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 №6

Тема **«Шаблони «Abstract Factory», «Factory Method», «Memento», «Observer», «Decorator»»**

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 «Abstract Factory», «Factory Method», «Memento», «Observer» і «Decorator», для створення ефективних і гнучких програмних рішень. Ця лабораторна робота спрямована на розвиток навичок у використанні різних патернів для вирішення задач у розробці програмного забезпечення.

Теоретичні відомості

Шаблони з даної лабораторної роботи допомагають вирішувати типові задачі у розробці програмного забезпечення, забезпечуючи структурованість, масштабованість та легкість підтримки. Основою їх застосування є принципи SOLID:

- **Single Responsibility Principle:** клас має виконувати лише одну логічну задачу, спрощуючи зміну або розширення його функціональності.

- **Open-Closed Principle:** класи повинні бути відкритими для розширення, але закритими для змін, тобто нову поведінку додають через спадкування чи композицію.

- **Liskov Substitution Principle:** підкласи мають коректно працювати у будьякому контексті, де використовується базовий клас.

- **Interface Segregation Principle:** інтерфейси слід розбивати на дрібні, вузько спрямовані частини, щоб уникати надлишкової реалізації.

- **Dependency Inversion Principle:** модулі вищого рівня не повинні залежати від модулів нижчого рівня, обидва мають залежати від абстракцій.

Розуміння цих принципів є основою правильного застосування шаблонів і створення якісного програмного забезпечення. **Abstract Factory**

Шаблон Abstract Factory дозволяє створювати цілі сімейства взаємопов'язаних об'єктів без вказання їхніх конкретних класів. Його основною ідеєю є надання інтерфейсу, який приховує процес створення об'єктів і робить систему гнучкішою до змін. Наприклад, цей шаблон можна використовувати для розробки графічного інтерфейсу, який має підтримувати 4 різні операційні системи. У цьому випадку фабрика створює набори компонентів, таких як кнопки чи текстові поля, які виглядають і поводяться відповідно до конкретної платформи. Ця гнучкість дозволяє легко замінювати реалізацію без змін у бізнес-логіці програми.

Factory Method

Factory Method забезпечує можливість створювати об'єкти через виклики фабричних методів, які визначаються у підкласах. Це дозволяє класам делегувати створення об'єктів, не прив'язуючись до конкретних реалізацій. Наприклад, у додатку для роботи з повідомленнями можуть бути різні типи повідомлень, такі як SMS, електронна пошта або push-сповіщення. Factory Method дозволяє створювати ці повідомлення залежно від контексту, не змінюючи клієнтський код. Це значно полегшує додавання нових типів повідомлень і робить код більш масштабованим.

Memento

Шаблон Memento використовується для збереження стану об'єкта, щоб його можна було відновити без порушення інкапсуляції. Його основна ідея полягає в тому, щоб створити "знімок" поточного стану об'єкта, який може бути збережений і використаний пізніше для відновлення. Наприклад, у текстових редакторах реалізується функціональність скасування і повтору дій за допомогою цього шаблону. Стан документа зберігається перед виконанням змін, і якщо користувач вирішить скасувати дію, система повертає попередній стан. Цей підхід дозволяє легко управляти змінами без змішування логіки управління станом із бізнес-логікою.

Observer

Observer, або "Спостерігач", дозволяє об'єкту-спостерігачу автоматично отримувати повідомлення про зміну стану іншого об'єкта. Це забезпечує динамічну залежність між об'єктами, дозволяючи їм взаємодіяти без жорсткого зв'язування. Наприклад, цей шаблон використовується у додатках новин, де підписники автоматично отримують оновлення, коли з’являється новина. Суб'єкт, у даному випадку сервер новин, повідомляє всіх підписників про зміни. Такий підхід значно спрощує процес синхронізації та забезпечує 5 легке додавання або видалення спостерігачів.

Decorator

Decorator дозволяє динамічно додавати нову поведінку об'єктам, зберігаючи їхню структуру незмінною. Це досягається шляхом обгортання об'єкта в інший клас, який реалізує додаткову функціональність. Наприклад, у графічних редакторах базовий об'єкт зображення може бути обгорнутий декоратором, який додає ефект тіні, рамки чи іншого візуального елементу. Decorator особливо корисний, коли потрібно розширити функціональність, уникаючи створення великої кількості підкласів для кожного можливого поєднання функцій.

Хід роботи

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

Тема: Архіватор

Як розробник, я переконаний, що для архіватора доцільніше застосувати патерн **Observer**, і зараз поясню чому.

Архіватор — це не просто набір статичних дій, як-от створення архіву конкретного типу (завдання, яке легко вирішує Factory Method). Це складна система, де події й динамічні зміни відбуваються постійно: додавання/видалення файлів, перевірка контрольної суми, тестування на пошкодження, оновлення метаданих, або розбиття архіву на частини. Важливо мати механізм, який дозволяє реагувати на ці зміни без необхідності переписувати всю архітектуру.

Чому Observer краще?

1. **Подійно-орієнтована природа задачі**
   У роботі архіватора постійно виникають події, на які потрібно реагувати динамічно:
   - Додавання файлу до архіву — це сигнал для перевірки контрольної суми або оновлення метаданих.
   - Видалення файлу може ініціювати автоматичне оновлення структури архіву чи перевірку його цілісності.
   - Тестування архіву на пошкодження може викликати ланцюг перевірок інших залежних компонентів.

   Observer дозволяє кожному компоненту реагувати на зміни без прив'язки до інших, що значно спрощує реалізацію.

2. **Модульність і масштабованість**
   Якщо в майбутньому потрібно додати новий функціонал (наприклад, додати функцію автоматичного резервного копіювання або оновлення архівів), це можна зробити, просто підключивши нового "спостерігача".
   Factory Method не має механізму для реагування на події, а додаючи нові функції, довелося б змінювати код створення об'єктів.

3. **Слабка зв'язаність компонентів**
   Архіватор складається з багатьох модулів, наприклад:
   - Модуль управління файлами.
   - Модуль перевірки контрольних сум.
   - Модуль відображення результатів.

   Завдяки Observer ці модулі працюють незалежно. Якщо змінюється логіка одного з них (наприклад, ви додаєте новий алгоритм перевірки), інші компоненти не потрібно переписувати. Factory Method же створює сильні залежності між об'єктами.

4. **Реакція в реальному часі**
   Уявімо ситуацію: користувач додав великий файл у архів. Observer дозволяє одразу сповістити відповідні модулі про зміни, запускати обробку файлу у фоновому режимі та оновлювати інтерфейс у реальному часі.
   Factory Method працює лише на етапі створення об'єкта і не дозволяє динамічно керувати станом системи.
5. **Забезпечення асинхронної роботи**
   Observer дозволяє налаштувати асинхронне сповіщення спостерігачів. Наприклад, перевірка цілісності архіву та оновлення метаданих можуть виконуватись одночасно, поки користувач працює з інтерфейсом.

Чому Factory Method тут недоречний?

Factory Method підходить, коли основна задача системи — створення об'єктів із певними характеристиками (наприклад, архівів різних форматів). Але архіватор — це не лише створення, а й керування складною динамікою роботи. Використання Factory Method лише додасть зайвого коду, адже він не вирішує питання взаємодії між компонентами та реакції на події.

Реалізація

1. **Клас** Observable: Відповідає за додавання, видалення та сповіщення спостерігачів.
2. **Клас** Observer: Інтерфейс для спостерігачів, який визначає метод update.
3. **Клас** ArchiveProcessObserver: Реалізація Observer, яка отримує сповіщення та виводить їх у термінал.
4. **Клас** Archiver: Реалізує функціонал архіватора з використанням шаблону **Observer** для сповіщень.

src > observable.py > ...

```python
class Observable:
    def __init__(self):
        self.observers = []

    def add_observer(self, observer):
        if observer not in self.observers:
            self.observers.append(observer)

    def remove_observer(self, observer):
        if observer in self.observers:
            self.observers.remove(observer)

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

Рис 1. - Створення класу Observable, який дозволяє додавати, видаляти та сповіщати наглядачів про зміни

src > observer.py > ...

```python
class Observer:
    def update(self, message):
        pass

```

Рис 2. - Створення базового класу Observer, який визначає інтерфейс для отримання сповіщень

src > archive_process_observer.py > ...

```python
from observer import Observer

class ArchiveProcessObserver(Observer):
    def update(self, message):
        print(f"Notification received: {message}")

```

Рис 3. - Реалізація класу ArchiveProcessObserver, який отримує сповіщення від архіватора і виводить їх у термінал

src > archiver.py > ...

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

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

    def create_archive(self, file_name):
        self.adapter.create_archive(file_name)
        self.notify_observers(f"Archive {file_name} created")

    def add_file(self, file_path):
        import os
        if os.path.exists(file_path) and 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")
        elif file_path in self.files_added:
            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 hasattr(self.adapter, 'close'):
            self.adapter.close()
        self.notify_observers("Archive process completed")

    def extract_archive(self, archive_path, destination_folder):
        self.adapter.extract_archive(archive_path, destination_folder)
        self.notify_observers(f"Archive {archive_path} successfully extracted to {destination_folder}")

```

Рис 4. - Реалізація класу Archiver, який успадковує Observable і використовує фабрику для створення адаптерів архівів, а також сповіщає наглядачів про створення та розпакування архівів

src > gui.py > ArchiverApp > add_files_to_archive

```python
import tkinter as tk
from tkinter import filedialog, messagebox
from tkinter import ttk
from archiver import Archiver
import tkinter.simpledialog

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

    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)

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

    def notify_observers(self, message):
        for observer in self.observers:
            observer.update(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.archiver = Archiver(archive_type)
                    for observer in self.observers:
                        self.archiver.add_observer(observer)
                    self.notify_observers(f"Archive {archive_name} created")
                    files_to_add = filedialog.askopenfilenames(title="Виберіть файли для додавання в архів")
                    if files_to_add:
                        for file_path in files_to_add:
                            self.archiver.add_file(file_path)
```

```
src > gui.py > ArchiverApp > add_files_to_archive
  7  class ArchiverApp:
 35      def create_archive(self):
 53                  except Exception as e:
 54                      self.notify_observers(f"Error: {str(e)}")
 55                      messagebox.showerror("Помилка", str(e))
 56
 57      def extract_archive(self):
 58          file_path = filedialog.askopenfilename(filetypes=[("Всі архіви", "*.zip;*.tar.gz;*.rar")])
 59          if file_path:
 60              destination_folder = filedialog.askdirectory()
 61              if destination_folder:
 62                  try:
 63                      self.archiver = Archiver(file_path.split('.')[-1])
 64                      for observer in self.observers:
 65                          self.archiver.add_observer(observer)
 66                      self.notify_observers(f"Extracting archive {file_path}")
 67                      self.archiver.extract_archive(file_path, destination_folder)
 68                      self.notify_observers(f"Archive {file_path} extraction completed successfully")
 69                      messagebox.showinfo("Успіх", f"Архів успішно розпаковано до {destination_folder}.")
 70                  except Exception as e:
 71                      self.notify_observers(f"Error: {str(e)}")
 72                      messagebox.showerror("Помилка", str(e))
 73
 74      def add_files_to_archive(self):
 75          archive_path = filedialog.askopenfilename(title="Виберіть архів", filetypes=[("Всі архіви", "*.zip;*.tar.gz;*.rar")])
 76          if archive_path:
 77              files_to_add = filedialog.askopenfilenames(title="Виберіть файли для додавання")
 78              if files_to_add:
 79                  try:
 80                      self.archiver = Archiver(archive_path.split('.')[-1])
 81                      for observer in self.observers:
 82                          self.archiver.add_observer(observer)
 83                      self.notify_observers(f"Adding files to archive {archive_path}")
 84                      for file_path in files_to_add:
 85                          self.archiver.add_file(file_path)
 86                      self.archiver.close()
 87                      self.notify_observers("Files added to archive successfully")
 88                      messagebox.showinfo("Успіх", f"Файли успішно додано до архіву {archive_path}.")
 89                  except Exception as e:
 90                      self.notify_observers(f"Error: {str(e)}")
 91                      messagebox.showerror("Помилка", str(e))
 92
 93      def get_archive_type(self):
 94          archive_type = tkinter.simpledialog.askstring("Тип архіву", "Введіть тип архіву (zip, rar, tar.gz):")
 95          if archive_type in ['zip', 'rar', 'tar.gz']:
 96              return archive_type
 97          else:
 98              messagebox.showerror("Помилка", "Невірний тип архіву.")
 99              return None
```

Рис 5-6. - Графічний інтерфейс для архіватора, який дозволяє користувачам створювати та розпаковувати архіви, а також сповіщати наглядачів про ці події

```
(venv) PS C:\Users\lubch\Desktop\TRPZ_labs_Liubchenko_IA24\lab4trpz> python src\main.py
User user123 logged in with cookie user123
Notification received: Archive C:/Users/lubch/Desktop/1.zip created
Notification received: File C:/Users/lubch/Desktop/1.txt added to archive
Notification received: Archive process completed
Notification received: Archive creation completed
```

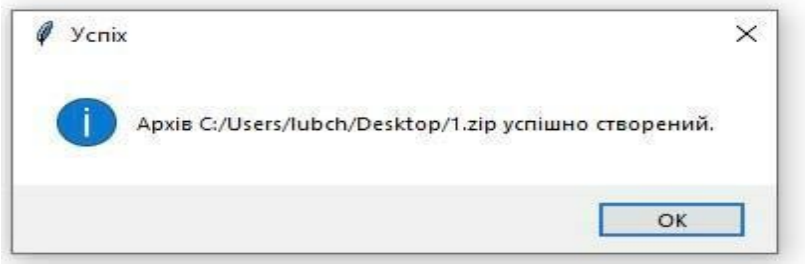

Рис 7. – Сповіщення про успішне створення архіву

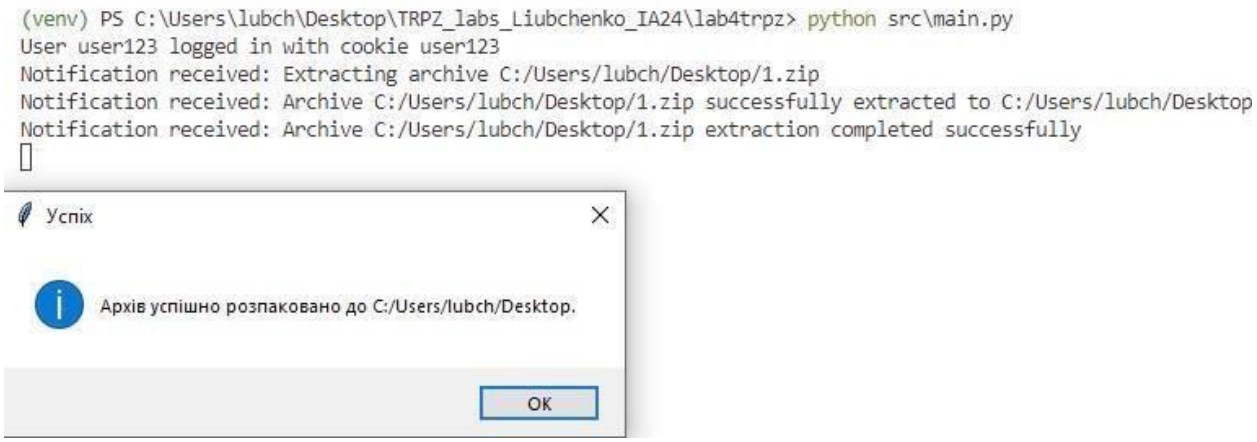

Рис 8. – Сповіщення про успішне розпакування архіву

Висновок: Виконання цієї лабораторної роботи з імплементацією шаблону Observer допомогло мені краще зрозуміти, як реалізовувати розподілену архітектуру сповіщень між об'єктами у складних системах. У результаті я зміг використати цей шаблон для створення функціоналу сповіщень у своєму проекті архіватора.

Особливо цікаво було розібратися в ролі кожного елемента — Observable, Observer та конкретні реалізації наглядачів. Кожен із них чітко виконує свою задачу, і це дозволяє зробити систему не тільки ефективною, але й зрозумілою для підтримки та розвитку.

Також робота змусила мене замислитися над тим, як правильно організувати взаємодію між об'єктами, які спостерігають за змінами, і об'єктами, що їх ініціюють. Це показало важливість балансу між гнучкістю і ефективністю комунікації. У реальних проєктах, де необхідна швидка і надійна реакція на події, шаблон Observer може вимагати додаткової оптимізації чи засобів синхронізації.

У процесі я також переконався, наскільки важливо використовувати готові патерни проектування. Вони допомагають уникнути повторення коду і спрощують роботу з проектом у довгостроковій перспективі. Загалом, робота не тільки розширила мої знання, а й додала впевненості у застосуванні шаблонів для розв'язання реальних задач.