Федеральное государственное бюджетное образовательное учреждение
высшего образования
«АЛТАЙСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ»

Архитектура программных систем

“Поездка до аэропорта”

Исполнитель:
студент гр. № 474б
Сердюк Егор

2020 г.

Постановка задачи

Егор собирался полететь к своим друзьям в Москву. Он купил заранее билеты, собрал вещи и ждал наступления дня вылета. В день вылета, Егор задумался, на каком транспорте будет быстрее и выгоднее добраться до аэропорта? При этом, он сам не хочет сильно тратиться или долго ехать. Он может взять свой велосипед и ехать по велодорожкам определенный промежуток времени, поехать на каршеринге, поехать на автобусе (хоть и экономично, но долго) или, все же, вызвать такси. Давайте посмотрим и определим более оптимальный (или неоптимальный) вариант, учитывая, что везде плата за проезд – по времени.

Описание примененных паттернов

1. Стратегия: поведенческий паттерн проектирования, который определяет семейство схожих алгоритмов и помещает каждый из них в собственный класс, после чего алгоритмы можно взаимозаменять прямо во время исполнения программы.
2. Декоратор: структурный паттерн проектирования, который позволяет динамически добавлять объектам новую функциональность, оборачивая их в полезные «обёртки».
3. Одиночка: порождающий паттерн проектирования, который гарантирует, что у класса есть только один экземпляр, и предоставляет к нему глобальную точку доступа.

Вывод

Мы предоставили выбор, по которому Егор должен выбрать оптимальный вариант по времени и финансовым затратам, опираясь на фиксированные цены, которые хранятся в одном неизменяемом объекте, на проезды по таким путям.

Приложение

```
from __future__ import annotations
from abc import ABC, ABCMeta, abstractmethod
from typing import Dict, List
from random import randint, seed

# Strategy implementation
class RouteContext():
    """
    Контекст, определяющий интерфейс
    """
```

```
    def __init__(self, strategy: RouteStrategy):
        """
        Принятие стратегии действий через конструктор
        """
        self._strategy = strategy

    @property
    def strategy(self):
        """
        Сохранение ссылки на один из объектов стратегии. Контекст должен работать
 со всеми стратегиями
        """
        return self._strategy

    @strategy.setter
    def strategy(self, strategy: RouteStrategy):
        """
        Возможность замены объекта стратегии во время выполнения
        """
        self._strategy = strategy

    def set_data(self):
        """
        Установка исходных данных, длина пути в минутах
        """
        seed(1)  # Если требуется изменить тестовые данные, можно изменить сид
        data = {'Bicycle': randint(60, 120),'Car': randint(10, 40),'Taxi': randin
t(20, 60),'Bus': randint(60, 180)}
        self.data = data
        return data

    def out_data(self):
        return self.data

    def logic(self):
        """
        Контекст делегирует некоторую работу объекту стратегии
        """
        result = self._strategy.analyze(self.data)
        self.result = result
        return result

    def print_result(self):
        for i in self.result:
            print(f'{i[0]} on time is around {i[1]} minutes')

class RouteStrategy(ABC):
    """
    Интерфейс стратегии, который объявляет операции для всех поддерживаемых верси
й некоторого алгоритма
```

```
    Контекст использует такой интерфейс для вызова алгоритма, определенного конкр
етными стратегиями
    """
    @abstractmethod
    def analyze(self, data: Dict):
        pass

"""
Конкретные стратегии реализуют алгоритм, следуя базовому интерфейсу стратегии
(и этот интерфейс делает стратегии взаимозаменяемыми в контексте)
"""

class Normal(RouteStrategy):
    def analyze(self, data: Dict):
        return data.items()

class Fastest(RouteStrategy):
    """
    Поиск быстрого пути
    """
    def analyze(self, data: Dict):
        return sorted(data.items(), key=lambda x: x[1])

class Slowest(RouteStrategy):
    """
    Поиск долгого пути
    """
    def analyze(self, data: Dict):
        return sorted(data.items(), key=lambda x: x[1], reverse=True)

# Decorator implementation
class Price(object):
    """
    Интерфейс, реализующий декоратор
    """
    __metaclass__ = ABCMeta

    @abstractmethod
    def operator(self):
        pass


class Component(Price):
    """
    Компонент программы, принимающий данные
    """
    def operator(self, data):
        return data

class EndPrice(Price):
    """
```

```
    Декоратор, считающий итоговый ценник поездки из данных компонента
    """
    def __init__(self, obj):
        self.obj = obj

    def operator(self, data: Dict):
        _data = {key: self.obj[key] * data[key] for key in self.obj}
        return _data

class OutPrice(Price):
    """
    Дополнение для вывода данных, использует структуру декоратора, но ничего не и
зменяет
    """
    def __init__(self, obj):
        self.obj = obj

    def operator(self):
        for i in self.obj.items():
            print(f'{i[0]} will cost around {i[1]} rubles')



# Singleton implementation (through metaclasses)
class DataView(type):
    """
    Реализация одиночки через метакласс
    """
    _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 DataSave(metaclass=DataView):
    """
    Логика одиночки - выдача данных
    """
    def __init__(self, data: List):
        self.data = data

    def get_data(self):
        return self.data

if __name__ == "__main__":
    context = RouteContext(Fastest())
    print("\nClient: Strategy is set to the fastest sorting\n")
    context.set_data()
```

```
context.logic()
context.print_result()

print("\nClient: Strategy is set to the slowest sorting\n")
context.strategy = Slowest()
context.logic()
context.print_result()
print()

data = context.out_data()

price = {'Bicycle': 0,'Car': 3,'Taxi': 5,'Bus': 1}

print("\nClient: Save actual price data in the singleton\n")
data_store_1 = DataSave(price)
data_store_2 = DataSave(price)

if data_store_1 == data_store_2: print("The data was saved successfully")
else: print("Something went wrong")

decorator = Component()
decorator = decorator.operator(data_store_1.get_data())
decorator = EndPrice(decorator)
decorator = decorator.operator(data)
decorator = OutPrice(decorator)
decorator = decorator.operator()
```