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

Лабораторна робота №6

з дисципліни «Технології розроблення програмного забезпечення»

Виконала:
студентка групи ІА-23
Проценко. В. І.

Перевірив:
Мягкий М. Ю

Київ 2024

Зміст

Тема: Шаблони «Abstract Factory», «Factory Method», «Memento», «Observer», «Decorator».

Завдання:

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.

1. Короткі теоретичні відомості

Принципи проектування SOLID

S	SRP	Принцип єдиного обов'язку На кожен об'єкт має бути покладений один єдиний обов'язок.
O	OCP	Принцип відкритості/закритості Програмні сутності мають бути відкриті для розширення, але закриті для зміни.
L	LSP	Принцип підстановки Барбари Лісков Об'єкти в програмі можуть бути замінені їх спадкоємцями без зміни властивостей програми.
I	ISP	Принцип розділення інтерфейсу Багато спеціалізованих інтерфейсів краще, ніж один універсальний.
D	DIP	Принцип інверсії залежностей Залежності усередині системи будуються на основі абстракцій. Модулі верхнього рівня не залежать від модулів нижнього рівня. Абстракції не повинні залежати від деталей. Деталі повинні залежати від абстракцій.

Розглянемо декілька шаблонів:

1. Шаблон «Abstract Factory».
   Шаблон «Abstract Factory» використовується для створення сімейств об'єктів без вказівки їх конкретних класів. Для цього виноситься загальний інтерфейс фабрики (AbstractFactory) і створюються його реалізації для різних сімейств продуктів.
   Хорошим прикладом використання абстрактної фабрики є ADO.NET: існує загальний клас DbProviderFactory, здатний створювати об'єкти типів DbConnection, DbDataReader, DbAdapter та ін.; існують реалізації цих фабрик і об'єктів - SqlProviderFactory, SqlConnection, SqlDataReader, SqlAdapter і так далі. Відповідно, якщо додатку необхідно працювати з різними базами даних , то досить використати базові реалізації (Db.) і підставити відповідну фабрику у момент ініціалізації фабрики (Factory = new SqlProviderFactory()).
   Цей шаблон передусім структурує знання про схожі об'єкти (що називаються сімействами, як класи для доступу до БД) і створює можливість взаємозаміни різних сімейств (робота з Oracle ведеться також, як і робота з SQL Server). Проте, при використанні такої схеми украй незручно розширювати фабрику - для додавання нового методу у фабрику необхідно додати його в усіх фабриках і створити відповідні класи, що створюються цим методом.
   + Гарантує поєднання створюваних продуктів.
   + Звільняє клієнтський код від прив’язки до конкретних класів продукту.
   + Реалізує принцип відкритості/закритості.
   - Вимагає наявності всіх типів продукту в кожній варіації.
   - Ускладнює код програми внаслідок введення великої кількості додаткових класів.

2. Шаблон «Factory Method».
   Шаблон "фабричний метод" визначає інтерфейс для створення об'єктів певного базового типу. Це зручно, коли хочеться додати можливість створення об'єктів не базового типу, а деякого дочірнього. Фабричний метод у такому разі є зачіпкою для впровадження власного конструктора об'єктів. Основна ідея полягає саме в заміні об'єктів їх підтипами, що при цьому зберігає ту ж функціональність; інша частина поведінки об'єктів не є інтерфейсною (AnOperation) і дозволяє взаємодіяти із створеними об'єктами як з об'єктами базового типу. Тому шаблон "фабричний метод" носить ще назву "Віртуальний конструктор".
   Розглянемо простий приклад. Нехай наше застосування працює з мережевими драйвер-мі і використовує клас Packet для зберігання даних, що передаються в

мережу. Залежно від використовуваного протоколу, існує два перевантаження - TcpPacket, UdpPacket. І відповідно два створюючі об'єкти (TcpCreator, UdpCreator) з фабричним методом (який створює відповідні реалізації). Проте базова функціональність (передача пакету, прийом пакету, заповнення пакету даними) нічим не відрізняється один від одного, відповідно поміщається у базовий клас PacketCreator. Таким чином поведінка системи залишається тим же, проте з'являється можливість підстановки власних об'єктів в процес створення і роботи з пакетами.

+ Позбавляє клас від прив’язки до конкретних класів продуктів.

+ Виділяє код виробництва продуктів в одне місце, спрощуючи підтримку коду.

+ Спрощує додавання нових продуктів до програми.

- Може призвести до створення великих паралельних ієрархій класів.

3. Шаблон «Memento» .

   Шаблон використовується для збереження і відновлення стану об'єктів без порушення інкапсуляції. Об'єкт "мементо" служить виключно для збереження змін над початковим об'єктом (Originator). Лише початковий об'єкт має можливість зберігати і отримувати стан об'єкту "мементо" для власних цілей, цей об'єкт є "порожнім" для кого-небудь ще. Об'єкт Caretaker використовується для передачі і зберігання мементо об'єктів в системі. Таким чином вдається досягти наступних цілей:

   - Зберігання стану повністю відділяється від початкових об'єктів, що полегшує їх реалізацію;

   - Передача об'єктів мементо лягає на плечі Caretaker об'єктів, що дозволяє гнучкіше управляти станами об'єктів і спростити дизайн класів початкових об'єктів;

   - Збереження і відновлення стану реалізовані у вигляді двох простих методів і є закритими для кого-небудь ще окрім початкових об'єктів, таким чином не порушуючи інкапсуляцію.

   Шаблон "мементо" дуже зручно використати разом з шаблоном "команда" для реалізації "скасовних" дій - дані про дію зберігаються в мементо, а команда має можливість відновити початкове положення відповідних об'єктів.

   + Не порушує інкапсуляцію вихідного об’єкта.

   + Спрощує структуру вихідного об’єкта. Не потрібно зберігати історію версій свого стану.

   - Вимагає багато пам’яті, якщо клієнти дуже часто створюють знімки.

   - Може спричинити додаткові витрати пам’яті, якщо об’єкти, що зберігають історію, не звільняють ресурси, зайняті застарілими знімками.

4. Шаблон «Observer».

   Шаблон визначає залежність "один-до-багатьох" таким чином, що коли один об'єкт змінює власний стан, усі інші об'єкти отримують про це сповіщення і мають можливість змінити власний стан також.

   Розглянемо цей шаблон на прикладі. Припустимо, є деяка банківська система і декілька користувачів переглядають баланс на рахунку пана И. У цей момент пан И. кладе на свій рахунок деяку суму, яка міняє загальний баланс. Кожен з користувачів, що переглядали баланс, отримує про це звістку (для користувачів ця звістка може бути прозорою - просто зміна цифр, або попередження про те, що баланс змінився). Раніше неможливі дії для користувачів (переведення до іншої категорії клієнтів і тому подібне) стають доступними. Цей шаблон дуже широко поширений в шаблоні MVVM і механізмі "прив'язок" (bindings) в WPF і частково в WinForms. Інша назва шаблону - підписка/розсилка. Кожен з оглядачів власноручно підписується на зміни конкретного об'єкту, а об'єкти зобов'язані сповіщати своїх передплатників про усі свої зміни (на даний момент конкретних механізмів автоматичного сповіщення про зміну стану в .NET мовах не існує).

   \+ Ви можете підписувати і відписувати одержувачів «на льоту».

   \+ Видавці не залежать від конкретних класів підписників і навпаки.

   \+ Реалізує принцип відкритості/закритості.

   \- Підписники сповіщуються у випадковій послідовності.

5. Шаблон «Decorator».

   Шаблон призначений для динамічного додавання функціональних можливостей об'єкту під час роботи програми. Декоратор деяким чином "обертає" початковий об'єкт зі збереженням його функцій, проте дозволяє додати додаткові дії. Такий шаблон надає гнучкіший спосіб зміни поведінки об'єкту чим просте спадкоємство, оскільки початкова функціональність зберігається в повному об'ємі. Більше того, таку поведінку можна застосовувати до окремих об'єктів, а не до усієї системи в цілому. Простим прикладом є накладення смуги прокрутки до усіх візуальних елементів. Кожен об'єкт, який може прокручуватися, обертається в " прокручуваному" елементі, і при необхідності з'являється смуга прокрутки. Початкові функції елементу (наприклад, рядки статусу) залишаються незмінними.

   \+ Дозволяє мати кілька дрібних об’єктів, замість одного «на всі випадки».

   \+ Дозволяє додавати обов’язки «на льоту».

   \+ Більша гнучкість, ніж у спадкування.

   \- Велика кількість крихітних класів.

   \- Важко конфігурувати об’єкти, які загорнуто в декілька обгорток одночасно.

2. Хід роботи

Пригадаємо обрану індивідуальну тему для виконання лабораторних робіт.

..17 System activity monitor (iterator, command, abstract factory, bridge, visitor, SOA)

Монітор активності системи повинен зберігати і запам'ятовувати статистику використовуваних компонентів системи, включаючи навантаження на процесор, обсяг займаної оперативної пам'яті, натискання клавіш на клавіатурі, дії миші (переміщення, натискання), відкриття вікон і зміна вікон; будувати звіти про використання комп'ютера за різними критеріями (% часу перебування у веб-браузері, середнє навантаження на процесор по годинах, середній час роботи комп'ютера по днях і т.д.); правильно поводитися з «простоюванням» системи – відсутністю користувача.

2.1. Реалізація класів відповідно до обраної теми.

Реалізуємо ще два класи-монітори MouseMonitor та KeyboardMonitor та вносимо мінімальні зміни у інші залежні класи для забезпечення виведення до інтерфейсу даних про рух та активність миші та клавіатури.

Оновимо клас основного монітору ActivityMonitor та додамо у інтерфейс висвітлення даних про активність миші та клавіатури.

Описану вище частину функціональності можна переглянути у спільному репозиторії, у папці system-activity-monitor за посиланням: https://github.com/protsenkoveronika/TRPZ/tree/new-test-branch/system-activity-monitor

2.2. Застосування одного з розглянутих шаблонів.

У системі моніторингу активності комп’ютера шаблон "Abstract factory" використовується для створення різних варіантів моніторів системних ресурсів, таких як процесор, пам'ять, активні вікна, миша та клавіатура.

```python
class ActivityMonitor:
    def __init__(self, db_file):
        self.db_file = db_file
        self.report = Report(self.db_file)
        self.current_index = 0
        self.is_monitoring = False
        self.is_active = False
        self.factory = ConcreteWindowsFactory()

        self.root = Tk()
        self.root.title("Activity Monitor")

        self.gui_vars = {…

        self.monitors = [
            self.factory.create_processor_monitor(self.db_file, self.gui_vars["cpu_usage"]),
            self.factory.create_memory_monitor(self.db_file, self.gui_vars["memory_usage"]),
            self.factory.create_window_monitor(self.db_file, self.gui_vars["active_window"]),
            self.factory.create_mouse_monitor(self.gui_vars["mouse_position"]),
            self.factory.create_keyboard_monitor(self.gui_vars["keyboard_activity"]),
        ]
```

Рисунок 1.1 - Ініціація моніторів через фабрику у класі ActivityMonitor

Абстрактна фабрика визначає методи для створення моніторів різних типів (наприклад, ProcessorMonitor, MemoryMonitor), не вказуючи їх конкретних класів. Конкретні фабрики (наприклад, ConcreteWindowsFactory) реалізують ці методи, створюючи монітори, специфічні для певної операційної системи (Windows, а у перспективі можливість додати і Linux).

```python
class ConcreteWindowsFactory(AbstractFactory):
    def create_processor_monitor(self, db_file, gui_var) -> AbstractProcessorMonitor:
        return ProcessorUsage(db_file, gui_var)

    def create_memory_monitor(self, db_file, gui_var) -> AbstractMemoryMonitor:
        return MemoryUsage(db_file, gui_var)

    def create_window_monitor(self, db_file, gui_var) -> AbstractWindowMonitor:
        return WindowMonitor(db_file, gui_var)

    def create_mouse_monitor(self, gui_var) -> AbstractMouseMonitor:
        return MouseMonitor(gui_var)

    def create_keyboard_monitor(self, gui_var) -> AbstractKeyboardMonitor:
        return KeyboardMonitor(gui_var)
```

Рисунок 1.2 - Клас ConcreteWindowsFactory що реалізує AbstractFactory

```python
class AbstractFactory(ABC):
    @abstractmethod
    def create_processor_monitor(self, db_file, gui_var) -> AbstractProcessorMonitor:
        pass

    @abstractmethod
    def create_memory_monitor(self, db_file, gui_var) -> AbstractMemoryMonitor:
        pass

    @abstractmethod
    def create_window_monitor(self, db_file, gui_var) -> AbstractWindowMonitor:
        pass

    @abstractmethod
    def create_mouse_monitor(self, gui_var) -> AbstractMouseMonitor:
        pass

    @abstractmethod
    def create_keyboard_monitor(self, gui_var) -> AbstractKeyboardMonitor:
        pass
```

Рисунок 1.3 - Абстрактний клас AbstractFactory

```python
class AbstractProcessorMonitor(ABC):
    @abstractmethod
    def update_widget(self):
        pass

class AbstractMemoryMonitor(ABC):
    @abstractmethod
    def update_widget(self):
        pass

class AbstractWindowMonitor(ABC):
    @abstractmethod
    def update_widget(self):
        pass

class AbstractMouseMonitor(ABC):
    @abstractmethod
    def update_widget(self):
        pass

class AbstractKeyboardMonitor(ABC):
    @abstractmethod
    def update_widget(self):
        pass
```

Рисунок 1.4 - Абстрактні класи моніторів із функцією оновлення віджетів

Таким чином, у даній системі моніторингу активності комп'ютера патерн Abstract Factory використовується без безпосереднього виклику конструктора для створення моніторів. Замість цього, фабричні методи викликаються через фабрику, яка відповідає за створення конкретних об'єктів моніторів. Це дозволяє уникнути прямого використання конструкторів у клієнтському коді та забезпечує абстракцію, яка сприяє більшій гнучкості й можливості для легкого додавання нових варіантів моніторів у систему.

3. Висновок

У ході даної лабораторної роботи було реалізовано частину функціональності системи моніторингу активності, зокрема класи пов’язані із збором даних про активність клавіатури та миші та виведенням цих даних на графічний інтерфейс користувача. Ми ознайомились із кількома шаблонами які використовують у програмуванні, у тому числі і шаблоном "Abstract factory", який було використано при реалізації даної частини проекту.