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 №6

із дисципліни «**Технології розроблення програмного забезпечення**»

Тема **«Шаблони «Abstract Factory», «Factory Method», «Memento», «Observer», «Decorator»»**

Виконав:

студент групи ІА–24

Яблонський Д.Б.

Перевірив:

Мягкий М.Ю.

Київ 2024

Зміст

Мета: метою виконання лабораторної роботи є вивчення та практичне застосування шаблонів проєктування, таких як «Abstract Factory», «Factory Method», «Memento», «Observer» і «Decorator», для створення ефективних і гнучких програмних рішень. Ця лабораторна робота спрямована на розвиток навичок у використанні різних патернів для вирішення задач у розробці програмного забезпечення.

Теоретичні відомості

Шаблони з даної лабораторної роботи допомагають вирішувати типові задачі у розробці програмного забезпечення, забезпечуючи структурованість, масштабованість та легкість підтримки. Основою їх застосування є принципи SOLID:

- **Single Responsibility Principle**: клас має виконувати лише одну логічну задачу, спрощуючи зміну або розширення його функціональності.
- **Open-Closed Principle**: класи повинні бути відкритими для розширення, але закритими для змін, тобто нову поведінку додають через спадкування чи композицію.
- **Liskov Substitution Principle**: підкласи мають коректно працювати у будь-якому контексті, де використовується базовий клас.
- **Interface Segregation Principle**: інтерфейси слід розбивати на дрібні, вузько спрямовані частини, щоб уникати надлишкової реалізації.
- **Dependency Inversion Principle**: модулі вищого рівня не повинні залежати від модулів нижчого рівня, обидва мають залежати від абстракцій.

Розуміння цих принципів є основою правильного застосування шаблонів і створення якісного програмного забезпечення.

Abstract Factory

Шаблон Abstract Factory дозволяє створювати цілі сімейства взаємопов'язаних об'єктів без вказання їхніх конкретних класів. Його основною ідеєю є надання інтерфейсу, який приховує процес створення об'єктів і робить систему гнучкішою до змін. Наприклад, цей шаблон можна використовувати для розробки графічного інтерфейсу, який має підтримувати

різні операційні системи. У цьому випадку фабрика створює набори компонентів, таких як кнопки чи текстові поля, які виглядають і поводяться відповідно до конкретної платформи. Ця гнучкість дозволяє легко замінювати реалізацію без змін у бізнес-логіці програми.

Factory Method

Factory Method забезпечує можливість створювати об'єкти через виклики фабричних методів, які визначаються у підкласах. Це дозволяє класам делегувати створення об'єктів, не прив'язуючись до конкретних реалізацій. Наприклад, у додатку для роботи з повідомленнями можуть бути різні типи повідомлень, такі як SMS, електронна пошта або push-сповіщення. Factory Method дозволяє створювати ці повідомлення залежно від контексту, не змінюючи клієнтський код. Це значно полегшує додавання нових типів повідомлень і робить код більш масштабованим.

Memento

Шаблон Memento використовується для збереження стану об'єкта, щоб його можна було відновити без порушення інкапсуляції. Його основна ідея полягає в тому, щоб створити "знімок" поточного стану об'єкта, який може бути збережений і використаний пізніше для відновлення. Наприклад, у текстових редакторах реалізується функціональність скасування і повтору дій за допомогою цього шаблону. Стан документа зберігається перед виконанням змін, і якщо користувач вирішить скасувати дію, система повертає попередній стан. Цей підхід дозволяє легко управляти змінами без змішування логіки управління станом із бізнес-логікою.

Observer

Observer, або "Спостерігач", дозволяє об'єкту-спостерігачу автоматично отримувати повідомлення про зміну стану іншого об'єкта. Це забезпечує динамічну залежність між об'єктами, дозволяючи їм взаємодіяти без жорсткого зв'язування. Наприклад, цей шаблон використовується у додатках новин, де підписники автоматично отримують оновлення, коли з’являється новина. Суб'єкт, у даному випадку сервер новин, повідомляє всіх підписників про зміни. Такий підхід значно спрощує процес синхронізації та забезпечує

легке додавання або видалення спостерігачів.

Decorator

Decorator дозволяє динамічно додавати нову поведінку об'єктам, зберігаючи їхню структуру незмінною. Це досягається шляхом обгортання об'єкта в інший клас, який реалізує додаткову функціональність. Наприклад, у графічних редакторах базовий об'єкт зображення може бути обгорнутий декоратором, який додає ефект тіні, рамки чи іншого візуального елементу. Decorator особливо корисний, коли потрібно розширити функціональність, уникаючи створення великої кількості підкласів для кожного можливого поєднання функцій.

Хід роботи

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.

Тема: Mind-mapping software

Реалізація шаблону Memento

Для застосунку побудови карт пам’яті важливим елементом роботи є можливість збереження стану карти пам’яті та можливість відновити карту до цього збереженого стану або принаймні передивлятися історію карти пам’яті та бачити етапи її розвитку та доповнення. Шаблон проектування Memento дозволяє імплементувати такий функціонал зручним способом.

Першим елементом, який потрібен для імплементації шаблону є Memento. Це об'єкт, який інкапсулює збережений стан. Його головною задачею є збереження даних без розголошення деталей реалізації.

MindMapMemento.java

```java
package org.example.mindmappingsoftware.mementos;

public class MindMapMemento {  10 usages  yablonya
    private final String snapshot;  2 usages

    public MindMapMemento(String snapshot) {  3 usage
        this.snapshot = snapshot;
    }

    public String getSnapshot() {  2 usages  yablonya
        return snapshot;
    }
}

```

Рис. 1 - Клас MindMapMemento

На рис. 1 зображено клас MindMapMemento, що має поле snapshot. Це поле буде містити серіалізовану карту пам’яті з усіма її елементами у вигляді рядка, шоб карту можна було зручно зберігати в базі даних.

Наступним елементом є Originator. Це об'єкт, стан якого потрібно зберігати або відновлювати. Він створює екземпляр Memento, коли необхідно зберегти свій стан, і використовує Memento для відновлення попереднього стану.

FullMindMap.java

```java
package org.example.mindmappingsoftware.dto;

import ...

public class FullMindMap {  16 usages  yablonya *
    private MindMap mindMap;  3 usages
    private List<Node> nodes;  3 usages
    private LocalDateTime savedAt;  2 usages

    public MindMap getMindMap() { return mindMap; }

    public void setMindMap(MindMap mindMap) { this.mindMap = mindMap; }

    public List<Node> getNodes() { return nodes; }

    public void setNodes(List<Node> nodes) { this.nodes = nodes; }

    public LocalDateTime getSavedAt() { return savedAt; }

    public void setSavedAt(LocalDateTime savedAt) { this.savedAt = savedAt; }

    public MindMapMemento saveState() throws Exception {  1 usage  yablonya
        ObjectMapper mapper = new ObjectMapper();
        String snapshot = mapper.writeValueAsString(this);
        return new MindMapMemento(snapshot);
    }

    public void restoreState(MindMapMemento memento) throws Exception {  2 usages  yablonya *
        ObjectMapper mapper = new ObjectMapper();
        FullMindMap restored = mapper.readValue(memento.getSnapshot(), FullMindMap.class);
        this.mindMap = restored.getMindMap();
        this.nodes = restored.getNodes();
    }
}
```

Рис. 2 - Клас FullMindMap

На рис. 2 бачимо клас FullMindMap, що збирає в себе сам об’єкт карти та всі вузли пов’язані з нею. Клас також має методи saveState що повертає об’єкт Memento, тобто серіалізує всі зібрані в собі дані та на основі них формує об’єкт Memento, а також метод restoreState, шо приймає об’єкт Memento і з цього десеріалізує карту з її елементами

MindMapHistoryService.java

```java
@Service  4 usages
public class MindMapHistoryService {
    private final MindMapHistoryRepository mindMapHistoryRepository;  4 usages
    private final NodeRepository nodeRepository;  3 usages
    private final MindMapService mindMapService;  3 usages
    private static final Logger logger = LoggerFactory.getLogger(MindMapHistoryService.class);  11 usages

    @Autowired
    public MindMapHistoryService(
            MindMapHistoryRepository mindMapHistoryRepository,
            NodeRepository nodeRepository,
            MindMapService mindMapService
    ) {
        this.mindMapHistoryRepository = mindMapHistoryRepository;
        this.nodeRepository = nodeRepository;
        this.mindMapService = mindMapService;
    }

    public void saveMindMapState(Long mindMapId) {  1 usage
        try {
            FullMindMap fullMindMap = mindMapService.getFullMindMap(
                    mindMapService.getMindMap(mindMapId).getCreator(),
                    mindMapId
            );
            MindMapMemento memento = fullMindMap.saveState();

            MindMapHistory history = new MindMapHistory();
            history.setMindMap(fullMindMap.getMindMap());
            history.setSnapshot(memento.getSnapshot());
            history.setSavedAt(LocalDateTime.now());

            mindMapHistoryRepository.save(history);

            logger.info("Saved state for mind map with ID {}", mindMapId);
        } catch (NoSuchElementException e) {
            logger.warn("Failed to save state: Mind map with ID {} not found", mindMapId);
            throw e;
        } catch (Exception e) {
            logger.error("Error saving state for mind map with ID {}: {}", mindMapId, e.getMessage());
            throw new RuntimeException("Failed to save mind map state", e);
        }
    }
```

Рис. 4 - Початок класу MindMapHistoryService

```java
@Transactional 1 usage
public void restoreMindMapState(Long mindMapId, LocalDateTime restoreDate) {
    try {
        MindMapHistory snapshot = mindMapHistoryRepository
                .findByMindMapIdAndSavedAt(mindMapId, restoreDate);

        if (snapshot == null) {
            logger.warn("No saved state found for mind map with ID {} before {}", mindMapId, restoreDate);
            throw new NoSuchElementException("No saved state found for MindMap with ID: " + mindMapId + " before " + restoreDate);
        }

        MindMapMemento memento = new MindMapMemento(snapshot.getSnapshot());
        FullMindMap restoredMap = new FullMindMap();
        restoredMap.restoreState(memento);

        nodeRepository.deleteAllByMindMapId(mindMapId);

        nodeRepository.saveAll(restoredMap.getNodes());

        logger.info("Restored state for mind map with ID {} to snapshot at {}", mindMapId, restoreDate);
    } catch (NoSuchElementException e) {
        logger.warn("Failed to restore state: {}", e.getMessage());
        throw e;
    } catch (Exception e) {
        logger.error("Error restoring state for mind map with ID {}: {}", mindMapId, e.getMessage());
        throw new RuntimeException("Failed to restore mind map state", e);
    }
}
```

Рис. 5 - Метод restoreMindMapState класу MindMapHistoryService

```java
public List<FullMindMap> getMindMapHistory(Long mindMapId) { 1 usage
    try {
        List<MindMapHistory> history = mindMapHistoryRepository.findAllByMindMapIdOrderBySavedAtDesc(mindMapId);

        if (history.isEmpty()) {
            logger.warn("No history found for mind map with ID {}", mindMapId);
            return Collections.emptyList();
        }

        List<FullMindMap> responseHistory = history.stream() Stream<MindMapHistory>
                .map(entry -> {
                    try {
                        MindMapMemento memento = new MindMapMemento(entry.getSnapshot());
                        FullMindMap response = new FullMindMap();
                        response.restoreState(memento);
                        response.setSavedAt(entry.getSavedAt());

                        return response;
                    } catch (Exception e) {
                        logger.error("Error deserializing snapshot for mind map with ID {}: {}", mindMapId, e.getMessage());
                        throw new RuntimeException("Error deserializing snapshot", e);
                    }
                }) Stream<FullMindMap>
                .collect(Collectors.toList());

        logger.info("Retrieved history for mind map with ID {}", mindMapId);
        return responseHistory;
    } catch (Exception e) {
        logger.error("Error retrieving history for mind map with ID {}: {}", mindMapId, e.getMessage());
        throw new RuntimeException("Failed to retrieve mind map history", e);
    }
}
```

Рис. 6 - Метод getMindMapHistory класу MindMapHistoryService

На рис. 4-6 зображено клас MindMapHistoryService, що є останнім елементом шаблону - Caretaker’ом, що завідує станом карти пам’яті, зберігає, відновлює цей стан та надає всю історію карти.

Весь оновлений код можна переглянути в даній директорії репозиторію проєкту:
https://github.com/yablonya/TRPZ_labs_yablonskyi_ia-24/tree/lab-5/src/main/java/org/example/mindmappingsoftware

Висновок: Виконання цієї лабораторної роботи з імплементацією шаблону Memento допомогло мені краще зрозуміти, як зберігати стан об'єктів у складних системах, не порушуючи принципів інкапсуляції. У результаті я зміг використати цей шаблон для створення функціоналу управління історією в своєму проекті Mind-mapping software.

Особливо цікаво було розібратися в ролі кожного елемента — Originator, Caretaker і Memento. Кожен із них чітко виконує свою задачу, і це дозволяє зробити систему не тільки ефективною, але й зрозумілою для підтримки та розвитку.

Також робота змусила мене замислитися над тим, як обмежити доступ до важливих даних у Memento. Це показало важливість балансу між безпекою і зручністю використання. У реальних проєктах, де безпека даних є критичною, шаблон може вимагати додаткового шифрування чи інших засобів захисту.

У процесі я також переконався, наскільки важливо використовувати готові патерни проєктування. Вони допомагають уникнути "велосипедів" і спрощують роботу з кодом у довгостроковій перспективі. Загалом, робота не тільки розширила мої знання, а й додала впевненості у застосуванні шаблонів для розв'язання реальних задач.