НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ ИТМО
Факультет инфокоммуникационных технологий
Инфокоммуникационные технологии и системы связи

Домашнее задание

«Разработка программного прототипа
по проекту инфокоммуникационной системы»

Выполнил:
Савчук А. А.
Группа К4112с

Проверил:
Марченко Е. В.

Санкт-Петербург
2024

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

Цель работы – реализовать классы разработанной диаграммы классов, применяя шаблоны GoF.

Вариант задания

Электронная система продажи билетов на междугородние маршруты.

Описание инфокоммуникационной системы

Платформа для продажи электронных билетов на междугородние автобусные поездки и получения онлайн-платежей за проезд. Покупатель самостоятельно распечатывает билеты для предъявления перед отправкой. Оплатить билет можно из-за рубежа РФ. Доступна покупка поездок «туда-обратно», включая пересадки и использование абонементов. Обновление таблиц в режиме реального времени.

Реализация проездных документов для людей с ограниченными возможностями не требует посещения кассы: средства можно перевести на выбор через SMS, электронные кошельки или банковские карты. Данные электронных расчетов интегрированы с бухгалтерией компании.

1. Шаблон итератор

Предлагается использовать шаблон итератор для ленивой загрузки данных из хранилища маршрутов. Такой подход избежать нехватки памяти при большом количестве маршрутов, организовать пагинацию для страницы маршрутов. Текст программы приведен в приложении А.

2. Шаблон декоратор

Предлагается применить шаблон декоратор для сбора метрик о времени работы отдельных функций. Это позволит оценивать производительность системы без изменения сигнатур или исходного кода этих функций. Текст программы приведен в приложении А.

3. Шаблоны строитель и стратегия

Предлагается использовать шаблон строить для формирования заявки на покупку билета, которая содержит много данных: пассажиры, маршрут, метод оплаты и т. д. Этот шаблон помогает структурировать и поэтапно создавать такие объекты. Это повышает читаемость и удобство поддержки кода, делая процесс создания сложных объектов более управляемым и гибким.

Также предлагается использовать шаблон стратегия для реализации различных способов оплаты билета. Каждый способ оплаты может быть реализован как отдельная стратегия, что позволяет легко изменять или добавлять новые способы оплаты без необходимости модификации основной логики покупки билета. Текст программы приведен в приложении А.

ЗАКЛЮЧЕНИЕ

В рамках данной работы для разработанной ранее диаграммы классы были реализованы классы с применением шаблонов проектирования GoF (Gang of Four), что позволило улучшить архитектуру системы, обеспечив гибкость, расширяемость и повторное использование компонентов.

Приложения А. Текст программы

Листинг 1 – iterator.py

```python
from dataclasses import dataclass
from datetime import datetime


@dataclass
class Route:
    id: int
    origin: str
    destination: str
    time: datetime


class RoutesRepoIterator:
    def __init__(self, routes, index) -> None:
        self.routes = routes
        self.index = index

    def __iter__(self):
        return self

    def __next__(self):
        if self.index == len(self.routes):
            raise StopIteration
        self.index += 1
        return self.routes[self.index - 1]


class RoutesRepo:
    def __init__(self):
        self.routes = [
            Route(1, "New York City", "Boston", datetime.now()),
            Route(2, "Los Angeles", "San Francisco", datetime.now()),
            Route(3, "San Diego", "Los Angeles", datetime.now())
        ]

    def __iter__(self):
        return RoutesRepoIterator(self.routes, 0)


repo = RoutesRepo()

for route in repo:
    print(route)
```

Листинг 2 – decorator.py

```python
from time import time, sleep
from random import randint
from numpy import mean

metrics_storage = {}


```

```python
def measure_execution_time(f):
    def inner(*args, **kwargs):
        start = time()
        result = f(*args, **kwargs)
        end = time()

        if f.__name__ not in metrics_storage:
            metrics_storage[f.__name__] = []
        metrics_storage[f.__name__].append(end - start)

        return result

    inner.__name__ = f.__name__

    return inner


def some_function():
    sleep(randint(0, 1))


some_function = measure_execution_time(some_function)

for _ in range(10):
    some_function()

metrics = metrics_storage[some_function.__name__]

print(metrics)
print(mean(metrics))
```

Листинг 3 – builder+strategy.py

```python
from dataclasses import dataclass
from datetime import datetime
from abc import ABC, abstractmethod


@dataclass
class Route:
    id: int
    origin: str
    destination: str
    time: datetime
    price: int


@dataclass
class Passenger:
    id: int
    name: str


@dataclass
class User:
    id: int
    email: str

```

```python

class PaymentMethod(ABC):
    @abstractmethod
    def pay(self) -> str:
        pass


class CardPaymentMethod(PaymentMethod):
    def pay(self):
        return "by card"


class EWalletPaymentMethod(PaymentMethod):
    def pay(self):
        return "by e-wallet"


class SMSPaymentMethod(PaymentMethod):
    def pay(self):
        return "by SMS"


@dataclass
class PaymentRequest:
    route: Route = None
    passengers: list[Passenger] = None
    payment_method: PaymentMethod = None

    def execute(self):
        print(f"""a ticket for a trip from "{self.route.origin}" to "{self.route.destination}" \
costs {self.route.price} dollars and will be paid {self.payment_method.pay()},
passengers {self.passengers}""")


class PaymentRequestBuilder:
    def __init__(self):
        self.request = PaymentRequest()

    def is_valid(self) -> bool:
        return all((
            self.request.passengers is not None and len(
                self.request.passengers) > 0,
            self.request.route is not None,
            self.request.payment_method is not None,
        ))

    def build(self):
        if not self.is_valid():
            raise RuntimeError("try to build invalid request")
        return self.request

    def add_passenger(self, passenger):
        if self.request.passengers is None:
            self.request.passengers = []
        self.request.passengers.append(passenger)

    def set_route(self, route):
        self.request.route = route

```

```
    def set_payment_method(self, payment_method):
        self.request.payment_method = payment_method


builder = PaymentRequestBuilder()

builder.set_route(Route(1, "Los Angeles", "San Francisco", datetime.now(), 29))
builder.add_passenger(Passenger(1, "John Doe"))
builder.add_passenger(Passenger(1, "Jane Doe"))
builder.set_payment_method(CardPaymentMethod())

request = builder.build()
request.execute()

builder.set_payment_method(EWalletPaymentMethod())

request = builder.build()
request.execute()

builder.set_payment_method(SMSPaymentMethod())

request = builder.build()
request.execute()
```