JOURNAL
COMPUTER SCIENCES INSTITUTE

JCSI 17 (2020) 339–344
Received: 29 July 2020
Accepted: 24 September 2020

Analysis and comparison of programming frameworks used for automated tests

Analiza i porównanie szkieletów programistycznych używanych do testów automatycznych

Damian Gromek*, Dariusz Gutek

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

Abstract
This article compares and discusses the programming skeletons for automatic tests. Two test skeletons have been selected for testing purposes and then a test environment has been installed and configured, in which appropriate test scenarios have been prepared. Once the test environment has been properly prepared, measurements of the time to launch both frameworks were performed. The results obtained were shown in the form of tables and charts for later analysis. The frameworks were also analyzed in terms of syntax and functionality. At the end, a summary was presented, containing more information and conclusions that resulted from the analysis.

Keywords: Testing; JUnit; TestNG; Cucumber

Streszczenie
W niniejszym artykule zostały porównane i omówione szkielety programistyczne służące do testów automatycznych. Do celów badawczych zostały wybrane dwa szkielety a następnie zostało zainstalowane i skonfigurowane środowisko badawcze, w którym przygotowano odpowiednie scenariusze testowe. Po odpowienim przygotwaniu środowiska badawczego zostały przeprowadzone pomiary czasu uruchomienia obydwu frameworków. Wyniki jakie otrzymano pokazano w postaci tabel i wykresów umożliwiających ich późniejszą analizę. Badane frameworki zostały również przeanlizowane pod kątem składni i funkcjonalności. Na koniec zostało przedstawione podsumowanie zawierające najistotniejsze informację i wnioski jakie wynikły z przeprowadzonej analizy.

Słowa kluczowe: Testowanie; JUnit; TestNG; Cucumber

*Corresponding author
Email address: **damian.gromek@pollub.edu.pl** (D. Gromek)

1. Wstęp

W dzisiejszych czasach zaobserwować można bardzo dynamiczny rozwój nowych technologii a także bardzo szybkie tempo życia co implikuję potrzebę kreowania bardziej wydajnych i bardziej niezawodnych systemów informatycznych mogących sprostać coraz to nowszym wymaganiom. Klienci oczekują niezawodności tworzonego dla nich oprogramowania, gdyż często w przypadku awarii systemów informatycznych może to ich doprowadzić do ogromnych strat finansowych wynikających z zaprzestania prowadzenia przez nich działalności a nawet do strat wizerunkowych wynikających z błędnie działającego oprogramowania.

Wzrost złożoności tworzonych systemów informatycznych a także zmiany w samych metodach zarządzania projektem zwiększają ryzyko wystąpienia potencjalnych błędów. Aby nabrać zaufania do tworzonego oprogramowania niezbędny jest proces testowy, którego celem jest zapewnienie jakości rozwijanemu oprogramowaniu.

Coraz to nowsze wymagania stawiane deweloperom wymuszają pośrednio wymyślanie i tworzenie bardziej skutecznych technik i narzędzi służących do testowania oprogramowania. Jednym z działań usprawniających proces testowy jest jego automatyzacja. Mnogość narzędzi i frameworków służących do automatyzacji testowania często sprawia istotny problem z doborem odpowiednich narzędzi na początku prac nad zautomatyzowaniem testów w projektach komercyjnych. Dokonanie błędnego lub nieoptymalnego wyboru narzędzi często powoduję późniejszą frustracje wynikającą z pracy z mało wydajnymi narzędziami jak i straty finansowe na poczet nowych licencji w przypadku wymiany narzędzi na inne w późniejszej fazie projektu.

W niniejszym artykule zostały omówione i porównane ze sobą dwa szkielety programistyczne służące do automatyzacji testów jednostkowych: JUnit oraz TestNG. Analiza porównawcza tych dwóch frameworków jest niezwykle istotna z racji tego, że oba te szkielety programistyczne ukierunkowane są na wspieranie testów jednostkowych dla aplikacji pisanych w języku Java. Język Java jest bowiem bardzo popularnym i wszechstronnym językiem programowania co przekłada się na to, że obecnie wiele firm pracuję w tej technologii szukając przy tym wydajnych i funkcjonalnych narzędzi wspierających proces wytwarzania oprogramowania, którego istotną częścią jest faza testów. Analiza tych dwóch szkieletów programistycznych miała na celu wskazanie bardziej wydajnej technologii i została oparta na porównaniu ich szybkości a także zakresu funkcjonalności.

2. Wprowadzenie do testowania oprogramowania

Istnieje wiele różnych podejść i metod testowania oprogramowania. W tym rozdziale zostało opisanych kilka z nich [1].

- Testy modułowe polegają na testowaniu najmniejszych komponentów możliwych do wyizolowania – mogą to być moduły, klasy, metody. Testy modułowe możemy podzielić na testy jednostkowe, testy pokrycia instrukcji, testy klas równoważności bądź testy wartości brzegowych.
- Testy integracyjne przeprowadzane są po testach modułowych i polegają na sprawdzeniu interakcji pomiędzy poszczególnymi modułami.
- Testy systemowe mają za zadanie przetestować zachowanie oprogramowania w ujęciu całościowym – testy na tym poziomie wykonywane są w momencie, gdy poszczególne komponenty i funkcje są ze sobą zintegrowane i tworzą spójny system.
- Testy akceptacyjne są najczęściej wykonywane przez użytkowników a celem tych testów nie jest już wykrywanie błędów a jedynie formalne potwierdzenie jakości i spójności gotowego już oprogramowania.
- Testy alfa mogą być rzeczywiste bądź symulowane i są zwykle przeprowadzane przez potencjalnych przyszłych użytkowników danego oprogramowania bądź przez niezależny zespół testerów. Przeprowadzane są najczęściej na miejscu w siedzibie producenta.
- Testy beta przeprowadzane są już na rzeczywistym środowisku produkcyjnym przez potencjalnych bądź też realnych użytkowników danego oprogramowania. Testy te nie odbywają się już w siedzibie producenta a poza nim. Celem tych testów jest ocena przez użytkowników czy system spełnia ich potrzeby i wymagania.
- Testy regresji są często mylone z retestami. Jest to duży błąd, gdyż te terminy nie są ze sobą jednoznaczne. Retest jest to sprawdzenie czy pojedynczy defekt został naprawiony przez dewelopera i czy już więcej nie występuje, natomiast testy regresji polegają na sprawdzeniu większego obszaru aplikacji w celu weryfikacji czy np. wprowadzone zmiany w implementacji wynikające z dodania nowej funkcjonalności bądź przy okazji poprawy defektu nie spowodowały błędów w innych częściach oprogramowania nawet pozornie nie związanego z wprowadzonymi zmianami.
- Testy pielęgnacyjne dokonywane są już na wdrożonym i użytkowanym systemie informatycznym. Testy te zwykle dokonywane są po wdrożeniu nowej wersji oprogramowania.

3. Automatyzacja testowania

Proces automatyzacji testów wnosi wiele wartości w obszarze zapewnienia jakości w projektach komercyjnych. Automatyzacja testów pozwala zaoszczędzić wiele czasu, który byłby niezbędny na wykonanie tych testów w sposób manualny. Mimo wielu zalet automatyzacji posiada on też pewne ograniczenia przez, które zwykle nie jest możliwe zautomatyzowanie wszystkich testów w projekcie. Niniejszy rozdział ma na celu omówić najważniejsze plusy automatyzacji testów jak i jej ograniczenia [2].

3.1. Korzyści ze stosowania zautomatyzowanych testów

Głównym celem automatyzacji jest zautomatyzowanie najczęściej powtarzanych procesów wykonywanych do tej pory w sposób manualny. W wyniku tego procesu następuje szereg korzyści, które zostały omówione poniżej [3].

- Wzrost produktywności. Dobrze zaimplementowany zestaw testów automatycznych powinien wykonywać się bez jakichkolwiek ingerencji ze strony testera – nie powinien on w trakcie działania wymagać od testera podejmowania żadnych dodatkowych działań ani pytać go np. o napotkany wyjątek. Obsługa takich przypadków powinna być z góry zaimplementowana w kodzie testów automatycznych. Dobrze napisany i skonfigurowany test automatyczny powinien przechodzić przez wszystkie kroki bez zatrzymywania się, nawet gdy po drodze napotka defekt – chyba, że defekt ten uniemożliwia dalsze wykonywanie testu. W związku z tym test automatyczny może być wykonywany poza godzinami pracy testera. W ciągu dnia tester automatyzujący może pracować nad kodem testu automatycznego a tuż przed wyjściem z pracy uruchomić go – wtedy następnego dnia wygenerowany raport z przeprowadzonych testów może zostać przekazany do zespołu deweloperskiego odpowiedzialnego za naprawę znalezionych defektów. Jako, że gotowe testy automatyczne mogą być przeprowadzane po opuszczeniu przez testera stanowiska pracy oznacza to zaoszczędzenie czasu potrzebnego na manualne wykonanie tychże testów na stanowisku pracy w firmie.
- Wzrost niezawodności. Dobry test powinien być łatwy do powtórzenia zarówno przez testera jak i przez dewelopera odpowiedzialnego za naprawę potencjalnego defektu znalezionego w czasie takiego testu. Podczas wykonywania testu manualnego możliwe jest pomimo wykonywania jednego i tego samego scenariusza przeprowadzenie go nieco inaczej np. używając za każdym razem innych danych testowych, bądź w przypadku manualnych testów frontendu możliwe jest kliknięcie nieco innego obszaru aplikacji co może skutkować zupełnie innym zachowaniem testowanej aplikacji. Test automatyczny z kolei uruchamiany jest zawsze w ten sam, przewidywalny sposób, który został zaprogramowany przez testera automatyzującego co oznacza, że w przypadku testów frontendu, obszar klikany na frontendzie aplikacji jest zawsze ten sam. Dzięki temu ewentualne rozbieżności przy ponownych uruchomieniach testów automatycznych są wykluczane. Wykonując po raz n-ty ten sam scenariusz testowy, można przestać zwracać uwagę na szczegóły i nie

zauważać np. literówek na stronie internetowej albo nawet i poważniejszych usterek. W przypadku testów automatycznych za wykonanie tych testów jest odpowiedzialny automat, którego zadaniem jest przetestowanie aplikacji dokładnie w taki sposób w jaki został on zaprogramowany przez testera. Takie zachowanie gwarantuje brak przypadkowego pominięcia sprawdzenia jakiejś funkcjonalności bądź jakiegoś obszaru aplikacji. Każdy niewykonany test np. poprzez złą implementacje jest wyraźnie komunikowany poprzez wyrzucenie kodu błędu wraz z informacją co poszło nie tak, zatem nie istnieje możliwość przypadkowego pominięcia testów w przypadku testowania automatycznego.

- Wzrost pokrycia testami. Testy automatyczne wykonywane są zdecydowanie szybciej niż testy manualne a dodatkowo dzięki nim możliwe jest symulowanie obciążeń bądź nawet przeciążeń danej aplikacji poprzez np. uruchomienie testu na kilku przeglądarkach naraz. Test manualny przeprowadzony może być tylko jeden w jednym czasie. Nie ma możliwości manualnego kliknięcia na dwa elementu naraz w dodatku na dwóch różnych przeglądarkach. Testowanie manualne nie jest zatem w stanie objąć testami pewnych specyficznych przypadków bądź wymagań, które z kolei mogą zostać bez problemu przetestowane dzięki użyciu testów automatycznych.

3.2. Ograniczenia testowania automatycznego

Opracowanie testów automatycznych niesie ze sobą liczne korzyści zarówno dla zespołu testerów – ograniczanie wykonywania monotonnych czynności wiele razy, jak i dla całego projektu – szybsze testowanie to szybsze informację o jakości oprogramowania. Zwykle jednak testy automatyczne nie mogą w pełni zastąpić testów manualnych w projekcie. Wynika to głównie z faktu, że część potencjalnych problemów może zostać znaleziona jedynie poprzez wykonanie manualnych testów przez człowieka.

Często spotykanym zabezpieczeniem na stronach internetowych jest tzw. CAPTCHA. Zabezpieczenie to ma na celu dopuszczenie przesyłania danych bądź formularzy tylko przez człowieka. W momencie, gdy test automatyczny natrafi na to zabezpieczenie to prawdopodobnie wynik tego testu będzie negatywny, co wcale nie musi być prawdą, gdyż zabezpieczenie typu CAPTCHA jest często używanym i świadomym zabezpieczeniem więc samo jej występowanie nie świadczy o błędzie w implementacji. W tym przypadku zatem, wynik z przeprowadzonego testu może wprowadzać testera w błąd i generować nieprawdziwy raport o poziomie jakości danego oprogramowania.

Żeby testy automatyczne zaczęły wnosić wartość w projekcie niezbędne jest spędzenie znacznej ilości czasu nad ich zaplanowaniem i przygotowaniem. W związku z tym w bardzo małych i krótkich projektach testy automatyczne mogą nie być najlepszym pomysłem ze względu na ograniczone zasoby czasowe. Testy manualne na poziomie frontendu aplikacji zwykle w takich przypadkach sprawdzają się lepiej, gdyż mogą one być wykonywane w każdym momencie po uprzednim przygotowaniu scenariuszy testowych.

Opis porównywanych szkieletów programistycznych

W niniejszym rozdziale zostały zaprezentowane i omówione badane szkielety programistyczne. Zostały również zestawione ze sobą poszczególne adnotacje jakie są używane w każdym z omawianych frameworków służące do wywoływania określonych funkcjonalności.

TestNG

Framework TestNG został opracowany przez Cedrica Beusta w celu automatyzacji testów dla języka Java. Szkielet ten został stworzony w czasie, gdy na rynku dostępny był framework JUnit w wersji 3 po to aby wyeliminować jego liczne wówczas ograniczenia. Opracowanie frameworka TestNG nie tylko wyeliminowało większość ograniczeń ówczesnego frameworka JUnit ale także dodało wiele nowych funkcjonalności obejmujących nie tylko testy jednostkowe ale także testy funkcjonalne a nawet testy end-to-end. Została wprowadzona obsługa adnotacji, która była niedostępna we frameworku JUnit w wersji 3 [4].

3.3. JUnit

Kent Beck, Erich Gamma, David Saff oraz Kris Vasudevan są autorami frameworka JUnit, którego celem jest tworzenie testów jednostkowych dla oprogramowania napisanego w języku Java. Najnowsza wersja frameworka – JUnit 5 – wprowadziła szereg zmian oraz aktualizacji wcześniej wykorzystywanych funkcjonalności. Zmiany te dotyczyły m.in. zastąpienia dotychczas używanych nazw poszczególnych adnotacji na bardziej samo opisujące się a przez to bardziej zrozumiałe dla użytkowników [5, 6]. Dokonano zmian w nazwach adnotacji odpowiedzialnych za uruchamianie konkretnego testu bądź grup testów. Dodatkowo w JUnit w wersji 5 zostały wprowadzone dodatkowe asercje odpowiadające za testowanie i sprawdzanie czy czas uruchomienia określonego fragmentu kodu został wykonany w czasie szybszym niż z góry zadeklarowany limit [7].

3.4. Porównanie składni frameworka TestNG i JUnit

Pomimo tego, że zarówno framework TestNG jak i framework JUnit służą do przeprowadzania testów jednostkowych w języku Java oraz że posiadają wiele cech wspólnych istnieją pomiędzy nimi pewne różnice zarówno w składni jak i w zakresie funkcjonalności. W Tabeli 1 zamieszczonej poniżej przedstawione zostało porównanie adnotacji używanych w każdym z omawianych frameworków [8].

Na podstawie zestawień przedstawionych w Tabeli 1 wynika, że framework TestNG posiada więcej funkcji możliwych do zastosowania podczas implementowania testów automatycznych.

Tabela 1: Porównanie składni i funkcji frameworka TestNG i JUnit

Opis adnotacji	Adnotacja	
	TestNG	JUnit 5
Uruchomienie testu	@Test	@Test
Uruchomienie kodu przed testem	@BeforeClass	@BeforeAll
Uruchomienie kodu po wszystkich metodach	@AfterClass	@AfterAll
Uruchomienie kodu przed każdą metodą testującą	@BeforeMethod	@BeforeEach
Uruchomienie kodu po każdej metodzie testującej	@AfterMethod	@AfterEach
Ignorowanie określonego testu	@Test (enabled=false)	@ignore
Określenie spodziewanego wyjątku	@Test(expectedException = Arithmetic Exception.class)	@Test(expected = Arithmetic Exception.class)
Określenie maksymalnego czasu na wykonanie testu	@Test (timeOut = 1000)	@Test (timeout = 1000)
Uruchomienie kodu przed wszystkimi testami w scenariuszu	@BeforeSuite	Brak
Uruchomienie kodu po wszystkich testach w scenariuszu	@AfterSuite	Brak
Uruchomienie kodu przed konkretnym testem	@BeforeTest	Brak
Uruchomienie kodu po konkretnym teście	@AfterTest	Brak
Uruchomienie kodu przed pierwszym testem w grupie	@BeforeGroups	Brak
Uruchomienie kodu po ostatnim teście w grupie	@AfterGroups	Brak

4. Przeprowadzenie badań pomiarowych

Przeprowadzenie badania miało na celu ustalić, który z porównywanych frameworków potrzebuje mniej czasu na wykonanie testów dla wszystkich przygotowanych scenariuszy testowych.

4.1. Opis badania

Badanie zostało przeprowadzone na laptopie z procesorem Intel Core i7-9750H z pamięcią RAM 32GB przy użyciu narzędzia IntelliJ IDEA w środowisku Windows. Użyta została przeglądarka Google Chrome w wersji 83. Posłużono się narzędziem Cucumber w celu przygotowania i uruchamiania scenariuszy testowych. Każdy scenariusz testowy został wykonany trzykrotnie dla obu badanych szkieletów programistycznych. Po dokonaniu pomiarów wyniki zostały ze sobą zestawione i przeanalizowane.

4.2. Przebieg badania

Na początku badań przygotowano dwa odrębne projekty – jeden dla testów z użyciem frameworka TestNG a drugi dla testów z użyciem frameworka JUnit. W następnym kroku przy pomocy narzędzia Cucumber zostało opracowanych siedem następujących scenariuszy testowych:

- TC 1 – Scenariusz sprawdzający logowanie się do serwisu przy użyciu poprawnego loginu i hasła.
- TC 2 – Scenariusz sprawdzający logowanie do serwisu przy użyciu poprawnego loginu i niepoprawnego hasła.
- TC 3 – Scenariusz sprawdzający wyszukiwanie produktów na stronie.
- TC 4 – Scenariusz sprawdzający formularz rejestracyjny.
- TC 5 – Scenariusz sprawdzający funkcjonalność dodawania produktu do koszyka.
- TC 6 – Scenariusz sprawdzający kolor czcionki napisu na stronie głównej serwisu.
- TC 7 – Scenariusz, który za każdym razem ma dać negatywny wynik.

Jeden z przykładowych scenariuszy testowych wykonany w narzędziu Cucumber został zamieszczony na Listingu 1 zamieszczonym poniżej.

Listing 1: Przykładowy scenariusz testowy opracowany w narzędziu Cucumber

```
@grupaScenariuszy
Feature: TestNG + Cucumber

  @test1_poprawneLogowanie
  Scenario: Użytkownik wprowadza poprawne dane logowania
    Given Użytkownik uruchamia strone serwisu
    Then Strona główna serwisu Media Expert jest wyświetlona
    And Użytkownik klika na przycisk Twoje konto
    And Użytkownik czeka na załadowanie się strony
    And użytkownik wprowadza w pole login swój login
    And użytkownik wprowadza w pole hasło swoje hasło
    And użytkownik klika na przycisk zaloguj
    And Strona powitalna jest wyświetlona
    And Test nr jeden zakonczony
```

Skrypt zamieszczony na Listingu 1 przedstawia wysokopoziomową implementację testu sprawdzającego funkcjonalność logowania się do serwisu internetowego. Po opracowaniu scenariuszy testowych w narzędziu Cucumber dla każdego kroku zostały zaimplementowane odpowiednie metody testujące. Do badań opracowano takie same scenariusze testowe dla każdego z analizowanych frameworków.

Po każdym przeprowadzonym pomiarze generowany był raport z wykonanych testów. Przykład raportu otrzymanego dla przypadku testowego sprawdzającego logowanie się do serwisu internetowego został ukazany na Rysunku 1 zamieszczonym poniżej.

Rysunek 1: Raport po wykonaniu testu

Raport z przeprowadzonego testu ukazuje dwie istotne informację. Pierwszą z nich jest status każdego poszczególnego kroku informujący o tym czy krok ten zakończył się pomyślnie czy nie. Drugą istotną informacją jest czas wykonania każdego kroku jak i łączny czas wykonania testów dla uruchomionego przypadku testowego. Czasy jakie zostały zawarte w raportach z przeprowadzonych testów wykorzystane zostały w późniejszej analizie porównawczej.

4.3. Wyniki przeprowadzonego badania

Po pomyślnym uruchomieniu i zakończeniu wszystkich testów, za każdym razem wygenerowany został raport z wynikami testów poszczególnych scenariuszy i kroków w nim zawartych a także czasy ich wykonania.
Na potrzeby badań zostały zebrane wszystkie zmierzone cząstkowe czasy a ich wyniki zostały przedstawione w postaci tabel.
Tabela 2 zamieszczona poniżej przedstawia zestawienie czasów dla frameworka TestNG.

Tabela 2: Pomiary dla frameworka TestNG

Przypadek testowy	Pomiar 1 (s)	Pomiar 2 (s)	Pomiar 3 (s)
TC 1	18,91	22,53	22,21
TC 2	21,45	21,71	21,89
TC 3	27,66	27,94	29,47
TC 4	18,74	17,21	17,73
TC 5	50,41	58,82	50,25
TC 6	9,96	12,45	13,38
TC 7	11,83	10,49	10,85
Suma	158,96	171,15	165,78

Z kolei Tabela 3 zamieszczona poniżej przedstawia zestawienie czasów wykonania testów dla frameworka JUnit.

Tabela 3: Pomiary dla frameworka JUnit

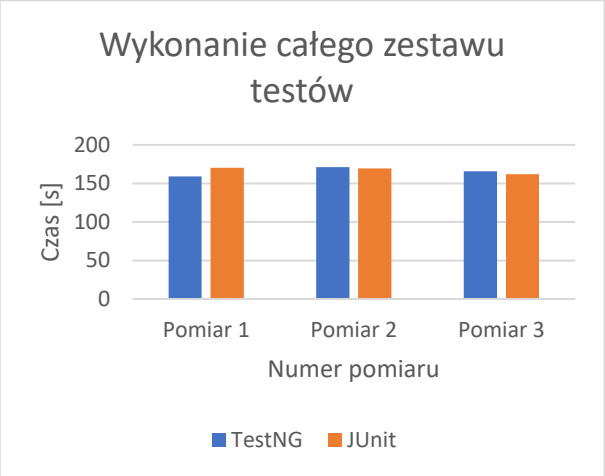
Przypadek testowy	Pomiar 1 (s)	Pomiar 2 (s)	Pomiar 3 (s)
TC 1	24,25	24,78	21,95
TC 2	24,64	20,72	20,54
TC 3	31,99	29,39	29,44
TC 4	18,56	17,57	16,98
TC 5	49,75	57,82	49,48
TC 6	9,68	8,39	12,37
TC 7	11,33	10,56	11,23
Suma	170,19	169,24	161,99

4.4. Porównanie wyników przeprowadzonego badania

Sumy wszystkich otrzymanych wyników cząstkowych dla poszczególnych pomiarów zostały przedstawione w formie wykresów kolumnowych. Każdego pomiaru dokonywano w 10-cio minutowym przedziale czasu a dodatkowo pomiędzy kolejnymi pomiarami został zachowany pewien odstęp czasowy. Ramy czasowe poszczególnych pomiarów zostały zaprezentowane poniżej:

- Pierwszego pomiaru dokonano w godzinach: 21:15 – 21:25.
- Drugiego pomiaru dokonano w godzinach 23:20 – 23:30.
- Trzeciego pomiaru dokonano w godzinach 00:40 – 00:50.

Na Rysunku 2 zamieszczonym poniżej zostały zestawione ze sobą łączne czasy wykonania wszystkich scenariuszy testowych wykonanych przy użyciu zarówno frameworka JUnit jak i frameworka TestNG.



Rysunek 2: Zestawienie wyników trzech pomiarów

Pierwszy pomiar ukazał, że czas potrzebny na uruchomienie wszystkich przypadków testowych był o 11,23s krótszy dla testów z wykorzystaniem frameworka TestNG. Drugi pomiar ukazał z kolei przewagę frameworka JUnit dla, którego czas uruchomienia wszystkich testów był o 1,91s krótszy niż dla frameworka TestNG. Podobną zależność wykazał pomiar trzeci, w którym to różnica wyniosła 3,79s na korzyść frameworka JUnit.

Przeprowadzony eksperyment miał za zadanie wykazać, który z badanych frameworków potrzebuje mniej czasu na wykonanie całego scenariusza testowego. Czasy wykonania testów dla poszczególnych scenariuszy były do siebie bardzo zbliżone i nie odbiegały znacząco od siebie. Pomiar pierwszy wykazał szybsze wykonanie testów opartych o framework TestNG, natomiast pomiary: drugi i trzeci wykazały odwrotną zależność. W czasie pomiaru drugiego i trzeciego to framework JUnit osiągnął nieznacznie lepszy wynik. Wpływ na te wartości mogły mieć różnice w obciążeniu testowanego serwisu internetowego co mogło się przełożyć na późniejsze wczytywanie się elementów strony, których pełne załadowanie było niezbędne do kontynuowania testów.

5. Wnioski

W świetle przeprowadzonej analizy czasów wykonania poszczególnych scenariuszy testowych z użyciem analizowanych szkieletów programistycznych wykazano, że czasy wykonania testów dla obu frameworków były bardzo do siebie zbliżone. Wyniki pierwszego pomiaru pokazały, że łączny czas uruchomienia wszystkich przypadków testowych był o 11,23s krótszy dla frame-

worka TestNG w porównaniu z frameworkiem JUnit, natomiast pomiary: drugi i trzeci pokazały odwrotne zależności i tak łączne czasy uruchomienia testów dla frameworka JUnit były krótsze o 1,91s w drugim pomiarze i o 3,79s w trzecim pomiarze.

Analiza funkcjonalności wykazała, że framework TestNG posiada więcej funkcji możliwych do wykorzystania przez testera bądź przez programistę. Framework TestNG w porównaniu do frameworka JUnit wspiera obsługę adnotacji wykorzystywanych do zarządzania testami w odniesieniu do całych grup jak i do całych scenariuszy testów.

Technologią częściej wybieraną przez użytkowników okazała się być technologia JUnit [9].

Pomimo trzykrotnego uruchomienia testów dla każdego z badanych frameworków nie było możliwe jednoznaczne wskazanie szybszego frameworka. Czasy wykonania testów dla oby szkieletów programistycznych były bardzo do siebie zbliżone i różnice pomiędzy nimi nie były znaczące. To pokazuje, że chcąc wybrać najlepszy framework służący do testów automatycznych spośród TestNG i JUnit czynnik czasu nie powinien być najważniejszym kryterium przy jego wyborze. Użytkownicy powinni skupić się raczej na zakresie funkcjonalności każdego z frameworków, których szerszy zakres posiada framework TestNG. Kolejnym istotnym czynnikiem dla wyboru najlepszego frameworka może też być jego popularność. W tym aspekcie to framework JUnit ma przewagę o czym świadczą liczne tematy na specjalistycznych forach programistycznych dotyczące tego frameworka poprzez co prawdopodobieństwo otrzymania pomocy od innych użytkowników jest większe w przypadku napotkania jakichkolwiek problemów w pracy z tą technologią.

Literatura

[1] ISTQB: Certified Tester - Foundation Level Sylabus, ISTQB, 2018

[2] ISTQB: Advanced Level – Test Automation Engineer Sylabus. ISTQB, 2016

[3] Zbiór informacji o testowaniu, https://pwicherski.gitbook.io/, [28.06.2020]

[4] Informacje o TestNG, https://testerzy.pl/baza-wiedzy/akcja-automatyzacja-czesc-2-junit-testng-porownanie , [30.06.2020]

[5] Informacje o JUnit, https://en.wikipedia.org/wiki/JUnit, [30.06.2020]

[6] B. Garcia, Mastering Software Testing with JUnit 5: Comprehensive guide to develop high quality Java applications, Packt Publishing Ltd, 2017

[7] Dokumentacja frameworka JUnit, https://junit.org/junit5/docs/current/user-guide/, [10.07.2020]

[8] Składnia frameworków JUnit oraz TestNG, https://www.toolsqa.com/testng/testng-vs-junit/, [28.06.2020]

[9] Popularność JUnit oraz TestNG, https://blog.overops.com/junit-vs-testng-which-testing-framework-should-you-choose/, [10.07.2020]