TESTY NUNIT

Programowanie wizualne

Wojciech Frohmberg, Instytut Informatyki, Politechnika Poznańska

2022

PO CO TESTY JEDNOSTKOWE?

- Testy jednostkowe stanowią minimalistyczny zestaw danych obrazujący sposób uruchomienia funkcjonalności
- Pozwalają na zweryfikowanie poprawności działania funkcjonalności po refaktorze lub wstrzyknięciu nowej funkcjonalności
- W przypadku podejścia typu test-driven development – testy pisane są przed właściwym kodem i stanowią o jego interfejsie zapewniając przy tym możliwość testowania jeszcze w trakcie kodowania

PLAN PREZENTACJI

1. Istotne atrybuty NUnit
2. Konwencje testów
3. Rodzaje asercji

```
public class Tests
{
    [Test(ExpectedResult = 10)]
    public int SumUpThreeValuesTest()
    {
        int[] valuesToSumUp = { 1, 4, 5 };
        int obtainedResult =
            StaticClass
                .SumUpFunctionality(valuesToSumUp);
        return obtainedResult;
    }
    [Test(ExpectedResult = 0)]
    public int SumUpNoValuesTest()
    {
        int[] valuesToSumUp = { };
        int obtainedResult =
            StaticClass
                .SumUpFunctionality(valuesToSumUp);
        return obtainedResult;
    }
}
```

ISTOTNE ATRYBUTY

TEST(1)

Jest najistotniejszym atrybutem testów jednostkowych - określa, które spośród (publicznych!) metod klasy będą metodami testującymi. Wystarczy jedna metoda określona w klasie tym atrybutem żeby cała klasa była rozpoznawana przez NUnit jako testowa. W celu określenia oczekiwanego wyniku testu możemy w jego ramach skorzystać z nazwanego parametru ExpectedResult.

ISTOTNE ATRYBUTY

TEST(2)

```
public class Tests
{
    [Test(ExpectedResult = 10,
          Author = "Wojciech Frohmberg",
          Description = "Testing if the real-case " +
                        "array with three elements " +
                        "will get aggregated to the " +
                        "expected value",
          TestOf = typeof(StaticClass))]
    public int SumUpThreeValuesTest()
    {
        int[] valuesToSumUp = { 1, 4, 5 };
        int obtainedResult =
            StaticClass
                .SumUpFunctionality(valuesToSumUp);
        return obtainedResult;
    }
}
```

Dodatkowo przy użyciu nazwanego parametru TestOf możemy określić jaka klasa podlega testowi, przy użyciu nazwanego parametru Description możemy określić zakres i cel testu, a przy użyciu nazwanego parametru Author możemy określić autora testu.

```
[TestFixture(typeof(SumUpImplementation),
    new int[] { 1, 3, 6 },
    10)]
[TestFixture(typeof(OtherSumUpImplementation),
    new int[] { 1, 6, 6 },
    13)]
public class Tests<SumUpAlgorithm>
    where SumUpAlgorithm :
        ISumUpAlgorithm,
        new()
{
    SumUpAlgorithm _algorithm;
    int[] _valuesToSumUp;
    int _expectedResult;


    public Tests(int[] valuesToSumUp,
        int expectedResult)
    {
        _valuesToSumUp = valuesToSumUp;
        _expectedResult = expectedResult;
        _algorithm = new SumUpAlgorithm();
    }

    [Test]
    public void SumUpTest()
    {
        Assert.That(_algorithm.Run(_valuesToSumUp),
            Is.EqualTo(_expectedResult));
    }
}
```

ISTOTNE ATRYBUTY

TESTFIXTURE (1)

Jest nieobligatoryjnym atrybutem określającym klasy, w ramach których tworzone są testy jednostkowe. Pozwala przy tym na przekazanie zestawów parametrów tworzenia klasy. Można tutaj podać parametry generyczne lub/i parametry, które powinny być przekazane do konstruktora. Atrybut może nadany wielokrotnie co będzie skutkowało utworzeniem testów dla wszystkich zestawów przekazanych parametrów.

```
[TestFixture(typeof(Oracle),
             new int[] { 2, 3, 4 },
             TestName = "Test with suming up oracle")]
public class Tests<OracleClass>
    where OracleClass: IOracle, new()
{
    IOracle _oracle;
    int[] _valuesToSumUp;

    public Tests(int[] valuesToSumUp)
    {
        _oracle = new OracleClass();
        _valuesToSumUp = valuesToSumUp;
    }

    [Test]
    public void SumUpThreeValuesTest()
    {
        int obtainedResult =
            StaticClass
                .SumUpFunctionality(_valuesToSumUp);
        int expectedResult =
            _oracle.Control(_valuesToSumUp);
        Assert.That(obtainedResult,
            Is.EqualTo(expectedResult));
    }
}
```

ISTOTNE ATRYBUTY

TESTFIXTURE (2)

W ramach atrybutu podobnie jak w przypadku atrybutu Test można skorzystać z nazwanych parametrów TestOf, Author oraz Description. Dodatkowo każdy spośród testów można określić indywidualną nazwą przy użyciu nazwanego parametru TestName. W ramach atrybutu dostępne są jeszcze nazwane parametry Ignore, IgnoreReason oraz Reason służące do przekazania informacji nt. przyczyny nieuwzględnienia testu lub ignorowania jego wyniku.

```
[TestFixture(typeof(Oracle),
             TestName = "Test with suming up oracle")]
public class Tests<OracleClass>
    where OracleClass: IOracle, new()
{
    IOracle _oracle;

    public Tests()
    {
        _oracle = new OracleClass();
    }

    [TestCase(2, 3, 4,
        TestName = "Testing sum of 3 elements")]
    [TestCase(1, 10,
        TestName = "Testing sum of 2 elements")]
    [TestCase(
        TestName = "Testing sum of 0 elements")]
    public void SumUpThreeValuesTest(params int[] parameters)
    {
        int obtainedResult =
            StaticClass
                .SumUpFunctionality(parameters);
        int expectedResult =
            _oracle.Control(parameters);
        Assert.That(obtainedResult,
            Is.EqualTo(expectedResult));
    }
}
```

ISTOTNE ATRYBUTY

TESTCASE (1)

Podobnie jak atrybut Test atrybutem TestCase określamy metody, które chcemy by były testami, dodatkowo jednak atrybut pełni funkcję ustawiania wartości zestawu parametrów testu. Dlatego też może wystąpić na jednej metodzie wielokrotnie. Każdy taki zestaw może otrzymać własną nazwę testu przy użyciu nazwanego parametru TestName.

```
[TestFixture(typeof(Oracle),
             TestName = "Test with suming up oracle")]
public class Tests<OracleClass>
    where OracleClass: IOracle, new()
{
    IOracle _oracle;

    public Tests()
    {
        _oracle = new OracleClass();
    }

    [TestCase(2, 3, 4,
        TestName = "Testing sum of 3 elements")]
    [TestCase(1, 10,
        TestName = "Testing sum of 2 elements")]
    [TestCase(
        TestName = "Testing sum of 0 elements")]
    public void SumUpThreeValuesTest(params int[] parameters)
    {
        int obtainedResult =
            StaticClass
                .SumUpFunctionality(parameters);
        int expectedResult =
            _oracle.Control(parameters);
        Assert.That(obtainedResult,
            Is.EqualTo(expectedResult));
    }
}
```

ISTOTNE ATRYBUTY

TESTCASE (2)

Oczywiście dla atrybutu TestCase dostępne są nazwane parametry jak dla atrybutów TestFixture oraz Test takie jak: Author, Description, Ignore, IgnoreReason, Reason, TestOf, czy ExpectedResult.

```
[TestFixture(typeof(Oracle),
             TestName = "Test with suming up oracle")]
public class Tests<OracleClass>
    where OracleClass: IOracle, new()
{
    IOracle _oracle;

    public static IEnumerable<int[]> GetCollection()
    {
        yield return new int[] { 1, 2, 3 };
        yield return new int[] { 3, 1, 8 };
    }

    public Tests()
    {
        _oracle = new OracleClass();

    }

    [TestCaseSource(nameof(GetCollection))]
    public void SumUpThreeValuesTest(params int[] parameters)
    {
        int obtainedResult =
            StaticClass
                .SumUpFunctionality(parameters);
        int expectedResult =
            _oracle.Control(parameters);
        Assert.That(obtainedResult,
            Is.EqualTo(expectedResult));
    }
}
```

ISTOTNE ATRYBUTY

TESTCASESOURCE

Gdy przekazywanie wartości parametrów metody testowej w formie stałych to zbyt mało możemy posłożyć się bardziej elastycznym rozwiązaniem. Atrybut TestCaseSource pozwala na określenie źródła danych do z którego czerpać parametry do testów. Przy czym źródłem może być przykładowo metoda, w ramach której w locie będziemy definiowali kolejne wartości, ale również zewnętrzna klasa implementująca interfejs IEnumerable.

```
[TestFixture(typeof(Oracle),
             TestName = "Test with suming up oracle")]
public class Tests<OracleClass>
    where OracleClass: IOracle, new()
{
    IOracle _oracle;

    public Tests()
    {
        _oracle = new OracleClass();
    }

    [Test]
    public void SumUpValuesTest([Values(3, 2)]int param1,
                                [Values(6, 2)]int param2,
                                [Values(0, 9)]int param3)
    {
        var array = new int[] { param1, param2, param3 };
        int obtainedResult =
            StaticClass
                .SumUpFunctionality(array);
        int expectedResult =
            _oracle.Control(array);
        Assert.That(obtainedResult,
            Is.EqualTo(expectedResult));
    }
}
```

ISTOTNE ATRYBUTY

VALUES

Jeśli zależy nam na przetestowaniu naszej metody dla wszystkich możliwych kombinacji wartości każdego z parametrów możemy skorzystać z atrybutu Values. W przeciwieństwie do poprzednich atrybutów atrybut Values powinien określać nie klasę czy metodę, ale parametr. Warto zwrócić uwagę na szybko rosnącą liczbę przeprowadzanych testów wraz ze wzrostem liczby parametrów przekazanych do atrybutu (przykład wygeneruje aż 8 przypadków testowych – iloczyn kartezjański zbiorów wartości dla każdego z parametrów).

```
[TestFixture(typeof(Oracle),
             TestName = "Test with suming up oracle")]
public class Tests<OracleClass>
    where OracleClass: IOracle, new()
{
    IOracle _oracle;

    public Tests()
    {
        _oracle = new OracleClass();
    }

    [Test]
    public void SumUpValuesTest([Values(3, 2)]int param1,
                                [Range(1, 7, 2)]int param2,
                                [Values(0, 9)]int param3)
    {
        var array = new int[] { param1, param2, param3 };
        int obtainedResult =
            StaticClass
                .SumUpFunctionality(array);
        int expectedResult =
            _oracle.Control(array);
        Assert.That(obtainedResult,
            Is.EqualTo(expectedResult));
    }
}
```

ISTOTNE ATRYBUTY

RANGE

Identycznie zachowującym się pod względem tworzenia iloczynu kartezjańskiego zbioru wartości parametrów jest atrybut Range. Atrybut ten pozwala jednak na nie wymienianie wszystkich wartości explicite a określenie bardziej jak je wygenerować - przyjmuje zatem wartość początkową końcową oraz (nie obowiązkowo) krok.

```
[TestFixture(typeof(Oracle),
             TestName = "Test with suming up oracle")]
public class Tests<OracleClass>
    where OracleClass: IOracle, new()
{
    IOracle _oracle;

    public Tests()
    {
        _oracle = new OracleClass();
    }

    [Test, Pairwise]
    public void SumUpValuesTest([Values(3, 2)]int param1,
                                [Values(6, 2)]int param2,
                                [Values(0, 9)]int param3)
    {
        var array = new int[] { param1, param2, param3 };
        int obtainedResult =
            StaticClass
                .SumUpFunctionality(array);
        int expectedResult =
            _oracle.Control(array);
        Assert.That(obtainedResult,
            Is.EqualTo(expectedResult));
    }
}
```

ISTOTNE ATRYBUTY

PAIRWISE

Domyślnie, gdy korzystamy z atrybutów Values oraz Range przypadki testowe są generowane za pomocą iloczynu kartezjańskiego zbiorów możliwych wartości na każdym z parametrów. Jeśli może spowodować to zbyt dużo przypadków testowych możemy skorzystać z atrybutu Pairwise, który nadpisze to domyślne zachowanie (domyślny atrybut Combinatorial). Pairwise zapewni, że każda para wartości podanych dla parametrów będzie musiała być użyta przynajmniej raz (nie ma gwarancji optymalności doboru par, a jedynie losowe użycie poszczególnych wartości w ramach przypadków testowych). Zarówno Cobinatorial jak i Pairwise są atrybutami nadawanymi na metodę.

```
[TestFixture(typeof(Oracle),
             TestName = "Test with suming up oracle")]
public class Tests<OracleClass>
    where OracleClass: IOracle, new()
{
    IOracle _oracle;

    [SetUp]
    public void Setup()
    {
        // cleaning oracle class before
        // each test
        _oracle = new OracleClass();
    }

    [Test, Pairwise]
    public void SumUpValuesTest([Values(3, 2)]int param1,
                                [Values(6, 2)]int param2,
                                [Values(0, 9)]int param3)
    {
        var array = new int[] { param1, param2, param3 };
        int obtainedResult =
            StaticClass
                .SumUpFunctionality(array);
        int expectedResult =
            _oracle.Control(array);
        Assert.That(obtainedResult,
            Is.EqualTo(expectedResult));
    }
}
```

ISTOTNE ATRYBUTY

SETUP

Czasem w ramach przeprowadzanych testów potrzebujemy przed uruchomieniem każdego testu zaaplikować ujednolitą inicjalizację, do tego celu wprowadzony został atrybut SetUp.

KONWENCJE TESTÓW JEDNOSTKOWYCH

Jak w przypadku większości elementów implementacji również w przypadku testów jednostkowych istnieje wiele sposobów ich zapisu nie wpływających na funkcjonalność testów. W ramach danego projektu (lub w firmie) warto jednak przyjąć i trzymać się jednego formatu tj. konwencji celem większej przejrzystości kodu. Konwencja może uwzględniać nazwy testów, użycie określonych parametrów nazwanych w ich ramach czy przykładowo rodzaju asercji z których korzystamy. Np. w niektórych firmach preferuje się użycie predykatowej wersji asercji:

```
Assert.That(value, Is.EqualTo(expected))
```

zamiast funkcyjnej:

```
Assert.AreEqual(expected, value)
```

```
[TestFixture(typeof(Oracle),
             TestName = "Test with suming up oracle")]
public class Tests<OracleClass>
    where OracleClass: IOracle, new()
{
    IOracle _oracle;

    [SetUp]
    public void Setup()
    {
        // cleaning oracle class before
        // each test
        _oracle = new OracleClass();
    }

    [Test, Pairwise]
    public void SumUpValuesTest([Values(3, 2)]int param1,
                                [Values(6, 2)]int param2,
                                [Values(0, 9)]int param3)
    {
        var array = new int[] { param1, param2, param3 };
        int obtainedResult =
            StaticClass
                .SumUpFunctionality(array);
        int expectedResult =
            _oracle.Control(array);
        Assert.That(obtainedResult,
            Is.Not.AnyOf(array));
    }
}
```

RODZAJE ASERCJI

DO PORÓWNYWANIA WARTOŚCI

Grupa asercji do porównywania wartości (konwencja predykatowa):

- Is.EqualTo
- Is.LessThan
- Is.AtMost
- Is.GreaterThan
- Is.AtLeast
- Is.InRange
- Is.AnyOf

```
[TestFixture(typeof(Oracle),
             TestName = "Test with suming up oracle")]
public class Tests<OracleClass>
    where OracleClass: IOracle, new()
{
    IOracle _oracle;

    [SetUp]
    public void Setup()
    {
        // cleaning oracle class before
        // each test
        _oracle = new OracleClass();
    }

    [Test, Pairwise]
    public void SumUpValuesTest([Values(3, 2)]int param1,
                                [Values(6, 2)]int param2,
                                [Values(0, 9)]int param3)
    {
        var array = new int[] { param1, param2, param3 };

        List<int> results = new List<int>();

        for (int i = 1; i <= 3; i++)
        {
            var poweres = array.Select(x => (int)Math.Pow(x, i));
            results.Add(StaticClass
                        .SumUpFunctionality(poweres));
        }

        Assert.That(results, Is.Ordered);
    }
}
```

RODZAJE ASERCJI

DO DZIAŁANIA NA KOLEKCJACH

Grupa asercji do działania na kolekcjach (konwencja predykatowa):

- Is.All.* (czy wszystkie elementy kolekcji spełniają predykat)
- Is.Equivalent - czy kolekcje zawierają takie same elementy
- Is.Unique - czy wszystkie elementy są unikalne
- Is.Subset, Is.Superset - czy elementy zbioru stanowią podzbiór nadzbiór określonego w parametrze zbioru
- Is.Ordered - czy kolekcja jest uporządkowana

```
[TestFixture(typeof(Oracle),
             typeof(SumUpImplementation),
             TestName = "Test with suming up oracle")]
public class Tests<OracleClass, SumUpAlgorithm>
    where OracleClass: IOracle, new()
    where SumUpAlgorithm : ISumUpAlgorithm, new()
{
    IOracle _oracle;
    ISumUpAlgorithm _sumUpAlgorithm;

    [SetUp]
    public void Setup()
    {
        // cleaning oracle class before
        // each test
        _oracle = new OracleClass();
        _sumUpAlgorithm = new SumUpAlgorithm();
    }

    [TestCase(3, 2, 1)]
    public void SumUpValuesTest(int param1,
                                int param2,
                                int param3)
    {
        var array = new int[] { param1, param2, param3 };
        int expectedResult =
            _oracle.Control(array);
        Assert.That(_sumUpAlgorithm.SumPredicate,
                    Is.AssignableTo<Func<int, int, int>>());
    }
}
```

RODZAJE ASERCJI

DO DZIAŁANIA NA REFERENCJACH ORAZ TYPACH

Grupa asercji do działania na referencjach i testowania typów (konwencja predykatowa):

- Is.AssignableFrom testuje czy zmienna podlegająca asercji może przyjąć obiekt o typie przekazanym w parametrze metody (np. generycznym)
- Is.AssignableTo testuje czy obiekt o typie podlegający asercji może być przypisany do zmiennej o typie przekazanym w parametrze (np. generycznym)
- Is.InstanceOf testuje czy obiekt podlegający asercji jest instancją określonego typu lub typu który z niego dziedziczy
- Is.TypeOf testuje czy obiekt, który podlega asercji jest dokładnie tego typu, który został określony w parametrze (np. generycznym)
- Is.SameAs testuje czy referencja obiektu podlegającego asercji jest taka sama jak obiektu przekazanego w parametrze metody

```
[TestFixture(typeof(Oracle),
             TestName = "Test with suming up oracle")]
public class Tests<OracleClass>
    where OracleClass: IOracle, new()
{
    IOracle _oracle;

    [SetUp]
    public void Setup()
    {
        // cleaning oracle class before
        // each test
        _oracle = new OracleClass();
    }

    [Test, Pairwise]
    public void SumUpValuesTest([Values(3, 2)]int param1,
                                [Values(6, 2)]int param2,
                                [Values(0, 9)]int param3)
    {
        var list = new List<int> { param1, param2, param3 };
        int expectedResult =
            _oracle.Control(list);
        Assert.That(() => {
                        StaticClass.SumUpFunctionality(list);
                    },
                    Throws.Exception
                          .TypeOf<CollectionTypeNotSupported>());
    }
}
```

RODZAJE ASERCJI

DO TESTOWANIA WYJĄTKÓW

Grupa asercji do testowania czy został wyrzucony wyjątek (konwencja predykatowa):

- Throws.Exception.TypeOf testuje czy w przekazanym delegacie podczas uruchomienia został wyrzucony wyjątek o typie przekazanym w parametrze (np. generycznym)
- Throws.InnerException.TypeOf testuje czy w przekazanym delegacie podczas uruchomienia został wyrzucony wyjątek o zagnieżdżonym wyjątku przekazanym w parametrze (np. generycznym)
- Throws.Nothing testuje czy w przekazanym delegacie podczas uruchomienia nie został wyrzucony żaden wyjątek

OPRÓCZ ASERCJI - ZAŁOŻENIA

W ramach testów jednostkowych można również skorzystać z klasy Assume. Assume weryfikuje czy spełnione są wszystkie założenia dotyczące danych wejściowych do testu, żeby ten mógł być uruchomiony. Jeśli, któreś z założeń nie jest spełnione - test nie jest uwzględniany w rozpisce niezależnie od jego wyniku.

Klasa Assume charakteryzuje się identycznym interfejsem do klasy Assert.