 - struktury, 380
 - struktury sterujące, 496
 - testowanie, 566
 - typy danych, 380
 - wskazniki, 381
 - wymagania, 75
 - zarządzanie konfiguracją, 704
 - zasady pracy z danymi, 295
 - zmienne, 295
- listingi asemblera, 758
- listy parametrów, 358, 600
- literały, 344, 670

- literówki, 553
 - Little Design Up Front, 155
 - live time, 284
 - locale, 82
 - localization, 82
 - logiczna struktura kodu, 773
 - lokalizacja, 82, 334
 - lokalizacja odwołań do zmiennych, 283
 - lokalizacja w pamięci, 359
 - lokalizowanie źródła błędu, 577
 - lokalna obsługa błędów, 232
 - lookup tables, 451
 - loose coupling, 135
 - LSP, 180
 - ludzkie traktowanie programistów, 715
 - luźna zależność, 135
 - luźne powiązania, 135
 - luźne wiązanie, 116
- Ł**
- łańcuchy instrukcji if-then-else, 396
 - łatwość wprowadzania zmian, 116
- M**
- M4, 757
 - magazyny dla programistów, 895
 - magiczne liczby, 328, 605
 - magiczne znaki, 333
 - mainframe, 101
 - make, 242, 755
 - makra, 218
 - nawiasy klamrowe, 219
 - nazwy makr, 219
 - ograniczenia w stosowaniu, 220
 - zarządzanie kodem wspomagającym debugowanie, 243
 - manipulowanie indeksem pętli, 415
 - McCabe Tom, 494
 - McCarthy Jim, 738
 - mechanizm blokowania, 377
 - mechanizm deklaracji niejawnych, 278
 - mechanizm indeksowania, 462
 - mechanizm informowania o wyjątkach, 237
 - mechanizm kontroli wersji, 242
 - mechanizm makr preprocesora, 756
 - mechanizm obsługi błędów, 230
 - mechanizm schodkowy, 464
 - member data, 311
 - metafory procesu programowania, 43
 - hodowla ostryg — przyrastanie systemu, 49
 - korzystanie, 46
 - łączenie metafor, 54
 - pisanie listów — pisanie kodu, 47
 - praca na roli — hodowanie systemu, 48
 - programista jako budowniczy — budowa oprogramowania, 50
 - przybornik programisty — narzędzia i moduły, 53
 - Metoda Fabrykująca, 140
 - metoda heurystyczna, 46
 - Metoda Szablonowa, 140
 - metoda wstępująca, 149
 - metody, 197
 - metody debugowania, 595
 - metody fabryczne, 489
 - metody integracji, 725
 - metody ograniczania zakresu, 286
 - metody oparte na tabelach, 449
 - decyzja o wprowadzeniu metod opartych na tabelach, 450
 - dni miesiąca, 451
 - elastyczne formaty komunikatu, 454
 - lista kontrolna, 467
 - preparowanie kluczy wyszukiwania, 461
 - stawki ubezpieczenia, 452
 - tabele o dostępie bezpośrednim, 451
 - tabele o dostępie indeksowym, 462
 - tabele o dostępie schodkowym, 464
 - metody optymalizacji kodu, 642, 645
 - metody pisania komentarzy, 828
 - metody podwyższania jakości, 502
 - koszt usuwania defektów, 508
 - koszt wyszukiwania defektów, 508
 - procent wykrytych defektów, 505
 - skuteczność metod, 505
 - współczynnik wykrywalności defektów, 506
 - metody programowania zespołowego, 514, 526
 - metody redukowania zagnieżdżeń, 490
 - metody szacowania ilości pracy, 706
 - metody usuwania defektów, 572
 - metody wprowadzania standardów, 696
 - metody wyszukiwania w tabelach, 467
 - metody zachęcania do budowy dobrego kodu, 696
 - metody zapewniania jakości, 514
 - metodyka pracy, 155, 691
 - metodyki wytwarzania oprogramowania, 91
 - Meyers Scott, 176
 - miary oprogramowania, 714
 - miary procesu wytwarzania oprogramowania, 713
 - miary zależności, 136
 - miejsce pracy, 718
 - mierzenie czasu aktywności zmiennej, 285

mierzenie złożoności, 494
 mikrooptymalizacje, 628
 Mills Harlan, 555
 minimalizacja liczby operacji dostępu do tabel, 662
 minimalizacja liczby wymiarów tablic, 661
 minimalizacja zakresu zmiennych, 288
 minimalny poziom złożoności, 116
 misja, 70
 mit stabilnych wymagań, 73
 mnożenie

- liczby całkowite, 660
- liczby zmiennoprzecinkowe, 660
- tablice, 424

 modelowanie

- obiekty abstrakcyjne, 188
- obiekty świata rzeczywistego, 188

 moderator, 520
 modularność, 144
 moduły, 53
 monitorowanie postępów w pracy, 729
 monitory pokrycia, 559
 Most, 140
 możliwość ponownego użycia, 116
 Myers Glenford, 204, 506

N

nadmiar konstrukcyjny, 84
 nadmiar wcięć, 482
 nadmierne rozproszenie informacji, 130
 namespace, 301
 narzędzia, 53

- narzędzia debugowania, 591
- narzędzia do projektowania, 122, 748
- narzędzia do refaktoryzacji, 753
- narzędzia do scalania, 750
- narzędzia formatujące kod źródłowy, 750
- narzędzia języka programowania, 882
- narzędzia kompilacji, 242, 754
- narzędzia porównujące kod, 558, 591
- narzędzia profilujące, 758
- narzędzia przekształcające strukturę kodu, 753
- narzędzia wspomagające opisywanie kodu, 843
- narzędzia wspomagające testowanie, 556
- narzędzia zakłócające pracę systemu, 560

 narzędzia do pracy z kodem wykonywalnym, 754

- biblioteki kodu, 755
- deasembler, 758
- debugowanie, 757
- generatory kodu, 756
- generowanie kodu, 754
- instalowanie, 756
- kompilatory, 754
- linkery, 754
- listingi asemblera, 758
- narzędzia kompilacji, 754
- narzędzia profilujące, 758
- optymalizacja kodu, 758
- preprocesor, 756
- testowanie, 757

 narzędzia do pracy z kodem źródłowym, 748

- analiza jakości kodu, 752
- Diff, 750
- edycja, 748
- formatowanie kodu źródłowego, 750
- generatory hierarchii klas, 751
- generatory odnośników, 751
- generatory opisu interfejsu, 751
- IDE, 748
- kontrola wersji, 753
- kontrolery semantyczne, 752
- kontrolery składniowe, 752
- narzędzia do refaktoryzacji, 753
- narzędzia do scalania, 750
- narzędzia przekształcające strukturę kodu, 753
- przekształcanie kodu źródłowego, 752
- słowniki danych, 754
- szablony, 751
- translatory kodu, 753
- wyszukiwanie w wielu plikach, 749
- zastępowanie w wielu plikach, 749
- zintegrowane środowiska programowania, 748

 narzędzia programowania, 747

- budowanie własnych narzędzi, 759
- lista kontrolna, 763
- narzędzia do pracy z kodem wykonywalnym, 754
- narzędzia przyszłości, 761
- narzędzia specyficzne dla projektu, 759
- skrypty, 760
- środowiska narzędzi programowania, 758

 naukowa metoda debugowania, 574
 nawiasy, 474, 475, 775
 nawiasy klamrowe, 480, 779
 nawyki, 869
 nazwy, 297, 308

- makra, 219
- procedury, 207, 257, 310, 386, 601
- stałe, 312

 nazwy zmiennych, 278, 297, 310, 605
 czytelność, 312
 definicje typów, 311
 długość nazwy, 300

- nazwy zmiennych
 - elementy typów wyliczeniowych, 312
 - homonimy, 323
 - indeksy pętli, 303
 - Java, 298
 - konwencje nazw, 308
 - konwencje niezależne od języka, 310
 - konwencje specyficzne dla języka, 313
 - krótkie nazwy, 319
 - kwalifikatory wyników obliczeń, 301
 - lista kontrolna, 325
 - nieformalne konwencje, 310
 - nieprawidłowe nazwy, 299
 - optymalna długość nazwy, 300
 - parametry wejściowe, 312
 - prawidłowe nazwy, 299
 - prefiksy semantyczne, 318
 - problemy, 322
 - programowanie w wielu językach, 314
 - przeciwieństwa, 302
 - rodzaje danych, 303
 - skrót, 319
 - skrót fonetyczny, 320
 - skrót typów użytkownika, 317
 - stałe nazwane, 308
 - standardowe prefiksy, 317
 - stosowanie standardowych prefiksów, 318
 - typy wyliczeniowe, 307
 - ukierunkowanie na problem, 299
 - wybór nazwy, 297
 - zakres, 300
 - zasady tworzenia skrótów, 319
 - zasady wyboru nazw, 298
 - zmiennie globalne, 311
 - zmiennie logiczne, 306
 - zmiennie składowe, 311
 - zmiennie stanu, 304
 - zmiennie tymczasowe, 305
 - znaczniki, 304
 - nieefektywne debugowanie, 573
 - nieformalne konwencje nazw, 310
 - nieformalne przeglądy techniczne, 503
 - niejawne deklarowanie zmiennych, 277
 - niejawne porównania zmiennych logicznych, 478
 - niekompletność instrukcji, 791
 - niepełne testowanie, 539
 - nieprawidłowe nazwy zmiennych, 299
 - niestandardowe cechy języka, 134
 - nieśmiertelnik, 362
 - nietypowe pętle przerywane, 409
 - nietypowe struktury sterowania, 427
 - niewłaściwe dane wejściowe, 224
 - niezawodność, 84, 339
 - niskie zwielokrotnienie wyjściowe, 116
 - niskopoziomowe metody pracy z dziedzina problemu, 883
 - niskopoziomowe pojęcia dziedziny problemu, 883
 - niskopoziomowe struktury implementacji, 882
 - niskopoziomowe typy danych, 166
 - normy IEEE, 849
 - normy programowania, 849
 - normy zapewniania jakości, 850
 - normy zarządzania, 850
 - NULL, 479
 - NUnit, 558
- O**
- objektowy język programowania, 99
 - obiekty, 122, 310
 - atrybuty, 123
 - definiowanie interfejsu, 124
 - identyfikacja, 123
 - obiekty abstrakcyjne, 188
 - obiekty odwołaniowe, 608
 - obiekty wartościowe, 608
 - objaśnienia do kodu, 822
 - Object Management Group, 154
 - obliczanie liczby ścieżek wykonania programu, 540
 - obliczanie wartości wielomianu, 666
 - Observer, 139, 140
 - Obserwator, 140
 - obsługa błędów, 82, 230, 258
 - lokalna obsługa błędów, 232
 - odporność, 233
 - podstawianie najbliższej dopuszczalnej wartości, 231
 - podstawianie następnego poprawnego elementu danych, 230
 - poprawność, 233
 - powtarzanie wcześniejszej odpowiedzi, 231
 - projekt wysokiego poziomu, 233
 - przerwanie pracy programu, 232
 - rejestrwanie ostrzeżenia w pliku, 231
 - schematy obsługi błędów, 230
 - wyświetlanie komunikatu błędu, 232
 - wywoływanie procedury obsługi błędu, 232
 - zwracanie kodu błędu, 231
 - zwracanie wartości neutralnej, 230
 - obsługa wyjątków, 83
 - obszary zmian, 133
 - niestandardowe cechy języka, 134
 - ograniczenia rozmiaru danych, 135
 - przewidywanie stopnia potencjalnych zmian, 135

- reguły biznesowe, 133
- składniki zależne od sprzętu, 133
- trudniejsze obszary projektu, 134
- wejście i wyjście, 134
- zmiennie stanu, 134
- ocena realizacji celów przez poszczególne zespoły, 505
- ocena statusu zmiennej, 282
- odmiany ewolucji oprogramowania, 598
- odporność, 233
- odwołania, 368
- oficjalna specyfikacja wymagań, 72
- ogłędziny, 527
 - zastosowanie, 527
- ogólna jakość architektury, 86, 89
- Ogólna Zasada Jakości Oprogramowania, 509, 555
- ograniczanie dostępności, 175
- ograniczanie wpływu zmian, 189
- ograniczanie zasięgu szkód powodowanych
 - przez błędy, 239
 - asercje, 240
- określanie
 - liczba dni w miesiącu, 451
 - problem, 70
 - rodzaj budowanego oprogramowania, 65
 - wymagania, 72
- operacje na blokach danych, 356
- operacje systemowe, 882
- operacje wejścia-wyjścia, 632
- operacje wskaźnikowe, 201
- opisywanie komentarzami wyszukanego kodu, 837
- opóźnianie obliczeń, 651
- oprogramowanie, 37
- oprogramowanie do kontroli wersji, 702
- Option Explicit, 278
- optymalizacja kodu, 621, 625, 645, 887
 - błędy, 635
 - buforowanie, 663
 - charakterystyka jakościowa, 622
 - eliminowanie wspólnych części wyrażeń, 673
 - indeks długości ciągu, 663
 - indeksy pomocnicze, 662
 - inicjalizowanie w czasie kompilacji, 667
 - iterowanie, 639
 - języki interpretowane, 634
 - koncentracja na optymalizacji w trakcie pisania programu, 629
 - koszt typowych operacji, 635
 - liczby wierszy kodu, 627
 - lista kontrolna, 642, 678
 - literały, 670
 - metody optymalizacji, 645
 - mikrooptymalizacje, 628
 - minimalizacja liczby operacji dostępu do tabel, 662
 - minimalizacja liczby wymiarów tablic, 661
 - moment optymalizacji, 630
 - narzędzia, 758
 - operacje wejścia-wyjścia, 632
 - opóźnianie obliczeń, 651
 - optymalizacje kompilatora, 631
 - pętle, 651
 - pomiary wydajności, 637, 638
 - porównywanie wydajności podobnych struktur logiki programu, 649
 - porządkowanie testów według częstości, 648
 - procedury, 674
 - procedury systemowe, 668
 - prostowanie pętli, 654
 - przekształcenia danych, 660
 - przenoszenie decyzji, 652
 - przenoszenie kodu do procedury wywołującej, 674
 - przerywanie testów po uzyskaniu wyniku, 646
 - przyczyny niskiej efektywności, 632
 - reimplementacja w języku niskiego poziomu, 675
 - rozmiar kodu maszynowego, 627
 - równoległa struktura indeksowa, 663
 - scalanie pętli, 653
 - specyfikacja wymagań, 623
 - stałe nazwane, 670
 - strategie, 640
 - stronicowanie, 633
 - struktury logiczne, 646
 - tożsamości algebraiczne, 665
 - umieszczanie najczęściej wykonywanej pętli wewnątrz, 658
 - wartości ograniczające, 657
 - wstępne obliczanie wartości, 670
 - wydajność kodu, 622
 - wrażenia, 665
 - wywołania systemowe, 634
 - zasada Pareto, 626
 - zastępowanie liczb zmiennoprzecinkowych całkowitymi, 660
 - zastępowanie złożonych wyrażeń wyszukiwaniem w tabeli, 650
 - zmniejszanie ilości pracy w pętli, 656
 - zmniejszanie obciążenia, 659
 - zmniejszanie obciążenia wyrażeń, 665
- optymalizacja wydajności, 621
- optymalizacje kompilatora, 631
- optymalna długość nazwy, 300
- optymalna liczba komentarzy, 828
- organizacja danych, 79
- organizacja klas, 807

organizacja kodu liniowego, 385
 asercje, 388
 grupowanie powiązanych instrukcji, 390
 kod obsługi błędów, 388
 kolejność czytania kodu, 389
 kolejność instrukcji, 385
 komentarze, 388
 lista kontrolna, 391
 nazwy procedur, 386
 niejasne zależności, 388
 parametry, 387
 sprawdzanie zależności, 388
 zależności, 385, 386
 Zasada Bliskości, 389
 organizacja programu, 78
 osadzone systemy wysokiego bezpieczeństwa, 65
 osobowość programisty, 856
 ostrzeżenia kompilatora, 329, 591
 OUT, 212
 overlay linker, 754
 Overridable, 180, 192
 Overrides, 192

P

pakiety, 117
 pakiety klas, 192
 pamięć podręczna, 664
 paradygmat wskaźników, 359
 parametry faktyczne, 216
 parametry nazwane, 216
 parametry procedur, 211, 224
 parametry wejściowe, 312
 Parnas David, 127
 part number, 462
 Pascal, 349
 peer reviews, 527
 pełne testowanie bazowe, 540
 performance, 622
 periodyki, 895
 Perl, 99, 750
 persistence, 289
 pętle, 292, 405
 break, 407, 409, 417, 418
 break z etykietą, 418
 budowanie pętli od wewnątrz, 422
 continue, 417, 418
 długość kodu, 422
 for, 409, 412
 foreach, 410
 For-Each, 410
 for-in, 410
 goto, 407
 indeks, 303, 414
 kod inicjalizujący, 411
 kod wewnątrz pętli, 413
 liczba zagnieżdżeń, 422
 licznik kontrolny, 416
 lista kontrolna, 425
 nazwy zmiennych, 420
 nietypowe pętle przerywane, 409
 operacje porządkowe, 414
 optymalizacja kodu, 651
 pętle nieskończone, 411
 pętle przerywane, 407
 prostowanie, 654
 przenoszenie decyzji, 652
 przesłuch indeksów, 420
 punkt wejścia, 411
 puste pętle, 413
 repeat, 414
 scalanie pętli, 653
 sprawdzanie wartości brzegowych, 419
 sterowanie pętlą, 410
 tablice, 424
 tworzenie, 422
 umieszczanie najczęściej wykonywanej pętli
 wewnątrz, 658
 wartości ograniczające, 657
 warunki zakończenia, 414
 wczesne wyjście z pętli, 417
 wejście do pętli, 411
 while, 406
 wybór rodzaju pętli, 405
 wyjście z pętli, 414
 zakres zmiennych sterujących, 421
 zmienne, 419
 zmniejszanie ilości pracy, 656
 zmniejszanie obciążenia, 659
 PHP, 99
 Pike Rob, 282
 pisanie kodu, 47
 kod procedury, 261
 pisanie listów — pisanie kodu, 47
 pisarz, 521
 plan czytelniczy programisty, 896
 plan kontroli jakości, 511
 plan zarządzania zasobami, 80
 planowanie, 52
 planowanie pod kątem rodzin programów, 190
 planowanie testów, 562
 platforma sprzętowa, 625

- platforma testowania, 592
- Plauger P.J., 143, 155
- pliki, 166
 - bas, 102
 - frm, 102
- pliki źródłowe C++, 808
- pluskwa, 569, 570
- podejście do testów programisty, 537
- podejście iteracyjne, 66, 69
- podejście sekwencyjne, 69
- podejście wstępujące, 148
- podejście zstępujące, 148
- podklasy, 201
- Podstawowa Zasada Formatowania, 770
- podstawowe klasy projektu, 79
- podstawowe praktyki programowania, 103
- podstawowe typy danych, 327
- podsystemy, 117, 120
- podwyższanie jakości, 502
- podział na klasy, 121
- podział na pakiety, 117
- podział na podsystemy, 117
- podział na procedury, 122
- pokaz fajerwerków, 529
- polimorfizm, 183
- Połya G., 145
- połączenia z bazą danych, 80
- pomiary przebiegu projektu, 712
- pomiary wydajności, 637
- pomiary wyników, 504
- pomiary złożoności, 495
- ponawianie testów, 558
- ponowne użycie kodu, 190
- popęnianie błędów, 572
- poprawianie kodu, 358
- poprawność, 233
- poprawność parametrów wejściowych, 281
- poprawność programu, 164
- porównania z zerem, 478
- porównywanie kodu źródłowego, 591
- porównywanie wydajności podobnych struktur
 - logiki programu, 649
- porządkowanie testów według częstości, 648
- powiązane instrukcje, 287
- powiązanie, 135
 - hybrydowe, 294
 - obiektywne, 137
 - parametryczne obiektywne, 137
 - parametryczne proste, 137
 - semantyczne, 138
- powielanie danych, 461
- powtarzanie testów, 562
- powtórzenia kodu, 599, 822
- poziomy precyzji projektu, 152
- poziomy abstrakcji, 882
- poziomy projektowania, 117
 - podział na klasy, 121
 - podział na podsystemy lub pakiety, 117
 - podział na procedury, 122
 - system, 117, 119
 - wewnętrzne projektowanie procedur, 122
- pożądaną poziom szczegółowości, 152
- PPP, 251
- praca na roli — hodowanie systemu, 48
- praca przyrostowa, 744
- praktyki programowania, 65, 103
- prawa de Morgana, 473
- prawo Freda Brooksa, 710
- prawo Greshama, 153
- prefiksy nazw, 317
- prefiksy semantyczne, 318
- preparowanie kluczy wyszukiwania, 461
- preprocesor, 242, 243, 756
 - zarządzanie kodem wspomagającym debugowanie, 243
- private, 177
- problem „złośliwy”, 110
- problemy akcydentalne, 113
- problemy istotne, 113
- problemy projektowania, 110
- problemy sterowania, 469
- problemy z samodzielnym testowaniem, 538
- procedura inspekcji, 521
- procedure, 217
- procedury, 122, 197
 - abstrakcje pośrednie, 200
 - budowanie, 251, 253
 - budowanie metodą PPP, 256
 - czytanie programów, 199
 - deklaracja, 262
 - długość kodu, 209
 - efektywność, 258
 - exit, 427
 - formatowanie kodu, 802
 - IN, 212
 - klauzule kontrolne, 428
 - kohezja, 204
 - kolejność parametrów, 211
 - komentarze, 841
 - kompilacja, 267
 - liczba parametrów, 214

- procedury
 - lista kontrolna, 221
 - listy parametrów, 358
 - małe procedury, 202
 - nazwy procedur, 207, 257
 - optymalizacja kodu, 674
 - OUT, 212
 - parametry, 211
 - parametry faktyczne, 216
 - parametry nazwane, 216
 - pisanie kodu, 261
 - projektowanie, 204, 256
 - przekazywanie parametrów, 221
 - przesłanki utworzenia procedury, 200, 203
 - pseudokod, 260
 - redukcja złożoności, 200
 - refaktoryzacja, 607
 - rekurencja, 429
 - return, 427
 - sprawdzanie kodu, 266
 - stosowanie, 217
 - strażnik, 428
 - ścieżka wykonania, 266
 - testowanie, 258
 - ukrywanie operacji wskaźnikowych, 201
 - ukrywanie sekwencji, 201
 - wcięcia kodu, 428
 - wiele wyjść, 427
 - założenia dotyczące parametrów, 214
 - zapobieganie powtórzeniom kodu, 200
 - zmiennne sygnalizacji stanu i błędów, 213
 - zwiększanie przenośności kodu, 201
 - zwiększanie wydajności, 201
- procedury dostępne, 375, 378
 - stosowanie, 376
 - zalety, 376
- procedury inline, 218, 220
 - stosowanie, 220
- procedury integracji, 726
- procedury kontroli zmian, 503, 700
- procedury nieprzesłaniające, 181
- procedury niskiej jakości, 198
- procedury programowania funkcjonalnego, 197
- procedury przesłaniające, 181
- procedury systemowe, 668
- procedury wysokiej jakości, 197
- procedury zastępcze, 244
- proces, 875
- proces budowy oprogramowania, 40, 503
 - pomiary wyników, 504
 - procedury kontroli zmian, 503
 - prototypowanie, 504
- proces iteracyjny, 147
- proces programowania dla zespołów, 555
- Proces Programowania w Pseudokodzie, 251, 613, 827
 - alternatywy dla pseudokodu, 269
 - budowanie klas, 251, 252
 - budowanie procedur, 251, 253, 256
 - lista kontrolna, 271
 - pisanie kodu procedury, 261
 - projektowanie procedur, 256
 - pseudokod, 253
 - sprawdzanie kodu procedury, 266
 - usuwanie pozostałości, 268
- proces wytwarzania oprogramowania, 40
- produkty, 689
- produkty systemowe, 689
- produktywność programistów, 40
- profilowanie, 201
- program, 689
- programista jako budowniczy
 - budowa oprogramowania, 50
- programiści, 715, 720
 - cechy charakteru, 855
 - miejsce pracy, 718
 - rozwój zawodowy, 861
 - różnice między programistami, 716
 - różnice między zespołami, 716
 - różnice wydajności i jakości pracy, 715
 - wojny religijne, 717
 - zajęcia programistów, 716
- programowanie, 40, 892
 - programowanie do języka, 879
 - programowanie obiektowe, 122
 - programowanie ofensywne, 241
 - programowanie sterylne, 555
 - programowanie w kategoriach dziedziny
 - problemu, 881
 - programowanie w pseudokodzie, 251
 - programowanie z wyprzedzeniem, 182
- programowanie defensywne, 223
 - asercje, 225
 - ilość kodu defensywnego w wersji finalnej, 245
 - kod wspomagający debugowanie, 241
 - lista kontrolna, 246
 - mechanizm obsługi błędów, 230
 - ograniczanie zasięgu szkód powodowanych
 - przez błędy, 239
 - podejście defensywne, 246
 - wyjątki, 234, 247
 - zabezpieczanie programu przed niewłaściwymi
 - danymi wejściowymi, 224
 - zabezpieczenia, 247

- programowanie strukturalne, 127, 490
 - iterowanie, 492
 - komponenty, 491
 - sekwencja, 491
 - wybór, 492
- programowanie w parach, 514, 517, 531
 - konwencje pisania kodu, 517
 - lista kontrolna, 519
 - rotacja w parach, 518
 - rotacja zadań, 518
 - tempo pracy, 518
 - zalety, 518
 - zasady, 517
- programowanie zespołowe, 513, 514
 - cele, 514
 - czytanie kodu, 528
 - dzielenie się wiedzą, 516
 - formalne inspekcje, 519
 - konwencje, 516
 - kultura organizacji, 516
 - metody, 526
 - metody zapewniania jakości, 514
 - normy, 532
 - ogłędziny, 527
 - pokaz fajerwerków, 529
 - porównanie metod, 530
 - programowanie w parach, 517
 - standardy, 516
 - wspólnota własności, 516
- programy profilujące, 592
- projekt, 65
- projekt informatyczny, 41
- projekt interfejsu użytkownika, 80
- projekt iteracyjny, 67
- projekt programu, 623
- projekt wysokiego poziomu, 233
- projektowanie, 109, 157
 - cechy projektu, 115
 - charakter niedeterministyczny, 112
 - dokumentowanie pracy, 153
 - dziel i rządź, 147, 148
 - heurystyki, 144
 - kompromisy, 112
 - lista kontrolna, 158
 - metodyki pracy, 155
 - normy, 158
 - ograniczenia, 112
 - podjęcie wstępujące, 148
 - podjęcie zstępujące, 148
 - podsumowania, 154
 - poziom precyzji projektu, 152
 - poziomy, 117
 - priorytety, 112
 - problem „złośliwy”, 110
 - problemy, 110
 - proces heurystyczny, 112
 - proces iteracyjny, 147
 - proces niedoskonały, 111
 - prototypowanie eksperymentalne, 150
 - refleksja, 147
 - techniki, 146
 - zakończenie pracy nad projektem, 152
 - zarządzanie złożonością, 113, 114
- projektowanie kontraktowe, 228, 269
- projektowanie na poziomie procedur, 204
- projektowanie oprogramowania, 155, 886
- projektowanie pod kątem testów, 142
- projektowanie procedur, 256
- projektowanie zespołowe, 151
- prostowanie pętli, 654
- prototype, 151
- prototypowanie, 504
- prototypowanie eksperymentalne, 150
- prywatne dziedziczenie, 179
- prywatny konstruktor, 187
- przechowywanie w pamięci podręcznej, 651
- przechwytywanie wyjątków, 234
- przeciwieństwa w nazwach zmiennych, 302
- przeglądy norm, 850
- przeglądy techniczne, 503
- przeglądy w gronie współpracowników, 527
- przejrzystość związków danych, 355
- przekazywanie parametrów, 221, 369, 370
- przekształcanie klucza, 461
- przekształcanie kodu źródłowego, 613, 752
- przekształcanie pseudokodu w komentarze
 - wysokiego poziomu, 262
- przekształcenia danych, 660
- przełożony, 721
- przenoszenie decyzji, 652
- przenoszenie kodu do procedury wywołującej, 674
- przenośność kodu, 116, 201
- przepelnienie
 - przepelnienie w wynikach pośrednich, 330
- przepływ danych, 543
- przerywanie testów po uzyskaniu wyniku, 646
- przesłuch indeksów, 420
- przestrzenie nazw, 301
- przetwarzanie błędów, 82, 437
- przetwarzanie tablic, 424
- przewidywanie błędów, 547
- przewidywanie zmian, 133, 135

- przybornik programisty — narzędzia i moduły, 53
 - przyczyny niskiej efektywności, 632
 - przygotowania, 57
 - analogie, 62
 - architektura oprogramowania, 77
 - decyzje o wykorzystaniu istniejącej bazy, 85
 - definicja problemu, 70, 71
 - ilość czasu poświęcanego na przygotowania, 89
 - kierownictwo, 60
 - koszt usuwania defektów, 63
 - lista kontrolna, 93
 - nadmiar konstrukcyjny, 84
 - oficjalna specyfikacja wymagań, 72
 - określanie rodzaju budowanego oprogramowania, 65
 - określanie wymagań, 72
 - podejście iteracyjne, 66, 69
 - podejście sekwencyjne, 69
 - projekt iteracyjny, 67
 - przyczyny braków w przygotowaniu, 59
 - rodzaje projektów, 65
 - strategia zmian, 85
 - test gotowości przełożonego, 64
 - uzasadnianie konieczności wykonania czynności wstępnych, 61
 - WIMP, 60
 - WISCA, 60
 - znaczenie, 58
 - przyrastanie systemu, 49
 - przyrostowe strategie integracji, 730
 - pseudokod, 251, 253, 259, 260
 - język naturalny, 253
 - przekształcanie w komentarze wysokiego poziomu, 262
 - publiczne składowe danych klasy, 601
 - publikacje branżowe, 896
 - publikacje specjalistyczne, 895
 - publikowanie kompilacji rano, 741
 - punkty kontroli, 143
 - puste pętle, 413
 - puste wiersze, 775
 - Python, 99
- Q**
- quicksort, 624
- R**
- raport z inspekcji, 523
 - recenzent, 521
 - redukowanie głębokości zagnieżdżeń, 482
 - redukowanie złożoności, 494, 873
 - miar złożoności, 495
 - technika Toma McCabe'a, 494
 - refaktoryzacja, 191, 269, 597, 599
 - bezpieczne przekształcanie kodu, 613
 - interfejs klasy, 600
 - kohezja, 600
 - komentarze, 602
 - lista kontrolna, 604, 611
 - lista niezbędnych kroków procedury, 613
 - lista oczekujących, 613
 - lista parametrów, 600
 - moduły o dużej złożoności, 616
 - narzędzia, 753
 - nazwy procedur, 601
 - obiekt pośredniczący, 601
 - obiekty odwołaniowe, 608
 - obiekty wartościowe, 608
 - początkowa wersja kodu, 613
 - powtórzenia kodu, 599
 - przeciążenie typu prostego, 601
 - przeglądy zmian, 614
 - przesłanki, 599
 - przy dodawaniu klasy, 616
 - przy dodawaniu procedury, 616
 - przy usuwaniu defektów, 616
 - publiczne składowe danych klasy, 601
 - refaktoryzacje implementacji klasy, 612
 - refaktoryzacje interfejsu klasy, 609, 612
 - refaktoryzacje na poziomie danych, 605, 611
 - refaktoryzacje na poziomie instrukcji, 606, 611
 - refaktoryzacje na poziomie procedury, 607, 612
 - refaktoryzacje na poziomie systemu, 610
 - równoległe modyfikacje w drzewach dziedziczenia, 600
 - równoległe modyfikacje w różnych klasach, 600
 - równoległe zmiany w instrukcjach case, 600
 - strategie refaktoryzacji, 615
 - testowanie, 614
 - ukrywanie informacji, 601
 - wersje pośrednie, 614
 - wybrane refaktoryzacje, 605
 - wyrażenia logiczne, 606
 - za długą procedurą, 599
 - za dużo zagnieżdżeń, 600
 - zastęp magiczną liczbę stałą nazwaną, 615
 - zbyt długa pętla, 600
 - zmiany w klasie, 600
 - zmiennie globalne, 602
 - referencje, 368
 - refleksja, 147

reguły biznesowe, 80, 120, 133
 reguły komunikacji, 120
 reimplementacja w języku niskiego poziomu, 675
 reinicjalizacja zmiennych, 281
 rejestrowanie danych, 560
 rekurencja, 429, 446

- licznik kontrolny, 432
- stos, 433
- stosowanie, 432
- warunek zakończenia, 432

 relacja „jest”, 180
 relacja „ma”, 179
 relacje z klientem, 730
 religia, 887
 repeat, 292, 414
 return, 427, 445, 446, 607
 Riel Arthur, 183
 Rittel Horst, 110
 robustness, 233
 rodzaje danych, 303
 rodzaje pętli, 406
 rodzaje projektów, 65
 rola testów programisty, 534
 routine, 197, 217
 rozmiar programu, 683
 rozmiar projektu, 684

- efektywność pracy, 687
- liczba błędów, 685
- stosowana metodyka pracy, 691
- udział poszczególnych czynności, 688
- wielkość zespołu, 689
- wydajność pracy, 687

 rozpiętość zmiennej, 283
 rozpoznawanie alternatyw konstrukcyjnych, 123
 rozproszenie informacji, 130
 rozszerzalność, 116
 rozwiązywanie problemów, 859
 rozwój zawodowy, 861
 równoległa struktura indeksowa, 663
 RTFM, 860
 rusztowanie, 53, 268, 556, 565, 592

S

SAFE_DELETE, 368
 SAFE_NEW, 367
 sandwich integration, 734
 scaffolding, 53, 268
 scalanie pętli, 653
 scentralizowany mechanizm informowania

- o wyjątkach, 237

schemat formatowania, 782
 SCM, 699, 700
 SDF, 813
 sed, 750
 sekwencja, 491
 selektywne zakłócanie pracy pamięci, 561
 semantyczne łamanie hermetyzacji, 177
 sentinel value, 657
 Set(), 610
 sformalizowane metody pracy, 691
 Shneiderman Ben, 772
 silnia, 433
 singleton, 187
 Singleton, 139, 140
 sizeof(), 371
 skala rozmiarów projektów, 684
 skalowalność, 81
 składniki zależne od sprzętu, 133
 składniki zależne od systemu, 121
 skromność, 857
 skróty fonetyczne, 320
 skróty typów użytkownika, 317
 skrypty, 760
 słowniki danych, 754
 Smalltalk, 96
 smart pointers, 370
 Social Security Number, 450
 software configuration management, 699
 software-development folder, 813
 spaghetti, 435, 753
 span, 283
 specyfikacja wymagań, 73
 spoiistość, 141
 sposoby przeliczania wyrażeń logicznych, 475
 spójność, 141
 sprawdzanie dostępu do pamięci, 561
 sprawdzanie kodu procedury, 266
 sprawdzanie wartości brzegowych, 419
 SQL, 100
 SSN, 450
 stabilne wymagania, 73
 stałe, 312
 stałe nazwane, 308, 343, 352, 670

- deklaracja danych, 343
- emulacja, 374
- stosowanie, 345

 standardowe metody, 117
 standardowe prefiksy nazw, 317
 standardy pisania kodu, 696
 stany danych, 543
 State, 177

static, 289
 stawki ubezpieczenia, 452
 sterowanie pętlą, 410
 Stevens Wayne, 204
 stopniowe przyrastanie, 49
 stos, 433
 stosowana metodyka pracy, 691
 stosowanie procedur systemowych, 668
 stowarzyszenia zawodowe, 898
 Strategia, 140
 strategię optymalizacji kodu, 621, 640
 strategię refaktoryzacji, 615
 strategię testowania, 502
 strategię zmian, 85
 Strategy, 139, 140
 stratyfikacja, 117
 strażnik, 657
 string, 335
 stronicowanie, 633
 Stroustrup Bjarne, 239
 struktury, 355

- listy parametrów, 358
- operacje na blokach danych, 356
- poprawianie kodu, 358
- przejrzystość związków danych, 355

 struktury logiczne, 646
 struktury sterujące, 493, 782

- komentarze, 840
- redukowanie złożoności, 494
- typy danych, 292
- złożoność, 493, 494

 styl programowania, 821
 styl programowania jako dokumentacja, 814
 style formatowania, 776

- czyste bloki, 776
- emulowane czyste bloki, 777
- wyrównywanie z głębokimi wcięciami, 780
- wyróżnianie granic bloków parami begin-end, 779

 Sub, 197
 success, 306
 sumowanie elementów macierzy, 637
 Sun Microsystems, 99
 switch, 398
 symulacja typów wyliczeniowych, 342
 syndrom WISCA, 60
 system, 119
 system kontroli wersji, 702
 system kopii zapasowych, 703
 system rozliczania czasu pracy, 124
 systemy, 689
 systemy biznesowe, 65

systemy pracy ciągłej, 65
 sytuacje awaryjne, 83
 szablony, 751
 szacowanie czasu, 863
 szacowanie ilości pracy, 706
 szacowanie ilości pracy przy programowaniu, 707
 szacowanie wielkości projektu, 711
 szczegóły implementacji, 126
 szczupłość, 116
 szukanie błędów, 584

Ś

ścieżka wykonania procedury, 266
 ścieżki komunikacji, 684
 śmieci na wyjściu, 224
 średnie wielokrotnienie wyjściowe, 116
 środowiska narzędzi programowania, 758

T

tabele o dostępie bezpośrednim, 451
 tabele o dostępie indeksowym, 462
 tabele o dostępie schodkowym, 464
 tabele referencyjne, 451
 tablice, 345, 352

- ARRAY_LENGTH(), 346
- C, 346
- granice, 345
- indeksy, 345
- interferencja indeksów, 346
- kolejność indeksów, 346
- pętle, 424
- przetwarzanie, 424

 tablice decyzyjne, 472
 Team Software Process, 555
 technika rozwiązywania problemów w matematyce, 145
 technika Toma McCabe'a, 494
 techniki formatowania kodu, 774
 techniki projektowania, 146
 techniki testowania, 539
 technologia, 101
 teczka pracy, 813

- teczka pracy jednostki, 813
- teczka pracy programu, 813

 template, 220
 Template Method, 139, 140
 teoria projektowania oprogramowania, 156
 test dymowy, 738, 739, 743
 test first, 537, 566
 test last, 537

- testowanie, 41, 142, 268, 269, 533, 534, 564, 614
 - automatyzacja testów, 562
 - bazy błędów, 561
 - błędy programisty, 553
 - błędy w procesie testowania, 555
 - brudny test, 538
 - cele testów, 535
 - czysty test, 538
 - debugger symboliczny, 560
 - generatory danych testowych, 558
 - gromadzenie informacji o testach, 563
 - kategorie błędów, 551
 - krokowe wykonywanie kodu, 560
 - liczba błędów, 554
 - lista kontrolna, 566
 - minimalna liczba testów, 540
 - monitory pokrycia, 559
 - narzędzia, 757
 - narzędzia do porównywania danych, 558
 - narzędzia wspomagające testowanie, 556
 - narzędzia zakłócające pracę systemu, 560
 - niepełne testowanie, 539
 - normy, 566
 - pełne testowanie bazowe, 540
 - planowanie testów, 562
 - podejście do testów programisty, 537
 - ponawianie testów, 558
 - powtarzanie testów, 562
 - problemy z samodzielnym testowaniem, 538
 - proces programowania, 536
 - rejestrwanie danych, 560
 - rusztowanie, 556
 - techniki, 539
 - testowanie białej skrzynki, 534
 - testowanie czarnej skrzynki, 534
 - testowanie integracyjne, 533
 - testowanie jednostek systemu, 730
 - testowanie jednostkowe, 533
 - testowanie klasy, 252
 - testowanie komponentów, 533
 - testowanie procedury, 258
 - testowanie regresyjne, 533, 558, 562
 - testowanie systemowe, 534
 - typowe błędy, 550
 - usprawnianie testów, 561
 - wyzwanie, 535
- testowanie przepływu danych, 543
 - analiza wartości granicznych, 547
 - dzielenie danych według równoważności, 546
 - Java, 545
 - klasy dobrych danych, 549
 - klasy złych danych, 548
 - kombinacje stanów, 543
 - przewidywanie błędów, 547
 - przy wejściu-używana, 544
 - przy wejściu-zwolniona, 544
 - stany danych, 543
 - ułatwianie ręcznego sprawdzania wyników, 550
 - używana-zdefiniowana, 544
 - zdefiniowana-przy wyjściu, 543
 - zdefiniowana-zdefiniowana, 543
 - zdefiniowana-zwolniona, 544
 - złożone wartości graniczne, 548
 - zwolniona-używana, 544
 - zwolniona-zwolniona, 544
- testy automatyczne, 562
- testy czyste, 538
- testy logiczne, 201, 337
- testy programisty, 537, 538
- Thomas Dave, 599
- throw, 234
- tożsamości algebraiczne, 665
- tracę monitor, 757
- translatory kodu, 753
- trudności akcydentalne, 113
- trudności istotne, 113
- true, 469, 470
- trwałość, 289
- try-catch, 234, 235
- try-catch-finally, 235
- try-finally, 440
- TSP, 555
- tworzenie
 - klasy, 252
 - mechanizm asercji, 227
 - oprogramowanie komputerowe, 37
 - pętle, 422
 - skrót, 320
- type, 350
- typedef, 220, 350
- typy danych, 259, 277, 292, 327, 355, 641
 - aliasy, 346
 - ciągi znakowe, 333
 - enumeracja, 338
 - liczby, 327
 - liczby całkowite, 329
 - liczby zmiennoprzecinkowe, 331
 - lista kontrolna, 351
 - stałe nazwane, 343
 - struktury, 355
 - tablice, 345
 - tworzenie własnych typów, 346

- wskaźniki, 359
- wyliczenia, 338
- zmiennie logiczne, 336
- znaki, 333
- typy predefiniowane, 350
- typy wyliczeniowe, 307, 338, 352
 - czytelny kod, 339
 - emulacja, 374
 - Java, 342
 - niezawodność, 339
 - pierwsza wartość wyliczenia, 341
 - symulacja typów wyliczeniowych, 342
 - wartości niedozwolone, 340
 - wprowadzanie zmian, 340

U

- uchwyty, 80
- uczciwość intelektualna, 862
- UDF, 813
- UDT, 317
- układ implementacji klasy, 804
- układ interfejsu klasy, 804
- układ kodu w plikach, 807
- układ kodu źródłowego, 767
 - cele formatowania, 773
 - ekstrema formatowania, 768
 - formatowanie instrukcji, 789
 - formatowanie klas, 804
 - formatowanie komentarzy, 800
 - formatowanie procedur, 802
 - formatowanie struktur sterujących, 782
 - interpretacja programu, 770
 - lista kontrolna, 809
 - Podstawowa Zasada Formatowania, 770
 - style formatowania, 776
 - techniki formatowania, 774
 - wartość dobrego układu, 771
- ukrywanie danych globalnych, 189
- ukrywanie informacji, 127, 128, 132, 163, 601
 - dane globalne, 131
 - groźba obniżenia wydajności, 131
 - nadmierne rozproszenie informacji, 130
 - problemy, 130
 - rodzaje tajemnic, 130
 - zależności cykliczne, 131
- ukrywanie operacji wskaźnikowych, 201
- ukrywanie sekwencji, 201
- ukrywanie szczegółów implementacji klasy, 176, 189
- ukrywanie złożoności, 128
- UML, 146, 154

- Unicode, 334
- unikanie głębokich zagnieżdżeń, 482
- unikanie niepowodzeń, 143
- unit-development folder, 813
- UNIX, 98, 758
- unrolling, 654
- upraszczanie
 - lista parametrów, 358
 - operacje na blokach danych, 356
 - złożone wyrażenia logiczne, 471
- uprawianie oprogramowania, 49
- user-defined type, 317
- usprawnianie
 - komunikacja, 684
 - testy, 561
- ustawienia regionalne i językowe, 82
- usuwanie błędów, 268
- usuwanie defektu, 585, 594
 - diagnoza defektu, 585
 - oryginalna wersja kodu, 586
 - technika wprowadzania w kodzie przypadkowych zmian, 587
 - testy jednostkowe, 588
 - wprowadzanie zmian, 588
- usuwanie kodu wspomagającego, 242

V

- virtual, 180, 192
- Visual Basic, 96, 100, 195
 - inicjalizowanie zmiennych, 279
 - konwencje nazw, 314, 316
 - mechanizm deklaracji niejawnych, 278
 - typy wyliczeniowe, 338
 - wyjątki, 235, 237

W

- walka ze złożonością, 873
- walk-through, 527, 890
- wartości graniczne, 547, 548, 657
- warunki końcowe, 228
- warunki wstępne, 228
- wątki, 80
- wcięcia, 775, 800
 - kod procedury, 428
- wczesne wyjście z pętli, 417
- Webber Melvin, 110
- Weinberg Gerald, 629
- wejście do pętli, 411, 425
- wejście-wyjście, 82
- wersja robocza oprogramowania, 241

- wersje narzędzi, 703
- wewnętrzne projektowanie procedur, 122
- while, 292, 406
 - pętla z testem na końcu, 407
 - pętla z testem na początku, 406
 - pętle nieskończone, 411
- wicked, 110
- widoczność, 136
- Wieggers Karl, 559
- wiele wyjść z procedury, 427
- wielkość projektu, 684
- wielkość zespołu, 689
- wielodziedziczenie, 184
- WIMP, 60
- Windows, 101
- WISCA, 60
- wiszące wskaźniki, 561
- wizja produktu, 70
- wojny religijne, 717
- wpływ podejścia iteracyjnego na przygotowania, 66
- wpływ różnych czynników na harmonogram pracy, 708
- wpływ wielkości projektu na efektywność pracy, 687
- wpływ wielkości projektu na liczbę błędów, 685
- wpływ wielkości projektu na wykonywaną pracę, 687
- wprowadzanie zmian, 340
 - zmiany w zawartości wielu plików, 750
- wskaźniki, 201, 282, 359
 - auto_ptr, 369
 - błędy wskaźników, 366
 - C, 370
 - C++, 368
 - deklaracja, 361
 - inicjalizacja, 361
 - konwersja typów, 370
 - lista kontrolna, 381
 - lokalizacja w pamięci, 359
 - monitorowanie alokacji, 367
 - nadmiarowość, 363
 - nieśmiertelnik, 362
 - NULL, 479
 - sposób interpretacji zawartości lokalizacji w pamięci, 359
 - stosowanie, 361
 - użycie zwolnionych wskaźników, 366
 - wskaźniki na NULL, 366
 - wstawianie węzła, 364
 - wykrywanie uszkodzeń pamięci, 362
 - wrażenia wskaźnikowe, 365
 - zasada gwiazdki, 370
 - zerowanie wskaźnika po usunięciu, 366
- wskaźniki inteligentne, 370
- wspólna klauzula else, 442
- wspólnota własności, 516
- współczynnik wykrywalności defektów, 506
- współdziałanie, 81
- współpraca, 865
- współpraca z przełożonymi, 721
- wstawianie węzła, 364
- wstępne obliczanie wartości, 670
- wstrząsanie pamięcią, 561
- WWW, 101
- wybór, 492
- wybór języka programowania, 95
- wybór nazwy zmiennej, 297
- wybór podstawowych praktyk programowania, 103
- wybór procesu, 875
- wybór rodzaju pętli, 405
- wybrane refaktoryzacje, 605
- wydajność, 81, 164, 201, 622
- wydajność kodu, 622
 - charakterystyka jakościowa, 622
 - interakcje z systemem operacyjnym, 624
 - kompilacja kodu, 624
 - optymalizacja kodu, 622, 625
 - platforma sprzętowa, 625
 - projekt klas, 624
 - projekt procedur, 624
 - projekt programu, 623
 - specyfikacja wymagań, 623
- wydajność pracy, 687
- wydajność programu, 642
- wydawnictwa popularne, 896
- wyjątki, 83, 234, 247, 248
 - biblioteki, 237
 - EOFException, 236
 - Exception, 238
 - hermetyzacja, 236
 - ignorowanie wyjątku, 237
 - języki programowania, 235
 - przechwytywanie, 234
 - puste bloki catch, 237
 - scentralizowany mechanizm informowania o wyjątkach, 237
 - standardy pracy z wyjątkami w projekcie, 238
 - try-catch, 234, 235
 - try-catch-finally, 235
 - wyrzucanie, 234
 - zgłaszanie, 234, 236
- wyjście z pętli, 414, 426
- wykonalność, 84
- wykrywanie błędów, 83
- wykrywanie nieudanych kompilacji, 739

- wykrywanie uszkodzeń pamięci, 362
 - wyliczenia, 338
 - wymagania, 72, 90
 - cel biznesowy projektu, 75
 - jakość wymagań, 76
 - kompletność wymagań, 77
 - koszty zmian w specyfikacji, 74
 - lista kontrolna wymagań, 75
 - mit stabilnych wymagań, 73
 - oficjalna specyfikacja, 72
 - procedura kontroli zmian, 74
 - wymagania funkcjonalne, 75
 - wymagania jakościowe, 76
 - zmiana wymagań w trakcie budowy oprogramowania, 74
 - wypełnianie pamięci, 561
 - wrażenia logiczne, 469, 471
 - C, 479
 - C++, 479
 - false, 469
 - Java, 480
 - jednoznaczność wyrażeń, 474
 - kierunek osi liczbowej, 477
 - makra preprocesora, 479
 - nawiasy, 474
 - porównania z zerem, 478
 - pozytywne formułowanie wyrażeń, 473
 - prawa de Morgana, 473
 - problemy, 479
 - sposoby przeliczania wyrażeń logicznych, 475
 - tablice decyzyjne, 472
 - true, 469
 - upraszczanie złożonych wyrażeń, 471
 - wrażenia wskaźnikowe, 365
 - wyrównywanie z głębokimi wcięciami, 780
 - wyróżnianie granic bloków parami begin-end, 779
 - wysokie zwielokrotnienie wejściowe, 116
 - wysokopoziomowe pojęcia dziedziny problem, 883
 - wystąpienie defektu, 571
 - wyszukiwanie
 - binarne, 466
 - sekwencyjne, 466
 - w tabelach, 467
 - w wielu plikach, 749
 - wyszukiwanie defektu, 574, 594
 - porady, 578
 - wyświetlanie komunikatu błędu, 232
 - wytrwałość, 867
 - wytwarzanie oprogramowania, 40
 - wywołania funkcji logicznych, 397
 - wywołania systemowe, 634
 - wywoływanie procedury obsługi błędu, 232
 - wyznaczanie ograniczeń, 112
 - wzorce projektowe, 139, 157
 - Abstract Factory, 140
 - Adapter, 140
 - Bridge, 140
 - Composite, 140
 - Decorator, 140
 - Dekorator, 140
 - Fabryka Abstrakcyjna, 140
 - Facade, 140
 - Factory Method, 139, 140
 - Fasada, 140
 - Iterator, 140
 - Kompozyt, 140
 - komunikacja, 140
 - Metoda Fabrykująca, 140
 - Metoda Szablonowa, 140
- ## Z
- zabezpieczanie programu przed niewłaściwymi danymi wejściowymi, 224
 - zabezpieczanie wartości globalnych, 374
 - zabezpieczenia, 81, 248
 - zabezpieczenia przed awariami, 83
 - zachęcanie do budowy dobrego kodu, 696
 - zagnieżdżanie pętli, 422
 - zajęcia programistów, 716
 - zakończenie pracy nad projektem, 152
 - zakres, 282
 - nazwy zmiennych, 300
 - zakresy odpowiedzialności, 142
 - zależność, 135
 - elastyczność, 136
 - liczba powiązań, 136
 - miary, 136
 - powiązania obiektowe, 137
 - powiązania parametryczne obiektowe, 137
 - powiązania parametryczne proste, 137
 - powiązania semantyczne, 138
 - rodzaje powiązań, 137
 - widoczność, 136
 - zależności cykliczne, 131
 - zapobieganie powtórzeniom kodu, 200
 - zarządzanie kodem wspomagającym debugowanie, 243
 - zarządzanie konfiguracją oprogramowania, 699
 - zarządzanie pamięcią, 80
 - zarządzanie procesem wytwarzania oprogramowania, 695
 - zarządzanie relacyjnymi bazami danych, 100

- zarządzanie w programowaniu, 695, 721
 - budowanie harmonogram, 705
 - ludzkie traktowanie programistów, 715
 - normy, 722
 - pomiary przebiegu projektu, 712
 - współpraca z przełożonymi, 721
 - zachęcanie do budowy dobrego kodu, 696
 - zarządzanie konfiguracją, 698
- zarządzanie zasobami, 80
- zarządzanie złożonością, 113, 114
- zarządzanie zmianami, 699
- Zasada Bliskości, 389
- zasada caveat emptor, 224
- zasada DRY, 599
- zasada gwiazdki, 370
- zasada jednego właściwego miejsca, 143
- zasada Pareto, 626
- zasada podstawienia Liskov, 180
- zasada świadomego wyboru procesu, 876
- zasady budowy oprogramowania, 503
- zasady wyboru nazw zmiennych, 298
- zasoby, 80
- zastępowanie liczb zmiennoprzecinkowych całkowitymi, 660
- zastępowanie w wielu plikach, 749
- zastępowanie złożonych wyrażeń wyszukiwaniem w tabeli, 650
- zawieranie, 179
- zbiory norm, 850
- zbiór konwencji nazw, 315
- zduplikowany kod, 599
- zerowanie wskaźnika po usunięciu, 366
- zestaw znaków, 82
- zewnętrzna dokumentacja programu, 813
 - dokument opisujący projekt szczegółowy, 814
 - teczki pracy, 813
- zgłaszanie wyjątków, 234, 236
- zintegrowane środowiska programowania, 748
- złe wskaźniki, 367
- złośliwy problem, 110
- złożone testy logiczne, 201
- złożone wartości graniczne, 548
- złożoność, 115, 188, 494, 873
 - pomiar złożoności, 495
 - redukowanie złożoności, 494
 - technika Toma McCabe'a, 494
 - złożoność przepływu sterowania, 494
- zły kod, 884
- zmiany, 700
 - zmiany w kodzie programu, 702
 - zmiany w projekcie, 700
 - zmiany w wymaganiach, 700
 - zmiany w wymaganiach w trakcie budowy oprogramowania, 74
- zmiennie, 275
 - akumulatory, 281
 - automatyczne inicjalizowanie wszystkich zmiennych, 281
 - czas aktywności, 284
 - czas wiązania, 290
 - deklaracja, 277
 - inicjalizacja, 278
 - jedno przeznaczenie zmiennej, 293
 - liczniki, 281
 - lista kontrolna, 295
 - lokalizacja odwołań, 283
 - metody ograniczania zakresu, 286
 - mierzenie czasu aktywności, 285
 - nazwy, 278, 297
 - optymalna długość nazwy, 300
 - ostrzeżenia kompilatora, 281
 - powiązania hybrydowe, 294
 - reinicjalizacja, 281
 - rozpiętość, 283
 - trwałość, 289
 - typy danych, 292
 - ukryte znaczenie, 294
 - wiązanie wartości, 290
- zmiennie logiczne, 306, 336, 352
- zmiennie pętli, 419
- zmiennie składowe, 311
- zmiennie stanu, 134, 304
- zmiennie statusu, 304
- zmiennie tymczasowe, 305
- zmiennie globalne, 311, 371, 602
 - nazwy, 379
- zmniejszanie
 - ilość pracy w pętli, 656
 - obciążenie, 659
 - obciążenie wyrażeń, 665
 - złożoność, 143, 183, 188
- znaczniki, 304
 - znaczniki logiczne, 470
 - znaczniki w kodzie, 823
- znaki, 333, 351
- zwartość, 141
- zwielokrotnienie wejściowe, 116
- zwiększanie
 - przejrzystość kodu, 790
 - przenośność kodu, 201
 - wydajność, 201
- zwracanie kodu błędu, 231
- zwykle pętle przerywane, 407

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**

Steve McConnell wie więcej o budowie oprogramowania niż ktokolwiek inny; mamy ogromne szczęście, że zdecydował się podzielić swoim doświadczeniem oraz wiedzą w tej ważnej i oryginalnej książce.

Alan Cooper, „ojciec” języka Visual Basic, autor książki *About Face*

Zapewne każdy zgodzi się ze stwierdzeniem, że jeśli jakiś proces odpowiada za nawet 70% błędów w gotowym produkcie, z pewnością wymaga znaczącego usprawnienia... Czy masz jednak świadomość, że właśnie tyle problemów generuje samo wytwarzanie oprogramowania? Te błędy nie tylko powodują usterki w już gotowych programach, niespełniających oczekiwań klientów — odpowiadają także za znaczne opóźnienia przy realizacji zleconych projektów i nagminne przekraczanie zaplanowanego budżetu. Każdy ambitny programista staje zatem przed koniecznością zdobycia wiedzy o takich metodach pracy, które pozwolą szybciej i efektywniej realizować projekty, a przy tym zapewniać najwyższą jakość tworzonego kodu. W końcu na podstawie tych umiejętności oceniana jest także wartość danego programisty w zespole.

Z tych właśnie powodów niniejsza książka, będąca przejrzystą kompilacją najlepszych technik programowania, zdobyła tak wielkie uznanie w środowisku zawodowców i studentów, osiągając miano podręcznika kultowego. Przed Tobą drugie, zaktualizowane wydanie słynnej publikacji, w której Steve McConnell przedstawia wszystkie aspekty budowy programów, takie jak jakość czy podejście do procesu wytwarzania. Autor rozwija tu tak istotne zagadnienia, jak przebieg budowy klasy, techniki pracy z danymi i strukturami sterującymi, debugowanie, refaktoryzowanie oraz metody i strategie optymalizacji. Znajdziesz tu dziesiątki list kontrolnych, pomocnych w ocenianiu architektury, jakości klas i procedur, nazw zmiennych czy struktur sterujących, a także ponad 500 przykładów dobrego i złego kodu. Dowiesz się, co było przyczyną wielu typowych problemów w przeszłości i jak ich dzisiaj unikać. Opisane metody pracy pomogą utrzymać kontrolę nad dużymi projektami oraz efektywnie rozwijać i modyfikować oprogramowanie w odpowiedzi na zmiany wymagań. Co ważne, można je skutecznie wykorzystywać niezależnie od stosowanego języka programowania!

Dzięki tej książce nauczysz się skutecznie:

- » projektować z zachowaniem minimalnej złożoności;
- » praktycznie wykorzystywać metody pracy zespołowej;
- » programować defensywnie, by unikać błędów w kodzie i jak najszybciej je z niego usuwać;
- » wykorzystywać okazje do refaktoryzacji oraz rozwijania kodu i robić to w sposób bezpieczny;
- » korzystać z metod programowania dopasowanych do projektu;
- » szybko i efektywnie debugować;
- » wcześniej i we właściwy sposób rozwiązywać najważniejsze problemy z konstrukcją oprogramowania;
- » dbać o jakość kodu od pierwszego do ostatniego dnia projektu.

Zdobądź kluczowe umiejętności tworzenia najwyższej jakości oprogramowania!

Steve McConnell jest jednym z najbardziej cenionych ekspertów w świecie informatyki. Jest głównym programistą w firmie Construx Software oraz znanym autorem bestsellerowych książek, m.in. *Kod doskonały* i *Rapid Development* — obie zostały uhonorowane nagrodą Jolt magazynu „Software Development”, przyznawaną co roku najlepszej książce poświęconej procesowi wytwarzania oprogramowania. Brał udział w projektach realizowanych dla takich firm, jak Microsoft czy Boeing, a w 1998 roku czytelnicy magazynu „Software Development” uznali go obok Billa Gatesa i Linusa Torvaldsa za jedną z trzech najbardziej wpływowych osób w branży oprogramowania.

Helion

księgarnia internetowa



<http://helion.pl>

Helion SA
ul. Kościuszki 1c, 44-100 Gliwice
tel.: 32 230 98 63
e-mail: helion@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/novosoci>



0 601 339900

Informatyka w najlepszym wydaniu

ISBN 978-83-283-3489-2



cena: 129,00 zł

sięgnij po WIĘCEJ



KOD KORZYŚCI

Microsoft