

Technologie Obiektowe (projekt)

Nazwa Uczelni:	Politechnika Świętokrzyska
Kierunek studiów:	Informatyka 2 (niestacjonarne)
Rok:	I
Semestr:	II
Grupa:	1IZ21A
Skład zespołu:	Pataleta Dominik
Temat projektu:	Porównanie rozwiązań związanych z testowaniem: JUnit i TestNG, Mockito i EasyMock

Spis treści

1.	Opis tematyki projektu	3
1.1.	Test jednostkowy.....	3
1.2.	Platforma Eclipse	3
1.3.	Język programowania Java	3
1.4.	Framework JUnit 5.....	4
1.5.	Framework TestNG	4
1.6.	Framework Mockito	5
1.7.	Framework EasyMock.....	5
2.	Instalacja środowiska	6
2.1.	Instalacja Eclipse	6
2.2.	Instalacja Frameworku JUnit	7
2.3.	Instalacja Frameworku TestNG.....	9
2.4.	Instalacja Frameworku Mockito	11
2.5.	Instalacja Frameworku EasyMock	12
3.	Porównanie Frameworków: JUnit z TestNG	13
3.1.	Podstawowe adnotacje w JUnit	15
3.2.	Przykłady adnotacji JUnit.....	16
3.3.	Podstawowe adnotacje w TestNG	18
3.4.	Przykłady adnotacji TestNG	19
3.5.	Porównanie adnotacji JUnit i TestNG	22
4.	Porównanie Frameworków: Mockito z EasyMock.....	24
4.1.	Adnotacje Mockito	25
4.2.	Adnotacje EasyMock.....	25
4.3.	Mockito vs EasyMock	26
4.3.1.	Weryfikacja w kolejności	27
4.3.2.	Stubbing metody void	28
4.3.3.	Dokładna liczba weryfikacji i dopasowania argumentów	28
4.4.	Porównanie kroków Mockito i EasyMock	29
5.	Technologie użyte do wykonania projektu	31
6.	Podsumowanie.....	32
7.	Literatura.....	33

1. Opis tematyki projektu

Tematem projektu jest porównanie rozwiązań związanych z testowaniem frameworków JUnit z TestNG oraz Mockito z EasyMock.

1.1. Test jednostkowy

Test jednostkowy (ang. unit test) to technika testowania tworzonego oprogramowania poprzez wykonywanie testów weryfikujących poprawność działania pojedynczych elementów (jednostek) programu - np. metod, obiektów. Testowany fragment programu poddawany jest testowi, który wykonuje go i porównuje wynik z oczekiwanymi wynikami. [1]

1.2. Platforma Eclipse

Eclipse - platforma (framework) napisana w 2004 roku w Javie do tworzenia aplikacji typu rich client. Na bazie Eclipse powstało zintegrowane środowisko programistyczne do tworzenia programów w Javie, które jest razem z tą platformą rozpowszechniane. Projekt został stworzony przez firmę IBM, a następnie udostępniony na zasadach otwartego oprogramowania. W chwili obecnej jest on rozwijany przez Fundację Eclipse. [2]

1.3. Język programowania Java

Java - współbieżny, oparty na klasach, obiektowy język programowania ogólnego zastosowania. Został stworzony przez grupę roboczą pod kierunkiem Jamesa Goslinga z firmy Sun Microsystems. Java jest językiem tworzenia programów źródłowych kompilowanych do kodu bajtowego, czyli postaci wykonywanej przez maszynę wirtualną. Język cechuje się silnym typowaniem. Jego podstawowe koncepcje zostały przejęte z języka Smalltalk (maszyna wirtualna, zarządzanie pamięcią) oraz z języka C++ (duża część składni i słów kluczowych). [3]

1.4. Framework JUnit 5

JUnit 5 to kolejna generacja JUnit. W przeciwieństwie do poprzednich wersji JUnit, JUnit 5 składa się z kilku różnych modułów z trzech różnych podprojektów: JUnit Platform, JUnit Jupiter oraz JUnit Vintage.

- Platforma JUnit służy jako podstawa do uruchamiania frameworków testowych na JVM. Definiuje również TestEngine interfejs API do tworzenia platformy testowej działającej na platformie. Ponadto platforma udostępnia konsolę do uruchamiania platformy z wiersza poleceń oraz silnik JUnit Platform Suite do uruchamiania niestandardowego zestawu testów przy użyciu jednego lub więcej silników testowych na platformie. Najwyższej klasy wsparcie dla platformy JUnit istnieje również w popularnych środowiskach IDE (IntelliJ IDEA, NetBeans, Visual Studio Code) oraz narzędziach do budowania (Gradle, Maven, Ant).
- JUnit Jupiter to połączenie nowego modelu programowania i modelu rozszerzeń do pisania testów i rozszerzeń w JUnit 5. Podprojekt Jupiter zapewnia TestEngine możliwość uruchamiania testów opartych na platformie Jupiter.
- JUnit Vintage zapewnia TestEngine możliwość uruchamiania testów opartych na JUnit 3 i JUnit 4 na platformie. Wymaga, aby w ścieżce klasy lub ścieżce modułu znajdował się JUnit 4.12 lub nowszy.

JUnit 5 wymaga Java 8 (lub nowszej) w czasie wykonywania. Jednak nadal można testować kod, który został skompilowany z poprzednimi wersjami JDK. [4]

1.5. Framework TestNG

TestNG to framework testowy zaprojektowany w celu uproszczenia szerokiego zakresu potrzeb testowych, od testów jednostkowych (testowanie klas w izolacji od pozostałych) po testy integracyjne (testowanie całych systemów złożonych z kilku klas, kilku pakietów, a nawet kilku zewnętrznych frameworków, takich jak serwery aplikacji).

TestNG to framework testowy inspirowany JUnit i NUnit, ale wprowadzający kilka nowych funkcji, które czynią go potężniejszym i łatwiejszym w użyciu, takich jak:

- Adnotacje,
- Metody zależne testowania serwerów aplikacji,
- Uruchamianie testów w dowolnie dużych pulach wątków z różnymi dostępnymi zasadami,
- Sprawdź, czy Twój kod jest bezpieczny w wielu wątkach,
- Elastyczna konfiguracja testu,
- Wsparcie dla testów opartych na danych (z @DataProvider),
- Wsparcie dla parametrów,
- Potężny model wykonania (bez TestSuite),
- Obsługiwane przez różne narzędzia i wtyczki (Eclipse, IDEA, Maven, itp),

TestNG został zaprojektowany tak, aby obejmował wszystkie kategorie testów: jednostkowe, funkcjonalne, kompleksowe, integracyjne, itp. [5]

1.6. Framework Mockito

Mockito to open source'owa platforma testowa dla Javy wydana na licencji MIT. Framework umożliwia tworzenie testowych obiektów podwójnych (mock objects) w zautomatyzowanych testach jednostkowych na potrzeby programowania sterowanego testami (TDD) lub programowania opartego na zachowaniu (BDD).

Mockito umożliwia programistom weryfikację zachowania testowanego systemu (SUT) bez wcześniejszego ustalania oczekiwań. Jedną z krytyki fałszywych obiektów jest to, że istnieje ścisłe sprzężenie kodu testowego z testowanym systemem. Mockito próbuje wyeliminować wzorzec oczekiwania-przebieg-weryfikuj poprzez usunięcie specyfikacji oczekiwań. Mockito dostarcza również kilka adnotacji, które pozwalają zredukować standardowy kod. [6]

Mockowanie Javy jest zdominowane przez biblioteki oczekujące na uruchomienie i weryfikację, takie jak EasyMock lub jMock. Mockito oferuje prostsze i bardziej intuicyjne podejście: zadajesz pytania dotyczące interakcji po wykonaniu. Za pomocą mockito możesz zweryfikować, czego chcesz. Korzystając z bibliotek oczekujących na uruchomienie i weryfikację, często jest się zmuszonym do pilnowania nieistotnych interakcji. Brak oczekiwania na uruchomienie weryfikacji oznacza również, że makiety Mockito są często gotowe bez kosztownej konfiguracji z góry. Mają być przejrzyste i pozwalają deweloperowi skupić się na testowaniu wybranych zachowań, a nie absorbować uwagę. Mockito ma bardzo smukłe API. Jest tylko jeden rodzaj makiety, jest tylko jeden sposób tworzenia makiet. Mockito ma podobną składnię do EasyMock, dlatego można bezpiecznie dokonać refaktoryzacji. Mockito nie rozumie pojęcia „oczekiwanie”. Jest tylko karczowanie i weryfikacje. [7]

1.7. Framework EasyMock

EasyMock to platforma testowa typu open source dla Javy wydana na licencji Apache License. Framework umożliwia tworzenie podwójnych obiektów testowych na potrzeby programowania sterowanego testami (TDD) lub rozwoju sterowanego zachowaniem (BDD). EasyMock zapewnia dynamicznie generowane obiekty Mock (w czasie wykonywania), bez konieczności ich implementacji. W EasyMock definicja Mock Object różni się od używania zaimplementowanego Mock Object. Obiekty makiety są budowane w czasie wykonywania i nie można zdefiniować dla tych obiektów dodatkowych implementacji. Początkowo pozwalał tylko na symulowanie interfejsów, z bezpiecznym mockowaniem typu i dodatkowymi funkcjami dodanymi w późniejszych opracowaniach. EasyMock może być używany w aplikacjach z często zmieniającymi się interfejsami. [8]

EasyMock był pierwszym dynamicznym generatorem Mock Object, odciążającym użytkowników od ręcznego pisania Mock Objects lub generowania dla nich kodu. EasyMock dostarcza Mock Objects, generując je w locie za pomocą mechanizmu proxy Java. [9]

2. Instalacja środowiska

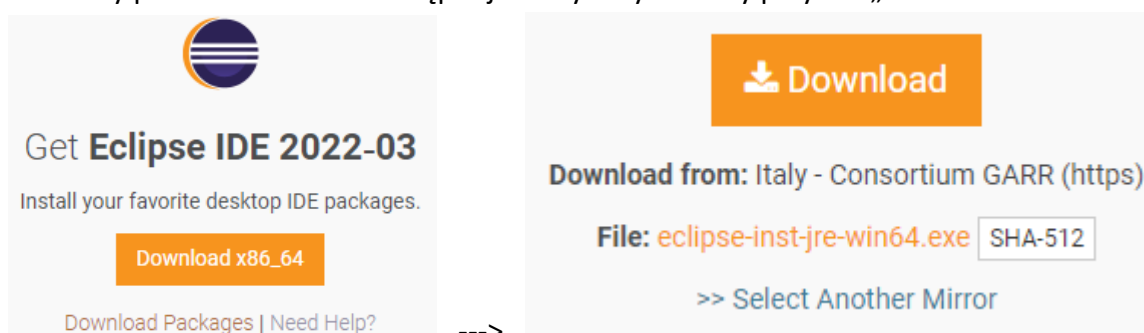
Ta sekcja została poświęcona na kilka etapów związanych z instalacją środowiska Eclipse oraz frameworków potrzebnych do wykonywania testów.

2.1. Instalacja Eclipse

Program Eclipse należy pobrać ze strony producenta dostępnej pod poniższym adresem:

<https://www.eclipse.org/downloads>

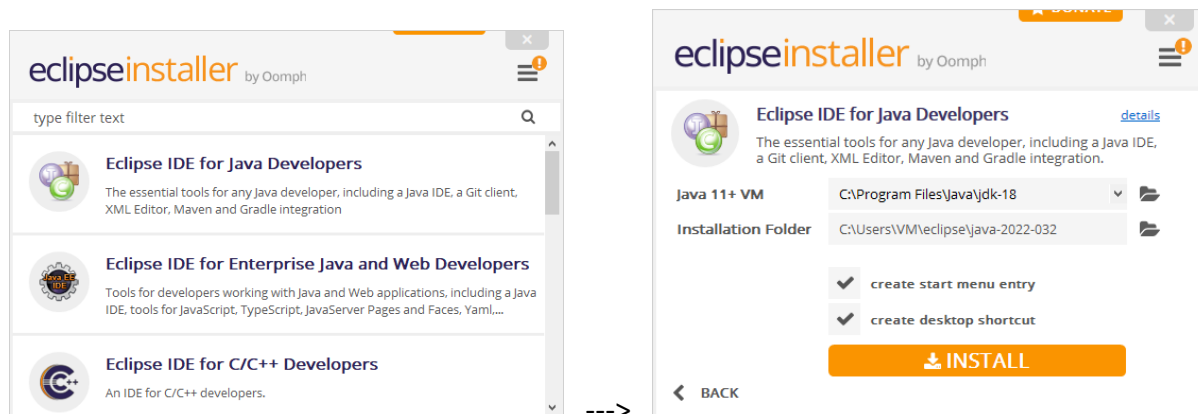
Pobieramy najnowszą wersję Eclipse IDE (2022-03) wybierając przycisk: „Download x86_64”, zostaniemy przekierowani do następnej strony - wybieramy przycisk „Download”.



Uwaga: przed przystąpieniem do instalacji należy zainstalować najpierw Jave (jdk), dostępną na stronie poniżej:

<https://www.oracle.com/java/technologies/downloads>

Podczas instalacja można dokonać wyboru jaki IDE nas interesuje. W przypadku wykonywania tego projektu będzie to pierwszy wybór – „Eclipse IDE for Java Developers”.

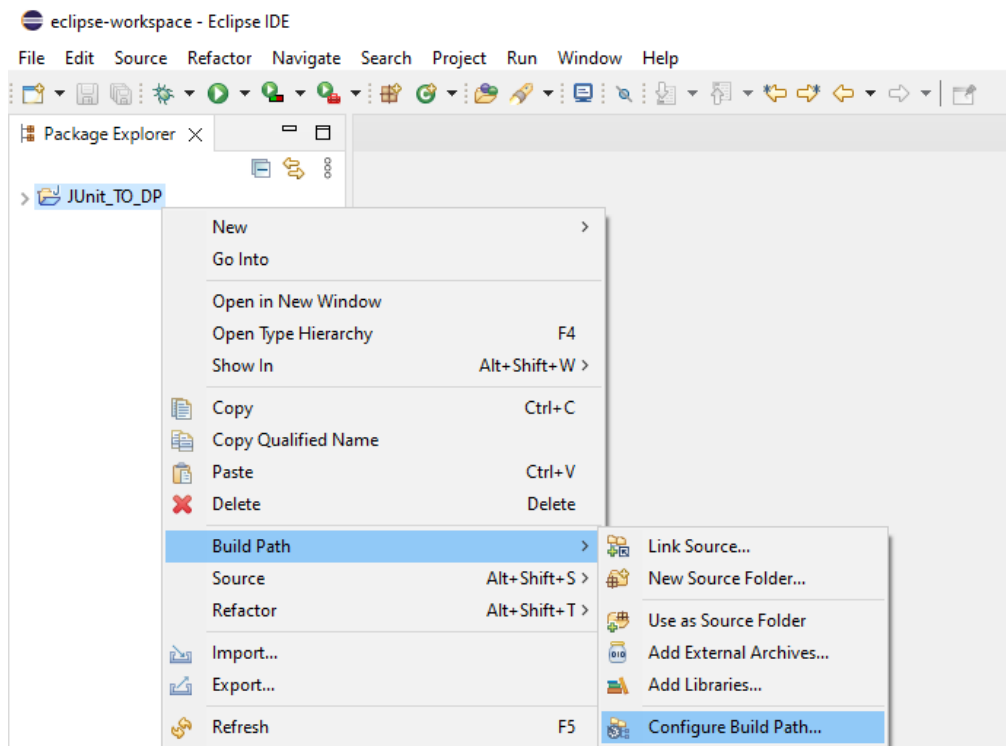


Na tym etapie możliwe jest wybranie innej lokalizacji instalacji aplikacji, należy również wskazać lokalizację wcześniej zainstalowanej Javy jeśli nie została wykryta przez instalator lub nie została zainstalowana, następnie pozostaje już tylko wybranie przycisku „INSTALL”. Jeśli chodzi o instalację Eclipse IDE to wszystko.

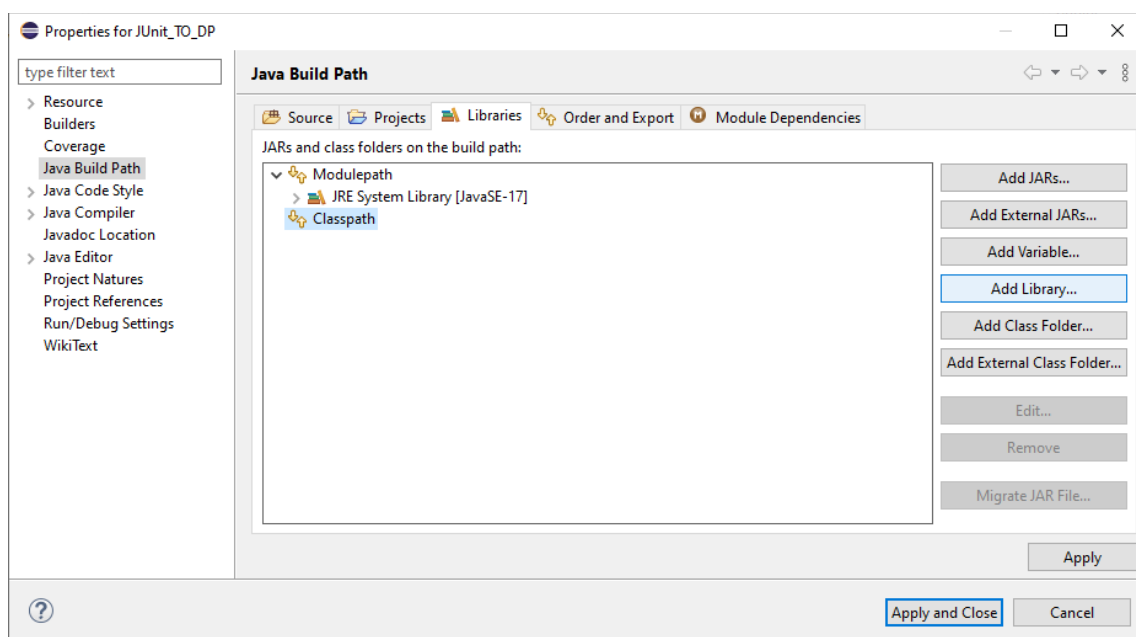
2.2. Instalacja Frameworku JUnit

W celu dodania Frameworku JUnit 5 do projektu postępujemy kolejno według instrukcji (link do pełnej instrukcji instalacji znajduje się na samym końcu).

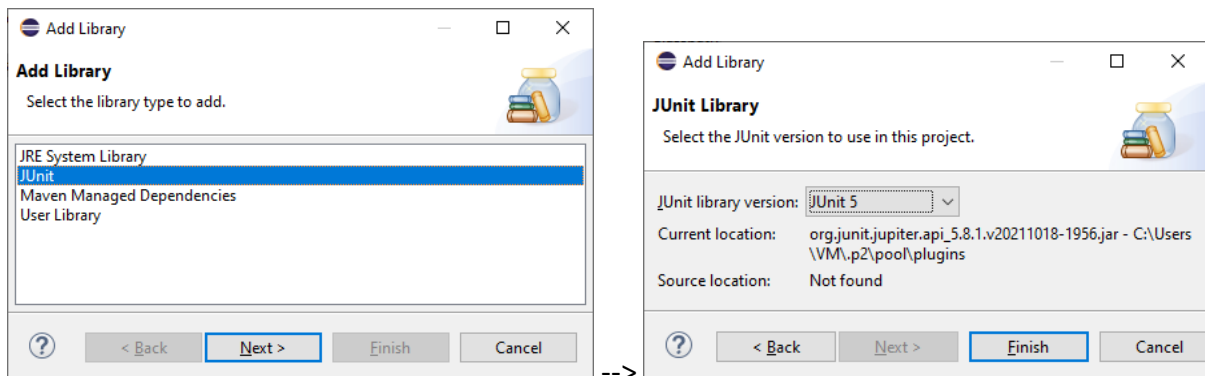
Do projektu należy dodać odpowiednią bibliotekę JUnit. Wybieramy główny katalog z projektem (w tym przypadku jest to „JUnit_TO_DP”, następnie prawym przyciskiem myszy rozwijamy menu, z którego wybieramy „Build Path”, a następnie „Configure Build Path...”.



W nowym oknie wybieramy „Java Build Path” następnie zakładkę „Libraries”. Zaznaczamy „Classpath”, następnie należy wybrać opcję „Add Library...”.



W nowym oknie należy zaznaczyć „JUnit” i nacisnąć „Next >”. W tym oknie można określić jaką wersja JUnit nas interesuje (dostępne 3, 4 oraz 5), kończymy instalację Frameworku klikając przycisk „Finish”, wracając do poprzedniego okna wybieramy „Apply and Close”.

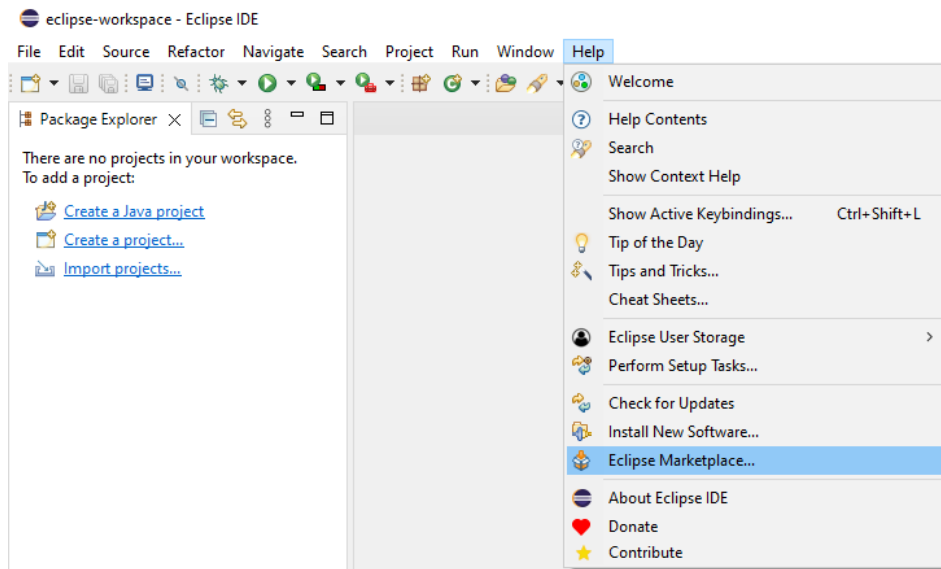


JUnit wymaga również przeprowadzenia kilku dodatkowych kroków, które należy wykonać podczas instalacji Frameworku. Do poprawnego działania nie są one jednak potrzebne w przypadku korzystania z platformy Eclipse. Poniżej link z dodatkowymi informacjami na temat konfiguracji i instalacji „JUNIT_HOME”:

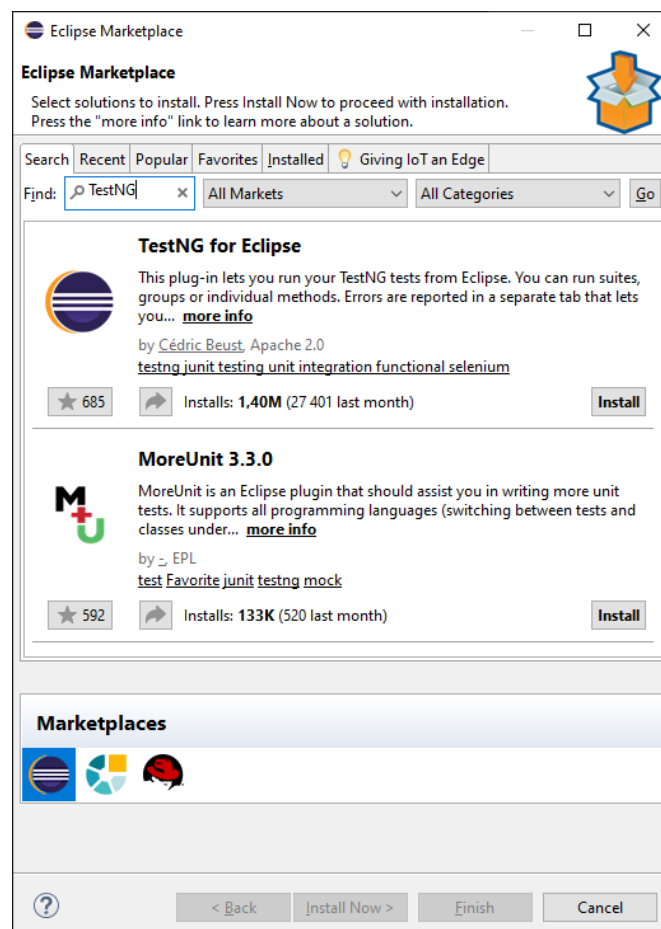
<https://www.softwaretestinghelp.com/download-and-install-junit>

2.3. Instalacja Frameworku TestNG

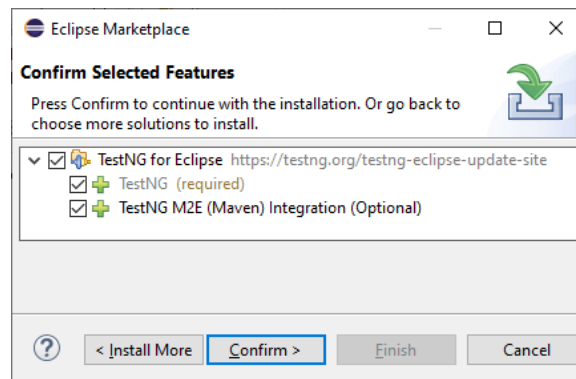
W celu dodania Frameworku TestNG do projektu postępujemy kolejno według instrukcji. TestNG należy zainstalować poprzez wybranie zakładki „Help”, a następnie „Eclipse Marketplace...”.



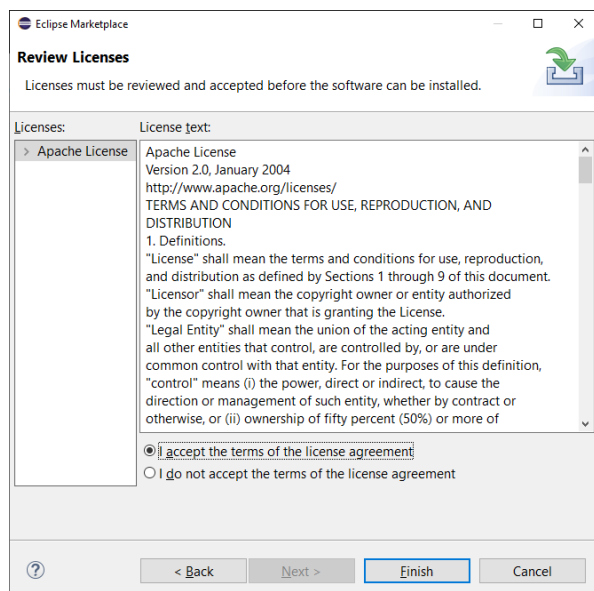
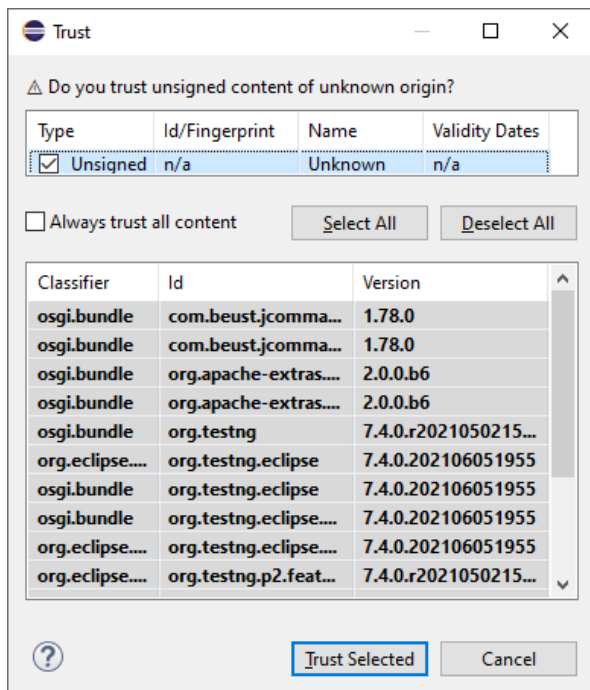
W nowym oknie w sekcji wyszukiwania (zakładka „Search”) wpisujemy „TestNG” („Find:”). Po znalezieniu Frameworku wybieramy przycisk „Install”.



Najlepiej jest zostawić domyślne ustawienia podczas instalacji, klikamy „Confirm >”, a następnie „Finish”.



W trakcie instalacji mogą pojawić się dodatkowe okna z akceptacją licencji czy okno potwierdzające zaufanie dla bibliotek TestNG.



Aby przejść dalej należy wybrać „Trust Selected”. W przypadku licencji należy ją przeczytać, zapoznać się i zaakceptować aby przejść dalej.

Po instalacji należy dodać bibliotekę TestNG do projektu, analogicznie jak w przypadku JUnit.

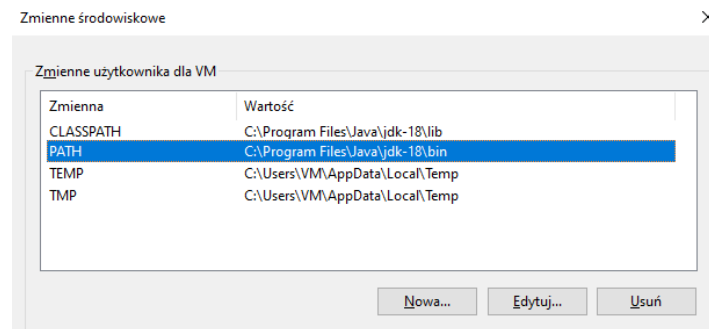
Więcej na temat instalacji i konfiguracji znajdują się na stronie w poniższym linku.

<https://www.lambdatest.com/blog/how-to-install-testng-in-eclipse-step-by-step-guide>

2.4. Instalacja Frameworku Mockito

Przed przystąpieniem do instalacji należy zweryfikować czy mamy już zainstalowaną Javę JDK minimum w wersji 1.5.

Należy również ustawić zmienną środowiskową „JAVA_HOME” jeśli jeszcze tego nie dokonaliśmy (np., podczas instalacji JUnit), tak aby wskazywała lokalizację instalacji oprogramowania Java. Robi się to w ustawieniach zaawansowanych -> „Zmienne środowiskowe...”.



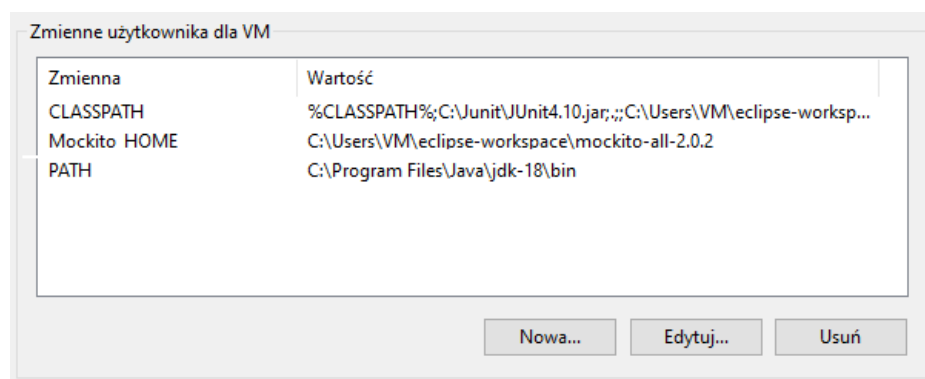
Kolejnym etapem jest pobranie najnowszego repozytorium Maven - Mockito-All. Aktualnie dostępna wersja to „mockito-all-2.0.2-beta.jar”. Link do strony poniżej:

<https://mvnrepository.com/artifact/org.mockito/mockito-all/2.0.2-beta>

Framework dodajemy do projektu: „Java Build Path” -> „Libraries”. Zaznaczamy „Classpath”, „Add External JARs...” -> zaznaczamy „mockito-all-2.0.2-beta.jar”.

Analogicznie jak w przypadku ustawiania zmiennej środowiskowej dla Java JDK, teraz należy zrobić to samo z Mockito dla „Mockito_Home” – z tym, że ustawiamy to dla katalogu, w którym rozpakowaliśmy uprzednio pobrany plik jar.

Należy również ustawić zmienną środowiskową „CLASSPATH”, w której należy wskazać plik jar Mockito w katalogu:



Pozostała konfiguracja dotyczy już dodania frameworku JUnit oraz ustawienie jej zmiennych środowiskowych.

Więcej na temat instalacji i konfiguracji znajdują się na stronie w poniższym linku.

https://www.tutorialspoint.com/mockito/mockito_environment.htm

2.5. Instalacja Frameworku EasyMock

W celu dodania Frameworku EasyMock do projektu postępujemy kolejno według instrukcji. Pierwszym krokiem jest pobranie biblioteki ze strony producenta:

<https://easymock.org>

EASYMOCK

Easy mocking. Better testing.

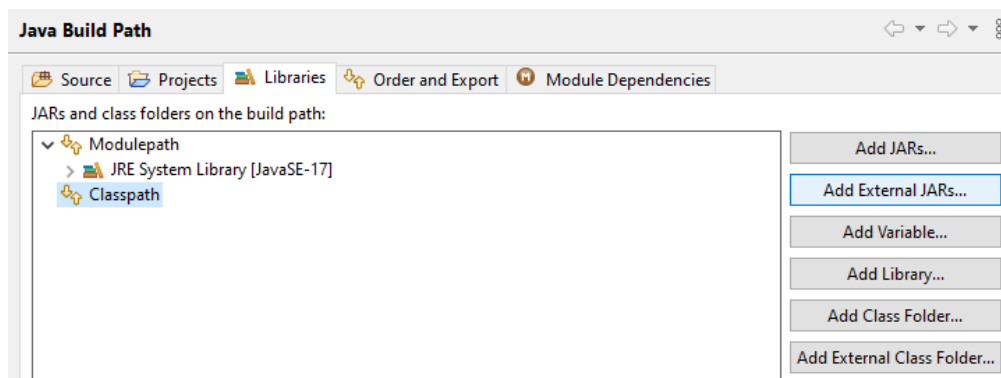
Getting started

Download (v4.3)

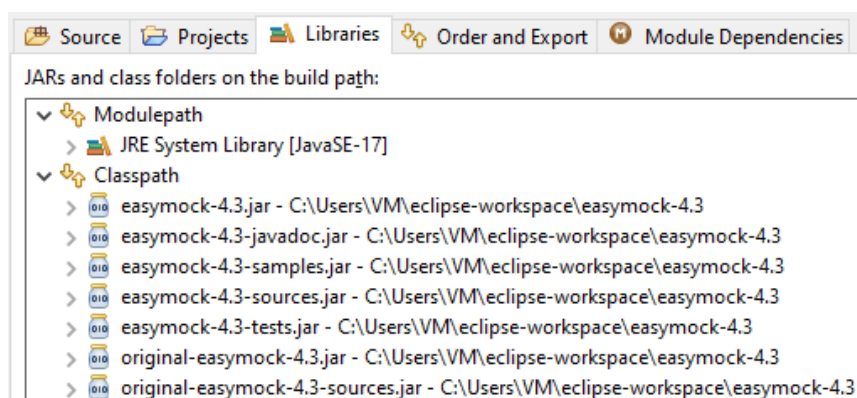
Wybieramy przycisk „Download (v4.3)” - w tym momencie powinno nastąpić automatyczne pobieranie. Po pobraniu należy wypakować zawartość i dodać ją do projektu.

Początek jest analogiczny jak w przypadku JUnit: Wybieramy główny katalog z projektem (w tym przypadku jest to „Mockito_EasyMock”, następnie prawym przyciskiem myszy rozwijamy menu, z którego wybieramy „Build Path”, a następnie „Configure Build Path...”.

W nowym oknie wybieramy „Java Build Path” następnie zakładkę „Libraries”. Zaznaczamy „Classpath”, następnie należy wybrać opcję „Add External JARs...”.



Wybieramy potrzebne składniki (najlepiej wszystkie) i dodajemy do projektu



Na koniec zapisujemy zmiany poprzez wybranie opcji „Apply and Close”.

Może zaistnieć potrzeba dodatnia dodatkowo, np. JUnit do projektu.

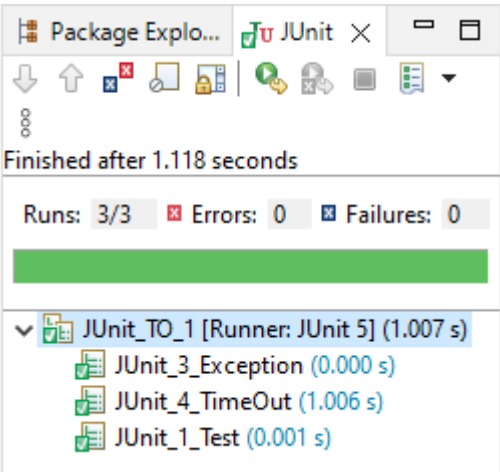
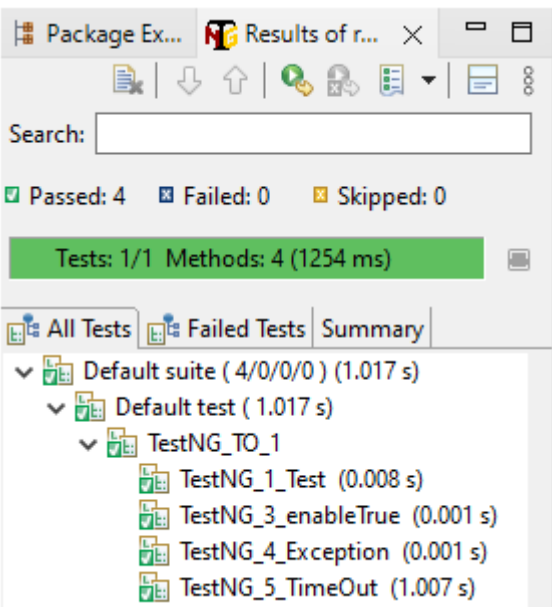
Przykładowo: w przypadku adnotacji: `@RunWith(EasyMockRunner.class)` gdzie `@RunWith` jest adnotacją JUnit, a pozostała składnia to EasyMock.

3. Porównanie Frameworków: JUnit z TestNG

JUnit	TestNG
Pisanie Testów	
JUnit oraz TestNG opierają się na odpowiednio przygotowanych Asercjach w Javie. Testowanie staje się trudnym procesem wraz z powiększaniem się i rozwojem projektu, dlatego sensowne jest używanie bibliotek do zarządzania testami. JUnit oraz TestNG są zgodne z konwencjami xUnit.	
Uruchomienie testów	
Obydwie biblioteki mają wsparcie CLI (działające przez ANT i wtyki IDE).	
Możliwość skorzystania z JDT (Eclipse Java Development Tools).	Brak.
Adnotacja <code>@RunWith</code> pozwala użyć własnego runnera.	Pomijanie domyślnego runnera nie jest takie proste.
Automatyzacja	
Są dostępne narzędzia oraz wtyczki wspomagające tworzenie raportów dla obu bibliotek. Obydwie biblioteki mają porównywalne możliwości w tym zakresie.	
Raportowanie wyników	
Raport jest niedostępny, a wszystkie dane są dostępne tylko za pośrednictwem pliku XML. Potrzeba dodawania wtyczki dla danego IDE.	Generowane domyślnie w formacie html z tabelami zawierające informacje o testach (udane, nieudane, pominięte, czasy testów, itp.). Dodatkowo export do pliku XML.
Testy sparametryzowane	
Użycie różnych kombinacji kilku argumentów (łatwy dostęp do listy parametrów).	Może obsługiwać XML do podawania danych, plików CSV lub tekstowych.
Testy równoległe	
Nie oferuje tak prostego rozwiązania.	Prosta w użyciu adnotacja: <code>@Test (threadPoolSize = 10, invocationCount = 20)</code> .
Potrzeba pisania niestandardową metodę runner (podawanie wielokrotnie tych samych parametrów testowych).	Możliwość uruchomienia całych pakietów testów równoległe.
Zależności	
Brak takiej funkcji. Możliwość emulowania za pomocą założeń (ryzyko nieudania założenia ze skutkiem zignorowania i pominięcia testu).	Możliwość zadeklarowania zależności między testami i pomijanie ich.
Grupy	
Możliwość tworzenia grup testowych i uruchomienie kodu przed/po grupach przypadków testowych. Pojedyncze testy mogą należeć do wielu grup, a te działać w różnych kontekstach.	

Istnieje podobna funkcja, ale brakuje jej adnotacji: @BeforeGroups i @AfterGroups. W wersji 5 wprowadzono @Tag, który ma podobne zastosowanie.	Oferuje dodatkowe adnotacje: @BeforeGroups, @AfterGroups.
Dostępność i społeczność	
Obydwie biblioteki oferują wiele dokumentacji, zarówno na głównych stronach bibliotek, jak i stron, a także dużą ilość prezentacji i wideo, dzięki czemu nie ma problemu ze znalezieniem pomocy w przypadku kłopotu.	
Dłuższa historia frameworku i większa baza użytkowników. Względem drugiego można stwierdzić, że zdefiniował standard testów jednostkowych.	Wiele dostępnych serwisów oraz grup internetowych zrzeszających wielu użytkowników.
Wnioski	
Zarówno JUnit oraz TestNG wspomagają pracę przy automatyzacji testów. Obie biblioteki mogą sprawdzić się lepiej lub gorzej. Wybór odpowiedniego frameworku zależy od tego co potrzebujemy w projekcie: czy jest potrzeba wykonywania równoległych testów, uzyskiwania bardziej dokładnych raportów z wykonywanych testów czy może więcej informacji dostępnych w sieci.	

Poniżej przykładowe zdjęcia interfejsów graficznych ukazujący wykonanie testów w przypadku obu bibliotek z wykorzystaniem platformy Eclipse.

JUnit	TestNG
	

3.1. Podstawowe adnotacje w JUnit [4]

JUnit	
@Test	Oznacza, że metoda jest metodą testową. W przeciwieństwie do adnotacji @Test JUnit 4, adnotacja ta nie deklaruje żadnych atrybutów, ponieważ rozszerzenia testów w JUnit Jupiter działają w oparciu o własne dedykowane adnotacje. Takie metody są dziedziczone, chyba że zostaną nadpisane.
@BeforeEach	Oznacza, że metoda z adnotacjami powinna być wykonywana przed każdą @Test metodą, @RepeatedTest, @ParameterizedTest lub @TestFactory w bieżącej klasie; analogiczne do JUnit 4's @Before. Takie metody są dziedziczone , chyba że zostaną nadpisane.
@AfterEach	Wskazuje, że metoda z adnotacjami powinna być wykonywana po każdej @Test metodzie, @RepeatedTest, @ParameterizedTest lub @TestFactory w bieżącej klasie; analogiczne do JUnit 4 @After. Takie metody są dziedziczone , chyba że zostaną nadpisane.
@BeforeAll	Oznacza, że metoda z adnotacjami powinna być wykonywana przed wszystkimi @Test metodami, @RepeatedTest, @ParameterizedTest i @TestFactory w bieżącej klasie; analogiczne do JUnit 4 @BeforeClass. Takie metody są dziedziczone (chyba że są ukryte lub przesłonięte) i muszą być (chyba że jest używany cykl życia instancji testowej static „na klasę”).
@AfterAll	Oznacza, że metoda z adnotacjami powinna zostać wykonana po wszystkich @Test metodach, @RepeatedTest, @ParameterizedTest i @TestFactory w bieżącej klasie; analogiczne do JUnit 4 @AfterClass. Takie metody są dziedziczone (chyba że są ukryte lub przesłonięte) i muszą być (chyba że jest używany cykl życia instancji testowej static „na klasę”).
@Tag	Używany do deklarowania tagów do testów filtrujących, na poziomie klasy lub metody; analogicznie do grup testowych w TestNG lub Categories w JUnit 4. Takie adnotacje są dziedziczone na poziomie klasy, ale nie na poziomie metody.
@RepeatedTest	Oznacza, że metoda jest szablonem testu dla powtórnego testu. Takie metody są dziedziczone , chyba że zostaną nadpisane.
@ParameterizedTest	Oznacza, że metoda jest testem sparametryzowanym. Takie metody są dziedziczone, chyba że zostaną nadpisane.
@Disabled	Służy do wyłączania klasy testowej lub metody testowej; analogiczne do JUnit 4 @Ignore. Takie adnotacje nie są dziedziczone.

Pełna dokumentacja opisująca wszystkie adnotacje dostępna jest pod linkiem:

<http://junit.org/junit5/docs/current/user-guide>

3.2. Przykłady adnotacji JUnit

Adnotacja @Test:

```
@Test
public void JUnit_1_Test()
{
    System.out.println("Test-JUnit");
}
```

Adnotacja @RepeatedTest - test zostanie powtórzony 500 razy:

```
@RepeatedTest(500)
public void JUnit_2_Test()
{
    System.out.println("Powtorzenie testu: 500-JUnit");
}
```

Adnotacja @Ignore - test zostanie pominięty/zignorowany:

```
@Ignore
public void JUnit_3_Ignore()
{
    System.out.println("Test zostanie zignorowana");
}
```

Adnotacja @Test(expected=ArithmeticException.class) - testowanie wyjątku:

```
@Test(expected=ArithmeticException.class)
public void JUnit_4_Exception()
{
    int i=1/0;
}
```

Adnotacja @Test(timeout) - ustawienie czasu oczekiwania na wykonanie testu:

```
@Test(timeout=2000)
public void JUnit_5_TimeOut() throws InterruptedException
{
    Thread.sleep(1000);
    System.out.println("Test: timeout");
}
@Test(timeout=1000)
public void JUnit_6_TimeOut() throws InterruptedException
{
    Thread.sleep(2000);
    System.out.println("Test: timeout");
}
```

W pierwszym przypadku („JUnit_5”) test zostanie pomyślnie wykonany, ponieważ timeout oczekiwania jest ustawiony na 2 sekundy, a test wykona się w 1 sekundę.

W drugim przypadku („JUnit_6”) test nie powiedzie się, ponieważ timeout jest ustawiony na 1 sekundę, a test wykona się w 2 sekundy.

Adnotacja `@BeforeClass` - uruchom raz przed dowolną metodą testową w klasie

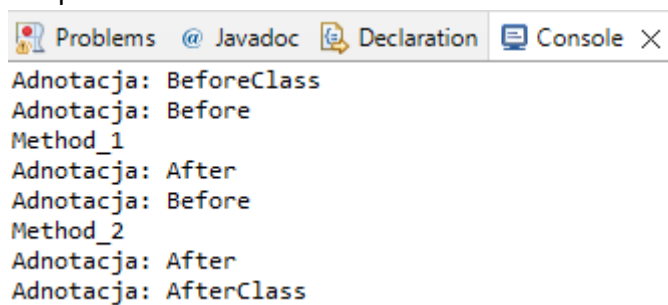
Adnotacja `@AfterClass` - uruchom raz po wykonaniu wszystkich testów w klasie

Adnotacja `@Before` - uruchom przed `@Test`

Adnotacja `@After` - uruchom po `@Test`

```
@BeforeClass
public static void BeforeClass()
{
    System.out.println("Adnotacja: BeforeClass");
}
@AfterClass
public static void AfterClass()
{
    System.out.println("Adnotacja: AfterClass");
}
@Before
public void Before()
{
    System.out.println("Adnotacja: Before");
}
@After
public void After()
{
    System.out.println("Adnotacja: After");
}
@Test
public void Metoda_1()
{
    System.out.println("Method_1");
}
@Test
public void Metoda_2()
{
    System.out.println("Method_2");
}
```

Output:



```
Problems  Javadoc  Declaration  Console X
Adnotacja: BeforeClass
Adnotacja: Before
Method_1
Adnotacja: After
Adnotacja: Before
Method_2
Adnotacja: After
Adnotacja: AfterClass
```

3.3. Podstawowe adnotacje w TestNG [5]

TestNG	
@Test	Oznacza klasę lub metodę jako część testu.
@BeforeSuite	Metoda z adnotacjami zostanie uruchomiona przed uruchomieniem wszystkich testów w tym pakiecie.
@AfterSuite	Metoda z adnotacjami zostanie uruchomiona po zakończeniu wszystkich testów w tym pakiecie.
@BeforeTest	Metoda z adnotacjami zostanie uruchomiona przed uruchomieniem jakiegokolwiek metody testowej należącej do klas wewnątrz znacznika <test>.
@AfterTest	Metoda z adnotacjami zostanie uruchomiona po uruchomieniu wszystkich metod testowych należących do klas wewnątrz znacznika <test>.
@BeforeGroups	Lista grup, które wcześniej uruchomi ta metoda konfiguracji. Gwarantuje się, że ta metoda zostanie uruchomiona na krótko przed wywołaniem pierwszej metody testowej należącej do dowolnej z tych grup.
@AfterGroups	Lista grup, po których będzie uruchamiana ta metoda konfiguracji. Gwarantuje się, że ta metoda zostanie uruchomiona wkrótce po wywołaniu ostatniej metody testowej należącej do dowolnej z tych grup.
@BeforeClass	Metoda z adnotacjami zostanie uruchomiona przed wywołaniem pierwszej metody testowej w bieżącej klasie.
@AfterClass	Metoda z adnotacjami zostanie uruchomiona po uruchomieniu wszystkich metod testowych w bieżącej klasie.
@BeforeMethod	Metoda z adnotacjami zostanie uruchomiona przed każdą metodą testową.
@AfterMethod	Metoda z adnotacjami zostanie uruchomiona po każdej metodzie testowej.

Zachowanie adnotacji w superklasie klasy TestNG.

Powyższe adnotacje będą również honorowane (dziedziczone) po umieszczeniu w nadklasie klasy TestNG. Jest to przydatne na przykład do scentralizowania konfiguracji testów dla wielu klas testowych we wspólnej superklasie.

W takim przypadku TestNG gwarantuje, że metody „@Before” są wykonywane w kolejności dziedziczenia (najpierw najwyższa nadklasa, a następnie w dół łańcucha dziedziczenia), a metody „@After” w odwrotnej kolejności (w górę łańcucha dziedziczenia).

Pełna dokumentacja opisująca wszystkie adnotacje dostępna jest pod linkiem:

<https://testng.org/doc/documentation-main.html>

3.4. Przykłady adnotacji TestNG

Adnotacja @Test:

```
@Test
public void TestNG_1_Test()
{
    System.out.println("Test-TestNG");
}
```

Adnotacja @Test(invocationCount) - test zostanie powtórzony 500 razy:

```
@Test(invocationCount=500)
public void TestNG_2_Test()
{
    System.out.println("Powtorzenie testu: 500-TestNG");
}
```

Adnotacja @Test(enabled) - test zostanie pominięty/zignorowany:

```
@Test(enabled=false)
public void TestNG_3_enableFalse()
{
    System.out.println("Test zostanie zignorowana");
}
```

Adnotacja @Test(expectedExceptions=ArithmeticException.class) - testowanie wyjątku:

```
@Test(expectedExceptions=ArithmeticException.class)
public void TestNG_4_Exception()
{
    int i=1/0;
}
```

Adnotacja @Test(timeOut) - ustawienie czasu oczekiwania na wykonanie testu:

```
@Test(timeOut=2000)
public void TestNG_5_TimeOut() throws InterruptedException
{
    Thread.sleep(1000);
    System.out.println("Test: timeOut");
}
@Test(timeOut=1000)
public void TestNG_6_TimeOut() throws InterruptedException
{
    Thread.sleep(2000);
    System.out.println("Test: timeOut");
}
```

W pierwszym przypadku („TestNG_5”) test zostanie pomyślnie wykonany, ponieważ timeout oczekiwania jest ustawiony na 2 sekundy, a test wykona się w 1 sekundę.

W drugim przypadku („TestNG_6”) test nie powiedzie się, ponieważ timeout jest ustawiony na 1 sekundę, a test wykona się w 2 sekundy.

Poniżej znajduje się kolejność, w jakiej zostaną wykonane metody z adnotacjami:

@BeforeSuite - metoda z adnotacjami zostanie uruchomiona przed uruchomieniem wszystkich testów w tym pakiecie.

@BeforeTest - metoda z adnotacjami zostanie uruchomiona przed uruchomieniem jakiegokolwiek metody testowej należącej do klas wewnątrz znacznika Test.

@BeforeClass - metoda z adnotacjami zostanie uruchomiona przed wywołaniem pierwszej metody testowej w bieżącej klasie.

@BeforeMethod - opisana metoda zostanie uruchomiona przed uruchomieniem wszystkich metod testowych w bieżącej klasie.

@AfterMethod - opisana metoda zostanie uruchomiona po każdej metodzie testowej.

@AfterClass - metoda z adnotacjami zostanie uruchomiona po uruchomieniu wszystkich metod testowych w bieżącej klasie.

@AfterTest - metoda z adnotacjami zostanie uruchomiona po uruchomieniu wszystkich metod testowych należących do klas wewnątrz znacznika Test.

@AfterSuite - metoda z adnotacjami zostanie uruchomiona po zakończeniu wszystkich testów w tym pakiecie.

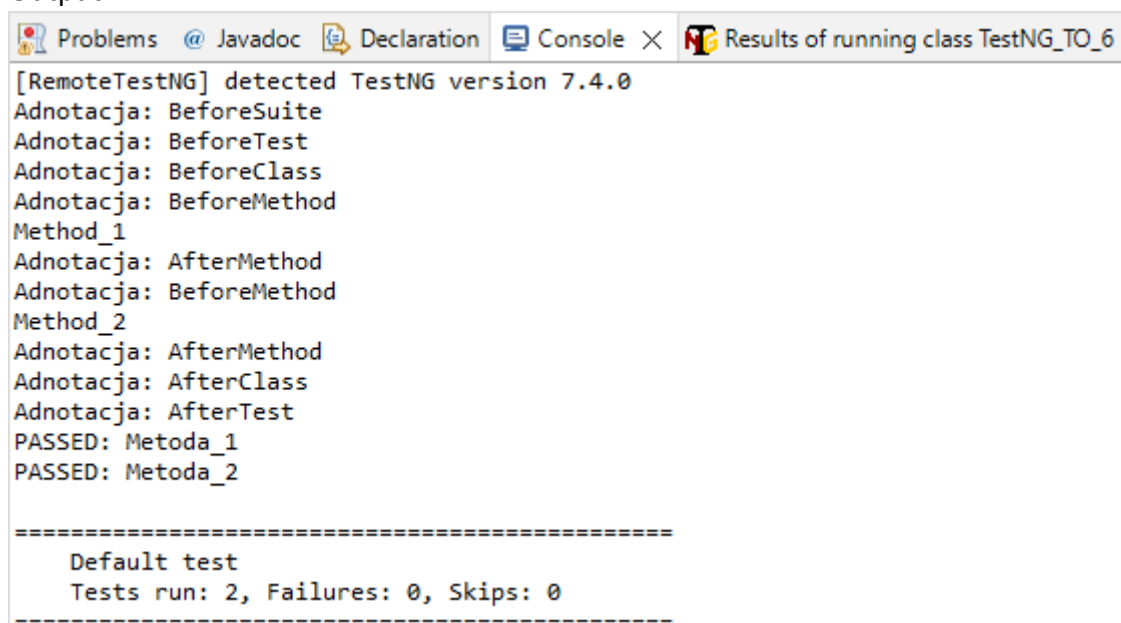
```
@Test
public void Metoda_1()
{
    System.out.println("Method_1");
}
@Test
public void Metoda_2()
{
    System.out.println("Method_2");
}
@BeforeSuite
public void BeforeSuite()
{
    System.out.println("Adnotacja: BeforeSuite");
}
@AfterSuite
public void AfterSuite()
{
    System.out.println("Adnotacja: AfterSuite");
}
@BeforeTest
public void BeforeTest()
{
    System.out.println("Adnotacja: BeforeTest");
}
@AfterTest
public void AfterTest()
{
    System.out.println("Adnotacja: AfterTest");
}
@BeforeClass
public void BeforeClass()
{
    System.out.println("Adnotacja: BeforeClass");
}
```

```

@AfterClass
public void AfterClass()
{
    System.out.println("Adnotacja: AfterClass");
}
@BeforeMethod
public void BeforeMethod()
{
    System.out.println("Adnotacja: BeforeMethod");
}
@AfterMethod
public void AfterMethod()
{
    System.out.println("Adnotacja: AfterMethod");
}

```

Output:



```

[RemoteTestNG] detected TestNG version 7.4.0
Adnotacja: BeforeSuite
Adnotacja: BeforeTest
Adnotacja: BeforeClass
Adnotacja: BeforeMethod
Method_1
Adnotacja: AfterMethod
Adnotacja: BeforeMethod
Method_2
Adnotacja: AfterMethod
Adnotacja: AfterClass
Adnotacja: AfterTest
PASSED: Metoda_1
PASSED: Metoda_2

=====
Default test
Tests run: 2, Failures: 0, Skips: 0
=====

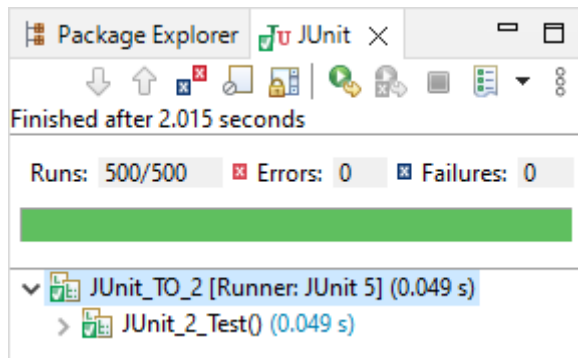
```

3.5. Porównanie adnotacji JUnit i TestNG

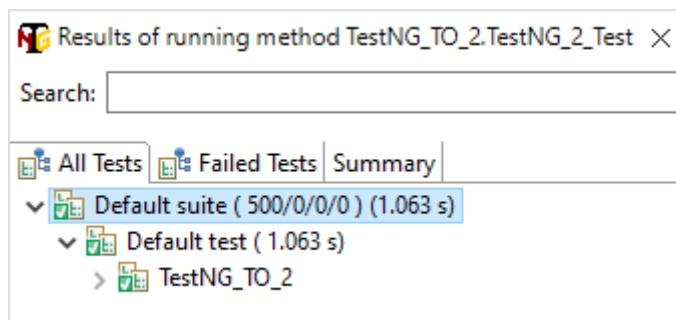
Opis	JUnit 4	TestNG
Adnotacja testowa	@Test	@Test
Wykonuje przed wszystkimi testami w pakiecie	Brak (JUnit 4)	@BeforeSuite
Wykonuje po wszystkich testach w pakiecie	Brak (JUnit 4)	@AfterSuite
Wykonuje przed uruchomieniem testu	Brak (JUnit 4)	@BeforeTest
Wykonuje po uruchomieniu testu	Brak (JUnit 4)	@AfterTest
Wykonuje przed wywołaniem pierwszej metody testowej należącej do którejkolwiek z tych grup	Brak (JUnit 4)	@BeforeGroups
Uruchomić po ostatniej metodzie testowej, która należy do którejkolwiek z grup tutaj	Brak (JUnit 4)	@AfterGroups
Wykonuje przed wywołaniem pierwszej metody testowej w bieżącej klasie	@BeforeClass	@BeforeClass
Wykonuje po wszystkich metodach testowych w bieżącej klasie	@AfterClass	@AfterClass
Wykonuje przed każdą metodą testową	@Before	@BeforeMethod
Wykonuje po każdej metodzie badawczej	@After	@AfterMethod
Adnotacja - ignorowanie testu	@Ignore	@Test(enable=false)
Adnotacja - końca czasu	@Test(timeout=100)	@Test(timeOut=100)
Adnotacja - wyjątki	@Test(expected=ArithmeticException.class)	@Test(expectedExceptions=ArithmeticException.class)
Adnotacja – powtórzenia testu	Brak (JUnit 4) Wprowadzony w JUnit 5 @RepeatedTest(500)	@Test(invocationCount=500)

Test powtórzeń dla JUnit i TestNG - porównanie czasów wykonania 500x powtórzeń wykonania testu. Test był wykonany na komputerze z ograniczoną pamięcią RAM do 1GB.

JUnit i adnotacja: @RepeatedTest(500)



TestNG i adnotacja: @Test(invocationCount=500)



Z powyższego testu można zaobserwować, że TestNG wykonał 500 powtórzeń w 1.063 sekundy, podczas gdy JUnit wykonał 500 powtórzeń w 2.015 sekund. Oznacza to, że przy dużej ilości testów do przeprowadzenia i ograniczonych zasobach w postaci sprzętu, TestNG ma niemal dwa razy lepszą wydajność. W przypadku gdy ilość testów będzie sięgała kilku tysięcy czas może się znacznie wydłużyć.

4. Porównanie Frameworków: Mockito z EasyMock

Mockito	EasyMock
Pisanie Testów	
Możliwość stosowania wraz z innymi frameworkami testowymi, takimi jak JUnit czy TestNG	
Rzucanie wyjątków	
Rzucanie wyjątków można wykpić za pomocą <code>.thenThrow(ExceptionClass.class)</code> po wywołaniu metody <code>Mockito.when(mock.method(args))</code> .	Rzucanie wyjątków można zakpić za pomocą <code>.andThrow(new ExceptionClass())</code> po wywołaniu metody <code>EasyMock.expect(..)</code> .
Wsparcie szpiegów	
Wspiera mocki jak i szpiegów.	Obsługuje tylko mocki. Nie obsługuje szpiegów.
Weryfikacja połączeń	
<code>Mockito.verify(mock).method(args)</code> - do weryfikowania wywołań makiety.	<code>EasyMock.verify(mock)</code> - służy do weryfikowania wywołań makiety, ale metoda używana zawsze po wywołaniu metody <code>EasyMock.replay(mock)</code> . Konieczność wywoływania powtórki za każdym razem.
Wyśmiewane wywołania metod	
<code>Mockito.when(mock.method(args)).thenReturn(value)</code> - do naśladowania wywołań metod.	<code>EasyMock.expect(mock.method(args)).andReturn(Value)</code> - służący do naśladowania wywołania metody.
Licencjonowanie	
MIT (Massachusetts Institute of Technology). Daje użytkownikom nieograniczone prawo do używania, kopiowania, modyfikowania i rozpowszechniania (w tym sprzedaży) oryginalnego lub zmodyfikowanego programu w postaci binarnej lub źródłowej. [10]	Apache License. Licencja ta dopuszcza użycie kodu źródłowego zarówno na potrzeby wolnego oprogramowania, jak i własnościowego oprogramowania. [11]
Dostępność i społeczność	
Obydwa frameworki mają duże grono zwolenników i rozwijającej się społeczności. Mockito jest mniej więcej najbardziej znany. Dostęp do różnego rodzaju dokumentacji i poradników jest szeroki w obydwu przypadkach.	
Wnioski	
Najpopularniejszy framework do testowania aplikacji Java.	Pewne ograniczenia sprawiają, że nie jest to framework tak powszechny i popularny jak Mockito.

4.1. Adnotacje Mockito [7]

Mockito	
@Mock	Służy do tworzenia atrap. <ul style="list-style-type: none">- opcjonalnie należy określić, jak ma się zachowywać za pomocą Answer/MockSettings,- when()/given()by określić jak makieta powinna się zachowywać,- jeśli udzielone odpowiedzi nie odpowiadają potrzebom, można napisać je samodzielnie, rozszerzając Answerinterfejs.
@RunWith	Służy do utrzymania czystości testu i usprawnia debugowanie. Inicjuje makiety z adnotacją @Mock.
@InjectMocks	Automatycznie wstrzykuje pola atrapy/szpiegów z adnotacjami @Spy lub @Mock. Pozwala na stenograficzne iniekcje próbne i szpiegowskie oraz minimalizuje powtarzające się próby i zastrzyki szpiegowskie.
@Captor	Umożliwia tworzenie przechwytywacza argumentów na poziomie pola. Jest używany z metodą Verify(), aby uzyskać wartości przekazywane po wywołaniu metody.
@Spy	Częściowe mockowanie, wywoływane są prawdziwe metody, ale nadal można je zweryfikować i zatuszować. Pozwala na tworzenie częściowo pozorowanych obiektów. Innymi słowy, umożliwia skrócone zawijanie instancji pola w obiekt szpiegowski.
Metoda verify()	Sprawdza, czy zostały wywołane metody z określonymi argumentami

4.2. Adnotacje EasyMock [9]

EasyMock	
Mock	Adnotacja do ustawienia w polu, tak aby EasyMockRunner, EasyMockRule lub EasyMockSupport.injectMocks(Object) wstrzykiwały do niego atrapę. Używany do określenia pola, które ma być naśladowane przez EasyMock.
TestSubject	Adnotacja do ustawienia na polu, tak aby EasyMockRunner, EasyMockRule lub EasyMockSupport.injectMocks(Object) wstrzykiwały makiety utworzone za pomocą Mock do jego pól. Używany do określenia obiektu, do którego EasyMock ma wstrzykiwać mocowane obiekty utworzone za pomocą adnotacji @Mock.
Preview	Wskazuje, że element z adnotacjami jest nowy i w razie potrzeby może zostać nieco zmodyfikowany.

4.3. Mockito vs EasyMock [7]

Podobieństwa

- Zezwala na weryfikację na tym samym poziomie co EasyMock (nieoczekiwane wywołania, nadmiarowe wywołania, weryfikacja w kolejności).
- Dopasowywanie argumentów (`anyInt()`, `anyObject()`, itp.).

Różnice

- Brak trybów record/replay - nie są potrzebne. Są tylko 2 rzeczy, które można zrobić z mockami Mockito - **verify** lub **stub**. Stubbing następuje przed wykonaniem i późniejszą weryfikacją.
- Wszystkie mocki są *nice*. Chociaż makiety są fajne, można je zweryfikować ściśle i wykryć niechcianą interakcję.
- Jawny język dla lepszej czytelności: `verify()` i `when()` kontra mieszanka `expect(mock.foo())` i `mock.foo()` - zwykle wywołanie metody bez oczekiwania.
- Uproszczony model skrótu - metody skrótu są odtwarzane przez cały czas z wartością skrótu, niezależnie od tego, ile razy są wywoływane. Działa dokładnie tak, jak EasyMock `andStubReturn()`, `andStubThrow()`. Można także użyć skrótu z różnymi wartościami zwracanymi dla różnych argumentów (jak w EasyMock).
- Weryfikacja metod skrótowych jest opcjonalna, ponieważ zwykle ważniejsze jest sprawdzenie, czy wartość skrótowna jest używana poprawnie, a nie skąd pochodzi.
- Weryfikacja jest jawna - błędy weryfikacji wskazują wiersz kodu pokazujący, jaka interakcja nie powiodła się.
- Weryfikacja w zamówieniu jest elastyczna i nie wymaga weryfikacji każdej interakcji.
- Dopasowywanie argumentów niestandardowych używa elementów dopasowujących hamcrest, więc można użyć istniejących elementów dopasowujących hamcrest. (EasyMock może również zintegrować się z Hamcrest, chociaż nie jest częścią EasyMock, ale Hamcrest [12]).

EasyMock:

```
List mock = createNiceMock(List.class);

expect(mock.get(0)).andReturn("one");
expect(mock.get(1)).andReturn("two");
mock.clear();

replay(mock);

someCodeThatInteractsWithMock();

verify(mock);
```

Mockito:

```
List mock = mock(List.class);

when(mock.get(0)).thenReturn("one");
when(mock.get(1)).thenReturn("two");

someCodeThatInteractsWithMock();

verify(mock).clear();
```

4.3.1. Weryfikacja w kolejności

EasyMock:

```
Control control = createStrictControl();

List one = control.createMock(List.class);
List two = control.createMock(List.class);

expect(one.add("one")).andReturn(true);
expect(two.add("two")).andReturn(true);

control.replay();

someCodeThatInteractsWithMocks();

control.verify();
```

Mockito:

```
List one = mock(List.class);
List two = mock(List.class);

someCodeThatInteractsWithMocks();

InOrder inOrder = inOrder(one, two);

inOrder.verify(one).add("one");
inOrder.verify(two).add("two");
```

4.3.2. Stubbing metody void

EasyMock:

```
List mock = createNiceMock(List.class);

mock.clear();
expectLastCall().andThrow(new RuntimeException());

replay(mock);
```

Mockito:

```
List mock = mock(List.class);

doThrow(new RuntimeException()).when(mock).clear();
```

4.3.3. Dokładna liczba weryfikacji i dopasowania argumentów

EasyMock:

```
List mock = createNiceMock(List.class);

mock.clear();
expectLastCall().times(3);

expect(mock.add(anyObject())).andReturn(true).atLeastOnce();

replay(mock);

someCodeThatInteractsWithMock();

verify(mock);
```

Mockito:

```
List mock = mock(List.class);

someCodeThatInteractsWithMock();

verify(mock, times(3)).clear();
verify(mock, atLeastOnce()).add(anyObject());
```

4.4. Porównanie kroków Mockito i EasyMock [15]

```
public class MyInjectedService
{
    public Integer getX(String anyString)
    {
        return new Integer (2);
    }
}

public class MyClass
{
    @Autowired
    private MyInjectedService injectedService;
    public String myMethod(String pString)
    {
        return (pString+"="+injectedService.getX("anyStringValue"));
    }
}

public class MyClassTest
{
    @Test
    public void testMyMethod_EasyMock()
    {
        MyClass myClassUnitUnderTest=new MyClass();
        MyInjectedService injectedServiceMock=
            EasyMock.createMock(MyInjectedService.class);
        EasyMock.expect(injectedServiceMock.getX
            (EasyMock.anyObject(String.class))).andReturn(new Integer(1));
        ReflectionTestUtils.setField(myClassUnitUnderTest,"injectedService",
            injectedServiceMock);
        EasyMock.replay(injectedServiceMock);
        String myTestResult=myClassUnitUnderTest.myMethod("my result is");
        assertEquals(new String("my result is=1"),myTestResult);
        EasyMock.verify(injectedServiceMock);
    }

    @Test
    public void testMyMethod_Mockito()
    {
        MyClass myClassUnitUnderTest=new MyClass();
        MyInjectedService injectedServiceMock=
            Mockito.mock(MyInjectedService.class);
        Mockito.when(injectedServiceMock.getX
            (Mockito.anyString())).thenReturn(new Integer(1));
        ReflectionTestUtils.setField(myClassUnitUnderTest,"injectedService",
            injectedServiceMock);
        String myTestResult=myClassUnitUnderTest.myMethod("my result is");
        assertEquals(new String("my result is=1"),myTestResult);
        Mockito.verify(injectedServiceMock);
    }
}
```

EasyMock - 7 podstawowych kroków.

("EasyMock create Mock expects and Return's a Easy replayed Mock"...)

1. Utwórz implementację Unit Under Test Class za pomocą java "new"
2. Użyj `.createMock (MockedClass.class)`, aby zaprojektować wymagane zmienne
3. Użyj `.expect (Class.method).andReturn(Variable(s))` do makiety zwróconych zmiennych MockedClass
4. W razie potrzeby użyj `ReflectionTestUtils.setField()`, aby wstrzyknąć zafałszowane elementy do testowanej jednostki
5. Użyj `.replay()`, aby zresetować atrapę i przygotować ją do użycia
6. Wykonaj `Class.method` testowanej jednostki i użyj tradycyjnego potwierdzenia JUnit dla pożądanых wyników
7. (opcjonalnie) użyj `.verify(MockedClass)`, aby sprawdzić, czy Twoja Mocked Class została faktycznie wywołana.

Mockito - 6 podstawowych kroków.

("The Mockito bird can mock when flying thenReturn's to you to verify"...)

1. Utwórz implementację Unit Under Test Class za pomocą java "new"
2. Użyj `.mock (MockedClass.class)`, aby zaprojektować wymagane zmienne
3. Użyj `.when (Class.method).thenReturn(Variable(s))` do makiety zwróconych zmiennych MockedClass
4. W razie potrzeby użyj `ReflectionTestUtils.setField()`, aby wstrzyknąć zafałszowane elementy do testowanej jednostki
5. Wykonaj `Class.method` testowanej jednostki i użyj tradycyjnego potwierdzenia JUnit dla pożądanых wyników
6. (opcjonalnie) użyj `.verify(MockedClass)`, aby sprawdzić, czy Twoja Mocked Class została faktycznie wywołana.

5. Technologie użyte do wykonania projektu

- Windows 10 Pro, 21H2, 19044.1706
- VMware Workstation 16 Player, Version: 16.2.3 (build-19376536)
- GitHub, <https://github.com>
- Eclipse IDE for Java Developers, Version: 2022-03 (4.23.0), Build id: 20220310-1457, <https://www.eclipse.org>
- Język programowania Java
- Framework JUnit 5, Version: 5.8.1.20211018-1956
- Framework TestNG, Version: 7.4.0.202106051955
- Framework Mockito , Version: 2.0.2 (beta)
- Framework EasyMock, Version: 4.3
- Java 8 Update 321, Build 1.8.0_321-b07 (jre)
- Java JDK-18, 2022-03-22

6. Podsumowanie

- Testy automatyczne mogą być częścią dobrze przemyślanego planu wsparcia jakości wytwarzanego oprogramowania.
- Wybór odpowiedniego środowiska do tworzenia testów zależy głównie od indywidualnych potrzeb do celów projektowych oraz wiedzy o danej bibliotece testowej.
- Wszystkie przetestowane biblioteki mają swoje mocne jak i słabsze strony. Większość z nich mają bardzo zbliżone możliwości. Ograniczenia lub braki można uzupełnić poprzez zastosowanie dodatkowych wtyczek lub narzędzi.
- Testowane biblioteki są profesjonalnymi projektami, które wspomagają pracę przy automatyzacji testów.

7. Literatura

- [1] Marek Grochowski, *Testy jednostkowe*, <https://www.is.umk.pl/~grochu>, [dostęp 01.05.2022].
- [2] Wikipedia, *Eclipse*, <https://pl.wikipedia.org/wiki/Eclipse>, [dostęp 01.05.2022].
- [3] Wikipedia, *Java*, <https://pl.wikipedia.org/wiki/Java>, [dostęp 01.05.2022].
- [4] JUnit, *JUnit 5*, <https://junit.org/junit5>, [dostęp 01.05.2022].
- [5] TestNG, *TestNG*, <https://testng.org>, [dostęp 01.05.2022].
- [6] Wikipedia, *Mockito*, <https://en.wikipedia.org/wiki/Mockito>, [dostęp 01.05.2022].
- [7] Mockito, *Mockito*, <https://site.mockito.org>, [dostęp 01.05.2022].
- [8] Wikipedia, *EasyMock*, <https://en.wikipedia.org/wiki/EasyMock>, [dostęp 01.05.2022].
- [9] EasyMock, *EasyMock*, <https://easymock.org>, [dostęp 01.05.2022].
- [10] Wikipedia, *Licencja MIT*, https://pl.wikipedia.org/wiki/Licencja_MIT, [dostęp 01.05.2022].
- [11] Wikipedia, *Apache License*, https://pl.wikipedia.org/wiki/Apache_License, [dostęp 01.05.2022].
- [12] Hamcrest, *The Hamcrest Tutorial*, <https://code.google.com/archive/p/hamcrest/wikis/Tutorial.wiki>, [dostęp 01.05.2022].
- [13] Tutorialspoint, *Mockito*, <https://www.tutorialspoint.com>, [dostęp 01.05.2022].
- [14] Tutorialspoint, *EasyMock*, <https://www.tutorialspoint.com>, [dostęp 01.05.2022].
- [15] Scott Kramer, *5mQuickTops:MockitoVsEasyMock*, [dostęp 01.05.2022].
- [16] Roy Oshero, *Testy jednostkowe. Świat niezawodnych aplikacji. Wydanie II*, 2014.
- [17] Khorikov Vladimir, *Testy jednostkowe. Zasady, praktyki i wzorce*, 2020.