John Callaway, Clayton Hunt

TDD z wykorzystaniem C# 7

Programowanie sterowane testami

Helion

Packt

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

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.

Spis treści

ROZDZIAŁ 4

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 JavaScripcie.

W tym rozdziale:

- Omówimy więcej praktyk związanych z TDD
- Przekażemy porady, jak uniknąć pułapek czyhających na nowych praktyków TDD
- Objaśnimy, 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 JavaScripcie 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 JavaScripcie nazywa się to małpim 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. ***S**ingle Responsibility*), zasady otwarte – zamknięte (ang. ***O**pen/Closed*), zasady podstawienia Liskov (ang. ***L**iskov Substitution*), zasady segregacji interfejsów (ang. ***I**nterface Segregation*) i zasady odwrócenia zależności (ang. ***D**ependency 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 modyfikacje". 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/babeb72287ad9490e1↪ebab84e7225ad6a9e5.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 oryginał.

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 zmienić bez ostrzeżenia.

Niemożność 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 połączonego. Chcemy mieć wiele kontraktów, aby zredukować liczbę daleko sięgających zależności.

Ilekroć 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. Mógłbyś 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 niepowodzeniem.

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 JavaScripcie.

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 JavaScripcie

Oto przykład klasy `DummyLogger` w JavaScripcie. 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) {
        }
    }
```

Zaślepki

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

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

Zaślepki powstają poprzez stworzenie kopii lub nadpisanie klasy bądź metody, która ma zwracać wartość zaślepioną, i ustawienie jej tak, aby tę wartość zwracała. Pamiętaj: zaślepki 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 JavaScripcie

Oto przykład funkcji `stubSpeakerReducer` w JavaScripcie. 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 JavaScripcie

Oto przykład funkcji `spySpeakerReducer` w JavaScripcie. 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 JavaScripcie

Oto przykład klasy `MockDateTimeService` w JavaScripcie. 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 sobowtórem. 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 JavaScripcie

Oto przykład klasy `FakeDataContext` w JavaScripcie:

```
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 wariancie 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 `Serach` 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 zbytnio zagłębiać w tematykę Moqa, ale pokażemy, w jaki sposób frameworki imitujące pomagają 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);
}
```

Koniecznie 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, przekaż `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 rozrastają się dość szybko i zanim się obejrzysz, 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 `GivenCaseInsensitveMatchThenSpeaker` ➥`InCollection`:

```
[Theory]
[InlineData("Janusz")]
[InlineData("janusz")]
[InlineData("JaNuSz")]
public void GivenCaseInsensitveMatchThenSpeakerInCollection(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 spróbowaliśmy 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

Ś

T

U

V

W

X

Z

Ź

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 zewnętrznego. W książce pojawiają się też — wprowadzane stopniowo — bardziej zaawansowane koncepcje, takie jak szpiedzy, 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 JavaScripcie.

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*.

ISBN 978-83-283-5653-5