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 №4

«Вступ до паттернів проектування.»

Виконав
студент групи ІА-34:
Сльота Максим

Перевірив:
Мягкий М. Ю.

Київ - 2025

Тема: Вступ до паттернів проектування.

Мета роботи: Вивчити структуру шаблонів «Singleton», «Iterator», «Proxy», «State», «Strategy» та навчитися застосовувати їх в реалізації програмної системи.

Вихідні дані: System activity monitor

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:

Патерн проєктування - використовуваний при розробці інформаційних систем, являє собою формалізований опис, який часто зустрічається в завданнях проєктування, вдале рішення даної задачі, а також рекомендації по застосуванню цього рішення в різних ситуаціях.

Шаблон «Singleton» - являє собою клас в термінах ООП, який може мати не більше одного об'єкта (звідси і назва «одинак»). 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.

Шаблон «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.

Шаблон «Proxy» - це 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.

Шаблон «State» - дозволяє змінювати логіку роботи об'єктів у випадку зміни їх внутрішнього стану.

Шаблон «Strategy» - дозволяє змінювати деякий алгоритм поведінки об'єкта іншим алгоритмом, що досягає ту ж мету іншим способом.

Хід роботи:

1. Діаграма класів, яка представляє використання шаблону Iterator

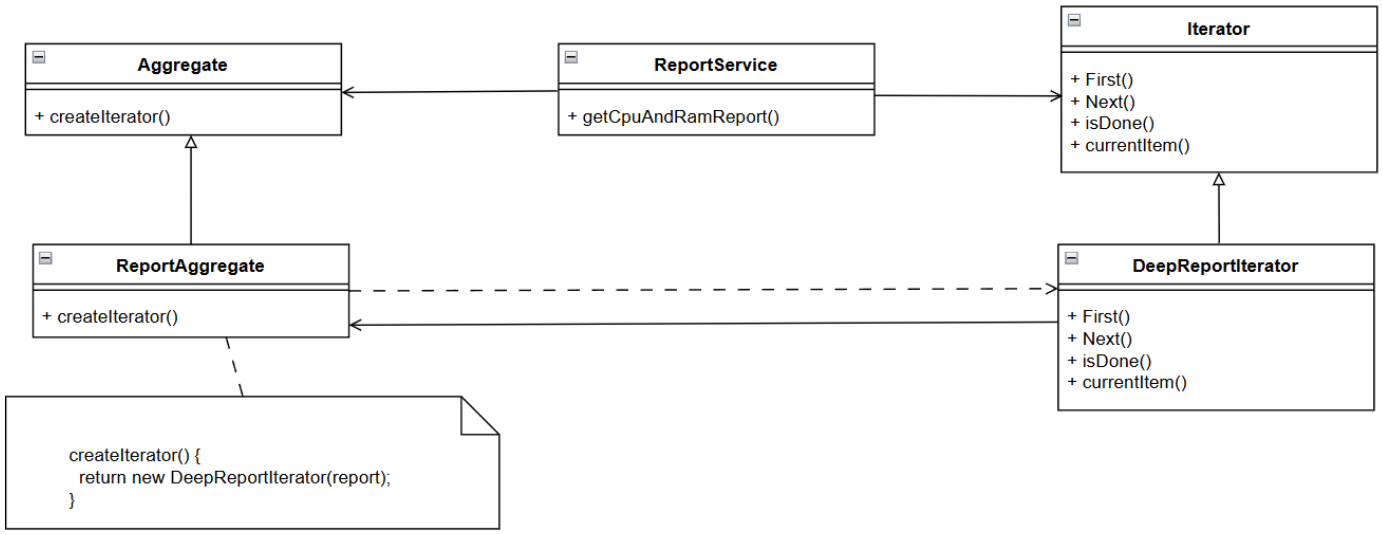

Рис 1. Діаграма класів

На діаграмі показано реалізацію патерну «Ітератор». Цей шаблон використовується для послідовного обходу вкладеної структури звіту. Кожен звіт містить кілька днів, а кожен день - список погодинних записів із середніми значеннями завантаження процесора та використання пам’яті. Ітератор дозволяє зручно отримувати ці дані по годинах без необхідності знати, як саме вони зберігаються всередині об’єкта звіту. Інтерфейс Aggregate визначає загальний контракт для створення ітератора через метод createIterator(). Клас ReportAggregate є конкретною реалізацією агрегату, який зберігає посилання на звіт і створює ітератор для його внутрішньої структури. Клас DeepReportIterator реалізує поведінку конкретного ітератора. Він відповідає за послідовний обхід усіх погодинних записів у межах звіту. Спочатку ітератор переходить між днями, а потім по годинах кожного дня. Клас ReportService виступає клієнтом патерну. Метод getCpuAndRamReport(user, report) створює об’єкт ReportAggregate, отримує ітератор і за його допомогою проходить по всіх елементах типу HourStat, формуючи звіт із середніми показниками CPU та RAM за кожну годину.

2. Доданий вихідний код системи

Aggregate.java:

```
package com.example.systemactivitymonitor.iterator;

public interface Aggregate<T> {
    Iterator<T> createIterator();
    int size();
```

```
    T get(int index);
}
```

DeepReportIterator.java:

```
package com.example.systemactivitymonitor.iterator;

import com.example.systemactivitymonitor.model.*;

import java.util.List;

public class DeepReportIterator implements Iterator<HourStat> {
    private final Report report;
    private List<DaySummary> days;
    private int dayIndex = 0;
    private int hourIndex = 0;

    public DeepReportIterator(Report report) {
        this.report = report;
        this.days = report != null ? report.getDays() : null;
    }

    @Override
    public void first() {
        dayIndex = 0;
        hourIndex = 0;
        advanceToNextValid();
    }

    @Override
    public void next() {
        if (isDone()) return;
        hourIndex++;
        advanceToNextValid();
    }

    @Override
    public boolean isDone() {
        return days == null || dayIndex >= days.size();
    }

    @Override
    public HourStat currentItem() {
```

```
        if (isDone()) return null;
        DaySummary day = days.get(dayIndex);
        if (day == null || day.getHourlyStats() == null) return null;
        if (hourIndex < 0 || hourIndex >= day.getHourlyStats().size()) return null;
        return day.getHourlyStats().get(hourIndex);
    }

    private void advanceToNextValid() {
        if (days == null) return;
        while (dayIndex < days.size()) {
            DaySummary day = days.get(dayIndex);
            if (day == null || day.getHourlyStats() == null || day.getHourlyStats().isEmpty()) {
                dayIndex++;
                hourIndex = 0;
                continue;
            }
            if (hourIndex >= day.getHourlyStats().size()) {
                dayIndex++;
                hourIndex = 0;
                continue;
            }
            break;
        }
    }
}
```

Iterator.java:

```
package com.example.systemactivitymonitor.iterator;
public interface Iterator<T> {
    void first();
    void next();
    boolean isDone();
    T currentItem();
}
```

ReportAggregate.java:

```
package com.example.systemactivitymonitor.iterator;

import com.example.systemactivitymonitor.model.HourStat;
import com.example.systemactivitymonitor.model.Report;

public class ReportAggregate implements Aggregate<HourStat> {
```

```
    private final Report report;

    public ReportAggregate(Report report) {
        this.report = report;
    }

    @Override
    public Iterator<HourStat> createIterator() {
        return new DeepReportIterator(report);
    }

    @Override
    public int size() {
        Iterator<HourStat> it = createIterator();
        int count = 0;
        it.first();
        while (!it.isDone()) {
            if (it.currentItem() != null) count++;
            it.next();
        }
        return count;
    }

    @Override
    public HourStat get(int index) {
        if (index < 0) return null;
        Iterator<HourStat> it = createIterator();
        int i = 0;
        it.first();
        while (!it.isDone()) {
            HourStat cur = it.currentItem();
            if (cur != null) {
                if (i == index) return cur;
                i++;
            }
            it.next();
        }
        return null;
    }
}
```

DaySummary.java:

```
package com.example.systemactivitymonitor.model;
import java.time.LocalDate;
import java.util.List;
public class DaySummary {
    private LocalDate date;
    private List<HourStat> hourlyStats;
    public DaySummary(LocalDate date, List<HourStat> hourlyStats) {
        this.date = date;
        this.hourlyStats = hourlyStats;
    }
    public LocalDate getDate() { return date; }
    public List<HourStat> getHourlyStats() { return hourlyStats; }
}
```

HourStat.java:

```
package com.example.systemactivitymonitor.model;
import java.math.BigDecimal;
public class HourStat {
    private int hour;
    private BigDecimal avgCpu;
    private BigDecimal avgRam;
    public HourStat(int hour, BigDecimal avgCpu, BigDecimal avgRam) {
        this.hour = hour;
        this.avgCpu = avgCpu;
        this.avgRam = avgRam;
    }
    public int getHour() { return hour; }
    public BigDecimal getAvgCpu() { return avgCpu; }
    public BigDecimal getAvgRam() { return avgRam; }
}
```

ReportService.java:

```
public String getCpuAndRamReport(User user, Report report) {
    if (report == null) return "Немає звіту.";
    if (user == null || user.getId() == null) return "Користувач не визначений.";
    if (report.getDays() == null || report.getDays().isEmpty()) {
        LocalDate start = report.getPeriodStart();
        LocalDate end   = report.getPeriodEnd();
        if (start == null || end == null) return "Немає періоду у звіті.";
        var stats = statsRepository.findByUserIdAndRecordedAtBetween(
                user.getId(),
                start.atStartOfDay(),
                end.atTime(23, 59, 59)
```

```
        );
        report.setDays(buildDaySummary(stats));
    }
    if (report.getDays() == null || report.getDays().isEmpty())
        return "Немає даних для звіту.";
    ReportAggregate aggregate = new ReportAggregate(report);
    Iterator<HourStat> it = aggregate.createIterator();
    StringBuilder sb = new StringBuilder("CPU/RAM по годинах:\n");
    for (it.first(); !it.isDone(); it.next()) {
        HourStat stat = it.currentItem();
        sb.append(String.format("Година %02d → CPU: %.2f%% | RAM: %.2fMB%n",
                stat.getHour(), stat.getAvgCpu(), stat.getAvgRam()));
    }
    return sb.toString();
}
```

Висновок: У результаті реалізації шаблону «Ітератор» у проєкті System Activity Monitor вдалося досягти чіткого розділення обов'язків між класами та підвищити гнучкість системи. Завдяки використанню ітератора клас ReportService отримує доступ до вкладених даних звіту - днів і погодинної статистики - без необхідності знати їхню внутрішню структуру.

Відповідь на контрольні питання:

1. Що таке шаблон проєктування

Шаблон проєктування — це перевірене рішення типової задачі, яка часто зустрічається під час розробки програмного забезпечення. Це не готовий код, а загальна структура або принцип, який допомагає правильно організувати взаємодію між класами та об'єктами.

2. Навіщо використовувати шаблони проєктування

Використання шаблонів дозволяє створювати більш гнучке, зрозуміле і підтримуване програмне забезпечення. Вони спрощують розширення коду, усувають дублювання, сприяють повторному використанню готових рішень і покращують взаєморозуміння між розробниками, бо базуються на загальновідомих архітектурних підходах.

3. Яке призначення шаблону «Стратегія»

Шаблон «Стратегія» дозволяє змінювати алгоритм роботи програми під час виконання без зміни структури класів. Він інкапсулює різні варіанти поведінки (алгоритми) в окремих класах, і контекст може динамічно вибирати потрібну стратегію.

4. Структура шаблону «Стратегія»

Структура складається з інтерфейсу Strategy, який визначає спільний метод для всіх стратегій; класів ConcreteStrategy, які реалізують цей інтерфейс різними способами; та класу Context, який містить посилання на об’єкт стратегії і викликає її метод, не знаючи деталей реалізації.

5. Класи шаблону «Стратегія» і взаємодія між ними

До шаблону входять три класи:

- Strategy - спільний інтерфейс для всіх алгоритмів.
- ConcreteStrategy - реалізує конкретний варіант алгоритму.
- Context - вибирає і використовує стратегію.
  Контекст має посилання на стратегію і викликає її методи, не знаючи, яка саме реалізація використовується.

6. Призначення шаблону «Стан»

Шаблон «Стан» дозволяє об’єкту змінювати свою поведінку залежно від внутрішнього стану. Кожен стан оформлюється як окремий клас, що описує власну поведінку, і контекст динамічно змінює стан без складних умовних конструкцій.

7. Структура шаблону «Стан»

Шаблон включає інтерфейс State, який визначає спільні методи для станів; класи ConcreteState, які реалізують ці методи для конкретних станів; і клас Context, який зберігає поточний стан і делегує йому виконання операцій.

8. Класи шаблону «Стан» і взаємодія між ними

До шаблону входять три класи:

- State — інтерфейс для всіх станів.
- ConcreteState — класи, які реалізують різну поведінку.
- Context — об’єкт, що містить поточний стан і викликає його методи.
  Контекст передає управління об’єктові стану, а стан може змінити контекст, щоб перейти до іншого стану.

9. Призначення шаблону «Ітератор»

Шаблон «Ітератор» використовується для послідовного доступу до елементів колекції без розкриття її внутрішньої структури. Він дозволяє обійти всі елементи колекції незалежно від способу їх зберігання.

10. Структура шаблону «Ітератор»

Шаблон складається з інтерфейсу Iterator, який визначає методи для обходу (first,

next, isDone, currentItem); інтерфейсу Aggregate, який створює ітератор; конкретного агрегату ConcreteAggregate, який реалізує метод створення; і класу ConcreteIterator, який реалізує логіку обходу колекції.

11. Класи шаблону «Ітератор» і взаємодія між ними

У шаблоні беруть участь чотири класи:

- Aggregate — інтерфейс для створення ітератора.
- ConcreteAggregate — створює конкретний ітератор.
- Iterator — описує інтерфейс обходу.
- ConcreteIterator — реалізує методи для переміщення по елементах.
  Клієнт взаємодіє лише через інтерфейси, не знаючи структури колекції, що забезпечує інкапсуляцію.

12. Ідея шаблону «Одинак»

Шаблон «Одинак» (Singleton) гарантує, що в системі існує лише один екземпляр певного класу, і надає глобальну точку доступу до нього. Його часто застосовують для логування, конфігурацій, доступу до бази даних або роботи з ресурсами.

13. Чому шаблон «Одинак» вважають «антишаблоном»

Його називають антишаблоном, бо він створює глобальний стан, що ускладнює тестування і розширення системи. Такий підхід порушує принципи інкапсуляції, робить залежності неявними і може призвести до прихованих помилок.

14. Призначення шаблону «Проксі»

Шаблон «Проксі» використовується для контролю доступу до об’єкта. Він створює замісник, який перехоплює виклики клієнта і може додавати додаткову поведінку — наприклад, кешування, перевірку доступу або ліниве створення об’єкта.

15. Структура шаблону «Проксі»

Структура включає інтерфейс Subject, який визначає методи реального об’єкта, клас RealSubject, що виконує основну роботу, і клас Proxy, який реалізує той самий інтерфейс і делегує виклики RealSubject, додаючи додаткову логіку.

16. Класи шаблону «Проксі» і взаємодія між ними

До шаблону входять три класи:

- Subject — інтерфейс спільний для проксі та реального об’єкта.
- Proxy — замісник, який контролює доступ до реального об’єкта.
- RealSubject — об’єкт, що виконує справжню роботу.
  Клієнт працює з проксі так само, як із реальним об’єктом, не помічаючи підміни.