Fundamentele programării

Moștenire

Permite claselor derivate care moștenește comportamentul (metode) și caracteristicile de la clasa de bază deja existentă.

Dacă A și B sunt două clase unde B moștenește de la clasa A (B este derivat din clasa A) atunci:

- clasa B are toate metodele și variabilele membre din clasa A
- clasa B poate redefini metode din clasa A
- clasa B poate adăuga noi membrii (variabile, metode) pe lângă cele moștenite de la clasa A.

Reutilizare de cod

Unul din motivele pentru care folosim moștenire este reutilizarea codului existent într-o clasă.

Moștenire în Python

Sintaxă:

```
class DerivedClassName(BaseClassName)
```

Clasa derivată moștenește:

- câmpuri

• metode

Când accesăm un membru, acesta 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"
```

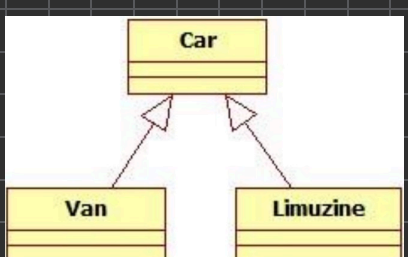
```
class A:
  def __init__(self):
    print ("Initialise A")

  def f(self):
   print("in method f from A")

  def g(self):
   print("in method g from A")
```

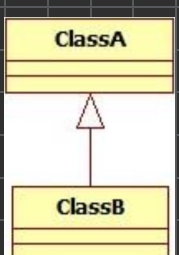
```
b = B()
#f is inherited from A
b.f()
b.g()
```

Clasele derivate pot suprascrie metodele clasei de bază.

Diagrama UML - moștenire

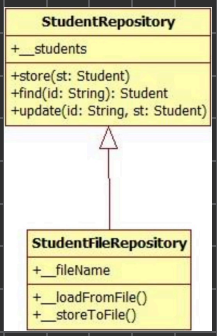
Orice instanță a clasei derivate este și o instanță a clasei de bază.





Repository cu fișiere

```
class StudentFileRepository(StudentRepository):
    """
      Repository for students (stored in a file)
    """
    pass
```

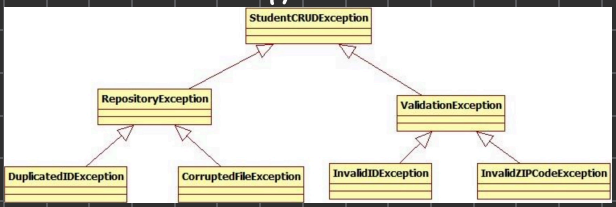



```
class StudentFileRepository(StudentRepository):
    """
      Store/retrieve students from file
    """
    def __init__(self,fileN):
        #properly initialise the base class
        StudentRepository.__init__(self)
        self.__fName = fileN
        #load student from the file
        self.__loadFromFile()

    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()
```

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 exceptii

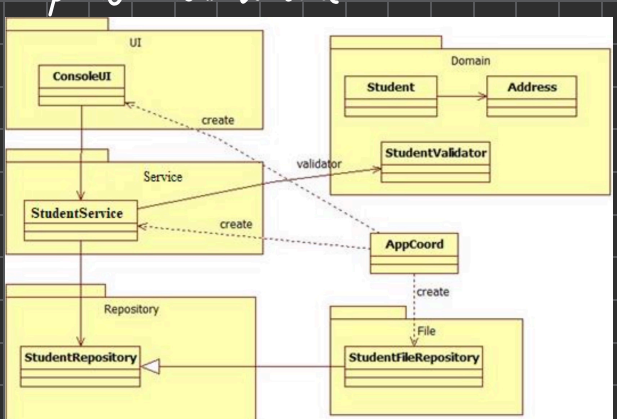



```
class StudentCRUDException(Exception):
    pass
```

```
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 RepositorExeception(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 architecture



Modele 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

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 limită, condiții de eroare.

- White box testing

Datele de test se aleg analizând codul sursă. Alegem datele astfel încât să acoperim toate ramurile de execuție (în urma testelor, fiecare instrucțiune din program este executată 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

Unit testing

Testarea se face pe funcționalități izolate, pe componente individuale. Se verifică dacă fiecare unitate funcționează corect independent de restul aplicației.

Integration testing

Se testează interacțiunea dintre componente, pentru a verifica dacă funcționează împreună așa cum ne-am așteptat.

Testare automată

Presupune scrierea de programe care realizează testarea. Acest lucru este posibil folosind PyUnit care este o bibliotecă Python pentru unit testing.

Modulul unittest oferă:

- teste automate
- modalități uniforme de pregătire / curățare necesare pentru teste

```
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()
```

Inspectarea programelor

Any fool can write code that a computer can understand.
Good programmers write code that humans can understand.

Stil de programare

Principalul atribut al codului sursă este considerat ușurința de a citi.
Elementele stilului de programare sunt:

- comentarii
- formatarea textului (identare, white spaces)
- specificații
- denumiri sugestive (pentru clase, funcții, variabile) din program