

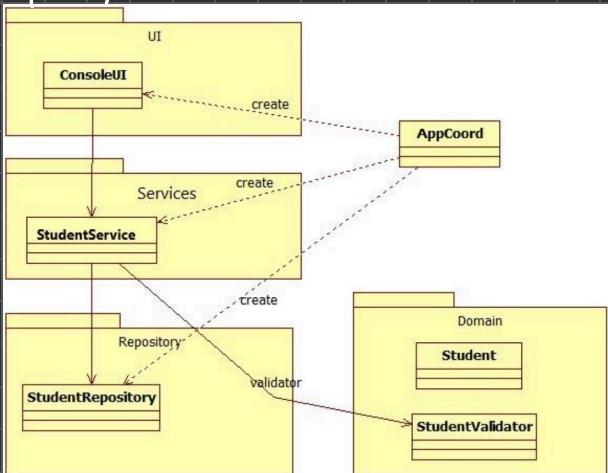
Fundamentele programării

Recapitulare

Concept	Principii	Python
Clase/Obiecte	Încapsulare Ascunderea reprezentării Abstractizare (TAD)	class NumereCl: def __init__(self): self.__numeCamp = 3
GRASP	Principii: cum gândim / proiectam / implementam, ce întrebări punem în timp ce dezvoltăm aplicația High Cohesion – fac metode/clase/module cu o singură responsabilitate Low coupling – reduc dependențele între metode/clase/module/pachete Information Expert – cum decid unde scriu codul pentru o funcționalitate GRASP Controller – creez o clasă care are metode pentru fiecare acțiune utilizator Protect Variation – dacă știu/mă aștepț să se modifice/să existe mai multe variante => creez o clasă care conține funcționalitatea Creator – cum decid cine creează obiecte Pure Fabrication – Repository – creez o clasă 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 Validator – Fa o clasă , folosește exceptii pentru a semnaliza erori, GRASP Protect Variation Repository – Persistență, cod fișiere – GRASP Pure Fabrication	Layered – organizam pe pachete Python Clase – cu responsabilități bine definite
Diagrame UML de clase	Reprezentare grafică pentru structura aplicației	

Într-o arhitectură multi-strat, straturile sunt folosite pentru a aloca responsabilități. Fiecare layer este un grup de clase care au același set de dependențe și se referă la circumstanțe similare.

Aplicația Student



Entitate

O entitate este un obiect care are definit prin identitatea sa, caracteristica principala a acestor obiecte este ca identitatea lor ramane constanta pe intreaga luna existenta. Este important ca identitatea sa fie consistenta si sa nu exista mai multe entitati care descriu acelasi obiect.

Pentru obiecte, egalitatea trebuie definita pe baza identitatii, nu pe atributelor

```
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 unei entitati pot sa se schimbe, dar identitatea ramane același mereu.

Obiecte valoare

Acestea sunt obiecte ce descriu caracteristici unui obiect din lumea reală, ele nu au identitate.

De obicei reprezintă aspecte descriptive din domeniu. Când ne preocupă doar atributeli unui obiect (nu și identitatea) clasificăm acești obiecte ca fiind Obiect Value.

```
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

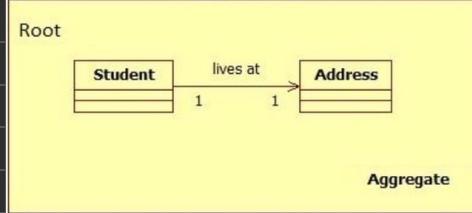
    def getCity(self):
        return self.__city

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

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

Aggregate

Entitățile și obiectele valoare se grupăază în agregat. Se alege o națădină (root) care va controla accesul la toate elementele din agregat.



Repository-ul crează iluzia unei colecții de obiecte de același tip. Se crează un repository doar pentru entitatea principală din agregat. (doar `StudentRepository` nu și `AddressRepository`)

Fisiere text în Python

Funcția built-in: `open()` returnează un obiect reprezentând fisierul. Funcția se folosește de obicei cu două argumente: `open(filename, mode)`. `filename`-ul este un string care reprezintă calea către fișier.

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 linie, returnează string

`read()` - citeste tot fișierul, returnează string

close() - închide fișier

Excepții: IOError - aruncă această excepție dacă aparu o eroare de intrare/iesire (no file, no disk space, etc)

Exemplu Python cu fișier 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 fișier

```

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.getAddr().getStreet()+" ; "+str(st.getAddr().getNr())+" ; "+st.getAddr().getCity()
            strf = strf+"\n"
        f.write(strf)
        f.close()

```

Gestarea fișierelor în prezență excepțiilor

Orică fișier pe care-l deschidem cu open trebuie închis folosind metoda close. Guare → Fălsoare → 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 *processLine* arunca excepție? Fișierul rămâne deschis

Rezolvă problema: se închide fișierul chiar dacă apare o excepție în metoda *processLine*

Codul pare complex, clauza try/finally face codul mai greu de urmărit

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)
```

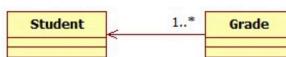
Codul de mai sus este echivalent cu codul care folosește `try/finally`. Fișierul se închide (se apelaază `fh.close()`) și la execuție normală și dacă apare o excepție în corpul instrucțiunii `with`.

Asocieri între obiecte din domeniu

În viața reală există relații complexe de tip many-to-many între obiecte. De ex: un student poate participa la mai multe cursuri, iar fiecare curs poate avea mai mulți studenți. Problema este că implementarea acestor relații este dificilă și complicată.

În loc să folosim relații bidirectionale se preferă relațiile unic方向ională pentru a simplifica implementarea și întreținerea.

Exemplu catalog:



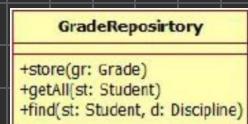
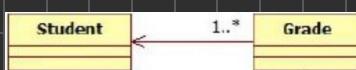
```
gr = ctr.assign("1", "FP", 10)
assert gr.getDiscipline() == "FP"
assert gr.getGrade() == 10
assert gr.getStudent().getId() == "1"
assert gr.getStudent().getName() == "Ion"
```

```
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-ul trebuie să ofere iluzia că obiectele sunt în memorie, chiar dacă datele sunt persistante pe un disc sau în alte mecanisme de stocare. Astfel codul nu trebuie să fie preocupat de modul de stocare efectiv a datelor.

În cazul în care repository-ul salvează datele în fisier, trebuie să avem în vedere urmările aspecte.



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 implementa în schimb o metodă care returneară

```

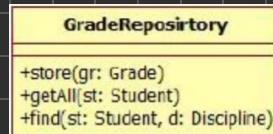
def store(self, gr):
    """
    Store a grade
    post: grade is in the repository
    raise GradeAlreadyAssigned exception if we already have a grade
        for the student at the given discipline
    raise RepositoryException if there is an IO error when writing to
        the file
    """
    if 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 +=grStr+", "+str(gr.getGrade())+"\n"
        f.write(grStr)
        f.close()
    except IOError:
        raise RepositoryException("Unable to write a grade to the file")
  
```

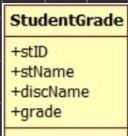
metoda Store salvează informația într-un fisier sub formă de text, inclusând doar ID-ul studentului, disciplina și nota

DTO - Obiecte de transfer

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



Aveam nevoie de obiecte speciale (obiecte de transfer) pentru acest caz de utilizare. Funcțiile din repository nu ajung pentru a implementa (nu avem `getAll()`). Se crează o nouă clasă care conține exact informațiile de care i-a nevoie.



```

def getAllForDisc(self,disc):
    """
        Return all the grades for all the students from a discipline
        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")
  
```

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

În limbajele cu dynamic typing, valorile au un tip (int, str...) dar variabilele nu, deoarece tipul variabilelor este verificat la timpul de execuție (runtime), nu la timpul de compilare. Asta înseamnă că variabilele nu au un tip fixat declarat, dar valorile le pot da.

Ex: $x = 10$; x este acum un integer
 $x = "text"$; x devine un string

Duck Typing

Ești un stil de dynamic typing în care se decide la timpul de execuție dacă un obiect este compatibil cu o anumită utilizare, pe baza comportamentului său, și nu pe baza relației

de menținere sau a tipului său explicit.

Exprisia "Duck Test":

"Dacă arată ca o rată, merge ca o rată și face ca o rată, atunci este o rată" - Nu contează de unde provine obiectul (ca clasă), dată timpul căt arătă metodele și atributele necesare pentru a fi folosit ca și cum ar fi un anumit tip.

```
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

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

l = [Student(1, "Ion"), Professor("1", "Popescu", "FB"), 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)
```

cum denă
clase difriri

Duck Typing: acest lucru funcționează deoarece, indiferent de clasa obiectului, atât Student, cât și Professor au metodele getName() și getId(). Python nu verifică explicit tipul obiectului; se bazează doar pe faptul că metodele necesare există.

Duck Typing - Repository

Clasele de tip Repository:

GradeRepository și GradeFileRepository sunt doar implementări care respectă aceeași interfață publică (1)

StudentRepository și StudentFileRepository sunt doar implementări care respectă aceeași interfață publică (2)

Din (1) și (2) ⇒ deoarece interfața publică este identică, ele pot fi utilizate interschimbabil în servicii. Asta înseamnă că un controller / serviciu poate lucra cu orice implementare fără a necesita 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()
```