
프로젝트 2, 3

보고서

전북대학교

소프트웨어공학

팀 연효

김용현(201818716), 유지호(202111581), 이재연(202112043)
공리(201812722), 정재호(201911898), 김제현(201918770)

목차

목차	2
1. 코드 완성본	3
1.1 최종 코드	3
1.2 코드 실행 스크린 샷	6
2. 프로젝트 최종 Gantt Chart	6
2.1 Gantt Chart	6
3. 테스트	7
3.1 유닛 테스트	7
3.1.1 덧셈	7
3.1.2 뺄셈	8
3.1.3 곱셈	9
3.1.4 팩토리얼	9
4. TDD 과정	10
4.1 TDD 중간 과정	10
4.2 유닛테스팅 코드	10
4.3 팩토리얼 코드 작성	11
5. 인스펙션 리포트	12
5.1 Operator 클래스 부여 (calculator.py)	12
5.2 모듈화와 파일 분할 (operatorasm.py)	12
5.3 오류 처리 방법	13
5.4 함수 기반 구조 (calculator.py)	14
5.5 입력 검증 로직 (calculator.py)	14
5.6 주석 사용과 네이밍의 중요성 (calculator.py)	15
6. Lesson Learned	16
6.1 어려웠던 점	16
6.2 배운 점	16
7. 기타 (Github, Notion)	17

1. 코드 완성본

1.1 최종 코드

calculator.py 코드 (계산기 코드)

```
#!/usr/bin/env python3
from constants import EASTER_EGG_CODE_1, EASTER_EGG_CODE_2, EASTER_EGG_CODE_3
from exceptions import EasterEggException_1, EasterEggException_2, EasterEggException_3, InputException, InvalidOperatorException
from operatorasm import *
from utils import is_integer

def get_user_input():
    user_inputs = []
    user_input = None

    # 동호가 입력할 때까지 반복
    while True:
        # 사용자 입력을 문자열 형태로 저장
        user_input = input()
        if user_input == '\n':
            break

        # 사용자 입력을 inputs 리스트에 삽입
        user_inputs.append(user_input)

        # 이스터에그 코드가 입력된 경우 즉시 반환
        if user_input == EASTER_EGG_CODE_1:
            raise EasterEggException_1
        elif user_input == EASTER_EGG_CODE_2:
            raise EasterEggException_2
        elif user_input == EASTER_EGG_CODE_3:
            raise EasterEggException_3

    return user_inputs

def check_input(user_inputs):
    # 사용자 입력이 없는 경우 ('>만 입력된 경우)
    is_empty = len(user_inputs) == 0 # 이 줄이 누락되었을 수 있습니다.

    # 마지막 입력이 연산자인 경우 (팩토리얼 제외)
    is_operator_last_elem = not is_empty and user_inputs[-1] in Operator.operate and user_inputs[-1] != '!'

    # 모든 홀수 번째 요소들이 연산자인 경우
    is_integer_even_elem = all(is_integer(element) for element in user_inputs[0::2])

    if any((element == '/') for element in user_inputs[1::2]):
        raise InvalidOperatorException

    # 모든 홀수 번째 요소들이 연산자인 경우 (팩토리얼 포함)
    is_operator_odd_elem = all((element in Operator.operate) for element in user_inputs[1::2])

    # 팩토리얼 연산자 '!'가 올바르게 사용되었는지 검증하는 로직
    is_factorial_used_correctly = all(
        (i == 0 or user_inputs[i-1] not in Operator.operate) and
        (i == len(user_inputs) - 1 or user_inputs[i+1] not in Operator.operate)
        for i, element in enumerate(user_inputs) if element == '!'
    )

    if (is_empty or
        is_operator_last_elem or
        not (is_integer_even_elem and is_operator_odd_elem) or
        not is_factorial_used_correctly):
        raise InputException

def calculate(user_inputs):
    result = None # 계산 결과
    operand = None # 피연산자
    operator = None # 연산자

    for user_input in user_inputs:
        # 정수가 입력된 경우
        if is_integer(user_input):
            operand = int(user_input)
            if operator:
                # 연산자가 팩토리얼일 경우
                if operator == '!':
                    result = Operator.operate[operator](operand)
                    operand = None
                else:
                    result = Operator.operate[operator](result, operand)
                    operand = None
            else:
                result = operand

        # 연산자가 입력된 경우
        elif user_input in Operator.operate:
            if user_input == '!':
                if operand is not None:
                    result = Operator.operate[user_input](operand)
                    operand = None
            else:
                operator = user_input

    return result

def run_calculator():
    try:
        user_inputs = get_user_input()
    except (EasterEggException_1, EasterEggException_2, EasterEggException_3) as message:
        return message

    try:
        check_input(user_inputs)
    except (InputException, InvalidOperatorException) as message:
        return message

    try:
        result = calculate(user_inputs)
    except OutOfRangeException as message:
        return message

    return result

def main():
    print(run_calculator())

if __name__ == "__main__":
    main()
```

constants.py 코드(에러 출력 코드)

```
# constants.py

# 상수
ERROR_MESSAGE_INVALID_OPERATOR = "[SYSTEM] Invalid Operator"
ERROR_MESSAGE_INPUT_ERROR = "[SYSTEM] Input Error"
ERROR_MESSAGE_OUT_OF_RANGE = "[SYSTEM] Out Of Range"

EASTER_EGG_CODE_1 = '987654321987654321'
EASTER_EGG_CODE_2 = '7503'
EASTER_EGG_CODE_3 = '1015'

EASTER_EGG_MESSAGE_1 = "[EVENT] \"Hello! This Is Team Yeonhyo Easter Egg!!\""
EASTER_EGG_MESSAGE_2 = "[EVENT] \"안녕! 7503은 사용할 수 없는 숫자야\""
EASTER_EGG_MESSAGE_3 = "[EVENT] 전복대 개교기념일입니다."
```

exception.py 코드 (에러 정의)

```
# exceptions.py

from constants import EASTER_EGG_MESSAGE_1, EASTER_EGG_MESSAGE_2, EASTER_EGG_MESSAGE_3, ERROR_MESSAGE_OUT_OF_RANGE, ERROR_MESSAGE_INPUT_ERROR, ERROR_MESSAGE_INVALID_OPERATOR

# 이스터에그 에러 정의
class EasterEggException_1(Exception):
    def __init__(self, message = EASTER_EGG_MESSAGE_1):
        super().__init__(message)

class EasterEggException_2(Exception):
    def __init__(self, message = EASTER_EGG_MESSAGE_2):
        super().__init__(message)

class EasterEggException_3(Exception):
    def __init__(self, message = EASTER_EGG_MESSAGE_3):
        super().__init__(message)

# 입력 오류 에러 정의
class InputException(Exception):
    def __init__(self, message = ERROR_MESSAGE_INPUT_ERROR):
        super().__init__(message)

# 범수 범위리탈 에러 정의
class OutOfRangeException(Exception):
    def __init__(self, message = ERROR_MESSAGE_OUT_OF_RANGE):
        super().__init__(message)

class InvalidOperatorException(Exception):
    def __init__(self, message = ERROR_MESSAGE_INVALID_OPERATOR):
        super().__init__(message)
```

operatorasm.py 코드

```
# operatorasm.py
import math
from exceptions import InputException, OutOfRangeException
from utils import is_integer

class Operator: # 연산자 클래스 Calculator_operator로 정의.
    def add(num1, num2): # 덧셈.
        if not (is_integer(num1) and is_integer(num2)):
            raise InputException
        return int(num1) + int(num2)

    def sub(num1, num2): # 뺄셈.
        if not (is_integer(num1) and is_integer(num2)):
            raise InputException
        return int(num1) - int(num2)

    def mul(num1, num2): # 곱셈.
        if not (is_integer(num1) and is_integer(num2)):
            raise InputException
        return int(num1) * int(num2)

    def factorial(num):
        if not is_integer(num):
            raise InputException
        if int(num) < 0:
            raise OutOfRangeException
        return math.factorial(int(num))

# 연산자 메소드 디렉터리
operate = {
    '+': add,
    '-': sub,
    '*': mul,
    '!': factorial
}
```

utils.py 코드

```
import re

# 정규표현식을 이용한 정수 확인 함수
def is_integer(s):
    """Returns True if the string is an integer."""
    return re.match("[-]?\\d+$", str(s)) != None
```

test_opearator.py 코드

```
1 import sys
2 from os.path import dirname, realpath
3
4 parent_dir = dirname(dirname(realpath(__file__)))
5 sys.path.append(parent_dir)
6
7 from operatorasm import Operator
8 from exceptions import InputException, OutOfRangeException
9 import unittest
10
11 class TestAdd(unittest.TestCase):
12     """
13     Test the add function from the operatorasm library
14     """
15
16     # Validation Testing
17     def test_add_integers(self):
18         self.assertEqual(Operator.add(1, 2), 3)
19         self.assertEqual(Operator.add(0, 0), 0)
20         self.assertEqual(Operator.add(-4, -5), -9)
21         self.assertEqual(Operator.add(0, -3), -3)
22
23     # Defect Testing - floats
24     def test_add_floats(self):
25         self.assertRaises(InputException, Operator.add, 10.5, 2.2)
26
27     # Defect Testing - strings
28     def test_add_strings(self):
29         self.assertRaises(InputException, Operator.add, 'abc', 'def')
30
31 class TestSub(unittest.TestCase):
32     """
33     Test the sub function from the operatorasm library
34     """
35
36     # Validation Testing
37     def test_sub_integers(self):
38         self.assertEqual(Operator.sub(3, 2), 1)
39         self.assertEqual(Operator.sub(0, 0), 0)
```

```
40         self.assertEqual(Operator.sub(-4, -5), 1)
41         self.assertEqual(Operator.sub(0, -3), 3)
42
43     # Defect Testing - floats
44     def test_sub_floats(self):
45         self.assertRaises(InputException, Operator.sub, 10.5, 2.2)
46
47     # Defect Testing - strings
48     def test_sub_strings(self):
49         self.assertRaises(InputException, Operator.sub, 'abc', 'def')
50
51 class TestMul(unittest.TestCase):
52     """
53     Test the mul function from the operatorasm library
54     """
55
56     # Validation Testing
57     def test_mul_integers(self):
58         self.assertEqual(Operator.mul(3, 2), 6)
59         self.assertEqual(Operator.mul(0, 0), 0)
60         self.assertEqual(Operator.mul(-4, -5), 20)
61         self.assertEqual(Operator.mul(0, -3), 0)
62
63     # Defect Testing - floats
64     def test_mul_floats(self):
65         self.assertRaises(InputException, Operator.mul, 10.5, 2.2)
66
67     # Defect Testing -- strings
68     def test_mul_strings(self):
69         self.assertRaises(InputException, Operator.mul, 'abc', 'def')
70
71 class TestFact(unittest.TestCase):
72     """
73     Test the factorial function from the operatorasm library
74     """
75
76     # Validation Testing
77     def test_mul_integer(self):
78         self.assertEqual(Operator.factorial(5), 120)
```

```
71 class TestFact(unittest.TestCase):
72     """
73     Test the factorial function from the operatorasm library
74     """
75
76     # Validation Testing
77     def test_mul_integer(self):
78         self.assertEqual(Operator.factorial(5), 120)
79         self.assertEqual(Operator.factorial(3), 6)
80         self.assertEqual(Operator.factorial(0), 1)
81
82     # Defect Testing - negative
83     def test_factorial_negative(self):
84         self.assertRaises(OutOfRangeException, Operator.factorial, -1)
85
86     # Defect Testing - float
87     def test_factorial_float(self):
88         self.assertRaises(InputException, Operator.factorial, 10.5)
89
90     # Defect Testing -- string
91     def test_factorial_string(self):
92         self.assertRaises(InputException, Operator.factorial, 'abc')
93
94 if __name__ == '__main__':
95     unittest.main()
```

1.2 코드 실행 스크린 샷

```
@luMir-Laptop MINGW64 /d/Cloud_Git/_test/SWE_yeonhyo (main)
$ C:/Users/ /AppData/Local/Programs/Python/Python311/python.exe d:/Cloud_Git/_test/SWE_yeonhyo/calculator.py
0
|
=
1

@luMir-Laptop MINGW64 /d/Cloud_Git/_test/SWE_yeonhyo (main)
$ C:/Users/ /AppData/Local/Programs/Python/Python311/python.exe d:/Cloud_Git/_test/SWE_yeonhyo/calculator.py
3
|
=
6

@luMir-Laptop MINGW64 /d/Cloud_Git/_test/SWE_yeonhyo (main)
$ C:/Users/ /AppData/Local/Programs/Python/Python311/python.exe d:/Cloud_Git/_test/SWE_yeonhyo/calculator.py
-1
|
=
[SYSTEM] Out Of Range

@luMir-Laptop MINGW64 /d/Cloud_Git/_test/SWE_yeonhyo (main)
$ C:/Users/ /AppData/Local/Programs/Python/Python311/python.exe d:/Cloud_Git/_test/SWE_yeonhyo/calculator.py
3
5
|
=
[SYSTEM] Input Error
```

```
@luMir-Laptop MINGW64 /d/Cloud_Git/_test/SWE_yeonhyo (main)
$ C:/Users/ /AppData/Local/Programs/Python/Python311/python.exe d:/Cloud_Git/_test/SWE_yeonhyo/calculator.py
3
+
=
8
```

```
@luMir-Laptop MINGW64 /d/Cloud_Git/_test/SWE_yeonhyo (main)
$ C:/Users/ /AppData/Local/Programs/Python/Python311/python.exe d:/Cloud_Git/_test/SWE_yeonhyo/calculator.py
3
/
5
=
[SYSTEM] Invalid Operator
```

```
@luMir-Laptop MINGW64 /d/Cloud_Git/_test/SWE_yeonhyo (main)
$ C:/Users/ /AppData/Local/Programs/Python/Python311/python.exe d:/Cloud_Git/_test/SWE_yeonhyo/calculator.py
7503
[EVENT] "안녕! 7503은 사용할 수 없는 숫자야"
```

```
@luMir-Laptop MINGW64 /d/Cloud_Git/_test/SWE_yeonhyo (main)
$ C:/Users/ /AppData/Local/Programs/Python/Python311/python.exe d:/Cloud_Git/_test/SWE_yeonhyo/calculator.py
1015
[EVENT] 전복대 개교기념일입니다.
```

2. 프로젝트 최종 Gantt Chart

2.1 Gantt Chart

- 1일 차에 이스터에그와. 에러 변경사항을 추가하고 유닛테스팅을 함께 구현한다. T1과 T2가 끝나면 TDD용 테스트를 구현한다. TDD용 테스트 구현을 완료한 후 팩토리얼 코드 구현을 진행한다. 최종 코드 중 필요한 부분을 선정하여 인스펙션을 진행하고 보고서도 함께 작업을 시작한다. 인스펙션 리포트까지 보고서에 작성 완료한 후에 발표자료로 ppt를 만들고 이를 통해 발표 준비를 마친다.

day	1	2	3	4	5	6	7	8
T1								
이스터에그 추가								
T2								
에러 추가								
T3								
테스팅								
T4								
TDD용 테스트								
T5				(T4)				
팩토리얼 추가								
T6					(T1&T2&T5)			
인스펙션					(T5)			
T7								
보고서 작성								
T8							(T7)	
PPT 제작								
T9								(T8)
발표 준비								

3. 테스트

3.1 유닛 테스트

- Specification에 의하면 피연산자로 정수만 입력되는 것을 가정한다.
 - Validation Test에서는 정수가 입력된 경우에 대해 올바른 연산 결과를 반환하는지 확인한다. 양수, 0, 음수 partition에 대해 대표값을 뽑아 테스트한다.
 - Defect Test에서는 정수 외의 실수, 문자열이 입력된 경우 오류를 반환하는지 확인한다.

3.1.1 덧셈

- **Validation Test**
 - $1 + 2 = 3$: 양수 입력, 정상 결과 출력
 - $0 + 0 = 0$: 0 입력, 정상 결과 출력
 - $-4 + -5 = -9$: 음수 입력, 정상 결과 출력
 - $0 + -3 = -3$: 0과 0이 아닌 정수 입력, 정상 결과 출력
- **Defect Test**
 - 실수가 입력된 경우 => Error
 - 문자열이 입력된 경우 => Error

```

11 class TestAdd(unittest.TestCase):
12     """
13     Test the add function from the operatorasm library
14     """
15
16     # Validation Testing
17     def test_add_integers(self):
18         self.assertEqual(Operator.add(1, 2), 3)
19         self.assertEqual(Operator.add(0, 0), 0)
20         self.assertEqual(Operator.add(-4, -5), -9)
21         self.assertEqual(Operator.add(0, -3), -3)
22
23     # Defect Testing - floats
24     def test_add_floats(self):
25         self.assertRaises(InputException, Operator.add, 10.5, 2.2)
26
27     # Defect Testing - strings
28     def test_add_strings(self):
29         self.assertRaises(InputException, Operator.add, 'abc', 'def')
30

```

3.12 ■■■

- Validation Test

- $3 - 2 = 1$: 양수 입력. 정상 결과 출력
- $0 - 0 = 0$: 0 입력. 정상 결과 출력
- $-4 - -5 = 1$: 음수 입력. 정상 결과 출력
- $0 - -3 = 3$: 0과 0이 아닌 정수 입력. 정상 결과 출력

- Defect Test

- 실수가 입력된 경우 => Error
- 문자열이 입력된 경우 => Error

```

31 class TestSub(unittest.TestCase):
32     """
33     Test the sub function from the operatorasm library
34     """
35
36     # Validation Testing
37     def test_sub_integers(self):
38         self.assertEqual(Operator.sub(3, 2), 1)
39         self.assertEqual(Operator.sub(0, 0), 0)
40         self.assertEqual(Operator.sub(-4, -5), 1)
41         self.assertEqual(Operator.sub(0, -3), 3)
42
43     # Defect Testing - floats
44     def test_sub_floats(self):
45         self.assertRaises(InputException, Operator.sub, 10.5, 2.2)
46
47     # Defect Testing - strings
48     def test_sub_strings(self):
49         self.assertRaises(InputException, Operator.sub, 'abc', 'def')
50

```


3.1.3 곱셈

- Validation Test

- $3 * 2 = 6$: 양수 입력. 정상 결과 출력
- $0 * 0 = 0$: 0 입력. 정상 결과 출력
- $-4 * -5 = 20$: 음수 입력. 정상 결과 출력
- $0 * -3 = 0$: 0과 0이 아닌 정수 입력. 정상 결과 출력

- Defect Test

- 실수가 입력된 경우 => Error
- 문자열이 입력된 경우 => Error

```
51 class TestMul(unittest.TestCase):
52     """
53     Test the mul function from the operatorasm library
54     """
55
56     # Validation Testing
57     def test_mul_integers(self):
58         self.assertEqual(Operator.mul(3, 2), 6)
59         self.assertEqual(Operator.mul(0, 0), 0)
60         self.assertEqual(Operator.mul(-4, -5), 20)
61         self.assertEqual(Operator.mul(0, -3), 0)
62
63     # Defect Testing - floats
64     def test_mul_floats(self):
65         self.assertRaises(InputException, Operator.mul, 10.5, 2.2)
66
67     # Defect Testing -- strings
68     def test_mul_strings(self):
69         self.assertRaises(InputException, Operator.mul, 'abc', 'def')
70
```

3.1.4 팩토리얼

- Validation Test

- $5! = 120$: 양수 입력. 정상 결과 출력
- $3! = 6$: 양수 입력. 정상 결과 출력
- $0! = 1$: 0 입력. 정상 결과 출력

- Defect Test

- 음수가 입력된 경우 => Error
- 실수가 입력된 경우 => Error
- 문자열이 입력된 경우 => Error

```
71 class TestFact(unittest.TestCase):
72     """
73     Test the factorial function from the operatorasm library
74     """
75
76     # Validation Testing
77     def test_mul_integer(self):
78         self.assertEqual(Operator.factorial(5), 120)
79         self.assertEqual(Operator.factorial(3), 6)
80         self.assertEqual(Operator.factorial(0), 1)
81
82     # Defect Testing - negative
83     def test_factorial_negative(self):
84         self.assertRaises(OutOfRangeException, Operator.factorial, -1)
85
86     # Defect Testing - float
87     def test_factorial_float(self):
88         self.assertRaises(InputException, Operator.factorial, 10.5)
89
90     # Defect Testing -- string
91     def test_factorial_string(self):
92         self.assertRaises(InputException, Operator.factorial, 'abc')
```

4. TDD 과정

4.1 TDD 중간 과정



- TDD 개발을 위해 먼저 유닛테스팅을 위한 코드를 짜는 과정으로. 어떤 종류의 테스트가 필요한지 조원들과 함께 토의하는 상황이다. 불가피하게 불참한 조원들은 notion 및 카카오톡 채팅으로 프로젝트 진행 상황에 대해 바로 알 수 있었다.

4.2 유닛테스팅 코드

- test_operator.py 코드

```
1 import sys
2 from os.path import dirname, realpath
3
4 parent_dir = dirname(dirname(realpath(__file__)))
5 sys.path.append(parent_dir)
6
7 from operatorasm import Operator
8 from exceptions import InputException, OutOfRangeException
9 import unittest
10
11 class TestAdd(unittest.TestCase):
12     """
13     Test the add function from the operatorasm library
14     """
15
16     # Validation Testing
17     def test_add_integers(self):
18         self.assertEqual(Operator.add(1, 2), 3)
19         self.assertEqual(Operator.add(0, 0), 0)
20         self.assertEqual(Operator.add(-4, -5), -9)
21         self.assertEqual(Operator.add(0, -3), -3)
22
23     # Defect Testing - floats
24     def test_add_floats(self):
25         self.assertRaises(InputException, Operator.add, 10.5, 2.2)
26
27     # Defect Testing - strings
28     def test_add_strings(self):
29         self.assertRaises(InputException, Operator.add, 'abc', 'def')
30
31 class TestSub(unittest.TestCase):
32     """
33     Test the sub function from the operatorasm library
34     """
35
36     # Validation Testing
37     def test_sub_integers(self):
38         self.assertEqual(Operator.sub(3, 2), 1)
39         self.assertEqual(Operator.sub(0, 0), 0)
40
41     self.assertEqual(Operator.sub(-4, -5), 1)
42     self.assertEqual(Operator.sub(0, -3), 3)
43
44     # Defect Testing - floats
45     def test_sub_floats(self):
46         self.assertRaises(InputException, Operator.sub, 10.5, 2.2)
47
48     # Defect Testing - strings
49     def test_sub_strings(self):
50         self.assertRaises(InputException, Operator.sub, 'abc', 'def')
51
52 class TestMul(unittest.TestCase):
53     """
54     Test the mul function from the operatorasm library
55     """
56
57     # Validation Testing
58     def test_mul_integers(self):
59         self.assertEqual(Operator.mul(3, 2), 6)
60         self.assertEqual(Operator.mul(0, 0), 0)
61         self.assertEqual(Operator.mul(-4, -5), 20)
62         self.assertEqual(Operator.mul(0, -3), 0)
63
64     # Defect Testing - floats
65     def test_mul_floats(self):
66         self.assertRaises(InputException, Operator.mul, 10.5, 2.2)
67
68     # Defect Testing -- strings
69     def test_mul_strings(self):
70         self.assertRaises(InputException, Operator.mul, 'abc', 'def')
71
72 class TestFact(unittest.TestCase):
73     """
74     Test the factorial function from the operatorasm library
75     """
76
77     # Validation Testing
78     def test_mul_integer(self):
79         self.assertEqual(Operator.factorial(5), 120)
```

```

71 class TestFact(unittest.TestCase):
72     """
73     Test the factorial function from the operatorasm library
74     """
75
76     # Validation Testing
77     def test_mul_integer(self):
78         self.assertEqual(Operator.factorial(5), 120)
79         self.assertEqual(Operator.factorial(3), 6)
80         self.assertEqual(Operator.factorial(0), 1)
81
82     # Defect Testing - negative
83     def test_factorial_negative(self):
84         self.assertRaises(OutOfRangeException, Operator.factorial, -1)
85
86     # Defect Testing - float
87     def test_factorial_float(self):
88         self.assertRaises(InputException, Operator.factorial, 10.5)
89
90     # Defect Testing -- string
91     def test_factorial_string(self):
92         self.assertRaises(InputException, Operator.factorial, 'abc')
93
94 if __name__ == '__main__':
95     unittest.main()

```

4.3 팩토리얼 코드 작성

```

def calculate(user_inputs):
    result = None # 계산 결과
    operand = None # 피연산자
    operator = None # 연산자


    for user_input in user_inputs:
        # 정수가 입력된 경우
        if is_integer(user_input):
            operand = int(user_input)
            if operator:
                # 연산자가 팩토리얼일 경우
                if operator == '!':
                    result = Operator.operate[operator](operand)
                    operand = None
                else:
                    result = Operator.operate[operator](result, operand)
                    operand = None
            else:
                result = operand
        # 연산자가 입력된 경우
        elif user_input in Operator.operate:
            if user_input == '!':
                if operand is not None:
                    result = Operator.operate[user_input](operand)
                    operand = None
            else:
                operator = user_input

    return result

```

5. 인스펙션 리포트

5.1 Operator 클래스 부여 (calculator.py)

```
1 from constants import EASTER_EGG_CODE_1, EASTER_EGG_CODE_2, EASTER_EGG_CODE_3
2 from exceptions import EasterEggException_1, EasterEggException_2, EasterEggException_3, InputException, InvalidOperatorException
3 from operatorasm import 
4 from utils import is_integer
```

- 해당 calculator 프로그램의 3행의 코드에서는 'operator' 클래스 즉 연산자 클래스를 사용하고 있지만 'operatorasm' 모듈에서 'operator' 클래스를 호출하는 부분을 구체적으로 명시하지 않았기 때문에 'operator'를 임포트를 해야 할 필요성이 부여된다. 다음과 같은 예시로 구체적으로 명시를 할 수 있다.

```
1 from constants import EASTER_EGG_CODE_1, EASTER_EGG_CODE_2, EASTER_EGG_CODE_3
2 from exceptions import EasterEggException_1, EasterEggException_2, EasterEggException_3, InputException, InvalidOperatorException
3 from operatorasm import Operator
4 from utils import is_integer
```

- 다음과 예시와 같이 구체적으로 'operator' 클래스를 임포트 함을 통해 calculator 프로그램의 클래스를 2행뿐 아니라 3행에서 'operatorasm' 모듈에서 'operator' 클래스를 임포트 했음을 구체적으로 확인 할 수 있다.

5.2 모듈화와 파일 분할 (operatorasm.py)

Jaeho-Jung fix unit test but fix 72a02650 · 8 hours ago 39 Commits		
tests	fix: unit test but fix	8 hours ago
.gitignore	chore(.gitignore): Fixed untracked files.	11 hours ago
calculator.code-workspace	공백 추가, 연산자 클래스 선언, 예외처리 루틴 구현, 코드 간...	last month
calculator.py	feat: handle invalid operator exception	9 hours ago
constants.py	feat: handle invalid operator exception	9 hours ago
exceptions.py	feat: handle invalid operator exception	9 hours ago
<u>operatorasm.py</u>	fix: handle negative factorial, input exceptions	9 hours ago
utils.py	test: add unit tests of Operator class	10 hours ago

```
1 # operatorasm.py
2 import math
3 from exceptions import InputException, OutOfRangeException
4 from utils import is_integer
5
6 class Operator: # 연산자 클래스 Calculator_operator로 정의.
7     def add(num1, num2): # 덧셈.
8         if not (is_integer(num1) and is_integer(num2)):
9             raise InputException
10        return int(num1) + int(num2)
11
12    def sub(num1, num2): # 뺄셈.
13        if not (is_integer(num1) and is_integer(num2)):
14            raise InputException
15        return int(num1) - int(num2)
16
17    def mul(num1, num2): # 곱셈.
18        if not (is_integer(num1) and is_integer(num2)):
19            raise InputException
20        return int(num1) * int(num2)
21
22    def factorial(num):
23        if not is_integer(num):
24            raise InputException
25        if int(num) < 0:
26            raise OutOfRangeException
27        return math.factorial(int(num))
28
29 # 연산자 메소드 디렉터리
30 operate = {
31     '+': add,
32     '-': sub,
33     '*': mul,
34     '!': factorial
35 }
```

- 기능별로 적절히 모듈화되어 있고, 각 모듈은 별도의 파일로 구성되어있다. 이러한 파일 분할 방식은 코드의 관리와 유지보수를 용이하게 한다.
- 새로운 기능을 추가하거나 기존 기능을 수정할 때, 관련된 파일만을 수정하면 되므로 전체 코드에 대한 광범위한 변경을 최소화할 수 있다. 프로젝트의 이러한 구조는 향후 발생할 수 있는 다양한 요구사항 변화에도 유연하게 대응할 수 있다.
- 예시로 `operatorasm.py`는 위의 `calculator.py`의 연산자의 로직을 구현하는 클래스를 구성함으로써 해당 팀프로젝트 같은 프로그램을 만들 때 같이 일반적으로 여러 사람이 작업을 진행하는 상황에서는 전체 프로그램을 모듈별로 설계하고 개인별로 나누어 코딩한 후 전체 모듈을 통합할 필요가 있다. 이처럼 모듈별로 구분해 코드를 작성하면 자신이 맡은 모듈만 신경 쓰면 되므로 공동 작업으로 인한 복잡성이 줄고 효율성 증가에 효과적이다.

5.3 오류 처리 방법

```

1  try:
2      user_inputs = get_user_input()
3  except (EasterEggException_1, EasterEggException_2, EasterEggException_3) as message:
4      return message
5
6  try:
7      check_input(user_inputs)
8  except (InputException, InvalidOperatorException) as message:
9      return message
10
11 try:
12     result = calculate(user_inputs)
13 except OutOfRangeException as message:
14     return message

```

- 프로젝트에서 'try-except' 블록은 주요 함수들에서 예외 상황을 처리하는 데 사용되며 이는 예기치 못한 오류로부터 프로그램을 보호하고, 사용자에게 보다 명확한 피드백을 제공할 수 있다.
- 사용자 친화적 오류 메시지: 오류가 발생했을 때 사용자에게 이해하기 쉬운 메시지를 제공함으로써, 사용자는 오류의 원인을 보다 쉽게 파악하고 적절한 조치를 취할 수 있다.
- 프로젝트에서 'try-except' 블록을 통한 오류처리는 프로그램의 전반적인 신뢰성을 크게 향상시킨다. 오류가 격리되고 사용자에게 명확한 피드백이 제공되며, 예기치 않은 상황에서의 프로그램이 안정적으로 실행할 수 있게끔 한다. 이는 특히 프로젝트의 신뢰도와 사용자 만족도를 높이는 데 있어 중요하다.

5.4 함수 기반 구조 (calculator.py)

```
1 def calculate(user_inputs):
2     result = None # 계산 결과
3     operand = None # 피연산자
4     operator = None # 연산자
5
6     for user_input in user_inputs:
7         # 정수가 입력된 경우
8         if is_integer(user_input):
9             operand = int(user_input)
10            if operator:
11                # 연산자가 팩토리얼일 경우
12                if operator == '!':
13                    result = Operator.operate(operator)(operand)
14                    operand = None
```

- 함수를 사용하여 기능을 구현함으로써, 코드를 확장하거나 수정하기가 용이하다. 각 함수는 특정 기능을 캡슐화하고, 이는 코드의 모듈성을 향상시킨다. 함수 기반 구조를 사용함으로써 한 부분을 변경하거나 확장하는 데 다른 부분이 영향을 받지 않는다.
- 잘 정의된 함수는 다른 부분의 코드에서 재사용될 수 있다. 이는 코드 중복을 줄이고, 일관된 기능 구현을 하기 용이하다.
- 새로운 기능을 추가하거나 기존 기능을 확장하고자 할 때, 새로운 함수를 추가하거나 기존 함수를 수정하기만 하면 되므로 전체 시스템의 변경을 최소화할 수 있다.

5.5 입력 검증 로직 (calculator.py)

```
1 def check_input(user_inputs):
2     # 사용자 입력이 없는 경우 (' '만 입력된 경우)
3     is_empty = len(user_inputs) == 0 # 이 줄이 누락되었을 수 있습니다.
4
5     # 마지막 입력이 연산자인 경우 (팩토리얼 제외)
6     is_operator_last_elem = not is_empty and user_inputs[-1] in Operator.operate and user_inputs[-1] != '!'
7
8     # 모든 짝수 번째 요소들이 정수인 경우
9     is_integer_even_elem = all(is_integer(element) for element in user_inputs[0::2])
10
11     if any((element == '/') for element in user_inputs[1::2]):
12         raise InvalidOperatorException
13
14     # 모든 홀수 번째 요소들이 연산자인 경우 (팩토리얼 포함)
15     is_operator_odd_elem = all((element in Operator.operate) for element in user_inputs[1::2])
16
17
```

- 위 코드는 기본적인 입력 검증만을 수행한다. 주로 사용자 입력이 올바른 형식과 타입을 가지고 있는지 확인하는데 초점을 맞추고 있다.
- 현재 검증 로직은 기본적인 요구사항은 충족하지만, 추후 더 다양한 시나리오와 복잡한 입력 조건을 처리할 수 있도록 확장할 필요가 있다. 이러한 확장은 프로그램이 더 복잡한 입력 요구사항을 수용하고, 프로젝트의 견고성을 향상 시키며 전체 소프트웨어의 품질을 향상시킬 것이다.

5.6 주석 사용과 네이밍의 중요성 (calculator.py)



```
1 def calculate(user_inputs):  
2     result = None # 계산 결과  
3     operand = None # 피연산자  
4     operator = None # 연산자
```

1. 주석의 역할과 중요성

- 주석은 코드의 이해를 돕고, 코드의 목적과 동작 방식을 설명하는 데 중요한 역할을 수행한다.
- 좋은 주석은 개발자가 코드를 더 빨리 이해하고, 유지보수를 쉽게 할 수 있도록 돕는다.

2. 네이밍의 중요성

- 변수와 함수의 이름은 코드의 이해에 중요한 역할을 한다. 명확하고 일관된 이름은 코드를 읽고 이해하는 데 있어 크게 도움을 준다.
- 위 프로젝트에서는 변수와 함수의 이름이 명확하고 일관성 있게 사용되었으며 이는 각 변수와 함수의 역할을 쉽게 이해할 수 있게 하고 코드 전반의 가독성을 향상시킨다.

6. Lesson Learned

6.1 어려웠던 점

1. 서로 충돌하는 요구사항의 이해와 해석

- 프로젝트의 지침과 요구사항에 대한 다양한 해석과 이해의 차이로 인해 팀원간 의사소통 문제가 생겼다. 예를 들어 팩토리얼 계산 기능에서 요구사항이 각기 다르게 해석되어, 팀 내 일관된 방향성을 설정하는 데 어려움을 겪었다.

2. 역할 분담의 부재

- 명확한 역할 분담이 없어, 팀원들이 무의식적으로 중복된 작업을 수행하는 일이 발생했다. 이는 팀의 효율성을 저하시키고, 프로젝트 진행에 있어 혼란을 야기했다.

6.2 배운 점

1. 프로젝트 수행의 다양성 인식

- 이번 경험을 통해 프로젝트를 수행하는 방법이 매우 다양하다는 것을 깨달았으며, 큰 규모의 프로젝트 경험이 부족했기 때문에 이전에는 이런 다양성을 인지하지 못했다.

2. 의사소통의 중요성

- 팀원 간의 활발한 의사소통의 필요성을 깨달았으며 명확하고 효과적인 소통은 요구사항의 정확한 이해와 올바른 역할 분담에 있어 매우 중요하다는 것을 알 수 있었다.

3. 명확한 지침과 계획의 중요성

- 프로젝트를 시작하기 전에 구체적인 지침과 계획을 세우는 것의 중요성을 깨달았고 이를 통해 혼란을 최소화하고 팀의 일관성과 효율성을 높일 수 있음을 알게 되었다.

4. 역할 분담의 체계화

- 각 팀원의 역할과 책임을 명확히 정의하고, 정기적인 진행 상황 점검을 통해 중복 작업을 방지하는 시스템의 필요성을 배울 수 있는 계기가 되었다.
- 각 역할에 맞게 할당된 작업을 팀원들이 부여 받으며, 프로그램을 TDD방향성에 따라 일관성 있게 유지함으로써 계획 주도 개발에 이점들을 확인하며 프로젝트를 진행 할 수 있었다.

7. 71E1 (Github, Notion)

- Github Link: https://github.com/dlsrks0631/SWE_yeonhyo

test: change test cases Jaeho-Jung committed 2 hours ago	cc44ba6		
test: add test case Jaeho-Jung committed 2 hours ago	02cae6d		
Commits on Dec 17, 2023			
fix: unit test but fix Jaeho-Jung committed 2 days ago	72d0b50		
fix: test_factorial_negative() Jaeho-Jung committed 2 days ago	7636f07		
refactor: rename unit test funcitons Jaeho-Jung committed 2 days ago	8866e2f		
feat: handle invalid operator exception Jaeho-Jung committed 2 days ago	0b1a532		
fix: handle negative factorial, input exceptions Jaeho-Jung committed 2 days ago	e8f6b65		
test: add unit tests of Operator.factorial() Jaeho-Jung committed 2 days ago	95d34ff		
test: add unit test of Operator class - negative Jaeho-Jung committed 2 days ago	c8c6dc1		

test: add unit test of Operator class - negative Jaeho-Jung committed 2 days ago	c8c6dc1		
test: add unit tests of Operator class Jaeho-Jung committed 2 days ago	ba3a9ec		
chore(.gitignore): Fixed untracked files. luminlumin committed 2 days ago	a0c1549		
feat(Exception): Add new exceptions(exception 1-3), mod error message luminlumin committed 2 days ago	795befd		
Commits on Dec 15, 2023			
팩토리얼 연산 기능 추가 dlsrks0631 committed 4 days ago	b4e8b96		
팩토리얼 연산 기능 추가 dlsrks0631 committed 4 days ago	b9f566b		

- Notion Link: <https://www.notion.so/2-16af1041f6704c97939dac44d6853b7a>