

Curs 4. Principii de organizarea aplicațiilor

- Organizarea aplicației pe funcții, module și pachete
- Arhitectura stratificata
- Excepții

Curs 3. Programare modulară

- Refactorizare
- Module
- Test Driven Development

Organizarea aplicației pe funcții și module

Scop:

Organizarea aplicației astfel încât:

- Sa fie ușor sa regăsim partea de cod care implementează un anumit lucru. Fiecare concept din aplicație sa aibă un loc bine definit unde se găsește implementarea conceptului
- Sa fie ușor de adăugat funcționalități noi. Ideal când adaug ceva nou in aplicație ar trebui sa modific cat mai puțin din ceea ce exista deja
- Sa fie ușor de testat automat. Testarea este fundamentala in crearea aplicațiilor.
- Sa permita colaborarea – mai mulți programatori lucrează pe același proiect. Sa fie ușor sa înțeleg părți ale aplicației (posibil scris de alt coleg) fără a fii nevoie sa știu detaliile de implementare

Responsabilități

Responsabilitate – motiv pentru a schimba ceva

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

Principul unei singure responsabilități - Single responsibility principle (SRP)

O funcție/modul trebuie să aibă o singură responsabilitate (un singur motiv de schimbare).

```
#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 conduc la:

- Dificultăți în înțelegere și utilizare
- Imposibilitatea de a testa
- Imposibilitatea de a refolosi
- Dificultăți la întreținere și evoluție

Separation of concerns

Principiu separării responsabilităților - Separation of concerns (SoC)

procesul de separare a unui program în responsabilități care nu se suprapun

<pre>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</pre>	<pre>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}]</pre>
--	---

Dependențe

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

Pentru a ușura întreținerea aplicației este nevoie de gestiunea dependențelor

Separarea interfeței de implementare, ascunderea detaliilor de implementare / reprezentare

Interfața funcției: este semnătura funcției + specificațiile

Interfața modul: semnătura și specificațiile tuturor funcțiilor din modul

Codul client – codul care folosește funcția modulului

Codul client nu ar trebui să depindă de detalii de implementare a modulului/funcției folosite sau de felul în care sunt reprezentate datele în interiorul funcției/modulului

<pre>#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))</pre>	<pre>#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])</pre>
--	---

Vers1: conceptul de calculator este reprezentat de funcțiile: reset, add_to, get_total (într-un modul separat). Conceptul de calculator este bine delimitat, apare într-un modul și expune operațiile posibile către restul programului

Codul client nu depinde de felul în care calculatorul arată în memorie => oricând se poate modifica reprezentarea calculatorului în memorie fără a modifica codul client

Pentru a înțelege/folosi codul legat de calculator este nevoie doar să ne uităm la semnătura funcțiilor reset, add_to, get_total și la specificațiile lor.

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 și devine dificilă izolarea prolemelor \Rightarrow cu cât gradul de cuplare este mai scăzut cu atât mai bine

Gradul de cuplare scăzut (Low coupling) facilitează dezvoltarea de aplicații care pot fi ușor modificate (interdependența între module/funcții este minimă astfel o modificare afectează doar o parte bine izolată din aplicație)

Coeziunea

Măsoară cât de relaționate sunt responsabilitățile unui element din program (pachet, modul, clasă)

Modulul poate avea:

- **Grad de coeziune ridicat (High Cohesion):** elementele implementează responsabilități înrudite
- Grad de coeziune scăzut (Low Cohesion): implementează responsabilități diverse din arii diferite (fără o legătură conceptuală între ele)

Un modul puternic coeziv ar trebui să realizeze o singură sarcină și să necesite interacțiuni minime cu alte părți ale programului.

Dacă elementele modulului implementează responsabilități disparate cu atât modulul este mai greu de înțeles/întreținut ⇒ Modulele ar trebui să aibă grad de coeziune ridicat

Arhitectura sistemelor software

Prin arhitectura aplicațiilor înțelegem structura la nivel general (macro) a unui sistem informatic de dimensiuni mari.

Describe structura generală a sistemului, interacțiunile și relațiile între diferite componente ale lui. Oferă o privire de ansamblu asupra structurii generale a aplicației și descrie aspecte care sunt costisitoare de modificat ulterior.

De ce trebuie să avem și să ne gândim la arhitectura aplicației: Scopul stabilirii și implementării unei arhitecturi este crearea de sisteme informatice ușor de modificat (adăugat funcționalități noi, schimbat tehnologiile folosite,...), ușor de testat, ușor de întreținut (în timp, de o echipă) și în general de a facilita crearea de sisteme software cu proprietăți deziderabile.

Există multe șabloane arhitecturale (abordări diferite în felul în care structurăm aplicația noastră): Arhitectura: Stratificată (Layered architecture), hexagonală, onion, clean, microservicii, bazată pe evenimente (event-driven), bazată pe servicii (service oriented), etc.

Orice arhitectura este un compromis între diferite atribute de dorit pentru o aplicație (performanță, extensibilitate, scalabilitate, flexibilitate,...)

Arhitectură stratificată (Layered Architecture)

Structurarea aplicației trebuie să aibă în vedere:

- Minimizarea cuplării între module (modulele nu trebuie să cunoască detalii despre alte module, astfel schimbările ulterioare sunt mai ușor de implementat)
- Maximizare coeziune pentru module (conținutul unui modul izolează un concept bine definit)

Arhitectură stratificată – este un șablon arhitectural care permite dezvoltarea de sisteme flexibile în care componentele au un grad ridicat de independență

- Fiecare strat comunică doar cu stratul imediat următor (depinde doar de stratul imediat următor)
- Fiecare strat are o interfață bine definită (se ascund detaliile), interfață folosită de stratul imediat superior

Arhitectură stratificată

- **Nivel prezentare** (User interface / Presentation)
 - implementează interfața utilizator (funcții/module/clase)
- **Nivel logic** (Domain / Application Logic)
 - oferă funcții determinate de cazurile de utilizare
 - implementează concepte din domeniul aplicației
- **Infrastructură**
 - funcții/module/clase generale, utilitare
- **Coordonatorul aplicației** (Application coordinator)
 - assemblează și pornește aplicația

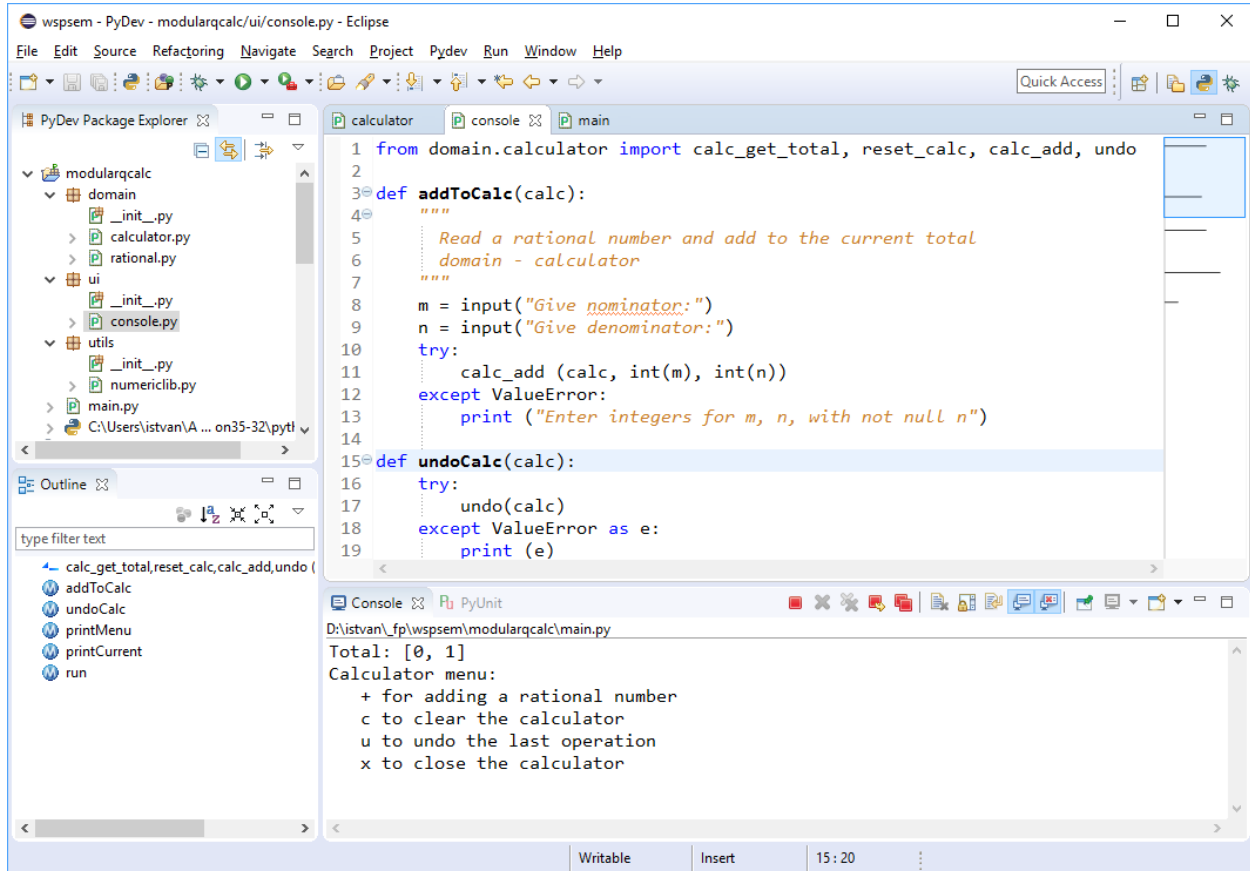
Layered Architecture – exemplu

```
#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
```

Organizarea proiectelor pe pachete/module



Erori și excepții

Erori de sintaxă – erori ce apar la parsarea codului

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

File "d:\wsp\hhh\aa.py", line 1

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

SyntaxError: invalid syntax

Codul nu e corect sintactic (nu respectă regulile limbajului)

Excepții

Erori detectate în timpul rulării.

Excepțiile sunt aruncate în momentul în care o eroare este detectată:

- pot fi aruncate de interpretorul python
- aruncate de funcții pentru a semnaliza o situație excepțională, o eroare
- ex. Nu sunt satisfăcute condițiile

```
>>> 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

```
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
```

Modelul de execuție (Execution flow)

Excepțiile întrerup execuția normală a instrucțiunilor

Este un mecanism prin care putem întrerupe execuția normală a unui bloc de instrucțiuni

Programul continuă execuția în punctul în care excepția este tratată (rezolvată) sau întrerupe de tot programul

```
def compute(a,b):  
    print ("compute :start ")  
    aux = a/b  
    print ("compute:after division")  
    rez = aux*10  
    print ("compute: return")  
    return rez  
  
def main():  
    print ("main:start")  
    a = 40  
    b = 1  
    c = compute(a, b)  
    print ("main:after compute")  
    print ("result:",c*c)  
    print ("main:finish")  
  
main()
```

Tratarea excepțiilor (Exception handling)

Procesul sistematic prin care excepțiile apărute în program sunt gestionate, executând acțiuni necesare pentru remedierea situației.

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

Excepțiile pot fi tratate în blocul de instrucțiuni unde apar sau în orice bloc exterior care în mod direct sau indirect a apelat blocul în care a apărut excepția (excepția a fost aruncată)

Dacă excepția este tratată, acesta oprește rularea programului

raise, try-except statements

```
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`,
- este posibil sa propagăm informații despre excepție
- clauza `finally` se execută în orice condiții (a apărut/nu a apărut excepția)
- putem arunca excepții proprii folosind `raise`

```
def f():  
    # x = 1/0  
    raise ValueError("Error Message") # aruncăm excepție  
  
try:  
    f()  
except ValueError as msg:  
    print ("handle value error:", msg)  
except KeyError:  
    print ("handle key error")  
except:  
    print ("handle any other errors")  
finally:  
    print ("Clean-up code here")
```

Folosiți excepții doar pentru:

- A semnala o eroare – semnalăm situația în care funcția nu poate respecta post condiția, nu poate furniza rezultatul promis în specificații
- Putem folosi pentru a semnala încălcarea precondițiilor

Nu folosiți excepții cu singurul scop de a altera fluxul de execuție

Specificații

- Nume sugestiv
- scurta descriere (ce face funcția)
- tipul și descrierea parametrilor
- condiții asupra parametrilor de intrare (precondiții)
- tipul, descrierea rezultatului
- relația între date și rezultate (postcondiții)
- **Excepții** care pot fi aruncate de funcție, și condițiile in 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
```

Curs 4. Principii de organizarea aplicațiilor

- Organizarea aplicației pe funcții, module și pachete
- Excepții

Curs 5: Tipuri definite de utilizator

- Programare bazată pe obiecte
- Principii de definire a tipurilor utilizator