**Ministerul Educației și Cercetării al Republicii Moldova**

**Universitatea Tehnică a Moldovei**

**Facultatea Calculatoare, Informatică şi Microelectronică**

**Departamentul Informatică şi Ingineria Sistemelor**

**RAPORT**

Lucrare de laborator nr.5

la cursul „Testarea Produselor Program”

Student : TI-214 Reguș Ruslan

Coordonator: asist. univ. Crîjanovschi Adriana

**Chișinău 2024**

Cuprins

[Testara unitară 2](#_Toc11888)

[SARCINA 1 2](#_Toc9183)

[SARCINA 2 5](#_Toc15920)

[SARCINA 3 6](#_Toc18515)

[SARCINA 4 7](#_Toc979)

[Concluzie 10](#_Toc21513)

# Testara unitară

Testara unitară este o metodă de testare a codului software în care fiecare componentă individuală (sau unitate) a programului este testată în mod izolat. Scopul principal al testării unitare este de a verifica dacă fiecare unitate de cod funcționează conform specificațiilor și de a detecta eventualele erori sau bug-uri înainte ca acestea să afecteze alte părți ale programului.

În timpul testării unitare, fiecare funcție, metodă sau clasă este testată separat, de obicei, prin furnizarea de date de intrare cunoscute și compararea rezultatelor obținute cu rezultatele așteptate. Aceasta implică adesea utilizarea de cadre de testare specializate sau scrierea de cod suplimentar pentru a simula comportamentul unităților de cod și pentru a automatiza procesul de testare.

Prin testarea unitară, dezvoltatorii pot identifica rapid și corecta erorile în codul lor, îmbunătățind calitatea și fiabilitatea software-ului înainte de a fi integrat în aplicații mai mari sau de a fi livrat clienților. De asemenea, testarea unitară facilitează refactorizarea și mentenanța codului, deoarece oferă un mecanism eficient de validare a modificărilor făcute în timpul dezvoltării.

# SARCINA 1

Creați un proiect cu clasa CustomMath.

Omiteți din metoda main a clasei CustomMath verificarea funcției sum.

Omiteți din metoda testSum apelul metodei fail. Asigurați-vă că testarea funcției sum trece

pentru datele de intrare curente.

Adăugați în raport codul funcției testSum și rezultatul testării (rezultatele testelor PrtSc al

ferestrei).

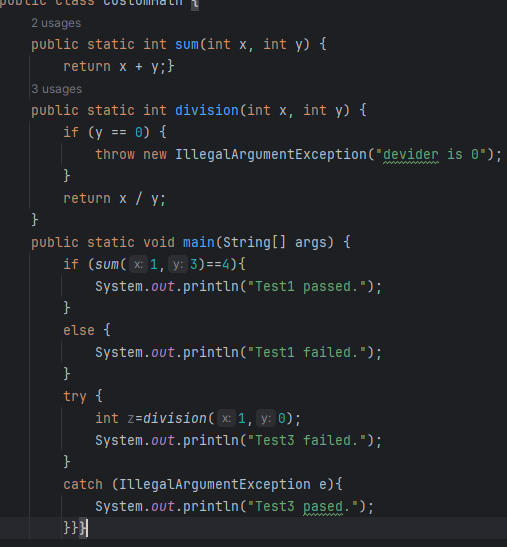


Figura 1 Codul Clasei CustomMath

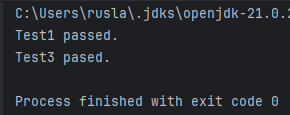


Figura 2 Rezultatul apelării clasei CustomMath



Figura 3 Codul clasei CustomMathTest

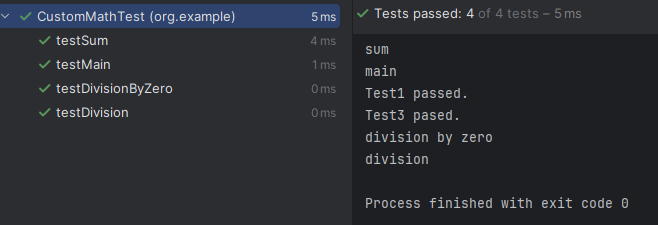


Figura 4 Rezultatul

SARCINA 2

Modificarea metodei de testare testDivisionByZero()

Cod sursă:

package org.example;

import org.junit.AfterClass;

import org.junit.BeforeClass;

import org.junit.Test;

import static org.junit.Assert.\*;

public class CustomMathTest {

@BeforeClass

public static void setUpClass() {

// Setup if needed

}

@AfterClass

public static void tearDownClass() {

// Teardown if needed

}

@Test

public void testSum() {

System.out.println("sum");

int x = 0;

int y = 0;

int expResult = 0;

int result = CustomMath.sum(x, y);

assertEquals(expResult, result);

}

@Test

public void testDivisionByZero() {

System.out.println("division");

int x = 10;

int y = 0;

try {

int result = CustomMath.division(x, y);

fail("Division by zero did not throw an exception");

} catch (IllegalArgumentException e) {

}

y = 2;

int expResult = 5;

int result = CustomMath.division(x, y);

assertEquals(expResult, result);

}

@Test

public void testMain() {

System.out.println("main");

String[] args = null;

CustomMath.main(args); }}

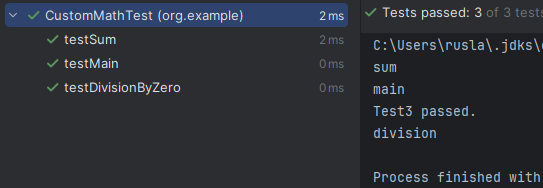


Figura 5 Rezultatul execuției sarcinii 2

# SARCINA 3

Modificați metoda de testare testDivisionByZero (), astfel încât funcția să verifice împărțirea la

zero, și de asemenea să furnizeze date de intrare corecte.

Cod sursă

package org.example;

import java.util.Arrays;

import java.util.Collection;

import org.junit.AfterClass;

import org.junit.BeforeClass;

import org.junit.Test;

import static org.junit.Assert.\*;

import org.junit.runner.RunWith;

import org.junit.runners.Parameterized;

import org.junit.runners.Parameterized.Parameters;

@RunWith(Parameterized.class)

public class CustomMathTest {

@Parameters

public static Collection<Object[]> divisionValues() {

return Arrays.asList(new Object[][] {

{10, 0, IllegalArgumentException.class}, // împărțire la 0

{10, 2, null} // împărțire normală

}); }

private int x;

private int y;

private Class<? extends Exception> expectedException;

public CustomMathTest(int x, int y, Class<? extends Exception> expectedException) {

this.x = x;

this.y = y;

this.expectedException = expectedException; }

@BeforeClass

public static void setUpClass() { // Setup if needed }

@AfterClass

public static void tearDownClass() {// Teardown if needed}

@Test

public void testSum() {

System.out.println("sum");

int x = 0;

int y = 0;

int expResult = 0;

int result = CustomMath.sum(x, y);

assertEquals(expResult, result);}

@Test

public void testDivisionByZero() {

System.out.println("division");

if (expectedException != null) {

try {

CustomMath.division(x, y);

fail("Expected exception: " + expectedException.getName());

} catch (Exception e) {

assertTrue(expectedException.isInstance(e));

}

} else {

int result = CustomMath.division(x, y);

int expResult = x / y;

assertEquals(expResult, result); }}}

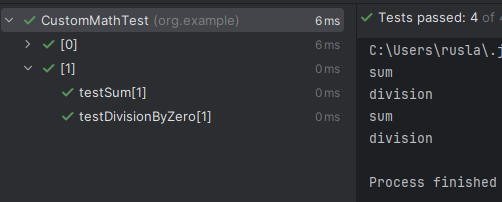


Figura 6 Rezultatul execuției sarcinii 3

# SARCINA 4

Extindeți clasa de testare, astfel încât să utilizeze metoda assertTrue și / sau assertFalse.

Cod sursă

package org.example;

import java.util.Arrays;

import java.util.Collection;

import org.junit.AfterClass;

import org.junit.BeforeClass;

import org.junit.Test;

import static org.junit.Assert.\*;

import org.junit.runner.RunWith;

import org.junit.runners.Parameterized;

import org.junit.runners.Parameterized.Parameters;

@RunWith(Parameterized.class)

public class CustomMathTest {

@Parameters

public static Collection<Object[]> divisionValues() {

return Arrays.asList(new Object[][]{

{10, 0, IllegalArgumentException.class}, // împărțire la 0

{10, 2, null} // împărțire normală

});}

private int x;

private int y;

private Class<? extends Exception> expectedException;

public CustomMathTest(int x, int y, Class<? extends Exception> expectedException) {

this.x = x;

this.y = y;

this.expectedException = expectedException; }

@BeforeClass

public static void setUpClass() { // Setup if needed

}

@AfterClass

public static void tearDownClass() { // Teardown if needed

}

@Test

public void testSum() {

System.out.println("sum");

int x = 0;

int y = 0;

int expResult = 0;

int result = CustomMath.sum(x, y);

assertEquals(expResult, result); }

@Test

public void testDivisionByZero() {

System.out.println("division");

if (expectedException != null) {

try {

CustomMath.division(x, y);

fail("Expected exception: " + expectedException.getName());

} catch (Exception e) {

assertTrue(expectedException.isInstance(e));

}

} else {

int result = CustomMath.division(x, y);

int expResult = x / y;

assertEquals(expResult, result); } }

@Test

public void testIsPositive() {

System.out.println("testIsPositive");

int num = 5;

assertTrue(num > 0); }

@Test

public void testIsNegative() {

System.out.println("testIsNegative");

int num = -5;

assertFalse(num > 0); }}

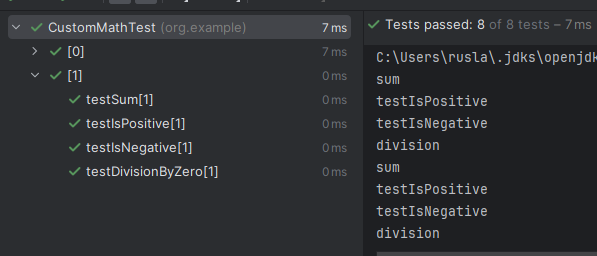


Figura 7 Rezultatul execuției sarcinii 4

# Concluzie

Prin realizarea sarcinilor din laboratorul de testare unitară am putut să înțelegem și să aplicăm practic principiile și tehnicile esențiale ale testării unitare în Java, folosind biblioteca JUnit. Iată principalele concluzii:

Importanța testării unitare: Testarea unitară este un aspect crucial al dezvoltării software care ne permite să verificăm corectitudinea și comportamentul fiecărei componente individuale a aplicației noastre.

JUnit: Biblioteca JUnit oferă un cadru robust pentru scrierea și rularea testelor unitare în Java. Cu ajutorul său, putem organiza și executa teste pentru a verifica funcționalitățile clasei noastre.

Structură și organizare: Am învățat cum să organizăm și să structurăm clasele de testare pentru a acoperi diferitele funcționalități ale codului nostru. Am aplicat metode precum testele parametrice și folosirea anotărilor pentru a simplifica procesul de testare.

Gestionarea excepțiilor: Testarea corectă a cazurilor de excepții este esențială. Am învățat cum să tratăm și să verificăm excepțiile generate de metodele testate, asigurându-ne că aplicația noastră se comportă corect în diverse scenarii.

Utilizarea metodelor assertTrue și assertFalse: Aceste metode sunt utile pentru a verifica expresii booleane în timpul testării. Ele adaugă flexibilitate și acoperire extinsă pentru testele noastre unitare.

Prin completarea și îndeplinirea sarcinilor din acest laborator, am consolidat înțelegerea noastră asupra testării unitare și am dobândit abilități practice în implementarea și utilizarea acestor teste în proiectele noastre Java.