Wzorzec MVVM w WPF
Testy jednostkowe
(w Visual Studio 2017)

Jacek Matulewski

Podyplomowe Studium Programowania i Zastosowań Komputerów
Wydział Fizyki, Astronomii i Informatyki Stosowanej
Uniwersytet Mikołaja Kopernika

WWW: *http://www.fizyka.umk.pl/~jacek*

E-mail: *jacek@fizyka.umk.pl*

Wersja z **13 maja 2020 r.**
Poniższy tekst to rozdział 22 z książki pt. *Visual Studio 2017. Tworzenie aplikacji Windows w języku C#* wydanej w 2018 w wydawnictwie Helion

Wielką zaletą wzorca architektonicznego MVVM jest to, że zwiększa ilość kodu, który może być testowany, w szczególności testami jednostkowymi. W przypadku tego wzorca obowiązek testowania dotyczy nie tylko modelu, ale również modelu widoku, przynajmniej w pewnym stopniu. Możliwe jest nawet testowanie niektórych fragmentów widoku, np. konwerterów.

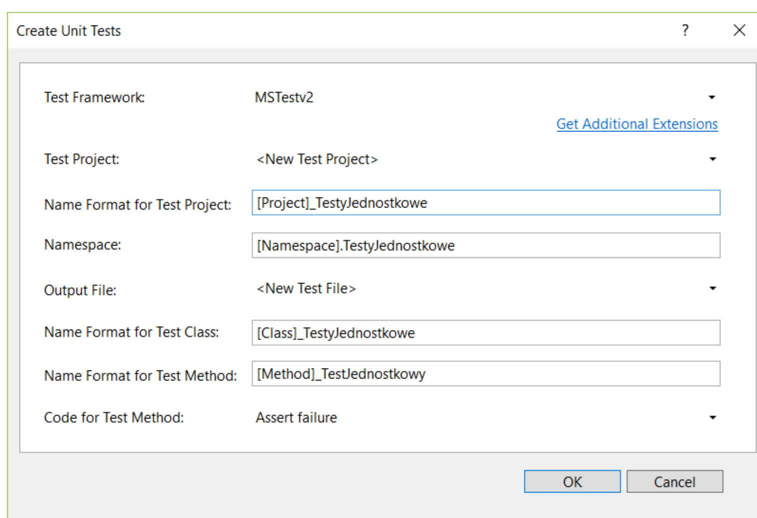
Testowanie oprogramowania, niezależnie od tego, czy traktowane jako osobny etap projektu, czy jako integralna część procesu wytwarzania kodu, jest tematem bardzo obszernym. Na pierwszej linii powinny jednak zawsze stać testy jednostkowe, które warto tworzyć bez względu na charakter i rozmiar projektu, a które mają za zadanie pilnować, aby kod w trakcie wielu zmian, jakie wprowadza się w projekcie w trakcie jego rozwoju, nie przestał robić tego, czego od niego oczekujemy. To ten rodzaj testów, z którym powinien być „zaprzyjaźniony" nie tylko wyspecjalizowany tester oprogramowania, ale również „zwykły" koder, programista i projektant. Są one, przynajmniej po części, gwarancją poprawności kodu, ale też fundamentem poczucia bezpieczeństwa w zespole zajmującym się rozwojem projektu.

Testy jednostkowe powinny powstawać równocześnie z zasadniczym kodem i powinny dotyczyć wszystkich metod i własności publicznych, a w niektórych przypadkach także prywatnej części klas. W poprzednich rozdziałach, pokazując konwersję projektu z architektury AV na MVVM, nie zastosowałem się do tej zasady. Teraz częściowo nadrobimy tę zaległość. Nie będę jednak przedstawiał wszystkich możliwych testów, jakie powinny być napisane dla aplikacji *KoloryWPF*. Tych dla nawet stosunkowo prostego projektu powinno być wiele. Przedstawię natomiast wybrane testy, które będą ilustrować kolejne zagadnienia związane z przygotowywaniem testów jednostkowych. Głównym celem tego rozdziału jest bowiem pomoc w rozpoczęciu testowania projektu — pokażę, jak utworzyć przeznaczony dla nich projekt i jak napisać pierwsze testy. Z pewnością nie jest to przewodnik po dobrych praktykach ani zbiór mądrościowych porad dotyczących testów.

Testy jednostkowe w Visual Studio 2015 i 2017

W Visual Studio 2015 do menu kontekstowego edytora kodu wróciły polecenia ułatwiające tworzenie testów jednostkowych. Przede wszystkim do dyspozycji mamy polecenie *Create Unit Tests*, które umożliwia utworzenie testu jednostkowego dla wybranej metody lub własności, a jeżeli to konieczne, także projektu dla testów. Poza tym w wersji Enterprise jest dostępne także polecenie *Create IntelliTest*, które umożliwia utworzenie zbioru testów dla całej klasy i przygotowuje ich standardowe fragmenty (zob. komentarz na ten temat poniżej).

1. Przejdźmy do pliku *Model\Kolor.cs*, ustawmy kursor edytora w konstruktorze klasy `Kolor` i z menu kontekstowego wybierzmy *Create Unit Tests*. Pojawi się okno widoczne na rysunku 22.1.



Rysunek 22.1. Kreator testów jednostkowych w Visual Studio 2015 i 2017

2. W rozwijanej liście *Test Framework* możemy wybrać platformę odpowiedzialną za zarządzanie testami i przeprowadzanie ich. Domyślnie jest to dostarczona razem z Visual Studio platforma MSTest, ale możliwe jest użycie innych, choćby popularnej NUnit.

3. Kolejna rozwijana lista pozwala na wybór istniejącego lub utworzenie nowego projektu testów jednostkowych. Ponieważ w bieżącym rozwiązaniu nie ma jeszcze takiego projektu, jedyną opcją będzie *<New Test Project>*.
4. W polu edycyjnym poniżej wpisujemy nazwę projektu. Nazwa może być dowolnym łańcuchem, ale możemy też wykorzystać nazwę bieżącego projektu i dodać do niego po znaku podkreślenia łańcuch „TestyJednostkowe”. Uzyskamy to, wpisując „[Project]_TestyJednostkowe”. W efekcie projekt testów będzie się nazywał *KoloryWPF_TestyJednostkowe*.
5. Przestrzeń nazw ustaliłem jako „[Namespace].TestyJednostkowe”, co spowoduje, że będzie ona miała postać *KoloryWPF.Model.TestyJednostkowe*.
6. W analogiczny sposób można wykorzystać nazwę zaznaczonej klasy i nazwę zaznaczonej metody do ustalenia nazwy klasy i nazwy metody zawierających testy (por. rysunek 22.1).
7. Wreszcie klikamy *OK*.

Efekt będzie taki, że kreator utworzy nowy projekt o nazwie *KoloryWPF_TestyJednostkowe*, a w nim katalog *Model* z plikiem *Kolor_TestyJednostkowe.cs*. Zaznaczenie w nazwie pliku tego, że znajdująca się w nim klasa zawiera testy jednostkowe, jest wygodne przy wielu zakładkach otwartych w edytorze. Do projektu zostanie dodana referencja do testowanego projektu, a do pliku — odpowiednia przestrzeń nazw. Wystarczy zastąpić instrukcję `Assert.Fail();` kodem widocznym w metodzie `TestKonstruktoraIWłasności` z listingu 22.1, aby pierwszy test był gotowy.

Listing 22.1. Klasa testów jednostkowych utworzona przez kreator w Visual Studio 2015 i 2017

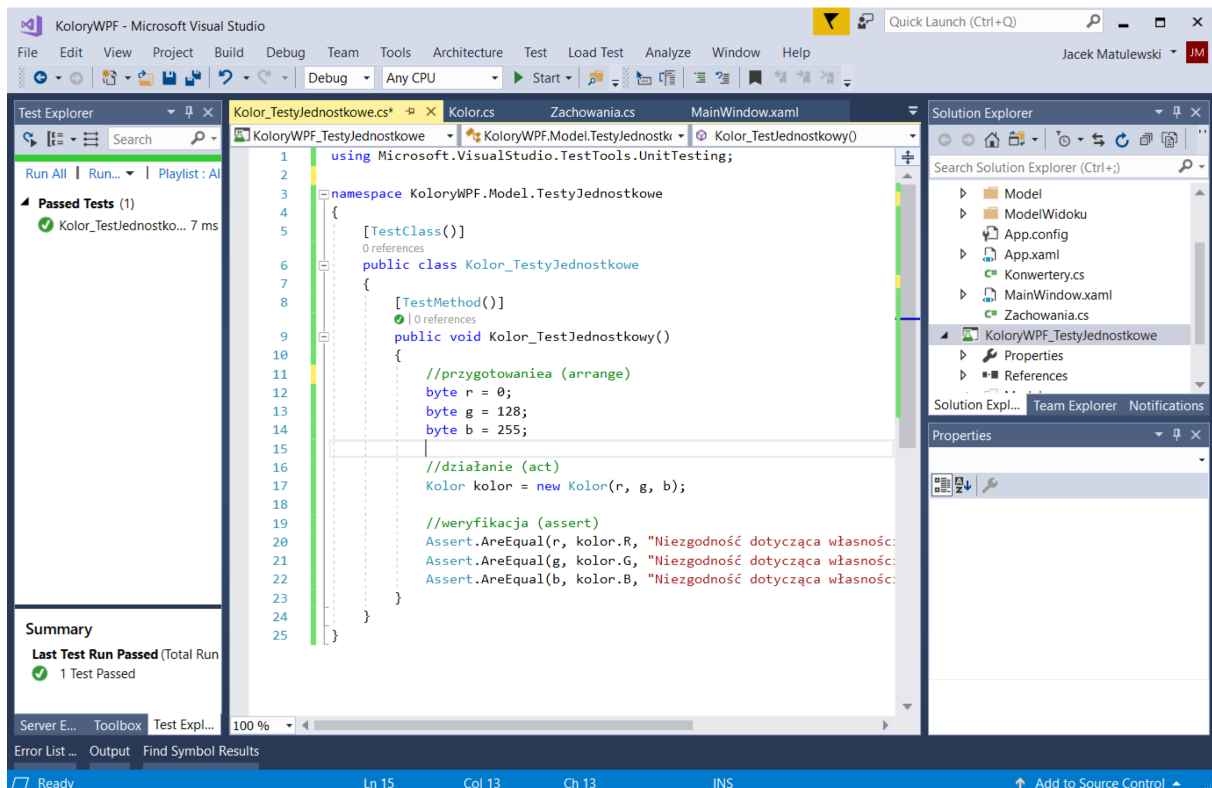
```
using Microsoft.VisualStudio.TestTools.UnitTesting;

namespace KoloryWPF.Model.TestyJednostkowe
{
    [TestClass()]
    public class Kolor_TestyJednostkowe
    {
        [TestMethod()]
        public void Kolor_TestJednostkowy()
        {
            Assert.Fail();

            //przygotowanie (arrange)
            byte r = 0;
            byte g = 128;
            byte b = 255;

            //działanie (act)
            Kolor kolor = new Kolor(r, g, b);

            //weryfikacja (assert)
            Assert.AreEqual(r, kolor.R, "Niezgodność dotycząca własności R");
            Assert.AreEqual(g, kolor.G, "Niezgodność dotycząca własności G");
            Assert.AreEqual(b, kolor.B, "Niezgodność dotycząca własności B");
        }
    }
}
```

Alternatywnym rozwiązaniem jest skorzystanie z polecenia menu kontekstowego edytora kodu *IntelliTests*, *Create IntelliTests*, które jest w stanie generować testy jednostkowe nie tylko dla pojedynczej metody, ale od

razu dla całej klasy, a konkretnie dla jej konstruktora oraz publicznych metod i własności. Testy, które powstają w ten sposób, zawierają kod tworzący instancję zwracaną przez metodę lub własność obiektu, należy jednak uzupełnić je o polecenia weryfikujące ich poprawność. Testy IntelliTest wymagają specjalnego projektu, nie można ich więc dołączać do utworzonego przed chwilą projektu testów jednostkowych. Można je uruchamiać poleceniem *Run IntelliTests* z menu kontekstowego edytora — inaczej niż zwykłe testy jednostkowe.

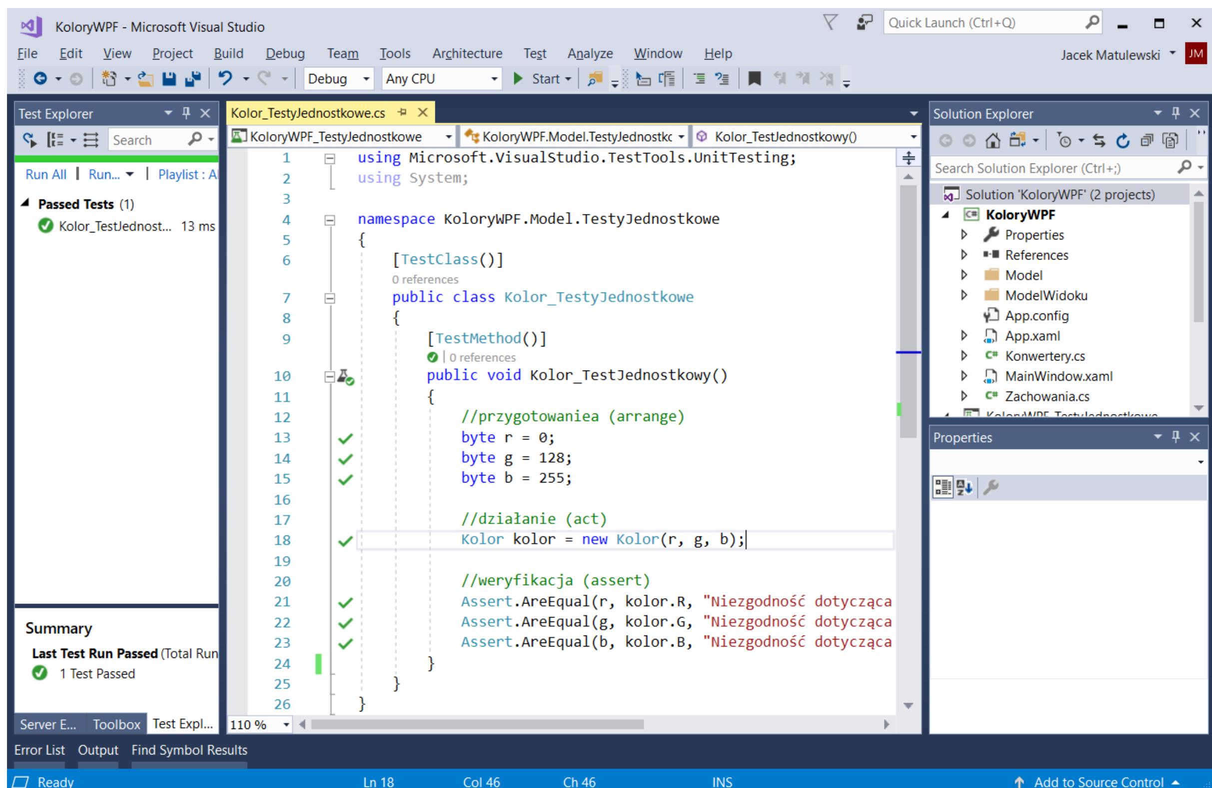
Uruchamianie testów

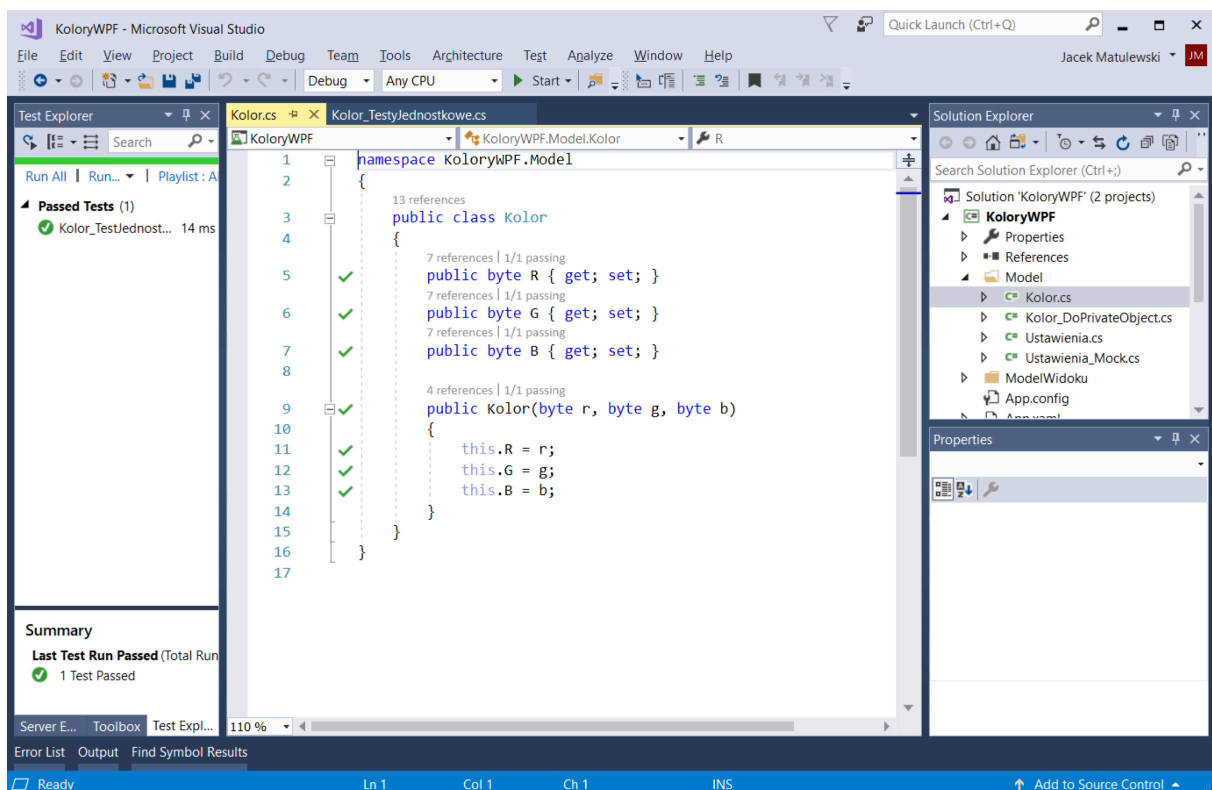
Sprawdźmy, czy kod naszego testu jest poprawny, kompilując całe rozwiązanie razem z projektem testów (*Ctrl+Shift+B* lub *F6*). Aby uruchomić test (nie dotyczy to testów IntelliTest), wybierzmy z menu *Test* polecenie *Run/All Tests*. Pojawi się wówczas wspomniane przed chwilą podokno o nazwie *Test Explorer* (z lewej strony na rysunku 22.2). W podoknie tym widoczne są wszystkie uruchomione testy i ich wyniki. W Visual Studio 2013 i 2015 ikona pokazująca efekt weryfikacji widoczna jest również w edytorze kodu nad sygnaturą metody testującej, obok liczby wywołań (jednak nie we wszystkich edycjach VS).



Rysunek 22.2. Podokno Test Explorer. W dolnej części rysunku widać ikony wstawiane przez mechanizm Live Unit Testing

W Visual Studio 2017 pojawiła się możliwość przeprowadzania testów jednostkowych „na żywo” w trakcie edycji kodu (zob. *https://docs.microsoft.com/en-us/visualstudio/test/live-unit-testing-intro*). Mechanizm ten możemy włączyć i wyłączyć z menu *Test*, *Live Unit Testing*. W efekcie w edytorze kodu z lewej strony pojawi się dodatkowa kolumna zawierająca ikony pokazujące wyniki testów, zarówno w klasie testującej, jak i w klasie testowanej (rysunek 22.3).





Rysunek 22.3. W edytorze kodu widoczna jest kolumna sygnalizująca wyniki przeprowadzanych „na żywo” testów jednostkowych

Testy wielokrotne

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. Oczywiście trudno się spodziewać, że zawsze będziemy w stanie przygotować pętlę iterującą po wszystkich możliwych wartościach pól testowanej klasy. W przypadku typów `int` lub `double` już dla jednego pola zajęłoby to o wiele za dużo czasu. Nawet w przypadku klasy `Kolor`, w której wszystkie trzy pola są typu `byte`, a więc przyjmują wartości od `0` do `255`, wszystkich możliwych stanów jest aż $256^3 =$ 16 777 216. To oznacza, że nawet w przypadku tak prostej klasy testowanie wszystkich możliwości (listing 22.2), choć daje pewność, że klasa poprawnie działa we wszystkich stanach, jest niepraktyczne, bo tak długiego testu nie można często powtarzać. Lepszym rozwiązaniem jest w tej sytuacji testowanie klasy dla wielu losowo wybranych składowych koloru. Musi być ich wystarczająco dużo, aby pokryły cały zakres możliwych wartości wszystkich pól (listing 22.3).

Listing 22.2. Powtórzenie testu dla wszystkich możliwych wartości trzech składowych koloru – ten test będzie trwał bardzo długo

```
[TestMethod]
public void TestKonstruktoraIWłasności_WszystkieWartości()
{
    for(byte r = 0; r <= 255; r++)
        for(byte g = 0; g <= 255; g++)
            for (byte b = 0; b <= 255; b++)
            {
                Kolor kolor = new Kolor(r, g, b);

                Assert.AreEqual(r, kolor.R, "Niezgodność dotycząca własności R");
                Assert.AreEqual(g, kolor.G, "Niezgodność dotycząca własności G");
                Assert.AreEqual(b, kolor.B, "Niezgodność dotycząca własności B");
            }
}
```

Listing 22.3. Testy zawierające elementy losowe mogą być powtarzane w jednej metodzie

```
private const int liczbaPowtórzeń = 100000;
private Random rnd = new Random();

[TestMethod]
public void TestKonstruktoraIWłasności_LosoweWartości()
{
    byte[] losoweWartościSkładowychKoloru = new byte[3 * liczbaPowtórzeń];
    rnd.NextBytes(losoweWartościSkładowychKoloru);

    for (int i = 0; i < liczbaPowtórzeń; i++)
    {
        byte r = losoweWartościSkładowychKoloru[3 * i];
        byte g = losoweWartościSkładowychKoloru[3 * i + 1];
        byte b = losoweWartościSkładowychKoloru[3 * i + 2];

        Kolor kolor = new Kolor(r, g, b);

        Assert.AreEqual(r, kolor.R, "Niezgodność dotycząca własności R");
        Assert.AreEqual(g, kolor.G, "Niezgodność dotycząca własności G");
```

```
            Assert.AreEqual(b, kolor.B, "Niezgodność dotycząca własności B");
        }
    }
```

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. W konsekwencji niezgodność choćby w jednym podteście powoduje negatywny wynik całego testu.

Dostęp do prywatnych pól testowanej klasy

Test konstruktora z listingów 22.1 – 22.3 ma zasadniczą wadę: testuje jednocześnie działanie konstruktora i własności klasy `Kolor`. W razie niepowodzenia nie wiemy, który z tych elementów jest wadliwy. Teoretycznie rzecz biorąc, możliwa jest też sytuacja, w której błędy kryją się zarówno w konstruktorze, jak i we własnościach i wzajemnie się kompensują. Oczywiście trudno to sobie wyobrazić w przypadku tak prostej klasy, jaką jest `Kolor`, w rozbudowanych klasach jest to jednak bardziej prawdopodobne. Warto byłoby wobec tego oprócz powyższego testu przygotować także test, w którym konstruktor jest sprawdzany poprzez bezpośrednią weryfikację zainicjowanych w nim wartości pól, oraz test własności sprawdzanych bez udziału konstruktora.

Warto byłoby się nauczyć, jak takie testy pisać. Problem w tym, że w klasie `Kolor` nie ma prywatnych pól — własności zdefiniowane są jako domyślnie zaimplementowane. Załóżmy jednak na chwilę, że zamiast korzystać z domyślnie implementowanych własności, użyliśmy klasycznego rozwiązania z prywatnymi polami, które przechowują wartości własności (listing 22.4). Wówczas moglibyśmy sprawdzić, czy konstruktor prawidłowo je inicjuje. 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 22.5. W nim do odczytu wartości prywatnego pola używam metody `PrivateObject.GetField`.

Listing 22.4. Klasa modelu z jawnie zdefiniowanymi prywatnymi polami przechowującymi wartości składowych koloru

```
namespace KoloryWPF.Model
{
    public class Kolor
    {
        private byte r, g, b;

        public byte R
        {
            get { return r; }
            set { r = value; }
        }
        public byte G
        {
            get { return g; }
            set { g = value; }
        }
        public byte B
        {
            get { return b; }
            set { b = value; }
        }
```

```
        public Kolor(byte r, byte g, byte b)
        {
            this.r = r;
            this.g = g;
            this.b = b;
        }
    }
}
```

Listing 22.5. Weryfikowanie wartości prywatnych pól

```
[TestMethod]
public void TestKonstruktora()
{
    byte r = 0;
    byte g = 128;
    byte b = 255;

    Kolor kolor = new Kolor(r, g, b);

    PrivateObject po = new PrivateObject(kolor);
    byte kolor_r = (byte)po.GetField("r");
    byte kolor_g = (byte)po.GetField("g");
    byte kolor_b = (byte)po.GetField("b");
    Assert.AreEqual(r, kolor_r, "Niezgodność dotycząca pola r");
    Assert.AreEqual(g, kolor_g, "Niezgodność dotycząca pola g");
    Assert.AreEqual(b, kolor_b, "Niezgodność dotycząca pola b");
}
```

Warto również zwrócić uwagę na metodę `PrivateObject.SetField`, umożliwiającą zmianę wartości prywatnego pola testowanej klasy (listing 22.6). Dzięki niej można ustawić wartość prywatnych pól i sprawdzić, czy własności poprawnie udostępniają ich wartości. Klasa `PrivateObject` ma również metody `GetProperty` i `SetProperty`, służące do testowania prywatnych własności, oraz metodę `Invoke`, pozwalającą testować prywatne metody.

Listing 22.6. Inicjacja prywatnych pól i testowanie własności udostępniających ich wartości

```
[TestMethod]
public void TestWłasności()
{
    byte r = 0;
    byte g = 128;
    byte b = 255;

    Kolor kolor = new Kolor(0, 0, 0);
    PrivateObject po = new PrivateObject(kolor);
    po.SetField("r", r);
    po.SetField("g", g);
    po.SetField("b", b);

    Assert.AreEqual(r, kolor.R, "Niezgodność dotycząca własności R");
    Assert.AreEqual(g, kolor.G, "Niezgodność dotycząca własności G");
```

```
        Assert.AreEqual(b, kolor.B, "Niezgodność dotycząca własności B");
    }
```

Klasa `PrivateObject` umożliwia dostęp do prywatnych składowych obiektu, który został wskazany w konstruktorze. Należy zwrócić uwagę na to, że jeżeli wskazany obiekt jest instancją struktury, to w konstruktorze następuje klonowanie i metodą `SetField` zmieniamy własności pól klona, a nie oryginału.

Przygotowywanie testów jednostkowych uzyskujących dostęp do prywatnych elementów testowanej klasy nie zawsze jest konieczne. W niektórych przypadkach testy jednostkowe powinny się ograniczyć do testowania klasy w takim zakresie, w jakim jest ona widoczna z 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).

Atrapy obiektów (mock objects)

Przejdźmy do testów klasy modelu widoku. Wiąże się z tym pewien zasadniczy problem, który często pojawia się w trakcie testów jednostkowych. Działanie tej klasy jest związane z działaniem klasy `Ustawienia`, która jest odpowiedzialna za trwałe przechowywanie stanu aplikacji. Możemy oczywiście podstawić do testów przygotowany plik i sprawdzić, czy model widoku ma po uruchomieniu odpowiedni stan. Wystarczy, że w pierwszej części metody testującej skopiujemy wcześniej przygotowany plik XML z ustawieniami do miejsca, z którego jest on czytany przez klasę `Ustawienia`. Wiąże się to jednak z problemami, takimi jak uzyskanie uprawnień do zapisu w folderze ustawień, kontrola ścieżek dostępu, które mogą zawierać bieżący klucz aplikacji, i w konsekwencji trudne do przewidzenia błędy po przeniesieniu projektu na inny komputer.

Aby tych problemów uniknąć, można na czas testów zastąpić całą klasę `Ustawienia` inną klasą, która tylko udaje odczyt danych z pliku, a tak naprawdę korzysta ze stałych przechowujących składowe koloru. Taki zastępczy obiekt jest nazywany atrapą (ang. *mock object*). Prosty przykład takiej klasy jest widoczny na listingu 22.7.

Listing 22.7. Atrapa klasy Ustawienia

```
public static class Ustawienia
{
    //magazyn
    public static byte r = 0;
    public static byte g = 128;
    public static byte b = 255;

    public static Kolor Czytaj()
    {
        return new Kolor(r, g, b);
    }

    public static void Zapisz(Kolor kolor)
    {
        r = kolor.R;
        g = kolor.G;
        b = kolor.B;
    }
}
```

Najbardziej elegancka byłaby możliwość „wstrzykiwania” obiektu odpowiedzialnego za przechowywanie ustawień (oryginalnej klasy `Ustawienia` we właściwym kodzie lub atrapy w testach) do modelu widoku np. poprzez argument konstruktora. Ale wówczas nie mogłyby to być statyczne klasy, jak jest w tej chwili, i

oczywiście należałoby zdefiniować dla nich wspólny interfejs, który obie by implementowały. Inną możliwością jest uczynienie z nich singletonów, których klasy byłyby przekazywane przez parametr klasy modelu widoku. Niestety, oba rozwiązania niezbyt współgrają z tym, że klasa `EdycjaKoloru` jest modelem widoku. Klasa modelu widoku jest tworzona w XAML za pomocą konstruktora domyślnego, co wyklucza pierwsze rozwiązanie. Utrudnione jest także użycie typu parametrycznego (ang. *generic type*) jako modelu widoku, co wyklucza także drugą z powyższych propozycji. Innym prostym rozwiązaniem jest kompilacja warunkowa. Jeżeli zdefiniowana byłaby stała-makro `TESTY`, kompilowana mogłaby być klasa-atrapa, w przeciwnym razie — „normalna” klasa ustawień. Innym sposobem, równie prostym, ale również wymagającym modyfikacji klasy modelu widoku, jest dodanie do tej klasy specjalnego konstruktora wykorzystywanego tylko w testach jednostkowych. Ponieważ to oznacza, że kompilator nie utworzy już konstruktora domyślnego, musimy go także dodać do klasy — jest on wymagany przez parser kodu XAML (nie wystarczy użyć wartości domyślnej w argumencie pierwszego konstruktora). Zastosowanie domyślnego konstruktora spowoduje użycie normalnej klasy ustawień, a dodatkowego — klasy atrapy. Poważną wadą takiego rozwiązania jest to, że w kodzie produkcyjnym umieszczany jest kod, który służy tylko do testowania, ja jednak to rozwiązanie wybrałem. Zmieniłem nazwę klasy z listingu 22.7 na `Ustawienia_Mock`, a do klasy `KoloryWPF.ModelWidoku.EdycjaKoloru` dodałem dwa konstruktory widoczne na listingu 22.8. Na listingu 22.9 widoczne są trzy testy korzystające z powyższej atrapy: test konstruktora klasy `EdycjaKoloru` i test zdefiniowanych w niej poleceń `Zapisz` i `Resetuj`. Zwróć uwagę, że instancja klasy modelu widoku tworzona jest w nich z podaniem argumentu konstruktora równego `true`.

Listing 22.8. Zmiany w klasie modelu widoku wprowadzone na potrzeby testów

```
public class EdycjaKoloru : ObservedObject
{
    private readonly Kolor kolor;

    public EdycjaKoloru()
    {
        kolor = Ustawienia.Czytaj();
    }

    //konstruktor używany w testach jednostkowych
    public EdycjaKoloru(bool użyjAtrapyUstawień = false)
    {
        if (!użyjAtrapyUstawień) kolor = Ustawienia.Czytaj();
        else kolor = Ustawienia_Mock.Czytaj();
    }

    ...
```

Listing 22.9. Kilka testów modelu widoku

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

namespace KoloryWPF.ModelWidoku.TestyJednostkowe
{
    [TestClass]
    public class EdytorKoloru_TestyJednostkowe
    {
        [TestMethod]
        public void TestKonstruktora()
        {
            EdycjaKoloru ek = new EdycjaKoloru(true);
```

```
            Assert.AreEqual(0, ek.R, "Niezgodność dotyczy własności R");
            Assert.AreEqual(128, ek.G, "Niezgodność dotyczy własności G");
            Assert.AreEqual(255, ek.B, "Niezgodność dotyczy własności B");
        }

        private const int liczbaPowtórzeń = 100000;
        private Random rnd = new Random();

        public void TestPoleceniaZapisz()
        {
            for (int i = 0; i < liczbaPowtórzeń; i++)
            {
                byte[] składoweKoloru = new byte[3];
                rnd.NextBytes(składoweKoloru);

                EdycjaKoloru ek = new EdycjaKoloru(true);
                ek.R = składoweKoloru[0];
                ek.G = składoweKoloru[1];
                ek.B = składoweKoloru[2];

                Assert.IsTrue(ek.Zapisz.CanExecute(null));
                ek.Zapisz.Execute(null);
                Assert.AreEqual(ek.R, KoloryWPF.Model.Ustawienia_Mock.r);
                Assert.AreEqual(ek.G, KoloryWPF.Model.Ustawienia_Mock.g);
                Assert.AreEqual(ek.B, KoloryWPF.Model.Ustawienia_Mock.b);
            }
        }

        [TestMethod]
        public void TestPoleceniaResetuj()
        {
            EdycjaKoloru ek = new EdycjaKoloru(true);

            Assert.IsTrue(ek.Resetuj.CanExecute(null),
                 "Niezgodność dotyczy metody CanExecute wywołanej przed zresetowaniem");
            ek.Resetuj.Execute(null);
            Assert.AreEqual(0, ek.R, "Niezgodność dotyczy własności R");
            Assert.AreEqual(0, ek.G, "Niezgodność dotyczy własności G");
            Assert.AreEqual(0, ek.B, "Niezgodność dotyczy własności B");
            Assert.IsFalse(ek.Resetuj.CanExecute(null),
                      "Niezgodność dotyczy metody CanExecute wywołanej po zresetowaniu");
        }
    }
}
```

Z atrap obiektów można korzystać w każdej sytuacji, w której testowana klasa zależy od jakichś zewnętrznych obiektów lub danych (np. odczytywanych z baz danych lub urządzeń fizycznych). Te obiekty 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 samym godne zaufania albo zwyczajnie nie są gotowe. Wielką zaletą atrap jest to, że możemy w pełni kontrolować ich zachowanie. Szczególnie wygodne jest to w sytuacji, w której chcemy odtworzyć błąd występujący dla pewnego trudnego do odtworzenia 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 stanowi źródło ewentualnego błędu — testowana klasa czy obiekt zewnętrzny.

Testowanie konwerterów

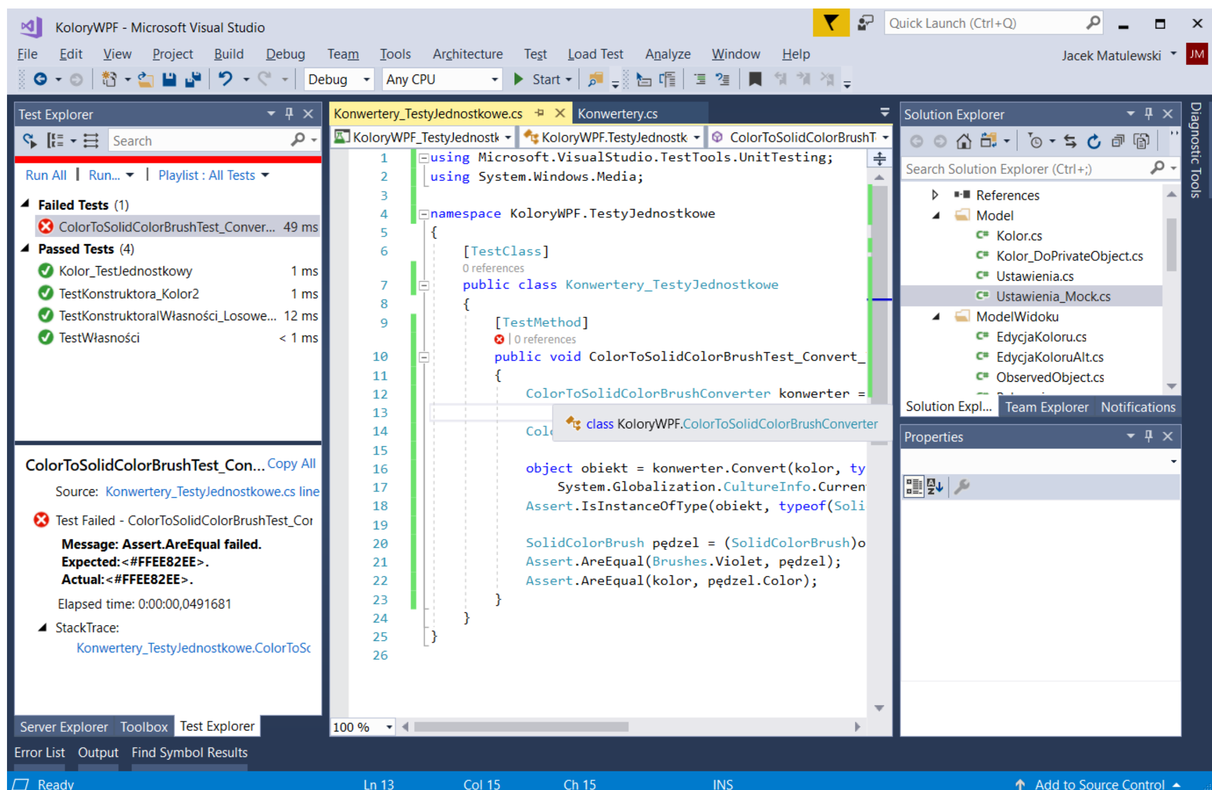
Testy jednostkowe modelu i modelu widoku są obowiązkowymi elementami poważnych projektów opartych na wzorcu MVVM. Ale testowane mogą być również niektóre klasy widoku. Dobrym przykładem są konwertery. To wdzięczny przedmiot testów — należy przygotować dane wejściowe, przeprowadzić konwersję i sprawdzić, czy uzyskaliśmy prawidłowy obiekt na wyjściu. Jako przykładu użyjmy konwertera `ColorToSolidColorBrushConverter`, przekształcającego wskazany kolor na pędzel typu `SolidColorBrush`. Jego przykładowy test jest widoczny na listingu 22.10. Warto rozważyć, czy dla testów klas należących do warstwy widoku nie utworzyć dodatkowego projektu, testy te wymagają bowiem dodania do projektu wielu referencji do bibliotek charakterystycznych właśnie dla widoku. W naszym przypadku konieczne będą referencje do *PresentationCore* (klasy `Color` i `Brush`), *PresentationFramework* (interfejs `IValueConverter`) i *WindowsBase*. Klasę testów konwertera umieściłem w osobnym folderze *Widok*.

Listing 22.10. Testy konwertera ColorToSolidColorBrushConverter

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

using KoloryWPF;
using System.Windows.Media;

namespace TestyJednostkowe.Widok
{
    [TestClass]
    public class Konwertery_TestyJednostkowe
    {
        [TestMethod]
        public void ColorToSolidColorBrushTest_Convert_TestJednostkowy()
        {
            ColorToSolidColorBrushConverter konwerter =
                new ColorToSolidColorBrushConverter();
            Color kolor = Colors.Violet;

            object obiekt = konwerter.Convert(kolor, typeof(SolidColorBrush), null,
                System.Globalization.CultureInfo.CurrentCulture);
            Assert.IsInstanceOfType(obiekt, typeof(SolidColorBrush));

            SolidColorBrush pędzel = (SolidColorBrush)obiekt;
            //Assert.AreEqual(Brushes.Violet, pędzel);
            Assert.AreEqual(kolor, pędzel.Color);
        }
    }
```

```
}
```

Zwróć uwagę na znajdujące się w komentarzu (przedostatnia linia metody) polecenie porównujące uzyskany w wyniku działania konwertera pędzel z pędzlem, który powinien być uzyskany po konwersji. Takie porównanie, choć naturalne, nie daje niestety poprawnego wyniku, pomimo że w komunikacie ewidentnie widać, że uzyskany kolor jest właściwy (rysunek 22.4). Dzieje się tak ze względu na dziwnie zaimplementowaną w przypadku pędzli metodę `Equals`. Dlatego aby obejść ten problem, zamiast całych pędzli porównuję, czy kolor pędzla uzyskanego z konwertera jest taki sam jak ten, który do niego przesłaliśmy.



Rysunek 22.4. Informacja o niepowodzeniu testu

Testowanie wyjątków

W konwerterze `ColorToSolidColorBrushConverter` jest także druga metoda, która jednak nie jest zaimplementowana — próba jej wykonania prowadzi do zgłoszenia wyjątku `NotImplementedException`. To, czy rzeczywiście taki wyjątek zostanie zgłoszony, możemy sprawdzić, jeżeli metodę testującą poprzedzimy atrybutem `ExpectedException` (listing 22.11). Jego parametrem jest oczekiwany typ wyjątku. Atrybut sprawia, że test się powiedzie, jeżeli w metodzie testującej zostanie zgłoszony wyjątek wskazany w parametrze lub po nim dziedziczący.

Listing 22.11. Prowokowanie wyjątku w metodzie testującej

```
[TestMethod]
[ExpectedException(typeof(NotImplementedException))]
public void ColorToSolidColorBrushTest_ConvertBack_TestJednostkowy()
{
    ColorToSolidColorBrushConverter konwerter = new ColorToSolidColorBrushConverter();
    konwerter.ConvertBack(null, null, null, null);
}
```

Ponownie uruchommy testy, aby sprawdzić, czy nowy test działa prawidłowo. Możemy to zrobić z okna *Test Explorer*, klikając *Run All*. Testy są uruchamiane równolegle, co jest szybkie i wygodne, ale może być kłopotliwe, jeżeli testy nie są całkowicie od siebie niezależne (zależą od stanu klasy testującej). Weźmy

chociażby często używaną w testach klasę `Random` — klasa ta nie jest bezpieczna ze względu na użycie w wielu wątkach. 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.

* * *

Celem testów jednostkowych jest szukanie ukrytych w kodzie błędów logicznych i pilnowanie poprawności kodu podczas zmian wprowadzanych do projektu. Oba te cele wymagają przygotowania jak największej liczby różnorodnych testów, nawet jeżeli wydaje się nam, że wszystkie błędy znaleźliśmy, a poważnych zmian już nie będzie. Takie przekonanie jest naturalne u programisty, który tworzy kod. Stąd częsty brak wystarczającego zaangażowania w proces testowania i pokusa, żeby ten etap projektu „odbębnić" jak najszybciej lub w ogóle „przeskoczyć". Na pośpiech często naciskają też kierownicy projektów i ich zwierzchnicy. Dlatego dobrze jest, jeżeli testowaniem nie zajmuje się programista, który przygotowuje kod, a ktoś inny, najlepiej osoba wyspecjalizowana w przeprowadzaniu testów lub w ogóle osobny dział testowania. Z drugiej strony, często można usłyszeć opinię zupełnie odwrotną, której też trudno odmówić słuszności, a mianowicie, że każdy programista jest odpowiedzialny za własny kod, dlatego powinien sam go testować, w szczególności pisać 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), 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.

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, a kod testów, pomimo jego stosunkowo niewielkiej złożoności, niejednokrotnie jest największą częścią projektu.