# Взаимодействие паттернов Flyweight и Proxy

## Паттерн Легковес (Flyweight)

Паттерн Легковес (Flyweight) используется для минимизации потребления памяти путем хранения общего состояния, которое может быть повторно использовано несколькими объектами. Он особенно полезен в случаях, когда приложение создает множество объектов, которые имеют одинаковое внутреннее состояние.

## Паттерн Заместитель (Proxy)

Паттерн Заместитель (Proxy) предоставляет объект-заместитель, который контролирует доступ к другому объекту. Этот паттерн используется для добавления дополнительного уровня абстракции и контроля, например, для логирования, кэширования, контроля доступа и так далее.

## Взаимодействие паттернов

В данном примере паттерн Легковес используется для хранения координат, которые могут быть повторно использованы, чтобы уменьшить нагрузку на память. Паттерн Заместитель используется для управления реальным дроном, добавляя логирование и контроль доступа.

Теперь давайте рассмотрим подробную инструкцию по реализации кода.

## Пошаговая инструкция

### Шаг 1: Импорт необходимых библиотек

import time  
import math  
import matplotlib.pyplot as plt

Импортируем библиотеки, которые будем использовать в коде:

* time для задержек.
* math для математических вычислений.
* matplotlib.pyplot для построения графика.

### Шаг 2: Паттерн Flyweight (Легковес)

class CoordinateFlyweight:  
 \_coordinates = {}  
  
 @staticmethod  
 def get\_coordinate(lat, lon):  
 key = (lat, lon)  
 if key not in CoordinateFlyweight.\_coordinates:  
 CoordinateFlyweight.\_coordinates[key] = key  
 return CoordinateFlyweight.\_coordinates[key]

Создаем класс CoordinateFlyweight, который реализует паттерн Легковес. Этот паттерн позволяет экономить память, создавая уникальные экземпляры координат и переиспользуя их.

### Шаг 3: Паттерн Proxy (Заместитель)

class DJIDroneProxy:  
 def \_\_init\_\_(self, real\_drone):  
 self.\_real\_drone = real\_drone  
  
 def global\_position\_control(self, lat=None, lon=None, alt=None):  
 print(f"Запрос на перемещение к широте: {lat}, долготе: {lon}, высоте: {alt}")  
 self.\_real\_drone.global\_position\_control(lat, lon, alt)  
  
 def connect(self):  
 print("Запрос на подключение к дрону через SDK")  
 self.\_real\_drone.request\_sdk\_permission\_control()  
  
 def takeoff(self):  
 print("Взлет инициирован")  
 self.\_real\_drone.takeoff()  
  
 def land(self):  
 print("Посадка инициирована")  
 self.\_real\_drone.land()  
  
 def arm(self):  
 print("Армирование дрона инициировано")  
 self.\_real\_drone.arm()

паттерн Заместитель. Этот класс выступает посредником для реального дрона и добавляет дополнительное логирование.

### Шаг 4: Класс реального дрона

class DJIDrone:  
 def global\_position\_control(self, lat=None, lon=None, alt=None):  
 print(f"Перемещение к широте: {lat}, долготе: {lon}, высоте: {alt}")  
  
 def request\_sdk\_permission\_control(self):  
 print("Запрос на управление через SDK")  
  
 def takeoff(self):  
 print("Выполняем взлет")  
  
 def land(self):  
 print("Выполняем приземление")  
  
 def arm(self):  
 print("Армирование дрона")

Создаем класс DJIDrone, который содержит основные функции управления дроном.

### Шаг 5: Определение начальных координат и параметров

*# Определение начальных и конечных координат зоны полета*min\_lat = 57.826873  
min\_lon = 55.475823  
  
max\_lat = 57.922174  
max\_lon = 55.671439  
  
*# Вычисление начальных координат (центр зоны)*begin\_lat = min\_lat + (max\_lat - min\_lat) / 2  
begin\_lon = min\_lon + (max\_lon - min\_lon) / 2  
  
*# Шаг изменения координат и высота полета*step = 0.00005  
altitude = 50

Определяем начальные и конечные координаты, а также шаг и высоту полета.

### Шаг 6: Создание экземпляров классов дрона

*# Создание объектов реального дрона и его заместителя*real\_drone = DJIDrone()  
drone = DJIDroneProxy(real\_drone)  
  
*# Список для хранения координат полета*coordinates = []

Создаем экземпляры классов реального дрона и его заместителя, а также список для хранения координат.

### Шаг 7: Функция полета по спирали

*# Функция полета по спирали*def spiral(drone):  
 radius = 0  
 angle = 0  
 while radius <= (max\_lon - min\_lon) / 2:  
 radius += step *# Увеличение радиуса спирали* angle += math.pi / 180 *# Увеличение угла (в радианах)* x = math.sin(angle) \* radius *# Вычисление смещения по X* y = math.cos(angle) \* radius *# Вычисление смещения по Y* lat\_current = begin\_lat + x *# Вычисление текущей широты* lon\_current = begin\_lon + y *# Вычисление текущей долготы  
 # Использование паттерна Flyweight для получения координат* coordinate = CoordinateFlyweight.get\_coordinate(lat\_current, lon\_current)  
 coordinates.append(coordinate) *# Добавление координат в список  
 # Перемещение дрона к новым координатам* drone.global\_position\_control(lat=lat\_current, lon=lon\_current, alt=altitude)

**Шаг 8: Подключение, армирование, взлет, выполнение миссии и посадка дрона**

*# Подключение дрона*drone.connect()  
time.sleep(2) *# Задержка для эмуляции времени выполнения  
  
# Армирование дрона*drone.arm()  
time.sleep(2) *# Задержка  
  
# Взлет дрона*drone.takeoff()  
time.sleep(4) *# Задержка  
  
# Выполнение миссии полета по спирали*spiral(drone)  
  
*# Возврат дрона на начальную позицию*drone.global\_position\_control(begin\_lat, begin\_lon, alt=altitude)  
time.sleep(2) *# Задержка  
  
# Посадка дрона*drone.land()

Последовательно вызываем методы подключения, армирования, взлета, выполнения миссии и посадки дрона. Между действиями добавляем задержки для имитации реального времени.

**Шаг 9: Построение графика**

*# Построение графика пути дрона*latitudes, longitudes = zip(\*coordinates) *# Извлечение широт и долгот*plt.plot(latitudes, longitudes) *# Построение графика*plt.xlabel("Longitude") plt.ylabel("Latitude") plt.title("Движение дрона по спирали") plt.show()

Собираем координаты и строим график движения дрона по спирали с помощью matplotlib.