Inżynieria oprogramowania

LAB 2 Warunki poprawności – asercje i testy jednostkowe

ASERCJA

Asercja to fragment kodu sprawdzający wewnętrzny stan programu i przerywający jego działanie w przypadku błędu.

ASERCJE A WYJĄTKI

Po co dodatkowy mechanizm do wykrywania błędów skoro mamy wyjątki?

Wyjątki są wykorzystywane do obsługi sytuacji mało prawdopodobnych, lecz możliwych i koniecznych do obsłużenia (np. brak pamięci, zerwane połączenie sieciowe, błędny format liczby, dzielenie przez zero, itp.).
Asercje natomiast definiują warunki poprawności, które zawsze muszą być poprawne (np. wynik funkcji obliczającej długość odcinka musi być większa od 0, parametr zawierający pole kwadratu musi być dodatnie, funkcja sortująca zwraca posortowane dane, itp.). Jeżeli te warunki będą fałszywe, to ewidentnie występuje błąd w programie.

KIEDY I GDZIE KORZYSTAĆ Z ASERCJI?

Wszędzie, gdziekolwiek przypuszczamy, że stan programu powinien być taki, jakiego oczekujemy, lecz nie jest to oczywiste z poprzedzających instrukcji, powinno się użyć asercji, aby to sprawdzić.

Najważniejsze miejsca:

- Na początku każdej procedury (aby sprawdzić poprawność parametrów wejściowych)
- Na końcu każdej procedury (aby sprawdzić poprawność generowanych wyników)
- Na początku każdej pętli (aby sprawdzić niezmienniki pętli)
- Na końcu każdej pętli (sprawdzenie wyników działania pętli)
- Przed użyciem zmiennej (sprawdzenie, czy nie jest równa null)

Z asercji możemy korzystać w dwojaki sposób:

- Sprawdzanie poprawności stanu przed i po wykonaniu operacji
- Duplikowanie kodu (np. napisanie skomplikowanego algorytmu w prostszy sposób), uruchomienie dwóch fragmentów i porównanie wyników

ASERCJE W JAVIE

Instrukcja ***assert*** przyjmuje dwie formy:

Pierwsza (prostsza):
assert *Expression*$_1$;
gdzie ***Expression***$_1$ jest wyrażeniem logicznym.
W momencie uruchomienia asercji obliczane jest wyrażenie ***Expression***$_1$ i w przypadku wartości *false* rzucany jest wyjątek **AssertionError** bez szczegółowego komunikatu.

Druga postać:
assert *Expression*$_1$: *Expression*$_2$;
gdzie:

- ***Expression***$_1$ jest wyrażeniem logicznym,
- ***Expression***$_2$ zawiera komunikat szczegółowy, który zostanie przekazany do wyjątku i wyświetlony na ekranie.

WŁĄCZENIE OPCJI URUCHAMIANIA ASERCJI

Aby uruchomić tę opcję należy wywoływać kompilator z parametrem **-ea.**

W **NetBeans** wystarczy kliknąć prawym przyciskiem myszy na projekcie i wybrać opcję **SetConfiguration -> Customize**. Następnie w okienku uruchomieniowym w rubryce **VMOptions** wpisać wyrażenia **-ea**. Od tej pory przy kompilacji asercje będą brane pod uwagę.

ZAD.1.

Przepisz poniższy program i prześledź jego działanie. Porównaj wynik działania programu dla wyłączonej i włączonej opcji uruchamiania asercji.

```
package asercjetest;
public class AsercjeTest {
    public static void main(String[] args) {
        System.out.println(dziel(2, 0));
    }
    public static int dziel(int a, int b) {
        int licznik = a;
        int dzielnik = b; // To jest zabronione, gdyz b=0
        int wynik;

        // asercja przed sytuacją niebezpieczną
        assert dzielnik != 0 : "Pamietaj! NIE DZIEL PRZEZ ZERO !!!";
        wynik = licznik / dzielnik;
```

```
        return  wynik;
    }
}
```

ZAD.2.

Utwórz klasę StrangeSet, której szkielet wraz z komentarzem wygląda następująco:

```
package asercje;

class StrangeSet {

    public static int MAX_SIZE = 1000;
    protected int[] set = new int[MAX_SIZE];
    protected int size;

    /**
     * Dodaje liczbę k do zbioru liczb. Jeżeli podana liczba już
     * istnieje dodawana jest po raz drugi
     * @param k liczba, którą należy dodać od zbioru
     * @throws Exception występuje w przypadku przepełnienia tablicy
     */
    public void add(int k) throws Exception {
    }

    /**
     * Usuwa liczbę k ze zbioru liczb. W przypadku gdy zbiór nie
     * posiada liczby podanej jako parametr rzucany jest wyjątek.
     * @param k liczba do usunięcia
     * @throws Exception w przypadku gdy zbiór nie posiada danej
     * liczby
     */
    public void remove(int k) throws Exception {
    }

    /**
     * Losuje jedną liczbę ze zbioru liczb oraz usuwa ją ze zbioru.
     * @return wylosowana liczba
     * @throws Exception występuje w przypadku pustego zbioru
     */
    public int drawAtRandom() throws Exception {
        return 0;
    }

    /**
     * Zwraca sumę wszystkich liczb ze zbioru.
     * @return Suma liczb. W przypadku pustego zbioru wynosi 0.
     */
```

```
    public int getSumOfElements() {
        return 0;
    }

    /**
     * Dzieli każdy element ze zbioru przez n bez reszty.
     * @param n liczba przez którą będzie wykonane dzielenie.
     */
    public void divideAllElementsBy(int n) {
    }

    /**
     * Sprawdza, czy w zbiorze istnieje element k
     * @param k element do sprawdzenia
     * @return true w przypadku odnalezienia elementu, false
     * w przeciwnym przypadku
     */
    public boolean contains(int k) {
        return false;
    }

    /**
     * Zwraca rozmiar zbioru, czyli faktycznie dodanych liczbę
     * elementów
     * @return rozmiar zbioru
     */
    public int getSize() {
        return 0;
    }
}
```

Do każdej funkcji należy wymyślić po kilka asercji (min. 2), które będą sprawdzać poprawność programu. Następnie stworzyć klasę testującą, która będzie wykonywać pewne operacje na zbiorze liczb (w ten sposób kod asercji będzie uruchamiany).

TESTY JEDNOSTKOWE

Test jednostkowy (ang. *unit test*) – w programowaniu obiektowym jest metodą testowania tworzonego oprogramowania poprzez wykonywanie testów weryfikujących poprawność działania pojedynczych elementów np. metod lub klas. Testowany fragment programu poddawany jest testowi, który wykonuje go i porównuje wynik (np. zwrócone wartości, stan obiektu, wyrzucone wyjątki) z oczekiwanymi wynikami.
Zaletą testów jednostkowych jest możliwość wykonywania na bieżąco w pełni zautomatyzowanych testów na modyfikowanych elementach programu, co umożliwia często wychwycenie błędu natychmiast po jego pojawieniu się i szybką jego lokalizację zanim dojdzie do wprowadzenia błędnego fragmentu do programu. W praktyce oznacza to, że po dokonaniu zmian w wybranych elementach systemu nie musimy uruchamiać całego systemu w celu weryfikacji poprawności dokonanych modyfikacji – wystarczy wykonać testy jednostkowe dla elementów, w których dokonano zmian.

JUNIT – testy jednostkowe w Javie

JUnit został stworzony przez dwójkę programistów - Ericha Gamma i Kenta Becka. Jest to narzędzie wspomagające pisanie i przeprowadzanie testów dla programów pisanych w Javie. Udostępnia on między innymi prosty interfejs graficzny, dzięki któremu możemy uruchamiać przygotowane testy. JUnit pozwala nam:

- Stworzyć test sprawdzający, czy wyniki działania metod spełniają wymagane zależności
- Organizować testy w spójne zestawy, testujące określoną część funkcjonalności projektu, lub wybrane klasy
- Uruchamiać napisane testy zarówno przy użyciu interfejsu graficznego jak i z linii poleceń.

ADNOTACJE

- *@Test* – występuję przed każdą metodą testową,
- *@BeforeClass* – występuje przed metodą, która jest wykonywana przed uruchomieniem testów. Można w niej utworzyć obiekt klasy testowej, zainicjalizować jej atrybuty itp.
- *@AfterClass* – oznacza metodę, która ma być wywołana po zakończeniu wszystkich testów. Można w niej np. zamknąć połączenie z bazą danych, zapisać plik lub po prostu wypisać na konsoli „Hura" lub coś innego co jest niezbędne po wykonaniu testów.
- *@Before* – metoda oznaczona tą adnotacją, będzie wywoływana przed wykonaniem każdej metody testowej.
- *@After* – oznacza metodę, która ma być wywoływana po każdej metodzie testowej.

TESTY JEDNOSTKOWE W NETBEANS

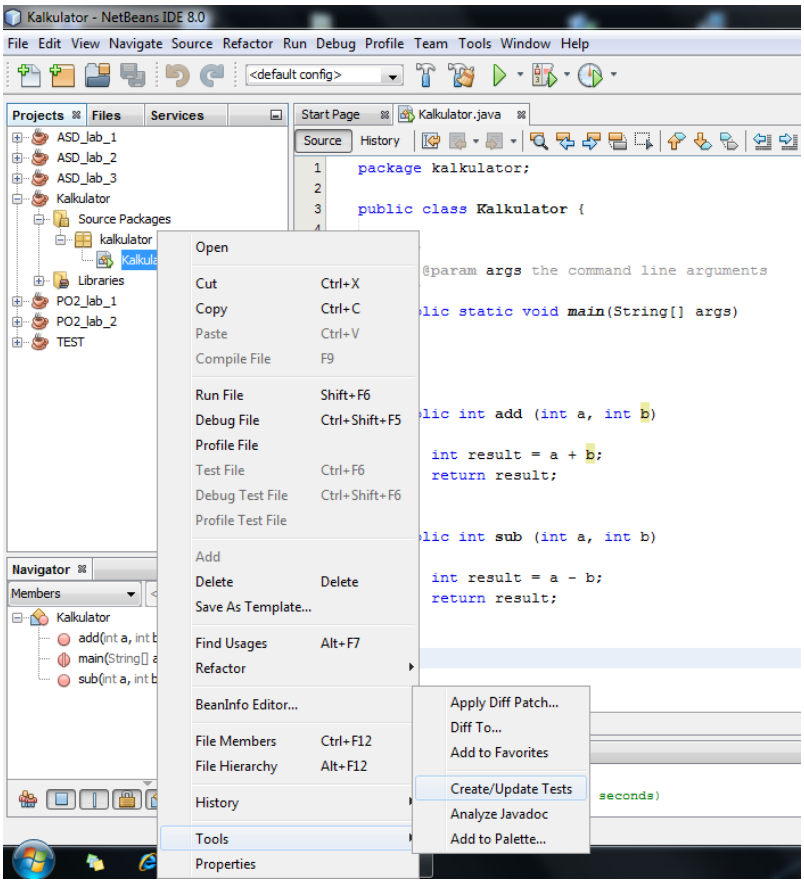
TWORZENIE TESTÓW JEDNOSTKOWYCH

Tworzymy klasę, dla której utworzone zostaną testy (zob. rys.1).

```java
package kalkulator;

public class Kalkulator {

    /**
     * @param args the command line arguments
     */
    public static void main(String[] args)
    {

    }

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

    public int sub (int a, int b)
    {
        int result = a - b;
        return result;
    }
}
```

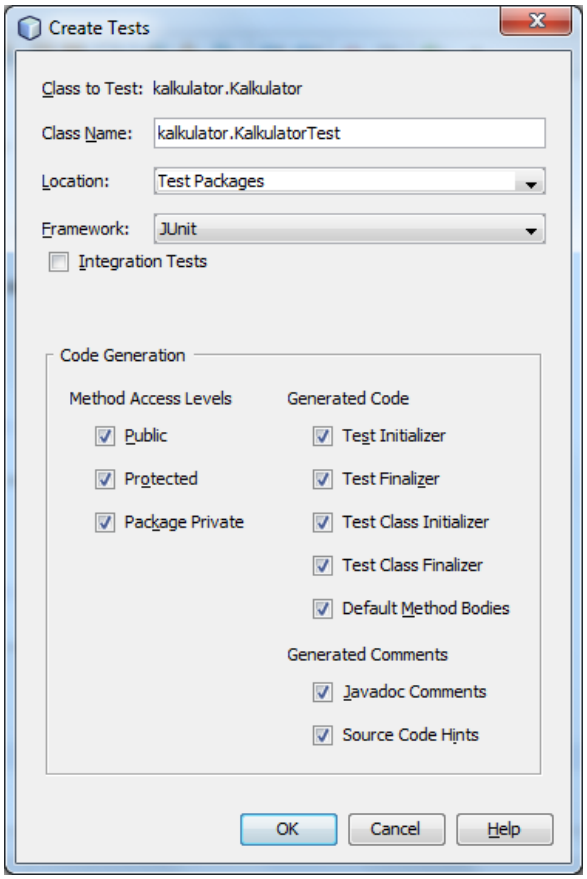
Rys.2. Klasa, dla której utworzone zostaną testy

Klikamy PPM na klasę, dla której chcemy utworzyć testy i wybieramy: Tools >> Create/Update Tests (zob. rys.2).



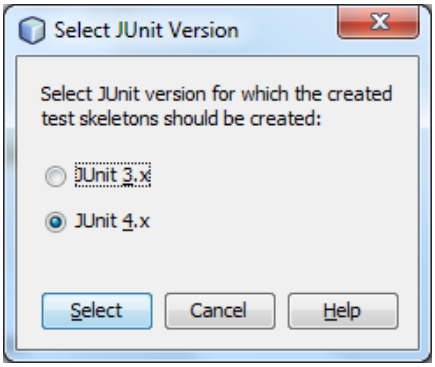
Rys.2. Tworzenie testów jednostkowych

Wprowadzamy nazwę klasy testowej (zaleca się pozostawienie nazwy domyślnej), wybieramy poziom testów (metody publiczne, chronione, prywatne) oraz testy (kod) i komentarze do wygenerowania (zob. rys.3.).



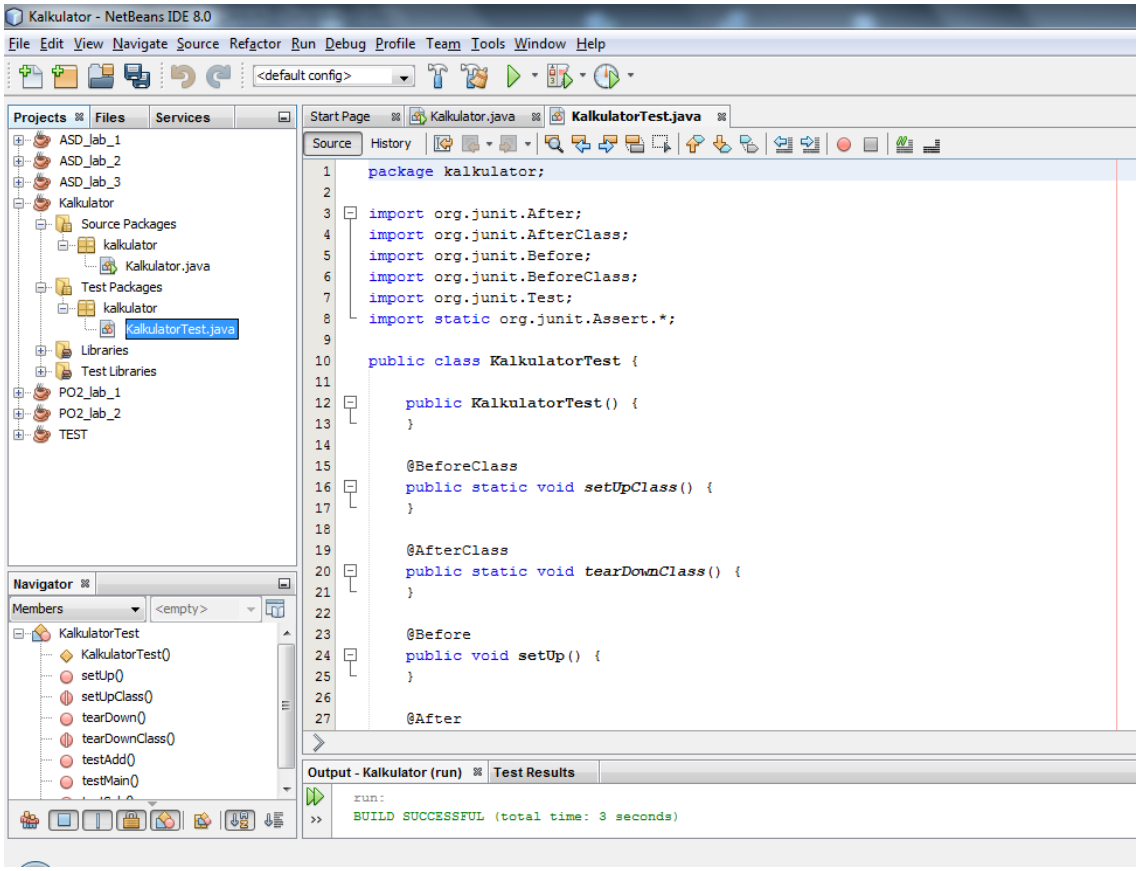
Rys.3. Tworzenie testów jednostkowych - kreator

Wybieramy wersje testów JUnit (zob. rys.4).



Rys.4. Tworzenie testów jednostkowych – wybór wersji JUnit

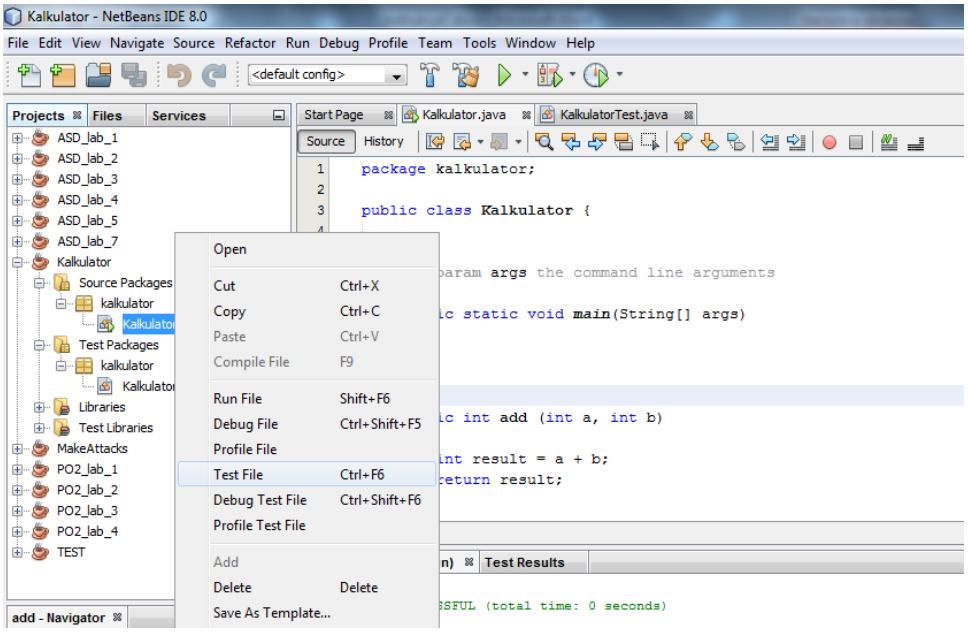
W wyniku do projektu zostają dodane pakiety testowe (Test Packages) oraz wygenerowana zostaje klasa testowa (zob. rys.5).



Rys.5. Tworzenie testów jednostkowych – rezultat

URUCHAMIANIE TESTÓW

W celu uruchomienia testów jednostkowych dla naszej klasy klikamy PPM na klasę, dla której chcemy uruchomić testy i wybieramy: Test File (zob. rys.6).



Rys.6. Tworzenie testów jednostkowych – rezultat

ZAD.3.

Przepisz poniższy program i prześledź jego działanie. Wygeneruj dla niego testy jednostkowe i wykonaj je.

```
package kalkulator;

public class Kalkulator {
    public static void main(String[] args)
    {

    }
    public int add (int a, int b)
    {
        int result = a + b;
        return result;
    }
    public int sub (int a, int b)
    {
        int result = a - b;
        return result;
    }
}
```

ZAD.4.

Zmodyfikuj kod wygenerowanej klasy testowej w następujący sposób:

```
package kalkulator;

import org.junit.After;
import org.junit.AfterClass;
import org.junit.Before;
import org.junit.BeforeClass;
import org.junit.Test;
import static org.junit.Assert.*;

public class KalkulatorTest {

    public static int i;

    public KalkulatorTest() {
    }

    @BeforeClass
    public static void setUpClass() {
        System.out.println(">> TEST START >>");
```

```
        i = 1;
    }

    @AfterClass
    public static void tearDownClass() {
        System.out.println(">> TEST STOP >>");
    }

    @Before
    public void setUp() {
        System.out.println("\tTest metody nr: " + i);
    }

    @After
    public void tearDown() {
        System.out.println("\tKoniec testu metody nr: " + i);
        i++;
    }

    /**
     * Test of main method, of class Kalkulator.
     */
    @Test
    public void testMain() {
        System.out.println("\t\tMetoda: main");
    }

    /**
     * Test of add method, of class Kalkulator.
     */
    @Test
    public void testAdd() {
        System.out.println("\t\tMetoda: add");
        for(int j=-100; j<100; j++)
        {
            for(int k=-100; k<100; k++)
            {
                Kalkulator instance = new Kalkulator();
                int result = instance.add(j, k);
                int expResult = j+k;
                assertEquals("Blad   przy   dodawaniu",   expResult,
result);
            }
        }
        System.out.println("\t\t\tWszystko OK");
    }

    /**
```

```
     * Test of sub method, of class Kalkulator.
     */
    @Test
    public void testSub() {
        System.out.println("\t\tMetoda: sub");
        for(int j=-100; j<100; j++)
        {
            for(int k=-100; k<100; k++)
            {
                Kalkulator instance = new Kalkulator();
                int result = instance.sub(j, k);
                int expResult = j-k;
                assertEquals("Blad  przy  odejmowaniu",  expResult,
result);
            }
        }
        System.out.println("\t\t\tWszystko OK");
    }

}
```

ZAD.5.

Dołącz do klasy Kalkulator następujące metody:

- `multi(int a, int b)`, obliczającą iloczyn dwóch liczb całkowitych będących parametrami;
- `div(int a, int b)`, obliczającą iloraz dwóch liczb całkowitych będących parametrami;
- `pow(int a, int n)`, obliczającą potęgę liczby całkowitej *a* o wykładniku *n*;
- `fact(int a)`, obliczającą silnię liczby całkowitej *a*;
- `min(int[] tab)`, obliczającą wartość najmniejszą dla tablicy *tab*;
- `max(int[] tab)`,obliczającą wartość największą dla tablicy *tab*;
- `addMatrix(int[][] tab1, int[][] tab2)`, obliczającą sumę macierzy zapisanych w postaci tablic dwuwymiarowych;
- `prodMatrix (int[][] tab, int n)`, obliczającą iloczyn macierzy zapisanej w postaci tablicy dwuwymiarowej przez liczbę całkowitą *n*;

Dla utworzonych metod napisz i wykonaj testy jednostkowe (min. 2 testy dla każdej z metod).