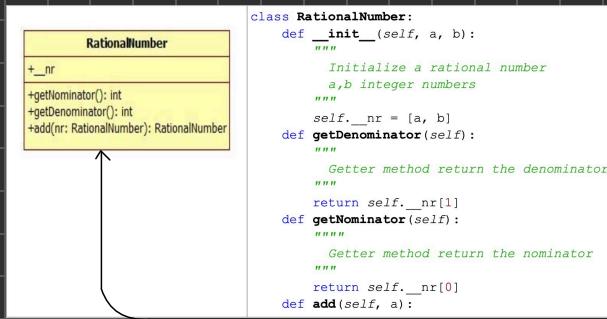


# Fundamentele programării

# Diagramme UML

Este un desen care te ajută să înțelegi cum este organizat parții unui sistem și cum funcționează acesta.

## Diagramme de clase

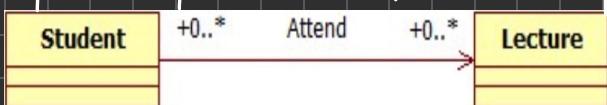


Clase sunt reprezentate prin dreptunghiuri ce conțin trei zone:

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

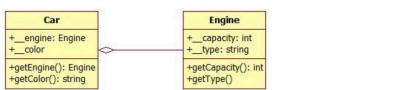
## Relații UML

Descriv o legătură logică între două clase. Asocierile se reprezintă printr-o linie, iar aceasta poate fi uni-directională sau bi-directională.



# Agregare

Este o asociere specializată și reprezintă apartenență.



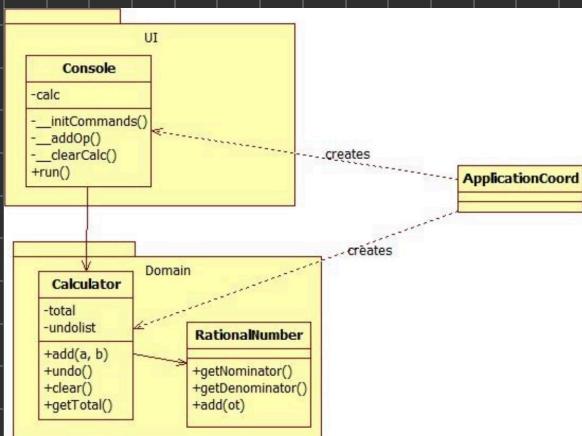
```

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 self.__color
    def getEngine(self):
        """
        Getter method for 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 self.__type
  
```

# Dependențe, Pachete

Exemplu de dependențe:

- crează instanțe
- au un parametru
- folosesc un obiect în interiorul unei metode



# Principii de proiectare

Caracteristici de proiecte care:

- sunt ușor de înțeles, modificat, întreținut, testat
- clasele sunt abstrakte, încapsulare, ascunderea reprezentării, ușor de folosit / nefolosit

Scop general: gestionarea 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 – Create Read Update Delete)

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

Plan iterativ: 11 : F1 , 12 : F2 , 13 : F3 , 14 : F4

Scenariu de rulare

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	

## Arhitectura stratificată

Ește 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. Un layer este un grup de clase care au același set de dependențe și se pot folosi în circumstanțe similare.

- UI (user interface layer)
- Application layer (Service layer / Controller layer)
- Domain layer (Model layer)
- Infrastructure layer (acces la date)

# Sabioane GRASP

Conțin recomandări pentru alocarea responsabilităților pentru clase și obiecte într-o aplicație orientată pe obiecte.

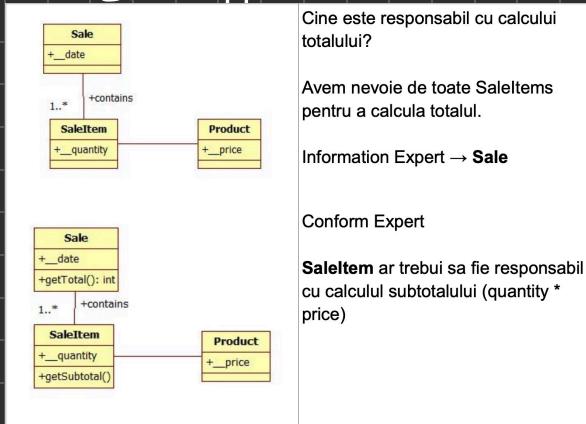
- High Cohesion
- Low Coupling
- Information Expert
- Controller
- Protected Variations
- Creator
- Pure Fabrication

Layer	
Domain	Entități din domeniul problemei
Validators	Protected variations
Repository	Gestionază colectiv entități
Controller	Creator
UI	Citiri Afisari
ALL	High Cohesion, Low Coupling

# Information Expert

Ești un principiu care ajută să determinăm care este clasa potrivită care ar trebui să primească responsabilitate.

Ex: Sale application



# Creator

Crearea de obiecte este o activitate importantă asociată unei clase. Clasa care are o relație puternică cu obiectele sunt cele care ar trebui să fie responsabile pentru crearea acestor obiecte.

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

<pre>def testCreateStudent():     """     Testing student creation     """     st = Student("1", "Ion", "Adr")     assert st.getId() == "1"     assert st.getName() == "Ion"     assert st.getAdr() == "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 getAdr(self):         return self.__adr</pre>
--	--

# Protected Variations

Acest principiu are scopul de a proteja sistemul astfel încât schimbările să nu necesite modificări majore. Soluția este să se creze o nouă clasă care să încapsuleze variabile.

## Validare

Ex: validate student

<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)&gt;0:             raise ValueError(errors)</pre>
---	--

Alte designuri posibile:

- Algoritmul de validare:

- poate fi o metodă în clasa student
- poate fi o metodă statică
- poate fi încapsulat în altă clasă

- Poate semnaliza erori prin:

- returnarea T/F
- returnarea listei de erori / exceptii care contin lista de erori

## Pure Fabrication (Repository)

Scopul acestui principiu este să rezolve probleme legate de design cum ar fi: high cohesion și low coupling. De ex: unele clase ajung să aibă prea multe responsabilități, spm ex. clasa „Student”. Acest lucru încalcă principiile de design, cum ar fi:

- High cohesion (o clasă trebuie să fie responsabilă doar pentru o sarcină bine definită)
- Low coupling (clasele ar trebui să fie cât mai puțin dependente între ele)

Soluția: introducem o clasă „artificială” Repository care este creată pentru a prelua o parte din responsabilități.

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

Această clasă se ocupă de adăugarea, stergerea și modificarea obiectelor.

Ex:

```

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

```

## Controller

Este o clasă care gestionază evenimentele generate de utilizator. Controllerul servește ca un intermedian între UI și restul aplicației, controllerul este cel care decide ce operații trebuie realizate pentru a răspunde cererii. Controllerul nu face lucruri propriu-zise, ci doar controlază fluxul de operații.

Ex:

```

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

DI este un principiu de proiectare prin care dependențele unui obiect sunt injectate din exterior pentru a reduce dependența directă. Ex:

```

#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()

```

StudentService primește validatorul și repository-ul ca parametri. StudentService nu știe cum sunt implementate acestea.