



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 №8

«Патерни проектування»

Виконав
студент групи ІА-34:
Сльота Максим

Перевірив:
Мягкий М. Ю.

Тема: Патерни проектування.

Мета роботи: Вивчити структуру шаблонів «Composite», «Flyweight» (Пристосуванець), «Interpreter», «Visitor» та навчитися застосовувати їх в реалізації програмної системи.

Вихідні дані:

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.

Теоретичні відомості:

Шаблон «Composite» - шаблон використовується для складання об'єктів в деревоподібну структуру для подання ієрархій типу «частина цілого». Даний шаблон дозволяє уніфіковано обробляти як поодинокі об'єкти, так і об'єкти з вкладеністю [6].

Шаблон «Flyweight» - Шаблон використовується для зменшення кількості об'єктів в додатку шляхом поділу цих об'єктів між ділянками додатку. Flyweight являє собою поділюваний об'єкт.

Шаблон «Interpreter» - даний шаблон використовується для подання граматики і інтерпретатора для вибраної мови (наприклад, скриптової) [6]. Граматика мови представлена термінальними і нетермінальними символами, кожен з яких інтерпретується в контексті використання.

Шаблон «Visitor» - Шаблон відвідувач дозволяє вказувати операції над елементами без зміни структури конкретних елементів [6]. Таким чином вкрай зручно додавати нові операції, проте дуже важко додавати нові елементи в ієрархію (необхідно додавати відповідні методи для обробки їх відвідувань в кожного відвідувача).

Хід роботи:

1. Діаграма класів, яка представляє використання шаблону Visitor

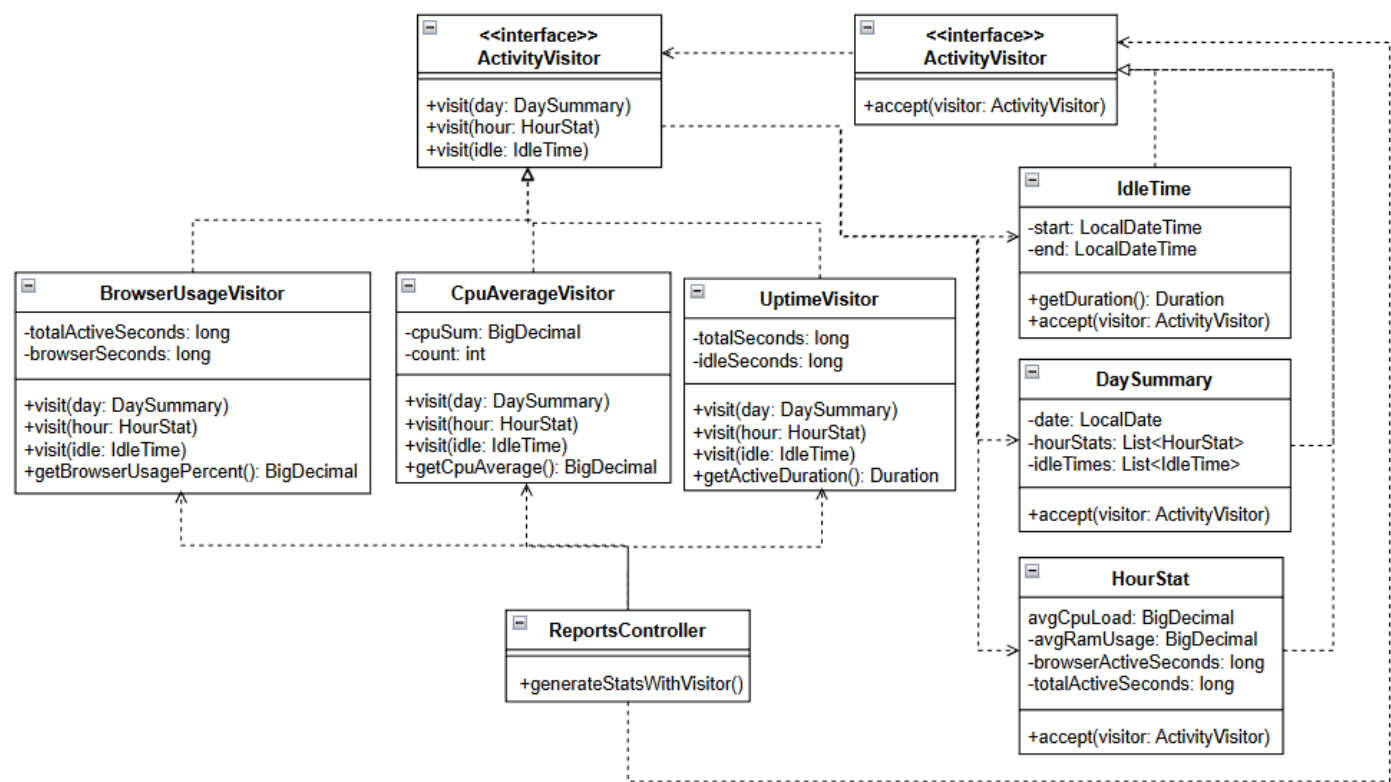


Рис 1. Діаграма класів

На діаграмі зображена реалізація класичного шаблону проєктування Visitor, який застосовується у проєкті System Activity Monitor для обробки статистичних даних та формування різних видів звітів.

У центрі структури знаходяться два основні інтерфейси — ActivityElement та ActivityVisitor.

Інтерфейс ActivityElement визначає метод accept(visitor), який має реалізувати кожен елемент статистики. Інтерфейс ActivityVisitor містить три методи visit(), призначені для обробки різних типів елементів: DaySummary, HourStat та IdleTime.

DaySummary описує статистику за один день і містить списки об'єктів HourStat та IdleTime. Клас HourStat зберігає середні показники навантаження на CPU, RAM, а також дані про активність у браузері та загальний активний час. Клас IdleTime містить інформацію про періоди простою та надає метод для обчислення тривалості цього простою.

Усі три моделі реалізують інтерфейс ActivityElement, тому кожна з них має власний метод accept(visitor).

Завдяки цьому моделі не містять логіки обробки статистики — вони лише передають контроль відвідувачу.

`BrowserUsageVisitor` — обчислює відсоток часу, проведений у веб-браузері;

`CpuAverageVisitor` — визначає середнє навантаження на процесор за вибраний період;

`UptimeVisitor` — аналізує загальний час роботи системи та час простою.

Кожен відвідувач реалізує інтерфейс `ActivityVisitor` та перевизначає методи `visit()` для кожного типу статистичного елемента. Таким чином, логіка обчислень зосереджена окремо в кожному відвідувачі, а не в моделях даних, що відповідає принципам SOLID. У нижній частині діаграми розташований `ReportsController`, який виступає клієнтом шаблону `Visitor`. Контролер створює об'єкти відвідувачів та передає їх у метод `accept()` кожного елемента статистики (`DaySummary`, `HourStat`, `IdleTime`). Таким чином, контролер отримує готові значення для формування звітів у інтерфейсі `JavaFX`. Уся структура діаграми демонструє чіткий поділ відповідальностей: моделі зберігають дані, відвідувачі реалізують обчислення, а контролер керує процесом збору та візуалізації результатів. Такий підхід спрощує розширення функціоналу — для додавання нового типу звіту достатньо створити новий клас-відвідувач без змін у моделях або контролерах.

Доданий вихідний код системи:

`DaySummary.java`

```
public class DaySummary implements ActivityElement {
    private LocalDate date;
    private List<HourStat> hourStats;
    private List<IdleTime> idleTimes;
    @Override
    public void accept(ActivityVisitor visitor) {
        visitor.visit(this);
        if (hourStats != null) {
            for (HourStat h : hourStats) {
                h.accept(visitor);
            }
        }
        if (idleTimes != null) {
```

```

        for (IdleTime idle : idleTimes) {
            idle.accept(visitor);
        }
    }
}

```

HourStat.java

```

public class HourStat implements ActivityElement {

```

```

    private LocalDateTime hourStart;
    private BigDecimal avgCpuLoad;
    private BigDecimal avgRamUsage;
    private long browserActiveSeconds;
    private long totalActiveSeconds;

```

```

    @Override

```

```

    public void accept(ActivityVisitor visitor) {
        visitor.visit(this);
    }
}

```

IdleTime.java

```

public class IdleTime implements ActivityElement {

```

```

    private LocalDateTime start;
    private LocalDateTime end;

```

```

    public Duration getDuration() {
        if (start != null && end != null) {
            return Duration.between(start, end);
        }
        return Duration.ZERO;
    }
}

```

```

@Override
public void accept(ActivityVisitor visitor) {
    visitor.visit(this);
}
}

BrowserUsageVisitor.java

public class BrowserUsageVisitor implements ActivityVisitor {

    private long totalActiveSeconds = 0;
    private long browserSeconds = 0;

    @Override
    public void visit(DaySummary daySummary) {
    }

    @Override
    public void visit(HourStat hourStat) {
        totalActiveSeconds += hourStat.getTotalActiveSeconds();
        browserSeconds += hourStat.getBrowserActiveSeconds();
    }

    @Override
    public void visit(IdleTime idleTime) {
    }

    public BigDecimal getBrowserUsagePercent() {
        if (totalActiveSeconds == 0) {
            return BigDecimal.ZERO;
        }
        BigDecimal total = BigDecimal.valueOf(totalActiveSeconds);
        BigDecimal browser = BigDecimal.valueOf(browserSeconds);
        return browser
            .multiply(BigDecimal.valueOf(100))
            .divide(total, 2, RoundingMode.HALF_UP);
    }
}

```

CpuAverageVisitor.java

```
public class CpuAverageVisitor implements ActivityVisitor {
```

```
    private BigDecimal cpuSum = BigDecimal.ZERO;
```

```
    private int count = 0;
```

```
    @Override
```

```
    public void visit(DaySummary daySummary) {  
    }
```

```
    @Override
```

```
    public void visit(HourStat hourStat) {  
        if (hourStat.getAvgCpuLoad() != null) {  
            cpuSum = cpuSum.add(hourStat.getAvgCpuLoad());  
            count++;  
        }  
    }
```

```
    @Override
```

```
    public void visit(IdleTime idleTime) {  
    }
```

```
    public BigDecimal getCpuAverage() {
```

```
        if (count == 0) {
```

```
            return BigDecimal.ZERO;
```

```
        }
```

```
        return cpuSum.divide(BigDecimal.valueOf(count), 2, RoundingMode.HALF_UP);
```

```
    }
```

```
}
```

UptimeVisitor.java

```
public class UptimeVisitor implements ActivityVisitor {
```

```
private long totalSeconds = 0;
```

```
private long idleSeconds = 0;
```

```
@Override
```

```
public void visit(DaySummary daySummary) {  
}
```

```
@Override
```

```
public void visit(HourStat hourStat) {  
    totalSeconds += hourStat.getTotalActiveSeconds();  
}
```

```
@Override
```

```
public void visit(IdleTime idleTime) {  
    idleSeconds += idleTime.getDuration().getSeconds();  
}
```

```
public Duration getTotalDuration() {  
    return Duration.ofSeconds(totalSeconds + idleSeconds);  
}
```

```
public Duration getIdleDuration() {  
    return Duration.ofSeconds(idleSeconds);  
}
```

```
public Duration getActiveDuration() {  
    return Duration.ofSeconds(totalSeconds);  
}
```

```
}
```

ReportsController.java

```
public class ReportsController {
```

```
@FXML private DatePicker startDatePicker;
```

```
@FXML private DatePicker endDatePicker;
```

```
@FXML private Label avgCpuLabel;
```

```
@FXML private Label uptimeAvgLabel;
```

```
@FXML private Label browserUsageLabel;
```

```
private final ReportService reportService = new ReportService();
```

```
@FXML
```

```
private void generateStatsWithVisitor() {
```

```
    LocalDate from = startDatePicker.getValue();
```

```
    LocalDate to = endDatePicker.getValue();
```

```
    if (from == null || to == null) {
```

```
        // покажи повідомлення користувачу
```

```
        return;
```

```
    }
```

```
    List<DaySummary> summaries =
```

```
        reportService.getDaySummariesForPeriod(from, to);
```

```
    BrowserUsageVisitor browserVisitor = new BrowserUsageVisitor();
```

```
    CpuAverageVisitor cpuVisitor = new CpuAverageVisitor();
```

```
    UptimeVisitor uptimeVisitor = new UptimeVisitor();
```

```
    for (DaySummary day : summaries) {
```

```
        day.accept(browserVisitor);
```

```
        day.accept(cpuVisitor);
```

```
        day.accept(uptimeVisitor);
```

```
    }
```

```

browserUsageLabel.setText(
    browserVisitor.getBrowserUsagePercent().toString() + " %"
);

avgCpuLabel.setText(
    cpuVisitor.getCpuAverage().toString() + " %"
);

long activeHours =
    uptimeVisitor.getActiveDuration().toHours();
uptimeAvgLabel.setText(activeHours + " год активної роботи");
}
}

```

Висновок: У ході лабораторної роботи було розроблено та реалізовано шаблон проектування Visitor для обробки даних системного моніторингу. Такий підхід дозволив чітко розділити структуру моделей, які зберігають статистичну інформацію, та логіку обчислень, що виконується у відвідувачах. Моделі DaySummary, HourStat та IdleTime залишилися простими контейнерами даних, тоді як конкретні відвідувачі (BrowserUsageVisitor, CpuAverageVisitor, UptimeVisitor) отримали відповідальність за формування різних видів статистики. Завдяки використанню методу accept() та поліморфізму шаблон Visitor забезпечив можливість гнучко додавати нові алгоритми обробки без зміни існуючих класів. Це підвищує розширюваність, спрощує підтримку, робить архітектуру більш чистою та відповідною принципам SOLID. Також було продемонстровано практичне застосування Visitor у клієнтському коді — JavaFX-контролері, який викликає відвідувачів для формування звітів.

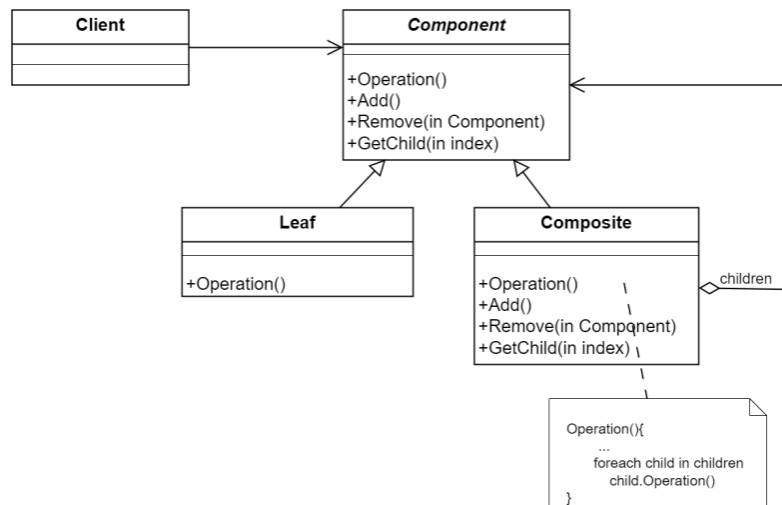
Відповідь на контрольні питання:

1. Яке призначення шаблону «Композит»?

Композит потрібен тоді, коли у тебе є об'єкти, які можуть бути як окремими елементами, так і контейнерами для інших елементів. Головна ідея — працювати з одиничними об'єктами і групами однаково. Наприклад, файл — це елемент, папка — контейнер. Але і файл, і папка поведуться подібно: їх можна відкрити, перейменувати, видалити. Композит якраз дозволяє будувати дерево об'єктів і обробляти їх єдиним

способом.

2. Нарисуйте структуру шаблону «Композит».



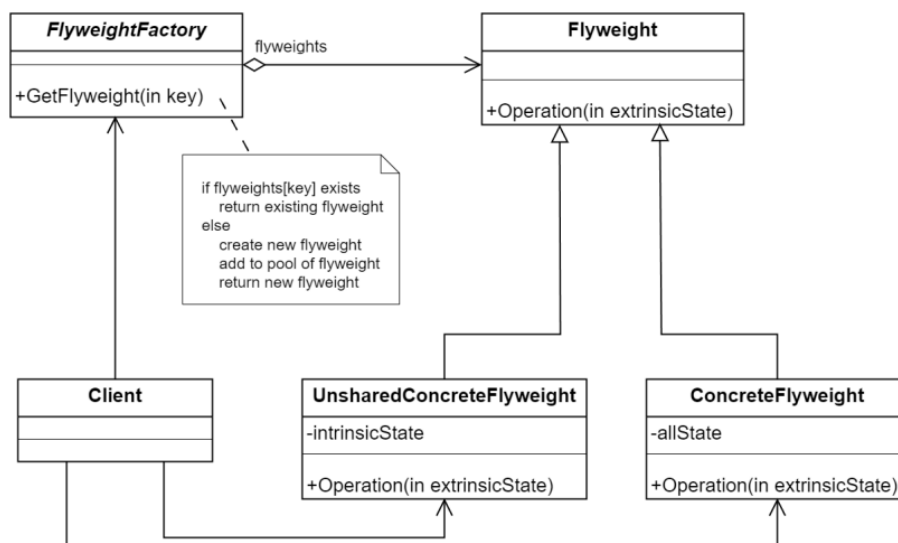
3. Які класи входять в шаблон «Композит», та яка між ними взаємодія?

Є абстрактний клас **Component** — це спільна основа для всіх. Далі є **Leaf** — це звичайний елемент без дітей. І є **Composite** — це складний елемент, який всередині містить дітей і може додавати або видаляти їх. Коли викликається операція на **Composite**, він передає виконання всім своїм дітям. Таким чином утворюється дерево, яке можна обходити однаковим методом.

4. Яке призначення шаблону «Легковаговик»?

Легковаговик створений для економії пам'яті. Він дозволяє не створювати однакові об'єкти багато разів, а ділити їх між собою. Це корисно, коли в програмі дуже багато однотипних елементів, наприклад символи тексту в редакторі, де форма літери однакова для всіх, а відрізняється тільки позиція.

5. Нарисуйте структуру шаблону «Легковаговик».



6. Які класи входять в шаблон «Легковаговик», та яка між ними взаємодія?

Є Flyweight — це об'єкт, який містить внутрішній стан, спільний для всіх клієнтів. Є FlyweightFactory — вона зберігає вже створені легковаговики і повертає їх при запиті. Є Client — він передає зовнішній стан, який не зберігається всередині легковаговика. Загалом, фабрика слідкує, щоб ти не створював дублікати, а клієнт працює з одним об'єктом, наче це унікальний екземпляр.

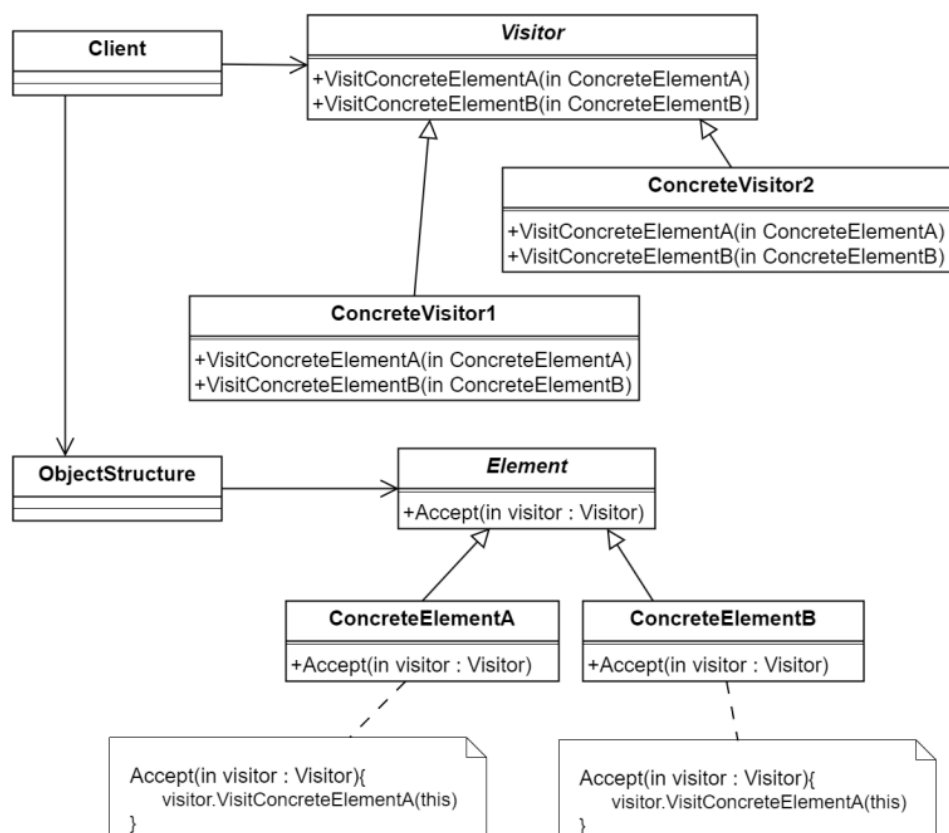
7. Яке призначення шаблону «Інтерпретатор»?

Інтерпретатор потрібен для розбору та виконання простих мов або правил. Наприклад, якщо є міні-мова для пошуку файлів типу “файл і (папка або дата)”, то Інтерпретатор будує дерево правил і обчислює його на основі контексту. Це спосіб перетворити текстові команди на структуру, яку можна виконати.

8. Яке призначення шаблону «Відвідувач»?

Відвідувач дозволяє додавати нові операції до об'єктів складної структури, не змінюючи самі ці об'єкти. Це зручно, коли структура стабільна, але треба час від часу додавати нові обчислення. Наприклад, в дереві елементів можна зробити відвідувача, який рахує суму, другий — який створює файл, третій — який генерує звіт. Об'єкти приймають відвідувача і дозволяють йому виконувати свою логіку.

9. Нарисуйте структуру шаблону «Відвідувач».



10. Які класи входять в шаблон «Відвідувач», та яка між ними взаємодія?

Є Visitor — інтерфейс або абстрактний клас, де описані методи візиту для різних типів

елементів. Є `ConcreteVisitor` — конкретна операція, яку ми хочемо виконати. Є `Element` — базовий клас для елементів структури. Кожен елемент має метод `accept`, який отримує відвідувача і викликає правильний метод візиту. Клієнт просто створює структуру елементів та передає туди відвідувача. Завдяки цьому в `Visitor` можна додавати нові обчислення, не змінюючи елементи.