Fundamentele programării

Organizare pe funcții și module

Scop:

- Mai ușor să regăsim o parte de cod care implementează un anumit lucru. Fiecare concept are un loc bine definit.
- Mai ușor de adăugat funcționalități noi
- Mai ușor de testat automat.
- Permite colaborarea.

Responsabilități

- responsabilitate pentru o funcție: efectuarea unui calcul
- responsabilitate modul. responsabilitățile tuturor funcțiilor din modul

Principiul unei singure responsabilități (SRP)

O funcție / modul trebuie să aibă o singură responsibilitate

```
#Function with multiple responsibilities
#implement user interaction (read/print)
#implement a computation (filter)
def filterScore():
    st = input("Start score:")
    end = input("End score:")
    for c in l:
        if c[1]>st and c[1]<end:
            print (c)
```

Multiple responsabilități duc la: dificultăți de înțelegere, imposibilitatea de a testa/a refolosi, dificultăți la întreținere/evoluție

Separation of concerns (SoC)

Principiul separării responsabilităților este procesul de separare a unui program în responsabilități care nu se suprapun.

```
def filterScoreUI():
    st = input("Start sc:")
    end = input("End sc:")
    rez = filterScore(l,st, end)
    for e in rez:
        print (e)


def filterScore(l,st, end):
    """
    filter participants
    l - list of participants
    st, end - integers -scores
    return list of participants
        filtered by st end score
    """
    rez = []
    for p in l:
        if getScore(p)>st and
                getScore(p)<end:
            rez.append(p)
    return rez
```

```
def testScore():
    l = [["Ana", 100]]
    assert filterScore(l,10,30)==[]
    assert filterScore(l,1,30)==l
    l = [["Ana", 100],["Ion", 40],["P", 60]]
    assert filterScore(l,3,50)==[["Ion", 40]]
```

Dependențe

- funcția : apelează o altă funcție
- modul : funcția din modul apelează o funcție din alt modul

Pentru a ușura întreținerea aplicației este nevoie de gestiunea dependențelor

Separarea interfeței de implementare

Interfața funcției : signatura funcției + specificații
Interfața modul: signatura modulului + specificațiile tuturor funcțiilor din el
Codul client – codul care folosește funcția modulului
Codul client nu ar trebui să depindă de detalii de de implementare, ele sunt reprezentate în interiorul funcției.

```
#Vers1
calc = reset()
add_to(calc, 1, 3)
print(get_total(calc))
undo(calc)
print(get_total(calc))
calc = reset ()
add_to(calc, 1, 3)
add_to(calc, 1, 3)
add_to(calc, 1, 3)
print(get_total(calc))
```

```
#Vers Bad
ca = [[0,1],[]]
ca[1].append(ca[0])
ca[0] = add(ca[0][0],ca[0][1],1,3)
print(ca[0])
ca[0] = ca[1].pop()
print(ca[0])
ca[1].clear()
ca[0] =[0,1]
ca[0] = add(ca[0][0],ca[0][1],1,3)
ca[0] = add(ca[0][0],ca[0][1],1,3)
ca[0] = add(ca[0][0],ca[0][1],1,3)
print(calc[0])
```

Cuplare

Măsoară intensitatea legăturilor dintre module/funcții
Cu cât există mai multe conexiuni între module cu atât modulul este mai greu de înțeles, întreținut, refolosit => cu cât gradul de cuplare este mai scăzut cu atât mai bine.

Coeziunea

Măsoară cât de relaționate sunt task-urile din program. Un modul puternic coezIV ar trebui să realizeze o singură sarcină și să necesite interacțiuni minime cu alte module ale programului.
Modulul poate avea:

- Grad de coeziune ridicat
- Grad de coeziune scăzut

Dacă elementele modulului implementează responsabilități care nu sunt înrudite atunci modulul este mai greu de înțeles / întreținut => modulele trebuie să aibă grad de coeziune ridicat.

Arhitectura stratificată

Structurarea aplicației trebuie să:

- minimizeze cuplarea între module (modulele nu trebuie să cunoască detalii despre alte module)

- maximizeze coeziunea pentru module (conținutul unui modul izolează un concept bine definit)

Straturile arhitecturii au o interfață bine definită:

- User Interface (Nivel prezentare)
  - implementează interfața utilizator (funcții / module / clase)
- Domain (Nivel logic)
  - oferă funcții determinate de cazurile de utilizare
  - implementează concepte din domeniul aplicației
- Infrastructure
  - funcții / module / clase generale, utilitare
- Application coordinator
  - asamblează și pornește aplicația

Exemplu:

```
#Ui
def filterScoreUI():                          #manage the user interaction
    st = input("Start sc:")
    end = input("End sc:")
    rez = filterScoreDomain(st, end)
    for e in rez:
        print (e)
```

```
#domain
all = [["Ion",50],["Ana",30],["Pop",100]]
def filterScoreDomain(all,st, end):                #filter the score board
    if end<st: return []
    rez = filterMatrix(1, 1, st, end)
    return rez
```

```
#Utility function - infrastructure
def filterMatrix(matrice, col, st, end):       #filter matrix lines
    linii = []
    for linie in matrice:
        if linie[col]>st and linie[col]<end:
            linii.append(linie)
    return linii
```

Erori și excepții

Erori de sintaxă:

```
while True print("Ceva"):
    pass
```

```
  File "d:\wsp\hhh\aa.py", line 1
    while True print("Ceva"):
                     ^
SyntaxError: invalid syntax
```

Codul nu este corect sintactic

Erori detectate în timpul rulării:

```
>>> x=0
>>> print 10/x

Trace back (most recent call last):
  File "<pyshell#1>", line 1, in <module>
    print 10/x
ZeroDivisionError: integer division or modulo by zero
```

Excepțiile sunt aruncate atunci când o eroare este detectată:

- pot fi aruncate de interpretorul python
- de funcții pentru a semnala o situație, o eroare

```
def rational_add(a1, a2, b1, b2):
    """
    Return the sum of two rational numbers.
    a1,a2,b1,b2 integer numbers, a2<>0 and b2<>0
    return a list with 2 int, representing a rational number a1/b2 + b1/b2
    Raise ValueError if the denominators are zero.
    """
    if a2 == 0 or b2 == 0:
        raise ValueError("0 denominator not allowed")
    c = [a1 * b2 + a2 * b1, a2 * b2]
    d = gcd(c[0], c[1])
    c[0] = c[0] / d
    c[1] = c[1] / d
    return c
```

Tratarea excepțiilor

```
try:
    #code that may raise exceptions
    pass
except ValueError:
    #code that handle the error
    pass
```

Excepțiile pot fi tratate în blocul în care a apărut excepția sau în orice bloc exterior care în mod direct sau indirect a apelat blocul în care a apărut excepția

```
try:
    calc_add (int(m), int(n))
    printCurrent()
except ValueError:
    print ("Enter integers for m, n, with n!=0")
```

Tratarea selectivă a excepțiilor

- avem mai multe clauze **except**
- clauza **finally** se execută în orice condiții
- putem arunca excepții proprii folosind **raise**

```
def impartire(a, b):
    try:
        rezultat = a / b
        print("Rezultatul este:", rezultat)
    except ZeroDivisionError:
        print("Eroare: Împărțirea la zero nu este permisă.")
    except TypeError:
        print("Eroare: Argumentele trebuie să fie numere.")
    finally:
        print("Acesta este blocul de curățare.")

# Testăm funcția cu mai multe cazuri
impartire(10, 2)    # Împărțire normală, ar trebui să afișeze rezultatul
impartire(10, 0)    # Împărțire la zero, va ridica o excepție ZeroDivisionError
impartire(10, "a")  # Argument invalid, va ridica o excepție TypeError
```

Specificații

- Nume sugestiv
- scurtă descriere ce face funcția
- tipul și descrierea parametrilor
- excepții care pot fi aruncate de funcții și condițiile în care se aruncă

```
def gcd(a, b):
    """
    Return the greatest common divisor of two positive integers.
    a,b integer numbers
    return an integer number, the  greatest common divisor of a and b
    Raise ValueError if a<=0 or b<=0
    """
```

Cazuri de testare pentru excepții

```
def test_rational_add():
    """
      Test function for rational_add
    """
    assert rational_add(1, 2, 1, 3) == [5, 6]
    assert rational_add(1, 2, 1, 2) == [1, 1]
    try:
        rational_add(2, 0, 1, 2)
        assert False
    except ValueError:
        assert True
    try:
        rational_add(2, 3, 1, 0)
        assert False
    except ValueError:
        assert True
```

```
def rational_add(a1, a2, b1, b2):
    """
    Return the sum of two rational numbers.
    a1,a2,b1,b2 integer numbers, a2<>0 and b2<>0
    return a list with 2 ints, representing a rational number a1/b2 + b1/b2
    Raise ValueError if the denominators are zero.
    """
    if a2 == 0 or b2 == 0:
        raise ValueError("0 denominator not allowed")
    c = [a1 * b2 + a2 * b1, a2 * b2]
    d = gcd(c[0], c[1])
    c[0] = c[0] / d
    c[1] = c[1] / d
    return c
```