Proiect de laborator Testarea Sistemelor Software

Decembrie 2023

Un număr natural nenul n se numește **norocos** dacă pătratul lui se poate scrie ca sumă de n numere naturale consecutive.

Se dă un număr natural nenul n, mai mic ca 100. Programul va afișa[1]:

- cele n numere consecutive separate prin câte un spațiu care adunate dau n * n, dacă n este norocos
- mesajul “Numarul nu este norocos”, dacă n nu este norocos
- mesajul de eroare “Numarul nu este valid”, dacă n nu este natural nenul și <100

Partiționarea în clase de echivalență

Intrările programului: n

Clase intrări

- N_1 = {n | n natural nenul, 1 <= n <= 100} - cazurile pentru n valid
- N_2 = {n | n < 1}
- N_3 = {n | n > 100}

Clase ieșiri

- I_1 = $a_1\ a_2\ a_3 \dots a_n$ (unde $\sum_{i=1}^{n} a_i$ = n * n)
- I_2 = “Numarul nu este norocos”
- I_3 = "Numarul nu este valid"

Clase de echivalență finale

- C1 = {n din N_1, ieșirea I_1} - n = 7
- C2 = {n din N_1, ieșirea I_2} - n = 40
- C3 = {n din N_2, ieșirea I_3} - n = 0
- C4 = {n din N_3, ieșirea I_3} - n = 101

Teste

n	programul afișează
7	4 5 6 7 8 9 10
40	Numarul nu este norocos

[1] Sursa problemei pbinfo.ro https://www.pbinfo.ro/probleme/1892/snorocos

0	Numarul nu este valid
101	Numarul nu este valid

Analiza valorilor de frontieră

Intrări: n
Frontiere pentru n: 0, 1, 100, 101
cazuri speciale: 1 (1 * 1 = 1)

Teste:
(0, “eroare”), (1, “1”), (100,”Numarul nu este norocos”), (101,”eroare”)

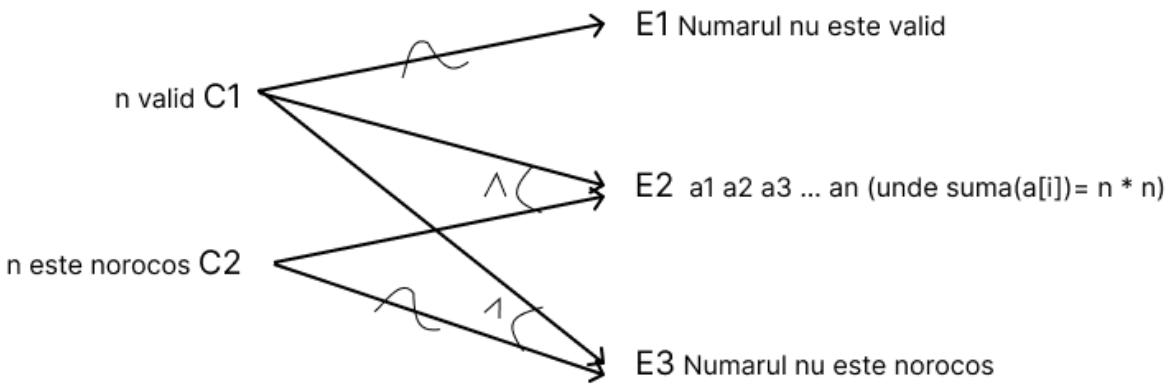
Graful cauză-efect

Cauze:

- C1 = n valid, 1 <= n <= 100
- C2 = n verifică enunțul, numărul este norocos

Efecte:

- E1: mesaj de eroare (Numarul nu este valid)
- E2: $a_1\ a_2\ a_3 \dots a_n$ (unde $\sum_{i=1}^{n} a_i$= n * n)
- E3: Numarul nu este norocos



Parcurgem cele 3 efecte și vom avea următoarele tabele de decizie:

1. E1: mesaj de eroare (Numarul nu este valid) ~C1 = 1 => C1 = 0
   Nu ne intereseaza C2, deci considerăm C2 = 0

C1	0

C2	0
E1(~C1)	1
E2(C1 ^ C2)	0
E3(C1 ^ ~C2)	0

2. E2: $a_1\ a_2\ a_3 \dots a_n$ (unde $\sum_{i=1}^{n} a_i$ = n * n) - Numarul este norocos
   C1 ^ C2 = 1 => C1 = 1; C2 = 1

C1	0	1
C2	0	1
E1(~C1)	1	0
E2(C1 ^ C2)	0	1
E3(C1 ^ ~C2)	0	0

3. E3: Numarul nu este norocos
   C1 ^ ~C2 = 1 => C1 = 1(n trebuie sa fie valid) ; C2 = 0

C1	0	1	1
C2	0	1	0
E1(~C1)	1	0	0
E2(C1 ^ C2)	0	1	0
E3(C1 ^ ~C2)	0	0	1

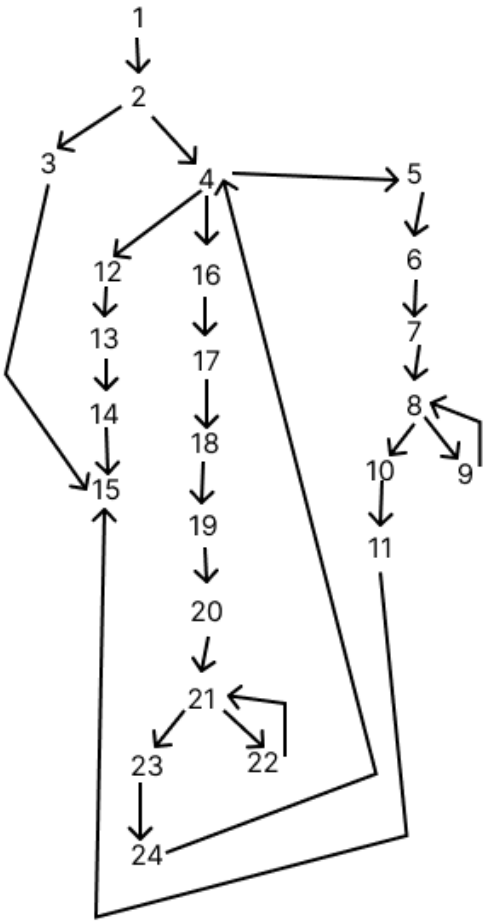
Tabelul final

C1(n valid)	0	1	1
C2(n este norocos)	0	1	0
E1	1	0	0
E2	0	1	0
E3	0	0	1
Input	-10	9	6
Output	eroare	5 6 7 8 9 10 11 12 13	Numarul nu este norocos

Implementarea programului în cod

```java
public static String testNumber(int n) {
	if (n < 1 || n > 100) {
		return "Numarul nu este valid";
	} else if (isNorocos(n)) {
		int start = n - ((n - 1) / 2);
		int end = (n + ((n + 1) / 2)) - 1;

		String result = "";
		for (int i = start; i <= end; i++) {
			result = result + i + " ";
		}
		return result;
	} else {
		return "Numarul nu este norocos";
	}
}

public static boolean isNorocos(int n) {
	int square = n * n;
	int start = n - ((n - 1) / 2);
	int end = (n + ((n + 1) / 2)) - 1;
	int sum = 0;

	for (int j = start; j <= end; j++) {
		sum += j;
	}

	return sum == square;
}
```

Graful orientat al programului



- Decizia D1 = (n < 1 || n > 100) = C1 || C2, unde C1: n < 1 si C2: n > 100

	C1	**C2**	**D1**
~~**T1**~~	~~true~~	~~true~~	~~true~~
T2	false	false	false
T3	false	true	true
T4	true	false	true

T2: n = 9
T3: n = 234
T4: n = -3

- Decizia D2 = isNorocos(n) => C3: isNorocos(n)

	C3	**D2**
T5	true	true

T6	false	false

T2 acopera T5

T6: n = 8

- Decizia D3 = i <= end => C4: i <= end

	C4	**D3**
T7	true	true
T8	false	false

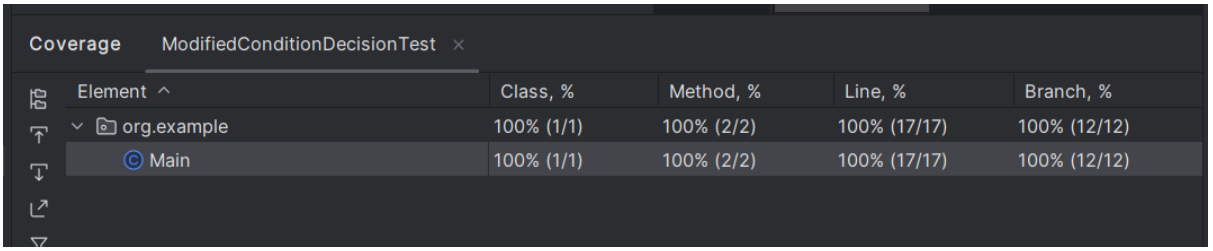
T2 acopera T7 si T8

- Decizia D4 = j <=end => C5: j <= end

	C5	**D4**
T9	true	true
T10	false	false

T6 acopera T9 si T10

```
public class ModifiedConditionDecisionTest {

    @Test
    public void test() {
            assertTrue(Main.testNumber(9).compareTo("5 6 7 8 9 10 11 12 13
") == 0);
            assertTrue(Main.testNumber(234).compareTo("Numarul nu este
valid") == 0);
            assertTrue(Main.testNumber(-3).compareTo("Numarul nu este
valid") == 0);
            assertTrue(Main.testNumber(8).compareTo("Numarul nu este
norocos") == 0);
      }
}
```

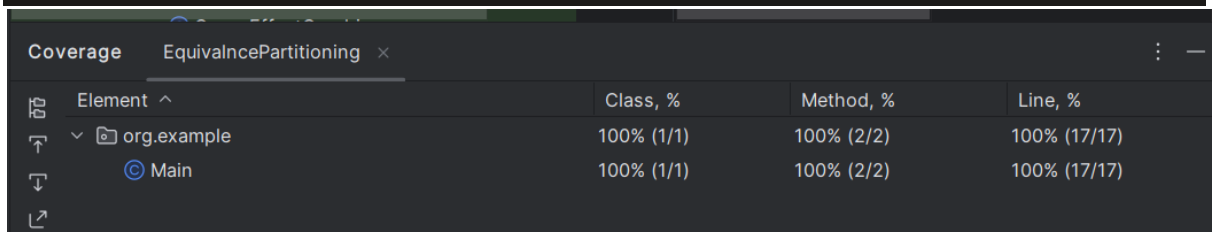


Testare în cod + procente de acoperire

1. Equivalence partitioning:

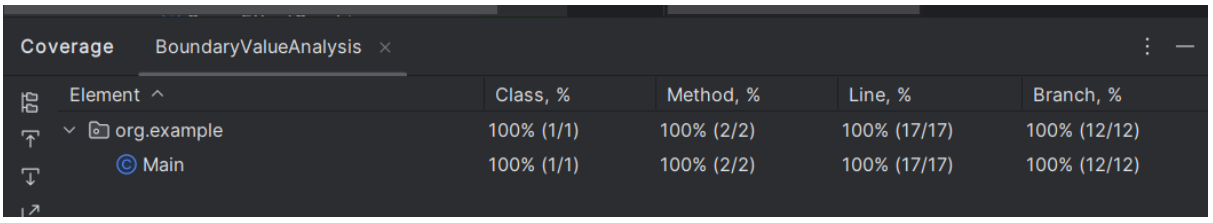
```
public class EquivalncePartitioning {

    @Test
    void testC1() {
            // input valid si este norocos
        assertTrue(DemoApplication.testNumber(7).compareTo("4 5 6 7 8 9 10
") == 0);
        assertTrue(DemoApplication.testNumber(3).compareTo("2 3 4 ") == 0);
        assertTrue(DemoApplication.testNumber(9).compareTo("5 6 7 8 9 10 11
12 13 ") == 0);
        assertTrue(DemoApplication.testNumber(1).compareTo("1 ") == 0);
    }

    @Test
    void testC2() {
            // input valid dar nu este norocos
        assertTrue(DemoApplication.testNumber(40).compareTo("Numarul nu este
norocos") == 0);
        assertTrue(DemoApplication.testNumber(2).compareTo("Numarul nu este
norocos") == 0);
        assertTrue(DemoApplication.testNumber(56).compareTo("Numarul nu este
norocos") == 0);
    }

      @Test
    void testC3() {
            // input invalid n < 1
        assertTrue(DemoApplication.testNumber(0).compareTo("Numarul nu este
valid") == 0);
        assertTrue(DemoApplication.testNumber(-10).compareTo("Numarul nu
este valid") == 0);
        assertTrue(DemoApplication.testNumber(-100).compareTo("Numarul nu
este valid") == 0);
    }

    @Test
```

```
    void testC4() {
            // input invalid n > 100
        assertTrue(DemoApplication.testNumber(333).compareTo("Numarul nu
este valid") == 0);
        assertTrue(DemoApplication.testNumber(124).compareTo("Numarul nu
este valid") == 0);
        assertTrue(DemoApplication.testNumber(101).compareTo("Numarul nu
este valid") == 0);
    }

}
```



2. Boundary value analysis

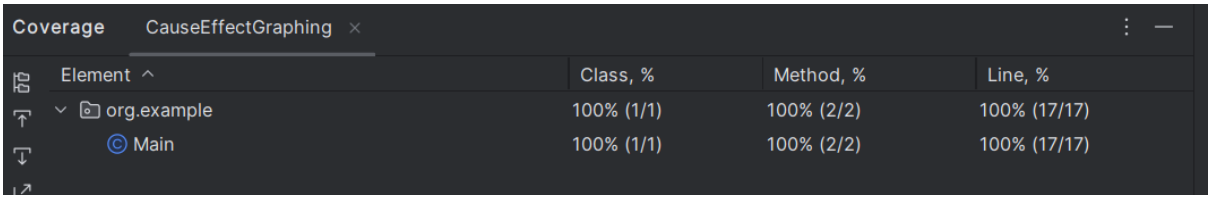
```
public class BoundaryValueAnalysis {

    @Test
    void test() {
        assertTrue(DemoApplication.testNumber(100).compareTo("Numarul nu
este norocos") == 0);
        assertTrue(DemoApplication.testNumber(1).compareTo("1 ") == 0);
        assertTrue(DemoApplication.testNumber(0).compareTo("Numarul nu este
valid") == 0);
        assertTrue(DemoApplication.testNumber(101).compareTo("Numarul nu
este valid") == 0);
    }
}
```



3. Cause-effect graphing

```
public class CauseEffectGraphing {

    @Test
    public void testIsNorocosForEvenNumber() {
```

```
            assertTrue(DemoApplication.testNumber(32).compareTo("Numarul nu
este norocos") == 0);
    }

    @Test
    public void testIsNorocosForOddNumber() {
      assertTrue(DemoApplication.testNumber(11).compareTo("6 7 8 9 10 11 12
13 14 15 16 ") == 0);
    }

    @Test
    public void testInvalidInput() {
      assertTrue(DemoApplication.testNumber(2392).compareTo("Numarul nu este
valid") == 0);
    }
}
```



Mutant de ordinul 1 echivalent al programului

Instrucțiunea 24 devine `return sum == n * n;`

Mutant ne-echivalent omorât de catre test

Considerăm testul:

```
      @Test
      void test() {
            assertTrue(DemoApplication.testNumber(100).compareTo("Numarul nu
este norocos") == 0);
                  }
```

Instrucțiunea 2 devine `if (n < 1 || n >= 100)`
Testul va pica deoarece 100 nu va mai fi testat dacă este un număr norocos sau nu, ci va fi trecut ca valoare invalidă => programul va afișa acum “Numarul nu este valid”.

Mutant ne-echivalent care nu e omorât de către test

Considerăm testul:

```
      @Test
      void test() {
```

```
            assertTrue(DemoApplication.testNumber(1).compareTo("1 ") == 0);
        }
```

Instrucțiunea 17 devine `int square = n;`
Testul nu va pica deoarece 1 * 1 = 1, deci înlocuirea calculării patratului lui n va avea același rezultat pentru valoarea 1.