Proiect de laborator

TSS

Vasile George-Cristian

Grupa 506 IS

Noiembrie 2024

Contents

Cerința

Se dă un număr întreg n, **prim** și **>= 10**. Dacă cifrele **numărului alternează par/impar** să se returneze “da”. În caz contrar să se returneze “nu”. Dacă numărul **nu respectă condițiile** enunțului să se returneze “invalid”.

Rezolvare in java

Metoda *verificaAlternantaPareImpare()* reprezintă rezolvarea cerinței de mai sus.

```
public static String verificaAlternantaPareImpare(int n)
{
    boolean respectaConditia = true;

    if (n < 10 || !estePrim(n)) {
        return "invalid";
    }
    boolean paritateAnterioara = ((n % 10) % 2 == 0);
    n /= 10;

    while (n > 0) {
        boolean paritateCurenta = ((n % 10) % 2 == 0);

        if (paritateCurenta == paritateAnterioara)
        {
            respectaConditia = false;
            break;
        }

        paritateAnterioara = paritateCurenta;
        n /= 10;
    }

    if (!respectaConditia)
        return "NU";
    else
        return "DA";
}

private static boolean estePrim(int numar)
{
    if (numar < 2)
        return false;
    for (int i = 2; i <= Math.sqrt(numar); i++) {
        if (numar % i == 0) {
            return false;
        }
    }
    return true;
}
```

Subpunctul 1

Pe baza cerintelor programului, sa se genereze date de test folosind a) *equivalence partitioning* b) *boundary value analysis* si c) *cause-effect graphing*. Sa se implementeze testele obtinute folosind JUnit.

Equivalence partitioning

Intrarile programului n

Clasele intrarilor

N_1 = {n | n >= 10, n prim}

N_2 = {n | n < 10, n prim}

N_3 = {n | n >= 10, n nu e prim}

N_4 = {n | n < 10, n nu e prim}

Clasele iesirilor

I_1 = “da”

I_2 = “nu”

I_3 = “invalid”

Clase de echivalenta

C_1 = {n din N_1, iesirea I_1}

C_2 = {n din N_1, iesirea I_2}

C_3 = {n din N_2, iesirea I_3}

C_4 = {n din N_3, iesirea I_3}

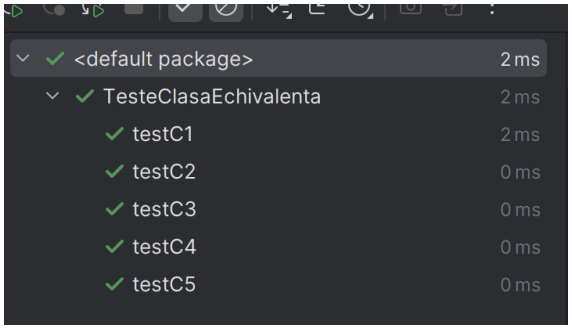
C_5 = {n din N_4, iesirea I_3}

Teste

Clasa de echivalenta	Valoarea lui N	Outputul programului
C_1	$23 \in N_1$	“da” $\in I1$
C_2	$13 \in N_1$	“nu” $\in I_2$
C_3	$5 \in N_2$	“invalid” $\in I_3$
C_4	$12 \in N_3$	“invalid” $\in I_3$
C_5	$6 \in N_4$	“invalid” $\in I_3$

Implementarile testelor

```java
public class TesteClasaEchivalenta {
    @Test
    public void testC1() {
        // Clasa de echivalență C1: n = 23, output = "DA"
        int n = 23;
        String rezultat = Main.verificaAlternantaPareImpare(n);
        assertEquals("DA", rezultat);
    }

    @Test
    public void testC2() {
        // Clasa de echivalență C2: n = 13, output = "NU"
        int n = 13;
        String rezultat = Main.verificaAlternantaPareImpare(n);
        assertEquals("NU", rezultat);
    }

    @Test
    public void testC3() {
        // Clasa de echivalență C3: n = 5, output = "invalid"
        int n = 5;
        String rezultat = Main.verificaAlternantaPareImpare(n);
        assertEquals("invalid", rezultat);
    }

    @Test
    public void testC4() {
        // Clasa de echivalență C4: n = 12, output = "invalid"
        int n = 12;
        String rezultat = Main.verificaAlternantaPareImpare(n);
        assertEquals("invalid", rezultat);
    }

    @Test
    public void testC5() {
        // Clasa de echivalență C5: n = 6, output = "invalid"
        int n = 6;
        String rezultat = Main.verificaAlternantaPareImpare(n);
        assertEquals("invalid", rezultat);
    }
}
```



Boundary value analysis

Intrari: n

Valori de frontieră 9, 10

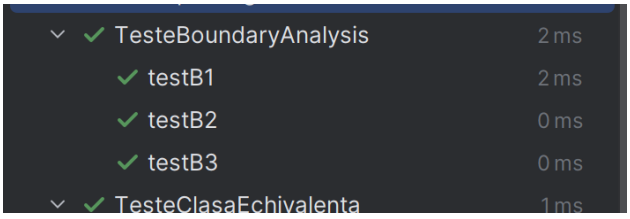
Cazuri speciale - numere prime mai mici decat 10 - 2, 3, 5, 7, 9. Toate sunt la fel, voi lua doar 3.

Teste

Teste	Valoarea lui N	Outputul programului
B_1	9	“invalid”
B_2	10	“invalid”
B_3	3	“invalid”

Implementarile testelor

```
public class TesteBoundaryAnalysis {

    @Test
    public void testB1() {
        // Testul B1: n = 9, output = "invalid"
        int n = 9;
        String rezultat = Main.verificaAlternantaPareImpare(n);
        assertEquals("invalid", rezultat);
    }

    @Test
    public void testB2() {
        // Testul B2: n = 10, output = "invalid"
        int n = 10;
        String rezultat = Main.verificaAlternantaPareImpare(n);
        assertEquals("invalid", rezultat);
    }

    @Test
    public void testB3() {
        // Testul B3: n = 3, output = "invalid"
        int n = 3;
        String rezultat = Main.verificaAlternantaPareImpare(n);
        assertEquals("invalid", rezultat);
    }
}
```



Cause-effect graphing

Cauze

C_1: n >= 10

C_2: n prim

C_3: cifrele lui n alternează par/impar

Efecte

E_1: returnează da

E_2: returnează nu

E_3: returnează invalid

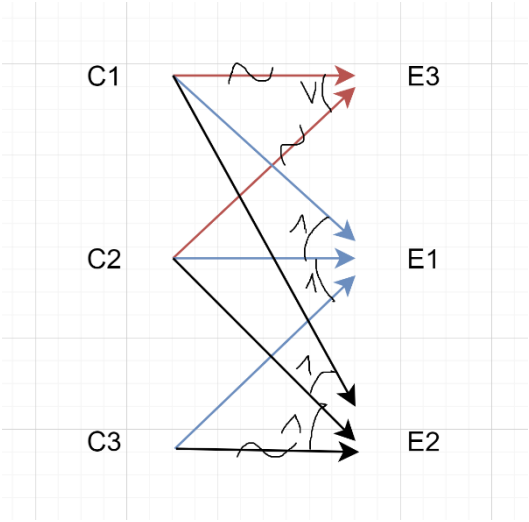


Figure 1 Diagrama cauză – efect

Tabelul cauză-efect

E_1: C_1 ^ C_2 ^ C_3

C_1	1
C_2	1
C_3	1
E_1	1
E_2	0
E_3	0

E_2: C_1 ^ C_2 ^ ~C_3

C_1	1	1
C_2	1	1
C_3	1	0
E_1	1	0
E_2	0	1
E_3	0	0

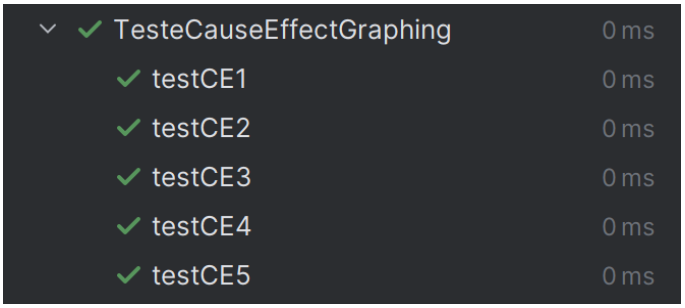
E_3: ~C_1 V ~C_2 echivalent cu ~(C_1 ^ C_2). Voi lua 3 cazuri cu toate valorile posibile care dau conditia adevărată.

C_1	1	1	0	0	1
C_2	1	1	0	1	0
C_3	1	0	0	0	0
E_1	1	0	0	0	0
E_2	0	1	0	0	0
E_3	0	0	1	1	1

Tabel final

	Ce_1	Ce_2	Ce_3	Ce_4	Ce_5
C_1	1	1	0	0	1
C_2	1	1	0	1	0
C_3	1	0	0	0	0
E_1	1	0	0	0	0
E_2	0	1	0	0	0
E_3	0	0	1	1	1
Input	29	19	8	3	10
Output	da	nu	invalid	invalid	invalid

```java
public class TesteCauseEffectGraphing {

    @Test
    public void testCE1() {
        // CE1: n = 29, output = "DA"
        // 29 este prim, >= 10, și cifrele alternează par/impar
        int n = 29;
        String rezultat = Main.verificaAlternantaPareImpare(n);
        assertEquals("DA", rezultat);
    }

    @Test
    public void testCE2() {
        // CE2: n = 19, output = "NU"
        // 19 este prim, >= 10, dar cifrele nu alternează par/impar
        int n = 19;
        String rezultat = Main.verificaAlternantaPareImpare(n);
        assertEquals("NU", rezultat);
    }

    @Test
    public void testCE3() {
        // CE3: n = 8, output = "invalid"
        // 8 este mai mic decât 10, dar nu este prim
        int n = 8;
        String rezultat = Main.verificaAlternantaPareImpare(n);
        assertEquals("invalid", rezultat);
    }

    @Test
    public void testCE4() {
        // CE4: n = 3, output = "invalid"
        // 3 este prim, dar este mai mic decat 10
        int n = 3;
        String rezultat = Main.verificaAlternantaPareImpare(n);
        assertEquals("invalid", rezultat);
    }

    @Test
    public void testCE5() {
        // CE5: n = 10, output = "NU"
        // 10 este >= 10, dar nu este prim
        int n = 10;
        String rezultat = Main.verificaAlternantaPareImpare(n);
        assertEquals("invalid", rezultat);
    }
}
```



Subpunctul 2

Sa se stabileasca nivelul de acoperire realizat de **fiecare** dintre seturile de teste de la 1) a), b) si c) folosind unul dintre utilitarele de code coverage prezentate in [6]. Sa se compare si sa se comenteze rezultatele obtinute de cele trei seturi de teste.

Am folosit utilitarul de *code coverage* implicită pentru JUnit in IntelliJ, care **cred** că este JaCoCo.

Equivalence partitioning

Element	Class, %	Method, %	Line, %	Branch, %
all	50% (2/4)	43% (7/16)	56% (34/60)	93% (15/16)
Main	100% (1/1)	66% (2/3)	90% (19/21)	93% (15/16)
TesteBoundaryAnalysis	0% (0/1)	0% (0/3)	0% (0/9)	100% (0/0)
TesteCauseEffectGraphing	0% (0/1)	0% (0/5)	0% (0/15)	100% (0/0)
TesteClasaEchivalenta	100% (1/1)	100% (5/5)	100% (15/15)	100% (0/0)

Boundary-value analysis

Element	Class, %	Method, %	Line, %	Branch, %
all	100% (2/2)	83% (5/6)	53% (16/30)	37% (6/16)
Main	100% (1/1)	66% (2/3)	33% (7/21)	37% (6/16)
TesteBoundaryAnalysis	100% (1/1)	100% (3/3)	100% (9/9)	100% (0/0)

Cause-effect graphing

Element	Class, %	Method, %	Line, %	Branch, %
all	50% (2/4)	43% (7/16)	56% (34/60)	93% (15/16)
Main	100% (1/1)	66% (2/3)	90% (19/21)	93% (15/16)
TesteBoundaryAnalysis	0% (0/1)	0% (0/3)	0% (0/9)	100% (0/0)
TesteCauseEffectGraphing	100% (1/1)	100% (5/5)	100% (15/15)	100% (0/0)
TesteClasaEchivalenta	0% (0/1)	0% (0/5)	0% (0/15)	100% (0/0)

Concluzii

Putem ignora 66% method coverage deoarece nu se testează metoda main() a programului, care este irelevantă. Un indice bun este branch coverage, care verifică dacă programul nostru intră prin toate cazurile posibile definite de if-uri. Atât *equivalence partitioning* cât și *cause-effect graphing* au un coverage foarte bun, de 93%, fiind net superioare analizei *boundary-value*.

```
37      private static boolean estePrim(int numar)  1 usage
38      {
39          if (numar < 2)
40              return false;
```

De asemenea, singurul *branch* nelovit este cel din figura de mai sus, care este imposibil de lovit in flow-ul nostru deoarece functia nu se apelează pentru numere mai mici decât 10.

Așadar, putem spune că am obținut un coverage satisfăcător pentru rezolvarea noastră.

Subpunctul 3

Sa se transforme programul intr-un graf orientat si, pe baza acestuia, sa se gaseasca un set de teste care satisface criteriul *modified condition/decision coverage* (MC/DC).

Graful

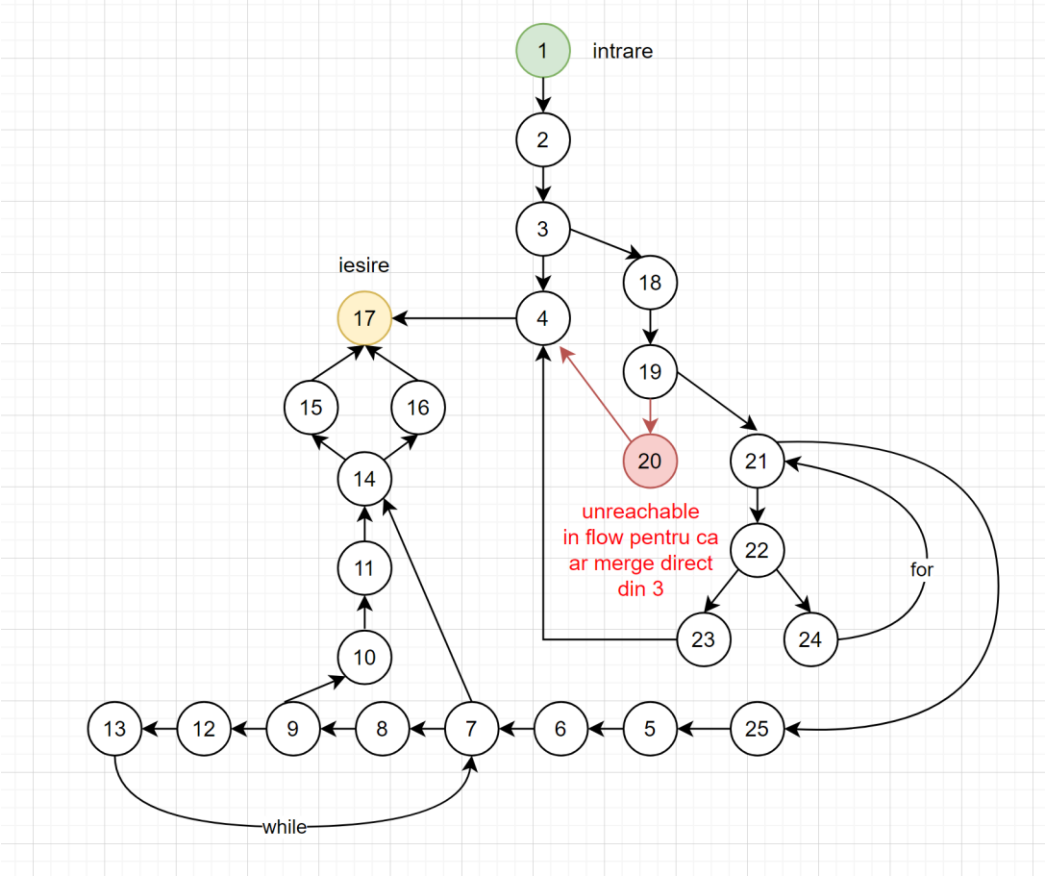


Figure 2 Graful orientat

Transformarea instructiunilor in noduri

```
1  public static String verificaAlternantaPareImpare(int n) {
2    boolean respectaConditia = true;
3     if (n < 10 || !estePrim(n)) {
4         return "invalid";
      }
5     boolean paritateAnterioara = ((n % 10) % 2 == 0);
6     n /= 10;
7     while (n > 0) {
8         boolean paritateCurenta = ((n % 10) % 2 == 0);
9         if (paritateCurenta == paritateAnterioara)
          {
10            respectaConditia = false;
11            break;
          }
12        paritateAnterioara = paritateCurenta;
13        n /= 10;
      }
14    if (!respectaConditia)
15        return "NU";
      else
16        return "DA";
17 }

18 private static boolean estePrim(int numar) {
19    if (numar < 2)
20        return false;
21    for (int i = 2; i <= Math.sqrt(numar); i++) {
22        if (numar % i == 0) {
23            return false;
24        }
      }
25    return true;
   }
```

Set de teste pentru MC/DC

Am ales condiția de la nodul 3 - **if (n < 10 || !estePrim(n))**

Conditii individuale

Co_1: n < 10

Co_2: estePrim(n)

Decizia

D: (n < 10 || !estePrim(n)) <=> Co_1 V ~Co_2

Restul cerinței nu este rezolvată.

Subpunctul 4

Să se scrie un mutant de ordinul 1 echivalent al programului.

Mutant echivalent

```
public static String verificaAlternantaPareImpare(int n)
{
    boolean respectaConditia = true;

    if (n < 10 || !estePrim(n)) {
        return "invalid";
    }
    boolean paritateAnterioara = ((n % 10) % 2 == 0);
    n /= 10;

    while (n > 0) {
        boolean paritateCurenta = ((n % 10) % 2 == 0);

        if (paritateCurenta == paritateAnterioara)
        {
            respectaConditia = false;
            ; //eliminarea break-ului
        }

        paritateAnterioara = paritateCurenta;
        n /= 10;
    }

    if (!respectaConditia)
        return "NU";
    else
        return "DA";
}

private static boolean estePrim(int numar)
{
    if (numar < 2)
        return false;
    for (int i = 2; i <= Math.sqrt(numar); i++) {
        if (numar % i == 0) {
            return false;
        }
    }
    return true;
}
```

Justificare

Prin eliminarea instructiunii break și înlocuirea cu una goală am făcut o singură modificare programului, deci mutantul este de ordinul 1. Eliminarea instrucțiunii va cauza dispariția “bypass-ului” pentru ieșirea din *while* și va cauza mai multe execuții ale loop-ului fără a modifica în vreun fel outputul programului, prin definiție fiind echivalent.

Subpunctul 5

Pentru unul dintre cazurile de testare de mai sus sa se scrie un mutant ne-echivalent care sa fie omorat de catre test si un mutant ne-echivalent care sa nu fie omorat de catre test.

Voi utiliza cazul C_3 de la equivalence partitioning, care trimite input 5 și se așteaptă la outputul “invalid”.

Mutant ne-echivalent omorât de către test

Mutantul este ne-echivalent deoarece programul își schimbă outputul pentru anumite inputuri, cu $n \in \{3, 4, 5, 6, 7, 8, 9\}$. În acest caz funcţia va returna “da” în loc de “invalid” deci testul îl va omorî.

```
public class MutantNeechivalentOmorat {
    public static String verificaAlternantaPareImpare(int n)
    {
        boolean respectaConditia = true;

        if (n < 3 || !estePrim(n)) { //schimbam n < 10 cu n < 3
            return "invalid";
        }

        boolean paritateAnterioara = ((n % 10) % 2 == 0);
        n /= 10;

        while (n > 0) {
            boolean paritateCurenta = ((n % 10) % 2 == 0);

            if (paritateCurenta == paritateAnterioara)
            {
                respectaConditia = false;
                break;
            }

            paritateAnterioara = paritateCurenta;
            n /= 10;
        }

        if (!respectaConditia)
            return "NU";
        else
            return "DA";
    }

    private static boolean estePrim(int numar)
    {
        if (numar < 2)
            return false;
        for (int i = 2; i <= Math.sqrt(numar); i++) {
            if (numar % i == 0) {
                return false;
            }
        }
        return true;
    }
}
```

Mutant ne-echivalent care nu este omorât de către test

Mutantul este ne-echivalent deoarece programul își schimbă outputul pentru anumite inputuri, cu $n \in \{8, 9\}$. În acest caz funcția va returna același output pentru $n = 5$ deci testul nu îl va omorî.

```
public static String verificaAlternantaPareImpare(int n)
{
    boolean respectaConditia = true;

    if (n < 8 || !estePrim(n)) { //schimbam n < 10 cu n < 8
        return "invalid";
    }

    boolean paritateAnterioara = ((n % 10) % 2 == 0);
    n /= 10;

    while (n > 0) {
        boolean paritateCurenta = ((n % 10) % 2 == 0);

        if (paritateCurenta == paritateAnterioara)
        {
            respectaConditia = false;
            break;
        }

        paritateAnterioara = paritateCurenta;
        n /= 10;
    }

    if (!respectaConditia)
        return "NU";
    else
        return "DA";
}

private static boolean estePrim(int numar)
{
    if (numar < 2)
        return false;
    for (int i = 2; i <= Math.sqrt(numar); i++) {
        if (numar % i == 0) {
            return false;
        }
    }
    return true;
}
```