Laboratorul 6: Utilizarea POO în Python

Introducere

După cum am discutat limbajul Python suportă patru tipuri de paradigme de programare. Dintre ele cele mai importante sunt cea orientată obiect și cea funcțională.

Deoarece aspectele specifice programării funcționale vor fi tratate în viitor s-a ales pentru acest laborator prezentarea abilităților limbajului cu privire la programarea orientată obiect.

Câteva observații legate de mediul de dezvoltare PyCharm:

- Pentru laborator vom utiliza versiunea gratuită Community care se ia de la același producător ca și Intellij
- NU utilizați NICIODATĂ copy/paste din orice sursă deoarece zăpăcește analizorul sintactic integrat al intelisense-ului și ca începători va fi mai dificil de pus codul într-o manieră funcțională în concordanță cu logica dorită a programului
- Pycharm are un detector automat de sintaxă și ne oferă o facilitate deosebită pentru Python și anume ne marchează explicit prin linii verticale fine buclele și blocurile asociate unei instrucțiuni

Vom începe prin a reaminti o serie de noțiuni minimale după cum urmează

- Tipurile de variabile sunt inițializate automat ca la Kotlin cu care se aseamănă parțial
- Python folosește spațierea (tab-ul nu mai este recomandat dar pycharm îl suportă) pe post de demarcare bloc de instrucțiuni asociat
- Toate clasele din Python sunt derivate din clasa mamă object. Dacă nu vom specifica altă clasă de bază clasa nou creată va fi derivată din object
- Două surse bune cu privire la limbaj se găsesc la https://www.python.org/doc/ și la https://docs.python.org/3/
- **self** este echivalentul lui **this**
- O variabilă simplă se citește cu val=input("dati valoarea")
- O variabilă se afișează cu print("\n val="+str(val))
- Conversiile de tip sunt realizate cu metode ajutătoare
- Comentariile încep cu #
- Scheletul unei funcții main():

```
def main():
    pass
    # functia main() nu este obligatorie în pycharm

if __name__ == "__main__":
    main()
```

- Când se utilizează o interfață metodele din clasă apelează:

```
return NotImplementedError("<nume_functie> nu este implementata")
```

pentru a forța clasele care o implementează (în Python se va folosi moștenirea) să definească funcțiile respective.

- Tratarea parametrilor din linia de comandă se bazează pe funcții din bibliotecasys, fiind limbaj interpretat primul parametru se află în **sys.argv[1]**

Specificarea tipurilor de date

De la versiunea Python >= 3.5, au fost introduse hint-urile în ceea ce privește tipurile variabilelor / funcțiilor. Limbajul Python determină în mod dinamic tipul variabilei la asignare. Deci specificarea tipului nu influențează acest aspect.

Spre exemplu:

```
example: int = 'String'
```

Deși tipul specificat este **int**, tipul variabilei **example** va fi String. Cu alte cuvinte, acestea au rolul de a ușura înțelegerea codului prin specificarea efectivă a tipului de date așteptat.

Se poate specifica și tipul returnat de o funcție:

```
def greeting(name: str) -> str:
    return 'Hello ' + name
```

Se poate crea și un alias:

```
from typing import List
Vector = List[float]

def scale(scalar: float, vector: Vector) -> Vector:
    return [scalar * num for num in vector]

# typechecks; a list of floats qualifies as a Vector.
new_vector = scale(2.0, [1.0, -4.2, 5.4])
```

De asemenea, se poate crea și un tip nou (pe baza unui tip existent):

```
from typing import NewType

UserId = NewType('UserId', int)
some_id = UserId(524313)
```

Pentru mai multe detalii, se recomandă parcurgerea documentației oficiale, disponibilă la adresa: https://docs.python.org/3/library/typing.html

Supraîncărcarea operatorilor („metodele magice”)

Operatori binari:

Operator	**Metoda**
+	object.__add__(self, other)
-	object.__sub__(self, other)
*	object.__mul__(self, other)
//	object.__floordiv__(self, other)
/	object.__truediv__(self, other)
%	object.__mod__(self, other)
**	object.__pow__(self, other[, modulo])
<<	object.__lshift__(self, other)
>>	object.__rshift__(self, other)
&	object.__and__(self, other)

^	object.__xor__(self, other)
|	object.__or__(self, other)

Asignări extinse:

Operator	**Metoda**
+=	object.__iadd__(self, other)
-=	object.__isub__(self, other)
*=	object.__imul__(self, other)
/=	object.__idiv__(self, other)
//=	object.__ifloordiv__(self, other)
%=	object.__imod__(self, other)
**=	object.__ipow__(self, other[, modulo])
<<=	object.__ilshift__(self, other)
>>=	object.__irshift__(self, other)
&=	object.__iand__(self, other)
^=	object.__ixor__(self, other)
|=	object.__ior__(self, other)

Operatori unari

Operator	**Metoda**
-	object.__neg__(self)
+	object.__pos__(self)
abs()	object.__abs__(self)
~	object.__invert__(self)
complex()	object.__complex__(self)
int()	object.__int__(self)
long()	object.__long__(self)
float()	object.__float__(self)
oct()	object.__oct__(self)
hex()	object.__hex__(self

Operatori de comparare

Operator	**Metoda**
<	object.__lt__(self, other)
<=	object.__le__(self, other)
==	object.__eq__(self, other)
!=	object.__ne__(self, other)
>=	object.__ge__(self, other)
>	object.__gt__(self, other)

Pentru mai multe detalii, se recomandă parcurgerea documentației oficiale, disponibilă la adresa: https://docs.python.org/3/library/unittest.mock.html#magicmock-and-magic-method-support, precum și următoarea resursă: https://www.python-course.eu/python3_magic_methods.php

Specificatori de acces

```
class AccessModifiers:
    public_variable = "This is public"
    _protected_variable = "This is theoretically protected"
    __private_variable = "This is theoretically private"

    def __init__(self):
        self.print_members()

    def set_private_variable(self, new_value):
        self.__private_variable = new_value

    def print_members(self):
        print(self.public_variable)
        print(self._protected_variable)
        print(self.__private_variable)

    def public_method(self):
        print("I'm a public method")

    def _protected_method(self):
        print("I'm a protected method")

    def __private_method(self):
        print("I'm a private method")


if __name__ == '__main__':
    access_modifiers = AccessModifiers()
    access_modifiers.public_variable = 0
    access_modifiers._protected_variable = 1
    access_modifiers.__private_variable = 2
    access_modifiers.print_members()

    # setarea variabilei private printr-o metoda setter
    access_modifiers.set_private_variable(2)
    access_modifiers.print_members()

    access_modifiers.public_method()
    access_modifiers._protected_method()
    try:
        access_modifiers.__private_method()
    except AttributeError as e:
        print("Error", e)
```

Specificatorii de acces sunt introduși practic prin convenția de notare:

- Pentru variabile si funcții publice, convenția **snake_case** (fără vreun underscore ca prefix)
- Pentru variabile și funcții protejate, convenția **_snake_case** (un underscore ca prefix)
- Pentru variabile și funcții private, convenția **__snake_case** (două underscore-uri ca prefix)

Convenții de denumire

- Pentru nume de clase: **PascalCase**
- Pentru nume de variabile / metode: **snake_case**
- Pentru constante: **NUME_CONSTANTA**

Reguli de scriere a codului: PEP8

A SE CONSULTA DOCUMENTAȚIA OFICIALĂ:
https://www.python.org/dev/peps/pep-0008/

DUCK TYPING (interfețele automate)

„Dacă arată ca o rață, înoată ca o rață și măcăie ca o rață, atunci probabil că e o rață”

Pentru o analogie cu C++, pentru polimorfism se dă codul de mai jos:

```
class Macaitoare:
    def quack(self):
        pass

class Duck(Macaitoare):
    def quack(self):
        print("Duck: Quack quack")

    def __repr__(self):
        return "Duck"


class Goose(Macaitoare):
    def quack(self):
        print("Goose: Quack quack")

    def __repr__(self):
        return "Goose"


if __name__ == '__main__':
    duck: Macaitoare = Duck()
    goose: Macaitoare = Goose()

    for macmac in [duck, goose]:
        macmac.quack()
        print(macmac)
```

Se remarcă Existența clasei **Macaitoare** si clasele Duck și Goose care o moștenesc. Datorită interfețelor automate din python (duck typing), nu este necesară o asemenea abordare, exemplul de mai sus reducându-se la:

```
class Duck:
    def quack(self):
        print("Duck: Quack quack")

    def __repr__(self):
        return "Duck"
```

```
class Goose:
    def quack(self):
        print("Goose: Quack quack")

    def __repr__(self):
        return "Goose"


if __name__ == '__main__':
    duck = Duck()
    goose = Goose()

    for macmac in [duck, goose]:
        macmac.quack()
        print(macmac)
```

Exemple

Exemplu de clasă utilizată în crearea unei liste

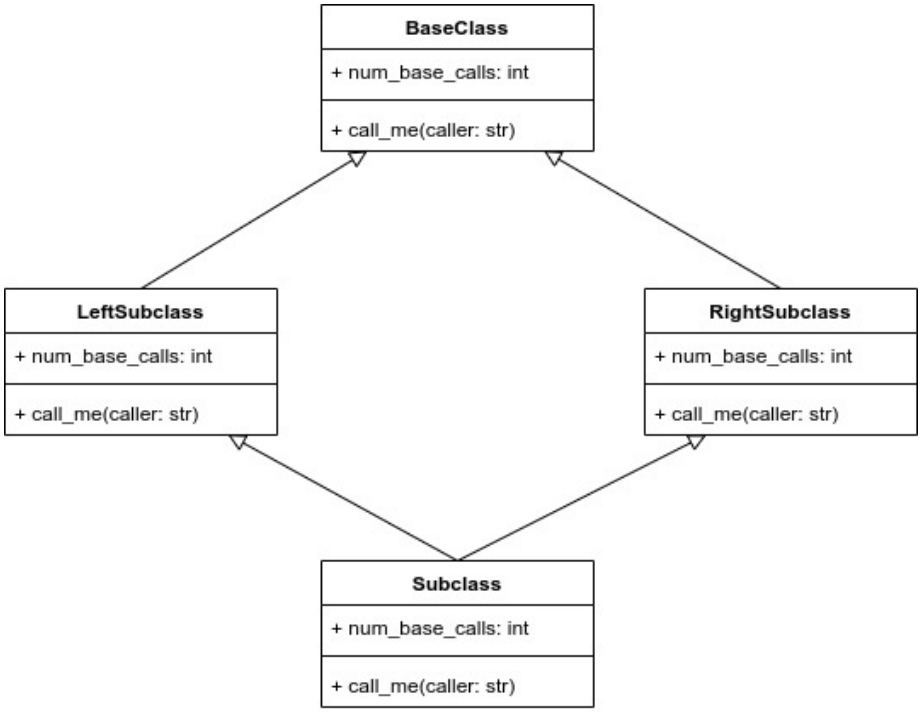
Observație: __init__ = underscore underscore **init** underscore underscore = **D**ouble **UNDER**score **init** **D**ouble **UNDER**score = dunder **init** dunder (o metodă mai simplă de a citi numele funcției)

```
class ContactList(list):
    def search(self, name):
        matching_contacts = []
        for contact in self:
            if name in contact.name:
                matching_contacts.append(contact)
        return matching_contacts


class Contact:
    def __init__(self, name, email):  # constructor
        self.name = name
        self.email = email

    # echivalentul supraincarcarii operator<< din c++
    def __repr__(self):
        return "Contact({}, {})".format(self.name, self.email)


class Agenda:
    all_contacts = ContactList()

    def add_contact(self, contact):
        self.all_contacts.append(contact)

    def print_agenda(self):
        for contact in self.all_contacts:
            print(contact)


# acest bloc se executa doar cand se apeleaza acest script
# se recomanda folosirea acestui bloc pentru a nu se executa la import
if __name__ == '__main__':
    agenda = Agenda()
    agenda.add_contact(Contact('Ion Popescu','ion_popescu@gmail.com'))
    agenda.print_agenda()
```

Exemplu de apel super

```
class Friend(Contact):
    def __init__(self, name, email, phone):
        super().__init__(name, email)
        self.phone = phone
```

Exemplu simplu de moștenire



Problema Diamant

```
class BaseClass:
    num_base_calls = 0

    def call_me(self, caller):
        print("Apel metoda din BaseClass, caller=", caller)
        self.num_base_calls += 1


class LeftSubclass(BaseClass):
    num_left_calls = 0

    def call_me(self, caller):
        print("Apel metoda din LeftSubclass, caller=", caller)
        super().call_me("LeftSubclass")
        self.num_left_calls += 1


class RightSubclass(BaseClass):
    num_right_calls = 0

    def call_me(self, caller):
        print("Apel metoda din RightSubclass, caller=", caller)
        super().call_me("RightSubclass")
        self.num_right_calls += 1
```

```
class Subclass(LeftSubclass, RightSubclass):
    num_sub_calls = 0

    def call_me(self, caller):
        print("Apel metoda din Subclass, caller=", caller)
        super().call_me("Subclass")
        super().call_me("Subclass")
        self.num_sub_calls += 1


if __name__ == '__main__':
    subclass_instance = Subclass()
    print(Subclass.__mro__)  # method resolution order
    subclass_instance.call_me("__main__")
    print(subclass_instance.num_sub_calls,
          subclass_instance.num_left_calls,
          subclass_instance.num_right_calls,
          subclass_instance.num_base_calls)
```

Exemplu de polimorfism

```
class AudioFile:
    def __init__(self, filename):
        if not filename.endswith(self.ext):
            raise Exception("Format nesuportat")
        self.filename = filename


class MP3File(AudioFile):
    ext = "mp3"

    def play(self):
        print("se canta {} un mp3".format(self.filename))


class WavFile(AudioFile):
    ext = "wav"

    def play(self):
        print("se canta {} un wav".format(self.filename))


class OggFile(AudioFile):
    ext = "ogg"

    def play(self):
        print("se canta {} un ogg".format(self.filename))


class FlacFile:
    def __init__(self, filename):
        if not filename.endswith(".flac"):
            raise Exception("Format necunoscut")
        self.filename = filename
```

```
    def play(self):
        print("se canta {} un flac".format(self.filename))


if __name__ == '__main__':
    filename = input('*.ogg file = ')
    ogg = OggFile(filename)
    ogg.play()
```

Să analizăm în continuare următoarea problemă. Trebuie modelate două categorii de elemente constitutive de granularitate mare ale corpului uman și anume mușchii și nervii. Alegerea a fost din rațiuni strict didactice și nu trebuie luată în considerare din punct de vedere al medicinei. Gruparea a fost bazată pe idea de fibre din corpul uman. Chiar dacă fibrele nervoase sunt utilizate și în controlul masei musculare ele sunt una din componentele similare cu magistralele unui sistem de calcul.

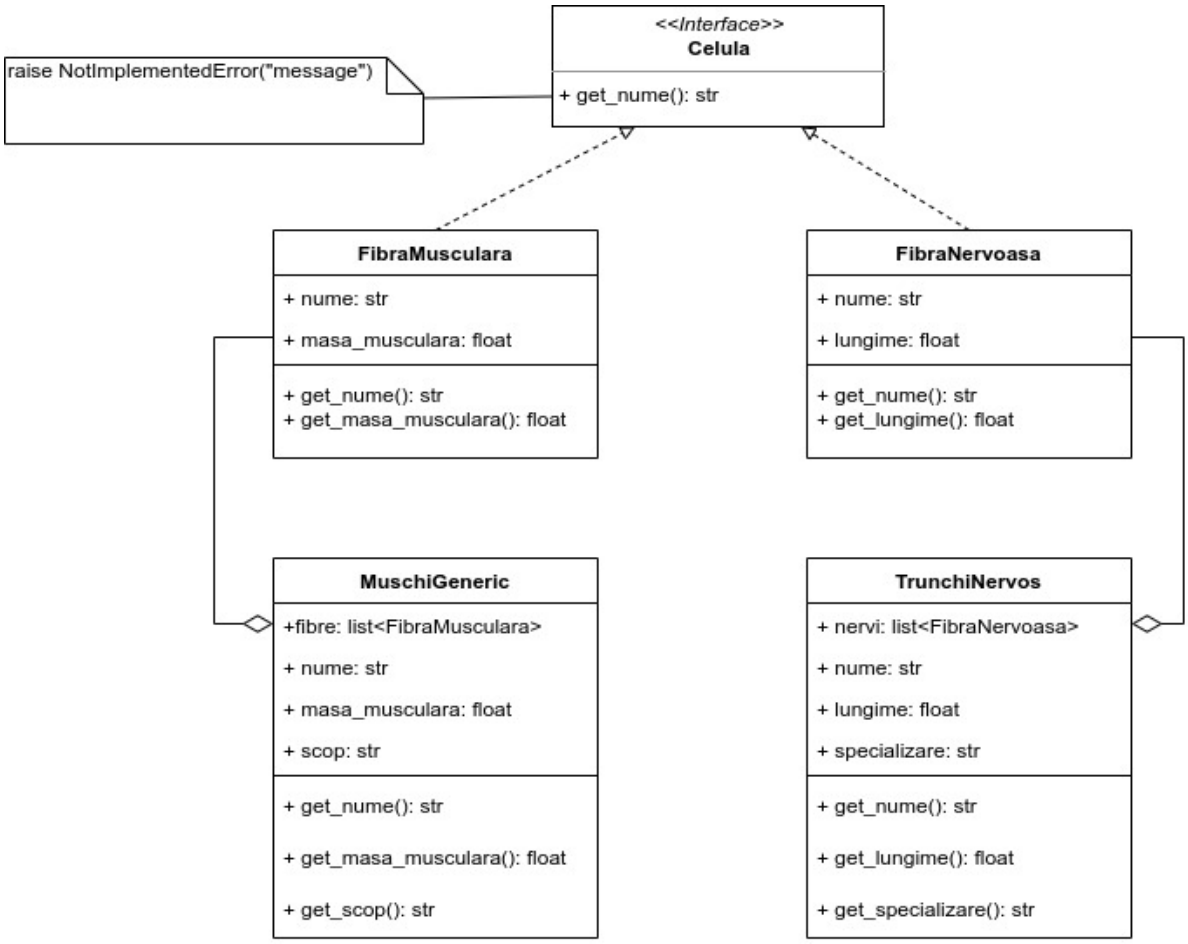


Diagrama de clase

Exemplu de output:

```
Masa musculara a muschilor [Biceps Stang] = 0.49429999999999996
Masa musculara a muschilor [Biceps Drept] = 0.3666
Masa musculara a muschilor [Triceps Stang] = 0.4447
Masa musculara a muschilor [Triceps Drept] = 0.4915999999999999
Masa musculara a muschilor [Gamba Stanga] = 0.43329999999999996
Masa musculara a muschilor [Gamba Dreapta] = 0.4665
Masa musculara a muschilor [Stomac] = 1.2517999999999996
```

```
masa totala a muschilor = 5.0489
Lungimea sistemului nervos [Emisfera Stanga] = 1439.1999999999998
Lungimea sistemului nervos [Emisfera Dreapta] = 1672.8
Lungimea sistemului nervos [Cerebel] = 1088.5
Lungimea sistemului nervos [Maduva] = 1210.9
Lungimea axonilor din sistemul nervos = 5411.4
Urmatorii muschi au functie locomotorie:
Biceps Stang ['locomotor', 'incordare']
Biceps Drept ['locomotor', 'incordare']
Triceps Stang ['locomotor', 'relaxare']
Triceps Drept ['locomotor', 'relaxare']
Gamba Stanga ['locomotor', 'incordare']
Gamba Dreapta ['locomotor', 'incordare']
```

Aplicații și teme

Aplicații de laborator:

1. Implementați o aplicație pornind de la exemplul cu apelul super (clasa Friend)
2. În cadrul problemei diamant, se cere înlocuirea apelurilor de forma:

```
super().call_me("LeftSubclass")
```

cu apeluri de forma:

```
BaseClass.call_me(self, "LeftSubclass")
```

Ce se observă?

3. Pentru exemplul cu polimorfism, se cere:
   - elaborarea diagramei de clase (în aplicația **dia** - sudo apt install dia, pe site-ul **draw.io** sau pe hârtie - se cere poza la final)
   - dezvoltarea unui program care să citească de la tastatură path-uri pentru fișiere audio, să utilizeze modulul **os** pentru a testa existența fișierelor și clasele create pentru a valida formatul (mp3, wav, ogg, flac)
4. Pentru problema cu celulele, se va porni de la această diagramă și (vai, vai :)) utilizând creion hârtie și radieră (în aplicația **dia** - sudo apt install dia, pe site-ul **draw.io** sau pe hârtie - se cere poza la final) se vor face mai întâi modificările conceptuale și abia apoi se va trece la implementarea codului

Studenții își pot alege să modifice proiectul inițial și apoi să implementeze modificările (câte una din a,b,c) din lista de mai jos:

A. Cum se modifică diagrama de clase și codul, dacă dorim să adăugăm:

- mușchii biceps de la cele două mâini (de ex.: scop mâna stângă: {"locomotor", "încordare braț stâng"}
- mușchii triceps de la cele două mâini (de ex.: scop mâna stângă: {"locomotor", "relaxare braț stâng"}
- mușchii piciorului (de ex. scop gamba stângă: {"locomotor", "mișcare gleznă stângă"}
- Trunchiuri nervoase specifice coloanei

B. Calculați:

- masa musculară a tuturor mușchilor prezenți
- lungimea tuturor nervilor prezenți

C. Afișați doar:

- mușchii care au ca scop, printre altele funcția locomotorie
- Fibrele nervoase din zona coloanei
- Se acceptă și sugestii de alte probleme (tot sub formă de UML inițial) care vor folosi moștenirea multiplă

Tema pe acasă:

Realizați o aplicație care să identifice dintr-un director dat ca parametru, categoriile de fișiere precizate mai jos, folosind frecventa caracterelor care apar în acestea:

- **text ASCII/UTF8** (frecvențe mari pentru caractere în limitele {9,10,13,32...127} și frecvențe foarte mici pentru caractere în limitele {0...8,11,12,14,15...31, 128...255})
- **text UNICODE/UTF16** (caracterul 0 apare în cel puțin 30% din tot textul)
- **binar** (frecvențele sunt distribuite oarecum uniform pe tot domeniul {0...255})

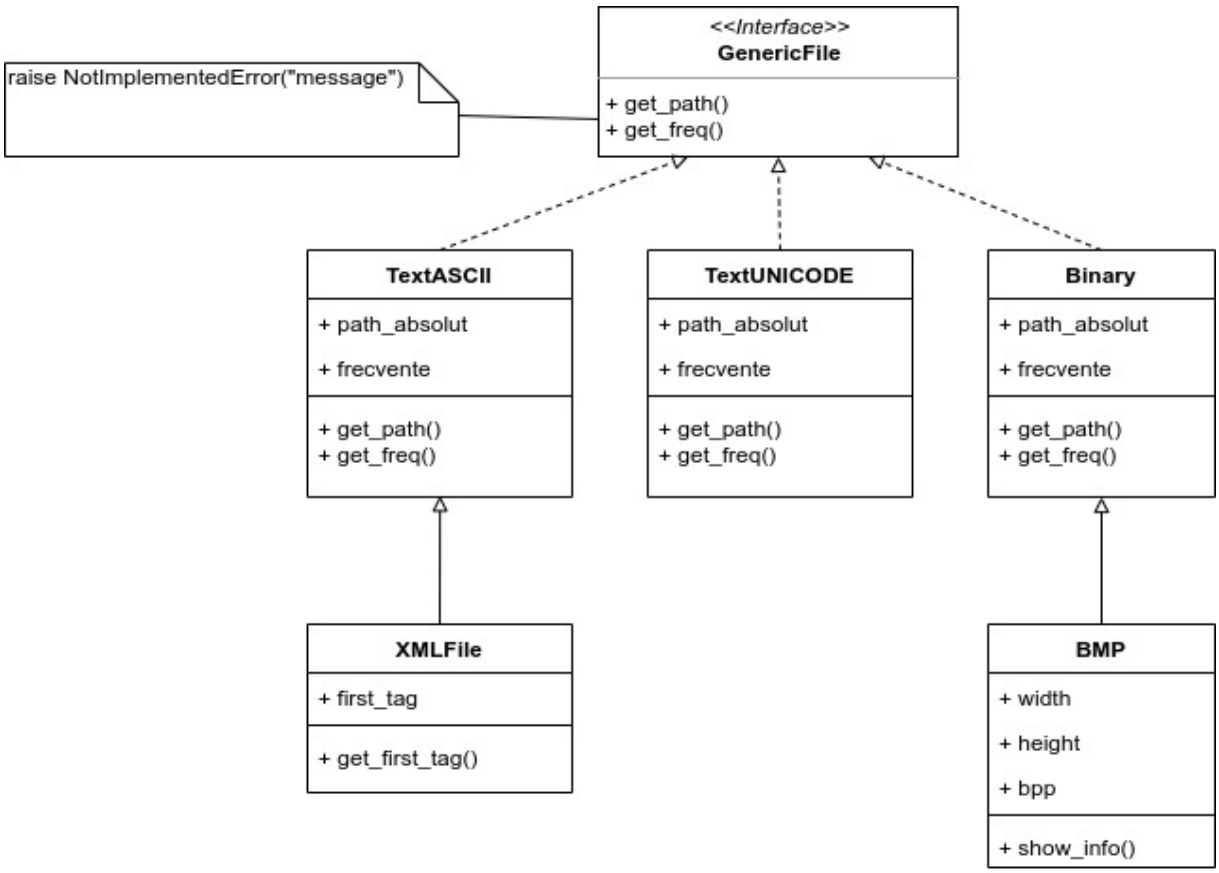


Diagrama de clase

Implementați diagrama de clase și utilizați-o pentru a rezolva următoarea problema:

Creați un script care va parcurge recursiv un director primit ca parametru și la final va afișa căile absolute ale:

- fișierelor XML ASCII
- fișierelor UNICODE
- fișierelor BMP + dimensiunea (width,height) și numărul de "bits per pixel"

Datorită mecanismului de scanare, ar trebui să puteți identifica tipul fișierului, indiferent de extensia acestuia.

Cod ajutător:

```
import os


ROOT_DIR = os.path.dirname(os.path.abspath(__file__))
for root, subdirs, files in os.walk(ROOT_DIR):
    for file in os.listdir(root):
        file_path = os.path.join(root, file)
        if os.path.isfile(file_path):
            # deschide fișierul spre acces binar
            f = open(file_path, 'rb')
            try:
                # în content se va depune o listă de octeți
                content = f.read()
                # TODO
            finally:
                f.close()
```