JOURNAL
COMPUTER SCIENCES INSTITUTE

JCSI 25 (2022) 323–329
Received: 29 June 2022
Accepted: 29 August 2022

Perfomance analysis of libraries for testing web applications on the ASP.NET Core platform

Analiza wydajności bibliotek do testowania aplikacji internetowych na platformie ASP.NET Core

Karol Niedziela*, Jakub Nieradko

Department of Computer Science, Lublin University of Technology, Nadbystrzycka 36B, 20-618 Lublin, Poland

Abstract
This paper conducts a performance analysis of three libraries: XUnit, NUnit, MSTest, aiming to compare the time performance. The performance was checked using load test, synchronous and asynchronous tests. The synchronous and asynchronous tests were divided into groups of 10, 25, 50 and 100 test cases. The tests were carried out using an in-house project written on the ASP.NET Core platform.

Keywords: software engineering; unit tests; performance; ASP.NET Core; C#

Streszczenie
W artykule została przeprowadzona analiza wydajności trzech bibliotek: XUnit, NUnit, MSTest, mająca na celu porównanie wydajności czasowej. Wydajność została sprawdzona przy wykorzystaniu testu obciążeniowego, testów synchronicznych oraz asynchronicznych. Testy synchroniczne oraz asynchroniczne zostały podzielone na grupy po 10, 25, 50 oraz 100 przypadków testowych. Dla każdej grupy zostało wykonane po trzydzieści pomiarów czasowych. Badania zostały wykonane przy pomocy autorskiego projektu napisanego na platformie ASP.NET Core.

Słowa kluczowe: inżynieria oprogramowania; testy jednostkowe; wydajność; ASP.NET Core; C#

*Corresponding author
Email address: **karol.niedziela@company.com** (K. Niedziela)

1. Wstęp

Na przestrzeni lat oprogramowanie komputerowe ewoluowało i zmieniało się wraz z rozwojem technologii. Obecny rynek IT oferuje coraz bardziej rozbudowane systemy informatyczne, których rozbudowa i utrzymanie wymaga odpowiednich narzędzi umożliwiających unikania błędów, pojawiających się w trakcie użytkowania aplikacji przez klientów. Wraz z rosnącym zapotrzebowaniem na skalowalne oprogramowania, powstało wiele technik, które miały zapewnić kod o wysokiej jakości i małej liczbie błędów. W tym celu zostały opracowane narzędzia umożliwiające sprawdzanie aplikacji bez konieczności manualnego testowania działania logiki kodu, które dotychczas było pierwszym etapem weryfikacji działania systemu. Jedną z tych technik jest pisanie testów jednostkowych [1].

Testowanie jednostkowe jest procesem rozwoju oprogramowania, w którym najmniejsze testowalne części aplikacji, zwane jednostkami, są indywidualnie i niezależnie sprawdzane pod kątem poprawności działania. Ta metoda testowania jest wykorzystywana podczas procesu tworzenia oprogramowania przez programistów. Test jednostkowy składa się zazwyczaj z trzech etapów, zdefiniowanych przez zasadę AAA (ang. Arrange – Act – Assert). W pierwszym etapie (ang. Arrange) przygotowywane są wszystkie dane wejściowe oraz warunki wstępne. W następnym etapie (ang. Act) zostaje wywołana metoda, która aktualnie jest testowana. W ostatnim etapie (ang. Assert) zostaje sprawdzona zgodność pomiędzy zwróconymi, a oczekiwanymi wartościami.

Głównym celem przeprowadzania testów jest sprawdzenie poprawności działania określonych funkcjonalności systemu. Aby testy były skuteczne, powinny być przeprowadzane głównie w miejscach występowania potencjalnych błędów oraz w miejscach, gdzie logika biznesowa jest złożona. Testowanie jednostkowe jest ważnym krokiem w procesie rozwoju oprogramowania, ponieważ jeśli jest wykonane poprawnie, pozwala wykryć wczesne błędy w kodzie, które mogą być trudniejsze do znalezienia w późniejszych etapach testowania [2]. Tak wykonane testy pozwalają uniknąć niezadowolenia klientów oraz użytkowników. Poza tym, są źródłem informacji o działaniu aplikacji dla osób pracujących aktualnie nad systemem oraz dla przyszłych programistów. Poprzez użycie narzędzi przeznaczonych do tworzenia testów, czas poświęcony na ich pisanie zostaje znacznie skrócony i ułatwia poprawne identyfikowanie przypadków testowych.

Celem pracy jest analiza wydajności bibliotek do testowania aplikacji internetowych na platformie ASP.NET Core. Celem analizy jest wskazanie czy istnieje biblioteka osiągająca znacznie lepsze rezultaty czasowe w różnych typach badań. Analizę przeprowadzono z wykorzystaniem aplikacji internetowej napisanej przy użyciu platformy .NET w wersji 6.0 oraz języka C# w wersji 10. W tym celu zbadano trzy najpopularniejsze biblioteki: XUnit, NUnit i MSTest. Zostały one porównane na podstawie szybkości czasu wykona-

nia testu obciążeniowego, testów synchronicznych oraz asynchronicznych. Przeprowadzone badania pozwolą na poznanie różnic pomiędzy wymienionymi bibliotekami. Przeprowadzona analiza zapoznaje z metodyką pisania testów oraz umożliwi wybór odpowiedniego narzędzia do ich implementacji. Otrzymane wyniki pozwolą wskazać najszybszą bibliotekę w badanych obszarach.

2. Przegląd literatury

Została wykonana analiza dostępnych materiałów badawczych przed przeprowadzeniem badań, które są tematem artykułu. Dotychczas opublikowano niewiele prac poświęconych analizie porównawczej bibliotek. Jednym z artykułów poruszających tematykę jest praca z Journal of Computer Science Institute pod tytułem "Porównanie wybranych narzędzi do przeprowadzenia testów jednostkowych" [3]. W artykule analizie zostały poddane trzy biblioteki: MSTest, NUnit i XUnit. Celem analizy było porównanie szybkości wykonywania testów w sposób szeregowy oraz równoległy. Testy zostały zaimplementowane dla aplikacji imitującej działanie internetowego konta bankowego. W pracy zostały krótko opisane wykorzystane biblioteki oraz pojęcia ściśle związane z testowaniem oprogramowania. Aby porównać szybkość działania każdej biblioteki została przetestowana funkcjonalność odpowiadająca za utworzenie stu tysięcy kont bankowych, a następnie dodanie ich do kolekcji. Najlepszy średni czas uzyskała biblioteka MSTest. Na tle przeprowadzonych badań najlepszą biblioteką okazała się biblioteka MSTest, następnie NUnit, natomiast najgorsze wyniki uzyskała biblioteka XUnit.

Poza wymienionym artykułem, większość prac skupia się na procesie wytwarzania oprogramowania, gdzie jednym z filarów są testy jednostkowe oraz wpływ testów jednostkowych na jakość kodu. Na Uniwersytecie Technologicznym Chalmers zostało przeprowadzone badanie [4], które miało odpowiedzieć na dwa pytania badawcze. Pierwsze pytanie brzmiało "W jakim stopniu pokrycie testami jednostkowymi jest skorelowane z liczbą błędów?". Drugie pytanie było związane z korelacją między pokryciem kodu testami jednostkowymi, a postrzeganą jakością kodu. Badanie składało się z dwóch niezależnych części. Pierwszą częścią była ankieta przeprowadzona na dwóch firmach w Szwecji oraz trzech przedsiębiorstwach i uniwersytecie w Brazylii. Drugą częścią było studium przypadku. Ankieta została przeprowadzona na 241 członkach zespołów programistów, gdzie nie było jasno zdefiniowanej procentowej wartości, jaką powinien być pokryty kod. W związku z tym, testy jednostkowe były pisane w różnych ilościach dla różnych części systemu. Ankietowani otrzymali do wypełnienia trzyczęściową ankietę. Pierwsza i druga część ankiety obejmowała sposób implementacji przypadku testowego względem głównej testowanej funkcjonalności. Trzecia część dotyczyła uruchamiania testów po zakończeniu zadania oraz przed integracją danej części systemu. Odpowiedzi udzieliło 201 osób. Studium przypadku obejmowało sprawdzenie pokrycia kodu w plikach przy użyciu narzędzia do pomiaru procentowego pokrycia kodu. Dodatkowo sprawdzono liczbę błędów kodu z małym procentowym pokryciem, wykorzystując zgłoszenia o błędach w systemie kontroli wersji. Na podstawie uzyskanych wyników w odniesieniu do pierwszego pytania badawczego nie znaleziono korelacji. Natomiast dla drugiego pytania badawczego korelacja była niska. Pozwala to stwierdzić, że jakość systemu nie musi być niska, jeśli pokrycie kodu testami jednostkowymi nie jest wysokie. Kolejna pozycja porusza tematykę mocnych i słabych stron stosowania testów jednostkowych oraz przyczynę ich pisania [5]. Badanie zostało przeprowadzono z wykorzystaniem ankiety przygotowanej dla kilku firm. Głównym powodem pisania testów jednostkowych były wymagania zewnętrzne. Testy miały służyć jako dokumentacja techniczna systemu. Innym częstym powodem było stosowanie przez organizację metodyki zwinnej (ang. Agile), która zakłada pisanie testów jednostkowych. Jako wady wskazano brak miar pozwalających na walidację użyteczności testów jednostkowych oraz trudność w wybraniu obszarów systemu, które powinny być testowane. Kolejną pozycją jest pytanie zadane na najpopularniejszym forum programistycznym StackOverflow. Pytanie brzmiało „Jakie są dobre sposoby, aby przekonać sceptycznych programistów w zespole o wartości testów jednostkowych?" [6]. Najczęściej pojawiającymi się odpowiedziami były:

- możliwość szybkiego sprawdzenia dużych zmian w kodzie i zapewnienie, że zmieniony lub dopisany kod działa prawidłowo,
- uzyskanie kodu o lepszej jakości,
- uzyskanie wizualnej informacji zwrotnej o poprawności działania (testy, które uzyskały zakładany rezultat są oznaczone zielonym kolorem),
- uzyskanie dokumentacji oraz przykładów, za co dany kod jest odpowiedzialny,
- zmniejszenie liczby błędów.

Następne badanie pokazuje, że przeprowadzenie testów zapewnia skalowalne i niepodatne na błędy oprogramowanie [7]. Porusza tematykę metodyki TDD (ang. Test Driven Development), powstałej wraz ze wzrostem popularności implementacji warstwy testowej i rozwojem procesu wytwarzania oprogramowania. Metodyka ta odwraca klasyczny model, gdzie najpierw zostaje napisany kod, a następnie zostaje on przetestowany. Dzięki zastosowaniu TDD, kod jest pokryty w 100% przez testy jednostkowe, co pozwala zapewnić wysoką jakość i minimalną liczbę błędów w samej logice kodu.

Dodatkowo istnieje kilka pozycji literaturowych opisujących sposób implementacji testów jednostkowych. Według książki Vladimira Khorikova [8], test jednostkowy jest zautomatyzowanym testem, które ma za zadanie weryfikację małej ilości kodu w szybkim czasie. Kolejną pozycją jest książka zawierająca część teoretyczną oraz część praktyczną [9]. W części teoretycznej wytłumaczono, czym jest testowanie jednostkowe, jakie plusy płyną z pisania testów jednostkowych i jaki wpływ mają one na system, ale także i programistów. W części praktycznej omówiono podstawy różnych bibliotek. Książka oferuje także instrukcje krok po

kroku w zakresie podstawowego tworzenia testów jednostkowych. Zawiera przykłady kodu z języków programistycznych takich jak Java, C++ oraz C#.

3. Badane biblioteki

W poniższym rozdziale zostały opisane analizowane biblioteki.

3.1. NUnit

NUnit jest open-source'ową biblioteką przeniesioną z JUnit [10]. Najnowsza wersja Nunit to NUnit3, która została przepisana z wieloma nowymi funkcjami i posiada wsparcie dla wszystkich platform .NET. Biblioteka NUnit jest licencjonowana na zasadach licencji MIT. Wcześniejsze wersje wydania korzystały z licencji NUnit. Obie licencje pozwalają na używanie tej biblioteki w komercyjnych jak i darmowych aplikacjach. W momencie pisania pracy, biblioteka NUnit została pobrana ponad 174 milionów razy. W momencie pisania pracy, na StackOverflow było około 26000 pytań oznaczonych jako NUnit.

3.2. XUnit

XUnit jest darmową, open-source'ową biblioteką do przeprowadzania testów jednostkowych [11]. Biblioteka jest tworzona przez społeczność. Została napisana przez twórcę jednej z wersji biblioteki NUnit. Spośród trzech analizowanych bibliotek jest najnowsza i najbardziej łatwa do rozbudowy. Biblioteka jest licencjonowana na zasadach licencji Apache 2. W momencie pisania pracy, biblioteka XUnit została pobrana ponad 200 milionów razy. Do tej pory na StackOverflow znajduje się około 1000 pytań oznaczonych jako XUnit.

3.3. MSTest

MSTest jest domyślną biblioteką do pisania testów, która jest dostarczana wraz z Visual Studio. Biblioteka jest wieloplatformowa, co umożliwia wykorzystywanie jej dla wszystkich platform .NET [12]. Pierwsza wersja MSTest nie była typu open-source. Jednakże wraz z wyjściem wersji drugiej, biblioteka jest typu open-source. Kod jest dostępny na platformie GitHub i jest licencjonowana na zasadach licencji MIT. W momencie pisania pracy, biblioteka MSTest została pobrana ponad 130 milionów razy. W momencie pisania pracy, na StackOverflow jest około 11000 pytań oznaczonych jako MSTest.

4. Przebieg badań

W poniższym rozdziale zostało opisane środowisko testowe wraz z użytymi aplikacjami oraz narzędziami. W celu porównania wydajności zostały zmierzone czasy wykonania testu wydajnościowego, testów synchronicznych oraz asynchronicznych.

4.1. Środowisko testowe

Przeprowadzone badania zostały wykonane przy pomocy urządzeń, których parametry zostały przedstawione w Tabeli 1.

Tabela 1: Komponenty środowiska testowego

Komponent	Komputer
System operacyjny	Windows 10, 64 bit
Procesor	Intel Core i5-10400
Dysk SSD	ADATA 512GB M.2 PCIe
Pamięć RAM	16 GB

4.2. Aplikacja testowa

Do przeprowadzenia badań została przygotowana aplikacja internetowa, dla której zostały wykonane testy jednostkowe. Główną funkcjonalnością aplikacji jest wsparcie zarządzania codziennymi potrzebami domowników. Aplikacja umożliwia współdzielenie informacji o listach zakupów, terminach rachunków do zapłacenia. Projekt aplikacji został podzielony na cztery warstwy, zgodnie z założeniami dotyczące czystej architektury. Poszczególne warstwy to:
• domenowa,
• aplikacji,
• infrastruktury,
• API (Application Programming Interface – interfejs programistyczny aplikacji).
Aplikacja została napisana w języku C# w wersji 10 oraz bazy danych MySQL. W Tabeli 2 zostały przedstawione użyte narzędzia do przeprowadzenia badań.

Tabela 2: Narzędzia wykorzystane na środowisku testowym

Narzędzie	Wersja
Język	10.0
.NET 6.0	6.0
Baza danych	MySQL 8.0.23
Visual studio	2022 Enterprise
XUnit	2.4.1
NUnit	3.13.2
MSTest	16.11.0
Shoudly	4.0.3

4.3. Test obciążeniowy

W celu porównania wydajności, pierwszym badaniem było przygotowanie testu, wymagającego dłuższego czasu wykonywania. W każdej z badanych bibliotek został przygotowany test jednostkowy dla metody, która odpowiada za dodanie do listy produktów określonej liczby elementów, w tym przypadku 5000. Dla każdej biblioteki test został uruchomiony trzydziestokrotnie, pomijając pierwsze uruchomienie. Dla każdej biblioteki został obliczony średni czas wykonania testów oraz zostało obliczone odchylenie standardowe.

4.4. Testy synchroniczne

Badanie miało na celu zmierzenie czasów wykonywania testów synchronicznych. Synchronicznie wykonywany kod oznacza, że linie kodu są wykonywane po kolei i dopiero po zakończeniu jednej linii następuje odblokowanie następnej. Innymi słowy, aby przejść do na-

stępnej linii, trzeba poczekać na zakończenie poprzedniej. Badanie zostało podzielone na grupy 10, 25, 50 i 100 testów dla każdej z trzech badanych bibliotek. Dla każdej grupy przypadków testowych zostało przeprowadzonych po 30 pomiarów czasowych.

4.5. Testy asynchroniczne

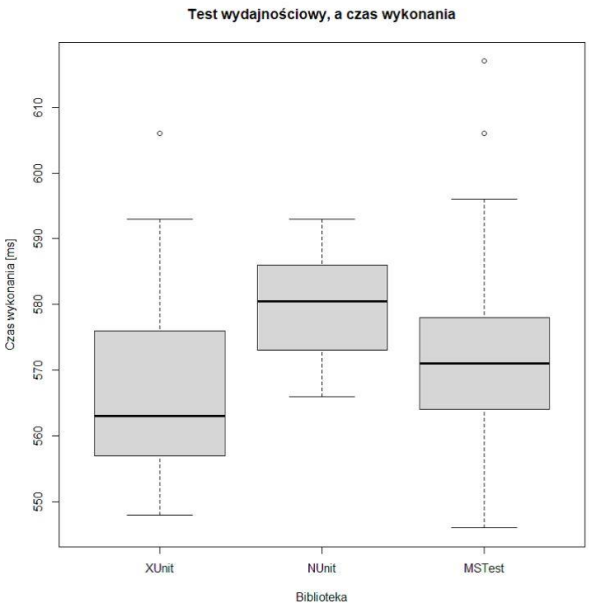
Następnym przeprowadzonym badaniem było wykonanie testów asynchronicznych i zmierzenie czasów ich wykonywania. Programowanie asynchroniczne to model programowania, w którym operacje wykonywane są w sposób niesekwencyjny. Model ten jest przeciwieństwem programowanie synchronicznego. W takim modelu operacje wymagającego czasu nie blokują programu, a kontynuują jego wykonywanie. Po zakończeniu operacji sukcesem lub porażką, wywoływane jest zdarzenie, które sygnalizuje, że wynik jest gotowy do wykonania na nim kolejnych kroków. Bez programowania asynchronicznego użytkownik miałby zablokowany graficzny interfejs. Dzięki programowaniu asynchronicznemu użytkownik może wykonywać swoje zadania w aplikacji, podczas gdy procesy działają w tle, co zwiększa komfort użytkowania. Przykładem problemu może być pobieranie danych z bazy danych zawierającej miliony rekordów. Aby zbadać jak radzą sobie badane biblioteki z kodem wykonywanym asynchronicznie, zostały przygotowane testy w grupach po 10, 25, 50 oraz 100 testów dla każdej z trzech badanych bibliotek. Dla każdej grupy przypadków testowych zostało przeprowadzonych po trzydzieści pomiarów czasowych.

5. Wyniki i analiza badań

W poniższym rozdziale zostały przedstawione wyniki przeprowadzonych badań opisanych w rozdziale 4.

5.1. Test obciążeniowy

Otrzymane pomiary zostały przedstawione na Rysunku 1, wykorzystując wykresy pudełkowe.



Rysunek 1: Wykresy pudełkowe dla testu obciążeniowego.

Ponadto została obliczona średnia czasu wykonywania testu oraz odchylenie standardowe. W Tabeli 3 zostały przedstawione wyniki.

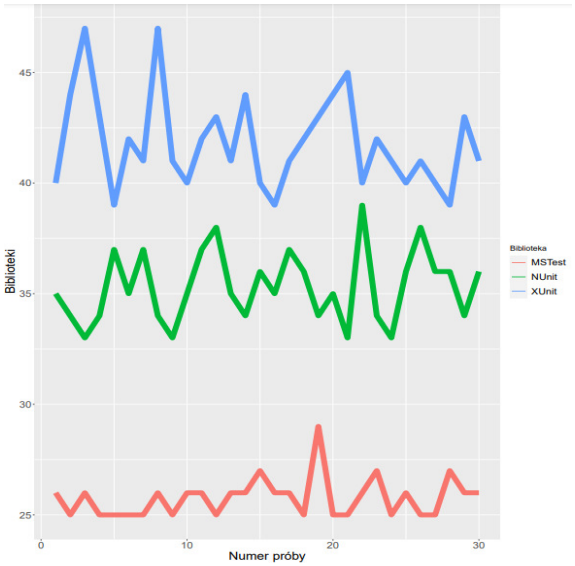
Tabela 3: Średnia oraz odchylenie standardowe testu

Biblioteka	Średni czas [ms]	Odchylenie standar-
XUnit	568,17	14,77
NUnit	579,77	7,85
MSTest	573,43	14.58

Na podstawie otrzymanych rezultatów można zauważyć, że trzy badane biblioteki wykonały test w podobnym czasie. Jednak najszybszy średni czas uzyskała biblioteka XUnit, natomiast najgorszy średni czas biblioteka NUnit. W przypadku odchylenia standardowego najmniejsze odchylenia uzyskała biblioteka NUnit, a najwyższe odchylenie biblioteka XUnit. Pomimo wyników różniących się maksymalnie o kilkanaście milisekund oraz najlepszego średniego czasu wykonania biblioteki XUnit. Biblioteka NUnit ma najmniejsze odchylenie, co przy każdorazowym uruchamianiu testów na przykład podczas wdrażania zmian na konkretnym środowisku może przemawiać na korzyść tejże biblioteki.

5.2. Testy synchroniczne

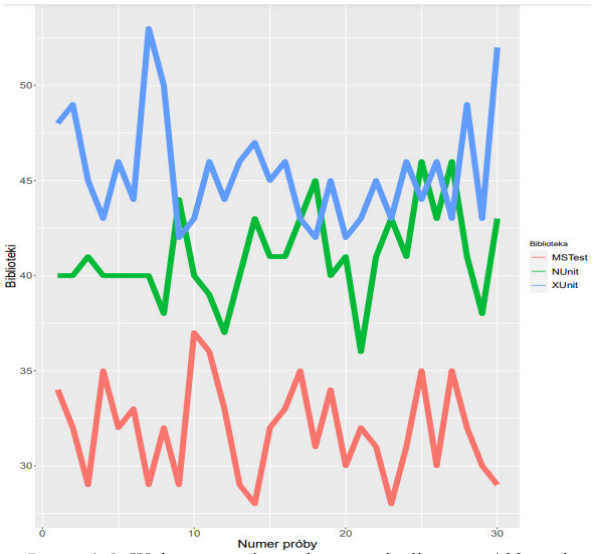
Na Rysunku 2 zostały przedstawione wykresy z wynikami pomiaru czasów dla grupy pięćdziesięciu testów synchronicznych, dla każdej z trzech badanych bibliotek w formie wykresu liniowego. Wykres koloru czerwonego odpowiada bibliotece MSTest, wykres koloru zielonego bibliotece NUnit, a wykres koloru niebieskiego bibliotece XUnit.



Rysunek 2: Wykresy czasów wykonywania dla grupy 50 testów synchronicznych.

Rysunek 3 przedstawia wykresy z wynikami pomiaru czasów dla grupy stu testów synchronicznych, dla każdej z trzech badanych bibliotek. Wykres koloru czerwonego odpowiada bibliotece MSTest, wykres koloru

zielonego bibliotece NUnit, a wykres koloru niebieskiego bibliotece XUnit.



Rysunek 3: Wykresy czasów wykonywania dla grupy 100 testów synchronicznych.

W Tabeli 4 został przedstawiony średni czas i odchylenie standardowe dla każdych grup testów oraz trzech badanych bibliotek.

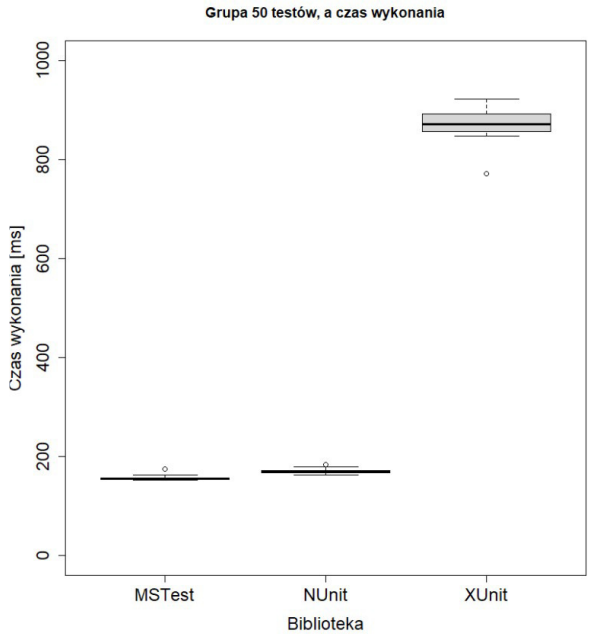
Tabela 4: Średnia oraz odchylenie standardowe dla różnych

Biblioteka	Grupa	Średni czas	Odchylenie
MSTest	10	13,17	0,46
	25	19,63	0,71
	50	25,77	0,90
	100	31,88	2,49
NUnit	10	21,07	0,25
	25	27,9	1,21
	50	35,3	1,62
	100	41,03	2,40
XUnit	10	21,07	0,52
	25	27,47	1,28
	50	41,83	2,13
	100	45,43	2,88

Na podstawie otrzymanych wyników dla poszczególnych bibliotek można wywnioskować, że jeśli testowany jest czysty kod języka wyniki są bardzo zbliżone. Najgorsze wyniki uzyskała biblioteka XUnit, gdzie jest dwukrotnie wolniejsza niż biblioteki NUnit oraz MSTest. Jedynie przy bardzo małej liczbie testów biblioteka XUnit uzyskuje podobne czasy do biblioteki NUnit. Dla każdej badanej grupy przypadków testowych najlepsze średnie rezultaty czasowe uzyskała biblioteka MSTest.

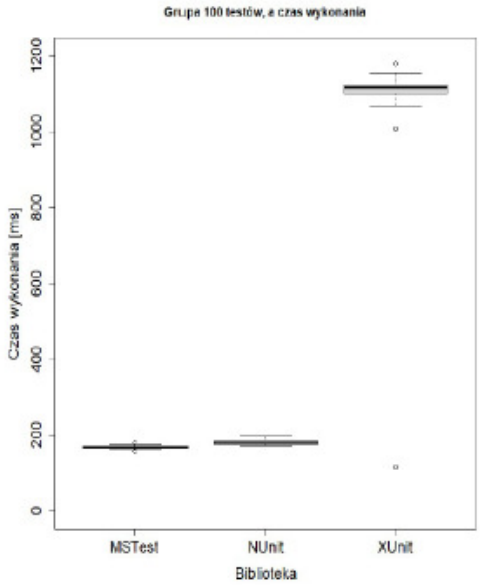
5.3. Testy asynchroniczne

Na Rysunku 4 został przedstawiony wykres z wynikami pomiaru czasów dla grupy pięćdziesięciu testów asynchronicznych.



Rysunek 4: Wykres pudełkowy dla grupy 100 testów asynchronicznych.

Na Rysunku 5 został przedstawiony wykres z wynikami pomiaru czasów dla grupy stu testów synchronicznych dla każdej z trzech badanych bibliotek.



Rysunek 5: Wykres pudełkowy dla grupy 100 testów asynchronicznych.

Dla każdej badanej grupy przypadków testowych zostały policzone średnia czasów wykonywania oraz odchylenie standardowe. Wyniki zostały przedstawione w Tabeli 5.

Tabela 5: Średnia oraz odchylenie standardowe dla różnych grup testów asynchronicznych

Biblioteka	Grupa	Średni czas	Odchylenie
MSTest	10	130,73	3,01
	25	138,57	3,15
	50	155,9	4,23
	100	167,03	4,59
NUnit	10	139,67	3,87
	25	150,27	4,38
	50	170,27	4,74
	100	180,43	6,00
XUnit	10	707,53	26,82
	25	767,83	26,47
	50	873,33	28,66
	100	1077,84	31,23

W porównaniu do wcześniej wykonanych testów synchronicznych widać dużą różnicę pomiędzy biblioteką XUnit, a biblioteką NUnit i MSTest. Dla każdej z badanych grup przypadków testowych biblioteka XUnit osiąga najwyższe średnie czasy oraz ma duże wahania pomiędzy wykonywaniem poszczególnych prób. Duża dysproporcja może być spowodowana samą budową biblioteki XUnit i jej różnicami względem dwóch pozostałych. Biblioteka NUnit została stworzona, wzorując się na bibliotece MSTest. Natomiast biblioteka XUnit stosuje inne podejście. Przykładowo biblioteka NUnit uruchamia wszystkie testy używając instancji tej samej klasy, natomiast biblioteka XUnit tworzy osobną instancję klasy dla każdego pojedynczego testu. Ponadto biblioteki NUnit i MSTest używają specjalnych znaczników, określających metodę służącą do zwalniania zasobów. W przypadku XUnit wykorzystywany jest interfejs wbudowany w język C#, który może wykonywać dodatkowe operacje, wpływające na czas wykonania testu. Pomimo dużej przyjemności z pisania testów w bibliotece XUnit, otrzymane rezultaty wskazują na dużą dysproporcję. W przeprowadzonym badaniu największą badaną grupą było 100 przypadków testowych. Dla porównania dla aplikacji używanych przez klientów, często liczba testów sięga kilku, a nawet kilkunastu tysięcy przypadków testowych. W związku z tym, używanie biblioteki XUnit może spowodować kilkusekundowe opóźnienie w czasie wykonywania testów z porównaniem do biblioteki NUnit i MSTest. Ma to duże znaczenie, ponieważ testy są najczęściej uruchamianie przy każdorazowym dodawaniu zmian w kodzie źródłowym.

5.4. Analiza wariancji

Kolejnym etapem było przeprowadzenie analizy wariancji. Analizie wariancji poddano test obciążeniowy, grupy testów synchronicznych i asynchronicznych. Z powodu otrzymania wartości p value znacząco odbiegającej od poziomu istotności przyjęto zapis < 0.05 dla wartości mniejszej od 0.05 i zapis > 0.05 dla wartości większej niż 0.05. Pierwszym etapem było wykonanie analizy wariancji dla testu obciążeniowego.

Tabela 6: Wyniki post-hoc dla grupy 50 testów asynchronicznych

Biblioteka	Biblioteka	Wartość p value
NUnit	XUnit	< 0.05
MSTest	XUnit	> 0.05
MSTest	NUnit	> 0.05

Na podstawie otrzymanych wyników z analizy wariancji można stwierdzić, że biblioteki MSTest i XUnit oraz MSTest i NUnit nie różnią się istotnie statystycznie. Kolejno została przeprowadzona analiza wariancji dla każdej grupy testów synchronicznych. W Tabeli 7 zostały przedstawione rezultaty dla grupy 50 testów synchronicznych.

Tabela 7: Wyniki post-hoc dla grupy 50 testów synchronicznych

Biblioteka	Biblioteka	Wartość p value
NUnit	XUnit	< 0.05
MSTest	XUnit	< 0.05
MSTest	NUnit	< 0.05

W Tabeli 8 zostały przedstawione rezultaty dla grupy 100 testów synchronicznych.

Tabela 8: Wyniki post-hoc dla grupy 100 testów synchronicznych

Biblioteka	Biblioteka	Wartość p value
NUnit	XUnit	< 0.05
MSTest	XUnit	< 0.05
MSTest	NUnit	< 0.05

Na podstawie otrzymanych wyników z analizy wariancji można stwierdzić, że tylko dla bibliotek NUnit i XUnit dla grupy 10 i 25 testów synchronicznych, biblioteki różnią się statystycznie istotnie. Dla pozostałych przypadków biblioteki nie różnią się istotnie statystycznie. Następnie została przeprowadzona analiza wariancji dla testów asynchronicznych. W Tabeli 9 zostały przedstawione rezultaty dla grupy 50 testów asynchronicznych.

Tabela 9: Wyniki post-hoc dla grupy 50 testów asynchronicznych

Biblioteka	Biblioteka	Wartość p value
NUnit	XUnit	< 0.05
MSTest	XUnit	< 0.05
MSTest	NUnit	< 0.05

W Tabeli 10 zostały przedstawione rezultaty dla grupy 100 testów asynchronicznych

Tabela 10: Wyniki post-hoc dla grupy 100 testów asynchronicznych

Biblioteka	Biblioteka	Wartość p value
NUnit	XUnit	< 0.05
MSTest	XUnit	< 0.05
MSTest	NUnit	< 0.05

Na podstawie otrzymanych wyników z analizy wariancji, we wszystkich przypadkach poza jednym, biblioteki różnią się istotnie statystycznie. Wyjątkiem jest grupa 10 testów asynchronicznych, gdzie biblioteki MSTest oraz NUnit nie różnią się istotnie statystycznie.

6. Podsumowanie

W celu porównania trzech najpopularniejszych bibliotek zostały wykonane pomiary czasów dla testu obciążeniowego, testów synchronicznych oraz asynchronicznych. Dla testu obciążeniowego zostało wykonanych po 30 pomiarów dla każdej z badanych bibliotek. Testy synchroniczne i asynchroniczne zostały podzielone na grupy po 10, 25, 50 oraz 100 testów. Dla każdej grupy zostało przeprowadzonych po 30 pomiarów. Dla przygotowanego testu obciążeniowego wszystkie badane biblioteki uzyskały przybliżone wyniki. Najlepszy średni czas uzyskała biblioteka XUnit.

W kolejnym etapie badań porównano czasy wykonania w grupach 10, 25, 50 i 100 testów synchronicznych. Wszystkie badane biblioteki w każdej grupie uzyskały niskie rezultaty. Jednak najlepszy średni czas w każdej grupie uzyskała biblioteka MSTest. Następnie zostały porównane czasy wykonywania testów asynchronicznych. Na podstawie przeprowadzonych badań można stwierdzić, że biblioteka XUnit osiąga znacznie gorsze wyniki czasowe w porównaniu do biblioteki NUnit oraz MSTest. Biblioteki NUnit i MSTest uzyskały bardziej zbliżone wyniki względem siebie. Jednakże najszybsza w badaniu przeprowadzonym dla testów asynchronicznych była biblioteka MSTest. Tak duża dysproporcja może wynikać z różnic w budowie biblioteki. Tworzenie nowej instancji dla każdego testu jednostkowego w przypadku biblioteki XUnit może mieć przełożenie w czasie wykonywania testów.

Czas wykonywania określonej grupy testów jest aspektem, który znacząco może wpłynąć na wydajność pracy programistów. W przeprowadzonych badaniach największą uruchamianą grupą za jednym razem była grupa 100 testów. Już dla tej grupy występowały kilkunasto lub kilkudziesięcio milisekundowe różnice. W aplikacjach komercyjnych liczba testów jest znacznie większa. Na podstawie przeprowadzonych badań najszybsza z badanych bibliotek jest biblioteka MSTest. Uzyskała najlepsze średnie czasy dla każdej grupy z przeprowadzonych testów synchronicznych i asynchronicznych.

Literatura

[1] J. Albahari, C# 9.0 in a nutshell: The definitive reference, O'Reilly Media, Sebastopol, 2021.

[2] R. Osherove, The Art of Unit Testing, Second Edition with examples in C#, Manning Publications, Shelter Island, 2013.

[3] P. Strzelecki, M. Skublewska-Paszkowska, Pórownanie wybrancych narzędzi do przeprowadzania testów jednostkowych, Journal of Computer Sciences Institute 9 (2018) 334-339.

[4] L. Gren, A. Vard, On the relation between unit testing and code quality, Proceedings of the 43rd Euro Micro Conference on Software Engineering and Advanced Applications (SEAA), IEEE Xplore (2017) 52-56.

[5] P. Runeson, A survey of unit testing practices IEEE software 23, 4 (2006) 22–29.

[6] Czy testowanie jednostkowe przynosi efekty?, https://stackoverflow.com/questions/67299/is-unit-testing-worth-the-effort, [21.05.2022].

[7] G. Sochacki, B. Pańczyk, Test-Driven Development jako narzędzie optymalizacji procesu wytwarzania oprogramowania na platformie JEE, Journal of Computer Sciences Institute, 4 (2017) 112-116.

[8] V. Khorikov, Unit testing principles, practices and pattterns, Simon and Schuster, New York, 2020.

[9] P. Hamill, Unit Test Frameworks: Tools for High-Quality Software Development, O'Reilly Media, Sebastopol, 2004.

[10] Dokumentacja XUnit, https://xunit.net/#documentation, [21.05.2022].

[11] Dokumentacja NUnit, https://docs.nunit.org/, [21.05.2022].

[12] Dokumentacja MSTest, https://docs.microsoft.com/en-us/dotnet/api/microsoft.visualstudio.testtools.unittesting?view=visualstudiosdk-2022, [21.05.2022].