Helion

Test-Driven Development w C# i .NET

Tworzenie wysokiej jakości kodu w architekturze DDD za pomocą znanych narzędzi i bibliotek

Adam Tibi

Tytuł oryginału: Pragmatic Test-Driven Development in C# and .NET:
Write loosely coupled, documented, and high-quality code
with DDD using familiar tools and libraries

Tłumaczenie: Robert Górczyński

ISBN: 978-83-289-1693-7

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

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

Printed in Poland.

Spis treści

Część II. Stworzenie aplikacji z zastosowaniem podejścia w stylu TDD

Część III. Zastosowanie programowania sterowanego testami we własnych projektach

Rozpoczęcie pracy z testami jednostkowymi

Rozdział 3

Testy jednostkowe to podstawa podejścia określanego mianem programowania sterowanego testami (ang. *test-driven development*, TDD) i jednocześnie warunek wstępny do jego praktycznego zastosowania w pracy. W tym rozdziale chciałbym pokrótce przedstawić minimalną ilość niezbędnej teorii oraz skoncentrować się na omówieniu narzędzi i technik, dzięki którym będzie można w codziennej pracy stosować testy jednostkowe.

Wyjaśnię więc, jak tworzyć testy jednostkowe zapewniające możliwość realizacji rozwiązań programistycznych o średnim poziomie złożoności. W części II książki wykorzystamy tę wiedzę jako punkt wyjścia do zdobycia kolejnych umiejętności i ich praktycznego zastosowania.

W poprzednim rozdziale zajęliśmy się tworzeniem **aplikacji dostarczającej prognozę pogody** (ang. *weather forecasting application*, **WFA**) i przystosowaliśmy ją do używania mechanizmu **wstrzykiwania zależności** (ang. *dependency injection*, **DI**). W tym użyjemy wymienionej aplikacji jako punktu wyjścia do poznania tematu testów jednostkowych. Jeżeli nie masz doświadczenia w pracy z mechanizmem wstrzykiwania zależności i kontenerami wstrzykiwania zależności, wówczas gorąco zachęcam do lektury poprzedniego rozdziału.

W tym rozdziale poruszam następujące tematy:

- wprowadzenie do testów jednostkowych,
- wyjaśnienie struktury projektu stosującego testy jednostkowe,
- analiza anatomii klasy testu jednostkowego,
- omówienie podstaw frameworka xUnit,
- omówienie powiązań zachodzących między regułami SOLID a testami jednostkowymi.

Zanim zakończysz lekturę rozdziału, nabędziesz umiejętności pozwalających na tworzenie prostych testów jednostkowych.

Wymagania techniczne

Kod dla tego rozdziału znajdziesz w repozytorium GitHub pod adresem *https://github.com/PacktPublishing/Pragmatic-Test-Driven-Development-in-C-Sharp-and-.NET/tree/main/ch03*.

Wprowadzenie do testów jednostkowych

W przypadku stosowania podejścia w stylu TDD tworzy się znacznie więcej testów jednostkowych niż produkcyjnego kodu źródłowego (czyli zwykłego kodu aplikacji). W przeciwieństwie do innych kategorii testowania, testy jednostkowe będą dyktowały pewne decyzje architekturalne dotyczące aplikacji oraz będą wymuszały stosowanie mechanizmu wstrzykiwania zależności.

Nie zamierzam tutaj rozwodzić się nad długimi definicjami. Zamiast tego omówię stosowanie testów jednostkowych i posłużę się przy tym wieloma przykładami. W tym podrozdziale zaprezentuję także framework xUnit, przeznaczony do przeprowadzania testów jednostkowych, oraz dokładnie omówię strukturę testu jednostkowego.

Czym jest testowanie jednostkowe?

Testowanie jednostkowe oznacza sprawdzenie sposobu działania kodu, w trakcie którego rzeczywiste zależności zostają zastąpione dublerami. Pozwól mi wesprzeć tę definicję przykładem pochodzącym z klasy `WeatherForecastController`.

```
private readonly ILogger <WeatherForecastController>
    _logger;
public double ConvertCToF(double c)
{
    double f = c * (9d / 5d) + 32;
    _logger.LogInformation("conversion requested");
    return f;
}
```

Działanie tej metody polega na przeprowadzeniu konwersji temperatury wyrażonej w stopniach Celsjusza na wartość w stopniach Fahrenheita oraz zarejestrowaniu każdego wywołania. Operacja rejestrowania danych ma tutaj mniejsze znaczenie, ponieważ funkcjonalnością metody jest konwersja wartości.

Działanie tej metody to *konwersja temperatury wyrażonej w stopniach Celsjusza na wartość w stopniach Fahrenheita*, a *zależność* w postaci funkcji rejestrowania danych jest dostępna za pomocą obiektu `_logger`. W trakcie działania aplikacji następuje wstrzyknięcie egzemplarza `Logger<>` odpowiedzialnego za zapis informacji do rzeczywistego nośnika. Jednak podczas testowania prawdopodobnie będziemy chcieli wyeliminować efekt uboczny w postaci operacji zapisu.

Opierając się na wcześniejszej definicji, trzeba *zastąpić rzeczywistą zależność* używaną przez obiekt `_logger` w trakcie działania aplikacji. Zamiast niego będzie użyty dubler, a testowanie dotyczy funkcjonalności konwersji. Sposób realizacji tego zadania dokładnie pokazuję w dalszej części rozdziału.

Spójrz na kolejny przykład pochodzący z tej samej klasy.

```
private readonly IClient _client;
public async Task<IEnumerable<WeatherForecast>> GetReal()
{
    ...
    OneCallResponse res = await _client.OneCallAsync(...
    ...
}
```

Działanie tej metody polega na pobraniu rzeczywistej prognozy pogody i jej przekazaniu do komponentu wywołującego metodę. W tym miejscu obiekt `_client` przedstawia zależność OpenWeather. Działanie metody *nie* wiąże się z pracą dotyczącą szczegółów protokołu RESTful API OpenWeather ani protokołu HTTP. Tym się zajmuje egzemplarz `_client`. Konieczne jest zastąpienie rzeczywistej zależności, `Client`, używanej przez obiekt `_client` w trakcie działania aplikacji, odpowiednią zależnością stosowaną w trakcie przeprowadzania testów (nazywamy ją **dublerem używanym podczas testów**). W następnym rozdziale dokładnie pokażę, jak można to zrobić na wiele sposobów.

Mam świadomość, że w tym momencie ta koncepcja może wydawać się nieco tajemnicza. Zaufaj mi, a wszystko wkrótce zacznie stawać się jaśniejsze. W następnym punkcie przejdę do omówienia frameworków testów jednostkowych. Są one niezbędne do przeprowadzania testów jednostkowych przedstawionych przykładów oraz kodu aplikacji WFA.

Frameworki testów jednostkowych

Platforma .NET 6 oferuje trzy ważne frameworki testów jednostkowych. Najpopularniejszym z nich jest **xUnit**, z którego będziemy korzystać w książce. Dwa pozostałe to **NUnit** i **MSTest**.

- *NUnit* to biblioteka otwartoźródłowa. Na początku powstała jako port dla frameworka JUnit z języka programowania Java, a następnie została całkowicie przepisana od nowa. Nadal można się z nim spotkać w starszych projektach. W większości nowych projektów używa się frameworka xUnit.
- *MSTest* to opracowany przez Microsoft framework testów jednostkowych, który zyskał popularność, ponieważ jest dostarczany razem z oprogramowaniem Visual Studio. Dzięki temu unika się jakiegokolwiek wysiłku związanego z instalowaniem frameworka, zwłaszcza że wcześniej nie były dostępne pakiety NuGet. Wersja druga MSTest została wydana jako oprogramowanie otwartoźródłowe. Pod względem funkcjonalności nowe wersje zawsze pozostają w tyle za NUnit, a nawet za xUnit.
- *xUnit* to otwartoźródłowy projekt stworzony przez programistów zajmujących się tworzeniem NUnit. Ten framework oferuje wiele funkcjonalności i jest nieustannie rozwijany.

Uwaga

XUnit można uznać za pewnego rodzaju ogólne określenie dla frameworków testów jednostkowych przeznaczonych dla różnych języków programowania np. **JUnit** (**Java**), **NUnit** (**.NET**), **xUnit** (**.NET**) i **CUnit** (**język C**). Nie należy tego pojęcia mylić z biblioteką o nazwie xUnit, czyli przeznaczoną dla platformy .NET biblioteką testów jednostkowych, której twórcy zdecydowali się na wykorzystanie już wcześniej używanej nazwy, co na pewno jest dezorientujące.

Poznanie jednego frameworka i następnie przejście do używania innego nie powinno wymagać zbyt wiele czasu, ponieważ wymienione projekty są podobne. Konieczne będzie jedynie opanowanie terminologii używanej przez dany framework. W następnym podrozdziale zajmiemy się dodaniem projektu xUnit do rozwiązania i przygotowaniem testów jednostkowych dla aplikacji WFA.

Wyjaśnienie struktury projektu stosującego testy jednostkowe

Szablony xUnit są dostarczane jako część oprogramowania Visual Studio. W tym podrozdziale pokażę, jak można dodać projekt xUnit za pomocą **wiersza poleceń platformy .NET**. Jeżeli na tym etapie jeszcze nie masz przykładowego kodu źródłowego dostarczonego dla rozdziału 2. książki, to zalecam jego pobranie.

Dodawanie projektu xUnit za pomocą wiersza poleceń

Obecnie mamy rozwiązanie składające się z jednego projektu ASP.NET Core. Teraz do tego rozwiązania chcemy dodać bibliotekę testów jednostkowych. W tym celu należy utworzyć nowy projekt xUnit o nazwie *Uqs.Weather.Tests.Unit* i umieścić go w tym samym katalogu co wspomniane rozwiązanie. Nowy projekt powinien używać platformy .NET 6.0.

```
dotnet new xunit -o Uqs.Weather.Tests.Unit -f net6.0
```

Kolejnym krokiem jest dodanie nowo utworzonego projektu do pliku rozwiązania.

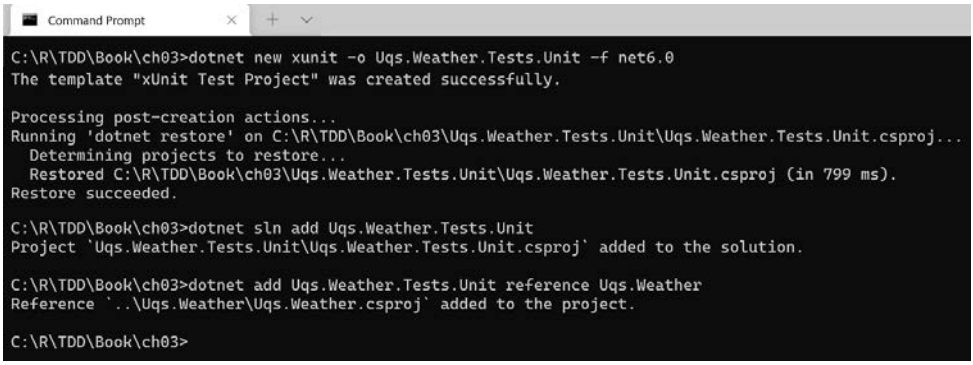
```
dotnet sln add Uqs.Weather.Tests.Unit
```

W tym momencie mamy dwa projekty w jednym rozwiązaniu. Skoro projekt testów jednostkowych będzie odpowiedzialny za testowanie projektu ASP.NET Core, ten pierwszy projekt powinien zawierać odwołanie do tego drugiego.

Dodaj odwołanie z projektu *Uqs.Weather.Tests.Unit* do *Uqs.Weather*.

```
dotnet add Uqs.Weather.Tests.Unit reference Uqs.Weather
```

Na tym etapie rozwiązanie zostało w pełni przygotowane z poziomu wiersza poleceń. Zapis wykonanych poleceń i wygenerowane przez nie dane wyjściowe możesz zobaczyć na rysunku 3.1.



Rysunek 3.1. Utworzenie za pomocą wiersza poleceń nowego projektu xUnit w rozwiązaniu ASP.NET Core

W tym momencie mamy projekt aplikacji wraz z projektem testów jednostkowych.

Konwencje nazw w projekcie testów jednostkowych

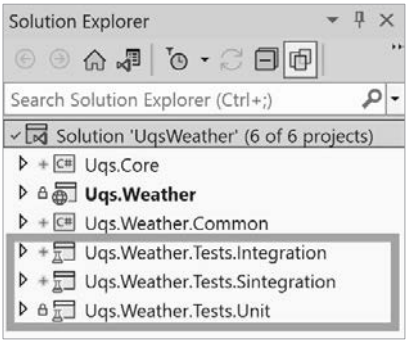
Zwróć uwagę na dodanie przyrostka `.Tests.Unit` do nazwy projektu aplikacji. W ten sposób nazwą projektu testów jednostkowych jest *Uqs.Weather.Tests.Unit*. To jest powszechnie stosowana konwencja w projektach testów jednostkowych.

Tę konwencję można rozbudować także na projekty innych rodzajów testów, np. testy integracyjne i testy sintegration — więcej informacji na ten temat znajdziesz w następnym rozdziale. W takim przypadku rozwiązanie będzie zawierało kolejne projekty o wymienionych tutaj nazwach.

- `Uqs.Weather.Tests.Integration`
- `Uqs.Weather.Tests.Sintegration`

Dzięki zastosowaniu takiej konwencji wystarczy krótkie spojrzenie na listę projektów, aby szybko odszukać te związane z testami i te zawierające kod produkcyjny, umieszczone obok siebie, jak pokazałem na rysunku 3.2.

Taka konwencja pomaga również w odwoływaniu się do poszczególnych projektów testów z poziomu rozwiązania ciągłej integracji, na którego temat więcej dowiesz się w rozdziale 11., gdy zechcesz wykonać wszystkie kategorie testów. Spójrz na przykład takiego odwołania: `Uqs.Weather.Tests.*`.



Rysunek 3.2. Projekty testów wyświetlone w oknie Solution Explorer rozwiązania

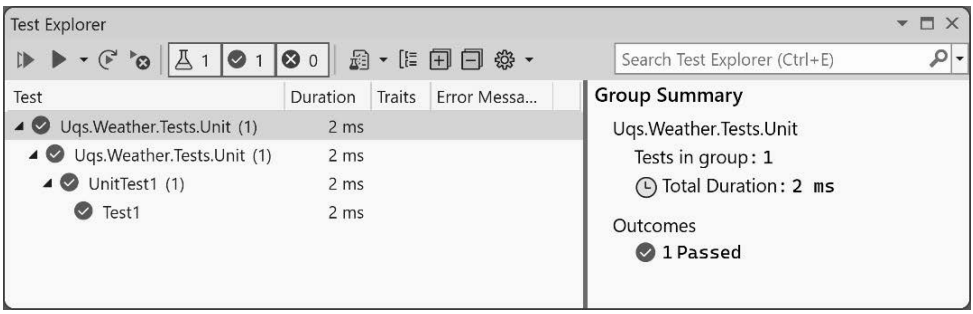
Wykonanie przykładowego testu jednostkowego

Szablon frameworka xUnit jest dostarczany z przykładową klasą testu jednostkowego, zdefiniowaną w pliku o nazwie *UnitTest1.cs*, która ma przykładową metodę razem z przedstawionym tutaj fragmentem kodu.

```
using Xunit;
namespace Uqs.Weather.Tests.Unit;
public class UnitTest1
{
    [Fact]
    public void Test1()
    {
    }
}
```

To jest pojedynczy test jednostkowy o nazwie `Test1`, obecnie pusty, a tym samym niewykonujący żadnego działania. Aby się upewnić co do działania frameworka xUnit oraz jego integracji z Visual Studio, możesz spróbować wykonać ten pojedynczy test.

Z poziomu menu Visual Studio wybierz opcję *Test/Run All Tests* bądź skorzystaj ze skrótu klawiszowego *Ctrl+R, A*. To spowoduje wykonanie wszystkich testów w projekcie (obecnie to będzie tylko jeden). Na ekranie zostanie wyświetlone okno dialogowe o nazwie *Test Explorer* (zob. rysunek 3.3).



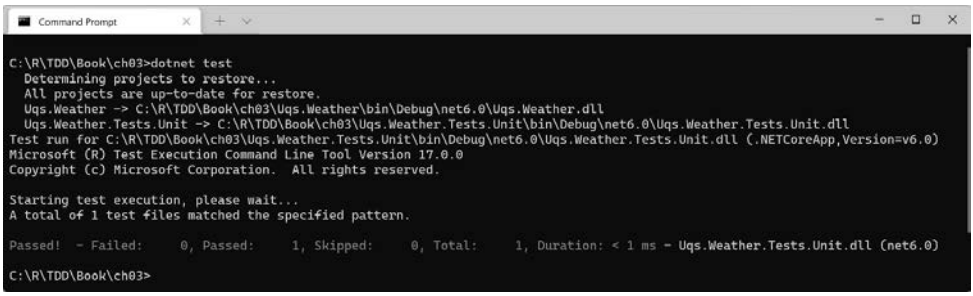
Rysunek 3.3. Okno dialogowe Test Explorer

W tym oknie została wyświetlona hierarchia w następującej postaci: **Nazwa projektu/Przestrzeń nazwy klasy testowej/Nazwa klasy testowej/Nazwa metody testowej**.

Jeżeli lubisz pracować z poziomu wiersza poleceń, wówczas możesz w nim przejść do katalogu projektu, a następnie wydać następujące polecenie:

```
dotnet test
```

To spowoduje wygenerowanie danych wyjściowych pokazanych na rysunku 3.4.



Rysunek 3.4. Uruchomienie testów za pomocą wiersza poleceń

Przekonałem się, że w trakcie codziennych zadań związanych z programowaniem z zastosowaniem programowania sterowanego testami, częściej używa się okna dialogowego *Test Explorer* niż wiersza polecenia. Wiersz poleceń okazuje się użyteczny podczas uruchamiania całego rozwiązania bądź w zadaniach związanych z ciągłą integracją lub zautomatyzowanymi operacjami.

Okno Test Explorer

Okno *Test Explorer* jest integralną częścią oprogramowania Visual Studio. Ponadto framework xUnit dodaje kilka bibliotek pozwalających na współdziałanie tego okna i Visual Studio z testami xUnit. Istnieją oferowane przez podmioty zewnętrzne produkty, które okazują się znacznie bardziej zaawansowanymi rozwiązaniami przeznaczonymi do wykonywania testów. Jednym z nich jest *JetBrains ReSharper Unit Test Explorer*. W tym momencie mamy wszystko, co jest potrzebne do rozpoczęcia tworzenia kodu testów jednostkowych.

Analiza anatomii klasy testu jednostkowego

Podczas używania testów jednostkowych zwykle tworzona jest tzw. **klasa testu jednostkowego**, przeznaczona dla klasy produkcyjnej — jedna klasa testu jednostkowego dla jednej klasy produkcyjnego kodu źródłowego.

Teraz tę koncepcję zastosujemy w projekcie WFA: klasą produkcyjnego kodu źródłowego jest `WeatherForecastController`, a klasa testu jednostkowego będzie nosiła nazwę `WeatherForecastControllerTests`. Nazwę przykładowej klasy `UnitTest1` zmień więc na `WeatherForecastControllerTests`.

Wskazówka

Wskaźnik myszy możesz umieścić w dowolnym miejscu nazwy klasy w kodzie źródłowym (w omawianym przykładzie to nazwa `UnitTest1`), by następnie nacisnąć klawisze *Ctrl+R, R* (naciśnij i przytrzymaj klawisz *Ctrl*, a potem w krótkim odstępie czasu dwukrotnie naciśnij klawisz *R*). Wpisz nową nazwę klasy, `WeatherForecastContro` ↪`llerTests`, i naciśnij klawisz *Enter*. Jeżeli będzie zaznaczone pole wyboru *Rename symbol's file*, to ta operacja spowoduje również zmianę nazwy pliku.

Teraz wyjaśnię, jak należy zorganizować klasę testu jednostkowego i jej metody.

Konwencja nadawania nazwy klasie

Przekonałem się, że najczęściej stosowana konwencja polega na nadaniu klasie testu jednostkowego nazwy odpowiadającej nazwie klasy produkcyjnego kodu źródłowego i dołączeniu do niej przyrostka `Tests`. W takim przypadku dla klasy produkcyjnej `MyProduc` ↪`tionCode` będzie istniała klasa testu jednostkowego o nazwie `MyProductionCodeTests`.

Podczas stosowania podejścia w stylu TDD trzeba będzie wielokrotnie w krótkim czasie przechodzić między klasami produkcyjnego kodu źródłowego i testów jednostkowych. Nadawanie im nazw zgodnych z wymienioną tutaj konwencją pozwala na łatwe odszukiwanie plików testów oraz odpowiadającego im kodu i na odwrót. Ponadto jasno zostaje wyrażony związek między dwiema klasami.

Metody testowe

Każda klasa testowa zawiera metody przeznaczone do przetestowania pewnych fragmentów funkcjonalności, nazywanych jednostkami, zdefiniowanych w klasie produkcyjnego kodu źródłowego. Zapoznaj się teraz z przykładem testowania metody `ConvertCToF()`.

Przykładowy test 1

Jednym z naszych wymagań jest sprawdzenie działania konwersji z dokładnością do jednego miejsca po przecinku. Rozważmy więc przypadek testowy dla wartości zera stopni Celsjusza (0,0 °C) i sprawdźmy, czy metoda zwróci dla niego wartość 32,0 stopni Fahrenheita. W tym celu należy usunąć metodę `Test1()` w klasie testów jednostkowych i zastąpić ją przedstawioną w kolejnym fragmencie kodu.

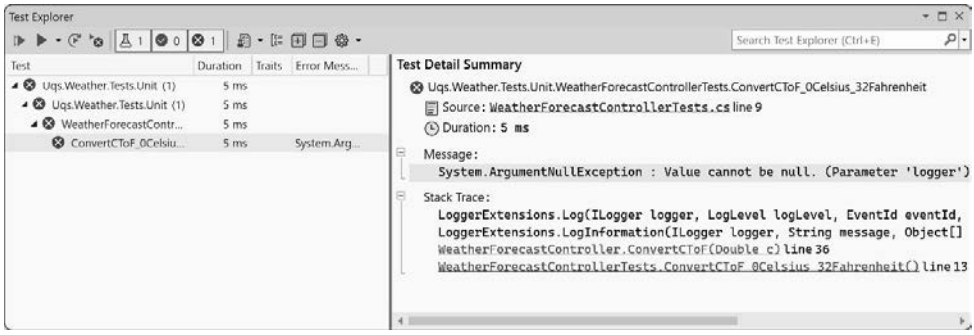
```
[Fact]
public void ConvertCToF_0Celsius_32Fahrenheit()
{
    const double expected = 32d;
    var controller = new WeatherForecastController(
```

```
            null!, null!, null!, null!);
        double actual = controller.ConvertCToF(0);
        Assert.Equal(expected, actual);
    }
```

Ten kod inicjalizuje klasę produkcyjnego kodu źródłowego, wywołuje testowaną metodę, a następnie sprawdza, czy otrzymany wynik jest zgodny z oczekiwaniami.

Dzięki atrybutowi `Fact` metoda staje się testem jednostkowym. Z kolei `Assert` to klasa statyczna oferująca użyteczne metody przeznaczone do porównywania wyników oczekiwanych z faktycznie otrzymanymi. Zarówno atrybut `Fact`, jak i klasa `Assert` są częścią frameworka xUnit.

Wykonaj ten test w oknie *Test Explorer*. W tym celu naciśnij klawisze *Ctrl+A, R*. To spowoduje wygenerowanie danych wyjściowych pokazanych na rysunku 3.5.



Rysunek 3.5. Dane wyjściowe w oknie Test Explorer wskazujące na zakończenie testu niepowodzeniem

Jeżeli spojrzysz na kod źródłowy kontrolera, zobaczysz, że Visual Studio oznacza kolorem czerwonym ścieżki kodu, których wykonanie zakończyło się niepowodzeniem, jak pokazałem na rysunku 3.6.

```
1 reference | 0/1 passing
public WeatherForecastController(ILogger<WeatherForecastController> logger,
    IClient client, INowWrapper nowWrapper, IRandomWrapper randomWrapper)
{
    _logger = logger;
    _client = client;
    _nowWrapper = nowWrapper;
    _randomWrapper = randomWrapper;
}

[HttpGet("ConvertCToF")]
1 reference | 0/1 passing
public double ConvertCToF(double c)
{
    double f = c * (9d / 5d) + 32;
    _logger.LogInformation("conversion requested");
    return f;
}
```

Rysunek 3.6. Zakończone niepowodzeniem ścieżki wykonywania kodu

Komunikat o błędzie wyraźnie wskazuje, co spowodowało zgłoszenie wyjątku `Argument ↪NullException`.

```
_logger.LogInformation("conversion requested");
```

To jest zgodne z oczekiwaniami, ponieważ parametr `logger` przekazany z testu jednostkowego ma wartość `null`. Nie chcemy, aby wywołanie `_logger.LogInformation()` podejmowało jakiekolwiek działania. W tym celu skorzystamy z egzemplarza `NullLogger<>`, który zgodnie z informacjami zamieszczonymi w oficjalnej dokumentacji faktycznie nic nie robi. Nasz test jednostkowy wymaga jeszcze przedstawionej tutaj zmiany, aby można było rzeczywisty komponent rejestrowania danych zastąpić jego imitacją.

```
var logger =
    NullLogger<WeatherForecastController>.Instance;
var controller = new WeatherForecastController(
    logger, null!, null!, null!);
```

Jeżeli ponownie uruchomisz ten test, wówczas wszystkie czerwone symbole zmienią kolor na zielony, a test zostanie zaliczony.

Przykładowy test 2

Aby przetestować inne dane wejściowe i wyjściowe metody, w klasie można zdefiniować kolejne testy jednostkowe. Zastosowanie ma ta sama konwencja nazw metod testowych. Możemy więc przygotować metody o przedstawionych tutaj sygnaturach.

```
public void ConvertCToF_1Celsius_33p8Fahrenheit() {...}
...
public void ConvertCToF_Minus1Celsius_3Op2Fahrenheit() {...}
```

Jednak istnieje pewne rozwiązanie, dzięki któremu można uniknąć konieczności tworzenia podobnego testu jednostkowego dla poszczególnych wariantów wartości. Spójrz na kolejny fragment kodu.

```
[Theory]
[InlineData(-100 , -148)]
[InlineData(-10.1,  13.8)]
[InlineData(10   ,  50)]
public void ConvertCToF_Cel_CorrectFah(double c, double f)
{
    var logger =
        NullLogger<WeatherForecastController>.Instance;
    var controller = new WeatherForecastController(
        logger, null!, null!, null!);
    double actual = controller.ConvertCToF(c);
    Assert.Equal(f, actual, 1);
}
```

Zwróć uwagę na użycie atrybutu `Theory` zamiast `Fact`. Każdy atrybut `InlineData` będzie działał w charakterze pojedynczego testu jednostkowego. Możesz nawet wyeliminować poprzedni test jednostkowy i zamiast niego przygotować nowy atrybut `InlineData`. Nie muszę chyba dodawać, że atrybuty `Theory` i `InlineDate` są dostarczane przez xUnit.

Teraz możesz śmiało wykonać zdefiniowane testy jednostkowe.

Inne przykłady zostały przedstawione w rozdziale 1. Są podobne do zamieszczonych w tym rozdziale, więc możesz do nich powrócić.

W przykładowych testach pierwszym i drugim sprawdzana była prosta metoda, `Convert➥CToF()`, mająca tylko jedną zależność, `_logger`. Więcej bardziej zaawansowanych scenariuszy testowych przedstawię po omówieniu w następnym rozdziale dublerów używanych podczas testów. W rzeczywistości Twój kod w środowisku produkcyjnym będzie znacznie bardziej skomplikowany niż używana tutaj prosta metoda konwersji temperatury oraz będzie zawierał wiele zależności. Jednak w tym miejscu wykonujesz dopiero pierwszy krok do świata testowania.

Konwencja nazw

Nazwy metod testów jednostkowych tworzy się w popularnej konwencji: `NazwaTestowanej➥Metody_Warunek_Oczekiwania`. Z tą konwencją mogłeś spotkać się już wcześniej. Spójrz na jeszcze inne hipotetyczne przykłady nazw:

- `SaveData_CannotConnectToDB_InvalidOperationException`
- `OrderShoppingBasket_EmptyBasket_NoAction`

W książce znajdziesz jeszcze wiele innych przykładów, które powinny dokładniej pokazać wspomnianą konwencję w akcji.

Wzorzec „przygotowanie, działanie, asercja”

Wcześniej przedstawiona metoda testowa i ogólnie wszystkie metody testów jednostkowych stosują następujący wzorzec:

1. Utworzenie stanu, zadeklarowanie pewnych zmiennych oraz poczynienie przygotowań.
2. Wywołanie testowanej metody.
3. Asercja faktycznie otrzymanego wyniku i oczekiwanego.

W świecie testowania te trzy etapy otrzymały wymienione tutaj nazwy:

Przygotowanie (ang. *arrange*), **działanie** (ang. *act*) i **asercja** (ang. *assert*), czyli **AAA**.

Tych nazw używa się w komentarzach umieszczanych w kodzie źródłowym, aby wskazać poszczególne etapy i je rozdzielić. Zgodnie z tym kod jednego z wcześniejszych testów można zapisać w przedstawionej tutaj postaci.

```
[Fact]
public void ConvertCToF_0Celsius_32Fahrenheit()
{
    // Przygotowanie.
    const double expected = 32d;
    var controller = new WeatherForecastController(...);

    // Działanie.
    double actual = controller.ConvertCToF(0);
```

```
        // Asercja.
        Assert.Equal(expected, actual);
    }
```

Zwróć uwagę na komentarze umieszczone w kodzie.

> **Uwaga**
>
> Czasami zdarza się, że zespół nie lubi takiej separacji osiągniętej dzięki użyciu komentarzy. Zamiast tego decyduje się na innego rodzaju oznaczenie AAA, np. pozostawienie pustego wiersza między poszczególnymi sekcjami.

Przedstawiony tutaj wzorzec AAA to więcej niż tylko konwencja. Dzięki niemu odczytywanie kodu metody okazuje się znacznie łatwiejsze. Ponadto kładzie nacisk na to, że w metodzie testu jednostkowego powinna istnieć tylko jedna sekcja *działania*. W efekcie, zgodnie z najlepszymi praktykami, przyjęło się, że test jednostkowy powinien obsługiwać tylko jedną strukturę AAA.

Używanie fragmentów kodu w Visual Studio

Każdy test jednostkowy będzie miał tę samą strukturę. Visual Studio pomaga programistom w tworzeniu takiej samej struktury dzięki wykorzystaniu tzw. **fragmentów kodu** (ang. *code snippets*). W katalogu o nazwie *CodeSnippets* w repozytorium wymienionym na początku rozdziału umieściłem przykładowy plik zawierający fragment kodu dla testu jednostkowego. Wspomniany plik nosi nazwę *aaa.snippet*. Możesz go otworzyć i wyświetlić jego zawartość za pomocą zwykłego edytora tekstu (a nie procesora tekstu).

Aby wykorzystać ten plik fragmentu kodu w Windows, skopiuj go do następującego katalogu (wybierz odpowiednią wersję Visual Studio):

```
%USERPROFILE%\Documents\Visual Studio 2022\Code Snippets\Visual C#\My Code
↪Snippets
```

Po skopiowaniu pliku *aaa.snippet* w klasie testu jednostkowego wpisz *aaa*, a następnie naciśnij klawisz *Tab*. To powinno spowodować wygenerowanie przedstawionego tutaj fragmentu kodu.

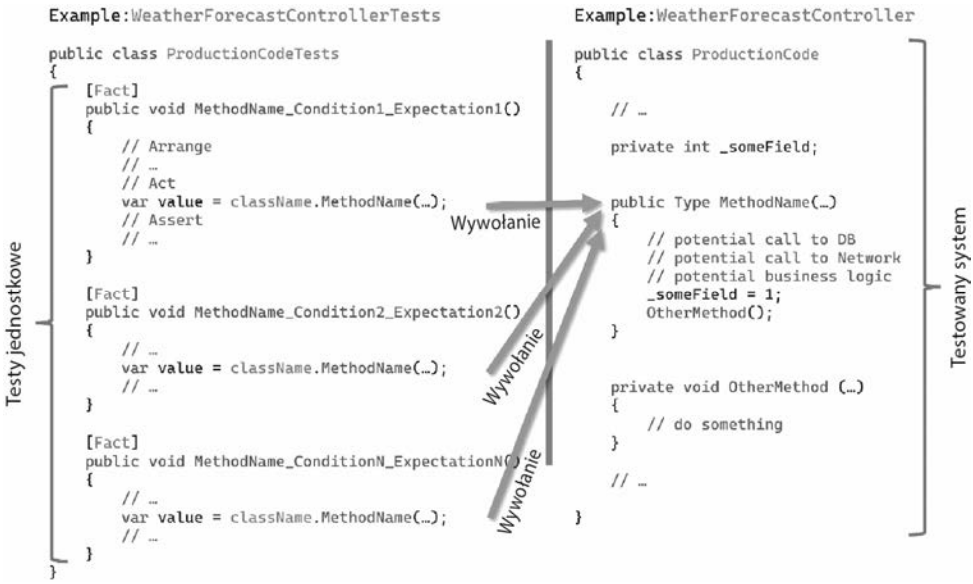
```
[Fact]
public void Method_Condition_Expectation()
{
    // Przygotowanie.

    // Działanie.

    // Asercja.
}
```

Zamiast zbyt wiele mówić na temat umieszczania pojedynczej reguły AAA w teście jednostkowym, takie podejście będę prezentować w książce i tym samym podkreślę ten styl tworzenia testów jednostkowych.

Na tym kończę ogólne omówienie anatomii klasy i struktury metody testu jednostkowego. Mogę teraz przejść do przedstawienia odpowiednika klasy testu jednostkowego, czyli testowanego systemu.

Testowany system

Celem testu jednostkowego jest przetestowanie pojedynczej funkcjonalności produkcyjnego kodu źródłowego. Poszczególne klasy testów jednostkowych mają odpowiadające im klasy produkcyjnego kodu źródłowego, które są przedmiotem testów. Tego rodzaju kod produkcyjny będziemy określać mianem **testowanego systemu** (ang. *system under test*, SUT). Na rysunku 3.7 pokazałem w sposób graficzny, czym jest testowany system.



Rysunek 3.7. Testy jednostkowe są wykonywane dla testowanego systemu

Wprawdzie pojęcie testowanego systemu spotyka się najczęściej, ale są też inne określenia, np. **testowana klasa** (ang. *class under test*, **CUT**), **testowany kod** (ang. *code under test*, **CUT** — tak, używany jest ten sam akronim) i **testowana metoda** (ang. *method under test*, **MUT**).

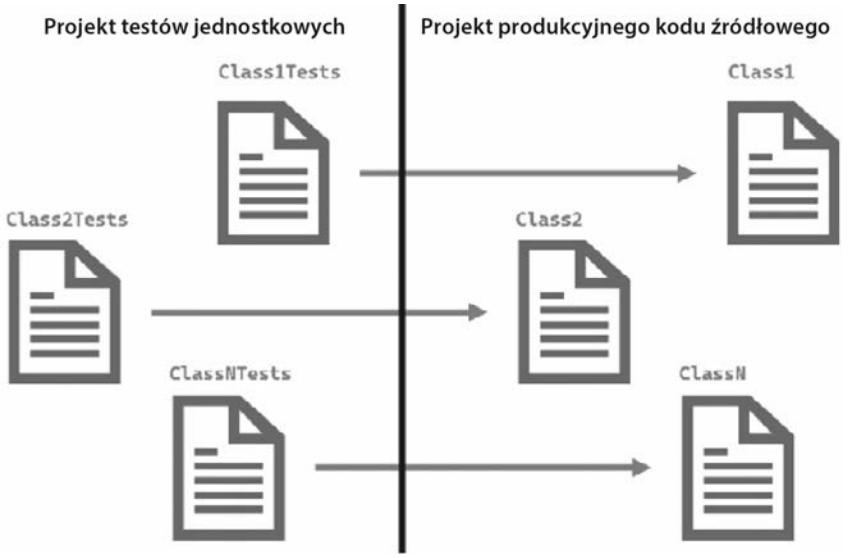
Określenie *SUT* pojawia się w komunikacji między programistami, a także powszechnie stosuje się je w kodzie, aby wyraźnie wskazać, co jest testowane. Spójrz na kolejny fragment kodu.

```
var sut = new ProductionCode(...);
```

Bardzo ważne jest, aby w przypadku tworzonych testów jednostkowych dokładnie określić, co jest testowanym systemem. W miarę rozbudowy projektu można dostrzec, że będzie on przyjmować postać pokazaną na rysunku 3.8.

Każdej klasie testu jednostkowego odpowiada klasa zawierająca produkcyjny kod źródłowy.

Tutaj oraz w rozdziale 1. przedstawiłem kilka wybranych funkcji frameworka xUnit. W następnym podrozdziale znacznie dokładniej omówię ten framework.



Rysunek 3.8. Projekty testów jednostkowych i produkcyjnego kodu źródłowego

Omówienie podstaw frameworka xUnit

Framework xUnit zapewnia środowisko hostingowe dla Twoich testów. Jedną z ważniejszych funkcjonalności xUnit jest zgodność z omówioną wcześniej regułą AAA. Ponadto świetnie integruje się ze środowiskiem IDE Visual Studio i jego oknem *Test Explorer*.

Możesz mieć pewność, że w książce pojawią się rozbudowane przykłady przedstawiające użycie xUnit. Mimo to warto w przynajmniej kilku punktach omówić najważniejsze funkcje oferowane przez ten framework.

Atrybuty Fact i Theory

W projekcie testów każda metoda udekorowana atrybutem `Fact` lub `Theory` stanie się metodą testową. Atrybut `Fact` jest przeznaczony dla niesparametryzowanego testu jednostkowego, podczas gdy atrybut `Theory` stosuje się w sparametryzowanych testach jednostkowych. W tym drugim przypadku można używać jeszcze innych atrybutów, np. `InlineData`, na potrzeby parametryzacji.

Uwaga

Visual Studio wyświetla nad nazwą metody wizualny symbol, co pozwala na wykonywanie metod udekorowanych żądanymi atrybutami. Jednak czasami te symbole nie będą się pojawiały aż do chwili wykonania wszystkich testów.

Wykonywanie testów

Każdy test jednostkowy będzie *wykonywany niezależnie* i spowoduje utworzenie egzemplarza klasy. Testy jednostkowe *nie współdzielą* między sobą informacji o stanie. Dlatego też klasa testu jednostkowego będzie wykonywana nieco inaczej niż zwykła klasa. Pozwól mi to wyjaśnić nieco dokładniej na podstawie zamieszczonego tutaj przykładowego fragmentu kodu.

```
public class SampleTests
{
    private int _instanceField = 0;
    private static int _staticField = 0;

    [Fact]
    public void UnitTest1()
    {
        _instanceField++;
        _staticField++;
        Assert.Equal(1, _instanceField);
        Assert.Equal(1, _staticField);
    }

    [Fact]
    public void UnitTest2()
    {
        _instanceField++;
        _staticField++;
        Assert.Equal(1, _instanceField);
        Assert.Equal(2, _staticField);
    }
}
```

Przedstawione wcześniej testy jednostkowe zostaną zaliczone. Zwróć uwagę na to, że pomimo inkrementacji w obu metodach testu jednostkowego wartości `_instanceField` wartość tej właściwości nie jest współdzielona między poszczególnymi metodami. W trakcie każdego tworzenia metody przez xUnit tworzone są również egzemplarze wszystkich klas. Dlatego też za każdym razem przed wykonaniem metody wartość właściwości zostaje wyzerowana. Taki sposób działania xUnit promuje regułę testów jednostkowych znaną jako **brak współzależności**, na której temat więcej dowiesz się z lektury rozdziału 6.

Z drugiej strony pole statyczne jest współdzielone między metodami i dlatego jego wartość się zmienia.

Uwaga

Wprawdzie zdecydowałem się na użycie pól egzemplarza i statycznych, by pokazać odmienny sposób działania klasy testu jednostkowego, ale chciałbym w tym miejscu podkreślić, że korzystanie ze statycznego pola `read-write` w teście jednostkowym jest uznawane za antywzorzec, ponieważ prowadzi do złamania reguły *braku współzależności*. Ogólnie rzecz biorąc, w klasach testów jednostkowych nie powinny istnieć wspólne pola typu `write`, a ponadto lepiej będzie oznaczać pola za pomocą słowa kluczowego `readonly`.

Jeżeli w klasie zwykłego kodu (nie w klasach testów jednostkowych) znajdują się metody o takich samych nazwach jak w testach i te metody są wywoływane, wówczas można się spodziewać inkrementacji właściwości `_instanceField` do wartości `2`. W omawianym przykładzie jednak tak nie jest.

Klasa Assert

`Assert` to klasa statyczna i jednocześnie część frameworka xUnit. Oto sposób, w jaki oficjalna dokumentacja tego frameworka definiuje klasę `Assert`:

Zawiera różne metody statyczne, które są używane do potwierdzenia, że zostały spełnione żądane warunki.

Warto pokrótce zapoznać się z najważniejszymi metodami klasy `Assert`.

- `Equal(wartość_oczekiwana, wartość_faktyczna)`. Mamy tutaj serię przeciążonych metod, których zadaniem jest porównywanie wartości oczekiwanych i faktycznie otrzymanych. Kilka przykładów użycia metody `Equal()` znalazło się w rozdziałach 1. i tym.
- `True(wartość_faktyczna)`. Zamiast do porównania dwóch obiektów używać wywołania `Equal()`, można skorzystać z omawianego, gdyż oznacza to większą czytelność. Warto przedstawić to na przykładzie.

  ```
  Assert.Equal(true, isPositive);
  // lub
  Assert.True(isPositive);
  ```

- `False(wartość_faktyczna)`. Działanie tej metody jest przeciwne do poprzedniej.
- `Contains(wartość_oczekiwana, kolekcja)`. Grupa przeciążonych metod sprawdzających istnienie pojedynczego elementu w kolekcji.
- `DoesNotContain(wartość_oczekiwana, kolekcja)`. Działanie tej metody jest przeciwne do poprzedniej.
- `Empty(kolekcja)`. Ta metoda sprawdza, czy kolekcja jest pusta.
- `Assert.IsType<Typ>(wartość_faktyczna)`. Ta metoda sprawdza, czy obiekt jest określonego typu.

To tylko kilka wybranych metod. Zachęcam do odwiedzenia oficjalnej witryny internetowej frameworka xUnit i zapoznania się z dostępnymi metodami. Ewentualnie możesz zrobić to, co większość programistów: w klasie testu jednostkowego wpisz `Assert` i następnie wpisz *kropkę*, co spowoduje wyświetlenie listy *IntelliSense* zawierającej dostępne metody.

Metody klasy `Assert` będą prowadziły komunikację z komponentem przeznaczonym do wykonywania testów, np. oknem *Test Explorer*, i przekazywały informacje na temat wyników asercji.

Klasa Record

Klasa `Record` to klasa statyczna przeznaczona do rejestrowania wyjątków. Dzięki niej możesz sprawdzić, czy metoda zgłasza właściwy wyjątek. Oto przykład jednej z metod statycznych tej klasy:

```
public static System.Exception Exception(Action testCode)
```

Poprzedni fragment kodu powoduje zwrot wyjątku zgłoszonego przez obiekt `Action`. Zapoznaj się z przedstawionym tutaj przykładem.

```
[Fact]
public void Load_InvalidJson_FormatException()
{
    // Przygotowanie.
    string input = "{niepoprawne dane JSON";

    // Działanie.
    var exception = Record.Exception(() =>
        JsonParser.Load(input));

    // Asercja.
    Assert.IsType<FormatException>(exception);
}
```

W tym miejscu sprawdzamy, czy metoda `Load` spowoduje zgłoszenie wyjątku `FormatExcep` ↪`tion` w razie przekazania jej niepoprawnych danych wejściowych w formacie JSON.

To było krótkie podsumowanie funkcjonalności oferowanej przez framework xUnit. Masz już wystarczającą wiedzę, aby rozpocząć tworzenie prostych testów jednostkowych.

Omówienie powiązań zachodzących między regułami SOLID a testami jednostkowymi

Reguły SOLID są szeroko omawiane i zachwalane zarówno w internecie, jak i w książkach. Jest bardzo prawdopodobne, że w tym miejscu nie spotykasz się z nimi po raz pierwszy. To również popularny temat rozmów kwalifikacyjnych. Reguły SOLID oznaczają:

- regułę jednej odpowiedzialności (ang. ***s**ingle responsibility*),
- regułę otwarte-zamknięte (ang. ***o**pen-close*),
- zasadę podstawień Barbary Liskov (ang. ***l**iskov substitution principle*),
- zasadę rozdzielania interfejsów (ang. ***i**nterface segregation principle*),
- zasadę odwrócenia zależności (ang. ***d**ependency inversion principle*).

W tym podrozdziale najbardziej interesują nas związki zachodzące między regułami SOLID a testami jednostkowymi. Wprawdzie nie wszystkie reguły mają silne powiązania z testami jednostkowymi, ale omówię je tutaj, żeby przedstawić pełny obraz sytuacji.

Reguła jednej odpowiedzialności

Reguła jednej odpowiedzialności wiąże się z tym, że każda klasa odpowiada tylko za pojedyncze zadanie. To prowadzi do sytuacji, w której istnieje wyłącznie jeden powód do jej zmiany. Zalety takiego podejścia są następujące:

- **Możliwość łatwiejszego odczytywania i zrozumienia klas**

  Klasy będą miały mniejszą liczbę metod, a to oznacza mniejszą ilość kodu źródłowego. Ponadto interfejs klasy będzie składał się z mniejszej liczby metod.

- **Mniej efektów ubocznych podczas wprowadzania zmian w funkcjonalności**

  Im mniej klas do zmiany, tym łatwiejsze będą te zmiany.

- **Mniejsze prawdopodobieństwo zmiany, co przekłada się na mniejszą liczbę potencjalnych błędów**

  Większa ilość kodu źródłowego przekłada się na więcej potencjalnych błędów, a modyfikacje kodu mogą prowadzić do powstawania błędów. Dlatego też mniejsza ilość kodu źródłowego oznacza mniejszą liczbę wprowadzanych w nim zmian.

Przykład

Reguła jednej odpowiedzialności nie jest sztywno ustalona i pewną trudność może stanowić ustalenie, czym jest ta *odpowiedzialność*. Każdy programista będzie miał własne zdanie na ten temat. Zamieszczony tutaj przykład powinien pokazać ogólną ideę.

Załóżmy, że utworzyłeś własny format pliku o nazwie *ABCML* w celu rozwiązania określonego problemu, ponieważ istniejące formaty plików (np. JSON czy XML) nie spełniają Twoich wymagań. Zbiór klas, z których każda będzie odpowiadała za pojedyncze zadanie, może przedstawiać się następująco:

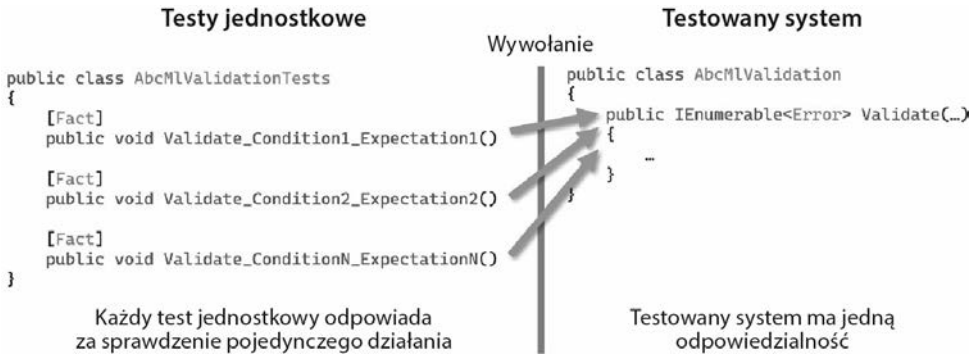
- Klasa przeznaczona do sprawdzenia, czy zawartość pliku ma właściwą strukturę.
- Klasa pozwalająca na wyeksportowanie ABCML do formatu ogólnego.
- Klasa dziedzicząca ogólną funkcjonalność eksportowania danych ABCML, aby zapewnić możliwość eksportowania do formatu JSON. Jeszcze inna klasa będzie oferowała eksportowanie danych do formatu XML.
- Klasa przedstawiająca węzeł w ABCML.
- Kolejne klasy.

Zwróć uwagę na podział odpowiedzialności między poszczególne klasy, pomimo że projekt nie jest przeznaczony do obsługi pojedynczej odpowiedzialności.

Reguła jednej odpowiedzialności i testy jednostkowe

Jest naturalne, że podczas przeprowadzania testów jednostkowych zastanawiasz się nad pojedynczą odpowiedzialnością dla klasy, a klasie testów jednostkowych nadajesz taką samą nazwę, jaką ma klasa kodu źródłowego, ale razem z przyrostkiem `tests`. Jeżeli więc zastanawiasz się nad przetestowaniem operacji sprawdzenia formatu pliku ABCML, wówczas możesz skorzystać z klasy `ABCMLValidationTests`.

W klasie testów jednostkowych poszczególne testy jednostkowe będą przeznaczone dla pojedynczego działania w testowanym systemie (SUT). Te działania razem prowadzą do powstania pojedynczej odpowiedzialności (zob. rysunek 3.9).



Rysunek 3.9. Wiele testów poszczególnych działań dotyczy pojedynczej odpowiedzialności testowanego systemu

Na rysunku 3.9 można zobaczyć wiele testów, z których każdy dotyczy pojedynczego działania. Wszystkie te działania są związane z pojedynczą odpowiedzialnością: *weryfikacja*. Wprawdzie po prawej stronie na rysunku 3.9 znajduje się tylko jedna metoda, ale to ma służyć jedynie do zilustrowania idei. W klasie możesz mieć wiele metod publicznych, a mimo to nadal będą one miały tylko jedną odpowiedzialność.

W rozdziale 6. poznasz wytyczną znaną jako *wytyczna pojedynczego działania*. Stosuje się ją w programowaniu sterowanym testami oraz w testach jednostkowych, aby zachęcać do stosowania reguły jednej odpowiedzialności.

Reguła otwarte-zamknięte

Reguła otwarte-zamknięte wiąże się z przygotowaniem klasy do dziedziczenia po niej (otwarte), aby wszelkie dodawane do niej funkcje mogły być dziedziczone bez konieczności jej modyfikacji (zamknięte).

Sednem tej reguły jest ograniczenie do minimum niepotrzebnych zmian za każdym razem, gdy dodawana jest nowa funkcjonalność.

Przykład

Spójrz teraz na przykład, który powinien rzucić nieco światła na tę koncepcję. Przyjmujemy założenie o utworzeniu biblioteki przeznaczonej do przeprowadzania operacji arytmetycznych. Na początku ta biblioteka *nie jest zgodna z regułą otwarte-zamknięte*, jak pokazałem w kolejnym fragmencie kodu.

```
public interface IArithmeticOperation {}
public class Addition : IArithmeticOperation
{
    public double Add(double left, double right) =>
        left + right;
}
public class Subtraction : IArithmeticOperation { … }
public class Calculation
{
    public double Calculate(IArithmeticOperation op,
        double left, double right) =>
        op switch
        {
          Addition addition => addition.Add(left, right),
          Subtraction sub => sub.Subtract(left, right),
          //Multiplication mul => mul.Multiply(left,right),
          _ => throw new NotImplementedException()
        };
}
```

Metoda `Calculate()` w tym fragmencie kodu będzie musiała być modyfikowana za każdym razem, gdy zostanie dodany nowy element `ArithmeticOperation`. Jeżeli na późniejszym etapie pracy zechcesz dodać obsługę operacji mnożenia, spójrz na polecenie umieszczone w komentarzu; wówczas metoda `Calculate()` będzie musiała być zmieniona, aby można było wykorzystać nową funkcjonalność.

Ta implementacja może być bardziej zgodna z regułą otwarte-zamknięte, jeśli zostanie wyeliminowana konieczność zmiany metody `Calculate()` za każdym razem, gdy dodawana jest obsługa nowej operacji. Spójrz na przykład, jak to zrobiono.

```
public interface IArithmeticOperation
{
    public double Operate(double left, double right);
}
public class Addition : IArithmeticOperation
{
    public double Operate(double left, double right) =>
        left + right;
}
public class Subtraction : IArithmeticOperation { ... }
// public class Multiplication : IArithmeticOperation { ... }
public class Calculation
{
```

```
        public double Calculate(IArithmeticOperation op,
            double left, double right) =>
                op.Operate(left, right);
    }
```

W poprzednim przykładzie polimorfizm wykorzystano do tego, aby metody `Calculation()` nie trzeba było zmieniać za każdym razem po dodaniu obsługi nowej operacji. Polecenie umieszczone w komentarzu pokazuje, jak można dodać nową operację mnożenia. Takie podejście jest bardziej zgodne z regułą otwarte-zamknięte.

Uwaga

Wprawdzie wszystkie przedstawione tutaj klasy i interfejsy znajdują się w pojedynczym pliku, który znajdziesz również w repozytorium GitHub, ale takie rozwiązanie ma jedynie ułatwić zaprezentowanie koncepcji. W rzeczywistości poszczególne klasy i interfejsy definiuje się we własnych plikach. Dzięki regule otwarte-zamknięte można również zmniejszyć niebezpieczeństwo modyfikacji pliku oraz ułatwić na poziomie kodu źródłowego sprawdzanie wprowadzonych zmian.

Reguła otwarte-zamknięte i testy jednostkowe

Testy jednostkowe chronią zmiany wprowadzane w klasach przez sprawdzenie, czy dana modyfikacja nie spowodowała przypadkowo uszkodzenia istniejącej funkcjonalności. Omawiana tutaj reguła i testy jednostkowe idą więc w parze. Zatem choć reguła otwarte-zamknięte zmniejsza niebezpieczeństwo niechcianych zmian, to testy jednostkowe zapewniają w przypadku modyfikacji dodatkową warstwę ochrony (weryfikacja reguł biznesowych).

Zasada podstawień Barbary Liskov

Zgodnie z **zasadą podstawień Barbary Liskov** egzemplarz klasy potomnej musi zastępować egzemplarz klasy nadrzędnej bez wpływu na wyniki, jakie zwrócił egzemplarz klasy bazowej. To oznacza, że klasa potomna powinna być faktycznym przedstawicielem jej klasy nadrzędnej.

Przykład

W tym miejscu posłużę się typowym przykładem akademickim, aby ułatwić zrozumienie omawianej koncepcji. Zapoznaj się z kolejnym fragmentem kodu.

```
public abstract class Bird
{
    public abstract void Fly();
    public abstract void Walk();
}
public class Robin : Bird
{
    public override void Fly() => Console.WriteLine("fly");
```

```
        public override void Walk() =>
            Console.WriteLine("walk");
    }
    public class Ostrich : Bird
    {
        public override void Fly() =>
            throw new InvalidOperationException();
        public override void Walk() =>
            Console.WriteLine("walk");
    }
```

W poprzednim fragmencie kodu, zgodnie z zasadą podstawień Barbary Liskov, klasa `Ostrich` nie powinna dziedziczyć po klasie `Bird`. Przystępujemy do modyfikacji kodu, aby zapewnić jego zgodność z omawianą zasadą.

```
public abstract class Bird
{
    public abstract void Walk();
}
public abstract class FlyingBird : Bird
{
    public abstract void Fly();
}
public class Robin : FlyingBird
{
    public override void Fly() => Console.WriteLine("fly");
    public override void Walk() =>
        Console.WriteLine("walk");
}
public class Ostrich : Bird
{
    public override void Walk() =>
        Console.WriteLine("walk");
}
```

W tym fragmencie kodu zmieniono hierarchię dziedziczenia przez wprowadzenie nowej klasy pośredniej o nazwie `FlyingBird`. To zapewnia zgodność kodu z zasadą podstawień Barbary Liskov.

Zasada podstawień Barbary Liskov i testy jednostkowe

Testy jednostkowe nie mają żadnego bezpośredniego wpływu na zasadę podstawień Barbary Liskov. Mimo to wspomniałem tutaj o niej, aby przedstawić pełny obraz sytuacji.

Zasada rozdzielania interfejsów

Zasada rozdzielania interfejsów głosi, że klas potomnych nie powinno się zmuszać do posiadania zależności od interfejsów, których nie używają. Interfejsy powinny być mniejsze, aby implementujące je komponenty mogły je łączyć i dopasowywać.

Przykład

Sposób implementowania kolekcji na platformie .NET zawsze uznawałem za najlepszy przykład wyjaśniający omawianą tutaj koncepcję. Zobacz, jak został zadeklarowany typ `List<T>`.

```
public class List<T> : ICollection<T>, IEnumerable<T>,
  IList<T>, IReadOnlyCollection<T>, IReadOnlyList<T>, IList
```

Implementuje on sześć interfejsów. Każdy interfejs zawiera ograniczoną liczbę metod. Typ `List<T>` udostępnia ogromną liczbę metod, ale to się odbywa przez wybór wielu interfejsów, z których każdy dodaje pewną liczbę metod.

Jedną z metod udostępnianych przez `List<T>` jest `GetEnumerator()`. Wymieniona metoda pochodzi z interfejsu `IEnumerable<T>`. W rzeczywistości to jest jedyna metoda tego interfejsu.

Dzięki posiadaniu małych interfejsów (interfejsy kilku powiązanych ze sobą metod), jak w omawianym przykładzie typu `List<T>`, można wybrać to, co powinno być zaimplementowane, i tylko to.

Zasada rozdzielania interfejsów i testy jednostkowe

Testy jednostkowe nie mają żadnego bezpośredniego wpływu na zasadę rozdzielania interfejsów. Mimo to wspomniałem tutaj o niej, aby przedstawić pełny obraz sytuacji.

Zasada odwrócenia zależności

Zasada odwrócenia zależności określa, że moduły wyższego poziomu nie powinny zależeć od modułów niskiego poziomu. W obu przypadkach moduły powinny zależeć od abstrakcji, które z kolei nie powinny mieć zależności od szczegółów. To szczegóły powinny być zależne od abstrakcji. Innymi słowy, zasada odwrócenia zależności promuje luźne powiązania między klasami z wykorzystaniem abstrakcji i wstrzykiwania zależności.

Przykład

W poprzednim rozdziale skoncentrowałem się na tym zagadnieniu i przedstawiłem w nim przykłady modyfikacji kodu pozwalające zapewnić obsługę mechanizmu wstrzykiwania zależności.

Zasada odwrócenia zależności i testy jednostkowe

Istnieje ścisły związek między zasadą odwrócenia zależności a testami jednostkowymi. Rzeczywiste testy jednostkowe nie mogą funkcjonować bez wstrzykiwania zależności. Tak naprawdę wysiłek poczyniony w celu zapewnienia obsługi wstrzykiwania zależności i przygotowania poprawnych projektów interfejsów dla klas pozbawionych interfejsów niejako jako produkt uboczny promuje zasadę odwrócenia zależności.

Można się przekonać, że testy jednostkowe promują regułę pojedynczej odpowiedzialności i zasadę odwrócenia zależności. Zatem nie tylko zwiększasz jakość tworzonego

kodu produkcyjnego, ale również jakość projektu. Nie ulega wątpliwości, że testy jednostkowe wymagają pewnego wysiłku, ale jedną z korzyści jest lepsza jakość projektu oraz większa czytelność kodu źródłowego.

Podsumowanie

W tym rozdziale omówiłem podstawy testów jednostkowych. Zamieściłem również kilka przykładów.

Jeżeli miałbym skategoryzować w skali od 1 do 5 doświadczenie związane z testami jednostkowymi, wówczas 1 określałoby początkującego, 5 zaś eksperta. Dzięki lekturze tego rozdziału znajdujesz się na poziomie 2. Bez obaw! Po zapoznaniu się z pozostałą częścią książki i bardziej rzeczywistymi przykładami znajdziesz się na poziomie 4. Jestem z dumny z postępu, jaki udało Ci się dotychczas osiągnąć. Tak trzymaj!

Już słyszę, jak pytasz, *czy lektura tej książki pozwoli Ci przejść na poziom 5.* Testy jednostkowe to jednak nie jest sprint, raczej maraton. Osiągnięcie poziomu 5 wymaga lat praktyki. Tutaj przedstawiłem jedynie krótkie wprowadzenie do tematu.

W rozdziale omówiłem także związek zachodzący między regułami SOLID a testami jednostkowymi, aby pokazać pełny obraz sytuacji i wyjaśnić, jak wszystko doskonale ze sobą współdziała.

Celowo zrezygnowałem z przykładów wymagających dokładnego zrozumienia zagadnienia dublerów używanych podczas testów, a zamiast tego postarałem się przedstawić proste wprowadzenie do testów jednostkowych. Jednak w rzeczywistości większość testów jednostkowych będzie wymagała wspomnianych dublerów. Zatem w następnym rozdziale dokładniej omówię koncepcję dublerów używanych podczas testów jednostkowych.

Dalsza lektura

Aby dowiedzieć się więcej na tematy poruszone w rozdziale, zapoznaj się z wymienionymi tutaj zasobami.

- *Walkthrough: Create a code snippet in Visual Studio*: *https://learn.microsoft.com/en-us/visualstudio/ide/walkthrough-creating-a-code-snippet?view=vs-2022*
- *xUnit*: *https://xunit.net/*

TDD wprowadza dobre praktyki i doskonali programistę

Spośród wielu koncepcji tworzenia oprogramowania na szczególną uwagę zasługuje model programowania sterowanego testami — TDD. Zastosowanie podejścia TDD ułatwia utrzymanie wysokiej jakości kodu. Technika ta opiera się na dodawaniu funkcjonalności do produktu dopiero po utworzeniu i przeprowadzeniu testów jednostkowych. TDD coraz częściej jest wyborem szanowanych firm programistycznych.

W tej praktycznej książce przedstawiono zasady TDD na rzeczywistych przykładach z użyciem popularnych frameworków, takich jak ASP.NET Core i Entity Framework. Po zapoznaniu się z solidnym wprowadzeniem do koncepcji TDD dowiesz się, jak można używać Visual Studio 2022 do tworzenia aplikacji internetowej z wykorzystaniem Entity Framework, a także baz danych SQL Server i Cosmos DB. Nauczysz się też korzystać z różnych wzorców, takich jak repozytorium, usługi i budowniczy. Ponadto omówiono tu architekturę DDD i inne najlepsze praktyki stosowane podczas tworzenia oprogramowania, w tym reguły SOLID i wskazówki FIRSTHAND. Nie zabrakło przydatnych uwag o biznesowych aspektach podejścia TDD.

W tej książce między innymi:

- testy jednostkowe i mechanizm wstrzykiwania zależności
- NSubstitute: imitacje i dublery używane podczas testów
- zastosowanie TDD dla ASP.NET API, Entity Framework i baz danych
- tworzenie potoków ciągłej integracji za pomocą GitHub
- zaawansowane scenariusze używania imitacji
- korzyści z wdrażania podejścia TDD przez zespoły i firmy

Adam Tibi od ponad 22 lat pracuje z technologiami .NET, Python, stosem Microsoft i Azure. Zajmował się mentoringiem zespołów, projektowaniem architektury, promowaniem podejścia zwinnego, dobrymi praktykami w zakresie opracowywania oprogramowania i tworzeniem kodu. Dzięki zdobytemu doświadczeniu posiada rozległą wiedzę na temat programowania sterowanego testami.

Helion
helion.pl
HELION S.A.
ul. Kościuszki 1c
44-100 Gliwice
tel.: 32 230 98 63
helion@helion.pl
KOD KORZYŚCI
Sięgnij po więcej!
ISBN 978-83-289-1693-7
9 788328 916937
Cena: 79,00 zł
<packt>