Microsoft

KOD DOSKONAŁY

2

Wydanie drugie

Jak tworzyć oprogramowanie pozbawione błędów

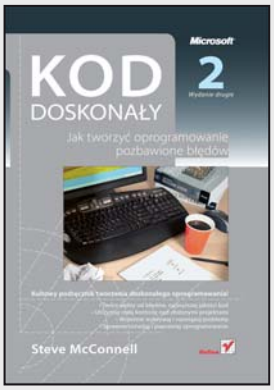
Kultowy podręcznik tworzenia doskonałego oprogramowania!

- 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

Kod doskonały. Jak tworzyć oprogramowanie pozbawione błędów. Wydanie II



Autor: Steve McConnell
Tłumaczenie: Paweł Koronkiewicz
ISBN: 978-83-246-2752-3
Tytuł oryginału: Code Complete: A Practical Handbook of Software Construction, Second Edition
Format: 172×245, stron: 960

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 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 powodują nie tylko 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 właśnie 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!

Posiądź kluczowe umiejętności tworzenia najwyższej jakości oprogramowania!

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

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. Pomiary ... 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. Pomiary 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. Periodyki ... 895
35.4. Plan czytelniczy programisty ... 896
35.5. Stowarzyszenia zawodowe ... 898
Bibliografia ... 899
Skorowidz ... 919
Steve McConnell ... 947

Rozdział 8.

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 spoczywa 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 zabezpieczaniu 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 procedur 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", przedstawię 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 zatonęła 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 znacznie 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 wyłączaniu 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 opisania warunków wstępnych i końcowych procedury `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 )
    ...

    '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 powinien 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 pojawią 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 )
    ...

    '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 asercji.

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, podstawiać 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 przerywać 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 obsługiwanie 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 niejednolitym 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. *corectness*) 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 — jakikolwiek 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 niefunkcjonalnych. 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 nieznający 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 zawiłej 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` (wyrzuć), aby go **zgłosić**, i przekazuje jednocześnie obiekt wyjątku. Kod w innej procedurze, wyżej w hierarchii wywołań, używa bloku `try-catch` (próbuj-przechwyć), aby wyjątek **przechwycić**.

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 `Exception` lub obiekt klasy pochodnej; wskaźnik do obiektu; odwołanie do obiektu; typ taki jak `string` lub `int`	obiekt `Exception` lub obiekt klasy pochodnej	obiekt `Exception` lub obiekt klasy pochodnej
Skutek nieprzechwycenia wyjątku	wywołanie procedury `std::unexpected()`, standardowo wywołującej procedurę `std::terminate()`, która standardowo wywołuje `abort()`	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 błędu 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 destruktora. 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 powstrzymywanie 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:

Klasa, która zgłasza wyjątek o niewłaściwym poziomie abstrakcji (Java)

```
class Employee {                                        // pracownik
    ...
    public TaxId GetTaxId() throws EOFException {   // wyjątek EOF
        ...
    }
    ...
}
```

Deklaracja wyjątku na złym poziomie abstrakcji.

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:

Klasa, która zgłasza wyjątek o właściwym poziomie abstrakcji (Java)

```
class Employee {                                        // pracownik
    ...
    public TaxId GetTaxId() throws EmployeeDataNotAvailable {
                                                        // dane pracownika niedostępne
        ...
    }
    ...
}
```

Deklaracja wyjątku na dopasowanym poziomie abstrakcji.

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 informację 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( _                          // raportuj wyjątek
    ByVal className, _                          // nazwa klasy
    ByVal thisException As Exception _          // 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 ujednolicić 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).

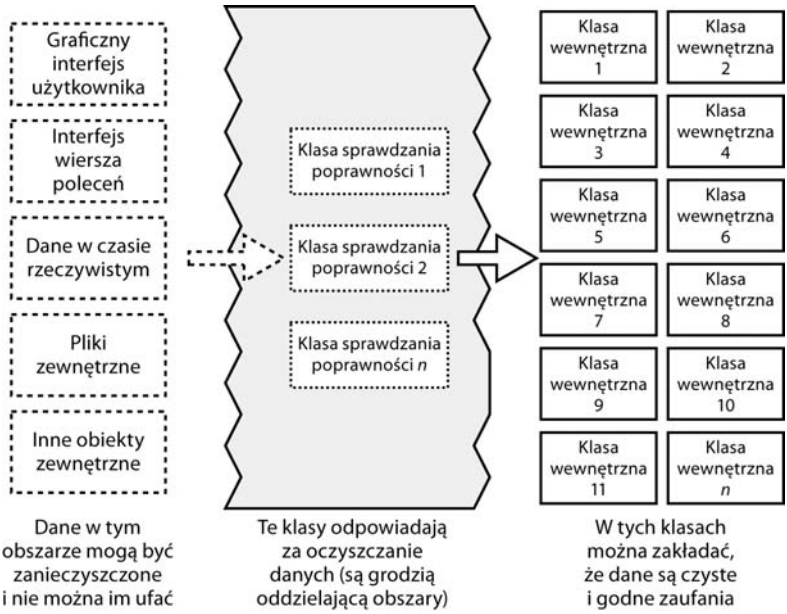
8.5. Ograniczanie zasięgu szkód powodowanych przez błędy

Ograniczanie zasięgu powstałych w wyniku błędów szkód ma na celu opanowanie trudnej sytuacji. Jest to działanie podobne do dzielenia statku szczelnie izolowanymi grodziami. 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.

Bezzwłocznie 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 wyliczeniowego (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 poczwórnie 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.

Wcześnie 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.
- Wypełniaj do końca przydzieloną pamięć, aby wykryć ewentualne błędy alokacji.
- Wypełniaj 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 wypełniaj 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 podrozdziale 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 skutkować 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 przyszła 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, zadbaj 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 doskonałe 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.