

# Projektowanie obiektowe oprogramowania

## Testowanie oprogramowania

### Wykład 13

## Wiktor Zychła 2019

---

### 1 Wprowadzenie

State-of-the-art współczesnego warsztatu narzędzi testujących obejmuje nie tylko metodologie tworzenia testów jednostkowych, ale również szereg narzędzi wspierających, wśród których warto wymienić:

- Ramy tzw. obiektów zastępczych (*mock objects*), które w wielu przypadkach zwalniają z konieczności dostarczania konkretnych zastępczych implementacji
- Narzędzia do automatycznego generowania przypadków testowych na podstawie struktury kodu
- Narzędzia do automatycznej dynamicznej i statycznej walidacji poprawności programów

Podczas wykładu dokonamy przeglądu pojęć i narzędzi.

## 2 TDD vs BDD

**Test-Driven Development (TDD)** – rozwijanie oprogramowania sterowane testami

**System Under Test (SUT)** – klasa użytkowa, która jest podmiotem testu jednostkowego.

**Collaborators** – klasy usług pomocniczych, z których korzysta klasa SUT, ale które nie są podmiotami testów. Myślmy o architekturze, w której usługi pomocnicze są wstrzykiwane do klas użytkowych.

**AAA – Arrange/Act/Assert** – metodyka pisania testów w sposób przejrzysty, który wyróżnia jawnie (strukturą testu, komentarzem, regionami (C#)) fazy:

1. **Arrange** – organizacja SUT i collaborators (tworzenie, inicjowanie)
2. **Act** – wykonanie właściwego scenariusza biznesowego
3. **Assert** – szereg sprawdzeń

Dopuszczalne jest wielokrotne powtarzanie sekwencji Act/Assert.

Czy do testu jednostkowego dobrze jest używać rzeczywistych implementacji usług pomocniczych?

**NIE!**

Jeśli na przykład usługa dodatkowa wysyła maile czy drukuje dokumenty, to skutki uboczne testów jednostkowych mogą być niepożądane.

Rozwiązanie? Obiekt zastępczy, dubler.

**Test Double (Dubler)** – klasa implementująca usługę, zastępująca prawdziwą implementację podczas testowania

Interfejs „udawanej” usługi można zaimplementować na różne sposoby:

- **Dummy** - implementacja, która w ogóle nie jest wykorzystywana, a jej jedynym celem jest wypełnienie listy usług wstrzykiwanych do klasy SUT. Poszczególne metody implementacji typu Dummy **mogą nawet wyrzucać wyjątki** typu **NotImplementedException**, bo klasa SUT w ogóle tej usługi nie będzie wykorzystywać w danym teście.
- **Stub** – implementacja, która niekoniecznie działa zgodnie ze specyfikacją funkcjonalną; poszczególne metody zwracają wyniki **spreparowane pod kątem konkretnego testu/testów**. Przydatne do testów na konkretnych danych i konkretnej sekwencji wywołań metod.
- **Fake** – implementacja, która faktycznie działa i nawet robi to co powinna; co prawda sposób jej implementacji wyklucza jej produkcyjne wykorzystanie, ale równocześnie **pozbawiona skutków ubocznych** rzeczywistej implementacji. Przykład to implementacja repozytorium, która utrzuła obiekty w pamięci operacyjnej zamiast w bazie danych. Przydatne do testów dowolnych danych i dowolnej sekwencji wywołań metod, na których miałyby pracować rzeczywista implementacja.

- **Mock** – *Mocking Object*, typ zastępczy, gotowa rama aplikacyjna dostarczająca implementacji usług pod kątem testowania BDD

**Behavior Driven Development (BDD)** – TDD, w którym testuje się *zachowanie* implementacji SUT i usług pomocniczych a nie ich stanu.

Testowanie zachowania polega na sprawdzaniu:

- Czy SUT wywołuje właściwe metody ze swoich collaborators
- Czy wywołuje je z właściwymi **parametrami**
- Czy wywołuje je właściwą **liczbę razy**
- Czy wywołuje je we właściwej **kolejności**

Testowanie zachowania jest ogólniejsze od testowania stanu, w wielu wypadkach pozwala unikać powtarzania sekwencji Act-Assert-Act-Assert, które przy testowaniu stanu są niezbędne do rozpoznawania sekwencji stanów.

Podczas wykładu rozważymy przykład implementacji klasy odpowiadającej za obsługę zamówień, która wykorzystuje dwie usługi – usługę obsługi magazynu i usługę obsługi powiadomień.

```
public class Order
{
    private IWarehouse _warehouse { get; set; }
    private IEmailService _emailService { get; set; }

    public bool IsFilled { get; set; }

    public Order( IWarehouse warehouse, IEmailService emailService )
    {
        this._warehouse = warehouse;
        this._emailService = emailService;
    }

    public bool ValidateOrder( string Ware, int Quantity )
    {
        return !string.IsNullOrEmpty( Ware ) && Quantity > 0;
    }

    public void Fill( string Ware, int Quantity )
    {
        if ( this.ValidateOrder( Ware, Quantity ) &&
            _warehouse.HasInventory( Ware, Quantity ) )
        {
            _warehouse.Remove( Ware, Quantity );
            _emailService.SendEmail( Ware, Quantity );
            this.IsFilled = true;
        }
        else
    }
}
```

```

        this.IsFilled = false;
    }
}

public interface IWarehouse
{
    bool HasInventory( string Ware, int Quantity );
    int GetQuantity( string Ware );
    void Remove( string Ware, int Quantity );
}

public interface IEmailService
{
    void SendEmail( string Ware, int Quantity );
}

```

Rozważymy jakie testy jednostkowe mogłyby być obsługiwane kolejnymi rodzajami dublerów (dummy, fake, stub, mock) oraz zaimplementujemy każdy z rodzajów dublerów do właściwego testu.

Poznamy też przykładową ramę obiektów zastępczych dla .NET - Moq.

Na jej przykładzie pokażemy jak wyglądają dwa tryby pracy obiektu zastępczego:

- programowanie zachowania (uczenie), dopasowywanie argumentów
- weryfikacja zaprogramowanego zachowania

Pokażemy różnice między obiektami zastępczymi luźnymi i ścisłymi (loose vs strict).

Pokażemy jak programować wywołania metod i odwołania do właściwości (properties).

Pokażemy jak do zastępowanych obiektów dodawać implementacje wybranych interfejsów.

Pokażemy jak sprawdzać kolejność oraz liczbę wywołań metod.

Pokażemy również, że dobra rama obiektów zastępczych zorientowana na paradygmat BDD (testowanie zachowania) pozwala prostym zabiegiem dowiązywania funkcji zwrotnych (callbacks) do programowanych metod zastępowanego obiektu osiągnąć identyczny efekt jak przy testowaniu stanu (klasyczne TDD).

```

using Moq;
using System;
using System.Diagnostics.Contracts;

namespace ConsoleApplication3
{
    class Program
    {
        static void Main(string[] args)
        {
            var mock = new Mock<IExample>(MockBehavior.Strict)
            {
                DefaultValue = DefaultValue.Mock
            };

            // 1. Uczenie:

```

```

// * obiekt dynamicznie implementuje interfejs
mock.As<IDisposable>()
    .Setup(m => m.Dispose())
    .Callback(() => { Console.WriteLine("IDisposable::Dispose"); });

// * pewne metody mogą być wywołane tylko w określonej sekwencji
// * dla metod zwracających wynik można określić funkcję wyliczającą w
artość

// zwracaną
var seq = new MockSequence();
mock.InSequence(seq).Setup(m => m.Open());
mock.InSequence(seq)
    .Setup(m => m.DoWork(It.IsAny<int>(), It.IsAny<string>()))
    .Returns((int n, string s) =>
        {
            return n + s.Length;
        });

mock.InSequence(seq).Setup(m => m.Close());

mock.SetupGet(m => m.Child).Returns(() => new Mock<IChild>().Object );

// 2. Użycie
var example = mock.Object;

example.Open();
example.DoWork(1, "foo");
example.Close();

((IDisposable)example).Dispose();

var prop = example.Child.Property;

// 3. Walidacja

// * sprawdzenie czy metoda była zawołana tylko raz
mock.Verify(m => m.Open(), Times.Once());
// * sprawdzenie innych ograniczeń (np. kolejności)
mock.VerifyAll();

    Console.ReadLine();
}
}
public interface IExample
{
    void Open();

    int DoWork(int n, string s);
    void Close();

    IChild Child { get; }
}

public interface IChild
{
    string Property { get; }
}
}

```



### 3 Design by Contract

**Design by Contract** – technika projektowania obiektowego, w której częścią interfejsu metod i klas są zobowiązania dotyczące **stanu** w określonych momentach obliczeń:

- **precondition** (warunek wejścia) – stan w chwili rozpoczęcia wykonywania się metody
- **postcondition** (warunek wyjścia) – stan w chwili zakończenia wykonywania się metody
- **invariant** (niezmiennik) – stan w określonym momencie wykonywania się metody

Warunki mają zwykle postać predykatów (formuł logicznych typu boole’owskiego) wyrażonych w języku logiki pierwszego rzędu.

**OCL** (Object Constraint Language) – uniwersalny formalizm zaprojektowany do wyrażania kontraktów DbC w językach obiektowych, stosunkowo mało rozpowszechniony.

Podstawowa technika weryfikacji kontraktów to weryfikacja dynamiczna. Należy umieć „przechwytać” moment wywołania metody i moment zakończenia wykonywania się metody i zweryfikować poprawność formuły logicznej (czyli czy po podstawieniu wartości za zmienne formuła ewaluuje się do **true**).

Z kolei samo przechwytywanie może mieć postać:

- **statyczną** – na etapie kompilacji do metod wstrzykiwane są dodatkowe wywołania funkcji z API technologii DbC, które służą weryfikacji kontraktów
- **dynamiczną** – przechwytywanie wywołania odbywa się w trakcie działania programu.

Alternatywą dla weryfikacji dynamicznej jest weryfikacja statyczna – co w ogólności oczywiście nie jest możliwe ponieważ problem statycznej weryfikacji kontraktów jest nierozstrzygalny. Oznacza to, że wydajny algorytm może mylić się na „niekorzyść”, tzn. uznać za niepoprawny taki kod, który w rzeczywistości jest poprawny.

Metody formalne są już na tyle dobrze rozwinięte, że odpowiednie narzędzia są częścią technologii przemysłowych.

Podczas wykładu obejrzymy przykładową implementację DbC - **Code Contracts**. Zobaczymy jak rama wspiera testy dynamicznie i ... uwaga ... statyczne (czyli wykrywanie błędnych kontraktów w trakcie kompilacji!).

```
using System;
using System.Diagnostics.Contracts;

namespace ConsoleApplication3
{
    class Program
    {
        static void Main(string[] args)
        {
            var test = new Test();

            var result = test.Abs(1);
            result = test.Abs(-1);
        }
    }
}
```

```

        Console.WriteLine(result);

        int a = 1, b = 2;
        test.Swap(ref a, ref b);

        Console.WriteLine( $"{a}, {b}" );
    }
}

public class Test
{
    /// <summary>
    /// Funkcja deklaruje Postcondition: result >=0
    /// </summary>
    public int Abs( int x )
    {
        Contract.Ensures(Contract.Result<int>() >= 0);

        if ( x > 0 )
        {
            return x;
        }
        else
        {
            return -x;
        }
    }

    /// <summary>
    /// Zamiana wartości argumentów przekazanych przez referencje.
    /// Język kontraktów jest wystarczająco pojemny żeby napisać
    /// właściwy Postcondition
    /// </summary>
    public void Swap( ref int x, ref int y )
    {
        Contract.Ensures(Contract.OldValue(x) == y);
        Contract.Ensures(Contract.OldValue(y) == x);

        var _temp = x;
        x          = y;
        y          = _temp;
    }
}
}

```

Próba kompilacji niepoprawnego kodu (w którym implementacja nie jest zgodna z zadeklarowanym kontraktem) może się nie udać:



```

Program.cs
ConsoleApplication3
ConsoleApplication3.Test
Abs(int x)
13 result = test.Abs(-1);
14
15 Console.WriteLine(result);
16
17 int a = 1, b = 2;
18 test.Swap(ref a, ref b);
19
20 Console.WriteLine( $"{a}, {b}" );
21
22 }
23
24 1 reference
25 public class Test
26 {
27     /// <summary>
28     /// Funkcja deklaruje Postcondition: result >= 0
29     /// </summary>
30     2 references
31     public int Abs( int x )
32     {
33         Contract.Ensures(Contract.Result<int>() >= 0);
34
35         if ( x > 0 )
36         {
37             return x;
38         }
39         else
40         {
41             return x;
42         }
43     }
44     /// <summary>
45     /// Zmiana wartości argumentów przekazanych przez referencje.
46     /// Język kontraktów jest wystarczająco pojemny żeby napisać
47     /// właściwy Postcondition
48     /// </summary>
49     1 reference
50     public void Swap( ref int x, ref int y )
51     {
52     }
53 }

```

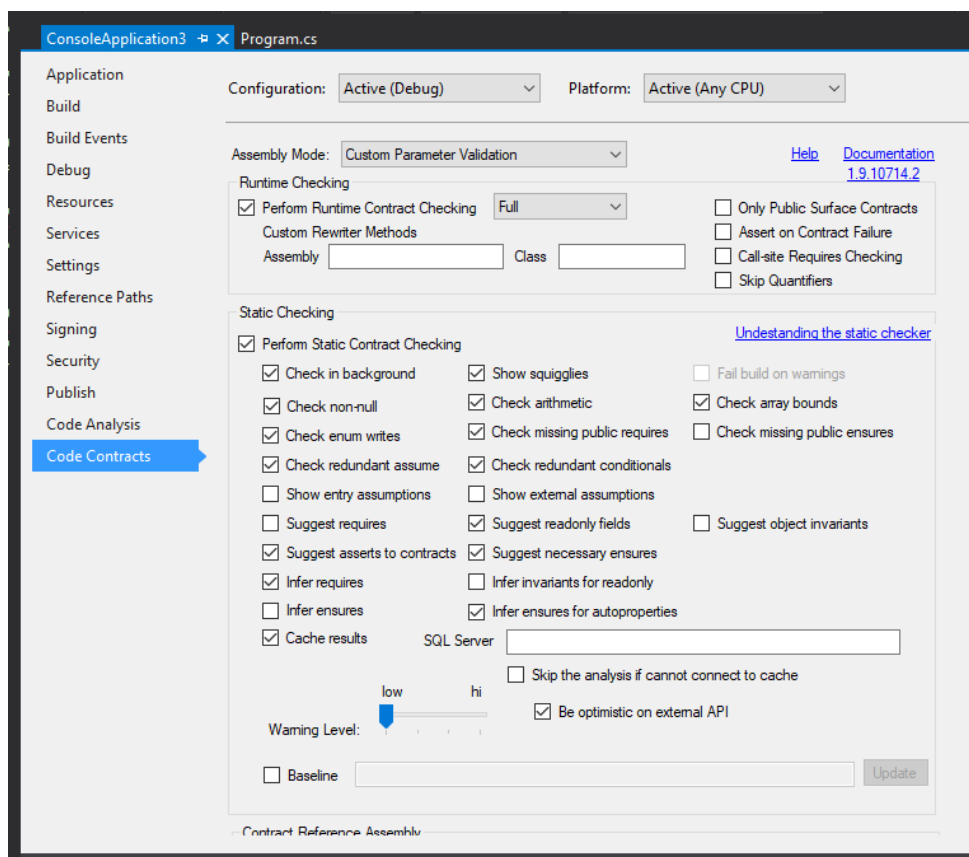
Output

Show output from: Build

```

ogram.cs(37,13): warning : CodeContracts: Missing precondition in an externally visible method. Consider adding Contract.Requires(x >= 0); for parameter validation
ogram.cs(17,13): warning : CodeContracts: requires is false: x >= 0. This sequence of invocations will bring to an error Main -> Abs, condition x >= 0
ogram.cs(37,13): warning : + location related to previous warning
3 assertions: 9 correct 1 unknown 2 unreachable 1 false

```



W tym przypadku statyczny analizator zwraca następujący komunikat diagnostyczny

Program.cs(37,13): warning : CodeContracts: Missing precondition in an externally visible method. Consider adding Contract.Requires(x >= 0); for parameter validation

Program.cs(17,13): warning : CodeContracts: requires is false: x >= 0. This sequence of invocations will bring to an error Main -> Abs, condition x >= 0

## 4 Testowanie interfejsu użytkownika – UI Automation

Za pomocą dedykowanych ram aplikacyjnych do automatyzacji interfejsu użytkownika, możliwe jest budowanie testów akceptacyjnych od strony interfejsu użytkownika aplikacji.

Podczas wykładu obejrzymy dwie przykładowe ramy testów interfejsu użytkownika

- **UI Automation** – niskopoziomowy dostęp do abstrakcyjnych formantów interfejsu użytkownika przy wsparciu systemu operacyjnego [https://msdn.microsoft.com/en-us/library/windows/desktop/ff486375\(v=vs.85\).aspx](https://msdn.microsoft.com/en-us/library/windows/desktop/ff486375(v=vs.85).aspx)
- **framework White** – opakowanie UI Automation w wygodny dostęp do silnie typowanych formantów (przykład <http://www.wiktorzychla.com/2013/07/a-basic-example-of-web-site-automation.html>).

## 5 Literatura

Martin Fowler – „Mocks Aren’t Stubs”, <http://martinfowler.com/articles/mocksArentStubs.html>

Gerard Meszaros – “xUnit Test Patterns”, <http://xunitpatterns.com/>