

Fundamentele programării

Prof. Dr. Czibula Istvan

Lect. Mircea Ioan Gabriel

Drd. Briciu Anamaria

Drd. Maier Mariana

Drd. Mali Imre

C.d. asociat Berciu Liviu

Orar

Curs: 2 ore / săptămâna – online pe Microsoft Teams

Seminar: 2 ore / săptămâna – prezenta fizica, grupa 217 hibrid

Laborator: 2 ore / săptămâna – prezenta fizica, grupa 217 hibrid

Pagina WEB

<http://csubbcluj.ro/~istvanc/fp>

Email: istvan.czibula@ubbcluj.ro

Trimiteti emailuri doar de la adrese ubbcluj.ro.

Microsoft Teams TeamCode: **f85587s**

Obiective

- Cunoașterea conceptelor fundamentale programării
- Introducere concepte de bază legate de ingineria software (design, arhitectură, implementare și întreținere)
- Înțelegerea instrumentelor software folosite în dezvoltarea de aplicații
- Învățarea limbajului Python și utilizarea lui pentru implementarea, testarea, rularea, depanarea de programe.
- Însușirea/Îmbunătățirea stilului de programare.

Programming in the small vs Programming in the large
Algoritmica/Programare vs Inginerie Software

Conținut

1. Introducere în procesul de dezvoltare software
2. Programare procedurală
3. Programare modulară
4. Tipuri definite de utilizator - Object based programming
5. Principii de dezvoltare – Arhitectură stratificată
6. Principii de dezvoltare – Șabloane GRASP, diagrame UML
7. Testarea și inspectarea programelor
8. Recursivitate
9. Complexitatea algoritmilor
10. Algoritmi de căutare
11. Algoritmi de sortare
12. Backtracking
13. Greedy, Programare dinamica
14. Recapitulare

Evaluare

Lab (30%) - o notă pe activitatea de laborator din timpul semestrului.

Simulare (10%) - examen practic (în timpul semestrului)

T (30%) - examen practic (în sesiune)

E (30%) - examen scris (în sesiune)

Pentru a intra in examen:

Minim 12 prezente la laborator

Minim 10 prezente la seminar

Pentru promovare trebuie să aveți cel puțin nota 5 la toate (Lab,T,E >= 5)

Toate activitățile sunt obligatorii.

Dacă nu obțineți nota 5 la laborator nu puteți intra in examen in sesiunea normală.

Restanțe

În sesiunea de restanțe puteți preda laboratoare(nota maximă este 5).

Se poate re-susține examenul practic

Se poate re-susține examenul scris

Curs 1. Procesul de dezvoltare software

- Ce este programarea
- Elemente de bază al limbajului Python
- Proces de dezvoltare bazat pe funcționalități

Ce este programarea

Hardware / software

Hardware - *computere*(desktop, laptop, etc) și alte dispozitive (mobile, atm, etc.)

Software - *programe sau sisteme* ce rulează pe hardware

Limbaj de programare – Notații și reguli pentru scrierea de programe (sintaxă și semantică)

Python: Limbaj de programare de nivel înalt (high level programming language).

Interpreter Python: un program care permite rularea/interpretarea programelor scrise în limbajul Python.

Biblioteci Python: Funcții, module, tipuri de date disponibile în Python, scrise de alți programatori

Program 1 - Hello world

```
print ('Hello world')
```

Python

Download: python.org - versiunea 3.x

După instalare:

- interpreter python – executa programe scrise in python
- IDLE Python – un mic mediu de dezvoltare pentru python

Autor principal limbaj (1989): **Guido van Rossum**

Rol curent: benevolent dictator for life (BDFL) – nu mai e cazul

PEP – Python Enhancement Proposal – design document providing information to the Python community, or describing a new feature for Python or its processes or environment.

PEP 20 The Zen of Python – câteva principii de baza

PEP 8 Style Guide for Python code

Documentație: docs.python.org

<https://docs.python.org/3/reference/index.html> - core syntax&semantics

<https://docs.python.org/3/library/index.html> - standard library

Ce fac computerele

- Stochează date
 - Memoria internă
 - Memoria externă (hard, stick, CD, etc)
- Operează
 - procesor
- Comunică
 - Prin tastatură, mouse, ecran
 - Conexiuni de tip rețea

Informații și date

Date - o colecție de simboluri stocate într-un computer (Ex. 123 decimal sau sirul de caractere ‘abc’) sunt stocate folosind reprezentarea binara

Informații - interpretarea unor date (Ex. 123, ‘abc’)

Procesarea datelor și informațiilor

- Dispozitivele de intrare transformă informațiile în date (ex. 123 citit de la tastatură)
- Datele sunt stocate în memorie (ex. 1111011 pentru numărul 123)
- Dispozitivele de ieșire produc informații din date

Operații de bază ale procesoarelor

- În reprezentare binară
- Operații (and, or, not; add, etc)

Elemente de bază ale unui program Python

Program 2 - Adding two integers

```
# Reads two integers and prints the sum of them
a = input("Enter the first number: ")
b = input("Enter the second number: ")
c = int(a) + int(b)
print("The sum of ", a, " + ", b, " is ", c)
```

Elemente lexicale

Un program Python este alcătuit din mai multe linii de cod

Comentarii

- Începe cu # și țin până la sfârșitul liniei
- Începe cu "" și țin mai multe rânduri, până la un nou ""

Identificatori: secvențe de caractere (litere, cifre, _) care încep cu o literă sau cu _

Literali: notații pentru valorile constante sau pentru tipuri definite de utilizator

Modelul de date

Toate datele într-un program Python – **obiecte**

Un obiect are :

- **o identitate** – adresa lui în memorie
- **un tip** – care determină operațiile posibile precum și valorile pe care le poate lua obiectul
- **o valoare.**

Odată creat, **identitatea** și **tipul** obiectului nu mai pot fi modificate. Valoarea unor obiecte se poate modifica.

- Obiecte **mutable** - se poate modifica
- Obiecte **ne-mutable** – nu se poate modifica, orice operatie efectuată creează un nou obiect

Tipuri de date standard

Tipul de date definește **domeniul** de valori posibile și **operațiile** permise asupra valorilor din domeniu.

Numerice – Numerele sunt imutabile – odată create valoare nu se mai poate schimba (operațiile creează noi obiecte).

int (numere întregi):

- numerele întregi (pozitive și negative), dimensiune limitată doar de memoria disponibilă
- Operații: +, -, *, /, //, **, % comparare: ==, !=, <, > operații pe biți: |, ^, &, <<, >>, ~
- Literali: 1, -3

bool (boolean):

- Valorile True și False.
- Operații: and, or, not
- Literali: False, True; 0, 1

float (numere reale):

- numerele reale (dublă precizie)
- Operații: +, -, *, / comparare: ==, !=, <, >
- Literali: 3.14

NoneType

- O singură valoare: None
- Operații: ==, !=
- Literali: None

Tipuri de date standard

Secvențe:

- Multimi finite și ordonate, indexate prin numere ne-negative.
 - Dacă a este o secvență atunci:
 - **len(a)** returnează numărul de elemente;
 - **a[0], a[1], ..., a[len(a)-1]** elementele lui a.
 - Exemplu: [1, ‘a’]
 - Exemplu de operații: accesare elemente, +, * cu un scalar, etc.
- String:** ‘abc’, “abc” - este o secvență imutabilă de caractere Unicode .
- List:** [2,3] , [1, ‘a’, [1, 3]] – secvență mutabilă
- Tuple:** (2,3), (1,’a’),(1,3)) – secvență mutabilă

Dicționar: { 'num': 1, 'denom': 2 }

- conține perechi (cheie – valoare)
- fiecare cheie apare o singură dată
- operația de găsire a unei valori după o cheie foarte eficient

Funcție:

- funcțiile in Python pot fi tratate ca orice alt tip de valoare
- Operații: singura operație permisa este apelul funcției: func(arg)
- obiecte de tip funcție se creează prin definirea de funcții
- funcțiile in Python pot fi asignate la variabile, transmise ca parametrii pentru o alta funcție, returnate dintr-o funcție.

Liste

operații:

- creare [7, 9]
- accesare valori, lungime (**index, len**), modificare valori (**listele sunt mutabile**), verificare daca un element este in lista (2 in [1, 2, 'a'])
- ștergere inserare valori (**append, insert, pop**) del a[3]
- slicing, liste eterogene
- listele se pot folosi in for
- lista ca stivă(**append, pop**)
- folosiți instrucțunea **help(list)** pentru mai multe detalii despre operații posibile

<pre># create a = [1, 2, 'a'] print (a) x, y, z = a print(x, y, z) # indices: 0, 1, ..., len(a) - 1 print (a[0]) print ('last element = ', a[len(a)-1]) # lists are mutable a[1] = 3 print (a)</pre>	<pre># slicing print (a[:2]) b = a[:] print (b) b[1] = 5 print (b) a[3:] = [7, 9] print(a) a[:0] = [-1] print(a) a[0:2] = [-10, 10] print(a)</pre>
<pre># lists as stacks stack = [1, 2, 3] stack.append(4) print (stack) print (stack.pop()) print (stack)</pre>	<pre># nesting a = [1, [1, 1, 9], 9] print (a) b = [1, 1, 9] c = [1, b, 9] print (c)</pre>
<pre>#generate lists using range l1 = range(10) print (list(l1)) l2 = range(0,10) print (list(l2)) l3 = range(0,10,2) print (list(l3)) l4 = list(range(9,0,-1)) print (l4)</pre>	<pre>#list in a for loop l = range(0,10) for i in l: print (i)</pre>

Tuplu

Sunt secvențe imutabile. Conține elemente, indexat de la 0

Operări:

- Crearea - packing (`(23, 32, 3)`)
- heterogen
- poate fi folosit în for
- unpacking

<pre># Tuples are immutable sequences # A tuple consists of a number # of values separated by commas # tuple packing t = 12, 21, 'ab' print(t[0]) # empty tuple (0 items) empty = ()</pre>	<pre># tuple with one item singleton = (12,) print(singleton) print(len(singleton)) #tuple in a for t = 1,2,3 for el in t: print(el)</pre>
<pre># sequence unpacking x, y, z = t print(x, y, z)</pre>	<pre># Tuples may be nested u = t, (23, 32) print(u)</pre>

Dicționar

Un dicționar este o mulțime de perechi (cheie, valoare).

Cheile trebuie să fie **înmutabile**.

Operații:

- creare {} sau {'num': 1, 'denom': 2}
- accesare valoare pe baza unei chei
- adăugare/modificare pereche (cheie, valoare)
- ștergere pereche (cheie, valoare)
- verificare dacă cheia există

<pre>#create a dictionary a = {'num': 1, 'denom': 2} print(a) #get a value for a key print(a['num'])</pre>	<pre>#set a value for a key a['num'] = 3 print(a) print(a['num'])</pre>
<pre>#delete a key value pair del a['num'] print(a)</pre>	<pre>#check for a key if 'denom' in a: print('denom = ', a['denom']) if 'num' in a: print('num = ', a['num'])</pre>

Variabile

Variabilă:	Variabilă în Python:
<ul style="list-style-type: none">• nume• valoare• tip<ul style="list-style-type: none">◦ domeniu◦ operații• locație de memorie	<ul style="list-style-type: none">• nume• valoare◦ tip<ul style="list-style-type: none">◦ domeniu◦ operații◦ locație de memorie

Introducerea unei variabile într-un program – asignare

Expresii

O combinație de valori, constante, variabile, operatori și funcții care sunt interpretate conform regulilor de precedență, calculate și care produc o altă valoare

Exemple:

- numeric : $1 + 2$
- boolean: $1 < 2$
- string : '1' + '2'

Funcții utile:

help(instrucțiune) - ajutor

id(x) – identitatea obiectului

dir()

locals() / globals() - nume definite (variabile, funcții, module, etc)

Instrucțiuni

Operațiile de bază ale unui program. Un program este o secvență de instrucțiuni

- **Atribuire/Legare**

- Instrucțiunea =.
- Atribuirea este folosit pentru a lega un nume de o variabilă
- Poate fi folosit și pentru a modifica un element dintr-o secvență mutabilă.
- Legare de nume:
 - `x = 1 #x is a variable (of type int)`
- Re-legare name:
 - `x = x + 2 #a new value is assigned to x`
- Modificare secvență:
 - `y = [1, 2] #mutable sequence`
 - `y[0] = -1#the first item is bound to -1`

- **Blocuri**

- Parte a unui program care este executată ca o unitate
- Secvență de instrucțiuni
- Se realizează prin indentarea liniilor (toate instrucțiunile indentate la același nivel aparțin aceluiași bloc

Instructiuni - If, While

```
if conditie:  
    bloc de instructiuni  
elif conditie:  
    bloc de instructiuni  
else:  
    bloc de instructiuni  
  
while conditie:  
    bloc de instructiuni  
    [break]  
    [continue]
```

```
def gcd(a, b):  
    """  
    Return the greatest common divisor of two positive integers.  
    """  
    if a == 0: return b  
    if b == 0: return a  
  
    while a != b:  
        if a > b:  
            a = a - b  
        else:  
            b = b - a  
    return a  
  
  
print (gcd(7,15))
```

Instructiuni – For

```
for el in secventa: #parcurgem element cu element
    bloc de instructiuni #el - element in secventa
    [break]
    [continue]
else:
    bloc de instructiuni #executat daca s-a dat break
```

```
#use a list literal
for i in [2,-6, "a", 5]:
    print (i)

#using a variable
x = [1,2,4,5]
for i in x:
    print (i)

#using range
for i in range(10):
    print (i)

for i in range(2,100,7):
    print (i)

#using a string
s = "abcde"
for c in s:
    print (c)
```

Parcure in Python

Pythonic	Programator C++/Java/C#/Pascal
<pre>for i in range(6): print (i)</pre>	<pre>for i in [0,1,2,4,5]: print (i)</pre>
<pre>x = [2,-6,"a",5] for el in x: print (el)</pre>	<pre>x = [2,-6,"a",5] for i in range(len(x)): print (x[i])</pre>
<pre>x = [2,-6,"a",5] for el in reversed(x): print (el)</pre>	<pre>x = [2,-6,"a",5] for i in range(len(x),-1,-1): print (x[i])</pre>
<pre>x = [2,-6,"a",5] for i, el in enumerate (x): print (i, "->", el)</pre>	<pre>x = [2,-6,"a",5] for i in range(len(x)): print (i, "->", x[i])</pre>
#parcure 2 liste simultan <pre>x = [2,-6,"a",5] y = [2,-6,"a"] for elx, ely in zip (x,y): print (elx, "<->", ely)</pre>	#parcure 2 liste simultan <pre>x = [2,-6,"a",5] y = [2,-6,"a"] n = min(len(x),len(y)) for i in range(n): print (x[i],y[i])</pre>

Transforming Code into Beautiful, Idiomatic Python <https://www.youtube.com/watch?v=OSGv2VnC0go>

Dictionar: dictionar = { 'num': 1, 'denom': 2 }

#parcure cheile din dictionar <pre>for cheie in dictionar: print (cheie)</pre>
#parcure valorile din dictionar <pre>for valoare in dictionar.values(): print (valoare)</pre>
#parcure perechile din dictionar <pre>for cheie,valoare in dictionar.items(): print (cheie,valoare)</pre>

Cum se scriu programe

Roluri în ingineria software

Programator/Dezvoltator

- Folosește calculatorul pentru a scrie/dezvolta aplicații

Client (stakeholders):

- Cel interesat/afectat de rezultatele unui proiect.

Utilizatori

- Folosesc/rulează programul.

Un proces de dezvoltare software este o abordare sistematică pentru construirea, instalarea, întreținerea produselor software. Indică:

- Pași care trebuie efectuați.
- Ordinea lor

Folosim la fundamentele programării: un proces de dezvoltare incrementală bazată pe funcționalități (simple feature-driven development process)

Enunț (problem statement)

Enunțul este o descriere scurtă a problemei de rezolvat.

Calculator - Problem statement
Profesorul (client) are nevoie de un program care ajută <i>elevii</i> (users) să învețe despre numere raționale. Programul ar trebui să permită elevilor să efectueze operații aritmetice cu numere raționale

Cerințe (requirements)

Cerințele definesc în detaliu de ce este nevoie în program din perspectiva clientului. Definește:

- Ce dorește clientul
- Ce trebuie inclus în sistemul informatic pentru a satisface nevoile clientului.

Reguli de elaborare a cerințelor:

- **Cerințele exprimate corect asigură dezvoltarea sistemului conform așteptărilor clientilor.** (Nu se rezolvă probleme ce nu s-au cerut)
- Descriu **lista de funcționalități** care trebuie oferite de sistem.
- Funcționalitățile trebuie să clarifice orice ambiguități din enunț.

Funcționalitate

- O funcție a sistemului dorit de client
- descrie datele rezultatele și partea sistemul care este afectat
- este de dimensiuni mici, poate fi implementat într-un timp relativ scurt
- se poate estima
- exprimată în forma acțiune rezultat obiect
 - Acțiunea – o funcție pe care aplicația trebuie să o furnizeze
 - Rezultatul – este obținut în urma execuției funcției
 - Obiect – o entitate în care aplicația implementează funcția

Calculator – Listă de Funcționalități
F1. Adună un număr <i>rational</i> în calculator.
F2. Șterge calculator.
F3. Undo – reface ultima operație (utilizatorul poate repeta această operație).

Proces de dezvoltare incrementală bazată pe funcționalități

- Se creează lista de funcționalități pe baza enunțului
- Se planifică iterațiile (o iterație conține una/mai multe funcționalități)
- Pentru fiecare funcționalitate din iterație
 - Se face modelare – scenarii de rulare
 - Se creează o lista de taskuri (activități)
- Se implementează și testează fiecare activitate

Iterație: O perioadă de timp în cadrul căreia se realizează o versiune stabilă și executabilă a unui produs, împreună cu documentația suport

La terminarea iterației avem un program funcțional care face ceva util clientului

Exemplu: plan de iterații

Iteration	Planned features
I1	F1. Adună un <i>număr rațional</i> în calculator.
I2	F2. Sterge calculator.
I3	F3. Undo – reface ultima operație (utilizatorul poate repeta această operație).

Modelare - Iteration modeling

La fiecare început de iterătie trebuie analizat funcționalitatea care urmează a fi implementată.

Acet proces trebuie sa sigure înțelegerea funcționalității si sa rezulte un set de pași

mai mici (work item/task), activități care conduc la realizarea funcționalității

Fiecare activitate se poate implementa/testa independent

Iterația 1 - Adună un *număr rațional* în calculator.

Pentru programe mai simple putem folosi **scenarii de rulare** (tabelară) pentru a înțelege problema și modul în care funcționalitatea se manifestă în program. Un scenariu descrie interacțiunea între utilizator și aplicație.

Scenariu pentru funcționalitatea de adăugare număr rațional

	Utilizator	Program	Descriere
a		0	Tipărește totalul curent
b	1/2		Adună un număr rațional
c		1/2	Tipărește totalul curent
d	2/3		Adună un număr rațional
e		5/6	Tipărește totalul curent
f	1/6		Adună un număr rațional
g		1	Tipărește totalul curent
h	-6/6		Adună un număr rațional
i		0	Tipărește totalul curent

Listă de activități

Recomandări:

- Definiți o activitate pentru fiecare operație care nu este implementată deja (de aplicație sa de limbajul Python), ex. T1, T2.
- Definiți o activitate pentru implementarea interacțiunii program-utilizator (User Interface), ex. T4.
- Definiți o activitate pentru a implementa operațiile necesare pentru interacțiune utilizator cu UI, ex. T3.
- Determinați dependențele între activități (ex. T4 --> T3 --> T2 -->T1, unde --> semnifică faptul ca o activitate depinde de o altă activitate).
- Faceți un mic plan de lucru (T1,T2,T3,T4)

T1	Determinare cel mai mare divizor comun (punctele g, I din scenariu)
T2	Sumă două numere raționale (c, e, g, i)
T3	Implementare calculator: init, add, and total
T4	Implementare interfață utilizator

Activitate 1. Determinare cel mai mare divizor comun

Cazuri de testare

Un **test case** conține un set de intrări și rezultatele așteptate pentru fiecare intrare.

Date: a, b	Rezultate: gcd (a, b): c, unde c este cel mai mare divizor comun
2 3	1
2 4	2
6 4	2
0 2	2
2 0	2
24 9	3
-2 0	ValueError
0 -2	ValueError

Curs 1. Procesul de dezvoltare software

- Ce este programarea
- Elemente de bază al limbajului Python
- Proces de dezvoltare bazat pe funcționalități

Curs 2. Programare procedurală

- Funcții în Python
- Cum se scriu funcții
- Dezvoltare dirijată de teste (Test Driven Development)

Referințe

1. The Python language reference. <http://docs.python.org/py3k/reference/index.html>
2. The Python standard library. <http://docs.python.org/py3k/library/index.html>
3. The Python tutorial. <http://docs.python.org/tutorial/index.html>

Fundamentele programării

Curs 2. Programare procedurală

- Funcții
- Cum se scriu funcții
- Funcții de test

Curs 1. Procesul de dezvoltare software

- Ce este programarea
- Elemente de bază al limbajului Python
- Proces de dezvoltare bazat pe funcționalități

Programare procedurală

Paradigmă de programare

stil fundamental de scriere a programelor, set de convenții ce dirijează modul în care gândim programele.

Programare imperativă

Calcule descrise prin instrucțiuni care modifică starea programului. Orientat pe acțiuni și efectele sale

Programare procedurală

Programul este format din mai multe proceduri (funcții, subroutines)

Ce este o funcție

O funcție este un bloc de instrucțiuni de sine stătător care are:

- un **nume**,
- poate avea o **listă de parametrii** (formali),
- poate **returna** o valoare
- are un **corp** format din instrucțiuni
- are o **documentație** (specificație) care include:
 - o scurtă descriere
 - *tipul* și descriere parametrilor
 - condiții impuse parametrilor de intrare (*precondiții*)
 - tipul și descrierea valorii returnate
 - condiții impuse rezultatului, condiții care sunt satisfăcute în urma executării (*post-condiții*).
 - Excepții ce pot să apară

```
def max(a, b):  
    """  
    Compute the maximum of 2 numbers  
    a, b - numbers  
    Return a number - the maximum of two integers.  
    Raise TypeError if parameters are not integers.  
    """  
    if a>b:  
        return a  
    return b  
  
def isPrime(a):  
    """  
    Verify if a number is prime  
    a an integer value (a>1)  
    return True if the number is prime, False otherwise  
    """
```

Funcții

Toate funcțiile noastre trebuie să:

- folosească nume sugestive (pentru numele funcției, numele variabilelor)
- să oferă specificații
- să includă comentarii
- să fie testată

O funcție ca și în exemplu de mai joi, este corectă sintactic (funcționează în Python) dar la laborator/examen nu consideram astfel de funcții:

```
def f(k):
    l = 2
    while l < k and k % l > 0:
        l=l+1
    return l>=k
```

Varianta acceptată este:

```
def isPrime(nr):
    """
        Verify if a number is prime
        nr - integer number, nr>1
        return True if nr is prime, False otherwise
    """
    div = 2 #search for divider starting from 2
    while div<nr and nr % div>0:
        div=div+1
    #if the first divider is the number itself than the number is prime
    return div>=nr;
```

Definiția unei funcții în Python

Folosind instrucțiunea `def` se pot definii funcții în python.

Interpretorul executa instrucțiunea `def`, acesta are ca rezultat introducerea numelui funcției (similar cu definirea de variabile)

Corpul funcției nu este executat, este doar asociat cu numele funcției

```
def max(a, b):
    """
    Compute the maximum of 2 numbers
    a, b - numbers
    Return a number - the maximum of two integers.
    Raise TypeError if parameters are not integers.
    """
    if a>b:
        return a
    return b
```

Apel de funcții

Un **bloc** de instrucțiuni în Python este un set de instrucțiuni care este executat ca o unitate. Blocurile sunt delimitate folosind indentarea.

Corpul unei funcții este un bloc de instrucțiuni și este executat în momentul în care funcția este apelată.

max(2,5)

La apelul unei funcții se creează un nou cadru de execuție, care :

- informații administrative (pentru depanare)
- determină unde și cum se continuă execuția programului (după ce execuția funcției se termină)
- definește două spații de nume: locals și globals care afectează execuția funcției.

Spații de nume (namespace)

- este o mapare între nume (identificatori) și obiecte
- are funcționalități similare cu un dicționar (in general este implementat folosind tipul dicționar)
- sunt create automat de Python
- un spațiu de nume poate fi referit de mai multe cadre de execuție

Adăugarea unui nume în spațiu de nume: legare ex: `x = 2`

Modificarea unei mapări din spațiu de nume: re-legare

În Python avem mai multe spațiile de nume, ele sunt create în momente diferite și au ciclu de viață diferit.

- General/implicit – creat la pornirea interpretorului, conține denumiri predefinite (built-in)
- global – creat la încărcarea unui modul, conține nume globale
 - `globals()` - putem inspecta spațiu de nume global
- local – creat la apelul unei funcții, conține nume locale funcției
 - `locals()` - putem inspecta spațiu de nume local

Transmiterea parametrilor

Parametru formal este un identificator pentru date de intrare. Fiecare apel trebuie să ofere o valoare pentru parametru formal (pentru fiecare parametru obligatoriu)

Parametru actual valoare oferită pentru parametrul formal la apelul funcției.

- Parametrii sunt transmiși prin referință. Parametru formal (identificatorul) este legat la valoarea (obiectul) parametrului actual.
- Parametrii sunt introdusi în spațiu de nume local

```
def change_or_not_immutable(a):
    print ('Locals ', locals())
    print ('Before assignment: a = ', a, ' id = ', id(a))
    a = 0
    print ('After assignment: a = ', a, ' id = ', id(a))

g1 = 1      #global immutable int
print ('Globals ', globals())
print ('Before call: g1 = ', g1, ' id = ', id(g1))
change_or_not_immutable(g1)
print ('After call: g1 = ', g1, ' id = ', id(g1))
```

```
def change_or_not mutable(a):
    print ('Locals ', locals())
    print ('Before assignment: a = ', a, ' id = ', id(a))
    a[1] = 0
    a = [0]
    print ('After assignment: a = ', a, ' id = ', id(a))

g2 = [0, 1] #global mutable list
print ('Globals ', globals())
print ('Before call: g2 = ', g2, ' id = ', id(g2))
change_or_not mutable(g2)
print ('After call: g2 = ', g2, ' id = ', id(g2))
```

Vizualizare memorie în timpul execuției

<http://www.pythontutor.com/visualize.html#mode=display>

Write code in Python 3.6

```
1 def change_or_not Mutable(a):
2     a[1] = 0
3     a = [0]
4
5 g2 = [0, 1] #global mutable list
6 change_or_not Mutable(g2)
7 print(g2)
```

Help improve this tool by completing a [short user survey](#).
Keep this tool free by making a [small donation](#) (PayPal, Patreon, credit/debit card)

Visualize Execution

Live Programming Mode

Python 3.6

```
→ 1 def change_or_not Mutable(a):
2     a[1] = 0
3     a = [0]
4
5 g2 = [0, 1] #global mutable list
→ 6 change_or_not Mutable(g2)
7 print(g2)
```

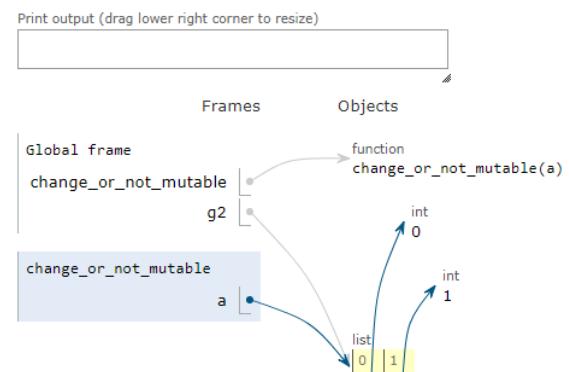
[Edit this code](#)

green arrow: line that has just executed
red arrow: next line to execute

Click a line of code to set a breakpoint; use the Back and Forward buttons to jump there.

<< First < Back Step 4 of 8 Forward > Last >>

Created by [@pgbovine](#). Support with a [small donation](#).



Vizibilitatea variabilelor

Domeniul de vizibilitate (scope) – Definește vizibilitatea unui nume într-un bloc.

- Variabilele definite într-o funcție au domeniul de vizibilitate locală (funcția) – se poate accesa doar în interiorul funcției
- Variabilele definite într-un modul au vizibilitate globală (globală pe modul)
- Orice nume (variabile, funcții) poate fi folosit doar după ce a fost legat (prima atribuire)
- Parametrii formali au domeniu de vizibilitate funcția (apartin spațiului de nume local)

```
global_var = 100

def f():
    local_var = 300
    print (local_var)
    print (global_var)
```

Domeniu de vizibilitate

Reguli de accesare a variabilelor (sau orice nume) într-o funcție:

- când se folosește un nume de variabilă într-o funcție se caută în următoarele ordine în spațiile de nume:
 - spațiu local
 - spațiu local funcției exterioare (doar dacă avem funcție declarată în interiorul altei funcții)
 - spațiu global (nume definite în modul)
 - spațiul built-in
- operatorul = schimba/creează variabile în spațiu de nume local
- Putem folosi declarația `global` pentru a referi/importa o variabilă din spațiu de nume global în cel local
- `nonlocal` este folosit pentru a referi variabile din funcția exterioară (doar dacă avem funcții în funcții)

```
a = 100
def f():
    a = 300
    print(a)

f()
print(a)
```

```
a = 100
def f():
    global a
    a = 300
    print(a)

f()
print(a)
```

`globals()` `locals()` - funcții built-in prin care putem inspecta spațiile de nume

```
a = 300
def f():
    a = 500
    print(a)
    print(locals())
    print(globals())

f()
print(a)
```

Cum scriem funcții – Cazuri de testare

Înainte să implementăm funcția scriem cazuri de testare pentru:

- a specifică funcția (ce face, pre/post condiții, exceptii)
- ca o metodă de a analiza problema
- să ne punem în perspectiva celui care folosește funcția
- pentru a avea o modalitate sa testam după ce implementăm

Un caz de testare specifică datele de intrare și rezultatele care le așteptam de la funcție

Cazurile de testare:

- se pot face în format tabelar, tabel cu date/rezultate.
- executabile: funcții de test folosind `assert`
- biblioteci/module pentru testare automată

Instrucțiunea `assert` permite inserarea de aserționi (expresii care ar trebui să fie adevărate) în scopul depanării/verificării aplicațiilor.

`assert expresie`

Folim `assert` pentru a crea teste automate

Functii de test - Calculator

1 Funcționalitate 1. Add a number to calculator.

2 Scenariu de rulare pentru adăugare număr

3 Activități (Workitems/Tasks)

T1	Calculează cel mai mare divizor comun
T2	Sumă două numere raționale
T3	Implementare calculator: init, add, and total
T4	Implementare interfață utilizator

T1 Calculează cel mai mare divizor comun

Cazuri de testare Format tabelar		Funcție de test
Input: (params a,b)	Output: gcd(a,b)	
2 3	1	
2 4	2	
6 4	2	
0 2	2	
2 0	2	
24 9	3	

Implementare gcd

```
def gcd(a, b):
    """
        Compute the greatest common divisor of two positive integers
        a, b integers a,b >=0
        Return the greatest common divisor of two positive integers.
    """
    if a == 0:
        return b
    if b == 0:
        return a
    while a != b:
        if a > b:
            a = a - b
        else:
            b = b - a
    return a
```

Cum se scriu funcții

Dezvoltare dirijată de teste (test-driven development - TDD)

Dezvoltarea dirijată de teste presupune crearea de teste automate, chiar înainte de implementare, care clarifică cerințele

Pașii TDD pentru crearea unei funcții:

- Adaugă un test
 - **Scriți o funcție de test (`test_f()`)** care conține cazuri de testare sub forma de aserțiuni (instrucțiuni assert).
 - La acest pas ne concentrăm la specificațiile funcției **f**.
 - Definim funcția **f**. nume, parametrii, precondiții, post-condiții, și corpul gol (instrucțiunea `pass`).
- Rulăm toate testele și verificăm ca noul test pică
 - Pe parcursul dezvoltării o să avem mai multe funcții, astfel o să avem mai multe funcții de test .
 - La acest pas ne asigurăm ca toate testele anterioare merg, iar testul nou adăugat pică.
- Scriem corpul funcției
 - La acest pas avem deja specificațiile, ne concentrăm doar la implementarea funcției conform specificațiilor și ne asigurăm ca noile cazuri de test scrise pentru funcție trec (funcția de test)
 - **La acest pas nu ne concentrăm la aspecte tehnice** (cod duplicat, optimizări, etc).
- Rulăm toate testele și ne asigurăm că trec
 - Rulând testele ne asigurăm că nu am stricat nimic și noua funcție este implementată conform specificațiilor
- Refactorizare cod
 - La acest pas îmbunătățim codul, folosind reautorizări

Curs 2. Programare procedurală

- Funcții
- Cum se scriu funcții
- Funcții de test

Curs 3. Programare modulară

- Module
- Organizarea aplicației pe module și pachete

Referințe

- *The Python language reference.* <http://docs.python.org/py3k/reference/index.html>
- *The Python standard library.* <http://docs.python.org/py3k/library/index.html>
- *The Python tutorial.* <http://docs.python.org/tutorial/index.html>
- Kent Beck. *Test Driven Development: By Example*. Addison-Wesley Longman, 2002. See also Test-driven development. http://en.wikipedia.org/wiki/Test-driven_development
- Martin Fowler. *Refactoring. Improving the Design of Existing Code*. Addison-Wesley, 1999. See also <http://refactoring.com/catalog/index.html>

Fundamentele programării

Curs 3. Programare modulară

- Refactorizare
- Module
- Organizarea aplicației pe module și pachete

Curs 2. Programare procedurală

- Funcții
- Cum se scriu funcții
- Funcții de test

Recapitulare

Ce am învățat pana acum:

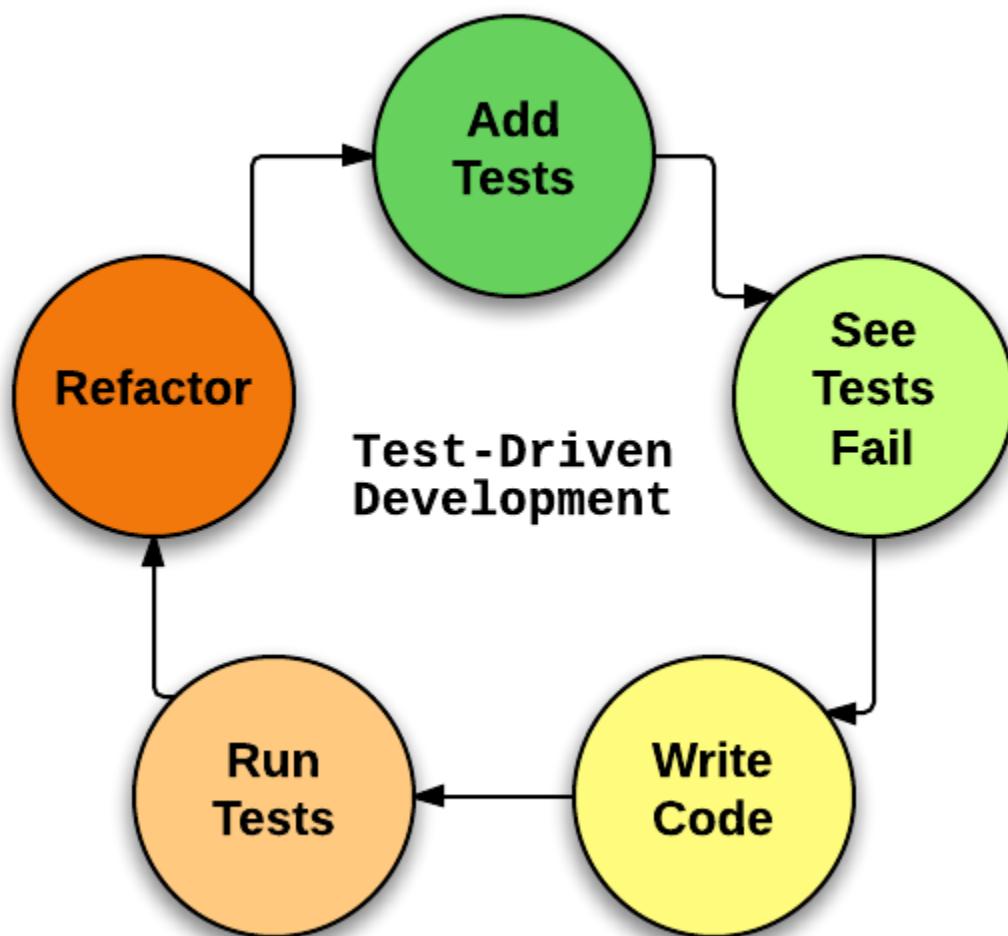
- Python
- Te gândești cum arata înainte sa te apuci de funcționalitate (Scenarii de rulare)
- Funcțiile sunt create odată si folosite pentru tot parcursul existentei aplicației -> este foarte important sa fie ușor de inteles/folosit
- Orice funcție are:
 - Documentație/Specificații
 - Teste automate
 - Înainte sa implementezi funcția te gândești la ce trebuie sa facă
 - Ne gândim la cum se folosește (apeleză) nu la cum e mai ușor sa implementam
- Testarea este fundamentală pentru crearea de aplicații

Dezvoltare dirijată de teste (test-driven development - TDD)

Dezvoltarea dirijată de teste presupune crearea de teste automate, chiar înainte de implementare, care clarifică cerințele

Pașii TDD pentru crearea unei funcții:

- Adaugă un test – creează teste automate
- Rulăm toate testele și verificăm ca noul test pică
- Scriem corpul funcției
- Rulăm toate testele și ne asigurăm că trec
- Refactorizăm codul



TDD Pas 1. Creare de teste automate

Când lucrăm la un task începem prin crearea unei funcții de test

Task: Calculează cel mai mare divizor comun

```
def test_gcd():
    """
        test function for gcd
    """
    assert gcd(0, 2) == 2
    assert gcd(2, 0) == 2
    assert gcd(2, 3) == 1
    assert gcd(2, 4) == 2
    assert gcd(6, 4) == 2
    assert gcd(24, 9) == 3
```

Ne concentrăm la specificarea funcției.

```
def gcd(a, b):
    """
        Return the greatest common divisor of two positive integers.
        a,b integer numbers, a>=0; b>=0
        return an integer number, the greatest common divisor of a and b
    """
    Pass
```

TDD Pas 2 - Rulăm testele

```
#run the test - invoke the test function
test_gcd()
```

Traceback (most recent call last):

```
  File "C:/curs/lect3/tdd.py", line 20, in <module>  test_gcd()
```

```
    File "C:/curs/lect3/tdd.py", line 13, in test_gcd
```

```
      assert gcd(0, 2) == 2
```

```
AssertionError
```

- Validăm că avem un test funcțional – se execută, eșuează.
- Astfel ne asigurăm că testul este executat și nu avem un test care trece fără a implementa ceva – testul ar fi inutil

TDD Pas 3 – Implementare

- implementare funcție conform specificațiilor (pre/post condiții), scopul este sa tracă testul
- soluție simplă, fără a ne concentra pe optimizări, evoluții ulterioare, cod duplicat, etc.

```
def gcd(a, b):  
    """  
    Return the greatest common divisor of two positive integers.  
    a,b integer numbers, a>=0; b>=0  
    return an integer number, the greatest common divisor of a and b  
    """  
    if a == 0:  
        return b  
    if b == 0:  
        return a  
    while a != b:  
        if a > b:  
            a = a - b  
        else:  
            b = b - a  
    return a
```

TDD Pas 4 – Executare funcții de test-toate cu succes

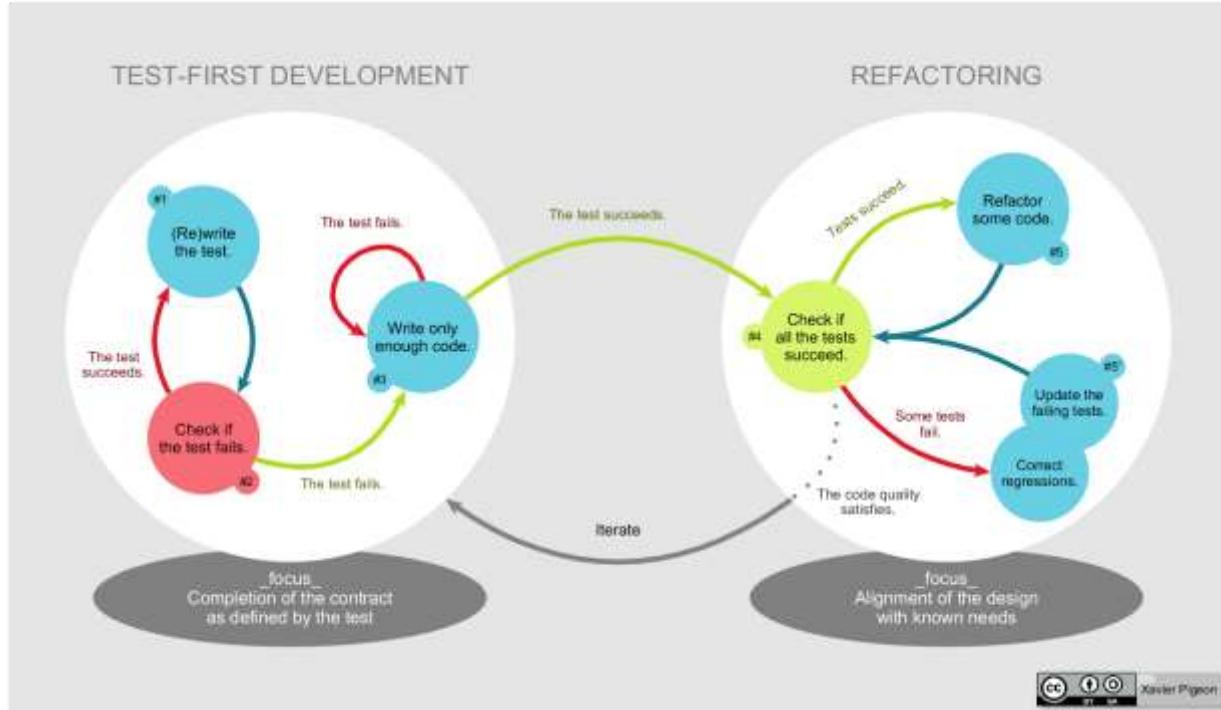
```
>>> test_gcd()  
>>>
```

Dacă toate testele au trecut – codul este testat, e conform specificațiilor și nu s-au introdus erori (au trecut și testele scrise anterior)

TDD Pas 5 – Refactorizare cod

- restructurarea codului folosind refactorizări

TDD - Refactorizare



By Xarawn - Own work, CC BY-SA 4.0,
<https://commons.wikimedia.org/w/index.php?curid=44782343>

Refactorizare

Restructurarea codului, alterând structura internă fără a modifica comportamentul observabil.

Scopul este de a face codul mai ușor de:

- înțeles
- întreținut
- extins

Semnale că este nevoie de refactorizare (**code smell**) – elemente ce pot indica probleme mai grave de proiectare:

- **Cod duplicat**
- **Metode lungi**
- **Liste lungi de parametrii**
- **Instrucțiuni condiționale care determină diferențe de comportament**

Refactorizare: Redenumire funcție/variabilă

- o redenumim funcția/variabila alegând un nume sugestiv

```
def verify(k):
    """
        Verify if a number is prime
        nr - integer number, nr>1
        return True if nr is prime
    """
    l = 2
    while l<k and k % l>0:
        l=l+1
    return l>=k
```

```
def isPrime(nr):
    """
        Verify if a number is prime
        nr - integer number, nr>1
        return True if nr is prime
    """
    div = 2 #search for divider
    while div<nr and nr % div>0:
        div=div+1
    #if the first divider is the
    # number itself than nr is prime
    return div>=nr;
```

Refactorizare: Extragerea de metode

- o parte dintr-o funcție se transformă într-o funcție separată
- o expresie se transformă într-o funcție

<pre>def startUI(): list=[] print (list) #read user command menu = """ Enter command: 1-add element 0-exit """ print(menu) cmd=input("") while cmd!=0: if cmd==1: nr=input("Give element:") add(list, nr) print list #read user command menu = """ Enter command: 1-add element 0-exit """ print(menu) cmd=input("") startUI()</pre>	<pre>def getUserCommand(): """ Print the application menu return the selected menu """ menu = """ Enter command: 1-add element 0-exit """ print(menu) cmd=input("") return cmd def startUI(): list=[] print list cmd=getUserCommand() while cmd!=0: if cmd==1: nr=input("Give element:") add(list, nr) print list cmd=getUserCommand() startUI()</pre>
---	--

Refactorizare: Substituire algoritm

```
def isPrime(nr):
    """
        Verify if a number is prime
        nr - integer number, nr>1
        return True if nr is prime
    """
    div = 2 #search for divider
    while div<nr and nr % div>0:
        div=div+1
    #if the first divider is the
    # number itself than nr is prime
    return div>=nr;
```

```
def isPrime(nr):
    """
        Verify if a number is prime
        nr - integer number, nr>1
        return True if nr is prime
    """
    for div in range(2,nr):
        if nr%div == 0:
            return False
    return True
```

Calculator – versiune procedurală

```
.....  
def run():  
    """  
        Implement the user interface  
    """  
    calc = reset_calc()  
    finish = False  
    while not finish:  
        printCurrent(calc)  
        printMenu()  
        m = input().strip()  
        if (m == 'x'):  
            finish = True  
        elif (m == '+'):  
            addToCalc(calc)  
        elif (m == 'c'):  
            calc = reset_calc()  
        elif (m == 'u'):  
            undo(calc)  
        else:  
            print ("Invalid command")  
  
    print ("By!!!")  
.....  
# run the test - invoke the test function  
test_gcd()  
test_rational_add()  
test_calculator_add()  
test_undo()  
  
run()
```

Funcții pure, funcții ca și obiecte

Funcție pură (pure function): funcție care nu are efecte secundare. Face o mapare intre datele de intrare (parametrii funcției) si valoarea returnata. Nu modifica parametrii de intrare, nu folosește variabile globale, nu are efecte secundare. Pentru același intrări produce tot timpul același rezultate.

Este de dorit (nu este tot timpul posibil) sa avem funcții pure pentru ca ele sunt:

- Ușor de înțeles/folosit/refolosit
- Ușor de testat
- Stau la baza programării funcționale
- Facilitează paralelizare/memoizare

Funcțiile in python sunt obiecte

- Se pot asigna la variabile
- Pot fi folosite ca parametru la alte funcții
- Se pot adăuga in containere (liste, dicționare, etc)

<pre>def patrat(a): return a*a</pre> <pre>#assign a function to a variable func = patrat print(func(-2)) func = modul print(func(-2))</pre> <pre>def aplica(lista, fnc): """ Aplica functia fnc pe fiecare element al listei Lista - Lista de elemente fnc - functie cu un singur parametru returneaza o lista noua """ rez = [] for el in lista: rez.append(fnc(el)) return rez</pre> <pre>print(aplica([1,2,-3,5,-2], patrat))</pre>	<pre>def modul(a): if a<0: return -a return a</pre> <pre>#list of functions lstFct = [patrat,modul] for fct in lstFct: print(fct(-2))</pre>
---	---

In python toate sunt obiecte. Funcțiile sunt „first class objects”

Programare modulară

Descompunerea programului în module (componente separate interschimbabile) având în vedere:

- separarea conceptelor
- coeziunea elementelor dintr-un modul
- cuplarea între module
- întreținerea și reutilizarea codului
- arhitectura stratificată

Modulul este o unitate structurală separată, interschimbabilă cu posibilitatea de a comunica cu alte module.

O colecție de funcții și variabile care implementează o funcționalitate bine definită

Modul în Python

Un modul în Python este un fișier ce conține instrucțiuni și definiții Python.

Modul

- **nume:** Numele fișierului este numele modulului plus extensia “.py”
 - variabila `__name__`
 - este `__main__` dacă modulul este executat de sine stătător
 - este numele modulului
- **docstring:** Comentariu multiline de la începutul modulului. Oferă o descriere a modulului: ce conține, care este scopul, cum se folosește, etc.
 - Variabila `__doc__`
- **instrucțiuni:** definiții de funcții, variabile globale per modul, cod de inițializare

Import de module

Modulul trebuie importat înainte de a putea folosi.

Instrucțiunea import:

- Caută în namespace-ul global, dacă deja există modulul înseamnă ca a fost deja importat și nu mai e nevoie de alte acțiuni
- Caută modulul și dacă nu găsește se aruncă o eroare **ImportError**
- Dacă modulul s-a găsit, se execută instrucțiunile din modul.

Instrucțiunile din modul (inclusiv definițiile de funcții) se execută doar o singură dată (prima dată când modulul este importat în aplicație).

```
from doted.package[module] import {module, function}
```

```
from utils.numericlib import gcd

#invoke the gcd function from module utils.numericlib
print gcd(2,6)

from rational import *

#invoke the rational_add function from module rational
print rational_add(2,6,1,6)

import ui.console

#invoke the run method from the module ui.console
ui.console.run()
```

Calea unde se caută modulele (Module search path)

Instrucțiunea `import` caută fișierul `modulname.py` în:

- directorul curent (directorul de unde s-a lansat aplicația)
- în lista de directoare specificată în variabila de mediu **PHYTONPATH**
- în lista de directoare specificată în variabila de mediu **PYTHONHOME** (este calea de instalare Python; de exemplu pe Unix, în general este `./usr/local/lib/python`).

Inițializare modul

Modulul poate conține orice instrucțiuni. Când modulul este importat prima dată se execută toate instrucțiunile. Putem include instrucțiuni (altele decât definițiile de funcții) care inițializează modulul.

Domeniu de vizibilitate în modul

La import:

- se creează un nou spațiu de nume
- variabilele și funcțiile sunt introduse în noul spațiu de nume
- doar numele modulului (`__name__`) este adăugat în spațiul de nume curent.

Putem folosi instrucțiunea built-in `dir()` `dir(module_name)` pentru a examina conținutul modulului

```
#only import the name ui.console into the current symbol table
import ui.console

#invoke run by providing the doted notation ui.console of the
package
ui.console.run()

#import the function name gcd into the local symbol table
from utils.numericlib import gcd

#invoke the gcd function from module utils.numericlib
print gcd(2,6)

#import all the names (functions, variables) into the local
symbol table
from rational import *

#invoke the rational_add function from module rational
print rational_add(2,6,1,6)
```

Pachete în Python

Modalitate prin care putem structura modulele.

Dacă avem mai multe module putem organiza într-o structură de directoare

Putem referi modulele prin notația pachet.modul

Fiecare director care conține pachete trebuie să conțină un fișier `__init__.py`. Acesta poate conține și instrucțiuni (codul de initializare pentru pachet)

Eclipse + PyDev IDE (Integrated Development Environment)

Eclipse: Mediu de dezvoltare pentru python (printre altele).

Pydev: Plugin eclipse pentru dezvoltare aplicații Python în Eclipse

Permite crearea, rularea, testarea, depanarea de aplicații python

Instalare:

- Python 3.x
- Eclipse (www.eclipse.org)
- Instalat pluginul de PyDev (www.pydev.org/)

Detalii pe: pydev.org

http://pydev.org/manual_101_install.html

Elemente de baza Eclipse

- **Proiect**
- **Editor Python**
- **ProjectExplorer: Pachete/Module**
- **Outline: Funcții**
- **Rulare/Depanare programe**

Cum organizăm aplicația pe module și pachete

Se creează module separate pentru:

- Interfață utilizator - Funcții legate de interacțiunea cu utilizatorul. Conține instrucțiuni de citire tipărire, este singurul modul care conține tipărire-citire
- Domeniu (Domain / Application) – Conține funcții legate de domeniul problemei
- Infrastructură – Funcții utilitare cu mare potențial de refolosire (nu sunt strict legate de domeniul problemei)
- Coordonator aplicație – Inițializare/configurare și pornire aplicație

Calculator – versiune modulară

The screenshot shows the Eclipse PyDev interface with the following details:

- Title Bar:** PyDev - calculatorModular/domain/rational.py - Eclipse
- Menu Bar:** File, Edit, Source, Refactoring, Navigate, Search, Project, Pydev, Run, Window, Help
- Toolbars:** Standard, PyDev, Quick Access, Java, PyDev
- PyDev Package Explorer:** Shows the project structure:
 - calc
 - qcalc
 - console
 - calculator
 - rational
 - numericlib- Outline View:** Shows the rational_add() function and its test.
- Type Filter Text:** type filter text
- Code Editor:** Displays the rational.py code:

```
7 def rational_add(a1, a2, b1, b2):
8     """
9         Return the sum of two rational numbers.
10        a1,a2,b1,b2 integer numbers, a2<>0 and b2<>0
11        return a List with 2 integer numbers, representing a rational number.
12        Raise ValueError if the denominators are zero.
13    """
14    if a2 == 0 or b2 == 0:
15        raise ValueError("0 denominator not allowed")
16    c = [a1 * b2 + a2 * b1, a2 * b2]
17    d = gcd(c[0], c[1])
18    c[0] = c[0] // d
19    c[1] = c[1] // d
20    return c
21
22 def test_rational_add():
23     """
24         Test function for rational_add
25     """
26     assert rational_add(1, 2, 1, 3) == [5, 6]
```
- Console View:** Shows the path to the qcalc.py file and its help text:

```
D:\Istvan\fp2014\curs\wsp\calculatorModular\qcalc.py
+ for adding a rational number
c to clear the calculator
u to undo the last operation
x to close the calculator
```
- Status Bar:** Writable, Insert, 17:1

Curs 3. Programare modulară

- Refactorizare
- Module

Curs 4. Tipuri definite de utilizator

- Organizarea aplicației pe module și pachete
- Excepții
- Tipuri definite de utilizator

Referințe

- The Python language reference. <http://docs.python.org/py3k/reference/index.html>
- The Python standard library. <http://docs.python.org/py3k/library/index.html>
- The Python tutorial. <http://docs.python.org/tutorial/index.html>
- Kent Beck. *Test Driven Development: By Example*. Addison-Wesley Longman, 2002. See also Test-driven development. http://en.wikipedia.org/wiki/Test-driven_development
- Martin Fowler. *Refactoring. Improving the Design of Existing Code*. Addison-Wesley, 1999. See also <http://refactoring.com/catalog/index.html>

Curs 4. Principii de organizarea aplicațiilor

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

Curs 3. Programare modulară

- Refactorizare
- Module
- Test Driven Development

Organizarea aplicației pe funcții și module

Scop:

Organizarea aplicației astfel încât:

- Sa fie ușor să regăsim partea de cod care implementează un anumit lucru. Fiecare concept din aplicație să aibă un loc bine definit unde se găsește implementarea conceptului
- Sa fie ușor de adăugat funcționalități noi. Ideal când adaug ceva nou în aplicație ar trebui să modific cat mai puțin din ceea ce există deja
- Sa fie ușor de testat automat. Testarea este fundamentală în crearea aplicațiilor.
- Sa permite colaborarea – mai mulți programatori lucrează pe același proiect. Sa fie ușor să înțeleagă parții ale aplicației (posibil scris de alt coleg) fără a fi nevoie să știi 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
- Impossibilitatea de a testa
- Impossibilitatea 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 dependentelor

Separarea interfeței de implementare, ascunderea detaliilor de implementare / reprezentare

Interfața funcției: este signatura funcției + specificațiile

Interfața modul: signatura și specificațiile tuturor funcțiilor din modul

Codul client – codul care folosește funcția modulului

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

#Vers1	#Vers Bad
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))	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])

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 signatura funcțiilor reset, add_to, get_total și la specificațiile lor.

Cuplare

Măsoară intensitatea legăturilor dintre module/functii

Cu cât există mai multe conexiuni între module cu atât modulul este mai greu de înțeles, întreținut, refolosit și devine dificilă izolarea problemelor ⇒ 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/functii 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

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 şablon arhitectural care permite dezvoltarea de sisteme flexibile în care componentele au un grad ridicat de independență

- Fiecare strat comunică doar cu startul 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)
 - asamblează și pornește aplicația

Layered Architecture – exemplu

```
#Ui
def filterScoreUI():
    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(l, 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

The screenshot shows the Eclipse PyDev interface with the following details:

- Project Explorer:** Shows a package named "modularqcalc" containing several modules: domain, ui, and utils.
- Code Editor:** Displays Python code for a calculator module. The code defines functions for adding rational numbers and undoing operations.
- Console:** Shows the output of running the main.py script, which includes the total value [0, 1] and a calculator menu with options for addition, clearing, undoing, and exiting.
- Bottom Status Bar:** Shows the current time as 15:20.

```
from domain.calculator import calc_get_total, reset_calc, calc_add, undo
def addToCalc(calc):
    """
    Read a rational number and add to the current total
    domain - calculator
    """
    m = input("Give nominator:")
    n = input("Give denominator:")
    try:
        calc_add (calc, int(m), int(n))
    except ValueError:
        print ("Enter integers for m, n, with not null n")
def undoCalc(calc):
    try:
        undo(calc)
    except ValueError as e:
        print (e)
```

Erori și exceptii

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 semnala o situație excepțională, o eroare
- ex. Nu sunt satisfăcute precondițiile

```
>>> 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 îintrerup execuția normală a instrucțiunilor

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

Programul continuă execuția în punctul în care excepția este tratată (rezolvată) sau îintrerupe 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 exceptiilor (Exception handling)

Procesul sistematic prin care exceptiile 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
```

Exceptiile 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 exceptia (exceptia a fost aruncată)

Dacă exceptia 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 să 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 semnalat o eroare – semnalată situația în care funcția nu poate respecta post condiția, nu poate furniza rezultatul promis în specificații
- Putem folosi pentru a semnalata 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 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 exceptii

```
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 bazata pe obiecte
- Principii de definire a tipurilor utilizator

Curs 5: Tipuri definite de utilizator

- Programare orientată obiect
- Principii de definire a tipurilor utilizator
- TAD - Tip abstract de date

Curs 4.

- Organizarea aplicației pe funcții, module și pachete
- Arhitectura stratificată
- Exceptii

Excepții recapitulare

Funcțiile pot **arunca excepții** pentru a semnala situații în care funcția nu poate efectua operația promisă:

```
int("bla")  
  
Traceback (most recent call last):  
  File "C:\Curs5\ex.py", line 6, in <module>  
    int("bla")  
ValueError: invalid literal for int() with base 10: 'bla'
```

Funcțiile create de noi pot arunca și ele excepții pentru a semnala că operația nu a putut fi executată:

```
def myBeautifulFunction():  
    ....  
    raise ValueError("Motivul pentru care arunc exceptie")  
    ....  
  
myBeautifulFunction()  
  
Traceback (most recent call last):  
  File "C:\Curs5\ex.py", line 13, in <module>  
    myBeautifulFunction()  
  File "C:\Curs5\ex.py", line 10, in myBeautifulFunction  
    raise ValueError("Motivul pentru care arunc exceptie")  
  
ValueError: Motivul pentru care arunc exceptie
```

Tratarea exceptiilor

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

```
try:  
    #cod in care posibil apar exceptii (arunca exceptii)  
    myBeautifulFunction()  
    #....  
except ValueError as ex:  
    #executa acest cod daca a aparut eroarea ValueError  
    print(ex) #putem accesa obiectul care a aruncat eroare  
except ZeroDivisionError:  
    #executa acest cod daca a aparut eroarea ZeroDivisionError  
    pass  
finally:  
    #executa tot timpul(si daca a aparut exceptie si daca nu)  
    pass
```

Exceptiile 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 exceptia

Review calculator modular

Câteva probleme:

- Starea calculatorului:
 - varianta cu variabilă globală:
 - avem mai multe variabile globale care pot fi cu ușurință accesate din exterior (posibil stricând starea calculatorului)
 - variabila globală face testarea mai dificilă
 - nu este o legătură clară între aceste variabile (starea calculatorului este împrăștiat în cod)
 - varianta fără variabile globale:
 - starea calculatorului este expus (nu există garanții ca metodele se apelează cu un obiect care reprezintă calculatorul)
 - trebuie să transmitem starea, ca parametru pentru fiecare funcție legată de calculator
- Numere raționale
 - reprezentarea numerelor este expusa: ex: rez=
suma(total[0],total[1],a,b) , putem cu ușurință altera numărul rațional (ex. Facem total[0] = 8 care posibil duce la încălcarea reguli cmmdc(a,b) ==1 pentru orice numărul rațional a/b)
 - codul pentru adunare, înmulțire, etc de numere raționale este diferit de modul în care facem operații cu numere întregi. Ar fi de preferat să putem scrie $r = r_1 + r_2$ unde r, r_1, r_2 sunt numere raționale

Programare orientată obiect

Este metodă de proiectare și dezvoltare a programelor:

- Oferă o abstractizare puternică și flexibilă
- Programatorul poate exprima soluția în mod mai natural (se concentrează pe structura soluției nu pe structura calculatorului)
- Descompune programul într-un set de obiecte, obiectele sunt elementele de bază
- Obiectele interacționează pentru a rezolva problema, există relații între clase
- Clasele introduc tipuri noi de date, modelează elemente din spațiul problemei, fiecare obiect este o instanță a unui tip de date (clasă)

Clasă

Definește în mod abstract caracteristicile unui lucru.

Descrie două tipuri de atribute:

- câmpuri (proprietăți) – descriu caracteristicile
- metode (operații) – descriu comportamentul

Clasele se folosesc pentru crearea de noi tipuri de date (tipuri de date definite de utilizator)

Tip de date:

- domeniu
- operații

Clasele sunt folosite ca un şablon pentru crearea de obiecte (instanțe), clasa definește elementele ce definesc starea și comportamentul obiectelor.

Definiție de clasă în python

```
class MyClass:  
    <statement 1>  
    ....  
    <statement n>
```

Este o instrucțiune executabilă, introduce un nou tip de date cu numele specificat.

Instrucțiunile din interiorul clasei sunt în general definiții de funcții, dar și alte instrucțiuni sunt permise

Clasa are un spațiu de nume propriu, definițiile de funcții din interiorul clasei (metode) introduc numele funcțiilor în acest spațiu de nume nou creat. Similar și pentru variabile

Obiect

Obiect (instanță) este o colecție de date și funcții care operează cu aceste date

Fiecare obiect are un tip, este de tipul clasei asociate: este instanță unei clase

Obiectul:

- înglobează o stare: valorile câmpurilor
- folosind metodele:
 - putem modifica starea
 - putem opera cu valorile ce descriu starea obiectelor

Fiecare obiect are propriul spațiu de nume care conține câmpurile și metodele.

Creare de obiecte. Creare de instanțe a unei clase (`__init__`)

Instantierea unei clase rezulta in obiecte noi (instanțe). Pentru crearea de obiecte se folosește notație similară ca și la funcții.

```
x = MyClass()
```

Operația de instantiere (“apelul” unei clase) creează un obiect nou, obiectul are tipul `MyClass`

O clasă poate defini metoda specială `__init__` care este apelată în momentul instantierii

```
class MyClass:  
    def __init__(self):  
        self.someData = []
```

`__init__` :

- creează o instanță
- folosește “self” pentru a referi instanța (obiectul) curent (similar cu “this” din alte limbaje orientate obiect)

Pot avea metoda `__init__` care are și alți parametrii în afară de `self`

Câmpuri

```
x = RationalNumber(1,3)
y = RationalNumber(2,3)
x.m = 7
x.n = 8
y.m = 44
y.n = 21
```

```
class RationalNumber:
    """
        Abstract data type for rational numbers
        Domain: {a/b where a and b are integer numbers b!=0}
    """

    def __init__(self, a, b):
        """
            Creates a new instance of RationalNumber
        """
        #create a field in the rational number
        #every instance (self) will have this field
        self.n = a
        self.m = b
```

self.n = a vs n=a

- 1 Creează un atribut pentru instanța curentă
- 2 Creează o variabilă locală funcției

Metode

Metodele sunt funcții definite în interiorul clasei care au acces la valorile câmpurilor unei instanțe.

În Python metodele au un prim argument: instanța curentă
Toate metodele primesc ca prim parametru obiectul curent (self)

```
def testCreate():
    """
        Test function for creating rational numbers
    """
    r1 = RationalNumber(1,3) #create the rational number 1/3
    assert r1.getNominator()==1
    assert r1.getDenominator()==3
    r1 = RationalNumber(4,3) #create the rational number 4/3
    assert r1.getNominator()==4
    assert r1.getDenominator()==3

class RationalNumber:
    """
        Abstract data type rational numbers
        Domain: {a/b where a,b integer numbers, b!=0, greatest common divisor
a, b =1}
    """
    def __init__(self, a, b):
        """
            Initialize a rational number
            a,b integer numbers
        """
        self.__nr = [a, b]

    def getDenominator(self):
        """
            Getter method
            return the denominator of the rational number
        """
        return self.__nr[1]

    def getNominator(self):
        """
            Getter method
            return the nominator of the method
        """
        return self.__nr[0]
```

Metode speciale. Supraîncărcarea operatorilor. (Operator overloading)

`__str__` - conversie in tipul string (print representation)

```
def __str__(self):
    """
        provide a string representation for the rational number
        return a string
    """
    return str(self.__nr[0])+"/"+str(self.__nr[1])
```

`__lt__`, `__le__`, `__gt__`, `__ge__` - comparații (<, <=, >, >=)

<pre>def testCompareOperator(): """ Test function for < > """ r1 = RationalNumber(1, 3) r2 = RationalNumber(2, 3) assert r2>r1 assert r1<r2</pre>	<pre>def __lt__(self, ot): """ Compare 2 rational numbers (Less than) self the current instance ot a rational number return True if self<ot, False otherwise """ if self.getFloat()<ot.getFloat(): return True return False</pre>
---	---

`__eq__` - verify if equals

<pre>def testEqual(): """ test function for == """ r1 = RationalNumber(1, 3) assert r1==r1 r2 = RationalNumber(1, 3) assert r1==r2 r1 = RationalNumber(1, 3) r1 = r1.add(RationalNumber(2, 3)) r2 = RationalNumber(1, 1) assert r1==r2</pre>	<pre>def __eq__(self, other): """ Verify if 2 rational are equals other - a rational number return True if the instance is equal with other """ return self.__nr==other.__nr</pre>
--	--

Operator overloading

__add__(self, other) - pentru a folosi operatorul “+”

<pre>def testAddOperator(): """ Test function for the + operator """ r1 = RationalNumber(1, 3) r2 = RationalNumber(1, 3) r3 = r1+r2 assert r3 == RationalNumber(2, 3)</pre>	<pre>def __add__(self,other): """ Overload + operator other - rational number return a rational number, the sum of self and other """ return self.add(other)</pre>
---	--

Metoda **__mul__(self, other)** - pentru operatorul “*”

Metoda **__setitem__(self,index, value)** – dacă dorim ca obiectele noastre să se comporte similar cu liste/dicționare, să putem folosi “[]”

```
a = A()
a[index] = value
```

__getitem__(self, index) – să putem folosi obiectul ca și o secvență

```
a = A()
for el in a:
    pass
```

__len__(self) - pentru len

__getslice__(self,low,high) - pentru operatorul de slicing

```
a = A()
b = a[1:4]
```

__call__(self, arg) - to make a class behave like a function, use the “()”

```
a = A()
a()
```

Vizibilitate și spații de nume în Python

Spațiu de nume (*namespace*) este o mapare între nume și obiecte
Namespace este implementat în Python folosind dicționarul

Cheie: Nume

Valoare – Object

Clasa introduce un nou spațiu de nume

Metodele sunt într-un spațiu de nume separat, spațiu de nume corespunzător clasei.

```
class Student:  
    def __init__(self, nume, prenume):  
        self.__nume = nume  
        self.__prenume = prenume  
  
    def getNume(self):  
        return self.__nume  
  
    def getFullName(self):  
        return "{} {}".format(self.__nume, self.__prenume)  
  
print (Student.__dict__)  
{'__module__': '__main__', '__doc__': None, 'getNume': <function  
Student.getNume at 0x0089F8A0>, '__dict__': <attribute '__dict__' of  
'Student' objects>, '__init__': <function Student.__init__ at 0x0089F810>,  
'getFullName': <function Student.getFullName at 0x0089F858>, '__weakref__':  
<attribute '__weakref__' of 'Student' objects>}
```

Fiecare instanță de clasa are propriu spațiu de nume (aici se țin atributele instanței)

```
st1 = Student("Ion", "Vasilescu")  
print (st1.__dict__)  
{'_Student__nume': 'Ion', '_Student__prenume': 'Ionescu'}
```

Toate regulile (legare de nume, vizibilitate/scope, parametrii formali/actuali, etc.) legate de denumiri (funcții, variabile) sunt același pentru atributele clasei (metode, câmpuri) ca și pentru orice alt nume în Python, doar trebuie luat în considerare că avem un namespace dedicat clasei

Atribute de clasă vs atribute de instanțe

Variabile membre (câmpuri)

- atribute de instanțe – valorile sunt unice pentru fiecare instanță (obiect)
- atribute de clasă – valoarea este partajată de toate instanțele clasei (toate obiectele de același tip)

```
class RationalNumber:  
    """  
        Abstract data type for rational numbers  
        Domain: {a/b where a and b are integer numbers b!=0}  
    """  
    #class field, will be shared by all the instances  
    numberofInstances = 0  
  
    def __init__(self, a, b):  
        """  
            Creates a new instance of RationalNumber  
        """  
        self.n = a  
        self.m = b  
        RationalNumber.numberofInstances+=1      # accessing class fields  
  
    def testNumberInstances():  
        assert RationalNumber.numberofInstances == 0  
        r1 = RationalNumber(1,3)  
        #show the class field numberofInstances  
        assert r1.numberofInstances==1  
        # set numberofInstances from the class  
        r1.numberofInstances = 8  
        assert r1.numberofInstances==8  #access to the instance field  
        assert RationalNumber.numberofInstances==1  #access to the class field  
  
    testNumberInstances()
```

Metode statice

Funcții din clasă care nu operează cu o instanță.

```
class RationalNumber:  
    #class field, will be shared by all the instances  
    numberofInstances = 0  
  
    def __init__(self,n,m):  
        """  
            Initialize the rational number  
            n,m - integer numbers  
        """  
        self.n = n  
        self.m = m  
        RationalNumber.numberofInstances+=1  
  
    @staticmethod  
    def getTotalNumberOfInstances():  
        """  
            Get the number of instances created in the app  
        """  
        return RationalNumber.numberofInstances  
  
    @classmethod  
    def fromString(cls,s):  
        """  
            Create a Rational number object from its string representation  
            cls - class  
            s - string representation 1/3  
        """  
        parts = s.split("/")  
        return RationalNumber(int(parts[0]),int(parts[1]))  
  
    def testNumberOfInstances():  
        """  
            test function for getTotalNumberOfInstances  
        """  
        assert RationalNumber.getTotalNumberOfInstances() == 0  
        r1 = RationalNumber(2, 3)  
        assert RationalNumber.getTotalNumberOfInstances() == 1  
  
    testNumberOfInstances()
```

ClassName.attributeName – folosit pentru a accesa un atribut asociat clasei (câmp, metoda)

Decoratorul `@staticmethod` este folosit pentru a marca o funcție statică. Aceste funcții nu au ca prim argument (`self`) obiectul curent.

Decoratorul `@classmethod` similar cu `@staticmethod` dar se primește un prim parametru clasa

Principii pentru crearea de noi tipuri de date

Încapsulare

Datele care reprezintă starea și metodele care manipulează datele sunt strâns legate, ele formează o unitate coezivă.

Starea și comportamentul ar trebui încapsulat în același unitate de program (clasa)

Ascunderea informațiilor

Reprezentarea internă a obiectelor (a stării) trebuie protejată față de restul aplicației.

Ascunderea reprezentării protejează integritatea datelor și nu permite modificarea stării din exteriorul clasei, astfel se evită setarea, accidentală sau voita, unei stări inconsistente.

Clasa comunica cu exteriorul doar prin interfața publică (mulțimea tuturor metodelor vizibile în exterior) și ascunde orice detalii de implementare (modul în care am reprezentat datele, algoritmii folosiți, etc).

De ce:

Definirea unei interfețe clare și ascunderea detaliilor de implementare asigură ca alte module din aplicație să nu pot face modificări care ar duce la stări inconsistente. Permite evoluția ulterioară (schimbare reprezentare, algoritmi etc) fără să afecteze restul aplicației

Limitați interfața (metodele vizibile în exterior) astfel încât să existe o libertate în modificarea implementării (modificare fără a afecta codul client)

Codul client trebuie să depindă doar de interfața clasei, nu de detaliile de implementare. Dacă folosiți acest principiu, atunci se pot face modificări fără a afecta restul aplicației

Membri publici. Membrii privați – Ascunderea implementării în Python

Trebuie să protejăm (ascundem) reprezentarea internă a clasei (implementarea)

În Python ascunderea implementării se bazează pe convenții de nume.

`_name` sau `__name` pentru un atribut semnalează faptul că atributul este “privat”

Un nume care începe cu `_` sau `__` semnalează faptul că atributul (câmp, metode) ar trebui tratat ca fiind un element care nu face parte din interfața publică. Face parte din reprezentarea internă a clasei, nu ar trebui accesat din exterior.

Recomandări

- Creați metode pentru a accesa câmpurile clasei (getter)
- folosiți convențiile de nume , pentru a delimita interfața publică a clasei de detaliile de implementare
- Codul client ar trebui să funcționeze (fără modificări) chiar dacă schimbăm reprezentarea internă, atât timp cât interfața publică rămâne neschimbată. Clasa este o abstractizare, o cutie neagră (black box)
- Specificațiile funcțiilor trebuie să fie independente de reprezentare

Cum creăm clase

Folosim Dezvoltare dirijată de teste

Specificațiile (documentația) pentru clase includ:

- scurtă descriere
- domeniul – ce fel de obiecte se pot crea. În general descrie câmpurile clasei
- Constrângeri ce se aplică asupra datelor membre: Ex. Invariant – condiții care sunt adevărate pentru întreg ciclu de viață al obiectului

```
class RationalNumber:  
    """  
        Abstract data type rational numbers  
        Domain:{a/b where a,b integer numbers, b!=0, greatest common divisor a, b =1}  
        Invariant:b!=0, greatest common divisor a, b =1  
    """  
    def __init__(self, a, b):
```

Se creează funcții de test pentru:

- Crearea de instanțe
- Fiecare metodă din clasă

Câmpurile clasei (reprezentarea) se declară private (`__nume`). Se creează metode getter pentru a accesa câmpurile clasei

Tipuri abstracte de date (Abstract data types)

Tip abstract de date:

- operațiile sunt specificate independent de felul în care operația este implementată
- operațiile sunt specificate independent de modul de reprezentare a datelor

Un tip abstract de date este: Tip de date + Abstractizarea datelor + Încapsulare

Review Calculator rațional – varianta orientat obiect

Putem schimba cu ușurință reprezentarea internă pentru clasa RationalNumber (folosim a,b în loc de lista [a,b])

Curs 5: Tipuri definite de utilizator

- Programare orientată obiect
- Principii de definire a tipurilor utilizator
- Tip abstract de date

Curs 6: Principii de proiectare

- Diagrame UML
- řabloane GRASP

Curs 6: Principii de proiectare

- Diagrame UML
- řabloane GRASP
- Arhitectură stratificată:
 - Ui – Service(GRASP Controller) – Domain - Repository

Curs 5: Tipuri definite de utilizator

- Programare orientată obiect
- Principii de definire a tipurilor utilizator
- Tip abstract de date

Diagrame UML

Unified Modeling Language (UML) - este un limbaj standardizat de modelare destinat vizualizării, specificării, modelării și documentării aplicațiilor.

UML include un set de notații grafice pentru a crea modele vizuale ce descriu sistemul.

Diagrame de clase

Diagrama UML de clase (UML Class diagrams) descrie structura sistemului prezentând clasele, atributele și relațiile între aceste clase

<p>RationalNumber</p> <p>+__nr</p> <p>+getNominator(): int</p> <p>+getDenominator(): int</p> <p>+add(nr: RationalNumber): RationalNumber</p>	<pre>class RationalNumber: def __init__(self, a, b): """ Initialize a rational number a,b integer numbers """ self.__nr = [a, b] def getDenominator(self): """ Getter method return the denominator """ return self.__nr[1] def getNominator(self): """ Getter method return the nominator """ return self.__nr[0] def add(self, a):</pre>
---	--

Clasele sunt reprezentate prin dreptunghiuri ce conțin trei zone:

- Partea de sus – numele clasei
- Partea din mijloc – câmpurile/atributele clasei
- Partea de jos – metodele/operațiile

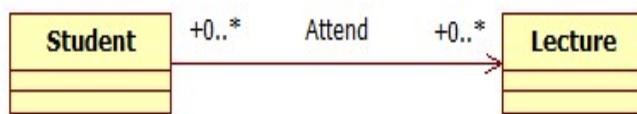
Relații UML

O relație UML este un termen general care descrie o legătură logică între două elemente de pe o diagramă de clase.

Un *Link* este relația între obiectele de pe diagramă. Este reprezentată printr-o linie care conectează două sau mai multe dreptunghiuri.

Asocieri

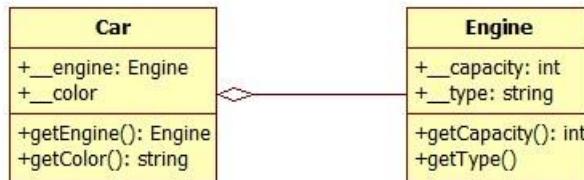
Asocierile binare se reprezintă printr-o linie între două clase.



O asociere poate avea nume, capetele asocierii pot fi anotate cu nume de roluri, multiplicitate, vizibilitate și alte proprietăți. Asocierea poate fi unidirectională sau bi-directională.

Agregare

Agregarea este o asociere specializată. Este o asociere ce reprezintă relația de parte-întreg (part-whole) sau apartenență (part-of).



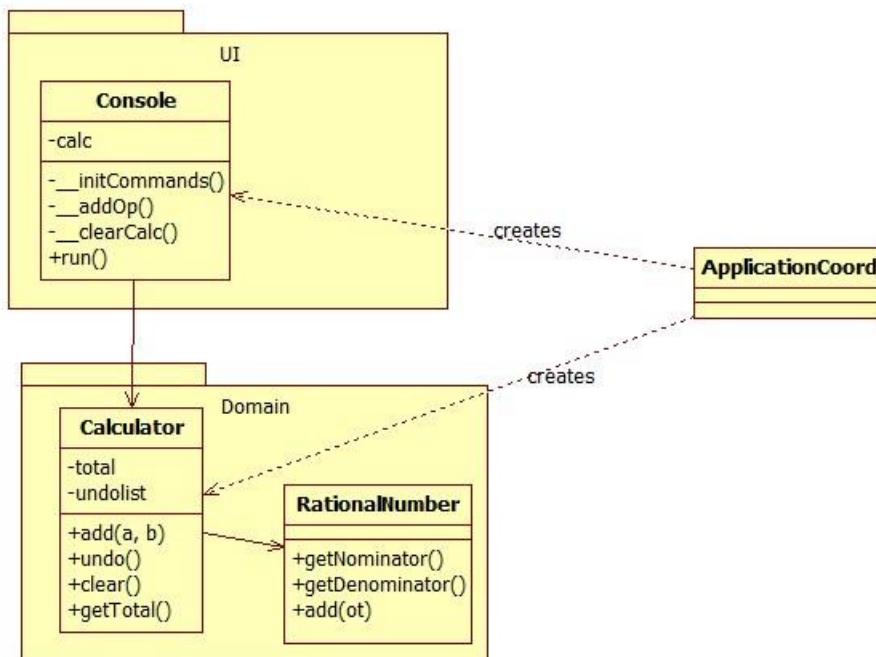
<pre>class Car: def __init__(self, eng, col): """ Initialize a car eng - engine col - string, ie White """ self.__engine = eng self.__color = col def getColor(self): """ Getter method for color return string """ return self.__color def getEngine(self): """ Getter method for engine return engine """ return self.__engine</pre>	<pre>class Engine: def __init__(self, cap, type): """ initialize the engine cap positive integer type string """ self.__capacity = cap self.__type = type def getCapacity(self): """ Getter method for the capacity """ return self.__capacity def getType(self): """ Getter method for type return string """ return self.__type</pre>
--	---

Dependențe, Pachete

- Relația de dependență este o asociere în care un element depinde sau folosește un alt element

Exemple de dependențe:

- creează instanțe
- are un parametru
- folosește un obiect în interiorul unei metode



Principii de proiectare

Creează aplicații care:

Sunt ușor de înțeles, modificat, întreținut, testat

Clasele – abstracte, încapsulare, ascunderea reprezentării, ușor de testat, ușor de folosit și refolosit

Scop general: gestiunea dependentelor

- Single responsibility
- Separation of concerns
- Low Coupling
- High Cohesion

Enunț (Problem statement)

scrieți un program care gestionează studenți de la o facultate
(operații CRUD – **Create Read Update Delete**)

	Funcționalități (Features)
F1	Adaugă student
F2	vizualizare studenți
F3	caută student
F4	șterge student

Plan de iterații

IT1 - F1; IT2 – F2; IT3 – F3; IT4 - F4

Scenariu de rulare (Running scenario)

user	app	description
'a'		add a student
	give student id	
1		
	give name	
'Ion'		
	new student added	
'a'		add student
	give student id	
1		
	give name	
"		
	id already exists, name can not be empty	

Arhitectură stratificată (Layered architecture)

Layer (strat) este un mecanism de structurare logică a elementelor ce compun un sistem software

Într-o arhitectură multi-strat, straturile sunt folosite pentru a aloca responsabilități în aplicație.

Layer este un grup de clase (sau module) care au același set de dependențe cu alte module și se pot refolosi în circumstanțe similare.

- User Interface Layer (View Layer, UI layer sau Presentation layer)
- Application Layer (Service Layer sau **GRASP** Controller Layer)
- Domain layer (Business Layer, Business logic Layer sau Model Layer)
- **Infrastructure Layer** (acces la date – modalități de persistență, logging, network I/O ex. Trimitere de email, sau alte servicii tehnice)

Şabloane Grasp

General Responsibility Assignment Software Patterns (or Principles) conțin recomandări pentru alocarea responsabilităților pentru clase obiecte într-o aplicație orientată obiect.

- High Cohesion
- Low Coupling
- Information Expert
- Controller
- Protected Variations
- Creator
- Pure Fabrication

High Cohesion

Alocă responsabilitățile astfel încât coeziunea în sistem rămâne ridicată

High Cohesion este un principiu care se aplică pe parcursul dezvoltării în încercarea de a menține elementele în sistem:

- responsabile de un set de activități înrudite
- de dimensiuni gestionabile
- ușor de înțeles

Coeziune ridicată (High cohesion) înseamnă ca responsabilitățile pentru un element din sistem sunt înrudite, concentrate în jurul aceluiași concept.

Împărțirea programelor în clase și starturi este un exemplu de activitate care asigură coeziune ridicată în sistem.

Alternativ, coeziune slabă (low cohesion) este situația în care elementele au prea multe responsabilități, din arii diferite. Elementele cu coeziune slabă sunt mai greu de înțeles, reutilizat, întreținut și sunt o piedică pentru modificările necesare pe parcursul dezvoltării unui sistem

Low Coupling

Alocă responsabilități astfel încât cuplarea rămâne slabă (redusă)

Low Coupling încurajează alocarea de responsabilități astfel încât avem:

- dependențe puține între clase;
- impact scăzut în sistem la schimbarea unei clase;
- potențial ridicat de refolosire;

Forme de cuplare:

- TypeX are un câmp care este de TypeY.
- TypeX are o metodă care referă o instanță de tipul TypeY în orice formă (parametrii, variabile locale, valoare returnată, apel la metode)
- TypeX este derivat direct sau indirect din clasa TypeY.

Information Expert

Alocă responsabilitatea clasei care are toate informațiile necesare pentru a îndeplini sarcina

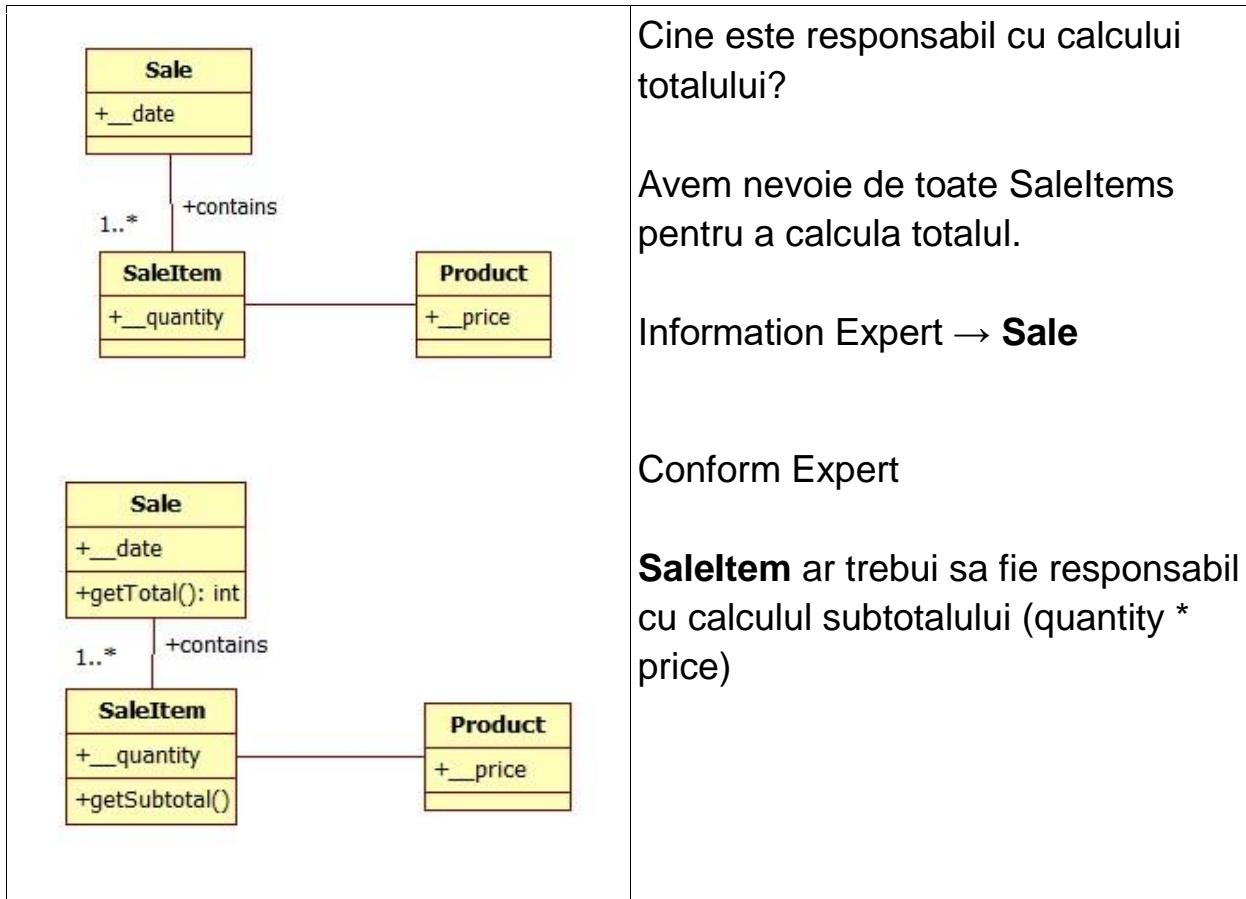
Information Expert este un principiu care ajută să determinăm care este clasa potrivită care ar trebui să primească responsabilitatea (o metodă nouă, un câmp, un calcul).

Folosind principiu Information Expert încercăm să determinăm care sunt informațiile necesare pentru a realiza ce se cere, determinăm locul în care sunt aceste informații și alocăm responsabilitatea la clasa care conține informațiile necesare.

Information Expert conduce la alocarea responsabilității în clasa care conține informația necesară pentru implementare. Ajută să răspundem la întrebarea Unde se pune – metoda, câmpul

Information Expert

Point of Sale application



1. Menține încapsularea
2. Promovează cuplare slabă
3. Promovează clase puternic coeziive
4. Poate să conduce la clase complexe - dezavantaj

Creator

Crearea de obiecte este o activitate importantă într-un sistem orientat obiect. Care este clasa responsabilă cu crearea de obiecte este o proprietate fundamentală care definește relația între obiecte de diferite tipuri.

Şablonul Creator descrie modul în care alocăm responsabilitatea de a crea obiecte în sistem

În general o clasa B ar trebui să aibă responsabilitatea de a crea obiecte de tip A dacă unul sau mai multe (de preferat) sunt adevărate:

- Instanța de tip B conține sau agregă instanțe de tip A
- Instanța de tip B gestionează instanțe de tip A
- Instanța de tip B folosește extensiv instanțe de tip A
- Instanța de tip B are informațiile necesare pentru a inițializa instanța A.

Work items

	Task
T1	Create Student
T2	Validate student
T3	Store student (Create repository)
T4	Add student (Create Controller)
T5	Create UI

Task: create Student

<pre>def testCreateStudent(): """ Testing student creation """ st = Student("1", "Ion", "Adr") assert st.getId() == "1" assert st.getName() == "Ion" assert st.getAddr() == "Adr"</pre>	<pre>class Student: def __init__(self, id, name, adr): """ Create a new student id, name, address String """ self.__id = id self.__name = name self.__adr = adr def getId(self): return self.__id def getName(self): return self.__name def getAddr(self): return self.__adr</pre>
---	---

Protected Variations

Cum alocăm responsabilitatea astfel încât variațiile curente și viitoare nu vor afecta sistemul (nu va fi necesar o revizuire a sistemului, nu trebuie să facem schimbări majore în sistem)?

Protected variations: Creăm o nouă clasă care încapsulează aceste variații.

Şablonul **Protected Variations** protejează elementele sistemului de variațiile/modificările altor elemente din sistem (clase, obiecte, subsisteme) încapsulând partea instabilă într-o clasă separată (cu o interfață publică bine delimitată care ulterior, folosind polimorfism, poate introduce variații prin noi implementări).

Task: Validate student

Design posibil pentru validare:

Algoritmul de validare:

- Poate fi o metoda in clasa student
- o metoda statica, o functie
- încapsulat într-o clasă separată

Poate semnala eroarea prin:

- returnare true/false
- returnare lista de erori
- excepții care conțin lista de erori

Clasă Validator : aplică Prinzipiu Protect Variation

<pre>def testStudentValidator(): """ Test validate """ validator = StudentValidator() st = Student("", "Ion", "str") try: validator.validate(st) assert False except ValueError: assert True st = Student("", "", "") try: validator.validate(st) assert False except ValueError: assert True</pre>	<pre>class StudentValidator: """ Class responsible with validation """ def validate(self, st): """ Validate a student st - student raise ValueError if: Id, name or address is empty """ errors = "" if (st.id==""): errors+="Id can not be empty;" if (st.name==""): errors+="Name can not be empty;" if (st.adr==""): errors+="Address can not be empty" if len(errors)>0: raise ValueError(errors)</pre>
---	--

Pure Fabrication

Când un element din sistem încalcă principiul coeziunii ridicate și cuplare slabă (în general din cauza aplicării succesive a şablonului expert):
Alocă un set de responsabilități la o clasă artificială (clasă ce nu reprezintă ceva în domeniul problemei) pentru a oferi coeziune ridicată, cuplare slabă și reutilizare

Pure Fabrication este o clasă ce nu reprezintă un concept din domeniul problemei este o clasă introdusă special pentru a menține cuplare slabă și coeziune ridicată în sistem.

Problema: Stocare **Student** (in memorie, fișier sau bază de date)

Expert pattern → Clasa Student este “expert”, are toate informațiile, pentru a realiza această operație

Pure Fabrication - Repository

Problema: Stocare **Student** (in memorie, fișier sau bază de date)

Expert pattern → Clasa Student este “expert”, are toate informațiile, pentru a realiza această operație

Dacă punem responsabilitatea persistenței în clasa Student, rezultă o clasă slab coeziva, cu potențial limitat de reutilizare

Soluție – Pure Fabrication

<pre>class StudentRepository { +store(st: Student) +update(st: Student) +find(id: string): Student +delete(st: Student) }</pre>	<p>Clasă creată cu responsabilitatea de a salva/persista obiecte Student</p> <p>Clasa student se poate reutiliza cu ușurință și are High cohesion, Low coupling</p> <p>Clasa StudentRepository este responsabil cu problema gestiunii unei liste de studenți (să ofere un depozit - persistent – pentru obiecte de tip student)</p>
---	---

Şablonul Repository

Un **repository** reprezintă toate obiectele de un anumit tip ca și o mulțime de obiecte.

Obiecte sunt adăugate, șterse, modificate iar codul din repository inserează, șterge obiectele dintr-un depozit de date persistent.

Task: Create repository

```
def testStoreStudent():
    st = Student("1", "Ion", "Adr")
    rep = InMemoryRepository()
    assert rep.size()==0
    rep.store(st)
    assert rep.size()==1
    st2 = Student("2", "Vasile", "Adr2")
    rep.store(st2)
    assert rep.size()==2
    st3 = Student("2", "Ana", "Adr3")
    try:
        rep.store(st3)
        assert False
    except ValueError:
        pass

class InMemoryRepository:
    """
    Manage the store/retrieval of students
    """
    def __init__(self):
        self.students = {}

    def store(self, st):
        """
        Store students
        st is a student
        raise RepositoryException if we have a student with the same id
        """
        if st.getId() in self.students:
            raise ValueError("A student with this id already exist")

        if (self.validator!=None):
            self.validator.validate(st)

        self.students[st.getId()] = st
```

GRASP Controller

Scop: decuplarea sursei de evenimente de obiectul care gestionează evenimentul. Decuplarea startului de prezentare de restul aplicației.

GRASP Controller este definit ca primul obiect după stratul de interfață utilizator. Interfața utilizator folosește un obiect controller, acest obiect este responsabil de efectuarea operațiilor cerute de utilizator.

Controller coordonează (controlează) operațiile necesare pentru a realiza acțiunea declanșată de utilizator.

Controllerul în general folosește alte obiecte pentru a realiza operația, doar coordonează activitatea.

Controllerul poate încapsula informații despre starea curentă a unui use-case. Are metode care corespund la o acțiune utilizator

Task: create service

```
def testCreateStudent():
    """
        Test store student
    """
    rep = InMemoryRepository()
    val = StudentValidator()
    srv = StudentService(rep, val)
    st = srv.createStudent("1", "Ion", "Adr")
    assert st.getId() == "1"
    assert st.getName() == "Ion"
    try:
        st = srv.createStudent("1", "Vasile", "Adr")
        assert False
    except ValueError:
        pass
    try:
        st = srv.createStudent("1", "", "")
        assert False
    except ValueError:
        pass
```

```
class StudentService:
    """
        Use case coordinator for CRUD Operations on student
    """
    def __init__(self, rep, validator):
        self.rep = rep
        self.validator = validator

    def createStudent(self, id, name, adr):
        """
            store a student
            id, name, address of the student as strings
            return the Student
            raise ValueError if a student with this id already exists
            raise ValueError if the student is invalid
        """
        st = Student(id, name, adr)
        if (self.validator != None):
            self.validator.validate(st)
        self.rep.store(st)
        return st
```

Application coordinator

Dependency injection (DI) este un principiu de proiectare pentru sisteme orientat obiect care are ca scop reducerea cuplării între componentele sistemului.

De multe ori un obiect folosește (deinde de) rezultatele produse de alte obiecte, alte părți ale sistemului.

Folosind **DI**, obiectul nu are nevoie să cunoască modul în care alte părți ale sistemului sunt implementate/create. Aceste dependențe sunt oferite (sunt injectate), împreună cu un contract (specificații) care descriu comportamentul componentei

```
#create validator
validator = StudentValidator()
#create repository
rep = InMemoryRepository(None)
#create console provide(inject) a validator and a repository
srv = StudentService(rep, validator)
#create console provide service
ui = Console(srv)
ui.showUI()
```

Review aplicația student manager – de revăzut şabloanele ce apar

Curs 6: Principii de proiectare

- Diagrame UML
- řabloane GRASP
- Arhitectură stratificată:
 - ui – service – domain - repository

Curs 7 – Principii de proiectare

- Entăti, ValueObject, Aggregate
- Fișiere in Python
- Moștenire – refolosire de cod

Curs 7 – Principii de proiectare

- Entăți, ValueObject, Aggregate
- Fișiere in Python
- Asocieri, Obiecte DTO

Curs 6: Principii de proiectare

- Diagrame UML
- řabloane GRASP
- Arhitectură stratificată:
 - ui – service – domain - repository

Recapitulare

Concept	Principii	Python
Clase/Obiecte	Încapsulare Ascunderea reprezentării Abstractizare (TAD)	class NumeCl: def __init__(self): self.__numeCamp = 3
GRASP	Principii: cum gândim / proiectam / implementam, ce întrebări punem în timp ce dezvoltam aplicația High Cohesion – fac metode/clase/module cu o singura responsabilitate Low coupling – reduc dependențele între metode/clase/module/pachete Information Expert – cum decid unde scriu codul pentru o funcționalitate GRASP Controller – creez o clasa care are metode pentru fiecare acțiune utilizator Protect Variation – dacă știu/mă aștept să se modifice/să existe mai multe variante => creez o clasa care conține funcționalitatea Creator – cum decid cine creează obiecte Pure Fabrication – Repository – creez o clasa care reprezintă un depozit de obiecte	
Layered	Arhitectura stratificată – GRASP High Cohesion, Low coupling UI – interfață utilizator Service – servicii oferite de aplicație, conține logica aplicației – GRASP Controller Domain – entități din domeniul problemei Validatori – Fa o clasa, folosește excepții pentru a semnala erori, GRASP Protect Variation Repository – Persistență, cod fișiere - GRASP Pure Fabrication	Layere – organizăm pe pachete Python Clase – cu responsabilități bine definite
Diagrame UML de clase	Reprezentare grafică pentru structura applicației	

Arhitectură stratificată (Layered architecture)

Layer (strat) este un mecanism de structurare logică a elementelor ce compun un sistem software

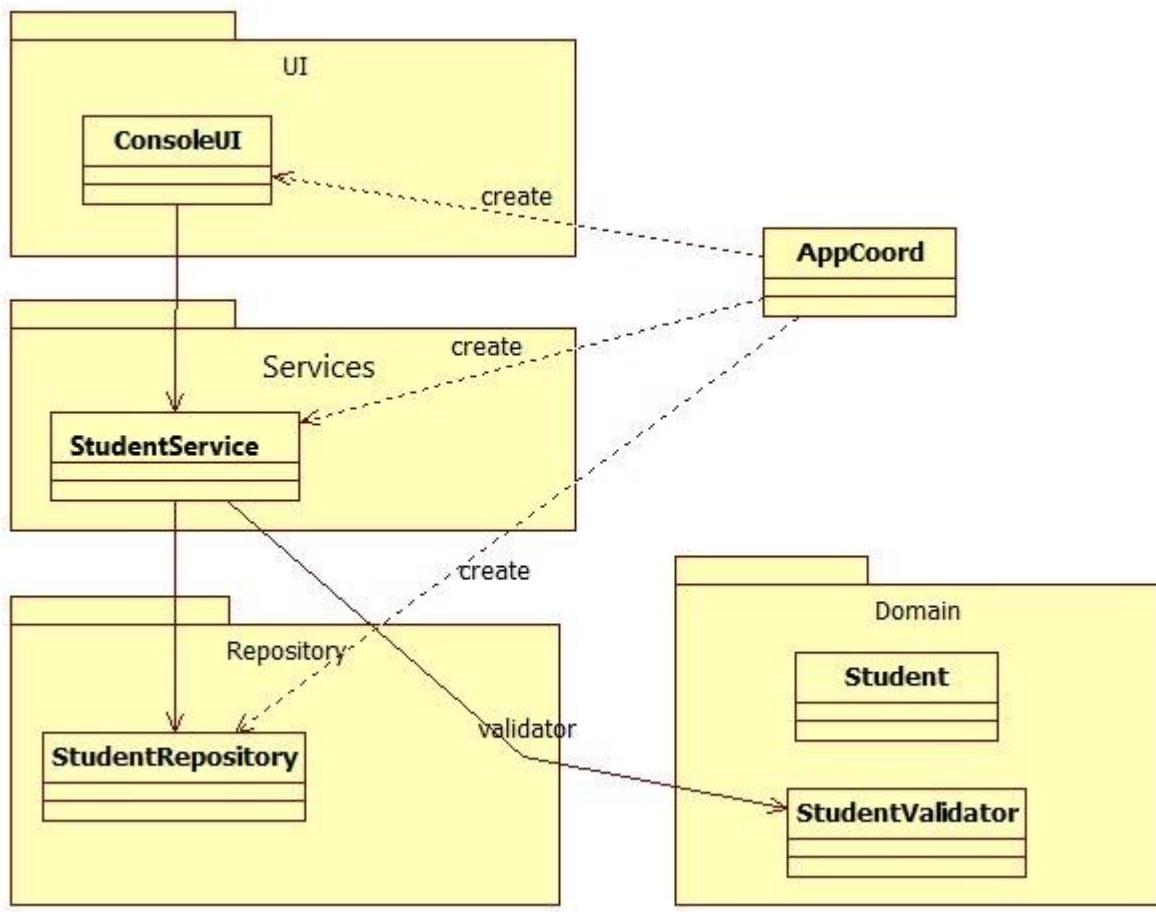
Într-o arhitectură multi-strat, straturile sunt folosite pentru a aloca responsabilități în aplicație.

Layer este un grup de clase (sau module) care au același set de dependențe cu alte module și se pot refolosi în circumstanțe similare.

- User Interface Layer (View Layer, UI layer sau Presentation layer)
- Service Layer (Application Layer sau **GRASP** Controller Layer)
- Domain layer (Business Layer, Business logic Layer sau Model Layer)
- **Infrastructure Layer** (acces la date – modalități de persistență, logging, network I/O ex. Trimitere de email, sau alte servicii tehnice)

Aplicația StudentCRUD

Review aplicație



Entități

Entitate (Entity) este un obiect care este definit de identitatea lui (se identifică cu exact un obiect din lumea reală).

Principala caracteristică a acestor obiecte nu este valoarea atributelor, este faptul ca pe întreg existența lor (in memorie, scris in fișier, încărcat, etc) se menține identitatea și trebuie asigurat consistența (sa nu existe mai multe entități care descriu același obiect).

Pentru astfel de obiecte este foarte important sa se definească ce înseamnă a fi egale.

```
def testIdentity():
    #attributes may change
    st = Student("1", "Ion", "Adr")
    st2 = Student("1", "Ion", "Adr2")
    assert st==st2

    #is defined by its identity
    st = Student("1", "Popescu", "Adr")
    st2 = Student("2", "Popescu", "Adr2")
    assert st!=st2

class Student:
    def __init__(self, id, name, adr):
        """
            Create a new student
            id, name, address String
        """
        self.__id = id
        self.__name = name
        self.__adr = adr

    def __eq__(self, ot):
        """
            Define equal for students
            ot - student
            return True if ot and the current instance represent the same student
        """
        return self.__id==ot.__id
```

Atributele entității se pot schimba dar identitatea rămâne același (pe întreg existența lui obiectul reprezintă același obiect din lumea reală)

O identitate greșită conduce la date invalide (data corruption) și la imposibilitatea de a implementa corect anumite operații.

Obiecte valoare (Value Objects)

Obiecte valoare: obiecte ce descriu caracteristicile unui obiect din lumea reală, conceptual ele nu au identitate.

Reprezintă aspecte descriptive din domeniu. Când ne preocupă doar atributele unui obiect (nu și identitatea) clasificăm aceste obiecte ca fiind Obiecte Valoare (Value Object)

```
def testCreateStudent():
    """
        Testing student creation
        Feature 1 - add a student
        Task 1 - Create student
    """
    st = Student("1", "Ion", Address("Adr", 1, "Cluj"))
    assert st.getId() == "1"
    assert st.getName() == "Ion"
    assert st.getAddr().getStreet() == "Adr"

    st = Student("2", "Ion2", Address("Adr2", 1, "Cluj"))
    assert st.getId() == "2"
    assert st.getName() == "Ion2"
    assert st.getAddr().getStreet() == "Adr2"
    assert st.getAddr().getCity() == "Cluj"

class Address:
    """
        Represent an address
    """
    def __init__(self, street, nr, city):
        self.__street = street
        self.__nr = nr
        self.__city = city

    def getStreet(self):
        return self.__street

    def getNr(self):
        return self.__nr

    def getCity(self):
        return self.__city

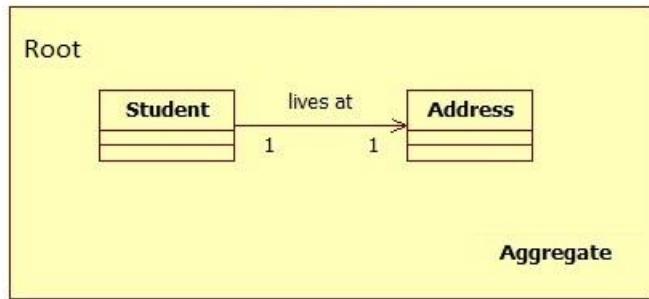
class Student:
    """
        Represent a student
    """
    def __init__(self, id, name, adr):
        self.__id = id
        self.__name = name
        self.__adr = adr

    def getId(self):
        """
            Getter method for id
        """
        return self.__id
```

Agregate și Repository

Grupați entități și obiecte valoare în aggregate. Alegeti o entitate rădăcină (root) care controlează accesul la toate elementele din agregat.

Obiectele din afara aggregatului ar trebui să aibă referință doar la entitatea principală.



Repository – creează iluzia unei colecții de obiecte de același tip. Creați Repository doar pentru entitatea principală din agregat

Doar StudentRepository (nu și AddressRepository)

Fișiere text în Python

Funcția built in: **open()** returnează un obiect reprezentând fișierul

Cel mai frecvent se folosește apelul cu două argumente: **open(filename,mode)**.

Filename – un string, reprezintă calea către fișier(absolut sau relativ)

Mode:

"**r**" – open for read

"**w**" – open for write (overwrites the existing content)

"**a**" – open for append

Metode:

write(str) – scrie string în fișier

readline() - citire linie cu line, returnează string

read() - citește tot fișierul, returnează string

close() - închide fișier, eliberează resursele ocupate

Excepții:

IOError – aruncă această excepție dacă apare o eroare de intrare/ieșire (no file, no disk space, etc)

Exemple Python cu fișiere text

```
#open file for write (overwrite if exists, create if not)
f = open("test.txt", "w")
f.write("Test data\n")
f.close()
```

```
#open file for write (append if exist, create if not)
f = open("test.txt", "a")
f.write("Test data line 2\n")
f.close()
```

```
#open for read
f = open("test.txt", "r")
#read a line from the file
line = f.readline()
print line
f.close()
```

```
#open for read
f = open("test.txt", "r")
#read a line from the file
line = f.readline().strip()
while line!="":
    print line
    line = f.readline().strip()
f.close()
```

```
#open for read
f = open("test.txt", "r")
#read the entire content from the file
line = f.read()
print line
f.close()
```

```
#use a for loop
f = open("etc/test.txt")
for line in f:
    print line
f.close()
```

Repository cu fișiere

```
class StudentFileRepository:
    """
        Store/retrieve students from file
    """
    def __loadFromFile(self):
        """
            Load students from file
        """
        try:
            f = open(self.__ fName, "r")
        except IOError:
            #file not exist
            return []
        line = f.readline().strip()
        rez = []
        while line!="":
            attrs = line.split(";")
            st = Student(attrs[0], attrs[1], Address(attrs[2], attrs[3], attrs[4]))
            rez.append(st)
            line = f.readline().strip()
        f.close()
        return rez

    def store(self, st):
        """
            Store the student to the file.Overwrite store
            st - student
            Post: student is stored to the file
            raise DuplicatedIdException for duplicated id
        """
        allS = self.__loadFromFile()
        if st in allS:
            raise DuplicatedIDException()
        allS.append(st)
        self.__storeToFile(allS)

    def __storeToFile(self,sts):
        """
            Store all the students in to the file
            raise CorruptedFileException if we can not store to the file
        """
        #open file (rewrite file)
        f = open(self.__ fName, "w")
        for st in sts:
            strf = st.getId()+" ; "+st.getName()+" ; "
            strf = strf +
st.getAdr().getStreet()+" ; "+str(st.getAdr().getNr())+" ; "+st.getAdr().getCity()
            strf = strf+"\n"
            f.write(strf)
        f.close()
```

Gestiunea resurselor în prezență a excepțiilor

Orice fișier pe care deschidem cu *open* ar trebui să închidem folosind metoda *close()*
Ciclu de viață pentru o resursă: Crearea/Achiziție -> Folosire Resursă ->
eliberație/distrugere

Problema:	Solutie
<pre>def applyToFile(fileName): """ process file line by line """ fh = open(fileName) for line in fh: processLine(line) fh.close()</pre>	<pre>def applyToFile(fileName): """ process file line by line """ fh = open(fileName) try: for line in fh: processLine(line) finally: fh.close()</pre>
Aparent codul de mai sus gestionează corect resursa (fișier) Ce se întâmplă dacă funcția <i>processLine</i> arunca excepție? Fișierul rămâne deschis	Rezolvă problema: se închide fișierul chiar dacă apare o excepție în metoda <i>processLine</i> Codul pare complex, clauza <i>try/finally</i> face codul mai greu de urmărit

Valabil și în cazul altor resurse pe care trebuie să le gestionam.

Instrucțiunea **with** rezolvă problema mai elegant:

```
def applyToFile(fileName):
    """
        process file line by line
    """

    with open(fileName) as fh:
        for line in fh:
            processLine(line)
```

Codul de mai sus este echivalent cu codul care folosește *try/finally*
Fișierul se închide (se apelează *fh.close()*) și la execuție normală și dacă apare o excepție în corpul instrucțiunii **with** fișierul se va închide.

În cazul în care apare o excepție în corpul instrucțiunii, excepția este aruncată (nu dispără excepția, doar se asigură că fișierul (resursa) se închide/eliberează). Pentru mai multe detalii vezi PEP 343 (<https://www.python.org/dev/peps/pep-0343/>)

Asocieri între obiecte din domeniu

În lumea reală, conceptual sunt multe relații de tip many-to-many dar modelarea acestor relații în aplicație nu este întotdeauna fezabilă.

Când modelăm obiecte din lumea reală în aplicațiile noastre, asocierile complică implementarea și întreținerea aplicației.

- Asocierile bidirectionale de exemplu presupun ca fiecare obiect din asociere se poate folosi/înțelege/refolosi doar împreună

Este important să simplificăm aceste relații cât de mult posibil, prin:

- Impunerea unei direcții (transformare din bi-directional în unidirectional)
- Reducerea multiplicării
- Eliminarea asocierilor ne-esențiale

Scopul este să modelăm lumea reală cât mai fidel dar în același timp să simplificăm modelul pentru a nu complica implementare.

Asocieri

Exemplu Catalog



```
gr = ctr.assign("1", "FP", 10)
assert gr.getDiscipline() == "FP"
assert gr.getGrade() == 10
assert gr.getStudent().getId() == "1"
assert gr.getStudent().getName() == "Ion"
```

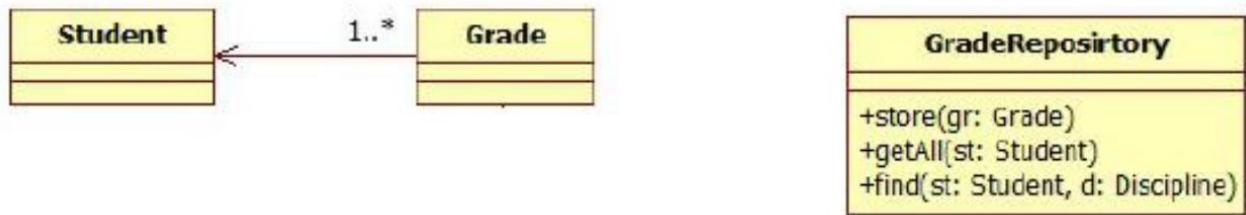
```
st = Student("1", "Ion",
             Address("Adr", 1, "Cluj"))

rep = GradeRepository()
grades = rep.getAll(st)
assert grades[0].getStudent() == st
assert grades[0].getGrade() == 10
```

Ascunderea detaliilor legate de persistență

Repository trebuie să ofere iluzia că obiectele sunt în memorie astfel codul client poate ignora detaliile de implementare.

În cazul în care repository salvează datele se în fișier, trebuie să avem în vedere anumite aspecte.



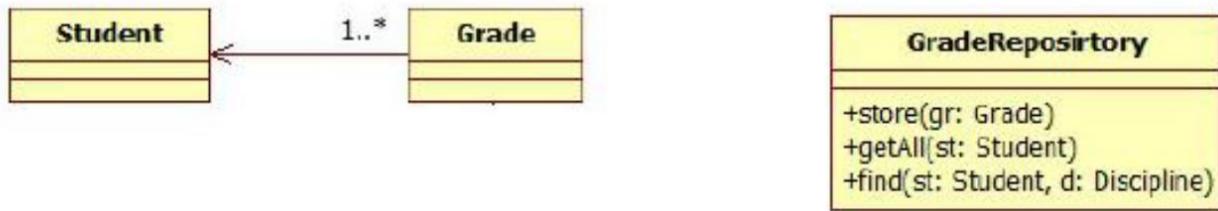
În exemplul de mai sus GradeRepository salvează doar id-ul studentului (nu toate câmpurile studentului) astfel nu se poate implementa o funcție getAll în care se returnează toate notele pentru toți studenții. Se poate în schimb oferi metoda getAll(st) care returnează toate notele pentru un student dat

```
def store(self, gr):
    """
        Store a grade
        post: grade is in the repository
        raise GradeAlreadyAssigned exception if we already have a grade
            for the student at the given discipline
        raise RepositoryException if there is an IO error when writing to
            the file
    """
    if self.find(gr.getStudent(), gr.getDiscipline()) !=None:
        raise GradeAlreadyAssigned()

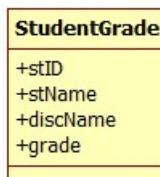
    #open the file for append
    try:
        f = open(self.__fname, "a")
        grStr = gr.getStudent().getId() + "," + gr.getDiscipline()
        grStr += str(gr.getGrade()) + "\n"
        f.write(grStr)
        f.close()
    except IOError:
        raise RepositoryException("Unable to write a grade to the file")
```

Obiecte de transfer (DTO - Data transfer objects)

Funcționalitate: Primi 5 studenți la o disciplină. Prezentați în format tabelar : nume student, nota la disciplina dată



Avem nevoie de obiecte speciale (obiecte de transfer) pentru acest caz de utilizare. Funcțiile din repository nu ajung pentru a implementa (nu avem getAll()). Se creează o nouă clasă care conține exact informațiile de care e nevoie.



În repository:

```
def getAllForDisc(self, disc):
    """
        Return all the grades for all the students from all disciplines
        disc - string, the discipline
        return list of StudentGrade's
    """
    try:
        f = open(self.__fname, "r")
    except IOError:
        #the file is not created yet
        return None
    try:
        rez = [] #StudentGrade instances
        line = f.readline().strip()
        while line!="":
            attrs = line.split(",")
            #if this line refers to the requested student
            if attrs[1]==disc:
                gr = StudentGrade(attrs[0], attrs[1], float(attrs[2]))
                rez.append(gr)
            line = f.readline().strip()
        f.close()
        return rez
    except IOError:
        raise RepositoryException("Unable to read grades from the file")
```

DTO – Data transfer object

In controller:

```
def getTop5(self,disc):
    """
        Get the best 5 students at a given discipline
        disc - string, discipline
        return list of StudentGrade ordered descending on
the grade
    """
    sds = self.__grRep.getAllForDisc(disc)
    #order based on the grade
    sortedsds = sorted(sds, key=lambda studentGrade:
studentGrade.getGrade(),reverse=True)
    #retain only the first 5
    sortedsds = sortedsds[:5]
    #obtain the student names
    for sd in sortedsds:
        st = self.__stRep.find(sd.getStudentID())
        sd.setStudentName(st.getName())
    return sortedsds
```

Dynamic Typing

Verificarea tipului se efectuează în timpul execuției (runtime) – nu în timpul compilării (compile-time).

În general în limbajele cu dynamic typing valorile au tip, dar variabilele nu. Variabila poate referi o valoare de orice tip

Duck Typing

Duck typing este un stil de dynamic typing în care metodele și câmpurile obiectelor determină semantica validă, nu relația de moștenire de la o clasă anume sau implementarea unei interfețe.

Interfața publică este dată de multimea metodelor și câmpurilor accesibile din exterior. Două clase pot avea același interfață publică chiar dacă nu există o relație de moștenire de la o clasă de bază comună

Duck test: When I see a bird that walks like a duck and swims like a duck and quacks like a duck, I call that bird a duck

<pre>class Student: def __init__(self, id, name): self.__name = name self.__id = id def getId(self): return self.__id def getName(self): return self.__name</pre>	<pre>class Professor: def __init__(self, id, name, course): self.__id = id self.__name = name self.__course = course def getId(self): return self.__id def getName(self): return self.__name def getCourse(self): return self.__course</pre>
<pre>l = [Student(1, "Ion"), Professor("1", "Popescu", "FP"), Student(31, "Ion2"), Student(11, "Ion3"), Professor("2", "Popescu3", "asd")] for el in l: print el.getName() + " id " + str(el.getId()) def myPrint(st): print el.getName(), " id ", el.getId() for el in l: myPrint(el)</pre>	

Duck typing – Repository

Fiindcă interfața publică a clasei:

- GradeRepository și GradeFileRepository
- StudentRepository și StudentFileRepository

sunt identice controllerul funcționează cu oricare obiect, fără modificări.

```
#create a validator
val = StudentValidator()
#create repository
repo = StudentFileRepository("students.txt")
#create controller and inject dependencies
srv = StudentService(val, repo)
#create Grade controller
gradeRepo = GradeFileRepository("grades.txt")
srvgr = GradingService(gradeRepo, GradeValidator(), repo)
#create console ui and provide (inject) the controller
ui = ConsoleUI(srv, srvgr)
ui.startUI()

#create a validator
val = StudentValidator()
#create repository
repo = StudentRepository()
#create controller and inject dependencies
srv = StudentService(val, repo)
#create Grade controller
gradeRepo = GradeRepository()
srvgr = GradingService(gradeRepo, GradeValidator(), repo)
#create console ui and provide (inject) the controller
ui = ConsoleUI(srv, srvgr)
ui.startUI()
```

Curs 7 – Principii de proiectare

- Entăți, ValueObject, Aggregate
- Fișiere in python
- Asocieri, Obiecte de transfer DTO

Curs 8 – Testarea programelor

- Moștenire, UML
- Unit teste in python
- Depanarea aplicațiilor python

Curs 8 – Testarea programelor

- Moștenire, UML
- Unit teste in Python
- Depanarea/inspectarea aplicațiilor

Curs 7 – Principii de proiectare

- Entăți, ValueObject, Aggregate
- Fișiere in python
- Asocieri, Obiecte de transfer DTO

Moștenire

Moștenirea permite definirea de clase noi (clase derivate) reutilizând clase existente (clasa de bază). Clasa nou creată moștenește comportamentul (metode) și caracteristicile (variabile membre, starea) de la clasa de bază.

Dacă A și B sunt două clase unde B moștenește de la clasa A (B este derivat din clasa A sau clasa B este o specializare a clasei A) atunci:

- clasa B are toate metodele și variabilele membre din clasa A
- clasa B poate redefini metode din clasa A
- clasa B poate adăuga noi membrii (variabile, metode) pe lângă cele moștenite de la clasa A.

Reutilizare de cod

Una din motivele pentru care folosim moștenire este reutilizarea codului existent într-o clasă (moștenire de implementare).

Comportamentul unei clase de baze se poate moșteni de clasele derivate.

Clasa derivată poate:

- poate lăsa metoda nemodificată
- apela metoda din clasa de bază
- poate modifica (suprascrie) o metodă.

Moștenire în Python

Sintaxă:

```
class DerivedClassName(BaseClassName):
```

Clasa derivată moștenește:

- câmpuri
- metode

Dacă accesăm un membru (câmp, metodă) : se caută în clasa curentă, dacă nu se găsește atunci căutarea continuă în clasa de bază

```
class B(A):  
    """  
        This class extends A  
        A is the base class,  
        B is the derived class  
        B is inheriting everything from class A  
    """  
  
    def __init__(self):  
        #initialise the base class  
        A.__init__(self)  
        print "Initialise B"  
  
    def g(self):  
        """  
            Overwrite method g from A  
        """  
  
        #we may invoke the function from the  
        #base class  
        A.f(self)  
        print "in method g from B"
```

```
class A:  
    def __init__(self):  
        print ("Initialise A")  
  
    def f(self):  
        print("in method f from A")  
  
    def g(self):  
        print("in method g from A")
```

```
b = B()  
#f is inherited from A  
b.f()  
b.g()
```

Clasele Derivate pot suprascrie metodele clasei de bază.

Suprascrierea poate înlocui cu totul metoda din clasa de bază sau poate extinde funcționalitatea (se execută și metoda din clasa de bază dar se mai adaugă cod)

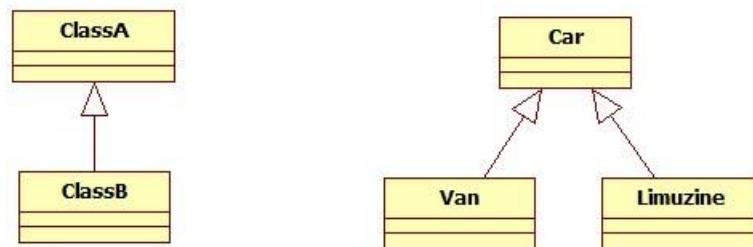
O metodă simplă să apelăm o metodă în clasa de bază:

BaseClassName.methodname (self,arguments)

Diagrame UML – Generalizare (moștenire)

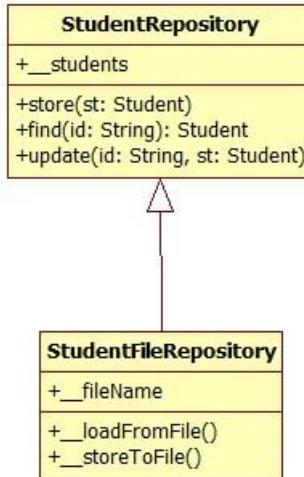
Relația de generalizare ("is a") indică faptul că o clasă (clasa derivată) este o specializare a altei clase (clasa de bază). Clasa de bază este generalizarea clasei deriveate.

Orice instanță a clasei deriveate este și o instanță a clasei de bază.



Repository cu Fișiere

```
class StudentFileRepository(StudentRepository):
    """
        Repository for students (stored in a file)
    """
    pass
```



```
class StudentFileRepository(StudentRepository):
    """
        Store/retrieve students from file
    """

    def __init__(self, fName):
        #properly initialise the base class
        StudentRepository.__init__(self)
        self.__fName = fName
        #load student from the file
        self.__loadFromFile()

    def __loadFromFile(self):
        """
            Load students from file
            raise ValueError if there is an error when reading from the file
        """
        try:
            f = open(self.__fName, "r")
        except IOError:
            #file not exist
            return
        line = f.readline().strip()
        while line!="":
            attrs = line.split(",")
            st = Student(attrs[0], attrs[1], Address(attrs[2], attrs[3], attrs[4]))
            StudentRepository.store(self, st)
            line = f.readline().strip()
        f.close()
```

Suprascriere metode

```
def testStore():
    fileName = "teststudent.txt"
    repo = StudentFileRepository(fileName)
    repo.removeAll()

    st = Student("1", "Ion", Address("str", 3, "Cluj"))
    repo.store(st)
    assert repo.size() == 1
    assert repo.find("1") == st
    #verify if the student is stored in the file
    repo2 = StudentFileRepository(fileName)
    assert repo2.size() == 1
    assert repo2.find("1") == st

def store(self, st):
    """
        Store the student to the file. Overwrite store
        st - student
        Post: student is stored to the file
        raise DuplicatedIdException for duplicated id
    """
    StudentRepository.store(self, st)
    self.__storeToFile()

def __storeToFile(self):
    """
        Store all the students in to the file
        raise CorruptedFileException if we can not store to the file
    """
    f = open(self.__ fName, "w")
    sts = StudentRepository.getAll(self)
    for st in sts:
        strf = st.getId() + ";" + st.getName() + ";" +
        strf = strf +
    st.getAddr().getStreet() + ";" + str(st.getAddr().getNr()) +
    ";" + st.getAddr().getCity()
        strf = strf + "\n"
        f.write(strf)
    f.close()
```

Exceptii

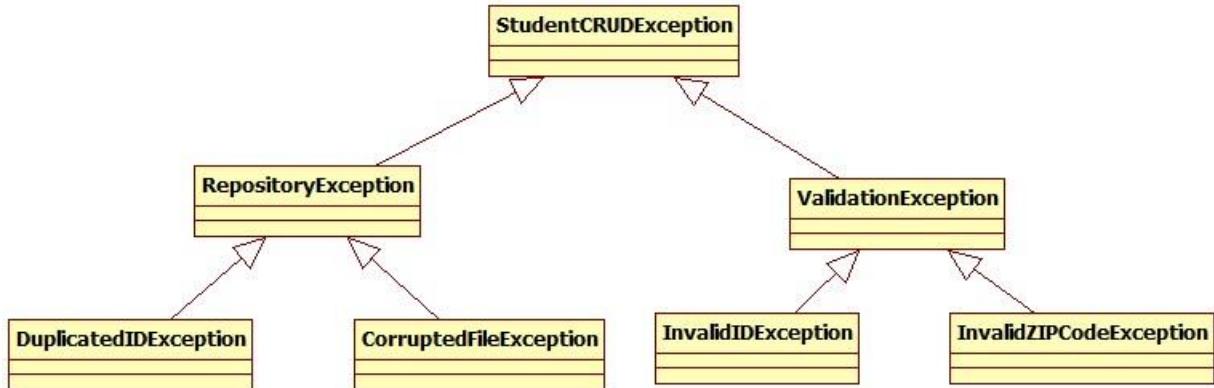
```
def __createdStudent(self):
    """
        Read a student and store in the application
    """
    id = input("Student id:").strip()
    name = input("Student name:").strip()
    street = input("Address - street:").strip()
    nr = input("Address - number:").strip()
    city = input("Address - city:").strip()
    try:
        self.__ctr.create(id, name, street, nr, city)
    except ValueError as msg:
        print (msg)

def __createdStudent(self):
    """
        Read a student and store in the application
    """
    id = input("Student id:").strip()
    name = input("Student name:").strip()
    street = input("Address - street:").strip()
    nr = input("Address - number:").strip()
    city = input("Address - city:").strip()
    try:
        self.__ctr.create(id, name, street, nr, city)
    except ValidationException as ex:
        print (ex)
    except DuplicatedIDException as ex:
        print (ex)

class ValidationException(Exception):
    def __init__(self, msgs):
        """
            Initialise
            msg is a list of strings (errors)
        """
        self.__msgs = msgs
    def getMsgs(self):
        return self.__msgs

    def __str__(self):
        return str(self.__msgs)
```

Ierarhie de exceptii



```

class StudentCRUDEException(Exception):
    pass

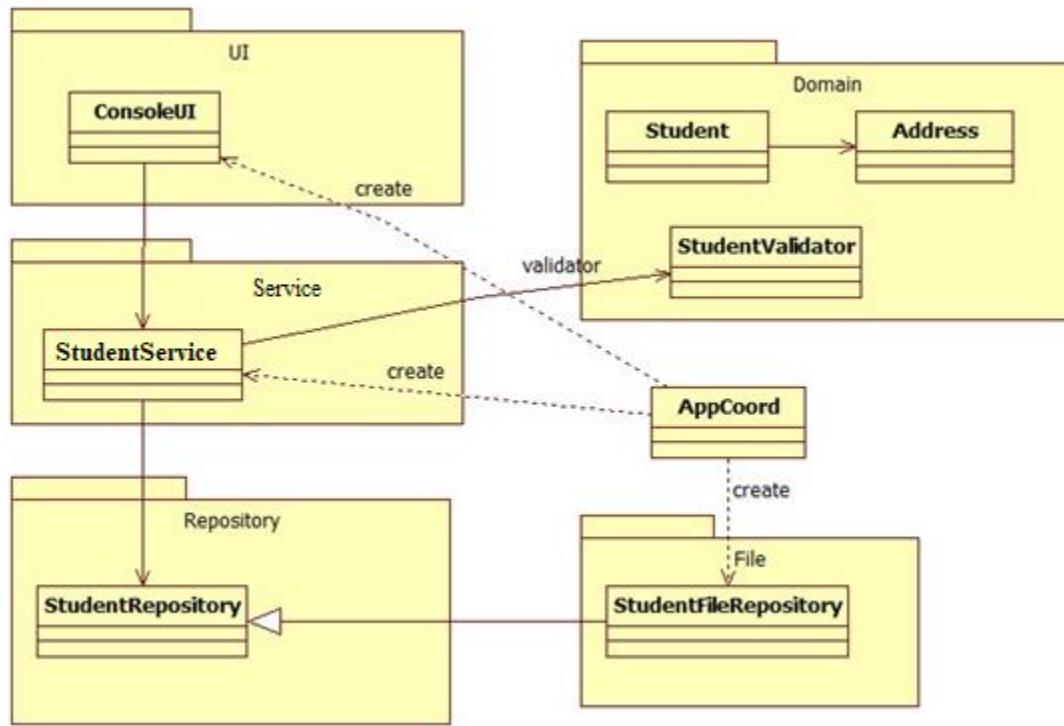
class ValidationException(StudentCRUDEException):
    def __init__(self, msgs):
        """
        msg is a list of strings (errors)
        """
        self.__msgs = msgs
    def getMsgs(self):
        return self.__msgs
    def __str__(self):
        return str(self.__msgs)

class RepositoryException(StudentCRUDEException):
    """
    Base class for the exceptions in the repository
    """
    def __init__(self, msg):
        self.__msg = msg
    def getMsg(self):
        return self.__msg
    def __str__(self):
        return self.__msg

class DuplicatedIDException(RepositoryException):
    def __init__(self):
        RepositoryException.__init__(self, "Duplicated ID")

def __createStudent(self):
    """
        Read a student and store in the application
    """
    id = input("Student id:").strip()
    name = input("Student name:").strip()
    street = input("Address - street:").strip()
    nr = input("Address - number:").strip()
    city = input("Address - city:").strip()
    try:
        self.__ctr.create(id, name, street, nr, city)
    except StudentCRUDEException as ex:
        print (ex)
  
```

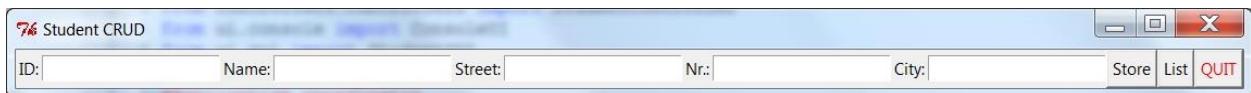
Layered arhitecture – Structură proiect



Layered architecture – Exemplu GUI / Web

Tkinter este un toolkit GUI pentru Python (este disponibil pe majoritatea platformelor Unix , pe Windows și Mac).

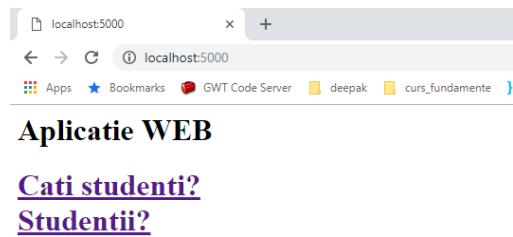
Review - aplicația StudentCRUD cu GUI



Tkinter (sau orice alt GUI) nu se cere la examen

Flask – framework pentru dezvoltare aplicații web.

Instalare: pip install Flask



La examen nu se cer aplicații web.

Testarea programelor

Testarea este observarea comportamentului unui program în multiple execuții.

Se execută programul pentru ceva date de intrare și se verifică dacă rezultate sunt corecte în raport cu intrările.

Testarea nu demonstrează corectitudinea unui program (doar oferă o anumită siguranță , confidență). În general prin testare putem demonstra că un program nu este corect, găsind un exemplu de intrări pentru care rezultatele sunt greșite.

Testarea nu poate identifica toate erorile din program.

Metode de testare

Testare exhaustivă

Verificarea programului pentru toate posibilele intrări.

Imposibil de aplicat în practică, avem nevoie de un număr finit de cazuri de testare.

Black box testing (metoda cutiei negre)

Datele de test se selectează analizând specificațiile (nu ne uităm la implementare).

Se verifică dacă programul respectă specificațiile.

Se aleg cazuri de testare pentru: valori obișnuite, valori limite, condiții de eroare.

White box testing (metoda cutiei transparente)

Datele de test se aleg analizând codul sursă. Alegem datele astfel încât se acoperă toate ramurile de execuție (în urma executării testelor, fiecare instrucțiune din program este executat măcar odată)

White box vs Black Box testing

```
def isPrime(nr):
    """
        Verify if a number is prime
        return True if nr is prime False if not
        raise ValueError if nr<=0
    """
    if nr<=0:
        raise ValueError("nr need to be positive")
    if nr==1:#1 is not a prime number
        return False
    if nr<=3:
        return True
    for i in range(2,nr):
        if nr%i==0:
            return False
    return True
```

Black Box

- test case pentru prim/compus
- test case pentru 0
- test case pentru numere negative

White Box (cover all the paths)

- test case pt. 0
- test case pt. negative
- test case pt. 1
- test case pt. 3
- test case pt. prime (fără divizor)
- test case pt. neprime

```
def blackBoxPrimeTest():
    assert (isPrime(5)==True)
    assert (isPrime(9)==False)
    try:
        isPrime(-2)
        assert False
    except ValueError:
        assert True
    try:
        isPrime(0)
        assert False
    except ValueError:
        assert True
```

```
def whiteBoxPrimeTest():
    assert (isPrime(1)==False)
    assert (isPrime(3)==True)
    assert (isPrime(11)==True)
    assert (isPrime(9)==True)
    try:
        isPrime(-2)
        assert False
    except ValueError:
        assert True
    try:
        isPrime(0)
        assert False
    except ValueError:
        assert True
```

Nivele de testare

Testele se pot categoriza în funcție de momentul în care se creează (în cadrul procesului de dezvoltare) sau în funcție de specificitatea testelor.

Unit testing

Se referă la testarea unei funcționalități izolate, în general se referă la testarea la nivel de metode. Se testează funcțiile sau părți ale programului, independent de restul aplicației

Integration testing

Consideră întreaga aplicație ca un întreg. După ce toate funcțiile au fost testate este nevoie de testarea comportamentului general al programului.

Testare automată (Automated testing)

Testare automată – presupune scrierea de programe care realizează testarea (în loc să se efectueze manual).

Practic se scrie cod care compara rezultatele efective pentru un set de intrări cu rezultatele așteptate.

TDD:

Pașii TDD:

- teste automate
- scrierea specificațiilor (inv, pre/post, excepții)
- implementarea codului

PyUnit - bibliotecă Python pentru unit testing

modulul **unittest** oferă:

- teste automate
- modalitate uniformă de pregătire/curățare (setup/shutdown) necesare pentru teste
 - fixture
- agregarea testelor
 - test suite
- independența testelor față de modalitatea de raportare

```
import unittest
class TestCaseStudentController(unittest.TestCase):
    def setUp(self):
        #code executed before every testMethod
        val=StudentValidator()
        self.ctr=StudentController(val, StudentRepository())
        st = self.ctr.create("1", "Ion", "Adr", 1, "Cluj")

    def tearDown(self):
        #cleanup code executed after every testMethod

    def testCreate(self):
        self.assertTrue(self.ctr.getNrStudents()==1)
        #test for an invalid student
        self.assertRaisess(ValidationEx, self.ctr.create, "1", "", "", 1, "Cj")

        #test for duplicated id
        self.assertRaisess(DuplicatedIDException, self.ctr.create, "1", "I",
                           "A", 1, "j")

    def testRemove(self):
        #test for an invalid id
        self.assertRaisess(ValueError, self.ctr.remove, "2")

        self.assertTrue(self.ctr.getNrStudents()==1)

        st = self.ctr.remove("1")
        self.assertTrue(self.ctr.getNrStudents()==0)
        self.assertEquals(st.getId(), "1")
        self.assertTrue(st.getName()=="Ion")
        self.assertTrue(st.getAdr().getStreet()=="Adr")

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

Depanare (Debugging)

Depanarea este activitatea prin care reparăm erorile găsite în urma testării.

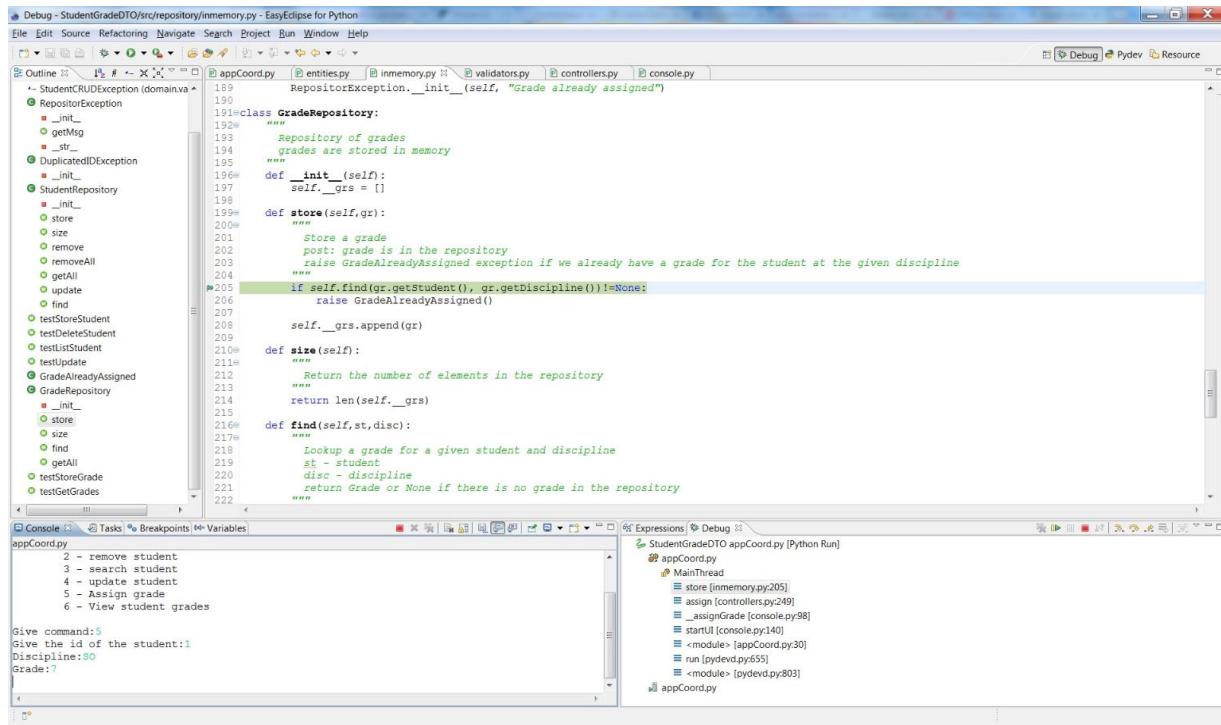
Dacă testarea indică prezența unei erori atunci prin depanare încearcăm să identificăm cauza erorii, modalități de rezolvare. Scopul este să eliminăm eroarea.

Se pate realiză folosind:

- instrucțiuni print
 - instrumente specializate oferite de IDE

Depanarea este o activitate neplăcută, pe cât posibil, trebuie evitată.

Perspectiva Eclipse pentru depanare



Debug view

- prezintă starea curentă de execuție (stack trace)
 - execuție pas cu pas, resume/pause

Variables view

- inspectarea variabilelor

Inspectarea programelor

Any fool can write code that a computer can understand. Good programmers write code that humans can understand

Prin stilul de programare înțelegem toate activitățile legate de scrierea de programe și modalitățile prin care obținem cod: ușor de citit, ușor de înțeles, ușor de întreținut.

Stil de programare

Principalul atribut al codului sursă este considerat ușurința de a citi (readability).

Un program, ca și orice publicație, este un text care trebuie citit și înțeles cu ușurință de orice programator.

Elementele stilului de programare sunt:

- comentarii
- formatarea textului (indentare, white spaces)
- specificații
- denumiri sugestive (pentru clase, funcții, variabile) din program
 - denumiri sugestive
 - folosirea convențiilor de nume

Convenții de nume (naming conventions):

- clase: Student, StudentRepository
- variabile: student, nrElem (nr_elem)
- funcții: getName, getAddress, storeStudent (get_name, get_address, store_student)
- constante: MAX

Este important să folosiți același reguli de denumire în toată aplicația

Curs 8 – Testarea programelor

- Moștenire, UML
- Unit teste in python
- Depanarea/Inspectarea aplicațiilor

Curs 9 – Complexitate

- Recursivitate
- Complexitate