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 №5

з дисципліни «Технології розроблення програмного забезпечення»

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

Перевірив:
Мягкий М. Ю

Київ 2024

Зміст

Тема: Шаблони «ADAPTER», «BUILDER», «COMMAND», «CHAIN OF RESPONSIBILITY», «PROTOTYPE»

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

«Анти-шаблони» проектування (anti-patterns), також відомі як пастки (pitfalls) - це класи найбільш часто впроваджуваних поганих рішень проблем. Вони вивчаються, як категорія, в разі коли їх хочуть уникнути в майбутньому, і деякі їхні окремі випадки можуть бути розпізнані при вивченні непрацюючих систем.

Анти-патерни в програмуванні:

• Непотрібна складність (Accidental complexity): Внесення непотрібної складності в рішення;

• Дія на відстані (Action at a distance): Несподівана взаємодія між широко розділеними частинами системи;

• Накопичити і запустити (Accumulate and fire): Установка параметрів підпрограм в наборі глобальних змінних;

• Сліпа віра (Blind faith): Недостатня перевірка (a) коректності виправлення помилки або (b) результату роботи підпрограми;

• Активне очікування (Busy spin): Споживання ресурсів ЦПУ (процесорного часу) під час очікування події, зазвичай за допомогою постійно повторюваної перевірки, замість того, щоб використовувати асинхронне програмування (наприклад, систему повідомлень або подій);

• Кешування помилки (Caching failure): Забувати скинути прапор помилки після її обробки;

• Смердючий підгузник (The Diaper Pattern Stinks): Скидання прапора помилки без її обробки або передачі вищестоящому оброблювачу;

• Перевірка типу замість інтерфейсу (Checking type instead of membership, Checking type instead of interface): Перевірка того, що об'єкт має специфічний тип в той час, коли потрібно тільки певний інтерфейс;

• Кодування шляхом виключення (Coding by exception): Додавання нового коду для підтримки кожного спеціального розпізнаного випадку;

• Таємничий код (Cryptic code): Використання абревіатур замість мнемонічних імен;

• Жорстке кодування (Hard code): Впровадження припущень про оточення системи в занадто великій кількості точок її реалізації;

• Потік лави (Lava flow): Збереження небажаного (зайвого або низькоякісного) коду через те, що його видалення занадто дороге або буде мати непередбачувані наслідки;

• Спагеті-код (Spaghetti code, іноді «макарони»): Код з надмірно заплутаним порядком виконання;

• Мильна бульбашка (Soap bubble): Клас, ініціалізований сміттям, максимально довго прикидається, що містить якісь дані;

• Мютексне пекло (Mutex hell): Впровадження занадто великої кількості об'єктів синхронізації між потоками;

А також детальніше розглянемо декілька добрих шаблонів:

1. Шаблон «Adapter».
   Шаблон "Adapter" (адаптер) використовується для адаптації інтерфейсу одного об'єкту до іншого. Наприклад, існує декілька бібліотек для роботи з принтерами, проте кожна має різний інтерфейс (хоча однакові можливості і призначення). Має сенс розробити уніфікований інтерфейс (сканування, асинхронне сканування, двостороннє сканування, потокове сканування і тому подібне), і реалізувати відповідні адаптери для приведення бібліотек до уніфікованого інтерфейсу. Це дозволить в програмі звертатися до загального інтерфейсу, а не приводити різні сценарії роботи залежно від способу реалізації бібліотеки. Адаптери також називаються "wrappers" (обгортками).
   + Відокремлює та приховує від клієнта подробиці перетворення різних інтерфейсів.
   - Ускладнює код програми внаслідок введення додаткових класів.

2. Шаблон «Builder».
   Шаблон "Builder" (будівельник) використовується для відділення процесу створення об'єкту від його представлення. Це доречно у випадках, коли об'єкт має складний процес створення (наприклад, Web- сторінка як елемент повної відповіді web- сервера) або коли об'єкт повинен мати декілька різних форм створення (наприклад, при конвертації тексту з формату у формат).
   + Дозволяє використовувати один і той самий код для створення різноманітних продуктів.
   - Клієнт буде прив’язаний до конкретних класів будівельників, тому що в інтерфейсі будівельника може не бути методу отримання результату.

3. Шаблон «Command».
   Шаблон "Command" (команда) перетворить звичайний виклик методу в клас. Таким чином дії в системі стають повноправними об'єктами. Це зручно в наступних випадках:
   - Коли потрібна розвинена система команд - команди будуть добавлятися;
   -Коли потрібна гнучка система команд - коли з'являється необхідність додавати командам можливість відміни, логування і інш.;
   - Коли потрібна можливість складання ланцюжків команд або виклику команд в певний час;
   Об'єкт команда сама по собі не виконує ніяких фактичних дій окрім перенаправлення запиту одержувачеві (команди виконуються одержувачем), однак ці об'єкти можуть зберігати дані для підтримки додаткових функцій відміни, логування... Наприклад, команда вставки символу може запам'ятовувати символ, і при виклику відміни викликати відповідну функцію витирання символу. Можна також визначити параметр "застосовності" команди (наприклад, на картинці писати не можна) - і використати цей атрибут для засвічування піктограми в меню. Такий підхід до команд дозволяє побудувати дуже гнучку систему команд, що настроюється. У більшості додатків це буде зайвим (використовується спрощений варіант), проте життєво важливий в додатках з великою кількістю команд (редактори).
   + Прибирає пряму залежність між об’єктами, що викликають операції, та об’єктами, які їх безпосередньо виконують.
   + Дозволяє реалізувати просте скасування і повтор операцій.
   + Дозволяє реалізувати відкладений запуск операцій.
   + Дозволяє збирати складні команди з простих.
   + Реалізує принцип відкритості/закритості.
   - Ускладнює код програми внаслідок введення багатьох додаткових класів.

4. Шаблон «Chain of Responsibility».
   Шаблон "Chain of responsibility" (ланцюг відповідальності) частково можна спостерігати в житті, коли підписання відповідного документу проходить від його складання у одного із співробітників компанії через менеджера і начальника до головного начальника, який ставить свій підпис.
   + Зменшує залежність між клієнтом та обробниками.
   + Реалізує принцип єдиного обов’язку.
   + Реалізує принцип відкритості/закритості.
   - Запит може залишитися ніким не опрацьованим.

5. Шаблон «Prototype».
   Шаблон "Prototype" (прототип) використовується для створення об'єктів за "шаблоном" (чи "кресленню", "ескізу") шляхом копіювання шаблонного об'єкту. Для цього визначається метод "клонувати" в об'єктах цього класу. Цей шаблон зручно використати, коли заздалегідь відомо як виглядатиме кінцевий об'єкт (мінімізується кількість змін до об'єкту шляхом створення шаблону), а також для видалення необхідності створення об'єкту - створення відбувається за рахунок клонування, і зухвалій програмі абсолютно немає необхідності знати, як створювати об'єкт. Також, це дозволяє маніпулювати об'єктами під час виконання програми шляхом настроювання відповідних шаблонів; значно зменшується ієрархія спадкоємства (оскільки в іншому випадку це були б не шаблони, а вкладені класи, що наслідують).
   + Дозволяє клонувати об’єкти без прив’язки до їхніх конкретних класів.
   + Менша кількість повторювань коду ініціалізації об’єктів.
   + Прискорює створення об’єктів.
   + Альтернатива створенню підкласів під час конструювання складних об’єктів.
   - Складно клонувати складові об’єкти, що мають посилання на інші об’єкти.

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, Report та вносимо мінімальні зміни у файли моніторів та репозиторіїв для забезпечення створення інтерфейсу та логіки створення звітів про стан системи відповідно до вимог функціоналу.

Оновимо клас основного монітору ActivityMonitor та додамо у інтерфейс кнопку «Generate report» та реалізуємо відкриття нового вікна з вибором типу репорту а також виведенням результатів та безпосередньо звіту до ще одного вікна.

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.

У системі моніторингу активності комп’ютера шаблон "Command" застосовано для організації виконання запитів на генерацію звітів у стандартизований спосіб. Це забезпечує розділення логіки запиту (команди) від її виконання (репорту), що підвищує гнучкість і розширюваність системи.

Ініціалізуємо Invoker у основному класі ActivityMonitor та створюємо команду яку виконуємо через Invoker, отримуючи результат.

```
def handle_report_submission(self, selection, day, start_date, end_date, report_type, window):
    report_generator = Report(self.db_file)
    invoker = ReportInvoker()
    try:
        if report_type == 0:
            print("Please select a report type.")
            return

        if selection == 1:
            if not day or not self.is_valid_date(day):
                print("Valid date (YYYY-MM-DD) is required.")
                return

            entered_date = datetime.datetime.strptime(day, "%Y-%m-%d")
            if entered_date > datetime.datetime.now():
                print("The entered date is in the future.")
                return

            print(f"Fetching daily statistics for {day}, Report Type: {report_type}")
            command = GenerateDailyReportCommand(report_generator, day, report_type)

        elif selection == 2:
            if not start_date or not end_date or not self.is_valid_date(start_date) or not self.is_valid_date(end_date):
                print("Valid start and end dates (YYYY-MM-DD) are required.")
                return

            start_date_obj = datetime.datetime.strptime(start_date, "%Y-%m-%d")
            end_date_obj = datetime.datetime.strptime(end_date, "%Y-%m-%d")

            if start_date_obj > end_date_obj:
                print("Start date cannot be later than end date.")
                return

            if end_date_obj > datetime.datetime.now():
                print("The entered dates include future dates.")
                return

            print(f"Fetching periodic statistics from {start_date} to {end_date}, Report Type: {report_type}")
            command = GeneratePeriodicReportCommand(report_generator, start_date, end_date, report_type)
            else:
                print("Invalid selection type.")
                return

            invoker.set_command(command)
            result = invoker.execute_command()

            if not result or isinstance(result, dict) and all(len(value) == 0 for value in result.values()):
                self.display_report("No data available for the selected report.")
            else:
                window.destroy()
                self.display_report(result)
        except Exception as e:
            print(f"Error: {e}")
```

Рисунок 1.1 - Функція у класі ActivityMonitor із ініціацією команди

Команди GenerateDailyReportCommand та GeneratePeriodicReportCommand інкапсулюють параметри запитів для генерації звітів за день або період і наслідують абстрактний клас Command.

```python
from abc import ABC, abstractmethod

class Command(ABC):
    @abstractmethod
    def execute(self):
        pass


class GenerateDailyReportCommand(Command):
    def __init__(self, report, date, report_type):
        self.report = report
        self.date = date
        self.report_type = report_type

    def execute(self):
        return self.report.generate_daily_report(self.date, self.report_type)

class GeneratePeriodicReportCommand(Command):
    def __init__(self, report, start_date, end_date, report_type):
        self.report = report
        self.start_date = start_date
        self.end_date = end_date
        self.report_type = report_type

    def execute(self):
        return self.report.generate_periodic_report(self.start_date, self.end_date, self.report_type)
```

Рисунок 1.2 - Класи GenerateDailyReportCommand, GeneratePeriodicReportCommand та абстрактний клас Command

Invoker (ReportInvoker) викликає задану команду, дозволяючи динамічно змінювати, яка саме операція виконується.

```
class ReportInvoker:
    def __init__(self):
        self.command = None

    def set_command(self, command):
        self.command = command

    def execute_command(self):
        if self.command:
            return self.command.execute()
        else:
            print("No command set.")
            return None
```

Рисунок 1.3 - Клас ReportInvoker

Клас Report безпосередньо відповідає за логіку створення звітів та комунікує з репозиторіями для отримання історичних даних

```python
class Report:
    def __init__(self, db_file):...

    def generate_daily_report(self, date, report_type):
        try:
            def is_current_date(date):
                today = datetime.datetime.now().strftime("%Y-%m-%d")
                return date == today

            data = {}

            if report_type == 1: # cpu
                data['processor_usage'] = self.processorrepo.get_processor_usage_by_date(date)
            elif report_type == 2: # browser usage %
                if is_current_date(date):
                    window_usage = self.windowmonitor.get_window_usage()
                else:
                    window_usage = self.windowrepo.get_window_usage_by_date(date)
                browser_names = ["Chrome", "Firefox", "Safari", "Edge", "Opera"]
                filtered_data = {name: usage for name, usage in window_usage.items() if any(browser in name for browser in browser_names)}
                data['browser_usage'] = filtered_data
            elif report_type == 3: # memory
                data['memory_usage'] = self.memoryrepo.get_memory_usage_by_date(date)
            elif report_type == 5: # programs used
                if is_current_date(date):
                    data['window_usage'] = self.windowmonitor.get_window_usage()
                else:
                    data['window_usage'] = self.windowrepo.get_window_usage_by_date(date)
            return data

        except Exception as e:
            print(f"Error generating daily report: {e}")
            return None

    def generate_periodic_report(self, start_date, end_date, report_type):
        try:
            data = []

            start = datetime.datetime.strptime(start_date, "%Y-%m-%d")
            end = datetime.datetime.strptime(end_date, "%Y-%m-%d")
            delta = datetime.timedelta(days=1)

            current_date = start
            while current_date <= end:
                date_str = current_date.strftime("%Y-%m-%d")
                daily_data = self.generate_daily_report(date_str, report_type)
                if daily_data:
                    data.append({
                        "date": date_str,
                        "data": daily_data
                    })
                current_date += delta

        except Exception as e:
            print(f"Error generating periodic report: {e}")
            return None
```

Рисунок 1.4 - Клас Report

Таким чином, використовуючи даний патерн легко додати нові типи звітів або операцій (наприклад, експорт у файл) шляхом створення нових класів команд а також підтримувати та розділяти логіку системи.

3. Висновок

У ході даної лабораторної роботи було реалізовано частину функціональності системи моніторингу активності, зокрема класи пов’язані із функціональністю створення звітів на базі зібраних даних. Ми ознайомились із кількома шаблонами які використовують у програмуванні, у тому числі і шаблон "Command", який було використано при реалізації даної частини проекту.