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 №8

Тема: **ШАБЛОНИ «COMPOSITE», «FLYWEIGHT», «INTERPRETER», «VISITOR»**

Варіант 14

Виконав

студент групи ІА – 24:

Любченко І.М

Перевірив:

Мягкий М. Ю

Київ 2024

Зміст

Мета: метою виконання лабораторної роботи є вивчення та практичне застосування шаблонів проєктування, таких як «COMPOSITE», «FLYWEIGHT», «INTERPRETER», «VISITOR»для роботи з БД при розробці корпоративних додатків. Ця лабораторна робота спрямована на розвиток навичок у використанні різних патернів для вирішення задач у розробці програмного забезпечення.

Теоретичні відомості

Шаблон "Composite"

Шаблон "Composite" дозволяє створювати деревоподібні структури об'єктів, де окремі об'єкти та їх колекції можуть оброблятися однаково. Це особливо корисно для роботи зі складними структурами даних, такими як ієрархії. Основні компоненти шаблону:

- Component: Абстрактний клас або інтерфейс, який визначає поведінку, яку мають реалізувати як окремі елементи, так і колекції елементів.
- Leaf: Клас, що представляє окремі елементи без підлеглих елементів.
- Composite: Клас, що представляє колекції елементів і зберігає підлеглі елементи.

Шаблон "Flyweight"

Шаблон "Flyweight" спрямований на оптимізацію використання пам'яті за рахунок повторного використання вже створених об'єктів. Він дозволяє зменшити кількість об'єктів, що створюються, шляхом розділення стану об'єктів на внутрішній і зовнішній:

- Внутрішній стан: Постійний і загальний для всіх об'єктів.
- Зовнішній стан: Відмінний і визначається контекстом використання.

Шаблон "Interpreter"

Шаблон "Interpreter" надає спосіб визначення граматики для мови та інтерпретації її речень. Він особливо корисний для створення мов, які використовуються для

специфічних задач: AbstractExpression: Абстрактний клас, що визначає інтерфейс для всіх виразів.

- TerminalExpression: Клас для термінальних виразів у граматиці.
- NonterminalExpression: Клас для нетермінальних виразів, що представляють правила граматики.

Шаблон "Visitor"

Шаблон "Visitor" дозволяє додавати нові операції до класів, не змінюючи їх. Він особливо корисний, коли ви хочете додати нові функціональні можливості до об'єктної структури, не змінюючи її:

- Visitor: Абстрактний клас або інтерфейс, що визначає операції, які можуть бути виконані над елементами об'єктної структури.
- ConcreteVisitor: Конкретні реалізації відвідувача, що визначають поведінку операцій.
- Element: Інтерфейс або абстрактний клас для елементів, які можуть бути прийняті відвідувачем.

Хід роботи

1. Ознайомитися з короткими теоретичними відомостями.
2. Реалізувати частину функціоналу робочої програми у вигляді класів і їх взаємодій для досягнення конкретних функціональних можливостей
3. Застосування одного з розглянутих шаблонів при реалізації програми.

Тема: Архіватор

У контексті архіватора, який має можливість працювати з різними типами архівів (tar.gz, zip, rar, ace), я вирішив застосувати шаблон **Composite** замість **Visitor** з кількох причин.

1. Ієрархічна структура даних

Архіви за своєю природою мають ієрархічну структуру, де є як окремі файли, так і вкладені папки, що створює дерево об'єктів. Шаблон **Composite** ідеально підходить для таких випадків, оскільки дозволяє представляти як одиничні елементи (файли), так і складені елементи (папки, що містять інші файли чи папки) через єдиний інтерфейс. Завдяки цьому можна однаково працювати як з окремими файлами, так і з цілими каталогами в архіві.

2. Спрощення обробки складених і простих об'єктів

У **Composite** всі об'єкти (файли та папки) обробляються за допомогою одного інтерфейсу, що дозволяє значно спростити код. Наприклад, операції додавання, видалення або перевірки метаданих можуть здійснюватися без необхідності розрізняти чи є об'єкт простим файлом чи складною папкою. В той час як **Visitor** потребує додаткових класів для кожної операції, що значно ускладнює код, особливо якщо кількість можливих операцій велика (створення архіву, перевірка архіву, додавання/видалення файлів, редагування метаданих тощо).

3. Легкість розширення

Якщо в майбутньому буде необхідно додати нові типи елементів у архів (наприклад, нові спеціалізовані типи файлів або папок), **Composite** дозволяє легко розширювати архітектуру без необхідності змінювати багато частин коду. Просто додається новий тип елемента, і все працює так, як і з попередніми. Водночас, у **Visitor** для кожної нової операції потрібно додавати нові відвідувачі або змінювати існуючі, що може швидко призвести до перевантаження коду і зниження його гнучкості.

4. Простота реалізації операцій на всіх рівнях ієрархії

У випадку архіватора необхідно виконувати кілька операцій на різних рівнях ієрархії: додавання/видалення файлів, редагування метаданих, перевірка архіву на наявність файлів тощо. **Composite** дозволяє реалізувати ці операції в одному методі для всіх об'єктів, незалежно від того, чи це файл, чи папка. У **Visitor** ж для кожної операції необхідно створювати окремі класи відвідувачів, що ускладнює підтримку і розвиток проекту.

5. Легкість інтеграції з іншими частинами архіватора

Архіватор використовує різні технології та підходи, такі як **Strategy**, **Adapter**, **Factory Method** та інші. Шаблон **Composite** органічно вписується в таку структуру, оскільки дозволяє зручно комбінувати ієрархічні об'єкти (папки та файли) з іншими компонентами архіватора без зайвих ускладнень. **Visitor** потребує додаткової обробки для інтеграції з іншими частинами системи, оскільки кожен новий елемент вимагає свого відвідувача, що може ускладнити архітектуру і створити додаткові зв'язки між компонентами.

6. Вища ефективність для нашого випадку

У випадку з архіватором, де більшість операцій стосується роботи з деревами файлів і папок, використання **Composite** дозволяє скоротити кількість необхідних класів і зменшити складність реалізації. Якщо б ми застосовували **Visitor**, ми б мали створювати безліч нових відвідувачів для кожної операції, що значно збільшило б обсяг коду і зробило його менш гнучким для майбутніх змін.

Реалізація Шаблону Composite в Лабораторній Роботі

Я використовував шаблон «Composite», оскільки він дозволяє працювати з окремими файлами та колекціями файлів однаково. Це ідеально підходить для архіватора, дозволяючи ефективно додавати файли до архівів, створювати архіви і розпаковувати їх, зберігаючи при цьому чисту структуру і організацію коду.

src > component.py > ...

```python
from abc import ABC, abstractmethod

class Component(ABC):
    @abstractmethod
    def add(self, component):
        pass

    @abstractmethod
    def remove(self, component):
        pass

    @abstractmethod
    def display(self, depth):
        pass

    @abstractmethod
    def archive(self, archiver):
        pass
```

Рис 1. - Абстрактний клас Component визначає методи add, remove, display, та archive, які мають реалізувати класи Leaf та Composite.

```python
from component import Component

class Leaf(Component):
    def __init__(self, name, path):
        self.name = name
        self.path = path

    def add(self, component):
        raise Exception("Cannot add to a leaf")

    def remove(self, component):
        raise Exception("Cannot remove from a leaf")

    def display(self, depth):
        print("-" * depth + self.name)

    def archive(self, archiver, db_manager, archive_id):
        archiver.add_file(self.path)
        db_manager.add_file(archive_id, self.path)
```

Рис 2. - Клас Leaf представляє окремі файли. Він реалізує методи Component, але методи add та remove не мають сенсу для окремих файлів, тому генерують виключення. Метод archive додає файл до архіву та записує інформацію до бази даних.

```python
from component import Component

class Composite(Component):
    def __init__(self, name):
        self.name = name
        self.children = []

    def add(self, component):
        self.children.append(component)

    def remove(self, component):
        self.children.remove(component)

    def display(self, depth):
        print("-" * depth + self.name)
        for child in self.children:
            child.display(depth + 2)

    def archive(self, archiver, db_manager, archive_id):
        for child in self.children:
            child.archive(archiver, db_manager, archive_id)
```

Рис 3. - Клас Composite представляє колекції файлів і забезпечує можливість додавання, видалення та отримання підлеглих елементів. Метод archive делегує роботу підлеглим елементам і записує інформацію до бази даних.

src > database_manager.py > ...

```python
import sqlite3

class DatabaseManager:
    def __init__(self, db_name="archiver.db"):
        self.connection = sqlite3.connect(db_name)
        self.cursor = self.connection.cursor()
        self.create_tables()

    def create_tables(self):
        self.cursor.execute("""
            CREATE TABLE IF NOT EXISTS archives (
                id INTEGER PRIMARY KEY AUTOINCREMENT,
                name TEXT NOT NULL
            )
        """)
        self.cursor.execute("""
            CREATE TABLE IF NOT EXISTS files (
                id INTEGER PRIMARY KEY AUTOINCREMENT,
                archive_id INTEGER,
                path TEXT NOT NULL,
                FOREIGN KEY (archive_id) REFERENCES archives (id)
            )
        """)
        self.cursor.execute("""
            CREATE TABLE IF NOT EXISTS notifications (
                id INTEGER PRIMARY KEY AUTOINCREMENT,
                message TEXT NOT NULL,
                timestamp DATETIME DEFAULT CURRENT_TIMESTAMP
            )
        """)
        self.connection.commit()

    def add_archive(self, name):
        self.cursor.execute("INSERT INTO archives (name) VALUES (?)", (name,))
        self.connection.commit()
        return self.cursor.lastrowid

    def add_file(self, archive_id, file_path):
        self.cursor.execute("INSERT INTO files (archive_id, path) VALUES (?, ?)", (archive_id, file_path))
        self.connection.commit()

    def add_notification(self, message):
        self.cursor.execute("INSERT INTO notifications (message) VALUES (?)", (message,))
        self.connection.commit()

    def get_archives(self):
        self.cursor.execute("SELECT * FROM archives")
        return self.cursor.fetchall()


    def get_files_in_archive(self, archive_id):
        self.cursor.execute("SELECT path FROM files WHERE archive_id = ?", (archive_id,))
        return self.cursor.fetchall()

    def close(self):
        self.connection.close()
```

Рис 4-5. - Клас DatabaseManager забезпечує роботу з базою даних SQLite, включаючи створення таблиць, додавання архівів, файлів і сповіщень, а також отримання даних з бази даних.

src > archiver.py > Archiver > add_file

```python
from archive_factory import ArchiveFactory
from observable import Observable
import os

class Archiver(Observable):
    def __init__(self, archive_type):
        super().__init__()
        self.adapter = ArchiveFactory.create_archiver(archive_type)
        self.files_added = set()
        self.archive_initialized = False

    def create_archive(self, file_name):
        if self.adapter:
            self.adapter.create_archive(file_name)
            self.archive_initialized = True
            self.notify_observers(f"Archive {file_name} created")
        else:
            self.notify_observers("Error: Adapter not initialized")

    def add_file(self, file_path):
        if not self.archive_initialized:
            self.notify_observers("Error: Archive not initialized")
            return

        if os.path.exists(file_path):
            if file_path not in self.files_added:
                self.adapter.add_file(file_path)
                self.files_added.add(file_path)
                self.notify_observers(f"File {file_path} added to archive")
            else:
                self.notify_observers(f"File {file_path} already added to archive")
        else:
            self.notify_observers(f"Error: File {file_path} not found")

    def close(self):
        if not self.archive_initialized:
            self.notify_observers("Error: Archive not initialized")
            return

        if hasattr(self.adapter, 'close'):
            self.adapter.close()
            self.archive_initialized = False
            self.notify_observers("Archive process completed")

    def extract_archive(self, archive_path, destination_folder):
        if self.adapter:
            self.adapter.extract_archive(archive_path, destination_folder)
            self.notify_observers(f"Archive {archive_path} successfully extracted to {destination_folder}")
        else:
            self.notify_observers(f"Archive {archive_path} successfully extracted to {destination_folder}")
        else:
            self.notify_observers("Error: Adapter not initialized")

    def archive_component(self, component, db_manager, archive_id):
        component.archive(self, db_manager, archive_id)
        self.close()
        self.notify_observers("Archiving components completed")
```

Рис 6-7. – Додаю метод archive_component, який використовує шаблон «Composite» для архівування компонентів і сповіщає спостерігачів.

rc > archive_facade.py > ...

```python
from archiver import Archiver
from database_manager import DatabaseManager

class ArchiveFacade:
    def __init__(self, archive_type, db_manager):
        self.archiver = Archiver(archive_type)
        self.db_manager = db_manager

    def create_archive(self, file_name, component):
        archive_id = self.db_manager.add_archive(file_name)
        self.archiver.create_archive(file_name)
        self.archiver.archive_component(component, self.db_manager, archive_id)

    def extract_archive(self, archive_path, destination_folder):
        self.archiver.extract_archive(archive_path, destination_folder)
```

Рис 8. - Додаю параметри db_manager та archive_id у метод create_archive для збереження дій у базу даних.

src > gui.py > ...

```python
import tkinter as tk
from tkinter import filedialog, messagebox
from tkinter import ttk
from archive_facade import ArchiveFacade
from archive_process_observer import ArchiveProcessObserver
from composite import Composite
from leaf import Leaf
import tkinter.simpledialog
from database_manager import DatabaseManager

class ArchiverApp:
    def __init__(self, root):
        self.root = root
        self.root.title("Архіватор")
        self.root.geometry("600x400")
        self.create_buttons()
        self.facade = None
        self.observers = []
        self.db_manager = DatabaseManager("archiver.db")

        # Ініціалізація компонента дерева
        self.root_component = Composite("root")

    def create_buttons(self):
        self.create_button = ttk.Button(self.root, text="Створити архів", command=self.create_archive)
        self.create_button.pack(pady=10)

        self.extract_button = ttk.Button(self.root, text="Розпакувати архів", command=self.extract_archive)
        self.extract_button.pack(pady=10)

        self.add_button = ttk.Button(self.root, text="Додати файли до архіву", command=self.add_files_to_archive)
        self.add_button.pack(pady=10)

        self.show_db_button = ttk.Button(self.root, text="Показати базу даних", command=self.show_database)
        self.show_db_button.pack(pady=10)

    def add_observer_to_archiver(self, observer):
        self.observers.append(observer)
        if self.facade is not None:
            self.facade.archiver.add_observer(observer)

    def notify_observers(self, message):
        for observer in self.observers:
            observer.update(message)
        self.db_manager.add_notification(message)

    def create_archive(self):
        archive_type = self.get_archive_type()
        if archive_type:
```

```
            archive_name = filedialog.asksaveasfilename(defaultextension=f".{archive_type}",
                                                        filetypes=[(f"{archive_type.upper()} файли", f"*.{archive_type}")])
            if archive_name:
                try:
                    self.facade = ArchiveFacade(archive_type, self.db_manager)
                    self.facade.create_archive(archive_name, self.root_component)
                    messagebox.showinfo("Успіх", f"Архів {archive_name} успішно створений.")
                    self.notify_observers(f"Архів {archive_name} успішно створений.")
                    self.show_database()  # Показати базу даних після створення архіву
                except Exception as e:
                    self.notify_observers(f"Error: {str(e)}")
                    messagebox.showerror("Помилка", str(e))

    def extract_archive(self):
        file_path = filedialog.askopenfilename(filetypes=[("Всі архіви", "*.zip;*.tar.gz;*.rar")])
        if file_path:
            destination_folder = filedialog.askdirectory()
            if destination_folder:
                try:
                    self.facade = ArchiveFacade(file_path.split('.')[-1], self.db_manager)
                    self.facade.extract_archive(file_path, destination_folder)
                    messagebox.showinfo("Успіх", f"Архів успішно розпаковано до {destination_folder}.")
                    self.notify_observers(f"Архів успішно розпаковано до {destination_folder}.")
                    self.show_database()  # Показати базу даних після розпакування архіву
                except Exception as e:
                    self.notify_observers(f"Error: {str(e)}")
                    messagebox.showerror("Помилка", str(e))

    def add_files_to_archive(self):
        files_to_add = filedialog.askopenfilenames(title="Виберіть файли для додавання")
        if files_to_add:
            try:
                for file_path in files_to_add:
                    leaf = Leaf(file_path.split("/")[-1], file_path)
                    self.root_component.add(leaf)
                self.notify_observers("Файли успішно додано до архіву.")
                messagebox.showinfo("Успіх", "Файли успішно додано до архіву.")
            except Exception as e:
                self.notify_observers(f"Error: {str(e)}")
                messagebox.showerror("Помилка", str(e))

    def get_archive_type(self):
        archive_type = tkinter.simpledialog.askstring("Тип архіву", "Введіть тип архіву (zip, rar, tar.gz):")
        if archive_type in ['zip', 'rar', 'tar.gz']:
            return archive_type
        else:
            messagebox.showerror("Помилка", "Невірний тип архіву.")
                return None

    def show_database(self):
        archives = self.db_manager.get_archives()
        db_info = ""
        for archive in archives:
            archive_id, archive_name = archive
            db_info += f"Архів: {archive_name}\n"
            files = self.db_manager.get_files_in_archive(archive_id)
            for file in files:
                db_info += f"  Файл: {file[0]}\n"
        messagebox.showinfo("База даних", db_info)
```

Рис 9-10. - Додаю кнопку для відображення вмісту бази даних і метод show_database, який отримує дані з бази даних і відображає їх у повідомленні.

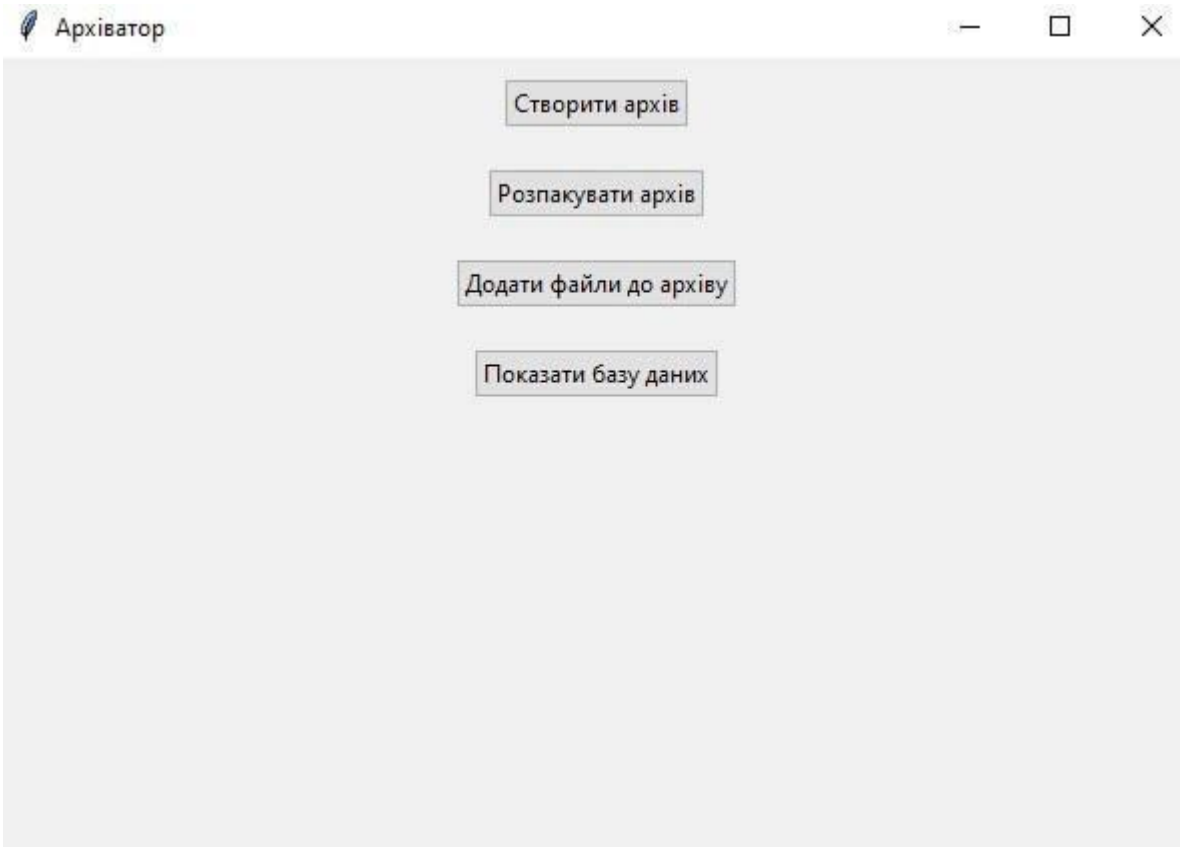

Рис 11. – Панель функціонала

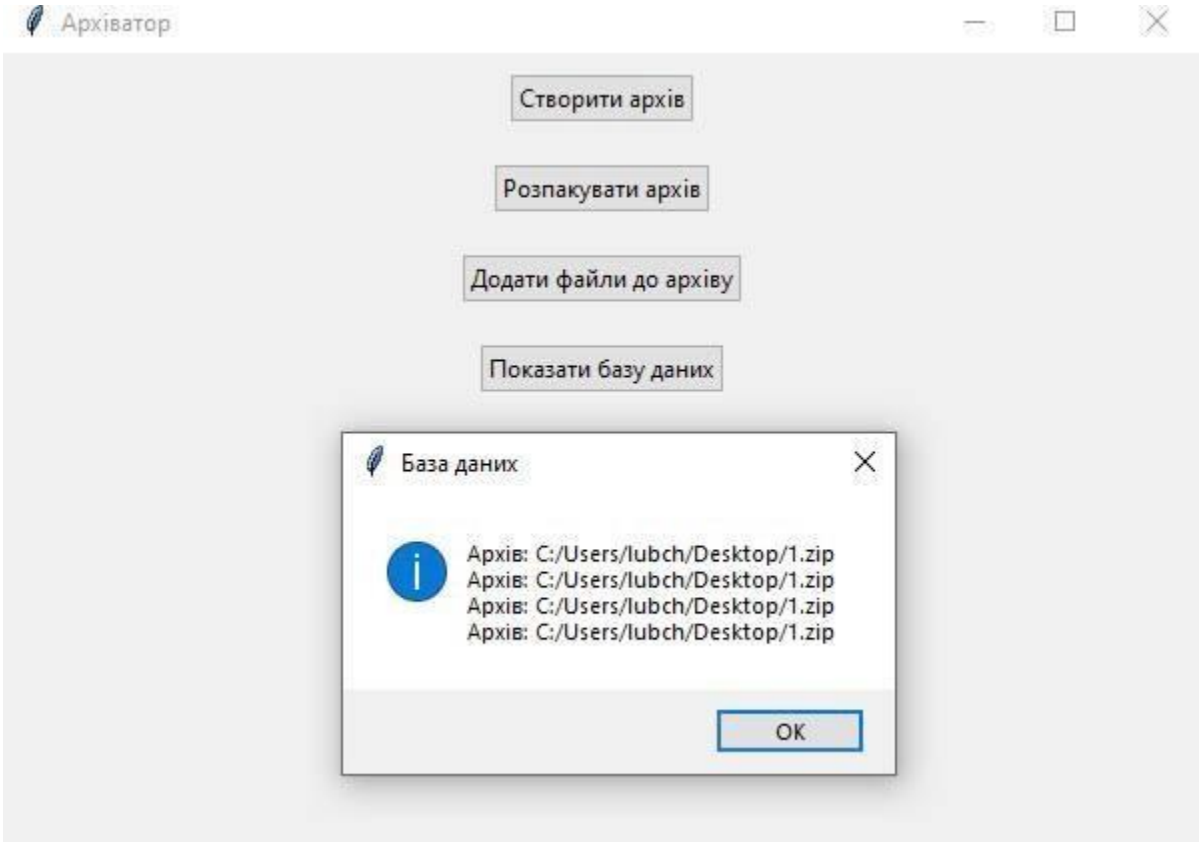

Рис 12. – Результати виклику функції: “Показати базу даних”

Висновок: У цій лабораторній роботі я зосередився на використанні шаблону «Composite» для створення ефективної структури для архівування файлів. Завдяки цьому шаблону я організував файли в деревоподібну структуру, де кожен файл або набір файлів може оброблятися однаково, що значно спрощує роботу з архівами.

Я також інтегрували роботу з базою даних для збереження інформації про створені архіви та файли в них. Це дозволило мені легко відстежувати дії користувача і забезпечує прозорість у роботі з архівами.

Загалом, використання шаблону «Composite» забезпечило мені гнучкість і зручність у роботі з файлами та архівами, полегшило підтримку і розширення системи. Інтеграція з базою даних додала додатковий рівень функціональності, що підвищило надійність і ефективність мого архіватора. Це дозволило створити модульну і добре організовану систему, яка може легко масштабуватися і адаптуватися до нових вимог