

POLITECHNIKA ŚWIĘTOKRZYSKA W KIELCACH

Wydział Elektrotechniki, Automatyki i Informatyki

Projekt Technologii Obiektowych

Temat projektu:

41 – Porównanie rozwiązań związanych z testowaniem

Autorzy:

Daniel Szwajkowski, Kacper Jurek

Podział pracy:

Daniel Szwajkowski – testy jednostkowe (JUnit, TestNG, Mockito),

Kacper Jurek – testy e2e (Mockito, Testim, Cucumber).

1. Testy jednostkowe

Test jednostkowy (ang. unit test) to sposób testowania programu, w którym wydzielamy jego mniejszą część, jednostkę i testujemy ją w odosobnieniu. Taką jednostką do testowania może być pojedyncza klasa lub metoda. Testy jednostkowe można pisać bez bibliotek zewnętrznych jednak jest to uciążliwe. Dodatkowo warto używać istniejących bibliotek ponieważ IDE dobrze się z nimi integrują. Test jednostkowy to metoda testująca naszą jednostkę, metodę w innej klasie z dodaną adnotacją @Test. Testy jednostkowe są zazwyczaj automatycznymi testami pisanymi i uruchamianymi przez programistów, aby upewnić się, że sekcja aplikacji (znana jako „jednostka”) spełnia projekt i zachowuje się zgodnie z przeznaczeniem. W programowaniu proceduralnym jednostka może być całym modulem, ale częściej jest to pojedyncza funkcja lub procedura. W programowaniu obiektowym jednostka jest często całym interfejsem, takim jak klasa lub pojedyncza metoda.^[5] Pisząc testy najpierw dla najmniejszych testowalnych jednostek, a następnie złożone zachowania między nimi, można zbudować kompleksowe testy dla złożonych aplikacji. Aby wyizolować problemy, które mogą się pojawić, każdy przypadek testowy należy przetestować niezależnie. Substytuty, takie jak kody metod, próbné obiekty, podróbki i wiązki testowe mogą być używane do wspomagania testowania modułu w izolacji.

Podczas opracowywania programista może zakodować kryteria lub wyniki, o których wiadomo, że są dobre, w teście, aby zweryfikować poprawność jednostki. Podczas wykonywania przypadków testowych frameworki rejestrują testy, które nie spełniają dowolnego kryterium, i raportują je w podsumowaniu. W tym celu najczęściej stosowanym podejściem jest test - funkcja - wartość oczekiwana.

Pisanie i utrzymywanie testów jednostkowych można przyspieszyć przy użyciu testów sparametryzowanych. Umożliwiają one wielokrotne wykonanie jednego testu z różnymi zestawami danych wejściowych, zmniejszając w ten sposób powielanie kodu testowego. W przeciwieństwie do tradycyjnych testów jednostkowych, które zwykle są metodami zamkniętymi i testowymi warunkami niezmiennymi, testy parametryczne przyjmują dowolny zestaw parametrów. Testy parametryczne są obsługiwane przez TestNG, JUnit. Odpowiednie parametry dla testów jednostkowych mogą być dostarczane ręcznie lub w niektórych przypadkach są automatycznie generowane przez framework testowy. W ostatnich latach dodano obsługę pisania potężniejszych (jednostkowych) testów, wykorzystujących koncepcję teorii, przypadków testowych, które wykonują te same kroki, ale przy użyciu danych testowych generowanych w czasie wykonywania, w przeciwieństwie do zwykłych testów parametrycznych, które wykorzystują te same kroki wykonania z zestawami wejściowymi które są wstępnie zdefiniowane.

Zalety:

Celem testów jednostkowych jest wyizolowanie każdej części programu i wykazanie, że poszczególne części są poprawne. Test jednostkowy zapewnia ścisłą, pisemną umowę, którą musi spełnić fragment kodu. W rezultacie daje kilka korzyści.

Testy jednostkowe znajdują problemy na wczesnym etapie cyklu rozwojowego. Obejmuje to zarówno błędy w implementacji programisty, jak i wady lub brakujące części specyfikacji urządzenia. Proces pisania dokładnego zestawu testów zmusza autora do przemyślenia danych wejściowych, wyjściowych i warunków błędów, a tym samym dokładniejszego zdefiniowania pożądanego zachowania jednostki. Koszt znalezienia błędu przed rozpoczęciem kodowania lub przy pierwszym napisaniu kodu jest znacznie niższy niż koszt późniejszego wykrycia, zidentyfikowania i poprawienia błędu. Błędy w wydanym kodzie mogą również powodować kosztowne problemy dla użytkowników końcowych oprogramowania. Kod może być niemożliwy lub trudny do testowania jednostkowego, jeśli jest źle napisany, dlatego testowanie jednostkowe może zmusić programistów do lepszego ustrukturyzowania funkcji i obiektów.

W programowaniu opartym na testach (TDD), które jest często używane zarówno w programowaniu ekstremalnym, jak i scrum, testy jednostkowe są tworzone przed napisaniem samego kodu. Gdy testy zakończą się pomyślnie, ten kod jest uważany za kompletny. Te same testy jednostkowe są często przeprowadzane w odniesieniu do tej funkcji, ponieważ większa baza kodu jest opracowywana w miarę zmiany kodu lub poprzez zautomatyzowany proces z kompilacją. Jeśli testy jednostkowe nie powiodą się, jest to uważane za błąd w zmienionym kodzie lub samych testach. Testy jednostkowe umożliwiają następnie łatwe prześledzenie lokalizacji usterki lub awarii. Ponieważ testy jednostkowe ostrzegają zespół programistów o problemie przed przekazaniem kodu testerom lub klientom, potencjalne problemy są wykrywane na wczesnym etapie procesu tworzenia.

Testy jednostkowe pozwalają programiście na refaktoryzację kodu lub aktualizację bibliotek systemowych w późniejszym terminie i upewnienie się, że moduł nadal działa poprawnie (np. w testach regresyjnych). Procedura polega na napisaniu przypadków testowych dla wszystkich funkcji i metod, tak aby za każdym razem, gdy zmiana powoduje błąd, można go szybko zidentyfikować. Testy jednostkowe wykrywają zmiany, które mogą zerwać umowę projektową.

Testowanie jednostkowe może zmniejszyć niepewność w samych jednostkach i może być stosowane w podejściu oddolnym. Testując najpierw części programu, a następnie sumę jego części, testowanie integracyjne staje się znacznie łatwiejsze.

Testy jednostkowe zapewniają rodzaj żywej dokumentacji systemu. Deweloperzy, którzy chcą dowiedzieć się, jakie funkcje zapewnia jednostka i jak z niej korzystać, mogą przyrzeć się testom jednostkowym, aby uzyskać podstawową wiedzę na temat interfejsu jednostki (API).

Przypadki testów jednostkowych zawierają cechy, które są krytyczne dla powodzenia jednostki. Te cechy mogą wskazywać na właściwe/niewłaściwe użycie jednostki, a także na negatywne zachowania, które mają zostać uwięzione przez jednostkę. Przypadek testu jednostkowego sam w sobie dokumentuje te krytyczne cechy, chociaż wiele środowisk programistycznych nie polega wyłącznie na kodzie dokumentującym produkt w fazie rozwoju.

Gdy oprogramowanie jest tworzone przy użyciu podejścia opartego na testach, połączenie napisania testu jednostkowego w celu określenia interfejsu oraz czynności refaktoryzacji wykonywanych po pomyślnym zakończeniu testu może zastąpić formalny projekt. Każdy test jednostkowy może być postrzegany jako element projektu określający klasy, metody i obserwowalne zachowanie.

Wady:

Testowanie nie wykryje każdego błędu w programie, ponieważ nie może ocenić każdej ścieżki wykonania w żadnym innym programie niż najbardziej trywialne. Ten problem jest nadzbiorem problemu zatrzymania, który jest nierozstrzygnięty. To samo dotyczy testów jednostkowych. Ponadto testowanie jednostkowe z definicji testuje tylko funkcjonalność samych jednostek. W związku z tym nie wykryje błędów integracji ani szerszych błędów na poziomie systemu (takich jak funkcje wykonywane w wielu jednostkach lub niefunkcjonalne obszary testowe, takie jak wydajność). Testy jednostkowe powinny być wykonywane w połączeniu z innymi testami oprogramowania czynności, ponieważ mogą jedynie wskazywać na obecność lub brak określonych błędów; nie mogą udowodnić całkowitego braku błędów. Aby zagwarantować prawidłowe zachowanie dla każdej ścieżki wykonania i każdego możliwego wejścia oraz zapewnić brak błędów, wymagane są inne techniki, a mianowicie zastosowanie metod formalnych do udowodnienia, że składnik oprogramowania nie zachowuje się nieoczekiwanie.

Skomplikowana hierarchia testów jednostkowych nie oznacza testów integracyjnych. Integracja z jednostkami peryferyjnymi powinna być uwzględniana w testach integracyjnych, ale nie w testach jednostkowych. Testowanie integracyjne zazwyczaj nadal w dużym stopniu opiera się na ręcznym testowaniu przez ludzi; Testowanie wysokiego poziomu lub testowanie o zakresie globalnym może być trudne do zautomatyzowania, tak że testowanie ręczne często wydaje się szybsze i tańsze.

Testowanie oprogramowania to problem kombinatoryczny. Na przykład, każda instrukcja decyzji logicznej wymaga co najmniej dwóch testów: jednego z wynikiem „prawda”, a drugiego z wynikiem „fałsz”. W rezultacie na każdy napisany wiersz kodu programiści często potrzebują od 3 do 5 wierszy kodu testowego. To oczywiście wymaga czasu, a jego inwestycja może nie być warta wysiłku. Istnieją problemy, których w ogóle nie da się łatwo przetestować – na przykład te, które są niedeterministyczne lub obejmują wiele wątków.

Kolejnym wyzwaniem związanym z pisananiem testów jednostkowych jest trudność tworzenia realistycznych i użytecznych testów. Konieczne jest stworzenie odpowiednich warunków początkowych, aby testowana część aplikacji zachowywała się jak część całego systemu. Jeśli te warunki początkowe nie zostaną ustawione poprawnie, test nie będzie wykonywał kodu w realistycznym kontekście, co umniejsza wartość i dokładność wyników testów jednostkowych.

Aby uzyskać zamierzone korzyści z testów jednostkowych, w całym procesie tworzenia oprogramowania potrzebna jest rygorystyczna dyscyplina. Niezbędne jest prowadzenie dokładnych zapisów nie tylko przeprowadzonych testów, ale także wszystkich zmian, które zostały wprowadzone w kodzie źródłowym tej lub jakiegokolwiek innej jednostki w oprogramowaniu. Niezbędne jest zastosowanie systemu kontroli wersji. Jeśli nowsza wersja urządzenia nie przejdzie pomyślnie określonego testu, który wcześniej przeszedł, oprogramowanie do kontroli wersji może dostarczyć listę zmian w kodzie źródłowym (jeśli w ogóle), które zostały zastosowane w urządzeniu od tego czasu.

Niezbędne jest również wdrożenie zrównoważonego procesu zapewniającego, że awarie przypadków testowych są regularnie przeglądane i natychmiast usuwane. Jeśli taki proces nie zostanie wdrożony i zakorzeniony w przepływie pracy zespołu, aplikacja nie będzie zsynchronizowana z zestawem testów jednostkowych, zwiększając liczbę fałszywych alarmów i zmniejszając skuteczność zestawu testów.

Testowanie jednostkowe oprogramowania wbudowanego systemu stanowi wyjątkowe wyzwanie: ponieważ oprogramowanie jest opracowywane na innej platformie niż ta, na której będzie ostatecznie działać, nie można łatwo uruchomić programu testowego w rzeczywistym środowisku wdrażania, jak to jest możliwe w przypadku programów komputerowych.

Testy jednostkowe są najłatwiejsze, gdy metoda ma parametry wejściowe i pewne dane wyjściowe. Tworzenie testów jednostkowych nie jest tak łatwe, gdy główną funkcją metody jest interakcja z czymś zewnętrznym w stosunku do aplikacji. Na przykład metoda, która będzie działać z bazą danych, może wymagać utworzenia makiety interakcji z bazą danych, która prawdopodobnie nie będzie tak obszerna, jak rzeczywiste interakcje z bazą danych.

Asercje to metody dostarczone przez bibliotekę JUnit, które pomagają przy testowaniu. Jeżeli metoda zwróci false, asercja `assertTrue` rzuci wyjątek, który przez IDE zostanie zinterpretowany jak test jednostkowy, który pokazuje błąd działania testowanego kodu. Mówimy wówczas, że „test nie przeszedł”. Testy jednostkowe łączymy w klasy z testami, bardzo często nazywamy je tak samo jak klasy, które testujemy dodając do nich `Test` na końcu. Asercje tworzą komunikaty błędów (w trakcie testów jednostkowych), które ułatwiają znalezienie błędu. Komunikaty te są bardziej czytelne niż standardowy wyjątek `AssertionError`. Asercje w bibliotece JUnit to nic innego jak metody statyczne w klasie `Assert`. Poniżej znajduje się kilka najczęściej stosowanych asercji:

- `assertTrue` sprawdza czy przekazany argument to true,
- `assertFalse` sprawdza czy przekazany argument to false,
- `assertNull` sprawdza czy przekazany argument to null,
- `assertNotNull` sprawdza czy przekazany argument nie jest nullem,
- `assertEquals` przyjmuje dwa parametry wartość oczekiwaną i wartość rzeczywistą, jeśli są różne wyrzuca wyjątek,

- `assertNotEquals` przyjmuje dwa parametry wartość oczekiwaną i wartość rzeczywistą, wyrzuci wyjątek jeśli są równe.

W języku Java trzeba importować klasy z innych pakietów, które chcemy użyć w definicji naszej klasy. Poza standardową konstrukcją ze słowem kluczowym `import` istnieją także tak zwane importy statyczne. Import statyczny pozwala na zaimportowanie metody/wszystkich metod statycznych znajdujących się w definicji jakiejś klasy. Czasami zdarza się przetestować pewną sytuację wyjątkową. Nie powinniśmy móc utworzyć instancji klasy z niepoprawnymi argumentami. Wywołanie takiego konstruktora kończyłoby się od razu rzuceniem wyjątku, czyli testem jednostkowym, który nie przeszedł. Testy jednostkowe wymagają pewnego „przygotowania”. Na przykład trzeba utworzyć instancję, która będzie później testowana. Twórcy biblioteki JUnit stworzyli adnotację `@Before`, którą można dodać do metody w klasie z testami. Metoda ta zostanie uruchomiona przed każdym testem jednostkowym. Adnotacja `@Before` jest jedną z czterech adnotacji, które pozwalają na wykonanie fragmentów kodu przed/po testach. Pozostałe trzy to:

- `@After`– metoda z tą adnotacją uruchamiana po każdym teście jednostkowym, pozwala na „posprzątanie” po teście,
- `@AfterClass`– metoda statyczna z tą adnotacją uruchamiana jest raz po uruchomieniu wszystkich testów z danej klasy,
- `@BeforeClass`– metoda statyczna z tą adnotacją uruchamiana jest raz przed uruchomieniem pierwszego testu z danej klasy.

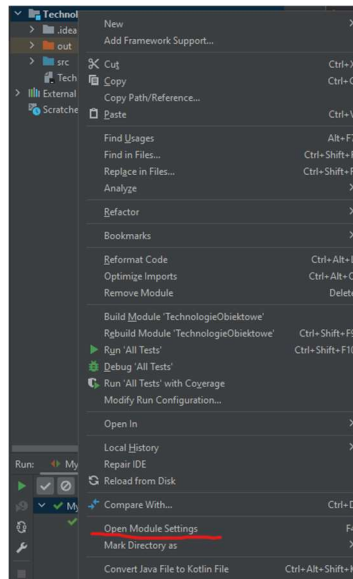
1.1. JUnit

JUnit – narzędzie służące do tworzenia powtarzalnych testów jednostkowych oprogramowania pisanego w języku Java. Cechy JUnit:

- najmniejszą jednostką testowania jest metoda,
- oddzielenie testów od kodu,
- wiele mechanizmów uruchamiania,
- tworzenie raportów,
- integracja z różnymi środowiskami programistycznymi.

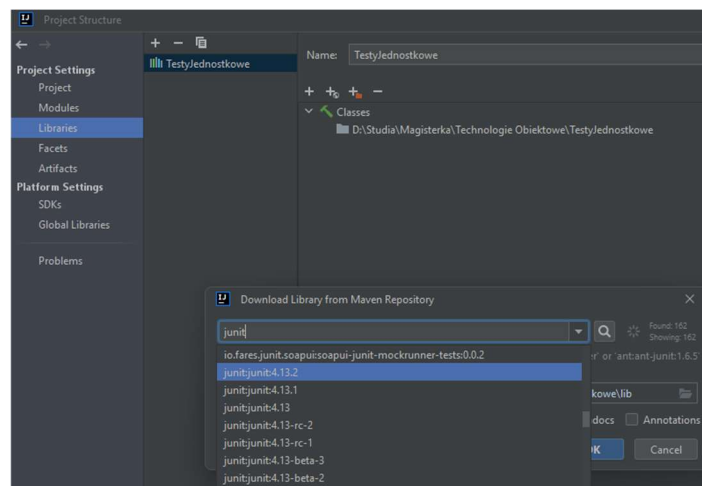
1.1.1. Instalacja JUnit

Aby zainstalować JUnit w środowisku IntelliJ IDEA należy kliknąć prawym przyciskiem myszy na utworzony projekt i wybrać opcję „Open Module Settings”.



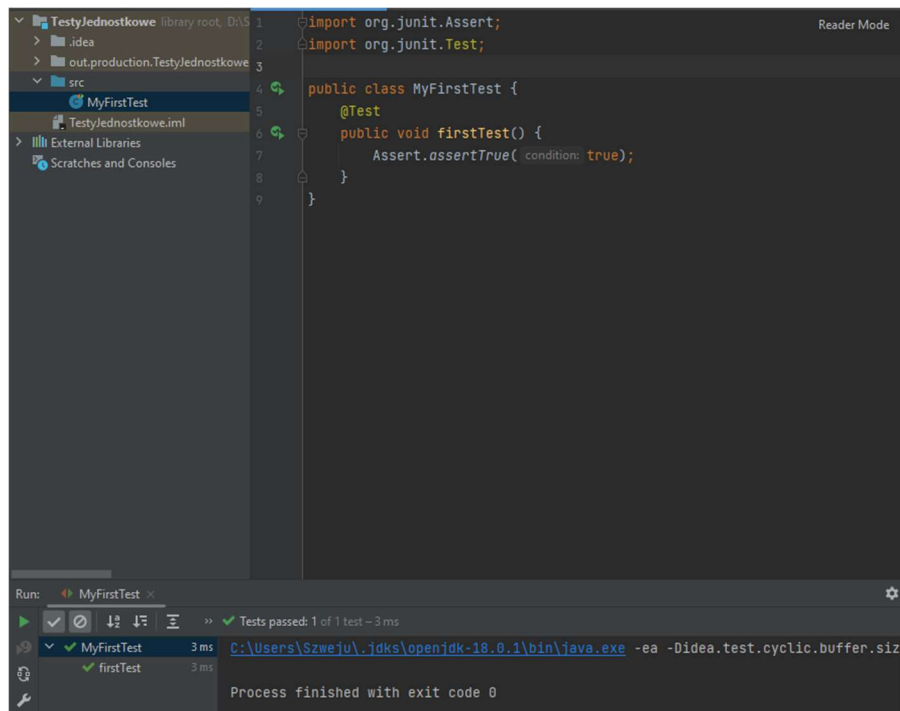
Rys. 1. Otworzenie ustawień projektu

Następnie wybieramy zakładkę „Libraries”, klikamy plus i „From Maven”. Wyszukujemy junit:junit:4.13.2 i zatwierdzamy.



Rys. 2. Dodanie biblioteki JUnit do projektu

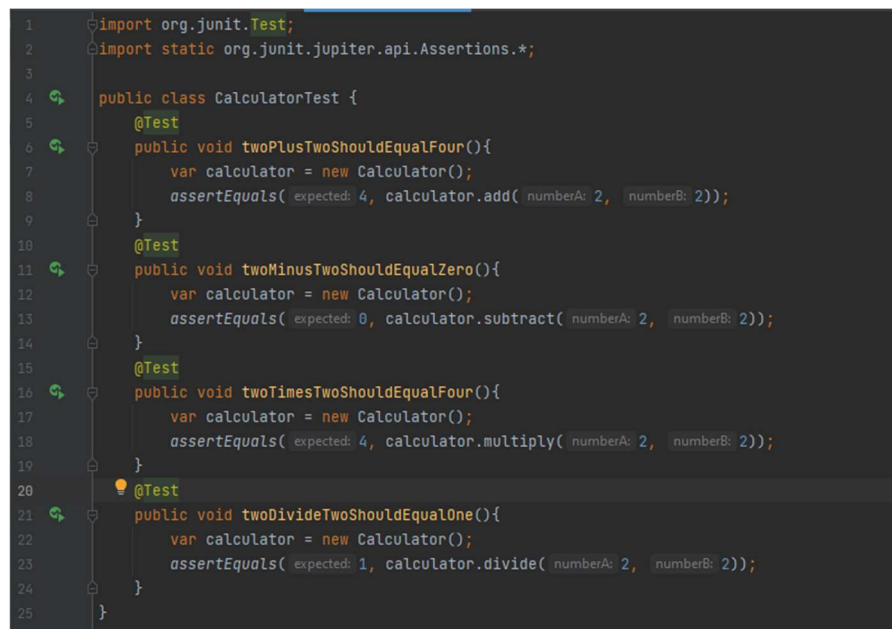
Aby zobaczyć, czy biblioteka została zainstalowana poprawnie należy utworzyć klasę z naszym pierwszym testem. W moim przypadku jest to klasa o nazwie MyFirstTest. Po jej uruchomieniu nie występują żadne błędy, a więc narzędzie działa.



Rys. 3. Pierwszy test sprawdzający poprawne dodanie JUnit do projektu

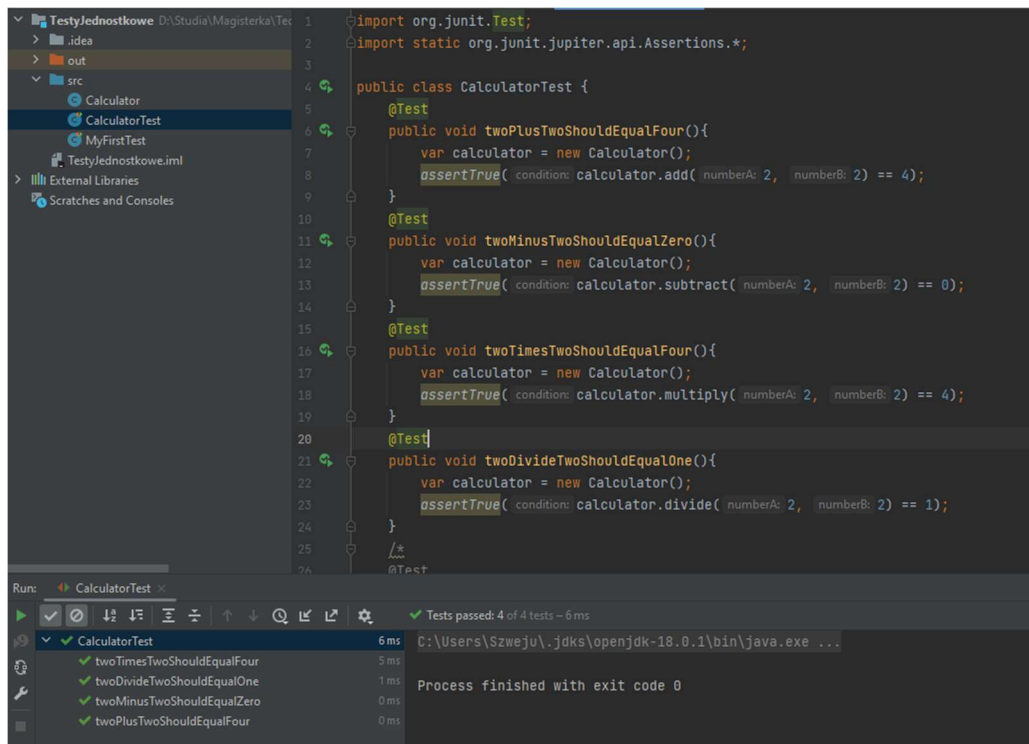
1.1.2. Testowanie

Jako pierwszy został przetestowany prosty kalkulator, który posiada cztery metody: dodawanie, odejmowanie, mnożenie i dzielenie. Aby utworzyć test wystarczy zaznaczyć nazwę klasy oraz wcisnąć kombinację klawiszy ctrl + shift + t. W katalogu została utworzona nowa klasa „CalculatorTest”. Asercja „assertEquals” wymaga dwóch parametrów: wartości oczekiwanej oraz zwracanej przez funkcję.



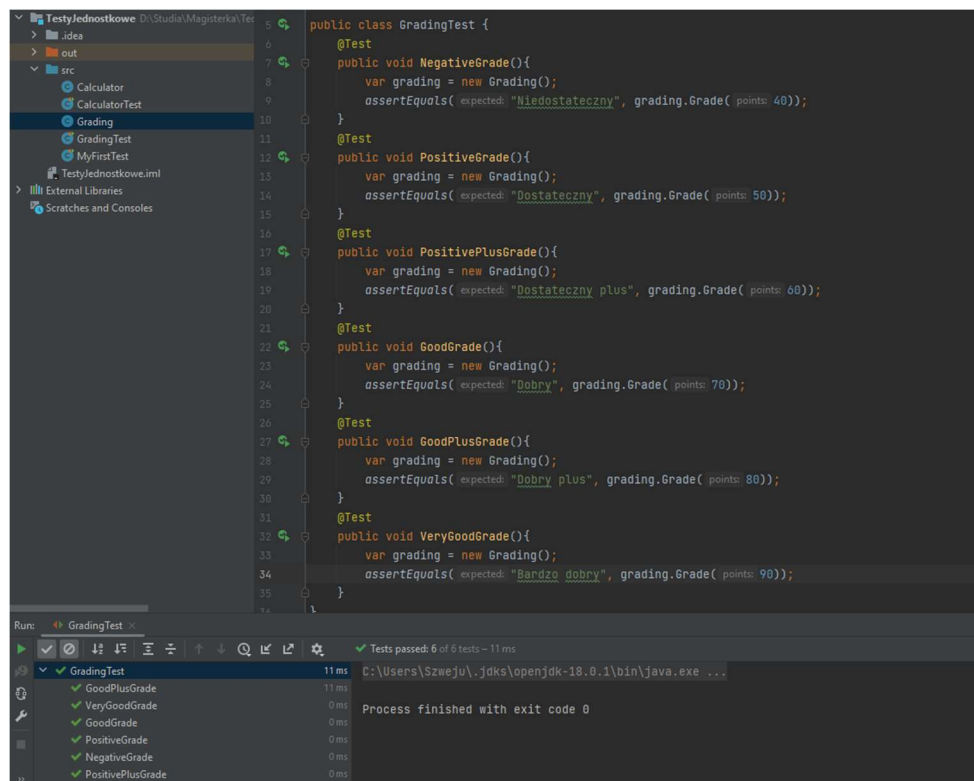
Rys. 4. Test prostego kalkulatora

Ten sam test możemy przeprowadzić używając „assertTrue”. Przyporównujemy w niej wartość zwracaną przez testowaną metodę z pożądaną. Jeżeli zwracana jest wartość „true”, test przebiegł pomyślnie.



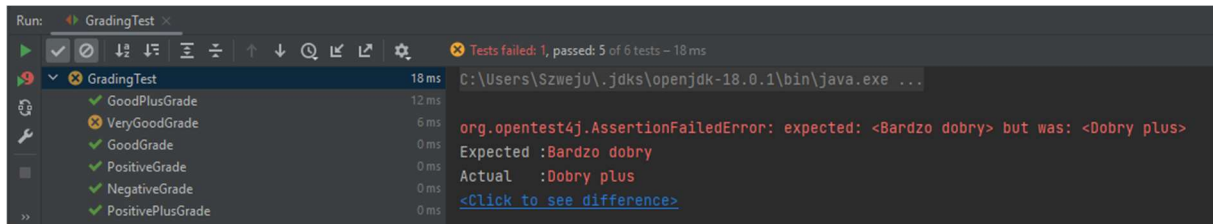
Rys. 5. Test prostego kalkulatora używając „assertTrue”

Kolejny test sprawdzający czy program poprawnie zwraca wartości typu string. Podajemy ilość zdobytych punktów, wtedy otrzymujemy ocenę w postaci słownej.



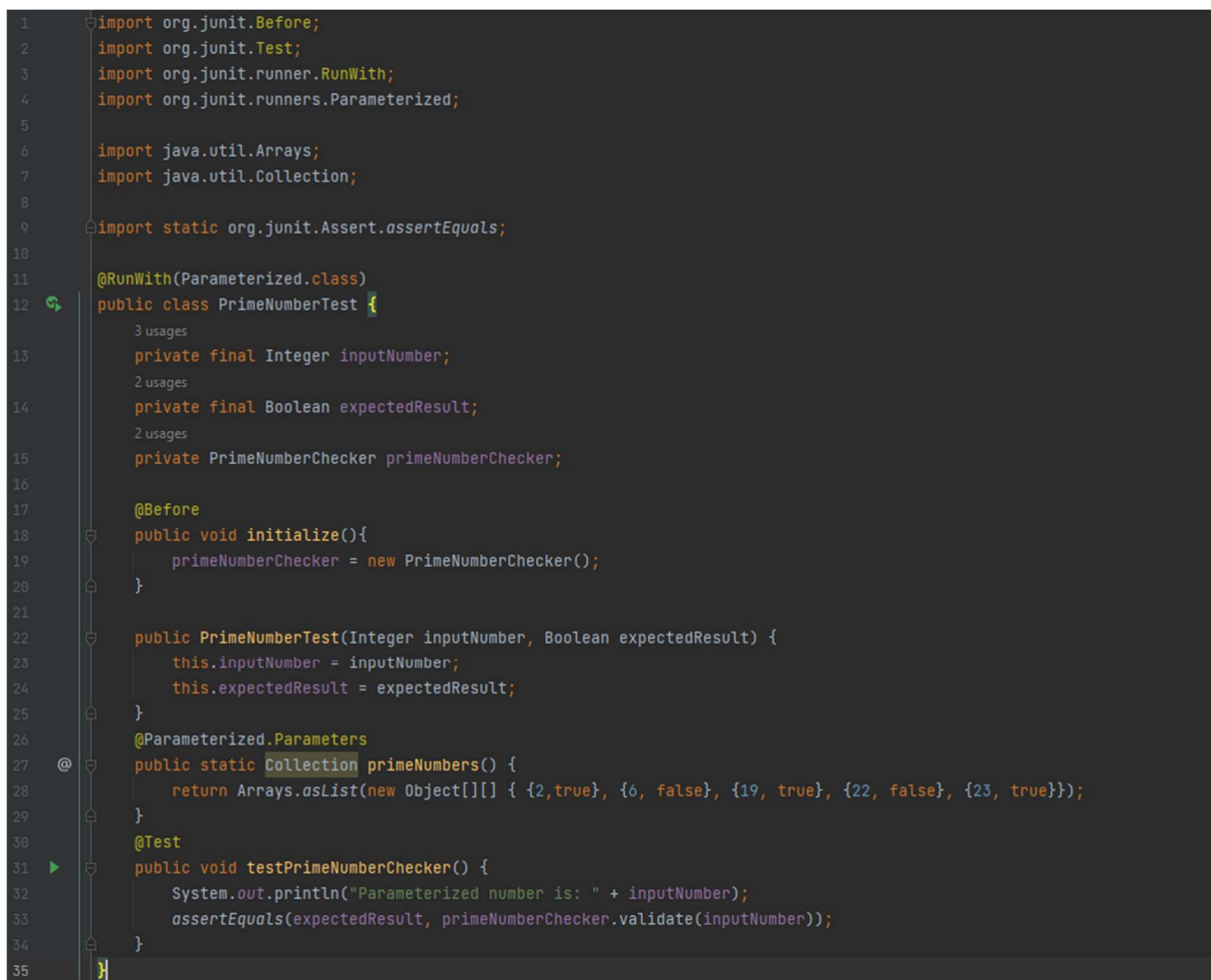
Rys. 6. Test sprawdzający poprawność zwracanych wartości typu string

Jeżeli w kodzie znalazłby się błąd i metoda zwracałaby niepoprawną wartość test nie przejdzie pomyślnie oraz zostanie wypisana otrzymany oraz oczekiwany wynik. Poniżej kod został zmieniony aby dla 90 punktów uczeń otrzymywał ocenę „Dobry plus”, a powinien otrzymać ocenę „Bardzo dobry”.

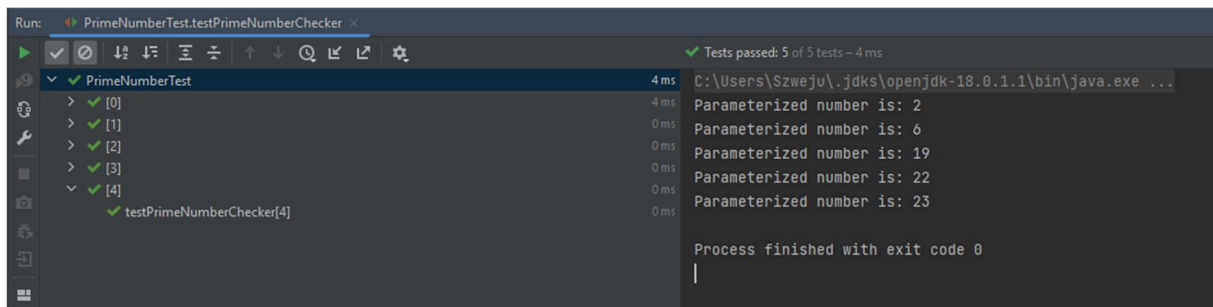


Rys. 7. Niepomyślny test JUnit

W JUnit 5 wprowadzono możliwość tworzenia testów parametryzowanych. Dodanie zależności modułu junit-jupiter-params pozwala nam na pisanie testów parametryzowanych. Samo pisanie testów wymaga podania adnotacji `@ParameterizedTest` oraz odpowiedniej adnotacji określającego źródło parametrów. Adnotacja ta jest traktowana jak adnotacja `@Test`.



Rys. 8. Parametryzowany test JUnit sprawdzający liczby pierwsze



Rys. 9. Wynik działania parametryzowanego testu

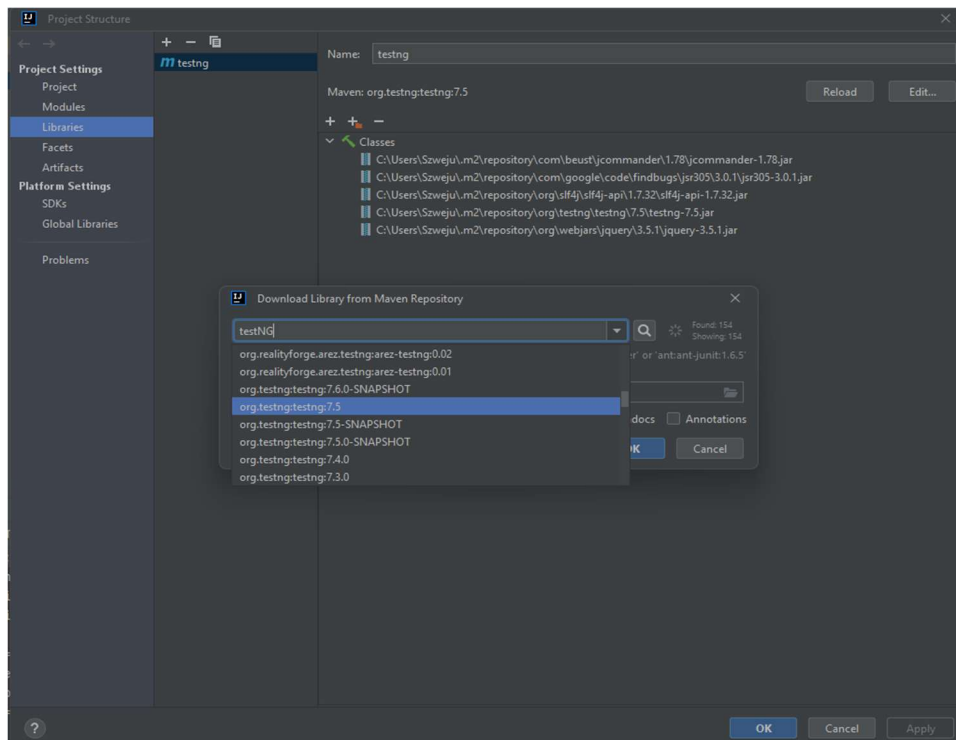
1.2. TestNG

TestNG jest biblioteką programistyczną służącą do pisania testów dla języka programowania Java. Jej autorem jest Cédric Beust. TestNG został napisany, aby stworzyć alternatywę dla narzędzia JUnit 3.x. Biblioteka obsługuje różne rodzaje testów m.in. testy jednostkowe, testy integracyjne, testy funkcjonalne. TestNG jest oprogramowaniem open-source udostępnianym na licencji Apache. Twórcy TestNG jako główne cechy charakterystyczne swojej biblioteki podają:

- wykorzystanie adnotacji Java 5,
- elastyczną konfigurację testów,
- obsługę Data-driven testing,
- obsługę parametryzacji testów,
- testowanie w środowisku rozproszonym,
- rozbudowany mechanizm tworzenia zestawów testów,
- integrację z popularnymi narzędziami (Maven, Apache Ant) i IDE (Eclipse, IntelliJ IDEA),
- rozszerzalność za pomocą języka skryptowego BeanShell,
- wykorzystanie tylko standardowego API Java SE.

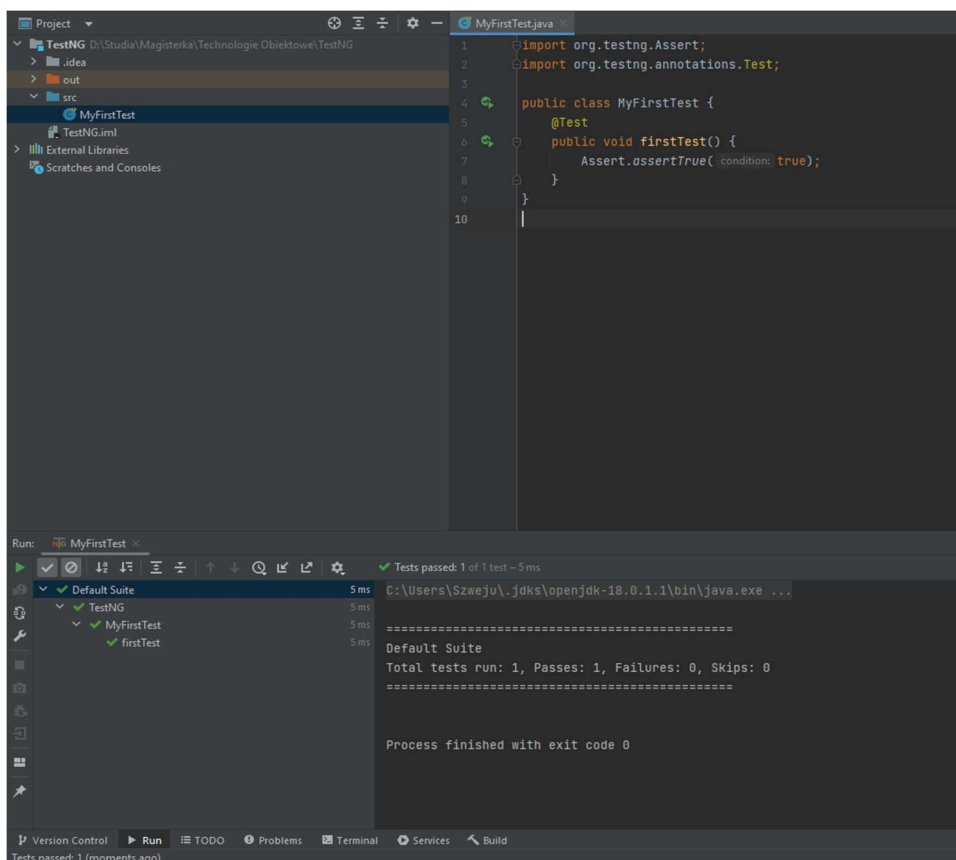
1.2.1. Instalacja TestNG

Aby zainstalować TestNG w środowisku IntelliJ IDEA należy kliknąć prawym przyciskiem myszy na utworzony projekt i wybrać opcję „Open Module Settings”. Następnie wybieramy zakładkę „Libraries”, klikamy plus i „From Maven”. Wyszukujemy „org.testng:testng:7.5” i zatwierdzamy



Rys. 9. Dodanie biblioteki TestNG do projektu

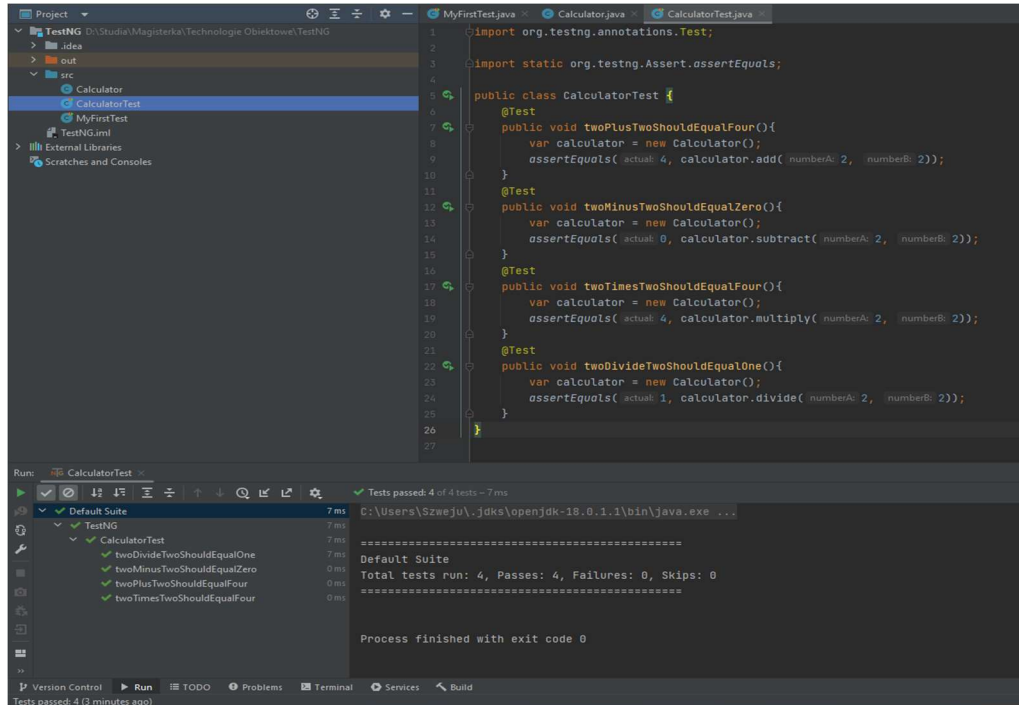
Podobnie jak w przypadku JUnit, aby zobaczyć, czy biblioteka została zainstalowana poprawnie należy utworzyć klasę z naszym pierwszym testem.



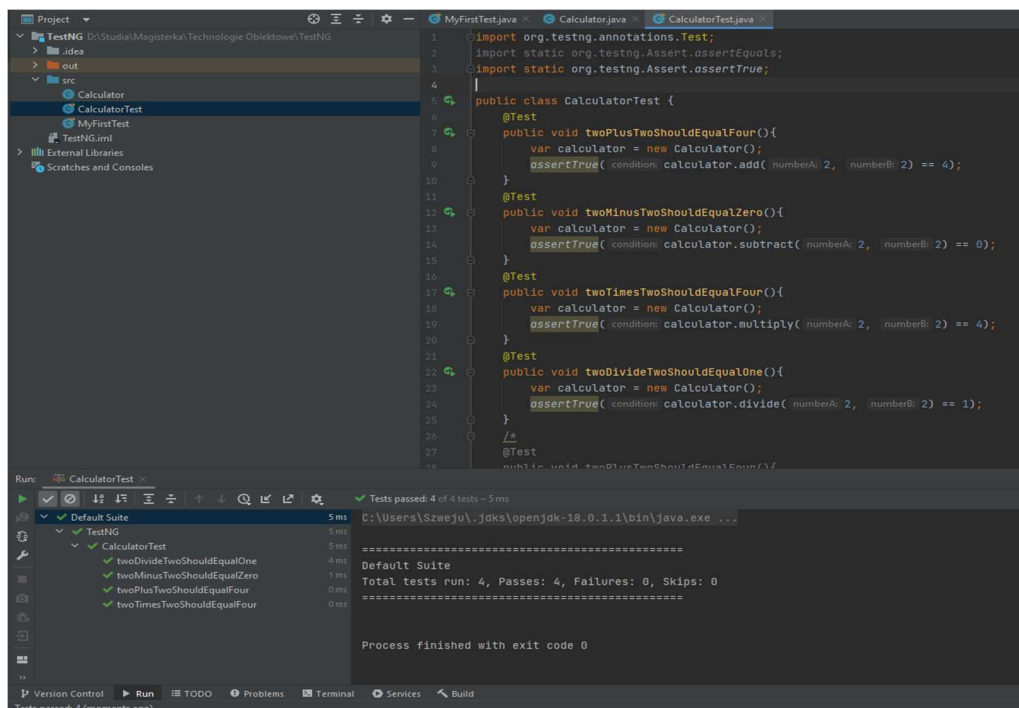
Rys. 10. Pierwszy test sprawdzający poprawne dodanie TestNG do projektu

1.2.2. Testowanie

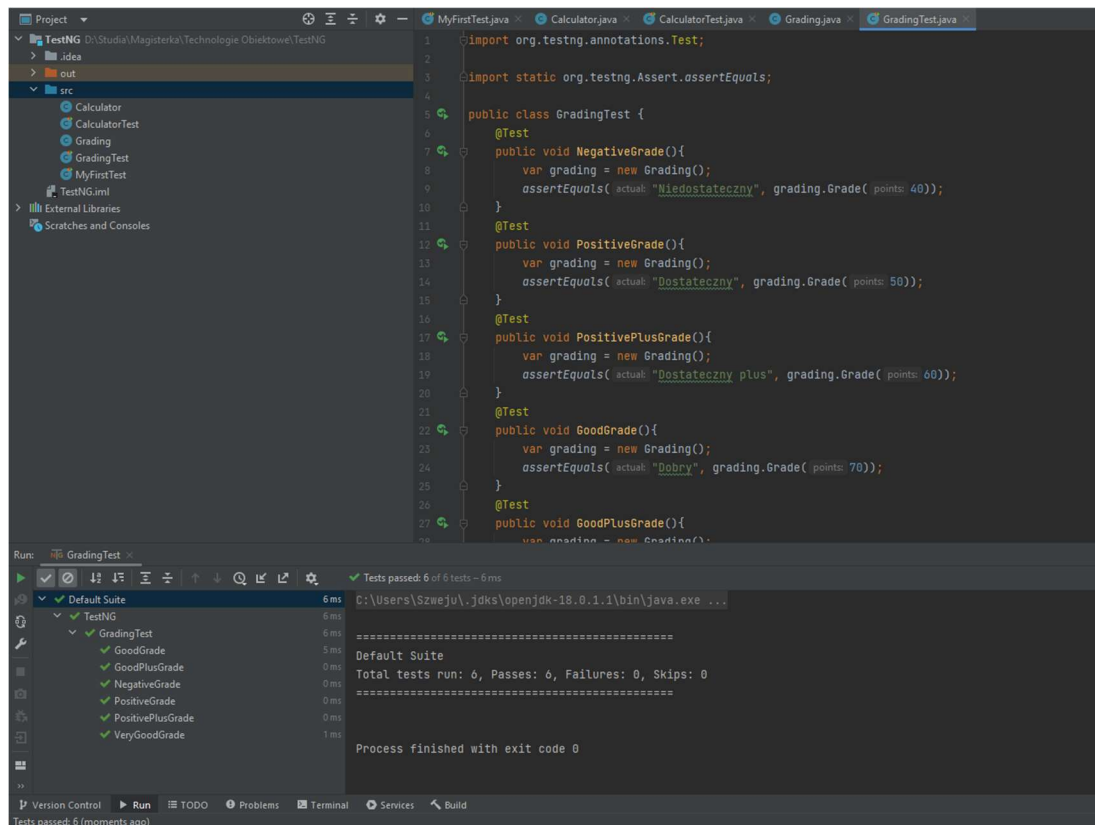
Narzędzie TestNG jest rozszerzeniem JUnit, więc testowanie może odbywać się w ten sam sposób. TestNG umożliwia użycie asercji co zostało pokazane na poniższym obrazie testując ten sam, prosty kalkulator.



Rys. 11. Test prostego kalkulatora narzędziem TestNG

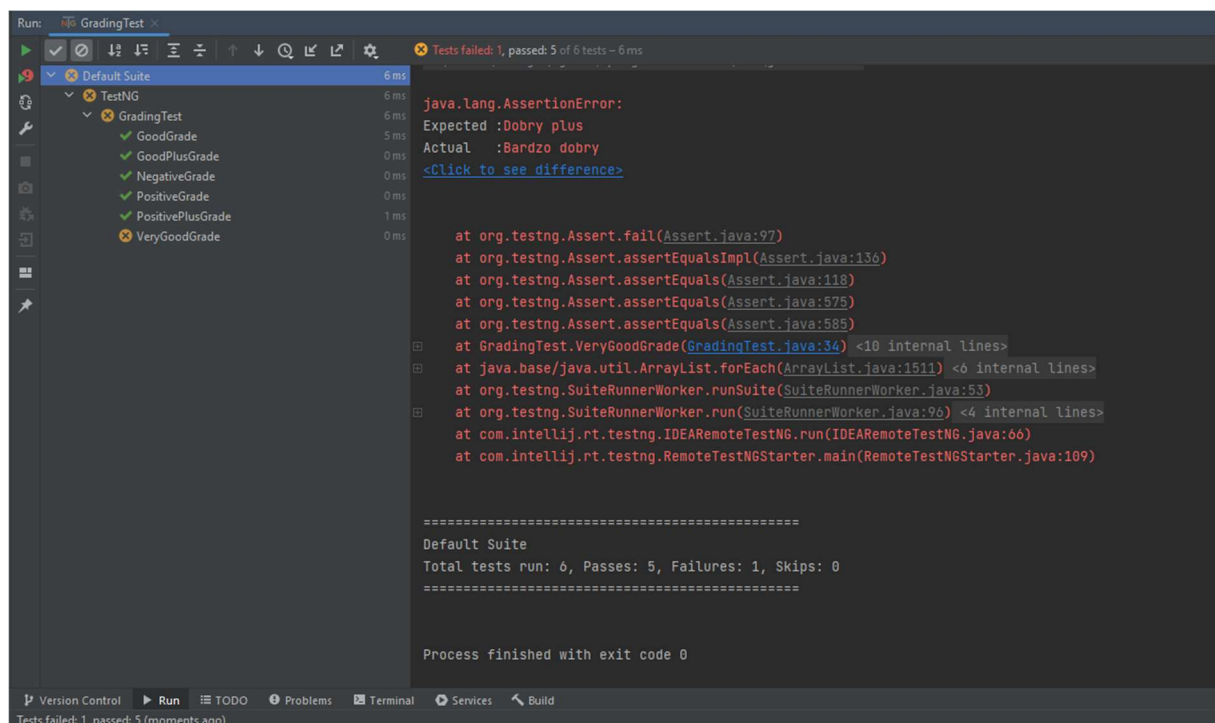


Rys. 12. Test prostego kalkulatora używając „assertTrue” narzędziem TestNG



Rys. 13. Test sprawdzający poprawność zwracanych wartości typu string za pomocą TestNG

W TestNG test, który nie został ukończony pomyślnie wygląda następująco.

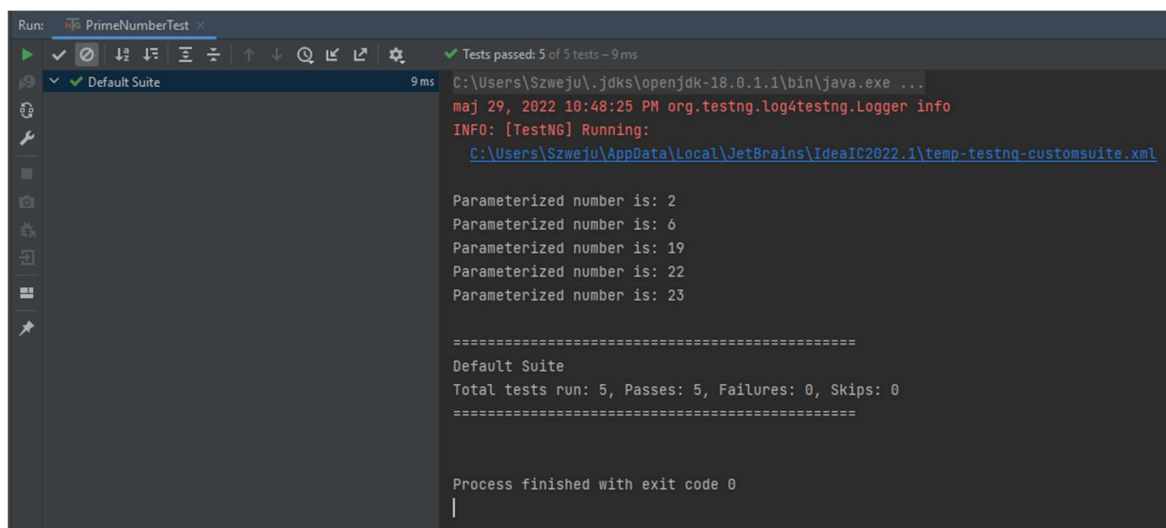


Rys. 14. Niepomyślny test TestNG

Kolejną interesującą funkcją dostępną w TestNG jest testowanie parametryczne. W większości przypadków logika biznesowa wymaga bardzo różnej liczby testów. Testy sparametryzowane umożliwiają programistom ciągłe uruchamianie tego samego testu przy użyciu różnych wartości. Gdy trzeba przekazać złożone parametry lub parametry, które trzeba utworzyć z Javy (złożone obiekty, obiekty odczytane z pliku lub bazy danych itp.), parametry można przekazać za pomocą Dataproviderów. Dostawca danych to metoda z adnotacją `@DataProvider`. Ta adnotacja ma tylko jeden atrybut ciągu: jego nazwę. Jeśli nazwa nie zostanie podana, nazwa dostawcy danych zostanie ustawiona na nazwę metody. Dostawca danych zwraca tablicę obiektów. Poniżej pokazano przykład użycia sparametryzowanego testu z użyciem adnotacji `@DataProvider`.

```
1 import org.testng.Assert;
2 import org.testng.annotations.BeforeMethod;
3 import org.testng.annotations.DataProvider;
4 import org.testng.annotations.Test;
5
6 import static org.testng.Assert.assertEquals;
7
8 public class PrimeNumberTest {
9
10     2 usages
11     private PrimeNumberChecker primeNumberChecker;
12
13     @BeforeMethod
14     public void initialize() {
15         primeNumberChecker = new PrimeNumberChecker();
16     }
17
18     1 usage
19     @DataProvider(name = "test1")
20     public static Object[][] primeNumbers() {
21         return new Object[][] {{2, true}, {6, false}, {19, true}, {22, false}, {23, true}};
22     }
23
24     // This test will run 4 times since we have 5 parameters defined
25
26     1 usage
27     @Test(dataProvider = "test1")
28     public void testPrimeNumberChecker(Integer inputNumber, Boolean expectedResult) {
29         System.out.println("Parameterized number is: " + inputNumber);
30         assertEquals(expectedResult, primeNumberChecker.validate(inputNumber));
31     }
32 }
```

Rys. 15. Sparametryzowany test sprawdzający liczby pierwsze



```
Run: PrimeNumberTest
Tests passed: 5 of 5 tests - 9 ms
C:\Users\Szweju\.jdk\openjdk-18.0.1.1\bin\java.exe ...
maj 29, 2022 10:48:25 PM org.testng.log4testng.Logger info
INFO: [TestNG] Running:
C:\Users\Szweju\AppData\Local\JetBrains\IdeaIC2022.1\temp-testng-customsuite.xml

Parameterized number is: 2
Parameterized number is: 6
Parameterized number is: 19
Parameterized number is: 22
Parameterized number is: 23

=====
Default Suite
Total tests run: 5, Passes: 5, Failures: 0, Skips: 0
=====

Process finished with exit code 0
```

Rys. 16. Wynik sparametryzowanego testu

1.3. Mockito

Mockito to popularny framework open source do wyśmiewania obiektów w testach oprogramowania. Korzystanie z Mockito znacznie upraszcza tworzenie testów dla klas z zależnościami zewnętrznymi. Obiekt *mock* to fikcyjna implementacja interfejsu lub klasy. Pozwala na zdefiniowanie wyjścia niektórych wywołań metod. Zwykle rejestrują interakcję z systemem, a testy mogą to potwierdzić. Najnowsze wersje Mockito mogą również mockować metody statyczne i klasy końcowe. Również metody prywatne nie są widoczne dla testów. Mockito rejestruje interakcję z mockiem i pozwala sprawdzić, czy obiekt mock został użyty poprawnie, np. czy została wywołana określona metoda. Pozwala to na zaimplementowanie testów zachowania zamiast tylko testowania wyników wywołań metod.

Mockito udostępnia kilka metod tworzenia atrap obiektów:

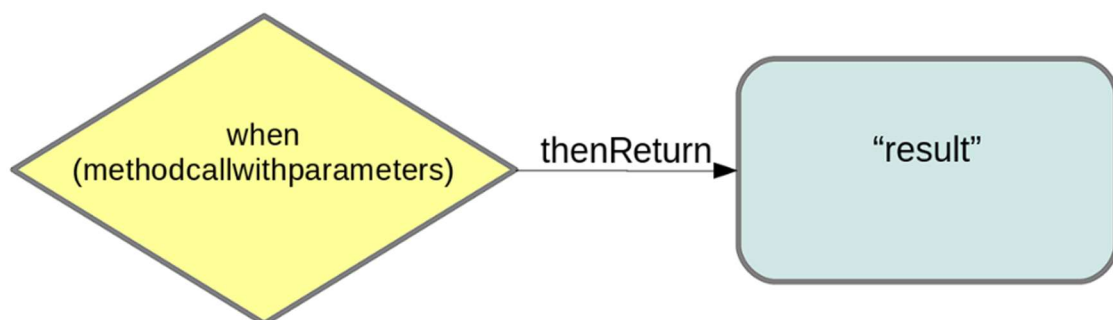
- Korzystanie z `@ExtendWith(MockitoExtension.class)` rozszerzenia dla JUnit 5 w połączeniu z adnotacją `@Mock` na polach
- Za pomocą metody statycznej `mock()`.
- Korzystanie z adnotacji `@Mock`.

Jeśli używana jest adnotacja `@Mock`, należy wywołać inicjalizację pól z adnotacjami. Robi to `MockitoExtension` wywołując metodę statyczną `MockitoAnnotations.initMocks(this)`.

Mockito pozwala skonfigurować wartości zwracane metod, które są wywoływane na makiecie za pośrednictwem interfejsu API Fluent. Nieokreślone wywołania metod zwracają „puste” wartości:

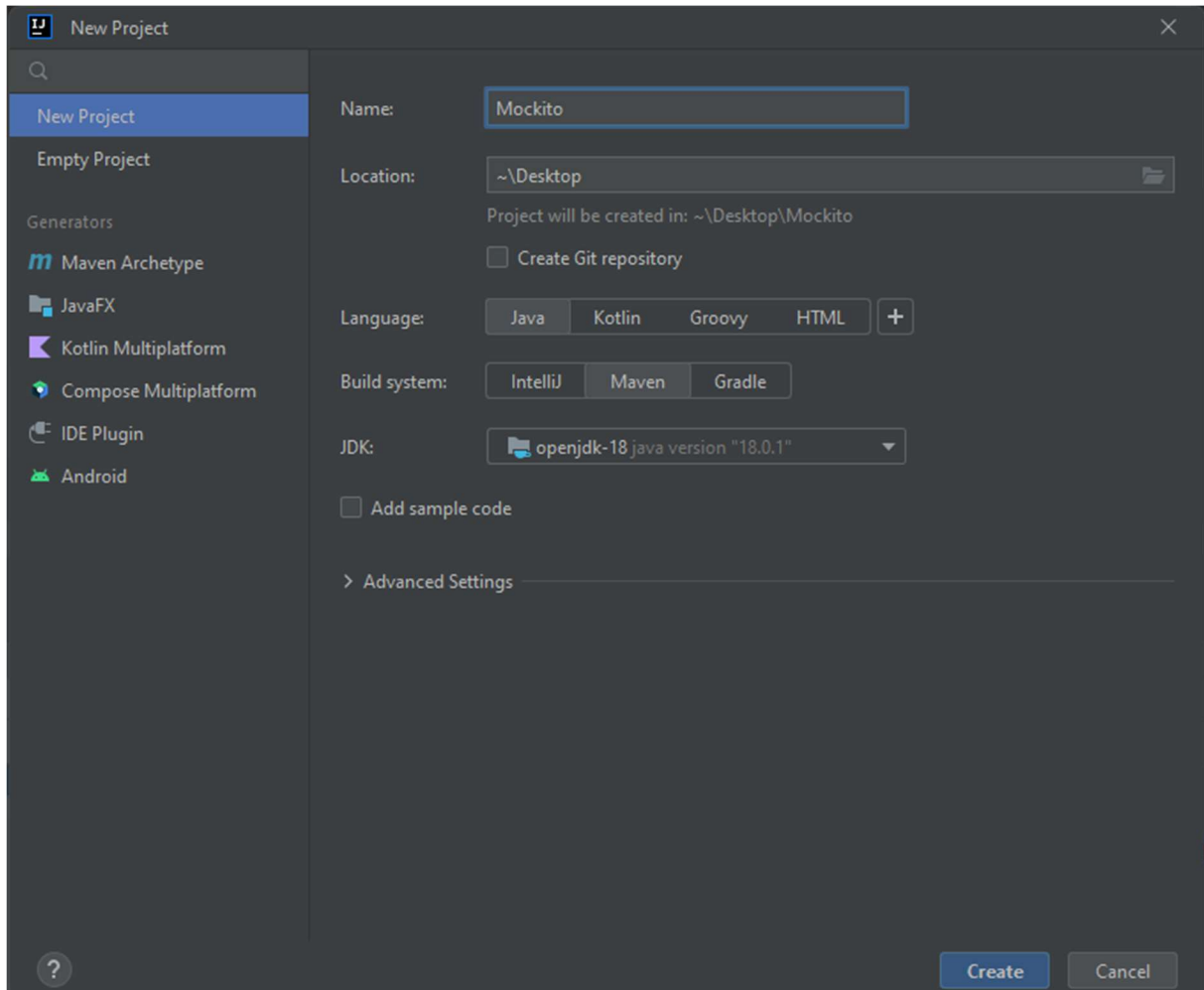
- null dla obiektów,
- 0 dla liczb,
- false dla wartości logicznej,
- puste kolekcje na kolekcje.

Mocki mogą zwracać różne wartości w zależności od argumentów przekazanych do metody. Łańcuch `when(...).thenReturn(...)` służy do określenia wartości zwracanej dla wywołania metody ze wstępnie zdefiniowanymi parametrami.

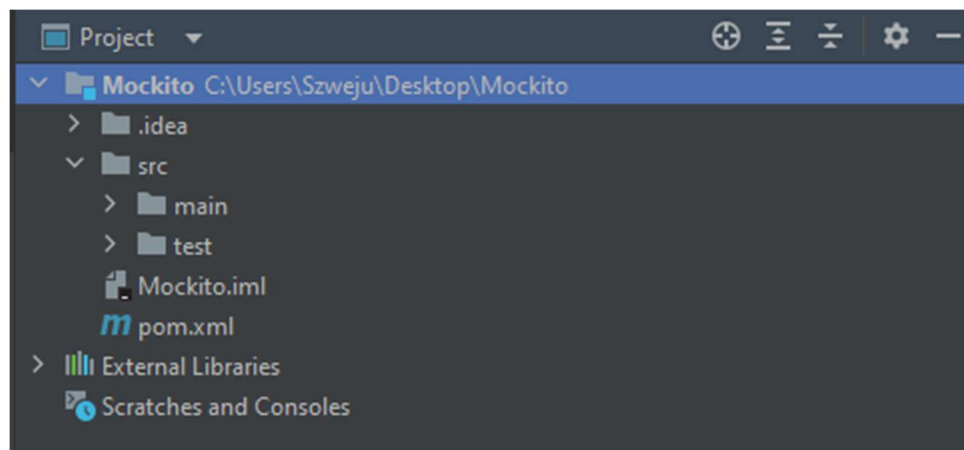


1.3.1. Dodanie Mockito do projektu

Korzystanie z bibliotek Mockito powinno odbywać się za pomocą nowoczesnego systemu zależności, takiego jak Maven czy Gradle. Wszystkie nowoczesne IDE (Eclipse, Visual Studio Code, IntelliJ) obsługują zarówno Maven, jak i Gradle. Aby zacząć korzystać z Mockito należy utworzyć projekt i w miejscu „Build system” wybrać opcję Maven.

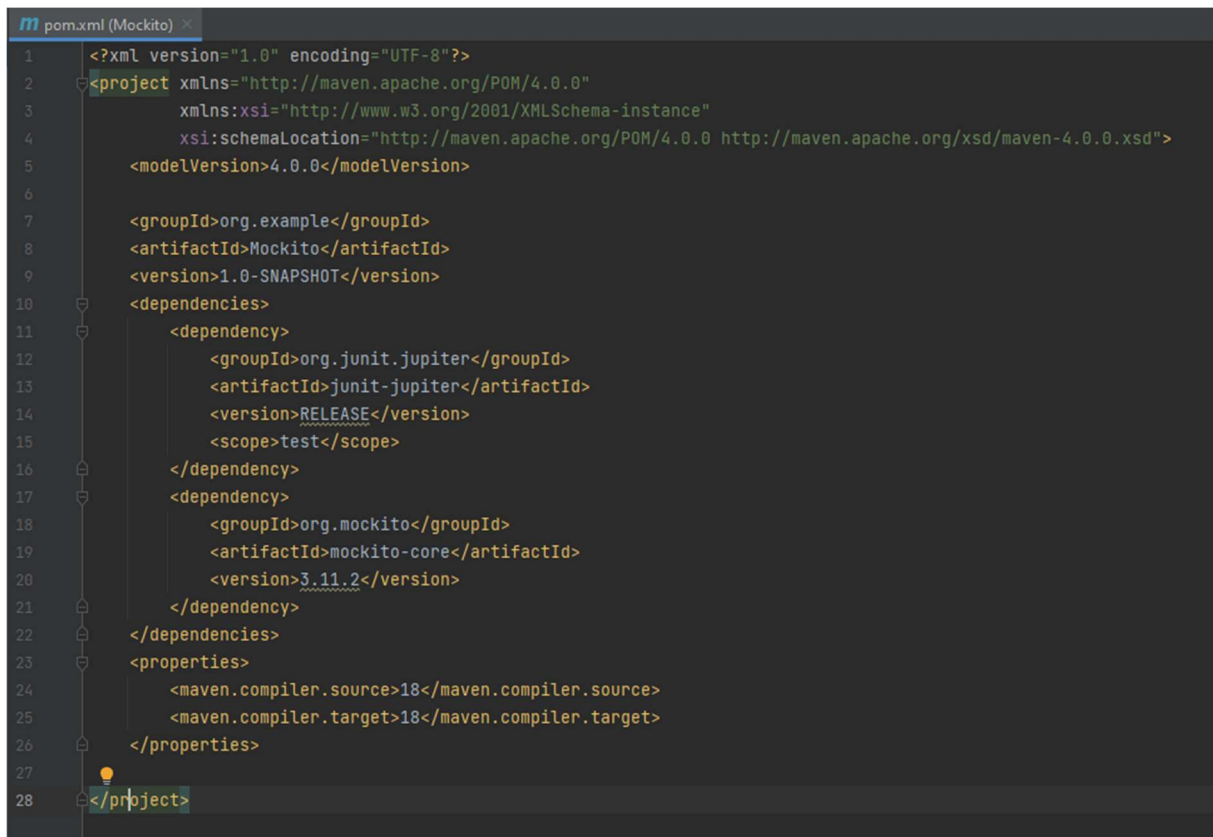


Rys. 17. Utworzenie nowego projektu



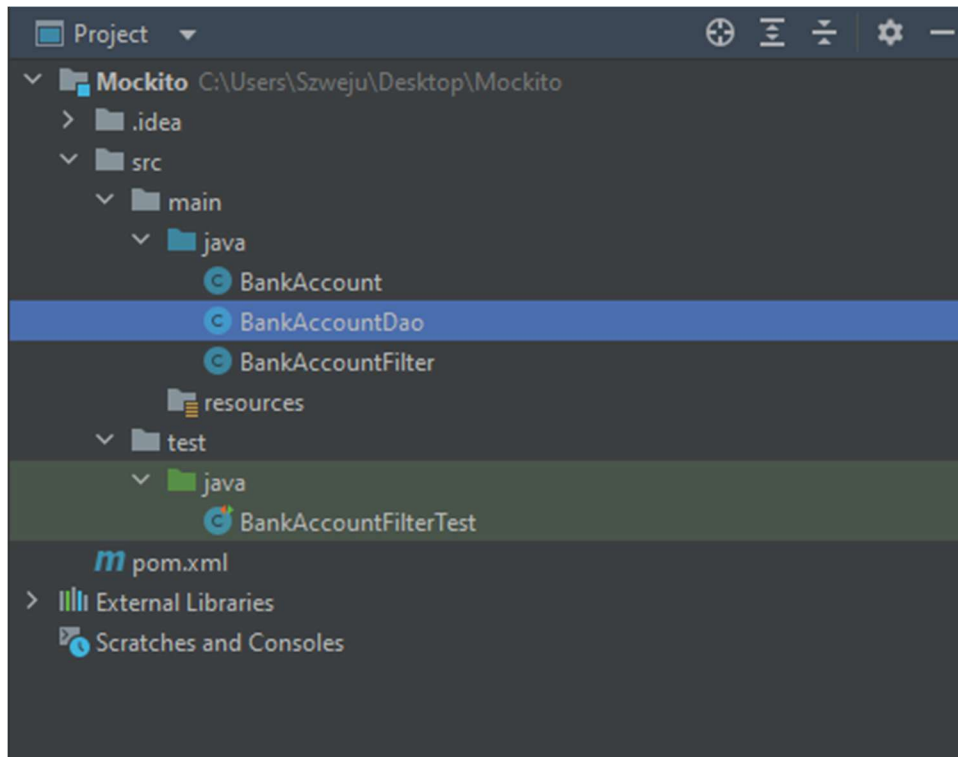
Rys. 18. Utworzony projekt

W pliku pom.xml dodajemy zależności jak na zrzucie ekranu umieszczonym poniżej.



```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <project xmlns="http://maven.apache.org/POM/4.0.0"
3         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
5     <modelVersion>4.0.0</modelVersion>
6
7     <groupId>org.example</groupId>
8     <artifactId>Mockito</artifactId>
9     <version>1.0-SNAPSHOT</version>
10    <dependencies>
11        <dependency>
12            <groupId>org.junit.jupiter</groupId>
13            <artifactId>junit-jupiter</artifactId>
14            <version>RELEASE</version>
15            <scope>test</scope>
16        </dependency>
17        <dependency>
18            <groupId>org.mockito</groupId>
19            <artifactId>mockito-core</artifactId>
20            <version>3.11.2</version>
21        </dependency>
22    </dependencies>
23    <properties>
24        <maven.compiler.source>18</maven.compiler.source>
25        <maven.compiler.target>18</maven.compiler.target>
26    </properties>
27
28 </project>
```

Rys. 19. Zależności dodane do pliku pom.xml



Rys. 20. Struktura projektu

```
m pom.xml (Mockito) x BankAccount.java x BankAccountDao.java x BankAccountFilterTest.java x
1 import org.junit.jupiter.api.Assertions;
2 import org.junit.jupiter.api.Test;
3 import org.mockito.Mockito;
4
5 import java.util.Arrays;
6 import java.util.List;
7
8 import static org.junit.jupiter.api.Assertions.*;
9
10 public class BankAccountFilterTest {
11     @Test
12     void filterAccounts() {
13         BankAccount acct1 = new BankAccount( acctNum: 1234, balance: 50, lastTransaction: -100);
14         BankAccount acct2 = new BankAccount( acctNum: 4356, balance: 150, lastTransaction: 100);
15
16         BankAccountDao dao = Mockito.mock(BankAccountDao.class);
17
18         List<BankAccount> results = Arrays.asList(acct1, acct2);
19         Mockito.when(dao.all()).thenReturn(results);
20
21         BankAccountFilter filter = new BankAccountFilter(dao);
22         List<BankAccount> result = filter.filter();
23
24         Assertions.assertEquals(acct1, result.size());
25     }
26 }
27
```

Rys. 21. Test operacji konta bankowego przy użyciu Mockito

```
Run: BankAccountFilterTest.filterAccounts x
Tests passed: 1 of 1 test - 1 sec 219 ms
Test Results 1 sec 219 ms
  BankAccountFilterTest 1 sec 219 ms
    filterAccounts() 1 sec 219 ms
Process finished with exit code 0
```

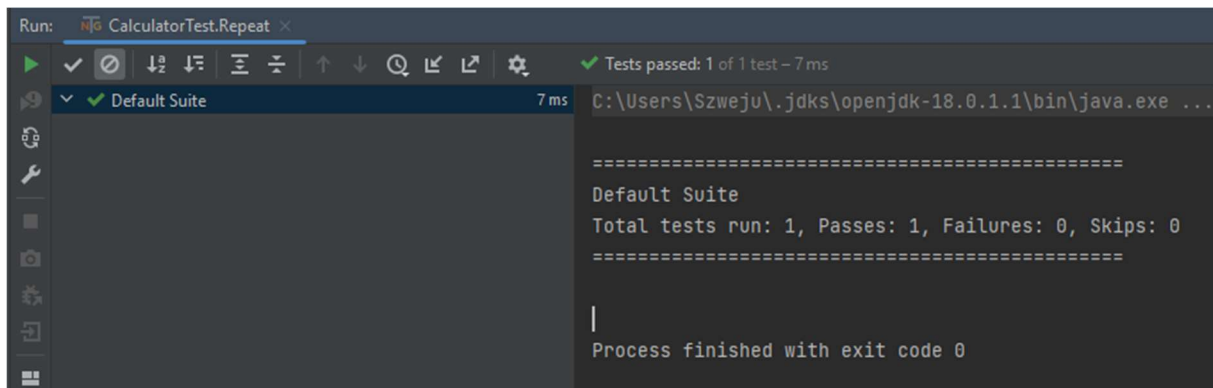
Rys. 22. Wynik działania testu

2. Porównanie

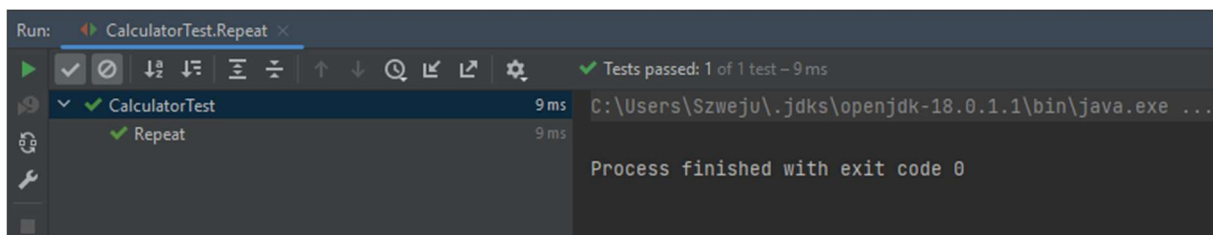
Wykonanie testu kalkulatora zostało uruchomione 100000 razy za pomocą pętli *for*.

```
26      @Test
27      public void Repeat(){
28          for(int i = 0; i< 100000; i++)
29          {
30              twoPlusTwoShouldEqualFour();
31              twoMinusTwoShouldEqualZero();
32              twoTimesTwoShouldEqualFour();
33              twoDivideTwoShouldEqualOne();
34          }
35      }
```

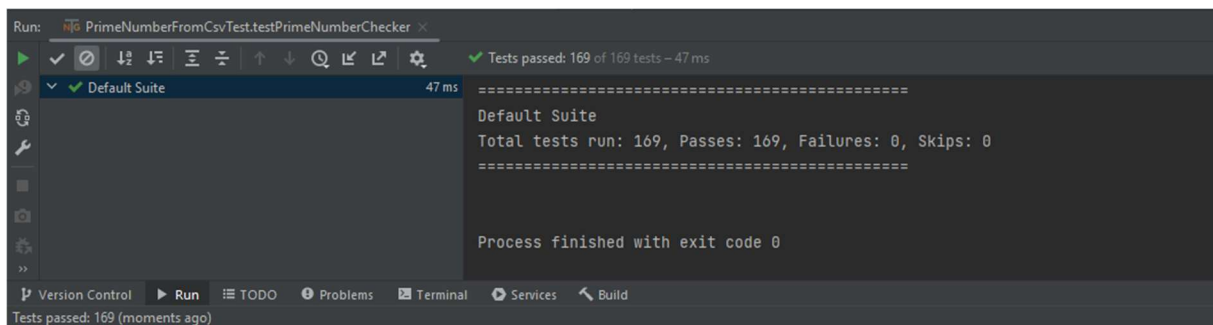
Rys. 15. Kod źródłowy testu.



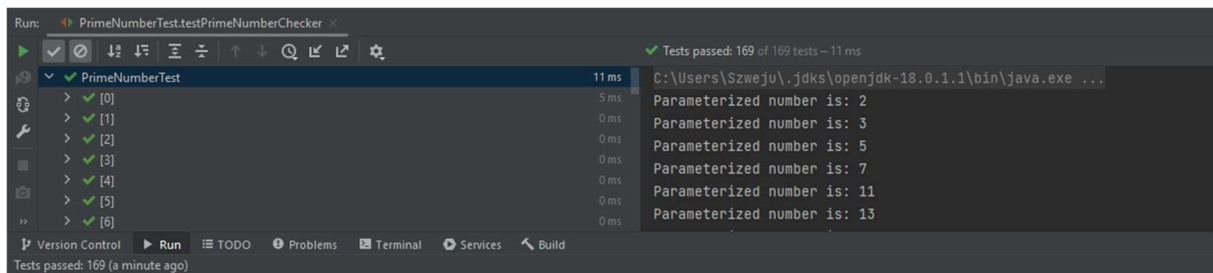
Rys. 16. Czas wykonania testu w TestNG.



Rys. 17. Czas wykonania testu w JUnit.



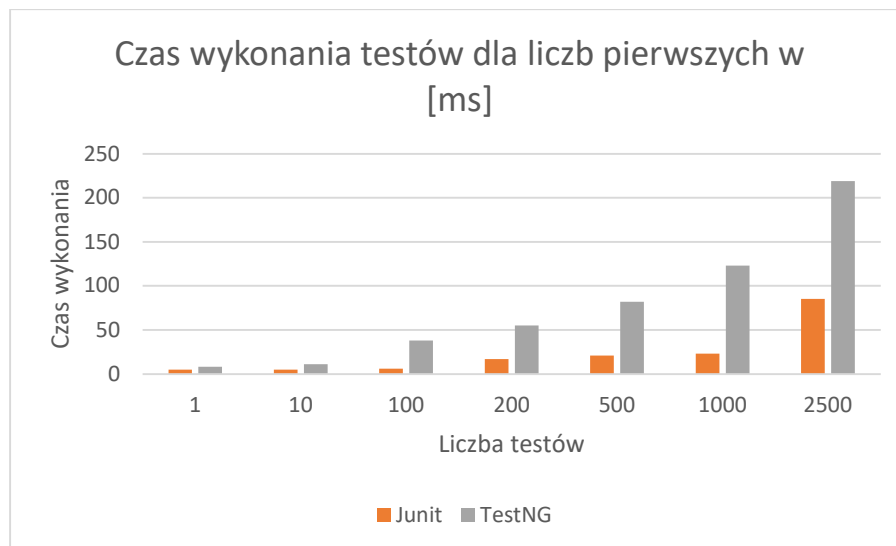
Rys. 18. Czas wykonania 169 testów liczby pierwszej w TestNG



Rys. 19. Czas wykonania 169 testów liczby pierwszej w JUnit

Poniżej przedstawiono wykres przedstawiający czas wykonania testów przy różnej ilości parametrów. Wprowadzono odpowiednio: 1, 10, 100, 200, 500, 1000 oraz 2500 parametrów. Przetestowano na dwóch komputerach o różnej specyfikacji. Na pierwszy rzut oka widać znaczną różnicę w czasie wykonania na korzyść JUnit. Różnice w czasach są bardziej widoczne wraz ze wzrostem podawanych parametrów co przedstawiono również w tabeli.

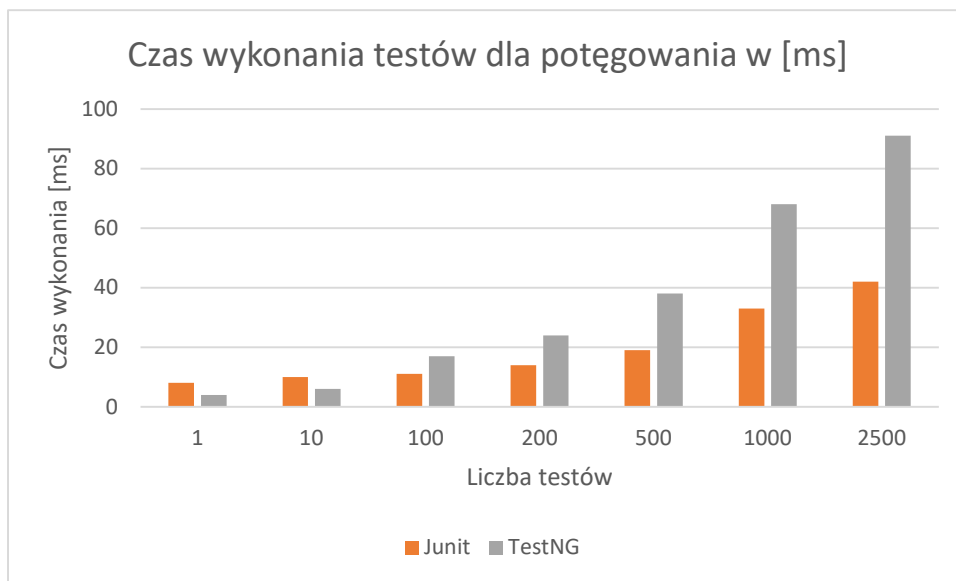
2.1.1. Komputer nr 1. (i5-12400, 16GB RAM 3600mhz)



Wykres 1. Różnice w czasach wykonania testów w JUnit oraz TestNG dla testów liczb pierwszych

Liczba testów	JUnit	TestNG
1	5	8
10	5	11
100	6	38
200	17	55
500	21	82
1000	23	123
2500	85	219

Tabela 1. Wyniki wykonanych testów liczb pierwszych na pierwszym komputerze

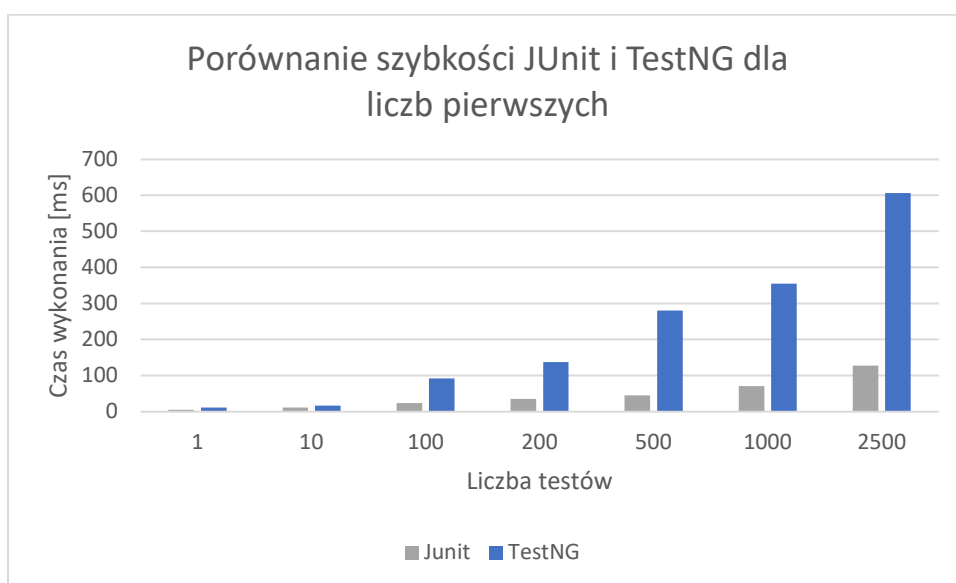


Wykres 2. Różnice w czasach wykonania testów w JUnit oraz TestNG dla testów potęgowania

Liczba testów	JUnit	TestNG
1	8	4
10	10	6
100	11	17
200	14	24
500	19	38
1000	33	68
2500	42	91

Tabela 1. Wyniki wykonanych testów potęgowania na pierwszym komputerze

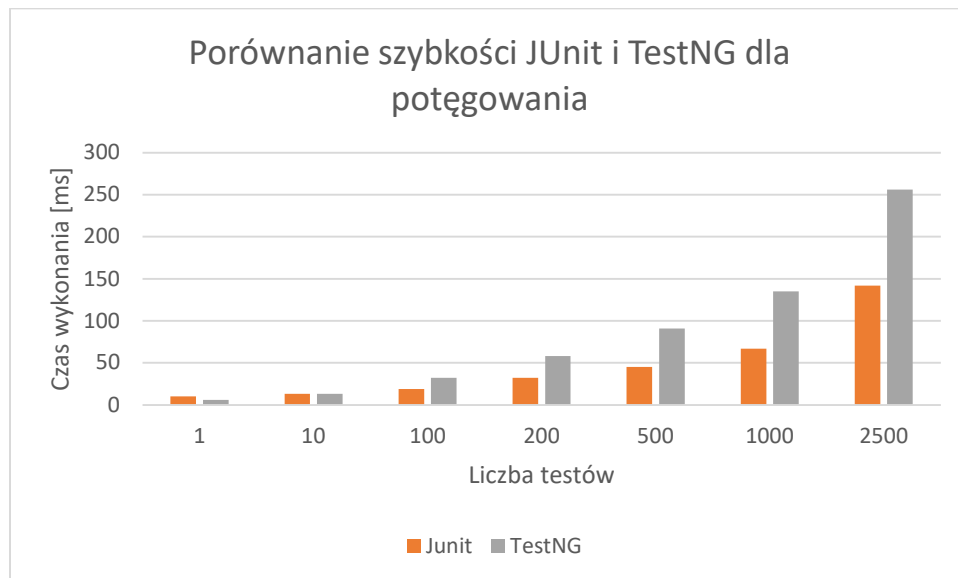
2.1.2. Komputer nr 2. (i5-8300h, 16GB RAM 2666mhz)



Wykres 2. Różnice w czasach wykonania testów w JUnit oraz TestNG

Liczba testów	Junit	TestNG
1	5	11
10	11	16
100	23	92
200	35	137
500	45	280
1000	70	355
2500	127	606

Tabela 2. Wyniki wykonanych testów liczb pierwszych na drugim komputerze



Wykres 2. Różnice w czasach wykonania testów w JUnit oraz TestNG

Liczba testów	Junit	TestNG
1	10	6
10	13	13
100	19	32
200	32	58
500	45	91
1000	67	135
2500	142	256

Tabela 3. Wyniki wykonanych testów liczb pierwszych na drugim komputerze

2.2. Różnice między JUnit i TestNG

JUnit	TestNG
Framework typu Open Source, używany do wyzwalania i pisania testów	Platforma oparta na Javie, która jest ulepszoną opcją do przeprowadzania testów
Nie obsługuje uruchamiania testów równoległych	Obsługa testów równoległych
Nie obsługuje zaawansowanych adnotacji	Obsługuje zaawansowane adnotacje
Brak testów zależności	Obsługa testów zależności
Brak możliwości grupowania testów	Obsługa grupowania i uruchamiania równoległego

3. Wnioski

Podczas pisania testów w narzędziach tj. JUnit, TestNG oraz Mockito zdobyłem praktyczną wiedzę na temat testowania jednostkowego w Javie. Testy jednostkowe znajdują problemy na wczesnym etapie pisania aplikacji. Obejmuje to zarówno błędy w implementacji programisty, jak i wady lub brakujące części specyfikacji urządzenia. Testowanie zmusza autora do przemyślenia danych wejściowych, wyjściowych i warunków błędów, a tym samym dokładniejszego zdefiniowania pożądanego zachowania jednostki. Koszt znalezienia błędu przed rozpoczęciem kodowania lub przy pierwszym napisaniu kodu jest znacznie niższy niż koszt późniejszego wykrycia, zidentyfikowania i poprawienia błędu. Porównanie narzędzi pokazało, że najlepsze czasy były osiągnięte dla JUnit. Praca z nim była według mnie najprzyjemniejsza i najbardziej intuicyjna. W internecie można znaleźć bardzo dużo poradników oraz kursów, dzięki czemu rozwiązywanie napotkanych problemów jest dużo łatwiejsze niż w pozostałych narzędziach. Instalacja narzędzi we wszystkich przypadkach przeszła bez żadnego problemu. Porównanie rozwiązań pokazało mi jak ważną rolę odgrywają testy jednostkowe podczas tworzenia oprogramowania i wiem, że należy ich używać.