Projektowanie obiektowe oprogramowania
Testowanie oprogramowania
Wykład 13
Wiktor Zychla 2024

---

1 Wprowadzenie

Współczesny warsztat 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 wybranych narzędzi.

2 TDD vs BDD

Przypomnienie pojęć:

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. Myślimy 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 należy 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. Przykład to implementacja repozytorium, która zwraca dokładnie 2 obiekty kategorii Person, spreparowane pod pewien konkretny test, nie obsługująca w ogóle dodawania, modyfikacji ani usuwania.

- **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 utrwala obiekty w pamięci operacyjnej zamiast w bazie danych. Przydatne do testów dowolnych danych i dowolnej sekwencji wywołań metod, na których miałaby 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.

Przykład: obiekt silnika dostępu do danych implementuje metody **OpenConnection**, **ExecuteQuery** i **CloseConnection**. Test usługi korzystającej z takiego silnika używałby implementacji typu Fake/Stub. Ale oprócz dostarczania danych, można chcieć upewnić się, że w pewnym scenariuszu, metody silnika są wywoływane:

- OpenConnection – dokładnie raz
- ExecuteQuery – co najmniej raz
- CloseConnection – dokładnie raz

w takiej właśnie kolejności.

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**[1].

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

[1] https://github.com/moq/moq4

programowanych metod zastępowanego obiektu osiągnąć identyczny efekt jak przy testowaniu stanu (klasyczne TDD).

```
using Moq;
using Moq.Sequences;
using System;

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

            // 4. bardziej skomplikowany przykład sekwencji,
            // ze wskazaniem ile razy w sekwencji można wołać metodę

            var mock2 = new Mock<IExample>(MockBehavior.Strict);

            using (var sequence = Sequence.Create())
            {
                mock2.Setup(m => m.Open()).InSequence(Times.Once());
                mock2.Setup(m => m.DoWork(It.IsAny<int>(), It.IsAny<string>()))
                    .InSequence(Times.AtLeastOnce())
                    .Returns((int n, string s) =>
                    {
                        return n + s.Length;
                    });
                mock2.Setup(m => m.Close()).InSequence(Times.Once());

                var example2 = mock2.Object;

                example2.Open();
                example2.DoWork(1, "2");
                example2.DoWork(3, "4");
                example2.Close();
            }

            mock2.VerifyAll();

            Console.WriteLine("finished");

            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 „przechwycić" 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.

```
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;
        }
    }
}
```

Technologia kontraktów dla .NET znajduje się aktualnie w nieprzyjemnym stadium przejściowym – jest wspierana poprawnie dla narzędzi Visual Studio <= 2015, a dla nowszych > 2017, z uwagi na zmianę architektury kompilatora, nadal nie doczekała się działającej wersji.

Poza tym więc, że można kontrakty w kodzie napisać, nie ma aktualnie żadnego łatwego sposobu żeby je weryfikować statycznie (sic!).

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.

Istnieją rożne podejścia do takiego testowania

- **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

- **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).

- technologie oparte o protokół WebDriver (2004 z późniejszymi zmianami)

- technologia Puppeteer (2017)

```
using PuppeteerSharp;
using System;
using System.Threading.Tasks;

namespace ConsoleApplication3
{
    class Program
    {
        static async Task Browser()
        {
            using ( var browserFetcher = new BrowserFetcher() )
            {
                await browserFetcher.DownloadAsync( BrowserFetcher.DefaultChr
omiumRevision );
                var browser = await Puppeteer.LaunchAsync(new LaunchOptions
                {
                    Headless = false
                });
                try
                {
                    var page = await browser.NewPageAsync();
                    await page.GoToAsync( "https://duckduckgo.com/" );

                    var element = await page.QuerySelectorAsync( "#search_for
m_input_homepage" );
                    await element.FocusAsync();

                    await page.Keyboard.TypeAsync( "hello world" );

                    await page.ClickAsync( "#search_button_homepage" );
                    await page.WaitForNavigationAsync();


                }
                finally
```

```
                {
                    await browser.CloseAsync();
                }
            }
        }

        static void Main( string[] args )
        {
            Browser().Wait();
        }
    }
}
```

5 Literatura

Martin Fowler – „Mocks Aren't Stubs", http://martinfowler.com/articles/mocksArentStubs.html

Gerard Meszaros – "xUnit Test Patterns", http://xunitpatterns.com/