ROY OSHEROVE

TESTY JEDNOSTKOWE

ŚWIAT NIEZAWODNYCH APLIKACJI

WYDANIE II

Poznaj możliwości
testów jednostkowych!

Helion

Tytuł oryginału: The Art of Unit Testing: With Examples in .NET, 2nd Edition

Tłumaczenie: Radosław Meryk

ISBN: 978-83-246-8774-9

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

Materiały graficzne na okładce zostały wykorzystane za zgodą Shutterstock Images LLC.

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

Drogi Czytelniku!
Jeżeli chcesz ocenić tę książkę, zajrzyj pod adres
http://helion.pl/user/opinie/tesjed
Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

Printed in Poland.

Spis treści

2 Pierwszy test jednostkowy

W tym rozdziale:

- Przegląd frameworków testów jednostkowych w .NET
- Piszemy pierwszy test za pomocą frameworka NUnit
- Korzystanie z atrybutów NUnit
- Trzy typy wyjścia jednostki pracy

Kiedy po raz pierwszy zacząłem pisać testy jednostkowe z wykorzystaniem prawdziwego frameworka do testów jednostkowych, nie było zbyt obszernej dokumentacji, a we frameworkach, z którymi pracowałem, nie było odpowiednich przykładów (w tamtych czasach kodowałem głównie z wykorzystaniem języków VB 5 i 6). Nauka posługiwania się nimi była wyzwaniem, dlatego zacząłem od pisania raczej słabych testów. Na szczęście czasy się zmieniły.

Ten rozdział pomoże czytelnikom rozpocząć pisanie testów, nawet jeśli nie mają pojęcia, od czego zacząć. Pomoże on znaleźć się na dobrej drodze do pisania rzeczywistych testów jednostkowych za pomocą frameworka NUnit — środowiska do tworzenia testów jednostkowych w .NET. Jest to mój ulubiony framework do tworzenia testów jednostkowych w .NET, ponieważ jest łatwy do posługiwania się, łatwy do zapamiętania i zawiera mnóstwo doskonałych własności.

Istnieją inne frameworki dla .NET, w tym takie, które mają więcej funkcji, ale zaczynam zawsze od frameworka NUnit. Jeśli jest taka potrzeba, to czasami posługuję się innymi frameworkami. Przyjrzymy się, jak działa środowisko NUnit, jakiej używa składni, jak je uruchomić i uzyskać informacje o tym, że test się powiódł bądź nie. Aby to osiągnąć, zaprezentuję niewielki projekt oprogramowania,

którego będziemy używać w całej książce do odkrywania technik testowania i najlepszych praktyk.

Czytelnik może odczuwać, że zmuszam go do używania frameworka NUnit w tej książce. Dlaczego nie używać wbudowanego w środowisku Visual Studio frameworka MSTest? Odpowiedź składa się z dwóch części:

- NUnit zawiera lepsze funkcje od MSTest w zakresie pisania testów jednostkowych i atrybutów testów. Dzięki temu możemy pisać testy bardziej czytelne i łatwiejsze w utrzymaniu.
- W Visual Studio 2012 wbudowany mechanizm uruchamiania testów pozwala na uruchamianie testów napisanych w innych frameworkach, w tym we frameworku NUnit. Aby to umożliwić, wystarczy zainstalować adapter testów NUnit dla Visual Studio — NuGet (środowisko NuGet omówiono w dalszej części tego rozdziału).

To sprawia, że wybór frameworka testów jednostkowych staje się dla mnie stosunkowo łatwy.

Po pierwsze, musimy przyjrzeć się temu, czym jest framework testów jednostkowych i co pozwala nam zrobić, czego nie moglibyśmy zrobić, gdybyśmy go nie stosowali.

2.1. Frameworki testów jednostkowych

Testy ręczne są kłopotliwe. Piszemy kod, uruchamiamy go w debugerze, wciskamy w aplikacji wszystkie potrzebne klawisze, aby uruchomić kod, który właśnie napisaliśmy, a następnie powtarzamy wszystko to po napisaniu nowego kodu. A do tego musimy pamiętać, żeby sprawdzić pozostały kod, na który mógł mieć wpływ nowy kod. To dodatkowa ręczna praca. Doskonale.

Wykonywanie testów i testowanie regresyjne całkowicie ręcznie, powtarzanie tych samych czynności w kółko jak małpa jest podatne na błędy i czasochłonne, a ludzie nie lubią tego robić — jest to najbardziej znienawidzona czynność w rozwoju oprogramowania. Problemy te można złagodzić dzięki zastosowaniu odpowiednich narzędzi. Frameworki do testów jednostkowych pozwalają pisać testy szybciej przy użyciu zestawu znanych API, wykonywać testy automatycznie i łatwo przeglądać wyniki tych testów. A do tego niczego nigdy nie zapominają! Spróbujmy przyjrzeć się bliżej, co mogą nam zaoferować.

2.1.1. Co oferują frameworki testów jednostkowych

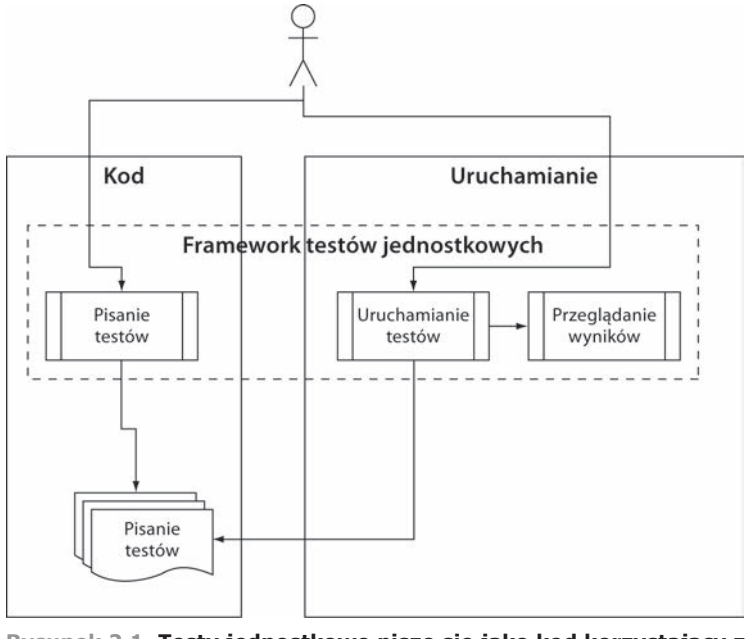
Do tej pory wielu czytelników tej książki wykonywało testy o ograniczonych możliwościach:

- *Nie były strukturalne*.
  Za każdym razem, gdy chcieliśmy przetestować jakąś własność, musieliśmy „odkrywać" koło na nowo. Jeden test mógł wyglądać jak aplikacja konsolowa, inny używał graficznego interfejsu użytkownika, a jeszcze

inny formularza webowego. Nie mieliśmy czasu na testowanie, a testy nie spełniały wymagania „łatwe w implementacji".

- *Nie były powtarzalne*.
  Ani autor testu, ani członkowie jego zespołu nie mogli uruchomić testów, które były napisane w przeszłości. To łamie zasadę „powtarzalności" i utrudnia znajdowanie błędów regresji. Dzięki stosowaniu frameworka można łatwiej i w zautomatyzowany sposób napisać testy, które są powtarzalne.
- *Nie pokrywały wszystkich ważnych części kodu*.
  Testy nie sprawdzały wszystkich istotnych części kodu. Oznacza to cały kod zawierający logikę, ponieważ każdy taki fragment może zawierać potencjalny błąd (gettery i settery właściwości nie liczą się jako logika, ale ostatecznie będą używane jako część pewnej jednostki pracy). Gdyby pisanie testów było łatwiejsze, bylibyśmy bardziej skłonni pisać ich więcej. Dzięki temu uzyskalibyśmy lepsze pokrycie.

Krótko mówiąc: to, czego nam brakowało, to framework do pisania, uruchamiania i przeglądania testów jednostkowych oraz ich wyników. Na rysunku 2.1 pokazano obszary wytwarzania oprogramowania, na które ma wpływ framework testów jednostkowych.



Rysunek 2.1. Testy jednostkowe pisze się jako kod korzystający z bibliotek frameworka testów jednostkowych. Następnie testy są uruchamiane za pomocą osobnego narzędzia do testów jednostkowych lub z poziomu IDE, a następnie przeglądane (w postaci tekstu wyjściowego, interfejsu IDE lub aplikacji frameworka testowania) przez programistę bądź zautomatyzowany proces kompilacji

Frameworki testów jednostkowych są bibliotekami kodu i modułami, które pomagają programistom uruchamiać testy jednostkowe dla swojego kodu zgodnie z tym, co przedstawiono w tabeli 2.1. Mają też drugą stronę — pozwalają na uruchomienie testów w ramach zautomatyzowanej kompilacji. Zagadnienie to opiszę w kolejnych rozdziałach.

Tabela 2.1. W jaki sposób frameworki testów jednostkowych pomagają programistom pisać i uruchamiać testy i przeglądać ich wyniki

Praktyka dotycząca testów jednostkowych	W jaki sposób framework pomaga?
Łatwe pisanie testów w usystematyzowany sposób.	Framework zapewnia programiście bibliotekę klas zawierającą następujące elementy: – klasy bazowe lub interfejsy do dziedziczenia; – atrybuty do umieszczenia w kodzie w celu oznaczenia testów; – klasy ASERCJI zawierające specjalne metody asercji, które wywołujemy w celu weryfikacji kodu.
Uruchamianie jednego lub wszystkich testów jednostkowych.	Framework dostarcza narzędzia do uruchamiania testów (programu konsolowego lub z interfejsem GUI), które: – identyfikuje testy w kodzie; – uruchamia testy automatycznie; – pokazuje status w czasie działania; – może być zautomatyzowane za pośrednictwem wiersza polecenia.
Przegląd wyników wykonanych testów.	Narzędzie do uruchamiania testów zazwyczaj dostarcza następujące informacje: – ile testów uruchomiono; – ile testów nie uruchomiło się; – ile testów się nie powiodło; – które testy się nie powiodły; – powody niepowodzenia testów; – komunikat ASSERT, który napisaliśmy; – miejsce w kodzie, dla którego test się nie powiódł; – jeśli to możliwe, pełny ślad stosu wyjątków, które spowodowały niepowodzenie testu; dzięki niemu dotrzemy do wywołań metod wewnątrz stosu wywołań.

W chwili pisania tej książki istniało ponad 150 frameworków testów jednostkowych — praktycznie jeden dla każdego języka programowania będącego w publicznym użytku. Obszerną listę można znaleźć pod adresem *http://en.wikipedia.org/wiki/List_of_unit_testing_frameworks*. Weźmy pod uwagę, że dla samego tylko środowiska .NET dostępne są co najmniej trzy różne frameworki testów jednostkowych: MS Test (firmy Microsoft), xUnit.NET oraz NUnit. Spośród nich framework NUnit był w przeszłości de facto standardem. Obecnie można zauważyć rywalizację pomiędzy zwolennikami MS Test i NUnit, występującą po prostu dlatego, że środowisko MS Test jest wbudowane w Visual Studio. Jednakże gdybym miał wybór, wybrałbym NUnit ze względu na niektóre funkcje, o których napiszę w dalszej części tego rozdziału, a także w dodatku poświęconym narzędziom i frameworkom.

UWAGA. Korzystanie z frameworka testów jednostkowych nie daje gwarancji, że testy, które piszemy, będą **czytelne**, **łatwe w utrzymaniu** lub **wiarygodne** albo że obejmą całość logiki, którą chcielibyśmy przetestować. W rozdziale 7. oraz w różnych innych miejscach w tej książce przyjrzymy się, jak zapewnić, aby nasze testy jednostkowe miały takie właściwości.

2.1.2. Frameworki xUnit

Łącznie frameworki do testów jednostkowych określane są nazwą **frameworków xUnit**, ponieważ ich nazwy zazwyczaj zaczynają się od pierwszych liter języka, dla którego zostały zbudowane. Istnieje framework CppUnit dla języka C++, JUnit dla Javy, NUnit dla .NET oraz HUnit dla języka Haskell. Nie dla wszystkich z nich przestrzega się tych zasad nazewnictwa, ale przeważnie się to robi.

W tej książce będziemy używać NUnit — frameworka testów jednostkowych dla środowiska .NET, które ułatwia pisanie testów, uruchamianie ich i analizę wyników. NUnit początkowo pojawił się jako bezpośredni port wszechobecnego JUnit dla Javy. Od tego czasu poczyniono ogromny postęp w jego konstrukcji i funkcjonalności. W efekcie NUnit oddzielił się od swojego rodzica i zaczął żyć własnym życiem w stale zmieniającym się ekosystemie frameworków do testów. Pojęcia, które będziemy omawiać, powinny być zrozumiałe dla programistów Javy i C++.

2.2. Wprowadzenie w tematykę projektu LogAn

Projekt, który wykorzystamy do testów w tej książce, początkowo będzie prosty. Będzie zawierał tylko jedną klasę. W kolejnych rozdziałach będziemy rozszerzać ten projekt o nowe klasy i funkcje. Projekt nazwiemy **LogAn** (skrót od *log and notification* — dosł. *dzienniki i powiadamianie*).

Oto scenariusz. Twoja firma ma wiele wewnętrznych produktów, których używa do monitorowania swoich aplikacji w siedzibie klientów. Wszystkie te produkty zapisują pliki dzienników i umieszczają je w specjalnym katalogu. Pliki dzienników są zapisywane we własnym formacie opracowanym przez Twoją firmę. Format ten nie może być parsowany za pomocą żadnych istniejących narzędzi zewnętrznych. Otrzymałeś zadanie stworzenia produktu LogAn, który będzie analizować te pliki dziennika w celu wyszukania w nich szczególnych przypadków i zdarzeń. Gdy znajdzie takie przypadki i zdarzenia, powinien zawiadomić zainteresowane strony.

W tej książce napiszemy testy weryfikujące możliwości systemu LogAn w zakresie parsowania, rozpoznawania zdarzeń i powiadamiania. Zanim jednak zaczniemy testowanie naszego projektu, przyjrzymy się, jak pisze się testy jednostkowe za pomocą frameworka NUnit. W pierwszym kroku należy go zainstalować.

2.3. Pierwsze kroki z NUnit

Tak jak w przypadku każdego nowego narzędzia najpierw trzeba framework zainstalować. Ponieważ NUnit jest programem typu *open source*, który można pobrać za darmo, to zadanie będzie raczej proste. Następnie przyjrzymy się, jak zacząć

pisanie testów z wykorzystaniem frameworka NUnit, używać różnych wbudowanych atrybutów dostarczanych przez NUnit oraz uruchamiać testy i uzyskiwać rzeczywiste wyniki.

2.3.1. Instalacja frameworka NUnit

Najlepszym i najprostszym sposobem, aby zainstalować NUnit, jest skorzystanie z NuGet — darmowego rozszerzenia do Visual Studio, które pozwala na wyszukiwanie, pobieranie i instalację referencji do popularnych bibliotek z poziomu Visual Studio za pomocą kilku kliknięć myszą i prostego tekstu polecenia.

Zalecam zainstalowanie rozszerzenia NuGet poprzez przejście do następującego menu w Visual Studio: *Tools/Extension Manager*, kliknięcie *Online Gallery* i zainstalowanie pierwszego na liście rozszerzenia *NuGet Package Manager*. Po instalacji należy zrestartować Visual Studio i voilà — mamy potężne i łatwe w obsłudze narzędzie do dodawania referencji i zarządzania nimi w projektach (programistom, którzy migrują ze środowiska Ruby, rozszerzenie NuGet może przypominać narzędzia Ruby Gems i GemFile, choć w dalszym ciągu będzie ono nowe, jeśli chodzi o kontrolę wersji oraz wdrażanie).

Po zainstalowaniu rozszerzenia NuGet możemy otworzyć następujące menu: *Tools/Library Package Manager/Package Manager Console*, a następnie w oknie tekstowym, które się wyświetli, wpisać polecenie: `Install-Package NUnit` (można również skorzystać z klawisza *Tab* w celu automatycznego uzupełnienia dostępnych poleceń oraz nazw pakietów bibliotek).

Po wykonaniu tych działań powinniśmy zobaczyć przyjemny komunikat `Nunit Installed Successfully`. Rozszerzenie NuGet pobrało archiwum zip zawierające pliki frameworka NUnit, dodało referencję do domyślnego projektu ustawionego w polu kombi okna konsoli menedżera pakietów oraz zakończyło pracę, wyświetlając komunikat informujący o tym, że wykonało te wszystkie działania. W naszym projekcie powinniśmy teraz zobaczyć referencję do biblioteki *NUnit.Framework.dll*.

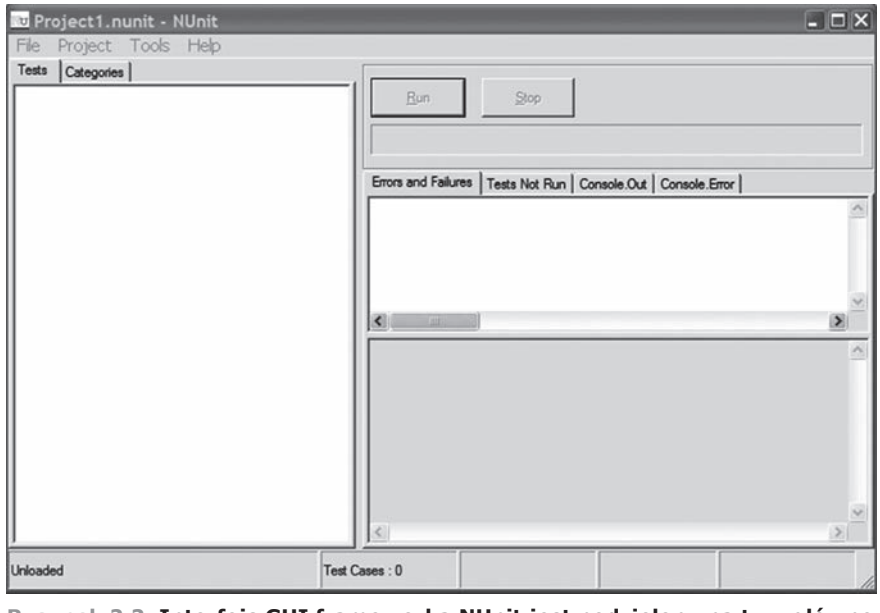
Krótka uwaga na temat interfejsu GUI frameworka NUnit: to podstawowe narzędzie uruchamiania testów frameworka NUnit. Narzędzie to omówię w dalszej części tego rozdziału, ale zazwyczaj go nie używam. Powinniśmy uznać je raczej za narzędzie edukacyjne pozwalające zrozumieć, jak działa NUnit jako osobne narzędzie, bez dodatków do Visual Studio. Narzędzie z interfejsem GUI nie jest również dołączone do wersji frameworka NUnit zainstalowanej za pomocą rozszerzenia NuGet. Rozszerzenie NuGet instaluje tylko potrzebne biblioteki DLL — bez interfejsu użytkownika (to ma pewien sens, ponieważ możemy mieć wiele projektów korzystających z NUnit, ale nie potrzebujemy wielu wersji interfejsu użytkownika frameworka, żeby je uruchomić). Aby pobrać interfejs użytkownika frameworka NUnit, który także pokażę nieco dokładniej w dalszej części tego rozdziału, można zainstalować pakiet *NUnit.Runners* za pomocą rozszerzenia NuGet albo można odwiedzić witrynę *Nunit.com* i zainstalować stamtąd pełną wersję. Ta pełna wersja jest także wbudowana w pakiet NUnit Console Runner, z którego korzystamy podczas uruchamiania testów na serwerze kompilacji.

Jeśli ktoś nie może pobrać rozszerzenia NuGet bądź nie ma do niego dostępu, może pobrać framework NUnit z witryny *www.NUnit.com* i ręcznie dodać do projektu referencje do bibliotek NUnit.

Jako bonus można wykorzystać fakt, że NUnit jest produktem *open source*. Można zatem pobrać kod źródłowy NUnit, samodzielnie go skompilować i swobodnie korzystać z kodu źródłowego w ramach licencji *open source* (ze szczegółami licencji można się zapoznać w pliku *license.txt* w katalogu z programem).

> **UWAGA.** Gdy powstawała ta książka, najnowszą wersją NUnit była wersja 2.6.0. Przykłady w tej książce powinny być zgodne z większością przyszłych wersji frameworka.

Jeśli wybraliśmy ręczny sposób instalacji NUnit, możemy uruchomić pobrany program *Setup*. Program instalacyjny umieści skrót do narzędzia z interfejsem GUI na pulpicie, ale główne pliki programu powinny być zapisane w katalogu o nazwie *C:\Program Files\NUnit-Net-2.6.0*. Dwukrotne kliknięcie ikony NUnit na pulpicie spowoduje uruchomienie narzędzia do wykonywania testów. Zrzut ekranu tego narzędzia pokazano na rysunku 2.2.



Rysunek 2.2. Interfejs GUI frameworka NUnit jest podzielony na trzy główne części: drzewo wyświetlające testy po lewej stronie, komunikaty i błędy w górnej prawej części oraz informacje śladu stosu w dolnej prawej części

2.3.2. Ładowanie rozwiązania

Jeśli pobrałeś kod źródłowy przykładów tej książki, załaduj w środowisku Visual Studio 2010 lub nowszym rozwiązanie *ArtOfUnitTesting2ndEd.Samples.sln* znajdujące się w folderze *Code*.

UWAGA. Do uruchamiania przykładów zamieszczonych w tej książce można korzystać z wersji Microsoft Visual C# 2010 Express (lub wersji nowszej).

Zaczniemy od przetestowania poniższej prostej klasy z jedną metodą (jednostką, którą testujemy):

```
public class LogAnalyzer
{
    public bool IsValidLogFileName(string fileName)
    {
        if(fileName.EndsWith(".SLF"))
        {
            return false;
    }
return true;
}
}
```

Zwróćmy uwagę, że celowo pominąłem znak ! przed warunkiem instrukcji `if`. Z tego powodu metoda ta ma błąd — zwraca `false` zamiast `true`, gdy plik ma rozszerzenie *SLF*. Zrobiłem to po to, abyśmy mogli zobaczyć, jak w programie do uruchamiania testów wygląda sytuacja, gdy test się nie powiedzie.

Powyższa metoda nie wydaje się skomplikowana; przetestujemy ją, aby upewnić się, że działa, ale głównie po to, by prześledzić procedurę testowania. W rzeczywistym świecie powinniśmy przetestować wszystkie metody, które zawierają logikę — nawet jeśli wydaje się ona być prosta. Logika może się nie powieść. Chcemy wiedzieć, kiedy tak się stanie. W kolejnych rozdziałach będziemy testować bardziej skomplikowane scenariusze i logiki.

Metoda sprawdza rozszerzenie pliku, aby ustalić, czy plik jest poprawnym plikiem dziennika, czy nie. Nasz pierwszy test będzie polegał na wysłaniu prawidłowej nazwy pliku i sprawdzeniu, czy metoda zwraca `true`.

Oto pierwsze kroki potrzebne do napisania automatycznego testu dla metody `IsValidLogFileName`:

1. Dodaj nowy projekt biblioteki klas do rozwiązania. Będzie ona zawierać nasze klasy testowe. Nadaj mu nazwę *LogAn.UnitTests* (przy założeniu, że testowany projekt ma nazwę *LogAn.csproj*).
2. Do tej biblioteki dodaj nową klasę, która będzie zawierała metody testowe. Nazwij ją `LogAnalyzerTests` (zakładając, że testowana klasa ma nazwę `LogAnalyzer`).
3. Dodaj nową metodę do powyższego przypadku testowego o nazwie `IsValidLogFileName_BadExtension_ReturnsFalse()`.

Więcej informacji na temat standardów nazewnictwa i układu testów podamy w dalszej części tej książki. Podstawowe zasady wymieniono w tabeli 2.2.

Na przykład dla naszego projektu LogAn projekt testowy będzie miał nazwę *LogAn.UnitTests*. Klasa testowa dla klasy `LogAnalyzer` będzie nosiła nazwę `LogAnalyzerTests`.

Tabela 2.2. Podstawowe reguły umieszczania testów i nadawania im nazw

Obiekt do przetestowania	Obiekt do stworzenia po stronie testującej
Projekt	Utwórz projekt testu o nazwie *[Testowany_ projekt].UnitTests*.
Klasa	Dla klasy będącej częścią projektu *Testowany_projekt* stwórz klasę o nazwie [*NazwaKlasy*]`Tests`.
Jednostka pracy (metoda lub logiczna grupa kilku metod albo kilku klas)	Dla każdej jednostki pracy stwórz metodę testu o następującej nazwie: [*NazwaJednostkiPracy*]_[*TestowanyScenariusz*]_[*OczekiwaneZachowanie*]. Nazwa jednostki pracy powinna odpowiadać nazwie metody (jeśli to jest cała jednostka pracy) lub być bardziej abstrakcyjna, jeśli jest to przypadek użycia obejmujący wiele metod lub klas, na przykład `LogowanieUżytkownika` lub `UsuńUżytkownika` albo `Inicjalizacja`. Bardziej komfortowe może być zaczęcie od nazwy metody i przejście później na nazwy bardziej abstrakcyjne. Jeśli są to nazwy metod, upewnij się, że są to nazwy publiczne oraz że nie reprezentują początku jednostki pracy.

Oto trzy części nazwy metody testowej:

- *NazwaJednostkiPracy* — nazwa metody, grupy metod bądź klas, które testujemy.
- *Scenariusz* — warunki, w jakich jednostka jest testowana, na przykład „zły login”, „nieprawidłowy użytkownik” albo „dobre hasło”. Na przykład możemy opisać parametry wysyłane do metody publicznej albo początkowy stan systemu, gdy jest wywoływana jednostka pracy, na przykład „brak pamięci w systemie”, „brak użytkowników” lub „użytkownik już istnieje”.
- *OczekiwaneZachowanie* — jakiego działania oczekujemy od testowanej metody w określonych warunkach. Może to być jedna z trzech możliwości: wynik jako zwracana wartość (rzeczywista wartość lub wyjątek), wynik jako zmiana stanu systemu (na przykład dodanie nowego użytkownika do systemu, dzięki czemu system będzie się zachowywał inaczej przy następnym logowaniu) lub wynik w postaci wywołania systemu zewnętrznego (na przykład zewnętrznej usługi sieciowej).

W naszym teście metody `IsValidLogFileName` scenariusz polega na tym, że wysyłamy do metody prawidłową nazwę pliku, a oczekiwanym działaniem jest zwrócenie przez metodę wartości `true`. Metodzie testu możemy zatem nadać nazwę `IsValidFileName_BadExtension_ReturnsFalse()`.

Czy testy powinniśmy pisać w projekcie kodu produkcyjnego? A może powinniśmy je wydzielić do innego projektu — specjalnie stworzonego dla testów? Zazwyczaj wolę oddzielić projekt testowy od kodu produkcyjnego, ponieważ to sprawia, że reszta prac związanych z testowaniem staje się łatwiejsza. Ponadto wiele osób nie chce dołączać testów do kodu produkcyjnego, co prowadzi do różnych brzydkich schematów kompilacji warunkowej lub innych złych pomysłów, które powodują, że kod staje się mniej czytelny.

Z drugiej strony, nie mam pod tym względem bardzo ścisłych wymagań. Podoba mi się również pomysł wykorzystywania testów obok działającej aplikacji

produkcyjnej, aby można było testować jej prawidłowe działanie już po zainstalowaniu. To wymaga uważnych przemyśleń, ale nie jest konieczne, aby testy i kod produkcyjny występowały w tym samym projekcie. Możemy mieć ciastko i jednocześnie możemy je zjeść.

Jeszcze nie używaliśmy frameworka NUnit, ale już jesteśmy blisko. Musimy jeszcze dodać referencję do projektu testującego w projekcie testowanym. Aby to zrobić, należy kliknąć prawym przyciskiem myszy w projekcie testowanym i wybrać polecenie *Add Reference*. Następnie należy wybrać zakładkę *Projects*, po czym wybrać projekt *LogAn*.

Następną rzeczą, jakiej powinniśmy się nauczyć, jest sposób oznaczenia metody, która ma być załadowana i uruchomiona przez framework NUnit automatycznie. Najpierw powinniśmy się upewnić, czy została dodana referencja do frameworka NUnit albo za pomocą rozszerzenia NuGet, albo ręcznie, zgodnie z opisem w punkcie 2.3.1.

2.3.3. Wykorzystanie atrybutów NUnit w kodzie

Framework NUnit korzysta z systemu atrybutów w celu rozpoznawania i ładowania testów. Podobnie jak zakładki w książce, te atrybuty pomagają frameworkowi zidentyfikować ważne części w zestawie, który go ładuje, oraz części będące testami do wywołania.

Framework NUnit dysponuje zestawem, który zawiera te specjalne atrybuty. Wystarczy tylko dodać referencję do projektu testowego (nie w kodzie produkcyjnym!) do zestawu *NUnit.Framework*. Można go znaleźć w zakładce *.NET* w oknie dialogowym *Add Reference* (nie trzeba tego robić, jeśli użyliśmy rozszerzenia NuGet do zainstalowania frameworka NUnit). Jeśli wpiszemy *NUnit*, wyświetli się kilka zestawów o nazwie zaczynającej się od tej części.

Należy dodać *nunit.framework.dll* jako referencję do projektu testowego (jeśli zainstalowaliśmy framework ręcznie, a nie za pośrednictwem rozszerzenia NuGet).

Narzędzie do uruchamiania testów NUnit potrzebuje co najmniej dwóch atrybutów do tego, aby wiedzieć, co należy uruchomić:

- `[TestFixture]` — atrybut `[TestFixture]` oznacza klasę, która zawiera zautomatyzowane testy NUnit (gdybyśmy zastąpili słowo „Fixture" słowem „Class", atrybut ten miałby znacznie więcej sensu. Ale taka zamiana jest dobra tylko jako ćwiczenie wykonywane w celu zapamiętania znaczenia atrybutu. Gdybyśmy dosłownie zmienili atrybut w ten sposób, kod się nie skompiluje). Atrybut `[TestFixture]` należy umieścić na początku nowej klasy `LogAnalyzerTests`.
- `[Test]` — atrybut `[Test]` można umieścić w metodzie w celu oznaczenia jej jako zautomatyzowanego testu do wywołania. Ten atrybut należy umieścić w nowej metodzie testowej.

Po zakończeniu pracy kod testu powinien wyglądać następująco:

```
[TestFixture]
public class LogAnalyzerTests
{
    [Test]
    public void IsValidFileName_BadExtension_ReturnsFalse()
    {

    }
}
```

WSKAZÓWKA. Framework NUnit w najbardziej podstawowej konfiguracji wymaga, aby metody testów były publiczne, zwracały `void` i nie przyjmowały żadnych parametrów, ale jak się przekonamy, czasami testy mogą również pobierać parametry!

W tym momencie oznaczyliśmy klasę i metodę do uruchomienia. Teraz framework NUnit wywoła każdy kod, który umieścimy wewnątrz metody testowej, kiedy tylko tego zażądamy.

2.4. Piszemy pierwszy test

W jaki sposób testujemy kod? Test jednostkowy zazwyczaj obejmuje trzy główne działania:

1. *Konfigurację* obiektów — utworzenie ich i ustawienie według potrzeb.
2. *Wykonanie operacji* na obiektach.
3. *Asercję* oczekiwanego rezultatu.

Oto prosty fragment kodu, który wykonuje te wszystkie trzy działania. Za asercję jest odpowiedzialna klasa `Assert` frameworka NUnit:

```
[Test]
public void IsValidFileName_BadExtension_ReturnsFalse()
{
    LogAnalyzer analyzer = new LogAnalyzer();

    bool result = analyzer.IsValidLogFileName("plikozłymrozszerzeniu.foo");

    Assert.False(result);
}
```

Zanim przejdziemy dalej, należy napisać trochę więcej o klasie `Assert`, ponieważ spełnia ona ważną rolę podczas pisania testów jednostkowych.

2.4.1. Klasa Assert

Klasa `Assert` zawiera metody statyczne i znajduje się w przestrzeni nazw `NUnit.Framework`. Jest mostem pomiędzy testowanym kodem a frameworkiem NUnit, a jej celem jest deklaracja, że określone założenie jest prawdziwe. Jeśli okaże się, że argumenty przekazane do klasy `Assert` będą inne niż założone, framework NUnit rozpozna, że test się nie powiódł, i powiadomi nas o tym. Opcjonalnie można

poinformować klasę `Assert`, jaki komunikat ostrzegawczy powinien się wyświetlić, jeśli asercja się nie powiedzie.

Klasa `Assert` zawiera wiele metod. Najważniejsza z nich to `Assert.True(jakieś_wyrażenie_logiczne)`, która sprawdza warunek `Boolean`. Ale istnieje wiele innych metod, które można uznać za cukier składniowy, a które sprawiają, że asercje pewnych elementów stają się bardziej czytelne (jedną z nich jest metoda `Assert.False`, której użyliśmy).

Oto metoda, która sprawdza, czy oczekiwany obiekt bądź wartość jest taka sama jak wartość rzeczywista:

```
Assert.AreEqual(oczekiwany_obiekt, rzeczywisty_obiekt, komunikat);
```

Oto przykład:

```
Assert.AreEqual(2, 1+1, "Nieprawidłowe działanie arytmetyczne");
```

Poniższa metoda sprawdza, czy dwa argumenty odwołują się do tego samego obiektu:

```
Assert.AreSame(oczekiwany_obiekt, rzeczywisty_obiekt, komunikat);
```

Oto przykład:

```
Assert.AreSame(int.Parse("1"),int.Parse("1"),
"ten test nie powinien się powieść").
```

Klasa `Assert` jest prosta do nauki, posługiwania się i zapamiętania.

Należy również zwrócić uwagę, że wszystkie metody klasy `Assert` przyjmują ostatni parametr typu `string`, który zostanie wyświetlony obok wyjścia generowanego przez framework w przypadku, gdy test się nie powiedzie. Nigdy nie należy używać tego parametru (jego użycie zawsze jest opcjonalne). Należy zadbać o to, by nazwa testu wyjaśniała, co ma się zdarzyć. Często programiści piszą trywialnie oczywiste rzeczy typu „test nie powiódł się" lub „oczekiwano x zamiast y", co framework sam dostarcza. Podobnie jak w przypadku komentarzy w kodzie — jeśli już musimy używać tego parametru, to nazwa metody powinna być czytelniejsza.

Po omówieniu podstaw interfejsu API spróbujmy uruchomić test.

2.4.2. Uruchomienie pierwszego testu za pomocą frameworka NUnit

Nadszedł czas, aby uruchomić pierwszy test i sprawdzić, czy powiedzie się, czy nie.

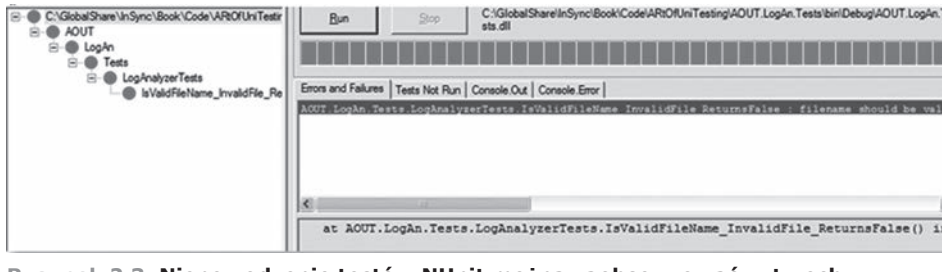
Istnieją co najmniej cztery różne sposoby uruchomienia testu:

- Za pomocą interfejsu GUI frameworka NUnit.
- Za pomocą narzędzia do uruchamiania testów środowiska Visual Studio 2012 z zainstalowanym rozszerzeniem do uruchamiania testów NUnit (w galerii NuGet występuje ono pod nazwą NUnit Test Adapter).
- Za pomocą narzędzia do uruchamiania testów programu ReSharper (dobrze znanej komercyjnej wtyczki do środowiska VS).
- Za pomocą narzędzia TestDriven.NET (innej znanej wtyczki do środowiska VS).

Chociaż w tej książce opisano tylko interfejs GUI frameworka NUnit, osobiście korzystam z narzędzia NCrunch, które jest szybkie, działa automatycznie, ale trzeba za nie zapłacić (to narzędzie razem z innymi opisano w dodatku poświęconym narzędziom). NCrunch dostarcza prostych, szybkich informacji wewnątrz okna edytora w środowisku Visual Studio. Uważam, że to narzędzie do uruchamiania testów tworzy idealną parę z technikami TDD. Więcej informacji na jego temat można znaleźć pod adresem *http://www.ncrunch.net/*.

Aby uruchomić test za pomocą interfejsu GUI frameworka NUnit, trzeba mieć skompilowany zestaw (w tym przypadku plik *.dll*), który można przekazać frameworkowi NUnit do sprawdzenia. Po skompilowaniu projektu należy znaleźć ścieżkę do pliku zestawu, który powstał po kompilacji.

Następnie należy załadować GUI frameworka NUnit (jeśli zainstalowaliśmy framework NUnit ręcznie, wystarczy znaleźć ikonę na pulpicie. Jeśli zainstalowaliśmy środowisko *NUnit.Runners* za pomocą rozszerzenia NuGet, należy odszukać plik wykonywalny środowiska GUI frameworka NUnit w folderze *packages* w głównym folderze rozwiązania, a następnie wybrać polecenie *File/Open*. Następnie wprowadzamy nazwę zestawu do testowania. Po lewej stronie okna wyświetli się pojedynczy test oraz hierarchia klas i przestrzeni nazw projektu (rysunek 2.3). Aby uruchomić testy, należy kliknąć przycisk *Run*. Testy są automatycznie pogrupowane według nazw (nazwy zestawu, typu), więc można wybrać i uruchomić tylko określone typy bądź przestrzenie nazw (zwykle uruchamiamy wszystkie testy, aby uzyskać pełniejsze informacje na temat awarii).



Rysunek 2.3. Niepowodzenie testów NUnit można zaobserwować w trzech miejscach: hierarchia testów po lewej stronie i pasek postępu u góry mają czerwony kolor, a po prawej stronie wyświetlają się komunikaty o błędach

Jak można zobaczyć, test nie wychodzi, co może sugerować, że jest błąd w kodzie. Nadszedł czas, żeby naprawić kod i przekonać się, że test wyjdzie. Modyfikujemy kod poprzez dodanie brakującego znaku `!` w instrukcji `if`, tak aby przyjęła następującą postać:

```
if(!fileName.EndsWith(".SLF"))
{
    return false;
}
```

2.4.3. Dodanie testów pozytywnych

Pokazaliśmy, że złe rozszerzenia spowodują takie oznaczenia, ale kto powiedział, że dobre rozszerzenie zostanie „zatwierdzone" przez naszą niewielką metodę? Gdybyśmy stosowali technikę TDD, brakujące testy byłyby oczywiste, ale ponieważ piszemy testy po kodzie, musimy wymyślić dobre przypadki testowe, które pokryją wszystkie ścieżki. Na listingu 2.1 dodano kilka nowych testów, aby zobaczyć, co się będzie działo, gdy przekażemy plik z dobrym rozszerzeniem. Jeden z testów sprawdza rozszerzenie złożone z wielkich liter, natomiast drugi z małych.

Listing 2.1. Logika walidacji nazw plików programu LogAnalyzer, którą chcemy przetestować

```
[Test] public void
    IsValidLogFileName_GoodExtensionLowercase_ReturnsTrue()
    {
        LogAnalyzer analyzer = new LogAnalyzer();
        bool result = analyzer
            .IsValidLogFileName("filewithgoodextension.slf");

    Assert.True(result);
}

[Test]public void IsValidLogFileName_GoodExtensionUppercase_ReturnsTrue()
    {
        LogAnalyzer analyzer = new LogAnalyzer();

        bool result =
            analyzer
            .IsValidLogFileName("filewithgoodextension.SLF");
    Assert.True(result);
}
```

Jeśli teraz skompilujemy rozwiązanie, zauważymy, że środowisko GUI frameworka NUnit potrafi wykryć, że zestaw się zmienił, i automatycznie ponownie załaduje zestaw w GUI. Kiedy ponownie uruchomimy testy, zauważymy, że test z rozszerzeniem złożonym z małych liter nie wychodzi. Aby test wyszedł, musimy zmodyfikować kod produkcyjny — zastosować dopasowywanie ciągów bez rozróżniania wielkości liter:

```
public bool IsValidLogFileName(string fileName)
{
    if (!fileName.EndsWith(".SLF",
                           StringComparison.CurrentCultureIgnoreCase))
                           {
                           return false;
                           }
    return true;
}
```

Jeśli ponownie uruchomimy testy, wszystkie powinny wyjść. W środowisku GUI frameworka NUnit powinien ponownie wyświetlić się piękny zielony pasek.

2.4.4. Od czerwonego do zielonego: dążenie do spełnienia testów

Środowisko GUI frameworka NUnit zostało zbudowane zgodnie z prostą zasadą: aby uzyskać „zielone" światło pozwalające przejść dalej, wszystkie testy muszą być spełnione. Jeśli chociaż jeden z testów nie powiedzie się, zobaczymy czerwone światło na górnym pasku postępu. Informuje nas ono, że coś niedobrego dzieje się z systemem (lub z testami).

Zasada przechodzenia od czerwonego do zielonego jest powszechna w całym świecie testów jednostkowych, a zwłaszcza w metodyce wytwarzania oprogramowania bazującej na testach (TDD). Jej mantra to czerwone-zielone-refaktoryzacja, co oznacza, że zaczynamy od testu, który nie wychodzi, następnie dążymy do jego spełnienia, a w dalszej kolejności staramy się, by kod stał się bardziej czytelny i łatwiejszy w utrzymaniu.

Testy mogą również nie powieść się, jeśli nagle zostanie zgłoszony nieoczekiwany wyjątek. Test, który zatrzyma się z powodu nieoczekiwanego wyjątku, jest uznawany za nieudany w większości (jeśli nie we wszystkich) frameworków testowych. Jest to część podejścia — czasami w programie są błędy w postaci wyjątków, których się nie spodziewaliśmy.

Jeśli chodzi o wyjątki, w dalszej części tego rozdziału pokażemy rodzaj testu, który jako specyficznego wyniku lub zachowania oczekuje zgłoszenia wyjątku z określonego kodu. Tego rodzaju test nie wychodzi, jeśli wyjątek *nie zostanie* zgłoszony.

2.4.5. Styl kodu testów

Zwróćmy uwagę, że testy, które napisałem, mają kilka cech dotyczących stylu i czytelności, które wyróżniają kod testów od „standardowego" kodu. Nazwy testów mogą być bardzo długie, ale znaki podkreślenia pomagają zadbać o to, abyśmy nie zapomnieli umieścić wszystkich ważnych informacji. Dodatkowo w każdym teście umieściłem pusty wiersz pomiędzy etapami konfiguracji, działania i asercji. Uważam, że to pomaga mi znacznie szybciej czytać testy i ułatwia znajdowanie w nich problemów.

Starałem się również jak najbardziej oddzielić kod asercji od kodu działania. Preferuję asercję na wartości, a nie bezpośrednio na wywołaniu funkcji. Dzięki temu kod staje się znacznie bardziej czytelny.

Czytelność jest jednym z najważniejszych aspektów pisania testu. Test powinien czytać bez wysiłku nawet ktoś, kto nigdy nie widział testu wcześniej, bez potrzeby zadawania zbyt wielu pytań, lub w ogóle żadnych pytań. Więcej na ten temat napiszę w rozdziale 7. Teraz zobaczmy, czy możemy wyeliminować trochę powtórzeń i poprawić spójność testów bez szkody dla ich czytelności.

2.5. Refaktoryzacja w kierunku testów z parametrami

Wszystkie testy, które napisaliśmy do tej pory, mają pewien problem związany z łatwością utrzymania. Wyobraźmy sobie, że teraz chcemy dodać parametr do konstruktora klasy `LogAnalyzer`. Będziemy więc mieli trzy testy, które się nie skompilują.

Poprawienie trzech testów może nie jest trudne, ale z łatwością w systemie może być 30 lub 100 testów. W realnym świecie programiści mają ważniejsze rzeczy do roboty niż ściganie kompilatora za to, co powinno być prostą zmianą. Jeśli testy spowodują, że nie wyjdzie nam sprint, nie będziemy chcieli ich uruchamiać, a nawet możemy chcieć usunąć tego rodzaju irytujące testy.

Spróbujmy zatem tak zrefaktoryzować testy, abyśmy nigdy nie natknęli się na tego rodzaju problem.

Framework NUnit ma doskonałą własność, która może nam w tym bardzo pomóc. Mowa o testach z parametrami. Aby z nich korzystać, weź jedną z istniejących metod testowych, które wyglądają dokładnie tak samo jak inne, i wykonaj poniższe czynności:

1. Zastąp atrybut `[Test]` atrybutem `[TestCase]`.
2. Wyodrębnij wszystkie zakodowane na sztywno wartości używane przez test do parametrów metody testowej.
3. Przenieść wartości, których używałeś wcześniej, do nawiasów atrybutu `[TestCase(param1,param2,..)]`.
4. Zmień nazwę tej metody na postać bardziej uniwersalną.
5. Dodaj atrybut `[TestCase(...)]` w tej samej metodzie testowej dla każdego z testów, które chcesz „scalić" z tą metodą testową, używając innych wartości testowych.
6. Usuń pozostałe testy, tak aby została tylko jedna metoda testowa zawierająca wiele atrybutów `[TestCase]`.

Te czynności należy wykonać krok po kroku. Po wykonaniu czwartej nasz ostatni test będzie miał następującą postać:

```
[TestCase("filewithgoodextension.SLF")]
public void
IsValidLogFileName_ValidExtensions_ReturnsTrue(string file)
              {
                  LogAnalyzer analyzer = new LogAnalyzer();
                  bool result = analyzer.IsValidLogFileName(file);
                  Assert.True(result);
              }
```

Atrybut TestCase wysyła parametr do metody

Parametr, do którego atrybuty przypadków testowych mogą dołączyć wartość

Parametr ten jest stosowany w sposób generyczny

Parametr przesyłany do atrybutu `TestCase` jest odwzorowywany w czasie działania programu przez narzędzie do uruchamiania testów na pierwszy parametr metody testowej. Do metody testowej oraz do atrybutu `TestCase` można dodać tyle parametrów, ile się chce.

A oto ciekawostka: można dodać wiele **atrybutów** `TestCase` w tej samej metodzie testowej. Zatem po wykonaniu kroku szóstego nasz test będzie miał następującą postać:

```
[TestCase("filewithgoodextension.SLF")]
    [TestCase("filewithgoodextension.slf")]
    public void
    IsValidLogFileName_ValidExtensions_ReturnsTrue(string file)
    {
        LogAnalyzer analyzer = new LogAnalyzer();
```

Inny atrybut oznacza inny test z inną wartością dołączoną do parametru metody

```
        bool result = analyzer.IsValidLogFileName(file);
        Assert.True(result);
    }
```

Teraz możemy śmiało usunąć poprzednią metodę testową, która używała rozszerzenia w postaci małych liter, ponieważ obejmuje ją atrybut przypadku testowego w naszej bieżącej metodzie testowej. Jeśli uruchomimy testy ponownie, zobaczymy, że wciąż mamy tę samą liczbę testów, ale kod jest łatwiejszy w utrzymaniu i bardziej czytelny.

Możemy także pójść o krok dalej i uwzględnić w bieżącej metodzie testowej test negatywny (który zakłada, że otrzymamy wartość `false`). Poniżej pokażę, jak to zrobić, ale ostrzegam, że w ten sposób prawdopodobnie otrzymamy mniej czytelną metodę testową, ponieważ nazwa będzie musiała stać się jeszcze *bardziej ogólna*. Rozważmy to demo składni, biorąc pod uwagę, że stosując tę technikę, możemy posunąć się zbyt daleko, a testy — bez głębszej analizy kodu — staną się mniej zrozumiałe.

Oto w jaki sposób można zrefaktoryzować wszystkie testy w tej klasie poprzez dodanie kolejnego parametru do przypadku testowego i metody testowej oraz przez zmianę asercji na `Assert.AreEqual`.

```
    [TestCase("filewithgoodextension.SLF",true)]
    [TestCase("filewithgoodextension.slf",true)]
    [TestCase("filewithbadextension.foo",false)]
    public void
IsValidLogFileName_VariousExtensions_ChecksThem(string file,
                                                bool expected)
    {
        LogAnalyzer analyzer = new LogAnalyzer();
        bool result = analyzer.IsValidLogFileName(file);
        Assert.AreEqual(expected,result);
    }
```

Dodanie kolejnego parametru do przypadku testowego

Otrzymanie drugiego parametru przypadku testowego

Użycie wartości drugiego parametru

Dzięki tej jednej metodzie testowej możemy pozbyć się wszystkich innych metod testowych w tej klasie. Należy jednak zauważyć, że nazwa testu stała się tak ogólna, że trudno rozpoznać różnicę pomiędzy testem spełnionym a niespełnionym. Ta informacja musi w sposób oczywisty wynikać z przesłanych wartości parametrów, dlatego powinny być one jak najprostsze i jak najbardziej oczywiste do tego, by udowodnić to, co chcemy udowodnić. Więcej informacji na temat takiego spojrzenia na czytelność zaprezentujemy w rozdziale 7.

Pod względem łatwości utrzymania zauważmy, że mamy teraz tylko jedno wywołanie do konstruktora. Tak jest lepiej, ale takie rozwiązanie nie jest wystarczająco dobre, ponieważ nie możemy doprowadzić do tego, aby wszystkie nasze testy sprowadzały się do jednej olbrzymiej, sparametryzowanej metody testowej. Więcej informacji na temat łatwości utrzymania kodu znajduje się w dalszej części tej książki (tak, w rozdziale 7.).

Kolejną techniką refaktoryzacji, którą możemy zastosować w tym momencie, jest zmiana postaci instrukcji `if` w kodzie produkcyjnym. Możemy sprowadzić ją do pojedynczej instrukcji `return`. Jeśli ktoś lubi tego rodzaju przekształcenia, to teraz jest dobry moment na taką refaktoryzację. Ja nie należę do takich osób. Akceptuję

nieco rozwlekłości w kodzie, tak aby czytelnik kodu nie musiał zbyt wiele o nim myśleć. Lubię kod, który nie jest zbyt zawiły dla jego własnego dobra, a instrukcje `return` zawierające instrukcje warunkowe kierują mnie w złą stronę. Ale jak pamiętacie, to nie jest książka o projektowaniu. Róbcie tak, jak uważacie za słuszne. Przede wszystkim będę się odnosił do książki „Czysty kod" Roberta Martina (wujka Boba).

2.6. Więcej atrybutów NUnit

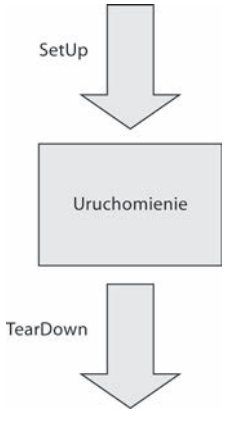
Teraz gdy zobaczyliśmy, jak łatwo jest tworzyć testy jednostkowe, które są uruchamiane automatycznie, przyjrzymy się, jak skonfigurować stan początkowy dla każdego testu i jak usunąć śmieci, który pozostawiły nasze testy.

Test jednostkowy ma określone punkty w swoim cyklu życia, nad którymi chcielibyśmy mieć kontrolę. Uruchamianie testów to tylko jeden z tych punktów. Jak dowiesz się w następnym punkcie, istnieją specjalne metody konfiguracji, które są uruchamiane przed uruchomieniem każdego z testów.

2.6.1. Atrybuty Setup i TearDown

W przypadku testów jednostkowych ważne jest, aby wszelkie pozostałe dane lub egzemplarze z poprzednich testów były zniszczone oraz by odtworzono stan dla nowych testów w taki sposób, jakby nigdy wcześniej nie uruchamiano żadnych testów. Jeśli pozostawimy stan z poprzedniego testu, może się okazać, że test nie powiedzie się, ale tylko wtedy, gdy został uruchomiony po innym teście, natomiast innym razem przejdzie pomyślnie. Wyszukiwanie tego rodzaju błędów zależności pomiędzy testami jest trudne i czasochłonne. Nie polecam tego nikomu. Tworzenie testów, które są całkowicie od siebie niezależne, to jedna z najlepszych praktyk, które będę omawiał w drugiej części tej książki.

Framework NUnit zawiera specjalne atrybuty pozwalające na większą kontrolę nad ustawianiem i czyszczeniem stanu przed i po wykonaniu testów. Są to atrybuty akcji `[SetUp]` i `[TearDown]`. Na rysunku 2.4 zaprezentowano proces uruchamiania testu z wykorzystaniem akcji `SetUp` (dosł. konfiguracja) i `TearDown` (dosł. rozbiórka).



Rysunek 2.4. Framework NUnit wykonuje akcje „konfigurowania" i „rozbiórki" odpowiednio przed i po każdej metodzie testowej

Na razie upewnij się, że każdy test, który piszemy, wykorzystuje nowy egzemplarz testowanej klasy, tak aby pozostały stan nie przeszkadzał w kolejnych testach.

Możemy kontrolować to, co będzie się działo w etapach konfigurowania i rozbiórki, za pomocą dwóch atrybutów NUnit:

- `[SetUp]` — ten atrybut można umieścić wewnątrz metody, podobnie jak atrybut `[Test]`. Powoduje on, że framework NUnit uruchamia metodę konfiguracji za każdym razem, gdy zostaną uruchomione dowolne testy wewnątrz klasy.
- `[TearDown]`— ten atrybut oznacza metodę, która zostanie wykonana raz po uruchomieniu każdego testu wewnątrz klasy.

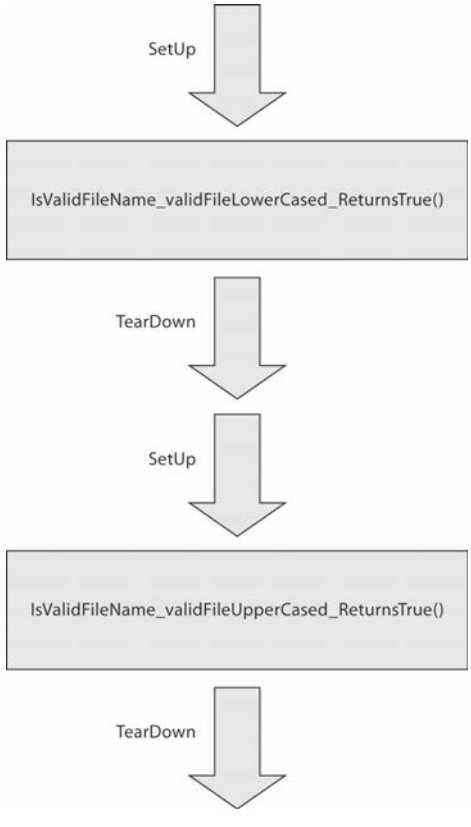
Na listingu 2.2 pokazano sposób, w jaki można skorzystać z atrybutów `[SetUp]` i `[TearDown]`, aby upewnić się, że każdy test otrzymuje nowy egzemplarz klasy `LogAnalyzer`. Sposób ten eliminuje niepotrzebne pisanie.

Listing 2.2. Wykorzystanie atrybutów [SetUp] i [TearDown]

```
using NUnit.Framework;
[TestFixture] public class LogAnalyzerTests
{
    private LogAnalyzer m_analyzer=null;
    [SetUp]                          <-- Atrybut SetUp
    public void Setup()
    {
        m_analyzer = new LogAnalyzer();
    }
    [Test]
    public void IsValidFileName_validFileLowerCased_ReturnsTrue()
    {
        bool result = m_analyzer.IsValidLogFileName("whatever.slf");

       Assert.IsTrue(result, "plik powinien mieć prawidłową nazwę!");
    }

    [Test]
    public void IsValidFileName_validFileUpperCased_ReturnsTrue()
    {
        bool result = m_analyzer
                    .IsValidLogFileName("whatever.SLF");

        Assert.IsTrue(result, "plik powinien mieć prawidłową nazwę!");
    }

    [TearDown]              <-- Atrybut TearDown
    public void TearDown()
    {
        //Wiersz poniżej zamieszczono jako przykład antywzorca.
        //Nie jest potrzebny. Nie należy umieszczać go w kodzie produkcyjnym.
        m_analyzer = null;      <-- Popularny antywzorzec — nie trzeba tego robić
    }
}
```

Należy jednak pamiętać o tym, że im częściej będziemy korzystać z atrybutu `[SetUp]`, tym mniej czytelne będą nasze testy, ponieważ aby zrozumieć, w jaki sposób egzemplarze trafiają do testu oraz jaki jest typ każdego obiektu wykorzystywanego przez test, czytelnicy kodu będą musieli czytać go w dwóch miejscach w pliku. Zawsze powtarzam swoim studentom: „wyobraźcie sobie, że czytelnik testu nigdy wcześniej nas nie spotkał i nigdy nie spotka. Czyta nasz test w 2 lata po tym, jak przestaliśmy pracować w firmie. Wszystko, co zrobimy, aby pomóc mu zrozumieć nasz kod bez potrzeby zadawania pytań, jest bardzo pomocne. Prawdopodobnie nie ma on nikogo, komu może zadać te pytania, więc jesteśmy jego jedyną nadzieją". Zmuszanie czytelnika do tego, aby nieustannie przeskakiwali pomiędzy dwoma regionami kodu, nie jest dobrym pomysłem.

Metody konfiguracji i rozbiórki można porównać do konstruktorów i destruktorów dla testów wewnątrz klasy. W każdej klasie testowej może występować tylko po jednej z każdej z nich i każda będzie wykonana raz dla każdego testu w klasie. Na listingu 2.1 są dwa testy jednostkowe, dlatego ścieżka wykonania dla frameworka NUnit będzie podobna do pokazanej na rysunku 2.5.



Rysunek 2.5. W jaki sposób framework NUnit wywołuje metody SetUp i TearDown z wieloma testami jednostkowymi w tej samej klasie: każdy test jest poprzedzony uruchomieniem metody SetUp, a po zakończeniu każdego testu uruchamiana jest metoda TearDown

Chciałbym również podkreślić, że w realnym życiu NIE korzystam z metod konfiguracji do inicjalizacji egzemplarzy klas. Pokazałem je tylko po to, żeby czytelni-

cy wiedzieli, że takie istnieją, i aby ich unikali. Początkowo stosowanie ich może wydawać się dobrym pomysłem, ale szybko okazuje się, że testy poniżej metody `SetUp` stają się mniej czytelne. Zamiast tego używam metod-fabryk do inicjalizacji testowanych egzemplarzy. Można o tym przeczytać w rozdziale 7.

Framework NUnit zawiera kilka innych atrybutów, które pomagają w konfiguracji oraz porządkowaniu stanu. Na przykład atrybuty `[TestFixtureSetUp]` i `[TestFixtureTearDown]` umożliwiają skonfigurowanie stanu raz przed wszystkimi testami dla konkretnej klasy oraz raz po zakończeniu działania wszystkich testów. Atrybuty te przydają się w przypadku, gdy konfigurowanie lub czyszczenie stanu zajmuje dużo czasu i chcemy wykonywać je tylko raz na zestaw testów. Korzystanie z tych atrybutów wymaga zachowania ostrożności. W przypadku braku zachowania ostrożności może dojść do współdzielenia stanów pomiędzy testami.

W projektach testów jednostkowych należy UNIKAĆ stosowania metod `TearDown` lub `TestFixture`. Jeśli je stosujemy, to zachodzi ryzyko, że piszemy test integracyjny, w którym korzystamy z systemu plików lub bazy danych, i musimy uporządkować dysk lub bazę danych po wykonaniu testów. Jedynym przypadkiem, gdy warto użyć metody `TearDown` w testach jednostkowych, jest sytuacja, kiedy pomiędzy testami trzeba „zresetować" stan zmiennej statycznej lub singletona w pamięci. W każdym innym przypadku zachodzi obawa, że wykonujemy testy integracyjne. Nie ma niczego złego w wykonywaniu testów integracyjnych, ale należy to robić w oddzielnym projekcie dedykowanym dla testów integracyjnych.

W następnym punkcie pokażę, w jaki sposób można przetestować zgłoszenie wyjątku przez kod tam, gdzie kod powinien zgłosić wyjątek.

2.6.2. Testowanie występowania oczekiwanych wyjątków

Jednym z popularnych scenariuszy testowania jest upewnienie się, że testowana metoda zgłosi wyjątek w tym miejscu, w którym powinna go zgłosić.

Załóżmy, że metoda powinna zgłosić wyjątek `ArgumentException` w przypadku przesłania pustej nazwy pliku. Jeśli kod nie zgłasza wyjątku, to znaczy, że test nie powiódł się. Logikę metody przetestujemy na listingu 2.3.

Listing 2.3. Logika walidacji nazw plików programu LogAnalyzer, którą chcemy przetestować

```
public class LogAnalyzer
    {
        public bool IsValidLogFileName(string fileName)
       {
            …
            if (string.IsNullOrEmpty(fileName))
            {
                   throw new ArgumentException(
                             "należy podać nazwę pliku");
            }
            …
      }
}
```

Są dwa sposoby, aby to sprawdzić. Zacznijmy od tego, którego nie należy używać, ale który jest bardzo powszechny, ponieważ kilka lat temu był jedynym interfejsem API pozwalającym na osiągnięcie tego celu. Framework NUnit zawiera specjalny atrybut umożliwiający testowanie wyjątków: jest to atrybut `[ExpectedException]`. Oto jak może wyglądać test, który sprawdza występowanie wyjątku:

```
[Test]
[ExpectedException(typeof(ArgumentException),
  ExpectedMessage ="należy podać nazwę pliku")]
public void IsValidFileName_EmptyFileName_ThrowsException()
{
    m_analyzer.IsValidLogFileName(string.Empty);
}
private LogAnalyzer MakeAnalyzer()
{
    return new LogAnalyzer();
}
```

W powyższym kodzie należy zwrócić uwagę na kilka istotnych elementów:

- Oczekiwany komunikat wyjątku został podany jako parametr atrybutu `[ExpectedException]`.
- W samym teście nie ma wywołania `Assert`. Asercję zawiera atrybut `[ExpectedException]`.
- Pobieranie wartości `Boolean` wyniku metody nie ma sensu, ponieważ oczekuje się, że wywołanie metody zgłosi wyjątek.

Niezwiązane z tym przykładem jest wyodrębnienie do metody-fabryki kodu, który tworzy egzemplarz klasy `LogAnalyzer`. Tej metody-fabryki użyję we wszystkich moich testach, aby poprawić łatwość konserwacji konstruktora bez konieczności poprawiania zbyt wielu testów.

Biorąc pod uwagę metodę z listingu 2.2 oraz test dla tej metody, ten test powinien się powieść. Gdyby metoda nie zgłosiła wyjątku `ArgumentException` lub gdyby komunikat wyjątku był inny niż oczekiwany, test nie powiódłby się. W tym przypadku otrzymalibyśmy informację, że albo wyjątek nie został zgłoszony, albo komunikat był inny, niż oczekiwano.

Dlaczego więc napisałem, że nie należy korzystać z tego sposobu? Ponieważ ten atrybut w zasadzie poleca silnikowi testów opakowanie całej metody w obszerny blok `try-catch`. Niepowodzenie testu następuje, gdy żaden wyjątek nie został przechwycony. Największy problem z tym sposobem polega na tym, że nie wiemy, *który* wiersz zgłosił wyjątek. W rzeczywistości mogło się zdarzyć, że w konstruktorze występuje błąd, który zgłasza wyjątek, a nasz test przechodzi, nawet jeśli konstruktor nigdy nie powinien zgłaszać tego wyjątku! W przypadku używania tego atrybutu test może nas okłamywać, dlatego staram się z niego nie korzystać.

Zamiast tego framework NUnit dostarcza nowego API: `Assert.Throws<T>(delegat)`. Oto przepisana wersja testu, która wykorzystuje metodę `Assert.Throws`:

```
    [Test]                                    <-- Nie jest potrzebny atrybut ExpectedException
public void IsValidFileName_EmptyFileName_Throws()
{
```

```
        LogAnalyzer la = MakeAnalyzer();
    var ex =
        Assert.Throws<Exception>(() => la.IsValidLogFileName(""));
StringAssert.Contains("należy podać nazwę pliku",
                      ex.Message);
}
```

Wykorzystanie metody Assert.Throws

Wykorzystanie obiektu Exception zwróconego przez metodę Assert.Throws

Powyższy kod zawiera wiele zmian w porównaniu z poprzednią wersją:

- Nie używamy już atrybutu `[ExpectedException]`.
- Wykorzystujemy metodę `Assert.Throws` oraz wyrażenie lambda bez argumentów. Treścią tego wyrażenia lambda jest wywołanie `la.IsValidLogFileName("")`.
- Jeśli kod wewnątrz wyrażenia lambda zgłosi wyjątek, test się powiedzie. Jeśli wyjątek zgłosi dowolny inny wiersz poza wyrażeniem lambda, test się nie powiedzie.
- `Assert.Throws` jest funkcją zwracającą egzemplarz obiektu wyjątku, który został zgłoszony wewnątrz wyrażenia lambda. Pozwala to nam na późniejszą asercję dotyczącą komunikatu obiektu wyjątku.
- Skorzystaliśmy z klasy `StringAssert` będącej częścią frameworka NUnit, której do tej pory nie omawialiśmy. Klasa ta zawiera przydatne metody pomocnicze, dzięki którym testowanie ciągów znaków staje się prostsze i bardziej czytelne.
- Nie zakładamy pełnej równości ciągów za pomocą funkcji `Assert.AreEqual`. Zamiast tego korzystamy z funkcji `StringAssert.Contains` — oznaczającej, że komunikat tekstowy **zawiera** ciąg znaków, którego szukamy. Dzięki temu testy są łatwiejsze w utrzymaniu, ponieważ ciągi często się zmieniają w miarę dodawania nowych funkcji. Ciągi znaków są rodzajem interfejsu użytkownika, dlatego mogą w nich występować nadmiarowe znaki przejścia do nowego wiersza, dodatkowe informacje, które nas nie interesują, itp. Gdybyśmy założyli, że cały ciąg znaków jest równy konkretnemu ciągowi, którego oczekujemy, musielibyśmy poprawiać test każdorazowo po dodaniu nowej „własności" na początku bądź końcu komunikatu, nawet gdyby dodana część nie miała wpływu na test (na przykład dodatkowe wiersze lub elementy formatowania użytkownika).

Istnieje mniejsze ryzyko, że ten test nas „okłamie". Polecam używanie funkcji `Assert.Throws` zamiast atrybutu `[ExpectedException]`.

Istnieją inne sposoby używania interfejsu *fluent* frameworka NUnit do sprawdzania komunikatów wyjątków. Właściwie niezbyt je lubię, ale stosowanie ich jest bardziej kwestią stylu. O składni *fluent* frameworka NUnit oraz sposobach sprawdzania komunikatów wyjątków można przeczytać w witrynie *NUnit.com*.

2.6.3. Ignorowanie testów

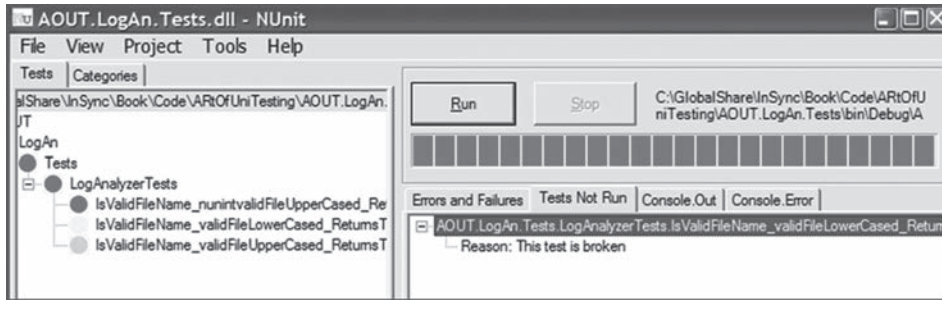
Czasami mamy testy, które nie są prawidłowe, a pomimo tego musimy sprawdzić kod w głównym drzewie kodu źródłowego. W tych rzadkich przypadkach (powinny

być naprawdę rzadkie!) możemy umieścić atrybut [Ignore] wewnątrz testów, które nie działają, ze względu na problem w teście, a nie w kodzie.

Użycie tego atrybutu może mieć następującą postać:

```
[Test]
[Ignore("Z tym testem jest problem")]
public void IsValidFileName_ValidFile_ReturnsTrue()
{
/// ...
}
```

Uruchomienie tego testu w interfejsie GUI narzędzia NUnit generuje efekt pokazany na rysunku 2.6.



Rysunek 2.6. We frameworku NUnit zignorowane testy są oznaczone żółtym kolorem (środkowy test), a powód rezygnacji z uruchomienia testu wyświetla się z prawej strony, w zakładce Tests Not Run

Co się dzieje, gdy chcemy uruchomić testy nie według przestrzeni nazw, ale według innego rodzaju grupowania? Do tego służą kategorie testowania. Omówię je w punkcie 2.6.5.

2.6.4. Składnia fluent frameworka NUnit

Framework NUnit ma również bardziej płynną (ang. *fluent*) składnię, którą można wykorzystywać zamiast wywoływania prostych metod `Assert.*`. Składnia *fluent* zawsze zaczyna się od `Assert.That(...)`. Oto ostatni test przepisany w taki sposób, by korzystał ze składni *fluent* frameworka NUnit:

```
[Test]
public void IsValidFileName_EmptyFileName_ThrowsFluent()
{
    LogAnalyzer la = MakeAnalyzer();

    var ex =
    Assert.Throws<ArgumentException>(() =>
                  la.IsValidLogFileName(""));
    Assert.That(ex.Message,
                Is.StringContaining("należy podać nazwę pliku"));
}
```

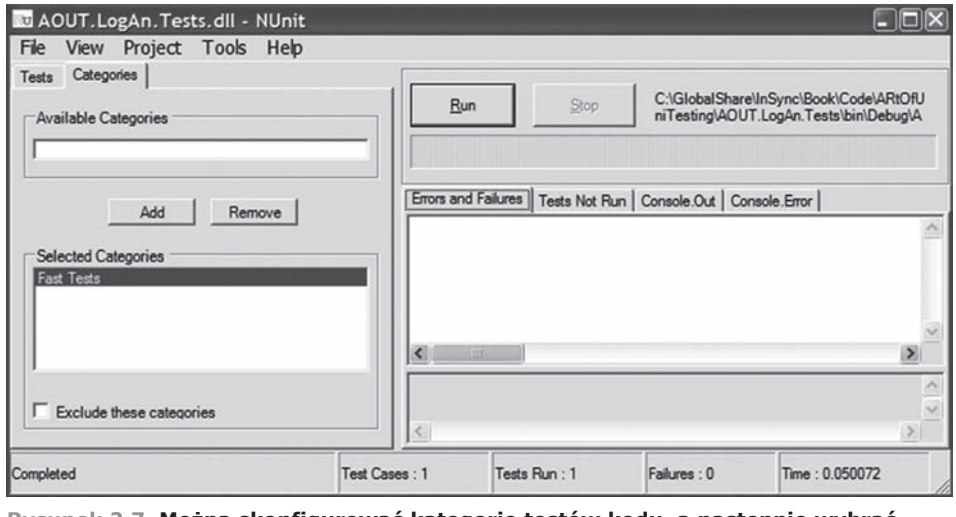
Osobiście wolę bardziej lakoniczną, prostszą i krótszą składnię `Assert.coś()` niż `Assert.That`. Chociaż składnia *fluent* na pierwszy rzut oka wydaje się bardziej przyjazna, zrozumienie, co testujemy, zajmuje więcej czasu (trzeba przeczytać wszystko aż do końca wiersza). Należy wybrać składnię zgodnie z własnymi upodobaniami. Należy tylko zadbać o spójność w całym projekcie testowym, ponieważ brak spójności prowadzi do wielu problemów z czytelnością.

2.6.5. Ustawianie kategorii testowych

Można tak ustawić testy, aby działały w określonych kategoriach testowych, na przykład testy wolne i testy szybkie. Do tego celu służy atrybut `[Category]` frameworka NUnit:

```
[Test]
[Category("Szybkie testy")]
public void IsValidFileName_ValidFile_ReturnsTrue()
{
/// ...
}
```

Po załadowaniu zestawu testowego w NUnit można zobaczyć, że testy są zorganizowane według kategorii, a nie według przestrzeni nazw. Aby uruchomić testy określonej kategorii, przełącz się do zakładki *Categories* w NUnit i dwukrotnie kliknij kategorię, którą chcesz uruchomić, tak aby przesunęła się do dolnej ramki *Selected Categories*. Następnie kliknij przycisk *Run*. Na rysunku 2.7 pokazano, jak może wyglądać ekran po wybraniu zakładki *Categories*.



Rysunek 2.7. Można skonfigurować kategorie testów kodu, a następnie wybrać określoną kategorię z poziomu interfejsu użytkownika frameworka NUnit

Jak dotąd uruchamialiśmy proste testy metod, które zwracały w wyniku jakąś wartość. Co zrobić, jeśli metoda nie zwraca wartości, ale zmienia stan obiektu?

2.7. Testowanie wyników metod, które nie zwracają wartości, tylko zmieniają stan systemu

Do tego punktu pokazywaliśmy, w jaki sposób testować pierwszy i najprostszy rodzaj rezultatu jednostki pracy: zwracane wartości (opisane w rozdziale 1.). W tym i w następnym rozdziale opiszemy również drugi rodzaj rezultatu: zmianę stanu systemu — sprawdzenie, czy zachowanie systemu jest inne po wykonaniu działania na testowanym systemie.

> **DEFINICJA. Testowanie przejść pomiędzy stanami** (nazywane także **weryfikacją stanów**) określa, czy testowana metoda zadziałała poprawnie, poprzez sprawdzenie zmienionego zachowania badanego systemu i jego współpracowników (zależności) po wykonaniu metody.

Jeśli system działa dokładnie tak samo jak przedtem, to w rzeczywistości nie zmienił swojego stanu albo wystąpił błąd.

Jeśli czytelnik zetknął się z definicjami testowania przejść pomiędzy stanami w innym miejscu, z pewnością zauważył, że zdefiniowałem je inaczej. To dlatego, że spojrzałem na ten obszar pod nieco innym kątem — tzn. łatwości utrzymania testu. Samo testowanie bezpośredniego stanu (co czasami sprowadza się do zapewnienia sprawdzalności) jest czymś, czego zwykle nie popieram, bo prowadzi ono do kodu trudniejszego w utrzymaniu i mniej czytelnego.

Rozważmy prosty przykład testowania przejść pomiędzy stanami dla klasy `LogAnalyzer`. Nie możemy jej przetestować, po prostu wywołując jedną metodę. Na listingu 2.4 pokazano kod dla tej klasy. W tym przypadku wprowadziliśmy nową właściwość `WasLastFileNameValid`, która powinna przechowywać ostatni stan sukcesu metody `IsValidLogFileName`. Zaprezentowałem najpierw kod, ponieważ nie próbuję w tym miejscu uczyć czytelników technik TDD, ale sposobów pisania dobrych testów. Dzięki stosowaniu technik TDD testy *mogłyby* stać się lepsze. Tutaj jednak pokazujemy sytuację, w której wiemy, jak należy pisać testy *po* napisaniu kodu.

Listing 2.4. Testowanie wartości właściwości poprzez wywołanie metody IsValidLogFileName

```
public class LogAnalyzer
{
    public bool WasLastFileNameValid { get; set; }

    public bool IsValidLogFileName(string fileName)
    {
        WasLastFileNameValid = false;          <-- Zmiana stanu systemu

        if (string.IsNullOrEmpty(fileName))
        {
            throw new ArgumentException("należy podać nazwę pliku");
        }
        if (!fileName.EndsWith(".SLF",
                StringComparison.CurrentCultureIgnoreCase))
        {
```

```
            return false;
        }

        WasLastFileNameValid = true;          ← Zmiana stanu systemu
        return true;
    }
}
```

Jak można zobaczyć na podstawie tego kodu, klasa `LogAnalyzer` pamięta, jaki był ostatni wynik sprawdzenia poprawności nazwy pliku. Ponieważ metoda `WasLastFileNameValid` zależy od wcześniejszego wywołania innej metody, nie możemy po prostu testować tej funkcjonalności poprzez napisanie testu, który pobiera z metody zwracany wynik. Aby sprawdzić, czy logika jest właściwa, trzeba użyć innego sposobu.

Najpierw należy zdefiniować, czym jest jednostka pracy, którą testujemy. Czy jest nią nowa właściwość o nazwie `WasLastFileNameValid`? Częściowo tak. Testowany kod znajduje się także w metodzie `IsValidLogFileName`, dlatego test powinien zaczynać się od nazwy tej metody. Jest to bowiem jednostka pracy, którą wywołujemy publicznie, aby zmienić stan systemu. Prosty test sprawdzający, czy zapamiętano wynik, pokazano na listingu 2.5.

Listing 2.5. Testowanie klasy przez wywołanie metody i sprawdzenie wartości właściwości

```
[Test]
        public void
        IsValidFileName_WhenCalled_ChangesWasLastFileNameValid()
        {
            LogAnalyzer la = MakeAnalyzer();

            la.IsValidLogFileName("badname.foo");

            Assert.False(la.WasLastFileNameValid);   ← Asercja dotycząca stanu systemu
        }
```

Zauważmy, że testujemy funkcjonalność metody `IsValidLogFileName` poprzez asercję w odniesieniu do kodu w innym miejscu w porównaniu z tym, gdzie występuje testowany kod.

Oto zrefaktoryzowany przykład, w którym dodano inny test dotyczący odwrotnego oczekiwania co do wartości stanu systemu:

```
[TestCase("badfile.foo", false)]
[TestCase("goodfile.slf", true)]
public void
IsValidFileName_WhenCalled_ChangesWasLastFileNameValid(string file,
                                                       bool expected)
{
    LogAnalyzer la = MakeAnalyzer();

    la.IsValidLogFileName(file);

    Assert.AreEqual(expected, la.WasLastFileNameValid);
}
```

Na listingu 2.6 pokazano inny przykład. W tym przykładzie sprawdzono funkcjonalność prostego kalkulatora.

Listing 2.6. Metody Add() i Sum()

```
public class MemCalculator
    {
        private int sum=0;

        public void Add(int number)
        {
        sum+=number;
        }

        public int Sum()
        {
            int temp = sum;
            sum = 0;
        return temp;
    }
}
```

Klasa `MemCalculator` działa w sposób zbliżony do kalkulatora kieszonkowego. Klikamy liczbę, następnie klikamy przycisk *Add*, następnie klikamy inną liczbę, potem ponownie klikamy *Add* itd. Po zakończeniu obliczeń klikamy *Equals* i otrzymujemy sumę dodanych dotąd składników.

Gdzie zacząć testowanie funkcji `Sum()`? Zawsze należy zaczynać od najprostszych testów — na przykład sprawdzenia, czy funkcja `Sum()` domyślnie zwraca `0`. Pokazano to na listingu 2.7.

Listing 2.7. Najprostszy test funkcji Sum() kalkulatora

```
[Test]
public void Sum_ByDefault_ReturnsZero()
{
        MemCalculator calc = new MemCalculator();

        int lastSum = calc.Sum();

        Assert.AreEqual(0,lastSum);        <-- Asercja domyślnie zwracanej wartości
}
```

Zwróćmy także uwagę na znaczenie użytej w tym miejscu nazwy metody. Możemy ją czytać tak jak zdanie.

Oto prosta lista konwencji nazewnictwa scenariuszy, z których korzystam w takich przypadkach:

- Przyrostek `ByDefault` można zastosować, gdy istnieje wartość oczekiwana zwracanej wartości bez wcześniejszego działania — tak jak pokazano w powyższym przykładzie.
- Przyrostki `WhenCalled` lub `Always` można wykorzystać w przypadku rezultatów jednostki pracy drugiego bądź trzeciego rodzaju (zmiana stanu lub wywołanie funkcji zewnętrznej) — gdy nastąpi zmiana stanu

bez wcześniejszej konfiguracji lub kiedy nastąpi wywołanie funkcji zewnętrznej bez wcześniejszej konfiguracji. Na przykład `Sum_WhenCalled_CallsTheLogger` lub `Sum_Always_CallsTheLogger`.

Nie możemy napisać żadnego innego testu bez wcześniejszego wywołania metody `Add()`, zatem następna metoda będzie wywoływała funkcję `Add()` i zakładała liczbę, jaką powinna zwrócić funkcja `Sum()`. Klasę testową razem z tym nowym testem pokazano na listingu 2.8.

Listing 2.8. Dwa testy. Drugi wywołuje metodę Add()

```
    [Test]
public void Sum_ByDefault_ReturnsZero()
{
    MemCalculator calc = MakeCalc();

    int lastSum = calc.Sum();

    Assert.AreEqual(0, lastSum);
}
[Test]
public void Add_WhenCalled_ChangesSum()
{
    MemCalculator calc = MakeCalc();

    calc.Add(1);
    int sum = calc.Sum();
    Assert.AreEqual(1, sum);
}

private static MemCalculator MakeCalc()
{
    return new MemCalculator();
}
```

Zachowanie systemu i stan zmienią się, jeśli w tym teście suma zwróci inną liczbę

Zwróćmy uwagę, że tym razem użyliśmy metody-fabryki do zainicjowania obiektu klasy `MemCalculator`. To jest dobry pomysł, ponieważ oszczędza czas na pisanie testów, sprawia, że kod wewnątrz każdego testu jest trochę mniejszy i nieco bardziej czytelny, oraz zapewnia zainicjowanie obiektu klasy `MemCalculator` zawsze w taki sam sposób. Gwarantuje również łatwiejsze utrzymanie testu, ponieważ jeśli konstruktor klasy `MemCalculator` zmieni się, wystarczy zmienić kod inicjalizacji w jednym miejscu, zamiast zmieniać wywołanie `new` we wszystkich testach.

Do tej pory wszystko powinno być jasne. Ale co się zdarzy, gdy metoda, którą testujemy, zależy od zasobów zewnętrznych, takich jak system plików, baza danych, usługa sieciowa lub cokolwiek innego, co jest dla nas trudne do kontrolowania? I w jaki sposób testujemy trzeci typ „rezultatu" jednostki pracy — wywołanie zewnętrzne? W takich przypadkach tworzy się namiastki testów, sztuczne obiekty i obiekty-makiety. Zostaną one omówione w kilku następnych rozdziałach.

2.8. Podsumowanie

W tym rozdziale analizowaliśmy wykorzystywanie frameworka NUnit w celu pisania prostych testów dla prostego kodu. Wykorzystaliśmy atrybuty `[TestCase]`, `[SetUp]` i `[TearDown]`, aby uzyskać pewność, że testy będą zawsze wykorzystywać nowy, niezmodyfikowany stan. Skorzystaliśmy z metody-fabryki, aby poprawić łatwość utrzymania testów. Używaliśmy także atrybutu `[Ignore]` w celu pominięcia testów, które powinny zostać poprawione. Kategorie testów pomagają pogrupować testy w sposób logiczny, a nie według klasy i przestrzeni nazw, natomiast dzięki wywołaniu `Assert.Throws()` kod zgłasza wyjątki tam, gdzie powinien. Na koniec przyjrzeliśmy się sytuacji, w której nie mamy do czynienia z prostą metodą, która zwraca wartość, ale musimy przetestować końcowy stan obiektu.

To jednak nie wszystko. W większości przypadków kod testu musi obsługiwać znacznie trudniejsze sytuacje.

W następnych kilku rozdziałach zaprezentujemy kilka podstawowych narzędzi do pisania testów jednostkowych. Spośród tych narzędzi trzeba będzie wybierać w przypadku pisania testów dla różnych trudnych sytuacji, na które możemy się natknąć.

Warto pamiętać o następujących regułach:

- Powszechną praktyką jest stosowanie jednej klasy testu dla jednej klasy testowanej, jednego projektu testu jednostkowego na jeden testowany projekt (oprócz projektu testów integracyjnych dla tego testowanego projektu) i co najmniej jednej metody testowej na jednostkę pracy (może to być zaledwie jedna metoda lub nawet kilka klas).
- Testom należy nadawać czytelne nazwy, stosując następujący wzorzec: `[JednostkaPracy]_[Scenariusz]_[OczekiwaneZachowanie]`.
- Należy stosować metody-fabryki w celu wielokrotnego wykorzystania kodu w testach, na przykład w kodzie do tworzenia i inicjowania obiektów wykorzystywanym przez wszystkie testy.
- Nie należy używać atrybutów `[SetUp]` i `[TearDown]`, o ile można tego uniknąć. Ich stosowanie sprawia, że testy stają się mniej zrozumiałe.

W następnym rozdziale przyjrzymy się bardziej rzeczywistym sytuacjom, w których testowany kod będzie bardziej realistyczny w porównaniu z tym, co widzieliśmy do tej pory. Będziemy rozwiązywali problemy zależności i zapewnienia sprawdzalności. Zaczniemy również omawiać sztuczne obiekty, makiety i namiastki oraz sposoby wykorzystywania tych mechanizmów do pisania testów.

Skorowidz

C

D

E

F

G

H

I

J

K

L

Ł

M

N

O

P

Q

R

S

Ś

T

U

V

W

X

Z

TESTY JEDNOSTKOWE
Świat niezawodnych aplikacji

WYDANIE II

System informatyczny to inteligentne połączenie modułów i zależności, otoczone setkami tysięcy, a nawet milionami linii kodu źródłowego. Zmiana w jednym obszarze może mieć fatalny wpływ na działanie systemu w zupełnie innym miejscu. Ta zależność powoduje, że koszty wprowadzenia nawet najdrobniejszej zmiany w oprogramowaniu są ogromne. Czy istnieje rozwiązanie tego problemu? Jak stworzyć system, w którym błyskawiczna weryfikacja lub wprowadzona zmiana nie spowodują nowych błędów w innej części? Oczywiście, że można to zrobić! Odpowiedzią na te i wiele innych problemów są testy automatyczne.

Ten przewodnik to doskonała okazja, by głębiej poznać temat testów jednostkowych. Jeżeli uważasz, że ich pisanie jest uciążliwe, czasochłonne, trudne, lub po prostu nie wiesz, jak je tworzyć, ta książka rozwiąże wszystkie Twoje problemy! W trakcie lektury dowiesz się, jak pisać testy, tworzyć zestawy testowe oraz przygotowywać makiety i namiastki. Poznasz narzędzia: Moq, FakeItEasy oraz Typemock Isolator. Ponadto zdobędziesz wiedzę na temat organizacji testów oraz strategii testowania kodu odziedziczonego. To pozycja obowiązkowa dla wszystkich programistów C# szukających świetnego przewodnika po świecie testów jednostkowych!

Dzięki tej książce:

- **rozwiejesz swoje wątpliwości dotyczące testów**
- **poznasz najpopularniejsze narzędzia wspomagające testowanie**
- **zorganizujesz swoje testy jednostkowe**
- **zapoznasz się z kluczowymi elementami dobrych testów**
- **stworzysz niezawodny i tani w utrzymaniu kod**

Niezawodny kod jest w Twoim zasięgu!

sięgnij po WIĘCEJ

KOD KORZYŚCI

helion.pl
księgarnia internetowa

Nr katalogowy: 20678

Księgarnia internetowa:
http://helion.pl

Zamówienia telefoniczne:
0 801 339900
0 601 339900

Helion

Sprawdź najnowsze promocje:
http://helion.pl/promocje
Książki najchętniej czytane:
http://helion.pl/bestsellery
Zamów informacje o nowościach:
http://helion.pl/nowosci

Helion SA
ul. Kościuszki 1c, 44-100 Gliwice
tel.: 32 230 98 63
e-mail: helion@helion.pl
http://helion.pl

ISBN: 978-83-246-8774-9

9 788324 687749

Cena: 57,00 zł

Informatyka w najlepszym wydaniu