

Dariusz Woźniak

TDD

TECHNIKI PROGRAMOWANIA
STEROWANEGO TESTAMI



Helion 

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

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

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

Redaktor prowadzący: Małgorzata Kulik

Projekt okładki: Studio Gravite / Olsztyn
Obarek, Pokoński, Pazdrijowski, Zaprucki

Grafika na okładce została wykorzystana za zgodą Shutterstock.com

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

Drogi Czytelniku!

Jeżeli chcesz ocenić tę książkę, zajrzyj pod adres

<http://helion.pl/user/opinie/tddppr>

Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

Kody źródłowe wybranych przykładów dostępne są pod adresem:

<ftp://ftp.helion.pl/przyklady/tddppr.zip>

ISBN: 978-83-283-3531-8

Copyright © Helion 2018

Printed in Poland.

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

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

Spis treści

Podziękowania	11
Przedmowa	13
Cytaty o TDD	15
Wstęp	17
Przewodnik po książce	19
Dla kogo jest ta książka?	20
W jakim stopniu muszę umieć programować?	20
Nie jestem programistą C#. Czy ta książka ma jakąś wartość dla mnie?	20
Umiem pisać testy jednostkowe i znam TDD. Czy ta książka jest dla mnie?	21
Jestem menadżerem/dyrektorem/właścicielem (a więc nie programuję), a mój zespół chce wdrożyć TDD. Czy z tej książki dowiem się, czy warto?	22
Jestem manualnym testerem. Czy powinienem znać TDD?	22
Kontekst jest królem	23
Atrapa to mock czy test double?	23
xUnit a xUnit.net	24
Funkcja a funkcjonalność oraz funkcyjny a funkcjonalny	24
Synonimy	25
Jak uczyć się metody Test-Driven Development?	25
Dogmat	26
Narzędzia użyte w tej książce	27
Kod źródłowy do książki	28
Rozdział 1. Wprowadzenie do TDD	29
Błędy w oprogramowaniu	31
Cykl Red-Green-Refactor	32
Podsumowanie	33

Rozdział 2. Co zyskujemy dzięki TDD?	35
Wysoka jakość kodu	35
Czy pisanie w metodyce TDD oznacza pisanie według zasad SOLID?	36
Prostota kodu: YAGNI i KISS	37
Żywa dokumentacja	38
Lepsze zrozumienie wymagań biznesowych	38
Automatyczna regresja	39
Informacja zwrotna	41
Mniej defektów	42
Czas programowania jest krótszy	43
Niższy koszt zmian	43
Przypadki użycia	45
Badanie Microsoft Research nad zespołami Microsoftu i IBM	45
Pilotażowy projekt dla dużej firmy	45
Mały projekt w parach	47
Metaanaliza	47
Podsumowanie	48
Rozdział 3. Trudności przy wdrażaniu TDD	49
Ścieżka nauki	49
Dyscyplina	50
Więcej narzędzi	50
Początkowa percepcja dłuższego czasu potrzebnego do napisania kodu	51
Jak „sprzedać” TDD kierownictwu?	51
Musimy dostarczać szybko, a w naszym projekcie nie ma czasu na TDD	51
Lista kontrolna: Definition of Done	52
Podsumowanie	56
Rozdział 4. Filozofia TDD	57
Test-First czy Test-Last?	58
Wszystkie testy naraz czy test po teście?	59
Weryfikacja stanu czy zachowania?	60
Jedna czy wiele asercji?	60
Scenariusz 1.: Jedna oczekiwana zmienna wyjściowa	60
Scenariusz 2.: Wiele oczekiwanych zmiennych wyjściowych	61
Scenariusz 3.: Asercje pośrednie	62
Wiele asercji w jednym teście	66
Kiedy pisanie testów jednostkowych nie ma sensu?	67
Czy należy pisać testy jednostkowe do bibliotek innych dostawców?	68

Czy TDD sprawdza się dla małych aplikacji?	69
Czy TDD sprawdza się dla dużych aplikacji?	70
Czy testy jednostkowe zastępują testera?	70
Jak pisanie testów jednostkowych wpływa na estymatę zadania?	71
Czy testy jednostkowe można pisać w innym języku programowania niż pisany jest kod?	71
Czy testy jednostkowe może pisać inna osoba?	72
Czy system może mieć zbyt dużo testów jednostkowych?	72
Czy testy jednostkowe to jedyne testy, jakie powinny znajdować się w aplikacji?	73
Jakich testów powinno być najwięcej (piramida testów)?	74
Podsumowanie	75

Rozdział 5. Rodzaje testów 77

Sposób wykonywania	77
Wiedza na temat struktury systemu (test skrzynki)	77
Poziom testowania	78
Testowanie po zmianach dokonanych w systemie	80
Testowanie niefunkcjonalne	80
Test wydajnościowy	81
Podsumowanie	82

Rozdział 6. Test jednostkowy 83

Struktura testu: Arrange-Act-Assert	83
Alternatywne struktury testu	84
Test jednostkowy a test integracyjny	85
Myśl jak tester: ścieżki optymistyczne i przypadki brzegowe	86
Jak nazywać klasy i metody testowe?	87
Podział testów i projekty testowe	88
Podsumowanie	89

Rozdział 7. Nasz pierwszy test jednostkowy 91

Wybór biblioteki do testowania	91
Zanim zaczniemy... ..	92
Dodanie solucji i projektów	92
Dodanie biblioteki NUnit	93
Etap red: pisanie testu do nieistniejącej metody	95
Jak uruchomić test?	98
Etap green: implementacja kodu	102
Etap trzeci (i ostatni): refaktoryzacja kodu	103
Podsumowanie	103

Rozdział 8. Piszemy kolejne testy jednostkowe 105

Drugi test jednostkowy	105
Kolejne przypadki użycia	106
Testy ułamków nieskończonych lub zaokrąglonych	108
Testowanie wyrzucenia wyjątku	109
Testowanie zdarzenia	111
Podsumowanie	114

Rozdział 9. Testowanie z NUnitem 115

Asercje	116
Model klasyczny i model oparty na twierdzeniach	118
Operacja równości	120
Porównanie dwóch typów wartościowych	121
Porównanie dwóch typów referencyjnych	122
Porównanie dwóch typów referencyjnych z nadpisanym operatorem porównania	122
Tolerancja: delta i procent	124
Tolerancja: czas	124
Własna klasa obsługująca porównanie	125
Metody pomocnicze	126
Operacje porównania	126
Własna klasa obsługująca porównanie	127
Należy do zakresu	128
Złożenia	128
Testowanie typów	129
Testowanie wyjątków	131
Testowanie, czy kod wyrzucił wyjątek	133
Testowanie, czy kod nie wyrzucił wyjątku	135
Testowanie parametru i komunikatu wyjątku	136
Testowanie wewnętrznego wyjątku	137
Typ tekstowy	137
Kolekcje	138
System plików	141
Komunikaty	142
Własne komunikaty błędów	144
Własne komunikaty informacyjne	145
Komunikat a nazwa testu	146
Współdzielenie danych	147
Kiedy korzystać ze współdzielenia danych?	148

Testy parametryzowane	150
TestCase	151
Values	152
Range	154
Random	154
TestCaseSource	155
ValueSource	160
Testy oparte na zewnętrznych źródłach	160
Strategie łączenia wartości testowych	161
Test kombinatoryczny	161
Test sekwencyjny	162
Test par	163
Teorie	166
Testowanie klas generycznych	170
Zasada podstawienia Liskov	172
Testowanie wywołań asynchronicznych	174
Równoległe uruchamianie testów	178
Poziom zrównoleglenia	179
Kiedy zrównoleglić uruchamianie testów?	179
Pozostałe atrybuty	179
Sterowanie wątkiem	180
Kategoria testu	180
Atrybuty informacyjne	182
Przekazywanie parametrów	182
Ignorowanie testów	183
Kolejność wykonywania testów	185
Ustawienia regionalne	185
Powtarzanie testu	188
Czas wykonywania testu	189
Platforma	190
Atrybuty a testy parametryzowane	192
Podsumowanie	192

Rozdział 10. Testowanie zależności i atrapy obiektów 193

Ręczne tworzenie atrapy	195
Kryterium akceptacji nr 1: wiek klienta niższy niż 18 lat	196
Kryterium akceptacji nr 2: wiek klienta większy bądź równy 18 lat	198
Kryterium akceptacji nr 3: jeśli obiekt klienta jest nullem, to wyrzuc wyjątek	200
Podsumowanie	200

Wprowadzenie do frameworku Moq	201
Składnia imperatywna i deklaratywna	202
Składnia imperatywna	203
Składnia deklaratywna	204
Wybór składni	205
Atrapa rekursywna (recursive mock)	205
Tryb zachowania właściwości (stubbing)	206
Zwracanie domyślnej wartości	207
Atrapa z sekwencyjnym rezultatem	208
Tryb zachowania atrapy (MockBehavior)	209
Przekazywanie parametrów w metodzie (argument matchers)	210
Ponowne użycie matcherów	215
Weryfikacja wywołań	216
Weryfikacja wywołania metody	217
Weryfikacja dostępu i zapisu właściwości	219
Testować stan czy zachowanie?	220
Komunikat błędu	221
Podsumowanie	222
Wywołanie zwrotne: Callback	222
Podsumowanie	224
Wywołanie składowej bazowej: CallBase	224
Atrapa wyrzucająca wyjątek	226
Inne poziomy dostępności	227
protected	228
internal	229
Podsumowanie	229
Klasyfikacja atrap	230
Dummy	230
Stub	232
Fake	233
Mock	235
Spy	235
Podsumowanie	236
Ograniczenia Moqa	237
Tworzenie atrap dla klas i metod statycznych	238
Rodzaje bibliotek do tworzenia atrap	241
Constrained	241
Unconstrained	241
Constrained czy unconstrained?	242
Podsumowanie	243

Rozdział 11. Dobre praktyki pisania testów jednostkowych 245

Test powinien być szybki, bardzo szybki!	246
Testy powinny być odizolowane i niezależne od siebie	247
Test powinien być powtarzalny	247
Test powinien być deterministyczny	248
Test nie powinien mieć zależności zewnętrznych	248
Test nie powinien mieć konfiguracji	249
Wynik testu nie powinien być interpretowany	249
Test nie powinien być pusty	250
Załączek kodu powinien wyrzucać wyjątek	250
Test powinien mieć jedną logiczną asercję	251
Testy nie powinny być dyskryminowane	251
Testy powinny być podzielone według kategorii	251
Test powinien mieć strukturę Arrange-Act-Assert	251
Test powinien obejmować ścieżki optymistyczne i przypadki brzegowe	252
Test powinien mieć odpowiednią nazwę	252
Testowane powinny być tylko publiczne składowe	252
Test powinien oczekiwać konkretnego typu wyjątku	253
Test powinien oczekiwać wyjątku w konkretnym wyrażeniu	253
Test nie powinien zawierać instrukcji warunkowych i pętli	253
Test powinien mieć wartości oczekiwane wpisane „na sztywno”	254
Test powinien mieć asercję	255
Test powinien być nieskomplikowany	255
Test nie powinien być „przespecyfikowany”	256
Test nie powinien zawierać metod z atrybutami SetUp i TearDown	257
Klasa testowa powinna być bezstanowa	257
Komunikaty asercji nie powinny być nadmiarowe	258
Podsumowanie	259

Rozdział 12. TDD i istniejący kod 261

Refaktoryzacja bezpieczna i mniej bezpieczna	261
Przykład bezpiecznej refaktoryzacji	264
Dodawanie testów do istniejącego kodu	271
Gdzie zacząć dodawać testy?	271
Jak pisać testy?	272
Narzędzia	274
Podsumowanie	275

Rozdział 13. Pokrycie kodu testami	277
Co to jest pokrycie kodu testami?	277
Narzędzia do mierzenia pokrycia kodu	278
W ile procent pokrycia powinniśmy celować?	281
Przykłady „fałszywych” testów o stuprocentowym pokryciu kodu	281
Podsumowanie	283
85%, 90% czy 100%?	283
Pokrycie kodu jako narzędzie do identyfikowania brakujących testów	284
 Rozdział 14. Ciągła integracja	 285
Serwer ciągłej integracji	286
Ciągła dostawa i ciągle wdrażanie	288
Podsumowanie	289
 Dodatek A. Biblioteki do testowania	 291
 Dodatek B. Biblioteki do tworzenia atrap	 293
 Dodatek C. Biblioteki do mierzenia pokrycia kodu testami	 297
 Dodatek D. Testy z danymi zewnętrznymi — przypadek użycia	 299
 Dodatek E. Rozszerzalność NUnita	 303
Atrybut informacyjny	305
Atrybut umożliwiający współdzielenie danych	307
 Dodatek F. Bibliografia	 311
Źródła internetowe	314
 Skorowidz	 319

Podziękowania

Tę książkę dedykuję mojej fantastycznej Żonie Agnieszce, kochającym Rodzicom oraz wspaniałemu Bratu.

Dziękuję Ani Merak, Łukaszowi Oliwie i Patrykowi Spytkowskiemu za wysiłek włożony w dodatkową korektę książki. Dziękuję też Markowi Krzemińskiemu za pomysł na książkę i motywację do jej pisania.

Chciałbym też podziękować całej mojej rodzinie, wszystkim przyjaciołom, współpracownikom oraz ludziom, z którymi zamieniłem parę zdań czy kilka słów.

Dziękuję też Tobie, Czytelniku, że zdecydowałeś się na lekturę tej książki. Mam ogromną nadzieję, że znajdziesz tutaj wiele informacji, które wykorzystasz w swojej pracy programisty i rozwoju w tym kierunku. Mam też nadzieję, że lektura tej książki będzie dla Ciebie czystą przyjemnością!

Przedmowa

Jakie jest najważniejsze zadanie programisty? Jedni powiedzą, że napisanie działającego kodu. Ja uważam jednak, że działający kod jest jedynie efektem ubocznym naszej pracy. Przede wszystkim powinniśmy tworzyć kod utrzymywalny. Dzięki temu poprawienie niedziałających rozwiązań nie stanowi karkołomnego wyczynu na miarę wyprawy na Mount Everest, a rozwijanie systemów informatycznych staje się procesem przewidywalnym, przyjemnym i — co niezmiennie istotne — jego koszt jest niezmienny w czasie.

Testowanie oprogramowania to praktyka wspomagająca nas w tych staraniach. Jedne środowiska bezdyskusyjnie stosują i promują pisanie testów, nieustannie dbając o swój rozwój w tym zakresie. Inne natomiast kwestionują sensowność takiego podejścia.

Negatywne nastawienie programistów do pisania testów często wynika z niewiedzy i braku odpowiedniej ścieżki nauki. Jest to trudna umiejętność, wymagająca praktyki i nakładu czasu. Niejednokrotnie trzeba popełnić wiele błędów i wyciągnąć z nich wnioski, by finalnie odnieść długoterminowe korzyści. Aspekt edukacji w zakresie tej sztuki jest często pomijany, a bez tego żadna zaawansowana technika nie ma szans na spełnienie oczekiwań.

Testy nie są lekiem na całe zło. Samo ich pisanie nie gwarantuje natychmiastowego wyeliminowania błędów. Dopiero stopniowe, świadome i konsekwentne pogłębianie wiedzy skutkuje usprawnieniem procesu wytwarzania oprogramowania oraz niezmiennie ułatwia jego rozwój, utrzymanie i modyfikację.

Umiejętność pisania wartościowych testów jednostkowych to jedna z najpewniejszych inwestycji, na jaką może zdecydować się programista.

— Maciej Aniserowicz, autor książki *Zawód: Programista* i bloga *devstyle.pl*

Cytaty o TDD

Kod bez testów to zły kod. Nie ma znaczenia, jak dobrze jest napisany; nie ma znaczenia, jaki jest ładny, jak bardzo zorientowany obiektowo czy też jak mocno hermetyczny. Za pomocą testów możemy zmienić zachowanie naszego kodu szybko i w sposób weryfikowalny. Bez nich tak naprawdę nie wiemy, czy kod zmierza ku lepszemu czy ku gorszemu.

— Michael Feathers, *Praca z zastanym kodem. Najlepsze techniki*

TDD nie jest po to, by pisać dobry kod, lecz raczej po to, by nie pisać złego kodu.

— Nat Pryce, *Growing Object Oriented System Guided by Tests*

Zdarza się często — i zdarzać się będzie — że TDD implementowane jest w kodzie niewyposażonym w należyty zestaw testowy. A jeżeli tak, to może się również zdarzyć, że przy refaktoryzacji kodu popełnisz błąd i nie zostanie on wykryty, bo wszystkie istniejące testy nadal będą zaliczane.

Refaktoryzacja stanie się więc niepewna, zatem ograniczysz ją do minimum. Twój projekt stanie się zagmatwany i także niepewny, Twoje samopoczucie się pogorszy, Twoja produktywność spadnie, zostaniesz zwolniony z pracy. Opuści Cię Twój pies, przestaniesz się należyście odżywiać. Nabawisz się awitaminozy, szkorbutu i chorób przyzębia. Zatem, przynajmniej w trosce o swe uzębienie, myśl retroaktywnie o testowaniu przed refaktoryzacją.

— Kent Beck, *TDD. Sztuka tworzenia dobrego kodu*

Najlepszym znanym mi sposobem pisania kodu jest kształtowanie go od początku za pomocą testów.

— Ron Jeffries, jeden z twórców programowania ekstremalnego

Najlepsze w TDD jest to, że metoda ta zapewnia, iż kod robi to, co programista myśli, że kod powinien robić.

— James Grenning

Testy czarnej skrzynki dowodzą, że kod pisany z zastosowaniem metody TDD ma lepszą jakość niż kod pisany według tradycyjnego, kaskadowego modelu.

— Bobby George i Laurie Williams,
An Initial Investigation of Test Driven Development in Industry

Kiedy ludzie pytają mnie, jak pisać lepszy kod, zawsze im odpowiadam: pisz testy jednostkowe. (...) Testowanie kodu nauczy cię, jak budować oprogramowanie.

— Scott Allen

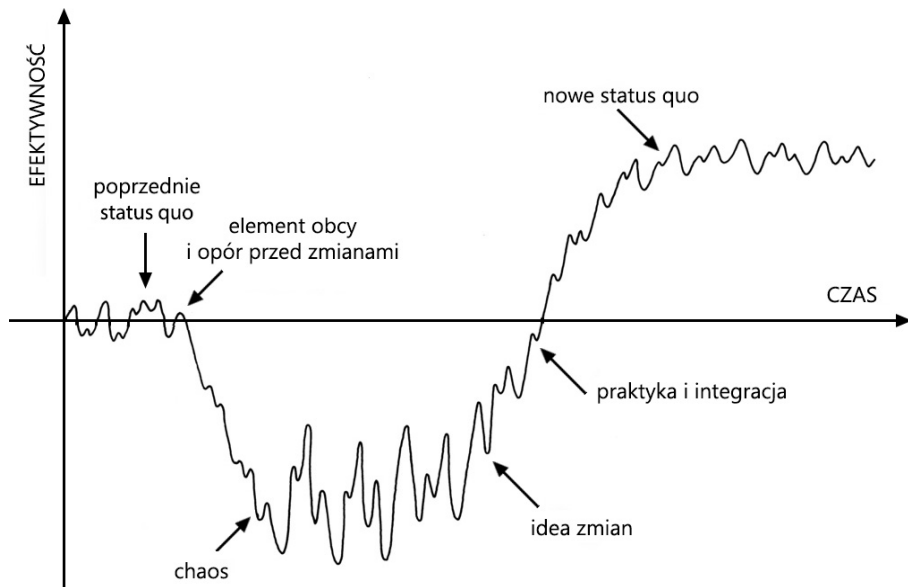
Wstęp

W języku japońskim istnieje pojęcie *kaizen*, które utożsamiane jest z filozofią nieustannych małych kroków i drobnych poprawek. Jest to przeciwieństwo filozofii *kaikaku*, która oznacza radykalną zmianę.

Zastanówmy się nad słowem kluczem, którym jest *zmiana*. Zmiana dla programisty może oznaczać nie tylko zmianę zawodu, technologii, stosowanych narzędzi, ale także zmianę w procesie wytwarzania oprogramowania czy zmianę metody programowania.

Zacznijmy od tej dużej zmiany, rewolucyjnej, *kaikaku*. Przykład? Wdrożenie frameworku Scrum w zespole. Przed wdrożeniem zespół znajduje się w sytuacji, do której jest przyzwyczajony, nie ponosi żadnego ryzyka związanego ze zmianami, doskonale wie, co go czeka dnia następnego. Status quo jest nienaruszone. Gdy pojawia się chęć zmian — a te często wprowadzane są odgórnie (element obcy) — rodzi się opór wewnątrz zespołu. Niekiedy nieznane są powody zmian, a bardzo często nie są znane również podstawy i wartości, które kryją się za tymi zmianami. Duże zmiany mogą stwarzać niekomfortowe warunki i powodować irracjonalne zachowania, których efektem jest na przykład lekceważenie nowego procesu czy brak chęci do jego zrozumienia. Po tym etapie następuje faza chaosu, w której zespół zdaje się pracować poza jakąkolwiek kontrolą. Spotkania przebiegają często bez ładu i składu. Zespół może obrać strategię powrotu do poprzedniego stanu za wszelką cenę lub strategię jak najszybszego wyjścia z tego stanu. Bywa, że te dwie strategie przenikają się w ramach zespołu i formują się dwa bastiony: jeden gotowy na zmiany, a drugi pragnący powrotu do stanu sprzed wprowadzenia zmian. Wraz z pojawieniem się idei zmian zespół zaczyna wychodzić z chaosu, a proces zmiany się stabilizuje. Gdy nowe pomysły zostaną zintegrowane, rodzi się nowe status quo, nowy stan, nowa strefa komfortu.

Jest to wzorcowy model według teorii Virginii Satir (rysunek W.1) [Appelo, 2010; Goździeniak, 2015; Brodziński, 2015]. Powyższy scenariusz wiąże się z wprowadzeniem zbyt dużej zmiany bądź zbyt dużej ilości zmian, oczekiwaniem zbyt krótkiego czasu na poprawę efektywności oraz wprowadzeniem zmian bez moderacji lub z moderacją nieudolną. Oznacza to również ryzyko w związku z wprowadzeniem zmiany, gdyż na etapie chaosu zespół może wycofać się ze zmian i powrócić do swojego poprzedniego stanu.



RYСУNEK W.1. Model zmian według Virginii Satir¹

Ale czy tak zawsze musi wyglądać każda zmiana?

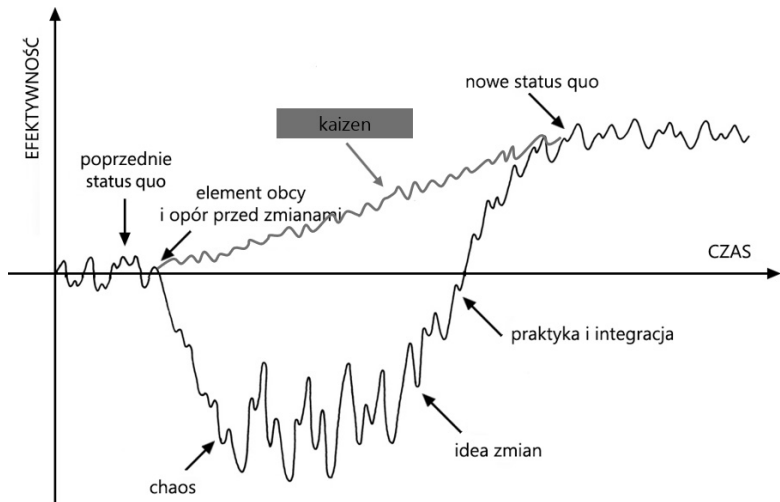
W podejściu ewolucyjnym, *kaizen*, stosujemy metodę niewielkich kroków. Wdrożenie Scruma? Można to zrobić w następujący sposób:

- Zespół analizuje poszczególne elementy Scruma i zaczyna je stosować, ale niekoniecznie wszystkie naraz.
- Zespół i kierownictwo nie powinny oczekiwać natychmiastowych efektów i błyskawicznego wzrostu wydajności.
- Zespół zbiera informacje zwrotne dotyczące każdej zmiany i adaptuje je do aktualnego środowiska.
- Jeżeli zastosowanie pewnej zmiany w pewnym czasie wpłynęło negatywnie na zespół, można rozważyć ponowną próbę jej wprowadzenia. Ta sama praktyka, która nie zadziałała wcześniej, może okazać się skuteczna w nowym kontekście.
- Nie każda zmiana wpływa pozytywnie na proces, ale dzięki temu, że jest mała, zespół uzyskuje szybką informację zwrotną na jej temat i może szybciej zareagować i zaadaptować się do nowej sytuacji.

¹ Na podstawie: <https://www.flickr.com/photos/jurgenappelo/5201852636/in/album-72157625328824303/>

Główna idea filozofii *kaizen* to drobne poprawki w procesie. Pozwalają one skutecznie wyeliminować fazę chaosu w modelu zmiany i nie wiążą zmiany z nagłą degradacją wydajności. Model małych zmian naniesiony na model Virginii Satir został przedstawiony na rysunku W.2.

RYСУNEK W.2.
Dzięki filozofii małych zmian, *kaizen*, możemy uniknąć etapu chaosu i utrzymać coraz lepszą efektywność w czasie



Należy pamiętać o jeszcze jednej, bardzo ważnej kwestii. Otóż nie ma globalnego optimum, do którego moglibyśmy zmierzać, są za to lokalne optima. A zatem zmiana procesu i jego poprawa to proces nieustanny, ciągły².

Kaizen i *kaikaku* to nie tylko zmiany zespołowe i nie tylko związane z procesem tworzenia oprogramowania. Filozofię małych kroków, drobnych poprawek i unikania nagłych, dużych zmian można przenieść na grunt życia osobistego i własnego rozwoju.

Test-Driven Development — najpierw piszemy test, później kod, a na końcu dokonujemy refaktoryzacji. Cykl bardzo zwężony i prosty do zapamiętania, prawda? Tak, to prawda. Z TDD związana jest jednak konieczność zmiany...

Przewodnik po książce

Niniejsza książka podzielona jest na rozdziały, które umożliwiają efektywną naukę, wprowadzając Czytelnika w metodę Test-Driven Development krok po kroku.

Pierwszy rozdział jest wprowadzeniem do tematyki TDD. Opisuję w nim historyczny kontekst tej metody i podstawy, na których ją wykreowano. Szybko dowiesz się, czym jest cykl Red-Green-Refactor, i poznasz sposób pisania według tej techniki.

² Jeśli ciekawi Cię ten proces, to polecam wyszukać w internecie hasło *fitness landscapes*.

W kolejnych rozdziałach przejdziemy do rozważań teoretycznych, w których skupimy się na tym, jakie korzyści może nam przynieść TDD, a z czym mogą być kłopoty. Przyjrzymy się też najczęściej zadawanym pytaniom, na które niekiedy nie ma jedynej dobrej odpowiedzi.

Następnie przejdziemy do tematów praktycznych. Napišemy pierwsze testy jednostkowe i przyjrzymy się dokładnie bibliotece do testowania i bibliotece do tworzenia atrap. Potem omówię wzorce i antywzorce w świecie testów jednostkowych. Dowiesz się również, jakie strategie zastosować w pracy z kodem już istniejącym.

W dwóch ostatnich rozdziałach omówię pokrycie kodu testami oraz systemy ciągłej integracji, ciągłej dostawy i ciągłego wdrażania.

W pięciu dodatkach do książki zamieściłem informacje na temat bibliotek do testowania, tworzenia atrap i mierzenia pokrycia kodu dla .NET. Omówiłem również scenariusz testów z danymi zewnętrznymi i bardzo prosty przykład sposobu rozszerzenia biblioteki NUnit o nowe funkcje.

Dla kogo jest ta książka?

Książka jest skierowana głównie do programistów chcących poznać metodę TDD i testy jednostkowe.

W jakim stopniu muszę umieć programować?

Warunkiem koniecznym do zrozumienia idei zawartych w tej książce jest podstawowe doświadczenie w zakresie programowania zorientowanego obiektowo. TDD wymaga myślenia obiektowego, a zatem Czytelnik powinien znać podstawy obiektowości.

Nie jestem programistą C#.

Czy ta książka ma jakąś wartość dla mnie?

Część teoretyczna książki jest uniwersalna i ma zastosowanie w przypadku każdego języka programowania. Część praktyczna również, ale wymaga zrozumienia języka C#. Ćwiczenia praktyczne można wykonywać poprzez analogię do rozwiązań występujących w bibliotece tego właśnie języka. A zatem odpowiedź brzmi: tak! Przykłady zawarte w tej książce napisane są w języku C#, ale nie znaczy to, że jedynie programiści tego języka mogą z niej korzystać.

C# jest językiem prostym w czytaniu i rozumieniu. Rozdziały traktujące o konkretnych rozwiązaniach w bibliotekach NUnit i Moq zostały zatytułowane tak, aby osoba szukająca informacji dla innej biblioteki i innego języka mogła znaleźć je w prosty i bezbolesny sposób. W gestii Czytelnika pozostaje wyszukanie odpowiednika w stosowanej przez niego

bibliotece. Pisząc tę książkę, miałem to na względzie i wyszukiwałem analogiczne tematy, ale dla innych bibliotek. Przykładem jest rozdział z NUnitem i testami parametryzowanymi. Chcąc odnaleźć tę samą funkcję dla biblioteki JUnit (Java), należy wpisać w wyszukiwarce internetowej frazę JUnit Parameterized tests. Zazwyczaj pierwsze rezultaty będą prowadzić do dokumentacji biblioteki lub odpowiedzi na Stack Overflow, a to w zupełności wystarczy.

Co, jeśli moja ulubiona biblioteka w moim ulubionym języku nie oferuje funkcji zawartej w książce? Cóż, jeżeli jesteś dobrym programistą, nie będzie to dla Ciebie przeszkodą. W takiej sytuacji możesz wykonać jeden z następujących kroków:

- wykorzystanie innych funkcji języka programowania lub biblioteki, aby osiągnąć ten sam cel;
- własna implementacja funkcji w bibliotece;
- własna implementacja funkcji w bibliotece i kontakt z autorem biblioteki, a jeśli jest to biblioteka oparta na otwartym źródle, *pull request* do repozytorium;
- stworzenie własnej biblioteki.

Jeśli implementacja jest zbyt trudna, biblioteka nie jest oparta na otwartym źródle lub nie masz czasu na dopisanie nowych funkcji, to pozostaje kontakt z innymi programistami, np.:

- wspólna sesja w ramach swojego zespołu programistycznego i kolektywna próba rozwiązywania problemu;
- kontakt z autorem biblioteki i pytanie o możliwe rozwiązania;
- kontakt ze społecznością, np. Stack Overflow, repozytorium danej biblioteki na GitHubie (społeczność anglojęzyczna), 4programmers.net (społeczność polskojęzyczna).

Umiem pisać testy jednostkowe i znam TDD. Czy ta książka jest dla mnie?

Testy jednostkowe można pisać według techniki TDD lub bez jej stosowania. Bardzo często, gdy prowadziłem ćwiczenia z tego tematu, okazywało się, że programista potwierdzający znajomość TDD tak naprawdę nie stosował cyklu Red-Green-Refactor. Co więcej, był zaskoczony iteracyjnym podejściem do pisania kodu, co jest przecież kluczową kwestią w TDD. A zatem nie do końca znał TDD.

Ponadto testy jednostkowe można pisać podobnie jak kod, ale jak zweryfikować, czy są one poprawne i cechuje je wysoka jakość? Czy aby na pewno wszystkie zależności są zaspokajane atrapami? A co z kwestiami pokrycia kodu testami i ciągłej integracji? Czy Twój

zespół ma postawiony cel osiągnięcia określonego pokrycia kodu testami? Czy ma to sens? Czy za każdą zmianą w repozytorium uruchomiony zostaje cały zestaw testów jednostkowych dla aplikacji?

W istocie, praktykując dane zagadnienie, często robimy to źle lub moglibyśmy robić to lepiej. Skąd możemy otrzymać informację zwrotną na temat ewentualnych poprawek? Możemy przedyskutować pewne zagadnienia w ramach zespołu, możemy taką dyskusję przeprowadzić z innymi programistami i innymi zespołami, możemy również zdać się na własną analizę tematu i wykorzystać dostępne źródła. Ta książka pozwala zrozumieć wiele kwestii, które bywają pomijane w rozważaniach o testach automatycznych. Jest to jedno ze źródeł, które dostarczy Ci informacji zwrotnej na temat tego, czy wykonujesz swoją pracę prawidłowo.

Warto zrobić rachunek sumienia i spróbować zidentyfikować brakujące obszary wiedzy w zakresie testowania automatycznego, testów jednostkowych i Test-Driven Development. A ta książka Ci w tym pomoże!

Jestem menadżerem/dyrektorem/właścicielem (a więc nie programuję), a mój zespół chce wdrożyć TDD. Czy z tej książki dowiem się, czy warto?

Książka jest skierowana do programistów i mocno przesiąknięta kodem źródłowym. Jeśli interesuje Cię Test-Driven Development z punktu widzenia czysto biznesowego, to temu zagadnieniu poświęconych jest kilka rozdziałów. Przede wszystkim zachęcam do przejrzenia biznesowych przypadków użycia, które umieściłem w książce. Scenariusze, w których nie ma czasu na TDD i testy automatyczne, brakuje czasu na naukę nowych technik, trzeba dostarczać oprogramowanie szybko i na czas, są bardzo typowe dla branży programistycznej i poświęciłem tym zagadnieniom kilka podrozdziałów. W szczególności pomocne będą następujące podrozdziały: „Błędy w oprogramowaniu” (rozdział 1.), „Co zyskujemy dzięki TDD?” (rozdział 2.) oraz „Trudności przy wdrażaniu TDD” (rozdział 3.).

Jestem manualnym testerem. Czy powinienem znać TDD?

Test-Driven Development to technika dla programisty. Jeśli jesteś testerem ręcznym i interesujesz się tematyką programowania oraz chcesz rozwijać to zainteresowanie, to ta książka jest również dla Ciebie.

Bardzo dobrą techniką nauki jest *coding dojo* (więcej na jej temat przeczytasz w podrozdziale „Ścieżka nauki” rozdziału 3.). Jeżeli Twój zespół praktykuje *dojo* lub ćwiczenia o podobnej formie, nie wahaj się dołączyć. Programiści są zazwyczaj pomocni i z chęcią dzielą się swoją wiedzą z innymi. Znam przypadki, w których w *coding dojo* uczestniczył cały zespół, łącznie z testerami ręcznymi, analitykiem biznesowym i... menadżerem! Nie

wszyscy na co dzień pracowali w środowisku programistycznym, ale to nie jest istotne. Istotne jest stworzenie środowiska do nauki, w którym osoby zarówno mniej, jak i bardziej doświadczone mogą uczestniczyć we wspólnym procesie tworzenia oprogramowania. A jeżeli zespół nie praktykuje *coding dojo*, zachęcam do podjęcia tematu i zaproponowania takich sesji (jedna godzina tygodniowo powinna być łatwa do wygospodarowania).

Kontekst jest królem

Context is king, w języku polskim „kontekst jest królem”, to znane powiedzenie, które bardzo lubię. Wskazuje ono, jak bardzo istotne jest zrozumienie kontekstu danego pojęcia. Test-Driven Development to dziedzina, w której pojęcia nie są unormowane i bywają używane zamiennie, w zależności od kontekstu. Doskonale obrazuje ten fakt tabela 10.3 umieszczona w podrozdziale „Klasyfikacja atrap” rozdziału 10. W jednym kontekście (książki, aplikacji, źródła internetowego) to samo pojęcie może mieć inne znaczenie niż w innym kontekście. Jest to źródłem wielu problemów ze zrozumieniem pewnych schematów. A, paradoksalnie, Test-Driven Development ma przecież służyć rozwiązywaniu problemów z niejednoznacznością wymagań biznesowych i zawartych w nich pojęć.

Poza nomenklaturą związaną z klasyfikacją atrap istnieje jeszcze kilka pojęć, które chciałbym omówić, aby ich rozumienie było całkowicie jednoznaczne.

Atrapa to mock czy test double?

W niniejszej książce atrapa to *test double*, czyli zamiennik zależności, którą chcemy imitować. Do grupy atrap należą wszystkie typy *test double*, czyli *dummy*, *stub*, *fake*, *mock*, *spy* itd. (niektóre źródła czy biblioteki mogą zawierać jeszcze inne typy atrap).

Należy mieć to na uwadze z dwóch powodów.

Po pierwsze, biblioteka do tworzenia atrap, którą opisuję w tej książce, to Moq. W Moqu nie ma rozróżnienia na typy atrap, a wszystko, co stworzymy za pomocą tej biblioteki, jest mockiem. Zatem w kontekście tej biblioteki *mock* to atrapa, a atrapa to *mock*. W pracy nad atrapami w polskojęzycznych zespołach rzadko stosuje się polskie słowo „atrapa” czy angielskie wyrażenie *test double*, dlatego używam spolszczenia „mockować”.

Po drugie, polska Wikipedia tłumaczy hasło *mock object* z angielskiej wersji encyklopedii jako „atrapa obiektu”. Hasło *test double* nie ma natomiast odpowiednika w polskojęzycznej Wikipedii. W literaturze polskiej *test double* zwykle nie ma tłumaczenia, a pojęcie to występuje na przykład jako *obiekt test double* [Martin, 2010]. Tłumaczenia pojęć *mock* i *test double* w kontekście niniejszej książki przedstawia tabela W.1.

TABELA W.1. *Tłumaczenia pojęć mock i test double w kontekście niniejszej książki*

Pojęcie (j. ang.)	Kontekst	Tłumaczenie na j. polski
<i>test double</i>	niniejsza książka w kontekście ogólnym	atrapa
<i>mock</i>	niniejsza książka w kontekście biblioteki Moq	atrapa, mock
	jeden z typów <i>test double</i>	imitacja

xUnit a xUnit.net

xUnit to rodzina bibliotek do tworzenia testów jednostkowych. Powstały one na bazie opracowanego w 1998 roku przez Kenta Becka frameworku SUnit dla języka Smalltalk.

xUnit.net to jedna z bibliotek do tworzenia testów jednostkowych dla frameworku .NET.

Funkcja a funkcjonalność oraz funkcyjny a funkcjonalny

W języku polskim słowo „funkcjonalność” bywa niekiedy używane jako synonim słowa „funkcja”. Oto co można przeczytać na ten temat w Poradni Językowej PWN:

„(...) rozróżnienie na funkcję i funkcjonalność w języku specjalistycznym nie wzięło się znikąd — w świecie IT czym innym jest funkcja, a czym innym funkcjonalność. Jednak zwykłym ludziom, którzy korzystają z karty kredytowej, Facebooka czy smartfona po to, by ułatwić sobie życie, odróżnienie na funkcję i funkcjonalność zapewne nie jest potrzebne. Możliwość robienia zdjęć telefonem komórkowym jest dla mnie funkcją tego telefonu, mimo że jego sprzedawca próbował mnie przekonać, że to jego funkcjonalność. Dla informatyka funkcja jest terminem, który znaczy trochę coś innego niż dla osoby, która nie zna tej terminologii. Zawsze na styku takich użyć (specjalistyczne — niespecjalistyczne) dojdzie do pewnych nieścisłości (np. potocznie używa się nazwy depresja na oznaczenie bardzo złego nastroju, a dla lekarza jest to choroba mająca określone cechy) i nic na to nie poradzimy”. [Kłosińska, 2017]

W niniejszej książce zastosowałem się do zacytowanej rady, a zatem nie mówimy o funkcjonalnościach aplikacji, lecz o jej funkcjach. Funkcja ma też znaczenie programistyczne w kontekście podprogramu.

Istnieją również pojęcia „język funkcyjny” i „test funkcjonalny”. Ich definicje zamieściłem w tabeli W.2.

TABELA W.2. Definicje pojęć zastosowanych w książce związanych z funkcją, funkcyjnością i funkcjonalnością

Pojęcie	Kontekst
funkcja	1) zadanie, które spełnia dana rzecz (mylone z funkcjonalnością danej rzeczy) 2) wydzielona część programu, podprogram
język funkcyjny	odmiana programowania deklaratywnego
test funkcjonalny	test oparty na elementach, które są bezpośrednio związane z funkcją aplikacji

Synonimy

Jeśli chodzi o pozostałe kwestie językowe, poniższe pojęcia są w niniejszej książce stosowane wymiennie, chociaż nie zawsze są synonimami:

- biblioteka, framework;
- program, aplikacja, projekt, system;
- API, interfejs programowania aplikacji;
- implementacja, logika biznesowa;
- test spełniony, test green, test zielony itp.;
- test niespełniony, test red, test czerwony itp.

Jak uczyć się metody Test-Driven Development?

TDD jest metodą, która wymaga nauki zarówno w kwestiach teoretycznych, jak i praktycznych. Owszem, sam cykl jest łatwy do zapamiętania i na rozmowie o pracę szybko go powtórzymy: najpierw test, później kod, a na koniec refaktoryzacja! Proste, prawda? Aby nauczyć się tej metody lub polepszyć swoje umiejętności w tym zakresie, należy przede wszystkim sporo ćwiczyć. Książka zawiera sporo przykładów praktycznych, których zrozumienie może okazać się trudne przez samo czytanie. Dlatego zachęcam do wykonania analogicznych kroków, eksperymentowania i zabawy z kodem, który jest opisany w danym dziale. Wymaga to więcej czasu, ale sprawia, że proces uczenia się jest bardziej efektywny. Czy należy przepisać każdą linijkę kodu zawartą w książce? Nie, bynajmniej! Najlepiej, rozpoczynając lekturę książki, stworzyć własną solucję, która będzie swoistą piaszownicą, miejscem służącym do uruchamiania testów i eksperymentowania z kodem.

Co można zrobić po przeczytaniu książki, aby lepiej poznać tematykę? Jeśli zaczniesz stosować TDD w praktyce, Twoja wydajność może z początku być niższa. Jest to naturalne w procesach nauki, i to nie tylko tych powiązanych z programowaniem. Z czasem Twoja wydajność będzie wzrastać, a korzyści płynące ze stosowania TDD, takie jak lepsza jakość kodu i mniejsza ilość defektów, staną się widoczne.

Dobrze jest szukać miejsca do zastosowania TDD w swoim miejscu pracy, jeśli metoda ta nie jest jeszcze praktykowana (należy jednak przedyskutować taką decyzję z zespołem i przełożonym). Nic nie stoi na przeszkodzie, aby zastosować TDD również do swoich własnych projektów. A jeżeli nie mamy własnego projektu, to możemy taki wymyślić, stworzyć i go opublikować lub po prostu wyrzucić. Nie musi to być skomplikowany projekt, wystarczy mały zestaw pomocniczych metod. Chodzi o to, aby metodykę TDD praktykować po przeczytaniu tej książki.

Warto też zastanowić się nad stworzeniem miejsca do nauki i rozwoju w swoim miejscu pracy. I nie mówię tutaj tylko o TDD. Wspólne dyskusje, prezentacje, ćwiczenia (np. *coding dojo*, *kata*) związane z programowaniem, testowaniem aplikacji i dziedzinami pochodnymi co najmniej raz w tygodniu przez godzinę mogą przynieść sporo korzyści obecnemu projektowi. To na tego typu spotkaniach mogą zrodzić się pomysły na udoskonalenie aplikacji w zakresie jej testowania, automatyzacji, jakości itd. Nie musi to też być kontekst projektu, nad którym się obecnie pracuje. Nauka TDD, ATDD, BDD, programowania funkcyjnego itp. może również przynieść wymierne efekty, jeśli wartość biznesowa tej nauki zostanie przeniesiona na grunt projektu. Wymiar nauki ma także aspekt rozwojowy. Dziedzina programowania jest skomplikowana i dynamiczna, a wspólna nauka może podnieść kwalifikacje zespołowe. Dlatego gorąco zachęcam do jej praktykowania (pamiętać należy o akceptacji ze strony przełożonego i zaproszeniu go do udziału).

Nie należy również zapominać o takich praktykach jak inspekcje kodu i programowanie w parach, pozwalają one bowiem podnieść poziom osób uczących się różnych zagadnień, w tym również metody TDD.

Dogmat

Głównym tematem książki jest TDD, ale czy to oznacza, że zawsze należy korzystać z tej metodyki? Odpowiedź brzmi: nie!

Co w przypadku, gdy ktoś preferuje pisanie testów jednostkowych po napisaniu implementacji (odwrotnie niż w TDD), ale uzyskuje w efekcie kod wysokiej jakości i bardzo dobre pokrycie testami? A co, jeśli projekt jest pokryty w 100% testami akceptacyjnymi i jednocześnie ma wysokiej jakości kod? Czy wówczas powinno się narzucić w projekcie pisanie testów jednostkowych i stosowanie TDD?

Należy pamiętać, że stosowanie TDD i pokrycie testami jednostkowymi stanowią drogę do celu, jakim są wysoka jakość kodu, brak obaw przed wprowadzeniem zmiany do systemu oraz niska liczba defektów. Cel ten można realizować za pomocą innych strategii, takich jak Test-After Development lub Behavior-Driven Development. Dobre pokrycie testami jednostkowymi jest jednym ze sposobów (obecnie najpopularniejszym) na dotarcie do tego celu, nie zaś celem samym w sobie.

Wiele kwestii z zakresu metody Test-Driven Development charakteryzuje się dowolnością, dlatego decyzja o ich zastosowaniu należy do programisty lub zespołu. Przykładowo: czy powinno się tworzyć architekturę aplikacji w taki sposób, aby kod był testowalny? W tej książce napisałem, że nie powinno się testować niepublicznych składowych klasy (poza kilkoma wyjątkami). Można zadać kilka pytań w tej materii:

- Czy kod powinien mieć publiczne API po to tylko, aby był testowalny?
- Czy może powinno się tworzyć takie testy, które za pomocą już dostępnego publicznego API pośrednio przetestują niepubliczne?
- A może skorzystać należy z biblioteki umożliwiającej testowanie prywatnych składowych?

Zasada testowania tylko publicznego API kieruje nas na tory konkretnego rozwiązania. Czy jest to najlepsze rozwiązanie? Odpowiedź na to pytanie pozostawiam Czytelnikowi.

Dlaczego o tym piszę? Każdy dobry programista powinien podawać w wątpliwość każdą regułę, każdą zasadę, każdy wzorec i każdą metodykę, jakie napotka na swojej drodze. Spojrzenie dogmatyczne na sprawy związane z programowaniem niesie ze sobą dwa niebezpieczeństwa. Po pierwsze, kontekst jest królem, więc to, co sprawdziło się w jednym projekcie, niekoniecznie ma sens w innym. Po drugie, programowanie to dziedzina bardzo złożona. Aby mogła się ona rozwijać, należy nieustannie podważać istniejące reguły gry i rozważać tworzenie nowych. Wymaga to sporego doświadczenia i kreatywności, ale jednocześnie jest motorem rozwoju dziedziny oprogramowania.

Informacje zawarte w tej książce to wynikowa moich osobistych i zawodowych doświadczeń i informacji zebranych z różnych źródeł, w tym innych książek, prac naukowych i najpopularniejszych odpowiedzi na Stack Exchange i Stack Overflow. TDD i zawarte w tej książce pomysły nie stanowią dogmatu, lecz jedną z optymalnych i najbardziej popularnych ścieżek, której celem jest wytwarzanie wysmienitego oprogramowania!

Narzędzia użyte w tej książce

Narzędzia wykorzystane w książce są zgodne z moimi i — jak wynika z moich obserwacji — uniwersalnymi preferencjami. Nie znaczy to jednak, że musisz korzystać z tych samych.

Środowisko programistyczne użyte do napisania książki to Visual Studio Community 2017. Jest ono darmowe dla użytku własnego i komercyjnego w zakresie indywidualnym. Alternatywą mogą być: JetBrains Rider (wersja płatna, 30-dniowa wersja próbna), Visual Studio Code (wersja darmowa) i MonoDevelop (wersja darmowa, środowisko wieloplatformowe).

Testy automatyczne były uruchamiane za pomocą JetBrains ReSharper (wersja płatna, 30-dniowa wersja próbna) i NUnit 3 Test Adapter (darmowy dodatek do Visual Studio).

Zamiast nich można wykorzystać oficjalne środowisko uruchomieniowe bazujące na konsoli — NUnit Console Runner (wersja darmowa).

Biblioteka użyta do testowania to NUnit. Alternatywą w C# są xUnit.net i MSTest. Niektóre funkcje dostępne w NUnicie mogą być jednak niedostępne w tych dwóch bibliotekach.

Biblioteka do tworzenia atrap to Moq. Alternatywa w C# to FakeItEasy i NSubstitute. Należy wiedzieć, że składnie tych dwóch bibliotek znacząco różnią się od siebie i od składni Moqa.

Biblioteka do mierzenia pokrycia kodu testami to JetBrains dotCover (wersja płatna, 30-dniowa wersja próbna). Visual Studio w wersji Enterprise zawiera wbudowane narzędzie do mierzenia pokrycia kodu testami. Alternatywa to NCover (wersja płatna) i AxoCover (wersja darmowa).

Narzędzie użyte do refaktoryzacji kodu to JetBrains ReSharper. Zamiast niego można wykorzystać funkcje wbudowane w Visual Studio.

Narzędzia firmy JetBrains, a mianowicie ReSharper i dotCover, są płatne. Bezpłatna licencja na produkty JetBrains jest dostępna dla studentów i nauczycieli.

Kod źródłowy do książki

Kod źródłowy do książki jest oparty na otwartym źródle i można pobrać go z GitHuba (<https://github.com/dariusz-wozniak/TddBook-Code>) lub ze strony wydawnictwa Helion (<https://helion.pl/pobierz-przyklady/tddppr/>).

Wersje zależności są następujące:

- .NET Framework: 4.7.1
- NUnit: 3.10.1
- Moq: 4.8.2
- SpecFlow: 2.3.1
- FluentAssertions: 5.2.0
- ExcelDataReader: 3.4.0

Zależne biblioteki zostały dodane do projektu przez menadżer pakietów NuGet.

Skorowidz

A

- AAA, *Patrz*: zasada AAA
- AAT, 78
- Acceptance Test-Driven Development, *Patrz*: ATDD
- Agile Acceptance Testing, *Patrz*: AAT
- algorytm jenny, 164
- all-pairs testing, *Patrz*: testowanie par
- API, 25
- aplikacja, 25
 - sieciowa, 274
 - składowe publiczne, 25
 - ustawienia regionalne, 185, 186, 187
 - język turecki, 187
- Arrange-Act-Assert, *Patrz*: zasada AAA
- asercja, 60, 61, 116, 251, 255
 - komunikat, 258
 - logiczna, 64, 65, 85
 - model
 - fluent builder, 119
 - klasyczny, 118, 119, 130, 133, 134, 137
 - oparty na twierdzeniach, 118, 119, 124, 130, 133, 134, 135, 137
 - należy do zakresu, 128
 - nierozstrzygnięta, 117
 - niestandardowa, 303
 - porównanie, 120, 126
 - typów, 121, 122
 - pośrednia, 62, 63
 - tworzenie, 118
 - złożenie, 128
- assumption, *Patrz*: założenie
- ATDD, 73, 78
- atrapa, 23, 67, 68, 85, 193, 194, 249, 293
 - argument matchers, 210, 215, 216
 - dummy, 230, 232
 - fake, 233, 235
 - historia wywołań, 216, 217
 - klasy, 224
 - klasyfikacja, 230
 - metoda z parametrami, 210
 - mock, 235
 - poziom dostępności
 - internal, 227, 229
 - protected, 227, 228
 - rekursywna, 205
 - składnia
 - deklaratywna, 202, 204, 205
 - imperatywna, 202, 203, 205
 - spy, 235
 - stub, 232, 233
 - tryb zachowania
 - luźny, 209
 - restrykcyjny, 209, 231
 - właściwości, 206, 207
 - tworzenie, 273
 - automatyczne, 201
 - ręczne, 195, 196, 198, 201
 - weryfikacja
 - dostępu, 219
 - wywołań, 217, 220
 - zachowań, 220, 221
 - zapisu właściwości, 219
 - wyrzucająca wyjątek, 226
 - z rezultatem sekwencyjnym, 208
- atrybut
 - Apartment, 180
 - Author, 182
 - Category, 180, 181
 - Combinatorial, 161
 - Culture, 186, 187
 - Datapoint, 166

atrybut

DatapointSource, 166, 168
Description, 182
dziedziczenie, 148
ExpectedException, 135, 253
Explicit, 183, 184
Ignore, 183
InternalsVisibleTo, 229
MaxTime, 189
NonParallelizable, 178
OneTimeSetUp, 147, 148
OneTimeTearDown, 147, 148
Order, 185
Pairwise, 161, 164
Parallelizable, 178
parametry, 107
Platform, 190, 191
Property, 182
Random, 154
Range, 154
Repeat, 188
RequiresThread, 180
Retry, 188
Sequential, 161, 162
SetCulture, 186
SetUp, 147, 148, 257
SetUpFixture, 148
SingleThreaded, 180
TearDown, 147, 148, 257
Test, 84, 97
TestCase, 97, 107, 151, 176, 184
TestCaseData, 184
TestCaseSource, 97, 155, 156, 157, 160
TestFixture, 170
Theory, 166, 170
Timeout, 189
Values, 152, 153
ValueSource, 160
Ayende Rahien, 201

B

Basili Victor, 30
BDD, 78
Beck Kent, 24, 29, 115
Behavior-Driven Development, *Patrz:* BDD

biblioteka, 25

Castle DynamicProxy, 202
csUnit, 291
do testowania, 50
do tworzenia atrapy, 50, 241, 242
dotCover, 297
Expecto, 291
FakeItEasy, 28, 201, 202, 293
Fluent Assertions, 119
Fock, 293
Foq, 293
FsUnit, 72, 291
Fuchu, 291
innego dostawcy, 68
JetBrains dotCover, 28
JUnit, 21, 115
JustMock, 293
MbUnit, 291, 304
Microsoft Fakes, 294
Microsoft Moles, 294
Moq, 20, 23, 28, 201, 294
 historia, 201
 LINQ to Mocks, 204
 ograniczenia, 237
 składnia, 202, 203, 204, 205
MSTest, 28, 91, 116, 291
NCover, 297
NCrunch, 297
NDepend, 297
NMock3, 294
NSubstitute, 28, 201, 202, 294
NUnit, 20, 28, 66, 84, 91, 115, 142, 291, 299
 atrybut, 305, 307
 dodawanie, 93, 94
 historia, 115, 116
 integracja z Visual Studio, 98
 porównanie typów, 129
 ReSharper, *Patrz:* ReSharper
 rozszerzalność, 303, 304
 Test Adapter, 98
 złożenie, 128
NUnit.Mocks, 294
NUnit3TestAdapter, 99
OpenCover, 298
PartCover, 298
podpisana cyfrowo, 229
Rhino Mocks, 201, 295
 typ atrapy, 202

Semantic Designs C# Test Coverage Tool, 298
 Shouldly, 119
 Software Verify .NET Coverage Validator, 298
 SpecFlow, 79
 Squish Coco, 298
 SUnit, 115
 TddBook.Tests.Unit, 229
 TestMatrix, 298
 testów jednostkowych, 91
 Typemock Isolator, 295
 Typemock Isolator Coverage, 298
 unconstrained, 273
 Unquote, 292
 uzależnienie od dostawcy, 149
 Visual Studio, 298
 xUnit, 115
 xUnit.net, 28, 91, 292
 black-box test, *Patrz*: test skrzynki czarnej
 blok try-catch, 132
 błąd runtime, 112
 boxed, *Patrz*: typ zapudełkowany
 brownfield, 261

C

Cazzulino Daniel „kzu”, 201
 CI, *Patrz*: ciągła integracja
 ciągła dostawa, 288
 ciągła integracja, 50, 285, 288
 raport, 287
 serwer, 286
 ciągle wdrażanie, 288, 289
 coding dojo, 22, 49
 constraint, *Patrz*: twierdzenie
 continuous delivery, *Patrz*: ciągła dostawa
 continuous deployment, *Patrz*: ciągle wdrażanie
 continuous integration, *Patrz*: ciągła integracja
 Craig Philip, 115
 cykl
 Red-Green-Refactor, 32, 78, 96, 170
 green, 102
 odwrócenie, 57, 58
 red, 95, 96, 175
 refaktoryzacja, 103, 113
 czas, 186

D

dane
 wrażliwe, 252
 współdzielone, 148, 150, 179, 257
 źródło, 167
 DAO, 68
 data, 186
 Data Access Object, *Patrz*: DAO
 debugowanie, 149
 definicja ukończenia, *Patrz*: DOD
 Definition of Done, *Patrz*: DOD
 delegat, 133
 dependency injection, *Patrz*: wstrzykiwanie zależności
 Dijkstra Edsger, 30
 dług techniczny, 52
 redukcja, 263
 DOD, 53, 54
 dojo, *Patrz*: coding dojo
 dummy, 23

E

EDD, 78
 Example-Driven Development, *Patrz*: EDD

F

fake, 23
 folder, 141, 142
 framework, *Patrz*: biblioteka
 funkcja, 24, 78
 funkcjonalność, 24

G

Generic Test Fixture, *Patrz*: atrybut TestFixture
 gray-box test, *Patrz*: test skrzynki szarej
 greenfield, 261

H

hasło, 252

I

implementacja, 25
 tworzenie, 96
 instrukcja warunkowa, 253
 interfejs
 IConfigurationWrapper, 249
 IEnumerable, 157
 IList, 171
 IResolveConstraint, 118
 IShape, 174
 ITestCaseData, 158
 programowania aplikacji, 25
 symulacja, 196
 użytkownika, 274
 InvariantCulture, *Patrz:* kultura niezmienna
 inversion of control, *Patrz:* odwrócenie sterowania
 isolation framework, *Patrz:* atropa

J

JetBrains dotCover, 278
 język
 funkcyjny, 24
 Gherkin, 78
 programowania, 20
 C#, 20
 Smalltalk, 24, 115

K

kaikaku, 17
 kaizen, 17, 18, 19
 kata, 50
 klasa
 Assert, 66, 84, 117, 118
 Assume, 166
 CollectionAssert, 138
 generyczna, 170, 172
 It, 210
 niezmienna, 174
 o prywatnym poziomie dostępności, 237
 statyczna, 237, 238
 StringAssert, 137
 System.Random, 154
 TestCaseData, 158, 159
 testowa, 87, 97

załączek, 92
 zależności, 195
 zamknięta, 237
 kod, 19
 asynchroniczny, 174
 czytelność, 149
 integracja, 67, 285
 ISO, 187
 istniejący, 261, 262
 dodawanie testów, 271
 jakość, 51, 52
 niedeterministyczny, 68
 powtarzanie, 245
 tworzenie przyrostowe, 106
 zarządzalność, 149
 zduplikowany, 274

kolekcja, 63, 138
 niezmienna, 233
 porównywanie, 138, 139, 140, 141
 komunikat, 142
 asercji, *Patrz:* asercja komunikat
 błędu, 221
 tworzenie, 144, 145
 nazwa testu, 146, 147
 wyjątku, *Patrz:* wyjątek komunikat
 kultura niezmienna, 187

L

Larman Craig, 30
 legacy code, 261
 Liskov Substitution Principle, *Patrz:* zasada podstawienia Liskov
 lista kontrolna, 52, 53
 logika biznesowa, *Patrz:* implementacja
 LSP, *Patrz:* zasada podstawienia Liskov

Ł

łańcuch tekstowy, 143

M

menadżer pakietów NuGet, 94, 95, 99
 metoda
 anonimowa, 133
 AreEqual, 118, 120
 AreNotEqual, 118
 Assert.Fail, 117, 132

Assert.Ignore, 117
Assert.Inconclusive, 117
Assert.Pass, 117, 132
async, 175
asynchroniczna, 176
Callback, 222, 223, 224
Catch, 134
Convert.ChangeType, 151
InRange, 128
IsEmpty, 118
Mock.Of, 204, 205
Multiple, 66
nazwa, 245
niewirtualna, 237
parametr, 152
pomocnicza, 117, 126
protected, 228
Returns, 203
SetReturnsDefault, 207
Setup, 203
SetupAllProperties, 207
SetupProperty, 206
SetupSequence, 208
sygnatura, 96, 194, 250
testowa, 107
That, 118, 166
Throws, 118, 134, 135, 226
ToUpperInvariant, 187
Verify, 216
VerifyGet, 216, 219
VerifySet, 216, 219
void, 136
wirtualna, 225
Microsoft Test Framework, *Patrz*: biblioteka
MSTest
mock, 23, 201
model zmian, 17, 19
modyfikator
 async, 175, 177
 await, 177
 IgnoreCase, 138
 Using, 125, 127

N

namespace, *Patrz*: przestrzeń nazw
NaN, 167
New Constraint, 171
Newkirk Jim, 115, 116
null, 86, 200

O

odwrócenie sterowania, 195
over-specified, *Patrz*: test jednostkowy
przespecyfikowany

P

pairwise testing, *Patrz*: testowanie par
pętla, 253
piramida testów, 74
platforma, 190, 191
plik, 141, 142
 konfiguracji, 249
 zewnętrzny, 160
Poole Charlie, 115, 116
program, *Patrz*: aplikacja
programowanie
 deklaratywne, 25
 ekstremalne, 29, 30
 obiektowe, 50, 68
projekt, *Patrz*: aplikacja
property behavior, *Patrz*: atrapa tryb
zachowania właściwości
prototyp, 68
Prouse Rob, 116
przestrzeń nazw, 97

R

refaktoryzacja, 19, 28, 33, 49
 bezpieczna, 261, 263, 264
 ekstrakcja do osobnej klasy, 266
 ekstrakcja do osobnej metody, 266, 267
 izolowanie zmienianego kodu, 264, 266
 istniejącego kodu, 261, 263
 niebezpieczna, 261, 263
reguła, *Patrz*: zasada
ReSharper, 101, 160, 264, 266
Rider, 264

S

Satir Virginia model zmian, 17, 19
 SBE, 78
 SBESpecification by Example, 78
 scenariusz, 78
 Scrum, 17, 18
 Seneka Starszy, 31
 serwer ciągłej integracji, *Patrz:* ciągła integracja serwer
 integracja serwer
 silnik testowy, *Patrz:* TestEngine
 składowa
 bazowa, 225
 CallBase, 225
 niepubliczna, 67
 publiczna, 252
 słowo kluczowe
 async, 174
 await, 174
 solucja, 92
 specyfikacja, 79
 spy, 23
 stack trace, *Patrz:* ślad stosu
 story testing, 78
 stos ślad, *Patrz:* ślad stosu
 stub, 23
 stubbing, *Patrz:* atrapa tryb zachowania
 właściwości
 system, *Patrz też:* aplikacja
 ciągłej integracji, *Patrz:* ciągła integracja
 operacyjny, 190, 191
 plików, 141, 142, 249
 zarządzania pakietów, *Patrz:* menadżer pakietów

Ś

ślad stosu, 142

T

TAD, *Patrz:* Test-After Development
 TDD, 19, 25, 26, 29, 78
 dla aplikacji
 dużej, 70
 małej, 69
 filozofia, 57

historia, 29
 mantra, *Patrz:* cykl Red-Green-Refactor
 nauka, 49, 50
 opłacalność, 51
 wdrażanie, 49, 51
 zasada, 59
 teoria, 166, 167, 169
 test, 19
 akceptacyjny, 67, 68, 71, 73, 78, 177, 193, 262, 273, 283
 narzędzia, 274
 automatyczny, 77, 78
 narzędzia, 50
 bezpieczeństwa, 81
 czas wykonania, 189
 czerwony, *Patrz:* test niespełniony
 double, 23, 230
 dymny, 80
 fałszywie pozytywny, 246, 250, 262
 funkcjonalny, 24
 green, *Patrz:* test spełniony
 integracyjny, 67, 68, 73, 78, 85, 177, 193
 dla zależności wewnętrznych, 273
 konfiguracja, 85
 szybkość, 85
 interakcyjny, 283
 interfejsu użytkownika, 80
 internacjonalizacji i lokalizacji, 81
 jednostkowy, 21, 29, 49, 57, 59, 73, 78, 83, 85
 asercja logiczna, 85
 bez asercji, 281
 debugowanie, 101
 dodawanie do istniejącego kodu, 271
 dzielenie, 64, 66
 interpretowany, 249
 izolowanie, 246, 247
 klasa testowa bezstanowa, 257
 nazwa, 87, 88, 146, 147, 252
 niedyskryminowany, 251
 niepotrzebny, 67
 pisane przez innego programistę, 72
 powtarzalność, 155, 246, 248
 prostota, 255, 271
 przespecyfikowany, 256
 przypadki brzegowe, 87, 96, 252, 282
 struktura, 83
 szybkość, 85, 194, 246

- tolerancja, 108, 109, 124
- tolerancja czasu, 124
- tworzenie, 24, 33, 97, 98, 105, 272
- uruchamianie, 98, 99
- w innym języku programowania, 71
- wartość oczekiwana, 254
- wpływ na pracę, 71
- zależności zewnętrzne, 85, 110, 142, 193, 194, 195, 246, 248, 249
- kolejność, 185
- konfiguracji, 81
- manualny, 70, 77
- nierozstrzygnięty, 117, 166
- niespełniony, 25, 89, 98, 117
 - warunki, 116, 117
- niezmiennika, 173
- niskopoziomowy, 74
- obciążeniowy, 81
- oparty na własnościach, 274
- parametryzowany, 61, 107, 150, 151, 169, 176, 192
 - oparty na zewnętrznych źródłach, 160
- piramida, *Patrz*: piramida testów
- poczytalności, 80
- pokrycie kodu, 28, 31, 50, 277, 278, 282, 297
 - brakujące testy, 284
 - narzędzia, 274
 - narzędzia do mierzenia, 278
 - skala, 281, 282, 283
- powtarzanie, 188
- półautomatyczny, 77
- przeciążeniowy, 81
- przystępności, 81
- pusty, 250
- red, *Patrz*: test niespełniony
- silnik, *Patrz*: TestEngine
- skalowalności, 81
- skrzynki, 77
 - białej, 77
 - czarnej, 77, 110
 - szarej, 78
- spełniony, 25, 98, 117
 - warunki, 116
- spike, 81
- strukturalny, *Patrz*: test skrzynki białej
- teoria, *Patrz*: teoria
- uruchamianie równoległe, 116
- użyteczności, 80
- wydajnościowy, 81
- wysokopoziomowy, 74
- wytrzymałościowy, 81
- z danymi zewnętrznymi, 299
- z zamiennikiem, 173
- zielony, *Patrz*: test spełniony
- zignorowany, 117
- zrównoleglenie, 178, 179
- Test-After Development, 58, 59, 72, 78, *Patrz* też: Test-Last Development
- Test-Driven Development, *Patrz*: TDD
- TestEngine, 116
- tester manualny, *Patrz*: test manualny
- Test-First Development, 58
- Test-Last Development, 57, 58
- testowanie
 - niefunkcjonalne, 80
 - par, 163
 - poziom, 78
 - regresyjne, 80
 - zdarzeń, 111
- TFD, *Patrz*: Test-First Development
- TLD, *Patrz*: Test-Last Development
- twierdzenie, 118, 166
 - Is.Empty, 118
 - Is.EqualTo, 118, 119, 120
 - Is.Not.EqualTo, 118
 - Throws, 135
 - Throws.Exception, 118
 - With.Message, 136
 - With.Parameter, 136
- Two Michael, 115
- typ
 - DateTime, 186
 - decimal, 170
 - double, 170
 - float, 170
 - generyczny, 171
 - int, 170
 - konwersja, 151
 - long, 170
 - ParallelScope, 178
 - porównanie, 129
 - tekstowy, 137, 138
 - zapudełkowany, 122

V

vendor lock-in, *Patrz:* biblioteka uzależnienie od dostawcy
Visual Studio, 264
Visual Studio Enterprise, 28
Voronstov Alexei, 115

W

wartość null, *Patrz:* null
white-box test, *Patrz:* test skrzynki białej
Wilson Brad, 116
właściwość
 Count, 171
 ExpectedResult, 152
 InnerException, 137
 ParamName, 136
 wirtualna, 225
wstrzykiwanie zależności, 195
wyjątek
 ArgumentNullException, 200
 ArithmeticException, 253
 AssertionException, 98
 DivideByZeroException, 97, 109, 253
 hierarchia dziedziczenia, 253
 InnerException, 137
 komunikat, 136
 MockException, 209
 NotImplementedException, 96, 198, 250
 NotSupportedException, 225
 NullReferenceException, 206
 parametr, 136
 przechwytywanie, 253
 SystemException, 253
 testowanie, 131, 132, 133, 135
 typ, 134
 wewnętrzny testowanie, 137

wyrażenie lambda, 134, 176, 201
wywołanie zwrotne, 222
wzorzec

 Arrange-Act-Assert, *Patrz:* zasada AAA
 Setup-Returns, 203, 205

Z

założenie, 166, 169
zasada
 AAA, 83, 84, 251
 AAAA, 84
 DRY, 245
 FIRST, 246
 Guard Assertion, 84
 GWT, 84
 podstawienia Liskov, 172, 173, 174
 Record & Play, 201
 Record-Playback, 84
 SOLID, 68, 172, 272
zdarzenie
 Calculated, 111, 112
 OnCalculated, 111, 113
zmienna
 GlobalSettings.DefaultFloating
 ↪PointTolerance, 124
wejściowa, 60
wyjściowa, 60, 61

PROGRAM PARTNERSKI

GRUPY WYDAWNICZEJ HELION



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

Zmień swoją stronę WWW
w działający bankomat!

Dowiedz się więcej i dołącz już dzisiaj!

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

Odnieś sukces dzięki TDD!

- Poznaj filozofię programowania sterowanego testami
- Dowiedz się, jak wdrożyć metodę TDD w praktyce
- Naucz się stosować właściwe narzędzia i techniki

Metoda test-driven development pozwala na pisanie lepszej jakości, bardziej elastycznego i łatwiejszego w utrzymaniu kodu, na którym można w pełni polegać. Większa wydajność pracy programistów, którzy o wiele lepiej rozumieją potrzeby biznesowe stawiane tworzonemu przez nich aplikacjom, znaczne przyspieszenie powtarzalnych testów — to tylko niektóre zalety TDD. Nic dziwnego, że świat zachwyił się tą techniką, a jej znajomość należy do podstawowych wymagań, które muszą spełnić inżynierowie pragnący rozwijać swoją karierę w branży IT.

Jeśli chcesz poznać metodę TDD i nauczyć się tworzyć zgodne z nią testy jednostkowe, trafiłeś na właściwą książkę! W prosty sposób przedstawi Ci ona cykl red-green-refactor, zaprezentuje zalety poprawnie zaimplementowanej techniki TDD, zwróci uwagę na trudności związane z jej wdrażaniem i podpowie, jak sobie z nimi poradzić. Nauczysz się pisać testy jednostkowe zgodnie z dobrymi praktykami oraz sprawdzać zależności i tworzyć atrapy obiektów. Dowiesz się, jak stosować TDD w przypadku już istniejącego kodu, a także jak mierzyć pokrycie kodu testami. Znajdziesz tu również podstawowe informacje na temat ciągłej integracji i jej znaczenia dla techniki TDD.

- Podstawowe informacje o testach i metodzie TDD
- Tworzenie dobrych testów jednostkowych
- Praktyczne zastosowanie biblioteki NUnit
- Atrapy i ich klasyfikacja
- Zastosowanie TDD do istniejącego kodu
- Pokrycie kodu testami i ciągła integracja
- Najważniejsze biblioteki wspierające TDD

Poznaj w praktyce najbardziej przebojową metodę tworzenia oprogramowania!

Helion 



helion.pl



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

Sprawdź nasze szkolenia!



AKADEMIA IT & BUSINESS

WWW.SZKOLENIA.HELION.PL

KOD KORZYŚCI
Sięgnij po więcej! ▶



ISBN 978-83-283-3531-8



9 788328 335318

INFORMATYKA W NAJLEPSZYM WYDANIU

Cena: 59,00 zł