

Curs 6: Principii de proiectare

- Diagrame UML
- Șabloane GRASP
- Arhitectură stratificată:
 - ui – controller – domain - repository

Curs 5: Tipuri definite de utilizator

- Programare orientată obiect
- Principii de definire a tipurilor utilizator

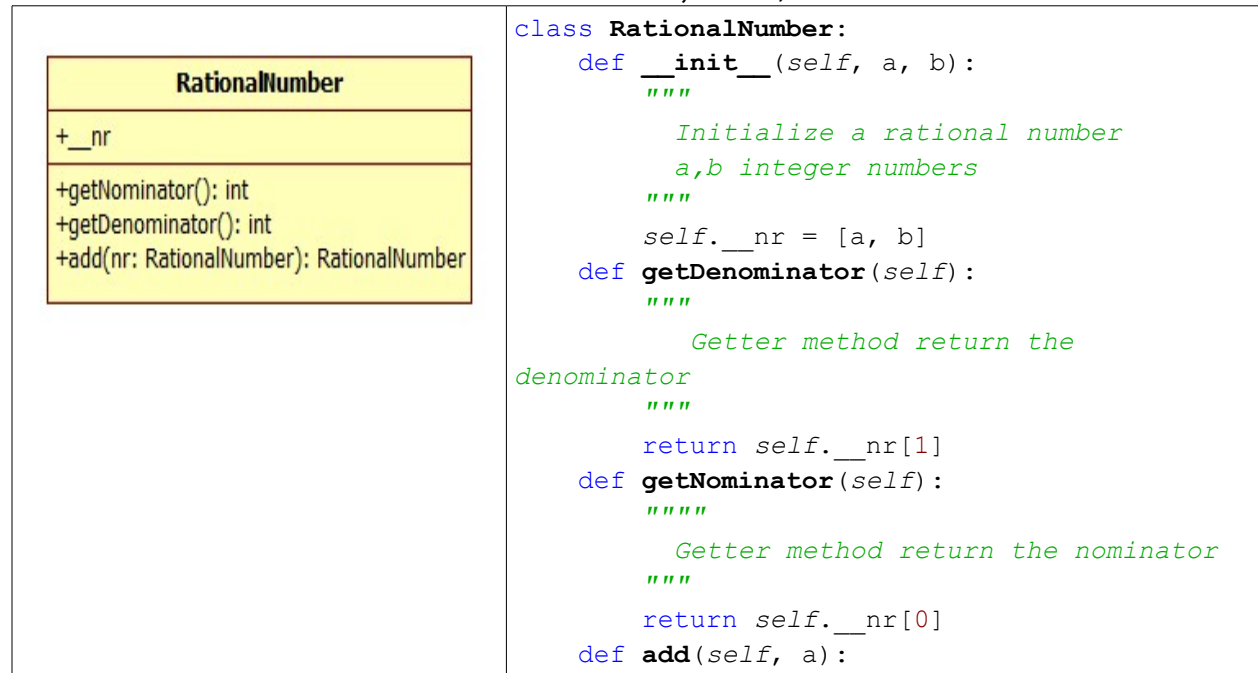
Diagrame UML

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



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

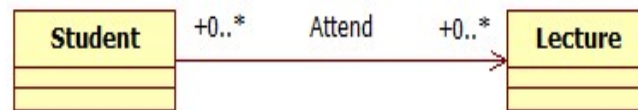
Relații UML

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

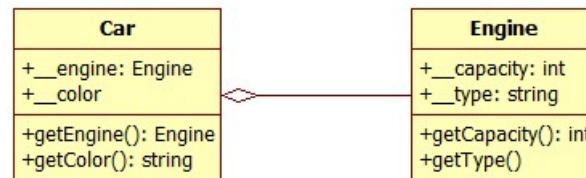
Asocierile binare se reprezintă printr-o linie între două clase.



O asociere poate avea nume, capetele asocieri pot fi adnotate cu nume de roluri, multiplicitate, vizibilitate și alte proprietăți. Asocierea poate fi uni-direcțională sau bi-direcțională

Agregare

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



```
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
```

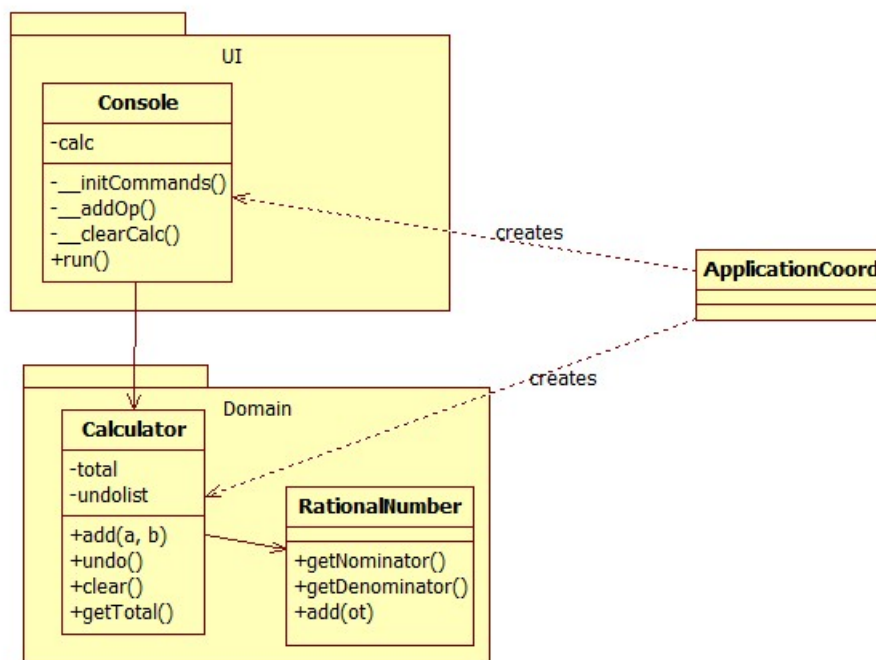
```
class Engine:
    def __init__(self, cap, type):
        """
        initialize the engine
        cap positive integer
        type string
        """
        self.__capacity = cap
        self.__type = type
    def getCapacity(self):
        """
        Getter method for the capacity
        """
        return self.__capacity
    def getType(self):
        """
        Getter method for type
        return string
        """
        return self.__type
```

Dependențe, Pachete

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

Exemple de dependențe:

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



Principii de proiectare

Crează 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 dependențelor

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

Enunț (Problem statement)
Scrieți un program care gestionează studenți de la o facultate (operații CRUD – C reate R ead U ppdate D eleate)

	Funcționalități (Features)
F1	Adauga 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	
'lon'		
	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)**

Șabloane Grasp

General Responsibility Assignment Software Patterns (or **Principles**)
conțin recomandări pentru alocarea responsabilităților pentru clase obiecte într-o aplicație orientat 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, reutilizate, întreținute și sunt o piedică pentru modificările necesare pe parcursul dezvoltării unui sistem

Low Coupling

Alocă responsabilități astfel încât cuplarea rămâne slabă (redușă)

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 (parameterii, 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

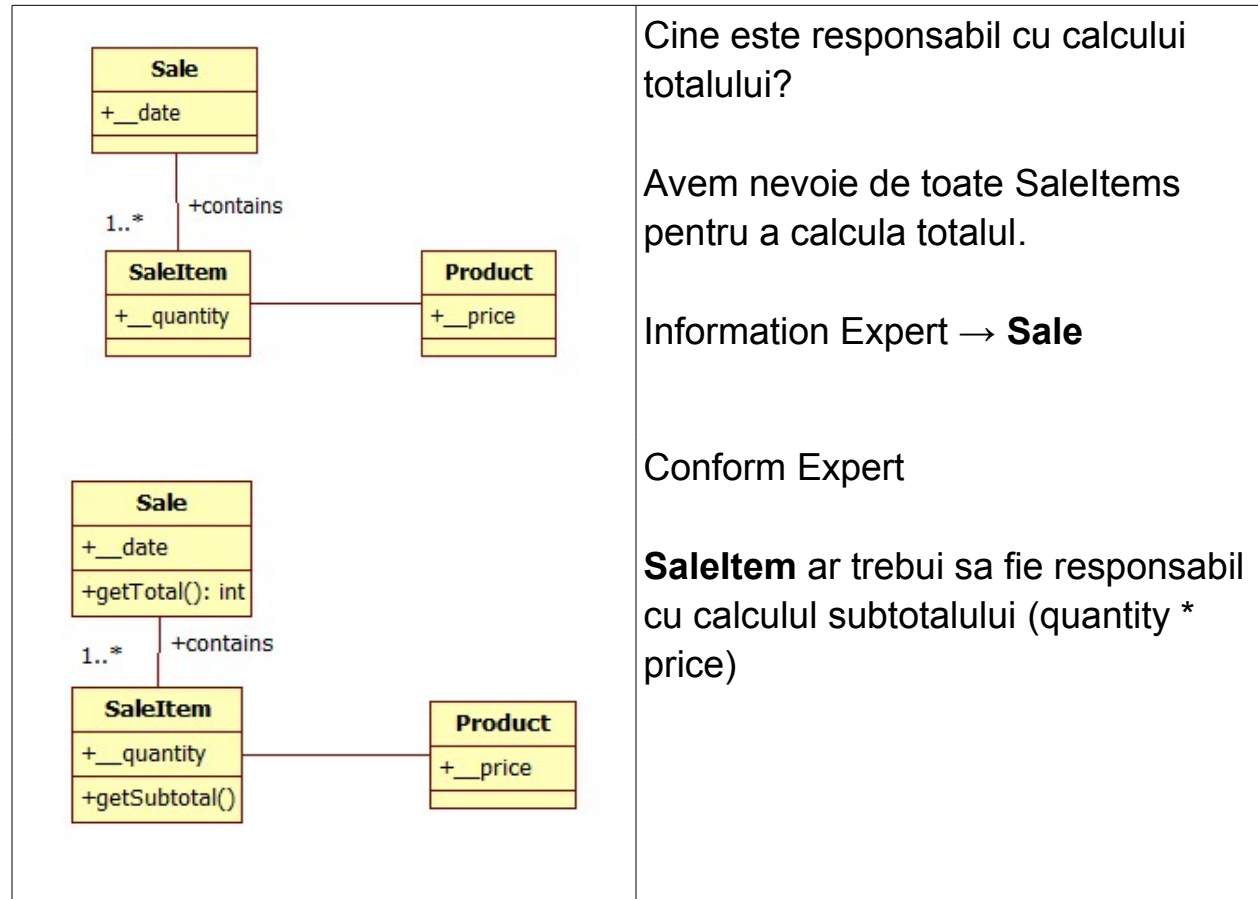
Information Expert este un principiu care ajută să determinăm care este clasa potrivită care ar trebui să primească responsabilitatea (o metodă nouă, un câmp, un calcul).

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 coezive
4. Poate sa conducă 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.

Șablonul 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

```
def testCreateStudent():  
    """  
        Testing student creation  
    """  
    st = Student("1", "Ion", "Adr")  
    assert st.getId() == "1"  
    assert st.getName() == "Ion"  
    assert st.getAdr() == "Adr"
```

```
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 getAdr(self):  
        return self.adr
```


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.

Șablonul **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 funcție
- 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 Principiu Protect Variation

```
def testStudentValidator():
    """
    Test validate functionality
    """
    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

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)
```

Pure Fabrication

Când un element din sistem încalcă principiul coeziunii ridicate și cuplare slabă (în general din cauza aplicării succesive a șablonului 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** (în memorie, fișier sau bază de date)

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

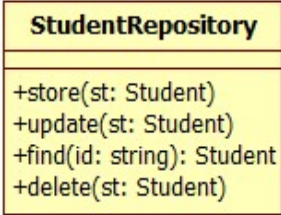
Pure Fabrication - Repository

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 in clasa Student, rezultă o clasă slab coeziva, cu potențial limitat de refolosire

Soluție – Pure Fabrication

 <pre>classDiagram 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 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>
--	--

Șablonul Repository

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

Obiecte sunt adăugate, șterse, modificate iar codul din repository insereaza, sterge 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.

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.

Controlerul 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 controller

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


class StudentController:
    """
    Use case controller 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 (depinde 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()
#crate repository
rep = InMemoryRepository(None)
#create console provide(inject) a validator and a repository
ctr = StudentController(rep, validator)
#create console provide controller
ui = Console(ctr)
ui.showUI()
```

Review aplicația student manager – de revazut șabloanele ce apar

Curs 6: Principii de proiectare

- Diagrame UML
- Șabloane GRASP
- Arhitectură stratificată:
 - ui – controller – domain - repository

Curs 7 – Principii de proiectare

- Entăți, ValueObject, Agregate
- Fișiere in python
- Moștenire – refolosire de cod