Министерство науки и высшего образования Российской Федерации Федеральное государственное бюджетное образовательное учреждение высшего образования «Московский государственный технический университет имени Н.Э. Баумана (национальный исследовательский университет)» (МГТУ им. Н.Э. Баумана)

Факультет «Информатика и системы управления»

Кафедра ИУ5 «Системы обработки информации и управления»

Лабораторная работа №4

по дисциплине «Базовые компоненты интернет-технологий»

Выполнил:
студент группы ИУ5-33Б
Слоква А. В.

Проверил:
Канев А.И.

2021 г.

Постановка задачи:

Задание:

- Необходимо для произвольной предметной области реализовать от одного до трех шаблонов проектирования: один порождающий, один структурный и один поведенческий. В качестве справочника шаблонов можно использовать следующий каталог. Для сдачи лабораторной работы в минимальном варианте достаточно реализовать один паттерн.
- Вместо реализации паттерна Вы можете написать тесты для своей программы решения биквадратного уравнения. В этом случае, возможно, Вам потребуется доработать программу решения биквадратного уравнения, чтобы она была пригодна для модульного тестирования.
- В модульных тестах необходимо применить следующие технологии:
    - TDD - фреймворк.
    - BDD - фреймворк.
    - Создание Mock-объектов.

Текст программы:

main.py

```
import os
import sys
from enum import Enum

from decorator.main import Pizza, PizzaWithTomato, PizzaWithSalt, show
from factory_method.main import OperatorMax, call, OperatorVova
from singleton.main import DB


class Operator(Enum):
    MAX = "max"
    VOVA = "vova"


class PizzaTypes(Enum):
    MAX = "max"
    VOVA = "vova"


def main(args) -> int:
    # logging.basicConfig(filename="sample.log", level=logging.INFO) # todo:
set log level from other file

    db = DB("my dsn from env")
    db2 = DB("bad move to create new db")
    if db == db2:
        print("db and db 2 are singe objects")
    # now create operator of pizza order
    if args[0] == Operator.MAX.value:
```

```
        print("our operator is max, lets call him")
        call(OperatorMax())
        print("\n")
    elif args[0] == Operator.VOVA.value:
        print("our operator is vova, lets call him")
        call(OperatorVova())
        print("\n")
    else:
        print("invalid operator:{}".format(args[0]))
        return 2
    # now create pizza
    simple = Pizza()
    for i in args[1:]:
        if i == "tomato":
            simple = PizzaWithTomato(simple)
        elif i == "salt":
            simple = PizzaWithSalt(simple)
        else:
            print("unknown ingredient {}".format(i))
    print("Pizza Maker: Now I've got a decorated component:")
    show(simple)
    print("\n")
    db.exec("INSERT INTO TABLE pizza_orders VALUES(...)")
    return 1


if __name__ == "__main__":
    if len(sys.argv) <= 1:
        sys.exit(2)
    sys.exit(main(sys.argv[1:]))
```

main_test.py

```
import unittest
from unittest.mock import patch, Mock

from .factory_method.main import OperatorMax


class TestOperatorMax(unittest.TestCase):

    @patch.object(OperatorMax, 'eat')
    def test_work_1(self, mock_eat):
        mock_eat.return_value = True
        operator_max = OperatorMax()
        result = operator_max.work()
        self.assertEqual('Work is done', result)
```

decorator -> main.py

```
class Pizza():
    def operation(self) -> str:
        pass


class Pizza(Pizza):
    def operation(self) -> str:
        return "Pizza"
```

```
class Decorator(Pizza):
    _component: Pizza = None

    def __init__(self, component: Pizza) -> None:
        self._component = component

    @property
    def component(self) -> str:
        return self._component

    def operation(self) -> str:
        return self._component.operation()


class PizzaWithTomato(Decorator):
    def operation(self) -> str:
        return f"PizzaWithTomato({self.component.operation()})"


class PizzaWithSalt(Decorator):
    def operation(self) -> str:
        return f"PizzaWithSalt({self.component.operation()})"


def show(component: Pizza) -> None:
    print(f"RESULT: {component.operation()}", end="")


if __name__ == "__main__":
    # Таким образом, клиентский код может поддерживать как простые
компоненты...
    simple = Pizza()
    print("Client: I've got a simple component:")
    show(simple)
    print("\n")

    # ...так и декорированные.
    #
    # Обратите внимание, что декораторы могут обёртывать не только простые
    # компоненты, но и другие декораторы.
    decorator1 = PizzaWithTomato(simple)
    decorator2 = PizzaWithSalt(decorator1)
    print("Client: Now I've got a decorated component:")
    show(decorator2)
```

factory_method -> main.py

```
from __future__ import annotations

import time
from abc import ABC, abstractmethod


class Operator(ABC):

    normal_sleep_time = 8*60*60

    @abstractmethod
    def factory_method(self):
        pass
```

```
    def some_operation(self) -> str:
        product = self.factory_method()
        result = f"PIZZA OPERATOR: Hi, your pizza at work! The same creator's
code has just worked with {product.operation()}"

        return result

    def can_repeat(self, sleep_time: int) -> bool:
        return sleep_time >= self.normal_sleep_time
    def work(self) -> None:
        print("Working")
        if self.eat():
            print("im eating")
        print("working")
    def eat(self) -> bool:
        time.sleep(30*100)
        return True

class OperatorMax(Operator):
    def factory_method(self) -> Product:
        return ConcreteProduct1()


class OperatorVova(Operator):
    def factory_method(self) -> Product:
        return ConcreteProduct2()


class Product(ABC):
    @abstractmethod
    def operation(self) -> str:
        pass


class ConcreteProduct1(Product):
    def operation(self) -> str:
        return "{Result of the ConcreteProduct1 from MAX}"


class ConcreteProduct2(Product):
    def operation(self) -> str:
        return "{Result of the ConcreteProduct2 from VOVA}"


def call(creator: Operator) -> None:
    print(f"Client: I'm not aware of the creator's class, but it still
works.\n"
          f"{creator.some_operation()}", end="")


if __name__ == "__main__":
    call(OperatorMax())
    print("\n")
    call(OperatorVova())
```

singleton -> main.py

```
class DBMeta(type):
    _instances = {}

    def __call__(cls, *args, **kwargs):
        if cls not in cls._instances:
```

```
            instance = super().__call__(*args, **kwargs)
            cls._instances[cls] = instance
        return cls._instances[cls]


class DB(metaclass=DBMeta):
    def __init__(self, dsn):
        return

    @staticmethod
    def exec(query: str):
        return
```

Использование BDD фреймворка

features -> steps -> test_steps.py

```
# -- FILE: features/steps/tests_steps.py
from behave import given, when, then, step

DATABASE = None  # global db connection


@given('create db connection one')
def step_impl(context):
    DATABASE = DB("test")
    pass


@when('we implement {number:d} connections')
def step_impl(context, number):  # -- NOTE: number is converted into integer
    assert number > 1 or number == 0
    context.tests_count = number
    for i in range(1, number):
        DATABASE = DB("test")
        assert id(DB("dsn")) == id(DATABASE)


@then('now behave will test them for us!')
def step_impl(context):
    assert context.failed is False
    assert context.tests_count >= 0


@given('pizza with ingridients')
def step_impl(context):
    context.pizza = Pizza
    pass


@when('we implement pizza')
def step_impl(context):  # -- NOTE: number is converted into integer
    context.pizza = PizzaWithSalt(PizzaWithTomato(context.pizza))
    pass


@then('Pizza is ok?')
def step_impl(context):
```

```
    assert context.failed is False


'''
THIS WAS BAD SOLUTION BUT THERE ARE STUPID PY PACKAGES I HATE IT
'''


class DBMeta(type):
    _instances = {}

    def __call__(cls, *args, **kwargs):
        if cls not in cls._instances:
            instance = super().__call__(*args, **kwargs)
            cls._instances[cls] = instance
        return cls._instances[cls]


class DB(metaclass=DBMeta):
    def __init__(self, dsn):
        return

    @staticmethod
    def exec(query: str):
        return


class Pizza():
    def operation(self) -> str:
        pass


class Pizza(Pizza):
    def operation(self) -> str:
        return "Pizza"


class Decorator(Pizza):
    _component: Pizza = None

    def __init__(self, component: Pizza) -> None:
        self._component = component

    @property
    def component(self) -> str:
        return self._component

    def operation(self) -> str:
        return self._component.operation()


class PizzaWithTomato(Decorator):
    def operation(self) -> str:
        return f"PizzaWithTomato({self.component.operation()})"


class PizzaWithSalt(Decorator):
    def operation(self) -> str:
        return f"PizzaWithSalt({self.component.operation()})"


def show(component: Pizza) -> None:
    print(f"RESULT: {component.operation()}", end=""
```