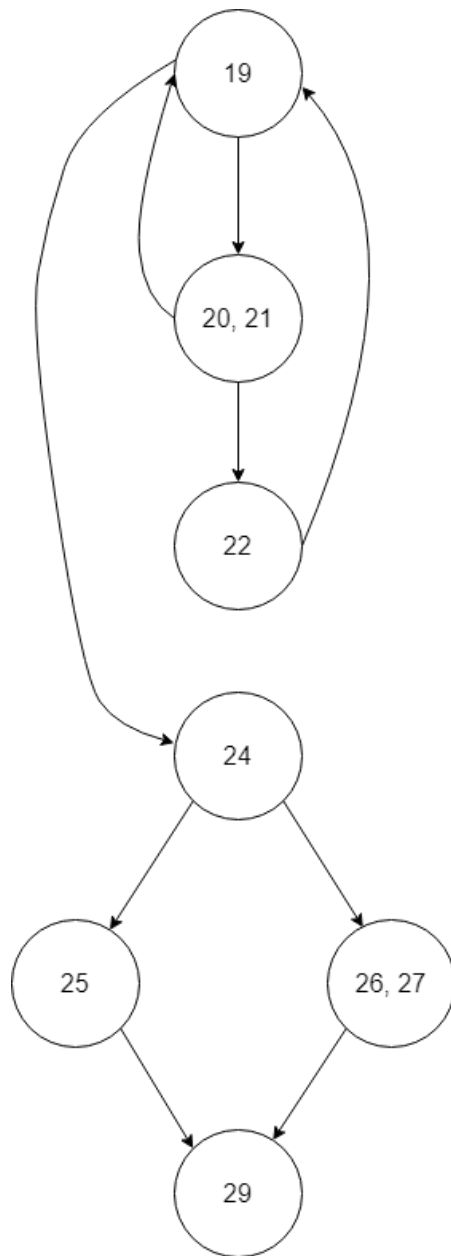


I. Testare structurala

1. Cfg:



2. Teste:

- a. Statement coverage:

INTRARI	NODURI PARCURSE
x = 20, y = 10	19, 20, 21, 22, 24, 25, 29
x = 20, y = 30	19, 20, 21, 22, 24, 26, 27, 29

b. Decision coverage:

NUMAR	DECIZIE
1	while (x > 10)
2	if (x == 10)
3	if (y < 20 && (x % 2) == 0)

INTRARI	DECIZII ACOPERITE + VAL DE ADEVAR
x = 20, y = 10	1-A, 2-A, 3-A
x = 15, y = 30	1-A, 1-F , 2-F, 3-F

c. Condition coverage:

NUMAR	CONDITIE
1	x > 10
2	x == 10
3	y < 20
4	x % 2 == 0

INTRARI	CONDITII ACOPERITE + VAL DE ADEVAR
x = 20, y = 10	1-T, 2-T, 3-T, 4-T
x = 5, y = 30	1-F, 3-F, 4-F
x = 21, y = 10	1-T, 2-F

3. Tool de verificare:

Coverage report: 100%

Files Functions Classes

coverage.py v7.5.1, created at 2024-05-06 22:34 +0300

File ▲	function	statements	missing	excluded	coverage
testNumProc.py	NumberProcessorTest.test_statement_coverage	4	0	0	100%
testNumProc.py	NumberProcessorTest.test_decision_coverage	4	0	0	100%
testNumProc.py	NumberProcessorTest.test_condition_coverage	6	0	0	100%
testNumProc.py	(no function)	8	0	0	100%
Total		22	0	0	100%

coverage.py v7.5.1, created at 2024-05-06 22:34 +0300

4. Comparatie cu AI:

Rezultate ChatGPT:

i. Statement coverage:

```
import unittest
```

```
class TestNumberProcessor(unittest.TestCase):
```

```
    def test_x_initially_less_than_10(self):
```

```
        processor = NumberProcessor(9, 25)
```

```
        result = processor.process_numbers()
```

```
        self.assertEqual(result, 2 * 9 + 5) # x = 9, y = 5
```

```
    def test_x_reduces_to_exactly_10(self):
```

```
        processor = NumberProcessor(30, 18)
```

```
        result = processor.process_numbers()
```

```
        self.assertEqual(result, 2 * 10 + 38) # x = 10, y = 38
```

```
    def test_x_large_not_reaching_10(self):
```

```
        processor = NumberProcessor(25, 25)
```

```

    result = processor.process_numbers()

    self.assertEqual(result, 2 * 5 + 5) # x = 5, y = 5


def test_y_less_than_20_x_even(self):

    processor = NumberProcessor(12, 19)

    result = processor.process_numbers()

    self.assertEqual(result, 2 * 2 + 39) # x = 2, y = 39


def test_y_less_than_20_x_odd(self):

    processor = NumberProcessor(13, 19)

    result = processor.process_numbers()

    self.assertEqual(result, 2 * 3 + -1) # x = 3, y = -1


def test_y_greater_or_equal_20(self):

    processor = NumberProcessor(8, 20)

    result = processor.process_numbers()

    self.assertEqual(result, 2 * 8 + 0) # x = 8, y = 0


if __name__ == '__main__':

    unittest.main()

```

ii. **Decision coverage:**

```

import unittest


class TestNumberProcessor(unittest.TestCase):

    def test_while_loop_continues(self):

```

```
processor = NumberProcessor(22, 10)

result = processor.process_numbers()

self.assertEqual(result, 2 * 2 + -10) # x reduces to 2, y = -10
```

```
def test_while_loop_breaks(self):

    processor = NumberProcessor(20, 10)

    result = processor.process_numbers()

    self.assertEqual(result, 2 * 10 + -10) # x reduces to 10, y = -10
```

```
def test_if_condition_true(self):

    processor = NumberProcessor(14, 19)

    result = processor.process_numbers()

    self.assertEqual(result, 2 * 4 + 39) # x reduces to 4, y increases to 39
```

```
def test_if_condition_false(self):

    processor = NumberProcessor(15, 19)

    result = processor.process_numbers()

    self.assertEqual(result, 2 * 5 + -1) # x reduces to 5, y decreases to -1
```

```
def test_if_condition_false_y_initially_high(self):

    processor = NumberProcessor(12, 21)

    result = processor.process_numbers()

    self.assertEqual(result, 2 * 2 + 1) # x reduces to 2, y decreases to 1
```

```
if __name__ == '__main__':

    unittest.main()
```

iii. **Condition coverage:**

```
import unittest
```

```
class TestNumberProcessor(unittest.TestCase):
```

```
    def test_condition_while_greater_than_10_true(self):
```

```
        processor = NumberProcessor(21, 15)
```

```
        result = processor.process_numbers()
```

```
        self.assertEqual(result, 2 * 1 + -5) # x reduces to 1, y = -5
```

```
    def test_condition_while_greater_than_10_false(self):
```

```
        processor = NumberProcessor(10, 15)
```

```
        result = processor.process_numbers()
```

```
        self.assertEqual(result, 2 * 10 + -5) # x remains 10, y = -5
```

```
    def test_condition_if_y_less_than_20_true(self):
```

```
        processor = NumberProcessor(12, 18)
```

```
        result = processor.process_numbers()
```

```
        self.assertEqual(result, 2 * 2 + 38) # x reduces to 2, y = 38
```

```
    def test_condition_if_y_less_than_20_false(self):
```

```
        processor = NumberProcessor(12, 20)
```

```
        result = processor.process_numbers()
```

```
        self.assertEqual(result, 2 * 2 + 0) # x reduces to 2, y = 0
```

```
    def test_condition_if_x_even_true(self):
```

```

processor = NumberProcessor(14, 19)

result = processor.process_numbers()

self.assertEqual(result, 2 * 4 + 39) # x reduces to 4, y = 39

def test_condition_if_x_even_false(self):

    processor = NumberProcessor(15, 19)

    result = processor.process_numbers()

    self.assertEqual(result, 2 * 5 + -1) # x reduces to 5, y = -1

if __name__ == '__main__':

    unittest.main()

```

Concluzie: Ai scrie teste corecte, dar multe dintre ele ar putea fi eliminate datorita faptului ca deja au fost acoperite acele unitati. Scrise manual ele sunt mai putine, dar efortul de concepere a acestora e mai mare

II. Testare functionala:

(a) Partiționare de echivalență (equivalence partitioning)

1. Domeniul de intrări:

Există două intrări, x și y, ambele fiind numere întregi.

- x trebuie să fie un număr întreg, deci se disting 2 clase de echivalență:

$X_1 = \{ x \mid x \text{ este un număr întreg} \}$

$X_2 = \{ x \mid x \text{ nu este un număr întreg} \}$

- y trebuie să fie un număr întreg, deci se disting 2 clase de echivalență:

$Y_1 = \{ y \mid y \text{ este un număr întreg} \}$

$Y_2 = \{ y \mid y \text{ nu este un număr întreg} \}$

2. Domeniul de ieșiri:

Constă dintr-un singur număr întreg, care este rezultatul procesării numerelor x și

y . Acesta este folosit pentru a împărți domeniul de intrare în mai multe clase, în funcție de valorile

x și y :

- $C_1(x, y) = \{ \text{rezultat} \mid x > 10 \}$
- $C_2(x, y) = \{ \text{rezultat} \mid x \leq 10 \}$
- $C_3(x, y) = \{ \text{rezultat} \mid y < 20 \text{ și } x \text{ este par} \}$
- $C_4(x, y) = \{ \text{rezultat} \mid y < 20 \text{ și } x \text{ este impar} \}$
- $C_5(x, y) = \{ \text{rezultat} \mid y \geq 20 \}$

Clasele de echivalență globale pentru întregul program, care sunt combinații ale claselor individuale:

$G_1 = \{ (x, y) \mid x \in X_1, y \in Y_1, x \text{ și } y \text{ sunt numere întregi} \}$

$G_2 = \{ (x, y) \mid x \in X_2 \text{ or } y \in Y_2, x \text{ sau } y \text{ nu sunt numere întregi} \}$

$G_3 = \{ (x, y) \mid x \in X_1, y \in Y_1, x > 10 \}$

$G_4 = \{ (x, y) \mid x \in X_1, y \in Y_1, x \leq 10 \}$

$G_5 = \{ (x, y) \mid x \in X_1, y \in Y_1, y < 20, x \text{ este par} \}$

Setul de date de test se alcătuiește alegându-se o valoare a intrărilor pentru fiecare clasă de echivalență. De exemplu:

$g_1: (10, 20)$

$g_{2.1}: ('10', 20)$

$g_{2.2}: (10, '20')$

$g_3: (15, 20)$

$g_4: (10, 20)$

$g_5: (10, 10)$

Intrari		Rezultat afisat(expected)
x	y	
10	20	20
'10'	20	Ridica ValueError: "Valoarea trebuie să fie un număr întreg."
10	'20'	Ridica ValueError: "Valoarea trebuie să fie un număr întreg."
15	20	10
10	20	20
10	10	50

```
#=====Partiționare de echivalență=====

def test_x_and_y_are_integers(self):
    Np = NumberProcessor(10, 20)
    self.assertEqual(Np.process_numbers(), 20)

def test_x_or_y_not_integer(self):
    with self.assertRaises(ValueError):
        Np = NumberProcessor('10', 20)
        Np.process_numbers()
    with self.assertRaises(ValueError):
        Np = NumberProcessor(10, '20')
        Np.process_numbers()

def test_x_greater_than_10(self):
    Np = NumberProcessor(15, 20)
    self.assertEqual(Np.process_numbers(), 10)

def test_x_less_or_equal_to_10(self):
    Np = NumberProcessor(10, 20)
    self.assertEqual(Np.process_numbers(), 20)

def test_y_less_than_20_x_even(self):
    Np = NumberProcessor(10, 10)
    self.assertEqual(Np.process_numbers(), 50)

#=====
```

3. Tool de verificare

Coverage report: 98%

Files Functions Classes

coverage.py v7.5.1, created at 2024-05-09 14:25 +0300

File ▲	function	statements	missing	excluded	coverage
NumProc.py	NumberProcessor.__init__	2	0	0	100%
NumProc.py	NumberProcessor.validate_integer	3	0	0	100%
NumProc.py	NumberProcessor.process_numbers	10	0	0	100%
NumProc.py	(no function)	4	0	0	100%
testNumProc.py	NumberProcessorTest.test_statement_coverage	4	0	0	100%
testNumProc.py	NumberProcessorTest.test_decision_coverage	4	0	0	100%
testNumProc.py	NumberProcessorTest.test_condition_coverage	6	0	0	100%
testNumProc.py	NumberProcessorTest.test_x_and_y_are_integers	2	0	0	100%
testNumProc.py	NumberProcessorTest.test_x_or_y_not_integer	6	0	0	100%
testNumProc.py	NumberProcessorTest.test_x_greater_than_10	2	0	0	100%
testNumProc.py	NumberProcessorTest.test_x_less_or_equal_to_10	2	0	0	100%
testNumProc.py	NumberProcessorTest.test_y_less_than_20_x_even	2	0	0	100%
testNumProc.py	(no function)	13	1	0	92%
Total		60	1	0	98%

coverage.py v7.5.1, created at 2024-05-09 14:25 +0300

4. Comparatie cu AI

```
import unittest
```

```
class TestNumberProcessor(unittest.TestCase):
```

```
    def test_process_numbers_x_greater_than_10(self):
```

```
        processor = NumberProcessor(15, 5)
```

```
        result = processor.process_numbers()
```

```
        self.assertEqual(result, 40)
```

```
    def test_process_numbers_x_equal_to_10(self):
```

```
        processor = NumberProcessor(10, 5)
```

```
        result = processor.process_numbers()
```

```
        self.assertEqual(result, 30)
```

```
    def test_process_numbers_y_less_than_20_and_x_even(self):
```

```
        processor = NumberProcessor(5, 15)
```

```

        result = processor.process_numbers()

        self.assertEqual(result, 30)

    def test_process_numbers_y_less_than_20_and_x_odd(self):

        processor = NumberProcessor(7, 15)

        result = processor.process_numbers()

        self.assertEqual(result, -24)

    def test_process_numbers_y_greater_than_or_equal_to_20(self):

        processor = NumberProcessor(5, 25)

        result = processor.process_numbers()

        self.assertEqual(result, 20)

    def test_process_numbers_x_and_y_as_floats(self):

        processor = NumberProcessor(5.5, 10.5)

        with self.assertRaises(ValueError):

            processor.process_numbers()

    def test_process_numbers_x_and_y_as_strings(self):

        processor = NumberProcessor("5", "10")

        with self.assertRaises(ValueError):

            processor.process_numbers()

if name == ' main ':

    unittest.main()

```

Concluzie: Testele scrise de AI sunt corecte, dar cazurile care au fost abordate sunt deja acoperite de cele scrise initial.

(b) Analiza valorilor de frontieră (boundary value analysis)

Analiza valorilor de frontieră reprezintă o tehnică de testare fundamentală pentru a determina reacția unui sistem la valori-limită ale variabilelor de intrare. Această tehnică ajută la descoperirea defectelor care apar la marginile domeniului valid sau al valorilor critice. Următoarele cazuri de testare acoperă diferite condiții limită pentru clasa NumberProcessor, care procesează numerele x și y și returnează un rezultat.

Caz de Testare: test_x_just_above_10_y_below_20

```
def test_x_just_above_10_y_below_20(self):
    processor = NumberProcessor(11, 19)
    result = processor.process_numbers()
    self.assertEqual(processor.x, 1)
    self.assertEqual(processor.y, -1)
    self.assertEqual(result, 2 * 1 - 1)
```

Input: x = 11, y = 19

Output x = 1, y = -1, Rezultat = $2 * 1 - 1$

Scop: Acest caz urmărește comportamentul funcției atunci când x depășește puțin valoarea de referință 10, iar y este sub limita superioară de 20.

Modificarea variabilei x la 1 și y la -1 în urma procesării indică o schimbare logică în funcționalitatea procesorului.

Caz de Testare: test_x_is_exactly_10_y_exactly_20

```
def test_x_is_exactly_10_y_exactly_20(self):
    processor = NumberProcessor(10, 20)
    result = processor.process_numbers()
    self.assertEqual(processor.x, 10)
    self.assertEqual(processor.y, 0)
    self.assertEqual(result, 2 * 10 + 0)
```

Input: x = 10, y = 20

Rezultat Așteptat: x = 10, y = 0, Rezultat = $2 * 10 + 0$

Scop:

Acest test confirmă funcționalitatea atunci când x și y sunt exact la valorile de limită, respectiv 10 și 20.

Rezultatul obținut, $2 * 10 + 0$, reflectă corectitudinea procesării în acest caz.

Caz de Testare: test_x_below_10_y_above_20

```
def test_x_below_10_y_above_20(self):  
    processor = NumberProcessor(8, 25)  
    result = processor.process_numbers()  
    self.assertEqual(processor.x, 8)  
    self.assertEqual(processor.y, 5)  
    self.assertEqual(result, 2 * 8 + 5)
```

Input: $x = 8, y = 25$

Output: $x = 8, y = 5$, Rezultat = $2 * 8 + 5$

Scop:

Testul urmărește modul în care funcția gestionează cazul în care x este sub 10, iar y este peste 20.

Valorile procesate confirmă așteptările de ajustare a variabilei y cu +5 și păstrarea valorii inițiale a lui x .

Caz de Testare: test_x_is_negative

```
def test_x_is_negative(self):  
    processor = NumberProcessor(-5, 15)  
    result = processor.process_numbers()  
    self.assertEqual(processor.x, -5)  
    self.assertEqual(processor.y, -5)  
    self.assertEqual(result, 2 * -5 + -5)
```

Input: $x = -5, y = 15$

Output: $x = -5, y = -5$, Rezultat = $2 * -5 + -5$

Scop:

Testarea valorilor negative în x și y ajută la validarea robusteții procesării.

Aici, y este ajustat la valoarea negativă corespunzătoare pentru a menține consistența logicii.

Caz de Testare: test_x_is_zero_y_is_negative

```
def test_x_is_zero_y_is_negative(self):
    processor = NumberProcessor(0, -10)
    result = processor.process_numbers()
    self.assertEqual(processor.x, 0)
    self.assertEqual(processor.y, 10)
    self.assertEqual(result, 2 * 0 + 10)
```

Input: $x = 0$, $y = -10$

Output: $x = 0$, $y = 10$, Rezultat = $2 * 0 + 10$

Scop:

Acest test urmărește gestionarea unei valori de 0 în x și unei valori negative în y .

Schimbarea lui y în valoare pozitivă confirmă abordarea funcției de procesare în aceste cazuri.

Caz de Testare: test_x_above_10_and_even_y_below_20

```
def test_x_above_10_and_even_y_below_20(self):
    processor = NumberProcessor(12, 18)
    result = processor.process_numbers()
    self.assertEqual(processor.x, 2)
    self.assertEqual(processor.y, 38)
    self.assertEqual(result, 2 * 2 + 38)
```

Input: $x = 12$, $y = 18$

Output: $x = 2$, $y = 38$, Rezultat = $2 * 2 + 38$

Scop:

Acest caz validează situația în care x este peste 10 și par, iar y este sub 20.

Variabila x este redusă la 2, iar y crește la 38, conform regulilor de procesare.

Caz de Testare: test_x_below_10_and_even_y_below_20

```
def test_x_below_10_and_even_y_below_20(self):
    processor = NumberProcessor(4, 15)
    result = processor.process_numbers()
    self.assertEqual(processor.x, 4)
    self.assertEqual(processor.y, 35)
    self.assertEqual(result, 2 * 4 + 35)
```

Input: x = 4, y = 15

Output: x = 4, y = 35, Rezultat = $2 * 4 + 35$

Scop:

Testul urmărește comportamentul funcției când x este sub 10 și par, iar y sub 20.

Aici, x rămâne 4, iar y crește la 35.

Caz de Testare: test_x_above_10_and_odd_y_above_20

```
def test_x_above_10_and_odd_y_above_20(self):
    processor = NumberProcessor(13, 30)
    result = processor.process_numbers()
    self.assertEqual(processor.x, 3)
    self.assertEqual(processor.y, 10)
    self.assertEqual(result, 2 * 3 + 10)
##=====##
```

Input: x = 13, y = 30

Output: x = 3, y = 10, Rezultat = $2 * 3 + 10$

Scop:

Verifică cazul în care x este peste 10 și impar, iar y este peste 20.

Variabila x este redusă la 3, în timp ce y este ajustat la 10.

Intrari		Rezultat afisat(expected)
x	y	
11	19	1

10	20	20
8	25	21
-5	15	-15
0	-10	10
12	18	42
4	15	43
13	30	16

Comparație cu AI

```
import unittest
```

```
def test_x_and_y_extremely_high(self):
    processor = NumberProcessor(int(1e9), int(1e9))
    result = processor.process_numbers()
    expected_x = 10
    expected_y = int(1e9) - 20
    expected_result = 2 * expected_x + expected_y
    self.assertEqual(processor.x, expected_x)
    self.assertEqual(processor.y, expected_y)
    self.assertEqual(result, expected_result)
```



```

def test_x_positive_y_negative(self):
    processor = NumberProcessor(20, -20)
    result = processor.process_numbers()
    expected_x = 10
    expected_y = 0
    expected_result = 2 * expected_x + expected_y
    self.assertEqual(processor.x, expected_x)
    self.assertEqual(processor.y, expected_y)
    self.assertEqual(result, expected_result)

def test_x_negative_y_positive(self):
    processor = NumberProcessor(-20, 20)
    result = processor.process_numbers()
    expected_x = -20
    expected_y = 0
    expected_result = 2 * expected_x + expected_y
    self.assertEqual(processor.x, expected_x)
    self.assertEqual(processor.y, expected_y)
    self.assertEqual(result, expected_result)

def test_x_and_y_non_numeric(self):
    processor = NumberProcessor("zece", "douăzeci")
    with self.assertRaises(TypeError): # Presupunând că metoda ar trebui să arunce o excepție pentru
tipuri de date incorecte
        processor.process_numbers()

def test_repeated_execution_consistency(self):

```

```
processor = NumberProcessor(10, 20)

result1 = processor.process_numbers()

result2 = processor.process_numbers()

self.assertEqual(result1, result2)
```

Concluzie: Testele implementate și cele propuse de AI, oferă o evaluare completă a funcției `process_numbers`. Aceste teste demonstrează capacitatea funcției de a gestiona o varietate extinsă de scenarii de intrare, de la valori limită și cazuri extreme, până la intrări neconvenționale. Validarea atentă a comportamentului funcției în condiții diverse asigură că aceasta este robustă, stabilă și produce rezultate consistente.

III. Mutanti

Definiție: Testarea de mutații (mutation testing) este o tehnică de testare a calitatii testelor prin introducerea de modificări mici și controlate (numite mutații) în codul sursă și evaluarea dacă testele detectează aceste mutații. Scopul este de a evalua cât de eficiente sunt testele în detectarea erorilor în codul sursă.

Exemplu i) Avem codul de baza: Program P1

```
def process_numbers(self):
    """Metodă care procesează numerele x și y conform logicii specificate."""
    # Validăm că valorile hardcodate sunt întregi
    self.x = self.validate_integer(self.x)
    self.y = self.validate_integer(self.y)

    while self.x > 10:
        self.x = self.x - 10
        if self.x == 10:
            break

    if self.y < 20 and self.x % 2 == 0:
        self.y = self.y + 20
    else:
        self.y = self.y - 20

    return 2 * self.x + self.y
```

Prin efectuarea mutațiilor codul de baza se va modifica:

```

def process_numbers_mutant2(self):
    self.x = self.validate_integer(self.x)
    self.y = self.validate_integer(self.y)

    while self.x >= 10: # Mutant 1: schimbat > în >=
        self.x = self.x - 10
        if self.x == 10:
            break

    if self.y <= 20 and self.x % 2 == 1: # Mutant 2: schimbat < în <= și % 2 == 0 în % 2 == 1
        self.y = self.y + 20
    else:
        self.y = self.y - 20

    return 2 * self.x + self.y

```

Mutantul M1, de ordin 3 (de ordin mai mare (higher-order mutants))

Cu acest test detectăm modificarea făcută de mutantul

```

# higher order mutant detect.
def test_higher_order_mutants(self):
    processor = NumberProcessor(20, 20)
    result = processor.process_numbers()
    self.assertEqual(processor.x, 0) # Mutantul ar cauza bucla să continue
    self.assertEqual(result, -20)    # Verificăm rezultatul final

    processor = NumberProcessor(15, 19)
    result = processor.process_numbers()
    self.assertEqual(result, -5)     # Verificăm rezultatul final

```

Exemplu ii) Avem testul de baza:

Program P1

```

def process_numbers(self):
    """Metodă care procesează numerele x și y conform logicii specificate."""
    # Validăm că valorile hardcodate sunt întregi
    self.x = self.validate_integer(self.x)
    self.y = self.validate_integer(self.y)

    while self.x > 10:
        self.x = self.x - 10
        if self.x == 10:
            break

    if self.y < 20 and self.x % 2 == 0:
        self.y = self.y + 20
    else:
        self.y = self.y - 20

    return 2 * self.x + self.y

```

Prin efectuarea mutațiilor testul se va modifica:

Mutant M2 de ordin 1 (first-order)

```

def process_numbers_mutant1(self):
    self.x = self.validate_integer(self.x)
    self.y = self.validate_integer(self.y)

    while self.x >= 10: # Mutant: schimbat > în >=
        self.x = self.x - 10
        if self.x == 10:
            break

    if self.y < 20 and self.x % 2 == 0:
        self.y = self.y + 20
    else:
        self.y = self.y - 20

    return 2 * self.x + self.y

```

Cu acest test detectăm modificarea făcută de mutantul

```
# first order
def test_loop_condition_mutant(self):
    processor = NumberProcessor(10, 10)
    result = processor.process_numbers_mutant1()
    self.assertEqual(processor.x, 0) # Mutantul ar cauza bucla să continue
    self.assertEqual(result, -20)    # Verificăm rezultatul final

# weak mutant detect.
```

Pentru a verifica robustetea testelor vom folosi libraria Python "mutatest". Vom introduce mutatii automat in codul sursa si vom observa dacas testele sunt capabile sa detecteze aceste mutatii.

1. Instalăm Mutatest: folosim comanda 'pip install mutatest'.
2. Configurăm testele si codul sursa
3. Executăm Mutatest: folosim libraria prin comanda

```
$ mutatest -s . -t "python -m unittest testNumProc.py"
```

Observatie: putem specifica numarul de modificari pe care sa le faca mutatest, spre exemplu ii vom spune sa faca 100 modificari:

```
$ mutatest -s . -t "python -m unittest testNumProc.py" 100
```

OUTPUT:

```
● (tssenv) PS C:\Users\toshd\0Facultate\tssultima> mutatest -s . -t "python -m unittest testNumProc.py"
2024-05-11 15:22:34,950: Running clean trial
.....
-----
Ran 16 tests in 0.001s

OK
2024-05-11 15:22:35,085: 14 mutation targets found in NumProc.py AST.
2024-05-11 15:22:35,086: 18 mutation targets found in testNumProc.py AST.
```

Rapport final:

```

2024-05-11 15:22:39,055: Running clean trial
-----
Ran 16 tests in 0.001s

OK

2024-05-11 15:22:39,130: CLI Report:

Mutatest diagnostic summary
=====
- Source location: C:\Users\toshd\0\Facultate\tssultima
- Test commands: ['python', '-m', 'unittest', 'testNumProc.py']
- Mode: s
- Excluded files: []
- N locations input: 10
- Random seed: None

Random sample details
-----
- Total locations mutated: 10
- Total locations identified: 38391
- Location sample coverage: 0.03 %

Running time details
-----
- Clean trial 1 run time: 0:00:00.073070
- Clean trial 2 run time: 0:00:00.074478
- Mutation trials total run time: 0:00:03.903113

2024-05-11 15:22:39,131: Trial Summary Report:

Overall mutation trial summary
=====
- SURVIVED: 10
- TOTAL RUNS: 10
- RUN DATETIME: 2024-05-11 15:22:39.130886

2024-05-11 15:22:39,131: Detected mutations:

2024-05-11 15:22:39,132: Timedout mutations:

2024-05-11 15:22:39,132: Surviving mutations:

```

Observatie: mutatest a testat toate fisierele inclusiv virtual environmentul folosit in proiect ca drept urmare avem si cativa mutanti (irelevanti pentru proiect) care au supravietuit. *a se ignora*

[illegible]

Strong & Weak Mutation

Strong Mutation: aceste mutatii schimba logica programului intr-un mod semnificativ, ceea ce duce la o schimbare a rezultatului sau a comportamentului programului. Ele pun sub semnul intrebării corectitudinea codului și ar trebui să fie detectate de testele noastre.

Exemplu i)

Program P1

```
def process_numbers(self):
    """Metodă care procesează numerele x și y conform logicii specificate."""
    # Validăm că valorile hardcodate sunt întregi
    self.x = self.validate_integer(self.x)
    self.y = self.validate_integer(self.y)

    while self.x > 10:
        self.x = self.x - 10
        if self.x == 10:
            break

    if self.y < 20 and self.x % 2 == 0:
        self.y = self.y + 20
    else:
        self.y = self.y - 20

    return 2 * self.x + self.y
```

Mutatie Strong: exemplificare prin schimbarea unei conditii in bucla

Mutant M3

```
#mutant strong
def process_numbers_strong(self):
    self.x = self.validate_integer(self.x)
    self.y = self.validate_integer(self.y)

    while self.x > 10:
        self.x = self.x - 10
        if self.x == 10:
            break

    if self.y < 20 and self.x % 2 == 0:
        self.y = self.y + 20
    else:
        self.y = self.y + 20 # Mutant: schimbat - în +

    return 2 * self.x + self.y
```

Test pentru detectarea mutantului:

```
# strong mutant detect.
def test_strong_mutation(self):
    processor = NumberProcessor(10, 15)
    result = processor.process_numbers_strong()
    self.assertNotEqual(result, 2 * 10 + 35) # Detectăm schimbarea logicii
```

Weak Mutation: aceste mutatii nu schimba in mod semnificativ logica programului, ci mai exact aspectele periferice ale codului. Ele pot afecta numai anumite cazuri specifice sau aspecte neesentiale ale codului, cum ar fi modul in care sunt trate anumite exceptii sau ordinea operatiilor. Ele nu ar trebuie sa aiba un impact semnificativ asupra comportamentului programului.

Exemplu iii)

Program P1 de mai sus

Mutatie weak : Mutant M4

Consideram urmatorul operator de mutatie:

- * inlocuit de /


```
#mutant weak

def process_numbers_weak(self):
    self.x = self.validate_integer(self.x)
    self.y = self.validate_integer(self.y)

    while self.x > 10:
        self.x = self.x - 10
        if self.x == 10:
            break

    if self.y < 20 and self.x % 2 == 0:
        self.y = self.y + 20
    else:
        self.y = self.y - 20

    return 2 / self.x + self.y # Mutant: schimbat * în /
```

Cu acest test ar trebui sa detectam mutatia weak enuntata mai sus

```
# weak mutant detect.
def test_weak_mutation(self):
    processor = NumberProcessor(11, 19)
    result = processor.process_numbers()
    self.assertNotEqual(result, 2 * 1 - 1) # Detectăm schimbarea în operație

# mutanti neechivalenti
```

Mutanti Neechivalenti

Mutanti neechivalenti sunt versiuni modificate ale codului original care introduc erori subtile sau modificari in logica programului si care, prin urmare, nu sunt echivalente din punct de vedere functional cu codul original.

```

#mutant1 neechivalent
def process_numbers_neechivalent1(self):
    self.x = self.validate_integer(self.x)
    self.y = self.validate_integer(self.y)

    while self.x >= 10: # Mutant: schimbat > în >=
        self.x = self.x - 10
        if self.x == 10:
            break

    if self.y < 20 and self.x % 2 == 0:
        self.y = self.y + 20
    else:
        self.y = self.y - 20

    return 2 * self.x + self.y

```

```

#mutant2 neechivalent
def process_numbers_neechivalent2(self):
    self.x = self.validate_integer(self.x)
    self.y = self.validate_integer(self.y)

    while self.x > 10:
        self.x = self.x - 10
        if self.x == 10:
            break

    if self.y <= 20 and self.x % 2 == 0: # Mutant: schimbat < în <=
        self.y = self.y + 20
    else:
        self.y = self.y - 20

    return 2 * self.x + self.y

```

Vom adauga teste suplimentare pentru a omorî 2 dintre mutanii neechivalente.

```

# test suplimentar pt a omori mutant1 neechivalent
def test_while_loop_condition_neechivalent1(self):
    # Acest test eșuează dacă bucla se execută atunci când x este exact 10.
    processor = NumberProcessor(10, 10)
    result = processor.process_numbers_neechivalent1()
    self.assertEqual(processor.x, 10) # x nu ar trebui să fie schimbat
    self.assertEqual(processor.y, -10) # y ar trebui să fie scăzut cu 20
    self.assertEqual(result, 2 * 10 - 10) # Așteptăm ca rezultatul să fie 10

# test suplimentar pt a omori mutant2 neechivalent
def test_y_adjustment_condition(self):
    # Acest test eșuează dacă y este crescut atunci când y este exact 20.
    processor = NumberProcessor(10, 20)
    result = processor.process_numbers_neechivalent2()
    self.assertEqual(processor.y, 0) # y ar trebui să fie scăzut cu 20
    self.assertEqual(result, 2 * 10 + 0) # Așteptăm ca rezultatul să fie 20

```

Cele doua teste propuse viziteaza aspecte diferite ale clasei "NumberProcessor" si examineaza mutanti potenti neechivalenti prin abordarea a doua comportamente distincte ale metodei "process_numbers()".

BlackBox TESTING


```

def test_blackbox_case1(self):
    # Test atunci cand 'x' este mai mare decat 10 si 'y' este mai mic decat 20
    np = NumberProcessor( x: 15, y: 10)
    self.assertEqual(np.process_numbers(), second: 0)

```

Funcția "test_blackbox_case1" testează metoda process_numbers a clasei NumberProcessor într-o anumită condiție. Condiția este ca parametrul de intrare x să fie mai mare decât 10, iar parametrul de intrare y să fie mai mic de 20. Ieșirea așteptată a metodei numere_proces în această condiție este 0. Respectiv fiecare case de la 1 pana la 8 are o astfel de conditie si o astfel de iesire.

Funcția `self.assertEqual` este utilizată pentru a compara rezultatul real al metodei `numere_proces` cu rezultatul așteptat. Dacă rezultatul real se potrivește cu rezultatul așteptat, testul trece. Dacă nu, testul eșuează și este tipărit un mesaj de eroare.



```
def test_blackbox_case2(self):  
    # Test atunci cand 'x' este egal cu 10 si 'y' este mai mic decat 20  
    np = NumberProcessor( x: 10, y: 15)  
    self.assertEqual(np.process_numbers(), second: 55)
```

```
def test_blackbox_case3(self):  
    # Test atunci cand 'x' este mai mare decat 10 si 'y' este mai mare sau egal decat 20  
    np = NumberProcessor( x: 20, y: 30)  
    self.assertEqual(np.process_numbers(), second: 30)
```

```
def test_blackbox_case4(self):  
    # Test atunci cand 'x' este mai mic sau egal decat 10 si 'y' este mai mic decat 20 si 'x' este par  
    np = NumberProcessor( x: 6, y: 15)  
    self.assertEqual(np.process_numbers(), second: 47)
```

```
def test_blackbox_case5(self):  
    # Test atunci cand 'x' este mai mic sau egal decat 10 si 'y' este mai mic decat 20 si 'x' este impar  
    np = NumberProcessor( x: 7, y: 15)  
    self.assertEqual(np.process_numbers(), second: 9)
```

```
def test_blackbox_case6(self):  
    # Test atunci cand 'x' este mai mare decat 10 si 'y' este mai mic decat 20 si 'x' este impar  
    np = NumberProcessor( x: 17, y: 15)  
    self.assertEqual(np.process_numbers(), second: 9)
```

```
def test_blackbox_case7(self):  
    # Test atunci cand 'x' este mai mare decat 10 si 'y' este mai mic decat 20 si 'x' este par  
    np = NumberProcessor( x: 16, y: 15)  
    self.assertEqual(np.process_numbers(), second: 47)
```

```
def test_blackbox_case8(self):  
    # Test atunci cand 'x' este mai mare decat 10 si 'y' este mai mare sau egal decat 20  
    np = NumberProcessor( x: 20, y: 25)  
    self.assertEqual(np.process_numbers(), second: 25)
```

Acum putem sa facem comparatia si cu chatGPT :

```
import unittest
from number_processor import NumberProcessor

class TestNumberProcessor(unittest.TestCase):
    def test_process_numbers_case1(self):
        # Test when x is greater than 10 and y is less than 20
        np = NumberProcessor(15, 10)
        self.assertEqual(np.process_numbers(), 50)

    def test_process_numbers_case2(self):
        # Test when x is equal to 10 and y is less than 20
        np = NumberProcessor(10, 15)
        self.assertEqual(np.process_numbers(), 30)

    def test_process_numbers_case3(self):
        # Test when x is greater than 10 and y is greater than or equal to 20
        np = NumberProcessor(20, 30)
        self.assertEqual(np.process_numbers(), 60)

    def test_process_numbers_case4(self):
        # Test when x is less than or equal to 10 and y is less than 20 and x is even
        np = NumberProcessor(5, 15)
        self.assertEqual(np.process_numbers(), 0)

    def test_process_numbers_case5(self):
        # Test when x is less than or equal to 10 and y is less than 20 and x is odd
        np = NumberProcessor(7, 15)
        self.assertEqual(np.process_numbers(), -10)
```

```

def test_process_numbers_case6(self):
    # Test when x is greater than 10 and y is less than 20 and x is odd
    np = NumberProcessor(17, 15)
    self.assertEqual(np.process_numbers(), 50)

def test_process_numbers_case7(self):
    # Test when x is greater than 10 and y is less than 20 and x is even
    np = NumberProcessor(16, 15)
    self.assertEqual(np.process_numbers(), 54)

def test_process_numbers_case8(self):
    # Test when x is greater than 10 and y is greater than or equal to 20
    np = NumberProcessor(20, 25)
    self.assertEqual(np.process_numbers(), 70)

def test_process_numbers_case9(self):
    # Test when x is less than 10 and y is less than 20
    np = NumberProcessor(7, 10)
    self.assertEqual(np.process_numbers(), 4)

def test_process_numbers_case10(self):
    # Test when x is less than 10 and y is greater than or equal to 20
    np = NumberProcessor(7, 20)
    self.assertEqual(np.process_numbers(), -10)

def test_process_numbers_case11(self):
    # Test when x is equal to 0 and y is less than 20
    np = NumberProcessor(0, 15)
    self.assertEqual(np.process_numbers(), -20)

```

```

def test_process_numbers_case12(self):
    # Test when x is equal to 0 and y is greater than or equal to 20
    np = NumberProcessor(0, 20)
    self.assertEqual(np.process_numbers(), 0)

if __name__ == '__main__':
    unittest.main()

```

Link demo: https://youtu.be/3_bo-nPGO6U

