

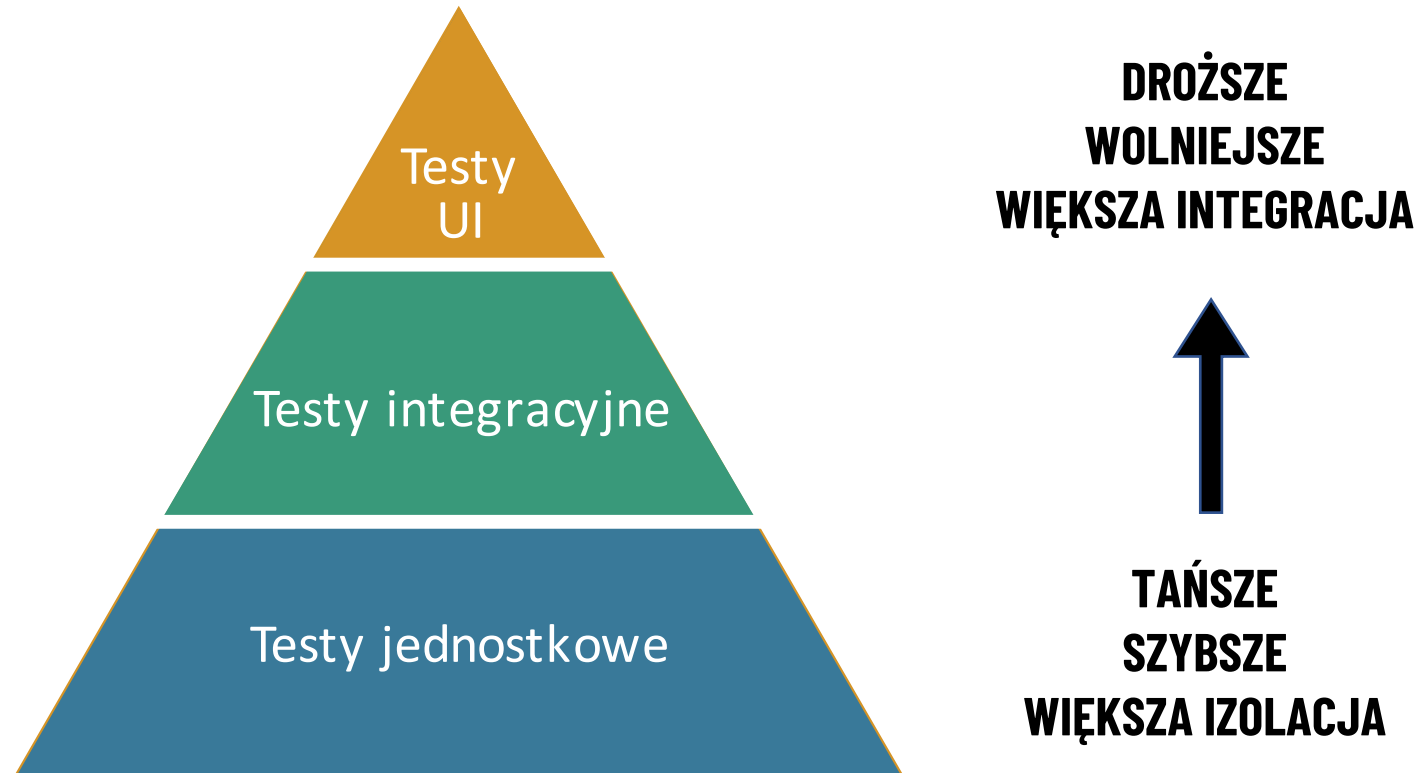
TESTY JEDNOSTKOWE

- Koncepcja testów jednostkowych
- Framework JUnit
- Zasady pisania testów

DEBUGGOWANIE A TESTOWANIE

- Debuggowanie polega na znajdowaniu, analizowaniu i usuwaniu konkretnych defektów w kodzie (zwykle przez programistów)
- Testowanie jest nastawione na ocenę poprawności działania programu, w tym jego zgodności z wymaganiami

PIRAMIDA TESTÓW



TESTY JEDNOSTKOWE (ang. Unit tests)

Także: *testy modułowe*, *testy komponentów*

- Skupiają się na modułach, które można przetestować w izolacji (klasach, metodach)
- Obejmują testowanie funkcjonalności (np. poprawności obliczeń, drzewa decyzji), rzadziej parametrów нефunkcjonalnych (np. zarządzania zasobami)
- Zwykle pisane przez programistę w oparciu o kod i szczegółowy projekt (“testowanie białoskrzynkowe”)

W tym wypadku każda metoda klasy Calculator stanowi oddzielny “moduł” (ang. unit) do przetestowania

```
public class Calculator {  
    public int add(int a, int b) {  
        return a + b;  
    }  
  
    public int subtract(int a, int b) {  
        return a - b;  
    }  
  
    public int multiply(int a, int b) {  
        return a * b;  
    }  
}
```

CO MOGĄ SPRAWDZAĆ TESTY JEDNOSTKOWE?

Przede wszystkim:

- Czy wartości zwracane z modułu są prawidłowe?
- Czy moduł rzuca wyjątki w odpowiednich sytuacjach?
- Czy moduł prawidłowo obsługuje różne dane i warunki wejściowe?

```
public int add(int a, int b) {  
    return a + b;  
}
```

```
public double divide(int a, int b) {  
    if (b == 0) {  
        throw new ArithmeticException("...")  
    } else {  
        return (double) a / b;  
    }  
}
```

```
public static double calculateTax(int income) {  
    if (income <= 50000) {  
        return 0.15 * income;  
    } else {  
        return 10750 + 0.31 * (income - 50000);  
    }  
}
```

ZASADA AAA, z ang. Arrange-Act-Assert

Arrange

Przygotowanie danych i warunków wejściowych,
np. stworzenie zmiennych i obiektów,
ustawienie konfiguracji

Act

Wykonanie testowanej operacji,
np. wywołanie funkcji, wysłanie zapytania

Assert

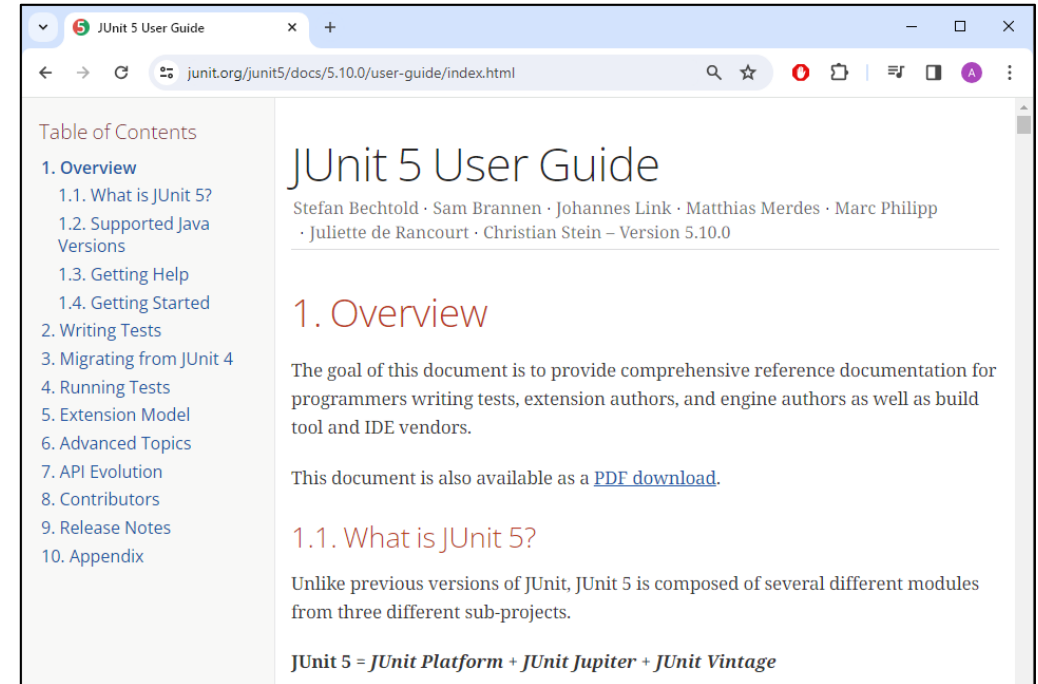
Sprawdzenie wyniku operacji

```
public class Calculator {  
    public int add(int a, int b) {  
        return a + b;  
    }  
}
```

```
@Test  
public void addTwoPositiveNumbers() {  
    Calculator c = new Calculator();  
  
    int result = c.add(2, 1);  
  
    assertEquals(result, 3);  
}
```

JUnit

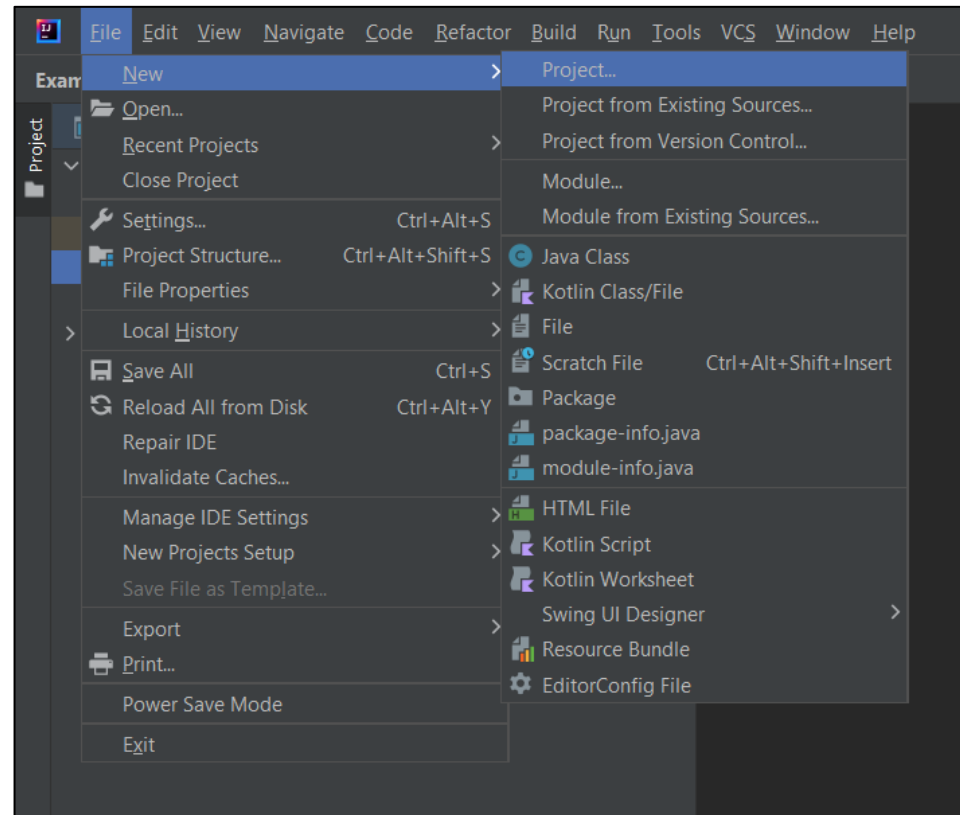
- framework open-source do testów jednostkowych w Javie
- najnowsza wersja, JUnit 5, składa się z trzech głównych części:
 - *JUnit Platform* (platforma do uruchamiania testów)
 - *JUnit Jupiter* (API do pisania testów)
 - *JUnit Vintage* (API do uruchamiania testów napisanych w starszych wersjach JUnit)



<https://junit.org/junit5/docs/5.10.0/user-guide/index.html>

TWORZENIE PROJEKTU W INTELIJ IDEA

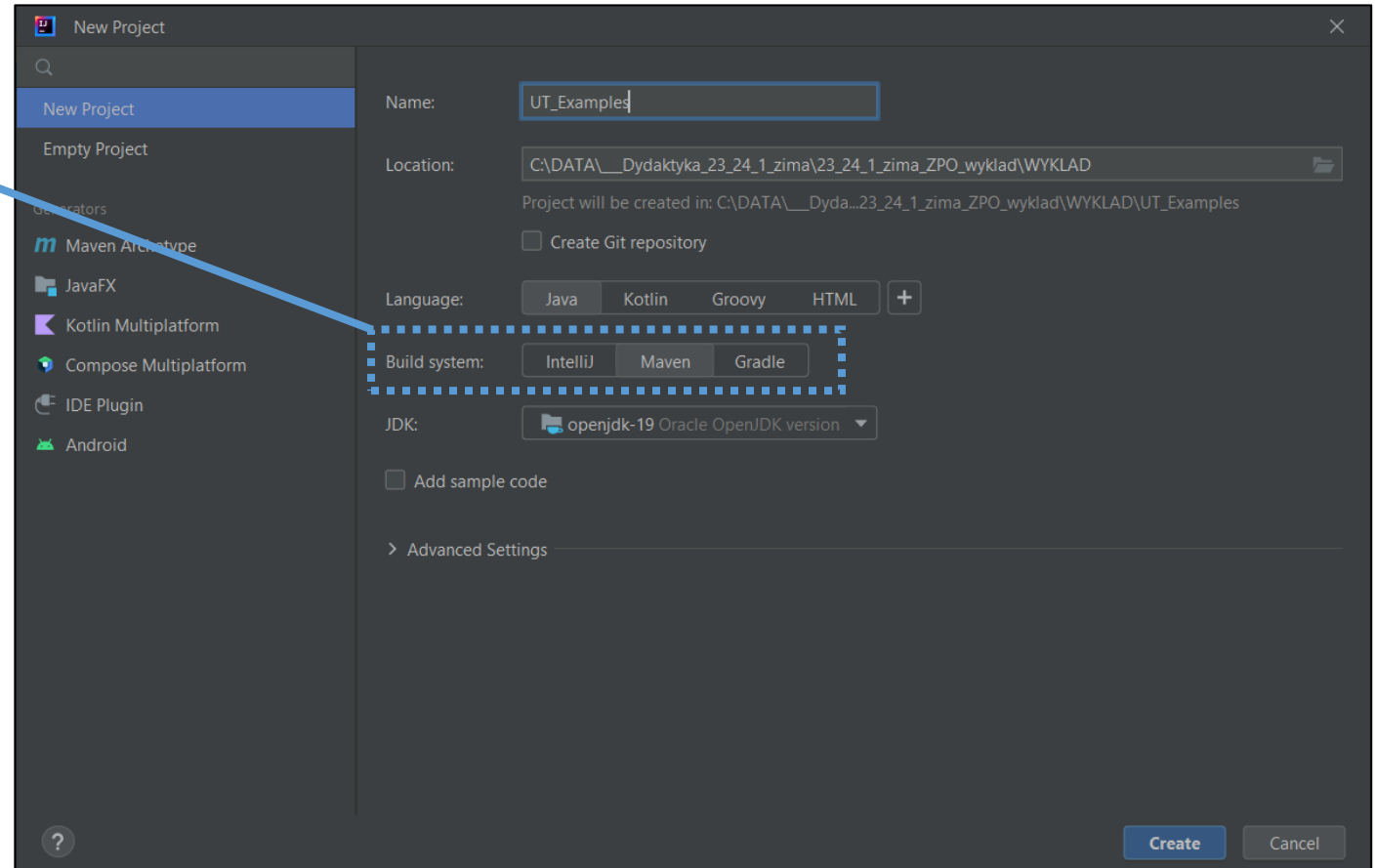
File > New > Project...



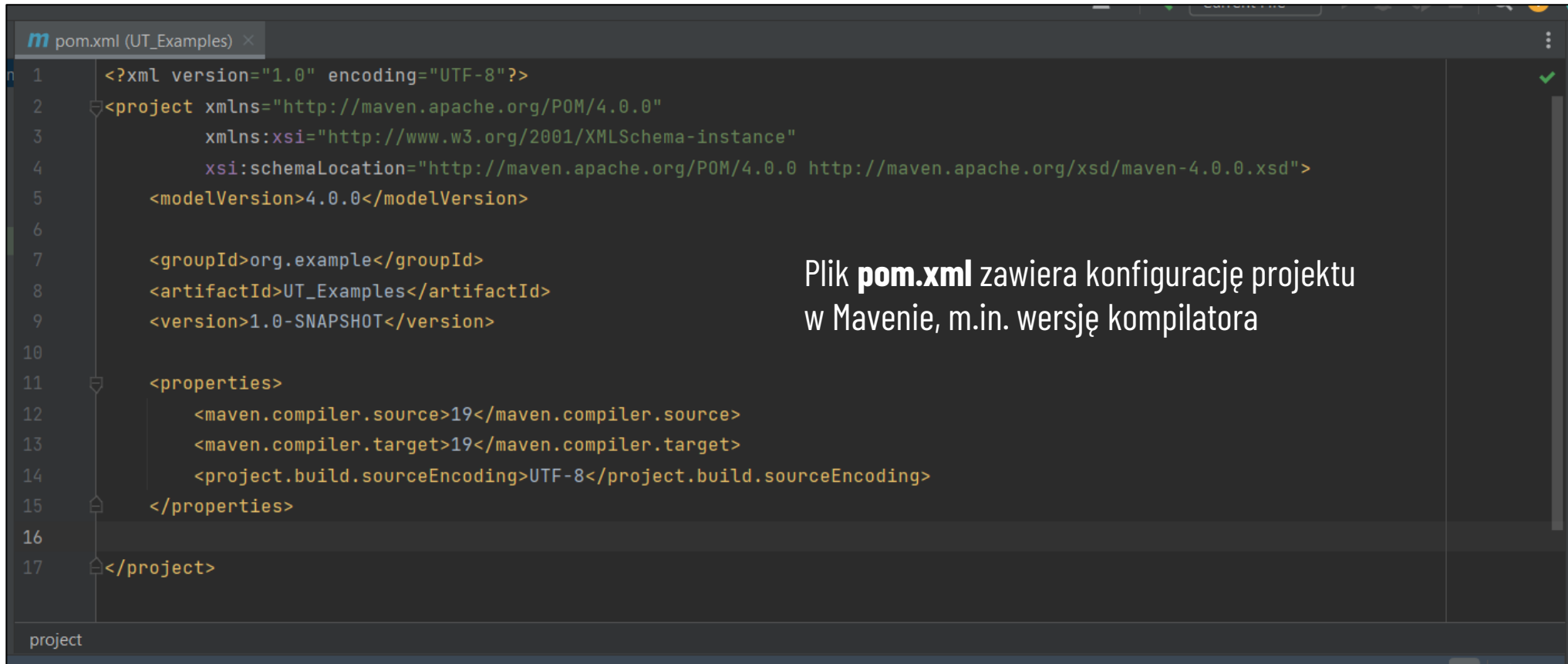
TWORZENIE PROJEKTU W INTELIJ IDEA

Build system: Maven

Maven jest narzędziem do zarządzania projektami – ułatwia konfigurację budowania projektu i zarządzanie zależnościami



TWORZENIE PROJEKTU W INTELIJ IDEA



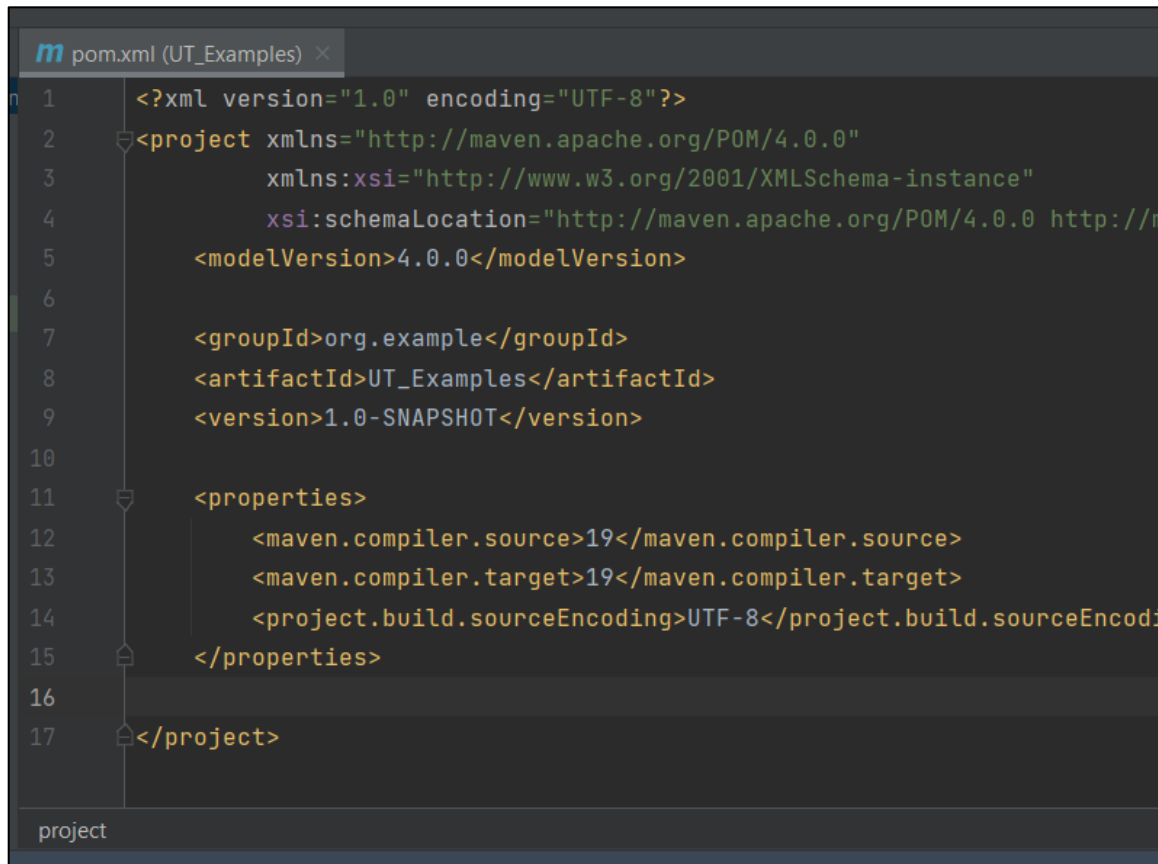
The screenshot shows the IntelliJ IDEA IDE with a file named `pom.xml (UT_Examples)` open. The XML content is as follows:

```
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>UT_Examples</artifactId>
9   <version>1.0-SNAPSHOT</version>
10
11   <properties>
12     <maven.compiler.source>19</maven.compiler.source>
13     <maven.compiler.target>19</maven.compiler.target>
14     <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
15   </properties>
16
17 </project>
```

Plik **pom.xml** zawiera konfigurację projektu w Mavenie, m.in. wersję kompilatora

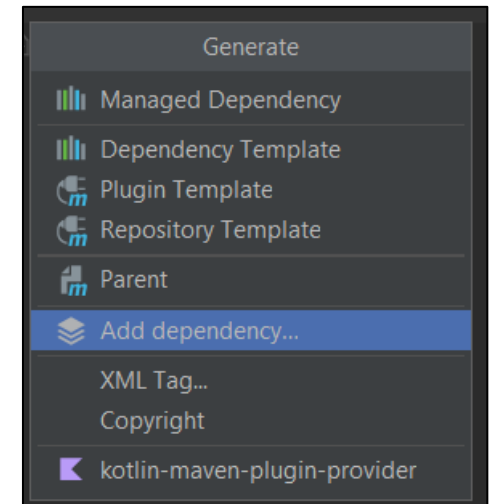
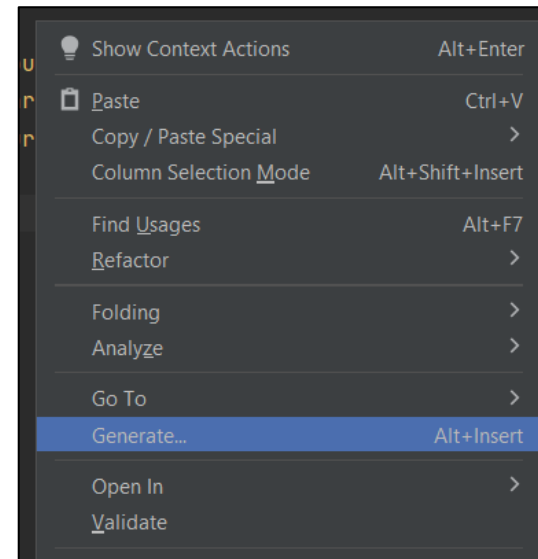
project

TWORZENIE PROJEKTU W INTELIJ IDEA



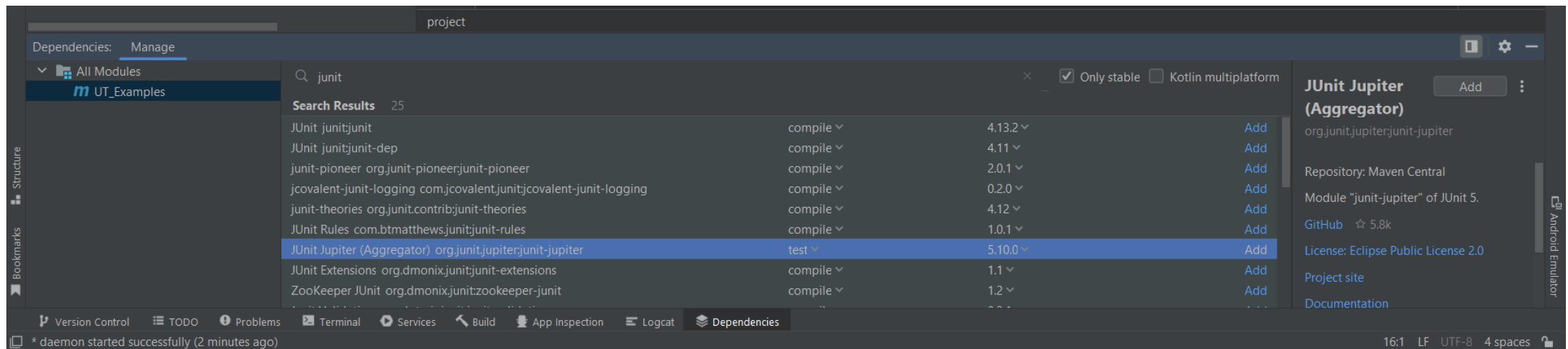
```
m pom.xml (UT_Examples) x
1  <?xml version="1.0" encoding="UTF-8"?>
2  <project xmlns="http://maven.apache.org/POM/4.0.0"
3      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4      xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/maven-v4_0_0.xsd">
5      <modelVersion>4.0.0</modelVersion>
6
7      <groupId>org.example</groupId>
8      <artifactId>UT_Examples</artifactId>
9      <version>1.0-SNAPSHOT</version>
10
11     <properties>
12         <maven.compiler.source>19</maven.compiler.source>
13         <maven.compiler.target>19</maven.compiler.target>
14         <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
15     </properties>
16
17 </project>
```

RMB > Generate... > Add dependency...

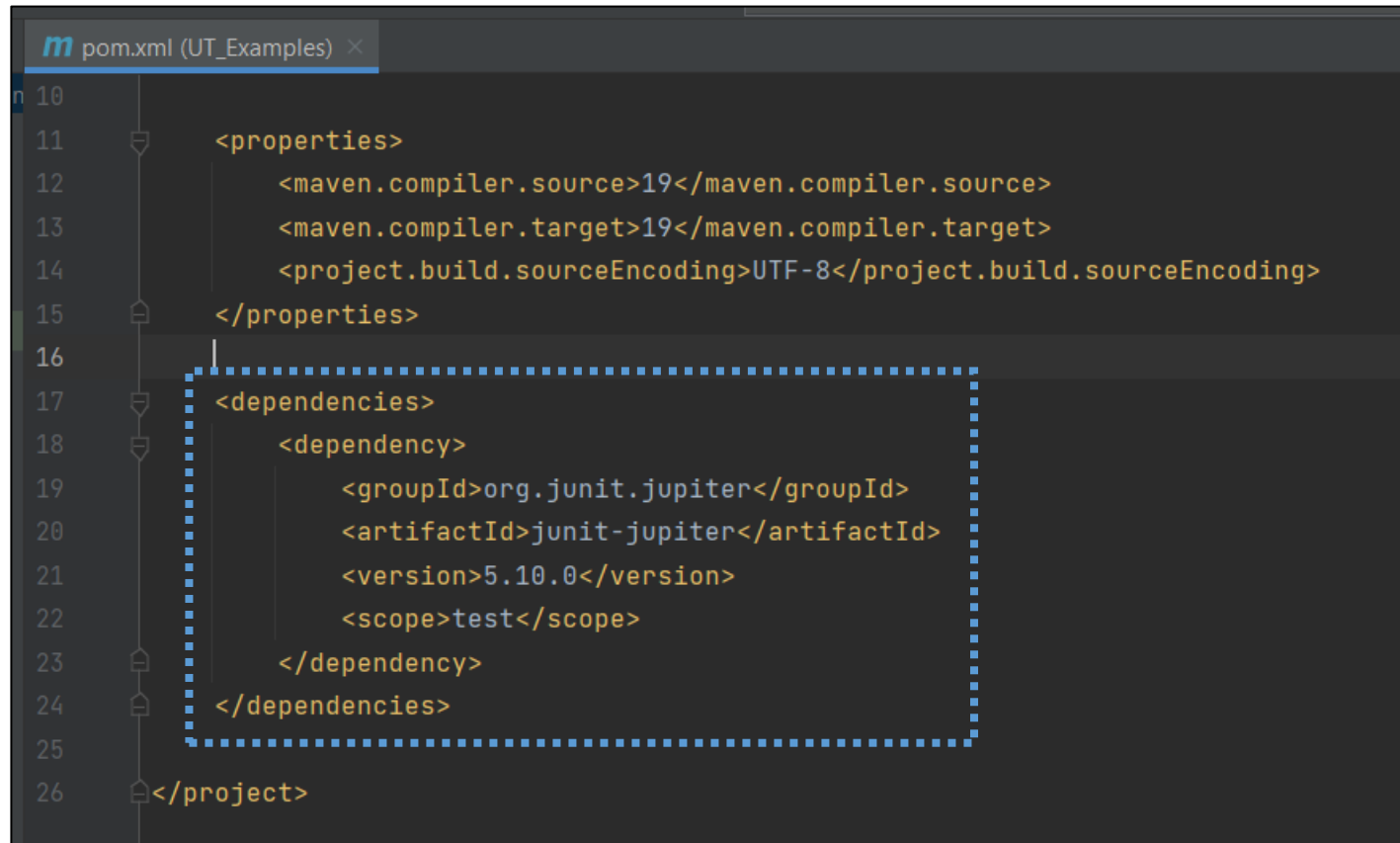


TWORZENIE PROJEKTU W INTELIJ IDEA

Search “junit” > Junit Jupiter (Aggregator) > konfiguracja “test” > Add



TWORZENIE PROJEKTU W INTELIJ IDEA

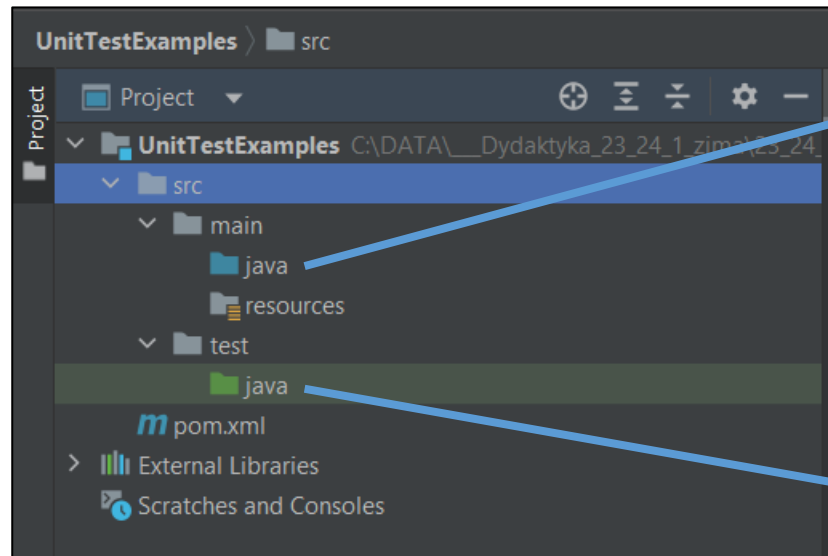


The screenshot shows the IntelliJ IDEA IDE with a Maven `pom.xml` file open. The file is titled "pom.xml (UT_Examples)". The XML content is as follows:

```
10
11     <properties>
12         <maven.compiler.source>19</maven.compiler.source>
13         <maven.compiler.target>19</maven.compiler.target>
14         <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
15     </properties>
16
17     <dependencies>
18         <dependency>
19             <groupId>org.junit.jupiter</groupId>
20             <artifactId>junit-jupiter</artifactId>
21             <version>5.10.0</version>
22             <scope>test</scope>
23         </dependency>
24     </dependencies>
25
26 </project>
```

A blue dashed rectangular box highlights the `<dependency>` block between lines 18 and 23, which specifies the JUnit Jupiter dependency.

STRUKTURA PROJEKTU



...\src\main\java

- kody źródłowe projektu

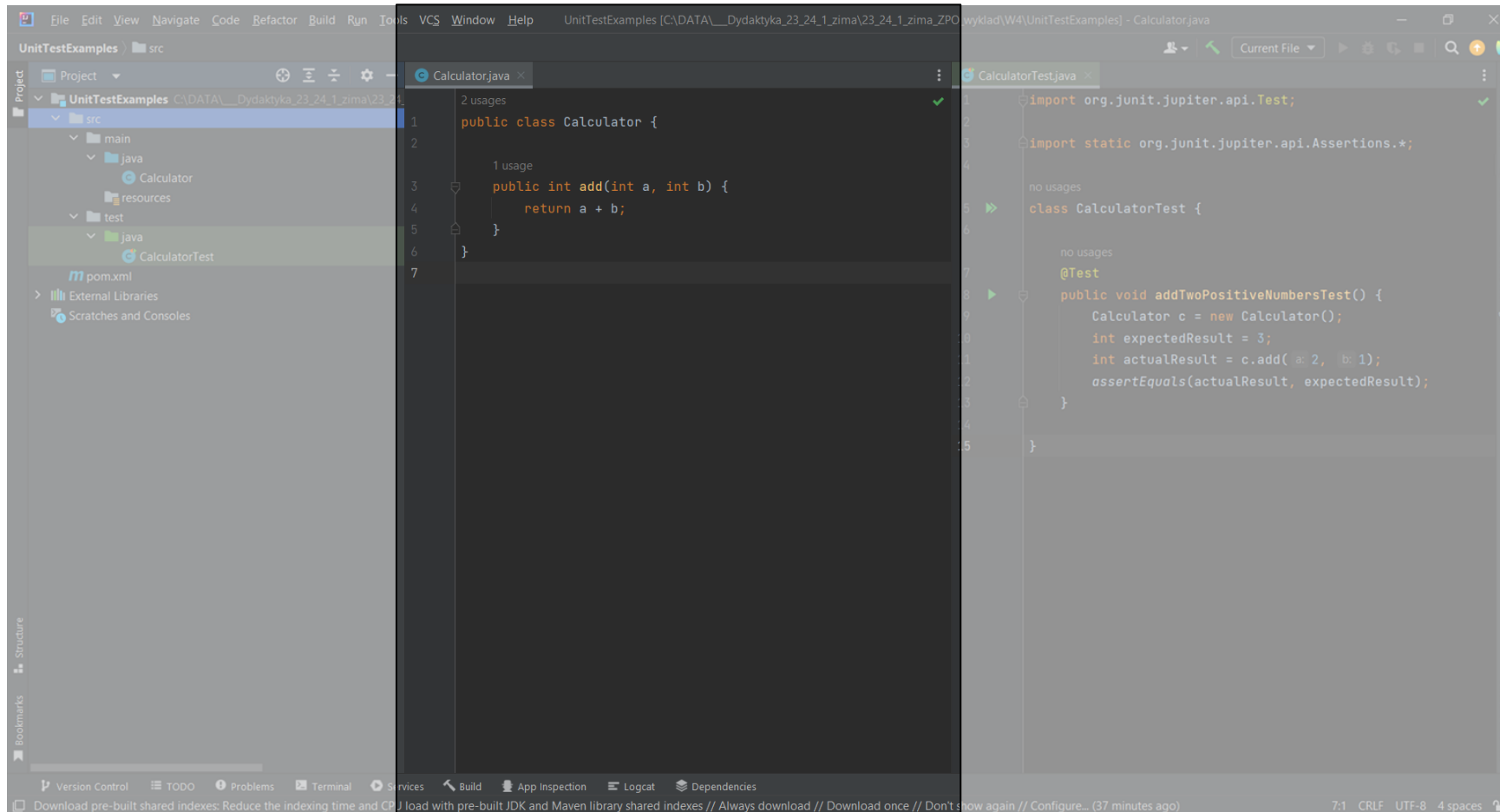
...\src\main\resources

- dodatkowe zasoby (pliki) projektu

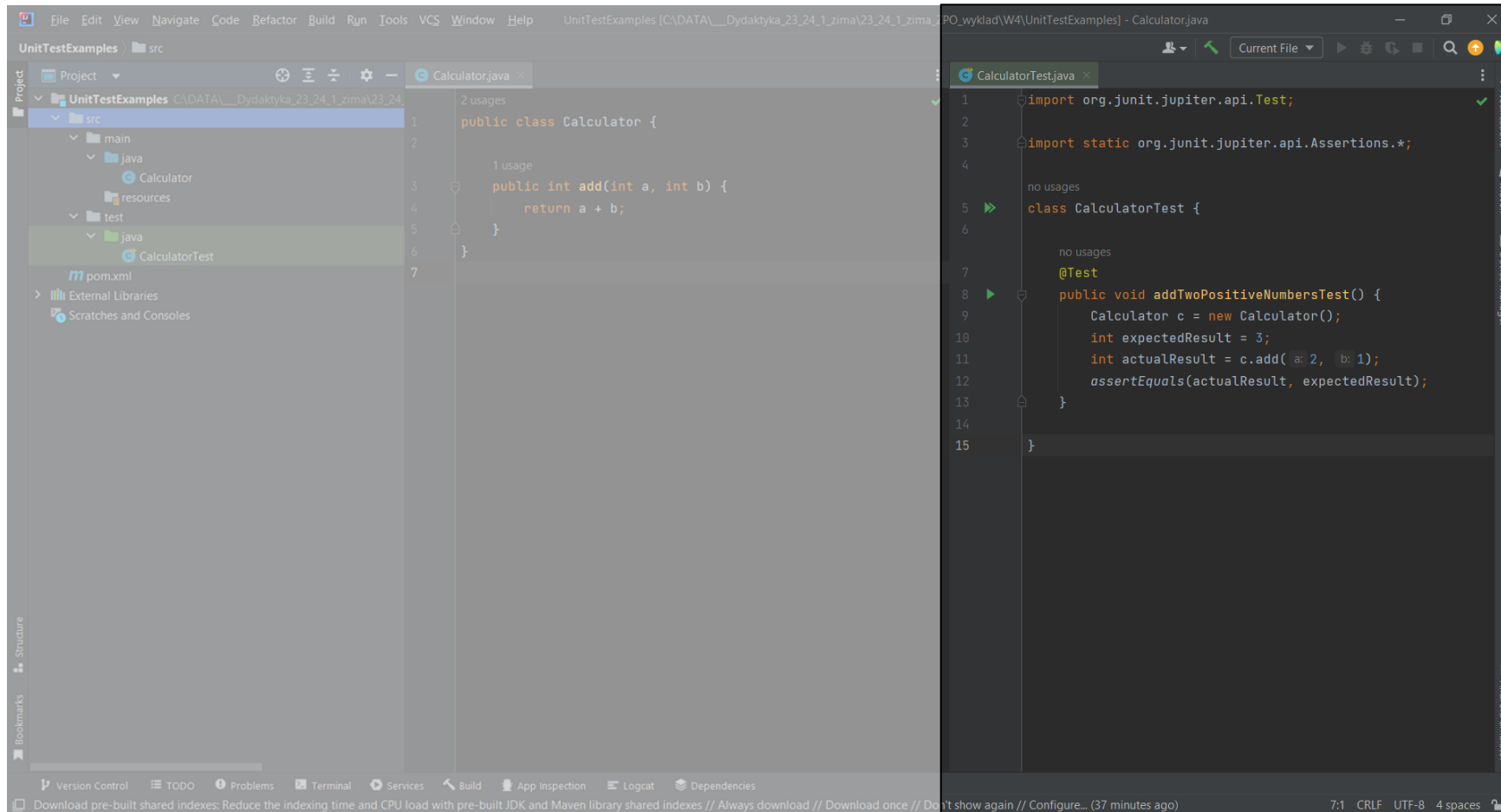
...\src\test\java

- kody testów jednostkowych

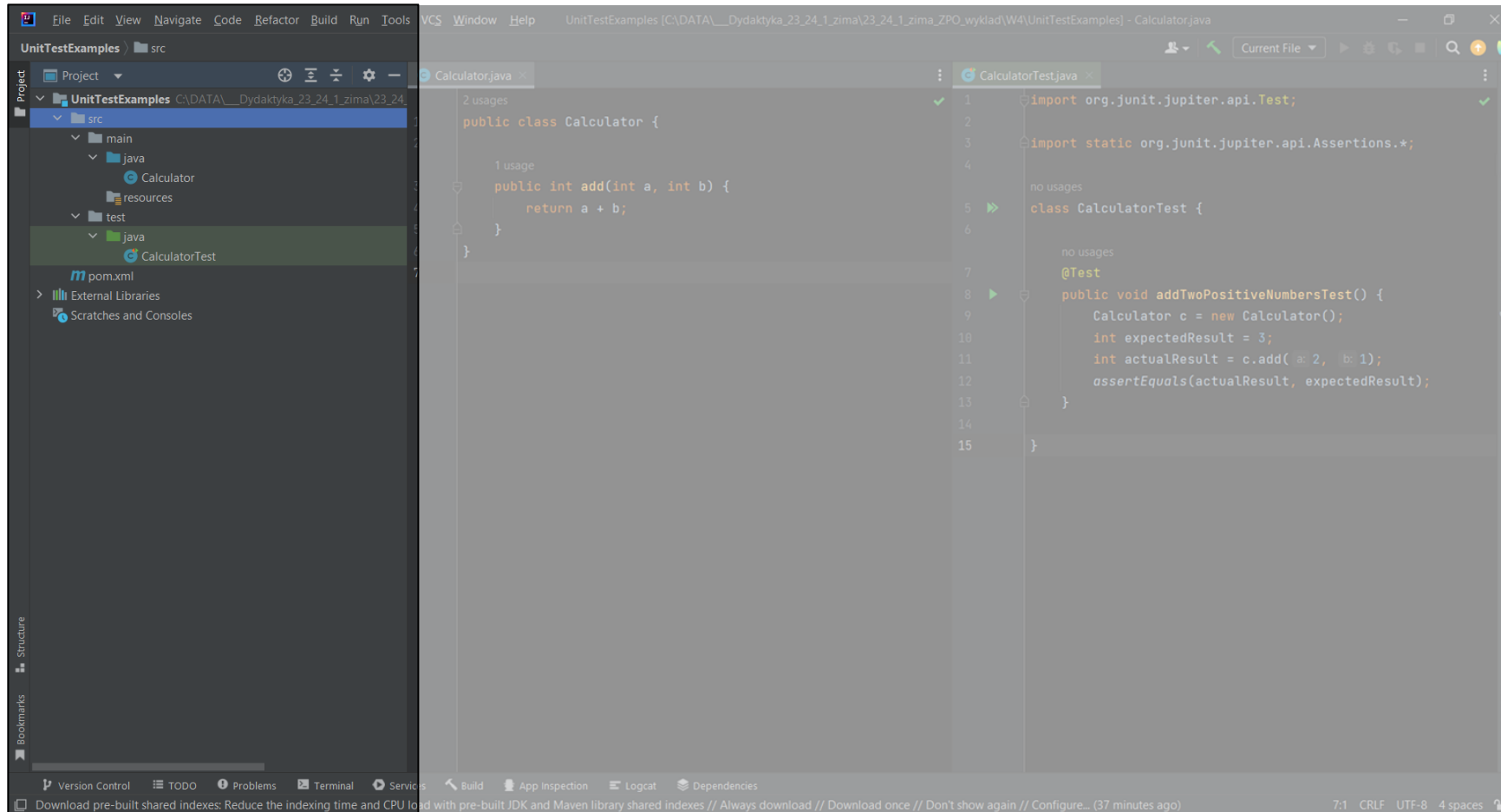
STRUKTURA PROJEKTU



STRUKTURA PROJEKTU



STRUKTURA PROJEKTU



STRUKTURA TESTU

```
public class Calculator {  
    public int add(int a, int b) {  
        return a + b;  
    }  
}
```

```
import org.junit.jupiter.api.Test;  
  
import static org.junit.jupiter.api.Assertions.*;  
  
class CalculatorTest {  
  
    @Test  
    public void addTwoPositiveNumbersTest() {  
        Calculator c = new Calculator();  
        int expectedResult = 3;  
        int actualResult = c.add(2, 1);  
        assertEquals(actualResult, expectedResult);  
    }  
}
```

STRUKTURA TESTU

```
public class Calculator {  
    public int add(int a, int b) {  
        return a + b;  
    }  
}
```

```
import org.junit.jupiter.api.Test;  
  
import static org.junit.jupiter.api.Assertions.*;  
  
class CalculatorTest {  
  
    @Test  
    public void addTwoPositiveNumbersTest() {  
        Calculator c = new Calculator();  
        int expectedResult = 3;  
        int actualResult = c.add(2, 1);  
        assertEquals(actualResult, expectedResult);  
    }  
}
```

Faza *Arrange* (także: *Given*)

Stworzenie obiektu testowanej klasy
i zdefiniowanie wartości oczekiwanej

```
Calculator c = new Calculator();  
int expectedResult = 3;
```

Faza *Act* (także: *When*)

Wykonanie testowanej operacji

```
int actualResult = c.add(2, 1);
```

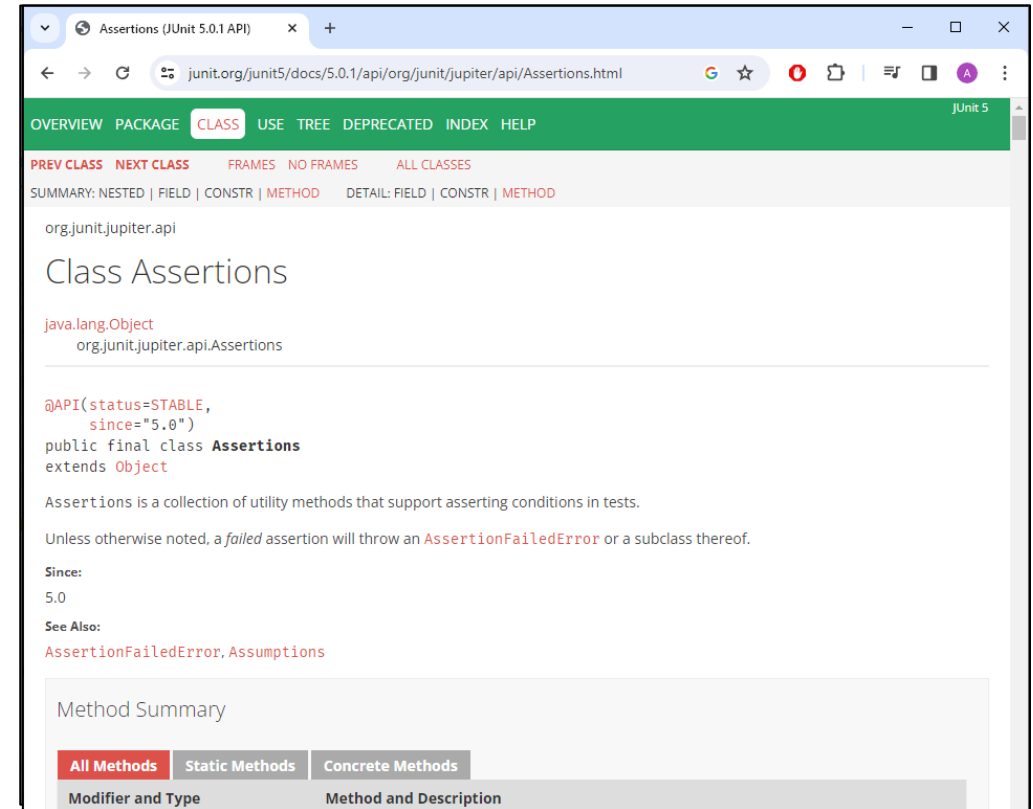
Faza *Assert* (także: *Then*)

Sprawdzenie wyników (asercja)

```
assertEquals(actualResult, expectedResult);
```

KLASA ASSERTIONS

- klasa JUnit 5 zawierająca metody użytkowe do wykonywania asercji w testach
- najważniejsze metody:
 - *assertEquals()*, *assertNotEquals()*
 - *assertArrayEquals()*, *assertIterableEquals()*
 - *assertTrue()*, *assertFalse()*
 - *assertNull()*, *assertNotNull()*
 - *assertThrows()*



<https://junit.org/junit5/docs/5.0.1/api/org/junit/jupiter/api/Assertions.html>

KLASA ASSERTIONS

- Dokładne porównanie wartości

```
int expected = 3;  
int actual = ...  
assertEquals(expected, actual);
```

```
String expected = "abcdef";  
String actual = ...  
assertEquals(expected, actual);
```

- Porównanie wartości zmiennoprzecinkowych z dopuszczalną różnicą

```
double expected = 0.1;  
double actual = ...  
double delta = 1e-15;  
assertEquals(expected, actual, delta);
```

KLASA ASSERTIONS

- Porównanie wartości logicznych

```
boolean actual = ...  
assertTrue(actual);
```

```
boolean actual = ...  
assertFalse(actual);
```

- Porównanie do null

```
String actual = ...  
assertNull(actual);
```

```
SomeClass actual = ...  
assertNotNull(actual);
```

KLASA ASSERTIONS

- Porównanie tablic i kolekcji

```
int[] expected = new int[]{1, 2, 3};  
int[] actual = ...  
assertArrayEquals(expected, actual);
```

```
List<String> expected = new ArrayList<>();  
expected.add("xxx");  
expected.add("yyy");
```

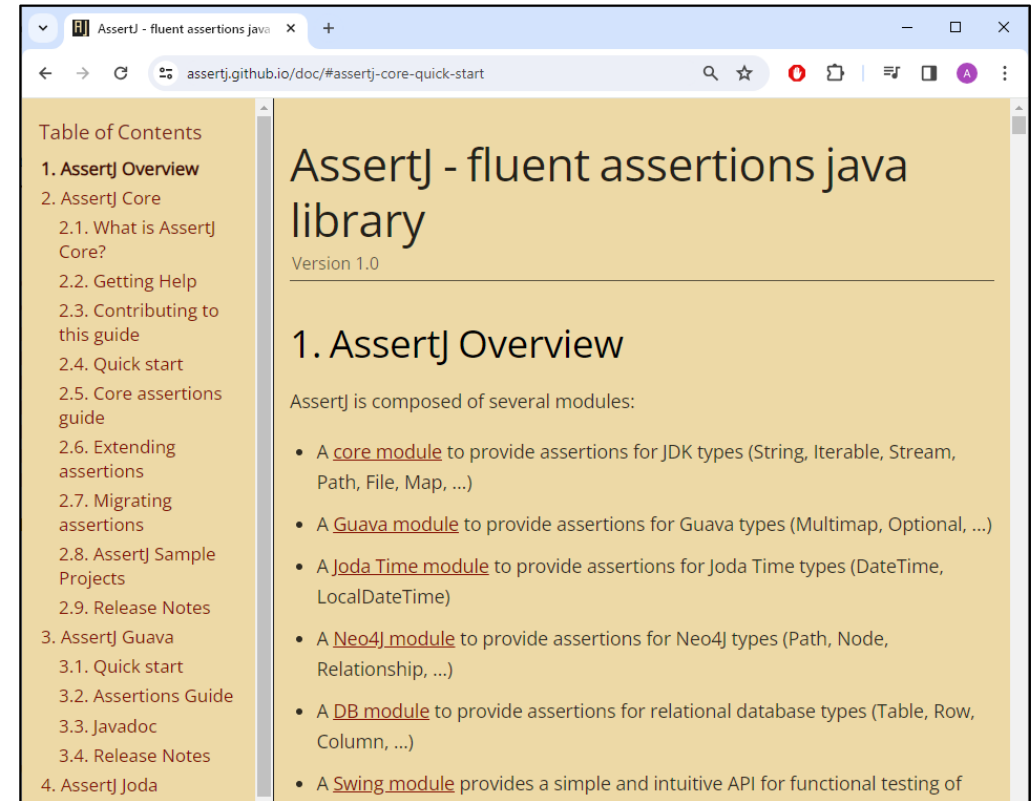
```
List<String> actual = ...  
assertIterableEquals(expected, actual);
```

- Sprawdzanie rzucanego wyjątku

```
Calculator c = new Calculator();  
  
assertThrows(ArithmeticException.class, () -> {  
    c.divide(1, 0);  
});
```

BIBLIOTEKA ASSERTJ

- biblioteka open-source do wykonywania asercji w testach
- oparta na metodzie *assertThat()*
- stanowi alternatywę dla klasy Assertions



<https://assertj.github.io/doc/#assertj-core-quick-start>

ADNOTACJE

Adnotacja @Test

- oznacza, że dana metoda jest testem jednostkowym

```
class CalculatorTest {  
  
    @Test  
    public void addTwoPositiveNumbersTest() {  
        Calculator c = new Calculator();  
        assertEquals(c.add(2, 1), 3);  
    }  
  
    @Test  
    public void addTwoNegativeNumbersTest() {  
        Calculator c = new Calculator();  
        assertEquals(c.add(-2, -1), -3);  
    }  
  
    @Test  
    public void addPositiveNegativeNumbersTest() {  
        Calculator c = new Calculator();  
        assertEquals(c.add(2, -3), 1);  
    }  
  
}
```

ADNOTACJE

Adnotacja @Test

- oznacza, że dana metoda jest testem jednostkowym do wykonania

Adnotacja @Disabled

- oznacza, że dany przypadek jest wyłączony z wykonywania (zwykle czasowo)
- może być używana do pojedynczych metod testowych lub całych klas

```
class CalculatorTest {  
  
    @Test  
    public void addTwoPositiveNumbersTest() {  
        Calculator c = new Calculator();  
        assertEquals(c.add(2, 1), 3);  
    }  
  
    @Test  
    public void addTwoNegativeNumbersTest() {  
        Calculator c = new Calculator();  
        assertEquals(c.add(-2, -1), -3);  
    }  
  
    @Disabled("Disabled until X happens")  
    @Test  
    public void addPositiveNegativeNumbersTest() {  
        Calculator c = new Calculator();  
        assertEquals(c.add(2, -3), 1);  
    }  
  
}
```

ADNOTACJE

Adnotacja **@BeforeAll**

- używana do oznaczania metod, które zostaną wykonane przed wszystkimi testami
- metoda oznaczona adnotacją *@BeforeAll* musi być statyczna

np. uruchamianie zewnętrznych serwisów,
ładowanie danych testowych

```
class StringProcessorTest {  
  
    private static String testData;  
  
    @BeforeAll  
    static void setUp() throws IOException {  
        String path = "testdata.txt";  
        StringBuilder data = new StringBuilder();  
        try (BufferedReader reader =  
            new BufferedReader(new FileReader(path)))  
        {  
            String line;  
            while ((line = reader.readLine()) != null) {  
                data.append(line);  
            }  
        }  
        testData = data.toString();  
    }  
  
    // test methods  
  
}
```

ADNOTACJE

Adnotacja **@BeforeEach**

- używana do oznaczania metod, które zostaną wykonane przed każdym testem

np. tworzenie obiektów, otwieranie połączenia z bazą danych

```
class CalculatorTest {  
  
    private Calculator calculator;  
  
    @BeforeEach  
    void setUpBeforeEach() {  
        calculator = new Calculator();  
    }  
  
    @Test  
    public void addTwoPositiveNumbersTest() {  
        assertEquals(calculator.add(2, 1), 3);  
    }  
  
    @Test  
    public void addTwoNegativeNumbers() {  
        assertEquals(calculator.add(-2, -1), -3);  
    }  
  
}
```

ADNOTACJE

Adnotacja @AfterEach

- używana do oznaczania metod, które zostaną wykonane po każdym teście

np. zamykanie połączeń, zwalnianie zasobów, cofanie zmian wprowadzonych przez test

```
public class ListProcessorTest {
    private List<Integer> list;

    @BeforeEach
    void setUpBeforeEach() {
        list = new ArrayList<>();
        list.add(1); list.add(2); list.add(3); }

    @Test
    void testRemove() {
        list.remove(Integer.valueOf(2));
        assertEquals(2, list.size()); }

    @Test
    void testAdd() {
        list.add(4);
        assertEquals(4, list.size()); }

    @AfterEach
    void tearDownAfterEach() {
        list.clear(); }
}
```

ADNOTACJE

Adnotacja @AfterAll

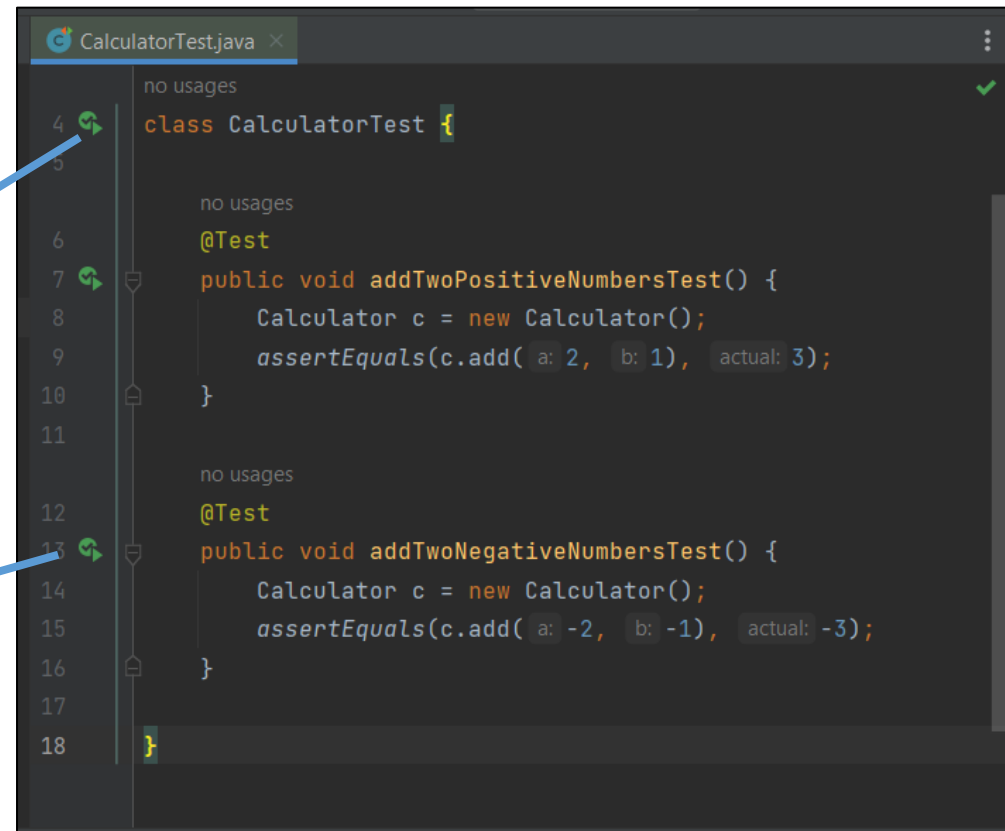
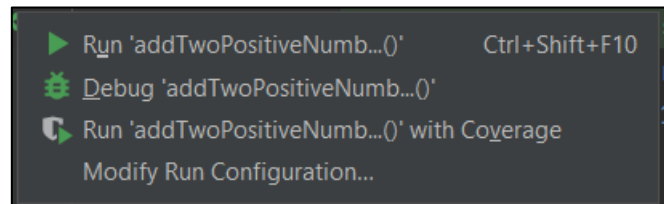
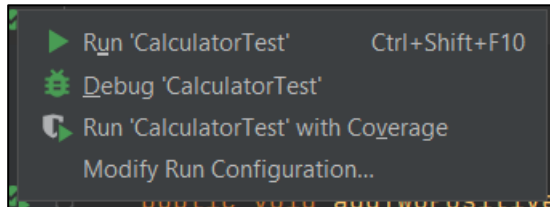
- używana do oznaczania metod, które zostaną wykonane po wszystkich testach
- metoda oznaczona adnotacją @AfterAll musi być statyczna

np. zatrzymanie zewnętrznych serwisów,
zwalnianie globalnych zasobów

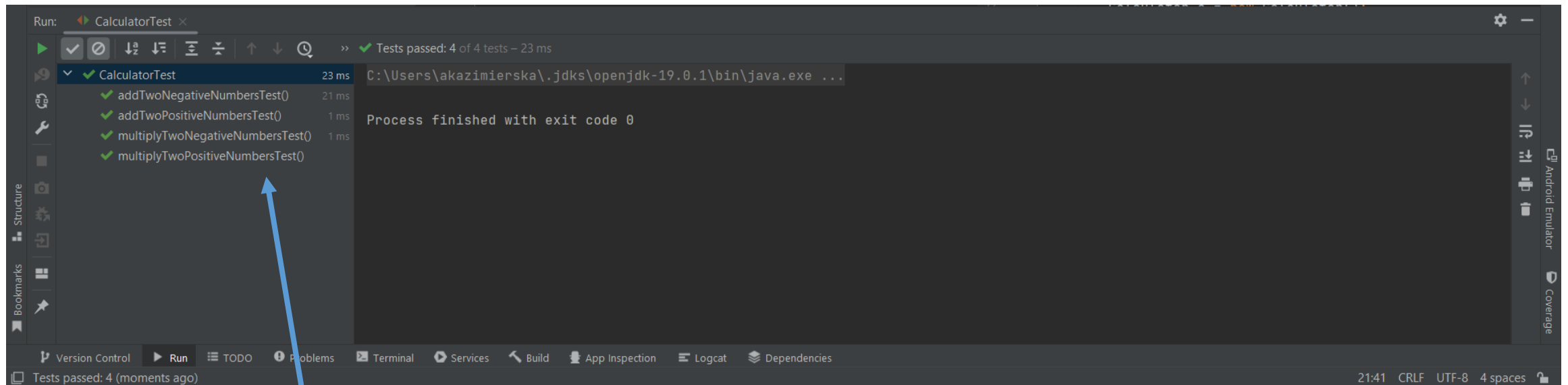
```
public class DatabaseManagerTest {  
    private static Connection connection;  
    private static final String DATABASE_URL = "testdb";  
  
    @BeforeAll  
    static void setUp() {  
        try {  
            connection = DriverManager.getConnection  
                (DATABASE_URL);  
        } catch (SQLException e) {  
            throw new RuntimeException("", e);  
        }  
  
        // test methods  
  
        @AfterAll  
        static void tearDown() {  
            try {  
                if (connection != null) {  
                    connection.close();  
                }  
            } catch (SQLException e) {  
                throw new RuntimeException("", e);  
            }  
        }  
    }  
}
```

URUCHAMIANIE TESTÓW

LMB na ikonie obok klasy
lub pojedynczego testu > Run

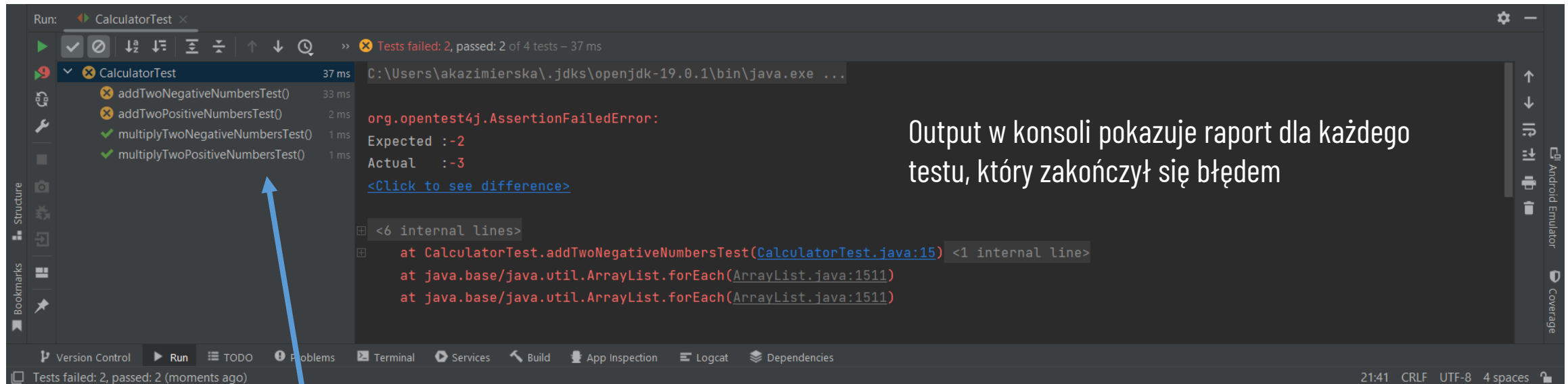


URUCHAMIANIE TESTÓW



Wszystkie testy zakończyły się sukcesem
(wyniki oczekiwane odpowiadały otrzymanym)

URUCHAMIANIE TESTÓW



Część testów zakończyła się błędem
(wyniki otrzymane były różne od oczekiwanych)

POKRYCIE KODU (ang. Code coverage)

```
public class Calculator {  
  
    public int add(int a, int b) {  
        return a + b + 1;  
    }  
  
    public int subtract(int a, int b) {  
        return a - b;  
    }  
  
    public int multiply(int a, int b) {  
        return a * b;  
    }  
  
    public double divide(int a, int b) {  
        if (b == 0) {  
            throw new ArithmeticException();  
        }  
        return (double) a/b;  
    }  
}
```

- określa, jak duża część kodu jest pokryta testami jednostkowymi
- zwykle wyrażane w % (klas, metod lub linii kodu)

POKRYCIE KODU (ang. Code coverage)

```
public class Calculator {  
  
    public int add(int a, int b) {  
        return a + b + 1;  
    }  
  
    public int subtract(int a, int b) {  
        return a - b;  
    }  
  
    public int multiply(int a, int b) {  
        return a * b;  
    }  
  
    public double divide(int a, int b) {  
        if (b == 0) {  
            throw new ArithmeticException();  
        }  
        return (double) a/b;  
    }  
}
```

- określa, jak duża część kodu jest pokryta testami jednostkowymi
- zwykle wyrażane w % (klas, metod lub linii kodu)

```
class CalculatorTest {  
  
    @Test  
    public void addTwoPositiveNumbersTest() {  
        Calculator c = new Calculator();  
        assertEquals(c.add(2, 1), 3);  
    }  
  
    @Test  
    public void addTwoNegativeNumbersTest() {  
        Calculator c = new Calculator();  
        assertEquals(c.add(-2, -1), -3);  
    }  
}
```

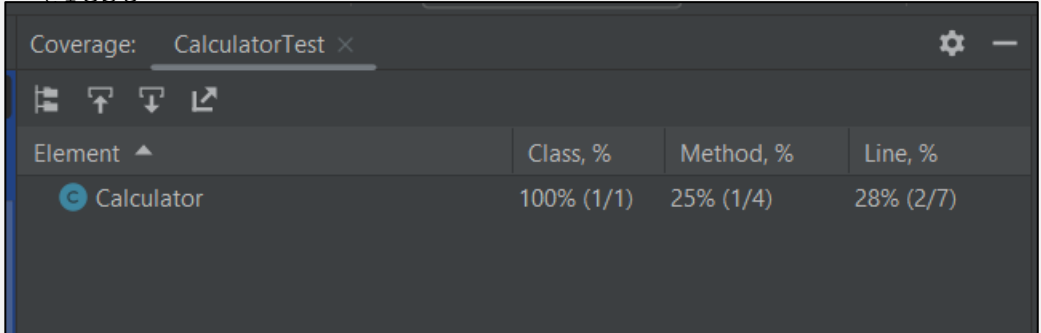
POKRYCIE KODU (ang. Code coverage)

```
public class Calculator {  
  
    public int add(int a, int b) {  
        return a + b + 1;  
    }  
  
    public int subtract(int a, int b) {  
        return a - b;  
    }  
  
    public int multiply(int a, int b) {  
        return a * b;  
    }  
  
    public double divide(int a, int b) {  
        if (b == 0) {  
            throw new ArithmeticException();  
        }  
        return (double) a/b;  
    }  
}
```

- określa, jak duża część kodu jest pokryta testami jednostkowymi
- zwykle wyrażane w % (klas, metod lub linii kodu)

```
class CalculatorTest {
```

```
    @Test
```



| Coverage: CalculatorTest × | | | |
|----------------------------|------------|-----------|-----------|
| Element ▲ | | | |
| | Class, % | Method, % | Line, % |
| Calculator | 100% (1/1) | 25% (1/4) | 28% (2/7) |

```
        assertEquals(C.add(-2, -1), -3);
```

```
    }
```

```
}
```

DOBRE PRAKTYKI

- Jeden test powinien sprawdzać jeden scenariusz (zwykle: zawierać jedną asercję)
- Dany test nie powinien zależeć od innych (kolejność wykonania testów nie powinna mieć znaczenia)
- Test powinien działać deterministycznie (jeżeli kod nie uległ zmianie, wynik testu powinien być taki sam)
- Nazwy testów powinny być znaczące i wykorzystywać tę samą konwencję

DOBÓR DANYCH TESTOWYCH

Podział na klasy równoważności

- polega na grupowaniu danych w tzw. klasy równoważności – wszystkie elementy danej klasy mają być przetwarzane w ten sam sposób
- przypadki testowe powinny uwzględniać co najmniej jeden przypadek dla każdej klasy

Przykład:

- *Funkcja ma sprawdzać długość nazwy użytkownika.*
- *Funkcja ma dopuszczać nazwy o długości od 3 do 10 znaków.*

| Klasa niepoprawna | Klasa poprawna | Klasa niepoprawna |
|----------------------|---------------------------------|----------------------|
| długość ≤ 2 | $3 \leq \text{długość} \leq 10$ | długość ≥ 11 |

np. "xx"

np. "abcdef"

np. "abcdefghijklxx"

DOBÓR DANYCH TESTOWYCH

Podział na klasy równoważności

- polega na grupowaniu danych w tzw. klasy równoważności – wszystkie elementy danej klasy mają być przetwarzane w ten sam sposób
- przypadki testowe powinny uwzględniać co najmniej jeden przypadek dla każdej klasy

Przykład:

- *Funkcja ma wyznaczać ocenę z testu na podstawie liczby zdobytych punktów (min. 0, maks. 50).*
- *Liczba punktów poniżej 25 oznacza ocenę niedostateczną, od 25 do 40 – dostateczną, powyżej 40 – dobrą.*

| Wynik | Klasa | np. |
|-------------------|-------------|-----|
| wynik < 0 | niepoprawna | -10 |
| 0 <= wynik < 25 | poprawna | 12 |
| 25 <= wynik <= 40 | poprawna | 34 |
| wynik > 40 | poprawna | 46 |
| wynik > 50 | niepoprawna | 78 |

DOBÓR DANYCH TESTOWYCH

Analiza wartości brzegowych

- rozszerzenie klas równoważności uwzględniające wartości graniczne każdej klasy
- przypadki testowe powinny uwzględniać trzy wartości dla każdej granicy: tuż poniżej, równą oraz tuż powyżej

Przykład:

- *Funkcja ma sprawdzać wiek użytkownika.*
- *Funkcja ma dopuszczać użytkowników w wieku od 18 do 55 lat.*

| Granica | Testowana wartość | Oczekiwany wynik |
|---------------------|-------------------|------------------|
| dolna: wiek = 18 | 17 | nie OK |
| | 18 | OK |
| | 19 | |
| górna: wiek = 55 | 54 | |
| | 55 | |
| | 56 | nie OK |

DOBÓR DANYCH TESTOWYCH

Testowanie instrukcji i decyzji

- każdy element instrukcji warunkowych generuje oddzielną gałąź drzewa decyzyjnego programu
- przypadki testowe powinny przynajmniej jeden przypadek dla każdej gałęzi

```
if (a > 0) {  
    if ((1 <= b) && (b <= 9)) {  
        // valid output  
    } else {  
        // invalid output 1  
    }  
} else {  
    // invalid output 2  
}
```

| a | b | Oczekiwany wynik |
|--------|-------------|------------------|
| a <= 0 | dowolne | invalid output 2 |
| a > 0 | b < 1 | invalid output 1 |
| a > 0 | b > 10 | invalid output 1 |
| a > 0 | 1 <= b <= 9 | valid output |

np.

a = -2, b = 1

a = 1, b = 0

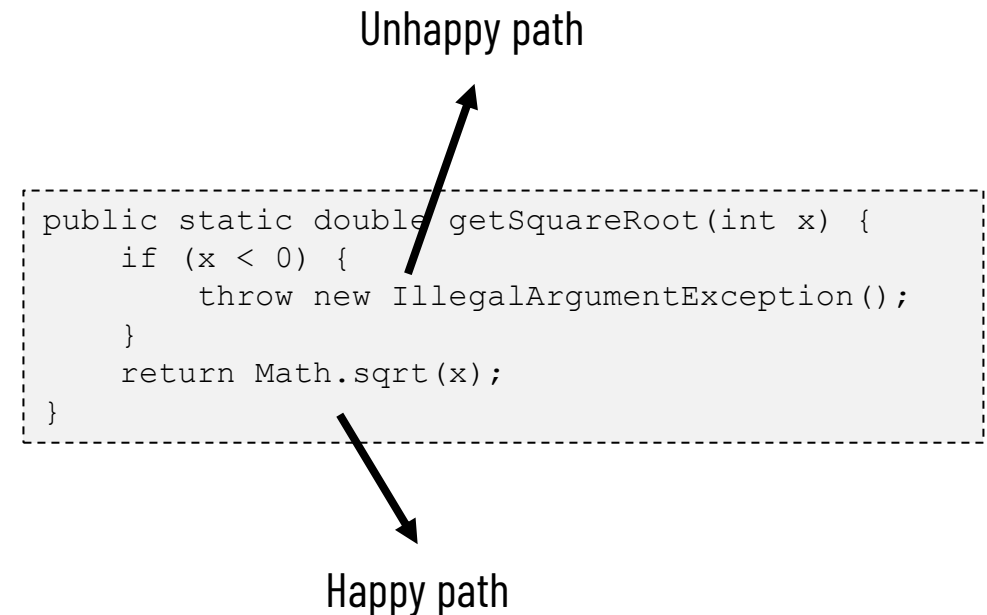
a = 1, b = 12

a = 1, b = 5

DOBÓR DANYCH TESTOWYCH

Testowanie błędów i wyjątków

- *happy path* to domyślna ścieżka wykonania program, która nie generuje wyjątków i nie zwraca błędów – test dla happy path przechodzi, jeżeli program nie wygeneruje błędu
- *unhappy paths* testują wyjątki i wartości niepoprawne – test dla unhappy path przechodzi, jeżeli program wygeneruje odpowiedni błąd lub wyjątek



TEST DOUBLES

- *test doubles* (“dublerzy”) to “fałszywe” reprezentacje zewnętrznych zależności, symulujące ich zachowanie na potrzeby testu
- mają na celu uniezależnienie testów od np. zmieniających się dat, istnienia plików o określonych ścieżkach, zapytań do zewnętrznych baz danych
- zwykle wyróżnia się pięć rodzajów test doubles: *dummy object*, *stub*, *mock*, *fake*, *spy*

TEST DOUBLES

```
public class TimeCalculator {
    private TimeProvider timeProvider;

    public TimeCalculator(TimeProvider timeProvider) {
        this.timeProvider = timeProvider;
    }

    public Duration calculateTimeDifference
        (LocalDateTime startTime)
    {
        LocalDateTime currentTime
            = timeProvider.getCurrentTime();
        return Duration.between(startTime, currentTime);
    }
}
```

```
public class TimeCalculatorTest {

    @Test
    void testTimeDifferenceCalculation() {
        TimeProvider timeProvider = new TimeProvider();

        TimeCalculator timeCalculator
            = new TimeCalculator(timeProvider);

        LocalDateTime startTime
            = LocalDateTime.of(2022, 12, 31, 18, 0);

        Duration timeDifference = timeCalculator
            .calculateTimeDifference(startTime);

        Duration expectedDifference = Duration.ofHours(18);
        assertEquals(expectedDifference, timeDifference);
    }
}
```

TEST DOUBLES

```
public class TimeCalculator {  
    private TimeProvider timeProvider;  
  
    public TimeCalculator(TimeProvider timeProvider) {  
        this.timeProvider = timeProvider;  
    }  
  
    public Duration calculateTimeDifference  
        (LocalDateTime startTime)  
    {  
        LocalDateTime currentTime  
            = timeProvider.getCurrentTime();  
        return Duration.between(startTime, currentTime);  
    }  
}
```

```
public class TimeCalculatorTest {  
  
    @Test  
    void testTimeDifferenceCalculation() {  
        TimeProvider timeProvider = new TimeProvider()  
  
        TimeCalculator timeCalculator  
            = new TimeCalculator(timeProvider);  
  
        LocalDateTime startTime  
            = LocalDateTime.of(2022, 12, 31, 18, 0);  
  
        Duration timeDifference = timeCalculator  
            .calculateTimeDifference(startTime);  
  
        Duration expectedDifference = Duration.ofHours(18);  
        assertEquals(expectedDifference, timeDifference);  
    }  
}
```

Problem:
wynik testu zależy od bieżącej daty i godziny

TEST DOUBLES

```
public class TimeStub implements TimeProvider {  
    private LocalDateTime fixedTime;  
  
    public TimeStub(LocalDateTime fixedTime) {  
        this.fixedTime = fixedTime;  
    }  
  
    @Override  
    public LocalDateTime getCurrentTime() {  
        return fixedTime;  
    }  
}
```

Rozwiązanie:

Tworzymy "fałszywy" obiekt, który zasymuluje działanie metody `getCurrentTime()`, ale zwróci predefiniowaną wartość

```
public class TimeCalculatorTest {  
  
    @Test  
    void testTimeDifferenceCalculation() {  
  
        LocalDateTime fixedTime  
            = LocalDateTime.of(2023, 1, 1, 12, 0);  
        TimeProvider timeProvider  
            = new TimeStub(fixedTime);  
  
        TimeCalculator timeCalculator  
            = new TimeCalculator(timeProvider);  
  
        LocalDateTime startTime  
            = LocalDateTime.of(2022, 12, 31, 18, 0);  
  
        Duration timeDifference = timeCalculator  
            .calculateTimeDifference(startTime);  
  
        Duration expectedDifference = Duration.ofHours(18);  
        assertEquals(expectedDifference, timeDifference);  
    }  
}
```