

John Callaway, Clayton Hunt

TDD z wykorzystaniem C# 7

Programowanie
sterowane testami

Tytuł oryginału: Practical Test-Driven Development using C# 7: Unleash the power of TDD by implementing real world examples under .NET environment and JavaScript

Tłumaczenie: Jakub Hubisz

ISBN: 978-83-283-5653-5

Copyright © Packt Publishing 2018. First published in the English language under the title ‘Practical Test-Driven Development using C# 7 – (9781788398787)’

Polish edition copyright © 2019 by Helion SA
All rights reserved.

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

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

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

Autor oraz Helion SA 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 Helion SA nie ponoszą również żadnej odpowiedzialności za ewentualne szkody wynikłe z wykorzystania informacji zawartych w książce.

Helion SA
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/tddwyk>
Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

Printed in Poland.

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

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

Spis treści

Przedmowa	9
O autorach	11
O korektorze merytorycznym	12
Wprowadzenie	13
Rozdział 1. Dlaczego TDD jest ważne?	17
Najpierw trochę o nas	18
Historia Johna	18
Historia Claytona	18
Czym jest TDD?	19
Podejście do TDD	19
Podejście alternatywne	20
Proces	20
Po co zwracać sobie tym głowę?	21
Argumenty przeciwko TDD	21
Testowanie wymaga czasu	21
Testowanie jest kosztowne	22
Testowanie jest trudne	22
Nie wiemy jak	22
Argumenty za TDD	23
Mniejsza pracochłonność testowania manualnego	23
Mniej błędów	23
Pewien poziom poprawności	23
Brak strachu przed refaktoryzacją	24
Lepsza architektura	24
Szybsza praca	24
Różne rodzaje testów	25
Testy jednostkowe	25
Testy akceptacyjne	25

Testy integracyjne	25
Testy typu end-to-end	26
Liczba testów poszczególnych rodzajów	26
Części testu jednostkowego	26
Aranżacja	26
Akcja	26
Asercja	27
Wymagania	27
Dlaczego wymagania są ważne?	27
Historie użytkownika	27
Gherkin	29
Nasze pierwsze testy w C#	31
Rozwijanie aplikacji z testami	33
Nasze pierwsze testy w JavaScriptcie	34
Dlaczego to ma znaczenie?	37
Podsumowanie	37
Rozdział 2. Przygotowanie środowiska testowego w .NET	39
Instalacja SDK .NET Core	39
Przygotowanie VS Code	40
Tworzenie projektu w VS Code	44
Przygotowanie Visual Studio Community	45
Pobieranie Visual Studio Community	46
Instalacja Visual Studio Community	46
Przesiadka na xUnit	46
Programistyczne kata	47
Stworzenie projektu	47
Czym jest Speaker Meet?	50
Projekt Web API	51
Podsumowanie	55
Rozdział 3. Przygotowanie środowiska testowego w JavaScriptcie	57
Node.js	57
Czym jest Node?	58
Po co nam Node?	58
Instalacja Node	58
NPM	61
Szybkie wprowadzenie do IDE dedykowanych dla JavaScriptu	62
Visual Studio Code	63
WebStorm	64
Create React App	65
Czym jest Create React App?	66
Instalacja modułu globalnego	66
Tworzenie aplikacji za pomocą Reacta	66
Mocha i Chai	67
Szybkie kata sprawdzające środowisko	72
Wymagania	72
Wykonanie	72
Rozpoczęcie kata	73
Podsumowanie	76

Rozdział 4. Co należy wiedzieć przed rozpoczęciem pracy?	77
Nietestowalny kod	78
Wstrzykiwanie zależności	78
Wyodrębnianie oprogramowania zewnętrznego	79
Sobowtóry testowe	79
Frameworki imitujące	80
Zasady SOLID	80
Powitanie zależne od czasu	83
Kruche testy	84
Rodzaje sobowtórów testowych	86
Przykład wielopoziomowy	93
Podsumowanie	100
Rozdział 5. Tabula rasa — podejście do aplikacji na sposób TDD	101
Gdzie zacząć?	101
Golenie jaka	102
Duży projekt od razu	103
Czysta kartka	103
Po jednym kawałku	103
Minimalny wykonalny produkt	104
Inny sposób myślenia	104
Nie będziesz tego potrzebować	104
Małe testy	105
Adwokat diabła	106
Najpierw testy ścieżek negatywnych	109
Kiedy testowanie jest bolesne	113
Symulacja	113
Najpierw asercja	114
Bądź zorganizowany	114
Rozbicie aplikacji Speaker Meet	114
Prelegenci	114
Społeczności	115
Konferencje	115
Wymagania techniczne	115
Podsumowanie	115
Rozdział 6. Podejście do problemu	117
Zdefiniowanie problemu	117
Przetrawienie problemu	118
Epiki, funkcje i historie — ojej!	118
Problem Speaker Meet	120
Architektura heksagonalna wielowarstwowa	126
Architektura heksagonalna	127
Podstawowe, ale wydajne podziały wielowarstwowe	127
Kierunek testowania	130
Od tyłu do przodu	130
Od przodu do tyłu	137
Od wewnątrz na zewnątrz	144
Podsumowanie	148

Rozdział 7. Sterowanie testami aplikacji C#	149
Przegląd wymagań	149
Lista prelegentów	150
API	150
Testy API	151
Usługa	156
Testy usługi	156
Czyste testy	160
Repozytorium	161
Wykorzystanie fabryki z FakeRepository	163
Szczegóły prelegentów	165
API	165
Testy API	165
Usługa	169
Testy usługi	169
Czyste testy	172
Coś więcej z repozytorium	172
Dodatkowa praca związana z fabryką	173
Testowanie przypadków wyjątkowych	174
Podsumowanie	176
Rozdział 8. Wyodrębnianie problemów na zewnątrz	177
Odseparowanie problemów	177
Gravatar	178
Planowanie na przyszłość	185
Abstrahowanie warstwy danych	185
Rozszerzanie wzorca repozytorium	186
Zapewnienie funkcji	191
Tworzenie generycznego repozytorium	210
Krok pierwszy: abstrahowanie interfejsu	210
Krok drugi: abstrahowanie klasy konkretnej	211
Krok trzeci: zmiana testów, aby wykorzystywały repozytorium generyczne	215
Entity Framework	219
Wstrzykiwanie zależności	222
Podsumowanie	223
Rozdział 9. Testowanie aplikacji napisanej w JavaScriptcie	225
Tworzenie aplikacji za pomocą Reacta	226
Wyodrębnienie aplikacji	226
Konfiguracja bibliotek Mocha, Chai, Enzyme i Sinon	226
Plan	228
Komponent React	228
Rzut oka na testowalność Reduxa	229
Testy jednostkowe usługi API	230
Lista prelegentów	230
Imitacja usługi API	231
Akcja pobierania wszystkich prelegentów	235
Reduktor dla pobierania wszystkich prelegentów	239
Komponent listy prelegentów	240

Szczegóły prelegenta	246
Rozbudowa imitacji usługi API	246
Akcja pobierania prelegenta	248
Reduktor dla pobierania prelegenta	254
Komponent dla szczegółów prelegenta	257
Podsumowanie	260
Rozdział 10. Kwestia integracji	261
Implementacja rzeczywistej usługi API	261
Zamiana imitacji API na prawdziwą usługę	262
Wykorzystanie biblioteki Sinon do imitacji odpowiedzi Ajaxa	264
Konfiguracja aplikacji	273
Testy integracyjne od początku do końca	273
Zalety	273
Wady	274
Ile testów end-to-end trzeba mieć?	274
Konfiguracja API	274
Projekt dla testów integracyjnych	275
Gdzie zacząć?	275
Weryfikacja odwołań repozytorium do bazy	275
Weryfikacja, czy usługa odwołuje się do bazy przez repozytorium	278
Weryfikacja odwołań API do usługi	280
Podsumowanie	284
Rozdział 11. Zmiany w wymaganiach	285
Witaj, świecie	286
Zmiana wymagań	286
FizzBuzz	287
Nowa funkcja	287
Aplikacja TODO	289
Oznaczanie jako wykonane	289
Dodanie testów	289
Kod produkcyjny	290
Dodanie testów	290
Kod produkcyjny	292
Zmiany w aplikacji Speaker Meet	293
Zmiany po stronie serwera	293
Zmiany po stronie interfejsu	295
Co teraz?	296
Przedwczesna optymalizacja	296
Podsumowanie	297
Rozdział 12. Problem z kodem zastanym	299
Czym jest kod zastany?	299
Dlaczego kod staje się zły?	300
Kiedy projekt staje się projektem zastanym?	300
Co możemy zrobić, aby zapobiec powstawaniu kodu zastanego?	301
Typowe problemy wynikające z kodu zastanego	302
Niezamierzone skutki uboczne	302
Nadmierna optymalizacja	303

Zbyt sprytny kod	304
Ścisłe łączenie z kodem zewnętrznym	304
Problemy przeszkadzające w dodawaniu testów	305
Bezpośrednia zależność od frameworka lub kodu zewnętrznego	305
Prawo Demeter	306
Praca w konstruktorze	306
Globalny stan	307
Metody statyczne	307
Duże klasy i funkcje	307
Radzenie sobie z problemami wynikającymi z kodu zastanego	308
Bezpieczna refaktoryzacja	308
Pierwsze testy	310
Idąc dalej	312
Naprawa błędów	313
Niebezpieczna refaktoryzacja	313
Podsumowanie	313
Rozdział 13. Sprzątanie bałaganu	315
Dziedziczenie kodu	315
Gra	316
Prośba o zmianę	316
Czasami dostajesz od życia cytryny	317
Zaczynamy	317
Abstrakcja klasy zewnętrznej	320
Niespodziewane dane wsadowe	324
Szukanie sensu w szaleństwie	329
Końcowe upiększanie	335
Gotowy na ulepszenia	337
Podsumowanie	349
Rozdział 14. Pokaż się z najlepszej strony	351
Co omówiliśmy?	351
Idąc naprzód	352
TDD to osobisty wybór	352
Nie potrzebujesz pozwolenia	353
Rozwijaj aplikacje poprzez testy	353
Wprowadzanie TDD do Twojego zespołu	353
Nie zmuszaj nikogo do TDD	354
Grywalizacja TDD	354
Pokaż zespołowi zalety	354
Kontroluj rezultaty	355
Powrót do świata jako ekspert TDD	355
Poszukaj mentora	355
Zostań mentorem	356
Praktykuj, praktykuj, praktykuj	356
Podsumowanie	357
Skorowidz	358

Co należy wiedzieć przed rozpoczęciem pracy?

Całkiem nieźle zacząłeś. Powinieneś już zaczynać czuć się komfortowo z podstawowymi koncepcjami programowania sterowanego testami. Znasz już podstawowe założenia TDD i wiesz, jak pisać testy jednostkowe w C# i JavaScriptcie.

W tym rozdziale:

- Omówimy więcej praktyk związanych z TDD
- Przekażemy porady, jak unikać pułapek czyhających na nowych praktyków TDD
- Objasnimy, dlaczego ważne jest określenie granic testowania, wyodrębniając kod zewnętrzny (włącznie z frameworkiem .NET)
- Zaczniemy wprowadzać bardziej zaawansowane koncepcje, takie jak szpiegi, imitacje i fałszywki

Najpierw omówimy problemy, jakie możesz napotkać, próbując stworzyć testy dla istniejących aplikacji. Mamy nadzieję, że pomoże Ci to uniknąć tego typu problemów podczas tworzenia od podstaw swojej kolejnej aplikacji.

Nietestowalny kod

Istnieje wiele oznak pozwalających przypuszczać, że aplikacja, klasa lub metoda będą trudne lub nawet niemożliwe do przetestowania. Oczywiście istnieją sposoby na obejście niektórych z poniższych przykładów, ale z reguły najlepiej unikać obejść i programistycznej akrobatyki. Proste rozwiązania są zwykle najlepsze, a osoby, które będą musiały utrzymywać Twoją aplikację, podziękują Ci za trzymanie się prostoty (albo sam sobie za to podziękujesz).

Wstrzykiwanie zależności

Jeżeli tworzysz instancje zewnętrznych zasobów wewnątrz konstruktorów lub wewnątrz metod zamiast przekazywać je do nich, bardzo ciężko będzie napisać testy pokrywające te klasy czy metody. We współczesnych aplikacjach do tworzenia i przekazywania do klas zewnętrznych zależności wykorzystywane są frameworki wstrzykiwania zależności. Wiele osób decyduje się na zdefiniowanie *interfejsu* jako kontraktu dla zależności, co zapewnia większą elastyczność na potrzeby testów w łączeniu zewnętrznych zasobów.

Elementy statyczne

Być może będziesz musiał odwoływać się do zewnętrznych statycznych klas i metod. Zamiast bezpośrednio odwoływać się do nich lepiej uzyskiwać do nich dostęp przez *interfejs*. Na przykład właściwość `Now` klasy `DateTime` jest statyczna, co nie pozwala Ci kontrolować wartości `DateTime` używanej przez testowaną klasę czy metodę. To utrudnia weryfikację przypadków testowych i zapewnienie poprawnego zachowania się logiki programu.

Singleton

Singletony stanowią esencję współdzielonego stanu. Aby zapewnić uruchamianie testu w izolowanym środowisku, najlepiej ich unikać. Jeżeli singleton jest wymagany (na przykład na potrzeby logowania czy kontekstu danych), większość frameworków wstrzykiwania zależności umożliwia podmianę klasy nie będącej singletonem na jedną instancję, co w efekcie daje funkcjonalność i elastyczność singletonu. W kodzie produkcyjnym pozwala to kontrolować zakres instancji singletonu.

Stan globalny

Od dawna wiadomo, że globalny stan w aplikacji powoduje ogromne zamieszanie w systemie i nieoczekiwane zachowanie, które może być trudne do wykrycia. Zmiana kodu w jednym miejscu może mieć daleko idące efekty uboczne w innych częściach systemu. Dla testowalności oznacza to często zwiększoną trudność w przygotowywaniu testów i wolniejsze ich wykonywanie.

Wyodrębnianie oprogramowania zewnętrznego

Wraz z rozbudową aplikacji będziesz prawdopodobnie wprowadzać zależności zewnętrzne. Z pewnością twórcy tych systemów, aplikacji i bibliotek dokładnie przetestowali swoje rozwiązania. Powinieneś skupić się na testowaniu swojego kodu, nie kodu zewnętrznego. Twoja aplikacja powinna być na tyle solidna, aby obsłużyć warunki brzegowe, i musisz brać pod uwagę oczekiwane i nieoczekiwane zachowanie. Szczegóły kodu zewnętrznego należy wyodrębnić, aby można było przetestować oczekiwane (i nieoczekiwane) zachowanie.

Czym jest **kod zewnętrzny**? Wszystkim, co nie zostało napisane przez Ciebie. Wlicza się w to także .NET Framework. Jednym ze sposobów wyodrębnienia kodu zewnętrznego są sobowtóry testowe.

Sobowtóry testowe

Sobowtóry testowe to funkcje i klasy pomocnicze w procesie testowania, pomagające zweryfikować funkcję lub ominąć zależność, która mogłaby być trudna do przetestowania. Sobowtóry testowe są używane na wszystkich poziomach do izolowania testowanego kodu. W wielu przypadkach konieczność posiadania sobowtórów testowych ma decydujący wpływ na architekturę kodu.

Przykładem jest obiekt `DateTime` w C#. `System.DateTime` jest częścią frameworka .NET i normalnie nie brałbyś pod uwagę wyodrębniania go z kodu. Instynkt większości programistów podpowiada im, aby dodać referencję za pomocą polecenia `using` i bezpośrednio odwoływać się do niego w kodzie.

Test, który nie może zostać powtórzony, jest złym testem.

Często trudno przetestować taki kod. Jeżeli chcielibyśmy przetestować metodę używającą właściwości `DateTime.Now`, nie bylibyśmy w stanie zapobiec wykonaniu domyślnego zachowania `DateTime.Now`. `DateTime.Now` zwraca aktualną datę i czas w obiekcie `DateTime`. Brak wpływu na to, jaką wartość zwraca obiekt, powoduje, że testy będą nieprzewidywalne i niepowtarzalne. Test, który nie może być powtórzony, jest złym testem.

Wielu programistów rozumie potrzebę przewidywalności. Być może słyszałeś powiedzenie: „Jeżeli nie można tego zreprodukować, nie jest to błąd”. To dlatego, że aby zweryfikować, czy udało nam się poprawić błąd, musimy być w stanie zreprodukować go w sposób przewidywalny. W ten sposób, kiedy kroki prowadzące do wystąpienia błędu już go nie powodują, możemy bezpiecznie stwierdzić, że został poprawiony. Przynajmniej dla tej sekwencji kroków.

Testowanie nie różni się specjalnie od naprawiania błędów; wykonać należy te same kroki. Wiemy po prostu dokładnie, co spowodowało błąd — kod nie został jeszcze napisany lub coś poszło nie tak podczas refaktoringu.

Tworzenie sobowtórów testowych może czasami być mocno angażujące. Z tego powodu w prawie każdym języku wspierającym testowanie stworzono frameworki pomagające w ich tworzeniu. Nazywa się je frameworkami lub bibliotekami imitującymi. Najczęściej stosowany framework w C# to *Moq*. W JavaScriptcie najczęściej pobieraną biblioteką imitującą jest *Sinon*.

Frameworki imitujące

Frameworki imitujące to doskonałe narzędzia pomagające złagodzić nieco trudy testowania w dużym projekcie. Są szczególnie użyteczne podczas prób zbudowania testów dla systemów zastanych. System zastany jest w tym przypadku definiowany jako aplikacja, która nie posiada testów automatycznych. Definicja ta pochodzi z książki Michaela Feathera pod tytułem *Praca z zastanym kodem*.

Musisz zachować czujność podczas nauki TDD i stosowania frameworków imitujących. Frameworki imitujące stanowią bardzo atrakcyjną alternatywę dla szczegółowego analizowania Twojego kodu. Możliwe jest napisanie kompletnego zestawu testów, które w końcowym rozrachunku testują tylko framework imitujący.

Wiele z tych frameworków jest pod tym względem zbyt potężnych. W C# istnieją frameworki imitujące, które pozwalają podmieniać kod zewnętrzny. Włącza się w to `DateTime.Now` i każda inna klasa, której nie kontrolujesz. W JavaScriptcie nazywa się to małym łataniem i każdy framework na to pozwala.

Pytasz, co w tym złego. Jedną z zalet TDD jest to, że zachęca do mądrych wyborów architektonicznych. Kiedy masz możliwość nadpisać funkcje zewnętrznego kodu, tracisz potrzebę wyodrębniania na potrzeby testów.

Dlaczego jest to problemem? Wyodrębnienie kodu zewnętrznego jest niezbędne, jeżeli chcemy, aby kod był elastyczny i jeżeli chcemy przestrzegać zasad SOLID.

Zasady SOLID

Zasady SOLID to zestaw koncepcji zgrupowanych przez Roberta C. Martina, zwanego również wujkiem Bobem. Opisywane są zwykle jako **zasady programowania zorientowanego obiektowo**, ale myśl o nich raczej jak o dobrych wyborach architektonicznych. SOLID składa się z pięciu zasad: zasady jednej odpowiedzialności (ang. *Single Responsibility*), zasady otwarte – zamknięte (ang. *Open/Closed*), zasady podstawienia Liskov (ang. *Liskov Substitution*), zasady segregacji interfejsów (ang. *Interface Segregation*) i zasady odwrócenia zależności (ang. *Dependency Inversion*).

Artykuły przedstawiające zasady SOLID znajdziesz pod adresem <http://butunclebob.com/ArticleS.UncleBob.PrinciplesOfOod>.

Zasada jednej odpowiedzialności

W artykule wujka Boba zasada jednej odpowiedzialności jest zdefiniowana następująco: „Klasa powinna mieć tylko jeden powód do zmiany”.

Co to oznacza? To trudna kwestia i można ją pojmować na wiele sposobów. Jeden z nich mówi, że klasa powinna mieć tylko jednego użytkownika biznesowego. Inny, że w ramach aplikacji klasa powinna być wykorzystywana tylko w ograniczonym lub bardzo konkretnym zakresie. Jeszcze inny, że klasa powinna mieć ograniczoną funkcjonalność. Wszystkie te sposoby są poprawne, jednak wciąż niewystarczające. Jednym ze sposobów zapewnienia przestrzegania reguły jednej odpowiedzialności jest wykorzystanie reguły nazywanej przez nas „trzy do pięciu”.

Jeżeli na przykład omawiamy wymagania, to kiedy wymaganie ma trzy do pięciu kryteriów akceptacji, najprawdopodobniej jest to liczba właściwa dla jego poziomu szczegółowości. Analogicznie w przypadku metody lub funkcji trzy do pięciu linii kodu to prawdopodobnie odpowiedni rozmiar.

Reguła „trzy do pięciu” to ogólny sposób określenia, czy przestrzegasz zasady jednej odpowiedzialności. Reguła mówi: „Mniej niż trzy to dobrze, pomiędzy trzy a pięć jest akceptowalnie, powyżej pięciu należy rozważyć refaktoryzację”. Reguła nie jest może tak elegancka jak wiele innych praw, zasad czy reguł, ale za to łatwo jej przestrzegać. Nie powinna jednak być stosowana tylko w ostateczności. Staraj się stosować ją do prawie wszystkich etapów wytwarzania oprogramowania. Widziałeś ją już zresztą w akcji w tej książce — została wykorzystana do określenia zakresu wymagań w rozdziale 1., „Dlaczego TDD jest ważne?”, oraz we wszystkich dotychczasowych przykładach kodu.

Jeżeli będziesz korzystać z reguły „trzy do pięciu”, mogę niemalże zagwarantować, iż będziesz przestrzegać reguły jednej odpowiedzialności. Dzięki temu również Twój kod, struktura plików i wymagania będą niewielkie i łatwe w utrzymaniu.

Zasada otwarte – zamknięte

Zasada otwarte – zamknięte mówi: „Encje oprogramowania (klasy, moduły, funkcje itd.) powinny być otwarte na rozszerzanie, ale zamknięte na modyfikację”. Druga zasada SOLID zdaje się nie mówić zbyt wiele, ale ma bardzo duży wpływ na strukturę kodu.

Jest wiele sposobów, by przestrzegać tej zasady. Ty lub Twój zespół programistyczny możecie wprowadzić w życie regułę pozwalającą na pisanie wyłącznie nowego kodu. Oznacza to, że żadna z istniejących funkcji programu nie może być aktualizowana czy modyfikowana — możliwe jest jedynie zastąpienie jej nową funkcją. Kiedy dotrzemy w kodzie do miejsca, gdzie następuje podział na starą i nową funkcję, możemy dokonać podmiany.

Zasada otwarte – zamknięte sprzyja również ciągłej integracji i ciągłemu dostarczaniu. Jest tak dlatego, że jeżeli Twoja aplikacja nigdy nie złamie kontraktu z użytkownikiem, samą sobą czy zewnętrznym oprogramowaniem, zawsze można ją umieścić w środowisku produkcyjnym bez obawy, że pojawią się jakieś problemy.

Zasada podstawienia Liskov

Zasada podstawienia Liskov może być trudna do zrozumienia przy pierwszym podejściu, ponieważ jej definicja jest dość skomplikowana i matematyczna. Barbara Liskov w pracy *Data Abstraction and Hierarchy* (<https://pdfs.semanticscholar.org/36be/babeb72287ad9490e1ebab84e7225ad6a9e5.pdf>) opisała zasadę następująco:

Chodzi nam o osiągnięcie podstawienia podobnego do następującego: jeżeli dla każdego obiektu o1 typu S istnieje obiekt o2 typu T taki, że dla wszystkich programów P zdefiniowanych w ramach warunków T zachowanie P pozostaje niezmienione, kiedy o1 zostanie zastąpione przez o2, wtedy S jest podtypem T.

Wujek Bob uprościł definicję w następujący sposób: „Funkcje używające wskaźników lub referencji do klas bazowych muszą być w stanie korzystać z obiektów klas pochodnych, nie mając o nich wiedzy”. Patrząc na tę zasadę, wydaje się, że chodzi tylko o dziedziczenie. Tak jednak nie jest. Zasada ta oznacza, że obiekt zastępujący inny obiekt nie tylko musi implementować ten sam interfejs lub kontrakt co obiekt oryginalny; musi również spełniać te same oczekiwania co oryginal.

Klasycznym przykładem pogwałcenia tej zasady jest użycie klasy kwadrat w miejsce klasy prostokąt. Typowa klasa prostokąt musi mieć właściwości szerokość i wysokość. W matematyce kwadrat jest specyficznym rodzajem prostokąta. Wiele osób zakłada więc, że stworzenie klasy kwadrat z szerokością i wysokością byłoby akceptowalnym substytutem dla klasy prostokąt.

Problem polega na tym, że kwadrat wymaga, aby szerokość i wysokość były identyczne. Jeżeli więc zmienisz jedną z właściwości klasy kwadrat, klasa zaktualizuje drugą tą samą wartością. Jest to problem, ponieważ aplikacja korzystająca z obiektu nie spodziewa się takiego zachowania. Aplikacja musi więc mieć wiedzę, że długość lub wysokość może się zmieniać bez ostrzeżenia.

Niemożliść spełnienia oczekiwań aplikacji nazywa się odrzuconym żądaniem. Odrzucone żądanie może powodować niespójne zachowanie aplikacji i wymaga przynajmniej większej ilości kodu do skompensowania niezgodności.

Zasada segregacji interfejsów

Zasada segregacji interfejsów dotyczy utrzymania niewielkich rozmiarów kontraktów interakcji klasy. Kontrakty powinny być jednak nie tylko niewielkie, ale też mieć jedną odpowiedzialność.

Czasami klasa z niewielkim lub mającym jedną odpowiedzialność kontraktem jest niemożliwa do osiągnięcia lub niepożądana. W takich przypadkach powinna ona implementować wiele kontraktów zamiast jednego połączonych. Chcemy mieć wiele kontraktów, aby zredukować liczbę daleko sięgających zależności.

Ileokroć klasa bazowa lub interfejs są modyfikowane, klasy pochodne również wymagają zmian. W najlepszym przypadku muszą zostać przekompilowane. Ograniczając zakres kontraktu, możemy ograniczyć wpływ zmian wprowadzanych w tym kontrakcie i poprawić jakość ogólnej architektury systemu.

Zasada odwrócenia zależności

Odwrócenie zależności jest ważne z kilku powodów, między innymi dlatego, że powoduje zwiększenie elastyczności kodu, zmniejszenie jego podatności na błędy i zwiększenie możliwości wielokrotnego użycia kodu.

Odwrócenie zależności pomaga w osiągnięciu architektury opartej na wtyczkach. Definiując kontrakt interakcji, moduł może określić, jak chce podejmować interakcje z zależnościami. W ten sposób zależności uzależniają się od kontraktu.

Ponieważ moduł na najwyższym poziomie nie ma zależności wychodzących, może być wdrażany niezależnie. Wdrażanie produkcyjne części aplikacji prawie nigdy się nie zdarza, ale posiadanie tak niezależnej biblioteki daje ogromną wygodę w postaci braku konieczności rekompilacji, kiedy zmieniane są zależności.

Podczas standardowego wytwarzania oprogramowania zależności zmieniają się znacznie częściej niż moduły wysokiego poziomu. Zmiany te powodują konieczność rekompilacji. Kiedy zależności w aplikacji spływają w dół hierarchii, rekompilacja zależności uruchamia również rekompilację biblioteki od niej zależnej. W efekcie zmiana klasy pomocniczej w niewielkiej, ale często wykorzystywanej bibliotece spowoduje rekompilację całej aplikacji.

Jeżeli jednak odwracasz zależności, tego typu zmiana wywoła tylko konieczność rekompilacji biblioteki zawierającej tę klasę pomocniczą i biblioteki aplikacji. Nie będzie wymagała rekompilacji wszystkich bibliotek pośrednich.

To tyle, jeśli chodzi o zasady SOLID. Pamiętaj o nich podczas wyboru biblioteki imitującej. Upewnij się, że biblioteka imitująca nie skłoni Cię do budowy sztywnego, wrażliwego i trudnego w utrzymaniu systemu.

Powitanie zależne od czasu

Rozbudujmy nieco klasyczny przykład „Witaj, świecie”. Może by tak wyświetlać powitanie uzależnione od pory dnia? Przykład jest następujący:

```

Jako odwiedzający stronę
  Chcę otrzymać powitanie odpowiednie dla aktualnej pory dnia
  Aby móc zaplanować zgłaszanie moich prelekcji
Zakładając że jest przed godziną 18
  Kiedy żądane jest powitanie
  Wtedy otrzymam powitanie dzienne
Zakładając że jest po godzinie 18
  Kiedy żądane jest powitanie
  Wtedy otrzymam powitanie wieczorne
  
```

Być może pomyślałeś sobie: „To proste, wystarczy, że szybko sklecę metodę zwracającą odpowiednią wiadomość”. Oczywiście miałbyś rację. To dość proste zadanie. Mogłbyś napisać coś takiego:

```
public string GetGreeting()
{
    if (DateTime.Now.Hour < 18)
        return "Dzień dobry";

    return "Dobry wieczór";
}
```

W rozdziale 1., „Dlaczego TDD jest ważne?”, omawialiśmy trzy prawa TDD. Pierwsze prawo mówi, że nie wolno napisać ani jednej linii kodu, nie utworzywszy testu kończącego się powodzeniem.

Kruche testy

Możesz powiedzieć, że przecież to jest tak banalna metoda. A gdybyś napotkał błąd? Albo gdybyś chciał napisać testy dla metody w późniejszym czasie? Musiałbyś uruchamiać zestaw testu o odpowiedniej porze dnia, żeby sprawdzić, czy test przechodzi pomyślnie? Czy musiałbyś zmieniać testy w zależności od pory dnia, o jakiej są uruchamiane?

Błędne wyniki pozytywne i błędne wyniki negatywne

Gdybyśmy pozostawili kod zwracający wiadomość w takim stanie, w jakim jest, i napisali test dla metody, mógłby wyglądać tak:

```
[Fact]
public void GivenEvening_ThenAfternoonMessage()
{
    // Arrange
    // Act
    var message = GetGreeting();

    // Assert
    Assert.Equal("Dobry wieczór", message);
}
```

Czy dostrzegasz problem? W samym kodzie testu nie ma żadnego błędu. Problem polega na tym, że kod produkcyjny zwróci inną wiadomość w zależności od godziny, w której zostanie uruchomiony. To oznacza, że jeżeli uruchomisz test do godziny 18, skończy się powodzeniem. Jeżeli uruchomisz go później, skończy się niepowodzeniem.

Wyodrębnianie klasy DateTime

Klasa `DateTime` jest częścią frameworka `.NET`, a co za tym idzie, powinna zostać wyodrębniona z naszego systemu. Zazwyczaj chcemy, aby system był zależny od interfejsów, pozwalając nam podmieniać implementacje w czasie wykonywania.

Oto przykład interfejsu `ITimeManager`:

```
public interface ITimeManager
{
    DateTime Now { get; }
}
```

Na potrzeby testów możesz zbudować taką implementację tego interfejsu:

```
public class TestTimeManager : ITimeManager
{
    public Func<DateTime> CurrentTime = () => DateTime.Now;

    public void SetDateTime(DateTime now)
    {
        CurrentTime = () => now;
    }

    public DateTime Now => CurrentTime();
}
```

To pozwala nam ustawić wartość aktualnego czasu, dzięki czemu będziemy mogli przekazać znaną wartość do metod testowych. Spójrzmy jeszcze raz na testy:

```
[Theory]
[InlineData(18)]
[InlineData(19)]
[InlineData(20)]
[InlineData(21)]
[InlineData(22)]
[InlineData(23)]
public void GivenAfternoon_ThenAfternoonMessage(int hour)
{
    // Arrange
    var afternoonTime = new TestTimeManager();
    afternoonTime.SetDateTime(new DateTime(2017, 7, 13, hour, 0, 0));
    var messageUtility = new MessageUtility(afternoonTime);

    // Act
    var message = messageUtility.GetGreeting();

    // Assert
    Assert.Equal("Dobry wieczór", message);
}

[Theory]
[InlineData(0)]
[InlineData(1)]
[InlineData(2)]
[InlineData(3)]
[InlineData(4)]
[InlineData(5)]
[InlineData(6)]
[InlineData(7)]
[InlineData(8)]
[InlineData(9)]
```

```

[InlineData(10)]
[InlineData(11)]
[InlineData(12)]
[InlineData(13)]
[InlineData(14)]
[InlineData(15)]
[InlineData(16)]
[InlineData(17)]
public void GivenMorning_ThenMorningMessage(int hour)
{
    // Arrange
    var morningTime = new TestTimeManager();
    morningTime.SetDateTime(new DateTime(2017, 7, 13, hour, 0, 0));
    var messageUtility = new MessageUtility(morningTime);

    // Act
    var message = messageUtility.GetGreeting();

    // Assert
    Assert.Equal("Dzień dobry", message);
}

```

Nasz kod produkcyjny wyglądałby tak:

```

public class MessageUtility
{
    private readonly ITimeManager _timeManager;

    public MessageUtility(ITimeManager timeManager)
    {
        _timeManager = timeManager;
    }

    public string GetMessage()
    {
        if (_timeManager.Now.Hour < 18)
            return "Dzień dobry";

        return "Dobry wieczór";
    }
}

```

Rodzaje sobowtórów testowych

Jest wiele rodzajów sobowtórów. Rodzaje te można ogólnie pogrupować jako atrapy, zaślepki, szpiegi, imitacje i fałszywki. W dalszej części omówimy poszczególne typy i dla każdego z nich pokażemy przykłady w C# i JavaScriptcie.

Atrapy

Atrapa (ang. *dummy*) to najprostsza forma sobowtórów testowych. Atrapa nie ma żadnej znaczącej funkcjonalności. Nie oczekujemy, że klasa lub metoda atrapy zostanie wykorzystana do wyprodukowania wyniku w metodzie testowanej.

Atrapy są stosowane najczęściej, kiedy testowana klasa ma zależności, z których tak naprawdę nie korzysta.

Atrapy powstają poprzez stworzenie kopii lub instancji klasy lub metody, nie robiąc absolutnie nic w jej ciele. Metody nie zwracające wartości będą puste, a metody zwracające wartość będą zwracały wyjątek lub najprostszą formę typu zwracanego.

Atrapa logowania

Usługa *logowania* to doskonały przykład elementu do zastąpienia atrapą. Mało prawdopodobne jest (i nie jest polecane), że podczas testowania metody zechcesz również testować logowanie.

Przykład w C#

Oto przykład klasy `DummyLogger` w C#. Zauważ, że kiedy wywoływana jest metoda `Log`, nic się nie dzieje:

```
enum LogLevel
{
    None = 0,
    Error = 1,
    Warning = 2,
    Success = 3,
    Info = 4
}

interface ILogger
{
    void Log(LogLevel type, string message);
}

class DummyLogger: ILogger
{
    public void Log(LogLevel type, string message)
    {
        // Nie rób nic
    }
}
```

Przykład w JavaScriptcie

Oto przykład klasy `DummyLogger` w JavaScriptcie. Zauważ, że kiedy wywoływana jest metoda `info`, `warn`, `error` lub `success`, nic się nie dzieje:

```
export class DummyLogger {
    info(message) {
    }

    warn(message) {
    }

    error(message) {
    }
}
```

```

    }

    success(message) {
    }
}

```

Zaśleпки

Zaśleпка (ang. *stub*) to kolejny poziom po atrapach. Zaśleпка daje zawsze tę samą odpowiedź, niezależnie od przekazanych do niej parametrów.

Zaśleпки są wykorzystywane, kiedy chcesz przetestować różne ścieżki wykonania kodu. Przykładem może być błąd, który musi zostać zwrócony w przypadku spełnienia konkretnych warunków.

Zaśleпки powstają poprzez stworzenie kopii lub nadpisanie klasy bądź metody, która ma zwracać wartość zaślepioną, i ustawienie jej tak, aby tę wartość zwracała. Pamiętaj: zaśleпки nie przetwarzają parametrów, musisz więc tylko zwrócić potrzebną wartość.

Przykład w C#

Oto przykład klasy `StubSpeakerContactServiceError` w C#. Zauważ, że po wywołaniu `MessageSpeaker` zwracany jest nowy wyjątek, `UnableToContactSpeakerException`:

```

class StubSpeakerContactServiceError : ISpeakerContactService
{
    public void MessageSpeaker(string message)
    {
        throw new UnableToContactSpeakerException();
    }
}

```

Przykład w JavaScriptcie

Oto przykład funkcji `stubSpeakerReducer` w JavaScriptcie. Zauważ, że niezależnie od akcji przekazywanej do funkcji do tablicy błędów w stanie aplikacji wstawiany jest błąd `UNABLE_TO_RETRIEVE_SPEAKERS`:

```

import { SpeakerErrors } from './errors';
import { SpeakerFilters } from './actions';

const initialState = {
    speakerFilter: SpeakerActions.SHOW_ALL,
    speakers: [],
    errors: []
};

export function stubSpeakerReducer(state, action) {
    state = state || initialState;
    state.speakerFilter = action.filter || SpeakerFilters.SHOW_ALL;
    state.errors.push(SpeakerErrors.UNABLE_TO_RETRIEVE_SPEAKERS);
    return state;
}

```

Szpiegi

Szpieg (ang. *spy*) to kolejna ewolucja sobowtórów. Szpieg zwraca wartość podobną do zaślepki, ale z jedną bardzo ważną i pomocną różnicą: może zwracać informacje powiązane z wywołaniem funkcji.

Szpiegów najczęściej się używa, kiedy chcesz zweryfikować, czy funkcja została wywołana z odpowiednimi parametrami. Jest to najbardziej użyteczne podczas testowania granic kodu zewnętrznego. Na przykład istotne jest, czy aplikacja poprawnie konfiguruje połączenie z bazą danych, korzystając z danych dostępowych pochodzących z usługi konfiguracji. Ponadto w niektórych przypadkach trudno zmierzyć efekty uboczne wynikające z użycia testowanej funkcji czy metody. W takich przypadkach możesz użyć szpiega, aby upewnić się, że ta funkcja czy metoda faktycznie jest wywoływana.

Tworzenia szpiega rozpoczyna się od zaślepki, dodając funkcjonalność pozwalającą określić, czy funkcja została wywołana lub ile razy została wywołana, albo zwracającą przekazane do funkcji parametry.

Przykład w C#

Oto przykład klasy `SpySpeakerContactService`, która pozwala określić, czy i ile razy usługa została użyta:

```
class SpySpeakerContactService : ISpeakerContactService
{
    public bool MessageSpeakerHasBeenCalled { get; private set; }

    public int MessageSpeakerCallCount { get; private set; }

    public void MessageSpeaker(string message)
    {
        MessageSpeakerHasBeenCalled = true;
        MessageSpeakerCallCount++;
    }
}
```

Przykład w JavaScriptcie

Oto przykład funkcji `spySpeakerReducer` w JavaScriptcie. Funkcja ta pozwala określić, ile razy została wywołana:

```
import { speakerReducer as original_speakerReducer } from './reducers';

export let callCounter = 0;

export function spySpeakerReducer(state, action) {
    callCounter++;
    return original_speakerReducer(state, action);
}
```

Imitacje

Imitacje (ang. *mocks*) to w zasadzie programowalne szpiegi. Imitacje są użyteczne, kiedy chcesz wykorzystać tego samego sobowtóra testowego w wielu testach. Zwracają taką wartość, jaką ustawisz. Należy jednak pamiętać, że imitacje wciąż nie mają w sobie żadnej logiki. Zwracają wartość, która została wyspecyfikowana, i nie sprawdzają przekazywanych do funkcji parametrów.

Imitacje są używane we wszystkich sytuacjach, w których wykorzystywane są atrapy, zaślepki i szpiegi. Imitacje są bardziej rozbudowanymi implementacjami sobowtórów testowych i z tego powodu nie wszędzie ich wykorzystanie może być pożądane. Imitacje są rzadziej używane w wielu miejscach, ponieważ dla każdego testu konieczne jest przygotowanie danych imitacji, podczas gdy atrapa, zaślepka czy szpieg mają ustalone wartości zwracane i nie muszą być konfigurowane. Przygotowanie zwracanych danych testowych jest często trudniejsze niż stworzenie całej zaślepki lub szpiega.

Imitacje powstają poprzez skopiowanie klasy lub metody i stworzenie właściwości, która może być ustawiona jako wartość zwracana metody. Następnie wartość ta jest zwracana w metodzie imitującej. Po utworzeniu, przed każdym testem, należy ustawić wartość imitacji.

Przykład w C#

Oto przykład klasy `MockDateTimeService` w C#. Klasa ta pozwala ustawić datę zwracaną przez klasę, co pozwoli w sposób niezawodny testować, jak inne części systemu będą zachowywały się w kontekście określonych dat:

```
class MockDateTimeService
{
    public DateTime CurrentDateTime { get; set; } = new DateTime();

    public DateTime UTCNow()
    {
        return CurrentDateTime.ToUniversalTime();
    }
}
```

Przykład w JavaScriptcie

Oto przykład klasy `MockDateTimeService` w JavaScriptcie. Tak jak w C#, pozwala ona ustawić datę zwracaną przez usługę, aby móc sprawdzić, jak inne części systemu zachowują się w kontekście zadanej daty:

```
export class MockDateTimeService {
    constructor() {
        this.currentDateTime = new Date(2000, 0, 1);
    }

    now() {
        return this.currentDateTime;
    }
}
```

Fałszywki

Fałszywka (ang. *fake*) to ostatni i najbardziej rozbudowany rodzaj sobowtórów testowych. Fałszywka to klasa próbująca zachowywać się tak, jakby nie była sobowtorem. Chociaż fałszywka nie będzie łączyła się z bazą danych, będzie próbowała zachowywać się tak, jakby faktycznie była z bazą połączona. Fałszywka nie będzie korzystała z zegara systemowego, ale będzie starała się jak najlepiej odwzorować korzystanie z niego.

Fałszywki albo dodają dodatkowe funkcje na potrzeby testowania, albo zapobiegają udziałowi zewnętrznych bibliotek i systemów w testach. Większość aplikacji jest połączonych z jakimś źródłem danych. Może zostać stworzone fałszywe repozytorium korzystające ze swojego własnego źródła danych w pamięci, ale poza tym zachowujące się zupełnie jak normalne połączenie z bazą danych.

Fałszywki są tworzone poprzez wygenerowanie nowej klasy lub metody i zawarcie w niej wystarczającej ilości logiki, aby była nieodróżnialna od produkcyjnej klasy lub metody. Jedynym istotnym czynnikiem odróżniającym ją od wersji produkcyjnej jest to, że fałszywka nie wykonuje połączeń na zewnątrz aplikacji i najprawdopodobniej daje testerowi kontrolę nad zestawem danych.

Przykład w C#

Oto przykład klasy `FakeRepository` i związanych z nią interfejsów. Jest to fałszywa implementacja generycznego repozytorium:

```
public interface IRepository<T>
{
    T Get(Func<T, bool> predicate);
    IQueryable<T> GetAll();
    T Save(T item);
    IRepository<T> Include(Expression<Func<T, object>> path);
}

public interface IIdentity
{
    int Id {get;set;}
}

public class FakeRepository<T> : IRepository<T> where T : IIdentity
{
    private int _identityCounter = 0;
    public IList<T> DataSet { get; set; } = new List<T>();

    public T Get(Func<T, bool> predicate)
    {
        return GetAll().Where(predicate).FirstOrDefault();
    }

    public IQueryable<T> GetAll()
    {
        return DataSet.AsQueryable();
    }
}
```

```

    }

    public T Save(T item)
    {
        return item.Id == default(int) ? Create(item) : Update(item);
    }

    public IRepository<T> Include(Expression<Func<T, object>> path)
    {
        // Tutaj nie ma nic do zrobienia, ponieważ jest to funkcja na potrzeby EntityFramework
        // Używamy Linq to Objects, nie ma więc potrzeby wykorzystania tej funkcji
        return this;
    }

    private T Create(T item)
    {
        item.Id = ++_identityCounter;
        DataSet.Add(item);
        return item;
    }

    private T Update(T item)
    {
        var found = Get(x => x.Id == item.Id);

        if(found == null)
        {
            throw new KeyNotFoundException($"Element o Id {item.Id} nie został
            ↳ znaleziony!");
        }

        DataSet.Remove(found);
        DataSet.Add(item);
        return item;
    }
}

```

Przykład w JavaScriptcie

Oto przykład klasy FakeDataContext w JavaScriptcie:

```

export class FakeDataContext {
    _identityCounter = 1;
    _dataSet = [];

    get DataSet() {
        return this._dataSet;
    }

    set DataSet(value) {
        this._dataSet = value;
    }

    get(predicate) {

```

```

    if (typeof(predicate) !== 'function') {
      throw new Error('Predykat musi być funkcją');
    }

    const resultSet = this._dataSet.filter(predicate);

    return resultSet.length >= 1 ? {...resultSet[0]} : null;
  }

  getAll() {
    return this._dataSet.map((x) => {
      return {...x};
    });
  }

  save(item) {
    return item.id ? this.update(item) : this.create(item);
  }

  update(item) {
    if (!this._dataSet.some(x => x.id === item.id)) {
      this._dataSet.push({...item});
    } else {
      let itemIndex = this._dataSet.findIndex(x => x.id === item.id);
      this._dataSet[itemIndex] = {...item};
    }
    return {...item};
  }

  create(item) {
    let newItem = {...item};
    newItem.id = this._identityCounter++;
    this._dataSet.push({...newItem});

    return {...newItem};
  }
}

```

Przykład wielopoziomowy

Wróćmy teraz do kontrolera API z rozdziału 2., „Przygotowanie środowiska testowego w .NET”. Wpisane w kodzie dane zwracane bezpośrednio przez kontroler nie stanowią dobrej podstawy dla naszej aplikacji. Większość nowoczesnych aplikacji .NET, niezależnie od ich rozmiaru, budowana jest w jakimś wariantcie architektury wielopoziomowej. Należy odseparowywać logikę biznesową od prezentacji — w tym przypadku prezentacji poprzez końcówkę API.

Przygotujemy interfejs dla usługi mówcy, co pozwoli nam wykorzystać wstrzykiwanie zależności i zapewnienie kontrolerowi konkretnej implementacji usługi. W kolejnym kroku zweryfikujemy, czy wywoływana jest odpowiednia metoda nowej usługi. Aby usunąć z kontrolera logikę biznesową, trzeba przebudować część testów.

Warstwa prezentacji

Na początek dodajmy nowy test, sprawdzający, czy kontroler przyjmuje interfejs `ISpeakerService`:

```
[Fact]
public void ItAcceptsInterface()
{
    // Arrange
    ISpeakerService testSpeakerService = new TestSpeakerService();

    // Act
    var controller = new SpeakerController(testSpeakerService);

    // Assert
    Assert.NotNull(controller);
}
```

Czas teraz sprawić, aby test kończył się powodzeniem. `SpeakerController` musi przyjmować interfejs `ISpeakerService`, musimy więc dodać pole i konstruktor w klasie kontrolera:

```
public SpeakerController(ISpeakerService speakerService)
{
}
```

Projekt nie powinien się teraz kompilować. Jest tak dlatego, że w poprzednim przykładzie z rozdziału 2., „Przygotowanie środowiska testowego w .NET”, definiujemy instancję kontrolera w konstruktorze klasy testowej. Zmodyfikuj konstruktor i dodaj tworzenie instancji `TestSpeakerService`, która implementuje interfejs `ISpeakerService`, a następnie przekaż ją do `SpeakerController`. Klasę `TestSpeakerService` możesz zdefiniować wewnątrz klasy testów.

```
public SpeakerControllerSearchTests()
{
    var testSpeakerService = new TestSpeakerService();
    _controller = new SpeakerController(testSpeakerService);
}
```

Teraz należy się upewnić, czy metoda `Search` klasy `SpeakerService` jest wywoływana wewnątrz kontrolera. Ale jak to zrobić? Jednym ze sposobów jest wykorzystanie frameworka imitującego o nazwie `Moq`.

Moq

Aby dodać `Moq` do projektu testowego, kliknij prawym przyciskiem na projekcie testowym i wybierz *Zarządzaj pakietami NuGet...* Odnajdź `Moq` i wybierz instalację najnowszej stabilnej wersji. Nie będziemy się zbyt głęboko zagłębiać w tematykę `Moq`, ale pokażemy, w jaki sposób framework imitujący pomaga w przygotowaniu testów możliwości aplikacji.

Dodaj test sprawdzający, czy metoda `Search` klasy `SpeakerService` została wywołana raz wewnątrz akcji `Search` kontrolera:

```
[Fact]
public void ItCallsSearchServiceOnce()
{
    // Arrange

    // Act
    _controller.Search("jan");

    // Assert
    _speakerServiceMock.Verify(mock => mock.Search(It.IsAny<string>()),
    Times.Once());
}
```

Aby test zaczął przechodzić pomyślnie, musisz wykonać jeszcze trochę konfiguracji w konstruktorze klasy testowej:

```
private readonly SpeakerController _controller;
private static Mock<ISpeakerService> _speakerServiceMock;

public SpeakerControllerSearchTests()
{
    var speaker = new Speaker
    {
        Name = "test"
    };

    // Zdefiniowanie imitacji
    _speakerServiceMock = new Mock<ISpeakerService>();

    // Kiedy search zostanie wywołane, zwróć listę mówców zawierającą mówcę
    _speakerServiceMock.Setup(x => x.Search(It.IsAny<string>()))
        .Returns(() => new List<Speaker> { speaker });

    // Przekaż obiekt jako ISpeakerService
    _controller = new SpeakerController(_speakerServiceMock.Object);
}
```

Koniecznien zmodyfikuj interfejs, aby aplikacja kompilowała się poprawnie:

```
public interface ISpeakerService
{
    IEnumerable<Speaker> Search(string searchString);
}
```

Teraz spraw, aby testy przechodziły, wywołując metodę Search klasy SpeakerService w metodzie Search kontrolera. Jeżeli jeszcze tego nie zrobiłeś, stwórz pole _speakerService, do którego w konstruktorze jest przypisywany parametr speakerService:

```
private readonly ISpeakerService _speakerService;

public SpeakerController(ISpeakerService speakerService)
{
    _speakerService = speakerService;
}

[Route("search")]
public IActionResult Search(string searchString)
```

```

{
    var hardCodedSpeakers = new List<Speaker>
    {
        new Speaker{Name = "Jan"},
        new Speaker{Name = "Janusz"},
        new Speaker{Name = "Janina"},
        new Speaker{Name = "Bartosz"},
    };

    _speakerService.Search("foo");

    var speakers = hardCodedSpeakers.Where(x =>
        x.Name.StartsWith(searchString,
            StringComparison.OrdinalIgnoreCase)).ToList();

    return Ok(speakers);
}

```

Następnie dodajmy test sprawdzający, czy searchString przekazany do metody Search kontrolera to searchString przekazywany do metody Search klasy SpeakerService:

```

[Fact]
public void GivenSearchStringThenSpeakerServiceSearchCalledWithString()
{
    // Arrange
    var searchString = "jan";

    // Act
    _controller.Search(searchString);

    // Assert
    _speakerServiceMock.Verify(mock => mock.Search(searchString),
        Times.Once());
}

```

Aby test przechodził, przekaz searchString do metody Search klasy obiektu z pola _speakerService:

```

_speakerService.Search(searchString);

```

Teraz musimy zapewnić, aby wyniki metody Search klasy SpeakerService były poprawnie zwracane z akcji:

```

[Fact]
public void GivenSpeakerServiceThenResultsReturned()
{
    // Arrange
    var searchString = "jan";

    // Act
    var result = _controller.Search(searchString) as OkObjectResult;

    // Assert
    Assert.NotNull(result);
    var speakers = ((IEnumerable<Speaker>)result.Value).ToList();
    Assert.Equal(_speakers, speakers);
}

```

Pamiętaj, że wyniki zwracane przez metodę `Search` klasy `SpeakerService` są definiowane przez imitację. Musisz wyekstrahować pole, aby móc przetestować, czy rezultaty zwracane przez akcje są takie same jak te zdefiniowane przez imitację:

```
private readonly SpeakerController _controller;
private static Mock<ISpeakerService> _speakerServiceMock;
private readonly List<Speaker> _speakers;

public SpeakerControllerSearchTests()
{
    _speakers = new List<Speaker> { new Speaker
    {
        Name = "test"
    } };

    _speakerServiceMock = new Mock<ISpeakerService>();
    _speakerServiceMock.Setup(x => x.Search(It.IsAny<string>()))
        .Returns(() => _speakers);

    _controller = new SpeakerController(_speakerServiceMock.Object);
}
```

Wciąż mamy problem z danymi wpisanymi w kodzie. Nie zapomnij usunąć niepotrzebnego kodu podczas pracy nad testami. Pamiętaj: czerwony, zielony, refaktoryzacja. Dotyczy to w takim samym stopniu testów co kodu produkcyjnego.

Po usunięciu danych z kodu możesz napotkać testy kończące się wynikiem negatywnym. Na razie pomiń te testy, ponieważ tę logikę będziemy przenosić do innej części aplikacji. Czas teraz stworzyć usługę `SpeakerService`:

```
xUnit
[Fact(Skip="Powód pominięcia")]
MSTest
[Skip]
```

Warstwa biznesowa

Zastanówmy się nad efektywnym organizowaniem testów. Nawigowanie po rozwiązaniu może stawać się coraz trudniejsze wraz z rozrostem aplikacji i liczby plików z testami. Jednym z rozwiązań jest tworzenie osobnych folderów dla każdej testowanej klasy i osobnych plików dla każdej publicznej metody w ramach danej klasy. Mogłoby to wyglądać tak:

```
SpeakerService -> Search
```

Nie musisz zabierać się za to teraz, ale nie zaszkodzi mieć planu na przyszłość. Aplikacje rosną dość szybko i zanim się obejrzyysz, będziesz mieć w swoim rozwiązaniu trzynaście projektów. Na tę chwilę możesz zdecydować się stworzyć projekt *Services* z folderem *ServicesTest*, aby oddzielić warstwę biznesową i związane z nią testy od warstwy prezentacji i testów z nią związanych. Zostawimy to jako ćwiczenie dla Czytelnika.

Stwórzmy teraz klasę `SpeakerService` — tutaj będziesz tworzyć wszystkie metody testowe dla metody `Search` klasy `SpeakerService`:

```
[Fact]
public void ItExists()
{
    var speakerService = new SpeakerService();
}
```

Kiedy sprawisz, że test zacznie przechodzić, stwórz kilka testów, które potwierdzą, iż metoda `Search` istnieje i zwraca kolekcję mówców:

```
[Fact]
public void ItHasSearchMethod()
{
    var speakerService = new SpeakerService();
    speakerService.Search("test");
}
```

Następnie sprawdź, czy `SpeakerService` implementuje interfejs `ISpeakerService`:

```
[Fact]
public void ItImplementsISpeakerService()
{
    var speakerService = new SpeakerService();
    Assert.True(speakerService is ISpeakerService);
}
```

Klasa `SpeakerService` powinna teraz wyglądać podobnie jak tu:

```
public class SpeakerService : ISpeakerService
{
    public IEnumerable<Speaker> Search(string searchString)
    {
        return new List<Speaker>();
    }
}
```

Pamiętaj: posuwaj się do przodu powoli i metodycznie. Nie wolno Ci napisać ani jednej linii kodu produkcyjnego, dopóki nie napiszesz działającego testu. Nie wolno Ci też napisać więcej kodu produkcyjnego niż to konieczne do pomyślnego przechodzenia testu.

Zacznij teraz przenosić *pominięte* testy z pliku z testami kontrolera do pliku z testami usługi wyszukiwania mówców. Zacznij od `GivenExactMatchThenOneSpeakerInCollection`:

```
[Fact]
public void GivenExactMatchThenOneSpeakerInCollection()
{
    // Arrange

    // Act
    var result = _speakerService.Search("Janusz");

    // Assert
    var speakers = result.ToList();
    Assert.Equal(1, speakers.Count);
    Assert.Equal("Janusz", speakers[0].Name);
}
```

Spraw, aby test kończył się sukcesem, i przejdź do `GivenCaseInsensitiveMatchThenSpeakerInCollection`:

```
[Theory]
[InlineData("Janusz")]
[InlineData("janusz")]
[InlineData("JaNuSz")]
public void GivenCaseInsensitiveMatchThenSpeakerInCollection(string searchString)
{
    // Arrange

    // Act
    var result = _speakerService.Search(searchString);

    // Assert
    var speakers = result.ToList();
    Assert.Equal(1, speakers.Count);
    Assert.Equal("Janusz", speakers[0].Name);
}
```

I w końcu `GivenNoMatchThenEmptyCollection` i `Given3MatchThenCollectionWith3Speakers`:

```
[Fact]
public void GivenNoMatchThenEmptyCollection()
{
    // Arrange

    // Act
    var result = _speakerService.Search("ZZZ");

    // Assert
    var speakers = result.ToList();
    Assert.Equal(0, speakers.Count);
}

[Fact]
public void Given3MatchThenCollectionWith3Speakers()
{
    // Arrange

    // Act
    var result = _speakerService.Search("jan");

    // Assert
    var speakers = result.ToList();
    Assert.Equal(3, speakers.Count);
    Assert.True(speakers.Any(s => s.Name == "Jan"));
    Assert.True(speakers.Any(s => s.Name == "Janusz"));
    Assert.True(speakers.Any(s => s.Name == "Janina"));
}
```

Wraz z nabieraniem doświadczenia będziesz się czuć coraz bardziej komfortowo. Przydatne może wtedy być listowanie testów, które chciałbyś zaimplementować. Może to być po prostu zapisanie ich na kartce papieru lub stworzenie pomijanych bądź ignorowanych zaślepek testów w IDE.

Jeżeli wszystko zrobiłeś poprawnie, powinieneś mieć coś podobnego jak tutaj:

```
public class SpeakerService : ISpeakerService
{
    public IEnumerable<Speaker> Search(string searchString)
    {
        var hardCodedSpeakers = new List<Speaker>
        {
            new Speaker{Name = "Jan"},
            new Speaker{Name = "Janusz"},
            new Speaker{Name = "Janina"},
            new Speaker{Name = "Bartosz"},
        };

        var speakers = hardCodedSpeakers.Where(x =>
            x.Name.StartsWith(searchString,
                StringComparison.OrdinalIgnoreCase)).ToList();

        return speakers;
    }
}
```

Przesunęliśmy teraz zapisane w kodzie dane z kontrolera do warstwy biznesowej w `SpeakerService`. Może Ci się wydawać, że dużo wysiłku włożyliśmy tylko po to, żeby problem przesunąć w inne miejsce. Jest to w pewnym stopniu prawda, ale daje nam to lepszą pozycję wyjściową do dalszej pracy. Logika została przeniesiona do klasy, która może być używana także w innych miejscach systemu i potencjalnie przez nowe interfejsy (pomyśl o aplikacjach natywnych i mobilnych), które nie miałyby dostępu do naszego pierwotnego kontrolera.

Pracę z tym przykładem będziemy kontynuować w dalszych rozdziałach. W *końcu* uda nam się pozbyć danych z kodu i zaimplementować warstwę dostępu do danych z wykorzystaniem Entity Framework. To wszystko można osiągnąć, korzystając z TDD.

Podsumowanie

W tym rozdziale omówiliśmy pułapki przeszkadzające w pracy z TDD, takie jak zależność od zewnętrznych bibliotek, bezpośrednie tworzenie instancji klas i wrażliwe testy. Omówiliśmy również sposoby na uniknięcie lub ominięcie tych problemów. Wprowadziliśmy pojęcie zasad SOLID. Przedstawiliśmy też różne typy sobowtórów testowych i wyjaśniliśmy, kiedy warto korzystać z poszczególnych typów. Na koniec spróbowałeś zbudować aplikację wielowarstwową i ją przetestować.

W rozdziale 5., „Tabula rasa — podejście do aplikacji na sposób TDD”, omówimy, jak zacząć budowanie aplikacji z uwzględnieniem podejścia TDD, jak zamienić teorię w praktykę i jak najlepiej rozwijać aplikację poprzez testy.

Skorowidz

A

- abstrahowanie
 - interfejsu, 210
 - klasy konkretnej, 211
 - warstwy danych, 185
- abstrakcja, 309
 - klasy zewnętrznej, 320
- Ajax, 264
- akcja
 - pobierania prelegenta, 235, 248
 - typu thunk, 237
- akcje w Reduxie, 229
- aktualizacja prelegenta, 204
- API, 51, 150, 165
- aplikacja
 - Speaker Meet, 114, 120, 293
 - TODO, 289
 - dodanie testów, 289, 290
 - kod produkcyjny, 290, 292
 - oznaczanie zadania, 289
- aplikacje
 - C#, 149
 - w JavaScriptcie, 225
- architektura
 - heksagonalna, 127
 - heksagonalna wielowarstwowa, 126
- asercja, 114
- atrapa, 86
 - logowania, 87

B

- baza danych w pamięci, 276
- BDD, Behavior Driven Development, 20
- bezpieczna refaktoryzacja, 308, 317
- biblioteka
 - Chai, 68, 226
 - Enzyme, 69, 226
 - Mocha, 226

- Redux, 229
- Sinon, 68, 226, 264
- błędne wyniki
 - negatywne, 84
 - pozytywne, 84

C

- Chai, 67, 68
 - konfiguracja biblioteki, 226
- ContextFixture, 278
- Create React App, 65
 - tworzenie aplikacji, 66
- CRUD, 186

D

- dane niespodziewane, 324
- DbContext, 220
- definiowanie
 - interfejsu użytkownika, 137
 - warstwy biznesowej, 144
 - źródła danych, 130
- dodawanie testów, 305
- dziedziczenie kodu, 315

E

- ekspert TDD, 355
- elementy statyczne, 78
- Entity Framework, 219
- Enzyme, 69
 - konfiguracja biblioteki, 226
- epiki, 118

F

- fabryka, 163, 173
- FakeRepository, 161, 163
- falszywka, 91
- FizzBuzz, 47, 287

- framework
 - Jest, 67
 - Mocha, 67
- frameworki imitujące, 80
- funkcja, 118, 191, 307

G

- generyczne repozytorium, 128, 210
- Gherkin, 29
- globalny stan, 307
- golenie jaka, 102
- gra Mastermind, 316
- Gravatar, 178

H

- historie, 119
 - użytkownika, 27
- Homebrew, 60

I

- IDE, 40, 62, 64
- imitacja, 90
 - odpowiedzi Ajaxa, 264
 - serwera, 266
 - usługi API, 231, 246, 261
- instalacja
 - Create React App, 66
 - Node, 58
 - Linux, 59
 - Mac OSX, 60
 - Windows, 60
 - NPM, 62
 - SDK .NET Core, 39
 - Visual Studio Code, 63
 - Linux, 63
 - Mac, 63
 - Windows, 63
 - Visual Studio Community, 46

VS Code, 40
 WebStorma, 64
 Linux, 64
 Mac, 65
 Windows, 65
 wtyczki, 65
 integracja, 261
 interfejs, 295
 abstrahowanie, 210
 IGravatarService, 179
 implementacja
 wersji produkcyjnej, 183
 wersji testowej, 179
 IRepository, 161
 użytkownika, 135, 137
 webowy, 125

J

Jest, 67

K

kata, 72, 73
 klasa, 307
 DateTime, 84
 klasy
 wyodrębnianie, 309
 kod
 dziedziczenie, 315
 produkcyjny, 290, 292
 zastany, 299, 301, 302
 problemy, 302, 308
 skutki uboczne, 302
 zapobieganie powstawaniu, 301
 zbyt sprytny, 304
 zewnętrzny, 79, 304
 komponent
 dla szczegółów prelegenta, 257
 listy prelegentów, 240
 React, 228
 konfiguracja
 API, 274
 aplikacji, 273
 biblioteki
 Chai, 70, 226
 Enzyme, 226
 Mocha, 70, 226
 Sinon, 226
 środowiska testowego, 64, 65
 konstruktor, 306
 kontrolowanie rezultatów, 355
 konwersja
 metody
 Create, 212
 Delete, 214
 Get, 213
 GetAll, 213
 Update, 214
 wartości na zmienne, 308

L

lista prelegentów, 230

M

magazyn Reduxa, 229
 mapowanie obiektowo-relacyjne,
 ORM, 219
 mechanizm Test Fixture, 278
 mentor, 355
 metoda
 Create, 188, 212, 215
 Delete, 189, 214
 Get, 187, 213, 216
 GetAll, 187, 213, 217
 Update, 190, 214, 218
 metody
 generyczne, 212, 213, 214
 statyczne, 307
 wyodrębnianie, 309
 miękkie usuwanie, 164
 mikrousługi, 128
 minimalny wykonalny produkt, 104
 Mocha, 67
 konfiguracja biblioteki, 226
 modele Entity Framework, 220
 Moq, 94, 152

N

naprawa
 błędów, 313
 testów, 264
 nietestowalny kod, 78
 Node.js, 57
 NPM, Node Package Manager, 61, 62

O

odseparowanie problemów, 177
 odwołania API do usługi, 280
 operacje CRUD, 186
 oprogramowanie wysokiej jakości,
 352
 optymalizacja
 nadmierna, 303
 przedwczesna, 296
 ORM, Object-Relation Mapping, 219

P

Palindrom, 72
 planowanie, 185
 pobieranie
 prelegenta, 195, 239
 wielu prelegentów, 202
 podziały
 technologiczne, 124
 wielowarstwowe, 127

Postman, 223
 prawo Demeter, 306
 prelegent
 akcja pobierania, 248
 aktualizacja, 204
 dodawanie do bazy, 277
 komponent dla szczegółów, 257
 komponent listy, 240
 lista, 230
 pobieranie, 195, 202, 235, 239
 reduktor pobierania, 254
 sortowanie, 295
 szczegóły, 246
 tworzenie, 191
 usuwanie, 208
 priorytetyzacja, 120
 programistyczne kata, 47
 programowanie
 sterowane testami, TDD, 17
 sterowane zachowaniem, BDD, 20
 projekt
 Web API, 51
 zastany, 300
 pułapka dużego projektu, 353

R

React, 226
 komponent, 228
 tworzenie aplikacji, 226
 reduktor, 229, 239, 254
 Redux, 229
 refaktoryzacja, 24, 297
 bezpieczna, 308, 317
 niebezpieczna, 313
 rejestr produktu, 118, 119
 repozytorium, 161, 172, 186
 generyczne, 210, 215, 221
 InMemoryRepository, 215, 216,
 217, 218
 weryfikacja odwołań, 275
 rodzaje
 sobowtórów testowych, 86
 testów, 25
 rozbicie aplikacji, 114
 rozszerzanie wzorca repozytorium, 186
 rozwijanie aplikacji, 33, 353

S

SDK .NET Core, 39
 ServerFixture, 281
 serwer, 293
 Singleton, 78
 Sinon, 68
 imitacja odpowiedzi Ajaxa, 264
 konfiguracja biblioteki, 226
 sobowtór testowy, 79, 86
 sortowanie prelegentów, 295

Speaker Meet, 50

- lista prelegentów, 150
- szczegóły prelegentów, 165
- zmiany
 - po stronie interfejsu, 295
 - po stronie serwera, 293
 - w aplikacji, 293

stan globalny, 78

symulacja, 113

szew, 309

szpieg, 89

Ś

środowisko

- Node.js, 57
- testowe
 - w .NET, 39
 - w JavaScriptcie, 57

T

TDD, Test-Driven Development, 17

test

- Given15ThenFizzBuzz, 48
- Given1Then1, 49
- Given3ThenFizz, 47
- Given5ThenBuzz, 48

testowanie, 130

- błędy, 23
- czas, 21
- koszt, 22
- manualne, 23
- od początku do końca, 273
- od przodu do tyłu, 137
- od tyłu do przodu, 130
- od wewnątrz na zewnątrz, 144
- przypadków wyjątkowych, 154, 174
- Reduxa, 229
- trudności, 22
- wszystkich wyników, 311

TestServer, 280

testy

- akceptacyjne, 25
- akcji, 235
- akcji typu thunk, 237
- API, 151, 165
- aplikacji C#, 149
- aplikacji w JavaScriptcie, 225
- czyste, 160, 172
- funkcji, 191
- integracyjne, 25, 275
- jednostkowe, 25
 - Akcja, 26
 - Aranżacja, 26
 - Asercja, 27
 - usługi API, 230
- małe, 105
- metody
 - Create, 215
 - Get, 216

- GetAll, 217
- Update, 218
- naprawa, 264
- problemy z dodawaniem, 305
- ścieżek negatywnych, 109
- tworzenie, 310
- typu end-to-end, 26, 274
- usługi, 156, 169
- w C#, 31
- w JavaScriptcie, 34, 57
- złotego standardu, 311

TODO, 289

trzy prawa TDD, 19

tworzenie

- generycznego repozytorium, 210
- prelegenta, 191
- projektu, 44
- warstwy biznesowej, 132, 141

U

ulepszenia, 335, 337

usługa, 156, 169

- pobieranie danych z bazy, 278
- API, 230, 231, 246, 261
- imitacja, 231, 246
- implementacja, 261
- rzeczywista, 261
- testy jednostkowe, 230
- Gravatar, 178

usuwanie prelegenta, 208

utrzymanie rejestru produktu, 119

V

Visual Studio

- Code, 40, 63
- dodawanie rozszerzeń, 44
- tworzenie projektu, 44
- Community, 45

W

warstwa

- adaptera interfejsu użytkownika, 129
- biznesowa, 97, 129, 132, 141, 144
- danych, 185
- dostępu do danych, 128
- interfejsu, 129
- interfejsu użytkownika, 129
- prezentacji, 94
- usług, 128
- źródła danych, 129

Web API, 51

WebStorm, 64

weryfikacja odwołań

- API, 280
- repozytorium, 275

wprowadzanie

TDD, 353

ulepszeń, 335

wstrzykiwanie zależności, 78, 222

wtyczka

npm, 64

npm-intellisense, 64

wymagania, 27, 149, 285

techniczne, 125

zmienianie, 286

wyodrębnianie

aplikacji, 226

klasy, 309

DateTime, 84

metody, 309

oprogramowania, 79

wzorzec repozytorium, 128, 186

X

xUnit, 46

Z

zalety TDD, 351, 354

zależność

- od frameworka, 305
- od kodu zewnętrznego, 305

zasada

- jednej odpowiedzialności, 81, 307
- odwrócenia zależności, 83
- otwarte – zamknięte, 81, 302
- podstawienia Liskov, 82, 303
- segregacji interfejsów, 82

zasady SOLID, 80

zaślepka, 88

zdefiniowanie problemu, 117

zmiana wymagań, 286

Ż

źródło danych, 144

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 

TDD: tak pracują najlepsi programiści!

Coraz więcej profesjonalnych środowisk produkcyjnych opiera się na oprogramowaniu. Ewentualne błędy w pracy kodu mogą prowadzić do poważnych konsekwencji — dlatego od rozwiązań informatycznych wymaga się solidności i poprawności. Równocześnie oczekuje się wydajnego działania, skalowalności i podatności na modyfikacje, a także możliwości łatwego utrzymania kodu. Aplikacje utworzone zgodnie z paradygmatem TDD są w większym stopniu testowalne i zapewniają wysoki poziom poprawnej, stabilnej pracy. Sprawia to, że coraz więcej zespołów programistycznych skłania się ku TDD, mimo że zautomatyzowane testowanie bywa czasochłonne, pracochłonne i dość trudne w implementacji.

To książka przeznaczona dla tych, którzy chcą dogłębnie zrozumieć istotę TDD. Omówiono tu wszystkie aspekty TDD, włączając w to podstawy, dzięki którym średnio zaawansowany programista komfortowo rozpocznie budowę aplikacji zgodnie z tym paradygmatem. Przedstawiono zasady definiowania i testowania granic, a także pojęcie abstrahowania kodu ze-wnętrznego. W książce pojawiają się też — wprowadzane stopniowo — bardziej zaawansowane koncepcje, takie jak szpiendzy, imitacje i fałszywki. Pokazano w niej, w jaki sposób za pomocą TDD można przekształcić wymagania i historie użytkownika w funkcjonującą aplikację. Sporo miejsca poświęcono pisaniu różnych rodzajów testów, również integracyjnych. Poszczególne koncepcje zostały zilustrowane praktycznymi fragmentami kodu napisanego w C# i JavaScriptcie.

W tej książce między innymi:

- koncepcje programowania sterowanego testami i przygotowanie środowiska do pracy
- różne podejścia do budowania aplikacji i sterowania testami
- poprawa elastyczności aplikacji i jej podatności na przyszłe modyfikacje
- TDD w warunkach zmieniających się wymagań
- rozwiązywanie problemów z kodem zastanym

John Callaway jest programistą z tytułem Microsoft MVP. Specjalizuje się w technologiach WWW. Znakomicie zna praktycznie wszystkie ważne technologie, od PHP po C#, od ReactJS po SignalR. Ceni czysty kod i profesjonalizm. Chętnie dzieli się wiedzą.

Clayton Hunt jest profesjonalnym programistą. Zajmuje się głównie programowaniem dla internetu w językach JavaScript i C#. Jest sygnatariuszem manifestów głoszących idee *agile* i *software craftsmanship*.

Helion helion.pl 0 801 339900 0 601 339900 INFORMATYKA W NAJLEPSZYM WYDANIU	Sprawdź nasze szkolenia! SZKOLENIA AKADEMIA IT & BUSINESS WWW.SZKOLENIA.HELION.PL	KOD KORZYŚCI Sięgnij po więcej! ▶ ISBN 978-83-283-5653-5 9 788328 356535 Cena: 69,00 zł
--	--	---

Packt