МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ "ЛЬВІВСЬКА ПОЛІТЕХНІКА"

Інститут **КНІТ**
Кафедра **ПЗ**

ЗВІТ

До лабораторної роботи № 6
З дисципліни: *"Архітектура та проектування програмного забезпечення"*
На тему: *"Використання структурних патернів"*

Лектор:

проф. каф. ПЗ

Фоменко А. В.

Виконали:

студ. групи ПЗ-41

Процiв О. М.

Малеєв А.
Бабілля О.

Прийняв:

асис. каф. ПЗ

Шкраб Р.Р.

«___» _____ 2024р.

∑ = _____ _____

Львів – 2024

Тема роботи: Використання структурних патернів.

Мета роботи: Ознайомитися з основними шаблонами проектування, навчитися застосовувати їх при проектуванні і розробці ПЗ.

ТЕОРЕТИЧНІ ВІДОМОСТІ

Шаблон проектування або патерн (англ. Design pattern) в розробці програмного забезпечення - повторювана архітектурна конструкція, що представляє собою рішення проблеми проектування в рамках деякого часто виникає контексту.

Зазвичай шаблон не є закінченим зразком, який може бути прямо перетворений в код; це лише приклад вирішення завдання, який можна використовувати в різних ситуаціях. Об'єктно-орієнтовані шаблони показують відносини і взаємодії між класами чи об'єктами, без визначення того, які саме кінцеві класи чи об'єкти докладання будуть використовуватися.

Використання патернів проектування дає розробнику ряд незаперечних переваг. Наведемо деякі з них. Модель системи, побудована в термінах патернів проектування, фактично є структурованим виділенням тих елементів і зв'язків, які значимі при вирішенні поставленого завдання. Крім цього, модель, побудована з використанням патернів проектування, більш проста і наочна у вивченні, ніж стандартна модель. Тим не менш, незважаючи на простоту і наочність, вона дозволяє глибоко і всебічно опрацювати архітектуру розроблюваної системи з використанням спеціальної мови. Застосування патернів проектування підвищує стійкість системи до зміни вимог і спрощує неминучу подальшу доопрацювання системи. Крім того, важко переоцінити роль використання патернів при інтеграції інформаційних систем організації. Також слід згадати, що сукупність патернів проектування, по суті, являє собою єдиний словник проектування, який, будучи уніфікованим засобом, незамінний для спілкування розробників один одним.

Але найголовніше ¬ будь шаблон проектування може стати палицею з двома кінцями: якщо він буде застосований не до місця, це може обернутися катастрофою і створити вам багато проблем у подальшому. У той же час, реалізований в потрібному місці, в потрібний час, він може стати для вас справжнім рятівником.

Є три основних види шаблонів проектування:

- структурні;
- породжувальні;
- поведінкові.

ЗАВДАННЯ

1. Вивчити поведінкові патерни. Вивчити породжувальні патерни. Вивчити структурні патерни , описати можливість використання 2-х патернів у розроблювальному модулі
2. Продемонструвати на діаграмі класів
3. Навести або код, або псевдокод
4. Описати і продемонструвати хоча б один архітектурний патерн.

ХІД РОБОТИ

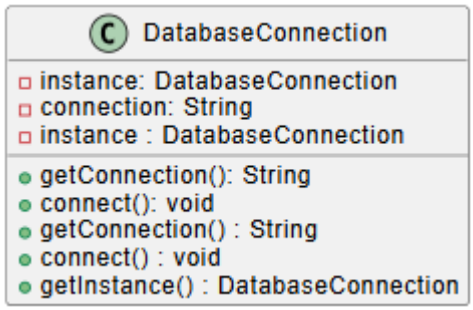
Патерн **Singleton** гарантує, що клас має лише один екземпляр, і надає глобальну точку доступу до цього екземпляра. Цей патерн корисний, коли потрібно контролювати доступ до єдиного ресурсу або об’єкта, наприклад, підключення до бази даних.

Singleton можна використати для створення єдиного екземпляра об’єкта, відповідального за підключення до бази даних. Це гарантує, що всі операції з базою даних виконуються через один об’єкт з єдиним підключенням, що зменшує витрати на ресурси та запобігає конфліктам при спробах виконати кілька паралельних запитів до бази даних.

Псевдокод реалізації шаблону:

```
class Database:

    _instance = None


    def __new__(cls):

        if cls._instance is None:

            cls._instance = super(Database, cls).__new__(cls)
```

```
            cls._instance.connection = cls.connect()
        return cls._instance

    @staticmethod
    def connect():
        return "Database connection established"

    def query(self, sql):
        print(f"Executing query: {sql}")
```




```
}
```

Рис. 1. Діаграма шаблону Одинак для класу підєднання до бази даних

Патерн **Observer** дозволяє одному об’єкту (суб’єкту) повідомляти інших об’єктів (спостерігачів) про зміни в його стані. Спостерігачі можуть підписатися на події в об’єкті і отримувати актуальні дані, коли ці події відбуваються.

Observer є ідеальним для відстеження змін статусу книг у каталозі. Наприклад, коли статус книги змінюється (доступна, забронювана, на ремонті), спостерігачі (адміністратори, бібліотекарі, користувачі) можуть отримувати актуальну інформацію та відповідно реагувати. Це дозволяє усім зацікавленим сторонам бути в курсі подій і швидко діяти, що особливо корисно для управління орендою, бронюванням та обробкою книг.

Псевдокод реалізації шаблону:

```
from abc import ABC, abstractmethod

# Абстрактний клас для спостерігача
class Observer(ABC):
    @abstractmethod
    def update(self, book_title: str, new_status: str):
        pass

# Клас для конкретного спостерігача (адміністратора)
class AdminObserver(Observer):
    def update(self, book_title: str, new_status: str):
        print(f"Admin received update: The status of the book '{book_title}' has changed to '{new_status}'")

# Клас для конкретного спостерігача (бібліотекаря)
class LibrarianObserver(Observer):
    def update(self, book_title: str, new_status: str):
        print(f"Librarian received update: The status of the book '{book_title}' has changed to '{new_status}'")

# Клас для конкретного спостерігача (звичайного користувача)
class UserObserver(Observer):
    def update(self, book_title: str, new_status: str):
        print(f"User received update: The status of the book '{book_title}' has changed to '{new_status}'")
```

```
# Абстрактний клас для суб'єкта
class Subject(ABC):
    @abstractmethod
    def attach(self, observer: Observer):
        pass

    @abstractmethod
    def detach(self, observer: Observer):
        pass

    @abstractmethod
    def notify(self):
        pass

# Клас для книги (суб'єкта)
class Book(Subject):
    def __init__(self, title: str):
        self._observers = []
        self._title = title
        self._status = "available"  # Початковий статус книги

    # Додаємо спостерігача
    def attach(self, observer: Observer):
```

```
        self._observers.append(observer)

    # Видаляємо спостерігача
    def detach(self, observer: Observer):
        self._observers.remove(observer)

    # Сповіщаємо всіх спостерігачів про зміни
    def notify(self):
        for observer in self._observers:
            observer.update(self._title, self._status)

    # Оновлення статусу книги
    def change_status(self, new_status: str):
        print(f"Changing status of book '{self._title}' to '{new_status}'")
        self._status = new_status
        self.notify()

# Використання
if __name__ == "__main__":
    # Створення книги
    book = Book("The Great Gatsby")

    # Створення спостерігачів
    admin = AdminObserver()
```

```
librarian = LibrarianObserver()
user = UserObserver()

# Додавання спостерігачів до книги
book.attach(admin)
book.attach(librarian)
book.attach(user)

# Зміна статусу книги
book.change_status("reserved")  # Книга заброньована
book.change_status("checked out")  # Книга позичена
book.change_status("available")  # Книга знову доступна

# Видалення спостерігача (наприклад, користувача)
book.detach(user)

# Зміна статусу книги після видалення користувача
book.change_status("under repair")  # Книга на ремонті
```

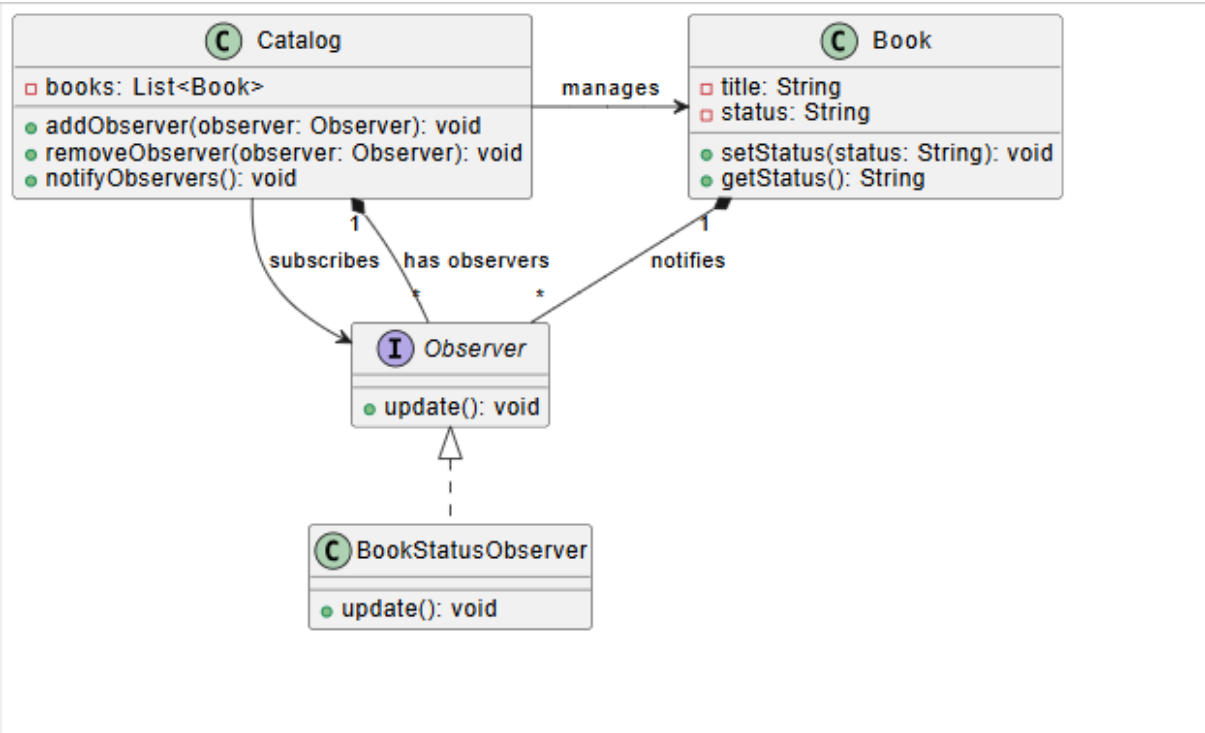


Рис. 2. Діаграма шаблону Observer

Будівельник — це породжувальний патерн проектування, що дає змогу створювати складні об’єкти крок за кроком. Будівельник дає можливість використовувати один і той самий код будівництва для отримання різних відображень об’єктів.

Об’єкти класу BookFilter створюються поступово. Для того, щоб не навантажувати пам’ять програми тимчасовими об’єктами, необхідними для створення екземпляру класу BookFilter та не порушувати принципу інкапсуляції, зручно використовувати породжувальний патерн проектування Будівельник.

Псевдокод реалізації шаблону:

```
class BookFilter:

    def __init__(self, genre=None, author=None, year=None, status=None):

        self.genre = genre

        self.author = author

        self.year = year
```

```
        self.status = status

    def __str__(self):
        return f"Genre: {self.genre}, Author: {self.author}, Year: {self.year}, Status: {self.status}"

class BookFilterBuilder:
    def __init__(self):
        self._filter = BookFilter()

    def set_genre(self, genre):
        self._filter.genre = genre
        return self

    def set_author(self, author):
        self._filter.author = author
        return self

    def set_year(self, year):
        self._filter.year = year
        return self

    def set_status(self, status):
        self._filter.status = status
        return self

    def build(self):
        return self._filter
```

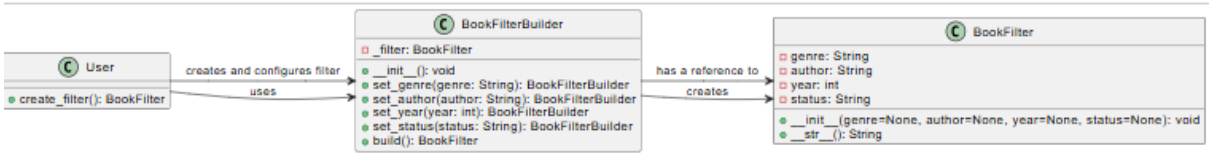


Рис. 3. Діаграма шаблону Builder

Патерн **Factory Method** визначає інтерфейс для створення об’єкта, але дозволяє підкласам змінювати тип створюваного об’єкта. Це дозволяє створювати об’єкти, не вказуючи конкретний клас, який має бути створений.

В системі керування бібліотекою **Factory Method** можна застосувати для створення різних типів користувачів (адміністратор, бібліотекар, користувач). У цьому випадку метод створення користувача буде абстракцією, яка дозволяє створювати об’єкти для кожної ролі з відповідними правами та обов'язками. Це дозволяє централізовано управляти процесом створення користувачів і забезпечує гнучкість у розширенні типів користувачів у майбутньому.

Псевдокод реалізації шаблону:

```
from abc import ABC, abstractmethod


class Book(ABC):
    @abstractmethod
    def info(self):
        pass


class PhysicalBook(Book):
```

```
    def info(self):
        return "This is a physical book."

class EBook(Book):
    def info(self):
        return "This is an electronic book."

class BookFactory:
    @staticmethod
    def create_book(book_type):
        if book_type == "physical":
            return PhysicalBook()
        elif book_type == "ebook":
            return EBook()
        else:
            raise ValueError("Unknown book type.")

# Використання
physical_book = BookFactory.create_book("physical")
ebook = BookFactory.create_book("ebook")
print(physical_book.info())
print(ebook.info())
```

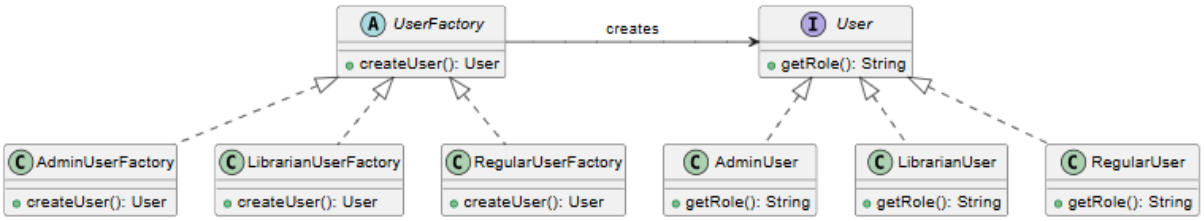


Рис. 4. Діаграма шаблону Factory

ВИСНОВКИ

У ході виконання лабораторної роботи ми ознайомилися з основними шаблонами проектування, навчилися застосовувати їх при проектуванні і розробці ПЗ.