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 №4

з дисципліни «Технології розроблення програмного забезпечення»

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

Перевірив:
Мягкий М. Ю

Київ 2024

Зміст

Тема: Шаблони «SINGLETON», «ITERATOR», «PROXY», «STATE», «STRATEGY»

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

Застосування патернів проектування підвищує стійкість системи до зміни вимог та спрощує неминуче подальше доопрацювання системи. Крім того, важко переоцінити роль використання патернів при інтеграції інформаційних систем організації. Також слід зазначити, що сукупність патернів проектування, по суті, являє собою єдиний словник проектування, який, будучи уніфікованим засобом, незамінний для спілкування розробників один одним. Таким чином шаблони представляють собою, підтверджені роками розробок в різних компаніях і на різних проектах, «ескізи» архітектурних рішень, які зручно застосовувати у відповідних обставинах.

Відповідне використання патернів проектування дає розробнику ряд незаперечних переваг:

- модель системи, побудована в межах патернів проектування, фактично є структурованим виокремленням тих елементів і зв’язків, які значимі при вирішенні поставленого завдання.
- модель, побудована з використанням патернів проектування, більш проста і наочна у вивченні, ніж стандартна модель.
- не дивлячись на простоту і наочність, вона дозволяє глибоко і всебічно опрацювати архітектуру розроблюваної системи з використанням спеціальної мови.
- зменшення трудовитрат і часу на побудову архітектури.
- надання проектованій системі необхідних якостей (гнучкість, адаптованість, ін.).
- зменшення накладних витрат на подальшу підтримку системи.

Розглянемо декілька шаблонів.

1. Шаблон «Singleton».
   «Singleton» (Одинак) являє собою клас в термінах ООП, який може мати не більше одного об'єкта. Насправді, кількість об'єктів можна задати (тобто не можна створити більш n об'єктів даного класу). Даний об'єкт найчастіше зберігається як статичне поле в самому класі.
   Однак слід зазначити, що в даний час «одинака» багато хто вважає «анти-шаблоном», тобто поганою практикою проектування. Це пов'язано з тим, що «одинаки» представляють собою глобальні дані, що мають стан. Стан глобальних об'єктів важко відслідковувати і підтримувати коректно, а також глобальні об'єкти важко тестуються і вносять складність в програмний код. При цьому реалізація контролю доступу можлива за допомогою статичних змінних, замикань, мютексов та інших спеціальних структур.
   + Гарантує наявність єдиного екземпляра класу.
   + Надає до нього глобальну точку доступу.
   - Порушує принцип єдиної відповідальності класу.
   - Маскує поганий дизайн

2. Шаблон «Iterator».
   «Iterator» являє собою шаблон реалізації об'єкта доступу до набору (колекції, агрегату) елементів без розкриття внутрішніх механізмів реалізації. Ітератор виносить функціональність перебору колекції елементів з самої колекції, таким чином досягається розподіл обов'язків: колекція відповідає за зберігання даних, ітератор - за прохід по колекції.
   При цьому алгоритм ітератора може змінюватися - при необхідності пройти в зворотньому порядку використовується інший ітератор. Тобто, шаблон ітератор дозволяє реалізовувати різноманітні способи проходження по колекції незалежно від виду і способу представлення даних в колекції.
   Цей шаблон дозволяє уніфікувати операції проходження по наборам об'єктів для всіх наборів. Тобто, незалежно від реалізації (масив, зв'язаний список, незв'язаний список, дерево та ін.), кожен з наборів може використовувати будь-який з реалізованих ітераторів.
   + Дозволяє реалізувати різні способи обходу структури даних.
   + Спрощує класи зберігання даних
   - Не виправданий, якщо можна обійтися простим циклом.

3. Шаблон «Proxy».

   У «Proxy» об'єкти є об'єктами-заглушками або двійниками/замінниками для об'єктів конкретного типу. Зазвичай, проксі об'єкти вносять додатковий функціонал або спрощують взаємодію з реальними об'єктами. Проксі об'єкт повністю повторює визначення об'єкта, який він замінює (тобто всі доступні методи, поля і властивості); проксі об'єкт може також зберігати посилання на реальний об'єкт, в тому випадку, якщо він розширює функціональність, тобто «обгортає» реальний об'єкт.

   + Дозволяє контролювати сервісний об'єкт непомітно для клієнта.
   + Може працювати, навіть якщо сервісний об'єкт ще не створений.
   - Ускладнює код програми через введення додаткових класів.
   - Збільшує час відгуку від сервісу.

4. Шаблон «State».

   Шаблон «State» (Стан) дозволяє змінювати логіку роботи об'єктів у випадку зміни їх внутрішнього стану. Наприклад, відсоток нарахованих на картковий рахунок грошей залежить від стану картки: Visa Electron, Classic, Platinum і т.д. Або обсяг послуг, які надані хостинг компанією, змінюється в залежності від обраного тарифного плану (стану членства - бронзовий, срібний або золотий клієнт). Реалізація даного шаблону полягає в наступному: пов'язані зі станом поля, властивості, методи і дії виносяться в окремий загальний інтерфейс (State); кожен стан являє собою окремий клас (ConcreteStateA, ConcreteStateB), які реалізують загальний інтерфейс. Об’єкти, що мають стан (Context), при зміні стану просто записують новий об'єкт в поле state, що призводить до повної зміни поведінки об'єкта. Це дозволяє легко додавати в майбутньому і обробляти нові стани, відокремлювати залежні від стану елементи об'єкта в інших об'єктах, і відкрито проводити заміну стану (що має сенс у багатьох випадках).

   + Позбавляє від безлічі великих умовних операторів машини станів.
   + Концентрує в одному місці код, пов'язаний з певним станом.
   + Спрощує код контексту.
   - Може невиправдано ускладнити код, якщо станів мало і вони рідко змінюються.

5. Шаблон «Strategy».

   Шаблон «Strategy» (Стратегія) дозволяє змінювати деякий алгоритм поведінки об'єкта іншим алгоритмом, що досягає ту ж мету іншим способом. Прикладом можуть служити алгоритми сортування: кожен алгоритм має власну реалізацію і визначений в окремому класі; вони можуть бути взаємозамінними в об'єкті, який їх використовує. Даний шаблон дуже зручний у випадках, коли існують різні «політики» обробки даних. По суті, він дуже схожий на шаблон «State» (Стан), проте використовується в абсолютно інших цілях - незалежно від стану об'єкта відобразити різні можливі поведінки об'єкта (якими досягаються одні й ті самі або схожі цілі).

   + Гаряча заміна алгоритмів на льоту.

   + Ізолює код і дані алгоритмів від інших класів.

   + Відхід від спадкування до делегування.

   + Реалізує принцип відкритості / закритості.

   - Ускладнює програму за рахунок додаткових класів.

   - Клієнт повинен знати, в чому полягає різниця між стратегіями, щоб вибрати відповідну.

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 класи WindowMonitor, ProcessorUsage та MemoryUsage для виконання збору даних відповідно до вимаганого функціоналу та періодичниго збереження.

У клас основного монітору ActivityMonitor додамо висвітлення отриманих від моніторів даних у графічному інтерфейсі та запити до моніторів для отримання цих даних.

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

2.2. Застосування одного з розглянутих шаблонів.

Шаблон "Iterator" доцільно застосувати у системі моніторингу активності комп’ютера для опитування моніторів один за одним з метою виведення даних у реальному часі та елементами GUI. Це рішення виправдане у таких випадках:

- Набір моніторів може змінюватися (наприклад, можуть додаватися нові типи моніторів або змінюватися кількість існуючих). Шаблон "Iterator" дозволяє легко адаптувати систему до таких змін.
- Монітори мають спільний метод (наприклад, collect_data()), через який отримуються їхні дані. Ітератор забезпечує зручний спосіб проходження по кожному монітору незалежно від його типу.

- Віджети, що відображають дані моніторів, можуть змінюватися або доповнюватися. Iterator забезпечує спрощений обхід елементів GUI, дозволяючи працювати з ними послідовно, незалежно від змін у їх структурі.
- У разі розширення системи можна додавати нові монітори, не змінюючи логіку обходу колекції.

У даній реалізації створено клас MonitorIterator, який виконує роль ітератора. Він забезпечує послідовний доступ до кожного монітора у списку monitors. Замість використання звичайного циклу (наприклад, for monitor in monitors), застосування шаблону "Iterator" покращує модульність коду та дозволяє ізолювати логіку обходу від інших частин системи.

```
class MonitorIterator:
    def __init__(self, monitors):
        self.monitors = monitors
        self.current_index = 0

    def __iter__(self):
        self.current_index = 0
        return self

    def __next__(self):
        if self.current_index >= len(self.monitors):
            raise StopIteration
        monitor = self.monitors[self.current_index]
        self.current_index += 1
        return monitor
```

Рисунок 1.1 - Клас MonitorIterator

Також, було створено клас WidgetIterator, який виконує роль ітератора для обходу GUI-компонентів. Цей клас забезпечує послідовний доступ до кожного віджета у списку widgets, що дозволяє ефективно оновлювати відображення даних у графічному інтерфейсі. Завдяки цьому, кожен віджет може бути оброблений окремо, а логіка обходу віджетів ізольована від основної частини коду. Це дозволяє легко змінювати набір елементів інтерфейсу або додавати нові, без необхідності змінювати загальну структуру програми.

```python
class WidgetIterator:
    def __init__(self, widgets):
        self.widgets = widgets
        self.current_index = 0

    def __iter__(self):
        self.current_index = 0
        return self

    def __next__(self):
        if self.current_index >= len(self.widgets):
            raise StopIteration
        widget = self.widgets[self.current_index]
        self.current_index += 1
        return widget
```

Рисунок 1.2 - Клас WidgetIterator

Поєднання обох ітераторів сприяє покращенню модульності та масштабованості системи. MonitorIterator відповідає за організацію роботи з моніторами системної активності, такими як монітор процесора, пам’яті та активного вікна, а WidgetIterator — за управління віджетами інтерфейсу, що відображають отримані дані.

```python
def __iter__(self):
    return MonitorIterator(self.monitors)

def _monitoring_loop(self):
    while self.is_monitoring:
        for monitor in self:
            data = monitor.collect_data()
            self.data[monitor.__class__.__name__] = data

        # оновлення GUI змінних
        for widget in self.widget_iterator:
            if widget == self.widgets[0]:
                self.gui_vars["cpu_usage"].set(
                    f"CPU Usage: {self.data.get('ProcessorUsage', {}).get('cpu_usage', 'N/A')}%"
                )
            elif widget == self.widgets[1]:
                self.gui_vars["memory_usage"].set(
                    f"Memory Usage: {self.data.get('MemoryUsage', {}).get('memory_usage', 'N/A')} MB"
                )
            elif widget == self.widgets[2]:
                self.gui_vars["active_window"].set(
                    f"Active Window: {self.data.get('WindowMonitor', {}).get('active_window', 'N/A')}"
                )

        time.sleep(1)
```

Рисунок 1.3 - Фрагмент коду класу ActivityMonitor, що викликає ітератори

Таким чином, використання ітератора у цій системі є обґрунтованим і відповідає принципам об'єктно-орієнтованого дизайну, зокрема Single Responsibility Principle (одна відповідальність) і Open/Closed Principle (відкритість до розширення). Тобто дозволяє зберігати чіткий поділ обов’язків між компонентами системи, полегшуючи подальше розширення її функціональності, а також підтримку й тестування коду.

3. Висновок

У ході даної лабораторної роботи було реалізовано частину функціональності системи моніторингу активності, зокрема 3 основні класи моніторів, що збирають дані роботи системи та клас відповідальний за висвітлення даних на графічному інтерфейсі користувача. Ми ознайомились із кількома шаблонами які використовують у програмуванні, у тому числі і шаблон "Iterator", який було використано упри реалізації даної частини проекту.