
Helion
Rzemiosło w czystej formie
Standardy i etyka rzetelnych programistów
Przedmowa
Stacia Heimgartner Viscardi, CST & Agile Mentor
Robert C. Martin

Tytuł oryginału: Clean Craftsmanship: Disciplines, Standards, and Ethics (Robert C. Martin Series)

Tłumaczenie: Krzysztof Bąbol

ISBN: 978-83-283-9056-0

Page xxix: Author photo courtesy of Robert C. Martin
Page 6: ". . . we shall need . . . what we are doing." A.M. Turing's ACE Report of 1946 and Other Papers – Vol. 10, "In the Charles Babbage Institute Reprint Series for the History of Computing", (B.E. Carpenter, B.W. Doran, eds.). The MIT Press, 1986.
Page 62: Illustration by Angela Brooks
Page 227: "You aren't gonna need it." Jeffries, Ron, Ann Anderson, and Chet Hendrickson. Extreme Programming Installed. Addison-Wesley, 2001.
Page 280: "We shall need . . . work of this kind to be done." A.M. Turing's ACE Report of 1946 and Other Papers – Vol. 10, "In the Charles Babbage Institute Reprint Series for the History of Computing", (B.E. Carpenter, B.W. Doran, eds.). The MIT Press, 1986.
Page 281: "One of our difficulties . . . what we are doing." A.M. Turing's ACE Report of 1946 and Other Papers – Vol. 10, "In the Charles Babbage Institute Reprint Series for the History of Computing", (B.E. Carpenter, B.W. Doran, eds.). The MIT Press, 1986.
Page 289: "This was a couple . . . for whatever reasons." Volkswagen North America CEO Michael Horn prior to testifying before the House Energy and Commerce Committee in Washington, October 8, 2015.
Page 310: "I have two . . . are never urgent." In a 1954 speech to the Second Assembly of the World Council of Churches, former U.S. President Dwight D. Eisenhower, who was quoting Dr J. Roscoe Miller, president of Northwestern University.
Page 310: Dwight D. Eisenhower, President of the United States, photo, February 1959. Niday Picture Library/Alamy Stock Photo.
Page 319: "Of course I . . . any size at all." Edsger W. Dijkstra: Notes on Structured Programming in Pursuit of Simplicity; the manuscripts of Edsger W. Dijkstra, Ed. Texas; 1969-1970; *http://www.cs.utexas.edu/users/EWD/*ewd02xx/EWD249.PDF
Page 328: Photo courtesy of Robert C. Martin
Page 331: Photo courtesy of Robert C. Martin

Authorized translation from the English language edition, entitled Clean Craftsmanship: Disciplines, Standards and Ethics, 1st Edition by Robert C. Martin, published by Pearson Education, Inc, publishing as Addison Wesley Professional, Copyright © 2022 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 © 2022.

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
https://helion.pl/user/opinie/rzemcz
Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

Helion S.A.
ul. Kościuszki 1c, 44-100 Gliwice
tel. 32 231 22 19, 32 230 98 63
e-mail: *helion@helion.pl*
WWW: *https://helion.pl* (księgarnia internetowa, katalog książek)

Printed in Poland.

- Kup książkę
- Poleć książkę
- Oceń książkę

- Księgarnia internetowa
- Lubię to! » Nasza społeczność

Spis treści

Kup książkę Poleć książkę

Kup książkę Poleć książkę

Kup książkę Poleć książkę

Kup książkę Poleć książkę

Kup książkę Poleć książkę

Kup książkę Poleć książkę

Kup książkę Poleć książkę

Kup książkę Poleć książkę

5 Refaktoryzacja

Kup książkę Poleć książkę

W 1999 r. przeczytałem *Refaktoryzację*[1] Martina Fowlera. Ta książka to klasyka gatunku i zachęcam do sięgnięcia po nią i lektury. Niedawno ukazało się jej drugie wydanie[2], w znacznym stopniu przeredagowane i unowocześnione. W pierwszym wydaniu przykłady zostały przedstawione w języku Java, w drugim — w języku JavaScript.

W czasach, gdy czytałem pierwsze wydanie, mój dwunastoletni syn, Justin, należał do zespołu hokejowego. Tym z Was, którzy nie mają pociech grających w hokeja, winien jestem informację, że dziecko przebywa pięć minut na lodzie, a potem schodzi z niego na dziesięć – piętnaście minut po to, by ochłonąć.

Gdy mój syn był poza lodowiskiem, zatapiałem się w lekturze wspaniałej książki Martina. To była pierwsza przeczytana przeze mnie pozycja, w której kod został przedstawiony jako coś *plastycznego*. Większość innych wydawanych wtedy i wcześniej książek prezentowała kod w ostatecznej formie. *Ta* jednak pokazywała, że można wziąć zły kod i go oczyścić.

Gdy ją czytałem, słyszałem, jak tłum dopingował dzieci na lodowisku, a ja wiwatowałem razem z nimi — ale nie kibicowałem grze. Kibicowałem temu, co czytałem w tej książce. Pod wieloma względami to właśnie ona naprowadziła mnie na drogę do napisania *Czystego kodu*[3].

Nikt nie ujął tego lepiej niż Martin:

> *Nie jest sztuką napisanie kodu zrozumiałego dla komputera.*
> *Dobrzy programiści tworzą kod*
> *zrozumiały dla ludzi4.*

W tym rozdziale przedstawiam sztukę refaktoryzacji z mojego osobistego punktu widzenia. Nie jest moją intencją konkurowanie z książką Martina.

---

[1] Martin Fowler, *Refaktoryzacja. Ulepszanie struktury istniejącego kodu*, Helion, Gliwice 2011.

[2] Martin Fowler, *Refaktoryzacja. Ulepszanie struktury istniejącego kodu*, wydanie II, Helion, Gliwice 2019.

[3] Robert C. Martin, *Czysty kod. Podręcznik dobrego programisty*, Helion, Gliwice 2010.

[4] Martin Fowler, *Refaktoryzacja. Ulepszanie struktury istniejącego kodu*, Helion, Gliwice 2011, s. 26.

Kup książkę Poleć książkę

CZYM JEST REFAKTORYZACJA?

Tym razem sparafrazuję Martina autocytatem:

> *Refaktoryzacja to sekwencja niewielkich zmian polepszających strukturę oprogramowania bez zmiany jego działania — o czym świadczy pomyślne wykonywanie się kompleksowego zestawu testów po każdej zmianie dokonanej w tej sekwencji.*

Definicja ta obejmuje dwie istotne kwestie.

Po pierwsze, przy refaktoryzacji *zachowywane* jest działanie oprogramowania. Zarówno po jednej refaktoryzacji, jak i po całej ich serii jego funkcjonowanie pozostaje niezmienione. Jedynym znanym przeze mnie sposobem, by tego dowieść, jest regularne pomyślne przechodzenie zestawu *kompleksowych* testów.

Po drugie, każda refaktoryzacja jest *niewielka*. Co to znaczy? Oto instrukcja: ma być *na tyle drobna, by nie trzeba było jej debugować*.

Istnieje wiele specyficznych technik refaktoryzacji, a na kolejnych stronach opiszę niektóre z nich. W kodzie można dokonać także wielu innych zmian nie zawartych w kanonie refaktoryzacji, które jednak przy modyfikacji struktury zachowują jej działanie. Niektóre refaktoryzacje są tak schematyczne, że może je przeprowadzać środowisko IDE. Niektóre są na tyle proste, że można ich bez obaw dokonywać ręcznie. Niektóre wreszcie są bardziej skomplikowane i wymagają znacznej dbałości. W tym ostatnim przypadku postępuję według wcześniej wspomnianej instrukcji. Jeśli obawiam się, że w końcu będę musiał użyć debugera, dzielę zmianę na mniejsze, bezpieczniejsze części. Jeśli mimo wszystko trafiam do debugera, koryguję swój próg obaw i staram się zachować większą ostrożność.

> *Zasada nr 15. Unikaj debugera.*

Kup książkę Poleć książkę

Celem refaktoryzacji jest oczyszczenie kodu. Odbywa się to w procesie czerwone → zielone → refaktoryzacja. Jest to działanie ciągłe, a nie figurujące w rozkładzie czy zaplanowane. Czystość kodu jest utrzymywana w ciągłej pętli czerwone → zielone → refaktoryzacja.

Niekiedy zachodzi potrzeba dokonania refaktoryzacji w szerszym zakresie. W sposób nieunikniony struktura systemu wymaga z czasem aktualizacji i tę zmianę projektową trzeba wprowadzić w treści kodu. Nie umieszczaj tego w harmonogramie. Nie przestawaj dodawać funkcji i poprawiać błędów. Włóż po prostu w cykl czerwone → zielone → refaktoryzacja nieco dodatkowych wysiłków i stopniowo wprowadzaj wymagane zmiany, nieprzerwanie dostarczając wartość biznesową.

PODSTAWOWY ZESTAW NARZĘDZIOWY

Jest kilka refaktoryzacji, które stosuję o wiele częściej niż inne. Są one zautomatyzowane w używanym przeze mnie środowisku IDE. Zachęcam, by nauczyć się ich na pamięć i poznać tajniki ich automatyzacji we własnym środowisku programistycznym.

ZMIANA NAZWY

Właściwemu nazewnictwu poświęciłem jeden z rozdziałów książki *Czysty kod*. Dostępnych jest też wiele innych materiałów[5] do nauki dobrego dobierania nazw. Ważne, by… nazywać rzeczy po imieniu.

Nazewnictwo jest trudne. Znajdowanie właściwej nazwy odbywa się często w procesie kolejnych iteracyjnych ulepszeń. Nie obawiaj się dochodzić do odpowiedniej nazwy. Ulepszaj nazwy, kiedy tylko możesz, póki projekt jest świeży.

5 Innym dobrym punktem odniesienia jest książka Erica Evansa *Domain-Driven Design: Zapanuj nad złożonym systemem informatycznym*, Helion, Gliwice 2015.

Kup książkę Poleć książkę

W miarę osiągania przez projekt dojrzałości zmiana nazw staje się coraz trudniejsza. Coraz większej liczbie programistów nazwy utrwalają się w pamięci i zmienianie ich bez ostrzeżenia nie spotka się z przychylną reakcją. Z upływem czasu zmiana ważnych klas i funkcji będzie wymagać spotkań i uzgodnień.

Jeśli więc piszesz nowy kod, eksperymentuj z nazwami, póki nie jest on zbyt znany. Często zmieniaj nazwy klas i metod. Gdy będziesz to robić, odkryjesz, że warto je inaczej pogrupować. Będziesz przenosić metody z jednej klasy do drugiej, by zachować spójność z nowymi nazwami. Będziesz zmieniać podział funkcji i klas tak, by odpowiadał nowemu schematowi nazewnictwa.

Mówiąc krótko, praktyka wyszukiwania najlepszych nazw najprawdopodobniej wpłynie w sposób wysoce pozytywny na podział kodu na klasy i moduły.

Staraj się więc często i poprawnie stosować refaktoryzację **zmiana nazwy** (ang. *Rename*).

WYODRĘBNIANIE METODY

Wyodrębnianie metody (ang. *Extract Method*) to być może najważniejsza ze wszystkich refaktoryzacji. Możliwe, że jest tak naprawdę najistotniejszym mechanizmem utrzymywania w kodzie ładu i dobrej organizacji.

Proponuję narzucić sobie wymóg *wyodrębniania do upadłego*.

Takie postępowanie zmierza do osiągnięcia dwóch celów. Po pierwsze, każda funkcja powinna wykonywać *jedną operację*[6]. Po drugie, kod powinien dać się czytać jak *dobrze napisana proza*[7].

Funkcja wykonuje *jedną operację* wtedy, gdy nie można wyodrębnić z niej żadnej innej. Aby więc wszystkie Twoje funkcje wykonywały *jedną operację*, wyodrębniaj, wyodrębniaj i jeszcze raz wyodrębniaj z nich kod, póki jest to możliwe.

[6] Martin, *Czysty kod,* s. 57.

[7] *Ibid.*, s. 30.

Kup książkę Poleć książkę

To oczywiście doprowadzi do powstania mnóstwa drobnych funkcji. A to może przeszkadzać. Być może czujesz, że taka ilość małych funkcji zaciemni przeznaczenie kodu. Być może martwisz się, że w takim gąszczu funkcji łatwo się pogubisz.

Stanie się jednak coś przeciwnego. Przeznaczenie kodu będzie dużo bardziej oczywiste. Poziomy abstrakcji staną się wyraźne, a granice pomiędzy nimi — jasne.

Pamiętaj, że w dzisiejszych językach programowania dozwolone jest istnienie wielu modułów, klas i przestrzeni nazw. Pozwala to zbudować hierarchię nazw do umieszczania w nich funkcji. Przestrzenie nazw zawierają klasy, te z kolei — funkcje. W funkcjach publicznych można się odnosić do prywatnych. Klasy mogą zawierać klasy wewnętrzne i zagnieżdżone.

I tak dalej. Wykorzystaj te narzędzia, by utworzyć strukturę ułatwiającą innym programistom znalezienie napisanych przez Ciebie funkcji.

A potem wybierz dobre nazwy. Pamiętaj, że długość nazwy funkcji powinna być *odwrotnie* proporcjonalna do głębokości zakresu, w którym się znajduje. Nazwy funkcji publicznych powinny być relatywnie krótkie, a prywatnych — dłuższe.

W miarę wyodrębniania nazwy funkcji będą się stawać coraz dłuższe, bo ich przeznaczenie będzie coraz mniej ogólne. Większość z wyodrębnionych funkcji będzie wywoływana tylko z jednego miejsca, więc będą się cechowały mocną specjalizacją i sprecyzowanym przeznaczeniem. Nazwy takich funkcji muszą być długie. Prawdopodobnie będą to całe wyrażenia, a nawet zdania.

Funkcje te będą wywoływane w nawiasach pętli `while` i instrukcji `if`. Będą też wywoływane z treści tych instrukcji, co sprawi, że kod zacznie wyglądać następująco:

```
if (employeeShouldHaveFullBenefits())
  AddFullBenefitsToEmployee();
```

Dzięki temu kod stanie się tak czytelny jak *dobrze napisana proza*.

Kup książkę Poleć książkę

Wyodrębnianie metody pozwala też doprowadzić do tego, by funkcje stały się zgodne z *zasadą zstępującą* (ang. *the stepdown rule*)[8]. Chcemy, by każdy wiersz funkcji znajdował się na tym samym poziomie abstrakcji, czyli o jeden niżej niż *nazwa* funkcji. W tym celu wyodrębniamy z funkcji wszystkie fragmenty kodu znajdujące się poniżej pożądanego poziomu.

WYODRĘBNIANIE ZMIENNEJ

Jeśli wyodrębnianie metody jest najważniejszą z refaktoryzacji, to stałym dla niej wsparciem jest **wyodrębnianie zmiennej** (ang. *Extract Variable*). Okazuje się, że aby wydzielić metodę, często trzeba najpierw wydzielić z niej zmienne.

Rozważmy na przykład refaktoryzację gry w kręgle opisaną w rozdziale 2., „Programowanie sterowane testami". Rozpoczęliśmy od tego:

```
@Test
public void allOnes() throws Exception {
  for (int i=0; i<20; i++)
    g.roll(1);
  assertEquals(20, g.score());
}
```

A skończyliśmy na tym:

```
  private void rollMany(int n, int pins) {
    for (int i = 0; i < n; i++) {
      g.roll(pins);
    }
  }

  @Test
  public void allOnes() throws Exception {
    rollMany(20, 1);
    assertEquals(20, g.score());
  }
```

Sekwencja refaktoryzacji była następująca:

[8] *Ibid.*, s. 58.

Kup książkę Poleć książkę

1. *Wyodrębnianie zmiennej*: liczba `1` w wywołaniu `g.roll(1)` została wyodrębniona do zmiennej o nazwie `pins` (kręgle).
2. *Wyodrębnianie zmiennej*: liczba `20` w wywołaniu `assertEquals(20, g.score());` została wyodrębniona do zmiennej `n`.
3. Te dwie zmienne zostały przeniesione przed pętlę `for`.
4. *Wyodrębnianie metody*: pętla `for` została wyodrębniona do funkcji `rollMany`. Nazwy zmiennych stały się nazwami argumentów.
5. *Wchłonięcie* (ang. *inline*): te dwie zmienne zostały wchłonięte. Spełniły swoje zadanie i przestały być potrzebne.

Innym częstym zastosowaniem wyodrębniania zmiennej jest tworzenie *zmiennej objaśniającej* (ang. *explanatory variable*)[9]. Weźmy na przykład pod uwagę taką instrukcję `if`:

```
if (employee.age > 60 && employee.salary > 150000)
     ScheduleForEarlyRetirement(employee);
```

Może ona być bardziej czytelna po dodaniu zmiennej objaśniającej:

```
boolean isEligibleForEarlyRetirement = employee.age > 60 &&
                                       employee.salary > 150000
if (isEligibleForEarlyRetirement)
     ScheduleForEarlyRetirement(employee);
```

WYODRĘBNIANIE POLA

Ta refaktoryzacja przynosi czasem bardzo pozytywne efekty. Nie stosuję jej często, ale gdy to robię, prowadzi mnie na drogę do znacznego ulepszenia kodu.

Wszystko zaczyna się od nieudanego wyodrębniania metody. Przyjrzyjmy się poniższej klasie, która przekształca plik z danymi w formacie CSV na raport. Jest trochę bałaganiarska.

```
public class NewCasesReporter {
  public String makeReport(String countyCsv) {
```

[9] Kent Beck, *Smalltalk Best Practice Patterns*, Addison-Wesley, 1997, s. 108.

Kup książkę Poleć książkę

```
int totalCases = 0;
Map<String, Integer> stateCounts = new HashMap<>();
List<County> counties = new ArrayList<>();

String[] lines = countyCsv.split("\n");
for (String line : lines) {
  String[] tokens = line.split(",");
  County county = new County();
  county.county = tokens[0].trim();
  county.state = tokens[1].trim();
  // policz średnią kroczącą
  int lastDay = tokens.length - 1;
  int firstDay = lastDay - 7 + 1;
  if (firstDay < 2)
    firstDay = 2;
  double n = lastDay - firstDay + 1;
  int sum = 0;
  for (int day = firstDay; day <= lastDay; day++)
    sum += Integer.parseInt(tokens[day].trim());
  county.rollingAverage = (sum / n);

  // policz sumę przypadków zachorowań
  int cases = 0;
  for (int i = 2; i < tokens.length; i++)
    cases += (Integer.parseInt(tokens[i].trim()));
  totalCases += cases;
  int stateCount = stateCounts.getOrDefault(county.state, 0);
  stateCounts.put(county.state, stateCount + cases);
  counties.add(county);
}
StringBuilder report = new StringBuilder("" +
  "Hrabstwo     Stan     Średnia liczba nowych przypadków\n" +
  "========     ====     ================================\n");
for (County county : counties) {
  report.append(String.format("%-13s%-9s%.2f\n",
    county.county,
    county.state,
    county.rollingAverage));
}
report.append("\n");
TreeSet<String> states = new TreeSet<>(stateCounts.keySet());
for (String state : states)
  report.append(String.format("Zachorowania w stanie %s: %d\n",
```


Kup książkę Poleć książkę

```
            state, stateCounts.get(state)));
        report.append(String.format("Suma przypadków: %d\n", totalCa-
ses));
        return report.toString();
      }

      public static class County {
        public String county = null;
        public String state = null;
        public double rollingAverage = Double.NaN;
      }
    }
```

Na szczęście autor był łaskaw napisać kilka testów. Nie są doskonałe, ale wystarczą.

```
    public class NewCasesReporterTest {
      private final double DELTA = 0.0001;
      private NewCasesReporter reporter;

      @Before
      public void setUp() throws Exception {
        reporter = new NewCasesReporter();
      }

      @Test
      public void countyReport() throws Exception {
        String report = reporter.makeReport("" +
          "c1, s1, 1, 1, 1, 1, 1, 1, 1, 7\n" +
          "c2, s2, 2, 2, 2, 2, 2, 2, 2, 7");
        assertEquals("" +
            "Hrabstwo     Stan     Średnia liczba nowych przypadków\n" +
            "========     ====     ================================\n" +
            "c1           s1        1,86\n" +
            "c2           s2        2,71\n\n" +
            "Zachorowania w stanie s1: 14\n" +
            "Zachorowania w stanie s2: 21\n" +
            "Suma przypadków: 35\n",
          report);
      }

      @Test
      public void stateWithTwoCounties() throws Exception {
```


Kup książkę Poleć książkę

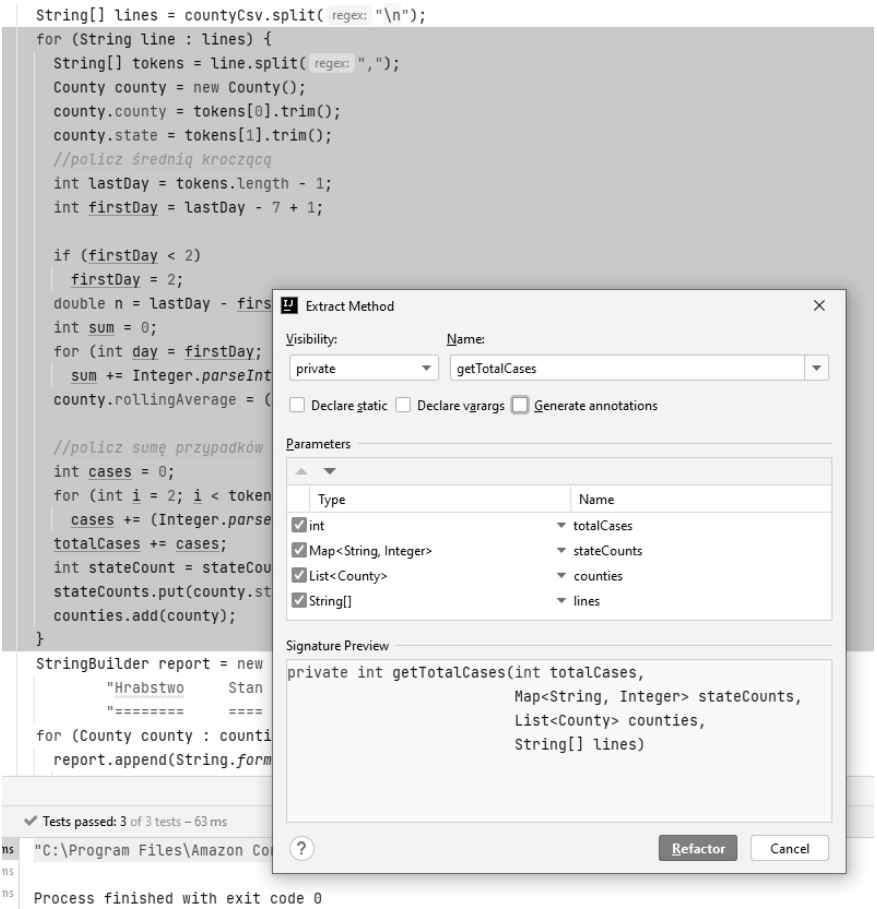
```
    String report = reporter.makeReport("" +
      "c1, s1, 1, 1, 1, 1, 1, 1, 1, 7\n" +
      "c2, s1, 2, 2, 2, 2, 2, 2, 2, 7");
    assertEquals("" +
        "Hrabstwo     Stan     Średnia liczba nowych przypadków\n" +
        "========     ====     ================================\n" +
        "c1            s1       1,86\n" +
        "c2            s1       2,71\n\n" +
        "Zachorowania w stanie s1: 35\n" +
        "Suma przypadków: 35\n",
      report);
  }

  @Test
  public void statesWithShortLines() throws Exception {
    String report = reporter.makeReport("" +
      "c1, s1, 1, 1, 1, 1, 7\n" +
      "c2, s2, 7\n");
    assertEquals("" +
        "Hrabstwo     Stan     Średnia liczba nowych przypadków\n" +
        "========     ====     ================================\n" +
        "c1            s1       2,20\n" +
        "c2            s2       7,00\n\n" +
        "Zachorowania w stanie s1: 11\n" +
        "Zachorowania w stanie s2: 7\n" +
        "Suma przypadków: 18\n",
      report);
  }
}
```

Testy te pozwalają się nam dobrze zorientować w tym, co robi program. Dane wejściowe to ciąg w formacie CSV. Każdy wiersz reprezentuje jedno hrabstwo i zawiera wykaz liczby notowanych codziennie nowych przypadków COVID. Dane wyjściowe to raport pokazujący siedmiodniową średnią kroczącą nowych przypadków w każdym hrabstwie oraz sumy dla poszczególnych stanów wraz z łączną sumą zachorowań.

Chcemy oczywiście zacząć wyodrębniać metody z tej przerażająco wielkiej funkcji. Rozpocznijmy od tej pętli na początku. Wykonywane są w niej wszystkie obliczenia dla poszczególnych hrabstw, powinniśmy chyba nazwać ją więc `calculateCounties`.

Kup książkę Poleć książkę

Jednak po zaznaczeniu tej pętli i podjęciu próby wyodrębnienia metody wyświetla się okno dialogowe pokazane na **rysunku 5.1**.



Rysunek 5.1. Okno dialogowe Extract Method

Środowisko IDE proponuje nadać tej funkcji nazwę `getTotalCases`. Twórcom tego środowiska trzeba przyznać, że sporo się napracowali nad podpowiedziami dla nazw. Wspomniana nazwa została wybrana dlatego, że w kodzie po pętli potrzebna jest liczba nowych przypadków, a nie można jej otrzymać, jeśli nie zwróci ich nowa funkcja.

Kup książkę Poleć książkę

Nie chcemy jednak nadawać tej funkcji nazwy `getTotalCases`. Nie mamy takiego zamiaru. Chcemy ją nazwać `calculateCounties`. Nie chcemy też przekazywać do niej tych czterech argumentów. Wystarczyłoby przekazać tablicę `lines`.

Kliknijmy więc przycisk *Cancel* (anuluj) i spójrzmy jeszcze raz.

Aby dokonać poprawnej refaktoryzacji, musimy wyodrębnić kilka lokalnych zmiennych zawartych w pętli do pól klasy, w której się ona znajduje. Zastosujemy do tego **wyodrębnianie pola** (ang. *Extract Field*):

```
public class NewCasesReporter {
  private int totalCases;
  private final Map<String, Integer> stateCounts = new HashMap<>();
  private final List<County> counties = new ArrayList<>();

  public String makeReport(String countyCsv) {
    totalCases = 0;
    stateCounts.clear();
    counties.clear();

    String[] lines = countyCsv.split("\n");
    for (String line : lines) {
      String[] tokens = line.split(",");
      County county = new County();
```

Zwróć uwagę że na początku funkcji `makeReport` nadaliśmy wartości zmiennym. W ten sposób zachowane zostało jej pierwotne działanie.

Możemy już wyodrębnić pętlę bez konieczności przekazywania większej liczby zmiennych, niż byśmy chcieli, i bez zwracania zmiennej `totalCases`:

```
public class NewCasesReporter {
  private int totalCases;
  private final Map<String, Integer> stateCounts = new HashMap<>();
  private final List<County> counties = new ArrayList<>();

  public String makeReport(String countyCsv) {
    String[] countyLines = countyCsv.split("\n");
    calculateCounties(countyLines);
```


Kup książkę Poleć książkę

```
    StringBuilder report = new StringBuilder("" +
      "Hrabstwo     Stan     Średnia liczba nowych przypadków\n" +
      "========     ====     ================================\n");
    for (County county : counties) {
      report.append(String.format("%-13s%-9s%.2f\n",
        county.county,
        county.state,
        county.rollingAverage));
    }
    report.append("\n");
    TreeSet<String> states = new TreeSet<>(stateCounts.keySet());
    for (String state : states)
      report.append(String.format("Zachorowania w stanie %s: %d\n",
        state, stateCounts.get(state)));
    report.append(String.format("Suma przypadków: %d\n", totalCases));
    return report.toString();
  }

  private void calculateCounties(String[] lines) {
    totalCases = 0;
    stateCounts.clear();
    counties.clear();

    for (String line : lines) {
      String[] tokens = line.split(",");
      County county = new County();
      county.county = tokens[0].trim();
      county.state = tokens[1].trim();
      // policz średnią kroczącą
      int lastDay = tokens.length - 1;
      int firstDay = lastDay - 7 + 1;
      if (firstDay < 2)
        firstDay = 2;
      double n = lastDay - firstDay + 1;
      int sum = 0;
      for (int day = firstDay; day <= lastDay; day++)
        sum += Integer.parseInt(tokens[day].trim());
      county.rollingAverage = (sum / n);

      // policz sumę przypadków zachorowań
      int cases = 0;
      for (int i = 2; i < tokens.length; i++)
        cases += (Integer.parseInt(tokens[i].trim()));
```


Kup książkę Poleć książkę

```
        totalCases += cases;
        int stateCount = stateCounts.getOrDefault(county.state, 0);
        stateCounts.put(county.state, stateCount + cases);
        counties.add(county);
      }
    }

    public static class County {
      public String county = null;
      public String state = null;
      public double rollingAverage = Double.NaN;
    }
  }
```

Skoro zmienne stały się polami, możemy dalej do woli wyodrębniać i zmieniać nazwy.

```
  public class NewCasesReporter {
    private int totalCases;
    private final Map<String, Integer> stateCounts = new HashMap<>();
    private final List<County> counties = new ArrayList<>();

    public String makeReport(String countyCsv) {
      String[] countyLines = countyCsv.split("\n");
      calculateCounties(countyLines);

      StringBuilder report = makeHeader();
      report.append(makeCountyDetails());
      report.append("\n");
      report.append(makeStateTotals());
      report.append(String.format("Suma przypadków: %d\n", totalCa-
  ses));
      return report.toString();
    }

    private void calculateCounties(String[] countyLines) {
      totalCases = 0;
      stateCounts.clear();
      counties.clear();

      for (String countyLine : countyLines)
        counties.add(calculateCounty(countyLine));
    }
```


Kup książkę Poleć książkę

```
private County calculateCounty(String line) {
  County county = new County();
  String[] tokens = line.split(",");
  county.county = tokens[0].trim();
  county.state = tokens[1].trim();

  county.rollingAverage = calculateRollingAverage(tokens);

  int cases = calculateSumOfCases(tokens);
  totalCases += cases;
  incrementStateCounter(county.state, cases);

  return county;
}

private double calculateRollingAverage(String[] tokens) {
  int lastDay = tokens.length - 1;
  int firstDay = lastDay - 7 + 1;
  if (firstDay < 2)
    firstDay = 2;
  double n = lastDay - firstDay + 1;
  int sum = 0;
  for (int day = firstDay; day <= lastDay; day++)
    sum += Integer.parseInt(tokens[day].trim());
  return (sum / n);
}

private int calculateSumOfCases(String[] tokens) {
  int cases = 0;
  for (int i = 2; i < tokens.length; i++)
    cases += (Integer.parseInt(tokens[i].trim()));
  return cases;
}

private void incrementStateCounter(String state, int cases) {
  int stateCount = stateCounts.getOrDefault(state, 0);
  stateCounts.put(state, stateCount + cases);
}

private StringBuilder makeHeader() {
  return new StringBuilder("" +
    "Hrabstwo     Stan     Średnia liczba nowych przypadków\n" +
    "========     ====     ================================\n");
}
```


Kup książkę Poleć książkę

```
  private StringBuilder makeCountyDetails() {
    StringBuilder countyDetails = new StringBuilder();
    for (County county : counties) {
      countyDetails.append(String.format("%-13s%-9s%.2f\n",
        county.county,
        county.state,
        county.rollingAverage));
    }
    return countyDetails;
  }

  private StringBuilder makeStateTotals() {
    StringBuilder stateTotals = new StringBuilder();
    TreeSet<String> states = new TreeSet<>(stateCounts.keySet());
    for (String state : states)
      stateTotals.append(String.format("Zachorowania w stanie %s:
%d\n",
        state, stateCounts.get(state)));
    return stateTotals;
  }

  public static class County {
    public String county = null;
    public String state = null;
    public double rollingAverage = Double.NaN;
  }
}
```

Jest dużo lepiej, ale nie podoba mi się to, że kod formatowania raportu znajduje się w tej samej klasie co wykonujący obliczenia na danych. To naruszenie reguły jednej odpowiedzialności, bo bardzo prawdopodobne, że format raportu i obliczenia będą się zmieniać z różnych powodów.

Aby przenieść część kodu związaną z obliczeniami do nowej klasy o nazwie `NewCasesCalculator`, zastosujemy refaktoryzację **wyodrębnianie nadklasy** (ang. *Extract Superclass*). Typ `NewCasesReporter` będzie po niej dziedziczyć.

```
public class NewCasesCalculator {
  protected final Map<String, Integer> stateCounts = new
HashMap<>();
  protected final List<County> counties = new ArrayList<>();
  protected int totalCases;
```


Kup książkę Poleć książkę

```
protected void calculateCounties(String[] countyLines) {
  totalCases = 0;
  stateCounts.clear();
  counties.clear();

  for (String countyLine : countyLines)
    counties.add(calculateCounty(countyLine));
}

private County calculateCounty(String line) {
  County county = new County();
  String[] tokens = line.split(",");
  county.county = tokens[0].trim();
  county.state = tokens[1].trim();

  county.rollingAverage = calculateRollingAverage(tokens);

  int cases = calculateSumOfCases(tokens);
  totalCases += cases;
  incrementStateCounter(county.state, cases);

  return county;
}

private double calculateRollingAverage(String[] tokens) {
  int lastDay = tokens.length - 1;
  int firstDay = lastDay - 7 + 1;
  if (firstDay < 2)
    firstDay = 2;
  double n = lastDay - firstDay + 1;
  int sum = 0;
  for (int day = firstDay; day <= lastDay; day++)
    sum += Integer.parseInt(tokens[day].trim());
  return (sum / n);
}

private int calculateSumOfCases(String[] tokens) {
  int cases = 0;
  for (int i = 2; i < tokens.length; i++)
    cases += (Integer.parseInt(tokens[i].trim()));
  return cases;
}
```


Kup książkę Poleć książkę

```
  private void incrementStateCounter(String state, int cases) {
    int stateCount = stateCounts.getOrDefault(state, 0);
    stateCounts.put(state, stateCount + cases);
  }

  public static class County {
    public String county = null;
    public String state = null;
    public double rollingAverage = Double.NaN;
  }
}

=======

public class NewCasesReporter extends NewCasesCalculator {
  public String makeReport(String countyCsv) {
    String[] countyLines = countyCsv.split("\n");
    calculateCounties(countyLines);

    StringBuilder report = makeHeader();
    report.append(makeCountyDetails());
    report.append("\n");
    report.append(makeStateTotals());
    report.append(String.format("Suma przypadków: %d\n", totalCa-
ses));
    return report.toString();
  }

  private StringBuilder makeHeader() {
    return new StringBuilder("" +
      "Hrabstwo     Stan     Średnia liczba nowych przypadków\n" +
      "========     ====     ================================\n");
  }

  private StringBuilder makeCountyDetails() {
    StringBuilder countyDetails = new StringBuilder();
    for (County county : counties) {
      countyDetails.append(String.format("%-13s%-9s%.2f\n",
        county.county,
        county.state,
        county.rollingAverage));
    }
    return countyDetails;
  }
```


Kup książkę Poleć książkę

```
  private StringBuilder makeStateTotals() {
    StringBuilder stateTotals = new StringBuilder();
    TreeSet<String> states = new TreeSet<>(stateCounts.keySet());
    for (String state : states)
      stateTotals.append(String.format("Zachorowania w stanie %s:
%d\n",
        state, stateCounts.get(state)));
    return stateTotals;
  }
}
```

Dzięki temu podziałowi poszczególne elementy zostały ładnie odseparowane od siebie. Raportowanie i obliczenia wykonywane są w oddzielnych modułach. A wszystko to dzięki początkowemu *wyodrębnieniu pola*.

KOSTKA RUBIKA

Do tej pory próbowałem pokazać możliwości kryjące się w małym zbiorze refaktoryzacji. W zwykłej pracy rzadko korzystam z innych niż te, które tu pokazałem. Rzecz w tym, by się ich dobrze nauczyć i poznać wszystkie niuanse środowiska IDE oraz związane z nimi kruczki.

Często porównywałem refaktoryzację do układania kostki Rubika. Jeśli nie udało Ci się nigdy rozwiązać jednej z tych logicznych łamigłówek, warto się tego nauczyć. Gdy poznasz zasady, stanie się to dość proste.

Okazuje się, że istnieje zestaw „operacji" wykonywanych na kostce, dzięki którym można zachować większość jej pozycji, a niektóre zmienić w przewidywalny sposób. Poznanie trzech lub czterech takich operacji pozwoli Ci stopniowo dojść do rozwiązania.

Im więcej znasz operacji i im sprawniej je wykonujesz, tym szybciej i w prostszy sposób ułożysz kostkę. Lepiej jednak naucz się ich dobrze. Wystarczy, że pominiesz jeden krok, a układanie kostki skończy się klapą i trzeba będzie zaczynać wszystko od nowa.

Kup książkę Poleć książkę

Z refaktoryzacją kodu jest bardzo podobnie. Im więcej znasz technik i im bardziej sprawnie się nimi posługujesz, tym łatwiej Ci będzie pchać, ciągnąć i rozszerzać kod w pożądanych przez siebie kierunkach.

Och, miej też w pogotowiu testy. Bez nich prawie na pewno skończy się to klapą.

PROCEDURY

Refaktoryzacja stosowana stale i w zdyscyplinowany sposób jest bezpieczna, łatwa i daje duże możliwości. Jeśli jest jednak czynnością doraźną, tymczasową i sporadyczną, jej bezpieczeństwo i potencjał szybko nikną.

TESTY

Podstawową procedurą są oczywiście testy. Testy, testy, testy i jeszcze raz testy. Aby bezpiecznie i niezawodnie refaktoryzować kod, musisz mieć zestaw testów, któremu możesz w pełni zaufać. Potrzebujesz testów.

SZYBKIE TESTY

Testy muszą też być szybkie. Refaktoryzacja po prostu nie będzie wychodzić, jeśli wykonanie testów będzie trwało godziny (czy nawet minuty).

W przypadku dużych systemów mimo najusilniejszych starań nie sposób ograniczyć czasu wykonywania testów do mniej niż kilku minut. Z tego powodu wolę organizować swój zestaw testów tak, by móc szybko i łatwo uruchamiać *odpowiedni podzestaw* testów sprawdzających tę część kodu, którą akurat refaktoryzuję. Zwykle pozwala to zmniejszyć czas ich wykonywania z kilku minut do ułamków sekundy. Cały zestaw testów uruchamiam mniej więcej co kilka godzin tylko po to, by upewnić się, czy nie wyszły na jaw jakieś błędy.

Kup książkę Poleć książkę

ZERWIJ Z GŁĘBOKĄ ZGODNOŚCIĄ JEDEN DO JEDNEGO

Zaprojektowanie testów w sposób pozwalający na uruchamianie odpowiednich podzestawów oznacza, że na poziomie modułów i komponentów struktura testów będzie odzwierciedlała strukturę kodu. Pomiędzy wysokopoziomowymi modułami testowymi a takimiż modułami kodu najprawdopodobniej będzie zachodzić zgodność jeden do jednego.

Jak dowiedzieliśmy się w poprzednim rozdziale, głęboka zgodność jeden do jednego pomiędzy testami a kodem prowadzi do kruchości testów.

Korzyści związane z szybkością, wynikające z możliwości uruchamiania odpowiednich podzestawów testów, są dużo większe niż koszty sprzężenia jeden do jednego *na tym poziomie*. Jednak dla zapobieżenia kruchości testów nie chcemy, by ta zgodność szła dalej. Poniżej poziomu modułów i komponentów celowo z nią zrywamy.

STALE REFAKTORYZUJ

Gdy przyrządzam jakieś danie, z reguły zmywam po tym naczynia[10]. Nie pozwalam, by nagromadziły się w zlewie. Gdy jedzenie się gotuje, zawsze jest czas, by umyć brudne naczynia i miski.

Podobnie jest z refaktoryzacją. Nie czekaj z nią. Refaktoryzuj w trakcie. Pilnuj w głowie pętli czerwone → zielone → refaktoryzacja i kręć się po niej co kilka minut. Dzięki temu nie pozwolisz, by powstał tak wielki bałagan, że aż zacznie Cię odstraszać.

REFAKTORYZUJ BEZWZGLĘDNIE

Bezwzględna refaktoryzacja to jeden z ukutych przez Kenta Becka sloganów programowania ekstremalnego. To dobre hasło. Technika ta polega po prostu na odważnym dokonywaniu refaktoryzacji. Nie bój się próbować. Nie wahaj

[10] Moja żona ma na ten temat inne zdanie.

Kup książkę Poleć książkę

się przed dokonywaniem zmian. Manipuluj kodem, tak jak robi z gliną rzeźbiarz. Strach przed kodem jest zgubny dla umysłu, jest niczym mroczna ścieżka. Jeśli zaczniesz nią iść, zdominuje na zawsze Twój los. Owładnie Tobą strach.

NIECH WYNIKI TESTÓW BĘDĄ STALE POZYTYWNE!

Czasami uświadamiasz sobie, że w strukturze kodu tkwi błąd i dużą jego część trzeba zmienić. Może to nastąpić wtedy, gdy pojawi się nowe wymaganie, przez które obecny projekt stanie się nieprawidłowy. Może to się też zdarzyć znienacka, gdy pewnego dnia nagle zdasz sobie sprawę, że istnieje struktura, która zapewni projektowi lepszą przyszłość.

Musisz się wykazać zarówno stanowczością, jak i sprytem. Nigdy nie psuj testów! Albo raczej nigdy nie zostawiaj ich w takim stanie dłużej niż przez kilka minut.

Jeśli do ukończenia restrukturyzacji konieczne są godziny albo dni, dokonuj jej małymi porcjami, tak by wszystkie testy były nadal wykonywane pomyślnie, a w międzyczasie zajmuj się innymi sprawami.

Załóżmy na przykład, że uświadamiasz sobie konieczność zmiany podstawowej struktury danych w systemie — struktury używanej w dużej części kodu. Gdyby to zrobić, te fragmenty kodu, podobnie jak wiele testów, przestałyby działać.

W tej sytuacji lepiej utworzyć nową strukturę danych dublującą zawartość starej, a potem stopniowo podmieniać w każdym fragmencie kodu starą strukturę na nową przy zachowaniu pomyślnych wyników testów.

W międzyczasie możesz też dodawać nowe funkcje i poprawiać błędy zgodnie ze zwykłym harmonogramem prac. Nie ma potrzeby prosić o specjalny czas na przeprowadzenie takiej restrukturyzacji. Możesz cały czas wykonywać inną pracę, a przy okazji dokonywać manipulacji w kodzie do chwili, gdy stara struktura danych przestanie być używana i będzie mogła zostać usunięta.

Kup książkę Poleć książkę

Może to zająć tygodnie, a nawet miesiące, zależnie od tego, na ile znacząca jest ta restrukturyzacja. Mimo tego system będzie gotowy do wdrożenia w każdej chwili. Nawet jeśli restrukturyzacja zostanie dokonana tylko częściowo, testy wciąż będą wykonywały się pomyślnie, a system będzie można wdrażać do środowiska produkcyjnego.

POZOSTAW SOBIE WYJŚCIE

Piloci są uczeni, by zawsze zostawiać sobie drogę ucieczki, na przykład wtedy, gdy wlatują w obszar o niezbyt dobrych warunkach pogodowych. Dość podobnie jest z refaktoryzowaniem. Czasem zaczynasz serię refaktoryzacji, która po godzinie lub dwóch doprowadza do ślepego zaułka. Pierwotna koncepcja z jakiegoś powodu okazuje się niewypałem.

W takich sytuacjach może się przydać polecenie `git reset --hard`.

Gdy bierzesz się za taką refaktoryzację, koniecznie dodaj tag do repozytorium źródłowego, by w razie potrzeby można było do niego wrócić.

ZAKOŃCZENIE

Celowo zachowałem zwięzłą formę tego rozdziału, bo chciałem dorzucić tylko kilka przemyśleń do *Refaktoryzacji* Martina Fowlera. Jeszcze raz zachęcam do zapoznania się z tą książką, by dogłębnie zrozumieć temat.

Najlepszym podejściem do refaktoryzacji jest opracowanie wygodnego repertuaru często stosowanych refaktoryzacji i posiadanie dobrej praktycznej znajomości wielu innych. Jeśli korzystasz ze środowiska IDE zapewniającego operacje refaktoryzacyjne, postaraj się je dokładnie poznać.

Refaktoryzacja bez testów nie ma sensu. Jest zbyt wiele okazji do pomyłki. Nawet automatyczne refaktoryzacje w środowisku IDE czasami powodują błędy. *Zawsze* więc wspieraj swoje starania refaktoryzacyjne kompleksowym zestawem testów.

Zachowaj wreszcie dyscyplinę. Refaktoryzuj często, bezwzględnie i bez przepraszania. Nigdy, przenigdy nie proś o pozwolenie na refaktoryzację.

Kup książkę Poleć książkę


PROGRAM
PARTNERSKI
GRUPY HELION
1. ZAREJESTRUJ SIĘ
2. PREZENTUJ KSIĄŻKI
3. ZBIERAJ PROWIZJĘ
Zmień swoją stronę WWW w działający bankomat!
Dowiedz się więcej i dołącz już dzisiaj!
http://program-partnerski.helion.pl
GRUPA
Helion

Szczyć się swoją pracą i utrzymuj wysoki standard!

Rzemieślnik to osoba, która jest starannie wyszkolona, doskonale zna się na swoim fachu i czuje dumę z własnej pracy. Zawsze zachowuje właściwą dla zawodu godność i profesjonalizm. Społeczeństwo okazuje zaufanie rzemieślnikom, wierząc, że należycie dbają o wysoką jakość i postępują etycznie. Pisanie oprogramowania to też rzemiosło. I programiści, podobnie jak inni rzemieślnicy, również chcą czuć dumę i satysfakcję ze swojej pracy. Problem w tym, że świat zdaje się dziś wymagać od nich przede wszystkim produktywności, a nie wysokiej jakości kodu.

W tej książce znajdziesz zasady definiujące rzemiosło, jakim jest wytwarzanie oprogramowania. Zebrano w niej procedury, standardy i normy etyczne, dzięki którym tworzony kod będzie niezawodny i efektywny, a całe oprogramowanie stanie się powodem do dumy. Zawarto tutaj szereg pragmatycznych wskazówek dotyczących procedur programistycznego rzemiosła. Omówiono też standardy, co powinno ułatwić zrozumienie oczekiwań wobec programistów. Ważnym zagadnieniem jest etyczny kontekst zawodu programisty, czyli fundamentalne zobowiązania, które programiści powinni podjąć względem swojego otoczenia i siebie samych. Istotą rzemiosła programistycznego bowiem jest tworzenie kodu, który budzi zaufanie użytkowników i całych społeczności

Najciekawsze zagadnienia:

- **czym jest prawdziwe rzemiosło programistyczne**
- **pięć podstaw: programowanie sterowane testami, refaktoryzacja, prostota projektu, programowanie zespołowe i testy akceptacyjne**
- **produktywność, jakość i odwaga w zespołach programistów**
- **czym w rzeczywistości jest uczciwość i praca zespołowa**
- **dziesięć zobowiązań profesjonalnego programisty**

Robert C. Martin (znany w środowisku jako **Wujek Bob**) jest legendarnym programistą, w zawodzie pracuje od 1970 roku. Obecnie prowadzi konsultacje, szkolenia i pomaga w rozwijaniu umiejętności w największych korporacjach. Napisał wiele bestsellerowych książek technicznych, opublikował dziesiątki artykułów w fachowych czasopismach i regularnie występuje na międzynarodowych konferencjach. Przez trzy lata pełnił funkcję redaktora naczelnego magazynu „C++ Report", był także pierwszym prezesem Agile Alliance.

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

KOD KORZYŚCI
Sięgnij po więcej!

ISBN 978-83-283-9056-0

9 788328 390560

Cena: 99,00 zł

Pearson
Addison-Wesley