*Федеральное государственное бюджетное образовательное учреждение* *высшего профессионального образования*

|  |  |
| --- | --- |
|  | ***«Московский государственный технический университет  имени Н.Э. Баумана»***  ***(МГТУ им. Н.Э. Баумана)*** |

ФАКУЛЬТЕТ Информатика и системы управления

КАФЕДРА Системы обработки информации и управления (ИУ5)

**Отчет**

**по лабораторной работе №4**

**Дисциплина: Разработка Интернет-Приложений**

Студент гр. ИУ5-53Б  **\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_** Зудин А.М.

(Подпись, дата) (И.О. Фамилия)

Преподаватель  **\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_** Гапанюк Ю.Е.

(Подпись, дата) (И.О. Фамилия)

Москва, 2020

1. **Цель работы**

**Цель лабораторной работы:** изучение реализации шаблонов проектирования и возможностей модульного тестирования в языке Python.

1. **Задание**
2. Необходимо для произвольной предметной области реализовать три шаблона проектирования: один порождающий, один структурный и один поведенческий. В качестве справочника шаблонов можно использовать следующий каталог.
3. Для каждой реализации шаблона необходимо написать модульный тест. В модульных тестах необходимо применить следующие технологии:

* TDD - фреймворк.
* BDD - фреймворк.
* Создание Mock-объектов.

1. **Код программы**

factorymethod.py

from \_\_future\_\_ import annotations

from abc import ABC, abstractmethod

"пораждающий паттерн"

class Deliver(ABC):

    @abstractmethod

    def devlivery\_method(self):

        pass

    def some\_operation(self) -> str:

        # Вызываем фабричный метод, чтобы получить объект-продукт.

        delivery = self.devlivery\_method()

        if type(delivery) == str:

            return f"DELIVER: {delivery}"

        else:

            result = f"DELIVER: The product {delivery.operation()}"

        return result

class PlaneDelivery(Deliver):

    dictofweather = {"Очень плохая погода": 0, "Плохая погода": 1, "средняя": 2, "нормальная": 3, "Хорошая": 4}

    def \_\_init\_\_(self):

        self.\_fuel = 0

        self.\_weather = None

    def prepareflight(self, fueltoadd, weather):

        self.refueling(fueltoadd)

        self.updateweather(weather)

    def updateweather(self, weather):

        if weather not in self.dictofweather.keys():

            print("Pls give right weather status")

        else:

            self.\_weather = weather

    def refueling(self, count):

        self.\_fuel += count

    def isfuel(self):

        if self.\_fuel >= 10:

            self.\_fuel -= 10

            return True

        else:

            return False

    def checkweather(self):

        if self.\_weather == None:

            return 0

        if self.dictofweather[self.\_weather] >= 2:

            return 1

        else:

            return -1

    def devlivery\_method(self):

        if self.isfuel():

            if self.checkweather() > 0:

                return Coal()

            elif self.checkweather() == -1:

                return "Sorry, can't deliver your product due to weahter."

            else:

                return "Pleas contact to dispatcher and update weather status"

        else:

            return "delivering can't be done no fuel."

class CarDelivery(Deliver):

    def \_\_init\_\_(self):

        self.\_fuel = 0

    def isfuel(self):

        if self.\_fuel >= 5:

            self.\_fuel -= 5

            return True

        else:

            return False

    def refueling(self, count):

        self.\_fuel += count

    def devlivery\_method(self):

        if self.isfuel():

            return Wood()

        else:

            return "delivering can't be done no fuel."

class Product(ABC):

    @abstractmethod

    def operation(self) -> str:

        pass

class Coal(Product):

    def operation(self) -> str:

        return "Coal are delivered"

class Wood(Product):

    def operation(self) -> str:

        return "Wood are delivered"

def client\_code(deliver: Deliver) -> None:

    print(f"Client: I'm don't now who is deliver exactly.\n"

          f"{deliver.some\_operation()}")

if \_\_name\_\_ == "\_\_main\_\_":

    Plane = PlaneDelivery()

    car = CarDelivery()

    print("Launched with the car.")

    client\_code(car)

    print("")

    print("Launched with the Plane.")

    client\_code(Plane)

    Plane.refueling(100)

    Plane.updateweather("Хорошая")

    client\_code(Plane)

proxy.py

from abc import ABC, abstractmethod

from contextlib import contextmanager

from datetime import datetime

from time import sleep

allowusers = ['maxzbox', "andrey-kireev", "valerdon"]

@contextmanager

def workwithfile():

    a = open("../log", 'a')

    yield a

    a.close()

class Subject(ABC):

    @abstractmethod

    def request(self, login) -> None:

        pass

class RealSubject(Subject):

    def request(self, login, request):

        print("RealSubject: Handling request.")

        return ("some data")

class Proxy(Subject):

    def \_\_init\_\_(self, real\_subject: RealSubject) -> None:

        self.\_real\_subject = real\_subject

    def request(self, login, request) -> None:

        if self.check\_access(login):

            self.log\_access(f"User {login} Logging successfully request: {request}")

            return(self.\_real\_subject.request(login, request))

        else:

            self.log\_access(f"No rights to enter, attempted by {login} with request: {request}")

            print("no permission")

            return None

    def check\_access(self, login) -> bool:

        print("Proxy: Checking access prior to firing a real request.")

        if (login in allowusers):

            return True

        else:

            return False

    @staticmethod

    def log\_access(massage) -> None:

        with workwithfile() as f:

            f.write(f"{datetime.isoformat(datetime.now())}- {massage}\n")

def client\_code(subject: Subject) -> None:

    subject.request('user', "select \* from passwd")

    sleep(0.2)

if \_\_name\_\_ == “\_\_main\_\_”:

    print(“Client: Executing the client code with a real subject:”)

    real\_subject = RealSubject()

    client\_code(real\_subject)

    print(“”)

    print(“Client: Executing the same client code with a proxy:”)

    proxy = Proxy(real\_subject)

    client\_code(proxy)

strategy.py

from \_\_future\_\_ import annotations

from abc import ABC, abstractmethod

trace1 = [‘A’, ‘B’, ‘C’, ‘D’, ‘E’, ‘F’]

trace2 = [‘L’, ‘E12’, ‘D16’, ‘C’, ‘D’, ‘E’, ‘F’]

placebyindex = {‘A’: 0, ‘B’: 1, ‘C’ :2, ‘D’: 3, ‘E’: 4, ‘F’: 5}

Busess = {“busA”: trace1, “busB”: trace2}

class Navigator():

    def \_\_init\_\_(self, strategy: Strategy) -> None:

        «»»

        Обычно Контекст принимает стратегию через конструктор, а также

        предоставляет сеттер для её изменения во время выполнения.

        «»»

        self.\_strategy = strategy

    @property

    def strategy(self) -> Strategy:

        return self.\_strategy

    @strategy.setter

    def strategy(self, strategy: Strategy) -> None:

        self.\_strategy = strategy

    def do\_some\_business\_logic(self, start, end):

        result = self.\_strategy.do\_algorithm(start, end)

        return result

class Strategy(ABC):

    @abstractmethod

    def do\_algorithm(self, startpoint, pointend):

        pass

class Strategypublictransport(Strategy):

    def do\_algorithm(self, pointstart, pointend):

        a = [I for I in Busess if pointstart in Busess[i] and pointend in Busess[i]]

        if len(a) > 0:

            print(f”trace from {pointstart} to {pointend} could be completed by public transport with such buses:{a}”)

            return a

        else:

            print(“no trace by public transport”)

            return None

class StrategybyFoot(Strategy):

    def \_\_init\_\_(self):

        self.\_matrix = [0] \* 6

        self.\_matrix[0] = 1

        self.\_matrix[1] = 2

        self.\_matrix[2] = 3

    def do\_algorithm(self, startpoint, endpoint):

        if startpoint in placebyindex.keys() and endpoint in placebyindex.keys():

            index1 = placebyindex[startpoint]

            index2 = placebyindex[endpoint]

            if self.\_matrix[index1] == index2:

                print(f”you can go by foot from place {startpoint } to {endpoint} by foot”)

                return f”you can go by foot from place {startpoint } to {endpoint} by foot”

            else:

                print(“no way by Foot”)

                return “no way by Foot”

        else:

            print(“Sorry no place in map”)

            return None

if \_\_name\_\_ == “\_\_main\_\_”:

    # Клиентский код выбирает конкретную стратегию и передаёт её в контекст.

    # Клиент должен знать о различиях между стратегиями, чтобы сделать

    # правильный выбор.

    Context = Navigator(Strategypublictransport())

    print(“Client: Strategy is set to public transport.”)

    context.do\_some\_business\_logic(‘A’, ‘B’)

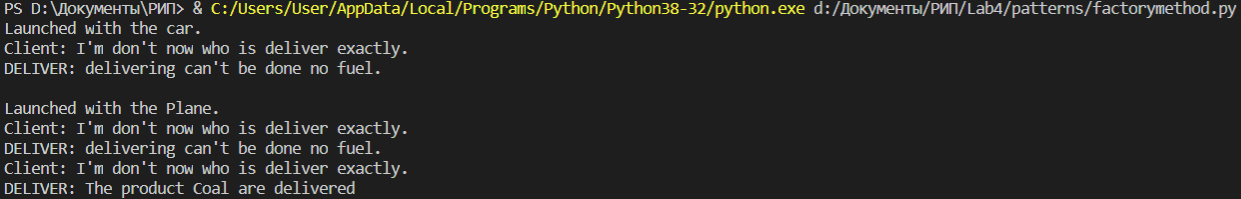
    print()

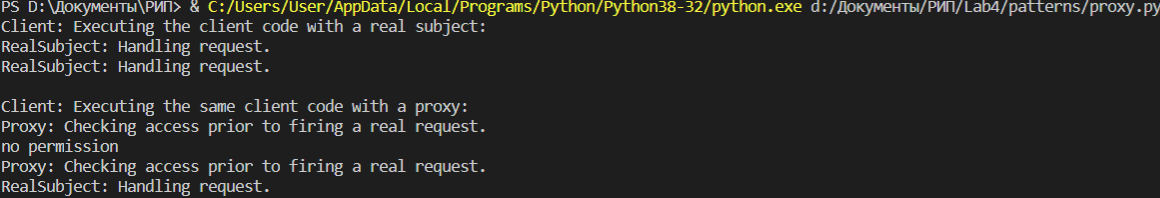
    print(“Client: Strategy is set to reverse sorting.”)

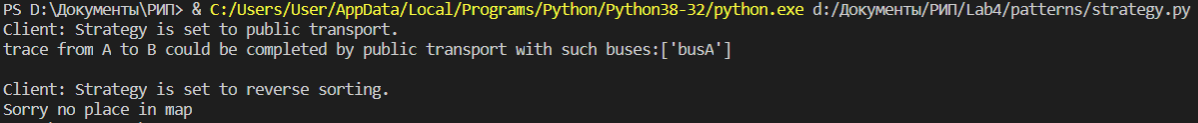
    context.strategy = StrategybyFoot()

    context.do\_some\_business\_logic(‘A’, ‘R’)

1. **Результат выполнения**

****

****

****