МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

ЛЬВІВСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ ПРИРОДОКОРИСТУВАННЯ

Факультет

Механіки, енергетики та ІТ

Кафедра

Інформаційних технологій

Практична робота №3

На тему: “Поведінкові шаблони проєктування”

Виконала студентка групи: IТ-31

Розвєзєва Христина Тарасівна

Перевірив: Татомир Андрій

Мета роботи - познайомитися з групою поведінкових шаблонів проєктування.

**Поведінкові шаблони проектування** визначають, як об'єкти взаємодіють між собою та розподіляють обов'язки. Вони забезпечують ефективну комунікацію та управління потоками в складних системах. Ці шаблони допомагають спростити контроль складних алгоритмів, взаємодію між об'єктами та полегшити розширення функціоналу.

шаблони поведінкової групи:

1. **Observer** — дозволяє одному об'єкту повідомляти інші про зміни стану.
2. **Strategy** — дозволяє вибирати алгоритм або спосіб виконання завдання під час виконання програми.
3. **Command** — перетворює запити на об'єкти, що дозволяє зберігати, передавати та виконувати запити як об'єкти.
4. **Chain of Responsibility** — дозволяє передавати запит по ланцюгу обробників до тих пір, поки його не обробить якийсь із них.
5. **State** — дозволяє об'єкту змінювати поведінку в залежності від його стану.

**Опис Observer:**

**Observer** — це поведінковий шаблон, який дозволяє одному об'єкту (суб'єкту) повідомляти інші об'єкти (спостерігачі) про зміни свого стану. Це корисно, коли у вас є багато об'єктів, які повинні реагувати на зміни в одному об'єкті, не вимагаючи жорсткого зв'язку між ними.

(Інтерфейс спостерігача)

class Observer:

def update(self, job):

pass

(Конкретний спостерігач)

class Worker(Observer):

def \_\_init\_\_(self, name):

self.name = name

def update(self, job):

print(f"{self.name} отримав нову вакансію: {job}")

(Суб'єкт)

class JobMarket:

def \_\_init\_\_(self):

self.\_observers = []

def attach(self, observer: Observer):

self.\_observers.append(observer)

def detach(self, observer: Observer):

self.\_observers.remove(observer)

def notify(self, job):

for observer in self.\_observers:

observer.update(job)

def post\_job(self, job):

print(f"Новий пост на ринку праці: {job}")

self.notify(job)

(Тестування)

if \_\_name\_\_ == "\_\_main\_\_":

job\_market = JobMarket()

worker1 = Worker("Олег")

worker2 = Worker("Анна")

job\_market.attach(worker1)

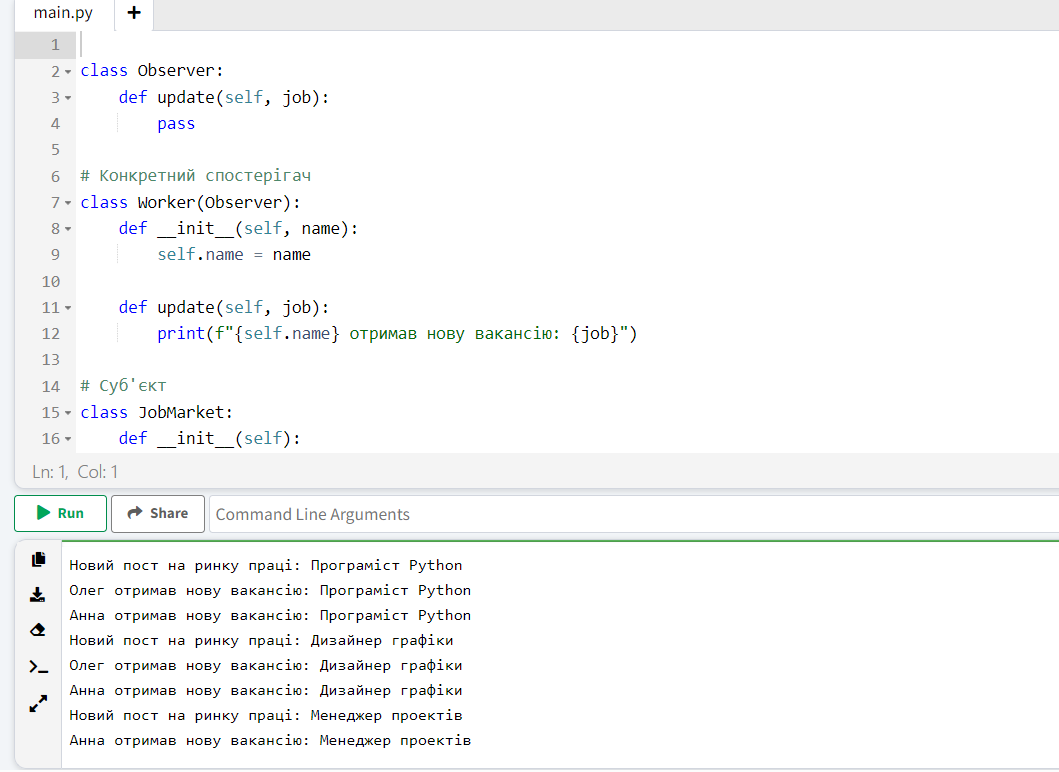
job\_market.attach(worker2)

job\_market.post\_job("Програміст Python") ( Повідомлення для всіх спостерігачів)

job\_market.post\_job("Дизайнер графіки") (Повідомлення для всіх спостерігачів)

job\_market.detach(worker1)

job\_market.post\_job("Менеджер проектів") (Повідомлення лише для Анни)



UML-діаграма

Observer

JobMarket

- name: str

+ update(job)

Worker

+ update(job)

- \_observers: List

+ attach(observer)

+ detach(observer)

+ notify(job)

+ post\_job(job)

**Observer** — базовий інтерфейс для всіх спостерігачів. Він містить метод update(), який буде викликаний при зміні в суб'єкті.

**Worker** — конкретна реалізація спостерігача, яка отримує нові вакансії від ринку праці.

**JobMarket** — клас, що представляє ринок праці. Він має методи для додавання (attach) та видалення (detach) спостерігачів, а також метод notify(), який сповіщає всіх спостерігачів про нові вакансії. Метод post\_job() використовується для публікації нових вакансій.

У блоці тестування створюється ринок праці та два робітники. Спочатку обидва робітники підписуються на нові вакансії, а потім один з них відписується.

### Висновок:

Шаблон **Observer** забезпечує зручний спосіб спостереження за змінами в одному об'єкті (в даному випадку ринку праці) і автоматично інформує всіх зацікавлених сторін (працівників) про ці зміни. Це дозволяє зберігати слабку зв'язаність між об'єктами та робить систему більш гнучкою і розширюваною.