Lecture 7 – Design principles

- Entity, ValueObject, Aggregates
- Files in python
- Inheritence

Layered arhitecture

Partition a complex program into layers.

The essential principle is that any element of a layer depends only on other elements in the same layer or on elements of the layers "beneath" it.

Develop a design within each layer that is cohesive and that depends only on the layers below

Layered Arhitecture – Responsabilities

Presentation Layer

User Interface : Responsible for showing information to the user, collect infromation from the user, and interpreting the user's commands

Middle Layer

Controller

The first object beyond the UI layer that receives and coordinates ("controls") a system operation.

Entities

Responsible for representing concepts of the business

Repositories

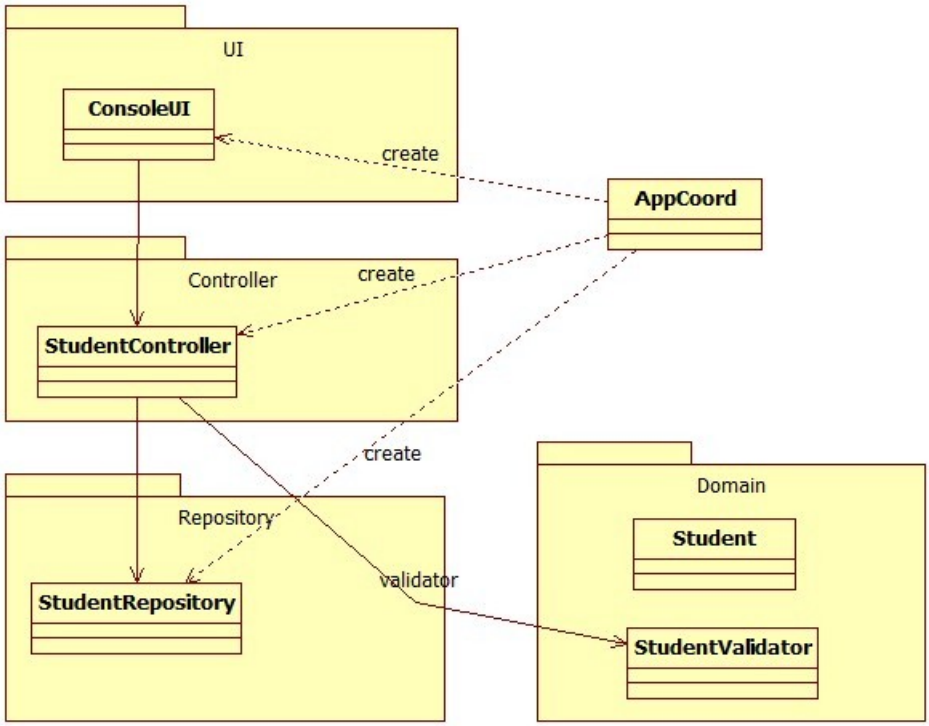
Provide methods to add and remove objects, will encapsulate the actual insertion or removal of data in the data store.
Provide methods that select objects based on some criteria and return

Infrastructure Layer

Provides generic technical capabilities (utility, files, database, etc)

StudentCRUD application

Review StudentCRUD application



Entities

Entity is an object that is not defined by its attributes, but rather by a thread of continuity and its identity
Does an object represent something with continuity and identity—something that is tracked through different states or even across different implementations?

- define what it means to be the same thing.

```
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
```

Attributes of the entity may change but the identity remain the same

Mistaken identity can lead to data corruption.

Value Objects

Many objects have no conceptual identity. These objects describe some characteristic of a thing

An object that represents a descriptive aspect of the domain with no conceptual identity is called a VALUE OBJECT.

When you care only about the attributes of an element of the model, classify it as a 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
```

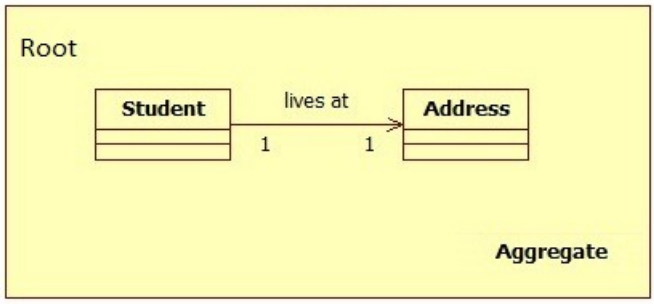
```
class Student:
    """
      Represent a student
    """
    def __init__(self, id, name, adr):
        """
         Create a new student
         id, name String
         address - Address
        """
        self.__id = id
        self.__name = name
        self.__adr = adr

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

Aggregates and Repositories

Cluster the ENTITIES and VALUE OBJECTS into AGGREGATES and define boundaries around each.

Choose one ENTITY to be the root of each AGGREGATE, and control all access to the objects inside the boundary through the root.

Allow external objects to hold references to the root only.



Repositories provide the illusion of an in-memory collection of all objects of that type

Provide REPOSITORIES only for AGGREGATE roots.

Only StudentRepository (no AddressRepository)

Working with text files in Python

Built in function: **open()** returns a file object, and is most commonly used with two arguments: open(filename,mode).

Filename – string representing the path to the file (absolute or relative path)

Mode:

"r" – open for read

"w" – open for write (overwrites the existing content)
"a" – open for append

Methods:
write(str) – write the string to the file
readline() - read a line from the file, return as a string
read() - read the entire file, return as a string
close() - close the file, free up any system resources

Exception:
IOError – raise this exception if there is an input/output error (no file, no disk space, etc)

Python text file examples

```
#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()
```

Dynamic Typing

Type checking is performed at run-time as opposed to at compile-time.

In dynamic typing values have types, but variables do not; that is, a variable can refer to a value of any type

Duck Typing

Duck typing is a style of dynamic typing in which an object's methods and properties determine the valid semantics, rather than its inheritance from a particular class or implementation of a specific interface.

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)
```

Inheritance

Classes can inherit attributes and behavior (i.e., previously coded algorithms associated with a class) from pre-existing classes called base classes or superclasses or parent classes.
The new classes are known as **derived classes** or **subclasses** or child classes.
The relationships of classes through inheritance gives rise to a hierarchy.

Code reuse

One of the motivations for using inheritance is the reuse of code that already exist in another class (implementation inheritance).

Before the object-oriented paradigm was in use, one had to write similar functionality over and over again.

With inheritance, behaviour of a superclass can be inherited by subclasses. It not only possible to call the overridden behaviour (method) of the ancestor (superclass) before adding other functionalities, one can override the behaviour of the ancestor completely.

Inheritance in Python

Syntax:
class DerivedClassName(BaseClassName):

DerivedClass will inherit:

- fields:
- methods

If a requested attribute (field,method) is not found in the class, the search proceeds to look in the base class

Derived classes may override methods of their base classes.

An overriding method in a derived class may in fact want to extend rather than simply replace the base class method of the same name.

There is a simple way to call the base class method directly: call BaseClassName.methodname(self,arguments)

```
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"
```

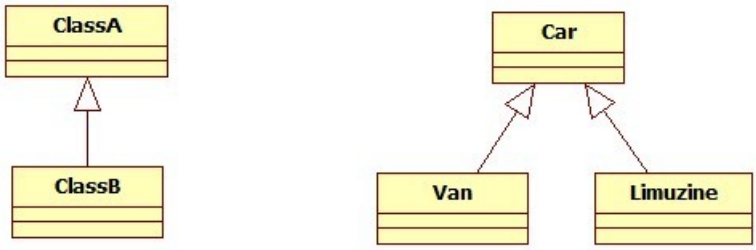
```
class A:
  def __init__(self):
     print "Initialise A"

  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()
```

UML Diagram – Generalization

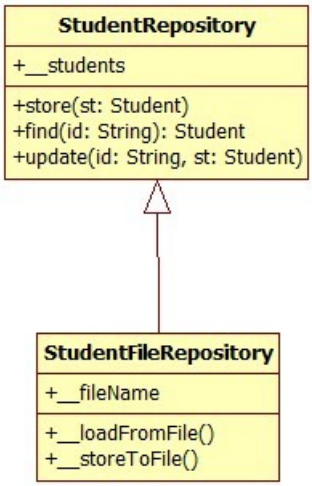
The Generalization relationship ("is a") indicates that one of the two related classes (the subclass) is considered to be a specialized form of the other (the super type) and superclass is considered as 'Generalization' of subclass.
Any instance of the subtype is also an instance of the superclass.

The generalization relationship is also known as the inheritance or "is a" relationship.



File Repositories

```
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()
```

Overwriting methods

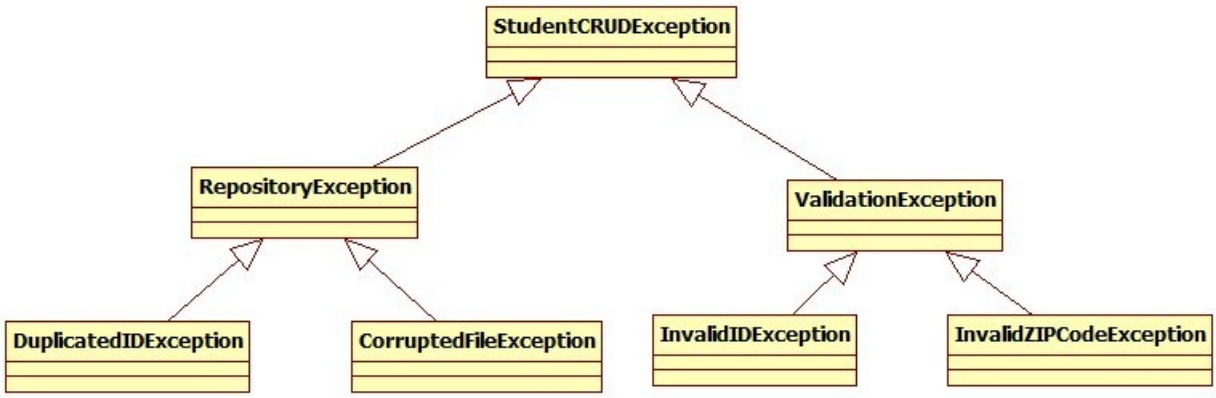
```
def testStore():
    fileName = "teststudent.txt"
    repo = StudentFileRepository(fileName)
    repo.removeAll()

    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()
```

Exceptions

```
def __createdStudent(self):
        """
          Read a student and store in the apllication
        """
        id = raw_input("Student id:").strip()
        name = raw_input("Student name:").strip()
        street = raw_input("Address - street:").strip()
        nr = raw_input("Address - number:").strip()
        city = raw_input("Address - city:").strip()
        try:
            self.__ctr.create(id, name,street,nr,city)
        except ValueError, msg:
            print msg


def __createdStudent(self):
        """
          Read a student and store in the apllication
        """
        id = raw_input("Student id:").strip()
        name = raw_input("Student name:").strip()
        street = raw_input("Address - street:").strip()
        nr = raw_input("Address - number:").strip()
        city = raw_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)
```

Exception hierarhies




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

```
class ValidationException(StudentCRUDException):
    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)

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 = raw_input("Student id:").strip()
        name = raw_input("Student name:").strip()
        street = raw_input("Address - street:").strip()
        nr = raw_input("Address - number:").strip()
        city = raw_input("Address - city:").strip()
        try:
            self.__ctr.create(id, name,street,nr,city)
        except StudentCRUDException as ex:
            print ex
```

Layered arhitecture - Project structure