



Czysty kod w C++17

Oprogramowanie łatwe w utrzymaniu

—

Stephan Roth

Tytuł oryginału: Clean C++: Sustainable Software Development Patterns and Best Practices with C++ 17

Tłumaczenie: Tomasz Walczak

ISBN: 978-83-283-4340-5

Original edition copyright © 2017 by Stephan Roth.
All rights reserved.

Polish edition copyright © 2018 by HELION SA.
All rights reserved.

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

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

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

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

Wydawnictwo HELION
ul. Kościuszki 1c, 44-100 GLIWICE
tel. 32 231 22 19, 32 230 98 63
e-mail: helion@helion.pl
WWW: <http://helion.pl> (księgarnia internetowa, katalog książek)

Pliki z przykładami omawianymi w książce można znaleźć pod adresem:
<ftp://ftp.helion.pl/przyklady/czkc17.zip>

Drogi Czytelniku!
Jeżeli chcesz ocenić tę książkę, zajrzyj pod adres
<http://helion.pl/user/opinie/czkc17>
Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

Printed in Poland.

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

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

Spis treści

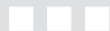
	O autorze	9
	O recenzencie technicznym	11
	Podziękowania	13
Rozdział 1	Wprowadzenie	15
	Entropia oprogramowania	16
	Czysty kod	17
	Dlaczego C++?	18
	C++11 — początek nowej ery	18
	Dla kogo przeznaczona jest ta książka?	19
	Konwencje stosowane w tej książce	19
	Ramki	20
	Uwagi, wskazówki i ostrzeżenia	20
	Przykładowy kod	20
	Witryna książki i repozytorium z kodem źródłowym	21
	Diagramy UML-a	21
Rozdział 2	Tworzenie siatki bezpieczeństwa	23
	Konieczność przeprowadzania testów	23
	Wprowadzenie do testów	25
	Testy jednostkowe	26
	A co z kontrolą jakości?	28
	Reguły tworzenia dobrych testów jednostkowych	29
	Jakość kodu testów	29
	Nazwy testów jednostkowych	29
	Niezależność testów jednostkowych	31
	Jedna asercja na test	31
	Niezależne inicjowanie środowisk testów jednostkowych	32
	Pomijanie testów getterów i setterów	32
	Pomijanie testów kodu innych programistów	32
	Pomijanie testów zewnętrznych systemów	33
	A co zrobić z bazą danych?	33

	Nie łącz kodu testów z kodem produkcyjnym	33
	Testy muszą działać szybko	36
	Zaślepki	36
Rozdział 3	Postępuj zgodnie z zasadami	39
	Czym są zasady?	39
	Zachowaj prostotę, głupku (KISS)	40
	Nie będziesz tego potrzebować (YAGNI)	40
	Nie powtarzaj się (DRY)	41
	Ukrywanie informacji	41
	Wysoka spójność	44
	Luźne powiązanie	46
	Nie przesadzaj z optymalizacją	49
	Zasada minimalizowania zaskoczenia	50
	Reguła harcerza	50
Rozdział 4	Podstawy czystego C++	53
	Dobre nazwy	54
	Nazwy powinny być czytywne	55
	Stosuj nazwy z dziedziny	56
	Dobieraj nazwy na odpowiednim poziomie abstrakcji	57
	Unikaj nadmiarowości, gdy wymyślasz nazwę	58
	Unikaj zagadkowych skrótów	58
	Unikaj notacji węgierskiej i przedrostków	59
	Unikaj używania tej samej nazwy do różnych celów	60
	Komentarze	60
	Niech kod opowiada historię	60
	Nie komentuj oczywistych rzeczy	61
	Nie dezaktywuj kodu za pomocą komentarzy	61
	Nie pisz komentarzy blokowych	62
	Rzadkie scenariusze, w których komentarze są przydatne	64
	Funkcje	67
	Jedna rzecz — nie więcej!	70
	Twórz małe funkcje	70
	Nazwy funkcji	71
	Stosuj nazwy opisujące intencje	72
	Argumenty i zwracane wartości	72
	Liczba argumentów	73
	Projekty C++ w dawnym stylu specyficznym dla C	82
	Przedkładaj łańcuchy znaków i strumienie z C++ nad dawne łańcuchy char* w stylu języka C	82
	Unikaj instrukcji printf(), sprintf(), gets() itd.	84
	Przedkładaj kontenery z biblioteki standardowej nad proste tablice w stylu języka C	87
	Używanie rzutowania z języka C++ zamiast dawnego rzutowania w stylu języka C	89
	Unikaj makr	90
Rozdział 5	Zaawansowane aspekty współczesnego C++	93
	Zarządzanie zasobami	93
	Idiom RAII	95

Inteligentne wskaźniki	95
Unikanie bezpośrednich wywołań new i delete	100
Zarządzanie niezależnymi zasobami	101
Warto się czasem gdzieś przenieść	102
Czym jest semantyka przenoszenia?	102
Czym są l-wartości i r-wartości?	103
Referencje do r-wartości	104
Nie wymuszaj wszędzie semantyki przenoszenia	106
Reguła zera	106
Kompilator to Twój współpracownik	110
Automatyczna dedukcja typów	110
Obliczenia na etapie kompilacji	113
Szablony zmiennych	115
Nie dopuszczaj do niezdefiniowanych skutków	116
Programowanie z użyciem typów semantycznych	117
Poznaj używane biblioteki	123
Korzystaj z pliku nagłówkowego <algorithm>	123
Korzystaj z biblioteki Boost	128
Inne biblioteki, które powinieneś znać	129
Prawidłowa obsługa wyjątków i błędów	130
Lepiej zapobiegać niż leczyć	130
Wyjątek jest wyjątkiem — dosłownie	134
Jeśli nie możesz przywrócić stanu, szybko zamknij program	135
Definiuj specyficzne typy wyjątków	135
Zgłaszanie przez wartość i przechwytywanie za pomocą stałej referencji	137
Zwracaj uwagę na właściwą kolejność klauzul catch	137
Rozdział 6 Podjęcie obiektowe	139
Myślenie obiektowe	140
Abstrakcja — klucz do opanowania złożoności	141
Zasady poprawnego projektowania klas	141
Twórz niewielkie klasy	141
Zasada jednej odpowiedzialności	142
Zasada otwarte – zamknięte	143
Zasada podstawiania Liskov	144
Zasada podziału interfejsu	154
Zasada zależności acyklicznych	156
Zasada odwracania zależności	158
Nie rozmawiaj z nieznanymi (prawo Demeter)	162
Unikaj „anemicznych” klas	166
Mów zamiast pytać	167
Unikaj statycznych składowych klasy	169
Rozdział 7 Programowanie funkcyjne	171
Czym jest programowanie funkcyjne?	172
Czym jest funkcja?	173
Funkcje czyste i „nieczyste”	174

	Programowanie funkcyjne w nowoczesnym C++	175
	Programowanie funkcyjne z użyciem szablonów języka C++	175
	Obiekty podobne do funkcji (funktory)	177
	Mechanizm wiązania i nakładki na funkcje	183
	Wyrażenia lambda	185
	Generyczne wyrażenia lambda (C++14)	187
	Funkcje wyższego poziomu	187
	Mapowanie, filtrowanie i redukcja	189
	Czysty kod w programowaniu funkcyjnym	192
Rozdział 8	Programowanie sterowane testami	195
	Wady zwykłych dawnych testów jednostkowych	196
	Podejście TDD jako rewolucja	197
	Proces pracy w TDD	197
	TDD na przykładzie — kata dotyczące liczb rzymskich	200
	Zalety TDD	216
	Kiedy nie stosować TDD?	217
Rozdział 9	Wzorce projektowe i idiomy	219
	Zasady projektowe a wzorce projektowe	220
	Wybrane wzorce i sytuacje, w których warto je stosować	220
	Wstrzykiwanie zależności	221
	Adapter	231
	Strategia	233
	Polecenie	237
	Procesor poleceń	240
	Kompozyt	242
	Obserwator	245
	Fabryka	250
	Fasada	252
	Klasa Money	253
	Obiekt reprezentujący specjalny przypadek (obiekt NULL)	256
	Czym jest idiom?	260
	Przydatne idiomy języka C++	260
Dodatek A	Krótki przewodnik po UML-u	271
	Diagramy klas	271
	Klasa	271
	Interfejs	273
	Asocjacja	275
	Generalizacja	277
	Zależność	278
	Komponenty	279
	Stereotypy	279
	Bibliografia	281
	Skorowidz	285

ROZDZIAŁ 3



Postępuj zgodnie z zasadami

Doradzam studentom, aby więcej uwagi poświęcali podstawowym zasadom niż najnowszym technologiom. Dana technologia stanie się przestarzała, zanim ukończą studia. Podstawowe zasady będą zawsze aktualne.

— David L. Parnas

W tym rozdziale przedstawiam najważniejsze i podstawowe zasady poprawnego projektowania oraz budowania oprogramowania. Wyjątkową cechą tych zasad jest to, że nie są powiązane z konkretnymi paradygmatami ani językami programowania. Niektóre z nich nie są nawet ograniczone do rozwoju oprogramowania. Na przykład omawiana tu zasada „zachowaj prostotę, głupku” ma zastosowania w wielu obszarach życia. Niezłym pomysłem jest używanie możliwie prostych rozwiązań we wszystkich obszarach życia, nie tylko w zakresie rozwoju oprogramowania.

Oznacza to, że nie powinieneś raz nauczyć się opisanych tu zasad, a później je zapomnieć. Przedstawione rady należy sobie przyswoić. Są one tak ważne, że w idealnym scenariuszu powinny stać się drugą naturą każdego programisty. Ponadto wiele bardziej specyficznych reguł, które omawiam w dalszej części książki, jest opartych na tych podstawowych zasadach.

Czym są zasady?

W tej książce zetkniesz się z różnymi zasadami tworzenia lepszego kodu w języku C++ i dobrze zaprojektowanego oprogramowania. Czym jednak są zasady?

Wiele osób ma zasady, którymi kieruje się w życiu. Na przykład jeśli z różnych przyczyn jesteś przeciwny jedzeniu mięsa, jest to Twoja zasada. Jeżeli chcesz chronić swoje dziecko, przekazujesz mu zasady pomagające w podejmowaniu właściwych decyzji, np.: „Bądź ostrożny i nie rozmawiaj z nieznajomymi!”. Na podstawie takich zasad dziecko może wywnioskować, jakie zachowanie będzie właściwe w konkretnych sytuacjach.

Zasady to pewnego rodzaju reguły, przekonania lub idee kierujące Twoim postępowaniem. Są one często bezpośrednio powiązane z systemem wartości. Na przykład nikomu nie trzeba mówić, że kanibalizm jest czymś złym, ponieważ z natury traktujemy ludzkie życie jako coś wartościowego. Innym przykładem jest *Manifest programowania zwinnego* [Beck01], zawierający 12 zasad kierujących zespołami projektowymi w trakcie realizowania projektów zgodnie z podejściem zwinnym.

Zasady nie są nieprzekraczalnymi prawami. Nie są wykute w skale. W programowaniu trzeba czasem świadomie łamać zasady. Jeśli masz ku temu bardzo dobre powody, możesz to zrobić, jednak zachowaj przy tym dużą ostrożność. Przekraczanie zasad powinno być wyjątkiem.

Niektóre z opisanych tu podstawowych zasad są w różnych dalszych miejscach książki omówione w bardziej szczegółowy sposób.

Zachowaj prostotę, głupku (KISS)

Wszystko powinno być tak proste, jak to tylko możliwe, ale nie prostsze.

— Albert Einstein, fizyka teoretyczny, 1879 – 1955

Zasada „zachowaj prostotę, głupku” jest też nazywana regułą KISS (ang. *keep it simple, stupid* lub *keep it simple and stupid* to dwa najczęstsze rozwinięcia tego akronimu, choć istnieją też inne). W programowaniu ekstremalnym ta zasada przyjmuje postać praktyki „zrób najprostszą rzecz, jaka zadziała”. Zgodnie z zasadą KISS głównym celem w rozwoju oprogramowania powinna być prostota, natomiast należy unikać nadmiernej złożoności.

Uważam, że KISS to jedna z zasad, o których deweloperzy zwykle zapominają w trakcie rozwoju oprogramowania. Deweloperzy mają skłonność do pisania kodu w wyrafinowany sposób i komplikowania go bardziej niż to konieczne. Mam świadomość, że wszyscy posiadamy wysokie umiejętności i motywację oraz wiemy wszystko na temat wzorców projektowych i architektonicznych, platform, technologii, narzędzi oraz innych ciekawych i wymyślnych rzeczy. Budowanie ciekawego oprogramowania nie jest tylko pracą, jaką wykonujemy od 8:00 do 16:00 — to także nasza misja, a dzięki pracy się realizujemy.

Musisz jednak pamiętać, że każdy system oprogramowania ma naturalną złożoność, która sama w sobie sprawia trudności. Złożone problemy oczywiście często wymagają skomplikowanego kodu. Naturalnej złożoności nie da się zmniejszyć. Występuje ona dlatego, że system musi spełniać określone wymogi. Jednak dodawanie do niej niepotrzebnej, samodzielnie spowodowanej złożoności to poważny błąd. Dlatego radzę, aby nie stosować wszystkich wymyślnych cech języka lub wzorców projektowych tylko dlatego, że potrafisz to zrobić. Z drugiej strony, nie należy przesadzać z upraszczaniem. Jeśli w bloku `switch-case` potrzebnych jest dziesięć opcji, należy się z tym pogodzić.

Staraj się, aby Twój kod był tak prosty, jak to możliwe. Oczywiście jeśli priorytetowo traktowane są wymogi dotyczące elastyczności i rozszerzalności, musisz zwiększyć złożoność, aby je zrealizować. Możesz np. zastosować znany wzorzec strategii (zob. poświęcony wzorcom projektowym rozdział 9.), aby dodać do kodu dający swobodę punkt modyfikacji, jeśli to konieczne. Zachowaj jednak ostrożność i zwiększaj złożoność tylko w takim stopniu, aby ułatwiać sobie pracę.

Koncentracja na prostocie to prawdopodobnie jedna z najtrudniejszych rzeczy dla programisty. Człowiek uczy się tego przez całe życie.

— Adrian Bolboaca (@adibolb), 3 kwietnia 2014 r., Twitter

Nie będziesz tego potrzebować (YAGNI)

Zawsze implementuj rzeczy, gdy naprawdę ich potrzebujesz — nigdy wtedy, gdy tylko przewidujesz, że mogą ci się przydać.

— Ron Jeffries, *You're NOT gonna need it!* [Jeffries98]

Ta zasada jest ściśle powiązana z wcześniej opisaną regułą KISS. „Nie będziesz tego potrzebować” (ang. *You Aren't Gonna Need It!* lub *You Ain't Gonna Need It!* — YAGNI) oznacza wypowiedzenie wojny

spekulatywnemu uogólnianiu i przekombinowaniu. Zgodnie z tą regułą nie należy pisać kodu, który w danym momencie jest niepotrzebny, ale może przydać się w przyszłości.

Prawdopodobnie każdy programista zna kuszące impulsy pojawiające się w codziennej pracy: „Może później się to przyda...” lub „Będziemy potrzebować...”. **Nie, nie będziesz tego potrzebować!** Powinieneś konsekwentnie opierać się pokusie tworzenia czegoś, co później może się przydać. Możliwe, że okaże się to zbędne. Implementowanie niepotrzebnej rzeczy oznacza marnotrawstwo czasu, a kod staje się wtedy bardziej skomplikowany, niż powinien być! Oczywiście stanowi to także naruszenie zasady KISS. Jeszcze gorsze konsekwencje pojawiają się, gdy pisane z myślą o przyszłości fragmenty kodu zawierają błędy i powodują poważne problemy. Oto moja rada: **zaufaj potędze refaktoryzacji i twórz rzeczy dopiero wtedy, gdy wiesz, że są potrzebne.**

Nie powtarzaj się (DRY)

Stosowanie techniki kopiuj-wklej to błąd projektowy.

— David L. Parnas

Choć jest to jedna z najważniejszych zasad, mam pewność, że często się ją narusza (świadomie lub nie). Zasada „nie powtarzaj się” (ang. *don't repeat yourself* — DRY) mówi, że powinniśmy unikać powtórzeń, ponieważ są one czymś złym. Inna nazwa tej zasady to „raz i tylko raz” (ang. *once and only once* — OAOO).

Powód, dla którego powtórzenia w kodzie są bardzo niebezpieczne, jest oczywisty — po zmianie jednego fragmentu trzeba odpowiednio zmodyfikować także kopie kodu. I nie powinieneś kierować się tu optymizmem — możesz być pewny, że zmiany się pojawiają. Uważam, że nie trzeba wspominać, iż wcześniej lub później zapomnisz o każdym skopiowanym fragmencie. Przywitaj się więc z błędami.

W porządku, to tyle na ten temat. Nie ma już nic do dodania? Nie, jest jeszcze coś i trzeba się temu bliżej przyjrzeć.

W swojej świetnej książce *The Pragmatic Programmer* [Hunt99] Dave Thomas i Andy Hunt piszą, że stosowanie zasady DRY oznacza, iż musimy zapewnić, że „każda porcja wiedzy ma jedną, jednoznaczną, nadrzędną reprezentację w systemie”. Warto zauważyć, że Dave i Andy nie wspominają bezpośrednio o kodzie, tylko o wiedzy. A wiedza w systemie obejmuje znacznie więcej aspektów niż kod. Zasada DRY dotyczy też np.: dokumentacji, projektu, planów testów lub danych konfiguracyjnych systemu. Jest ona związana z wszystkimi materiałami. Zapewne się domyślasz, że ściśle przestrzeganie tej zasady nie jest tak łatwe, jak może się początkowo wydawać.

Ukrywanie informacji

Ukrywanie informacji to znana od dawna i podstawowa zasada rozwoju oprogramowania. Po raz pierwszy została udokumentowana w przełomowej pracy „On the Criteria to Be Used in Decomposing Systems Into Modules” [Parnas72] opublikowanej w 1972 r. przez wspomnianego już Davida L. Parnasa.

Wedle tej zasady jeden fragment kodu, który wywołuje inny kod, nie powinien znać wewnętrznych mechanizmów tego ostatniego. Dzięki temu można modyfikować wewnętrzne elementy wywoływanego fragmentu kodu bez konieczności wprowadzania zmian w kodzie zgłaszającym wywołania.

Parnas opisuje ukrywanie informacji jako podstawową zasadę stosowaną przy podziale systemów na moduły. Dowodzi, że modularyzacja systemu powinna obejmować ukrywanie trudnych decyzji projektowych lub rozwiązań, które zapewne zostaną zmienione. Im mniej wewnętrznych mechanizmów jednostka oprogramowania (np. klasa lub komponent) udostępnia środowisku, tym mniejsze powiązanie między implementacją danej jednostki i jej klientami. W efekcie zmiany w wewnętrznej implementacji jednostki programowej nie będą odczuwalne w środowisku.

Jest wiele zalet ukrywania informacji:

- Ograniczenie konsekwencji zmian w modułach.
- Minimalizacja wpływu na inne moduły, gdy konieczne są poprawki błędów.
- Znaczny wzrost możliwości ponownego wykorzystania modułów.
- Ułatwienie testowania modułów.

Ukrywanie informacji jest często mylone z hermetyzacją, nie jest to jednak to samo. Wiem, że w wielu znanych książkach oba te pojęcia są stosowane jako synonimy, jednak nie zgadzam się z tym podejściem. Ukrywanie informacji to zasada projektowa wspomagająca deweloperów w tworzeniu odpowiednich modułów. Ta zasada działa na wielu poziomach abstrakcji i ujawnia na nich swoje pozytywne skutki (zwłaszcza w dużych systemach).

Hermetyzacja często jest zależna od języka programowania i służy do ograniczania dostępu do wewnętrznych elementów modułu. Na przykład w C++ możesz poprzedzić listę składowych klasy słowem kluczowym `private`, aby zagwarantować, że nie będzie możliwy dostęp do nich spoza klasy. Jednak choć można wykorzystać tego rodzaju zabezpieczenia do kontrolowania dostępu, nadal dużo nam brakuje do automatycznego ukrywania informacji. Hermetyzacja wspomaga ukrywanie informacji, ale go nie gwarantuje.

Pokazany niżej przykładowy kod (listing 3.1) ilustruje klasę z hermetyzacją, ale z niskiej jakości ukrywaniem informacji.

Listing 3.1. Klasa do automatycznego sterowania drzwiami (fragment)

```
class AutomaticDoor {
public:
    enum class State {
        closed = 1,
        opening,
        open,
        closing
    };

private:
    State state;
    // ...tu więcej atrybutów...

public:
    State getState() const;
    // ...tu więcej funkcji składowych...
};
```

To nie jest ukrywanie informacji, ponieważ fragmenty wewnętrznej implementacji tej klasy są udostępniane w środowisku, choć hermetyzacja klasy wygląda poprawnie. Zauważ, że typ zwracanej wartości to `getState`. Klienci używające przedstawionej klasy potrzebują klasy wyliczeniowej `State`, co ilustruje listing 3.2.

Listing 3.2. Przykład używania klasy `AutomaticDoor` do sprawdzania aktualnego stanu drzwi

```
#include "AutomaticDoor.h"

int main() {
    AutomaticDoor automaticDoor;
    AutomaticDoor::State doorsState = automaticDoor.getState();
    if (doorsState == AutomaticDoor::State::closed) {
        // Wykonywanie operacji...
    }
    return 0;
}
```

Klasy (i struktury) wyliczeniowe [C++11]

W C++11 wprowadzono innowację w postaci typów wyliczeniowych. Aby zachować zgodność ze starszymi wersjami C++, nadal dostępne są dobrze znane wyliczenia z użyciem słowa kluczowego `enum`. Od wersji C++11 dostępne są też klasy i struktury wyliczeniowe.

Jednym z problemów z dawnymi wyliczeniami z C++ jest to, że powodowały eksport literałów z wyliczeń do zewnętrznej przestrzeni nazw, co skutkowało konfliktami nazw — tak jak w poniższym przykładzie:

```
const std::string bear;
// ...a w innym miejscu tej samej przestrzeni nazw...
enum Animal { dog, deer, cat, bird, bear };
// Błąd: 'bear' ponownie zadeklarowany jako symbol innego rodzaju.
```

Ponadto dawne wyliczenia z C++ niejawnie były przekształcane na typ `int`, co prowadziło do subtelnych błędów, gdy taka konwersja była nieoczekiwana lub niepożądana:

```
enum Animal { dog, deer, cat, bird, bear };
Animal animal = dog;
int aNumber = animal; // Niejawna konwersja: działa.
```

Gdy używane są klasy wyliczeniowe (nazywane też „nowymi wyliczeniami” lub „silnymi wyliczeniami”), te problemy nie występują. Literały z wyliczeń są wtedy lokalne, a ich wartości nie są niejawnie przekształcane na inne typy (np. na inne wyliczenia lub na typ `int`).

```
const std::string bear;
// ...a w innym miejscu tej samej przestrzeni nazw...
enum class Animal { dog, deer, cat, bird, bear }; // Brak konfliktu z łańcuchem znaków 'bear'.
Animal animal = Animal::dog;
int aNumber = animal; // Błąd kompilatora!
```

Zdecydowanie zachęcam do stosowania w nowoczesnych programach w C++ klas wyliczeniowych zamiast zwykłych dawnych wyliczeń. Dzięki temu kod będzie bezpieczniejszy, a ponieważ klasy wyliczeniowe też są klasami, można stosować dla nich deklaracje wyprzedzające.

Co się stanie, gdy trzeba będzie zmodyfikować wewnętrzną implementację klasy `AutomaticDoor` i usunąć z niej klasę wyliczeniową `State`? Łatwo dostrzec, że będzie miało to istotny wpływ na kod klienta. Spowoduje to zmiany wszędzie, gdzie używana jest funkcja składowa `AutomaticDoor::getState()`.

Na listingu 3.3 pokazana jest poddana hermetyzacji klasa `AutomaticDoor` z poprawnym ukrywaniem informacji. Listing 3.4 pokazuje, jak korzystać z tej klasy.

Listing 3.3. Lepiej zaprojektowana klasa do automatycznego sterowania drzwiami

```
class AutomaticDoor {
public:
    bool isClosed() const;
    bool isOpening() const;
    bool isOpen() const;
    bool isClosing() const;
    // ...dalsze operacje...

private:
    enum class State {
```

```

    closed = 1,
    opening,
    open,
    closing
};

State state;
// ...dalsze atrybuty...
};

```

Listing 3.4. Przykład pokazujący, jak w elegancki sposób używać klasy `AutomaticDoor` po wprowadzeniu zmian

```

#include "AutomaticDoor.h"

int main() {
    AutomaticDoor automaticDoor;
    if (automaticDoor.isClosed()) {
        // Wykonywanie operacji...
    }
    return 0;
}

```

Teraz modyfikowanie wewnętrznych elementów klasy `AutomaticDoor` jest znacznie łatwiejsze. Kod kliencki nie jest już zależny od wewnętrznych fragmentów tej klasy. Możesz więc usunąć wyliczenie `State` i zastąpić je implementacją innego rodzaju, a użytkownicy klasy nawet tego nie zauważą.

Wysoka spójność

Zgodnie z ogólnymi zaleceniami z obszaru rozwoju oprogramowania każdy element oprogramowania (synonimy: moduł, komponent, jednostka, klasa, funkcja) powinien mieć wysoką spójność. W ogólnym ujęciu spójność jest wysoka, jeśli moduł wykonuje dobrze zdefiniowane zadanie.

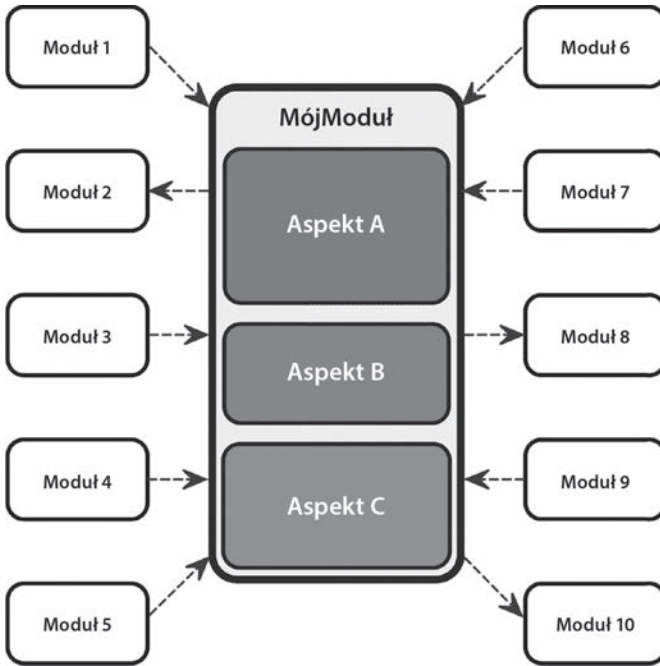
Aby lepiej zrozumieć tę zasadę, warto przyrzeć się dwóm przykładom, w których spójność jest niska. Zacznij od rysunku 3.1.

Na tej ilustracji podziału systemu na moduły trzy różne aspekty z dziedziny biznesowej są obsługiwane przez jeden moduł. Aspekty A, B i C nie mają nic (lub prawie nic) wspólnego ze sobą, jednak wszystkie trzy zostały uwzględnione w module `MójModuł`. Analiza kodu modułu powinna ujawnić, że funkcje związane z aspektami A, B i C operują na różnych i zupełnie niezależnych danych.

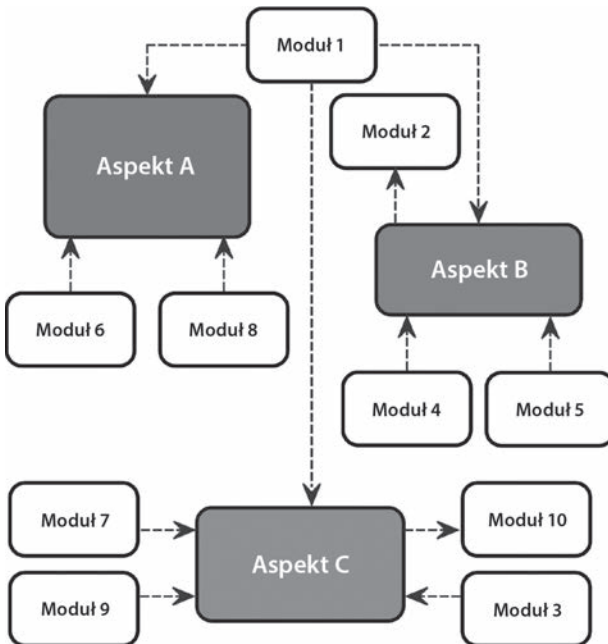
Teraz przyjrzyj się przerywanym kreskom na rysunku. Każda z tych kresek reprezentuje zależność. Element na początku strzałki wymaga w implementacji elementu wskazywanego przez grot. W tym scenariuszu każdy moduł systemu, który chce korzystać z usług A, B lub C, jest zależny od całego modułu `MójModuł`. Podstawowa wada tego projektu jest oczywista — powstaje w nim zbyt wiele zależności, a łatwość konserwacji poważnie spada.

Aby zwiększyć spójność, aspekty A, B i C należy oddzielić od siebie i przenieść do odpowiadających im modułów (rysunek 3.2).

Teraz łatwo zobaczyć, że każdy moduł ma znacznie mniej zależności niż dawny moduł `MójModuł`. Wyraźnie widać, że A, B i C nie mają bezpośrednio nic wspólnego ze sobą. Jedyny moduł, który zależy od wszystkich trzech (A, B i C), to moduł 1.

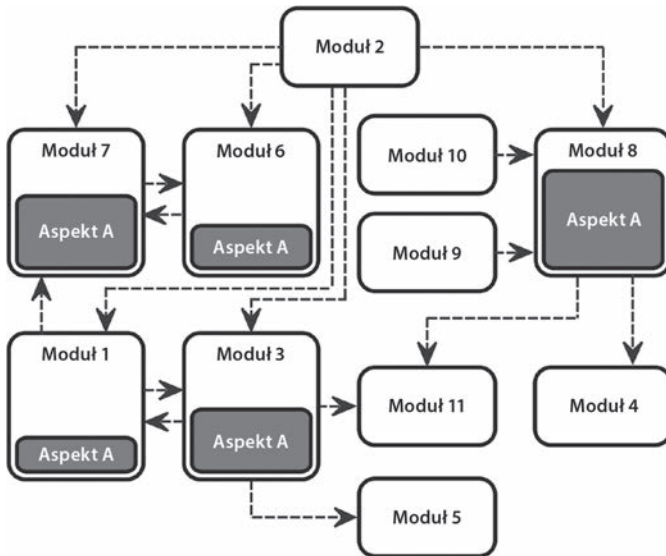


Rysunek 3.1. Moduł *MójModuł* ma zbyt wiele zadań, co skutkuje licznymi zależnościami (tego modułu od innych jednostek i na odwrót)



Rysunek 3.2. Wysoka spójność — połączone wcześniej aspekty A, B i C zostały rozdzielone między odrębne moduły

Inny rodzaj niskiej spójności jest opisany w **antywzorcu śrutówka** (ang. *shotgun anti-pattern*). Śrutówka to broń, która wyrzeliwuje mnóstwo małego okrągłego śrutu i ma zwykle duży rozrzut. W rozwoju oprogramowania ta metafora oznacza, że określony aspekt dziedziny lub jeden pomysł jest wysoce rozdrobniony i rozdzielony między wiele modułów. Ten scenariusz jest przedstawiony na rysunku 3.3.



Rysunek 3.3. Aspekt A został rozdrobniony między pięć modułów

Nawet przy tej postaci niskiej spójności powstaje wiele niekorzystnych zjawisk. Rozproszone fragmenty aspektu A muszą ściśle ze sobą współdziałać. To oznacza, że każdy moduł implementujący część aspektu A musi komunikować się z przynajmniej jednym innym modułem odpowiedzialnym za inną część tego aspektu, co prowadzi do dużej liczby zależności w projekcie. W najgorszym scenariuszu może to skutkować zależnościami cyklicznymi, takimi jak między modułami 1 i 3 lub 6 i 7. Wpływa to negatywnie na łatwość konserwacji i rozszerzania kodu. Ponadto, co oczywiste, testowanie jest wtedy znacznie utrudnione.

Tego rodzaju projekt wywołuje zjawisko nazywane **chirurgią śrutówką** (ang. *shotgun surgery*). Zmiana dotycząca aspektu A prowadzi do wielu drobnych modyfikacji w licznych modułach. Jest to bardzo niekorzystne i należy tego unikać. Trzeba rozwiązać ten problem, łącząc w jeden spójny moduł wszystkie elementy kodu będące fragmentami tego samego aspektu logicznego.

Istnieją też inne zasady sprzyjające wysokiej spójności — np. zasada jednej odpowiedzialności (ang. *single responsibility principle*) w projektowaniu obiektowym (zob. rozdział 6.). Wysoka spójność często współwystępuje z luźnym powiązaniem i na odwrót.

Luźne powiązanie

Przyjrzyj się krótkiemu przykładowi z listingu 3.5.

Listing 3.5. Przełącznik służący do włączania i wyłączania lampy

```
class Lamp {
public:
    void on() {
        //...
```

```

}

void off() {
    //...
}
};

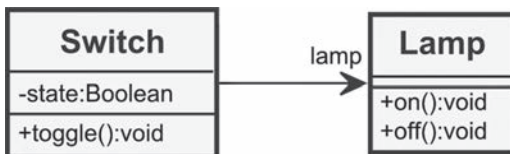
class Switch {
private:
    Lamp& lamp;
    bool state {false};

public:
    Switch(Lamp& lamp) : lamp(lamp) { }

    void toggle() {
        if (state) {
            state = false;
            lamp.off();
        } else {
            state = true;
            lamp.on();
        }
    }
};

```

Ten kod będzie działał. Najpierw możesz utworzyć obiekt typu `Lamp`, a następnie przekazać go przez referencję w trakcie tworzenia obiektu typu `Switch`. Na diagramie UML-a ten prosty przykład wygląda tak jak na rysunku 3.4.



Rysunek 3.4. Diagram klas `Switch` i `Lamp`

Na czym polega problem z tym projektem?

Na tym, że klasa `Switch` obejmuje referencję prowadzącą do konkretnego obiektu klasy `Lamp`. Oznacza to, że przełącznik wie o istnieniu lampy.

Możliwe, że zaczniesz argumentować tak: „No cóż, po to w końcu jest przełącznik — ma włączać i wyłączać lampy”. Odpowiem na to następująco: „Tak, jeśli jest to jedyne zadanie przełącznika, wtedy ten projekt może być odpowiedni. Jednak zachęcam do wizyty w sklepie z artykułami budowlanymi i przyjrzenia się dostępnym w nim przełącznikom. Czy wiedzą o istnieniu lamp?”

A co myślisz o testowaniu przedstawionego projektu? Czy przełącznik można przetestować niezależnie, jak jest to konieczne w testach jednostkowych? Nie, to niemożliwe. A co zrobić, jeśli przełącznik ma włączać nie tylko lampę, ale też wentylator lub elektryczne rolety?

W zaprezentowanym przykładzie przełącznik i lampa są **ściśle powiązane**.

W rozwoju oprogramowania należy dążyć do luźnego powiązania między modułami. Oznacza to, że należy budować systemy, w których każdy moduł ma (lub wykorzystuje) tylko niewielką lub zerową wiedzę na temat definicji z innych, odrębnych modułów.

Kluczem do luźnego powiązania w rozwoju oprogramowania są interfejsy. Interfejs obejmuje deklarację publicznie dostępnych operacji klasy bez zobowiązywania się do udostępniania konkretnej implementacji.

Interfejs przypomina kontrakt. Klasy implementujące dany interfejs zobowiązują się do realizacji kontraktu, czyli muszą udostępnić implementacje sygnatur metod z tego interfejsu.

W C++ interfejsy są tworzone za pomocą klas abstrakcyjnych, tak jak na listingu 3.6.

Listing 3.6. *Interfejs Switchable*

```
class Switchable {
public:
    virtual void on() = 0;
    virtual void off() = 0;
};
```

Klasa Switch nie musi teraz obejmować referencji do obiektu typu Lamp. Zamiast tego zawiera referencję do nowego interfejsu Switchable, co przedstawia listing 3.7.

Listing 3.7. *Zmodyfikowana klasa Switch z usuniętym obiektem typu Lamp*

```
class Switch {
private:
    Switchable& switchable;
    bool state {false};

public:
    Switch(Switchable& switchable) : switchable(switchable) {}

    void toggle() {
        if (state) {
            state = false;
            switchable.off();
        } else {
            state = true;
            switchable.on();
        }
    }
};
```

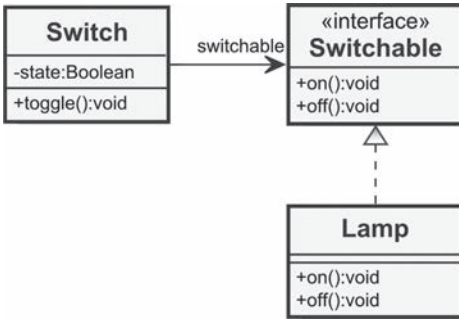
Klasa Lamp implementuje nowy interfejs (listing 3.8).

Listing 3.8. *Klasa Lamp implementuje interfejs Switchable*

```
class Lamp : public Switchable {
public:
    void on() override {
        // ...
    }

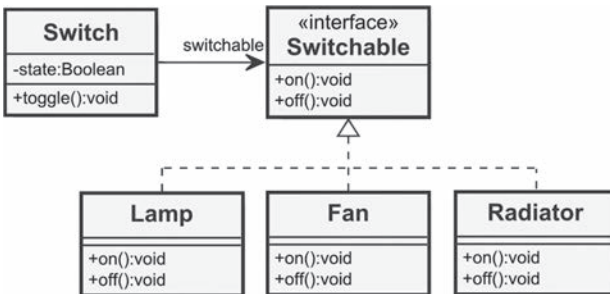
    void off() override {
        // ...
    }
};
```

Na diagramie UML-a nowy projekt wygląda tak jak na rysunku 3.5.



Rysunek 3.5. Klasy `Switch` i `Lamp` luźno powiązane za pomocą interfejsu

Zalety takiego projektu są oczywiste. Klasa `Switch` jest teraz w pełni niezależna od klas konkretnych, które ma kontrolować. Ponadto można ją niezależnie testować, tworząc zaślepkę z implementacją interfejsu `Switchable`. Chcesz kontrolować wentylator zamiast lampy? Nie ma problemu — ten projekt jest gotowy do rozszerzania. Wystarczy utworzyć klasę `Fan` lub inne reprezentujące urządzenia elektryczne klasy z implementacją interfejsu `Switchable`. Ilustruje to rysunek 3.6.



Rysunek 3.6. Dzięki interfejsowi klasa `Switch` może kontrolować różne klasy reprezentujące urządzenia elektryczne

Przykładanie wagi do luźnego powiązania może skutkować wysokim poziomem autonomii poszczególnych modułów systemu. Ta zasada może być skuteczna na różnych poziomach: zarówno w najmniejszych modułach, jak i w dużych komponentach na poziomie architektury systemu. Wysoka spójność sprzyja luźnemu powiązaniu, ponieważ moduł o ściśle zdefiniowanych zadaniach zwykle jest zależny od mniejszej liczby współdziałających komponentów.

Nie przesadzaj z optymalizacją

W programowaniu przedwczesna optymalizacja jest źródłem wszelkiego (lub prawie całego) zła.

— Donald E. Knuth, amerykański informatyk [Knuth74]

Widziałem, jak deweloperzy zaczęli nadmiernie czasochłonne optymalizacje z jedynie ogólnym wyobrażeniem kosztów, ale bez konkretnej wiedzy na temat źródła niskiej wydajności. Często majstrowali przy pojedynczych instrukcjach lub starali się zoptymalizować krótkie lokalne pętle, aby wykorzystać nawet najmniejszą okazję do poprawy wydajności. Na marginesie warto wspomnieć, że sam byłem takim programistą.

Takie działania zwykle przynoszą bardzo niewielkie korzyści. Oczekiwany wzrost wydajności przeważnie nie następuje. Ostatecznie cały wysiłek oznacza tylko zmarnowanie cennego czasu. Natomiast łatwość zrozumienia i konserwacji pozornie zoptymalizowanego kodu znacznie spada. W wyjątkowo niekorzystnym scenariuszu zdarza się nawet, że w ramach optymalizacji w kodzie pojawiają się subtelne błędy. Oto wskazówka: **jeśli nie musisz spełnić bezpośrednio podanych wymogów z zakresu wydajności, zrezygnuj z optymalizacji.**

Nadrzędnym celem powinna być łatwość zrozumienia i konserwacji kodu. W punkcie „A co z kosztami wywołań?” w rozdziale 4. wyjaśniam, że kompilatory obecnie świetnie sobie radzą z optymalizowaniem kodu. Gdy tylko poczujesz ochotę, by coś zoptymalizować, pomyśl o zasadzie YAGNI.

Do optymalizacji powinieneś przystępować tylko wtedy, gdy nie są spełnione wymogi z zakresu wydajności otwarcie przedstawione przez interesariusza. Wtedy powinieneś najpierw starannie przeanalizować, z czego wynika niska wydajność. Nie wprowadzaj optymalizacji wyłącznie na podstawie przeczuć. Możesz np. posłużyć się profilerem do ustalenia, gdzie występują wąskie gardła. Po zastosowaniu takiego narzędzia deweloperzy zazwyczaj są zaskoczeni, że spadek wydajności następuje w zupełnie innych miejscach, niż początkowo podejrzewali.

■ **Uwaga** Profiler to narzędzie do dynamicznych analiz programu. Mierzy on m.in. częstotliwość i czas wykonywania wywołań funkcji. Zebrane informacje mogą posłużyć jako pomoc w trakcie optymalizowania programu.

Zasada minimalizowania zaskoczenia

Zasada minimalizowania zaskoczenia (ang. *Principle of Least Astonishment* — POLA/PLA lub *Principle of Least Surprise* — POLS) jest dobrze znana w dziedzinie projektowania i ergonomii interfejsów użytkownika. Zgodnie z nią klient nie powinien być zaskakiwany nieoczekiwanymi reakcjami interfejsu użytkownika. Nie powinien być zadziwiany pojawiającymi się lub znikającymi kontrolkami, mylącymi komunikatami o błędach, nietypowymi reakcjami na popularne skróty klawiszowe (pamiętaj, że w systemie Windows skrót *Ctrl+C* jest niemal standardowym sposobem kopiowania informacji i nie służy do zamykania programu) lub innymi niespodziewanymi działaniami.

Tę zasadę można przenieść do obszaru projektowania interfejsów API w rozwoju oprogramowania. Wywołanie funkcji nie powinno zaskakiwać jednostki wywołującej niespodziewanymi lub zagadkowymi efektami ubocznymi. Funkcja powinna robić dokładnie to, na co wskazuje jej nazwa (zob. punkt „Nazwy funkcji” w rozdziale 4.). Na przykład wywołanie gettera dla obiektu danej klasy nie powinno skutkować modyfikacją wewnętrznego stanu tego obiektu.

Reguła harcerza

Ta zasada dotyczy Ciebie i Twojego postępowania. Oto jej treść: **zawsze pozostawiaj obozowisko czystsze, niż je zastałeś.**

Harcerze przestrzegają wielu zasad. Jedna z nich mówi, że powinni natychmiast posprzątać bałagan lub śmieci w otoczeniu po ich napotkaniu. Jako odpowiedzialni twórcy oprogramowania powinniśmy stosować tę regułę w codziennej pracy. Za każdym razem, gdy natrafisz na fragment kodu, który można usprawnić lub w którym występuje „brzydki zapach”, powinieneś natychmiast go poprawić.

Zaletą takiego postępowania jest to, że zapobiega się degradacji kodu. Jeśli wszyscy będziemy postępować w ten sposób, kod się nie zepsuje, ponieważ entropia będzie miała niewielkie szanse na opanowanie systemu. Wprowadzane usprawnienia nie muszą być duże. Mogą to być bardzo drobne poprawki. Oto przykłady:

- Przemianowanie klas, zmiennych, funkcji lub metod o nieodpowiednich nazwach (zob. punkt „Dobre nazwy” w rozdziale 4.).
- Podział kodu dużej funkcji na mniejsze fragmenty (zob. punkt „Twórz małe funkcje” w rozdziale 4.).

- Usunięcie komentarza przez sprawienie, że opatrzony nim kod stanie się czytelny (zob. punkt „Komentarze” w rozdziale 4.).
- Uporządkowanie złożonego i zagadkowego bloku `if-else`.
- Usunięcie niewielkiego fragmentu powtarzającego się kodu (zob. punkt poświęcony zasadzie DRY z tego rozdziału).

Ponieważ większość tych usprawnień to refaktoryzacje kodu, niezbędna jest solidna siatka bezpieczeństwa składająca się z dobrych testów jednostkowych, co opisano w rozdziale 2. Bez testów jednostkowych nie możesz mieć pewności, że czegoś nie uszkodzisz.

Oprócz wysokiego pokrycia kodu testami jednostkowymi niezbędna jest też specjalna kultura pracy w zespole: **współwłasność kodu**.

Współwłasność kodu oznacza, że powinniśmy pracować jak prawdziwa wspólnota. Każdy członek zespołu może w dowolnym momencie modyfikować lub rozszerzać wybrany fragment kodu. Należy unikać nastawienia: „To kod Piotrka, a to moduł Jacka — nie mam zamiaru tego ruszać!”. Powinniśmy uważać za korzystne, że inne osoby mogą przejąć napisany przez nas kod. Nikt w prawdziwym zespole nie powinien obawiać się porządkowania kodu lub dodawania nowych funkcji ani prosić o pozwolenie na to. W kulturze współwłasności kodu reguła harcerza dobrze się sprawdzi.

Skorowidz

A

Abrahams David, 130
abstrakcja, 36, 161
action, *Patrz:* wzorzec projektowy polecenie
adapter, 231, 232
ADL, 110
adnotacja, 65, 66
agregacja, 277, *Patrz:* redukcja
algorytm
 Euklidesa, 176
 Mersenne Twister, 178
 sortowania, 233
 std::accumulate, 190
 std::equal, 127
 std::for_each, 126
 std::gcd, 177
 std::remove_if, 180, 189
 std::reverse, 124
 std::sort, 126
 std::transform, 189
 szybkiego sortowania, 77
antywzorzec
 babeczka, 26, 27
 klasa – bóg, 142, 169
 nie wymyślono u nas, 123
 rozek lodowy, 26, 27
 singleton, 221, 222, 223, 224
 sklep ze starzyzną, 169
 śrutówka, 46
Apache OpenOffice, 54
architektura klient-serwer, 237, 240
argument, 72
 liczba, 73
 nazwa, 55

 w postaci
 referencji, 80
 wskaźnika, 80, *Patrz też:* wskaźnik
 wyjściowy, 69, 75, 76
arkusz kalkulacyjny, 246
assembler, 230
asercja, 31
 niestandardowa, 208
asocjacja, 275
 agregacja, *Patrz:* agregacja
 bez możliwości nawigacji, 276
 etykieta, 276
 kierunek czytania, 276
 kompozycja, *Patrz:* kompozycja
 krotność, 276
 nazwa, 276
 skierowana, 276

B

banda czterech, 219
Bay Jeff, 142
baza danych, 36
 testowanie, 33
Beck Kent, 195
bezpieczeństwo
 ze względu na typ, 121
 ze względu na wątki, 260
 ze względu na wyjątki, 130, 131, 132
 najwyższe, *Patrz:* niewidoczność niepowodzeń
 podstawowe, 131
 wysokie, 132
biblioteka
 Boost, 128
 Boost.Filesystem, 129

biblioteka

- Boost.Thread, 125
- Boost.Units, 123
- BoostLog, 231, 232
- Filesystem, 129
- Range-v3, 129
- równoległych struktur danych, 129
- Thread Support, 125
- wartości fizycznych, 123
- wyrażeń regularnych, 129

big ball of mud, *Patrz:* bryła błotna

blob, *Patrz:* antywzorzec klasa – bóg

blokada, 93

błąd, 130

Boost.Text, 28

bryła błotna, 15

C

cecha typu, 181, 182

chain of responsibility, *Patrz:* wzorzec projektowy

łańcuch odpowiedzialności

chirurgia śrutówką, 46

Church Alonzo, 171

Cline Marshall, 106

Cohn Mike, 25

command, *Patrz:* wzorzec projektowy polecenie

copy elision, *Patrz:* oprogramowanie optymalizacja RVO

CppUnit, 28

cross-cutting concern, *Patrz:* zagadnienie przekrojowe

cup cake, *Patrz:* antywzorzec babeczka

CUTE, 28

D

Dahl Ole-Johan, 139

DDD, 56, 57

debugowanie, 27, 28

deklaracja wyprzedzająca, 156, 158

dekorator, 143

delegowanie, 154

Demeter

- prawo, *Patrz:* prawo Demeter
- projekt, *Patrz:* projekt Demeter

dereferencja, 80

destruktor, 106, 109

- nietrzywalny, 108

diagram

- klas, 271
- UML-a, 21

Digital Revolution, *Patrz:* rewolucja cyfrowa

dług techniczny, 17, 54

dokumentacja, 65, 66

domain-driven design, *Patrz:* DDD

domknięcie, 185, 186, 187

Doxygen, 65, 66

drzewo, 242

dyrektywa #pragma once, 260

dziedziczenie, 143, 144, 146, 154, 277, 278

dzielnik największy wspólny, *Patrz:* NWD

E

Evans Eric, 56

exception-safety, *Patrz:* bezpieczeństwo ze względu na wyjątki

eXtreme Programming, *Patrz:* programowanie ekstremalne

F

failure transparency, *Patrz:* niewidoczność niepowodzeń

fake objects, *Patrz:* zasłlepka

Fernandes Martinho, 109

filtrowanie, 189

flaga, 74

fluent interface, *Patrz:* interfejs płynny

Footo Brian, 15

Fowler Martin, 167

funkcja, 67, 173

- anonimowa, 17, 185
- argument, *Patrz:* argument
- begin, 88
- bezargumentowa, 73
- cbegin, 88
- cend, 88
- częściowe wywołanie, 183, 184
- czysta, 172, 174
- długość, 70, 71
- dwuargumentowa, 177
- end, 88
- get, 98
- jako argument, 172
- jednoargumentowa, 177, 179
- koszt wywołań, 71
- lambda, *Patrz:* wyrażenie lambda
- łańcuch wywołań, 76
- matematyczna, 173

nazwa, 54, 55, 58, 60, 65, 69, 70, 71
 intencja, 72
 predykat, 71
 nieczysta, 175
 nieskładowa, 88
 printf, 84
 puts, 87
 sprintf, 84, 87
 składowa, 88
 specjalna, 35
 statyczna, 170
 std::begin, 88
 std::cbegin, 88
 std::cend, 88
 std::enable_if, 263
 std::end, 88
 std::iota, 178
 std::make_shared, 101
 std::make_unique, 101
 std::rbegin, 88
 std::rend, 88
 std::to_string, 84
 std::to_wstring, 84
 to_string, 85
 wieloargumentowa, 73
 wirtualna, 75, 151
 wyższego poziomu, 187
 zwijania bloków kodu, 64
 funktor, 177, 180, 181
 dwuargumentowy, 182
 jednoargumentowy, 181

G

Gamma Erich, 219, 221
 generalizacja, 277, 278
 generator, 177
 generator dokumentacji, *Patrz:* dokumentacja
 generator liczb losowych, 178
 getter, 32
 gl-wartość, 103
 Google Test, 28, 201, 204
 gwarancje Abrahamsa, 130

H

Helm Richard, 219
 hermetyzacja, 42, 167, 233, 253
 high performance computing, *Patrz:* HPC
 historia użytkownika, 60

Hoare Tony, 77
 Holland Iana, 165
 HPC, 113
 Hunt Andy, 41

I

ice cream cone, *Patrz:* antywzorzec rożek lodowy
 idiom, 260
 erase-remove, 180
 niepowodzenie podstawiania nie jest błędem,
 Patrz: SFINAE
 PIMPL, 267, 268
 RAII, 80, 81, 95, 101
 skopiuj i przestaw, 264, 266
 wartownik dołączania, 260
 include guard, *Patrz:* idiom wartownik dołączania
 informacje o typie w czasie wykonywania programu,
 Patrz: RTTI
 interfejs, 273
 API, 80
 dostosowywanie, 231, 232
 płynny, 236
 podział, 154, 155
 realizacja, 273
 roli, 155
 udostępnianie, 275
 internet rzeczy, *Patrz:* IoT
 Internet of Things, *Patrz:* IoT
 IoT, 24
 iterator, 88, 220

J

język
 ALGOL, 77
 C, 82, 84, 139
 C++, 18
 C++, 17, 18, 53
 Clojure, 171
 Erlang, 171
 F#, 171
 funkcyjny, 171, 172
 Haskell, 171
 interpretowany, 18
 Java, 18
 JavaScript, 172
 kompletny w sensie Turinga,
 Patrz: kompletność w sensie Turinga
 Lisp, 171

język

- Miranda, 171
- obiektowy z obsługą polimorfizmu, 60
- OMG UML, 271
- Scala, 172
- Scheme, 171
- Simula-67, 139
- Smalltalk, 28, 139, 140
- UML, 21
- zarządzany, 18
- ze słabą kontrolą typów, 59
- Johnson Ralph, 219
- junk shop, *Patrz:* antywzorzec sklep ze starzyzną

K

- kata, 200
- Kay Alan, 139
- Kay Alan Curtis, 140
- Kiziński Maks, 129
- klasa, 271
 - abstrakcyjna, 272
 - anemiczna, 166, 167
 - atrybut, 272
 - bazowa, 149, 150
 - element
 - funkcjonalny, 141
 - strukturalny, 141
 - instancja, 271, 273
 - Logger, 224
 - Money, 253, 255, 256
 - MyString, 107, 108
 - nazwa, 55, 58, 60, 272
 - niemodyfikowalna, 150, 260
 - niezmiennik, 149, 150
 - operacja, 272
 - pochodna, 149, 150
 - polimorficzna, 151
 - poziom widoczności, 272
 - projektowanie, 141
 - std::string_view, 225
 - tworzenie, 141
 - wielkość, 142
 - wyliczeniowa, 43
- kod, *Patrz też:* oprogramowanie
 - architektura, *Patrz:* kod struktura
 - czysty, 17
 - degradacja, 50
 - jakość, 15, 16, 17
 - wewnętrzna, 15, 93
 - zewnętrzna, 15, 93

- pokrycie testami, 15, 17, 25, 26, 27, 28, 199, 217
 - produkcyjny, 33, 34
 - samodokumentujący się, 55
 - struktura, 15
 - testów, 33
 - współwłasność, 51
 - źródłowy
 - czytelność, 55
 - parsowanie, 65
- Koenig Andrew, 110
- Koeniga wyszukiwanie, *Patrz:* ADL
- kolekcja o stałej wielkości, 76
- komentarz, 15, 54, 60, 61, 64, 65
 - adnotacja, *Patrz:* adnotacja
 - blokowy, 62, 63, 64, 67
 - dezaktywujący fragment kodu, 61
 - system kontroli wersji, 64
- kompilator, 113, 121
 - funkcja, 110
 - implementacja, 116
- kompletność w sensie Turinga, 175, 176
- komponent, 275, 279
- kompozycja, 154, 277
- kompozyt, 243, 245
- konstruktor
 - kopiujący, 102, 106, 107, 108, 109
 - przenoszący, 102, 106, 107, 108, 109, 132
- kontener, 101
 - asocjacyjny, 124
 - inicjowanie za pomocą literału, 111
 - sekwencyjny, 124
- kwalifikator const, 81

L

- lambda, *Patrz:* wyrażenie lambda
- Lieberherr Karl, 165
- Liskov Barbara, 150
- literał
 - definiowany przez użytkownika, 122
 - funkcyjny, *Patrz:* wyrażenie lambda
- l-wartość, 103, 104

Ł

- łańcuch
 - odpowiedzialności, 240
 - znaków, 82, 83, 225

M

makro, 90
 mapowanie, 189
 Martin Robert, 70, 73, 150, 200
 Mascitti Rick, 139
 maszyna Turinga, 175
 McCabe Thomas, 69
 metaprogramowanie z użyciem szablonów, *Patrz:* TMP
 metoda, *Patrz:* funkcja
 przesłanie, 153
 Meyers Scott, 222
 mock-ups, *Patrz:* zaślepka
 model dziedziny anemiczny, 167
 model-widok-kontroler, *Patrz:* MVC
 MVC, 246

N

nakładka, *Patrz:* adapter
 niewidoczność niepowodzeń, 132
 notacja
 kółko – gniazdo, 275, 279
 węgierska, 59
 z odejmowaniem, 201, 213
 NULL, 77, 78, 80
 NWD, 175, 176
 Nygaard Kristen, 139

O

O'Brien Larry, 221
 obiekt, 140, 271
 fabryka, 279
 niemodyfikowalny, 76, 150, 260
 pośredniczący, 225
 tuple, 77
 tworzenie, 79, 141
 typu T, 97
 własność, 78
 object-oriented analysis and design, *Patrz:* OOAD
 obserwator, 246
 OOAD, 57
 operacja, *Patrz:* funkcja
 operator
 dwuargumentowy, 182
 przypisania, 265
 kopiujący, 102, 106, 107, 108, 109
 przenoszący, 102, 106, 107, 108, 109, 132
 typeid, 151

oprogramowanie, 24, *Patrz też:* kod
 architektura, *Patrz:* kod struktura
 entropia, 16, 53
 optymalizacja, 49, 71, 81
 RVO, 106
 sterowane poleceniami, 237
 złożoność cyklomatyczna, *Patrz:* złożoność
 cyklomatyczna

P

pamięć, 93
 nieograniczona, 175
 odzyskiwanie, 97, 107
 wyciekanie, 99, 100
 parametr, *Patrz:* argument
 Parnas David, 41
 pętla, 124
 piramida testów zdegenerowana, 26, 27
 piramida testów, 25
 Plain Old Unit Testing, *Patrz:* POUT
 plik
 algorithm, 124, 126, 178, 188, 189
 chrono, 129
 copyright.txt, 64
 filesystem, 129
 functional, 183
 konfiguracyjny, 69
 libcds, 129
 license.txt, 64
 limits, 254
 memory, 96
 nagłówkowy, 67
 tuple, 76
 numeric, 177, 178, 188
 random, 178
 regex, 129
 stdexcept, 135
 type_traits, 181, 182, 263
 typeid, 151
 uchwyt, 93
 utility, 124
 polimorfizm, 144, 224, 257
 dynamiczny, 150
 polityka
 std::execution::par, 126
 std::execution::par_unseq, 126
 std::execution::seq, 126
 postać kanoniczna, 219
 POUT, 196, 197

powiązanie
 luźne, 36, 46, 47, 49, 224, 228, 230
 ścisłe, 47
 prawa autorskie, 64
 prawo Demeter, 162, 164, 165, 166
 predykat
 dwuargumentowy, 182
 jednoargumentowy, 180
 procedura, *Patrz:* funkcja
 Profiler, 50
 programowanie
 ekstremalne, 195
 funkcyjne, 171, 172, 173, 175, 187, 192, 220
 generyczne, 182
 obiektowe, 56, 140, 141, 220, 277
 historia, 139
 podstawy, 141
 równoległe, 125, 126
 sterowane testami, *Patrz:* TDD
 wielowątkowe, *Patrz:* wielowątkowość
 zasady, 39
 projekt
 Demeter, 165
 Mercury, 195
 prototyp, 218
 pr-wartość, 103
 przejrzystość referencyjna, 173
 przestrzeń nazw, 111, 141
 nazwa, 55, 58, 60
 przetwarzanie współbieżne, *Patrz:* współbieżność,
 wielowątkowość
 przezroczystość referencyjna, *Patrz:* przejrzystość
 referencyjna

R

rachunek lambda, 171, 172
 RAI, *Patrz:* idiom RAI
 Ram Stefan, 140
 redukcja, 189, 190
 lewostronna, 190, 191
 prawostronna, 190, 191
 refaktoryzacja, 27, 55, 199, 207, 211
 wzorców projektowych, 221
 referencja
 NULL, 77
 z deklaracją wyprzedzającą, 156, 158
 referential transparency, *Patrz:* przejrzystość
 referencyjna
 regresja, 27

reguła, *Patrz:* zasada
 rekurencja, 175, 233
 Resource Acquisition Is Initialization, *Patrz:* idiom RAI
 Return Value Optimization, *Patrz:* oprogramowanie
 optymalizacja RVO
 rewolucja cyfrowa, 24
 Ritchie Dennis, 139
 RTTI, 151, 152
 run-time type information lub run-time type
 identification, *Patrz:* RTTI
 r-wartość, 103, 104
 referencja, 104, 105
 rzutowanie
 dynamic_cast, 151
 typów, 89

S

Scrum, 60
 semantyka przenoszenia, 79, 80, 102, 103, 104, 106
 setter, 32, 148
 SFINAE, 262, 263
 shotgun anti-pattern, *Patrz:* antywzorzec śrutówka
 shotgun surgery, *Patrz:* chirurgia śrutówka
 Simonyi Charles, 59
 single responsibility principle, *Patrz:* zasada jednej
 odpowiedzialności
 singleton, 31, 221, 222, 223, 224
 Meyersa, 222
 składowa statyczna, 31
 słowo kluczowe
 auto, 77, 110, 111, 112, 113, 184
 const, 81, 113
 constexpr, 120
 enum, 43
 interface, 273
 Software Craftmanship, 200
 Sommerlad Peter, 109
 sonda Mars Climate Orbiter I, 117
 specyfikacja Parallelism TS, 125
 specyfikator
 constexpr, 113, 115, 116, 133
 final, 146
 noexcept, 133
 throw, 133
 SRP, *Patrz:* zasada jednej odpowiedzialności
 stała
 nazwa, 55, 58, 60, 71
 poprawność, 81

standard
 C++11, 17, 18, 93, 110
 C++14, 19, 93
 C++17, 19, 93, 96, 126
 C++20, 18
 ISO/IEC TS 19570:2015, *Patrz:* specyfikacja
 Parallelism TS
 stereotyp, 279
 sterta, 54, 79, 80, 94
 stos, 79, 94, 100
 strategia, 143
 zero tolerancji, 16
 Stroustrup Bjarne, 139
 Substitution Failure Is Not an Error, *Patrz:* SFINAE
 SUnit, 28
 Sutter Herb, 125
 system
 kontroli wersji, 62, 64, 67, 201
 zastany, 31
 zewnętrzny, 33
 szablon, 182
 klasa, 76, 77, 96, 181
 nazwa, 55
 o zmiennej liczbie argumentów, 76
 std::array, 87
 std::auto_ptr, 96
 std::function, 184
 std::generate, 177
 std::optional, 259
 std::shared_ptr, 97
 std::tuple, 76, 77
 std::unary_function, 183
 std::unique_ptr, 96
 zmiennych, 115

T

TDD, 23, 67, 195, 197
 cykl, 199
 CZERWONE, 199, 200, 203, 204, 205
 REFAKTORYZACJA, 199
 ZIELONE, 199, 200, 204
 wyjątki, 217
 zalety, 216
 template metaprogramming, *Patrz:* TMP
 teoria wybitych okien, 16
 test
 akceptacyjny, 15, 25
 automatyzacja, 25
 czarnoskrzynkowy, 15

integracyjny, 36
 interfejsu użytkownika, 25
 jednostkowy, 23, 25, 26, 27, 28, 66, 67, 224
 kolejność, 31
 nazwa, 29, 30
 niezależność, 31
 szybkość, 36
 środowisko, 32
 tworzenie, 28, 29, 196, 197, 198, 201, 202
 zwykły dawny, *Patrz:* POUT
 kodu innych programistów, 32
 piramida, *Patrz:* piramida testów
 systemu, 25, 26
 wartości NULL, 78
 test doubles, *Patrz:* zaślepka
 test-driven development, *Patrz:* TDD
 testowanie, 23
 Thomas Dave, 41, 200
 TMP, 17, 175, 176
 transakcja, 93, 130
 Turing Alan, 175
 Turinga
 kompletność, *Patrz:* kompletność w sensie Turinga
 maszyna, *Patrz:* maszyna Turinga
 typ
 arytmetyczny, 254
 cecha, *Patrz:* cecha typu
 Customer, 276
 dedukcja automatyczna, 110, 112
 double, 253, 254
 float, 253, 254
 HANDLE, 101
 inferencja, 110
 long double, 253, 254
 Momentum, 119, 120
 niekompletny, 157
 rzutowanie, *Patrz:* rzutowanie typów
 semantyczny, 119
 std::array, 87
 std::chrono::duration, 129
 std::chrono::system_clock, 129
 std::exception, 135
 std::initializer_list, 111
 std::shared_ptr, 97, 101, 102
 std::string, 107, 124
 std::string_view, 225
 std::type_info, 151
 std::type_info::hash_code, 151
 std::unique_ptr, 102
 std::vector, 111

typ
 std::weak_ptr, 98
 T, 97, 108
 wyliczeniowy, 43
 type trait, *Patrz:* cecha typu

U

uchwyt, 101
 usługa, *Patrz:* funkcja
 użytkownika historia, *Patrz:* historia użytkownika

V

Vlissides John, 219

W

wartościowanie leniwe, 171
 wartość z lokalizacją, 103
 wątek, 93
 wielowątkowość, 125, 172
 Wing Jeannette, 150
 Wirth Niklaus, 77
 wskaźnik
 inteligentny, 80, 81, 95, 117
 implementacja, 97
 jako argument, *Patrz:* argument w postaci
 wskaźnika
 NULL, 77, 78, 80, 256
 nullptr, 77, 78, 107, 134, 256
 z deklaracją wyprzedzającą, 156, 158
 współbieżność, 140, 172
 wstrzykiwanie
 z użyciem settera, 231
 za pomocą konstruktora, 230
 zależności, 221, 224, 230, 250
 zaślepki, 231
 wyjątek, 134
 definiowanie, 135
 obsługa, 94, 130
 przechwytywanie, 94
 std::bac_alloc, 134
 unikanie, 130
 wyrażenie
 lambda, 17, 112, 184, 185
 generyczne, 187
 implementacja, 185
 lista przechwytywania, 185
 nazwa, 185

wywołanie, 186
 zapowiedź, 185
 redukcji, 191
 stałe, 113
 throw, 133
 wyszukiwanie Koeniga, *Patrz:* ADL
 wywołanie
 delete, 100, 101
 new, 100, 101
 wzorzec projektowy, 77, 143, 219, 220
 fabryka, 250
 fasada, 252, 253
 iterator, 220
 kompozyt, 243, 245
 łańcuch odpowiedzialności, 240
 MVC, 245
 obserwator, 246
 polecenie, 237, 238
 przypadek specjalny, 257, 258
 singleton, 221, 222, 223, 224
 strategia, 233, 236

X

XP, *Patrz:* programowanie ekstremalne
 xUnit, 28
 x-wartość, 103

Y

Yoder Joseph, 15

Z

zagadnienie przekrojowe, 130
 zależność, 73, 278
 cykliczna, 99, 100, 156, 158
 zrywanie, 159, 160
 odwracanie, 160
 tworzenia, 278
 użytkowania, 278
 zarządzanie zasobami, 93
 zasada
 DRY, 41, 65, 148, 152, 192
 harcerza, 50
 jednego zadania, 74
 jednej odpowiedzialności, 46, 142, 163, 193, 252
 KISS, 40, 192, 198, 220
 minimalizowania zaskoczenia, 149
 minimalnej wiedzy, *Patrz:* prawo Demeter

- mów zamiast pytać, 167, 168
- nie będziesz tego potrzebować, *Patrz:* zasada YAGNI
- nie powtarzaj się, *Patrz:* zasada DRY
- nie rozmawiaj z nieznanymi, *Patrz:* prawo Demeter
- OAOO, 41
- odwracania zależności, 158, 161, 162
- otwarte – zamknięte, 143, 233, 235, 238
- pięciu, 106, 107, 108
- podstawiania Liskov, 144, 150
- podziału interfejsu, 154, 155, 240
- podziału odpowiedzialności, 250, 252
- projektowa, 220
- przedkładaj kompozycję nad dziedziczenie, 154
- spójności, 44, 46, 74
- TDD, 200, 203, 208
- trzech, 106
- ukrywania informacji, 41, 42, 79, 252
- YAGNI, 40, 192
- zachowaj prostotę, głupku, *Patrz:* zasada KISS
- zależności acyklicznych, 100, 156, 158
- zera, 109
- zasoby
- dealokacja, 108
- niezależne, 101
- wyciekanie, 94
- zarządzanie, *Patrz:* zarządzanie zasobami
- zaśleпка, 36, 143, 199, 224
- wstrzykiwanie, *Patrz:* wstrzykiwanie zaślepki
- zdarzenie rejestrowania, 130
- zegar, 93
- złożoność cyklotematyczna, 69, 207
- zmienna, 81
- lokalna, 185
- nazwa, 55, 58, 60, 65, 71
- składowa statyczna, 170
- typ, 59
- zakres, 59
- znacznik @addtogroup, 67
- znak łańcuch, *Patrz:* łańcuch znaków
- zwijanie, *Patrz:* redukcja
- zwykle dawne testy jednostkowe, *Patrz:* POUT

PROGRAM PARTNERSKI

GRUPY WYDAWNICZEJ HELION



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

Zmień swoją stronę WWW
w działający bankomat!

Dowiedz się więcej i dołącz już dzisiaj!

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

Czysty kod C++17: elegancja, prostota i moc.

C++ jest wszechstronnym, potężnym językiem programowania, który ma bardzo różne zastosowania. To klasyczne, wciąż udoskonalane i unowocześniane narzędzie. Problemem jednak jest to, że programiści piszący w C++ dość często mają złe nawyki. Nie przestrzegają zasad manifestu Software Craftsmanship, stosują okropną składnię, całkowicie ignorują podstawowe reguły prawidłowego projektowania i pisania oprogramowania, a tworzony przez nich kod, choć niejednokrotnie wydajny i szybki, bywa niemal niemożliwy w utrzymaniu.

Jeśli chcesz, aby Twój kod C++ stał się optymalny i zyskał wyższą jakość, powinieneś uważnie przestudiować tę książkę. Pisanie dobrego kodu oznacza przyjemniejszą i wydajniejszą pracę. Niezależnie od tego, czy dopiero zaczynasz przygodę z C++, czy jesteś doświadczonym deweloperem, znajdziesz w tej publikacji cenne i przydatne informacje dotyczące zasad pisania kodu. Akronimy KISS, YAGNI i DRY zyskają dla Ciebie nowe znaczenie! Otrzymasz też mnóstwo przydatnych wskazówek odnoszących się do pisania funkcji, prowadzenia testów, obsługi wyjątków i błędów, a nawet umieszczania komentarzy w kodzie. Na licznych przykładach pokazano, jak pisać zrozumiały, elastyczny, łatwy w konserwacji i wydajny kod w C++.

W książce między innymi:

- solidne wyjaśnienie zasad pisania czystego kodu w C++
- programowanie funkcyjne i obiektowe
- wskazówki dotyczące programowania sterowanego testami (test-driven development)
- wzorce projektowe i idiomy z C++
- praktyczne wykorzystanie wzorców projektowych podczas programowania

Stephan Roth — jest pełnym pasji coachem, konsultantem i szkoleniowcem specjalizującym się w inżynierii systemów i oprogramowania. Ma doświadczenie jako architekt oprogramowania w obszarach rozpoznania radiowego oraz telekomunikacyjnych systemów wywiadowczych. Zabiera głos na specjalistycznych konferencjach i jest autorem kilku publikacji. To aktywny zwolennik ruchu Software Craftsmanship, którego interesują zasady i praktyki podejścia Clean Code Development.

 Helion	<i>Sprawdź nasze szkolenia!</i>	KOD KORZYŚCI <i>Sięgnij po więcej!</i>	
 helion.pl	 SZKOLENIA	ISBN 978-83-283-4340-5	
 HELION SA ul. Kościuszki 1c 44-100 Gliwice tel.: 32 230 98 63 helion@helion.pl	AKADEMIA IT & BUSINESS WWW.SZKOLENIA.HELION.PL	 9 788328 343405	
INFORMATYKA W NAJLEPSZYM WYDANIU		Cena: 59,00 zł	