Helion

Wydanie II

Czysty kod

Podręcznik dobrego programisty

Robert C. Martin

Przedmowa Micah D. Martin

Posłowie Justin M. Martin

Tytuł oryginału: Clean Code: A Handbook of Agile Software Craftsmanship, 2nd Edition

Tłumaczenie: Anna Mizerska

ISBN: 978-83-289-3791-8

Authorized translation from the English language edition, entitled CLEAN CODE:
A HANDBOOK OF AGILE SOFTWARE CRAFTSMANSHIP, 2nd EDITION
by ROBERT C. MARTIN, published by Pearson Education, Inc, publishing as
Addison Wesley Professional, Copyright © 2026 Pearson Education, Inc.

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 Pearson Education, Inc.

Polish language edition published by Helion S.A., Copyright © 2026.
Pearson uses AI, including to support the creation of content.
We claim copyright to the fullest extent permissible by law.

Wszelkie prawa zastrzeżone. Nieautoryzowane rozpowszechnianie całości lub fragmentu niniejszej publikacji w jakiejkolwiek 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
helion.pl/user/opinie/czysk2
Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

Helion S.A.
ul. Kościuszki 1c, 44-100 Gliwice
tel. 32 230 98 63
e-mail: *helion@helion.pl*
WWW: *helion.pl* (księgarnia internetowa, katalog książek)

Printed in Poland.

- Kup książkę
- Poleć książkę
- Oceń książkę

- Księgarnia internetowa
- Lubię to! » Nasza społeczność

SPIS TREŚCI

Przedmowa **19**

Wstęp **25**

Wstęp (sprzed wieków) **29**

O autorze **31**

Rozdział 1 Czysty kod **33**
Niech stanie się kod… 34
W poszukiwaniu doskonałego kodu… 35
Postawa 36
Największa zagadka 37
Sztuka czystego kodu? 38
Co to jest czysty kod? 38
Zebranie wszystkiego w całość 41
Dlaczego powinniśmy być czyści? 42
Produktywność 44
Łatwość pracy z kodem 46
Czytamy więcej, niż piszemy 47
Zasada skautów 48

Kup ksi k Pole ksi k

CZĘŚĆ I Kod 49

Rozdział 2 Wyczyść ten kod! 51
Proces czyszczenia 61
Słowo końcowe 66
Epilog: Bob z przyszłości bawi się modelem Grok3 66
Słowo końcowe dla epilogu 69

Rozdział 3 Podstawowe zasady 70
Wszystko krótkie, dobrze nazwane, zorganizowane i poukładane 71
Funkcje 71
Bardziej znaczący przykład 72
Możliwość niezależnego wdrażania 93
Słowo końcowe 94

Rozdział 4 Sensowne nazwy 95
Używaj nazw wyrażających zamysł 96
Twórz systemy nazw 97
Unikaj dezinformacji 98
Twórz wyraźne różnice 99
Twórz nazwy, które można wymówić 101
Korzystaj z nazw łatwych do wyszukania 101
Używaj nazw o odpowiedniej długości 102
Unikaj kodowania 104
Używaj odpowiednich części mowy 106
Rozważ użycie parametrów nazwanych 107
Nie bądź dowcipny 107
Wybieraj jedno słowo na pojęcie 107
Korzystaj z nazw dziedziny rozwiązania 108
Korzystaj z nazw dziedziny problemu 108
Dostarczaj znaczący kontekst 109
Nie dodawaj nadmiarowego kontekstu 111
Słowo końcowe 111

Rozdział 5 Komentarze 113
Kompensowanie błędów 114
Ukryte lub niezrozumiałe komentarze 114
Kłamliwe komentarze 115
Komentarze zbyt związane z kodem 116

Kup ksi k Pole ksi k

Komentarze nie są szminką dla złego kodu 116
Przedstawiaj swój zamiar w kodzie 116
Dobre komentarze 117
Komentarze prawne 117
Komentarze informacyjne 117
Wyjaśnianie zamierzeń 118
Wyjaśnianie 119
Ostrzeżenia o konsekwencjach 119
Wzmocnienie 120
Komentarze Javadoc (i im podobne) w publicznym API 121
Złe komentarze 121
Bełkot 121
Powtarzające się komentarze 122
Mylące komentarze 123
Nadmiar i brak dokładności 123
Komentarze wymagane 125
Komentarze dziennika 125
Komentarze wprowadzające szum informacyjny 126
Przerażający szum 128
Komentarze TODO 128
Nie używaj komentarzy, jeżeli można użyć funkcji lub zmiennej 129
Znaczniki pozycji 129
Atrybuty i dopiski 129
Zakomentowany kod 130
Komentarze HTML 131
Informacje nielokalne 131
Nadmiar informacji 132
Nieoczywiste połączenia 132
Nagłówki funkcji 132
Komentarze Javadoc w niepublicznym kodzie 133
Przykład 133
Słowo końcowe 136

Rozdział 6 Formatowanie 137
Przeznaczenie formatowania 138
Formatowanie pionowe 138
Pionowe odstępy pomiędzy segmentami kodu 140
Gęstość pionowa 141
Odległość pionowa 141

Kup ksi k Pole ksi k

Formatowanie poziome 146
  Poziome odstępy i gęstość 147
  Rozmieszczenie poziome 148
  Wcięcia 149
Zasady zespołowe 151
Zasady formatowania wujka Boba 152

Rozdział 7 Czyste funkcje 154
Małe funkcje! 155
  Dobrze napisana opowieść 156
  Jeden poziom abstrakcji w funkcji 156
Czytanie kodu od góry do dołu — zasada zstępująca 157
  Plątanina 159
Instrukcje switch 159
Czyste funkcje — wnikliwe spojrzenie 161
  Funkcja powinna być kontekstowa 161
  Funkcja powinna mieć odpowiednią nazwę 162
  Nazwa powinna być opisowa 162
  Nazwa powinna być wygodna 164
  Funkcja powinna być odizolowana 164
  Funkcja powinna być jednorodna 167
  Funkcja powinna być czysta 169
  Częściowa czystość 171
Słowo końcowe 172

Rozdział 8 Heurystyki funkcji 173
Argumenty funkcji 174
  Zmienna liczba argumentów 175
  Więcej niż trzy? 175
  Argumenty nazwane 175
  Argumenty znacznikowe 176
  Argumenty wyjściowe 177
  Kody błędów 178
Rozdzielanie poleceń i zapytań 178
Stosowanie wyjątków zamiast zwracania kodów błędów 179
  Na własne ryzyko 180
  Wyodrębnienie bloków try/catch 181
  Obsługa błędów jest jedną operacją 181
  Przyciąganie zależności w kodach błędów 182

Kup ksi k Pole ksi k

Nie powtarzaj się 183
Prosty powtórzony kod 183
Podobny kod 184
Powtórzenia pętli 187
Powtórzenia przypadkowe a zamierzone 190
Skutki uboczne 190
Nie jesteśmy w tym dobrzy 191
Języki funkcyjne 192
Języki zorientowane obiektowo 193
Programowanie strukturalne 194
Sekwencja 195
Wybór 195
Pętla 195
To zbyt wiele do zapamiętania 196
Słowo końcowe 197

Rozdział 9 Metoda czyszczenia 198
Spraw, by było dobre 199
Przykład 201
Projekt i architektura 214
Słowo końcowe 220

Rozdział 10 Jedna rzecz 221
Refaktoryzacja — wydzielanie metody 222
To nie powinno wzbudzać emocji 224
1. Utonięcie 224
2. Małe funkcje nie zaciemniają celu 226
3. Wydajność 226
4. Skakanie po kodzie 226
5. Zaplątanie 227
A czym są duże funkcje? 227
Wydzielanie i klasy 242
Słowo końcowe 246

Rozdział 11 Bądź uprzejmy 247
Metafora gazety 249
Bądź miły 250
Zasada zstępująca — jeszcze raz 251
Huśtawka poziomów abstrakcji 252
Tak piszemy, ale nie chcemy tak czytać 252

Kup ksi k Pole ksi k

Rozdział 12 Obiekty i struktury danych 253

Czym jest obiekt? 254
Abstrakcja danych 255
Antysymetria danych i obiektów 257
Prawo Demeter 259
  Wraki pociągów 260
  Hybrydy 260
  Ukrywanie struktury 261
Obiekty transferu danych 262
  Obiektowo-relacyjna niezgodność impedancji 263
  Stosowanie obiektów i struktur danych 263
Instrukcje switch 264
  Rozwiązanie zorientowane obiektowo 267
  Nie tak szybko, Kowalski 268
Kompromisy rozwiązań zorientowanych obiektowo i proceduralnych 268
A co z wydajnością? 269
Słowo końcowe 270

Rozdział 13 Czyste klasy 271

Klasy i moduły a pliki 271
Co powinna zawierać klasa? 272
  Projektowanie klasy 273
  Heurystyki i charakterystyki 274
  Kiedy klasa jest zbyt duża? 275
  Wytyczne w kodzie 277
  Gdzie chowają się powody zmian? 278
  Rozwiązywanie problemu 279
  Zbyt otwarte klasy 281
  Czy powinniśmy coś teraz zrobić? 282
  A co teraz? 282
  Zamknięte, zwarte klasy z jedną odpowiedzialnością 285
  Gdy wytyczne się zmieniają 287
  Czy to nie przesada? 287
  Łatwiejsze testowanie 289
  Wykorzystanie sztucznej inteligencji 289
  Będzie się mylił 290

Kup ksi k   Pole ksi k

Rozdział 14 Metodyki testowania 292
Metodyka 1. Programowanie sterowane testami (TDD) 294
Trzy prawa TDD 294
Metodyka 2. Testuj i zatwierdzaj albo cofaj (TCR) 295
Metodyka 3. Małe pakiety 296
Projektowanie 296
Metodyka 297
Żmudne, nudne i wolne 297
Debugowanie 297
Dokumentacja 298
Niezawodność 299
Projektowanie 299
Powtórzenie 300
Anioły i demony 300
Ujarzmienie demona 301
Utrudnienia i luki 301
Koszty i następstwa 303
Zachowanie czystości testów 303
Testy zwiększają możliwości 304

Rozdział 15 Czyste testy 305
Języki testowania specyficzne dla domeny 308
Asercje złożone 308
Złożone wyniki testów 308
Podwójny standard 310
Jedna asercja na test 310
Jedna koncepcja na test 311
FIRST 311
Szybkie (Fast) 311
Niezależne (Isolated) 311
Powtarzalne (Repeatable) 312
Samokontrolujące się (Self-Validating) 312
O czasie (Timely) 312
Projektowanie testów 312
Słowo końcowe 313

Rozdział 16 Testy akceptacyjne 314
Metodyka testów akceptacyjnych 315
Metodyka 316
Ciągłe budowanie 317
Słowo końcowe 317

Kup ksi k Pole ksi k

Rozdział 17 Sztuczna inteligencja, duże modele językowe i Bóg wie, co jeszcze 318
Programowanie za pomocą promptów 320
Początki 324
Strzelanie w ciemno 325
Słowo końcowe 328

CZĘŚĆ II Projekt 329

Rozdział 18 Prostota projektu 331
YAGNI 333
Pokrycie kodu testami 334
Cel asymptotyczny 334
Projekt? 334
Zwiększenie wyrazistości 335
Bazowa abstrakcja 336
Testy: druga część problemu 337
Ograniczenie duplikacji 338
Przypadkowa duplikacja 339
Zmniejszanie rozmiaru 339
Prosta konstrukcja 339

Rozdział 19 Zasady SOLID 341
Zasada jednej odpowiedzialności (SRP) 343
Przypadkowa duplikacja 344
Rozwiązania 346
Wyższe poziomy 347
Zasada otwarte – zamknięte (OCP) 348
Eksperyment myślowy 349
Kontrola kierunku 352
Ukrywanie informacji 352
Wnioski 353
Zasada podstawień Liskov (LSP) 353
Zasada LSP i architektura 354
Agregator usług taksówkowych 355
Zasada rozdzielania interfejsów (ISP) 356
Zasada ISP i język 357
Zasada ISP i architektura 358

Kup ksi k Pole ksi k

Zasada odwrócenia zależności (DIP) 359
Stabilne abstrakcje 361
Fabryki 362
Komponenty konkretne 364

Rozdział 20 Zasady komponentów 365
Komponenty 366
Krótka historia komponentów 366
Relokacje 369
Konsolidatory 369
Spójność komponentów 371
Zasada istotności numeru wydania (REP) 371
Zasada wspólnego domknięcia (CCP) 372
Zasada wspólnego użycia (CRP) 373
Diagram napięć dla zasad spójności komponentów 375
Wnioski 376
Łączenie komponentów 376
Zasada acyklicznych zależności (ADP) 376
Zasada stabilnych zależności (SDP) 382
Zasada stabilnych abstrakcji (SAP) 387
Słowo końcowe 391

Rozdział 21 Ciągłe projektowanie 393
Ciągła zmiana 394
Ciągłe projektowanie 396
Zasady ciągłego projektowania 396
Jasność 397
Zwięzłość 402
Sprawdzalność 410
Spójność 418
Kiedy projektujemy? 421
Projektowanie z wyprzedzeniem 422
Przygotowywanie się do pracy 422
Rozpoczynanie pracy 423
Wykonywanie pracy 423

Rozdział 22 Współbieżność 424
W jakim celu stosować współbieżność? 425
Mity i nieporozumienia 426
Wyzwania 427

Kup ksi k Pole ksi k

Zasady obrony współbieżności 428
Zasada jednej odpowiedzialności 428
Wniosek: ograniczenie zakresu danych 428
Wniosek: korzystanie z kopii danych 429
Wniosek: wątki powinny być na tyle niezależne, na ile to tylko możliwe 429
Poznaj swój język i bibliotekę 430
Kolekcje bezpieczne dla wątków 430
Poznaj modele wykonania 430
Producent – konsument 431
Czytelnik – pisarz 431
Ucztujący filozofowie 432
Uwaga na zależności pomiędzy synchronizowanymi metodami 432
Tworzenie małych sekcji synchronizowanych 433
Pisanie prawidłowego kodu wyłączającego jest trudne 433
Testowanie kodu wielowątkowego 434
Traktuj przypadkowe awarie jako potencjalne problemy z wielowątkowością 435
Na początku uruchamiaj kod niekorzystający z wątków 435
Kod wielowątkowy powinien dać się włączać 435
Kod wielowątkowy powinien dać się dostrajać 436
Uruchamiaj więcej wątków, niż masz do dyspozycji procesorów 436
Uruchamiaj testy na różnych platformach 436
Uzbrajaj kod w elementy próbujące wywołać awarie i wymuszające awarie 436
Instrumentacja ręczna 437
Instrumentacja automatyczna 438
Nowości w 2025 roku i raport z terenu 439
Integralność danych 439
Słowo końcowe 444

CZĘŚĆ III Architektura 447

Rozdział 23 Dwie wartości oprogramowania 449
Otwarte furtki 450

Rozdział 24 Niezależność 452
Przypadki użycia 453
Operacje 453
Programowanie 454
Wdrażanie 454
Pozostawianie wyboru 454

Kup książkę Poleć książkę

Rozdział 25 Granice architektury 455

Jakie linie rysujesz i kiedy? 456

Architektura z wtyczkami 458

Studium przypadku: FitNesse 459

Słowo końcowe 461

Rozdział 26 Czyste granice 462

Zewnętrzny framework IoT: wiele granic 463

Granice interfejsu użytkownika/aplikacji 467

- Zasady SOLID i architektura heksagonalna 469
- Granice odkrywania i nauki 471
- Korzystanie z nieistniejącego kodu 474

Czyste granice 475

Rozdział 27 Czysta architektura 476

Zasada zależności 477

- Encje 478
- Przypadki użycia 478
- Adaptery interfejsu 479
- Frameworki i sterowniki 479
- Tylko cztery okręgi? 479
- Przekraczanie granic 480
- Jakie dane przekraczają granice? 480
- Typowy scenariusz 481

Słowo końcowe 482

CZĘŚĆ IV Rzemiosło 483

„Ogromna liczba” 484

Osiem dekad 484

- Nerdy i zbawiciele 488
- Zła sława 488
- Wzory do naśladowania i złoczyńcy 490
- Rządzimy światem 491
- Katastrofy 491

Przysięga 493

Rozdział 28 Szkoda 495

Nie szkodzić społeczeństwu 496

Szkoda przez sposób działania 497

Kup ksi k Pole ksi k

Nie szkodzić strukturze 499
Elastyczność 501
Testy 502

Rozdział 29 Bezbłędne działanie i struktura 504
Spraw, by było dobre 505
Jaka struktura jest dobra? 506
Macierz Eisenhowera 507
Programiści są interesariuszami 508
Dawaj z siebie wszystko 510

Rozdział 30 Powtarzalny dowód 512
Dijkstra 512
Dowody poprawności 513
Programowanie strukturalne 515
Dekompozycja funkcjonalna 517
Programowanie sterowane testami i inne techniki 518

Rozdział 31 Krótkie cykle 520
Historia kontroli wersji kodu źródłowego 520
Karty perforowane 520
Ciągła integracja 525
Krótkie cykle 526
Ciągła integracja 526
Gałęzie a przełączniki 527
Ciągłe wdrażanie 529
Ciągłe budowanie 530

Rozdział 32 Nieustanne ulepszenia 531
Pokrycie testami 531
Testowanie mutacyjne 532
Stabilność semantyczna 533
Czyszczenie 533
Dzieło 534

Rozdział 33 Utrzymanie wysokiej produktywności 535
Lepkość 536
Budowanie 536
Testowanie 536

Kup książkę Poleć książkę

Debugowanie 537
Wdrażanie 538
Zarządzanie rozproszeniem uwagi 538
Spotkania 538
Muzyka 539
Nastrój 539
Flow 540
Zarządzanie czasem 540

Rozdział 34 Praca zespołowa 542
Programowanie zespołowe 542
Otwarte/wirtualne biuro 543

Rozdział 35 Uczciwe szacowanie 545
Kłamstwa 546
Uczciwość, konkretność i precyzja 546
Nauka na moich błędach 547
Historia nr 1: wektory 547
Historia nr 2: pCCU 549
Lekcja 550
Konkretność i precyzja 550
Agregacja 551
Uczciwość 552
Naciski 553

Rozdział 36 Szacunek do kolegów z zespołu 555

Rozdział 37 Ciągła nauka 556

Posłowie 559

Dodatek Debata na temat czystego kodu 563
Wprowadzenie 564
Długość metody 565
Długość metody — podsumowanie 576
Komentarze 577
Komentarze — podsumowanie 589
Klasa PrimeGenerator według Johna 590
Bajka o dwóch programistach 595

Kup ksi k Pole ksi k

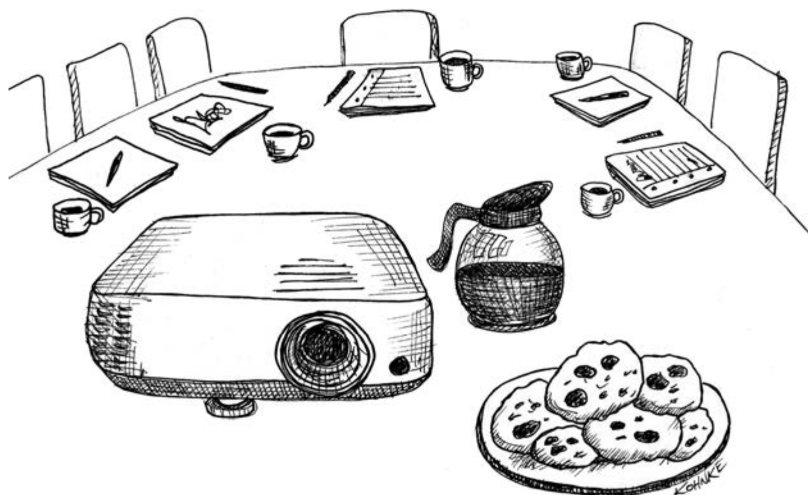
Klasa PrimeGenerator2 według Boba 598
Programowanie sterowane testami 601
Programowanie sterowane testami — podsumowanie 611
Uwagi końcowe 612

Bibliografia 614

Skorowidz 617

Kup ksi k Pole ksi k

3 PODSTAWOWE ZASADY



Jeśli przeczytasz ten rozdział i zapamiętasz wszystkie zasady w nim opisane, to w 90% będziesz gotowy czyścić kod. Koncepcje tu zawarte są stare i dobrze sprawdzone. Przetrwały próbę czasu i każdy programista powinien je znać.

Pamiętaj, że nie są to sztywne reguły. To nie są prawa, których trzeba bezwzględnie przestrzegać. Są to bardziej wytyczne. Każda z nich powinna być stosowana odpowiednio do kontekstu i problemu, z którym przychodzi nam się zmierzyć, i powinniśmy móc swobodnie podejmować wszelkie działania niezbędne do rozwiązania tego problemu.

Mimo że nie są jedyne, moim zdaniem przedstawione tu zasady wynikają z dobrze oczyszczonego kodu.

Kup ksi k Pole ksi k

Jeśli podczas lektury tego rozdziału napotkasz koncepcje, które są obce lub nowe, bądź pewien, że wszystko zostanie wyjaśnione w kolejnych rozdziałach. Gdy skończysz czytać tę książkę, możesz chcieć wrócić do tego rozdziału.

Wszystko krótkie, dobrze nazwane, zorganizowane i poukładane

Wystarczy przeczytać tytuł tego podrozdziału, by poznać istotę czystego kodu.

Staraj się, aby wszystko było małe. Wybieraj dla tych małych elementów takie nazwy, aby były zrozumiałe dla innych programistów. Zdefiniuj i utrzymuj strukturę i organizację, które dobrze oddają to, co oprogramowanie robi. Dbaj o porządek elementy, tak aby jedna koncepcja następowała po drugiej w sensowny i racjonalny sposób.

Pamiętaj, że to, jak Ty rozumiesz kod, zapewne wpłynie na Twoje starania w doborze dobrych nazw i struktur. Dlatego zawsze staraj się wejść w skórę kogoś, kto nie rozumie, co kod robi.

Funkcje

Funkcje powinny być krótkie. Większość powinna mieć tylko kilka linii. To pozwala nadać im ładne, opisowe nazwy i wspiera podział odpowiedzialności. Ponadto możemy mieć pewność, że każda funkcja robi jedną i tylko jedną rzecz.

Zacznijmy od nazw. Gdy widzisz fragment kodu, który robi coś, co można sensownie nazwać, to taki fragment kodu powinien zostać przeniesiony do funkcji, której nazwa oddaje, co ten fragment kodu robi. Nazwa powinna być czasownikiem, który mówi, co robi funkcja, i dzięki któremu sekwencję wywołującą tę funkcję będzie się czytać „jak dobrze napisaną prozę”[1].

W ramach przykładu przyjrzyj się funkcji pokazanej poniżej, wziętej z biblioteki `JCommon`. Znajduje ona datę podanego dnia tygodnia przed podaną datą. Dni tygodnia są oznaczane za pomocą liczb całkowitych, od 1 (niedziela) do 7 (sobota).

```
public static Date getPreviousDayOfWeek(int weekday, Date from) {
   // Sprawdzanie argumentów...
   if (!isValidWeekdayCode(weekday)) {
      throw new IllegalArgumentException(
         "Niepoprawny kod dnia tygodnia"
      );
   }
   // Znajdowanie daty...
   final int adjust;
```

---

[1] Grady Brooch, prywatny e-mail.

Kup książkę Poleć książkę

```
   final int baseDOW = from.getDayOfWeek();
   if (baseDOW > weekday) {
      adjust = Math.min(0, weekday - baseDOW);
   } else {
      adjust = -7 + Math.max(0, weekday - baseDOW);
   }

   return Date.addDays(adjust, from);
}
```

Są tutaj przynajmniej dwa fragmenty, które moglibyśmy wyodrębnić do mniejszych, lepiej nazwanych funkcji. Dzięki temu kod będzie prostszy.

```
public static Date getPreviousDayOfWeek(int weekday, Date from) {
   checkWeekdayArgument(weekday);
   return addDays(-daysBefore(weekday, from), from);
}
private static void checkWeekdayArgument(int weekday) {
   if (!isValidWeekdayCode(weekday))
      throw new IllegalArgumentException(
         "Niepoprawny kod dnia tygodnia");
}

private static int daysBefore(int targetWeekday, Date from) {
   int fromWeekday = from.getDayOfWeek();
   int diff = fromWeekday - targetWeekday;
   return diff + (diff > 0 ? 0 : 7);
}
```

Zwróć uwagę, jak się czyta uproszczoną funkcję `getPreviousDayOfWeek`. Jest krótka i oczywista. Jeśli musisz wiedzieć więcej, wystarczy, że przewiniesz w dół i spojrzysz na wyodrębnione funkcje. Jednak w większości przypadków nie będziesz musiał wiedzieć więcej. Po prostu spojrzysz na tę funkcję, zgodzisz się z nią i przejdziesz dalej. Może tego kodu nie będzie się czytać jak powieści Crichtona, ale lektura będzie całkiem przyjemna.

BOB Z PRZYSZŁOŚCI: Nie lubię stawiać znaku minusa z przodu. Wolałbym utworzyć funkcję dla odejmowania dni, `subtractDays`, by pozbyć się minusa. To drobnostka, ale małe rzeczy robią różnicę.

Bardziej znaczący przykład

Poniżej przedstawiam system rezerwacji salek konferencyjnych Wujka Boba. Przeczytaj go pobieżnie. Nie jest bardzo skomplikowany (chyba).

```
——Statement.java——
package ubConferenceCenter;

import java.util.ArrayList;
import java.util.List;
```


Kup ksi k   Pole ksi k

```
public class Statement {
   public enum CatalogItem {SMALL_ROOM, LARGE_ROOM,
                            PROJECTOR, COFFEE, COOKIES}

   public record RentalItem(CatalogItem type,
                            int days,
                            int unitPrice,
                            int price,
                            int tax) {
   }

   public record Totals(int subtotal, int tax) {
   }

   private String customerName;
   private int subtotal = 0;
   private int tax = 0;
   private List<RentalItem> items = new ArrayList<>();

   public Statement(String customerName) {
      this.customerName = customerName;
   }

   public void rent(CatalogItem item, int days) {
      int unitPrice = switch (item) {
         case SMALL_ROOM -> 100;
         case LARGE_ROOM -> 150;
         case PROJECTOR -> 50;
         case COFFEE -> 10;
         case COOKIES -> 15;
      };

      boolean eligibleForDiscount = switch (item) {
         case SMALL_ROOM, LARGE_ROOM -> days == 5;
         case PROJECTOR, COFFEE, COOKIES-> false;
      };

      int price = unitPrice * days;

      if (eligibleForDiscount) price = (int) Math.round(price * .9);

      subtotal += price;
      int thisTax = switch (item) {
         case SMALL_ROOM, LARGE_ROOM, PROJECTOR ->
            (int) Math.round(price * .05);
         case COFFEE,COOKIES -> 0;
      };
      tax += thisTax;
      items.add(new RentalItem(item, days, unitPrice, price, thisTax));
   }

   public RentalItem[] getItems() {
      List<RentalItem> items = new ArrayList<>(this.items);
      boolean largeRoomFiveDays = items.stream().anyMatch(
         item -> item.type() == CatalogItem.LARGE_ROOM && item.days() == 5);
      boolean coffeeFiveDays = items.stream().anyMatch(
         item -> item.type() == CatalogItem.COFFEE && item.days() == 5);
```


Kup ksi k   Pole ksi k

```
        if (largeRoomFiveDays && coffeeFiveDays)
            items.add(new RentalItem(CatalogItem.COOKIES, 5, 0, 0, 0));
        return items.toArray(new RentalItem[0]);
    }

    public String getCustomerName() {
        return customerName;
    }

    public Totals getTotals() {
        return new Totals(subtotal, tax);
    }
}
```

Jestem pewien, że zrozumiałeś, co ten kod robi. Użytkownicy mogą wynajmować małe salki, duże salki, ekspresy do kawy, projektory i ciastka (jak można wynająć ciastka?). Wynajem małej salki kosztuje 400 złotych za dzień. Za dużą salkę trzeba zapłacić 600 złotych. Można dostać 10% opustu przy wynajmie na cały tydzień. Do ceny należy doliczyć 5% podatku na wszystkie niespożywcze rzeczy. Ekspres do kawy to koszt 40 złotych za dzień, a ciastka 60 złotych. Projektor kosztuje 200 złotych za dzień. Przy wynajmie dużej salki i ekspresu do kawy na cały tydzień jeden dzień ciastek będzie gratis.

Klasa `Statement` reprezentuje zestawienie sprzedaży, które obejmuje każdy wynajęty przedmiot, cenę przedmiotu, liczbę dni wynajmu, sumę częściową oraz podatek. Klasa `Statement` zawiera również sumy częściowe i podatek dla wszystkich rzeczy.

Łatwizna.

To nie jest zbyt trudne do zrozumienia, ale można mieć wrażenie, że zostało to poskładane na szybko. Kod jest nieco niewygładzony, ale nie tragiczny. Moglibyśmy go trochę oczyścić, ale czy byłoby to warte naszych wysiłków?

Ale pozwól, że zadam Ci pytanie: czy widziałeś kiedyś system, który na początku był prosty, a z czasem stawał się coraz bardziej skomplikowany? Jeśli pracujesz w branży więcej niż rok, *jestem pewny*, że tak. A jeśli pracujesz w branży ponad pięć lat, to prawdopodobnie widziałeś system nieco zaniedbany, który z czasem przekształcił się w okropny bałagan, który był piekłem na ziemi dla programistów i całej organizacji.

Urzeczywistnijmy trochę przykład z rezerwacją salek konferencyjnych. Jak zapewne przypuszczasz, jego kod nie był taki od początku. Najpierw za pomocą tego programu można było wynająć tylko jedną małą salkę. Później kilka małych salek. Następnie duże salki za inną stawkę. Na późniejszych etapie doszła kawa, bez podatku. Potem dodałem upust za wynajem na cały tydzień. A na końcu ciastka i bonus za wynajęcie dużej salki na cały tydzień.

Innymi słowy: ten program się *rozrastał*.

Załóżmy, że właśnie wróciliśmy ze spotkania całego zespołu, na którym prezes przedstawił oszałamiający raport o tym, jak firma się rozwija. Są nowe rynki, nowe

Kup ksi k Pole ksi k

regiony i nowe przepisy, których trzeba przestrzegać. Będą nowe upusty i nowe promocje oraz nowe regulacje podatkowe. To świetna wiadomość dla firmy, ale Ty patrzysz na tę małą funkcję `rent` i myślisz „będzie dalej rosnąć”, nieprawdaż? A jak się ona rozrośnie, to stanie się jeszcze bardziej zawiła i trudna do zrozumienia. Będzie się psuć jak gnijące mięso.

Musimy temu zapobiec. Zastanówmy się, jak przygotować się na nadchodzący rozwój biznesu i pozwolić tej funkcji rozwijać się bez gnicia od środka. Uporządkujmy najpierw kod[2].

Spójrz ponownie na funkcję `rent` i zadaj sobie pytanie, co Ci się w niej nie podoba. Moim zdaniem jest trochę za długa i nieuporządkowana. Ta funkcja robi wiele rzeczy, które mogę rozdzielić i umieścić w osobnych funkcjach, które wykonują jedno zadanie.

Zatem najpierw przeprowadźmy refaktoryzację polegającą na wyodrębnieniu metod, aby utworzyć funkcje, które robią jedną rzecz, zebrać razem rzeczy, które są powiązane, i oddzielić rzeczy, które są różne[3]:

```
——Statement.java——
…
public void rent(CatalogItem item, int days) {
   int unitPrice = getUnitPrice(item);
   int price = calculatePrice(item, days, unitPrice);
   int thisTax = getTax(item, price);
   items.add(new RentalItem(item, days, unitPrice, price, thisTax));
   subtotal += price;
   tax += thisTax;
}

private int getUnitPrice(CatalogItem item) {
   return switch (item) {
      case SMALL_ROOM -> 100;
      case LARGE_ROOM -> 150;
      case PROJECTOR -> 50;
      case COFFEE -> 10;
      case COOKIES -> 15;
   };
}

private int calculatePrice(CatalogItem item, int days, int unitPrice) {
   boolean eligibleForDiscount = isEligibleForDiscount(item, days);
   int price = unitPrice * days;
   if (eligibleForDiscount) price = (int) Math.round(price * .9);
   return price;
}

private boolean isEligibleForDiscount(CatalogItem item, int days) {
   return switch (item) {
      case SMALL_ROOM, LARGE_ROOM -> days == 5;
```

[2] [TIDY].

[3] Zasada jednej odpowiedzialności.

Kup ksi k   Pole ksi k

```
      case PROJECTOR, COFFEE, COOKIES-> false;
   };
}
private int getTax(CatalogItem item, int price) {
   return switch (item) {
      case SMALL_ROOM, LARGE_ROOM, PROJECTOR ->
         (int) Math.round(price * .05);
      case COFFEE,COOKIES -> 0;
   };
}
```

BOB Z PRZYSZŁOŚCI: W ostatnim rozdziale użyłem modelu Grok3 w celu ulepszenia modułu `fromRoman` i pomyślałem, że tu zrobię to samo. Poprosiłem więc model Grok3 o wprowadzenie ulepszeń do mojej pierwotnej implementacji. Nie zaskoczyło mnie, że rozwiązanie zaproponowane przez niego było niemal identyczne jak moje.

Możliwe, że patrzysz na ten kod i myślisz, iż jest go więcej. Ale nie ma więcej kodu, który się *wykonuje*. Jest po prostu więcej nazw i większa struktura. Dzięki tej dodatkowej strukturze kod będzie mógł się rozrastać. Na przykład jeśli przepisy podatkowe staną się bardziej skomplikowane, to nie funkcja `rent` będzie rosnąć, ale prawdopodobnie nowa logika zostanie wprowadzona do funkcji `getTax`.

To przykład zastosowania zasady jednej odpowiedzialności (SRP, ang. *Single Responsibility Principle*)[4]. Klienci naszego systemu, którzy są najbardziej zainteresowani podatkami, prawdopodobnie będą jedynymi osobami, które będą prosić o zmiany w funkcji `getTax`. Natomiast interesariusze zainteresowani najbardziej upustami mogą prosić o zmiany w funkcjach `calculatePrice` i `isEligibleForDiscount`.

To bardzo pomocne, ponieważ po tej zmianie ryzyko zepsucia funkcji obsługującej upusty po wprowadzeniu zmian w funkcji obsługującej podatki jest minimalne i podobnie mało prawdopodobne jest, że zepsujemy obsługę podatków po wprowadzeniu zmian w obliczaniu cen. Dopóki będziemy mogli utrzymywać te odpowiedzialności osobno, kod będzie bezpieczniejszy podczas wprowadzania zmian.

Jeżeli zignorujemy zasadę jednej odpowiedzialności, będziemy narażeni na objawy *kruchości*. Kruchy system psuje się w niespodziewany sposób, na przykład gdy interesariusze proszą o zmiany w obsłudze podatków i po ich wprowadzeniu przypadkowo psujemy obsługę upustów. Takie przypadki przerażają interesariuszy, menedżerów i użytkowników, ponieważ to daje im wyobrażenie, co się dzieje pod maską, i nie podoba im się to.

Nasi interesariusze mają prawo oczekiwać, że zmiana w podatkach nie popsuje upustów. Gdy nie sprostamy tym oczekiwaniom, jedynym wnioskiem, jaki wysnują, będzie, że nie mamy kontroli nad systemem i nie wiemy, co robimy.

[4] Zobacz rozdział 19. „Zasady SOLID”.

Kup książkę Poleć książkę

Teraz spójrz na nowy kod. Co Ci się w nim nie podoba? Jedną kwestią, nad którą możemy się pochylić, są instrukcje `switch`.

Instrukcje warunkowe `switch` nie są z natury złe. Jednak jeśli liczba przypadków będzie prawdopodobnie rosnąć, to zostanie pogwałcona zasada otwarte – zamknięte (OCP, ang. *Open-Closed Principle*).

Zasada OCP sugeruje, że gdy dodajemy nową funkcjonalność, powinniśmy dodać tę nową funkcjonalność w jednym miejscu, a nie w kilku. Teraz, gdybyśmy musieli dodać nową pozycję w katalogu (`CatalogItem`), na przykład notatniki (`NotePads`), musielibyśmy wprowadzić zmiany w pięciu różnych miejscach. Możemy zmniejszyć tę liczbę do dwóch poprzez zamianę pozycji `CatalogItem` w struktury danych i zamienić instrukcje `switch` na dostępy do tych struktur danych:

```
——Statement.java——
…
public enum CatalogItem {
   SMALL_ROOM(100, .05),
   LARGE_ROOM(150, .05),
   PROJECTOR(50, .05),
   COFFEE(10, 0),
   COOKIES(15, 0);

   private double taxRate;
   private int unitPrice;

   CatalogItem(int unitPrice, double taxRate) {
      this.unitPrice = unitPrice;
      this.taxRate = taxRate;
   }
}
…
public void rent(CatalogItem item, int days) {
   int unitPrice = item.unitPrice;
   int price = calculatePrice(item, days, unitPrice);
   int thisTax = (int) Math.round(price * item.taxRate);
   items.add(new RentalItem(item, days, unitPrice, price, thisTax));
   subtotal += price;
   tax += thisTax;
}
…
```

Ten kod jest lepszy. Muszę przyznać, iż podoba mi się to, że `enum` to w Javie tak naprawdę klasa i że każdy enumerator to instancja. Teraz po dodaniu `NotePads` musimy jedynie zmienić enumerator i może funkcję `getItems`. Jednak zasada otwarte – zamknięte mówi, że powinniśmy izolować nowe zmiany od starych modułów. Zatem z punktu widzenia tej zasady będzie lepiej wyjąć wszystkie `enum` z modułu `Statement`, tak byśmy po dodaniu `NotePads` nie musieli nic zmieniać w module `Statement`.

```
——CatalogItem.java——
package ubConferenceCenter;
```


Kup ksi k   Pole ksi k

```
public enum CatalogItem {
   SMALL_ROOM(100, .05),
   LARGE_ROOM(150, .05),
   PROJECTOR(50, .05),
   COFFEE(10, 0),
   COOKIES(15, 0);

   public double taxRate;
   public int unitPrice;

   CatalogItem(int unitPrice, double taxRate) {
      this.unitPrice = unitPrice;
      this.taxRate = taxRate;
   }
}
```

Radzimy sobie całkiem nieźle. Wyodrębniliśmy enumeratory `CatalogItem` i pozbyliśmy się instrukcji `switch`. Przestrzegamy zasad jednej odpowiedzialności i otwarte – zamknięte. Ale jest jeszcze jedna zasada, która bije na alarm.

Które moduły powinniśmy skompilować ponownie po dodaniu `NotePads` do `CatalogItem`? Odpowiedź brzmi: `CatalogItem.java` (oczywiście) i prawdopodobnie[5] `Statement.java`, ponieważ zależy od `CatalogItem.java`. A teraz zastanów się, czy moduł `Statement.java` *powinien być* skompilowany ponownie.

Uważam, że przynajmniej funkcja `rent` modułu `Statement.java` nie powinna być kompilowana ponownie po dodaniu `NotePads`, ponieważ żaden element tej funkcji nie musi nic wiedzieć o `NotePads`.

Jest to pogwałcenie zasady odwrócenia zależności (DIP, ang. *Dependency Inversion Principle*). Zasada ta mówi, że zasady wysokiego poziomu nie mogą bezpośrednio zależeć od niskopoziomowych szczegółów. Dodanie `NotePads` jest niskopoziomowym szczegółem, od którego funkcja `rent` nie powinna zależeć.

Możesz pomyśleć, że to niewielki problem. Ponowna kompilacja modułu jest szybka i łatwa, więc dlaczego trzeba się przejmować tym, że kompilowanych jest więcej modułów, niż to jest wymagane. W przypadku małych projektów to może być dobra decyzja. Ale w większych wysiłek związany z ponownym kompilowaniem i wdrażaniem może być spory. Dotyczy to szczególnie aplikacji napisanych w języku JavaScript, gdzie ponowna publikacja jest pobierana do przeglądarek. Ponadto programiści muszą pamiętać, które moduły od siebie zależą. Zatem załóżmy, że ten projekt jest na tyle duży, iż należy wziąć pod uwagę te kwestie. Jak możemy odwrócić zależności, by zmniejszyć narzut związany z ponową kompilacją, ponownym wdrożeniem i obciążeniem poznawczym?

---

[5] Kompilator języka Java jest często na tyle inteligentny, by określić, czy moduły nadrzędne muszą być ponownie kompilowane w wyniku zmian w modułach zależnych. Ale najbezpieczniej z góry przyjąć, że te moduły muszą zostać skompilowane ponownie.

Kup książkę Poleć książkę

Tutaj pomoże nam zasada jednej odpowiedzialności. Moduł Statement.java ma dwa główne zadania. Metoda rent dodaje instancje CatalogItem, a metoda getItems je zlicza, by obliczyć bonus w postaci ciastek. No cóż, nazwa getItems nie jest najlepsza. Musimy się tym bezzwłocznie zająć. Najpierw powinniśmy rozdzielić te dwa główne zadania.

Przede wszystkim musimy wyodrębnić rekord RentalItem do własnego modułu:

```
——RentalItem.java___
package ubConferenceCenter;

public record RentalItem(CatalogItem type,
                         int days,
                         int unitPrice,
                         int price,
                         int tax) {
}
```

Następnie tworzymy nowy moduł dla ItemList:

```
——ItemList.java——
package ubConferenceCenter;

import java.util.ArrayList;
import java.util.List;

public class ItemList {
   private List<RentalItem> items = new ArrayList<>();

   public void add(CatalogItem item, int days, int unitPrice,
                   int price, int thisTax) {
      items.add(new RentalItem(item, days, unitPrice, price, thisTax));
   }

   public RentalItem[] getItems() {
      boolean largeRoomFiveDays = items.stream().anyMatch(
         item -> item.type() == CatalogItem.LARGE_ROOM && item.days() == 5);
      boolean coffeeFiveDays = items.stream().anyMatch(
         item -> item.type() == CatalogItem.COFFEE && item.days() == 5);
      if (largeRoomFiveDays && coffeeFiveDays)
         items.add(new RentalItem(CatalogItem.COOKIES, 5, 0, 0, 0));
      return items.toArray(new RentalItem[0]);
   }
}
```

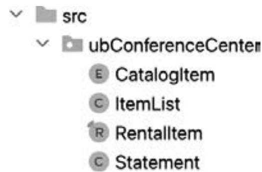
Będziemy musieli wrócić do tego modułu, ponieważ ta metoda getItems przyprawia mnie o gęsią skórkę. To wygląda na złamanie zasady jednej odpowiedzialności. Ale na razie wprowadzimy kilka zmian w module Statement.java:

```
…
public class Statement {
   …
   private ItemList items = new ItemList();
   …
```


Kup ksi k   Pole ksi k

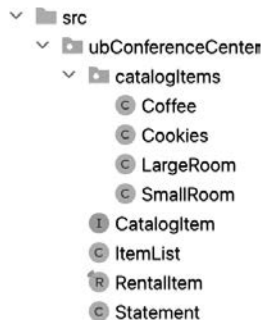
```
   public RentalItem[] getItems() {
      return items.getItems();
   }
   …
}
```

Możesz być zaniepokojony, że tniemy kod na zbyt wiele kawałków i że te wszystkie kawałki sprawią, iż kod będzie trudniejszy do zrozumienia. Ale założyliśmy, że ten kod będzie się rozrastał i że chcemy zrobić miejsce na nowe elementy. W każdym razie, gdy spojrzysz na układ folderów (widoczny na rysunku poniżej), przekonasz się, że tworzy czytelny plan rozwoju, zrozumiały dla każdego w organizacji.



Patrząc na to, możesz stwierdzić, że nazwy nie są najlepsze, ale wrócimy do tego później, gdy już będziemy wiedzieć lepiej, jak projekt będzie się zmieniał. W międzyczasie będziemy musieli jeszcze popracować nad narzutem związanym z ponownymi kompilacją i wdrożeniem.

W tym celu zmienimy `CatalogItem` z `enum` na `interface`, z każdym typem wyliczeniowym jako klasą pochodną. To tworzy wiele nowych klas i układ folderów wygląda teraz tak jak na rysunku poniżej.




```
——CatalogItem.java——
package ubConferenceCenter;

public interface CatalogItem {
   boolean isEligibleForDiscount(int days);
   int getUnitPrice();
   double getTaxRate();
   String getName();
}
```


Kup ksi k   Pole ksi k

To istotna zmiana w sposobie działania aplikacji. Teraz zamiast instrukcji `switch` i `if` sprawdzających enumeratory używamy metod polimorficznych interfejsu `CatalogItem`. Metody `getTaxRate` i `getUnitPrice` już znamy, ale pozostałe dwie są nowe. Możesz zobaczyć, jak zostały użyte w modułach `Statement.java` i `ItemList.java`:

```
——Statement.java——
package ubConferenceCenter;

public class Statement {
   …
   public void rent(CatalogItem item, int days) {
      int unitPrice = item.getUnitPrice();
      int price = calculatePrice(item, days, unitPrice);
      int thisTax = (int) Math.round(price * item.getTaxRate());
      items.add(item, days, unitPrice, price, thisTax);
      subtotal += price;
      tax += thisTax;
   }

   private int calculatePrice(CatalogItem item, int days, int unitPrice)
   {
      boolean eligibleForDiscount = item.isEligibleForDiscount(days);
      int price = unitPrice * days;
      if (eligibleForDiscount) price = (int) Math.round(price * .9);
      return price;
   }
   …
}

——ItemList.java——
package ubConferenceCenter;

import java.util.ArrayList;
import java.util.List;

public class ItemList {
   private List<RentalItem> items = new ArrayList<>();

   public void add(CatalogItem item, int days, int unitPrice,
                   int price, int thisTax) {
      items.add(new RentalItem(item.getName(), days,
                unitPrice, price, thisTax));
   }

   public RentalItem[] getItems() {
      boolean largeRoomFiveDays = items.stream().anyMatch(
         item -> item.type().equals("LARGE_ROOM") && item.days() == 5);
      boolean coffeeFiveDays = items.stream().anyMatch(
         item -> item.type().equals("COFFEE") && item.days() == 5);
      if (largeRoomFiveDays && coffeeFiveDays)
         items.add(new RentalItem("COOKIES", 5, 0, 0, 0));
      return items.toArray(new RentalItem[0]);
   }
}
```


Kup ksi k Pole ksi k

Zwróć uwagę, że pole type rekordu RentalItem zostało zmienione z enum na String. To było konieczne, ponieważ gdy pozbyliśmy się enum, potrzebowaliśmy jakiegoś tokena, by reprezentować typ CatalogItem, a typ String wydawał się najbardziej oczywistym wyborem. Jednak użycie tego łańcucha znaków wiązało się z poświęceniem bezpieczeństwa, jakie dają nam typy statyczne. Kompilator nie może sprawdzić, czy te nazwy są poprawne. Ta niewielka strata związana z bezpieczeństwem statycznego typu zawsze idzie w parze ze zmniejszeniem wysiłku ponownych kompilacji i wdrożenia oraz z izolacją niskopoziomowych szczegółów od zasad wysokiego poziomu.

```
——RentalItem.java——
package ubConferenceCenter;

public record RentalItem(String type,
                         int days,
                         int unitPrice,
                         int price,
                         int tax) {
}
```

Poza metodą getItems klasy ItemList ostatnie kilka listingów pokazanych powyżej przedstawia wysokopoziomowe zasady aplikacji. Będziemy musieli zrobić coś z metodą getItems. Na razie przyjrzyjmy się, jak niskopoziomowe szczegóły zostały wydzielone w klasach pochodnych CatalogItem:

```
——SmallRoom.java——
package ubConferenceCenter.catalogItems;

import ubConferenceCenter.CatalogItem;

public class SmallRoom implements CatalogItem {
   public String getName() {
       return "SMALL_ROOM";
   }

   public boolean isEligibleForDiscount(int days) {
      return days == 5;
   }

   public int getUnitPrice() {
      return 100;
   }

   public double getTaxRate() {
      return 0.05;
   }
}

——LargeRoom.java——
package ubConferenceCenter.catalogItems;

import ubConferenceCenter.CatalogItem;

public class LargeRoom implements CatalogItem {
   public boolean isEligibleForDiscount(int days) {
```


Kup ksi k Pole ksi k

```
      return days == 5;
   }

   public int getUnitPrice() {
      return 150;
   }

   public double getTaxRate() {
      return 0.05;
   }

   public String getName() {
      return "LARGE_ROOM";
   }
}

---Coffee.java---
package ubConferenceCenter.catalogItems;

import ubConferenceCenter.CatalogItem;

public class Coffee implements CatalogItem {
   public boolean isEligibleForDiscount(int days) {
      return false;
   }

   public int getUnitPrice() {
      return 10;
   }

   public double getTaxRate() {
      return 0;
   }

   public String getName() {
      return "COFFEE";
   }
}

---Cookies.java---
package ubConferenceCenter.catalogItems;

import ubConferenceCenter.CatalogItem;

public class Cookies implements CatalogItem {
   public boolean isEligibleForDiscount(int days) {
      return false;
   }

   public int getUnitPrice() {
      return 15;
   }

   public double getTaxRate() {
      return 0;
   }
```


Kup ksi k Pole ksi k

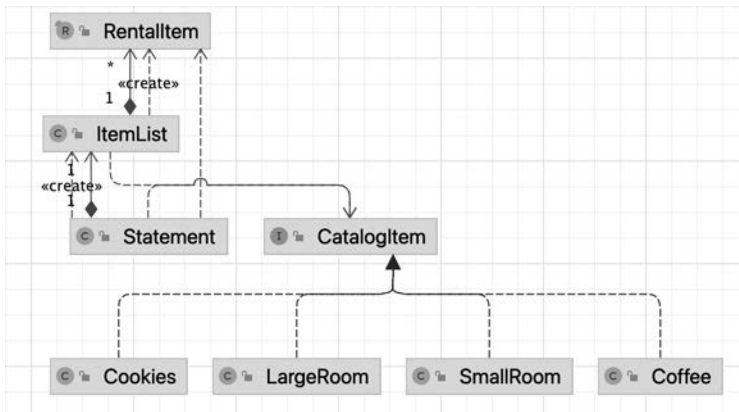
```
    public String getName() {
        return "COOKIES";
    }
}
```

Jedna z osób, które skomentowały pierwsze wydanie tej książki, spojrzała na taki kod i nazwała go „*po prostu okropny*”. Możliwe, że teraz też masz takie odczucia. Ale pamiętaj, że spodziewamy się, iż biznes będzie się rozwijał, i zdecydowaliśmy, że trzeba zrobić miejsce na nowe funkcjonalności.

Niektórzy mogą nas posądzić o łamanie zasady YAGNI (ang. *You aren't gonna need it*, nie będziesz tego potrzebować). Ale założeniem zasady YAGNI jest zadanie pytania: a jeśli będziemy tego potrzebować? Trzeba zadać sobie to pytanie, by ocenić koszt robienia miejsca w kodzie na rzeczy, których nie będziemy potrzebować.

Ale patrząc na entuzjazm prezesa i rynki, które zamierza zdobyć, oceniliśmy, że musimy zrobić miejsce. Zatem w naszym przypadku na pytanie związane z YAGNI postanowiliśmy odpowiedzieć: *tak, będziemy tego potrzebować*.

Dlaczego ten kod jest lepszy? Spójrzmy na poniższy rysunek, na którym widać schemat zależności przygotowany przez zintegrowane środowisko programistyczne (IDE).



Cztery klasy reprezentujące naszą wysokopoziomową logikę — `RentalItem`, `ItemList`, `Statement` i `CatalogItem` — są ze sobą powiązane kłębowiskiem zależności. Zajmiemy się tym później. Ale spójrz, jak dobrze są odizolowane niskopoziomowe szczegóły. Poprzednie zależności zostały odwrócone. Nic w obrębie wysokopoziomowych zasad nie zależy od niskopoziomowych szczegółów. Szczegóły niskiego poziomu zależą tylko od `CatalogItem`. Powtarzaj to sobie cały czas: *wysokopoziomowe zasady nie powinny zależeć od niskopoziomowych szczegółów*.

To znaczy, że jeżeli któryś z niskopoziomowych szczegółów się zmieni, nic w wysokopoziomowych zasadach nie musi być ponownie kompilowane i wdrażane.

Kup ksi k Pole ksi k

Zmiany w zasadach wysokiego poziomu nie będą też wymuszały ponownej kompilacji ani ponownego wdrażania szczegółów niskiego poziomu.

Pozwól, że powiem to inaczej: jeśli się zmieni cena jednostkowa ciastek, stawka podatkowa dla kawy albo upust za małe salki, żadna z klas wysokopoziomowych zasad nie będzie musiała być kompilowana i wdrażana ponownie. Zmiany tych niskopoziomowych szczegółów są zupełnie odizolowane od zasad wysokiego poziomu.

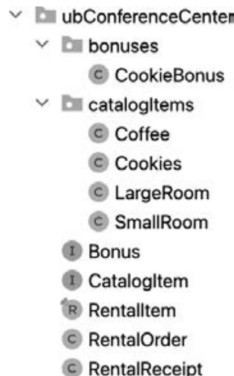
Tak naprawdę niskopoziomowe szczegóły można *włożyć* do wysokopoziomowych zasad. Albo przynajmniej niskopoziomowe szczegóły mogą stać się takimi wtyczkami.

Jednak teraz musimy coś zrobić z tym kłębowiskiem spowodowanym przez `ItemList`.

Początkowo nazwaliśmy tę klasę `ItemList`, ponieważ zawierała naszą listę elementów, a potrzebowaliśmy miejsca, by gdzieś umieścić ten dodatkowy kod. Teraz chcę wyjąć stamtąd ten dodatkowy kod i zrobić to samo co z instancjami `CatalogItem`. Utworzę więc interfejs `Bonus` i sprawię, że `CookieBonus` będzie go implementować. `ItemList` może zawierać listę instancji `Bonus` i po prostu przechodzić przez nie w pętli, by zsumować wszystkie bonusy. Czyli `ItemList` *domyka* kwestię wszystkich bonusów. To z kolei sugeruje, że klasa `ItemList` powinna mieć inną nazwę, coś w stylu `RentalReceipt`, gdyż reprezentuje końcowy stan zamówienia.

Ta zmiana nazwy sugeruje, że nazwa modułu `Statement` powinna być również zmieniona na `RentalOrder` lub coś podobnego. Zmiany nazw są ważne. Gdy rozbieramy projekt na części i zaczynamy robić miejsce na jego rozwój, dowiadujemy się więcej, co się dzieje w kodzie. Kiedy problem jest mały, nazwy nie muszą być spójne lub ogólnikowe, ponieważ nie ma zbyt wiele myśli, które trzeba śledzić w kodzie. Ale gdy problem rośnie, nazwy zyskują na znaczeniu, bo pomagają zachować jasność idei.

Po wprowadzeniu zmian struktura folderów wygląda jak na rysunku poniżej.

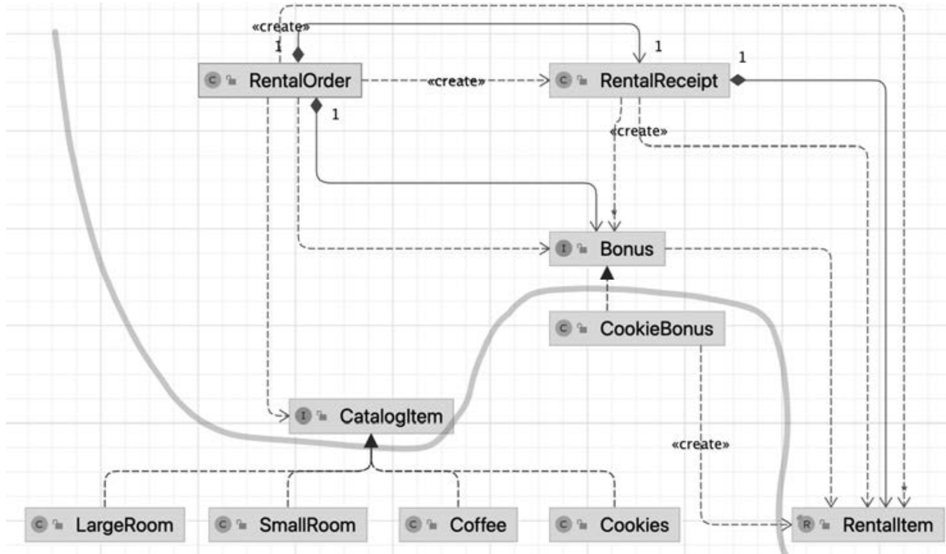


Kup ksi k Pole ksi k

Schemat programu pokazałem poniżej. `RentalOrder` zależy od `RentalReceipt`, ale nie na odwrót. Oba zależą od `Bonus`, a `CookieBonus` jest niezależnym niskopoziomowym szczegółem.

Szczegółem na najniższym poziomie jest rekord `RentalItem`. To trochę zastanawiające, ponieważ jest zwarty. Jeśli wprowadzimy w nim jakiekolwiek zmiany, niemal wszystko będzie musiało zostać ponownie skompilowane i wdrożone. Można to rozwiązać, ale myślę, że należy to zostawić na inny dzień.

Zwróć uwagę na krzywą linię graniczną. Ta linia dzieli projekt na dwie składowe: jedna zawiera wysokopoziomowe zasady, a druga zawiera niskopoziomowe szczegóły. Wszystkie zależności przekraczają tę linię w jednym kierunku, w stronę wyższego elementu. Ta linia to *granica architektoniczna* (granica warstw), która oddziela te dwa elementy składowe (wartswy), a zasada granic architektonicznych mówi, że zależności przekraczają tę granicę tylko w stronę wyższego poziomu.



Najpierw spójrzmy na kod w obrębie warstwy wysokiego poziomu:

```
——RentalOrder.java——
package ubConferenceCenter;

import java.util.ArrayList;
import java.util.List;

public class RentalOrder {
   public record Totals(int subtotal, int tax) {
   }

   private String customerName;
```


Kup ksi k Pole ksi k

```
   private int subtotal = 0;
   private int tax = 0;
   private RentalReceipt receipt = new RentalReceipt();
   private List<Bonus> bonuses = new ArrayList<>();

   public RentalOrder(String customerName) {
      this.customerName = customerName;
   }

   public void addBonus(Bonus bonus) {
      bonuses.add(bonus);
   }

   public void rent(CatalogItem item, int days) {
      int unitPrice = item.getUnitPrice();
      int price = item.getDiscountedPrice(days);
      int thisTax = (int) Math.round(price * item.getTaxRate());
      receipt.add(new RentalItem(item.getName(), days, unitPrice, price, thisTax));
      subtotal += price;
      tax += thisTax;
   }

   public RentalItem[] getReceipt() {
      return receipt.finalize(bonuses);
   }

   public String getCustomerName() {
      return customerName;
   }

   public Totals getTotals() {
      return new Totals(subtotal, tax);
   }
}
```

Zauważ, że prawie wszystkie reguły biznesowe wyjęliśmy z tego modułu. Zostały tylko obliczenia kwoty podatku i sumy całkowitej. Ale obliczenia upustów i bonusów zostały przeniesione w odpowiednie pochodne `CatalogItem` i `Bonus`:

```
——RentalReceipt.java——
package ubConferenceCenter;

import java.util.ArrayList;
import java.util.List;

public class RentalReceipt {
   private List<RentalItem> items = new ArrayList<>();

   public void add(RentalItem item) {
      items.add(item);
   }

   public RentalItem[] finalize(List<Bonus> bonuses) {
      List<RentalItem> finalItems = new ArrayList<>(this.items);
      finalItems.addAll(addBonuses(bonuses));
      return finalItems.toArray(new RentalItem[0]);
   }
```


Kup ksi k Pole ksi k

```
    private List<RentalItem> addBonuses(List<Bonus> bonuses) {
        List<RentalItem> bonusItems = new ArrayList<>();
        for (Bonus bonus : bonuses)
            bonus.checkAndAdd(items, bonusItems);
        return bonusItems;
    }
}
```

Klasa `RentalReceipt` przechodzi przez bonusy i je stosuje, ale nie zna żadnych szczegółów na temat tych bonusów. Tworzy końcowy rachunek z dodanymi bonusami do każdego elementu zamówienia.

Następnie mamy dwa interfejsy, które odwracają zależności między wysokopoziomową zasadą a niskopoziomowym szczegółem:

```
---CatalogItem.java---
package ubConferenceCenter;

public interface CatalogItem {
    int getDiscountedPrice(int days);
    int getUnitPrice();
    double getTaxRate();
    String getName();
}

---Bonus.java---
package ubConferenceCenter;

import java.util.List;

public interface Bonus {
    void checkAndAdd(List<RentalItem> items, List<RentalItem> bonusItems);
}
```

Moduł `RentalItem` jest ostatni w wysokopoziomowej warstwie. To prosta, zwarta struktura danych:

```
---RentalItem.java---
package ubConferenceCenter;

public record RentalItem(String type,
                         int days,
                         int unitPrice,
                         int price,
                         int tax) {
}
```

Jak już wcześniej wspomniałem, ten moduł jest w pewien sposób problematyczny. Każda zmiana wprowadzona w tej strukturze danych wymusi ponowne kompilowanie i wdrożenie dwóch warstw. A zmiany tej struktury danych są prawdopodobne.

Są sposoby na rozwiązanie tej kwestii. Moglibyśmy na przykład zmienić `RentalItem` w mapę haszującą. Ale to prawdopodobnie mocno wykracza poza nasz obecny cel robienia miejsca dla nowych funkcjonalności w rozrastającym się kodzie. Zachowamy ten pomysł na później.

Kup ksi k   Pole ksi k

Teraz spójrzmy na niskopoziomową warstwę:

```
——SmallRoom.java——
package ubConferenceCenter.catalogItems;

import ubConferenceCenter.CatalogItem;

public class SmallRoom implements CatalogItem {
   public String getName() {
      return "SMALL_ROOM";
   }

   public int getDiscountedPrice(int days) {
      double discountRate = (days == 5) ? 0.9 : 1.0;
      return (int) Math.round(getUnitPrice() * days * discountRate);
   }

   public int getUnitPrice() {
      return 100;
   }

   public double getTaxRate() {
      return 0.05;
   }
}

——LargeRoom.java___
package ubConferenceCenter.catalogItems;

import ubConferenceCenter.CatalogItem;

public class LargeRoom implements CatalogItem {
   public int getDiscountedPrice(int days) {
      double discountRate = (days == 5) ? 0.9 : 1.0;
      return (int) Math.round(getUnitPrice() * days * discountRate);
   }

   public int getUnitPrice() {
      return 150;
   }

   public double getTaxRate() {
      return 0.05;
   }

   public String getName() {
      return "LARGE_ROOM";
   }
}

——Coffee.java——
package ubConferenceCenter.catalogItems;

import ubConferenceCenter.CatalogItem;

public class Coffee implements CatalogItem {
   public int getDiscountedPrice(int days) {
```


Kup ksi k   Pole ksi k

```
        return getUnitPrice() * days;
    }

    public int getUnitPrice() {
        return 10;
    }

    public double getTaxRate() {
        return 0;
    }

    public String getName() {
        return "COFFEE";
    }
}

---Cookies.java---
package ubConferenceCenter.catalogItems;

import ubConferenceCenter.CatalogItem;

public class Cookies implements CatalogItem {
    public int getDiscountedPrice(int days) {
        return getUnitPrice() * days;
    }

    public int getUnitPrice() {
        return 15;
    }

    public double getTaxRate() {
        return 0;
    }

    public String getName() {
        return "COOKIES";
    }
}

---CookieBonus.java---
package ubConferenceCenter.bonuses;

import ubConferenceCenter.Bonus;
import ubConferenceCenter.RentalItem;

import java.util.List;

public class CookieBonus implements Bonus {
    public void checkAndAdd(List<RentalItem> items,
                            List<RentalItem> bonusItems) {
        boolean largeRoomFiveDays = items.stream().anyMatch(
            item -> item.type().equals("LARGE_ROOM") && item.days() == 5);
        boolean coffeeFiveDays = items.stream().anyMatch(
            item -> item.type().equals("COFFEE") && item.days() == 5);
        if (largeRoomFiveDays && coffeeFiveDays)
            bonusItems.add(new RentalItem("COOKIES", 5, 0, 0, 0));
    }
}
```


Kup ksi k Pole ksi k

Jak widzisz, wszystkie reguły biznesowe zostały ukryte w odpowiednich klasach pochodnych. Jeśli musimy wprowadzić nowy upust na kawę, możemy umieścić go w klasie `Coffee`. Jeśli musimy dodać nowy bonus na małe salki konferencyjne, możemy dla niego utworzyć pochodną klasy `Bonus`. Jeśli chcemy dodać notatniki (`NotePads`) lub projektory (`Projectors`), możemy to zrobić poprzez dodanie tych klas do warstwy niskopoziomowej. Żadna z tych zmian nie będzie wymagała ponownych kompilacji i wdrożenia warstwy wysokopoziomowej. Udało nam się to osiągnąć dzięki stosowaniu się do zasady odwrócenia zależności.

Jednak może Cię martwić jedna rzecz. Możliwe, że zastanawiasz się, gdzie są tworzone te wszystkie pochodne. Nie pokazałem Ci moich testów, prawda? To tam się wszystko zadziało. Oto moje testy:

```
——RentalOrderTest.java——
package ubConferenceCenter;

import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.Test;
import ubConferenceCenter.RentalOrder.Totals;
import ubConferenceCenter.bonuses.CookieBonus;
import ubConferenceCenter.catalogItems.Coffee;
import ubConferenceCenter.catalogItems.Cookies;
import ubConferenceCenter.catalogItems.LargeRoom;
import ubConferenceCenter.catalogItems.SmallRoom;

import static org.junit.jupiter.api.Assertions.assertEquals;

public class RentalOrderTest {
   private final SmallRoom SMALL_ROOM = new SmallRoom();
   private final LargeRoom LARGE_ROOM = new LargeRoom();
   private final Coffee COFFEE = new Coffee();
   private final Cookies COOKIES = new Cookies();
   private RentalOrder order;

   @BeforeEach
   void setUp() {
      order = new RentalOrder("Customer Name");
      order.addBonus(new CookieBonus());
   }

   private void assertTotalAndTax(int subtotal, int tax) {
      Totals totals = order.getTotals();
      assertEquals(subtotal, totals.subtotal());
      assertEquals(tax, totals.tax());
   }

   private void assertTypeDaysUnitTotalTax(RentalItem item, String type,
                                           int days, int unitPrice,
                                           int price, int tax) {
      assertEquals(type, item.type());
      assertEquals(days, item.days());
      assertEquals(unitPrice, item.unitPrice());
      assertEquals(price, item.price());
      assertEquals(tax, item.tax());
   }
```

```
@Test
public void oneSmallRoomForOneDay() throws Exception {
   assertEquals("Customer Name", order.getCustomerName());
   order.rent(SMALL_ROOM, 1);
   RentalItem[] receipt = order.getReceipt();
   assertEquals(1, receipt.length);
   assertTypeDaysUnitTotalTax(receipt[0], "SMALL_ROOM", 1, 100, 100, 5);
   assertTotalAndTax(100, 5);
}

@Test
public void oneSmallRoomForTwoDays() throws Exception {
   order.rent(SMALL_ROOM, 2);
   RentalItem[] receipt = order.getReceipt();
   assertEquals(1, receipt.length);
   assertTypeDaysUnitTotalTax(receipt[0], "SMALL_ROOM", 2, 100, 200, 10);
   assertTotalAndTax(200, 10);
}

@Test
public void oneLargeRoomForThreeDays() throws Exception {
   order.rent(LARGE_ROOM, 3);
   RentalItem[] receipt = order.getReceipt();
   assertEquals(1, receipt.length);
   assertTypeDaysUnitTotalTax(receipt[0], "LARGE_ROOM", 3, 150, 450, 23);
   assertTotalAndTax(450, 23);
}

@Test
public void oneSmallRoomForOneWeek() throws Exception {
   order.rent(SMALL_ROOM, 5);
   RentalItem[] receipt = order.getReceipt();
   assertEquals(1, receipt.length);
   assertTypeDaysUnitTotalTax(receipt[0], "SMALL_ROOM", 5, 100, 450, 23);
   assertTotalAndTax(450, 23); /* 10% upustu */
}

@Test
public void twoSmallRoomsForOneDay() throws Exception {
   order.rent(SMALL_ROOM, 1);
   order.rent(SMALL_ROOM, 1);
   RentalItem[] receipt = order.getReceipt();
   assertEquals(2, receipt.length);
   assertTypeDaysUnitTotalTax(receipt[0], "SMALL_ROOM", 1, 100, 100, 5);
   assertTypeDaysUnitTotalTax(receipt[1], "SMALL_ROOM", 1, 100, 100, 5);
   assertTotalAndTax(200, 10);
}

@Test
public void oneSmallRoomAndCoffeeForOneDay() throws Exception {
   order.rent(SMALL_ROOM, 1);
   order.rent(COFFEE, 1);
   RentalItem[] receipt = order.getReceipt();
   assertEquals(2, receipt.length);
   assertTypeDaysUnitTotalTax(receipt[0], "SMALL_ROOM", 1, 100, 100, 5);
   assertTypeDaysUnitTotalTax(receipt[1], "COFFEE", 1, 10, 10, 0);
   assertTotalAndTax(110, 5); /* Brak podatku dla kawy */
}
```

```
    @Test
    public void oneSmallRoomAndCookiesForOneDay() throws Exception {
       order.rent(SMALL_ROOM, 1);
       order.rent(COOKIES, 1);
       RentalItem[] receipt = order.getReceipt();
       assertEquals(2, receipt.length);
       assertTypeDaysUnitTotalTax(receipt[0], "SMALL_ROOM", 1, 100, 100, 5);
       assertTypeDaysUnitTotalTax(receipt[1], "COOKIES", 1, 15, 15, 0);
       assertTotalAndTax(115, 5); /* Brak podatku dla ciastek */
    }

    @Test
    public void oneSmallRoomCookiesCoffeeForFiveDays() throws Exception {
       order.rent(SMALL_ROOM, 5);
       order.rent(COFFEE, 5);
       order.rent(COOKIES, 5);
       RentalItem[] receipt = order.getReceipt();
       assertEquals(3, receipt.length);
       assertTypeDaysUnitTotalTax(receipt[0], "SMALL_ROOM", 5, 100, 450, 23);
       assertTypeDaysUnitTotalTax(receipt[1], "COFFEE", 5, 10, 50, 0);
       assertTypeDaysUnitTotalTax(receipt[2], "COOKIES", 5, 15, 75, 0);
       /* 10% upustu na salkę, a nie na kawę */
       assertTotalAndTax(575, 23);
    }

    @Test
    public void oneLargeRoomAndCoffeeForWeekGetsCookies() throws Exception {
       order.rent(LARGE_ROOM, 5);
       order.rent(COFFEE, 5);
       RentalItem[] receipt = order.getReceipt();
       assertEquals(3, receipt.length);
       assertTypeDaysUnitTotalTax(receipt[0], "LARGE_ROOM", 5, 150, 675, 34);
       assertTypeDaysUnitTotalTax(receipt[1], "COFFEE", 5, 10, 50, 0);
       assertTypeDaysUnitTotalTax(receipt[2], "COOKIES", 5, 0, 0, 0);
    }
}
```

Napisałem te testy jeszcze przed refaktoryzacją, którą przeprowadzałem w tym rozdziale. Po każdym kroku refaktoryzacji sprawdzałem, czy kod te testy przechodzi. Niektóre zmiany wprowadzone w kodzie produkcyjnym pociągały za sobą zmiany w testach, ale byłem w stanie je zminimalizować poprzez odseparowanie kodu testującego i kodu produkcyjnego oraz odizolowanie szczegółów kodu produkcyjnego od szczegółów testów. Na przykład pola dla instancji `CatalogItem` zastąpiły starą enumerację bez większego zamieszania.

Możliwość niezależnego wdrażania

Nasz projekt ma teraz wyraźne granice architektoniczne, które dzielą kod na dwie części składowe. Te dwie części mogą być wdrażane niezależnie. Można je kompilować do dwóch osobnych plików JAR. Gdy jedna część się zmienia, tylko odpowiadający jej plik JAR się zmienia, a drugi nie musi być wdrażany ponownie.

Kup ksi k Pole ksi k

Wyobraź sobie, że ten kod został napisany w JavaScripcie, a nie w Javie. Wyobraź sobie, że będzie wysyłany do przeglądarki i tam uruchamiany. Podział na dwie części ma bardzo szczególną zaletę. Jeśli jedna z tych dwóch części ulegnie zmianie, a przeglądarka utrzymuje pamięć podręczną tych części, wtedy tylko zmieniona część musi zostać ponownie wczytana do przeglądarki. W przypadku wolnych łączy internetowych to będzie dużą zaletą.

Słowo końcowe

Porządki z tego rozdziału bardzo ułatwią dalszy rozwój projektu. Czas, który poświęciliśmy na czyszczenie kodu, zwróci się nam, ponieważ teraz jest oczywiste, gdzie dodawać te zmiany, które ponadto prawdopodobnie nie będą kolidować ze sobą lub ogólnym przebiegiem systemu.

Jednak ta nowa struktura niesie również koszty. Ci, którzy narzekają, że teraz kod czyta się gorzej z powodu wielu różnych modułów i niebezpośredniości, mają trochę racji. Tak, ustrukturyzowanie kodu skutkuje wzrostem złożoności, a ta złożoność nie jest darmowa. Musimy ją zrozumieć. Ale byliśmy świadomi, że obciążenie poznawcze również będzie rosnąć, a naszym celem było zminimalizowanie tego obciążenia poprzez elegancką organizację kodu.

A co z wydajnością? Czy nowa struktura jest wolniejsze od starej? Prawdopodobnie tak. Jest o kilka odwołań więcej, które trzeba śledzić, a kompilatory łatwiej sobie radzą z instrukcjami `swtich` w celu optymalizacji niż z dyspozycją polimorficzną. Jednak różnica w wydajności jest prawdopodobnie na tyle mała, że niełatwo byłoby ją zmierzyć. Dla większości aplikacji ten malutki spadek wydajności nie byłby w ogóle zauważalny.

BOB Z PRZYSZŁOŚCI: Model Grok3, choć jest dobry, nie był w stanie doprowadzić kodu do takiego stanu, jak zrobiliśmy to my. Nie odwrócił zależności ani nie oddzielił wysokopoziomowych zasad od niskopoziomowych szczegółów. Duże modele językowe/sztuczna inteligencja mają swoje zastosowania, ale nie są ludźmi, więc nie są biegłe w rozumieniu wyższych celów architektonicznych.

Ten rozdział i poprzedni były błyskawiczną podróżą po szybkich i łatwych usprawnieniach czyszczących kod. Ale to już koniec tej wycieczki i czas przejść do podstaw. I oczywiście zaczniemy od nazw.

Kup ksi k Pole ksi k

Skorowidz

A

AAA, Arrange/Act/Assert, 308
abstrakcja, 281, 332, 513
 bazowa, 336
 danych, 255
abstrakcje stabilne, 361
abstrakcyjne komponenty, 387
Active-X, 370
adaptery interfejsu, 479
ADC, analog-to-digital converter, 466
aktor, 344
algorytm współbieżny, 426
antyidiomy, 408
antywzorzec kopiuj-wklej, 274
API, 471, 474
architektura
 BCE, 476
 DCI, 476
 heksagonalna, 469, 476
 komponentów-wtyczek, 371
 MVC, 479
 oprogramowania, 450, 452, 455
  FitNesse, 459
  granice, 455, 458
  operacje systemu, 453
  programowanie systemu, 454
  przypadki użycia, 453
  wdrażanie systemu, 454
  wtyczki, 458
 portów i adapterów, 469, 476
argumenty funkcji, 174
 nazwane, 175
 wyjściowe, 177
 znacznikowe, 176
asercje złożone, composed assertions, 308
automatyczna instrumentacja kodu, 438
automatyczne dokańczanie kodu, 99

B

Baden-Powell Robert, 531
bałagan
 w kodzie, 300
 w testach, 303
baza danych, 351, 450, 456–459
BCE, Boundary–Control –Entity, 476
BDD, behavior-driven development, 315
Beck Kent, 35, 52, 252, 274, 295, 504
blok try/catch, 181
blokady, 433, 441
blokowanie optymistyczne, 524, 526
Bluetooth, 463
błędy współbieżności, 427
Booch Grady, 39

C

CCP, Common Closure Principle, 347, 371
cel asymptotyczny, 334
chirurgia strzelbowa, shotgun surgery, 274
ciągła
 integracja, 525, 526
 nauka, 556
 zmiana, 394
ciągłe
 budowanie, 317, 530
 projektowanie, continuous design, 394, 396
  jasność kodu, 397

Kup ksi k
Pole ksi k

ciągłe
projektowanie
z wyprzedzeniem, 422
przygotowywanie się, 422
rozpoczynanie pracy, 423
spójność modułów, 418
sprawdzalność kodu, 410
wykonywanie pracy, 423
zasady, 396
zwięzłość kodu, 402
wdrażanie, 529
Codd Edgar, 255
Copilot, 52
CRP, Common Reuse Principle, 371
Cunningham Ward, 40, 247
CVS, Concurrent Versions System, 524
cykl życia systemu, 454
czysta architektura, 476, 477
adaptery interfejsu, 479
encje, 478
frameworki i sterowniki, 479
przepływ sterowania, 480
przypadki użycia, 478
czyste
funkcje, 154
granice, 462, 475
cechy, 469
klasy, 271
testy, 304, 305, 313
zasady, 311
czysty kod, 33, 291, 296
definicje, 38
nazewnictwo, 85, 95
podstawowe zasady, 70
współbieżny, 425
czyszczenie kodu, 55, 61, 199, 504, 533
czytanie kodu, 47, 158
czytelność kodu, 46, 58, 98, 305

D

Dahl Ole-Johann, 254
DCI, Data, Context, Interaction, 476
debugowanie, 297, 300
degradacja kodu, 36, 301
dekompozycja funkcjonalna, functional decomposition, 158, 517
diagramy
napięć komponentów, 375
sekwencji, 465
zależności, 381
Dijkstra Edsger W., 194, 512
DIP, Dependency Inversion Principle, 78, 343, 359
dokumentacja, 298, 300
dowody poprawności oprogramowania, 513
DRY, Don't Repeat Yourself, 262, 428
duplikacja kodu, 338, 406
przypadkowa, accidental duplication, 339, 344, 346
duże modele językowe, LLM, 34, 290, 323, 328
dziedziczenie, 362

E

efekt uboczny, 190
Eisenhower Dwight D., 507
elastyczność
oprogramowania, 304, 313, 501
encje, 478
enumeracja, 513, 516

F

Feathers Michael, 40, 342
FIRST
Fast, szybkie, 311
Isolated, niezależne, 311
Repeatable, powtarzalne, 312
Self-Validating, samokontrolujące się, 312
Timely, o czasie, 312
FitNesse, 315
formatowanie
kodu, 137
komunikacja, 138
przeznaczenie, 138
zasady, 138
zasady zespołowe, 151
pionowe, 138
deklaracje zmiennych, 142
funkcje zależne, 144
gęstość, 141
koligacja koncepcyjna, 145
odległość, 141
odstępy, 140
rozmiar plików, 139
zmienne instancyjne, 143
poziome, 146
długość wierszy, 146
gęstość, 147
odstępy, 147
wyrównywanie, 148
szczegółów, 236
Fowler Matrin, 66, 229, 274
framework
Flask, 467
IoT, 463
React Native, 442
SOA, 450
wstrzykiwania zależności, 450
frameworki, 479
funkcje, 271
argumenty, 174
nazwane, 175
wyjściowe, 177
zmienna liczba, 175
znacznikowe, 176
bloki try/catch, 181
czystość, 169
czystość częściowa, 171

Kup ksi k   Pole ksi k

drzewo wywołań, 249
duże, 227
efekt uboczny, 190
jeden poziom abstrakcji, 156
jednorodność, 167
kody błędów, 178, 179, 182
kontekst, 161
łańcuch wywołań, 248
małe, 155, 226
nazewnictwo, 71, 162
odizolowane, 164
plątanina, 159
pokrywanie, 362
poziom wcięć, 156
prywatne, 248
przenoszenie, 400
publiczne, 248
rozdzielanie poleceń i zapytań, 178, 262
skakanie po kodzie, 226
stosowanie wyjątków, 179
w Clojure, 245
wcielanie, 246
wydajność, 226
wydzielanie, 242
wykonujące jedną rzecz, 222
wyodrębnianie, 400
zaplątanie logiki, 227

G

generowanie kodu przez AI, 322
gettery i settery, 242, 255, 262
Git, 525
gałęzie, 527
Goldstine Herman, 273
graf
komponentów
usuwanie cykli, 379
skierowany, 378
skierowany acykliczny, 378
graficzny interfejs użytkownika, 456
granice architektoniczne, 86, 93, 217, 241, 347, 455, 458
Grenning James, 359
Grok, 322, 327, 328
Grok3, 66, 69

H

HAL, hardware abstraction layer, 474
hermetyzacja, 164, 228, 233, 339
heurystyki, 274
hierarchia
ochrony komponentów, 352
przestrzeni nazw i klas, 225
typów, 239
Hopper Grace, 319
Hunt Andy, 262

I

IDE, 264
menu Refactor, 222
wydzielanie kodu, 227
indukcja, 513, 516
instrukcja
GOTO, 516
if-else, 409
return, 409
switch, 77, 159, 264
integralność danych, 439
interaktor, 351
interfejs użytkownika, UI, 467
interfejsy, 80, 332, 351, 456
abstrakcyjne, 362
polimorficzne, 359
internet rzeczy, IoT, 463
ISP, Interface Segregation Principle, 343, 356
izolacja
IoT, 469
UI, 469

J

jakość kodu, 504
jasność kodu, 397, 420
język
Algol, 225
C, 225
C++, 254, 271
Clojure, 245, 272
COBOL, 224, 325
Elixir, 271, 272
F#, 271
FORTRAN, 224, 319, 515
Go, 178, 229
Golang, 156
Haskell, 272
Java, 271
JavaScript, 271
Obejctive-C, 254
promptów, 324
Python, 254, 464
Ruby, 188, 254, 271
Simula, 254
Smalltalk, 254, 272
SNOBOL, 524
TypeScript, 271
UML, 275
języki
funkcyjne, 192
typowane dynamicznie, 358
typowane statycznie, 357
wszechobecne, 108
zorientowane obiektowo, 193
JIRA, 130

K

katastrofy oprogramowania, 42
Kay Alan, 254
klasy, 225, 272, 342, 456
duże, 278
interfejs, 277
mniejsze, 288
projektowanie, 273

Kup ksi k Pole ksi k

klasy
ukryte w funkcji, 229
usług, 287
wytyczne w kodzie, 277
zamknięte, 285
zasada jednej odpowiedzialności, 275, 285
zbyt duże, 275
zbyt otwarte, 281
kod
idiomatyczny, 408
otwartoźródłowy, 462
podobieństwa, 184
powtórzenia, 183
pętli, 187
przypadkowe, 190
zamierzone, 190
proceduralny, 257, 259
produkcyjny, 234
wielowątkowy, 425, 435, 436
kody błędów, 180–183
komentarze, 68, 279, 335
dobre, 117
dziennika, 125
HTML, 131
informacje nielokalne, 131
informacyjne, 117
jako graffiti w kodzie, 129
jako znaczniki pozycji, 129
Javadoc, 121, 128, 133
kłamliwe, 115
mylące, 123
nadmiar informacji, 132
nadmiarowe, 123, 126, 403
niezrozumiałe, 114, 116
ostrzegające, 119
połączenie z kodem, 132
powtarzające się, 122
prawne, 117
redukcja, 400, 404
tłumaczące znaczenie argumentów, 119
TODO, 128
ukryte, 114
wprowadzające szum informacyjny, 126, 128
wyjaśniające zamierzenia, 118
wymagane, 125
wzmacniające wagę operacji, 120
złe, 121, 130
komponenty, 247, 366
abstrakcyjne, 363, 387
diagram napięć, 375
konkretne, 364
łączenie, 376
spójność, 371
strefy wykluczenia, 389
zasady, 365–392
komunikacja asynchroniczna, 465
konsolidator, linker, 369
kontrola
nad zależnościami, 360
wersji, 520
ciągła integracja, 526
karty perforowane, 520
system CVS, 524
system Git, 525, 527
system RCS, 524
system SCCS, 524
system Subversion, 526
tablice ścienne, 522
kontroler, 351
kruchość
kodu, 266
systemu, 76
Kurzweil Ray, 290

L

lambda, 187, 401
Langr Jeff, 33
Laszczak Robert, 242
Liskov Barbara, 353
LLM, Large Language Model, 323, 328, 398
logika biznesowa, 87, 287
LSP, Liskov Substitution Principle, 343, 353
lukier składniowy, 408
luźne powiązanie, 164

Ł

łańcuch wywołań, 260
łatka, patch, 500

M

magiczne liczby, 98
mapa haszująca, 88
mapowanie obiektowo--relacyjne, ORM, 263
Marick Brian, 502
Martin Bob, 274
maszyna stanowa, 165
metafora gazety, 249
metody, 259, 271
metodyka
małe pakiety, 296
TCR, 295
TDD, 294
testów akceptacyjnych, 315, 316
metodyki zwinne, 293
Meyer Bertrand, 183, 348
miara
abstrakcji, 388
stabilności komponentu, 383
mikrokontroler, 466
mikrousługi, 287, 425
modularyzacja kodu, 273
moduł, 247, 342, 361
MVC, Model–View––Controller, 479
MVP, Minimum Viable Product, 420

N

narzędzie
ASM, 438
CGLIB, 438
FitNesse, 315
GitHub Copilot, 52
nazewnictwo
części mowy, 106
dezinformacja, 98
dziedzina problemu, 108

Kup ksi k   Pole ksi k

dziedzina rozwiązania, 108
funkcji, 103, 132, 162, 225, 242
intencja kodu, 96
interfejsy i implementacje, 105
klas, 103, 106, 225
kontekst, 97
kontekst haseł, 109
metody, 106
przestrzeni nazw, 104, 225
spójność terminologii, 107
systemy nazw, 97
unikanie kodowania, 104
unikanie nadmiarowego kontekstu, 111
właściwości i atrybuty, 104
wymowność nazw, 101
wyraźne różnice, 99
wyszukiwalność haseł, 101
zmienne, 102
niestabilność, instability, 384
niezawodność oprogramowania, 299
niskopoziomowe szczegóły, 78, 84, 86, 337
notacja węgierska, 104
Nygaard Kristen, 254

O

obiekt, 254
delegujący, 223
metody, 290
obiektowo-relacyjna niezgodność impedancji, 263
obiekty, 253, 263
POJO, 435
transferu danych, DTO, 262
wiązanie, 469
obsługa błędów, 181, 406
ochrona
kontrolera przed zmianami, 351
prezentera przed zmianami, 351
OCP, Open-Closed Principle, 77, 342, 348
odśmiecanie pamięci, garbage collection, 191
odwrócenie zależności, 78
operator trójargumentowy, 408, 409
oprogramowanie
dowody poprawności, 513
działanie, 505
elastyczność, 501
polityka, 450
sposób działania, 449
struktura, 449, 505
szczegóły architektoniczne, 450
testowanie, 502
warstwy, 477
orkiestracja logiki, 287
ORM, Object-Relational Mapping, 263
Ousterhout John, 57, 227, 248, 275, 296

P

paradoks Zenona z Elei, 45
perspektywy projektowe, 274
pętla, 187
plik źródłowy, 249, 272
pliki
.dll, 366, 371
.gem, 366
.jar, 268, 366, 370
.war, 366
podprocedura, 273
podprogramy, 154, 338
pokrycie kodu testami, 334, 531
polimorfizm, 238, 258, 288, 332
polityka, 450
potoki funkcyjne, 401
poziom abstrakcji, 248–252, 256
praca zdalna, 543
prawo
Conwaya, 342, 454
Demeter, 259, 396
LeBlanca, 36
Moore'a, 370
Murphy'ego, 370
predykaty, 279
produktywność, 300, 535
lepkość, 536
budowanie, 536
debugowanie, 537
testowanie, 536
wdrażanie, 538
zarządzanie
czasem, 540
rozproszeniem uwagi, 538
zespołu, 44
program, 367
ładujący, 369
ładujący konsolidujący, linking loader, 369
współbieżny, 427
programista jako interesariusz, 508
programowanie
asystent AI, 66
ekstremalne, 293, 294
funkcyjne, 58, 245
modularne, 225
obiektowe, 58, 354
podszyte strachem, 410
sterowane budowaniem, 325
sterowane testami, TDD, 155, 201–214, 294, 502, 508, 518
sterowane zachowaniem, BDD, 315
strukturalne, 158, 194, 515
pętla, 195
sekwencja, 195
wybór, 195
w parach, 543
współbieżne, 425, 427
za pomocą promptów, 320
zespołowe, 542
zorientowane obiektowo, 253, 267
zwinne, 294, 376

Kup ksi k
Pole ksi k

projekt, 394
typu top-down, 381
projektowanie, 296, 299
jako proces ciągły, 394
strategiczne, 296
testów, 312
w stylu classitis, 217
z wyprzedzeniem, 422
prompty, 320
scenariusze testowe, 327
proste projektowanie, 332
ograniczenie duplikacji, 338
pokrycie kodu testami, 334
wymóg YAGNI, 333
zmniejszanie rozmiaru, 339
zwiększenie wyrazistości, 335
przekazywanie wiadomości, 254
przełączanie zadań, 436
przepływ sterowania, 359, 480
przestrzeń nazw, 208, 225
przetwarzanie równoległe, 426
przetwornik analogowo-cyfrowy, ADC, 466
przypadki użycia, 478
przysięga programisty, 495, 504, 512, 520, 531, 535, 542, 545, 555, 556

Q

quasi-hermetyzacja obiektów, 262

R

RCS, Revision Control System, 524
React Native, 443
refaktoryzacja, 38, 66, 159, 200, 229, 282, 533
wyodrębnienie metod, 75, 222
refleksja, 285
reguła
80/20, 290
zależności, 217
reguły
biznesowe, 91, 240, 241, 450, 456, 457
formatowania, 241
wysokiego poziomu, 387
relacje między komponentami, 351
relacyjna baza danych, 255, 440
REP, Reuse/Release Equivalence Principle, 371
REST, 450
rozmiar kodu, 339
rozwiązanie zorientowane obiektowo, 268
RPC, Remote Procedure Call, 464

S

SAP, Stable Abstractions Principle, 387
scenariusz systemu webowego, 481
SDP, Stable Dependencies Principle, 382
Seeman Mark, 41
semafory, 119
separacja odpowiedzialności, 477
serwer aplikacji, 450
słowo kluczowe synchronized, 428, 433
SOLID, 274, 282, 340, 341, 469
DIP, 343
ISP, 343
LSP, 343
OCP, 342
SRP, 342
spójność kodu, 238, 418, 420
sprawdzalność kodu, 410, 420
sprzężenie
kodu, 238, 300, 312, 334
komponentów, 379
SRP, Single Responsibility Principle, 76, 342, 343
stabilność
komponentu, 382
semantyczna, 533
sterowniki, 479
strach przed zmianą kodu, 300, 304
strefy wykluczenia komponentów, 389
Stroustrup Bjarne, 38
struktura
danych, 253, 259, 263, 351, 480
hybrydowa, 260
systemu, 505
cechy, 506
macierz decyzyjna, 507
system rozproszony, peer-to-peer, 525
szacowanie terminów, 545
kłamstwa, 546
konkretność i precyzja, 550
nauka na błędach, 547
technika PERT, 552
uczciwość, 546, 552
szkodliwy kod, 495
przypadek Healthcare.gov, 496
przypadek Knight Capital, 497, 499
przypadek Toyoty, 498
przypadek Volkswagena, 495
sztuczna inteligencja, 34, 68, 289, 302, 320
sztywność kodu, 265
szum informacyjny, 126

Ś

środowisko wielowątkowe, 245

T

TCR, Test && commit || revert, 295
TDD, Test Driven Development, 294
prawa, 294

Kup ksi k
Pole ksi k

testowanie, 502
języki specyficzne
dla domeny, 308
kodu wielowątkowego, 434
kodu współbieżnego, 425
metodyki, 292–304
mutacyjne, 532
oprogramowania, 52, 69, 201–214, 290
podwójny standard, 310
projektowanie testów, 312
rozproszone, 289
scenariusze testowe
w promptach, 327
wzorzec AAA, 308, 311
zachowanie czystości testów, 303
zasada jednej asercji, 310
zasada jednej koncepcji, 311
zasady FIRST, 311
złożone wyniki, 308
testuj i zatwierdzaj albo cofaj, TCR, 295
testy, 286
akceptacyjne, 314, 317
metodyka, 315, 316
całościowe, 288
funkcjonalne, 410
graniczne, 473
integracyjne, 410
jako dokumentacja, 400
jako mechanizm projektowania, 423
jednostkowe, 289, 292, 404, 410
kontraktowe, 410
manualne, 299
obciążeniowe, 410
projektowanie, 232
uczące, 471, 473
wydajnościowe, 410
Thomas Dave, 262
typ wyliczeniowy, 80, 182, 265, 283
typy
prymitywne, 109
statyczne, 82

U

ukrywanie
danych, 254
implementacji, 256
struktur danych, 261, 263
UML, Unified Modeling Language, 275
upraszczanie kodu, 72
usługi, 425
typu restful, 355
zewnętrzne, 287
uszkodzenie kodu przez duplikację, 338
utrzymywanie systemu, 45

V

von Neumann John, 273

W

warstwa abstrakcji sprzętowej, HAL, 474
warunek wyścigu, 119
wątek
główny, 433
potomny, 433
wątki, 427
niezależne, 429
uwięzienie, 431
wzajemne wykluczanie, 431
zagłodzenie, 431
zakleszczenie, 431
wcięcia, 149, 156
łamanie, 151
znaczenie, 150
wdrażanie oprogramowania, 93, 299, 300
Wheelerm David, 273
wielowątkowość, 244
Wilkes Maurice, 273
Wirth Niklaus, 253, 516
wirtualne biuro, 543
właściciele produktu, product owners, 393
współbieżność, 425–445, 469
cel stosowania, 425
dołączanie do gry, 439
dostrajanie kodu, 436
eleganckie wyłączenie, 433
hermetyzacja danych, 429
instrumentacja automatyczna, 438
instrumentacja ręczna, 437
integralność danych, 439
kolekcje bezpieczne dla wątków, 430
korzystanie z kopii danych, 429
modele wykonania, 430
niezależne wątki, 429
oddzielanie kodu, 428
podwójne usuwanie zasobów, 442
problem
czytelnik – pisarz, 431
producent – konsument, 431
ucztujący filozofowie, 432
przypadkowe awarie, 435
testowanie, 434
testowanie na różnych platformach, 436
wczytywanie zasobów ikon, 443
włączanie kodu, 435
współdzielenie danych, 443
wymuszanie awarii, 436
zależności pomiędzy metodami, 432
zasady obrony współbieżności, 428
wstrzykiwanie zależności, 417, 450, 465
wtyczka IntelliJ's, 220
wtyczki, 85, 458
wydajność kodu, 269
wydzielanie metody, 222, 279
wyjątki, 179
wymagania użytkowników, 34
wyrazistość kodu, 335, 336
wyrażenia
lambda, 339, 401
regularne, 118

Kup książkę Poleć książkę

wywołanie polimorficzne, 238
wzorzec projektowy
Adapter, 474
Dekorator, 291
Fabryka, 291, 362
Fasada, 346
Kompozyt, 291
Łańcuch odpowiedzialności, 291
Metoda szablonowa, 187, 339
Most, 291
Odwiedzający, 108
Polecenie, 187, 291
Potok funkcyjny, 291
Singleton, 117, 285
Skromny Obiekt, 302
Stan, 288
Strategia, 187, 282, 291, 339
Visitor, 258

Y

YAGNI, You aren't gonna need it, 84, 333

Z

zaciemnianie kodu, 402
zakres klasy, 228
zależności
cykliczne, 377
komponentów, 379
od abstrakcji, 360
pomiędzy synchronizowanymi metodami, 432
przejściowe, 352
w kodzie źródłowym, 361
wchodzące, fan-in, 384
wychodzące, fan-out, 384
zapach kodu, 274
zarządzanie
czasem, 540
projektem, 303
rozproszeniem uwagi, 538
zależnościami, 506
zasada
acyklicznych zależności, ADP, 376
DRY, 262, 428
istotności numeru wydania, REP, 371
jednej asercji, 311
jednej koncepcji, 311
jednej odpowiedzialności, SRP, 76, 160, 250, 342, 454
jednej rzeczy, 39
odwrócenia zależności, DIP, 78, 266, 332, 343, 363, 386, 461
otwarte – zamknięte, OCP, 77, 160, 242, 265, 287, 342, 348
podstawień Liskov, LSP, 343, 353
rozdzielania interfejsów, ISP, 343, 356, 374
skautów, 48, 300, 301
stabilnych abstrakcji, SAP, 387
stabilnych zależności, SDP, 382
wspólnego domknięcia, CCP, 347, 372
wspólnego użycia, CRP, 373
YAGNI, 84
zależności, 364, 477
zależności architektonicznych, 461
zstępująca, 157, 167, 249, 251
zasady
ciągłego projektowania, 396
komponentów, 365–392
obrony współbieżności, 428
SOLID, 282, 341, 469
wysokopoziomowe, 84, 86, 337
zatrucie architektury systemu, 356
zazdrość o funkcje, feature envy, 280
zdalne wywołanie procedury, RPC, 464
zestaw testów, 301
zintegrowane środowisko programistyczne, IDE, 84
zły kod
konsekwencje biznesowe, 36
zmienne
instancyjne, 228
lokalne, 227
prywatne, 255
statyczne, 244
znaczniki pozycji, 129
zwięzłość kodu, 402, 420

Kup ksi k Pole ksi k

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

Kod, który przetrwa próbę czasu — oto profesjonalizm w erze AI

W erze sztucznej inteligencji, mikrousług i ciągłego wdrażania umiejętność pisania czystego, zrozumiałego kodu stała się istotą sukcesu każdego projektu IT. **Robert C. Martin, znany jako „Wujek Bob"**, powraca z całkowicie przepracowanym i rozszerzonym wydaniem swojego kultowego bestsellera. Książka *Czysty kod* od lat kształtuje sposób myślenia programistów na całym świecie — pierwsze wydanie zmieniło podejście tysięcy deweloperów do jakości kodu. Drugie wydanie odpowiada na wyzwania współczesnego programowania, zachowując ponadczasowe zasady rzemiosła programistycznego. To obowiązkowa lektura dla każdego, kto traktuje programowanie jako profesję, a nie tylko zajęcie.

Książka została podzielona na cztery kluczowe części: **Kod, Projektowanie, Architektura i Rzemiosło**. Autor prowadzi czytelnika przez praktyczne przykłady w wielu językach (**Java, Python, Go, JavaScript, Clojure, C**), pokazując, jak przekształcać nieczytelny kod w eleganckie rozwiązania. Szczegółowo omawia zasady SOLID, wzorce projektowe, testowanie (TDD, TCR), refaktoryzację i architekturę heksagonalną. Nowością są rozdziały poświęcone etyce programowania, współpracy z AI i odpowiedzialności zawodowej. We wszystkich rozdziałach znajdują się konkretne przykłady transformacji kodu — od wersji „brudnej" do „czystej" — z dokładnym wyjaśnieniem każdego kroku refaktoryzacji.

Najważniejsze zagadnienia:

- **Zasada „najpierw spraw, by działało, potem spraw, by było dobrze"**
- **Zasady SOLID i wzorce projektowe w praktyce**
- **Programowanie sterowane testami (TDD)**
- **Czysta architektura**
- **Współpraca z AI i dużymi modelami językowymi**
- **Etyka zawodowa programisty**
- **Praktyczne studia przypadku**

Robert C. Martin (Wujek Bob) programuje od 1970 roku. Założyciel Uncle Bob Consulting i współzałożyciel Clean Coders. Autor bestsellerowych książek: *Zwinne wytwarzanie oprogramowania*, *Mistrz czystego kodu*, *Czysta architektura*, *Rzemiosło w czystej formie*. Pierwszy przewodniczący Agile Alliance, redaktor naczelny „The C++ Report". Regularnie występuje na międzynarodowych konferencjach, prowadzi szkolenia i publikuje na swoim blogu. Uznawany za jednego z najważniejszych myślicieli w dziedzinie inżynierii oprogramowania.

Helion

helion.pl

HELION S.A.
ul. Kościuszki 1c
44-100 Gliwice
tel.: 32 230 98 63
helion@helion.pl

KOD KORZYŚCI
Sięgnij po więcej!

ISBN 978-83-289-3791-8

Cena: 129,00 zł

Pearson