Laboratorul 9: Design patterns în Python

Introducere

Pentru o imagine de ansamblu asupra modelelor de proiectare, vezi:

- **„Design Patterns Elements of Reusable Object-Oriented Software”** scrisă de Erich Gamma, Richard Helm, Ralph Johnson și John Vlissides
- „Practical python design patterns: Pythonic solutions to common problems” scrisă de Wessel Badenhorst
- Cursul 9 de la disciplina *Paradigme de programare* - pentru design patterns
- Cursul 13 de la disciplina *Paradigme de programare* - pentru iterator, generator, list comprehension

Exemple

Exemplul 1: Decorator clasă (cu și fără parametri)

```
import time
import math
from datetime import datetime


class Accepts(object):
    def __init__(self, data_type):
        self.data_type = data_type

    def __call__(self, f):
        def wrapped_f(*args, **kwargs):
            for arg in args:
                if not isinstance(arg, self.data_type):
                    raise Exception(
                        "Invalid parameter {} for function {}".format(
                            arg, f.__name__))
            for key in kwargs:
                if not isinstance(kwargs[key], self.data_type):
                    raise Exception(
                        "Invalid parameter {} for function {}".format(
                            kwargs[key], f.__name__))
            return f(*args, **kwargs)

        return wrapped_f


class Returns(object):
    def __init__(self, data_type):
        self.data_type = data_type

    def __call__(self, f):
        def wrapped_f(*args, **kwargs):
            output = f(*args, **kwargs)
            if not isinstance(output, self.data_type):
                raise Exception(
                    "The function {} returned a {} type instead of
{}".format(
```

```
                        f.__name__, type(output), self.data_type))
            return output

        return wrapped_f


class TimeIt(object):
    def __init__(self, f):
        self.f = f
        self.__name__ = f.__name__

    def __call__(self, *args, **kwargs):
        begin = time.time()
        output = self.f(*args)
        end = time.time()
        print("Total time taken in : ", self.f.__name__, end - begin)
        return output


class LogIt(object):
    def __init__(self, f):
        self.f = f
        self.__name__ = f.__name__

    def __call__(self, *args, **kwargs):
        timestamp = datetime.now().timestamp()
        output = self.f(*args)
        with open('log_decorator.txt', 'a') as fp:
            fp.write("[{}]: function '{}' returned {}\n".format(
                timestamp, self.f.__name__, output))
        return output


@Accepts(int)
@Returns(int)
@LogIt
@TimeIt
def factorial(num):
    time.sleep(1)
    return math.factorial(num)


if __name__ == '__main__':
    for i in range(3, 11):
        print("factorial({})={}".format(i, factorial(i)))
```

Exemplul 2: Decorator funcție (cu și fără parametri)

```
import time
import math
from datetime import datetime


def time_it(func):
    def wrapper(*args, **kwargs):
```

```
        begin = time.time()
        output = func(*args, **kwargs)
        end = time.time()
        print("Total time taken in : ", func.__name__, end - begin)
        return output

    return wrapper


def log_it(func):
    def wrapper(*args, **kwargs):
        timestamp = datetime.now().timestamp()
        output = func(*args, **kwargs)
        with open('log_decorator.txt', 'a') as fp:
            fp.write("[{}]: function '{}' returned {}\n".format(
                timestamp, func.__name__, output))
        return output

    return wrapper


def accepts(data_type):
    def wrap(f):
        def wrapped_f(*args, **kwargs):
            for arg in args:
                if not isinstance(arg, data_type):
                    raise Exception(
                        "Invalid parameter {} for function {}".format(
                            arg, f.__name__))
            for key in kwargs:
                if not isinstance(kwargs[key], data_type):
                    raise Exception(
                        "Invalid parameter {} for function {}".format(
                            kwargs[key], f.__name__))
            return f(*args, **kwargs)

        return wrapped_f

    return wrap


def returns(data_type):
    def wrap(f):
        def wrapped_f(*args, **kwargs):
            output = f(*args, **kwargs)
            if not isinstance(output, data_type):
                raise Exception(
                    "The function {} returned a {} type instead of
{}".format(
                        f.__name__, type(output), data_type))
            return output

        return wrapped_f

    return wrap
```

```
@accepts(int)
@returns(int)
@log_it
@time_it
def factorial(num):
    time.sleep(1)
    return math.factorial(num)


if __name__ == '__main__':
    for i in range(3, 11):
        print("factorial({})={}".format(i, factorial(i)))
```

Exemplul 3: Decorator funcție (cu argumente) care simulează programarea orientată pe aspecte (AOP)

```
import types

def decorator_cu_argumente(before_fct, after_fct,
replacement_fct=None):
    def wrap(f):
        def wrapped_f(*args, **kwargs):
            if isinstance(before_fct, types.FunctionType):
                before_fct()  # execute before function
            if replacement_fct and isinstance(replacement_fct,
types.FunctionType):
                output = replacement_fct(*args, **kwargs)
            else:
                output = f(*args, **kwargs)
            if isinstance(after_fct, types.FunctionType):
                after_fct()  # execute after function
            return output
        return wrapped_f
    return wrap

def before():
    print("Before")

def after():
    print("After")

def replacement(*args, **kwargs):
    print("Replacement")
    return "replaced"

@decorator_cu_argumente(before_fct=before, after_fct=after,
replacement_fct=replacement)
def func0(name, question):
    print("func0")
    return f"Hello {name}, {question}"

@decorator_cu_argumente(before_fct=before, after_fct=after)
def func1(name):
```

```
    print("func1")
    return f"Hello {name}"

if __name__ == '__main__':
    print("Returned:", func0("Ion Popescu", "how are you?"))
    print("Returned:", func1("Ion Popescu"))
```

Exemplul 4: Iterator

Pentru a crea un iterator în Python, este necesară definirea unei clase și implementarea metodelor *__iter__()* și *__next__()*, precum și memorarea stărilor interne și generarea excepției *StopIteration* când nu mai există valori de returnat.

```
import math

class FactorialIterator:
    def __init__(self, maximum=0):
        self.maximum = maximum

    def __iter__(self):
        self.n = 2
        return self

    def __next__(self):
        if self.n <= self.maximum:
            result = math.factorial(self.n)
            self.n += 1
            return result
        else:
            raise StopIteration

if __name__ == '__main__':
    iterator = iter("car")
    print(next(iterator))
    print(next(iterator))
    print(next(iterator), "\n")

    factorial_iterator = iter(FactorialIterator(10))
    for item in factorial_iterator:
        print(item)
```

Exemplul 5: Generator

Generatoarele reprezintă o modalitate simplă de a crea iteratori, toate cerințele anterioare fiind gestionate în mod automat de generator.

Definirea unui generator presupune utilizarea cuvântului cheie *yield* în loc de *return.* Acesta trebuie să conțină cel puțin un *yield* (poate conține mai multe alte yield-uri, return-uri).

Diferența între *return* și *yield*:

- *return* - încheie complet execuția funcției
- *yield* - pune pauză funcției, salvând stările și continuând mai târziu de unde a rămas.

Avantajele utilizării unui *generator* în loc de *iterator:*

- ușor de implementat
- eficient din punct de vedere al memoriei
- permite reprezentarea unui ***stream* infinit**

- generatoarele pot fi folosite pentru a realiza un **pipeline** cu o serie de operații.

```
import math

def factorial_generator(maximum):
    for i in range(2, maximum):
        yield math.factorial(i)

if __name__ == '__main__':
    generator = factorial_generator(10)
    for item in generator:
        print(item)
```

Exemplul 6.1: Pipeline de generatoare cu sintaxă similară list comprehension

```
import re

def to_pascal_case(string):
    split_string_generator = (substr for substr in string.split(' '))
    filter_generator = (re.search(r'[a-zA-Z]+', s).group(0)
                        for s in split_string_generator
                        if re.search(r'[a-zA-Z]+', s))
    capitalize_generator = (string.capitalize()
                            for string in filter_generator)
    return ''.join(capitalize_generator)

if __name__ == '__main__':
    string = "This text!@$2.;, should be converted,&*% to pascal case"
    print(to_pascal_case(string))
```

Exemplul 6.2: Pipeline de generatoare clasic

```
def even_filter(nums):
    for num in nums:
        if num % 2 == 0:
            yield num
def multiply_by_three(nums):
    for num in nums:
        yield num * 3
def convert_to_string(nums):
    for num in nums:
        yield 'The Number: %s' % num
if __name__ == '__main__':
    nums = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
    pipeline = convert_to_string(multiply_by_three(even_filter(nums)))
    for num in pipeline:
        print(num)
```

Aplicații și teme

Aplicații de laborator:

1) Să se implementeze o aplicație care citește de la tastatură titlul unei lucrări, autorul, numărul de paragrafe și paragrafele propriu-zise, apoi creează prin intermediul modelului *fabrică de obiecte* un HTMLFile, JSONFile, sau TextFile (cu două implementări particularizate: Article și Blog). Pentru fișierul text se va utiliza modelul *prototip*, plecând de la o reprezentare default a datelor (pentru clonare se va utiliza modulul *copy* din python).

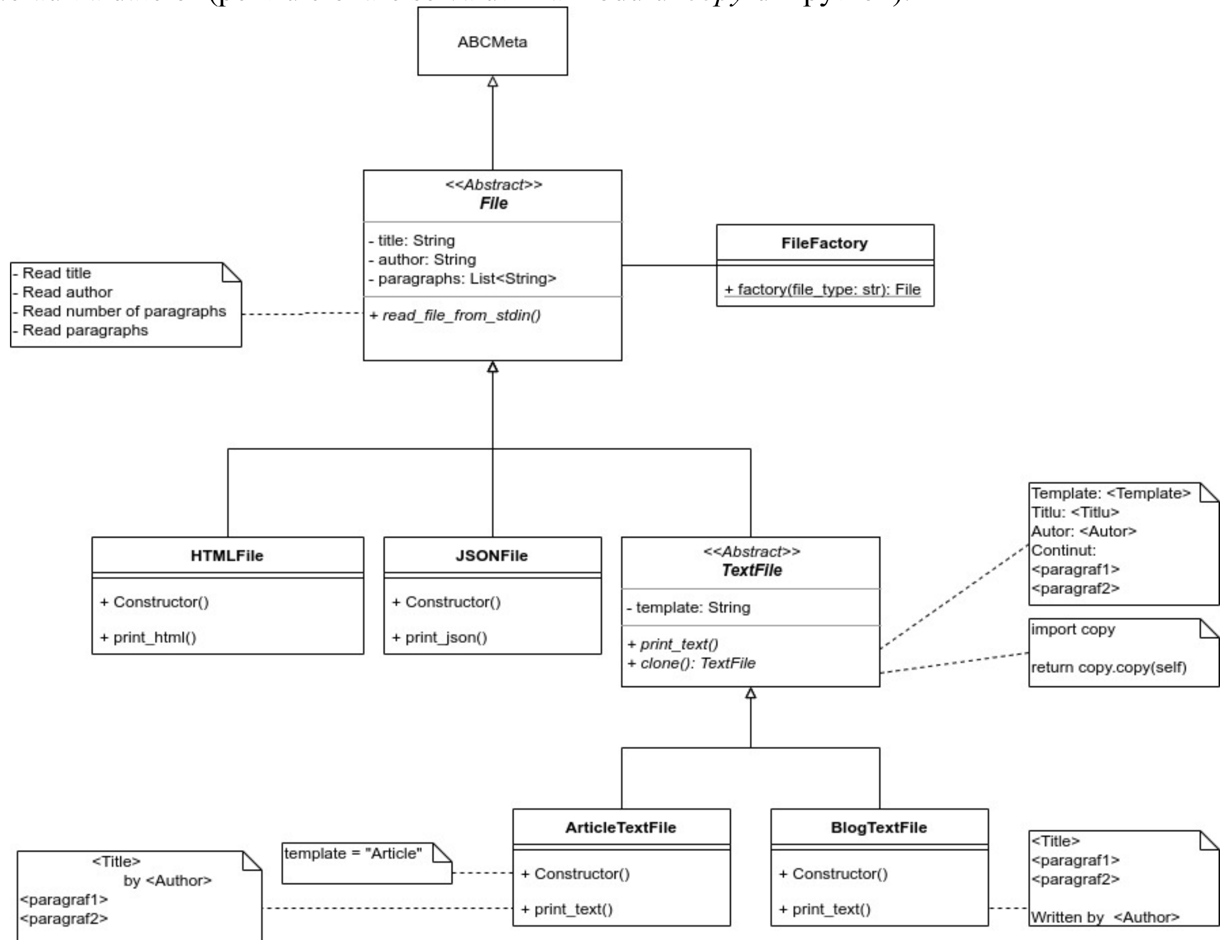

Diagrama de clase

2) Să se implementeze un pipeline de generatoare care să:

- filtreze fișierele existente dintr-o listă de path-uri
- filtreze fișierele cu extensia *.txt*
- numere liniile din fișierele text
- afișeze numele fișierului: număr de linii

Tema pe acasă:

1) Să proiecteze și să se implementeze o aplicație care primește de la tastatură un fișier fără extensie, conținutul acestuia fiind cod Kotlin / Python / Bash / Java. Aplicația va determina prin intermediul modelului ***lanț de responsabilități*** ce fel de fișier este, fiecare handler fiind capabil să determine un singur tip de fișier. În cazul în care handler-ul determină că nu poate gestiona tipul respectiv de fișier, îl va trimite mai departe în lanț. După determinarea formatului corect de fișier, prin intermediul modelului ***comandă*** se va executa conținutul fișierului (cu comanda corespunzătoare) prin intermediul unei funcții din modulul *subprocess* ce returnează output-ul generat în urma execuției.

Observație: tipul fișierului va fi determinat strict pe baza conținutului.

Hint: Se poate verifica existența unor cuvinte cheie specifice limbajului (ex.: fun. when), se poate verifica lista de importuri de la începutul fișierului, se poate verifica existența funcției main (cu forma ei specifică) sau directiva cu interpretorul introdusă prin *shebang*.

2) Să se implementeze un automat de sucuri prin intermediul modelului *automat finit de stări*. Automatul este alcătuit practic din mai multe automate, gestionate de o entitate centrală:

- Automatul pentru introducerea banilor: *TakeMoneySTM*
- Automatul pentru selectarea produsului: *SelectProductSTM*
- Entitatea centrală: *VendingMachineSTM*

De asemenea, se va utiliza modelul *observator* pentru:

- afișarea unui mesaj la consolă cu suma de bani introdusă de fiecare dată când aceasta este actualizată (TakeMoneySTM)
- a anunța *VendingMachineSTM* că a fost selectat un produs și trebuie validată tranzacția, după care se poate opta pentru returnarea restului sau selectarea altui produs.

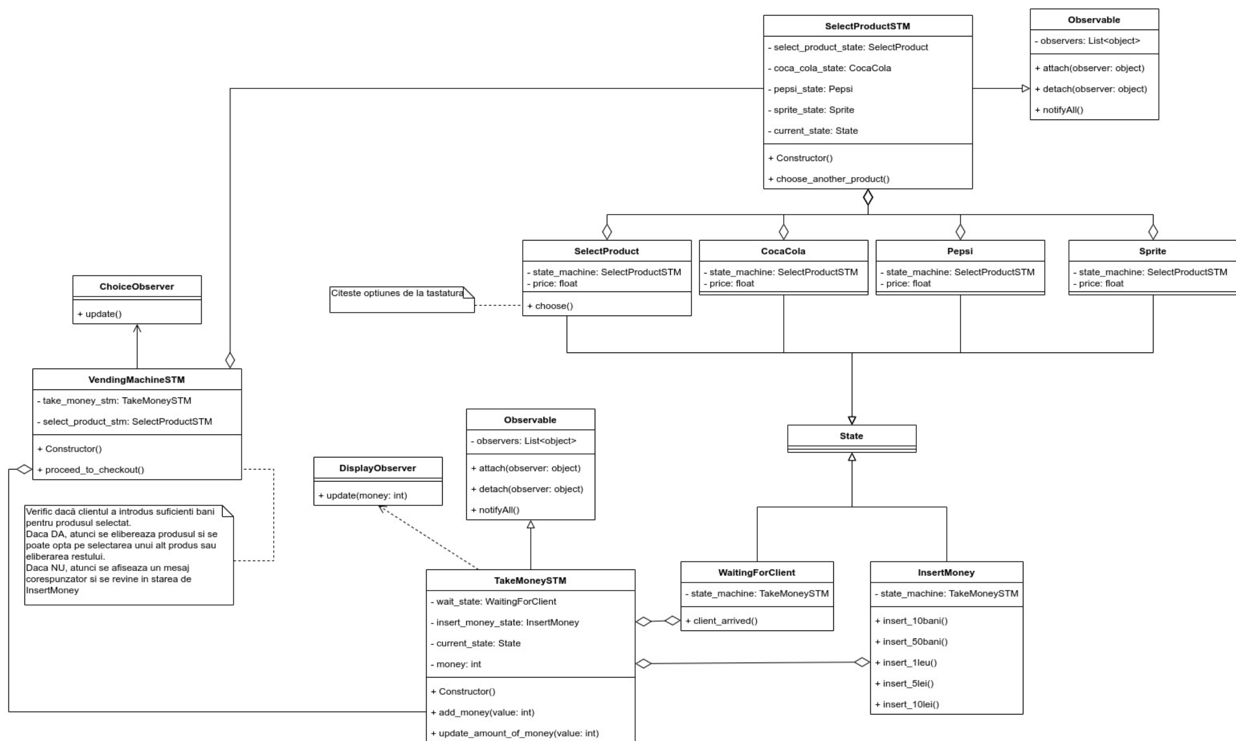

Diagrama de clase

3) Să se proiecteze și să se implementeze o aplicație care realizează prin intermediul modulului *requests* cereri HTTP de tip GET către diverse URL-uri. Se va aplica modelul *intermediar (proxy)* peste cererea GET pentru a realiza un mecanism de caching care să stocheze

într-un fișier text URL-ul la care s-a făcut cererea, timestamp-ul, precum și răspunsul returnat. La fiecare cerere, se va verifica întâi cache-ul: dacă conține URL-ul respectiv, iar timestamp-ul nu a depășit o oră de la momentul curent, se va returna răspunsul din cache. În caz contrar, dacă a expirat intrarea din cache, se va realiza din nou cererea și va fi actualizată intrarea din cache. Dacă nu exista în cache, se va realiza cererea și se va adăuga în cache răspunsul.

[BONUS]: Utilizând modelul *strategy* (combinat cu modelul *proxy*), să se creeze un mecanism de load balancing care monitorizează numărul de cereri sosit într-o anumită cuantă de timp. Dacă numărul de cereri a crescut de 10 ori în acea cuantă de timp, aplicația se va replica, creând un nou proces care să gestioneze jumătate din cereri.