POLITECHNIKA ŚWIĘTOKRZYSKA W KIELCACH

Wydział Elektrotechniki, Automatyki i Informatyki

Projekt Technologii Obiektowych

Temat projektu:	Autorzy:
41 – Porównanie rozwiązań związanych z testowaniem	Daniel Szwajkowski, Kacper Jurek

Podział pracy:

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

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. Wynik tej metody jest przekazywany do metody Assert.assertTrue(), jest to tak zwana asercja.

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.

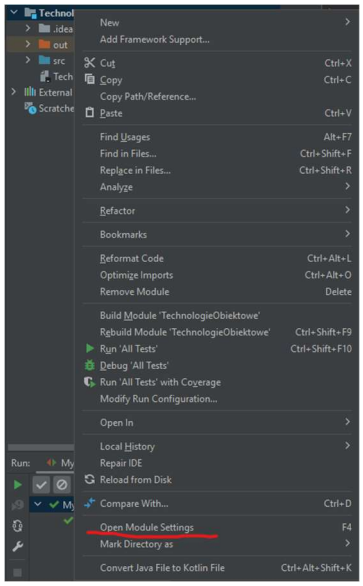
1.1. JUnit

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.

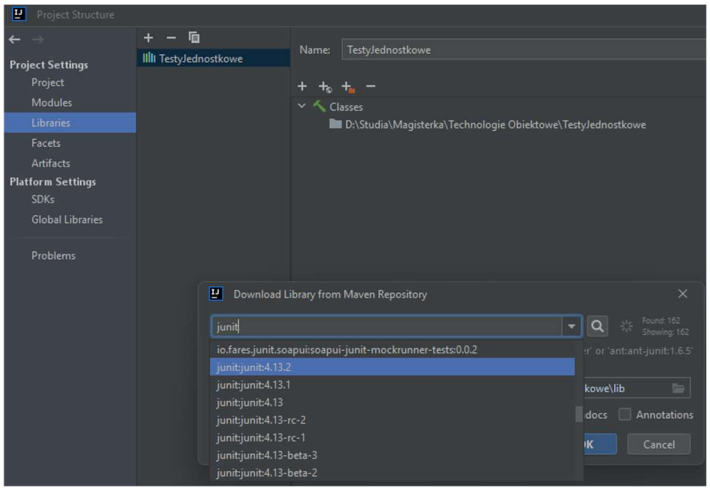
1.1.1. Instalacja JUnit

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



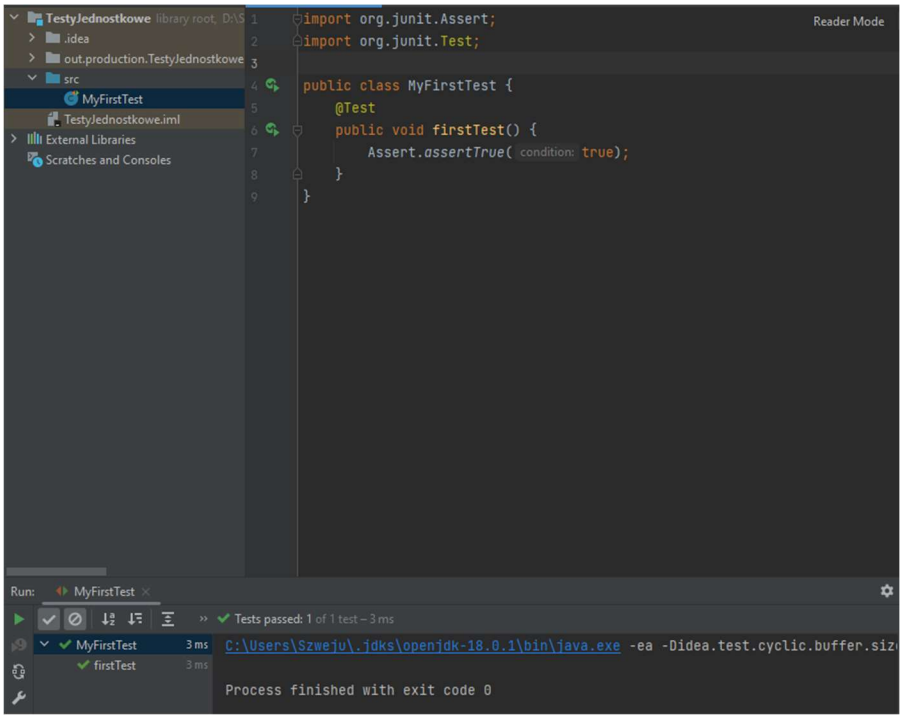
Rys. 1. Otworzenie ustawień projektu

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.



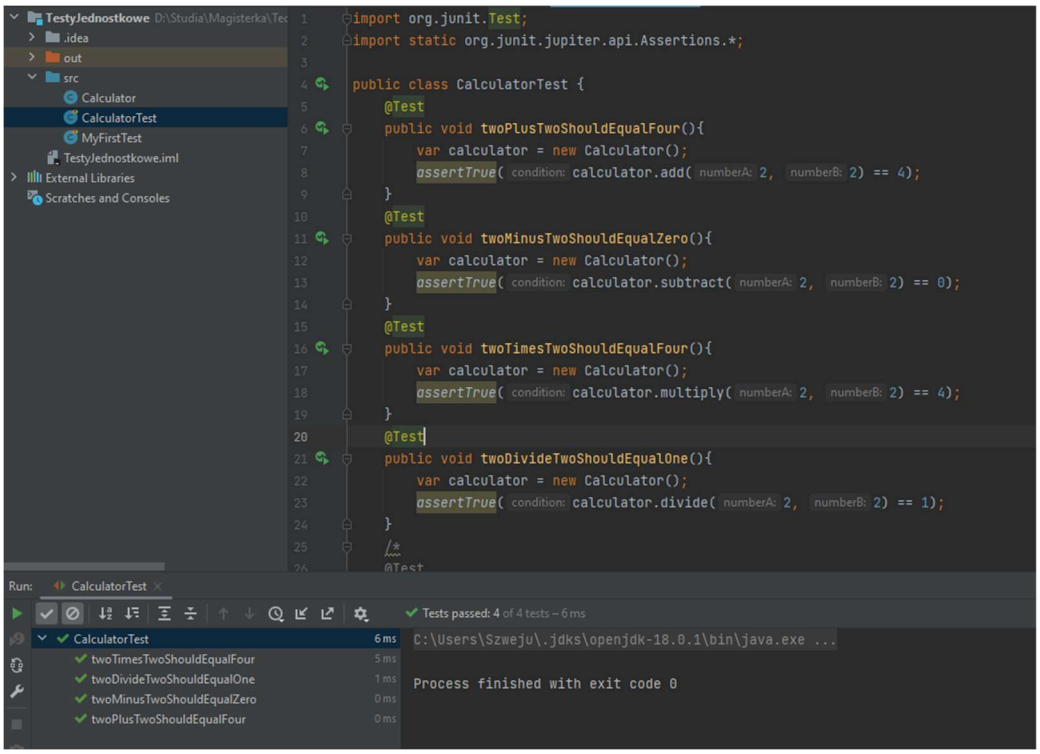
Rys. 3. Pierwszy test sprawdzający poprawne dodanie JUnit do projektu

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.

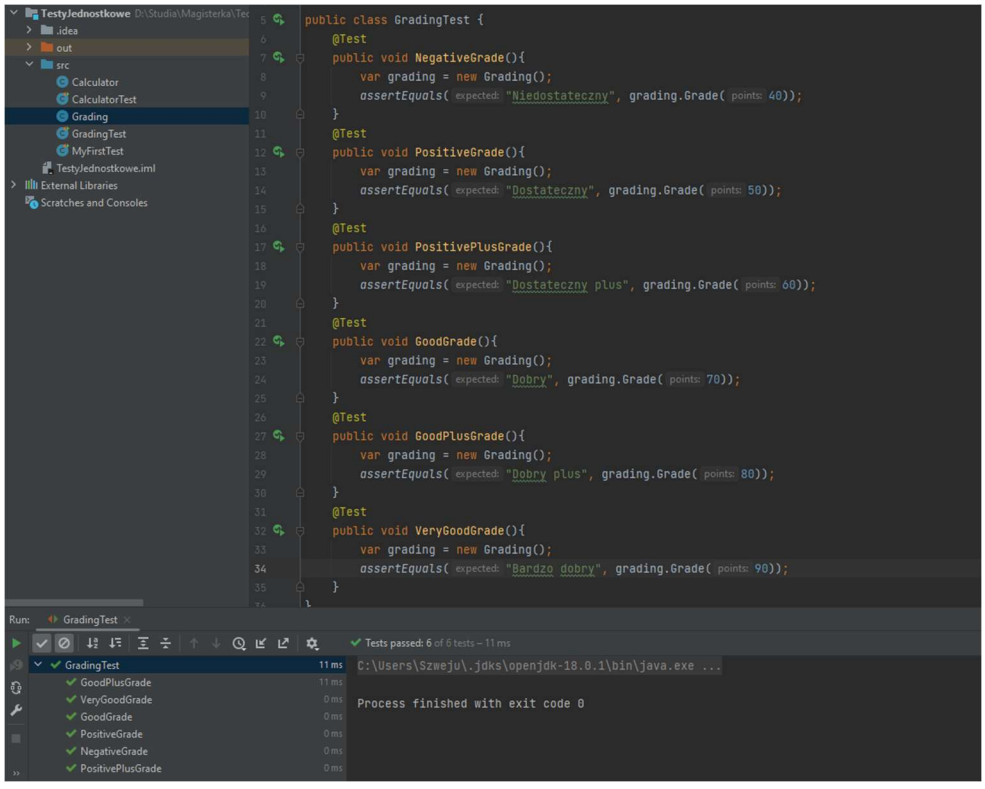
Rys. 4. Test prostego kalkulatora

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



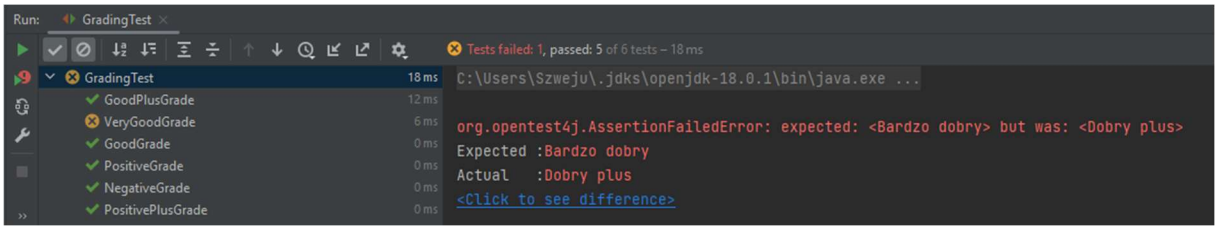
Rys. 5. Test prostego kalkulatora używając „assertTrue”

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".

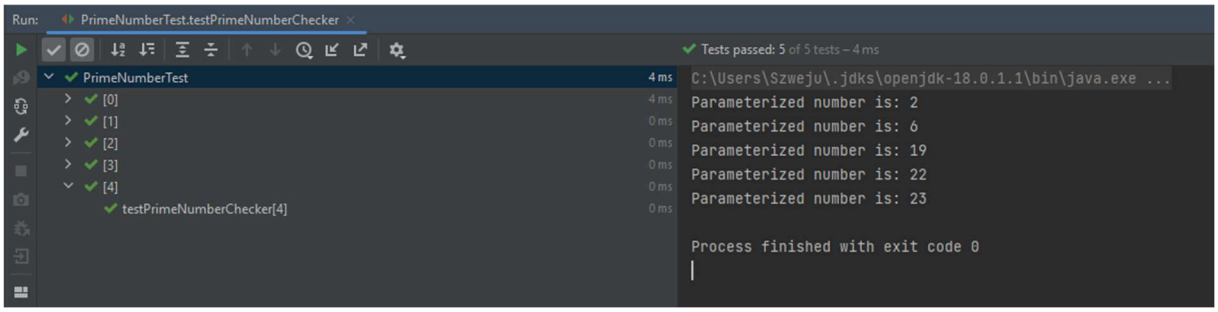


Rys. 7. Niepomyślny test JUnit

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 @ParametrizedTest oraz odpowiedniej adnotacji określającego źródło parametrów. Adnotacja ta jest traktowana jak adnotacja @Test.

```java
import org.junit.Before;
import org.junit.Test;
import org.junit.runner.RunWith;
import org.junit.runners.Parameterized;

import java.util.Arrays;
import java.util.Collection;

import static org.junit.Assert.assertEquals;

@RunWith(Parameterized.class)
public class PrimeNumberTest {
    3 usages
    private final Integer inputNumber;
    2 usages
    private final Boolean expectedResult;
    2 usages
    private PrimeNumberChecker primeNumberChecker;

    @Before
    public void initialize(){
        primeNumberChecker = new PrimeNumberChecker();
    }

    public PrimeNumberTest(Integer inputNumber, Boolean expectedResult) {
        this.inputNumber = inputNumber;
        this.expectedResult = expectedResult;
    }
    @Parameterized.Parameters
    public static Collection primeNumbers() {
        return Arrays.asList(new Object[][] { {2,true}, {6, false}, {19, true}, {22, false}, {23, true}});
    }
    @Test
    public void testPrimeNumberChecker() {
        System.out.println("Parameterized number is: " + inputNumber);
        assertEquals(expectedResult, primeNumberChecker.validate(inputNumber));
    }
}
```

Rys 8. Parametryzowany test JUnit sprawdzający liczby pierwsze



Rys. 9. Wynik działania parametryzowanego testu

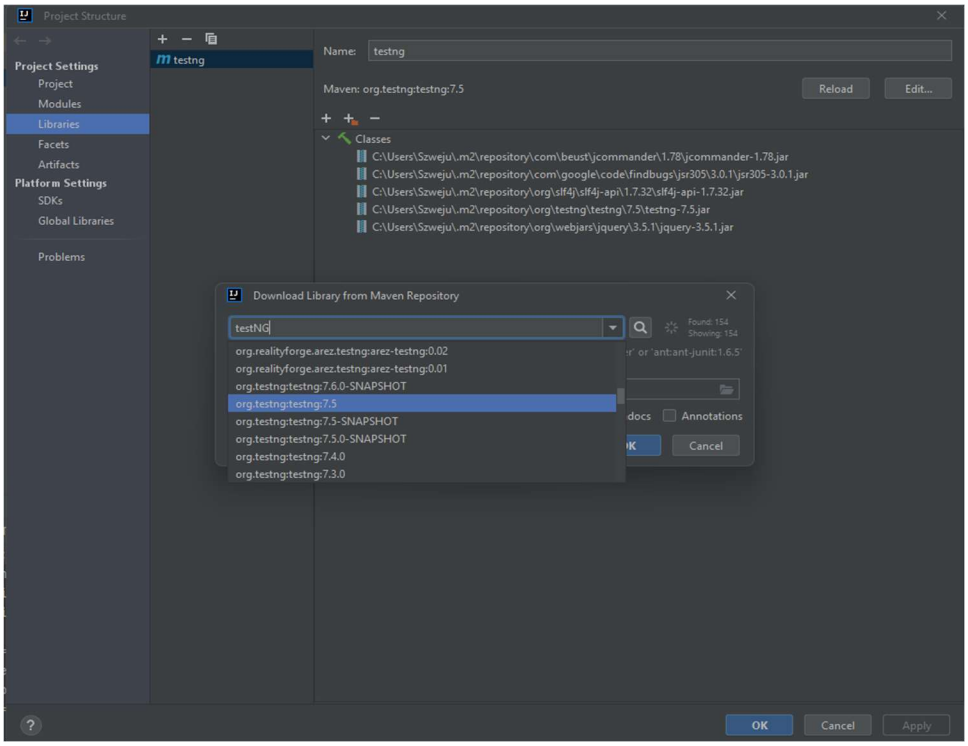
1.2. TestNG

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.

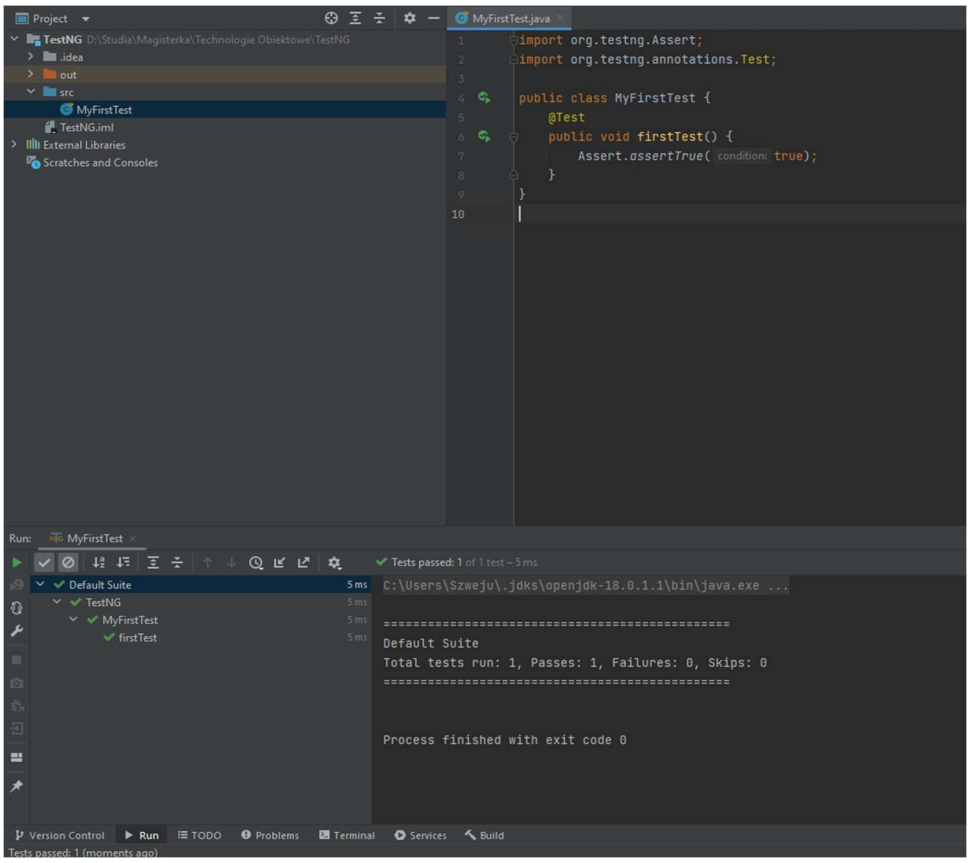
1.2.1. Instalacja TestNG

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

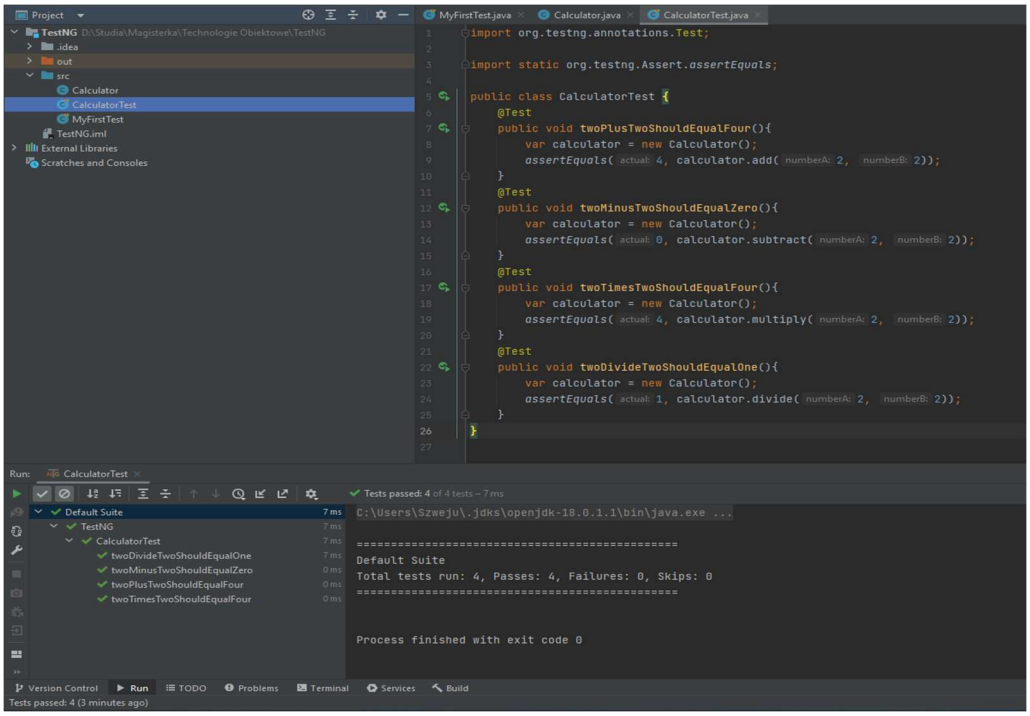
Podobnie jak w przypadku JUnit, aby zobaczyć, czy biblioteka została zainstalowana poprawnie należy utworzyć klasę z naszym pierwszym testem.



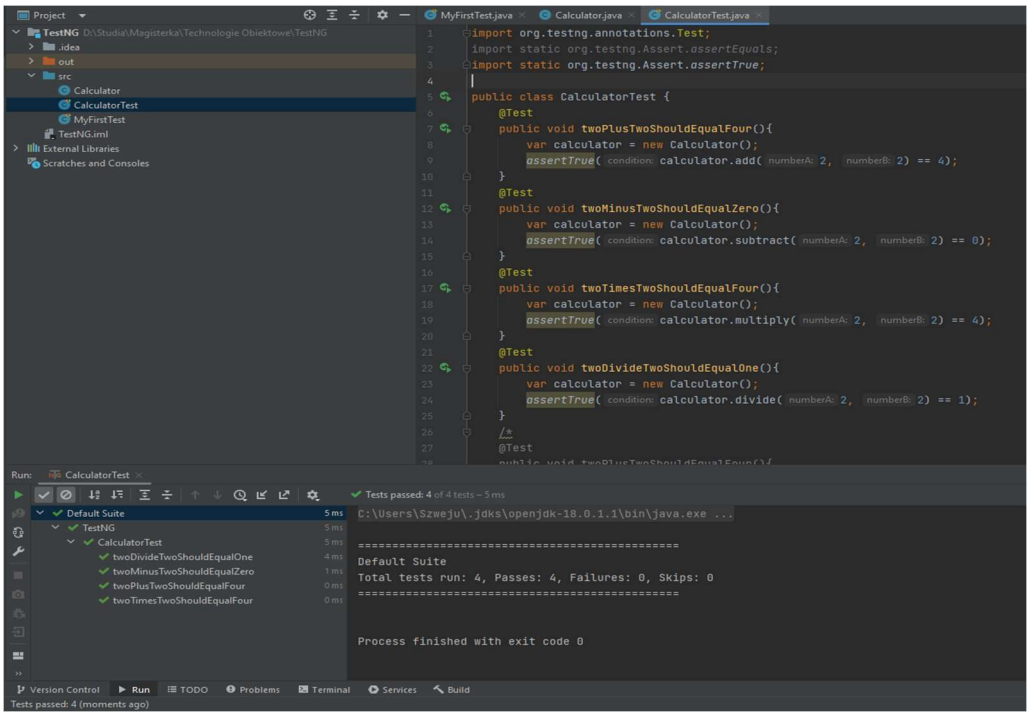
Rys. 10. Pierwszy test sprawdzający poprawne dodanie TestNG do projektu

1.2.2. Testowanie

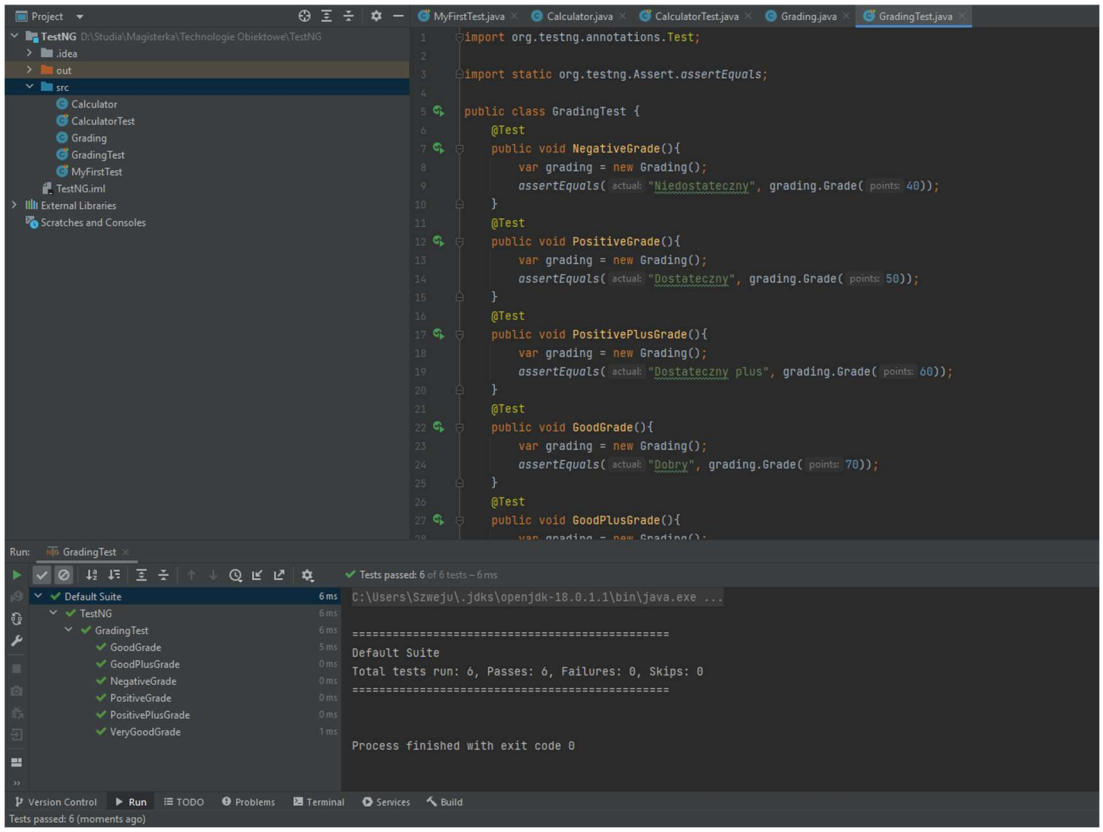
Narzędzie TestNG jest rozszerzeniem JUnit, więc testowanie może odbywać się w ten sam sposób. TestNG umożliwia użycie asercji co zostało pokazane na poniższym obrazie testując ten sam, prosty kalkulator.



Rys. 11. Test prostego kalkulatora narzędziem TestNG

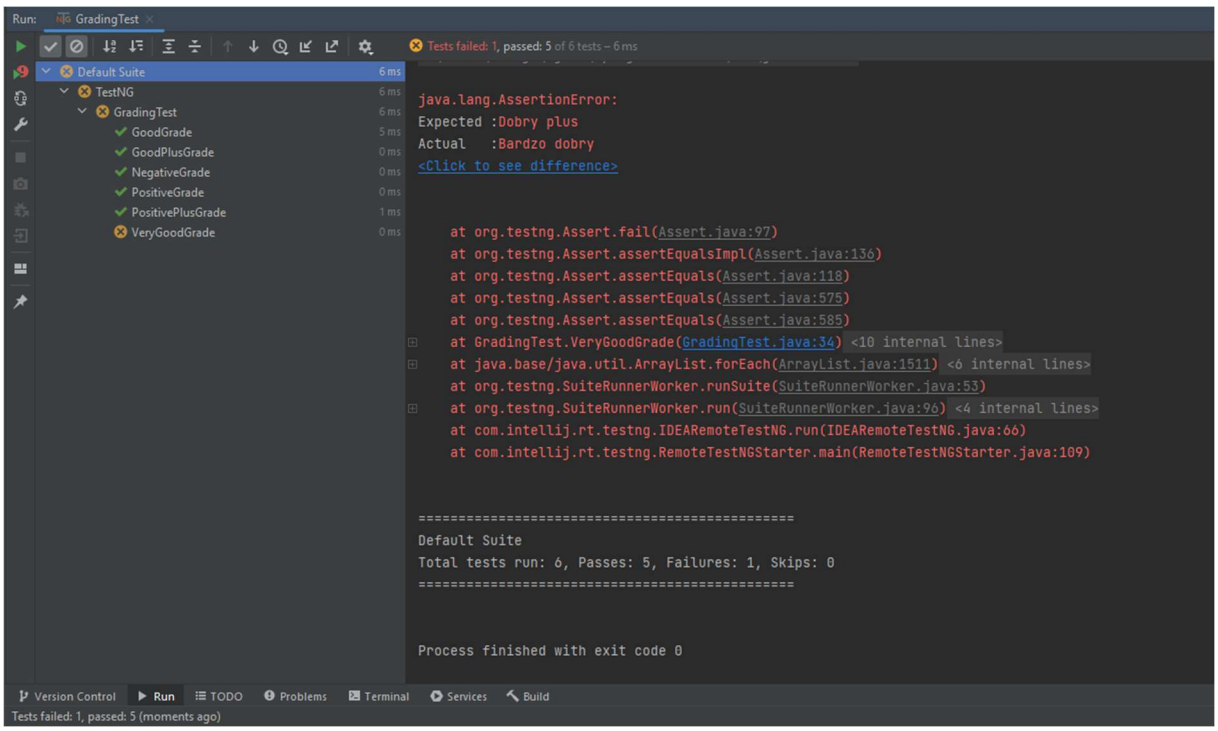


Rys. 12. Test prostego kalkulatora używając „assertTrue” narzędziem TestNG



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.

```java
import org.testng.Assert;
import org.testng.annotations.BeforeMethod;
import org.testng.annotations.DataProvider;
import org.testng.annotations.Test;

import static org.testng.Assert.assertEquals;

public class PrimeNumberTest {

    2 usages
    private PrimeNumberChecker primeNumberChecker;

    @BeforeMethod
    public void initialize() {
        primeNumberChecker = new PrimeNumberChecker();
    }

    1 usage
    @DataProvider(name = "test1")
    public static Object[][] primeNumbers() {
        return new Object[][] {{2, true}, {6, false}, {19, true}, {22, false}, {23, true}};
    }

    // This test will run 4 times since we have 5 parameters defined
    1 usage
    @Test(dataProvider = "test1")
    public void testPrimeNumberChecker(Integer inputNumber, Boolean expectedResult) {
        System.out.println("Parameterized number is: " + inputNumber);
        assertEquals(expectedResult, primeNumberChecker.validate(inputNumber));
    }
}
```

Rys.? Sparametryzowany test sprawdzający liczby pierwsze

```
Run:  PrimeNumberTest
Tests passed: 5 of 5 tests – 9 ms
Default Suite 9 ms
C:\Users\Szweju\.jdks\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. ? Wynik sparametryzowanego testu

1.3. Jtest

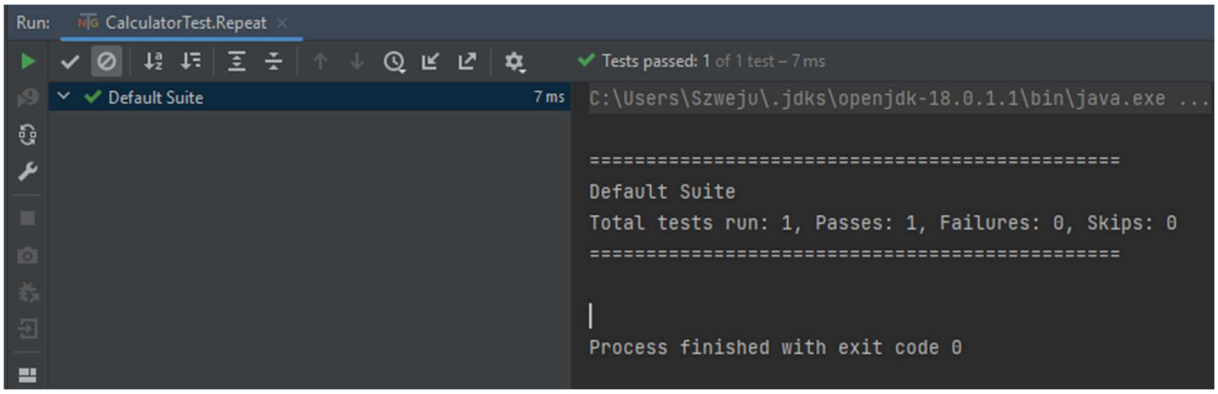
Parasoft Jtest to zintegrowane rozwiązanie do testowania programistycznego, które ma pomóc programistom Java poprawić jakość oprogramowania. Pomagając w procesie walidacji kodu Java i aplikacji, pomaga zapobiegać błędom, skuteczniej wykrywać błędy i zwiększać produktywność zespołu programistów. Jtest jest dostępny jako silnik, który można zintegrować z narzędziami do budowania (Maven, Ant lub Gradle) i infrastrukturą ciągłej integracji. Integracje te pozwalają zautomatyzować szeroki zakres najlepszych praktyk kodowania, generować raporty analityczne i automatycznie wysyłać informacje do Parasoft DTP. DTP może analizować zagregowane dane i przekształcać je w przydatne wyniki, które mogą pomóc w procesie doskonalenia kodu. Jtest Engine może również monitorować i zbierać dane pokrycia podczas wykonywania testów jednostkowych lub testów funkcjonalnych wykonywanych na uruchomionej aplikacji. Informacje o pokryciu i wyniki testów jednostkowych są wysyłane do DTP i pomagają ocenić jakość Twoich testów i aplikacji Java. Parasoft Jtest jest dostarczany z wtyczkami dla Eclipse i IntelliJ IDEA, które umożliwiają integrację Jtest z twoim IDE. Pozwala to wykorzystać możliwości silnika na pulpicie programistycznym, importować wyniki z DTP do IDE, a także wykorzystać dodatkowe możliwości dostępne dla Jtest Desktop, takie jak tworzenie artefaktów testów jednostkowych.

2. Porównanie szybkości

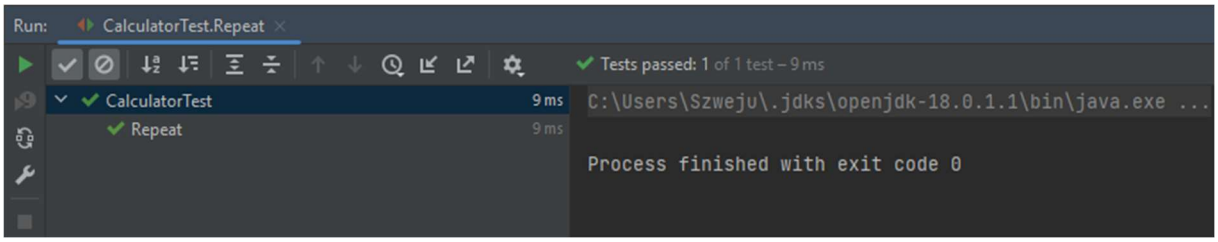
Wykonanie testu kalkulatora zostało uruchomione 100000 razy za pomocą pętli *for.*

```
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      }
```

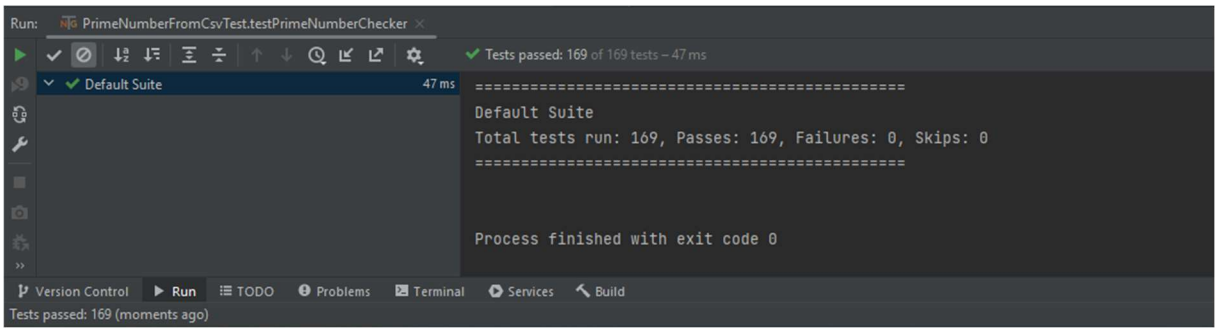
Rys. 15. Kod źródłowy testu.



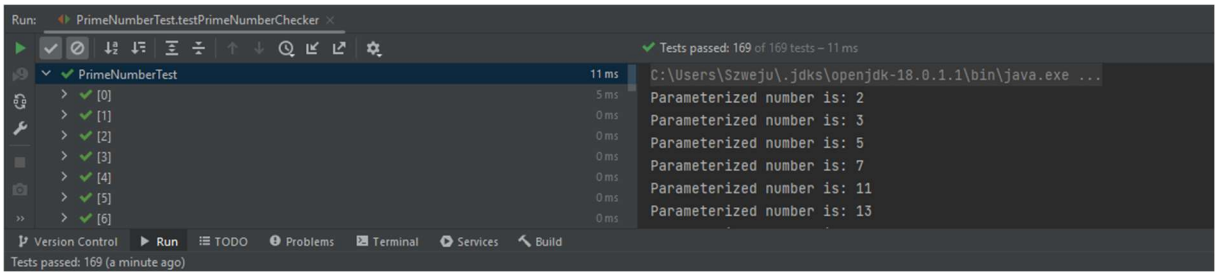
Rys. 16. Czas wykonania testu w TestNG.



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