Міністерство освіти і науки України
Національний технічний університет України
“Київський політехнічний інститут імені Ігоря Сікорського”
Факультет інформатики та обчислювальної техніки
Кафедра інформаційних систем та технологій

Лабораторна робота №4
ШАБЛОНИ «SINGLETON», «ITERATOR», «PROXY», «STATE», «STRATEGY»

Варіант 14

Виконав:
студент групи ІА – 24:
Любченко І.М

Перевірив:
Мягкий М.Ю

Київ 2024

Завдання:

1. Ознайомитися з короткими теоретичними відомостями.
2. Реалізувати частину функціонала робочої програми у вигляді класів та їхньої взаємодії для досягнення конкретних функціональних можливостей.
3. Застосування одного з розглянутих шаблонів при реалізації програми

Варіант:

..14 Архіватор (strategy, adapter, factory method, facade, visitor, p2p)

Архіватор повинен являти собою візуальний додаток з можливістю створення і редагування архівів різного типу (.tar.gz, .zip, .rar, .ace) - додавання/ видалення файлів / папок, редагування метаданих (по можливості), перевірка checksum архівів, тестування архівів на наявність пошкоджень, розбиття архівів на частини.

Короткі теоритичні відомості

Шаблони проектування — це стандартизовані рішення для типових задач, які часто виникають під час розробки програмного забезпечення. Вони є своєрідними схемами, які допомагають ефективно вирішувати проблеми, роблячи код гнучкішим, розширюваним та зрозумілим.

Шаблон Singleton призначений для забезпечення єдиного екземпляра класу з глобальною точкою доступу до нього. Це зручно в ситуаціях, коли необхідно централізовано зберігати стан, наприклад, для конфігурацій або доступу до бази даних. Таким чином, кожен раз, коли необхідний екземпляр класу, програма звертається до єдиного доступного об’єкта, що забезпечує унікальність і захищає від випадкових змін.

Шаблон Iterator створений для роботи з колекціями та дозволяє отримувати доступ до елементів послідовно, не розкриваючи внутрішньої структури колекції. Це особливо корисно, коли потрібно пройтися по елементах великого масиву або списку, адже ітератор надає єдиний стандартний інтерфейс для доступу, незалежно від того, яким чином колекція реалізована.

Proxy надає об'єкт-замісник, який контролює доступ до реального об'єкта. Це допомагає у випадках, коли необхідно обмежити або розширити функціональність об’єкта, наприклад, через додаткову логіку безпеки або кешування. Замісник перехоплює виклики до основного об'єкта, додає потрібні

операції (такі як перевірка прав доступу або журналювання), а потім пересилає запит далі.

State призначений для управління поведінкою об'єкта залежно від його поточного стану. Цей шаблон корисний, коли об'єкт має різні стани, і кожен з них вимагає відмінної поведінки. Наприклад, в автоматі з продажу товарів зміна станів впливає на те, як автомат реагує на дії користувача. Об’єкт автоматично переходить у відповідний стан, і його поведінка змінюється відповідно.

Strategy надає можливість обирати різні алгоритми для вирішення певної задачі під час виконання програми. Це дозволяє об'єкту змінювати поведінку, коли змінюються зовнішні умови або потреби користувача, не змінюючи при цьому основний код. Кожен алгоритм інкапсулюється в окремому класі, і об'єкт просто підставляє потрібну стратегію в залежності від конкретної ситуації, що робить код гнучким і розширюваним.

Хід роботи

1. Теоретичне введення У рамках даної роботи було застосовано патерн проектування **Стратегія**, який дозволяє визначати сімейство схожих алгоритмів та розміщувати їх у окремих класах, що дозволяє заміняти один алгоритм на інший під час виконання програми. У цьому випадку патерн застосовувався для створення і розпакування архівів різних форматів, таких як ZIP, RAR і TAR.GZ. Кожен формат архіву має свою стратегію для реалізації операцій створення та розпакування архівів.

src > strategies.py > ...

```python
class ZipStrategy:
    def create_archive(self, files):
        # Логіка для створення ZIP архіву
        print("Creating ZIP archive...")

    def extract_archive(self, archive):
        # Логіка для розпакування ZIP архіву
        print("Extracting ZIP archive...")

class RarStrategy:
    def create_archive(self, files):
        # Логіка для створення RAR архіву
        print("Creating RAR archive...")

    def extract_archive(self, archive):
        # Логіка для розпакування RAR архіву
        print("Extracting RAR archive...")

class TarGzStrategy:
    def create_archive(self, files):
        # Логіка для створення TAR.GZ архіву
        print("Creating TAR.GZ archive...")

    def extract_archive(self, archive):
        # Логіка для розпакування TAR.GZ архіву
        print("Extracting TAR.GZ archive...")

```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

```
(venv) PS C:\Users\lubch\Desktop\TRPZ_labs_Liubchenko_IA24\lab4trpz\src> python .\main.py
```

Рисунок 1 – код для визначення трьох стратегій

2. **Реалізація патерну Стратегія** Використання патерну Стратегія дозволяє зробити алгоритми архівації ізольованими один від одного, що робить код більш гнучким і підтримуваним. Це дозволяє змінювати алгоритм архівації без необхідності змінювати бізнес-логіку програми.

У нашій реалізації були визначені три стратегії для кожного формату архіву:

- **ZipStrategy**: для роботи з ZIP архівами.
- **RarStrategy**: для роботи з RAR архівами.
- **TarGzStrategy**: для роботи з TAR.GZ архівами.

src > archiver.py > ...

```python
import zipfile
import tarfile
import rarfile


class Archiver:
    def __init__(self, archive_type):
        self.archive_type = archive_type
        self.archive = None

    def create_archive(self, file_name):
        if self.archive_type == 'zip':
            self.archive = zipfile.ZipFile(file_name, 'w', zipfile.ZIP_DEFL
        elif self.archive_type == 'tar.gz':
            self.archive = tarfile.open(file_name, 'w:gz')
        elif self.archive_type == 'rar':
            raise NotImplementedError("RAR підтримується лише для читання")
        else:
            raise ValueError(f"Непідтримуваний тип архіву: {self.archive_ty

    def add_file(self, file_path):
        if not self.archive:
            raise RuntimeError("Архів ще не створено!")

        if self.archive_type == 'zip':
            self.archive.write(file_path, arcname=file_path.split('/')[-1])
        elif self.archive_type == 'tar.gz':
            self.archive.add(file_path)

    def close(self):
        if self.archive:
            self.archive.close()

```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

```
(venv) PS C:\Users\lubch\Desktop\TRPZ_labs_Liubchenko_IA24\lab4trpz\src> python .\main.py
```

Рисунок 2 – код функціоності для створення та додавання файлів до архівів різних форматів: ZIP, TAR.GZ і RAR.

3. **Основна логіка програми** Клас **Archiver** працює з обраною стратегією для створення і розпакування архівів. Клас **Archiver** дозволяє змінювати стратегію архівації під час виконання програми, що робить систему гнучкою.

src > main.py > ...

```python
from gui import ArchiverApp
import tkinter as tk

if __name__ == "__main__":
    root = tk.Tk()
    app = ArchiverApp(root)
    root.mainloop()

```

Рисунок 3 - запуск графічного інтерфейсу програми,за допомогою бібліотеки **Tkinter**

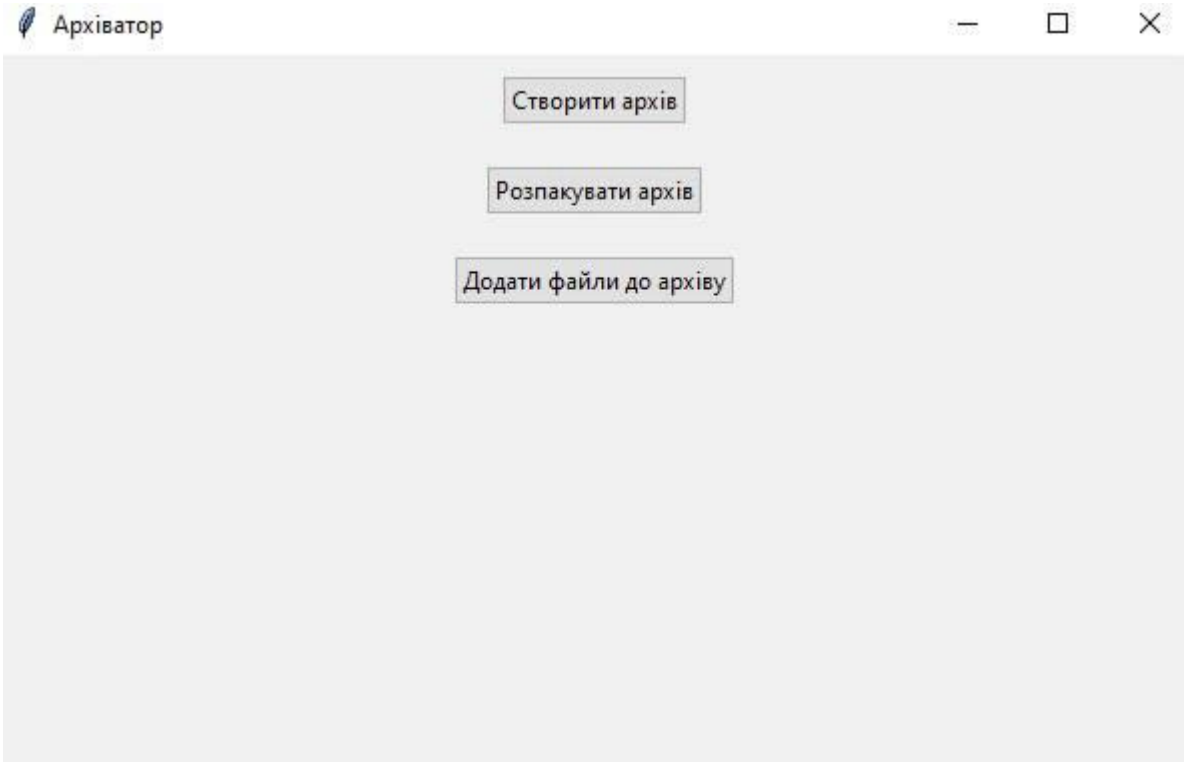


Рисунок 4 - Отриманий функціонал

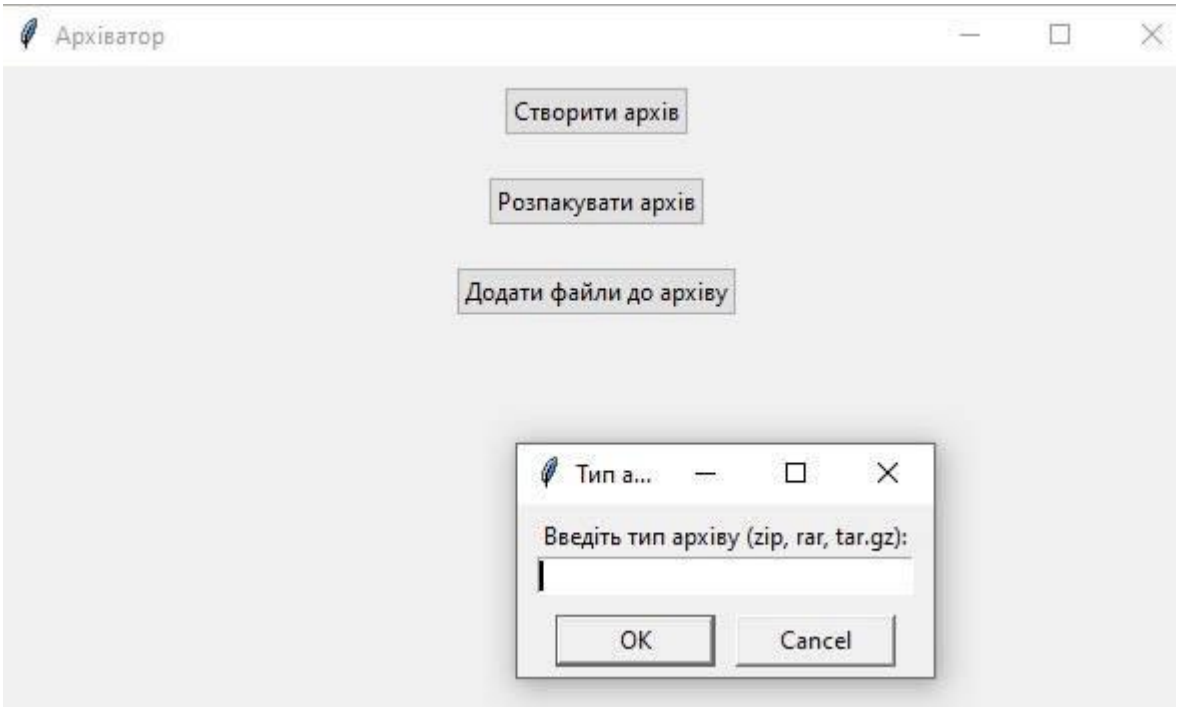


Рисунок 5 - Вибір “Створити архів”

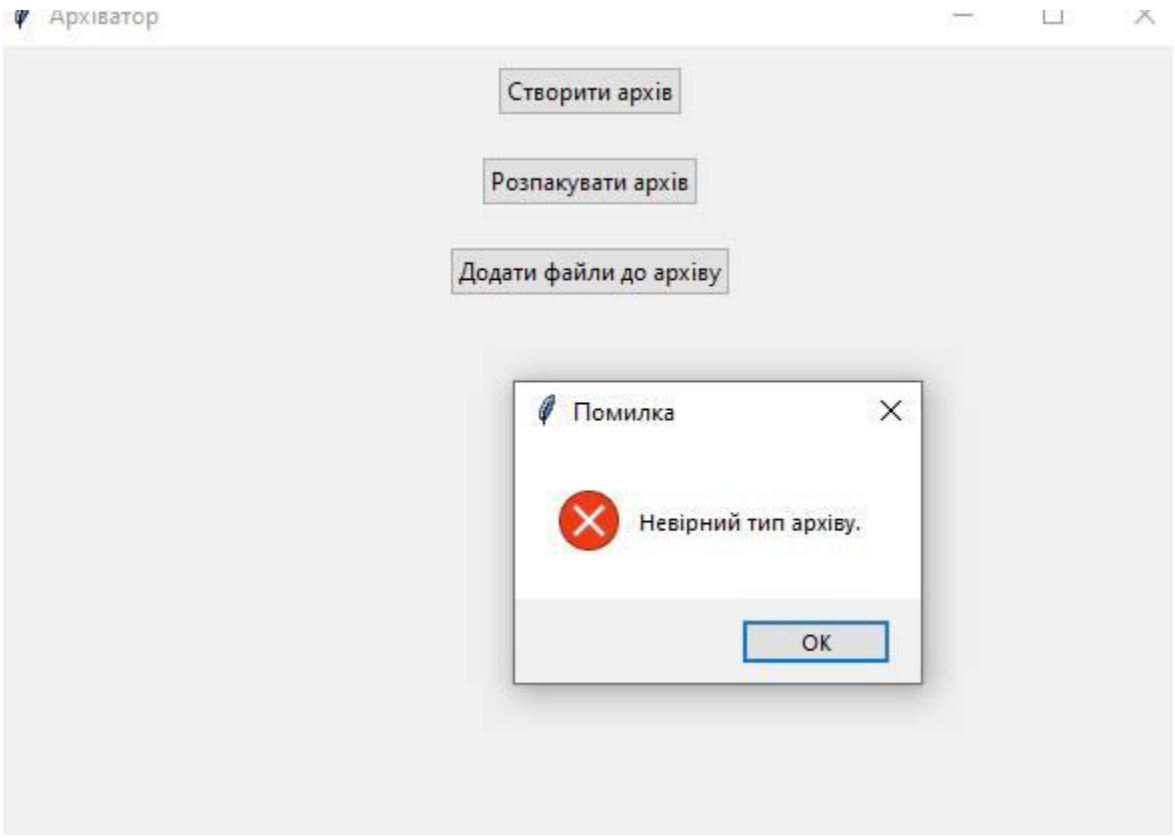
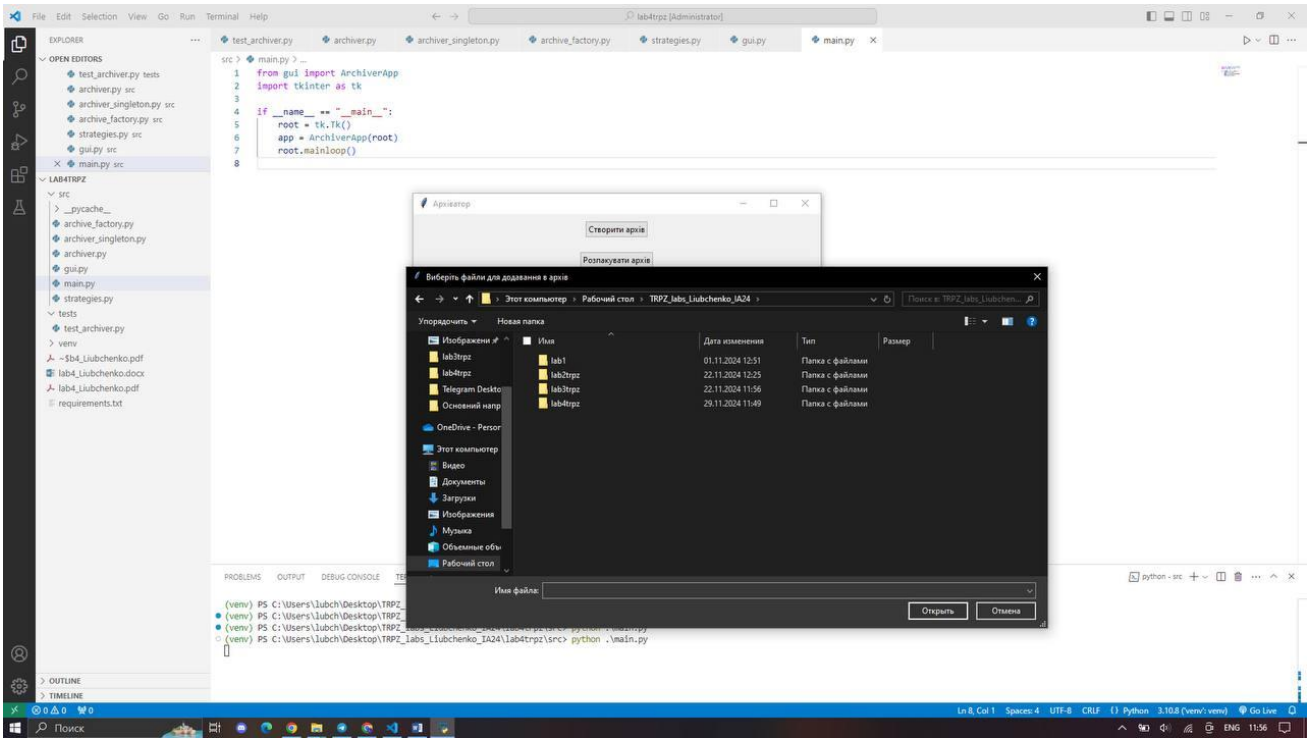


Рисунок 6- некоректний тип архіву



Рисункок 7 - коректний тип архіву

У разі “ Розпакувати архів” і “Додати файли до архіву” ми отримаємо можливість обирання файлів як на минулій картинці.

Висновок Реалізувавши патерн **Стратегія**, мені вдалося реалізувати функціональність для створення та розпакування архівів різних форматів. Заміна алгоритмів архівації на льоту значно спрощила мені роботу з різними форматами архівів. Це дозволило зробити код більш підтримуваним і зрозумілим.