JOURNAL
COMPUTER SCIENCES INSTITUTE

JCSI 23 (2022) 123–127
Received: 24 March 2022
Accepted: 6 April 2022

Performance Comparison of Unit Test Isolation Frameworks

Porównanie wydajności szkieletów programistycznych do izolacji kodu w testach jednostkowych

Mateusz Domański*, Michał Dołęga*, Grzegorz Kozieł

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

Abstract
The goal of unit testing is to verify that individual parts of application are correct. For external dependencies mock objects should be created. This process is supported by dedicated libraries. The paper compares three unit test isolation frameworks for .NET: Moq 4.16.1, FakeItEasy 7.2.0 and NSubstitute 4.2.2. The performance research included comparison of benchmark execution times and comparison of unit test execution times in which selected methods of tested libraries were used. The results are shown on box plots. The analysis shows that Moq is optimal mocking framework.

Keywords: code isolation; unit testing; mock objects

Streszczenie
Celem testów jednostkowych jest weryfikacja poprawności działania pojedynczych elementów programu. Dla zależności wychodzących poza ten zakres powinny zostać utworzone atrapy obiektów. Proces ten wspomagają dedykowane biblioteki. W niniejszej pracy przedstawiono porównanie trzech szkieletów programistycznych do izolacji kodu w testach jednostkowych dla platformy programistycznej .NET: Moq 4.16.1, FakeItEasy 7.2.0 oraz NSubstitute 4.2.2. Badanie wydajności objęło porównanie czasów wykonania testów wydajnościowych oraz porównanie czasów wykonania testów jednostkowych, w których wykorzystane zostały wybrane metody badanych bibliotek. Wyniki przedstawiono na wykresach pudełkowych. Z przeprowadzonej analizy wynika, że optymalnym szkieletem programistycznym do tworzenia atrap obiektów jest Moq.

Słowa kluczowe: izolacja kodu; testy jednostkowe; atrapy obiektów

*Corresponding author
Email address: mateusz.domanski1@pollub.edu.pl (M. Domański), michal.dolega@pollub.edu.pl (M. Dołęga)

1. Wstęp

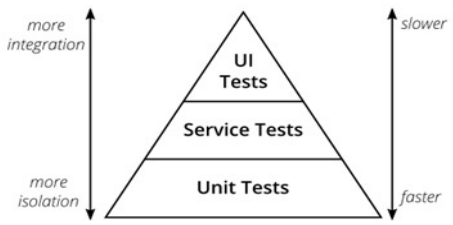
Testy jednostkowe są jednym ze sposobów poprawy jakości oprogramowania. Są metodami testowymi, które weryfikują poprawność pojedynczych metod programu. Ich niewątpliwą zaletą jest bardzo krótki czas wykonania oraz możliwość wykrywania błędów podczas początkowych faz tworzenia oprogramowania. W dużej mierze ma to wpływ na zmniejszenie kosztów projektu, ponieważ im defekt zostanie wcześniej zauważony, tym koszt jego naprawy będzie mniejszy. Dodatkowo testy jednostkowe wpływają także na skrócenie całego procesu testowania. Związane jest to z tym, że zespoły złożone z testerów mogą skupić się w większej mierze na sprawdzaniu funkcjonalności oraz integracji pomiędzy modułami.

Testy jednostkowe nie powinny wykorzystywać zewnętrznych zależności w celu sprawdzenia poprawności działania aplikacji. Dlatego istnieją biblioteki, które ułatwiają izolacje testów poprzez tworzenie atrap obiektów. Są to obiekty, które imitują wymagane zależności. Programista może zdefiniować zachowanie atrapy oraz zwracane przez nią wartości na potrzeby danego testu. Dzięki temu test jednostkowy sprawdza poprawność działania pojedynczego fragmentu logiki aplikacji. Wykorzystanie tego typu narzędzi znacząco ułatwia i przyspiesza proces tworzenia testów.

W niniejszym artykule przedstawiono porównanie szkieletów programistycznych do izolacji kodu w testach jednostkowych. Ma on na celu zbadanie najpopularniejszych bibliotek dla platformy programistycznej .NET pod względem wydajności.

2. Testy jednostkowe

Wyróżnia się cztery typy testów automatycznych: akceptacyjne, systemowe, integracyjne oraz jednostkowe. Głównym podejściem do ich tworzenia jest tzw. piramida testów, która została opisana przez H. Vocke [1]. Liczba testów jednostkowych powinna być największa, ponieważ ich wytworzenie kosztuje najmniej oraz działają najszybciej. Ich zaletą jest również fakt, że tworzone są równolegle z testowaną metodą. Uruchomienie testu bezpośrednio po zaimplementowaniu metody pozwala na wykrycie błędów na wczesnym etapie, a co za tym idzie, obniżenie kosztów ich usunięcia. Opisane podejście przedstawione jest na Rysunku 1.



Rysunek 1: Piramida testów [1].

Testy jednostkowe są funkcjonalną metodą testowania. Polegają na sprawdzaniu pojedynczych metod w celu potwierdzenia, że działają zgodnie z przeznaczeniem. Są one wytwarzane przez programistów w celu poprawy jakości kodu oraz zniwelowania potencjalnych błędów, które mogą przydarzyć się podczas tworzenia oprogramowania. J. V. Petersen w swoim artykule przedstawia 10 powodów, dlaczego testy jednostkowe są ważne [2]. Są to między innymi:

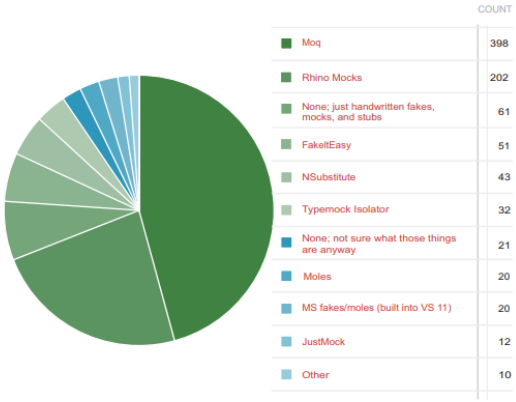
- sprawdzenie poprawności oprogramowania,
- zmniejszenie złożoności kodu,
- traktowanie testów jednostkowych jako formę dokumentacji,
- pomiar wymaganego wysiłku niezbędnego do zmiany metody,
- wymuszenie stosowania wzorców projektowych, wstrzykiwanie zależności pomiędzy nimi,
- sprawdzenie pokrycia kodu,
- sprawdzenie wydajności,
- przyspieszenie procesu wytwarzania oprogramowania poprzez możliwość wykorzystywania testów w przypadku ciągłej integracji (CI).

Współczesne metodyki pracy kładą duży nacisk na odpowiednie pokrycie aplikacji testami, dlatego zazwyczaj projekt zawiera bardzo dużą liczbę testów jednostkowych. Są one często uruchamiane, zarówno w lokalnym środowisku jak i w procesie ciągłej integracji, za każdym razem, gdy w repozytorium kodu zostanie wprowadzona zmiana. Właśnie z powodu dużej liczby testów jednostkowych oraz potrzeby częstego ich uruchamiania istnieje konieczność rozważenia kwestii czasu uruchomienia owych testów. Zbyt długi czas potrzebny na ich wykonanie może mieć negatywny wpływ na proces wytwarzania oprogramowania. Z tego względu ważny jest dobór bibliotek, które nie tylko będą oferowały szeroki wachlarz możliwości ale również zapewnią efektywność testów.

3. Analiza literatury

W niniejszym artykule dokonano przeglądu literatury dotyczącej testów jednostkowych oraz bibliotek do tworzenia atrap obiektów. W pracy J. Copliene próbuje udowodnić, że tworzenie testów jednostkowych jest niewydajne [3]. Autor stara się pokazać, że są one kwestią przeszkadzającą w jego pracy. Po analizie okazuje się jednak, że to nie same testy stwarzają problem, lecz sposób podejścia do nich, narzucany przez klientów. M. Fowler w swoim artykule skupia się zaś na charakterystyce obiektów typu mock i stub [4]. Ponadto pokazuje czytelnikowi różnice pomiędzy danymi typami, jednak nie wskazuje konkretnie, który rodzaj atrapy jest lepszy w użytkowaniu.

Istnieje wiele bibliotek do izolacji kodu w testach jednostkowych, jednak dostępność publikacji, porównujących konkretne szkielety programistyczne w ramach jednego języka programowania jest niewielka. R. Osherove jest jednym z nielicznych autorów poruszających ten temat [5]. W swojej książce przedstawił zestawienie bibliotek FakeItEasy oraz NSubstitute. Drugie wydanie książki zostało rozszerzone o bibliotekę Moq. Warto wspomnieć, że autor przeprowadził ankietę badającą popularność bibliotek. Jej wyniki zaprezentowano na Rysunku 2. Kolumna COUNT określa liczbę osób, które używały danego szkieletu programistycznego.



Rysunek 2: Wykres popularności bibliotek [5].

T. Haukilehto jest z kolei autorem porównania szkieletów Moq i FakeItEasy [6]. Twórca sugeruje, aby nie skupiać się na wyborze jednej, konkretnej biblioteki do izolacji testów jednostkowych, lecz używać ich obu. We wpisie autorstwa T. Ardal znajduje się zaś opis przedstawiający wady i zalety bibliotek badanych w ramach niniejszego artykułu [7]. W artykule T. Hyttinen można zaś poznać sposób wykorzystania szkieletu programistycznego BenchmarkDotNet, z pomocą którego tworzy się testy wydajnościowe dla rozwiązań opartych na platformie .NET [8].

4. Wybór rozwiązań do testów

Na rynku dostępnych jest wiele bibliotek do izolacji kodu w testach jednostkowych, są to między innymi:

- Moq – umożliwia definiowanie atrap przy użyciu dwóch podejść: imperatywnego lub funkcyjnego. Pozwala także na imitowanie zarówno interfejsów jak i klas, jednak rekomendowane jest tworzenie atrap dla interfejsów. Bardzo popularna, ogólna liczba jej pobrań wynosi ponad 230 miliony.
- FakeItEasy – charakteryzuje się nietypową semantyką, gdyż posiada tylko obiekty zwane fake. To w jaki sposób są używane determinuje jaka atrapa obiektu została użyta. Dokumentacja tej biblioteki jest dokładna i przejrzysta. Na stronie głównej dostępna jest opcja czatu, z którego można skorzystać w razie problemów.
- NSubstitute – wyróżnia ją składnia, która jest najłatwiejsza do nauki, ponieważ w założeniu ma przypominać naturalny język. Użycie wyrażeń lambda zostało zmniejszone do minimum. Warta uwagi jest również dokumentacja, która jest bardzo dobrze napisana.
- Rhino Mocks – jest to biblioteka programistyczna wydana na zasadach licencji BSD (Berkeley Software Distribution). Nie pozwala ona na tworzenie imitacji statycznych metod. Na rynku znajduje się już od 2011 roku, a od 9 lat nie wydano nowej wersji tego szkieletu programistycznego, co sugeruje zaprzestanie prac nad nią.

- JustMock Lite, JustMock – wersja Lite jest darmowa i zawiera przejrzystą dokumentację, jednak nie posiada wielu ważnych funkcjonalności, które są zawarte w Moq, FakeItEasy i NSubstitute. Wynika to z tego, że nie jest ona pełną wersją oprogramowania. Aby uzyskać dostęp do pozostałych funkcji, należy wykupić komercyjną wersję, o nazwie JustMock, która jest jedyną płatną biblioteką w tym zestawieniu.
- AutoFixture – spośród innych bibliotek wyróżnia ją fakt, że generowanie domyślnych wartości imitowanych obiektów może zostać zautomatyzowane. Ułatwia to pisanie testów, aczkolwiek takie działanie jest ryzykowne, ponieważ każde uruchomienie testu generuje różne wyniki. Mogą one negatywnie wpływać na końcowe sprawdzenie oczekiwanych rezultatów. Istnieje jednak sposób, dzięki któremu można otrzymywać te same instancje obiektów cały czas. Należy do tego wykorzystać typ Frozen.

Powyżej opisano najciekawsze biblioteki do tworzenia atrap obiektów w testach jednostkowych dla technologii .NET. Jednak na potrzeby niniejszej pracy zdecydowano się na wybranie trzech szkieletów programistycznych. Biblioteka Rhino Mocks nie zostanie wzięta pod uwagę, ponieważ nie jest już wspierana. Również narzędzie JustMock zostało odrzucone z powodu, że jest to rozwiązanie komercyjne i wymaga wykupienia licencji. Wersja Lite jest natomiast mocno ograniczona. Narzędzie AutoFixture ma trochę inne przeznaczenie w porównaniu do pozostałych bibliotek, ponieważ umożliwia generowanie domyślnych wartości, co nie jest do końca dobrą praktyką. Do badań zostały wytypowane biblioteki Moq, FakeItEasy oraz NSubstitute, ponieważ są to rozwiązania najpopularniejsze, darmowe, oferują podobne możliwości oraz przejrzystą dokumentację.

5. Plan badań

Celem niniejszej pracy jest porównanie wydajności wybranych szkieletów programistycznych do izolacji kodu. Z tego powodu przeprowadzone zostały badania na podstawie dwóch scenariuszy:

- pomiar czasów wykonania testów wydajnościowych (benchmarki) dla wybranych metod badanych bibliotek,
- pomiar czasów wykonania testów jednostkowych w których wykorzystane zostały wybrane metody badanych szkieletów programistycznych.

Na potrzeby badań wybrane zostały podstawowe funkcjonalności, które oferowane są przez wszystkie analizowane biblioteki do izolacji kodu i dla których zostały zaimplementowane powyższe scenariusze badawcze. Są to:

- wywołanie zwrotne,
- zwrócenie zadanej wartości,
- weryfikacja jednokrotnego wywołania,
- wywołanie metody bez parametrów,
- wywołanie metody z jednym parametrem,
- wywołanie metody z dwoma parametrami.

W celu uzyskania wiarygodnych wyników, zarówno w przypadku testów wydajnościowych, jak i testów jednostkowych, zadbano o powtarzalność eksperymentu. Każdy z przypadków został zaprojektowany tak, by realizował tę samą operację i uzyskiwał ten sam rezultat przy każdym uruchomieniu. Dla obu scenariuszy wyznaczono liczbę pomiarów, która umożliwi porównanie bibliotek. Następnie każdy z przypadków wykonano określoną liczbę razy. Spośród uzyskanych wyników odrzucono odstające, a pozostałe wzięto pod uwagę podczas oceniania. W ramach analizy wyznaczone zostały wielkości statystyczne, takie jak średni czas wykonania, odchylenie standardowe, mediana, minimum, maksimum oraz pierwszy i trzeci kwartyl.

5.1. Testy wydajnościowe

Pierwszy scenariusz badawczy to pomiar czasów wykonania testów wydajnościowych dla wybranych metod badanych bibliotek. Dla każdej z rozpatrywanych funkcjonalności zostały utworzone trzy benchmarki, po jednym dla każdego szkieletu programistycznego. Wywołują one odpowiednie metody, które realizują daną funkcjonalność. Następnie metody te zostały przekształcone w testy wydajnościowe przy użyciu biblioteki BenchmarkDotNet. Umożliwia ona uruchomienie poszczególnych metod określoną liczbę razy oraz pomiar ich czasów wykonania. W tym przypadku zastosowano następującą konfiguracje: 3 uruchomienia, 10 powtórzeń, 100000 iteracji dla każdego powtórzenia. Dzięki czemu wykonano 3000000 pomiarów dla każdej z badanych funkcjonalności. Scenariusz ten pozwoli na porównanie wydajności badanych szkieletów w warunkach izolowanych.

5.2. Testy jednostkowe

Drugi scenariusz badawczy obejmuje stworzenie prostych testów jednostkowych, które wykorzystują wybrane funkcjonalności badanych szkieletów programistycznych. Testy zostały przygotowane z wykorzystaniem biblioteki NUnit i były uruchamiane przy użyciu środowiska programistycznego Visual Studio z zainstalowanym pakietem NUnit3TestAdapter. Daje to możliwość rejestrowania czasu wykonania w łatwy sposób. Przy realizacji tego scenariusza istotne było uruchamianie testów pojedynczo w celu uzyskania niezależnych wyników. Wykonanie pomiarów nie było zautomatyzowane, każdy test był uruchamiany manualnie. Wykonano po 50 pomiarów dla każdej z badanych funkcjonalności. Rezultaty uzyskane przy użyciu tego scenariusza pozwolą porównać wpływ badanych bibliotek na czas wykonania testów.

5.3. Środowisko testowe

W celu uzyskania powtarzalności wyników wszystkie pomiary wykonano na tym samym środowisku. Specyfikacje stanowiska badawczego przedstawiono w Tabeli 1. Przed rozpoczęciem eksperymentu, ponownie zainstalowany został system operacyjny oraz niezbędne oprogramowanie zaprezentowane w Tabeli 2. Podczas wykonywania badań wyłączone pozostawały wszelkie

inwazyjne aplikacje oraz procesy, na przykład program antywirusowy. Stanowisko było także odłączone od sieci internetowej.

Tabela 1: Specyfikacja stanowiska badawczego

Element	Specyfikacja
System operacyjny	Windows 10 Education
Procesor	Intel Core i5-6300HQ 2.30GHz (4679 punktów według strony www.cpubenchmark.net)
Pamięć RAM	16 GB
Dysk	Plextor PX-256M7VG

Tabela 2: Wykorzystane narzędzia i biblioteki

Narzędzie	Wersja
Visual Studio	2019 Community
.NET	5.0
Język C#	9.0
NUnit	3.13.2
NUnit3TestAdapter	4.0.0
BenchmarkDotNet	0.13.1
FakeItEasy	7.2.0
Moq	4.16.1
NSubstitute	4.2.2

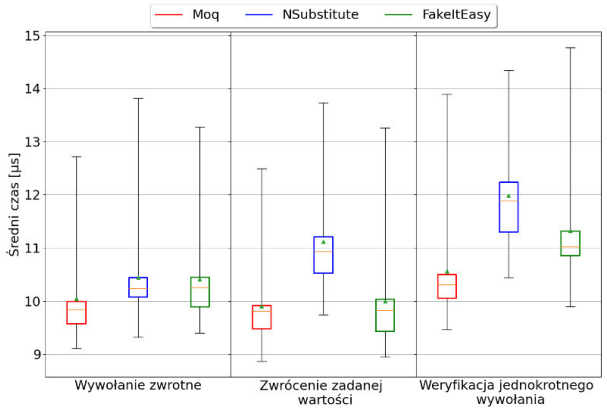
6. Wyniki badań

Na podstawie rezultatów uzyskanych zarówno dla testów wydajnościowych jak i dla testów jednostkowych, wyznaczono podstawowe wielkości statystyczne. Wyniki zostaną przedstawione na wykresach, co ułatwi ich porównanie.

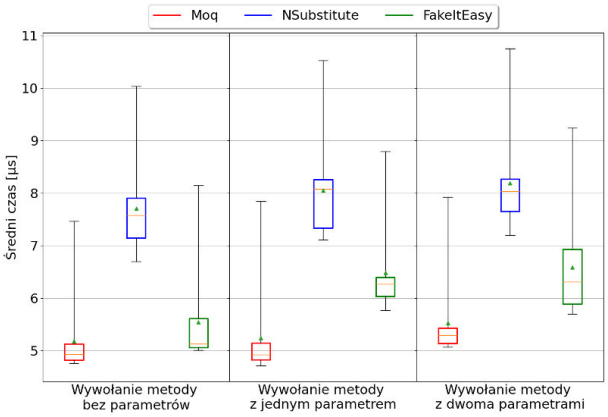
6.1. Testy wydajnościowe

Na Rysunku 3 oraz Rysunku 4 przedstawiono wykresy pudełkowe obrazujące wyniki otrzymane dla testów wydajnościowych. Rysunek 3 zawiera wykres przedstawiający rozkład wartości dla trzech funkcjonalności: wywołania zwrotnego, zwrócenia zadanej wartości oraz weryfikacji jednokrotnego wywołania. We wszystkich przypadkach najmniejszy czas osiągnęła biblioteka Moq, a nieco gorsze wyniki uzyskał szkielet programistyczny FakeItEasy. Warto jednak zaznaczyć, szczególnie biorąc pod uwagę fakt, że przedstawione wartości są rzędu mikro sekund, że różnice pomiędzy bibliotekami są bardzo niewielkie.

Na Rysunku 4 zaprezentowano wykres pudełkowy przedstawiający wyniki dla trzech kolejnych funkcjonalności. Jest to wywołanie metody bez parametrów, wywołanie metody z jednym parametrem oraz wywołanie metody z dwoma parametrami. Również w tym przypadku Moq osiąga najkorzystniejsze wyniki. Bardzo małe pudełko oraz mała odległość od wartości minimalnej, dla tego szkieletu programistycznego, świadczy o tym, że większość (co najmniej 75%) pomiarów ma bardzo zbliżone wartości. Można także zauważyć, że im większa jest liczba przekazanych parametrów, tym większy jest średni czas wykonania metod atrapy obiektu dla wszystkich badanych bibliotek.



Rysunek 3: Wykres pudełkowy – testy wydajnościowe – wywołanie zwrotne, zwrócenie zadanej wartości oraz weryfikacja jednokrotnego wywołania.



Rysunek 4: Wykres pudełkowy – testy wydajnościowe – wywołanie metody bez parametrów, wywołanie metody z jednym parametrem oraz wywołanie metody z dwoma parametrami.

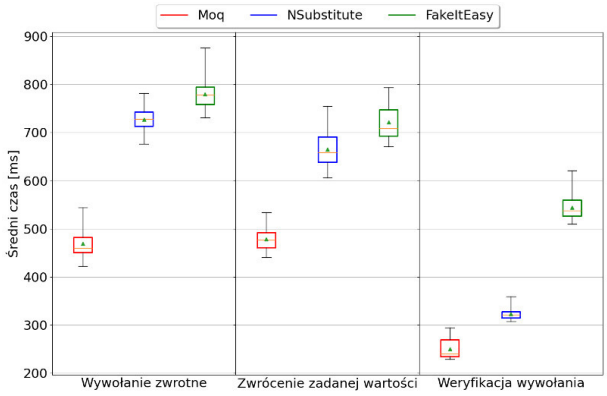
W przypadku wykresów przedstawionych na Rysunku 3 oraz Rysunku 4 wyraźnie widać, że rozkład wartości jest asymetryczny prawostronnie. Oznacza to, że odległość wartości maksymalnej od mediany jest znacząco większa niż odległość wartości minimalnej od mediany. Świadczy to o sporym rozproszeniu wartości większych od mediany.

6.2. Testy jednostkowe

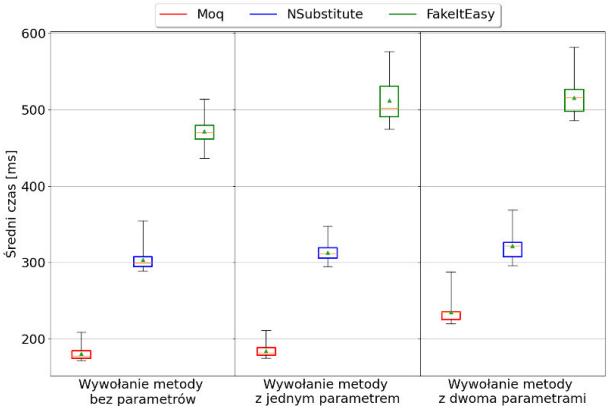
Drugi scenariusz badawczy to sprawdzenie wpływu wyboru biblioteki do izolacji kodu na czas wykonania testów jednostkowych. Na Rysunku 5 oraz Rysunku 6 zaprezentowano wykresy pudełkowe przedstawiające wyniki czasów uruchomienia owych testów.

Podobnie jak w przypadku benchmarków, Moq osiąga najlepsze wyniki. Różnicą jest jednak to, że w tym scenariuszu, szkielet ten ma większą przewagę nad pozostałymi narzędziami. Biblioteką z najgorszymi wynikami nie jest, jak dla testów wydajnościowych, NSubstitute. Największe czasy wykonania uzyskał szkielet programistyczny FakeItEasy. Dla czterech funkcjonalności jest to czas ponad dwa razy dłuższy niż czas uzyskany przez bibliotekę Moq. Jest to zaskakujący wynik, biorąc pod uwagę fakt, że narzędzie FakeItEasy osiągało niedużo gorsze wyniki niż Moq w testach wydajnościowych. Może to oznaczać słabe zarządzanie

cyklem życia obiektów przez bibliotekę FakeItEasy, ponieważ w przypadku testów jednostkowych atrapa jest wykorzystywana wielokrotnie w obrębie jednego testu.



Rysunek 5: Wykres pudełkowy – testy jednostkowe – wywołanie zwrotne, zwrócenie zadanej wartości oraz weryfikacja wywołania.



Rysunek 6: Wykres pudełkowy – testy jednostkowe – wywołanie metody bez parametrów, wywołanie metody z jednym parametrem oraz wywołaniem metody z dwoma parametrami.

7. Wnioski

Celem niniejszej pracy było porównanie wydajności szkieletów programistycznych do izolacji kodu w testach jednostkowych, takich jak: FakeItEasy, Moq i NSubstitute. Są to najpopularniejsze tego typu narzędzia dla platformy programistycznej .NET.

Przeprowadzone badania obejmują benchmarki oraz testy jednostkowe wykorzystujące analizowane biblioteki. Wskazały one, że narzędzie Moq osiąga najlepsze wyniki dla obu scenariuszy badawczych. W przypadku pozostałych szkieletów programistycznych wyniki nie są jednoznaczne. W testach wydajnościowych drugie miejsce zajęła biblioteka FakeItEasy, jednak w testach jednostkowych drugie najmniejsze czasy wykonania uzyskała biblioteka NSubstitute. Pomiar czasów uruchomienia testów jednostkowych pokazuje wpływ szkieletów programistycznych na wydajność testów. Są one głównym przeznaczeniem badanych bibliotek, dlatego lepszy rezultat NSubstitute dla tego scenariusza wskazuje, że jest to bardziej efektywne narzędzie niż FakeItEasy.

Na podstawie przeprowadzonej analizy można stwierdzić, że Moq to najlepszy szkielet programistyczny do izolacji kodu w .NET pod względem wydajności. Mimo to, pozostałe dwie biblioteki również są godnymi uwagi możliwościami. Przedstawione porównanie można by rozszerzyć o inne, mniej popularne oraz płatne rozwiązania.

Literatura

[1] H. Vocke, The Practical Test Pyramid, martinfowler.com, 2018.

[2] J. Petersen, 10 Reasons Why Unit Testing Matters, CODE Magazine, 2019 January/February.

[3] J. Coplien, Why Most Unit Testing is Waste, RBCS-US.com, 2015.

[4] Porównanie obiektu typu mock a stub, https://martinfowler.com/articles/mocksArentStubs.html, [07.03.2022]

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

[6] T. Haukilehto, Isolated unit tests in. Net, Seinäjoki University of Applied Sciences, 2013.

[7] Porównanie testów jednostkowych z wykorzystaniem bibliotek Moq, NSubstitute i FakeItEasy, https://blog.elmah.io/moq-vs-nsubstitute-vs-fakeiteasy-which-one-to-choose/, [07.03.2022]

[8] T. Hyttinen, .NET Core 3.1 & .NET 5, Performance benchmarking in Web API use, JAMK University of Applied Sciences, May 2021.