Drukuj Zamknij

# Testowanie modułów oraz tworzenie metodą „testy-najpierw”

2003-10-15 08:00:44
Użytkownik systemowy - Administratorzy, Portal .NET

Jeśli czytelnik jest jednym z nielicznych zapewne ludzi, którzy przeczytali moją biografię na końcu,  może już wie, że zanim stałem się kierownikiem programowym, byłem głównym testerem dla kompilatora C#, a jeszcze wcześniej – dla kompilatora C++. To czyni mnie bardziej niż nieco zainteresowanym analizowaniem i, kiedy to tylko możliwe, unikaniem błędów.

Jednym ze sposobów na zredukowanie liczby błędów trafiających się w tworzonym oprogramowaniu jest zatrudnianie profesjonalnego zespołu testowego, którego zadaniem są agresywne próby spowodowania awarii tego programu. Niestety, tam gdzie jest taki zespół, pojawia się tendencja – nawet u doświadczonych programistów – by poświęcać mniejszą ilość czasu na zwiększanie niezawodności ich kodu.

Popularną w świecie oprogramowania tezą jest, że „programiści nie powinni testować własnego kodu”. Opiera się to na teorii, że programiści wiedzą tak dużo o swoim własnym kodzie, że będą mieli z góry wyrobione opinie, jak należy go właściwie przetestować. Jest w tym więcej niż tylko ziarno prawdy, ale teoria ta pomija jedną ważną kwestię: jeśli programiści nie będą testować własnego kodu, skąd będą wiedzieli, czy działa tak, jak powinien działać?

Prosta odpowiedź brzmi: nie będą wiedzieli. A programista piszący kod, który nie działa lub który działa tylko w pewnych sytuacjach, to duży problem. Zamiast sprawdzić, czy ich kod działa we wszystkich przypadkach, zwykle testują ich tylko kilka.

## Wykrywanie błędów

Istnieje wiele sytuacji, w których wykrywa się błędy:

1. przez programistę podczas pisania kodu pierwszy raz,
2. przez programistę podczas prób doprowadzenia kodu do działania,
3. przez innego programistę lub testera w zespole.
4. w ramach większego testu produktu,
5. przez użytkownika końcowego.

Jeśli błąd został wykryty w sytuacji 1), jest łatwy i tani do usunięcia. W miarę przesuwania się w dół listy staje się to coraz kosztowniejsze, a naprawienie błędu odkrytego przez użytkownika końcowego może kosztować 100 lub 1000 razy więcej. Nie wspominając już o fakcie, że użytkownik będzie się borykał z tym błędem aż do wydania kolejnej wersji programu.

Najlepiej by było, gdyby wszystkie błędy były wykrywane już podczas pisania kodu przez programistę. W tym celu powinien mieć on testy, które można uruchamiać podczas pisania kodu. Istnieje ciekawa metodologia, która właśnie temu służy.

## Tworzenie metodą „testy-najpierw”

Przy tworzeniu kodu z zastosowaniem metody „testy-najpierw” pisze się testy *przed* napisaniem kodu. Gdy wszystkie testy działają, wiadomo, że kod też działa poprawnie, a w miarę dodawania nowych funkcji te testy również sprawdzają, czy nic nie zostało zepsute.

Pojęcie to powstało we wczesnych latach 90.  XX wieku w świecie Smalltalk, gdy Kent Beck napisał SmalltalkUnit. Przez następne lata narzędzia do testowania modułów napisano dla większości środowisk; między innymi  powstało bardzo dobre narzędzie dla Infrastruktury .NET, znane jako nUnit.

## Przykład

Aby wyjaśnić sposób działania tej metody tworzenia, napiszę klasę **IntegerList**. Jest to odmiana klasy **ArrayList** przechowującej liczby całkowite w postaci rodzimej, dzięki czemu nie ma narzutu spowodowanego pakowaniem (ang. *boxing*) oraz rozpakowywaniem (ang. *unboxing*).

Pierwszym moim krokiem będzie stworzenie projektu konsoli i dodanie do niego pliku źródłowego IntegerList.cs source. Aby podłączyć infrastrukturę nUnit, muszę dodać odwołania do infrastruktury nUnit. W moim systemie znajdują się one w d:\program files\nUnit v2.0\bin.

Kolejną rzeczą będzie zastanowienie się nad tym, jak tę klasę będziemy testować. Przypomina to decydowanie,

jakie możliwości powinna mieć dana klasa, ale skupia się na konkretnych jej użyciach (dodaj wartość 1 do listy, a następnie sprawdź, czy się to udało), a nie na samych możliwościach (dodaj element do listy). Zanim więc stworzymy klasę, najpierw układamy listę testów, które wobec niej wykonamy:

1. Test sprawdzający, czy ta klasa może zostać skonstruowana.
2. Dodanie dwóch liczb całkowitych do listy i upewnienie się, że liczba elementów jest prawidłowa oraz że same elementy są prawidłowe.
3. Test jak w punkcie 2, ale z większą liczbą elementów.
4. Konwersja listy na łańcuch.
5. Wyliczenie listy za pomocą **foreach**.

Ten przykład jest nieco nietypowy w tym sensie, że już na samym początku wiemy, co ta klasa powinna robić. Większość klas jest budowanych krok po kroku i w miarę jak klasa rośnie, powinny być dodawane także testy.

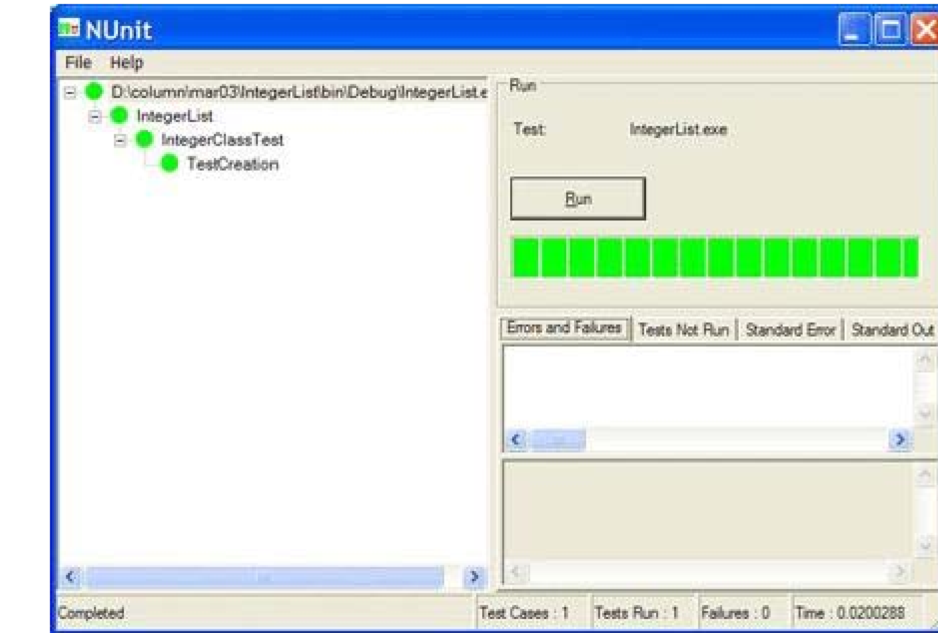
Teraz możemy już zacząć pracę. Utworzę plik nowej klasy C#, który nazwę IntegerListTest.cs i który będzie zawierał wszystkie moje testy. Oto plik z pierwszym testem:

```
using System;
using System.Collections;
using NUnit.Framework;


namespace IntegerList
{
   /// <streszczenie>
   /// Streszczony opis IntegerClassTest.
   /// </streszczenie>
   [TestFixture]
   public class IntegerClassTest
   {
       [Test]
       public void ListCreation()
       {
           IntegerList list = new IntegerList();
           Assertion.AssertNotNull(list);
       }
   }
}
```

Atrybut[TestFixture] oznacza tę klasę jako klasę testową, zaś atrybut [Test] oznacza ListCreation() jako metodę testową. W tej metodzie tworzymy listę, a następnie sprawdzamy, czy obiekt został utworzony, za pomocą klasy **Assertion**.

Uruchamiamy tester GUI nUnit, otwieramy plik wykonywalny, a następnie wydajemy polecenie uruchomienia testów. W wyniku  uzyskujemy widok z Rysunku 1.




**Rysunek 1. GUI nUnit wyświetla wyniki testów**

Na rysunku widać, że klasa przeszła wszystkie nasze testy. Teraz będę chciał dodać jakieś prawdziwe funkcje. Pierwszą rzeczą, którą chcę być w stanie zrobić, jest dodanie do listy liczby całkowitej. Oto ten test:

```
[Test]
public void TestSimpleAdd()
{
    IntegerList list = new IntegerList();
    list.Add(5);
    list.Add(10);
    Assertion.AssertEquals(2, list.Count);
    Assertion.AssertEquals(5, list[0]);
    Assertion.AssertEquals(10, list[1]);
}
```

W tym teście postanowiłem przetestować kilka rzeczy na raz:

- Czy lista utrzymuje prawidłową wartość właściwości **Count**.
- Czy lista może przechowywać dwa elementy.

Niektórzy orędownicy tworzenia kierowanego testami postulują, żeby testy były tak rozdrobnione, jak to tylko możliwe, ale dla mnie dziwne by było testowanie liczby elementów bez testowania samych elementów, więc takie właśnie rozwiązanie wybrałem.

Nie można tego kodu skompilować, ponieważ w klasie **IntegerList** nie ma metod. Dodaję więc do niej następujące puste metody, dzięki którym kompilacja się powiedzie:

```
public int Count
{
    get
    {
        return -1;
    }
}
```

```
public void Add(int value)
{
}

public int this[int index]
{
    get
    {
        return -1;
    }
}
```

Powracam do testów, które i tym razem wyświetlają się na czerwono – co sygnalizuje niepowodzenie testów. To dobrze, ponieważ to oznacza, że moje testy rzeczywiście testują coś, co w chwili obecnej jeszcze nie działa. Teraz mogę wykonać implementację – zrobię coś prostego, ale nieefektywnego:

```
public int Count
{
    get
    {
        return elements.Length;
    }
}

public void Add(int value)
{
    int newIndex;
    if (elements != null)
    {
        int[] newElements = new int[elements.Length + 1];
        for (int index = 0; index < elements.Length;
             index++)
        {
            newElements[index] = elements[index];
        }
        newIndex = elements.Length;
        elements = newElements;
    }
    else
    {
        elements = new int[1];
        newIndex = 0;
    }
    elements[newIndex] = value;
```

```
        }

        public int this[int index]
        {
            get
            {
                return elements[index];
            }
        }
```

Mam już zrobioną małą część mojej klasy oraz testy zapewniające, że działa ona poprawnie, ale przetestowałem ją tylko dla małej liczby elementów. Napiszę więc test sprawdzający ją dla 1000 elementów:

```
        [Test]
        public void TestOneThousandItems()
        {
            list = new IntegerList();

            for (int i = 0; i < 1000; i++)
            {
                list.Add(i);
            }

            Assertion.AssertEquals(1000, list.Count);
            for (int i = 0; i < 1000; i++)
            {
                Assertion.AssertEquals(i, list[i]);
            }
        }
```

Ten test został zakończony pomyślnie, więc nie muszę już dokonywać żadnych zmian.

## Dodawanie metody ToString()

Dodam teraz kod testujący, czy **ToString()** działa poprawnie:

```
        [Test]
        public void TestToString()
        {
            IntegerList list = new IntegerList();
            list.Add(5);
            list.Add(10);
            string t = list.ToString();
            Assertion.AssertEquals("5, 10", t.ToString());
        }
```

To jednak zawodzi. Poniżej znajduje się kod, który sprawia, że klasa przechodzi test:

```
        public override string ToString()
        {
            string[] items = new string[elements.Length];
```

```
        for (int index = 0; index < elements.Length; index++)
        {
            items[index] = elements[index].ToString();
        }
        return String.Join(", ", items);
    }
```

## Włączanie Foreach

Wielu użytkowników zapewne będzie chciało wykonać **foreach** dla mojej listy. W tym celu muszę dla tej klasy zaimplementować **IEnumerable** oraz zdefiniować osobną klasę implementującą **IEnumerable**. Najpierw więc test:

```
    [Test]
    public void TestForeach()
    {
        IntegerList list = new IntegerList();
        list.Add(5);
        list.Add(10);
        list.Add(15);
        list.Add(20);

        ArrayList items = new ArrayList();

        foreach (int value in list)
        {
            items.Add(value);
        }

        Assertion.AssertEquals("Liczba", 4, items.Count);
        Assertion.AssertEquals("indeks 0", 5, items[0]);
        Assertion.AssertEquals("indeks 1", 10, items[1]);
        Assertion.AssertEquals("indeks 2", 15, items[2]);
        Assertion.AssertEquals("indeks 3", 20, items[3]);
    }
```

Zaimplementowałem także **Ienumerable** w **IntegerList**:

```
    public IEnumerator GetEnumerator()
    {
        return null;
    }
```

To powoduje wystąpienie wyjątku po uruchomieniu testu. Aby zaimplementować to poprawnie, zastosuję w charakterze mechanizmu wyliczeniowego klasę zagnieżdżoną.

```
class IntegerListEnumerator: IEnumerator
{
    IntegerList    list;
    int index = -1;
```

```
        public IntegerListEnumerator(IntegerList list)
        {
            this.list = list;
        }
        public bool MoveNext()
        {
            index++;
            if (index == list.Count)
                return(false);
            else
                return(true);
        }
        public object Current
        {
            get
            {
                return(list[index]);
            }
        }
        public void Reset()
        {
            index = -1;
        }
    }
```

Klasa ta otrzymuje wskaźnik na obiekt **IntegerList**, a potem po prostu zwraca jej elementy.

To włącza **foreach** dla naszej listy, ale niestety właściwość **Current** jest typu obiektowego, co oznacza, że każda wartość będzie opakowywana w celu przekazania jej z powrotem. Ten problem można rozwiązać przyjmując podejście oparte na wzorcach, które wygląda tak samo jak obecne, z tą tylko różnicą, że **GetEnumerator()** zwraca prawdziwą klasę (a nie **IEnumerator**), oraz właściwość **Current** w tej klasie jest typu **int**.

Gdy jednak już to zrobię, chcę mieć pewność, że nadal mogę używać podejścia opartego na interfejsie w przypadku języków nieobsługujących wzorca. Skopiuję ostatni napisany test i zmodyfikuję **foreach** tak, aby było ono rzucane na interfejs:

```
            foreach (int value in (IEnumerable) list)
```

Kilka zmian i lista działa już dla obu przypadków. Więcej szczegółów oraz więcej testów można znaleźć w kodzie przykładowym.

## Komentarze

Napisanie kodu oraz tekstu do artykułu na ten miesiąc zajęło mi około godziny. Miłą stroną pisania testów jest przede wszystkim to, że ma się dobre pojęcie o tym, co trzeba dodać do klasy, żeby ten test przeszła, więc już pisanie kodu jest łatwiejsze.

To podejście działa najlepiej w przypadku małych, przyrostowych testów. Zachęcam do wypróbowania tej metody na jakimś małym projekcie. Tworzenie metodą „testy-najpierw” jest częścią czegoś określanego mianem zwinnych metodologii (ang. Agile Methodologies). Więcej informacji o „zwinnym” tworzeniu można znaleźć pod adresem http://www.agilealliance.com/home.

## Spojrzenie w przyszłość

Bawiłem się ostatnio Microsoft DirectX® 9, więc nie zdziwcie się, jeśli to będzie kolejny opisany temat.

ERIC GUNNERSON

Microsoft Corporation

1 marca 2003 roku

**Eric Gunnerson** jest kierownikiem programowym zespołu Visual C#, byłym członkiem zespołu projektowego języka C# oraz autorem Wstęp do C# dla Programistów, Drugie Wydanie (A Programmer´s Introduction to C#, 2nd Edition). Był programistą wystarczająco długo, by wiedzieć, jak wyglądają 8-calowe dyskietki i kiedyś potrafił zakładać taśmy jedną ręką. W wolnym czasie próbował ściąć drzewo za pomocą śledzia.

Drukuj Zamknij