Curs 7 – Principii de proiectare

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

Curs 6: Principii de proiectare

- Diagrame UML
- Șabloane 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 in timp ce dezvoltam aplicația **High Cohesion** – fac metode/clase/module cu o singura responsabilitate **Low coupling** – reduc dependentele intre 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** – daca știu/mă aștept sa se modifice/sa existe mai multe variante => creez o clasa care conține functionalitatea **Creator** – cum decid cine creează obiecte **Pure Fabrication** – Repositoy – creez o clasa care reprezintă un depozit de obiecte	
Layered	Arhitectura stratificata – GRASP High Cohesion, Low coupling **UI** – interfața 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** –Persistenta, cod fișiere - GRASP Pure Fabrication	Layere – organizam pe pachete Python Clase – cu responsabilități bine definite
Diagrame UML de clase	Reprezentare grafica pentru structura applicatiei	

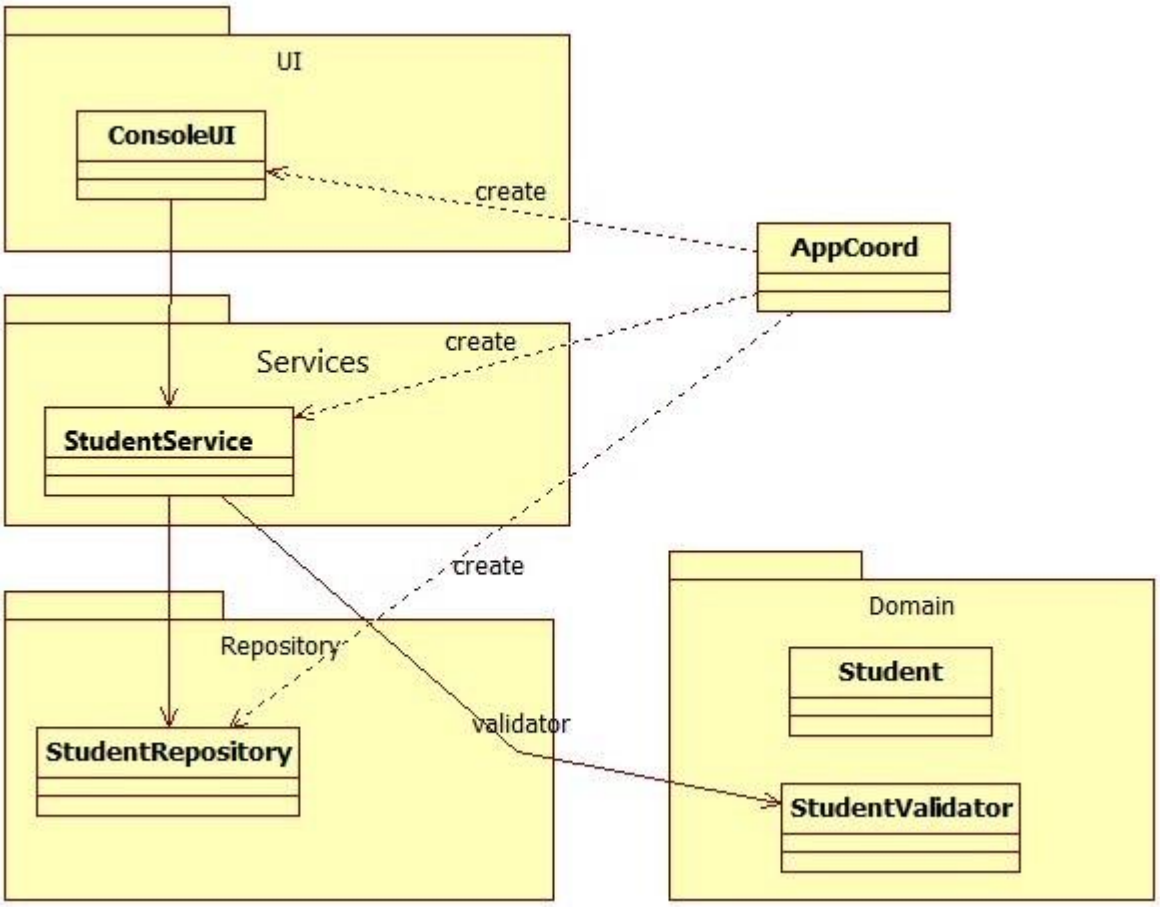
Arhitectură stratificată (Layered architecture)

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 applicație



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 existenta lor (in memorie, scris in fișier, încărcat, etc) se menține identitatea si trebuie asigurat consistenta (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 reala, conceptual ele nu au identitate.
Reprezintă aspecte descriptive din domeniu. Când ne preocupă doar atributele unui obiect (nu și identitatea) clasificam 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.getAdr().getStreet()=="Adr"

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

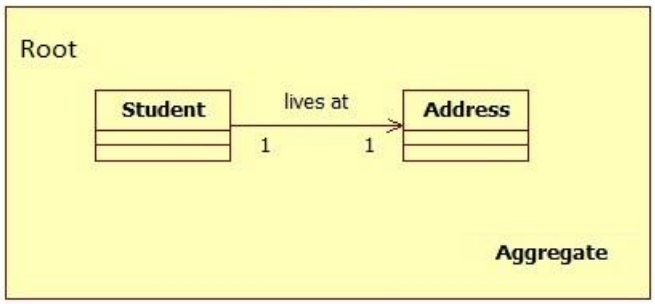
```
class Student:
    """
      Represent a student
    """
    def __init__(self, id, name, adr):
        """
         Create a new student
         id, name String
         address - Address
        """
        self.__id = id
        self.__name = name
        self.__adr = adr

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

Agregate și Repository

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

Obiectele din afara agregatului ar trebui să aibă referința 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 daca 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 in prezenta excepțiilor

Orice fișier pe care deschidem cu *open* ar trebui sa închidem folosind metoda *close*()
Ciclu de viată pentru o resursa: Crearea/Achiziție -> Folosire Resursa -> eliberare/distrugere

Problema:	Solutie

`def applyToFile(fileName):`
`  """`
`  process file line by line`
`  """`
`  fh = open(fileName)`
`  for line in fh:`
`  processLine(line)`
`  fh.close()`
	`def applyToFile(fileName):`
	`  """`
	`  process file line by line`
	`  """`
	`  fh = open(fileName)`
	`  try:`
	`  for line in fh:`
	`  processLine(line)`
	`  finally:`
	`  fh.close()`

Aparent codul de mai sus gestionează corect resursa (fișier) Ce se întâmpla daca funcția *processLine* arunca excepție? Fișierul rămâne deschis	Rezolva problema: se închide fișierul chiar daca apare o excepție in metoda *processLine* Codul pare complex, clauza try/finally face codul mai greu de urmărit

Valabil si in cazul altor resurse pe care trebuie sa gestionam.
Instrucțiunea `with` rezolva 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()) si la execuție normala si daca apare o excepție in corpul instrucțiunii `with` fișierul se va închide.

In cazul in care apare o excepție in corpul instrucțiunii , excepția este aruncata (nu dispare excepția, doar se asigura ca 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 bidirecționale de exemplu presupun ca fiecare obiect din asociere se poate folosi/înțelege/refolosi doar împreună

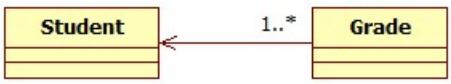
Este important să simplificăm aceste relații cât de mult posibil, prin:

- Impunerea unei direcții (transformare din bi-direcțional în unidirecțional)
- Reducerea multiplicității
- Eliminarea asocierilor ne-esențiale

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

Asocieri

Exemplu Catalog



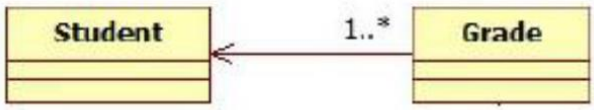

```
gr = ctr.assign("1", "FP", 10)
assert gr.getDiscipline()=="FP"
assert gr.getGrade()==10
assert gr.getStudent().getId()=="1"
assert gr.getStudent().getName()=="Ion"
```

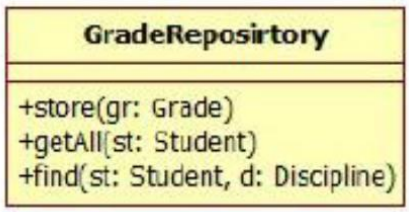
```
st = Student("1", "Ion",
       Address("Adr", 1, "Cluj"))

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 sa avem în vedere anumite aspecte.





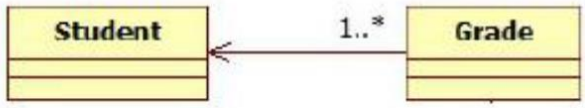
În exemplul de mai sus GradeRepository salvează doar id-ul studentului (nu toate câmpurile studentului) astfel nu se poate implementa o funcție getAll în care se returnează toate notele pentru toți studenții. Se poate în schimb oferi metoda getAll(st) care returnează toate notele pentru un student dat

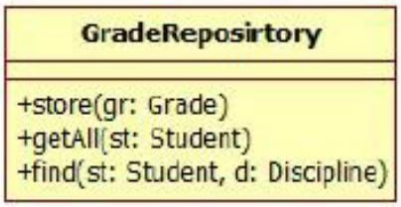
```
def store(self, gr):
        """
          Store a grade
          post: grade is in the repository
          raise GradeAlreadyAssigned exception if we already have a grade
                        for the student at the given discipline
          raise RepositoryException if there is an IO error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

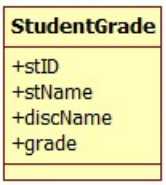
Obiecte de transfer (DTO - Data transfer objects)

Funcționalitate: Primi 5 studenți la o disciplină. Prezentați in 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 noua 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 RepositorException("Unable to read grades from the file")
```

DTO – Data transfer obiect

În 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 publica este dată de multimea metodelor și câmpurilor accesibile din exterior. Două clase pot avea același interfața publică chiar dacă nu exista 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

```
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", "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)
```

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, Agregate
- Fișiere in python
- Asocieri, Obiecte de transfer DTO

Curs 8 – Testarea programelor

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