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
Технології розроблення програмного забезпечення
ШАБЛОНИ «COMPOSITE», «FLYWEIGHT», «INTERPRETER», «VISITOR»
CI-server

Виконав:
Студент групи ІА-22
Вдовиченко А.Х.

Перевірив:
Мягкий М. Ю

Київ 2024

Зміст

Тема: шаблони «COMPOSITE», «FLYWEIGHT», «INTERPRETER», «VISITOR»
Мета: ознайомитись з теоретичними відомостями, розробити частину функціоналу системи з використанням одного із шаблонів проектування.

Теоретичні відомості

Розробка програмного забезпечення для масштабних корпоративних додатків потребує використання спеціальних шаблонів роботи з базами даних. Одним із таких є шаблон "Active Record", який об'єднує дані та поведінку в одному об'єкті. Цей підхід передбачає, що кожен об'єкт виступає "обгорткою" для рядка бази даних, а логіка доступу до даних інтегрована в сам об'єкт. Хоча цей шаблон є простим і популярним, особливо в Ruby on Rails, зі збільшенням складності системи логіку запитів часто переносять в окремі об'єкти.

Інший підхід – шаблон "Table Data Gateway". У ньому взаємодія з базою даних виконується через окремі класи для кожного типу даних. Цей підхід забезпечує гнучкість і тестованість, розділяючи логіку запитів і збереження даних. Однак повторення коду для шлюзів часто вимагає абстракції через базові класи. Шаблон також відомий під назвою "репозиторій".

Шаблон "Data Mapping" зосереджується на проблемі перетворення об’єктів даних у формат реляційної бази. Відображення дозволяє гармонізувати типи даних і забезпечує взаємодію об’єктів з джерелом даних. Наприклад, маппер може відповідати за зіставлення властивостей об’єкта з колонками таблиці.

Шаблон "Composite" допомагає представляти об’єкти у вигляді дерева, що спрощує роботу з ієрархіями. Наприклад, форма в інтерфейсі може містити різні дочірні елементи, які можна обробляти як однорідні об’єкти. Це дозволяє легко масштабувати і додавати нові типи компонентів, хоча загальний дизайн класів може стати занадто складним.

Шаблон "Flyweight" вирішує проблему надмірного використання пам’яті за рахунок поділу об’єктів між різними частинами програми. Наприклад, у грі можна створити один об’єкт для кольору чи текстури, а зовнішні характеристики, як координати чи швидкість, винести в контекст. Це економить пам'ять, хоча може ускладнити код і збільшити витрати на обчислення.

Шаблон "Interpreter" забезпечує створення граматики та інтерпретатора для обраної мови, використовуючи деревоподібні структури. Це спрощує додавання нових правил і змін у граматиці, хоча для складних систем кількість класів може зрости до надмірної.

Шаблон "Visitor" дозволяє визначати операції для об’єктів без зміни їхньої структури. Це зручно для додавання нових операцій, наприклад, для експорту графів у XML. Проте кожен новий елемент ієрархії вимагає модифікації всіх існуючих відвідувачів, що може ускладнити підтримку.

Хід роботи

Для реалізації запуску групи контейнерів був обраний патерн COMPOSITE. Його основне призначення — робота з ієрархічними структурами, які можуть складатися як з окремих елементів, так і з груп. Цей підхід дозволяє обробляти окремі елементи та їхні групи однаково, забезпечуючи гнучкість і простоту управління складними структурами. Проблема, яку вирішує цей патерн, полягає в необхідності управління як окремими контейнерами, так і їхніми групами через спільний інтерфейс.

Діаграма класів реалізованого патерну COMPOSITE зображена на рисунку 1.1.

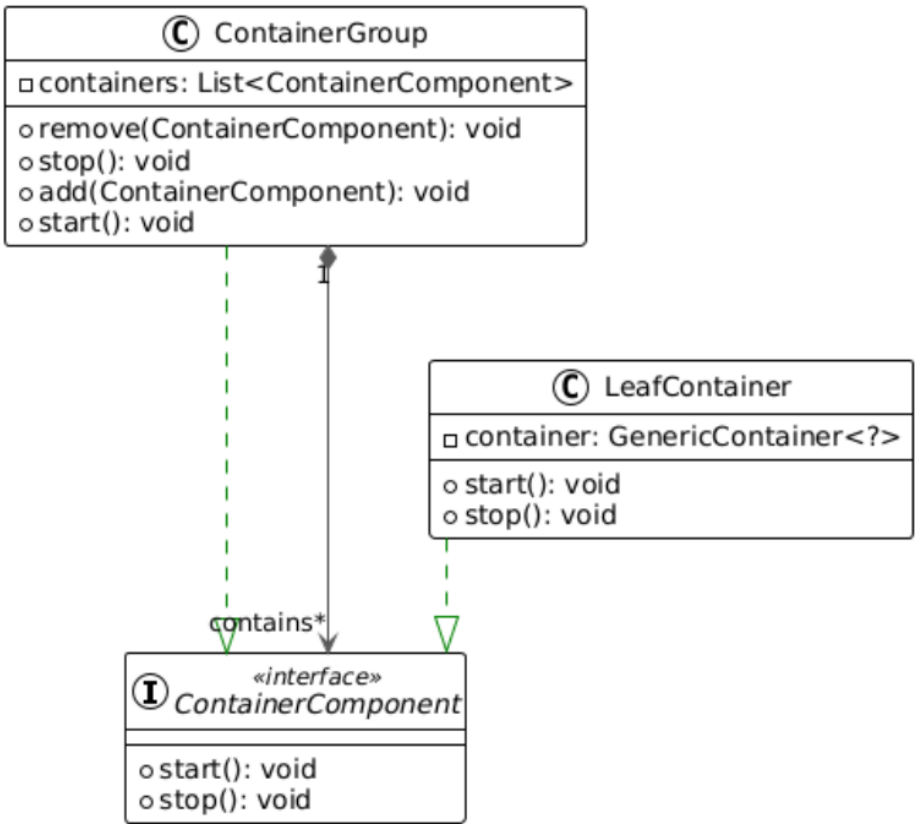

Рис 1.1. – Реалізація патерну Composite для контейнерів

У цій реалізації патерн COMPOSITE використовується для управління TestContainers. Було створено інтерфейс ContainerComponent, який забезпечує уніфікований набір методів для запуску та зупинки контейнерів. Окремі контейнери представлені класом LeafContainer, тоді як групи контейнерів представлені класом ContainerGroup. Групи можуть містити як окремі контейнери, так і інші групи, що дозволяє будувати складні ієрархії. (Рис 1.2.)

```java
public class ContainerGroup implements ContainerComponent {  3 usages
    private final List<ContainerComponent> containers = new ArrayList<>();  4 usages

    public void add(ContainerComponent component) {
        containers.add(component);
    }

    public void remove(ContainerComponent component) {  no usages
        containers.remove(component);
    }

    @Override  2 usages
    public void start() {
        for (ContainerComponent container : containers) {
            container.start();
        }
    }

    @Override  2 usages
    public void stop() {
        for (ContainerComponent container : containers) {
            container.stop();
        }
    }
}
```

Рис 1.2. – Реалізація класу ContainerGroup

Результатом роботи стала система, яка дозволяє керувати як окремими контейнерами, так і їх групами однаково просто. Завдяки цьому підходу зменшилася складність коду, спростилося масштабування та тестування. Нові контейнери або групи можна легко додавати без значних змін до існуючої архітектури. Це робить систему більш гнучкою і зручною для використання в проектах, де потрібно працювати з великою кількістю контейнерів.

Висновки

У цій лабораторній роботі було спрощено роботу як з окремими контейнерами так і з групами контейнерів за допомогою патерну Composite.

Вихідний код

```
package org.example.service.facade;

public interface ContainerComponent {
    void start();
    void stop();
}

package org.example.service.facade;

import java.util.ArrayList;
import java.util.List;

public class ContainerGroup implements ContainerComponent {
    private final List<ContainerComponent> containers = new ArrayList<>();

    public void add(ContainerComponent component) {
        containers.add(component);
    }

    public void remove(ContainerComponent component) {
        containers.remove(component);
    }

    @Override
    public void start() {
        for (ContainerComponent container : containers) {
            container.start();
        }
    }

    @Override
    public void stop() {
        for (ContainerComponent container : containers) {
            container.stop();
        }
    }
}


package org.example.service.facade;

import org.testcontainers.containers.GenericContainer;

public class LeafContainer implements ContainerComponent {
    private final GenericContainer<?> container;

    public LeafContainer(GenericContainer<?> container) {
        this.container = container;
    }

    @Override
    public void start() {
        container.start();
    }

    @Override
    public void stop() {
        container.stop();
    }
}
```