

O'REILLY®

Mistrz języka C

Najlepsze
zasady, praktyki
i wzorce



Helion 

Christopher Preschern

Tytuł oryginału: Fluent C: Principles, Practices, and Patterns

Tłumaczenie: Tomasz Walczak

ISBN: 978-83-8322-722-1

© 2023 Helion S.A.

Authorized Polish translation of the English edition of *Fluent C* ISBN 9781492097334 © 2023
Christopher Preschern.

This translation is published and sold by permission of O'Reilly Media, Inc., which owns or controls 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 wydawca 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 wydawca nie ponoszą również żadnej odpowiedzialności za ewentualne szkody wynikłe z wykorzystania informacji zawartych w książce.

Drogi Czytelniku!

Jeżeli chcesz ocenić tę książkę, zajrzyj pod adres

<https://helion.pl/user/opinie/misjec>

Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

Pliki z przykładami omawianymi w książce można znaleźć pod adresem:

<https://ftp.helion.pl/przyklady/misjec.zip>

Helion S.A.

ul. Kościuszki 1c, 44-100 Gliwice

tel. 32 230 98 63

e-mail: helion@helion.pl

WWW: <https://helion.pl> (księgarnia internetowa, katalog książek)

Printed in Poland.

- Kup książkę
- Poleć książkę
- Oceń książkę

- Księgarnia internetowa
- Lubię to! » Nasza społeczność

Spis treści

Przedmowa.....	13
----------------	----

Część I. Wzorce w języku C	27
-----------------------------------	-----------

1. Obsługa błędów	29
Przykład roboczy	30
Podział funkcji	31
Kontekst	31
Problem	32
Rozwiązanie	32
Skutki	33
Znane zastosowania	33
Zastosowanie w przykładzie roboczym	33
Klauzula zabezpieczająca	34
Kontekst	34
Problem	34
Rozwiązanie	35
Skutki	36
Znane zastosowania	36
Zastosowanie w przykładzie roboczym	36
Reguła samuraja	37
Kontekst	37
Problem	37
Rozwiązanie	38
Skutki	39
Znane zastosowania	40
Zastosowanie w przykładzie roboczym	40
Obsługa błędów z użyciem instrukcji goto	41
Kontekst	41
Problem	41

Rozwiązanie	41
Skutki	42
Znane zastosowania	43
Zastosowanie w przykładzie roboczym	43
Zapisywanie informacji o porządkowaniu	44
Kontekst	44
Problem	44
Rozwiązanie	44
Skutki	45
Znane zastosowania	45
Zastosowanie w przykładzie roboczym	46
Obiektowa obsługa błędów	46
Kontekst	46
Problem	46
Rozwiązanie	47
Skutki	47
Znane zastosowania	48
Zastosowanie w przykładzie roboczym	48
Podsumowanie	50
Dalsza lektura	50
Co dalej?	51
2. Zwracanie informacji o błędach	52
Przykład roboczy	53
Zwracanie kodów stanu	55
Kontekst	55
Problem	55
Rozwiązanie	55
Skutki	56
Znane zastosowania	57
Zastosowanie w przykładzie roboczym	58
Zwracanie adekwatnych informacji o błędach	61
Kontekst	61
Problem	61
Rozwiązanie	62
Skutki	63
Znane zastosowania	64
Zastosowanie w przykładzie roboczym	65
Specjalne zwracane wartości	66
Kontekst	66
Problem	66
Rozwiązanie	67

Skutki	68
Znane zastosowania	68
Zastosowanie w przykładzie roboczym	69
Rejestrowanie błędów	70
Kontekst	70
Problem	70
Rozwiązanie	71
Skutki	72
Znane zastosowania	73
Zastosowanie w przykładzie roboczym	73
Podsumowanie	76
Dalsza lektura	76
Co dalej?	76
3. Zarządzanie pamięcią	77
Przechowywanie danych i problemy z pamięcią dynamiczną	77
Przykład roboczy	80
Zacznij od stosu	81
Kontekst	81
Problem	82
Rozwiązanie	82
Skutki	83
Znane zastosowania	84
Zastosowanie w przykładzie roboczym	84
Pamięć wieczna	84
Kontekst	84
Problem	85
Rozwiązanie	85
Skutki	86
Znane zastosowania	87
Zastosowanie w przykładzie roboczym	87
Leniwe porządkowanie	88
Kontekst	88
Problem	88
Rozwiązanie	88
Skutki	89
Znane zastosowania	90
Zastosowanie w przykładzie roboczym	90
Określony właściciel	91
Kontekst	91
Problem	92
Rozwiązanie	92

Skutki	93
Znane zastosowania	93
Zastosowanie w przykładzie roboczym	94
Nakładka do alokacji pamięci	95
Kontekst	95
Problem	95
Rozwiązanie	96
Skutki	97
Znane zastosowania	98
Zastosowanie w przykładzie roboczym	98
Sprawdzanie wskaźników	99
Kontekst	99
Problem	99
Rozwiązanie	100
Skutki	100
Znane zastosowania	101
Zastosowanie w przykładzie roboczym	101
Pula pamięci	102
Kontekst	102
Problem	102
Rozwiązanie	103
Skutki	104
Znane zastosowania	105
Zastosowanie w przykładzie roboczym	106
Podsumowanie	108
Dalsza lektura	108
Co dalej?	109
4. Zwracanie danych z funkcji w C	110
Przykład roboczy	110
Zwracana wartość	112
Kontekst	112
Problem	112
Rozwiązanie	112
Skutki	113
Znane zastosowania	114
Zastosowanie w przykładzie roboczym	114
Parametry wyjściowe	115
Kontekst	115
Problem	115
Rozwiązanie	116
Skutki	117

Znane zastosowania	118
Zastosowanie w przykładzie roboczym	118
Zagregowana instancja	119
Kontekst	119
Problem	119
Rozwiązanie	120
Skutki	122
Znane zastosowania	122
Zastosowanie w przykładzie roboczym	123
Niemodyfikowalna instancja	123
Kontekst	123
Problem	123
Rozwiązanie	124
Skutki	125
Znane zastosowania	126
Zastosowanie w przykładzie roboczym	126
Bufor należący do jednostki wywołującej	127
Kontekst	127
Problem	127
Rozwiązanie	128
Skutki	129
Znane zastosowania	129
Zastosowanie w przykładzie roboczym	130
Alokacja w jednostce wywoływanej	131
Kontekst	131
Problem	131
Rozwiązanie	131
Skutki	133
Znane zastosowania	133
Zastosowanie w przykładzie roboczym	134
Podsumowanie	135
Co dalej?	135
5. Czas życia i własność danych	136
Bezstanowy moduł oprogramowania	138
Kontekst	138
Problem	138
Rozwiązanie	138
Skutki	140
Znane zastosowania	140
Zastosowanie w przykładzie roboczym	141

Moduł oprogramowania ze stanem globalnym	142
Kontekst	142
Problem	142
Rozwiązanie	142
Skutki	144
Znane zastosowania	145
Zastosowanie w przykładzie roboczym	145
Instancja należąca do jednostki wywołującej	146
Kontekst	146
Problem	147
Rozwiązanie	147
Skutki	149
Znane zastosowania	149
Zastosowanie w przykładzie roboczym	150
Współdzielona instancja	151
Kontekst	151
Problem	152
Rozwiązanie	152
Skutki	154
Znane zastosowania	155
Zastosowanie w przykładzie roboczym	156
Podsumowanie	157
Dalsza lektura	158
Co dalej?	158
6. Elastyczne API	159
Pliki nagłówkowe	161
Kontekst	161
Problem	161
Rozwiązanie	161
Skutki	162
Znane zastosowania	163
Zastosowanie w przykładzie roboczym	163
Uchwyty	164
Kontekst	164
Problem	164
Rozwiązanie	164
Skutki	165
Znane zastosowania	166
Zastosowanie w przykładzie roboczym	166
Dynamiczny interfejs	167
Kontekst	167
Problem	167

Rozwiązanie	167
Skutki	168
Znane zastosowania	169
Zastosowanie w przykładzie roboczym	169
Kontrolowanie funkcji	170
Kontekst	170
Problem	170
Rozwiązanie	171
Skutki	172
Znane zastosowania	172
Zastosowanie w przykładzie roboczym	172
Podsumowanie	174
Dalsza lektura	174
Co dalej?	175
7. Elastyczne interfejsy iteratorów	176
Przykład roboczy	178
Dostęp za pomocą indeksu	178
Kontekst	178
Problem	178
Rozwiązanie	179
Skutki	180
Znane zastosowania	181
Zastosowanie w przykładzie roboczym	181
Iterator w postaci kursora	183
Kontekst	183
Problem	183
Rozwiązanie	183
Skutki	185
Znane zastosowania	185
Zastosowanie w przykładzie roboczym	186
Iterator z wywołaniami zwrotnymi	187
Kontekst	187
Problem	187
Rozwiązanie	188
Skutki	189
Znane zastosowania	190
Zastosowanie w przykładzie roboczym	191
Podsumowanie	192
Dalsza lektura	193
Co dalej?	193

8. Organizowanie plików w modułowych programach	194
Przykład roboczy	196
Zabezpieczanie instrukcji include	198
Kontekst	198
Problem	198
Rozwiązanie	198
Skutki	199
Znane zastosowania	200
Zastosowanie w przykładzie roboczym	200
Katalogi modułów oprogramowania	200
Kontekst	200
Problem	201
Rozwiązanie	201
Skutki	202
Znane zastosowania	203
Zastosowanie w przykładzie roboczym	203
Globalny katalog na dołączane pliki	205
Kontekst	205
Problem	205
Rozwiązanie	205
Skutki	207
Znane zastosowania	207
Zastosowanie w przykładzie roboczym	207
Samodzielny komponent	209
Kontekst	209
Problem	209
Rozwiązanie	209
Skutki	211
Znane zastosowania	211
Zastosowanie w przykładzie roboczym	211
Kopiowanie API	214
Kontekst	214
Problem	214
Rozwiązanie	214
Skutki	216
Znane zastosowania	217
Zastosowanie w przykładzie roboczym	217
Podsumowanie	221
Co dalej?	222

9. Ucieczka z piekła instrukcji #ifdef	223
Przykład roboczy	225
Unikanie wariantów	226
Kontekst	226
Problem	226
Rozwiązanie	227
Skutki	228
Znane zastosowania	228
Zastosowanie w przykładzie roboczym	228
Izolowane podstawowe jednostki kodu	229
Kontekst	229
Problem	230
Rozwiązanie	230
Skutki	231
Znane zastosowania	231
Zastosowanie w przykładzie roboczym	232
Atomowe podstawowe jednostki kodu	233
Kontekst	233
Problem	233
Rozwiązanie	233
Skutki	234
Znane zastosowania	234
Zastosowanie w przykładzie roboczym	235
Warstwa abstrakcji	236
Kontekst	236
Problem	236
Rozwiązanie	236
Skutki	237
Znane zastosowania	238
Zastosowanie w przykładzie roboczym	238
Rozdzielanie implementacji wariantów	241
Kontekst	241
Problem	241
Rozwiązanie	241
Skutki	242
Znane zastosowania	243
Zastosowanie w przykładzie roboczym	243
Podsumowanie	246
Dalsza lektura	247
Co dalej?	247

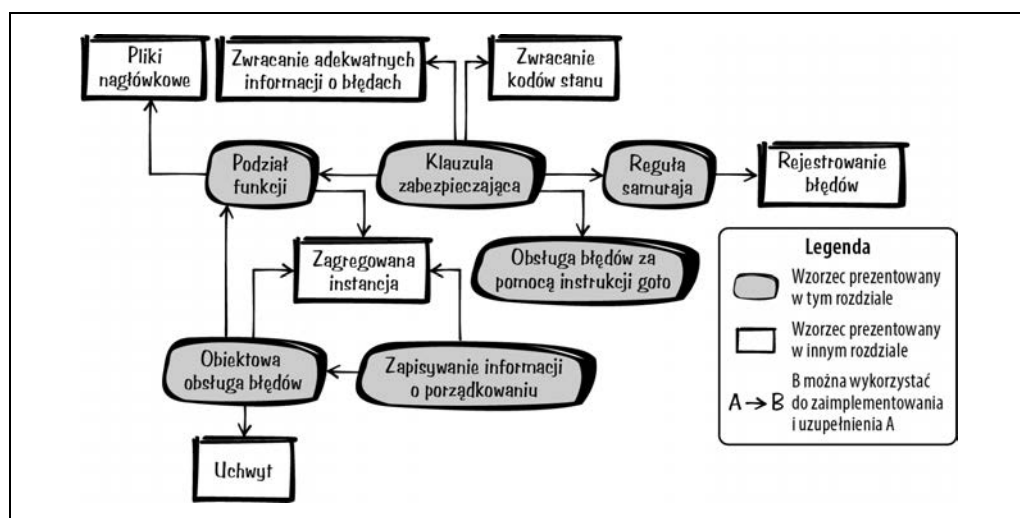
10. Implementowanie mechanizmu rejestrowania informacji	251
Historia wykorzystana do prezentowania wzorców	251
Organizowanie plików	252
Centralna funkcja rejestrowania informacji	252
Filtr źródła rejestrowanych informacji	254
Warunkowe rejestrowanie informacji	256
Rejestrowanie informacji w różnych miejscach	257
Rejestrowanie informacji w pliku	258
Pliki działające na różnych platformach	259
Stosowanie mechanizmu rejestrowania informacji	262
Podsumowanie	263
11. Tworzenie systemu zarządzania kontami użytkowników	265
Historia wykorzystana do prezentowania wzorców	265
Organizowanie danych	265
Organizowanie plików	266
Obsługa błędów w procesie uwierzytelniania	268
Rejestrowanie błędów w procesie uwierzytelniania	270
Obsługa błędów w procesie dodawania użytkowników	271
Iterowanie	273
Korzystanie z systemu zarządzania kontami użytkowników	275
Podsumowanie	276
12. Zakończenie	278
Czego się nauczyłeś?	278
Dalsza lektura	278
Uwagi końcowe	279

Obsługa błędów

Obsługa błędów jest ważnym aspektem pisania oprogramowania. Jeśli o nią odpowiednio nie zadbasz, oprogramowanie będzie trudne do rozszerzania i konserwacji. Języki programowania takie jak C++ lub Java udostępniają wyjątki i destruktory, dzięki którym obsługa błędów jest łatwiejsza. W C takie mechanizmy nie są natywnie dostępne, a literatura poświęcona solidnej obsłudze błędów w tym języku jest rozproszona w wielu miejscach w internecie.

W tym rozdziale znajdziesz zebraną wiedzę na temat należytej obsługi błędów. Przedstawiam ją w formie wzorców obsługi błędów w C i przykładu, w którym stosuję te wzorce. Ilustrują one dobre decyzje projektowe. Wyjaśniam też, kiedy należy je stosować i jakie są tego skutki. Z perspektywy programisty wzorce eliminują ciężar podejmowania wielu szczegółowych decyzji. Programista może skupić się na wiedzy ujętej we wzorcach i wykorzystać je jako punkt wyjścia do pisania kodu wysokiej jakości.

Na rysunku 1.1 przedstawiam przegląd wzorców omawianych w tym rozdziale i relacje między nimi. W tabeli 1.1 znajdziesz zestawienie tych wzorców.



Rysunek 1.1. Schemat wzorców obsługi błędów

Tabela 1.1. Wzorce obsługi błędów

Nazwa wzorca	Podsumowanie
Podział funkcji	Funkcja wykonuje kilka zadań, przez co jest mało czytelna i trudna w konserwacji. Dlatego należy ją podzielić. Wyodrębnić fragment funkcji, który samodzielnie wykonuje przydatne operacje, utwórz na jego podstawie nową funkcję i ją wywołaj.
Klauzula zabezpieczająca	Funkcja jest mało czytelna i trudna w konserwacji, ponieważ łączy sprawdzanie warunków wstępnych z główną logiką programu. Dlatego należy sprawdzać, czy wymagane warunki wstępne są spełnione, a jeśli nie są, natychmiast zwracać sterowanie z funkcji.
Reguła samuraja	Gdy zwracasz informacje o błędzie, zakładasz, że jednostka wywołująca je sprawdza. Jednak jednostka wywołująca może pominąć takie testy, przez co błąd pozostanie niezauważony. Dlatego należy zwracać informacje tylko po udanym wykonaniu funkcji albo nie zwracać ich w ogóle. Jeśli wystąpi sytuacja, w której wiadomo, że błędu nie da się obsłużyć, należy zamknąć program.
Obsługa błędów z użyciem instrukcji goto	Kod staje się mało czytelny i trudny w konserwacji, gdy pozyskuje i zwalnia wiele zasobów w różnych miejscach funkcji. Dlatego cały kod do porządkowania zasobów i obsługi błędów należy umieścić na końcu funkcji. Jeśli zasobów nie da się pozyskać, należy użyć instrukcji goto do przeskoczenia do kodu do porządkowania zasobów.
Zapisywanie informacji o porządkowaniu	Trudno jest zapewnić czytelność i łatwą konserwację fragmentu kodu, jeśli pozyskuje on i zwalnia wiele zasobów, zwłaszcza gdy te zasoby są od siebie zależne. Dlatego należy wywoływać funkcje pozyskujące zasoby, dopóki można je wykonywać z powodzeniem, a także zapisywać, które z tych funkcji wymagają porządkowania. Funkcje porządkujące zasoby należy wywoływać w zależności od tych zapisanych wartości.
Obiektowa obsługa błędów	Gdy jedna funkcja wykonuje wiele zadań, na przykład pozyskuje zasoby, porządkuje je i z nich korzysta, kod jest trudny do implementacji, konserwacji i testowania, a także mało czytelny. Dlatego inicjalizowanie i porządkowanie należy umieszczać w odrębnych funkcjach. Mechanizm ten przypomina używanie konstruktorów i destruktorów w programowaniu obiektowym.

Przykład roboczy

Chcesz zaimplementować funkcję, która sprawdza występowanie określonych słów kluczowych w pliku i zwraca informacje o tym, czy je znalazła.

Standardowym sposobem informowania o błędzie w C jest używanie wartości zwracanej przez funkcję. Aby przekazać dodatkowe informacje o błędzie, w starszych funkcjach w C często przypisywano do zmiennej `errno` (zobacz plik `errno.h`) konkretny kod błędu. Jednostka wywołująca może wtedy sprawdzić zmienną `errno`, aby pobrać informacje o błędzie.

Jednak w poniższym kodzie zamiast zmiennej `errno` używane są zwracane wartości, ponieważ nie potrzebujesz bardzo szczegółowych informacji o błędzie. Pisziesz więc następujący początkowy fragment kodu:

```
int parseFile(char* file_name)
{
    int return_value = ERROR;
    FILE* file_pointer = 0;
    char* buffer = 0;

    if(file_name!=NULL)
    {
```

```

if(file_pointer=fopen(file_name, "r"))
{
    if(buffer=malloc(BUFFER_SIZE))
    {
        /* Przetwarzanie zawartości pliku. */
        return_value = NO_KEYWORD_FOUND;
        while(fgets(buffer, BUFFER_SIZE, file_pointer)!=NULL)
        {
            if(strcmp("KEYWORD_ONE\n", buffer)==0)
            {
                return_value = KEYWORD_ONE_FOUND_FIRST;
                break;
            }
            if(strcmp("KEYWORD_TWO\n", buffer)==0)
            {
                return_value = KEYWORD_TWO_FOUND_FIRST;
                break;
            }
        }
        free(buffer);
    }
    fclose(file_pointer);
}
return return_value;
}

```

W tym kodzie trzeba sprawdzać wartości zwracane przez wywołane funkcje, aby stwierdzić, czy wystąpił błąd. Z tego powodu powstają głęboko zagnieżdżone instrukcje `if`. Prowadzi to do następujących problemów:

- Funkcja jest długa i łączy kod obsługi błędów, inicjalizacji, porządkowania oraz wykonywania operacji, przez co jest trudna w konserwacji.
- Główny kod, który wczytuje i interpretuje dane z pliku, jest głęboko zagnieżdżony w klauzulach `if`, przez co trudno jest prześledzić logikę programu.
- Funkcje porządkujące są wyraźnie oddzielone od funkcji inicjalizujących, przez co łatwo jest zapomnieć o zwolnieniu niektórych zasobów. Jest to prawdą przede wszystkim w sytuacji, gdy funkcja zawiera kilka instrukcji `return`.

Aby ulepszyć kod, najpierw zastosuj **Podział funkcji**.

Podział funkcji

Kontekst

Masz funkcję, która wykonuje wiele operacji, na przykład alokuje zasób (pamięć dynamiczną, uchwyt pliku itp.), używa tego zasobu i go porządkuje.

Problem

Funkcja ma kilka zadań, przez co jest mało czytelna i trudna w konserwacji.

Taka funkcja może odpowiadać za alokację zasobów, wykonywanie operacji na tych zasobach i ich porządkowanie. Możliwe nawet, że kod porządkowania jest rozrzucony po funkcji i powielony w kilku miejscach. Przede wszystkim obsługa błędów związanych z nieudaną alokacją zasobów powoduje, że funkcja jest mało czytelna, ponieważ często konieczne są wtedy zagnieżdżone instrukcje i f.

Konieczność alokacji, porządkowania i użytkowania wielu zasobów w jednej funkcji powoduje, że łatwo jest zapomnieć o porządkowaniu zasobów. Dzieje się tak przede wszystkim w sytuacji, jeśli kod zostanie później zmodyfikowany. Na przykład jeżeli dodasz pośrodku kodu instrukcję return, łatwo będzie zapomnieć o porządkowaniu zasobów zaalokowanych już w tym miejscu funkcji.

Rozwiązanie

Podziel funkcję. Wyodrębnij fragment funkcji, który samodzielnie wykonuje przydatne operacje, utwórz na jego podstawie nową funkcję i ją wywołaj.

Aby ustalić, który fragment funkcji wyodrębnić, sprawdź, czy możesz przypisać mu sensowną nazwę i czy podział powoduje odizolowanie zadań. Może to na przykład spowodować, że w jednej funkcji znajdzie się tylko kod odpowiedzialny za wykonywanie operacji, a w drugiej sam kod obsługi błędów.

Przydatną wskazówką sugerującą, że funkcję warto podzielić, jest porządkowanie tego samego zasobu w wielu miejscach. W takiej sytuacji znacznie lepiej jest utworzyć jedną funkcję, która alokuje i porządkuje zasoby, i drugą funkcję, która z tych zasobów korzysta. W funkcji używającej zasobów można wtedy wygodnie wywołać wiele instrukcji return bez konieczności porządkowania zasobów przed każdą z nich, ponieważ odpowiada za to inna funkcja. Ilustruje to poniższy kod:

```
void someFunction()
{
    char* buffer = malloc(LARGE_SIZE);
    if(buffer)
    {
        mainFunctionality(buffer);
    }
    free(buffer);
}

void mainFunctionality()
{
    // Tu umieść implementację.
}
```

Teraz masz dwie funkcje zamiast jednej. To oczywiście oznacza, że wywołująca funkcja nie jest samodzielna i zależy od innej. Musisz określić miejsce, gdzie należy umieścić tę inną funkcję. Pierwszym pomysłem jest zapisanie jej w tym samym pliku co wywołującą funkcję, jeśli jednak nie są one ściśle powiązane, możesz rozważyć umieszczenie wywoływanej funkcji w odrębnym pliku z implementacją i dołączenie deklaracji tej funkcji w *Pliku nagłówkowym*.

Skutki

Kod został ulepszony, ponieważ dwie krótkie funkcje są czytelniejsze i łatwiejsze w konserwacji niż jedna długa funkcja. Kod jest czytelniejszy, gdyż funkcje porządkujące są bliżej funkcji wymagających porządkowania, a alokacja i porządkowanie zasobów nie mieszają się z główną logiką programu. Dzięki temu główna logika programu jest łatwiejsza w konserwacji i prościej będzie ją rozbudować w przyszłości.

Wywoływana funkcja może teraz obejmować kilka instrukcji `return`, ponieważ nie trzeba martwić się o porządkowanie zasobów przed każdą z nich. Porządkowanie odbywa się w jednym miejscu w funkcji wywołującej.

Jeśli funkcja wywoływana używa wielu zasobów, trzeba je wszystkie do niej przekazać. Stosowanie wielu parametrów funkcji zmniejsza czytelność funkcji, a przypadkowa zmiana kolejności parametrów w funkcji wywołującej może prowadzić do błędów programistycznych. Aby tego uniknąć, można wtedy zastosować **Zagregowaną instancję**.

Znane zastosowania

Oto przykłady ilustrujące zastosowania tego wzorca:

- W prawie każdym kodzie w C znajdują się fragmenty, w których ten wzorec jest stosowany, i trudniejsze w konserwacji miejsca, gdzie nie jest on używany. Zgodnie z tym, co Robert C. Martin pisze w książce *Czysty kod. Podręcznik dobrego programisty* (Helion, 2009), każda funkcja powinna mieć dokładnie jedno zadanie (zasada jednego zadania), dlatego obsługę zasobów i logikę programu zawsze należy umieszczać w odrębnych funkcjach.
- W portlandzkim repozytorium wzorców ten wzorec nosi nazwę **Nakładka na funkcję**.
- W programowaniu obiektowym wzorec **Metoda szablonowa** opisuje strukturyzowanie kodu przez jego podział.
- Kryteria określające, kiedy i gdzie dzielić funkcje, są opisane w książce *Refaktoryzacja. Ulepszanie struktury istniejącego kodu* Martina Fowlera (Helion, 2011). Są one ujęte we wzorcu **Wyodrębnianie metody**.
- W grze NetHack zastosowano ten wzorec w funkcji `read_config_file`, w której obsługiwane są zasoby i wywoływana jest funkcja `parse_conf_file` operująca tymi zasobami.
- W kodzie projektu OpenWrt ten wzorec jest używany w kilku miejscach do obsługi bufora. Na przykład kod odpowiedzialny za obliczanie skrótów MD5 alokuje bufor, przekazuje go do innej funkcji operującej tym buforem, a następnie porządkuje bufor.

Zastosowanie w przykładzie roboczym

Kod już wygląda znacznie lepiej. Zamiast jednej olbrzymiej funkcji obecnie masz dwie długie funkcje o różnych zadaniach: jedna funkcja odpowiada za pobieranie i zwalnianie zasobów, a druga za wyszukiwanie słów kluczowych. Ilustruje to poniższy kod:

```
int searchFileForKeywords(char* buffer, FILE* file_pointer)
{
```

```

while(fgets(buffer, BUFFER_SIZE, file_pointer)!=NULL)
{
    if(strcmp("KEYWORD_ONE\n", buffer)==0)
    {
        return KEYWORD_ONE_FOUND_FIRST;
    }
    if(strcmp("KEYWORD_TWO\n", buffer)==0)
    {
        return KEYWORD_TWO_FOUND_FIRST;
    }
}
return NO_KEYWORD_FOUND;
}

int parseFile(char* file_name)
{
    int return_value = ERROR;
    FILE* file_pointer = 0;
    char* buffer = 0;

    if(file_name!=NULL)
    {
        if(file_pointer=fopen(file_name, "r"))
        {
            if(buffer=malloc(BUFFER_SIZE))
            {
                return_value = searchFileForKeywords(buffer, file_pointer);
                free(buffer);
            }
            fclose(file_pointer);
        }
    }
    return return_value;
}

```

Głębokość kaskady wywołań if się zmniejszyła, jednak funkcja `parseFile` nadal zawiera trzy instrukcje if sprawdzające błędy alokacji zasobów, czyli wciąż zdecydowanie za dużo. Możesz zwiększyć przejrzystość funkcji przez zastosowanie wzorca *Klauzula zabezpieczająca*.

Klauzula zabezpieczająca

Kontekst

Masz funkcję wykonującą zadanie, które można z powodzeniem ukończyć tylko w określonych warunkach (na przykład dla poprawnych parametrów wejściowych).

Problem

Funkcja jest mało czytelna i trudna w konserwacji, ponieważ łączy sprawdzanie warunków wstępnych z główną logiką programu.

Alokacja zasobów zawsze wymaga ich uporządkowania. Jeśli zaalokujesz zasób, a później wykryjesz, że jakiś warunek wstępny funkcji nie jest spełniony, trzeba uporządkować ten zasób.

Trudno jest prześledzić przepływ programu, jeśli w funkcji rozproszonych jest kilka testów warunków wstępnych. Dzieje się tak zwłaszcza w sytuacji, gdy takie testy znajdują się w zagnieżdżonych instrukcjach `if`. Kiedy takich testów jest wiele, funkcja staje się bardzo długa, co samo w sobie jest zapachem kodu.



Zapach kodu

Kod źle pachnie, jeśli ma niewłaściwą strukturę lub został zaprogramowany w sposób utrudniający konserwację. Przykładowe zapachy kodu to bardzo długie funkcje lub powtarzający się kod. Więcej przykładowych zapachów kodu i metod zapobiegania im znajdziesz w książce *Refaktoryzacja. Ulepszanie struktury istniejącego kodu* Martina Fowlera (Helion, 2011).

Rozwiązanie

Sprawdź, czy wymagane warunki wstępne są spełnione, a jeśli nie są, natychmiast zwracaj sterowanie z funkcji.

Możesz na przykład sprawdzać poprawność parametrów wejściowych lub badać, czy program znajduje się w stanie umożliwiającym wykonanie reszty funkcji. Dobrze rozważ, jakiego rodzaju warunki wstępne do wywołania funkcji chcesz zastosować. Z jednej strony ułatwisz sobie pracę, jeśli będziesz bardzo ściśle kontrolować dozwolone dane wejściowe funkcji. Jednak z drugiej strony ułatwisz życie jednostkom wywołującym Twoją funkcję, jeżeli dasz im więcej swobody w zakresie dopuszczalnych danych wejściowych (opisuje to prawo Postela: „Bądź konserwatywny w tym, co robisz; bądź liberalny w tym, co akceptujesz od innych”).

Jeśli masz wiele testów warunków wstępnych, możesz wywołać odrębną funkcję do ich wykonania. Zawsze jednak przeprowadzaj takie testy przed alokacją zasobów, pozwala to bowiem bardzo łatwo zwrócić sterowanie z funkcji, gdyż nie trzeba porządkować zasobów.

Wyraźnie opisz warunki wstępne funkcji w jej interfejsie. Najlepszym miejscem na ich udokumentowanie jest plik nagłówkowy zawierający deklarację danej funkcji.

Jeśli jest ważne, aby jednostka wywołująca wiedziała, który warunek wstępny nie został spełniony, możesz przekazać jej informacje o błędzie. Możesz na przykład zastosować **Zwracanie kodów stanu**, ale upewnij się, że **Zwracasz adekwatne informacje o błędach**.

someFile.h

```
/* Ta funkcja operuje na parametrze 'user_input', który musi być różny od NULL. */  
void someFunction(char* user_input);
```

someFile.c

```
void someFunction(char* user_input)  
{  
    if(user_input == NULL)
```

```

{
    return;
}
operateOnData(user_input);
}

```

Skutki

Natychmiastowe zwracanie sterowania, gdy warunki wstępne nie są spełnione, zwiększa czytelność kodu w porównaniu z wersją z zagnieżdżonymi instrukcjami `if`. Jest to świetnie widoczne w kodzie, w którym wykonywanie funkcji nie jest kontynuowane, gdy warunki wstępne nie są spełnione. Ten wzorzec pozwala ściśle oddzielić warunki wstępne od reszty kodu.

Jednak niektóre wytyczne dotyczące pisania kodu zabraniają zwracania sterowania pośrodku funkcji. Na przykład w kodzie, który wymaga formalnego udowodnienia poprawności, instrukcje `return` są zwykle dozwolone tylko na samym końcu funkcji. W takiej sytuacji można zastosować **Zapisywanie informacji o porządkowaniu**. Ten sam wzorzec jest dobrym wyborem także w sytuacji, kiedy potrzebujesz scentralizowanej obsługi błędów.

Znane zastosowania

Ten wzorzec jest stosowany w następujących przykładowych miejscach:

- Wzorzec **Klauzula zabezpieczająca** jest opisany w portlandzkim repozytorium wzorców.
- W artykule *Error Detection* (Proceedings of the 2nd EuroPLoP conference, 1997) Klaus Renzel opisuje bardzo podobny wzorzec wykrywania błędów, w którym sugeruje wprowadzenie testów warunków wstępnych i warunków końcowych.
- W kodzie gry NetHack ten wzorzec jest stosowany w kilku miejscach, na przykład w funkcji `placebc`. Funkcja ta nakłada w ramach kary łańcuch na bohatera gry, co spowalnia jego szybkość. Jeśli obiekt łańcucha jest niedostępny, funkcja natychmiast zwraca sterowanie.
- Ten wzorzec jest używany w kodzie biblioteki *OpenSSL*. Na przykład funkcja `SSL_new` natychmiast zwraca sterowanie, gdy otrzyma błędne parametry wejściowe.
- W narzędziu Wireshark kod funkcji `capture_stats`, która odpowiada za zbieranie statystyk w trakcie przechwytywania pakietów sieciowych, najpierw sprawdza poprawność parametrów wejściowych, a jeśli są one błędne, natychmiast zwraca sterowanie.

Zastosowanie w przykładzie roboczym

W poniższym kodzie pokazuję, jak zastosować **Klauzulę zabezpieczającą** w funkcji `parseFile` do sprawdzania jej warunków wstępnych:

```

int parseFile(char* file_name)
{
    int return_value = ERROR;
    FILE* file_pointer = 0;
    char* buffer = 0;

```

```

if(file_name==NULL) ❶
{
    return ERROR;
}
if(file_pointer=fopen(file_name, "r"))
{
    if(buffer=malloc(BUFFER_SIZE))
    {
        return_value = searchFileForKeywords(buffer, file_pointer);
        free(buffer);
    }
    fclose(file_pointer);
}
return return_value;
}

```

❶ Jeśli zostaną podane niepoprawne parametry, funkcja natychmiast zwraca sterowanie i nie jest konieczne porządkowanie, ponieważ żadne zasoby nie zostały jeszcze zajęte.

Kod *Zwraca kody stanu*, aby zaimplementować wzorzec *Klauzula zabezpieczająca*. Gdy parametr ma wartość NULL, kod zwraca stałą ERROR. Jednostka wywołująca może wtedy sprawdzić *Zwracaną wartość*, aby stwierdzić, czy do funkcji przekazano błędny parametr NULL. Jednak taki parametr zwykle oznacza błąd programistyczny, a przekazywanie takich informacji w kodzie nie jest dobrym pomysłem. W takiej sytuacji łatwiej jest zastosować *Regułę samuraja*.

Reguła samuraja

Kontekst

Masz kod ze skomplikowaną obsługą błędów, a niektóre z nich są bardzo poważne. System nie wykonuje operacji o krytycznym znaczeniu dla bezpieczeństwa, a wysoka dostępność nie jest bardzo istotna.

Problem

Gdy zwracasz informacje o błędzie, zakładasz, że jednostka wywołująca je sprawdza. Jednak jednostka wywołująca może pominąć takie testy, przez co błąd pozostanie niezauważony.

W C nie trzeba sprawdzać wartości zwracanych przez wywoływane funkcje, a jednostka wywołująca może zignorować taką wartość. Jeśli błąd, który wystąpił w funkcji, jest poważny i nie można go w kontrolowany sposób obsłużyć w jednostce wywołującej, ta jednostka nie powinna decydować, czy i jak poradzić sobie z takim błędem. Zamiast tego należy się upewnić, że zostaną podjęte odpowiednie działania.

Nawet jeśli jednostka wywołująca obsłuży błąd, programy i tak dość często ulegają wtedy awarii lub występuje jakiś problem. Usterka może się pojawić w innym miejscu, na przykład gdzieś w kodzie wywołującym daną jednostkę wywołującą, który nie potrafi poprawnie obsłużyć błędu. Wtedy obsługa błędu powoduje jego ukrycie, przez co znacznie trudniej jest go debugować i znaleźć źródło problemu.

Niektóre błędy w kodzie występują bardzo rzadko. **Zwracanie kodu stanu** w takiej sytuacji i obsługiwanie błędu w kodzie jednostki wywołującej znacznie zmniejsza czytelność, ponieważ odwraca uwagę od głównej logiki programu i celu kodu jednostki wywołującej. W jednostce wywołującej potrzebnych może być wiele wierszy kodu do obsługi bardzo rzadkich sytuacji.

Zwracanie informacji o błędzie rodzi też inny problem: jak zwracać te informacje? Zastosowanie **Zwracanej wartości** lub **Parametru wyjściowego funkcji** do zwracania informacji o błędzie sprawia, że sygnatura funkcji staje się bardziej skomplikowana, a kod trudniejszy do zrozumienia. Z tego powodu niepożądane może być używanie dodatkowych parametrów dla funkcji, która zwraca jedynie informacje o błędach.

Rozwiązanie

Dlatego należy zwracać informacje tylko po udanym wykonaniu funkcji albo nie zwracać ich w ogóle (jest to Reguła samuraja). Jeśli wystąpi sytuacja, w której wiadomo, że błędu nie da się obsłużyć, należy zamknąć program.

Nie stosuj **Parametrów wyjściowych** lub **Zwracanej wartości** do zwracania informacji o błędzie. Masz wszystkie dostępne dane na temat błędu, dlatego obsłuż go od razu. Po wystąpieniu błędu możesz pozwolić na zamknięcie programu. Zakończ go w ustrukturyzowany sposób, z wykorzystaniem instrukcji `assert`. Za pomocą tej instrukcji możesz dodatkowo udostępnić informacje diagnostyczne, tak jak w poniższym kodzie:

```
void someFunction()
{
    assert(checkPreconditions() && "Warunki wstępne nie są spełnione");
    mainFunctionality();
}
```

Ten fragment kodu sprawdza warunek w instrukcji `assert`. Jeśli ten warunek nie jest spełniony, tekst podany po prawej stronie w instrukcji `assert` zostanie wyświetlony w standardowym wyjściu, a program zakończy pracę. Akceptowalne jest też zamykanie programu w mniej ustrukturyzowany sposób, przez dostęp do wskaźnika bez sprawdzania, czy nie ma on wartości `NULL`. Wystarczy się upewnić, że program zakończy pracę w miejscu wystąpienia błędu.

Klauzula zabezpieczająca często jest dobrym miejscem na zamykanie programu w reakcji na błędy. Na przykład jeśli wiesz, że wystąpi błąd w kodzie (jednostka wywołująca przekazała wskaźnik `NULL`), możesz zamknąć program i zarejestrować informacje diagnostyczne zamiast zwracać informacje o błędzie do jednostki wywołującej. Nie zamykaj jednak programu przy każdym błędzie. Na przykład błędy czasu wykonania, takie jak niepoprawne dane wejściowe od użytkownika, z pewnością nie powinny prowadzić do zakończenia pracy aplikacji.

Jednostka wywołująca musi wiedzieć, jak działa funkcja, dlatego w API funkcji trzeba udokumentować scenariusze, w których zamyka ona program. Na przykład w dokumentacji funkcji trzeba określić, czy program zakończy pracę, jeśli do funkcji zostanie przekazany parametr w postaci wskaźnika `NULL`.

Reguła samuraja oczywiście nadaje się tylko dla niektórych błędów i dziedzin aplikacji. Nie chcesz przecież pozwolić programowi na zakończenie pracy po otrzymaniu nieoczekiwanych danych wejściowych od użytkownika. Jednak po wystąpieniu błędu programistycznego właściwym podejściem jest szybkie zareagowanie na problem i pozwolenie programowi na zamknięcie. Maksymalnie ułatwia to programistom wykrycie błędów.

Taka awaria nie musi być jednak widoczna dla użytkownika. Jeśli dany program jest tylko niekrytyczną częścią większej aplikacji, możesz pozwolić mu na zakończenie pracy. Jednak w kontekście tej aplikacji program może zostać dyskretnie zamknięty bez zakłócenia pracy reszty aplikacji lub użytkownika.



Instrukcje assert w udostępnianych programach wykonywalnych

W temacie instrukcji assert toczą się czasem dyskusje o to, czy pozostawić je aktywne tylko w programach wykonywalnych w trybie diagnostycznym, czy także w udostępnianym kodzie. Instrukcje assert można dezaktywować przez zdefiniowanie makra NDEBUG w kodzie przed dołączeniem pliku `assert.h` lub przez bezpośrednie zdefiniowanie tego makra w zestawie używanych narzędzi. Głównym argumentem za dezaktywacją instrukcji assert w udostępnianym kodzie jest to, że programista już wykrył błędy programistyczne związane z tymi instrukcjami w trakcie testowania programów w trybie diagnostycznym. Nie ma więc potrzeby ryzykować zamykania programów z powodu tych instrukcji w udostępnianym kodzie. Z kolei głównym argumentem za pozostawieniem aktywnych instrukcji assert w udostępnianym kodzie jest to, że można ich używać dla błędów krytycznych, których nie da się obsłużyć w kontrolowany sposób, a takie usterki nigdy nie powinny pozostać niezauważone, nawet w kodzie już używanym przez klientów.

Skutki

Błąd nie może pozostać niezauważony, ponieważ jest obsługiwany w miejscu jego wykrycia. Jednostka wywołująca nie musi wykrywać takiego błędu, dzięki czemu jej kod staje się prostszy. Jednak teraz jednostka wywołująca nie może wybrać sposobu reagowania na taki błąd.

W niektórych sytuacjach zamykanie aplikacji jest akceptowalne, ponieważ szybkie jej zakończenie jest lepsze niż późniejsze nieprzewidywalne zachowanie. Mimo to trzeba rozważyć, jak poinformować o takim błędzie użytkownika. Możliwe, że użytkownik zobaczy na ekranie komunikat o zamknięciu programu. Jednak w aplikacjach wbudowanych, które do interakcji ze środowiskiem używają czujników i serwowatorów, trzeba wykazać się większą starannością, rozważyć wpływ zamknięcia programu na środowisko i ustalić, czy jest on akceptowalny. W wielu takich scenariuszach aplikacja musi być bardziej odporna i jej proste zamknięcie jest nie do przyjęcia.

Zamknięcie programu i zarejestrowanie błędu w miejscu jego wystąpienia sprawia, że łatwiej jest znaleźć i rozwiązać problem, ponieważ nie jest on ukrywany. Dlatego w długim terminie stosowanie tego wzorca prowadzi do oprogramowania, które jest bardziej odporne na usterki i wolne od błędów.

Znane zastosowania

Oto przykłady ilustrujące zastosowania tego wzorca:

- Podobny wzorec, sugerujący dodawanie łańcucha znaków z informacjami diagnostycznymi do instrukcji `assert`, został nazwany **Kontekstem asercji** i opisany w książce *Patterns in C* Adama Tornhilla (Leanpub, 2014).
- W snifferze sieciowym Wireshark ten wzorec jest stosowany w wielu miejscach kodu. Na przykład w funkcji `register_capture_dissector` instrukcja `assert` jest używana do sprawdzania, czy zarejestrowany parser protokołu jest unikatowy.
- Instrukcje `assert` są używane w kodzie źródłowym projektu Git. Na przykład funkcje do zapisywania wartości skrótów SHA1 używają takich instrukcji do sprawdzania, czy poprawna jest ścieżka do pliku przeznaczonego na te wartości.
- Kod systemu OpenWrt odpowiedzialny za obsługę dużych liczb używa instrukcji `assert` do sprawdzania warunków wstępnych funkcji.
- Podobny wzorec o nazwie **Pozwól na awarię** został zaprezentowany przez Pekkę Alho i Jariego Rauhamäkiego w artykule *Patterns for Light-Weight Fault Tolerance and Decoupled Design in Distributed Control Systems* (<https://oreil.ly/x0tQW>). Ten wzorec jest przeznaczony dla systemów z rozproszoną kontrolą, a zachęca do zezwalania na zamykanie pojedynczych procesów bezpiecznych w razie uszkodzenia i późniejsze szybkie ich ponowne uruchamianie.
- Funkcja `strcpy` z biblioteki standardowej języka C nie sprawdza poprawności danych wejściowych od użytkownika. Jeśli przekażesz do tej funkcji wskaźnik `NULL`, nastąpi awaria.

Zastosowanie w przykładzie roboczym

Funkcja `parseFile` wygląda teraz znacznie lepiej. Zamiast zwracać kod błędu masz prostą instrukcję `assert`. Dzięki temu pokazany poniżej kod jest krótszy, a jednostka wywołująca nie musi sprawdzać **Zwracanej wartości**:

```
int parseFile(char* file_name)
{
    int return_value = ERROR;
    FILE* file_pointer = 0;
    char* buffer = 0;

    assert(file_name!=NULL && "Błędna nazwa pliku");
    if(file_pointer=fopen(file_name, "r"))
    {
        if(buffer=malloc(BUFFER_SIZE))
        {
            return_value = searchFileForKeywords(buffer, file_pointer);
            free(buffer);
        }
        fclose(file_pointer);
    }
    return return_value;
}
```

Choć instrukcje `if`, które nie wymagają porządkowania zasobów, zostały wyeliminowane, w kodzie wciąż znajdują się zagnieżdżone instrukcje `if` dla wszystkich zasobów wymagających zwolnienia. Ponadto na razie brakuje obsługi błędu związanego z niepowodzeniem wywołania `malloc`. Wszystkie te niedoskonałości można poprawić przez zastosowanie **Obsługi błędów z użyciem wywołania `goto`**.

Obsługa błędów z użyciem instrukcji `goto`

Kontekst

Masz funkcję, która zajmuje i zwalnia wiele zasobów. Możliwe, że podjęte zostały już próby ograniczenia złożoności przez zastosowanie wzorców **Klauzula zabezpieczająca**, **Podział funkcji** lub **Reguła samuraja**, jednak w kodzie nadal znajdują się głęboko zagnieżdżone instrukcje `if` związane z zajmowaniem zasobów. Możliwe nawet, że występują powtórzenia w kodzie do porządkowania zasobów.

Problem

Kod staje się mało czytelny i trudny w konserwacji, gdy pozyskuje i zwalnia wiele zasobów w różnych miejscach funkcji.

Taki kod staje się trudny w użytkowaniu, ponieważ zwykle każda operacja pozyskiwania zasobów może zakończyć się niepowodzeniem, a każda operacja porządkowania zasobów może zostać wywołana tylko w sytuacji, jeśli zasób został poprawnie zaalokowany. Aby zaimplementować to rozwiązanie, potrzebnych jest wiele instrukcji `if`. Jeżeli implementacja jest niskiej jakości, zagnieżdżone instrukcje `if` w jednej funkcji zmniejszają czytelność i utrudniają konserwację kodu.

Ponieważ musisz porządkować zasoby, zwracanie sterowania w środku funkcji w reakcji na problemy nie jest dobrym rozwiązaniem. Wynika to z tego, że wszystkie już zajęte zasoby trzeba uporządkować przed każdą instrukcją `return`. Powstaje więc wiele miejsc w kodzie, w których porządkowany jest ten sam zasób, a powielanie kodu do obsługi błędów i porządkowania zasobów jest niepożądane.

Rozwiązanie

Cały kod do porządkowania zasobów i obsługi błędów umieść na końcu funkcji. Jeśli zasobów nie da się pozyskać, użyj instrukcji `goto`, aby przeskoczyć do kodu porządkującego zasoby.

Zajmuj zasoby w kolejności, w jakiej są potrzebne, a na końcu funkcji porządkuj je w odwrotnej kolejności. Dla porządkowania zasobów używaj odrębnej etykiety, do której możesz przeskoczyć w każdej funkcji, gdzie jest to potrzebne. Wystarczy przeskoczyć do tej etykiety po wystąpieniu błędu lub po nieudanej próbie zajęcia zasobów. Unikaj jednak wielokrotnego wykonywania tej operacji i przeskakuj tylko do przodu, tak jak w poniższym kodzie:

```
void someFunction()
{
    if(!allocateResource())
```

```

{
    goto cleanup1;
}
if(!allocateResource2())
{
    goto cleanup2;
}
mainFunctionality();
cleanup2:
cleanupResource2();
cleanup1:
cleanupResource1();
}

```

Jeśli Twoje standardy pisania kodu nie dopuszczają używania instrukcji `goto`, możesz ją zasymulować za pomocą pętli `do{ ... }while(0);` umieszczonej wokół kodu. Po wystąpieniu błędu użyj instrukcji `break`, aby przeskoczyć poza pętlę, gdzie umieszczony jest kod obsługi błędów. Jednak takie obchodzenie zasad jest zwykle złym pomysłem, ponieważ jeśli zgodnie ze standardami niedozwolone jest używanie instrukcji `goto`, nie należy też ich symulować, aby nadal programować we własnym stylu. Wtedy zamiast instrukcji `goto` zastosuj **Zapisywanie informacji o porządkowaniu**.

Niezależnie od tych uwag stosowanie instrukcji `goto` może wskazywać na to, że funkcja jest zbyt skomplikowana. Wtedy lepszym rozwiązaniem może być jej podział, na przykład z wykorzystaniem **Obiektowej obsługi błędów**.



Instrukcja `goto` — hit czy kit?

Toczy się wiele dyskusji na temat tego, czy należy stosować instrukcję `goto`. Najbardziej znanym artykułem przeciwko używaniu tej instrukcji jest tekst Edsgera W. Dijkstry (<https://oreil.ly/yXkyq>), który argumentuje, że utrudnia ona dojrzanie przepływu sterowania w programie. Jest to prawda, jeśli instrukcja `goto` służy do przeskakiwania w programie tam i z powrotem w różne miejsca, jednak w C nie da się jej nadużywać w takim stopniu, jak w językach programowania, o jakich pisał Dijkstra. W C `goto` pozwala przeskakiwać w inne miejsca tylko w ramach funkcji.

Skutki

Funkcja zwraca teraz wartość w jednym miejscu, a główny przepływ sterowania w programie jest dobrze oddzielony od obsługi błędów i porządkowania zasobów. Żadne zagnieżdżone instrukcje `if` nie są już potrzebne do uzyskania tych efektów, ale nie każdy jest przyzwyczajony do czytania kodu z instrukcjami `goto`; część osób tego nie lubi.

Jeśli stosujesz instrukcje `goto`, musisz zachować ostrożność, ponieważ kuszące jest korzystanie z nich do wykonywania innych zadań niż obsługa błędów i porządkowanie zasobów, co prowadzi do powstawania nieczytelnego kodu. Ponadto trzeba postępować bardzo uważnie, aby powiązać odpowiednie funkcje porządkujące z właściwymi etykietami. Często pomyłką jest przypadkowe umieszczanie funkcji porządkujących przy nieodpowiednich etykietach.

Znane zastosowania

Oto przykładowe zastosowania tego wzorca:

- W kodzie jądra Linuksa do obsługi błędów używane jest przede wszystkim podejście wykorzystujące instrukcje goto. W książce *Linux Device Drivers* (<https://oreil.ly/linux-device-drivers>) Alessandro Rubini i Jonathan Corbet (O'Reilly, 2001) opisują obsługę błędów opartą na instrukcjach goto w programowaniu sterowników urządzeń dla Linuksa.
- W pozycji *The CERT C Coding Standard* (Addison-Wesley Professional, 2014) Robert C. Seacord sugeruje używanie instrukcji goto do obsługi błędów.
- Symulowanie instrukcji goto za pomocą pętli do-while jest opisane w portlandzkim repozytorium wzorców we wzorcu *Trywialna pętla do-while*.
- Instrukcja goto jest używana w kodzie biblioteki *OpenSSL*. Na przykład funkcje do obsługi certyfikatów X509 używają instrukcji goto do przeskakiwania do przodu do głównego kodu obsługi błędów.
- W kodzie narzędzia Wireshark instrukcje goto są używane do przeskakiwania z funkcji `main` do głównego kodu obsługi błędów umieszczonego na końcu tej funkcji.

Zastosowanie w przykładzie roboczym

Choć sporo osób zdecydowanie odradza korzystanie z instrukcji goto, obsługa błędów wygląda teraz lepiej niż w poprzednim przykładzie. W poniższym kodzie nie ma zagnieżdżonych instrukcji `if`, a kod porządkujący zasoby jest dobrze oddzielony od głównego przepływu sterowania w programie:

```
int parseFile(char* file_name)
{
    int return_value = ERROR;
    FILE* file_pointer = 0;
    char* buffer = 0;

    assert(file_name!=NULL && "Błędna nazwa pliku");
    if(!(file_pointer=fopen(file_name, "r")))
    {
        goto error_fileopen;
    }
    if(!(buffer=malloc(BUFFER_SIZE)))
    {
        goto error_malloc;
    }
    return_value = searchFileForKeywords(buffer, file_pointer);
    free(buffer);
error_malloc:
    fclose(file_pointer);
error_fileopen:
    return return_value;
}
```

Żałujemy jednak, że nie chcesz używać instrukcji goto lub obowiązujące Cię zasady programowania zabraniają korzystania z niej, jednak musisz w jakiś sposób porządkować zasoby. Dostępne są wtedy inne podejścia. Możesz na przykład użyć wzorca *Zapisywanie informacji o porządkowaniu*.

Zapisywanie informacji o porządkowaniu

Kontekst

Masz funkcję, która zajmuje wiele zasobów i je porządkuje. Możliwe, że stosujesz już wzorce *Klauzula zabezpieczająca*, *Podział funkcji* lub *Reguła samuraja*, aby ograniczyć złożoność kodu, jednak z powodu zajmowania zasobów nadal występują głęboko zagnieżdżone instrukcje `if`. Możliwe nawet, że kod do porządkowania zasobów się powtarza. Twoje standardy programowania nie zezwalają na zastosowanie *Obsługi błędów z życiem instrukcji goto* lub nie chcesz korzystać z tej instrukcji.

Problem

Trudno jest zapewnić czytelność i łatwą konserwację fragmentu kodu, jeśli pozyskuje on i zwalnia wiele zasobów, zwłaszcza gdy te zasoby są od siebie zależne.

Zadanie jest trudne, ponieważ zazwyczaj każda operacja pozyskiwania zasobów może się zakończyć niepowodzeniem, a kod do porządkowania zasobów należy wywoływać tylko po ich udanym zajęciu. Aby zaimplementować takie rozwiązanie, potrzeba wielu instrukcji `if`. Jeśli implementacja jest niskiej jakości, zagnieżdżone instrukcje `if` w jednej funkcji sprawiają, że kod jest mniej czytelny i trudny w konserwacji.

Ponieważ musisz porządkować zasoby, zwracanie sterowania w środku funkcji, gdy coś się nie powiedzie, jest złym pomysłem. Wynika to z tego, że wszystkie już zajęte zasoby trzeba uporządkować przed każdą instrukcją `return`. Tak więc w kodzie pojawia się wiele punktów, w których porządkowany jest ten sam zasób. Nie chcesz jednak powielać kodu obsługi błędów i porządkowania zasobów.

Rozwiązanie

Wywołuj funkcje zajmujące zasoby, dopóki działają one poprawnie, i zapisuj, które funkcje wymagają uporządkowania zasobów. Wywołuj funkcje porządkujące zasoby zgodnie z zarejestrowanymi informacjami.

W C potrzebny efekt można uzyskać za pomocą leniwego przetwarzania instrukcji `if`. Wystarczy wywoływać sekwencję funkcji w jednej instrukcji `if`, dopóki te wywołania kończą się sukcesem. Dla każdego takiego wywołania zapisz pozyskany zasób w zmiennej. W ciele takiej instrukcji `if` umieść kod operujący zasobami, a kod do porządkowania po instrukcji `if` wykonuj tylko wtedy, jeśli dany zasób został poprawnie pozyskany. Oto przykładowy kod tego typu:

```
void someFunction()
{
    if((r1=allocateResource1()) && (r2=allocateResource2()))
    {
        mainFunctionality();
    }
    if(r1) ❶
    {
        cleanupResource1();
    }
}
```

```

    }
    if(r2) ❶
    {
        cleanupResource2();
    }
}

```

- ❶ Aby zwiększyć czytelność kodu, możesz też umieścić te testy w funkcjach porządkujących zasoby. Jest to dobre rozwiązanie, jeśli i tak musisz przekazywać zmienne powiązane z zasobami do funkcji odpowiedzialnej za porządkowanie.

Skutki

Nie ma już zagnieżdżonych instrukcji `if`, a porządkowanie zasobów odbywa się w jednym centralnym punkcie na końcu funkcji. Dzięki temu kod jest znacznie czytelniejszy, ponieważ główna logika programu nie jest już zakrywana przez kod obsługi błędów.

Funkcja jest też czytelniejsza dzięki temu, że ma jeden punkt wyjścia. Jednak konieczność używania wielu zmiennych do śledzenia, które zasoby zostały z powodzeniem zaalokowane, komplikuje kod. Możliwe, że **Zagregowana instancja** pomoże ustrukturyzować zmienne reprezentujące zasoby.

Jeśli zajmowanych jest wiele zmiennych, to wiele funkcji jest wywoływanych w jednej instrukcji `if`. Z tego powodu instrukcja `if` jest mało czytelna i bardzo trudna w konserwacji. Jeśli więc zajmujesz wiele zasobów, znacznie lepszym rozwiązaniem jest zastosowanie **Obiektowej obsługi błędów**.

Innym powodem do użycia **Obiektowej obsługi błędów** jest to, że przedstawiony kod nadal jest skomplikowany, ponieważ w jednej funkcji znajduje się główna logika, a także kod do alokowania i porządkowania zasobów. Tak więc jedna funkcja ma wiele zadań.

Znane zastosowania

Oto przykładowe zastosowania tego wzorca:

- W portlandzkim repozytorium wzorców jest przedstawione podobne rozwiązanie, w którym każda z wywoływanych funkcji rejestruje kod porządkowania zasobów na liście wywołań zwrotnych. W momencie porządkowania zasobów wywoływane są wszystkie funkcje z tej listy.
- W funkcji `dh_key2buf` w bibliotece *OpenSSL* jest stosowane leniwe przetwarzanie w instrukcji `if`, aby rejestrować zaalokowane bajty, które są później zwalniane.
- W funkcji `cap_open_socket` w snifferze sieciowym Wireshark stosowane jest leniwe przetwarzanie w instrukcji `if`, a zasoby zaalokowane w tej instrukcji są zapisywane w zmiennych. W momencie porządkowania zasobów ma miejsce sprawdzanie tych zmiennych, a jeśli alokacja zakończyła się powodzeniem, zasoby są porządkowane.
- Funkcja `nvrn_commit` z kodu źródłowego systemu OpenWrt alokuje zasoby w instrukcji `if` i zapisuje je w zmiennej w tej samej instrukcji.

Zastosowanie w przykładzie roboczym

Teraz zamiast instrukcji `goto` i zagnieżdżonych instrukcji `if` występuje jedna instrukcja `if`. Zaletą rezygnacji z instrukcji `goto` w tym kodzie jest to, że obsługa błędów jest dobrze oddzielona od logiki głównego programu:

```
int parseFile(char* file_name)
{
    int return_value = ERROR;
    FILE* file_pointer = 0;
    char* buffer = 0;

    assert(file_name!=NULL && "Błędna nazwa pliku");
    if((file_pointer=fopen(file_name, "r")) &&
        (buffer=malloc(BUFFER_SIZE)))
    {
        return_value = searchFileForKeywords(buffer, file_pointer);
    }
    if(file_pointer)
    {
        fclose(file_pointer);
    }
    if(buffer)
    {
        free(buffer);
    }
    return return_value;
}
```

Mimo to kod nie wygląda dobrze. Jedna pokazana tu funkcja wykonuje wiele zadań: alokuje zasoby, dealokuje zasoby, przetwarza plik i obsługuje błędy. Te zadania należy rozdzielić między różne funkcje za pomocą *Obiektowej obsługi błędów*.

Obiektowa obsługa błędów

Kontekst

Masz funkcję, która pozyskuje i porządkuje wiele zasobów. Możliwe, że w ramach próby ograniczenia złożoności stosujesz już *Klauzulę zabezpieczającą*, *Podział funkcji* lub *Regułę samuraja*, jednak z powodu pozyskiwania zasobów w kodzie nadal występuje głęboko zagnieżdżona instrukcja `if`. Możliwe nawet, że kod do porządkowania zasobów się powtarza. Niewykluczone jednak, że udało Ci się już wyeliminować zagnieżdżone instrukcje `if` za pomocą *Obsługi błędów z użyciem instrukcji `goto`* lub *Zapisywania informacji o porządkowaniu*.

Problem

Gdy jedna funkcja wykonuje wiele zadań, na przykład pozyskuje zasoby, porządkuje je i z nich korzysta, kod jest trudny do implementacji, konserwacji i testowania, a także mało czytelny.

Zadanie jest trudne, ponieważ przeważnie każda operacja pozyskiwania zasobów może się zakończyć niepowodzeniem, a kod do porządkowania zasobów należy wywoływać tylko po ich udanym zajęciu. Aby zaimplementować takie rozwiązanie, potrzeba wielu instrukcji i f. Jeśli implementacja jest niskiej jakości, zagnieżdżone instrukcje i f w jednej funkcji sprawiają, że kod jest mniej czytelny i trudny w konserwacji.

Ponieważ musisz porządkować zasoby, zwracanie sterowania w środku funkcji, jeśli coś się nie powiedzie, jest złym pomysłem. Wynika to z tego, że wszystkie już zajęte zasoby trzeba uporządkować przed każdą instrukcją return. Tak więc w kodzie pojawia się wiele punktów, w których porządkowany jest ten sam zasób. Nie chcesz jednak powielać kodu obsługi błędów i porządkowania zasobów.

Nawet jeśli już stosujesz *Zapisywanie informacji o porządkowaniu* lub *Obsługę błędów za pomocą instrukcji goto*, funkcja nadal jest mało czytelna, ponieważ wykonuje różne zadania. Odpowiada za pozyskiwanie wielu zasobów, obsługę błędów i porządkowanie licznych zasobów. Jednak funkcja powinna mieć tylko jedno zadanie.

Rozwiązanie

Umieść kod inicjalizacji i porządkowania w odrębnych funkcjach, podobnych do konstruktorów i destruktorów z programowania obiektowego.

W funkcji głównej wywołuj jedną funkcję, która pozyskuje wszystkie zasoby, jedną funkcję operującą tymi zasobami i jedną funkcję porządkującą te zasoby.

Jeśli pozyskiwane zasoby nie są globalne, trzeba je przekazywać między funkcjami. Gdy masz wiele zasobów, możesz przekazywać do funkcji *Zagregowaną instancję* zawierającą wszystkie zasoby. Jeżeli chcesz ukryć zasoby przed jednostką wywołującą, możesz użyć *Uchwytu* do przekazywania informacji o zasobach między funkcjami.

Jeśli alokacja zasobów zakończy się niepowodzeniem, zapisz informacje o tym w zmiennej (użyj na przykład wskaźnika NULL po nieudanej alokacji pamięci). W trakcie korzystania z zasobów lub ich porządkowania należy najpierw sprawdzić, czy są one poprawne. Ten test należy wykonywać nie w głównej funkcji, lecz w wywoływanych funkcjach. Dzięki temu kod głównej funkcji będzie znacznie czytelniejszy:

```
void someFunction()
{
    allocateResources();
    mainFunctionality();
    cleanupResources();
}
```

Skutki

Teraz funkcja jest czytelna. Choć wymaga alokacji i porządkowania wielu zasobów, a także wykonywania operacji na tych zasobach, te różne zadania są wyraźnie rozdzielone między różne funkcje.

Używanie podobnych do obiektów instancji przekazywanych do funkcji to tak zwany „obiekto-
towy” styl programowania. W tym podejściu programowanie proceduralne jest bardziej zbliżone
do programowania obiektowego, dlatego kod pisany w tym stylu także wygląda bardziej znajomo
dla programistów przyzwyczajonych do modelu obiektowego.

W głównej funkcji nie ma już powodu do tworzenia wielu instrukcji `return`, ponieważ nie zawiera już
ona zagnieżdżonych instrukcji `if` z kodem do alokowania i porządkowania zasobów. Jednak
oczywiście nie wiąże się to z całkowitym usunięciem tego kodu. Znajduje się on w całości w od-
rębnych funkcjach, przy czym nie jest już wymieszany z operacjami wykonywanymi na zasobach.

Zamiast jednej funkcji masz ich teraz kilka. Choć może to mieć negatywny wpływ na wydajność,
zwykle nie ma to istotnego znaczenia. Zmiany w wydajności są niewielkie i w większości aplikacji
nie jest to istotne.

Znane zastosowania

Oto przykładowe zastosowania tego wzorca:

- Ta forma porządkowania zasobów jest używana w programowaniu obiektowym, gdzie konstruk-
tory i destruktory są wywoływane automatycznie.
- Ten wzorec jest stosowany w bibliotece *OpenSSL*. Na przykład do alokowania i porządko-
wania buforów służą funkcje `BUF_MEM_new` i `BUF_MEM_free` wywoływane w kodzie związanym
z obsługą buforów.
- Funkcja `show_help` w kodzie źródłowym systemu *OpenWrt* wyświetla informacje o pomocy
w menu kontekstowym. Funkcja `show_help` wywołuje funkcję inicjalizującą tworzącą struk-
turę, następnie operuje tą strukturą, a na końcu wywołuje funkcję zwalnającą tę strukturę.
- Funkcja `cmd__windows_named_pipe` z projektu *Git* używa *Uchwytu* do utworzenia potoku,
następnie operuje tym potokiem i wywołuje odrębną funkcję do zwolnienia potoku.

Zastosowanie w przykładzie roboczym

Ostatecznie otrzymujemy pokazany niżej kod, w którym funkcja `parseFile` wywołuje inne funkcje
w celu utworzenia i zwolnienia instancji parsera:

```
typedef struct
{
    FILE* file_pointer;
    char* buffer;
}FileParser;

int parseFile(char* file_name)
{
    int return_value;
    FileParser* parser = createParser(file_name);
    return_value = searchFileForKeywords(parser);
    cleanupParser(parser);
    return return_value;
}
```

```

int searchFileForKeywords(FileParser* parser)
{
    if(parser == NULL)
    {
        return ERROR;
    }
    while(fgets(parser->buffer, BUFFER_SIZE, parser->file_pointer)!=NULL)
    {
        if(strcmp("KEYWORD_ONE\n", parser->buffer)==0)
        {
            return KEYWORD_ONE_FOUND_FIRST;
        }
        if(strcmp("KEYWORD_TWO\n", parser->buffer)==0)
        {
            return KEYWORD_TWO_FOUND_FIRST;
        }
    }
    return NO_KEYWORD_FOUND;
}

FileParser* createParser(char* file_name)
{
    assert(file_name!=NULL && "Błędna nazwa pliku");
    FileParser* parser = malloc(sizeof(FileParser));
    if(parser)
    {
        parser->file_pointer=fopen(file_name, "r");
        parser->buffer = malloc(BUFFER_SIZE);
        if(!parser->file_pointer || !parser->buffer)
        {
            cleanupParser(parser);
            return NULL;
        }
    }
    return parser;
}

void cleanupParser(FileParser* parser)
{
    if(parser)
    {
        if(parser->buffer)
        {
            free(parser->buffer);
        }
        if(parser->file_pointer)
        {
            fclose(parser->file_pointer);
        }
        free(parser);
    }
}

```

W kodzie nie ma już kaskadowych instrukcji `if` w głównej logice programu. Dzięki temu funkcja `parseFile` jest teraz dużo czytelniejsza, a także łatwiejsza w debugowaniu i konserwacji. Funkcja główna nie odpowiada już za alokację i dealokację zasobów ani obsługę błędów. Tymi operacjami zajmują się odrębne funkcje, dlatego każda funkcja ma jedno zadanie.

Zauważ, jak elegancka jest ostateczna wersja kodu w porównaniu z pierwszym przykładem. Zastosowane wzorce pomogły krok po kroku zwiększyć czytelność kodu i ułatwić jego konserwację. W każdym kroku usuwałem kaskadę zagnieżdżonych instrukcji `if` i ulepszałem obsługę błędów.

Podsumowanie

W tym rozdziale pokazałem, jak obsługiwać błędy w C. **Podział funkcji** pozwala rozdzielić funkcję na mniejsze fragmenty, aby ułatwić obsługę błędów w ich ramach. **Klauzula zabezpieczająca** funkcji sprawdza ich warunki wstępne i natychmiast zwraca sterowanie, jeśli nie są one spełnione. Dzięki temu pozostałe fragmenty funkcji mają mniej zadań związanych z obsługą błędów. Zamiast zwracać sterowanie z funkcji można też zamknąć program, co jest zgodne z **Regułą samuraja**. Jeśli chodzi o bardziej zaawansowaną obsługę błędów (zwłaszcza w połączeniu z pozyskiwaniem i zwalnianiem zasobów), dostępnych jest kilka możliwości. **Obsługa błędów z użyciem instrukcji `goto`** umożliwia przeskakiwanie w funkcji naprzód, do sekcji obsługi błędów. We wzorcu **Zapisywanie informacji o porządkowaniu** zamiast przeskakiwania należy zapisać, które zasoby wymagają zwolnienia, i wykonać tę operację na końcu funkcji. **Obiektowa obsługa błędów** to technika pozyskiwania zasobów działająca podobnie jak w programowaniu obiektowym. W tym wzorcu używane są odrębne funkcje do inicjalizowania i zwalniania zasobów, podobne w działaniu do konstruktorów i destruktorów.

Dzięki znajomości wzorców obsługi błędów masz teraz umiejętności potrzebne do pisania niewielkich programów, które obsługują błędy w sposób gwarantujący, że kod pozostanie łatwy w konserwacji.

Dalsza lektura

Jeśli szukasz dodatkowych informacji, oto materiały, które pomogą Ci rozwinąć wiedzę z zakresu obsługi błędów.

- Portlandzkie repozytorium wzorców (<https://oreil.ly/qFLdA>) obejmuje opis wielu wzorców oraz omówienie obsługi błędów i innych zagadnień. Większość wzorców obsługi błędów dotyczy obsługi wyjątków i stosowania asercji, jednak znajdziesz także wzorce związane z językiem C.
- Kompleksowe omówienie obsługi błędów znajdziesz w pracy magisterskiej *Error Handling in Structured and Object-Oriented Programming Languages* Thomasa Aglassingera (University of Oulu, 1999). Ten tekst zawiera opis powstawania błędów różnego rodzaju, omówienie mechanizmów obsługi błędów w językach C, Basic, Java i Eiffel, a także przegląd dobrych praktyk obsługi błędów w tych językach, na przykład odwracania kolejności zwalniania zasobów w porównaniu z porządkiem ich alokowania. W tej pracy autor wspomina też o kilku zewnętrznych rozwiązaniach w postaci bibliotek języka C udostępniających zaawansowane funkcje obsługi błędów, na przykład obsługę wyjątków za pomocą instrukcji `setjmp` i `longjmp`.

- W artykule *Error Handling for Business Information Systems* (<https://oreil.ly/bQnfx>) Klausa Renzela opisanych jest 15 obiektowych wzorców obsługi błędów dostosowanych do biznesowych systemów informatycznych. Większość tych wzorców można zastosować także w kodzie nie-obiektowym. Prezentowane wzorce dotyczą wykrywania, rejestrowania i obsługi błędów.
- W książce *Patterns in C* Adama Tornhilla (Leanpub, 2014) znajdziesz fragmenty kodu w C z implementacją wybranych wzorców projektowych „bandy czworga”. Ta pozycja zawiera też dobre praktyki w postaci wzorców języka C. Niektóre z nich dotyczą obsługi błędów.
- W artykułach *Patterns for Generation, Handling and Management of Errors* i *More Patterns for the Generation, Handling and Management of Errors* Andy’ego Longshawa i Eoina Woodsa (<https://oreil.ly/7Yj8h>) znajdziesz omówienie różnych wzorców rejestrowania i obsługi błędów. Większość tych wzorców dotyczy obsługi błędów z wykorzystaniem wyjątków.

Co dalej?

W następnym rozdziale zobaczysz, jak obsługiwać błędy w większych programach, które za pomocą interfejsów przekazują informacje o problemach do innych funkcji. Z tych wzorców dowiesz się, jakiego rodzaju informacje o błędach zwracać i w jaki sposób to robić.

PROGRAM PARTNERSKI

— GRUPY 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
Helion 

Ta książka pokaże początkującym, jak tworzyć w C produkty klasy przemysłowej!

Robert Hanmer, architekt oprogramowania

Język C mimo upływu lat wciąż wydaje się niezastąpiony. Umożliwia pisanie zwięzłego kodu, który działa szybko mimo niewielkich zasobów sprzętowych. Choć wielu programistów używa C, trudno jest znaleźć eksperckie wskazówki dotyczące programowania w tym języku. Tymczasem w profesjonalnych zastosowaniach podjęcie optymalnych decyzji projektowych warunkuje uzyskanie wysokiej jakości gotowego kodu.

W tym poradniku, skierowanym do początkujących i doświadczonych programistów języka C, zawarto wiele informacji o podejmowaniu decyzji projektowych, pokazano też krok po kroku, jak wpływają one na tworzenie złożonego oprogramowania. Znajdziesz tu odpowiedzi na trudne pytania o projektowanie struktury programów w C, obsługę błędów czy tworzenie elastycznych interfejsów. Liczne wskazówki i przykłady ułatwią Ci przekładanie wiedzy projektowej na działające implementacje. Druga część książki stanowi omówienie zastosowania licznych wzorców projektowych z języka C do tworzenia większych aplikacji. Dowiesz się, jakie wzorce stosuje się w konkretnych sytuacjach, a także w jaki sposób mogą ułatwić Ci pracę dzięki wskazywaniu dobrych decyzji projektowych.

W książce omówiono wzorce dotyczące:

- obsługi błędów i komunikatów o błędach
- zarządzania pamięcią
- elastycznych API i interfejsów iteratorów
- organizowania plików w programach modułowych
- ucieczki z piekła instrukcji `#ifdef`

Dr Christopher Preschern jest austriackim programistą w firmie ABB; pracuje z językiem C, pisze oprogramowanie klasy przemysłowej. Jest też wykładowcą na Graz University of Technology. Często organizuje konferencje branżowe i podejmuje inicjatywy związane z tworzeniem wzorców projektowych.

To świetna pozycja, dzięki której Twój kod będzie bardziej przejrzysty i łatwiejszy w konserwacji

David Griffiths, autor książki „C. Rusz głową!”



KOD KORZYŚCI
Sięgnij po więcej! ▶





helion.pl



HELION SA
ul. Kościuski 1c
44-100 Gliwice
tel.: 32 230 98 63
helion@helion.pl

ISBN 978-83-8322-722-1



9 788383 227221

Cena: 67,00 zł