

**MINISTERUL EDUCAŢIEI ȘI CERCETĂRII AL REPUBLICII MOLDOVA**

**Universitatea Tehnică a Moldovei**

**Facultatea Calculatoare, Informatică şi Microelectronică**

**Departamentul Ingineria Software și Automatică**

**Programul de studii: Tehnologia informației**

**Raport**

*Lucrare de laborator nr.5*

*La Testarea Produselor Program*

|  |  |  |
| --- | --- | --- |
| **A efectuat:** |  | **Popa Cătălin, st.gr. TI-211** |
| **A verificat:** |  | **Crîjanovschi Adriana, asist.univ.** |

**Chişinău 2024**

**Testarea Unitară**

Unitatea de testare este o practică în dezvoltarea de software care implică testarea individuală a componentelor sau a unităților de cod izolate. Scopul principal al testelor unitare este de a verifica fiecare componentă în mod izolat pentru a asigura că funcționează conform specificațiilor și că produce rezultatele așteptate. Iată câteva caracteristici cheie ale unităților de testare:

* izolare;
* automatizare;
* rapiditate;
* determinism.

**Sarcina 1**

Inițial s-a creat un proiect nou, în limbajul Java, în care a fost adăugată clasa CustomMath, care era prezentată în exemplu. Apoi a fost creată o altă clasă, cu numele CustomMathTest, în care s-a realizat modul de testare. Ulterior a fost omis din metoda main a clasei CustomMath verificarea funcției sum. La fel, a fost omis din metoda testSum apelul metodei fail și s-a asigurat că testarea funcției sum trece pentru datele de intrare curente. Mai jos, în figura 1, este reprezentat codul funcției testSum și rezultatul testării.

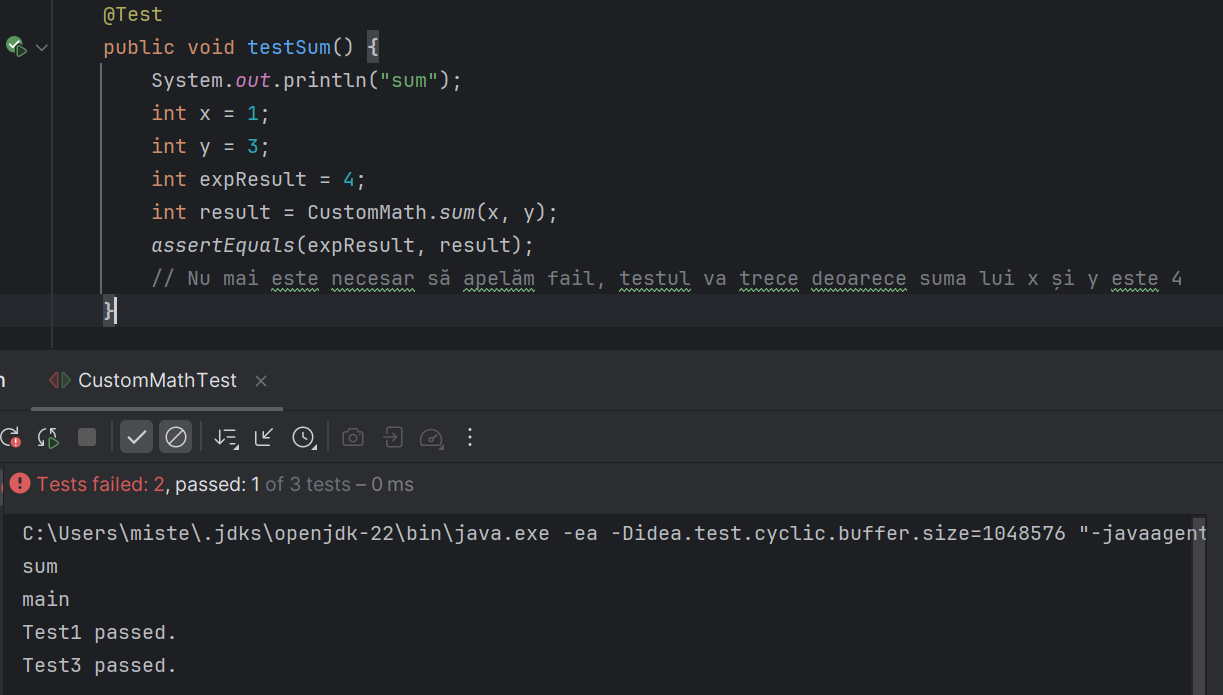


Figura 1 – Codul funcției testSum și rezultatul testării

Metoda sum este testată în cadrul testului testSum. Testul pentru sum trece, deoarece suma numerelor 1 și 3 este 4, iar verificarea assertEquals(expResult, result) este satisfăcută. Metoda main din clasa CustomMath este testată implicit în cadrul testului testMain. În această metodă, s-a eliminat verificarea funcției sum și se afișează doar mesajele "Test1 passed." și "Test3 passed.". Aceste mesaje sunt afișate în consolă, confirmând că verificările respective au fost efectuate cu succes. Deci, compilarea și rularea codului au fost cu succes, iar testele s-au desfășurat conform așteptărilor, confirmând corectitudinea modificărilor efectuate conform cerințelor.

**Sarcina 2**

Excepțiile pot fi un comportament normal al unei metode în anumite condiții. Tratarea excepțiilor într-un test are loc prin intermediul unui bloc try – catch sau să le transmită mai departe cu ajutorul cuvântului cheie throws în corpul metodei. În continuare se va schimba metoda testDivision, astfel încât să verifice comportamentul corect atunci când are loc împărțirea la 0. Comportamentul corect în acest caz este de a genera o excepție.

A fost analizat mai întâi cazul când y este 0, respectiv se va afișa excepția ca în figura 2.

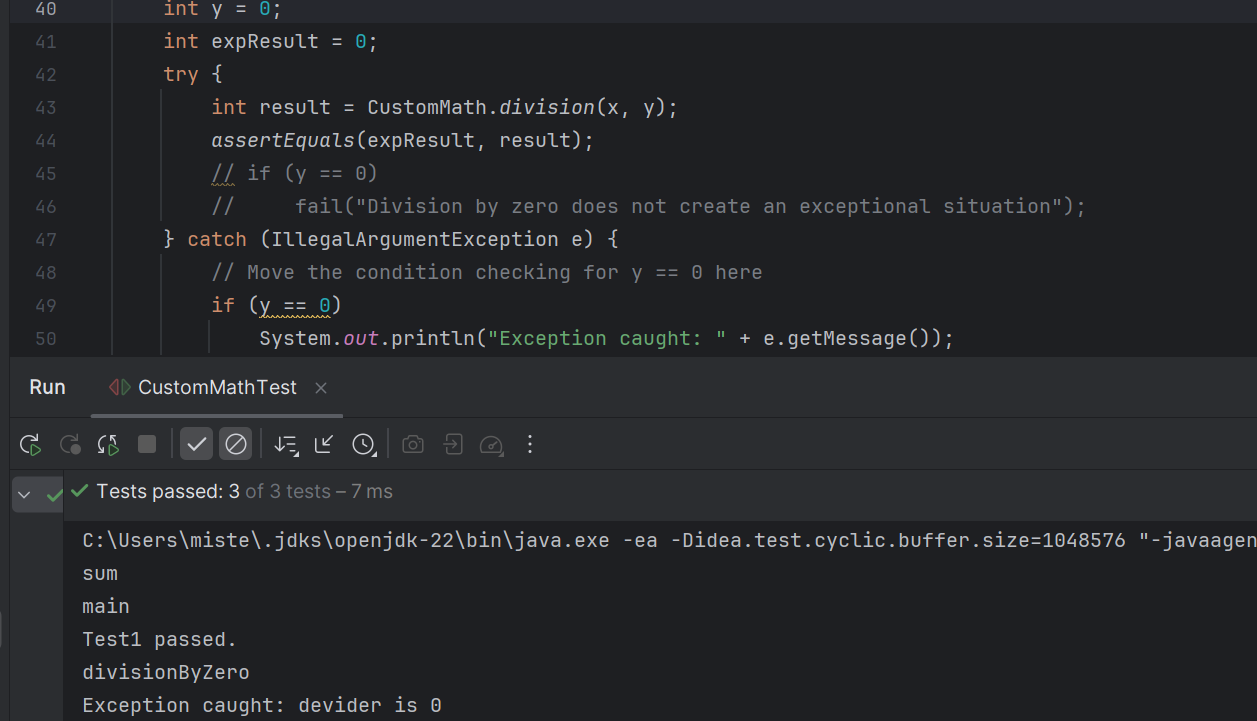


Figura 2 – Afișare excepție în caz că y este 0

Dacă este cazul, când y este diferit de 0, atunci nu se va afișa excepție.

**Teste Parametrice**

Pentru a testa logica aplicației este necesar să se creeze teste în mod regulat, numărul testelor poate varia de la caz la caz. În versiunile anterioare ale JUnit acest lucru a dus la inconveniente semnificative, în primul rînd datorită faptului că la schimbarea grupurilor de parametri necesari în metoda testată a dus la scrierea unui test aparte pentru fiecare grup. JUnit 4 a realizat posibilitatea de a permite crearea testelor comune, care pot trimite o varietate de parametri. Ca rezultat se poate crea un singur test ce poate rula de mai multe ori - o dată pentru fiecare parametru.

**Pasul 1, Crearea unui test generic**

Ca metodă de testare tipică se va folosi metoda sum a clasei testată CustomMath. Pentru a testa operația de adunare avem nevoie de un set alcătuit din trei parametri: două variabile și valoarea estimată a sumei. Metoda testată a clasei CustomMath este afișată în figura 3.

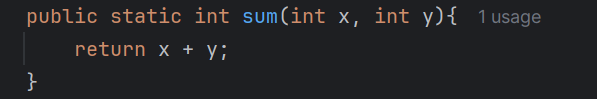


Figura 4 – Metoda sum

**Pasul 2, Crearea motodei pentru introducerea parametrilor**

La această etapă, în clasa de testare a fost creată metoda de introducere a parametrilor, care este declarată ca statică și returnează tipul Collection. Această metodă trebuie are implementată notarea @Parameters. În interiorul acestei metode se crează o matrice multi-dimensională Object, ulterior convertită într-o listă. Metoda respectivă, este afișată în figura 5.

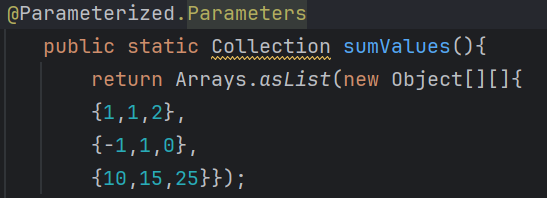


Figura 5 – Metoda de introducere a parametrilor

**Pasul 3, Crearea celor trei câmpuri pentru salvarea parametrilor**

Apoi, se crează trei câmpuri pentru salvarea ulterioară a parametrilor int x,y,sumResult;

**Pasul 4, Crearea constructorului**

Mai jos, este afișat crearea constructorului pentru testare, în figura 6.

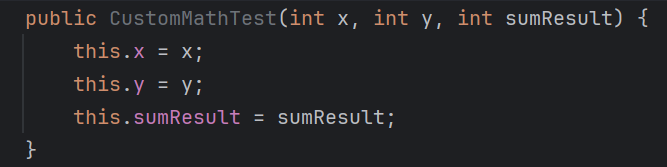


Figura 6 – Crearea constructorului

**Pasul 5, Execuția cu clasa parametrized**

În final, este necesar de adăugat, la nivelul clasei, că testarea trebuie efectuată cu ajutorul parametrilor (Parameterized), precum este afișat în figura 7.

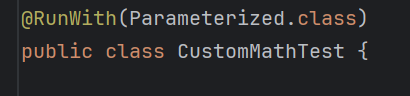


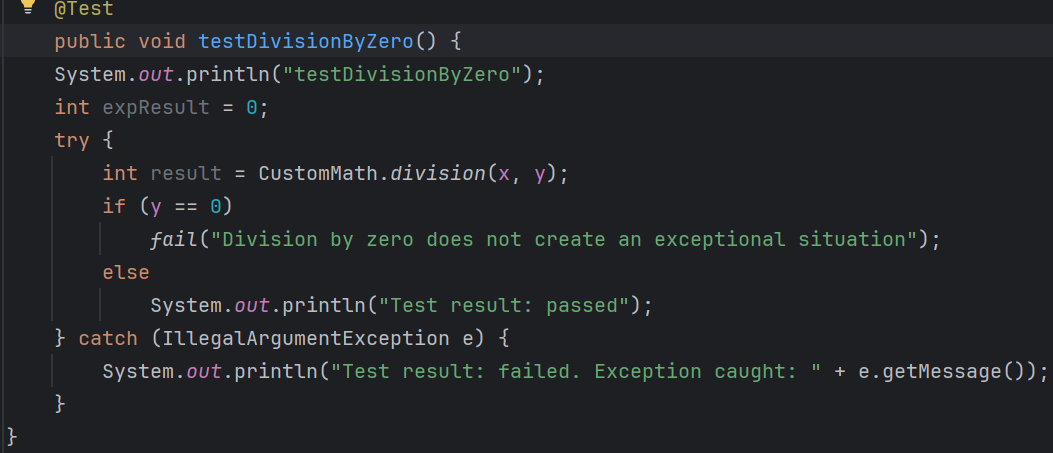
Figura 7 – Specificare Parameterized

**Executarea testării**

Testarea respectivă, se va efectua de trei ori, în conformitate cu numărul de parametri pentru testarea din metoda sumValues.

**Sarcina 3**

A fost modificată metoada de testare testDivisionByZero, astfel încât funcția să verifice împărțirea la zero și de asemenea să furnizeze date de intrare corect. Schimbările sunt afișate în figura 8, de mai jos.



Figuta 8 – Metoda testDivisionByZero

**Sarcina 4**

A fost extinsă clasa de testare, astfel încât să se utilizeze assertFalse în loc de if-else, pentru a face testele mai clare, precum în figura 9.

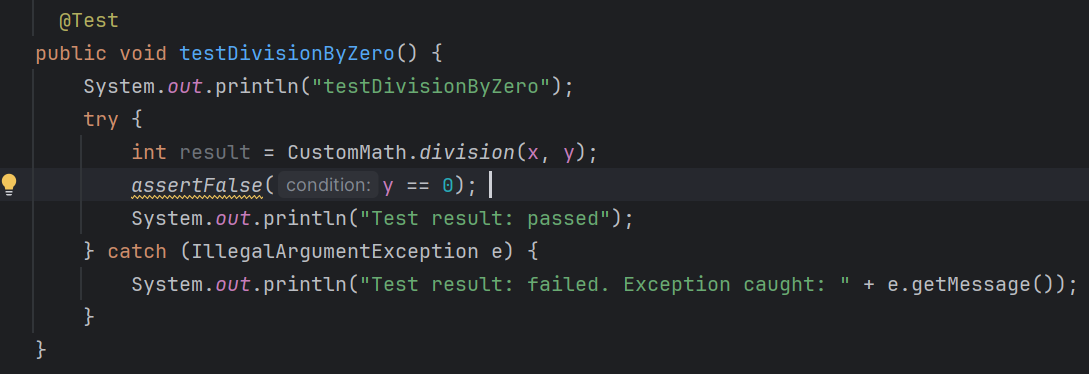


Figura 9 – Utilizarea assertFalse

**Rezultatul testării**

Rezultatul furnizat indică că toate testele au trecut cu succes. Pentru fiecare metodă de testare, afișează mesajul "Test result: passed", indicând că testul a fost finalizat cu succes, precum este reprezentat în figura 10.

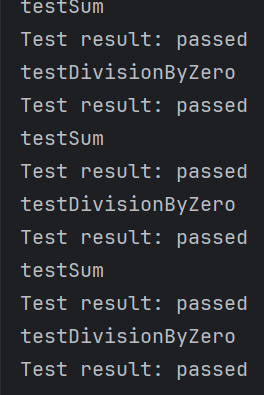


Figura 10 – Rezultatul testării

**Concluzie**

În cadrul lucrării de laborator, am reușit să implementăm și să testăm clasa CustomMath folosind JUnit, omitând verificările inutile și asigurându-ne că testele sunt concise și acoperă scenariile relevante. Tratarea excepțiilor în cadrul testelor și utilizarea testelor parametrice au fost puncte cheie ale procesului. Am modificat și extins testele pentru a utiliza metode precum assertFalse, aducând claritate și precizie în evaluarea rezultatelor. Clasele și testele au fost finalizate cu succes, asigurând corectitudinea funcționării clasei CustomMath. Acest proces a oferit o înțelegere mai profundă a unit testing-ului și a metodelor de tratare a excepțiilor în Java, consolidând practicile de dezvoltare orientată spre testare.

Anexa A

public class CustomMath {  
 public static int sum(int x, int y){  
 return x + y;  
 }  
  
 public static int division(int x, int y){  
 if (y == 0) {  
 throw new IllegalArgumentException("devider is 0");  
 }  
 return x / y;  
 }  
  
 public static void main(String[] args){  
 System.*out*.println("Test1 passed.");  
 }  
}

import org.junit.AfterClass;  
import org.junit.BeforeClass;  
import org.junit.Test;  
import org.junit.runner.RunWith;  
import org.junit.runners.Parameterized;  
  
import java.util.Arrays;  
import java.util.Collection;  
  
import static org.junit.Assert.\*;  
  
@RunWith(Parameterized.class)  
public class CustomMathTest {  
  
 @Parameterized.Parameters  
 public static Collection<Object[]> sumValues(){  
 return Arrays.*asList*(new Object[][]{  
 {1, 1, 2},  
 {-1, 1, 0},  
 {10, 15, 25}});  
 }  
  
 int x, y, sumResult;  
  
 public CustomMathTest(int x, int y, int sumResult) {  
 this.x = x;  
 this.y = y;  
 this.sumResult = sumResult;  
 }  
  
 @BeforeClass  
 public static void setUpClass() {  
 }  
  
 @AfterClass  
 public static void tearDownClass() {  
 }  
  
 @Test  
 public void testDivisionByZero() {  
 System.*out*.println("testDivisionByZero");  
 try {  
 int result = CustomMath.*division*(x, y);  
 *assertFalse*(y == 0);  
 System.*out*.println("Test result: passed");  
 } catch (IllegalArgumentException e) {  
 System.*out*.println("Test result: failed. Exception caught: " + e.getMessage());  
 }  
 }  
  
  
 @Test  
 public void testSum() {  
 System.*out*.println("testSum");  
 int expResult = sumResult;  
 int result = CustomMath.*sum*(x, y);  
 if (expResult != result) {  
 *fail*("Test result: failed");  
 } else {  
 System.*out*.println("Test result: passed");  
 }  
 }  
}