|  |  |
| --- | --- |
| Technologie Obiektowe | |
| Projekt - **Porównanie rozwiązań związanych z testowaniem** | |
| **Patryk Banaś**  **Studia II stopnia** | 27.06.2022 |
| **1ID21A**  2021/2022 | |

**1. Po co testy?**

Bardzo szybki i dynamiczny rozwój technologii informatycznych zmusza do tworzenia wydajnych i niezawodnych systemów, które muszą sprostać coraz większym i bardziej skomplikowanym wymaganiom klientów. Klienci wymagają niezawodności, dostarczane przez nich usługi kojarzą się z ich wizerunkiem, nie można więc dopuszczać do tego aby oprogramowanie działało w błędny sposób. Czym wcześniej zostanie wykryty błąd tym są mniejsze koszty jego naprawienia, dzięki temu pozwala to firmą na zaoszczędzenie pieniędzy. W historii bywały sytuacje które kosztowały niektóre firmy nawet i kilkaset milionów dolarów których przyczyną były niedokładne testy. Przykładem może być Błąd dzielenia zmiennoprzecinkowego w procesorze „Pentium” w 1994. Pisanie testów jednostkowych, pozwala na wykrycie błędów w najwcześniejszej możliwej fazie, już w trackie pisania programu. Każdy programista powinien przed oddaniem kodu do użytku przetestować, czy wszystko działa jak należy. Manualne testowanie to jednak żmudna, mozolna i czasochłonna praca. Często zdarza się również że programista przeoczony jakiś mały błąd który zakończy się błędem w programie. Jako cel testu przyjmuję się weryfikację określonego zachowania obiektu z faktycznym. Dlatego pisze się testy jednostkowe, polegające na wydzielaniu, mniejszych części/jednostek programu i testowanie ich w odosobnieniu. Testy zabierają całkiem sporo czasu deweloperom, dodatkowo żeby napisać dobre testy, należy dobrze znać projekt, najlepiej więc kiedy testy te pisane są przez osobę która jest autorem testowanych funkcjonalności. Dla przykładu jeżeli projekt składa się z kilku różnych modułów i jeden z nich ulega modyfikacji, w fazie testów przez testera, może okazać się że któryś z nich przestał działać w odpowiedni sposób. Może również wystąpić taka sytuacja, że pomimo wprowadzonej poprawce przez programistę dla jednego z modułów, kolejny przestaje działać, co znowu zmusza do tego, aby program ponownie wrócił do poprawek przez dewelopera. Testy jednostkowe mogłyby zapobiec błędom, jeszcze przed wyjściem oprogramowania do testów manualnych. Każdy test jednostkowy powinien zawierać w pierwszej kolejności, dane wejściowe, następnie przeprowadzany jest test i jego wynik jest sprawdzany. Pianie testów jednostkowych powinno polegać na sprawdzenie kodu w możliwie jak najlepszy sposób, natomiast należy mieć na uwadze to, że czym będzie więcej kodu tym ciężej go będzie utrzymać dlatego należy w jakiś sposób ujednolicić testy. Trzeba pisać testy jednostkowe z głową i wybierać to co najistotniejsze, ponieważ uruchomienie testów mogłoby trwać bardzo długo, jeżeli będą pisane w drobny sposób.

Automatyzacja ma na celu zautomatyzowanie najczęściej powtarzanych czynności wykonywanych w ręczny sposób. Dzięki wykorzystaniu testów wzrasta produktywność, jeżeli testy zostaną dobrze zaprojektowane, tester nie musi w żaden sposób podejmować dodatkowych działań. Testy powinno gładko przechodzić po kolei przez wszystkie kroki bez zatrzymywania się, nawet gdy po drodze zostanie napotkany błąd. Możliwe jest więc wykonywania testów poza godzinami pracy testera, otrzymując po ich przebiegu raportów. Testy manualne są trudne do powtórzenia, natomiast testy automatyczne są uruchamiane zawsze w taki sam sposób, dzięki czemu ewentualne rozbieżności przy uruchomieniach testów są wykluczane i wzrasta niezawodność. Wzrasta również pokrycie testami aplikacji, testy automatyczne są wykonywane zdecydowanie szybciej niż manuale. Możliwe jest dzięki temu również przetestowanie obciążeń czy przeciążeń systemu, np. klikniecie w jednym czasie na ten sam punkt w aplikacji klienckiej.

**2. Metody testowania.**

Wyróżnione są róże typy testów:

* Testy modułowe polegają na testowaniu najmniejszych komponentów możliwych do wyizolowania mogą to być moduły, klasy, metody.
* Testy polegają na sprawdzeniu interakcji pomiędzy poszczególnymi modułami.
* Testy systemowe mają za zadanie przetestować zachowanie oprogramowania w ujęciu całościowym.
* Testy akceptacyjne mają na celu potwierdzenie jakości i spójności gotowego już oprogramowania.
* Testy alfa są przeprowadzane przez potencjalnych przyszłych użytkowników danego oprogramowania.
* Testy beta przeprowadzane są już na rzeczywistym środowisku produkcyjnym przez potencjalnych użytkowników.
* Testy regresji polegają na sprawdzeniu większego obszaru aplikacji  
  w celu weryfikacji czy np. wprowadzone zmiany w implementacji wynikające z dodania nowej funkcjonalności nie spowodowały błędów w innych częściach oprogramowania.
* Testy pielęgnacyjne dokonywane są już na wdrożonym i użytkowanym systemie informatycznym.

Są również różne podejścia do pisania testów wyróżniane są np. takie jak:

**Test-Driven Development –** tworzenie aplikacji zaczyna się od napisania do niej testów, deweloper otrzymuje wiedze jak ma działać dana funkcjonalność i pisze kod do momentu aż testy nie będą miały zadowalających wyników. Takie podejście sprawdza się w przypadku bardzo dużych projektów, pozwala zminimalizować ryzyko wystąpienia błędów.

**Behaviour-Driven Development –** jest to podejście porównywane do TDD, tutaj również pracę zaczyna się od przygotowania testów, dla nieistniejącej jeszcze aplikacji i powtarza cykl testowania. Różnicą jednak jest to że dokumentację i testy piszę się w języku natułanym, czyli zbliżonym do tego jaki wykorzystuje się w codziennym życiu. Takie podejście ułatwia pracę zespołowi technicznemu jak i osobie zamawiającej aplikację. Przy takim podejściu często wykorzystuje się narzędzie Cucumber.

**Domain-Driven Design –** jest to podejście oparte o dziedzinę, zespół pracujący nad systemem, jako główny cel przyjmuję aby dostarczyć zakładaną wartość biznesową. Programiści musza mieć śwadomośc jaka jest największa wartość tworzonego oprogramowania w którym źródłem wiedzy jest ekspert domenowy. Zaletą takiego podejścia jest lepsze zrozumienie pomiędzy wszystkimi osobami zaangażowanymi w proces wytwarzania aplikacji.

Testy można zapisywać w postaci: Given-When-Then, przykład:

|  |
| --- |
| @Test public void analyzeHealthByVitalSigns\_allValuesOk\_resultTrue() {   //given  Integer systolicBloodPressure = 80;  Integer diastolicBloodPressure = 80;  Integer pulse = 80;  HealthClinic clinic = new HealthClinic();   //when  boolean result = clinic.analyzeHealthByVitalSigns(systolicBloodPressure, diastolicBloodPressure, pulse);   //then  Assertions.*assertTrue*(result); } |

Przy pisaniu testów warto kierować się zasadami zebranymi pod akronimem **FIRST**:

* **F**ast – Testy jednostkowe powinny być szybkie, służą one do testowania najmniejszych elementów systemów, zwykle metod,
* **I**solated – Powinny być izolowane, co oznacza że w żadnym wypadku jeden test jednostkowy nie powinien być zależy od innego.
* **R**epeatable – (powtarzalność) Konkretne testy powinny dawać identyczny rezulatat, jak za pierwszym i każdym kolejnym razem.
* **S**elf-validating – Test jedostkowy mudi dawać jednoznaczną odpowiedź czy przeszedł pozytywnie czy też nie, manualna interpretacja rezultatu nie powinna mieć miejsca.
* **T**imely/**T**horough – Test powinien być napisany w dowolnym momencie czasu, należy myśleć o testach w trakcie pisania kodu produkcyjnego.

**3. Technologie.**

**Jenkins** to jeden z najbardziej znanych systemów Continuous Integration i Continuous Delivery. Przede przede wszystkim Jenkis to narzędzie CI/CD, wiec pozwala on na pełną automatyzację wszystkich procesów:

* kompilowania projektu;
* wykonania testów jednostkowych;
* wykonania testów integracyjnych;
* budowanie i publikowania aplikacji.

**Cucumber** to narzędzie oparte na frameworku Behavior Driven Development (BDD). BDD to metodologia umożliwiająca zrozumienie funkcjonalności aplikacji w postaci prostej reprezentacji tekstowej. Głównym celem tej platformy jest to żeby wszyscy pracujący nad projektem, zrozumieli aplikację, bez zrozumienia aspektów technicznych. W korniszonie są używane miedzy innymi następujące terminy:

* Funkcja - Plik funkcji musi zawierać opis wysokiego poziomu testowanej aplikacji.
* Scenariusz - Scenariusz to testowa specyfikacja testowanej funkcjonalności.

|  |  |  |
| --- | --- | --- |
|  | TestNG | JUnit |
| **Raportowanie** | TAK | TAK |
| **Ignorowanie testów** | TAK | TAK |
| **Implementacja Listeners** | TAK | TAK |
| **Kolejność wykonywania testów** | TAK | TAK |
| **Realizacje założeń** | NIE | TAK |
| **Integracja z Maven** | TAK | TAK |
| **Równoległe wykonywanie testów** | TAK | NIE |
| **Test zależności** | TAK | NIE |
| **Pakiet testowy** | TAK | TAK |
| **Testy parametryzowane** | TAK | TAK |
| **Wyjątki** | TAK | TAK |
| **Realizacja twierdzeń** | TAK | TAK |
| **Implementacja adnotacji** | TAK | TAK |
| **Główne wsparcie IDE IntelliJ** | TAK | TAK |
| **Przyjazny użytkownikowi** | TAK | NIE |
| **Framework open source** | TAK | TAK |
| **Limit czasu na testy** | TAK | TAK |

**TestNG** pozwala na testy zależności, dzięki czemu jedna metoda może być uzależniona od drugiej, będzie ona działać dopiero po uruchomieniu zależnej metody i jej poprawnym wykonaniem, w innym wypadku test zależny nie zostanie uruchomiony. TestNG pozwala również w prosty sposób na uruchomienie równolegle wykonywanych testów, np. przy użyciu adnotacji @Test(threadPoolSize = 3, invocationCount = 9), daje również możliwość uruchomienia całych pakietów testów równolegle, jeśli są one określone w pliku konfiguracyjnym XML. W przypadku JUnit trzeba napisać niestandardową metodę runner i wielokrotnie podawać te same parametry testowe.

**JUnit** składa się z trzech różnych podprojektów takich jak: JUnit Platform, JUnit Jupiter, JUnit Vintage, dlatego tez jest mniej przyjazny dla użytkownika. W tej bibliotece używa się adnotacji, do oznaczania metod jako testy, lub ich konfigurowania. Do głównych z nich można zaliczyć:

@Test – określa, że dana metoda będzie metodą testową;

@BeforeEach – metoda zostanie wywołana przed każdą metodą testową;

@AfterEach – metoda zostanie wywołania po każdej metodzie testowej;

@BeforeAll – metoda zostanie wykonana przed wszystkimi metodami testowymi;

@AfterAll – metoda zostanie wywołana po wszystkich metodach testowych;

TestNG to biblioteka do automatyzacji testów dla Javy, stworzona przez Cedrica Beusta, został zainspirowany do jest stworzenia przez ograniczenia występujące w JUnit.

@ignore - ignorowanie określonego testu;

@Test(expected= ArithmeticException.class) – określenie spodziewanego wyjątku;

@Test(timeout = 1000) – określenie maksymalnego czasu na wykonywanie testu;

@RepeatedTest(10) - powtarzanie testu;

Przykładowe adnotacje w **TestNG**:

@Test - oznacza klasę lub metodę jako część testu;

@Test(enabled=false) – ignorowanie określonego testu;

@Test(invocationCount = 10) –powtarzanie testu;

@Test(expectedException= ArithmeticException.class) – określenie spodziewanego wyjątku;

@Test(timeOut = 1000) – określenie maksymalnego czasu na wykonywanie testu;

@BeforeSuite - metoda zostanie uruchomiona, zanim wszystkie testy w tym pakiecie zostaną uruchomione (Nie występuje w JUnit);

@AfterSuite - metoda zostanie uruchomiona po uruchomieniu wszystkich testów w tym pakiecie (Nie występuje w JUnit);

@BeforeTest - metoda zostanie uruchomiona, zanim zostanie uruchomiona jakakolwiek metoda testowa należąca do klas wewnątrz znacznika <test>(Nie występuje w JUnit);

@AfterTest -metoda zostanie uruchomiona po uruchomieniu wszystkich metod testowych należących do klas wewnątrz znacznika <test>(Nie występuje w JUnit);

@BeforeGroups - lista grup, dla których ta metoda konfiguracji będzie wcześniej uruchamiana. Ta metoda jest uruchamiana na krótko przed wywołaniem pierwszej metody testowej należącej do którejkolwiek z tych grup (Nie występuje w JUnit);

@AfterGroups - lista grup, dla których ta metoda konfiguracji będzie wcześniej uruchamiana. Ta metoda jest uruchamiana na krótko po wywołaniu ostatniej metody testowej należącej do którejkolwiek z tych grup(Nie występuje w JUnit);

@BeforeClass - metoda zostanie uruchomiona przed wywołaniem pierwszej metody testowej w bieżącej klasie;

@AfterClass - metoda zostanie uruchomiona po uruchomieniu wszystkich metod testowych w bieżącej klasie;

@BeforeMethod - metoda zostanie uruchomiona przed uruchomieniem każdej metody testowej w bieżącej klasie;

@AfterMethod - metoda zostanie uruchomiona po każdej metodzie testu;

TestNG umożliwia tworzenia adnotacji jako grupy testowe, co pozwoli np. uruchomienie kodu przed lub po grupach przypadków testowych. Pojedyncze testy mogą również należeć do wielu grup, aby następnie mogły być uruchomione w różnych kontekstach (jak np. testy szybkie lub wolne). W JUnit do podobnych zastosowań używa się adnotacji @Tag.

Obie biblioteki umożliwiają uruchamianie testów na wielu wątkach, w przypadku kiedy korzysta się z JUnit Jupiter, możliwe jest dodanie do pliku junit-platform.properties w folderze our src/test/resources następującej linii:

junit.jupiter.execution.parallel.enabled = true. W przypadku TestNG, można to zrobić np. w następujący sposób:

@Test(threadPoolSize = 3, invocationCount = 9)

Zarówno w JUnit jak i TestNG, mogą być wykonywane testy sparametryzowane.

JUnit pozwala na użycie TestInfo co pozwala uzyskać standardowe informacje na temat naszego testu.

**Generator testów jednostkowych Diffblue.**

Diffblue Cover to zautomatyzowane narzędzie do pisania testów jednostkowych. Analizuje istniejącą aplikację Java i pisze testy jednostkowe, które odzwierciedlają bieżące zachowanie, zwiększając pokrycie testów i pomagając znaleźć regresje w przyszłych zmianach kodu, używając przy tym mock’ów. Obsługuje Java 8, 11 , Spring i Spring Boot. Wykorzystuje tak zwane wyszukiwanie probabilistyczne, aby szukać najbardziej prawdopodobnych rozwiązań w obszarze testowania.

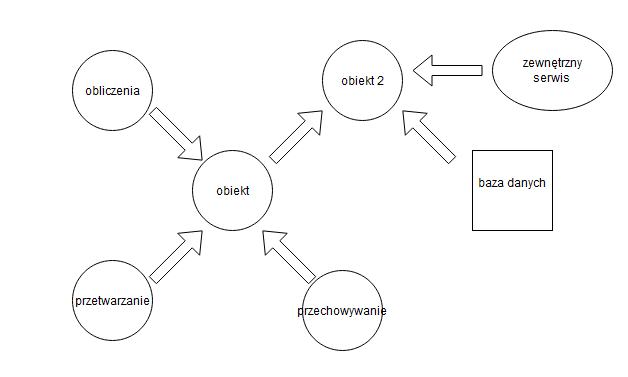
|  |
| --- |
| @Test void testAddUser() {  User user = new User();  user.setAppUserRole(AppUserRole.*USER*);  user.setEmail("jane.doe@example.org");  user.setEnabled(true);  user.setId(123L);  user.setLocked(true);  user.setPassword("iloveyou");  user.setUsername("janedoe");   User user1 = new User();  user1.setAppUserRole(AppUserRole.*USER*);  user1.setEmail("jane.doe@example.org");  user1.setEnabled(true);  user1.setId(123L);  user1.setLocked(true);  user1.setPassword("iloveyou");  user1.setUsername("janedoe");  Optional<User> ofResult = Optional.*of*(user1);   User user2 = new User();  user2.setAppUserRole(AppUserRole.*USER*);  user2.setEmail("jane.doe@example.org");  user2.setEnabled(true);  user2.setId(123L);  user2.setLocked(true);  user2.setPassword("iloveyou");  user2.setUsername("janedoe");  Optional<User> ofResult1 = Optional.*of*(user2);  UserRepository userRepository = *mock*(UserRepository.class);  *when*(userRepository.save((User) *any*())).thenReturn(user);  *when*(userRepository.findByEmail((String) *any*())).thenReturn(ofResult);  *when*(userRepository.findByUsername((String) *any*())).thenReturn(ofResult1);  ObjectMapper objectMapper = new ObjectMapper();  ConfirmationTokenRepository confirmationTokenRepository = *mock*(ConfirmationTokenRepository.class);  UserDetailRepository userDetailRepository = *mock*(UserDetailRepository.class);  PostRepository postRepository = *mock*(PostRepository.class);  InterestRepository interestRepository = *mock*(InterestRepository.class);  CommentRepository commentRepository = *mock*(CommentRepository.class);  BCryptPasswordEncoder bCryptPasswordEncoder = new BCryptPasswordEncoder();  UserService userService = new UserService(userRepository, objectMapper, confirmationTokenRepository,  userDetailRepository, postRepository, interestRepository, commentRepository, bCryptPasswordEncoder,  new ConfirmationTokenService(*mock*(ConfirmationTokenRepository.class)));   User user3 = new User();  user3.setAppUserRole(AppUserRole.*USER*);  user3.setEmail("jane.doe@example.org");  user3.setEnabled(true);  user3.setId(123L);  user3.setLocked(true);  user3.setPassword("iloveyou");  user3.setUsername("janedoe");  ResponseEntity actualAddUserResult = userService.addUser(user3);  *assertNull*(actualAddUserResult.getBody());  *assertEquals*(HttpStatus.*UNPROCESSABLE\_ENTITY*, actualAddUserResult.getStatusCode());  *assertTrue*(actualAddUserResult.getHeaders().isEmpty());  *verify*(userRepository).findByEmail((String) *any*());  *verify*(userRepository).findByUsername((String) *any*()); } |

Przykładowy test testujący metodę dodającą użytkownika do serwisu napisany przez Diffblue.

|  |
| --- |
| @Test void testValidate() {  *assertTrue*(UserService.*validate*("jane.doe@example.org"));  *assertFalse*(UserService.*validate*("Email Str")); } |

Kolejny test który sprawdza metodę walidującą email na podstawie regexa, czy jego format jest poprawny, również wygenerowany przez Diffblue.

**Mockowanie**

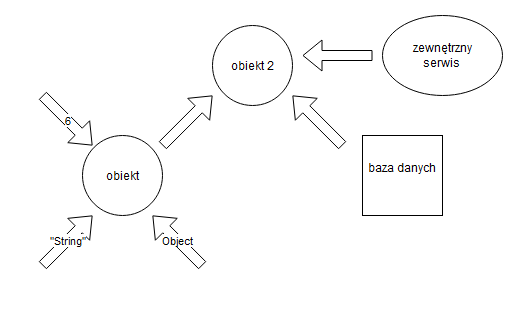


Obiekt javovy do swojego działania potrzebuje innych obiektów realizujących np.:

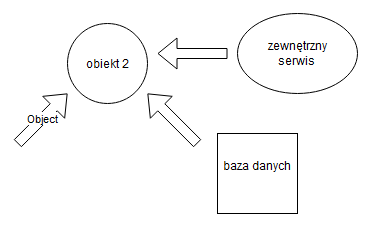
* Obliczenia
* Przetwarzających dane
* Przechowujących dane do przetworzenia przez obiekt

Z obiektu korzysta obiekt 2, który korzysta z zewnętrznego serwisu oraz bazą danych. Stworzenie testów dla obiekt 2 polegałoby na stworzeniu obiektów znajdujących się najgłębiej na grafie i zasilić je danymi(obliczenia, przetwarzanie, przechowywanie), następnie stworzyć obiekt który będzie je wykorzystał (obiekt), zapewnić połączenie z bazą danych oraz serwisem, ostatecznie tworząc instancję obiektu 2 i go przetestować. Należałyby dodatkowo przygotować kilka przypadków testowych, powtarzając cały cykl kilkukrotnie. Pisanie w taki sposób testów jest bardzo pracochłonne.

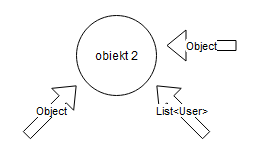
Mock – obiekt imitacji, ma za zadanie symulować działanie całego obiektu. Dla przykładu, obiekt przeprowadzający kalkulacje, posiada dane wejściowe oraz wynik. Wynik ten używany jest przez obiekt, nie potrzebne jest, aby obiekt obliczenia, wykonywał te kalkulacje, ponieważ potrzebny jest tylko wynik jaki ten obiekt zwróci, np. liczbę 6, nie są ważne w takim wypadku szczegóły implementacyjne, więc i cała klasa obliczenia nie będzie potrzebna, wystarczy użyć tylko wyniku. Podobnie ma się sytuacja z pozostałymi obiektami, uzyskując następującą sytuację:



Można to jeszcze bardziej uprościć, ponieważ obiekt, jest w tej samej relacji co obiekty za imitowane, do obiektu 2.



Zamiast łączyć się z bazą danych, np. w celu uzyskania listy użytkowników, można również stworzyć imitację która zwróci taką listę, tak samo można zrobić tak w przypadku zewnętrznego serwisu. Wszystkie te operacje znacznie poprawią wydajność testów i uzyskując następujący graf:



**Mockito** jest dużo bardziej popularnym framework’iem od **EasyMock**, co za tym idzie, jest łatwiejsze znalezienie problemów z użyciem tej technologii wśród społeczności. Do głównych różnić między tymi narzędziami należy między innymi składnia:

Zamockowanie metod:

* EasyMock:

expect(userRepository.save(isA(User.class))).andReturn(user);

* Mockito:

when(userRepository.save((User) any())).thenReturn(user);

Przed użyciem verify() w EasyMock, należy użyć replay(), co nie występuje w Mockito.

**4. Testy.**

Testy zostały przeprowadzone na komputerze Inspiron 5577 z procesorem Intel(R) Core(TM) i7-7700HQ CPU @ 2.80GHz 2.80 GHz, 8GB RAM, systemem operacyjnym x64 Windows Home.

W poniżej tabeli przedstawione zostały czasy testowania jednej metody testAddUser(), w klasie UserServiceTest dla JUnit i TestNG z wykorzystaniem Mockito. Wszystkie wyniki testów są podawane w mili sekundach.

|  |  |
| --- | --- |
| testAddUser() | |
| JUnit | TestNG |
| 1905 | 1481 |
| 1935 | 1322 |
| 2524 | 1661 |
| 1642 | 1557 |

*Testy metody testAddUser()*

*Wykres kolumnowy testów metody testAddUser()*

Kolejnym krokiem było stworzenie testów z wykorzystaniem Mockito dla serwisu UserService, zarówno dla JUnit, jak i TestNG. Testy zostały przeprowadzone po cztery razy, a wyniki przedstawiono w tabeli. Wygląd struktury wykonanych testów w porównywanych narzędziach różni się, co przedstawiono na poniższych zrzutach ekranu.

Obraz zawierający tekst

Opis wygenerowany automatycznie

*Metody testowane w UserService - JUnit*

Obraz zawierający tekst

Opis wygenerowany automatycznie

*Metody testowane w UserService - TestNG*

Następnie przeprowadzono testy na całym serwisie, wyniki zostały przedstawione poniżej:

|  |  |
| --- | --- |
| UserService | |
| JUnit | TestNG |
| 2404ms | 1735ms |
| 2828ms | 1773ms |
| 2588ms | 1820ms |
| 2694ms | 2094ms |

*Testy serwisu UserService*

*Wykres kolumnowy testów serwisu UserService*

Przeprowadzono również testy na powtarzaniu poszczególnych metod, wybrane z nich zostały powtórzone po 1000 razy.

|  |  |
| --- | --- |
| UserService x1000 | |
| JUnit | TestNG |
| 9378 | 5914 |
| 8982 | 6383 |
| 8061 | 6509 |
| 8354 | 6398 |

*Testy serwisu UserService x1000 – wybrane metody*

*Wykres kolumnowy testów serwisu UserService x1000 – wybrane metody*

W testach wykorzystano również uruchamianie wielowątkowe, zostały one uruchomione z powtarzaniem wybranych metod razy 1000, w celu zwiększenia operacji.

|  |  |  |  |
| --- | --- | --- | --- |
| UserService x1000 – TestNG – ThreadPoolSize | | | |
| TestNG | TestNG x2 | TestNG x4 | TestNG x8 |
| 5914 | 9733 | 22187 | 38403 |
| 6383 | 9560 | 25807 | 38492 |
| 6509 | 9224 | 22490 | 38453 |
| 6398 | 9349 | 23453 | 38438 |

*Testy serwisu UserService x1000 - TestNG – wybrane metody*

*Wykres kolumnowy testów serwisu UserService x1000-* TestNG *– wybrane metody*

|  |  |  |  |
| --- | --- | --- | --- |
| UserService x1000 – JUnit – Parrarel | | | |
| JUnit | JUnit x2 | JUnit x4 | JUnit x8 |
| 9066 | 9932 | 10080 | 5942 |
| 8306 | 9733 | 7926 | 6599 |
| 8709 | 9681 | 8075 | 6802 |
| 9013 | 9548 | 9177 | 6206 |

*Testy serwisu UserService x1000- JUnit – wybrane metody*

*Wykres kolumnowy testów serwisu UserService x1000-* JUnit *– wybrane metody*

Poniżej został przedstawiony przykładowy test stworzony dla EasyMock:

|  |
| --- |
| @Test public void whenUserUsernameIsProvided\_thenRetrievedUser() {  UserRepository userRepository = *mock*(UserRepository.class);   User user = new User();  user.setAppUserRole(AppUserRole.*USER*);  user.setEmail("jane.doe@example.org");  user.setEnabled(true);  user.setId(123L);  user.setLocked(true);  user.setPassword("iloveyou");  user.setUsername("janedoe");  Optional<User> ofResult = Optional.*of*(user);   EasyMock.*expect*(userRepository.findByUsername(EasyMock.*anyString*())).andReturn(ofResult);  EasyMock.*replay*(userRepository);  Optional<User> userMock = userRepository.findByUsername("Marek");   *assertNotNull*(userMock);  *assertEquals*(user.getUsername(), userMock.get().getUsername()); } |

Poniżej został przedstawiony przykładowy test stworzony dla Mockito:

|  |
| --- |
| @Test public void whenUserUsernameIsProvided\_thenRetrievedUser() {  UserRepository userRepository = *mock*(UserRepository.class);   User user = new User();  user.setAppUserRole(AppUserRole.*USER*);  user.setEmail("jane.doe@example.org");  user.setEnabled(true);  user.setId(123L);  user.setLocked(true);  user.setPassword("iloveyou");  user.setUsername("janedoe");  Optional<User> ofResult = Optional.*of*(user);   *when*(userRepository.findByUsername(*anyString*())).thenReturn(ofResult);   Optional<User> userMock = userRepository.findByUsername("Marek");   *assertNotNull*(userMock);  *assertEquals*(user.getUsername(), userMock.get().getUsername()); } |

Poniżej przedstawiono testy z wykorzystaniem Mockito, jak i EasyMockito, w TestNG, a następnie w JUnit, na repozytorium.

|  |  |
| --- | --- |
| UserRepository - TestNG | |
| Mockito | EasyMock |
| 1492 | 116 |
| 1134 | 146 |
| 949 | 78 |
| 966 | 71 |

*Testy repozytorium UserRepository - TestNG*

*Wykres kolumnowy testów Repozytorium UserRepository*

|  |  |
| --- | --- |
| UserRepository - JUnit | |
| Mockito | EasyMock |
| 306 | 163 |
| 247 | 184 |
| 285 | 163 |
| 254 | 163 |

*Testy repozytorium UserRepository – Junit -Mockowanie*

*Wykres kolumnowy testów Repozytorium UserRepository - Mockowanie*

|  |  |
| --- | --- |
| UserRepository – TestNG - Repeat | |
| Mockito | EasyMock |
| 2421 | 478 |
| 2502 | 580 |
| 2017 | 520 |
| 2223 | 589 |

*Testy repozytorium UserRepository – TestNG -Mockowanie – wiele testów*

Przy Mockowaniu zostały również przeporwadzone testy z powtarzalnością metod, zarówno dla JUnit jak i TestNG, wyniki zostały przedstawione poniżej.

*Wykres kolumnowy testów Repozytorium UserRepository- TestNG – Mockowanie -wiele testów*

|  |  |
| --- | --- |
| UserRepository – JUnit Repeat | |
| Mockito | EasyMock |
| 8054 | 6371 |
| 7969 | 6715 |
| 8114 | 6518 |
| 7956 | 6681 |

*Testy repozytorium UserRepository – JUnit - Mockowanie -wiele testów*

*Wykres kolumnowy testów Repozytorium UserRepository- JUnit – Mockowanie -wiele testów*

**5. Wnioski.**

Wszystkie testy zostały odwzorowane na poszczególne biblioteki dla swoich specyfikacji, w taki sposób, aby były do siebie jak najbardziej podobne, a wyniki bardziej wiarygodne. Spośród wszystkich wywoływanych metod wykonanie pierwszej testowej metody trwa najdłużej. Przy wykorzystaniu „threadPoolSize”, w TestNG czas przedstawiony w IDE różnił się od czasu rzeczywistego, był on większy, najprawdopodobniej był zliczany czas pracy wątków jednocześnie. Testy wykazały że w każdym przypadku oprócz wielowątkowości, wyniki szybciej były uzyskiwane dla TestNG. Te Framweorki, różnią się w dość spory sposób od siebie, większość adnotacji różni się od siebie, a niektóre o danej funkcjonalności nie występują w drugiej tehcnologii. W Przypadku Mockowania, to EasyMock wygrało z Mockito pod względem szybkości działania, jednak to Mockito, ma większe grono użytkowników i dzięki czemu jego użycie, może czasami być mniej problematyczne, natomiast pomimo niewielkich różnic, są one do siebie bardzo podobne. Testowanie różnych bibliotek w jednym projekcie, nie jest dobrym pomysłem, stwarza bardzo wiele problemów, które są trudne w rozwiązaniu. Wykorzystanie wielowątkowości w testach nie zawsze, może przynieść dobre rezultaty, natomiast jeżeli w projekcie występują metody które są czasochłonne i wywoływanie testów zostanie ustawione na wielowątkowe podzielone na funkcje, może przynieść spore korzyści. Testy są bardzo istotne w projekcie, natomiast trzeba również przemyśleć sprawę ich doboru i wykorzystania.