JOURNAL of COMPUTER SCIENCES INSTITUTE

JCSI 9 (2018) 334-339
Wysłane: 2018-09-14
Przyjęte: 2018-09-19

Porównanie wybranych narzędzi do przeprowadzania testów jednostkowych

Piotr Strzelecki*, Maria Skublewska - Paszkowska
Politechnika Lubelska, Instytut Informatyki, Nadbystrzycka 36B, 20-618 Lublin, Polska

Streszczenie. Artykuł przedstawia analizę porównawczą wybranych narzędzi służących do przeprowadzania testów jednostkowych. Analizie zostały poddane trzy najpopularniejsze frameworki: MSTest, NUnit oraz xUnit.net. Analiza polega na porównaniu szybkości wykonywania testów przez narzędzia w sposób szeregowy oraz równoległy. Badania przeprowadzono z wykorzystaniem autorskiej aplikacji na platformę .NET.

Słowa kluczowe: testy jednostkowe; testowanie; .NET

*Autor do korespondencji.

Adres e-mail: piotr.strzelecki93@gmail.com

Comparison of selected tools to perform unit tests

Piotr Strzelecki*, Maria Skublewska - Paszkowska
Institute of Computer Science, Lublin University of Technology, Nadbystrzycka 36B, 20-618 Lublin, Poland

Abstract. The paper presents a comparative analysis of selected tools to perform unit tests. The analysis covers three most popular frameworks: MSTest, NUnit and xUnit.net. The analysis concerns the comparison of the speed of tests performing by the tools in serial and parallel manner. The tests were carried out by the author's application dedicated to .NET platform.

Keywords: unit tests; testing; .NET

*Corresponding author.

E-mail address: piotr.strzelecki93@gmail.com

1. Wstęp

W dzisiejszych czasach powstające projekty są bardzo złożone, wielkie i podatne na błędy. Z tego powodu testowanie wytwarzanego oprogramowania ma kluczowe znaczenie. Testy powinny być pisane profesjonalnie i skutecznie, czyli tak, aby wynik końcowy był sprawdzony pod każdym możliwym kątem, dzięki czemu użytkownicy z niego korzystający byliby zadowoleni z wydanej aplikacji. Korzystanie z narzędzi przeznaczonych do tworzenia testów jednostkowych znaczne upraszcza pisanie i zarządzanie zestawami takich testów, które sprawdzają poprawność działania wytwarzanej aplikacji.

Artykuł podejmuje temat porównania wybranych narzędzi do przeprowadzania testów jednostkowych. Analizy dokonano na bazie wykonanej aplikacji internetowej opartej o platformę .NET. Celem badania było porównanie narzędzi MSTest, NUnit oraz xUnit.net pod kątem szybkości czasów wykonywania tworzonych przez nich testów budowanych w sposób szeregowy oraz równoległy. Przeprowadzone badanie pozwoli programiście na wybranie odpowiedniego narzędzia, służącego do tworzenia testów jednostkowych.

2. Przegląd literatury

Zanim zostały przeprowadzone badania, które są tematem tego artykułu, dokonano analizy dostępnych materiałów badawczych. Nie było dotychczas analizy porównawczej narzędzi, które tutaj są porównywane. W literaturze natomiast, można znaleźć kilka przykładów dotyczących przeprowadzania testów jednostkowych. Według pracy Augustyna D. R. [1] framework NUnit stał się standardem dla rozwiązań w zakresie testów jednostkowych w technologii .NET. Badania [2, 3] pokazują, że stosowanie testów jednostkowych jest jednym z podstawowych technik pozwalających zapewnić niezawodność wytwarzanego oprogramowania, a wraz z jego rozwojem powstały inne metodyki, ułatwiające tworzenie takich testów, m.in. TDD (ang. Test-Driven Development).

3. Testy jednostkowe

Testy jednostkowe (ang. unit test), nazywane również modułowymi, to w programowaniu sposób testowania wytwarzanego oprogramowania poprzez wykonywanie testów, które weryfikują poprawność działania pojedynczych elementów programu. Najczęściej tymi elementami są klasy, funkcje, interfejsy oraz procedury [4]. Badane części kodu testowane są oddzielnie od innych komponentów, przez co są izolowane od innych zależności, które mogłyby wpłynąć na wynik testu. Izolację taką otrzymuje się przy pomocy stosowanych zaślepek, obiektów imitacji (ang. mock) [5].

Sam test polega na tym, że jego autor dostarcza dane wejściowe, test wykonuje pewne instrukcje i sprawdza, czy rezultat działań zgodny jest z oczekiwaniami. Sprawdzenie poprawności jego działania wykonuje się najczęściej przy pomocy metod pomocniczych tzn. asercji.

Testy jednostkowe przeprowadzane są najczęściej dla przygotowanych uprzednio danych testowych i mają na celu ujawnienie jak największej liczby błędów. Pomagają one wykryć nieprawidłowości już na poziomie implementacji danej aplikacji, dzięki czemu mogą być one naprawione zaraz po ich odnalezieniu. Testy modułowe powinny być z założenia szybkie w uruchomieniu oraz proste w implementacji, przez co stają się one świetnym elementem do automatyzacji, co z kolei ułatwia natychmiastowe niemal sprawdzenie, czy zmiany w kodzie nie naruszyły funkcjonalności, która działała poprawnie przed wprowadzeniem zmian. Testy jednostkowe zyskały na popularności w szczególnej technice wytwarzania oprogramowania tj. TDD (ang. Test-Driven Development), czyli wytwarzaniem sterowanym testowaniem, gdzie pisanie oprogramowania zaczyna się od tworzenia testów [6].

4. Badane technologie

Wszystkie porównywane narzędzia były typu open source. W przeprowadzonej analizie brały udział:

- MSTest – (Microsoft's Test Framework) narzędzie wbudowane do IDE Visual Studio. Stworzone za jego pomocą testy można uruchomić w środowisku Visual Studio lub używając poleceń w konsoli. Aktualna wersja stabilna to 1.3.2, która jest rozwinięciem produktu Microsoft Test Framework pod nazwą MSTest V2 [7].
- NUnit – framework, stworzony dla platformy .NET i wywodzący się z rodziny programów xUnit. Testy stworzone przy pomocy tego narzędzia można uruchomić w konsoli, w Visual Studio po uprzednim pobraniu Test Adaptera lub za pomocą programu NUnit Gui – dedykowanego, niezależnego programu z interfejsem graficznym do uruchamiania i zarządzania testami napisanymi w tym frameworku. Aktualna wersja frameworka to 3.10.1 [8].
- xUnit.net – najnowsze (powstałe w 23.04.2008 r. [9]) narzędzie do tworzenia testów jednostkowych na platformie .NET, stworzone przez autora frameworka NUnit. Testy można uruchamiać z poziomu: wiersza poleceń oraz w Visual Studio, po uprzednim pobraniu pakietu xunit.runner.visualstudio. Aktualną wersją stabilną jest wersja 2.4.0 [10].

4.1. Asercje

Podczas pisania testów jednostkowych przy pomocy narzędzi do tego przeznaczonych głównymi elementami, z których korzysta programista są asercje. Dzięki nim właśnie możliwe jest ustalenie, czy dany przypadek testowy uznany jest za pomyślny lub nieudany.

Asercje są to statyczne metody znajdujące się w klasie Assert. Klasa ta jest mostem między testowanym kodem, a frameworkami. W przypadku przekazania do klasy Assert argumentów niezgodnych z założeniem, framework rozpoznaje, że dany test się nie powiódł i informuje o tym odpowiednim komunikatem [11].

NUnit używa dwóch modeli asercji: pierwszy to model klasyczny, a drugi to model ograniczenia (ang. constraint model), w którym wykorzystywana jest składnia płynna (ang. fluent) [12]. Składnia ta zawsze zaczyna się od Assert.That, po którym następuje ograniczenie np.: `Assert.That(myString, Is.EqualTo("Hello"));`

W tabeli 1 znajduje się porównanie asercji w poszczególnych frameworkach. W przypadku NUnit podane są dwa modele. Znaczenie podanych asercji można określić poprzez ich nazwę, dlatego umożliwia to korzystanie z nich bez wcześniejszej wiedzy na temat pracy z tymi narzędziami.

Tabela 1. Asercje dostępne we frameworkach [13]

NUnit	MSTest	xUnit.net
AreEqual, Is.EqualTo	AreEqual	Equal
AreNotEqual, Is.Not.EqualTo	AreNotEqual	NotEqual
AreNotSame, Is.Not.SameAs	AreNotSame	NotSame
AreSame, Is.SameAs	AreSame	Same
Contains, Does.Contain	brak	Contains
Does.Not.Contain	brak	DoesNotContain
Throws.Nothing	brak	brak
Fail	Fail	brak
Greater, Is.GreaterThan	brak	brak
Is.InRange	brak	InRange
IsAssignableFrom, Is.AssignableFrom	brak	IsAssignableFrom
IsEmpty, Is.Empty	brak	Empty
False, Is.False	IsFalse	False
IsInstanceOf, Is.InstanceOf<T>	IsInstanceOfType	IsType<T>
IsNaN, Is.NaN	brak	brak
IsNotAssignableFrom, Is.Not.AssignableFrom<T>	brak	brak
IsNotEmpty, Is.Not.Empty	brak	NotEmpty
IsNotInstanceOf, Is.Not.InstanceOf<T>	IsNotInstanceOfType	IsNotType<T>
IsNotNull, Is.Not.Null	IsNotNull	NotNull
IsNull, Is.Null	IsNull	Null
IsTrue, Is.True	IsTrue	True
Less, Is.LessThan	brak	brak
Is.Not.InRange	brak	NotInRange
Throws<T>, Throws.TypeOf<T>	ThrowsException<T>	Throws<T>

5. Przebieg badań

5.1. Środowisko testowe

Przeprowadzone badania zostały wykonywane na maszynie, której środowisko zostało przedstawione w tabeli 2.

Tabela 2. Parametry sprzetu komputerowego

Element	Stan
System operacyjny	Windows 10, 64 bit
Procesor	Intel Core i5-3210M, 2.50 GHz x2
Pamięć RAM	8 GB
Dysk HDD	SATA III, 500 GB

5.2. Aplikacja testowa

W celu przeprowadzenia badań testowych została utworzona aplikacja internetowa, na podstawie której zostały przeprowadzane testy jednostkowe. Aplikacja ta imituje

działanie internetowego konta bankowego. System ten został napisany w języku C# z użyciem frameworka ASP.NET MVC (ang. model-view-controller) 5.2.6, w wersji frameworka .NET 4.6.1 oraz bazy danych Microsoft SQL Server 2016 LocalDB.

Aplikację można podzielić na dwa moduły: część przeznaczoną dla użytkownika oraz część zastrzeżoną tylko dla administratora systemu. W części użytkownika znajdują się funkcyjności takie jak: rejestracja, logowanie, edytowanie danych konta, przeglądanie wykonanych płatności na swoim koncie, wykonanie nowej płatności. Administrator systemu dodatkowo ma możliwość zarządzania wszystkimi danymi znajdującymi się w bazie danych.

Lista najważniejszego oprogramowania użytego w projekcie została przedstawiona w tabeli 3.

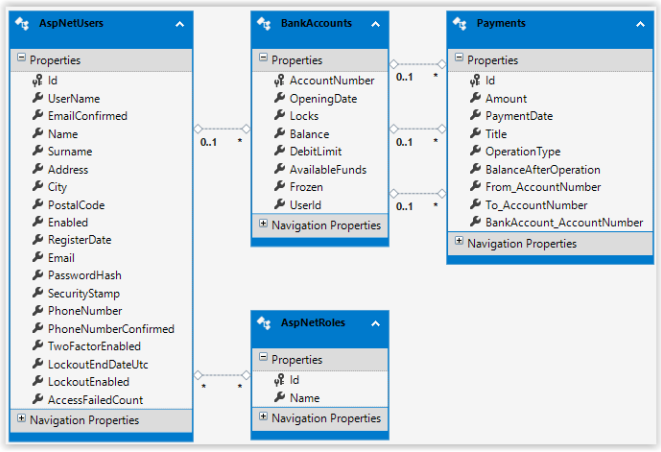
Tabela 3. Narzędzia zastosowane do budowy aplikacji

Narzędzie	Wersja
Język C#	6.0
.NET Framework	4.6.1
Serwer IIS	IIS 10.0. Express
Baza danych	MS SQL Server 2016 LocalDB
Visual Studio	2017 Community
Entity Framework	6.2.0
ASP.NET MVC	5.2.6
Ninject	3.3.4
Castle Core	4.2.1
MSTest	1.3.1
MSTest Adapter	1.3.1
NSubstitute	3.1.0
NUnit	3.10.1
NUnit3TestAdapter	3.10.0
xUnit	2.3.1
xUnit Runner Visual Studio	2.3.1

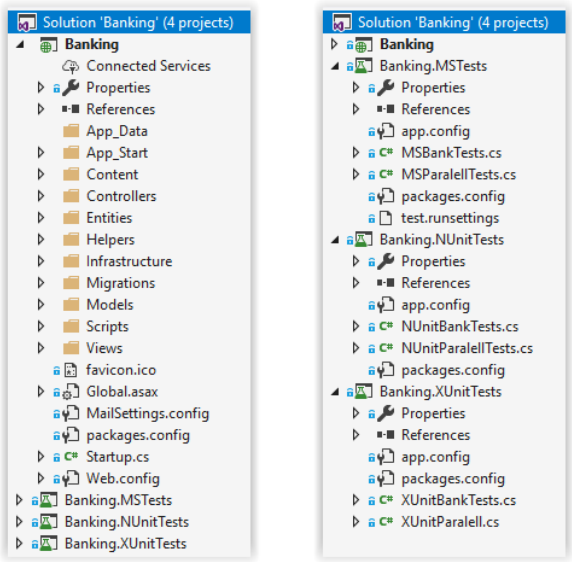
Na rysunku 1 przedstawiony został częściowy schemat bazy danych użytej w aplikacji. Zawiera on najważniejsze tabele, wykorzystane przy pracy z systemem. Główną tabelą jest tabela BankAccounts, która jest odzwierciedleniem konta bankowego. Każdy użytkownik może mieć przypisane wiele ról zdefiniowanych w systemie, które są przechowywane w tabeli AspNetRoles. Użytkownik ma możliwość posiadania wielu kont bankowych, co zostało przedstawione na relacji pomiędzy tabelą użytkowników oraz tabelą kont bankowych. Kolejną encją są płatności, które są bezpośrednio związane z kontem bankowym posiadacza. W tabeli tej znajdują się wszystkie informacje niezbędne do wykonania przelewu na konto innego użytkownika istniejącego w systemie.

5.3. Struktura aplikacji

Na rysunku 2 przedstawiona została struktura badanej aplikacji. Na całe rozwiązanie składają się 4 projekty. Pierwszym jest aplikacja MVC, której struktura widnieje po lewej stronie rysunku. Pozostałymi projektami są projekty testów jednostkowych, w których testowana jest aplikacja główna. Każdy z tych projektów dotyczy innego wybranego narzędzia do przeprowadzania testów jednostkowych, które były badane. Przykładowy projekt testowy został podzielony na część, gdzie wykonywane są testy szeregowe oraz część z testami uruchamianymi równolegle.



Rys. 1. Fragment schematu bazy danch aplikacji



Rys. 2. Struktura rozwiązania w Visual Studio

5.4. Porównanie wydajności testów uruchamianych szeregowo

Na potrzebę porównania szybkości działania frameworków został utworzony test, w którym ma miejsce czasochłonna i obliczeniowo wymagająca operacja. W metodzie tej symulowane jest utworzenie stu tysięcy kont bankowych oraz dodanie ich do kolekcji, jaką jest lista. Zadaniem frameworka jest sprawdzenie, czy którykolwiek z elementów tej listy nie jest równy null. Przykład 1 przedstawia tę metodę napisaną przy pomocy narzędzia MSTest. W pozostałych dwóch frameworkach kod tego testu wygląda identycznie.

W tabeli 4 zostały przedstawione wyniki trzykrotnego uruchomienia testu wydajnościowego. Różnica czasów wykonania pomiędzy tymi narzędziami jest bardzo widoczna. Najlepszy średni czas otrzymał MSTest, którego wynik to 243 ms na wykonanie testu. Na drugim miejscu znajduje się NUnit

z prawie dwukrotnie większym czasem. Natomiast najsłabsze wyniki osiągnęło narzędzie xUnit.net, które miało średni czas prawie 3,5 razy dłuższy od frameworka MSTest z miejsca pierwszego, tj. oraz dwa razy gorszy od frameworka na NUnit z drugiego miejsca.

Przykład 1. Kod testu wydajnościowego

```
[TestMethod]
public void All_Bank_Account_In_List_Are_Not_Null()
{
    List<BankAccount> bankList = new List<BankAccount>();

    for (int i = 0; i < 100000; i++)
    {
        BankAccount ba = new BankAccount(Guid.NewGuid(), 1000);
        bankList.Add(ba);
    }

    CollectionAssert.AllItemsAreNotNull(bankList);
}
```

Odchylenie standardowe obrazuje jak bardzo wartości jakieś zmiennej są rozrzucone wokół średniej. Obliczona wartość odchylenia standardowego w każdej z prób w danym narzędziu dała podobny wynik do różnic czasów wykonania pomiędzy narzędziami.

Tabela 4. Czasy wykonania testu wydajnościowego w poszczególnych narzędziach

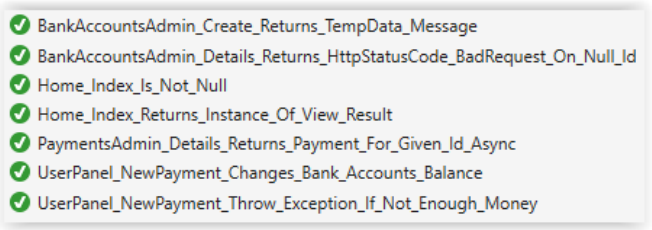
Narzędzie	MSTest			NUnit			xUnit.net		
Próba	1	2	3	1	2	3	1	2	3
Czas wykonania [ms]	242	239	248	425	411	430	814	840	832
Średni czas [ms]	243			422			829		
Odchylenie standardowe [ms]	4,58			9,85			13,32		

Na rysunku 3 przedstawiona jest lista testów jakie były uruchamiane w następnym badaniu. Nazwa każdego z testu jest równocześnie opisem mówiącym, co w danym teście się wykonuje. Nazwa testu ma schemat:

NazwaKontrolera_NazwaAkcji_OczekiwanyWynik.

Testy te kolejno mają za zadanie:

- Utworzenie nowego kontra bankowego z poziomu administratora na podstawie przekazanego modelu tworzy oczekiwany widok, do którego zostaje przekazana tymczasowa informacja pod postacią wiadomości.
- Wywołanie akcji Details konta bankowego po przekazaniu nieprawidłowego argumentu zwraca kod http 400.
- Akcja strony głównej z kontrolera Home zwraca odpowiedni widok, który nie jest wartością null.
- Kontroler Home po wywołaniu akcji Index zwraca obiekt typu ViewResult.
- Wywołanie metody Details z kontrolera PaymentsAdmin zwraca do widoku model, który jest odpowiednim obiektem płatności.
- Wywołanie przez użytkownika płatności powoduje odpowiednią zmianę stanu jego konta bankowego oraz stanu konta bankowego odbiorcy.
- Próba wykonania płatności z wartością większą niż aktualne saldo użytkownika wykonującego przelew powoduje wyrzucenie odpowiedniego wyjątku.



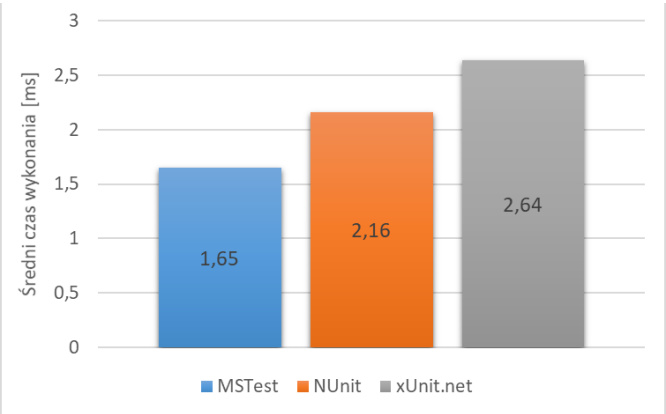
Rys. 3. Lista uruchamianych testów

Tabela 5 zawiera porównanie czasów uruchomienia szeregowo wszystkich testów , jakie znajdują się w projekcie (oprócz wcześniej opisanego testu wydajnościowego). Każdy z narzędzi uruchamiał w sumie 7 testów, których lista została przedstawiona na rysunku 3, a czas trwania takiego uruchomienia był zapisywany w tabeli.

Tabela 5. Czasy uruchomienia wszystkich testów z projektu w poszczególnych narzędziach

Narzędzie	MSTest			NUnit			xUnit.net		
Próba	1	2	3	1	2	3	1	2	3
Czas wykonania [s]	1,57	1,89	1,50	2,17	2,19	2,11	2,62	2,71	2,59
Odchylenie standardowe [s]	0,21			0,04			0,06		

W celu porównania wyników czynność ta była powtarzana trzykrotnie. Wyniki badania, przedstawione w postaci wykresu na rysunku 5, są porównywalne do poprzednio wykonanego testu wydajnościowego. Analizowane narzędzia plasują się na tych samych miejscach tj. najlepiej wypadł MSTest, a najgorzej xUnit.net. Porównanie wyników przeprowadzonych badań pokazało, że tak samo w przypadku uruchamiania jednego testu i w przypadku uruchamiania kilka testów, średnie różnice czasów między narzędziami są porównywalne.



Rys. 4. Porównanie średnich czasów uruchomienia wszystkich testów z projektu

5.5. Porównanie wydajności testów uruchamianych równolegle

Nie jest niczym niezwykłym, że projekt ma tysiące lub dziesiątki tysięcy testów jednostkowych. Deweloperzy chcą mieć pewność, że będą w stanie szybko przeprowadzić wszystkie te testy przed zatwierdzeniem ich kodu. Współczesna maszyna deweloperska może mieć procesor z 8

wirtualnymi rdzeniami oraz od 8 do 16 GB pamięci RAM (lub więcej). Te procesory są marnowane, gdy tylko jeden z nich może zostać przypisany do danego zadania. Dlatego najlepszym sposobem zapewnienia, że testy jednostkowe mogą działać z pełną prędkością komputera hosta, jest uruchomienie wielu z nich w tym samym czasie. Narzędzia porównywane w tej pracy posiadają sposób wykorzystania całej mocy obliczeniowej procesora, jakim jest równoległe wykonywanie testów jednostkowych. Każde z badanych w tej pracy narzędzi posiada możliwość zrównoleglania testów jednostkowych.

W celu zademonstrowania jak wielką rolę odgrywa równoległe uruchamianie testów został utworzony zestaw identycznych metod, które zostaną włączone sekwencyjnie oraz równolegle, po czym porównane zostaną czasy ich wykonania.

Przykład 2. Metoda do zrównoleglenia

```
[Test]
public void Test1()
{
    Thread.Sleep(1000);
    Assert.IsTrue(true);
}
```

Kod z przykładu 2 przedstawia metodę, która ma za zadanie zasymulowanie pewnego odstępu czasu, w tym przypadku jest to jedna sekunda oraz wykonanie asercji, która jest zawsze udana, tj. sprawdzenie, czy wartość true jest prawdą. Kod testu w każdym z narzędzi wygląda tak samo. Porównywane są czasy kolejno pięciu, dziesięciu, dwudziestu oraz pięćdziesięciu takich uruchomionych metod.

Tabela 6. Porównanie czasów uruchomionych testów w sposób szeregowy oraz równoległy

Narzędzie	Uruchomienie	Czas wykonania [s]		
		Szeregowe	Równoległe	Szeregowe/ równoległe
MSTest	5 testów	6,15	3,24	1,90
	10 testów	11,13	4,41	2,52
	20 testów	21,25	6,28	3,38
	50 testów	51,32	14,33	3,58
	Średnia	**22,46**	**7,07**	**2,85**
	SD	**20,24**	**5,00**	**0,78**
NUnit	5 testów	6,32	3,34	1,89
	10 testów	11,36	4,3	2,64
	20 testów	21,48	6,28	3,42
	50 testów	51,63	14,51	3,56
	Średnia	**22,70**	**7,11**	**2,88**
	SD	**20,29**	**5,08**	**0,77**
xUnit.net	5 testów	6,47	4,45	1,45
	10 testów	11,6	4,6	2,52
	20 testów	21,63	6,55	3,30
	50 testów	54,36	14,87	3,66
	Średnia	**23,52**	**7,2**	**2,73**
	SD	**21,51**	**4,93**	**0,98**

W tabeli 6 zostały przedstawione czasy uruchamiania różnej liczby testów w dwóch trybach: szeregowym oraz równoległym. Otrzymane wyniki dla każdego z narzędzi są do siebie bardzo podobne. Wraz ze wzrostem liczby jednocześnie uruchomionych metod testowych wzrasta stosunek czasu uruchomienia szeregowego do równoległego. Iloraz ten dąży do wartości 4, ponieważ testy były uruchamiane na maszynie posiadającej dwa rdzenie procesora, z czego każdy jest dwuwątkowy. Wszystkie narzędzia zostały skonfigurowane tak, aby używać wszystkich możliwych rdzeni komputera, na którym pracuje. Najszybszym z badanych w tym teście narzędzi okazał się MSTest, któremu średnia czasu uruchomienia testów wyniosła 22,46s w trybie szeregowym oraz 7,07s w trybie równoległym. Jednak różnica czasów między badanych frameworkami jest niewielka.

6. Wnioski

Testy jednostkowe powinny uruchamiać się bardzo szybko. Programista powinien być w stanie uruchomić cały zestaw testów jednostkowych w ciągu kilku sekund, a zdecydowanie nie w minutach i godzinach. Powinien mieć możliwość szybkiego uruchomienia ich po zmianie kodu. Stosowanie narzędzi do wspomagających tworzenie takich testów znacznie to upraszcza. Przedstawione w artykule trzy najpopularniejsze do tego narzędzia różnią się miedzy sobą składnią, jednak ich funkcjonalność pozostaje podobna. Jak wynika z przeprowadzonego badania wszystkie z frameworków równie dobrze radzą sobie z równoległym uruchamianiem testów, co jest ogromnym atutem dla każdego z nich. Natomiast w szeregowym, czyli klasycznym, uruchamianiu testów, wyniki nie są już takie jednoznaczne. Pod względem szybkości uruchamiania najlepszym narzędziem okazał się produkt firmy Microsoft, czyli MSTest. Drugie miejsce przypadło dla narzędzia NUnit, natomiast najsłabszy wynik otrzymało narzędzie xUnit.net. Choć szybkość wykonywania testów jednostkowych przez narzędzia do tego przeznaczone, nie jest jedynym kryterium, jakie należałoby brać pod uwagę przy ich porównywaniu, to jednak jest to jeden z ważniejszych mierników, na które zwraca uwagę programista przy wyborze narzędzia, z którym ma zamierzać pracować.

Literatura

[1] Augustyn D. R.: Rozwój narzędzi programowych wspierających automatyzację testów jednostkowych dla technologii .NET. Bazy danych. Rozwój metod i technologii. Bezpieczeństwo, wybrane technologie i zastosowania. WKŁ, Warszawa 2008

[2] Sochacki, G. , Pańczyk, B.: Test-Driven Development jako narzędzie optymalizacji procesu wytwarzania oprogramowania na platformie JEE. Journal of Computer Sciences Institute, 2017

[3] Jabłoński M., Karbowańczyk M.: Automatyczna ocena kompletności projektu programistycznego. Zeszyty Naukowe Wyższej Szkoły Informatyki, 2017

[4] Jureczko M.: Testowanie oprogramowania, Politechnika Wrocławska, 2011

[5] Bukowski M., Paterek P.: Obiekt pozorny. A może coś innego? Kierunek rozwoju testów jednostkowych, TESTER.pl, 2007, nr 10

[6] Certyfikowany tester. Plan poziomu podstawowego Wer. 1.0, przeł. Bereza-Jarociński B., Jaszcz W., Klitenik H.,

Nowakowska J., Sabak J., Seredyn A., Stapp L., Ślęzak P., Żebrowski Ł., SJSI, 2006

[7] Dokumentacja narzędzia MSTest https://msdn.microsoft.com/en-us/library/hh694602.aspx [09.03.2018]

[8] Dokumentacja narzędzia NUnit https://github.com/nunit/docs/wiki [09.03.2018]

[9] Pierwsze wydanie frameworka xUnit.net http://jamesnewkirk.typepad.com/posts/2008/04/xunitnet-10-rel.html [09.03.2018]

[10] Dokumentacja narzędzia xUnit.net https://xunit.github.io/#documentation [09.03.2018]

[11] Osherove R.: Testy jednostkowe. Świat niezawodnych aplikacji. Wydanie II, Helion, 2014

[12] NUnit - asercje https://github.com/nunit/docs/wiki/Assertions [09.03.2018]

[13] Porównanie xUnit.net z innymi frameworkami – asercje http://xunit.github.io/docs/comparisons.html#assertions [31.03.2018]