Rozdział 6.
Testy jednostkowe

Jacek Matulewski

Jeszcze przed napisaniem kodu, a najpóźniej w trakcie jego pisania, powinniśmy przygotowywać sprawdzające go testy jednostkowe. Rozwijając opisaną w rozdziale 4. strukturę `Ulamek` nie zastosowałem się do tej zasady, nie chcąc mnożyć zbyt wielu nowych elementów. W zamian testowałem przygotowywany kod w prostych testach umieszczonych wprost w metodzie `Main`, wyświetlających wyniki w konsoli. Teraz nadrobimy tę zaległość i przygotujemy wygodniejsze do obsługi i zarządzania testy jednostkowe.

Projekt testów jednostkowych

Wczytajmy do Visual Studio 2013 rozwiązanie *UlamekDemo* z poprzedniego rozdziału. To w którym oprócz projektu aplikacji konsolowej `UlamekDemo` są również dwa projekty bibliotek DLL i PCL. Do tego rozwiązania dodamy kolejny projekt przeznaczony do testów jednostkowych. W tym celu:

1. W podoknie *Solution Explorer* rozwiń menu kontekstowe i wybierz *Add*, *New Project...*.
2. Pojawi się okno *Add New Project*, w którego lewym panelu wybierzmy kategorię projektów: *Visual C#*, *Test*.
3. W środkowym panelu zaznaczmy pozycję *Unit Test Project*.
4. Ustalmy nazwę projektu na *UlamekTestyJednostkowe* i kliknijmy *OK*.

Powstanie nowy projekt, a do edytora zostanie wczytany plik *UnitTest1.cs*, dodany do tego projektu. W pliku tym zdefiniowana jest przykładowa klasa `UnitTest1` z pustą metodą `TestMethod1`. Klasa ozdobiona jest atrybutem `TestClass`, a metoda `TestMethod`.

Przygotowania do tworzenia testów

W Visual Studio 2010, w menu kontekstowym edytora dostępne było bardzo wygodne polecenie *Create Unit Test...* pozwalające na tworzenie testów jednostkowych dla wskazanej w edytorze metody. W Visual Studio 2012 i 2013, w których zmodyfikowany został moduł odpowiedzialny za testy jednostkowe, to wygodne polecenie zniknęło. W zamian zmuszeni jesteśmy do ręcznego programowania testów.

1. Aby przygotować projekt do testowania klasy `Ulamek` z biblioteki DLL *UlamekBiblioteka* lub biblioteki PCL *UlamekBibliotekaPrzenosna* dodaj referencję jednej z nich do utworzonego przed chwilą projektu. W tym celu z menu kontekstowego projektu *UlamekTestyJednostkowe* wybierz *Add*, *Reference...* i postępuj analogicznie, jak w poprzednim rozdziale.

Testy jednostkowe w Visual Studio wcale nie wymagają, aby testowany kod był zamknięty w bibliotece DLL lub PCL. W podany sposób można do projektu testów dodać również moduł programu *UlamekDemo.exe*.

2. Na początku pliku *UnitTest1.cs* dodaj polecenie `using Helion;`, aby klasa `Helion.Ulamek` była łatwo dostępna.

3. Zmieńmy nazwę pliku *UnitTest1.cs* na *UlamekTesty.cs*. Pojawi się pytanie o to, czy zmienić także nazwę klasy. Pozwólmy na to klikając *Tak*.

Projekt testów jednostkowych może mieć wiele klas. Dla wygody i przejrzystości warto dbać o to, żeby każda testowana klasa miała osobną klasę z testami.

Pierwszy test jednostkowy

Przygotujemy teraz pierwszy test jednostkowy, który będzie sprawdzał działanie konstruktora klasy `Ulamek` i jej własności `Licznik` i `Mianownik`. Teoretycznie rzecz biorąc, w metodzie, która przeprowadza test można wyróżnić trzy etapy: przygotowanie (ang. *arrange*), działanie (ang. *act*) i weryfikacja (ang. *assert*). Etapy te zaznaczone zostały w komentarzach widocznych na listingu 6.1. W praktyce granica między tymi etapami czasem się zaciera.

1. Przygotujmy dwie metody pomocnicze `losujLiczbeCalkowita` i `losujLiczbeCalkowitaRoznaOdZera` (listing 6.1), które będziemy wykorzystywali w testach. Korzystają one z generatora liczb losowych, który deklarujemy jako pole klasy `UlamekTesty`. Ani to pole, ani metody nie wymagają specjalnych atrybutów.

Dodatkowe pola wspomagające testy mogą wymagać specjalnej inicjacji lub innego typu przygotowania. Można do tego użyć dodatkowych metod opatrzonych specjalnymi atrybutami. I tak metoda z atrybutem `ClassInitialize` będzie uruchamiana na początku wszystkich testów, gdy tworzona jest instancja klasy testującej. W tej metodzie można by np. zainicjować generator liczb pseudolosowych. Natomiast metoda z atrybutem `TestInitialize` będzie uruchamiana przed pojedynczym testem. Tym atrybutom odpowiadają atrybuty `ClassCleanup` i `TestCleanup`. Opatrzone nimi metody uruchamiane są odpowiednio po wykonaniu wszystkich testów i po pojedynczym teście; służą do usunięcia zbędnych już obiektów lub przywrócenia pierwotnego stanu obiektów pomocniczych.

2. Następnie zmieńmy nazwę metody `TestMethod1` na `TestKonstruktoraIWlasnosci` i umieśćmy w niej kod z listingu 6.1. Pamiętajmy, że metoda testująca nie może zwracać wartości ani pobierać parametrów, a dodatkowo musi być ozdobiona atrybutem `TestMethod`.

Listing 6.1. Dwie metody pomocnicze i pierwsza metoda testująca

```
using System;
using Microsoft.VisualStudio.TestTools.UnitTesting;

using Helion;

namespace UlamekTestyJednostkowe
{
    [TestClass]
    public class UlamekTesty
    {
        Random r = new Random();

        private int losujLiczbeCalkowita(int? maksymalnaBezwzglednaWartosc = null)
        {
            if (!maksymalnaBezwzglednaWartosc.HasValue)
                return r.Next(int.MinValue, int.MaxValue);
            else
                return r.Next(-maksymalnaBezwzglednaWartosc.Value,
                              maksymalnaBezwzglednaWartosc.Value);
```

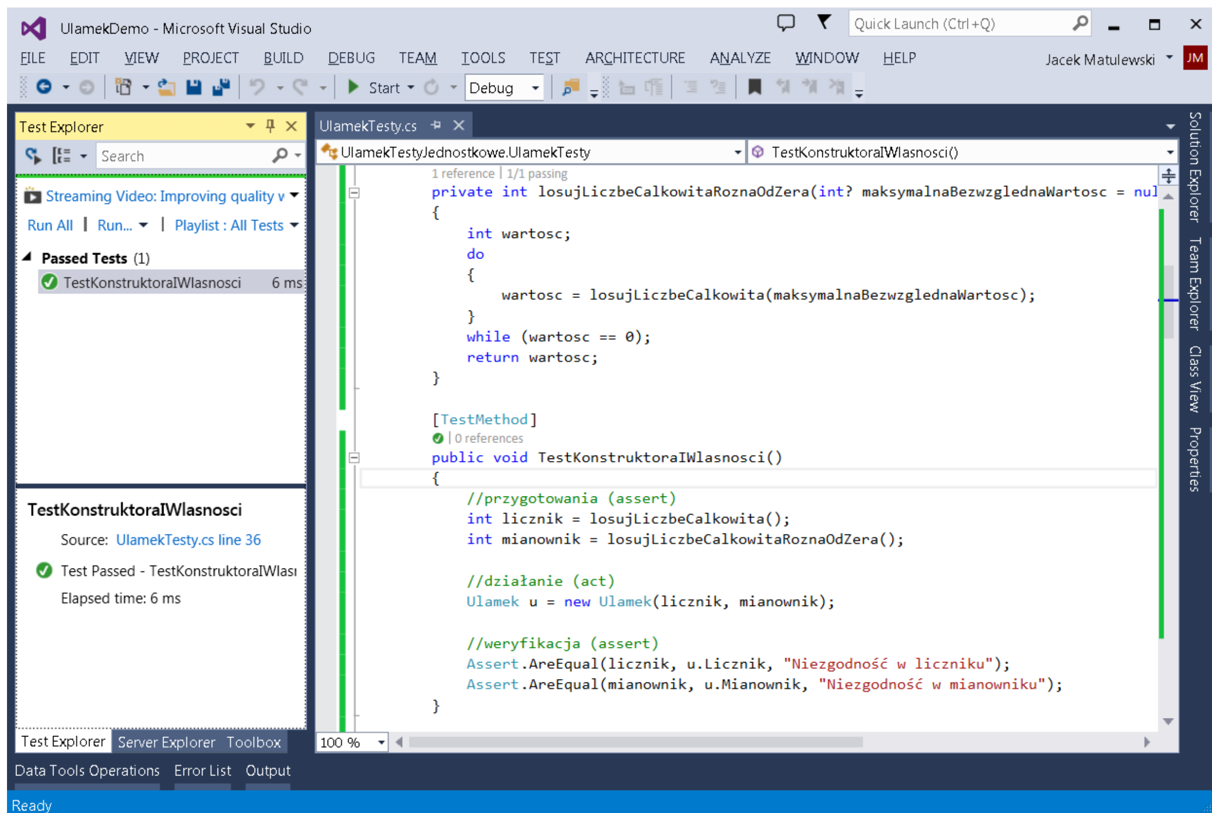
```
        }

        private int losujLiczbeCalkowitaRoznaOdZera(int? maksymalnaBezwzglednaWartosc = 
null)
        {
            int wartosc;
            do
            {
                wartosc = losujLiczbeCalkowita(maksymalnaBezwzglednaWartosc);
            }
            while (wartosc == 0);
            return wartosc;
        }

        [TestMethod]
        public void TestKonstruktoraIWlasnosci()
        {
            //przygotowania (assert)
            int licznik = losujLiczbeCalkowita();
            int mianownik = losujLiczbeCalkowitaRoznaOdZera();

            //działanie (act)
            Ulamek u = new Ulamek(licznik, mianownik);

            //weryfikacja (assert)
            Assert.AreEqual(licznik, u.Licznik, "Niezgodność w liczniku");
            Assert.AreEqual(mianownik, u.Mianownik, "Niezgodność w mianowniku");
        }
    }
}
```

Powyższy test należy do często używanej grupy testów, w których weryfikacja kodu polega na porównaniu jakiejś wartości otrzymanej w wyniku działania (drugi etap testu) z wartością oczekiwaną. W tego typu testach należy użyć wywołania statycznej metody `Assert.AreEquals`. Decyduje ona o powodzeniu testu. Wywołań może być wiele. Cały test powiedzie się, jeżeli we wszystkich wywołaniach wartość oczekiwana będzie równa uzyskanej w działaniu. I odwrotnie – jedna niezgodność powoduje, że cały test uznany zostanie jako „niezaliczony".

W przypadku, gdy w metodzie testującej jest kilka poleceń weryfikujących, warto użyć możliwości podania komunikatu, który wyświetlany jest w oknie *Test Explorer* w razie niepowodzenia testu. Możliwość ta jest jednak opcjonalna i metoda `Assert.AreEqual` może być wywoływana tylko z dwoma argumentami, czyli porównywanymi wartościami.

Uruchamianie testów

Sprawdźmy nasz test kompilując projekt testów (*Ctrl+Shift+B* lub *F6*). Aby uruchomić test, wybierzmy z menu *Test* polecenie *Run*, *All Tests*. Pojawi się wówczas nowe podokno Visual Studio o nazwie *Test Explorer* (widoczne z lewej strony na rysunku 6.1). W podoknie tym widoczne są wszystkie uruchomione testy oraz efekt przeprowadzonej w nich weryfikacji. Ikona pokazująca efekt weryfikacji widoczna jest również w edytorze kodu nad sygnaturą metody testującej, obok liczby wywołań.



Rysunek 6.1. Podokno Test Explorer – nowość z Visual Studio 2012. W Visual Studio 2013 niewiele się zmieniło

Dostęp do prywatnych pól testowanej klasy

Test konstruktora zaproponowany w listingu 6.1 ma zasadniczą wadę. Nie jest prawdziwym testem jednostkowym, bo jednocześnie testuje działanie konstruktora i własności `Licznik` i `Mianownik`. A co jeżeli zarówno w konstruktorze, jak i we własnościach są błędy, które wzajemnie się kompensują? Warto byłoby wobec tego oprócz powyższego testu przygotować także test, w którym konstruktor sprawdzany jest przez bezpośrednią weryfikację zainicjowanych w nim wartości pól `licznik` i `mianownik`. Ale jak to zrobić, skoro są one prywatne? Pomocą służy klasa `PrivateObject` z przestrzeni nazw *Microsoft.VisualStudio.TestTools.UnitTesting*, tej samej, w której zdefiniowana jest klasa `Assert`. Przykład jej użycia pokazuje listing 6.2. W nim do odczytu wartości prywatnego pola używam metody `PrivateObject.GetField`.

Listing 6.2. Weryfikowania wartości prywatnych pól

```
[TestMethod]
public void TestKonstruktora()
{
    //przygotowania (assert)
    int licznik = losujLiczbeCalkowita();
    int mianownik = losujLiczbeCalkowitaRoznaOdZera();

    //działanie (act)
    Ulamek u = new Ulamek(licznik, mianownik);

    //weryfikacja (assert)
    PrivateObject po = new PrivateObject(u);
```

```
        int u_licznik = (int)po.GetField("licznik");
        int u_mianownik = (int)po.GetField("mianownik");
        Assert.AreEqual(licznik, u_licznik, "Niezgodność w liczniku");
        Assert.AreEqual(mianownik, u_mianownik, "Niezgodność w mianowniku");
    }
```

Warto również zwrócić uwagę na metodę `PrivateObject.SetField` pozwalającą na zmianę prywatnego pola. Można jej użyć na przykład przy testowaniu własności w oderwaniu od konstruktora. Klasa `Private` posiada również metody `GetProperty` i `SetProperty` do testowania prywatnych własności oraz metodę `Invoke` do testowania prywatnych metod.

Dostęp do prywatnych elementów testowanej klasy w testach jednostkowych nie jest jednoznacznie oceniany. Niektórzy uważają, że testy jednostkowe powinny testować klasę w takim zakresie, w jakim jest ona dostępna dla innych modułów projektu, nie wnikając w szczegóły jej implementacji. Ja jednak uważam, że przydatne są wszystkie testy, które dają możliwość znalezienia błędu.

Testowanie wyjątków

Ułamek nie może mieć mianownika równego zeru. Próba ustawienia takiej wartości w konstruktorze powinna spowodować wywołanie wyjątku. To, czy rzeczywiście tak się stanie możemy sprawdzić w metodzie testującej widocznej na listingu 6.3.

Listing 6.3. Prowokowanie wyjątku w metodzie testującej

```
    [TestMethod]
    [ExpectedException(typeof(Exception))]
    public void TestKonstruktoraWyjatek()
    {
        Ulamek u = new Ulamek(losujLiczbeCalkowita(), 0);
    }
```

Przed metodą widoczny jest nowy atrybut `ExpectedException`. Jego parametrem jest oczekiwany typ wyjątku. Dzięki temu atrybutowi test się powiedzie, jeżeli w metodzie testującej zgłoszony zostanie wyjątek typu `Exception`.

Ponownie uruchommy testy, aby sprawdzić czy nowy test działa prawidłowo. Możemy to zrobić z okna *Test Explorer* klikając *Run All*. Testy uruchamiane są równolegle, co jest zarówno wygodne, jak i kłopotliwe. Wygodne, bo to w oczywisty sposób przyspiesza ich wykonanie, a dodatkowo pozwala sprawdzić ukryte zależności przy jednoczesnym dostępie do zasobów. Kłopotliwe bo utrudnia separację testów w takich przypadkach.

Kolejne testy weryfikujące otrzymane wartości

Idąc za ciosem przygotujmy kolejne testy (listing 6.4), w których sprawdzane są metody klasy `Ulamek` dla z góry ustalonych, przykładowych wartości licznika i mianownika.

Listing 6.4. Testowanie metod dla przykładowych wartości parametrów

```
    [TestMethod]
    public void TestPolaStatycznegoPolowa()
    {
        Ulamek uP = Ulamek.Polowa;
        Assert.AreEqual(1, uP.Licznik);
        Assert.AreEqual(2, uP.Mianownik);
    }
```

```
[TestMethod]
public void TestMetodyUprosc()
{
    Ulamek u = new Ulamek(4, -2);

    u.Uprosc();

    Assert.AreEqual(-2, u.Licznik);
    Assert.AreEqual(1, u.Mianownik);
}

[TestMethod]
public void TestOperatorow()
{
    Ulamek a = Ulamek.Polowa;
    Ulamek b = Ulamek.Cwierc;

    Assert.AreEqual(new Ulamek(3, 4), a + b, "Niepowodzenie przy dodawaniu");
    Assert.AreEqual(Ulamek.Cwierc, a - b, "Niepowodzenie przy odejmowaniu");
    Assert.AreEqual(new Ulamek(1, 8), a * b, "Niepowodzenie przy mnożeniu");
    Assert.AreEqual(new Ulamek(2), a / b, "Niepowodzenie przy dzieleniu");
}
```

Test ze złożoną weryfikacją

Jak sprawdzić poprawność sortowania dla typu `Ulamek`. Można oczywiście przygotować tabelę ze z góry ustalonymi wartościami i jej posortowaną wersję, która będzie wartością oczekiwaną używaną do weryfikacji. Co jednak zrobić, jeżeli chcemy przetestować działanie sortowania dla różnych, losowo wybranych wartości? Wówczas w etapie weryfikacji zamiast prostego porównania wartości oczekiwanej z uzyskaną, należy wykonać bardziej złożone operacje sprawdzające, czy uzyskana w wyniku sortowania tabela jest rzeczywiście prawidłowo posortowana. Pokazuje to przykład widoczny na listingu 6.5, w którym sprawdzam, czy w posortowanej tabeli każda kolejna wartość jest nie mniejsza niż poprzednia. Używam do tego zmiennej logicznej `tablicaJestPosortowanaRosnaco`, która inicjowana jest wartością `true`. Wartość ta zmieni się jedynie w przypadku wykrycia niemonotonicznej zmiany wartości w tabeli. O powodzeniu testu decyduje wywołanie metody `Assert.IsTrue`, która sprawdza, czy wartość zmiennej `tablicaJestPosortowanaRosnaco` pozostała równa `true`.

Listing 6.5. W tym przypadku etap weryfikacji nie jest prostym porównaniem wartości dwóch zmiennych

```
[TestMethod]
public void TestSortowania()
{
    Ulamek[] tablica = new Ulamek[100];
    for (int i = 0; i < tablica.Length; i++)
        tablica[i] = new Ulamek(losujLiczbeCalkowita(),
                                losujLiczbeCalkowitaRoznaOdZera());

    Array.Sort(tablica);
```

```
    bool tablicaJestPosortowanaRosnaco = true;
    for (int i = 0; i < tablica.Length - 1; i++)
        if (tablica[i] >= tablica[i + 1]) tablicaJestPosortowanaRosnaco = false;
    Assert.IsTrue(tablicaJestPosortowanaRosnaco);
}
```

Powtarzane wielokrotnie testy losowe

Choć testowanie działania metod lub operatorów dla wybranych wartości jest potrzebne i użyteczne, to konieczne jest również przeprowadzenie testów dla większego zakresu wartości parametrów, szczególnie w końcowym etapie prac nad klasą. Trudno jednak spodziewać się, że przygotujemy pętlę iterującą po wszystkich możliwych wartościach licznika i mianownika. To zajęło by wieki. Możemy się ratować testując metody klasy `Ulamek` dla wartości losowych – robiliśmy tak już w przypadku konstruktora. Żeby to miało jednak sens, należy takich testów wykonać wiele. Na tyle dużo, żeby losowe wartości pokryły cały zakres obu pól klasy `Ulamek`. Przykłady takich testów dla operatorów konwersji widoczne są na listingu 6.6.

Listing 6.6. Testy zawierające elementy losowe mogą być powtarzane w jednej metodzie

```
const int liczbaPowtorzen = 100;

[TestMethod]
public void TestKonwersjiDoDouble()
{
    for (int i = 0; i < liczbaPowtorzen; ++i)
    {
        int licznik = losujLiczbeCalkowita();
        int mianownik = losujLiczbeCalkowitaRoznaOdZera();
        Ulamek u = new Ulamek(licznik, mianownik);

        double d = (double)u;

        Assert.AreEqual(licznik / (double)mianownik, d);
    }
}

[TestMethod]
public void TestKonwersjiZInt()
{
    for (int i = 0; i < liczbaPowtorzen; ++i)
    {
        int licznik = losujLiczbeCalkowita();

        Ulamek u = licznik;

        Assert.AreEqual(licznik, u.Licznik);
        Assert.AreEqual(1, u.Mianownik);
    }
}
```

Wielokrotne powtarzanie testów, i co za tym idzie wielokrotne wywoływanie metod `Assert.AreEqual` lub `Assert.IsTrue`, nie naraża nas na zafałszowanie wyniku testu. Jak pamiętamy do „zaliczenia" testu niezbędne jest, żeby wszystkie wywołania tych metod potwierdziły poprawność kodu. Z kolei do uzyskania negatywnego wyniku wystarczy, że niezgodność pojawi się choćby w jednym z ich wywołań.

Niepowodzenie testu

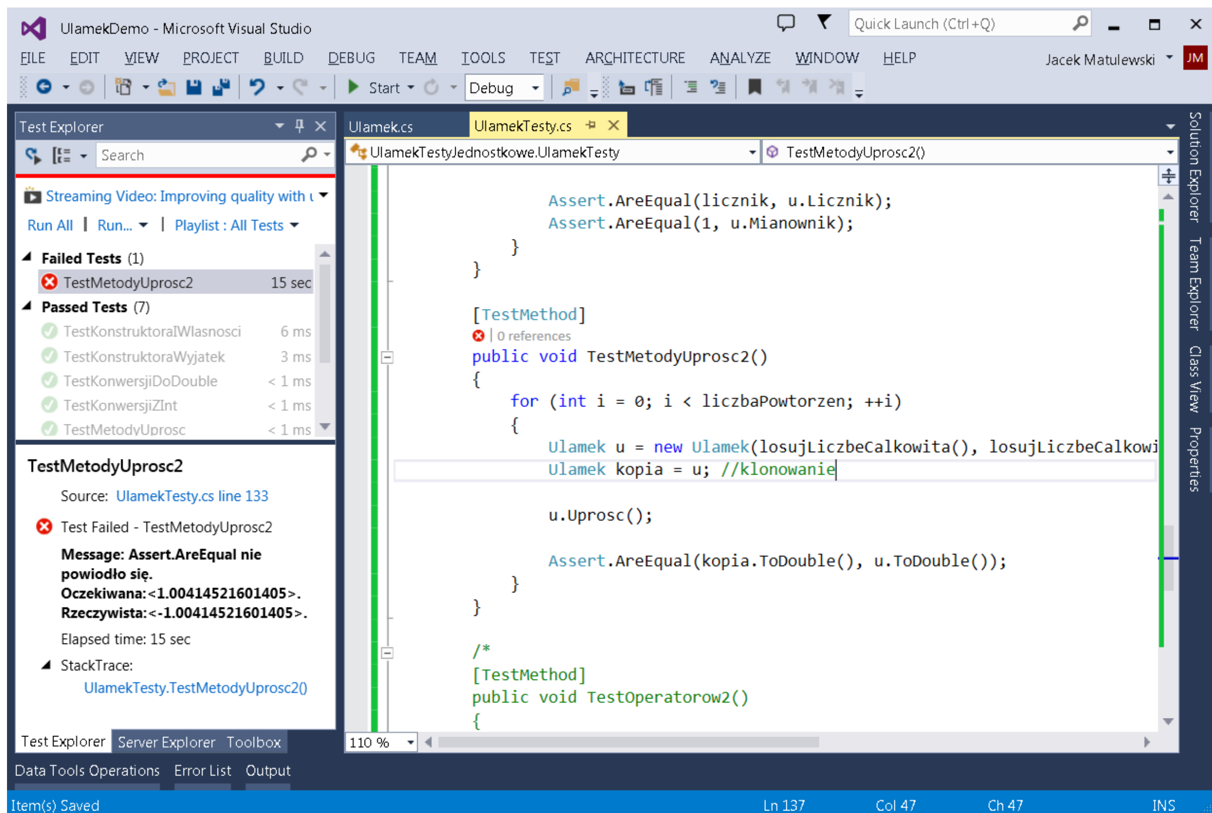
Celem testów jest znalezienie błędów logicznych w przygotowywanym kodzie. Wymaga to przeprowadzania jak największej liczby różnorodnych testów, nawet jeżeli wydaje się nam, że wszystkie błędy już znaleźliśmy[1]. Aby się o tym przekonać wykonajmy test z listingu 6.7. W teście tym tworzone są dwa równe sobie obiekty `Ulamek` o losowych wartościach licznika i mianownika. Następnie jeden z nich jest upraszczany metodą `Ulamek.Uprosc`, a następnie porównywane są wartości oby ułamków po ich konwersji do liczby rzeczywistej typu `double`. Testujemy zatem, czy uproszczenie ułamka nie spowodowało zmiany jego rzeczywistej wartości. Pamiętajmy jednak, że w przypadku losowo wybieranych wartości uproszczenie będzie możliwe dość rzadko, zatem stosunkowo niewielka część iteracji będzie miała rzeczywisty wkład do testu.

Listing 6.7. Test zakończony niepowodzeniem jest szansą na poprawienie testowanego kodu

```
[TestMethod]
public void TestMetodyUprosc2()
{
    for (int i = 0; i < liczbaPowtorzen; ++i)
    {
        Ulamek u = new Ulamek(losujLiczbeCalkowita(),
                              losujLiczbeCalkowitaRoznaOdZera());
        Ulamek kopia = u; //klonowanie

        u.Uprosc();

        Assert.AreEqual(kopia.ToDouble(), u.ToDouble());
    }
}
```

Uruchamiając ów test, przekonamy się, że klasa `Ulamek` go nie przejdzie (rysunek 6.2). Obok testu pojawi się czerwona ikona, a zaznaczając ów test na liście testów w podoknie Test Explorer możemy obejrzeć szczegóły informacji o niepowodzeniu. Okazuje się, że po konwersji ułamków do liczb typu `double` mają one różne znaki, choć są równej co do bezwzględnej wartości. Możemy powtórzyć ten test wielokrotnie (co ma sens przy losowo wybieranych wartościach), jednak wynik jest zawsze taki sam.

[1] Takie przekonanie jest naturalne u programisty, który tworzy kod. Stąd częsty brak zaangażowania w proces testowania i pokusa, żeby ten etap projektu „odbębnić" jak najszybciej. I dlatego najlepiej jest, jeżeli testowaniem nie zajmuje programista, który przygotowuje kod, a ktoś inny, najlepiej osoba wyspecjalizowana w przeprowadzaniu testów.



Rysunek 6.2. Szczegóły testu można obejrzeć po jego kliknięciu w podoknie Test Explorer

W takiej sytuacji nie od razu wiadomo, czy niepoprawny jest sprawdzany kod, czy sprawdzający go test. To drugie zdarza się zresztą nader często. Oszczędzę Czytelnikowi wspólnego szukania błędów i od razu wyjaśnię gdzie tkwi błąd. Problemem jest testowana metoda `Ulamek.Uprosc`, a dokładnie warunek sprawdzający znaki licznika i mianownika (wyróżniony na listingu 6.8). Jak pamiętamy z rozdziału 4. warunek ten ma na celu znalezienie par licznik i mianownika, w których tylko jedno z nich jest ujemne. Wówczas znak licznika ustalamy na minus, a mianownik pozostawiamy dodatni. Aby to sprawdzić w prosty i elegancki sposób, sprawdzamy znak iloczynu licznika i mianownika. Problem w tym, że dla dużych wartości licznika i mianownika, iloczyn łatwo może przekroczyć zakres liczb całkowitych typu `int` co prowadzi do zafałszowania wyników.

Listing 6.8. Metoda Uprosc z wyróżnionym problematycznym fragmentem kodu

```
public void Uprosc()
{
    //NWD
    int mniejsza = Math.Min(Math.Abs(licznik), Math.Abs(mianownik));
    for (int i = mniejsza; i > 0; i--)
    if ((licznik % i == 0) && (mianownik % i == 0))
    {
        licznik /= i;
        mianownik /= i;
    }

    //znaki
    if (licznik * mianownik < 0)
    {
        licznik = -Math.Abs(licznik);
        mianownik = Math.Abs(mianownik);
    }
```

```
        else
        {
            licznik = Math.Abs(licznik);
            mianownik = Math.Abs(mianownik);
        }
    }
```

Jak zmienić ów warunek tak, aby działał on prawidłowo dla dowolnych wartości licznika i mianownika? Najprostsze co możemy zrobić, to rozłożyć ów warunek na warunki elementarne:

```
    if ((licznik < 0 && mianownik > 0) || (licznik >= 0 && mianownik < 0))
    {
        ...
```

Można także pozostać przy iloczynie, ale zmniejszyć wartości czynników:

```
    if (Math.Sign(licznik) * Math.Sign(mianownik) < 0)
    {
```

Oba rozwiązania są poprawne i dzięki nim metoda `Ulamek.Uprosc` zda wreszcie test. Test ten będzie teraz trwał znacznie dłużej bo minuty zamiast sekund (czas procesora zużywa szukanie największego wspólnego dzielnika używane w metodzie `Uprosc`). Wcześniej test przerywała bowiem pierwsza znaleziona niezgodność, teraz wykonywany jest pełen zestaw stu powtórzeń.

Nieuniknione błędy

O ile w przypadku metody upraszczającej ułamek przekroczenie zakresu należy traktować jak błąd, to w przypadku operatorów arytmetycznych jest to nie do uniknięcia. Wówczas to nie działanie tych operatorów, ale ich użycie dla zbyt dużych wartości argumentów, może być uważane za błąd. Można się oczywiście zastanawiać, czy operatory nie powinny zwracać typu o większym zakresie. Tego rozwiązania się jednak nie praktykuje, bo prowadzi do kolejnych trudności. W końcu iloczyn dwóch liczb całkowitych typu `int` jest wartością typu `int` nawet, gdyby miało to prowadzić do przekroczenia zakresu[2]. Podobnie jest w przypadku typu `Ulamek`. Dlatego testowanie operatorów arytmetycznych dla losowo wybieranych wartości licznika i mianownika z całego zakresu typu `int` nie miałoby sensu – z pewnością skończyłoby się to przekroczeniem zakresu, a tym samym niepowodzeniem testu. Zakres testowanych wartości należy wobec tego ograniczyć. Ograniczenie na możliwe wartości liczników i mianowników sumowanych, odejmowanych, mnożonych i dzielonych ułamków wynika wprost z definicji ich operatorów. We wszystkich pojawiają się iloczyny liczników i mianowników (listing 4.***). Zatem pierwszym ograniczeniem jest, że nie mogą one być jednocześnie większe od pierwiastka kwadratowego maksymalnej wartości typu `int` (czyli `Math.Sqrt(int.MaxValue)`). Bardziej restrykcyjne ograniczenie dotyczy tylko operatorów dodawania i odejmowania, i związane jest obliczeniem liczników. Dla przykładu w operatorze dodawania licznik jest równy sumie dwóch iloczynów liczników i mianowników. Wynika z tego, że liczniki i mianowniki dodawanych ułamków nie mogą być jednocześnie większe od pierwiastka z połowy maksymalnej wartości typu `int` (`Math.Sqrt(int.MaxValue/2)`). I właśnie do takich liczb ograniczyłem się w teście widocznym na listingu 6.9. Można oczywiście pójść nieco dalej. W końcu, jeżeli wylosowany licznik pierwszego składnika nie jest duży, to większe mogą być pozostałe trzy wartości. Można zatem maksymalną wartość mianownika pierwszego ułamka uzależnić od wylosowanej wcześniej wartości licznika. Z kolei wartości te wpływają na dopuszczalne wartości licznika i mianownika drugiego ułamka. Metoda testująca stałaby się jednak zbyt zawiła, jak na wprowadzający charakter tej książki i dlatego postanowiłem zrezygnować z takiego rozwiązania.

Listing 6.9. Uwzględnienie bezpiecznego zakresu

```
    [TestMethod]
    public void TestOperatorow2()
    {
```

[2] Należy pamiętać, że przy domyślnych ustawieniach przekroczenie zakresu nie jest nawet sygnalizowane za pomocą wyjątku. Aby to uzyskać należy użyć słowa kluczowego `checked` (zob. rozdział 2).

```
        //ograniczenie maksymalnej wartosci
        int limit = (int)(Math.Sqrt(int.MaxValue / 2) - 1);

        for (int i = 0; i < liczbaPowtorzen; ++i)
        {
            Ulamek a = new Ulamek(losujLiczbeCalkowita(limit),
                                  losujLiczbeCalkowitaRoznaOdZera(limit));
            Ulamek b = new Ulamek(losujLiczbeCalkowita(limit),
                                  losujLiczbeCalkowitaRoznaOdZera(limit));

            double suma = (a + b).ToDouble();
            double roznica = (a - b).ToDouble();
            double iloczyn = (a * b).ToDouble();
            double iloraz = (a / b).ToDouble();

            Assert.AreEqual(a.ToDouble() + b.ToDouble(), suma,
                            "Niepowodzenie przy dodawaniu");
            Assert.AreEqual(a.ToDouble() - b.ToDouble(), roznica,
                            "Niepowodzenie przy odejmowaniu");
            Assert.AreEqual(a.ToDouble() * b.ToDouble(), iloczyn,
                            "Niepowodzenie przy mnożeniu");
            Assert.AreEqual(a.ToDouble() / b.ToDouble(), iloraz,
                            "Niepowodzenie przy dzieleniu");
        }
    }
```

Przekraczanie zakresu to jednak nie jedyny problem, jaki pojawi się przy okazji tego testu. Przekonajmy się o tym uruchamiając go. Okaże się, że uzyskamy wynik negatywny. Tym razem problemem nie jest jednak testowana klasa, a sam test. Porównujemy skonwertowany do liczb rzeczywistych typu `double` wynik działania operatorów i porównujemy do wyników odpowiednich operatorów, ale dla typu `double`. Należy wziąć pod uwagę, że oba typy, `Ulamek` i `double`, mają ograniczoną dokładność. Ograniczona jest najmniejsza możliwa do zapisania w nich wartość absolutna. W przypadku typu `Ulamek` jest to `1/int.MaxValue`, czyli `4.656612875245797eE10`. W przypadku typu `double` wartość tę można odczytać z własności `double.Epsilon`, która jest równa `4.94066E-324`. Należy się zatem spodziewać, że wyniki działania operatorów dla typów `Ulamek` i `double` będą są równe z dokładnością nie większą niż 10^{-10} (lub jak kto woli 0,0000000001 lub `1E-10`). Na szczęście metoda `Assert.AreEqual` pozwala na uwzględnienie dokładności (listing 6.10). Jeżeli to zrobimy, wynik testu będzie pozytywny.

Listing 6.10. Skorygowana metoda testująca

```
    [TestMethod]
    public void TestOperatorow2()
    {
        //ograniczenie maksymalnej wartosci
        int limit = (int)(Math.Sqrt(int.MaxValue / 2) - 1);
        //dopuszczalna roznica w wyniku
        const double dokladnosc = 1E-10;

        for (int i = 0; i < liczbaPowtorzen; ++i)
        {
            Ulamek a = new Ulamek(losujLiczbeCalkowita(limit),
```

```
                      losujLiczbeCalkowitaRoznaOdZera(limit));
        Ulamek b = new Ulamek(losujLiczbeCalkowita(limit),
                      losujLiczbeCalkowitaRoznaOdZera(limit));

        double suma = (a + b).ToDouble();
        double roznica = (a - b).ToDouble();
        double iloczyn = (a * b).ToDouble();
        double iloraz = (a / b).ToDouble();

        Assert.AreEqual(a.ToDouble() + b.ToDouble(), suma, dokladnosc,
                "Niepowodzenie przy dodawaniu");
        Assert.AreEqual(a.ToDouble() - b.ToDouble(), roznica, dokladnosc,
                "Niepowodzenie przy odejmowaniu");
        Assert.AreEqual(a.ToDouble() * b.ToDouble(), iloczyn, dokladnosc,
                "Niepowodzenie przy mnożeniu");
        Assert.AreEqual(a.ToDouble() / b.ToDouble(), iloraz, dokladnosc,
                "Niepowodzenie przy dzieleniu");
    }
}
```

* * *

Pisanie testów jest trudnym, a jednocześnie często niedocenianym elementem projektów informatycznych. Wymaga wyobraźni, ogromnej skrupulatności i odpowiedzialności. I z reguły jest bardziej pracochłonne niż samo pisanie kodu. Zauważmy, że przedstawiony w tym rozdziale kod testów jest większy niż kod testowanej klasy, a to przecież nie wszystkie testy, jakie można wymyśleć. Nie wszystkie scenariusze użycia klasy `Ulamek` zostały już sprawdzone. Zwykle testów jest tak dużo i ich wykonanie zajmuje tyle czasu, że przeprowadza się je w nocy po połączeniu fragmentów kodu na serwerze wersji.