Instrukcja 9

Laboratorium 12

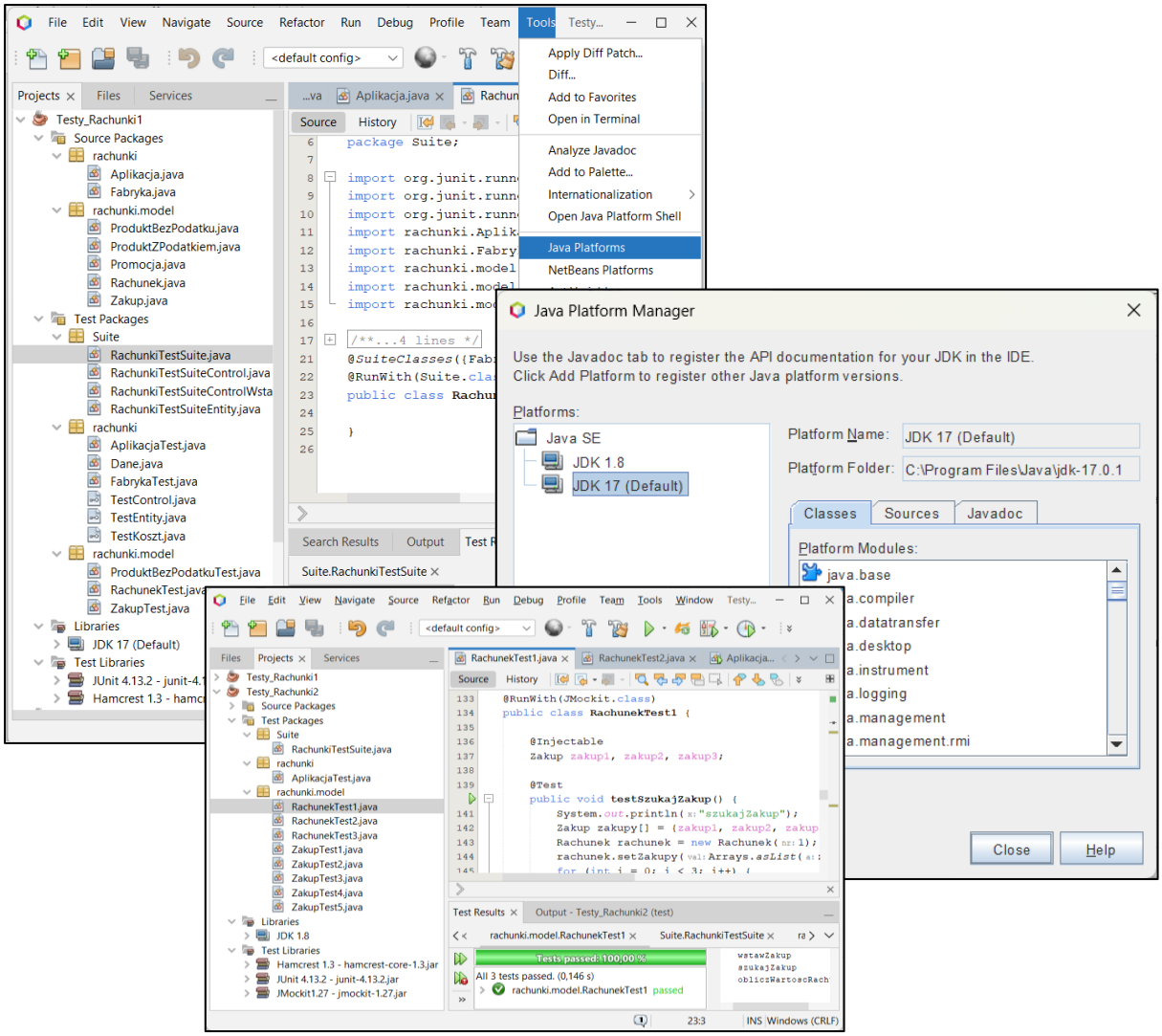
Testy jednostkowe z użyciem narzędzi JUnit oraz JMockit

Część I - JUnit 5 oraz JMockit 1.49

Cel laboratorium:
Nabycie umiejętności tworzenia testów jednostkowych za pomocą narzędzi JUnit oraz JMockit - (20 Most Popular Unit Testing Tools in 2022 (softwaretestinghelp.com), Mockito vs EasyMock vs JMockit | Baeldung**).**

1. Wg wskazówek podanych w **Dodatku 1**, należy zainstalować biblioteki ***JUnit 5*** i ***JMockit 1.49*** wykonując projekt typu **Maven** w środowisku ***Apache NetBeans 18***.
   Instalacja narzędzia ***Apache Netbeans 18*** z wybraną domyślną wersją ***Java SE 17*** przedstawiono na stronie przedmiotu:
   http://zofia.kruczkiewicz.staff.iiar.pwr.wroc.pl/index.php?id=INEK011.
   Należy wybrać taką wersję ***Java SE*** w tworzonym projekcie programistycznym, aby była nie była wyższa od wersji domyślnej ***Java SE*** w narzędziu ***Apache NetBeans 18***. Projekt powinien zawierać pakiet z klasami do testowania wykonanymi podczas lab2-11. Następnie, wg kolejnych wskazówek poniżej, należy dodawać testy ***JUnit*** wybranych klas. **Projekt z testami** oparty na bibliotekach ***JUnit 5*** oraz ***JMockit 1.49*** może używać platformy Java: ***Java SE 17***, dopasowanej do wersji domyślnej ***Java SE*** w narzędziu ***Apache NetBeans 18***.

   Poniżej pokazano sposób dodania dodatkowych platform ***Java SE***, które należy dodać do tworzonych aplikacji z zakładki ***Tools*** i pozycji ***Java Platforms.***



2. Należy wykonać test jednostkowy metod klasy, która stanowi klasę końcową w łańcuchu powiązań na diagramie klas lub/i może być powiązana w relacji 1 do 0..1 z inną klasą – podobnie jak klasa typu ***Fabryka*** lub klasy z rodziny ***ProduktBezPodatku***. Należy zastosować w metodach testowych metod z klasy ***Assertions*** biblioteki ***JUnit 5*** oraz adnotacje: ***Test, ParameterizedTest, CsvSource, MethodSource,*** dobraną adnotację ***@BeforeEach*** lub ***@BeforeAll, ExtendWith*** razem z obsługą wyjątków ***TestExecutionExceptionHandler.***
   Dane wzorcowe, wykorzystywane do weryfikacji wyników testowanych metod za pomocą metod klasy ***Assertions*** należy umieścić w dodatkowej klasie, podobnie jak klasa ***Dane*** z p.2.1 **Dodatku 1**. Przykłady testów podano w p.2.2 i p.2.3 **Dodatku 1**.
   W tabelce poniżej podano informację dotyczącą wyboru metod do testowania oraz przykładów rozwiązań.

Grupa	Liczba metod do testowania	Przykłady testowanych metod		Przykłady testów	
		Fabryka	rodzina **ProduktBezPodatku**	**FabrykaTest** (**p.2.2 Dodatek1**)	**ProduktBezPodatkuTest** (**p.2.3 Dodatek1**)
1 osoba	1	wykonajProdukt	obliczCeneBrutto	testWykonajProdukt	testObliczCeneBrutto
2 osoby	2	wykonajProdukt, obliczCeneBrutto		testWykonajProdukt,	testObliczCeneBrutto

3. Należy wykonać test jednostkowy metod klasy, która stanowi klasę w łańcuchu powiązań na diagramie klas lub/i może być powiązana w relacji „1 do 1” lub „1 do 1..*” z inną/innymi klasami – podobnie jak klasy typu ***Zakup*** z klasą z rodziny ***ProduktBezPodatku*** lub klasa ***Rachunek*** z klasą ***Zakup***.
   Należy zastosować w metodach testowych metody klasy ***Assertions*** z biblioteki ***JUnit 5*** oraz adnotacje: ***Test, ParameterizedTest, CsvSource, MethodSource,*** dobraną adnotację ***@BeforeEach*** lub ***@BeforeAll, TestMethodOrder(OrderAnnotation.class)*** i ***Order, ExtendWith*** razem z obsługą wyjątków np. ***TestExecutionExceptionHandler.***
   Przykłady testów podano w p.2.4 i p.2.5 **Dodatku 1**.
   Dane wzorcowe wykorzystywane do weryfikacji wyników testowanych metod za pomocą metod klasy ***Assertions***, należy umieścić w dodatkowej klasie (zdefiniowanej w p. 2), podobnie jak klasa ***Dane*** z p.2.1 **Dodatku 1**. Kryterium wyboru metod powinno uwzględniać fakt, że metody wybrane w p.1 są wywoływane w metodach klas wybranych w p.2.
   Poniżej, w tabelce poniżej podano informację dotyczącą wyboru metod do testowania oraz przykładów rozwiązań.

Grupa	Liczba metod do testowania	Przykłady testowanych metod		Przykłady testów	
		Zakup	**Rachunek**	**ZakupTest** (**p.2.4 Dodatek1**)	**RachunekTest** (**p.2.5 Dodatek1**)
1 osoba	1	obliczWartosc	wstawZakup,	testObliczWartosc	test1WstawZakup
2 osoby	2	-	wstawZakup, obliczWartoscRachunku	-	test1WstawZakup test2ObliczWartoscRachunku

4. Należy wykonać testy jednostkowe wybranych metod klasy opartej na wzorcu ***Fasada,*** podobnie jak klasa ***Aplikacja***. Wybrane metody tej klasy do testowania powinny wywoływać wybrane metody z p.2 lub p.1.
   Należy zastosować w metodach testowych metody klasy ***Assertions*** z biblioteki ***JUnit 5*** oraz adnotacje: ***Test, ParameterizedTest, CsvSource, MethodSource,*** dobraną adnotację ***@BeforeEach*** lub ***@BeforeAll, TestMethodOrder(OrderAnnotation.class)*** i ***Order, ExtendWith*** razem z obsługą wyjątków np. ***TestExecutionExceptionHandler***. Przykłady testów podano w p.2.6 **Dodatku 1**. Poniżej, w tabelce podano informację dotyczącą wyboru metod do testowania oraz przykładów rozwiązań.

Grupa	Liczba metod do testowania	Przykłady testowanych metod **Aplikacja** (p.2.6 Dodatek1)	Przykłady testów **AplikacjaTest** (p.2.6 Dodatek1)
1 osoba	2	dodajProdukt, wstawZakup,	test1DodajProdukt, test2WstawZakup
2 osoby	3	dodajProdukt, podajWartoscRachunku, wstawZakup,	test1DodajProdukt, test3PodajWartoscRachunku, test2WstawZakup

5. Należy wykonać zestawy testów, podobnie jak pokazano w p.2.7 **Dodatku 1** stosując adnotację ***Tag*** w klasach z metodami testującymi, wykonanych w p. 1, 2, 3 oraz ***@Suite, @SelectPackages, @IncludeTags, @ExcludeTags, @SelectClasses*** i dobraną adnotację ***@BeforeEach*** lub ***@BeforeAll***. Poniżej, w tabelce podano informację dotyczącą wyboru metod do testowania oraz przykładów rozwiązań.

Grupa	Zestaw wszystkich testów	Przykłady zestawu testów wyznaczonych wg kategorii (adnotacja Category) (p.2.7 Dodatek1)
1 osoba	RachunkiTestSuite	RachunkiTestSuiteEntity RachunkiTestSuiteControl
2 osoby	RachunkiTestSuite	RachunkiTestSuiteEntity RachunkiTestSuiteControl RachunkiTestSuiteControlWstaw

6. Należy metody, wybrane do testowania w jednym punktów instrukcji 3-4, przetestować wykorzystując mechanizm symulowania obiektów powiązanych. Dodatkowo, należy kierować się przykładami z p. 3.1-3.3 **Dodatku 1** oraz przykładami z **Dodatku 2** instrukcji**.** Należy uwzględnić proponowane tam elementy symulacji technologii ***JMockit***. Poniżej, w tabelce podano przykłady testów, wykonanych przez grupy: jedno- i dwuosobowe.

Grupa Liczba osób	Liczba metod do testowania	Przykłady metod: 1-przykład		Przykłady metod: 2-przykład		Przykłady testów –
		Przykłady symulowanych metod	Przykłady testowanych metod	Przykłady symulowanych metod	Przykłady testowanych metod	
		Klasy z rodziny ProduktBezPodatku	Klasa Zakup	Klasa Zakup	Klasa Rachunek	
1	2	equals	equals	getIlosc, dodajIlosc	wstawZakup,	**p. 3.1 Dodatek 1** lub **p. 3.2 Dodatek 1**
		obliczCeneBrutto, getPodatek	obliczWartosc	obliczWartosc	obliczWartosc Rachunku	

Grupa Liczba osób	Liczba metod do testowania	Przykłady metod: 1-przykład		Przykłady metod: 2-przykład		Przykłady testów –
		Przykłady symulowanych metod	Przykłady testowanych metod	Przykłady symulowanych metod	Przykłady testowanych metod	
		Klasa Zakup	Klasa Rachunek	Klasa Fabryka	Klasa Aplikacja	
2	4	getIlosc, dodajIloscProduktu	wstawZakup,	wykonajProdukt bez generowania wyjatku	dodajProdukt bez generowania wyjątku	**p. 3.2 Dodatek 1**, **p.3.3 Dodatek 1**
		obliczWartosc	obliczWartosc Rachunku	wykonajProdukt z generowaniem wyjatku	dodajProdukt z generowaniem wyjątku	

Dodatek 1

Testy jednostkowe oprogramowania "System sporządzania rachunków”

1. **Modyfikacje kodu przedstawionego w Dodatku 1 z instrukcji laboratoryjnych 5-7 -** po wykonaniu projektu wg informacji podanej w p. 2 (bieżący Dodatek 1) i umieszczeniu tam kodu przedstawianego w bieżącym Dodatku 1 z instrukcji laboratoryjnych 5-7, należy dokonać podanych dalej w p.1.1-1.2 modyfikacji tego kodu.

1.1. **Dodanie generowania wyjątku** w przypadku niepoprawnej wartości pierwszego elementu tablicy ***dane*** jako parametru metody ***WykonajProdukt*** klasy ***Fabryka*** - zmiana definicji tej klasy podanej w instrukcji laboratoryjnej 6. **Pełna walidacja poprawności formatu i wartości danych wejściowych powinna być realizowana przez *Warstwę klienta* aplikacji!**

```
public class Fabryka {
  public ProduktBezPodatku wykonajProdukt(String dane[]) {
      ProduktBezPodatku produkt = null;
      Promocja promocja;
      switch (Integer.parseInt(dane[0])) {
        case 0:
          produkt = new ProduktBezPodatku(dane[1], Float.parseFloat(dane[2]));
          break;
        case 1:
          promocja = new Promocja(Float.parseFloat(dane[3]));
          produkt = new ProduktBezPodatku(dane[1], Float.parseFloat(dane[2]), promocja);
          break;
        case 2:
          produkt = new ProduktZPodatkiem(dane[1], Float.parseFloat(dane[2]), Float.parseFloat(dane[3]));
          break;
        case 3:
          promocja = new Promocja(Float.parseFloat(dane[4]));
          produkt = new ProduktZPodatkiem(dane[1], Float.parseFloat(dane[2]), Float.parseFloat(dane[3]),
                                          promocja);
          break;
          default:
          throw new IllegalFormatCodePointException(0); //generowanie wyjątku z powodu niepoprawnej
                                                        //wartości elementu tablicy dane o indeksie 0.
      }
      return produkt; }
}
```

W klasie ***Aplikacja*** dodano do definicji metod, wywołujących metodę klasy ***Fabryka*** dodano klauzulę **throws IllegalFormatCodePointException -** zmiana definicji podanej w instrukcjach 6 i 7:

```
public void dodajProdukt (String dane[]) throws IllegalFormatCodePointException  //instrukcja 6

public void wstawZakup (int nr, int ile, String dane[]) throws IllegalFormatCodePointException //instrukcja 7

public static void main(String args[]) throws IllegalFormatCodePointException       // instrukcje 6 i 7
```

1.2. **Dodatkowo, klasy pakietu *rachunki* i *rachunki.model*, podane w części *Dodatek* 1 instrukcji laboratoryjnej 5 powinny być klasami publicznymi**:

```
public class Rachunek
public class Zakup
public class ProduktBezPodatku
public class ProduktZPodatkiem
public class Promocja
```

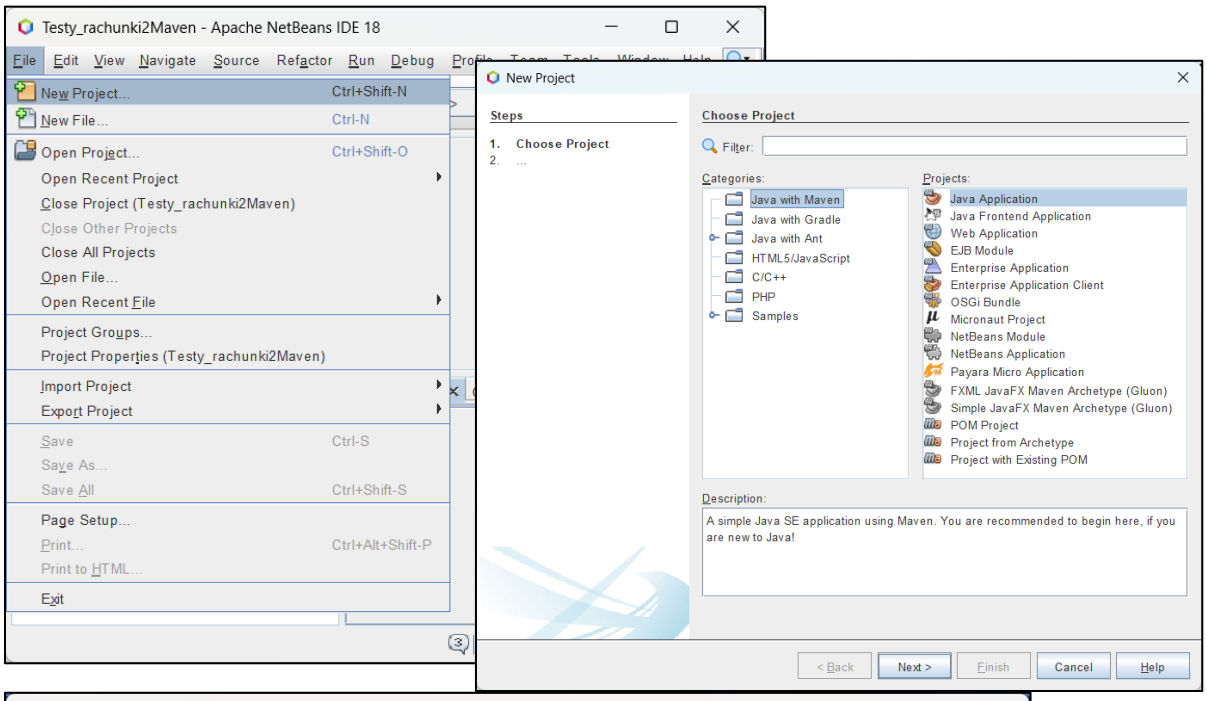
2. Testy jednostkowe z wykorzystaniem narzędzia *JUnit 5*

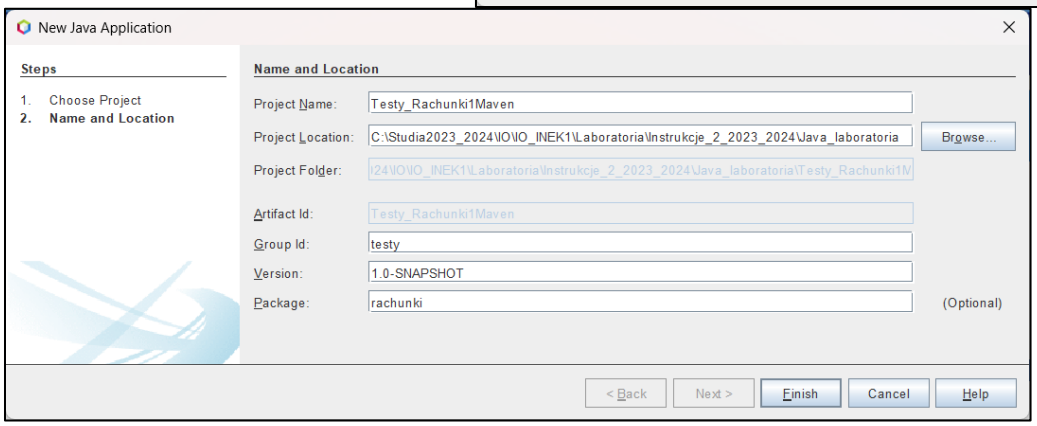
Wykonanie projektu typu ***Maven*** w środowisku ***ApacheNetbeans18*** do testowania z wykorzystaniem **JUnit5**. Pomocnicze informacje są podane w następujących materiałach:

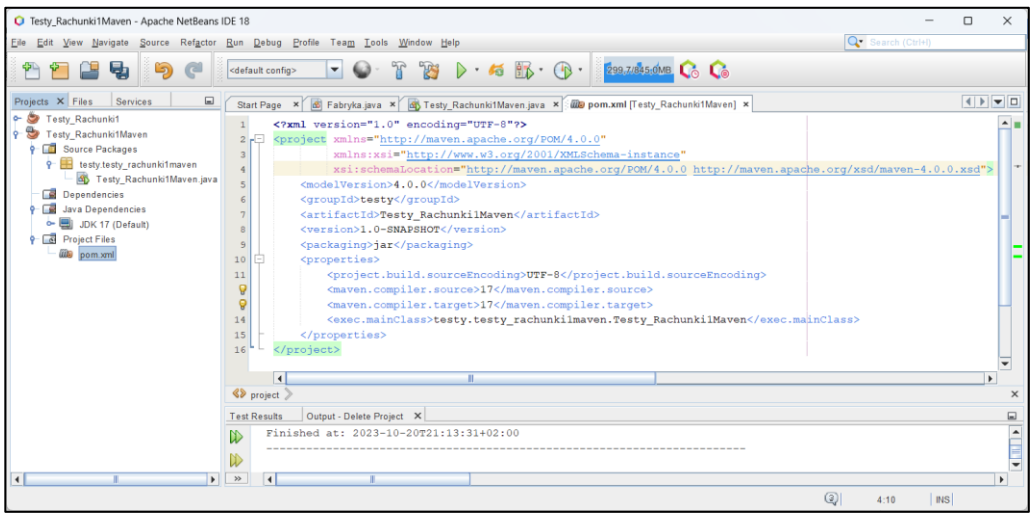
JUnit 5 User Guide

Maven – Welcome to Apache Maven

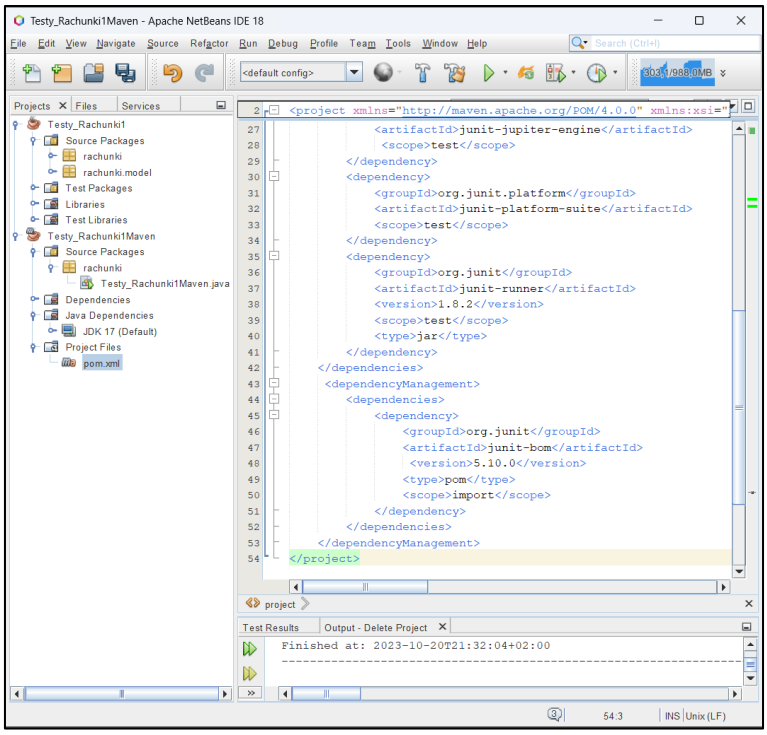
Po wykonaniu następujących czynności:



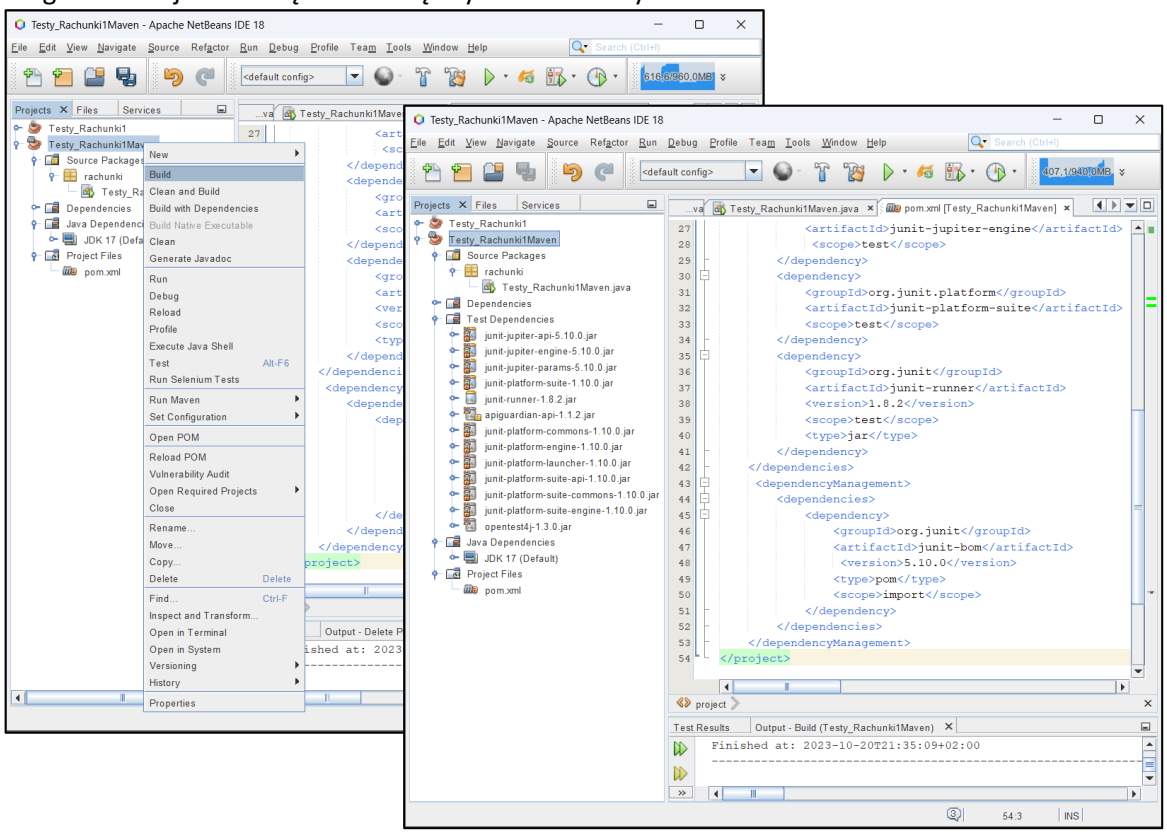




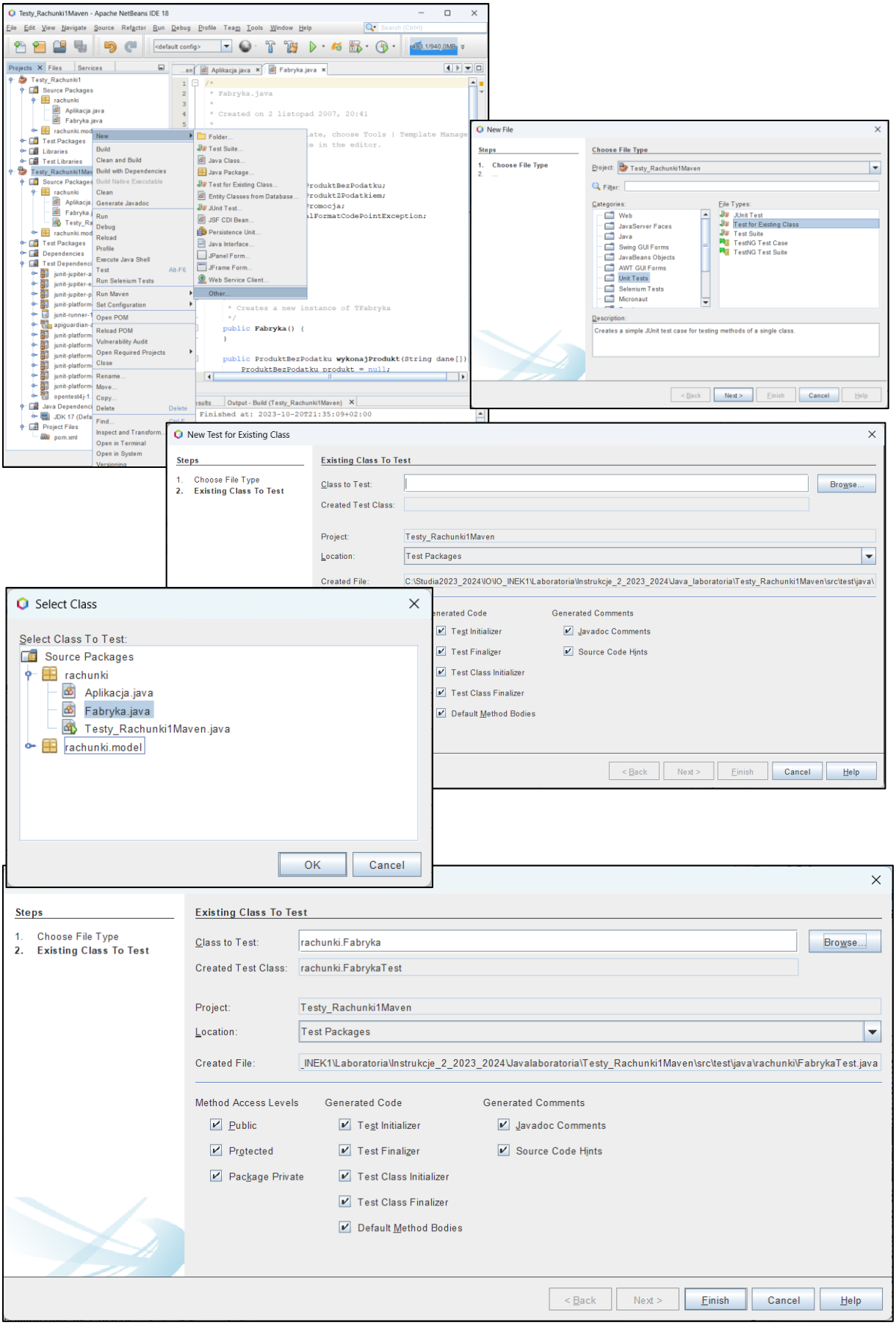
- Dodanie pakietów ***rachunki*** oraz **rachunki.model** klasami do testowania do folderu ***Source Packages***
- wypełnienie dodatkowymi adnotacjami pliku **pom**.xml, służącymi do przeprowadzenia zaplanowanych testów w ***JUnit 5.*** Docelową zawartość tego pliku podano na str.10.



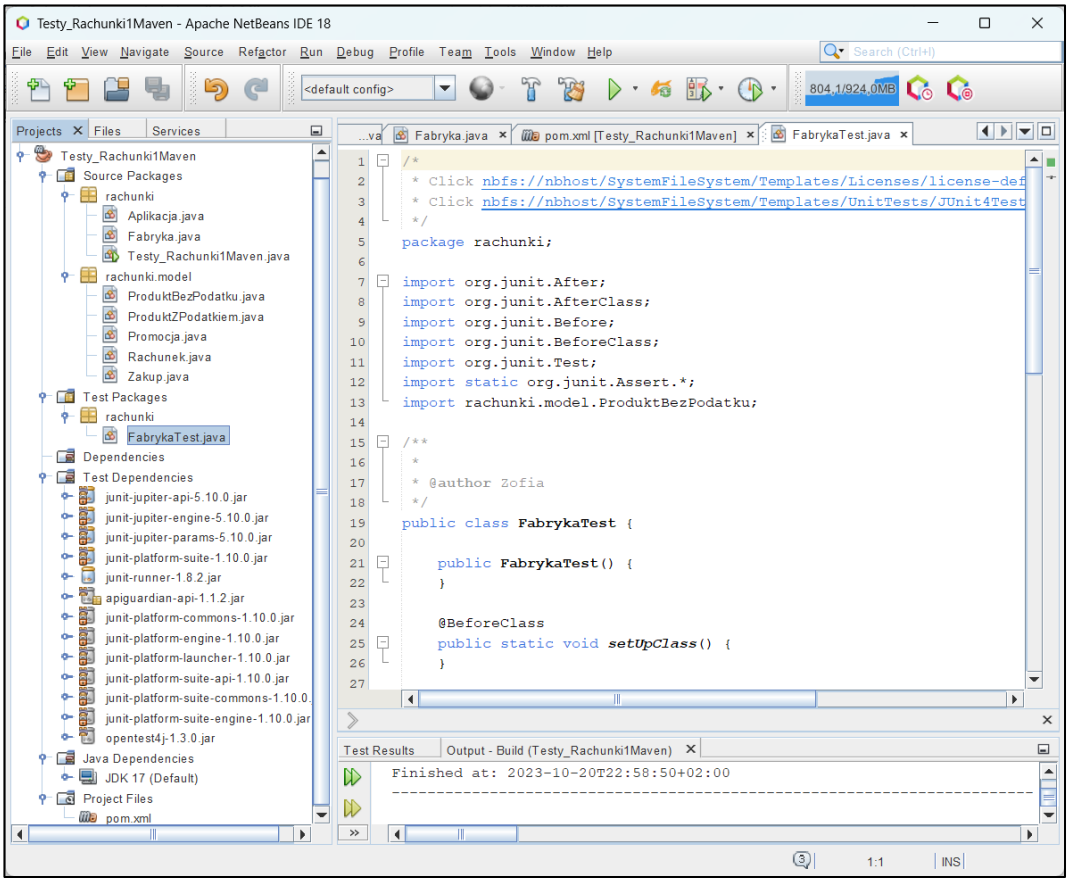
- wykonaniu kompilacji kodu projektu za pomocą ***Build*** zostanie wygnerowany folder ***Test Dependences*** zawierający odwołania do bibliotek projektu, zdefiniowanych w pliku **pom.xml.** Te biblioteki są umieszczone domyślnie w katalogu **C:\Users\User\.m2\repository\org\junit**, gdzie ***User*** jest nazwą nadawaną użytkownikowi systemu



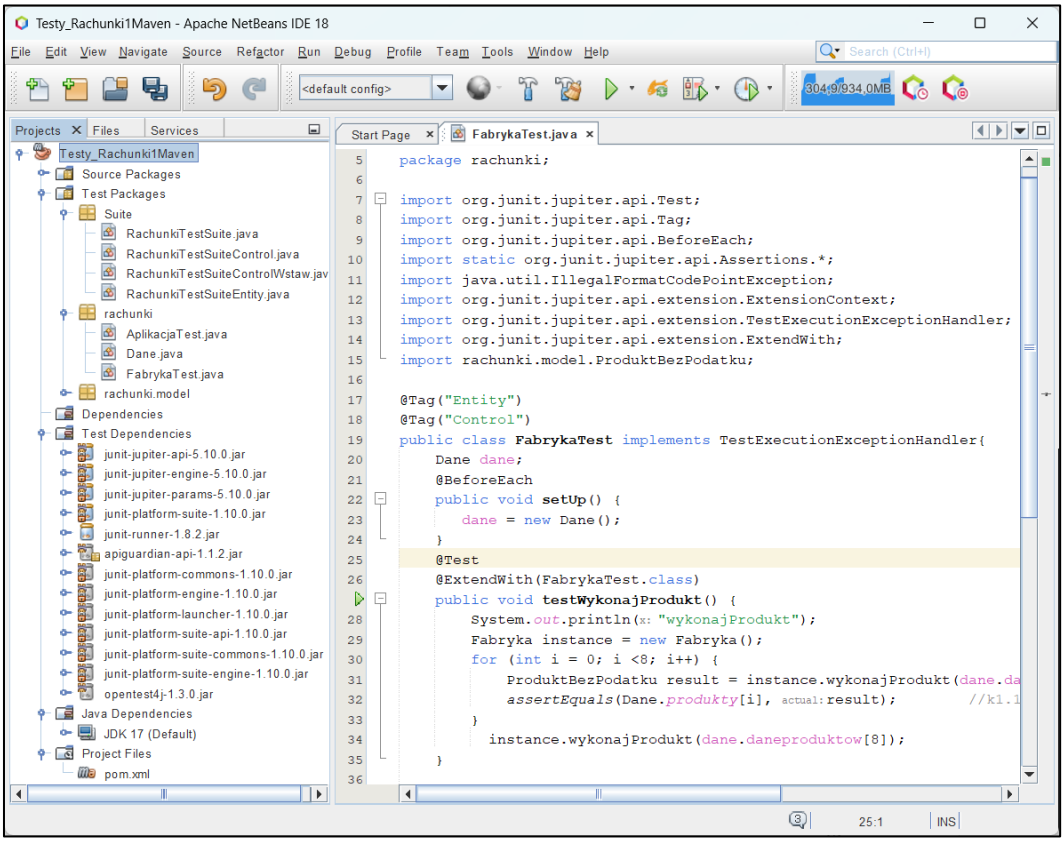
- dodanie automatycznie wygenerowanego testu dla wybranej klasy np. ***Fabryka***



Widok projektu po wygenerowaniu folderu typu ***Test Packages*** z plikiem zawierającym standardową postac testu wybranej klasy ***Fabryka***. Należy ten test zmodyfikować wg p.2.2.



Widok projektu po zdefiniowaniu testów jednostkowych opisanych w punktach 2.1 – 2.7.



Oto zawartość pliku **pom.xml** zawierająca między innymi definicję **standardowych** odwołań do biblioteki **JUnit5.** Dane projektu: nazwa projektu, nazwy pakietów itd. należy zaktualizować.

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
       http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>testy</groupId>
  <artifactId>Testy_Rachunki1Maven</artifactId>
  <version>1.0-SNAPSHOT</version>
  <packaging>jar</packaging>
  <properties>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
    <maven.compiler.source>17</maven.compiler.source>
    <maven.compiler.target>17</maven.compiler.target>
    <exec.mainClass>rachunki.Testy_Rachunki1Maven</exec.mainClass>
  </properties>
  <dependencies>
    <dependency>
      <groupId>org.junit.jupiter</groupId>
      <artifactId>junit-jupiter-api</artifactId>
       <scope>test</scope>
    </dependency>
    <dependency>
      <groupId>org.junit.jupiter</groupId>
      <artifactId>junit-jupiter-params</artifactId>
      <scope>test</scope>
    </dependency>
    <dependency>
      <groupId>org.junit.jupiter</groupId>
      <artifactId>junit-jupiter-engine</artifactId>
       <scope>test</scope>
    </dependency>
    <dependency>
      <groupId>org.junit.platform</groupId>
      <artifactId>junit-platform-suite</artifactId>
      <scope>test</scope>
    </dependency>
    <dependency>
      <groupId>org.junit</groupId>
      <artifactId>junit-runner</artifactId>
      <version>1.8.2</version>
      <scope>test</scope>
      <type>jar</type>
    </dependency>
   </dependencies>
  <dependencyManagement>
    <dependencies>
      <dependency>
        <groupId>org.junit</groupId>
        <artifactId>junit-bom</artifactId>
         <version>5.10.0</version>
        <type>pom</type>
        <scope>import</scope>
      </dependency>
    </dependencies>
  </dependencyManagement>
</project>
```

2.1. Definicja danych wzorcowych - należy wstawić pomocniczą klasę ***Dane*** z danymi wzorcowymi do testowania w metodach klas testujących zdefiniowanych w **Dodatku 1** w p 2.2-2.7

Opis danych wzorcowych do testowania tworzenia i zawartości obiektów typu *Zakup* i z rodziny *ProduktBezPodatku*

2.1.1. String **daneproduktow**[][] – dwuwymiarowa tablica zawierająca w pierwszych ośmiu wierszach dane do utworzenia ośmiu obiektów z rodziny ***ProduktBezPodatku*** (**Dodatek 1**, Instrukcja 7, klasy: ***Fabryka, ProduktBezPodatku, ProduktZPodatkiem, Promocja***). Wiersz dziewiąty zawiera dane niepoprawne, powodujące generowanie wyjątku ***IllegalFormatCodePointException*** przez metodę klasy ***Fabryka***.

2.1.2. **ProduktBezPodatku produkty**[] – jednowymiarowa tablica ośmiu obiektów wzorcowych z rodziny ***ProduktBezPodatku***, zdefiniowanych na podstawie danych z tabeli ***dane_produktow*** z p. 2.1.1

2.1.3. **float cenyprodukty**[] – jednowymiarowa tablica ośmiu wartości ceny jednostkowej każdego z produktów podanych w tabeli ***produkty*** z p.2.1.2, wynikającej z promocji i podatku oraz ceny netto produktu.

2.1.4. **Zakup zakupy**[] – tablica ośmiu obiektów typu ***Zakup***, zdefiniowanych z wykorzystaniem obiektów z rodziny ***ProduktBezPodatku***, zdefiniowanych w tablicy ***produkty*** z p. 2.1.2.

2.1.5. **float cenyzakupow**[] – jednowymiarowa tablica zawierająca osiem wartości kosztów zakupów ośmiu obiektów typu ***Zakup***, zdefiniowanych w tablicy ***zakupy*** z p.2.1.4.

2.1.6. **int podatkizakupow**[] - jednowymiarowa tablica zawierająca osiem wartości podatków ośmiu obiektów typu ***Zakup***, zdefiniowanych w tablicy ***zakupy*** z p.2.1.4.

Opis danych wzorcowych do testowania tworzenia i zawartości dwóch rachunków

2.1.7. **Rachunek rachunki**[] – jednowymiarowa tablica zawierająca dwa obiekty typu ***Rachunek*** z pustymi kolekcjami obiektów typu ***Zakup***

2.1.8. String **daneproduktowrachunki**[][][] –tablica zawierająca dane wejściowe produktów (dane jako elementy tablicy p.2.1.1) należących do zakupów dwóch rachunków, gdy każdy z nich zawiera po pięć zakupów.

2.1.9. **Zakup zakupyrachunki[]**[] –dwuwymiarowa tablica obiektów typu ***Zakup.*** W testach stanowi ona zbiory wzorcowych obiektów typu ***Zakup,*** należących do dwóch obiektów typu ***Rachunek.*** Obiekty typu ***Zakup*** zawierają obiekty z rodziny ***ProduktBezPodatku,*** zdefiniowane w tablicy z p. 2.1.2.

2.1.10. **int ileproduktowrachunki**[][] – dwuwymiarowa tablica zawierająca w każdym z dwóch wierszy pięć danych o liczbie produktów w każdym z pięciu zakupów, gdy każdy z dwóch wierszy reprezentuje dane jednego z dwóch rachunków.

2.1.11. **int kategorie**[] – jednowymiarowa tablica zawierająca wartości różnych kategorii wyznaczania ceny rachunku, gdzie wartość -1 oznacza wyznaczenie ceny zakupu produktów bez podatku, wartości: 3, 7, 14, 22 oznaczają kategorie cen rachunku wynikające z wysokości podatku produktów oraz -2 oznacza, że należy podać całkowity koszt rachunku – uwzględniając produkty bez podatku oraz wszystkie z podatkami.

2.1.12. **float kategoriewartoscirachunki**[][] – dwuwymiarowa tablica wartości sześciu wartości dwóch rachunków, wynikających z kategorii cen podanych w tabeli z p.2.1.11.

```
package rachunki;
import rachunki.model.ProduktBezPodatku;
import rachunki.model.ProduktZPodatkiem;
import rachunki.model.Promocja;
import rachunki.model.Rachunek;
import rachunki.model.Zakup;

public class Dane {
    //dane wzorcowe do testowania obiektów typu Zakup i z rodziny ProduktBezPodatku
public String daneproduktow[][] = new String[][]{
    {"0", "1", "1", "0", "0"}, {"0", "2", "2", "0", "0"}, {"2", "3", "3", "14", "0"}, {"2", "4", "4", "22", "0"},
    {"1", "5", "1", "30", "0"}, {"1", "6", "2", "50", "0"}, {"3", "7", " 5.47", "3", "30"}, {"3", "8", " 12.4", "7", "50"},
    {"4", "1", "1", "0", "0"} }; // 9-y el. zawiera dane do testowania generowania wyjątku przez klasę Fabryka

public static ProduktBezPodatku produkty[] = {new ProduktBezPodatku("1", 1),
  new ProduktBezPodatku("2", 2), new ProduktZPodatkiem("3", 3, 14), new ProduktZPodatkiem("4", 4, 22),
  new ProduktBezPodatku("5", 1, new Promocja(30)), new ProduktBezPodatku("6", 2, new Promocja(50)),
  new ProduktZPodatkiem("7", 5.47F, 3, new Promocja(30)), new ProduktZPodatkiem("8", 12.4F, 7,
                                                                    new Promocja(50)) };
  public float cenyprodukty[] = { 1F, 2F, 3.42F, 4.88F, 0.7F, 0.9F, 3.9930997F, 6.4479995F};
```

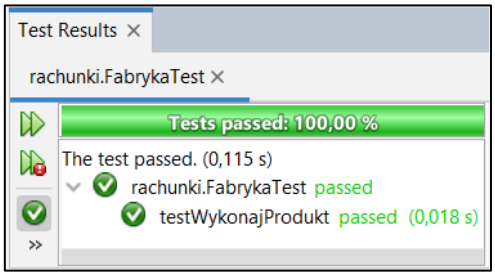
```
  public Zakup zakupy[] = {
        new Zakup(1, produkty[0]), new Zakup(4, produkty[1]),
        new Zakup(1, produkty[2]), new Zakup(1, produkty[3]),
        new Zakup(1, produkty[4]), new Zakup(1, produkty[5]),
        new Zakup(3, produkty[6]), new Zakup(1, produkty[7])
  };

   public float cenyzakupow[] = {1F, 8F, 3.42F, 4.88F, 0.7F, 0.9F, 11.9793F, 6.4479995F };
    public int podatkizakupow[] = {-1, -1, 14, 22, -1, -1, 3, 7  };

//dane zdefiniowane powyżej są zastosowane w definicji danych dwóch rachunków
 public Rachunek rachunki[] = { new Rachunek(1), new Rachunek(2)      };

  public String daneproduktowrachunki[][][] = new String[][][] {
   { daneproduktow[0], daneproduktow[1], daneproduktow[2], //dane rachunku 1
     daneproduktow[3], daneproduktow[4]
   },
   {  daneproduktow[5], daneproduktow[6], daneproduktow[7], //dane rachunku 2
      daneproduktow[1], daneproduktow[3]
    }
};

public Zakup zakupyrachunki[][] = {
   {   new Zakup(2, produkty[0]), new Zakup(2, produkty[1]), //obiekty typu Zakup rachunku 1
       new Zakup(1, produkty[2]), new Zakup(4, produkty[3]),
       new Zakup(1, produkty[4])
   },
   {   new Zakup(2, produkty[5]), new Zakup(3, produkty[6]), //obiekty typu Zakup rachunku 2
       new Zakup(2, produkty[7]),
       new Zakup(4, produkty[1]), new Zakup(1, produkty[3])
  }
};

 public int ileproduktowrachunki[][] = {
        {1, 2, 1, 4, 1},  //początkowa ilość produktów w kolejnych pięciu zakupach rachunku 1
        {1, 3, 2, 4, 1}  //początkowa ilość produktów w kolejnych pięciu zakupach rachunku 2
   };
 public  int kategorie[] = { -1, 3, 7, 14, 22, -2 };                    //kategorie wartości rachunków

  public float  kategoriewartoscirachunki[][] = {
        { 6.7F,  0F, 0F, 3.42F, 19.52F, 29.640001F},                 //wartości rachunku 1 wg kategorii
        { 9.8F, 11.9793F, 12.895999F, 0.0F, 4.88F, 39.5553F} //wartości rachunku 2 wg kategorii
    };
}
```

2.2. Test jednostkowy klasy ***Fabryka*** **(wynik działania: p.2.7.1, 2.7.3, 2.7.4)** – przykłady prostego testu (k1.2) porównującego osiem wyników działania metody ***wykonajProdukt()*** tworzącej cztery różne typy obiektów z rodziny ***ProduktBezPodatku*** z wzorcowymi wynikami z tabeli ***produkty*** z klasy ***Dane*** za pomocą metody ***assertEquals*** klasy ***Assertions (***pakiet źródłowy ***junit.jupiter.api)*** oraz reakcję na niepoprawną wartość pierwszego elementu tablicy reprezentującej dane wejściowe testowanej metody ***wykonajProdukt*** (k1.2). Obsługę testowania generowanego wyjątku wykonano za pomocą adnotacji ***@ExtendWith(FabrykaTest.class)***, która określa, że w klasie ***FabrykaTest*** zaimplementowano metodę ***handleTestExecutionException*** interfejsu ***TestExecutionExceptionHandler***, która wykrywa wyjątek typu ***IllegalFormatCodePointException*** generowany przez klasę ***Fabryka*** dla niepoprawnych danych.

Zastosowanie adnotacji
@Test
@BeforeEach
@Tag
@ExtendWith

Zastosowanie metod
static public void assertEquals(Object expected, Object actual) //k1.1
public void handleTestExecutionException(ExtensionContext context, Throwable throwable)//k1.2

```
package rachunki;
import org.junit.jupiter.api.Test;
import org.junit.jupiter.api.Tag;
import org.junit.jupiter.api.BeforeEach;
import static org.junit.jupiter.api.Assertions.*;
import java.util.IllegalFormatCodePointException;
import org.junit.jupiter.api.extension.ExtensionContext;
import org.junit.jupiter.api.extension.TestExecutionExceptionHandler;
import org.junit.jupiter.api.extension.ExtendWith;
import rachunki.model.ProduktBezPodatku;

@Tag("Entity")          //określenie kategorii testu, zastosowanie -  p.2.7.1, 2.7.3
@Tag("Control")         //określenie kategorii testu, zastosowanie -  p.2.7.1, 2.7.3
public class FabrykaTest  implements TestExecutionExceptionHandler {//k1.2 implementacja podanego
//interfejsu umożliwia obsługę wyjątku typu IllegalFormatCodePointException generowanego przez metodę
//wykonajProdukt przez klasę Fabryka
Dane dane;

@BeforeEach
   public void setUp(){
      dane= new Dane();  }

@Test
@ExtendWith(FabrykaTest.class)
   public void testWykonajProdukt() {
      System.out.println("wykonajProdukt");
      Fabryka instance = new Fabryka();
      for (int i = 0; i < 8; i++) {
           ProduktBezPodatku result = instance.wykonajProdukt(dane.daneproduktow[i]);
           assertEquals(dane.produkty[i], result);  }          //k1.1 – test poprawności tworzonych produktów
      instance.wykonajProdukt(dane.daneproduktow[8]);  //k1.2 - wywołanie obsługi wyjątku
    }
//k1.2 – definicja zachowania metody testowej testWykonajProdukt podczas testowania generowania
  @Override  //wyjątku IllegalFormatCodePointException przez metodę wykonajProdukt
  public void handleTestExecutionException(ExtensionContext context, Throwable throwable)
         throws Throwable {
        if (throwable instanceof IllegalFormatCodePointException) {  }
        else throw throwable;
   }
}
```

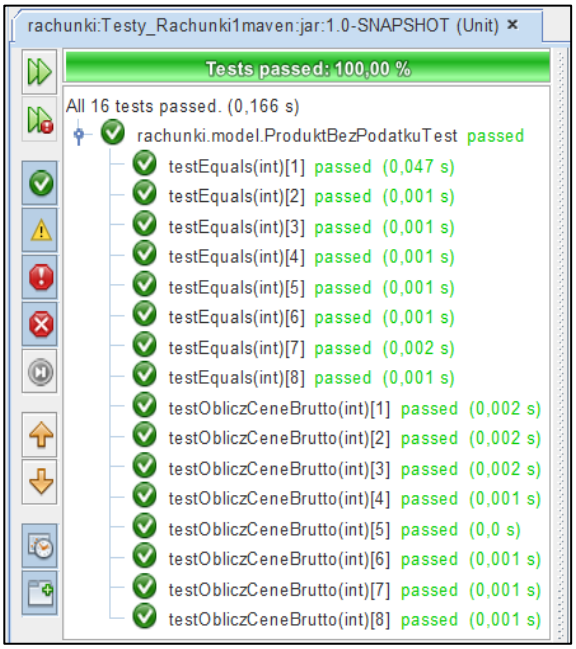


2.3. Testy jednostkowe klas *ProduktBezPodatku* i *ProduktZPodatkiem* (wynik działania: p.2.7.2, 2.7.4) – zastosowanie adnotacji **@ParameterizedTest, @MethodSource** i metody powiązanej **intProvider** dla parametru ***numer1*** powoduje wywołanie metody testowej ***testEquals*** osiem razy, podstawiając w kolejnej iteracji wartość kolejnego elementu typu **int** z ośmio-elementowego strumienia danych zwracanego przez metodę ***intProvider*** do parametru ***numer1***. Taki sam mechanizm wykorzystany jest w uruchomieniu osiem razy metody testowej ***testObliczCeneBrutto(),*** która sprawdza wyniki zwracane przez metodę ***obliczCeneBrutto*** *(***k2***)* dla ośmiu obiektów z rodziny ***ProduktBezPodatku***, porównując je z wynikami wzorcowymi. Metoda testowa ***testEquals()*** umożliwia weryfikację działania metody ***equals*** na każdej parze obiektów z tabeli *produkty (***k1.1** i **k1.2***)*.

Zastosowanie adnotacji
@ParameterizedTest
@MethodSource
@Tag
@BeforeAll

Zastosowanie metod
static public void assertTrue(boolean condition), //k1.1
static public void assertFalse(boolean condition), //k1.2
static public void assertEquals(float expected, float actual, float delta)//k2

```
package rachunki.model;

import java.util.stream.Stream;
import static org.junit.jupiter.api.Assertions.*;
import org.junit.jupiter.api.Tag;
import org.junit.jupiter.api.BeforeAll;
import org.junit.jupiter.params.ParameterizedTest;
import org.junit.jupiter.params.provider.MethodSource;
import rachunki.Dane;


@Tag("Entity") //określenie kategorii testu, p.2.7.2
public class ProduktBezPodatkuTest {
   static Dane dane;

 @BeforeAll
 public static void setUpClass() {
       dane = new Dane();    }

static Stream<Integer> intProvider() {
      return Stream.of(0, 1, 2, 3, 4, 5, 6, 7); }

  @ParameterizedTest()
  @MethodSource("intProvider")
  public void testEquals(int numer1) {
      System.out.println("equals");
       for(int j=numer1;j<8;j++)
        if(numer1==j)
          assertTrue(dane.produkty[numer1].equals(Dane.produkty[j])); //k1.1 –test porównania
        else                                                          // równych produktów
          assertFalse(dane.produkty[numer1].equals(Dane.produkty[j])); //k1.2–test porównania
          }                                                           //różnych produktów

  @ParameterizedTest()
  @MethodSource("intProvider")
   public void testObliczCeneBrutto(int numer1) {
      System.out.println("obliczCeneBrutto");
      float result1 = dane.produkty[numer1].obliczCeneBrutto();
      float result2 = dane.cenyprodukty[numer1];
      assertEquals(result1, result2, 0F); //k2 – test wyznaczania poprawnej wartości cen brutto produktów
   }
}
```

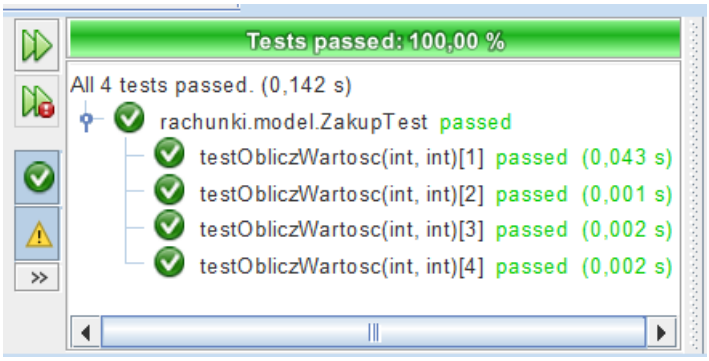


2.4. Testy jednostkowe klasy *Zakup* (wynik działania: p.2.7.2, 2.7.4) – zastosowanie adnotacji **@ParameterizedTest oraz @CsvSource** dla atrybutów ***numer1*** i ***numer2***, które powodują wywołanie jednej metody testowej cztery razy, podstawiając w kolejnej iteracji wartość elementów z kolejnego z czterech 2-elementowych wierszy tablicy ***@CsvSource*** do parametrów: ***numer1*** *i* ***numer***. W rezultacie w metodzie testowej ***testObliczWartosc(int numer1, int numer2)*** sprawdza się wyniki zwracane przez metodę ***obliczWartosc*** dla ośmiu obiektów z rodziny ***Zakup***, porównując je z wynikami wzorcowymi.

Zastosowanie adnotacji
@ParameterizedTest
@MethodSource
@Tag
@BeforeAll

Zastosowanie metod
static public void assertEquals(float expected, float actual, float delta), //k1.1, k1.2

```
package rachunki.model;

import static org.junit.jupiter.api.Assertions.*;
import org.junit.jupiter.api.Tag;
import org.junit.jupiter.api.BeforeAll;
import org.junit.jupiter.params.ParameterizedTest;
import org.junit.jupiter.params.provider.CsvSource;
import rachunki.Dane;

@Tag("Entity")    //określenie kategorii testu, zastosowanie -  p.2.7.2
public class ZakupTest {

 static Dane dane;

 @BeforeAll
 public static void setUpClass() {
     dane = new Dane();
  }

@ParameterizedTest
@CsvSource({"0,1","2,3","4,5","6,7"})
 public void testObliczWartosc(int numer1, int numer2) {
     System.out.println("obliczWartosc");
     assertEquals(dane.cenyzakupow[numer1],  //k1.1 test wyznaczania wartości zakupów: 1-y zbiór danych
                  dane.zakupy[numer1].obliczWartosc(dane.podatkizakupow[numer1]), 0.0F);
     assertEquals(dane.cenyzakupow[numer2],
                  dane.zakupy[numer2].obliczWartosc(dane.podatkizakupow[numer2]), 0.0F);//k1.2
                                        //test wyznaczania wartości zakupów: 2-i zbiór danych
  }
}
```



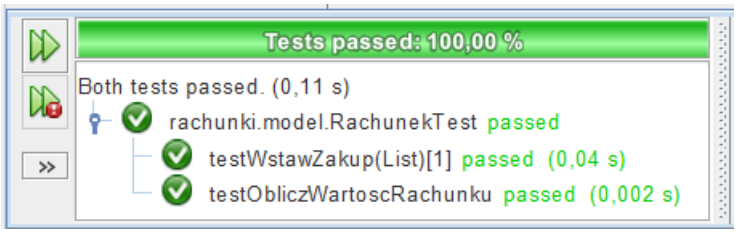
2.5. Testy jednostkowe klasy *Rachunek* (wynik działania: p.2.7.2, 2.7.4) - zastosowanie adnotacji **@ParameterizedTest i @MethodSource** dla parametru ***List<List<Zakup>>zakupy* zwróconego jako wynik wywołanej metody *provideArguments*** oraz adnotacji **@TestMethodOrder** i **@Order** powoduje wywołanie najpierw metody testowej ***testWstawZakup***, a następnie metody ***testObliczWartoscRachunku***, która działa na danych wstawionych i przetestowanych w pierwszej metodzie testowej. Lista dwuelementowa elementów typu lista, każda zawierająca pięć danych typu ***Zakup*** jako dane obiektu typu ***Rachunek*** – czyli główna lista zawiera dane dwóch obiektów typu ***Rachunek***. Za pomocą adnotacji **@BeforeAll** wykonano w metodzie ***SetUp*** jednorazowo (przed wykonaniem wszystkich dwóch testów) tablicę obiektów typu ***Rachunek***, które w metodzie testowej ***testWstawZakup*** są zapełnione obiektami z listy ***lista*** pobranych z dwuelementowej listy ***zakupy***. Dzięki narzuceniu kolejności wykonania metod testowych za pomocą adnotacjiom ***@Order*** i ***@TestMethodOrder(OrderAnnotation.class)*** ustalono kolejność wykonania metod testowych: ***testWstawZakup***, ***testObliczWartoscRachunku***. Kolejna metoda testowa korzysta z danych utworzonych w poprzedniej metodzie testowej, czyli metoda ***testObliczWartoscRachunku()*** operuje na rachunkach wypełnionych obiektami typu ***Zakup*** w metodzie ***testWstawZakup*.**

Zastosowanie adnotacji
@Tag
@Test
@BeforeAll
@ParameterizedTest
@TestMethodOrder(OrderAnnotation.class)

Zastosowanie metod
static public void assertSame(Object expected, Object actual) //k1.1
static public void assertEquals(long expected, long actual) //k1.2, k1.3
static public void assertEquals(float expected, float actual, float delta) //k2

```
package rachunki.model;
import java.util.Arrays;
import java.util.List;
import java.util.stream.Stream;
import org.junit.jupiter.api.Tag;
import org.junit.jupiter.api.Test;
import org.junit.jupiter.api.BeforeAll;
import org.junit.jupiter.api.MethodOrderer.OrderAnnotation;
import org.junit.jupiter.api.TestMethodOrder;
import org.junit.jupiter.api.Order;
import org.junit.jupiter.params.ParameterizedTest;
import org.junit.jupiter.params.provider.Arguments;
import org.junit.jupiter.params.provider.MethodSource;
import static org.junit.jupiter.params.provider.Arguments.arguments;
import static org.junit.jupiter.api.Assertions.*;
import rachunki.Dane;
@Tag("Entity")                    //określenie kategorii testu, zastosowanie - p.2.7.2
@TestMethodOrder(OrderAnnotation.class)
public class RachunekTest {
    static Dane dane;
   static Rachunek instances[];
   static int numer_rachunku;
 @BeforeAll
  public static void SetUp() {
       instances=new Rachunek[2];
       instances[0] = new Rachunek(1);
       instances[1] = new Rachunek(2);
       dane = new Dane();
       numer_rachunku= 0;   }
```

```
@ParameterizedTest()
@MethodSource("provideArguments")
@Order(1)
public void testWstawZakup(List<List<Zakup>>zakupy) {
   System.out.println("wstawZakup");
   for(List<Zakup>lista:zakupy){
      for(Zakup zakup: lista){
         instances[numer_rachunku].wstawZakup(zakup);
        int ktory=instances[numer_rachunku].getZakupy().size() - 1;
        Zakup zakup1 = instances[numer_rachunku].getZakupy().get(ktory);
        assertSame(zakup1, zakup); //k1.1 – test sprawdzenia równości referencyjnej danych
      }
      int rozmiar1 = instances[numer_rachunku].getZakupy().size();
      int ile = instances[numer_rachunku].getZakupy().get(0).getIlosc();
      instances[numer_rachunku].wstawZakup(lista.get(0));
      assertEquals(rozmiar1, instances[numer_rachunku].getZakupy().size());
                        //k1.2- test spójności danych podczas  dodawania podobnych zakupów
      assertEquals(instances[numer_rachunku].getZakupy().get(0).getIlosc(), ile * 2);
                        //k1.3 test algorytmu dodawania podobnych zakupów
     numer_rachunku++;
   }
 }
@Test
@Order(2)
public void testObliczWartoscRachunku() {
   System.out.println("obliczWartoscRachunku");
    for (int i = 0; i < 2; i++)
      for (int j = 0; j < 5; j++)
        assertEquals(dane.kategoriewartoscirachunki[i][j],
                instances[i].obliczWartoscRachunku(dane.kategorie[j]), 0F);
   }                          //k2 – test obliczania wartości rachunku w różnych kategoriach

static public Stream<Arguments> provideArguments() {
    return Stream.of(arguments(Arrays.asList(
                Arrays.asList
               (new Zakup(1, Dane.produkty[0]),
                new Zakup(2, Dane.produkty[1]),
                new Zakup(1, Dane.produkty[2]),
                new Zakup(4, Dane.produkty[3]),
                new Zakup(1, Dane.produkty[4])),
                Arrays.asList(
                new Zakup(1, Dane.produkty[5]),
                new Zakup(3, Dane.produkty[6]),
                new Zakup(2, Dane.produkty[7]),
                new Zakup(4, Dane.produkty[1]),
                new Zakup(1, Dane.produkty[3])))
          ));
  }
}
```



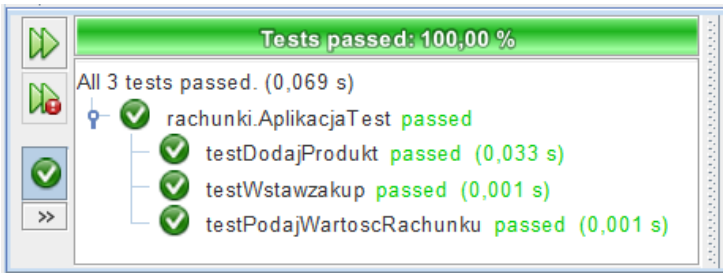
2.6. Testy jednostkowe klasy *Aplikacja* (wynik działania: p.2.7.1, 2.7.4) opierają się na wywołaniu trzech metod testowych, działających w ustalonej kolejności dzięki zastosowaniu adnotacjom ***@TestMethodOrder(OrderAnnotation.class)*** **i** ***@Oder***: ***testDodajProdukt, testWstawZakup, testPodajWartoscRachunku***. Przed wywołaniem metod testowych wywołana jest metoda ***SetUp*** dzięki zastosowaniu adnotacji **@BeforeAll** – metoda ta tworzy obiekty typu ***Aplikacja*** i ***Dane***. Metody działające w podanym porządku umożliwiają po dodaniu produktów w metodzie ***testDodajProdukt*** wykonać testy: dodawania zakupów w metodzie ***testWstawZakup***, obliczania wartości rachunków w różnych kategoriach w metodzie ***testPodajWartoscRachunku***. Metody testowe ***testDodajProdukt*** oraz ***testWstawZakup*** testują również przypadek podania niepoprawnej wartości w danych wejściowych, które powodują generowanie wyjątku przez metodę ***wykonajProdukt*** klasy ***Fabryka*** – wyjątek ten jest obsługiwany w podanych metodach testowych za pomocą adnotacji ***@ExtendWith(AplikacjaTest.class)***, która określa, że klasa testująca ***AplikacjaTest*** implementuje metodę ***handleTestExecutionException*** interfejsu ***TestExecutionExceptionHandler*** obsługującą podany wyjątek.

2.7. Za pomocą adnotacji **@Tag** dokonano różnych klasyfikacji wszystkich metod testowych i wybranej metody ***testPodajWartoscRachunku***.

Zastosowanie adnotacji
@Test
@BeforeAll
@TestMethodOrder(OrderAnnotation.class)
@Order
@Tag

Zastosowanie metod
static public void assertEquals(Object expected, Object actual)// k1.1, k2.1
static public void assertEquals(long expected, long actual) //k1.2, k2.2, k2.3
static public void assertEquals(float expected, float actual, float delta) //k3
public void handleTestExecutionException(ExtensionContext context, Throwable throwable)//k4

```
package rachunki;
import java.util.Arrays;
import java.util.IllegalFormatCodePointException;
import org.junit.jupiter.api.BeforeAll;
import org.junit.jupiter.api.Test;
import static org.junit.jupiter.api.Assertions.*;
import org.junit.jupiter.api.MethodOrderer.OrderAnnotation;
import org.junit.jupiter.api.Order;
import org.junit.jupiter.api.Tag;
import org.junit.jupiter.api.TestMethodOrder;
import org.junit.jupiter.api.extension.ExtendWith;
import org.junit.jupiter.api.extension.ExtensionContext;
import org.junit.jupiter.api.extension.TestExecutionExceptionHandler;
import rachunki.model.ProduktBezPodatku;
import rachunki.model.Zakup;
@TestMethodOrder(OrderAnnotation.class)
@Tag("Entity")
@Tag("Control")
public class AplikacjaTest implements TestExecutionExceptionHandler{
   static Dane dane;
   static Aplikacja instance;
//k4 – implementacja metody do obsługi wyjątku w testowych metodach testDodajProdukt
@Override                                                    //i testWstawZakup
 public void handleTestExecutionException(ExtensionContext context, Throwable throwable)
        throws Throwable {
    if (throwable instanceof IllegalFormatCodePointException) {  }
       else throw throwable; }
```

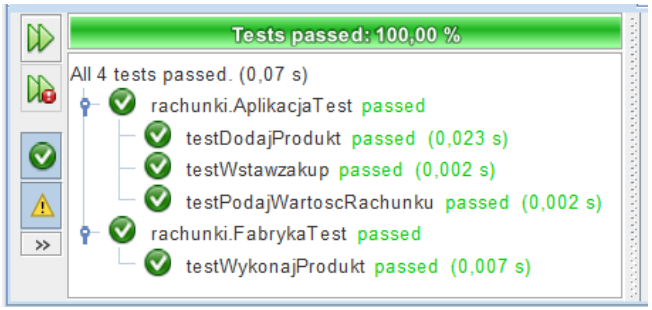



```
@BeforeAll
  static public void SetUp() {
      instance = new Aplikacja();
      dane = new Dane();
   }
@Test
@Order(1)
@ExtendWith(AplikacjaTest.class)
public void testDodajProdukt() {
  System.out.println("dodajProdukt");
  int indeksyproduktow[] = {0, 1, 2, 3, 4, 5, 6, 7, 7, 7};
   for (int i = 0; i < 2; i++)
    for (int j = 0; j < 5; j++) {
      instance.dodajProdukt(dane.daneproduktowrachunki[i][j]);
      int ile1 = instance.getProdukty().size();
      instance.dodajProdukt(dane.daneproduktowrachunki[i][j]);//powtórzenia wartości elementów
                                                              // dla i=1 oraz j=3, j=4
      int ile2 = instance.getProdukty().size();
      ProduktBezPodatku result = instance.getProdukty().get(ile2 - 1);
      assertEquals(dane.produkty[indeksyproduktow[i * 5 + j]], result); //k1.1 test dodawania produktów
      assertEquals(ile1, ile2);    }              //k1.2 – test spójności danych podczas dodawanie produktów
   instance.dodajProdukt(dane.daneproduktow[8]); //k4 - obsługa wyjątku w testowanej metodzie
}

@Test
@Order(2)
@ExtendWith(AplikacjaTest.class)
 public void testWstawZakup() {
      System.out.println("wstawZakup");
      instance.setRachunki(Arrays.asList(dane.rachunki));
      for (int i = 0; i < 2; i++) {
         for (int j = 0; j < 5; j++) {
            instance.wstawZakup(i + 1, dane.ileproduktowrachunki[i][j],  dane.daneproduktowrachunki[i][j]);
            Zakup zakup1 = instance.getRachunki().get(i).getZakupy().get(j);
            assertEquals(zakup1, dane.zakupyrachunki[i][j]);     //k2.1 test dodawania zakupów
         }
         int rozmiar = instance.getRachunki().get(i).getZakupy().size();
         instance.wstawZakup(i + 1, dane.ileproduktowrachunki[i][0],  dane.daneproduktowrachunki[i][0]);
         assertEquals(instance.getRachunki().get(i).getZakupy().size(), rozmiar); //k2.2 test spójności danych
                                                                                  //podczas dodawania zakupów
         assertEquals(instance.getRachunki().get(i).getZakupy().get(0).getIlosc(),
                      dane.zakupyrachunki[i][0].getIlosc());   //k2.3 test algorytmu dodawania
      }                                                        // podobnych zakupów
     instance.wstawZakup(1, 1, dane.daneproduktow[8]); //k4 obsługa wyjątku w testowanej metodzie
   }

@Test
@Tag("Koszt")//określenie kategorii testu – przykład zastosowania w p.2.7.4
@Order(3)
public void testPodajWartoscRachunku() {
       System.out.println("podajWartoscRachunku");
       for (int i = 0; i < 2; i++)
          for (int j = 0; j < 6; j++)
             assertEquals(dane.kategoriewartoscirachunki [i][j],              //k3 – test obliczania wartości
               instance.podajWartoscRachunku(i + 1, dane.kategorie[j]), 0F); //rachunku w różnych kategoriach
   }
}
```

2.8. Tworzenie zestawów testów

2.8.1. Wyniki testów wykonanych przez klasy należące również do kategorii @Tag("Control"): *FabrykaTest, AplikacjaTest*

```
package Suite;

import org.junit.platform.suite.api.SelectPackages;
import org.junit.platform.suite.api.Suite;
import org.junit.platform.suite.api.IncludeTags;

@Suite
@SelectPackages("rachunki")
@IncludeTags("Control")
public class RachunkiTestSuiteControl { }
```

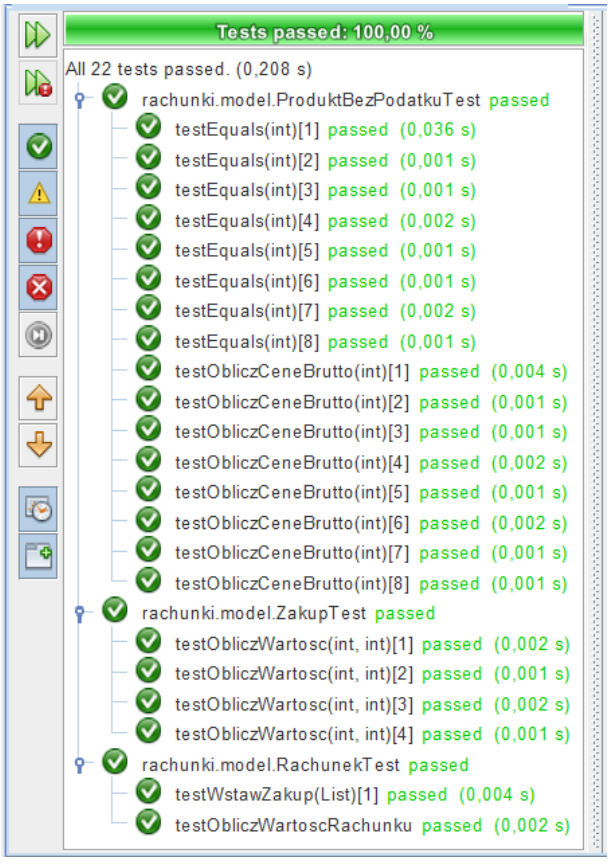


dodajProdukt
wstawZakup
podajWartoscRachunku
wykonajProdukt

Wynik testu: AplikacjaTest, FabrykaTest

2.8.2. Wyniki testów wykonanych przez klasy należące tylko do kategorii @Tag("Entity"): *ProduktBezPodatkuTest, ZakupTest, RachunekTest*

```
package Suite;

import org.junit.platform.suite.api.IncludeTags;
import org.junit.platform.suite.api.ExcludeTags;
import org.junit.platform.suite.api.SelectPackages;
import org.junit.platform.suite.api.Suite;

@Suite
@SelectPackages("rachunki")
@IncludeTags("Entity")
@ExcludeTags("Control")
public class RachunkiTestSuiteEntity
{ }
```



equals
equals
equals
equals
equals
equals
equals
equals

obliczCeneBrutto
obliczCeneBrutto
obliczCeneBrutto
obliczCeneBrutto
obliczCeneBrutto
obliczCeneBrutto
obliczCeneBrutto
obliczCeneBrutto

Wyniki testów *ProduktBezPodatkuTest*

obliczWartosc
obliczWartosc
obliczWartosc
obliczWartosc

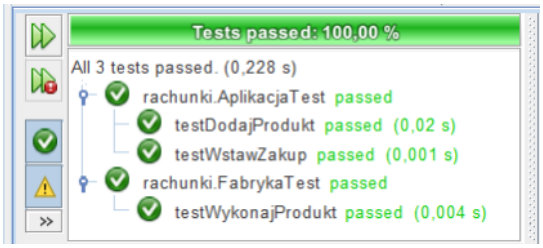
Wyniki testów *ZakupTest*

wstawZakup
obliczWartoscRachunku

Wyniki testów RachunekTest

2.8.3. Wyniki testów wykonanych przez klasy należące do kategorii @Tag("Control") z wyłączeniem metody *testPodajWartoscRachunku* klasy *AplikacjaTest* zaliczonej do kategorii @Tag("Koszt"): *FabrykaTest, AplikacjaTest*

```
package Suite;
import org.junit.platform.suite.api.IncludeTags;
import org.junit.platform.suite.api.ExcludeTags;
import org.junit.platform.suite.api.SelectClasses;
import org.junit.platform.suite.api.Suite;

@Suite
@SelectClasses({RachunkiTestSuiteControl.class})
@IncludeTags("Control")
@ExcludeTags("Koszt")
public class RachunkiTestSuiteControlWstaw {}
```

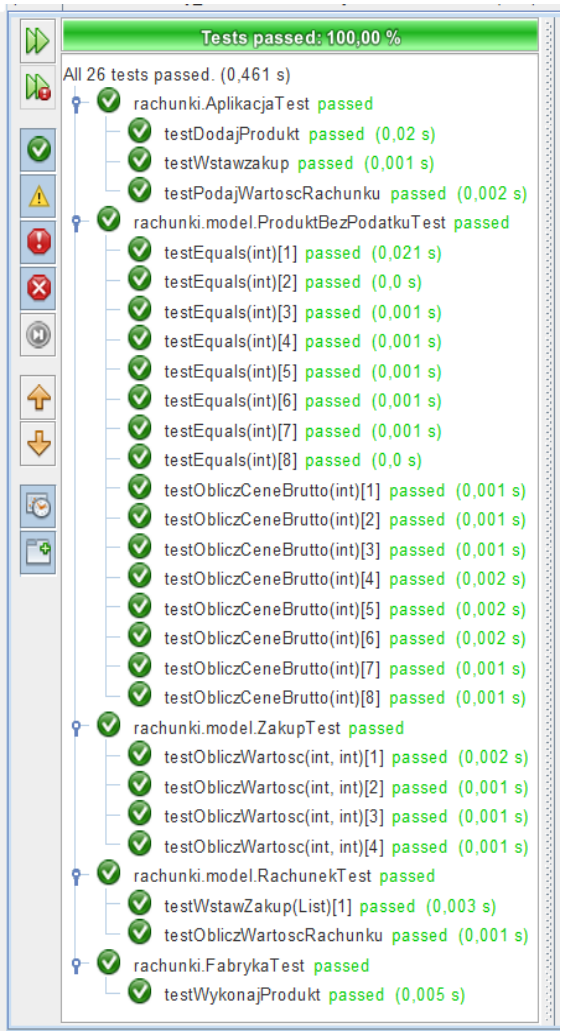


dodajProdukt
wstawZakup
wykonajProdukt

Wynik testu: *AplikacjaTest, FabrykaTest* z wyłączeniem metody testowej ***testPodajWartoscRachunku****()*

2.8.4. Wyniki testów wykonanych przez wszystkie klasy testujące – niezależnie od przypisanych kategorii

```
package Suite;

import org.junit.platform.suite.api.SelectPackages;
import org.junit.platform.suite.api.Suite;
@Suite
@SelectPackages("rachunki")
public class RachunkiTestSuite { }
```



dodajProdukt
wstawZakup
podajWartoscRachunku

Wynik testu klasy: Aplikacja

equals
equals
equals
equals
equals
equals
equals
equals

obliczCeneBrutto
obliczCeneBrutto
obliczCeneBrutto
obliczCeneBrutto
obliczCeneBrutto
obliczCeneBrutto
obliczCeneBrutto
obliczCeneBrutto

Wynik testu klasy: ProduktBezPodatku

obliczWartosc
obliczWartosc
obliczWartosc
obliczWartosc

Wynik testu klasy: Zakup

wstawZakup
obliczWartoscRachunku

Wynik testu klasy: Rachunek

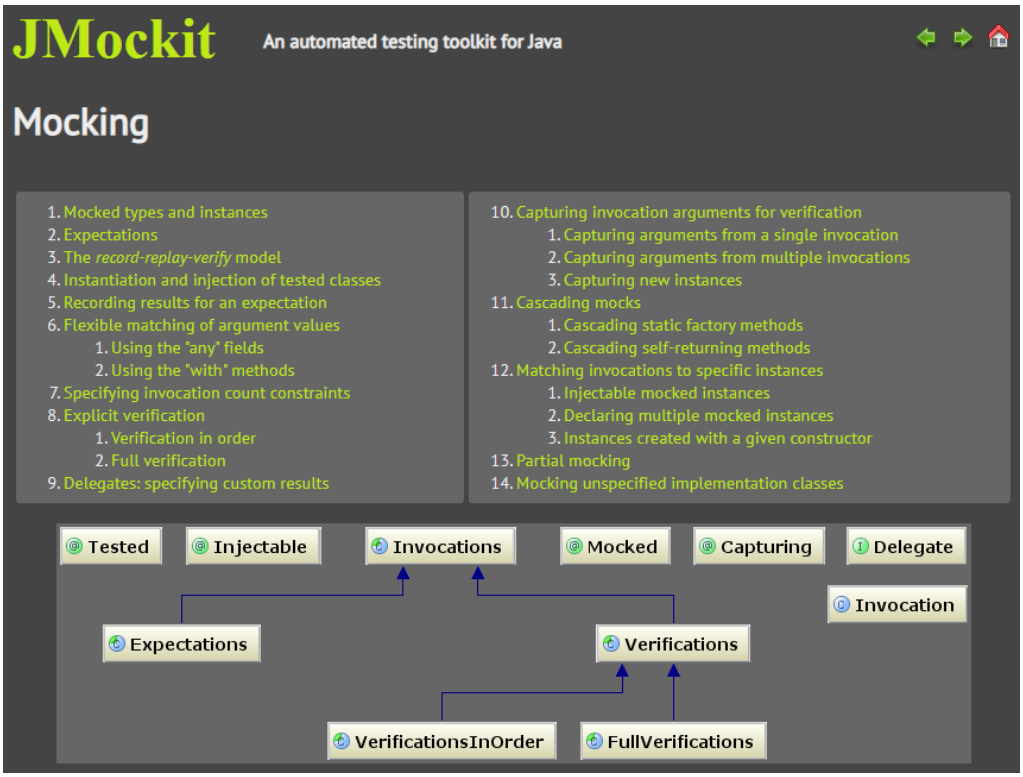
wykonajProdukt

Wynik *testu klasy*: *Fabryka*

3. Przykłady testowania oparte na symulacji obiektów za pomocą obiektów typu *JMockit* z biblioteki *JMockit-1.49*

W kontekście testowania zachowania obiektów powiązanych z obiektami, których zachowanie symuluje się za pomocą obiektów typu ***JMockit*,** możemy wyróżnić następujące 3 alternatywne fazy testowania (rysunek poniżej):

- **Faza zapisu** (nagrywania), podczas którego nagrywane są wywołania metod symulowanego obiektu za pomocą obiektów z rodziny ***Expectations.*** Symulację przeprowadza się za pomocą adnotacji ***@Mocked, @Injectable(p. 3.1)*** oraz ***@Capturing (p. 3.2, p. 3.3).***
- **Faza odtwarzania**, podczas której odtwarzane są wywołania nagranych wywołań metod, używane przez powiązane obiekty. Często nie jest to odwzorowanie jeden do jednego między wywołaniami nagranymi i odtwarzanymi.
- **Faza sprawdzenia**, w trakcie której można zweryfikować nagrane wywołania, które zostały wykorzystane w fazie odtwarzania za pomocą obiektu z rodziny ***Verifications***.



Rysunek przedstawiony powyżej pochodzi z tutoriala ***JMockit 1.49:*** JMockit - Tutorial - Mocking.

Biblioteka ***JMockit*** zapewnia bogate wsparcie w realizacji zautomatyzowanych symulacyjnych testów deweloperskich. Gdy używana jest symulacja, badanie skupia się na testowaniu metod klasy powiązanej z symulowaną klasą za pomocą testów jednostkowych, które zawierają interakcje z symulowanym kodem obiektów powiązanych. Zazwyczaj testowany kod w jednym teście jednostkowym jest zależny od kodu jednej powiązanej klasy, jednak w przypadku powiązań z wieloma klasami należy w tym teście jednostkowym zastosować interakcje z symulowanym kodem ważniejszych klas z tego zbioru.

Nie należy jednak zbyt rygorystycznie opierać testowanie jednostkowe o symulację kodu każdego powiązanego obiektu. Można je zastąpić testami integracyjnymi. Jednak w przypadku testów integracyjnych czasem warto zastosować symulację w przypadku braku implementacji fragmentów kodu lub trudności użycia kodu (odwołania do baz danych, wysłanie e-mail itp.) podczas uruchamiania testów integracyjnych.

Interakcja pomiędzy dwiema klasami zawsze przybiera formę wywołania metody lub konstruktora. Celem symulacji, w zakresie jednego testu jednostkowego, jest wywołanie metody lub zestawu wywołań metod klasy zależnej wraz z wartościami parametrów i zwracanych wyników. Często ważna jest kolejność wywołań metod klasy zależnej podczas symulacji zestawu wywołań.

Symulację przeprowadza się za pomocą adnotacji ***@Mocked*** (przykłady: 3.1, 3.3 - Dodatek 1; przykłady 1.5, 1.6 - Dodatek 2)***, @Injectable*** (przykład 3.2 - Dodatek 1; przykład 1.1 - Dodatek 2) oraz ***@Capturing*** (przykłady 1.2, 1.3 - Dodatek 2). Opisy tych adnotacji podano w podanych przykładach.
Symulacja wywołania metody może opierać się na specyfikacji jej algorytmu dzięki zastosowaniu obiektu typu ***Delegate*** (przykład 1.5 - Dodatek 2). Testowany obiekt w klasie testującej może być wystąpić w roli atrybutu tej klasy za pomocą adnotacji ***@Tested*** (przykład 1.1 - Dodatek 2)***.***

Tutoriale dla wersji ***JMockit 1.49*** są dostępne na stronach:
http://jmockit.github.io/tutorial/Introduction.html
http://jmockit.github.io/tutorial/Mocking.html
http://jmockit.github.io/changes.html

Wykonanie nowego projektu do testowania jednostkowego z wykorzystaniem biblioteki *JMockit 1.49* oraz JUnit 5

- Wykonanie nowego projektu typu ***Maven*** w środowisku ***ApacheNetbeans18*** wg p.2
- Poniżej podano zawartość pliku **pom.xml** reprezentującą definicję odwołań do bibliotek **JUnit5** i **JMockit 1.49** wspierających budowę testów wykonanych wg p. 3.1-3.4 oraz testów podanych w **Dodatek 2**.
- **<argLine>-javaagent:C:/Users/User/.m2/repository/org/jmockit/jmockit/1.49/jmockit-1.49.jar</argLine>**

W pliku pom.xml w zależności <configuration> podano ścieżkę dostępu do biblioteki **jmockit-1.49** umieszczanej przez ***Maven***, w której słowo ***User*** należy zastąpić nazwą użytkownika systemu nadaną np. podczas instalacji systemu operacyjnego ***Windows***.
Również dane projektu: nazwa projektu, nazwy pakietów itd. należy zaktualizować.

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>rachunki</groupId>
  <artifactId>Testy_Rachunki2Maven</artifactId>
  <version>1.0-SNAPSHOT</version>
  <packaging>jar</packaging>
  <properties>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
    <maven.compiler.source>17</maven.compiler.source>
    <maven.compiler.target>17</maven.compiler.target>
    <jmockit.version>1.49</jmockit.version>
   </properties>
  <dependencies>
     <dependency>
       <groupId>org.jmockit</groupId>
       <artifactId>jmockit</artifactId>
       <version>1.49</version>
       <scope>test</scope>
     </dependency>
     <dependency>
       <groupId>org.junit.jupiter</groupId>
       <artifactId>junit-jupiter-api</artifactId>
       <scope>test</scope>
     </dependency>
     <dependency>
       <groupId>org.junit.jupiter</groupId>
       <artifactId>junit-jupiter-params</artifactId>
       <scope>test</scope>
     </dependency>
```

```
    <dependency>
      <groupId>org.junit.jupiter</groupId>
      <artifactId>junit-jupiter-engine</artifactId>
      <scope>test</scope>
    </dependency>
     <dependency>
      <groupId>org.junit.platform</groupId>
      <artifactId>junit-platform-suite</artifactId>
      <scope>test</scope>
    </dependency>
    <dependency>
      <groupId>org.junit.platform.runner</groupId>
      <artifactId>junit-platform-runner</artifactId>
      <version>1.8.2</version>
      <scope>test</scope>
      <type>jar</type>
    </dependency>
    <dependency>
      <groupId>org.junit</groupId>
      <artifactId>junit-runner</artifactId>
      <version>1.8.2</version>
      <scope>test</scope>
      <type>jar</type>
    </dependency>
  </dependencies>
  <dependencyManagement>
    <dependencies>
      <dependency>
        <groupId>org.junit</groupId>
        <artifactId>junit-bom</artifactId>
         <version>5.10.0</version>
        <type>pom</type>
        <scope>import</scope>
      </dependency>
    </dependencies>
  </dependencyManagement>
  <repositories>
    <repository>
     <snapshots>
       <enabled>false</enabled>
     </snapshots>
     <id>central</id>
     <name>Central Repository</name>
     <url>https://repo.maven.apache.org/maven2</url>
    </repository>
  </repositories>
  <build>
  <pluginManagement>
   <plugins>
    <plugin>
     <groupId>org.apache.maven.plugins</groupId>
     <artifactId>maven-surefire-plugin</artifactId>
     <version>3.1.2</version>
      <configuration>
       <argLine>-javaagent:C:/Users/User/.m2/repository/org/jmockit/jmockit/1.49/jmockit-1.49.jar</argLine>
      </configuration>
    </plugin>
    </plugins>
  </pluginManagement>
 </build>
  <name>Testy_Rachunki2Maven</name>
</project>
```

3.1. Testowanie wybranych metod klasy *Zakup* są oparte na jednej jawnie deklarowanej instancji symulowanej klasy ***ProduktBezPodatku*** za pomocą adnotacji **@Mocked** w każdym teście i tworzeniu obiektu z rodziny ***ProduktBezPodatku*** za pomocą odpowiedniego konstruktora, który jest automatycznie symulowanym obiektem.

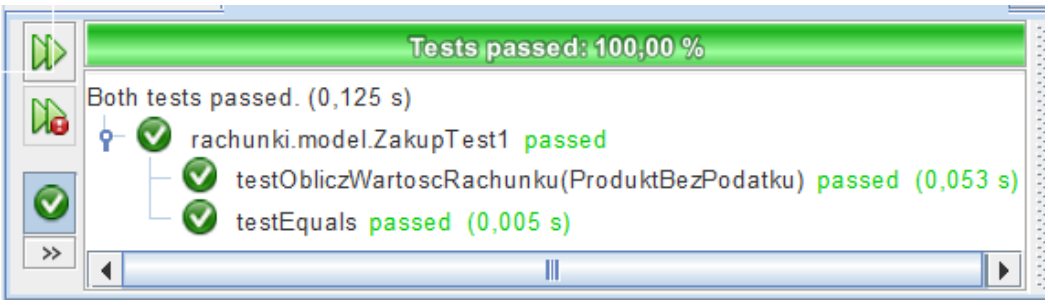
Zastosowanie adnotacji
@Test
@Tag
@Mocked

Fazy testowania metody **equals** klasy **Zakup** w metodzie testowej **testEquals**, opartej na domyślnej symulacji metody **equals** klasy **ProduktBezPodatku**
1)Bez jawnie zdefiniowanej fazy nagrywania
2) Odtwarzanie metody **equlas** klasy **Zakup**
3) Faza weryfikacji - **new VerificationsInOrder(), maxTimes**

Fazy testowania metody **obliczWartosc** klasy **Zakup** w metodzie testowej **testObliczWartoscRachunku**(), opartej na symulacji metod **getPodatek** oraz **obliczCeneBrutto** klasy **ProduktBezPodatku**
1)Faza nagrywania-**new Expectations(), result**
2) Odtwarzanie metody **obliczWartosc** klasy **Zakup**
3)Faza weryfikacji - **new VerificationsInOrder(), maxTimes**

```
package rachunki.model;
import mockit.Expectations;
import mockit.VerificationsInOrder;
import mockit.Mocked;
import static org.junit.jupiter.api.Assertions.*;
import org.junit.jupiter.api.Tag;
import org.junit.jupiter.api.Test;

@Tag("Entity")
public class ZakupTest1 {

  @Mocked
  ProduktBezPodatku produkt;

  @Test
  public void testEquals() {
    ProduktBezPodatku produkt2 = new ProduktZPodatkiem("8", 4, 7, new Promocja(50));
                                                          // dowolny konstruktor
    Zakup zakupy[] ={ new Zakup(2, produkt), new Zakup(2, produkt2)  };
     System.out.println("equals");
     for (int i = 0; i < 1; i++)
        for (int j = i; j < 2; j++)
           if (i == j)
              assertTrue(zakupy[i].equals(zakupy[i]));
           else
              assertFalse(zakupy[i].equals(zakupy[j]));

     new VerificationsInOrder() {
        {
           produkt.equals(any);          maxTimes = 2;   }
     };
}
```

```
@Test
public void testObliczWartoscRachunku(@Mocked ProduktBezPodatku produkt1) {
   ProduktBezPodatku produkt2 = new ProduktBezPodatku("8", 4);   // dowolny konstruktor
   Zakup zakupy[] =  { new Zakup(2, produkt1),  new Zakup(2, produkt2)};
   int podatki[] = {-1, 7};
   float ceny1[] = {0.9F,  6.48F};          //ceny brutto produktow
   float ceny2[] = {1.8F,  12.96F};         //ceny brutto zakupow

   System.out.println("obliczWartoscRachunku");
   new Expectations() {
     {
        produkt1.getPodatek();                result = podatki[0];
        produkt1.obliczCeneBrutto();          result = ceny1[0];
        produkt2.getPodatek();                result = podatki[1];
        produkt2.obliczCeneBrutto();          result = ceny1[1];
     }
   };
   for (int j = 0; j < 2; j++)
     assertEquals(zakupy[j].obliczWartosc(podatki[j]), ceny2[j], 0F);  //dodatkowy test assertEquals

   new VerificationsInOrder() {
     {
        produkt1.getPodatek();                maxTimes = 1;
        produkt1.obliczCeneBrutto();          maxTimes = 1;
        produkt2.getPodatek();                maxTimes = 1;
        produkt2.obliczCeneBrutto();          maxTimes = 1;
     }
   };
  }
}
```



3.2. Testowanie klasy ***Rachunek*** – testowanie za pomocą symulowania konkretnych instancji powiązanych klas (**@Injectable**)

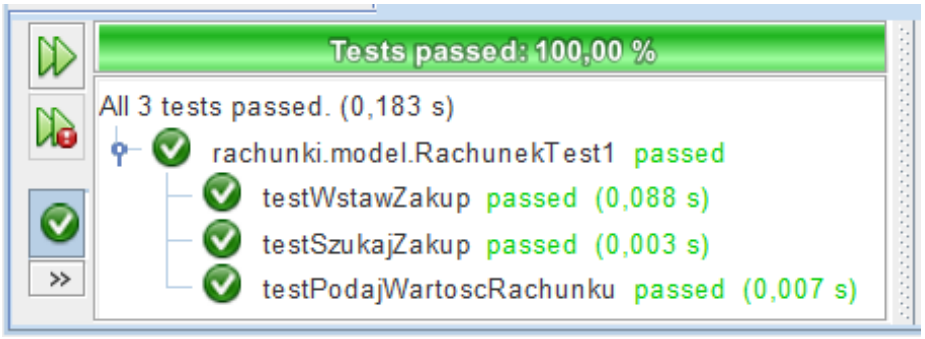
Zastosowanie adnotacji
@Test
@Tag
@Injectable

Fazy testowania metody **szukajZakup** w metodzie testowej **testSzukajZakup** klasy **Rachunek,** powiązanej z instancjami klasy **Zakup**, opartej na domyślnej symulacji metody **equals** klasy **Zakup.**
1)Domyślna faza nagrywania metody **equals** z klasy **Zakup**
2) Faza odtwarzania metody **szukajZakup** klasy **Rachunek**
3)Faza weryfikacji - **new VerificationsInOrder(), times**

Fazy testowania metody **wstawZakup** w metodzie testowej **testWstawZakup**() klasy **Rachunek,** powiązanej z instancjami klasy **Zakup**, opartej na symulacji metod **getIlosc** klasy **Zakup**
1)Faza nagrywania – **new Expectations(), returns**
2) Faza odtwarzania metody **wstawZakup**
3)Faza weryfikacji -new VerificationsInOrder(), maxTimes

Fazy testowania metody **obliczWartoscRachunku** w metodzie testowej **testPodajWartoscRachunku**() klasy **Rachunek**, powiązanej z instancjami klasy **Zakup**, opartej na symulacji metody **obliczWartosc** klasy **Zakup**
1)Faza nagrywania- **new Expectations(), result**
2)Faza odtwarzania metody **obliczWartoscRachunku**
3)Faza weryfikacji - **new VerificationsInOrder(), times**

```
package rachunki.model;

import java.util.Arrays;
import mockit.Injectable;
import mockit.Expectations;
import mockit.VerificationsInOrder;
import static org.junit.jupiter.api.Assertions.*;
import org.junit.jupiter.api.Tag;
import org.junit.jupiter.api.Test;

@Tag("Entity")
public class RachunekTest1 {

  @Injectable
  Zakup zakup1,zakup2, zakup3;

  @Test
  public void testSzukajZakup() {
    System.out.println("szukajZakup");
    Zakup zakupy[] = {zakup1, zakup2, zakup3};
    Rachunek rachunek = new Rachunek(1);
    rachunek.setZakupy(Arrays.asList(zakupy));
    for (int i = 0; i < 3; i++)
      assertEquals(rachunek.szukajZakup (zakupy[i]), zakupy[i]);     //dodatkowy test assertEquals

    new VerificationsInOrder() {
      {
        zakup1.equals(any);          times = 2;
        zakup2.equals(any);          times = 3;
        zakup3.equals(any);          times = 4;
      }
    };
  }
```



```
    @Test
    public void testWstawZakup() {
        System.out.println("wstawZakup");
        Zakup zakupy[] = {zakup1, zakup2, zakup3, zakup1};
        Rachunek rachunek = new Rachunek(1);
        new Expectations() {
            {
                zakup1.getIlosc();                  returns(1,1,1);
                zakup1.getIlosc();                  returns(2,2,2);
            }
        };
       for (int i = 0; i < 4; i++)
            rachunek.wstawZakup(zakupy[i]);
        assertEquals(rachunek.getZakupy().get(0).getIlosc(), 2);        //dodatkowy test assertEquals
        assertEquals(rachunek.getZakupy().size(), 3);                   //dodatkowy test assertEquals
        new VerificationsInOrder() {
            {
                zakup2.equals(any);         maxTimes = 1;
                zakup3.equals(any);         maxTimes = 2;
                zakup1.equals(any);         maxTimes = 1;
            }
        };
    }
    @Test
    public void testPodajWartoscRachunku() {
        Zakup zakupy[] = {zakup1, zakup2, zakup3};
        float wartoscirachunku[] = {9.8F, 0.0F, 0.0F, 0.0F, 4.88F, 14.68F};
        int podatki[] = {-1, 3, 7, 14, 22, -2};
        System.out.println("obliczWartoscRachunku");
        Rachunek rachunek = new Rachunek(1);
        new Expectations() {
            {
                zakupy[0].obliczWartosc(-1);        result = 1.8F;
                zakupy[1].obliczWartosc(-1);        result = 8F;
                zakupy[2].obliczWartosc(22);        result = 4.88F;
                zakupy[0].obliczWartosc(-2);        result = 1.8F;
                zakupy[1].obliczWartosc(-2);        result = 8.0F;
                zakupy[2].obliczWartosc(-2);        result = 4.88F;
            }
        };
        rachunek.setZakupy(Arrays.asList(zakupy));
        for (int i = 0; i < 6; i++)
            assertEquals(wartoscirachunku[i], rachunek.obliczWartoscRachunku(podatki[i]), 0F);
        new VerificationsInOrder() {
            {
                zakupy[0].obliczWartosc(-1);        times = 1;
                zakupy[1].obliczWartosc(-1);        times = 1;
                zakupy[2].obliczWartosc(22);        times = 1;
                zakupy[0].obliczWartosc(-2);        times = 1;
                zakupy[1].obliczWartosc(-2);        times = 1;
                zakupy[2].obliczWartosc(-2);        times = 1;
            }
        };
    }
}
```

3.3. Testowanie klasy ***Aplikacja*** – symulacja metody klasy ***Fabryka*** powiązanej z klasą ***Aplikacja***; testowanie metody ***dodajProdukt*** w zakresie poprawnych danych i niepoprawnych danych.

Zastosowanie adnotacji
@Test
@Tag
@Mocked

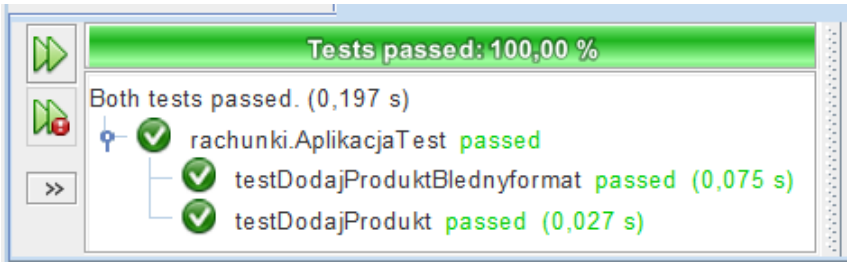
Fazy testowania metody **dodajProdukt** w metodzie testowej **testDodajProdukt** klasy **Aplikacja**, powiązanej z instancjami klasy **Fabryka**, opartej na symulacji metody **wykonajProdukt** klasy **Fabryka**
1)Faza nagywania-**new Expectations(), result**
2)Faza odtwarzania – wykonanie metody **dodajProdukt** klasy **Aplikacja**
3)Brak jawnej fazy weryfikacji

Fazy testowania metody **dodajProdukt** w metodzie testowej **testDodajProduktBlednyformat** klasy **Aplikacja**, powiązanej z instancjami klasy **Fabryka**, opartej na symulacji metody **wykonajProdukt** klasy **Fabryka** generującej wyjątek typu **IllegalFormatCodePointException** po podaniu niepoprawnych danych.
1)Faza nagywania-**new Expectations(), withNotNull**
2)Faza odtwarzania -– wykonanie metody **dodajProdukt** klasy **Aplikacja** po podaniu niepoprawnych danych
3)Brak jawnej fazy werfikacji

-

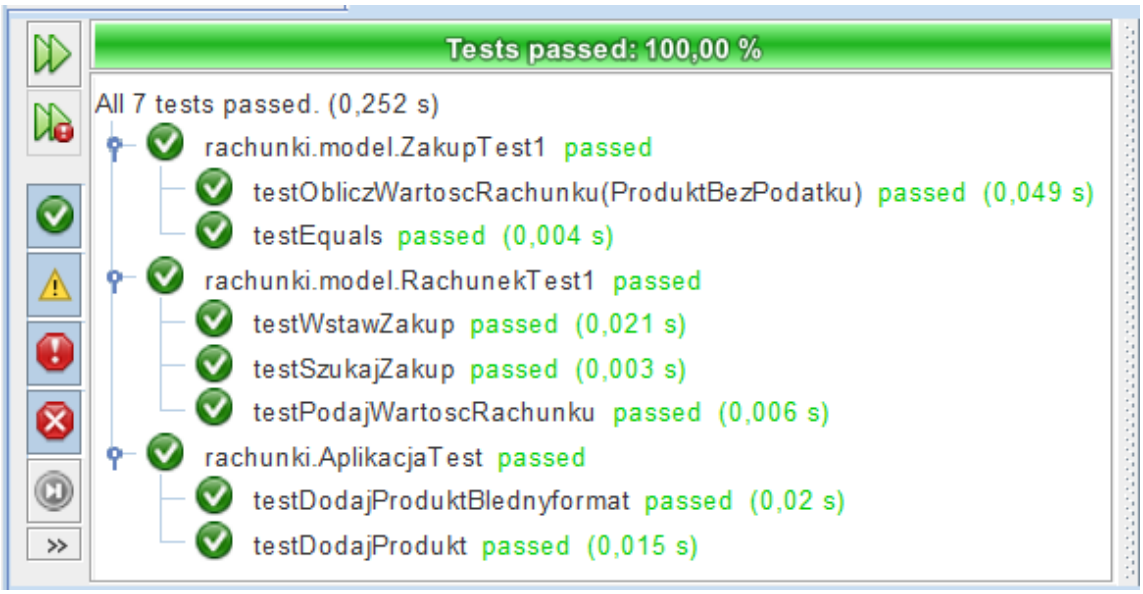
```
package rachunki;
import mockit.Expectations;
import mockit.Mocked;
import org.junit.jupiter.api.Test;
import static org.junit.jupiter.api.Assertions.*;
import org.junit.jupiter.api.Tag;
import rachunki.model.ProduktBezPodatku;
import rachunki.model.ProduktZPodatkiem;
import rachunki.model.Promocja;

@Tag("Entity")
@Tag("Control")
public class AplikacjaTest {
        ProduktBezPodatku produkty[] = {  new ProduktBezPodatku("1", 1),
                                          new ProduktZPodatkiem("3", 3, 14),
                                          new ProduktBezPodatku("5", 1, new Promocja(30)),
                                          new ProduktZPodatkiem("7", 3, 3, new Promocja(30)),
                                          new ProduktZPodatkiem("7", 3, 3, new Promocja(30)) };
      String dane[][] = new String[][]{
         {"0", "1", "1", "", ""}, {"2", "3", "3", "14", ""}, {"1", "5", "1", "30", ""},
         {"3", "7", "3", "3", "30"},{"3", "7", "3", "3", "30"}, {"4","1","1","",""} };
@Mocked
Fabryka fabryka;
@Test
public void testDodajProdukt() {
      System.out.println("dodajProdukt");
      new Expectations() {
        {
           fabryka.wykonajProdukt(dane[0]);          result = produkty[0];
           fabryka.wykonajProdukt(dane[1]);          result = produkty[1];
           fabryka.wykonajProdukt(dane[2]);          result = produkty[2];
           fabryka.wykonajProdukt(dane[3]);          result = produkty[3];
        }
      };
      Aplikacja aplikacja = new Aplikacja();
      for (int i = 0; i < 5; i++) {
        aplikacja.dodajProdukt(dane[i]);
        if(i<4)
            assertEquals(produkty[i], aplikacja.getProdukty().get(i));
        else
            assertEquals(produkty[i], aplikacja.getProdukty().get(i-1)); } }
```

```
    @Test
      public void testDodajProduktBlednyformat() {
        System.out.println("dodajProdukt_niepoprawny_format_danych");
        new Expectations() {
          {
            fabryka.wykonajProdukt((String[]) withNotNull());
          }
        };
        Aplikacja aplikacja = new Aplikacja();
         aplikacja.dodajProdukt(dane[5]);
      }
}
```



3.4. Tworzenie zestawów testów

```
package Suite;

import org.junit.platform.suite.api.SelectClasses;
import org.junit.platform.suite.api.Suite;
import rachunki.model.*;
import rachunki.*;

@Suite
@SelectClasses({ZakupTest1.class, RachunekTest1.class, AplikacjaTest.class})
public class RachunkiTestSuite { }
```



Dodatek 2

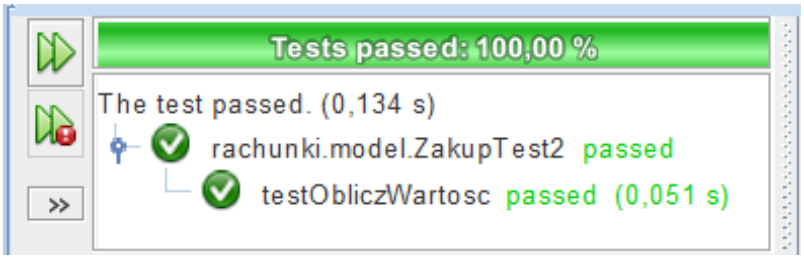
Pozostałe testy z wykorzystaniem biblioteki *JMockit 1.49,* prezentujące wybrane z możliwości symulowania własności obiektów podczas tworzenia oprogramowania.

1.1. Testowanie klasy *Zakup* – oparte na instancji symulowanej instancji klasy *ProduktBezPodatku* (**@Injectable**) oraz symulacji atrybutu *ilosc* klasy *Zakup* oraz definicja instancji klasy testowanej *Zakup* (**@Tested**)

Zastosowanie adnotacji
@Test
@Tag
@Injectable
@Tested

Fazy testowania metody **obliczWartosc** klasy **Zakup** w metodzie testowej **testObliczWartosc**(), opartej na symulacji metod **getPodatek** oraz **obliczCeneBrutto** klasy **ProduktBezPodatku**
1)Nagrywanie **– new Expectations**
2)Odtwarzanie metody **obliczWartosc** klasy **Zakup**
3)Bez jawnej fazy weryfikacji

```
package rachunki.model;

import mockit.Expectations;
import mockit.Injectable;
import mockit.Tested;
import static org.junit.jupiter.api.Assertions.*;
import org.junit.jupiter.api.Tag;
import org.junit.jupiter.api.Test;

@Tag("Entity")
public class ZakupTest2 {
  @Tested
  Zakup tested;       //przykład automatycznego tworzenia testowanej klasy wraz z definicją symulowanych pól: produkt i ilosc
  @Injectable
  ProduktBezPodatku produkt1;   //symulowanie konkretnej instancji symulowanej klasy powiązanego z testowaną klasą Zakup
  @Injectable
  int ilosc = 2;                //symulowanie wartosci pola ilosc w klasie testowanej Zakup

  @Test
  public void testObliczWartosc(/*@Injectable ("2")  int ilosc*/) {      //lub jako parametr
    new Expectations() {
      {
        produkt1.getPodatek();                result = -1;
        produkt1.obliczCeneBrutto();          result = 14;
      }
    };
    assertEquals(tested.obliczWartosc(-1), 28.0F, 0F);                  //dodatkowy test assertEquals
  }
}
```

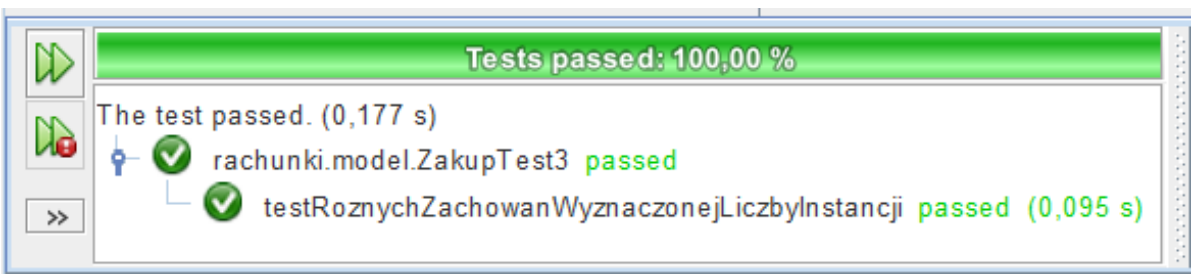


1.2. Testowanie klasy *Zakup* – specyfikacja zachowania kolejno tworzonych instancji w przyszłości (**@Capturing**)

Zastosowanie adnotacji
@Test
@Tag
@Capturing

Fazy testowania metody **obliczWartosc** klasy **Zakup** w metodzie testowej **testRoznychZachowanWyznaczonejLiczbyInstancji**(), opartej na symulacji metod **getPodatek** oraz **obliczCeneBrutto** klasy **ProduktBezPodatku**
1)Nagrywanie **– new Expectations**
2)Odtwarzanie metody **obliczWartosc** klasy **Zakup**
3)Bez jawnej fazy weryfikacji

```
package rachunki.model;

import mockit.Expectations;
import mockit.Capturing;
import static org.junit.jupiter.api.Assertions.*;
import org.junit.jupiter.api.Tag;
import org.junit.jupiter.api.Test;

@Tag("Entity")
public class ZakupTest3 {
   @Capturing
    ProduktBezPodatku produkt1;     // instancja odtwarzająca nagraną metodę z rodziny obiektów ProduktBezPodatku
    @Capturing
    ProduktBezPodatku produkt2;   // instancja odtwarzająca nagraną metodę z rodziny obiektów ProduktBezPodatku

   @Test
   public void testRoznychZachowanWyznaczonejLiczbyInstancji(
        /*@Capturing ProduktBezPodatku produkt1,
         @Capturing ProduktBezPodatku produkt2*/)                    //lub jako parametry metody testującej
   {
      new Expectations() {
        {
           produkt1.obliczCeneBrutto();    result = 6.48F;        //nagranie wyniku metody
           produkt2.obliczCeneBrutto();    result = 4.88F;        //nagranej wyniku metody
        }
      };
      assertEquals(6.48F, produkt1.obliczCeneBrutto(), 0F); //test 1 pierwszej   symulowanej instancji produkt1
      assertEquals(6.48F, produkt1.obliczCeneBrutto(), 0F); //test 2 pierwszej   symulowanej instancji produkt1
      assertEquals(4.88F, produkt2.obliczCeneBrutto(), 0F); //test 1 pierwszej   symulowanej instancji produkt2
      assertEquals(4.88F, produkt2.obliczCeneBrutto(), 0F); //test 2 pierwszej   symulowanej instancji produkt2
      Zakup zakup1 = new Zakup(1, produkt1);
      assertEquals(zakup1.obliczWartosc(0), 6.48F, 0F);  //test metody klasy powiązanej: 1*4.88F
      Zakup zakup2 = new Zakup(2, produkt2);
      assertEquals(zakup2.obliczWartosc(0), 9.76F, 0F); //test metody klasy powiązanej: 2*4.88F
      Zakup zakup3 = new Zakup(3, produkt2);
      assertEquals(zakup3.obliczWartosc(0), 14.64F, 0F); //test metody klasy powiązanej: 3*4.88F
   }
}
```

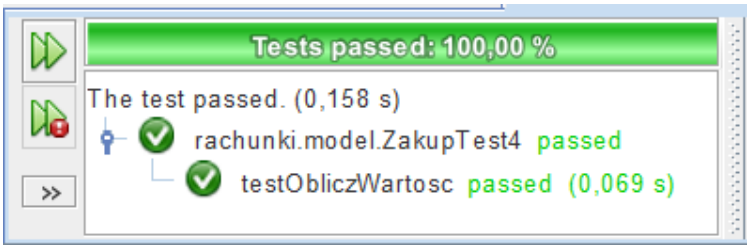


1.3. Testowanie klasy ***Zakup*** – Symulowanie metod klas potomnych lub implementacji interfejsów (**@Capturing**)

Zastosowanie adnotacji
@Test
@Tag
@Capturing

Fazy testowania metody **obliczWartosc** klasy **Zakup** w metodzie testowej **testObliczWartosc**() powiązanej z instancją klasy ProduktZPodatkiem, opartej na symulacji metod **getPodatek** oraz **obliczCeneBrutto** klasy **ProduktBezPodatku**
1)Nagrywanie **– new Expectations**
2)Odtwarzanie metody **obliczWartosc** klasy **Zakup**
3)Bez jawnej fazy weryfikacji

```
package rachunki.model;

import mockit.Capturing;
import mockit.Expectations;
import static org.junit.jupiter.api.Assertions.*;
import org.junit.jupiter.api.Tag;
import org.junit.jupiter.api.Test;

@Tag("Entity")
public class ZakupTest4 {
   @Capturing
   ProduktBezPodatku produkt1;

   @Test
   public void testObliczWartosc() {
      new Expectations() {
         {
            produkt1.getPodatek();                result = 7F;
            produkt1.obliczCeneBrutto();          result = 3.21F;
         }
      };
      ProduktZPodatkiem produkt2 = new ProduktZPodatkiem("2", 3, 7);
      Zakup zakup = new Zakup(2, produkt2);
      assertEquals(6.42F, zakup.obliczWartosc(7) , 0F);
   }
}
```



1.4. Testowanie klasy ***Zakup*** – dwa przypadki częściowej symulacji: symulacja wybranych metod wybranej klasy oraz symulacja metod instancji wybranej klasy realizowane za pomocą przeciążonych konstruktorów klas z rodziny ***Expectations*.**

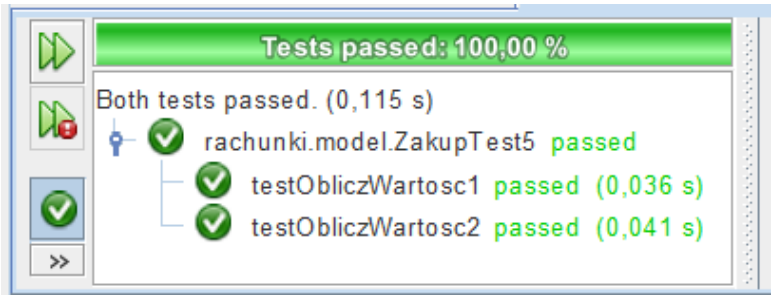
Zastosowanie adnotacji
@Test
@Tag

Częściowe symulowanie metod wielu instancji danej klasy (**ProduktBezPodatku**) Fazy testowania metody **obliczWartosc** klasy **Zakup** w metodzie testowej **testObliczWartosc1** powiązanej z instancją klasy **ProduktBezPodatku**, opartej na symulacji metody **getPodatek** klasy **ProduktBezPodatku**
1)Nagrywanie **– new Expectations ()**
2)Odtwarzanie metody **obliczWartosc** klasy **Zakup**
3)Bez jawnej fazy weryfikacji

Częściowe symulowanie metod jednej instancji Fazy testowania metody **obliczWartosc** klasy **Zakup** w metodzie testowej **testObliczWartosc2** powiązanej z instancją klasy **ProduktBezPodatku**, opartej na symulacji metod **getPodatek** oraz **obliczCzescBruttoCeny** klasy **ProduktBezPodatku**
1)Nagrywanie **– new Expectations(produkt1)**
2)Odtwarzanie metody **obliczWartosc** klasy **Zakup**
3)Bez jawnej fazy weryfikacji

```
package rachunki.model;

import mockit.Expectations;
import static org.junit.jupiter.api.Assertions.*;
import org.junit.jupiter.api.Tag;
import org.junit.jupiter.api.Test;

@Tag("Entity")
public class ZakupTest5 {

  @Test
  public void testObliczWartosc1() {
    ProduktBezPodatku produkt1 = new ProduktBezPodatku("1", 1);
    new Expectations() {
      {
        produkt1.getPodatek();   result = -1F;
      }
    };
      //użycie niesymulowanych konstruktorów
    ProduktBezPodatku produkt2 = new ProduktBezPodatku("2", 2);
    ProduktBezPodatku produkt3 = new ProduktBezPodatku("6", 2, new Promocja(50));

      // odtwarzanie metody symulowanej przez dwie instancje
    assertEquals(-1F, produkt2.getPodatek(), 0F);
    assertEquals(-1F, produkt3.getPodatek(), 0F);

      //wykonanie metod niesymulowanych
    assertEquals(produkt2.obliczCeneBrutto(), 2F, 0F);
    assertEquals(produkt3.obliczCeneBrutto(), 0.9F, 0F);

      //klasa korzystająca z metod symulowanych i niesymulowanych typu ProduktBezPodatku
    Zakup zakup1 = new Zakup(4, produkt2);
    Zakup zakup2 = new Zakup(1, produkt3);
    assertEquals(zakup1.obliczWartosc(-1), 8F, 0F);
    assertEquals(zakup2.obliczWartosc(-1), 0.9F, 0F);
  }
```

```
@Test
 public void testObliczWartosc2() {
     ProduktBezPodatku produkt1 = new ProduktBezPodatku("6", 2, new Promocja(50));
     new Expectations(produkt1) {
       {
          produkt1.obliczCzescBruttoCeny();          result = -1.1F;
          produkt1.getPodatek();                     result = -1;
       }
     };
        // odtwarzanie nagranych metod
     assertEquals(-1.1F, produkt1.obliczCzescBruttoCeny(), 0F);
     assertEquals(-1, produkt1.getPodatek(), 0F);

        // odtwarzanie nienagranych metod symulowanej instancji
     assertEquals(produkt1.obliczCeneBrutto(), 0.9F, 0F);
     assertEquals(produkt1.getNazwa(), "6");

        //testowanie klasy powiązanej z jedną instancją klasy częściowo symulowanej
     Zakup zakup = new Zakup(1, produkt1);
     assertEquals(zakup.obliczWartosc(-1), 0.9F, 0F);
   }
}
```

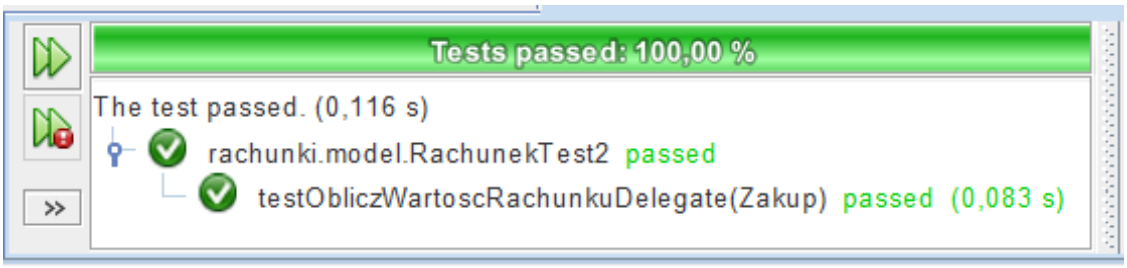


1.5. Testowanie klasy ***Rachunek***- symulowanie metody za pomocą specyfikacji jej działania (**Delegate**)

Zastosowanie adnotacji
@Test
@Tag
@Mocked

Fazy testowania metody **obliczWartoscRachunku** w metodzie testowej **testObliczWartoscRachunkuDelegate** klasy **Rachunek**, powiązanej z instancjami klasy **Zakup**, opartej na symulacji metody **obliczWartosc** klasy **Zakup** za pomocą konstruktora klasy **Delegate**
1)Faza nagrywania- **new Expectations(), result, new Delegate**
2)Faza odtwarzania metody **obliczWartoscRachunku** klasy **Rachunek**
3)Brak jawnej fazy weryfikacji

```
package rachunki.model;

import mockit.Delegate;
import mockit.Expectations;
import mockit.Mocked;
import static org.junit.jupiter.api.Assertions.*;
import org.junit.jupiter.api.Tag;
import org.junit.jupiter.api.Test;

@Tag("Entity")
public class RachunekTest2 {

   @Test
   public void testObliczWartoscRachunkuDelegate(@Mocked final Zakup zakup) {
      new Expectations() {
         {
            zakup.obliczWartosc(anyFloat);
            result = new Delegate() {
               float DelegateMethod(float i)
                { if (i == -2 || i == 14)
                     return 3.42F;
                  else
                     return 0F;}
            };
         }
      };
      Rachunek rachunek = new Rachunek(1);
      rachunek.wstawZakup(zakup);
      assertEquals(rachunek.obliczWartoscRachunku(-2), 3.42F, 0F);
      assertEquals(rachunek.obliczWartoscRachunku(14), 3.42F, 0F);
      assertEquals(rachunek.obliczWartoscRachunku(7), 0F, 0F);
   }
}
```



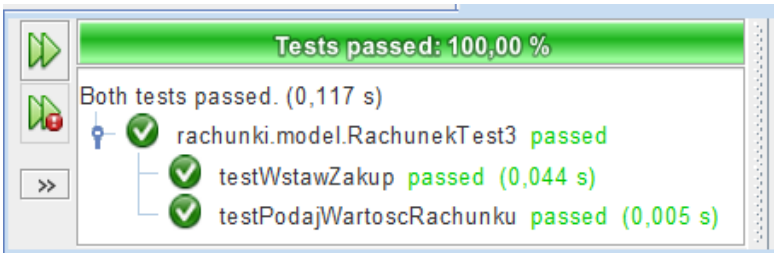
1.6. Testowanie klasy ***Rachunek***- przechwytywanie argumentów metod symulowanych klas (metoda **withCapture**)

Zastosowanie adnotacji
@Test
@Tag
@Mocked

Pobranie parametrów z jednego wywołania symulowanej metody. Fazy testowania metody **obliczWartoscRachunku** w metodzie testowej **testPodajWartoscRachunku**() klasy **Rachunek**
1)Brak jawnej fazy nagrywania
2)Faza odtwarzania metody **obliczWartoscRachunku**
3)Faza weryfikacji – **new Verifications, withCapture**

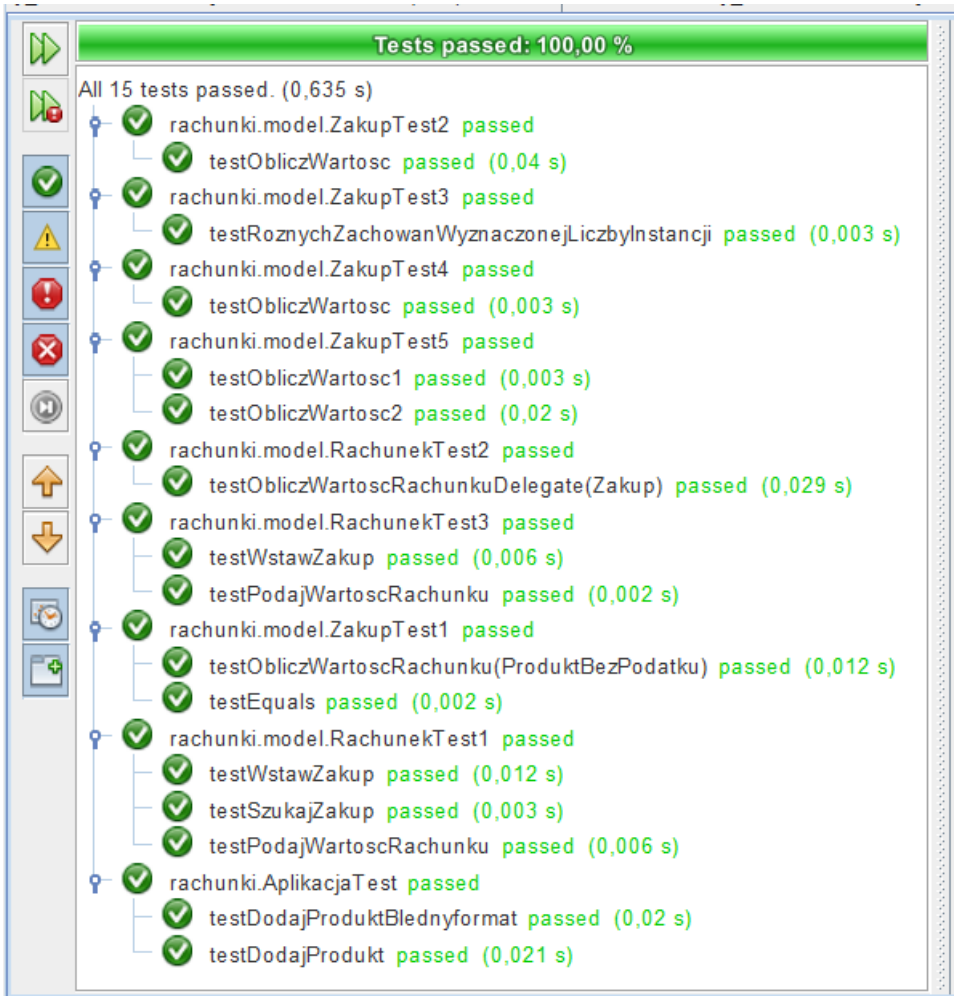
Pobranie parametrów z wielu wywołań symulowanej metody. Fazy testowania metody **wstawZakup** w metodzie testowej **testWstawZakup**() klasy **Rachunek**
1)Brak jawnej fazy nagrywania
2)Faza odtwarzania metody **wstawZakup**
3)Faza weryfikacji – **new Verifications, withCapture**

```
package rachunki.model;

import java.util.ArrayList;
import java.util.List;
import mockit.Mocked;
import mockit.Verifications;
import static org.junit.jupiter.api.Assertions.*;
import org.junit.jupiter.api.Tag;
import org.junit.jupiter.api.Test;

@Tag("Entity")
public class RachunekTest3 {

@Mocked
Rachunek rachunek;
@Test
public void testPodajWartoscRachunku() {
   System.out.println("Podaj_-wartosc - pobranie parametrów z jednego symulowanego wywołania metody");
   new Rachunek(1).obliczWartoscRachunku(-2);
     new Verifications() {
       {  float d;
          rachunek.obliczWartoscRachunku(d = withCapture());
          assertTrue(d < 0.0); }
     };
  }
@Test
public void testWstawZakup() {
   System.out.println("WstawZakup - pobranie parametrów z wielu symulowanych wywołań metody");
   ProduktBezPodatku produkty[] = {
      new ProduktBezPodatku("2", 2), new ProduktZPodatkiem("4", 4, 22),
      new ProduktBezPodatku("6", 2, new Promocja(50)),
      new ProduktZPodatkiem("8", 4, 7, new Promocja(50))  };
     Zakup zakupy[] = {  new Zakup(2, produkty[0]), new Zakup(3, produkty[1]),
                         new Zakup(2, produkty[2]), new Zakup(1, produkty[3])   };
     for (int i = 0; i < 4; i++)
       rachunek.wstawZakup(zakupy[i]);
     new Verifications() {
       {
          List<Zakup> lista = new ArrayList<>();
          rachunek.wstawZakup(withCapture(lista));
          assertEquals(4, lista.size());  }
     };
  }
}
```



1.7. Tworzenie zestawów testów – uzupełnienie p.3.4 z Dodatek 1.

```
package Suite;

import org.junit.platform.suite.api.SelectClasses;
import org.junit.platform.suite.api.Suite;
import rachunki.model.*;

@Suite
@SelectClasses({RachunkiTestSuite.class, ZakupTest2.class, ZakupTest3.class, ZakupTest4.class,
                ZakupTest5.class, RachunekTest2.class, RachunekTest3.class})
public class RachunkiTestSuite2 {    }
```



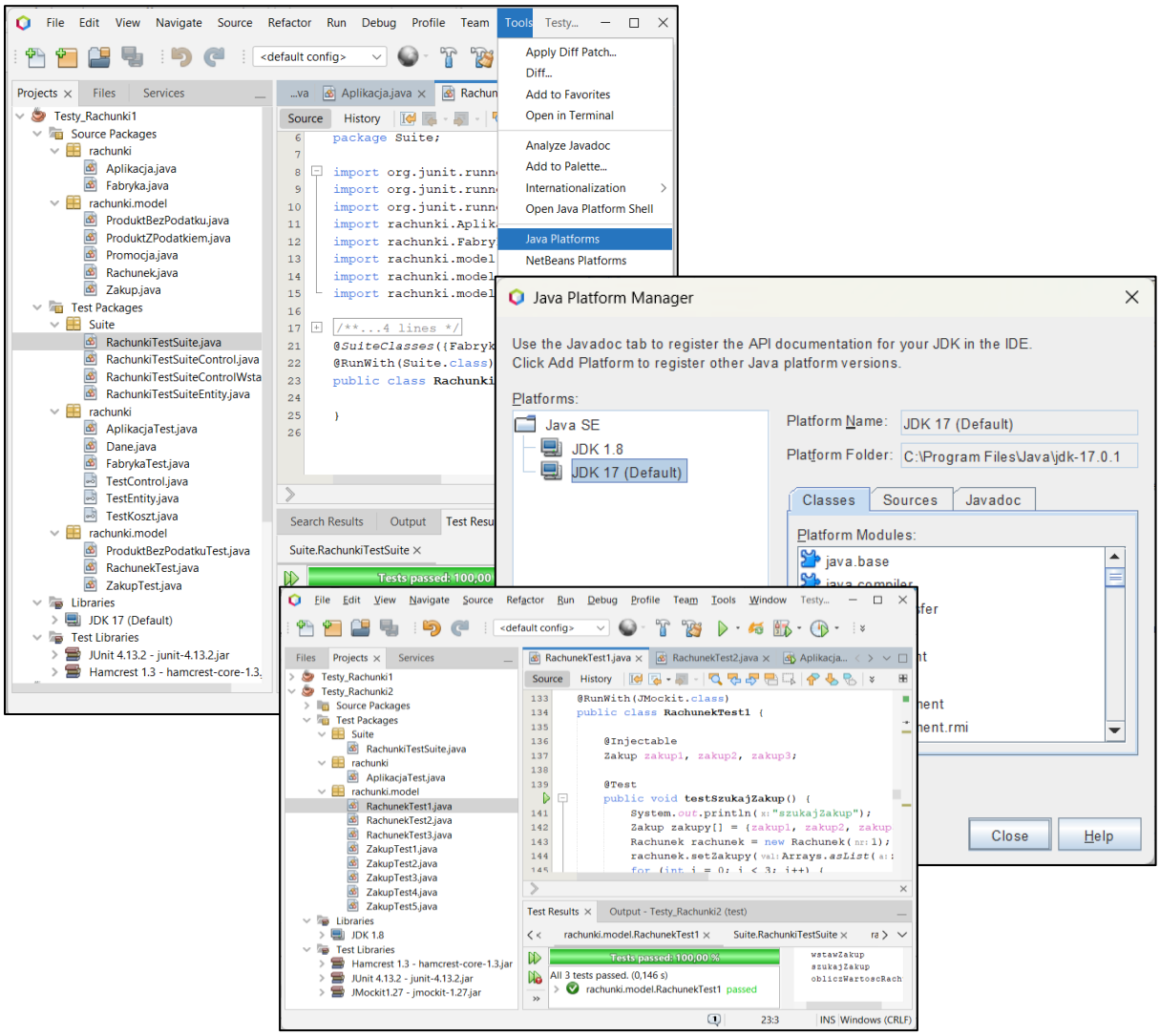
Część 2

Testy jednostkowe
JUnit 4, JMockit 1.27

Cel laboratorium:
Nabycie umiejętności tworzenia testów jednostkowych za pomocą narzędzi JUnit oraz JMockit - (20 Most Popular Unit Testing Tools in 2022 (softwaretestinghelp.com), Mockito vs EasyMock vs JMockit | Baeldung**).**

7. Wg wskazówek podanych w **Dodatku 2**, należy zainstalować biblioteki ***JUnit 4.13.2***, ***Hamcrest 1.3*** i ***JMockit 1.27*** oraz wykonać projekt w środowisku ***Apache NetBeans 18***. Instalacja narzędzia ***Apache Netbeans 18*** z wybraną domyślną wersją ***Java SE 17*** przedstawiono na stronie przedmiotu:
   http://zofia.kruczkiewicz.staff.iiar.pwr.wroc.pl/index.php?id=INEK011.
   Należy wybrać taką wersję ***Java SE*** w tworzonym projekcie programistycznym, aby była nie była wyższa od wersji domyślnej ***Java SE*** w narzędziu ***Apache NetBeans 18***. Projekt powinien zawierać pakiet z klasami do testowania wykonanymi podczas lab2-11. Następnie, wg kolejnych wskazówek poniżej, należy dodawać testy ***JUnit*** wybranych klas. **Projekt z testami** oparty na bibliotekach ***JUnit 4.13.2*** i ***Hamcrest 1.3*** może używać platformy Java: ***Java SE 17*** (dopasowanej do wersji domyślnej ***Java SE*** w narzędziu ***Apache NetBeans 18***) natomiast dodatkowo oparty na bibliotece ***JMockit 1.27*** może jedynie używać platformy Java: ***Java SE 8***.

   Poniżej pokazano sposób dodania dodatkowych platform ***Java SE***, które należy dodać do tworzonych aplikacji z zakładki ***Tools*** i pozycji ***Java Platforms.***



8. Należy wykonać test jednostkowy metod klasy, która stanowi klasę końcową w łańcuchu powiązań na diagramie klas lub/i może być powiązana w relacji 1 do 0..1 z inną klasą – podobnie jak klasa typu ***Fabryka*** lub klasy z rodziny ***ProduktBezPodatku***. Należy zastosować w metodach testowych metody klasy ***Assert*** z pakietu org.junit biblioteki ***JUnit 4.12*** oraz adnotacje: ***Test, Parameter, Parameters, RunWith(Parameterized.class), Rule***.
Dane wzorcowe, wykorzystywane do weryfikacji wyników testowanych metod za pomocą metod klasy ***Assert*** należy umieścić w dodatkowej klasie, podobnie jak klasa ***Dane*** z p.2.1 **Dodatku 1**. Przykłady testów podano w p.2.2 i p.2.3 **Dodatku 1**.
W tabelce poniżej podano informację dotyczącą wyboru metod do testowania oraz przykładów rozwiązań.

Grupa	Liczba metod do testowania	Przykłady testowanych metod		Przykłady testów	
		Fabryka	rodzina **ProduktBezPodatku**	**FabrykaTest** (p.2.2 Dodatek1)	**ProduktBezPodatkuTest** (p.2.3 Dodatek1)
1 osoba	1	wykonajProdukt	obliczCeneBrutto	testWykonajProdukt	testObliczCeneBrutto
2 osoby	2	wykonajProdukt, obliczCeneBrutto		testWykonajProdukt,	testObliczCeneBrutto

9. Należy wykonać test jednostkowy metod klasy, która stanowi klasę w łańcuchu powiązań na diagramie klas lub/i może być powiązana w relacji „1 do 1” lub „1 do 1..*” z inną/innymi klasami – podobnie jak klasy typu ***Zakup*** z klasą z rodziny ***ProduktBezPodatku*** lub klasa ***Rachunek*** z klasą ***Zakup***. Należy zastosować w metodach testowych metody klasy ***Assert*** z biblioteki ***JUnit 4.13.2*** oraz adnotacje: ***Test, Parameter, Parameters, RunWith(Parameterized.class), FixMethodOrder(MethodSorters.NAME_ASCENDING), Rule***.

   Przykłady testów podano w p.2.4 i p.2.5 **Dodatku 1**.
   Dane wzorcowe wykorzystywane do weryfikacji wyników testowanych metod za pomocą metod klasy ***Assert***, należy umieścić w dodatkowej klasie (zdefiniowanej w p. 2), podobnie jak klasa ***Dane*** z p.2.1 **Dodatku 1**. Kryterium wyboru metod powinno uwzględniać fakt, że metody wybrane w p.1 są wywoływane w metodach klas wybranych w p.2.
   Poniżej, w tabelce poniżej podano informację dotyczącą wyboru metod do testowania oraz przykładów rozwiązań.

Grupa	Liczba metod do testowania	Przykłady testowanych metod		Przykłady testów	
		Zakup	**Rachunek**	**ZakupTest** (p.2.4 Dodatek1)	**RachunekTest** (p.2.5 Dodatek1)
1 osoba	1	obliczWartosc	wstawZakup,	testObliczWartosc	test1WstawZakup
2 osoby	2	-	wstawZakup, obliczWartoscRachunku	-	test1WstawZakup test2ObliczWartoscRachunku

10. Należy wykonać testy jednostkowe wybranych metod klasy opartej na wzorcu ***Fasada***, podobnie jak klasa ***Aplikacja***. Wybrane metody tej klasy do testowania powinny wywoływać wybrane metody z p.2 lub p.1.
    Należy zastosować w metodach testowych metody klasy ***Assert*** z biblioteki ***JUnit 4.13.2*** oraz adnotacje: ***Test, Parameter, Parameters, RunWith(Parameterized.class), FixMethodOrder(MethodSorters.NAME_ASCENDING), Rule***. Przykłady testów podano w p.2.6 **Dodatku 1**. Poniżej, w tabelce podano informację dotyczącą wyboru metod do testowania oraz przykładów rozwiązań.

Grupa	Liczba metod do testowania	Przykłady testowanych metod **Aplikacja** (p.2.6 Dodatek1)	Przykłady testów **AplikacjaTest** (p.2.6 Dodatek1)
1 osoba	2	dodajProdukt, wstawZakup,	test1DodajProdukt, test2WstawZakup
2 osoby	3	dodajProdukt, podajWartoscRachunku, wstawZakup,	test1DodajProdukt, test3PodajWartoscRachunku, test2WstawZakup

11. Należy wykonać zestawy testów, podobnie jak pokazano w p.2.7 **Dodatku 1** stosując adnotację ***Category*** w klasach z metodami testującymi, wykonanych w p. 1, 2, 3 oraz ***@Categories.SuiteClasses, @RunWith(Categories.class), @Categories.IncludeCategory, Categories.ExcludeCategory***. Poniżej, w tabelce podano informację dotyczącą wyboru metod do testowania oraz przykładów rozwiązań. Aby zastosować adnotację ***Category,*** należy utworzyć puste interfejsy reprezentujące wybrane kategorie (***Java Interface***) – przykłady zastosowania tej adnotacji podano w **Dodatku 1**.

Grupa	Zestaw wszystkich testów	Przykłady zestawu testów wyznaczonych wg kategorii (adnotacja Category) (p.2.7 Dodatek1)
1 osoba	RachunkiTestSuite	RachunkiTestSuiteEntity RachunkiTestSuiteControl
2 osoby	RachunkiTestSuite	RachunkiTestSuiteEntity RachunkiTestSuiteControl RachunkiTestSuiteControlWstaw

12. Należy metody, wybrane do testowania w jednym punktów instrukcji 3-4, przetestować wykorzystując mechanizm symulowania obiektów powiązanych. Dodatkowo, należy kierować się przykładami z p. 3.1-3.3 **Dodatku 1** oraz przykładami z **Dodatku 3** instrukcji**.** Należy uwzględnić proponowane tam elementy symulacji technologii ***JMockit***. Poniżej, w tabelce podano przykłady testów, wykonanych przez grupy: jedno- i dwuosobowe.

Grupa Liczba osób	Liczba metod do testowania	Przykłady metod: 1-przykład		Przykłady metod: 2-przykład		Przykłady testów –
		Przykłady symulowanych metod	Przykłady testowanych metod	Przykłady symulowanych metod	Przykłady testowanych metod	
		Klasy z rodziny ProduktBezPodatku	Klasa Zakup	Klasa Zakup	Klasa Rachunek	
1	2	equals	equals	getIlosc, dodajIlosc	wstawZakup,	**p. 3.1 Dodatek 1** lub **p. 3.2 Dodatek 1**
		obliczCeneBrutto, getPodatek	obliczWartosc	obliczWartosc	obliczWartosc Rachunku	

Grupa Liczba osób	Liczba metod do testowania	Przykłady metod: 1-przykład		Przykłady metod: 2-przykład		Przykłady testów –
		Przykłady symulowanych metod	Przykłady testowanych metod	Przykłady symulowanych metod	Przykłady testowanych metod	
		Klasa Zakup	Klasa Rachunek	Klasa Fabryka	Klasa Aplikacja	
2	4	getIlosc, dodajIloscProduktu	wstawZakup,	wykonajProdukt bez generowania wyjatku	dodajProdukt bez generowania wyjątku	**p. 3.2 Dodatek 1**, **p.3.3 Dodatek 1**
		obliczWartosc	obliczWartosc Rachunku	wykonajProdukt z generowaniem wyjatku	dodajProdukt z generowaniem wyjątku	

Dodatek 1

Testy jednostkowe oprogramowania "System sporządzania rachunków" – Dodatek 2 zawiera ważne informacje wspomagające tworzenie testów oraz informacje o przydatnych tutorialach.

1. **Modyfikacje kodu przedstawionego w Dodatku 1 z instrukcji laboratoryjnych 5-7 -** po wykonaniu projektu wg informacji podanej w p. 2 (bieżący Dodatek 1) i umieszczeniu tam kodu przedstawianego w bieżącym Dodatku 1 z instrukcji laboratoryjnych 5-7, należy dokonać podanych dalej w p.1.1-1.2 modyfikacji tego kodu.

1.1. **Dodanie generowania wyjątku** w przypadku niepoprawnej wartości pierwszego elementu tablicy ***dane*** jako parametru metody ***WykonajProdukt*** klasy ***Fabryka*** - zmiana definicji tej klasy podanej w instrukcji laboratoryjnej 6. **Pełna walidacja poprawności formatu i wartości danych wejściowych powinna być realizowana przez *Warstwę klienta* aplikacji!**

```
public class Fabryka {
  public ProduktBezPodatku wykonajProdukt(String dane[]) {
      ProduktBezPodatku produkt = null;
      Promocja promocja;
      switch (Integer.parseInt(dane[0])) {
        case 0:
          produkt = new ProduktBezPodatku(dane[1], Float.parseFloat(dane[2]));
          break;
        case 1:
          promocja = new Promocja(Float.parseFloat(dane[3]));
          produkt = new ProduktBezPodatku(dane[1], Float.parseFloat(dane[2]), promocja);
          break;
        case 2:
          produkt = new ProduktZPodatkiem(dane[1], Float.parseFloat(dane[2]), Float.parseFloat(dane[3]));
          break;
        case 3:
          promocja = new Promocja(Float.parseFloat(dane[4]));
          produkt = new ProduktZPodatkiem(dane[1], Float.parseFloat(dane[2]), Float.parseFloat(dane[3]),
                                          promocja);
          break;
          default:
          throw new IllegalFormatCodePointException(0); //generowanie wyjątku z powodu niepoprawnej
                                                        //wartości elementu tablicy dane o indeksie 0.
      }
      return produkt; }
}
```

W klasie Aplikacja dodano do definicji metod, wywołujących metodę klasy ***Fabryka*** dodano klauzulę **throws IllegalFormatCodePointException -** zmiana definicji podanej w instrukcjach 6 i 7:

```
public void dodajProdukt (String dane[]) throws IllegalFormatCodePointException  //instrukcja 6

public void wstawZakup (int nr, int ile, String dane[]) throws IllegalFormatCodePointException //instrukcja 7

public static void main(String args[]) throws IllegalFormatCodePointException       // instrukcje 6 i 7
```

1.2. **Dodatkowo, klasy pakietu *rachunki* i *rachunki.model*, podane w części *Dodatek* 1 instrukcji laboratoryjnej 5 powinny być klasami publicznymi**:

```
public class Rachunek
public class Zakup
public class ProduktBezPodatku
public class ProduktZPodatkiem
public class Promocja
```

2. Testy jednostkowe z wykorzystaniem narzędzia *JUnit 4.13.2*

2.1. Definicja danych wzorcowych - należy wstawić pomocniczą klasę ***Dane*** z danymi wzorcowymi do testowania metod klas zdefiniowanych w **Dodatku 1** w p 2.2-2.7 oraz w katalogu ***Test Package*** projektu w pakiecie ***rachunki*** (pakiet klas testujących dedykowanych klasom testowanym w projekcie) należy dodać następujące puste interfejsy (***Java Interface***): ***TestControl, TestEntity*** oraz ***TestKoszt*** w celu obsługi mechanizmu nadawania kategorii poszczególnym klasom testującym oraz wybranym metodom testującym za pomocą adnotacji ***Category***.

Opis danych wzorcowych do testowania tworzenia i zawartości obiektów typu *Zakup* i z rodziny *ProduktBezPodatku*

2.1.1. String **daneproduktow**[][] – dwuwymiarowa tablica zawierająca w pierwszych ośmiu wierszach dane do utworzenia ośmiu obiektów z rodziny ***ProduktBezPodatku*** (**Dodatek 1**, Instrukcja 7, klasy: ***Fabryka, ProduktBezPodatku, ProduktZPodatkiem, Promocja***). Wiersz dziewiąty zawiera dane niepoprawne, powodujące generowanie wyjątku ***IllegalFormatCodePointException*** przez metodę klasy ***Fabryka***.

2.1.2. **ProduktBezPodatku produkty**[] – jednowymiarowa tablica ośmiu obiektów wzorcowych z rodziny ***ProduktBezPodatku***, zdefiniowanych na podstawie danych z tabeli ***dane_produktow*** z p. 2.1.1

2.1.3. **float cenyprodukty**[] – jednowymiarowa tablica ośmiu wartości ceny jednostkowej każdego z produktów podanych w tabeli ***produkty*** z p.2.1.2, wynikającej z promocji i podatku oraz ceny netto produktu.

2.1.4. **Zakup zakupy**[] – tablica ośmiu obiektów typu ***Zakup***, zdefiniowanych z wykorzystaniem obiektów z rodziny ***ProduktBezPodatku***, zdefiniowanych w tablicy ***produkty*** z p. 2.1.2.

2.1.5. **float cenyzakupow**[] – jednowymiarowa tablica zawierająca osiem wartości kosztów zakupów ośmiu obiektów typu ***Zakup***, zdefiniowanych w tablicy ***zakupy*** z p.2.1.4.

2.1.6. **int podatkizakupow**[] - jednowymiarowa tablica zawierająca osiem wartości podatków ośmiu obiektów typu ***Zakup***, zdefiniowanych w tablicy ***zakupy*** z p.2.1.4.

Opis danych wzorcowych do testowania tworzenia i zawartości dwóch rachunków

2.1.7. **Rachunek rachunki**[] – jednowymiarowa tablica zawierająca dwa obiekty typu ***Rachunek*** z pustymi kolekcjami obiektów typu ***Zakup***

2.1.8. String **daneproduktowrachunki**[][][] –tablica zawierająca dane wejściowe produktów (dane jako elementy tablicy p.2.1.1) należących do zakupów dwóch rachunków, gdy każdy z nich zawiera po pięć zakupów.

2.1.9. **Zakup zakupyrachunki[]**[][] –dwuwymiarowa tablica obiektów typu ***Zakup.*** W testach stanowi ona zbiory wzorcowych obiektów typu ***Zakup,*** należących do dwóch obiektów typu ***Rachunek.*** Obiekty typu ***Zakup*** zawierają obiekty z rodziny ***ProduktBezPodatku,*** zdefiniowane w tablicy z p. 2.1.2.

2.1.10. **int ileproduktowrachunki**[][] – dwuwymiarowa tablica zawierająca w każdym z dwóch wierszy pięć danych o liczbie produktów w każdym z pięciu zakupów, gdy każdy z dwóch wierszy reprezentuje dane jednego z dwóch rachunków.

2.1.11. **int kategorie**[] – jednowymiarowa tablica zawierająca wartości różnych kategorii wyznaczania ceny rachunku, gdzie wartość -1 oznacza wyznaczenie ceny zakupu produktów bez podatku, wartości: 3, 7, 14, 22 oznaczają kategorie cen rachunku wynikające z wysokości podatku produktów oraz -2 oznacza, że należy podać całkowity koszt rachunku – uwzględniając produkty bez podatku oraz wszystkie z podatkami.

2.1.12. **float kategoriewartoscirachunki**[][] – dwuwymiarowa tablica wartości sześciu wartości dwóch rachunków, wynikających z kategorii cen podanych w tabeli z p.2.1.11.

```
package rachunki;
import rachunki.model.ProduktBezPodatku;
import rachunki.model.ProduktZPodatkiem;
import rachunki.model.Promocja;
import rachunki.model.Rachunek;
import rachunki.model.Zakup;
public class Dane {
    //dane wzorcowe do testowania obiektów typu Zakup i z rodziny ProduktBezPodatku
public String daneproduktow[][] = new String[][]{
    {"0", "1", "1", "0", "0"}, {"0", "2", "2", "0", "0"}, {"2", "3", "3", "14", "0"}, {"2", "4", "4", "22", "0"},
    {"1", "5", "1", "30", "0"}, {"1", "6", "2", "50", "0"}, {"3", "7", " 5.47", "3", "30"}, {"3", "8", " 12.4", "7", "50"},
    {"4", "1", "1", "0", "0"} }; // 9-y el. zawiera dane do testowania generowania wyjątku przez klasę Fabryka
```

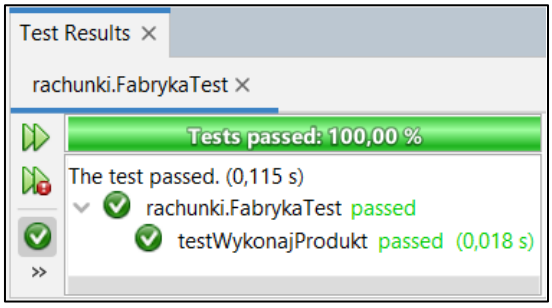
```
public static ProduktBezPodatku produkty[] = {new ProduktBezPodatku("1", 1),
   new ProduktBezPodatku("2", 2), new ProduktZPodatkiem("3", 3, 14), new ProduktZPodatkiem("4", 4, 22),
   new ProduktBezPodatku("5", 1, new Promocja(30)), new ProduktBezPodatku("6", 2, new Promocja(50)),
   new ProduktZPodatkiem("7", 5.47F, 3, new Promocja(30)), new ProduktZPodatkiem("8", 12.4F, 7,
                                                                    new Promocja(50))  };
  public float cenyprodukty[] = { 1F, 2F, 3.42F, 4.88F, 0.7F, 0.9F, 3.9930997F, 6.4479995F};
  public Zakup zakupy[] = {
        new Zakup(1, produkty[0]), new Zakup(4, produkty[1]),
        new Zakup(1, produkty[2]), new Zakup(1, produkty[3]),
        new Zakup(1, produkty[4]), new Zakup(1, produkty[5]),
        new Zakup(3, produkty[6]), new Zakup(1, produkty[7])
  };

   public float cenyzakupow[] = {1F, 8F, 3.42F, 4.88F, 0.7F, 0.9F, 11.9793F, 6.4479995F };
    public int podatkizakupow[] = {-1, -1, 14, 22, -1, -1, 3, 7  };

//dane zdefiniowane powyżej są zastosowane w definicji danych dwóch rachunków

 public Rachunek rachunki[] = {
        new Rachunek(1), new Rachunek(2)
    };

  public String daneproduktowrachunki[][][] = new String[][][] {
   {
    daneproduktow[0], daneproduktow[1], daneproduktow[2], //dane rachunku 1
    daneproduktow[3], daneproduktow[4]
   },
   {
     daneproduktow[5], daneproduktow[6], daneproduktow[7], //dane rachunku 2
     daneproduktow[1], daneproduktow[3]
    }
};

public Zakup zakupyrachunki[][] = {
   {
      new Zakup(2, produkty[0]), new Zakup(2, produkty[1]), //obiekty typu Zakup rachunku 1
      new Zakup(1, produkty[2]), new Zakup(4, produkty[3]),
      new Zakup(1, produkty[4])
   },
   {
      new Zakup(2, produkty[5]), new Zakup(3, produkty[6]), //obiekty typu Zakup rachunku 2
      new Zakup(2, produkty[7]),
      new Zakup(4, produkty[1]), new Zakup(1, produkty[3])
  }
};
public int ileproduktowrachunki[][] = {
        {1, 2, 1, 4, 1},  //początkowa ilość produktów w kolejnych pięciu zakupach rachunku 1
        {1, 3, 2, 4, 1}  //początkowa ilość produktów w kolejnych pięciu zakupach rachunku 2
   };
 public  int kategorie[] = { -1, 3, 7, 14, 22, -2 };                    //kategorie wartości rachunków

  public float  kategoriewartoscirachunki[][] = {
        { 6.7F,  0F, 0F, 3.42F, 19.52F, 29.640001F},                  //wartości rachunku 1 wg kategorii
        { 9.8F, 11.9793F, 12.895999F, 0.0F, 4.88F, 39.5553F}}//wartości rachunku 2 wg kategorii
    };
}
```

2.2. Test jednostkowy klasy ***Fabryka*** **(wynik działania: p.2.7.1, 2.7.3, 2.7.4)** – przykłady prostego testu (k1.2) porównującego osiem wyników działania metody ***wykonajProdukt()*** tworzącej cztery różne typy obiektów z rodziny ***ProduktBezPodatku*** z wzorcowymi wynikami z tabeli ***produkty*** za pomocą metody ***assertEquals*** klasy ***Assert*** oraz reakcję na niepoprawną wartość pierwszego elementu tablicy reprezentującej dane wejściowe testowanej metody ***wykonajProdukt*** (k1.2).

Zastosowanie adnotacji
@Test
@Before
@Category
@Rule

Zastosowanie metod
static public void assertEquals(Object expected, Object actual) //k1.1
ExpectedException //k1.2

```
package rachunki;
import java.util.IllegalFormatCodePointException;
import org.junit.Test;
import static org.junit.Assert.*;
import org.junit.Before;
import org.junit.Rule;
import org.junit.experimental.categories.Category;
import org.junit.rules.ExpectedException;
import rachunki.model.ProduktBezPodatku;

@Category({TestControl.class, TestEntity.class})  //określenie kategorii testu, zastosowanie -  p.2.7.1, 2.7.3
public class FabrykaTest {
   Dane dane;
 @Rule
 public ExpectedException exception = ExpectedException.none(); //definicja obiektu odpowiedzialnego
                  //za zachowanie metody testującej podczas generowania wyjątku przez testowaną metodę

@Before
   public void setUp(){
      dane= new Dane();
   }

@Test
   public void testWykonajProdukt() {
      System.out.println("wykonajProdukt");
      Fabryka instance = new Fabryka();
      for (int i = 0; i < 8; i++) {
         ProduktBezPodatku result = instance.wykonajProdukt(dane.daneproduktow[i]);
         assertEquals(dane.produkty[i], result);    //k1.1 – test poprawności tworzonych produktów
      }
      exception.expect(IllegalFormatCodePointException.class);   //k1.2 – definicja zachowania metody
      exception.expectMessage("Code point = 0x0");              //testowej podczas testowania generowania
      instance.wykonajProdukt(dane.daneproduktow[8]);           // wyjątku IllegalFormatCodePointException
                                                                // przez metodę wykonajProdukt
   }
 }
```

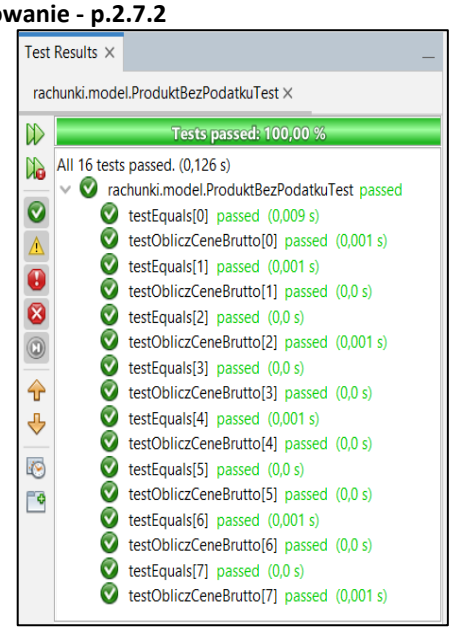


2.3. Testy jednostkowe klas *ProduktBezPodatku* i *ProduktZPodatkiem* (wynik działania: p.2.7.2, 2.7.4) – zastosowanie adnotacji **@Parameter** dla atrybutu ***numer1*** i wykonanie metody ***data()*** z adnotacją **@Parameters** powoduje wywołanie dwóch metod testowych osiem razy, podstawiając w kolejnej iteracji wartość elementu z kolejnego wiersza z ośmiu jednoelementowych wierszy tablicy ***data1*** do parametru ***numer1***. W rezultacie w metodzie testowej ***testObliczCeneBrutto()*** sprawdza się wyniki zwracane przez metodę ***obliczCeneBrutto*** *(***k2***)* dla ośmiu obiektów z rodziny ***ProduktBezPodatku***, porównując je z wynikami wzorcowymi. Metoda testowa ***testEquals()*** umożliwia weryfikację działania metody ***equals*** na każdej parze obiektów z tabeli *produkty (***k1.1** i **k1.2***)*.

Zastosowanie adnotacji
@Test
@Parameter, @Parameters
@Category
@ RunWith

Zastosowanie metod
static public void assertTrue(boolean condition), //k1.1
static public void assertFalse(boolean condition), //k1.2
static public void assertEquals(float expected, float actual, float delta)//k2

```
package rachunki.model;

import java.util.Arrays;
import java.util.Collection;
import org.junit.Test;
import static org.junit.Assert.*;
import org.junit.experimental.categories.Category;
import org.junit.runner.RunWith;
import org.junit.runners.Parameterized;
import rachunki.Dane;
import rachunki.TestEntity;

@Category({ TestEntity.class}) //określenie kategorii testu, zastosowanie - p.2.7.2
@RunWith(Parameterized.class)
public class ProduktBezPodatkuTest {

Dane dane=new Dane();
   @Parameterized.Parameter
   public int numer1;

  @Parameterized.Parameters
   public static Collection<Object[]> data() {
      Object[][] data1 =
            new Object[][]{ {0}, {1}, {2}, {3}, {4}, {5}, {6}, {7} };
      return Arrays.asList(data1);
   }

  @Test
   public void testEquals() {
      System.out.println("equals");
       for(int j=numer1;j<7;j++)
        if(numer1==j)
          assertTrue(dane.produkty[numer1].equals(dane.produkty[j])); //k1.1 –test porównania
        else                                                          // równych produktów
          assertFalse(dane.produkty[numer1].equals(dane.produkty[j]); //k1.2–test porównania
          }                                                           //różnych produktów

   @Test
   public void testObliczCeneBrutto() {
      System.out.println("obliczCeneBrutto");
      float result1 = dane.produkty[numer1].obliczCeneBrutto();
      float result2 = dane.cenyprodukty[numer1];
      assertEquals(result1, result2, 0F);  //k2 – test wyznaczania poprawnej wartości cen brutto produktów
   }
}
```

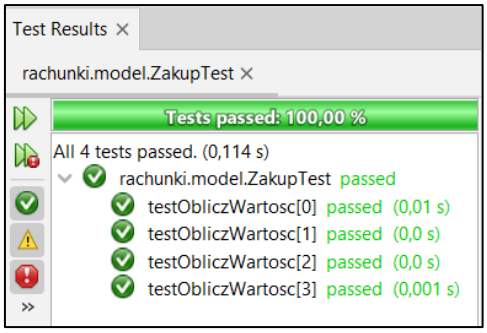


2.4. Testy jednostkowe klasy *Zakup* (wynik działania: p.2.7.2, 2.7.4) – zastosowanie adnotacji **@Parameter** dla atrybutów ***numer1*** i ***numer2*** i wykonanie metody ***data()*** z adnotacją **@Parameters**, która powoduje wywołanie jednej metody testowej cztery razy, podstawiając w kolejnej iteracji wartość elementu z kolejnego z czterech 2-elementowych wierszy tablicy ***data1*** do parametrów: ***numer1*** *i* ***numer***. W rezultacie w metodzie testowej ***testObliczWartosc()*** sprawdza się wyniki zwracane przez metodę ***obliczWartosc*** dla ośmiu obiektów z rodziny ***Zakup***, porównując je z wynikami wzorcowymi.

Zastosowanie adnotacji
@Test
@Parameter, @Parameters
@Category

Zastosowanie metod
static public void assertEquals(float expected, float actual, float delta), //k1.1, k1.2

```
package rachunki.model;

import java.util.Arrays;
import java.util.Collection;
import org.junit.Test;
import static org.junit.Assert.*;
import org.junit.experimental.categories.Category;
import org.junit.runner.RunWith;
import org.junit.runners.Parameterized;
import org.junit.runners.Parameterized.Parameter;
import rachunki.Dane;
import rachunki.TestEntity;

@Category({ TestEntity.class})          //określenie kategorii testu, zastosowanie -  p.2.7.2
@RunWith(Parameterized.class)
public class ZakupTest {

   Dane dane = new Dane();

  @Parameter(value = 0)
   public int numer1;
  @Parameter(value = 1)
   public int numer2;

  @Parameterized.Parameters
   public static Collection<Object[]> data() {
      Object[][] data1 = new Object[][]{
        {0, 1}, {2, 3}, {4, 5}, {6, 7}  };
      return Arrays.asList(data1);
   }

@Test
 public void testObliczWartosc() {
      System.out.println("obliczWartosc");
      assertEquals(dane.cenyzakupow[numer1], //k1.1 test wyznaczania wartości zakupów: 1-y zbiór danych
                   dane.zakupy[numer1].obliczWartosc(dane.podatkizakupow[numer1]), 0.0F);
      assertEquals(dane.cenyzakupow[numer2],
                   dane.zakupy[numer2].obliczWartosc(dane.podatkizakupow[numer2]), 0.0F);//k1.2
                                                  //test wyznaczania wartości zakupów: 2-i zbiór danych
   }
}
```



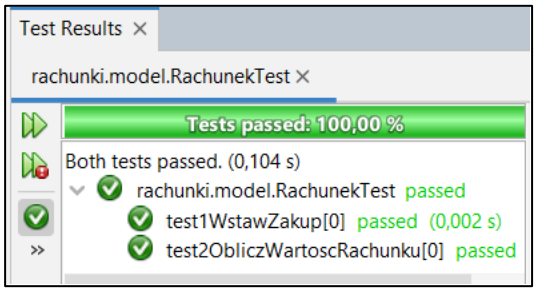
2.5. Testy jednostkowe klasy Rachunek (wynik działania: p.2.7.2, 2.7.4) - zastosowanie adnotacji **@Parameter** dla atrybutu ***zakupy1*** i wykonanie metody ***data()*** z adnotacją **@Parameters** powoduje wywołanie dwóch metod testowych tylko raz, podstawiając w jedynej iteracji dwuwymiarową tablicę obiektów typu ***Zakup*** do parametru ***zakupy1***. Tablica ta zawiera dwa 5-elementowe wiersze – pierwszy wiersz zawiera elementy należące do pierwszego rachunku, a drugi wiersz do drugiego rachunku. Za pomocą adnotacji **@BeforeClass** wykonano w metodzie ***SetUp*** jednorazowo (przed wykonaniem wszystkich dwóch testów) tablicę obiektów typu ***Rachunek***, które w metodzie testowej ***test1WstawZakup*** są zapełnione obiektami z tablicy ***zakupy1***. Dzięki narzuceniu kolejności wykonania metod testowych za pomocą adnotacji **@FixMethodOrder(MethodSorters.NAME_ASCENDING)** ustalono alfabetyczną kolejność wykonania metod testowych: ***test1WstawZakup***, ***test2ObliczWartoscRachunku***. Kolejna metoda testowa korzysta z danych utworzonych w poprzedniej metodzie testowej. Metoda testowa ***test2ObliczWartoscRachunku()*** operuje na rachunkach wypełnionych obiektami typu ***Zakup*** w metodzie ***test1WstawZakup*.**

Zastosowanie adnotacji
@Test
@BeforeClass
@Parameter, @Parameters
@FixMethodOrder(MethodSorters.NAME_ASCENDING)
@Category

Zastosowanie metod
static public void assertSame(Object expected, Object actual) //k1.1
static public void assertEquals(long expected, long actual) //k1.2, k1.3
static public void assertEquals(float expected, float actual, float delta) //k2

```
package rachunki.model;
import java.util.Arrays;
import java.util.Collection;
import org.junit.Test;
import static org.junit.Assert.*;
import org.junit.BeforeClass;
import org.junit.FixMethodOrder;
import org.junit.experimental.categories.Category;
import org.junit.runner.RunWith;
import org.junit.runners.MethodSorters;
import org.junit.runners.Parameterized;
import rachunki.Dane;
import rachunki.TestEntity;

@Category({TestEntity.class})              //określenie kategorii testu, zastosowanie - p.2.7.2
@FixMethodOrder(MethodSorters.NAME_ASCENDING)
@RunWith(Parameterized.class)
public class RachunekTest {
   static Dane dane;
   static Rachunek instances[];
   @Parameterized.Parameter
   public Zakup[][] zakupy1;
   @Parameterized.Parameters
   public static Collection<Object[][]> data() {
    Object[][][][] data1 = new Zakup[][][][] { {
    {
     { new Zakup(1,Dane.produkty[0]), new Zakup(2, Dane.produkty[1]),
       new Zakup(1, Dane.produkty[2]), new Zakup(4, Dane.produkty[3]),  new Zakup(1, Dane.produkty[4]) },
     { new Zakup(1, Dane.produkty[5]), new Zakup(3, Dane.produkty[6]),
       new Zakup(2, Dane.produkty[7]), new Zakup(4, Dane.produkty[1]),  new Zakup(1, Dane.produkty[3]) }
    }   } };
    return Arrays.asList(data1);  }
```

```
@BeforeClass
 public static void SetUp() {
     instances=new Rachunek[2];
     instances[0] = new Rachunek(1);
     instances[1] = new Rachunek(2);
     dane = new Dane();
  }

 @Test
 public void test1WstawZakup() {
    System.out.println("wstawZakup");
    for (int i = 0; i < 2; i++) {
       for (int j = 0; j < 5; j++) {
          instances[i].wstawZakup(zakupy1[i][j]);
         Zakup zakup1 = instances[i].getZakupy().get(j);
          assertSame(zakup1, zakupy1[i][j]); //k1.1 – test sprawdzenia równości referencyjnej danych
       }
       int rozmiar1 = instances[i].getZakupy().size();
       int ile = instances[i].getZakupy().get(0).getIlosc();
       instances[i].wstawZakup(zakupy1[i][0]);
       assertEquals(rozmiar1, instances[i].getZakupy().size());//k1.2- test spójności danych podczas
                                                               // dodawania podobnych zakupów
       assertEquals(instances[i].getZakupy().get(0).getIlosc(), ile * 2); //k1.3 test algorytmu dodawania
                                                                          //podobnych zakupów
    }
 }
 @Test
 public void test2ObliczWartoscRachunku() {
    System.out.println("obliczWartoscRachunku");
     for (int i = 0; i < 2; i++)
        for (int j = 0; j < 5; j++)
          assertEquals(dane. kategoriewartoscirachunki[i][j],
                  instances[i].obliczWartoscRachunku(dane.kategorie[j]), 0F);//k2 – test obliczania wartości
                                                                             //rachunku w różnych kategoriach
 }
}
```



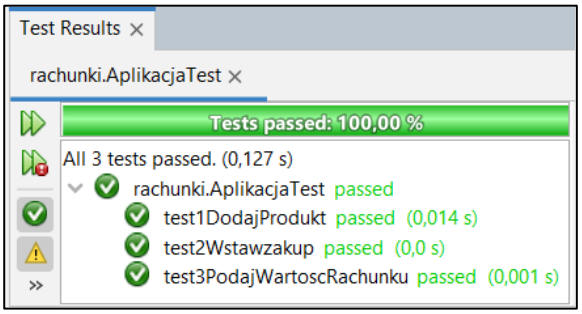
2.6. Testy jednostkowe klasy *Aplikacja* (wynik działania: p.2.7.1, 2.7.4) opierają się na wywołaniu trzech metod testowych, działających w kolejności alfabetycznej dzięki zastosowaniu adnotacji **@FixMethodOrder(MethodSorters.NAME_ASCENDING)**: ***test1DodajProdukt, test2WstawZakup, test3PodajWartoscRachunku***. Przed wywołaniem metod testowych wywołana jest metoda ***SetUp*** dzięki zastosowaniu adnotacji **@BeforeClass** – metoda ta tworzy obiekt typu ***Aplikacja***. Metody działające w podanym porządku umożliwiają po dodaniu produktów w metodzie ***test1DodajProdukt*** wykonać testy: dodawania zakupów w metodzie ***test2WstawZakup***, obliczania wartości rachunków w różnych kategoriach w metodzie ***test3PodajWartoscRachunku***. Metody testowe ***test1DodajProdukt*** oraz ***test2WstawZakup*** testują również przypadek podania niepoprawnej wartości w danych wejściowych, które powodują generowanie wyjątku przez metodę ***wykonajProdukt*** klasy ***Fabryka*** – wyjątek ten jest obsługiwany w podanych metodach testowych za pomocą mechanizmu **@Rule.** Za pomocą adnotacji **@Category** dokonano różnych klasyfikacji wszystkich metod testowych i wybranej metody ***test3PodajWartoscRachunku***.

Zastosowanie adnotacji
@Test
@BeforeClass
@FixMethodOrder(MethodSorters.NAME_ASCENDING)
@Category
@Rule

Zastosowanie metod
static public void assertEquals(Object expected, Object actual)// k1.1, k2.1
static public void assertEquals(long expected, long actual) //k1.2, k2.2, k2.3
static public void assertEquals(float expected, float actual, float delta) //k3

```
package rachunki;

import java.util.Arrays;
import java.util.IllegalFormatCodePointException;
import org.junit.Test;
import static org.junit.Assert.*;
import org.junit.BeforeClass;
import org.junit.FixMethodOrder;
import org.junit.Rule;
import org.junit.experimental.categories.Category;
import org.junit.rules.ExpectedException;
import org.junit.runners.MethodSorters;
import rachunki.model.ProduktBezPodatku;
import rachunki.model.Zakup;

@Category({TestControl.class, TestEntity.class})  //określenie kategorii testu, zastosowanie -  p.2.7.1, 2.7.3
@FixMethodOrder(MethodSorters.NAME_ASCENDING)
public class AplikacjaTest {

   static Dane dane;

   static Aplikacja instance;

  @Rule
   public ExpectedException exception = ExpectedException.none();

 @BeforeClass
  static public void SetUp() {
      instance = new Aplikacja();
      dane = new Dane();
   }
```

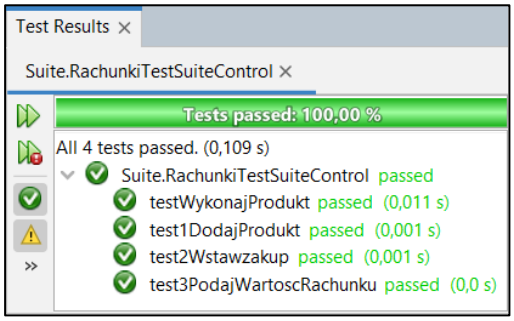
```
@Test
public void test1DodajProdukt() {
    System.out.println("dodajProdukt");
 int indeksyproduktow[] = {0, 1, 2, 3, 4, 5, 6, 7, 7, 7};
  for (int i = 0; i < 2; i++)
   for (int j = 0; j < 5; j++) {
    instance.dodajProdukt(dane.daneproduktowrachunki[i][j]);
    int ile1 = instance.getProdukty().size();
    instance.dodajProdukt(dane.daneproduktowrachunki[i][j]);//powtórzenia wartości elementów
                                                            // dla i=1 oraz j=3, j=4
    int ile2 = instance.getProdukty().size();
    ProduktBezPodatku result = instance.getProdukty().get(ile2 - 1);
    assertEquals(dane.produkty[indeksyproduktow[i * 5 + j]], result); //k1.1 test dodawania produktów
    assertEquals(ile1, ile2);                  //k1.2 – test spójności danych podczas dodawanie produktów
   }
  exception.expect(IllegalFormatCodePointException.class); //obsługa wyjątku w testowanej metodzie
  exception.expectMessage("Code point = 0x0");
  instance.dodajProdukt(dane.daneproduktow[8]);
}

@Test
public void test2WstawZakup() {
    System.out.println("wstawZakup");
    instance.setRachunki(Arrays.asList(dane.rachunki));
    for (int i = 0; i < 2; i++) {
      for (int j = 0; j < 5; j++) {
        instance.wstawZakup(i + 1, dane.ileproduktowrachunki[i][j],
                                   dane.daneproduktowrachunki[i][j]);
        Zakup zakup1 = instance.getRachunki().get(i).geZakupy().get(j);
        assertEquals(zakup1, dane.zakupyrachunki[i][j]);     //k2.1 test dodawania zakupów
      }
      int rozmiar = instance.getRachunki().get(i).getZakupy().size();
      instance.wstawZakup(i + 1, dane.ileproduktowrachunki[i][0],
                                 dane.daneproduktowrachunki[i][0]);
      assertEquals(instance.getRachunki().get(i).getZakupy().size(), rozmiar); //k2.2 test spójności danych
                                                                               //podczas dodawania zakupów
      assertEquals(instance.getRachunki().get(i).getZakupy().get(0).getIlosc(),
                   dane.zakupyrachunki[i][0].getIlosc());   //k2.3 test algorytmu dodawania
    }                                                       // podobnych zakupów
   exception.expect(IllegalFormatCodePointException.class);
   exception.expectMessage("Code point = 0x0");
    instance.wstawZakup(1, 1, dane.daneproduktow[8]); //obsługa wyjątku w testowanej metodzie
  }

@Test
@Category(TestKoszt.class)              //określenie kategorii testu – przykład zastosowania w p.2.7.4
  public void test3PodajWartoscRachunku() {
     System.out.println("podajWartoscRachunku");
     for (int i = 0; i < 2; i++)
       for (int j = 0; j < 6; j++)
         assertEquals(dane.kategoriewartoscirachunki [i][j],          //k3 – test obliczania wartości
           instance.podajWartoscRachunku(i + 1, dane.kategorie[j]), 0F); //rachunku w różnych kategoriach
  }
}
```



2.7. Tworzenie zestawów testów

2.7.1. Wyniki testów wykonanych przez klasy należące również do kategorii @Category(TestControl.class): *FabrykaTest, AplikacjaTest*

```
package Suite;

import org.junit.experimental.categories.Categories;
import org.junit.runner.RunWith;
import rachunki.AplikacjaTest;
import rachunki.FabrykaTest;
import rachunki.model.ProduktBezPodatkuTest;
import rachunki.model.RachunekTest;
import rachunki.model.ZakupTest;
import rachunki.TestControl;

@Categories.SuiteClasses({FabrykaTest.class,AplikacjaTest.class, ProduktBezPodatkuTest.class,
ZakupTest.class, RachunekTest.class })
@RunWith(Categories.class)
@Categories.IncludeCategory(TestControl.class)
public class RachunkiTestSuiteControl {  }
```

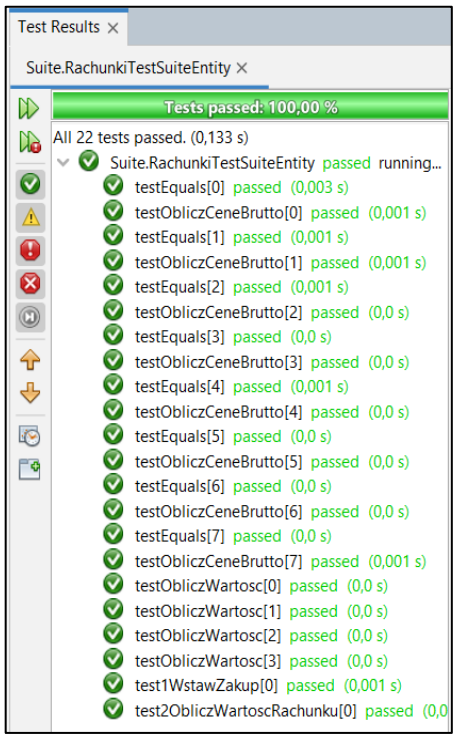


wykonajProdukt
dodajProdukt
wstawZakup
podajWartoscRachunku

Wynik testu: FabrykaTest, AplikacjaTest

2.7.2. Wyniki testów wykonanych przez klasy należące tylko do kategorii @Category(TestEntity.class): *ProduktBezPodatkuTest, ZakupTest, RachunekTest*

```
package Suite;

import org.junit.experimental.categories.Categories;
import org.junit.runner.RunWith;
import rachunki.AplikacjaTest;
import rachunki.FabrykaTest;
import rachunki.model.ProduktBezPodatkuTest;
import rachunki.model.RachunekTest;
import rachunki.model.ZakupTest;
import rachunki.TestControl;

@Categories.SuiteClasses({FabrykaTest.class, AplikacjaTest.class,
 ProduktBezPodatkuTest.class, ZakupTest.class, RachunekTest.class,})
@RunWith(Categories.class)
@Categories.ExcludeCategory(TestControl.class)
public class RachunkiTestSuiteEntity { }
```



equals
obliczCeneBrutto
equals
obliczCeneBrutto
equals
obliczCeneBrutto
equals
obliczCeneBrutto

equals
obliczCeneBrutto
equals
obliczCeneBrutto
equals
obliczCeneBrutto
equals
obliczCeneBrutto

Wyniki testu ProduktBezPodatkuTest

obliczWartosc
obliczWartosc
obliczWartosc
obliczWartosc

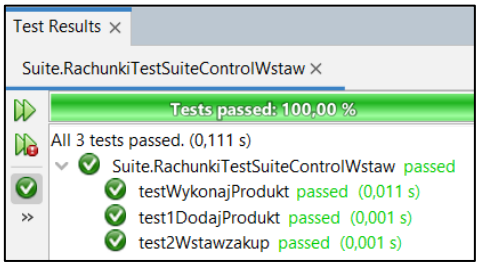
Wyniki testu ZakupTest

wstawZakup
obliczWartoscRachunku

Wyniki testu RachunekTest

2.7.3. Wyniki testów wykonanych przez klasy należące do kategorii @Category(TestControl.class) z wyłączeniem metody *test3PodajWartoscRachunku* klasy *AplikacjaTest* zaliczonej do kategorii @Categorii(Testkoszt.class) : *FabrykaTest, AplikacjaTest*

```
package Suite;

import org.junit.experimental.categories.Categories;
import org.junit.runner.RunWith;
import rachunki.TestControl;
import rachunki.TestKoszt;

@Categories.SuiteClasses({RachunkiTestSuiteControl.class})
@RunWith(Categories.class)
@Categories.IncludeCategory(TestControl.class)
@Categories.ExcludeCategory(TestKoszt.class)
public class RachunkiTestSuiteControlWstaw {}
```

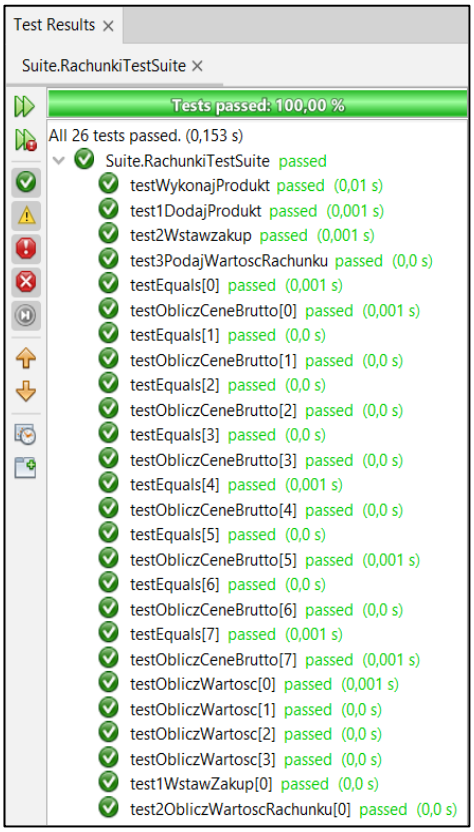


wykonajProdukt
dodajProdukt
wstawZakup

Wynik testu: *FabrykaTest, AplikacjaTest* z wyłączeniem metody testowej *test3PodajWartoscRachunku()*

2.7.4. Wyniki testów wykonanych przez wszystkie klasy testujące – niezależnie od przypisanych kategorii

```
package Suite;

import org.junit.runner.RunWith;
import org.junit.runners.Suite;
import org.junit.runners.Suite.SuiteClasses;
import rachunki.AplikacjaTest;
import rachunki.FabrykaTest;
import rachunki.model.ProduktBezPodatkuTest;
import rachunki.model.RachunekTest;
import rachunki.model.ZakupTest;

@SuiteClasses({FabrykaTest.class, AplikacjaTest.class,
ProduktBezPodatkuTest.class, ZakupTest.class, RachunekTest.class})
@RunWith(Suite.class)
public class RachunkiTestSuite { }
```



wykonajProdukt
dodajProdukt
wstawZakup
podajWartoscRachunku

Wynik testu:
FabrykaTest,AplikacjaTest

equals
obliczCeneBrutto
equals
obliczCeneBrutto
equals
obliczCeneBrutto
equals
obliczCeneBrutto

Wyniki testu
ProduktBezPodatkuTest

equals
obliczCeneBrutto
equals
obliczCeneBrutto
equals
obliczCeneBrutto
equals
obliczCeneBrutto

obliczWartosc
obliczWartosc
obliczWartosc
obliczWartosc

Wyniki testu
ZakupTest

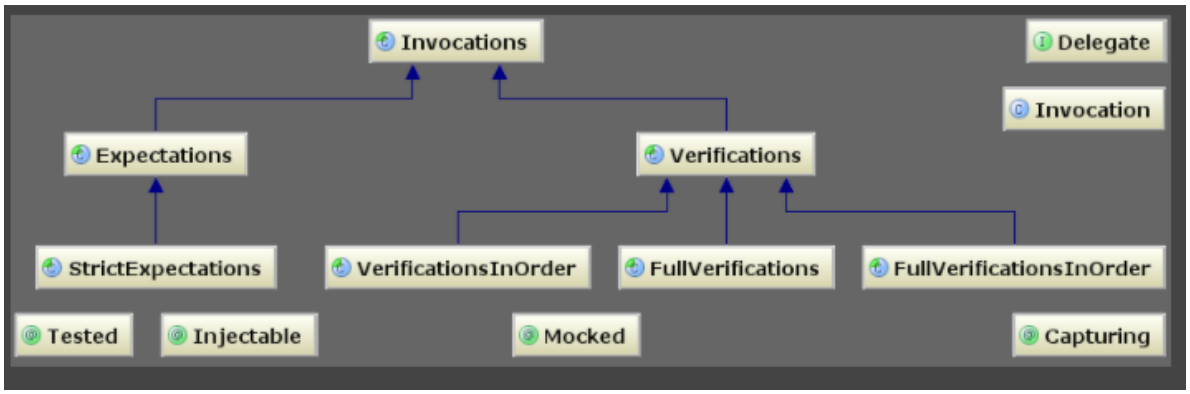
wstawZakup
obliczWartoscRachunku

Wyniki testu
RachunekTest

3. Przykłady testowania oparte na symulacji obiektów za pomocą obiektów typu *JMockit* – ważne informacje dotyczące tego narzędzia podano w Dodatku 2, p.2 i 4.

W kontekście testowania zachowania obiektów powiązanych z obiektami, których zachowanie symuluje się za pomocą obiektów typu ***JMockit*,** możemy wyróżnić następujące 3 alternatywne fazy testowania (rysunek poniżej):

- **Faza zapisu** (nagrywania), podczas którego nagrywane są wywołania metod symulowanego obiektu za pomocą obiektów z rodziny ***Expectations.*** Symulację przeprowadza się za pomocą adnotacji ***@Mocked, @Injectable(3.1)*** oraz ***@Capturing (3.2, 3.3).***
- **Faza odtwarzania**, podczas której odtwarzane są wywołania nagranych wywołań metod, używane przez powiązane obiekty. Często nie jest to odwzorowanie jeden do jednego między wywołaniami nagranymi i odtwarzanymi.
- **Faza sprawdzenia**, w trakcie której można zweryfikować nagrane wywołania, które zostały wykorzystane w fazie odtwarzania za pomocą obiektu z rodziny ***Verifications***.



Rysunek przedstawiony powyżej pochodzi ze z tutoriala JMockit 1.27: JMockit_Tutorial_Mocking.pdf.
Biblioteka ***JMockit*** zapewnia bogate wsparcie w realizacji zautomatyzowanych symulacyjnych testów deweloperskich. Gdy używana jest symulacja, badanie skupia się na testowaniu metod klasy powiązanej z symulowaną klasą za pomocą testów jednostkowych, które zawierają interakcje z symulowanym kodem obiektów powiązanych. Zazwyczaj testowany kod w jednym teście jednostkowym jest zależny od kodu jednej powiązanej klasy, jednak w przypadku powiązań z wieloma klasami należy w tym teście jednostkowym zastosować interakcje z symulowanym kodem ważniejszych klas z tego zbioru.

Nie należy jednak zbyt rygorystycznie opierać testowanie jednostkowe o symulację kodu każdego powiązanego obiektu. Można je zastąpić testami integracyjnymi. Jednak w przypadku testów integracyjnych czasem warto zastosować symulację w przypadku braku implementacji fragmentów kodu lub trudności użycia kodu (odwołania do baz danych, wysłanie e-mail itp.) podczas uruchamiania testów integracyjnych.

Interakcja pomiędzy dwiema klasami zawsze przybiera formę wywołania metody lub konstruktora. Celem symulacji, w zakresie jednego testu jednostkowego, jest wywołanie metody lub zestawu wywołań metod klasy zależnej wraz z wartościami parametrów i zwracanych wyników. Często ważna jest kolejność wywołań metod klasy zależnej podczas symulacji zestawu wywołań.

Symulację przeprowadza się za pomocą adnotacji ***@Mocked*** (przykłady: 3.1, 3.3, Dodatek 1; przykłady 1.5, 1.6, Dodatek 3)***, @Injectable*** (przykład 3.2, Dodatek 1; przykład 1.1, Dodatek 3) oraz ***@Capturing*** (przykłady 1.2, 1.3, Dodatek 3). Opisy tych adnotacji podano w podanych przykładach.
Symulacja wywołania metody może opierać się na specyfikacji jej algorytmu dzięki zastosowaniu obiektu typu ***Delegate*** (przykład 1.5, Dodatek 3). Testowany obiekt w klasie testującej może być wystąpić w roli atrybutu tej klasy za pomocą adnotacji ***@Tested*** (przykład 1.1, Dodatek 3)***.***

3.1. Testowanie klasy wybranych metod ***Zakup*** oparte na jednej jawnie deklarowanej instancji symulowanej klasy ***ProduktBezPodatku*** za pomocą adnotacji **@Mocked** w każdym teście i tworzeniu obiektu z rodziny ProduktBezPodatku za pomocą odpowiedniego konstruktora, który jest automatycznie symulowanym obiektem.

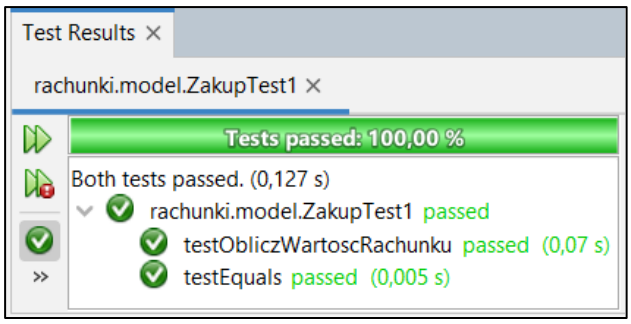
Zastosowanie adnotacji
@Test
@RunWith(JMockit.class)
@Mocked

Fazy testowania metody **equals** klasy **Zakup** w metodzie testowej **testEquals**, opartej na domyślnej symulacji metody **equals** klasy **ProduktBezPodatku**
1)Bez jawnie zdefiniowanej fazy nagrywania
2) Odtwarzanie metody **equlas** klasy **Zakup**
3) Faza weryfikacji **-new FullVerificationsInOrder(), maxTimes**

Fazy testowania metody **obliczWartosc** klasy **Zakup** w metodzie testowej **testObliczWartoscRachunku**(), opartej na symulacji metod **getPodatek** oraz **obliczCeneBrutto** klasy **ProduktBezPodatku**
1)Faza nagrywania**-new Expectations(), result**
2) Odtwarzanie metody **obliczWartosc** klasy **Zakup**
3)Faza weryfikacji - **new FullVerificationsInOrder(), maxTimes**

```
package rachunki.model;

import mockit.Expectations;
import mockit.FullVerificationsInOrder;
import mockit.Mocked;
import mockit.integration.junit4.JMockit;
import org.junit.Test;
import static org.junit.Assert.*;
import org.junit.runner.RunWith;

@RunWith(JMockit.class)
public class ZakupTest1 {

  @Mocked
  ProduktBezPodatku produkt;

  @Test
  public void testEquals() {
    ProduktBezPodatku produkt2 = new ProduktZPodatkiem("8", 4, 7, new Promocja(50));
                                                                  // dowolny konstruktor
    Zakup zakupy[] ={ new Zakup(2, produkt), new Zakup(2, produkt2)  };
     System.out.println("equals");
     for (int i = 0; i < 1; i++)
        for (int j = i; j < 2; j++)
           if (i == j)
              assertTrue(zakupy[i].equals(zakupy[i]));
           else
              assertFalse(zakupy[i].equals(zakupy[j]));

     new FullVerificationsInOrder() {
        {
           produkt.equals(any);          maxTimes = 2;   }
     };
}
```

```
@Test
   public void testObliczWartoscRachunku(@Mocked ProduktBezPodatku produkt1) {
      ProduktBezPodatku produkt2 = new ProduktBezPodatku("8", 4);// dowolny konstruktor
      Zakup zakupy[] =  { new Zakup(2, produkt1),  new Zakup(2, produkt2)};
      int podatki[] = {-1, 7};
      float ceny1[] = {0.9F,  6.48F};          //ceny brutto produktow
      float ceny2[] = {1.8F,  12.96F};         //ceny brutto zakupow

      System.out.println("obliczWartoscRachunku");
      new Expectations() {
        {
           produkt1.getPodatek();                result = podatki[0];
           produkt1.obliczCeneBrutto();          result = ceny1[0];
           produkt2.getPodatek();                result = podatki[1];
           produkt2.obliczCeneBrutto();          result = ceny1[1];
        }
      };
      for (int j = 0; j < 2; j++)
        assertEquals(zakupy[j].obliczWartosc(podatki[j]), ceny2[j], 0F);  //dodatkowy test assertEquals

      new FullVerificationsInOrder() {
        {
           produkt1.getPodatek();                maxTimes = 1;
           produkt1.obliczCeneBrutto();          maxTimes = 1;
           produkt2.getPodatek();                maxTimes = 1;
           produkt2.obliczCeneBrutto();          maxTimes = 1;
        }
      };
   }
}
```



3.2. Testowanie klasy ***Rachunek*** – testowanie za pomocą symulowania konkretnych instancji powiązanych klas (**@Injectable**)

Zastosowanie adnotacji
@Test
@RunWith(JMockit.class)
@Injectable

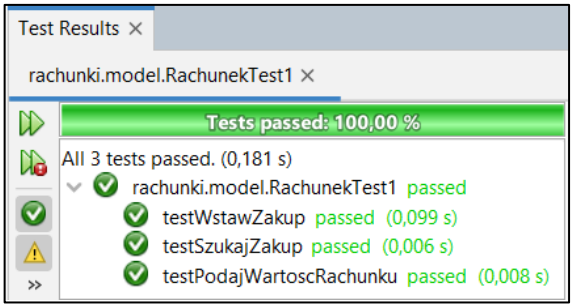
Fazy testowania metody **szukajZakup** w metodzie testowej **testSzukajZakup** klasy **Rachunek,** powiązanej z instancjami klasy **Zakup**, opartej na domyślnej symulacji metody **equals** klasy **Zakup.**
1)Domyślna faza nagrywania metody **equals** z klasy **Zakup**
2) Faza odtwarzania metody **szukajZakup** klasy **Rachunek**
3)Faza weryfikacji - **new FullVerificationsInOrder(), times**

Fazy testowania metody **wstawZakup** w metodzie testowej **testWstawZakup**() klasy **Rachunek,** powiązanej z instancjami klasy **Zakup**, opartej na symulacji metod **getIlosc** oraz **dodajIloscProduktu** klasy **Zakup**
1)Faza nagrywania – **new StrictExpectations(), returns**
2) Faza odtwarzania metody **wstawZakup**
3)Faza weryfikacji -new FullVerificationsInOrder(), maxTimes

Fazy testowania metody **obliczWartoscRachunku** w metodzie testowej **testPodajWartoscRachunku**() klasy **Rachunek**, powiązanej z instancjami klasy **Zakup**, opartej na symulacji metody **obliczWartosc** klasy **Zakup**
1)Faza nagrywania- **new Expectations(), result**
2)Faza odtwarzania metody **obliczWartoscRachunku**
3)Faza weryfikacji - **new VerificationsInOrder(), times**

```
package rachunki.model;

import java.util.Arrays;
import mockit.Expectations;
import mockit.FullVerificationsInOrder;
import mockit.Injectable;
import mockit.StrictExpectations;
import mockit.VerificationsInOrder;
import mockit.integration.junit4.JMockit;
import org.junit.Test;
import static org.junit.Assert.*;
import org.junit.runner.RunWith;

@RunWith(JMockit.class)
public class RachunekTest1 {

  @Injectable
  Zakup zakup1,zakup2, zakup3;

  @Test
  public void testSzukajZakup() {
    System.out.println("szukajZakup");
    Zakup zakupy[] = {zakup1, zakup2, zakup3};
    Rachunek rachunek = new Rachunek(1);
    rachunek.setZakupy(Arrays.asList(zakupy));
    for (int i = 0; i < 3; i++)
      assertEquals(rachunek.szukajZakup (zakupy[i]), zakupy[i]);     //dodatkowy test assertEquals

    new FullVerificationsInOrder() {
      {
        zakup1.equals(any);          times = 2;
        zakup2.equals(any);          times = 3;
        zakup3.equals(any);          times = 4;
      }
    };
  }
```

```
@Test
  public void testWstawZakup() {
    System.out.println("wstawZakup");
    Zakup zakupy[] = {zakup1, zakup2, zakup3, zakup1};
    Rachunek rachunek = new Rachunek(1);
    new StrictExpectations() {
      {
        zakup1.getIlosc();                    returns(1);
        zakup1.dodajIloscProduktu(1);         returns(2);
        zakup1.getIlosc();                    returns(2);
      }
    };
   for (int i = 0; i < 4; i++)
      rachunek.wstawZakup(zakupy[i]);
    assertEquals(rachunek.getZakupy().get(0).getIlosc(), 2);        //dodatkowy test assertEquals
    assertEquals(rachunek.getZakupy().size(), 3);                   //dodatkowy test assertEquals
    new FullVerificationsInOrder() {
      {
        zakup2.equals(any);          maxTimes = 1;
        zakup3.equals(any);          maxTimes = 2;
        zakup1.equals(any);          maxTimes = 1;
      }
    };
  }
  @Test
  public void testPodajWartoscRachunku() {
    Zakup zakupy[] = {zakup1, zakup2, zakup3};
    float wartoscirachunku[] = {9.8F, 0.0F, 0.0F, 0.0F, 4.88F, 14.68F};
    int podatki[] = {-1, 3, 7, 14, 22, -2};
    System.out.println("obliczWartoscRachunku");
    Rachunek rachunek = new Rachunek(1);
    new Expectations() {
      {
        zakupy[0].obliczWartosc(-1);          result = 1.8F;
        zakupy[1].obliczWartosc(-1);          result = 8F;
        zakupy[2].obliczWartosc(22);          result = 4.88F;
        zakupy[0].obliczWartosc(-2);          result = 1.8F;
        zakupy[1].obliczWartosc(-2);          result = 8.0F;
        zakupy[2].obliczWartosc(-2);          result = 4.88F;
      }
    };

    rachunek.setZakupy(Arrays.asList(zakupy));
    for (int i = 0; i < 6; i++)
      assertEquals(wartoscirachunku[i], rachunek.obliczWartoscRachunku(podatki[i]), 0F);

    new VerificationsInOrder() {
      {
        zakupy[0].obliczWartosc(-1);          times = 1;
        zakupy[1].obliczWartosc(-1);          times = 1;
        zakupy[2].obliczWartosc(22);          times = 1;
        zakupy[0].obliczWartosc(-2);          times = 1;
        zakupy[1].obliczWartosc(-2);          times = 1;
        zakupy[2].obliczWartosc(-2);          times = 1;
      }
    };
  }
}
```



3.3. Testowanie klasy ***Aplikacja*** – symulacja metody klasy ***Fabryka*** powiązanej z klasą ***Aplikacja***; testowanie metody ***dodajProdukt*** w zakresie poprawnych danych i niepoprawnych danych.

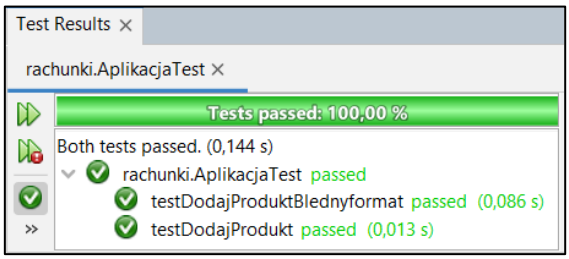
Zastosowanie adnotacji
@Test
@RunWith(JMockit.class)
@Mocked

Fazy testowania metody **dodajProdukt** w metodzie testowej **testDodajProdukt** klasy **Aplikacja**, powiązanej z instancjami klasy **Fabryka**, opartej na symulacji metody **wykonajProdukt** klasy **Fabryka**
1)Faza nagywania-**new Expectations(), result**
2)Faza odtwarzania – wykonanie metody **dodajProdukt** klasy **Aplikacja**
3)Brak jawnej fazy weryfikacji

Fazy testowania metody **dodajProdukt** w metodzie testowej **testDodajProduktBlednyformat** klasy **Aplikacja**, powiązanej z instancjami klasy **Fabryka**, opartej na symulacji metody **wykonajProdukt** klasy **Fabryka** generującej wyjątek typu **IllegalFormatCodePointException** po podaniu niepoprawnych danych.
1)Faza nagywania-**new Expectations(), withNotNull, result**
2)Faza odtwarzania -– wykonanie metody **dodajProdukt** klasy **Aplikacja** po podaniu niepoprawnych danych
3)Brak jawnej fazy werfikacji

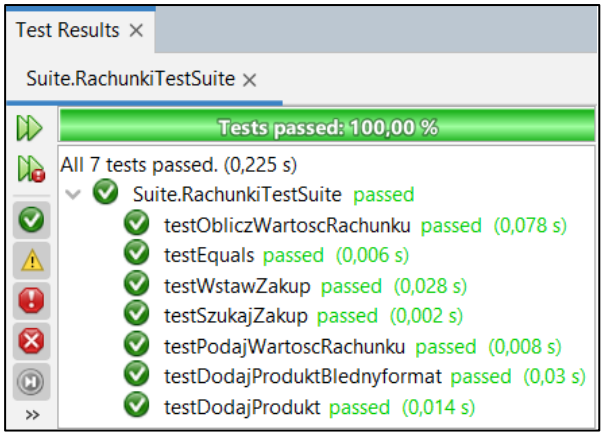
```
package rachunki;
import java.util.IllegalFormatCodePointException;
import mockit.Expectations;
import mockit.Mocked;
import mockit.integration.junit4.JMockit;
import org.junit.Test;
import static org.junit.Assert.*;
import org.junit.runner.RunWith;
import rachunki.model.ProduktBezPodatku;
import rachunki.model.ProduktZPodatkiem;
import rachunki.model.Promocja;
@RunWith(JMockit.class)
public class AplikacjaTest {
        ProduktBezPodatku produkty[] = {
        new ProduktBezPodatku("1", 1),
        new ProduktZPodatkiem("3", 3, 14),
        new ProduktBezPodatku("5", 1, new Promocja(30)),
        new ProduktZPodatkiem("7", 3, 3, new Promocja(30)),
        new ProduktZPodatkiem("7", 3, 3, new Promocja(30)) };
      String dane[][] = new String[][]{
        {"0", "1", "1", "", ""}, {"2", "3", "3", "14", ""}, {"1", "5", "1", "30", ""},
        {"3", "7", "3", "3", "30"},{"3", "7", "3", "3", "30"}, {"4","1","1","",""} };
@Mocked  Fabryka fabryka;
@Test
public void testDodajProdukt() {
      System.out.println("dodajProdukt");
      new Expectations() {
        {
           fabryka.wykonajProdukt(dane[0]);          result = produkty[0];
           fabryka.wykonajProdukt(dane[1]);          result = produkty[1];
           fabryka.wykonajProdukt(dane[2]);          result = produkty[2];
           fabryka.wykonajProdukt(dane[3]);          result = produkty[3];
        }
      };
      Aplikacja aplikacja = new Aplikacja();
      for (int i = 0; i < 5; i++) {
        aplikacja.dodajProdukt(dane[i]);
        if(i<4)
            assertEquals(produkty[i], aplikacja.getProdukty().get(i));
        else
             assertEquals(produkty[i], aplikacja.getProdukty().get(i-1));  }
}
```

```
@Test(expected=IllegalFormatCodePointException.class)
   public void testDodajProduktBlednyformat() {
      System.out.println("dodajProdukt_niepoprawny_format_danych");
      new Expectations() {
         {
            fabryka.wykonajProdukt((String[]) withNotNull());
            result=new IllegalFormatCodePointException(0);
         }
      };
      Aplikacja aplikacja = new Aplikacja();
       aplikacja.dodajProdukt(dane[5]);
   }
}
```



3.4. Tworzenie zestawów testów

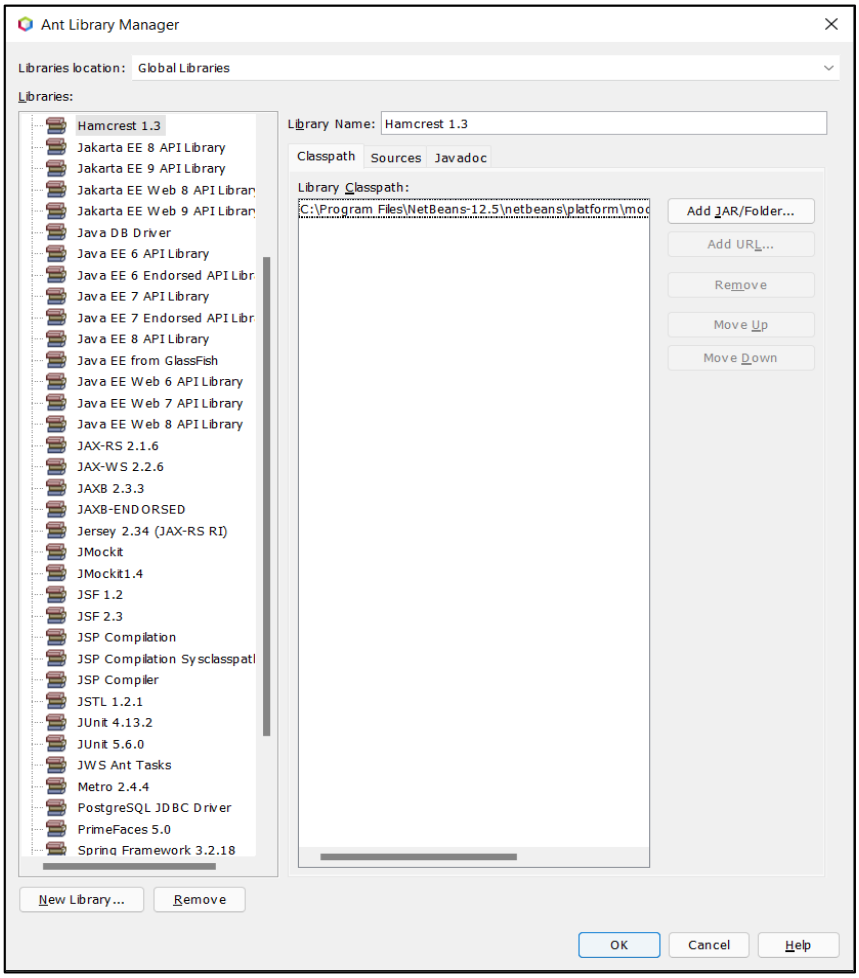
```
package Suite;

import org.junit.runner.RunWith;
import org.junit.runners.Suite;
import rachunki.AplikacjaTest;
import rachunki.model.RachunekTest1;
import rachunki.model.ZakupTest1;

@RunWith(Suite.class)
@Suite.SuiteClasses({ZakupTest1, RachunekTest1.class, AplikacjaTest.class})
public class RachunkiTestSuite { }
```



Dodatek 2

1. Instalacja biblioteki *JUnit 4.13.2*

1.1. W środowisku Apache ***NetBeans 18*** powinny być zainstalowane dwie biblioteki: ***Hamcrest 1.3*** oraz ***JUnit 4.13.2.*** Można to sprawdzić w następujący sposób: wybrać w ***Menu Bar*** pozycję ***Tools***. Na liście ***Tools*** należy kliknąć na pozycję ***Libraries*** i w oknie ***Ant Library Manager***, na liście ***Libraries*** wyszukać podane biblioteki. W przypadku istnienia tych bibliotek należy przejść do p.2.



1.2. W przypadku braku bibliotek, podanych w p. 1.1, należy wybrać pozycję ***Tools*** w ***Menu Bar***. Na tej liście kliknąć na pozycję ***Plugins*** i w oknie ***Plugins*** wybrać zakładkę ***Available Plugins*** i następnie, po kliknięciu na kolumnę ***Name*** (w celu posortowania nazw dodatków) wybrać z listy dodatek ***JUnit.*** Po zaznaczeniu dodatku w kolumnie ***Install,*** należy kliknąć na przycisk ***Install*** i zainstalować dodatek. W efekcie powinny pojawić się dwie nowe biblioteki podane w p. 1.1. W przypadku braku podanego dodatku należy skorzystać z informacji podanej na stronie: *https://github.com/junit-team/junit4/wiki/Download-and-Install*, pobrać podane pliki (należy wybrać wersję *junit 4.13.2* i *hamcrest 1.3* z pozycji *Latest Version*), umieścić je np. w katalogu ***… \NetBeans-18\platform\modules\ext*** i wykonać biblioteki po uruchomieniu okna ***Ant Library Manager*** **(**otworzone z **Tools/Libraries)** i kliknięciu na przycisk ***New Library.***

2. Instalacja narzędzia *JMockit* i wykonanie biblioteki *JMockit 1.27.*

2.1. Należy pobrać spakowany plik ***jmockit-1.27.zip***, który zawiera: jars, źródła, dokumentację, pliki konfiguracyjne **Maven**, pod adresem:
https://jar-download.com/artifacts/org.jmockit/jmockit/1.27/source-code

2.2. Po rozpakowaniu pobranego pliku ***jar_files.zip*** należy w oparciu o plik ***jmockit-1.27.jar*** wykonać bibliotekę ***JMockit 1.27,*** podobnie jak opisano tworzenie biblioteki ***JUnit 4.13.2*** oraz ***Hamcrest 1.3*** w p.1.2 w katalogu: ***\NetBeans-18\platform\modules***

3. Wykonanie projektu i dodanie plików testujących – zastosowanie *JUnit*

3.1. W celu utworzenia nowego projektu należy wybrać w ***Menu Bar*** pozycję ***Files***. Na tej liście kliknąć na pozycję ***New Project*****.** W oknie ***New Project****,* w liście ***Categories*** należy wybrać pozycję ***Java with Ant*****,** a w liście ***Projects*** należy wybrać pozycję ***Java Class Library*** i kliknąć na przycisk ***Next***. W kolejnym formularzu należy wpisać nazwę projektu w polu ***Project Name*** *i* wybrać położenie projektu w polu ***Project Location***.

3.2. W zakładce ***Projects*****,** w folderze ***Source Packages*** umieścić kopię pakietu z oprogramowaniem do testowania, wykonanym podczas lab 2- lab 11.

3.3. W oknie ***Project*** należy kliknąć prawym klawiszem myszy na nazwę projektu, wybrać z listy pozycje ***New/Other*****.** W oknie ***New File*** wybrać ***Unit Tests*** z listy ***Categories***, a z listy ***File Types*** wybrać ***Test for Existing Class***. W kolejnym oknie ***New Test for Existing Class*** w polu ***Class to Test*** wybrać klasę do testowania z pakietu utworzonego w p.3.2 w ***Source Packages*** projektu. Podczas tworzenia nowej klasy testującej należy z grupy ***Generated Code*** usunąć zaznaczenia typu ***Test Initializer*** oraz ***Test Finalizer*****.** Należy powtórzyć te czynności podczas tworzenia testów pozostałych wytypowanych klas do testowania. Pozostałe zaznaczenia określające zawartość wygenerowanych klas testujących należy dostosować do przyjętego sposobu testowania.

3.4. Wygenerowany plik zawiera szkielet kodu do testowania wybranej klasy. Należy go przystosować do potrzeb testowania. Biblioteki ***JUnit 4.13.2*** oraz ***Hamcrest 1.3*** powinny automatycznie być wstawione do folderu ***Test Libraries*** projektu (widok zakładki ***Projects***). Jeśli w środowisku ***Apache NetBeans*** zainstalowano kilka wersji bibliotek ***JUnit****,* wtedy podczas tworzenia plików do testowania należy wybrać wersję ***JUnit 4.13.2***.

3.5. Metody testujące należy wykonać zgodnie z poleceniami podanymi w instrukcji, opierając się na przykładach w **Dodatku 1**.

3.6. W celu uruchomienia testu należy w oknie zakładki ***Projects*** kliknąć prawym klawiszem myszy na nazwę pliku z testami i wybrać pozycję ***Test File***.

3.7. W przypadku tworzenia zestawu testów, należy wybrać projekt z klasami do testowania klikając prawym klawiszem myszy na nazwę projektu, następnie wybrać pozycje ***New/Other***. W oknie ***New File*** wybrać ***Unit Tests*** z listy ***Categories***, a z listy ***File Types*** wybrać ***Test Suite***. Następnie, należy postępować zgodnie z wytycznymi podanymi w p.2.7 przy definiowaniu zawartości pliku.

3.8. Na stronach *https://docs.oracle.com/javame/test-tools/javatest-441/html/junit.htm* i *https://www.vogella.com/tutorials/JUnit4/article.html,* znajdują się przydatne tutoriale, dotyczące testowania z wykorzystaniem narzędzia ***JUnit (należy wybrać właściwą wersję)*****.**

4. Tworzenie testów typu *JMockit*

4.1. Należy powtórzyć czynności z p.3.1, 3.2 oraz 3.3.

4.2. W oknie ***Project****,* w katalogu typu ***Test Libraries*** należącym do projektu z pakietem klas do testowania, należy dodać bibliotekę ***JMockit 1.27*** W tym celu należy prawym klawiszem myszy zaznaczyć katalog ***Test Libraries*** tego projektu i z listy wybrać pozycję ***Add Library…*** i następnie w oknie ***Add Library***, w liście ***Available Libraries*** zaznaczyć bibliotekę ***JMockit 1.27***. i kliknąć na przycisk ***Add Library*****.**

4.3. Metody testujące z wykorzystaniem narzędzia ***JMockit 1.27*** należy wykonać zgodnie z poleceniami podanymi w instrukcji (p. 6) opierając się na przykładach z **Dodatku 1**, p.3.1-3.3 oraz przykładach z **Dodatku 3**.

4.4. Na stronach:

JMockit_Tutotrial_Introduction.pdf
JMockit_Tutorial_Mocking.pdf

znajdują się tutoriale zawierające w rozdziałach: 1 (***Introduction***) i 2 (***Mocking***) przydatne informacje i przykłady dotyczące tworzenia testów z użyciem narzędzi ***JMockit 1.27*** oraz ***JUnit***.

Tutoriale dla wersji ***JMockit 1.49*** są dostępne ze stron:

http://jmockit.github.io/tutorial/Introduction.html
http://jmockit.github.io/tutorial/Mocking.html
http://jmockit.github.io/changes.html

Dodatek 3

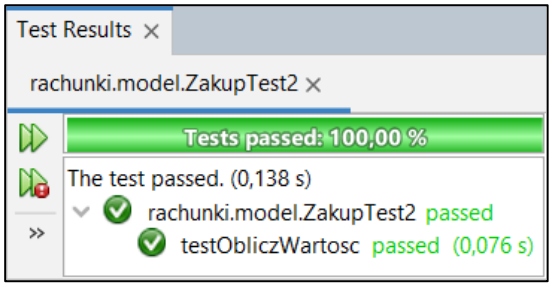
Pozostałe testy z wykorzystaniem biblioteki *JMockit 1.27,* prezentujące wybrane z możliwości symulowania własności obiektów podczas tworzenia oprogramowania.

1.8. Testowanie klasy *Zakup* – oparte na jednej instancji symulowanej instancji klasy *ProduktBezPodatku* (**@Injectable**) oraz symulacji atrybutu *ilosc* klasy *Zakup* oraz definicja instancji klasy testowanej *Zakup* (**@Test**ed)

Zastosowanie adnotacji
@Test
@RunWith(JMockit.class)
@Injectable
@Tested

Fazy testowania metody **obliczWartosc** klasy **Zakup** w metodzie testowej **testObliczWartosc**(), opartej na symulacji metod **getPodatek** oraz **obliczCeneBrutto** klasy **ProduktBezPodatku**
1)Nagrywanie – **new Expectations**
2)Odtwarzanie metody **obliczWartosc** klasy **Zakup**
3)Bez jawnej fazy weryfikacji

```
package rachunki.model;

import mockit.Expectations;
import mockit.Injectable;
import mockit.Tested;
import mockit.integration.junit4.JMockit;
import org.junit.Test;
import org.junit.runner.RunWith;
import static org.junit.Assert.assertEquals;

@RunWith(JMockit.class)
public class ZakupTest2 {
  @Tested
  Zakup tested;       //przykład automatycznego tworzenia testowanej klasy wraz z definicją symulowanych pól: produkt i ilosc
  @Injectable
  ProduktBezPodatku produkt1;    //symulowanie konkretnej instancji symulowanej klasy powiązanego z testowaną klasą Zakup
  @Injectable
  int ilosc = 2;                 //symulowanie wartosci pola ilosc w klasie testowanej Zakup

  @Test
  public void testObliczWartosc(/*@Injectable ("2")  int ilosc*/) {      //lub jako parametr
    new Expectations() {
      {
        produkt1.getPodatek();                result = -1;
        produkt1.obliczCeneBrutto();          result = 14;
      }
    };
    assertEquals(tested.obliczWartosc(-1), 28.0F, 0F);                  //dodatkowy test assertEquals
  }
}
```

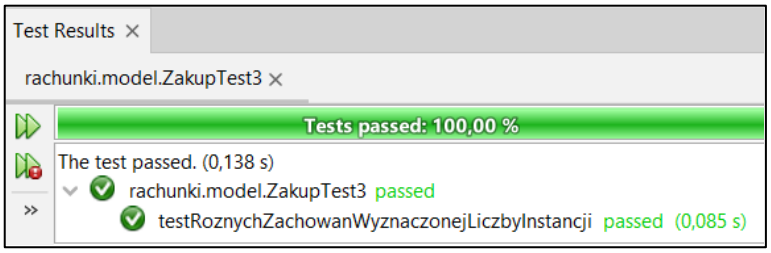


1.9. Testowanie klasy *Zakup* – specyfikacja zachowania kolejno tworzonych instancji w przyszłości (**@Capturing**)

Zastosowanie adnotacji
@Test
@RunWith(JMockit.class)
@Capturing

Fazy testowania metody **obliczWartosc** klasy **Zakup** w metodzie testowej **testRoznychZachowanWyznaczonejLiczbyInstancji**(), opartej na symulacji metod **getPodatek** oraz **obliczCeneBrutto** klasy **ProduktBezPodatku**
1)Nagrywanie **– new Expectations**
2)Odtwarzanie metody **obliczWartosc** klasy **Zakup**
3)Bez jawnej fazy weryfikacji

```
package rachunki.model;

import mockit.Capturing;
import mockit.Expectations;
import mockit.integration.junit4.JMockit;
import org.junit.Test;
import static org.junit.Assert.*;
import org.junit.runner.RunWith;

@RunWith(JMockit.class)
public class ZakupTest3 {
   @Capturing(maxInstances = 1)
    ProduktBezPodatku produkt1;     //tylko jedna instancja odtwarzająca nagraną metodę z rodziny obiektów ProduktBezPodatku
    @Capturing
    ProduktBezPodatku produkt2;   //dowolna ilość nowych następców jako instancji z rodziny obiektów ProduktBezPodatku

   @Test
   public void testRoznychZachowanWyznaczonejLiczbyInstancji(
        /*@Capturing(maxInstances = 1) ProduktBezPodatku produkt1,
         @Capturing ProduktBezPodatku produkt2*/)                    //lub jako parametry metody testującej
   {
      new Expectations() {
        {
           produkt1.obliczCeneBrutto();    result = 6.48F;        //1 raz może być użyte nagranie
           produkt2.obliczCeneBrutto();    result = 4.88F;        //dowolna liczba razy użycia nagranej metody
        }
      };
      ProduktZPodatkiem produkt11 = new ProduktZPodatkiem("8", 4, 7, new Promocja(50));
      ProduktZPodatkiem produkt21 = new ProduktZPodatkiem("4", 4, 22);
      ProduktBezPodatku produkt22 = new ProduktBezPodatku("1", 9.76F, new Promocja(50));

      assertEquals(6.48F, produkt11.obliczCeneBrutto(), 0F); //test tylko jednej instancji od symulowanej instancji produkt1
      assertEquals(4.88F, produkt21.obliczCeneBrutto(), 0F); //test pierwszej instancji od symulowanej instancji produkt2
      assertEquals(4.88F, produkt22.obliczCeneBrutto(), 0F); //test drugiej instancji od symulowanej instancji produkt2
      Zakup zakup1 = new Zakup(1,produkt11);
      assertEquals(zakup1.obliczWartosc(0), 6.48F, 0F);  //test metody klasy powiązanej: 1*4.88F
      Zakup zakup2 = new Zakup(2,produkt21);
      assertEquals(zakup2.obliczWartosc(0), 9.76F, 0F); //test metody klasy powiązanej: 2*4.88F
      Zakup zakup3 = new Zakup(3,produkt22);
      assertEquals(zakup3.obliczWartosc(0), 14.64F, 0F); //test metody klasy powiązanej: 3*4.88F
   }
}
```

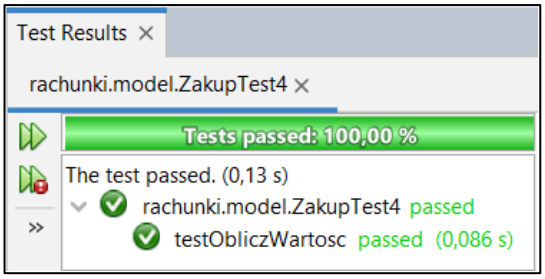


1.10. Testowanie klasy ***Zakup*** – Symulowanie metod klas potomnych lub implementacji interfejsów (**@Capturing**)

Zastosowanie adnotacji
@Test
@RunWith(JMockit.class)
@Capturing

Fazy testowania metody **obliczWartosc** klasy **Zakup** w metodzie testowej **testObliczWartosc**() powiązanej z instancją klasy ProduktZPodatkiem, opartej na symulacji metod **getPodatek** oraz **obliczCeneBrutto** klasy **ProduktBezPodatku**
1)Nagrywanie **– new Expectations**
2)Odtwarzanie metody **obliczWartosc** klasy **Zakup**
3)Bez jawnej fazy weryfikacji

```
package rachunki.model;

import mockit.Capturing;
import mockit.Expectations;
import mockit.integration.junit4.JMockit;
import static org.junit.Assert.assertEquals;
import org.junit.Test;
import org.junit.runner.RunWith;

@RunWith(JMockit.class)
public class ZakupTest4 {
   @Capturing
   ProduktBezPodatku produkt1;

   @Test
   public void testObliczWartosc() {
      new Expectations() {
         {
            produkt1.getPodatek();              result = 7F;
            produkt1.obliczCeneBrutto();        returns(3.21F);
         }
      };
      ProduktZPodatkiem produkt2 = new ProduktZPodatkiem("2", 3, 7);
      Zakup zakup = new Zakup(2, produkt2);
      assertEquals(6.42F, zakup.obliczWartosc(7) , 0F);
   }
}
```



1.11. Testowanie klasy ***Zakup*** – dwa przypadki częściowej symulacji: symulacja wybranych metod wybranej klasy oraz symulacja metod instancji wybranej klasy realizowane za pomocą przeciążonych konstruktorów klas z rodziny ***Expectations*.**

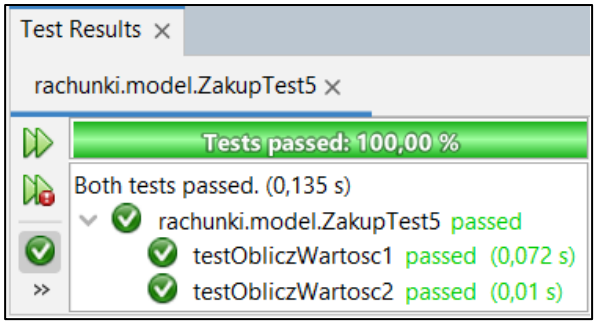
Zastosowanie adnotacji
@Test
@RunWith(JMockit.class)

Częściowe symulowanie metod wielu instancji danej klasy (**ProduktBezPodatku**) Fazy testowania metody **obliczWartosc** klasy **Zakup** w metodzie testowej **testObliczWartosc1** powiązanej z instancją klasy **ProduktBezPodatku**, opartej na symulacji metody **getPodatek** klasy **ProduktBezPodatku**
1)Nagrywanie **– new Expectations (ProduktBezPodatku.class)**
2)Odtwarzanie metody **obliczWartosc** klasy **Zakup**
3)Bez jawnej fazy weryfikacji

Częściowe symulowanie metod jednej instancji Fazy testowania metody **obliczWartosc** klasy **Zakup** w metodzie testowej **testObliczWartosc2** powiązanej z instancją klasy **ProduktBezPodatku**, opartej na symulacji metod **getPodatek** oraz **obliczCzescBruttoCeny** klasy **ProduktBezPodatku**
1)Nagrywanie **– new Expectations(produkt1)**
2)Odtwarzanie metody **obliczWartosc** klasy **Zakup**
3)Bez jawnej fazy weryfikacji

```
package rachunki.model;

import mockit.Expectations;
import mockit.integration.junit4.JMockit;
import static org.junit.Assert.assertEquals;
import org.junit.Test;
import org.junit.runner.RunWith;

@RunWith(JMockit.class)
public class ZakupTest5 {

  @Test
  public void testObliczWartosc1() {
    ProduktBezPodatku produkt1 = new ProduktBezPodatku("1", 1);
    new Expectations(ProduktBezPodatku.class) {
      {
        produkt1.getPodatek();   result = -1F;
      }
    };
      //użycie niesymulowanych konstruktorów
    ProduktBezPodatku produkt2 = new ProduktBezPodatku("2", 2);
    ProduktBezPodatku produkt3 = new ProduktBezPodatku("6", 2, new Promocja(50));

      // odtwarzanie metod symulowanych przez dwie instancje
    assertEquals(-1F, produkt2.getPodatek(), 0F);
    assertEquals(-1F, produkt3.getPodatek(), 0F);

      //wykonanie metod niesymulowanych
    assertEquals(produkt2.obliczCeneBrutto(), 2F, 0F);
    assertEquals(produkt3.obliczCeneBrutto(), 0.9F, 0F);

      //klasa korzystająca z metod symulowanych i niesymulowanych typu ProduktBezPodatku
    Zakup zakup1 = new Zakup(4, produkt2);
    Zakup zakup2 = new Zakup(1, produkt3);
    assertEquals(zakup1.obliczWartosc(-1), 8F, 0F);
    assertEquals(zakup2.obliczWartosc(-1), 0.9F, 0F);
  }
```

```
@Test
 public void testObliczWartosc2() {
      ProduktBezPodatku produkt1 = new ProduktBezPodatku("6", 2, new Promocja(50));
      new Expectations(produkt1) {
        {
           produkt1.obliczCzescBruttoCeny();          result = -1.1F;
           produkt1.getPodatek();                     result = -1;
        }
      };
         // odtwarzanie nagranych metod
      assertEquals(-1.1F, produkt1.obliczCzescBruttoCeny(), 0F);
      assertEquals(-1, produkt1.getPodatek(), 0F);

         // odtwarzanie nienagranych metod symulowanej instancji
      assertEquals(produkt1.obliczCeneBrutto(), 0.9F, 0F);
      assertEquals(produkt1.getNazwa(), "6");

         //testowanie klasy powiązanej z jedną instancją klasy częściowo symulowanej
      Zakup zakup = new Zakup(1, produkt1);
      assertEquals(zakup.obliczWartosc(-1), 0.9F, 0F);
   }
}
```

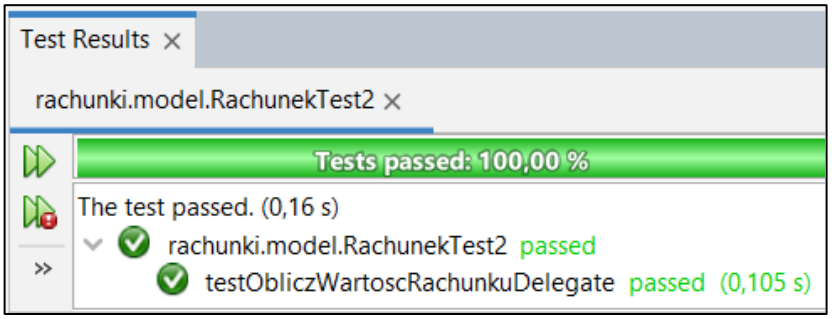


1.12. Testowanie klasy ***Rachunek***- symulowanie metody za pomocą specyfikacji jej działania (**Delegate**)

Zastosowanie adnotacji
@Test
@RunWith(JMockit.class)
@Mocked

Fazy testowania metody **obliczWartoscRachunku** w metodzie testowej **testObliczWartoscRachunkuDelegate** klasy **Rachunek**, powiązanej z instancjami klasy **Zakup**, opartej na symulacji metody **obliczWartosc** klasy **Zakup** za pomocą konstruktora klasy **Delegate**
1)Faza nagrywania- **new Expectations(), result, new Delegate**
2)Faza odtwarzania metody **obliczWartoscRachunku** klasy **Rachunek**
3)Brak jawnej fazy weryfikacji

```
package rachunki.model;

import mockit.Delegate;
import mockit.Expectations;
import mockit.Mocked;
import mockit.integration.junit4.JMockit
import static org.junit.Assert.assertEquals;
import org.junit.Test;
import org.junit.runner.RunWith;

@RunWith(JMockit.class)
public class RachunekTest2 {

  @Test
  public void testObliczWartoscRachunkuDelegate(@Mocked final Zakup zakup) {
    new Expectations() {
      {
        zakup.obliczWartosc(anyFloat);
        result = new Delegate() {
          float DelegateMethod(float i)
           {  if (i == -2 || i == 14)
               return 3.42F;
             else
               return 0F;}
        };
      }
    };
    Rachunek rachunek = new Rachunek(1);
    rachunek.wstawZakup(zakup);
    assertEquals(rachunek.obliczWartoscRachunku(-2), 3.42F, 0F);
    assertEquals(rachunek.obliczWartoscRachunku(14), 3.42F, 0F);
    assertEquals(rachunek.obliczWartoscRachunku(7), 0F, 0F);
  }
}
```



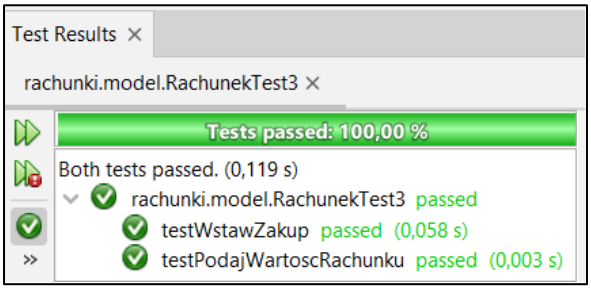
1.13. Testowanie klasy ***Rachunek*-** przechwytywanie argumentów metod symulowanych klas (metoda **withCapture**)

Zastosowanie adnotacji
@Test
@RunWith(JMockit.class)
@Mocked

Pobranie parametrów z jednego wywołania symulowanej metody. Fazy testowania metody **obliczWartoscRachunku** w metodzie testowej **testObliczWartoscRachunku**() klasy **Rachunek**
1)Brak jawnej fazy nagrywania
2)Faza odtwarzania metody **obliczWartoscRachunku**
3)Faza weryfikacji – **new Verifications, withCapture**

Pobranie parametrów z wielu wywołań symulowanej metody. Fazy testowania metody **wstawZakup** w metodzie testowej **testWstawZakup**() klasy **Rachunek**
1)Brak jawnej fazy nagrywania
2)Faza odtwarzania metody **wstawZakup**
3)Faza weryfikacji – **new Verifications, withCapture**

```
package rachunki.model;

import java.util.ArrayList;
import java.util.List;
import mockit.Mocked;
import mockit.Verifications;
import mockit.integration.junit4.JMockit;
import static org.junit.Assert.assertEquals;
import static org.junit.Assert.assertTrue;
import org.junit.Test;
import org.junit.runner.RunWith;

@RunWith(JMockit.class)
public class RachunekTest3 {
@Mocked
Rachunek rachunek;
  @Test
    public void testPodajWartoscRachunku() {
    System.out.println("Podaj_-wartosc - pobranie parametrów z jednego symulowanego wywołania metody");
    new Rachunek(1).obliczWartoscRachunku(-2);
       new Verifications() {
         {  float d;
            rachunek.obliczWartoscRachunku(d = withCapture());
            assertTrue(d < 0.0);
         }
       };
   }
@Test
public void testWstawZakup() {
    System.out.println("WstawZakup - pobranie parametrów z wielu symulowanych wywołań metody");
    ProduktBezPodatku produkty[] = { new ProduktBezPodatku("2", 2), new ProduktZPodatkiem("4", 4, 22),
        new ProduktBezPodatku("6", 2, new Promocja(50)),
        new ProduktZPodatkiem("8", 4, 7, new Promocja(50))  };
      Zakup zakupy[] = {  new Zakup(2, produkty[0]), new Zakup(3, produkty[1]),
                          new Zakup(2, produkty[2]), new Zakup(1, produkty[3])   };
    for (int i = 0; i < 4; i++)
       rachunek.wstawZakup(zakupy[i]);
    new Verifications() {
       {
          List<Zakup> lista = new ArrayList<>();
          rachunek.wstawZakup(withCapture(lista));
          assertEquals(4, lista.size());  }
    };
   }
}
```



1.14. Tworzenie zestawów testów – uzupełnienie p.3.4 z Dodatek 1.

```
package Suite;

import org.junit.runner.RunWith;
import org.junit.runners.Suite;
import rachunki.AplikacjaTest;
import rachunki.model.RachunekTest1;
import rachunki.model.RachunekTest2;
import rachunki.model.RachunekTest3;
import rachunki.model.ZakupTest1;
import rachunki.model.ZakupTest2;
import rachunki.model.ZakupTest3;
import rachunki.model.ZakupTest4;
import rachunki.model.ZakupTest5;

@RunWith(Suite.class)
@Suite.SuiteClasses({ZakupTest1, ZakupTest2.class, ZakupTest3.class, ZakupTest4.class, ZakupTest5.class,
RachunekTest1.class, RachunekTest2.class, RachunekTest3.class, AplikacjaTest.class})
public class RachunkiTestSuite {  }
```