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 №7

з дисципліни «Технології розроблення програмного забезпечення»

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

Перевірив:
Мягкий М. Ю

Київ 2024

Зміст

Тема: Шаблони «MEDIATOR», «FACADE», «BRIDGE», «TEMPLATE METHOD».

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

Розглянемо кілька принципів проектування:

- Принцип Don't Repeat Yourself (DRY) - принцип націлений на уникнення дублювання коду. Замість цього рекомендується створювати загальні методи, функції, класи, які можуть бути використані в різних частинах програми. Це не тільки знижує кількість коду, а й полегшує процес виявлення помилок та тестування.
- Принцип Keep it Simple, Stupid! (KISS) - принцип наголошує на тому, щоб робити системи простими і зручними для розуміння. Замість створення великих і складних рішень рекомендується розбивати задачі на невеликі, прості компоненти, кожен з яких має чітко визначену функцію. Це також дозволяє знизити ймовірність помилок і спрощує тестування.
- Принцип You only load it once! (YOLO) - принцип наголошує на необхідності оптимізації завантаження даних, що є важливим для продуктивності програм. Якщо програма постійно завантажує однакову інформацію, наприклад, з диска чи з мережі, це може призвести до значних втрат у швидкості. Тому рекомендується зберігати налаштування та конфігураційні дані в пам'яті під час запуску програми, щоб уникнути повторних операцій зчитування.
- Принцип Парето або закон 80/20, стверджує, що більшість результатів (80%) досягається завдяки невеликій частині зусиль (20%). У контексті розробки програмного забезпечення це може означати, що 80% навантаження на сервер можуть бути спричинені лише 20% функцій програми, або що основну частину помилок можна виправити за допомогою змін у невеликій частині коду. Знання цього принципу дозволяє зосередитись на найбільш важливих аспектах системи, що приносить найбільшу вигоду.

- Принцип You ain’t gonna need it (YAGNI) - принцип радить не додавати зайву функціональність, яка ймовірно не буде використовуватись. Розробники часто схильні до створення універсальних рішень, які в майбутньому можуть не знадобитись. Замість цього, необхідно концентруватися лише на реалізації тих функцій, які дійсно потрібні зараз.

Розглянемо декілька шаблонів:

1. Шаблон «Mediator».
   Шаблон «Mediator» (посередник) використовується для визначення взаємодії об'єктів за допомогою іншого об'єкта (замість зберігання посилань один на одного). Даний шаблон схожий на шаблон «команда», проте в даному випадку замість зберігання даних про конкретну дію, зберігаються дані про взаємодії між компонентами.
   Даний шаблон зручно застосовувати у випадках, коли безліч об'єктів взаємодіє між собою деяким структурованим чином, однак складним для розуміння. У такому випадку вся логіка взаємодії виноситься в окремий об'єкт. Кожен із взаємодіючих об'єктів зберігає посилання на об'єкт «медіатор».
   «Медіатор» нагадує диригента при управлінні оркестром. Диригент стежить за тим, щоб кожен інструмент грав в правильний час і в злагоді з іншими інструментами. Функції «медіатора» повністю це повторюють.
   + Усуває залежності між компонентами, дозволяючи повторно їх використовувати.
   + Спрощує взаємодію між компонентами.
   + Централізує управління в одному місці.
   - Посередник може сильно роздутися.

2. Шаблон «Facade».
   Шаблон «Facade» (фасад) передбачає створення єдиного уніфікованого способу доступу до підсистеми без розкриття внутрішніх деталей підсистеми. Оскільки підсистема може складатися з безлічі класів, а кількість її функцій - не більше десяти, то щоб уникнути створення «спагеті-коду» (коли все тісно пов'язано між собою) виділяють один загальний інтерфейс доступу, здатний правильним чином звертатися до внутрішніх деталей.
   Це також відволікає користувачів від змін в підсистемі (внутрішня реалізація може змінюватися, а наданої послуги немає), що також скоротить кількість змін в використовуваних фасад класах (без фасаду довелося б змінювати вихідні коди в безлічі точок). Звичайно, твердої умови повного закриття

внутрішніх класів підсистеми не стоїть - при необхідності можна звертатися до окремих класів безпосередньо, минаючи об'єкт фасад.

+ Ізолює клієнтів від компонентів складної підсистеми.

- Фасад ризикує стати божественним об'єктом, прив'язаним до всіх класів програми.

3. Шаблон «Bridge» .

   Шаблон «Bridge» (міст) використовується для поділу інтерфейсу і його реалізації. Це необхідно у випадках, коли може існувати кілька різних абстракцій, над якими можна проводити дії різними способами.

   + Дозволяє будувати платформо-незалежні програми.

   + Приховує зайві або небезпечні деталі реалізації від клієнтського коду.

   + Реалізує принцип відкритості / закритості.

   - Ускладнює код програми через введення додаткових класів. історію, не звільняють ресурси, зайняті застарілими знімками.

4. Шаблон «Template Method».

   Шаблон «Template Method» (шаблонний метод) дозволяє реалізувати покроково алгоритм в абстрактному класі, але залишити специфіку реалізації подклассам. Можна привести в приклад формування веб-сторінки: необхідно додати заголовки, вміст сторінки, файли, що додаються, і нижню частину сторінки. Код для додавання вмісту сторінки може бути абстрактним і реалізовуватися в різних класах - AspNetCompiler, HtmlCompiler, PhpCompiler і т.п. Додавання всіх інших елементів виконується за допомогою вихідного абстрактного класу з алгоритмом.

   Даний шаблон дещо нагадує шаблон «фабричний метод», однак область його використання абсолютно інша - для покрокового визначення конкретного алгоритму; більш того, даний шаблон не обов'язково створює нові об'єкти - лише визначає послідовність дій.

   + Полегшує повторне використання коду.

   - Ви жорстко обмежені скелетом існуючого алгоритму.

   - Ви можете порушити принцип підстановки Барбари Лісков, змінюючи базову поведінку одного з кроків алгоритму через підклас.

   - З ростом кількості кроків шаблонний метод стає занадто складно підтримувати.

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 функціонал забезпечення правильного підрахунку часу використання комп’ютера при умові «простою», тобто відсутності користувача. У перспективі, даний функціонал змушує до створення нового окремого монітору із перенесенням створеної логіки.

Передаємо на інтерфейс статус активності пристрою і для цього оновимо клас основного монітору ActivityMonitor.

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 "Bridge" використовується для розділення абстракції і її реалізації таким чином, щоб вони могли змінюватися незалежно одне від одного. У коді це реалізовано через використання абстракційних класів для моніторів (як-от BaseMonitorAbstraction, SaveableMonitorAbstraction, ActivityFlagMonitorAbstraction) та їх реалізацій (SaveableMonitorImplementation, ActivityFlagMonitorImplementation).

```python
class BaseMonitorAbstraction(ABC):
    def __init__(self, implementation: SaveableMonitorImplementation | ActivityFlagMonitorImplementation):
        self.implementation = implementation

    def update_widget(self) -> None:
        self.implementation.perform_update()

class SaveableMonitorAbstraction(BaseMonitorAbstraction):
    def save_data(self) -> None:
        self.implementation.save_data()

class ActivityFlagMonitorAbstraction(BaseMonitorAbstraction):
    def get_activity_flag(self) -> bool:
        return self.implementation.get_activity_flag()
```

Рисунок 1.1 - Абстрактний базовий клас BaseMonitorAbstraction та його розширення SaveableMonitorAbstraction і ActivityFlagMonitorAbstraction

BaseMonitorAbstraction є базовим класом для абстракцій моніторів. Він отримує реалізацію монітора як параметр в конструкторі та делегує виклики її методам (наприклад, perform_update).
SaveableMonitorAbstraction і ActivityFlagMonitorAbstraction розширюють BaseMonitorAbstraction і додають конкретну функціональність, таку як збереження даних або отримання флагу активності, залежно від типу монітора.

```python
class SaveableMonitorImplementation(ABC):
    @abstractmethod
    def perform_update(self) -> None:
        pass

    @abstractmethod
    def save_data(self) -> None:
        pass

class ActivityFlagMonitorImplementation(ABC):
    @abstractmethod
    def perform_update(self) -> None:
        pass

    @abstractmethod
    def get_activity_flag(self) -> bool:
        pass
```

Рисунок 1.2 - Класи SaveableMonitorImplementation та ActivityFlagMonitorImplementation, що реалізують абстрактні класи

Bridge дозволяє мати різну конкретну реалізацію моніторів незалежно від абстракції.

```
class ProcessorMonitorImplementation(SaveableMonitorImplementation):
    def __init__(self, db_file, gui_var):
        self.processor_usage = ProcessorUsage(db_file, gui_var)

    def perform_update(self) -> None:
        self.processor_usage.update_widget()

    def save_data(self) -> None:
        self.processor_usage.save_data()

class MemoryMonitorImplementation(SaveableMonitorImplementation):
    def __init__(self, db_file, gui_var):
        self.memory_usage = MemoryUsage(db_file, gui_var)

    def perform_update(self) -> None:
        self.memory_usage.update_widget()

    def save_data(self) -> None:
        self.memory_usage.save_data()

class WindowMonitorImplementation(SaveableMonitorImplementation):
    def __init__(self, db_file, gui_var):
        self.window_monitor = WindowMonitor(db_file, gui_var)

    def perform_update(self) -> None:
        self.window_monitor.update_widget()

    def save_data(self) -> None:
        self.window_monitor.save_data()

class MouseMonitorImplementation(ActivityFlagMonitorImplementation):
    def __init__(self, gui_var):
        self.mouse_monitor = MouseMonitor(gui_var)

    def perform_update(self) -> None:
        self.mouse_monitor.update_widget()

    def get_activity_flag(self) -> bool:
        return self.mouse_monitor.activity_flag

class KeyboardMonitorImplementation(ActivityFlagMonitorImplementation):
    def __init__(self, gui_var):
        self.keyboard_monitor = KeyboardMonitor(gui_var)

    def perform_update(self) -> None:
        self.keyboard_monitor.update_widget()

    def get_activity_flag(self) -> bool:
        return self.keyboard_monitor.activity_flag
```

Рисунок 1.3 - Класи конкретних реалізацій моніторів із необхідними методами

Направимо реалізацію фабрики із попередньої частини роботи для відповідності інтеграції патерну AbstractFactory із Bridge.

```python
class ConcreteWindowsFactory(AbstractFactory):
    def create_processor_monitor(self, db_file, gui_var) -> SaveableMonitorAbstraction:
        implementation = ProcessorMonitorImplementation(db_file, gui_var)
        return SaveableMonitorAbstraction(implementation)

    def create_memory_monitor(self, db_file, gui_var) -> SaveableMonitorAbstraction:
        implementation = MemoryMonitorImplementation(db_file, gui_var)
        return SaveableMonitorAbstraction(implementation)

    def create_window_monitor(self, db_file, gui_var) -> SaveableMonitorAbstraction:
        implementation = WindowMonitorImplementation(db_file, gui_var)
        return SaveableMonitorAbstraction(implementation)

    def create_mouse_monitor(self, gui_var) -> ActivityFlagMonitorAbstraction:
        implementation = MouseMonitorImplementation(gui_var)
        return ActivityFlagMonitorAbstraction(implementation)

    def create_keyboard_monitor(self, gui_var) -> ActivityFlagMonitorAbstraction:
        implementation = KeyboardMonitorImplementation(gui_var)
        return ActivityFlagMonitorAbstraction(implementation)
```

Рисунок 1.4 - Оновлена версія класу ConcreteWindowsFactory, що підтримує патерн Міст

Таким чином, тепер абстракція та реалізація моніторів можуть змінюватися незалежно, даючи більшу гнучкість у розширенні системи (наприклад, ми можемо додавати нові типи моніторів або змінювати їх реалізацію без змін в іншій частині програми).

3. Висновок

У ході даної лабораторної роботи було реалізовано частину функціональності системи моніторингу активності, зокрема функціональність обробки стану «простою» системи без користувача. Ми ознайомились із кількома шаблонами які використовують у програмуванні, у тому числі і шаблоном "Bridge", який було використано при реалізації даної частини проекту.