

Testy jednostkowe

Bartosz Miąskowski

Po co testować kod?

- Poprawne działanie programu dla jednego zestawu danych nie oznacza, że działa on poprawnie w każdej sytuacji.
- Ręczne testowanie często jest bardzo pracochłonne i czasochłonne.
- Zmiana kodu może powodować **regresję** – błąd w funkcjonalności, która wcześniej działała poprawnie.
- Testy automatyczne pozwalają wielokrotnie sprawdzać działanie kodu i porównywać wyniki z oczekiwaniami.



Czym są testy jednostkowe?

Test jednostkowy (*unit test*) to fragment kodu, który **automatycznie sprawdza konkretne zachowanie** niewielkiej części programu (*np. metody*):

- przygotowuje dane, wykonuje testowaną operację i **porównuje wynik z oczekiwanym**,
- działa w izolacji – nie wymaga uruchomienia całej aplikacji, łączenia z rzeczywistą bazą danych czy z serwerem,
- powinien być szybki, powtarzalny i niezależny od innych testów.



Dobrze napisany test jednostkowy powinien testować bardzo mały fragment programu.

Rodzaje testów

Rodzaj testu	Co sprawdza?	Przykład
Jednostkowy	Wybrany fragment logiki w izolacji.	Czy dla koszyka o wartości 150 zł rabat 10% daje kwotę 135 zł?
Integracyjny	Współpracę pomiędzy elementami systemu.	Czy zapisane zamówienie można poprawnie odczytać z bazy danych?
End-to-end (E2E)	Pełny scenariusz z perspektywy użytkownika.	Czy użytkownik może dodać produkt do koszyka, złożyć zamówienie i otrzymać potwierdzenie?



Testy jednostkowe sprawdzają poszczególne zachowania, a integracyjne i E2E pomagają sprawdzić, czy elementy aplikacji poprawnie współpracują.

Przygotowanie rozwiązania

Rozwiązanie posiadające testy jednostkowe z reguły ma następującą strukturę:

- biblioteka z logiką (np. *IPCaLcuLator.Core*) – zwykła biblioteka klas, w której znajduje się cała logika aplikacji,
- biblioteka z testami (np. *IPCalculator.Tests*) – biblioteka zawierająca testy jednostkowe,
- aplikacja dla użytkownika (np. *IPCalculator.App*) – aplikacja, która ma zostać dostarczona do użytkownika końcowego.

Takie podejście umożliwia **dokładne i szybkie testowanie** głównej części programu, która jest w całości zawarta w bibliotece klas, którą pozostałe projekty **referencjonują**.



Pierwszy test jednostkowy

Naszym zadaniem jest przetestowanie metody, służącej do konwersji zapisu maski sieci z formy skróconej (np. 24) do pełnej (np. 255.255.255.0).

```
using System;

namespace IpCalculator;

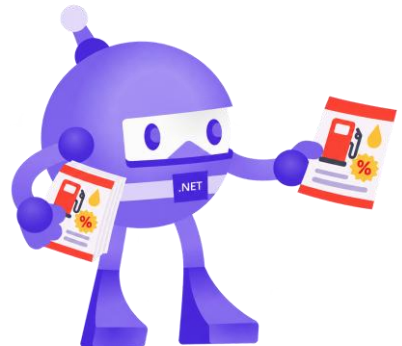
public class NetworkCalculator
{
    public string GetSubnetMask(int prefixLength)
    {
        if (prefixLength < 0 || prefixLength > 32)
        {
            throw new ArgumentOutOfRangeException(nameof(prefixLength));
        }

        byte[] octets = new byte[4];

        for (int bit = 0; bit < prefixLength; bit++)
        {
            int octetIndex = bit / 8;
            int bitPosition = 7 - bit % 8;

            octets[octetIndex] |= (byte)(1 << bitPosition);
        }

        return string.Join(".", octets);
    }
}
```



Pierwszy test jednostkowy

Test dzielimy na trzy części:

- **Arrange** – przygotowanie obiektu i danych wejściowych,
- **Act** – wykonanie testowanej operacji,
- **Assert** – sprawdzenie, czy wynik jest zgodny z oczekiwaniem.

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

namespace IpCalculator.Tests;

[TestClass]
public class NetworkCalculatorTests
{
    [TestMethod]
    public void GetSubnetMask_Prefix24_ReturnsCorrectMask()
    {
        // Arrange
        var calculator = new NetworkCalculator();
        int prefixLength = 24;
        string expected = "255.255.255.0";

        // Act
        string actual = calculator.GetSubnetMask(prefixLength);

        // Assert
        Assert.AreEqual(expected, actual);
    }
}
```



MSTests – atrybuty

Framework MSTests dostarcza użytkownikowi zestaw atrybutów, ułatwiających pisanie testów:

Atrybut	Zastosowanie
[TestClass]	Oznacza klasę zawierającą testy.
[TestMethod]	Oznacza metodę będącą testem.
[DataRow(...)]	Przekazuje zestaw argumentów do metody testowej. Pozwala sprawdzić wiele przypadków tym samym testem.
[TestInitialize]	Oznacza metodę wykonywaną przed każdym testem, np. przygotowującą obiekty.
[TestCleanup]	Oznacza metodę wykonywaną po każdym teście, np. zwalniającą zasoby.
[TestCategory("Mask")]	Przypisuje test do kategorii, ułatwiając grupowanie i filtrowanie.
[Ignore("Powód pominięcia")]	Powoduje pominięcie testu podczas uruchamiania.



Klasa i metoda testowa

Pisząc testy jednostkowe musimy pamiętać, że:

- testy zapisujemy w klasie oznaczonej atrybutem [TestClass],
- każdą metodę testową oznaczamy atrybutem [TestMethod],
- w podstawowym wariacie metoda testowa jest publiczna, niestaticzna, bez parametrów i zwraca void,
- nazwa metody powinna wskazywać testowaną operację, przypadek i oczekiwany rezultat.



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

namespace IpCalculator.Tests;

[TestClass]
public class NetworkCalculatorTests
{
    [TestMethod]
    public void GetSubnetMask_Prefix24_ReturnsCorrectMask()
    {
        // Tutaj umieszczamy kod testu.
    }
}
```

Asercje

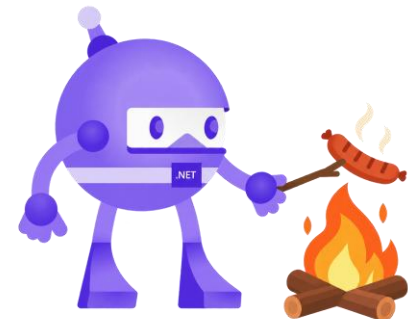
Asercja sprawdza, czy określone **oczekiwanie zostało spełnione**. Jeżeli nie – test zakończy się niepowodzeniem.

Metoda	Co sprawdza?
<code>Assert.AreEqual(expected, actual)</code>	Czy wartości są równe.
<code>Assert.AreNotEqual(notExpected, actual)</code>	Czy wartości są różne.
<code>Assert.IsTrue(condition)</code>	Czy warunek ma wartość true.
<code>Assert.IsFalse(condition)</code>	Czy warunek ma wartość false.
<code>Assert.IsNull(value)</code>	Czy wartość jest równa null.
<code>Assert.IsNotNull(value)</code>	Czy wartość jest różna od null.

```
var calculator = new NetworkCalculator();

string mask = calculator.GetSubnetMask(24);

// Pierwszy argument - expected value
// Drugi argument - wartość otrzymana
Assert.AreEqual("255.255.255.0", mask);
```



Wyjątki – testowanie błędnych danych

Dla niepoprawnych danych **oczekiwanym zachowaniem może być zgłoszenie wyjątku**. Metoda `Assert.ThrowsExactly<T>()` sprawdza, czy wskazana operacja zgłasza wyjątek dokładnie typu T – test kończy się niepowodzeniem, jeżeli wyjątek nie wystąpi lub będzie innego typu niż T.

```
[TestMethod]
public void GetSubnetMask_Prefix33_ThrowsArgumentOutOfRangeException()
{
    // Arrange
    var calculator = new NetworkCalculator();

    // Act & Assert
    Assert.ThrowsExactly<ArgumentOutOfRangeException>(() => calculator.GetSubnetMask(33)); // <-- delegata
}
```

Sprawdzamy czy dla zadanych danych metoda wyrzuci wyjątek `ArgumentOutOfRangeException`



Kolekcja wyników

Niektóre metody zwracają kolekcje – możemy sprawdzić ich zawartość, kolejność elementów oraz ich liczbę.

Asercja	Co sprawdza?
<code>Assert.AreSequenceEqual(expected, actual)</code>	Te same elementy w tej samej kolejności.
<code>Assert.AreSequenceEqual(expected, actual, SequenceOrder.InAnyOrder)</code>	Te same elementy i liczby ich wystąpień, niezależnie od kolejności.
<code>Assert.HasCount(4, actual)</code>	Czy kolekcja zawiera dokładnie cztery elementy.
<code>Assert.IsEmpty(actual)</code>	Czy kolekcja jest pusta.
<code>Assert.IsNotEmpty(actual)</code>	Czy kolekcja zawiera przynajmniej jeden element.

Uwaga!

Dwie różne tablice mogą zawierać takie same elementy. Do porównywania ich zawartości używamy asercji kolekcji, zamiast `Assert.AreEqual`.



```
string[] expected =  
{  
    "192.168.1.0/25",  
    "192.168.1.128/25"  
};  
  
string[] actual =  
{  
    "192.168.1.128/25",  
    "192.168.1.0/25"  
};  
  
// Niepowodzenie - inna kolejność elementów.  
Assert.AreSequenceEqual(expected, actual);
```

Przykładowy kod, do sprawdzania wyniku w postaci kolekcji

Testy parametryzowane

Test parametryzowany pozwala na sprawdzenie tej samej operacji dla zestawu różnych danych wejściowych:

- atrybut `[DataRow(...)]` przekazuje argumenty do metody testowej,
- każdy zestaw danych jest uruchamiany i oceniany osobno,
- liczba, kolejność i typy argumentów muszą odpowiadać parametrom metody.

```
[TestMethod]
[DataRow(0, "0.0.0.0")]
[DataRow(16, "255.255.0.0")]
[DataRow(24, "255.255.255.0")]
[DataRow(26, "255.255.255.192")]
[DataRow(32, "255.255.255.255")]
public void GetSubnetMask_ValidPrefix_ReturnsCorrectMask(int prefixLength, string expected)
{
    // Arrange
    var calculator = new NetworkCalculator();

    // Act
    string actual = calculator.GetSubnetMask(prefixLength);

    // Assert
    Assert.AreEqual(expected, actual);
}
```



Przygotowanie testów

Do przygotowania testów jednostkowych w obrębie danej klasy testowej służy atrybut `[TestInitialize]` – oznacza się nim metodę, wykonywaną przed każdym testem:

- pozwala wydzielić **wspólne przygotowanie obiektów**, zamiast powtarzać je w metodach testowych,
- dla testów parametryzowanych inicjalizacja odbywa się **przed każdym zestawem danych**,
- metoda inicjalizująca jest **publiczna, niestatyczna** i **nie przyjmuje parametrów**.

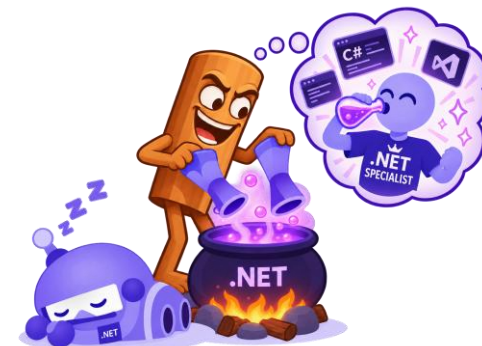
```
[TestClass]
public class NetworkCalculatorTests
{
    private NetworkCalculator _calculator = null!;

    [TestInitialize]
    public void Initialize()
    {
        _calculator = new NetworkCalculator();
    }

    [TestMethod]
    [DataRow(24, "255.255.255.0")]
    [DataRow(26, "255.255.255.192")]
    public void GetSubnetMask_ValidPrefix_ReturnsCorrectMask(int prefixLength, string expected)
    {
        string actual = _calculator.GetSubnetMask(prefixLength);
        Assert.AreEqual(expected, actual);
    }
}
```

Uwaga!

W tym przypadku każdy przypadek testowy rozpoczyna się od utworzenia nowego obiektu kalkulatora.



Porządkowanie po testach

Atrybut `[TestCleanup]` opisuje metodę wykonywaną po każdym teście, również gdy asercja zakończy się niepowodzeniem:

- służy do **zwalniania zasobów** i **usuwania danych tymczasowych** utworzonych przez test,
- w testach parametryzowanych sprzątanie odbywa się **po każdym zestawie danych osobno**,
- metoda jest **publiczna**, **niestatyczna** i **nie przyjmuje parametrów**.



```
private string _temporaryFilePath = null!;  
  
[TestInitialize]  
public void Initialize()  
{  
    _temporaryFilePath = Path.Combine(Path.GetTempPath(), $"{Guid.NewGuid()}.txt");  
}  
  
[TestCleanup]  
public void Cleanup()  
{  
    if (File.Exists(_temporaryFilePath))  
    {  
        File.Delete(_temporaryFilePath);  
    }  
}
```

Pomijanie przypadku testowego



Atrybut `[Ignore(„powód”)]` powoduje pominięcie oznaczonego testu podczas uruchomienia:

- test pozostaje w kodzie i jest widoczny w Test Explorerze jako pominięty,
- warto podać przyczynę, aby było wiadomo, dlaczego test został wyłączony,
- aby ponownie włączyć test, usuwamy atrybut `[Ignore]`.

```
[TestMethod]
[Ignore("Test tymczasowo wyłączony – trwa zmiana obsługi prefiksu /0.")]
public void GetSubnetMask_Prefix0_ReturnsZeroMask()
{
    var calculator = new NetworkCalculator();

    string actual = calculator.GetSubnetMask(0);

    Assert.AreEqual("0.0.0.0", actual);
}
```

Ignorowanie testów jest dobrą praktyką, kiedy jedna z testowanych metod jest już przestarzała – nie ma do niej referencji w kodzie.

Komunikaty diagnostyczne w testach

Metoda `TestContext.WriteLine(...)` pozwala wypisać dane wejściowe, wyniki pośrednie i informacje o kolejnych etapach testu:

- framework automatycznie przypisuje obiekt do publicznej właściwości `TestContext`,
- komunikaty pomagają znaleźć przyczynę błędu, ale nie zastępują asercji ani pełnego debugowania.

```
[TestClass]
public class NetworkCalculatorTests
{
    public TestContext TestContext { get; set; }

    [TestMethod]
    public void GetSubnetMask_Prefix24_ReturnsCorrectMask()
    {
        TestContext.WriteLine("Przygotowanie kalkulatora.");

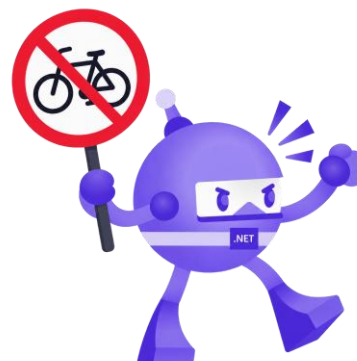
        var calculator = new NetworkCalculator();
        int prefixLength = 24;
        string expected = "255.255.255.0";

        TestContext.WriteLine($"Obliczanie maski dla /{prefixLength}.");

        string actual = calculator.GetSubnetMask(prefixLength);

        TestContext.WriteLine($"Oczekiwano: {expected}");
        TestContext.WriteLine($"Otrzymano: {actual}");

        Assert.AreEqual(expected, actual);
    }
}
```



```
▶ 📄 🔄 Copilot ▼
Podsumowanie szczegółów testu
✅ GetSubnetMask_Prefix24_ReturnsCorrectMask
📄 Źródło: SubnetMaskTests.cs wiersz 21
🕒 Czas trwania: 3 ms

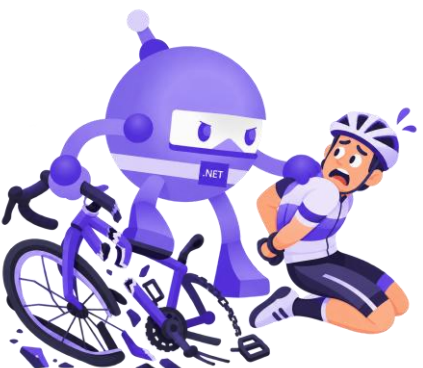
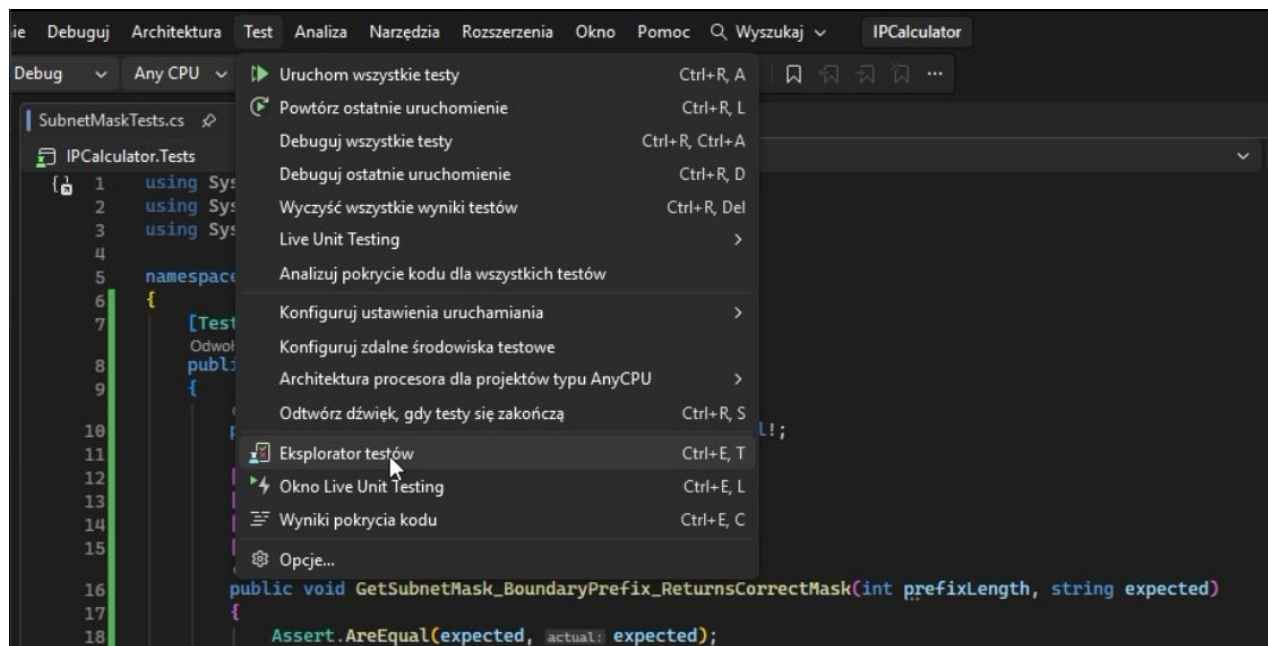
Wyjście standardowe:

Komunikaty TestContext:
Przygotowanie kalkulatora.
Obliczanie maski dla /24.
Oczekiwano: 255.255.255.0
Otrzymano: 255.255.255.0
```

Wypisane dane w podsumowaniu testu w VS

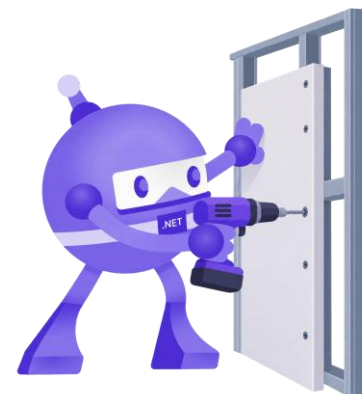
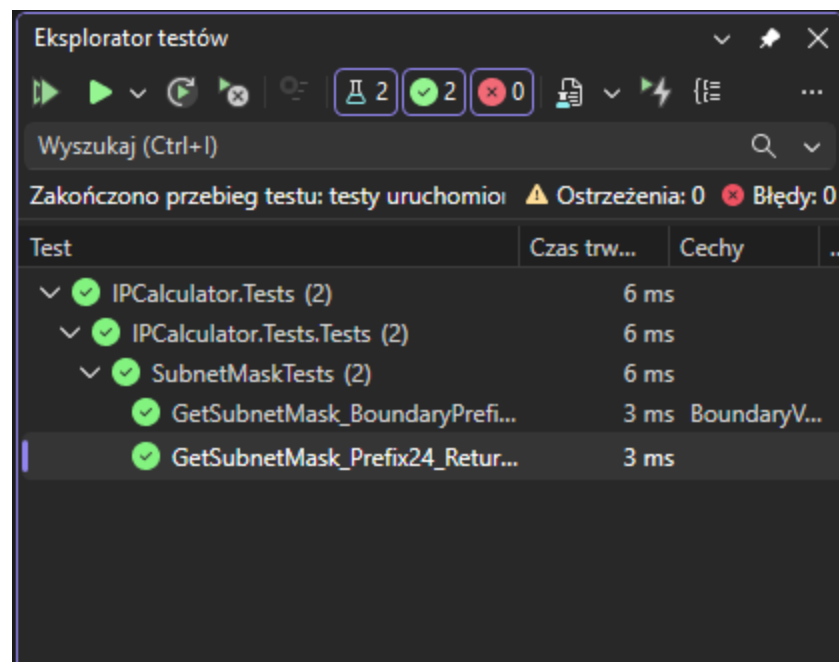
Uruchamianie testów – Visual Studio

Testy jednostkowe w [Visual Studio](#) uruchamiamy przez [Eksplorator testów](#) – aby go włączyć, należy na pasku Visual Studio kliknąć Test -> Eksplorator testów.



Uruchamianie testów – Visual Studio

Eksplorator testów daje nam możliwość ich uruchomienia (sekwencyjnie, równoległe) wszystkich testów lub pojedynczych przypadków. Wyświetla również krótkie podsumowanie – ile testów przeszło, ile z nich upadło – oraz pozwala na wyświetlenie szczegółowych informacji opisujących dany przebieg testu.



Dobór przypadków testowych



Aby testy jednostkowe pokrywały jak największą ilość możliwych przypadków, rozróżnia się je na trzy rodzaje:

- **przypadki typowe** – sprawdzają działanie dla często używanych, poprawnych danych,
- **przypadki graniczne** – obejmują krańce dozwolonego zakresu i miejsca, w których zmienia się sposób działania algorytmu,
- **dane niepoprawne** – pozwalają sprawdzić, czy metoda prawidłowo odrzuca niedozwolone wartości.

Rodzaj przypadku	Prefiks	Oczekiwany rezultat
Typowy	/24	255.255.255.0
Maska z częściowo wypełnionym oktetem	/26	255.255.255.192
Dolna granica zakresu	/0	0.0.0.0
Górna granica zakresu	/32	255.255.255.255
Przed granicą oktetu	/7	254.0.0.0
Na granicy oktetu	/8	255.0.0.0
Za granicą oktetu	/9	255.128.0.0
Poniżej dozwolonego zakresu	-1	ArgumentOutOfRangeException
Powyżej dozwolonego zakresu	33	ArgumentOutOfRangeException

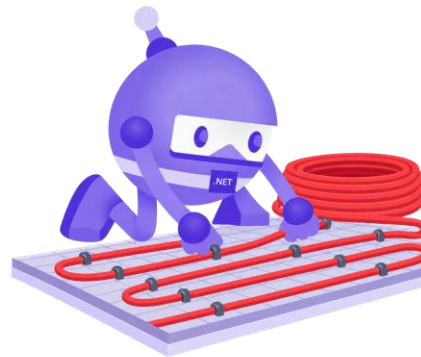
Tabela przedstawiająca różne przypadki testów na przykładzie obliczania maski sieciowej

Dobieraj dane tak, aby ujawniały różne możliwe błędy, zamiast wielokrotnie sprawdzać podobne przypadki.

Cechy dobrego testu jednostkowego

Dobry test jednostkowy powinien być:

- **szybki** - pozwala często uruchamiać cały zestaw testów,
- **niezależny** - nie wymaga wcześniejszego wykonania innych testów ani określonej kolejności,
- **powtarzalny** - dla tego samego kodu i danych daje ten sam wynik, niezależnie od czasu czy dostępności internetu,
- **czytelny** - nazwa i treść jasno wskazują sprawdzany przypadek oraz oczekiwane zachowanie,
- **skupiony na jednym zachowaniu** - jego niepowodzenie pomaga szybko ustalić, co działa niepoprawnie.



Jeden test powinien sprawdzać jedno zachowanie – nie oznacza to jednak obowiązku użycia dokładnie jednej asercji.

Frameworki do testów jednostkowych



Framework testowy dostarcza narzędzia do definiowania testów, sprawdzania wyników i współpracy z narzędziem uruchamiającym testy. W .NET największą popularnością cieszą się trzy frameworki.

Cecha	MSTest	xUnit	NUnit
Podstawowy test	[TestMethod]	[Fact]	[Test]
Test z wieloma zestawami danych	[TestMethod] + [DataRow(...)]	[Theory] + [InlineData(...)]	[TestCase(...)]
Porównanie wartości	Assert.AreEqual(expected, actual)	Assert.Equal(expected, actual)	Assert.That(actual, Is.EqualTo(expected))
Przygotowanie przed każdym testem	[TestInitialize]	<i>Konstruktor klasy testowej</i>	[SetUp]
Sprzątanie po każdym teście	[TestCleanup]	IDisposable.Dispose()	[TearDown]

Tabela przedstawiająca składnię w zależności od wybranego frameworka do testów

Składnia zależy od frameworku – różnią się głównie semantyką. Zasady dobierania przypadków, budowy testów i sprawdzania wyników pozostają takie same. Na zajęciach będziemy korzystali z MSTest.