Testowanie i weryfikacja oprogramowania
Projekt 2 - Test-Driven Development

Mateusz Supronowicz (kierownik)
Norbert Grzyb
Paweł Polański

9 grudnia 2014

1 Wprowadzenie

1.1 Cel projektu

Celem projektu jest zapoznanie się z metodyką wytwarzania oprogramowania jaką jest Test-Driven Development. W poniższym dokumencie odnosząc się do niej będę posługiwał się skrótem TDD. Głównymi zadaniami do wykonania w terminie 10.12.2014r. było przeprowadzenie przykładów ze stron zamieszczonych w specyfikacji projektu otrzymanej na zajęciach, opis tych projektów, a także zaproponowanie własnego programu, który będzie następnie wytwarzany zgodnie z metodyką TDD. Program ten ma za zadanie realizować co najmniej 7 funkcjonalności.

1.2 Środowisko, narzędzia

System operacyjny: Windows 8.1
IDE: Netbeans 8.0.1
Biblioteki: JUnit 4.10, TestNG 6.8.1, Mockito 1.9.5

2 Projekt 1a z ISOD

2.1 Test 1

Przed pierwszym testem powstaje szkielet systemu w postaci klasy i 2 interfejsów. Następnie powstaje test, na obiektach pozornych, sprawdzający czy pojedynczy zarejestrowany klient otrzymuje wiadomość.

```
package rrs;

public class RaceResultsService {

    public void addSubscriber(Client client) {

    }
    public void send(Message message) {
```

```
    }
}
```

```
package rrs;

public interface Client {
    public void receive(Message message);
}
```

```
package rrs;

public interface Message {

}
```

Test:

```
package rrs;

import org.testng.annotations.*;
import static org.mockito.Mockito.*;

@Test
public class RaceResultsServiceTest {

    public void subscribedClientShouldReceiveMessage() {
        RaceResultsService raceResults = new RaceResultsService();
        Client client = mock(Client.class);
        Message message = mock(Message.class);

        raceResults.addSubscriber(client);
        raceResults.send(message);

        verify(client).receive(message);
    }
}
```

Wynik testu: Negatywny

Po implementacji:

```
package rrs;

public class RaceResultsService {
    private Client client;

    public void addSubscriber(Client client) {
        this.client = client;
    }
    public void send(Message message) {
        client.receive(message);
    }
}
```

Wynik testu: Pozytywny

2.2 Test 2

W drugim teście sprawdzamy czy każdy zarejestrowany klient otrzymuje wiadomość.

```java
package rrs;

import static org.testng.Assert.*;
import org.testng.annotations.*;
import static org.mockito.Mockito.*;

@Test
public class RaceResultsServiceTest {

    public void subscribedClientShouldReceiveMessage() {
        RaceResultsService raceResults = new RaceResultsService();
        Client client = mock(Client.class);
        Message message = mock(Message.class);

        raceResults.addSubscriber(client);
        raceResults.send(message);

        verify(client).receive(message);
    }

    public void messageShouldBeSendToAllSubscribedClients() {
        RaceResultsService raceResults = new RaceResultsService();
        Client clientA = mock(Client.class);
        Client clientB = mock(Client.class);
        Message message = mock(Message.class);

        raceResults.addSubscriber(clientA);
        raceResults.addSubscriber(clientB);
        raceResults.send(message);

        verify(clientA).receive(message);
        verify(clientB).receive(message);
    }
}
```

Wynik testu: Negatywny

Test nie przechodzi ponieważ dana funkcjonalność nie jest zaimplementowana. Dopisujemy obsługę wielu klientów i oba testy powinny przejść.

```java
package rrs;

import java.util.ArrayList;
import java.util.Collection;

public class RaceResultsService {
    private Collection<Client> clients = new ArrayList<Client>();

    public void addSubscriber(Client client) {
        clients.add(client);
    }
    public void send(Message message) {
        for( Client client:clients) {
            client.receive(message);
        }
    }
}
```

Wynik testu: Pozytywny

Ponieważ oba testy korzystają z tych samych obiektów pozornych następuje refaktor i wyciągnięcie mocków do metody wykonywanej przed każdym testem.

```
package rrs;

import org.testng.annotations.*;
import static org.mockito.Mockito.*;

@Test
public class RaceResultsServiceTest {
    private RaceResultsService raceResults;
    private Message message;
    private Client clientA;
    private Client clientB;

    @BeforeMethod        // wykonywane przed kazdym testem!!!!!
    public void setUp() {
        raceResults = new RaceResultsService();
        clientA = mock(Client.class);
        clientB = mock(Client.class);
        message = mock(Message.class);
    }
    public void subscribedClientShouldReceiveMessage() {
        raceResults.addSubscriber(clientA);
        raceResults.send(message);

        verify(clientA).receive(message);
    }

    public void messageShouldBeSendToAllSubscribedClients() {
        raceResults.addSubscriber(clientA);
        raceResults.addSubscriber(clientB);
        raceResults.send(message);

        verify(clientA).receive(message);
        verify(clientB).receive(message);
    }
}
```

2.3 Test 3

Test ten sprawdza czy osoby które nie są zarejestrowane nie dostają wiadomości.

```
package rrs;

import org.testng.annotations.*;
import static org.mockito.Mockito.*;

@Test
public class RaceResultsServiceTest {
    private RaceResultsService raceResults;
    private Message message;
    private Client clientA;
    private Client clientB;

    @BeforeMethod
    public void setUp() {
        raceResults = new RaceResultsService();
        clientA = mock(Client.class);
```

```
        clientB = mock(Client.class);
        message = mock(Message.class);
    }
    public void subscribedClientShouldReceiveMessage() {
        raceResults.addSubscriber(clientA);
        raceResults.send(message);

        verify(clientA).receive(message);
    }

    public void messageShouldBeSendToAllSubscribedClients() {
        raceResults.addSubscriber(clientA);
        raceResults.addSubscriber(clientB);
        raceResults.send(message);

        verify(clientA).receive(message);
        verify(clientB).receive(message);
    }
    public void notSubscribedClientShouldNotReceiveMessage() {
        raceResults.send(message);

        verify(clientA, never()).receive(message);
        verify(clientB, never()).receive(message);
    }
}
```

Wynik testu: Pozytywny

Ponieważ funkcjonalność ta została automatycznie zaimplementowana test od razu przechodzi. Po napisaniu następuje mały refaktor aby posortować metody.

```
package rrs;

import org.testng.annotations.*;
import static org.mockito.Mockito.*;

@Test
public class RaceResultsServiceTest {
    private RaceResultsService raceResults;
    private Message message;
    private Client clientA;
    private Client clientB;

    @BeforeMethod
    public void setUp() {
        raceResults = new RaceResultsService();
        clientA = mock(Client.class);
        clientB = mock(Client.class);
        message = mock(Message.class);
    }
    // zero subscribers
    public void notSubscribedClientShouldNotReceiveMessage() {
        raceResults.send(message);

        verify(clientA, never()).receive(message);
        verify(clientB, never()).receive(message);
    }
    // one subscriber
    public void subscribedClientShouldReceiveMessage() {
        raceResults.addSubscriber(clientA);
```

```
        raceResults.send(message);

        verify(clientA).receive(message);
    }
    // two subscribers
    public void messageShouldBeSendToAllSubscribedClients() {
        raceResults.addSubscriber(clientA);
        raceResults.addSubscriber(clientB);
        raceResults.send(message);

        verify(clientA).receive(message);
        verify(clientB).receive(message);
    }
}
```

2.4 Test4

Sprawdzamy czy klient zarejestrowany kilkukrotnie nie dostaje więcej niż jednej wiadomości.

```
package rrs;

import org.testng.annotations.*;
import static org.mockito.Mockito.*;

@Test
public class RaceResultsServiceTest {
    private RaceResultsService raceResults;
    private Message message;
    private Client clientA;
    private Client clientB;

    @BeforeMethod
    public void setUp() {
        raceResults = new RaceResultsService();
        clientA = mock(Client.class);
        clientB = mock(Client.class);
        message = mock(Message.class);
    }
    // zero subscribers
    public void notSubscribedClientShouldNotReceiveMessage() {
        raceResults.send(message);

        verify(clientA, never()).receive(message);
        verify(clientB, never()).receive(message);
    }
    // one subscriber
    public void subscribedClientShouldReceiveMessage() {
        raceResults.addSubscriber(clientA);
        raceResults.send(message);

        verify(clientA).receive(message);
    }
    // two subscribers
    public void messageShouldBeSendToAllSubscribedClients() {
        raceResults.addSubscriber(clientA);
        raceResults.addSubscriber(clientB);
        raceResults.send(message);
```

```
        verify(clientA).receive(message);
        verify(clientB).receive(message);
    }
    // client subscribed more then once
    public void shouldSendOnlyOneMessageToMultiSubscriber() {
        raceResults.addSubscriber(clientA);
        raceResults.addSubscriber(clientA);
        raceResults.send(message);

        verify(clientA).receive(message);
    }
}
```

Wynik testu: Negatywny

Test nie przechodzi więc dopisujemy funkcjonalność do głównego programu. Po zmianie rodzaju kolekcji napisany test i poprzednie przechodzą.

```
package rrs;

import java.util.Collection;
import java.util.HashSet;

public class RaceResultsService { // zmiana rodzaju kolekcji
    private Collection<Client> clients = new HashSet<>();

    public void addSubscriber(Client client) {
        clients.add(client);
    }
    public void send(Message message) {
        for( Client client:clients) {
            client.receive(message);
        }
    }
}
```

Wynik testu: Pozytywny

2.5 Test 5

Do programu dopisujemy nową pustą metodę. Piszemy test który sprawdza czy usunięci klienci dostają wiadomości.

```
package rrs;

import org.testng.annotations.*;
import static org.mockito.Mockito.*;

@Test
public class RaceResultsServiceTest {
    private RaceResultsService raceResults;
    private Message message;
    private Client clientA;
    private Client clientB;

    @BeforeMethod
    public void setUp() {
        raceResults = new RaceResultsService();
        clientA = mock(Client.class);
```

```
        clientB = mock(Client.class);
        message = mock(Message.class);
    }
    // zero subscribers
    public void notSubscribedClientShouldNotReceiveMessage() {
        raceResults.send(message);

        verify(clientA, never()).receive(message);
        verify(clientB, never()).receive(message);
    }
    // one subscriber
    public void subscribedClientShouldReceiveMessage() {
        raceResults.addSubscriber(clientA);
        raceResults.send(message);

        verify(clientA).receive(message);
    }
    // two subscribers
    public void messageShouldBeSendToAllSubscribedClients() {
        raceResults.addSubscriber(clientA);
        raceResults.addSubscriber(clientB);
        raceResults.send(message);

        verify(clientA).receive(message);
        verify(clientB).receive(message);
    }
    // client subscribed more then once
    public void shouldSendOnlyOneMessageToMultiSubscriber() {
        raceResults.addSubscriber(clientA);
        raceResults.addSubscriber(clientA);
        raceResults.send(message);

        verify(clientA).receive(message);
    }
    // removed subscriber should not received messages
    public void unsubscribedClientShouldNotReceiveMessages() {
        raceResults.addSubscriber(clientA);
        raceResults.removeSubscriber(clientA);
        raceResults.send(message);

        verify(clientA, never()).receive(message);
    }
}
```

Wynik testu: Negatywny

Test nie przechodzi więc zabieramy się do implementacji. Po poprawnej implementacji wszystkie testy powinny przechodzić.

```
package rrs;

import java.util.Collection;
import java.util.HashSet;

public class RaceResultsService {
    private Collection<Client> clients = new HashSet<>();

    public void addSubscriber(Client client) {
        clients.add(client);
```

```
    }
    public void removeSubscriber(Client client) {
        clients.remove(client);
    }
    public void send(Message message) {
        for( Client client:clients) {
            client.receive(message);
        }
    }
}
```

Wynik testu: Pozytywny

3 Wymagania na własne oprogramowanie z zastosowaniem metodyki TDD

Zespół decyduje się na zrealizowanie oprogramownia wykonującego proste operacje na macierzach. Główną i jedyną klasą będzie klasa Matrix zawierająca następujące funkcjonalności.

3.1 Tworzenie macierzy o zadanych wartościach.

Opis: Celem jest stworzenie obiektu typu Matrix wypełnionego liczbami wejściowymi.
Dane wejściowe: Tablica z wartościami typu double, liczba wierszy, liczba kolumn.
Wynik: Stworzenie obiektu Matrix wypełnionego podanymi wartościami wejściowymi.

3.2 Obsługa niepoprawnej liczby elementów przy tworzeniu macierzy.

Opis: W przypadku, gdy liczba elementów macierzy nie wypełnia podanej liczby wierszy i kolumn, pozostałe miejsca są wypełniane zerami.
Dane wejściowe: Tablica z wartościami typu double, liczba wierszy, liczba kolumn.
Wynik: Stworzenie obiektu Matrix wypełnionego podanymi wartościami wejściowymi, wypełnionego o wartości 0.

3.3 Tworzenie macierzy jednostkowej.

Dane wejściowe: n-liczba wierszy (kolumn).
Wynik: Stworzenie obiektu Matrix będącego n-wymiarową macierzą jednostkową.

3.4 Dodawanie dwóch macierzy.

Opis: Funkcja realizująca dodawanie dwóch macierzy.
Dane wejściowe: Dwie macierze.
Wynik: Nowa macierz będąca rezultatem dodawania dwóch macierzy wejściowych.

3.5 Dodawanie dwóch macierzy - rozmiary macierzy muszą być identyczne.

Opis: Funkcja realizująca obsługę błędnych rozmiarów macierzy przy dodawaniu dwóch macierzy.

Dane wejściowe: Dwie macierze o różnych rozmiarach.
Wynik: Rzucenie wyjątku RuntimeException informującego o niepoprawnych rozmiarach macierzy.

3.6 Mnożenie macierzy przez skalar.

Opis: Funkcja realizująca mnożenie macierzy przez skalar, czyli mnożąca poszczególne elementy macierzy przez zadaną wartość.
Dane wejściowe: Macierz wejściowa, skalar.
Wynik: Nowa macierz będąca rezultatem pomnożenia macierzy wejściowej przez podany skalar.

3.7 Obliczanie wyznacznika zadanej macierzy kwadratowej.

Dane wejściowe: Macierz kwadratowa.
Wynik: Wyznacznik macierzy podanej na wejściu.

3.8 Błąd obliczania wyznacznika zadanej macierzy niekwadratowej.

Dane wejściowe: Macierz niekwadratowa.
Wynik: Rzucenie wyjątku RuntimeException mówiącego o niepoprawnym formacie macierzy.

4 Przebieg realizacji klasy Matrix zgodnie z metodyką TDD.

4.1 Tworzenie macierzy o zadanych wartościach.

Pierwszym etapem naszego projektu jest stworzenie testu sprawdzającego czy klasa Matrix została utworzona poprawnie. Test wykazuje błąd, ponieważ kontruktor klasy nie został jeszcze zaimplementowany.

```
// Write tests for Matrix constructor -> FAIL (not implemented)
public class MatrixTest {

  @Test
  public void testConstructor() {
    Matrix actual = new Matrix(new double[]{1, 2, 0.1, -2}, 2, 2);
    double[][] expected = new double[][]{{1, 2}, {0.1, -2}};
    Assert.assertArrayEquals(actual.Value, expected);
  }
}
```

```
// Empty implementation of constructor
public class Matrix {
```

```
    public double[][] Value;
    public int rows, columns;

    public Matrix(double[] array, int rows, int columns) {

    }
}
```

Wynik testu: Negatywny

A więc zabieramy się za implementację konstruktora ustawiającego odpowiednie właściwości dla klas tj. liczbę wierszy, kolumn, oraz wartości będące elementami tablicy.

```
// Implementation of Matrix constructor
public Matrix(double[] array, int rows, int columns) {
    this.rows = rows;
    this.columns = columns;

    Value = new double[rows][columns];
    for (int i = 0; i < rows; i++) {
        System.arraycopy(array, i * columns, Value[i], 0, columns);
    }
}
```

Po ponownym uruchomieniu testu - przechodzi on w 100%.

Wynik testu: Pozytywny

4.2 Obsługa niepoprawnej liczby elementów przy tworzeniu macierzy.

Warto zwrócić uwagę na przypadek, gdy liczba podanych elementów będzie za mała, by wypełnić macierz. Gdy tak się stanie, zostanie ona wypełniona podanymi wartościami, a pozostałe niewypełnione pola będą miały wartość 0. Ta funkcjonalność obsługuje także możliwość braku tablicy wejściowej. Cała macierz jest wtedy zainicjalizowana zerami.

Tradycyjnie zatem piszemy test sprawdzający tą funckjonalność. Test kończy się niepowodzeniem.

```
// Test if the rest is filled with 0 -> FAIL (not implemented)
@Test
public void testTooShortArray() {
    Matrix actual = new Matrix(new double[]{1, 2, 0.1, -2}, 3, 3);

    double[][] expected = new double[][]{{1, 2, 0.1}, {-2, 0, 0}, {0, 0, 0}};
    Assert.assertArrayEquals(actual.Value, expected);
}
```

Wynik testu: Negatywny

Po zaimplementowaniu tej funkcjonalności (edycja konstruktora), program zachowuje się

zgodnie z założonymi oczekiwaniami.

```
// Constructor with filling functionality
public Matrix(double[] array, int rows, int columns) {
  this.rows = rows;
  this.columns = columns;
  double[] longerArray = new double[rows * columns];
  System.arraycopy(array, 0, longerArray, 0, array.length);

  Value = new double[rows][columns];
  for (int i = 0; i < rows; i++) {
    System.arraycopy(longerArray, i * columns, Value[i], 0, columns);
  }
}
```

Wynik testu: Pozytywny

4.3 Tworzenie macierzy jednostkowej.

Kolejny etap to stworzenie macierzy jednostkowej.

Testy konstruktora i pusta implementacja:

```
@Test
public void testIdentity() {
  Matrix actual = new Matrix(3);

  double[][] expected = new double[][]{{1, 0, 0}, {0, 1, 0}, {0, 0, 1}};
  Assert.assertArrayEquals(actual.Value, expected);
}
```

```
// Empty implementation of Identity Matrix creation
public Matrix(int dimension) {
}
```

Wynik testu: Negatywny

Po zaimplementowaniu tej funkcjonalności (edycja konstruktora), program zachowuje się zgodnie z założonymi oczekiwaniami.

```
// Implementation of Identity Matrix creation
public Matrix(int dimension) {
  this.rows = this.columns = dimension;
  Value = new double[dimension][dimension];

  for (int i = 0; i < dimension; i++) {
    Value[i][i] = 1;
  }
}
```

Wynik testu: Pozytywny

4.4 Lekka refaktoryzacja.

Do programu zostały wprowadzone lekkie zmiany w strukturze, a mianowicie kreowanie macierzy jednostkowej jest od teraz dostępne tylko i wyłącznie ze statycznej metody **Identity**. Sam konstruktor został zmieniony na **private** by do niego miała dostęp jedynie powyższa metoda.

```
// Implementation as before
private Matrix(int dimension) {
  this.rows = this.columns = dimension;
  Value = new double[dimension][dimension];

  for (int i = 0; i < dimension; i++) {
    Value[i][i] = 1;
  }
}

// Interface for the user
public static Matrix Identity(int dimension) {
  return new Matrix(dimension);
}
```

Dodatkowo została zmieniona kolejność przyjmowanych elementów w konstruktorze przyjmującym wiele elementów (tablica jest podawana na końcu dla zwiększenia czytelności).

```
public Matrix(int rows, int columns, double[] array) {
  // as before
  ...
}
```

Jak widzimy, refaktoryzacja nie miała żadnego wpływu na wynik testów:

```
@Test
public void testIdentity() {
  Matrix actual = Matrix.Identity(3);

  double[][] expected = new double[][]{{1, 0, 0}, {0, 1, 0}, {0, 0, 1}};
  Assert.assertArrayEquals(actual.Value, expected);
}
```

Wynik testu: Pozytywny

4.5 Dodawanie dwóch macierzy.

Kolejną funkcjonalność stanowi dodawanie dwóch macierzy. Muszą one mieć te same rozmiary, jednak tą funkcjonalnością zajmiemy się później. Póki co, zbadamy poprawne dodawanie.

Testy metody AddMatrices i pusta implementacja:

```
@Test
public void testAddMatrices() {
  Matrix m1 = new Matrix(2, 2, new double[]{1, 2, 0.1, -2});
```

```
    Matrix m2 = new Matrix(2, 2, new double[]{2, 3, 4, 0.1});

    Matrix result = Matrix.AddMatrices(m1, m2);

    double[][] expected = new double[][]{{3, 5}, {4.1, -1.9}};
    Assert.assertArrayEquals(expected, result.Value);
  }
```

```
  // Empty implementation
  public static Matrix AddMatrices(Matrix m1, Matrix m2) {
    throw new NotImplementedException();
  }
```

Wynik testu: Negatywny

Po zaimplementowaniu tej funkcjonalności, program zachowuje się zgodnie z oczekiwaniami.

```
  // Implementation of adding 2 Matrices funcionality
  public static Matrix AddMatrices(Matrix m1, Matrix m2) {
    Matrix result = new Matrix(m1.rows, m1.columns, new double[]{});
    for (int i = 0; i < m1.rows; i++) {
      for (int j = 0; j < m1.columns; j++) {
        result.Value[i][j] = m1.Value[i][j] + m2.Value[i][j];
      }
    }
    return result;
  }
```

Wynik testu: Pozytywny

4.6 Dodawanie dwóch macierzy - rozmiary macierzy muszą być identyczne.

Zgodnie z obietnicą nadszedł czas na implementację obsługi błędnych rozmiarów macierzy. W sytuacji gdy rozmiary te nie są sobie równe, rzucany jest wyjątek **RuntimeException**.

Zaczynamy od testów. Problem polega na tym, że wartości zostaną przekopiowane prawidłowo dla macierzy [2][2], a reszta wartości jest zwyczajnie pominięta:

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

  @Test
  public void testAddMatricesWhenWrongSizesThenRuntimeException() {
    exception.expect(RuntimeException.class);

    Matrix m1 = new Matrix(2, 2, new double[]{2, 3, 4, 0.1});
    Matrix m2 = new Matrix(3, 2, new double[]{2, 2, 3, 2, 0.1, -2});

    Matrix result = Matrix.AddMatrices(m1, m2);
  }
```

Wynik testu: Negatywny - nie rzucono wyjątku RuntimeException.

Po zaimplementowaniu tej funkcjonalności, program zachowuje się zgodnie z oczekiwaniami.

```
// Complete implementation
public static Matrix AddMatrices(Matrix m1, Matrix m2) {
  if (m1.rows != m2.rows || m1.columns != m2.columns) {
    throw new RuntimeException("Sizes of given Matrices don't match.");
  }

  Matrix result = new Matrix(m1.rows, m1.columns, new double[]{});
  for (int i = 0; i < m1.rows; i++) {
    for (int j = 0; j < m1.columns; j++) {
      result.Value[i][j] = m1.Value[i][j] + m2.Value[i][j];
    }
  }
  return result;
}
```

Wynik testu: Pozytywny

4.7 Mnożenie macierzy przez skalar.

Kolejną funkcjonalność stanowi mnożenie macierzy przrz skalar. Nie ma tu żadnych ograniczeń ani specjalnych przypadków więc bezpośrednio zabieramy się do implementacji.

Oczywiście najpierw piszemy testy i pustą implementację:

```
@Test
public void testMultiplyMatrixByScalar() {
  Matrix matrix = new Matrix(2, 2, new double[]{5, 1, 0, 4});
  double scalar = 2.1;

  Matrix result = Matrix.MultiplyMatrixByScalar(matrix, scalar);

  double[][] expected = new double[][]{{10.5, 2.1}, {0, 8.4}};
  Assert.assertArrayEquals(expected, result.Value);
}
```

```
// Empty implementation
public static Matrix MultiplyMatrixByScalar(Matrix matrix, double scalar) {
  throw new NotImplementedException();
}
```

Wynik testu: Negatywny

Po zaimplementowaniu tej funkcjonalności, program zachowuje się zgodnie z oczekiwaniami.

```
// Complete implementation
public static Matrix MultiplyMatrixByScalar(Matrix m, double scalar) {
  double[] array = new double[m.rows * m.columns];

  for (int i = 0; i < m.rows; i++) {
    for (int j = 0; j < m.columns; j++) {
```

```
            array[i * m.rows + j] = m.Value[i][j] * scalar;
        }
    }

    return new Matrix(m.rows, m.columns, array);
}
```

Wynik testu: Pozytywny

4.8 Obliczanie wyznacznika zadanej macierzy kwadratowej.

Podstawową operacją wykonywaną na macierzach kwadratowych jest obliczenie wyznacznika macierzy. Zadanie to jest dość skomplikowane, warto więc starannie przygotować testy. Przypadkiem macierzy niekwadratowej zajmiemy się później.

```
@Test
public void testDeterminant() {
    Matrix matrix = new Matrix(4, 4, new double[]{
        1, 5, 6, 2.2,
        3.3, 9, 10, 1,
        7, 9, 3.2, 5.1,
        5, 8, 6.3, 2
    });

    double determinant = Matrix.Determinant(matrix);
    double expected = 60.729;
    Assert.assertEquals(expected, determinant, 0.001);
}
```

```
// Empty implementation
public static double Determinant(Matrix matrix) {
    throw new NotImplementedException();
}
```

Wynik testu: Negatywny

Do obliczeń posługujemy się klasą pomocniczą **DeterminantHelper**, która rekurencyjnie oblicza wyznacznik według rozwinięcia Laplace'a. Użytkownik jednak korzysta z metody **Determinant**. Ułatwi to w przyszłości obsługę błędów. Po zaimplementowaniu tej funkcjonalności, program zachowuje się zgodnie z oczekiwaniami.

```
// Method with recursive calculations
private static double DeterminantHelper(double[][] matrix) {
    double sum = 0;
    int sign;

    // if a matrix is a number - return that number
    if (matrix.length == 1) {
        return matrix[0][0];
    }

    for (int i = 0; i < matrix.length; i++) {
        //create smaller matrix - with values not in same row, column
```

```
      double[][] smaller = new double[matrix.length - 1][matrix.length - 1];
      for (int x = 1; x < matrix.length; ++x) {
        for (int y = 0; y < matrix.length; ++y) {
          if (y < i) {
            smaller[x - 1][y] = matrix[x][y];
          } else if (y > i) {
            smaller[x - 1][y - 1] = matrix[x][y];
          }
        }
      }
      if (i % 2 == 0) {
      sign = 1;
    } else {
      sign = -1;
    }
    sum += sign * matrix[0][i] * (DeterminantHelper(smaller)); // recursive
  }
  return sum;
  }
```

```
  // Interface for the user
  public static double Determinant(Matrix matrix) {
    return DeterminantHelper(matrix.Value);
  }
```

Wynik testu: Pozytywny

4.9 Błąd obliczania wyznacznika zadanej macierzy niekwadratowej.

Warto zwrócić uwagę, że obliczanie wyznacznika jest możliwe tylko dla macierzy kwadratowych. Chcemy zatem mieć obsługę błędów, gdy użytkownik poda na wejściu macierz, która nie jest macierzą kwadratową. W takim przypadku rzucany jest wyjątek **RuntimeException** ze stosowną informacją o błędzie.

```
  // przypomnienie
  @Rule
  public ExpectedException exception = ExpectedException.none();

  @Test
  public void testDeterminantWhenWrongMatrixSizeThrowRuntimeException() {
    exception.expect(RuntimeException.class);

    Matrix matrix = new Matrix(3, 4, new double[]{
      1, 5, 6, 2.2,
      3.3, 9, 10, 1,
      7, 9, 3.2, 5.1,});

    double determinant = Matrix.Determinant(matrix);
  }
```

Wynik testu: Negatywny - nie rzucono błędu RuntimeException

Po zaimplementowaniu tej funkcjonalności, program zachowuje się zgodnie z oczekiwaniami.

```
// Error handling implementation
public static double Determinant(Matrix matrix) {
      if (matrix.rows != matrix.columns) {
          throw new RuntimeException("Row number must be equal to Column
              number");
      }
      return DeterminantHelper(matrix.Value);
 }
```

Wynik testu: Pozytywny

5 Podsumowanie.