Instrukcja 9

Laboratorium 12
Testy jednostkowe z użyciem narzędzi JUnit oraz JMockit

Cel laboratorium:
Nabycie umiejętności tworzenia testów jednostkowych za pomocą narzędzia JUnit oraz JMockit.

1. Wg wskazówek podanych w **Dodatku 2**, należy zainstalować biblioteki ***JUnit 4.12***, ***Hamcrest 1.3*** oraz ***JMockit 1.27*** oraz wykonać projekt w środowisku ***NetBeans 8.1***. Projekt powinien zawierać pakiet z klasami do testowania wykonanymi podczas lab2-11. Następnie, wg kolejnych wskazówek poniżej, należy dodawać testy ***JUnit*** wybranych klas.

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 ***TFabryka*** lub klasy z rodziny ***TProdukt1***. 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**.
   W tabelce poniżej podano informację dotyczącą wyboru metod do testowania oraz przykładów rozwiązań.

Grupa	Liczba metod do testowania	Przykłady testowanych metod		Przykłady testów	
		TFabryka	rodzina **TProdukt1**	**TFabrykaTest** (p.2.2 Dodatek1)	**TProdukt1Test** (p.2.3 Dodatek1)
1 osoba	1	Podaj_produkt	Podaj_cene	testPodaj_produkt	testPodaj_cene
2 osoby	2	Podaj_produkt,	Podaj_cene	testPodaj_produkt,	testPodaj_cene

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 ***TZakup*** z klasą z rodziny ***TProdukt1*** lub klasa ***TRachunek*** z klasą ***TZakup***. Należy zastosować w metodach testowych metody klasy ***Assert*** z biblioteki ***JUnit 4.12*** 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	
		TZakup	**TRachunek**	**TZakupTest** (p.2.4 Dodatek1)	**TRachunekTest** (p.2.5 Dodatek1)
1 osoba	1	Podaj_wartosc	Dodaj_Zakup,	testPodaj_wartosc	testDodaj_zakup
2 osoby	2	-	Dodaj_Zakup, Podaj_wartosc	-	testDodaj_zakup testPodaj_wartosc

4. Należy wykonać testy jednostkowe wybranych metod klasy opartej na wzorcu ***Fasada,*** podobnie jak klasa ***TAplikacja***. 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 4.12*** 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	Przykłady testów
		TAplikacja (p.2.6 Dodatek1)	**TAplikacjaTest** (p.2.6 Dodatek1)
1 osoba	2	Dodaj_produkt, Dodaj_Zakup,	testDodaj_produkt, testDodaj_zakup
2 osoby	3	Dodaj_produkt, Podaj_wartosc, Dodaj_Zakup,	testDodaj_produkt, testPodaj_wartosc, testDodaj_zakup

5. 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)
1 osoba	RachunkiTestSuite	RachunkiTestSuite_Entity RachunkiTestSuite_Control
2 osoby	RachunkiTestSuite	RachunkiTestSuite_Entity RachunkiTestSuite_Control RachunkiTestSuite_Control_Wstaw

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 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 rodzinny TProdukt1	Klasa TZakup	Klasa TZakup	Klasa TRachunek	
1	2	equals	equals	Podaj_ilosc, Dodaj_ilosc	Dodaj_zakup,	p. 3.1 Dodatek 1 lub p. 3.2 Dodatek 1
		Podaj_cene, Podaj_podatek	Podaj_wartosc	Podaj_wartosc	Podaj_wartosc	

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 TZakup	Klasa TRachunek	Klasa TFabryka	Klasa TAplikacja	
2	4	Podaj_ilosc, Dodaj_ilosc	Dodaj_zakup,	Podaj_produkt bez generowania wyjatku	Dodaj_produkt bez generowania wyjątku	p. 3.2 Dodatek 1, p.3.3, Dodatek 1
		Podaj_wartosc	Podaj_wartosc	Podaj_produkt z generowaniem wyjatku	Dodaj_produkt 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 instrukcji 5-7 -** po wykonaniu projektu wg informacji podanej w Dodatku 2 i umieszczeniu tam kodu przedstawianego w Dodatku 1 instrukcji 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 ***Podaj_produkt*** klasy ***TFabryka*** - zmiana definicji podanej w instrukcji 6. **Pełna walidacja poprawności danych wejściowych powinna być realizowana przez warstwę klienta aplikacji!**

```
public class TFabryka {
  public TProdukt1 Podaj_produkt(String dane[]) {
    TProdukt1 produkt = null;
    TPromocja promocja;
    switch (Integer.parseInt(dane[0])) {
      case 0:
        produkt = new TProdukt1(dane[1], Float.parseFloat(dane[2]));
        break;
      case 1:
        promocja = new TPromocja(Float.parseFloat(dane[3]));
        produkt = new TProdukt1(dane[1], Float.parseFloat(dane[2]), promocja);
        break;
      case 2:
        produkt = new TProdukt2(dane[1], Float.parseFloat(dane[2]), Float.parseFloat(dane[3]));
        break;
      case 3:
        promocja = new TPromocja(Float.parseFloat(dane[4]));
        produkt = new TProdukt2(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 TAplikacja dodano do definicji metod, wywołujących metodę klasy ***TFabryka*** dodano klauzulę **throws IllegalFormatCodePointException -** zmiana definicji podanej w instrukcjach 6 i 7:

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

public void Wstaw_zakup(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 rachunek1, podane w części Dodatek 1 instrukcji 5 powinny być klasami publicznymi**:

```
public class TRachunek
public class TZakup
public class TProdukt1
public class TProdukt2
public class TPromocja
```

2. Testy jednostkowe z wykorzystaniem narzędzia *JUnit 4.12*

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 ***rachunek1*** (pakiet klas testujących dedykowanych klasom testowanym w projekcie) należy dodać następujące puste interfejsy (***Java Interface***): ***Test_Control, Test_Entity*** oraz ***Test_Koszt*** 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 *TZakup* i z rodziny *TProdukt1*

2.1.1. String **dane_produktow**[][] – dwuwymiarowa tablica zawierająca w pierwszych ośmiu wierszach dane do utworzenia ośmiu obiektów z rodziny ***TProdukt1*** (**Dodatek 1**, Instrukcja 7, klasy: ***TFabryka, TProdukt1, TProdukt2, TPromocja***). Wiersz dziewiąty zawiera dane niepoprawne, powodujące generowanie wyjątku ***IllegalFormatCodePointException*** przez metodę klasy ***TFabryka***.

2.1.2. **TProdukt1 produkty**[] – jednowymiarowa tablica ośmiu obiektów wzorcowych z rodziny ***TProdukt1***, zdefiniowanych na podstawie danych z tabeli ***dane_produktow*** z p. 2.1.1

2.1.3. **float ceny_produkty**[] – 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. **TZakup zakupy**[] – tablica ośmiu obiektów typu ***TZakup***, zdefiniowanych z wykorzystaniem obiektów z rodziny ***TProdukt1***, zdefiniowanych w tablicy ***produkty*** z p. 2.1.2.

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

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

Opis danych wzorcowych do testowania tworzenia i zawartości dwóch rachunków

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

2.1.8. String **dane_produktow_rachunki**[][][] –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. **TZakup zakupy_rachunki[]**[][] –dwuwymiarowa tablica obiektów typu ***TZakup.*** W testach stanowi ona zbiory wzorcowych obiektów typu ***TZakup,*** należących do dwóch obiektów typu ***TRachunek.*** Obiekty typu ***TZakup*** zawierają obiekty z rodziny ***TProdukt1,*** zdefiniowane w tablicy z p. 2.1.2.

2.1.10. **int ile_produktow_rachunki**[][] – 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 kategorie_wartosci_rachunku**[][] – dwuwymiarowa tablica wartości sześciu wartości dwóch rachunków, wynikających z kategorii cen podanych w tabeli z p.2.1.11.

```
package rachunek1;

public class Dane {
    //dane wzorcowe do testowania obiektów typu TZakup i z rodziny TProdukt1
   public String dane_produktow[][] = 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", "3", "3", "30"}, {"3", "8", "4", "7", "50"},
   {"4", "1", "1", "0", "0"} }; // 9-y el. zawiera dane do testowania generowania wyjątku przez klasę TFabryka
  public static TProdukt1 produkty[] = {
        new TProdukt1("1", 1), new TProdukt1("2", 2), new TProdukt2("3", 3, 14), new TProdukt2("4", 4, 22),
        new TProdukt1("5", 1, new TPromocja(30)), new TProdukt1("6", 2, new TPromocja(50)),
        new TProdukt2("7", 3, 3, new TPromocja(30)), new TProdukt2("8", 4, 7, new TPromocja(50)) };
  public float ceny_produkty[] = { 1F, 2F, 3.42F, 4.88F, 0.7F, 0.9F,3.99F, 6.48F };
```

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

   public float ceny_zakupy[] = {1F, 8F, 3.42F, 4.88F, 0.7F, 0.9F, 11.97F, 6.48F  };
    public int podatki_zakupy[] = {-1, -1, 14, 22, -1, -1, 3, 7  };

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

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

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

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

 public int ile_produktow_rachunki[][] = {
        {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  kategorie_wartosci_rachunkow[][] = {
         { 6.7F,  0F, 0F, 3.42F, 19.52F, 29.640001F},//wartości rachunku 1 wg kategorii
         { 9.8F, 11.97F, 12.96F, 0.0F, 4.88F, 39.61F}}//wartości rachunku 2 wg kategorii
    };
}
```

2.2. **Test jednostkowy** klasy ***TFabryka*** **(wynik działania: p.2.7.1, 2.7.3, 2.7.4)** – przykłady prostego testu (k1.2) porównującego osiem wyników działania metody ***Podaj_produkt()*** tworzącej cztery różne typy obiektów z rodziny ***TProdukt1*** 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 ***Podaj_produkt*** (k1.2).

Zastosowanie adnotacji
@Test
@Before
@Category
@Rule

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

```
package rachunek1;
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;

@Category({Test_Control.class, Test_Entity.class})  //określenie kategorii testu, zastosowanie -  p.2.7.1, 2.7.3
public class TFabrykaTest {
   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 testPodaj_produkt() {
      System.out.println("Podaj_produkt");
      TFabryka instance = new TFabryka();
      for (int i = 0; i < 8; i++) {
         TProdukt1 result = instance.Podaj_produkt(dane.dane_produktow[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.Podaj_produkt(dane.dane_produktow[8]);           // wyjątku IllegalFormatCodePointException
                                                                // przez metodę Podaj_produkt
   }
 }
```

2.3. **Testy jednostkowe klas *TProdukt1* i *TProdukt2* (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 ***testPodaj_cene()*** sprawdza się wyniki zwracane przez metodę ***Podaj_cene*** *(***k2***)* dla ośmiu obiektów z rodziny ***TProdukt1***, 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

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 rachunek1;

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;

@Category({ Test_Entity.class})          //określenie kategorii testu, zastosowanie - p.2.7.2
@RunWith(Parameterized.class)
public class TProdukt1Test {
   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 testPodaj_cene() {
      System.out.println("Podaj_cene");
      float result1 = dane.produkty[numer1].Podaj_cene();
      float result2 = dane.ceny_produkty[numer1];
      assertEquals(result1, result2, 0F);  //k2 – test wyznaczania poprawnej wartości cen brutto produktów
   }
}
```

2.4. **Testy jednostkowe klasy *TZakup* (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 ***testPodaj_wartosc()*** sprawdza się wyniki zwracane przez metodę ***Podaj_wartosc*** dla ośmiu obiektów z rodziny ***TZakup***, 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 rachunek1;

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;

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

   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 testPodaj_wartosc() {
      System.out.println("Podaj_wartosc");
      assertEquals(dane.ceny_zakupy[numer1], //k1.1 test wyznaczania wartości zakupów: 1-y zbiór danych
                  dane.zakupy[numer1].Podaj_wartosc(dane.podatki_zakupy[numer1]), 0.0F);
      assertEquals(dane.ceny_zakupy[numer2],
                   dane.zakupy[numer2].Podaj_wartosc(dane.podatki_zakupy[numer2]), 0.0F);//k1.2
                          //test wyznaczania wartości zakupów: 2-i zbiór danych
   }
}
```

2.5. **Testy jednostkowe klasy TRachunek (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 ***TZakup*** 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 ***TRachunek***, które w metodzie testowej ***testDodaj_zakup*** 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: ***testDodaj_zakup, testPodaj_wartosc***. Kolejna metoda testowa korzysta z danych utworzonych w poprzedniej metodzie testowej. Metoda testowa ***testPodaj wartosc()*** operuje na rachunkach wypełnionych obiektami typu ***TZakup*** w metodzie ***TestDodaj_zakup*.**

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 rachunek1;

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;

@Category({Test_Entity.class})          //określenie kategorii testu, zastosowanie - p.2.7.2
@FixMethodOrder(MethodSorters.NAME_ASCENDING)
@RunWith(Parameterized.class)
public class TRachunekTest {

   static Dane dane;
   static TRachunek instances[];

   @Parameterized.Parameter
   public TZakup[][] zakupy1;

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

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

 @Test
 public void testDodaj_zakup() {
    System.out.println("Dodaj_zakup");
    for (int i = 0; i < 2; i++) {
       for (int j = 0; j < 5; j++) {
          instances[i].Dodaj_zakup(zakupy1[i][j]);
          TZakup 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).Podaj_ilosc();
        instances[i].Dodaj_zakup(zakupy1[i][0]);
        assertEquals(rozmiar1, instances[i].getZakupy().size());//k1.2- test spójności danych podczas
                                                                // dodawania podobnych zakupów
        assertEquals(instances[i].getZakupy().get(0).Podaj_ilosc(), ile * 2); //k1.3 test algorytmu dodawania
                                                                //podobnych zakupów
    }
  }
 @Test
 public void testPodaj_wartosc() {
    System.out.println("Podaj_wartosc");
     for (int i = 0; i < 2; i++)
        for (int j = 0; j < 5; j++)
          assertEquals(dane. kategorie_wartosci_rachunkow[i][j],
                instances[i].Podaj_wartosc(dane.kategorie[j]), 0F);//k2 – test obliczania wartości
                                                                //rachunku w różnych kategoriach
  }
}
```

2.6. **Testy jednostkowe klasy *TAplikacja* (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)**: ***testDodaj_produkt, testDodaj_zakup, testPodaj_wartosc***. Przed wywołaniem metod testowych wywołana jest metoda ***SetUp*** dzięki zastosowaniu adnotacji **@BeforeClass** – metoda ta tworzy obiekt typu ***TAplikacja***. Metody działające w podanym porządku umożliwiają po dodaniu produktów w metodzie ***testDodaj_produkt*** wykonać testy: dodawania zakupów w metodzie ***testDodaj_zakup***, obliczania wartości rachunków w różnych kategoriach w metodzie ***testPodaj_wartosc***. Metody testowe ***testDodaj_produkt*** oraz ***testDodaj_zakup*** testują również przypadek podania niepoprawnej wartości w danych wejściowych, które powodują generowanie wyjątku przez metodę ***Podaj_produkt*** klasy ***TFabryka*** – wyjątek ten jest obsługiwany w metodzie testowej ***testPodaj_wartosc*** za pomocą mechanizmu **@Rule.** Za pomocą adnotacji **@Category** dokonano różnych klasyfikacji wszystkich metod testowych i wybranej metody ***testPodaj_wartosc***.

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 rachunek1;

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;

@Category({Test_Control.class, Test_Entity.class})  //określenie kategorii testu, zastosowanie -  p.2.7.1, 2.7.3
@FixMethodOrder(MethodSorters.NAME_ASCENDING)
public class TAplikacjaTest {

   static Dane dane;

   static TAplikacja instance;

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

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

```
@Test
public void testDodaj_produkt() {
  int indeksy_produktow[] = {0, 1, 2, 3, 4, 5, 6, 7, 7, 7};
  System.out.println("Dodaj_produkt");
  for (int i = 0; i < 2; i++)
   for (int j = 0; j < 5; j++) {
    instance.Dodaj_produkt(dane.dane_produktow_rachunki[i][j]);
    int ile1 = instance.getProdukty().size();
    instance.Dodaj_produkt(dane.dane_produktow_rachunki[i][j]);//powtórzenia wartości elementów
                                                               // dla i=1 oraz j=3, j=4
    int ile2 = instance.getProdukty().size();
    TProdukt1 result = instance.getProdukty().get(ile2 - 1);
    assertEquals(dane.produkty[indeksy_produktow[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.Dodaj_produkt(dane.dane_produktow[8]);
}

@Test
public void testDodaj_zakup() {
    System.out.println("Wstaw_zakup");
    instance.setRachunki(Arrays.asList(dane.rachunki));
    for (int i = 0; i < 2; i++) {
      for (int j = 0; j < 5; j++) {
        instance.Wstaw_zakup(i + 1, dane.ile_produktow_rachunki[i][j],
                                    dane.dane_produktow_rachunki[i][j]);
        TZakup zakup1 = instance.getRachunki().get(i).getZakupy().get(j);
        assertEquals(zakup1, dane.zakupy_rachunki[i][j]); //k2.1 test dodawania zakupów
      }
      int rozmiar = instance.getRachunki().get(i).getZakupy().size();
      instance.Wstaw_zakup(i + 1, dane.ile_produktow_rachunki[i][0],
                                  dane.dane_produktow_rachunki[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).Podaj_ilosc(),
                   dane.zakupy_rachunki[i][0].Podaj_ilosc());   //k2.3 test algorytmu dodawania
    }                                                           // podobnych zakupów
   exception.expect(IllegalFormatCodePointException.class);
   exception.expectMessage("Code point = 0x0");
    instance.Wstaw_zakup(1, 1, dane.dane_produktow[8]); //obsługa wyjątku w testowanej metodzie
  }

@Test
@Category(Test_Koszt.class)          //określenie kategorii testu – przykład zastosowania w p.2.7.4
  public void testPodaj_wartosc() {
     System.out.println("Podaj_wartosc");
     for (int i = 0; i < 2; i++)
       for (int j = 0; j < 6; j++)
         assertEquals(dane.kategorie_wartosci_rachunkow[i][j],          //k3 – test obliczania wartości
                      instance.Podaj_wartosc(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(Test_Control.class): *TFabrykaTest, TAplikacjaTest*

```
package Suite;

import org.junit.experimental.categories.Categories;
import org.junit.runner.RunWith;
import rachunek1.TAplikacjaTest;
import rachunek1.TFabrykaTest;
import rachunek1.TProdukt1Test;
import rachunek1.TRachunekTest;
import rachunek1.TZakupTest;
import rachunek1.Test_Control;

@Categories.SuiteClasses({TFabrykaTest.class,TAplikacjaTest.class, TProdukt1Test.class, TZakupTest.class,
                          TRachunekTest.class })
@RunWith(Categories.class)
@Categories.IncludeCategory(Test_Control.class)
public class RachunkiTestSuite_Control {  }
```

```
Podaj_produkt
Dodaj_produkt
Wstaw_zakup
Podaj_wartosc
```

Wynik testu: TFabrykaTest, TAplikacjaTest

2.7.2. Wyniki testów wykonanych przez klasy należące tylko do kategorii @Category(Test_Entity.class): *TProdukt1Test, TZakupTest, TRachunekTest*

```
package Suite;

import org.junit.experimental.categories.Categories;
import org.junit.runner.RunWith;
import rachunek1.TAplikacjaTest;
import rachunek1.TFabrykaTest;
import rachunek1.TProdukt1Test;
import rachunek1.TRachunekTest;
import rachunek1.TZakupTest;
import rachunek1.Test_Control;

@Categories.SuiteClasses({TFabrykaTest.class, TAplikacjaTest.class, TProdukt1Test.class, TZakupTest.class,
                          TRachunekTest.class,})
@RunWith(Categories.class)
@Categories.ExcludeCategory(Test_Control.class)
public class RachunkiTestSuite_Entity { }
```

```
equals
Podaj_cene
equals
Podaj_cene
equals
Podaj_cene
equals
Podaj_cene
```

```
equals
Podaj_cene
equals
Podaj_cene
equals
Podaj_cene
equals
Podaj_cene
```

Wyniki testu TProdukt1Test

```
Podaj_wartosc
Podaj_wartosc
Podaj_wartosc
Podaj  wartosc
```

Wyniki testu TZakupTest

```
Dodaj_zakup
Podaj_wartosc
```

Wyniki testu TRachunekTest

2.7.3. Wyniki testów wykonanych przez klasy należące do kategorii @Category(Test_Control.class) z wyłączeniem metody *testPodaj_wartosc* klasy *TAplikacjaTest* zaliczonej do kategorii @Categorii(Test_koszt.class) *: TFabrykaTest, TAplikacjaTest*

```
package Suite;

import org.junit.experimental.categories.Categories;
import org.junit.runner.RunWith;
import rachunek1.Test_Control;
import rachunek1.Test_Koszt;

@Categories.SuiteClasses({RachunkiTestSuite_Control.class})
@RunWith(Categories.class)
@Categories.IncludeCategory(Test_Control.class)
@Categories.ExcludeCategory(Test_Koszt.class)
public class RachunkiTestSuite_Control_Wstaw {}
```

```
Podaj_produkt
Dodaj_produkt
Wstaw_zakup
```

Wynik testu: *TFabrykaTest, TAplikacjaTest* z wyłączeniem metody testowej *testPodaj_wartosc()*

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 rachunek1.TAplikacjaTest;
import rachunek1.TFabrykaTest;
import rachunek1.TProdukt1Test;
import rachunek1.TRachunekTest;
import rachunek1.TZakupTest;

@SuiteClasses({TFabrykaTest.class, TAplikacjaTest.class, TProdukt1Test.class, TZakupTest.class
                TRachunekTest.class})
@RunWith(Suite.class)
public class RachunkiTestSuite { }
```

```
Podaj_produkt
Dodaj_produkt
Wstaw_zakup
Podaj_wartosc
```

Wynik testu: TFabrykaTest,TAplikacjaTest

```
equals
Podaj_cene
equals
Podaj_cene
equals
Podaj_cene
equals
Podaj_cene
```

```
equals
Podaj_cene
equals
Podaj_cene
equals
Podaj_cene
equals
Podaj_cene
```

Wyniki testu TProdukt1Test

```
Podaj_wartosc
Podaj_wartosc
Podaj_wartosc
Podaj_wartosc
```

Wyniki testu TZakupTest

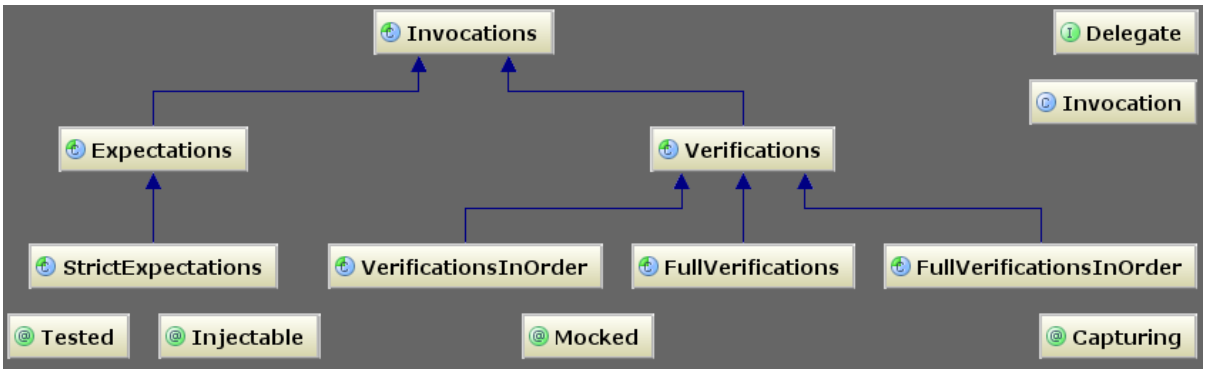
```
Dodaj_zakup
Podaj_wartosc
```

Wyniki testu TRachunekTest

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 strony http://jmockit.org/tutorial.html.
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 ***TZakup*** oparte na jednej jawnie deklarowanej instancji symulowanej klasy ***TProdukt1*** za pomocą adnotacji **@Mocked** w każdym teście i tworzeniu obiektu z rodziny TProdukt1 za pomocą odpowiedniego konstruktora, który jest automatycznie symulowanym obiektem.

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

Fazy testowania metody **equals** klasy **TZakup** w metodzie testowej **testEquals**, opartej na domyślnej symulacji metody **equals** klasy **TProdukt1**
1)Bez jawnie zdefiniowanej fazy nagrywania
2) Odtwarzanie metody **equlas** klasy **TZakup**
3) Faza weryfikacji **-new FullVerificationsInOrder(), maxTimes**

Fazy testowania metody **Podaj_wartosc** klasy **TZakup** w metodzie testowej **testPodaj_wartosc**(), opartej na symulacji metod **Podaj_podatek** oraz **Podaj_cene** klasy **TProdukt1**
1)Faza nagrywania**-new Expectations(), result**
2) Odtwarzanie metody **Podaj_wartosc** klasy **TZakup**
3)Faza weryfikacji - **new FullVerificationsInOrder(), maxTimes**

```
package rachunek1;

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 TZakupTest1 {

   @Mocked
   TProdukt1 produkt;

   @Test
   public void testEquals() {
     TProdukt1 produkt2 = new TProdukt2("8", 4, 7, new TPromocja(50));// dowolmy konstruktor
     TZakup zakupy[] ={ new TZakup(2, produkt), new TZakup(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 testPodaj_wartosc((@Mocked TProdukt1 produkt1) {
      TProdukt1 produkt2 = new TProdukt1("8", 4);// dowolny konstruktor
      TZakup zakupy[] =  { new TZakup(2, produkt1),  new TZakup(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("Podaj_wartosc");
      new Expectations() {
        {
           produkt1.Podaj_podatek();          result = podatki[0];
           produkt1.Podaj_cene();             result = ceny1[0];
           produkt2.Podaj_podatek();          result = podatki[1];
           produkt2.Podaj_cene();             result = ceny1[1];
        }
      };
      for (int j = 0; j < 2; j++)
        assertEquals(zakupy[j].Podaj_wartosc(podatki[j]), ceny2[j], 0F);  //dodatkowy test assertEquals

     new FullVerificationsInOrder() {
        {
           produkt1.Podaj_podatek();          maxTimes = 1;
           produkt1.Podaj_cene();             maxTimes = 1;
           produkt2.Podaj_podatek();          maxTimes = 1;
           produkt2.Podaj_cene();             maxTimes = 1;
        }
     };
   }
}
```

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

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

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

Fazy testowania metody **Dodaj_zakup** w metodzie testowej **testDodaj_zakup**() klasy **TRachunek,** powiązanej z instancjami klasy **TZakup**, opartej na symulacji metody **Podaj_ilosc** oraz **Dodaj_ilosc** klasy **TZakup**
1)Faza nagrywania – **new StrictExpectations(), returns**
2) Faza odtwarzania metody **Dodaj_zakup**
3)Faza weryfikacji -new FullVerificationsInOrder(), maxTimes

Fazy testowania metody **Podaj_wartosc** w metodzie testowej **testPodaj_wartosc**() klasy **TRachunek**, powiązanej z instancjami klasy **TZakup**, opartej na symulacji metody **Podaj_wartosc** klasy **TZakup**
1)Faza nagrywania**- new Expectations(), result**
2)Faza odtwarzania metody **Podaj_wartosc**
3)Faza weryfikacji **- new VerificationsInOrder(), times**

```
package rachunek1;

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 TRachunekTest1 {

  @Injectable
  TZakup zakup1,zakup2, zakup3;

  @Test
  public void testSzukaj_zakup() {
    System.out.println("Szukaj_zakup");
    TZakup zakupy[] = {zakup1, zakup2, zakup3};
    TRachunek rachunek = new TRachunek(1);
    rachunek.setZakupy(Arrays.asList(zakupy));
    for (int i = 0; i < 3; i++)
      assertEquals(rachunek.Szukaj_zakup(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 testDodaj_zakup() {
    System.out.println("Dodaj_zakup");
    TZakup zakupy[] = {zakup1, zakup2, zakup3, zakup1};
    TRachunek rachunek = new TRachunek(1);
    new StrictExpectations() {
      {
        zakup1.Podaj_ilosc();          returns(1);
        zakup1.Dodaj_ilosc(1);         returns(2);
        zakup1.Podaj_ilosc();          returns(2);
      }
    };
   for (int i = 0; i < 4; i++)
      rachunek.Dodaj_zakup(zakupy[i]);
    assertEquals(rachunek.getZakupy().get(0).Podaj_ilosc(), 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 testPodaj_wartosc() {
    TZakup zakupy[] = {zakup1, zakup2, zakup3};
    float wartosci_rachunku[] = {9.8F, 0.0F, 0.0F, 0.0F, 4.88F, 14.68F};
    int podatki[] = {-1, 3, 7, 14, 22, -2};
    System.out.println("Podaj_wartosc");
    TRachunek rachunek = new TRachunek(1);
    new Expectations() {
      {
        zakupy[0].Podaj_wartosc(-1);          result = 1.8F;
        zakupy[1].Podaj_wartosc(-1);          result = 8F;
        zakupy[2].Podaj_wartosc(22);          result = 4.88F;
        zakupy[0].Podaj_wartosc(-2);          result = 1.8F;
        zakupy[1].Podaj_wartosc(-2);          result = 8.0F;
        zakupy[2].Podaj_wartosc(-2);          result = 4.88F;
      }
    };
    rachunek.setZakupy(Arrays.asList(zakupy));
    for (int i = 0; i < 6; i++)
      assertEquals(wartosci_rachunku[i], rachunek.Podaj_wartosc(podatki[i]), 0F);
    new VerificationsInOrder() {
      {
        zakupy[0].Podaj_wartosc(-1);          times = 1;
        zakupy[1].Podaj_wartosc(-1);          times = 1;
        zakupy[2].Podaj_wartosc(22);          times = 1;
        zakupy[0].Podaj_wartosc(-2);          times = 1;
        zakupy[1].Podaj_wartosc(-2);          times = 1;
        zakupy[2].Podaj_wartosc(-2);          times = 1;
      }
    };
  }
}
```

3.3. Testowanie klasy ***TAplikacja*** – symulacja metody klasy ***TFabryka*** powiązanej z klasą ***TAplikacja***; testowanie metody ***Dodaj_produkt*** w zakresie poprawnych danych i niepoprawnych danych.

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

Fazy testowania metody **Dodaj_produkt** w metodzie testowej **testDodaj_produkt** klasy **TAplikacja**, powiązanej z instancjami klasy **TFabryka**, opartej na symulacji metody **Podaj_produkt** klasy **TFabryka**
1)Faza nagywania-**new Expectations(), result**
2)Faza odtwarzania – wykonanie metody **Dodaj_produkt** klasy **TAplikacja**
3)Brak jawnej fazy weryfikacji

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

-

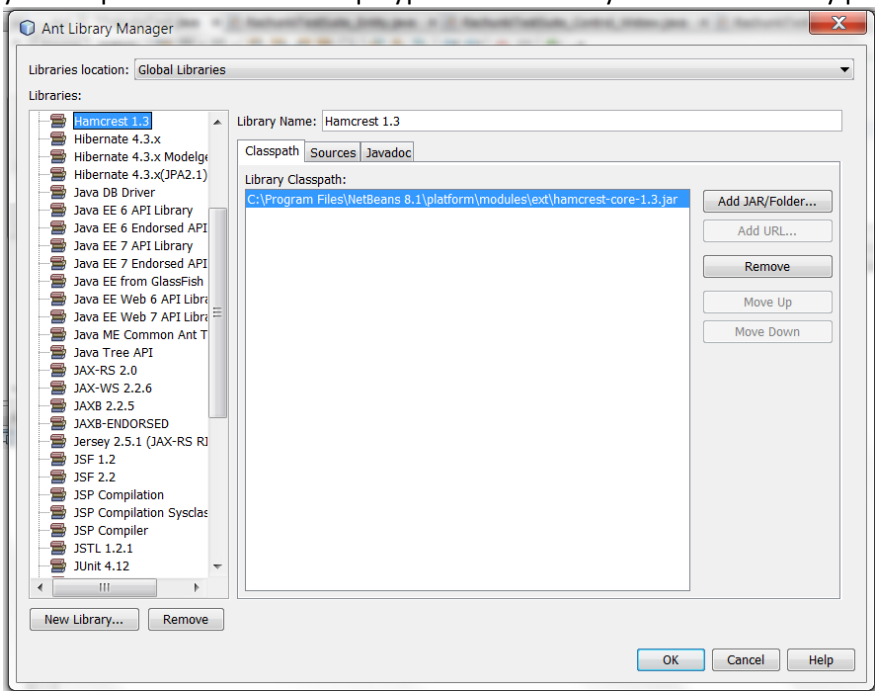
```
package rachunek1;
import java.util.IllegalFormatCodePointException;
import mockit.Expectations;
import mockit.Mocked;
import mockit.integration.junit4.JMockit;
import org.junit.Test;
import static org.junit.Assert.*;
import org.junit.runner.RunWith;

@RunWith(JMockit.class)
public class TAplikacjaTest {
        TProdukt1 produkty[] = {
        new TProdukt1("1", 1),
        new TProdukt2("3", 3, 14),
        new TProdukt1("5", 1, new TPromocja(30)),
        new TProdukt2("7", 3, 3, new TPromocja(30)), new TProdukt2("7", 3, 3, new TPromocja(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
TFabryka fabryka
@Test
public void testDodaj_produkt() {
      System.out.println("Dodaj_produkt");
      new Expectations() {
        {
          fabryka.Podaj_produkt(dane[0]);          result = produkty[0];
          fabryka.Podaj_produkt(dane[1]);          result = produkty[1];
          fabryka.Podaj_produkt(dane[2]);          result = produkty[2];
          fabryka.Podaj_produkt(dane[3]);          result = produkty[3];
        }
      };
      TAplikacja aplikacja = new TAplikacja();
      for (int i = 0; i < 5; i++) {
        aplikacja.Dodaj_produkt(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 testDodaj_produkt_blednyformat() {
        System.out.println("Dodaj_produkt_niepoprawny_format_danych");
        new Expectations() {
          {
            fabryka.Podaj_produkt((String[]) withNotNull());
            result=new IllegalFormatCodePointException(0);
          }
        };
        TAplikacja aplikacja = new TAplikacja();
         aplikacja.Dodaj_produkt(dane[5]);
      }
}
```

Dodatek 2

1. Instalacja biblioteki *JUnit 4.12*

1.1. W środowisku ***NetBeans 8.1*** powinny być zainstalowane dwie biblioteki: ***Hamcrest 1.3*** oraz ***JUnit 4.12.*** 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, umieścić je np. w katalogu ***… \NetBeans 8.1\platform\modules\ext*** i wykonać biblioteki po uruchomieniu okna ***Ant Library Manager*** 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://github.com/jmockit/artifacts1x/raw/master/jmockit-1.27.zip.

2.2. Po rozpakowaniu pobranego pliku należy w oparciu o plik ***jmockit.jar*** znajdujący się w podkatalogu***…..\jmockit-1.27\jmockit1.org*** wykonać bibliotekę ***JMockit 1.27,*** podobnie jak opisano tworzenie biblioteki ***JUnit 4.12*** oraz ***Hamcrest 1.3*** w p.1.2.

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*,** 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 4.12*** oraz ***Hamcrest 1.3*** powinny automatycznie być wstawione do folderu ***Test Libraries*** projektu (widok zakładki ***Projects***). Jeśli w środowisku ***NetBeans*** zainstalowano kilka wersji bibliotek ***JUnit***, wtedy podczas tworzenia plików do testowania należy wybrać wersję ***JUnit 4.12***.

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 *http://www.vogella.com/tutorials/JUnit/article.html*, znajdują się przydatne tutoriale, dotyczące testowania z wykorzystaniem narzędzia ***JUnit*.**

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*** 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 stronie http://jmockit.org/tutorial.html znajduje się tutorial zawierający w rozdziałach: 1 (***Introduction***) i 2 (***Mocking***) przydatne informacje i przykłady dotyczące tworzenia testów z użyciem narzędzi ***JMockit*** oraz ***JUnit***.

Dodatek 3

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

1.1. Testowanie klasy *TZakup* – oparte na jednej instancji symulowanej instancji klasy *TProdukt1* (**@Injectable**) oraz symulacji atrybutu *ilosc* klasy *TZakup* oraz definicja instancji klasy testowanej *TZakup* (**@Test**ed)

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

Fazy testowania metody **Podaj_wartosc** klasy **TZakup** w metodzie testowej **testPodaj_wartosc**(), opartej na symulacji metod **Podaj_podatek** oraz **Podaj_cene** klasy **TProdukt1**
1)Nagrywanie **– new Expectations**
2)Odtwarzanie metody **Podaj_wartosc** klasy **TZakup**
3)Bez jawnej fazy weryfikacji

```
package rachunek1;

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 TZakupTest2 {
  @Tested
  TZakup tested;      //przykład automatycznego tworzenia testowanej klasy wraz z definicją symulowanych pól: produkt i ilosc
  @Injectable
  TProdukt1 produkt1;      //symulowanie konkretnej instancji symulowanej klasy powiązanego z testowaną klasą TZakup
  @Injectable
  int ilosc = 2;           //symulowanie wartosci pola ilosc w klasie testowanej TZakup

  @Test
  public void testPodaj_wartosc(/*@Injectable ("4")  int ilosc*/) {      //lub jako parametr
    new Expectations() {
      {
        produkt1.Podaj_podatek();          result = -1;
        produkt1.Podaj_cene();             result = 14;
      }
    };
    assertEquals(tested.Podaj_wartosc(-1), 28.0F, 0F);          //dodatkowy test assertEquals
  }
}
```

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

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

Fazy testowania metody **Podaj_wartosc** klasy **TZakup** w metodzie testowej **testPodaj_wartosc**(), opartej na symulacji metod **Podaj_podatek** oraz **Podaj_cene** klasy **TProdukt1**
1)Nagrywanie **– new Expectations**
2)Odtwarzanie metody **Podaj_wartosc** klasy **TZakup**
3)Bez jawnej fazy weryfikacji

```
package rachunek1;

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 TZakupTest3 {
  @Capturing(maxInstances = 1)
   TProdukt1 produkt1;          //tylko jedna instancja odtwarzająca nagraną metodę z rodziny obiektów TProdukt1
   @Capturing
   TProdukt1 produkt2;          //dowolna ilość nowych następców jako instancji z rodziny obiektów TProdukt1

  @Test
  public void test_roznych_zachowan_dla_wyznaczonej_liczby_instancji(
      /*@Capturing(maxInstances = 1) TProdukt1 produkt1,
       @Capturing TProdukt1 produkt2*/)                    //lub jako parametry
  {
    new Expectations() {
      {
        produkt1.Podaj_cene();      result = 6.48F;     //1 raz może być użyte nagranie
        produkt2.Podaj_cene();      result = 4.88F;     //dowolna liczba razy użycia nagranej metody
      }
    };
    TProdukt2 produkt11 = new TProdukt2("8", 4, 7, new TPromocja(50));
    TProdukt2 produkt21 = new TProdukt2("4", 4, 22);
    TProdukt1 produkt22 = new TProdukt1("1", 9.76F, new TPromocja(50));

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

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

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

Fazy testowania metody **Podaj_wartosc** klasy **TZakup** w metodzie testowej **testPodaj_wartosc**() powiązanej z instancją klasy TProdukt2, opartej na symulacji metod **Podaj_podatek** oraz **Podaj_cene** klasy **TProdukt1**
1)Nagrywanie **– new Expectations**
2)Odtwarzanie metody **Podaj_wartosc** klasy **TZakup**
3)Bez jawnej fazy weryfikacji

```
package rachunek1;

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 TZakupTest4 {
   @Capturing
   TProdukt1 produkt1;

   @Test
   public void test podaj_wartosc() {
      new Expectations() {
         {
            produkt1.Podaj_podatek();          result = 7F;
            produkt1.Podaj_cene();             returns(3.21F);
         }
      };
      TProdukt2 produkt2 = new TProdukt2("2", 3, 7);
      TZakup zakup = new TZakup(2, produkt2);
      assertEquals(6.42F, zakup.Podaj_wartosc(7) , 0F);
   }
}
```

1.4. Testowanie klasy ***TZakup*** – 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 (**TProdukt1**) Fazy testowania metody **Podaj_wartosc** klasy **TZakup** w metodzie testowej **testPodaj_wartosc1** powiązanej z instancją klasy **TProdukt1**, opartej na symulacji metod **Podaj_podatek** klasy **TProdukt1**
1)Nagrywanie **– new Expectations (TProdukt1.class)**
2)Odtwarzanie metody **Podaj_wartosc** klasy **TZakup**
3)Bez jawnej fazy weryfikacji

Częściowe symulowanie metod jednej instancji Fazy testowania metody **Podaj_wartosc** klasy **TZakup** w metodzie testowej **testPodaj_wartosc2** powiązanej z instancją klasy **TProdukt1**, opartej na symulacji metod **Podaj_podatek** oraz **Czesc_brutto** klasy **TProdukt1**
1)Nagrywanie **– new Expectations(produkt1)**
2)Odtwarzanie metody **Podaj_wartosc** klasy **TZakup**
3)Bez jawnej fazy weryfikacji

```
package rachunek1;

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 TZakupTest5 {

  @Test
  public void testPodaj_wartosc1 {
    TProdukt1 produkt1 = new TProdukt1("1", 1);
    new Expectations(TProdukt1.class) {
      {
        produkt1.Podaj_podatek();       result = -1F;
      }
    };
      //użycie niesymulowanych konstruktorów
    TProdukt1 produkt2 = new TProdukt1("2", 2);
    TProdukt1 produkt3 = new TProdukt1("6", 2, new TPromocja(50));

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

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

      //klasa korzystająca z metod symulowanych i niesymulowanych typu TProdukt1
    TZakup zakup1 = new TZakup(4, produkt2);
    TZakup zakup2 = new TZakup(1, produkt3);
     assertEquals(zakup1.Podaj_wartosc(-1), 8F, 0F);
    assertEquals(zakup2.Podaj_wartosc(-1), 0.9F, 0F);
  }
```

```
@Test
 public void testPodaj_wartosc2() {
     TProdukt1 produkt1 = new TProdukt1("6", 2, new TPromocja(50));

     new Expectations(produkt1) {
       {
          produkt1.Czesc_brutto();          result = -1.1F;
          produkt1.Podaj_podatek();         result = -1;
       }
     };
        // odtwarzanie nagranych metod
     assertEquals(-1.1F, produkt1.Czesc_brutto(), 0F);
     assertEquals(-1, produkt1.Podaj_podatek(), 0F);

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

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

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

Zastosowanie adnotacji	Fazy testowania metody **Podaj_wartosc** w metodzie testowej **testPodaj_wartosc_Delegate** klasy **TRachunek**, powiązanej z instancjami klasy **TZakup**, opartej na symulacji metody **Podaj_wartosc** klasy **TZakup** za pomocą konstruktora klasy **Delegate**
@Test	**1)**Faza nagrywania- **new Expectations(), result, new Delegate**
@RunWith(JMockit.class)	**2)**Faza odtwarzania metody **Podaj_wartosc** klasy **TRachunek**
@Mocked	**3)**Brak jawnej fazy weryfikacji

```
package rachunek1;

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 TRachunekTest2 {

  @Test
  public void testPodaj_wartosc_Delegate(@Mocked final TZakup zakup) {
    new Expectations() {
      {
        zakup.Podaj_wartosc(anyInt);
        result = new Delegate() {
          float aDelegateMethod(int i)
            if (i == -2 || i == 14)
              return 3.42F;
            else
              return 0F;
        };
      }
    };
    TRachunek rachunek = new TRachunek(1);
    rachunek.Dodaj_zakup(zakup);
    assertEquals(rachunek.Podaj_wartosc(-2), 3.42F, 0F);
    assertEquals(rachunek.Podaj_wartosc(14), 3.42F, 0F);
    assertEquals(rachunek.Podaj_wartosc(7), 0F, 0F);
  }
}
```

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

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

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

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

```
package rachunek1;

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 TRachunekTest3 {
@Mocked
TRachunek rachunek;
  @Test
    public void testPodaj_wartosc() {
    System.out.println("Podaj_-wartosc - pobranie parametrów z jednego symulowanego wywołania metody");
    new TRachunek(1).Podaj_wartosc(-2);
       new Verifications() {
         {
            int d;
            rachunek.Podaj_wartosc(d = withCapture());
            assertTrue(d < 0.0);
         }
       };
  }
@Test
public void testDodaj_zakup() {
    System.out.println("Dodaj_zakup - pobranie parametrów z wielu symulowanych wywołań metody");
    TProdukt1 produkty[] = { new TProdukt1("2", 2), new TProdukt2("4", 4, 22),
        new TProdukt1("6", 2, new TPromocja(50)),  new TProdukt2("8", 4, 7, new TPromocja(50))  };
      TZakup zakupy[] = {  new TZakup(2, produkty[0]), new TZakup(3, produkty[1]),
                           new TZakup(2, produkty[2]), new TZakup(1, produkty[3])   };
      for (int i = 0; i < 4; i++)
         rachunek.Dodaj_zakup(zakupy[i]);
      new Verifications() {
         {
            List<TZakup> lista = new ArrayList<>();
            rachunek.Dodaj_zakup(withCapture(lista));
            assertEquals(4, lista.size());
         }
      };
  }
```