Curs 8 – Testarea programelor

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

Curs 7 – Principii de proiectare

- Entăți, ValueObject, Agregate
- 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 si variabilele membre din clasa A
- clasa B poate redefini metode din clasa A
- clasa B poate adaugă 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 baza.

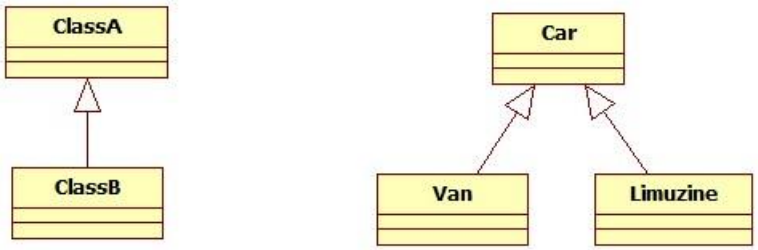
Suprascrierea poate înlocui cu totul metoda din clasa de bază sau poate extinde funcționalitatea (se execută și metoda din clasa de bază dar se mai adaugă cod)

O metodă simplă să apelăm o metodă în clasa de bază:
BaseClassName.methodname (self,arguments)

Diagrame UML – Generalizare (moștenire)

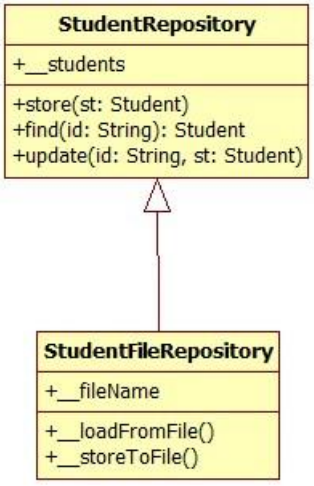
Relația de generalizare ("is a") indică faptul că o clasă (clasa derivată) este o specializare a altei clase (clasa de bază). Clasa de bază este generalizarea clasei derivate.

Orice instanță a clasei derivate este si o instanța 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,fileN):
        #properly initialise the base class
        StudentRepository.__init__(self)
        self.__fName = fileN
        #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.getAdr().getStreet()+";"+str(st.getAdr().getNr())
+";"+st.getAdr().getCity()
            strf = strf+"\n"
            f.write(strf)
        f.close()
```

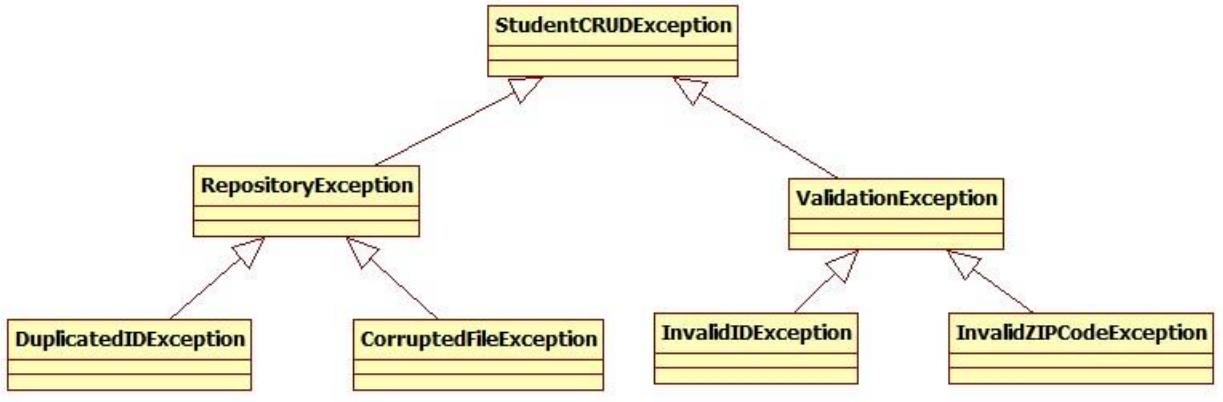
Excepții

```
def __createdStudent(self):
        """
          Read a student and store in the apllication
        """
        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 apllication
        """
        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 excepții

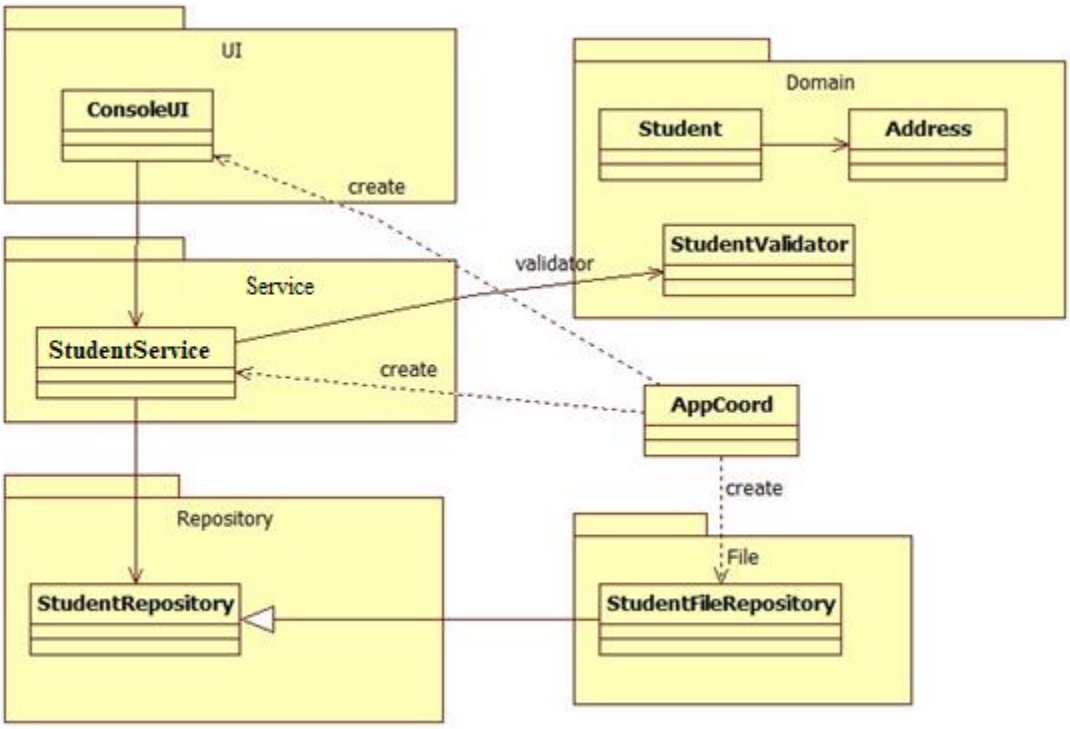



```
class StudentCRUDException(Exception):
    pass
```

```
class ValidationException(StudentCRUDException):
    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 RepositorException(StudentCRUDException):
    """
      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(RepositorException):
    def __init__(self):
        RepositorException.__init__(self, "Duplicated ID")
```

```
def __createdStudent(self):
        """     Read a student and store in the apllication     """
        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 StudentCRUDException as ex:
            print (ex)
```

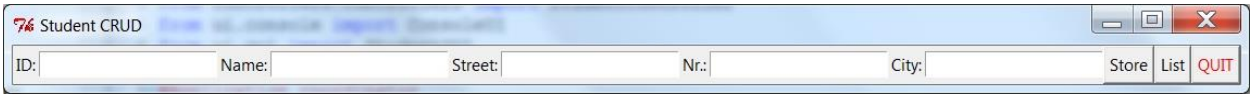
Layered arhitecture – Structură proiect



Layered architecture – GUI Example

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

Testarea programelor

Testarea este observarea comportamentului unui program în multiple execuții.
Se execută programul pentru ceva date de intrare și se verifică daca rezultate sunt corecte în raport cu intrările.

Testarea nu demonstrează corectitudinea unui program (doar oferă o anumită siguranța , confidență). In 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 executări 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.assertRaises(ValidationEx,self.ctr.create,"1", "", "", 1, "Cj")

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

    def testRemove(self):
        #test for an invalid id
        self.assertRaises(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()
```

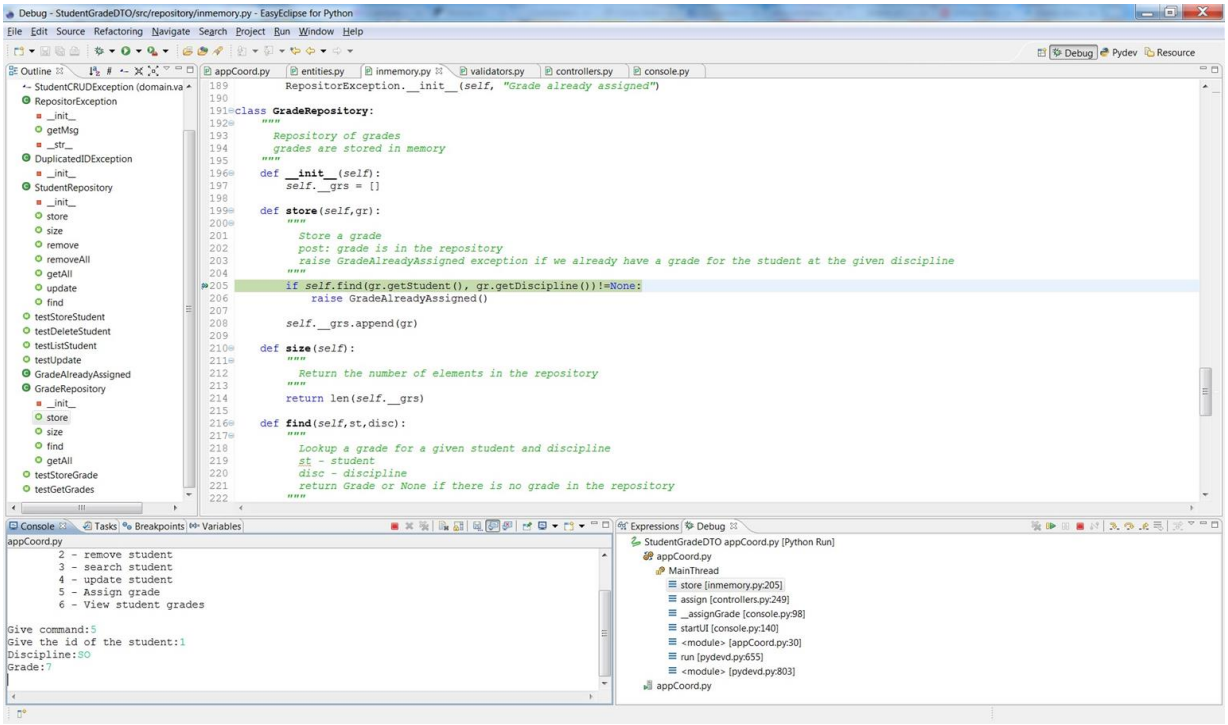
Depanare (Debugging)

Depanarea este activitatea prin care reparăm erorile găsite în urma testării.
Dacă testarea indică prezența unei erori atunci prin depanare în cercăm să identificăm cauza erorii, modalități de rezolvare. Scopul este sa 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 sa folosiți același reguli de denumire in toată aplicația

Curs 8 – Testarea programelor

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

Curs 9 – Complexitate

- Recursivitate
- Complexitate