Samouczek testów jednostkowych w Visual Studio 2013

Jacek Matulewski
Materiały dla Podyplomowego Studium Programowania i Zastosowania Komputerów, sekcja Projektowanie i tworzenie aplikacji dla platformy .NET (pod patronatem Microsoft)

<LEAD>

Testowanie oprogramowania, niezależnie od tego czy traktowane jest jako osobny etap projektu, czy jako integralna część procesu wytwarzania kodu, jest tematem ogromnym. Wystarczy wspomnieć rodzaje testów, jakie mogą i powinny być przeprowadzone: testy funkcjonalne aplikacji, testy integracyjne stosowane gdy cały produkt składamy jest z modułów często tworzonych przez różne osoby, testy systemowe, które mają ułatwić późniejsze bezproblemowe wdrożenia, wreszcie testy wydajnościowe, testy wykonywane wspólnie z klientem i testy interakcji z innymi systemami działającymi u klienta. Na pierwszej linii powinny jednak zawsze stać testy jednostkowe, które warto tworzyć zawsze, bez względu na charakter i rozmiar projektu. To ten rodzaj testów, z którym powinien być „zaprzyjaźniony" nie tylko wyspecjalizowany tester oprogramowania, ale również „zwykły" koder, programista i projektant. Są one po prostu, przynajmniej w części, gwarancją poprawności kodu. Poniższy artykuł ma za zadanie wprowadzić Czytelnika, którym według mojego założenia jest początkujący programista, do tworzenia i zarządzania testami jednostkowymi w najnowszej wersji Microsoft Visual Studio.

</LEAD>

<RAMKA>

Opisane w tym artykule narzędzie nie jest dostępne w wersji Express środowiska Visual Studio. Wówczas warto zainteresować się NUnit, czyli narzędziem z rodziny xUnit przeznaczonym dla platformy .NET i języków zarządzanych.

</RAMKA>

Jeszcze przed napisaniem kodu, a najpóźniej równocześnie z jego tworzeniem, powinniśmy przygotowywać sprawdzające go testy jednostkowe. To prawda powszechnie znana i nader często lekceważona. Załóżmy więc, że zaniedbaliśmy tego obowiązku i chcemy teraz tę zaległość teraz nadrobić. Mamy zamiar przygotować testy jednostkowe dla gotowej, napisanej w C#, struktury `Ulamek` (listing 0). Struktura ta umieszczona została w bibliotece PCL (ang. *Portable Class Library*). Towarzyszy jej projekt aplikacji konsolowej, w której jest używana, a która z punktu widzenia testów jest zupełnie nieistotna.

Listing 0. Najważniejsze fragmenty kodu testowanej klasy Ulamek

```
using System;

namespace ProgramistaMag
{
    public struct Ulamek : IComparable<Ulamek>
    {
```

```
private int licznik, mianownik;

public Ulamek(int licznik, int mianownik = 1)
{
    if (mianownik == 0)
        throw new ArgumentException("Mianownik musi być różny od zera");
    this.licznik = licznik;
    this.mianownik = mianownik;
}

public static Ulamek Zero = new Ulamek(0);
public static Ulamek Jeden = new Ulamek(1);
public static Ulamek Polowa = new Ulamek(1, 2);
public static Ulamek Cwierc = new Ulamek(1, 4);

public override string ToString()
{
    return licznik.ToString() + "/" + mianownik.ToString();
}

public double ToDouble()
{
    return licznik / (double)mianownik;
}

public void Uprosc()
{
    int a = licznik;
    int b = mianownik;

    //NWD
    int c;
    while (b != 0)
    {
        c = a % b;
        a = b;
        b = c;
    }

    licznik /= a;
    mianownik /= a;

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

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

    #region Wlasnosci
    public int Licznik { get { return licznik; } set { licznik = value; } }
    public int Mianownik
    {
        get { return mianownik; }
        set //zapis
        {
            if (value == 0)
                throw new ArgumentException("Mianownik musi być różny od zera");
            mianownik = value;
        }
    }
    #endregion

    //operatory arytmetyczne
    public static Ulamek operator +(Ulamek u1, Ulamek u2)
    {
        Ulamek wynik = new Ulamek(
            u1.Licznik * u2.Mianownik + u2.Licznik * u1.Mianownik,
            u1.Mianownik * u2.Mianownik);
        wynik.Uprosc();
        return wynik;
    }

    public static Ulamek operator -(Ulamek u1, Ulamek u2) ...
    public static Ulamek operator -(Ulamek u) ...
    public static Ulamek operator *(Ulamek u1, Ulamek u2) ...
    public static Ulamek operator /(Ulamek u1, Ulamek u2) ...

    //operatory porównania
    public static bool operator ==(Ulamek u1, Ulamek u2)
    {
        return (u1.ToDouble() == u2.ToDouble());
    }

    public static bool operator !=(Ulamek u1, Ulamek u2) ...
    public override bool Equals(object obj) ...
```

```
            public override int GetHashCode() ...

            public static bool operator >(Ulamek u1, Ulamek u2) ...
            public static bool operator >=(Ulamek u1, Ulamek u2) ...
            public static bool operator <(Ulamek u1, Ulamek u2) ...
            public static bool operator <=(Ulamek u1, Ulamek u2) ...

            //operatory konwersji
            public static explicit operator double(Ulamek u) ...
            public static implicit operator Ulamek(int n) ...
            public int CompareTo(Ulamek u) ...
        }
    }
```

Projekt testów jednostkowych

Wczytajmy do Visual Studio 2013 rozwiązanie *UlamekDemo* dołączone do artykułu. Do dwóch obecnych w tym rozwiązaniu projektów dodamy teraz kolejny, przeznaczony dla klas, w których zdefiniujemy metody testów jednostkowych. W tym celu: w podoknie *Solution Explorer* rozwiń menu kontekstowe i wybierz *Add*, *New Project...*. Pojawi się okno *Add New Project*, w którego lewym panelu wybierz kategorię projektów: *Visual C#*, *Test*. W środkowym panelu zaznacz pozycję *Unit Test Project*. Zmień nazwę projektu na *UlamekTestyJednostkowe* i kliknij *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 – atrybutem `TestMethod`.

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 będziemy ręcznie przygotowywać metody testów. Aby przygotować projekt do testowania klasy `Ulamek` z biblioteki PCL (projekt *UlamekBibliotekaPrzenosna*) dodaj jej referencję do utworzonego przed chwilą projektu. W tym celu z menu kontekstowego projektu *UlamekTestyJednostkowe* wybierz *Add*, *Reference...* zaznacz bibliotekę w zakładce *Solution* i potwierdź klikając *OK*. Oczywiście testy jednostkowe w Visual Studio wcale nie wymagają, aby testowany kod był zamknięty w bibliotece DLL lub PCL. W podany wyżej sposób można do projektu testów dodać również moduł programu *.exe*. Na początku pliku *UnitTest1.cs* dodaj polecenie `using ProgramistaMag;`, aby klasa `ProgramistaMag.Ulamek` była łatwo dostępna. Zmieńmy także 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 1. W praktyce granica między tymi etapami czasem się zaciera.

Zmieńmy nazwę metody `TestMethod1` na `TestKonstruktoraIWlasnosci` i umieśćmy w niej kod widoczny na listingu 1. Inicjuje o ułamek z liczbami 1 i 2, a następnie sprawdza czy odczytane z tego ułamka za pomocą własności licznik i mianownik mają odpowiednie wartości. Test ten należy do chyba najczęściej używanego rodzaju testów, w którym weryfikacja 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.AreEqual`. Decyduje ona o powodzeniu testu. Może być wywoływana wielokrotnie. Wówczas o

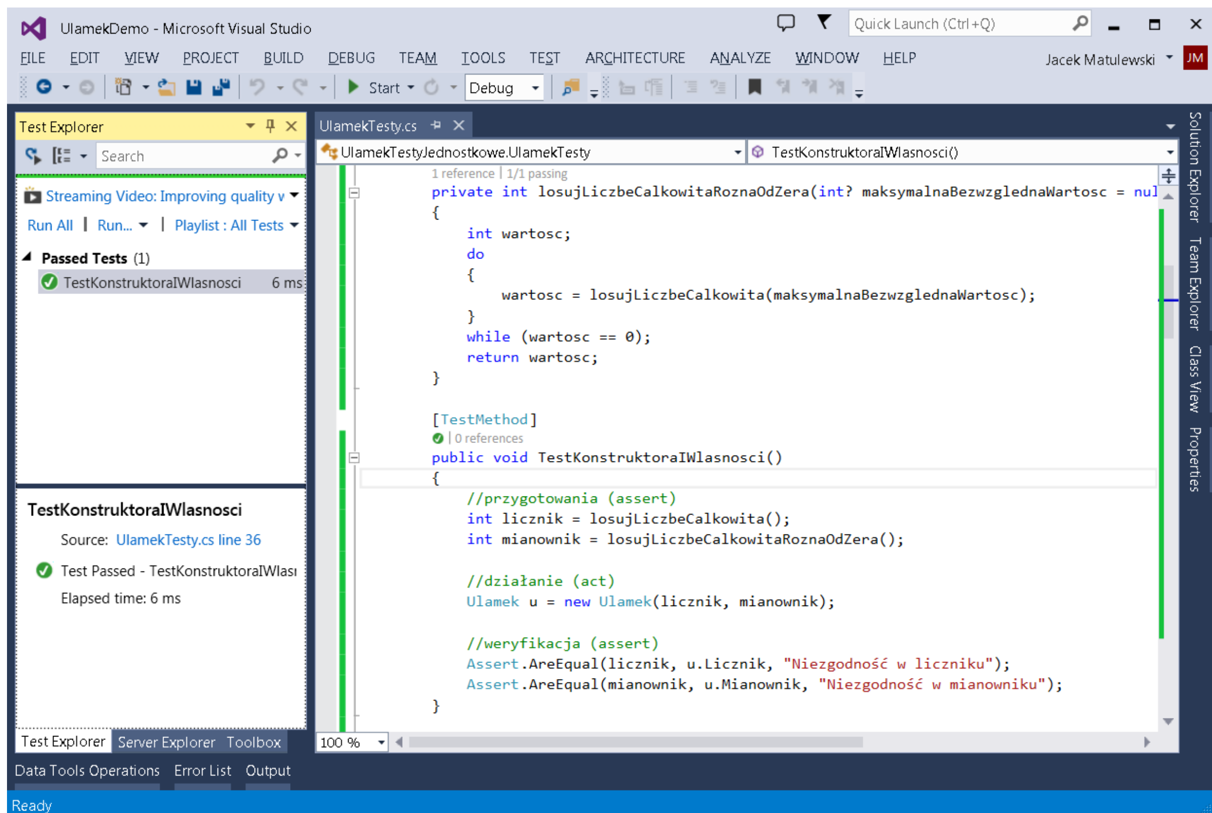
zaliczeniu całego testu decyduje zaliczenie wszystkich wywołań tej metody. I odwrotnie – jedna niezgodność powoduje, że cały test uznany zostanie jako „niezaliczony" i natychmiast przerwany. W przypadku, gdy w metodzie testującej jest kilka poleceń weryfikujących, warto użyć możliwości podania komunikatu, 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.

Listing 1. Metoda testująca nie może zwracać wartości ani pobierać parametrów, a dodatkowo musi być ozdobiona atrybutem TestMethod.

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

using Helion;

namespace UlamekTestyJednostkowe
{
    [TestClass]
    public class UlamekTesty
    {
        [TestMethod]
        public void TestKonstruktoraIWlasnosci()
        {
            //przygotowania (arrange)
            int licznik = 1;
            int mianownik = 2;

            //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");
        }
    }
}
```

Uruchamianie testów

Sprawdźmy nasz test kompilując rozwiązanie razem z projektem testów (*Ctrl+Shift+B* lub *F6*). Aby uruchomić test, wybierzmy z menu *Test* polecenie *Run*, *All Tests*. Pojawi się wówczas, choć po dłuższej chwili, nowe podokno Visual Studio o nazwie *Test Explorer* (widoczne z lewej strony na rysunku 1). W podoknie tym widoczne są wszystkie uruchomione testy oraz efekt przeprowadzonej w nich weryfikacji. W Visual Studio 2013 ikona pokazująca efekt weryfikacji widoczna jest również w edytorze kodu nad sygnaturą metody testującej, obok liczby wywołań.



Rysunek 1. Test Explorer – nowość Visual Studio 2012. W Visual Studio 2013 niewiele się zmienił

Dostęp do prywatnych pól testowanej klasy

Test konstruktora z listingu 1 ma zasadniczą wadę - testuje jednocześnie działanie konstruktora i własności `Licznik` i `Mianownik`. W razie niepowodzenia, nie wiemy który z tych elementów jest wadliwy. 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 2. W nim do odczytu wartości prywatnego pola używam metody `PrivateObject.GetField`. 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` służące do testowania prywatnych własności oraz metodę `Invoke` do testowania prywatnych metod.

Listing 2. Weryfikowanie 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");
    }
```

Dostęp do prywatnych elementów testowanej klasy w testach jednostkowych nie zawsze jest pożądany. W niektórych przypadkach testy jednostkowe powinny ograniczyć się do testowania klasy w takim zakresie, w jakim jest ona widoczna dla innych modułów projektu, nie wnikając w szczegóły jej implementacji (testy czarnej skrzynki).Natomiast na etapie tworzenia oprogramowania przydatne są wszystkie testy, które dają możliwość znalezienia błędu, także te, które zależą od szczegółów implementacji (testy białej skrzynki).

Testowanie wyjątków

Ułamek nie może mieć mianownika równego zeru. Próba użycia takiej wartości w konstruktorze powinna spowodować wywołanie wyjątku. To, czy rzeczywiście tak się stanie możemy sprawdzić, jeżeli metodę testującą poprzedzimy atrybutem `ExpectedException` (listing 3). Jego parametrem jest oczekiwany typ wyjątku – w naszym przypadku jest to `ArgumentException` – taki wyjątek jest zgłaszany w konstruktorze klasy `Ulamek`. Dzięki temu atrybutowi test się powiedzie, jeżeli w metodzie testującej zgłoszony zostanie wyjątek typu `ArgumentException` lub potomny.

Listing 3. Prowokowanie wyjątku w metodzie testującej

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

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. Brak separacji, w szczególności nieujawnione zależności mogą być zresztą bardzo trudne do wytropienia. Weźmy chociażby często używaną w testach klasę `Random` – wbrew temu, co mogłoby się wydawać, nie jest ona wcale bezpieczna ze względu na wątki. Przy dużej liczbie równoległych wywołań metoda `Random.Next` może zwracać niepoprawne wyniki. W takiej sytuacji lepiej korzystać z generatorów liczb pseudolosowych tworzonych lokalnie w metodach testujących, a inicjowanych ziarnem pobranym z generatora zdefiniowanego jako pole klasy testującej.

Test ze złożoną weryfikacją

Jak sprawdzić poprawność sortowania kolekcji z elementami typu `Ulamek`? Można oczywiście przygotować niewielką tabelę i jej posortowaną wersję, która będzie wartością oczekiwaną używaną do weryfikacji. Warto jednak 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 5, w którym sprawdzam, czy w 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`. Dodatkowo przygotujmy dwie

metody pomocnicze `losujLiczbeCalkowita` i `losujLiczbeCalkowitaRoznaOdZera` (także widoczne na listingu 5), które będziemy wykorzystywali w tym i kolejnych testach. Korzystają one z generatora liczb losowych, który deklarujemy jako pole klasy `UlamekTesty`. Ani to pole, ani metody nie wymagają specjalnych atrybutów (pamiętajmy jednak, że klasa `Random` nie jest w pełni bezpieczna przy współbieżnym uruchamianiu testów.

<<RAMKA>>

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.

<</RAMKA>>

Listing 5. W tym przypadku etap weryfikacji nie jest prostym porównaniem wartości dwóch zmiennych

```
Random r = new Random();

private int losujLiczbeCalkowita(int? maksymalnaBezwzglednaWartosc = null)
{
    if (!maksymalnaBezwzglednaWartosc.HasValue)
        return r.Next(int.MinValue, int.MaxValue);
    else
    {
        maksymalnaBezwzglednaWartosc = Math.Abs(maksymalnaBezwzglednaWartosc.Value);
        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 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. Wielokrotne powtarzanie testów, i co za tym idzie wielokrotne wywoływanie metod `Assert.AreEqual` lub `Assert.IsTrue` nie naraża nas na zafałszowanie wyniku całego 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 nich.

Listing 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);
        }
    }
```

Niepowodzenie testu

<RAMKA>

Czy programista może być jednocześnie testerem swojego kodu? W naturalny sposób programista tworzący kod jest skłonny uważać, że kod jest poprawny i pozbawiony zasadniczych błędów. Stąd częsty brak zaangażowania w proces testowania i pokusa, żeby ten etap projektu „odbębnić" jak najszybciej. I dlatego dobrze jest, jeżeli testowaniem nie zajmuje programista, który przygotowuje kod, a ktoś inny, najlepiej osoba wyspecjalizowana w przeprowadzaniu testów. Z drugiej strony często można usłyszeć opinię zupełnie odwrotną, której też trudno odmówić słuszności: według niej każdy programista jest odpowiedzialny za własny kod, dlatego powinien sam pisać dla siebie testy jednostkowe, które pozwolą mu pilnować jego poprawności. Oba poglądy mogą być jednocześnie słuszne, bo dotyczą innych płaszczyzn. Pierwszy mówi o tym, jak jest (psychologia pracy programisty), a drugi o tym, jak być powinno (etyka programisty). W praktyce trzeba oba w jakimś zakresie połączyć w czym pomagają nowoczesne metodyki wytwarzania oprogramowania i pracy w zespołach.

<</RAMKA>

Według optymistów celem testów jednostkowych jest potwierdzenie poprawności kodu. Według realistów jest nim znalezienie ukrytych w nim błędów logicznych. W praktyce oba cele wymagają przeprowadzania jak największej liczby różnorodnych testów, nawet jeżeli wydaje się nam, że wszystkie błędy już znaleźliśmy (zob. ramka „Czy programista może być jednocześnie testerem swojego kodu?"). Jednych i drugich zachęcam do wykonania testu z listingu 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`. Sprawdzamy zatem, czy uproszczenie ułamka nie spowodowało zmiany jego rzeczywistej wartości. Pamiętajmy jednak, że w przypadku losowo wybieranych wartości uproszczenie będzie możliwe dość rzadko – wymaga to licznika i mianownika, które mogą być podzielone przez tę samą liczbę większą od jedności, zatem stosunkowo niewielka część iteracji będzie miała rzeczywisty wkład do testu.

Listing 7. Hasło programisty: „Test zakończony niepowodzeniem jest szansą na poprawienie testowanego kodu"

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

            u.Uprosc();

            Assert.IsTrue(u.Mianownik > 0);
            Assert.AreEqual(kopia.ToDouble(), u.ToDouble());
        }
    }
```

Uruchamiając ów test, przekonamy się, że klasa `Ulamek` go nie „zaliczy" (rysunek 2). Obok testu pojawi się czerwona ikona, a zaznaczając ów test na liście testów w podoknie *Test Explorer* możemy obejrzeć szczegóły

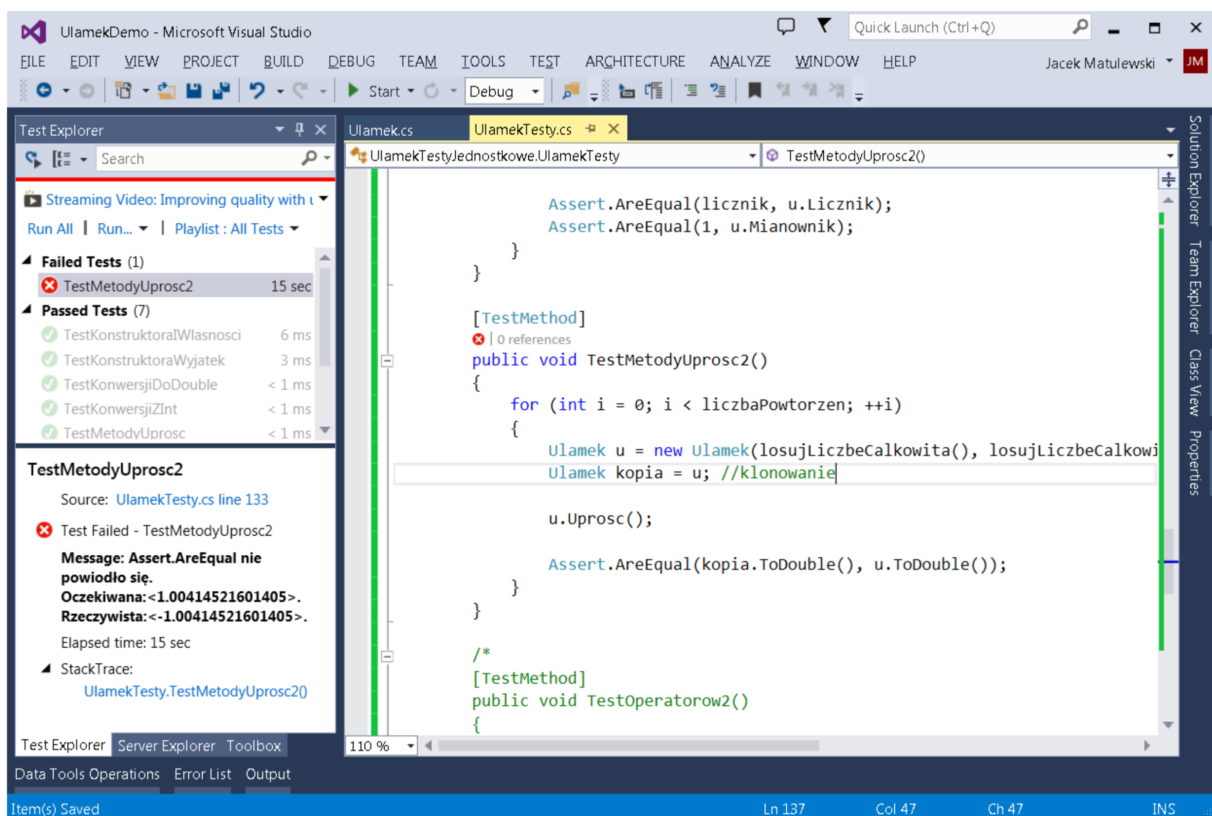
informacji o niepowodzeniu. Okazuje się, że po konwersji ułamków do liczb typu `double` mają one różne znaki, choć są równe co do bezwzględnej wartości. Możemy powtórzyć ten test wielokrotnie (co ma sens przy losowo wybieranych wartościach), jednak wynik zawsze będzie taki sam. W takiej sytuacji nie 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 (fragment wyróżniony na listingu 0). Znaki tych pól są różne, jeżeli ich iloczyn jest mniejszy od zera. Wówczas znak licznika ustalamy na minus, a mianownika na plus. Problem w tym, że dla dużych wartości licznika i mianownika, iloczyn łatwo może przekroczyć zakres liczb całkowitych typu `int` co prowadzi do zafałszowania wyników. Jak można poprawić ten warunek tak, aby działał 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 poprawiona metoda `Ulamek.Uprosc` „zda 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 pierwsza znaleziona niezgodność, teraz wykonywany jest pełen zestaw stu powtórzeń.



Rysunek 2. Szczegóły testu można obejrzeć po jego kliknięciu w podoknie Test Explorer

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 lub czy nie powinny zgłaszać wyjątków, gdy zakres zostanie przekroczony. Pierwszego rozwiązania się 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. (Należy pamiętać, że przy domyślnych ustawieniach przekroczenie zakresu nie jest nawet sygnalizowane za pomocą wyjątku. Aby to uzyskać należy użyć słowa kluczowego `checked` lub zmienić ustawienia całego projektu.) Podobnie jest w przypadku typu `Ulamek`. Kłopotliwy jest również drugi sposób tj. zgłaszanie wyjątku `OverflowException` w przypadku przekroczenia zakresu. Jest to nawet proste do realizacji – wystarczy umieścić mnożenia liczb typu całkowitego, jakie znajdują się w definicjach operatorów arytmetycznych w kontekście słowa kluczowego `checked`. Wówczas wyjątki takie będą zgłaszane. Dramatycznie wpłynie to jednak na wydajność tych operatorów. Idealnie byłoby, gdyby użytkownik klasy `Ulamek` mógł sam decydować o tym, czy przekroczenie zakresu ma być sprawdzane otaczając operacje na ułamkach konstrukcją `checked()`. To jednak nie jest możliwe do zrealizowania, ze względu na sposób implementacji tego słowa kluczowego. Jest ono przeznaczone wyłącznie dla typów całkowitych, dla których w języku MSIL (kod pośredni) dostępne są osobne instrukcje dla operacji arytmetycznych i dla operacji arytmetycznych z kontrolą zakresu. Użycie słowa kluczowego `checked` jest sygnałem dla kompilatora, że ma użyć tych drugich. Najbardziej eleganckim sposobem wydaje mi się wobec tego zdefiniowanie dodatkowych metod klasy `Ulamek`, które będą realizowały operacje arytmetyczne z kontrolą zakresu (listing 9) lub przełącznika, który decydowałby o tym, czy taką kontrolę stosować w przypadku operatorów. Natomiast najlepszym rozwiązaniem dla użytkownika klasy `Ulamek` w jej obecnej postaci jest włączenie na czas projektowania i testów kontroli przekraczania globalnie w ustawieniach projektu.

Listing 9. Dodawanie z kontrolą zakresu

```
public static Ulamek DodajOvf(Ulamek u1, Ulamek u2)
{
    Ulamek wynik =
        new Ulamek(checked(u1.Licznik * u2.Mianownik + u2.Licznik * u1.Mianownik),
                   checked(u1.Mianownik * u2.Mianownik));
    wynik.Uprosc();
    return wynik;
}
```

Jak widać testowanie operatorów arytmetycznych, w takiej postaci, w jakiej są one obecnie zdefiniowane, dla losowo wybieranych wartości licznika i mianownika z całego zakresu typu `int` nie ma sensu – z pewnością skończyłoby się to przekroczeniem zakresu, a tym samym negatywnym wynikiem 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 0). 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 z obliczanymi w nich licznikami. 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` (czyli `Math.Sqrt(int.MaxValue/2)`). I właśnie do takich liczb ograniczyłem się w teście widocznym na listingu 10. 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 testującą stałaby się jednak dość zawiła, co zmniejszałoby zaufanie do jej poprawności (a nie chcemy testować testów).

Listing 10. Uwzględnienie bezpiecznego zakresu liczników i mianowników w operacjach arytmetycznych

```
[TestMethod]
public void TestOperatorow_Losowy()
{
    //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 odpowiadających im operatorów dla typu `double`. Należy wziąć pod uwagę, że oba typy, `Ulamek` i `double`, mają ograniczoną dokładność. Ograniczona jest bowiem najmniejsza możliwa do zapisania w nich wartość absolutna. W przypadku typu `Ulamek` jest to `1/int.MaxValue`, czyli `4.656612875245797eE-10`. 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 11). To pozwoli wreszcie na prawidłowe przeprowadzenie testu.

Listing 11. Skorygowana metoda testująca

```
    [TestMethod]
    public void TestOperatorow_Losowy()
    {
        //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 artykule kod testów jest większy niż kod testowanej klasy, a to przecież tylko reprezentacja testów, jakie należy przeprowadzić. Nie wszystkie scenariusze użycia klasy `Ulamek` zostały już sprawdzone.

Chciałbym gorąco podziękować Tomkowi Dziubakowi za dyskusję nad tekstem artykułu i podpowiedzeniem zagadnień omówionych w ramkach.

<RAMKA>

Warto wspomnieć o częstej sytuacji, w której testowana klasa zależy od jakichś zewnętrznych obiektów lub danych np. odczytywanych z baz danych lub urządzeń fizycznych. Obiekty te lub dane mogą być trudne do kontroli, choćby dlatego że odczytywane z nich wartości zależą od jeszcze innych czynników, nie są deterministyczne, zależą od decyzji użytkownika programu lub po prostu nie są jeszcze przetestowane i tym same godne zaufania. Wówczas wygodne może być zastąpienie tych zewnętrznych kłopotliwych obiektów, od których zależy testowana klasa przez tzw. obiekty zastępcze, bardziej obrazowo nazywane „zaślepkami" (ang. *mock objects*), których stan i dynamikę możemy w pełni kontrolować. Szczególnie wygodne to jest w sytuacji, w której chcemy odtworzyć błąd występujący dla pewnego, trudnego do otworzenia w rzeczywistym obiekcie zewnętrznym stanu. Wówczas obiekty zastępcze są wręcz nieocenione. Pozwalają one również uniknąć sytuacji, w których nie jest jasne, co jest testowane i co jest źródłem ewentualnego błędu – testowana klasa czy obiekt zewnętrzny.

</RAMKA>

<LINKI>

Dodatkowe materiały w języku polskim:

http://msdn.microsoft.com/pl-pl/library/dd264975.aspx

http://wazniak.mimuw.edu.pl/images/e/e9/Zpo-3-wyk.pdf

</LINKI>